

O'REILLY®

командная строка

# Linux

лучшие  
практики



Дэниел Джей Барретт

# Linux

# Командная строка

Лучшие практики

Дэниел Джей Барретт



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.2  
УДК 004.451  
Б25

### **Барретт Дэниел Джей**

**Б25** Linux. Командная строка. Лучшие практики. — СПб.: Питер, 2023. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2300-1

Перейдите на новый уровень работы в Linux! Если вы системный администратор, разработчик программного обеспечения, SRE-инженер или пользователь Linux, книга поможет вам работать быстрее, элегантнее и эффективнее. Вы научитесь создавать и запускать сложные команды, которые решают реальные бизнес-задачи, обрабатывать и извлекать информацию, а также автоматизировать ручную работу.

Узнайте, что происходит внутри командной оболочки Linux. Вне зависимости от используемых команд вы повысите эффективность работы в Linux и станете более конкурентоспособным специалистом.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2  
УДК 004.451

Права на издание получены по соглашению с O'Reilly.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1098113407 англ.

Authorized Russian translation of the English edition of Efficient Linux at the Command Line, ISBN 9781098113407 © 2022 Daniel Barrett.  
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-2300-1

© Перевод на русский язык ООО «Прогресс книга», 2022  
© Издание на русском языке, оформление ООО «Прогресс книга», 2023  
© Серия «Бестселлеры O'Reilly», 2023

# Оглавление

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

<b>Предисловие .....</b>	<b>10</b>
Чему вы научитесь .....	11
Чем эта книга не является .....	11
Для кого эта книга .....	12
Ваша командная оболочка .....	13
Условные обозначения .....	13
Использование исходного кода примеров .....	14
<b>Благодарности .....</b>	<b>15</b>
<b>От издательства .....</b>	<b>16</b>

## ЧАСТЬ 1 ОСНОВНЫЕ ПОНЯТИЯ

<b>Глава 1. Объединение команд .....</b>	<b>18</b>
Ввод, вывод и каналы .....	19
Шесть команд для начала .....	21
Команда #1: wc .....	21
Команда #2: head .....	24
Команда #3: cut .....	24
Команда #4: grep .....	26
Команда #5: sort .....	28
Команда #6: uniq .....	30
Обнаружение дубликатов файлов .....	32
Резюме .....	34
<b>Глава 2. Знакомство с командной оболочкой .....</b>	<b>35</b>
Терминология командной оболочки .....	36
Сопоставление шаблонов имен файлов .....	37
Вычисление переменных .....	40
Откуда берутся значения переменных .....	40
Переменные и заблуждения .....	41
Шаблоны vs переменные .....	42
Сокращение команд с помощью псевдонимов .....	43
Перенаправление ввода и вывода .....	44
Отключение вычисления с помощью кавычек и экранирования .....	47

Расположение исполняемых программ .....	49
Окружение и файлы инициализации, краткая версия.....	50
Резюме.....	52
<b>Глава 3. Повторный запуск команд .....</b>	<b>53</b>
Просмотр истории команд .....	54
Повторный вызов команд из истории .....	55
Перемещение курсора по истории команд .....	55
Расширение истории команд.....	57
Забудьте об ошибочном удалении файлов (спасибо расширению истории) .....	60
Инкрементальный поиск по истории команд.....	62
Редактирование командной строки.....	64
Перемещение курсора внутри команды .....	65
Расширение истории с помощью знака вставки .....	65
Редактирование командной строки в стилях Emacs или Vim .....	67
Резюме.....	69
<b>Глава 4. Перемещение по файловой системе .....</b>	<b>70</b>
Лучшие способы перехода в нужный каталог .....	71
Переход в домашний каталог .....	71
Перемещайтесь быстрее с автозавершением командной строки .....	72
Переход к часто посещаемым каталогам с использованием псевдонимов или переменных .....	73
Уменьшите пространство поиска с помощью CDPATH .....	75
Организуйте свой домашний каталог для быстрой навигации .....	77
Лучшие способы вернуться в каталог .....	79
Переключение между двумя каталогами с помощью «cd -» .....	79
Переключение между несколькими каталогами с помощью pushd и popd.....	80
Резюме.....	86
 <b>ЧАСТЬ 2</b> <b>ПРОДВИНУТЫЕ НАВЫКИ</b>	
<b>Глава 5. Расширяем ваш инструментарий.....</b>	<b>88</b>
Создание текста.....	89
Команда date.....	89
Команда seq.....	90
Расширение команд с помощью фигурных скобок.....	91
Команда find.....	93

Команда <code>yes</code> .....	94
Извлечение текста .....	95
Команда <code>grep</code> . Более глубокий взгляд.....	96
Команда <code>tail</code> .....	100
Команда <code>awk {print}</code> .....	101
Объединение текста .....	103
Команда <code>tac</code> .....	104
Команда <code>paste</code> .....	104
Команда <code>diff</code> .....	105
Преобразование текста .....	106
Команда <code>tr</code> .....	107
Команда <code>rev</code> .....	107
Команды <code>awk</code> и <code>sed</code> .....	108
Как расширить инструментарий .....	117
Резюме .....	119
<b>Глава 6. Родители, потомки и окружение .....</b>	<b>120</b>
Оболочки — это исполняемые файлы .....	121
Родительский и дочерний процессы .....	123
Переменные окружения .....	125
Создание переменных окружения .....	126
Предупреждение о мифе: «глобальные» переменные .....	127
Дочерние оболочки vs подоболочки.....	128
Настройка окружения .....	129
Повторное считывание файла конфигурации .....	132
Путешествие с вашим окружением.....	132
Резюме .....	133
<b>Глава 7. Еще 11 способов запуска команды .....</b>	<b>134</b>
Способы, использующие списки .....	134
Способ #1. Условные списки.....	135
Способ #2. Безусловные списки .....	137
Способы, использующие подстановку .....	137
Способ #3. Подстановка команд.....	138
Способ #4. Подстановка процесса.....	140
Команда как строка .....	143
Способ #5. Передача команды в <code>bash</code> в качестве аргумента.....	144
Способ #6. Передача команды в <code>bash</code> через стандартный ввод.....	145
Способ #7. Удаленное выполнение однострочника с помощью <code>ssh</code> .....	147
Способ #8. Запуск списка команд с помощью <code>xargs</code> .....	148

Способы, использующие управление процессами .....	153
Способ #9. Фоновое выполнение команды.....	153
Способ #10. Явные подболочки.....	159
Способ #11. Замена процесса .....	161
Резюме.....	162
<b>Глава 8. Создание дерзких однострочников.....</b>	<b>164</b>
Приготовьтесь быть дерзкими .....	166
Будьте гибкими .....	166
Подумайте, с чего начать .....	167
Изучите инструменты тестирования .....	169
Вставка имени файла в последовательность .....	169
Проверка совпадающих пар файлов.....	172
Создание CDPATH из вашего домашнего каталога .....	175
Создание тестовых файлов.....	177
Создание пустых файлов .....	180
Резюме.....	181
<b>Глава 9. Использование текстовых файлов .....</b>	<b>182</b>
Первый пример: поиск файлов.....	184
Проверка срока действия домена.....	185
Создание базы данных телефонных кодов.....	188
Создание менеджера паролей.....	190
Резюме .....	197

## ЧАСТЬ 3

### ДОПОЛНИТЕЛЬНЫЕ ПЛЮСЫ

<b>Глава 10. Эффективное использование клавиатуры .....</b>	<b>200</b>
Работа с окнами.....	200
Мгновенный запуск оболочек и браузера.....	201
Одноразовые окна .....	202
Горячие клавиши в браузере .....	202
Переключение окон и рабочих столов .....	203
Доступ в интернет из командной строки .....	204
Запуск окон браузера из командной строки .....	204
Получение HTML-страниц с помощью curl и wget .....	207
Обработка кода HTML с помощью HTML-XML-utils .....	208
Получение и отображение содержимого веб-сайтов с помощью текстового браузера.....	212

Управление буфером обмена из командной строки .....	213
Подключение буферов обмена к stdin и stdout.....	214
Улучшение работы менеджера паролей .....	216
Резюме.....	219
<b>Глава 11. Финальные советы по экономии времени .....</b>	<b>220</b>
Способы решения задач легко и быстро.....	220
Переход в текстовый редактор напрямую из команды less .....	220
Редактирование файлов, содержащих заданную строку .....	221
Смиритесь с опечатками .....	222
Быстрое создание пустых файлов .....	222
Обработка файла построчно.....	223
Список команд, поддерживающих рекурсию .....	223
Читайте справочные страницы.....	224
Способы решения задач, требующие затрат времени на изучение .....	224
Прочтите справочную страницу команды bash .....	224
Изучите команды cron, crontab и at.....	225
Изучите команду rsync .....	226
Изучите другой язык для написания сценариев.....	227
Используйте make для задач, не связанных с программированием.....	228
Применяйте контроль версий к повседневным файлам .....	230
Прощание.....	231
<b>Приложение А. Памятка по Linux.....</b>	<b>233</b>
Команды, аргументы и параметры.....	233
Файловая система, каталоги и пути .....	234
Перемещение по каталогам .....	236
Создание и редактирование файлов .....	236
Работа с файлами и каталогами .....	237
Просмотр файлов.....	239
Права доступа к файлам.....	239
Процессы.....	240
Просмотр документации .....	241
Сценарии оболочки .....	242
Получение привилегий суперпользователя .....	243
Дополнительная литература.....	244
<b>Приложение Б. Если вы используете не bash.....</b>	<b>245</b>
<b>Об авторе .....</b>	<b>251</b>
<b>Иллюстрация на обложке.....</b>	<b>252</b>



# Предисловие

Эта книга позволит перейти на новый уровень использования командной строки Linux, чтобы вы могли работать быстрее, умнее и эффективнее.

Если вы похожи на большинство пользователей Linux, то приобрели первоначальные навыки использования командной строки на работе, или прочитав вводную книгу, или установив Linux дома и просто попробовав. Моя книга должна помочь вам сделать следующий шаг — развить навыки работы с командной строкой Linux от среднего до продвинутого уровня. Она наполнена методами и концепциями, которые, я надеюсь, изменят ваше взаимодействие с Linux и повысят производительность. Воспринимайте ее как вторую книгу по использованию Linux, которая выведет вас за рамки азов.

Командная строка — самый простой и вместе с тем самый сложный из интерфейсов. Он прост, потому что содержит только приглашение командной строки, которое ждет ваших действий<sup>1</sup>:

```
$
```

Он сложный, потому что за все, что выходит за рамки приглашения, отвечаете только вы. Нет понятных значков, кнопок или меню, которые могли бы помочь. Вместо этого каждая введенная команда является результатом ваших усилий. Это верно как для основных команд, таких, например, как вывод списка файлов:

```
$ ls
```

так и для более сложных, например:

```
$ paste <(echo {1..10}.jpg | sed 's/ /\n/g') \  
        <(echo {0..9}.jpg | sed 's/ /\n/g') \  
      | sed 's/^/mv /' \  
      | bash
```

Если вы смотрите на предыдущую команду и думаете: «*Что это*, черт возьми?» или «Мне бы никогда не понадобилась такая сложная команда», то эта книга для вас<sup>2</sup>.

---

<sup>1</sup> Для отображения приглашения командной строки в этой книге используется символ \$. В вашей операционной системе этот символ может быть другим.

<sup>2</sup> Вы узнаете назначение этой таинственной команды в главе 8.

## Чему вы научитесь

Эта книга поможет вам быстро и эффективно освоить три основных навыка:

- Выбор и создание команд для решения текущей задачи.
- Эффективный запуск команд.
- Удобная навигация по файловой системе Linux.

Прочитав книгу, вы будете понимать, что остается за кадром при запуске команды, и научитесь лучше предсказывать ее результаты (а не руководствоваться суевериями). Вы познакомитесь с дюжиной разных способов запуска команд и узнаете, когда лучше использовать каждый из них. Вы также узнаете приемы и хитрости, которые сделают вашу работу продуктивнее. Например:

- Построение сложных команд из простых, шаг за шагом, для решения практических задач, например управления паролями или создания тысяч тестовых файлов.
- Экономия времени за счет продуманной организации домашнего каталога, чтобы вам не приходилось искать нужные файлы.
- Преобразование текстовых файлов и получение из них данных для решения поставленных задач.
- Управление из командной строки теми операциями в Linux, которые обычно выполняются мышкой. Например, копирование и вставка с помощью буфера обмена или получение и обработка веб-данных. И все это, не отрывая рук от клавиатуры.

И самое главное, вы изучите лучшие практические методы работы в командной строке, поэтому независимо от того, какие команды вы запускаете, вы сможете добиться большего в повседневном использовании Linux и стать более конкурентоспособными на рынке труда. Я бы хотел иметь эту книгу, когда сам только начинал изучение Linux.

## Чем эта книга не является

Эта книга не поможет оптимизировать и не сделает эффективнее ваш компьютер с установленной операционной системой Linux. Она лишь сделает *вас* более эффективными при использовании Linux.

Эта книга также не является исчерпывающим справочником по командной строке. Существуют сотни команд и функций, которые здесь не упомянуты. Эта книга об опыте. Она содержит тщательно отобранный набор практических знаний о командной строке для развития ваших навыков. В качестве справочного руководства используйте мою предыдущую книгу *Linux Pocket Guide* (издательство O'Reilly).

## Для кого эта книга

Предполагается, что у вас есть опыт работы с Linux, поскольку *эта книга не знакомит с Linux*. Она предназначена, прежде всего, для пользователей, которые хотят улучшить свои навыки работы с командной строкой, таких как студенты, системные администраторы, разработчики программного обеспечения, IT-инженеры, тестировщики и энтузиасты операционной системы Linux. Опытные пользователи Linux также смогут найти для себя полезный материал, особенно если они учились методом проб и ошибок и хотят улучшить свое понимание командной строки.

Чтобы извлечь максимальную пользу из этой книги, вы уже должны быть знакомы со следующими темами (если нет, см. приложение А, где дается их краткий обзор):

- Создание и редактирование текстовых файлов с помощью редакторов `vim` (`vi`), `emacs`, `nano`, `pico`.
- Основные команды для работы с файлами, такие как `cp` (копирование), `mv` (перемещение или переименование), `rm` (удаление), `chmod` (изменение прав доступа).
- Основные команды для просмотра файлов, такие как `cat` (вывод содержимого файла) и `less` (постраничный вывод текста).
- Основные команды для работы с каталогами, такие как `cd` (переход в другой каталог), `ls` (вывод списка файлов и каталогов), `mkdir` (создание каталога), `rmdir` (удаление каталога), `pwd` (вывод пути к текущему каталогу).
- Основы сценариев командной оболочки: хранение команд Linux в файле, создание исполняемого файла (с помощью `chmod 755` или `chmod +x`) и его запуск.
- Просмотр встроенной документации Linux, известной как ман-страницы, с помощью команды `man` (например, `man cat` отображает документацию по команде `cat`).
- Умение стать суперпользователем с помощью команды `sudo` для полного доступа к операционной системе (например, команда `sudo nano /etc/hosts` позволяет редактировать файл `/etc/hosts`, изменение которого запрещено обычному пользователю).

Если вы также знакомы с общими функциями командной строки, такими как сопоставление имен файлов с шаблоном (символы `*` и `?`), перенаправление ввода/вывода (`<` и `>`) и каналы (`|`), то вы уже хорошо подготовлены к изучению этой книги.

## Ваша командная оболочка

Предполагается, что вашей командной оболочкой является `bash`, которая используется по умолчанию в большинстве дистрибутивов Linux. Под термином «оболочка» в тексте имеется в виду именно `bash`. Большинство идей, представленных в книге, применимы и к другим оболочкам, таким как `zsh` или `dash` (см. Приложение Б для перевода примеров в другие оболочки). Большая часть материала будет работать без изменений и в Apple Mac-терминале, который по умолчанию запускает `zsh`, но также может запускать и `bash`<sup>1</sup>.

## Условные обозначения

В этой книге используются следующие условные обозначения.

### *Курсив*

Курсивом выделены новые термины и важные понятия, а также имена файлов и каталогов.

### **Моноширинный шрифт**

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

### **Моноширинный полужирный шрифт**

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

### ***Моноширинный курсив***

Показывает текст, который должен быть заменен значениями, введенными пользователем или определяемыми контекстом. Кроме того, этот шрифт используется для кратких пояснений в правой части листингов.

### **Моноширинный шрифт в рамке**

Используется для привлечения внимания к отдельным частям листингов.

### **Шрифт без засечек**

Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса.

<sup>1</sup> Версия `bash` для macOS устарела, в ней отсутствуют важные функции. Чтобы обновить `bash`, см. статью Дэниела Вейбеля (Daniel Weibel) *Upgrading Bash on macOS* (<https://oreil.ly/35jux>).



Этот рисунок указывает на совет или предложение.



Этот рисунок указывает на примечание.



Этот рисунок указывает на предупреждение.

## Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: <https://efficientlinux.com/examples>. Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За разрешением на использование значительных объемов программного кода из книги обращайтесь по адресу [permissions@oreilly.com](mailto:permissions@oreilly.com).

# Благодарности

Эту книгу было приятно писать. Спасибо замечательным людям из издательства O'Reilly, особенно редакторам Вирджинии Уилсон (Virginia Wilson) и Джону Девинсу (John Devins), редакторам производства Кейтлин Геган (Caitlin Ghegan) и Грегори Хайману (Gregory Human), контент-менеджеру Кристен Браун (Kristen Brown), редактору Киму Уимпсетту (Kim Wimpsett), редактору предметного указателя Сюю Клефстад (Sue Klefstad) и всегда готовой помочь команде производственного отдела. Я также очень благодарен рецензентам книги Полу Байеру (Paul Bayer), Джону Бонезио (John Bonesio), Дэну Риттеру (Dan Ritter) и Карле Шредер (Carla Schroder) за ценные комментарии и критические замечания. Также спасибо Boston Linux Users Group за предложения по названию. Особая благодарность Мэгги Джонсон (Maggie Johnson) из Google за ее любезное разрешение написать книгу.

Я бы хотел выразить глубочайшую благодарность Чипу Эндрюсу (Chip Andrews), Мэтью Диазу (Matthew Diaz) и Роберту Странду (Robert Strandh), которые 35 лет назад учились со мной в Университете Джонса Хопкинса (The Johns Hopkins University). Они заметили мой растущий интерес к операционной системе Unix и, к моему крайнему удивлению, порекомендовали Департаменту компьютерных наук (Computer Science Department) нанять меня в качестве системного администратора. Их вера в меня изменила траекторию моей жизни (кстати, благодаря Роберту добавлен совет касательно навыков слепой печати в главе 3). Спасибо также тем, кто создает и поддерживает проекты Linux, GNU Emacs, Git, AsciiDoc и многие другие инструменты с открытым исходным кодом — без этих умных и щедрых людей моя карьера действительно была бы совсем другой.

Как всегда, спасибо моей замечательной семье, Лизе и Софии, за их любовь и терпение.

# От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу [comp@piter.com](mailto:comp@piter.com) (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

# ЧАСТЬ 1

---

## Основные понятия

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

Первые четыре главы должны быстро повысить эффективность вашей работы в командной строке, познакомив вас с принципами и методами, которые можно сразу применить на практике. Вы научитесь комбинировать команды с помощью каналов, поймете функции командной оболочки Linux, сможете повторно вызывать и редактировать команды, которые ранее уже запускали, и быстро перемещаться по файловой системе Linux.



## ГЛАВА 1

---

# Объединение команд

В Windows, macOS и большинстве других операционных систем вы тратите время на запуск таких приложений, как веб-браузеры, текстовые редакторы, программы для работы с таблицами и игры. Типичное приложение содержит множество функций: все, что, по мнению разработчиков, может понадобиться пользователям. Поэтому большинство приложений самодостаточно и не связаны с другими приложениями. Вы можете время от времени копировать и вставлять данные из одного приложения в другое, но по большей части они независимы друг от друга.

Командная строка в Linux работает по-другому. Вместо больших приложений с множеством функций Linux предоставляет тысячи небольших команд с ограниченным набором возможностей. Команда `cat`, например, выводит на экран содержимое файлов и больше ничего. `ls` отображает список файлов в каталоге, `mv` переименовывает файлы и т. д. Каждая команда имеет простую, четко определенную цель.

Что делать, если вам нужно произвести более сложные действия? Не волнуйтесь, Linux позволяет легко *объединять команды*, чтобы их функции работали вместе для достижения поставленной цели. Такой способ порождает совсем другое отношение к работе за компьютером. Вместо того чтобы спрашивать себя «Какое приложение мне запустить?» для достижения необходимого результата, возникает вопрос: «Какие команды мне объединить?»

В этой главе вы узнаете, как упорядочивать и запускать команды в различных комбинациях, чтобы решать поставленные задачи. Для простоты начнем с шести команд Linux и их основных способов применения. Это позволит сосредоточиться на более сложной и интересной части — их объединении — без долгого обучения. Это все равно, что учиться готовить, используя только шесть ингредиентов, или осваивать столярное дело с применением только молотка и пилы (дополнительные команды добавятся в ваш набор в главе 5).

Вы будете объединять команды, используя *канал (pipe)* — функцию Linux, которая соединяет вывод одной команды с вводом другой. При представлении каждой команды (`wc`, `head`, `cut`, `grep`, `sort` и `uniq`) мы сразу же будем рассматривать пример использования канала. Некоторые примеры будут полезны

для повседневной работы в Linux, в то время как другие являются искусственно выдуманными для демонстрации важной функции.

## Ввод, вывод и каналы

Большинство команд Linux считают ввод с клавиатуры, отображают вывод на экране или делают и то, и другое. В Linux эти операции чтения и записи имеют собственные названия:

*stdin* (стандартный ввод, или стандартный входной поток)

Поток данных, который Linux считывает при вводе с клавиатуры. Когда вы вводите любую команду в терминале после приглашения командной строки, вы передаете данные в стандартный входной поток *stdin*.

*stdout* (стандартный вывод, или стандартный выходной поток)

Поток данных, который Linux выводит на ваш дисплей. Когда вы запускаете команду `ls` для печати имен файлов, результаты появляются в стандартном выходном потоке *stdout*.

Теперь самое интересное. Вы можете подключить стандартный вывод одной команды к стандартному вводу другой, чтобы первая команда передавала свои выходные данные во вторую. Давайте начнем со знакомой команды `ls -l` для просмотра большого каталога, такого как `/bin`, в длинном формате:

```
$ ls -l /bin
total 12104
-rwxr-xr-x 1 root root 1113504 Jun  6  2019 bash
-rwxr-xr-x 1 root root 170456 Sep 21  2019 BSD-csh
-rwxr-xr-x 1 root root 34888 Jul  4  2019 bunzip2
-rwxr-xr-x 1 root root 2062296 Sep 18  2020 busybox
-rwxr-xr-x 1 root root 34888 Jul  4  2019 bzip2
:
-rwxr-xr-x 1 root root 5047 Apr 27  2017 znew
```

Этот каталог содержит значительно больше файлов, чем, строк на вашем дисплее, поэтому вывод быстро прокручивается за пределы экрана. Жаль, что `ls` не может печатать информацию по одному экрану за раз, делая паузу, пока вы не нажмете клавишу для продолжения. Но постойте: у другой команды Linux есть такая возможность. Команда `less` отображает файл по одному экрану за раз:

```
$ less myfile
```

Вы можете соединить эти две команды, потому что `ls` передает данные в стандартный вывод, а `less` может считывать данные из стандартного ввода. Используем канал для отправки выходных данных `ls` на вход команды `less`:

```
$ ls -l /bin | less
```

### ЧТО ТАКОЕ КОМАНДА?

Слово *команда* имеет три различных значения в Linux, представленные на рис. 1.1:

#### Программа

Исполняемая программа, названная и выполняемая одним словом, например `ls`, или аналогичная функция, встроенная в оболочку, например `cd` (*встроенная команда оболочки*)<sup>1</sup>.

#### Простая команда

Имя программы или встроенной команды оболочки, за которым могут следовать аргументы, например `ls -l /bin`.

#### Комбинированная команда

Несколько простых команд, рассматриваемых как единое целое, например конвейер `ls -l /bin | less`



**Рис. 1.1.** Программы, простые команды и комбинированные команды называются «командами»

В этой книге слово «команда» используется во всех этих смыслах. Обычно из контекста понятно, что имеется в виду, но если нет, будем использовать один из более конкретных терминов.

Эта объединенная команда отображает содержимое каталога по одному экрану за раз. Вертикальная черта (|) между командами — это символ<sup>2</sup> канала в Linux. Он соединяет стандартный вывод первой команды со стандартным вводом следующей. Командная строка, содержащая каналы, называется *конвейером* (*pipeline*).

<sup>1</sup> Стандарт POSIX называет такие команды *утилитами*.

<sup>2</sup> На стандартной клавиатуре символ вертикальной черты находится на той же клавише, что и обратная косая черта (\), обычно расположенная между клавишами **Enter** и **Backspace** или между левой клавишей **Shift** и **Z**.

Команды обычно не знают, что они являются частью конвейера. `ls` считает, что выводит данные на дисплей, хотя на самом деле ее вывод был перенаправлен на `less`. А `less` верит, что читает данные с клавиатуры, когда на самом деле получает вывод `ls`.

## Шесть команд для начала

Использование каналов — неотъемлемая часть работы с Linux. Давайте развивать ваши навыки работы с каналами с помощью небольшого набора команд Linux. Когда вы столкнетесь с какими-то из этих команд в будущем, то будете готовы их объединять.

Команды `wc`, `head`, `cut`, `grep`, `sort` и `uniq` имеют множество опций и режимов работы, про которые пока умолчим, чтобы сосредоточиться на каналах. Если хотите узнать больше о конкретной команде, запустите команду `man`, которая отобразит полную документацию. Например:

```
$ man wc
```

Для демонстрации этих шести команд в действии будем использовать файл с именем *animals.txt*, в котором перечислены некоторые сведения из книг издательства O'Reilly. Содержимое файла представлено в примере 1.1.

### Пример 1.1. Содержимое файла animals.txt

python	Programming Python	2010	Lutz, Mark
snail	SSH, The Secure Shell	2005	Barrett, Daniel
alpaca	Intermediate Perl	2012	Schwartz, Randal
robin	MySQL High Availability	2014	Bell, Charles
horse	Linux in a Nutshell	2009	Siever, Ellen
donkey	Cisco IOS in a Nutshell	2005	Boney, James
oryx	Writing Word Macros	1999	Roman, Steven

Каждая строка содержит четыре факта о какой-либо книге издательства O'Reilly, разделенные одним символом табуляции: животное на обложке, название книги, год публикации и имя первого автора.

## Команда #1: `wc`

Команда `wc` выводит количество строк, слов и символов в файле:

```
$ wc animals.txt
 7 51 325 animals.txt
```

`wc` сообщает, что в файле *animals.txt* 7 строк, 51 слово и 325 символов. Если вы посчитаете символы, включая пробелы и табуляции, то найдете только 318 символов, но `wc` также учитывает скрытые символы новой строки.

Опции `-l`, `-w` и `-c` указывают `wc` печатать только количество строк, слов и символов соответственно:

```
$ wc -l animals.txt
7 animals.txt
$ wc -w animals.txt
51 animals.txt
$ wc -c animals.txt
325 animals.txt
```

Вывод параметров текста — настолько полезная и часто встречающаяся задача, что авторы `wc` разработали свою команду специально для работы с каналами. Если вы не указываете имя файла, она читает данные из стандартного ввода и отправляет данные в стандартный вывод. Давайте воспользуемся `ls` для вывода содержимого текущего каталога и используем `wc` для подсчета строк. Этот конвейер отвечает на вопрос «Сколько файлов в моем текущем каталоге?»:

```
$ ls -l
animals.txt
myfile
myfile2
test.py
$ ls -l | wc -l
4
```

Опция `-l`, указывающая `ls` выводить результаты в один столбец, здесь не обязательна. Чтобы узнать, почему мы ее использовали, см. врезку «`ls` меняет свое поведение при перенаправлении вывода» на с. 23.

`wc` — это первая команда, с которой мы познакомились в этой главе, поэтому пока сложно использовать каналы. Ради интереса направим вывод `wc` самой себе, чтобы продемонстрировать, что одна и та же команда может появляться в конвейере более одного раза. Эта комбинированная команда сообщает, что количество слов в выводе `wc` равно четырем (три целых числа и имя файла):

```
$ wc animals.txt
 7 51 325 animals.txt
$ wc animals.txt | wc -w
4
```

Не будем останавливаться. Добавим третий вызов `wc` в конвейер и посчитаем количество строк, слов и символов в выводе 4:

```
$ wc animals.txt | wc -w | wc
 1      1      2
```

### LS МЕНЯЕТ СВОЕ ПОВЕДЕНИЕ ПРИ ПЕРЕНАПРАВЛЕНИИ ВЫВОДА

В отличие от большинства других команд Linux, `ls` знает, выводит ли она данные на экран или перенаправляет в канал. Когда `stdout` — это экран, `ls` упорядочивает вывод в несколько колонок для удобства чтения:

```
$ ls /bin
bash      dir      kmod      networkctl  red      tar
bsd-csh   dmesg    less      nisdomainname  rm      tempfile
:
```

При перенаправлении `stdout` `ls` создает одну колонку. Убедимся в этом, передав вывод `ls` команде, которая воспроизводит свой ввод, такой как `cat`<sup>1</sup>:

```
$ ls /bin | cat
bash
bsd-csh
bunzip2
busybox
:
```

Это может привести к неожиданным результатам, как в следующем примере:

```
$ ls
animals.txt myfile myfile2 test.py
$ ls | wc -l
4
```

Первая команда `ls` выводит все имена файлов в одну строку, а вторая сообщает, что выводятся четыре строки. Если вы не знаете об особых свойствах `ls`, это несоответствие может удивить.

`ls` позволяет переопределить свое поведение по умолчанию. Вы можете заставить `ls` вывести на экран одну колонку с помощью параметра `-1` или несколько колонок с параметром `-C`.

Вывод указывает на одну строку (содержащую число 4), одно слово (само число 4) и два символа. Почему два? Потому что строка 4 заканчивается скрытым символом новой строки.

Думаю, что достаточно простейших конвейеров с `wc`. По мере того как мы изучим больше команд, конвейеры станут более практичными.

<sup>1</sup> В зависимости от ваших настроек `ls` может использовать также другие функции форматирования, например изменение цвета при выводе на экране. Но не при перенаправлении вывода.

## Команда #2: head

Команда `head` выводит первые строки файла. Выведем первые три строки файла *animals.txt*, используя `head` с параметром `-n`:

```
$ head -n3 animals.txt
python  Programming Python      2010    Lutz, Mark
snail    SSH, The Secure Shell      2005    Barrett, Daniel
alpaca   Intermediate Perl           2012    Schwartz, Randal
```

Если вы запрашиваете больше строк, чем содержится в файле, `head` выведет весь файл (как это делает `cat`). Если вы опустите параметр `-n`, заголовок по умолчанию будет состоять из 10 строк (`-n10`).

Сама по себе функция `head` удобна для просмотра начала файла, когда вас не интересует остальное содержимое. Эта команда работает быстро и эффективно даже с очень большими файлами, поскольку ей не нужно считывать весь файл. Кроме того, `head` умеет считывать данные из стандартного ввода и использует стандартный вывод, что делает ее полезной в конвейерах. Подсчитаем количество слов в первых трех строках файла *animals.txt*:

```
$ head -n3 animals.txt | wc -w
20
```

Обычное использование `head` заключается в сокращении вывода данных от другой команды, когда вам не нужно видеть полную информацию, например длинный список каталогов. Перечислим первые пять имен файлов в каталоге */bin*:

```
$ ls /bin | head -n5
bash
bsd-csh
bunzip2
busybox
bzipcat
```

## Команда #3: cut

Команда `cut` выводит одну или несколько колонок из файла. Например, выведем все названия книг, которые расположены во второй колонке файла *animals.txt*:

```
$ cut -f2 animals.txt
Programming Python
SSH, The Secure Shell
Intermediate Perl
MySQL High Availability
Linux in a Nutshell
```

Cisco IOS in a Nutshell  
Writing Word Macros

Команда `cut` поддерживает два способа определения, что считать колонкой. Первый — разделение по полям (`-f`), когда входные данные состоят из строк (полей), каждая из которых разделена одним символом табуляции. Именно такой формат используется в файле *animals.txt*. Команда `cut` из предыдущего примера печатает второе поле каждой строки благодаря опции `-f2`.

Чтобы сократить вывод, передадим его в `head`. Выведем на экран только первые три строки:

```
$ cut -f2 animals.txt | head -n3
Programming Python
SSH, The Secure Shell
Intermediate Perl
```

Вы также можете вырезать несколько полей, разделив их номера запятыми:

```
$ cut -f1,3 animals.txt | head -n3
python    2010
snail     2005
alpaca    2012
```

или указав диапазон значений:

```
$ cut -f2-4 animals.txt | head -n3
Programming Python    2010    Lutz, Mark
SSH, The Secure Shell  2005    Barrett, Daniel
Intermediate Perl      2012    Schwartz, Randal
```

Также можно определить колонку для команды `cut` по положению символа в строке с использованием параметра `-c`. Выведем первые три символа из каждой строки файла, которые можно указать либо через запятую (1, 2, 3), либо в формате диапазона (1–3):

```
$ cut -c1-3 animals.txt
pyt
sna
alp
rob
hor
don
ory
```

Теперь, когда вы ознакомились с основными функциями, попробуем сделать что-нибудь более практичное с помощью команды `cut` и с использованием каналов. Допустим, что файл *animals.txt* состоит из нескольких тысяч строк



и вам нужно извлечь только фамилии авторов. Сначала выделим четвертое поле — имя и фамилия автора:

```
$ cut -f4 animals.txt
Lutz, Mark
Barrett, Daniel
Schwartz, Randal
:
```

Затем передадим выходные данные снова в команду `cut`, используя параметр `-d` (`delimiter` — разделитель), чтобы изменить символ-разделитель на запятую вместо табуляции. Это позволит выделить только фамилии авторов:

```
$ cut -f4 animals.txt | cut -d, -f1
Lutz
Barrett
Schwartz
:
```



### Экономьте время благодаря истории команд и редактированию

Приходилось ли вам набирать одни и те же команды заново? Вместо этого несколько раз нажмите клавишу со стрелкой вверх, чтобы просмотреть команды, которые вы запускали ранее (эта функция оболочки называется *историей команд*). Когда дойдете до нужной команды, нажмите **Enter**, чтобы немедленно запустить ее, или сначала отредактируйте, используя клавиши со стрелками влево и вправо для позиционирования курсора и клавишу **Backspace** для удаления (эта функция называется *редактирование командной строки*). В главе 3 мы узнаем о значительно более мощных функциях истории команд и редактирования.

## Команда #4: `grep`

`grep` — чрезвычайно мощная команда, но пока скроем большую часть ее возможностей и ограничимся тем, что она печатает строки, соответствующие заданному шаблону (более подробная информация будет представлена в главе 5). Например, следующая команда отображает строки из файла *animals.txt*, содержащие текст `Nutshell`:

```
$ grep Nutshell animals.txt
horse   Linux in a Nutshell      2009    Siever, Ellen
donkey  Cisco IOS in a Nutshell  2005    Boney, James
```

Также можно вывести строки, которые не соответствуют заданному шаблону, с опцией `-v`. Обратите внимание, что строки, содержащие `Nutshell`, отсутствуют:

```
$ grep -v Nutshell animals.txt
python      Programming Python      2010      Lutz, Mark
snail       SSH, The Secure Shell      2005      Barrett, Daniel
alpaca      Intermediate Perl      2012      Schwartz, Randal
robin       MySQL High Availability      2014      Bell, Charles
oryx        Writing Word Macros      1999      Roman, Steven
```

Таким образом, команда `grep` полезна для поиска определенного текста в некотором списке файлов. Следующая команда печатает строки, содержащие текст `Perl`, в файлах с расширением `.txt`:

```
$ grep Perl *.txt
animals.txt:alpaca      Intermediate Perl      2012      Schwartz, Randal
essay.txt:really love the Perl programming language, which is
essay.txt:languages such as Perl, Python, PHP, and Ruby
```

В данном случае команда `grep` нашла три соответствия — в одной строке файла *animals.txt* и в двух строках файла *essay.txt*.

`grep` читает стандартный ввод и записывает стандартный вывод, что делает эту команду идеальной для конвейеров. Допустим, мы хотим узнать, сколько подкаталогов находится в каталоге `/usr/lib`. Нет простой команды Linux для получения ответа, поэтому создадим конвейер. Начнем с команды `ls -l`:

```
$ ls -l /usr/lib
drwxrwxr-x  12 root root   4096 Mar  1  2020 4kstogram
drwxr-xr-x   3 root root   4096 Nov 30  2020 GraphicsMagick-1.4
drwxr-xr-x   4 root root   4096 Mar 19  2020 NetworkManager
-rw-r--r--   1 root root 35568 Dec  1  2017 attica_kde.so
-rwxr-xr-x   1 root root   684 May  5  2018 cnf-update-db
:
```

Обратите внимание, что `ls -l` помечает каталоги буквой `d` в начале строки. Используем `cut`, чтобы вывести первый столбец:

```
$ ls -l /usr/lib | cut -c1
d
d
d
-
-
:
```

Затем используем `grep`, чтобы оставить только строки, содержащие букву `d`:

```
$ ls -l /usr/lib | cut -c1 | grep d
d
d
d
:
```

Наконец, подсчитаем строки с помощью команды `wc` — и мы получим ответ, созданный конвейером из четырех команд: `/usr/lib` содержит 145 подкаталогов:

```
$ ls -l /usr/lib | cut -c1 | grep d | wc -l
145
```

## Команда #5: sort

Команда `sort` сортирует строки файла в порядке возрастания (по умолчанию):

```
$ sort animals.txt
alpaca      Intermediate Perl      2012      Schwartz, Randal
donkey      Cisco IOS in a Nutshell 2005      Boney, James
horse       Linux in a Nutshell    2009      Siever, Ellen
oryx        Writing Word Macros    1999      Roman, Steven
python      Programming Python     2010      Lutz, Mark
robin       MySQL High Availability 2014      Bell, Charles
snail       SSH, The Secure Shell  2005      Barrett, Daniel
```

или в порядке убывания (с параметром `-r`):

```
$ sort -r animals.txt
snail       SSH, The Secure Shell  2005      Barrett, Daniel
robin       MySQL High Availability 2014      Bell, Charles
python      Programming Python     2010      Lutz, Mark
oryx        Writing Word Macros    1999      Roman, Steven
horse       Linux in a Nutshell    2009      Siever, Ellen
donkey      Cisco IOS in a Nutshell 2005      Boney, James
alpaca      Intermediate Perl      2012      Schwartz, Randal
```

`sort` может сортировать строки в алфавитном порядке (по умолчанию) или в числовом порядке (с опцией `-n`). Продемонстрируем это на примере конвейеров, которые вырезают третье поле (год публикации) в `animals.txt`:

```
$ cut -f3 animals.txt                                несортированный
2010
2005
2012
2014
2009
2005
1999
$ cut -f3 animals.txt | sort -n                       по возрастанию
1999
2005
2005
2009
2010
2012
2014
```

```
$ cut -f3 animals.txt | sort -nr          по убыванию
2014
2012
2010
2009
2005
2005
1999
```

Чтобы узнать год выхода самой новой книги в *animals.txt*, направим вывод `sort` на ввод `head` и напечатаем только первую строку:

```
$ cut -f3 animals.txt | sort -nr | head -n1
2014
```



### Максимальные и минимальные значения

`sort` и `head` — мощные партнеры при работе с числовыми данными, если они расположены по одному в каждой новой строке. Вы можете вывести максимальное значение с помощью такого конвейера:

```
... | sort -nr | head -n1
```

а минимальное значение с помощью следующего:

```
... | sort -n | head -n1
```

Рассмотрим другой пример с использованием файла */etc/passwd*, содержащего список пользователей, которые могут запускать процессы в системе<sup>1</sup>. Создадим список всех пользователей в алфавитном порядке. Первые пять строк выглядят примерно так:

```
$ head -n5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
smith:x:1000:1000:Aisha Smith,,,:/home/smith:/bin/bash
jones:x:1001:1001:Bilbo Jones,,,:/home/jones:/bin/bash
```

Каждая строка состоит из значений, разделенных двоеточиями, и первое из них — это имя пользователя (логин). Поэтому мы можем изолировать имена пользователей с помощью команды `cut`:

```
$ head -n5 /etc/passwd | cut -d: -f1
root
daemon
bin
smith
jones
```

---

<sup>1</sup> Некоторые системы Linux хранят информацию о пользователе в другом месте.

а затем отсортировать их:

```
$ head -n5 /etc/passwd | cut -d: -f1 | sort
bin
daemon
jones
root
smith
```

Создадим сортированный список всех имен пользователей, а не только первых пяти из них, заменив `head` на `cat`:

```
$ cat /etc/passwd | cut -d: -f1 | sort
```

Чтобы определить, есть ли у данного пользователя учетная запись в системе, сопоставим его логин с помощью `grep`. Пустой вывод означает отсутствие учетной записи:

```
$ cut -d: -f1 /etc/passwd | grep -w jones
jones
$ cut -d: -f1 /etc/passwd | grep -w rutabaga      пустой вывод
```

Параметр `-w` указывает команде `grep` сопоставлять только слова целиком, а не их части. Если в системе имеется имя пользователя, содержащее *jones*, например *sallyjones2*, то команда из примера выше его не отобразит.

## Команда #6: `uniq`

Команда `uniq` обнаруживает повторяющиеся соседние строки в файле. По умолчанию она удаляет повторы. Продемонстрируем это на файле, содержащем только заглавные буквы:

```
$ cat letters
A
A
A
B
B
A
C
C
C
C
$ uniq letters
A
B
A
C
```

Обратите внимание, что команда `uniq` сократила первые три строки А до одной А, но оставила последнюю А на месте, потому что она не была *соседней* с первыми тремя.

С помощью параметра `-c` можно подсчитать количество повторяющихся строк:

```
$ uniq -c letters
 3 A
 2 B
 1 A
 4 C
```

Признаюсь честно, когда я впервые столкнулся с командой `uniq`, то не увидел в ней особой пользы, но она быстро стала одной из моих любимых. Предположим, у нас есть разделенный табуляцией файл с итоговыми оценками студентов университета в диапазоне от А (лучшая) до F (худшая):

```
$ cat grades
C    Geraldine
B    Carmine
A    Kayla
A    Sophia
B    Haresh
C    Liam
B    Elijah
B    Emma
A    Olivia
D    Noah
F    Ava
```

Нам нужно вывести на экран оценку, которую получила большая часть студентов (если будет равное количество у нескольких оценок, в вывод попадет первая из них). Начнем с извлечения столбца с оценками с помощью команды `cut`:

```
$ cut -f1 grades | sort
A
A
A
B
B
B
B
C
C
D
F
```

Затем используем команду `uniq` для подсчета совпадающих строк:

```
$ cut -f1 grades | sort | uniq -c
 3 A
```

```
4 B
2 C
1 D
1 F
```

Затем отсортируем строки в порядке убывания, чтобы переместить наиболее часто встречающуюся оценку в верхнюю строку:

```
$ cut -f1 grades | sort | uniq -c | sort -nr
4 B
3 A
2 C
1 F
1 D
```

и оставим только первую строку с помощью команды `head`:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1
4 B
```

И наконец, поскольку нам нужна только буквенная оценка, а не количество, извлечем ее с помощью `cut`:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1 | cut -c9
B
```

И вот вам ответ, благодаря конвейеру из шести команд — пока что самому длинному из использованных нами. Такого рода пошаговое построение конвейера — не просто обучающее упражнение. Именно так на самом деле работают эксперты Linux. Этой технике посвящена глава 8.

## Обнаружение дубликатов файлов

Давайте объединим в большом примере все то, что уже узнали. Предположим, мы находимся в каталоге, который забит файлами JPEG, и хотим узнать, не дублируются ли они:

```
$ ls
image001.jpg image005.jpg image009.jpg image013.jpg image017.jpg
image002.jpg image006.jpg image010.jpg image014.jpg image018.jpg
:
```

Ответить на этот вопрос можно с помощью конвейера. Нам понадобится еще одна команда, `md5sum`, которая проверяет содержимое файла и вычисляет 32-символьную строку, называемую *контрольной суммой*:

```
$ md5sum image001.jpg
146b163929b6533f02e91bdf21cb9563 image001.jpg
```

Контрольная сумма данного файла по математическим законам с очень большой вероятностью будет уникальной. Если два файла имеют одинаковую контрольную сумму, то они почти наверняка являются копиями. Здесь `md5sum` указывает, что первый и третий файлы являются копиями друг друга:

```
$ md5sum image001.jpg image002.jpg image003.jpg
146b163929b6533f02e91bdf21cb9563 image001.jpg
63da88b3ddd0843c94269638dfa6958 image002.jpg
146b163929b6533f02e91bdf21cb9563 image003.jpg
```

Повторяющиеся контрольные суммы легко обнаружить на глаз, когда файлов всего три, но что делать, если их три тысячи? На помощь приходят каналы. Вычислим все контрольные суммы, с помощью `cut` изолируем первые 32 символа каждой строки и отсортируем строки, чтобы дубликаты располагались рядом друг с другом:

```
$ md5sum *.jpg | cut -c1-32 | sort
1258012d57050ef6005739d0e6f6a257
146b163929b6533f02e91bdf21cb9563
146b163929b6533f02e91bdf21cb9563
17f339ed03733f402f74cf386209aeb3
:
```

Далее добавим команду `uniq` для подсчета повторяющихся строк:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c
 1 1258012d57050ef6005739d0e6f6a257
 2 146b163929b6533f02e91bdf21cb9563
 1 17f339ed03733f402f74cf386209aeb3
:
```

Если нет дубликатов, все результаты подсчета, выведенные на экран командой `uniq`, будут равны 1. Отсортируем результаты по убыванию, и значения, которые больше 1, будут выведены в верхней части списка:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
 3 f6464ed766daca87ba407aede21c8fcc
 2 c7978522c58425f6af3f095ef1de1cd5
 2 146b163929b6533f02e91bdf21cb9563
 1 d8ad913044a51408ec1ed8a204ea9502
:
```

Теперь давайте удалим все недублированные элементы. Их контрольным суммам предшествуют шесть пробелов, число 1 и один пробел. Мы используем `grep -v` для удаления этих строк<sup>1</sup>:

<sup>1</sup> Технически команда `sort -nr` не нужна в этом конвейере, потому что `grep` удаляет все недублированные элементы.



```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr | grep -v "      1 "  
3 f6464ed766daca87ba407aede21c8fcc  
2 c7978522c58425f6af3f095ef1de1cd5  
2 146b163929b6533f02e91bdf21cb9563
```

И наконец, у нас есть список повторяющихся контрольных сумм, отсортированный по количеству вхождений, созданный изящным конвейером из шести команд. Если он не выводит ничего, значит, дубликатов файлов в каталоге нет.

Эта команда была бы еще полезней, если бы отображала имена файлов, но для этой операции требуются функции, которые мы еще не обсуждали, вы узнаете о них в разделе «Улучшенный способ обнаружения дубликатов файлов» на с. 112. А пока что идентифицируем файлы с заданной контрольной суммой, выполнив поиск с помощью `grep`:

```
$ md5sum *.jpg | grep 146b163929b6533f02e91bdf21cb9563  
146b163929b6533f02e91bdf21cb9563 image001.jpg  
146b163929b6533f02e91bdf21cb9563 image003.jpg
```

и почистим вывод с помощью `cut`:

```
$ md5sum *.jpg | grep 146b163929b6533f02e91bdf21cb9563 | cut -c35-  
image001.jpg  
image003.jpg
```

## Резюме

Теперь вы увидели всю мощь стандартного ввода, стандартного вывода и каналов. Они превращают небольшую горстку команд в набор комбинируемых инструментов, доказывая, что целое всегда есть нечто большее, чем простая сумма его частей. Любая команда, которая читает стандартный ввод или записывает стандартный вывод, может участвовать в создании конвейера<sup>1</sup>. По мере изучения дополнительных команд вы сможете применять общие принципы из этой главы для создания собственных эффективных комбинаций.

---

<sup>1</sup> Некоторые команды не используют `stdin` и `stdout` и поэтому не могут читать из каналов или записывать в них. Примеры: `mv` и `rm`. Однако конвейеры могут использовать эти команды другими способами, примеры рассмотрены в главе 8.

# Знакомство с командной оболочкой

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

Итак, вы можете запускать команды после приглашения командной строки. Но *что означает* это приглашение, откуда оно появляется и как выполняются ваши команды?

Приглашение создает *командная оболочка* Linux. Это интерфейс пользователя, который обеспечивает взаимодействие между вами и операционной системой. Linux поддерживает несколько командных оболочек, но наиболее распространенная из них — `bash`. В этой книге по умолчанию используется именно `bash`, а описание других командных оболочек вы найдете в Приложении Б.

Функции `bash` и других оболочек значительно шире, чем простое выполнение команд. Например, команда может включать в себя символ групповых операций (\*) для одновременной ссылки на несколько файлов:

```
$ ls *.py
data.py main.py user_interface.py
```

Этот символ обрабатывается командной оболочкой, а не программой `ls`. Командная оболочка заменяет выражение `*.py` списком имен подходящих файлов еще до запуска программы `ls`. Иными словами, `ls` не увидит символ групповой операции. С точки зрения `ls`, вы ввели команду следующего вида:

```
$ ls data.py main.py user_interface.py
```

Командная оболочка также обрабатывает данные в каналах, с которыми вы познакомились в главе 1. Она незаметно перенаправляет стандартные потоки `stdin` и `stdout`, поэтому исполняемые программы не знают, что они взаимодействуют друг с другом.

При каждом запуске команды за одни этапы ее выполнения отвечает вызываемая программа, например `ls`, а за другие — оболочка. Опытные пользователи умеют создавать длинные сложные команды, поскольку *знают, что сделает команда*, еще до того, как нажмут Enter, и понимают разделение обязанностей между оболочкой и вызываемыми ею программами.

В этой главе мы познакомим вас с оболочкой Linux. Я буду использовать тот же минималистский подход, что и для команд и конвейеров в главе 1. Вместо того чтобы описывать десятки функций оболочки, я дам вам ровно столько информации, сколько нужно, чтобы перейти к следующему шагу вашего обучения:

- сопоставление шаблонов имен файлов;
- переменные для хранения значений;
- экранирование и использование кавычек для отключения некоторых функций оболочки;
- путь для поиска исполняемых программ;
- сохранение изменений в окружении оболочки.

## Терминология командной оболочки

Слово *оболочка* употребляется в разных значениях. Например, иногда имеется в виду *концепция* оболочки Linux в целом, как во фразах «командная оболочка — это мощный инструмент» или «`bash` — это командная оболочка». В других случаях речь идет о конкретном *экземпляре* оболочки на компьютере с Linux, ожидающем вашей следующей команды.

В этой книге смысл понятия *оболочка* в большинстве случаев должен быть ясен из контекста. При необходимости я буду уточнять, что имеется в виду: *экземпляр* оболочки, *работающая* или ваша *текущая* командная оболочка.

Некоторые экземпляры оболочки предоставляют пользователю приглашение командной строки, чтобы он мог взаимодействовать с ними. Для обозначения этих экземпляров я буду использовать термин *интерактивная оболочка*. Другие экземпляры оболочки не интерактивны — они выполняют последовательность команд и завершают работу.

## Сопоставление шаблонов имен файлов

В главе 1 вы работали с командами `cut`, `sort` и `grep`, которые могут принимать одно или несколько имен файлов в качестве аргументов. Например, вы можете искать слово `Linux` в сотне файлов с именами от *chapter1* до *chapter100*:

```
$ grep Linux chapter1 chapter2 chapter3 chapter4 chapter5...
```

Перечислять множество имен файлов — довольно утомительное занятие. Поэтому оболочка позволяет использовать специальные символы для сокращений, которые применяются к файлам и каталогам. Часто их называют символами групповых операций, но более общая концепция называется *сопоставлением с шаблоном* (*pattern matching*) или *подстановкой* (*globbing*). Сопоставление с шаблоном — это один из двух наиболее популярных приемов для повышения скорости работы в Linux (другой прием — нажать клавишу со стрелкой вверх, чтобы вызвать предыдущую команду оболочки, — мы рассмотрим его в главе 3).

В Linux символ звездочки `*` соответствует любой последовательности (за исключением начальной точки в именах файлов или каталогов)<sup>1</sup> из любого числа символов в путях к файлам или каталогам:

```
$ grep Linux chapter*
```

Остается за кадром, как командная оболочка (не программа `grep`!) преобразует шаблон `chapter*` в список подходящих имен файлов. И только после этого оболочка запускает `grep`.

Еще один специальный символ — знак вопроса `?`, который соответствует любому единичному символу, за исключением начальной точки в именах файлов или каталогов. Например, вы можете выполнить поиск слова *Linux* в главах с 1 по 9, используя вместо цифр знак вопроса, чтобы командная оболочка смогла найти совпадения:

```
$ grep Linux chapter?
```

Для поиска в главах с 10 по 99 придется использовать два знака вопроса:

```
$ grep Linux chapter??
```

---

<sup>1</sup> По этой причине команда `ls *` не отобразит список так называемых скрытых файлов и каталогов (*dotfiles*), которые начинаются с точки.

Менее известно использование квадратных скобок [ ] для запроса у командной оболочки соответствия одному из символов набора. Например, вы можете искать только в первых пяти главах:

```
$ grep Linux chapter[12345]
```

Задать диапазон можно с помощью тире:

```
$ grep Linux chapter[1-5]
```

Вы также можете искать главы с четными номерами, комбинируя звездочку и квадратные скобки, чтобы оболочка искала соответствия именам файлов, оканчивающимся на четную цифру:

```
$ grep Linux chapter*[02468]
```

Любые символы, не только цифры, могут быть помещены в квадратные скобки для сопоставления. Например, следующая команда заставит оболочку искать имена файлов, начинающиеся с заглавной буквы, содержащие символ подчеркивания и заканчивающиеся символом @:

```
$ ls [A-Z]*_*@
```



### Терминология: вычисление выражений и расширение шаблонов

Строки, которые вы вводите в командной строке, например `chapter*` или `Efficient linux`, называются *выражениями*. Команда, например `ls -l chapter*`, также является выражением.

Когда оболочка интерпретирует и обрабатывает специальные символы в выражении, такие как звездочка и вертикальная черта, мы говорим, что оболочка *вычисляет* выражение.

Сопоставление с шаблоном — это один из видов вычислений. Когда командная оболочка вычисляет выражение, которое содержит символы групповых операций, например `chapter*`, и заменяет его именами файлов, соответствующими шаблону, мы говорим, что оболочка *расширяет* шаблон.

Шаблоны применимы практически всегда, когда вы указываете пути к файлам или каталогам в командной строке. Например, вы можете перечислить все файлы в каталоге `/etc` с именами, заканчивающимися на `.conf`, следующим образом:

```
$ ls -l /etc/*.conf
/etc/adduser.conf
/etc/appstream.conf
```

```
:  
/etc/wodim.conf
```

Будьте осторожны, используя шаблон с командой, которая принимает только один аргумент — имя файла или каталога. Например, с командой `cd` вы можете получить совсем не то, что ожидали:

```
$ ls  
Pictures Poems Politics  
$ cd P*          Будет найдено три подходящих каталога  
bash: cd: too many arguments
```

Если шаблон не соответствует ни одному файлу, оболочка передает его в качестве аргумента команды. В команде ниже шаблон `*.doc` не соответствует ни одному файлу в текущем каталоге, поэтому команда `ls` ищет файл с именем `*.doc` и сообщает о неудаче:

```
$ ls *.doc  
/bin/ls: cannot access '*.doc': No such file or directory
```

При работе с шаблонами важно помнить два момента. Во-первых, как я уже подчеркивал, сопоставление с образцом выполняет командная оболочка, а не вызываемая программа. Удивительно, что многие пользователи Linux не знают об этом и недоумевают, почему одни команды выполняются успешно, а другие — нет.

Второй важный момент заключается в том, что сопоставление с образцом работает только с именами файлов и каталогов. Оно неприменимо для имен пользователей, хостов и других типов аргументов. Вы также не можете ввести, например, `s?rt` в начале командной строки и ожидать, что оболочка запустит программу сортировки (некоторые команды Linux, такие как `grep`, `sed` и `awk`, поддерживают особые виды сопоставления с шаблоном, которые мы рассмотрим в главе 5).



### Сопоставление имен файлов с шаблоном и ваши программы

Все программы, которые принимают имена файлов в качестве аргументов, автоматически поддерживают сопоставление с шаблоном, поскольку командная оболочка обрабатывает шаблоны до запуска программы. Это верно и для написанных вами программ и скриптов. Например, вы написали программу `english2swedish`, которая переводит файлы с английского языка на шведский и принимает несколько имен файлов в командной строке. Вы можете без проблем запустить ее с шаблоном:

```
$ english2swedish *.txt
```

## Вычисление переменных

Командная оболочка может определять переменные и сохранять в них значения. Переменная оболочки очень похожа на переменную в алгебре — у нее есть имя и значение. Примером может служить переменная оболочки `HOME`. Ее значением является путь к вашему домашнему каталогу Linux, например `/home/smith`. Другой пример — переменная `USER`, значением которой является ваше имя пользователя Linux. В этой книге `smith` будет использоваться как значение переменной `USER`.

Для вывода на экран значений переменных `HOME` и `USER` запустите команду `printenv`:

```
$ printenv HOME
/home/smith
$ printenv USER
smith
```

Когда командная оболочка вычисляет переменную, она заменяет имя переменной на ее значение. Чтобы вычислить переменную, просто поместите знак доллара перед именем. Например, `$HOME` возвращает строку `/home/smith`.

Самый простой способ увидеть, как оболочка обрабатывает командную строку, — запустить команду `echo`, которая выводит свои аргументы (после того как оболочка завершит их вычисление):

```
$ echo My name is $USER and my files are in $HOME      Вычисление переменных
My name is smith and my files are in /home/smith
$ echo ch*ter9                                         Вычисление шаблона
chapter9
```

## Откуда берутся значения переменных

Такие переменные, как `USER` и `HOME`, предопределены оболочкой. Их значения устанавливаются автоматически при входе в систему (подробности об этом мы узнаем позже). Имена предопределенных переменных традиционно набираются прописными буквами.

Вы можете определить или изменить переменную в любое время: присвоить ей значение, используя следующий синтаксис:

```
name=value
```

Например, если вы часто работаете с каталогом `/home/smith/Projects`, вы можете присвоить его имя переменной:

```
$ work=$HOME/Projects
```

и использовать его как удобное сокращение при работе с `cd`:

```
$ cd $work
$ pwd
/home/smith/Projects
```

Вы можете передавать `$work` любой команде, ожидающей имя каталога, в качестве аргумента:

```
$ cp myfile $work
$ ls $work
Myfile
```

При определении переменной не допускается использование пробелов около знака равенства. Если вы об этом забудете, оболочка примет (ошибочно), что первое слово в командной строке — это запускаемая программа, а знак равенства и значение — ее аргументы, и вы увидите сообщение об ошибке:

```
$ work = $HOME/Projects      Командная оболочка решила, что «work» — это команда
work: command not found
```

Правила работы с определяемыми пользователями переменными, как, например, `work`, — такие же, как и с системными переменными вроде `HOME`. Единственное отличие состоит в том, что некоторые программы Linux меняют свое поведение на основе значений `HOME`, `USER` и других *системных* переменных. Например, программа Linux с графическим интерфейсом может получить ваше имя пользователя из оболочки и отобразить его. Такие программы не обращают внимания на созданную пользователем переменную, такую, например, как `work`.

## Переменные и заблуждения

Когда вы выводите на экран значение переменной с помощью команды `echo`:

```
$ echo $HOME
/home/smith
```

вы можете подумать, что команда `echo` проверяет переменную `HOME` и выводит ее значение. На самом деле *это не так*. Команда `echo` ничего не знает о переменных. Она просто выводит на экран любые аргументы, которые вы ей передаете. Значение `$HOME` вычисляет командная оболочка перед запуском команды `echo`. С точки зрения `echo`, вы набрали:

```
$ echo /home/smith
```

Этот принцип важно понять перед тем, как мы углубимся в более сложные команды. Оболочка вычисляет переменные, шаблоны и другие конструкции перед выполнением команды.



## Шаблоны vs переменные

Давайте проверим ваше понимание шаблонов и вычисления переменных. Предположим, вы находитесь в каталоге с двумя подкаталогами, *mammals* и *reptiles*<sup>1</sup>. И, как ни странно, в подкаталоге *mammals* находятся файлы *lizard.txt* и *snake.txt*<sup>2</sup>:

```
$ ls
mammals reptiles
$ ls mammals
lizard.txt snake.txt
```

Поскольку ящерицы и змеи не являются млекопитающими, эти два файла следует переместить в подкаталог *reptiles*. Вот два способа сделать это, но один работает, а другой нет:

```
mv mammals/*.txt reptiles    Метод 1
```

```
FILES="lizard.txt snake.txt"
mv mammals/$FILES reptiles    Метод 2
```

Метод 1 работает, потому что шаблон соответствует всему пути к файлу — имя каталога *mammals* является частью обоих совпадений для *mammals/\*.txt*:

```
$ echo mammals/*.txt
mammals/lizard.txt mammals/snake.txt
```

Таким образом, метод 1 работает так, как если бы вы набрали следующую корректную команду:

```
$ mv mammals/lizard.txt mammals/snake.txt reptiles
```

В методе 2 используются переменные, имеющие только свои буквенные значения, и нет специального инструмента для вычисления путей к файлам:

```
$ echo mammals/$FILES
mammals/lizard.txt snake.txt
```

Следовательно, метод 2 работает так, как если бы вы набрали следующую не совсем корректную команду:

```
$ mv mammals/lizard.txt snake.txt reptiles
```

Эта команда ищет файл *snake.txt* в текущем каталоге, а не в каталоге *mammals*, и выдает ошибку:

---

<sup>1</sup> Mammals — млекопитающие, reptiles — рептилии. — *Примеч. перев.*

<sup>2</sup> Lizard — ящерица, snake — змея. — *Примеч. перев.*

```
$ mv mammals/$FILES reptiles
/bin/mv: cannot stat 'snake.txt': No such file or directory
```

Чтобы сделать метод 2 рабочим, используйте цикл `for`, который добавляет имя каталога *mammals* к каждому имени файла:

```
FILES="lizard.txt snake.txt"
for f in $FILES; do
mv mammals/$f reptiles
done
```

## Сокращение команд с помощью псевдонимов

Переменная — это имя, которому соответствует определенное значение. В командной оболочке также есть имена, которым соответствуют команды. Они называются *псевдонимами* (*aliases*). Определим псевдоним, поставив после него знак равенства и команду:

```
$ alias g=grep          Пример команды без аргументов
$ alias ll="ls -l"      Пример команды с аргументами, кавычки необходимы
```

Запустите псевдоним, введя его имя в качестве команды. Когда псевдонимы короче команд, которые они вызывают, вы экономите время на наборе текста:

```
$ ll          Запускается «ls -l»
-rw-r--r-- 1 smith smith 325 Jul 3 17:44 animals.txt
$ g Nutshell Запускается «grep Nutshell animals.txt»
horse      Linux in a Nutshell      2009 Siever, Ellen
donkey     Cisco IOS in a Nutshell   2005 Boney, James
```



Всегда определяйте псевдоним в отдельной строке, а не как часть комбинированной команды (технические подробности смотрите в `man bash`).

Вы можете определить псевдоним с тем же именем, что и у существующей команды, фактически заменив эту команду в вашей оболочке. Эта практику можно назвать *затенением* (*shadowing*) команды. Предположим, вам нравится команда `less` для чтения файлов, но вы хотите, чтобы она очищала экран перед отображением каждой страницы. Эта функция активируется опцией `-c`, поэтому определите псевдоним с именем `less`, который запускает `less -c`:

```
$ alias less="less -c"
```

<sup>3</sup> `bash` предотвращает бесконечную рекурсию, не расширяя второе значение `less` как псевдоним.

Псевдонимы имеют приоритет над командами с тем же именем, следовательно, теперь вы затенили команду `less` в текущей оболочке. К вопросу о приоритете мы вернемся в разделе «Путь для поиска и псевдонимы».

Чтобы вывести все псевдонимы и их значения, которые определены на данный момент в вашей командной оболочке, запустите `alias` без аргументов:

```
$ alias
alias g='grep'
alias ll='ls -l'
```

Чтобы вывести значение конкретного псевдонима, запустите `alias` и передайте его имя в качестве аргумента:

```
$ alias g
alias g='grep'
```

Чтобы удалить псевдоним из оболочки, выполните `unalias`:

```
$ unalias g
```

## Перенаправление ввода и вывода

Оболочка управляет вводом и выводом запускаемых команд. Вы уже видели пример каналов, которые направляют стандартный вывод одной команды в стандартный ввод другой. Использование вертикальной черты `|` — это особенность оболочки.

Еще одна функция оболочки — перенаправление стандартного вывода в файл. Например, если вы используете `grep` для поиска подходящих строк в файле *animals.txt* из примера 1.1, то по умолчанию команда направит их в стандартный вывод:

```
$ grep Perl animals.txt
alpaca Intermediate Perl    2012  Schwartz, Randal
```

Вместо этого вы можете направить вывод команды в файл, используя функцию оболочки, называемую *перенаправлением вывода (output redirection)*. Просто добавьте символ `>` с именем файла, в который будет перенаправлен вывод:

```
$ grep Perl animals.txt > outfile      на экран ничего не выводится
$ cat outfile
alpaca Intermediate Perl    2012  Schwartz, Randal
```

Вы только что перенаправили стандартный вывод команды в файл *outfile* вместо вывода его на экран. Если файл *outfile* не существует, он будет создан. Если же он

существует, перенаправление перезапишет его содержимое. Если вы хотите добавлять данные в файл, а не перезаписывать его каждый раз, используйте символ `>>`:

```
$ grep Perl animals.txt > outfile          Создание или перезапись файла outfile
$ echo There was just one match >> outfile  Добавление записи в конец файла outfile
$ cat outfile
alpaca Intermediate Perl          2012      Schwartz, Randal
There was just one match
```

Помимо перенаправления вывода, возможно и *перенаправление ввода* (*input redirection*), обеспечивающее стандартный ввод из файла, а не с клавиатуры. Используйте символ `<` и имя файла, чтобы перенаправить стандартный ввод.

Многие команды Linux, которые принимают в качестве аргументов имена файлов и затем считывают данные из них, также могут получать данные из стандартного ввода. Примером является команда `wc` для подсчета строк, слов и символов в файле:

```
$ wc animals.txt                          Чтение из файла
 7 51 325 animals.txt
$ wc < animals.txt                        Чтение из перенаправленного ввода
 7 51 325
```

*Очень важно* понимать, чем отличается поведение этих двух команд `wc`:

- В первом случае команда `wc` получает в качестве аргумента имя файла *animals.txt*, поэтому она находит, открывает и читает содержимое файла на диске.
- Во втором случае команда `wc` вызывается без аргументов, поэтому они считываются со стандартного ввода. Обычно стандартный ввод выполняется с клавиатуры, но командная оболочка перенаправляет его. При этом `wc` понятия не имеет о существовании файла *animals.txt*.

Командная оболочка может перенаправлять ввод и вывод в рамках одной команды:

```
$ wc < animals.txt > count
$ cat count
 7 51 325
```

и даже может использовать каналы одновременно. Ниже `grep` читает из перенаправленного стандартного ввода и передает результаты в команду `wc`, которая перенаправляет стандартный вывод в файл *count*, создавая его:

```
$ grep Perl < animals.txt | wc > count
$ cat count
 1          6          47
```

Такие комбинированные команды будут детально рассмотрены в главе 8, но примеры перенаправления будут встречаться во всей книге.

### СТАНДАРТНЫЙ ПОТОК ОШИБОК (STDERR) И ЕГО ПЕРЕНАПРАВЛЕНИЕ

Некоторые выходные данные, такие как сообщения об ошибках, не могут быть перенаправлены с помощью `>`. Например, попросите `cp` скопировать несуществующий файл — и он выдаст следующее сообщение об ошибке:

```
$ cp nonexistent.txt file.txt
cp: cannot stat 'nonexistent.txt': No such file or directory
```

Если вы перенаправите стандартный вывод такой команды `cp` в файл, например `errors`, то сообщение все равно будет выведено на экран:

```
$ cp nonexistent.txt file.txt > errors
cp: cannot stat 'nonexistent.txt': No such file or directory
```

а файл `errors` останется пустым:

```
$ cat errors Ничего не выводит
```

Почему так происходит? Команды Linux могут создавать более одного потока вывода. В дополнение к `stdout` есть также `stderr` (стандартный поток ошибок) — второй поток вывода, который традиционно зарезервирован для сообщений об ошибках. Поток `stderr` и `stdout` выглядят одинаково на дисплее, но по своей сути они различны. Вы можете перенаправить `stderr` с помощью символов `2>`, за которым следует имя файла:

```
$ cp nonexistent.txt file.txt 2> errors
$ cat errors
cp: cannot stat 'nonexistent.txt': No such file or directory
```

или же сообщение об ошибке можно дописать в конец файла с помощью `2>>`:

```
$ cp nonexistent.txt file.txt 2> errors
$ cp another.txt file.txt 2>> errors
$ cat errors
cp: cannot stat 'nonexistent.txt': No such file or directory
cp: cannot stat 'another.txt': No such file or directory
```

Для перенаправления обоих потоков `stdout` и `stderr` в один файл используйте символы `&>`:

```
$ echo This file exists > goodfile.txt Создание файла
$ cat goodfile.txt nonexistent.txt &> all.output
$ cat all.output
This file exists
cat: nonexistent.txt: No such file or directory
```

## Отключение вычисления с помощью кавычек и экранирования

Обычно оболочка использует пробелы в качестве разделителей между словами. Следующая команда состоит из четырех слов — имени программы, за которым следуют три аргумента:

```
$ ls file1 file2 file3
```

Однако иногда нужно, чтобы оболочка воспринимала пробелы как символы, а не как разделители. Типичным примером являются пробелы в имени файла, например *Efficient Linux Tips.txt*:

```
$ ls -l
-rw-r--r-- 1 smith smith 36 Aug 9 22:12 Efficient Linux Tips.txt
```

Если вы обратитесь к файлу с таким именем в командной строке, ваша команда, скорее всего, завершится ошибкой, поскольку оболочка интерпретирует символы пробела как разделители:

```
$ cat Efficient Linux Tips.txt
cat: Efficient: No such file or directory
cat: Linux: No such file or directory
cat: Tips.txt: No such file or directory
```

Чтобы заставить оболочку рассматривать пробелы как часть имени файла, у вас есть три варианта — одинарные кавычки, двойные кавычки и обратная косая черта:

```
$ cat 'Efficient Linux Tips.txt'
$ cat "Efficient Linux Tips.txt"
$ cat Efficient\ Linux\ Tips.txt
```

Одинарные кавычки сообщают командной оболочке, что каждый символ в строке следует обрабатывать буквально, даже если он имеет особое значение для оболочки, как, например, пробелы и знаки доллара:

```
$ echo '$HOME'
$HOME
```

Двойные кавычки указывают оболочке воспринимать все символы буквально, за исключением знаков доллара и некоторых других, о которых вы узнаете позже:

```
$ echo "Notice that $HOME is evaluated"      Двойные кавычки
Notice that /home/smith is evaluated
$ echo 'Notice that $HOME is not'           Одинарные кавычки
Notice that $HOME is not
```

Обратная косая черта, также называемая *экранирующим символом*, указывает оболочке воспринимать буквально символ, находящийся после нее. В следующей команде экранируется знак доллара:

```
$ echo \$HOME
$HOME
```

Обратная косая черта действует как экранирующий символ в двойных кавычках:

```
$ echo "The value of \$HOME is $HOME"
The value of $HOME is /home/smith
```

но не в одинарных:

```
$ echo 'The value of \$HOME is $HOME'
The value of \$HOME is $HOME
```

Можно использовать обратную косую черту, чтобы экранировать символ двойных кавычек внутри двойных кавычек:

```
$ echo "This message is \"sort of\" interesting"
This message is "sort of" interesting
```

Обратная косая черта в конце строки отключает функцию невидимого символа новой строки, позволяя командам оболочки занимать несколько строк:

```
$ echo "This is a very long message that needs to extend \
onto multiple lines"
This is a very long message that needs to extend onto multiple lines
```

Обратная косая черта в конце строки — это прекрасный способ сделать конвейер более читаемым, как, скажем, вот этот пример из подраздела «Команда #6: `uniq`» со с. 30:

```
$ cut -f1 grades \
| sort \
| uniq -c \
| sort -nr \
| head -n1 \
| cut -c9
```

При таком использовании обратную косую черту иногда называют символом продолжения строки.

Начальная обратная косая черта перед псевдонимом экранирует его, заставляя оболочку искать команду с тем же именем. Затенение при этом игнорируется:

<pre>\$ alias less="less -c"</pre>	<i>Определение псевдонима</i>
<pre>\$ less myfile</pre>	<i>Запуск псевдонима, который определен как less -c</i>
<pre>\$ \less myfile</pre>	<i>Запуск стандартной команды less, а не псевдонима</i>

## Расположение исполняемых программ

Когда оболочка впервые встречает простую команду, например `ls *.py`, то разбивает строку на два слова: `ls` и `*.py`. В этом случае первое слово — это имя программы на диске, и оболочка должна ее найти, чтобы запустить.

Оказывается, программа `ls` представляет собой исполняемый файл в каталоге `/bin`. Вы можете проверить ее местоположение с помощью команды:

```
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 133792 Jan 18 2018 /bin/ls
```

или изменить текущий каталог с помощью команды `cd /bin` и запустить загадочную команду:

```
$ ls ls
ls
```

которая использует программу `ls` для отображения исполняемого файла `ls`.

Как оболочка находит `ls` в каталоге `/bin`? За кадром остается то, как оболочка сверяется с предварительно составленным списком каталогов, хранящимся в ее памяти, который называется *путем для поиска исполняемых программ*. Этот список хранится как значение переменной оболочки `PATH`:

```
$ echo $PATH
/home/smith/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/java/bin
```

Каталоги в пути для поиска разделяются двоеточиями (:). Для большей наглядности преобразуйте двоеточия в переводы строки, передав вывод команде `tr`, которая заменяет символы (подробнее см. главу 5):

```
$ echo $PATH | tr : "\n"
/home/smith/bin
/usr/local/bin
/usr/bin
/bin
/usr/games
/usr/lib/java/bin
```

Оболочка просматривает каталоги в списке, пытаясь найти программу `ls`. «Существует ли `/home/smith/bin/ls`? Нет. Существует ли `/usr/local/bin/ls`? Нет. Как насчет `/usr/bin/ls`? Снова нет! Может, `/bin/ls`? Да, вот оно! Запускаю `/bin/ls`. Этот поиск происходит слишком быстро, чтобы можно было его заметить<sup>1</sup>.

---

<sup>1</sup> Некоторые оболочки запоминают (кэшируют) пути к программам по мере их обнаружения, сокращая количество поисковых запросов в будущем.



Чтобы найти программу в списке каталогов для поиска, используйте команду `which`:

```
$ which cp
/bin/cp
$ which which
/usr/bin/which
```

или встроенную в оболочку более мощную и информативную команду `type`, которая также находит псевдонимы, функции и встроенные в оболочку команды<sup>1</sup>.

```
$ type cp
cp is hashed (/bin/cp)
$ type ll
ll is aliased to '/bin/ls -l'
$ type type
type is a shell builtin
```

Если есть программы с одинаковыми названиями, расположенные в разных каталогах, например `/usr/bin/less` и `/bin/less`, оболочка запустит тот файл, который найдет первым. Используя эту особенность, вы можете переопределить команду, поместив соответствующий файл в каталог, который будет просматриваться раньше, например в ваш личный каталог `$HOME/bin`.



### Путь для поиска и псевдонимы

Когда оболочка ищет команду по имени, она прежде всего проверяет, является ли это имя псевдонимом. Именно поэтому псевдоним затеняет команду (имеет более высокий приоритет) с таким же именем.

Путь для поиска — отличный пример того, как можно найти простое объяснение загадочному явлению. Оболочка не берет команды из воздуха и не находит их по волшебству. Она методично просматривает каталоги в списке, пока не найдет запрошенный исполняемый файл.

## Окружение и файлы инициализации, краткая версия

Запущенная оболочка хранит в переменных множество важной информации: путь для поиска, текущий каталог, предпочитаемый вами текстовый редактор,

<sup>1</sup> Обратите внимание, что команда `type which` записывает в стандартный вывод данные, а команда `which type` этого не делает.

настраиваемое приветствие командной строки и многое другое. Переменные запущенной командной оболочки в совокупности называются *окружением*. Когда пользователь выходит из командной оболочки, ее окружение уничтожается.

Было бы чрезвычайно утомительно при каждом запуске оболочки определять ее окружение вручную. Поэтому оно определяется в сценариях, называемых *файлами запуска* и *файлами инициализации*, которые автоматически выполняются при запуске командной оболочки. В результате определенная информация кажется «глобальной» или «известной» всем запущенным экземплярам оболочки.

Мы рассмотрим тонкости этой темы в разделе «Настройка окружения» на с. 129. А здесь расскажем только об одном файле инициализации, чтобы вы могли понять следующие несколько глав. Он находится в вашем домашнем каталоге и называется *.bashrc*. Поскольку имя начинается с точки, *ls* по умолчанию не отображает его:

```
$ ls $HOME
apple banana carrot
$ ls -a $HOME
.bashrc apple banana carrot
```

Если *\$HOME/.bashrc* не существует, создайте его с помощью текстового редактора. Команды, которые вы поместите в этот файл, будут выполняться автоматически при запуске экземпляра оболочки<sup>2</sup>, поэтому он отлично подойдет для определения переменных окружения и псевдонимов. Вот пример файла *.bashrc* (строки, начинающиеся с #, являются комментариями):

```
# Задать путь для поиска
PATH=$HOME/bin:/usr/local/bin:/usr/bin:/bin
# Задать приглашение командной строки
PS1='$ '
# Задать предпочитаемый текстовый редактор
EDITOR=emacs
# Начать в моем рабочем каталоге
cd $HOME/Work/Projects
# Определить псевдоним
alias g=grep
# Сердечно поприветствовать
echo "Welcome to Linux, friend!"
```

Изменения, которые вы вносите в *\$HOME/.bashrc*, не влияют на запущенные экземпляры оболочки. Но вы можете заставить запущенный экземпляр заново выполнить *\$HOME/.bashrc* с помощью одной из следующих команд:

<b>\$ source \$HOME/.bashrc</b>	Используется встроенная команда <i>source</i>
<b>\$ . \$HOME/.bashrc</b>	Используется точка

<sup>2</sup> Это очень упрощенное утверждение; более подробная информация представлена в табл. 6.1.

Этот процесс известен как *считывание* файла инициализации. Говоря «использовать исходный файл `.bashrc`», имеют в виду выполнение одной из предыдущих команд.



В реальной жизни не помещайте всю конфигурацию вашей оболочки в `$HOME/.bashrc`. Прочитав подробности в разделе «Настройка окружения» на с. 129, изучите файл `$HOME/.bashrc` и при необходимости переместите команды в соответствующие файлы.

## Резюме

Мы рассмотрели только немногие функции `bash` и их основные способы применения. Вы узнаете гораздо больше в следующих главах, особенно в главе 6. Сейчас ваша основная задача — усвоить следующие принципы работы оболочки:

- Оболочка выполняет важные функции.
- Оболочка интерпретирует командную строку перед запуском любых команд.
- Команды могут перенаправлять стандартные потоки ввода, вывода и ошибок.
- Кавычки и экранирование предотвращают интерпретацию командной оболочкой специальных символов.
- Оболочка находит программы, используя путь для поиска файлов.
- Вы можете изменить настройки командной оболочки, добавив команды в файл `$HOME/.bashrc`.

Чем лучше вы будете понимать разделение обязанностей между оболочкой и программами, которые она вызывает, тем больше смысла будет иметь для вас командная строка и тем лучше вы сможете предсказать, что произойдет, прежде чем вы нажмете `Enter` для запуска команды.

## Повторный запуск команд

Предположим, вы только что выполнили длинную команду со сложным конвейером, как, например, следующая команда из раздела «Обнаружение дубликатов файлов» (см. с. 32)

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
```

и вы хотите запустить ее еще раз. Только не печатайте ее заново! Вместо этого попросите оболочку обратиться к истории. За кадром остается то, как оболочка ведет запись вызываемых команд, но вы можете повторно запускать их несколькими нажатиями клавиш. Эта функция оболочки называется *историей команд*. Опытные пользователи Linux используют ее, чтобы ускорить свою работу и не тратить время впустую.

Точно так же предположим, что вы допустили ошибку при наборе предыдущей команды, например написав `jpg` вместо `jpg`:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
```

Чтобы исправить ошибку, не нажимайте клавишу **Backspace** десятки раз и не вводите все заново. Вместо этого просто измените команду. Оболочка поддерживает *редактирование командной строки* для исправления опечаток и выполнения всевозможных модификаций, так же как это делает текстовый редактор.

Эта глава покажет вам, как сэкономить время и силы при наборе текста, используя историю команд и редактирование командной строки. Как обычно, я не буду пытаться объять необъятное, а сосредоточусь на наиболее практических и полезных частях этих функций командной оболочки (если вы используете оболочку, отличную от `bash`, дополнительные примечания см. в Приложении Б).



### Научитесь печатать вслепую

Знания из этой книги послужат вам лучше, если вы сможете быстро печатать. Если вы печатаете 40 слов в минуту, а ваш не менее знающий друг набирает 120, значит, он способен работать в три раза быстрее, чем вы. Поищите в интернете «тест скорости набора текста», чтобы измерить свою скорость, затем поищите «клавиатурный тренажер» и выработайте этот полезный навык на всю жизнь. Попробуйте достичь 100 слов в минуту. Результат стоит потраченных усилий.

## Просмотр истории команд

*История команд* — это список команд, которые вы выполняли в интерактивной оболочке. Чтобы просмотреть историю, запустите команду `history`, встроенную в оболочку. Команды отображаются в хронологическом порядке с идентификационными номерами для удобства. Вывод выглядит примерно так:

```
$ history
1000 cd $HOME/Music
1001 ls
1002 mv jazz.mp3 jazzy-song.mp3
1003 play jazzy-song.mp3
:
1481 cd
1482 firefox https://google.com
1483 history
```

*Пропустим 479 строк*

*Включая только что запущенную команду*

Вывод `history` может состоять из сотен (и более) строк. Ограничьте его самыми последними командами, добавив целочисленный аргумент, который указывает количество выводимых строк:

```
$ history 3
1482 firefox https://google.com
1483 history
1484 history 3
```

*Выведем три последние команды*

Поскольку `history` записывает в стандартный вывод, вы можете обрабатывать ее вывод с помощью конвейеров. Например, просмотрите свою историю по одному экрану за раз:

```
$ history | less
$ history | sort -nr | less
```

*От самой последней команды к первой запущенной*  
*От самой первой команды к последней запущенной*

или выведите только команды, содержавшие `cd`:

```
$ history | grep -w cd
1000 cd $HOME/Music
1092 cd ..
1123 cd Finances
1375 cd Checking
1481 cd
1485 history | grep -w cd
```

Чтобы очистить историю для текущего экземпляра оболочки, используйте параметр `-c`:

```
$ history -c
```

## Повторный вызов команд из истории

Рассмотрим три экономящих время способа вызова команд из истории оболочки.

### *Перемещение курсора*

Чрезвычайно прост в освоении, но часто неудобен на практике.

### *Расширение истории*

Сложнее в освоении (честно говоря, причины этого загадочны), но может быть очень быстрым.

### *Инкрементальный поиск*

И просто, и быстро.

Каждый метод удобен для определенных ситуаций, поэтому рекомендуется изучить все три. Чем больше техник вы знаете, тем легче подберете нужную в любой ситуации.

## Перемещение курсора по истории команд

Чтобы вызвать предыдущую команду в конкретном экземпляре оболочки, нажмите клавишу со стрелкой вверх. Просто, не правда ли? Продолжайте нажимать стрелку вверх, чтобы вызывать более ранние команды в обратном хронологическом порядке. Нажмите стрелку вниз, чтобы перейти к более поздней команде. Когда вы дойдете до нужной команды, нажмите `Enter`, чтобы запустить ее.

Перемещение по истории команд является одним из двух наиболее популярных способов увеличения скорости работы в командной строке Linux (другой — это сопоставление имен файлов с шаблоном (\*), которое мы изучили в главе 2). Перемещение эффективно, если нужная команда находится недалеко в истории — не более двух или трех шагов. Но крайне утомительно добираться этим способом до команд, которые вы вызывали давно. Нажимание стрелки быстро надоедает.

Наилучший вариант использования перемещения по истории — это вызов и выполнение непосредственно предыдущей команды. На многих клавиатурах клавиша со стрелкой вверх находится рядом с клавишей **Enter**, поэтому вы можете последовательно нажимать две клавиши быстрым движением пальцев. На полноразмерной QWERTY-клавиатуре я кладу безымянный палец правой руки на стрелку вверх, а указательный палец — на **Enter**, чтобы эффективно нажимать обе клавиши (попробуйте сами).

### ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ ПРО ИСТОРИЮ КОМАНД

*Сколько команд хранится в истории?*

Максимальное число определяется значением переменной оболочки `HISTSIZE`, которую вы можете изменить:

```
$ echo $HISTSIZE
500
$ HISTSIZE=10000
```

Компьютерная память настолько дешева и доступна, что имеет смысл установить большое значение `HISTSIZE`, чтобы вы могли вызывать и повторно запускать команды из далекого прошлого (история из 10 000 команд занимает всего около 200 Кбайт на жестком диске). Можно даже хранить неограниченное количество команд, установив значение `-1`.

*Какой текст добавляется к истории команд?*

Командная оболочка добавляет в историю именно то, что вы вводите, без результатов вычислений. Если вы запустите `ls $HOME`, история будет содержать «ls \$HOME», а не «ls /home/smith» (исключение описано во врезке «Выражения, которые не отображаются в истории команд» на с. 60).

*Повторяющиеся команды добавляются в историю?*

В зависимости от значения переменной `HISTCONTROL`. По умолчанию, если эта переменная не установлена, добавляется каждая команда. Если присвоить ей значение `ignoredups`

(что я рекомендую), то повторяющиеся команды не добавляются, при условии что они идут одна за другой (другие значения см. в `man bash`):

```
$ HISTCONTROL=ignoredups
```

*Каждый экземпляр оболочки имеет свою историю команд, или она общая?*

Каждый экземпляр интерактивной оболочки имеет свою независимую от других историю команд.

*Я запустил новый экземпляр интерактивной оболочки, и у него уже есть история. Почему?*

Всякий раз, когда интерактивная оболочка завершает работу, она записывает свою историю в файл `$HOME/.bash_history` или другой, хранящийся в переменной оболочки `HISTFILE`:

```
$ echo $HISTFILE  
/home/smith/.bash_history
```

Новые экземпляры оболочки считывают при запуске историю из этого файла. Если вы используете много оболочек, каждая из них при выходе сохраняет свою историю в `$HISTFILE`, поэтому не всегда предсказуемо, какую историю загрузит новая оболочка.

Переменная `HISTFILESIZE` определяет, сколько строк истории записывается в файл. Если вы изменяете `HISTSIZE`, управляя размером истории в памяти, подумайте также об обновлении значения `HISTFILESIZE`:

```
$ echo $HISTFILESIZE  
500  
$ HISTFILESIZE=10000
```

## Расширение истории команд

Расширение истории — это функция оболочки, которая обращается к истории команд с помощью специальных выражений. Эти выражения начинаются с восклицательного знака, который пользователи Linux традиционно называют *bang*. Два восклицательных знака подряд (*bang-bang*) означают предыдущую команду:

```
$ echo Efficient Linux  
Efficient Linux
```



```
$ !!                                     Bang bang = предыдущая команда
echo Efficient Linux                   Оболочка выводит команду, которая будет запущена
Efficient Linux
```

Чтобы обратиться к последней команде, начинающейся с определенного выражения, поставьте перед ним восклицательный знак. Повторно запустить последнюю команду `grep` можно следующим образом:

```
$ !grep
grep Perl animals.txt
alpaca Intermediate Perl  2012  Schwartz, Randal
```

Чтобы обратиться к последней команде, которая содержала заданное выражение *где-либо*, а не только в начале строки, окружите его вопросительными знаками<sup>1</sup>:

```
$ !?grep?
history | grep -w cd
 1000 cd $HOME/Music
 1092 cd ..
:
```

Вы также можете найти команду из истории оболочки по ее абсолютной позиции — идентификационному номеру слева от нее в выводе истории. Например, выражение `!1203` означает «команда на позиции 1023 в истории»:

```
$ history | grep hosts
 1203 cat /etc/hosts
$ !1203                                     Команда под номером 1023
cat /etc/hosts
127.0.0.1    localhost
127.0.1.1    example.oreilly.com
:::1        example.oreilly.com
```

Отрицательное значение извлекает команду из истории по ее относительной, а не абсолютной позиции. Например, `!-3` означает «команда, которую вы выполнили три шага назад»:

```
$ history
4197 cd /tmp/junk
4198 rm *
4199 head -n2 /etc/hosts
4199 cd
4200 history
```

---

<sup>1</sup> В этой команде можно убрать завершающий вопросительный знак: `!grep`, но в некоторых случаях он необходим, например при расширении истории в стиле `sed` (см. раздел «Более эффективная замена с расширением истории»).

```
$ !-3                                Команда, запущенная три шага назад
head -n2 /etc/hosts
127.0.0.1    localhost
127.0.1.1    example.oreilly.com
```

Расширение истории быстро и удобно в использовании. Однако оно может приводить к ошибкам, если вы укажете неправильное значение. Посмотрите внимательно на предыдущий пример. Если бы вы просчитались и набрали `!-4` вместо `!-3`, то запустили бы `rm *` вместо предполагаемой команды `head` и по ошибке удалили бы все файлы в домашнем каталоге! Для снижения рисков добавьте модификатор `:p`, чтобы напечатать команду из вашей истории, но не выполнять ее:

```
$ !-3:p
head -n2 /etc/hosts                Вывести на экран, но не исполнять
```

Оболочка добавляет невыполненную команду (`head`) в историю. Таким образом, если вы видите, что все в порядке, можно быстро запустить ее с помощью *bang-bang*:

```
$ !-3:p
head -n2 /etc/hosts                Вывести на экран, не исполнять и записать в историю
$ !!                               Запустить команду
head -n2 /etc/hosts                Выведена на экран и исполнена
127.0.0.1 localhost
127.0.1.1 example.oreilly.com
```

Расширение истории иногда называют *bang-командами*, но выражения вроде `!!` и `!grep` не являются командами. Это строковые выражения, которые вы можете поместить в любом месте команды. Используйте команду `echo` для вывода значения `!!` в `stdout`, не выполняя его, и подсчитайте количество слов с помощью `wc`:

```
$ ls -l /etc | head -n3             Запустим любую команду
total 1584
drwxr-xr-x 2 root    root      4096 Jun 16 06:14 ImageMagick-6/
drwxr-xr-x 7 root    root      4096 Mar 19 2020 NetworkManager/

$ echo "!!" | wc -w                 Подсчитаем количество слов в предыдущей команде
echo "ls -l /etc | head -n3" | wc -w
6
```

Этот пример демонстрирует, что расширения истории могут использоваться не только для повторного выполнения команд. В следующем разделе вы познакомитесь с другими способами их применения. Для получения полной информации об особенностях истории команд запустите `man history`.



### Выражения, которые не отображаются в истории команд

Оболочка добавляет команды в историю без вычислений, как указано во врезке «Часто задаваемые вопросы про историю команд» на с. 56. Единственным исключением из этого правила является расширение истории. Его выражения всегда вычисляются перед добавлением в историю команд:

```
$ ls                Запустим любую команду
hello.txt
$ cd Music          Запустим другую команду
$ !-2              Используем расширение истории
ls
song.mp3
$ history           Посмотрим историю
1000 ls
1001 cd Music
1002 ls            «ls» записана в историю вместо «!-2»
1003 history
```

Это правило имеет смысл. Представьте, что вы пытаетесь понять историю команд, полную таких выражений, как `!-15` и `!-92`, которые ссылаются на другие записи в истории. Возможно, вам придется проследить путь через всю историю, чтобы понять всего лишь одну команду.

## Забудьте об ошибочном удалении файлов (спасибо расширению истории)

Случалось ли такое, что вы хотели удалить файлы с помощью шаблона, например `*.txt`, но из-за ошибки в наборе стерли не те файлы? Вот пример с пробелом после звездочки:

```
$ ls
123 a.txt    b.txt    c.txt    dont-delete-me    important-file    passwords
$ rm *.txt    ОПАСНО! Не запускайте это!
```

Популярным способом избежать этой опасности является псевдоним `rm`, запускающий команду `rm -i`, которая запрашивает подтверждение перед каждым удалением:

```
$ alias rm='rm -i'          Часто уже есть в конфигурационном файле оболочки
$ rm *.txt
/bin/rm: remove regular file 'a.txt'? y
/bin/rm: remove regular file 'b.txt'? y
/bin/rm: remove regular file 'c.txt'? y
```

В результате лишний пробел не будет фатальным, потому что подсказки от `rm -i` помогут понять, что вы удаляете не те файлы:

```
$ rm *.txt
/bin/rm: remove regular file '123'?           Что-то не так: отменить команду
```

Однако решение с псевдонимом не всегда эффективно, потому что в большинстве случаев запрос подтверждения при удалении файла не нужен и только раздражает. Кроме того, оно не сработает, если вы войдете в систему на другом компьютере с Linux, где такой псевдоним не настроен.

Есть лучший способ избежать ошибочного выполнения сопоставления имен файлов с шаблоном. Этот метод состоит из двух этапов и основан на расширении истории:

1. *Проверка.* Перед запуском `rm` запустите `ls` с нужным шаблоном, чтобы увидеть, какие файлы совпадают.

```
$ ls *.txt
a.txt b.txt c.txt
```

2. *Удаление.* Если вывод `ls` прошел проверку, запустите `rm !$`, чтобы удалить те же файлы, которые были сопоставлены с шаблоном<sup>1</sup>.

```
$ rm !$
rm *.txt
```

Расширение истории `!$` означает «последнее слово, которое вы ввели в предыдущей команде». Таким образом, `rm !$` здесь является сокращением для «удалить все, что я только что перечислил с помощью `ls`», а именно `*.txt`. Если вы случайно добавите пробел после звездочки, вывод `ls` предупредит, что что-то пошло не так:

```
$ ls *.txt
/bin/ls: cannot access '*.txt': No such file or directory
123 a.txt  b.txt  c.txt  dont-delete-me  important-file  passwords
```

Как здорово, что вы сначала запустили `ls`, а не `rm`! Теперь вы можете изменить команду, удалив лишний пробел, и продолжить без потери данных. Описанная последовательность из команд `ls` и `rm !$` — отличный прием, который заслуживает включения в инструментарий работы с Linux.

---

<sup>1</sup> Предполагается, что после выполнения `ls` без вашего ведома не были добавлены или удалены файлы, соответствующие шаблону. Не полагайтесь на этот способ при работе с каталогами, содержимое которых быстро изменяется.

Похожий метод заключается в просмотре содержимого файла с помощью команды `head` перед его удалением. Убедившись, что выбрали правильный файл, можете затем запустить `rm !$`:

```
$ head myfile.txt
(отображаются первые 10 строк файла)
$ rm !$
rm myfile.txt
```

Также в оболочке имеется расширение истории `!*`, которое соответствует всем аргументам, введенным вами в предыдущей команде, а не только последнему:

```
$ ls *.txt *.o *.log
a.txt b.txt c.txt main.o output.log parser.o
$ rm !*
rm *.txt *.o *.log
```

На практике я использую `!*` значительно реже, чем `!$`, так как есть риск, что в случае ошибки звездочка `*` будет интерпретирована как символ шаблона для имен файлов. Поэтому применение `!*` не намного безопаснее, чем набор шаблона, например `*.txt`.

## Инкрементальный поиск по истории команд

Было бы здорово, если после ввода нескольких символов команды остальные появлялись бы мгновенно — и команда готова к запуску! И это возможно. Быстрая функция оболочки, называемая *инкрементальным поиском*, похожа на интерактивные подсказки поисковых систем. В большинстве случаев инкрементальный поиск — это самый простой и быстрый способ вспомнить команды из истории, даже те, которые вы запускали давно. Я настоятельно рекомендую добавить инкрементальный поиск в свой набор инструментов Linux:

1. В приглашении оболочки нажмите `Ctrl-R` (`R` означает обратный (reverse) инкрементальный поиск).
2. Вводите любую часть ранее запущенной команды — начало, середину или конец.
3. С каждым введенным символом оболочка находит ближайшую в истории команду, которая соответствует вашему вводу.
4. Когда вы увидите нужную команду, нажмите `Enter`, чтобы запустить ее.

Предположим, вы недавно запускали команду `cd $HOME/Finances/Bank` и хотите повторить ее. Нажмите `Ctrl-R` в командной строке. Подсказка изменится, указывая на инкрементальный поиск:

```
(reverse-i-search)`':
```

Начните вводить нужную команду. Например, введите `c`:

```
(reverse-i-search)`': c
```

Оболочка отображает самую последнюю команду, содержащую символ `c`, выделяя то, что вы набрали:

```
(reverse-i-search)`': less /etc/hosts
```

Введите следующий символ, `d`:

```
(reverse-i-search)`': cd
```

Оболочка отображает самую последнюю команду, содержащую `cd`, снова выделяя то, что вы набрали:

```
(reverse-i-search)`': cd /usr/local
```

Продолжайте вводить команду, добавляя пробел и знак доллара:

```
(reverse-i-search)`': cd $
```

Командная строка приобретает следующий вид:

```
(reverse-i-search)`': cd $HOME/Finances/Bank
```

Это команда, которую вы искали, поэтому нажмите **Enter**, чтобы запустить ее. Результат получен всего лишь за пять нажатий клавиш.

Здесь я предположил, что `cd $HOME/Finances/Bank` была последней соответствующей запросу командой в истории. А что, если это не так? Если вы ввели множество команд, содержащих одну и ту же строку, то предыдущий инкрементальный поиск показал бы другое совпадение, например:

```
(reverse-i-search)`': cd $HOME/Music
```

Что теперь? Вы можете ввести больше символов, чтобы конкретизировать команду, но лучше нажмите **Ctrl-R** еще раз. Это нажатие клавиши заставляет оболочку перейти к *следующей* соответствующей шаблону поиска команде в истории:

```
(reverse-i-search)`': cd $HOME/Linux/Books
```

Продолжайте нажимать **Ctrl-R**, пока не дойдете до нужной команды:

```
(reverse-i-search)`': cd $HOME/Finances/Bank
```

и нажмите **Enter** для запуска.

Вот еще несколько трюков с инкрементальным поиском:

- Чтобы вызвать последнюю строку, которую вы искали и выполняли, начните с нажатия **Ctrl-R** два раза подряд.
- Чтобы остановить инкрементальный поиск и продолжить работу над текущей командой, нажмите **Escape**, **Ctrl-J** или любую клавишу для редактирования в командной строке (см. следующий раздел в этой главе), например клавишу со стрелкой влево или вправо.
- Чтобы выйти из инкрементального поиска и очистить командную строку, нажмите **Ctrl-G** или **Ctrl-C**.

Не жалейте времени на тренировку навыков инкрементального поиска. Вскоре вы будете находить команды с невероятной скоростью<sup>1</sup>.

## Редактирование командной строки

Есть множество причин для редактирования команды во время ее ввода или после того, как вы ее запустили:

- Чтобы исправить ошибки.
- Чтобы сконструировать команду по частям. Например, сначала набрав конец команды, затем перейдя к началу строки и набрав начало.
- Для создания новой команды на основе предыдущей из истории команд (как вы увидите в главе 8, это ключевой навык для создания сложных конвейеров).

В этом разделе я покажу вам три метода, которые позволят улучшить навыки редактирования команд и повысить скорость работы:

### *Перемещение курсора*

Самый медленный и наименее мощный метод, но простой в освоении.

### *Использование знака вставки (caret)*

Один из видов расширения истории.

### *Нажатия клавиш в стиле Emacs или Vim*

Максимально эффективное редактирование командной строки.

Изучите все три метода для большей гибкости в редактировании командной строки.

---

<sup>1</sup> При написании этой книги я часто перезапускал команды системы контроля версий, такие как `git add`, `git commit` и `git push`. Инкрементальный поиск упростил повторный запуск этих команд.

## Перемещение курсора внутри команды

Нажимайте клавиши со стрелками влево и вправо для перемещения на один символ вперед и назад по командной строке. Используйте клавиши **Backspace** или **Delete**, чтобы удалять текст. Таблица 3.1 обобщает стандартные комбинации клавиш для редактирования командной строки.

Перемещаться вперед и назад легко, но неэффективно. Этот метод подходит, когда необходимо внести небольшие и простые изменения.

**Таблица 3.1.** Клавиши перемещения курсора для простого редактирования в командной строке

Сочетание клавиш	Действие
Стрелка влево	Перейти влево на один символ
Стрелка вправо	Перейти вправо на один символ
Ctrl + стрелка влево	Перейти влево на одно слово
Ctrl + стрелка вправо	Перейти вправо на одно слово
Home	Перейти к началу командной строки
End	Перейти к концу командной строки
Backspace	Удалить один символ перед курсором
Delete	Удалить один символ под курсором

## Расширение истории с помощью знака вставки

Предположим, вы по ошибке запустили следующую команду, набрав `jpg` вместо `jpeg`:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
md5sum: '*.jpg': No such file or directory
```

Чтобы выполнить команду правильно, вы можете вызвать ее из истории, переместить курсор и исправить ошибку, но есть более быстрый способ достичь цели. Просто введите старый (неправильный) текст, новый (исправленный) текст и пару знаков вставки (^), например:

```
$ ^jpg^jpeg
```



Нажмите **Enter** — появится и сразу же будет запущена правильная команда:

```
$ ^jg^jpg
md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
:
```

Синтаксис со знаком вставки, который является видом расширения истории, означает: «В предыдущей команде вместо **jg** подставьте **jpg**». Обратите внимание, что оболочка выводит новую команду перед ее выполнением, что является стандартным поведением для расширения истории.

Этот метод изменяет только первое вхождение исходного кода (**jg**). Если ваша исходная команда содержала **jg** более одного раза, только первый экземпляр изменится на **jpg**.

#### БОЛЕЕ МОЩНАЯ ЗАМЕНА С ПОМОЩЬЮ РАСШИРЕНИЯ ИСТОРИИ

Возможно, вы знакомы с использованием команд **sed** или **ed** для преобразования исходной строки в необходимую вам:

```
s/source/target/
```

Командная оболочка поддерживает аналогичный синтаксис. Начните с выражения для расширения истории, чтобы вызвать команду, например **!!**. Затем добавьте двоеточие и закончите заменой в стиле **sed**. Например, чтобы вызвать предыдущую команду и заменить **jg** на **jpg** (только при первом вхождении) запустите:

```
$ !!:s/jg/jpg/
```

Вы можете начать с любого расширения истории, которое вам нравится. Например, **!md5sum** вызовет последнюю команду, начинающуюся с **md5sum**, и выполнит ту же замену **jg** на **jpg**:

```
$ !md5sum:s/jg/jpg/
```

Этот способ может показаться сложным, но иногда он позволяет быстрее достичь цели, чем другие методы редактирования в командной строке. Запустите **man history** для получения полной информации.

## Редактирование командной строки в стилях Emacs или Vim

Самый эффективный способ редактирования командной строки — знакомые сочетания клавиш, вдохновленные текстовыми редакторами Emacs и Vim. Если вы уже имеете опыт работы с одним из этих редакторов, то можете сразу перейти к этому стилю редактирования командной строки. Если нет, табл. 3.2 поможет вам начать работу с наиболее распространенными сочетаниями клавиш для перемещения и редактирования. Обратите внимание, что клавиша **Meta** в Emacs обычно заменяется на **Escape** (нажата и отпущена) или **Alt** (нажата и удерживается).

По умолчанию в оболочке используется редактирование в стиле Emacs, и я рекомендую его как более простой в освоении и использовании. Если вы предпочитаете редактирование в стиле Vim, выполните следующую команду (или добавьте ее в свой файл `$HOME/.bashrc` и загрузите):

```
$ set -o vi
```

Нажмите **Escape** для входа в режим редактирования команды, а затем используйте клавиши из столбца Vim в табл. 3.2. Чтобы вернуться к редактированию в стиле Emacs, запустите:

```
$ set -o emacs
```

А теперь практика, практика и еще раз практика, пока сочетания клавиш не станут вашей второй натурой. Поверьте, потраченное время быстро окупится.

**Таблица 3.2.** Сочетания клавиш для редактирования в стилях Emacs и Vim<sup>1</sup>

Действие	Emacs	Vim
Перейти вперед на один символ	Ctrl-f	h
Перейти назад на один символ	Ctrl-b	l
Перейти вперед на одно слово	Meta-f	w
Перейти назад на одно слово	Meta-b	b
Перейти в начало строки	Ctrl-a	0
Перейти в конец строки	Ctrl-e	\$
Поменять местами два символа	Ctrl-t	xp

<sup>1</sup> Действия, помеченные как *n/a*, не выполняются простым сочетанием клавиш, но могут выполняться с помощью более длинных последовательностей.

Действие	Emacs	Vim
Поменять местами два слова	Meta-t	n/a
Сделать заглавной первую букву следующего слова	Meta-c	w~
Изменить регистр следующего слова на верхний	Meta-u	n/a
Изменить регистр следующего слова на нижний	Meta-l	n/a
Изменить регистр текущего символа	n/a	~
Вставить следующий символ (в т. ч. управляющие символы)	Ctrl-v	Ctrl-v
Удалить один символ перед курсором	Ctrl-d	x
Удалить один символ после курсора	Backspace или Ctrl-h	X
Удалить одно слово перед курсором	Meta-d	dw
Удалить одно слово после курсора	Meta-Backspace или Ctrl-w	db
Удалить от курсора до начала строки	Ctrl-u	d^
Удалить от курсора до конца строки	Ctrl-k	D
Удалить строку целиком	Ctrl-e Ctrl-u	dd
Вставить (скопировать) последний удаленный текст	Ctrl-y	p
Вставить (скопировать) предпоследний удаленный текст	Meta-y	n/a
Отменить предыдущую операцию редактирования	Ctrl-_	u
Отменить все изменения	Meta-r	U
Переключение из режима вставки в командный режим	n/a	Escape
Переключение из командного режима в режим вставки	n/a	i
Прервать текущую операцию редактирования	Ctrl-g	n/a
Очистить экран	Ctrl-l	Ctrl-l

Дополнительные сведения о редактировании в стиле Emacs см. в разделе *Bindable Readline Commands*<sup>1</sup> в руководстве GNU по bash. О редактировании в стиле Vim см. документ *Readline VI Editing Mode Cheat Sheet*<sup>2</sup>.

<sup>1</sup> [https://www.gnu.org/software/bash/manual/html\\_node/Bindable-Readline-Commands.html](https://www.gnu.org/software/bash/manual/html_node/Bindable-Readline-Commands.html).

<sup>2</sup> <https://catonmat.net/ftp/bash-vi-editing-mode-cheat-sheet.pdf>.

---

## Резюме

Три техники упрощают использование Linux:

- Безопасное удаление файлов с помощью `!$`.
- Инкрементальный поиск через `Ctrl-R`.
- Редактирование командной строки в стиле Emacs.

Практикуйтесь в приемах, описанных в этой главе, и вы значительно ускорите работу с командной строкой.

## ГЛАВА 4

---

# Перемещение по файловой системе

[https://t.me/it\\_boooks/2](https://t.me/it_boooks/2)

В фильме «Приключения Бакару Банзая в восьмом измерении», классической американской комедии 1984 года, лихой главный герой произносит следующие мудрые слова в духе дзен: «Помни, куда бы ты ни пошел... ты здесь». Бакару вполне мог говорить о файловой системе Linux:

```
$ cd /usr/share/lib/etc/bin      Куда бы ты ни пошел
$ pwd
/usr/share/lib/etc/bin          ...ты здесь.
```

Другими словами, где бы вы ни находились в файловой системе Linux, в итоге все равно перейдете из текущего каталога куда-то еще (в другой каталог). Чем быстрее и эффективнее вы сможете выполнять эту навигацию, тем продуктивнее вы будете.

Приемы, описанные в этой главе, помогут вам быстрее ориентироваться в файловой системе, сокращая количество операций ввода. Они выглядят обманчиво простыми, но приносят *огромную* пользу. Эти методы делятся на две большие категории:

- Быстрое перемещение в конкретный каталог.
- Быстрое возвращение в каталог, который вы посещали ранее.

Чтобы освежить в памяти ваши знания о каталогах Linux, см. Приложение А. Если вы используете оболочку, отличную от **bash**, то найдете дополнительную информацию в Приложении Б.

## Лучшие способы перехода в нужный каталог

Если вы спросите десять опытных пользователей Linux, что самое утомительное в командной строке, не менее семи из них ответят: «Ввод длинных путей к каталогам»<sup>1</sup>. В конце концов, если ваши рабочие файлы находятся в `/home/smith/Work/Projects/Apps/Neutron-Star/src/include`, финансовые документы лежат в `/home/smith/Finances/Bank/Checking/Statements`, а ваши видео — в `/data/Arts/Video/Collection`, утомительно вводить эти пути снова и снова. В этом разделе вы узнаете, как удобнее всего перейти к заданному каталогу.

### Переход в домашний каталог

Начнем с основ. Независимо от того, где вы находитесь в файловой системе, вы можете вернуться в свой домашний каталог, запустив `cd` без аргументов:

```
$ pwd
/etc
$ cd
$ pwd
/home/smith
```

*Вы начали где-то в файловой системе  
Запустите cd без аргументов...  
...и вы снова в домашнем каталоге*

Чтобы перейти к подкаталогам в вашем домашнем каталоге из любой точки файловой системы, обратитесь к домашнему каталогу с помощью сокращения. Одним из сокращений является переменная оболочки `HOME`:

```
$ cd $HOME/Work
```

Еще один способ — это символ тильда (`~`):

```
$ cd ~/Work
```

И `$HOME`, и `~` являются выражениями, вычисляемыми оболочкой. Это можно проверить с помощью стандартного вывода:

```
$ echo $HOME ~
/home/smith /home/smith
```

Тильда также может привести к домашнему каталогу другого пользователя, если помещена непосредственно перед его именем:

```
$ echo ~jones
/home/jones
```

---

<sup>1</sup> Я не проводил опрос, но, безусловно, это вполне реалистичный вариант.

## Перемещайтесь быстрее с автозавершением командной строки

Когда вы вводите команду `cd`, не печатайте путь полностью. Вместо этого нажмите клавишу **Tab**, чтобы автоматически подставлять имена каталогов. В качестве тренировки посетите каталог, содержащий подкаталоги, например `/usr`:

```
$ cd /usr
$ ls
bin games include lib local sbin share src
```

Предположим, вы хотите перейти в подкаталог `share`. Введите `sha` и один раз нажмите **Tab**:

```
$ cd sha<Tab>
```

Командная оболочка дополнит имя каталога за вас:

```
$ cd share/
```

Это полезное сокращение называется *автозавершением командной строки* (*tab completion*). Оно срабатывает сразу, когда введенный вами текст приходит в соответствие с именем одного из каталогов. Когда текст соответствует именам нескольких каталогов, вашей оболочке требуется дополнительная информация. Предположим, вы набрали только `s` и нажали **Tab**:

```
$ cd s<Tab>
```

Оболочка не может завершить имя каталога `share` (пока что), потому что имена других каталогов тоже начинаются с `s`: `sbin` и `src`. Нажмите **Tab** второй раз, и оболочка выведет все возможные варианты завершения, чтобы помочь вам:

```
$ cd s<Tab><Tab>
sbin/ share/ src/
```

Теперь она будет ожидать вашего следующего действия. Чтобы устранить двусмысленность, введите символ `h` и еще раз нажмите клавишу **Tab**:

```
$ cd share/
```

В общем, нажимайте **Tab** один раз, когда остался единственно возможный вариант, и дважды, чтобы вывести все возможные варианты. Чем больше символов вы вводите, тем меньше двусмысленности.

Автозавершение с помощью клавиши **Tab** отлично подходит для ускорения навигации. Вместо того чтобы вводить длинный путь, такой как `/home/smith/`

*Projects/Web/src/include*, введите столько символов, сколько считаете нужным, и продолжайте нажимать клавишу **Tab**. Попрактикуйтесь, и вы быстро освоитесь.



### Автозавершение зависит от программы

Автозавершение командной строки подходит не только для команд `cd`. Оно работает для большинства команд, хотя его поведение может отличаться. С командой `cd` клавиша **Tab** завершает имена каталогов. Для команд, работающих с файлами, таких как `cat`, `grep` и `sort`, автозавершение дополняет имена файлов. Если используется команда `ssh` (*secure shell*), **Tab** дополняет имена хостов. Для команды `chown` (изменить владельца файла) она дополняет имена пользователей. Вы можете даже создать свои собственные правила завершения для увеличения скорости работы, как мы увидим далее в листинге 4.1. Также вы можете запустить `man bash` и прочитать раздел *Programmable completion*.

## Переход к часто посещаемым каталогам с использованием псевдонимов или переменных

Если вы часто посещаете удаленный от корня файловой системы каталог, такой, например, как */home/smith/Work/Projects/Web/src/include*, создайте псевдоним, выполняющий операцию `cd`:

```
# В файле конфигурации оболочки:  
alias work="cd $HOME/Work/Projects/Web/src/include"
```

Теперь запустите псевдоним, чтобы добраться до нужного каталога:

```
$ work  
$ pwd  
/home/smith/Work/Projects/Web/src/include
```

В качестве альтернативы можно создать переменную для хранения пути к нужному каталогу:

```
$ work=$HOME/Work/Projects/Web/src/include  
$ cd $work  
$ pwd  
/home/smith/Work/Projects/Web/src/include  
$ ls $work/css  
main.css mobile.css
```

*Использование переменной другими способами*





### Редактируйте часто используемые файлы с помощью псевдонимов

Иногда причиной частого посещения каталога является редактирование определенного файла. Тогда есть смысл определить псевдоним для редактирования этого файла. Определение псевдонима `rcedit` в примере ниже позволит вам редактировать `$HOME/.bashrc` независимо от того, где вы находитесь в файловой системе. Запуск команды `cd` не требуется:

```
# Поместите в файл конфигурации оболочки и примените его:
alias rcedit='$EDITOR $HOME/.bashrc'
```

Если вы регулярно посещаете множество каталогов с длинными путями, можете создать псевдонимы или переменные для каждого из них. Однако этот подход имеет недостатки:

- Трудно запомнить все эти псевдонимы и переменные.
- Вы можете случайно создать псевдоним с тем же именем, что и у существующей команды, и это вызовет конфликт.

В качестве альтернативы можно создать функцию оболочки, подобную `qcd` (*quick cd*) в листинге 4.1. Эта функция принимает строковый ключ в качестве аргумента, например `work` или `recipes`, и запускает команду `cd` для выбранного пути к каталогу.

#### Листинг 4.1. Функция для перехода в отдаленные каталоги

```
# Определение функции qcd
qcd () {
    # Принимает 1 аргумент, который является строковым ключом
    case "$1" in
        work)
            cd $HOME/Work/Projects/Web/src/include
            ;;
        recipes)
            cd $HOME/Family/Cooking/Recipes
            ;;
        video)
            cd /data/Arts/Video/Collection
            ;;
        beatles)
            cd $HOME/Music/mp3/Artists/B/Beatles
            ;;
        *)
            # Введенный аргумент не совпал ни с одним из ключей
            echo "qcd: unknown key '$1'"
            return 1
            ;;
    esac
}
```

```
esac
# Вывести на экран имя текущего каталога, чтобы указать, где вы находитесь
pwd
}
# Настройка автозавершения
complete -W "work recipes video beatles" qcd
```

Сохраните функцию в файле конфигурации оболочки, таком как *\$HOME/.bashrc* (см. раздел «Окружения и файлы инициализации, краткая версия» на с. 50), обновите экземпляр оболочки — и она готова к запуску. Введите *qcd*, а затем один из поддерживаемых ключей, чтобы быстро перейти в соответствующий каталог:

```
$ qcd beatles
/home/smith/Music/mp3/Artists/B/Beatles
```

В качестве бонуса последняя строка сценария запускает встроенную в оболочку команду *complete*, которая настраивает автозавершение для команды *qcd*. Поэтому сценарий завершается четырьмя поддерживаемыми ключами. Теперь вам не нужно запоминать аргументы *qcd*! Просто введите *qcd*, пробел, дважды нажмите клавишу *Tab* — и оболочка напечатает все доступные аргументы. Тогда вы сможете ввести любой из них обычным способом:

```
$ qcd <Tab><Tab>
beatles recipes video work
$ qcd v<Tab><Enter>           Завершает 'v' до 'video'
/data/Arts/Video/Collection
```

## Уменьшите пространство поиска с помощью CDPATH

Функция *qcd* обрабатывает только указанные вами каталоги. Но командная оболочка предоставляет более общее решение для создания быстрых функций по переходу в нужный каталог, которое лишено этого недостатка. *Путь поиска команды cd (cd search path)* в свое время изменил мой подход к навигации по файловой системе Linux.

Предположим, у вас есть важный подкаталог */home/smith/Family/Memories/Photos*, который вы часто посещаете. Когда вы перемещаетесь по файловой системе, чтобы попасть в каталог *Photos*, вам может потребоваться ввести длинный путь, например:

```
$ cd ~/Family/Memories/Photos
```

Было бы здорово, если бы вы могли сократить этот путь до *Photos*, независимо от того, где вы находитесь в файловой системе.

```
$ cd Photos
```

Обычно эта команда не работает:

```
bash: cd: Photos: No such file or directory      Нет такого файла или каталога
```

Сработает она, если вы случайно оказались в исходном каталоге (`~/Family/Memories`) или в каком-либо другом каталоге с подкаталогом *Photos*. Но путем несложных манипуляций вы можете указать команде `cd` искать подкаталог *Photos* не только в текущем каталоге. Поиск работает практически мгновенно и только в указанных вами родительских каталогах. Например, вы можете указать `cd` искать в `$HOME/Family/Memories` в дополнение к текущему каталогу. После этого `cd Photos` завершится успешно из любой точки файловой системы:

```
$ pwd
/etc
$ cd Photos
/home/smith/Family/Memories/Photos
```

Путь поиска команды `cd` работает так же, как путь `$PATH`, но вместо команд он находит подкаталоги. Настройте его с помощью переменной оболочки `CDPATH`, которая имеет тот же формат, что и `PATH` — список каталогов, разделенных двоеточиями. Например, ваша переменная `CDPATH` состоит из четырех каталогов:

```
$HOME:$HOME/Projects:$HOME/Family/Memories:/usr/local
```

и вы печатаете:

```
$ cd Photos
```

тогда команда `cd` проверит наличие следующих каталогов по порядку:

1. Подкаталог *Photos* в текущем каталоге
2. `$HOME/Photos`
3. `$HOME/Projects/Photos`
4. `$HOME/Family/Memories/Photos`
5. `/usr/local/Photos`

В нашем случае `cd` с четвертой попытки успешно меняет каталог на `$HOME/Family/Memories/Photos`. Если два каталога в `$CDPATH` имеют подкаталог с именем *Photos*, переход произойдет в тот из них, который стоит раньше в списке.



Обычно успешно выполненная команда `cd` не выводит никаких результатов. Однако, когда `cd` находит каталог, используя `CDPATH`, она выводит абсолютный путь, чтобы сообщить о новом текущем каталоге:

```
$ CDPATH=/usr          Установить CDPATH
$ cd /tmp              Нет вывода: CDPATH не использовался
$ cd bin               cd использовала CDPATH...
/usr/bin               ...и вывела путь к новому рабочему каталогу
```

Заполните `CDPATН` наиболее важными и часто используемыми родительскими каталогами, и вы сможете перейти в любой их подкаталог из любой точки файловой системы, не вводя большую часть пути. Поверьте мне, это *потрясающе*, и следующий пример докажет это.

## Организируйте свой домашний каталог для быстрой навигации

Давайте воспользуемся переменной `CDPATН`, чтобы упростить навигацию по домашнему каталогу. После короткой настройки каталоги в вашей домашней директории станут легкодоступными. Независимо от того, где вы находитесь в файловой системе, потребуется минимальный набор текста. Этот метод особенно эффективен, если ваш домашний каталог хорошо организован и содержит не более двух уровней подкаталогов. На рис. 4.1 показан пример хорошо организованного домашнего каталога.

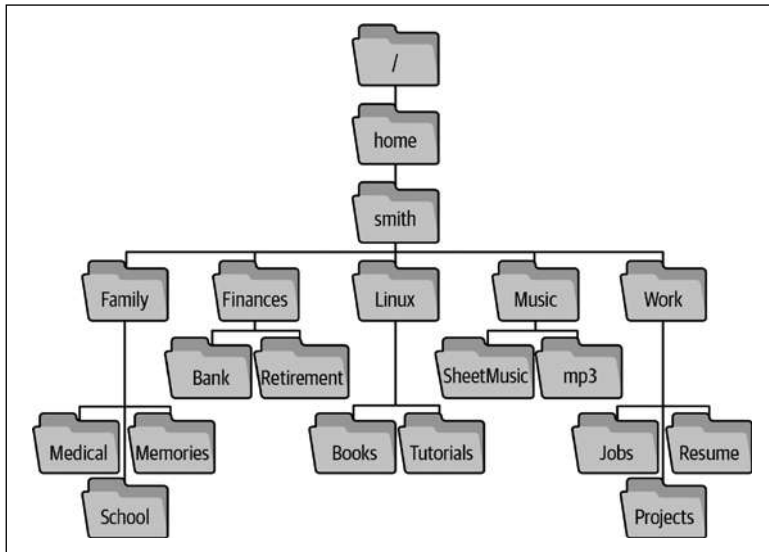


Рис. 4.1. Два уровня подкаталогов в каталоге `/home/smith`

Хитрость заключается в том, чтобы ваш `CDPATН` включал следующее:

1. `$HOME`.
2. Ваш набор подкаталогов `$HOME`.
3. Относительный путь к родительскому каталогу, обозначенный двумя точками `(..)`.

Включив `$HOME` в `CDPATH`, вы можете сразу перейти к любому из его подкаталогов (*Family*, *Finances*, *Linux*, *Music* и *Work*) из любого места в файловой системе, не вводя начальный путь:

```
$ pwd
/etc                                     Начните за пределами вашего домашнего каталога
$ cd Work
/home/smith/Work
$ cd Family/School                     Вы перешли на один уровень ниже $HOME
/home/smith/Family/School
```

Включив подкаталоги `$HOME` в `CDPATH`, вы сможете сразу перейти в их подкаталоги:

```
$ pwd
/etc                                     Где-то за пределами вашего домашнего каталога
$ cd School
/home/smith/Family/School             Вы перешли на два уровня ниже $HOME
```

Все каталоги в вашей переменной `CDPATH` до сих пор являлись абсолютными путями в `$HOME` и его подкаталогах. Однако, включив относительный путь `..`, вы поменяете поведение команды `cd`. Независимо от того, где вы находитесь в файловой системе, вы сможете перейти к любому каталогу того же уровня по его имени. Например, если вы находитесь в `/usr/bin` и хотите перейти в `/usr/lib`:

```
$ pwd
/usr/bin                                Ваш текущий каталог
$ ls ..
bin include lib src                    Ваши одноуровневые каталоги
$ cd lib
/usr/lib                                Вы перешли в другой одноуровневый каталог
```

Или, если вы программист, работающий с подкаталогами `src`, `include` и `docs`:

```
$ pwd
/usr/src/myproject
$ ls
docs include src
```

вы можете быстро перемещаться между подкаталогами:

```
$ cd docs                               Измените ваш текущий каталог
$ cd include
/usr/src/myproject/include              Вы перешли в другой одноуровневый каталог
$ cd src
/usr/src/myproject/src                  Снова перешли
```

`CDPATH` для дерева каталогов на рис. 4.1 может содержать шесть элементов: ваш домашний каталог, четыре его подкаталога и относительный путь к родительскому каталогу:

*# Поместите в файл настроек оболочки и примените его*

```
export CDPATH=$HOME:$HOME/Work:$HOME/Family:$HOME/Linux:$HOME/Music:..
```

После применения файла конфигурации вы сможете перейти к важным каталогам, вводя только их короткие имена. Ура!

Этот метод работает лучше всего, если все подкаталоги в каталогах, указанных в CDPATH, имеют уникальные имена. Если у вас есть повторяющиеся имена, например *\$HOME/Music* и *\$HOME/Linux/Music*, то команда `cd Music` всегда будет проверять *\$HOME* перед *\$HOME/Linux* и, следовательно, не найдет *\$HOME/Linux/Music*.

Чтобы проверить наличие повторяющихся имен подкаталогов на первых двух уровнях *\$HOME*, попробуйте однострочник, который перечисляет все каталоги и подкаталоги *\$HOME*, изолирует имена подкаталогов с помощью `cut`, сортирует список и подсчитывает вхождения с помощью `uniq`:

```
$ cd
$ ls -d */ && (ls -d */* | cut -d/ -f2-) | sort | uniq -c | sort -nr | less
```

Вы могли узнать команду из раздела «Обнаружение дубликатов файлов» на с. 32. Если в выходных данных отображается число больше 1, у вас есть дубликаты. Эта команда включает в себя несколько функций, которые мы еще не рассматривали. Вы познакомитесь с двойным амперсандом (`&&`) в разделе «Способ 1: Условные списки» на с. 135, а со скобками — в разделе «Способ 10: Явные подоболочки» на с. 159.

## Лучшие способы вернуться в каталог

Вы только что научились эффективно переходить в нужные каталоги. Теперь я покажу вам, как быстро снова перейти в каталог, который вы уже посещали.

### Переключение между двумя каталогами с помощью «`cd -`»

Предположим, вы работали в отдаленном вложенном каталоге и запустили `cd`, чтобы перейти куда-то еще:

```
$ pwd
/home/smith/Finances/Bank/Checking/Statements
$ cd /etc
```

но затем захотели вернуться в каталог *Statements*, где только что были. Не вводите повторно длинный путь к каталогу. Просто запустите `cd` с дефисом в качестве аргумента:

```
$ cd -  
/home/smith/Finances/Bank/Checking/Statements
```

Эта команда возвращает ваш экземпляр оболочки в предыдущий каталог и выводит абсолютный путь, чтобы вы знали, где находитесь.

Чтобы перемещаться туда-обратно между парой каталогов, несколько раз запустите `cd`. Это экономит время, когда вы сосредоточены на работе в двух каталогах в одном экземпляре оболочки. Однако есть одна загвоздка: оболочка запоминает только один предыдущий каталог за раз. Например, если вы переключаетесь между */usr/local/bin* и */etc*:

```
$ pwd  
/usr/local/bin  
$ cd /etc  
$ cd -  
/usr/local/bin  
$ cd -  
/etc
```

*Экземпляр оболочки помнит /usr/local/bin*  
*Экземпляр оболочки помнит /etc*  
*Экземпляр оболочки помнит /usr/local/bin*

и вы запускаете `cd` без аргументов, чтобы перейти в ваш домашний каталог:

```
$ cd  
Экземпляр оболочки помнит /etc
```

оболочка теперь забыла */usr/local/bin* как предыдущий каталог:

```
$ cd -  
/etc  
$ cd -  
/home/smith
```

*Экземпляр оболочки помнит ваш домашний каталог*  
*Экземпляр оболочки помнит /etc*

Следующий метод позволяет обойти это ограничение.

## Переключение между несколькими каталогами с помощью `pushd` и `popd`

Команда `cd` позволяет переключаться между двумя каталогами, но что, если вам нужно отслеживать три или более каталогов? Предположим, вы создаете локальный веб-сайт на своем компьютере с Linux. Эта задача часто подразумевает использование четырех или более каталогов:

- Расположение активных, развернутых веб-страниц, таких как `/var/www/html`.
- Каталог конфигурации веб-сервера, например `/etc/apache2`.
- Расположение SSL-сертификатов, например `/etc/ssl/certs`.
- Ваш рабочий каталог, например `~/Work/Projects/Web/src`.

Поверьте мне, утомительно постоянно вводить:

```
$ cd ~/Work/Projects/Web/src
$ cd /var/www/html
$ cd /etc/apache2
$ cd ~/Work/Projects/Web/src
$ cd /etc/ssl/certs
```

Если у вас дисплей с большой диагональю, вы можете упростить задачу, открыв отдельное окно оболочки для каждого каталога. Но если вы работаете в одном экземпляре оболочки (скажем, через соединение SSH), воспользуйтесь функцией оболочки, называемой *стеком каталогов*. Он позволяет легко перемещаться между несколькими каталогами, используя встроенные команды оболочки `pushd`, `popd` и `dirs`. На изучение этой функции вы потратите, может быть, 15 минут, а пользу от увеличения скорости работы будете получать всю жизнь<sup>1</sup>.

*Стек каталогов* — это список каталогов, которые вы посетили в текущем экземпляре оболочки и решили отслеживать. Вы управляете стеком, выполняя две операции, называемые *вталкиванием* (*pushing*) и *извлечением* (*popping*). Вталкивание каталога добавляет его в начало списка, который называется *вершиной* стека. Извлечение удаляет верхний каталог из стека<sup>2</sup>. Первоначально стек содержит только ваш текущий каталог, но вы можете добавлять (вталкивать) и удалять (извлекать) каталоги и быстро переходить между ними.



Каждый экземпляр оболочки поддерживает свой собственный стек каталогов.

<sup>1</sup> Альтернативой является открытие нескольких виртуальных дисплеев с помощью терминальных программ, таких как `screen` и `tmux`, которые называются терминальными мультитекстами. Их изучение требует больше усилий, чем изучение стека каталогов, но на них тоже стоит обратить внимание.

<sup>2</sup> Если вы знакомы со стеками из информатики, стек каталогов — это такой же стек, только для имен каталогов.



Я начну с основных операций (вталкивание, извлечение, просмотр), а затем перейду к другим вещам.

### Поместить каталог в стек

Команда `pushd` (сокращение от *push directory*) выполняет следующие действия:

1. Добавляет заданный каталог в вершину стека.
2. Выполняет `cd` в этот каталог.
3. Выводит на экран стек сверху вниз для справки.

Я создам стек из четырех каталогов, добавляя их по одному:

```
$ pwd
/home/smith/Work/Projects/Web/src
$ pushd /var/www/html
/var/www/html ~/Work/Projects/Web/src
$ pushd /etc/apache2
/etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ pushd /etc/ssl/certs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ pwd
/etc/ssl/certs
```

Оболочка печатает стек после каждой команды `pushd`. Текущий каталог — левый (верхний).

### Просмотр стека каталогов

Выведите на экран стек каталогов текущего экземпляра оболочки с помощью команды `dirs`. Она не изменяет стек:

```
$ dirs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
```

Если вы предпочитаете печатать стек сверху вниз, используйте параметр `-p`:

```
$ dirs -p
/etc/ssl/certs
/etc/apache2
/var/www/html
~/Work/Projects/Web/src
```

Вы можете передать вывод команде `nl` для нумерации строк с нуля:

```
$ dirs -p | nl -v0
0 /etc/ssl/certs
1 /etc/apache2
2 /var/www/html
3 ~/Work/Projects/Web/src
```

Это можно сделать еще проще, запустив `dirs -v`. Этот параметр выводит на экран стек с пронумерованными строками:

```
$ dirs -v
0 /etc/ssl/certs
1 /etc/apache2
2 /var/www/html
3 ~/Work/Projects/Web/src
```

Если вы предпочитаете такой формат, подумайте о создании псевдонима:

```
# Поместите в файл конфигурации оболочки и примените его
alias dirs='dirs -v'
```

## Извлечь каталог из стека

Команда `popd` (*pop directory*) противоположна команде `pushd`. Она делает следующее:

1. Удаляет один каталог из вершины стека.
2. Выполняет команду `cd` в новый верхний каталог.
3. Выводит на экран стек сверху вниз для справки.

Например, если в вашем стеке четыре каталога:

```
$ dirs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
```

Несколько раз запустим команду `popd`, которая будет проходить по этим каталогам сверху вниз:

```
$ popd
/etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ popd
/var/www/html ~/Work/Projects/Web/src
$ popd
~/Work/Projects/Web/src
$ popd
bash: popd: directory stack empty
$ pwd
~/Work/Projects/Web/src
```



Команды `pushd` и `popd` настолько экономят время, что я рекомендую создавать двухбуквенные псевдонимы, которые набираются так же быстро, как `cd`:

```
# Поместите в файл конфигурации оболочки и примените его
alias gd=pushd
alias pd=popd
```

### Поменять местами каталоги в стеке

Теперь, когда вы умеете создавать и очищать стек каталогов, давайте сосредоточимся на практических примерах его использования. `pushd` без аргументов меняет местами два верхних каталога в стеке и переходит к новому верхнему каталогу. Давайте несколько раз переместимся между `/etc/apache2` и вашим рабочим каталогом, запустив `pushd`. Посмотрите, как третий каталог `/var/www/html` остается там же, когда первые два меняются местами:

```
$ dirs
/etc/apache2 ~/Work/Projects/Web/src /var/www/html
$ pushd
~/Work/Projects/Web/src /etc/apache2 /var/www/html
$ pushd
/etc/apache2 ~/Work/Projects/Web/src /var/www/html
$ pushd
~/Work/Projects/Web/src /etc/apache2 /var/www/html
```

Обратите внимание, что `pushd` при переключении между двумя каталогами аналогичен команде `cd`, но не ограничивается запоминанием только одного каталога.

### Превратить ошибочный `cd` в `pushd`

Предположим, вы перемещаетесь между несколькими каталогами с помощью `pushd`, а потом случайно запускаете `cd` и теряете каталог:

```
$ dirs
~/Work/Projects/Web/src /var/www/html /etc/apache2
$ cd /etc/ssl/certs
$ dirs
/etc/ssl/certs /var/www/html /etc/apache2
```

Команда `cd` заменила каталог `~/Work/Projects/Web/src` в стеке на `/etc/ssl/certs`. Но не волнуйтесь. Вы можете добавить отсутствующий каталог обратно в стек, не вводя его длинный путь. Просто запустите `pushd` дважды, один раз с дефисом и один раз без:

```
$ pushd -
~/Work/Projects/Web/src /etc/ssl/certs /var/www/html /etc/apache2
$ pushd
/etc/ssl/certs ~/Work/Projects/Web/src /var/www/html /etc/apache2
```

Давайте разберем, почему это работает:

- Первая команда `pushd` возвращает вас в предыдущий каталог, посещенный в этом экземпляре оболочки `~/Work/Projects/Web/src`, и помещает его в стек. `pushd`, как и `cd`, принимает дефис в качестве аргумента, означающего «вернуться в мой предыдущий каталог».

- Вторая команда `pushd` меняет местами два верхних каталога, возвращая вас в `/etc/ssl/certs`. В результате вы восстановили `~/Work/Projects/Web/src` на второй позиции в стеке, где он был бы, если бы вы не допустили ошибки.

Команда «ой, я забыл `pushd`» достаточно полезна, чтобы дать ей псевдоним. Я называю ее `slurp` (потому что она «*slurps back*» потерянный мной каталог):

```
# Поместите в файл конфигурации оболочки и примените его
alias slurp='pushd - && pushd'
```

## Углубиться в стек

Что делать, если вы хотите перейти в другие каталоги в стеке, помимо двух верхних? `pushd` и `popd` принимают положительный или отрицательный целочисленный аргумент для дальнейшей работы со стеком. Команда

```
$ pushd +N
```

сдвигает `N` каталогов с вершины стека вниз, а затем переходит к новому верхнему каталогу. Отрицательный аргумент (`-N`) сдвигает каталоги в противоположном направлении, снизу вверх, перед выполнением `cd`:

```
$ dirs
/etc/ssl/certs ~/Work/Projects/Web/src /var/www/html /etc/apache2
$ pushd +1
~/Work/Projects/Web/src /var/www/html /etc/apache2 /etc/ssl/certs
$ pushd +2
/etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src /var/www/html
```

Таким образом, вы можете перейти к любому другому каталогу в стеке с помощью простой команды. Однако если ваш стек длинный, трудно определить числовую позицию нужного каталога. Поэтому выведите числовую позицию каждого каталога с помощью `dirs -v`, как вы делали в разделе «Просмотр стека каталогов» на с. 82:

```
$ dirs -v
0 /etc/apache2
1 /etc/ssl/certs
2 ~/Work/Projects/Web/src
3 /var/www/html
```

Чтобы переместить `/var/www/html` на вершину стека (и сделать его текущим каталогом), запустите `pushd +3`.

---

<sup>1</sup> «Выплывает». — *Примеч. пер.*

Чтобы перейти к каталогу в нижней части стека, запустите `pushd -0`:

```
$ dirs
/etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src /var/www/html
$ pushd -0
/var/www/html /etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src
```

Вы также можете удалить каталоги из стека, используя `popd` с числовым аргументом. Команда

```
$ popd +N
```

удаляет каталог, расположенный на позиции N, из стека, считая сверху вниз. Отрицательный аргумент (-N) означает отсчет снизу стека. Подсчет начинается с нуля, поэтому `popd +1` удаляет второй каталог сверху:

```
$ dirs
/var/www/html /etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src
$ popd +1
/var/www/html /etc/ssl/certs ~/Work/Projects/Web/src
$ popd +2
/var/www/html /etc/ssl/certs
```

## Резюме

Все приемы, описанные в этой главе, легко освоить, если немного попрактиковаться, и они сэкономят вам много времени и усилий. Вот техники, которые будут особенно полезны:

- `CDPATH` для быстрой навигации.
- `pushd` и `popd` для быстрого возврата.
- Исправление случайно запущенной команды `cd`.

## ЧАСТЬ 2

---

# Продвинутые навыки

Теперь, когда вы понимаете основы команд, конвейеров, оболочки и навигации по файловой системе, пришло время сделать шаг вперед. В следующих пяти главах я познакомлю вас с множеством новых программ для Linux и некоторыми важными концепциями оболочки. Вы научитесь применять их для создания сложных команд и решения реальных задач на компьютере с Linux.

## ГЛАВА 5

---

# Расширяем ваш инструментарий

Системы, основанные на Linux, поставляются с тысячами встроенных в командную строку команд. Опытные пользователи обычно полагаются на меньший набор — основной инструментарий, к которому они возвращаются снова и снова. Глава 1 добавила в ваш набор шесть весьма полезных команд, а теперь я познакомлю вас с еще примерно дюжиной. Я кратко опишу каждую команду и покажу несколько примеров ее использования (чтобы увидеть все доступные параметры, просмотрите справочную страницу соответствующей команды). Я также познакомлю вас с двумя мощными командами, которые труднее освоить, но они того стоят: `awk` и `sed`. В целом, команды в этой главе служат четырем общим практическим задачам конвейеров и других сложных команд.

### *Создание текста*

Вывод дат, времени, последовательностей цифр и букв, путей к файлам, повторяющихся строк и другого текста для быстрого запуска конвейеров.

### *Извлечение текста*

Извлечение какой-либо части текстового файла с помощью комбинации `grep`, `cut`, `head` или `tail` с функцией `awk`.

### *Объединение текста*

Объединение текста из разных файлов с помощью `cat` и `tac` или `echo` и `paste`. Вы также можете чередовать текстовые файлы с помощью `paste` и `diff`.

### *Преобразование текста*

Преобразование текста с помощью простых команд, таких как `tr` и `rev`, или более мощных команд, таких как `awk` и `sed`.

Эта глава представляет собой краткий обзор. В последующих главах вы познакомитесь с практическим использованием этих команд.

## Создание текста

Каждый конвейер начинается с простой команды, которая выводит данные в стандартный вывод. Иногда это команда вроде `grep` или `cut`, которая извлекает определенные данные из файла:

```
$ cut -d: -f1 /etc/passwd | sort
```

*Вывести все имена пользователей  
и отсортировать их*

Или команда `cat` для передачи полного содержимого нескольких файлов другим командам:

```
$ cat *.txt | wc -l
```

*Общее количество строк*

Исходный текст может поступать в конвейер и из других источников. Вы уже знаете команду `ls`, которая печатает имена файлов и каталогов и связанную с ними информацию. Давайте взглянем на некоторые другие команды и методы создания текста:

`date`

Вывод даты и времени в различных форматах.

`seq`

Вывод последовательности чисел.

*Расширение команд с помощью фигурных скобок*

Функция оболочки, которая выводит последовательность цифр или символов.

`find`

Выводит путь к файлу.

`yes`

Повторно печатает одну и ту же строку.

## Команда `date`

Команда `date` выводит текущую дату и/или время в различных форматах:

```
$ date
```

*Формат по умолчанию*

```
Mon Jun 28 16:57:33 EDT 2021
```

```
$ date +%Y-%m-%d
```

*Формат год-месяц-день*

```
2021-06-28
```



```
$ date +%H:%M:%S          Формат часы:минуты:секунды
16:57:33
```

Чтобы управлять форматом вывода, укажите аргумент, начинающийся со знака плюс, за которым следует любой текст. Текст может содержать специальные выражения, начинающиеся со знака процента, например %Y для четырехзначного значения текущего года и %H для текущего часа в 24-часовом формате. Полный список выражений находится на справочной странице команды `date`.

```
$ date +»I cannot believe it's already %A!«    День недели
I cannot believe it's already Tuesday!
```

## Команда `seq`

Команда `seq` печатает последовательность чисел из диапазона. Укажите два аргумента, нижнее и верхнее значения диапазона, и `seq` напечатает весь диапазон:

```
$ seq 1 5          Выводит все целые числа от 1 до 5 включительно
1
2
3
4
5
```

Если вы задаете три аргумента, первый и третий определяют диапазон, а среднее число — это шаг:

```
$ seq 1 2 10       Увеличение на 2 вместо 1
1
3
5
7
9
```

Используйте отрицательный шаг, например `-1`, для создания уменьшающейся последовательности:

```
$ seq 3 -1 0
3
2
1
0
```

или дробный шаг для получения чисел с плавающей запятой:

```
$ seq 1.1 0.1 2          Увеличение на 0,1
1.1
1.2
1.3
:
2.0
```

По умолчанию значения разделяются символом новой строки, но вы можете изменить разделитель с помощью параметра `-s`, после которого можно указать любой символ на ваш выбор:

```
$ seq -s/ 1 5             Разделение значений с помощью косой черты
1/2/3/4/5
```

Параметр `-w` приводит все значения к одинаковой ширине (в символах), добавляя ведущие нули по мере необходимости:

```
$ seq -w 8 10
08
09
10
```

`seq` может выводить числа во многих других форматах (см. справочную страницу), но мои примеры показывают наиболее распространенные варианты использования.

## Расширение команд с помощью фигурных скобок

Командная оболочка предоставляет собственный способ вывода последовательности чисел, известный как *расширение фигурных скобок* (*brace expansion*). Начните с левой фигурной скобки, добавьте два целых числа, разделенных двумя точками, и закончите правой фигурной скобкой:

```
$ echo {1..10}            Вперед, начиная с 1
1 2 3 4 5 6 7 8 9 10
$ echo {10..1}            Назад, начиная с 10
10 9 8 7 6 5 4 3 2 1
$ echo {01..10}           С ведущими нулями (для равной ширины)
01 02 03 04 05 06 07 08 09 10
```

В более общем случае выражение оболочки `{x..y..z}` генерирует значения от `x` до `y` с шагом `z`:

```
$ echo {1..1000..100}          Приращение сотнями, начиная с 1
1 101 201 301 401 501 601 701 801 901
$ echo {1000..1..100}          Уменьшение сотнями, начиная с 1000
1000 900 800 700 600 500 400 300 200 100
$ echo {01..1000..100}         С ведущими нулями
0001 0101 0201 0301 0401 0501 0601 0701 0801 0901
```



### Фигурные скобки vs квадратные

Квадратные скобки — это оператор сопоставления имен файлов с шаблоном (см. главу 2). Расширение фигурных скобок никак не связано с именами файлов. Это просто вычисление последовательности строковых значений. Вы можете использовать расширение фигурных скобок для *вывода* имен файлов, но сопоставления с шаблоном при этом не происходит:

```
$ ls
file1 file2 file4
$ ls file[2-4]          Соответствует существующим именам
file2 file4             файлов
$ ls file{2..4}         Вычисляется в: file2 file3 file4
ls: cannot access 'file3': No such file or directory
file2 file4
```

Расширение с помощью фигурных скобок также может создавать последовательности букв, которые не может вывести команда `seq`:

```
$ echo {A..Z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Расширение с помощью фигурных скобок всегда записывает вывод в виде одной строки, разделенной пробелами. Изменить это можно, направив вывод другим командам, таким как `tr` (см. подраздел «Команда `tr`» на с. 107):

```
$ echo {A..Z} | tr -d ' '      Удалить пробелы
ABCDEFGHIJKLMNOPQRSTUVWXYZ
$ echo {A..Z} | tr ' ' '\n'   Заменить пробелы на символ новой
A                               строки
B
C
:
Z
```

Создайте псевдоним, который печатает *n*-ю букву английского алфавита:

```
$ alias nth="echo {A..Z} | tr -d ' ' | cut -c"
$ nth 10
J
```

## Команда `find`

Команда `find` рекурсивно выводит список файлов в каталоге, спускаясь по подкаталогам и выводя полные пути<sup>1</sup>. Результаты выводятся не в алфавитном порядке (при необходимости отправьте вывод в команду `sort`):

```
$ find /etc -print
```

```
/etc
/etc/issue.net
/etc/nanorc
/etc/apache2
/etc/apache2/sites-available
/etc/apache2/sites-available/default.conf
:
```

*Список всех каталогов в /etc  
рекурсивно*

`find` имеет множество опций, которые вы можете комбинировать. Рассмотрим несколько наиболее полезных.

Ограничьте вывод только файлами или каталогами с опцией `-type`:

```
$ find . -type f -print
```

*Только файлы*

```
$ find . -type d -print
```

*Только каталоги*

Ограничьте вывод именами, которые соответствуют шаблону имени файла с опцией `-name`. Заключите шаблон в кавычки или экранируйте, чтобы оболочка его не вычисляла:

```
$ find /etc -type f -name "*.conf" -print
```

```
/etc/logrotate.conf
/etc/systemd/logind.conf
/etc/systemd/timesyncd.conf
:
```

*Файлы, заканчивающиеся на .conf*

Сделайте соответствие имен шаблону нечувствительным к регистру букв с помощью опции `-iname`:

```
$ find . -iname "*.txt" -print
```

`find` также может выполнить команду Linux для всех файлов в выходных данных, используя `-exec`. Синтаксис выглядит несколько громоздким:

1. Наберите команду `find` без параметра `-print`.
2. Добавьте параметр `-exec`, а затем команду для выполнения. Используйте скобки `{}`, чтобы указать, где в команде должен отображаться путь к файлу.
3. В конце выражения используйте точку с запятой в кавычках `;"` или экранированную `\;`.

<sup>1</sup> Родственная команда `ls -R` выдает данные в формате, менее удобном для использования в конвейерах.

Вот простой пример вывода символа @ по обеим сторонам пути к файлу:

```
$ find /etc -exec echo @ {} @ ";"
@ /etc @
@ /etc/issue.net @
@ /etc/nanorc @
:
```

В более показательном примере все файлы с расширением *.conf* в каталоге */etc* и его подкаталогах выводятся в формате длинного списка (*ls -l*):

```
$ find /etc -type f -name "*.conf" -exec ls -l {} ";"
-rw-r--r-- 1 root root 703      Aug 21 2017 /etc/logrotate.conf
-rw-r--r-- 1 root root 1022     Apr 20 2018 /etc/systemd/logind.conf
-rw-r--r-- 1 root root 604      Apr 20 2018 /etc/systemd/timesyncd.conf
:
```

*find -exec* можно использовать для удаления файлов во вложенных каталогах (но будьте осторожны!). Давайте удалим файлы с именами, оканчивающимися на тильду (~), в каталоге *\$HOME/tmp* и его подкаталогах. В целях безопасности сначала запустите команду *echo rm*, чтобы увидеть, какие файлы будут удалены, а затем уберите *echo* из выражения, чтобы удалить по-настоящему:

```
$ find $HOME/tmp -type f -name "*~" -exec echo rm {} ";"  Добавим echo
rm /home/smith/tmp/file1~
rm /home/smith/tmp/junk/file2~
rm /home/smith/tmp/vm/vm-8.2.0b/lisp/vm-cus-load.el~
$ find $HOME/tmp -type f -name "*~" -exec rm {} ";"        Удалить по-настоящему
```

## Команда *yes*

Команда *yes* выводит одну и ту же строку снова и снова, пока вы ее не остановите:

```
$ yes                                     Выводит «у» по умолчанию
у
у
у ^C                                     Остановить команду с помощью Ctrl-C
$ yes woof!                             Повторять любую другую строку
woof!
woof!
woof! ^C
```

Какая практическая польза от этого? **yes** может вводить данные для интерактивных программ, чтобы они могли работать без вашего вмешательства. Например, программа **fsck**, которая проверяет файловую систему Linux на наличие ошибок, может спросить пользователя о необходимости продолжения и будет ожидать ответа **y** или **n**. Выходные данные команды **yes**, переданные в **fsck**, отвечают на каждое приглашение от вашего имени, так что вы можете уйти и позволить **fsck** выполниться до конца<sup>1</sup>.

Основное использование команды **yes** в наших примерах — печать строки определенное количество раз путем передачи **yes** в **head** (вы увидите практический пример использования в разделе «Создание тестовых файлов» на с. 177):

```
$ yes "Efficient Linux" | head -n3          Вывести строку 3 раза
Efficient Linux
Efficient Linux
Efficient Linux
```

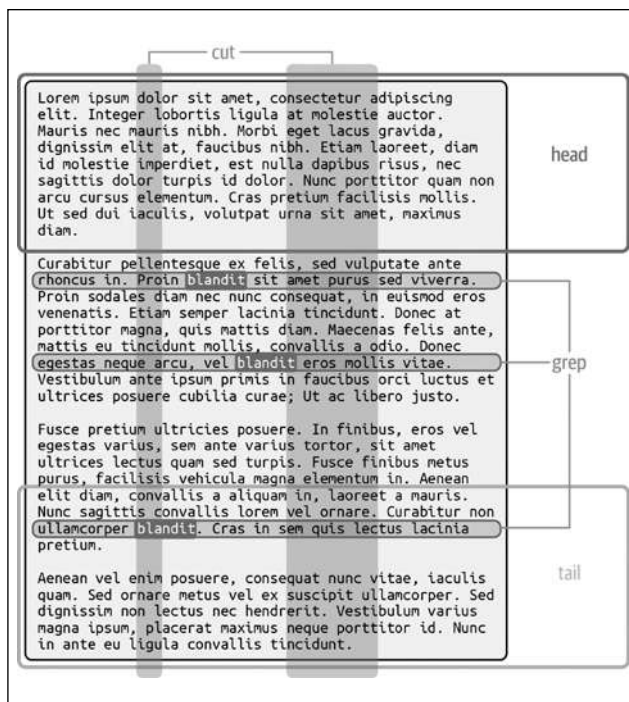
## Извлечение текста

Когда вам нужна только часть файла, самые простые команды для объединения в конвейер — это **grep**, **cut**, **head** и **tail**. Вы уже видели первые три в действии в главе 1: **grep** выводит строки, соответствующие заданной строке; **cut** выводит столбцы из файла; **head** выводит первые строки файла. Новая команда **tail** является противоположностью **head** и печатает последние строки файла. На рис. 5.1 показана совместная работа этих четырех команд.

В этой главе рассмотрим подробно команду **grep**, которая может делать значительно больше, чем просто сопоставление строк. Более формально будет объяснена **tail**. Также остановимся на функции команды **awk** для извлечения столбцов способом, недоступным для **cut**. Комбинация этих пяти команд может извлечь практически любой текст с помощью одного конвейера.

---

<sup>1</sup> Некоторые современные реализации **fsck** имеют опции **-y** и **-n** для ответов «да» или «нет» на каждое приглашение, поэтому команда **yes** для них не требуется.



**Рис. 5.1.** head, grep и tail извлекают строки и столбцы. В этом примере grep соответствует строкам, содержащим слово *blandit*

## Команда grep. Более глубокий взгляд

Вы уже видели, как grep выводит из файла строки, соответствующие заданному шаблону:

```
$ cat frost
```

```
Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.
This is not the end of the poem.
```

```
$ grep his frost
```

```
To watch his woods fill up with snow.
This is not the end of the poem.
```

*Вывести строки, содержащие «his»*

*«This» содержит «his»*

grep имеет несколько очень полезных параметров. Используйте -w для нахождения соответствий только целым словам:

```
$ grep -w his frost
To watch his woods fill up with snow.
```

Искать точное соответствие «his»

Используйте параметр -i, чтобы игнорировать регистр букв:

```
$ grep -i his frost
His house is in the village though;
To watch his woods fill up with snow.
This is not the end of the poem.
```

Содержит «His»  
Содержит «his»  
«This» содержит «his»

Используйте параметр -l, чтобы вывести только имена файлов, содержащих совпадающие строки, но не сами строки:

```
$ grep -l his *
frost
```

В каком файле содержится «his»?

Однако настоящая сила `grep` проявляется, когда вы переходите от сопоставления простых строк к сопоставлению шаблонов, называемых *регулярными выражениями*<sup>1</sup>. В этом случае синтаксис отличается от шаблонов имен файлов. Частичное описание синтаксиса приведено в табл. 5.1.

**Таблица 5.1.** Синтаксис регулярных выражений, используемый командами `grep`, `awk` и `sed`<sup>2</sup>

Соответствие	Используемые выражения	Пример
Начало строки	^	^a = строка, начинающаяся с a
Конец строки	\$	!\$ = строка, заканчивающаяся восклицательным знаком
Любой одиночный символ (кроме новой строки)	.	... = любые три последовательных символа
Знаки вставки, доллара или любой другой специальный символ c	\c	\$ = знак доллара
Ноль или более вхождений выражения E	E*	_* = ноль или более знаков подчеркивания
Любой одиночный символ в наборе	[characters]	[aeiouAEIOU] = любая гласная

<sup>1</sup> Название `grep` является сокращением от «получить регулярное выражение и распечатать» (get regular expression and print).

<sup>2</sup> Эти три команды различаются в обработке регулярных выражений. В табл. 5.1 представлен неполный список выражений.



Соответствие	Используемые выражения	Пример
Любой одиночный символ, не входящий в набор	[^characters]	[^aeiouAEIOU] = любая не гласная
Любой символ в диапазоне между <i>c1</i> и <i>c2</i>	[c1-c2]	[0-9] = любая цифра
Любой символ вне диапазона между <i>c1</i> и <i>c2</i>	[^c1-c2]	[^0-9] = любой нецифровой символ
Любое из двух выражений <i>E1</i> или <i>E2</i>	<i>E1</i> \  <i>E2</i> для grep и sed, <i>E1</i> / <i>E2</i> для awk	one \  two = или <i>one</i> , или <i>two</i> one   two = или <i>one</i> , или <i>two</i>
Группировка выражения <i>E</i> с учетом приоритета	\( <i>E</i> \) для grep и sed <sup>1</sup> , ( <i>E</i> ) для awk	\(one \  two\) \  * = ноль или более вхождений <i>one</i> или <i>two</i> , (one   two) \  * = ноль или более вхождений <i>one</i> или <i>two</i>

Рассмотрим несколько примеров команд `grep` с регулярными выражениями.

Найти совпадение всех строк, начинающихся с заглавной буквы:

```
$ grep '^[A-Z]' myfile
```

Найти совпадение со всеми непустыми строками (то есть совпадение с пустыми строками и их пропуск благодаря параметру `-v`):

```
$ grep -v '^$' myfile
```

Найти все строки, содержащие либо *cookie*, либо *cake*:

```
$ grep 'cookie\|cake' myfile
```

Найти все строки длиной не менее пяти символов:

```
$ grep '.....' myfile
```

Найти все строки, в которых символ «меньше» появляется перед символом «больше», например строки кода HTML:

```
$ grep '<.*>' page.html
```

<sup>1</sup> Для `sed` этот синтаксис делает больше, чем просто группировка, см. «Сопоставление подвыражений `sed`» на с. 116.

Регулярные выражения прекрасны, но иногда они приводят к неожиданным результатам. Предположим, вы хотите найти в файле *frost* две строки, содержащие букву *w*, за которой следует точка. Следующая команда дает неправильные результаты, поскольку точка — это регулярное выражение, означающее «любой символ»:

```
$ grep w. frost
Whose woods these are I think I know.
He will not see me stopping here
To watch his woods fill up with snow.
```

Чтобы обойти эту проблему, вы можете экранировать специальный символ:

```
$ grep 'w\.' frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
```

Такое решение становится громоздким, если вам нужно экранировать много специальных символов. К счастью, вы можете заставить *grep* забыть о регулярных выражениях и искать буквально каждый символ во входных данных, используя параметр *-F* (fixed). Или, в качестве альтернативы с таким же результатом, запустите *fgrep* вместо *grep*:

```
$ grep -F w. frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
$ fgrep w. frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
```

*grep* имеет много других параметров. Распространенную проблему решает параметр *-f* (строчный символ, не путайте его с *F*), служащий для сопоставления с набором строк, а не с одной строкой. В качестве практического примера давайте перечислим все оболочки, найденные в файле */etc/passwd*, который я представил в разделе «Команда #5: *sort*» на с. 28. Как вы помните, каждая строка в */etc/passwd* содержит информацию о пользователе, а двоеточия служат разделителями полей. Последнее поле в каждой строке — имя программы, запускаемой при входе пользователя в систему. Эта программа зачастую, но не всегда, является оболочкой:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
:
```

Седьмое поле — оболочка  
Седьмое поле — не оболочка

Как узнать, является ли программа оболочкой? В файле */etc/shells* перечислены все допустимые оболочки входа в систему Linux:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/bin/csh
```

Таким образом, вы можете перечислить все действующие оболочки из */etc/passwd*, извлекая седьмое поле с помощью `cut`, удаляя дубликаты с помощью `sort -u` и сверяя результаты с */etc/shells* с помощью `grep -f`. Также добавим параметр `-F`, так как все строки в */etc/shells* воспринимаются буквально, даже если они содержат специальные символы:

```
$ cut -d: -f7 /etc/passwd | sort -u | grep -f /etc/shells -F
/bin/bash
/bin/sh
```

## Команда `tail`

Команда `tail` печатает последние строки файла — по умолчанию 10 строк. Это сестра команды `head`. Предположим, у вас есть файл с именем *alphabet*, содержащий 26 строк, по одной на букву:

```
$ cat alphabet
A is for aardvark
B is for bunny
C is for chipmunk
:
X is for xenorhabdus
Y is for yak
Z is for zebu
```

Выведите последние три строки с помощью `tail`. Опция `-n` задает количество печатаемых строк, как и для команды `head`:

```
$ tail -n3 alphabet
X is for xenorhabdus
Y is for yak
Z is for zebu
```

Если поставить перед номером знак `+`, печать начнется со строки этого номера и продолжится до конца файла. Следующая команда начинается с 25-й строки файла:

```
$ tail -n+25 alphabet
Y is for yak
Z is for zebu
```

Объедините `tail` и `head`, чтобы напечатать любой диапазон строк из файла. Например, чтобы напечатать только четвертую строку, извлеките первые четыре строки и выведите последнюю:

```
$ head -n4 alphabet | tail -n1
D is for dingo
```

В общем, чтобы напечатать строки от *M* до *N*, извлеките первые *N* строк с помощью `head`, затем выведите последние *N-M+1* строк с помощью `tail`. Выведите строки с шестой по восьмую файла *alphabet*:

```
$ head -n8 alphabet | tail -n3
F is for falcon
G is for gorilla
H is for hawk
```



И `head`, и `tail` поддерживают более простой синтаксис для указания количества строк без параметра `-n`. Этот синтаксис — недокументированный и устаревший, но, вероятно, будет поддерживаться всегда:

\$ head -4 alphabet	<i>То же самое, что u head -n4 alphabet</i>
\$ tail -3 alphabet	<i>То же самое, что u tail -n3 alphabet</i>
\$ tail +25 alphabet	<i>То же самое, что u tail -n+25 alphabet</i>

## Команда `awk {print}`

Команда `awk` представляет собой обработчик текста общего назначения с сотнями применений. Давайте рассмотрим одну небольшую функцию, `print`, которая извлекает столбцы из файла способами, недоступными для `cut`. Рассмотрим системный файл `/etc/hosts`, который включает IP-адреса и имена хостов, разделенные любым количеством пробелов:

```
$ less /etc/hosts
127.0.0.1      localhost
127.0.1.1      myhost          myhost.example.com
192.168.1.2    frodo
192.168.1.3      gollum
192.168.1.28   gandalf
```

Предположим, вы хотите вывести на экран имена хостов — второе слово в каждой строке. Проблема в том, что каждому имени хоста предшествует произвольное количество пробелов. `cut` требует, чтобы столбцы были либо аккуратно выровнены (`-c`), либо разделены одним определенным символом (`-f`). Вам же нужна команда для печати второго слова в каждой строке, и ее предоставляет `awk`:

```
$ awk '{print $2}' /etc/hosts
localhost
myhost
frodo
gollum
gandalf
```

`awk` ссылается на любой столбец знаком доллара, за которым следует номер столбца, например `$7` для седьмого столбца. Если номер столбца состоит из нескольких цифр, заключите число в круглые скобки, например `$(25)`. Чтобы обратиться к последнему полю, используйте `$NF` (number of fields, то есть число полей). Чтобы обратиться ко всей строке, используйте `$0`.

`awk` по умолчанию не печатает пробелы между значениями. Если вам нужны пробелы, разделите значения запятыми:

```
$ echo Efficient fun Linux | awk '{print $1 $3}'      Без пробела
EfficientLinux
$ echo Efficient fun Linux | awk '{print $1, $3}'      С пробелом
Efficient Linux
```

Оператор `print` команды `awk` отлично подходит для обработки вывода команд, даже если колонки совсем неаккуратны. Примером может служить команда `df`, которая выводит объем свободного и занятого дискового пространства в системе Linux:

```
$ df / /data
Filesystem 1K-blocks    Used    Available    Use% Mounted on
/dev/sda1  1888543276    902295944    890244772    51% /
/dev/sda2   7441141620   1599844268    5466214400    23% /data
```

Расположение столбцов может различаться в зависимости от длины путей в столбце `Filesystems`, размеров диска и параметров, которые вы передаете `df`, поэтому вы не можете с уверенностью извлекать значения с помощью `cut`. Однако с помощью `awk` можно легко вывести, например, четвертое значение в каждой строке с данными о доступном месте на диске:

```
$ df / /data | awk '{print $4}'
Available
890244772
5466214400
```

При этом можно не выводить первую строку (заголовок):

```
$ df / /data | awk 'FNR>1 {print $4}'
890244772
5466214400
```

Если вы столкнулись с вводом, разделенным чем-то другим, кроме пробелов, `awk` с параметром `-F` может изменить разделитель полей на любое регулярное выражение:

```
$ echo efficient:::::linux | awk -F':*' '{print $2}'    Любое количество двоеточий
linux
```

Вы узнаете больше о команде `awk` в разделе «Основы `awk`» на с. 109.

## Объединение текста

Вы уже знаете несколько команд, объединяющих текст из разных файлов. Первая — это `cat`, которая выводит содержимое нескольких файлов. Это последовательное объединение текстов «сверху вниз». Отсюда и его название — он объединяет (*concatenates*) файлы:

```
$ cat poem1
It is an ancient Mariner,
And he stoppeth one of three.
$ cat poem2
'By thy long grey beard and glittering eye,
$ cat poem3
Now wherefore stopp'st thou me?
$ cat poem1 poem2 poem3
It is an ancient Mariner,
And he stoppeth one of three.
'By thy long grey beard and glittering eye,
Now wherefore stopp'st thou me?
```

Вторая команда для объединения текста, которую вы видели, — это встроенная в оболочку `echo`. Она печатает любые переданные вами аргументы, разделяя их одним пробелом, и соединяет тексты в одну строку:

```
$ echo efficient linux in $HOME
efficient linux in /home/smith
```

Давайте рассмотрим еще несколько команд, объединяющих текст:

`tac` — последовательное объединение текстовых файлов.

`paste` — объединение строк текстовых файлов.

`diff` — команда, которая чередует текст из двух файлов, отображая только их различия.

## Команда `tac`

Команда `tac` построчно переворачивает каждый текст. Ее имя — это написанное задом наперед название команды `cat`.

```
$ cat poem1 poem2 poem3 | tac
Now wherefore stopp'st thou me?
'By thy long grey beard and glittering eye,
And he stoppeth one of three.
It is an ancient Mariner,
```

Обратите внимание, что три файла были объединены, прежде чем перевернуть текст. Если вместо этого указать команде `tac` несколько файлов в качестве аргументов, она перевернет строки каждого файла по очереди, выведя на экран другие данные:

```
$ tac poem1 poem2 poem3
And he stoppeth one of three.           Первый файл перевернут
It is an ancient Mariner,
'By thy long grey beard and glittering eye,   Второй файл
Now wherefore stopp'st thou me?           Третий файл
```

`tac` отлично подходит для обработки данных, которые уже находятся в определенном порядке, но не могут быть отсортированы с помощью команды `sort -r`. Типичным случаем является преобразование файла журнала веб-сервера для обработки его строк от самых новых к самым старым:

```
192.168.1.34 - - [30/Nov/2021:23:37:39 -0500] "GET / HTTP/1.1" ...
192.168.1.10 - - [01/Dec/2021:00:02:11 -0500] "GET /notes.html HTTP/1.1" ...
192.168.1.8 - - [01/Dec/2021:00:04:30 -0500] "GET /stuff.html HTTP/1.1" ...
:
```

Строки расположены в хронологическом порядке с отметками времени, но не в алфавитном или числовом порядке, поэтому команда `sort -r` бесполезна. Команда `tac` может перевернуть эти строки, не обращая внимания на метки времени.

## Команда `paste`

Команда `paste` объединяет строки текстов в столбцы, разделенные одним символом табуляции. Это сестра команды `cut`, которая извлекает разделенные табуляцией столбцы из текста:

```
$ cat title-words1
EFFICIENT
AT
COMMAND
```

```
$ cat title-words2
linux
the
line
$ paste title-words1 title-words2
EFFICIENT    linux
AT the
COMMAND line
$ paste title-words1 title-words2 | cut -f2      cut и paste дополняют друг друга
linux
the
line
```

Измените разделитель на другой символ, например запятую, с помощью параметра `-d` (delimiter):

```
$ paste -d, title-words1 title-words2
EFFICIENT,linux
AT,the
COMMAND,line
```

Измените вывод, поменяв последовательность объединения с помощью параметра `-s`:

```
$ paste -d, -s title-words1 title-words2
EFFICIENT,AT,COMMAND
linux,the,line
```

Команда `paste` также может чередовать строки из двух или более файлов, если вы измените разделитель на символ новой строки (`\n`):

```
$ paste -d "\n" title-words1 title-words2
EFFICIENT
linux
AT
the
COMMAND
line
```

## Команда diff

Команда `diff` сравнивает два файла построчно и выводит краткий отчет об их различиях:

```
$ cat file1
Linux is all about efficiency.
I hope you will enjoy this book.
$ cat file2
MacOS is all about efficiency.
```



```

I hope you will enjoy this book.
Have a nice day.
$ diff file1 file2
1c1
< Linux is all about efficiency.
---
> MacOS is all about efficiency.
2a3
> Have a nice day.

```

Код `1c1` означает, что строка 1 в первом файле отличается от строки 1 во втором файле. За этим кодом следует соответствующая строка из *file1*, разделитель из трех дефисов (`---`) и соответствующая строка из *file2*. Начальный символ `<` всегда указывает на строку из первого файла, а `>` указывает на строку из второго файла.

Код `2a3` означает, что в файле *file2* есть третья строка, отсутствующая после второй строки файла *file1*. За этим обозначением следует дополнительная строка из файла 2: *Have a nice day.*

Вывод `diff` может содержать другие обозначения и принимать другие формы. Однако этого краткого объяснения достаточно для нашей основной цели — использования `diff` для чередования строк из двух файлов. Многие пользователи не думали о таком применении `diff`, но эта команда отлично подходит для формирования конвейеров при решении определенных задач. Например, вы можете вывести разные строки с помощью `diff`, `grep` и `cut`:

```

$ diff file1 file2 | grep '^[<>]'
< Linux is all about efficiency.
> MacOS is all about efficiency.
> Have a nice day.
$ diff file1 file2 | grep '^[<>]' | cut -c3-
Linux is all about efficiency.
MacOS is all about efficiency.
Have a nice day.

```

Практические примеры приведены в разделах «Способ #4: Подстановка процесса» на с. 140 и «Проверка совпадающих пар файлов» на с. 172.

## Преобразование текста

В главе 1 было представлено несколько команд, которые считывают текст со стандартного ввода и преобразуют его в стандартном выходе. Команда `wc` выводит количество строк, слов и символов; `sort` упорядочивает строки в алфавитном или числовом порядке; а `uniq` объединяет повторяющиеся строки. Давайте обсудим еще несколько команд, которые преобразуют ввод:

`tr` — преобразует одни символы в другие.

`rev` — переворачивает символы в строке задом наперед.

`awk` и `sed` — преобразователи текста общего назначения.

## Команда `tr`

Команда `tr` переводит один набор символов в другой. В главе 2 мы видели пример преобразования двоеточий в символы новой строки для вывода переменной `PATH`:

```
$ echo $PATH | tr : "\n"
/home/smith/bin
/usr/local/bin
/usr/bin
/bin
/usr/games
/usr/lib/java/bin
```

*Преобразование двоеточий в символы новой строки*

`tr` принимает два набора символов в качестве аргументов и переводит элементы первого набора в соответствующие элементы второго. Часто выполняется преобразование текста в верхний или нижний регистр:

```
$ echo efficient | tr a-z A-Z
EFFICIENT
$ echo Efficient | tr A-Z a-z
efficient
```

*Перевод а в А, b в В и т. д.*

Преобразование пробелов в символы новой строки:

```
$ echo Efficient Linux | tr " " "\n"
Efficient
Linux
```

Удаление пробелов с помощью параметра `-d`:

```
$ echo efficient linux | tr -d ' \t'
efficientlinux
```

*Удаление пробелов и знаков табуляции*

## Команда `rev`

Команда `rev` переворачивает символы задом наперед в каждой строке ввода<sup>1</sup>:

```
$ echo Efficient Linux! | rev
!xuniL tneiciffE
```

<sup>1</sup> Вопрос к читателям: что делает конвейер `rev myfile | tac | rev | tac`?

Помимо очевидной развлекательной ценности, **rev** удобна для извлечения сложной информации из файлов. Предположим, у вас есть файл с именами знаменитостей:

```
$ cat celebrities
Jamie Lee Curtis
Zooey Deschanel
Zendaya Maree Stoermer Coleman
Rihanna
```

и вы хотите вывести на экран последнее слово из каждой строки (Curtis, Deschanel, Coleman, Rihanna). Это было бы легко сделать с помощью команды **cut -f**, если бы в каждой строке было одинаковое количество полей, но это число различается. С помощью **rev** вы можете перевернуть все строки, вырезать первое слово и снова перевернуть, чтобы добиться желаемого:

```
$ rev celebrities
sitruC eel eimaJ
lenahcseD yeooZ
nameloC remreotS eeraM ayadneZ
annahiR
$ rev celebrities | cut -d' ' -f1
sitruC
lenahcseD
nameloC
annahiR
$ rev celebrities | cut -d' ' -f1 | rev
Curtis
Deschanel
Coleman
Rihanna
```

## Команды **awk** и **sed**

**awk** и **sed** — это универсальные «суперкоманды» для обработки текста. Они позволяют решать практически все задачи, которые рассматривались ранее в этой главе, но с помощью более сложного синтаксиса. Например, они могут вывести первые 10 строк файла, как это делает **head**:

```
$ sed 10q myfile           Выводит 10 строк и завершается
$ awk 'FNR<=10' myfile     Выводит строки на экран, до тех пор пока их количество не достигнет 10
```

Их возможности шире всех ранее изученных нами команд для работы с текстовыми файлами. Например, они позволяют заменять или менять местами строки:

```
$ echo image.jpg | sed 's/\.jpg/\.png/'          Заменить .jpg на .png
image.png
$ echo "linux efficient" | awk '{print $2, $1}'    Менять местами два слова
efficient linux
```

`awk` и `sed` сложнее изучить, чем описанные выше команды, потому что каждая из них имеет встроенный миниатюрный язык программирования. Об их возможностях написаны книги<sup>1</sup>. Я настоятельно рекомендую потратить время на изучение обеих команд или хотя бы одной из них.

Чтобы начать путешествие в мир `awk` и `sed`, рассмотрим основные принципы каждой команды и некоторые распространенные варианты их использования. Я также порекомендую несколько онлайн-руководств.

Не беспокойтесь о том, чтобы запомнить каждую функцию `awk` или `sed`. Действительно успешное использование этих команд возможно при двух условиях:

- Если вы понимаете их возможности. Тогда в нужный момент вы скажете себе: «Ага! Вот эту работу сможет сделать `awk` (или `sed`)!»
- Если научитесь использовать справочные страницы команд и находить готовые решения на Stack Exchange (<https://oreil.ly/0948M>) и других онлайн-ресурсах.

## Основы `awk`

`awk` преобразует строки текста из файлов (или из стандартного ввода) в любой другой текст, используя последовательность инструкций, которую называют *awk-программой*<sup>2</sup>. Чем больше опыта написания `awk`-программ вы накапливаете, тем гибче можете манипулировать текстом. Вы можете указать такую программу в командной строке:

```
$ awk program input-files
```

но также можете хранить одну или несколько `awk`-программ в файлах и обращаться к ним с параметром `-f`. Программы будут запускаться последовательно:

```
$ awk -f program-file1 -f program-file2 -f program-file3 input-files
```

Программа `awk` включает одно или несколько действий, таких как вычисление значений или печать текста, которые запускаются, когда входная строка соответствует шаблону. Каждая инструкция в программе имеет вид:

```
шаблон {действие}
```

<sup>1</sup> Например, «`sed & awk`», изданная O'Reilly.

<sup>2</sup> `awk` является аббревиатурой от фамилий создателей программы: Ахо (Aho), Вайнбергера (Weinberger) и Кернигана (Kernighan).

Типичные шаблоны включают:

- .Слово **BEGIN** — действие запускается только один раз, перед обработкой ввода команды **awk**.

- .Слово **END** — действие запускается только один раз, после обработки ввода команды **awk**.

- .Регулярное выражение (см. табл. 5.1), окруженное слешами. Например, `/^[A-Z]/` соответствует строкам, начинающимся с заглавной буквы.

Пара примеров других выражений, специфичных для **awk**: `$3~/^[A-Z]/` проверяет, начинается ли третье поле в строке ввода (**\$3**) с заглавной буквы; `FNR>5` указывает **awk** пропустить первые пять строк ввода.

Действие без шаблона выполняется для каждой строки ввода (несколько **awk**-программ в разделе «Команда **awk** {print}» на с. 101 относятся к этому типу). Например, **awk** элегантно решает задачу «напечатать фамилию знаменитости» из примера в разделе «Команды **rev**» на с. 107, напрямую печатая последнее слово из каждой строки:

```
$ awk '{print $NF}' celebrities
Curtis
Deschanel
Coleman
Rihanna
```



При вводе **awk**-программы в командной строке заключайте ее в кавычки, чтобы оболочка не вычисляла специальные символы **awk**. При необходимости используйте одинарные или двойные кавычки.

Шаблон без указания действия запускает действие по умолчанию **{print}**, которое просто печатает любые совпадающие входные строки без изменений:

```
$ echo efficient linux | awk '/efficient/'
efficient linux
```

Для более полной демонстрации давайте обработаем разделенный табуляцией текст файла *animals.txt* из примера 1.1 на с. 21, чтобы создать аккуратную библиографию. Необходимо преобразовать строки вида

```
python Programming Python 2010 Lutz, Mark
```

в следующий формат:

```
Lutz, Mark (2010). "Programming Python"
```

Необходима перестановка трех полей и добавление некоторых символов, таких как круглые скобки и двойные кавычки. Следующая `awk`-программа выполняет это, используя параметр `-F` для изменения разделителя ввода с пробелов на табуляцию (`\t`):

```
$ awk -F'\t' '{print $4, "(" $3 ").", "\"" $2 "\""}' animals.txt
Lutz, Mark (2010). "Programming Python"
Barrett, Daniel (2005). "SSH, The Secure Shell"
Schwartz, Randal (2012). "Intermediate Perl"
Bell, Charles (2014). "MySQL High Availability"
Siever, Ellen (2009). "Linux in a Nutshell"
Boney, James (2005). "Cisco IOS in a Nutshell"
Roman, Steven (1999). "Writing Word Macros"
```

Добавим регулярное выражение для обработки только названия книги, содержащего *horse*:

```
$ awk -F'\t' ' /^horse/ {print $4, "(" $3 ").", "\"" $2 "\""}' animals.txt
Siever, Ellen (2009). "Linux in a Nutshell"
```

Или отберем только книги, изданные не раньше 2010 года, проверив, соответствует ли поле `$3` шаблону `^201`:

```
$ awk -F'\t' ' $3~/^201/ {print $4, "(" $3 ").", "\"" $2 "\""}' animals.txt
Lutz, Mark (2010). "Programming Python"
Schwartz, Randal (2012). "Intermediate Perl"
Bell, Charles (2014). "MySQL High Availability"
```

Наконец, добавим инструкцию `BEGIN`, чтобы напечатать понятный заголовок, дефисы для отступов и инструкцию `END`, чтобы направить читателя к дополнительной информации:

```
$ awk -F'\t' \
' BEGIN {print "Recent books:"} \
$3~/^201/{print "-", $4, "(" $3 ").", "\"" $2 "\""} \
END {print "For more books, search the web"}' \
animals.txt
Recent books:
- Lutz, Mark (2010). "Programming Python"
- Schwartz, Randal (2012). "Intermediate Perl"
- Bell, Charles (2014). "MySQL High Availability"
For more books, search the web
```

Команда `awk` умеет гораздо больше, чем просто вывод данных — она также может выполнять вычисления, например суммировать числа от 1 до 100:

```
$ seq 1 100 | awk '{s+=$1} END {print s}'
5050
```

Чтобы изучить `awk`, воспользуйтесь учебными пособиями на [tutorialspoint.com/awk](http://tutorialspoint.com/awk) или [riptutorial.com/awk](http://riptutorial.com/awk) либо выполните поиск в интернете по запросу *awk tutorial*. Результат вам понравится.

### Улучшенный способ обнаружения дубликатов файлов

В разделе «Обнаружение дубликатов файлов» на с. 32 вы построили конвейер, который обнаруживает и подсчитывает дубликаты файлов JPEG по контрольной сумме, но его возможностей не хватает для вывода имен файлов:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr | grep -v " 1 "
```

```
3 f6464ed766daca87ba407aede21c8fcc
2 c7978522c58425f6af3f095ef1de1cd5
2 146b163929b6533f02e91bdf21cb9563
```

Теперь, когда мы знаем команду `awk`, у нас также есть инструменты для печати имен файлов. Давайте создадим новую команду, которая считывает каждую строку вывода `md5sum`:

```
$ md5sum *.jpg
146b163929b6533f02e91bdf21cb9563 image001.jpg
63da88b3ddde0843c94269638dfa6958 image002.jpg
146b163929b6533f02e91bdf21cb9563 image003.jpg
:
```

но не только подсчитывает вхождения каждой контрольной суммы, но и сохраняет имена файлов для вывода на экран. Нам понадобятся две дополнительные возможности команды `awk` — *массивы* и *циклы*.

Массив — это переменная, содержащая набор значений. Если массив называется `A` и содержит семь значений, то к ним можно обращаться через *элементы* массива `A[1]`, `A[2]`, `A[3]`, вплоть до `A[7]`. Значения от 1 до 7 называются *ключами* массива. Вы можете создать любые ключи, какие захотите. Если вы предпочитаете обращаться к элементам массива, используя имена персонажей Диснея, назовите их `A["Doc"]`, `A["Grumpy"]`, `A["Bashful"]`, вплоть до `A["Dopey"]`.

Чтобы подсчитать повторяющиеся изображения, создадим массив `counts` с одним элементом для каждой контрольной суммы. Каждый ключ массива представляет собой контрольную сумму, а связанный с ним элемент содержит количество раз, которое контрольная сумма встречается во входных данных. Например, элемент массива `counts["f6464ed766daca87ba407aede21c8fcc"]` может иметь значение 3. Следующий сценарий команды `awk` проверяет каждую строку вывода `md5sum`, выделяет контрольную сумму (`$1`) и использует ее в качестве ключа для массива

`counts`. Оператор `++` увеличивает элемент на 1 каждый раз, когда `awk` встречается связанную с ним контрольную сумму:

```
$ md5sum *.jpg | awk '{counts[$1]++}'
```

Пока что `awk`-программа ничего не выводит, она просто обрабатывает каждую контрольную сумму и завершает работу. Чтобы вывести на экран количество посчитанных значений, нам понадобится вторая функция `awk`, называемая циклом `for`. Цикл `for` проходит по всем ключам массива и последовательно обрабатывает каждый его элемент. Например, следующая команда выводит значения элементов массива `counts` по их ключам:

```
for (key in counts) print counts[key]
```

Поместим этот цикл в инструкцию `END`, чтобы он выполнялся после вычисления всех элементов массива:

```
$ md5sum *.jpg \
| awk '{counts[$1]++} \
END { for (key in counts) print counts[key] }'
```

```
1
2
2
:
```

Затем добавим контрольные суммы в вывод. Каждый ключ массива является контрольной суммой, поэтому просто выведем их:

```
$ md5sum *.jpg \
| awk '{counts[$1]++} \
END {for (key in counts) print counts[key] " " key }'
```

```
1 714ecee06b43c03fe20eb96474f69b8
2 146b163929b6533f02e91bdf21cb9563
2 c7978522c58425f6af3f095ef1de1cd5
:
```

Для сбора и вывода имен файлов используем массив `names` также с контрольными суммами в качестве ключей. Так как `awk` обрабатывает каждую строку вывода, добавим имя файла (`$2`) к соответствующему элементу массива имен вместе с пробелом в качестве разделителя. В цикле `END` после печати контрольной суммы (`key`) выведем двоеточие и собранные имена файлов для этой контрольной суммы:

```
$ md5sum *.jpg \
| awk '{counts[$1]++; names[$1]=names[$1] " " $2} \
END {for (key in counts) print counts[key] " " key ":" names[key] }'
```

```
1 714ecee06b43c03fe20eb96474f69b8: image011.jpg
2 146b163929b6533f02e91bdf21cb9563: image001.jpg image003.jpg
2 c7978522c58425f6af3f095ef1de1cd5: image019.jpg image020.jpg
:
```



Строки, начинающиеся с единицы, представляют собой контрольные суммы, которые встречаются только один раз, то есть не дублируются. Направим вывод в `grep -v`, чтобы удалить эти строки, затем отсортируем результаты от большего к меньшему с помощью `sort -nr` — и получим желаемый результат:

```
$ md5sum *.jpg \
| awk '{counts[$1]++; names[$1]=names[$1] " " $2} \
END {for (key in counts) print counts[key] " " key ":" names[key]}' \
| grep -v '^1 ' \
| sort -nr
3 f6464ed766daca87ba407aede21c8fcc: image007.jpg image012.jpg image014.jpg
2 c7978522c58425f6af3f095ef1de1cd5: image019.jpg image020.jpg
2 146b163929b6533f02e91bdf21cb9563: image001.jpg image003.jpg
```

## Основы sed

Команда `sed` преобразует текст из файлов или из стандартного ввода, используя последовательность инструкций, которую называют *sed-сценарием*<sup>1</sup>. Сценарии `sed` на первый взгляд малопонятны. Примером может служить `s/windows/Linux/g`, заменяющий каждое вхождение строки *Windows* на *Linux*. Термин *сценарий* в данном случае означает не файл (например, сценарий оболочки), а строку<sup>2</sup>. Вызовите сценарий `sed` в командной строке:

```
$ sed script input-files
```

или используйте параметр `-e` для поддержки нескольких сценариев, которые последовательно обрабатывают ввод:

```
$ sed -e script1 -e script2 -e script3 input-files
```

Также можно хранить `sed`-сценарии в файлах и обращаться к ним с параметром `-f`, тогда они будут запускаться последовательно:

```
$ sed -f script-file1 -f script-file2 -f script-file3 input-files
```

Как и в случае с `awk`, преимущества использования `sed` зависят от вашего умения создавать сценарии. Наиболее часто используемым сценарием является подстановка, которая заменяет одни строки другими. Ее синтаксис:

```
s/regexp/replacement/
```

<sup>1</sup> Название команды `sed` является сокращением от *stream editor*, потому что она редактирует текстовый поток.

<sup>2</sup> Если вы знакомы с редакторами `vi`, `vim`, `ex` или `ed`, синтаксис сценария `sed` может показаться вам знакомым.

где *regex* — регулярное выражение для сопоставления с каждой входной строкой (см. табл. 5.1); *replacement* — строка для замены совпадающего текста. В качестве простого примера замена слов:

```
$ echo Efficient Windows | sed "s/Windows/Linux/"
Efficient Linux
```



При вводе *sed*-сценария в командной строке заключайте его в кавычки, чтобы оболочка не вычисляла специальные символы *sed*. При необходимости используйте одинарные или двойные кавычки.

*sed* легко решает задачу вывода фамилии знаменитости из раздела «Команда rev» на с. 107 с помощью регулярного выражения. Просто сопоставьте все символы (*.\**) до последнего пробела и ничем их не заменяйте:

```
$ sed 's/.* //' celebrities
Curtis
Deschanel
Coleman
Rihanna
```



### Подстановка и символ слеша

Косая черта в подстановке может быть заменена любым другим удобным символом. Это полезно, когда само регулярное выражение включает слеш, который в противном случае пришлось бы экранировать. Следующие три сценария *sed* эквивалентны:

```
s/one/two/
s_one_two_
s@one@two@
```

В сценарии после подстановки могут следовать несколько параметров, влияющих на результат. Например, параметр *i* делает совпадения нечувствительными к регистру:

```
$ echo Efficient Stuff | sed "s/stuff/linux/"          Чувствительно к регистру. Нет
Efficient Stuff                                       совпадений
$ echo Efficient Stuff | sed "s/stuff/linux/i"        Нечувствительно к регистру
Efficient linux
```

Параметр *g* (*global*) заменяет все вхождения регулярного выражения, а не только первое:

```
$ echo efficient stuff | sed "s/f/F/"      Заменяет только первую «f»
efficient stuff
$ echo efficient stuff | sed "s/f/F/g"      Заменяет все вхождения «f»
eFFicient stuFF
```

Другим распространенным типом *sed*-сценария является удаление. Сценарий, удаляющий строки по их номеру:

```
$ seq 10 14 | sed 4d      Удаляет четвертую строку
10
11
12
14
```

Сценарий, удаляющий строки, которые соответствуют регулярному выражению:

```
$ seq 101 200 | sed '/[13579]$/d'      Удаляет строки, заканчивающиеся
                                         на нечетные цифры
102
104
106
:
200
```

## Сопоставление подвыражений с *sed*

Предположим, у нас есть несколько имен файлов:

```
$ ls
image.jpg.1 image.jpg.2 image.jpg.3
```

и мы хотим создать новые имена, *image1.jpg*, *image2.jpg* и *image3.jpg*. *sed* может разбить имена файлов на части и изменить их порядок с помощью функции, называемой *подвыражениями*. Сначала создадим регулярное выражение, соответствующее именам файлов:

```
image\.jpg\.[1-3]
```

Чтобы переместить последнюю цифру в имени файла на другую позицию, изолируем ее, окружив символами  $\backslash$  (и  $\backslash$ ). Это определяет подвыражение — выделенную часть регулярного выражения:

```
image\.jpg\.\(\[1-3]\)
```

*sed* может ссылаться на подвыражения по номеру и управлять ими. Мы создали только одно подвыражение, поэтому его имя  $\backslash 1$ . Вторым подвыражением будет

\2 и т. д., максимум — \9. Новые имена файлов будут иметь вид `image\1.jpg`. Следовательно, `sed`-сценарий будет таким:

```
$ ls | sed "s/image\.jpg\.\([1-3]\)/image\1.jpg/"
image1.jpg
image2.jpg
image3.jpg
```

Чтобы усложнить ситуацию, предположим, что имена файлов имеют больше отличий и состоят из слов нижнего регистра:

```
$ ls
apple.jpg.1 banana.png.2 carrot.jpg.3
```

Создадим три подвыражения для захвата начального имени файла, расширения и последней цифры:

<code>\([a-z][a-z]*\)</code>	<code>\1</code> = Начальное имя файла из одной буквы или более
<code>\([a-z][a-z][a-z]\)</code>	<code>\2</code> = Расширение файла из трех букв
<code>\([0-9]\)</code>	<code>\3</code> = Цифра

Соединим их с помощью экранированных точек (`\.`), чтобы сформировать следующее регулярное выражение:

```
\([a-z][a-z]*\) \. \([a-z][a-z][a-z]\) \. \([0-9]\)
```

Преобразуем имена файлов в формате `sed` как `\1\3.\2`, тогда окончательный сценарий будет выглядеть так:

```
$ ls | sed "s/\([a-z][a-z]*\) \. \([a-z][a-z][a-z]\) \. \([0-9]\)/\1\3.\2/"
apple1.jpg
banana2.png
carrot3.jpg
```

Эта команда не переименовывает файлы, она просто выводит на экран новые имена. В разделе «Вставка имени файла в последовательность» на с. 169 показан аналогичный пример, в котором также выполняется переименование.

Чтобы изучить `sed`, воспользуйтесь учебниками <https://tutorialspoint.com/sed> или <https://grymoire.com/Unix/Sed.html> либо выполните поиск в интернете по запросу *sed tutorial*.

## Как расширить инструментарий

Большинство систем Linux поставляются с тысячами программ командной строки, и большинство из них имеют множество параметров, изменяющих их

поведение. Вы вряд ли это все выучите и запомните. Отсюда возникает вопрос: как найти нужную программу или адаптировать другую, которую вы уже знаете, для достижения своих целей?

Первый и самый очевидный шаг — поиск в интернете. Например, если вам нужна команда, которая ограничивает ширину строк в текстовом файле, перенося слишком длинные строки, поищите по фразе «перенос строк команд Linux» («Linux command wrap lines») — и вы увидите команду `fold`:

```
$ cat title.txt
This book is titled "Efficient Linux at the Command Line"
$ fold -w40 title.txt
This book is titled "Efficient Linux at
the Command Line"
```

Чтобы найти команды, которые уже установлены в вашей системе Linux, запустите `man -k` (или, что то же самое, команду `apropos`). Получив слово, `man -k` ищет его в кратких описаниях справочных страниц:

```
$ man -k width
DisplayWidth (3) - image format functions and macros
DisplayWidthMM (3) - image format functions and macros
fold (1) - wrap each input line to fit in specified width
:
```

`man -k` работает с регулярными выражениями в стиле `awk` в строках поиска (см. таблицу 5.1):

```
$ man -k "wide|width"
```

Команда, которая отсутствует в вашей системе, может быть установлена через менеджер пакетов вашей версии Linux. Менеджер пакетов — это программное обеспечение для установки программ Linux, которые поддерживаются вашей системой. Популярными менеджерами пакетов являются `apt`, `dnf`, `emerge`, `pacman`, `rpm`, `yum` и `zypper`. Используйте команду `man`, чтобы выяснить, какой менеджер пакетов установлен в вашей системе, и узнайте, как искать неустановленные пакеты. Часто требуется последовательность из двух команд: первой — для копирования последних данных о доступных пакетах (метаданных) из интернета в вашу систему, второй — для поиска метаданных. Например, для систем на базе *Ubuntu* или *Debian Linux* команды следующие:

```
$ sudo apt update           Скачать новейшие метаданные
$ apt-file search string     Искать строку string
```

Если после долгих поисков вы не нашли команду, отвечающую вашим задачам, обратитесь за помощью на онлайн-форум. Отличной отправной точкой, чтобы задавать правильные вопросы, является статья «*How do I ask a good question?*»

на форуме *Stack Overflow* (<https://stackoverflow.com/help/how-to-ask>). Формулируйте свои вопросы, уважая время других людей, тогда более опытные пользователи будут охотнее на них отвечать. Ваш вопрос должен быть кратким и по существу, включать сообщения об ошибках или другие выходные данные и объяснять, что вы уже пробовали сделать самостоятельно. Потратив время, чтобы сформулировать качественный вопрос, вы увеличите шансы на полезный ответ не только для себя, но и для других людей, которые столкнулись со схожей проблемой.

## Резюме

Теперь вы вышли за рамки инструментария размером с чайную ложку из главы 1 и готовы решать сложные задачи. Следующие главы наполнены практическими примерами использования команд, которые вы узнали, в самых разных ситуациях.

## ГЛАВА 6

---

# Родители, потомки и окружение

Назначение оболочки — выполнение команд — настолько фундаментально, что можно подумать, что оболочка встроена в Linux каким-то особым образом. Но это не так. Оболочка — это обычная программа, такая как `ls` или `cat`. Она запрограммирована на повторение следующих шагов снова и снова, снова и снова...

1. Вывести приглашение командной строки.
2. Прочитать команду из стандартного ввода.
3. Интерпретировать и запустить команду.

Linux прекрасно скрывает тот факт, что оболочка — это обычная программа. Когда вы входите в систему, Linux автоматически запускает для вас экземпляр оболочки, известный как *командная оболочка входа в систему* (*login shell*). Она запускается так плавно, что кажется, будто *это и есть сам Linux*, хотя на самом деле это просто программа, запущенная от вашего имени для взаимодействия с Linux.



### Где находится ваша оболочка входа?

Если вы входите в систему без графического интерфейса, скажем, с помощью клиентской программы SSH, оболочка входа в систему является начальной оболочкой, с которой вы взаимодействуете. Она печатает первое приглашение и ожидает вашей команды.

Если же вы используете версию с графическим интерфейсом, ваша оболочка входа в систему выполняется незаметно для вас. Она запускает среду рабочего стола, такую как GNOME, Unity, Cinnamon или KDE Plasma. Затем вы можете открыть окна терминала для запуска дополнительных интерактивных экземпляров оболочки.

Чем больше вы понимаете в устройстве оболочки, тем эффективнее сможете работать с Linux и тем меньше «магии» останется. В этой главе более детально, чем в главе 2, рассмотрены следующие загадки оболочки:

- Где находятся программы оболочки.
- Как разные экземпляры оболочки могут быть связаны друг с другом.
- Почему разные экземпляры оболочки могут иметь одни и те же переменные, значения, псевдонимы и другой контекст.
- Как изменить поведение оболочки по умолчанию, отредактировав файлы конфигурации.

К концу главы вы поймете, что все эти тайны не такие уж и загадочные.

## Оболочки — это исполняемые файлы

Оболочкой по умолчанию в большинстве систем Linux является `bash`<sup>1</sup>, и это обычная программа — исполняемый файл, расположенный в системном каталоге `/bin` вместе с `cat`, `ls`, `grep` и другими знакомыми командами:

```
$ cd /bin
$ ls -l bash cat ls grep
-rwxr-xr-x 1 root root 1113504 Jun 6 2019 bash
-rwxr-xr-x 1 root root 35064 Jan 18 2018 cat
-rwxr-xr-x 1 root root 219456 Sep 18 2019 grep
-rwxr-xr-x 1 root root 133792 Jan 18 2018 ls
```

Скорее всего, `bash` — не единственная возможная оболочка в вашей системе. Допустимые оболочки обычно перечислены в файле `/etc/shells`:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/bin/csh
/bin/zsh
```

Чтобы узнать, какую оболочку вы используете, примените команду `echo` к переменной оболочки `SHELL`:

```
$ echo $SHELL
/bin/bash
```

Теоретически система Linux может рассматривать любую программу как допустимую оболочку входа, если учетная запись пользователя настроена на вызов ее при входе в систему и она указана в `/etc/shells` (если это требуется в вашей системе). Обладая привилегиями суперпользователя, вы даже можете написать и установить свою собственную оболочку, такую, например, как в листинге 6.1.

---

<sup>1</sup> Если вы используете другую оболочку, обратите внимание на Приложение Б.



Она читает любую команду и отвечает: «Извините, боюсь, я не могу этого сделать». Эта пользовательская оболочка намеренно сделана глупой, но она демонстрирует, что другие программы могут быть такой же легитимной оболочкой, как */bin/bash*.

**Листинг 6.1.** *halshell*: оболочка, которая отказывается выполнять ваши команды

```
#!/bin/bash
# Вывод приглашения командной строки
echo -n '$ '
# Чтение ввода пользователя в цикле. Выход, когда пользователь нажимает Ctrl-D
while read line; do
# Игнорировать входную строку $line и вывести сообщение
echo "Извините, боюсь, я не могу этого сделать"
# Вывод следующего приглашения командной строки
echo -n '$ '
done
```

Поскольку *bash* — это просто программа, вы можете запустить ее вручную, как и любую другую команду:

```
$ bash
```

Если вы это сделаете, вы просто увидите еще одно приглашение, как будто ваша команда не имела никакого эффекта:

```
$
```

Но на самом деле вы запустили новый экземпляр *bash*. Этот новый экземпляр печатает приглашение и ожидает вашей команды. Чтобы как-то выделить новый экземпляр, измените его приглашение командной строки (скажем, на *%%*) с помощью переменной оболочки *PS1* и выполните несколько команд:

```
$ PS1="%% "
%% ls                                Приглашение изменилось
animals.txt
%% echo "This is a new shell"
This is a new shell
```

Теперь запустите команду *exit*, чтобы завершить работу нового экземпляра *bash*. Вы вернетесь к исходной оболочке, в которой используется приглашение со знаком доллара:

```
%% exit
$
```

Необходимо отметить, что изменение с *%%* обратно на *\$* не было просто изменением приглашения командной строки. Это была полная смена оболочки. Работа

нового экземпляра `bash` завершена, поэтому исходная оболочка запрашивает следующую команду.

Запуск `bash` вручную нужен не только для развлечения. Вы будете использовать вызываемые вручную экземпляры оболочки в главе 7.

## Родительский и дочерний процессы

Когда один экземпляр оболочки вызывает другой, как только что было показано, исходная оболочка называется *родительской*, а новый экземпляр — *дочерней*. То же самое верно для любой программы, которая вызывает другую программу Linux. Вызывающая программа является родительской, а вызываемая — дочерней. Работающая программа Linux называется *процессом*, поэтому вы также встретите термины *родительский процесс* и *дочерний процесс*, или *потомок*. Процесс может иметь любое количество потомков, но у каждого потомка есть только один родитель.

У каждого процесса есть свое окружение. Окружение, которое вы, возможно, помните из раздела «Окружение и файлы инициализации, краткая версия» на с. 50, включает в себя текущий каталог, путь поиска, приглашение оболочки и другую важную информацию, хранящуюся в переменных оболочки. Когда создается дочерний элемент, его окружение в значительной степени является копией окружения его родителя (подробности см. в разделе «Переменные окружения» на с. 125).

*Каждый раз, когда вы запускаете простую команду, вы создаете дочерний процесс. Это настолько важный момент для понимания Linux, что повторим еще раз: даже когда вы запускаете простую команду, такую как `ls`, она выполняется внутри нового дочернего процесса со своим собственным (скопированным) окружением. Это означает, что любые изменения, которые вы вносите в дочерний процесс, например изменение переменной приглашения `PS1` в дочерней оболочке, влияют только на дочерний процесс и теряются при выходе из него. Точно так же любые изменения в родительском элементе не повлияют на его дочерние элементы, которые уже запущены. Однако изменения в родительском объекте *могут* повлиять на его *будущие* дочерние элементы, поскольку окружение каждого дочернего объекта копируется при запуске из родительского.*

Почему важно, чтобы команды выполнялись в дочерних процессах? Прежде всего, любая программа, которую вы запускаете, может выполнять команду `cd` по всей файловой системе, но, когда она завершается, ваша текущая оболочка (родительская) не меняет свой текущий каталог. Проведем эксперимент, чтобы доказать это. Создайте в своем домашнем каталоге небольшой сценарий с именем `cdtest`, содержащий команду `cd`:

```
#!/bin/bash
cd /etc
echo "Here is my current directory:"
pwd
```

Сделайте его исполняемым:

```
$ chmod +x cctest
```

Выведите имя текущего каталога и запустите скрипт:

```
$ pwd
/home/smith
$ ./cctest
Here is my current directory:
/etc
```

Теперь проверьте ваш текущий каталог:

```
$ pwd
/home/smith
```

Ваш текущий каталог не изменился, хотя скрипт `cctest` переместился в каталог `/etc`. Это потому, что `cctest` запускается внутри дочернего процесса со своим собственным окружением. Изменения в дочернем окружении не могут повлиять на родительское, поэтому текущий каталог родителя не изменился. То же самое происходит, когда вы запускаете исполняемую программу, такую как `cat` или `grep`, — она запускается в дочернем процессе, который завершается после окончания работы программы, забирая с собой любые изменения окружения.



### Почему команда `cd` должна быть встроенной в оболочку

Если программы Linux не могут изменить текущий каталог вашей оболочки, то как команда `cd` может изменить его? Оказывается, `cd` — это не программа, а встроенная функция оболочки (*shell builtin*). Если бы `cd` была внешней по отношению к оболочке программой, изменения каталога были бы невозможны, так как они выполнялись бы в дочернем процессе и не могли бы повлиять на родительский процесс.

Конвейеры запускают несколько дочерних процессов: по одному для каждой команды в конвейере. Например, эта команда из раздела «Команда 6: `uniq`» на с. 30 запускает шесть потомков:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1 | cut -c9
```

## Переменные окружения

Каждый экземпляр оболочки имеет свой набор переменных, как вы узнали из раздела «Вычисление переменных» на с. 40. Некоторые переменные существуют только в одной конкретной оболочке. Они называются *локальными переменными*. Другие переменные автоматически копируются из оболочки во все ее дочерние элементы. Это *переменные окружения*, и все вместе они формируют окружение оболочки.

Некоторые примеры переменных окружения и их использования.

**HOME** — путь к вашему домашнему каталогу. Его значение автоматически устанавливается оболочкой при входе в систему. Текстовые редакторы, такие как **vim** и **emacs**, читают переменную **HOME**, чтобы найти свои файлы конфигурации (**\$HOME/.vim** и **\$HOME/.emacs** соответственно).

**PWD** — текущий каталог вашей оболочки. Его значение автоматически устанавливается и поддерживается оболочкой каждый раз, когда вы переходите в другой каталог с помощью команды **cd**. Команда **pwd** считывает переменную **PWD**, чтобы вывести имя текущего каталога вашей оболочки.

**EDITOR** — имя или путь предпочитаемого вами текстового редактора. Его значение обычно задается в файле конфигурации оболочки. Другие программы читают эту переменную, чтобы запустить соответствующий редактор от вашего имени.

Просмотрите переменные окружения вашей оболочки с помощью команды **printenv**. Вывод представляет собой несортированный набор строк — по одной переменной в строке, и может быть довольно длинным, поэтому используйте конвейер из **sort** и **less** для более удобного просмотра<sup>1</sup>:

```
$ printenv | sort -i | less
:
DISPLAY=:0
EDITOR=emacs
HOME=/home/smith
LANG=en_US.UTF-8
PWD=/home/smith/Music
SHELL=/bin/bash
TERM=xterm-256color
USER=smith
:
```

<sup>1</sup> Я сократил вывод, оставив только самые общие переменные оболочки. Ваш вывод, вероятно, намного длиннее и полон неясных имен переменных.

Локальные переменные не отображаются в выводе `printenv`. Проверьте их значения, поставив перед именем переменной знак доллара, с помощью команды `echo`:

```
$ title="Efficient Linux"
$ echo $title
Efficient Linux
$ printenv title
```

*(пустой вывод)*

## Создание переменных окружения

Чтобы превратить локальную переменную в переменную окружения, используйте команду `export`:

```
$ MY_VARIABLE=10
$ export MY_VARIABLE
$ export ANOTHER_VARIABLE=20
```

*Присвоим значение локальной переменной  
Экспортируем, чтобы она стала переменной окружения  
Все вместе в одной команде*

`export` означает, что переменная и ее значение будут скопированы из текущей оболочки во всех ее будущих потомков. Локальные переменные не копируются в будущие дочерние элементы:

```
$ export E="I am an environment variable"
$ L="I am just a local variable"
$ echo $E
I am an environment variable
$ echo $L
I am just a local variable
$ bash
$ echo $E
I am an environment variable
$ echo $L
$ exit
```

*Установить переменную оболочки  
Установить локальную переменную  
  
Запустить дочернюю оболочку  
Переменная оболочки была скопирована  
  
Локальная переменная не была скопирована  
Вывод пустой строки  
Выход из дочернего процесса*

Помните, что дочерние переменные — это *копии*. Любые изменения в копиях не влияют на родительскую оболочку:

```
$ export E="I am the original value"
$ bash
$ echo $E
I am the original value
$ E="I was modified in a child"
$ echo $E
I was modified in a child
$ exit
$ echo $E
I am the original value
```

*Установить переменную оболочки  
Запустить дочернюю оболочку  
  
Родительское значение было скопировано  
Изменить дочернюю копию  
  
Выход из дочерней оболочки  
  
Родительское значение переменной осталось без изменений*

Запустите новую оболочку и измените что-либо в ее окружении, но все изменения исчезнут, когда вы выйдете из оболочки. Это означает, что вы можете безопасно экспериментировать с функциями оболочки — просто запустите оболочку вручную, создав дочернюю, и завершите ее, когда закончите.

## Предупреждение о мифе: «глобальные» переменные

Иногда Linux слишком хорошо скрывает свою внутреннюю работу. Отличный пример — поведение переменных окружения. Каким-то образом, как по волшебству, такие переменные, как `HOME` и `PATH`, имеют постоянное значение во всех экземплярах вашей оболочки. В каком-то смысле они кажутся «глобальными переменными» (это утверждение встречается в других книгах по Linux, не выходящих в издательстве O'Reilly). Но переменная окружения *не является глобальной*. Каждый экземпляр оболочки имеет свою собственную копию. Изменение переменной окружения в одной оболочке не может изменить ее значение ни в одной другой запущенной оболочке. Модификации влияют только на будущих потомков этой оболочки (еще не вызванных).

Но каким образом такая переменная, как `HOME` или `PATH`, сохраняет свое значение во всех экземплярах вашей оболочки? Для этого есть две причины, которые проиллюстрированы на рис. 6.1. Если коротко:

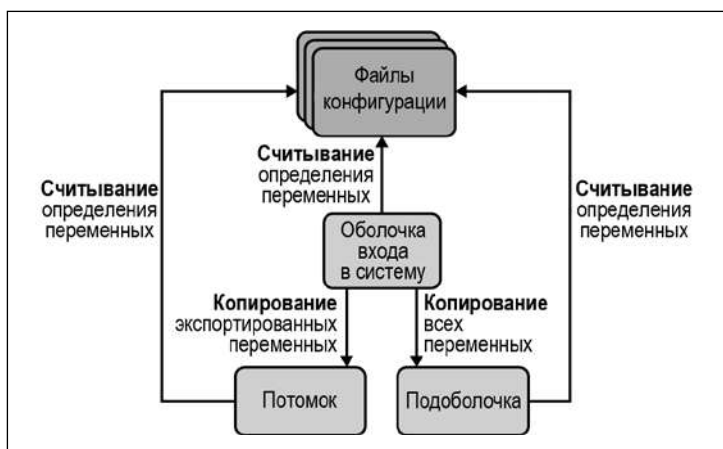
*Потомки копируют своих родителей.*

Значения таких переменных, как `HOME`, обычно устанавливаются и экспортируются вашей оболочкой входа в систему. Все будущие оболочки (пока вы не выйдете из системы) являются потомками оболочки входа в систему, поэтому они получают копию переменной и ее значение. Такого рода определяемые системой переменные окружения настолько редко изменяются в реальной работе, что кажутся глобальными, но на самом деле это обычные переменные, подчиняющиеся общим правилам (вы даже можете изменить их значения в работающей оболочке, но при этом возможно нарушение ожидаемого поведения этой командной оболочки и других программ).

*Разные экземпляры оболочки читают одни и те же файлы конфигурации.*

Значения локальных переменных, которые не копируются в дочерние элементы, могут быть установлены в файле конфигурации Linux, например `$HOME/.bashrc` (подробнее см. в разделе «Настройка окружения» на с. 129). Каждый экземпляр оболочки при вызове считывает и выполняет соответствующие файлы конфигурации. В результате эти локальные переменные кажутся копируемыми из оболочки в оболочку. То же самое относится и к другим неэкспортируемым сущностям оболочки, например псевдонимам.

Такое поведение заставляет некоторых пользователей полагать, что команда `export` создает глобальную переменную. Это не так. Команда `export WHATEVER` просто объявляет, что переменная `WHATEVER` будет скопирована из текущей оболочки во все будущие дочерние элементы.



**Рис. 6.1.** Экземпляры оболочки могут использовать одни и те же значения переменных благодаря экспорту или чтению общих файлов конфигурации

## Дочерние оболочки vs подоболочки

Потомок является частичной копией своего родителя. Например, он включает копии переменных окружения своего родителя, но не локальные (неэкспортированные) переменные или псевдонимы родителя:

<b>\$ alias</b>	<i>Список псевдонимов</i>
<b>alias gd='pushd'</b>	
<b>alias l='ls -CF'</b>	
<b>alias pd='popd'</b>	
<b>\$ bash --norc</b>	<i>Запустить дочернюю оболочку и игнорировать файл .bashrc</i>
<b>\$ alias</b>	<i>Список псевдонимов – ни один не известен</i>
<b>\$ echo \$HOME</b>	<i>Переменные окружения известны</i>
<b>/home/smith</b>	
<b>\$ exit</b>	<i>Выход из дочерней оболочки</i>

Если вы когда-нибудь задумывались, почему ваши псевдонимы недоступны в сценариях оболочки, теперь вы знаете ответ. Сценарии запускаются в дочернем элементе, который не получает копии псевдонимов своего родителя.

Подоболочка, напротив, является полной копией своего родителя<sup>1</sup>. Она включает в себя все родительские переменные, псевдонимы, функции и многое другое. Чтобы запустить команду в подоболочке, заключите ее в круглые скобки:

```
$ (ls -l)                  Запускает ls -l в подоболочке
-rw-r--r-- 1 smith smith 325 Oct 13 22:19 animals.txt
$ (alias)                  Просмотр псевдонимов в подоболочке
alias gd=pushd
alias l=ls -CF
alias pd=popd
:
$ (l)                     Запуск псевдонима от родителя
animals.txt
```

Чтобы проверить, является ли экземпляр оболочки подоболочкой, выведите переменную BASH\_SUBSHELL. В подоболочках значение будет отлично от нуля:

```
$ echo $BASH_SUBSHELL      Проверяем текущий экземпляр оболочки
0                           Это не подоболочка
$ bash                     Запускаем дочернюю оболочку
$ echo $BASH_SUBSHELL      Проверяем дочернюю оболочку
0                           Это не подоболочка
$ exit                     Выходим из дочерней оболочки
$ (echo $BASH_SUBSHELL)    Явно запускаем подоболочку
1                           Это подоболочка
```

Некоторые практические применения подоболочек мы рассмотрим в разделе «Способ 10: Явные подоболочки» на с. 159. А пока просто имейте в виду, что вы можете создавать их и они копируют псевдонимы родителя.

## Настройка окружения

Когда команда `bash` запускается, она настраивает себя, читая и выполняя последовательность *файлов конфигурации*. Эти файлы определяют переменные, псевдонимы, функции, другие свойства оболочки и могут запускать любую команду Linux (они похожи на сценарии настройки оболочки). В каталоге `/etc` файлы конфигурации определяются системным администратором и применяются ко всем пользователям системы. Другие файлы конфигурации принадлежат отдельным пользователям и изменяются ими. Они расположены в домашнем каталоге пользователя. В табл. 6.1 перечислены стандартные файлы конфигурации `bash`. Они делятся на нескольких типов:

<sup>1</sup> Копия полная, за исключением ловушек, которые «сбрасываются до значений, унаследованных оболочкой от своего родителя при вызове» (`man bash`). В этой книге мы не будем обсуждать ловушки.



*Файлы запуска*

Файлы конфигурации, которые выполняются автоматически при входе в систему, то есть они применяются только к вашей оболочке входа в систему. К примеру, команда в таком файле может установить и экспортировать переменную оболочки. Однако определение псевдонима в этом файле малополезно, поскольку псевдонимы не копируются в дочерние элементы.

*Файлы инициализации*

Файлы конфигурации, которые выполняются для каждого экземпляра оболочки, за исключением оболочки входа в систему, например когда вы вручную запускаете интерактивную оболочку или сценарий неинтерактивной оболочки. Команды файла инициализации могут установить переменную или определить псевдоним.

*Файлы очистки при выходе*

Файлы конфигурации, которые выполняются непосредственно перед выходом из оболочки входа в систему. Например, в таком файле может быть команда `clear`, очищающая экран при выходе из системы.

**Таблица 6.1.** Стандартные файлы конфигурации, используемые в `bash`

Тип файла	Чем запускается	Общесистемное местоположение	Расположение личных файлов (в порядке вызова)
Файлы запуска	Оболочка входа в систему (при вызове)	<code>/etc/profile</code>	<code>\$HOME/.bash_profile</code> , <code>\$HOME/.bash_login</code> , and <code>\$HOME/.profile</code>
Файлы инициализации	Интерактивная оболочка (без авторизации). Сценарии оболочки (при вызове)	<code>/etc/bash.bashrc</code> Задайте для переменной <code>BASH_ENV</code> абсолютный путь к файлу инициализации (пример: <code>BASH_ENV=/usr/local/etc/bashrc</code> )	<code>\$HOME/.bashrc</code> Задайте для переменной <code>BASH_ENV</code> абсолютный путь к файлу инициализации (пример: <code>BASH_ENV=/usr/local/etc/bashrc</code> )
Файлы очистки	Оболочка входа в систему (при выходе)	<code>/etc/bash.bash_logout</code>	<code>\$HOME/.bash_logout</code>

Обратите внимание, что у вас есть три варианта файлов запуска в вашем домашнем каталоге: `.bash_profile`, `.bash_login` и `.profile`. Большинство пользователей могут выбрать только один из них. Ваш дистрибутив Linux, вероятно,

уже предоставляет один из этих файлов предварительно заполненным полезными командами. Но, если вы используете не `bash`, а, например, `Bourne (/bin/sh)` или `Korn (/bin/ksh)`, они читают `.profile` и могут дать сбой, если этот файл содержит команды, специфичные для `bash`. Поэтому поместите специфичные для `bash` команды в `.bash_profile` или `.bash_login` (выберите один из них и используйте только его).

Иногда пользователей сбивает с толку возможность использования разных файлов запуска и инициализации. Почему вы хотите, чтобы ваша оболочка входа в систему вела себя иначе, чем другие оболочки, например те, которые вы открываете в нескольких окнах? Как правило, вам не нужно, чтобы они вели себя по-разному. Ваш файл запуска может делать немного больше, чем файл инициализации `$HOME/.bashrc`, тогда все интерактивные оболочки (требующие или не требующие входа в систему) будут иметь одинаковую конфигурацию.

Когда вы входите в графическое окружение рабочего стола (GNOME, Unity, KDE Plasma и т. д.), оболочка входа может быть скрыта. В этом случае оболочка входа в систему не имеет значения, так как вы взаимодействуете только с ее дочерними элементами. Поэтому вы можете поместить большую часть или даже всю конфигурацию в файл `$HOME/.bashrc`<sup>1</sup>. С другой стороны, если вы входите в систему из неграфической терминальной программы, такой, например, как SSH-клиент, то напрямую взаимодействуете со своей оболочкой входа, поэтому ее конфигурация имеет большое значение.

В каждом из перечисленных случаев, как правило, имеет смысл, чтобы ваш личный файл запуска имел в качестве источника файл инициализации:

```
# Поместите в $HOME/.bash_profile или другой персональный файл запуска
if [ -f "$HOME/.bashrc" ]
then
    source "$HOME/.bashrc"
fi
```

В любом случае старайтесь не помещать одинаковые команды в два разных файла конфигурации. Это приводит к путанице, такую систему трудно поддерживать, потому что любое изменение в одном файле надо не забыть продублировать в другом (и вы обязательно забудете, поверьте мне). Вместо этого используйте один файл в качестве источника для другого, как показано выше.

<sup>1</sup> Чтобы еще немного запутать ситуацию, некоторые окружения рабочего стола имеют свои собственные файлы конфигурации оболочки. Например, в GNOME есть `$HOME/gnomerc`, а в базовой системе X Window — `$HOME/xinitrc`.

## Повторное считывание файла конфигурации

Когда вы изменяете файлы запуска или инициализации, то можете заставить работающую оболочку повторно считать его, как описано в разделе «Окружения и файлы инициализации, краткая версия» на с. 50:

```
$ source ~/.bash_profile  Используется встроенная команда source  
$ . ~/.bash_profile       Используется точка
```



### Зачем нужно считывать файл конфигурации

Почему вы считываете файл конфигурации, вместо того чтобы сделать его исполняемым с помощью `chmod` и запустить как сценарий? Потому что сценарий запускается в дочернем процессе. Команды в сценарии не повлияют на вашу родительскую оболочку, а после завершения дочернего процесса все изменения будут потеряны.

## Путешествие с вашим окружением

Если вы используете несколько компьютеров с Linux в разных местах, в какой-то момент может понадобиться установить отложенные файлы конфигурации более чем на одну машину. Не копируйте отдельные файлы из одной системы в другую — такой подход в конечном итоге приведет к путанице. Вместо этого храните и обновляйте файлы в бесплатном сервисе GitHub (<https://github.com/>) или аналогичном, поддерживающем контроль версий. В этом случае вы сможете быстро загружать, устанавливать и обновлять файлы конфигурации на любом компьютере с Linux. Если вы допустили ошибку при редактировании файла конфигурации, то можете вернуться к предыдущей версии, выполнив одну или две команды. Контроль версий выходит за рамки этой книги. Чтобы узнать больше, см. раздел «Применяйте контроль версий к повседневным файлам» на с. 230.

Если вам не нравятся системы контроля версий вроде Git или Subversion, храните файлы конфигурации в облачном хранилище, таком как Dropbox, Google Drive или One-Drive. Обновление ваших файлов конфигурации будет менее удобным, но по крайней мере файлы будут легко доступны для копирования в другие системы Linux.

## Резюме

Многие пользователи Linux понятия не имеют о родительских и дочерних процессах, окружениях и назначении многих файлов конфигурации оболочки. Я надеюсь, что после прочтения главы у вас сложится четкое представление обо всех этих вещах. Оно понадобится в главе 7 как основа инструментария для гибкого выполнения команд.

## ГЛАВА 7

---

# Еще 11 способов запуска команды

Теперь, когда вы знаете множество команд и хорошо разбираетесь в оболочке, пришло время научиться... запускать команды. «Разве мы не запускали команды с самого начала книги?» — спросите вы. Ну да, но только двумя способами. Первый — это обычное выполнение простой команды:

```
$ grep Nutshell animals.txt
```

Второй — это конвейер простых команд, описанный в главе 1:

```
$ cut -f1 grades | sort | uniq -c | sort -nr
```

В этой главе мы рассмотрим еще 11 способов запуска команды и узнаем, почему их следует изучить. У каждого способа есть свои плюсы и минусы, и чем больше их вы знаете, тем гибче и эффективнее сможете взаимодействовать с Linux. Пока что ознакомимся с основами каждого способа. В следующих двух главах вы увидите более сложные примеры.

## Способы, использующие списки

Список — это последовательность команд, расположенных в одной командной строке. Вы уже видели один тип списка — конвейер, но оболочка поддерживает и другие, которые ведут себя не так, как конвейеры:

### *Условные списки*

Каждая последующая команда зависит от результата выполнения предыдущей.

### *Безусловные списки*

Команды выполняются одна за другой.

## Способ #1. Условные списки

Предположим, вы хотите создать файл *new.txt* в каталоге *dir*. Типичная последовательность команд может быть следующей:

```
$ cd dir                Перейти в каталог
$ touch new.txt         Создать файл
```

Обратите внимание, что запуск второй команды зависит от успешного выполнения первой. Если каталог *dir* не существует, нет смысла запускать команду *touch*. Оболочка позволяет сделать эту зависимость явной, поместив оператор *&&* (произносится как *и*) между двумя командами в одной строке:

```
$ cd dir && touch new.txt
```

Теперь вторая команда (*touch*) выполнится только в том случае, если первая команда (*cd*) выполнена успешно. Предыдущий пример представляет собой *условный список* из двух команд (чтобы узнать, что означает успешное выполнение команды, см. раздел «Коды возврата, указывающие на успех или неудачу» на с. 136).

Скорее всего, вы каждый день запускаете команды, которые зависят от выполнения предыдущих. Например, делали ли вы когда-нибудь резервную копию файла, а после изменения оригинала ее удаляли?

```
$ cp myfile.txt myfile.safe    Создать резервную копию
$ nano myfile.txt              Изменить оригинальный файл
$ rm myfile.safe               Удалить резервную копию
```

Каждая из этих команд имеет смысл только в том случае, если предыдущая выполнена успешно. Таким образом, эта последовательность команд является кандидатом для составления условного списка:

```
$ cp myfile.txt myfile.safe && nano myfile.txt && rm myfile.safe
```

Если вы используете систему контроля версий Git, то, вероятно, знакомы со следующей последовательностью команд после изменения некоторых файлов: запустить *git add*, чтобы подготовить файлы для снимка состояния (*commit*), затем *git commit* и, наконец, *git push*, чтобы поделиться изменениями. Если какая-либо из этих команд завершится ошибкой, остальные вы не запустите (пока не устраните причину ошибки). Поэтому эти три команды хорошо работают в формате условного списка:

```
$ git add . && git commit -m"fixed a bug" && git push
```

Так же, как оператор *&&* запускает вторую команду только в случае успеха первой, оператор *||* (произносится как *или*) запускает вторую команду только в случае

сбоя первой. Например, следующая команда пытается войти в каталог и, если это не удастся, создает его<sup>1</sup>:

```
$ cd dir || mkdir dir
```

Вы часто будете видеть оператор `||` в сценариях. В частности, он используется для выхода в случае возникновения ошибки:

```
# Если не удастся войти в каталог, выйдите с кодом ошибки 1
cd dir || exit 1
```

Объединяйте операторы `&&` и `||`, чтобы настраивать более сложные действия, которые используют успешные или неудачные попытки выполнить команду. Следующая команда пытается войти в каталог *dir*; если это не удастся, пробует создать каталог и войти в него; а если и это не удастся, выводит сообщение об ошибке:

```
$ cd dir || mkdir dir && cd dir || echo "I failed"
```

Команды в условном списке не обязательно должны быть простыми, это могут быть конвейеры и другие комбинированные команды.

### КОДЫ ВОЗВРАТА, УКАЗЫВАЮЩИЕ НА УСПЕХ ИЛИ НЕУДАЧУ

Что означает успешное или неудачное выполнение команды Linux? Каждая команда Linux при завершении выдает результат, называемый *кодом возврата (exit code)*. По соглашению, нулевой код возврата означает успех, а любое ненулевое значение — сбой<sup>2</sup>. Просмотрите код возврата последней выполненной команды оболочки, напечатав специальную переменную оболочки `?`:

```
$ ls myfile.txt
myfile.txt
$ echo $?
0
                                Вывести значение переменной ?
                                Ls выполнена успешно
$ cp nonexistent.txt somewhere.txt
cp: cannot stat 'nonexistent.txt': No such file or directory
$ echo $?
1
                                cp выполнена неудачно
```

<sup>1</sup> Команда `mkdir -p dir`, которая создает путь к каталогу, только если он еще не существует, была бы более элегантным решением для этого случая.

<sup>2</sup> Это противоположно многим языкам программирования, где ноль означает неудачу.

## Способ #2. Безусловные списки

Команды в списке не обязательно должны зависеть друг от друга. Если вы разделяете команды точкой с запятой, они просто выполняются по порядку. Успех или неудача одной команды не влияет на более поздние в списке.

Безусловные списки для запуска команд удобны, если надо уйти с работы на весь день. Вот пример команды, которая спит (ничего не делает) в течение двух часов (7200 секунд), а затем создает резервную копию всех важных файлов:

```
$ sleep 7200; cp -a ~/important-files /mnt/backup_drive
```

А вот аналогичная команда, которая работает как простейшая система напоминаний, спит пять минут, а затем отправляет вам электронное письмо<sup>3</sup>:

```
$ sleep 300; echo "remember to walk the dog" | mail -s reminder $USER
```

Безусловные списки — удобная функция, в большинстве случаев они дают те же результаты, что и ввод команд по отдельности и нажатие **Enter** после каждой. Единственное существенное различие касается кодов возврата. В безусловном списке коды возврата отдельных команд отбрасываются, кроме последней. Только код возврата последней команды в списке присваивается переменной оболочки `?`:

```
$ mv file1 file2; mv file2 file3; mv file3 file4
$ echo $?
0                                Код возврата команды «mv file3 file4»
```

## Способы, использующие подстановку

*Подстановка* означает автоматическую замену текста команды. Мы рассмотрим два ее типа:

### *Подстановка команд*

Команда заменяется ее выводом.

### *Подстановка процесса*

Команда заменяется файлом.

---

<sup>3</sup> В качестве альтернативы вы можете использовать **cron** для задания резервного копирования и команду **at** для напоминаний, но в Linux главное гибкость — наличие нескольких решений для достижения одного и того же результата.



## Способ #3. Подстановка команд

Предположим, у вас есть несколько тысяч текстовых файлов с песнями. Каждый файл содержит название песни, имя исполнителя, название альбома и текст песни:

```
Title: Carry On Wayward Son
Artist: Kansas
Album: Leftoverture
Carry on my wayward son
There'll be peace when you are done
:
```

Вы хотите распределить файлы в подкаталоги по исполнителям. Выполняя эту задачу вручную, вы можете найти все файлы песен исполнителя Kansas с помощью команды `grep`:

```
$ grep -l "Artist: Kansas" *.txt
carry_on_wayward_son.txt
dust_in_the_wind.txt
belexes.txt
```

а затем переместить каждый файл в каталог *kansas*:

```
$ mkdir kansas
$ mv carry_on_wayward_son.txt kansas
$ mv dust_in_the_wind.txt kansas
$ mv belexes.txt kansas
```

Утомительно, правда? Было бы неплохо, если бы вы могли сказать оболочке: «Переместите все файлы, содержащие строку *Artist: Kansas*, в каталог *kansas*». В терминах Linux требуется взять список имен из предыдущей команды `grep -l` и передать его команде `mv`. Что ж, вы можете сделать это с помощью функции оболочки, называемой *подстановкой команд*:

```
$ mv $(grep -l "Artist: Kansas" *.txt) kansas
```

Следующий синтаксис:

```
$(команда)
```

выполняет команду в круглых скобках и заменяет команду ее выводом. Таким образом, в предыдущей командной строке команда `grep -l` заменяется списком имен файлов, которые она выводит, как если бы вы вводили имена файлов следующим образом:

```
$ mv carry_on_wayward_son.txt dust_in_the_wind.txt belexes.txt kansas
```

Вы можете включать даже псевдонимы в подстановку команд, потому что она запускается в подоболочке, которая включает копии псевдонимов своего родителя.



### Специальные символы и подстановка команд

Предыдущий пример с `grep -l` прекрасно работает для большинства имен файлов, если они не содержат пробелы или другие специальные символы. оболочка интерпретирует эти символы перед передачей вывода команде `mv`, что может привести к неожиданным результатам. Например, если `grep -l` выведет *dust in the wind.txt*, оболочка будет рассматривать пробелы как разделители, а `mv` попытается переместить четыре несуществующих файла с именами *dust*, *in*, *the* и *wind.txt*.

Рассмотрим еще один пример. Предположим, у вас есть банковские выписки за несколько лет в формате PDF. Файлы имеют имена, включающие год, месяц и день выписки, например *eStmt\_2021-08-26.pdf* для даты 26 августа 2021 года<sup>1</sup>. Вы хотите просмотреть последнюю выписку в текущем каталоге. Это можно сделать вручную: просмотреть каталог, найти файл с самой последней датой (который будет последним файлом в списке) и открыть его с помощью программы просмотра PDF для Linux, такой как `okular`.

Но зачем делать всю эту ручную работу? Позвольте подстановке команд сэкономить ваше время. Создадим команду, которая выводит имя последнего файла PDF в каталоге:

```
$ ls eStmt*.pdf | tail -n1
```

и отправим его программе `okular` с помощью подстановки команд:

```
$ okular $(ls eStmt*.pdf | tail -n1)
```

Команда `ls` выводит список всех подходящих файлов, а `tail` — только последний из них, например *eStmt\_2021-08-26.pdf*. Подстановка команд помещает это единственное имя файла прямо в командную строку, как если бы вы набрали `okular eStmt_2021-08-26.pdf`.



Исходным синтаксисом для подстановки команд были обратные кавычки. Следующие две команды эквивалентны:

```
$ echo Today is $(date +%A).  
Today is Saturday.  
$ echo Today is `date +%A`.  
Today is Saturday.
```

<sup>1</sup> Выписки Bank of America до сих пор имеют такой формат.

Обратные кавычки поддерживаются большинством оболочек. Однако синтаксис `$()` проще для восприятия при использовании вложенных команд:

```
$ echo $(date +%A) | tr a-z A-Z          Одиночная команда
SATURDAY
echo Today is $( echo $(date +%A) | tr a-z A-Z )!  Вложенная команда
Today is SATURDAY!
```

В сценариях подстановка команд обычно используется для сохранения вывода команды в переменной:

```
переменная=$(команда)
```

Например, чтобы получить имена файлов, содержащих песни исполнителя Kansas, и сохранить их в переменной, используйте подстановку команд следующим образом:

```
$ kansasFiles=$(grep -l "Artist: Kansas" *.txt)
```

Вывод может состоять из нескольких строк, поэтому, чтобы сохранить все символы новой строки, убедитесь, что значение заключено в кавычки везде, где вы его используете:

```
$ echo "$kansasFiles"
```

## Способ #4. Подстановка процесса

Подстановка команд, которую вы только что видели, заменяет команду ее выводом в виде строки. Подстановка процесса также заменяет команду ее выводом, но обрабатывает вывод так, как если бы он был сохранен в файле. Поначалу это различие может показаться непонятным, поэтому разберемся в нем шаг за шагом.

Предположим, вы находитесь в каталоге файлов изображений JPEG с именами от *1.jpg* до *1000.jpg*, но некоторые файлы загадочным образом отсутствуют, и вы хотите выяснить, какие именно. Создайте такой каталог с помощью следующих команд:

```
$ mkdir /tmp/jpegs && cd /tmp/jpegs
$ touch {1..1000}.jpg
$ rm 4.jpg 981.jpg
```

Плохой способ найти отсутствующие файлы — составить список файлов из каталога, отсортированных по номерам, и искать пропущенные файлы глазами:

```
$ ls -1 | sort -n | less
1.jpg
2.jpg
3.jpg
5.jpg                                4.jpg пропущен
:
```

Более надежным автоматизированным решением станет сравнение существующих имен файлов с полным списком имен от *1.jpg* до *1000.jpg* с помощью команды *diff*. Один из способов добиться этого — использовать временные файлы. Сохраните отсортированные существующие имена файлов в одном временном файле *original-list*:

```
$ ls *.jpg | sort -n > /tmp/original-list
```

Затем выведите полный список имен файлов от *1.jpg* до *1000.jpg* в другой временный файл, *full-list*. Это можно сделать, сгенерировав целые числа от 1 до 1000 с помощью команды *seq* и добавив *.jpg* к каждой строке с помощью *sed*:

```
$ seq 1 1000 | sed 's/$/.jpg/' > /tmp/full-list
```

Сравните два временных файла с помощью команды *diff*, чтобы обнаружить, что *4.jpg* и *981.jpg* отсутствуют, а затем удалите временные файлы:

```
$ diff /tmp/original-list /tmp/full-list
3a4
> 4.jpg
979a981
> 981.jpg
$ rm /tmp/original-list /tmp/full-list    Очистить после завершения
```

Потребовалось довольно много шагов. Разве не было бы здорово сравнить два списка имен напрямую и не возиться с временными файлами? Проблема в том, что команда *diff* не может сравнить два списка из стандартного ввода, в качестве аргументов ей требуются файлы<sup>1</sup>. Подстановка процесса решает эту проблему. Она заставляет оба списка выглядеть для *diff* как файлы (раздел «Как работает подстановка процессов» на с. 143 содержит технические подробности). Синтаксис

<(команда)

---

<sup>1</sup> Технически команда *diff* может прочитать один список из стандартного ввода, если вы укажете тире в качестве имени файла. Два списка она прочитать не может.

запускает команду в подболочке и представляет ее вывод так, как будто он находится в файле. Например, следующее выражение представляет вывод `ls -1 | sort -n` в виде содержимого файла:

```
<(ls -1 | sort -n)
```

Вы можете проверить это командой `cat`:

```
$ cat <(ls -1 | sort -n)
1.jpg
2.jpg
:
```

Можно скопировать файл с помощью `cp`:

```
$ cp <(ls -1 | sort -n) /tmp/listing
$ cat /tmp/listing
1.jpg
2.jpg
:
```

и сравнить один файл с другим. Начните с двух команд, которые создали два временных файла:

```
ls *.jpg | sort -n
seq 1 1000 | sed 's/$/.jpg/'
```

Примените подстановку процессов, чтобы `diff` мог обрабатывать их как файлы, и вы получите тот же результат, что и раньше, но без использования временных файлов:

```
$ diff <(ls *.jpg | sort -n) <(seq 1 1000 | sed 's/$/.jpg/')
3a4
> 4.jpg
979a981
> 981.jpg
```

Очистите вывод, выполнив поиск строк, начинающихся с `>`, и удалив первые два символа с помощью `cut`. Теперь вы получите отчет о пропавших файлах:

```
$ diff <(ls *.jpg | sort -n) <(seq 1 1000 | sed 's/$/.jpg/') \
| grep '>' | cut -c3-
4.jpg
981.jpg
```

Подстановка процесса изменяет подход к использованию командной строки. Оказывается, что команды, которые читают только из файлов на диске,

могут читать и из стандартного ввода. Со временем команды, которые раньше казались невозможными, создаются сами собой.

### КАК РАБОТАЕТ ПОДСТАНОВКА ПРОЦЕССОВ

Когда операционная система Linux открывает файл на диске, он представляется целым числом, называемым *файловым дескриптором*. Подстановка процесса имитирует файл, запуская команду и связывая ее вывод с файловым дескриптором, поэтому с точки зрения программ вывод выглядит как файл на диске. Вы можете проверить дескриптор файла с помощью команды `echo`:

```
$ echo <(ls)
/dev/fd/63
```

В этом случае дескриптор файла для `<(ls)` равен 63, и он расположен в системном каталоге `/dev/fd`.

Кстати, `stdin`, `stdout` и `stderr` представлены файловыми дескрипторами 0, 1 и 2 соответственно. Вот почему перенаправление `stderr` имеет синтаксис `2>`.

Выражение `<(...)` создает файловый дескриптор для чтения. Похожее выражение `>(...)` создает файловый дескриптор для записи, но за 25 лет он мне ни разу не понадобился.

Подстановка процессов — это функция, не относящаяся к POSIX, поэтому она может быть отключена в вашей оболочке. Чтобы включить функции, отличные от POSIX, в вашем текущем экземпляре оболочки, запустите `set +o posix`.

## Команда как строка

Каждая команда представляет собой строку, но некоторые команды больше подходят под это представление, чем другие. Рассмотрим несколько способов создания строки по частям с последующим запуском ее как команды:

- Передача команды в `bash` в качестве аргумента.
- Передача команд в `bash` через стандартный ввод.
- Отправка команд на другой компьютер в сети с помощью `ssh`.
- Запуск последовательности команд с помощью `xargs`.



Следующие способы связаны с определенным риском, поскольку они отправляют неотображаемый текст в оболочку для выполнения. Никогда не делайте этого вслепую. Всегда проверяйте отправляемые данные. Вы ведь не хотите по ошибке выполнить строку `rm -rf $HOME` и стереть все ваши файлы.

## Способ #5. Передача команды в `bash` в качестве аргумента

`bash` — это обычная команда, такая же, как и любая другая, как описано в разделе «Оболочки — это исполняемые файлы» на с. 121, поэтому вы можете запускать ее по имени в командной строке. По умолчанию команда `bash`, как вы уже знаете, запускает интерактивную оболочку для ввода и выполнения команд. Кроме того, вы можете передать команду в `bash` в виде строки с помощью параметра `-c`, и `bash` запустит эту строку как команду, а после выполнения завершит работу:

```
$ bash -c "ls -l"
-rw-r--r-- 1 smith smith 325 Jul 3 17:44 animals.txt
```

Почему это бывает полезно? Потому что новый процесс `bash` будет дочерним со своим собственным окружением, включая текущий каталог, переменные и их значения и так далее. Любые изменения в дочерней оболочке не повлияют на вашу текущую. Например, команда `bash -c` поменяет каталог на `/tmp` только для того, чтобы удалить файл, а затем завершит работу:

```
$ pwd
/home/smith
$ touch /tmp/badfile          Создаем временный файл
$ bash -c "cd /tmp && rm badfile"
$ pwd
/home/smith                  Текущий каталог не изменился
```

Однако наиболее показательным и элегантно использование `bash -c`, когда вы запускаете определенные команды в качестве суперпользователя. В частности, сочетание `sudo` и перенаправления ввода/вывода приводит к интересной (иногда сводящей с ума) ситуации, в которой `bash -c` является ключом к успеху.

Предположим, вы хотите создать файл журнала в системном каталоге `/var/log`, недоступном для записи обычным пользователям. Вы запускаете следующую команду `sudo`, чтобы получить привилегии суперпользователя и создать файл журнала, но она загадочным образом не исполняется:

```
$ sudo echo "New log file" > /var/log/custom.log
bash: /var/log/custom.log: Permission denied
```

Минуту, команда `sudo` должна дать вам разрешение на создание любого файла в любом месте. Как эта команда может потерпеть неудачу? Почему `sudo` даже не запрашивает пароль? Ответ: потому что `sudo` вообще не запускалась. Вы применили `sudo` к команде `echo`, но не к перенаправлению вывода, которое запустилось первым и провалилось. Опишем подробно.

Вы нажали клавишу `Enter`.

Оболочка начала вычислять всю команду, включая перенаправление (`>`).

Оболочка попыталась создать файл `custom.log` в защищенном каталоге `/var/log`.

У вас не было разрешения на запись в `/var/log`, поэтому оболочка не смогла этого сделать и сообщила, что в доступе отказано (*Permission denied*).

Вот почему команда `sudo` даже не запускалась. Чтобы решить эту проблему, вам нужно сообщить оболочке: «Выполнить всю команду, включая перенаправление вывода, от имени суперпользователя». Это именно та ситуация, которую так хорошо решает `bash -c`. Создайте команду, которую вы хотите запустить, в виде строки

```
'echo "Новый log-файл" > /var/log/custom.log'
```

и передайте ее в качестве аргумента команде `sudo bash -c`:

```
$ sudo bash -c 'echo "New log file" > /var/log/custom.log'
[sudo] password for smith: xxxxxxxx
$ cat /var/log/custom.log
New log file
```

На этот раз вы запустили `bash`, а не просто `echo`, от имени суперпользователя, и `bash` выполняет всю строку как команду. Перенаправление проходит успешно. Помните об этом способе, когда `sudo` сочетается с перенаправлением.

## Способ #6. Передача команды в `bash` через стандартный ввод

Оболочка считывает каждую команду, которую вы вводите в стандартный ввод. Это означает, что `bash` может участвовать в конвейерах. Например, напечатайте строку `ls -l` и передайте ее в `bash`, который обработает строку как команду и запустит ее:

```
$ echo "ls -l"
ls -l
$ echo "ls -l" | bash
-rw-r--r-- 1 smith smith 325 Jul 3 17:44 animals.txt
```





Никогда не передавайте текст в **bash** вслепую. Будьте уверены в том, что вы выполняете.

Этот метод отлично подходит, когда вам нужно запустить много одинаковых команд подряд. Если вы можете представить команды в виде строк, то можете передать эти строки в **bash** для выполнения. Предположим, вы находитесь в каталоге с большим количеством файлов и хотите упорядочить их в подкаталоги по первому символу. Файл с именем *apple* будет перемещен в подкаталог *a*, файл с именем *cantaloupe* — в подкаталог *c* и так далее<sup>1</sup> (для простоты будем считать, что все имена файлов начинаются со строчной буквы и не содержат пробелов или специальных символов).

Сначала отсортируем файлы в список. Предположим, что все имена имеют длину не менее двух символов (соответствуют шаблону `??*`), чтобы наши команды не конфликтовали с именами подкаталогов от *a* до *z*:

```
$ ls -l ??*
apple
banana
cantaloupe
carrot
:
```

Создадим 26 подкаталогов с помощью расширения фигурных скобок:

```
$ mkdir {a..z}
```

Теперь сгенерируем команды `mv` в виде строк. Начнем с регулярного выражения для `sed`, которое захватывает первый символ имени файла при помощи выражения #1 (`\1`):

```
^\(.\)
```

Захватим остальную часть имени файла с помощью выражения #2 (`\2`):

```
\(.*\) $
```

Соединим вместе эти два выражения:

```
^\(.\) \(.*\) $
```

Теперь сформируем команду `mv` со словом *mv*, за которым следует пробел, полное имя файла (`\1\2`), еще один пробел и первый символ (`\1`):

<sup>1</sup> Эта структура каталогов похожа на хеш-таблицу.

```
mv \1\2 \1
```

Полученная команда выглядит так:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2 \1/'
mv apple a
mv banana b
mv cantaloupe c
mv carrot c
:
```

Ее вывод содержит именно те команды `mv`, которые нам нужны. Убедимся в правильности выведенных данных. Возможно, потребуется перенаправить их в команду `less` для страничного просмотра:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2\t\1/' | less
```

Убедившись, что команды верны, отправьте вывод в `bash` для выполнения:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2\t\1/' | bash
```

Шаги, которые мы только что выполнили, составляют шаблон:

1. Сформируйте последовательность команд.
2. Проверьте результаты с помощью команды `less`.
3. Передайте вывод в `bash`.

## Способ #7. Удаленное выполнение однострочника с помощью `ssh`

*Внимание:* этот метод будет иметь смысл только в случае, если вы знакомы с SSH, безопасной оболочкой для входа на удаленные хосты. Настройка соединения SSH между хостами выходит за рамки этой книги, поэтому обратитесь к руководству по SSH.

В дополнение к обычному способу входа на удаленный хост:

```
$ ssh myhost.example.com
```

вы можете выполнять отдельные команды на удаленном хосте, передавая их в виде строк в `ssh`. Просто добавьте строку в конец команды `ssh`:

```
$ ssh myhost.example.com ls
remotefile1
remotefile2
remotefile3
```

Этот метод обычно быстрее, чем вход в систему, запуск команды и выход из системы. Если команда использует специальные символы, например символы перенаправления, которые необходимо интерпретировать на удаленном хосте, заключайте их в кавычки или экранируйте. В противном случае они будут интерпретироваться вашей локальной оболочкой. Следующие две команды запускают `ls` удаленно, но перенаправление вывода происходит на разных хостах:

```
$ ssh myhost.example.com ls > outfile      Создает outfile на локальном хосте
$ ssh myhost.example.com "ls > outfile"    Создает outfile на удаленном хосте
```

Помимо этого, вы можете передавать команды в `ssh` для запуска на удаленном хосте так же, как в `bash` для локального запуска:

```
$ echo "ls > outfile" | ssh myhost.example.com
```

При передаче команд в `ssh` удаленный хост может вывести диагностические или другие сообщения. Как правило, они не влияют на удаленную команду, и вы можете избавиться от них:

- Если вы видите сообщения о псевдотерминалах (*pseudo-ttys*), например *Pseudoterminal will not be allocated because stdin is not a terminal* (псевдо-терминал не будет выделен, поскольку стандартный ввод не является терминалом), запустите `ssh` с параметром `-T`, чтобы удаленный SSH-сервер не выделял терминал:

```
$ echo "ls > outfile" | ssh -T myhost.example.com
```

- Если вы видите приветственные сообщения, которые обычно появляются при входе в систему (*Welcome to Linux!*) или другие нежелательные сообщения, попробуйте указать команде `ssh` явно запустить `bash` на удаленном хосте, и сообщения должны исчезнуть:

```
$ echo "ls > outfile" | ssh myhost.example.com bash
```

## Способ #8. Запуск списка команд с помощью `xargs`

Многие пользователи Linux никогда не слышали о команде `xargs`, хотя это мощный инструмент для создания и запуска нескольких похожих команд. Знакомство с `xargs` стало еще одним переломным моментом в моем освоении Linux, и я надеюсь, что оно станет таковым и для вас.

`xargs` обрабатывает входные данные из двух источников:

- стандартного потока ввода: списки строк, разделенных пробелом. Примером могут служить пути к файлам, создаваемые `ls` или `find`, но подойдут любые строки. Назовем их *входными строками*;
- командной строки: неполные команды, в которых отсутствуют некоторые аргументы (назовем их *шаблонами команд*).

`xargs` объединяет входные строки с шаблоном команды для создания и запуска новых полноценных команд, которые будем называть *сгенерированными командами*.

Рассмотрим простой пример. Предположим, вы находитесь в каталоге с тремя файлами:

```
$ ls -l
apple
banana
cantaloupe
```

Передадим список каталогов в `xargs`, чтобы он служил входными строками, и `wc -l` в качестве шаблона команды:

```
$ ls -l | xargs wc -l
3 apple
4 banana
1 cantaloupe
8 total
```

Как и было обещано, `xargs` применила шаблон команды `wc -l` к каждой из входных строк, подсчитывая строки в каждом файле. Чтобы вывести те же три файла с помощью `cat`, просто измените шаблон команды:

```
$ ls -l | xargs cat
```

У этих примеров есть два существенных недостатка: фатальный и практический. Фатальный недостаток заключается в том, что `xargs` может сделать что-то неправильно, если входная строка содержит специальные символы, например пробелы. Надежное решение находится в разделе «Безопасность с помощью `find` и `xargs`» на с. 151.

Практический недостаток заключается в том, что в данном примере не требуется `xargs`, можно решить те же задачи проще с помощью сопоставления файлов с шаблоном:

```
$ wc -l *
3 apple
4 banana
1 cantaloupe
8 total
```

Зачем тогда использовать команду `xargs`? Ее мощь становится очевидной, когда входные строки сложнее, чем простой список каталогов. Предположим, вы хотите рекурсивно посчитать количество строк во всех файлах каталога и *всех его подкаталогах*, но только для исходных файлов Python с именами, оканчивающимися на `.py`. Такой список путей к файлам легко создать с помощью команды `find`:

```
$ find . -type f -name \*.py -print
fruits/raspberry.py
vegetables/leafy/lettuce.py
:
```

Теперь `xargs` может применить шаблон команды `wc -l` к каждому пути к файлу, создавая рекурсивный результат, который было бы трудно получить другим способом. В целях безопасности заменим опцию `-print` на `-print0`, а `xargs` на `xargs -0` (см. врезку «Безопасность при использовании `find` и `xargs`» на с. 151):

```
$ find . -type f -name \*.py -print0 | xargs -0 wc -l
6 ./fruits/raspberry.py
3 ./vegetables/leafy/lettuce.py
:
```

Комбинируя `find` и `xargs`, можно дать возможность любой команде рекурсивно выполняться с обходом всей файловой системы, затрагивая только те файлы и/или каталоги, которые соответствуют указанным критериям (в некоторых случаях можно получить тот же эффект просто с помощью `find`, используя параметр `-exec`, но `xargs` часто является более правильным решением).

Команда `xargs` имеет множество опций (см. `man xargs`), которые управляют тем, как она создает и запускает сгенерированные команды. На мой взгляд, наиболее важными (кроме `-0`) являются `-n` и `-I`. Параметр `-n` определяет, сколько аргументов добавляется с помощью `xargs` к каждой сгенерированной команде (по умолчанию добавляется столько аргументов, сколько позволят ограничения оболочки<sup>1</sup>):

<code>\$ ls   xargs echo</code>	<i>Уместить как можно больше входных строк:</i>
apple banana cantaloupe carrot	<code>echo apple banana cantaloupe carrot</code>
<code>\$ ls   xargs -n1 echo</code>	<i>Один аргумент в каждой команде echo:</i>
apple	<code>echo apple</code>
banana	<code>echo banana</code>
cantaloupe	<code>echo cantaloupe</code>
carrot	<code>echo carrot</code>

<sup>1</sup> Точное число зависит от ограничения длины строки в вашей системе Linux, см. `man xargs`.

```
$ ls | xargs -n2 echo
apple banana
cantaloupe carrot
$ ls | xargs -n3 echo
apple banana cantaloupe
carrot echo carrot
```

*Два аргумента в каждой команде echo:*  
*echo apple banana*  
*echo cantaloupe carrot*  
*Три аргумента в каждой команде echo:*  
*echo apple banana cantaloupe*

### БЕЗОПАСНОСТЬ ПРИ ИСПОЛЬЗОВАНИИ FIND И XARGS

При объединении `find` и `xargs` используйте `xargs -0` (ноль) для защиты на случай неожиданных специальных символов во входных строках. А для вывода применяйте `find -print0`:

```
$ find параметры... -print0 | xargs -0 параметры...
```

Обычно `xargs` ожидает, что входные строки будут разделены пробелами. А если сами входные строки содержат другие пробелы, например имена файлов с пробелами в них? По умолчанию `xargs` будет рассматривать эти пробелы как разделители ввода и обрабатывать неполные строки, выдавая неверные результаты. Например, если входные данные для `xargs` содержат строку *prickly pear.py*, `xargs` воспримет ее как две входные строки и, скорее всего, выведет ошибку:

```
prickly: No such file or directory
pear.py: No such file or directory
```

Поэтому используйте `xargs -0`, чтобы разделителем служил нулевой символ (ноль в ASCII). Нули редко появляются в тексте, поэтому они являются идеальными разделителями для входных строк.

Как разделить входные строки нулями вместо символов новой строки? К счастью, у команды `find` есть возможность сделать это: используйте `-print0` вместо `-print`.

Команда `ls`, к сожалению, не может использовать ноль в качестве разделителя, поэтому предыдущие простые примеры с `ls` небезопасны. Вы можете преобразовать символы новой строки в нули с помощью команды `tr`:

```
$ ls | tr '\n' '\0' | xargs -0 ...
```

Или используйте псевдоним, который выводит список файлов и каталогов с записями, разделенными нулями:

```
alias ls0="find . -maxdepth 1 -print0"
```

Параметр `-I` определяет место входных строк в сгенерированной команде. По умолчанию они добавляются к шаблону команды, но вы можете настроить их отображение в другом месте. После `-I` введите любую строку (по вашему выбору), и она станет прототипом, указывающим, куда следует вставлять входные строки:

```
$ ls | xargs -I XYZ echo XYZ is my favorite food    XYZ в качестве прототипа
apple is my favorite food
banana is my favorite food
cantaloupe is my favorite food
carrot is my favorite food
```

Строка `XYZ` расположена после команды `echo`, что перемещает входную строку в начало каждой выходной строки. Обратите внимание, что параметр `-I` ограничивает `xargs` одной входной строкой на сгенерированную команду. Изучите справочную страницу `xargs`, чтобы узнать, что еще вы можете контролировать.



### Длинные списки аргументов

`xargs` позволяет решать проблему чрезмерно длинных командных строк. Предположим, что ваш текущий каталог содержит миллион файлов с именами от `file1.txt` до `file1000000.txt`, и вы пытаетесь удалить их путем сопоставления с шаблоном:

```
$ rm *.txt
bash: /bin/rm: Argument list too long
```

Шаблон `*.txt` вычисляется как строка из более чем 14 миллионов символов, что превышает ограничение Linux. Чтобы обойти его, передайте список файлов в `xargs` для удаления. `xargs` разделит список файлов на несколько команд `rm`. Сформируйте список файлов с помощью `grep`, обрабатывая только имена файлов, заканчивающиеся на `.txt`, а затем передайте его в `xargs`:

```
$ ls | grep '\.txt$' | xargs rm
```

Это решение лучше, чем сопоставление с шаблоном файлов (`ls *.txt`), которое приведет к той же ошибке *Argument list too long*. Еще лучше запустить `find print0`, как описано в разделе «Безопасность при использовании `find` и `xargs`» на с. 151:

```
$ find . -maxdepth 1 -name \*.txt -type f -print0 \
  | xargs -0 rm
```

## Способы, использующие управление процессами

Все обсуждавшиеся ранее команды выполняются в родительской оболочке. Давайте рассмотрим несколько способов, которые ведут себя в этом отношении иначе:

### *Фоновые команды*

Возвращают приглашение командной строки и выполняются в фоновом режиме.

### *Явные подоболочки*

Могут быть запущены в середине составной команды.

### *Замена процесса*

Заменяют родительскую оболочку.

## Способ #9. Фоновое выполнение команды

До сих пор во всех способах, пока команда выполнялась, вы ждали, когда появится приглашение командной строки. Его появление означало, что команда завершила работу. Но ждать этого не обязательно, особенно для команд, выполнение которых занимает много времени. Вы можете запускать команды особым образом, чтобы они исчезали из поля зрения (в каком-то смысле), но продолжали выполняться, немедленно освобождая текущую оболочку для других команд. Этот метод называется *фоновым выполнением команды (backgrounding)*. Если же команда занимает оболочку, ее называют *командой переднего плана (foreground)*. Экземпляр оболочки одновременно запускает не более одной команды переднего плана и любое количество фоновых команд.

### Запуск команды в фоновом режиме

Чтобы запустить команду в фоновом режиме, просто добавьте амперсанд (&). Оболочка отвечает сообщением, свидетельствующим, что команда запущена в фоновом режиме, и выводит приглашение командной строки:

```
$ wc -c my_extremely_huge_file.txt &  
[1] 74931  
$
```

*Подсчет символов в огромном файле  
Ответ оболочки*

Затем вы можете продолжить выполнение команд переднего плана или других фоновых команд в этой оболочке. Вывод фоновых команд может появиться



в любое время, даже когда вы печатаете. Если фоновая команда завершится успешно, оболочка выведет сообщение *Done*:

```
59837483748 my_extremely_huge_file.txt
[1]+ Done          wc -c my_extremely_huge_file.txt
```

Если команда не выполнится, вы увидите сообщение о выходе с кодом возврата:

```
[1]+ Exit 1 wc -c my_extremely_huge_file.txt
```



**Амперсанд является оператором списка, как && и ||:**

```
$ command1 & command2 & command3 &
[1] 57351
[2] 57352
[3] 57353
$ command4 & command5 & echo hi
[1] 57431
[2] 57432
hi
```

*Все три команды выполняются фоном*

*Все команды, кроме echo, выполняются фоном*

## Приостановка и перевод команды в фоновый режим

Можно запустить команду переднего плана, а потом передумать и отправить ее в фоновый режим. Нажмите **Ctrl-Z**, чтобы временно остановить выполнение команды. В появившемся приглашении командной строки введите **bg**, чтобы возобновить выполнение команды в фоновом режиме.

## Задания и управление ими

Фоновые команды являются частью функции оболочки, называемой *управлением заданиями*. Она позволяет манипулировать запущенными командами, например переводить их в фоновый режим, приостанавливать и возобновлять. *Задание* — это единица работы оболочки, один экземпляр команды, выполняемой в оболочке. Простые команды, конвейеры и условные списки — все это примеры заданий, которые можно запускать из командной строки.

Задание — это больше, чем просто процесс Linux. Задание может состоять из одного или нескольких процессов. Конвейер из шести программ, например, представляет собой одно задание, включающее (как минимум) шесть процессов. Задания — это конструкция оболочки. Linux отслеживает не задания, а только лежащие в их основе процессы.

В оболочке может быть запущено одновременно несколько заданий. Каждое задание имеет положительный целочисленный идентификатор, называемый также

номером задания. Когда вы запускаете команду в фоновом режиме, оболочка печатает номер задания и идентификатор первого процесса, запущенного в задании. В следующем примере номер задания равен 1, а идентификатор процесса — 74931:

```
$ wc -c my_extremely_huge_file.txt &
[1] 74931
```

Общие команды управления заданиями

Оболочка имеет встроенные команды для управления заданиями, перечисленные в табл. 7.1. Рассмотрим наиболее распространенные операции управления заданиями. Для начала запустим команду `sleep`, которая ничего не делает (спит) в течение заданного количества секунд, а затем завершает работу. Например, `sleep 10` спит в течение 10 секунд.

Таблица 7.1. Команды управления заданиями

Команда	Значение
<code>bg</code>	Переместить текущее приостановленное задание в фоновый режим
<code>bg %n</code>	Переместить приостановленное задание номер <code>n</code> в фоновый режим (пример: <code>bg %1</code> )
<code>fg</code>	Переместить текущее фоновое задание на передний план
<code>fg %n</code>	Переместить фоновое задание номер <code>n</code> на передний план (пример: <code>fg %2</code> )
<code>kill %n</code>	Завершить фоновое задание номер <code>n</code> (пример: <code>kill %3</code> )
<code>jobs</code>	Просмотр списка заданий оболочки

Запустим задание в фоновом режиме до его завершения:

```
$ sleep 20 &                                Запуск в фоновом режиме
[1] 126288
$ jobs                                       Вывод списка заданий
[1]+  Running                  sleep 20 &
$
...Завершение...
[1]+  Done                     sleep 20
```



По завершении задания сообщение *Done* может не появиться до тех пор, пока вы в очередной раз не нажмете **Enter**.

Запустим фоновое задание и переведем его на передний план:

```
$ sleep 20 &                                Запуск в фоновом режиме
[1] 126362
$ fg                                         Перемещение на передний план
sleep 20
...Завершение...
$
```

Запустим задание переднего плана, приостановим его и вернем снова на передний план:

```
$ sleep 20                                Запуск задания на переднем плане
^Z                                         Прерывание выполнения
[1]+  Stopped                  sleep 20
$ jobs                                    Вывод списка заданий
[1]+  Stopped                  sleep 20
$ fg                                     Перемещение на передний план
sleep 20
...Завершение...
[1]+  Done                     sleep 20
```

Запустим задание переднего плана и отправим его в фоновый режим:

```
$ sleep 20                                Запуск в фоновом режиме
^Z                                         Прерывание выполнения
[1]+  Stopped                  sleep 20
$ bg                                     Перемещение в фоновый режим
[1]+  sleep 20 &
$ jobs                                    Вывод списка заданий
[1]+  Running                  sleep 20 &
$
...Завершение...
[1]+  Done                     sleep 20
```

Попробуем с несколькими фоновыми заданиями. Обратимся к заданию по его номеру, которому предшествует знак процента (%1, %2 и т. д.):

```
$ sleep 100 &                             Запуск трех команд в фоновом режиме
[1] 126452
$ sleep 200 &
[2] 126456
$ sleep 300 &
[3] 126460
$ jobs                                    Вывод списка заданий
[1]  Running                  sleep 100 &
[2]-  Running                  sleep 200 &
[3]+  Running                  sleep 300 &
$ fg %2                                  Перемещение задания 2 на передний план
sleep 200
```

```

^Z                                Прерывание выполнения задания 2
[2]+ Stopped                    sleep 200
$ jobs                          Задание 2 приостановлено (stopped)
[1]  Running                    sleep 100 &
[2]+ Stopped                    sleep 200
[3]- Running                    sleep 300 &
$ kill %3                        Завершение работы задания 3
[3]+ Terminated               sleep 300
$ jobs                          Задание 3 исчезло из списка
[1]- Running                    sleep 100 &
[2]+ Stopped                    sleep 200
$ bg %2                          Возобновление приостановленного задания 2
                                  в фоновом режиме
$ jobs                          Задание 2 снова запущено
[1]- Running                    sleep 100 &
[2]+ Running                    sleep 200 &
$

```

## Вывод и ввод в фоновом режиме

Фоновая команда может использовать стандартный вывод, но иногда это может произойти в самый неудобный момент. Что произойдет, если вы отсортируете файл словаря Linux длиной 100 000 строк и зададите вывод на экран двух первых строк в фоновом режиме? Вначале оболочка печатает номер задания (1), идентификатор процесса (81089) и приглашение к вводу:

```

$ sort /usr/share/dict/words | head -n2 &
[1] 81089
$

```

Когда задание завершится, оно выведет две строки, где бы ни находился курсор в этот момент, например:

```

$ sort /usr/share/dict/words | head -n2 &
[1] 81089
$ A
A's

```

Нажмите **Enter** — и оболочка сообщит, что задание выполнено:

```

[1]+ Done sort /usr/share/dict/words | head -n2
$

```

Вывод на экран при выполнении фонового задания может появиться в любое время. Чтобы избежать беспорядка, перенаправьте *stdout* в файл, а затем просмотрите его, когда вам будет удобно:

```

$ sort /usr/share/dict/words | head -n2 > /tmp/results &
[1] 81089

```

```
$
[1]+ Done sort /usr/share/dict/words | head -n2 > /tmp/results
$ cat /tmp/results
A
A's
$
```

Неожиданности случаются и тогда, когда фоновое задание пытается читать со стандартного ввода. Оболочка приостанавливает задание, печатает сообщение *Stopped* и ожидает ввода. Продемонстрируем это, запустив команду `cat` в фоновом режиме без аргументов, чтобы она читала стандартный ввод:

```
$ cat &
[1] 82455
[1]+ Stopped cat
```

Задания не могут читать ввод в фоновом режиме, поэтому сначала переведем его на передний план с помощью `fg`, а затем введем данные:

```
$ fg
cat
Here is some input
Here is some input
:
```

После ввода всех данных выполните одно из следующих действий:

- Продолжайте выполнять команду на переднем плане, пока она не завершится.
- Приостановите и снова запустите команду, нажав **Ctrl-Z**, а затем **bg**.
- Завершите ввод с помощью **Ctrl-D** или завершите команду с помощью **Ctrl-C**.

## Советы по работе с фоновым режимом

Фоновый режим идеально подходит для команд, выполнение которых занимает много времени, таких, например, как обработка текста во время длительных сеансов редактирования, и программ, которые открывают собственные окна. Например, программисты могут сэкономить много времени, приостанавливая работу своего текстового редактора, вместо того чтобы выходить из него. Я видел, как опытные инженеры правили код в текстовом редакторе, сохраняли результат и закрывали редактор, тестировали код, затем перезапускали редактор и искали место в коде, на котором они остановились. При такой организации процесса они теряют 10–15 секунд на смену задания каждый раз, когда выходят из редактора. Если бы вместо этого они приостанавливали работу редактора (**Ctrl-Z**), тестировали свой код и возобновляли работу редактора (**fg**), то избежали бы ненужной траты времени.

Фоновый режим также отлично подходит для запуска последовательности команд с использованием условного списка. Если какая-либо команда в списке завершается неудачно, остальные не будут выполняться и задание заканчивается:

```
$ command1 && command2 && command3 &
```

Но будьте внимательны с командами, которые считывают ввод, так как они могут приостановить выполнение текущего задания и ждать ввод!

## Способ #10. Явные подболочки

Каждый раз, когда мы запускаем простую команду, она выполняется в дочернем процессе, как мы видели в разделе «Родительский и дочерний процессы» на с. 123. Подстановка команд и подстановка процессов создают подболочки. Однако бывают случаи, когда полезно явно запустить дополнительную подболочку. Для этого просто заключим команду в круглые скобки, и она запустится в подболочке:

```
$ (cd /usr/local && ls)
```

```
bin etc games lib man sbin share
```

```
$ pwd
```

```
/home/smith
```

*cd /usr/Local выполнено в подболочке*

Применительно ко всей команде этот метод не очень полезен, за исключением, возможно, того, что нам не нужно запускать вторую команду `cd` для возврата в предыдущий каталог. Однако если мы заключим в круглые скобки только часть комбинированной команды, то получим интересные возможности. Типичным примером является конвейер, который меняет каталог в середине команды. Предположим, мы загрузили архив *package.tar.gz* и хотим извлечь файлы. Команда `tar` в этом случае выглядит так:

```
$ tar xvf package.tar.gz
```

```
Makefile
src/
src/defs.h
src/main.c
:
```

Извлечение файлов происходит в текущий каталог<sup>1</sup>. Но что делать, если мы хотим извлечь их в другой каталог? Можно сначала перейти туда и запустить `tar` (а затем вернуться), но также можно выполнить эту задачу с помощью одной команды. Хитрость заключается в том, чтобы передать `tar`-данные

---

<sup>1</sup> Предполагается, что `tar`-архив был создан с относительными, а не абсолютными путями, что типично для скачиваемого программного обеспечения.

в подоболочку, которая выполняет операции с каталогами и запускает `tar` при чтении из стандартного ввода<sup>1</sup>:

```
$ cat package.tar.gz | (mkdir -p /tmp/other && cd /tmp/other && tar xzvf -)
```

Этот метод также работает для копирования файлов из каталога `dir1` в другой существующий каталог `dir2` с использованием двух процессов `tar`, один из которых записывает в стандартный вывод, а другой читает из стандартного ввода:

```
$ tar czf - dir1 | (cd /tmp/dir2 && tar xvf -)
```

Тот же метод может копировать файлы в существующий каталог на другом хосте через SSH:

```
$ tar czf - dir1 | ssh myhost '(cd /tmp/dir2 && tar xvf -)'
```

### КАКИЕ ИЗ ИЗУЧЕННЫХ СПОСОБОВ СОЗДАЮТ ПОДОБОЛОЧКИ?

Многие способы, описанные в этой главе, запускают подболочку, которая наследует родительскую среду (переменные и их значения) плюс другой контекст оболочки, такой как псевдонимы. Другие методы запускают только дочерний процесс. Самый простой способ различить их — вычислить переменную `BASH_SUBSHELL`, которая будет отличной от нуля для подболочки и равна нулю в противном случае. Дополнительные сведения см. в разделе «Дочерние оболочки vs подболочки» на с. 128.

<code>\$ echo \$BASH_SUBSHELL</code>	<i>Обычное выполнение</i>
<code>0</code>	<i>Не подболочка</i>
<code>\$ (echo \$BASH_SUBSHELL)</code>	<i>Явная подболочка</i>
<code>1</code>	<i>Подболочка</i>
<code>\$ echo \$(echo \$BASH_SUBSHELL)</code>	<i>Подстановка команды</i>
<code>1</code>	<i>Подболочка</i>
<code>\$ cat &lt;(echo \$BASH_SUBSHELL)</code>	<i>Подстановка процесса</i>
<code>1</code>	<i>Подболочка</i>
<code>\$ bash -c 'echo \$BASH_SUBSHELL'</code>	<i>bash -c</i>
<code>0</code>	<i>Не подболочка</i>



Заманчиво рассматривать круглые скобки `bash` так, как если бы они просто группировали команды вместе, подобно скобкам в арифметике. Но это не так. Каждая пара скобок вызывает запуск подболочки.

<sup>1</sup> Эту конкретную задачу можно решить более просто с помощью параметра `tar -C` или `--directory`, указывающего целевой каталог.

## Способ #11. Замена процесса

Обычно оболочка запускает команду в отдельном процессе, который уничтожается при выходе из команды, как описано в разделе «Родительский и дочерний процессы» на с. 123. Это поведение можно изменить с помощью встроенной в оболочку команды `exes`. Она заменяет запущенную оболочку (процесс) другой командой (процессом) по вашему выбору. Когда такая команда завершится, приглашение оболочки не появится, потому что исходная оболочка исчезла.

Чтобы продемонстрировать такую замену, запустим новую оболочку вручную и изменим ее приглашение:

<b>\$ bash</b>	<i>Запустим дочернюю оболочку</i>
<b>\$ PS1="Doomed&gt; "</b>	<i>Изменим приветствие командной строки</i>
Doomed> echo hello	<i>Запустим любую команду на ваш выбор</i>
hello	

Теперь выполним команду `exes` и посмотрим, как завершит работу новая оболочка:

Doomed> <b>exes ls</b>	<i>Ls заменяет дочернюю оболочку, запускается и завершает работу</i>
animals.txt	
<b>\$ A</b>	<i>Приветствие начальной (родительской) оболочки</i>



### Запуск команды `exes` может быть фатальным

Если вы запустите `exes` в оболочке, оболочка после этого закроется. Если оболочка была запущена в окне терминала, терминал закроется. Если оболочка была оболочкой входа в систему, вы выйдете из системы.

Зачем вообще запускать `exes`? Одна из причин — чтобы экономить ресурсы, не запуская второй процесс. Сценарии оболочки иногда используют эту оптимизацию, запуская `exes` для последней команды в сценарии. Если сценарий запускается много раз (скажем, миллионы или миллиарды выполнений), экономия может оказаться оправданной.

Также у команды `exes` есть вторая функциональность — она может переназначать `stdin`, `stdout` и/или `stderr` для текущей оболочки. Это наиболее применимо в сценариях оболочки. Как, например, в следующем довольно простом сценарии, который печатает информацию в файл `/tmp/outfile`:

```
#!/bin/bash
echo "My name is $USER" > /tmp/outfile
echo "My current directory is $PWD" >> /tmp/outfile
```



```
echo "Guess how many lines are in the file /etc/hosts?" >> /tmp/outfile
wc -l /etc/hosts >> /tmp/outfile
echo "Goodbye for now" >> /tmp/outfile
```

Вместо перенаправления вывода каждой команды в */tmp/outfile* по отдельности используйте *exes* для перенаправления *stdout* в */tmp/outfile* для всего скрипта. Последующие команды могут использовать стандартный вывод:

```
#!/bin/bash
# Перенаправление вывода для всего скрипта
exes > /tmp/outfile2
# Все последующие команды печатаются в /tmp/outfile2
echo "My name is $USER"
echo "My current directory is $PWD"
echo "Guess how many lines are in the file /etc/hosts?"
wc -l /etc/hosts
echo "Goodbye for now"
```

Запустите этот сценарий и проверьте файл */tmp/outfile2*, чтобы увидеть результаты:

```
$ cat /tmp/outfile2
My name is smith
My current directory is /home/smith
Guess how many lines are in the file /etc/hosts?
122 /etc/hosts
Goodbye for now
```

Вы, вероятно, не будете использовать команду *exes* часто, но и не забывайте о ее возможностях.

## Резюме

Теперь вы знаете 13 способов выполнения команды — 11 из этой главы плюс простые команды и конвейеры. В табл. 7.2 рассматриваются некоторые распространенные варианты использования различных методов.

**Таблица 7.2.** Распространенные идиомы для запуска команд

Задача	Решение
Отправка <i>stdout</i> из одной программы в <i>stdin</i> другой	Конвейеры
Вставка вывода ( <i>stdout</i> ) в команду	Подстановка команды

Задача	Решение
Предоставление вывода ( <i>stdout</i> ) команде, которая читает не из <i>stdin</i> , а из файла	Подстановка процесса
Выполнение одной строки как команды	<code>bash -c</code> , или канал с участием <code>bash</code>
Вывод нескольких команд в <i>stdout</i> и их выполнение	Конвейер с участием <code>bash</code>
Выполнение множества похожих команд подряд	<code>xargs</code> или создание команд в виде строк и передача их в <code>bash</code>
Управление командами, которые зависят от успеха друг друга	Условные списки
Запуск нескольких команд одновременно	Фоновый режим работы
Одновременный запуск нескольких команд, которые зависят от успеха друг друга	Условные списки в фоновом режиме
Запуск одной команды на удаленном хосте	Запуск <code>ssh host command</code>
Изменение каталога в середине конвейера	Явные подболочки
Выполнение команды позже	Безусловный список с командой <code>sleep</code> , за которым следует основная команда
Перенаправление в/из защищенных файлов	Запуск <code>sudo bash -c "command &gt; file"</code>

Следующие две главы научат вас комбинировать эти методы для эффективного решения поставленных задач.

## ГЛАВА 8

---

# Создание дерзких однострочников

Помните эту длинную запутанную команду из предисловия?

```
$ paste <(echo {1..10}.jpg | sed 's/ /\n/g') \  
        <(echo {0..9}.jpg | sed 's/ /\n/g') \  
      | sed 's/^mv /' \  
      | bash
```

Такие «магические заклинания» принято называть *дерзкими однострочниками* (*brash one-liners*)<sup>1</sup>. Давайте разберем этот код, чтобы понять, что он делает и как работает. Команды `echo` используют расширение фигурных скобок для создания списков имен файлов JPEG:

```
$ echo {1..10}.jpg  
1.jpg 2.jpg 3.jpg ... 10.jpg  
$ echo {0..9}.jpg  
0.jpg 1.jpg 2.jpg ... 9.jpg
```

Команда `sed` заменяет в именах файлов символы пробела символами новой строки:

```
$ echo {1..10}.jpg | sed 's/ /\n/g'  
1.jpg  
2.jpg  
:  
10.jpg  
$ echo {0..9}.jpg | sed 's/ /\n/g'  
0.jpg
```

---

<sup>1</sup> Самое раннее использование этого термина (известное мне) — это `man`-страница для команды `lorder(1)` в BSD Unix 4.x (<https://www.unix.com/man-page/bsd/1/lorder>). Спасибо Бобу Бернсу (Bob Byrnes) за то, что нашел его. *Примеч. пер.:* здесь имеется в виду игра слов *bash* (самая популярная оболочка Linux) — *brash* (дерзкий); и *one-liner* — острота, а также однострочная команда (однострочник).

```
1.jpg
:
9.jpg
```

Команда `paste` соединяет два списка. Подстановка процесса позволяет команде `paste` читать два списка, как если бы они были файлами:

```
$ paste <(echo {1..10}.jpg | sed 's/ /\n/g') \
      <(echo {0..9}.jpg | sed 's/ /\n/g') \
1.jpg  0.jpg
2.jpg  1.jpg
:
10.jpg  9.jpg
```

Добавление `mv` к каждой строке выводит последовательность строк, являющихся командами `mv`:

```
$ paste <(echo {1..10}.jpg | sed 's/ /\n/g') \
      <(echo {0..9}.jpg | sed 's/ /\n/g') \
      | sed 's/^/mv /'
mv 1.jpg  0.jpg
mv 2.jpg  1.jpg
:
mv 10.jpg  9.jpg
```

Назначение однострочника теперь раскрыто: он генерирует 10 команд для переименования файлов изображений с *1.jpg* по *10.jpg*. Новые имена — от *0.jpg* до *9.jpg* соответственно. Передачу вывода в `bash` выполняют команды `mv`:

```
$ paste <(echo {1..10}.jpg | sed 's/ /\n/g') \
      <(echo {0..9}.jpg | sed 's/ /\n/g') \
      | sed 's/^/mv /' \
      | bash
```

Дерзкие однострочники похожи на головоломки, они бросают вызов вашим творческим способностям и позволяют оттачивать навыки.

В этой главе вы будете создавать дерзкие однострочники, подобные предыдущему примеру, шаг за шагом, используя следующую волшебную формулу:

1. Придумайте команду, которая решает часть задачи.
2. Запустите команду и проверьте вывод.
3. Вызовите команду из истории повторно и внесите необходимые правки.
4. Повторяйте шаги 2 и 3, пока ваша команда не даст желаемый результат.

Эта глава будет настоящей тренировкой для вашего мозга. Не удивляйтесь, что временами вы будете озадачены примерами. Двигайтесь шаг за шагом и запускайте команды на компьютере по мере их прочтения.



Некоторые дерзкие однострочники в этой главе слишком длинны для одной строки, поэтому разделены на несколько строк обратной косой чертой. Мы, однако, не станем называть их «дерзкими двустрочниками» или «дерзкими семистрочниками».

## Приготовьтесь быть дерзкими

Прежде чем вы начнете создавать дерзкие однострочники, найдите время, чтобы настроиться на правильный лад:

- Будьте гибкими.
- Подумайте, с чего начать.
- Изучите инструменты тестирования.

Рассмотрим каждую из этих идей по очереди.

### Будьте гибкими

Ключом к написанию дерзких однострочников является *гибкость*. К этому моменту вы изучили несколько замечательных инструментов — базовый набор программ Linux (и множество способов их запуска), а также использование истории команд, редактирование командной строки и многое другое. Вы можете комбинировать эти инструменты разными способами, и у каждой задачи обычно есть несколько решений.

Даже самые простые задачи Linux можно решить разными способами. Подумайте, как вывести на экран все файлы *.jpg* в вашем текущем каталоге. Готов поспорить, что 99,9% пользователей Linux запустили бы такую команду:

```
$ ls *.jpg
```

Но это лишь одно из многих решений. Например, вы можете перечислить все файлы в каталоге, а затем использовать *grep* для поиска только тех имен, которые заканчиваются на *.jpg*:

```
$ ls | grep '\.jpg$'
```

Почему стоит выбрать именно это решение? Вы видели пример в разделе «Длинные списки аргументов» на с. 152, когда каталог содержал так много файлов, что их нельзя было перечислить с помощью сопоставления с шаблоном. Техника поиска расширения имени файла является надежным общим подходом к решению всех видов задач. Но важно быть гибким и понимать свои инструменты, чтобы

вы могли применить лучший из них в нужное время. Можно сказать, что это некое волшебство, которое вы используете при создании дерзких однострочников.

Все следующие команды отображают файлы *.jpg* в текущем каталоге. Попробуйте разобраться, как работает каждая команда:

```
$ echo $(ls *.jpg)
$ bash -c 'ls *.jpg'
$ cat <(ls *.jpg)
$ find . -maxdepth 1 -type f -name \*.jpg -print
$ ls > tmp && grep '\.jpg$' tmp && rm -f tmp
$ paste <(echo ls) <(echo \*.jpg) | bash
$ bash -c 'exec $(paste <(echo ls) <(echo \*.jpg))'
$ echo 'monkey *.jpg' | sed 's/monkey/ls/' | bash
$ python -c 'import os; os.system("ls *.jpg")'
```

Одинаковы ли результаты? Можно ли придумать другие варианты команд для этой задачи?

## Подумайте, с чего начать

Каждый дерзкий однострочник начинается с вывода простой команды. Этот вывод может быть содержимым файла, частью файла, списком каталогов, последовательностью цифр или букв, списком пользователей, датой и временем или другими данными. Таким образом, ваша первая задача состоит в том, чтобы определить исходные данные для вашей команды.

Например, если вы хотите узнать 17-ю букву английского алфавита, ваши исходные данные могут состоять из 26 букв, полученных путем расширения фигурных скобок:

```
$ echo {A..Z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Когда вы получите этот результат, следующим шагом будет решить, как манипулировать этими данными, чтобы достичь вашей цели. Вам нужно разделить вывод на строки или столбцы? Соединить вывод с другой информацией? Преобразовать вывод более сложным способом? Чтобы выполнить это задание, обратитесь к программам из глав 1 и 5, например *grep*, *sed* и *cut*, и примените их, используя приемы из главы 7.

В этом примере вы можете напечатать 17-е поле с помощью *awk* или удалить пробелы с помощью *sed* и найти 17-й символ с помощью *cut*:

```
$ echo {A..Z} | awk '{print $(17)}'
Q
```

```
$ echo {A..Z} | sed 's/ //g' | cut -c17
Q
```

Если вы хотите напечатать месяцы года, ваши исходные данные могут быть числами от 1 до 12, полученными также с помощью расширения фигурных скобок:

```
$ echo {1..12}
1 2 3 4 5 6 7 8 9 10 11 12
```

Далее измените расширение фигурной скобки, чтобы оно формировало даты для первого дня каждого месяца (с 01.01.2021 по 01.12.2021). Затем запустите `date -d` в каждой строке, чтобы получить названия месяцев:

```
$ echo 2021-{01..12}-01 | xargs -n1 date +%B -d
January
February
March
:
December
```

Или предположим, что вы хотите узнать количество символов в самом длинном имени файла в текущем каталоге. Ваши исходные данные могут быть списком каталогов:

```
$ ls
animals.txt cartoon-mascots.txt ... zebra-stripes.txt
```

Используйте `awk` для подсчета символов в каждом имени файла с помощью `wc -c`:

```
$ ls | awk '{print "echo -n", $0, "| wc -c"}'
echo -n "animals.txt" | wc -c
echo -n "cartoon-mascots.txt" | wc -c
:
echo -n "zebra-stripes.txt" | wc -c
```

Здесь параметр `-n` не позволяет команде `echo` печатать символы новой строки, которые мешали бы правильному подсчету. Наконец, передайте команды в `bash` для их запуска, отсортируйте числовые результаты от большего к меньшему и выведите максимальное (первое) значение с помощью `head -n1`:

```
$ ls | awk '{print "echo -n", $0, "| wc -c"}' | bash | sort -nr | head -n1
23
```

Последний пример был сложным, так как решение включает конвейеры, генерирующие строки с передачей их следующему конвейеру. Тем не менее общий принцип тот же: определитесь с исходными данными и манипулируйте ими в соответствии со своими задачами.

## Изучите инструменты тестирования

Создание дерзкого однострочника может быть сопряжено со многими проблемами и ошибками. Следующие инструменты и методы помогут вам быстро опробовать различные решения.

*Используйте историю команд и редактирование командной строки.*

Не вводите команды повторно во время эксперимента. Используйте приемы из главы 3, чтобы вызывать предыдущие команды, настраивать их и запускать.

*Добавляйте echo, чтобы проверить свои выражения.*

Если вы не уверены, как будет вычислено выражение, используйте команду `echo`, чтобы увидеть результаты вычислений в стандартном выводе.

*Используйте ls или добавьте echo для проверки деструктивных команд.*

Если ваша команда вызывает `rm`, `mv`, `cp` или другие команды, которые могут перезаписывать или удалять файлы, поместите перед ними `echo`, чтобы контролировать, какие файлы будут затронуты (например, вместо `rm` выполните команду `echo rm`). Еще один способ обезопасить себя — заменить `rm` на `ls` для получения списка файлов, которые будут удалены.

*Вставляйте tee для просмотра промежуточных результатов.*

Если вы хотите контролировать вывод (`stdout`) в середине длинного конвейера, вставьте команду `tee`, чтобы сохранить вывод в файл. Следующая команда сохраняет вывод `command3` в файле `outfile`, одновременно перенаправляя его в `command4`:

```
$ command1 | command2 | command3 | tee outfile | command4 | command5
$ less outfile
```

Теперь давайте создадим несколько дерзких однострочников!

## Вставка имени файла в последовательность

Этот однострочник похож на тот, что был в начале главы (переименование файлов `.jpg`), но более детализирован. И это также реальная ситуация, с которой я столкнулся, когда писал эту книгу. Как и предыдущий однострочник, он сочетает в себе два метода из главы 7: подстановку процесса и передачу команды в `bash` через стандартный ввод. В результате можно получить шаблон для решения подобных задач.



Я написал эту книгу на компьютере с Linux, используя язык набора текста AsciiDoc (<https://asciidoc.org/>). Детали языка здесь неважны, важно то, что каждая глава была отдельным файлом, и изначально их было 10:

```
$ ls
ch01.asciidoc ch03.asciidoc ch05.asciidoc ch07.asciidoc ch09.asciidoc
ch02.asciidoc ch04.asciidoc ch06.asciidoc ch08.asciidoc ch10.asciidoc
```

В какой-то момент я решил вставить дополнительную главу между второй и третьей. Это требовало переименования некоторых файлов. Главы 3–10 становились 4–11, а перед ними добавлялась новая глава 3 (*ch03.asciidoc*). Я мог бы переименовать файлы вручную, начиная с *ch11.asciidoc* и двигаясь в обратном направлении<sup>1</sup>:

```
$ mv ch10.asciidoc ch11.asciidoc
$ mv ch09.asciidoc ch10.asciidoc
$ mv ch08.asciidoc ch09.asciidoc
:
$ mv ch03.asciidoc ch04.asciidoc
```

Но этот метод утомителен (представьте, если бы было 1000 файлов вместо 11!), и вместо этого я написал необходимые команды `mv` и передал их в `bash`. Внимательно посмотрите на предыдущие команды `mv` и подумайте, как бы вы их написали.

Начнем с исходных имен файлов от *ch03.asciidoc* до *ch10.asciidoc*. Их можно напечатать, используя расширение фигурных скобок, такое как `ch{10..03}.asciidoc`, как в первом примере в этой главе, но для большей гибкости используем команду `seq -w` для вывода чисел:

```
$ seq -w 10 -1 3
10
09
08
:
03
```

Затем превратим эту числовую последовательность в имена файлов, передав ее в `sed`:

```
$ seq -w 10 -1 3 | sed 's/\(.*\)/ch\1.asciidoc/'
ch10.asciidoc
ch09.asciidoc
:
ch03.asciidoc
```

---

<sup>1</sup> Начинать с *ch03.asciidoc* и двигаться в прямом направлении было бы опасно, понимаете, почему? Если нет, создайте эти файлы с помощью команды `touch ch{01..10}.asciidoc` и попробуйте сами.

Теперь у нас есть список исходных имен файлов. Сделаем то же самое для глав 4–11, чтобы создать требуемые имена файлов:

```
$ seq -w 11 -1 4 | sed 's/\(.*\)\/ch\1.asciidoc/'
ch11.asciidoc
ch10.asciidoc
:
ch04.asciidoc
```

Чтобы сформировать команды `mv`, нужно напечатать исходное и новое имена файлов рядом. В первом примере в этой главе проблема «напечатать рядом» решалась с помощью `paste` и использовалась подстановка процесса для обработки двух напечатанных списков как файлов. Здесь сделаем так же:

```
$ paste <(seq -w 10 -1 3 | sed 's/\(.*\)\/ch\1.asciidoc/') \
      <(seq -w 11 -1 4 | sed 's/\(.*\)\/ch\1.asciidoc/') \
ch10.asciidoc ch11.asciidoc
ch09.asciidoc ch10.asciidoc
:
ch03.asciidoc ch04.asciidoc
```



Предыдущая команда выглядит так, словно вам нужно напечатать очень много текста, но, используя историю команд и редактирование командной строки в стиле Emacs, этого можно избежать. Чтобы перейти от простой команды `seq` и `sed` к команде `paste`, необходимо выполнить следующее:

1. Вызвать предыдущую команду из истории с помощью стрелки вверх.
2. Нажать `Ctrl-A`, а затем `Ctrl-K`, чтобы вырезать всю строку.
3. Ввести слово `paste`, а затем пробел.
4. Дважды нажать `Ctrl-Y`, чтобы создать две копии команд `seq` и `sed`.
5. Использовать сочетания клавиш для перемещения и редактирования, чтобы изменить вторую копию.

И так далее.

Добавляйте `mv` к каждой строке, направляя вывод в `sed`:

```
$ paste <(seq -w 10 -1 3 | sed 's/\(.*\)\/ch\1.asciidoc/') \
      <(seq -w 11 -1 4 | sed 's/\(.*\)\/ch\1.asciidoc/') \
      | sed 's/^/mv /'
mv ch10.asciidoc      ch11.asciidoc
mv ch09.asciidoc      ch10.asciidoc
:
mv ch03.asciidoc      ch04.asciidoc
```

В качестве последнего шага передайте команды в `bash` для выполнения:

```
$ paste <(seq -w 10 -1 3 | sed 's/\(.*\)/ch\1.asciidoc/') \
      <(seq -w 11 -1 4 | sed 's/\(.*\)/ch\1.asciidoc/') \
      | sed 's/^/mv /' \
      | bash
```

Я использовал именно это решение для своей книги. После выполнения команд `mv` итоговыми файлами стали главы 1, 2 и 4–11, оставив место для новой главы 3:

```
$ ls ch*.asciidoc
ch01.asciidoc ch04.asciidoc ch06.asciidoc ch08.asciidoc ch10.asciidoc
ch02.asciidoc ch05.asciidoc ch07.asciidoc ch09.asciidoc ch11.asciidoc
```

Описанный шаблон действий можно использовать во всех ситуациях для запуска последовательности связанных команд:

1. Передайте аргументы команды в виде списков в стандартный вывод.
2. Выведите списки столбцами с помощью `paste` и подстановки процесса.
3. Добавьте `sed` перед именем команды, заменив символ начала строки (^) именем программы и пробелом.
4. Передайте результаты в `bash`.

## Проверка совпадающих пар файлов

Этот однострочник вдохновлен практикой использования Mediawiki, программного обеспечения, на котором работает Wikipedia и тысячи других вики. Mediawiki позволяет пользователям загружать изображения для просмотра. Большинство пользователей делают это вручную через веб-формы: нажимают «Выбрать файл», чтобы открыть диалоговое окно, выбирают файл, добавляют небольшое описание в форму и нажимают «Загрузить». Администраторы вики используют сценарий, который считывает весь каталог и загружает изображения из него. Каждый файл изображения (скажем, *bald\_eagle.jpg*) связан с текстовым файлом (*bald\_eagle.txt*), содержащим описание изображения.

Представьте, что имеется каталог с сотнями файлов изображений и текста. Вы хотите убедиться, что каждому файлу изображения соответствует текстовый файл, и наоборот. Вот уменьшенная версия этого каталога:

```
$ ls
bald_eagle.jpg blue_jay.jpg cardinal.txt robin.jpg wren.jpg
bald_eagle.txt cardinal.jpg oriole.txt robin.txt wren.txt
```

Давайте придумаем два разных решения для выявления несоответствий файлов. Для первого решения создадим два списка: для файлов JPEG и текстовых файлов. Потом используем `cut`, чтобы удалить расширения `.txt` и `.jpg`:

```
$ ls *.jpg | cut -d. -f1
bald_eagle
blue_jay
cardinal
robin
wren
$ ls *.txt | cut -d. -f1
bald_eagle
cardinal
oriole
robin
wren
```

Затем сравним списки с помощью `diff`, используя подстановку процесса:

```
$ diff <(ls *.jpg | cut -d. -f1) <(ls *.txt | cut -d. -f1)
2d1
< blue_jay
3a3
> oriole
```

На этом можно было бы остановиться, потому что в выводе указано, что в первом списке есть непарный файл `blue_jay` (т. е. `blue_jay.jpg`), а во втором списке — `oriole` (т. е. `oriole.txt`). Тем не менее давайте сделаем результаты более информативными. Удалим ненужные строки, выполнив поиск символов `<` и `>` в начале каждой строки:

```
$ diff <(ls *.jpg | cut -d. -f1) <(ls *.txt | cut -d. -f1) \
| grep '^[<>]'
< blue_jay
> oriole
```

Затем используем `awk`, чтобы добавить правильное расширение к каждому имени файла (`$2`), в зависимости от того, какой символ предшествует имени файла, `<` или `>`:

```
$ diff <(ls *.jpg | cut -d. -f1) <(ls *.txt | cut -d. -f1) \
| grep '^[<>]' \
| awk '/^</{print $2 ".jpg"} /^>/{print $2 ".txt"}'
blue_jay.jpg
oriole.txt
```

Теперь у нас есть список непарных файлов. Однако в этом решении есть небольшая ошибка. Предположим, что текущий каталог содержит имя файла

*yellow.canary.jpg*, в котором есть две точки. Предыдущая команда выдаст неправильный результат:

```
blue_jay.jpg
oriole.txt
yellow.jpg
```

Это неверно

Эта проблема возникает из-за того, что две команды `cut` удаляют символы, начиная с первой, а не последней точки, поэтому *yellow.canary.jpg* усекается до *yellow*, а не до *yellow.canary*. Для решения проблемы заменим `cut` на `sed`, чтобы удалить символы от последней точки до конца строки:

```
$ diff <(ls *.jpg | sed 's/\.[^.]*$//') \
      <(ls *.txt | sed 's/\.[^.]*$//') \
    | grep '^[<>]' \
    | awk '/</{print $2 ".jpg"} />/{print $2 ".txt"}'
blue_jay.txt
oriole.jpg
yellow.canary.txt
```

Первое решение готово.

Второе решение использует другой подход. Вместо применения `diff` к двум спискам создадим один список и отсеем совпадающие пары имен файлов. Начнем с удаления расширений файлов с помощью `sed` (используя тот же сценарий `sed`, что и раньше) и подсчитаем количество вхождений каждой строки с помощью `uniq -c`:

```
$ ls *.{jpg,txt} \
  | sed 's/\.[^.]*$//' \
  | uniq -c
    2 bald_eagle
    1 blue_jay
    2 cardinal
    1 oriole
    2 robin
    2 wren
    1 yellow.canary
```

Каждая строка вывода содержит либо число 2, представляющее совпадающую пару имен файлов, либо 1, представляющее несовпадающее имя файла. Используем `awk`, чтобы изолировать строки, начинающиеся с пробела и 1, и вывести только второе поле:

```
$ ls *.{jpg,txt} \
  | sed 's/\.[^.]*$//' \
  | uniq -c \
  | awk '/^ *1 /{print $2}'
```

```
blue_jay
oriole
yellow.canary
```

Как теперь добавить отсутствующие расширения файлов? Не нужно сложных манипуляций со строками, просто используем `ls`, чтобы получить список файлов в текущем каталоге. Вставим звездочку (\*) в конец каждой строки вывода с помощью `awk`:

```
$ ls *.{jpg,txt} \
| sed 's/\.[^.]*$//' \
| uniq -c \
| awk '/^ *1 /{print $2 ""}'
blue_jay*
oriole*
yellow.canary*
```

Затем передадим строки в `ls` через подстановку команд. Оболочка выполнит сопоставление с шаблоном, и `ls` перечислит несопоставленные имена файлов. Готово!

```
$ ls -1 $(ls *.{jpg,txt} \
| sed 's/\.[^.]*$//' \
| uniq -c \
| awk '/^ *1 /{print $2 ""}')
blue_jay.jpg
oriole.txt
yellow.canary.jpg
```

## Создание CDPATH из вашего домашнего каталога

В разделе «Организируйте свой домашний каталог для быстрой навигации» на с. 77 мы вручную написали сложное выражение для переменной `CDPATH`. Оно начинается с `$HOME`, за которым следуют все подкаталоги `$HOME`, и заканчивается относительным путем `..` (родительский каталог):

```
CDPATH=$HOME:$HOME/Work:$HOME/Family:$HOME/Finances:$HOME/Linux:$HOME/Music:..
```

Давайте разработаем однострочник для автоматического создания этой строки `CDPATH`, чтобы ее можно было вставить в файл конфигурации `bash`. Начнем со списка подкаталогов в `$HOME`, используя подоболочку, чтобы команда `cd` не могла изменить текущий каталог оболочки:

```
$ (cd && ls -d */)
Family/ Finances/ Linux/ Music/ Work/
```

Добавим `$HOME/` перед каждым каталогом с помощью `sed`:

```
$ (cd && ls -d */) | sed 's/^/$HOME\\/g'
$HOME/Family/
$HOME/Finances/
$HOME/Linux/
$HOME/Music/
$HOME/Work/
```

Синтаксис предыдущего сценария `sed` усложнен, поскольку строка замены `$HOME/` содержит слеш, а подстановки `sed` также используют косую черту в качестве разделителя. Вот почему один слеш экранирован: `$HOME\`. Вспомните из раздела «Подстановка и символ слеша» на с. 115, что `sed` принимает любой символ в качестве разделителя. Давайте использовать `@` вместо косой черты, тогда экранирование не потребует:

```
$ (cd && ls -d */) | sed 's^@$HOME/@g'
$HOME/Family/
$HOME/Finances/
$HOME/Linux/
$HOME/Music/
$HOME/Work/
```

Затем уберем последнюю косую черту еще одним выражением `sed`:

```
$ (cd && ls -d */) | sed -e 's^@$HOME/@' -e 's@/$@@'
$HOME/Family
$HOME/Finances
$HOME/Linux
$HOME/Music
$HOME/Work
```

Выведем результат в одной строке, используя `echo` и подстановку команд. Обратите внимание, что нам больше не нужны круглые скобки вокруг `cd` и `ls` для явного создания подболочки, потому что подстановка команд создает собственную подболочку:

```
$ echo $(cd && ls -d */ | sed -e 's^@$HOME/@' -e 's@/$@@')
$HOME/Family $HOME/Finances $HOME/Linux $HOME/Music $HOME/Work
```

Добавим в начало каталог `$HOME`, а в конец родительский каталог `..`:

```
echo '$HOME' \
$(cd && ls -d */ | sed -e 's^@$HOME/@' -e 's@/$@@') \
..
$HOME $HOME/Family $HOME/Finances $HOME/Linux $HOME/Music $HOME/Work ..
```

Заменяем пробелы на двоеточия, перенаправив весь вывод в команду `tr`:

```
$ echo '$HOME' \
      $(cd && ls -d */ | sed -e 's@^@$HOME/@' -e 's@/$@@') \
      .. \
      | tr ' ' ':'
$HOME:$HOME/Family:$HOME/Finances:$HOME/Linux:$HOME/Music:$HOME/Work:..
```

Наконец, добавим переменную окружения `CDPATH` — и определение переменной для вставки в файл конфигурации `bash` создано. Сохраним эту команду в сценарии, чтобы генерировать строку в любое время, например когда добавляем в `$HOME` новый подкаталог:

```
$ echo 'CDPATH=$HOME' \
$(cd && ls -d */ | sed -e 's@^@$HOME/@' -e 's@/$@@') \
.. \
| tr ' ' ':'
CDPATH=$HOME:$HOME/Family:$HOME/Finances:$HOME/Linux:$HOME/Music:$HOME/Work:..
```

## Создание тестовых файлов

Распространенной задачей при разработке программного обеспечения является тестирование — передача в программу самых разнообразных данных для проверки того, что программа ведет себя как задумано. Следующий однострочник генерирует тысячу файлов, содержащих случайный текст; эти файлы можно использовать при тестировании программного обеспечения. Тысяча — условное число, вы можете генерировать столько файлов, сколько захотите.

Однострочник будет случайным образом выбирать слова из большого текстового файла и создавать тысячу файлов меньшего размера со случайным содержимым. Идеальным исходным файлом является системный словарь `/usr/share/dict/words`, содержащий 102 305 слов, каждое из которых находится в отдельной строке.

```
$ wc -l /usr/share/dict/words
102305 /usr/share/dict/words
```

Чтобы создать этот дерзкий однострочник, нам нужно решить четыре головоломки:

1. Случайное перемешивание файла словаря.
2. Выбор случайного количества строк из файла словаря.
3. Создание выходного файла для хранения результатов.
4. Запуск полученной команды тысячу раз.

Чтобы перемешать слова в случайном порядке, используем команду `shuf`. При каждом выполнении команды `shuf /usr/share/dict/words` выводится более



ста тысяч строк, поэтому посмотрим первые несколько случайных строк с помощью команды `head`:

```
$ shuf /usr/share/dict/words | head -n3
evermore
shirttail
tertiary
$ shuf /usr/share/dict/words | head -n3
interactively
opt
perjurer
```

Первая головоломка решена. Теперь о том, как выбрать случайное количество строк из перемешанного словаря. `shuf` имеет параметр `-n` для печати заданного количества строк, но мы хотим, чтобы значение изменялось для каждого создаваемого выходного файла. К счастью, в `bash` есть переменная `RANDOM`, которая содержит случайное положительное целое число от 0 до 32 767. Его значение меняется каждый раз, когда мы обращаемся к переменной:

```
$ echo $RANDOM $RANDOM $RANDOM
7855 11134 262
```

Поэтому запустим `shuf` с параметром `-n $RANDOM`, чтобы вывести случайное количество случайных строк. Опять же, полный вывод может быть очень длинным, поэтому передадим результаты `wc -l`, чтобы убедиться, что количество строк меняется при каждом выполнении:

```
$ shuf -n $RANDOM /usr/share/dict/words | wc -l
9922
$ shuf -n $RANDOM /usr/share/dict/words | wc -l
32465
```

Мы решили вторую головоломку. Далее потребуется тысяча выходных файлов, точнее, тысяча файлов с разными именами. Чтобы сгенерировать имена файлов, запустим программу `pwgen`, которая генерирует случайные строки букв и цифр:

```
$ pwgen
eng9nooG ier6YeVu AhZ7naeG Ap3quail poo2Ooj9 OYiuri9m iQuash0E voo3Eph1
IeQu7mi6 eipaC2ti exah8iNg oeGhahm8 airooJ8N eiZ7neez Dah8Vooj dixiV1fu
Xiejoti6 ieshei2K iX4isohk Ohm5gaol Ri9ah4eX Aiv1ahg3 Shaew3ko zohB4geu
:
```

Добавим параметр `-N1`, чтобы сгенерировать только одну строку, и укажем длину строки (10) в качестве аргумента:

```
$ pwgen -N1 10
ieb2ESheiw
```

При необходимости сделаем строку более похожей на имя текстового файла, используя подстановку команд:

```
$ echo $(pwgen -N1 10).txt
ohTie8aifo.txt
```

Третья головоломка решена! Теперь у нас есть все необходимое для создания одного случайного текстового файла. Используем опцию `-o` команды `shuf`, чтобы сохранить результат в файле:

```
$ mkdir -p /tmp/randomfiles && cd /tmp/randomfiles
$ shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words
```

и проверим результаты:

```
$ ls                                Вывести новый файл
Ahxiedie2f.txt
$ wc -l Ahxiedie2f.txt              Сколько строк он содержит?
13544 Ahxiedie2f.txt
$ head -n3 Ahxiedie2f.txt           Посмотреть первые несколько строк
saviors
guerillas
forecaster
```

Выглядит неплохо! Последняя головоломка состоит в выполнении предыдущей команды `shuf` тысячу раз. Конечно, можно использовать цикл:

```
for i in {1..1000}; do
    shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words
done
```

но это не так весело, как создание дерзкого однострочника. Вместо этого давайте заранее сгенерируем команды в виде строк и передадим их в `bash`. В качестве теста напечатаем желаемую команду один раз с помощью `echo`. Добавим одинарные кавычки, чтобы гарантировать, что `$RANDOM` не вычисляется и `pwgen` не запускается:

```
$ echo 'shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words'
shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words
```

Эту команду можно передать в `bash` для выполнения:

```
$ echo 'shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words' | bash
$ ls
eiFohpies1.txt
```

Теперь повторим это тысячу раз, используя команду `yes`, переданную в `head`, а затем отправим результаты в `bash` — и мы решили четвертую головоломку:

```
$ yes 'shuf -n $RANDOM -o $(pwgen -N1 10).txt /usr/share/dict/words' \
| head -n 1000 \
| bash
$ ls
Aen1lee0ir.txt IeKaveixa6.txt ahDee9lah2.txt paeR1Poh3d.txt
Ahxiedie2f.txt Kas8ooJahK.txt aoc0Yooohoh.txt soh17Nohho.txt
CudieNgee4.txt Oe5ophae8e.txt haiV9mahNg.txt uchiek3Eew.txt
:
```

Если вы предпочитаете тысячу случайных файлов изображений вместо текстовых файлов, используйте ту же технику (`yes`, `head` и `bash`) и замените `shuf` командой, которая генерирует случайное изображение. Вот доработанный дерзкий однострочник из решения Марка Сетчелла (Mark Setchell) в Stack Overflow (<https://stackoverflow.com/questions/29011391/generate-random-bmp-in-cli>). Он запускает команду `convert` из графического пакета *ImageMagick* для создания случайных изображений размером 100×100 пикселей, состоящих из разноцветных квадратов:

```
$ yes 'convert -size 8x8 xc: +noise Random -scale 100x100 $(pwgen -N1 10).png' \
| head -n 1000 \
| bash
$ ls
Bahdo4Yaop.png Um8ju8gie5.png aing1QuaiX.png ohi4ziNuwo.png
Eem5leijae.png Va7ohchiep.png eiMoog1kou.png ohnohwu4Ei.png
Eozaing1ie.png Zaev4Quien.png hiecima2Ye.png quaepaiY9t.png
:
$ display Bahdo4Yaop.png          Просмотр первой картинки
```

## Создание пустых файлов

Иногда все, что вам нужно для тестирования, — это множество файлов с разными именами, даже если они пусты. Создать тысячу пустых файлов с именами от *file0001.txt* до *file1000.txt* очень просто:

```
$ mkdir /tmp/empties Create a directory for the files
$ cd /tmp/empties
$ touch file{01..1000}.txt Generate the files
```

Если вы предпочитаете более интересные имена файлов, выберите их случайным образом из системного словаря. Используйте `grep`, чтобы ограничить имена строчными буквами для простоты (избегая пробелов, апострофов и других символов, которые были бы специфичны для оболочки):

```
$ grep '^[a-z]*$' /usr/share/dict/words
a
aardvark
aardvarks
:
```

Перемешайте имена с помощью `shuf` и выведите первую тысячу с помощью `head`:

```
$ grep '^[a-z]*$' /usr/share/dict/words | shuf | head -n1000
triplicating
quadruplicates
podiatrists
:
```

Наконец, передайте результаты в `xargs`, чтобы создать файлы с помощью `touch`:

```
$ grep '^[a-z]*$' /usr/share/dict/words | shuf | head -n1000 | xargs touch
$ ls
abases                distinctly            magnolia              sadden
abets                 distrusts             maintaining           sales
aboard               divided              malformation         salmon
:
```

## Резюме

Надеюсь, что примеры из этой главы помогли вам развить навыки написания дерзких однострочников. Некоторые из них можно использовать как шаблоны, которые могут оказаться полезными в разных ситуациях.

Но не забывайте: дерзкие однострочники — не единственное решение. Это всего лишь один из подходов к эффективной работе в командной строке. Иногда вы можете добиться большего эффекта, написав сценарий. В других случаях вы найдете лучшие решения с помощью языка программирования, такого как Perl или Python. Тем не менее написание дерзких однострочников является важным навыком для быстрого и стильного выполнения критически важных задач.

## ГЛАВА 9

---

# Использование текстовых файлов

Обычный текст является наиболее распространенным форматом данных в системах Linux. Содержимое, передаваемое между командами в большинстве конвейеров, представляет собой текст. Файлы исходного кода программ, конфигурации системы в каталоге */etc*, а также файлы HTML и Markdown — все это текстовые данные. Письма электронной почты являются текстовыми, даже вложения сохраняются для транспортировки как текст. И файлы для повседневных забот, такие как списки покупок и личные заметки, можно хранить в виде текста.

Однако сегодняшний интернет представляет собой мешанину из потокового аудио и видео, сообщений в социальных сетях, документов в Google Docs и Office 365, PDF-файлов и других мультимедийных данных (не говоря уже о данных, обрабатываемых мобильными приложениями, которые скрыли понятие «файл» от целого поколения пользователей). В этих условиях простые текстовые файлы кажутся старомодными.

Тем не менее любой текстовый файл может стать богатым источником данных, которые вы можете извлечь с помощью команд Linux, особенно если текст структурирован. Каждая строка в файле */etc/passwd*, например, представляет пользователя Linux и имеет семь полей, включая имя пользователя, его числовой идентификатор, домашний каталог и т. д. Поля разделены двоеточиями, что упрощает анализ файла с помощью `cut -d:` или `awk -F:`. Вот, например, команда, которая печатает все имена пользователей (первое поле) в алфавитном порядке:

```
$ cut -d: -f1 /etc/passwd | sort
avahi
backup
daemon
:
```

А теперь перейдем к команде, которая отделяет пользователей-людей от системных учетных записей по их числовым идентификаторам и отправляет пользователям приветственное электронное письмо. Давайте построим этот дерзкий однострочник шаг за шагом. Во-первых, используем `awk` для вывода

имен пользователей (поле 1), когда числовой идентификатор пользователя (поле 3) равен 1000 или больше:

```
$ awk -F: '$3>=1000 {print $1}' /etc/passwd
jones
smith
```

Затем создадим приветствие, передав его в `xargs`:

```
$ awk -F: '$3>=1000 {print $1}' /etc/passwd \
| xargs -I@ echo "Hi there, @"
Hi there, jones!
Hi there, smith!
```

Затем сгенерируем команды (строки) для передачи каждого приветствия команде `mail`, которая отправляет электронное письмо данному пользователю с заданной строкой темы (`-s`):

```
$ awk -F: '$3>=1000 {print $1}' /etc/passwd \
| xargs -I@ echo 'echo "Hi there, @" | mail -s greetings @'
echo "Hi there, jones!" | mail -s greetings jones
echo "Hi there, smith!" | mail -s greetings smith
```

Наконец, передаем сгенерированные команды в `bash` для отправки электронных писем:

```
$ awk -F: '$3>=1000 {print $1}' /etc/passwd \
| xargs -I@ echo 'echo "Hi there, @" | mail -s greetings @' \
| bash
echo "Hi there, jones!" | mail -s greetings jones
echo "Hi there, smith!" | mail -s greetings smith
```

Большинство решений в этой книге начиналось с условия, что существует текстовый файл и его содержимое необходимо обработать с помощью команд. Пришло время изменить этот подход и научиться *формировать новые текстовые файлы*, которые хорошо сочетаются с командами Linux<sup>1</sup>. Это выигрышная стратегия для эффективной работы в Linux. Всего четыре шага позволяют добиться этого:

1. Проанализируйте конкретную задачу обработки данных, которую хотите решить.
2. Сохраните данные в текстовом файле в удобном формате.
3. Придумайте команды Linux, которые обрабатывают файл и решают задачу.
4. (Необязательно) Оформите эти команды как сценарии, псевдонимы или функции, чтобы упростить их выполнение.

<sup>1</sup> Этот подход аналогичен разработке схемы базы данных, которая хорошо работает с известными запросами.

В этой главе вы познакомитесь со множеством структурированных текстовых файлов и создадите команды для их обработки, чтобы решить несколько конкретных задач.

## Первый пример: поиск файлов

Предположим, что ваш домашний каталог содержит десятки тысяч файлов и подкаталогов, и время от времени вы не можете вспомнить, что где лежит. Команда `find` находит файл по имени, например *animals.txt*:

```
$ find $HOME -name animals.txt -print
/home/smith/Work/Writing/Books/Lists/animals.txt
```

Но команда `find` медленная, потому что она обыскивает весь ваш домашний каталог. Это шаг 1, намечающий практическую задачу — быстрый поиск файлов в вашем домашнем каталоге по имени.

Шаг 2 — сохранение данных в текстовом файле в удобном формате. Запустите `find`, чтобы создать список всех ваших файлов и каталогов, по одному пути к файлу в строке, и сохраните его в скрытом файле:

```
$ find $HOME -print > $HOME/.ALLFILES
$ head -n3 $HOME/.ALLFILES
/home/smith
/home/smith/Work
/home/smith/Work/resume.pdf
:
```

Теперь у вас есть данные — построчный индекс ваших файлов. На шаге 3 нужно создать команды для ускорения поиска файлов. Используем `grep`, так как быстрее просмотреть один большой файл, чем выполнять поиск в большом дереве каталогов:

```
$ grep animals.txt $HOME/.ALLFILES
/home/smith/Work/Writing/Books/Lists/animals.txt
```

Шаг 4 — упростить выполнение команды. Напишем однострочный сценарий с именем *ff* (*find files*) для поиска файлов, который запускает `grep` с любыми параметрами, заданными пользователем, и строкой поиска:

### Листинг 9.1. Сценарий `ff`

```
#!/bin/bash
# $@ означает "все аргументы, переданные скрипту"
grep "$@" $HOME/.ALLFILES
```

Сделаем сценарий исполняемым и поместим его в любой каталог в пути поиска, например в ваш личный подкаталог *bin*:

```
$ chmod +x ff
$ echo $PATH                               Проверьте ваш путь для поиска команд
/home/smith/bin :/usr/local/bin:/usr/bin:/bin
$ mv ff ~/bin
```

Используйте *ff*, чтобы быстро найти файлы, если не можете вспомнить, куда их положили.

```
$ ff animal
/home/smith/Work/Writing/Books/Lists/animals.txt
$ ff -i animal | less                      Нечувствительная к регистру команда grep
/home/smith/Work/Writing/Books/Lists/animals.txt
/home/smith/Vacations/Zoos/Animals/pandas.txt
/home/smith/Vacations/Zoos/Animals/tigers.txt
:
$ ff -i animal | wc -l                     Сколько совпадений?
16
```

Время от времени перезапускайте команду *find*, чтобы обновить индекс (еще лучше — создайте плановое задание с помощью *cron*, см. «Изучите команды *cron*, *crontab* и *at*» на с. 225). Вуаля! Из двух небольших команд мы создали быструю и гибкую утилиту для поиска файлов. Конечно, системы Linux предоставляют другие приложения для быстрой индексации и поиска файлов, такие как команда *locate* и утилиты поиска в GNOME, KDE Plasma и других окружениях рабочего стола, но это к делу не относится. Посмотрите, как легко было разработать такой инструмент самостоятельно. И ключом к успеху было создание текстового файла.

## Проверка срока действия домена

В следующем примере предположим, что вы владеете несколькими доменными именами в интернете и хотите отслеживать, когда истекает срок действия регистрации. Шаг 1 выполнен — практическая задача определена. Шаг 2 — создать файл со списком этих доменных имен, например *domains.txt*, по одному доменному имени в строке:

```
example.com
oreilly.com
efficientlinux.com
:
```



Шаг 3 — написать команды, которые дополняют этот текстовый файл датами истечения срока действия. Начнем с команды `whois`, которая запрашивает у регистратора информацию о домене:

```
$ whois example.com | less
Domain Name: EXAMPLE.COM
Registry Domain ID: 2336799_DOMAIN_COM-VRSN
Registrar WHOIS Server: whois.iana.org
Updated Date: 2021-08-14T07:01:44Z
Creation Date: 1995-08-14T04:00:00Z
Registry Expiry Date: 2022-08-13T04:00:00Z
:
```

Дате истечения срока действия предшествует строка *Registry Expiry Date*, которую можно изолировать с помощью `grep` и `awk`:

```
$ whois example.com | grep 'Registry Expiry Date:'
Registry Expiry Date: 2022-08-13T04:00:00Z
$ whois example.com | grep 'Registry Expiry Date:' | awk '{print $4}'
2022-08-13T04:00:00Z
```

Сделаем дату более читаемой с помощью команды `date --date`, которая преобразовывает строку даты из одного формата в другой:

```
$ date --date 2022-08-13T04:00:00Z
Sat Aug 13 00:00:00 EDT 2022
$ date --date 2022-08-13T04:00:00Z +%Y-%m-%d'          формат ГГГГ-ММ-ДД
2022-08-13
```

Используем подстановку команд, чтобы передать строку даты из `whois` в `date`:

```
$ echo $(whois example.com | grep 'Registry Expiry Date:' | awk '{print $4}')
2022-08-13T04:00:00Z
$ date \
  --date $(whois example.com \
            | grep 'Registry Expiry Date:' \
            | awk '{print $4}') \
  +%Y-%m-%d'
2022-08-13
```

Теперь у нас есть команда, которая запрашивает регистратора и выводит дату истечения срока действия. Создадим скрипт `check-expiry`, показанный в листинге 9.2, который запускает предыдущую команду и выводит на экран дату истечения срока действия и через символ табуляции — имя домена:

```
$ ./check-expiry example.com
2022-08-13          example.com
```

**Листинг 9.2.** check-expiry скрипт

```
#!/bin/bash
expdate=$(date \
    --date $(whois "$1" \
        | grep 'Registry Expiry Date:' \
        | awk '{print $4}') \
    +%Y-%m-%d')
echo "$expdate $1"          # Два значения, разделенные символом табуляции
```

Теперь проверим все домены в файле *domains.txt*, используя цикл. Создадим новый сценарий *check-expiry-all*, как показано в листинге 9.3.

**Листинг 9.3.** check-expiry-all сценарий

```
#!/bin/bash
cat domains.txt | while read domain; do
    ./check-expiry "$domain"
    sleep 5          # Будьте снисходительны к серверу регистратора
done
```

Запустим сценарий в фоновом режиме, так как выполнение может занять некоторое время, если у вас много доменов, и перенаправим весь вывод (*stdout* и *stderr*) в файл:

```
$ ./check-expiry-all &> expiry.txt &
```

Когда сценарий завершится, файл *expiry.txt* будет содержать нужную информацию:

```
$ cat expiry.txt
2022-08-13    example.com
2022-05-26    oreilly.com
2022-09-17    efficientlinux.com
:
```

Ура! Но не останавливайтесь на достигнутом. Файл *expiry.txt* хорошо структурирован для дальнейшей обработки, так как содержит два столбца, разделенные символом табуляции. Можно, например, отсортировать даты и найти следующий домен, требующий продления:

```
$ sort -n expiry.txt | head -n1
2022-05-26    oreilly.com
```

Или можно использовать *awk*, чтобы найти домены, срок действия которых (поле 1) истек или истекает сегодня, то есть меньше или равен сегодняшней дате (*date +%Y-%m-%d*):

```
$ awk "\$1<=\"$(date +%Y-%m-%d)\" \" expiry.txt
```

Несколько замечаний к предыдущей команде `awk`:

1. Экранирование знака доллара (перед `$1`) и двойные кавычки вокруг строки даты нужны, чтобы оболочка не вычислила их до того, как это сможет сделать `awk`.
2. Мы немного схитрили, используя строковый оператор `<=` для сравнения дат. Это не математическое сравнение, а просто сравнение строк, но оно работает, потому что формат даты `ГГГГ-ММ-ДД` сортируется одинаково как в алфавитном, так и в хронологическом порядках.

Приложив немного усилий, можно было бы вычислять даты в `awk`, чтобы предупредить об истечении срока действия, скажем, за две недели, а затем создать запланированное задание для запуска скрипта каждую ночь и отправки вам отчета по электронной почте. Не стесняйтесь экспериментировать. Суть здесь, однако, в том, что с помощью нескольких команд мы создали полезную утилиту, работающую с текстовым файлом.

## Создание базы данных телефонных кодов

В следующем примере используется файл с тремя полями, которые можно обрабатывать разными способами. Файл с именем *areacodes.txt* содержит телефонные коды городов США. Используйте один из дополнительных материалов к этой книге в каталоге *Chapter09/build\_area\_code\_database* (<https://resources.oreilly.com/examples/0636920601098>) или создайте свой собственный файл, например с помощью «Википедии»<sup>1</sup>:

```
201    NJ    Hackensack, Jersey City
202    DC    Washington
203    CT    New Haven, Stamford
:
989    MI    Saginaw
```



Первыми расположите поля предсказуемой длины, чтобы столбцы выглядели аккуратно выровненными. Посмотрите, каким сумбурным получится файл, если вы поместите названия городов в первую колонку:

```
Hackensack, Jersey City 201 NJ
Washington           202   DC
:
```

<sup>1</sup> В официальном списке кодов городов в формате CSV, составленном Администрацией Северо-американского плана нумерации, отсутствуют названия городов (<https://oreil.ly/SptWL>).

Когда файл будет готов, можно посмотреть коды городов по штатам с помощью `grep`, добавив параметр `-w` для соответствия только полным словам (на случай, если другой текст случайно содержит *NJ*):

```
$ grep -w NJ areacodes.txt
201    NJ      Hackensack, Jersey City
551    NJ      Hackensack, Jersey City
609    NJ      Atlantic City, Trenton, southeast and central west
:
```

или найти города по коду:

```
$ grep -w 202 areacodes.txt
202    DC      Washington
```

либо по любой строке в файле:

```
$ grep Washing areacodes.txt
202    DC      Washington
227    MD      Silver Spring, Washington suburbs, Frederick
240    MD      Silver Spring, Washington suburbs, Frederick
:
```

Подсчитаем количество кодов городов с помощью `wc`:

```
$ wc -l areacodes.txt
375 areacodes.txt
```

Найдем штат с наибольшим количеством кодов городов (победителем стала Калифорния с 38 кодами):

```
$ cut -f2 areacodes.txt | sort | uniq -c | sort -nr | head -n1
38 CA
```

Преобразуем файл в формат CSV для импорта в табличный процессор. Выведем третье поле, заключенное в двойные кавычки, чтобы его запятые не интерпретировались как символы-разделители CSV:

```
$ awk -F'\t' '{printf "%s,%s,\"%s\"\n", $1, $2, $3}' areacodes.txt \
> areacodes.csv
$ head -n3 areacodes.csv
201,NJ,"Hackensack, Jersey City"
202,DC,"Washington"
203,CT,"New Haven, Stamford"
```

Объединим все коды городов для данного штата в одну строку:

```
$ awk '$2~/^NJ$/{{ac=ac FS $1} END {print "NJ:" ac}}' areacodes.txt
NJ: 201 551 609 732 848 856 862 908 973
```

или сделаем все то же самое для каждого штата, используя массивы и циклы `for`, как в разделе «Улучшенный способ обнаружения дубликатов файлов» на с. 112:

```
$ awk '{arr[$2]=arr[$2] " " $1} \
      END {for (i in arr) print i ":" arr[i]}' areacodes.txt \
    | sort
AB: 403 780
AK: 907
AL: 205 251 256 334 659
:
WY: 307
```

Превратим любую из предыдущих команд в псевдонимы, функции или сценарии, как вам удобно. Простым примером является сценарий `areacode`, представленный в листинге 9.4.

#### Листинг 9.4. `areacode` сценарий

```
#!/bin/bash
if [ -n "$1" ]; then
    grep -iw "$1" areacodes.txt
fi
```

Сценарий `areacode` ищет любое слово целиком в файле `areacodes.txt`, например код города, аббревиатуру штата или название населенного пункта:

```
$ areacode 617
617      MA      Boston
```

## Создание менеджера паролей

В качестве последнего подробного примера давайте сохраним имена пользователей, пароли и примечания в зашифрованном текстовом файле в структурированном формате для удобного поиска в командной строке. Получившаяся команда представляет собой базовый менеджер паролей — приложение, которое облегчает запоминание большого количества сложных паролей.



Управление паролями — сложная тема в компьютерной безопасности. Этот пример создает чрезвычайно простой менеджер паролей в качестве учебного упражнения. Не используйте его для критически важных приложений.

Файл паролей с именем *vault* состоит из трех полей, разделенных одиночными символами табуляции:

- Имя пользователя.
- Пароль.
- Примечания (любой текст).

Создадим файл *vault* и добавим данные. Файл еще не зашифрован, поэтому пока вставляйте в него только фальшивые пароли:

<b>\$ touch vault</b>	<i>Создать пустой файл</i>
<b>\$ chmod 600 vault</b>	<i>Установить разрешения</i>
<b>\$ emacs vault</b>	<i>Отредактировать файл</i>
<b>\$ cat vault</b>	
sally fake1	google.com account
ssmith fake2	dropbox.com account for work
s999 fake3	Bank of America account, bankofamerica.com
smith2 fake4	My blog at wordpress.org
birdy fake5	dropbox.com account for home

Поместим файл *vault* в известное место:

```
$ mkdir ~/etc
$ mv vault ~/etc
```

Идея состоит в том, чтобы использовать программу сопоставления с шаблоном, такую как **grep** или **awk**, для вывода строк, соответствующих заданной строке. Этот простой, но мощный метод может сопоставить любую часть любой строки, а не только имена пользователей или веб-сайты. Например:

<b>\$ cd ~/etc</b>	
<b>\$ grep sally vault</b>	<i>Совпадение с именем пользователя</i>
sally fake1	google.com account
<b>\$ grep work vault</b>	<i>Совпадение с примечанием</i>
ssmith fake2	dropbox.com account for work
<b>\$ grep drop vault</b>	<i>Совпадение в нескольких строках</i>
ssmith fake2	dropbox.com account for work
birdy fake5	dropbox.com account for home

Зафиксируем эту простую функциональность в сценарии. Затем давайте улучшим его шаг за шагом, включая окончательное шифрование файла хранилища. Назовем сценарий **rtan** и создадим его простую версию, как в листинге 9.5.

**Листинг 9.5.** Сценарий **rtan**, версия 1: просто настолько, насколько это вообще возможно

```
#!/bin/bash
# Просто выводит совпадающие строки
grep "$1" $HOME/etc/vault
```

Сохраним скрипт в каталоге из вашего пути для поиска исполняемых файлов:

```
$ chmod 700 pman
$ mv pman ~/bin
```

Попробуем выполнить сценарий:

```
$ pman goog
sally fake1 google.com account
$ pman account
sally fake1 google.com account
ssmith fake2 dropbox.com account for work
s999 fake3 Bank of America account, bankofamerica.com
birdy fake5 dropbox.com account for home
$ pman facebook
```

(нет вывода)

Следующая версия этого сценария из листинга 9.6 добавляет проверки ошибок и сохраняемые имена переменных.

#### Листинг 9.6. Сценарий pman, версия 2: добавлена проверка ошибок

```
#!/bin/bash
# Захватить имя сценария
# $0 — это путь к сценарию, а basename печатает окончательное имя файла
PROGRAM=$(basename $0)
# Расположение файла - хранилища паролей
DATABASE=$HOME/etc/vault

# Убедитесь, что сценарию был предоставлен хотя бы один аргумент
# Выражение >&2 перенаправляет echo для печати в stderr вместо stdout
if [ $# -ne 1 ]; then
    >&2 echo "$PROGRAM: look up passwords by string"
    >&2 echo "Usage: $PROGRAM string"
    exit 1
fi

# Сохраните первый аргумент в понятной именованной переменной
searchstring="$1"
# Поиск в файле-хранилище и вывод сообщения об ошибке, если ничего не совпадает
grep "$searchstring" "$DATABASE"
if [ $? -ne 0 ]; then
    >&2 echo "$PROGRAM: no matches for '$searchstring'"
    exit 1
fi
```

Запустим сценарий:

```
$ pman
pman: look up passwords by string
Usage: pman string
$ pman smith
ssmith fake2 dropbox.com account for work
```

```
smith2 fake4 My blog at wordpress.org
$ rman xyzyy
rman: no matches for 'xyzyy'
```

Недостатком этого решения является неудобство поиска. Если бы хранилище содержало сотни строк, а команда **grep** сопоставила и напечатала 63 из них, нам пришлось бы искать нужный пароль на глаз. Улучшим сценарий, добавив уникальный ключ (строку) в третий столбец, и обновим **rman**, чтобы он сначала искал этот уникальный ключ. Файл хранилища с выделенным полужирным шрифтом третьим столбцом теперь выглядит так:

```
sally fake1 google google.com account
ssmith fake2 dropbox dropbox.com account for work
s999 fake3 bank Bank of America account, bankofamerica.com
smith2 fake4 blog My blog at wordpress.org
birdy fake5 dropbox2 dropbox.com account for home
```

В листинге 9.7 показан обновленный скрипт, использующий **awk** вместо **grep**. Он также использует подстановку команд для захвата вывода и проверки, не пуст ли он (проверка **-z** означает «строка нулевой длины»). Обратите внимание: если вы ищете ключ, которого нет в хранилище, **rman** возвращается к исходному поведению и печатает все строки, соответствующие строке поиска.

#### Листинг 9.7. Сценарий **rman**, версия 3: приоритет поиска ключа в третьем столбце

```
#!/bin/bash
PROGRAM=$(basename $0)
DATABASE=$HOME/etc/vault
if [ $# -ne 1 ]; then
    >&2 echo "$PROGRAM: look up passwords"
    >&2 echo "Usage: $PROGRAM string"
    exit 1
fi
searchstring="$1"

# Искать точные совпадения в третьем столбце
match=$(awk '$3~/^'$searchstring'$/' "$DATABASE")

# Если строка поиска не соответствует ключу, найти все совпадения
if [ -z "$match" ]; then
    match=$(awk "/$searchstring/" "$DATABASE")
fi

# Если совпадений по-прежнему нет, вывести сообщение об ошибке и выйти.
if [ -z "$match" ]; then
    >&2 echo "$PROGRAM: no matches for '$searchstring'"
    exit 1
fi

# Вывод совпадений
echo "$match"
```



Запустим сценарий:

```
$ pman dropbox
ssmith fake2 dropbox dropbox.com account for work
$ pman drop
ssmith fake2 dropbox dropbox.com account for work
birdy fake5 dropbox2 dropbox.com account for home
```

Хранилище паролей в виде файла с простым текстом представляет собой угрозу безопасности, поэтому зашифруем его с помощью стандартной программы шифрования Linux GnuPG, которая вызывается как `gpg`. Если у вас уже настроен GnuPG, это прекрасно. В противном случае настройте его с помощью следующей команды, указав свой адрес электронной почты<sup>1</sup>:

```
$ gpg --quick-generate-key your_email_address default default never
```

Вам будет дважды предложено ввести пароль для ключа. Укажите надежный пароль. Когда `gpg` завершится, вы будете готовы зашифровать файл паролей, создав файл *vault.gpg*:

```
$ cd ~/etc
$ gpg -e -r your_email_address vault
$ ls vault*
vault vault.gpg
```

В качестве проверки расшифруем файл *vault.gpg* в стандартный вывод<sup>2</sup>:

```
$ gpg -d -q vault.gpg
Passphrase: xxxxxxxx
sally fake1 google google.com account
ssmith fake2 dropbox dropbox.com account for work
:
```

Затем обновим свой сценарий, чтобы использовать зашифрованный файл *vault.gpg* вместо обычного текстового файла хранилища. Это означает расшифровку *vault.gpg* в стандартный вывод и передачу его содержимого в `awk` для сопоставления, как указано в листинге 9.8.

**Листинг 9.8.** Сценарий `pman`, версия 4: использование зашифрованного хранилища

```
#!/bin/bash
PROGRAM=$(basename $0)
# Использовать зашифрованный файл
DATABASE=$HOME/etc/vault.gpg
if [ $# -ne 1 ]; then
```

<sup>1</sup> Эта команда создает пару «публичный/приватный ключ» со всеми параметрами по умолчанию и бесконечным сроком действия. Чтобы узнать больше, см. `man gpg` или найдите учебник по GnuPG в интернете.

<sup>2</sup> Если `gpg` продолжает работу без запроса пароля, значит, он временно кэширован (сохранен).

```

>&2 echo "$PROGRAM: look up passwords"
>&2 echo "Usage: $PROGRAM string"
exit 1
fi
searchstring="$1"
# Сохранить расшифрованный текст в переменной
decrypted=$(gpg -d -q "$DATABASE")
# Искать точные совпадения в третьем столбце
match=$(echo "$decrypted" | awk '$3~/^'$searchstring'$/' )
# Если строка поиска не соответствует ключу, найти все совпадения
if [ -z "$match" ]; then
    match=$(echo "$decrypted" | awk "/$searchstring/")
fi
# Если совпадений по-прежнему нет, вывести сообщение об ошибке и выйти
if [ -z "$match" ]; then
    >&2 echo "$PROGRAM: no matches for '$searchstring'"
    exit 1
fi
# Вывод совпадений
echo "$match"

```

Теперь сценарий отображает пароли из зашифрованного файла:

```

$ pman dropbox
Passphrase: xxxxxxxx
ssmith fake2 dropbox dropbox.com account for work
$ pman drop
Passphrase: xxxxxxxx
ssmith fake2 dropbox dropbox.com account for work
birdy fake5 dropbox2 dropbox.com account for home

```

Теперь у нас есть полноценный менеджер паролей. Некоторые заключительные шаги:

- Когда вы убедитесь, что можете надежно расшифровывать файл *vault.gpg*, удалите исходный файл *vault*.
- При желании замените поддельные пароли реальными. См. раздел «Прямое редактирование зашифрованных файлов» на с. 196 с рекомендациями по редактированию зашифрованного текстового файла.
- Не забывайте про комментарии в хранилище паролей — строки, начинающиеся со знака решетки (#), — чтобы вы могли делать примечания к записям. Обновите сценарий, чтобы передать расшифрованное содержимое команде `grep -v`. Это позволит отфильтровать строки, начинающиеся со знака решетки:

```
decrypted=$(gpg -d -q "$DATABASE" | grep -v '^#')
```

Печать паролей в стандартный вывод — не очень хорошая практика с точки зрения безопасности. В разделе «Улучшение работы диспетчера паролей» на с. 216 этот сценарий будет доработан для копирования и вставки паролей вместо их вывода на экран.

### ПРЯМОЕ РЕДАКТИРОВАНИЕ ЗАШИФРОВАННЫХ ФАЙЛОВ

Чтобы изменить зашифрованный файл, наиболее прямым, трудоемким и небезопасным методом является расшифровка файла, его редактирование и повторное шифрование.

```
$ cd ~/etc
```

```
$ gpg vault.gpg
```

*Расшифровка*

```
Passphrase: xxxxxxxx
```

```
$ emacs vault
```

*Использование текстового редактора*

```
$ gpg -e -r your_email_address vault
```

*Шифрование*

```
$ rm vault
```

Для упрощения редактирования файла *vault.gpg* и в Emacs, и в Vim есть режимы редактирования файлов, зашифрованных с помощью GnuPG. Начните с добавления следующей строки в файл конфигурации *bash* и применения его ко всем связанным оболочкам:

```
export GPG_TTY=$(tty)
```

Для Emacs настройте встроенный пакет EasyPG. Добавьте следующие строки в файл конфигурации *\$HOME/.emacs* и перезапустите Emacs (замените строку *GnuPG ID here* адресом электронной почты, связанным с вашим ключом, например *smith@example.com*):

```
(load-library "pinentry")
(setq epa-pinentry-mode 'loopback)
(setq epa-file-encrypt-to "GnuPG ID here")
(pinentry-start)
```

После этого при редактировании зашифрованного файла Emacs запросит ваш пароль и расшифрует файл в буфер для редактирования. При сохранении Emacs шифрует содержимое буфера.

Для Vim попробуйте плагин *vim-gnupg* (<https://oreil.ly/mnwYc>) и добавьте следующие строки в файл конфигурации *\$HOME/.vimrc*:

```
let g:GPGPreferArmor=1
let g:GPGDefaultRecipients=["GnuPG ID here"]
```

Подумайте о создании псевдонима для удобного редактирования хранилища паролей, используя прием из раздела «Редактируйте часто используемые файлы с помощью псевдонима» на с. 74:

```
alias pwedit="$EDITOR $HOME/etc/vault.gpg"
```

## Резюме

Пути к файлам, имена доменов, коды городов и учетные данные для входа — это лишь несколько примеров данных, которые удобны для обработки в структурированном текстовом файле. Есть много других возможностей:

- Ваши музыкальные файлы — используйте команду Linux, например `id3tool`, для извлечения информации ID3 из файлов MP3 и помещения ее в файл.
- Контакты на вашем мобильном устройстве — используйте приложение для экспорта контактов в формат CSV, загрузите их в облачное хранилище, а затем загрузите на свой Linux-компьютер для обработки.
- Ваши оценки в школе — используйте `awk` для отслеживания среднего балла.
- Список просмотренных фильмов или прочитанных книг с дополнительными данными — рейтингами, авторами, актерами и т. д.

Таким образом, вы можете создать целую систему команд, связанных с работой и отдыхом и экономящих ваше время. Перспективы ограничены только вашим воображением.



## ЧАСТЬ 3

---

# Дополнительные плюсы

Заключительные главы посвящены специальным темам: некоторые из них описаны подробно, другие лишь вкратце, чтобы пробудить в вас желание узнать больше.

## ГЛАВА 10

---

# Эффективное использование клавиатуры

В самый обычный день на обычном компьютере с Linux у вас может быть открыто множество окон: веб-браузеры, текстовые редакторы, среды разработки программного обеспечения, музыкальные проигрыватели, видеоредакторы, виртуальные машины и т. д. Некоторые приложения ориентированы на графический интерфейс, например программы для рисования, и приспособлены для работы с мышью или трекболом. Другие более ориентированы на клавиатуру, например командная оболочка внутри терминальной программы. Типичный пользователь Linux может переходить с клавиатуры на мышь и обратно десятки (или даже сотни) раз в час. Каждое такое переключение требует времени и замедляет работу. Если вы сможете уменьшить количество переключений, то станете работать более эффективно.

В этой главе рассказывается о том, как проводить больше времени за клавиатурой и меньше пользоваться мышью. Десять пальцев, нажимающих на сотню клавиш, обычно более быстры, чем пара пальцев на мыши. Речь не только об использовании горячих клавиш — я уверен, что вы найдете информацию о них и без этой книги (хотя некоторые сочетания все-таки будут приведены). Я говорю о подходе к ускорению некоторых повседневных задач, которые по своей сути требуют использования мыши: работа с окнами, получение информации из интернета, копирование и вставка с помощью буфера обмена.

## Работа с окнами

В этом разделе я поделюсь советами по эффективному запуску окон, особенно окон командной оболочки (терминалов) и браузера.

## Мгновенный запуск оболочек и браузера

Большинство сред рабочего стола Linux, такие как GNOME, KDE Plasma, Unity и Cinnamon, позволяют назначить горячие клавиши — специальные сочетания клавиш, которые запускают команды или выполняют другие операции. Настоятельно рекомендуется определить сочетания клавиш для следующих распространенных операций:

- Запуск терминальной программы (окна командной оболочки).
- Запуск окна веб-браузера.

После этого вы сможете мгновенно открывать терминал или браузер в любой момент, независимо от того, какое приложение вы используете<sup>1</sup>. Для настройки необходимо знать следующее:

*Команда, запускающая предпочитаемую вами терминальную программу*

Самые популярные — `gnome-terminal`, `konsole` и `xterm`.

*Команда, запускающая предпочитаемый вами браузер*

Самые популярные — `firefox`, `google-chrome` и `opera`.

*Способ определения собственного сочетания клавиш*

Инструкции различаются для каждого типа окружения рабочего стола и могут меняться от версии к версии, поэтому лучше уточнить их в интернете. Выполните поиск по имени вашего окружения рабочего стола с ключевыми словами «определение сочетания клавиш» («define keyboard shortcut»).

На своем компьютере я назначил сочетание клавиш **Ctrl-Windows-T** для запуска терминала `konsole` и **Ctrl-Windows-C** для запуска `google-chrome`.



### Рабочие каталоги

Когда вы запускаете новый экземпляр оболочки с помощью сочетания клавиш в среде рабочего стола, он является дочерним по отношению к вашей оболочке входа в систему. Его текущий каталог — ваш домашний каталог (если только вы не настроили его иначе).

Сравните это с открытием новой оболочки из вашей терминальной программы, явно запустив, например, `gnome-terminal` или `xterm` в командной

<sup>1</sup> Если только вы не работаете с приложением, которое перехватывает все нажатия клавиш, например с виртуальной машиной в отдельном окне.



строке либо используя меню вашей терминальной программы. В этом случае новая оболочка является потомком оболочки этого терминала. Его текущий каталог — такой же, как у его родителя, который может не быть вашим домашним каталогом.

## Одноразовые окна

Предположим, вы используете несколько приложений, и вдруг вам нужна оболочка для выполнения одной команды. Многие пользователи хватают мышь и ищут в открытых окнах работающий терминал. Не делайте этого — вы просто теряете время. Откройте новый терминал с помощью горячих клавиш, запустите команду и сразу же выйдите из терминала.

Если у вас назначены горячие клавиши для запуска терминальных программ и окон браузера, смело открывайте и не забывайте закрывать эти окна. Не оставляйте терминалы и окна браузера открытыми на долгое время! Я называю эти недолговечные окна *одноразовыми* — они быстро открываются, используются несколько секунд и затем закрываются.

Есть смысл оставить несколько оболочек открытыми в течение длительного времени, если вы разрабатываете программное обеспечение или выполняете другую длительную работу, но одноразовые окна терминала идеально подходят для команд, которые могут понадобиться один раз в течение дня. *Часто быстрее открыть новый терминал, чем искать на экране существующий.* Не спрашивайте себя: «Где нужное окно терминала?» — и не теряйте время на поиск. Лучше создайте новое окно и закройте его? после того как оно выполнило свою задачу.

Тот же принцип действует для окон веб-браузера. Вы когда-нибудь поднимали голову после долгого дня работы в Linux и обнаруживали, что в вашем браузере всего одно окно и 83 открытые вкладки? Это симптом неиспользования одноразовых окон. Откройте одно, просмотрите веб-страницу, а потом закройте его. Нужно вернуться на страницу позже? Обратитесь к истории браузера.

## Горячие клавиши в браузере

Поскольку мы заговорили об окнах браузера, убедитесь, что знаете наиболее важные сочетания клавиш из табл. 10.1. Если ваши руки уже находятся на клавиатуре и вы хотите перейти на новый веб-сайт, зачастую быстрее нажать **Ctrl-L**, чтобы перейти к адресной строке, или **Ctrl-T**, чтобы открыть вкладку, чем двигать указатель мыши.

**Таблица 10.1.** Наиболее важные горячие клавиши для Firefox, Google Chrome и Opera

Действие	Горячие клавиши
Открыть новое окно	Ctrl-N
Открыть новое окно в режиме инкогнито	Ctrl-Shift-P (Firefox), Ctrl-Shift-N (Chrome и Opera)
Открыть новую вкладку и перейти на нее	Ctrl-T
Заккрыть активную вкладку	Ctrl-W
Перейти на следующую открытую вкладку	Ctrl-Tab (переход вперед) и Ctrl-Shift-Tab (переход назад)
Перейти на адресную строку	Ctrl-L (или Alt-D, или F6)
Искать текст на активной вкладке	Ctrl-F
Открыть историю	Ctrl-H

## Переключение окон и рабочих столов

Когда ваш рабочий стол заполнен окнами, как быстро найти нужное? Вы можете водить указателем мыши и пробираться сквозь тернии к нужному окну, но быстрее использовать сочетание клавиш **Alt-Tab**. Продолжайте нажимать **Alt-Tab**, и вы последовательно просмотрите все окна на рабочем столе. Когда дойдете до нужного окна, отпустите клавиши — и это окно окажется в фокусе и будет готово к использованию. Чтобы прокрутить в обратном направлении, нажмите **Alt-Shift-Tab**.

Чтобы просмотреть все окна на рабочем столе, принадлежащие одному и тому же приложению, например все окна Firefox, нажмите **Alt-`** (обратная кавычка — клавиша над **Tab**). Чтобы вернуться, добавьте нажатие клавиши **Shift** (**Alt-Shift-`**).

Когда вы научились переключать окна, пришло время поговорить о переключении рабочих столов. Если вы используете только один рабочий стол (также называемый рабочим пространством или виртуальным рабочим столом), значит упускаете отличный способ улучшить организацию своей работы в Linux. Вместо одного у вас может быть четыре, шесть или более рабочих столов, каждый со своими окнами, и вы можете переключаться между ними.

На рабочем компьютере под управлением Ubuntu Linux с KDE Plasma я запускаю шесть виртуальных рабочих столов и назначаю им разные цели. Рабочий стол № 1 — это мое основное рабочее пространство с электронной почтой и браузером, № 2 — для семейных задач, № 3 — это место, где я запускаю виртуальные машины VMware, № 4 — для написания книг, подобных этой, а № 5–6 — для разнообразных разовых или срочных задач. Такая группировка рабочих столов позволяет мне быстро и легко находить открытые окна из разных приложений.

Каждое окружение рабочего стола Linux, такое как GNOME, KDE Plasma, Cinnamon и Unity, имеет свой собственный способ реализации виртуальных рабочих столов, но все они предоставляют механизм для переключения между ними. Я рекомендую определить сочетания клавиш в окружении вашего рабочего стола, чтобы быстро переходить на нужный рабочий стол. На своем компьютере я использую сочетания клавиш от Windows + F1 до Windows + F6 для перехода к рабочим столам с № 1 по № 6 соответственно.

Существует множество других стилей работы с рабочими столами и окнами. Некоторые люди используют один рабочий стол для каждого приложения: терминала, просмотра веб-страниц, обработки текстов и т. д. Пользователи компактных ноутбуков часто открывают только одно окно в полноэкранном режиме на каждом рабочем столе. Найдите стиль, который вам подходит, главное, чтобы он способствовал быстрой и эффективной работе.

## Доступ в интернет из командной строки

Браузеры с управлением типа «укажи и щелкни» почти синонимичны интернету, но вы также можете получать доступ к веб-сайтам из командной строки Linux, что иногда бывает полезно.

## Запуск окон браузера из командной строки

Возможно, вы привыкли запускать веб-браузер, щелкая мышкой или касаясь значка, но вы также можете делать это из командной строки Linux. Если браузер еще не запущен, добавьте символ амперсанда, чтобы запустить его в фоновом режиме и вернуть приглашение командной строки:

```
$ firefox &  
$ google-chrome &  
$ opera &
```

Если браузер уже запущен, уберите амперсанд. Такая команда указывает существующему экземпляру браузера открыть новое окно или вкладку, после чего завершает работу и возвращает приглашение командной строки.



Команда запуска браузера в фоновом режиме может печатать диагностические сообщения и загромождать окно оболочки. Чтобы предотвратить это, перенаправьте весь вывод при первом запуске браузера в `/dev/null`. Например:

```
$ firefox &> /dev/null &
```

Чтобы открыть браузер и перейти по URL-адресу из командной строки, укажите URL-адрес в качестве аргумента:

```
$ firefox https://oreilly.com
$ google-chrome https://oreilly.com
$ opera https://oreilly.com
```

По умолчанию предыдущие команды открывают новую вкладку и переходят на нее. Если вместо этого надо заставить их открыть новое окно, добавьте параметр:

```
$ firefox --new-window https://oreilly.com
$ google-chrome --new-window https://oreilly.com
$ opera --new-window https://oreilly.com
```

Чтобы открыть приватное окно или окно браузера в режиме инкогнито, добавьте соответствующий параметр командной строки:

```
$ firefox --private-window https://oreilly.com
$ google-chrome --incognito https://oreilly.com
$ opera --private https://oreilly.com
```

Приведенные выше команды могут показаться трудоемкими в наборе, но вы можете повысить свою эффективность, определив псевдонимы для сайтов, которые часто посещаете:

```
# Поместите в файл конфигурации оболочки и используйте:
alias oreilly="firefox --new-window https://oreilly.com"
```

Аналогичным образом, если у вас есть файл, содержащий интересующий URL-адрес, извлеките этот адрес с помощью `grep`, `cut` или других команд Linux и передайте его в браузер, используя подстановку команд. Вот пример работы с файлом, содержащим разделенные табуляциями два столбца:

```
$ cat urls.txt
duckduckgo.com My search engine
nytimes.com My newspaper
spotify.com My music
$ grep music urls.txt | cut -f1
spotify.com
$ google-chrome https://$( grep music urls.txt | cut -f1 )
```

*Переход на сайт*

Или предположим, что вы отслеживаете ожидаемые посылки с помощью файла с номерами треков:

```
$ cat packages.txt
1Z0EW7360669374701 UPS Shoes
568733462924 FedEx Kitchen blender
9305510823011761842873 USPS Care package from Mom
```

Сценарий в листинге 10.1 открывает страницы отслеживания для отправителей (UPS, FedEx или почтовая служба США), добавляя номера отслеживания к соответствующим URL-адресам.

### **Листинг 10.1.** Сценарий track-it, открывающий страницу отслеживания грузоотправителей

```
#!/bin/bash
PROGRAM=$(basename $0)
DATAFILE=packages.txt
# Выберите браузер: firefox, opera или google-chrome
BROWSER="opera"
errors=0

cat "$DATAFILE" | while read line; do
    track=$(echo "$line" | awk '{print $1}')
    service=$(echo "$line" | awk '{print $2}')
    case "$service" in
        UPS)
            $BROWSER "https://www.ups.com/track?tracknum=$track" &
            ;;
        FedEx)
            $BROWSER "https://www.fedex.com/fedextrack/?trknbr=$track" &
            ;;
        USPS)
            $BROWSER "https://tools.usps.com/go/TrackConfirmAction?tLabels=$track" &
            ;;
        *)
            >&2 echo "$PROGRAM: Unknown service '$service'"
            errors=1
            ;;
    esac
done
exit $errors
```

## Получение HTML-страниц с помощью curl и wget

Не только веб-браузеры посещают веб-сайты. Программы `curl` и `wget` могут загружать веб-страницы и другой веб-контент без использования браузера. По умолчанию `curl` выводит в *stdout*, а `wget` сохраняет вывод в файл:

```
$ curl https://efficientlinux.com/welcome.html
Welcome to Efficient Linux.com!
$ wget https://efficientlinux.com/welcome.html
--2021-10-27 20:05:47-- https://efficientlinux.com/
Resolving efficientlinux.com (efficientlinux.com)...
Connecting to efficientlinux.com (efficientlinux.com)...
:
2021-10-27 20:05:47 (12.8 MB/s) - 'welcome.html' saved [32/32]
$ cat welcome.html
Welcome to Efficient Linux.com!
```



Некоторые сайты не поддерживают выгрузку данных с помощью `wget` и `curl`. В таких случаях обе команды могут маскироваться под браузер. Просто укажите каждой программе изменить свой пользовательский агент — строку, которая идентифицирует веб-клиент для веб-сервера. Удобный пользовательский агент — «Mozilla»:

```
$ wget -U Mozilla url
$ curl -A Mozilla url
```

И у `wget`, и у `curl` есть множество опций и функций, которые вы можете найти на их справочных страницах. А пока давайте посмотрим, как включить эти команды в дерзкие однострочники. Предположим, что на веб-сайте *Effectivelinux.com* есть каталог *images*, содержащий файлы с *1.jpg* по *20.jpg*, и вы хотите их загрузить. Их URL-адреса:

```
https://efficientlinux.com/images/1.jpg
https://efficientlinux.com/images/2.jpg
https://efficientlinux.com/images/3.jpg
:
```

Неэффективно посещение каждого URL-адреса по одному и последовательная загрузка изображений (поднимите руку, если вы когда-либо делали подобное!). Лучший способ — использовать `wget`. Сгенерируйте URL-адреса с помощью `seq` и `awk`:

```
$ seq 1 20 | awk '{print "https://efficientlinux.com/images/" $1 ".jpg"}'
https://efficientlinux.com/images/1.jpg
https://efficientlinux.com/images/2.jpg
https://efficientlinux.com/images/3.jpg
:
```

Затем добавьте строку `wget` в программу `awk` и передайте полученные команды в `bash` для выполнения:

```
$ seq 1 20 \
  | awk '{print "wget https://efficientlinux.com/images/" $1 ".jpg"}' \
  | bash
```

В качестве альтернативы используйте `xargs` для создания и выполнения команд `wget`:

```
$ seq 1 20 | xargs -I@ wget https://efficientlinux.com/images/@.jpg
```

Вариант с командой `xargs` предпочтителен, если ваши команды `wget` содержат специальные символы. Решение, использующее *канал в bash (pipe to bash)*, заставит оболочку вычислять эти символы (чего вы не хотите), а решение с `xargs` — нет.

Пример выше был надуманным, потому что имена файлов изображений, как правило, схожи. В более реалистичном примере вы можете загрузить все изображения с веб-страницы с помощью `curl`, пропустив ее через хитроумную последовательность команд, чтобы изолировать URL-адреса изображений, по одному на строку, а затем применяя один из методов, которые показаны выше:

```
curl URL | ...какой-то умный конвейер... | xargs -n1 wget
```

## Обработка кода HTML с помощью HTML-XML-utils

Если вы знакомы с HTML и CSS, то сможете анализировать исходный HTML-код веб-страниц из командной строки. Иногда это более эффективно, чем ручное копирование и вставка фрагментов веб-страницы из окна браузера. Удобным набором инструментов является HTML-XML-utils, который доступен во многих дистрибутивах Linux от World Wide Web Consortium (<https://www.w3.org/Tools/HTML-XML-utils/>). Общий порядок действий:

1. Захватите исходный код HTML с помощью `curl` (или `wget`).
2. Оптимизируйте формат HTML с помощью `hxnormalize`.
3. Определите CSS-селекторы для HTML-элементов.
4. Используйте `hxselect`, чтобы изолировать значения, и передайте вывод для последующей обработки.

Давайте дополним пример из раздела «Создание базы данных телефонных кодов» на с. 188, чтобы создать файл *areacodes.txt* с кодами городов. Для вашего удобства создана HTML-таблица кодов городов (рис. 10.1), которую вы можете загрузить и обработать.

Area code	State	Location
201	NJ	Hackensack, Jersey City
202	DC	Washington
203	CT	New Haven, Stamford
204	MB	entire province
205	AL	Birmingham, Tuscaloosa
206	WA	Seattle
207	ME	entire state
208	ID	entire state
209	CA	Modesto, Stockton
210	TX	San Antonio
212	NY	New York City, Manhattan

**Рис. 10.1.** Таблица кодов городов на <https://efficientlinux.com/areacodes.html>

Вначале обработаем исходный HTML-код с помощью `curl`, используя параметр `-s` для подавления экранных сообщений. Направим вывод команде `hxnormalize -x` для очистки. Затем передадим его в `less`, чтобы просмотреть вывод поэкранно:

```
$ curl -s https://efficientlinux.com/areacodes.html \
| hxnormalize -x \
| less
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
:
  <body>
    <h1>Area code test</h1>
    :
```

Таблица HTML, показанная в листинге 10.2, имеет CSS ID `#ac`, а ее три столбца (код города — Area code, штат — State и местоположение — Location) используют классы CSS `ac`, `state` и `cities` соответственно.

### Листинг 10.2. Часть HTML-кода таблицы на рис. 10.1

```
<table id="ac">
  <thead>
    <tr>
      <th>Area code</th>
      <th>State</th>
      <th>Location</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td class="ac">201</td>
```



```

        <td class="state">NJ</td>
        <td class="cities">Hackensack, Jersey City</td>
    </tr>
    :
</tbody>
</table>

```

Запустим `hxselect`, чтобы извлечь данные кода города из каждой ячейки таблицы, и укажем параметр `-c`, чтобы исключить теги `td` из вывода. Выведем на экран результаты в виде одной строки с полями, разделенными символом `@`, используя опцию `-s`<sup>1</sup> (символ `@` выбран для улучшения читаемости):

```

$ curl -s https://efficientlinux.com/areacodes.html \
  | hxnormalize -x \
  | hxselect -c -s@ '#ac .ac, #ac .state, #ac .cities'
201@NJ@Hackensack, Jersey City@202@DC@Washington@203@CT@New Haven, Stamford@...

```

Направим вывод в `sed`, чтобы превратить эту длинную строку в три колонки, разделенных табуляцией.

Теперь нам требуется регулярное выражение со следующей структурой:

1. Код города, состоящий из цифр, `[0-9]*`.
2. Символ `@`.
3. Аббревиатура штата, состоящая из двух заглавных букв из диапазона `[A-Z]`.
4. Символ `@`.
5. Название города — любой текст, не содержащий символа `@`, `( [^@]*)`.
6. Символ `@`.

Объединим все части, чтобы получить следующее регулярное выражение:

```
[0-9]* @ [A-Z][A-Z] @ [^@]* @
```

Запишем код города, штата и название города в виде трех подвыражений, обозначив их символами `\`. Теперь у нас есть полное регулярное выражение для `sed`:

```
\( [0-9]* \)\( [A-Z][A-Z] \)\( [^@]* \)@
```

Для строки замены `sed` укажем три подвыражения, разделенные символами табуляции и оканчивающиеся символами новой строки, что соответствует формату файла *areacodes.txt*:

```
\1\t\2\t\3\n
```

<sup>1</sup> В этом примере используются три селектора CSS, но некоторые старые версии `hxselect` могут обрабатывать только два. Если ваша `hxselect` имеет этот недостаток, загрузите последнюю версию на странице World Wide Web Consortium (<https://www.w3.org/Tools/HTML-XML-utils/>) и выполните сборку с помощью команды `configure && make`.

Объединим предыдущее регулярное выражение и строку замены, чтобы создать sed-сценарий:

```
s/ \([0-9]*\)@\([A-Z][A-Z]\)\@(\^[@]*\)@ / \1\t2\t3\n /g
```

Готовая команда использует данные файла *areacodes.txt*:

```
$ curl -s https://efficientlinux.com/areacodes.html \
| hxnormalize -x \
| hxselect -c -s'@' '#ac .ac, #ac .state, #ac .cities' \
| sed 's/ \([0-9]*\)@\([A-Z][A-Z]\)\@(\^[@]*\)@ / \1\t2\t3\n/g'
201 NJ Hackensack, Jersey City
202 DC Washington
203 CT New Haven, Stamford
:
```

### ОБРАБОТКА ДЛИННЫХ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Если ваши sed-сценарии становятся слишком длинными, то они выглядят как случайный набор символов:

```
s/ \([0-9]*\)@\([A-Z][A-Z]\)\@(\^[@]*\)@ / \1\t2\t3\n /g
```

Постарайтесь разделить их. Сохраните части регулярного выражения в нескольких переменных оболочки и объедините переменные позже, как в следующем сценарии:

```
# Три части регулярного выражения.
# Используем одинарные кавычки, чтобы предотвратить вычисление
# выражения оболочкой
areacode=' \([0-9]*\)'
state=' \([A-Z][A-Z]\)'
cities=' \([^\@]*\)'

# Объединяем три части, разделенные символами @.
# Используем двойные кавычки, чтобы позволить оболочке вычислить
# переменную
regex="$areacode@$state@$cities@"

# Строка для замены
# Используем одинарные кавычки, чтобы предотвратить вычисление
# выражения оболочкой
replacement=' \1\t2\t3\n'

# sed-сценарий стал проще для восприятия:
# s/$regex/$replacement/g
# Запускаем полную команду:
curl -s https://efficientlinux.com/areacodes.html \
| hxnormalize -x \
| hxselect -c -s'@' '#ac .ac, #ac .state, #ac .cities' \
| sed "s/$regex/$replacement/g"
```

## Получение и отображение содержимого веб-сайтов с помощью текстового браузера

При получении данных из интернета в командной строке вам может понадобиться не HTML-код веб-страницы, а визуализированная версия. Визуализированный текст легче анализируется. Для этого используйте текстовый браузер, такой как `lynx` или `links`. Текстовые браузеры отображают веб-страницы в урезанном формате без изображений и других модных функций. На рис. 10.2 показана страница кодов городов из предыдущего раздела, отображаемая с помощью `lynx`.

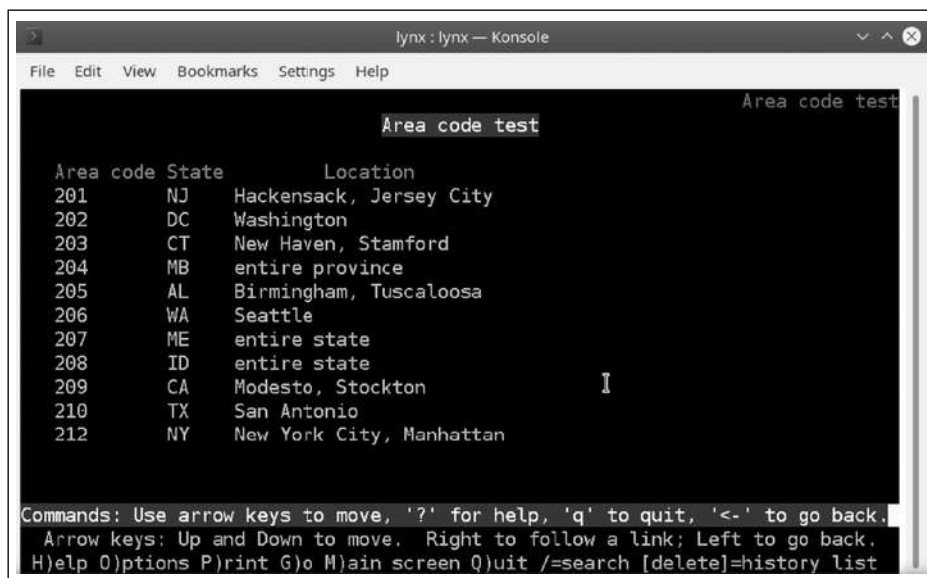


Рис. 10.2. Отображение страницы <https://efficientlinux.com/areacodes.html> в `lynx`

`lynx` и `links` загружают веб-страницу с параметром `-dump`:

```
$ lynx -dump https://efficientlinux.com/areacodes.html > tempfile
$ cat tempfile
```

```

                                Area code test
Area code State  Location
201      NJ     Hackensack, Jersey City
202      DC     Washington
203      CT     New Haven, Stamford
:
```



`lynx` и `links` отлично подходят для проверки ссылок, когда вы не уверены в их происхождении. Эти текстовые браузеры не поддерживают JavaScript и не отображают изображения, поэтому они менее уязвимы для атак (конечно, они не гарантируют безопасность, поэтому действуйте на свой страх и риск).

## Управление буфером обмена из командной строки

Каждый современный программный пакет, имеющий меню Правка (Edit), содержит операции вырезания, копирования и вставки для переноса содержимого в системный буфер обмена и из него. Возможно, вы знаете сочетания клавиш для этих операций. Но знаете ли вы, что можете обрабатывать содержимое буфера обмена прямо из командной строки?

Сначала немного предыстории: операции копирования и вставки в Linux являются частью более общего механизма, называемого *X-буферами обмена* (*X selections*). Буфер обмена — это общее название для места нахождения скопированного содержимого, например системного буфера обмена. X — это название оконной системы Linux.

Большинство построенных на X окружений рабочего стола Linux, таких как GNOME, Unity, Cinnamon и KDE Plasma, поддерживают два варианта буфера обмена<sup>1</sup>. Во-первых, это *системный буфер обмена* (*clipboard*), и он работает так же, как буфер обмена в других операционных системах. Когда вы выполняете операции вырезания или копирования в приложении, содержимое помещается в буфер обмена, и затем вы извлекаете его с помощью операции вставки. Менее знаком пользователям *первичный буфер обмена* (*primary selection*). Когда вы выбираете текст в определенных приложениях, он записывается в первичный буфер обмена, даже если вы не запустили операцию копирования. Примером может служить выделение текста в окне терминала с помощью мыши. Этот текст автоматически записывается в первичный буфер обмена.



Если вы подключаетесь к хосту Linux удаленно с помощью SSH или подобных программ, копирование/вставка обычно выполняются локальным компьютером, а не буфером обмена на удаленном хосте Linux.

<sup>1</sup> На самом деле существуют три буфера обмена, но один из них, называемый *вторичным* (*secondary selection*), редко используется в современных окружениях рабочего стола.

В таблице 10.2 перечислены операции с мышью и клавиатурой для доступа к буферам обмена в терминалах GNOME (`gnome-terminal`) и KDE Konsole (`konsole`). Если вы используете другую терминальную программу, проверьте, есть ли в ее меню Правка (Edit) операции, эквивалентные копированию (Copy) и вставке (Paste).

**Таблица 10.2.** Доступ к буферам обмена в терминальных программах

Действие	Системный буфер обмена	Первичный буфер обмена
Копирование мышью	Откройте меню правой кнопки и выберите пункт Copy	Нажмите левую кнопку и переместите или дважды нажмите, чтобы выбрать текущее слово, либо нажмите трижды, чтобы выбрать текущую строку
Вставка мышью	Откройте меню правой кнопки и выберите пункт Paste	Нажмите среднюю кнопку мыши (обычно это колесо прокрутки)
Копирование с клавиатуры	Ctrl-Shift-C	—
Вставка с клавиатуры, <code>gnome-terminal</code>	Ctrl-Shift-V или Ctrl-Shift-Insert	Shift-Insert
Вставка с клавиатуры, <code>konsole</code>	Ctrl-Shift-V или Shift-Insert	Ctrl-Shift-Insert

## Подключение буферов обмена к `stdin` и `stdout`

В Linux есть команда `xclip`, которая соединяет X-буферы обмена со `stdin` и `stdout`. Благодаря ей можно вставлять операции копирования и вставки в конвейеры и другие составные команды. Например, можно скопировать текст в приложение следующим образом:

1. Запустить команду Linux и перенаправить ее вывод в файл.
2. Просмотреть файл.
3. С помощью мыши скопировать содержимое файла в буфер обмена.
4. Вставить содержимое в другое приложение.

С `xclip` мы можем значительно сократить этот процесс:

1. Направить вывод команды Linux в `xclip`.
2. Вставить содержимое в другое приложение.

И, наоборот, можно вставить текст в файл, чтобы обработать его с помощью команд Linux. Обычная последовательность действий:

1. Использовать мышь, чтобы скопировать текст в текущую программу.
2. Вставить его в текстовый файл.
3. Обработать текстовый файл командами Linux.

С помощью `xclip -o` можно обойтись без промежуточного текстового файла:

1. С помощью мыши скопировать текст в текущую программу.
2. Передать вывод `xclip -o` другим командам Linux для обработки.



Если вы читаете цифровую версию этой книги на устройстве с ОС Linux и хотите попробовать некоторые команды `xclip` из этого раздела, не копируйте и не вставляйте команды в окно оболочки. Вводите команды вручную. Почему? Потому что ваша операция копирования может перезаписать тот же X-буфер обмена, к которому команды обращаются с помощью `xclip`, что приведет к неожиданным результатам.

По умолчанию команда `xclip` читает стандартный ввод и записывает в первичный буфер обмена. Она может читать из файла:

```
$ xclip < myfile.txt
```

или из канала:

```
$ echo "Efficient Linux at the Command Line" | xclip
```

Теперь выведем текст в стандартный вывод или передадим выделенное содержимое другим командам, таким, например, как `wc`:

<code>\$ xclip -o</code>	<i>Вставка в stdout</i>
<code>Efficient Linux at the Command Line</code>	
<code>\$ xclip -o &gt; anotherfile.txt</code>	<i>Вставка в файл</i>
<code>\$ xclip -o   wc -w</code>	<i>Подсчет количества слов</i>
<code>6</code>	

Любая составная команда, записывающая данные в стандартный вывод, может передавать свои результаты в `xclip`, как, например, следующая из раздела «Команда #6: `uniq`» на с. 30:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1 | cut -c9 | xclip
```

Очистим первичный буфер обмена, поместив в него пустую строку с помощью команды `echo -n`:

```
$ echo -n | xclip
```

Параметр `-n` важен, так как в противном случае `echo` выводит в стандартный вывод символ новой строки, который оказывается в первичном буфере обмена.

Чтобы скопировать текст в системный буфер обмена вместо первичного, запустим `xclip` с параметром `-selection clipboard`:

```
$ echo https://oreilly.com | xclip -selection clipboard      Копировать
$ xclip -selection clipboard -o                             Вставить
https://oreilly.com
```

Параметры `xclip` могут быть сокращены, если они недвусмысленны:

```
$ xclip -sel c -o То же самое, что и xclip -selection clipboard -o
https://oreilly.com
```

Запустим окно браузера Firefox, чтобы посетить предыдущий URL-адрес, используя подстановку команд:

```
$ firefox $(xclip -selection clipboard -o)
```

В Linux имеется и другая команда — `xsel`, которая также считывает и записывает X-буферы обмена. У нее есть несколько дополнительных функций, таких как очистка выделения (`xsel -c`) и добавление к выделению (`xsel -a`). Изучите справочную страницу и поэкспериментируйте с `xsel`.

## Улучшение работы менеджера паролей

Давайте воспользуемся вашими новыми знаниями о `xclip`, чтобы интегрировать X-буферы обмена в менеджер паролей `pass` из раздела «Создание менеджера паролей» на с. 190. Когда модифицированный сценарий `pass` находит соответствие одной строке в файле `vault.gpg`, он записывает имя пользователя в системный буфер обмена, а пароль — в первичный. После этого вы можете, например, заполнить любую страницу входа в интернете, вставив имя пользователя с помощью `Ctrl-V`, а пароль — с помощью средней кнопки мыши.



Убедитесь, что вы не используете менеджер буфера обмена или другие приложения, которые отслеживают X-буферы обмена и их содержимое. В противном случае имена пользователей и/или пароли станут видны в диспетчере буфера обмена, что представляет угрозу безопасности.

Новая версия `pass` приведена в листинге 10.3. Поведение `pass` изменилось следующим образом:

- Новая функция `load_password` загружает связанные имя пользователя и пароль в X-буферы обмена.
- Если сценарий `rmap` находит единственное совпадение либо по ключу (поле 3), либо по любой другой части строки, он запускает `load_password`.
- Если `rmap` находит несколько совпадений, он печатает все ключи и примечания (поля 3 и 4) из совпадающих строк, чтобы пользователь мог снова выполнить поиск по ключу.

**Листинг 10.3.** Улучшенный сценарий `rmap`, загружающий имя пользователя и пароль в качестве элементов буферов обмена

```
#!/bin/bash
PROGRAM=$(basename $0)
DATABASE=$HOME/etc/vault.gpg
load_password () {
    # Помещает имя пользователя (поле 1) в системный буфер обмена
    echo "$1" | cut -f1 | tr -d '\n' | xclip -selection clipboard
    # Помещает пароль (поле 2) в первичный буфер обмена
    echo "$1" | cut -f2 | tr -d '\n' | xclip -selection primary
    # Сообщение для пользователя
    echo "$PROGRAM: Found» $(echo "$1" | cut -f3- --output-delimiter ': ')"
    echo "$PROGRAM: username and password loaded into X selections"
}
if [ $# -ne 1 ]; then
    >&2 echo "$PROGRAM: look up passwords"
    >&2 echo "Usage: $PROGRAM string"
    exit 1
fi
searchstring="$1"

# Сохраняет расшифрованный текст в переменной
decrypted=$(gpg -d -q "$DATABASE")
if [ $? -ne 0 ]; then
    >&2 echo "$PROGRAM: could not decrypt $DATABASE"
    exit 1
fi

# Ищет точные совпадения в третьем столбце
match=$(echo "$decrypted" | awk '$3~/^'$searchstring'$/' )
if [ -n "$match" ]; then
    load_password "$match"
    exit $?
fi

# Ищет любые совпадения
match=$(echo "$decrypted" | awk "/$searchstring/")
if [ -z "$match" ]; then
    >&2 echo "$PROGRAM: no matches"
    exit 1
fi
```



*# Подсчитывает количество совпадений*

```
count=$(echo "$match" | wc -l)

case "$count" in
  0)
    >&2 echo "$PROGRAM: no matches"
    exit 1
    ;;
  1)
    load_password "$match"
    exit $?
    ;;
  *)
    >&2 echo "$PROGRAM: multiple matches for the following keys:"
    echo "$match" | cut -f3
    >&2 echo "$PROGRAM: rerun this script with one of the keys"
    exit
    ;;
esac
```

Запустим сценарий:

```
$ pman dropbox
Passphrase: xxxxxxxx
pman: Found dropbox: dropbox.com account for work
pman: username and password loaded into X selections
$ pman account
Passphrase: xxxxxxxx
pman: multiple matches for the following keys:
google
dropbox
bank
dropbox2
pman: rerun this script with one of the keys
```

Пароли находятся в первичном буфере обмена, пока он не будет перезаписан. Чтобы автоматически сбросить пароль через, например, 30 секунд, добавьте следующую строку в функцию `load_password`:

```
(sleep 30 && echo -n | xclip -selection primary) &
```

Эта строка запускает подоболочку в фоновом режиме, а через 30 секунд ожидания команда очищает первичный буфер обмена, записывая в него пустую строку. Число секунд ожидания может быть установлено на ваше усмотрение.

Если вы определили сочетание клавиш для запуска окон терминала (см. раздел «Мгновенный запуск оболочек и браузера» на с. 201), то теперь у вас есть быстрый способ доступа к своим паролям. Откройте терминал с помощью горячей клавиши, запустите `pman` и закройте терминал.

## Резюме

Надеюсь, что эта глава вдохновила вас на использование новых приемов, позволяющих «держать руки на клавиатуре». Сначала они могут показаться трудными, но с практикой дойдут до автоматизма. Тогда ваши навыки станут предметом зависти друзей и коллег, поскольку вы сможете быстро манипулировать окнами рабочего стола, веб-контентом и буферами обмена, так, как не умеет большая часть пользователей, привязанных к мышке.

## ГЛАВА 11

---

# Финальные советы по экономии времени

Мне было весело и приятно писать эту книгу, и я надеюсь, что вы получили удовольствие от ее чтения. Напоследок давайте рассмотрим несколько более мелких тем, которым не нашлось места в предыдущих главах. Эти темы позволили мне лучше использовать возможности Linux, и, возможно, они помогут и вам.

## Способы решения задач легко и быстро

Некоторые способы экономии времени легко освоить за несколько минут.

### Переход в текстовый редактор напрямую из команды `less`

Когда вы просматриваете текстовый файл с помощью `less` и хотите отредактировать его, не выходите из программы `less`. Просто нажмите `v`, чтобы запустить ваш текстовый редактор, установленный по умолчанию. Он загружает файл и помещает курсор в место, на котором вы находились при просмотре с помощью `less`. Выйдите из редактора, и вы вернетесь к стандартному выводу команды `less`.

Чтобы этот трюк работал лучше, установите в переменные окружения `EDITOR` и/или `VISUAL` команду вызова вашего любимого текстового редактора. В этих переменных окружения задается текстовый редактор Linux по умолчанию, который можно запустить при выполнении различных команд, включая `less`,

lynx, git, crontab и многочисленных программ для работы с электронной почтой. Например, чтобы установить Emacs в качестве редактора по умолчанию, поместите одну из следующих строк (или обе) в файл конфигурации оболочки и примените его:

```
VISUAL=emacs  
EDITOR=emacs
```

Если вы сами не установите эти переменные, по умолчанию будет запускаться редактор, предустановленный в вашей системе Linux, обычно это Vim. Если вы оказались в Vim и не знаете, как его использовать, не паникуйте. Выйдите, нажав **Escape**, а затем, набрав **:q!**, нажмите **Enter**. Чтобы выйти из Emacs, нажмите **Ctrl-X**, а затем **Ctrl-C**.

## Редактирование файлов, содержащих заданную строку

Хотите отредактировать каждый файл в текущем каталоге, который содержит определенную строку (или регулярное выражение)? Сгенерируйте список имен файлов с помощью **grep -l** и передайте их вашему редактору с помощью подстановки команд. Если ваш редактор — Vim, тогда команда выглядит следующим образом:

```
$ vim $(grep -l string *)
```

Отредактируйте все файлы, содержащие заданную строку, во всем дереве каталогов (текущем каталоге и всех подкаталогах), добавив параметр **-r** (рекурсивный) в **grep** и начав с текущего каталога (точка):

```
$ vim $(grep -lr string .)
```

Для более быстрого поиска в разветвленных каталогах используйте **find** вместе с **xargs** вместо **grep -r**:

```
$ vim $(find . -type f -print0 | xargs -0 grep -l string)
```

В разделе «Способ #3: подстановка команд» на с. 138 эта техника уже обсуждалась, но напомнить о ней еще раз было необходимо, так как она действительно очень полезна. Не забывайте про имена файлов, содержащие пробелы и другие специальные символы, поскольку они могут привести к неожиданным результатам, как описано в разделе «Специальные символы и подстановка команд» на с. 139.

## Смиритесь с опечатками

Если вы постоянно ошибаетесь в написании команд, определите псевдонимы для ваших наиболее распространенных ошибок, чтобы правильная команда все равно выполнялась:

```
alias firfox=firefox
alias les=less
alias meacs=emacs
```

Будьте осторожны, чтобы случайно не затенить (переопределить) существующую команду Linux, определив псевдоним с тем же именем. Сначала поищите предложенный псевдоним с помощью команд `which` или `type` (см. раздел «Расположение исполняемых программ» на с. 49) и запустите команду `man`, чтобы убедиться, что нет другой команды с таким же именем:

```
$ type firfox
bash: type: firfox: not found
$ man firfox
No manual entry for firfox
```

## Быстрое создание пустых файлов

В Linux существует несколько способов создания пустых файлов. Команда `touch` обновляет метку времени в файле или создает файл, если он еще не существует:

```
$ touch newfile1
```

`touch` отлично подходит для создания большого количества пустых файлов для тестирования:

<code>\$ mkdir tmp</code>	<i>Создать каталог</i>
<code>\$ cd tmp</code>	
<code>\$ touch file{0000..9999}.txt</code>	<i>Создать 10 000 файлов</i>
<code>\$ cd ..</code>	
<code>\$ rm -rf tmp</code>	<i>Удалить каталог и файлы</i>

Команда `echo` создаст пустой файл, когда ее вывод перенаправляется в файл, но только если будет указана опция `-n`:

```
$ echo -n > newfile2
```

Если вы забудете параметр `-n`, итоговый файл будет содержать один символ новой строки, поэтому не будет пустым.

## Обработка файла построчно

Когда вам нужно обработать файл построчно, запустите его в цикле `while read`:

```
$ cat myfile | while read line; do
...делайте что-нибудь здесь...
done
```

Например, чтобы вычислить длину каждой строки файла `/etc/hosts`, передайте каждую строку в `wc -c`:

```
$ cat /etc/hosts | while read line; do
    echo "$line" | wc -c
done
65
31
1
:
```

Более практичный пример этого метода приведен в листинге 9.3.

## Список команд, поддерживающих рекурсию

В разделе «Команда `find`» на с. 93 мы познакомились с командой `find -exec`, которая рекурсивно применяет любую команду Linux ко всему дереву каталогов:

```
$ find . -exec ваша команда здесь \;
```

Некоторые другие команды поддерживают рекурсию, и если вы знаете о них, то сэкономите время, используя эту возможность вместо создания команды `find`:

```
ls -R
```

чтобы рекурсивно перебирать каталоги и их содержимое;

```
cp -r или cp -a
```

для рекурсивного копирования каталогов и их содержимого;

```
rm -r
```

для рекурсивного удаления каталогов и их содержимого;

```
grep -r
```

для поиска по регулярному выражению в дереве каталогов;

`chmod -R`

чтобы рекурсивно изменять права на доступ к файлам;

`chown -R`

чтобы рекурсивно изменять владельцев файлов;

`chgrp -R`

чтобы рекурсивно изменять принадлежность файлов к группе.

## Читайте справочные страницы

Выберите часто используемую команду, например `cut` или `grep`, и внимательно прочитайте ее справочную страницу. Наверняка вы обнаружите возможности, которыми никогда не пользовались, и найдете их полезными. Периодически обращайтесь к справке, чтобы оттачивать и расширять ваш инструментарий.

## Способы решения задач, требующие затрат времени на изучение

Следующие методы требуют определенных усилий для изучения, но затраты сил и времени окупятся за счет сэкономленного времени в будущем. Краткие обзоры каждой темы приводятся для того, чтобы побудить вас узнать больше самостоятельно.

### Прочтите справочную страницу команды `bash`

Запустите `man bash`, чтобы отобразить полную официальную документацию по `bash`, и прочитайте ее целиком — да, более 40 тыс. слов:

```
$ man bash | wc -w
46318
```

Не торопитесь, прочтите все внимательно, даже если потребуется несколько дней. После этого вы наверняка сможете упростить работу в Linux.

## Изучите команды `cron`, `crontab` и `at`

В разделе «Первый пример: поиск файлов» на с. 184 есть краткое примечание о планировании автоматического запуска команд через равные промежутки времени. Изучите программу `crontab`, чтобы настроить планирование команд. Например, вы можете создавать резервные копии файлов на внешнем диске по расписанию или отправлять себе напоминания по электронной почте о регулярных мероприятиях.

Вначале определите редактор по умолчанию (см. «Переход в текстовый редактор напрямую из команды `less`» на с. 220). Запустите `crontab -e`, чтобы отредактировать личный файл запланированных команд. `crontab` запускает редактор по умолчанию и открывает пустой файл `crontab` для указания команд.

Запланированная команда в файле `crontab`, часто называемая *заданием cron*, состоит из шести полей, расположенных в одной строке. В первых пяти полях указывается расписание задания — минуты, часы, день, месяц и день недели. Шестое поле предназначено для команды Linux. Вы можете запускать команду ежечасно, ежедневно, еженедельно, ежемесячно, ежегодно, в определенные дни или время либо задавать другие, более сложные комбинации. Примеры:

<code>* * * * * command</code>	<i>Запуск команды каждую минуту</i>
<code>30 7 * * * command</code>	<i>Запуск команды в 07:30 каждый день</i>
<code>30 7 5 * * command</code>	<i>Запуск команды в 07:30 пятого числа каждого месяца</i>
<code>30 7 5 1 * command</code>	<i>Запуск команды в 07:30 пятого января каждого года</i>
<code>30 7 * * 1 command</code>	<i>Запуск команды в 07:30 каждый понедельник</i>

Когда вы заполнили все шесть полей, сохранили файл и вышли из редактора, команда запускается автоматически программой `cron` в соответствии с заданным расписанием. Синтаксис расписаний хорошо задокументирован на справочной странице (`man 5 crontab`) и в многочисленных онлайн-учебниках (поищите в интернете «*учебник по cron*» («*cron tutorial*»)).

Я также рекомендую изучить команду `at`, которая планирует однократное выполнение команд в указанную дату и время. За подробностями обращайтесь к справочной странице (`man at`). Пример команды, которая отправит вам напоминание по электронной почте завтра в 22:00 о чистке зубов (*brush your teeth*):

```
$ at 22:00 tomorrow
```

```
warning: commands will be executed using /bin/sh
```

```
at> echo brush your teeth | mail $USER
```

```
at> ^D
```

```
job 699 at Sun Nov 14 22:00:00 2021
```

*Нажмите Ctrl-D для завершения ввода*



Чтобы получить список ожидающих выполнения заданий **at**, запустите **atq**:

```
$ atq
699      Sun Nov 14 22:00:00 2021 a smith
```

Чтобы просмотреть команды в задании **at**, запустите **at -c** с номером задания и выведите на экран последние несколько строк:

```
$ at -c 699 | tail
:
echo brush your teeth | mail $USER
```

Чтобы удалить ожидающее задание до его выполнения, запустите **atrm** с номером задания:

```
$ atrm 699
```

## Изучите команду **rsync**

Для копирования полного каталога, включая его подкаталоги, из одного места на диске в другое многие пользователи Linux используют команду **cp -r** или **cp -a**:

```
$ cp -a dir1 dir2
```

**cp** отлично справляется с задачей в первый раз, но, если позже вы измените несколько файлов в каталоге *dir1* и снова выполните копирование, команда **cp** будет не очень эффективна. Она добросовестно копирует все файлы и каталоги из *dir1* снова и снова, даже если идентичные копии уже существуют в *dir2*.

Команда **rsync** — более умная программа, которая копирует только *различия* между первым и вторым каталогами:

```
$ rsync -a dir1/ dir2
```



Слеш в предыдущей команде означает копирование файлов из *dir1*. Без косой черты **rsync** скопировала бы и сам каталог *dir1*, создав *dir2/dir1*.

Если вы позже добавите файл в каталог *dir1*, **rsync** скопирует только его. Если вы измените одну строку внутри файла в каталоге *dir1*, **rsync** скопирует только эту строку! Это значительно экономит время при многократном копировании больших каталогов. Также **rsync** может копировать на удаленный сервер через SSH-соединение.

`rsync` имеет десятки параметров. Вот некоторые особенно полезные:

`-v` (от *verbose* — *подробный*)

Вывод на экран имен файлов по мере их копирования.

`-n`

Имитация копирования. Комбинируйте с `-v`, чтобы увидеть, какие файлы будут скопированы.

`-x`

Указывает `rsync` не пересекать границы файловой системы.

Я настоятельно рекомендую освоить работу с `rsync` для более эффективного копирования. Прочтите справочную страницу и просмотрите примеры в статье *Rsync Examples in Linux* Корбина Брауна (Korbin Brown), <https://linuxconfig.org/rsync-command-examples>.

## Изучите другой язык для написания сценариев

Сценарии оболочки удобны и эффективны, но имеют ряд серьезных недостатков. Например, они плохо обрабатывают имена файлов, содержащие пробельные символы. Рассмотрим короткий сценарий `bash` для удаления файла:

```
#!/bin/bash
BOOKTITLE="Slow Inefficient Linux"
rm $BOOKTITLE # Ошибка! Не делайте этого!
```

Кажется, что вторая строка указывает на файл с именем *Slow Inefficient Linux*, но это не так. Сценарий будет пытаться удалить три файла с именами *Slow*, *Inefficient* и *Linux*. Оболочка вычисляет переменную `$BOOKTITLE` перед вызовом `rm`, и ее расширение состоит из трех слов, разделенных пробелами, как если бы мы набрали следующее:

```
rm Slow Efficient Linux
```

Затем оболочка вызывает `rm` с тремя аргументами, что может привести к неприятностям, поскольку она попытается удалить не те файлы. В правильной команде удаления `$BOOKTITLE` необходимо заключить в двойные кавычки:

```
rm "$BOOKTITLE"
```

которые оболочка расширяет до:

```
rm "Slow Efficient Linux"
```

Такая неочевидная потенциально опасная особенность — лишь один из многих примеров, показывающих, что сценарии оболочки непригодны для сложных проектов. Поэтому рекомендую изучить и использовать для написания сценариев такой язык, как, например, Perl, PHP, Python или Ruby. Все они правильно обрабатывают пробелы, поддерживают реальные структуры данных, имеют мощные функции обработки строк и удобны для математических расчетов. Список преимуществ можно продолжить.

Используйте оболочку для запуска сложных команд и создания простых скриптов, но в случае ответственных задач обратитесь к другому языку. Попробуйте один из многочисленных онлайн-курсов по понравившемуся языку.

## Используйте `make` для задач, не связанных с программированием

Программа `make` автоматически обновляет файлы на основе списка правил. Она предназначена, в первую очередь, для ускорения разработки программного обеспечения, но, если приложить небольшие усилия, может упростить и другие задачи.

Рассмотрите использование `make` при следующих условиях:

- Есть группа файлов, требующих обновления.
- Имеется правило, которое связывает файлы, например: *book.txt* требует обновления всякий раз, когда изменяется какой-либо файл главы.
- Есть команда, выполняющая обновление.

`make` считывает файл конфигурации, обычно называемый *Makefile*, в котором собраны правила и команды. Например, в следующем *Makefile* указано, что *book.txt* зависит от трех файлов глав:

```
book.txt: chapter1.txt chapter2.txt chapter3.txt
```

Если целевой файл (в данном случае *book.txt*) старше любого из зависимых (файлов глав), то `make` считает его устаревшим. Если в строке после правила указана команда, `make` запускает ее для обновления целевого файла:

```
book.txt: chapter1.txt chapter2.txt chapter3.txt
cat chapter1.txt chapter2.txt chapter3.txt > book.txt
```

Чтобы применить правило, просто запустим `make`:

```
$ ls
Makefile chapter1.txt chapter2.txt chapter3.txt
$ make
cat chapter1.txt chapter2.txt chapter3.txt > book.txt Выполняется команда из
Makefile
$ ls
Makefile book.txt chapter1.txt chapter2.txt chapter3.txt
$ make
make: 'book.txt' is up to date.
$ vim chapter2.txt Обновим файл chapter2.txt
$ make
cat chapter1.txt chapter2.txt chapter3.txt > book.txt
```

Команда `make` была разработана для программистов, но после недолгого изучения вы сможете использовать ее для задач, не связанных с программированием. Если вам нужно обновить файлы, которые зависят от других файлов, вы сможете упростить работу, написав *Makefile*.

Команда `make` помогла мне при написании и редактировании этой книги. Я использовал язык форматирования текста AsciiDoc и регулярно преобразовывал главы в HTML для просмотра в браузере. Вот правило `make` для преобразования файла AsciiDoc в файл HTML:

```
%.html: %.asciidoc
asciidoc -o $@ $<
```

Оно гласит: чтобы создать файл с расширением *.html* (`%.html`), надо использовать файл с расширением *.asciidoc* (`%.asciidoc`). Если файл HTML старше, чем файл AsciiDoc, следует повторно создать файл HTML, запустив команду `asciidoc` для зависимого файла (`$<`) и отправив вывод в целевой файл HTML (`-o $@`). С помощью этого короткого правила достаточно набрать команду `make`, чтобы получить HTML-версию главы, которую вы сейчас читаете. `make` запускает `asciidoc` для выполнения обновления:

```
$ ls ch11*
ch11.asciidoc
$ make ch11.html
asciidoc -o ch11.html ch11.asciidoc
$ ls ch11*
ch11.asciidoc ch11.html
$ firefox ch11.html Просмотр HTML файла
```

Требуется менее часа, чтобы научиться использовать `make` для небольших задач. И это стоит потраченных усилий. Полезное руководство находится по адресу: [makefiletutorial.com](http://makefiletutorial.com).

## Применяйте контроль версий к повседневным файлам

Допустим, что нам требуется отредактировать файл, но мы опасаемся его испортить. Можно сделать резервную копию для надежности и отредактировать оригинал, зная, что сможем восстановить резервную копию, если допустим ошибку:

```
$ cp myfile myfile.bak
```

Но это решение не масштабируется. Что, если у нас есть десятки или сотни файлов и над ними работают десятки или сотни людей? Системы контроля версий, такие как Git и Subversion, были изобретены для решения этой проблемы путем отслеживания нескольких версий файла.

Git широко используется для поддержки исходного кода программного обеспечения, но я рекомендую изучить и использовать его для любых важных текстовых файлов, в которые могут вноситься изменения. Возможно, это личные файлы или файлы операционной системы в */etc*. В разделе «Путешествие с вашим окружением» на с. 132 предлагалось отслеживать файлы конфигурации *bash* с помощью контроля версий.

При написании этой книги я использовал Git, чтобы сравнивать разные способы подачи материала. Без особых усилий я создал и отслеживал три разные версии книги: одну с полным текстом, другую, содержащую только главы, которые я отправил своему редактору на проверку, и одну для экспериментов, в которой я пробовал новые идеи. Если мне не нравилось написанное, всего лишь одна команда восстанавливала предыдущую версию файла.

Изучение Git выходит за рамки этой книги, но вот несколько примеров команд, которые покажут базовый рабочий процесс и, возможно, понравятся вам. Преобразуем текущий каталог (и все его подкаталоги) в репозиторий Git:

```
$ git init
```

Отредактируем некоторые файлы. После этого добавим измененные файлы в невидимую «промежуточную область», что заявляет о нашем намерении создать новую версию:

```
$ git add .
```

Создаем новую версию, оставив комментарий для описания изменений в файлах:

```
$ git commit -m"Changed X to Y"
```

Просматриваем историю версий:

```
$ git log
```

На самом деле Git умеет гораздо больше, например получать старые версии файлов и сохранять (передавать) версии на другой сервер. Прочитайте руководство по git (<https://www.w3schools.com/git/>) и приступайте к работе!

## Прощание

Большое спасибо за то, что дочитали эту книгу до конца. Надеюсь, что мне удалось выполнить данное в предисловии обещание, и после прочтения книги ваши навыки работы с командной строкой Linux вышли на новый уровень. Поделитесь своими впечатлениями по email [dbarrett@oreilly.com](mailto:dbarrett@oreilly.com). Работайте с Linux с удовольствием!



# Памятка по Linux

Если ваши навыки работы с Linux необходимо освежить, в этом приложении приведен краткий обзор знаний, которые вам понадобятся при чтении этой книги. Если вы новичок, обзор может оказаться слишком лаконичным, тогда ознакомьтесь с дополнительной литературой, указанной в конце приложения.

## Команды, аргументы и параметры

Чтобы запустить команду Linux в командной строке, введите команду и нажмите **Enter**. Чтобы завершить выполняемую команду, нажмите **Ctrl-C**.

Простая команда Linux состоит из одного слова, которое обычно является именем программы, за ним следуют дополнительные строки, называемые *аргументами*. Например, следующая команда состоит из имени программы `ls` и двух аргументов:

```
$ ls -l /bin
```

Аргументы, начинающиеся с тире, такие как `-l`, называются *параметрами*, потому что они изменяют поведение команды. Другими аргументами могут быть имена файлов, каталогов, пользователей и хостов или любые другие строки, необходимые программе. Параметры обычно (но не всегда) предшествуют остальным аргументам.

Параметры команд бывают разных форм, в зависимости от того, какую программу вы запускаете:

- За параметрами, обозначенными одной буквой с тире, иногда следует значение, например `-n 10`. Обычно пробел между буквой и значением может быть опущен: `-n10`.



- За словом, которому предшествуют два тире, также может следовать значение, например `--block-size 100`. Пробел между параметром и его значением часто может быть заменен знаком равенства: `--block-size=100`.
- За словом, которому предшествует одно тире, может следовать значение, как в случае `-type f`, но этот вариант формата встречается редко. Одна из команд, которые его используют, — это `find`.
- Формат «одна буква без тире» встречается редко. Одна из команд, которые его используют, — `tar`.

Несколько параметров иногда могут быть объединены за одним тире (это зависит от команды). Например, команда `ls -al` эквивалентна `ls -a -l`.

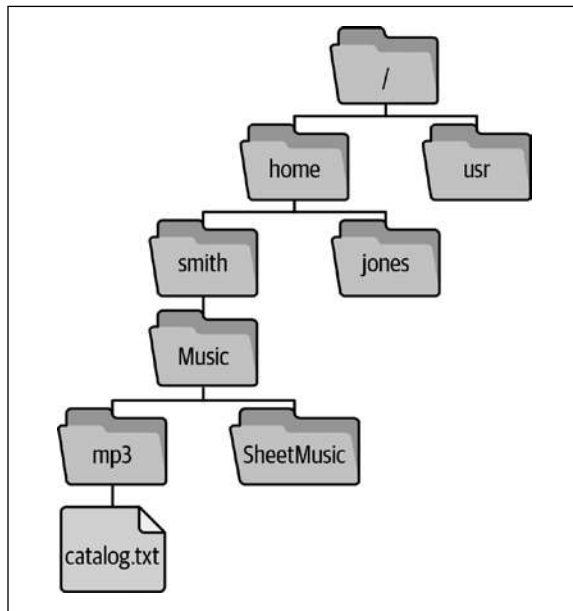
Варианты форматов различаются не только по внешнему виду, но и по смыслу. В команде `ls -l` параметр `-l` означает «длинный вывод», а в команде `wc` параметр `-l` — «строки текста». При этом разные программы могут использовать параметры с различными обозначениями для одной и той же операции, например `-q` и `-s` для «тихого запуска». Подобные несоответствия затрудняют изучение Linux, но со временем к ним привыкаешь.

## Файловая система, каталоги и пути

Файлы Linux содержатся в каталогах (папках), организованных в виде древовидной структуры (рис. А.1). Дерево начинается в каталоге, который называется *корневым* и обозначается косой чертой (`/`). Он может содержать файлы и другие каталоги, называемые *подкаталогами*. Например, каталог *Music* имеет два подкаталога: *mp3* и *SheetMusic*. Мы называем *Music* родительским каталогом для *mp3* и *SheetMusic*. Каталоги с одним и тем же родителем называются *одноуровневыми*.

Путь в дереве записывается как иерархическая последовательность имен каталогов, разделенных косой чертой, например `/home/smith/Music/mp3`. Путь может заканчиваться именем файла, например `/home/smith/Music/mp3/catalog.txt`. Эти пути называются *абсолютными*, поскольку они начинаются в корневом каталоге. Пути, которые не начинаются в корневом каталоге (и их обозначения не начинаются с косой черты), называются *относительными*, поскольку они относятся к текущему каталогу. Если ваш текущий каталог — `/home/smith/Music`, то относительными будут, в том числе, следующие пути: *mp3* (подкаталог); *mp3/catalog.txt* (файл). Даже имя файла само по себе, например *catalog.txt*, — это относительный путь по отношению к `/home/smith/Music/mp3`.

Два специальных относительных пути — это одна точка (`.`), которая указывает на текущий каталог, и две точки подряд (`..`), которые указывают на родительский текущий каталог<sup>1</sup>. Оба могут быть частью более длинных путей. Например, если ваш текущий каталог — `/home/smith/Music/mp3`, то путь `..` относится к `Music`, путь `../../../../..` относится к корневому каталогу, а путь `../SheetMusic` относится к одноуровневому каталогу `mp3`.



**Рис. А.1.** Пример дерева каталогов Linux

У каждого пользователя Linux есть назначенный каталог, называемый *домашним*, где можно свободно создавать, редактировать и удалять файлы и каталоги. Его путь обычно начинается с `/home/`, за которым следует имя пользователя, например `/home/smith`.

<sup>1</sup> Точка и двойная точка не являются выражениями, вычисляемыми оболочкой. Это жесткие ссылки, присутствующие в каждом каталоге.

## Перемещение по каталогам

Ваша командная строка (оболочка) работает в каталоге, называемом *текущим*, или *рабочим*. Просмотрите путь к вашему текущему каталогу с помощью команды `pwd` (*print working directory*):

```
$ pwd
/home/smith          Домашний каталог пользователя smith
```

Перемещайтесь между каталогами с помощью команды `cd` (*change directory*), указав путь — абсолютный или относительный — к месту назначения:

```
$ cd /usr/local      Абсолютный путь
$ cd bin             Относительный путь, ведущий к /usr/Local/bin
$ cd ../etc          Относительный путь, ведущий к /usr/Local/etc
```

## Создание и редактирование файлов

Редактируйте файлы в стандартном текстовом редакторе Linux, выполнив любую из следующих команд:

`emacs`

После запуска `emacs` нажмите `Ctrl-h`, а затем `t` для обучения.

`nano`

Посетите <https://nano-editor.org/>, чтобы ознакомиться с документацией.

`vim` или `vi`

Запустите команду `vimtutor` для обучения.

Чтобы создать файл, просто укажите его имя в качестве аргумента — и редактор создаст его:

```
$ nano newfile.txt
```

Другой способ — создать пустой файл с помощью команды `touch`, указав желаемое имя файла в качестве аргумента:

```
$ touch funky.txt
$ ls
funky.txt
```

## Работа с файлами и каталогами

Выведите список файлов в каталоге (по умолчанию — в вашем текущем каталоге) с помощью команды `ls`:

```
$ ls
animals.txt
```

Атрибуты файла или каталога можно посмотреть в формате «длинного» (*long*) списка (`ls -l`):

```
$ ls -l
-rw-r--r-- 1 smith smith 325 Jul 3 17:44 animals.txt
```

Слева направо выводятся права доступа к файлу (`-rw-r--r--`), описанные в разделе «Права доступа к файлу» на с. 239, владелец (`smith`), группа (`smith`), размер в байтах (325), дата и время последнего изменения (3 июля этого года, 17:44), а также имя файла (`animals.txt`).

По умолчанию `ls` не печатает информацию о файлах, начинающихся с точки. Чтобы просмотреть такие файлы, которые часто называют *dotfiles* или *скрытыми*, добавьте параметр `-a`:

```
$ ls -a
.bashrc .bash_profile animals.txt
```

Скопируйте файл с помощью команды `cp`, указав исходное и новое имена:

```
$ cp animals.txt beasts.txt
$ ls
animals.txt beasts.txt
```

Переименуйте файл с помощью команды `mv` (*move*), указав исходное и новое имена:

```
$ mv beasts.txt creatures.txt
$ ls
animals.txt creatures.txt
```

Удалите файл с помощью команды `rm` (*remove*):

```
$ rm creatures.txt
```



Операция удаления в Linux не имеет дружелюбного интерфейса. Команда `rm` не спрашивает «Вы уверены?», и нет корзины для восстановления файлов.

Создайте каталог с помощью `mkdir`, переименуйте его с помощью `mv` и удалите (если он пустой) с помощью `rmdir`:

```
$ mkdir testdir
$ ls
animals.txt testdir
$ mv testdir newname
$ ls
animals.txt newname
$ rmdir newname
$ ls
animals.txt
```

Скопируйте один или несколько файлов (или каталогов) в каталог:

```
$ touch file1 file2 file3
$ mkdir dir
$ ls
dir file1 file2 file3
$ cp file1 file2 file3 dir
$ ls
dir file1 file2 file3
$ ls dir
file1 file2 file3
$ rm file1 file2 file3
```

Переместите один или несколько файлов (или каталогов) в другой каталог:

```
$ touch thing1 thing2 thing3
$ ls
dir thing1 thing2 thing3
$ mv thing1 thing2 thing3 dir
$ ls
dir
$ ls dir
file1 file2 file3 thing1 thing2 thing3
```

Удалите каталог и все его содержимое с помощью `rm -rf`. Будьте осторожны перед запуском этой команды, поскольку она необратима. Советы по безопасности см. в разделе «Забудьте об ошибочном удалении файлов (спасибо расширению истории)» на с. 60.

```
$ rm -rf dir
```

## Просмотр файлов

Выведите текстовый файл на экран с помощью команды `cat`:

```
$ cat animals.txt
```

Для поэкранного просмотра текстового файла выполните команду `less`:

```
$ less animals.txt
```

Во время работы команды `less` для перехода к отображению следующей страницы нажимайте пробел. Чтобы выйти, нажмите `q`. Для получения помощи нажмите `h`.

## Права доступа к файлам

Команда `chmod` делает файл доступным для чтения, записи и выполнения одним или определенной группой пользователей или всеми желающими. Рис. А.2 — краткое напоминание о правах доступа к файлам.

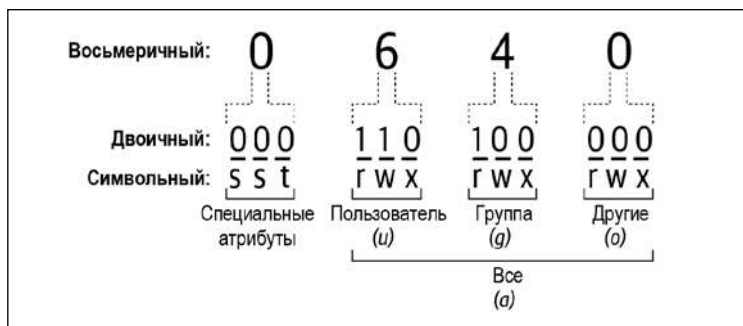


Рис. А.2. Кодирование прав доступа к файлам

Рассмотрим самые распространенные варианты команды `chmod`. Сделайте файл доступным для чтения и редактирования вами, и только для чтения всеми остальными:

```
$ chmod 644 animals.txt
```

```
$ ls -l
```

```
-rw-r--r-- 1 smith smith 325 Jul 3 17:44 animals.txt
```

Защитите его от других пользователей:

```
$ chmod 600 animals.txt
$ ls -l
-rw----- 1 smith smith 325 Jul 3 17:44 animals.txt
```

Сделайте каталог доступным для чтения и входа для всех, но для записи — только вам:

```
$ mkdir dir
$ chmod 755 dir
$ ls -l
drwxr-xr-x 2 smith smith 4096 Oct 1 12:44 dir
```

Защитите каталог от других пользователей:

```
$ chmod 700 dir
$ ls -l
drwx----- 2 smith smith 4096 Oct 1 12:44 dir
```

Обычные разрешения не распространяются на суперпользователя, который может читать и перезаписывать все файлы и каталоги в системе.

## Процессы

При запуске команды Linux запускаются один или несколько процессов Linux, каждый из которых имеет числовой идентификатор — *PID*. С помощью команды `ps` выведите текущие процессы вашей оболочки:

```
$ ps
  PID TTY          TIME CMD
  5152 pts/11        00:00:00 bash
 117280 pts/11        00:00:00 emacs
 117273 pts/11        00:00:00 ps
```

или все запущенные процессы для всех пользователей:

```
$ ps -uax
```

Завершите запущенный вами процесс с помощью команды `kill`, указав *PID* в качестве аргумента. Суперпользователь (администратор Linux) может завершить запущенный процесс любого пользователя.

```
$ kill 117280
[1]+  Exit 15                  emacs animals.txt
```

## Просмотр документации

Команда `man` выводит на экран документацию по любой стандартной команде вашей системы Linux. Просто введите `man`, а затем имя команды. Например, чтобы просмотреть документацию по команде `cat`, выполните следующее:

```
$ man cat
```

Отображаемый документ называется справочной страницей команды (`manpage`). Когда говорят «просмотрите справочную страницу для `grep`», имеют в виду запуск команды `man grep`.

`man` отображает документацию по одной странице за раз, используя программу `less`<sup>1</sup>, поэтому стандартные сочетания клавиш для `less` будут работать. В таблице A.1 перечислены некоторые распространенные сочетания клавиш.

**Таблица A.1.** Сочетания клавиш для просмотра справочных страниц с помощью команды `less`

Сочетание клавиш	Действие
<code>h</code>	Справка — отображение списка сочетаний клавиш для команды <code>less</code>
Пробел	Посмотреть следующую страницу
<code>b</code>	Посмотреть предыдущую страницу
<code>Enter</code>	Прокрутить вниз на одну строку
<code>&lt;</code>	Перейти к началу документа
<code>&gt;</code>	Перейти к концу документа
<code>/</code>	Поиск текста вперед (введите текст и нажмите <code>Enter</code> )
<code>?</code>	Поиск текста назад (введите текст и нажмите <code>Enter</code> )
<code>n</code>	Найти следующее вхождение искомого текста
<code>q</code>	Выйти из <code>man</code>

<sup>1</sup> Или другую программу, если вы переопределите значение переменной оболочки `PAGER`.



## Сценарии оболочки

Чтобы запустить несколько команд Linux как единое целое, выполните следующие действия:

1. Поместите команды в файл.
2. Вставьте волшебную первую строку.
3. Сделайте файл исполняемым с помощью `chmod`.
4. Запустите файл.

Этот файл называется *сценарием (оболочки)*. Волшебной первой строкой должны быть символы `#!`, а после них указывается путь к программе, которая считывает и запускает сценарий<sup>1</sup>:

```
#!/bin/bash
```

Ниже приведен сценарий оболочки, который передает приветствие и печатает сегодняшнюю дату. Строки, начинающиеся с `#`, являются комментариями:

```
#!/bin/bash
# Это просто пример скрипта
echo "Hello there!"
date
```

С помощью текстового редактора сохраните эти строки в файл с именем *howdy*. Затем сделайте файл исполняемым с помощью одной из команд:

```
$ chmod 755 howdy    Установите все разрешения, включая разрешение на выполнение
$ chmod +x howdy     Просто добавьте разрешение на выполнение
```

и запустите его:

```
$ ./howdy
Hello there!
Fri Sep 10 17:00:52 EDT 2021
```

---

<sup>1</sup> Если вы не укажете `#!`, то сценарий запустит ваша оболочка по умолчанию. Хорошим тоном считается явное указание используемой оболочки.

Начальная точка и косая черта (./) указывают на то, что сценарий находится в вашем текущем каталоге. Без них оболочка Linux не найдет сценарий<sup>2</sup>:

```
$ howdy
howdy: command not found
```

Оболочки Linux предоставляют некоторые функции языка программирования, полезные в сценариях. В `bash`, например, можно использовать операторы `if`, циклы `for`, циклы `while` и другие управляющие структуры. Несколько примеров разбросаны по всей книге. Описание синтаксиса ищите в `man bash`.

## Получение привилегий суперпользователя

Некоторые файлы, каталоги и программы защищены от обычных пользователей, включая вас:

```
$ touch /usr/local/avocado      Попробуйте создать файл в системном каталоге
touch: cannot touch '/usr/local/avocado': Permission denied
```

*Permission denied* (отказано в доступе) обычно означает, что вы пытались получить доступ к защищенным ресурсам. Они доступны только суперпользователю Linux (имя пользователя `root`). Большинство систем Linux поставляются с программой `sudo`, которая позволяет вам стать суперпользователем на время выполнения одной команды. Если вы установили Linux самостоятельно, ваша учетная запись, вероятно, уже настроена для запуска `sudo`. Если вы являетесь пользователем в чужой системе Linux, привилегии суперпользователя могут быть недоступны для вас — выясните это у системного администратора.

Предположим, что вы можете получить привилегии суперпользователя. Тогда просто выполните команду `sudo`, указав команду для запуска от имени суперпользователя. Вам будет предложено ввести пароль для входа в систему, после чего команда будет выполняться с привилегиями `root`:

---

<sup>2</sup> Это связано с тем, что текущий каталог обычно не указывается в пути поиска оболочки из соображений безопасности. В противном случае злоумышленник мог бы поместить вредоносный исполняемый сценарий, например с именем `ls`, в ваш текущий каталог. Тогда при запуске `ls` вместо этой команды выполнялся бы вредоносный сценарий.

```
$ sudo touch /usr/local/avocado      Создайте файл как root
[sudo] password for smith: password here
$ ls -l /usr/local/avocado           Посмотрите атрибуты файла
-rw-r--r-- 1 root root 0 Sep 10 17:16 avocado
$ sudo rm /usr/local/avocado         Удалите файл как root
```

sudo может запомнить (кэшировать) ваш пароль на некоторое время, в зависимости от того, как она настроена. Поэтому пароль может не запрашиваться каждый раз.

## Дополнительная литература

Чтобы узнать больше об основах использования Linux, прочитайте мою предыдущую книгу *Linux Pocket Guide*, которая вышла в издательстве O'Reilly, или поищите онлайн-руководства (<https://ubuntu.com/tutorials/command-line-for-beginners>).

## Если вы используете не `bash`

В этой книге предполагается, что вы используете оболочку `bash`, но если это не так, таблица Б.1 поможет адаптировать примеры книги для других оболочек. Символ галочки ✓ указывает на совместимость — данная функция достаточно похожа на `bash`, поэтому примеры в книге должны работать правильно. Однако в других ситуациях поведение такой функции может отличаться от функции `bash`. Внимательно прочитайте все сноски.



Независимо от используемой оболочки входа в систему, сценарии, начинающиеся с `#!/bin/bash`, обрабатываются `bash`.

Если хотите поэкспериментировать с другой оболочкой, установленной в вашей системе, просто запустите ее, назвав по имени (например, `ksh`), а когда захотите закончить, нажмите `Ctrl-D`. Чтобы изменить оболочку входа в систему, прочтите `man chsh`.

**Таблица Б.1.** Функции `bash`, поддерживаемые другими оболочками

Функция <code>bash</code>	<code>dash</code>	<code>fish</code>	<code>ksh</code>	<code>tcsh</code>	<code>zsh</code>
Alias	✓	✓, но <code>alias</code> <i>ИМЯ</i> не выводит псевдоним	✓	Знак равенства (=) не нужен: <code>alias g grep</code>	✓
Запуск в фоновом режиме (&)	✓	✓	✓	✓	✓
<code>bash -c</code>	<code>dash -c</code>	<code>fish -c</code>	<code>ksh -c</code>	<code>tcsh -c</code>	<code>zsh -c</code>

Функция bash	dash	fish	ksh	tcsh	zsh
bash команда	dash	fish	ksh	tcsh	zsh
Расположение команды, запускающей оболочку (для bash — /bin/bash)	/bin/dash	/bin/fish	/bin/ksh	/bin/tcsh	/bin/zsh
Переменная BASH_SUBSHELL					
Расширение фигурных скобок { }	Используйте seq	Формат только вида {a,b,c}, а не {a..c}	✓	Используйте seq	✓
cd -	✓	✓	✓	✓	✓
cd	✓	✓	✓	✓	✓
Переменная CDPATH	✓	set CDPATH значение	✓	set cdpath = (каталог1 каталог2 ...)	✓
Подстановка команд с помощью \$( )	✓	Используйте ( )	✓	Используйте обратные кавычки	✓
Подстановка команд с помощью обратных кавычек	✓	Используйте ( )	✓	✓	✓
Редактирование командной строки с помощью клавиш-стрелок		✓	✓ <sup>1</sup>	✓	✓
Редактирование командной строки в стиле Emacs		✓	✓ <sup>1</sup>	✓	✓
Редактирование командной строки в стиле Vim с помощью set -o vi			✓	Запустите bindkey -v	✓
complete		Другой синтаксис <sup>2</sup>	Другой синтаксис <sup>2</sup>	Другой синтаксис <sup>2</sup>	compdef <sup>2</sup>

Функция bash	dash	fish	ksh	tcsh	zsh
Условные списки с использованием <code>  </code> и <code>&amp;&amp;</code>	✓	✓	✓	✓	✓
Файлы конфигурации в каталоге \$HOME (см. документацию для подробностей)	<i>.profile</i>	<i>.config/fish/config.fish</i>	<i>.profile, .kshrc</i>	<i>.cshrc</i>	<i>.zprofile, .zshrc, .zlogin, .zlogout</i>
Структуры управления: <code>for</code> , <code>if</code> и т. д.	✓	Другой синтаксис	✓	Другой синтаксис	✓
<code>dirs</code>		✓		✓	✓
<code>echo</code>	✓	✓	✓	✓	✓
Экранирование псевдонима с помощью <code>\</code>	✓		✓	✓	✓
Экранирование с помощью <code>\</code>	✓	✓	✓	✓	✓
<code>exec</code>	✓	✓	✓	✓	✓
Код возврата с помощью <code>\$?</code>	✓	<code>\$status</code>	✓	✓	✓
<code>export</code>	✓	<code>set -x имя значение</code>	✓	<code>setenv имя значение</code>	✓
Функции	✓ <sup>3</sup>	Другой синтаксис	✓		✓
Переменная HISTCONTROL					См. переменные с именами, начинающимися с HIST_, в документации
Переменная HISTFILE		<code>set fish_history путь</code>	✓	<code>set histfile = путь</code>	✓
Переменная HISTFILESIZE				<code>set savehist = значение</code>	+SAVEHIST

Функция bash	dash	fish	ksh	tcsh	zsh
history		✓, но без нумерации команд	history как псевдоним для hist -l	✓	✓
history -c		history clear	Удалите ~/.sh_history и перезапустите ksh	✓	history -p
Расширение истории с помощью ! и ^				✓	✓
Пошаговый поиск по истории с помощью Ctrl-R		Введите начало команды, затем нажмите стрелку вверх для поиска, стрелку вправо для выбора	✓ <sup>1, 4</sup>	✓ <sup>5</sup>	✓ <sup>6</sup>
history <i>число</i>		history - <i>число</i>	history -N <i>число</i>	✓	history - <i>число</i>
Перемещение по истории команд с помощью клавиш-стрелок		✓	✓ <sup>1</sup>	✓	✓
Перемещение по истории команд в стиле Emacs		✓	✓ <sup>1</sup>	✓	✓
Перемещение по истории команд в стиле Vim с помощью set -o vi			✓	Запустите bindkey -v	✓
Переменная HISTSIZE			✓		✓
Управление заданиями с помощью fg, bg, Ctrl-Z, jobs	✓	✓	✓	✓ <sup>7</sup>	✓

Функция bash	dash	fish	ksh	tcsch	zsh
Сопоставление с шаблоном с помощью *, ?, [ ]	✓	✓	✓	✓	✓
Каналы	✓	✓	✓	✓	✓
popd		✓		✓	✓
Подстановка процесса с помощью < ( )			✓		✓
Переменная PS1	✓	set PS1 значение	✓	set prompt = значение	✓
pushd		✓		✓, но без отрицательных аргументов	✓
Двойные кавычки	✓	✓	✓	✓	✓
Одинарные кавычки	✓	✓	✓	✓	✓
Перенаправление stderr (2>)	✓	✓	✓		✓
Перенаправление stdin (<) и stdout (>, >>)	✓	✓	✓	✓	✓
Перенаправление stdout + stderr (&>)	Добавьте 2>&1 <sup>8</sup>	✓	Добавьте 2>&1 <sup>8</sup>	>&	✓
source или . (точка)	Только точка <sup>9</sup>	✓	✓ <sup>9</sup>		✓ <sup>9</sup>
Подоболочки с помощью ( )	✓		✓	✓	✓
Завершение имен файлов и каталогов с помощью табуляции		✓	✓ <sup>1</sup>	✓	✓
type	✓	✓	type — это псевдоним для whence -v	which	✓



Функция bash	dash	fish	ksh	tcsh	zsh
unalias	✓	functions --erase	✓	✓	✓
Определение переменной с помощью записи <i>имя=значение</i>	✓	set <i>имя</i> <i>значение</i>	✓	set <i>имя</i> = <i>значение</i>	✓
Вычисление переменной с помощью <i>\$name</i>	✓	✓	✓	✓	✓

- 1 Эта функция отключена по умолчанию. Запустите `set -o emacs`, чтобы включить ее. Старые версии `ksh` могут вести себя иначе.
- 2 Пользовательское завершение команды с использованием команды `complete` или аналогичной значительно отличается в разных оболочках. См. документацию к конкретной оболочке.
- 3 Не поддерживается стиль определения функций, в котором они начинаются с ключевого слова `function`.
- 4 Пошаговый поиск по истории команд работает в `ksh` иначе. Нажмите Ctrl-R, введите строку и нажмите Enter, чтобы вызвать самую последнюю команду, содержащую эту строку. Нажмите Ctrl-R и Enter еще раз, чтобы найти следующую совпадающую команду в обратном направлении и т. д. Нажмите Enter, чтобы выполнить ее.
- 5 Чтобы включить пошаговый поиск в истории команд с помощью Ctrl-R в `tcsh`, выполните команду `bindkey ^R i-search-back` (и добавьте ее в файл конфигурации оболочки). Поведение немного отличается от `bash`. См. `man tcsh`.
- 6 В режиме `vi` введите `/`, а затем строку поиска, затем нажмите Enter. Нажмите `n`, чтобы перейти к следующему результату поиска.
- 7 `tcsh` не отслеживает номер задания по умолчанию так удобно, как другие оболочки, поэтому вам может потребоваться указать номер задания, например `%1`, в качестве аргумента для `fg` и `bg`.
- 8 Синтаксис в этой оболочке следующий: команда `> file 2>&1`. Последнее выражение `2>&1` означает «перенаправить `stderr`, который является дескриптором файла 2, на `stdout`, который является дескриптором файла 1».
- 9 Для повторного считывания файлов конфигурации в этой оболочке требуется указывать явный путь к исходному файлу, например `./myfile` для файла в текущем каталоге, иначе оболочка не найдет файл. В качестве альтернативы поместите файл в каталог, указанный в путях поиска файлов текущей оболочки.

## Об авторе

Дэниел Джей Барретт преподает Linux и родственные технологии и пишет о них уже более 30 лет. Автор множества книг издательства O'Reilly, таких как *Linux Pocket Guide*, *Linux Security Cookbook*, *SSH*, *The Secure Shell: The Definitive Guide*, *Macintosh Terminal Pocket Guide* и *MediaWiki*. Помимо этого Дэн в разное время был разработчиком программного обеспечения, рок-певцом, системным администратором, преподавателем в университете, веб-дизайнером и стендапером. Работает в Google. Более подробную информацию ищите по адресу <https://danieljbarrett.com/>.

# Иллюстрация на обложке

На обложке книги изображен балобан (*Falco cherrug*).

Быстрые, мощные и агрессивные, эти пернатые хищники ценятся любителями соколиной охоты на протяжении тысячелетий. Сегодня они являются национальной птицей нескольких стран, например Венгрии, Монголии и Объединенных Арабских Эмиратов.

Длина тела взрослых балобанов часто превышает 50 см, при этом размах крыльев составляет 97–126 см. Самки этого вида значительно крупнее самцов, их вес — 970–1300 г против 730–990 г у последних. Цвет оперения у птиц сильно различается: от темно- до бледно-коричневого или даже белого с коричневыми полосами.

В дикой природе балобаны охотятся в основном на птиц и грызунов, развивая скорость полета до 120–150 км/ч, прежде чем наброситься на добычу. Типичные места обитания включают луга, скалистые районы и галерейные леса, где соколы занимают гнезда, покинутые другими птицами. За исключением самых южных ареалов обитания, балобаны являются перелетными птицами, они ежегодно отправляются на зимовку из Восточной Европы и Центральной Азии в северные части Африки и Южной Азии.

У балобанов нет врагов в дикой природе. Тем не менее их популяция сокращается и балобан считается видом, находящимся под угрозой исчезновения, как и многие другие животные на обложках книг издательства O'Reilly. Все они важны для нашего мира.

Иллюстрацию для обложки выполнила Карен Монтгомери (Karen Montgomery) на основе старинной гравюры из книги 1894 г. Ричарда Лидеккера (Richard Lydekker) *The Royal Natural History*.

*Дэниел Джей Барретт*  
**Linux. Командная строка. Лучшие практики**

*Перевел с английского А. Гаврилов*

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Пителимов</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>Д. Гудилин</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>Л. Галаганова, М. Молчанова</i>
Верстка	<i>Е. Цыцен</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».  
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2023. Наименование: книжная продукция.  
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции  
ОК 034-2014, 58.11.12 — Книги печатные  
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 06.04.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.