**7. Взгляд на мир глазами командной оболочки**

В этой главе мы посмотрим, что происходит в командной строке после нажатия клавиши **ENTER**. И в процессе исследования некоторых интересных и сложных механизмов командной оболочки мы будем пользоваться только одной новой ­командой:

• echo — выводит строку текста.

**Подстановка**

Каждый раз, когда вы вводите команду и нажимаете **ENTER**, bash выполняет несколько операций с текстом, прежде чем выполнит вашу команду. Мы уже видели пару примеров, где простая последовательность символов, например \*, может много значить для командной оболочки. Процесс, который происходит при этом, называется *подстановкой* (expansion). То есть вы вводите что-то, и это что-то замещается чем-то другим, прежде чем командная оболочка продолжит обработку. Чтобы показать, что все это значит, возьмем для примера команду echo — встроенную команду, выполняющую очень простую операцию: она выводит свои текстовые аргументы в стандартный поток вывода.

[me@linuxbox ~]$ **echo this is a test**

this is a test

Все очень просто. echo выведет любой свой аргумент. Давайте попробуем другой пример:

[me@linuxbox ~]$ **echo \***

Desktop Documents ls-output.txt Music Pictures Public Templates Videos

Что это? Почему echo не вывела символ \*? Как вы помните из опытов с групповыми символами, символ \* означает «последовательность любых символов в имени файла», правда, в том обсуждении не рассказывалось, как командная оболочка делает это. На самом деле все просто: перед тем, как выполнить команду echo, оболочка замещает символ \* чем-то другим (в данном случае именами файлов в текущем рабочем каталоге). После нажатия клавиши **ENTER** командная оболочка автоматически производит подстановку любых условных символов в командной строке, прежде чем выполнить ее, поэтому команда echo не увидела \* — она получила уже готовый результат подстановки. Теперь вы понимаете, что в действительности echo действует в точности с нашими ожиданиями?

**Подстановка путей**

Механизм работы групповых символов называется *подстановкой пути* (pathname expansion). Если вернуться к некоторым приемам, продемонстрированным в предыдущих главах, мы увидим, что в действительности они основаны на подстановке. Допустим, содержимое домашнего каталога выглядит вот так:

[me@linuxbox ~]$ **ls**

Desktop    ls-output.txt  Pictures  Templates

Documents  Music          Public    Videos

Мы могли бы выполнить следующую подстановку:

[me@linuxbox ~]$ **echo D\***

Desktop Documents

или

[me@linuxbox ~]$ **echo \*s**

Documents Pictures Templates Videos

или даже

[me@linuxbox ~]$ **echo [[:upper:]]\***

Desktop Documents Music Pictures Public Templates Videos

И заглянуть за пределы домашнего каталога:

[me@linuxbox ~]$ **echo /usr/\*/share**

/usr/kerberos/share /usr/local/share

**Подстановка тильды**

Как вы помните из вводного обсуждения команды cd, символ тильды (~) имеет специальное значение. Если он используется в начале слова, то замещается именем домашнего каталога указанного пользователя или, если пользователь не указан, именем домашнего каталога текущего пользователя:

[me@linuxbox ~]$ **echo ~**

/home/me

Если в системе существует учетная запись пользователя *foo*, тогда

[me@linuxbox ~]$ **echo ~foo**

/home/foo

**подстановка пути для скрытых файлов**

Как мы знаем, файлы с именами, начинающимися с точки, считаются скрытыми. Механизм подстановки пути также учитывает это. Подстановка, такая как

echo \*

не покажет скрытые файлы.

На первый взгляд кажется, что можно было бы включить скрытые файлы в подстановку, добавив в начало шаблона символ точки, например:

echo .\*

Да, такой подход даст желаемое. Однако, если внимательно исследовать результаты, можно заметить, что в них также присутствуют имена . (точка) и .. (две точки). Так как эти имена соответствуют текущему рабочему каталогу и родительскому каталогу, применение такого шаблона может привести к неправильным результатам. Убедимся в этом с помощью команды

ls -d .\* | less

Чтобы обеспечить правильную подстановку пути в такой ситуации, следует использовать специализированный шаблон. Следующий шаблон действует правильно:

ls -d .[!.]?\*

Этот шаблон замещается именами файлов, начинающимися с точки, за которой следует хотя бы один символ, кроме точки, за которым в свою очередь может следовать любое количество других символов.

**Подстановка результатов арифметических выражений**

Командная оболочка поддерживает также подстановку результатов арифметических выражений. Это позволяет использовать командную строку как калькулятор:

[me@linuxbox ~]$ **echo $((2 + 2))**

4

Для подстановки арифметических выражений используется следующий формат:

$((*выражение*))

где *выражение* — это арифметическое выражение, состоящее из значений и арифметических операторов.

Механизм подстановки арифметических выражений позволяет использовать только целые числа (невещественные), зато поддерживает множество арифметических операций. В табл. 7.1 перечислены некоторые из поддерживаемых операторов.

**Таблица 7.1. Арифметические операторы**

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| + | Сложение |
| - | Вычитание |
| \* | Умножение |
| / | Деление (но помните: из-за того, что подстановка поддерживает только целочисленную арифметику, результатом будет целое число) |
| % | Деление по модулю или остаток от деления |
| \*\* | Возведение в степень |

Пробелы в арифметических выражениях не играют роли, а выражения могут содержать вложенные выражения. Например, умножение 52 на 3:

[me@linuxbox ~]$ **echo $(($((5\*\*2)) \* 3))**

75

Для группировки подвыражений допускается использование одиночных круглых скобок. С помощью этого приема выражение, приведенное выше, можно переписать, как показано ниже, и получить тот же результат, но при этом будет использоваться одна операция подстановки вместо двух:

[me@linuxbox ~]$ **echo $(((5\*\*2) \* 3))**

75

Следующий пример демонстрирует использование операторов деления и получения остатка. Обратите внимание, как действует целочисленное деление:

[me@linuxbox ~]$ **echo Пять разделить на два будет $((5/2))**

Пять разделить на два будет 2

[me@linuxbox ~]$ **echo и $((5%2)) в остатке.**

и 1 в остатке.

Подстановка результатов арифметических выражений подробнее будет рассматриваться в главе 34.

**Подстановка фигурных скобок**

Самым малопонятным, пожалуй, выглядит результат подстановки фигурных скобок. С помощью этого механизма из одного шаблона, содержащего фигурные скобки, создается множество текстовых строк. Например:

[me@linuxbox ~]$ **echo Впереди-{A,B,C}-позади**

Впереди-A-позади Впереди-B-позади Впереди-C-позади

Шаблоны с фигурными скобками могут содержать начальную часть, которая называется *преамбулой*, и заключительную часть, которая называется *эпилогом*. Внутри фигурных скобок находится список строк, разделенных запятыми, или диапазон целых чисел или одиночных символов. Использование пробелов внутри фигурных скобок не допускается. Ниже приводится пример с использованием диапазона целых чисел:

[me@linuxbox ~]$ **echo Число\_{1..5}**

Число\_1 Число\_2 Число\_3 Число\_4 Число\_5

В следующем примере используется диапазон символов в обратном порядке:

[me@linuxbox ~]$ **echo {Z..A}**

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

Допускается вложение фигурных скобок:

[me@linuxbox ~]$ **echo a{A{1,2},B{3,4}}b**

aA1b aA2b aB3b aB4b

Какую пользу можно извлечь из этого? Такая возможность может пригодиться для формирования списков файлов или каталогов, которые требуется создать. Например, фотограф, имеющий огромную коллекцию фотографий и желающий организовать ее по годам и месяцам, мог бы начать с создания группы каталогов с именами, состоящими из номера года и месяца. Благодаря этому имена каталогов будут отсортированы в хронологическом порядке. Можно было бы ввести полный список каталогов, но это обременительно и чревато ошибками. Вместо этого выполним следующую команду:

[me@linuxbox ~]$ **mkdir Pics**

[me@linuxbox ~]$ **cd Pics**

[me@linuxbox Pics]$ **mkdir {2009..2011}-0{1..9} {2009..2011}-{10..12}**

[me@linuxbox Pics]$ **ls**

2009-01 2009-07 2010-01 2010-07 2011-01 2011-07

2009-02 2009-08 2010-02 2010-08 2011-02 2011-08

2009-03 2009-09 2010-03 2010-09 2011-03 2011-09

2009-04 2009-10 2010-04 2010-10 2011-04 2011-10

2009-05 2009-11 2010-05 2010-11 2011-05 2011-11

2009-06 2009-12 2010-06 2010-12 2011-06 2011-12

Однако!

**Подстановка параметров**

В этой главе мы лишь кратко коснемся подстановки параметров, а детально обсудим ее позже. Эта возможность полезнее в сценариях на языке командной оболочки, чем непосредственно в командной строке. Многие из ее возможностей имеют отношение к способности системы хранить маленькие фрагменты данных и присваивать этим фрагментам имена. Многие такие фрагменты, правильнее их называть *переменными*, уже существуют и доступны для исследования. Например, переменная с именем USER хранит ваше имя пользователя. Подстановка параметра и получение содержимого переменной USER выполняется следующим образом:

[me@linuxbox ~]$ **echo $USER**

me

Чтобы увидеть список доступных переменных, выполните следующую команду:

[me@linuxbox ~]$ **printenv | less**

Возможно, вы обратили внимание, что если в других вариантах подстановки допустить ошибку в шаблоне, подстановка не будет выполнена и команда echo просто выведет ошибочный шаблон. В случае с подстановкой параметров все иначе: если ошибиться в имени переменной, подстановка все равно будет выполнена, но результатом будет пустая строка:

[me@linuxbox ~]$ **echo $SUER**

[me@linuxbox ~]$

**Подстановка команд**

Подстановка команд позволяет использовать поток вывода команд в качестве аргументов других команд:

[me@linuxbox ~]$ **echo $(ls)**

Desktop Documents ls-output.txt Music Pictures Public Templates Videos

Один из моих любимых вариантов выглядит так:

[me@linuxbox ~]$ **ls -l $(which cp)**

-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp

Здесь результат команды which cp передается как аргумент команде ls, благодаря чему мы получаем информацию о программе cp, не зная полного пути к ней. Подстановка команд не ограничивается такими простыми командами. Можно использовать целые конвейеры (здесь показана только часть вывода):

[me@linuxbox ~]$ **file $(ls /usr/bin/\* | grep zip)**

/usr/bin/bunzip2:      symbolic link to `bzip2'

/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV ), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped

/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped

/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped

/usr/bin/gpg-zip:      Bourne shell script text executable

/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'

/usr/bin/gzip:         symbolic link to `../../bin/gzip'

/usr/bin/mzip:         symbolic link to `mtools'

В этом примере результаты конвейера превратились в список аргументов команды file.

Механизм подстановки команд имеет альтернативный синтаксис, унаследованный от более старых командных оболочек, который также поддерживается в bash. В нем вместо знака доллара и круглых скобок используются обратные апострофы:

[me@linuxbox ~]$ **ls -l `which cp`**

-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp

**Экранирование**

Теперь, после знакомства с множеством способов подстановки, поддерживаемых командной оболочкой, можно начинать учиться управлять ими. Например, взгляните на эту команду:

[me@linuxbox ~]$ **echo this is a     test**

this is a test

Или на эту:

[me@linuxbox ~]$ **echo Итого $100.00**

Итого 00.00

В первом примере механизм *разбиения на слова* удалил дополнительные пробелы из списка аргументов команды echo. Во втором — механизм подстановки параметров подставил пустую строку вместо $1, потому что не нашел такую переменную. Командная оболочка предоставляет механизм, который называется *экранированием* (quoting), для выборочного подавления нежелательной подстановки.

**Двойные кавычки**

Первый тип экранирования, который мы рассмотрим, — *двойные кавычки*. Если заключить текст в двойные кавычки, все специальные символы потеряют свое специальное значение и будут интерпретироваться как обычные символы. Исключение составляют: $ (знак доллара), \ (обратный слеш) и ` (обратный апостроф). То есть разбиение на слова, подстановка путей, подстановка тильды и подстановка фигурных скобок выполняться не будут, но подстановка параметров, подстановка значений арифметических выражений и подстановка команд все еще будут выполняться. Благодаря двойным кавычкам мы сможем обрабатывать имена файлов с пробелами. Представьте, что мы по ошибке создали файл с именем *Два слова.txt*. Если попытаться использовать это имя в командной строке, механизм разбиения слов будет интерпретировать его как два отдельных аргумента:

[me@linuxbox ~]$ **ls -l Два слова.txt**

ls: невозможно получить доступ к 'Два': Нет такого файла или каталога

ls: невозможно получить доступ к 'слова.txt': Нет такого файла или каталога

Добавив двойные кавычки, можно запретить разбиение слов и получить желаемый результат; кроме того, с помощью двойных кавычек мы исправим ошибку:

[me@linuxbox ~]$ **ls -l "Два слова.txt"**

-rw-rw-r-- 1 me me 18 2012-02-20 13:03 Два слова.txt

[me@linuxbox ~]$ **mv "Два слова.txt" Два\_слова.txt**

Вот так! Теперь не нужно вводить эти противные двойные кавычки.

Запомните: подстановка параметров, подстановка значений арифметических выражений и подстановка команд все еще выполняются в двойных кавычках:

[me@linuxbox ~]$ **echo "$USER $((2+2)) $(cal)"**

me 4   February 2012

Su Mo Tu We Th Fr Sa

          1  2  3  4

5  6  7  8  9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29

Давайте отвлечемся и посмотрим, какой эффект оказывают двойные кавычки на подстановку команд. Сначала рассмотрим действие механизма разбиения на слова. В одном из примеров, приведенных выше, мы видели, как механизм разбиения на слова удаляет дополнительные пробелы из текста:

[me@linuxbox ~]$ **echo this is a     test**

this is a test

По умолчанию этот механизм находит пробелы, символы табуляции и символы перевода строки и интерпретирует их как разделители слов. То есть вне кавычек упомянутые символы не считаются частью текста. Они являются лишь *разделителями*. Поскольку они делят слова на аргументы, получается, что в нашем примере командная строка состоит из команды и четырех аргументов. Однако если добавить двойные кавычки, разбиение на слова выполняться не будет и внутренние пробелы не будут считаться разделителями — они станут частью аргумента:

[me@linuxbox ~]$ **echo "this is a     test"**

thisis a     test

После добавления двойных кавычек командная строка будет состоять из команды и одного аргумента.

Тот факт, что символы перевода строки интерпретируются механизмом разбиения на слова как разделители, вызывает интересный и трудноуловимый эффект при подстановке команд. Взгляните:

[me@linuxbox ~]$ **echo $(cal)**

February 2012 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

[me@linuxbox ~]$ **echo "$(cal)"**

   February 2012

Su Mo Tu We Th Fr Sa

          1  2  3  4

5  6  7  8  9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29

В первом случае подстановка команд без кавычек привела к созданию командной строки с 38 аргументами, а во втором случае получилась командная строка с одним аргументом, включающим внутренние пробелы и символы перевода строки.

**Одиночные кавычки**

Если вам требуется подавить *все* подстановки, используйте одиночные кавычки. Ниже для сравнения приводятся результаты неэкранированной команды и коман­ды, экранированной двойными и одиночными кавычками:

[me@linuxbox ~]$ **echo text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER**

text /home/me/ls-output.txt a b foo 4 me

[me@linuxbox ~]$ **echo "text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER"**

text ~/\*.txt {a,b} foo 4 me

[me@linuxbox ~]$ **echo 'text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER'**

text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER

Как видите, каждый следующий уровень экранирования все больше и больше подавляет подстановку.

**Экранирование символов**

Иногда бывает необходимо экранировать только один символ. Для этого достаточно добавить перед символом обратный слеш, который в данном случае называется *экранирующим символом* (escape character). Часто этот прием используется в двойных кавычках, чтобы выборочно предотвратить подстановку.

[me@linuxbox ~]$ echo "Баланс счета пользователя $USER: \$5.00"

Баланс счета пользователя me: $5.00

Экранирование символов также широко применяется для подавления специального значения символов в именах файлов. Например, в именах файлов допускается использование символов, которые имеют специальное значение для командной оболочки. К их числу относятся: $, !, &, (пробел) и др. Чтобы включить специальный символ в имя файла, его достаточно экранировать, как показано ниже:

[me@linuxbox ~]$ **mv bad\&filename good\_filename**

Чтобы включить сам экранирующий символ, его также нужно экранировать, введя \\. Имейте в виду, что внутри одиночных кавычек обратный слеш теряет свое специальное значение и интерпретируется как обычный символ.

**Заключительное замечание**

По мере накопления опыта использования командной оболочки мы все чаще будем использовать возможности подстановки и экранирования, поэтому важно хорошо понимать, как они работают. Фактически можно смело утверждать, что эти два механизма являются наиболее важными для изучения аспектами командной оболочки. Без надлежащего понимания того, как действует подстановка, командная оболочка будет оставаться источником непонимания и домыслов, при этом многие ее возможности останутся неиспользованными.

**управляющие последовательности**

Обратный слеш используется не только в роли экранирующего символа, но и как часть специальных символов, которые называют *управляющими кодами* (control codes). Первые 32 символа в схеме кодирования ASCII использовались для передачи различных команд в устройствах, таких как телетайп. Некоторые из этих кодов хорошо знакомы вам (табуляция, забой, перевод строки и возврат каретки), тогда как другие — нет (пустой символ, конец передачи и подтверждение), как показано в табл. 7.2.

**Таблица 7.2. Управляющие последовательности**

|  |  |
| --- | --- |
| **Управляющая последовательность** | **Значение** |
| \a | Звонок («предупреждение» — заставляет компьютер подать звуковой сигнал) |
| \b | Забой (backspace) |
| \n | Новая строка (в Unix-подобных системах этот символ выполняет перевод строки) |
| \r | Возврат каретки |
| \t | Табуляция |

В этой таблице перечислены некоторые наиболее известные управляющие последовательности. Идея использования обратного слеша зародилась в языке программирования C и была заимствована многими другими языками, включая язык командной оболочки. Параметр -e команды echo включает интерпретацию управляющих последовательностей. Их можно также заключать в конструкцию $' '. Ниже демонстрируется использование команды sleep, простой программы, которая всего лишь ждет указанное число секунд и завершается, для создания элементарного таймера.

sleep 10; echo -e "Time's up\a"

То же самое можно выразить так:

sleep 10; echo "Time's up" $'\a'