Дата последней сборки документа:
22 мая 2020 г. в 01:15

Практика использования и наиболее полезные конструкции командной оболочки bash

Содержание

1. Сценарии командной оболочки bash

1.1. Особенности обработки выражений, заключенных в кавычки

Строка в форме $'...' умеет обрабатывать управляющие последовательности

```
$ echo $'python\nfortran'
# выведет две строки, разбитые символом перевода строки
# python
# fortran
```

Строка в форме "..." умеет корректно обрабатывать подстановку значений переменных, арифметических выражений и подстановку команд

```
$ EMAIL = 'leor.finkelberg@yandex.ru'
$ echo $EMAIL  # leor.finkelberg@yandex.ru
$ echo "My email: '$EMAIL'"  # My email:'leor.finkelberg@yandex.ru'
$ echo "In this catalogy `ls -l | wc -l` files"  # In this catalogy 10 files
$ echo "$INT % 2 -> $(($INT % 2))"  # 8 % 2 -> 0
$ echo "result = <<`find . -name 'cheat*.tex' | xargs grep -inE 'tab.*ofcont*'`>>"
                                                    # result = <<21:\tableofcontents>>
```

1.2. Переменные, встроенные в оболочку

1.3. Формат файла сценария

Для того чтобы успешно создать и запустить сценарий командной оболочки требуется:

1. Написать сценарий,
2. Сделать сценарий исполняемым,
3. Поместить сценарий в каталог, где командная оболочка сможет найти его.

Простой пример в качестве иллюстрации

bash_test.sh

```
#!/bin/bash
echo "This is number of command line args: $#"
```

Сочетание символов `#!` – это специальная конструкция, которая называется *shebang* (произносится как «ше-бенг») и сообщает системе имя интерпретатора, который должен использоваться для выполнения следующего за ним текста сценария. Каждый сценарий командной оболочки должен включать это определение в первой строке.

Команда «точка» (`.`) является синонимом `source`, встроенной команды, которая читает указанный файл и интерпретирует его как ввод с клавиатуры [2, стр. 349].

1.4. Разрешение на выполнение

Теперь нужно сделать этот файл исполняемым

```
$ chmod +x bash_test.sh
$ ls -l bash_test.sh
# выведет
# -rwxr-xr-x 1 ADM 197121 31 май 19 02:55 bash_test.sh*
```

1.5. Местоположение файла сценария

Теперь можно вызывать этот скрипт

```
$ ./bash_test.sh 10 20
# выведет
# This is number of command line args: 2
```

Но чтобы вызвать сценарий, необходимо добавить явный путь перед его именем. Для того чтобы можно было вызывать этот сценарий из любой точки системы, следует добавить его в переменную окружения `PATH`. Как известно, система просматривает каталоги по списку всякий раз, когда требуется найти исполняемую программу, если путь к ней не указан явно. Список каталогов храниться в как раз в переменной окружения `PATH`.

Она содержит список каталогов, перечисленных через двоеточие

```
$ echo $PATH
# выведет
# /c/Users/ADM/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:/usr/bin:\
# /c/Users/ADM/bin:/c/Program Files (x86)/Common Files/Oracle/Java/javapath:\
# /c/Program Files (x86)/Intel/iCLS Client:/c/Program Files/Intel/iCLS Client ... etc
```

К слову, для более удобного просмотра вывода (пути разделяются символом «:»), можно воспользоваться следущей процедурой с привлечением редактора `Vim`:

- Перенаправляем стандартный поток вывода в файл `paths.txt`:
  `echo $PATH > paths.txt`
- Открываем файл редактором `Vim`
  `vim paths.txt`
- Сочетанием клавиш `qa`[1] открываем журнал записи пользовательских сценариев (внизу экрана появится строка «`запись @a`»),
- Теперь можно выполнить нужную нам последовательность команд для первого символа «:». Например `f:xi<CR><Esc>q`, что означает следующее: найти слева направо символ «:», удалить символ, который будет находится под курсором (т.е. символ «:»), перейти в режим ВСТАВКИ (`i`), нажать клавишу `Enter` для перехода на следующую строку в режиме ВСТАВКИ, нажать клавишу `Esc` для выхода из режима ВСТАВКИ и, наконец, закончить запись (`q`)[2],
- Записав сценарий, можно воспользоваться командой `normal`[3]
  `:%normal! 100@a`

Результат будет выглядеть так

```
/c/Users/ADM/bin
/mingw64/bin
/usr/local/bin
/usr/bin
/bin
/mingw64/bin
```

В большинстве `Unix`-подобных операционных систем в переменную `PATH` включается каталог `bin` в *домашнем каталоге пользователя*, чтобы дать пользователям выполнять собственные программы.

То есть если создать каталог `~/bin` и поместить сценарий в него, то наш сценарий можно будет запускать из любой точки системы

```
$ mkdir ~/bin
$ mv bash_test.sh ~/bin
```

Если каталог отсутствует в переменной `PATH`, его легко туда добавить, включив следующую строку в файл `~/.bash_profile`

```
export PATH=~/bin:${PATH}
```

Команда `export` устанавливает *переменную окружения*, или другими словами экспортирует переменную, делая ее переменной окружения. Здесь «:» это просто символ-разделитель, а конструкция `${PATH}` разворачивается в список путей, разделенных символом «:».

[1] Начать запись в именованный регистр `"a`

[2] Проверить содержимое регистра можно так `:reg a`

[3] Здесь задается произвольное большое число, указывающее число повторений команды. Еще очень важный момент заключается в том, что эта команда выполняется параллельно, а не последовательно

1.6. Выбор местоположения для сценариев

Каталог `~/bin` хорошо подходит для сценария, если этот сценарий предназначен для *личного* использования. Сценарии, которые должны быть доступны *всем* пользователям в системе, лучше размещать в традиционном местоположении – в каталоге `/usr/local/bin`.

В большинстве случаев программное обеспечение созданное в *локальной* системе, будь то сценарий или скомпилированные программы, следует помещать в иерархию каталогов `/usr/local`, а не `/bin` или `/usr/bin`. Последние два каталога, как определено стандартом иерархии файловой системы `Linux`, предназначены только для файлов, поставляемых создателями дистрибутива `Linux`.

1.7. Присваивание значений переменным и константам

Командная оболочка не заботиться о типах значений, присваиваемых переменным. Она все значения интерпретирует как *строки*. Между именем переменной и оператором присваивания не должно быть пробелов. Значение может состоять из чего угодно, что можно развернуть в строку

```
a=z                     # Присвоить переменной a строку "z"
b="a string"            # Внутренние пробелы должны находиться в кавычках
c="a string and $b"     # При присваивании допускается выполнять подстановку,
                        # например, значений других переменных
d=$(ls -l foo.txt)      # Результат выполнения команды
e=$((5*7))              # Подстановка результата арифметического выражения
f="\t\ta string\n"      # Экранирование последовательности, такие как
                        # символы табуляции и перевода строки
```

При использовании подстановки имена переменных можно заключить в *необязательные фигурные скобки* `{}`. Это пригодится в том случае, когда имя переменной становится неоднозначным в окружающем контексте.

Например

```
$ filename="myfile"
$ touch $filename
$ mv $filename ${filename}1  # переименовывает файл myfile -> myfile1
```

Добавив фигурные скобки, мы гарантировали, что командная оболочка не будет интерпретировать последний символ `1` как часть имени новой (и пустой) переменной.

Существует еще один метод вывода текста, который называется *встроенным документом* или *встроенным сценарием*. Встроенный документ – это дополнительная форма перенаправления ввода/вывода, которая передает текст, встроенный в сценарий, на стандартный ввод команд.

Действует это перенаправление так:

```
команда << индикатор
текст
индикатор
```

Например

```
#!/bin/bash

TITLE="System Information Report Fort $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
```

```
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
    </BODY>
</HTML>
_EOF_
```

К слову, для того чтобы заменить `$name` на `${name}` в сценарии можно воспользоваться такой конструкцией

```
:%s/>\$\(.*\)</>${\1}</c
```

Эта конструкция ищет совпадение с шаблоном во *всех* строках сценария (`%`). Подшаблон `>\$\(.*\)<` совпадает, например, с подстрокой `>$TITLE<`, и при этом `TITLE` попадает в группу №1 (к ней можно обращаться как `\1`). Теперь можно `>$TITLE<` заменить на `>${\1}<`. Эта часть шаблона воспринимается «как есть», т.е. ничего дополнительно экранировать не нужно. Параметр `c` требует подтверждение замены каждый раз, когда находит подстроку отвечающую поисковому шаблону.

На роль индикатора была выбрана строка `_EOF_`, и она отмечает конец встроенного текста. Строка-индикатор должна находиться в отдельной строке, одна, и за ней не должно следовать никаких пробелов.

Если заменить оператор перенаправления `<<` на `<<-`, то командная оболочка будет игнорировать начальные символы табуляции во встроенном документе. Благодаря этому во встроенный документ можно добавить отступы для большей удобочитаемости

```
#!/bin/bash

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
    open $FTP_SERVER
    user anonymous me@linuxbox
    cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
    _EOF_
ls -l $REMOTE_FILE
```

1.8. Функции

Функции имеют две синтаксические формы. Первая выглядит так

```
function fun_name {
    commands
    return
}
```

Вторая форма выгладит так

```
fun_name () {
    commands
    return
}
```

Обе формы эквивалентны и могут использоваться одна вместо другой. Ниже приводится сценарий, демонстрирующий использование функций командной оболочки

```
1  #!/bin/bash
2
3  function funct {
4      echo "Step 2"
5      return
6  }
7
8  echo "Step 1"  # Step 1
9  funct          # Step 2
10 echo "Step 3"  # Step 3
```

Когда командная оболочка читает сценарий, она пропускает строки с 1-ой по 7-ую, так как они содержат только определение функции. Выполнение начинается со строки 8 с команды `echo`. Строка 9 вызывает функцию `funct`, и командная оболочка выполняет функцию как любую другую команду. Управление передается в строку 4, и выполняется вторая команда `echo`.

Команда `return` в этой строке завершает выполнение функции и возвращает управление в строку, следующую за вызовом функции. После этого выполняется заключительная команда `echo`. Функции должны быть определены в сценарии до их вызова.

Имена функций подчиняются тем же правилам, что и имена переменных. Функция должна содержать хотя бы одну команду. Команда `return` (которая является необязательной) помогает удовлетворить это требование.

Примеры использования функций с *локальными переменными*

```
#!/bin/bash

foo=0  # глобальная переменная

funct_1 () {
    local foo  # переменная ‘foo‘ локальная для ‘funct_1‘
    foo=1
    echo "funct_1: foo = $foo"
}
```

Локальные переменные объявляются добавлением слова `local` перед именем переменной. В результате создается переменная, локальная по отношению к функции, в которой она определена. Когда выполнение выйдет за пределы функции, переменная перестанет существовать.

Рассмотрим такую функцию

```
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
        _EOF_
    return
}
```

Она получает информацию о дисковом пространстве с помощью команды `df -h`.

1.9. Ветвление

Инструкция `if` имеет следующий синтаксис

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

В командной оболочке `bash` поддерживается еще один способ ветвления. Операторы `&&` и || действуют подобно логическим операторам в составной команде `[[...]]`. Они имеют следующий синтаксис

```
command1 && command2
```

и

```
command1 || command2
```

В последовательности с оператором `&&` первая команда выполняется всегда, а вторая – только если первая завершилась успешно. В последовательности с оператором || первая команда выполняется всегда, а вторая – только если первая завершилась неудачей.

Например

```
# если каталога ‘temp‘ не существует, то его нужно создать
$ [[ -d temp ]] || mkdir temp
```

1.10. Современная версия команды test

Улучшенная версия команды `test` выглядит так

```
[[ выражение ]]
```

Команда `test` и ее формы (`[...]`, `[[...]]`) возвращают *код завершения*, показывающий, что выражение либо истинно – `true` (0), либо ложно – `false` (не 0). А вот самое `выражение` возвращает истинное (`true`) или ложное (`false`) значение.

Команда `[[...]]` очень похожа на команду `[...]`, но добавляет новое выражение для проверок строк `строка1 =~ регулярное_выражение`.

Например можно проверить отвечает ли заданное число регулярному выражению

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2 )) -eq 0 ]; then
```

```
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

В команде `[[...]]` поддерживаются *классы символов*[4] POSIX. Ключевые слова описывают различные классы символов, такие как алфавитные, управляющие символы и пр. POSIX, например

```
$ EMAIL="leor.finkelberg@yandex.ru"
$ if [[ "$EMAIL" =~ ^[[:alpha:]]+\.[[:alpha:]]+@[[:alpha:]]+\.ru$ ]]; then echo 'OK'; fi
```

Еще одна дополнительная особенность `[[...]]`: оператор `==` поддерживает сопоставление с шаблонами по аналогии с механизмом подстановки путей. Например

```
$ FILE="foo.bar"
$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
>fi
# foo.bar matches pattern 'foo.*'
```

Или можно воспользоваться регулярным выражением

```
$ FILE="foo.bar"
$ if [[ $FILE =~ ^foo\..+$ ]]; then echo 'OK'; fi
# OK
```

В дополнение к составной команде `[[...]]` `bash` поддерживает также составную команду `((...))`, которую удобно использовать для работы с целыми числами.

Команда `((...))` применяется для *проверки истинности арифметических выражений*. Арифметическое выражение считается *истинным*, если его *результат отличается от нуля*.

Пример

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if (( $INT == 0 )); then
        echo "INT is zero."
    ...
    fi
    if (( $INT % 2 )); then
        echo "INT is even."
    else
        echo "INT is odd."
...
```

[4]Класс символов POSIX состоит из ключевых слов, заключенных между `[:` и `:]`. Эти конструкции должны находится в квадратных скобках скобкового выражения. Например, `[[:alpha:]!]` соответствует любому одиночному символу или восклицательному знаку [3, стр. 98]

1.11. Объединение выражений

Команда `[[...]]` поддерживает три логические операции: И (`&&`), ИЛИ (`||`) и НЕ (`!`).

Пример

```
#!/bin/bash

MIN_VAL=1
MAX_VAL=100
INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
        echo "$INT is whithin $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    else "INT is not an integer." >&2
    exit 1
fi
```

Логические блоки можно заключать в круглые скобки

```
...
if [[ ($INT -lt 0) && ($INT -gt 10) ]]; then
    echo ...
```

Еще с `[[...]]` можно сочетать подстановку `$(...)` и вычисления `(...)`

```
...
if [[ ($INT -gt 0) && $(($INT % 2)) -eq 0 ]]; then
    echo ...
```

2. Полезные конструкции оболочки `bash`

Найти в корневом каталоге и всех подкатлогах (`/`), обычные файлы (`-type f`), измененные за последний день (`-mtime -1`), за исключением тех файлов, у которых есть суффикс `.o` (`! -name '*.o'`)

```
find / -type f -mtime -1 ! -name '*.o'
```

Вывод имен файлов и удаление файлов с именами `core` или `junk` из рабочего каталога и всех его подкаталогов (круглые скобки обязательно отделяются пробелами)

```
find . \( -name core -o -name junk \) -print -exec rm {} \;
```

Скопировать все `csv`-файлы из родительской директории (`..`) в текущую (`.`)

```
cp -ip ../*.csv
```

Скопировать файл из родительской директории в текущую директорию

```
cp -ip ../Cheat_sheet_Git/cheat_sheet_git.tex .
```

Скопировать одну директорию в другую

```
cp -rip ../Cheat_sheet_Git/style_packages/ .
```

Переименовать файл

```
mv cheat_sheet_git.tex cheat_sheet_bash.tex
```

Найти все файлы с расширением `*.csv` и выбрать из них те, в которых содержится строка `'state'` (для каждого файла, отвечающего поисковому шаблону, запускается свой процесс)

```
find . -name '*.csv' -exec grep -niE 'state' {} \;
```

Вывести список файлов из текущей директории и всех поддиректорий

```
ls -l *
```

Найти среди файлов с расширением `*.py` те, в именах которых есть подстрока `'spark'` (используется конвейер)

```
ls -l *.py | grep -iE 'spark'
```

Найти файлы с расширением `*.py` и к каждому из них применить команду `grep`, которая будет искать в файле подстроку `'argparse'` без учета регистра, с выводом номера строки, на которой она нашла искомую строку по регулярному выражению `'argparse'` (работает медленно, так как для каждого файла, отвечающего поисковому шаблону запускается свой процесс)

```
find . -maxdepth 1 -name '*.py' -exec grep -inE 'argparse' {} \;
```

Альтернативный вариант с использованием `xargs` (работает значительно быстрее варианта с `-exec`)

```
find . -maxdepth 1 -name '*.py' | xargs grep -inE 'argparse'
```

Найти в файлах с расширением `*.tex` строку `'section'` без учета регистра и вывести три строки контекста

```
find . -name '*.tex' | xargs grep -iE 'section' -3
```

Вывести список пакетов, в именах которых встречается подстрока `'python'` с контекстом `'sql'`

```
conda list | grep -inE 'python.*sql'
```

Получить информацию о доступном метсе на диске

```
df -h
```

Скачать файл с тем же именем, что на удаленном репозитории

```
curl -O http://merionet.ru/yourfile.tar.gz
```

Скачать файл с удаленного репозитория с новым именем и/или путем

```
curl -o newfile.tar.gz http:// merionet.ru /yourfile.tar.gz
```

Возобновить прерванную загрузку с того места, где она остановилась

```
curl -C - -O http://merionet.ru/yourfile.tar.gz
```

Скачать несколько файлов

```
curl -O http://merionet.ru/info.html -O http://wiki.merionet.ru/about.html
```

2.1. Переадресация ввода-вывода

Перенаправить *стандартный поток вывода* данных (дескриптор файла 1) и *стандартный поток вывода ошибок* (дескриптор файла 2), которые возвращает `conda` с захватом всех пакетов, в именах которых встречается подстрока `'python'`, в файл с именем `test_file.log` (временный поток вывода данных `&1`). Если команда вернет ошибку, то сообщение ошибки перепишет содержимое файла `test_file.log`

```
conda list | grep -inE 'python' > test_file.log 2>&1
```

Более короткий вариант рассмотренной выше конструкции

```
conda list | grep -inE 'python' &> test_file.log
```

Присоединить стандартный поток вывода данных и стандартный поток вывода ошибок к содержимому файла. Конструкция `rm df` возвращает сообщение об ошибке «`rm: cannot remove 'df': No such file or directory`», которое можно добавить в файл

```
rm df &>> test_file.txt
```

Найти в текущей директории файлы с расширением `.tex`, в именах которых встречается подстрока «`bash`», перенаправить эти файлы утилите `grep`, которая будет искать в теле файлов строки, в которых встречается подстрока «`vim`», причем стандартный поток вывода ошибок перенаправляется в «черную дыру»

```
find . -name '*bash*.tex' | xargs grep -inE 'vim' 2>/dev/null
```

2.2. Информация об использовании дискового пространства

Вывести размер директорий в Мегабайтах (`M`)

```
du -BM
```

Вывести итоговый размер директории (`c`) в Мегабайтах (`M`)

```
du -cBM
```

Перевести размеры директорий в понятный человеку формат

```
du -h
```

Вывести информацию об указанной директории в дружественном формате

```
du -sh style_packages/
```

Вывести размер папок текущей директории, не погружаясь глубже корневых папок

```
du -h -d 1
```

или так

```
du --max-depth=1 -h
```

Вывести информацию об использовании дискового пространства иерархией каталога с подсчетом общего размера каталога и перенаправлением стандартного потока вывода ошибок в «черную дыру»

```
du -ch -d 1 2>/dev/null
```

3. Вспомогательные конструкции Vim

3.1. Регистры Vim

Регистры Vim – это всего лишь контейнеры для хранения текста. Их можно использовать как своеобразные буферы обмена копируя текст в регистры и вставляя его из регистров, или для записи макросов, сохраняя в них последовательности нажатий на клавиши.

3.2. Макросы

Клавиша q действует как своеобразная кнопка «Запись» и одновременно как кнопкаа «Стоп». Чтобы начать запись последовательности нажатий на клавиши, необходимо ввести q{register}, указав имя регистра, где будет сохранен макрос.

Посмотреть содержимое регистра "a можно с помощью :reg

```
:reg a
```

Команда @{register} служит для выполнения содержимого указанного регистра. То есть, если макрос записывался в регистр "a, то вызывать нужно макрос из того регистра как @a.

Макрос записанный, например в регистр "a, можно выполнить заданное число раз, указав число повторений, например так 25@a.

Если требуется, чтобы макрос выполнялся для каждой строки из заданного диапазона, то следует использовать команду :normal[5]

```
:normal! 25@a
```

3.3. Базовые концептуальные конструкции

Перейти в ВИЗУАЛЬНЫЙ режим, выделить 2 «слова» (фрагмент текста) и скопировать в безымянный регистр ("")

```
v2wy
```

Вставить данные из регистра (именованного, неименованного и др.), не покидая режима вставки, можно с помощью сочетания клавиш <C-r>{registr} (т.е. Ctrl+C и имя регистра). Пусть в *регистре выделенного фрагмента* ("*) хранится какая-нибудь строка. С помощью команды A переходим в режим ВСТАВКИ в конец строки. Теперь строку и регистра можно вставить сочетанием <C-r>. После этого под курсором появится символ ". Клавиша * заменит символ " содержимым регистра "*.

Фрагмент строки можно вставить слева от курсора с помощью команды P или справа от курсора с помощью команды p.

Если же копировалась строка *целиком*, то команда p вставляет строку под текущей, а команда P – над текущей.

Удалить слово, находясь внутри слова (т.е. на любой позиции в пределах слова), можно так

```
diw
```

Аналогично можно удалить слово и перейти в режим ВСТАВКИ, находясь внутри слова

[5] Восклицательный знак для игноирования пользовательских настроек

```
ciw
```

Используя подобную конструкцию, можно удалить текст, заключенный в кавычки, и тут же перейти в режим ВСТАВКИ

```
ci"
```

Захватить текущее слово в регистр `"a`

```
"ayiw
```

Вырезать текущую строку в регистр `"b`

```
"bdd
```

Захватить последовательность символов, заключенную в кавычки, в регистр `"b`

```
"byi'
```

Захватить слово в неименованный регистр (`""`)

```
""yiw
```

Захватить последовательность символов от текущего положения курсора до конца строки в *регистр захвата* (`"0`). Все что копируется с помощью `y` попадает в регистр захвата

```
y$
```

Вставить данные из регистра захвата

```
"0p
```

Захватить в *регистр выделенного фрагмента* (`"*`) несколько слов

```
v3wey
```

Вставить данные из именованного регистра `b`

```
"bp
```

Список литературы

1. Собель М. Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.
2. Шоттс У. Командная строка Linux. Полное руководство. – СПб.: Питер, 2017. – 480 с.
3. Роббинс А., Ханна Э., Лэмб Л. Изучаем редакторы vi и Vim, 7-е издание. – СПб.: Символ-Плюс, 2013. – 512 с.