How To Be A Shell Good Coder

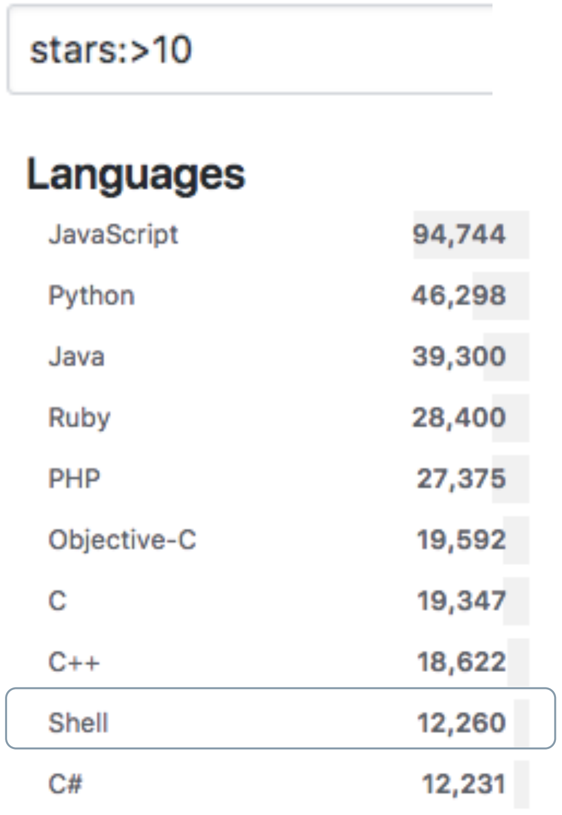
刘卓 liuzhuo1@xiaomi.com

参考材料

- abs
- bash 用户手册
- wwy 的bash编程讲义
- google shell编程规范
- …

Shell是

- 命令行解释器
- 用户和系统调用中间的一层
- 一类编程语言
- 必备技能
- 很多代码仓库中都有shell代码

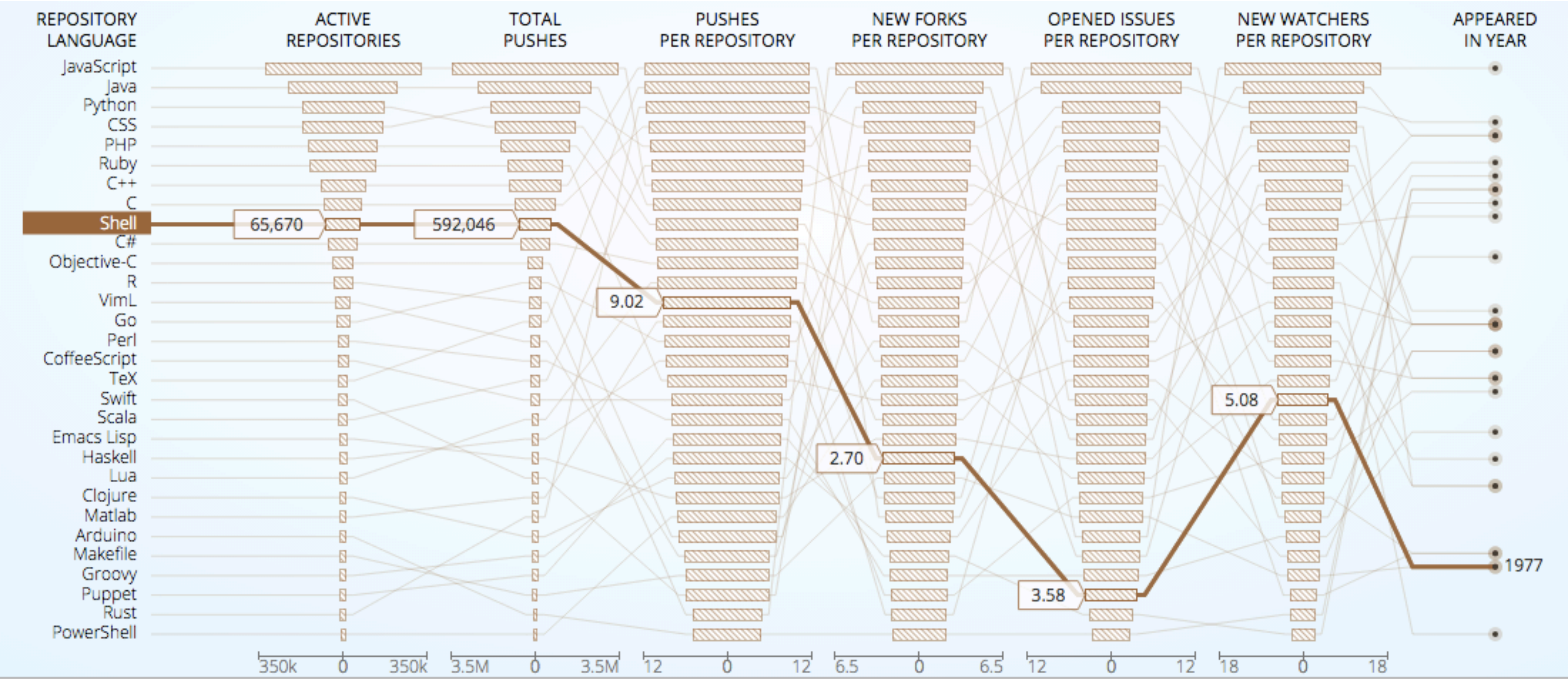


stars:>10

Languages

JavaScript	94,744
Python	46,298
Java	39,300
Ruby	28,400
PHP	27,375
Objective-C	19,592
C	19,347
C++	18,622
Shell	12,260
C#	12,231

Shell是



REPOSITORY LANGUAGE
ACTIVE REPOSITORIES
TOTAL PUSHES
PUSHES PER REPOSITORY
NEW FORKS PER REPOSITORY
OPENED ISSUES PER REPOSITORY
NEW WATCHERS PER REPOSITORY
APPEARED IN YEAR
JavaScript
Java
Python
CSS
PHP
Ruby
C++
C
Shell
C#
Objective-C
R
VimL
Go
Perl
CoffeeScript
TeX
Swift
Scala
Emacs Lisp
Haskell
Lua
Clojure
Matlab
Arduino
Makefile
Groovy
Puppet
Rust
PowerShell
65,670
592,046
9.02
2.70
3.58
5.08
1977
350k
0
350k
3.5M
0
3.5M
12
0
12
6.5
0
6.5
12
0
12
18
0
18

热身

bc 计算器

```
$ echo "1+2" | bc
3

$ echo "1/3" | bc
0

$ echo "scale=4; 1/3" | bc -l
.3333

$ echo "scale=10; 4*a(1)" | bc -l
3.1415926532
```

热身

seq 序列生成

```
$ seq 0 5
0
1
2
3
4
5


$ seq -s '-' 0 5
0-1-2-3-4-5
```

热身

1 + 2 + ... + 99

通常解法

```
for (( i=1; i<=99; i++ )); do
  sum=$(( sum + i ))
done

echo ${sum}
```

Quick解法

```
seq -s '+' 0 99 | bc
```

Shell编程的特点

- Hacker精神（Quick and Dirty），快速上手
- Unix哲学，一个程序只关注并做好一个目标，用文本做接口
- 一切都是字符
- 非单进程运行
- 面向过程
- ……

知识目标

- 成为高效、高质量的 Shell Coder

任务目标

- 不逐条讲解编程规范
- 不多讲Shell基础知识
- 从规范编写代码出发，讲解背后的原理，举一反三
- 介绍编程规范背后遵循的原则
- 以实例和操练来消化知识

目录 Agenda

1.1 Bash 和 POSIX Shell

- #!
  - sha-bang
  - hashbang
  - pound-bang

- 使用bash
  - #!/bin/bash
  - #!/usr/bin/env bash
  - bash script

```
$ cat posix_shell_test.sh
#!/bin/bash

diff <(echo xxx) <(echo yyy)

$ sh posix_shell_test.sh
posix_shell_test.sh: line 4: syntax error near unexpected token `('
posix_shell_test.sh: line 4: `diff <(echo xxx) <(echo yyy)'

$ bash --posix posix_shell_test.sh
posix_shell_test.sh: line 4: syntax error near unexpected token `('
posix_shell_test.sh: line 4: `diff <(echo xxx) <(echo yyy)'

$ bash posix_shell_test.sh
1c1
< xxx
---
> yyy
```

1.2 类型（type）

Aliases: 别名
```
$ type ll
```

Functions: 函数
```
$ type cd
```

Builtins: 内置命令。不fork进程。
```
$ type [
```

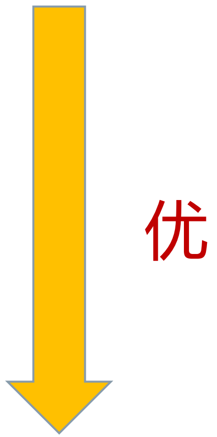
Keywords: 关键字。shell的保留字
```
$ type [[
```

Executables: 外部命令
```
$ type rm
```

Executables

Builtins

Keywords



1.3 进程（process）

进程和子shell

外部命令：awk，grep，ls
子shell: ()
管道: |

```
#!/bin/bash

sleep 20
```

```
#!/bin/bash

# ()中只有1个命令时，不生成subshell
( sleep 10 )


# ()中有2个以上命令时，生成subshell
( sleep 20; echo "ok" )
```

```
# 不生成subshell
echo "something" | sleep 20

# 生成subshell
echo "something" | while true; do
  sleep 1
done

# 同上
echo "something" | {
  sleep 20
}

# 不生成subshell
while true; do
  sleep 1
done
```

1.4 I/O重定向（I/O Redirection）

重定向的种类

1. File Descriptor

```
fd > filename
fd1 >&fd2
&> /dev/null
```

3. Here Documents & String

```
cat <<EOF
something
EOF

sed 's/a/A/g' <<< 'abcdeab'
```

2. Pipe

```
cat *.txt | sort | uniq
```

4. Process Substitution

```
diff <(cat file1) <(cat file2)
```

1.4 I/O重定向（I/O Redirection）

file descriptor

- exec [n]<>file

```
#!/bin/bash


echo 1234567890 > File    # Write string to "File".
exec 3<> File             # Open "File" and assign fd 3 to it.
read -n 4 <&3             # Read only 4 characters.
echo -n . >&3             # Write a decimal point there.
exec 3>&-                 # Close fd 3.
cat File                  # ==> 1234.67890
```

1.4 I/O重定向（I/O Redirection）

Output Redirect

```
dir=/home/not_exist

if cd "${dir}" ; then
  echo "Now in ${dir}."
else
  echo "Can't change to ${dir}."
fi
```

- 仅判断返回值的场景
- 捕获命令的正确输出，不希望被错误输出干扰
- 使用&>/dev/null，不用 >/dev/null 2>&1

```
dir=/home/not_exist

if cd "${dir}" &>/dev/null; then   # ”&>/dev/null" hides message.
  echo "Now in ${dir}."
else
  echo "Can't change to ${dir}."
fi
```

1.4 I/O重定向（I/O Redirection）

进程替换（process substitution）

```
$ echo “something” | grep “o”
```

另一种写法？

```
$ grep “o” <(echo “something”)
```

vs. Pipe

需要2个文件的命令

```
$ diff <(ls dir1) <(ls dir2)
```

```
#!/bin/bash

diff <(pwd) <(echo $0)
echo "$(pwd)" | diff - <(echo $0)

cat $0 | cat - | grep 'pwd' -
```

“-”代表标准输入

1.5 Globs

Globs：匹配filenames

- *   : 0或多个字符
- ?   : 单个字符
- [...] : 括号中的任一字符

Regular Expression： 匹配strings

```
#!/bin/bash

ls *.sh
ls her?.sh
ls [12].jpg
```

```
$ echo ’3.5 * 4’ | bc
14.0

$ touch +
$ ls
+
$ echo 3.5 * 4 | bc
7.5
$ echo 3.5 * 4
3.5 + 4
```

1.5 Globs

Quote

quote可以避免word splitting

```
List="one two three"


for a in $List ; do
  echo "$a"
done
```

```
List="one two three"


for a in “$List” ; do
  echo "$a"
done
```

1.6 Brace expansion

```
$ echo {A..C}{1..3}
A1 A2 A3 B1 B2 B3 C1 C2 C3
```

- 简化字符串生成
- 用于for in循环

```
#!/bin/bash

for i in {0..9}; do
  echo ${i}
done
```

将文件备份，生成.bak文件

```
#!/bin/bash

## backup file
## mv file file.bak
mv file{,.bak}
```

不靠谱的备份

```
12474.txt    17075.txt    17919.txt    22165.txt    29984.txt     backup.sh
15897.txt    17665.txt    20685.txt    25073.txt    769.txt      random_files.sh
```

```
12474.txt.bak  17075.txt.bak  17919.txt.bak  22165.txt.bak  29984.txt.bak  backup.sh
15897.txt.bak  17665.txt.bak  20685.txt.bak  25073.txt.bak  769.txt.bak   random_files.
sh
```

1.6 Brace expansion

彩蛋

0 + 1 +…… + 99

```
$ echo {0..9}{0..9} | sed ‘s/ /+/g’ | bc
```

```
$ seq -s" " 00 99|sed -e 's/ /+/g' -e 's/+$//' | bc
```

```
$ seq -s“+” 00 99| sed ‘s/+$//’ | bc
```

$ Shell不适合做什么

•Resource–intensive tasks, especially where speed is a factor (sorting, hashing, recursion

•Procedures involving heavy–duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use *C++*)

•Cross–platform portability required (use *C* or *Java* instead)

•Complex applications, where structured programming is a necessity (type–checking of variables, function prototypes, etc.)

•Mission–critical applications upon which you are betting the future of the company

•Situations where *security* is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism

•Project consists of subcomponents with interlocking dependencies

•Extensive file operations required (*Bash* is limited to serial file access, and that only in a particularly clumsy and inefficient line–by–line fashion.)

•Need native support for multi–dimensional arrays

•Need data structures, such as linked lists or trees

•Need to generate / manipulate graphics or GUIs

•Need direct access to system hardware or external peripherals

•Need port or socket I/O

•Need to use libraries or interface with legacy code

•Proprietary, closed–source applications (Shell scripts put the source code right out in the open for all the world to see.)

目录 Agenda

2.1 变量

再谈变量安全使用

```
$ song="My song.mp3”
$ rm ${song}
rm: My: No such file or directory
rm: song.mp3: No such file or directory
```

```
$ foo=bar
$ echo "$foos, ${foo}s”
```

Bash解释为：
$ rm My song.mp3

应写为：
$ rm "${song} "

使用${parameter}，避免截断

2.2 参数扩展（Parameter Expansion）

字符串操作

${parameter#pattern}	去头，最短
${parameter##pattern}	去头，最长
${parameter%pattern}	去尾，最短
${parameter%%pattern}	去尾，最长

```
file=/home/zed/scripts/string.sh.bak

echo ${file%.*}
echo ${file%%.*}
echo ${file#*.}
echo ${file##*.}

## 以下的代码等同
echo ${file%/*}
echo $(dirname ${file})
echo ${file##*/}
echo $(basename ${file})
```

问题：如何去掉*.bak文件的.bak后缀？

```
mv ${backup} ${backup%.bak}
```

不靠谱的备份

```
12474.txt    17075.txt    17919.txt    22165.txt    29984.txt    backup.sh
15897.txt    17665.txt    20685.txt    25073.txt    769.txt      random_files.sh
```

```
12474.txt.bak  17075.txt.bak  17919.txt.bak  22165.txt.bak  29984.txt.bak  backup.sh
15897.txt.bak  17665.txt.bak  20685.txt.bak  25073.txt.bak  769.txt.bak    random_files.
sh
```

2.2 参数替换（ Parameter Expansion）

默认值

- :=, :-常用
- :- 使用默认值
- :=使用默认值，并赋值

- 考虑到记忆成本和安全使用，只用

:-

```
#!/bin/bash
# var已定义， 但为空
var=

# 如果var
echo ${var-default}
echo ${var:-default}
echo ${var}

echo ${var=default}
echo ${var:=default}
echo ${var}
```

输出

```
          #空
default
          #空
          #空
default
default
```

```
#!/bin/bash
# var未定义
# var=

# 如果var
echo ${var-default}
echo ${var:-default}
echo ${var}

echo ${var=default}
echo ${var:=default}
echo ${var}
```

输出

```
default
default
          #空
default
default
default
```

2.3 数组（array）

- 易于遍历
- 易于注释
- ${array[0]}
- ${array[@]}

```
#!/bin/bash

# 不易于注释
server_list="jx.server1.jx.baidu.com
tc.server1.tc.baidu.com"

## 更方便注释, 迭代也更方便
servers=(
  jx.server1.jx.baidu.com
  tc.server1.tc.baidu.com
)

for server in ${servers[@]}; do
  echo ${server}
done

echo ${#servers[@]}
echo ${#server[0]}
echo ${servers[0]%.baidu.com}
echo ${servers[0]%%.*}

## 对数组中所有元素都执行字符串截取
echo ${servers[@]%%.*}
```

输出

```
jx.server1.jx.baidu.com
tc.server1.tc.baidu.com



2
23
jx.server1.jx
jx



jx tc
```

$ 替换（Substitution）共有哪几种

Parameter Substitution/Expansion Manipulating and/or expanding variables variable value -> variable value	${}
Command Substitution command output -> string	$()
Process Substitution command output -> file name	<()

目录 Agenda

3.1 条件表达式

[[ ]]

- test等价于[
- 使用[[，忽略其他写法

```
if test -e foo.txt && test -e bar.txt; then
  echo "file: foo.txt and bar.txt exist.”
fi

if [ -e foo.txt ] && [ -e bar.txt ]; then
  echo "file: foo.txt and bar.txt exist.”
fi

if [[ -e foo.txt && -e bar.txt ]]; then
  echo "file: foo.txt and bar.txt exist.”
fi
```

```
$ type test
test is a shell builtin

$ type [
[ is a shell builtin

$ type ]
bash: type: ]: not found

$ type [[
[[ is a shell keyword

$ type ]]
]] is a shell keyword
```

3.1 条件表达式

(( ))

A. 整数运算
(( ))　：　运算
$(( ))　：　捕获运算结果

```
n=3

if (( n > 2 )); then
  echo "${n} > 2"
fi

(( n++ ))
echo "n is : ${n}"

m="$(( n * 6 ))"
echo "m is : ${m}"
```

3.2 && ||

- if else 可以简写为 && ||
- if 可简写为 &&
- if ！ 可简写为 ||

```
$ sl
bash: sl: command not found

$ [[ $? -ne 0 ]] && echo fail
fail

$ sl
bash: sl: command not found

$ [[ $? -eq 0 ]] | | echo fail
fail
```

```
#!/bin/bash

dir="./output"

## if statement
rmdir ${dir} &>/dev/null
if [ -d ${dir} ]; then
  echo "${dir} exist."
else
  echo "${dir} not exist."
  mkdir ${dir}
fi

## && | | statement
## 只有一个if else的语句， 这样写比较简洁
rmdir ${dir} &>/dev/null
[[ -d ${dir} ]] && {
  echo "${dir} exist."
} | | {
  echo "${dir} not exist."
  mkdir ${dir}
}

## 默认要存在的目录, 这样写最简单
rmdir ${dir} &>/dev/null
mkdir -p ${dir}
```

3.3 条件语句的简写

A. 短判断条件，可以简写

```
if grep "^#.*" <(echo "${line}"); then continue ; fi
if [[ -z "${line}" ]] || [[ "${line}" == \#* ]]; then continue; fi
[[ "$TRACE" ]] && set -x
```

B. 短执行语句，可以简写

```
[[ $? -ne 0 ]] && exit 1
[[ ${is_valid} == true ]] && return 0 || return 1
```

3.4 安全的更改环境变量

环境变量

```
cd "${directory}"
cd –
```

?

如果第一条cd语句失败，则当前路径就不符合预期

```
cd "${directory}”  && {
    cd -
}
```

确保cd成功

```
( cd "${directory}" )
```

确保环境变量不影响到当前shell

3.5 循环语句

- while loop常用用于无限循环
- for in循环最好用
- 循环条件的不同形式
  - (( ))
  - [[ ]]
  - :, true
  - { }
  - command

```
#!/bin/bash

n=10

i=0
while (( i<${n} )); do
  echo ${i}
  (( i++ ))
done

i=0
while : ; do
  echo ${i}
  (( i++ ))
  [[ $i == 10 ]] && break
done
```

```
for i in {0..9}; do
  echo ${i}
done
```

```
for (( i=0; i<${n}; i++ )); do
  echo ${i}
done
```

3.5 循环语句

- while loop常用用于无限循环
- for in循环最好用
- 循环条件的不同形式
  - (( ))
  - [[ ]]
  - :, true
  - { }
  - command

```
while true; do
   echo "inifinite loop”
done


while sleep 300; do
   command
done
```

```
until ping -c 1 -w 1 "${host}"; do
   echo "${host} is still unavailable”
done
```

3.5 循环语句

彩蛋

```
Lines=0

cat $0 \
  | while read line ; do
     (( Lines++ ));
    done

echo "Number of lines read = ${Lines}"
```

?

解决方法：消除subshell

```
Lines=0

while read line ; do
  (( Lines++ ));
done < $0

echo "Number of lines read = ${Lines}"
```

fd 0

```
Lines=0

exec 3<> $0
while read line <&3 ; do
  (( Lines++ ));
done
exec 3>&-

echo "Number of lines read = ${Lines}"
```

fd 3

目录 Agenda

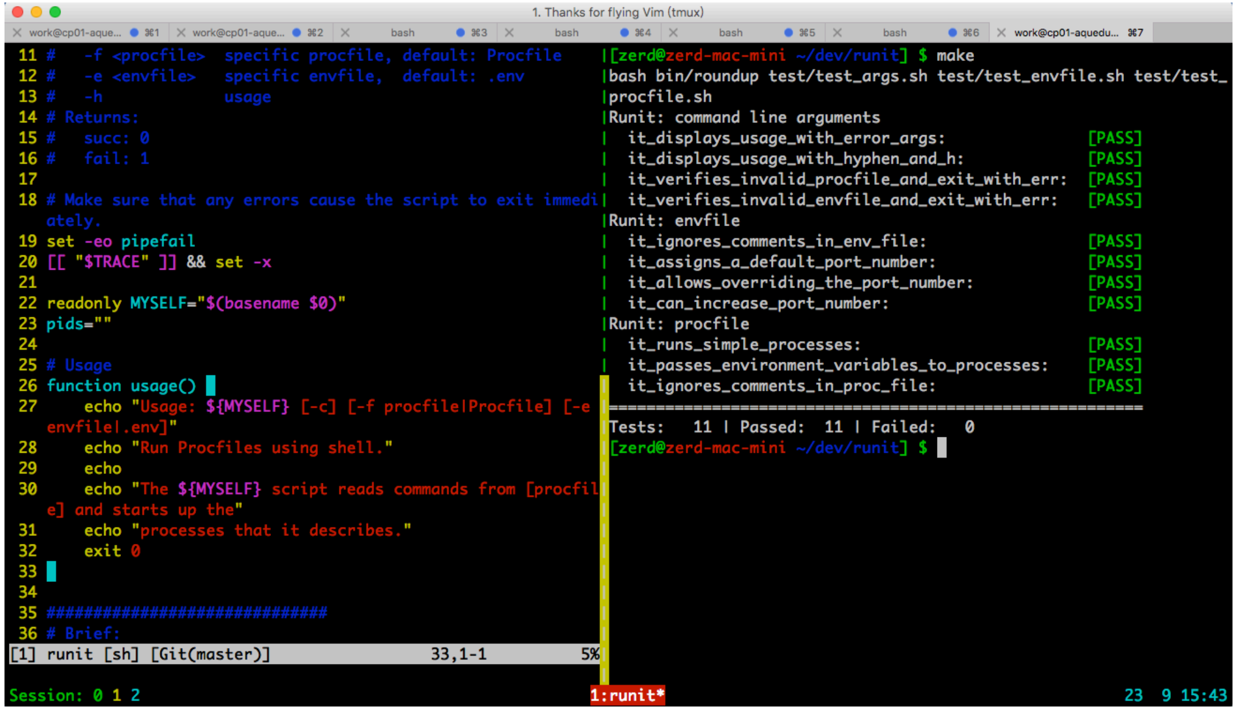
4.1 Terminal

Terminal 提效

更舒服的Ctrl
Caps lock -> Ctrl

光标移动：
Ctrl + A
Ctrl + E

Terminal多窗口管理：Tmux
[Ctrl + A] + |
[Ctrl + A] + -
[Ctrl + A] + ,
[Ctrl + A] + c
[Ctrl + A] + H
[Ctrl + A] + L
[Ctrl + A] + J
[Ctrl + A] + K



```
#   -f <procfile>  specific procfile, default: Procfile
#   -e <envfile>   specific envfile,  default: .env
#   -h             usage
# Returns:
#   succ: 0
#   fail: 1

# Make sure that any errors cause the script to exit immedi
ately.
set -eo pipefail
[[ "$TRACE" ]] && set -x

readonly MYSELF="$(basename $0)"
pids=""

# Usage
function usage()
    echo "Usage: ${MYSELF} [-c] [-f procfile|Procfile] [-e
envfile|.env]"
    echo "Run Procfiles using shell."
    echo
    echo "The ${MYSELF} script reads commands from [procfil
e] and starts up the"
    echo "processes that it describes."
    exit 0


###############################
# Brief:
[1] runit [sh] [Git(master)]                    33,1-1        5%
```

```
[zerd@zerd-mac-mini ~/dev/runit] $ make
bash bin/roundup test/test_args.sh test/test_envfile.sh test/test_
procfile.sh
Runit: command line arguments
  it_displays_usage_with_error_args:                    [PASS]
  it_displays_usage_with_hyphen_and_h:                  [PASS]
  it_verifies_invalid_procfile_and_exit_with_err:       [PASS]
  it_verifies_invalid_envfile_and_exit_with_err:        [PASS]
Runit: envfile
  it_ignores_comments_in_env_file:                      [PASS]
  it_assigns_a_default_port_number:                     [PASS]
  it_allows_overriding_the_port_number:                 [PASS]
  it_can_increase_port_number:                          [PASS]
Runit: procfile
  it_runs_simple_processes:                             [PASS]
  it_passes_environment_variables_to_processes:         [PASS]
  it_ignores_comments_in_proc_file:                     [PASS]
=========================================================
Tests:    11 | Passed:   11 | Failed:    0
[zerd@zerd-mac-mini ~/dev/runit] $
```

```
Session: 0 1 2                           1:runit*                           23  9 15:43
```

4.2 调试

set -x
bash -x script

```
#!/bin/bash

set -x

echo 1
echo 2
echo 3
```

```
#!/bin/bash

function my_debug {
  set -x
  sleep 1
}

trap my_debug DEBUG

echo 1
echo 2
echo 3
```

```
#!/bin/bash

function my_debug {
  set -x
  read -t 2
}

trap my_debug DEBUG

echo 1
echo 2
echo 3
```

单步调试
trap function DEBUG 在每条命令执行前调用function
sleep不能加速，read可以

4.3 安全设置

- set -u 检查变量都被初始化
- set -e 检查命令运行结果
- set -o pipefail 检查管道中的命令运行结果

```
#!/bin/bash


set -ue
#set -o pipefail

var=NotNull
echo $var


false || {
  echo Something false
}

true | false | true || {
  echo xx
}

echo End
```

```
NotNull
Something false
End
```

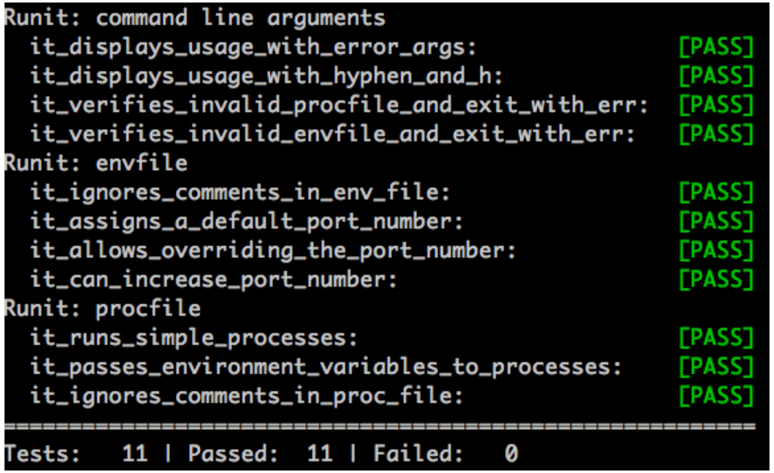
```
#!/bin/bash


set -ue
set -o pipefail

var=NotNull
echo $var


false || {
  echo Something false
}

true | false | true || {
  echo xx
}

echo End
```

```
NotNull
Something false
xx
End
```

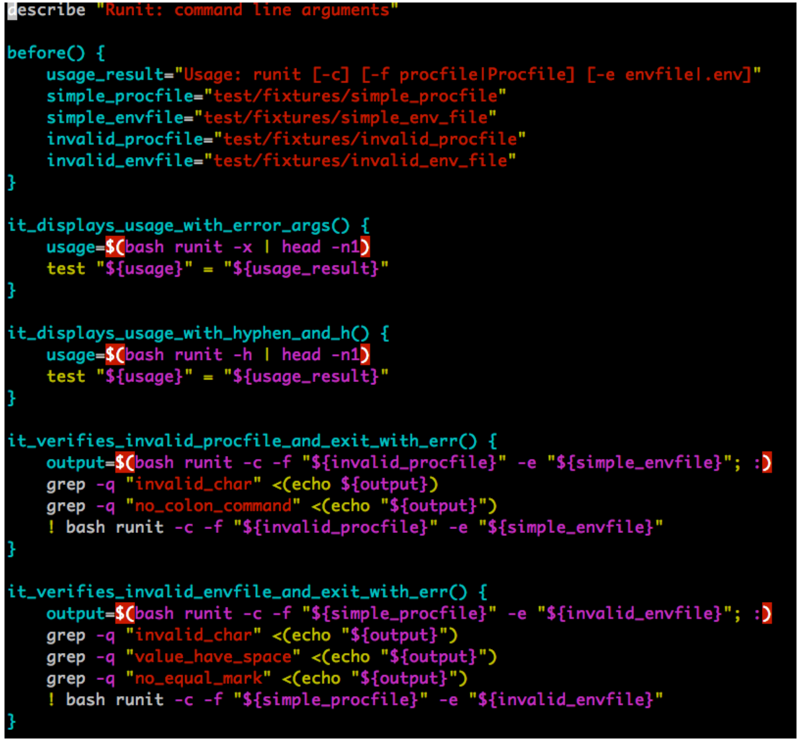
4.4 测试

选择一个库，如：roundup

- 设计测试用例fixtures
- 调用命令，捕获输出
- 测试输出是否符合预期



```
Runit: command line arguments
  it_displays_usage_with_error_args:                [PASS]
  it_displays_usage_with_hyphen_and_h:              [PASS]
  it_verifies_invalid_procfile_and_exit_with_err:   [PASS]
  it_verifies_invalid_envfile_and_exit_with_err:    [PASS]
Runit: envfile
  it_ignores_comments_in_env_file:                  [PASS]
  it_assigns_a_default_port_number:                 [PASS]
  it_allows_overriding_the_port_number:             [PASS]
  it_can_increase_port_number:                      [PASS]
Runit: procfile
  it_runs_simple_processes:                         [PASS]
  it_passes_environment_variables_to_processes:     [PASS]
  it_ignores_comments_in_proc_file:                 [PASS]
=========================================================
Tests:    11 | Passed:   11 | Failed:    0
```



```
describe "Runit: command line arguments"

before() {
    usage_result="Usage: runit [-c] [-f procfile|Procfile] [-e envfile|.env]"
    simple_procfile="test/fixtures/simple_procfile"
    simple_envfile="test/fixtures/simple_env_file"
    invalid_procfile="test/fixtures/invalid_procfile"
    invalid_envfile="test/fixtures/invalid_env_file"
}

it_displays_usage_with_error_args() {
    usage=$(bash runit -x | head -n1)
    test "${usage}" = "${usage_result}"
}

it_displays_usage_with_hyphen_and_h() {
    usage=$(bash runit -h | head -n1)
    test "${usage}" = "${usage_result}"
}

it_verifies_invalid_procfile_and_exit_with_err() {
    output=$(bash runit -c -f "${invalid_procfile}" -e "${simple_envfile}"; :)
    grep -q "invalid_char" <(echo ${output})
    grep -q "no_colon_command" <(echo "${output}")
    ! bash runit -c -f "${invalid_procfile}" -e "${simple_envfile}"
}

it_verifies_invalid_envfile_and_exit_with_err() {
    output=$(bash runit -c -f "${simple_procfile}" -e "${invalid_envfile}"; :)
    grep -q "invalid_char" <(echo "${output}")
    grep -q "value_have_space" <(echo "${output}")
    grep -q "no_equal_mark" <(echo "${output}")
    ! bash runit -c -f "${simple_procfile}" -e "${invalid_envfile}"
}
```

$ 规范制定遵循的原则

连连看

安全: 避免踩坑

视觉易辨识: 代码更可读

简洁: 写法更简单

```
command1 \
    | command2 \
    | command3 \
    | command4
```

```
(( i += 1 ))
```

```
#:<<\###
    do_something
    do_other_thing
###
```

```
main "$@"
```

```
[[ -z "${my_var}" ]]
```

```
false || {
    echo "Something false."
}
```

```
#!/bin/bash
```

```
command1 \
    && command2 \
    && command3
```

```
if [[ -z "${my_var}" ]]; then
    do_something
fi
```

```
set -u
set -e
set -o pipefail
```

```
function usage() {

}
```

$ 规范制定遵循的原则

连连看

安全

```
[[ -z "${my_var}" ]]
```

```
main "$@"
```

```
#!/bin/bash
```

```
set -u
set -e
set -o pipefail
```

视觉易辨识

```
(( i += 1 ))
```

```
command1 \
    | command2 \
    | command3 \
    | command4
```

```
command1 \
    && command2 \
    && command3
```

```
function usage() {

}
```

简洁

```
#:<<\###
    do_something
    do_other_thing
###
```

```
false || {
    echo "Something false."
}
```

```
if [[ -z "${my_var}" ]]; then
    do_something
fi
```

$ 符号的视觉一致性

```
[[ ]]     (( ))                              var    ~   $var

                                             ( )    ~   $( )     ≠   ` `
```

语言开发者设计语言时考虑易于记忆和理解
规范制定者考虑选择易于记忆和理解的写法

```
                                             (( ))  ~   $(( ))   ≠   let/expr

  < file        <<        <<<        <( )           ( )

  > file        >&2       >( )
```

$ 符号的重载

&

cmd &	后台运行
cmd &>/dev/null	stdout和stderr
cmd >&2	文件描述符标识
cmd1 && cmd2	“与”操作

来个复杂的

```
cmd1 && cmd2 &>/dev/null &
```

目录 Agenda

5.1 代码精进

啰嗦

```
grep -q pattern file
if (( $? == 0 )) ; then
   echo "pattern was found"
fi
```

尽量省略$?

```
if grep -q pattern file ; then
   echo "pattern was found"
fi
```

```
cat file \
   | grep pattern \
   | awk '{print $1}'
```

无用的grep命令
无用的cat命令

```
awk '/pattern/{print $1}' file
```

```
echo text | command
```

无用的echo命令

```
command <<< text
```

5.1 代码精进

错误

```
sed 's/p/P/g' $(pwd)
```

需要fd，
但给了string

```
sed 's/p/P/g' <(pwd)
```

```
sed 's/p/P/g' <<< $(pwd)
```

```
PATH=/home/work
ls ${PATH}
```

修改了环境变量，
所有命令都找不到了

```
home=/home/work
ls ${home}
```

```
{  echo  no  ;  echo  last  ;  echo
semicolon  }
```

code block中,
最后一个cmd后要有分号

```
{  echo  no  ;  echo  last  ;  echo
semicolon ;  }
```

5.2 代码维修-1

原始代码

```
#!/bin/bash


LOGS_PATH=/home/work/local/nginx/logs
PID_FILE=/home/work/local/nginx/logs/nginx.pid
LOG_FILE=nginx_access.log
BACKDIR=log.bak


cd ${LOGS_PATH} && mkdir -p ${BACKDIR}
if [ $? -ne 0 ];then
	echo "wrong dir"
	exit 1
fi


# log rotation
mv ${LOG_FILE} ${BACKDIR}/"${LOG_FILE}.$(date +%Y-%m-%d)" && kill -USR1 $(cat ${PID_FILE})


cd ${LOGS_PATH}/${BACKDIR}
# rm too old logs
nice -19 find . -type f -name "${LOG_FILE}.*" -mtime +7 | xargs -r rm -v > nginx_log.del.filelist
```

5.2 代码维修-1

代码 + 评注

```
#!/bin/bash

## 绝对路径，路径重复
LOGS_PATH=/home/work/local/nginx/logs
PID_FILE=/home/work/local/nginx/logs/nginx.pid
LOG_FILE=nginx_access.log
## 备份路径也应放在log目录中
BACKDIR=log.bak

## 逻辑不够清晰, if 判断多余
cd ${LOGS_PATH} && mkdir -p ${BACKDIR}
if [ $? -ne 0 ];then
        echo "wrong dir"
        exit 1
fi

# log rotation
mv ${LOG_FILE} ${BACKDIR}/"${LOG_FILE}.$(date +%Y-%m-%d)" && kill -USR1 $(cat ${PID_FILE})

## cd 命令不安全
cd ${LOGS_PATH}/${BACKDIR}
# rm too old logs
nice -19 find . -type f -name "${LOG_FILE}.*" -mtime +7 | xargs -r rm -v > nginx_log.del.filelist
```

5.2 代码维修-1

维修之后

```
#!/bin/bash

set -ue
set -o pipefail

LOGS_PATH=../logs
PID_FILE=${LOGS_PATH}/nginx.pid
LOG_FILE=${LOGS_PATH}/nginx_access.log
BACKDIR=${LOGS_PATH}/log.bak

mkdir -p ${BACKDIR}

# log rotation
cd ${LOGS_PATH} && {
  mv ${LOG_FILE} ${BACKDIR}/"${LOG_FILE}.$(date +%Y-%m-%d)" \
    && kill -USR1 $(cat ${D_FILE})
}

cd ${BACKDIR} && {
# rm too old logs
  nice -19 find . -type f -name "${LOG_FILE}.*" -mtime +7 \
    | xargs -r rm -v > nginx_log.del.filelist
}
```

5.3 代码维修-2

原始代码

```
#!/bin/bash

cd /home/work/das-bd/bd-bm/
rm ./data/*
cd /home/work/das-bd/bd-bm/data

echo "sequenid:0" >> beidou.info.n
echo "indexid:0"  >> beidou.info.n
echo "line:0"     >> beidou.info.n

 source /home/work/.bash_profile; cd /home/work/script_adv/bin && /usr/bin/python outAdvFee.py
source /home/work/.bash_profile; cd /home/work/script_adv/bin && /usr/bin/python freqOut_3600.py

BS_LIST="m1-mobads-se00.m1.baidu.com cq01-mobads-se00.cq01.baidu.com"
for bs in $BS_LIST; do
  ssh -n $bs "sh /home/work/mobads/product/bs/script/changeIndex.sh"
done
```

5.3 代码维修-2

代码评注

```
#!/bin/bash

## 没有安全设置

## cd是否成功没有判断， 不安全
cd /home/work/das-bd/bd-bm/
rm ./data/*
cd /home/work/das-bd/bd-bm/data

## 如果文件存在， 则一味追加会产生问题
## 可以1次写文件， 写了3次
echo "sequenid:0" >> beidou.info.n
echo "indexid:0"  >> beidou.info.n
echo "line:0"     >> beidou.info.n

 ## 只需要source 1次.bash_profile
source /home/work/.bash_profile; cd /home/work/script_adv/bin && /usr/bin/python outAdvFee.py
source /home/work/.bash_profile; cd /home/work/script_adv/bin && /usr/bin/python freqOut_3600.py

## 使用数组则更加灵活
## 变量的使用没有用${}
BS_LIST="m1-mobads-se00.m1.baidu.com cq01-mobads-se00.cq01.baidu.com"
for bs in $BS_LIST; do
  ## sh 的风险
  ssh -n $bs "sh /home/work/mobads/product/bs/script/changeIndex.sh"
done
```

5.3 代码维修-2

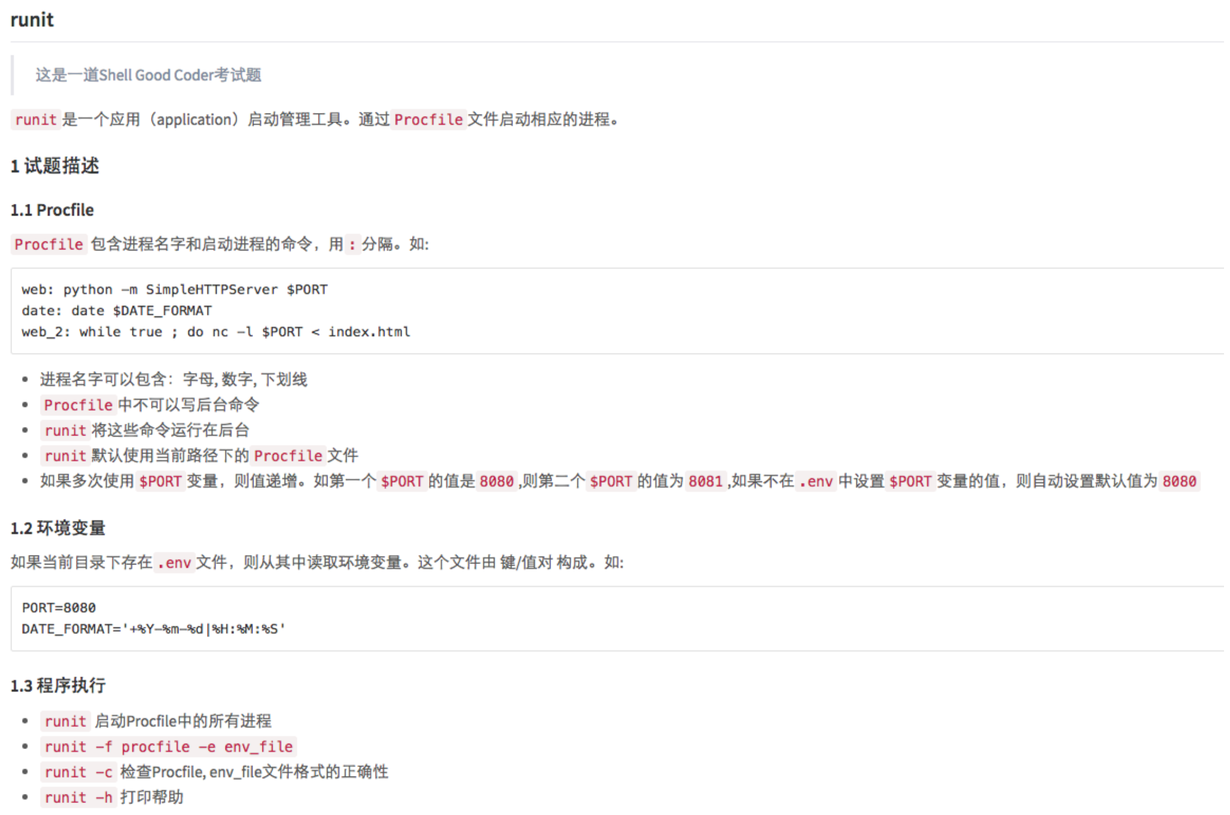
维修之后

```
#!/bin/bash
set -ue
set -o pipefail
source xxx.conf

bd_dm_home=/home/work/das-bd/bd-bm
bd_dm_data=${bd_dm_home}/data
script_adv=/home/work/script_adv/bin
change_index_script=/home/work/mobads/product/bs/script/changeIndex.sh

cd ${bd_dm_dir}/ && {
  rm ./data/*
}
cd ${bd_dm_dir}
cat > beidou.info.n <<- End
sequenid:0
indexid:0
line:0
End

cd ${script_adv} && {
  python outAdvFee.py
  python freqOut_3600.py
}
BS_LIST=(
m1-mobads-se00.m1.baidu.com
cq01-mobads-se00.cq01.baidu.com
)
for bs in ${BS_LIST[@]}; do
  ssh -n ${bs} "bash ${change_index_script}"
done
```

目录 Agenda

6.1 典型任务

Runit：http://git.pt.xiaomi.com/liuzhuo1/runit/tree/master



runit

> 这是一道Shell Good Coder考试题

`runit` 是一个应用（application）启动管理工具。通过 `Procfile` 文件启动相应的进程。

1 试题描述

1.1 Procfile

`Procfile` 包含进程名字和启动进程的命令，用 `:` 分隔。如:

```
web: python -m SimpleHTTPServer $PORT
date: date $DATE_FORMAT
web_2: while true ; do nc -l $PORT < index.html
```

- 进程名字可以包含：字母, 数字, 下划线
- `Procfile` 中不可以写后台命令
- `runit` 将这些命令运行在后台
- `runit` 默认使用当前路径下的 `Procfile` 文件
- 如果多次使用 `$PORT` 变量，则值递增。如第一个 `$PORT` 的值是 `8080` ,则第二个 `$PORT` 的值为 `8081` ,如果不在 `.env` 中设置 `$PORT` 变量的值，则自动设置默认值为 `8080`

1.2 环境变量

如果当前目录下存在 `.env` 文件，则从其中读取环境变量。这个文件由 键/值对 构成。如:

```
PORT=8080
DATE_FORMAT='+%Y-%m-%d|%H:%M:%S'
```

1.3 程序执行

- `runit` 启动Procfile中的所有进程
- `runit -f procfile -e env_file`
- `runit -c` 检查Procfile, env_file文件格式的正确性
- `runit -h` 打印帮助

6.2 任务拆解

最基本的拆分

usage()
main()
log()

进一步拆分

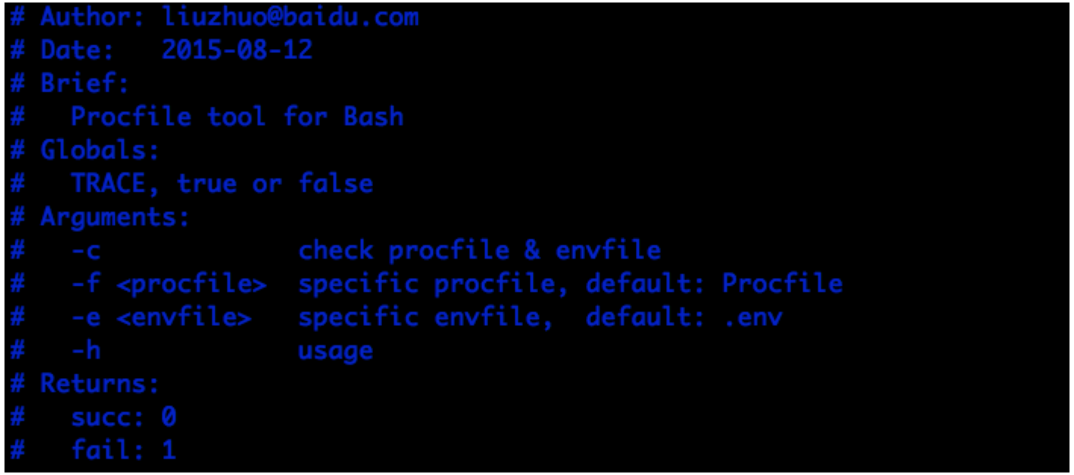
verify()
load_env_file()
run_procfile()

更精细的拆分

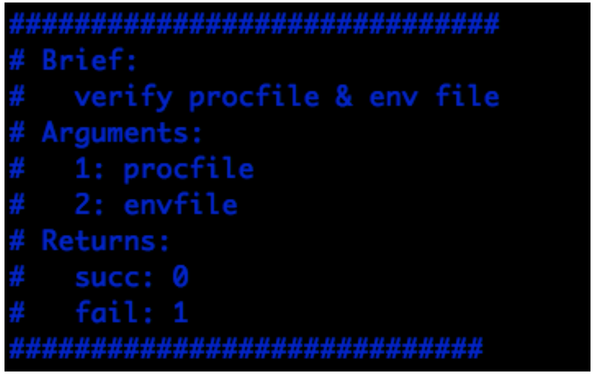
store_pids()
start_command()
on_exit()

6.3 代码版式

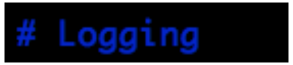
注释



```
# Author: liuzhuo@baidu.com
# Date:   2015-08-12
# Brief:
#   Procfile tool for Bash
# Globals:
#   TRACE, true or false
# Arguments:
#   -c              check procfile & envfile
#   -f <procfile>   specific procfile, default: Procfile
#   -e <envfile>    specific envfile,  default: .env
#   -h              usage
# Returns:
#   succ: 0
#   fail: 1
```



```
################################
# Brief:
#   verify procfile & env file
# Arguments:
#   1: procfile
#   2: envfile
# Returns:
#   succ: 0
#   fail: 1
###############################
```

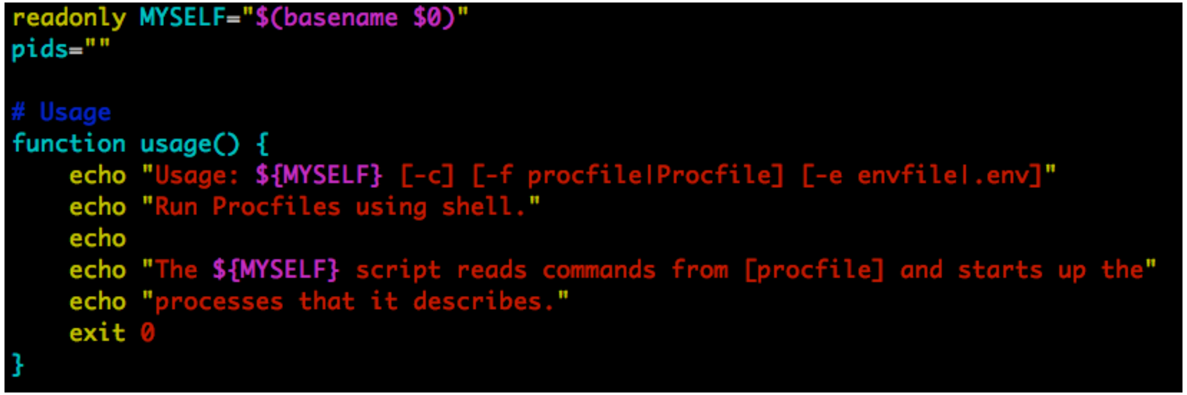


```
# Logging
```

6.4 usage

帮助

获取脚本名字（不受文件改名的影响）：
$(basename $0)



```
readonly MYSELF="$(basename $0)"
pids=""

# Usage
function usage() {
    echo "Usage: ${MYSELF} [-c] [-f procfile|Procfile] [-e envfile|.env]"
    echo "Run Procfiles using shell."
    echo
    echo "The ${MYSELF} script reads commands from [procfile] and starts up the"
    echo "processes that it describes."
    exit 0
}
```

目录 Agenda

$ colon

返回值为true

```
:
echo "return value: $?"   #
0


while :; do
    echo 2
    sleep 1
done
```

空语句

```
if true; then
  :
else
  echo "fail"
fi

function func() {
  :
}
func
```

: 等价于true

$ colon

常用于清空文件

```
# truncate file
: > file

# same as
> file
# or
cat /dev/null > file
```

可用于函数名

```
:
function :() {
  echo "I'm colon function"
}
:
```

多行注释

```
:<<Comments
echo 1
echo 2
echo 3
Comments
```

再懒一点儿

```
:<<\#
echo 1
echo 2
echo 3
#
```

等价于

```
true<<\#
echo 1
echo 2
echo 3
#
```

$ 那些幂等的命令

幂等：不管运行多少次，结果都一样
（输入相同，输出相同）

```
rm -f filename
```

有文件，删除（有权限的前提下）
没文件，返回
结果：文件没了

```
: > filename
```

有文件则清空
没有则创建文件
结果：有了一个空文件

```
mkdir -p dirname
```

有目录则不创建
没有则创建
结果：目录有了

$ [[和 [的困惑

Keyword和 Builtin

[ ] 不支持&& , ||

```
 [[ -f $file1 && ( -d $dir1 | | -d $dir2 ) ]]

[ -f "$file1" -a \( -d "$dir1" -o -d "$dir2" \) ]
```

[[ ]]对变量做引用保护|

```
file="file name"
 [[ -f $file ]] && echo "$file is a regular file"

 file="file name"
 [ -f "$file" ] && echo "$file is a regular file"
```

2.2 参数替换（ Parameter Expansion）

默认值

- :=, :-常用
- :- 使用默认值
- :=使用默认值，并赋值

- 考虑到记忆成本和安全使用，只用

:-

```
#!/bin/bash
# var已定义，但为空
var=

# 如果var
echo ${var:-default}
echo ${var}

echo ${var:=default}
echo ${var}
```

输出

```
default
          #空
default
default
```

```
#!/bin/bash
# var未定义
# var=

# 如果var
echo ${var:-default}
echo ${var}

echo ${var:=default}
echo ${var}
```

输出

```
default
          #空
default
default
```

1.4 I/O重定向（I/O Redirection）

Here Doc & Here String

- ″ - “忽略前置的tab
- ″ \″ 变量替换开关

```
#!/bin/bash

i='something'

cat <<-End
		The documents.
			\\
		$i
End
```

```
#!/bin/bash

i='something'

# “-”忽略前置的tab
# “\”不对'\' '$'做替换
cat <<-\End
		The documents.
			\\
		$i
End
```