

echo

So simple, yet so complex

NAME

- echo -- write arguments to the standard output
- 将输入参数写入标准输出（屏幕）



echo

简单上手

- `$ echo`
 - 输出空行即返回。输入参数为空，因此输出空值后，`echo`自动加上一个换行符"`\n`"
- `$ echo -n`
 - `-n`：不在输出最后添加换行符
- `$ echo hello world`

其它echo参数 (in GNU echo)

- **-E** : 使转义字符失效 (默认如此)
- **-e** : 使转义字符生效

No more ...

echo 转义字符

转义字符	字符的意义
\a	ALERT / BELL(从系统的喇叭送出铃声)
\b	BACKSPACE, 也就是向左退格键
\c	取消行末之换行符号
\E	ESCAPE, 脱字符键
\f	FORMFEED, 换页字符
\n	NEWLINE, 换行字符
\r	RETURN, 回车键
\t	TAB, 表格跳位键
\v	VERTICAL TAB, 垂直表格跳位键
\n	ASCII 八进制编码(以x开头的十六进制和以0开头的十进制), 此处的n为数字
\\	反斜杠本身

echo examples with -e

- `$ echo -e "a\tb\tc\nd\tf"`
- `$ echo -e "\0141\011\0142\011\0143\012\0144\011\0145\011\0146"`
- `\0141: "a" in ASCII; \011: "\t" in ASCII`

echo examples with -e

- `$ echo -e`

```
"\x61\x09\x62\x09\x63\x0a\x64\x09\x65\x09\x66"
```

- ASCII in Hex

- `$ echo -e "a\tb\tc\nd\te\bf\a"`

- `\b` : Backspace

- `\a` : 从系统的喇叭送出铃声

echo 环境变量

- `$ echo $HOSTNAME`
- `$ echo ${USER}`
- `$ echo $?`
 - 获取最后一条指令的返回值，一般"0"表示指令正常退出，而非"0"则表示异常
 - `$ touch a ; ls a 1>/dev/null; echo $?`
 - `$ rm b 2>/dev/null; ls b 1>/dev/null 2>1; echo $?`

SHELL 变量

- `$ A=B` # 设定一个变量A，其值为B
 - `=` 两边不可有**IFS**，并尽量避免包含meta
 - `IFS` : `<Space>` , `<Tab>` , `<Enter>`
 - 变量名不可用\$符
 - 变量首字母不可是数字
 - 变量长度不可超过256字母
 - 变量与值是大小写敏感的

查看 IFS 字符

- `$ echo "$IFS"`
- `$ echo -n "$IFS"`
- `$ echo -n "$IFS" | od -c`

```
$ echo $IFS | od -c
```

```
$ echo "$IFS" | od -c
```

比较一下这两个命令，想想为什么？

SHELL变量定义不会传递

- `$ A=B`
- `$ B=C`
- `$ echo $A`

- `$ B=$A`
- `$ A=C`
- `$ echo $B`



取消变量定义

- `$ unset variable_name`

```
$ A=B  
$ echo $A  
$ unset A  
$ echo $A  
  
$ export A=B  
$ echo $A  
$ unset A  
$ echo $A
```

变量的引用

- `$ echo ${A}`

Easy ?

变量的引用

Elegant !

- `$ file=/dir1/dir2/dir3/my.file.txt`
- `$ echo ${file#*/}` #拿掉第一條 / 及其左邊的字串 : `dir1/dir2/dir3/my.file.txt`
- `$ echo ${file##*/}` #拿掉最後一條 / 及其左邊的字串 : `my.file.txt`
- `$ echo ${file#*.}` #拿掉第一個 . 及其左邊的字串 : `file.txt`
- `$ echo ${file##*.}` #拿掉最後一個 . 及其左邊的字串 : `txt`
- `$ echo ${file%/*}` #拿掉最後條 / 及其右邊的字串 : `/dir1/dir2/dir3`
- `$ echo ${file%%/*}` #拿掉第一條 / 及其右邊的字串 : (空值)
- `$ echo ${file%.*}` #拿掉最後一個 . 及其右邊的字串 : `/dir1/dir2/dir3/my.file`
- `$ echo ${file%%.*}` #拿掉第一個 . 及其右邊的字串 : `/dir1/dir2/dir3/my`

变量的引用

: 去掉左边的 (# 在\$ 的左边)
% : 去掉右边的 (% 在\$ 的右边)

- `$ file=/dir1/dir2/dir3/my.file.txt`
- `$ echo ${file#*/}` #拿掉第一條 / 及其左邊的字串 : `dir1/dir2/dir3/my.file.txt`
- `$ echo ${file##*/}` #拿掉最後一條 / 及其左邊的字串 : `my.file.txt`
- `$ echo ${file#*.}` #拿掉第一個 . 及其左邊的字串 : `file.txt`
- `$ echo ${file##*.}` #拿掉最後一個 . 及其左邊的字串 : `txt`
- `$ echo ${file%/*}` #拿掉最後條 / 及其右邊的字串 : `/dir1/dir2/dir3`
- `$ echo ${file%%/*}` #拿掉第一條 / 及其右邊的字串 : (空值)
- `$ echo ${file%.*}` #拿掉最後一個 . 及其右邊的字串 : `/dir1/dir2/dir3/my.file`
- `$ echo ${file%%.*}` #拿掉第一個 . 及其右邊的字串 : `/dir1/dir2/dir3/my`

变量的引用

- `$ file=/dir1/dir2/dir3/my.file.txt`
- `$ echo ${file:0:5}` # 提取0位置开始的5个字符, /dir1
- `$ echo ${file:5:5}` # 提取5位置开始的5个字符, /dir2
- `$ echo ${#file}` # 计算变量长度

变量的替换

- `$ file=/dir1/dir2/dir3/my.file.txt`
- `$ echo ${file/dir/path}` # 将第一个dir替换为path :
`/path1/dir2/dir3/my.file.txt`
- `$ echo ${file//dir/path}` # 将所有dir替换为path :
`/path1/path2/path3/my.file.txt`

SHELL 数组变量

- `$ A=(a b c def)`
 - 不是 `$ A="a b c def" / A=("a b c def")`
- `$ echo ${A[*]} ; echo ${A[@]} # 全部变量`
- `$ echo ${#A[*]} ; echo ${#A[@]} # 数组长度`
- `$ echo ${A[0]} ; echo ${A[3]} # 特定元素`

变量计算

- `$ A=1 ; B=1; echo $A+$B`
- 结果是 ??

变量计算

- `$ A=1 ; B=1; echo $A+$B`
 - 结果是 ??
- `$ A=1 ; B=1; echo $(($A+$B))` # or `$(A+B)`
- `+` , `-` , `*` , `/` , `%` : 加、减、乘、除、取余

SHELL 变量是无类型的

变量计算



❖ 浮点计算？

❖ 布尔计算？

开始自学！！

SHELL 变量是无类型的

<https://www.shell-tips.com/2010/06/14/performing-math-calculation-in-bash/>

变量值变化与比较

- 变量值变化

- `$ A=5; echo $(A++); echo $A`

- `$ B=5; echo $(B--); echo $B`

- 变量值比较

- `$ echo $(($A > $B))` # 1 为 true

- `$ echo $(($A < $B))` # 0 为 false

- `$ echo $(A > B)` # 比较对象为ASCII符

SHELL变量与环境变量

- 一般的SHELL变量属于local variable
- 经export 命令设置后, 属于environment variable
 - 只有环境变量才可以被子shell继承

```
$ A=B
$ export B=C
$ bash
$ echo $A
$ echo $B
$ echo -e '#!/bin/bash\necho $B' > echoEVB.sh
$ chmod +x echoEVB.sh
$ ./echoEVB.sh
```

环境变量的设置

- `$ export A=B` #设置A为环境变量，并将其值赋为B
- `$ A=B ; export A` #设置变量A值为B，并将其设为环境变量
- `$ declare -x A=B` #设置A为环境变量，并将其值赋为B
- `$ unset A`

SHELL命令的执行过程

- 在Shell中执行命令，其基本过程是：
 - 在当前进程（`shell`）`fork`一个子进程，用于执行对应程序
 - 程序结束后，再返回当前进程（父进程）。
- 下列指令会发生什么？
 - `$ echo "cd $HOME/.ssh" > goto.sh`
 - `$ chmod +x goto.sh`
 - `$./goto.sh`

SHELL命令的执行过程

- `$./*.sh` # 默认的，以`fork`方式执行脚本
- `$ source *.sh` # 在当前进程执行脚本
- `$ exec ./*.sh` # 终止当前进程，并在其中执行新程序

SHELL命令的执行过程

~/demo/ch05/forkShell/1.sh

#!/bin/bash

```
A=B
echo "PID for 1.sh before exec/source/fork:$$"
export A
echo "1.sh: $A is $A"
case $1 in
  exec)
    echo "using exec..."
    exec ./2.sh ;;
  source)
    echo "using source..."
    source 2.sh ;;
  fork)
    echo "using fork by default..."
    ./2.sh ;;
esac

echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: $A is $A"
```

~/demo/ch05/forkShell/2.sh

#!/bin/bash

```
echo "PID for 2.sh: $$"
echo "2.sh get $A=$A from 1.sh"
A=C
export A
echo "2.sh: $A is $A"
```

管道与 标准输入输出

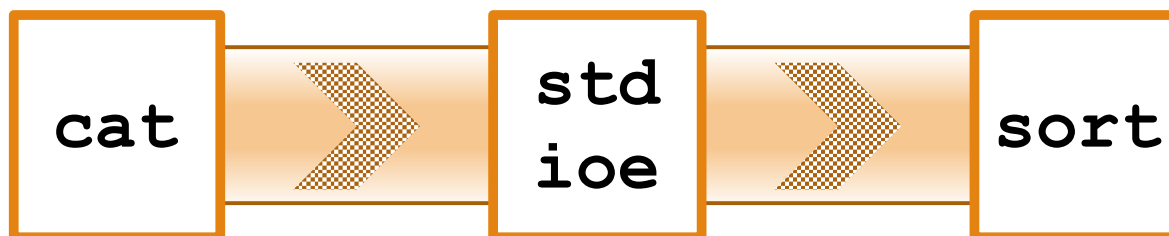
STDIN

STDOUT

STDERR

<, >, >>

- stdin (standard in) : 标准输入
- stdout (standard out) : 标准输出
- stderr (standard error) : 标准错误



Linux 管道

- `pipeline` 是 UNIX 的传统进程通信方式
- 按照 **FIFO** 方式，在两个并发进程间进行数据传送
- 仅需在两个指令间插入 "`|`"，即可建立匿名管道



Linux I/O

- 输入和输出操作是每个程序的必备环节
- 导致大量重复代码，程序与数据文件高度耦合
- 不利于构建通用程序



[~/demo/ch05/pipe/readFile.cpp](#)

记得引用 fstream 库！

```
#include <iostream>
#include <fstream>
using namespace std;
```

只能打开 data 数据文件！

```
int main(int argc, char** argv){
    ifstream myFile;
    myFile.open("data");
    string line;
    while(getline(myFile, line)){
        cout << "Get a line: " << line << endl;
    }
    myFile.close();
    cerr << "Oops, it's the end ! " << endl;
}
```

千万记得关闭文件！

Linux I/O

~/demo/ch05/pipe/readFile.cpp

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main(int argc, char** argv){
6      ifstream myFile;
7      myFile.open("data");
8      string line;
9      while(getline(myFile, line)){
10         cout << "Get a line: " << line << endl;
11     }
12     myFile.close();
13     cerr << "\nOops, it's the end! " << endl;
14 }
```

~/demo/ch05/pipe/stdioe.cpp

```
#include <iostream>

using namespace std;

int main(int argc, char** argv){

    string line;
    while(getline(cin, line)){
        cout << "Get a line: " << line << endl;
    }

    cerr << "\nOops, it's the end ! " << endl;
}
```

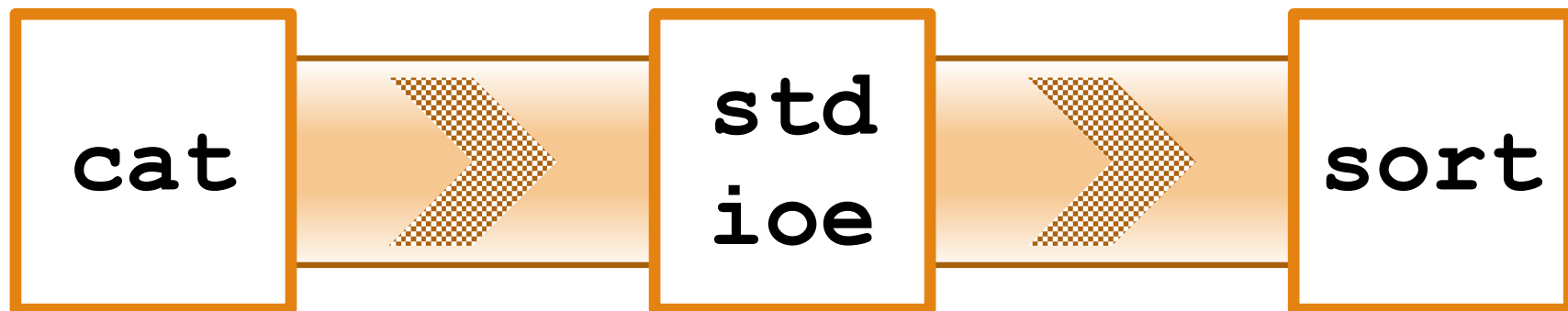
Linux 管道

- `cat` 指令用于在Linux 操作系统中读取文件内容
- `$ cat data | stdioe`



Linux 管道

- `cat` 指令用于在Linux 操作系统中读取文件内容
- `$ cat data | stdioe | sort`



C++

- `cin >> var ;`
- `cout << var << endl ;`
- `cerr << var << endl ;`

Java

- `System.in.read() ;`
- `System.out.println() ;`
- `System.err.println() ;`



标准输入输出

- `stdin` (`standard in`) : 标准输入
 - `cin`, `fread(stdin, ...)`
- `stdout` (`standard out`) : 标准输出
 - `cout`, `fprintf(stdout, ...)`
- `stderr` (`standard error`) : 标准错误
 - `cerr`, `fprintf(stderr, ...)`

重定向符

- < : 标准输入重定向
- > : 标准输出 (**stdout**) 重定向
 - 若文件不存在, 则新建 \$./stdioe > result
 - 若文件已存在, 则覆盖! \$./stdioe < stdioe.cpp > result
- >> : 标准输出 (**stdout**) 重定向
 - 若文件不存在, 则新建
 - 若文件已存在, 则追加

如何区分 **STDOUT** 和 **STDERR** ?

重定向符

- 文件描述符 (`file descriptor`) : 文件的数字标识
 - `STDIN: 0`
 - `STDOUT: 1`
 - `STDERR: 2`
- 重定向符的正式语法是在符号前使用文件描述符
 - `0< ; 1> ; 2>>`
 - `$./stdioe 0< stdioe.cpp 1> result 2>> error`

重定向符

- echo不同类别消息至 `stdout` 和 `stderr`
 - `$ cd ~/demo/ch05`
 - `$./redirect.sh`
 - `$./redirect.sh >/dev/null`

组合STDOUT & STDERR

- `$ command x>outputfile y>&x`
 - `x` 输出到`outputfile`, 然后`y`输出到`x`
 - `$./stdioe 0< stdioe.cpp 1> result 2>&1`
- 注意！ `$ command y>&x x> outputfile` 无效！
 - 在处理 `y>&x` 时, `x`仍输出向原始位置
 - 然后`x`才更改了输出位置

组合STDOUT & STDERR

- 在 Bourne shell 家族中，可以采用下述方法
 - `$./stdioe2 0< stdioe2.cpp &> result`
- 如果要追加，那么只能
 - `$./stdioe2 0< stdioe2.cpp >> result 2>&1`

抛弃输出

- `$./stdioe2 0< stdioe2.cpp >/dev/null`
 - 只显示错误信息
- `$./stdioe2 0< stdioe2.cpp 2>/dev/null`
 - 忽略错误信息
- `/dev/null` 是一个伪设备文件，`/dev/zero`类似。
任何写入这两种设备上的输出会被抛弃

Shell 编程基础环境



师父领进门
修行靠个人

