Shell

是否需要分号

可以用可以不用，但在同一行的语句需要用分号

`echo "Hello World !"` 输出

`echo` 命令在 Linux 中非常常用，它的主要作用是**显示文本或变量的值**到终端。简单来说，`echo` 就是用来“打印”一些内容，通常用于输出信息、调试、或者在脚本中显示结果。

- 基本用法：

  echo "Hello, World!"

  这条命令会在终端输出：

  Hello, World!

- 变量使用：

你也可以用 `echo` 来显示变量的值，比如：

```
name="Alice"
echo "Hello, $name!"
```

这会输出：

```
Hello, Alice!
```

- 特性：
  - **换行**：默认情况下，`echo` 会输出后跟换行符。
  - **不换行**：如果你不想换行，可以加上 `-n` 参数：

echo –n "Hello, "
echo "World!"

这会输出：

Hello, World!

（没有换行）

- 使用转义字符：

`echo` 也支持一些特殊的转义字符，比如：

- `\n`：换行
- `\t`：制表符（Tab）

例如：

```
echo -e "Hello\nWorld"
```

这会输出：

```
Hello
World
```

`-e` 参数启用转义字符。

> ### 总结：
>
> `echo` 是一个输出工具，可以用来显示文本、变量、甚至执行简单的格式化输出。它在脚本中经常用来输出调试信息或者提示用户一些操作结果。

运行 Shell 脚本有两种方法：

1、作为可执行程序

将上面的代码保存为 test.sh，并 cd 到相应目录：

```
chmod +x ./test.sh  #使脚本具有执行权限
./test.sh  #执行脚本
```

注意，一定要写成 ./test.sh，而不是 **test.sh**，运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有 /bin, /sbin, /usr/bin，/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用 ./test.sh 告诉系统说，就在当前目录找。

2、作为解释器参数

这种运行方式是，直接运行解释器，其参数就是 shell 脚本的文件名，如：

```
Plain Text
1   /bin/sh test.sh
2   /bin/php test.php
```

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

变量

- 注意
  - 命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
  - 中间不能有空格，可以使用下划线(_)。
  - 不能使用标点符号。
  - 不能使用bash里的关键字（可用help命令查看保留关键字）。
- 使用变量

  使用一个定义过的变量，只要在变量名前面加美元符号即可，如：

  ```
  Plain Text
  1   your_name="qinjx"
  2   echo $your_name
  3   echo ${your_name}
  ```

- 只读变量

  使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。

- 删除变量

  使用 unset 命令可以删除变量。语法：

  `unset variable_name`

- 变量类型
  - **1) 局部变量**局部变量在脚本或命令中定义，仅在当前shell示例中有效，其他shell启动的程序不能访问局部变量。
  - **2) 环境变量**所有的程序，包括shell启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候shell脚本也可以定义环境变量。
  - **3) shell变量**shell变量是由shell程序设置的特殊变量。shell变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了shell的正常运行
- 字符串

  字符串是shell编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

  - 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
  - 单引号字串中不能出现单独一个的单引号（对单引号使用转义符后也不行），但可成对出现，作为字符串拼接使用。
  - 双引号里可以有变量
  - 双引号里可以出现转义字符
- 提取字符串

  以下示例从字符串第 2 个字符开始截取 4 个字符：

```
Plain Text
1   string="nhooo is a great site"
2   echo ${string:1:4} # 输出 hooo
```

  注意：第一个字符的索引值为 0。

- 定义数组

  在 Shell 中，用括号来表示数组，数组元素用"空格"符号分割开。定义数组的一般形式为：

  `数组名=(值1 值2 ... 值n)`

  例如：

  `array_name=(value0 value1 value2 value3)`

  或者

```
Plain Text
1  array_name=(
2  value0
3  value1
4  value2
5  value3
6  )
```

  还可以单独定义数组的各个分量：

```
Plain Text
1  array_name[0]=value0
2  array_name[1]=value1
3  array_name[n]=valuen
```

  可以不使用连续的下标，而且下标的范围没有限制。

- 读取数组

  读取数组元素值的一般格式是：

  `${数组名[下标]}`

  例如：

  `valuen=${array_name[n]}`

  使用 @ 符号可以获取数组中的所有元素，例如：

  `echo ${array_name[@]}`

- 获取数组长度

  获取数组的的方法与获取字符串长度的方法相同，例如：

取得数组元素的个数

```
length=${#array_name[@]}
```

或者

```
length=${#array_name[*]}
```

取得数组单个元素的长度

```
lengthn=${#array_name[n]}
```

- 注释

多行注释

多行注释还可以使用以下格式：

```Plain Text
:<<EOF
注释内容...
注释内容...
注释内容...
EOF
```

EOF 也可以使用其他符号:

```Plain Text
:<<'
注释内容...
注释内容...
注释内容...
'
:<<!
注释内容...
注释内容...
注释内容...
!
```

基本运算符

- 格式
  - 表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。
  - 完整的表达式要被 ` ` ` 包含，注意这个字符不是常用的单引号，在 Esc 键下边。
  - 原生bash不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。
  - expr 是一款表达式计算工具，使用它能完成表达式的求值操作。
  - 引用变量时需要$符号

- 例子

```sh
a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a 等于 b"
fi
if [ $a != $b ]
then
   echo "a 不等于 b"
fi
```

- 注意
  - 乘号(*)前边必须加反斜杠(\)才能实现乘法运算；
  - if...then...fi 是条件语句
  - 在 MAC 中 shell 的 expr 语法是：**$((表达式))**，此处表达式中的 "*" 不需要转义符号 "\" 。

- 关系运算符

  关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

  下表列出了常用的关系运算符，假定变量 a 为 10，变量 b 为 20：

  1. 运算符：`-eq` --> 检测两个数是否相等，相等返回 true。 -->> [$a -eq $b] 返回 false。
  2. 运算符：`-ne` --> 检测两个数是否不相等，不相等返回 true。 -->> [$a -ne $b] 返回 true。
  3. 运算符：`-gt` --> 检测左边的数是否大于右边的，如果是，则返回 true。 -->> [$a -gt $b] 返回 false。
  4. 运算符：`-lt` --> 检测左边的数是否小于右边的，如果是，则返回 true。 -->> [$a -lt $b] 返回 true
  5. 运算符：`-ge` --> 检测左边的数是否大于等于右边的，如果是，则返回 true。 -->> [$a -ge $b] 返回 false。
  6. 运算符：`-le` --> 检测左边的数是否小于等于右边的，如果是，则返回 true。 -->> [$a -le $b] 返回 true。

  - **eq** (equal to): This operator checks if two numbers are equal. If they are, it returns true. Example: `[ $a -eq $b ]` returns `true` if `$a` equals `$b`.
  - **ne** (not equal to): This operator checks if two numbers are not equal. If they are not equal, it returns true. Example: `[ $a -ne $b ]` returns `true` if `$a` does not equal `$b`.
  - **gt** (greater than): This operator checks if the number on the left is greater than the number on the right. If true, it returns true. Example: `[ $a -gt $b ]` returns `true` if `$a` is greater than `$b`.
  - **lt** (less than): This operator checks if the number on the left is less than the number on the right. If true, it returns true. Example: `[ $a -lt $b ]` returns `true` if `$a` is less than `$b`.
  - **ge** (greater than or equal to): This operator checks if the number on the left is greater than or equal to the number on the right. If true, it returns true. Example: `[ $a -ge $b ]` returns `true` if `$a` is greater than or equal to `$b`.
  - **le** (less than or equal to): This operator checks if the number on the left is less than or equal to the number on the right. If true, it returns true. Example: `[ $a -le $b ]` returns `true` if `$a` is less than or equal to `$b`.

例子

Bash

```bash
a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a 等于 b"
else
   echo "$a -eq $b: a 不等于 b"
fi
if [ $a -ne $b ]
then
   echo "$a -ne $b: a 不等于 b"
else
   echo "$a -ne $b : a 等于 b"
fi
if [ $a -gt $b ]
then
   echo "$a -gt $b: a 大于 b"
else
   echo "$a -gt $b: a 不大于 b"
fi
if [ $a -lt $b ]
then
   echo "$a -lt $b: a 小于 b"
else
   echo "$a -lt $b: a 不小于 b"
fi
if [ $a -ge $b ]
then
   echo "$a -ge $b: a 大于或等于 b"
else
   echo "$a -ge $b: a 小于 b"
fi
if [ $a -le $b ]
then
   echo "$a -le $b: a 小于或等于 b"
else
   echo "$a -le $b: a 大于 b"
fi
```

- 字符串运算符

1. `=` 相等为true
2. `!=` 不相等为true
3. `-z` 字符串长度是否为0，为0返回true
4. `-n` 字符串长度是否不为0，不为0返回true
5. `$` 字符串是否为空，不为空返回true

Plain Text

```
a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a 等于 b"
else
   echo "$a = $b: a 不等于 b"
fi
if [ $a != $b ]
then
   echo "$a != $b : a 不等于 b"
else
   echo "$a != $b: a 等于 b"
fi
if [ -z $a ]
then
   echo "-z $a : 字符串长度为 0"
else
   echo "-z $a : 字符串长度不为 0"
fi
if [ -n "$a" ]
then
   echo "-n $a : 字符串长度不为 0"
else
   echo "-n $a : 字符串长度为 0"
fi
if [ $a ]
then
   echo "$a : 字符串不为空"
else
   echo "$a : 字符串为空"
fi
```

文件测试运算符：

Operator	Description	Example
-b	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-c	检测文件是否是字符设备文件，如果是，则返回 true。	[-c $file] 返回 false。
-d	检测文件是否是目录，如果是，则返回 true。	[-d $file] 返回 false。
-f	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f $file] 返回 true。
-g	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g $file] 返回 false。
-k	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k $file] 返回 false。
-p	检测文件是否是有名管道，如果是，则返回 true。	[-p $file] 返回 false。
-u	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u $file] 返回 false。
-r	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-w	检测文件是否可写，如果是，则返回 true。	[-w $file] 返回 true。
-x	检测文件是否可执行，如果是，则返回 true。	[-x $file] 返回 true。
-s	检测文件是否为空（文件大小是否大于0），不为空返回 true。	[-s $file] 返回 true。
-e	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file] 返回 true。

-S	判断某文件是否 socket。	[-S $file] 返回 true。
-L	检测文件是否存在并且是一个符号链接。	[-L $file] 返回 true。

实例：

Plain Text

```
file="/var/www/nhooo/test.sh"
if [ -r $file ]
then
   echo "文件可读"
else
   echo "文件不可读"
fi
if [ -w $file ]
then
   echo "文件可写"
else
   echo "文件不可写"
fi
if [ -x $file ]
then
   echo "文件可执行"
else
   echo "文件不可执行"
fi
if [ -f $file ]
then
   echo "文件为普通文件"
else
   echo "文件为特殊文件"
fi
if [ -d $file ]
then
   echo "文件是个目录"
else
   echo "文件不是个目录"
fi
if [ -s $file ]
then
   echo "文件不为空"
else
   echo "文件为空"
fi
if [ -e $file ]
then
   echo "文件存在"
else
   echo "文件不存在"
fi
```

执行脚本，输出结果如下所示：

```
文件可读
文件可写
文件可执行
文件为普通文件
文件不是个目录
文件不为空
文件存在
```

循环

大部分循环和其他变成语言的一致，没有太特别的地方

- until循环

  until 循环执行一系列命令直至条件为 true 时停止。

  until 循环与 while 循环在处理方式上刚好相反。

  一般 while 循环优于 until 循环，但在某些时候—也只是极少数情况下，until 循环更加有用。

  until 语法格式:

  ```
  until condition
  do
      command
  done
  ```

  condition 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

- case语句

case ... esac 为多选择语句，与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构，每个 case 分支用右圆括号开始，用两个分号 ;; 表示 break，即执行结束，跳出整个 case ... esac 语句，esac（就是 case 反过来）作为结束标记。

可以用 case 语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。

case ... esac 语法格式如下：

Plain Text

```
case 值 in
模式1)
    command1
    command2
    ...
    commandN
    ;;
模式2)
    command1
    command2
    ...
    commandN
    ;;
esac
```

case 工作方式如上所示，取值后面必须为单词 **in**，每一模式必须以右括号结束。取值可以为变量或常数，匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

函数

linux shell 可以用户定义函数，然后在shell脚本中可以随便调用。

shell中函数的定义格式如下：

```
[ function ] funname [()]
{
    action;
    [return int;]
}
```

说明：

- 1、可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。
- 2、参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return后跟数值n(0-255)

例如：

```
demoFun(){
    echo "这是我的第一个 shell 函数!"
}
echo "-----函数开始执行-----"
demoFun
echo "-----函数执行完毕-----"
```

- 函数参数

在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 n的形式来获取参数的值，例如， 1表示第一个参数，$2表示第二个参数...

带参数的函数示例：

```
funWithParam(){
    echo "第一个参数为 $1 !"
    echo "第二个参数为$2 !"
    echo "第十个参数为 $10 !"
    echo "第十个参数为${10} !"
    echo "第十一个参数为 ${11} !"
    echo "参数总数有$# 个!"
    echo "作为一个字符串输出所有参数 $* !"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出结果：

```
第一个参数为 1 !
第二个参数为 2 !
第十个参数为 10 !
第十个参数为 34 !
第十一个参数为 73 !
参数总数有 11 个!
作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 !
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。

shell输入 / 输出重定向

重定向命令列表如下：

command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。

n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

需要注意的是文件描述符 0 通常是标准输入(STDIN)，1 是标准输出(STDOUT)，2 是标准错误输出(STDERR)。

- 什么是输入输出重定向?

  Shell 输入/输出重定向是将命令的输入或输出从默认的标准输入（通常是键盘）和标准输出（通常是屏幕）转移到文件或其他设备的过程。这让你能够控制数据流，操作命令和文件的交互。

1. 输出重定向（`>` 和 `>>`）

输出重定向将命令的输出（标准输出）写入指定的文件中，而不是显示在屏幕上。

- `>`：将输出重定向到文件，若文件存在则覆盖。
  - 例如：`echo "Hello, World!" > output.txt`，这个命令会把 `"Hello, World!"` 写入 `output.txt`，如果文件已存在，会被覆盖。
- `>>`：将输出追加到文件末尾，不会覆盖文件内容。
  - 例如：`echo "Hello again!" >> output.txt`，这个命令会把 `"Hello again!"` 添加到 `output.txt` 的末尾。

2. 输入重定向（`<`）

输入重定向将文件的内容作为命令的输入。

- 例如：`sort < input.txt`，这个命令将 `input.txt` 文件的内容传递给 `sort` 命令进行排序。

3. 错误输出重定向（`2>` 和 `2>>`）

标准错误输出（通常是错误消息）可以被重定向到文件中。

- `2>`：将错误输出重定向到文件，若文件存在则覆盖。
  - 例如：`command_that_fails 2> error.log`，将失败命令的错误消息写入 `error.log`。
- `2>>`：将错误输出追加到文件末尾。
  - 例如：`command_that_fails 2>> error.log`，将错误信息追加到 `error.log`。

4. 同时重定向输出和错误（`&>`）

可以将标准输出和标准错误一起重定向到同一个文件。

- 例如：`command &> output_and_error.log`，会把标准输出和错误信息都写入 `out`

`put_and_error.log`。

5. 管道（`|`）

管道将一个命令的输出传递给另一个命令作为输入。

- 例如：`ls | grep "file"`，会列出当前目录下的所有文件，并将这些文件名通过管道传给 `grep` 命令，查找包含 "file" 的文件。

通过输入/输出重定向，能更灵活地控制命令的执行结果与错误信息，进而优化工作流程和数据处理。

无标题折叠块

Shell 输入/输出重定向深入讲解

在 Unix/Linux 系统中，每个进程默认打开三个文件流：

1. **标准输入（stdin）**：
   - 文件描述符：`0`。
   - 默认从键盘输入数据。
2. **标准输出（stdout）**：
   - 文件描述符：`1`。
   - 默认输出到屏幕，显示程序的正常输出。
3. **标准错误（stderr）**：
   - 文件描述符：`2`。
   - 默认输出错误信息到屏幕。

基本重定向

- **标准输出重定向**：
  - `command > file`：将 `command` 的标准输出（stdout）重定向到 `file` 文件。
    - 若 `file` 已存在，则覆盖。
  - 例如：`echo "Hello, World!" > output.txt`。
- **标准输入重定向**：
  - `command < file`：将 `file` 文件的内容作为 `command` 的输入（stdin）。
  - 例如：`sort < input.txt`。
- **标准错误输出重定向**：
  - `command 2> file`：将错误信息（stderr）重定向到 `file` 文件。
  - 例如：`command_that_fails 2> error.log`。
- **标准错误输出追加重定向**：
  - `command 2>> file`：将错误信息追加到 `file` 文件末尾。
  - 例如：`command_that_fails 2>> error.log`。

合并标准输出和标准错误

- **合并输出与错误**：

例如：

```
command > output.log 2>&1
```

- `command > file 2>&1`：将标准输出（stdout）和标准错误（stderr）合并后一起重定向到 `file`。
- `command >> file 2>&1`：将标准输出和标准错误追加到 `file` 中。
- **解释**：`2>&1` 的意思是将 `stderr`（文件描述符 `2`）重定向到与 `stdout`（文件描述符 `1`）相同的目的地。

同时重定向标准输入和标准输出

- `command < file1 > file2`：将标准输入从 `file1` 中获取，将标准输出写入 `file2` 中。
  - 例如：`command < input.txt > output.txt`。

特殊的重定向方式

Here Document（`<<`）

Here Document 用于将多行内容传递给命令或脚本，而不需要通过外部文件。

- **基本语法**：

```
command << delimiter
    document
delimiter
```

  - `delimiter` 是自定义的标识符，标识输入内容的开始与结束。
  - `document` 是输入给命令的多行文本。

- **示例**：

```
wc -l << EOF
欢迎来到
基础教程网
www.cainiaoplus.com
EOF
```

  - 结果为：`3`，表示 `EOF` 之间有三行内容。

- **脚本中使用**：

```
#!/bin/bash
cat << EOF
欢迎来到
基础教程网
www.cainiaoplus.com
EOF
```

- 执行该脚本会输出：

```
欢迎来到
基础教程网
www.cainiaoplus.com
```

`/dev/null` 文件

`/dev/null` 是一个特殊设备文件，所有写入它的数据都会被丢弃，相当于“黑洞”。

- **禁用标准输出**：
  - `command > /dev/null`：将标准输出重定向到 `/dev/null`，即禁止显示输出。
  - 例如：`echo "This will not show" > /dev/null`。
- **同时禁用标准输出和标准错误**：
  - `command > /dev/null 2>&1`：将标准输出和标准错误都重定向到 `/dev/null`，即不显示任何输出和错误信息。
  - 例如：`command > /dev/null 2>&1`。

文件描述符解释

- **标准输入（stdin）**：文件描述符 `0`。
- **标准输出（stdout）**：文件描述符 `1`。
- **标准错误（stderr）**：文件描述符 `2`。

总结

- 重定向允许你将标准输入、输出或错误流定向到文件或设备，增强了 Shell 命令的灵活性。
- 通过将多个文件描述符组合使用，可以实现更复杂的重定向操作。
- Here Document 提供了一种将多行文本作为输入传递给命令的便捷方法。
- `/dev/null` 可以帮助我们丢弃不需要的输出或错误信息，防止它们干扰屏幕显示。

Shell文件包含

- Shell 文件包含

在 Shell 脚本中，可以使用 **文件包含** 来引入外部脚本文件，这样可以将常用的代码封装在独立文件中，然后在多个脚本中重复使用。Shell 提供了两种方式来包含外部脚本文件：

1. `.` (dot) 命令
2. `source` 命令

这两者的作用基本相同，都可以将外部脚本的内容引入到当前脚本中。二者的区别只是语法形式，`.` 更简洁，而 `source` 则更具可读性。

语法格式

1. 使用 `.` 命令：

```
. filename
```

2. 使用 `source` 命令：

```
source filename
```

其中，`filename` 是要被包含的脚本文件路径。**注意：** 引用时，`.` 和文件名之间要有一个空格。

示例

假设我们有两个脚本文件 `test1.sh` 和 `test2.sh`：

test1.sh

```
#!/bin/bash
# author: 基础教程网
# url: www.cainiaoplus.com
url="http://www.cainiaoplus.com"
```

test2.sh

```
#!/bin/bash
# author: 基础教程网
# url: www.cainiaoplus.com

# 使用 . 号来引用 test1.sh 文件
. ./test1.sh

# 或者使用以下包含文件代码：
# source ./test1.sh

echo "基础教程网官网地址：$url"
```

步骤

1. **为 test2.sh 脚本添加执行权限：**

```
chmod +x test2.sh
```

2. **执行 test2.sh 脚本：**

```
./test2.sh
```

输出结果：

```
基础教程网官网地址：http://www.cainiaoplus.com
```

说明

- `test1.sh` 文件包含了一个变量 `url`，该变量在 `test2.sh` 中被引入并使用。
- `test2.sh` 文件通过 `.` 或 `source` 引入 `test1.sh`，然后使用 `url` 变量。
- 被包含的脚本 `test1.sh` **不需要**具有可执行权限，因为它是作为一个源文件被引入执行的。

小结

- 使用 `.` 或 `source` 可以将外部脚本的内容引入到当前脚本中，从而使代码更加模块化和可重用。
- 外部脚本可以包含变量、函数、配置等内容，提供了很大的灵活性。

- 被包含的脚本文件只需要具备读取权限，无需执行权限。