

# STAT243 Lecture 3.3 Bash Shell-Using Commands

## 1 基本命令

### 1.1 命令中的括号

#### 1.1.1 花括号 {} 的使用场景

- 变量拓展:

```
>- Shell
1 # 明确变量边界
2 echo "${variable}_suffix"
3
4 # 默认值设置
5 echo "${VAR:-default_value}"
6
7 # 字符串操作
8 echo "${string:0:5}" # 子字符串
9 echo "${string#prefix}" # 删除前缀
```

- 序列生成:

```
>- Shell
1 # 数字序列
2 echo {1..5} # 输出: 1 2 3 4 5
3 echo {5..1} # 输出: 5 4 3 2 1
4 echo {01..10} # 输出: 01 02 03 04 05 06 07 08 09 10
5
6 # 字符序列
7 echo {a..e} # 输出: a b c d e
```

- 文件名拓展 (通配符):

```
>- Shell
1 # 创建多个文件
2 touch file{1..3}.txt # 创建 file1.txt, file2.txt, file3.txt
3
4 # 复制多个文件
5 cp file.{txt,bak} # 复制 file.txt 到 file.bak
```

- 命令分组:

```
>- Shell
1 # 命令组合并重定向
2 { command1; command2; } > output.txt
3
4 # 注意: 花括号内命令必须以分号结尾, 且花括号与命令间必须有空格
```

#### 1.1.2 圆括号 () 的使用场景

- 创建子 Shell (在新进程中执行)

```
Shell
```

```
1 # 在子 Shell 中执行命令
2 ( cd /tmp; ls )          # 目录改变不会影响当前 Shell
3
4 # 后台进程组
5 ( sleep 10; echo "Done" ) &
```

- 命令替换

```
>- Shell
1 # 捕获命令输出
2 current_date=$(date)    # 等同于 `date`
3
4 # 嵌套命令
5 files=$(ls $(pwd))
```

- 数组定义

```
>- Shell
1 # 创建数组
2 fruits=("apple" "banana" "cherry")
3
4 # 访问数组元素
5 echo "${fruits[0]}"     # 输出: apple
```

- 算术运算

```
>- Shell
1 # 使用 $(( )) 进行算术运算
2 result=$(( 5 + 3 * 2 )) # 输出: 11
```

- 进程替换

```
>- Shell
1 # 比较两个命令的输出
2 diff <(ls dir1) <(ls dir2)
```

## | 1.2 重要命令

有许多有用的命令和工具来查看和操作文件:

- `cat` – concatenate 文件与打印到 standard output 详见 [cat](#)
- `cp` – 复制文件和目录, 详见 [cp](#)
- `cut` – 从文件的每一行中提取部分内容, 详见 [cut](#)
- `diff` – 查找两个文件之间的差异, 详见 [diff](#)
- `grep` – 打印匹配特定模式的行, 详见 [grep](#)
- `head` – 输出文件的开头部分, 详见 [head](#)
- `find` – 在 directory hierarchy 中搜索文件, 详见 [find](#)
- `less` – 是 `more` 的增强版, 详见 [less](#)
- `more` – 用于分页显示文件内容, 详见 [more](#)
- `mv` – 移动 (重命名) 文件, 详见 [mv](#)
- `nl` – 为文件中的行编号, 详见 [nl](#)
- `paste` – 合并文件中的行, 详见

- `rm` – 删除文件或目录, 详见
- `rmdir` – 删除空目录, 详见
- `sort` – 对文本文件中的行排序, 详见
- `split` – 将文件分割成多个部分, 详见
- `tac` – 反向连接并打印文件, 详见
- `tail` – 输出文件的最后一部分, 详见
- `touch` – 更改文件时间戳, 详见
- `tr` – translate 或 delete 字符, 详见
- `uniq` – 删除 sorted file 中的重复行 (若相邻行中有重复, 则仅保留一个), 详见
- `wc` – 显示文件中的字节数、单词数和行数, 详见
- `wget` and `curl` – non-interactive 式网络下载, 详见

## 1.3 UNIX 命令的一般语法

### 1.3.1 一般语法结构

UNIX 命令的一般语法结构如下:

```

>- Shell
1 $ command -options argument1 argument2 ...

```

#### Example ▾

```

>- Shell
1 $ grep -i graphics file.txt

```

会在 `file.txt` (参数 2) 中查找字面字符串 `graphics` (参数 1), 并且选项 `-i` 表示忽略字母的大小写

#### Example ▾

```

>- Shell
1 $ less file.txt

```

允许我们简单地翻阅文本文件 (可以用空格键和上下箭头键上下导航), 以便了解文件内容。可以通过输入 `q` 来退出 `less`

### 1.3.2 Options (Flags)

- 单破折号选项 (`-`): 短格式 (如 `-n 10`)。
- 双破折号选项 (`--`): 长格式 (如 `--help`)。

```

>- Shell
1 $ tail --help

```

- 常见规则:
  - 选项可连写, 如 `-al` 等价于 `-a -l`;
  - 参数与选项之间的空格可省略或保留。

### 1.3.3 Example: `tail` Command

```
Shell
1 $ wget https://raw.githubusercontent.com/berkeley-scf/tutorial-using-bash/master/cpds.csv
2 $ tail -n 10 cpds.csv # 输出文件末尾10行
3 $ tail -f cpds.csv # 实时跟踪文件变化
```

#### Logic

`wget` 与 `curl` 都可用于下载网络文件。

- Linux 通常自带 `wget` ；
- macOS 默认仅提供 `curl` 。

### 1.3.4 Example: `grep` Command

`grep` 用于按 **模式 (pattern)** 搜索文本。

```
Shell
1 $ grep ^2001 cpds.csv # 以 '2001' 开头的行
2 $ grep 0$ cpds.csv # 以 '0' 结尾的行
3 $ grep 19.0 cpds.csv # 匹配 19任意字符0
4 $ grep 19.*0 cpds.csv # 匹配 19 与 0 之间任意字符
5 $ grep -o 19.0 cpds.csv # 仅输出匹配部分
```

#### Remark

`grep` 使用的 **正则表达式语法** 与文件通配符不同。  
在需要匹配特殊字符（如 `.`、`"`）时必须使用反斜杠转义：

```
Shell
1 $ grep "\"Canada\"" cpds.csv # 匹配含引号的 "Canada"
2 $ grep "19\\.0" cpds.csv # 匹配文本 19.0
```

### 1.3.5 Quoting Patterns

在包含空格或特殊字符的模式时，使用引号可避免 `shell` 错误解析：

```
Shell
1 $ grep "George .* Bush" cpds.csv
```

#### Logic

用双引号包裹字符串能确保它被视为单一参数。  
在 `shell` 中 **"pattern with space"** 是安全做法。

### 1.3.6 Example: Working with Large Data Files

- 使用 `grep` 按行筛选数据，或使用 `cut` 提取字段：

```
Shell
1 $ grep "Canada" bigdata.csv > subset.csv
2 $ cut -d',' -f1,3 subset.csv
```

- 相比在 R/Python/SAS 中读入大文件，这种方式速度更快且占用更少内存。

🔗 Logic ▾

Unix 工具如 `grep`，`cut`，`awk`，`sort`，`uniq` 能在命令行快速处理 GB 级文件，常被用于数据预处理和日志筛查。

## | 2 Streams, Pipes, and Redirects

### | 2.1 Streams (stdin / stdout / stderr)

在 Unix 系统中，程序通过 **数据流 (streams)** 与外部交互：

名称	缩写	默认方向	默认设备	说明
标准输入	<code>stdin</code>	输入	键盘	程序从此处读取数据
标准输出	<code>stdout</code>	输出	屏幕	程序的正常输出结果
标准错误	<code>stderr</code>	输出	屏幕	程序的错误信息与警告

在交互式 Shell 会话中：

- **输入** 默认来自键盘；
- **输出** 与 **错误信息** 默认显示在屏幕。

🔗 Logic ▾

通过重定向 (redirection)，可改变这三个流的输入输出位置，例如将输出保存到文件、将输入读取自文件，或隐藏错误信息。

### | 2.2 Overview of Redirection

Shell 提供了通用的 **重定向操作符**，可改变程序的输入输出位置。

下表总结了常见语法与功能：

重定向语法	功能说明
<code>cmd &gt; file</code>	将标准输出 (stdout) 写入文件 (覆盖原内容)
<code>cmd 1&gt; file</code>	同上 (1 代表 stdout)
<code>cmd 2&gt; file</code>	将标准错误 (stderr) 写入文件
<code>cmd &gt; file 2&gt;&amp;1</code>	将 stdout 与 stderr 同时写入文件
<code>cmd &lt; file</code>	从文件中读取输入 (stdin)
<code>cmd &gt;&gt; file</code>	将标准输出追加到文件末尾
<code>cmd 2&gt;&gt; file</code>	将标准错误追加到文件末尾
<code>cmd &gt;&gt; file 2&gt;&amp;1</code>	同时追加 stdout 与 stderr
<code>cmd1   cmd2</code>	将 cmd1 的标准输出输入到 cmd2
<code>cmd1 2&gt;&amp;1   cmd2</code>	将 cmd1 的标准输出和标准错误输入到 cmd2
<code>cmd1   tee file1   cmd2</code>	将 cmd1 的标准输出写入 file1 的同时输入到 cmd2

⚠ Remark ▾

重定向由 **Shell 提供**，并非具体命令的特性。  
因此这些语法适用于所有标准 Unix 程序。

## 2.3 Standard Redirection (Pipes 管道)

管道 (pipe) 操作符 |

用于将一个命令的输出 (stdout) 作为下一个命令的输入 (stdin)。

### 2.3.1 示例 1：统计字符串单词数

```
Shell
1 $ echo "hey there" | wc -w
2 2
```

### 2.3.2 示例 2：大小写转换

```
Shell
1 $ echo "user1" | tr 'a-z' 'A-Z'
2 USER1
```

### 2.3.3 示例 3：提取数据文件第二列中的唯一条目数

```
Shell
1 $ cut -d',' -f2 cpds.csv | sort | uniq | wc
```

或保存结果到文件：

```
Shell
1 $ cut -d',' -f2 cpds.csv | sort | uniq > countries.txt
```

#### ⚠ Remark: 执行逻辑分解

1. `cut -d',' -f2 cpds.csv`
  - 提取以逗号分隔的第二列；
2. `sort`
  - 对输出进行排序；
3. `uniq`
  - 删除重复值（仅保留唯一条目）；
4. `wc`
  - 统计输出行数、单词数与字节数；
5. `>`
  - 将最终输出保存为文件。

#### 💡 Logic

许多 Unix 命令（如 `sort`，`grep`，`cut`，`wc`）  
若未指定文件名，会自动从 `stdin` 读取输入，  
这使得管道机制能自由组合命令链。

### 2.3.4 示例 4：大规模文件检测

查找 22,000 个文件（5GB 数据）中是否有字段值为 “S”：

```
Shell
1
```

```
1 $ cut -b29,37,45,53,61,69,77,85,93,101,109,117,125,133,141,149, \
2     157,165,173,181,189,197,205,213,221,229,237,245,253, \
3     261,269 USC*.dly | grep S | less
```

#### ⚠ Remark ▾

此命令仅用约 5 分钟即可完成，  
若用 R 或 Python 读入同样体量的数据则可能耗时数小时。

## | 2.4 The tee Command

tee 命令可 将一个流复制成两份：

- 一份传递到下一个命令；
- 另一份保存到文件。

### | 2.4.1 示例

传统方式（重复执行两次命令）：

```
>- Shell
1 $ cut -d',' -f2 cpds.csv | sort | uniq
2 $ cut -d',' -f2 cpds.csv | sort | uniq > countries.txt
```

更高效方式（使用 tee）：

```
>- Shell
1 $ cut -d',' -f2 cpds.csv | sort | uniq | tee countries.txt
```

输出效果：

- 在终端显示结果；
- 同时写入文件 countries.txt。

#### 💡 Logic ▾

tee 是数据分析中常用的辅助工具，  
特别适用于监控长时间运行的管道命令结果。

## | 3 Command Substitution and the xargs Command

### | 3.1 Command Substitution

Command substitution 允许我们把 一个 command 的 output 作为另一个 command 的 argument

其语法为：

```
>- Shell
1 $(command)
```

当 shell 遇到 \$( ) 包裹的命令时，它会：

1. 执行括号内的命令；
2. 将该命令的输出结果替换到当前位置。

这与 **管道 (|)** 相似，但适用于 **命令需要从命令行参数读取数据** 的情况，而不是从标准输入 (**stdin**) 读取。

### 3.1.1 Example

假设我们想在当前目录下的 **最新修改的四个 R 文件** 中查找文本 **pdf**：

Shell

```
1 $ grep pdf $(ls -t *.{R,r} | head -4)
```

执行逻辑：

- 1. `ls -t *.{R,r}` → 列出以 `.R` 或 `.r` 结尾的文件，按修改时间排序；
- 2. `head -4` → 取前四个文件名；
- 3. `$( ... )` → 将上一步输出作为 `grep` 的命令行参数；
- 4. Shell 最终执行：

Shell

```
1 grep pdf test.R run.R analysis.R process.R
```

⚠ Remark ▾

如果改用管道 `ls -t *.{R,r} | head -4 | grep pdf`，将 **不会** 达到同样效果，因为 `grep` 读取的文件名来自命令行参数，而非标准输入。

### 3.1.2 总结规律表

命令行为	数据来源	典型命令	举例
只从参数读	命令行参数（文件名、字符串等）	<code>ls</code> ， <code>rm</code> ， <code>cp</code> ， <code>echo</code>	<code>ls /home</code>
默认从 stdin 读（可选参数）	标准输入或文件参数	<code>cat</code> ， <code>grep</code> ， <code>sort</code> ， <code>wc</code> ， <code>awk</code> ， <code>cut</code>	<code>grep "a" file.txt</code> 或 <code>echo hi   grep "h"</code>
只从 stdin 读	标准输入流	一些专用工具或交互式程序	<code>cat</code> ， <code>tr</code> ， <code>head</code>
可从两者读	根据是否提供参数而定	<code>grep</code> ， <code>awk</code> ， <code>wc</code> ， <code>sort</code>	都可用

### 3.2 The **xargs** Command

管道无法直接将输出作为“命令行参数”传递，但可以使用 **xargs** 工具实现这一功能。

示例：

Shell

```
1 $ ls -t *.{R,r} | head -4 | xargs grep pdf
```

执行逻辑与上节等价于：

Shell

```
1 $ grep pdf $(ls -t *.{R,r} | head -4)
```

🔗 Logic ▾



- `xargs` 将标准输入 (stdin) 的内容拼接成命令行参数;
- 常用于解决“一个命令输出 → 另一个命令参数”类型的问题。

### 3.2.1 Exercise

请尝试以下命令，理解命令替换与 `xargs` 的区别：

```
Shell
1 $ ls -l tr # 若当前目录下没有 tr，则会报错
2 $ type -p tr # /usr/bin/tr
3 $ ls -l type -p tr # 报错
4 $ ls -l $(type -p tr) # -rwxr-xr-x 1 root root 43840 Jan 19 2024 /usr/bin/tr
```

提示：

- `type -p tr` 输出命令路径；
- `$(type -p tr)` 将路径替换进 `ls -l` 命令中。

## 4 Brace Expansion

### 4.1 Overview

**Brace expansion (花括号扩展)** 是 shell 的一种语法功能，用于自动生成多个字符串或文件名。Shell 在执行命令前，会先展开花括号的内容。

#### 4.1.1 Example 1: 文件重命名

```
Shell
1 $ mv my_long_filename.{txt,csv}
2 $ ls my_long_filename*
3 my_long_filename.csv
```

相当于：

```
Shell
1 mv my_long_filename.txt my_long_filename.csv
```

再例如：

```
Shell
1 $ mv my_long_filename.csv{,-old}
2 $ ls my_long_filename*
3 my_long_filename.csv-old
```

等价于：

```
Shell
1 mv my_long_filename.csv my_long_filename.csv-old
```

花括号展开在命令执行前由 shell 完成，  
生成的结果会被直接传递给命令。

### 4.1.2 Example 2: 使用序列展开

>-

Shell

```
1 $ echo {1..15}
2 $ echo c{c..e}
3 $ echo {d..a}
4 $ echo {1..5..2}
5 $ echo {z..a..-2}
```

说明：

- {1..15} → 从 1 到 15；
- {c..e} → c, d, e；
- {1..5..2} → 从 1 到 5，步长为 2；
- {z..a..-2} → 从 z 向 a 逆序，步长为 2。

可用于批量命令，例如结束多个连续进程：

>-

Shell

```
1 $ kill 1397{62..81}
```

## 5 Quoting

### 5.1 Single vs. Double Quotes

引号类型	名称	特性
' '	硬引用 (hard quote)	禁止变量替换
" "	软引用 (soft quote)	允许变量替换

#### 5.1.1 Example

>-

Shell

```
1 $ echo "My home directory is $HOME"
2 My home directory is /home/jarrold
3
4 $ echo 'My home directory is $HOME'
5 My home directory is $HOME
```

#### Logic

使用双引号 " " 可让变量在字符串中被解释，  
使用单引号 ' ' 则输出字面量。

### 5.2 Handling Spaces in Filenames

当路径或文件名中包含空格时，  
可用转义符或引号防止 shell 将空格误认为参数分隔符。

```
Shell
1 $ ls $HOME/with\ space
2 file1.txt
```

或更简洁的写法：

```
Shell
1 $ ls "$HOME/with space"
2 file1.txt
```

若使用硬引号（单引号）：

```
Shell
1 $ ls '$HOME/with space'
2 ls: cannot access $HOME/with space: No such file or directory
```

因为 `$HOME` 未被解析。

### 5.3 Escaping Double Quotes

若目录或文件名本身包含双引号 (`"with"quote`)：

```
Shell
1 $ ls "$HOME/\"with\"quote"
```

通过 `\` 转义内部双引号。

⚠ Remark ▾

- 尽量避免文件或目录名中出现空格与引号。
- 弯引号（curly quotes，如 “ ” 或 ‘ ’）在代码中无效，仅在字符串文本中可用。

## 6 Powerful Tools for Text Manipulation: grep, sed, and awk

### 6.1 Overview

在文本编辑器出现之前，Unix 用户依靠 **行编辑器（line editor）** 来修改文件。  
早期的编辑器 `ed` 仅在需要时显示特定行，而非整个文件。  
许多现代工具（如 `grep`，`sed`，`awk`，`vim`）都源自 `ed`：

工具	起源与特性
grep	源自 <code>ed</code> 的命令 <code>g/&lt;re&gt;/p</code> ，用于全局匹配正则表达式
sed	“stream editor”——基于流的 <code>ed</code> 版本，可批量处理文本
awk	更通用的文本处理语言，语法上受 <code>ed</code> 启发
vim	<code>vi</code> 的改进版，同样继承了 <code>ed</code> 的操作逻辑

优势：

- 逐行处理文件（line by line），内存占用低；
- 能高效处理大型文本文件（如日志、CSV、配置文件）；
- 可结合管道与正则表达式实现复杂操作。

## 6.2 grep

`grep` 是最常用的文本搜索命令。  
它用于打印文件中符合指定 **模式 (pattern)** 或 **正则表达式** 的行。

### 6.2.1 基本用法

假设我们有文件 `testfile.txt`：

```
1 This is the first line.
2 Followed by this line.
3 And then ...
```

#### 1. 查找包含某模式的行

```
>- Shell
1 $ grep is testfile.txt
2 This is the first line.
3 Followed by this line.
```

#### 2. 查找不包含该模式的行

```
>- Shell
1 $ grep -v is testfile.txt
2 And then ...
```

`-v` 表示反选（打印不匹配的行）。

#### 3. 只打印匹配内容

```
>- Shell
1 $ grep -o is testfile.txt
2 is
3 is
4 is
```

#### 4. 彩色高亮匹配结果

```
>- Shell
1 $ grep --color is testfile.txt
```

🔗 Logic ▾

`grep` 名称来自 `ed` 命令 `g/re/p`，即“globally search for a regular expression and print”。

## 6.3 sed

`sed` (**s**tream **e**ditor) 是一个强大的流式文本编辑器。  
它逐行读取输入，对匹配的行执行替换、删除或打印操作。  
默认输出结果到 **stdout**，除非使用 `-i` 参数修改文件本身。

### 6.3.1 打印特定行

Shell

```
1 $ sed -n '1,9p' file.txt      # 打印第 1-9 行
2 $ sed -n '/^#/p' file.txt     # 打印以 '#' 开头的行
```

- `-n`：抑制默认输出，仅输出匹配结果；
- `/^#/`：正则表达式，匹配行首为 `#` 的行。

### 6.3.2 删除特定行

Shell

```
1 $ sed -e '1,9d' file.txt
2 $ sed -e '/^;/d' -e '/^$/d' file.txt
```

- 第一行：删除第 1-9 行；
- 第二行：
  - `/^;/d` 删除以分号开头的行；
  - `/^$/d` 删除空行。

Logic

`-e` 用于同时执行多个表达式，  
若只需一个命令可省略。

### 6.3.3 文本替换

Shell

```
1 $ sed 's/old_pattern/new_pattern/' file.txt > new_file.txt
2 $ sed 's/old_pattern/new_pattern/g' file.txt > new_file.txt
3 $ sed -i 's/old_pattern/new_pattern/g' file.txt
```

选项	说明
无 <code>g</code>	每行只替换第一个匹配项
<code>g</code>	全局替换行内所有匹配项
<code>-i</code>	原地修改文件（谨慎使用）

Remark

使用 `-i` 时不会自动备份，  
建议先输出到新文件以防止数据丢失。

### 6.4 awk

`awk` 是一种专为文本与表格数据处理设计的 **轻量编程语言**。  
它按行处理文件，并根据条件执行操作。  
其语法结构为：

```
1 awk 'pattern { action }' file
```

### 6.4.1 Example 1: 选择列



Shell

```
1 $ ps -f | awk '{ print $2 }'
```

输出进程列表中的第 2 列 (PID)。

#### ⚠ Remark ▾

- `$1`, `$2`, ... → 对应第 1、2、... 列；
- `$0` → 表示整行。

### 6.4.2 Example 2: 文件双倍行距



Shell

```
1 $ awk '{ print } { print "" }' file.txt
```

- 第一个 `{ print }` 输出原行；
- 第二个 `{ print "" }` 输出空行。

### 6.4.3 Example 3: 筛选长行



Shell

```
1 $ awk 'length($0) > 80' file.txt
```

输出长度大于 80 字符的行。

### 6.4.4 Example 4: 提取用户主目录



Shell

```
1 $ awk -F: '{ print $6 }' /etc/passwd
```

解析说明：

- `-F:` → 设置分隔符为冒号 `:`；
- `$6` → 第 6 个字段，对应用户主目录。

查看文件格式：



Shell

```
1 $ head -n 1 /etc/passwd
2 root:x:0:0:root:/root:/bin/bash
```

结果：

```
1 /root
```

### 6.4.5 Example 5: 列求和



Shell

```
1 $ awk '{ print $1 + $2 }' file.txt
```

输出文件中第 1 列与第 2 列的和。

#### Logic ▾

`awk` 既可处理文本，也可执行算术操作，是轻量数据分析和日志提取的理想工具。

## 6.5 Summary

工具	功能	特点
<code>grep</code>	搜索匹配行	支持正则表达式，快速定位文本
<code>sed</code>	编辑文本流	支持替换、删除、打印、批量修改
<code>awk</code>	结构化文本处理	支持条件判断、字段操作与计算

#### Logic ▾

这三者常搭配使用：

- `grep` 用于过滤；
- `sed` 用于编辑；
- `awk` 用于分析。

## 7 Aliases (Command Shortcuts) and `.bashrc`

### 7.1 What Are Aliases?

**Aliases** 是 Bash 提供的命令快捷方式，用于：

- 简化常用命令（缩写）；
- 为现有命令设置默认选项；
- 自定义命令行为。

基本语法：

```
>_ Shell
1 $ alias name='command'
```

示例 1：创建退出命令的快捷方式

```
>_ Shell
1 $ alias q=exit
```

现在输入 `q` 就等同于输入 `exit`。

示例 2：让 `ls` 命令始终显示文件类型标记

```
>_ Shell
1 $ alias ls="ls -F"
```

此时：

- 目录会显示 `/`
- 可执行文件显示 `*`
- 链接显示 `@`

如果需要临时使用未被 `alias` 修改的原始命令，可使用反斜杠：

```
>- Shell
1 $ \ls
```

#### Logic

Alias 的本质是 **命令替换 (text substitution)**，  
当你输入命令时，Bash 会先将别名替换成定义的完整命令。

## 7.2 Making Aliases Permanent

命令行直接设置的别名在关闭终端后会失效。

若要 **自动加载 alias 设置**，需将定义写入：

```
>- Shell
1 ~/.bashrc
```

该文件会在每次打开新的 Bash 会话时自动执行。

## 7.3 Example: A Typical `.bashrc` Configuration

```
>- Shell
1 # .bashrc
2
3 # 载入全局设置
4 if [ -f /etc/bashrc ]; then
5     . /etc/bashrc
6 fi
7
8 # 用户自定义函数
9 pushdp () {
10     pushd "$(python -c "import os.path as _, ${1}; \
11         print _.dirname(_.realpath(${1}.__file__[:-1]))")"
12 }
13
14 # 默认编辑器
15 export EDITOR=vim
16
17 # Git 命令提示增强
18 source /usr/share/git-core/contrib/completion/git-prompt.sh
19 export PS1='[\u@\h \W$__git_ps1 " (%s)"]\>'
20
21 # 历史记录设置
22 export HISTCONTROL=ignoredups # 不记录重复命令
23 shopt -s histappend          # 追加记录而非覆盖
24
25 # R 环境设置
26 export R_LIBS=$HOME/usr/lib64/R/library
27 alias R="/usr/bin/R --quiet --no-save"
```



```
28
29 # 路径设置
30 mybin=$HOME/usr/bin
31 export PATH=$mybin:$HOME/.local/bin:$HOME/usr/local/bin:$PATH
32 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/usr/local/lib
33
34 # 常用别名
35 alias grep='grep --color=auto'
36 alias hgrep='history | grep'
37 alias l.='ls -d .* --color=auto'
38 alias ll='ls -l --color=auto'
39 alias ls='ls --color=auto'
40 alias more=less
41 alias vi=vim
```

## 7.4 Explanation of Key Sections

区块	功能说明
Global Settings	检查并载入系统级配置 <code>/etc/bashrc</code>
Custom Functions	定义用户自用函数（如 <code>pushdp</code> ）
Environment Variables	设置编辑器、历史记录行为、R 库路径等
PATH & LD_LIBRARY_PATH	添加自定义执行与库文件路径
Aliases	统一格式输出、颜色显示、常用命令缩写等

### ⚠ Remark ▾

修改 `.bashrc` 后需执行以下命令以立即生效：



Shell

```
1 $ source ~/.bashrc
```

或重新打开一个新的终端窗口。

### 💡 Logic ▾

`.bashrc` 不仅可以存放 `alias`，还可定义函数、变量、命令提示符样式、路径设置等。  
熟练运用 `.bashrc` 是打造个性化 Shell 环境的关键。