像黑客一样使用命令行

徐小东

献给

海燕和铭基

目录

表格

插图

致谢

感谢 Bash 及 Zsh 开源社区，你们永远是最棒的家伙！

更新

你可以从 https://selfhostedserver.com/usingcli-book 获取本书的更新版本。另外，本书也包括视频版本，请通过 https://selfhostedserver.com/usingcli 了解详情。

- Version 2019.3.17：初版
- Version 2019.8.24：修订版，增加 “高效查询 Shell 历史：HSTR” 小节。

作者简介

徐小东，网名 ~toy。GNU/Linux 爱好者，DevOps 践行者。喜技术，好分享。通过 https://linuxtoy.org 网站数年间原创及翻译文章达 3000 余篇。另著有《像黑客一样使用命令行》[1]、《容器化工具三剑客：Podman、Buildah 和 Skopeo》[2]、《Terraform：自动化管理云基础设施》[3]，译有《笨办法学 Git》[4]、《Perl 程序员应该知道的事》等图书。Twitter：https://twitter.com/linuxtoy，Mail：xuxiaodong@pm.me[5]。

[1] https://selfhostedserver.com/usingcli-book

[2] https://selfhostedserver.com/nextcontainer

[3] https://selfhostedserver.com/terraform

[4] https://selfhostedserver.com/learngit

[5] mailto:xuxiaodong@pm.me

第一章　入门指引

虽然如今计算机图形化界面大行其道，然而在计算机诞生之初却是命令行界面的天下。在图形化界面中，我们惯常使用鼠标来操作图标或窗口，从而完成各种任务。对于命令行界面来说，情况则有很大的不同。要在命令行界面下执行操作，我们需要更多的依赖键盘。那么，什么是命令行界面呢？在回答这个问题之前，不妨让我们先来谈谈控制台、终端、终端模拟器、以及 Shell 这几个基本概念。

1.1　控制台

控制台（Console），又称为系统控制台（System console）、计算机控制台（Computer console）、根控制台（Root console）、以及操作员控制台（Operator's console）。事实上，早先的控制台是一种用来操作计算机的硬件，如图 **??** 所示。[1]从这幅图片中，我们可以看到 IBM 1620 计算机的控制台由左边的操作前面板（参考图 **??**）和右边的打字机组成。通过控制台，操作员将文本数据或待执行的指令录入到计算机，并最终通过计算机读取或执行。

图 1.1: IBM 1620 的控制台

图 1.2: IBM 1620 控制台的操作前面板

随着计算机的发展，控制台从硬件概念变成了一个软件概念。于是，控制台有了新的称呼：虚拟控制台。虚拟控制台正好与物理的控制台硬件相对。通过观

[1] https://en.wikipedia.org/wiki/System_console#/media/File:IBM_1620_Model_1.jpg

察 Linux 系统的启动过程，我们不难发现：在经过计算机硬件自检之后，一旦由引导载入程序接管，不一会儿便会进入系统控制台。在这个过程中，通常会显示如图 ?? 所示的 Linux 系统引导信息。[2]

图 1.3: Linux 系统虚拟控制台

1.2 终端

跟控制台一样，起初的终端（Terminal）也是一种计算机硬件设备。从外形上看，终端类似于我们今天所看到的显示器和键盘的结合体。通过终端，用户将指令和数据输入到计算机。同时，终端也将计算机执行的结果展示给用户。图 ?? 中显示的是曾经广为流行的终端 DEC VT100。[3]

图 1.4: DEC VT100 终端

或许你会产生疑问，为什么会出现终端这种硬件设备呢？以今天的眼光来看，显得似乎有些难以理解。诞生之初的计算机造价相当昂贵，可不像现在人人都能拥有一台那么简单。除了大型商业组织或大学研究机构，很难在别处看到计算机的身影。为了能够共享计算机资源，终端应运而生。然而，伴随着科技的进步，终端最终掉进了历史的黑洞。不过，它后来却以新的形式重生，这就是终端模拟器（Terminal emulator），或称之为虚拟终端。

1.3 终端模拟器

终端模拟器，即用来模拟终端硬件设备的应用程序。在物理终端中存在的某些显示体系结构，比如用来控制色彩的转义序列、光标位置等在终端模拟器中也得到了支持。图 ?? 显示 Linux 中流行的终端程序之一 XTerm。

图 1.5: XTerm 终端模拟器

[2] https://en.wikipedia.org/wiki/Linux_console#/media/File:Knoppix-3.8-boot.png

[3] https://en.wikipedia.org/wiki/Computer_terminal#/media/File:DEC_VT100_terminal.jpg

不管是 Linux 操作系统，还是 macOS 操作系统，乃至 Windows 操作系统，今天都有许多终端模拟器可以选择。以下罗列的是这三个操作系统中比较流行的终端模拟器。

1.3.1 Linux

- XTerm[4]：XTerm 是 X 窗口环境的默认终端。它提供了与 DEC VT102 和 Tektronix 4014 终端兼容的特性。此外，它也支持 ISO/ANSI 彩色模式。
- GNOME Terminal[5]：GNOME Terminal 是 GNOME 桌面环境的默认终端。它提供了与 XTerm 相似的特性。除此之外，它也包括支持多配置、标签页、鼠标事件等其它功能。
- Konsole[6]：Konsole 是 KDE 桌面环境的默认终端。它包括标签页、多配置、书签支持、搜索等特性。
- rxvt-unicode[7]：rxvt-unicode 原本克隆自 rxvt，但加入了 unicode 支持，具有很强的定制特性。另外，rxvt-unicode 还包含 Daemon 模式、嵌入了 Perl 编程语言等功能。本书作者使用的就是这款终端模拟器。

1.3.2 macOS

- Terminal.app：Terminal.app 是 macOS 操作系统默认的终端。它的功能不多，除了提供设置 `TERM` 环境变量的选项外，还包括能够使用其搜索功能来查找 Man pages。
- iTerm2[8]：iTerm2 是 macOS 系统上针对默认终端的开源替代品。它非常流行，包含许多很棒的功能，比如窗口分割、自动补全、无鼠拷贝、粘贴历史等等。如果你在 macOS 上工作，那么不妨使用 iTerm2 这款终端模拟器，相信它所具有的功能一定不会让你失望。

[4] https://invisible-island.net/xterm/
[5] https://gitlab.gnome.org/GNOME/gnome-terminal/
[6] https://kde.org/applications/system/konsole/
[7] http://software.schmorp.de/pkg/rxvt-unicode.html
[8] https://www.iterm2.com/

1.3.3 Windows

- Mintty[9]：Mintty 是一个支持 Cygwin、MSYS、WSL 等多种环境的终端模拟器。它的功能与 XTerm 兼容，包括 256 色和真彩色、unicode、以及 Emoji 表情支持。
- ConEmu[10]：ConEmu 是 Windows 上一款相当流行的开源终端模拟器。它包含标签页、多种图形窗口模式、用户友好的文本块选择等功能。

1.4 Shell

Shell 是一种命令解释程序，它负责用户输入命令的读取、解析和执行。现代 Shell 除了具有与用户直接交互的特性之外，通常也包含编程功能，支持变量、数组、函数、循环、条件等编程基本要素。

Shell 之所以如此称呼，是由于它相对 Unix 及 Linux 的核心——内核（Kernel）而言，处于整个操作系统的最外层，就像乌龟的壳一样。也正因为如此，Shell 提供用来访问系统服务的用户界面，扮演着与内核交互的角色，如图 ?? 所示。

图 1.6: Shell 与内核

在 Unix 及 Linux 的发展过程中，出现了许多种 Shell，其中比较知名的包括：sh、csh、ksh、bash、zsh 等等。

1.4.1 sh

sh，即 Bourne shell，它是 Unix 第 7 版的默认 Shell。Bourne shell 由贝尔实验室的 Stephen Bourne 开发，于 1979 年发布。随着《Unix 编程环境》（Brian Kernighan 与 Rob Pike 著）一书的出版，sh 变得大为流行。

Bourne shell 早已被后来的 Shell 所取代，现代 Linux 系统中的 sh 通常是符号链接的某个兼容 Shell。例如，本书作者所用的 Debian 9 里的 sh 为 dash。

[9] https://mintty.github.io/
[10] https://conemu.github.io/

```
root@toydroid:~# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jan 24  2017 /bin/sh -> dash
```

而在作者的另一个系统 Arch Linux 上，sh 则为 bash。

```
root@codeland:~# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Feb  7 15:15 /bin/sh -> bash
```

1.4.2 csh

csh 是 C shell 的简称，它由 Bill Joy 开发，通过 BSD 得到了广泛的分发。在风格上，开发者将 csh 设计得像 C 编程语言一样，因而由此得名。同时，csh 具有很不错的交互使用体验。后来被其它 Shell 所吸收的诸如历史、别名、目录栈、文件名补全、作业控制等特性均出自 csh。

csh 有一个改进版本叫 tcsh，目前是 FreeBSD 的默认 Shell。

1.4.3 ksh

ksh 指 Korn Shell，其开发者为 David Korn，在 1983 年公布于世。ksh 遵循 POSIX 标准，能够向下兼容 Bourne shell，整合了来自 C shell 的诸多特性。ksh 的一大亮点是引入了 vi 和 Emacs 风格的命令行编辑模式，使用户完全可以按照自己的按键习惯操作。此外，在 ksh 中还增加了关联数组的特性。

由于 ksh 最初以私有软件的形式进行分发，从而被限制了传播。代之以出现的替代品包括 pdksh（public domain ksh，公有域的 ksh）、mksh（后成为 Android 的默认 Shell）等。

1.4.4 bash

bash 作为理查德 • 斯托曼 GNU 工程的一部分出现，从它诞生之初就是为了用来取代 Bourne shell（参考 **??** 节）。Brian Fox 开发了最初的 bash，首个版本发布于 1989 年。如今，bash 已变得十分流行，它是大多数 Linux 发行版以及

macOS 的默认 Shell。此外，通过 WSL（Windows Subsystem for Linux），在 Windows 10 中也可以安装并使用 bash。

bash 的名称来自于 **B**ourne-**a**gain **sh**ell，它也遵循 POSIX 标准，其特性吸收自 sh、csh、ksh 等多种 Shell。

1.4.5 zsh

zsh 是 Z shell 的简称，最初的版本由 Paul Falstad 所开发，发布于 1990 年。zsh 极大的扩展了 Bourne shell 的功能，并包含来自 tcsh、ksh、bash 等 Shell 的特性。

在交互用户体验上，zsh 尤其出彩。比如，它支持对命令的选项进行补全、可以设置右提示符等，如图 **??** 所示。

图 1.7: zsh 的右提示符

本书主要讨论 bash 和 zsh 这两种目前市面上最流行的 Shell。

1.5 命令行界面

命令行界面（Command-line interface），经常缩写为 CLI，亦即用户输入命令的地方。一旦用户将命令输入完毕并加以提交后，后续对命令的解析以及执行的任务都由 Shell 来完成。

与 CLI 相对的是 GUI，即 Graphical user interface，意为图形用户界面，它采用图形化的方式让用户与计算机进行交互。因其具有容易使用的优点，包括 Linux、macOS、Windows 等在内的现代操作系统无一例外都提供了图形用户界面。

既然图形用户界面要比命令行界面更加易用，那么是否说明可以完全抛弃命令行界面呢？答案是并非如此。事实上，有经验的用户尤其擅长使用命令行界面，其理由至少包括以下几个方面。

1.5.1 功能强大

让我们先来看一个例子：

```
xiaodong@codeland:~$ history |
awk '{CMD[$2]++;count++;}END \
{ for (a in CMD)print CMD[a] " " \
CMD[a]/count*100 "% " a;}' |
grep -v "./" |
column -c3 -s " " -t |
sort -nr |
nl |
head -n10
```

在作者的 macOS 系统上执行这条命令后，其输出结果如下：

```
 1  1348  14.3771%  cd
 2  1034  11.0282%  l
 3  838   8.93771%  git
 4  569   6.06869%  ssh
 5  513   5.47142%  cat
 6  405   4.31954%  vim
 7  372   3.96758%  brew
 8  360   3.83959%  scp
 9  265   2.82637%  rm
10  264   2.8157%   grep
```

这条命令虽然看起来似乎有些“吓人”，因为它由 history、awk、grep、column、sort、nl、head 等 7 个命令组成，并通过管道符（|）串接在一起；然而其结果却颇为有趣。它将作者平时在命令行中执行的所有命令都进行了统计，最终展示出 10 个最常用的命令，并相应列出每个命令的使用次数和所占百分比。

管道符将前一命令的输出作为后一命令的输入，使这些表面上不相干的命令进行协同工作，犹如搭积木一般。这是命令行的真正威力所在。

1.5.2 灵活高效

再看另一个例子，假如我们打算从 photos 目录中找出今年三月份拍摄的照片，并将其文件名称保存到 mar_photos.txt 这个文本文件中。在图形用户界面中，首先，我们可能会打开一个文件管理器（在 Linux 下也许是 GNOME Files，macOS 中则是 Finder）。接着，导航到 photos 这个目录，同时切换成详细视图模式。然后，我们睁大双眼逐一找出符合要求的照片。可是，现在怎么把照片的文件名称写到文本文件中呢？我们当然可以直接输入，或者想省点力使用复制和粘贴也行。要是找出的文件数量比较多，那可绝对是体力活。

但是，如果在命令行下，那么我们只需通过执行一条命令即可达到目的：

```
xiaodong@codeland:~$ cd photos; \
ls -l | grep 'Mar' | awk '{ print $9 }' > mar_photos.txt
```

1.5.3 能自动化

使用命令行还有一个很棒的优势，那就是能够自动化各种操作。Shell 允许我们将所用的命令编写成函数（Function）或脚本（Script）。这样，我们不仅可以反复执行它们，而且函数或脚本比手动输入效率更高。由此，我们得以从重复的劳动中解放出来，从而能够腾出时间去做其它有意义的事情。

```
xiaodong@codeland:~$ ./script.sh
```

1.6 如何进入命令行

通过前面的描述，现在你应当了解：我们想要输入命令的界面是由 Shell 提供的。那么，如何执行 Shell 呢？我们可以通过下面两种方法来进入命令行。

1.6.1 通过控制台进入命令行

为了节省系统资源，Linux 服务器通常没有附带图形用户界面。当它启动完毕时，在控制台按照提示输入用户帐号及密码并登录后，所进入的即是命令行界面。以下为 Linux 服务器的登录提示：

```
login:
Password:
```

作为普通用户来说，一般使用的是具有图形用户界面的 Linux 桌面系统。在它启动后就直接进入了桌面，那么此时想要进入控制台，可以按照下列步骤执行：

1. 按 Ctrl + Alt + F1 组合键，进入编号为 1 的控制台。
2. 按 Ctrl + Alt + F2 组合键，进入编号为 2 的控制台。
3. 依次类推，可以分别进入 3 号、4 号、5 号、以及 6 号控制台。在默认情况下，Linux 一般提供 6 个控制台。
4. 如果要从控制台返回到桌面，则可以按 Ctrl + Alt + F7 组合键。

1.6.2 通过终端模拟器进入命令行

另外一种进入命令行界面的方法是使用终端模拟器。在不同的操作系统中，可以选择的终端模拟器程序也有所不同（参考 **??** 节）。本书作者在 Linux 下常用 rxvt-unicode，macOS 中则使用 iTerm2。

一般而言，终端模拟器程序会跟系统的登录 Shell（或称默认 Shell）绑定在一起。有些终端模拟器程序提供了更改 Shell 的特性，从而使用户可以方便的选择自己惯用的 Shell。如果不能从终端程序中直接更改 Shell，那么也可以通过 `chsh` 命令来改变登录 Shell。假如我们想把默认 Shell 更改成 zsh，则可以执行以下命令：

```
xiaodong@codeland:~$ chsh -s /bin/zsh
```

1.7 你好，命令行

在《C 程序设计语言》中，作者 Brian W. Kernighan 和 Dennis M. Ritchie 介绍的第一个程序是在屏幕上输出一行 “Hello world” 的消息。为了说明命令行的使用，我们也将在屏幕上输出类似的消息——“你好，命令行”。

当我们进入控制台或打开终端模拟器时，通常会看到跟图 **??** 相似的命令行界面。

图 1.8: 命令行界面

从图 **??** 中我们可以看到命令行一般由下面几个部分组成：

1. 当前登录的用户名称，在本例中是 `xiaodong`。
2. `codeland` 是主机名称，跟 `hostname -s` 的输出一致。
3. 当前工作目录，`~` 代表用户的主目录，在 Linux 系统下也就是 `/home/<用户名>`，macOS 中则为 `/Users/<用户名>`。
4. `$` 为命令提示符。通常普通用户的命令行提示符与超级用户（root）的不同，以 bash 为例，root 用户的命令行提示符为 `#`。
5. 待执行的命令，在本例中是 `echo -e "\t 你好，命令行"`，除 `echo` 命令本身外，还包括该命令的选项（`-e`）以及参数（`\t 你好，命令行`）等部分。命令的选项参数一般由引号（`"`）引起，以避免诸如空格之类的特殊字符所导致的岐义。可以使用单引号（`'`）或双引号（`"`），但语意会不同。

除了这 5 个部分之外，在这个命令行中，我们还可以看到 `@`、`:`、以及 `` ` ``（空格）等字符。`@` 一般用来分隔用户名和主机名，其形式跟电子邮箱地址一样。`:` 在这里起到提示说明作用。空格则常常用来分隔命令的选项和参数。因为命令行提示符可以定制，所以你的命令行界面可能跟我们在这里介绍的不同。

现在，请你跟我们一起，在命令行的提示符（`$` 或 `#`）后面输入 `echo -e "\t 你好，命令行"`。如果在输入过程中有错误，不必慌张，按**退格键**（BackSpace）或**删除键**（Delete）删除后重新输入即可。当所有字符全部输入完成后，按下**回车键**（Enter）。

发现了什么？命令行向我们回显了一条“你好，命令行”的消息。而且 `echo` 命令参数中的 `\t` 在输出中产生了一个制表符（Tab），从而让消息有了缩进效果。

```
xiaodong@codeland:~$ echo -e "\t 你好，命令行"
        你好，命令行
```

恭喜！你刚刚在命令行成功执行了一条命令，是否感觉并没有想象中那么恐怖呢？在后面的章节中，我们将教你如何更加高效的使用命令行，从而提升你的工作效率。

第二章　神奇补全

如果你编写过代码，那么一定听说过 “代码补全” 吧。在如今流行的代码编辑器和 IDE (集成开发环境) 中，这绝对是一项深受大家喜欢的功能。在此我要讲的 Shell 补全与它很相似。我相信，在学习了本章所讲的内容后，你肯定会爱上它。首先，我们会谈谈什么叫自动补全，然后看看如何触发自动补全，接着详细探讨诸如文件名或路径名、程序名或命令名、用户名、主机名、以及变量名等各种自动补全类型，最后再介绍可编程补全。

2.1　何谓补全

现在回过头来看，在学习命令行时，我最想率先学习的功能一定是自动补全。为什么这么说呢？因为自动补全这项功能让我们只需输入开头的一个或几个字符便能通过 Shell 自动补全剩下的内容。对于痛恨输入长命令或文件名的朋友而言，自动补全绝对是福音。自动补全不仅减少了输入，而且节省了时间，从而极大的提高了我们的操作效率。

让我们通过一个例子来说明何谓自动补全。首先，我通过在 bash 中直接输入完整的命令行

```
xiaodong@codeland:~$ ls -l reallylongname.txt
```

来查看 `reallylongname.txt` 这个文本文件的信息。

然后，我在输入

```
xiaodong@codeland:~$ ls -l r
```

之后按 **Tab 键**，于是 bash 帮我自动补全了该文件名剩下的部分。

```
xiaodong@codeland:~$ ls -l reallylongname.txt
```

比较两次输入，bash 帮我少输了 17 个字符。是不是感觉很爽呢？

再看一个例子：这次，我在输入

```
xiaodong@codeland:~$ ls -l f
```

后按 **Tab**，bash 自动补全了 `file`。

```
xiaodong@codeland:~$ ls -l file
```

接着，我连按两下 **Tab**，这时 bash 向我们展示了可以自动补全的文件名列表，总共包括 5 个项目。

```
xiaodong@codeland:~$ ls -l file
file1  file2  file3  file4  file5
```

我输入 `1` 来完成 bash 自动补全过程。

比较这两个例子，我们可以发现，如果我们输入的开头字符唯一，那么 bash 将直接自动补全余下的内容。反之，则提供一个可供补全的备选列表。不过，这时候需要我们连按两下 **Tab 键**。这样的话，经常操作起来感觉还是有点麻烦。

下面我们对 bash 自动补全的配置进行一番优化，使之更加好用。利用文本编辑器打开 `~/.inputrc` 文件 (若不存在，则创建一个)，加入下列内容：

```
# completion
set show-all-if-ambiguous on
set visible-stats on
set colored-completion-prefix on
```

其中，开启 `show-all-if-ambiguous` 这个选项后，我们只需按一次 **Tab** 即可看到备选补全列表；`visible-stats` 选项通过在列表项目尾部添加指示符号来说明类型，例如：`@` 代表符号链接、`/` 代表目录等；最后的 `colored-completion-prefix` 选项则给补全的前缀字符加上颜色。如图 **??** 所示。

图 2.1: bash 自动补全配置结果

2.2 补全触发按键

通过这些例子，我们也可以知道，要触发自动补全，一般只要按 **Tab 键**即可。bash 和 zsh 都是这个默认设定。

2.3 文件名、路径名补全

前面的例子显示的是在 bash 中文件名自动补全的情况。下面我们看一个在 zsh 中自动补全文件名的例子。当我输入

```
xiaodong@codeland:~$ ls -l f
```

后按 **Tab**，zsh 为我自动补全了 `file`。

```
xiaodong@codeland:~$ ls -l file
```

我接着再按 **Tab 键**，这时 zsh 提供了可以备选的自动补全菜单。

```
xiaodong@codeland:~$ ls -l file
file1  file2  file3  file4  file5
```

再次按 **Tab** 则可以选择具体的菜单项目。

```
xiaodong@codeland:~$ ls -l file2
file1  **file2**  file3  file4  file5
```

然后按**回车键**完成自动补全过程。

```
xiaodong@codeland:~$ ls -l file2
```

最后再按一次**回车键**执行命令。

不知大家有没有发现与 bash 补全的区别呢？bash 提供的备选补全列表不能选择具体的项目，而 zsh 则可以。这也说明与 bash 相比，zsh 具有更棒的用户交互功能。所以我平常也更喜欢使用 zsh 一些。如果你还没有用过 zsh，那么我在此建议你一定要试一试。

说到备选补全列表，在 bash 中我喜欢使用的一组快捷键是 **Alt + ?**。当 bash 补全 `file` 后，我没有按 **Tab**，而是按 **Alt + ?**，bash 就立即呈现了备选补全列表。

```
xiaodong@codeland:~$ ls -l file
file1  file2  file3  file4  file5
```

还有一种情况，有时候我们希望 Shell 不要补全某些特别的文件类型。为了达到这种效果，我们可以使用 `FIGNORE` 变量。在下面的例子中，我想查看 `Welcome.java` 的内容，因此只想 Shell 补全 `.java` 文件，并排除 `.class` 文件。

```
xiaodong@codeland:~$ cat W
Welcome.class  Welcome.java
xiaodong@codeland:~$ cat Welcome.
```

在将 `.class` 扩展名赋给 `FIGNORE` 变量后，Shell 就为我剔除掉了 `.class` 文件类型。

```
xiaodong@codeland:~$ FIGNORE='.class'
xiaodong@codeland:~$ cat W<Tab>
xiaodong@codeland:~$ cat Welcome.java
```

如果想要排除多种文件类型，则只需用 : (冒号) 分隔即可。例如：

```
xiaodong@codeland:~$ FIGNORE='.o:.class'
```

这将让 Shell 在自动补全时排除 `.o` 和 `.class` 文件。无论是 bash，还是 zsh，当前都支持 `FIGNORE`。

路径名补全和文件名补全很相似，只是在补全后自动追加上 `/` (斜杠)，便于我们输入下一级的路径名。在下例中，我在输入

```
xiaodong@codeland:~$ cd g
```

后按 **Tab**，Shell 补完了全名，并在其后添加了一个 `/`。

```
xiaodong@codeland:~$ cd guessing_game/
```

之后我接着输入 `s` 并按 **Tab**，这次 Shell 补全了下级目录 `src`。

```
xiaodong@codeland:~$ cd guessing_game/src/
```

对于开头字符不唯一的情况，跟文件名补全也是一样，bash 中只要按 **Tab** 即可看到备选补全列表。

```
xiaodong@codeland:~$ cd h
hello/       hello_world/
xiaodong@codeland:~$ cd hello
```

顺便提一句，自动补全不光在命令行下使用，即便在有些图形化程序中也能使用。比如，在 GIMP 中，我也可以通过自动补全来打开文件。如图 **??** 所示。

图 2.2: 在 GIMP 中自动补全文件名

2.4 程序名、命令名补全

不带选项的程序名、命令名补全几乎跟文件名补全一样，让我们来看一个例子。要是你看过《黑客帝国》这部电影，那么下面的画面你应该会很熟悉。当我输入

```
xiaodong@codeland:~$ cmat
```

后按 **Tab**，bash 立即为我自动补全了 `cmatrix` 命令。

```
xiaodong@codeland:~$ cmatrix
```

而当我在输入

```
xiaodong@codeland:~$ cma
```

后按 **Tab**，bash 为我提供了一个备选补全列表。

```
xiaodong@codeland:~$ cma
cmake         cmake-gui     cmapcube      cmark         cmark-gfm     cmatrix
```

此时，需要再输入 `t` 才能完成补全。

```
xiaodong@codeland:~$ cmat
```

如果我在仅仅输入 `c` (命令开头的第一个字符)

```
xiaodong@codeland:~$ c
```

后便按 **Tab**，这时 bash 询问我：“Display all 474 possibilities? (y or n)” (是否显示所有 474 个补全列表项目) 按 **y** 予以显示。按 **n** 则不显示。

```
xiaodong@codeland:~$ c
Display all 474 possibilities? (y or n)
```

因为可供自动补全的列表项目太多，一屏已经显示不下了，所以 bash 使用 `more` 这个页面查看程序来呈现。现在按 **Space (空格键)** 可以翻页，如果想退出，那么按 **q** 即可。如图 **??** 所示。

图 2.3: 命令自动补全备选列表

除了直接补全命令名之外，Shell 也能自动补全程序的子命令，例如：`git status` 中的 `status` 以及命令的选项。不过，bash 需要安装一个单独的 `bash-completion` 包；而 zsh 因为内置了对此功能的支持，所以不需要额外的包。

bash-completion 的源代码[1]位于 GitHub 上，在此可以了解如何对其安装和配置。例如：

在 Debian 中，我们可以通过执行

```
xiaodong@codeland:~# apt install bash-completion
```

来安装它。

在 CentOS 上，除了安装 `bash-completion` 外，我推荐把 `bash-completion-extras` 也装上。

```
xiaodong@codeland:~# yum install bash-completion bash-completion-extras
```

而在 Arch Linux 上，则可以执行

```
xiaodong@codeland:~# pacman -S bash-completion
```

进行安装。

[1] https://github.com/scop/bash-completion

要配置 bash-completion，则只需要将下面这行指令加入 ~/.bashrc (个人) 或 /etc/bash.bashrc (全局) 即可。

```
[ -r /usr/share/bash-completion/bash_completion ] \
&& . /usr/share/bash-completion/bash_completion
```

在正常使用 zsh 的命令补全功能之前，我们也需要将下列内容加入到 ~/.zshrc 配置文件中：

```
# completion
autoload -U compinit
compinit -i
```

让我们先来看一个自动补全命令选项的例子。我在输入

```
xiaodong@codeland:~$ find -
```

后便立即按 **Tab**，bash 马上列出了可以自动补全的选项列表。如图 **??** 所示。

图 2.4: 命令选项自动补全备选列表

我接着输入 ina

```
xiaodong@codeland:~$ find -ina
```

并再次按下 **Tab**，此时 bash 自动补全了 -iname 选项。

```
xiaodong@codeland:~$ find -iname
```

下面的例子演示了 bash 补全子命令的情形。在输入

```
xiaodong@codeland:~$ git in
```

后，按 **Tab**，bash 提供可以自动补全的子命令列表。

```
xiaodong@codeland:~$ git in
info       init       instaweb
```

跟着输入 i

```
xiaodong@codeland:~$ git ini
```

并再按 **Tab**，这次 bash 便自动补全了子命令 init。对于执行 git status 子命令的过程同样如此。

```
xiaodong@codeland:~$ git init
```

与 bash 比较而言，zsh 对于命令选项的补全提供更好的用户体验。在下面的例子中，你将看到，zsh 不仅列出了可供补全的选项列表，更有对该选项用途的解释。如图 **??** 所示。此外，正如前面提到的，你还可以选择这些列表项目。

图 2.5: zsh 中的命令选项自动补全

对于子命令的补全，zsh 提供与命令选项补全相同的效果。

此外，子命令以及选项补全也可以合用。在下例中，我先补全了子命令 git status，然后又补全了选项 --verbose。

```
xiaodong@codeland:~$ git sta
stash  -- stash away changes to dirty working directory
status -- show working-tree status
xiaodong@codeland:~$ git status --v
xiaodong@codeland:~$ git status --verbose
```

2.4.1 Zsh 自动建议插件

对于使用 zsh 的朋友，我在此推荐一个好用的命令自动建议插件。这个插件叫做 zsh-autosuggestions[2]。针对命令进行自动建议这项功能源自于 `fish` shell，现在，zsh 从其借鉴过来，使得我们这些 zsh 的忠实拥趸也能使用这项好功能。

zsh-autosuggestions 的安装很简单，只需从 GitHub 将其克隆到本机，然后在 `.zshrc` 中引用 `zsh-autosuggestions.zsh` 并重新打开终端即可。

```
xiaodong@codeland:~$ git clone \
https://github.com/zsh-users/zsh-autosuggestions.git \
~/.zsh-autosuggestions
xiaodong@codeland:~$ echo source \
~/.zsh-autosuggestions/zsh-autosuggestions.zsh \
>> ~/.zshrc
xiaodong@codeland:~$ source ~/.zshrc
```

下面让我们来看看 `zsh-autosuggestions` 的用法，首先我在没有开启 `zsh-autosuggestions` 插件的情况下输入 `ls -l`、`cd hello_world` 等命令，除了能够使用命令补全之外，这儿没有命令的自动建议。当 `zsh-autosuggestions` 插件开启后，我一旦输入 `ls`，其后便会出现灰色的自动建议 `-la`。这是因为 zsh 知道我先前曾输入过 `ls -la` 这条命令，所以它给出了自动建议。如图 **??** 所示。

图 2.6: zsh 中的命令自动建议

这时，我们有两种选择：一是按 → **(右方向箭)** 接受建议，二是继续输入新的内容，这样也就放弃建议了。对于输入 `cd h` 后，zsh 同样给出了自动建议 `ello_world`。

[2] https://github.com/zsh-users/zsh-autosuggestions

2.5 用户名、主机名及变量名补全

除了常见的文件名、命令名补全外，Shell 自动补全还支持其它补全类型。这充分展现了 Shell 自动补全多才多艺的一面。下面我们就来看一看 Shell 如何自动补全用户名。

当我输入

```
xiaodong@codeland:~$ ls ~
```

之后按 **Tab**，此时 bash 为我呈现了系统中存在的用户名列表。如图 **??** 所示。

图 2.7: bash 中的用户名自动补全备选列表

我继续输入 `x` 并再次按 **Tab**，于是 bash 补全了 `xiaodong` 这个用户名。

```
xiaodong@codeland:~$ ls ~x<Tab>
xiaodong@codeland:~$ ls ~xiaodong/
```

在 zsh 中，我们可以看到，与 bash 相比，提供的补全用户名列表表现形式略有差异。bash 中包含 `~` 前缀，并在结尾带有 `/` (斜杠)。zsh 中则仅有用户名本身。如图 **??** 所示。

图 2.8: zsh 中的用户名自动补全备选列表

如果你经常使用 `ssh` 登录远程机器的话，那么主机名自动补全将助你一臂之力。同样，我们先来看一个例子。我在输入

```
xiaodong@codeland:~$ ssh xiaodong@l
```

后按 **Tab**，这时 bash 展示了可以自动补全的主机名列表。

```
xiaodong@codeland:~$ ssh xiaodong@l
xiaodong@lab.github.com          xiaodong@localhost
```

```
xiaodong@linuxtoy.org          xiaodong@localhost.localdomain
xiaodong@codeland:~$ ssh xiaodong@l
```

我接着输入 `i` 并按 **Tab**，这次 bash 就自动补全了完整的主机名 `linuxtoy.org`。你也可以直接在 `@` 后按 **Tab**，这样的话就会显示全部主机名了。

```
xiaodong@codeland:~$ ssh xiaodong@li<Tab>
xiaodong@codeland:~$ ssh xiaodong@linuxtoy.org
```

不仅是主机名，而且 IP 地址也同样支持自动补全。另一种情况是，直接在输入 `ssh l` 后按 **Tab**，bash 也能对主机名进行自动补全。

```
xiaodong@codeland:~$ ssh l
lab.github.com          linuxtoy.org          localhost
```

看到这里，你或许会想，bash 从哪里找到这些可以用来自动补全的主机名呢？一个是 `/etc/hosts` 文件的内容，另一个是 `ssh` 的配置文件，比如 `~/.ssh/config`。如图 **??** 所示。所以，如果你打算让 bash 为你自动补全常用的主机名的话，那么不妨考虑将其添加到这两个文件中。此外，还包括 `~/.ssh/known_hosts` 文件。凡是通过 `ssh` 登录过的主机，便会包含其中。

图 2.9: 自动补全的主机名来源

zsh 对主机名的自动补全与 bash 类似，此不赘述。

最后，让我们来看看对变量名的自动补全情况。当我输入

```
xiaodong@codeland:~$ echo $
```

后按 **Tab** 并根据提示按 **y**，这时 bash 显示了全部可供补全的变量名。如图 **??** 所示。

图 2.10: bash 中的变量名自动补全备选列表

然后，我继续输入并搭配 **Tab** 按键，从而补全了变量 BASH_VERSION。

```
xiaodong@codeland:~$ echo $BASH_VERSION
```

zsh 对变量的自动补全与 bash 相似，不过，在我的系统上比 bash 提供的补全列表更多一些。如图 **??** 所示。

图 2.11: zsh 中的变量名自动补全备选列表

综合来看，这几种补全类型跟前面我们所讲的文件名、命令名自动补全还是有一点差异，那就是它们带着一个特殊的前缀字符，参考表 **??**。

表 2.1: 用户名、主机名及变量名自动补全前缀字符

前缀字符	自动补全类型
~	用户名
@	主机名
$	变量名

2.6 可编程补全

在熟悉了命令行自动补全的用法之后，如果你是一位开发人员的话，那么或许会问到这样的问题：“如何为自己所写的程序或脚本添加命令补全呢？”利用 bash 和 zsh 提供的可编程补全特性，我们可以方便地对命令补全加以定制。下面我们就从示例出发来一探究竟。

2.6.1 bash 示例

假设我编写的程序名叫 mycmd，它具有 --help 和 --version 两个命令选项。让我们先来看看它的命令补全效果。当我输入

```
xiaodong@codeland:~$ mycmd -
```

并按 **Tab** 后，这时 bash 为我呈现了该命令的全部选项列表，同时补全成了 `mycmd --`。

```
xiaodong@codeland:~$ mycmd -
--help     --version
xiaodong@codeland:~$ mycmd --
```

我接着输入 `h`，再按 **Tab**，bash 这次就自动补全了命令选项 `--help`。啊哈，这正是我想要的命令补全。那么，如何实现可编程补全呢？

```
xiaodong@codeland:~$ mycmd --h<Tab>
xiaodong@codeland:~$ mycmd --help
```

首先，我们需要在 `/etc/bash_completion.d` 目录下创建 `mycmd` 文件（亦即 `/etc/bash_completion.d/mycmd`）。这样，bash 就会自动加载我们在 `mycmd` 中编写的补全代码。

其次，我们在 `mycmd` 中编写如下用于处理命令自动补全的代码。如图 **??** 所示。

```
#
# Completion for mycmd
#
_mycmd() {
    local cur opts

    cur="${COMP_WORDS[COMP_CWORD]}"
    opts="--help --version"

    if [[ ${cur} == -* ]]; then
        COMPREPLY=($(compgen -W "${opts}" -- ${cur}))
        return 0
    fi
```

```
}

complete -F _mycmd mycmd
```

图 2.12: bash 可编程补全示例

这是一个典型的 bash 脚本。开头的 # 为注释，用于说明补全的用途。

接着我们定义了一个名为 _mycmd 的函数，该函数包含用来处理 mycmd 命令的选项的逻辑。

local 声明了两个变量：cur 和 opts。其中，cur 存储当前在命令行正输入的字，它通过 bash 内置的变量 COMP_WORDS 和 COMP_CWORD 获取。

- COMP_WORDS：数组变量，包含当前命令行中单独的字。
- COMP_CWORD：表示当前光标位置在 ${COMP_WORDS} 中的索引。

而 opts 则用来保存 mycmd 命令所有的命令选项。

然后，我们判断 $cur 是否为 - 打头，若为真，那么就用 compgen 命令来生成可供补全的选项列表。-W 选项后跟我们需要的 mycmd 命令选项。

与此同时，我们将 compgen 产生的输出赋给又一个 bash 内置变量 COMPREPLY。这样，当需要补全时，bash 就会采用 compgen 生成的补全列表了。

最后，我们用 complete 将补全函数 _mycmd（-F 选项）与程序 mycmd 绑定在一起即可。

2.6.2 zsh 示例

现在，让我们来看看在 zsh 中又怎么实现可编程补全吧。

假如我们把 mycmd 的补全代码保存到 $HOME/.zsh/_mycmd 中的话，那么需要在 $HOME/.zshrc 里设置 $fpath，以便 zsh 能够加载我们的补全代码。

```
fpath=($HOME/.zsh $fpath)
```

下面就是我们针对 zsh 而改写的 mycmd 自动补全代码。如图 ?? 所示。

```
#compdef mycmd
#
# Completion for mycmd
#
_mycmd() {
    local cur opts

    cur="${words[CURRENT]}"
    opts=(--help --version)

    if [[ ${cur} == -* ]]; then
        compadd -- ${opts}
        return 0
    fi
}

_mycmd "$@"
```

图 2.13: zsh 可编程补全示例

第一行的注释并非普通注释（#compdef mycmd），它允许 zsh 为我们自动载入补全代码。

接下来定义的函数与变量跟 bash 示例相似，其中已经替换成 zsh 里等价的内容。

- words 相当于 bash 中的 COMP_WORDS
- CURRENT 与 bash 中的 COMP_CWORD 类似
- COMPREPLY 则和 compadd 这个内置的 zsh 命令相同

要试验 mycmd 在 zsh 中的补全效果，只需先执行一下 source ~/.zshrc。从下面的例子中，你可以看到 mycmd 的命令补全跟 bash 中几乎一样，当然也带着 zsh 原本的补全功能。

```
xiaodong@codeland:~$ mycmd --
--help     --version
```

值得一提的是，zsh 本身还提供了一些辅助函数以用于补全，比如 `_arguments`、`_describe`、`_message` 等等，各位读者诸君不妨参考 zsh 的官方文档详加了解，以便用到自己的补全代码中。

第三章　重温历史

在编程领域有一个十分重要的原则，那就是如何想办法来重复利用代码。比如，通过把具有相同逻辑的代码抽象成函数，从而能够加以反复调用。与之类似，在使用命令行时，我们也可以贯彻这个原则——重复利用已经执行过的命令。如果想要达到这样的效果，那么就该轮到 Shell 的历史功能出场了。

我们将先从设置历史变量谈起，接着讨论如何查看、搜索、以及前后移动历史命令，然后来看看怎样快速修改并执行历史命令，最后介绍快速引用历史命令的参数。

为了更好的重温历史，让我们首先来了解历史命令的保存位置和记录大小吧。

3.1　设置历史变量

无论是 bash 还是 zsh，都能够将我们已经执行过的命令存储到一个文件中。这样，便于我们以后对其加以重复使用。要查看 bash 或 zsh 的历史文件位置，不妨执行：

```
xiaodong@codeland:~$ echo $HISTFILE
```

在 bash 中，我们可以看到，这个文件默认是存储到 `~/.bash_history` 的。但是，因为 zsh 默认并没有设置该变量，所以内容为空。

通过向 `$HISTFILE` 变量赋予新值，从而能够更改历史文件的保存位置。下面，我们将 zsh 的历史文件设置为 `~/.zsh_history`。先使用文本编辑器（比如 `nvim`）打开 `~/.zshrc`，然后添加下行内容：

```
HISTFILE=~/.zsh_history
```

对 bash 而言，另外两个重要的历史变量是 $HISTFILESIZE 和 $HISTSIZE。其中，前者为 $HISTFILE 文件所能保存的最大行数，而后者则为 Shell 中记忆的最大历史命令数。这两个变量默认的设置都是 500，换句话说 ~/.bash_history 文件最多保留 500 行，且最多 500 个命令。为了最大限度的利用历史文件的价值，我们不妨考虑把这两个变量的值设得更大一些，比如 5000：

```
HISTFILESIZE=5000
HISTSIZE=5000
```

将以上两行内容追加到 ~/.bashrc 中以便永久保存设置。通常将这两个变量设置的值保持一致，否则在 $HISTFILE 中保存的内容可能会被截断。比如，在 $HISTSIZE 设为 1000 的情况下，而 $HISTFILESIZE 却为 500。因为历史命令数大于文件的行数，所以有部分历史命令不能保存到历史文件中。

$HISTSIZE 变量在 zsh 中同样有效，但与 $HISTFILESIZE 变量等价的却变成了 $SAVEHIST。类似的，我们将 ~/.zsh_history 保留的最大行数和命令数也设为 5000：

```
SAVEHIST=5000
HISTSIZE=5000
```

把上面两行内容添加到 ~/.zshrc 中以便永久保存设置。

既然存储的这些历史命令如此重要，那么就很有必要维护一个整洁、有价值的命令清单了。比如，剔除掉那些重复的命令、开头包含空格的命令、以及常用的简单命令等等。要实现这个目的，在 bash 中我们可以使用 $HISTCONTROL 变量。

$HISTCONTROL 采用冒号分隔的列表来决定是否将命令保存到历史文件中。例如，erasedups 表示去掉重复的命令，而 ignorespace 则意为除去开头具有空格的命令。

```
HISTCONTROL='erasedups:ignorespace'
```

在 zsh 中没有与 bash 对应的内置变量 $HISTCONTROL，不过可以通过设置选项来达到同样的效果：

```
setopt HIST_IGNORE_ALL_DUPS # 去掉重复的命令
setopt HIST_IGNORE_SPACE    # 去掉开头具有空格的命令
```

3.2 查看历史命令

Shell 本身提供了 history 这个内置命令来让我们随时查看所记录的历史命令。当我们执行 history 后，Shell 记录的所有历史命令便被回显出来。如果历史命令太多，不妨将其管道给页面查看程序 less，这样可以分屏查看：

```
xiaodong@codeland:~$ history | less
    1  echo $HISTSIZE
    2  sudo -i
    3  cat .bashrc
    4  cat .bash_profile
    5* cat .bash_history
```

每行命令前面的数字是该行命令的编号。数字后面带 * 号的行则说明已经被修改过。

history 比较有用的一个选项是，它后面可以跟一个数字（比如 5）。这样，在 bash 中就可以看到倒数的 5 个历史命令。

```
xiaodong@codeland:~$ history 5
```

值得注意的是，zsh 中需要在 5 前面加个 - 号：

```
xiaodong@codeland:~$ history -5
```

另外，在 zsh 中，我们也可以给 history 两个负数，以便查看中间的一段历史命令：

```
xiaodong@codeland:~$ history -10 -5
```

这表示从倒数第 10 个到倒数第 5 个之间的历史命令。

对 zsh 来说，它还能向我们提供更多的历史命令细节，包括命令执行的日期和时间，以及每个命令持续运行的时间：

```
xiaodong@codeland:~$ history -i -D
```

这里的 -i 选项向我们展示了命令执行的日期及时间，而 -D 选项则说明了命令运行了好久。

除了 history 之外，另一个用来查看历史命令列表的是 fc。我们利用 fc 的 -l 选项可以将历史命令列出来。例如：

```
xiaodong@codeland:~$ fc -l       # 列出最后 16 条命令
xiaodong@codeland:~$ fc -l -5    # 列出倒数 5 条命令
xiaodong@codeland:~$ fc -l 20 30 # 列出编号 20 到 30 的命令
xiaodong@codeland:~$ fc -l 100   # 列出编号为 100 后的所有命令
xiaodong@codeland:~$ fc -l cat   # 列出 cat 后的所有命令
```

通过 fc 的 -e 选项，我们还能够编辑历史命令列表。比如：

```
xiaodong@codeland:~$ fc -e vi 5 10
```

这将打开 vi 来编辑 5 到 10 条历史命令。

3.3 搜索历史命令

在搜索历史命令时，大家平时用得比较多的是将 `history` 与 `grep` 联用，从而过滤出需要的命令：

```
xiaodong@codeland:~$ history | grep 'xxx'
```

我个人比较喜欢使用的方式是按 **Ctrl + r** 组合键，这样 Shell 会让我们逆向搜索历史命令，比用 `grep` 更加方便。

当我在 bash 中按 **Ctrl + r** 后，Shell 给我提示 reverse-i-search（在 zsh 中这个提示略有不同，为 bck-i-search），然后我可以在冒号后面键入要搜索的字符串，比如 `hi`。此时，Shell 从历史命令中找到了 `history`，按**回车键**可以立即执行该命令。如果想要对命令加以修改，则只需按 → **(右方向键)**。如图 **??** 所示。

图 3.1: 逆向搜索历史命令

这是一个增量搜索引擎，我们每键入一个字符，Shell 便对历史命令列表进行匹配。若是匹配成功，则显出结果。要是匹配失败，我们还可以按**退格键**删除字符，然后重新输入来继续搜索。

3.4 前后移动历史命令

除 **Ctrl + r** 之外，我经常使用的另外两组快捷键是 **Ctrl + p** 和 **Ctrl + n**。这两组快捷键能够让我们在历史命令列表中前后移动。参考表 **??**。

表 3.1: 前后移动历史命令

按键	作用
Ctrl + p	移到前一条命令
Ctrl + n	移到后一条命令

如果我们多次按这两组快捷键，则可以连续前移或后移。这些快捷键 bash 和 zsh 都支持。

3.5 快速修改并执行上一条命令

平常在使用命令行时，我经常会遇到的情况是，要么不小心，要么手太快，总之命令没有输入正确就执行了。这时候，我可不想再次重新输入命令，只想对上一条命令稍微作一下修改。那么，Shell 有没有什么快速而简便的操作方法呢？回答是肯定的，且听我慢慢道来。

3.5.1 删掉多余内容

例如，我在使用 `grep` 过滤日志时，不幸多输入了一个 `o`（原本是 lolcat）：

```
xiaodong@codeland:~$ grep loolcat /var/log/pacman.log
```

我们没有必要重新输入这条命令，只需执行 `^o` 即可将多余的 `o` 字符删除。

```
xiaodong@codeland:~$ ^o
xiaodong@codeland:~$ grep lolcat /var/log/pacman.log
```

Shell 在回显出正确的命令后立即执行了它。这里的 `^o` 将上一条命令中找到的第一个 `o` 字符删除，从而纠正了输错的命令。

3.5.2 替换内容

让我们来看另一个例子，我在查看 `file1` 这个文件的内容时错输成了 `flie1`：

```
xiaodong@codeland:~$ cat flie1
```

现在我们可以用 `^li^il` 来将输错的 `li` 替换为 `il`。同样，Shell 回显出正确的命令并予以执行。

```
xiaodong@codeland:~$ ^li^il
xiaodong@codeland:~$ cat file1
```

即便在没有输错的情况下，^old^new 也是很实用的。假如我在查看 file1 后接着想查看 file4，那么只要执行 ^1^4：

```
xiaodong@codeland:~$ cat file1
xiaodong@codeland:~$ ^1^4
xiaodong@codeland:~$ cat file4
```

3.5.3 全局替换

还有一种情况，有时候我们想不只替换一处，而是把上一条命令中的每处内容都替换掉。要实现这种效果，可以使用 !:gs/old/new，其中，! 表示引用上一条命令（在后续的章节中我们将详细讲解），: (冒号) 后边的 gs 意为全局（g）替换（s），/old/new 则与 ^old^new 相似。

```
xiaodong@codeland:~$ ansible nginx -a 'which nginx'
```

这条命令让我通过 Ansible 了解 nginx 分组的所有机器是否都包含 nginx 程序。接下来，我想看看 haproxy 分组的情况，于是我执行：

```
xiaodong@codeland:~$ !:gs/nginx/haproxy
xiaodong@codeland:~$ ansible haproxy -a 'which haproxy'
```

顺便提一句，在 zsh 中除了支持上述方式外，也可以使用：

```
xiaodong@codeland:~$ ansible nginx -a 'which nginx'
xiaodong@codeland:~$ ^nginx^haproxy^:G
xiaodong@codeland:~$ ansible haproxy -a 'which haproxy'
```

3.6 快速执行历史命令

既然我们把已经执行过的命令存储到 Shell 的历史文件中，那么自然想有一天能够再次用到它。正所谓“养兵千日，用兵一时”。下面，我们就来看一看如何快速的执行已有的历史命令。

3.6.1 重复执行上一条命令

一种常见的使用场景是，我在使用 `htop` 查看系统状态并退出后，过一会儿想再次查看它。此时，我们无需重新输入 `htop` 命令，只需按两下 `!!` 并敲**回车**即可。

```
xiaodong@codeland:~$ htop
xiaodong@codeland:~$ !!
```

`!!` 被称为 bang bang，是我最喜欢使用，同时也是使用频率极高的历史命令调用表示。`!!` 让我们以最快的方式重复执行上一条命令。

`!!` 经常与 `sudo` 联用，用来解决缺少权限的问题。例如：

```
xiaodong@codeland:~$ pacman -S figlet
error: you cannot perform this operation unless you are root.
```

在此，我用 `pacman` 来安装 figlet，但由于是普通账户，所以没有权限操作。要解决这个问题，我们只要输入：

```
xiaodong@codeland:~$ sudo !!
xiaodong@codeland:~$ sudo pacman -S figlet
```

3.6.2 执行以某些字符打头的命令

利用 `!foo` 这种表示法允许我们执行以 `foo` 这三个字符打头的命令。Shell 将以逆序的方式搜索历史命令列表，一旦与给定的开头字符匹配到，便予以执行

该条命令。例如：

```
xiaodong@codeland:~$ !he
xiaodong@codeland:~$ help
```

该表示从历史命令列表中找到 help 后执行。

3.6.3 执行历史列表中第 n 个命令

在 ! 后面除了可以跟一个字符串之外，也可以跟一个数字。这个数字代表历史命令列表中的编号。当我们用 history 查看历史命令列表时，命令左边显示的即是该行命令的编号。例如：

```
xiaodong@codeland:~$ history 5
```

这里显示 htop 的编号为 52，如图 ?? 所示。

图 3.2: history 5 执行结果

所以我们可以用：

```
xiaodong@codeland:~$ !52
```

来再次执行 htop。

利用 !-2，我经常使用的一个场景是，先用文本编辑器编辑源代码，接着再编译源代码。如果我需要再次编辑和编译，那么只要反复执行 !-2 即可。如此不断循环。

```
xiaodong@codeland:~$ nvim first.c
xiaodong@codeland:~$ gcc -o first first.c
xiaodong@codeland:~$ !-2 # 再编辑
xiaodong@codeland:~$ !-2 # 再编译
```

顺便说一句，因为 !-1 是如此常见，所以 Shell 提供了简写形式 !!。

3.7 快速引用上一条命令的参数

很多时候，我们即将执行的命令与之前的命令具有相同的参数，比如同样的文件名、路径名等等。所以，我们在执行新的命令时无需重新输入这些同样的参数，只要直接从其引用过来即可。

3.7.1 引用最后一位参数

我最常用的是 `!$`,它允许我直接复用上一条命令的最后一位参数。当我用 `mkdir` 创建目录后，使用它来立即转到该目录：

```
xiaodong@codeland:~$ mkdir videos
xiaodong@codeland:~$ cd !$
```

这里，`cd` 命令后的 `!$` 等同于上一条命令中的 `videos`。

3.7.2 引用最开头的参数

与最后一位参数相反，`!^` 能够让我们引用上一条命令中最开头的参数。这里的 `^` 和 `$` 与正则表达式中的锚点类似。请看例子：

```
xiaodong@codeland:~$ ls /usr/share/doc /usr/share/man
xiaodong@codeland:~$ cd !^
```

在该例中，`!^` 相当于上一条命令中的路径 `/usr/share/doc`。

3.7.3 引用所有参数

不光是开头或结尾的参数，有时候我们想要引用的是上一条命令的所有参数。此时，我们可以使用 `!*`，这里的 `*` 意为全部。比如：

```
xiaodong@codeland:~$ ls src code
xiaodong@codeland:~$ cp -r !*
```

cp 命令中的 !* 跟 src code 同样，它表示两个参数都要引用。

3.7.4 引用第 n 个参数

对于引用上一条命令中的参数，我们甚至可以要求 Shell 精确到具体的第几个。因为 Shell 按照空白来解析命令行，所以它给命令本身编号为 0，后续的选项和参数按 1、2、3 等依次编号。如图 **??** 所示。这就好比程序中的数组一样。在下面的例子中，假如我们想要引用 bar.txt，除开 touch，按顺序它应该是第 2 个参数，因此可以像这样表示：

图 3.3: 命令及选项参数编号

```
xiaodong@codeland:~$ touch foo.txt bar.txt baz.txt
xiaodong@codeland:~$ nvim !:2
```

nvim 命令中的 !:2 就相当于上一条命令中的 bar.txt 文本文件。

3.7.5 引用从 m 到 n 的参数

还有一种情况可能会遇到，即同时引用上一条命令的好几个参数。此时，我们可以使用 !:m-n 表示法，m 为开始端，n 为结束端。我们继续以上例来说明：

```
xiaodong@codeland:~$ touch foo.txt bar.txt baz.txt
xiaodong@codeland:~$ nvim !:1-2
```

这里的 !:1-2 让我们引用 touch 命令中的前两个参数。

3.7.6 引用从 n 到最后的参数

我们最后再介绍一种情况，通过 `!:n*` 这种表示让我们能够从上一条命令中引用从第 n 个到最后的参数。例如：

```
xiaodong@codeland:~$ cat /etc/resolv.conf /etc/hosts /etc/hostname
xiaodong@codeland:~$ nvim !:2*
```

此处的 `!:2*` 允许我将 `hosts` 和 `hostname` 同时打开进行编辑。

值得提及的是，我们在此主要介绍的是如何引用上一条命令的参数，因为这是最为常见的使用场景。结合我们前面所讲的快速执行历史命令，我们也可以引用历史列表中其它命令的参数。比如：

```
xiaodong@codeland:~$ !hi:2
```

这将引用以 `hi` 打头的命令的第 2 个参数。

```
xiaodong@codeland:~$ !10:2-3
```

而这将引用第 10 条命令的 2、3 两个参数。

3.8 快速引用参数的部分内容

在上一节我们介绍了如何引用历史命令中的参数，除此之外，Shell 甚至比我们想要得到的做得更多。利用 Shell 提供的历史展开模式修饰符，使我们得以快速引用参数中的部分内容。

3.8.1 引用路径开头

请看例子：

```
xiaodong@codeland:~$ ls /usr/share/fonts/truetype
xiaodong@codeland:~$ cd !$:h
```

在此，我想引用该路径的开头部分 /usr/share/fonts。为了达到这个目的，我在 !$（最后一位参数）的基础上添加了 :h。此处的 :h 为修饰符，意味着截取路径的开头部分，正如 dirname 的效果一样。

3.8.2 引用路径结尾

有头就有尾。通过 :t 修饰符，我们可以引用路径的结尾部分，其效果跟 basename 类似。

```
xiaodong@codeland:~$ wget http://nginx.org/download/nginx-1.15.8.tar.gz
xiaodong@codeland:~$ tar zxvf !$:t
```

经过 !$:t 引用后，我们的命令变成了：

```
xiaodong@codeland:~$ tar zxvf nginx-1.15.8.tar.gz
```

3.8.3 引用文件名

对于存在文件名的情形，我们还可以利用 :r 修饰符来只引用文件名部分（这将排除掉扩展名）。例如：

```
xiaodong@codeland:~$ unzip hello.zip
xiaodong@codeland:~$ cd !$:r
```

这里的 !$:r 将 hello.zip 去掉扩展名，只保留 hello 部分。

3.8.4 将引用部分更改为大写

下面介绍的两个修饰符为 zsh 所特有，bash 目前尚不支持。通过 :u 修饰符，我们能够将所引用的部分更改为大写字母。

```
xiaodong@codeland:~$ echo histchars
xiaodong@codeland:~$ echo !$:u
```

这里的 !$:u 将 histchars 全部更改为大写字母。

3.8.5 将引用部分更改为小写

与 :u 相对的是，:l 则使我们能够将所引用的参数全部更改为小写字母。

```
xiaodong@codeland:~$ echo SAVEHIST
xiaodong@codeland:~$ echo !$:l
```

这里的 !$:l 将 SAVEHIST 全部更改为小写字母。

需要特别指出的是，Shell 还支持将多个修饰符进行联用，在它们之间只需使用 : (冒号) 分隔即可。例如：

```
xiaodong@codeland:~$ ls /usr/share/fonts/truetype
xiaodong@codeland:~$ echo !$:t:u
xiaodong@codeland:~$ echo TRUETYPE
```

这里，我们先用 :t 引用了路径的结尾部分，然后又使用 :u 将其更改为了大写字母。

3.9 历史命令展开模式总结

最后，我们来总结一下历史命令展开的模式。从前面我们所讲的内容来看，历史展开模式包括以下三个部分：

1. `!! !foo !n`：用来调用历史列表中的命令
2. `$ ^ * n m-n n*`：引用命令参数的各个部分
3. `h t r u l`：修饰符，对所引用的内容进行修改

如图 **??** 所示。

图 3.4: 历史命令展开模式

模式的每个部分之间都用 :（冒号）进行分隔。让我们来看一个包含三个部分的例子：

```
xiaodong@codeland:~$ !ec:$:t
```

这个模式的含义是，引用 `ec` 打头命令的最后一位参数，并只保留路径尾部。

第四章　编辑大法

当我们在 Vim、Emacs、Sublime、VS Code 等熟悉的编辑器中编辑文本时，通常会有一种十分舒服的感觉。这是因为我们已经习惯了这些编辑器的操作方法。要是 Shell 命令行也能像文本编辑器一样编辑命令，那样的话我们的命令行编辑效率一定会大大提升。相信我，产生这种想法的人绝不止你我。无论是 bash，还是 zsh 的开发者，他们都同样想到了这个问题。正因为如此，所以我们今天才能沿用 Emacs 和 vi 这两个经典的文本编辑器的编辑习惯来编辑命令行。

在本章内容中，我们将先介绍在 Shell 中如何选择 Emacs 或 vi 编辑模式。接着进入 Emacs 编辑模式实战，包括按字、“词”、行来移动和删除的操作方法。最后，我们再讲解怎样在 vi 编辑模式中移动操作、重复执行命令、添加文本、删除文本、替换文本、以及搜索字符等内容。在学完这些内容后，对于编辑命令行而言，你将变得更加游刃有余。

4.1　设置编辑模式

既然 bash 与 zsh 都提供了 Emacs 和 vi 两种编辑模式，那么如何在这两种编辑模式之间进行选择呢？一般而言，在 Emacs 模式下，编辑操作显得更加自然，上手起来相对也更快一些。如果你从来没有使用过 vi 编辑器，那么选用 vi 编辑模式，一开始将会有找不到北的感觉。在 vi 模式下，按键要么能插入文本，要么能执行编辑指令，你需要在两种状态间不断来回切换。Emacs 模式跟 vi 模式相比更加简单，在使用上也会更容易一些。因此，推荐大家优先选择使用 Emacs 编辑模式。这也是 bash 和 zsh 都将 Emacs 作为默认的命令行编辑模式的原因。但是，假如你对 vi 的操作方法非常感兴趣的话，那么不妨选择使用 vi 编辑模式，去做那个敢于吃螃蟹的人。

bash 和 zsh 两个都支持使用 `set` 指令来设置命令行编辑模式。例如，假如我们想要设为 vi 编辑模式，只需执行：

```
xiaodong@codeland:~$ set -o vi
```

要重新设为 Emacs 编辑模式，则执行：

```
xiaodong@codeland:~$ set -o emacs
```

在 zsh 中，我们也可以通过 `bindkey` 来设置 Emacs 或 vi 编辑模式。

```
xiaodong@codeland:~$ bindkey -e
```

该命令将 Emacs 作为编辑模式。如果打算设置为 vi 编辑模式，那么使用 `-v` 选项即可：

```
xiaodong@codeland:~$ bindkey -v
```

为了永久保存设置，我们需要将 bash 的设置选项添加到 `~/.bashrc` 配置文件。

而 zsh 的设置选项则需添加到 `~/.zshrc` 配置文件。

4.2 Emacs 编辑模式实战

4.2.1 按字移动和删除

让我们先在命令行输入一些字符，然后来看看如何一个一个的移动字符：

```
xiaodong@codeland:~$ echo 像骇客一样
```

糟糕，我将“黑客”错输成了“骇客”，现在我要将光标向左移到“客”字，接着删除“骇”字，然后再重新输入正确的“黑”字。

对于命令行新手来说，可能会使用**左方向键**（←）向左移动字符。好，我们来试一下。按 3 下**左方向键**（←），于是光标停留在“客”字上。此时，我们再按**退格键**则删除光标左边的“骇”字。我们接着重新输入正确的“黑”字。因为我们还要继续输入新的内容，所以按 3 下**右方向键**（→）往右移动光标，直到行尾。之后，我们能够再输入新的内容“使用命令行”。

在上面的操作中，我们通过分别按**左方向键**（←）和**右方向键**（→）来向左或往右移动一个字符。利用 Emacs 编辑模式，其实有比按**左右方向键**更好的操作方法。因为**左右方向键**通常远离键盘中心区域，所以与 Emacs 编辑模式提供的操作方法比起来在效率上会有所降低。

下面，我们就用 Emacs 编辑模式所提供的操作方法来一个个的移动字符。在 Emacs 编辑模式中，向左移动一个字符可以按 **Ctrl + b**。因此，我们按 3 次 **Ctrl + b**，此时光标位于“客”字上。同样的，我们通过按**退格键**来删除“骇”字。在输入“黑”字后，我们需要将光标移到最右边。现在，我们可以按 **Ctrl + f**，以便往右移动一个字符。需要往右移动几个字符就按几下 **Ctrl + f**。顺便说一句，在后面的内容中，我们会讲到如何一下子就移到最右边的操作方法。此刻，就先让我们熟悉按字移动的操作吧。

上述操作中我们使用**退格键**删除光标左边的字符。假如我们向左多移动了一个字符，那么此时光标将在“骇”字上。要想删除“骇”字，难道再往右移动吗？当然不必。Emacs 编辑模式为我们提供了 **Ctrl + d** 来删除光标下的字符。

在此，我们再介绍一组快捷键：**Ctrl + t**。这组按键的作用是将光标左边的两个字符交换顺序。例如，当我在输入

```
xiaodong@codeland:~$ sl
```

后，通过按下 **Ctrl + t** 便能够将其变成 `ls`。

```
xiaodong@codeland:~$ ls
```

总结起来，按字移动和删除的操作方法参考表 **??**。

表 4.1: Emacs 模式按字移动和删除的操作方法

按键	作用
Ctrl + b	向左移动一个字符
Ctrl + f	往右移动一个字符
退格键	删除光标左边的字符
Ctrl + d	删除光标下的字符
Ctrl + t	将光标左边的两个字符交换顺序

4.2.2 按 “词” 移动和删除

按字符移动和删除毕竟有些琐碎，为了完成某个操作，有时候需要我们按下很多次快捷键。有鉴于此，Emacs 编辑模式针对 “词” 这个更大的范围提供了移动和删除的操作方法。值得注意的是，我们在此给 “词” 添加了引号。换句话说，这里的 “词” 是 Shell 所理解的 “词” 的含义，与现实生活中所谓的词含义并不相同。在 bash 中，“词” 即字母和数字的组合，例如 file1 这种，其中并不包含特殊字符在内。而在 zsh 中，对 “词” 的界定跟 bash 又有所不同，除了字母和数字之外，还包括比如 *、?、_、- 等这样的符号。通过变量 WORDCHARS，我们可以看到这类字符到底有哪些。在我的系统中，执行

```
xiaodong@codeland:~$ echo $WORDCHARS
```

后，可以见到有如下字符：

```
*?_-.[]~=/&;!#$%^(){}<>
```

如果你想要让 zsh 判定 “词” 的行为跟 bash 一样的话，那么不妨将 WORDCHARS 变量的值设为空：

```
xiaodong@codeland:~$ WORDCHARS=
```

下面我们来看一个按 “词” 移动和删除的操作例子：

```
xiaodong@codeland:~$ grep 'figlet' /var/log/pacman.log
```

在我输入该命令行后，忽然想到应该查询 lolcat 这个包。为了将 figlet 改成 lolcat，我按 **Alt + b** 向左一个“词”一个“词”的移动。按 5 下后，此时光标停留在 figlet 的 f 上。接着，我按 **Alt + d** 删除光标右边的“词”（也就是 figlet）。然后输入新的内容 lolcat。

现在，如果不需要再输入内容的话，则可以直接按**回车键**执行命令。但因为我还想将标准输出的内容保存起来，所以继续按 **Alt + f** 来往右按“词”移动。当抵达最右边时，再输入新的内容 > /tmp/output.log。此刻，我们的命令行是：

```
xiaodong@codeland:~$ grep 'logcat' /var/log/pacman.log \
> /tmp/output.log
```

最后，我按 **Alt + 退格键**（或 **Ctrl + w**）删除光标左边的 log，并输入 txt 来完成该命令行的编辑。

正如前面所述，在 zsh 中按“词”移动的行为跟 bash 略有不同。这是因为它们对“词”界定的含义不一样的缘故。我们仍旧以上面的例子来加以说明。在光标都位于该命令行最右边的情况下，bash 中按 **Alt + b** 向左移动到的是 txt 中开头的 t 上；而在 zsh 中同样的操作却移动到了 /tmp/output.txt 开头的 /。在此，我们可以发现 zsh 将 / 和 . 都看作“词”的一部分。两个 Shell 比较起来，zsh 移动更快，按键也更少，但是粒度却要粗一些。因为 zsh 对“词”的界定范围比 bash 更宽，所以对“词”的删除内容也更多。bash 中按 **Alt + d** 删除的是 txt，而 zsh 中删除的却是 /tmp/output.txt。

需要特别指出的是，通过 **Alt + 退格键**和 **Alt + d** 删除的内容，Shell 并没有丢弃，而是将其保存在 kill ring 中。你可以将 kill ring 看作一个特殊的剪贴板。当然，它里面的内容我们也是可以获取的。我们只需按 **Ctrl + y** 即可获取上次删除的内容。

此外，与 **Ctrl + t** 交换光标左边的两个字符相似，**Alt + t** 能够用来交换光标左边两个“词”的顺序。例如：

```
xiaodong@codeland:~$ echo bar foo
```

当我们按 **Alt + t** 后，该命令行将变成：

```
xiaodong@codeland:~$ echo foo bar
```

以下介绍的几组快捷键用于更改“词”的大小写。例如，我们在命令行输入

```
xiaodong@codeland:~$ echo foo
```

并按 **Alt + b** 将光标移到 `f` 上后，此时，按 **Esc + c** 把 `foo` 变成 `Foo`；若是按 **Esc + u**，则将变为 `FOO`；最后，按 **Esc + l**，又将成为 `foo`。

按“词”移动和删除的操作方法参考表 **??**。

表 4.2: Emacs 模式按“词”移动和删除的操作方法

按键	作用
Alt + b	向左移动一个“词”
Alt + f	往右移动一个“词”
Alt + 退格键	删除光标左边的“词”
Ctrl + w	同上
Alt + d	删除光标右边的“词”
Ctrl + y	获取上次删除的内容
Alt + t	交换光标左边两个“词”的顺序
Esc + c	将光标右边的“词”的开头字母变成大写
Esc + u	将光标右边的“词”全部更改为大写字母
Esc + l	将光标右边的“词”全部更改为小写字母

4.2.3 按行移动和删除

比“词”范围更大的移动及删除操作是行。相对“词”而言，我们只需要更少的按键即可移到更广的区域。由此，也将删除更多的命令行内容。让我们通过一个例子来说明：

```
xiaodong@codeland:~$ grep 'set' *.txt
```

在此，我想找出当前目录中包含 set 的所有文件。假如我想将这行命令改成：

```
xiaodong@codeland:~$ egrep 'set' *.md
```

首先，我按 **Ctrl + a** 将光标移到该命令行的行首（也就是最左边）。接着输入 e 将命令改成 egrep。然后，我再按 **Ctrl + e** 将光标移到此行的结尾处（也就是最右边）。在通过 **Alt + 退格键**删除 txt 后，重新输入 md 即可完成对该命令行的修改。

下面，我们来看看针对行的删除操作。依然请出我们的老朋友 foo、bar、baz：

```
xiaodong@codeland:~$ echo foo bar baz
```

我们先按两下 **Alt + b** 将光标移到 bar 的 b 上。现在，如果我们打算删除 bar 和 baz，那么只要按 **Ctrl + k**；反之，但假设我们想要删除 echo 及 foo 的话，则需按 **Ctrl + u**。值得说明的是，在 zsh 中，**Ctrl + u** 的行为与 bash 中并不相同，它是删除整行的全部内容。

按行移动和删除的操作方法参考表 **??**。

表 4.3: Emacs 模式按行移动和删除的操作方法

按键	作用
Ctrl + a	将光标移到行首（最左边）
Ctrl + e	将光标移到行尾（最右边）
Ctrl + k	从光标处往右删除至行尾
Ctrl + u	从光标处向左删除至行首

4.2.4 Emacs 编辑模式总结

从前面我们所讲的内容来看，Emacs 编辑模式的内容编辑范围主要包括下列 3 种：

1. 字
2. “词”
3. 行

针对每一种范围，又包含两种编辑操作，分别为移动和删除。如图 **??** 所示。

图 4.1: Emacs 编辑模式图解

4.3 vi 编辑模式实战

与 Emacs 编辑模式相比，vi 编辑模式为我们提供了更多的控制命令。相应地，在 vi 编辑模式下，我们能够操作的粒度也将更细。如果你以前从未使用过 vi 编辑模式的话，那么不妨在亲自体验一番之后再来作决定是否要继续用它。跟 vi 文本编辑器一样，Shell 的 vi 编辑模式也包含两种模式：插入模式和命令模式。在插入模式下，我们输入的字符为字符本身，并没有什么特殊含义，如：h 就是 h；而在命令模式中，我们所输入的字符则为用来执行编辑过程的命令，如：**h** 用来向左移动一个字符。默认情况下，我们进入的是插入模式。若是要进入命令模式，则需要我们按 **Esc 键**。在使用 vi 编辑模式时，我们有时候可能会感到迷糊，此刻到底处于哪种模式呢？遇到这种情况，不妨先按 **Esc 键**回到命令模式再作进一步的操作。或许这算是选择 vi 编辑模式的小小代价吧。

4.3.1 移动命令

在 vi 编辑模式中，我们可以使用的光标移动命令参考表 **??**。

表 4.4: vi 模式移动命令

命令	作用
h	向左移动一个字符
l	往右移动一个字符
b	向左移动一个单词
w	往右移动一个单词
e	移到单词结尾

命令	作用
B、W、E	与 b、w、e 类似，按不同的单词定义进行移动
0	移到行首
^	移到行首，但第一个字符为非空白字符
$	移到行尾

让我们输入一行命令来试试这些 vi 编辑模式中的移动命令：

```
xiaodong@codeland:~$ echo hello, this is a command
```

首先，按 **Esc 键**进入命令模式，此时光标位于 command 结尾的 d 上。

其次，通过 **h** 和 **l** 按一个个字符左右移动很直白，无需我们多言。值得讨论的是 **b**、**w**、**e** 跟它们对应的大写形式的区别：简而言之，**b**、**w**、**e** 将停留在空白或标点符号处（如该命令行中的 ,（逗号）），而 **B**、**W**、**E** 则仅仅停留在空白处。例如，我们按 **b** 会经过 ,，而按 **B** 将跳过 ,。假如你想要移动更快的话，那么可以用大写字母的命令。而小写字母命令则对于更细粒度的移动有用，比如路径名这种情形。

最后，**0** 和 ^ 都是移到命令行的开头，不过其差异是 **0** 的开头可以为空白，而 ^ 的开头则不允许。例如：

```
xiaodong@codeland:~$  cd /var/log/nginx
```

按 **0** 移到开头的空格，按 ^ 移到 cd 的 c 上，按 **$** 移到行尾的 x。

4.3.2 重复命令

在 vi 命令模式下，每个移动命令之前可以跟一个数字，用来将该命令重复执行多次。例如，**3b** 表示向左移动 3 个单词，**5l** 则表示往右移动 5 个字符。值得注意的是，因为 **0** 本身也是一个命令，所以将它放在命令前面是无效的重复计数。

4.3.3 添加文本

我们已经知道通过按 **Esc 键**可以进入 vi 命令模式，但是，在命令模式下又如何回到插入模式呢？你只需参考表 **??** 中的命令来执行即可。

表 4.5: vi 模式添加文本的命令

命令	作用
i	在光标左边插入新的文本内容
a	在光标右边追加新的文本内容
I	在行开头插入新的文本内容
A	在行结尾追加新的文本内容

请看例子：

```
xiaodong@codeland:~$ hello vi
```

假如我打算将该命令改为

```
xiaodong@codeland:~$ echo hello, vi editing mode
```

那么可按以下方式执行编辑操作：

按 **2h** 将光标移到空格处，接着按 **i** 进入插入模式，然后输入新的 **,**。按 **Esc 键**回到命令模式后，继续按 **I** 以便在命令行开头插入 `echo`。再次按 **Esc 键**进入命令模式，最后按 **A** 在命令行的尾部追加 `editing mode`。

4.3.4 删除文本

利用 vi 模式提供的删除命令，我们不仅可以删除字符，而且也能删除单词，甚至整个命令行。这些删除命令参考表 **??**。

表 4.6: vi 模式删除文本的命令

命令	作用
x	删除光标下的字符
X	删除光标左边的字符
dm	**m** 为某个移动指令，如 **db** 删除光标左边的单词
D	从光标处删除到行尾
dd	删除整行内容

跟移动命令一样，在上述删除命令之前也可以带一个数字，以便多次执行该命令。例如，**5x** 将删除 5 个字符，而 **3dw** 将删除 3 个单词，这里 3 的顺序并不重要，**d3w** 仍然同样有效。

通过删除命令删除的内容，Shell 并没有丢弃，而是将其保留在了删除缓冲器中。稍后，我们可以执行 **u** 命令来恢复这些删除的内容。如果想要恢复更早时间删除的内容，则只需按 **u 键**多次即可。

另外一种更有用的方式是复制和粘贴。这样，我们能够在保留原有内容的同时，再储存一份拷贝，以便后续使用。vi 模式中复制及粘贴的命令参考表 **??**。

表 4.7: vi 模式复制及粘贴命令

命令	作用
ym	**m** 为某个移动命令，如 **yw** 用来复制光标右边的单词
p	在光标右边粘贴文本
P	**p** 的大写形式，在光标左边粘贴文本

在下面的例子中，我们将看到上述命令的用法：

```
xiaodong@codeland:~$ echo command-line interface
```

先按 **Esc 键**进入命令模式，此时光标位于结尾的 e 上。按 **x** 将删除 e，按 **X** 将删除 a，按 **db** 将删除 interface 剩下的部分（只剩下字符 c），按 **dd** 则把整行内容都删掉。按 **u** 又还原刚删除的内容。

4.3.5 替换文本

当我们需要替换命令行中的内容时，除了在删除该内容后再进入插入模式重新输入外，也可以使用 vi 编辑模式所提供的文本替换命令。这些命令组合了删除与插入操作，用起来将更加直接。vi 编辑模式提供的文本替换命令参考表 **??**。

表 4.8: vi 模式替换文本的命令

命令	作用
cm	**m** 为某个移动命令，如 `cw`
C	从光标处删除到行尾，并进入插入模式
cc	删除整行，并进入插入模式
r	替换光标下的字符
R	进入替换文本模式
s	利用输入的字符来替换光标下的字符

要想搞明白这些替换命令如何工作，不妨来试试以下编辑练习：

```
xiaodong@codeland:~$ echo talk is cheap. show me the kode.
```

同样的，我们按 **Esc 键**先进入命令模式，按 **cb** 将 kode 删除后进入了插入模式，我们输入新的内容 code。再次按 **Esc**，接着按 **4b** 左移到 show，按 **r** 将 s 替换成 S。

4.3.6 搜索字符

vi 编辑模式还提供了一组命令用于搜索命令行中的字符。利用这些命令，我们可以移动光标到特定的字符上。此外，将其跟 **d** 和 **c** 命令组合使用，还能够删除或更改从光标处到该字符的这一段文本。这些用于搜索字符的命令参考表 **??**。

表 4.9: vi 模式搜索字符的命令

命令	作用
fc	移动光标到 **c** 的下一处
Fc	与 **f** 相反方向搜索，移动光标到 **c** 的上一处
tc	移动光标到 **c** 左边的字符
Tc	移动光标到 **c** 右边的字符
;	重复上次的 **f** 或 **F** 命令
,	以相反的方向重复上次的 **f** 或 **F** 命令

在下面的练习中，我们可以尝试上述字符搜索命令的用法：

```
xiaodong@codeland:~$ echo A program which handles the interface
```

在按 **Esc 键**进入命令模式后，按 **fp** 光标移到了 p 上，按 **th** 移到了 h 左边的 w。按 **Fm** 光标左移到 m 上。我们还可以试试其它的命令，以便熟悉这些命令的用途。

4.3.7 vi 编辑模式总结

从前面我们所讲的内容来看，vi 编辑模式比 Emacs 编辑模式提供了更多的编辑命令。乍一看，似乎很复杂。我们除了勤加练习以期熟悉这些编辑命令之外，还可以总结出以下规律，从而帮助我们更好的加深理解。如图 **??** 所示。

- 跟 Emacs 编辑模式一样，我们同样可以按照字、“词”、行这 3 个维度来梳理操作命令
- 删除命令 **d** 和更改命令 **c** 能够与移动命令组合使用
- 移动命令、删除命令、更改命令之前可以加数字用来多次执行

图 4.2: vi 编辑模式图解

第五章　必备锦囊

在本章内容中，我们将向大家介绍几个实用的命令行使用妙招，包括如何快速导航文件系统、使用别名来省时省力、通过 {} (花括号) 构造命令参数、使用命令替换和变量减少操作步骤、以及重复执行命令等内容。学完本章内容后，我们希望大家在使用命令行时都能够做到常备锦囊，心中不慌。

5.1　快速导航

在命令行下，你如何穿越文件系统的“丛林”而不致迷路？你又如何快速定位所需的文件和目录？如果你仅仅了解导航的基本用法，那么恐怕是不够的。在本节，我将教你几个必备的技能，使你能够轻车熟路的驾驭命令行导航。

5.1.1　回到用户主目录

也许你已经知道 ~ (波浪线) 这个特殊字符代表用户的主目录，若是想要回到自己的主目录，那么我们可以执行：

```
xiaodong@codeland:~$ cd ~
```

但是且慢，我想在此告诉你的是，不带任何参数的 cd 命令同样能够将你带回主目录：

```
xiaodong@codeland:~$ cd
```

换句话说，这两个命令行所达到的效果是相同的。然而，两相比较起来，后者比前者可以少输两个字符，和乐而不为呢？

说到 ~ (波浪线)，你还应该了解的一个技巧是，我们可以利用它来转到别的用户的主目录。例如：

```
xiaodong@codeland:~$ cd ~mingji
```

我们在 ~ (波浪线) 后面直接跟上 mingji 这个用户名，于是 cd 将我们带到了该用户的主目录。值得注意的是，以下命令行与它并不相同：

```
xiaodong@codeland:~$ cd ~/mingji
```

请注意 ~ (波浪线) 后的 / (斜杠)，这行命令的作用是转到当前用户主目录下的子目录 mingji。

5.1.2 回到上次工作的目录

我经常使用的一个导航场景是，在目录 A 中处理了任务之后，接着转到目录 B 中处理任务，一旦完成，我需要再次回到目录 A 继续工作。此时，我们可以执行下面的命令来回到上次工作的目录：

```
xiaodong@codeland:~$ cd ~/prj/usingcli
xiaodong@codeland:~/prj/usingcli$ pwd
/home/xiaodong/prj/usingcli
xiaodong@codeland:~/prj/usingcli$ cd ~/cli
xiaodong@codeland:~/cli$ pwd
/home/xiaodong/cli
xiaodong@codeland:~/cli$ cd -
/home/xiaodong/prj/usingcli
xiaodong@codeland:~/prj/usingcli$ pwd
/home/xiaodong/prj/usingcli
```

在此，我第一次工作的目录是 ~/prj/usingcli，第二次工作的目录是 ~/cli。

通过执行 cd 后跟一个 - (减号)，我们快速的回到了第一次工作的目录。cd - 命令相当于执行 cd "$OLDPWD" 及 pwd 两条命令。

```
xiaodong@codeland:~$ cd "$OLDPWD" && pwd
```

我们还可以继续重复执行 cd -，这样就会在 ~/prj/usingcli 和 ~/cli 两个目录之间反复切换。

5.1.3 访问常用目录

对于需要频繁访问的深层次目录，直接导航起来感觉还是比较麻烦。幸运的是，bash 和 zsh 两个都为我们提供了 $CDPATH 变量。这是一个与 $PATH 类似的变量，它由 : (冒号) 分隔的路径列表组成。利用 $CDPATH，我们能够将常用的目录保存起来，以便 cd 为我们直接转到这些目录。比如：

```
xiaodong@codeland:~$ CDPATH=:~:~/src:~/prj/usingcli
```

在这里，我们将 ~ (用户主目录)、~/src、以及 ~/prj/usingcli 等目录加到了 $CDPATH 中。注意 = (等号) 后面 : (冒号) 的左边为空，它表示当前目录，你应当予以保留。否则，在相对路径的情况下，cd 就不能转到当前目录下的子目录了。

现在，假如我们打算转到 ~/prj/usingcli/build 目录下的话，那么只要执行下列命令即可：

```
xiaodong@codeland:~$ cd build
xiaodong@codeland:~/prj/usingcli/build$
```

这个例子同时也告诉我们，加到 $CDPATH 路径列表的目录为待导航的目标目录的父目录。

除了 $CDPATH 变量，zsh 也支持 $cdpath 变量。

5.1.4 自动纠正错误

在用 cd 导航目录时，我们免不了偶尔会输错目录的名称。bash 有一个名为 cdspell 的选项可以帮助我们自动纠正拼写错误，并导航到正确的目录。像是不正确的字母顺序、缺少或者多余的字符等错误，cdspell 都能纠正。

我们在享用如此好的功能之前，需要首先开启 bash 的控制选项：

```
xiaodong@codeland:~$ shopt -s cdspell
```

shopt 命令的 -s 用于启用 cdspell 选项。

现在我们来试一下 cdspell 的效果，我们原本是想要导航到 /etc 目录，但是我们却错输成了 /ect。不过没有关系，bash 已经帮我们自动纠正了错误，并且转到了正确的目录。

```
xiaodong@codeland:~$ cd /ect
/etc
xiaodong@codeland:/etc$ cd -
xiaodong@codeland:~$ cd /et
/etc
xiaodong@codeland:/etc$ cd -
xiaodong@codeland:~$ cd /etcd
/etc
xiaodong@codeland:/etc$
```

在 zsh 中，我们可以给 cd 命令两个参数，它们分别是搜索与替换字符串。zsh 将根据搜索字符串来查看当前工作目录，然后使用第二个字符串替换它，并转到替换后的目录。

```
xiaodong@codeland:~/cli/1.15.8/src$ pwd
/home/xiaodong/cli/1.15.8/src
xiaodong@codeland:~/cli/1.15.8/src$ cd 1.15.8 1.15.9
xiaodong@codeland:~/cli/1.15.9/src$ pwd
/home/xiaodong/cli/1.15.9/src
```

本例中，我们的当前工作目录为 /home/xiaodong/cli/1.15.8/src，cd 命令的第一个参数 1.15.8 跟当前工作目录相匹配。zsh 用 1.15.9 替换了 1.15.8，然后转到了新的目录 /home/xiaodong/cli/1.15.9/src。

5.1.5 自动导航

因为 cd 命令是如此常用，我们使用它的频率又那么高，所以 bash 和 zsh 两个都为 cd 命令提供了一个捷径，对像我一样的 “懒人” 来说，autocd 选项极其有用。

在 bash 中，我们可以通过下面的命令来启用 autocd 选项：

```
xiaodong@codeland:~$ shopt -s autocd
```

与此对应的 zsh 指令为：

```
xiaodong@codeland:~$ setopt autocd
```

现在假设我们想导航到 ~/prj 目录，代替执行：

```
xiaodong@codeland:~$ cd prj
```

我们可以省略 cd 命令，直接执行：

```
xiaodong@codeland:~$ prj
xiaodong@codeland:~/prj$ pwd
/home/xiaodong/prj
```

5.1.6 使用目录栈

前面我们讲到的 cd - 允许我们在两个目录之间进行切换，如果我们想要在更多个目录之间切换，那么它就无能为力了。不过，为了帮助我们解决这个问题，bash 和 zsh 提供了目录栈功能。

两个最基本的目录栈命令是 pushd 和 popd。其中，pushd 命令将一个目录添加到目录栈中，而 popd 命令则从目录栈中移除上次添加的目录。

```
xiaodong@codeland:~$ pwd
/home/xiaodong
xiaodong@codeland:~$ pushd ~/cli
~/cli ~
xiaodong@codeland:~/cli$ pushd ~/prj
~/prj ~/cli ~
xiaodong@codeland:~/prj$ popd
~/cli ~
xiaodong@codeland:~/cli$ popd
~
xiaodong@codeland:~$
```

在这个例子中，我们第一次执行 pushd ~/cli 后，将 ~/cli 添加到目录栈的同时，并且转到了该目录。接着，我们继续执行 pushd ~/prj，又将 ~/prj 目录添加到了目录栈。此刻，我们位于 ~/prj 目录，目录栈中包括 3 个条目：

```
~/prj ~/cli ~
```

出栈的顺序跟入栈的顺序相反，第一次执行 popd 命令后，移除了目录栈中的最左边的条目 ~/prj，并转到了相邻的 ~/cli 目录。再次执行 popd 命令，则又移除 ~/cli 条目，然后转到 ~ 目录。

如果你在执行多次入栈与出栈后忘了目录栈中还有哪些条目的话，那么可以执行 dirs -v 命令来查看：

```
xiaodong@codeland:~$ dirs -v
 0  ~/prj
 1  ~/cli
 2  ~
```

dirs -v 为我们列出了目录栈中的所有条目，每行一条，而且开头具有编号，以便我们引用。例如：

```
xiaodong@codeland:~/prj$ pushd +1
~/cli ~ ~/prj
xiaodong@codeland:~/cli$
```

执行 `pushd +1` 将使 `~/cli` 成为目录栈的顶端，并变成当前工作目录。我们可以通过再次执行 `dirs -v` 命令来确认这一点：

```
xiaodong@codeland:~/cli$ dirs -v
 0  ~/cli
 1  ~
 2  ~/prj
xiaodong@codeland:~/cli$ pwd
/home/xiaodong/cli
```

`pushd` 命令中的 `+` (加号) 用于从上往下计数。我们也可以使用 `-` (减号) 来从下往上计数。比如：

```
xiaodong@codeland:~/cli$ popd -1
~/cli ~/prj
xiaodong@codeland:~/cli$ dirs -v
 0  ~/cli
 1  ~/prj
```

执行 `popd -1` 命令后从目录栈中移除了倒数第二个条目 `~`。

5.2 使用别名

别名是命令行下最常用的省时技巧之一。它通过对频繁使用的命令及选项重新定义一个较短的名称，从而使我们能够减少输入，最终达到提高操作效率的目的。让我们先来看看如何定义别名。

5.2.1 定义别名

不管是 bash，还是 zsh，它们都能使用 `alias` 命令来定义别名。例如，假如我们要将 `ls -lah --color=auto` 定义成 `l` 的话，那么可以执行下列命令：

```
xiaodong@codeland:~$ alias l='ls -lah --color=auto'
```

代替输入长长的 `ls -lsh` 命令，现在我们只需直接执行 `l` 即可。

```
xiaodong@codeland:~$ l
```

比较常见的别名定义包括下面这些：

```
alias ..='cd ..'
alias ...='cd ../..'
alias ....='cd ../../..'
alias ls='ls --color=auto'
alias l='ls -lah --color=auto'
alias la='ls -AF --color=auto'
alias ll='ls -lFh --color=auto'
```

除了节省时间，利用别名我们也可以避免经常性的输入错误。如果你常常将 `ls` 错输成 `sl`，那么不妨为它定义一个别名：

```
xiaodong@codeland:~$ alias sl='ls'
```

在 zsh 中，`alias` 命令还能用 `-s` 选项来定义后缀别名。例如，当我们将文件扩展名 `pdf` 定义成 `zathura` 后缀别名后，直接执行 `pdf` 文件名，就会调用 `zathura` 打开该 `pdf` 文件。

```
xiaodong@codeland:~$ alias -s pdf=zathura
xiaodong@codeland:~$ cheat_sheet_ssh_v4.pdf
```

在这个例子中，我们通过执行 `cheat_sheet_ssh_v4.pdf` 来代替执行 `zathura cheat_sheet_ssh_v4.pdf` 命令。

5.2.2 查看别名

时间久了，也许你将忘记所定义别名的具体内容。为了查看别名 sd 的内容，我们将它作为参数传递给 alias 命令：

```
xiaodong@codeland:~$ alias sd
alias sd='shutdown -h now'
```

从 alias 列出的内容，我们知道 sd 是 shutdown -h now 的别名。

alias 命令也可以不带任何参数，直接予以执行则会列出当前 Shell 中的所有别名。

```
xiaodong@codeland:~$ alias
```

5.2.3 取消别名

如果某个别名不再适用，那么我们可以使用 unalias 命令来取消它。

```
xiaodong@codeland:~$ unalias sl
```

执行 unalias sl 命令后，将取消我们先前定义的 sl 别名。

```
xiaodong@codeland:~$ sl
```

此时我们再执行 sl 就不是列出当前工作目录的内容了。

比永久取消别名更加有用的一个技巧是临时取消别名。

```
xiaodong@codeland:~$ ls
xiaodong@codeland:~$ \ls
```

第一次 ls 执行的是别名，第二次我们在 ls 前面放了一个 \ (反斜杠)，用来临时取消别名。请比较两个命令先后执行的输出结果。

此外，unalias 命令还支持 -a 选项，利用这个选项可以移除所有定义的别名。

```
xiaodong@codeland:~$ unalias -a
```

5.2.4 别名的缺憾

使用别名固然好，但它也有一些缺憾。一方面，别名无法参数化，打算实现一个参数化别名的想法注定会失败。这时候，你应当考虑使用的是函数。另一方面，别名可能覆盖真实的命令，从而误导你原本想要执行命令的意图。

通过 type -a 我们可以确定别名极其真实的命令：

```
xiaodong@codeland:~$ type -a sl
sl is aliased to `ls'
sl is /usr/bin/sl
sl is /sbin/sl
sl is /usr/sbin/sl
```

这里，我们既能看到 sl 是 ls 的别名，也能看到它有一个真实的命令。

5.3 利用 {} 构造参数

在命令行下，我们经常会遇到针对多个参数条目执行操作的使用场景，其中文件名算得上是最常见的情形。为了应对这种情况，bash 和 zsh 都提供了逗号分隔的花括号列表，例如：

```
xiaodong@codeland:~$ echo {one,two,three}
```

将参数传递给 echo 命令之前，Shell 会首先展开花括号列表，并生成以下 3 个参数条目：

```
one two three
```

利用 {} (花括号)，我们可以实现许多有意思的功能，下面是我常用的几个。大家在学习时不妨举一反三，以便灵活运用。

5.3.1 备份文件

我发现很多朋友在备份文件时执行的命令是：

```
xiaodong@codeland:~$ cp file file.bak
```

这条命令将 file 备份为 file.bak。通过 {} (花括号)，我们可以将该命令改写成：

```
xiaodong@codeland:~$ cp file{,.bak}
```

cp 命令的参数 file{,.bak} 展开后将变成 file file.bak。这里的 , (逗号) 必不可少，否则 Shell 就不会将其展开了。

类似的，我们也可以利用 tar 结合 {} (花括号) 来创建存档：

```
xiaodong@codeland:~$ tar cf docs{.tar,}
```

这里，tar 命令将 docs 目录存档为 docs.tar。

5.3.2 生成序列

对于按照一定顺序排列的条目，代替 , (逗号)，Shell 也支持通过 .. (两点) 来指定一个区间。比如：

```
xiaodong@codeland:~$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

echo 将回显从 a 到 z 所有的小写字母。

又如：

```
xiaodong@codeland:~$ echo {0..9}
0 1 2 3 4 5 6 7 8 9
```

echo 将回显 0 到 9 的数字。

在区间开头数字的前面也可以添加前导的 0，比如：

```
xiaodong@codeland:~$ echo {01..10}
```

这将罗列出以下数字序列：

```
01 02 03 04 05 06 07 08 09 10
```

我们甚至还可以在区间的尾端添加一个步进值：

```
xiaodong@codeland:~$ echo {1..9..2}
```

末尾的 2 为步进值，这样就只会罗列奇数：

```
1 3 5 7 9
```

在 zsh 中，步进值可以为负数，这种情况下将按倒序罗列数字，例如：

```
xiaodong@codeland:~$ echo {1..9..-2}
9 7 5 3 1
```

bash 中想要达到同样的效果需要将区间的首尾端对调，比如：

```
xiaodong@codeland:~$ echo {9..1..2}
9 7 5 3 1
```

最后，让我们来看一个实际使用序列的例子。通过生成的序列，将其与路径组合，在下载多个文件时尤其有用。

```
xiaodong@codeland:~$ wget https://linuxtoy.org/img/{1..5}.png
```

上述命令中，wget 将从 https://linuxtoy.org 依次下载 1.png、2.png、3.png … 等图片文件。

值得一提的是，除了 .. (两点) 的区间表示法，zsh 也支持 - (减号) 这种区间表示，不过需要启用 braceccl 选项。

```
xiaodong@codeland:~$ setopt braceccl
xiaodong@codeland:~$ echo {A-Za-z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

我们在使用 setopt 开启 braceccl 选项后，执行 echo {A-Za-z} 罗列出了所有大写和小写字母。

5.3.3 连用与嵌套

Shell 的 {} (花括号) 结构非常灵活和强大，特别是在连用和嵌套时更是威力无穷。先来让我们看一个 {} (花括号) 连用的例子。

```
xiaodong@codeland:~$ mkdir -p 2019/{01..12}/{baby,photo}
```

在本例中，我们连用两个 {} (花括号)，这样在每个月份的目录下又分别创建了 baby 和 photo 两个子目录。这条命令实际上执行的是以下命令：

```
xiaodong@codeland:~$ mkdir -p 2019/01/baby 2019/01/photo \
2019/02/baby 2019/02/photo \
2019/03/baby 2019/03/photo \
2019/04/baby 2019/04/photo \
2019/05/baby 2019/05/photo \
2019/06/baby 2019/06/photo \
2019/07/baby 2019/07/photo \
```

```
2019/08/baby 2019/08/photo \
2019/09/baby 2019/09/photo \
2019/10/baby 2019/10/photo \
2019/11/baby 2019/11/photo \
2019/12/baby 2019/12/photo
```

我们不妨将两个命令比较一下，如果直接手动输入后者该是多么枯燥和乏味的事情。但是，有了 {} (花括号) 的帮助，我们就变轻松了不少。

{} (花括号) 结构不仅可以连用，而且能够嵌套。例如：

```
xiaodong@codeland:~$ echo {{A..Z},{a..z},{0..9}}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
```

echo 命令的外层 {} (花括号) 中包含 3 个内层 {} (花括号)，这样就将所有的大写字母、小写字母以及从 0 到 9 的数字都罗列出来了。

5.4 其它妙招

在本节中，我们介绍其它几个必备的命令行使用妙招。

5.4.1 命令替换

假设我们想要编辑包含 error 的 Python 源代码文件 (*.py)，在此之前，我们首先需要确定哪些 Python 文件具有这些字符串。为此，我们可以使用下面的命令将其找出来：

```
xiaodong@codeland:~$ grep -l error *.py
```

这里的 -l 选项让 grep 命令将找出的文件名输出到终端，然后我们就可以用文本编辑器 (如 nvim) 来编辑这些文件：

```
xiaodong@codeland:~$ nvim godns.py
```

代替这种需要两步才能完成的操作，我们也可以要求 Shell 在执行 grep 命令后直接将文件名传递给文本编辑器 (如 nvim)：

```
xiaodong@codeland:~$ nvim `grep -l error *.py`
```

Shell 对上述命令 `` (反引号) 中的内容执行命令替换，也就是说，把 grep 命令的输出作为 nvim 命令的参数。我们从而将两步操作得以合并成一步操作来完成。

使用 `` (反引号) 算是老式的命令替换用法，更新式的用法是 $()。所以上面的命令也能改写成如下形式：

```
xiaodong@codeland:~$ nvim $(grep -l error *.py)
```

我们推荐使用 $() 这种新用法，因为在嵌套命令替换时具有更好的可读性。例如：

```
xiaodong@codeland:~$ nvim $(grep -l failed $(date +'%Y%m%d').log)
xiaodong@codeland:~$ nvim `grep -l failed \`date +'%Y%m%d'\`.log`
```

在嵌套时，$() 看起来一目了然，而 `` (反引号) 则需要转义，其可读性较差。

5.4.2 使用变量

虽然 Shell 本身具有内置变量，而且我们在执行程序时还会碰到环境变量，但是我们在此要讲的却是另一种变量，即用户变量。

用户变量是由我们自己所设置的变量，其目的在于临时保存需要多次使用的数据。这样，当我们需要使用数据时就可以通过变量名来引用它了。因为通常变量名比我们要引用的数据要短很多，所以也让我们减少了不必要的重复输入。

让我们看一个例子：

```
xiaodong@codeland:~$ LOG=/var/log/pacman.log
xiaodong@codeland:~$ head $LOG
xiaodong@codeland:~$ grep -i error $LOG
```

我们将变量 LOG 的值设置为 /var/log/pacman.log，接着通过 $LOG 的形式分别在 head 和 grep 命令中引用它。

5.4.3 重复执行命令

当我为演讲主题准备材料时，我想用 figlet 这个工具来制作一些有趣的 ASCII 艺术字。虽然 figlet 提供了很多艺术样式，但是我并没有见过每一种。要想选择最酷的 ASCII 艺术字，所以我必须把每种样式都浏览一遍。于是，我执行了下面的命令：

```
xiaodong@codeland:~$ figlet -f ascii9 Linux
xiaodong@codeland:~$ figlet -f bigmono9 Linux
xiaodong@codeland:~$ figlet -f emboss Linux
...
```

可是，figlet 还有很多样式，这样一个一个的查看实在枯燥乏味。Shell 有没有什么重复执行命令的快捷方法呢？回答是肯定的。利用 Shell 提供的 for 循环，我们可以一遍又一遍的重复执行命令。下面让我们来一次性查看 figlet 的所有样式。

```
xiaodong@codeland:~$ cd /usr/share/figlet
xiaodong@codeland:~$ for font in *.tlf
> do
>     echo "Font: $font"
>     figlet -f $(basename $font .tlf) Linux
> done
```

for 和 in 中间的 font 为循环的变量，*.tlf 表示扩展名为 tlf 的所有文件。do 与 done 之间的内容为循环体，针对当前目录下的每个 tlf 文件都会执行这两条命令。

除了上面多行形式的 `for ... in` 循环结构外，我们也可以使用其单行形式，例如：

```
for font in *.tlf; do figlet -f $(basename $font .tlf) Linux; done
```

第六章　周边好品

在前面的章节中，我们主要讨论的是 Shell 本身自带的特性。因为 bash 和 zsh 都是开源软件，所以有许多热心的用户为其添砖加瓦，以便增强它们的功能，使其变得更加好用。本章我们将眼光从 Shell 自身移到周边，来看看由社区贡献的一些好工具。

6.1　配置框架

无论是 bash，还是 zsh，我们都能根据自己的好恶来进行配置。这样的好处是配置完全由自己所掌控。但其缺点也很明显，那就是不便于公开分享和贡献。正因为如此，一些 Shell 配置框架应运而生。所谓“众人拾柴火焰高”，积社区之力，Shell 配置框架的内容极其丰富。这使我们得以能够直接将其拿来享用。

虽然现在有很多 Shell 配置框架可以选择，有的宣称很轻量，也有的号称速度快，但是我们将根据流行度和活跃性来进行选择。经过综合比较，我们选择的 bash 配置框架是 Bash-it，zsh 配置框架是 Oh My Zsh。下面，我们将分别进行介绍。

6.1.1　bash 配置框架

Bash-it 配置框架从社区收集了许多实用的命令和脚本，主要包括别名、自动补全代码、定制函数、以及提示符主题等四大类型。得益于 Bash-it 良好的模块化架构，我们也可以添加自己的定制内容。

6.1.1.1 安装

因为 Bash-it 位于 GitHub[1] 上，所以在安装 Bash-it 之前，首先需要确认的是系统中是否含有 `git` 命令：

```
xiaodong@codeland:~$ which git
```

如果输出内容为：

```
/usr/bin/git
```

则可进行下一个安装步骤。否则，可通过所用操作系统的软件包管理器（如 `apt`、`yum`、`pacman`、`emerge` 等）来安装。

接着，使用 `git` 命令将 Bash-it 克隆到用户主目录下的 `.bash_it` 子目录：

```
xiaodong@codeland:~$ git clone --depth=1 \
https://github.com/Bash-it/bash-it.git \
~/.bash_it
```

然后，执行 `install.sh` 安装脚本来安装 Bash-it：

```
xiaodong@codeland:~$ cd ~/.bash_it
xiaodong@codeland:~/.bash_it$ ./install.sh -h
xiaodong@codeland:~/.bash_it$ ./install.sh
```

`install.sh` 脚本包括下列 3 个选项，大家可根据需要使用：

1. `interactive (-i)`：这个选项允许我们交互式选择要启用哪些别名、自动补全和插件。
2. `--silent (-s)`：静默安装，没有任何输入提示。
3. `--no-modify-config (-n)`：不修改现有的 bash 配置文件 `.bashrc` 或 `.bash_profile`。

[1] https://github.com/Bash-it/bash-it

我们不加任何选项，采用默认安装。在安装脚本询问是否保留 .bashrc 并追加 Bash-it 模板内容时，回答 “N”。这样，我们原有的 .bashrc 配置文件将备份为 .bashrc.bak。

当看到 “安装成功完成” 的消息时，则说明 Bash-it 安装成功。如图 ?? 所示。

图 6.1: Bash-it 安装过程

最后，我们关闭并重新打开终端（远端机器则注销并重新登录）或者执行下面的命令就可以开始使用 Bash-it 了：

```
xiaodong@codeland:~/.bash_it$ source ~/.bashrc
```

6.1.1.2 查看别名、补全和插件

Bash-it 的 install.sh 脚本在默认情况下只会启用少量的别名、自动补全和插件，下面就让我们找出来。

首先，我们来看看启用了哪些别名：

```
xiaodong@codeland:~$ bash-it show aliases | less
```

该命令的输出分为 3 列，第一列为别名的名称，第二列显示该别名是否启用（启用的别名在 [] 中有 X），最后一列是有关别名的说明。如图 ?? 所示。

图 6.2: 在 Bash-it 中查看别名

在此我们可以看到 Bash-it 启用了 general、apt、ansible 等别名。

接着，我们看看启用了哪些自动补全：

```
xiaodong@codeland:~$ bash-it show completions | less
```

我们发现 system、bash-it 等自动补全已经启用了。如图 ?? 所示。

最后，我们再看看启用了哪些插件：

图 6.3: 在 Bash-it 中查看补全

```
xiaodong@codeland:~$ bash-it show plugins | less
```

在输出中显示 base、alias-completion 等插件已经启用。如图 **??** 所示。

图 6.4: 在 Bash-it 中查看插件

6.1.1.3 搜索内容

除了直接列出所有的别名、自动补全和插件之外，Bash-it 还提供了一个非常快捷的方式来查找所需的内容。比如，我们想要看看有关 tmux 和 ansible 的情况，不妨执行以下命令：

```
xiaodong@codeland:~$ bash-it search tmux ansible
    aliases:  ansible tmux
    plugins:  tmux tmuxinator
completions:  tmux
```

从命令输出中我们看到 tmux 的别名、补全和插件都没有启用，而 ansible 的别名已经启用，因为其颜色为绿色。

6.1.1.4 启用别名、补全和插件

既然 Bash-it 为我们提供了如此丰富的别名、自动补全和插件，那么不用的话就太可惜了。下面，我们就来分别看看如何启用别名、自动补全和插件。

时下非常流行的版本控制系统 Git，相信大家都有过接触和使用。让我们先启用 git 别名：

```
xiaodong@codeland:~$ bash-it enable alias git
git enabled with priority 150.
```

Bash-it 为我们回显出启用的结果，并设置优先级 150。

接着，我们执行以下命令：

```
xiaodong@codeland:~$ bash-it reload
```

以便让 Bash-it 自动为我们重新载入配置。这样，我们就能够立即开始使用刚才启用的 git 别名。但是，到底有哪些 git 别名呢？别着急，我们可以通过下面的命令了解：

```
xiaodong@codeland:~$ bash-it help aliases git | less
```

大致估算一下，Bash-it 提供的 git 别名差不多有近 80 个，实在是非常多。如图 **??** 所示。

图 6.5: Bash-it 提供的 git 别名

最后，我们拿 Bash-it 的源代码目录来验证 git 别名是否生效：

```
xiaodong@codeland:~$ cd ~/.bash_it
xiaodong@codeland:~/.bash_it$ gs
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

从 `gs` 所显示出的 git 仓库状态，我们可以确信 git 别名一切正常。

再来让我们启用 git 自动补全：

```
xiaodong@codeland:~$ bash-it enable completion git
git enabled with priority 350.
```

与启用 git 别名类似，在启用 git 自动补全时，Bash-it 也为其分配了一个优先级 350。

还有 git 插件，我们也将其启用：

```
xiaodong@codeland:~$ bash-it enable plugin git
git enabled with priority 250.
```

Bash-it 命令的输出仍然与启用别名和自动补全时相似。

除了通过 `bash-it enable` 命令来启用别名、自动补全和插件外，我们也可以在搜索模块和组件时加以启用。例如：

```
xiaodong@codeland:~$ bash-it search git --enable
    aliases:  git gitsvn
    plugins:  autojump fasd git git-subrepo jgitflow jump
completions:  git git_flow git_flow_avh
```

这样，将把包含 git 关键字的所有别名、自动补全和插件全都启用。本例中的 gitsvn、jgitflow、git_flow 也一并启用了。

此外，我们也可以通过下列命令来分别启用所有的别名、自动补全和插件：

```
xiaodong@codeland:~$ bash-it enable alias all
xiaodong@codeland:~$ bash-it enable completion all
xiaodong@codeland:~$ bash-it enable plugin all
```

禁用别名、自动补全和插件跟启用时类似，只是把 enable 换成 disable 即可。比如，假设我们想要禁用 gitsvn 别名，则可以执行：

```
xiaodong@codeland:~$ bash-it disable alias gitsvn
```

如果禁用成功，Bash-it 命令输出如下结果：

```
gitsvn disabled.
```

6.1.1.5 更改主题

Bash-it 随附了大约 50 多个提示符主题样式，如果想要看看这些主题的真实外观，那么我们可以执行下面的命令：

```
xiaodong@codeland:~$ BASH_PREVIEW=true bash-it reload
```

Bash-it 的默认主题为 bobby。要是你不喜欢的话，那么可以将其更改为别的主题。不过，Bash-it 并没有提供相关的更改命令，我们需要直接编辑 `.bashrc` 配置文件。

使用文本编辑器（如 vim）打开 `~/.bashrc`：

```
xiaodong@codeland:~$ vim ~/.bashrc
```

然后找到下行内容：

```
export BASH_IT_THEME='bobby'
```

将单引号中的内容（bobby）替换成别的主题名称（如 powerline），并保存即可。

为了使新设置的提示符主题生效，你需要关闭并重新打开终端，或者注销并重新登录。如图 **??** 所示。

图 6.6: Bash-it 提示符主题

6.1.1.6 定制别名、插件和主题

Bash-it 的确为我们提供了不少好用的别名、自动补全和插件，然而，我们还是有不能满足的时候。为此，Bash-it 提供了一种方便我们进行定制的机制，可以定制的内容包括别名、自动补全、插件、主题样式等等，它们的路径和名称如下：

- `aliases/custom.aliases.bash`：别名

- completion/custom.completion.bash：自动补全
- lib/custom.bash：库
- plugins/custom.plugins.bash：插件
- custom/themes/<theme name>/<theme name>.theme.bash：主题样式

在此，我们以如何定制别名为例来说明，其它类型的定制方法类似，无非就是以特定的名称命名并放在确定的目录。

首先,我们在 aliases 目录下使用文本编辑器（如 vim）创建 custom.aliases.bash 文件：

```
xiaodong@codeland:~$ cd ~/.bash_it/aliases
xiaodong@codeland:~$ vim custom.aliases.bash
```

接着，添加具体的别名内容：

```
alias sd='shutdown -h now'
alias up='uptime'
```

编辑完成后保存。

然后，我们可以利用以下命令来查看：

```
xiaodong@codeland:~$ bash-it help aliases
custom:
sd='shutdown -h now'
up='uptime'
```

末尾显示的 custom 正是我们添加的内容。

我们再重新加载一下配置：

```
xiaodong@codeland:~$ bash-it reload
```

现在，我们刚刚添加的定制别名就可以开始使用了：

```
xiaodong@codeland:~$ up
07:54:08 up  5:23,  1 user,  load average: 0.00, 0.00, 0.00
```

6.1.1.7 更新 Bash-it

Bash-it 是社区化项目，隔段时间便会增添新的模块和组件，或是修正过往版本中的缺陷。我们可以通过将 Bash-it 更新到最新版本，以保持同步。为了更新 Bash-it，我们可执行以下命令：

```
xiaodong@codeland:~$ bash-it update
```

6.1.2 zsh 配置框架

与 Bash-it 一样，Oh My Zsh[2] 也是开源的社区性项目。在所有的 zsh 配置框架中，Oh My Zsh 算得上是最流行的。Oh My Zsh 包含了许多来自 zsh 社区的好东东，它主要体现在插件、实用的函数、辅助例程、提示符主题样式等方面。

6.1.2.1 安装 Oh My Zsh

因为 Oh My Zsh 的安装依赖 git 和 curl（或 wget），所以在安装它之前，我们先检查一下系统中是否存在它们：

```
xiaodong@codeland:~$ which git curl wget
```

如果以上命令没有返回结果，那么我们需要先把它们安装到系统中。

好了，若是一切准备就绪，现在就可以执行下面的命令来安装 Oh My Zsh：

```
xiaodong@codeland:~$ sh -c \
"$(curl -fsSL
https://raw.github.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"
```

[2] https://github.com/robbyrussell/oh-my-zsh

安装过程要花一会儿功夫，我们需耐心等待。Oh My Zsh 在安装时将对原来的 .zshrc 配置文件备份，若是出现 "....is now installed!" 的字样时，就说明 Oh My Zsh 安装成功了。如图 ?? 所示。

图 6.7: Oh My Zsh 安装过程

6.1.2.2 启用插件

在安装时，Oh My Zsh 只启用了一个 git 插件。若想用得更舒服，我们就得启用其它插件。对 Oh My Zsh 而言，所谓插件通常包括别名、实用函数、自动补全等内容。在启用其它插件之前，也许你想要了解一下 Oh My Zsh 到底有哪些插件？为此，我们可以转到 Oh My Zsh 安装目录下的 plugins 子目录查看：

```
xiaodong@codeland:~$ cd ~/.oh-my-zsh/plugins
xiaodong@codeland:~/.oh-my-zsh/plugins$ ls
```

我们可以看到，在 Oh My Zsh 中，每个插件都有一个单独的目录。例如，我们继续进入 systemd 目录：

```
xiaodong@codeland:~/.oh-my-zsh/plugins$ cd systemd
xiaodong@codeland:~/.oh-my-zsh/plugins/systemd$ ls
```

其下包含 README.md 和 systemd.plugin.zsh 两个文件。前者为有关该插件的使用说明，而后者为 systemd 插件的源代码。如图 ??。

图 6.8: Oh My Zsh 插件目录

另外，我们也可以通过浏览器查看 Oh My Zsh 的 Wiki 页面[3]来了解它有哪些插件。

在知晓了有哪些插件及其插件的用途后，现在我们就来启用插件。我们先使用文本编辑器打开 .zshrc 配置文件：

[3] https://github.com/robbyrussell/oh-my-zsh/wiki/Plugins

```
xiaodong@codeland:~$ vim ~/.zshrc
```

然后找到下行内容：

```
plugins=(git)
```

将需要启用的插件追加到 git 后面即可。

```
plugins=(git colored-man-pages systemd)
```

我们在此启用了 colored-man-pages 和 systemd 插件。

为了使启用的插件生效，我们还需要执行下面的命令：

```
xiaodong@codeland:~$ source ~/.zshrc
```

现在让我们来验证一下启用的插件，如图 ?? 和 ?? 所示。

```
xiaodong@codeland:~$ man zsh
xiaodong@codeland:~$ sc-status sshd
xiaodong@codeland:~$ sc-list-units
```

图 6.9: 执行 man zsh 的输出结果

图 6.10: 执行 sc-status sshd 的输出结果

6.1.2.3 更换主题

Oh My Zsh 的默认提示符主题样式是 robbyrussell，若是你不喜欢，那么可以更换成别的主题样式。在 Oh My Zsh 中包含有上百个主题样式，每种主题样式的外观可通过其 Wiki 页面[4]查看。

[4] https://github.com/robbyrussell/oh-my-zsh/wiki/Themes

一旦选定了 Oh My Zsh 的主题样式，便可通过编辑 .zshrc 配置文件的方式来更换。

```
xiaodong@codeland:~$ vim ~/.zshrc
```

在 zshrc 文件中找到下面这行：

```
ZSH_THEME="robbyrussell"
```

将双引号中的内容替换成另外的主题样式名称（如 simple）。如果你拿不定主意选哪个主题样式，那么不妨将其设置为 random。这样，Oh My Zsh 就会为你随机选择一种主题样式。

保存编辑后执行以下命令以便使更改生效，如图 ?? 所示。

```
xiaodong@codeland:~$ source ~/.zshrc
```

图 6.11: Oh My Zsh 的 simple 主题样式

6.1.2.4 定制插件和主题

虽然 Oh My Zsh 已经涵盖了不少插件和主题，但是我们还是有可能去添加一些个性化的东西。直接修改原有内容的方法明显不可取。为了解决这个问题，Oh My Zsh 提供了一个名为 custom 的目录。我们只需把要定制的内容保存到 .zsh 结尾的文件即可。例如，如果我们打算添加一些新的别名，那么在创建 myaliases.zsh 文件后，再加入下列内容：

```
xiaodong@codeland:~$ cd ~/.oh-my-zsh/custom
xiaodong@codeland:~$ vim myaliases.zsh
alias sd='shutdown -h now'
alias up='uptime'
```

然后保存编辑结果，并执行：

```
xiaodong@codeland:~$ source ~/.zshrc
```

现在我们就可以使用新加的别名了：

```
xiaodong@codeland:~$ up
```

类似的，如果要定制插件和主题，它们有专门存放的目录。定制的插件代码需放在以下目录：

```
~/.oh-my-zsh/custom/plugins
```

除了我们自己的插件外，这个目录也可以用来安装外部插件。zsh-syntax-highlighting[5] 是一个为 zsh 提供语法加亮的插件，其实现灵感来自于 fish shell。遗憾的是，该插件并没有包含到 Oh My Zsh 中。为了使用它，我们可以把它安装到上述目录：

```
xiaodong@codeland:~$ cd ~/.oh-my-zsh/custom/plugins
xiaodong@codeland:~/.oh-my-zsh/custom/plugins$ git clone \
https://github.com/zsh-users/zsh-syntax-highlighting.git
```

然后在 .zshrc 配置文件中启用：

```
plugins=(< 原有插件> zsh-syntax-highlighting)
```

添加 zsh-syntax-highlighting 前，如图 ?? 所示。

```
xiaodong@codeland:~$ echo $'hello, world\x21'
```

图 6.12: 未启用 zsh-syntax-highlighting 时

启用 zsh-syntax-highlighting 后，如图 ?? 所示。

[5] https://github.com/zsh-users/zsh-syntax-highlighting

```
xiaodong@codeland:~$ source ~/.zshrc
xiaodong@codeland:~$ echo $'hello, world\x21'
```

图 6.13: 启用 zsh-syntax-highlighting 后

此外，要把定制的主题代码放在下面的目录：

```
~/.oh-my-zsh/custom/themes
```

6.1.2.5 其它命令和配置

Oh My Zsh 还带了其它几个有趣而实用的命令。下面简单对其作个介绍：

- take：创建新目录，并转到该目录，而且支持多级目录，如：take www/oops
- zsh_stats：列出使用次数最多的 20 个命令，包含使用次数和所占百分比统计，如图 ?? 所示。

图 6.14: zsh stats

此外，Oh My Zsh 也包含其它一些有用的配置，我们可根据自己的需要设置：

```
xiaodong@codeland:~$ vim ~/.zshrc

ENABLE_CORRECTION="true" # 启用命令自动纠正
```

6.1.2.6 获取更新

Oh My Zsh 在默认情况下将每隔几周检查是否有更新。若有更新，则予以提示并进行升级。如果你觉得提示烦人的话，那么可以将其关闭。关闭提示更新的方法是，把下行内容追加到 ~/.zshrc 配置文件中：

```
DISABLE_UPDATE_PROMPT=true
```

我们也可以手动更新 Oh My Zsh，为此，可以执行以下命令，结果如图 ?? 所示。

```
xiaodong@codeland:~$ upgrade_oh_my_zsh
```

图 6.15: 更新 Oh My Zsh

6.2 增强工具

除了像 Bash-it 和 Oh My Zsh 这种大而全的 Shell 配置框架外，市面上也包括一些单独的工具。利用这些工具，我们不仅能够增强 Shell 的功能，而且可以达到更加愉悦的 Shell 使用体验。下面，我们就来介绍既实用又有意思的第三方增强工具。

6.2.1 快速路径切换：z.lua

z.lua[6] 是一个使用 Lua 编程语言实现的类似 z.sh、autojump、fasd 等功能的快速路径切换工具。其特点包括：

- 对所有访问路径进行跟踪，具备学习功能，使用正则匹配从而能够更加准确的带你到想去的地方；
- 与 z.sh、autojump、fasd 相比，速度更快；
- 支持广泛的 shell，包含 bash、zsh、fish 等等。

6.2.1.1 安装 z.lua

因为 z.lua 是通过 Lua 编程语言实现的，所以在安装它之前，确保我们的系统中已经包含 `lua`。

[6] https://github.com/skywind3000/z.lua

```
xiaodong@codeland:~$ which lua lua5.1 lua5.2 lua5.3
```

z.lua 支持 Lua 5.1、5.2、5.3 以上版本，上述命令只要有一个正确输出便可。

接下来，我们将 z.lua 的源代码克隆到用户主目录的 `~/.z.lua` 子目录：

```
xiaodong@codeland:~$ git clone \
https://github.com/skywind3000/z.lua.git \
~/.z.lua
```

待克隆完毕，我们将 z.lua 的初始化语句添加到相应的 Shell 配置文件。

对 bash 来说，将下面这行追加到 `~/.bashrc` 配置文件中：

```
eval "$(lua ~/.z.lua/z.lua --init bash)
```

若是 zsh，则添加以下内容到 `~/.zshrc` 配置文件：

```
eval "$(lua ~/.z.lua/z.lua --init zsh)
```

另外，我们也可以在 `--init` 选项后面添加 `once enhanced` 参数，以便执行增强匹配模式。我们推荐使用这种匹配模式，从而让 z.lua 更准确的切换到我们想去的路径。

保存编辑后，分别执行下列命令以便使 z.lua 即时生效：

```
xiaodong@codeland:~$ source ~/.bashrc # bash
xiaodong@codeland:~$ source ~/.zshrc  # zsh
```

6.2.1.2 使用 z.lua

z.lua 在安装成功后会创建一个名为 `z` 的别名。让我们先用 `z` 来切换一些目录：

```
xiaodong@codeland:~$ z cli
xiaodong@codeland:~/cli$ z ~/prj
xiaodong@codeland:~/prj$ z usingcli
xiaodong@codeland:~/prj/usingcli$ z ~/tmp
xiaodong@codeland:~/tmp$ z ~/.z.lua
xiaodong@codeland:~/.z.lua$
```

在默认设置下，z.lua 将所有访问的路径都保存到了 `~/.zlua` 文件中。我们可以用 `cat` 来查看它的内容：

```
xiaodong@codeland:~$ cat ~/.zlua
/home/xiaodong/cli|6|1551946470
/home/xiaodong/.z.lua|3|1551946500
/home/xiaodong/prj|8|1551944681
/home/xiaodong/tmp|3|1551944599
/home/xiaodong/prj/usingcli|1|1551944549
```

从上面命令的输出中，我们可以看到每个访问路径的次数和时间都有记录。它们使用 `|` 分隔。z.lua 正是通过这个访问路径数据库来帮助我们快速切换到想去的目录。

另外，我们也可以直接执行 `z`，这将使 z.lua 列出所访问的路径条目：

```
xiaodong@codeland:~$ z
6           /home/xiaodong/.z.lua
12          /home/xiaodong/cli/1.15.8/src/event/modules
12          /home/xiaodong/cli/1.15.8/src/event
16          /home/xiaodong/prj/usingcli
20          /home/xiaodong/cli/1.15.8/src
28          /home/xiaodong/tmp
32          /home/xiaodong/cli
52          /home/xiaodong/prj
```

其中，开头的数字为每个访问路径的权重，该权重由 z.lua 根据算法计算得到。

了解了 z.lua 的基本原理，下面我们就来进行实战：

```
xiaodong@codeland:~$ z p
xiaodong@codeland:~/prj$
```

在执行该命令后，z.lua 通过参数 p 匹配到了我们先前访问的路径：

```
/home/xiaodong/prj
```

并转到了该目录。

我们再来试试：

```
xiaodong@codeland:~/prj$ z p us
xiaodong@codeland:~/prj/usingcli$
```

这次，z.lua 将我们带到了下面的目录：

```
/home/xiaodong/prj/usingcli
```

在这种情况下，z.lua 必须同时匹配 p（匹配 prj）和 us（匹配 usingcli），只有两个条件都满足，才会转到相应的目录。

此外，z 命令还包含一些选项以实现其它的功能。比如：

- -i：交互模式，如果有多个匹配结果的话，那么 z.lua 将展示一个列表：

  ```
  xiaodong@codeland:~$ z -i p
  2:  24          /home/xiaodong/tmp
  1:  48          /home/xiaodong/prj
  >
  ```

  从列表中，我们可以看到有两个路径条目，中间列为权重。在提示符 > 后输入编号，z.lua 将带我们到相应的目录。直接按**回车键**将不进行跳转。

- -b：这个选项在深层次目录中跳转特别有用，它可以将我们快速带回某一级的父目录：

```
xiaodong@codeland:~$ z mod
xiaodong@codeland:~/cli/1.15.8/src/event/modules$ z -b sr
xiaodong@codeland:~/cli/1.15.8/src$
```

第一次，z.lua 跳转到了以下目录：

```
~/cli/1.15.8/src/event/modules
```

在添加 `-b` 选项后，z.lua 根据给定的匹配关键字 `sr` 匹配到了 `src` 这级父目录，并跳转到了该目录：

```
~/cli/1.15.8/src
```

值得一提的是，z.lua 还支持自动补全。如果我们在执行 `z p` 时按 **Tab 键**，z.lua 则会显示一个列表：

```
xiaodong@codeland:~$ z p<Tab>
xiaodong@codeland:~$ z /home/xiaodong/prj
/home/xiaodong/prj  /home/xiaodong/tmp
```

经过一段时间的使用，我们认为 z.lua 确实是相当不错的路径切换工具，每一个使用命令行的用户都应当将其纳入自己的工具箱。

6.2.2 高效查询 Shell 历史：HSTR

虽然在 bash 和 zsh 中，我们可以使用 **Ctrl + r** 来搜索历史命令列表，但是它还不够高效。为此，程序员又开发出了比 **Ctrl + r** 更好用的工具：HSTR[7]。我们可以把 HSTR 看作 **Ctrl + r** 的增强版本，它使用起来也更加方便。

6.2.2.1 安装 HSTR

HSTR 支持 bash 和 zsh，其本身是使用 C 编写而成，在使用它之前，我们必须先安装它。HSTR 为常见的 Linux 发行版都提供有二进制包，包括 Debian、Ubuntu、Fedora、CentOS 等等。

在 Debian 和 Ubuntu 中，可按如下方式安装：

[7] https://github.com/dvorka/hstr

```
xiaodong@codeland:~$ sudo -i
root@codeland:~# echo -e "\ndeb https://www.mindforger.com/debian \
      stretch main" >> /etc/apt/sources.list
root@codeland:~# wget -qO - https://www.mindforger.com/gpgpubkey.txt \
      | apt-key add -
root@codeland:~# apt update && apt install hstr
```

如果要在 Fedora 和 CentOS 中安装 HSTR，那么可以执行：

```
xiaodong@codeland:~$ sudo yum install hstr
```

Arch Linux 可从 AUR[8] 安装 HSTR。

6.2.2.2 配置 HSTR

一旦安装完毕 HSTR，我们便可通过 `hstr` 命令来调用它。不过，在此之前，我们需要先对其进行配置。

对 bash 来说，执行以下命令：

```
xiaodong@codeland:~$ hstr -s
alias hh=hstr                    # hh to be alias for hstr
export HSTR_CONFIG=hicolor       # get more colors
shopt -s histappend              # append new history items to
                                 # .bash_history
export HISTCONTROL=ignorespace   # leading space hides commands
                                 # from history
export HISTFILESIZE=10000        # increase history file size
                                 # (default is 500)
export HISTSIZE=${HISTFILESIZE}  # increase history size
                                 # (default is 500)
# ensure synchronization between Bash memory and history file
```

[8] https://aur.archlinux.org/packages/hstr/

```
export PROMPT_COMMAND="history -a; history -n; ${PROMPT_COMMAND}"
# if this is interactive shell, then bind hstr to Ctrl-r
# (for Vi mode check doc)
if [[ $- =~ .*i.* ]]; then bind '"\C-r": "\C-a hstr -- \C-j"'; fi
# if this is interactive shell, then bind 'kill last command' to
# Ctrl-x k
if [[ $- =~ .*i.* ]]; then bind '"\C-xk": "\C-a hstr -k \C-j"'; fi
```

```
xiaodong@codeland:~$ hstr -s >> ~/.bashrc
```

如果是 zsh，则执行：

```
xiaodong@codeland:~$ hstr -z
alias hh=hstr                    # hh to be alias for hstr
export HISTFILE=~/.zsh_history # ensure history file visibility
export HSTR_CONFIG=hicolor       # get more colors
bindkey -s "\C-r" "\eqhstr\n"  # bind hstr to Ctrl-r
                                 # (for Vi mode check doc)
```

```
xiaodong@codeland:~$ hstr -z >> ~/.zshrc
```

这段配置不仅为 HSTR 定义了别名 hh，而且为其绑定了快捷键 **Ctrl + r**。为了使其生效，分别重新载入 bash 和 zsh 的配置文件：

```
xiaodong@codeland:~$ source ~/.bashrc
xiaodong@codeland:~$ source ~/.zshrc
```

6.2.2.3 HSTR 的用法

HSTR 已经准备就绪，现在我们可以开始使用它了。有两种启动 HSTR 的方式，一种是执行命令，另一种是按快捷键。下面，我们分别对其进行介绍。

首先，我们来看看如何通过命令的方式来启动 HSTR。前面在配置 HSTR 时，我们为其定义了别名 `hh`。所以我们在命令行直接输入 `hh` 并按**回车键**：

```
xiaodong@codeland:~$ cd ~/cli
xiaodong@codeland:~/cli$ hh
```

HSTR 为我们呈现了一个可以交互的文本界面，如图 **??** 所示。该界面大致可以分为 4 个部分，从上往下依次为：

1. 命令输入行：我们输入命令，或者查询关键字的地方。
2. 使用提示行：包含如何使用的说明以及按键的作用。
3. 当前状态行：显示当前视图的排列方式、关键字的匹配方法、以及是否区分大小写等信息。
4. 历史命令列表：根据视图的排列方式展示历史命令。

图 6.16: HSTR 的界面

让我们在命令输入行输入一些字符看看：

```
xiaodong@codeland$ vag # vagrant
```

我们每输入一个字符，HSTR 都会进行搜索，并将列表中匹配的命令用高亮颜色显示。如果输错了，则按**退格键**或 **Ctrl + u** 删除。然后，再重新输入。

现在，根据我们的需要，我们可以进行如下选择：

- 直接按**回车键**，这将执行列表中的第一行命令。
- 如果想要对命令进行编辑，那么我们可以按 **Tab 键**。
- 或者，按 **Ctrl + g** 取消本次操作。

在执行 `hh` 的时候，我们也可以带一个查询关键字参数。这样，HSTR 启动时就会直接将过滤结果展示给我们，如图 **??** 所示。

```
xiaodong@codeland:~/cli$ hh nvim
```

除了执行命令外，HSTR 还支持通过快捷键启动。我们只需要按 **Ctrl + r** 即可。

图 6.17: 执行 hh nvim 的结果

6.2.2.3.1 列表操作

前面我们只提到了如何使用列表中的第一条命令，如果我们想要使用其它命令，那么又该如何操作呢？

首先，在 HSTR 的交互文本界面中，我们可以通过下面的快捷键来上下移动：

- **下方向键**或 **Ctrl + n**：向下移动一行
- **上方向键**或 **Ctrl + p**：往上移动一行

一旦选定某行命令，除了执行我们先前介绍的操作之外，我们还能够将其加入收藏夹，或者删除不再需要的命令。

- **Ctrl + f**：将命令添加到收藏夹
- **Delete 键**：删除命令，根据提示按 **y** 将确认删除

6.2.2.3.2 控制选项

在 HSTR 的交互文本界面中，我们可以通过更改它的控制选项，从而改变其行为。HSTR 的控制选项主要包括以下 3 种：

1. 视图排序方式：包含按 HSTR 的排名算法排序、历史顺序、以及收藏夹。按 **Ctrl + /**可以在这 3 中视图方式中循环。
2. 匹配方法：包括关键字匹配、精确匹配、以及正则匹配。按 **Ctrl + e** 可以在 3 中匹配方法中切换。
3. 是否区分大小写：按 **Ctrl + t** 进行切换。

随着对 HSTR 的使用次数越多，我们越感觉 HSTR 确实是很棒的工具。对于想要追求操作效率的各位读者来说，一定不要错过。

第七章　结语

在本书所讲的内容中，其实自始至终是贯穿着一些基本原则的。正是因为有了这些原则的指引，我们才能万变不离其终。举一反三，方能灵活运用。我们认为最重要的原则包括：

- **少打多做**：所谓“少打”就是说要尽量少打字。我们输入越少，越能省时。那么，我们出错的机会也将随之变少，自然效率便会升高。我们使用自动补全，我们重用历史命令，其背后都是这个道理。另一方面，不要害怕做试验，要多练，只有通过实践才能了解事物的原理。熟能生巧，练习最终会转化成生产力。

- **不要重复自己**：这条原则又称为 DRY，也就是 Don’t repeat yourself，我从很喜欢的一本书《程序员修炼之道：从小工到专家》上学到的。人是创造性的动物，对于重复性的任务往往会感到厌烦。我们最好把这类任务交给计算机去做。计算机不仅比我们做的效率更高，一般也更少出错。举个例子，我们用 `for` 循环重复执行命令便是应用的该原则。

- **关心你的工具**：有句俗话叫“磨刀不误砍柴工”，如果工具不拿来时不时的打磨一番，那么再好的工具久而久之也会变得不那么好用了。我们不断打磨 bash 和 zsh，将它们调配到最适合我们操作的状态，用起来自然得心应手。

到此为止，我们的命令行旅程就算结束了。通过学习，我相信你已经具有了独自前行的能力和勇气。最后，我们向大家推荐一些深入的学习材料。虽然“RTFM”充满戏谑的成分，但是我们认为手册还是应当读的。

- Bash Reference Manual[1]：这份《Bash 参考手册》是 bash 的权威指南，

[1] https://www.gnu.org/software/bash/manual/html_node/index.html

如果你立志将其作为自己的主力 Shell 使用，那么每年都应该温习一遍。

- A User's Guide to Zsh[2]：《Zsh 用户指南》，这份文档由 zsh 的作者所写，全面的介绍了 zsh 的用法。
- 《Unix Power Tools》：我非常喜欢的一本书，Unix 工具和技巧的集大成者。虽然这本书是面向 Unix 进行的讲解，但是可以同样应用于 Linux。

[2] http://zsh.sourceforge.net/Guide/zshguide.html