



中国海洋大学
OCEAN UNIVERSITY OF CHINA

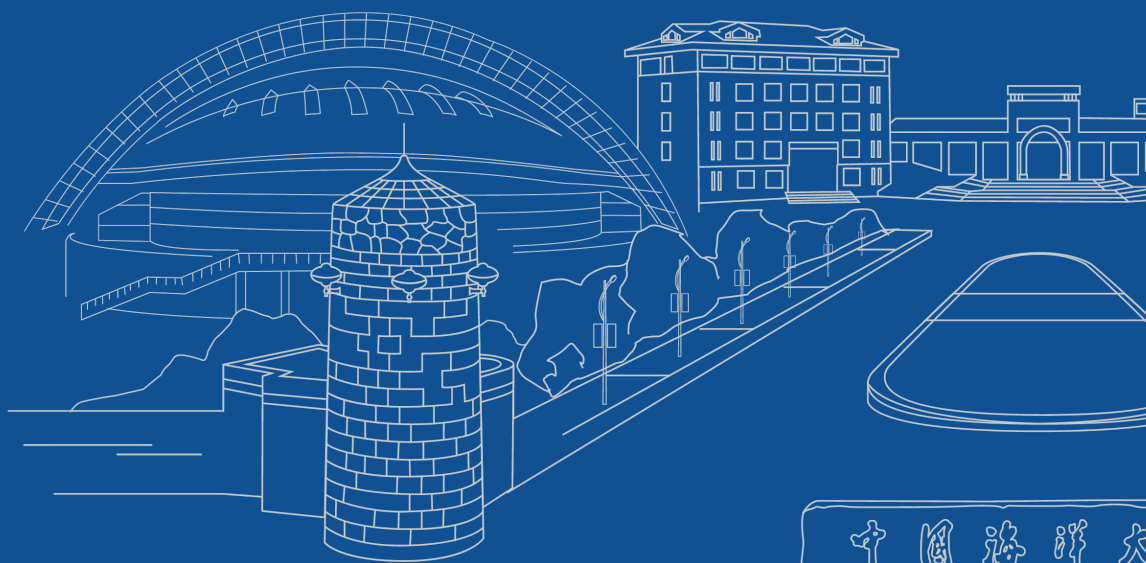
系统开发工具基础第二次实验报告

掌握 Shell 的基本用法 学习数据整理与练习 Vim

于景一

23090032047

信息科学与工程学部
计算机学院
计算机科学与技术专业



OUC, September 2024

中国海洋大学



中国海洋大学
OCEAN UNIVERSITY OF CHINA

系统开发工具基础第二次实验报告

掌握 Shell 的基本用法 学习数据整理与练习 Vim

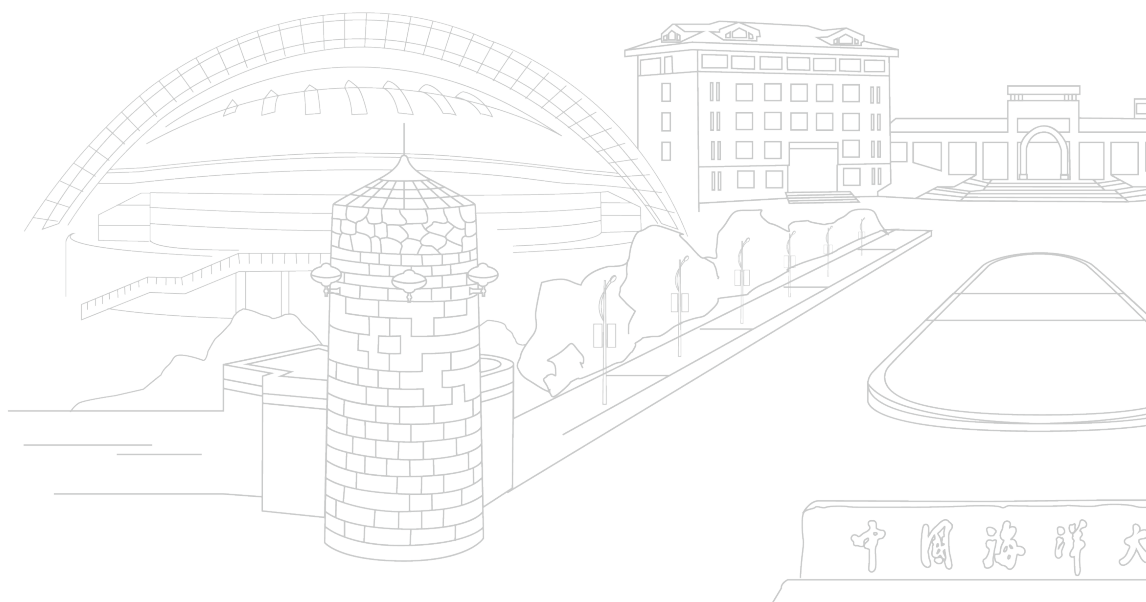
于景一

23090032047

指导老师: 周小伟

系统开发工具基础, 计算机学院

信息科学与工程学部
计算机学院
计算机科学与技术专业



OUC, September 2024

概述

本实验报告使用由于景一制作的 \LaTeX 模板完成。关于此模板的信息,您可以前往GitHub模板仓库具体了解。¹² 本文章使用的是“实验报告”模板。³

本实验报告是系统开发工具基础课程的第二次实验报告,主要关于掌握 **Shell** 的基本用法,主要讨论各个发行版、数据管道、简单的 *Shell* 脚本、*Shell* 内置的工具,学习使用 *Shell* 工具进行数据整理,尝试并熟练 **Vim** 的使用,总结实战经验,记录心得体会。

本实验报告的仓库地址为 ToolBasics2 by jstar0。

在 **Shell** 的基本用法上,首先对各个发行版的 *Shell* 进行分析比对,主要包括 **Bash**, **Zsh** 等,并分析其工作原理与交互模式,了解主要的 *Shell* 内置工具,并学习数据管道的使用,主要是 `|`, `>`, `>>`, `<` 等,最后了解 *Shell* 脚本语法,并以系统运维为主题编写一个简单的 *Shell* 脚本。

在数据整理方面,首先了解其基本概念,尔后,学习常用的数据处理工具,最后学习基本的正则表达式,并尝试与 *Shell* 管道组合使用。

在 **Vim** 的使用方面,包括 *Vim* 的哲学及其类似 *TUI* 编辑器的介绍,了解并熟悉 *Vim* 的基本操作,完成 `vimtutor`。

NOTE

知识之海是无边无际的,只是尽力游弋,就已倍感费时费力,然而学习的过程是美好的。本次实验在 **Shell** 和 **Vim** 方面的粗浅研究,仅可窥得其冰山一角。本文十分惭愧地呈现了我在实验中对两个领域的浅薄了解,如有谬误还请批评斧正。

¹ 或您可直接搜索 GitHub 账号 @jstar0 了解更多

² 您请注意,本模板基于 LPPL v1.3c 分发,本项目在原模板Polytechnic University of Leiria: \LaTeX Thesis Template的基础上进行了合法地大量二改,包括但不限于自定义风格、中文化支持、样式重定义、功能增加等。

³ 模板提供两种样式,一种为学术论文样式,另一种为实验报告样式,具体区别请检查 GitHub 仓库上的两个分支。

目录

目录	ii
1 Shell 的基本用法	2
1.1 Shell 简介	2
1.1.1 概述	2
1.1.2 Shell 的类型	2
1.2 Shell 基本指令	3
1.2.1 基本指令概述	3
1.2.2 实例演示	3
1.3 数据管道与重定向	5
1.3.1 何谓数据管道与重定向	5
1.3.2 管道与重定向操作概述	5
1.3.3 联用组合命令	6
1.3.4 实例演示	6
1.4 基本的 Shell 脚本编写	7
1.4.1 Shell 脚本概述	7
1.4.2 Shell 脚本基本语法	7
1.4.3 实战：编写一个垃圾清理脚本	8
2 数据整理	10
2.1 概述	10
2.2 数据筛选	10
2.2.1 grep 命令	10
2.2.2 sed 命令	10
2.2.3 awk 命令	11
2.3 数据的排序与去重	11
2.3.1 sort 命令	11
2.4 Shell 中的正则表达式 *	11
3 Vim 的使用	12
3.1 Vim 简介与类似工具	12
3.1.1 Vim 简介	12
3.1.2 类似工具	12
3.2 Vim 的基本操作	12
3.3 完成 Vimtutor	13

SHELL 的基本用法

1.1 Shell 简介

1.1.1 概述

Shell 是计算机中一种重要的命令行解释器，其主要功能是接收用户输入的命令，并将其传递给操作系统内核进行处理。实际上，它提供了与操作系统交互的接口，使得用户能够通过命令行完成各种任务，如文件管理、系统配置和进程控制。**Shell** 的发展历史悠久，其在 **Unix** 和类 **Unix** 发行版中都扮演着关键角色。

从工作原理上看，**Shell** 通过解析用户输入的命令，将其转化为内核能够理解的系统调用，进而控制系统资源完成任务。其基本操作模式是交互式的：用户输入一条命令后，**Shell** 会对其进行解析，并决定如何执行。若该命令是系统内置的 *Shell* 命令，则直接调用 *Shell* 的内置函数处理；若是外部命令，则通过查找系统中的可执行文件，调用相应的程序。用户与 **Shell** 的这种互动一般通过命令行界面 (CLI) 进行，每次操作都包括输入、执行和输出的完整流程。

更进一步地，**Shell** 还支持脚本编写功能，是更高级的定制和自动化，用户可以将一系列命令写入脚本文件，**Shell** 会按顺序逐条执行。这是 **Shell** 作为系统管理和自动化任务中最重要工具的原因，也正因此，用户可以轻松实现批量文件处理、系统监控以及软件安装等复杂任务。

1.1.2 Shell 的类型

在实际应用中，**Shell** 有多种类型，其中最常见的包括 *Bash* (*Bourne Again Shell*)、*Zsh* (*Z Shell*) 和 *Fish* (*Friendly Interactive Shell*) 等。

Bash 是目前最广泛使用的 **Shell** 类型，作为 **GNU** 项目的一部分，不仅兼容早期的 **Bourne Shell**，还增加了许多新功能，如命令行自动补全和历史记录机制。**Zsh** 则是一种功能更强大且高度定制化的 **Shell**，继承了 **Bash** 的所有特性，同时引入了更为先进的自动补全、插件支持以及更灵活的脚本编写能力。**Fish** 则以简单易用著称，设计目标是为用户提供更直观的命令行体验，尤其适合新手。

这些不同的 **Shell** 工具，实际上都在不同的类 **Unix** 发行版中被使用，不同的 **Linux** 发行版对 **Shell** 的支持也存在差异。大多数现代 **Linux** 发行版，如 *Ubuntu*、*Debian*、*CentOS* 等，都默认使用 *Bash* 作为其标准的命令解释器。然而，有些发行版，如 *Arch Linux* 和 *macOS*，在提供 *Bash* 的同时，也预装了 *Zsh* 或其他类型的 *Shell*。发行版选择 **Shell** 的标准通常基于其稳定性、功能性和用户的使用习惯。由于开源系统的性质，用户可以根据个人偏好，在系统中安装并切换不同的 **Shell**。

1.2 Shell 基本指令

1.2.1 基本指令概述

下表中的 **Shell** 指令，是我在大一时学习 *Missing Semester* 课程时总结的，也包含了我在 **Linux** 服务器运维的一些经验。具体见表1.1。¹

Name	Usage	Details
ls	ls -l: long format	List directory contents
man	Shows you its manual page	Displays the manual page of a command for detailed help
cp	Copy files or directories	cp is used to copy one or more files to another location. It can also copy directories when used with -r .
mkdir	Make directory	Creates a new directory with the specified name
rmdir	Can only remove empty directory	Deletes a directory, but only if it is empty. For non-empty directories, rm -r is required.
rm	rm -r: remove directories and their contents recursively	Removes files or directories. Be careful when using rm -r as it will delete the directory and all of its contents recursively.
su	sudo su: switch to root	Switches user (usually to root). su allows the user to assume another user's identity, typically root.
sudo	Do something as superuser ("root")	Allows a permitted user to execute a command as the superuser or another user. Commonly used for administrative tasks.
tee	-a = --append (not overwrite)	Reads from standard input and writes to both standard output and files. For example, echo 3 sudo tee brightness modifies a file with the new input.
cat	Concatenate and display file contents	Concatenates files and displays their content. Often used for reading files.
tail	tail -n 10: display the last 10 lines	Displays the last part of a file, often useful for viewing logs. The -f option can be used to follow updates in real-time.

表 1.1: 常用 Shell 指令

特别的，**ls** 指令有现代的衍生版本，如 **eza**，可以提供更多的功能和更好的用户体验。

1.2.2 实例演示

下面是一些常见的 **Shell** 指令的实例演示，以更好地理解其用法。所有演示均在我的 *Ubuntu 22.04.04 LTS Azure SG Server* 上完成。

¹ 由于当时接触 MIT - Missing Semester 我更偏向于使用英文，所以这里的表格也是根据当时使用英文编写的完善。

第一组示例，我们将演示基本的文件操作。请见代码1.1。

```
1 # 确认当前目录
2 pwd
3 ## 输出: /home/JStar0Y
4
5 # 列出当前目录下的所有文件
6 ls
7 # 如果想要更加详细的信息，可以使用 -l 选项；至于 -a 选项，可以显示隐藏文件
8 ls -al
9 ## drwxr-x--- 5 JStar0Y JStar0Y 4096 Sep 13 03:27 .
10 ## drwxr-xr-x 3 root root 4096 Jul 8 17:18 ..
11 ## -rw-r--r-- 1 JStar0Y JStar0Y 807 Jan 6 2022 .profile
12 ## drwx----- 2 JStar0Y JStar0Y 4096 Jul 8 17:18 .ssh
13 ## ... 其他文件省略 ..
14
15 # 创建一个新的目录
16 mkdir test
17
18 # 移动到新创建的目录
19 cd test
20
21 # 创建一个新的文件
22 touch test.txt
23
24 # 删除刚才创建的文件
25 rm test.txt
26 # 如果要删除非空文件夹，需要使用 -r 选项 (recursively)
27
28 # 返回上一级目录
29 cd ..
30
31 # 删除刚才创建的目录（如果非空则无法使用 rmdir）
32 rmdir test
```

代码 1.1: Shell 常用指令实例一：文件操作

第二组演示，我们将展示如何使用 `sudo` 和 `su` 命令。请见代码1.2。

```
1 # 查看当前用户
2 whoami
3 ## 输出: JStar0Y
4
5 # 使用 sudo 切换到 root 用户
6 sudo su
7 # 此处我因为我修改过 sudoers 文件，所以不需要输入密码
8 # 更一般的操作系统，需要输入当前用户的密码（前提是当前用户拥有 sudo 权限）
9
10 # 查看当前用户
11 whoami
12 ## 输出: root
13
14 # 退出 root 用户
15 exit
16
17 # 使用 su 切换到 root 用户
18 su
19 # 输入 root 用户的密码后，切换成功
20 # 一般的，su 后面可以跟用户名，用于切换到其他用户，密码也是对应的用户密码
```

代码 1.2: Shell 常用指令实例二：sudo 和 su 命令

第三个示例，我们将展示如何使用 `tee`、`tail` 和 `cat` 命令。请见代码1.3。


```
1 # 使用 tee 命令将输入写入文件
2 echo "Hello, World!" | sudo tee test.txt
3
4 # 使用 cat 命令查看文件内容
5 cat test.txt
6
7 # 使用 tee 命令追加内容
8 echo "Hello, World Again!" | sudo tee -a test.txt
9
10 # 使用 tail 命令查看文件末尾内容
11 tail test.txt
12 # 如果想要实时查看文件内容, 可以使用 -f 选项
13 # 此处的 & 用于将进程放入后台, 以便继续输入其他命令
14 tail -f test.txt &
15 # 此时, 对文件修改
16 echo "Hello, World Again and Again!" | sudo tee -a test.txt
17 ## 输出: Hello, World Again and Again!
18 ## 因为后台有 tail 命令, 所以会实时显示文件内容
19
20 # 结束 tail 命令
21 kill %1
```

代码 1.3: Shell 常用指令实例三: tee 和 cat 命令

此处需要注意, | 左右两侧的命令不可以互换。关于这点, 我们在第1.3节中马上介绍。

1.3 数据管道与重定向

1.3.1 何谓数据管道与重定向

数据管道是 Shell 中的一种重要概念, 它允许将一个命令的输出作为另一个命令的输入, 从而实现多个命令之间的数据传递。数据管道的符号是 |, 它将两个命令连接在一起, 使得第一个命令的输出成为第二个命令的输入。例如, `ls | grep .txt` 将列出当前目录下的所有文件, 并将包含 `.txt` 的文件名过滤出来。

而重定向是 Shell 中的另一种重要概念, 允许将命令的输入和输出重定向到文件。重定向的符号有 `<` 和 `>`, 分别表示输入重定向和输出重定向。例如, `ls > files.txt` 将列出当前目录下的所有文件, 并将结果保存到 `files.txt` 文件中。

1.3.2 管道与重定向操作概述

类似的, 该部分也是我在大一时学习 *Missing Semester* 课程时总结的, 是根据当时使用英文编写的完善。

Operations like |, >, and < are done *by the shell*, not by the individual program:

1. pipe: | **take the output** of the program to the *left* and **make it the input** of the program to the *right*.
2. < indicates rewire the **input** for this program to be the **contents of the file** (*READ FILE*)
> means rewire the **output** of the preceding program **into this file** (*WRITE FILE*)
>> is like >, but **append to a file**.

特别的，| 两侧的权限是不同的，如果在左侧使用 `sudo`，右侧的命令不会继承 `sudo` 的权限。

1.3.3 联用组合命令

Shell 中的管道和重定向操作可以联用其他工具，以实现更复杂的数据处理需求。例如，`ls | grep .txt > files.txt` 将列出当前目录下的所有文件，并将包含 `.txt` 的文件名保存到 `files.txt` 文件中。

这种联用组合命令的方式，可以实现更多的功能，如文件搜索、数据过滤、日志分析等。根据具体需求，将多个命令组合在一起，就实现了更为复杂的数据处理任务。

1.3.4 实例演示

依然如1.2.2节，我们将展示如何使用管道和重定向操作，并联用组合命令。

第一组示例，我们将演示如何使用基本的管道操作。请见代码1.4。

```
1  # 列出当前目录下的所有文件，并将包含 .txt 的文件名过滤出来
2  ls | grep .txt
3
4  # 写入 ./test/test.txt 文件一些内容
5  echo "Hello, World!" > ./test/test.txt
6
7  # 追加内容
8  echo "Hello, World Again!" >> ./test/test.txt
9
10 # 查看文件内容
11 cat ./test/test.txt
12 ## 输出: Hello, World!
13
14 # 查看文件内容，并将包含 World 的行过滤出来
15 cat ./test/test.txt | grep World
16 ## 输出: Hello, World!
17 ##      Hello, World Again!
18
19 # 然而，如果再使用 >，会覆盖原有文件内容
20 echo "Hello, World, Twice!" > ./test/test.txt
21 cat ./test/test.txt
22 ## 输出: Hello, World, Twice!
23 ## 这告诉我们，如果要追加内容，需要使用 >>
24
25 # 从文件中读取内容
26 cat < ./test/test.txt
27 # 其实，这个命令等价于 cat ./test/test.txt
28 # 但是这里使用了输入重定向的用法
29
30 # 从文件中读取内容，并将包含 World 的行过滤出来
31 cat < ./test/test.txt | grep World
32 ## 输出: Hello, World, Twice!
33
34 # 从一个文件中读取内容，并将其写入另一个文件，并查看另一个文件内容
35 cat < ./test/test.txt > ./test/test2.txt && cat ./test/test2.txt
36 # 注意此处 && 的用法在部分 Linux 发行版中可能会有不同
```

代码 1.4: Shell 管道和重定向操作实例一

第二组示例，我们将展示如何使用联用组合命令，主要展示 `grep`，`awk`，`sed` 这几个工具。请见代码1.5

```
1 # 使用 grep 过滤文件内容
2 cat /etc/passwd | grep root
3 ## 将输出系统中存储的所有包含 root 的行，即
4 ## root:x:0:0:root:/root:/bin/bash
5
6 # 使用 awk 过滤文件内容
7 # awk options 'pattern {action}' file
8 cat /etc/passwd | awk -F: '{print $1}'
9 # 这里的 -F: 表示使用 : 作为分隔符
10 # $1 表示第一个字段，即用户名
11 ## 将输出系统中存储的所有用户名
12
13 # 使用 sed 过滤文件内容
14 # sed [-hnV][-e<script>][-f<script 文件>][文本文件]
15 cat /etc/passwd | sed -n '1,3p'
16 # 这里的 -n 表示只输出匹配的行
17 # 1,3p 表示输出第 1 行到第 3 行
18 ## 将输出系统中存储的前三行内容，即
19 ## root:x:0:0:root:/root:/bin/bash
20 ## daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
21 ## bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

代码 1.5: Shell 管道和重定向操作实例二

1.4 基本的 Shell 脚本编写

1.4.1 Shell 脚本概述

Shell 脚本强大的工具，广泛应用于自动化任务和系统管理。脚本语言拥有简洁和灵活的特性，用户能够快速编写脚本来完成复杂的任务。编写脚本的基础是掌握其基本语法结构、变量处理、条件判断与循环等操作，这些是实现功能的核心。

1.4.2 Shell 脚本基本语法

变量的定义

Shell 脚本中，变量的使用和运算非常重要。变量无需预先声明，可以直接通过赋值来创建。例如，通过 `my_var=5` 可以定义一个名为 `my_var` 的变量，其值为 5。

特别需要注意的是，变量在 Shell 中默认为字符串形式，进行数值运算时，必须通过 `expr` 或 `$(())` 等方式进行。例如，`result=$((my_var + 2))` 才是正确的。

脚本中，还可以通过命令替换，将命令的输出赋值给变量，例如 `current_date=$(date)` 会将当前日期赋值给 `current_date` 变量，而其中的 `$(date)` 是一个命令替换操作，会将 `date` 命令的输出作为变量的值。

条件判断

条件判断是脚本控制流中不可或缺的一部分。在 Shell 脚本中，*if-else* 语句用于实现逻辑判断。*if* 语句通过 `[]` 或 `test` 命令进行判断操作。

特别地，**Shell** 判断语法中的空格要求非常严格，`[]` 内的表达式必须用空格分隔。我们给出一个简单的例子，如代码1.6所示。

```
1  # 定义一个变量
2  my_var=5
3
4  # 判断变量是否大于 2
5  if [ $my_var -gt 2 ]; then
6      echo "my_var is greater than 2"
7  else
8      echo "my_var is less than or equal to 2"
9  fi
10
11 ## 或者使用 test 命令来进行判断
12 if test $my_var -gt 2; then
13     echo "my_var is greater than 2"
14 else
15     echo "my_var is less than or equal to 2"
16 fi
```

代码 1.6: Shell 脚本基本语法示例一

循环结构

提供两种循环结构：*for* 循环和 *while* 循环。*for* 循环用于遍历一个列表，*while* 循环则用于根据条件执行循环。具体的语法如代码1.7所示。

```
1  # for 循环
2  for i in 1 2 3 4 5
3  do
4      echo "Number: $i"
5  done
6
7  for file in /tmp/*; do
8      echo "发现文件: $file"
9  done
10
11 # while 循环
12 count=0
13 while [ $count -lt 5 ]; do
14     echo "Count: $count"
15     count=$((count + 1))
16 done
```

代码 1.7: Shell 脚本基本语法示例二

1.4.3 实战：编写一个垃圾清理脚本

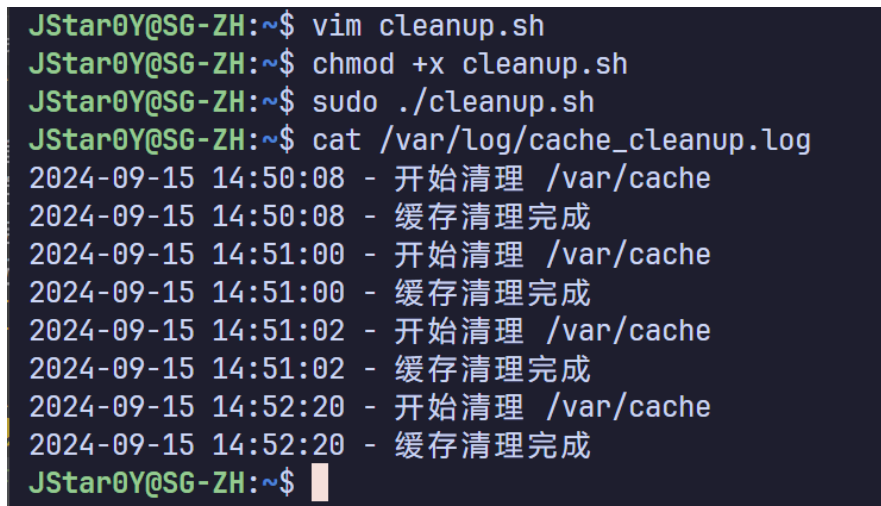
首先建立基本的思路——脚本应该检查缓存目录是否存在，如果不存在则输出错误日志并退出。接着，使用 `rm -rf` 命令清空缓存目录，并将结果记录到日志文件中。通过将 *date* 命令与变量结合，我们能够在日志中记录清理操作的时间戳。具体实现见代码1.8。

```
1  #!/bin/bash
2
3  # 定义变量
4  CACHE_DIR="/var/cache"
5  LOG_FILE="/var/log/cache_cleanup.log"
6  TIMESTAMP=$(date +%Y-%m-%d %H:%M:%S)
7
8  # 检查缓存目录是否存在
9  if [ ! -d "$CACHE_DIR" ]; then
10     echo "$TIMESTAMP - 错误: 目录 $CACHE_DIR 不存在" >> $LOG_FILE
11     exit 1
12 fi
13
14 # 清理缓存
15 echo "$TIMESTAMP - 开始清理 $CACHE_DIR" >> $LOG_FILE
16 rm -rf $CACHE_DIR/*
17
18 # 确认清理成功
19 if [ $? -eq 0 ]; then
20     echo "$TIMESTAMP - 缓存清理完成" >> $LOG_FILE
21 else
22     echo "$TIMESTAMP - 错误: 清理失败" >> $LOG_FILE
23     exit 1
24 fi
```

代码 1.8: Shell 脚本实战: 垃圾清理脚本

将上述脚本保存在 `cleanup.sh` 文件中, 然后通过 `chmod +x cleanup.sh` 命令添加执行权限。接着, 运行 `./cleanup.sh` 命令, 即可执行脚本。或者, 不添加权限, 我们直接使用 `bash cleanup.sh` 命令执行脚本。

直接运行我们会发现, 权限不足, 此时我们需要使用 `sudo` 命令来执行脚本。执行结果见图1.1。



```
JStar0Y@SG-ZH:~$ vim cleanup.sh
JStar0Y@SG-ZH:~$ chmod +x cleanup.sh
JStar0Y@SG-ZH:~$ sudo ./cleanup.sh
JStar0Y@SG-ZH:~$ cat /var/log/cache_cleanup.log
2024-09-15 14:50:08 - 开始清理 /var/cache
2024-09-15 14:50:08 - 缓存清理完成
2024-09-15 14:51:00 - 开始清理 /var/cache
2024-09-15 14:51:00 - 缓存清理完成
2024-09-15 14:51:02 - 开始清理 /var/cache
2024-09-15 14:51:02 - 缓存清理完成
2024-09-15 14:52:20 - 开始清理 /var/cache
2024-09-15 14:52:20 - 缓存清理完成
JStar0Y@SG-ZH:~$
```

图 1.1: Shell 脚本实战: 垃圾清理脚本执行结果

这种自动化缓存清理脚本也可以通过 `cron` 定时任务配置定期运行, 可以有效简化系统运维工作量。

数据整理

2.1 概述

本部分致力于练习数据整理。

数据量很大的情况下，如何高效地对数据进行清理、排序、去重、提取关键信息成为了不可忽视的挑战。Shell 作为系统中操作文件和数据的核心工具，具备强大的数据处理能力。通过内置命令和管道机制，Shell 能够灵活应对各种数据整理任务。我们主要探究 `grep`、`awk`、`sed`、`sort` 和 `uniq` 等工具，以实现数据清洗。

2.2 数据筛选

2.2.1 `grep` 命令

`grep` 是用于数据筛选的首选工具。能够根据给定的模式（通常是正则表达式）从文件或数据流中提取匹配的行。支持多种选项，如 `-i` 用于忽略大小写，`-v` 用于反向匹配，`-r` 则可以递归地搜索目录。一个使用 `grep` 筛选包含“error”关键字的日志文件的例子，见代码2.1。

```
1 grep "error" /var/log/syslog
```

代码 2.1: 使用 `grep` 筛选日志文件

2.2.2 `sed` 命令

`sed` 则是用于文本处理的强大工具，适合执行替换、删除、插入等操作。与 `grep` 不同，`sed` 可以直接修改文件中的内容，例如，将文件中的所有 `foo` 替换为 `bar`，见代码2.2。

```
1 sed 's/foo/bar/g' input.txt
```

代码 2.2: 使用 `sed` 替换文本

2.2.3 awk 命令

awk 则特别适合处理结构化数据，它能够基于字段操作文本，例如根据某一列的值筛选、打印或统计结果。代码2.3展示了如何使用 **awk** 输出文件的第二列内容。

```
1 awk '{print $2}' data.txt
```

代码 2.3: 使用 *awk* 输出第二列内容

2.3 数据的排序与去重

2.3.1 sort 命令

sort 命令可以根据字典序或数值大小对数据进行排序，且能够根据指定字段进行排序。代码2.4展示了如何按照文件的第三列进行数值排序。

```
1 sort -k 3n data.txt
```

代码 2.4: 使用 *sort* 按第三列进行数值排序

sort 还可以与 **uniq** 命令配合使用，用于去重和统计。**uniq** 应该在排序后使用，只能处理相邻的重复项。代码2.5展示了如何使用 **sort** 和 **uniq** 去除重复行。

```
1 sort data.txt | uniq
```

代码 2.5: 使用 *sort* 和 *uniq* 去除重复行

此外，**uniq** 还提供了 **-c** 选项，还可以返回重复的次数并输出统计结果。

2.4 Shell 中的正则表达式 *

此部分由于需要正则表达式的基础知识，仅作简单介绍。正则表达式是一种用于描述字符串模式的工具，常用于文本处理和匹配。Shell 中的正则表达式主要用于 **grep**，**sed**，**awk** 等工具中，用于匹配和处理文本。例如，**.*** 表示匹配任意字符任意次数，**[0-9]+** 表示匹配一个或多个数字。详细的正则表达式语法和用法需要参考相关资料。

给出一个示例，从日志文件提取所有 IP 地址，见代码2.6。

```
1 grep -Eo '([0-9]{1,3}\.){3}[0-9]{1,3}' /var/log/syslog
2 # 其中，-E 表示启用扩展正则表达式，-o 表示只输出匹配的部分
```

代码 2.6: 使用正则表达式提取 IP 地址

VIM 的使用

3.1 Vim 简介与类似工具

3.1.1 Vim 简介

作为 Unix 系统中的默认编辑器之一，**Vim** 具有极高的灵活性和扩展性。熟练掌握 **Vim** 能够极大提升工作效率，尤其是在编写 *Shell* 脚本时，**Vim** 的各种模式和快捷键可以帮助用户快速完成代码编辑、调试等任务。

Vim 的核心概念在于三种主要模式的切换：**插入模式**、**命令模式**和 **可视模式**。默认情况下，**Vim** 进入的是 **命令模式**，可以在此模式下使用键盘命令来导航、编辑和操作文本。按下 **i** 键，**Vim** 会切换到 **插入模式**，此时用户可以像在普通编辑器中一样输入文本。编辑完成后，按 **Esc** 键可以返回 **命令模式**。可视模式通过按下 **v** 键进入，可以在该模式下选择文本区域，进行复制、删除等操作。掌握这三种模式的切换，是高效使用 **Vim** 的基础。

3.1.2 类似工具

除了 **Vim** 之外，还有一些其他的文本编辑器也具有类似的模式切换机制，如 **Emacs**、**Nano** 等。这些编辑器各有特点，用户可以根据自己的需求选择适合的编辑器。本文中，我们主要关注 **Vim** 的使用。

3.2 Vim 的基本操作

在文本编辑过程中，**Vim** 提供了一系列强大的命令用于快速操作。在命令模式下，**x** 用于删除当前光标处的字符，**dd** 则删除整行，而 **y** 代表复制（*Yank*），配合 **p** 命令可以将复制的内容粘贴到光标处。这些快捷键极大简化了文本的操作，无需离开键盘就可以完成复制、粘贴和删除等常见任务。

Vim 的操作是可组合的，例如 **3dd** 会一次性删除三行，**yy** 可以复制当前整行，而 **yw** 则仅复制光标所在单词。

在 *Shell* 脚本编写中，**Vim** 的高效性尤为突出。*Shell* 脚本往往涉及大量的重复操作、格式化和调试需求，**Vim** 的命令模式可以帮助用户快速调整代码结构。比如，使用 **/** 键进入

搜索模式，可以快速查找脚本中的关键字；使用 `:s` 命令进行文本替换，可以迅速修改变量名或函数名。例如代码3.1展示了如何使用 **Vim** 替换变量名。

```
1 :%s/old_variable/new_variable/g
```

代码 3.1: 使用 Vim 替换变量名

命令将当前文件中所有出现的 `old_variable` 替换为 `new_variable`，并且通过添加 `g` 标志实现全局替换。类似的命令可以应用于快速修复或批量修改代码中的特定部分，极大提升了脚本编写的效率。

Vim 的分屏和标签页功能也是编写脚本时的常用工具。分屏功能允许用户在同一窗口中同时查看多个文件或多个不同部分的代码。通过 `:vsp` 命令可以实现垂直分屏，而 `:sp` 命令则用于水平分屏。使用 `Ctrl-w` 快捷键可以快速在不同窗口之间切换。此外，**Vim** 还支持多文件编辑，通过 `:tabnew` 命令打开新标签页，可以在多个文件之间快速切换。

3.3 完成 Vimtutor

这个简单而可互动的教程，给我们学习了解 **Vim** 提供了很好的机会。我已经基本完成了这部分，如图3.1所示。



图 3.1: Vimtutor 学习

