
MOS 操作系统实验 指导书

操作系统课程实验任务及相关说明

带你体验自己动手完成一个小操作系统的乐趣



*SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
BEIHANG UNIVERSITY*

2023 年 2 月 26 日

POWERED BY L^AT_EX

引言	1
引言	1
实验内容	2
实验设计	2
实验环境	3
虚拟机平台	4
Git 服务器	4
自动评测	5
0 初识操作系统	6
0.1 实验目的	6
0.2 初识实验	6
0.2.1 了解实验环境	6
0.2.2 通过网站远程访问本实验环境	7
0.2.3 命令行界面 (CLI)	7
0.3 基础操作介绍	8
0.3.1 命令行	8
0.3.2 Linux 基本操作命令	8
0.4 实用工具介绍	11
0.4.1 Vim	11
0.4.2 GCC	12
0.4.3 Makefile	13
0.4.4 ctags	15
0.5 Git 专栏-轻松维护和提交代码	16
0.5.1 Git 是什么?	17
0.5.2 Git 基础指引	18
0.5.3 Git 文件状态	20

0.5.4	Git 三棵树	22
0.5.5	Git 版本回退	23
0.5.6	Git 分支	24
0.5.7	Git 远程仓库与本地	27
0.6	进阶操作	27
0.6.1	Linux 操作补充	27
0.6.2	shell 脚本	30
0.6.3	重定向和管道	32
0.7	实战测试	34
0.8	实验思考	37
A	补充知识	38
A.1	实验目的	38
A.1.1	<code>printk</code> 格式具体说明	38

引言

操作系统是计算机系统中软件与硬件联系的纽带，课程内容丰富，既包含操作系统的基础理论，又涉及实际操作系统的设计与实现。操作系统实验设计是操作系统课程实践环节的集中表现，旨在巩固学生理论课学习的概念和原理，同时培养学生的工程实践能力。一些国内外著名大学都非常重视操作系统的实验设计，例如麻省理工学院的 Frans Kaashoek 等设计的 JOS 和 xv6 教学操作系统、哈佛大学的 David A. Holland 等设计的 OS161 操作系统用于实现操作系统实验教学。

我们尝试了 MINIX、Nachos、Linux、Windows 等操作系统实验，发现以 Linux 和 Windows 为基础的实验，由于系统规模庞大，很难让学生建立起完整的操作系统概念，导致专业知识碎片化，不符合系统能力培养目标。此外，操作系统涉及硬件的很多相关知识，本身还包含并发程序设计等比较难理解的概念，因此学生的学习曲线很陡峭，如何让学生由浅入深，平滑地掌握这些知识是实验设计的难点。本书设计实验的基本目标是：一学期内设计实现一个可在实际硬件平台上运行的小型操作系统，该系统具备现代操作系统特征（如虚存管理、多进程等），符合工业标准。

基于系统能力培养的理念和目标希望构建计算机组成原理、操作系统和编译原理等课程的一体化实验体系，因而本书操作系统课程设计采用了计算机组成原理课程中的 MIPS 指令系统（MIPS R3000）作为硬件基础，参考 JOS 的设计思路、方法和源代码，实现一个可以在 MIPS 平台上运行的小型操作系统，包括操作系统启动、物理内存管理、虚拟内存管理、进程管理、中断处理、系统调用、文件系统、Shell 等主要操作系统主要功能。为了降低学习难度，采用增量式实验设计思想，每个实验包含的内核代码量（C、汇编、注释）在几百行左右，提供代码框架和代码示例。每个实验可以独立运行和评测，但是后面的实验依赖前面的实验，学生实现的代码从 Lab1 贯穿到 Lab6，最后实现一个完成的小型操作系统。

实验内容

本书设计的操作系统实验分为 6 个实验 (Lab1 ~ Lab6)，目标是在一学期内自主开发一个小型操作系统。各个实验的相互关系如图 1 所示，具体实验内容如下。

1. **内核、启动和 printf**: 通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写裸机代码。
2. **内存管理**: 理解虚拟内存和物理内存的管理，实现操作系统对虚拟内存空间的管理。
3. **进程与异常**: 通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。
4. **系统调用与 fork**: 掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。
5. **文件系统**: 通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。
6. **管道与 shell**: 实现具有管道，重定向功能的命令解释程序 shell，能够执行一些简单的命令。最后将 6 部分链接起来，使之成为一个能够运行的操作系统。

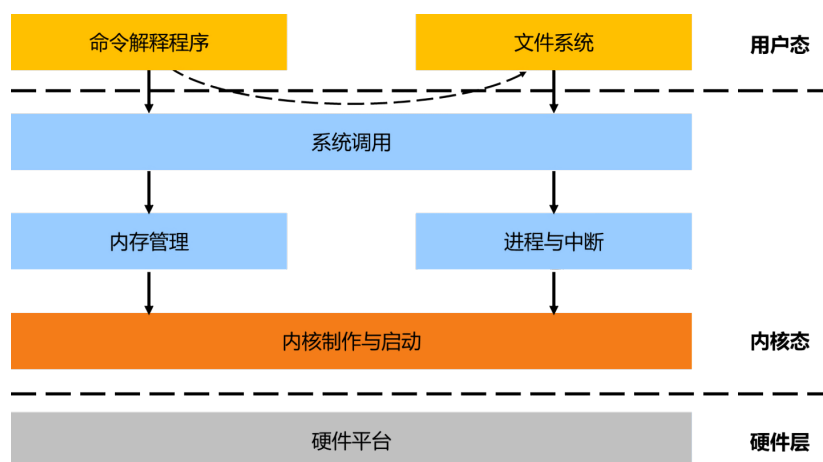


图 1: 实验内容的关系

另外，考虑有些学生对 Linux 系统、GCC 编译器、Makefile 和 git 等工具不熟悉，专门设置了一个 Lab0，主要介绍 Linux、Makefile、git、vi 和仿真器的使用以及基本的 shell 编程等，为后续实验的顺利实施打好基础。

实验设计

由于开发一个实际的操作系统难度大、工作量繁重，为了保证教学效果，在核心能力部分采用微内核结构和增量式设计的原则，因此可以从最基本的硬件管理功能逐步扩充，最后完成一个完整的系统。实验内容的设计满足以下条件。

1. 每个实验可独立运行与测试，便于调试与评测，可获得阶段性成果。
2. 每个实验内容包含相对独立的知识点，并只依赖其前序实验。
3. 基本保证在两周内完成一个实验，这样在一学期内可以完成整个实验。
4. 各个实验提交的代码一直伴随整个实验过程，可以不断改进、完善代码。

整个系统结构如图 2 所示，蓝色部分是每次实验需要新增加的模块，绿色部分是需要修改完善的模块，灰色部分是不用修改的模块。在增量式设计下，可以从基本的功能出发，逐步完善整个系统，从而降低了学习操作系统的难度。

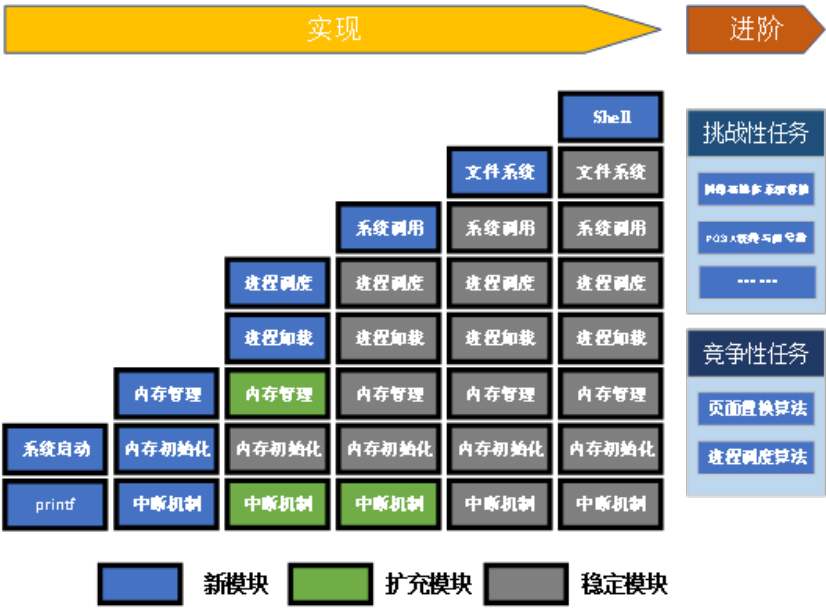


图 2: 增量式实验方法

为了适应不同读者的学习要求，本书的实验采用分层的方式，从基础到复杂逐步实现实验的基本目标。因此，可将实验基本目标分为三个层次：

- 第一层次，掌握基本的系统使用与编程能力：包括Linux、Makefile、git、vi 和仿真器的使用，基本的 shell 编程和使用系统调用编程；
- 第二层次，掌握操作系统核心能力：包括 6 个实验，从操作系统内核构造、内存管理、进程管理、系统调用、文件系统和命令解释程序，构成一个完整的小型操作系统；
- 第三层次，锻炼操作系统提升能力：主要包括若干挑战性任务，学生需要独立在某一方面实现若干新的系统功能。

实验环境

一次实验的整个流程包括，初始代码发布、代码编写、调试运行、代码提交、编译测试以及评分结果的反馈。为了方便学生和教师，我们设计了操作系统实验集成环境，采

用 git 进行版本管理，保证学生之间的代码互不可见，而教师和助教可以方便地查看每位学生的代码。整个环境的结构如图 3 所示。为了满足实验需求，整个系统分为以下几个部分。

- a) 虚拟机平台，包含实验需要的开发环境，例如 Linux 环境、交叉编译器、MIPS 仿真器等。
- b) git 服务器，包括学生各个实验的代码以及相关信息。
- c) 自动评测和反馈，这部分集成在 git 服务器中，后面会详细介绍。

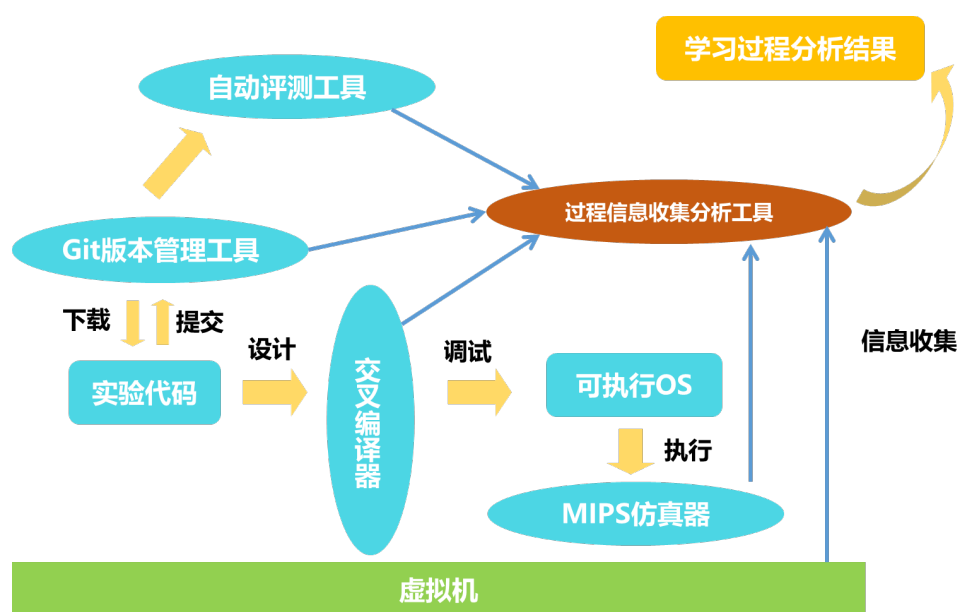


图 3: 操作系统实验集成环境

虚拟机平台

为了方便地收集学习过程数据，同时尽量降低学生端设备的要求，我们提供虚拟机作为实验后端，虚拟机中使用 Linux 系统，方便部署环境，整个实验过程在虚拟机中完成。在虚拟机中，需要部署相应仿真器、编译器、文件编辑器等环境。推荐使用 GXemul 作为仿真器，MIPS 的 gcc 交叉编译器作为编译器，vim 作为文件编辑器。这些工具已经部署在虚拟机环境中，避免了软件版本冲突问题，方便自动评测系统的部署与实施。同时，节省安装实验环境花费的时间。每位学生可以登录到实验系统，完成实验。

Git 服务器

为了保证学生代码安全，同时提供方便的代码版本管理工具，提供一个 git 服务器，每位学生的代码编写过程在虚拟机中完成，同时将代码托管在 git 服务器中，避免了故障导致的损失。同时，实验发布及测试结果反馈均通过 git 服务器完成。在 git 服务器中，每个学生拥有一个独立的代码库，对于每位学生来说，只有自己的代码库是可见的，

每位学生可以随时下载和提交自己代码库中的代码。同时，所有学生的代码库对于助教和教师是可见的，所有的助教和教师都可以下载学生代码。

自动评测

由于学生人数众多，对于学生实验代码的评判是一个繁复、机械化的过程，助教和教师手动评测非常困难。自动评测系统能对学生提交的代码自动给出相应的评分。当学生执行代码提交后，可提交评测。评测系统将获取学生代码，依次完成编译、运行和测试，给出评测结果反馈结果给学生查看。

CHAPTER 0

初识操作系统

0.1 实验目的

1. 认识操作系统实验环境。
2. 掌握操作系统实验所需的基本工具。

在本实验中，需要了解实验环境，熟悉 Linux 操作系统 (Ubuntu)，了解控制终端，掌握一些常用工具并能够脱离可视化界面进行工作。本实验难度不大，重点是熟悉操作系统实验环境的各类工具，为后续实验的开展奠定基础。

0.2 初识实验

“工欲善其事必先利其器”，应对我们的环境和工具有足够的了解，才能顺利开展实验工作。

0.2.1 了解实验环境

实验环境整体配置如下：

- 操作系统：Linux 内核，Ubuntu 操作系统
- 硬件模拟器：GXemul
- 编译器：GCC
- 版本控制：Git

Ubuntu 操作系统是一款开源的 GNU/Linux 操作系统，它基于 Linux 内核实现，是目前较为流行的 Linux 发行版之一。GNU（GNU is Not Unix 的递归缩写）是一套开源计划，其中包含了三个协议条款，为用户提供了大量开源软件；而人们常说的 Linux，

从严格意义上是指 Linux 内核，基于该内核的操作系统众多，具有免费、可靠、安全、稳定、多平台等特点。

GXemul 是一款计算机架构仿真器，可以模拟所需硬件环境，例如本实验需要的 MIPS 架构下的 CPU。

GCC 是一套免费、开源的编译器，诞生并服务于 GNU 计划，最初名称为 GNU C Compiler，后来因支持更多编程语言而改名为 GNU Compiler Collection，很多集成开发环境 (Integrated Development Environment, IDE) 的编译器用的就是 GCC 套件，例如 Dev-C++，Code::Blocks 等。本实验将使用基于 MIPS 的 GCC 交叉编译器。

Git 是一款免费、开源的版本控制系统，本书实验将利用它来提供管理、发布、提交、评测等功能。第 0.5 节将会详细介绍 Git 如何使用。



图 0.1: Ubuntu, GNU, Linux

0.2.2 通过网站远程访问本实验环境

为了方便访问，本实验提供通过网站远程访问虚拟机的方法，不需要手动进行复杂的配置。可直接按照下面的流程，进入和上文下相同的界面。

本书实验的虚拟环境的网站是<http://lab.os.buaa.edu.cn>，如果在校外的话需要先使用 EasyConnect 连上北航 VPN 再进行访问。

0.2.3 命令行界面 (CLI)

对于目前主流的操作系统，如 Windows、Mac OS、Ubuntu 等均使用图形用户界面 (Graphical user interface, GUI)，但是在操作系统课程中，需要了解命令行界面 (Command line Interface, CLI)。

其实，上文关于“操作系统”的描述并不严谨，用户接触的并不是真正的“Ubuntu 操作系统”，而是它的“壳” (shell)。一般我们将操作系统核心部分称为内核 (kernel)，与其相对的是它最外层的“壳” (shell) 即为命令解释程序，是访问操作系统服务的用户界面。操作系统“壳”有命令行界面 (CLI) 或图形用户界面 (GUI) 两种形式。上文提到的黑色界面就是命令行界面，它是纯文本界面，用于接收、解释、执行用户输入的命令，完成相应的功能。

在 Ubuntu 中，命令行界面默认的 Shell 是 bash，它也是一款基于 GNU 的免费、开源软件。

0.3 基础操作介绍

0.3.1 命令行

在命令行界面 (Command Line Interface, CLI) 中, 用户或客户端通过单条或连续命令行 (command line) 的形式向程序发出命令, 从而达到与计算机程序进行人机交互的目的。在 Linux 系统中, 命令用于对 Linux 系统进行管理, 其一般格式为: 命令名 [选项] [参数]。其中, 方括号表示可选, 意为可根据需要选用, 例如 `ls -a directory`。shell 命令有两种类型: shell 内建命令和外置程序。

下面介绍 Linux 的基本操作命令, 熟悉此部分内容的读者可跳过此节。

0.3.2 Linux 基本操作命令

进入终端后, 首先会看到光标前的如下内容。

```
1 开始连接到 git@xxx.xxx.xxx.xxx 0.1
2 Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-46-generic x86_64)
3
4      * Documentation:  https://help.ubuntu.com
5      * Management:    https://landscape.canonical.com
6      * Support:        https://ubuntu.com/advantage
7 Last login: Wed Jan  4 12:50:58 2023 from 10.134.170.231
8 git@21xxxxxx:~$
```

其中 @ 符号前的是用户名, @ 符号后的是计算机名, 冒号后为当前所在的文件目录 (/表示根目录, ~ 表示家目录, 家目录即 /home/<user_name>), 最后 \$ 或 # 分别表示当前用户为普通用户或超级用户 root。然后通过键盘输入命令, 按回车后即可执行相应命令。

要想知道当前目录中包含哪些文件, 可使用 `ls` 命令, 其输出信息可以通过彩色加亮显示, 来区分不同类型的文件, `ls` 是使用率较高的命令, 其详细信息如图所示。一般情况下, 该命令的参数省略则默认显示该目录下所有文件, 所以只需使用 `ls` 即可看到所有非隐藏文件, 若要看到隐藏文件则需要加上 `-a` 选项, 若要查看文件的详细信息则需要加上 `-l` 选项。

```
1  ls
2  用法:ls [选项]... [文件]...
3  选项 (常用):
4  -a          不隐藏任何以 . 开始的项目
5  -l          每行只列出一个文件
```

针对该命令, 一般只会用到 `ls`, 以及 `ls -a` 和 `ls -l` 三种形式。现在尝试在命令行输入 `ls` 后, 按下回车会出现一个命名为学号的目录, 这就是读者进行实验的目录。如果想尝试创建一个新的空文件, 可以用 `touch` 命令。

```
1  touch
2  用法:touch [选项]... [文件名]...
```

输入 `touch hello_world.c`, 即可在该目录下成功创建一个新的文件, 再输入 `ls`, 就可看到 `hello_world.c` 文件了, 可以试一试 `ls -a` 和 `ls -l` 命令进行操作。关于如何编辑 `hello_world.c` 文件, 将在下一节关于 Vim 使用工具的介绍中展开。

为了通过目录对文件来进行组织和管理, 可以使用 `mkdir` 命令创建文件目录, 该命令的参数为创建的新目录的名称, 如 `mkdir newdir` 可创建一个名为 `newdir` 的目录。

```
1 mkdir
2 用法:mkdir [选项]... 目录...
```

现在输入 `mkdir newdir` 就在目录下创建了一个名为 `newdir` 的目录, 读者可以再使用 `ls` 命令查看是否新增了 `newdir` 目录。在这里可使用 `cd` 命令进入 `newdir` 目录。

```
1 cd
2 用法:cd [选项]... 目录
```

在命令行输入 `cd newdir` 命令, 进入 `newdir` 目录。为了返回上一级目录, 可以使用 `..` 命令。在 Linux 系统里 `..` 表示上一级目录, `.` 表示当前目录, 因此在输入 `cd ..` 后, 返回上一级目录, 输入 `cd .` 进入当前所在的目录。

查看 `cd` 目录时, 命令行发生一些变化, 在命令行的 `>` 的左边, 从 `~` 变成了, `~/newdir`, 这个字符串是指当前所在的目录, `~` 表示所登录用户的目录, 输入 `pwd` 可查看当前的绝对路径。同时 `cd` 命令也可直接把要跳转到的目录改为绝对路径, 如 `cd /home` 跳转到 `home` 根目录下 `home` 这个目录。

Note 0.3.1 在需要键入文件名或目录名时, 可以使用 `Tab` 键补足全名。当有多种补足时, 双击 `Tab` 键可以显示所有可能选项。在屏幕上输入 `cd /h` 然后按下 `Tab`, 就会自动补全为 `cd /home`, 如果输入的是 `cd /`, 再按两次 `Tab`, 会看到所有可选项, 和 `ls` 类似。

输入 `rmdir` 可以删除一个空的目录。

```
1 rmdir
2 用法:rmdir [选项]... 目录...
```

如果目录非空, 则使用 `rm` 命令。`rm` 命令可以删除一个目录中的一个或多个文件或目录, 也可以将某个目录及其下属的所有文件及其子目录均删除掉。对于链接文件, 只是删除整个链接文件, 而原有文件保持不变。

```
1 rm
2 用法:rm [选项]... 文件...
3 选项 (常用):
4 -r          递归删除目录及其内容, 如果不加这个命令,
5 删除一个有内容的目录会提示不能删。
6 -f          强制删除。忽略不存在的文件, 不提示确认
```

此外, `rm` 命令还有 `-i` 选项, 这个选项在使用文件扩展名字符删除多个文件时特别有用。使用这个选项, 系统会要求逐一确定是否要删除。这时, 必须输入 `y` 并按回车键, 才能删除文件。如果仅按回车键或其他字符, 文件不会被删除。与之相对应的就是 `-f` 选项, 其作用是强制删除文件或目录, 并不询问用户。使用该选项并配合 `-r` 选项, 可实现递归强制删除, 强制将指定目录下的所有文件与子目录一并删除, 这肯导致灾难性后果。例如 `rm -rf /` 即可强制递归删除全盘文件, 绝对不要轻易尝试!

Note 0.3.2 使用 `rm` 命令要格外小心，因为一旦删除了一个文件，就无法再恢复它。所以，在删除文件之前，最好再看一下文件的内容，确定是否真要删除。一些用户会在家目录下建一个类似回收站的目录，如果要使用 `rm` 命令，先把要删除的文件移到这个目录里，然后再进行 `rm`，一定时间之后再对回收站里的文件进行删除。

了解以上注意事项后，再来尝试使用 `rm`。读者先回到 `~` 目录下，假设要删除开始创建的 `hello_world.c`，首先在该目录下创建一个回收站目录叫 `.trash`，然后把 `hello_world.c` 拷贝到这个目录中，这里需要使用 `cp` 命令，该命令的第一个参数为源文件路径，命令的第二个参数为目标文件路径。

```
1 cp
2 用法:cp [选项]... 源文件... 目录
3 选项 (常用):
4 -r          递归复制目录及其子目录内的所有内容
```

下面我们介绍移动的命令。移动命令为 `mv`，与 `cp` 的用法相似。命令 `mv hello_world.c ~/.trash/` 就是将 `hello_world.c` 移动到 `~/.trash` 这个目录中。此时使用 `ls` 命令可以看到 `hello_world.c` 文件已被移除。

另外，在 Linux 系统中若想对文件进行重命名操作，使用 `mv oldname newname` 命令即可。

```
1 mv
2 用法:mv [选项]... 源文件... 目录
```

最后介绍回显命令 `echo`，如果输入 `echo hello_world`，就会回显 `hello_world`，这个命令看起来像一个复读机的功能，本书后文会介绍一些它的有趣用法。

以上就是 Linux 系统入门级的部分常用操作命令以及这些命令的常用选项，如果想要查看这些命令的其他功能选项或者新命令的详尽说明，可使用 Linux 下的帮助命令——`man` 命令，通过 `man` 命令可以查看 Linux 中的命令帮助、配置文件帮助和编程帮助等信息。

```
1 man - manual
2 用法:man page
3 e.g.
4 man ls
```

以下为几个常用的快捷键可供参考。

- `Ctrl+C` 终止当前程序的执行
- `Ctrl+Z` 挂起当前程序
- `Ctrl+D` 终止输入（若正在使用 Shell，则退出当前 Shell）
- `Ctrl+L` 清屏

其中，如果你写了一个死循环，或者程序执行到一半你不想让它执行了，`Ctrl+C` 是你的很好的选择。`Ctrl+Z` 挂起程序后会显示该程序挂起编号，若想要恢复该程序可以使

用 `fg [job_spec]`, `job_spec` 即为挂起编号, 不输入时默认为最近挂起进程。而如果你写了一个等到识别到 `EOF` 才停止的程序, 你就需要输入 `Ctrl+D` 来当作输入了一个 `EOF`。

对其他内容感兴趣的同学可以自行网络搜索或用 `man` 命令看帮助手册进行学习和了解。以上述内容为基础，接下来讲如何在 Linux 下写代码。

Note 0.3.3 在多数 Shell 中，四个方向键也是有各自特定的功能的：← 和 → 可以控制光标的位置，↑ 和 ↓ 可以切换最近使用过的命令

0.4 实用工具介绍

学会了 Linux 基本操作命令，接下来就可以得心应手地使用命令行界面的 Linux 操作系统了。想要使用 Linux 系统完成工作，仅靠命令行还远远不够。在开始动手阅读并修改代码之前，读者还需要掌握一些实用工具的使用方法。这里首先介绍一种常用的文本编辑器：Vim。

0.4.1 Vim

Vim 被誉为编辑器之神，是程序员为程序员设计的编辑器，编辑效率高，十分适合编辑代码。对于习惯了图形化界面文本编辑软件的读者来说，刚接触 Vim 时一定会觉得非常不习惯非常不顺手，所以下面通过创建一个 `helloworld.c`，来让读者熟悉一下 Vim 的基本操作：

- (1) 创建文件：利用之前学到的 `touch` 命令创建 `helloworld.c` 文件（这时，使用 `ls` 命令，可以发现已经在当前目录下创建了 `helloworld.c` 文件）。
- (2) 打开文件：在命令行界面输入 `vim helloworld.c` 打开新建的文件；
- (3) 输入内容：刚打开文件时，无法向其中直接输入代码，需要按“`i`”键进入插入模式，之后便可以向其中输入 `helloworld` 程序，如下图所示：

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

图 0.2: 写入 `helloworld.c` 中的内容

(4) 保存并回到命令行界面：完成文件修改后，按”Esc”回到命令模式，再按下”:”进入底线命令模式，此时可以看到屏幕的左下角出现了一个冒号，输入”w”，并按下回车，文件便得到了保存；再进入底线命令模式，输入”q”便可以关闭文件，回到命令行界面（保存和退出的命令，可以如上述分两步完成，也可以由”:wq”一条命令完成，如图）。

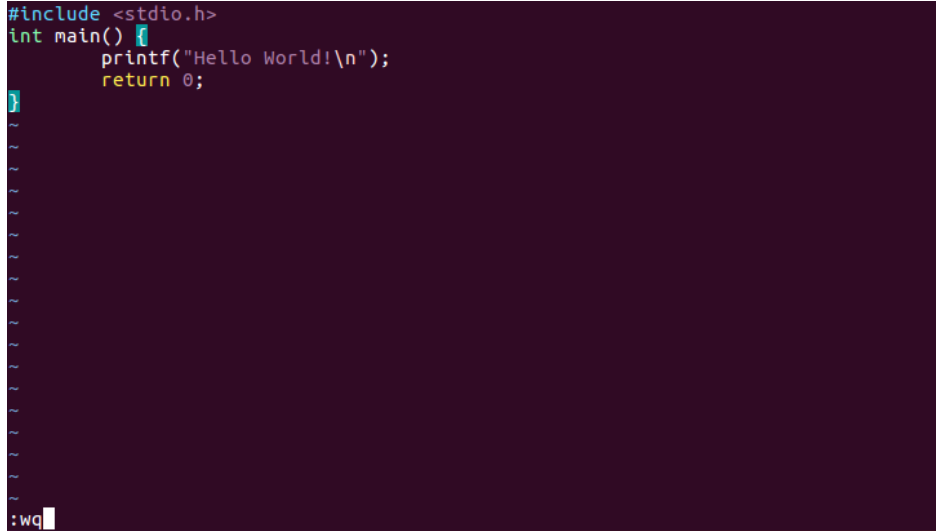


图 0.3: 保存并退出

接着简单了解一下对 Vim 进行一些自定义配置的方法：

```
1 # 首先按如下方法打开（如果原本没有则是新建） ~/.vimrc 文件
2 vim ~/.vimrc
```

在这个文件中写入如下内容：

```
1 set cursorline
```

保存并退出文件后，我们便完成了对 Vim 的配置（上述配置能够让光标所在行下加下划线）。想要了解更多配置参数的同学可以自行在网上搜索。

相关的内容可在网上查询，随用随查即可。

0.4.2 GCC

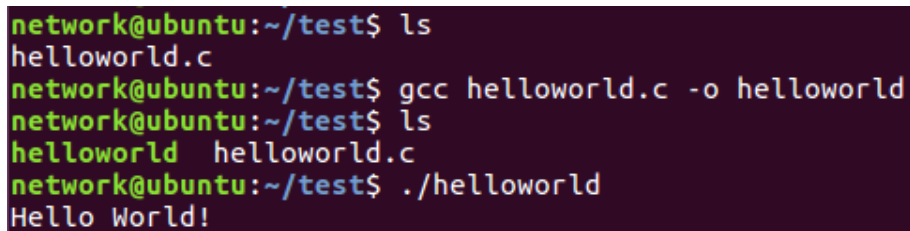
GCC (GNU Compiler Collection, GNU 编译器套件) 包含了著名的 C 语言编译器 gcc (GNU project C and C++ compiler)。我们的实验课程使用 gcc 作为 C 语言编译器。其常用的使用方法如下图所示，可以自己动手实践，写一些简单的 C 代码来编译运行。如果想要同时编译多个文件，可以直接用 -o 选项将多个文件进行编译连接：gcc testfun.c test.c -o test，也可以先使用 -c 选项将每个文件单独编译成 .o 文件，再用 -o 选项将多个 .o 文件进行链接：gcc -c testfun.c && gcc -c test.c && gcc testfun.o test.o -o test，两者等价。


```
1  语法: gcc [选项]... [参数]...
2  选项 (常用):
3  -o          指定生成的输出文件
4  -S          将 C 代码转换为汇编代码
5  -Wall       显示一些警告信息
6  -c          仅执行编译操作, 不进行链接操作
7  -M          列出依赖
8  -I<path>    编译时指定头文件目录, 使用标准库时不需要指定目录, -I 参数可以用相对
    ↪ 路径, 比如头文件在当前目录, 可以用 -I. 来指定
9  参数:
10 C 源文件: 指定 C 语言源代码文件
11 e.g.
12
13 $ gcc test.c -o test
14 # 使用 -o 选项生成名为 test 的可执行文件
```

下面, 我们通过编译之前完成的 `helloworld.c` 来熟悉 GCC 的最基本使用方法:

(1) 在命令行中, 使用 `gcc helloworld.c -o helloworld` 命令, 便可以创建由 `helloworld.c` 文件编译成的 `helloworld` 的可执行文件 (使用 `ls` 可以看到目录内出现了 `helloworld` 可执行文件)。

(2) 在命令行中, 输入 `./helloworld` 运行可执行文件, 观察现象。



```
network@ubuntu:~/test$ ls
helloworld.c
network@ubuntu:~/test$ gcc helloworld.c -o helloworld
network@ubuntu:~/test$ ls
helloworld helloworld.c
network@ubuntu:~/test$ ./helloworld
Hello World!
```

图 0.4: GCC 编译可执行文件并运行

0.4.3 Makefile

如果想了解操作系统这样的大型软件, 首先面临一个很大的问题: 这些代码应当从那里开始阅读? 答案是 Makefile。当你不知所措的时候, 从 Makefile 开始往往会是一个不错的选择。那么 `make` 是什么, 而 Makefile 又是什么呢? `make` 工具一般用于维护软件开发的工程项目, 它可以根据时间戳自动判断项目的哪些部分需要重新编译, 每次只重编译必要的部分。`make` 工具会读取 Makefile 文件, 并根据 Makefile 的内容来执行相应的编译操作。

相较于手动编译而言, Makefile 具有更高的便捷性, 可以方便地管理大型项目。而且理论上 Makefile 支持任意语言, 只要其编译器可以通过 `shell` 命令来调用即可。如果一个项目有着复杂的构建流程, Makefile 便能充分展现其优势。为了更为清晰地介绍 Makefile 的基本概念, 下面通过编制一个简单的 Makefile 来说明。假设我们手头有一个 Hello World 程序需要编译。如果我们没有 Makefile, 则需动手编译这个程序, 执行以下命令:


```
1 # 直接使用 gcc 编译 Hello World 程序
2 $ gcc -o hello_world hello_world.c
```

那么, 如果想把它写成 Makefile, 该如何操作呢? Makefile 最基本的格式如下:

```
1 target: dependencies
2     command 1
3     command 2
4     ...
5     command n
```

其中, **target** 是构建 (build) 的目标, 可以是目标文件、可执行文件, 也可以是一个标签。而 **dependencies** 是构建该目标所需的其他文件或其他目标。之后是构建该目标所需执行的命令。有一点尤为需要注意, 每一个命令 (command) 之前必须用一个制表符 (Tab) 缩进。这里必须使用制表符而不能是空格, 否则 **make** 会报错。

通过在 Makefile 中书写显式规则来告诉 **make** 工具文件间的依赖关系: 如果想要构建 **target**, 那么首先要准备好 **dependencies**, 接着执行 **command** 中的命令, 最终完成构建 **target**。在编写完恰当的规则之后即在 shell 中输入 **make target** (**target** 是目标名), 即可执行相应的命令、生成相应的目标。

前面提到, **make** 工具根据时间戳来判断是否需要编译, **make** 只有依赖文件中存在文件的修改时间比目标文件的修改时间晚时 (也就是对依赖文件做了改动), shell 命令才会被执行, 编译生成新的目标文件。

简易 Makefile 内容如下, 之后执行 **make all** 或是 **make** 命令, 即可产生 **hello_world** 可执行文件。

```
1 all: hello_world.c
2     gcc -o hello_world hello_world.c
```

下面, 我们尝试编写一个简单的 Makefile 文件:

(1) 在命令行中, 创建名为 "Makefile" 的文件, 使用 Vim 打开它, 并写入如下内容:

```
1 all: hello_world.c
2     gcc -o hello_world hello_world.c
3 clean:
4     rm -f helloworld
```

其中, 前两行定义了编译 **helloworld.c** 的命令 ("all" 为 **target**, "helloworld.c" 为 **dependencies**, 第二行为编译 **helloworld** 程序的命令); "clean" 后的部分为删除可执行文件的命令。

(2) 保存并回到命令行界面后, 输入 **make clean**, 执行 Makefile 文件中的 **rm helloworld** 命令, 删除了之前编译出的可执行文件; 接着, 输入 **make all**, 执行 Makefile 文件中的 **gcc helloworld.c -o helloworld** 命令, 编译出可执行文件。过程如下图:

```
network@ubuntu:~/test$ make clean
rm helloworld
network@ubuntu:~/test$ ls
helloworld.c Makefile
network@ubuntu:~/test$ make all
gcc helloworld.c -o helloworld
network@ubuntu:~/test$ ls
helloworld helloworld.c Makefile
network@ubuntu:~/test$ ./helloworld
Hello World!
```

图 0.5: make 命令执行过程和效果

在 lab0 阶段，建议自己尝试编写一个简单的 Makefile 文件，体会 make 工具的使用方法。

0.4.4 ctags

ctags 是一个方便 Vim 下代码阅读的工具。这里只进行一些最基础功能的介绍：(1) 为了能够跨目录使用 ctags，我们需要添加 Vim 的相关配置，按照上文所述的方法打开 `.vimrc` 文件，添加以下内容并保存：

```
1 set tags=tags;
2 set autochdir
```

(2) 为了进行测试，我们修改 `helloworld.c` 文件（在其中添加一个函数），并新建一个文件 `ctags_test.c`，在其中调用 `helloworld.c` 中新建的函数，如下图：

[illegible]

图 0.6: helloworld.c 文件中的内容

[illegible]

图 0.7: ctags_test.c 文件中的内容

(3) 我们回到命令行界面, 执行命令 `ctags -R *`, 会发现在该目录下出现了新的文件 `tags`, 接下来就可以使用一些 `ctags` 的功能了:

使用 Vim 打开 `ctags_test.c`，将光标移到调用的函数 (`test_ctags`) 上，按下 `Ctrl+]`，便可以跳转到 `helloworld.c` 中的函数定义处；再按下 `Ctrl+T` 或 `Ctrl+O`（有些浏览器 `Ctrl+T` 是新建页面，会出现热键冲突），便可以回到跳转前的位置。

```
#include "helloworld.c"

int main() {
    test_ctags();
    return 0;
}
```

图 0.8: 光标处于函数名上

使用 Vim 打开 `ctags_test.c`, 按“:”进入底线命令模式, 再输入“`tag test_ctags`”, 也可以跳转到该函数定义的位置。

正式开始操作系统实验后，需要阅读和理解的代码量会增加很多，不同文件之间的函数调用会给阅读代码带来很大的阻力。熟练运用 `ctags` 的相关功能，可以为读者阅读、理解代码提供很大的帮助。

0.5 Git 专栏-轻松维护和提交代码

本书设计的实验通过 Git 版本控制系统进行管理，在这里就来了解一下 Git 相关内容。

0.5.1 Git 是什么?

最初的版本控制是纯手工完成的：修改文件，保存文件副本。如果保存副本时命名比较随意，时间长了就不知道哪个是新的，哪个是旧的了，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版做了哪些修改，当几个版本过去后，很可能就像图 0.9 一样糟糕了。

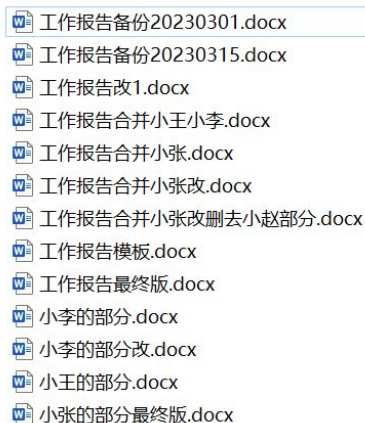


图 0.9: 手工版本控制

在很多情况下，工程项目也往往由多人一起完成的，在项目刚刚开始时，分工，制定计划，埋头苦干，但版本管理可能会让人头疼不已。其本质原因在于每个人都会对目的内容进行改动，结果就可能是 A 把添加完功能的项目打包发给了 B，然后自己继续添加功能。一天后，B 把他修改后的项目包又发给了 A，这时 A 就必须非常清楚发给 B 之后到他发回来的这段时间，自己究竟对哪些地方做了改动，然后还要进行合并，相当困难。

这时会面临一个无法避免的事实：如果每一次小小的改动，开发者之间都要相互通知，那么一些错误的改动将会令我们付出很大的代价：一个错误的改动通知几方同时纠正。如果一次性做了大幅度的修改，那么只有在概览项目的很多文件后才能知道改动在哪里，也才能做合并修改。

由此产生了需求，大家希望：

- 自动帮助记录每次文件的改动，而且最好是有撤回功能，改错了可以轻松撤销。
- 支持多人协作编辑，命令与操作简洁。
- 能够在不同的历史版本中切换。
- 可方便地在软件中查看某次改动。

版本控制系统就是能够解决上述需求的一种系统，而 Git 则是一种先进的分布式版本控制系统。

Git 是由 Linux 的创始人林纳斯·托瓦尔兹 (Linus Torvalds) 创造，最初用于管理自己的 Linux 开发过程。他对于 Git 的解释是：The stupid content tracker(傻瓜内容追踪器)。

Note 0.5.1 版本控制是一种记录若干文件内容变化,以便将来查阅特定版本修订情况的系统。

0.5.2 Git 基础指引

通过前几节的学习,完成如下操作:

1. 回到主目录,创建一个名为 `learnGit` 的目录
2. 进入 `learnGit` 目录
3. 输入 `git init`
4. 用 `ls` 命令添加适当参数看看多了什么

可以发现,新建的目录下面多了一个名为 `.git` 的隐藏目录,这个目录就是 Git 版本库,常被称为仓库 (repository)。需要注意的是,实验中不会对 `.git` 目录进行任何直接操作,不要轻易对该目录做任何操作。

`git init` 执行后就拥有了一个仓库。建立的 `learnGit` 目录就是 Git 里的工作区。目前除了 `.git` 版本库目录以外空无一物。

Note 0.5.2 在我们的 MOS 操作系统实验中我们不需要使用到 `git init` 命令,每个人一开始就都有一个名为 `21xxxxxx`(你的学号) 的版本库。

目前,在工作区 `learnGit` 中仅有 Git 版本库,下面新建一个文件 `readme.txt`,内容为“BUAA_OSLAB”。执行以下命令得到该文件添加至版本库:

```
1 $ git add readme.txt
```

注意,执行此命令后,并未真正地把 `readme.txt` 提交到版本库,Git 同其他大多数版本控制系统一样,需要 `add` 之后再执行一次提交操作,提交操作的命令如下:

```
1 $ git commit
```

如果不带任何附加选项,执行后会弹出一个说明窗口,如下所示,其中 **Notes to test** 就是本次提交所附加的说明。

```
1 # 请为您的变更输入提交说明。以 '#' 开始的行将被忽略,而一个空的提交
2 # 说明将会终止提交。
3 #
4 # 位于分支 master
5 # 您的分支与上游分支 'origin/master' 一致。
6 #
7 # 要提交的变更:
8 #   修改:      readme.txt
9 #
```

注意,弹出的窗口中我们**必须**得添加本次 `commit` 的说明,这意味着我们不能提交空白说明,否则我们的提交不会成功。而且在添加评论之后,报错退出就可以成功提交。

Note 0.5.3 初学者一般不重视 `git commit` 内容的有效性，总是使用说明意义不明的字符串作为说明提交。但以后可能就会发现写一个自己看得懂，别人也能看得懂的提交说明是多么必要。所以为了提高可读性，尽量每次提交都能见名知义，比如“fixed a bug in ...”这样的描述，推荐一条命令：`git commit --amend` 重新书写最后一次提交的说明。

以窗口提交方式是一种更简洁的方式，使用以下命令：

```
1 $ git commit -m [comments]
```

[comments] 格式为“评论内容”，上述的提交过程我们可以简化为下面一条命令

```
1 $ git commit -m "Notes to test."
```

如果提交之后看到类似的提示，就说明提交成功了。

```
1 [master 955db52] Notes to test.  
2 1 file changed, 1 insertion(+), 1 deletion(-)
```

本次提交中可以得到以下提示信息中，后续会详细说明提交提示信息的含义。

- 本次提交的分支是 master
- 本次提交的 ID 是 955db52
- 提交说明是 Notes to test
- 共有 1 个文件相比之前发生了变化：1 行的添加与 1 行的删除行为

在实验过程中，提交后可能会出现如下提示信息，说明这是要求设置提交者身份。

```
1 *** Please tell me who you are.  
2  
3 Run  
4  
5 git config --global user.email "you@example.com"  
6 git config --global user.name "Your Name"  
7  
8 # to set your account's default identity.  
9 # Omit --global to set the identity only in this repository.
```

Note 0.5.4 可以用以下两条命令设置用户名和邮箱：

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"  
示例：  
git config --global user.email "qianlxc@126.com"  
git config --global user.name "Qian"
```

现在已设置了提交者的信息，提交者信息是为了告知所有负责该项目的人每次提交是由谁提交的，并提供联系方式以进行交流。

0.5.3 Git 文件状态

首先对于任何一个文件, 在 Git 中都只有四种状态: 未跟踪 (untracked)、未修改 (unmodified)、已修改 (modified)、已暂存 (staged):

未跟踪 表示没有跟踪 (add) 某个文件的变化, 使用 `git add` 即可跟踪文件。

未修改 表示某文件在跟踪后一直没有改动过或者改动已经被提交。

已修改 表示修改了某个文件, 但还没有加入 (add) 到暂存区中。

已暂存 表示把已修改的文件放在下次提交 (commit) 时要保存的清单中。

这里使用一张图来说明文件的四种状态的转换关系:

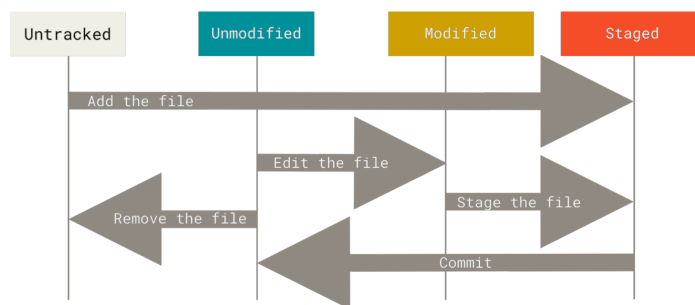


图 0.10: Git 中的四种状态转换关系

完成以下练习以进一步熟悉 Git 的使用方法。

Thinking 0.1 思考下列有关 Git 的问题:

- 在 `/home/21xxxxxx/learnGit` (已初始化) 目录下创建一个名为 `README.txt` 的文件。执行命令 `git status > Untracked.txt`。
- 在 `README.txt` 文件中添加任意文件内容, 然后使用 `add` 命令, 再执行命令 `git status > Stage.txt`。
- 提交 `README.txt`, 并在提交说明里写入自己的学号。
- 执行命令 `cat Untracked.txt` 和 `cat Stage.txt`, 对比两次运行的结果, 体会 `README.txt` 两次所处位置的不同。
- 修改 `README.txt` 文件, 再执行命令 `git status > Modified.txt`。
- 执行命令 `cat Modified.txt`, 观察其结果和第一次执行 `add` 命令之前的 `status` 是否一样, 并思考原因。

Note 0.5.5 `git status` 是一个查看当前文件状态的有效命令，而 `git log` 则是提交日志，每提交一次，Git 会在提交日志中记录一次。`git log` 将在版本切换时发挥很大的作用。

实施前述实验后，`Untracked.txt`，`Stage.txt` 和 `Modified.txt` 的内容如下。

```

1  Untracked.txt 的内容如下
2
3  # On branch master
4  # Untracked files:
5  #   (use "git add <file>..." to include in what will be committed)
6  #
7  #       README.txt
8  nothing added to commit but untracked files present (use "git add" to track)
9
10 Stage.txt 的内容如下
11
12 # On branch master
13 # Changes to be committed:
14 #   (use "git reset HEAD <file>..." to unstage)
15 #
16 #       new file:   README.txt
17 #
18
19 Modified.txt 的内容如下
20
21 # On branch master
22 # Changes not staged for commit:
23 #   (use "git add <file>..." to update what will be committed)
24 #   (use "git checkout -- <file>..." to discard changes in working directory)
25 #
26 #       modified:   README.txt
27 #
28 no changes added to commit (use "git add" and/or "git commit -a")

```

通过仔细观察,我们看到第一个文本文件 `Untracked.txt` 中第2行是:Untracked files,而第二个文本文件 `Stage.txt` 中第二行内容是: Changes to be committed,而第三个文件 `Modified.txt` 中则是 Changes not staged for commit。

这三种不同的提示分别意味着:在 `README.txt` 新建的时候,其处于为未跟踪状态(untracked);在 `README.txt` 中任意添加内容,接着用 `add` 命令之后,文件处于暂存状态(staged);在修改 `README.txt` 之后,其处于被修改状态(modified)。

Note 0.5.6 关于 思考-Git 的使用 1,实际上是因为 `git add` 命令本身是有多义性的,虽然差别较小但是不同情境下使用依然是有区别。因此需注意:新建文件后要 `git add`,修改文件后也需要 `git add`。

Thinking 0.2 仔细看看0.10,思考一下箭头中的 add the file、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢? ■

此时相信读者对 Git 的设计有了初步的认识。下一步就来深入理解一下 Git 里的一些机制。

0.5.4 Git 三棵树

本地仓库由 Git 维护的三棵“树”组成。第一个是工作区，它持有实际文件；第二个是暂存区（Index 有时也称 Stage），它像个暂时存放的区域，临时保存你的改动；最后是 HEAD，指向你最近一次提交后的结果。

Git 的对象库位于 `.git/objects` 中，所有需要实施版本控制的文件会被压缩成二进制文件，压缩后的二进制文件成为一个 Git 对象，保存在 `.git/objects` 目录。Git 计算当前文件内容的哈希值（长度为 40 的字符串），并作为该对象的文件名。

在 `.git` 目录中，文件 `.git/index` 实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等），但是文件的内容并不存储在其中，而是保存在 Git 对象库（`.git/objects`）中，文件索引建立了文件和对象库中对象实体之间的对应。下图展示了工作区、版本库中的暂存区和版本库之间的关系，揭示了不同操作所带来的不同影响。

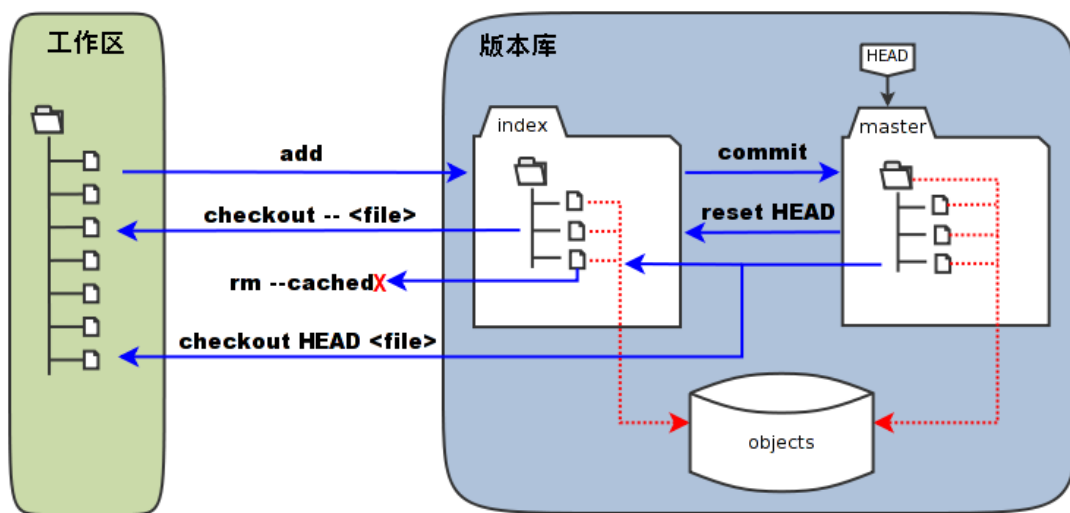


图 0.11: 工作区、暂存区和版本库

- 图中 `objects` 标识的区域为 Git 的对象库，实际位于 `.git/objects` 目录下。
- 图中左侧为工作区，右侧为版本库。在版本库中标记为“index”的区域是暂存区 (stage, index)，标记为 `master` 的是 `master` 分支所代表的目录树。
- 图中我们可以看出此时 `HEAD` 实际是指向 `master` 分支的一个“指针”。所以图示的命令中出现 `HEAD` 的地方可以用 `master` 来替换。
- 当对工作区修改（或新增）的文件执行 `git add` 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的 ID 被记录在暂存区的文件索引中。
- 当执行提交操作 (`git commit`) 时，会将暂存区的目录树写到版本库（对象库）中，`master` 分支会做相应的更新。即 `master` 指向的目录树就是提交时暂存区的目录树。

- 当执行 `git rm --cached <file>` 命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行 `git reset HEAD` 命令时，暂存区的目录树会被重写，由 `master` 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 `git checkout -- <file>` 命令时，会用暂存区指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行 `git checkout HEAD <file>` 命令时，会用 `HEAD` 指向的 `master` 分支中的指定文件来替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

在 Git 中引入**暂存区**的概念是 Git 里较难理解却是最有亮点的设计之一，在这里不再详细介绍其能快速快照与回滚原理。有兴趣的同学不妨去看看[Pro Git](#)这本书。

0.5.5 Git 版本回退

在编写代码时，可能遇到过因错误地删除文件或因一个修改导致程序再也无法运行等情况。Git 允许进行版本回退。首先，有必要学习一些撤销命令：

`git rm --cached <file>` 这条命令是指从暂存区中删除不再想跟踪的文件，比如调试用的文件等。

`git checkout -- <file>` 如果在工作区中对多个文件经过多次修改后，发现编译无法通过了。如果尚未执行 `git add`，则可使用本命令将工作区恢复成原来的样子。

`git reset HEAD <file>` 上一条命令是在未执行 `git add` 命令便修改文件生效并放入暂存区，可使用 `git checkout` 命令。那么如果不慎已经执行了 `git add`，则可使用本命令。再对需要恢复的文件使用上一条命令即可。

`git clean <file> -f` 如果你的工作区混入了未知内容，你没有追踪它，但是想清除它的话就可以使用本命令，它可以帮你把未知内容剔除出去。

Thinking 0.3 思考下列问题：

1. 代码文件 `print.c` 被错误删除时，应当使用什么命令将其恢复？
2. 代码文件 `print.c` 被错误删除后，执行了 `git rm print.c` 命令，此时应当使用什么命令将其恢复？
3. 无关文件 `hello.txt` 已经被添加到暂存区时，如何在不删除此文件的前提下将其移出暂存区？

关于上面那些撤销命令，可在你不慎删除不应删除内容是再行查阅，当然更推荐使用 `git status` 来看当前状态下 Git 的推荐命令。现阶段主要掌握 `git add` 和 `git commit` 的用法。当然，一定要慎用撤销命令。否则撤销之后如何撤除撤销命令将是一件难事。

介绍完上面三条撤销命令，下面介绍 Git 版本回退命令。

```
1 git reset --hard
```

为了体会 `reset` 命令的作用，下面先做一个小练习：

Thinking 0.4 思考下列有关 Git 的问题：

- 找到在 `/home/21xxxxxx/learnGit` 下刚刚创建的 `README.txt` 文件，若不存在则新建该文件。
- 在文件里加入 `Testing 1`, `git add`, `git commit`, 提交说明记为 1。
- 模仿上述做法，把 1 分别改为 2 和 3，再提交两次。
- 使用 `git log` 命令查看提交日志，看是否已经有三次提交，记下提交说明为 3 的哈希值^a。
- 进行版本回退。执行命令 `git reset --hard HEAD^` 后，再执行 `git log`，观察其变化。
- 找到提交说明为 1 的哈希值，执行命令 `git reset --hard <hash>` 后，再执行 `git log`，观察其变化。
- 现在已经回到了旧版本，为了再次回到新版本，执行 `git reset --hard <hash>`，再执行 `git log`，观察其变化。

^a使用 `git log` 命令时，在 `commit` 标识符后的一长串数字和字母组成的字符串

使用这条命令可以进行版本回退或者切换到任何一个版本。它有两种用法：第一种是使用 `HEAD` 类似形式，如果想退回上个版本就用 `HEAD^`，上上个版本的话就用 `HEAD^^`，要是回退到前 50 个版本则可使用 `HEAD~50` 来代替；第二种就是使用 `hash` 值，使用 `hash` 值可以在不同版本之间任意切换，足见 `hash` 值的强大。

必须注意，`--hard` 是 `git reset` 命令唯一的危险用法，它也是 Git 会真正地销毁数据的几个操作之一。其他任何形式的 `git reset` 调用都可以轻松撤销，但是 `--hard` 选项不能，因为它强制覆盖了工作目录中的文件。若该文件还未提交，Git 会覆盖它从而导致无法恢复。

0.5.6 Git 分支

如果你还有印象的话，我们之前提到过分支这个概念，那么分支是个什么东西呢？分支就是科幻电影里面的平行宇宙，不同的分支间不会互相影响。或许当你正在电脑前

努力学习操作系统的时候，另一个你正在另一个平行宇宙里努力学习面向对象。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线工作的同时继续工作。在我们实验中也会多次使用到分支的概念。首先我们来讲一条创建分支的命令。

```
1 # 创建一个基于当前分支产生的分支，其名字为 <branch-name>
2 $ git branch <branch-name>
```

这条命令将会在实验课考试进行的时候用到。其功能相当于把当前分支的内容拷贝一份到新的分支里去，然后我们在新的分支上做测试功能的添加即可，不会影响实验分支的效果等。假如我们当前在 master¹分支下已经有过三次提交记录，这时我们使用 `git branch` 命令新建了一个分支为 `testing`（参考图 0.12）。

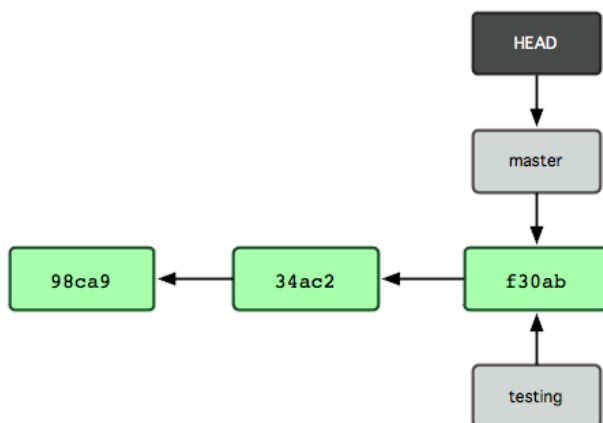


图 0.12: 分支建立后

删除一个分支也很简单，只要加上 `-d` 选项（`-D` 是强制删除）即可，就像这样

```
1 # 强制删除一个指定的分支
2 $ git branch -D <branch-name>
```

想查看所有的远程分支和本地分支，只需要加上 `-a` 选项即可

```
1 # 查看所有的远程与本地的所有分支
2 $ git branch -a
3
4 # 使用该命令的效果如下
5 # 前面带 * 的分支是当前分支
6 lab1
7 lab1-exam
8 * lab1-result
9 master
10 remotes/origin/HEAD -> origin/master
11 remotes/origin/lab1
12 remotes/origin/lab1-exam
13 remotes/origin/lab1-result
14 remotes/origin/master
15 # 带 remotes 是远程分支，在后面提到远程仓库的时候我们会知道
```

¹master 分支是我们的主分支，一个仓库初始化时自动建立的默认分支

我们建立了分支并不代表会自动切换到分支，那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HEAD 的特别指针。在 Git 中，它是一个指向你正在工作中的本地分支的指针，可以将 HEAD 想象为当前分支的别名。运行 `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 master 分支里工作。

那么我们如何切换到另一个分支去呢，这时候我们就要用到这个我们在实验中更常见的分支命令了

```
1 # 切换到 <branch-name> 代表的分支，这时候 HEAD 游标指向新的分支
2 $ git checkout <branch-name>
```

比如这时候我们使用 `git checkout testing`，这样 HEAD 就指向了 testing 分支（见图0.13）。

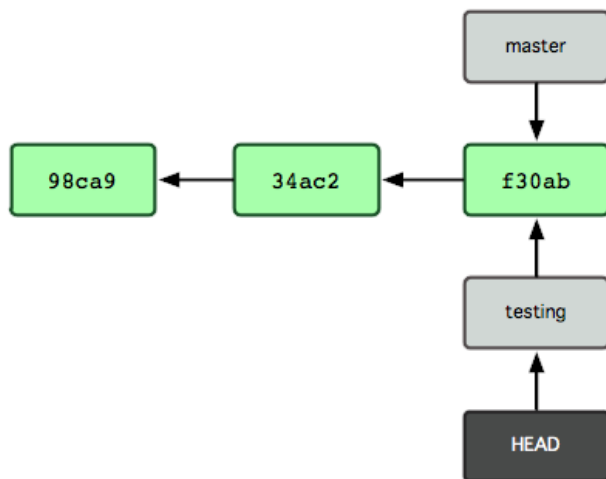


图 0.13: 分支切换后

这时候你会发现你的工作区就是 testing 分支下的工作目录，而且在 testing 分支下的修改，添加与提交不会对 master 分支产生任何影响。

我们之前所介绍的这些命令只是在本地进行操作的，其中必须掌握

1. `git add`
2. `git commit`
3. `git branch`
4. `git checkout`

其余命令可以临时查阅，当然掌握对你益处现在体会不出来，但当你们小团队哪天一起做项目的时候，你就会体会到掌握这么多 Git 的知识是件多么幸福的事情了。之前我们所有的操作都是在本地版本库上操作的，下面我们要介绍的是一组和远程仓库有关的命令。这组命令是最容易出错的，所以你一定要认真学习。

0.5.7 Git 远程仓库与本地

下面介绍两条跟远程仓库有关的命令，其作用很简单，但要用好却是比较难。

```
1 # git push 用于从本地版本库推送到服务器远程仓库
2 $ git push
3
4 # git pull 用于从服务器远程仓库抓取到本地版本库
5 $ git pull
```

`git push` 只是将本地版本库里已经 `commit` 的部分同步到服务器上去，不包括暂存区里存放的内容。在我们实验中除了还可能会加些选项使用

```
1 # origin 在我们实验里是固定的，以后就明白了。branch 是指本地分支的名称。
2 $ git push origin [branch]
```

这条命令可以将我们本地创建的分支推送到远程仓库中，在远程仓库建立一个同名的本地追踪的远程分支。

`git pull` 是条更新用的命令，如果助教老师在服务器端发布了新的分支，下发了新的代码或者进行了一些改动的话，我们就需要使用 `git pull` 来让本地版本库与远程仓库保持同步。

如果你还想进一步学习 Git 的知识，可以查看教程²，和游玩[GitHug](#)

0.6 进阶操作

0.6.1 Linux 操作补充

首先，是两种常用的查找命令：`find` 和 `grep`

使用 `find` 命令并加上 `-name` 选项可以在当前目录下递归地查找符合参数所示文件名的文件，并将文件的路径输出至屏幕上。

```
1 find - search for files in a directory hierarchy
2 用法:find -name 文件名
```

`grep` 是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。简单来说，`grep` 命令可以从文件中查找包含 `pattern` 部分字符串的行，并将该文件的路径和该行输出至屏幕。当你需要在整个项目目录中查找某个函数名、变量名等特定文本的时候，`grep` 将是你手头一个强有力的工具。

```
1 grep - print lines matching a pattern
2 用法:grep [选项]... PATTERN [FILE]...
3 选项 (常用):
4 -a          不忽略二进制数据进行搜索
5 -i          忽略文件大小写差异
6 -r          从目录递归查找
7 -n          显示行号
```

`tree` 命令可以根据文件目录生成文件树，作用类似于 `ls`。

²推荐廖雪峰老师的网站: <http://www.liaoxuefeng.com/>

```

1 tree
2 用法: tree [选项] [目录名]
3 选项(常用):
4 -a 列出全部文件
5 -d 只列出目录

```

Linux 的文件调用权限分为三级: 文件拥有者、群组、其他。利用 `chmod` 可以藉以控制文件如何被他人所调用。

```

1 chmod
2 用法: chmod 权限设定字符串 文件...
3 权限设定字符串格式:
4 [ugoa...][[+=][rwxX]...][,...]

```

其中: `u` 表示该文件的拥有者, `g` 表示与该文件的拥有者属于同一个群组, `o` 表示其他以外的人, `a` 表示这三者皆是。+ 表示增加权限、- 表示取消权限、= 表示唯一设定权限。`r` 表示可读取, `w` 表示可写入, `x` 表示可执行, `X` 表示只有当该文件是个子目录或者该文件已经被设定过为可执行。

此外 `chmod` 也可以用数字来表示权限, 格式为:

```

1 chmod abc 文件

```

`abc` 为三个数字, 分别表示拥有者, 群组, 其他人的权限。`r=4`, `w=2`, `x=1`, 用这些数字的加和来表示权限。

`diff` 命令用于比较文件的差异。

```

1 diff [选项] 文件 1 文件 2
2 常用选项
3 -b 不检查空白字符的不同
4 -B 不检查空行
5 -q 仅显示有无差异, 不显示详细信息

```

`sed` 是一个文件处理工具, 可以将数据行进行替换、删除、新增、选取等特定工作。

```

1 sed
2 sed [选项] '命令' 输入文本
3 选项(常用):
4 -n: 安静模式, 只显示经过 sed 处理的内容。否则显示输入文本的所有内容。
5 -i: 直接修改读取的档案内容, 而不是输出到屏幕。否则, 只输出不编辑。
6 命令(常用):
7 [行号]a[内容]: 新增, 在行号后新增一行相应内容。行号可以是“数字”, 在这一行之后新增,
8 也可以是“起始行, 终止行”, 在其中的每一行后新增。
9 当不写行号时, 在每一行之后新增。使用 $ 表示最后一行。后面的命令同理。
10 [行号]c[内容]: 取代。用内容取代相应行的文本。
11 [行号]i[内容]: 插入。在当前行的上面插入一行文本。
12 [行号]d: 删除当前行的内容。
13 [行号]p: 输出选择的内容。通常与选项 -n 一起使用。
14 s/re (正则表达式) /string: 将 re 匹配的内容替换为 string。

```

`sed` 中正则表达式的相关语法可以查阅[sed 文档](#)。`sed` 等工具中的正则表达式语法和 Java 等语言中不完全相同, 请注意区分。

下面是一些 `sed` 的使用实例:


```

1 sed -n '3p' my.txt
2 # 输出 my.txt 的第三行
3 sed '2d' my.txt
4 # 删除 my.txt 文件的第二行
5 sed '2,$d' my.txt
6 # 删除 my.txt 文件的第二行到最后一行
7 sed 's/str1/str2/g' my.txt
8 # 在整行范围内把 str1 替换为 str2
9 # 如果没有 g 标记, 则只有每行第一个匹配的 str1 被替换成 str2
10 sed -e '4a\str ' -e 's/str/aaa/' my.txt
11 # -e 选项允许在同一行里执行多条命令。例子的第一条是第四行后添加一个 str,
12 # 第二个命令是将 str 替换为 aaa。命令的执行顺序对结果有影响。

```

awk 是一种处理文本文件的语言, 是一个强大的文本分析工具。这里只举几个简单的例子, 学有余力的同学可以自行深入学习。

```

1 awk '$1>2 {print $1,$3}' my.txt

```

这个命令的格式为 **awk 'pattern action' file**, **pattern** 为条件, **action** 为命令, **file** 为文件。命令中出项的 **\$n** 代表每一行中用空格分隔后的第 **n** 项。所以该命令的意义是文件 **my.txt** 中所有第一项大于 2 的行, 输出第一项和第三项。

```

1 awk -F, '{print $2}' my.txt

```

-F 选项用来指定用于分隔的字符, 默认是空格。所以该命令的 **\$n** 就是用, 分隔的第 **n** 项了。

tmux 是一个优秀的终端复用软件, 可用于在一个终端窗口中运行多个终端会话。窗格, 窗口, 会话是 **tmux** 的三个基本概念, 一个会话可以包含多个窗口, 一个窗口可以分割为多个窗格。突然中断退出后 **tmux** 仍会保持会话, 通过进入会话可以直接从之前的环境开始工作。

窗格操作

tmux 的窗格 (pane) 可以做出分屏的效果。

- **Ctrl+B %** 垂直分屏 (组合键之后按一个百分号), 用一条垂线把当前窗口分成左右两屏。
- **Ctrl+B "** 水平分屏 (组合键之后按一个双引号), 用一条水平线把当前窗口分成上下两屏。
- **Ctrl+B O** 依次切换当前窗口下的各个窗格。
- **Ctrl+B Up|Down|Left|Right** 根据按箭方向选择切换到某个窗格。
- **Ctrl+B Space** (空格键) 对当前窗口下的所有窗格重新排列布局, 每按一次, 换一种样式。
- **Ctrl+B Z** 最大化当前窗格。再按一次后恢复。
- **Ctrl+B X** 关闭当前使用中的窗格, 操作之后会给出是否关闭的提示, 按 **y** 确认即关闭。

窗口操作

每个窗口 (window) 可以分割成多个窗格 (pane)。

- `Ctrl+B C` 创建之后会多出一个窗口
- `Ctrl+B P` 切换到上一个窗口。
- `Ctrl+B N` 切换到下一个窗口。
- `Ctrl+B 0` 切换到 0 号窗口，依此类推，可换成任意窗口序号
- `Ctrl+B W` 列出当前 session 所有串口，通过上、下键切换窗口
- `Ctrl+B &` 关闭当前 window，会给出提示是否关闭当前窗口，按下 `y` 确认即可。

会话操作

一个会话 (session) 可以包含多个窗口 (window)

- `tmux new -s 会话名` 新建会话
- `Ctrl+B D` 退出会话，回到 shell 的终端环境
- `tmux ls` 终端环境查看会话列表
- `tmux a -t 会话名` 从终端环境进入会话
- `tmux kill-session -t 会话名` 销毁会话

0.6.2 shell 脚本

在以后的工作中，可能会遇到重复多次用到单条或多条长而复杂命令的情况，初学者可能会想把这些命令保存在一个文件中，以后再打开文件复制粘贴运行，其实大可不必复制粘贴，将文件按照批处理脚本运行即可。简单来说，批处理脚本就是存储了一条或多条命令的文本文件。当有很多想要执行的 Linux 命令来完成复杂的工作，或者有一个或一组命令会经常执行时，我们可以通过 shell 脚本来完成。本节我们将学习使用 bash shell。首先执行 `vim my.sh` 创建并打开一个文件 `my.sh`，使用 Vim 将其打开，并向其中写入以下内容（别忘了在 Vim 里要输入要先按 `i`）：

```
1  #!/bin/bash
2  #balabala
3  echo "Hello World!"
```

我们可以使用 `bash` 来运行这个脚本：

```
1  bash my.sh
```

另外，我们也可以通过命令 `chmod +x my.sh` 来为脚本添加执行权限，然后使用

```
1 ./my.sh
```

来运行。在修改权限之前，我们自己创建的 shell 脚本一般是不能直接运行的，需要用 `chmod +x my.sh` 添加运行权限：在脚本中我们通常会加入 `#!/bin/bash` 到文件首行，以保证直接执行我们的脚本时使用 `bash` 作为解释器。第二行的内容是注释，以 `#` 开头。第三行是命令。

参数与函数

我们可以向 shell 脚本传递参数。`my2.sh` 的内容

```
1 echo $1
```

执行命令

```
1 ./my2.sh msg
```

则 shell 会执行 `echo msg` 这条命令。`$n` 就代表第几个参数，而 `$0` 也就是命令，在例子中就是 `./my2.sh`。除此之外还有一些可能有用的符号组合

- `$#` 传递的参数个数
- `$*` 一个字符串显示传递的全部参数

shell 中的函数也用类似的方式传递参数。

```
1 function <函数名> () {  
2     <commands>  
3 }
```

`function` 或者 `()` 可以省略其中一个。举例

```
1 fun() {  
2     echo "$1"  
3     echo "$2"  
4     echo "the number of parameters is $#"  
5 }  
6 fun 1 str2
```

流程控制语句

shell 脚本中也可以使用分支和循环语句：

`if` 的格式：

```
1 if <condition>  
2 then  
3     <command1>  
4     <command2>  
5     # ...  
6 fi
```

或者写到一行

```
1 if condition; then command1; command2; ... fi
```

举例

```
1 a=1
2 if [ $a -ne 1 ]; then echo ok; fi
```

条件部分可能会让同学们感到疑惑, 实际上 `<condition>` 的位置上也是命令, 一条命令的返回值为 0 时表示其成功执行, 作为条件时则视为成立。可以使用特殊变量 `$?` 获取前一个命令的返回值, 比如刚执行完 `diff <file1> <file2>` 后, 若两文件内容相同, 则 `$?` 为 0。

左中括号 `[` 是一种常用作条件的命令, 其参数是一个条件表达式和末尾的 `]`, 在上例中为 `$a`、`-ne`、`1` 和 `]`。该命令在关系成立时返回 0, 其完整形式为 `test` 命令, 可以在终端中使用 `man test` 查看其详细用法。此外, `true` 命令能够直接返回 0, `!` 命令能够反转参数中命令的返回值, 也经常在条件中使用, 例如 `! diff <file1> <file2>` 在两文件内容不同时返回 0。

条件表达式中的 `-ne` 是一种关系运算符, 它们和 C 语言的比较运算符对应如下:

```
-eq  == (equal)
-ne  != (not equal)
-gt  > (greater than)
-lt  < (less than)
-ge  >= (greater or equal)
-le  <= (less or equal)
```

`while` 语句格式如下

```
1 while <condition>
2 do
3     <commands>
4 done
```

`while` 语句体中可以使用 `continue` 和 `break` 这两个循环控制语句。

例如创建 9 个目录, 名字是 `file1` 到 `file9`。

```
1 a=1
2 while [ $a -ne 10 ]
3 do
4     mkdir file$a
5     a=$((a+1))
6 done
```

除了以上内容, shell 还有 `for`、`case` 语句, `else` 子句, 以及逻辑运算符等语法, 对这些内容有兴趣的同学可以自行了解。

0.6.3 重定向和管道

这部分我们将学习如何实现 Linux 命令的输入输出怎样定向到文件, 以及如何将多个命令组合实现更强大的功能。Linux 定义了三种流:

- 标准输入: `stdin`, 由 0 表示
- 标准输出: `stdout`, 由 1 表示
- 标准错误: `stderr`, 由 2 表示

重定向和管道可以重定向以上的流。>可以重定向命令的标准输出到文件。例如：`ls / > filename` 可以将根目录下的文件名输出到当前目录下的 `filename` 中。与之类似的，还有重定向追加输出>>，将>>前命令的输出追加输出到>>后指定的文件中；以及重定向输入"<"，将<后指定的文件中的数据输入到<前的命令中去，同学们可以自己动手实践一下。"2>>"可以将标准错误重定向。三种流可以同时重定向，举例：

```
1 | command < input.txt 1>output.txt 2>err.txt
```

管道：

管道符号“|”可以连接命令：

```
1 | command1 | command2 | command3 | ...
```

以上内容是将 `command1` 的 `stdout` 发给 `command2` 的 `stdin`, `command2` 的 `stdout` 发给 `command3` 的 `stdin`，依此类推。举例：

```
1 | cat my.sh | grep "Hello"
```

上述命令的功能为将 `my.sh` 的内容输出给 `grep` 命令，`grep` 在其中查找字符串。

```
1 | cat < my.sh | grep "Hello" > output.txt
```

上述命令重定向和管道混合使用，功能为将 `my.sh` 的内容作为 `cat` 命令参数，`cat` 命令 `stdout` 发给 `grep` 命令的 `stdin`，`grep` 在其中查找字符串，最后将结果输出到 `output.txt`。

Thinking 0.5 执行如下命令，并查看结果

- `echo first`
- `echo second > output.txt`
- `echo third > output.txt`
- `echo forth >> output.txt`

Thinking 0.6 使用你知道的方法（包括重定向）创建下图内容的文件（文件命名为 `test`），将创建该文件的命令序列保存在 `command` 文件中，并将 `test` 文件作为批处理文件运行，将运行结果输出至 `result` 文件中。给出 `command` 文件和 `result` 文件的内容，并对最后的结果进行解释说明（可以从 `test` 文件的内容入手）。具体实现的过程中思考下列问题：`echo echo Shell Start` 与 `echo `echo Shell Start`` 效果是否有区别；`echo echo $c>file1` 与 `echo `echo $c>file1`` 效果是否有区别。

```
echo Shell Start...
echo set a = 1
a=1
echo set b = 2
b=2
echo set c = a+b
c=$((a+b))
echo c = $c
echo save c to ./file1
echo $c>file1
echo save b to ./file2
echo $b>file2
echo save a to ./file3
echo $a>file3
echo save file1 file2 file3 to file4
cat file1>file4
cat file2>>file4
cat file3>>file4
echo save file4 to ./result
cat file4>>result
```

图 0.14: 文件内容

0.7 实战测试

随着 Lab0 学习的结束，下面就要开始通过实战检验水平了，请同学们按照下面的题目要求完成所需操作。

Exercise 0.1 Lab0 第一道练习题包括以下四题，如果你四道题全部完成且正确，即可获得 50 分。

1、在 Lab0 工作区的 `src` 目录中，存在一个名为 `palindrome.c` 的文件，使用刚刚学过的工具打开 `palindrome.c`，使用 C 语言实现判断输入整数 n ($1 \leq n \leq 10000$) 是否为回文数的程序 (输入输出部分已经完成)。通过 `stdin` 每次只输入一个整数 n ，若这个数字为回文数则输出 `Y`，否则输出 `N`。[注意：正读倒读相同的整数叫回文数]

2、在 `src` 目录下，存在一个未补全的 `Makefile` 文件，借助刚刚掌握的 `Makefile` 知识，将其补全，以实现通过 `make` 命令触发 `src` 目录下的 `palindrome.c` 文件的编译链接的功能，生成的可执行文件命名为 `palindrome`。

3、在 `src/sh_test` 目录下，有一个 `file` 文件和 `hello_os.sh` 文件。`hello_os.sh` 是一个未完成的脚本文档，请同学们借助 shell 编程的知识，将其补完，以实现通过命令 `bash hello_os.sh AAA BBB`，在 `hello_os.sh` 所处的目录新建一个名为 `BBB` 的文件，其内容为 `AAA` 文件的第 8、32、128、512、1024 行的内容提取 (`AAA` 文件行数一定超过 1024 行)。[注意：对于命令 `bash hello_os.sh AAA BBB`，`AAA` 及 `BBB` 可为任何合法文件的名称，例如 `bash hello_os.sh file hello_os.c`，若已有 `hello_os.c` 文件，则将其原有内容覆盖]

4、补全后的 `palindrome.c`、`Makefile`、`hello_os.sh` 依次复制到路径 `dst/palindrome.c`，`dst/Makefile`，`dst/sh_test/hello_os.sh` [注意：文件名和路径必须与题目要求相同]

要求按照要求完成后，最终提交的文件树图示如下

```

1 |-- dst
2 |   |-- Makefile
3 |   |-- palindrome.c
4 |   |-- sh_test
5 |   |-- hello_os.sh
6 |-- src
7 |   |-- Makefile
8 |   |-- palindrome.c
9 |   |-- sh_test
10 |       |-- file
11 |       |-- hello_os.sh

```

第一题最终提交的文件树

Exercise 0.2 Lab0 第二道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 Lab0 工作区 ray/sh_test1 目录中，含有 100 个子目录 file1~file100，还存在一个名为 changefile.sh 的文件，将其补完，以实现通过命令 `bash changefile.sh`，可以删除该目录内 file71~file100 共计 30 个子目录，将 file41~file70 共计 30 个子目录重命名为 newfile41~newfile70。[注意：评测时仅检测 changefile.sh 的正确性]

要求按照要求完成后，最终提交的文件树图示如下 (file 下标只显示 1~12, newfile 下标只显示 41~55)

```

1 |-- sh_test1
2 |   |-- file1
3 |   |-- file10
4 |   |-- file11
5 |   |-- file12
6 |   |-- file2
7 |   |-- file3
8 |   |-- file4
9 |   |-- file5
10 |   |-- file6
11 |   |-- file7
12 |   |-- file8
13 |   |-- file9
14 |   |-- newfile41
15 |   |-- newfile42
16 |   |-- newfile43
17 |   |-- newfile44
18 |   |-- newfile45
19 |   |-- newfile46
20 |   |-- newfile47
21 |   |-- newfile48
22 |   |-- newfile49
23 |   |-- newfile50
24 |   |-- newfile51
25 |   |-- newfile52
26 |   |-- newfile53
27 |   |-- newfile54
28 |   |-- newfile55

```

第二题最终提交的文件树

Exercise 0.3 Lab0 第三道练习题包括以下一题，如果你完成且正确，即可获得 12 分。

1、在 Lab0 工作区的 `ray/sh_test2` 目录下，存在一个未补全的 `search.sh` 文件，将其补完，以实现通过命令 `bash search.sh file int result`，可以在当前目录下生成 `result` 文件，内容为 `file` 文件含有 `int` 字符串所在的行数，即若有多行含有 `int` 字符串需要全部输出。[注意：对于命令 `bash search.sh file int result`，`file` 及 `result` 可为任何合法文件名称，`int` 可为任何合法字符串，若已有 `result` 文件，则将其原有内容覆盖，匹配时大小写不忽略]

要求按照要求完成后，`result` 内显示样式如下（一个答案占一行）：

```
1 39
2 123
3 134
4 147
5 344
6 395
7 446
8 471
9 735
10 908
11 1207
12 1422
13 1574
14 1801
15 1822
16 1924
17 1940
18 1984
```

第三题完成后结果

Exercise 0.4 Lab0 第四道练习题包括以下两题，如果你均完成且正确，即可获得 26 分。

1、在 Lab0 工作区的 `csc/code` 目录下，存在 `fibo.c`、`main.c`，其中 `fibo.c` 有点小问题，还有一个未补全的 `modify.sh` 文件，将其补完，以实现通过命令 `bash modify.sh fibo.c char int`，可以将 `fibo.c` 中所有的 `char` 字符串更改为 `int` 字符串。[注意：对于命令 `bash modify.sh fibo.c char int`，`fibo.c` 可为任何合法文件名，`char` 及 `int` 可以是任何字符串，评测时评测 `modify.sh` 的正确性，而不是检查修改后 `fibo.c` 的正确性]

2、Lab0 工作区的 `csc/code/fibo.c` 成功更换字段后 (`bash modify.sh fibo.c char int`)，现已有 `csc/Makefile` 和 `csc/code/Makefile`，补全两个 `Makefile` 文件，要求在 `csc` 目录下通过命令 `make` 可在 `csc/code` 目录中生成 `fibo.o`、`main.o`，在 `csc` 目录中生成可执行文件 `fibo`，再输入命令 `make clean` 后只删除两个 `.o` 文件。[注意：不能修改 `fibo.h` 和 `main.c` 文件中的内容，提交的文件中 `fibo.c` 必须是修改后正确的 `fibo.c`，可执行文件 `fibo` 作用是输入一个整数 `n` (从 `stdin` 输入 `n`)，可以输出斐波那契数列前 `n` 项，每一项之间用空格分开。比如 `n=5`，输出 `1 1 2 3 5`]

要求成功使用脚本文件 `modify.sh` 修改 `fibo.c`，实现使用 `make` 命令可以生成 `.o` 文件和可执行文件，再使用命令 `make clean` 可以将 `.o` 文件删除，但保留 `fibo` 和 `.c` 文件。

最终提交时文件中 `fibo` 和 `.o` 文件可有可无。

```
1 |-- code
2 |   |-- Makefile
3 |   |-- fibo.c
4 |   |-- fibo.o
5 |   |-- main.c
6 |   |-- main.o
7 |   `-- modify.sh
8 |-- fibo
9 |-- include
10 |   `-- fibo.h
11 `-- Makefile
```

第四题 `make` 后文件树

```
1 |-- code
2 |   |-- Makefile
3 |   |-- fibo.c
4 |   |-- main.c
5 |   `-- modify.sh
6 |-- fibo
7 |-- include
8 |   `-- fibo.h
9 `-- Makefile
```

第四题 `make clean` 后文件树

0.8 实验思考

- 思考-箭头与命令
- 思考-Git 的一些场景
- 思考-文件的操作
- 思考-Git 的使用 1
- 思考-Git 的使用 2
- 思考-echo 的使用

附录 A

补充知识

A.1 实验目的

1. 了解系统实验部分补充知识

在本章中，我们将介绍一些操作系统实验的补充知识，有助于我们理解并实现操作系统。

A.1.1 printf 格式具体说明

下面是我们需要完成的 `printf` 函数的具体说明，同学们可以参考 `cppreference` 中有关 C 语言 `printf` 函数的文档¹或者 C++ 文档²，对 `printf` 函数进行更加详细的了解。

函数原型：

```
1 void printf(const char* fmt, ...)
```

参数 `fmt` 是 `printf` 中类似的格式字符串，除了可以包含一般字符，还可以包含格式符 (format specifiers)，但略去并新添加了一些功能，格式符的原型为：

`%[flags][width][.precision][length]specifier`

其中 `specifier` 指定了输出变量的类型，参见下表A.1:

¹<https://zh.cppreference.com/w/c/io/printf/>

²<http://www.cplusplus.com/reference/cstdio/printf/>

表 A.1: Specifiers 说明

Specifier	输出	例子
b	无符号二进制数	110
d D	十进制数	920
o O	无符号八进制数	777
u U	无符号十进制数	920
x	无符号十六进制数, 字母小写	1ab
X	无符号十六进制数, 字母大写	1AB
c	字符	a
s	字符串	sample

除了 specifier 之外, 格式符也可以包含一些其它可选的副格式符 (sub-specifier), 有 flag(表A.2):

表 A.2: flag 说明

flag	描述
-	在给定的宽度 (width) 上左对齐输出, 默认为右对齐
0	当输出宽度和指定宽度不同的时候, 在空白位置填充 0

和 width(表A.3):

表 A.3: width 说明

width	描述
数字	指定了要打印数字的最小宽度, 当这个值大于要输出数字的宽度, 则对多出的部分填充空格, 但当这个值小于要输出数字的宽度的时候则不会对数字进行截断。

以及 precision(表A.4):

表 A.4: precision 说明

.precision	描述
. 数字	指定了精度, 不同标识符下有不同的意义, 但在我们实验的版本中这个值只进行计算而没有具体意义, 所以不赘述。

另外, 还可以使用 length 来修改数据类型的长度, 在 C 中我们可以使用 1、ll、h 等, 但这里我们只使用 1, 参看下表A.5

表 A.5: length 说明

length	Specifier	
	d D	b o O u U x X
l	long int	unsigned long int