环境与工具

Environment and Tools

本章内容

Topic

- Linux操作系统

  Linux Operating System
- 文本编辑器 vi/vim

  Text Editor vi/vim
- GNU 工具链

  GNU Toolchain
- 其他常用工具

  Other Commonly Used Tools
- 代码版本管理

  Code Version Control

什么是操作系统?
What is an Operating System?

- **现代计算机由以下部件组成：**

  A modern computer consists of:

  - **一个或多个处理器**

    One or more processors

  - **主存（内存）**

    Main Memory

  - **磁盘**

    Disks

  - **各种输入输出设备（打印机、显示器、键盘等）**

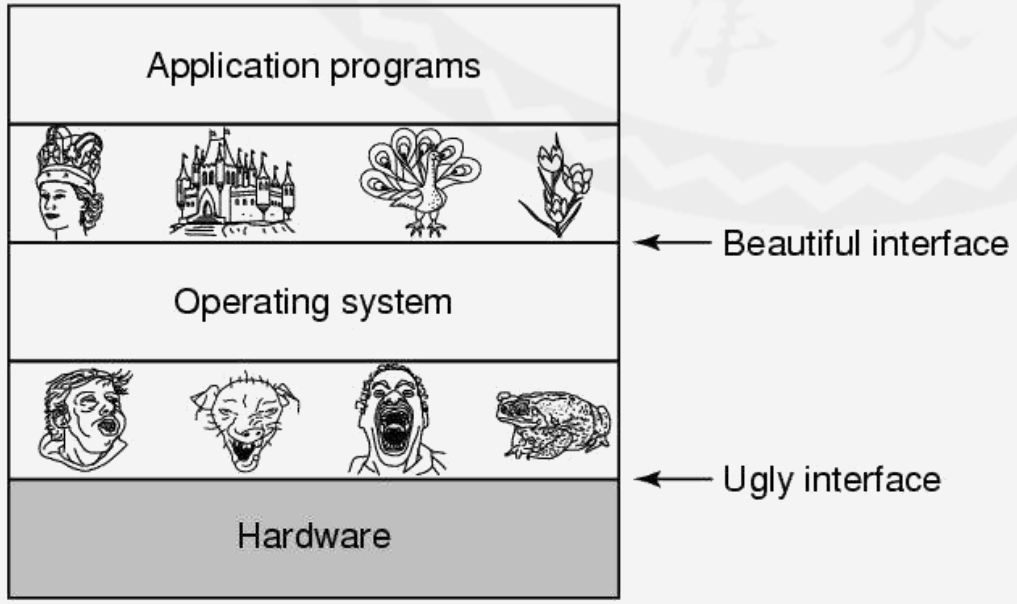
    Various input/output devices (printer, monitor, keyboard …)

- **需要有一层软件管理以上全部的组件**

  Managing all these components requires a layer of software

- **为各种奇怪硬件抽象为统一的软件接口**

  Turn ugly hardware into beautiful abstractions



Linux

- **1991年，Linus写出了第一个版本的Linux内核**

1991: Linus Torvalds writes 1st version of Linux kernel

- **Linus' UNIX -> Linux**
- **与GNU等工具进行结合，形成了一个完整的类UNIX系统**

Combined with the GNU and other tools forms a complete UNIX-like system

Linus Torvalds

Mascot: Tux

Linux操作系统

Linux Operating System

Linux的各发行版
Linux Distributions

- RedHat: Fedora, RHEL, CentOS
- Ubuntu: from Debian
- SuSE: OpenSuSE
- 麒麟
- OpenEuler： 欧拉
- ……

今天的Linux
Linux Today

- **Linux的应用领域覆盖了计算机领域的整个光谱**

  Linux covers the whole spectrum of computing

  - **嵌入式设备**

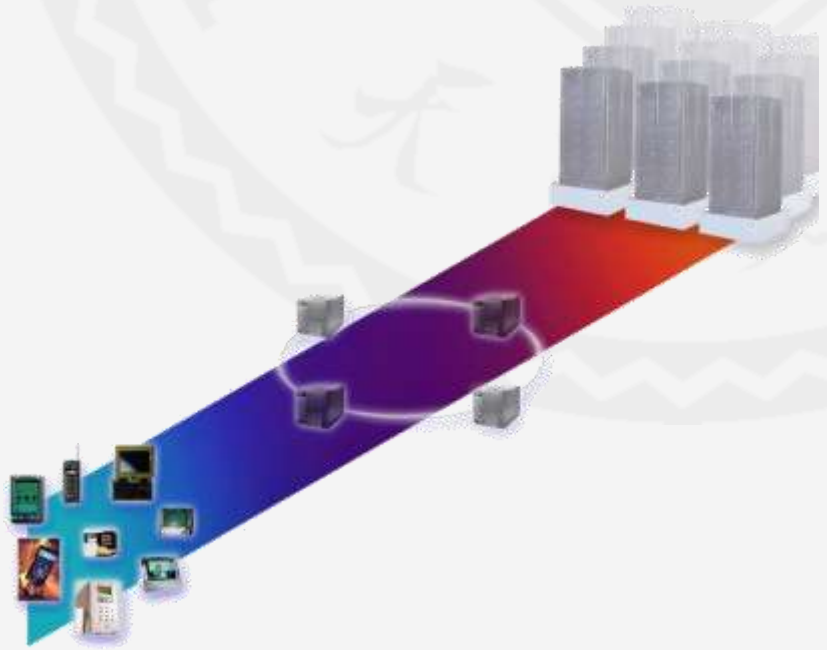
    Embedded devices
  - **笔记本电脑**

    Laptops
  - **桌面系统**

    Desktop systems
  - **小型和大型服务器**

    Small and large servers
  - **超级计算机/巨型集群**

    Megaclusters/supercomputers



Linux操作系统

Linux Operating System

通过终端命令操作Linux
Operating Linux through terminal commands

- **Linux支持文字界面和图形界面的操作**

  Linux supports both text-based and graphical user interfaces (GUI)

- **但图形服务对于Linux是可选择，不是所有的Linux操作系统都能够提供图形服务，例如：**

  However, GUIs are optional for Linux, not all Linux operating systems can provide GUI services. For example:

  - **远程服务器**

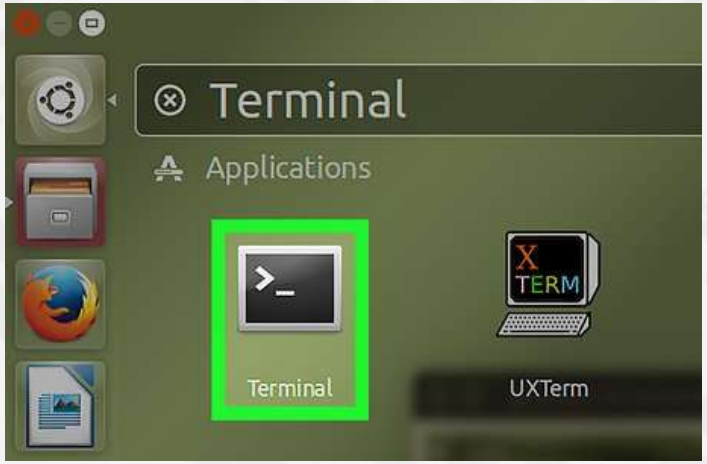
    Remote Servers

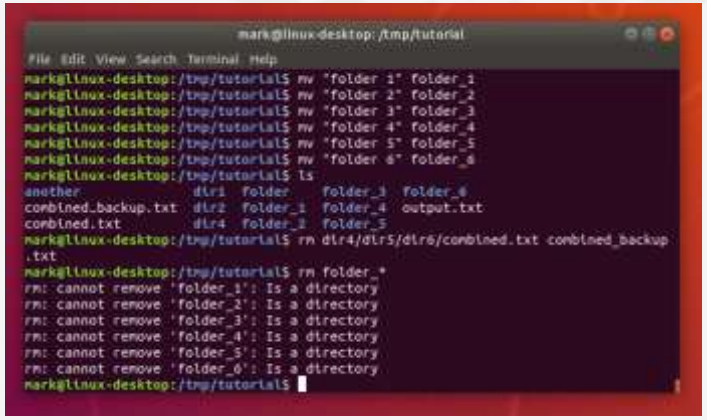
  - **嵌入式设备**

    Embedded Devices

- **通过终端使用命令操作Linux是必备的技能，也是高效的**

  The essential skill of operating Linux through the terminal using commands is not only necessary but also efficient.





工作目录
Working Directory

- **在进入字符终端后，用户始终处于一个特定的目录中，这个目录被称为工作目录或当前目录。**

After entering the character terminal, the user always resides in a specific directory, which is referred to as the working directory or current directory.

- **工作目录指的是用户当前正在处理的目录，在这个目录下进行的所有操作都是针对该目录进行的，因此得名工作目录。**

The working directory refers to the directory that the user is currently dealing with. All operations carried out within this directory are specific to it. Hence, it is called the working directory.

Linux操作系统

Linux Operating System

路径 Path

- **Linux 的目录结构为树状结构，最顶级的目录为根目录 /**

The directory structure of Linux is a tree, with the top-level directory being the root directory /.

- **使用路径表示Linux中资源（文件、目录、设备、……）所在的位置**

Path representation is used to indicate the location of resources (files, directories, devices, etc.) in Linux.

- **绝对路径：以根目录为起点描述资源的位置**

Absolute Path: Describes the location of resources starting from the root directory.

  - **由根目录 / 写起，例如： /usr/src这个目录**

  Example: /usr/src represents the directory "src" under the "usr" directory.

- **相对路径：以当前工作目录为起点描述资源的位置**

Relative Path: Describes the location of resources starting from the current working directory.

  - **表示方法：当前工作目录为 /usr/src，访问/usr/bin 时，可以写为 ../bin**

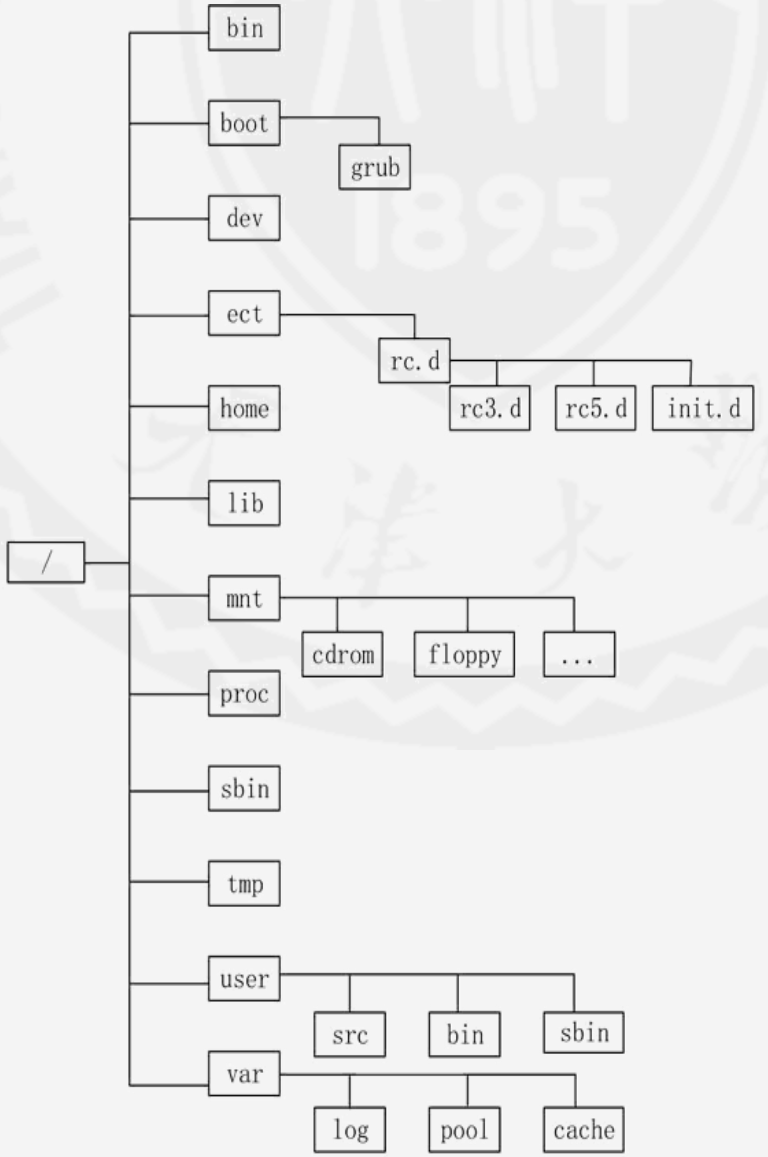
  Example: If the current working directory is /usr/src, accessing /usr/bin can be written as ../bin.

  - **“.” 表示当前目录**

  "." represents the current directory

  - **“..” 表示上一级目录**

  ".." represents the parent directory



文件和目录操作命令 （1）
File and Directory Operation Commands （1）

- **ls: 列出指定目录下的子目录及文件名称**

ls： List the names of subdirectories and files in the specified directory.

- **格式：ls [选项] 路径**

Format: ls [options] path

```
/usr$ ls
bin     include   lib32   libexec   local   share
games   lib       lib64   libx32    sbin    src
```

/usr$ 为命令行提示符，不同的系统默认提示符样式可能会有所不同，用户可以通过脚本配置提示符样式。
当前提示符样式的配置为：显示当前工作目录的信息 /usr，随后显示分割符 $

- 路径：需要列出的指定路径

Path: The specified path for which the listing is required.

  - 缺省：当前工作路径

  Default: Specifies the current working directory for the specified path.

- 选项-a： 连同隐藏文件(文件名开头为 . 的文件) 一起列出来

Option -a: List along with hidden files (filename starting with ".")

- 选项-l： 列出文件和目录的属性与权限等详细数据

Option -l: Display detailed information such as properties and permissions for files and directories.

文件和目录操作命令 （2）
File and Directory Operation Commands （2）

- **cd：改变当前工作目录**

  cd: Change the current working directory.

  - **格式：cd 路径**

    Format: cd path

```
/usr$ cd bin
```

- **pwd：显示当前工作目录**

  pwd： Print the current working directory.

```
/usr/bin$ pwd
/usr/bin
```

文件和目录操作命令 （3）
File and Directory Operation Commands （3）

mkdir：创建新的目录

mkdir: Create new directories.

- 格式：mkdir [选项] 路径

  Format: mkdir [options] path

```
/tmp$ mkdir test                # 在/tmp下创建test目录
/tmp$ mkdir -p test1/test2      # 在/tmp下创建test1目录
                                # 同时在test1下创建test2目录
```

rmdir：删除空的目录

rmdir: Remove empty directories.

- 格式： rmdir [选项] 路径

  Format: rmdir [options] path

```
/tmp$ rmdir test                # 删除/tmp下的test空目录
/tmp$ rmdir -p test1/test2      # 将test1/test2这两个空目录删除
```

文件和目录操作命令 （4）
File and Directory Operation Commands （4）

cp: 复制文件或目录

cp: Copy files or directories.

- **格式：cp [选项] 源路径1 源路径2 …… 源路径n 目标路径**

  Format: cp [options] source_path1 source_path2 …… source_pathn target_path

- **选项 -r：递归复制，用于目录的复制**

  Option -r: Recursively copy, used for copying directories.

- **选项 -f：若目标文件已经存在，则强行覆盖，不进行询问**

  Option -f: Force overwrite if the target file already exists, without prompting.

* 为通配符，用于匹配任意数量(包括零个)字符

```
/tmp$ cp f1 f2              # 将f1文件复制到/tmp目录下，文件名为f2
/tmp$ cp f1 f2 test         # 将f1、f2文件复制到/tmp/test目录下
/tmp$ cp -rf test/* test1   # 将/tmp/test目录下全部内容强制复制到/tmp/test1目录下
```

文件和目录操作命令 （5）
File and Directory Operation Commands （5）

rm: 删除文件或目录

rm: Delete files or directories.

- **格式：rm [选项] 路径1 路径2 …… 路径n**

Format: rm [options] path1 path2 …… pathn

```
/tmp$ rm f1          # 删除f1文件
/tmp$ rm f1 f2       # 删除f1、f2两个文件
/tmp$ rm -rf test    # 强制将/tmp/test目录及其下全部
                     # 内容删除，删除文件夹需要-r选项
```

mv: 移动文件与目录

mv: Move files and directories.

- **格式：mv 源路径1 源路径2 …… 源路径n 目标路径**

Format: mv source_path1 source_path2 …… source_pathn target_path

```
/tmp$ mv f1 f2 test # 将f1和f2移动至/tmp/test目录下
/tmp$ mv f1 test/f3 # 将f1移动至/tmp/test目录下
                    # 并将文件命名为f3
/tmp$ mv f1 f4      # 将f1移动至当前目录下，并命名为f4
                    # 等价于文件重命名
```

用户和用户组
Users and User groups

- **用户：Linux系统是一个多用户多任务的分时操作系统，任何一个要使用系统资源的用户，都必须首先向系统管理员申请一个账号，然后以这个账号的身份进入系统**

  User: The Linux system is a multi-user, multitasking time-sharing operating system. Any user who wants to use system resources must first apply for an account from the system administrator and then enter the system with this account.

- **根用户：根用户账号是root，拥有最高的管理权限。为了系统安全，通常不通过根用户登录系统**

  Root User: The root user account is named 'root' and has the highest administrative privileges. For security reasons, it is typically not recommended to log in directly as the root user.

- **用户组：每个用户都有一个用户组，系统可以对一个用户组中的所有用户进行集中管理**

  User Group: Each user belongs to a user group, and the system can centrally manage all users within a user group.

文件和目录权限
File and Directory Permissions

- **在Linux中，文件和目录具有相应的访问权限，目的是防止用户访问其他用户的私有文件以及保护重要的系统文件**

In Linux, files and directories have corresponding access permissions with the aim of preventing users from accessing other users' private files and protecting critical system files.

- **每个文件包含九个权限位，定义了所有者、所有者所在用户组和其他用户对文件的访问权限**

Each file in Linux contains nine permission bits, defining the access permissions for the owner, the group the owner belongs to, and other users.

```
~$ ls -l README.md
-rwxr--r-- 1 user user 54130 Jan  8 12:14 README.md
```

- **最前面的-表示README.md为文件，d表示目录，l表示符号链接**

The leading '-' indicates that README.md is a regular file, 'd' indicates a directory, and 'l' indicates a symbolic link.

- **前三位表示文件所有者的权限，中间三位表示文件所属组的权限，而最后三位适用于其他人的权限**

The first three bits represent the owner's permissions, the middle three bits represent the group's permissions, and the last three bits apply to other users' permissions.

- **r表示读取，w表示写入，x表示执行，“-”表示无权限**

'r' stands for read, 'w' stands for write, 'x' stands for execute, and "-" indicates no permission.

改变权限操作命令
Change Permissions Operation Command

- **chmod: 改变文件权限**

chmod: Change file permissions.

- **格式：chmod 模式 路径**

  Format: chmod mode path

- **模式：[ugoa...][[+-=][rwx]...][,...]**

  mode： [ugoa...][[+-=][rwx]...][,...]

  - **u 表示该文件的拥有者，g 表示与该文件的拥有者属于同用户组，o 表示其他用户，a 表示这三者皆是**

  'u' represents the file's owner; 'g' represents users in the same group as the file's owner; 'o' represents other users; 'a' represents all three

  - **+ 表示增加权限、- 表示取消权限、= 表示唯一设定权限**

  '+' adds permission, '-' removes permission, '=' sets the permission.

  - **r 表示可读取，w 表示可写入，x 表示可执行**

  'r' stands for read, 'w' stands for write, 'x' stands for execute.

```
~$chmod a+r file    # 给file的所有用户增加读权限
~$chmod a-x file    # 删除file的所有用户的执行权限
~$chmod a+rw file   # 给file的所有用户增加读写权限
~$chmod +rwx file   # 给file的所有用户增加读写执行权限
~$chmod u=rw,go= file # 对file的所有者设置读写权限，
                      # 清空该用户组和其他用户对file
                      # 的所有权限（空格代表无权限）
```

文件链接
File Links

- **Linux系统允许在文件之间创建链接。这种操作实际上是给系统中已有的某个文件指定另外一个可用于访问它的名称**

In the Linux system, it is allowed to create links between files. This operation is essentially assigning another name that can be used to access a file that already exists in the system.

- **链接可分为两种：硬链接与软链接（符号链接）**

Linking can be divided into two types: Hard links and Soft links (Symbolic links).

  - **硬链接类似于文件的别名**

  Hard links are akin to aliases for files.

  - **软链接更像是指向文件的快捷方式**

  Symbolic links are more like shortcuts pointing to files.

文件链接命令
Command for file linking

- **ln ： 创建文件链接**

ln: Create file links

  - **格式：ln [选项] 源路径 目标路径**

  Format: ln [options] source_path target_path

  - **选项 –f： 强制执行**

  Option -f: Force execution.

  - **选项 –s： 软链接(符号链接)**

  Option -s: Create a symbolic link (soft link).

```
~$ ln 1.log 2.log       # 为1.log创建硬链接2.log
                        # 两个文件指向同一内容

~$ ln -s 1.log 2.log    # 为1.log创建符号链接2.log
                        # 当1.log删除后，符号链接2.log将失效

~$ ln -sf 1.log 2.log   # 为1.log创建符号链接2.log
                        # 如果2.log已存在则强制覆盖
```

文件打包和拆包（1）
File Packaging and Unpackaging Commands（1）

文件打包

File Packaging

```
~$tar czf file.tar.gz  file/   #将file文件夹下的所有内容压缩为file.tar.gz，压缩算法为gzip
```

文件拆包

File Unpackaging

```
~$tar xzf file.tar.gz  # 将file.tar.gz 解压缩到当前文件夹
                       # (参数z可省略，由tar自动选择解压算法)
```

几种常用压缩算法所对应的参数

压缩算法	参数	文件名后缀
gzip	z	.tar.gz
bzip2	j	.tar.bz2
Lempress-Ziv	Z	.tar.Z

文件打包和拆包（2）
File Packaging and Unpackaging Commands（2）

将文件打包为zip

Package files into a zip file

```
~$ zip -q -r html.zip /home/html    # 将/home/html/ 这个目录下所有文件和文件夹打包为当前目录下的 html.zip
```

将zip文件拆包

Unpack a zip file

```
~$ unzip html.zip        # html.zip解压到当前文件夹
```

文件显示命令
File Display Commands

cat 将文件以文本方式输出至控制台

cat: Outputs the contents of a file to the console in text format.

hexdump 将文件以编码形式输出至控制台

hexdump: Outputs the contents of a file in hexadecimal format to the console.

head 显示文件前几行的数据

head: Displays the first few lines of a file.

tail 输出文件尾部的数据

tail: Outputs the end of a file.

objdump 查看elf格式文件

objdump: Views the contents of an ELF format file.

more 分页显示文件

more: Displays a file page by page.

以上命令参数都为文件路径

文件搜索命令
File Search Commands

find：根据条件搜索文件

find: Search for files based on conditions

```
~$ find ./test1 -name a.txt     # 在当前目录下的test1子目录中搜索a.txt文件

~$ find . -name "*.cpp"         # 在当前目录下的test1搜索所有后缀为.cpp的文件
```

grep：根据条件搜索文件内容（常用于搜索文本文件）

grep: Search for file contents based on conditions (commonly used for searching text files).

```
~$ grep hello file.txt          # 在文件 file.txt 中查找字符串 "hello"，并打印匹配的行

~$ grep main -r *.cpp           # 在当前目录下及其子目录下搜索所有后缀为.cpp的文件
                                # 并打印这些文件中包含字符串 “main”的行
```

使用根用户权限执行命令
Execute a command with root user privileges

- **sudo：以根用户权限执行一条命令（执行前需要输入当前用户的密码）**

sudo: Execute a command with root user privileges (Password for the current user is required before execution)

- **格式：sudo 要执行的指令**

Format: sudo command_to_execute

```
~$ sudo mkdir test          # 以根用户权限创建一个目录test

                            # test的所有者是root，当前账号无法操作该目录
```

注意事项
Notes

- **尝试使用TAB健补全命令和路径**

Try using the TAB key for command and path completion

- **不要自以为按照资料上的要求输入命令就一定会执行成功，要随时观察命令的执行结果**

Don't assume that entering commands according to the instructions in the documentation will always lead to successful execution. Always observe the results of command execution

- **命令和参数输入错误很常见：拼写错误、空格、中英文符号混用等都有可能是错误产生的原因**

Command and parameter input errors are common: spelling mistakes, spaces, and the mixed use of Chinese and English symbols can all be reasons for errors.

练习
Exercises

- **在命令中使用输出重定向符号 > 可以将标准输出的内容写入文件**

The use of the output redirection symbol > in a command allows writing the content of standard output to a file.

```
~$ ls > abc        # abc 为文件名
```

- **要求：**

Requirement:

  - **在用户主目录下~创建my_test目录**

  Create a directory named my_test in the user's home directory (~).

  - **使用列出 /bin目录下全部内容，并写入 result.txt 文件**

  Use the ls command to list all contents of the /bin directory and write them to the result.txt file.

  - **使用grep命令搜索result.txt中所有包含字符‘a’的内容，并将结果写入result2.txt文件**

  Use the grep command to search for all content containing the character 'a' in result.txt and write the results to the result2.txt file.

  - **使用tar命令，对my_test目录进行打包，生成my_test.tar.gz**

  Use the tar command to package the my_test directory, creating my_test.tar.gz.

本章内容

Topic

- Linux操作系统

  Linux Operating System
- 文本编辑器 vi/vim

  Text Editor vi/vim
- GNU 工具链

  GNU Toolchain
- 其他常用工具

  Other Commonly Used Tools
- 代码版本管理

  Code Version Control

文本编辑器 vi/vim

Text Editor vi/vim

VI是基于字符界面的文本编辑器，VIM是VI的增强版，可以在非图形环境下使用，是Linux中最常用的编辑器之一

VI is a character-based text editor, while VIM is an enhanced version of VI that can be used in non-graphical environments. It is one of the most commonly used editors in Linux.

执行以下命令安装VIM

Execute the following command to install VIM

```
$ sudo apt install vim
```

文本编辑器 vi/vim

Text Editor vi/vim

启动VIM
Launch VIM

- **在命令行输入vi，即创建一个新的未命名文件**

  Enter 'vi' in the command line to create a new unnamed file.

```
$ vi
```

- **或者，输入vi 文件路径，打开一个已存在/新创建的文件**

Alternatively, enter 'vi file_path' in the command line to open an existing/newly created file.

```
$ vi a.c
```

文本编辑器 vi/vim

Text Editor vi/vim

VIM的几种工作模式
Various Modes in VIM

- **普通模式： 用于导航、删除、复制等操作，是默认模式**

Normal Mode: Used for navigation, deletion, copying, and other operations; it is the default mode.

- **插入模式： 用于输入文本，在普通模式下按 "i" 键进入**

Insert Mode: Used for entering text; press the "i" key in Normal Mode to enter.

- **查找模式： 用于搜索文本，在普通模式下按 “/” 或 “?" 键进入**

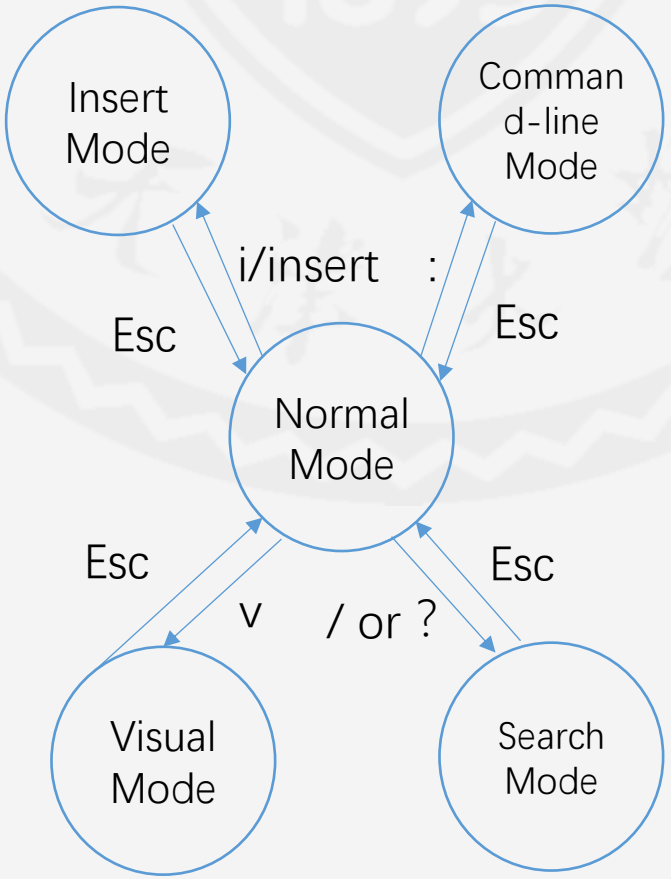
Search mode: Used for searching text. Press the "/“ or “?” key in Normal mode to enter.

- **命令模式： 用于执行保存、退出等高级命令，在普通模式下按 ":" 键进入**

Command-Line Mode: Used for executing advanced commands such as saving and quitting; press the ":" key in Normal Mode to enter.

- **可视模式： 用于选择文本块，可以是字符、行或块，在普通模式下按 "v" 键进入**

Visual Mode: Used for selecting text blocks, which can be characters, lines, or blocks; press the "v" key in Normal Mode to enter



文本编辑器 vi/vim

Text Editor vi/vim

普通模式下的操作
The Operations in Normal Mode

- **上下左右键移动光标**

Move the cursor using the arrow keys.

- **Home 移动至行首，End 移动至行尾**

Home moves to the beginning of the line, End moves to the end of the line.

- **PageUp 向上翻页，PageDown 向后翻页**

PageUp scrolls up, PageDown scrolls down.

- **gg 移动至文件开始，Shift+g移动至文件结尾**

gg moves to the start of the file, Shift+g moves to the end of the file.

- **yy 复制当前行，dd 剪切当前行**

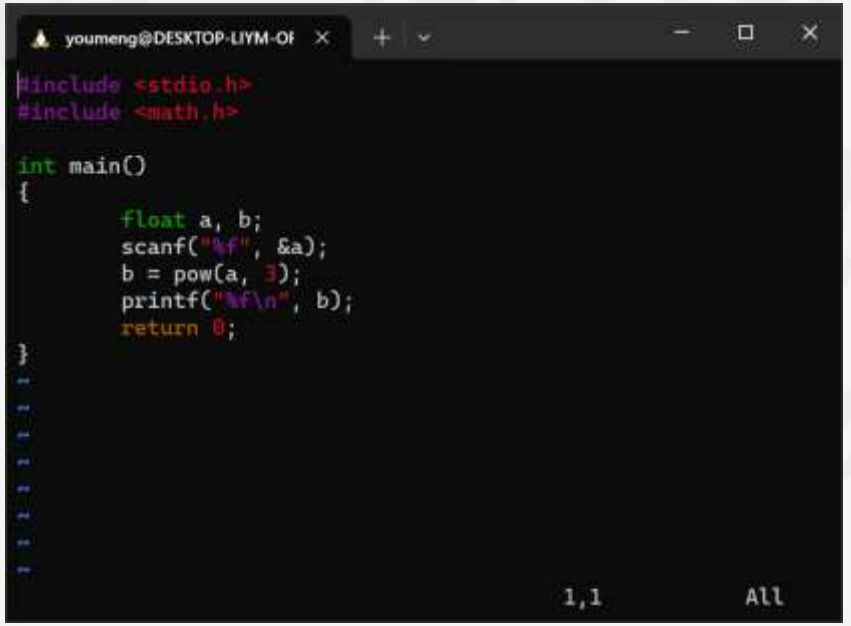
yy copies the current line, dd cuts the current line.

- **p 将剪贴板的内容粘贴至光标后**

p pastes the content from the clipboard after the cursor.

- **u 撤销上一步操作, Ctrl + r 恢复上一次撤销操作**

u: Undo the last operation，Ctrl + r: Redo the last undo operation.



文本编辑器 vi/vim

Text Editor vi/vim

插入模式下的操作
The Operations in Insert Mode

- **在普通模式下，按 i键或Insert键进入插入模式**

In Normal mode, press the i key or the Insert key to enter Insert mode.

- **屏幕底部显示--INSTERT--**

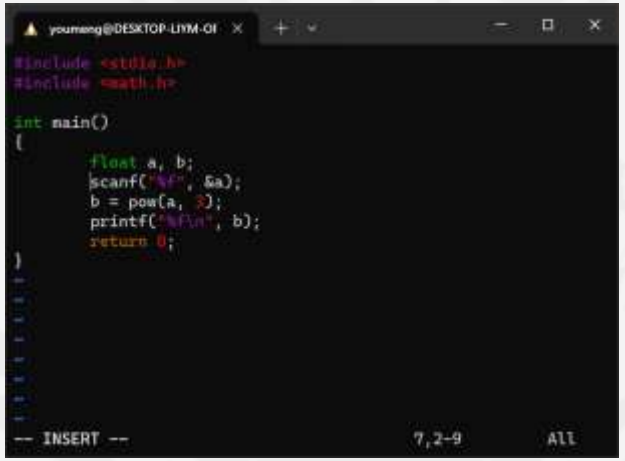
At the bottom of the screen, "--INSERT--" is displayed.

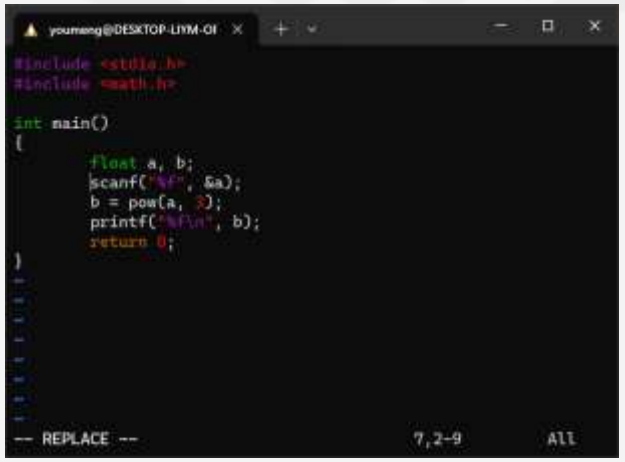
- **在插入模式下支持方向键、Home/End、PageUp/PageDown进行光标移动**

In Insert mode, you can use the arrow keys, Home/End, and PageUp/PageDown to move the cursor.

- **在插入模式下，按Insert键可以实现插入和替换模式的切换**

In Insert mode, pressing the Insert key toggles between insert and replace modes.





查找模式下的操作
The Operations in Search Mode

- **在普通模式下，按 / 键进入正向查找模式，按 ? 键进入反向查找模式**

In Normal mode, press the / key to enter forward search mode, and press the ? key to enter reverse search mode.

- **屏幕底部显示 /（正向查找模式） 或 ?（反向查找模式）**

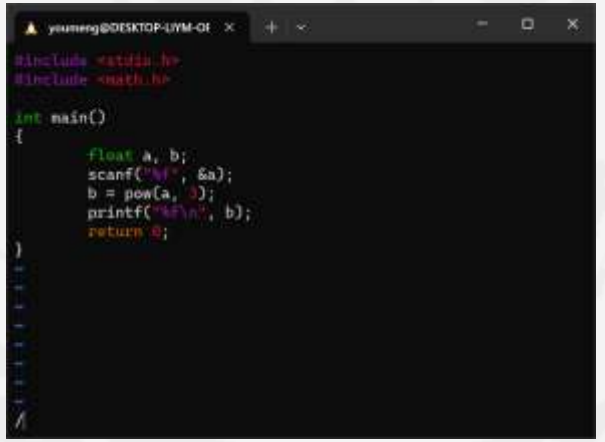
At the bottom of the screen, it displays / (for forward search mode) or ? (for reverse search mode).

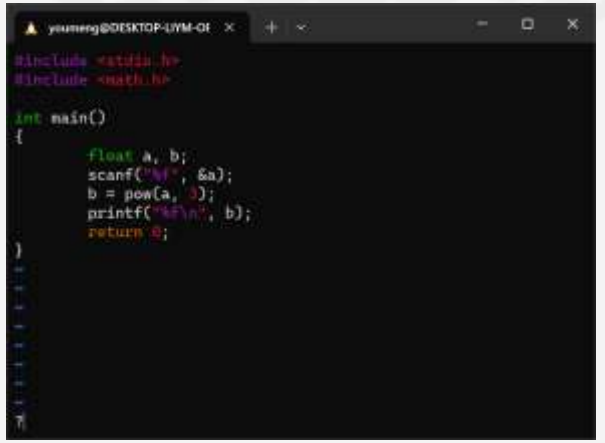
- **在提示符 / 或 ? 后输入需要查找的字符串，并按回车键进行查找**

After entering the prompt / or ?, input the desired search string and press Enter to initiate the search.

- **按 n 键查找下一项，按Shift + n 查找上一项**

Press the 'n' key to find the next match, and press 'Shift + n' to find the previous match.





文本编辑器 vi/vim

Text Editor vi/vim

命令模式下的操作
The Operations in Command-Line Mode

- **在普通模式下，按：键进入命令模式**

In Normal mode, press the ':' key to enter command-line mode.

- **屏幕底部显示：，输入命令并回车即可执行命令**

At the bottom of the screen, it displays ‘:’， enter the command and press Enter to execute it.

- **w 保存当前文件，q退出，wq保存并退出**

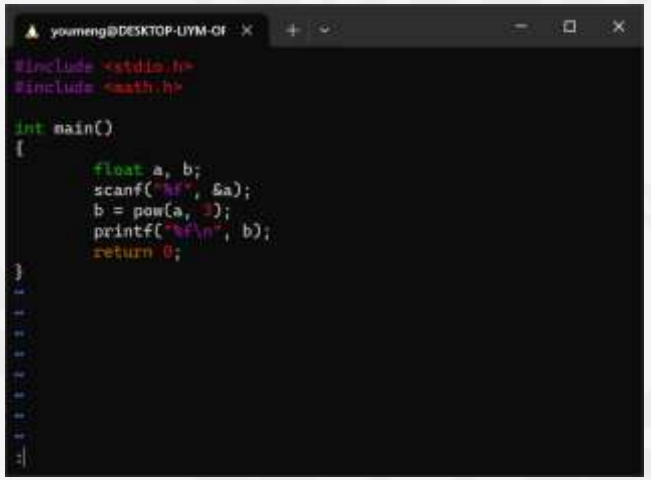
"w" saves the current file, "q" exits, "wq" saves and exits.

- **w !强制保存，wq !强制保存并退出 （用于只读文件）；q! 退出不保存**

"w!" forces the save, and "wq!" forces save and exits (useful for read-only files); "q!" exit without saving.

- **数字：跳转至指定行**

Number: Navigate to the specified line.



文本编辑器 vi/vim

Text Editor vi/vim

可视模式下的操作
The Operations in Visual Mode

- **在普通模式下，按 v 键进入可视模式**

In Normal mode, press the ‘v' key to enter visual mode.

- **屏幕底部显示 --VISUAL--**

At the bottom of the screen, it displays ‘—VISUAL--’.

- **移动光标可进行文本块的选择**

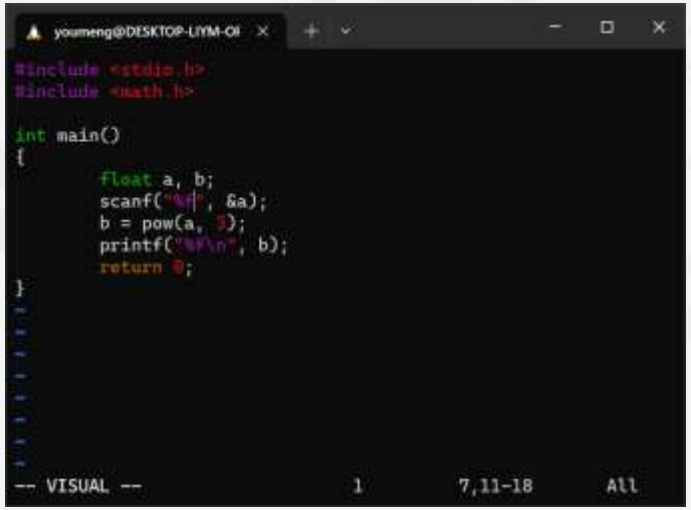
Moving the cursor allows you to select a text block.

- **y键复制文本块，d键剪切文本块**

Pressing 'y' copies the selected text block, and 'd' cuts (deletes) the text block.

- **在普通模式下，p键粘贴文本块**

In Normal mode, pressing 'p' pastes the copied or cut text block.



练习
Exercises

- 使用 vi 编写C程序Hello World
Write a C program “Hello World” using the vi editor.

```
#include <stdio.h>

int main()

{

  printf("Hello World!\n");

  return 0;

}
```
main.c

本章内容

Topic

- Linux操作系统

  Linux Operating System
- 文本编辑器 vi/vim

  Text Editor vi/vim
- GNU 工具链

  GNU Toolchain
- 其他常用工具

  Other Commonly Used Tools
- 代码版本管理

  Code Version Control

GNU 工具链

GNU Toolchain

GNU = GNU is not Unix

GNU 工具链简介
Introduction to the GNU Toolchain

- **GNU工具链是一组开源的编程工具，常用于开发和构建软件。这个工具链由GNU计划提供，其中包含了许多用于编译、调试和构建软件的实用程序**

The GNU Toolchain is a set of open-source programming tools commonly used for software development and building. Provided by the GNU Project, this toolchain includes many utilities for compiling, debugging, and constructing software.

- **GCC：一个功能强大的编译器集合，支持多种编程语言**

GCC：A powerful collection of compilers supporting multiple programming languages

- **GNU Binutils：包括汇编器、连接器、目标文件转换工具等**

GNU Binutils：including as (assembler), ld (linker), objcopy (object file manipulation tool), and others.

- **GDB：用于调试程序的强大工具，可以支持多种编程语言**

A robust tool for debugging programs, with support for various programming languages.

- **Make 一个用于构建和管理项目的工具，通过Makefile文件描述构建规则**

A tool for building and managing projects, defining build rules through Makefile files.

在Ubuntu中安装GNU工具链
Install the GNU toolchain on Ubuntu

```
$ sudo apt-get install build-essential    # 基本开发包，包括 gcc、make、二进制工具等
$ sudo apt-get install gdb                # GNU 调试器
```

安装软件需要使用根用户权限
Installing software requires using root user privileges

第一次使用apt安装软件前需要执行以下命令更新索引
Before using apt to install software for the first time, execute the following command to update the index.

```
$ sudo apt update
```

GNU 工具链

GNU Toolchain

构建由单个源文件组成的程序
Building a program consisting of a single source file

方法一：最简单的方法

Method 1：The simplest method

```
$ gcc main.c     # 编译并链接main.c,默认生成文件名为a.out
$ ./a.out        # 运行a.out文件
Hello World      # 程序输出
```

默认的文件名称为a.out

方法二：指定生成程序的名称

Method 2：Specify the name of the generated program

```
$ gcc -o main main.c # 编译并链接main.c,生成main文件
$ ./main             # 运行main文件
Hello World          # 程序输出
```

-o 用于指定生成文件的名称

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

main.c

Linux中，当前工作目录不是默认搜索路径，即使运行当前工作目录下的程序也需要包含路径信息，如：./a.out

In Linux, the current working directory is not in the default search path. Therefore, even when running a program from the current working directory, you need to include the path information, such as: ./a.out.

GNU 工具链

GNU Toolchain

构建由多个源文件组成的程序
Building a program consisting of a single source file

方法一：最简单的方法

Method 1：The simplest

```
$ gcc -o p a.c b.c   # 编译并链接a.c、b.c,生成p文件
$ ./p                # 运行p文件
25                   # 程序输出
```

```
#include <stdio.h>
int power(int a);

int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```
a.c

方法二：先编译，后链接

Method 2：Compile first, then link

```
$ gcc -c a.c b.c     # 将a.c和b.c进行编译
                     # 并生成目标文件a.o和b.o
$ gcc -o p a.o b.o   # 将目标文件a.o和b.o进行链接，生成p
$ ./p                # 运行p文件
25                   # 程序输出
```

```
int power(int a)
{
   return a*a;
}
```
b.c

增量编译
Incremental Compilation

只编译b.c，然后链接
Compile only b.c, then link.

```
$ gcc -c b.c          # 将b.c编译为目标文件b.o
$ gcc -o p a.o b.o    # 重新生成的b.o和原先的a.o进行链接
$ ./p                 # 运行p文件
125                   # 程序输出
```

```c
#include <stdio.h>
int power(int a);

int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```
a.c

```c
int power(int a)
{
   return a*a*a;
}
```
b.c

引用自定义头文件
Include a custom header file

```
$ gcc -c a.c -Iinc    # 将a.c编译为目标文件a.o
                      # 并使用-I指定在inc目录下搜索头文件
$ gcc -c b.c          # 将b.c编译为目标文件b.o
$ gcc -o p a.o b.o    # a.o和b.o进行链接，生成p文件
$ ./p                 # 运行p文件
125                   # 程序输出
```

```c
#include <stdio.h>
#include <global.h>
int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```
a.c

```c
int power(int a)
{
   return a*a*a;
}
```
b.c

```c
int power(int a);
```
inc/global.h

链接第三方库
Linking with a Third-Party Library

```
$ gcc -c a.c          # 将a.c编译为目标文件a.o
$ gcc -o p a.o -lm    # a.o和数学库进行链接，生成p文件
                      # 使用-l链接数学库
$ ./p                 # 运行p文件
5.5                   # 输入
166.375000            # 输出
```

```c
#include <stdio.h>
#include <math.h>

int main()
{
    float a, b;
    scanf("%f", &a);  // 从标准输入中读取a
    b = pow(a, 3);    // 使用数学库中的pow
    printf("%f\n", b);
    return 0;
}
```
a.c

-l 用于指定链接库的名称，m为数学库的名称
The -l option is used to specify the name of a library, and 'm' is used for the name of the math library.
Linux中，库文件命名格式为libxxx.a (静态库）或 libxxx.so（动态库），xxx即为库的名称
In Linux, the naming convention for library files is libxxx.a for static libraries or libxxx.so for dynamic libraries, where xxx is the name of the library.
如果库不在默认搜索路径，则需要使用-L指定路径，类似于-l
If the library is not in the default search path, use -L to specify the path, similar to the way -l is used.

更多关于库的内容会在第七章链接中介绍

使用Makefile实现自动化编译 （1）
Implementing Automated Compilation with Makefile （1）

编译过程如下:
The compilation process is as follows:

```
$ gcc -c a.c -Iinc
$ gcc -c b.c
$ gcc -o p a.o b.o
```

简单的自动化，编写一个shell脚本
Simple Automation: Writing a Shell Script.

```
#!/bin/bash
gcc -c a.c -Iinc
gcc -c b.c
gcc -o p a.o b.o
```
compile.sh

```
$ chmod a+x compile.sh   # 增加执行权限
$ ./compile.sh           # 运行编译脚本
```

```
#include <stdio.h>
#include <global.h>
int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```
a.c

```
int power(int a)
{
   return a*a*a;
}
```
b.c

```
int power(int a);
```
inc/global.h

使用Makefile实现自动化编译 （2）
Implementing Automated Compilation with Makefile （2）

- **当项目中涉及的文件和依赖更多时，编译脚本就会出现局限性：**

When the project involves more files and dependencies, limitations arise in the compilation script:

  - **更复杂的脚本，脚本内部有很多重复的命令和选项**

    More complex scripts may have internal redundancy with repeated commands and options

  - **无法实现增量编译，提高编译效率**

    Incremental compilation, to enhance efficiency, becomes unachievable

  - **无法做编译之外的其他工作，如：清除上一次的编译结果**

    It's not possible to perform tasks beyond compilation, such as clearing the results of the previous build.

使用Makefile实现自动化编译 （3）
Implementing Automated Compilation with Makefile （3）

GNU Make：一个基于Makefile脚本的自动化编译工具

GNU Make: An Automated Compilation Tool Based on Makefile Scripts

1. **在工程的根目录下，创建一个Makefile文件**

   Create a Makefile in the project's root directory.

2. **编写Makefile文件**

   Write the Makefile.

3. **在Makefile文件所在目录下，运行make命令，进行编译**

   In the directory where the Makefile is located, run the make command to perform the compilation.

变量声明

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p
```

规则定义

```
$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

Makefile

使用Makefile实现自动化编译 （4）
Implementing Automated Compilation with Makefile （4）

- **Makefile中的变量：CC、CFLAGS、LDFLAGS、SRCS、OBJS、TARGET**

Variables in the Makefile: CC, CFLAGS, LDFLAGS, SRCS, OBJS, TARGET

  - **赋值即声明**

    Assignment is also a declaration

  - **在使用时: $(变量名)**

    When using, use $(variable name)

- **$(SRCS:.c=.o) 表示 将 $(SRCS) 变量中所有的.c替换为.o**

$(SRCS:.c=.o) means replacing all occurrences of .c with .o in the $(SRCS) variable

  - **等价于 OBJS = a.o b.o**

    Equivalent to OBJS = a.o b.o

变量声明

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

Makefile

使用Makefile实现自动化编译 （5）
Implementing Automated Compilation with Makefile （5）

Makefile中的规则

Rules in the Makefile

格式:
Format:

```
target: prerequisites
[ TAB ]command
```

依赖 / 目标 / 命令

目标：该条规则的所实现的目标（可以为文件名，也可以是伪目标）

Target: The goal achieved by this rule (can be a filename or a phony target).

依赖：生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件

Prerequisites: The list of filenames needed to generate the rule's target. Typically, a target depends on one or more files.

命令：规则所要执行的动作（任意的 shell 命令或者是可在 shell 下执行的程序）

Command: The action to be executed by the rule (any arbitrary shell command or a program executable in the shell).

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

规则定义

Makefile

使用Makefile实现自动化编译 （6）
Implementing Automated Compilation with Makefile （6）

- **编译规则**

  Compilation Rule

  - **目标 %.o：Makefile目录下的.o文件，%为通配符**

    Target %.o: .o files in the Makefile directory, where % is a wildcard.

  - **依赖 %.c ： Makefile目录下对应的.c文件**

    Prerequisites %.c: Corresponding .c files in the Makefile directory.

  - **$@ 表示目标，$^表示所有的依赖文件，$<表示第一个依赖文件**

    $@ represents the target, $^ represents all the prerequisites, $< represents the first prerequisite file.

- **规则说明：将当前目录下所有的.c使用命令分别编译为.o文件**

  Rule Description: Compile all .c files in the current directory into .o files using a respective command.

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

编译规则

Makefile

使用Makefile实现自动化编译 （7）
Implementing Automated Compilation with Makefile （7）

- **链接规则：将所有的OBJS链接生成TARGET**

  Linking Rule： Linking Rule: Link all the OBJS to generate the TARGET.

  - **链接规则必须在编译规则执行完，所有的.o文件准备好后才能执行**

    Linking rule must be executed after the compilation rule, when all the .o files are ready.

- **清理规则：删除所有的OBJS和TARGET文件**

  Clean Rule: Delete all OBJS and TARGET files.

  - **清理规则没有依赖条件，在任何情况下都可以执行**

    The clean rule has no prerequisites and can be executed under any circumstances.

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

链接规则

清理规则

Makefile

使用Makefile实现自动化编译 （8）
Implementing Automated Compilation with Makefile （8）

- **在执行 make 命令时，make 工具会按照 Makefile 中规则的依赖关系和指定顺序执行规则。默认情况下，make 将尝试构建第一个目标。然后，它会递归地处理该目标的依赖关系，并按照依赖关系的规则进行构建。**

When executing the make command, the make tool will execute rules according to the dependencies and specified order in the Makefile. By default, make attempts to build the first target. Subsequently, it recursively processes the dependencies of that target and builds them according to the rules specified in their dependencies.

- **也可以在参数中指定make所构建的目标，例如，使用 make clean执行清理规则**

Indeed, it's possible to specify the target that make should build by providing it as an argument. For example, using **make clean** would execute the clean rule for cleaning up files.

```
CC = gcc
CFLAGS = -Iinc
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

第一个目标
$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<


clean:
    rm -f $(OBJS) $(TARGET)
```

Makefile

使用Makefile实现自动化编译 （9）
Implementing Automated Compilation with Makefile （9）

- **make 通过比较源文件和目标文件的时间戳来确定哪些文件需要重新编译**

The make tool determines which files need to be recompiled by comparing the timestamps of source and target files.

- **运行 make 时，它会检查每个源文件和相应的目标文件的最后修改时间。如果源文件的修改时间比目标文件新，或者目标文件不存在，make 将执行相应的编译规则**

When running make, it checks the last modification time of each source file and its corresponding target file. If the modification time of the source file is newer than that of the target file, or if the target file doesn't exist, make will execute the corresponding compilation rule.

- **如果源文件没有被修改则无需重新编译。这提高了编译的效率，只重新编译那些发生变化的文件**

If a source file has not been modified, there is no need for recompilation. This enhances the efficiency of the compilation process, as only files that have undergone changes are recompiled.

在编译程序时增加调试信息
Adding debugging information during program compilation

- **增加编译选项 –g，通知编译器在生成目标文件时包含调试信息，以便在调试程序时能够查看源代码、设置断点等。**

Add the compilation option -g, instructing the compiler to include debugging information in the generated object files. This enables inspecting the source code, setting breakpoints, and performing debugging activities when debugging the program.

```
CC = gcc
CFLAGS = -Iinc -g
LDFlAGS =
SRCS = a.c b.c
OBJS = $(SRCS:.c=.o)
TARGET = p

$(TARGET): $(OBJS)
    $(CC) $(LDFlAGS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET)
```

Makefile

GNU 工具链

GNU Toolchain

使用GDB调试程序
Using GDB to debug programs

```
$ make      # 构建程序，生成可执行文件p
$ gdb p     # 使用gdb调试程序p
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from p...
(gdb)
```

进入gdb控制台

```c
#include <stdio.h>
#include <global.h>
int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```

a.c

```c
int power(int a)
{
   return a*a*a;
}
```

b.c

```c
int power(int a);
```

inc/global.h

使用GDB调试程序
Using GDB to debug programs

```
(gdb) b b.c:3          设置断点，在b.c文件的第三行
Breakpoint 1 at 0x40115c: file b.c, line 3.
(gdb) r                运行程序至断点处停止
Starting program: /home/youmeng/test/p

Breakpoint 1, power (a=5) at b.c:3
3           return a*a*a;
(gdb) p a              打印断点处变量a的值
$1 = 5
(gdb) s                执行一行语句
4       }
(gdb) s                执行一行语句
main () at a.c:6
6           printf("%d\n", a);
(gdb) continue         执行到下一个断点处，如无端点，则执行至程序结束
Continuing.
125                    程序输出
[Inferior 1 (process 139) exited normally]
(gdb) q                退出gdb控制台
```

```c
#include <stdio.h>
#include <global.h>
int main()
{
  int a = power(5);
  printf("%d\n", a);
  return 0;
}
```
a.c

```c
int power(int a)
{
   return a*a*a;
}
```
b.c

```c
int power(int a);
```
inc/global.h

练习
Exercises

编写Makefile，完成对main.c的自动化编译

Write a Makefile to automate the compilation of main.c.

```
#include <stdio.h>

int main()

{

  printf("Hello World!\n");

  return 0;

}
```

main.c

本章内容

Topic

- Linux操作系统

  Linux Operating System
- 文本编辑器 vi/vim

  Text Editor vi/vim
- GNU 工具链

  GNU Toolchain
- 其他常用工具

  Other Commonly Used Tools
- 代码版本管理

  Code Version Control

其他常用工具

Other Commonly Used Tools

通过SSH协议远程访问Linux
Remote access to Linux via the SSH protocol

SSH，安全外壳协议，是一种在不安全网络上用于安全远程登录和其他安全网络服务的协议。

SSH, Secure Shell Protocol, is a protocol used for secure remote login and other secure network services over an insecure network.

在Ubuntu上安装SSH服务

Installing SSH Service on Ubuntu.

```
$ sudo apt install openssh-server
```

实验平台上的Linux镜像已配置好ssh服务，不需要用户独立安装

常用的SSH客户端：Putty、Xshell （远程终端），FileZilla （文件传输）

Common SSH clients: Putty, Xshell (remote terminal), FileZilla (file transfer).

其他常用工具

Other Commonly Used Tools

使用Putty进行远程登录（1）
Remote Login with PuTTY（1）

- **登录必需的四个要素：IP地址、端口、账号、密码**

The four essential elements for login: IP address, port, username, and password.

  - **IP地址：远程Linux主机的地址**

    IP Address: The address of the remote Linux host.

  - **端口SSH服务端口，默认为22**

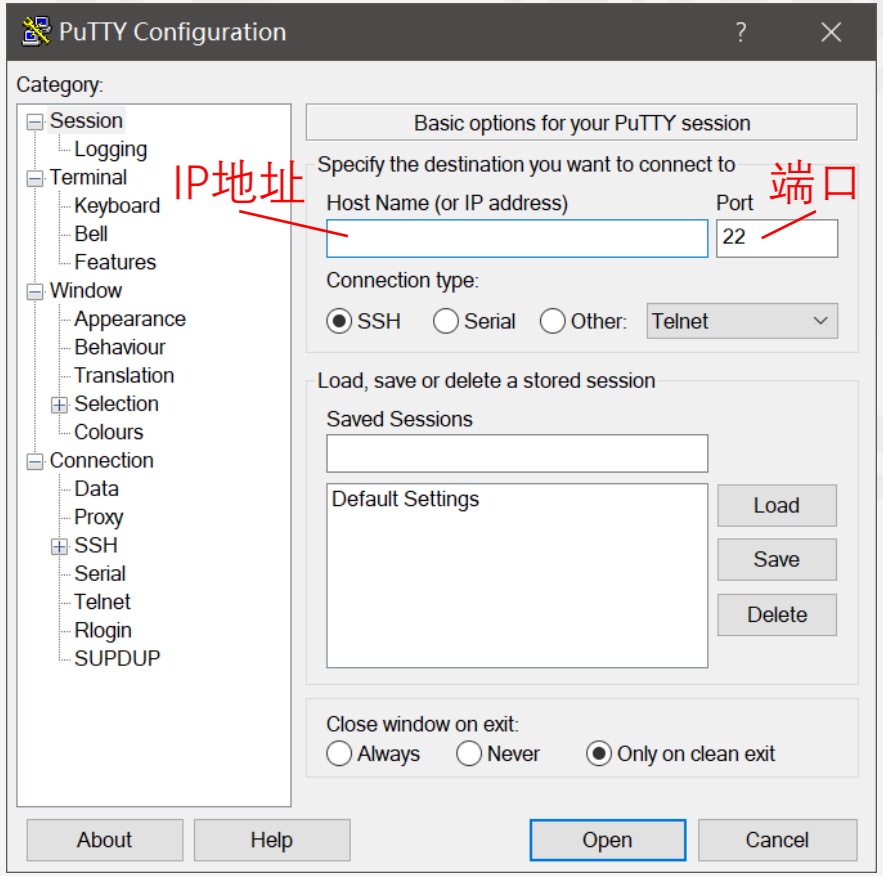
    Port: SSH service port, typically 22.

  - **账号：登录Linux的账号**

    Username: The account used to log in to Linux.

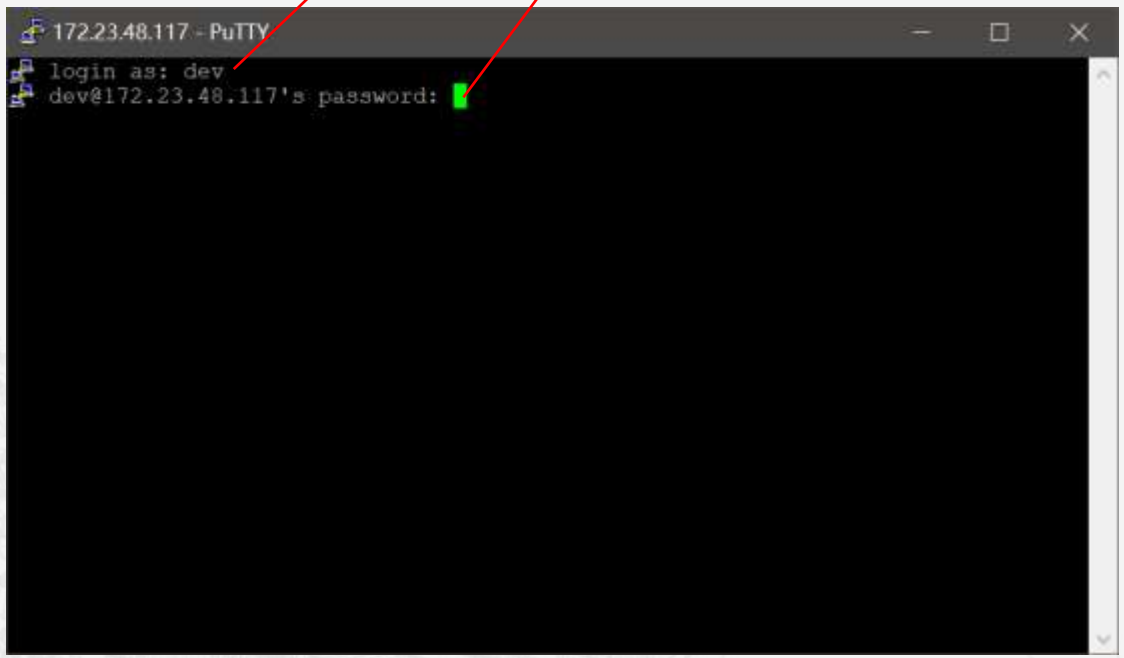
  - **密码：账号所对应的密码**

    Password: The password associated with the account.

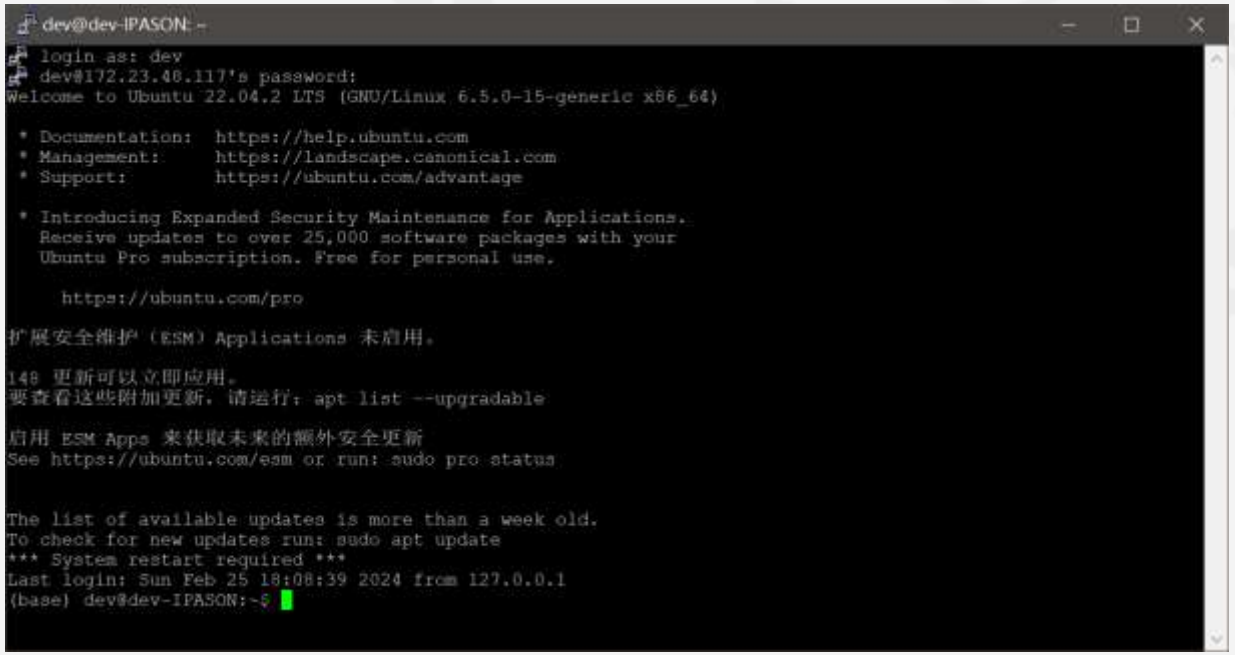


使用Putty进行远程登录（2）
Remote Login with PuTTY（2）

账号　　密码，Linux中密码输入无回显



输入账号和密码



登录成功

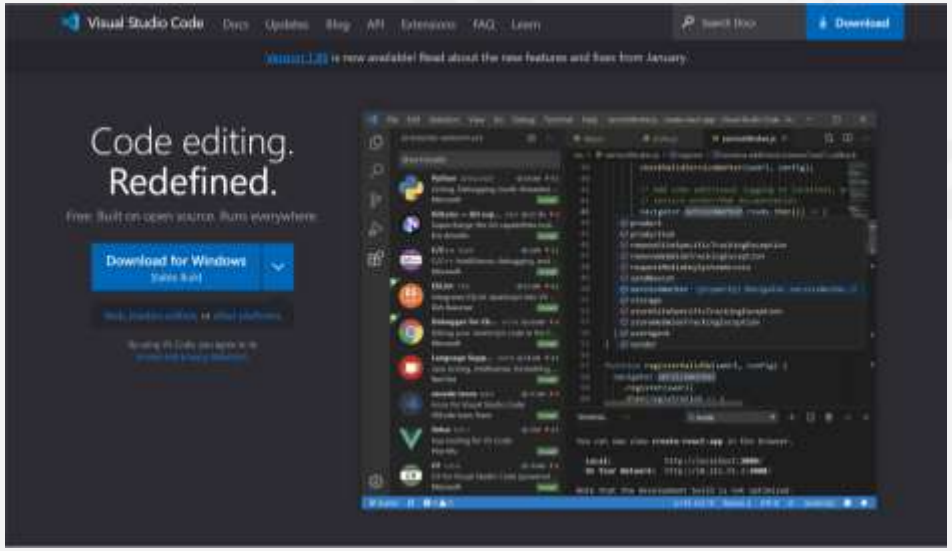
其他常用工具

Other Commonly Used Tools

开发利器 Visual Studio Code
Powerful Development Tool: Visual Studio Code

- **Visual Studio Code（VS Code）是一个开源的、跨平台编辑器，支持多种开发语言**

Visual Studio Code (VS Code) is an open-source, cross-platform editor that supports multiple programming languages.



- URL：https://code.visualstudio.com/

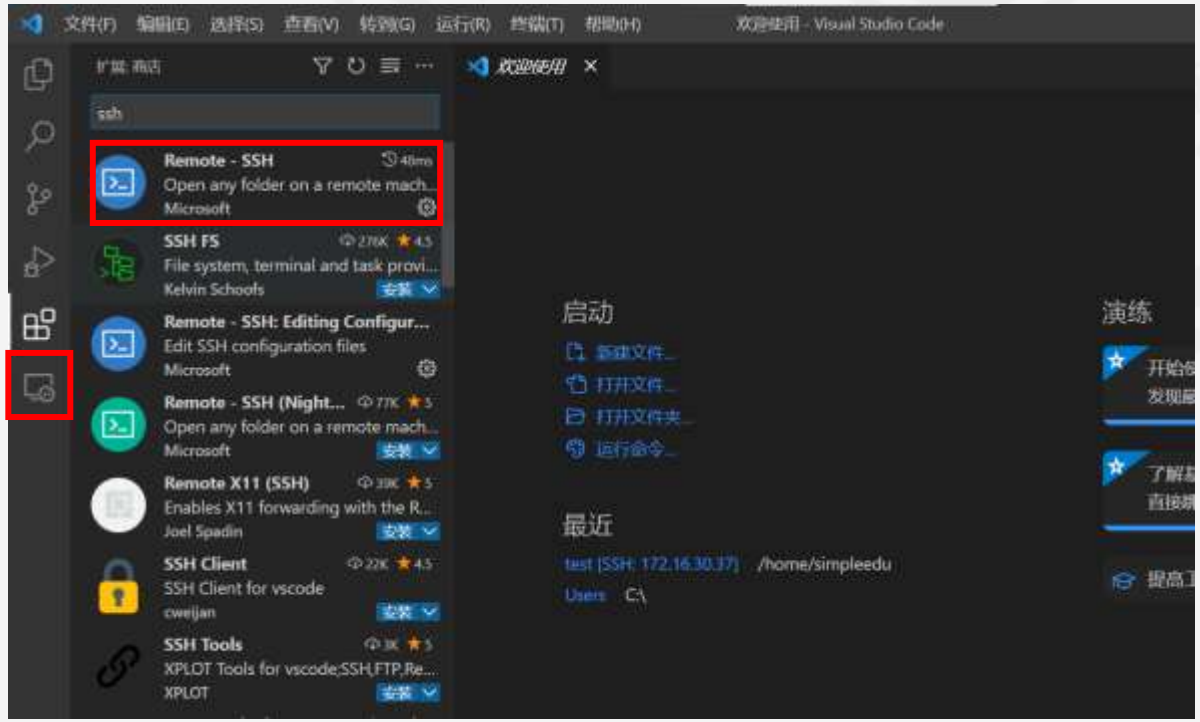
其他常用工具

Other Commonly Used Tools

在本地VS Code上连接远程Linux进行开发（1）
Developing on Remote Linux by Connecting to VS Code Locally（1）

- **第一步：为VS Code安装扩展 Remote-SSH，安装完成后VS Code左侧会出现“远程资源管理器图标”**

Step 1: Install the Remote-SSH extension for VS Code. Once installed, you will see the "Remote Explorer" icon on the left side of VS Code.



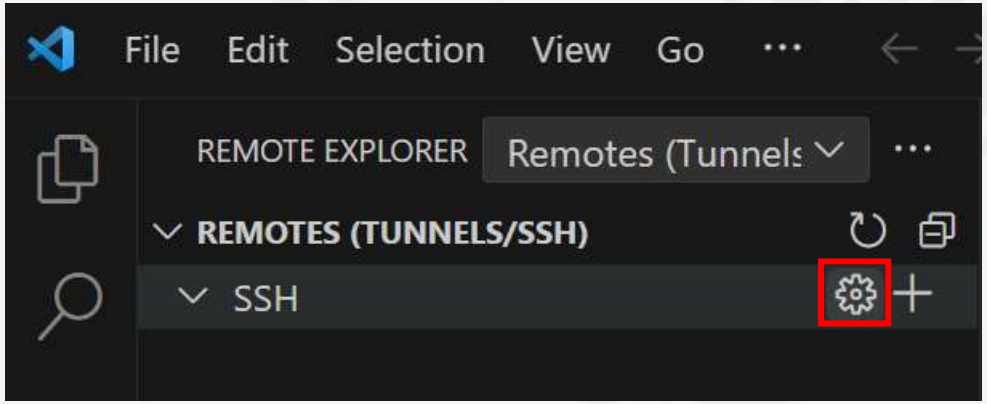
其他常用工具

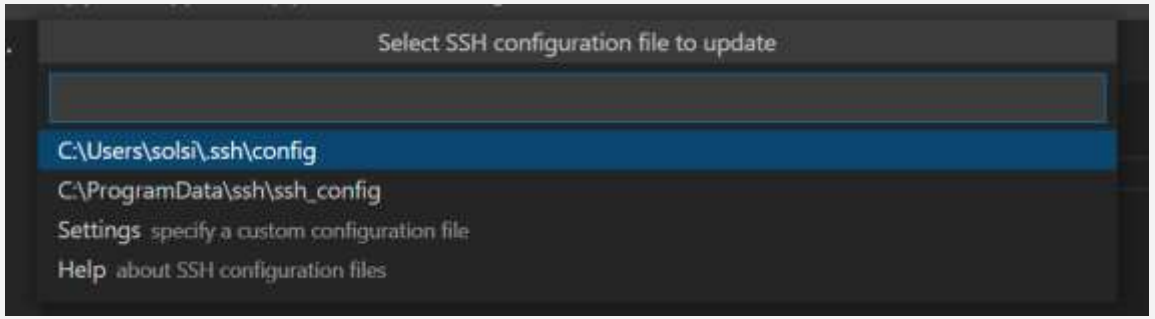
Other Commonly Used Tools

在本地VS Code上连接远程Linux进行开发（2）
Developing on Remote Linux by Connecting to VS Code Locally（2）

- **第二步：进入“远程资源管理器”，点击Configure按钮，在弹出框里选择一个路径作为配置文件路径（配置文件路径中支持中文）**

  Step 2: Enter the "Remote Explorer," click the Configure button, and in the popup box, choose a path as the configuration file path (Chinese characters are supported in the configuration file path).





其他常用工具

Other Commonly Used Tools

在本地VS Code上连接远程Linux进行开发（3）
Developing on Remote Linux by Connecting to VS Code Locally（3）

第三步：写入SSH连接信息，并保存配置文件

Step 3: Input SSH connection information and save the configuration file.

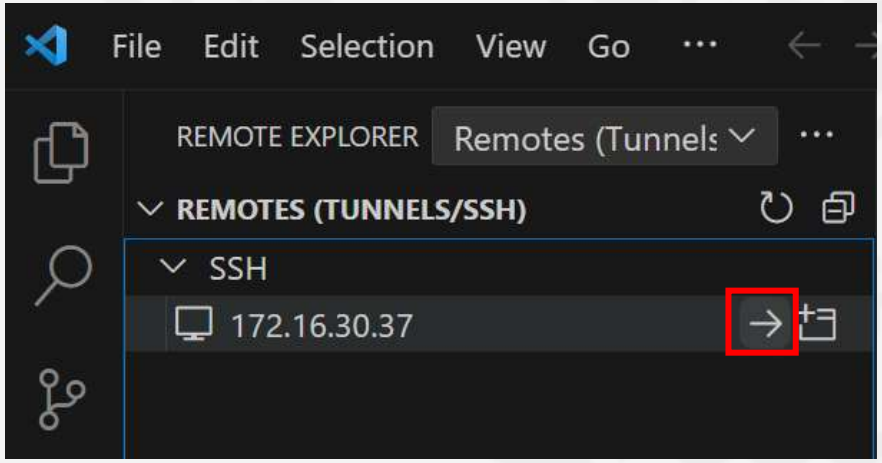
其他常用工具

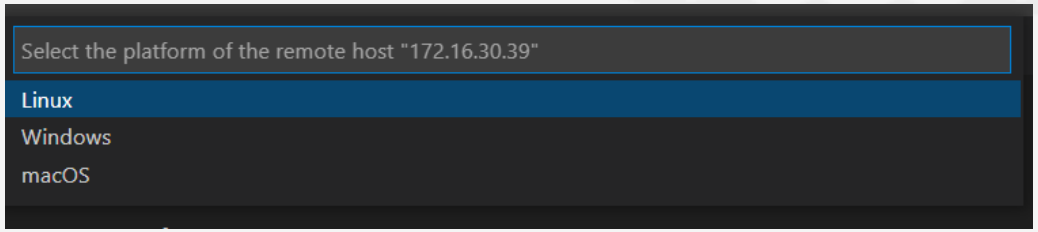
Other Commonly Used Tools

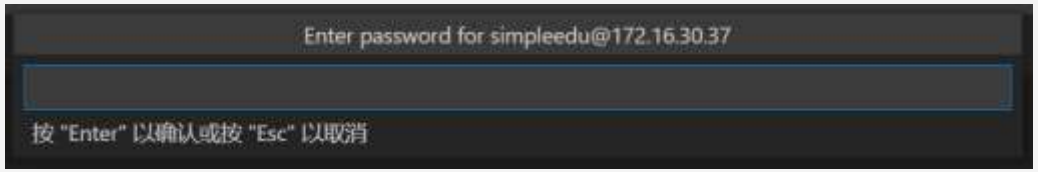
在本地VS Code上连接远程Linux进行开发（4）
Developing on Remote Linux by Connecting to VS Code Locally（4）

- **第四步：“远程资源管理器”页面会出现一个连接，点击connect按钮，会弹出一个新的VS Code窗口；第一次连接时会询问操作系统类型，选择Linux，然后选择continue； VS Code会远程安装一些必要的组件，这个过程可能会多次询问密码 。**

Step 4: A connection will appear on the "Remote Explorer" page. Click the "Connect" button to open a new VS Code window. Upon the first connection, it will prompt for the operating system type; choose Linux and then select "Continue." VS Code will remotely install necessary components, and during this process, it may prompt for the password multiple times.







其他常用工具

Other Commonly Used Tools

在本地VS Code上连接远程Linux进行开发（5）
Developing on Remote Linux by Connecting to VS Code Locally（5）

- **第五步：新弹出的VS Code窗口，已经实现了远程Linux系统的连接。在这个窗口中选择菜单“文件”->“打开文件夹”，此时列表中显示的即为远程Linux系统的用户根目录路径中的内容。**

Step 5: In the newly opened VS Code window, the connection to the remote Linux system has been established. In this window, select the menu "File" -> "Open Folder." The displayed list will now show the contents of the user's home directory on the remote Linux system.

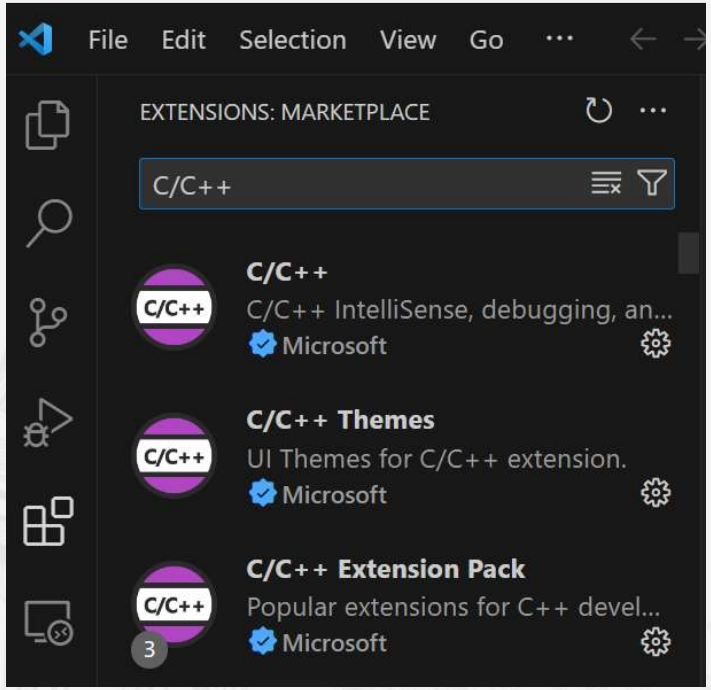
其他常用工具

Other Commonly Used Tools

使用VS Code开发C程序（1）
Developing C programs using VS Code（1）

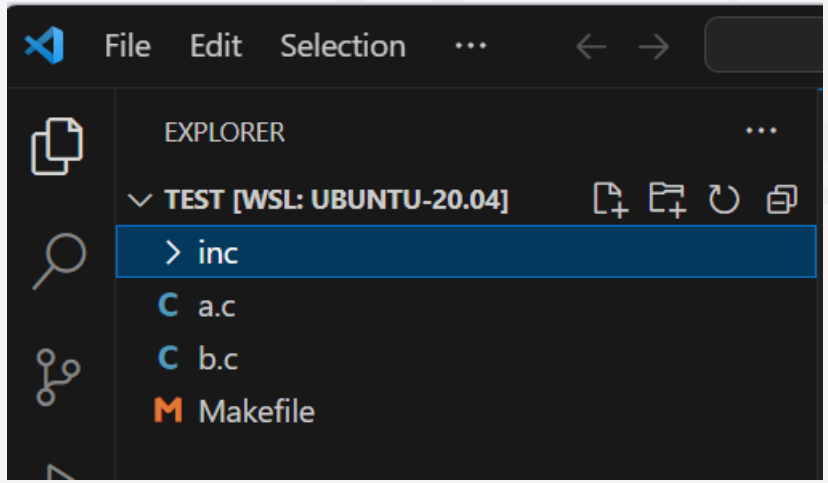
第一步：安装C/C++扩展

Step 1: Install the C/C++ Extension.



第二步：打开工程文件夹

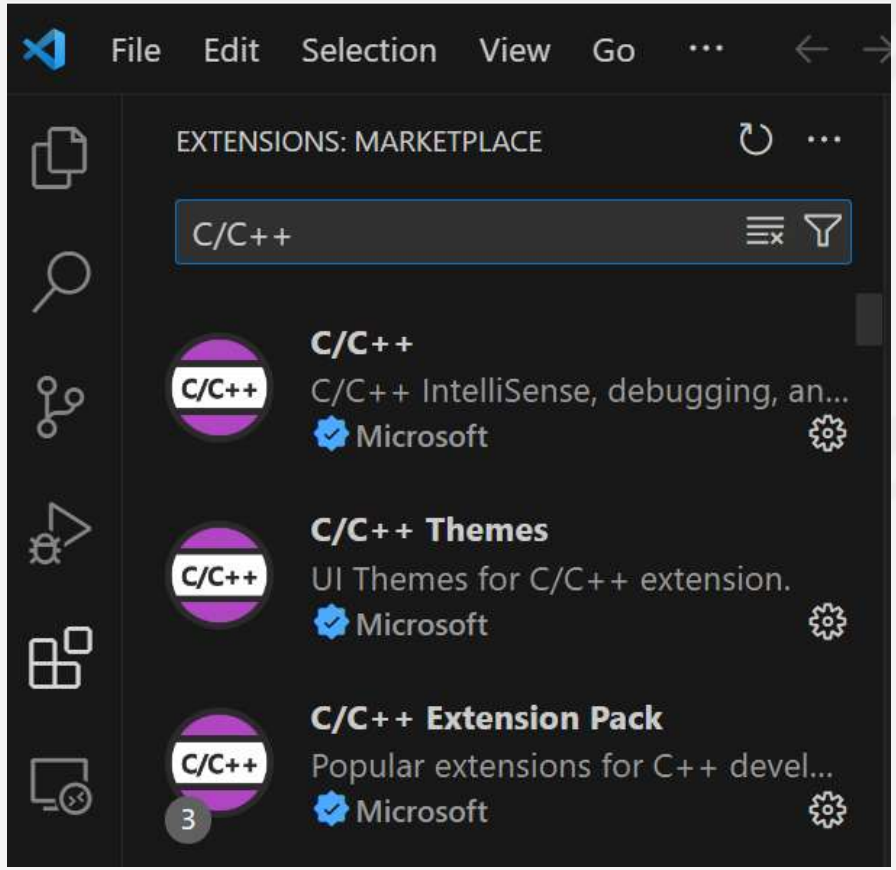
Step Two: Open the project folder.



使用VS Code开发C程序（2）
Developing C programs using VS Code（2）

第三步：在工程文件夹下创建子文件夹.vscode，在.vscode下创建launch.json和tasks.json配置文件

Step 3: Create a subfolder named ".vscode" within the project folder. Within the ".vscode" folder, create configuration files "launch.json" and "tasks.json."



```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "调试",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/p",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "build",
            "miDebuggerPath": "/usr/bin/gdb"
        }
    ]
}
```

launch.json

此处为调试的程序名称

调试前需要启动的任务

调试器的路径

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": "/usr/bin/make",
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

tasks.json

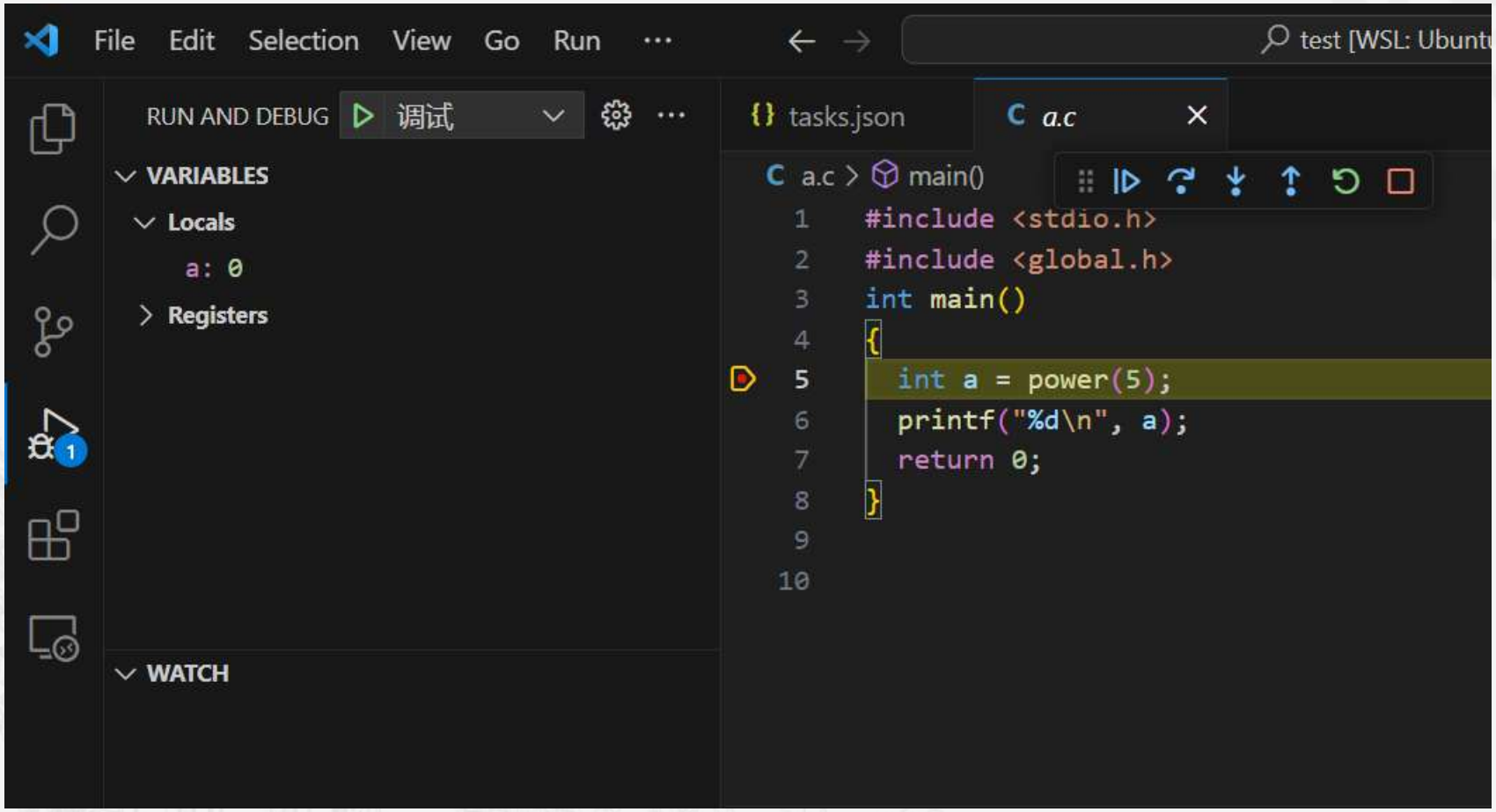
在调试前启动make进行编译

使用VS Code开发C程序（3）
Developing C programs using VS Code（3）

其他常用工具

Other Commonly Used Tools

使用VS Code开发C程序（4）
Developing C programs using VS Code（4）



执行至下一断点
Run to the next breakpoint.

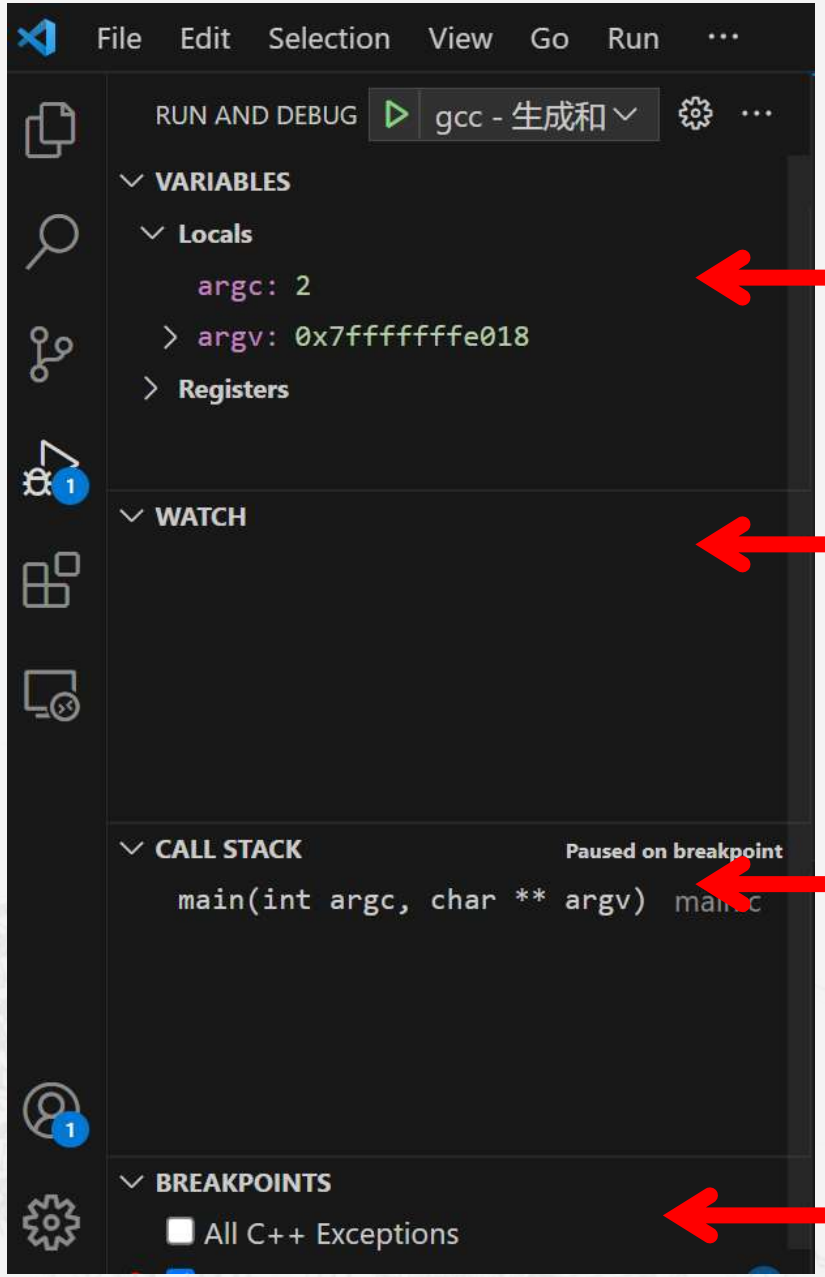
执行一条语句
Execute a single statement.

进入到函数中
Step into the function.

将函数执行完毕并跳出
Finish executing the function and return.

重新开始调试
Restart debugging.

结束调试
End debugging.



当前函数中变量的值
Values of variables in the current function

用户自定义观测的变量的值（通过右侧加号添加）
Values of user-defined observed variables (added via the plus icon on the right).

函数调用栈
Function call stack

断点信息（勾选中表示断点有效，调试时会暂停）
Breakpoint information (checked indicates an active breakpoint, causing a pause during debugging).

小结
Summary

- **工欲善其事，必先利其器**

To do a good job, one must first sharpen one's tools.

- **磨刀不误砍柴工**

It is important preparation and skill improvement before undertaking a task.

本章内容

Topic

- Linux操作系统

  Linux Operating System
- 文本编辑器 vi/vim

  Text Editor vi/vim
- GNU 工具链

  GNU Toolchain
- 其他常用工具

  Other Commonly Used Tools
- 代码版本管理

  Code Version Control

代码版本管理

Code Version Control

软件开发中经常遇到的问题
Frequently Encountered Issues in Software Development

- **如何管理代码的版本?**

How to Manage Code Versions

- **新代码的引入，软件出现bug，如何定位原因?**

When new code is introduced, leading to software bugs, how do you pinpoint the root cause?

- **如何查看代码变迁的历史过程?**

How can you view the historical progression of code changes?

- **项目开发中，多人如何协同进行工作?**

In project development, how do multiple individuals collaborate effectively?

代码版本管理

Code Version Control

Git 开源的分布式版本控制系统
Git, an open-source distributed version control system

- **可以有效、高速地处理从很小到非常大的项目版本管理。**

Capable of efficiently and swiftly handling version management for projects ranging from small to very large.

- **Linus Torvalds为了帮助管理Linux内核开发而开发的一个开放源码的版本控制软件。**

An open-source version control software developed by Linus Torvalds to aid in managing the development of the Linux kernel.

- URL：https://git-scm.com

```
$ sudo apt install git
```

配置Git的基本信息
Configure Git Basic Information

在Git第一次使用前必须设置用户姓名和邮箱，配置后方可使用

Before the first use of Git, it is necessary to set the user name and email. Git must be configured with these details before it can be utilized.

```
# 设置用户姓名为John Doe
$ git config --global user.name "John Doe"
# 设置邮箱为johndoe@example.com
$ git config --global user.email johndoe@example.com
# 设置git的默认编辑器为vim（推荐），不设置时默认为nano
$ git config --global core.editor vim
# 设置配色方案（可选）
$ git config --global color.ui true
```

关于Git的一些基本概念（1）
Some Basic Concepts About Git（1）

- **仓库：项目的版本控制库，包含项目的所有文件和历史记录。它可以存在本地计算机上，也可以托管在远程服务器上。**

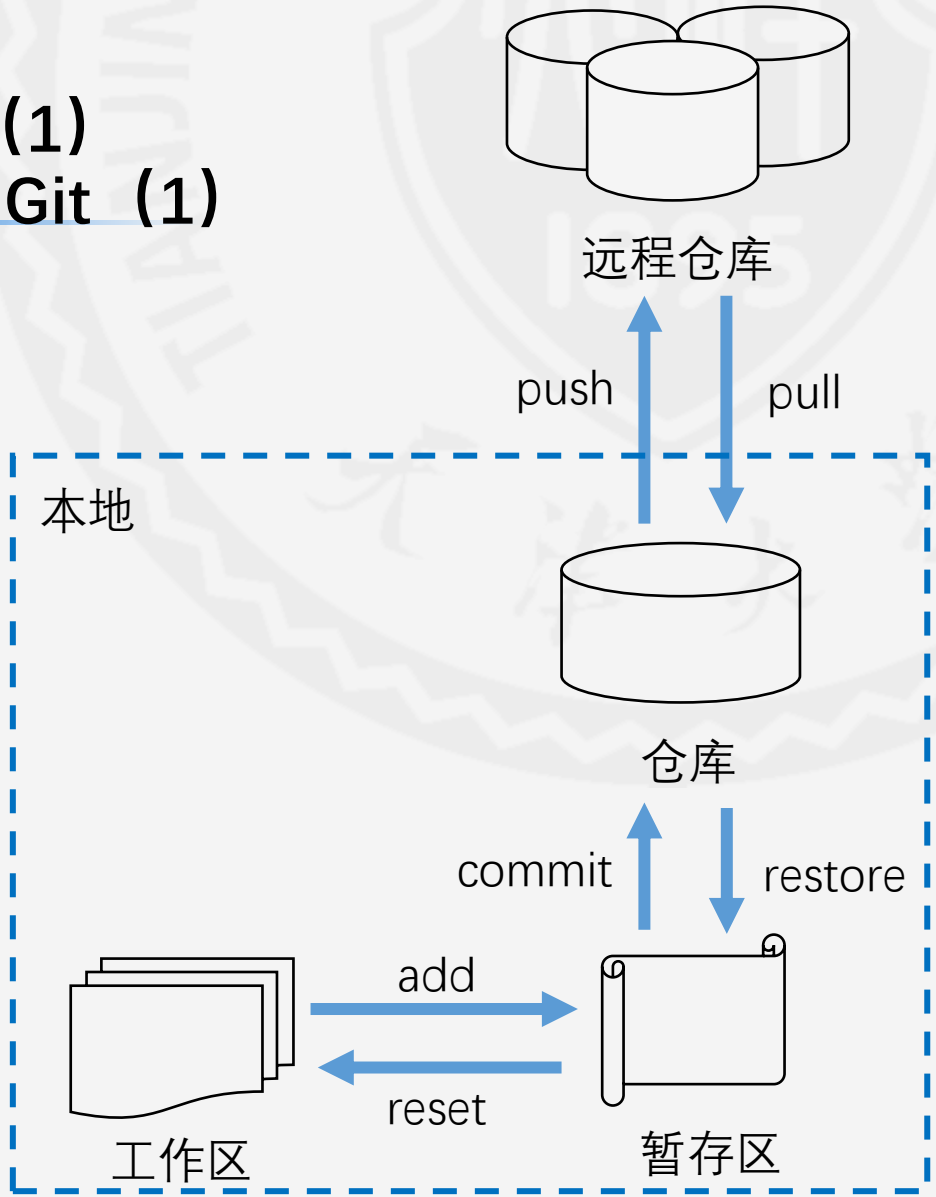
Repository: The version control library for a project, containing all files and their history. It can exist on a local machine or be hosted on a remote server.

- **提交：提交是对代码库的一次保存，包括对代码的修改、添加或删除。**

Commit: A commit represents a save in the codebase, including modifications, additions, or deletions.

- **还原：用本地仓库中的记录还原本地工作目录中文件的更改**

Restore: Restoring is the process of using the records in the local repository to revert changes made to files in the local working directory.



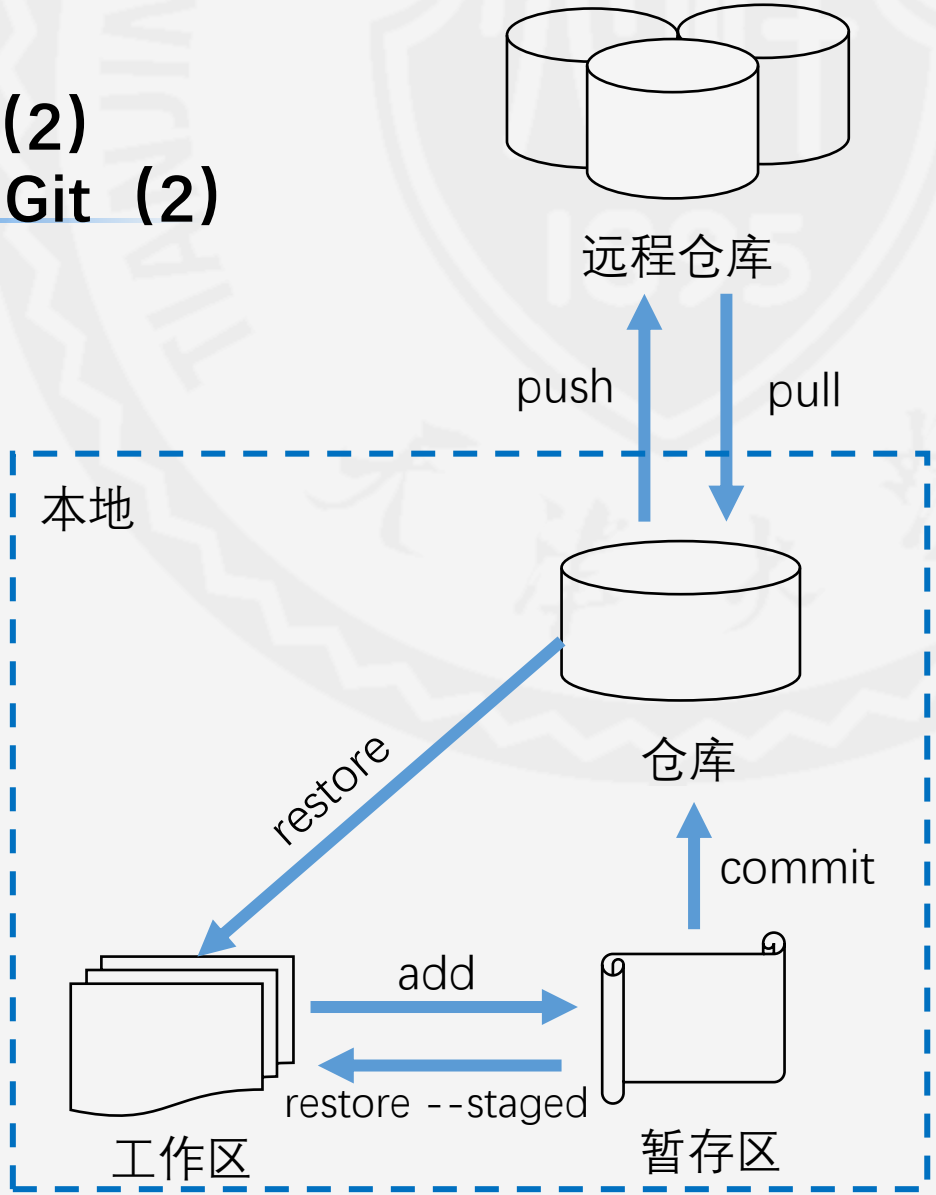
关于Git的一些基本概念（2）
Some Basic Concepts About Git（2）

- **推送：将本地仓库的更改上传到远程仓库的过程，使得远程仓库与本地仓库保持同步。**

Push: the process of uploading local changes to a remote repository, keeping the two repositories in sync.

- **拉取：从远程仓库获取最新更改并将其应用到本地仓库的过程。**

Pull: the process of fetching the latest changes from a remote repository and applying them to the local repository.



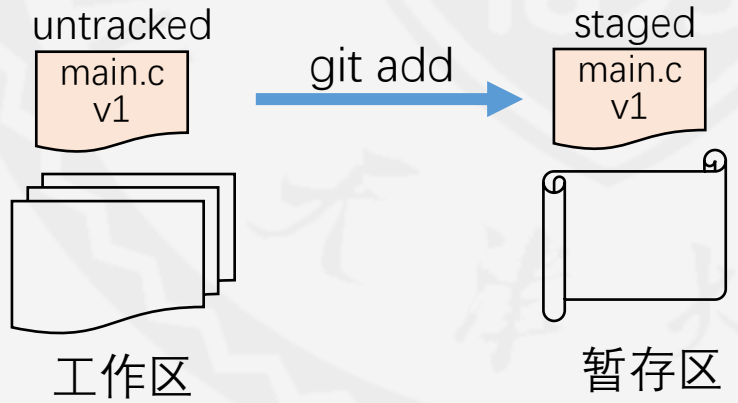
创建一个本地仓库
Create a Local Repository

```
# 创建本地工作目录
$ mkdir myproject

# 进入目录
$ cd myproject

# 在本地目录下创建一个空的本地仓库
# 命令执行完成后myproject下会出现一个隐藏目录.git
$ git init
```

.git 目录下存放着与本地仓库相关的全部记录，删除本地工作目录下的其它任意文件，都可以通过仓库进行回复，删除.git则不能恢复

向仓库提交第一个文件（1）
Committing the first file to the repository（1）

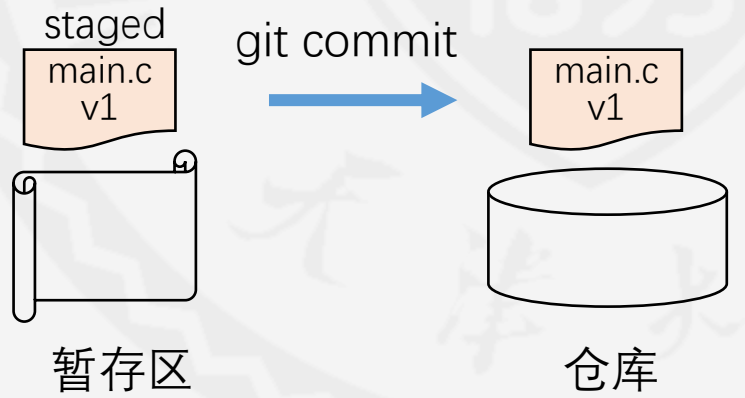
```
# 创建main.c文件，并编辑其内容
$ vi main.c
# 将main.c转换为staged状态
$ git add main.c

# 查看当前状态
$  git status
On branch master
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   main.c
```

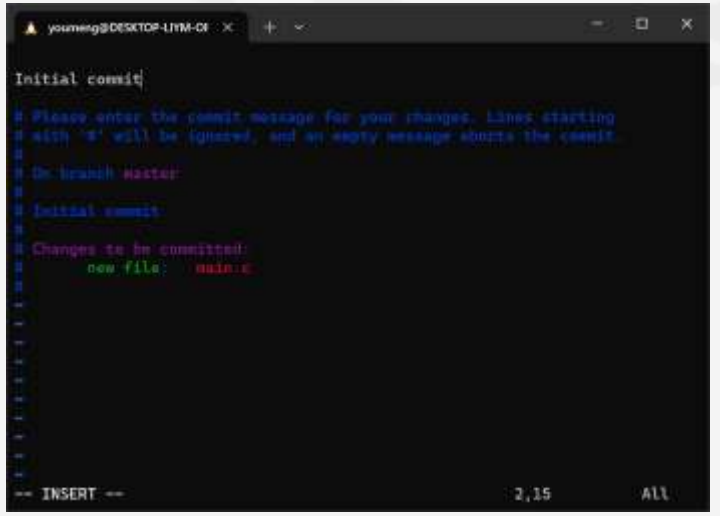



```
#include <stdio.h>
int main()
{
  printf("Hello World!\n");
  return 0;
}
```
main.c

向仓库提交第一个文件（2）
Committing the first file to the repository（2）

```
# 将main.c提交至版本库
# 命令运行后会出现vi编辑器界面，用于编辑提交说明
$ git commit main.c
[master (root-commit) 23eae96] Initial commit
 1 file changed, 7 insertions(+)
 create mode 100644 main.c
# 查看当前状态
$ git status
On branch master
nothing to commit, working tree clean
```

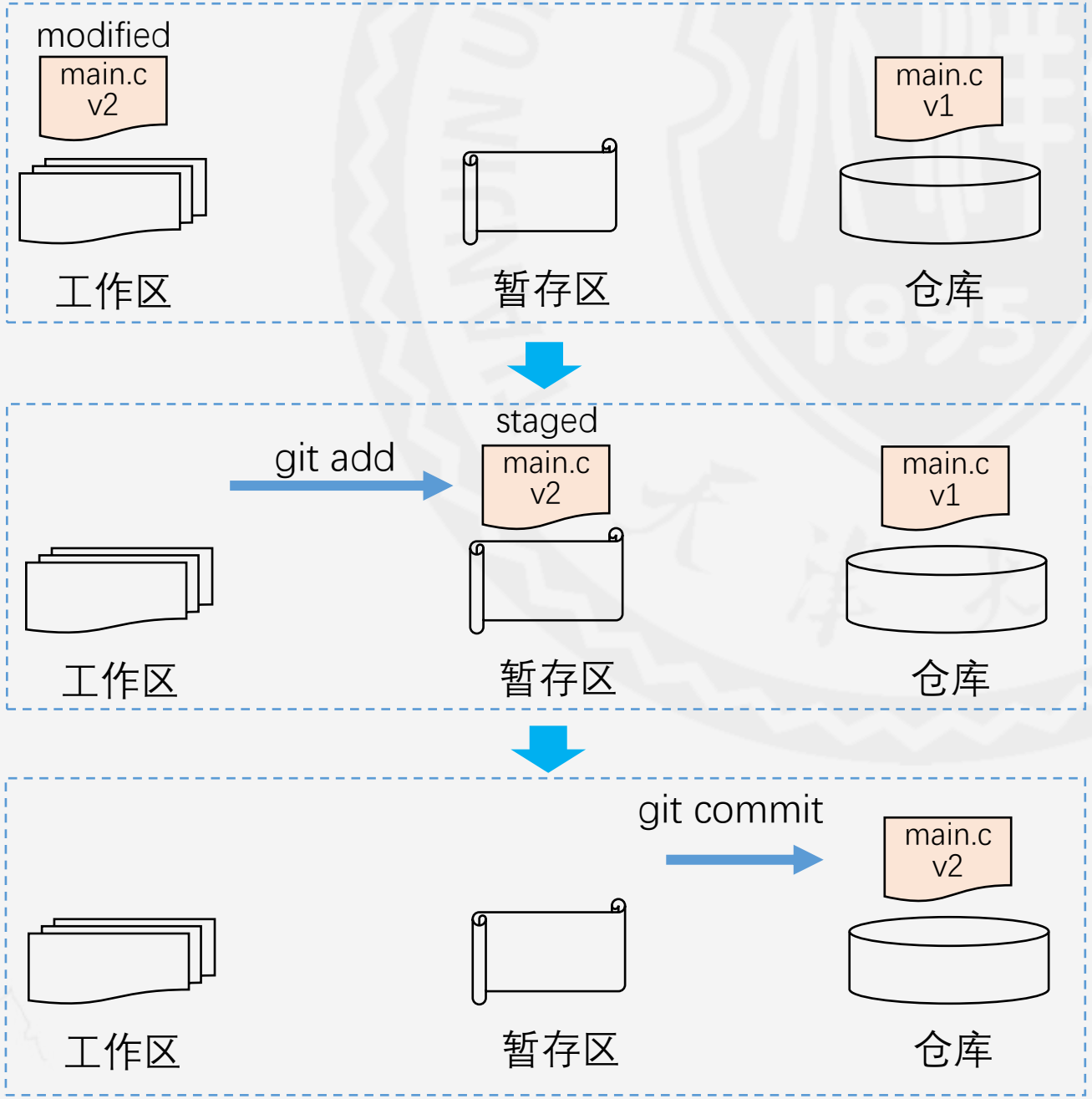




代码版本管理

Code Version Control

修改main.c
Modifying main.c



修改暂存区中的文件
Modifying a file in the staging area

- **修改暂存区的文件，会将文件从暂存区移出**

Modifying a file in the staging area will remove it from the staging area.

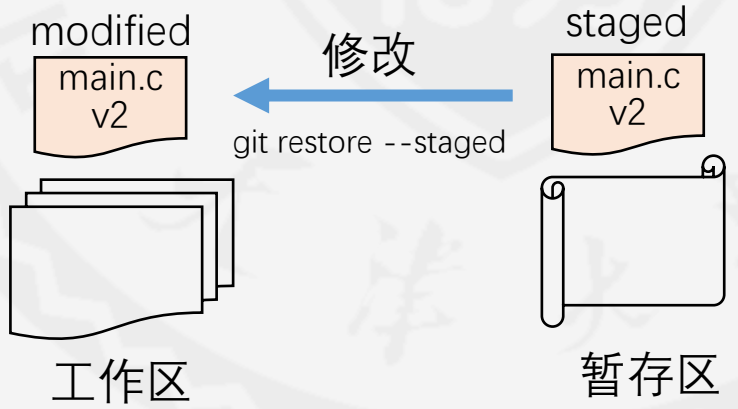
- **提交时需要重新使用git add命令，将文件放入暂存区**

To commit, you need to use the 'git add' command again to place the file back into the staging area.

- **使用git restore –staged命令也可以将暂存区文件移出**

Using the 'git restore --staged' command can also remove a file from the staging area.

```
$ git restore --staged main.c
```



暂存区中的文件内容，没有被仓库记录，修改后无法还原
The content of a file in the staging area is not recorded in the repository. Once modified, it cannot be restored to its previous state unless the changes are committed.

查看提交日志
Viewing Commit Logs

使用git log命令查看提交日志

Using the ‘git log’ command to view commit logs.

```
$ git log                                                    提交记录
commit b74ba1613063af15d91e3568bf5d79ec282fe9bb (HEAD -> master)
Author: John Doe <johndoe@example.com>
Date:   Mon Feb 26 11:07:25 2024 +0800

    Second commit

commit 23eae96be15306a1e3270b701f217315d8c7e144
Author: John Doe <johndoe@example.com>
Date:   Mon Feb 26 09:40:15 2024 +0800

    Initial commit
```

master分支的最近一次提交

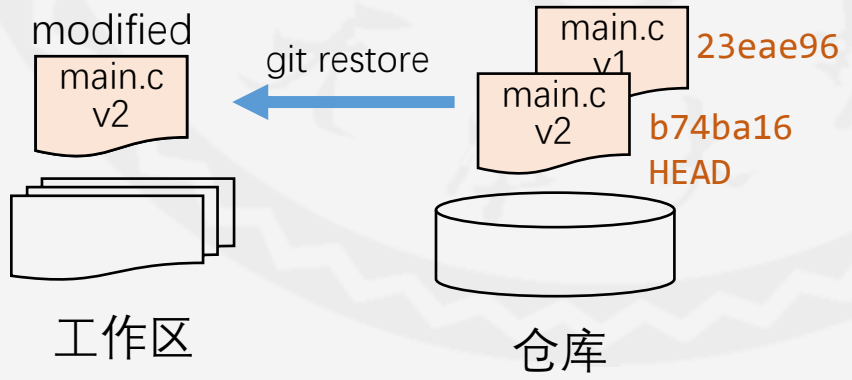
提交记录在仓库中的唯一标识

代码还原（1）
Code Restoration（1）

使用git restore命令将仓库中最近一次提交记录覆盖至工作区

Using the ‘git restore’ command to overwrite the working directory with the latest commit from the repository.

```
$ git restore main.c
```



restore 命令自v2.23版本后开始支持，早期版本使用checkout命令实现
The restore command has been supported since version 2.23; in earlier versions, the functionality was achieved using the checkout command.

代码还原（2）
Code Restoration（2）

- **格式：git restore --source=提交记录标识 文件名**

Format：git restore --source=commit_id file_name

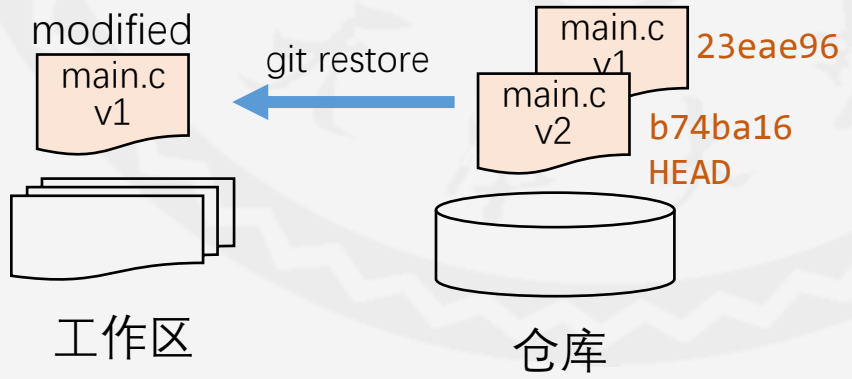
- **将仓库中指定的提交记录覆盖至工作区**

Overwriting the working directory with a specified commit from the repository.

只需要提交记录的前几位即可

```
$ git restore --source=23eae96 main.c
```

```
# 将整个项目的文件还原至某一次提交（包括工作目录和暂存区）
$ git restore --source=23eae96 --worktree --staged .
```



代码版本管理

Code Version Control

创建一个空的远程仓库：以GITEE为例
Creating an Empty Remote Repository: Taking GITEE as an Example

空仓库

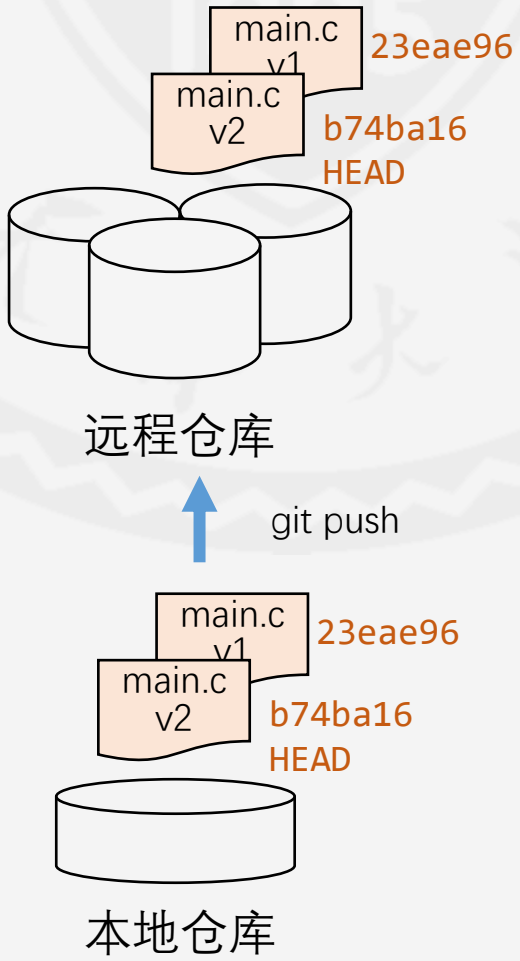
代码版本管理

Code Version Control

从第二次推送开始，只需执行git push即可（不需要参数）
Starting from the second push, simply execute 'git push' without additional parameters.

关联远程仓库并推送代码
Associating with a Remote Repository and Pushing Code

```
# 在仓库工作路径下执行下面的命令，关联远程仓库
$ git remote add origin https://gitee.com/tjuics/ics_demo.git

# 第一次推送代码
$ git push -u origin master
Username for 'https://gitee.com': 输入用户名
Password for 'https://liyoumeng@tju.edu.cn@gitee.com': 输入密码
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 576 bytes | 288.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
remote: Powered by GITEE.COM [GNK-6.4]
To https://gitee.com/tjuics/ics_demo.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```



再次查看提交日志
Viewing Commit Logs Again

```
$ git log
commit b74ba1613063af15d91e3568bf5d79ec282fe9bb (HEAD -> master, origin/master)
Author: John Doe <johndoe@example.com>
Date:   Mon Feb 26 11:07:25 2024 +0800

    Second commit

commit 23eae96be15306a1e3270b701f217315d8c7e144
Author: John Doe <johndoe@example.com>
Date:   Mon Feb 26 09:40:15 2024 +0800

    Initial commit
```

最近一次同步时远程仓库的位置

另一种创建仓库的方法：复制开源的仓库
Another Way to Create a Repository: Copying From an Open-source Repository

- 在实践中，另一种常用的创建仓库的方法是复制现有的开源仓库

In practice, another commonly used method to create a repository is by duplicating an existing open-source repository.

- 应用场景：

Application scenarios:

  - 获取开源项目代码并进行二次开发

  Fetching the code of an open-source project and engaging in secondary development.

  - 本课程实验代码的分发

  Distribution of the experimental code for this course.

代码版本管理

Code Version Control

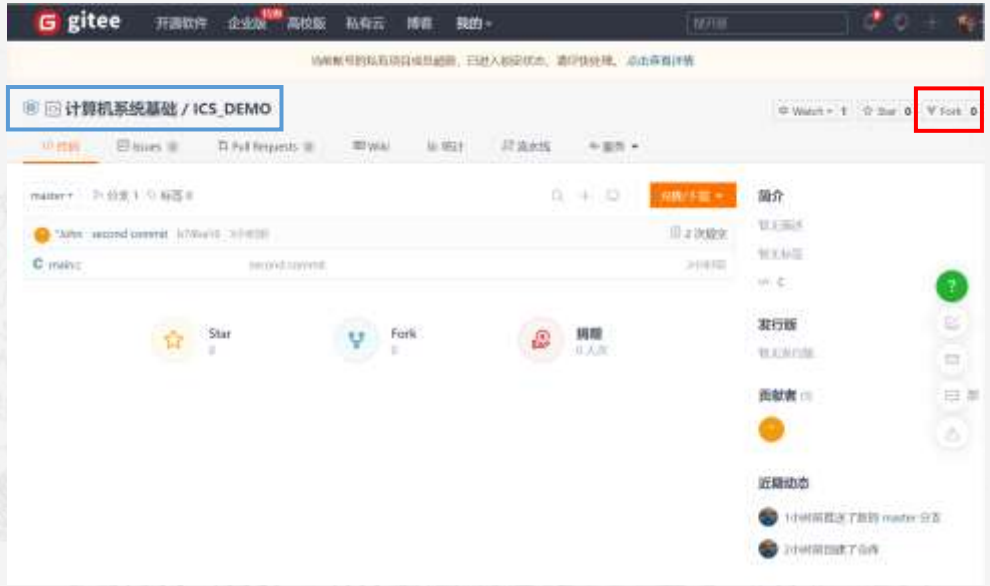
复制一个远程仓库
Forking a Remote Repository

- 根据URL访问开源仓库的页面，点击Fork

Accessing the webpage of an open-source repository using its URL, then click on "Fork".

- 将第三方仓库复制为我的仓库

Forking the third-party repository to create a copy in my own repository.



将远程仓库克隆至本地
Clone a Remote Repository to the Local Machine

- **可以使用git clone命令，实现将远程仓库复制到本地**

You can use the git clone command to copy a remote repository to your local machine.

```
$ git clone https://gitee.com/tjuics/lab1.git    # 实验一的代码
```

- **本地仓库与远程仓库之间已存在同步关系，本地仓库的推送和拉取操作都是基于这个远程仓库进行的**

The local repository is already synchronized with the remote repository. Both push and pull operations in the local repository are based on this remote repository..

多人协作开发（1）
Collaborative Development Among Multiple Individuals（1）

- **两个开发者如何同时为一个版本库贡献代码呢?**

How can two developers contribute code to a repository simultaneously?

- **场景**

Scenario:

  - **开发者A创建了本地仓库和远程仓库，并将远程仓库地址分享给开发者B**

  Developer A creates a local repository and a remote repository, then shares the remote repository's address with Developer B.

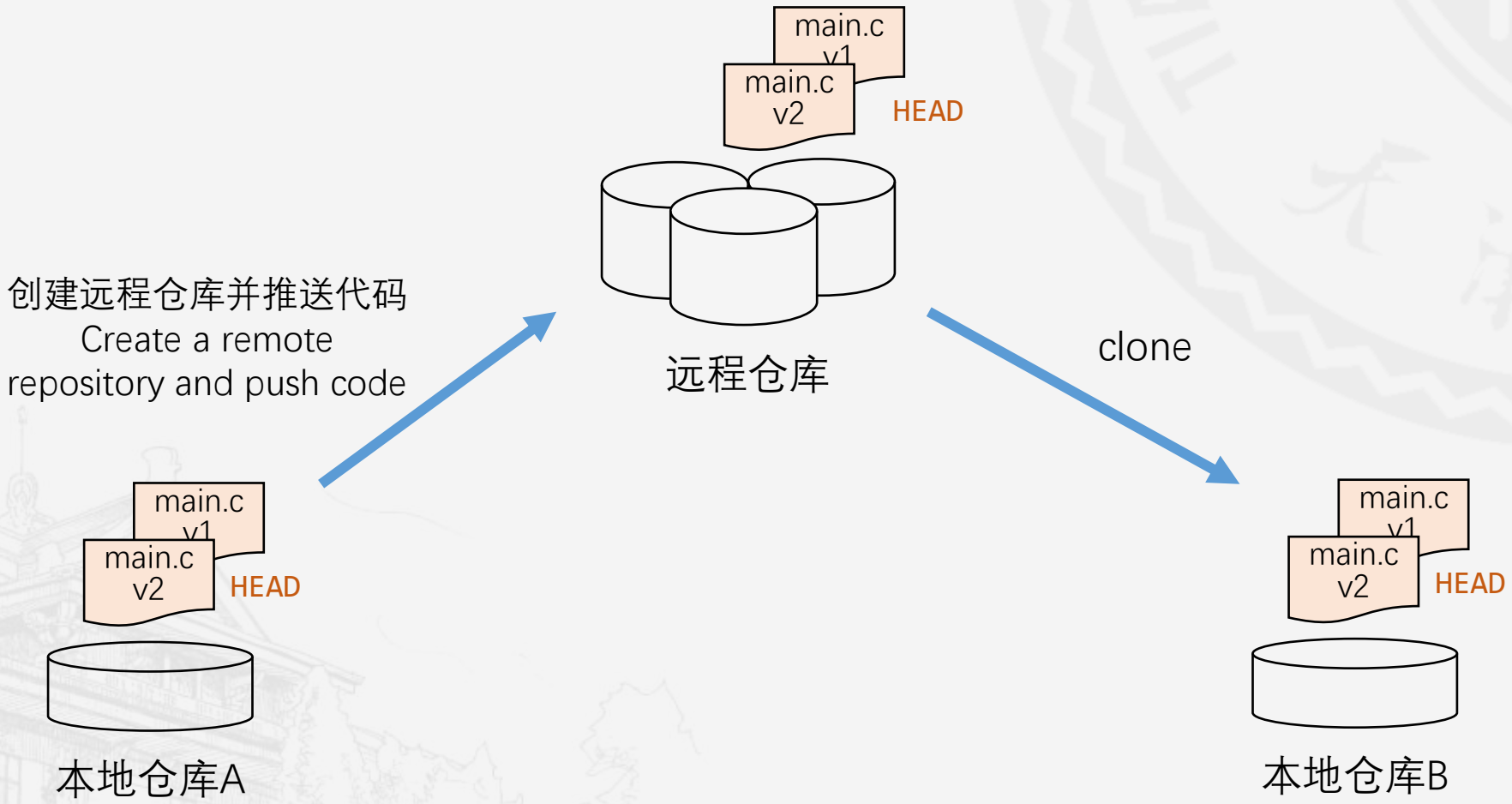
  - **开发者B克隆仓库至本地**

  Developer B clones the repository locally.

  - **开发者A和开发者B同时向这个远程仓库提交代码**

  Both Developer A and Developer B simultaneously commit code to this remote repository.

多人协作开发（2）
Collaborative Development Among Multiple Individuals（2）



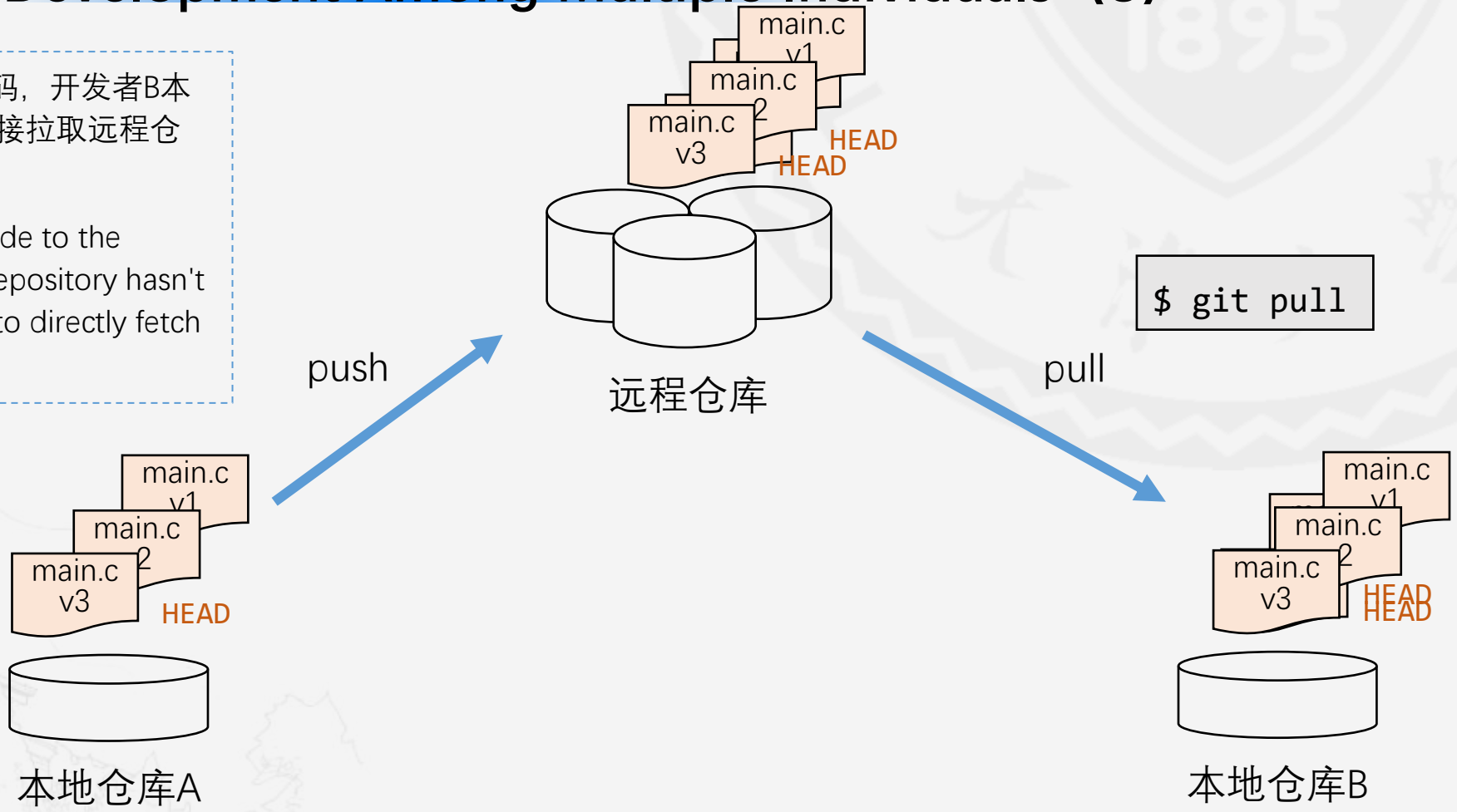
代码版本管理

Code Version Control

多人协作开发（3）
Collaborative Development Among Multiple Individuals（3）

- 开发者A向远程仓库push了新版本的代码，开发者B本地仓库未发射变化，可以通过git pull直接拉取远程仓库中的更新

Developer A pushed a new version of the code to the remote repository, and Developer B's local repository hasn't reflected the changes. They can use git pull to directly fetch updates from the remote repository.



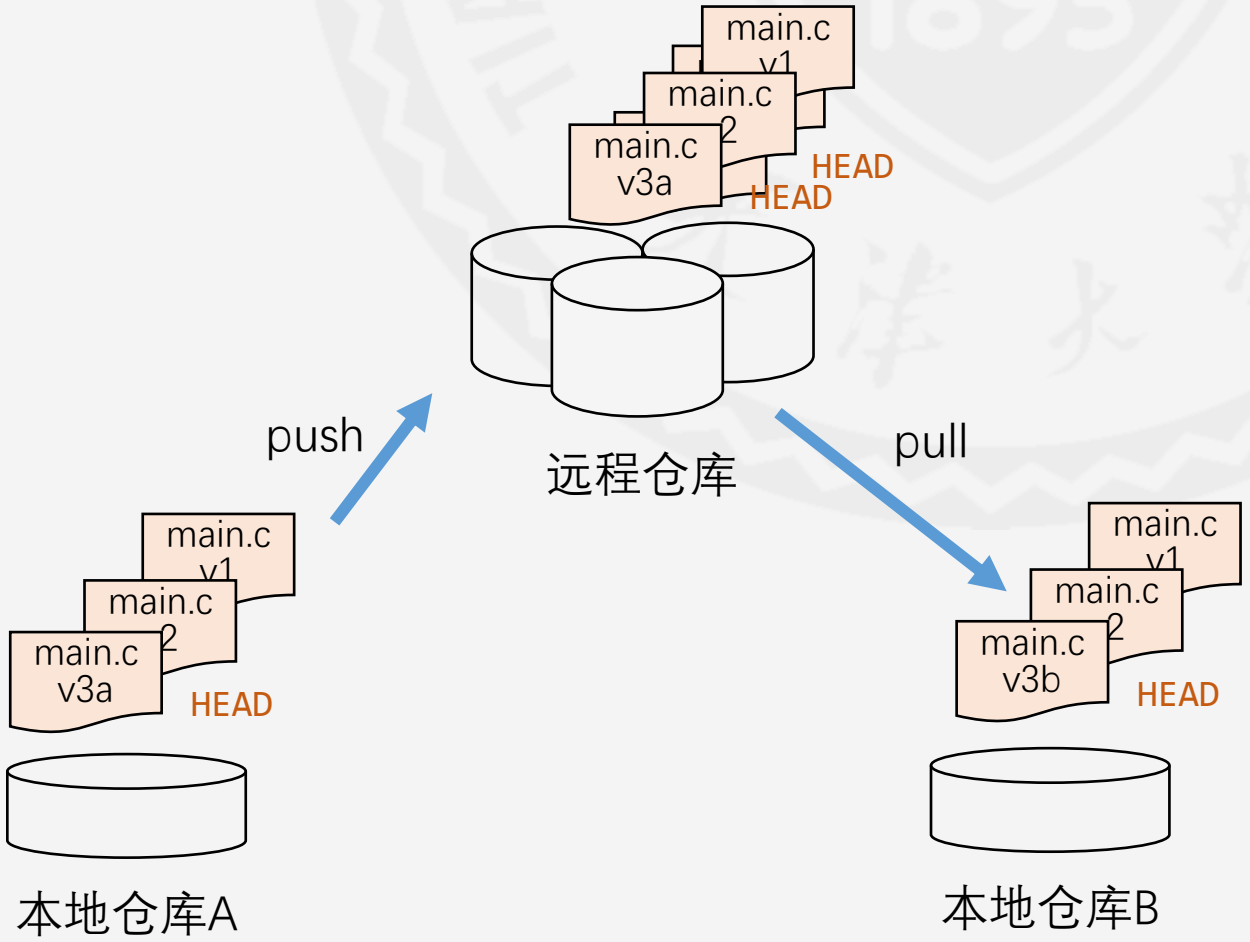
多人协作开发（4）
Collaborative Development Among Multiple Individuals（4）

- 开发者A和开发者B的本地仓库同时出现变化。开发者A先于开发者B将本地仓库的变化推送至远程仓库

Both Developer A and Developer B's local repositories have changes. Developer A, being ahead, pushed the changes to the remote repository before Developer B.

- B在推送代码前需要先拉取远程仓库中的最新代码

Before pushing code, Developer B should first pull the latest code from the remote repository.



多人协作开发（5）
Collaborative Development Among Multiple Individuals（5）

- 由于A和B修改了相同的文件，pull操作后会产生冲突

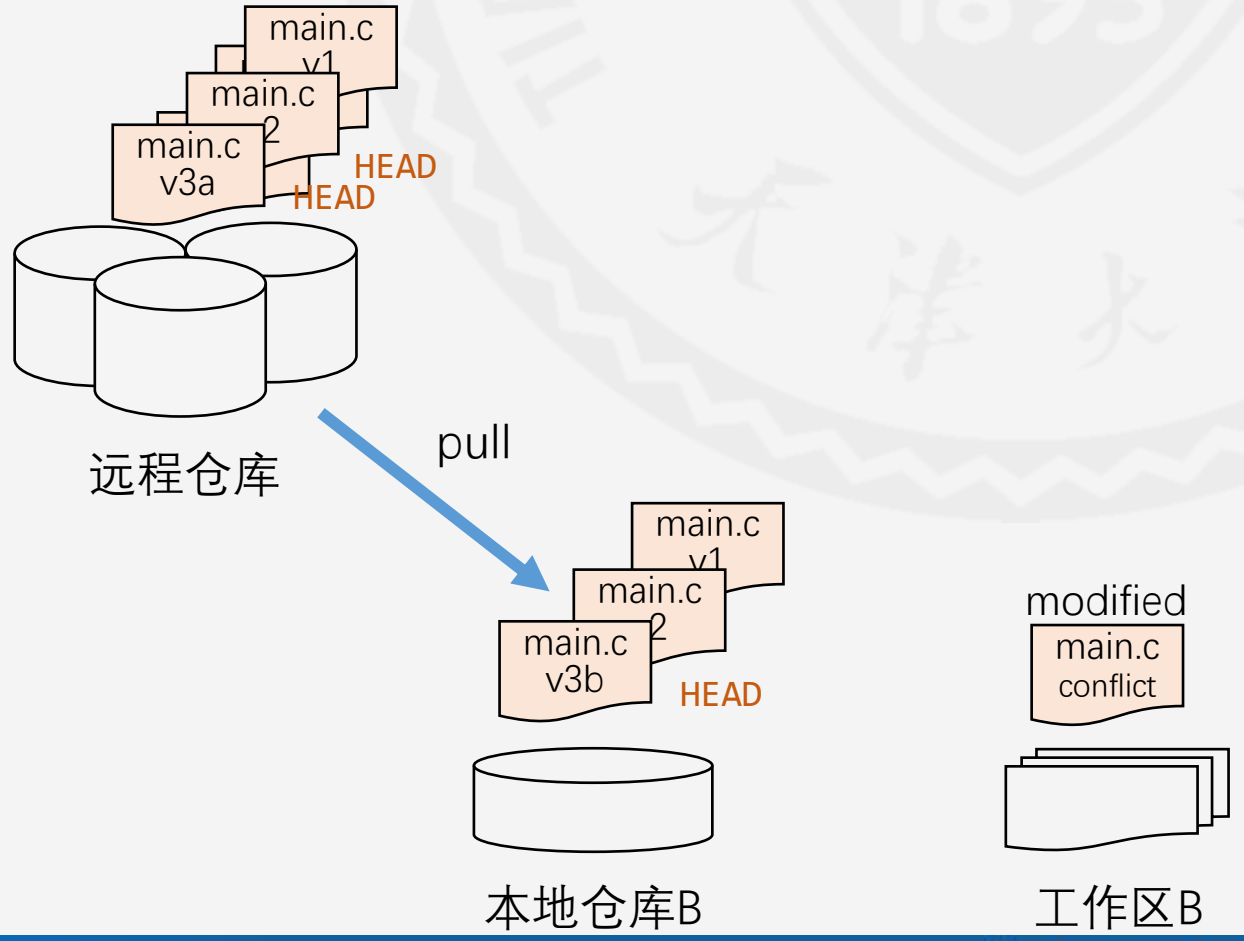
Due to changes made by both A and B in the same file, a pull operation will result in conflicts.

- 多数冲突可以由git自动解决

Most conflicts can be automatically resolved by Git.

- 当A和B的修改位置相同时，冲突无法自动解决，需要人工干预。此时会在B的工作区生成一个修改文件，在文件中对冲突进行了标记

However, when A and B's modifications conflict at the same location, automatic resolution is not possible, requiring manual intervention. In such cases, Git will create a modified file in B's working directory, marking the conflicts for manual resolution.



多人协作开发（6）
Collaborative Development Among Multiple Individuals（6）

```
#include <stdio.h>
int main()
{
  printf("aaa\n");
  printf("Hello World!\n");
  return 0;
}
```
main.c/v3a

```
#include <stdio.h>
int main()
{
  printf("bbb\n");
  printf("Hello World!\n");
  return 0;
}
```
main.c/v3b

本地仓库的近期修改

远程仓库的近期修改

```
#include <stdio.h>
int main()
{                                          代码冲突区域
<<<<<<< HEAD
  printf("bbb\n");
=======
  printf("aaa\n");
>>>>>>> 938d9e1c1549b13cd919ec67133f4120bddcdb99
  printf("Hello World!\n");
  return 0;
}
```
main.c/modified

在开发者B的工作区中

多人协作开发（7）
Collaborative Development Among Multiple Individuals（7）

- 开发者B需要编辑冲突区域的代码，以解决冲突

Developer B needs to edit the code in the conflicted areas to resolve the conflict.

- 编译通过，并测试无误后，将修改后的代码提交至本地仓库

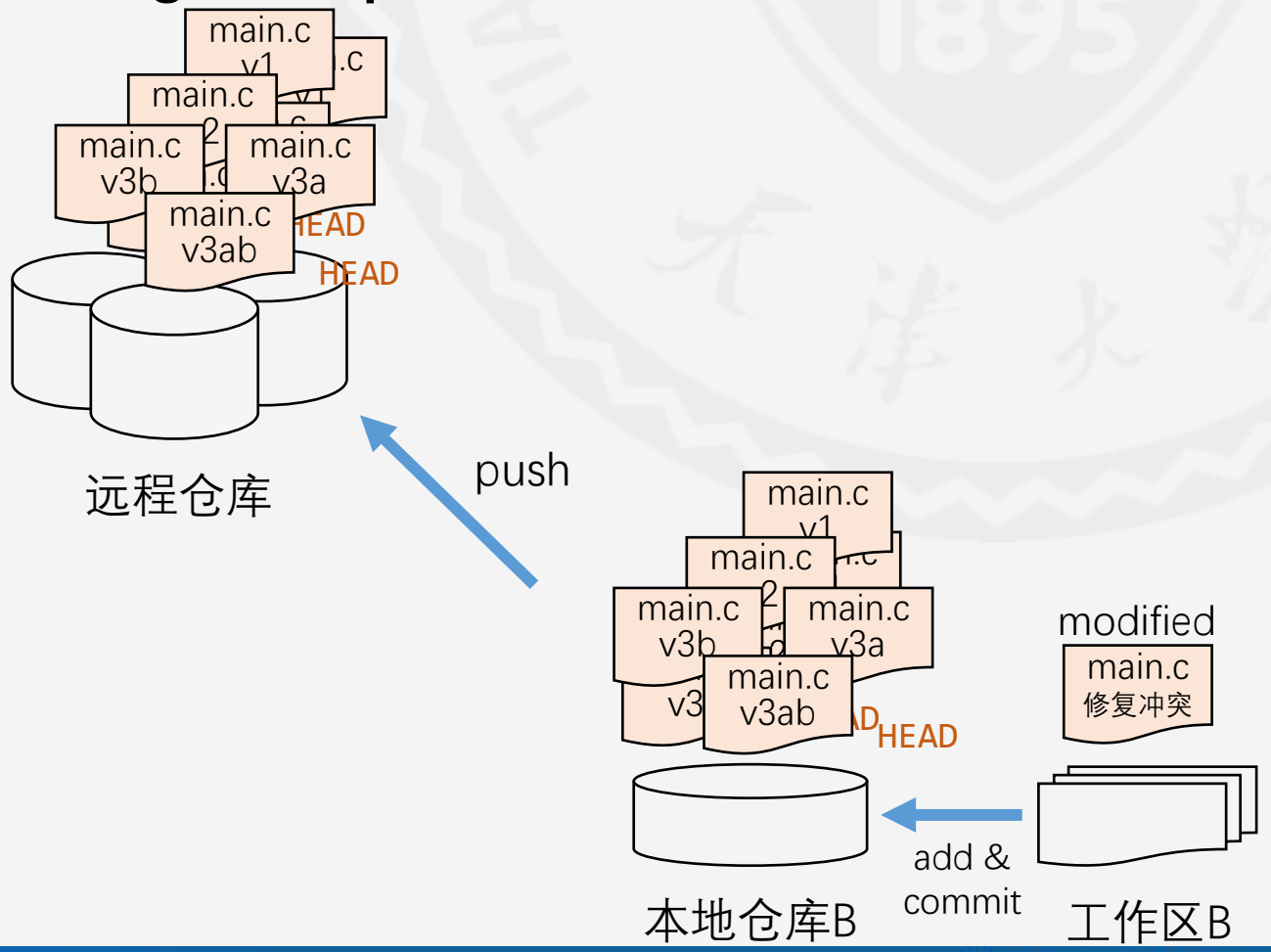
After successfully compiling and testing without errors, commit the modified code to the local repository.

- 冲突解决的提交记录会由git自动生成，开发者可以进行二次编辑

The commit record for conflict resolution is automatically generated by Git, and developers have the option to perform additional edits if needed.

- 最后，使用git push将代码推送至远程仓库

Finally, use git push to push the code to the remote repository.



多人协作开发（8）
Collaborative Development Among Multiple Individuals（8）

- 在多人开发场景下，推送代码至远程仓库时正确的操作习惯：

In a multi-developer scenario, the correct operational habits when pushing code to a remote repository are as follows:

  - 应该首先尝试从远程仓库拉取代码

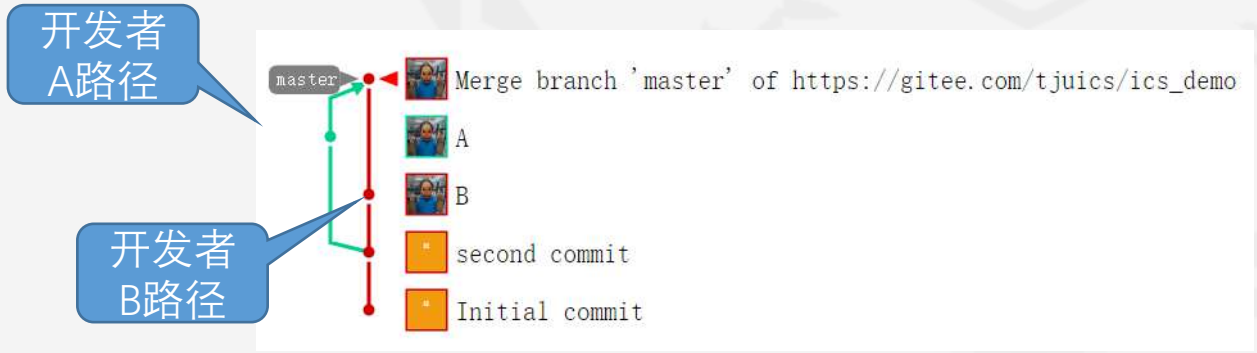
    First, attempt to fetch the latest code from the remote repository

  - 如有冲突需要解决，在本地解决冲突

    Resolve conflicts locally if they arise

  - 推送代码至远程仓库

    Push the code to the remote repository.



远程仓库的“仓库网络图”

练习
Exercises

- **从gitee上复制实验一的远程仓库，并克隆至本地**

Fork the remote repository for Experiment 1 from Gitee and clone it to local machine.

- **URL：https://gitee.com/tjuics/lab1.git**

总结
Summary

- GIT有很多强大的功能，受时间所限，课程中只介绍了最基本的用法

GIT has many powerful features. Due to time constraints, the course only covered the most basic usage.

- 更复杂的开发场景下，需要更强大的GIT功能的支撑

In more complex development scenarios, the support of more powerful GIT features is necessary.

- 更多的功能需要在实践中探索，如：

More functionalities need to be explored in practice, such as:

  - 分支

  Branch

  - 变基合并

  Rebase merging

  - 多远程仓库间的pull-request

  Pull requests between multiple remote repositories