



北京大学计算机基础科学与开发手册

作者：臧炫懿

组织：北京大学信息科学技术学院、北京大学学生 Linux 俱乐部




“把初高中失去的计算机基础知识和大学失去的开发能力补回来”

前言

”把丢失的初高中计算机基础知识和大学丢失的开发能力补回来”

计算机基础科学教育是我国近些年一直努力推进的教育之一，北京大学的所有学生都应修习《计算概论》课程。然而，大学计算机基础教育的内容往往过于理论化，缺乏实用性，这使得同学们在学习过程中容易感到枯燥乏味，在学习之后也很难将所学知识灵活应用到实际生产与生活中。所以最终就导致了一个问题：同学们在学习完《计算概论》之后，仍然对计算机的使用和实际的开发工作感到极为陌生。同时，我国初高中乃至小学阶段，计算机的教育水平参差不齐，同学们的基础也不尽相同，这导致部分基础较差的同学在学习《计算概论》时会遇到困难，遑论进阶课程。

在本手册正式编写之前，已经有很多学长为了抹平基础知识的差距做出了相关的努力。几个广为人知的项目：北京大学为新生提供了《计算概论衔接课》，旨在帮助同学们快速入门计算机基础知识（这门课的前半部分由我所讲授）；北京大学学生 Linux 俱乐部（LCPU）启动了 Getting Started 项目，旨在帮助同学们快速入门 Linux 和计算机科学（该项目亦由我全权负责）；一位  学长发起了 CS 自学指南项目，受到了广泛的关注和认可，该项目迄今已有一百五十余位贡献者；PKUHub 等其他官方或非官方的组织也在积极推动计算机基础教育的普及。

然而，Getting Started 和自学指南对于大一新生而言，存在的最大不足之处就是：不够基础。而对于有能力就读北大的学生而言，大约是已经不再幻想上课有用了，真正有用的知识还是要靠自己去学习实践。因此，我认为给同学们一本手册要比给同学们数小时的课程视频有用得多。笔者最终决定：制作这份手册，帮助同学们把初高中缺失的计算机知识，以及大学丢失的开发能力补回来。

比起“CPU 是怎么构成的”“软件是怎么工作的”这种理论知识，本手册更侧重于“我们应该购买什么 CPU”“我们应该怎么搭建一个软件开发平台”这类的实用知识。简而言之，我们手册中会讲一些正课几乎不会讲、但是用处极大的知识。

本手册的目标是将同学们的计算机水平快速提升至能够接受大学计算机基础教育的水平。我们认为使用本手册的同学都已经具备了最基本的计算机操作能力，也就是说我们不会涉及诸如“怎样使用鼠标”“怎样关机”这种内容。

本手册以本人实践和经验为基底讲授。如果本手册中的内容和正课中的内容或要求有差异，请以正课为准。

本手册中有大量内容的文字风格和其他地方不同，例如本段。这样排版的内容属于阅读材料（通常是为了解释说明上文或者下文可能涉及到的一些更艰深的理论知识，或者一些作者本人觉得很有意思的小知识），较为晦涩，读者可以选择性阅读——弄不明白也没关系，不影响使用，只需要知道有这么回事就可以了！

本手册参考了 LCPU Getting Started 以及诸多博文、指南的内容，并在此基础上进行了增删和修改。

希望本手册能够对同学们有所帮助。

如有疑问，欢迎向手册作者发送电子邮件反馈：zangxuanyi@stu.pku.edu.cn。

也欢迎来我的 GitHub 主页（ ZangXuanyi/getting-started-handout）查看本手册的源代码，并提出 Issue 与 Pull Request。所有的贡献者都会被列在最后的致谢名单中。

目录

前言	i
第一部分 零基础起步	9
第一章 初步认识计算机	10
1.1 计算机的鼻祖	10
1.2 现代计算机的组成	10
1.2.1 计算机的硬件	10
1.2.2 计算机的软件	13
1.2.3 实用软件推荐	15
1.2.4 怎样卸载软件	16
1.3 计算机间的通讯	16
1.3.1 网络基本名词及其解释	17
1.3.2 校内网络配置指南	18
1.4 网安相关知识	21
1.4.1 网络的风险	21
1.4.2 从根源断绝问题	21
1.4.3 亡羊补牢，为时未晚	22
第二章 搜索和信息获取	24
2.1 搜索	24
2.1.1 搜索引擎的选择	24
2.1.2 搜索技巧	24
2.2 信息平台	25
2.2.1 官方文档、Wiki、论坛	25
2.2.2 Stack Overflow	26
2.2.3 GitHub	26
2.2.4 Wikipedia	26
2.2.5 其他著名博客和教程	26
2.2.6 国内优质平台	27
2.3 LLM、VLM 和提示词工程	27
2.3.1 选择 LLM	27
2.3.2 使用 LLM 的基本原则	28
2.3.3 LLM 的局限性	28

2.3.4 提示词工程简介	29
2.3.5 在 Cherry Studio 上使用 LLM	31
2.4 提问的艺术	31
第三章 初步使用计算机	33
3.1 维护你的系统	33
3.1.1 保持更新系统的习惯	33
3.1.2 定期备份数据	33
3.1.3 定期清理系统	34
3.1.4 碎片整理	34
3.2 善用快捷键	34
3.3 终端初步	35
3.4 包管理器	35
3.5 Git 初步	36
3.5.1 Git 的工作原理	36
3.5.2 下载 Git	37
3.5.3 Git 信息设置	37
3.5.4 Git 的最基本使用	38
第四章 购买计算机	40
4.1 获取计算机的途径	40
4.2 买机器的原则	40
4.2.1 笔记本电脑的简单分类	40
4.2.2 奸商常见套路	41
4.2.3 专业测评	41
4.3 整机	42
4.3.1 品牌的选择	42
4.3.2 渠道：线上和线下	42
4.3.3 验机、保修	42
4.4 组装机	43
4.4.1 核心配件	44
4.4.2 其他配件	44
4.4.3 显示器	45
4.5 捡垃圾	47
第二部分 大学计算机前置	48
第五章 正式踏入编程世界	49

5.1 编程语言初探	49
5.1.1 编程语言的发展简史	49
5.1.2 编程语言的特点和选择	50
5.2 编程环境的搭建	50
5.2.1 C 系编译器及其环境配置	50
5.2.2 虚拟环境及其配置	51
5.3 选择合适的 IDE 或者文本编辑器	53
5.3.1 选择编辑器	53
5.3.2 安装 VS Code	54
5.3.3 配置 VS Code	54
5.4 美化你的终端	57
5.4.1 安装 Oh My Posh	57
5.4.2 配置 Oh My Posh	57
5.4.3 在 VS Code 中配置终端	58
5.5 编写程序的基本素养	58
5.5.1 编写你的第一个程序	58
5.5.2 学会阅读错误信息	60
5.5.3 学会调试	62
第六章 文字排版	65
6.1 Markdown	65
6.1.1 Markdown 的语法	65
6.2 L ^A T _E X	68
6.2.1 LaTeX 的安装	68
6.2.2 LaTeX 在 VS Code 的配置	70
6.2.3 LaTeX 的语法	71
6.2.4 LaTeX 的常用命令	73
6.2.5 LaTeX 的宏包	77
6.2.6 LaTeX 与汉语专题	77
6.2.7 LaTeX 与多文件	78
6.3 Typst*	79
6.3.1 Typst 的安装	79
6.3.2 Typst 的语法	80
第七章 计算机进阶	83
7.1 Git 进阶	83
7.1.1 分支管理	83
7.1.2 标签管理	85

7.1.3 “摘樱桃”	86
7.1.4 远程仓库	86
7.1.5 GitHub 指南	88
7.1.6 多人协作	90
7.2 密钥进阶	92
7.2.1 SSH 密钥的生成	92
7.2.2 密钥的使用	93
7.2.3 使用 VS Code 建立 SSH 连接	94
7.3 Windows 的文件管理和自动化操作	94
7.3.1 文件系统基础知识	94
7.3.2 Windows 的文件系统结构	95
7.3.3 文件的默认打开方式	96
7.3.4 高效整理与搜索文件	96
7.3.5 搜索工具	96
7.3.6 自动化脚本	96
第八章 开玩 Linux	99
8.1 获取 Linux	99
8.1.1 CLab	99
8.1.2 实机安装	99
8.1.3 使用虚拟机	100
8.1.4 WSL	100
8.2 Linux 的基本操作	101
8.3 Linux 的文件系统	101
8.3.1 文件的组织	102
8.3.2 文件是谁的，有什么属性	102
8.3.3 文件的联系	103
8.4 Linux 的进一步使用	103
8.4.1 root 权限的配置	103
8.4.2 软件的安装及其源的配置	104
8.4.3 再临终端命令行	104
第三部分 走向开发	108
第九章 实用主义编程	109
9.1 写代码的基本素养	109
9.1.1 通用代码风格指南	109

9.1.2 特定语言与社区风格指南	110
9.1.3 注释	111
9.2 防御式编程	112
9.2.1 异常处理	112
9.2.2 断言	113
9.3 监控程序的运行情况	114
9.3.1 日志	114
9.3.2 其他监控手段	115
9.4 常见的代码架构	115
9.4.1 MVC 架构	116
9.4.2 MVVM 架构	116
9.4.3 洋葱架构（干净架构）	116
9.4.4 微服务架构	117
第十章 调试、测试和部署	118
10.1 先救命	118
10.1.1 使用调试器	118
10.1.2 尸检	120
10.2 再治病	120
10.2.1 CPU 吃满但是程序显然不应该吃满 CPU	121
10.2.2 内存吃满但是程序显然不应该吃满内存	121
10.2.3 IO 卡死	121
10.3 再调养	121
10.3.1 测试第一步：隔离	121
10.3.2 单元测试	123
10.3.3 集成测试	123
10.4 买保险	124
10.5 去工作	124
第十一章 常用构建工具链概览	126
11.1 CMake	126
11.1.1 最小运行实例	126
11.1.2 语法	128
11.1.3 工具链	131
11.1.4 安装、导出、打包	132
11.1.5 常见坑	132
11.2 XMake	133
11.2.1 最小运行实例	133

11.2.2 语法速通	134
11.2.3 打包发布	136
11.2.4 常见坑	136
11.2.5 从 CMake 迁移到 XMake	137

第四部分 附录 138

附录 A 迷你 ICS 139

A.1 信息怎么被表示?	139
A.1.1 整数	139
A.1.2 浮点数	140
A.2 程序怎么跑起来?	140
A.2.1 预处理	140
A.2.2 编译和汇编	141
A.2.3 链接	141
A.2.4 常见汇编码	141
A.3 内存怎么被管理?	142
A.3.1 虚拟内存	142
A.3.2 磁盘交换区	142
A.3.3 页面、页表、缺页异常	142
A.3.4 内存分配器	143
A.3.5 一个例子	143
A.4 缓存怎么被管理?	143
A.4.1 缓存的分级	144
A.4.2 缓存行和局部性原理	144
A.4.3 组相联和标签	144
A.4.4 未命中常见工作流程	145
A.4.5 怎么提高代码运行效率?	145
A.5 系统怎么被调用?	145
A.5.1 为什么要有这个系统调用?	145
A.5.2 系统调用长什么样?	146
A.5.3 系统调用的处理流程	146
A.5.4 系统调用的代价与实践尝试	146

附录 B C/C++ 高速入门 147

B.1 C++ 的基本语法	147
B.1.1 基本变量及其运算	148

B.1.2 注释	149
B.1.3 常量	149
B.1.4 判断和循环	150
B.1.5 break 和 continue 语句	154
B.1.6 输入输出	154
B.1.7 基本语法小练	156
B.2 C++ 的进阶使用	158
B.2.1 更进阶的变量类型	158
B.2.2 联合体	159
B.2.3 指针	160
B.2.4 引用	161
B.2.5 函数和变量的作用域	161
B.2.6 函数的递归调用	162
B.2.7 函数的传参	163
B.2.8 小练	164
B.3 C++ 的高级特性	166
B.3.1 命名空间	166
B.3.2 面向对象编程	166
B.3.3 重载	170
B.3.4 模板	170
B.3.5 类型推断	171
B.3.6 类型别名	172
B.3.7 类型强转	172
B.4 STL	174
B.4.1 容器	174
B.4.2 迭代器	175
B.4.3 算法	176
附录 C Python 高速入门	177
C.1 Python 的基本语法	177
C.1.1 Python 的变量	177
C.1.2 Python 的运算	178
C.1.3 输入、输出	178
C.1.4 注释	179
C.1.5 类型强转	179
C.2 控制程序的执行流程	180
C.2.1 条件语句	180

C.2.2 循环语句	180
C.3 复合数据类型	181
C.3.1 列表 (list)	181
C.3.2 元组 (tuple)	182
C.3.3 集合 (set)	182
C.3.4 字典 (dict)	183
C.3.5 高级操作	183
C.3.6 字符串	184
C.4 函数和模块	185
C.4.1 函数	185
C.4.2 模块	187
C.5 文件操作	187
C.6 文科生的 Python	188
C.6.1 Pandas	188
C.6.2 Matplotlib	188
C.6.3 jieba	189
后记	190

第一部分

零基础起步

第一章 初步认识计算机

在真正学习计算机的有关内容之前，我们需要对计算机有一个最初步的认识。计算机是一个复杂的系统，它由许多不同的部件和组件组成，这些部件和组件共同工作以完成各种任务。

1.1 计算机的鼻祖

人们一直在思考怎样进行更高效的计算。在古代，算盘、算筹等工具被用来进行计算。后来，随着科学技术的发展，人们对计算的需求越来越高，开始设计和制造更复杂的计算设备。查尔斯·巴贝奇（Charles Babbage）被认为是计算机的鼻祖，他在 19 世纪设计了第一台机械计算机——差分机和分析机。虽然他的设计在当时没有被完全实现，但他的思想为后来的计算机发展奠定了重要的基础。

我们一般认为，图灵是赋予现代计算机灵魂的人。他在 1936 年提出了“图灵机”的概念，图灵机是一个抽象的计算模型，能够模拟任何计算机的计算过程。自此，问题被分为两类：能用图灵机解决的（或者称作“图灵可计算的”）和不能用图灵机解决的；遇到前者，我们就可以掏出图灵机计算。图灵机的提出为现代计算机科学奠定了基础。关于图灵机本身的内容已经超出了本节课的范围，我们在这里不做过多介绍。

然而，图灵机更多的是一种抽象概念，真正把计算机变成现实的是冯·诺依曼。冯·诺依曼在 1945 年提出了“冯·诺依曼架构”，也就是我们现在所说的计算机的基本结构。冯·诺依曼架构的核心思想是将程序和数据存储在同一块内存中，这样计算机就可以根据程序的指令来操作数据。冯·诺依曼架构是现代计算机的基础，目前几乎所有的计算机都遵循这一架构。

1.2 现代计算机的组成

计算机的组成可以分为**硬件**和**软件**两大部分。硬件是指计算机的物理部件，如中央处理器（CPU）、内存、硬盘、显示器等；软件是指计算机上运行的程序和操作系统，如 Windows、Linux、macOS 等操作系统，以及 Chrome、VS Code、Tencent QQ 等应用程序。

1.2.1 计算机的硬件

计算机的**硬件**，也可以叫做**设备**，可以简单分为两类：一类叫做**主机设备**，是计算机用来进行计算等工作的设备；另一类叫做**外设设备**（也可以叫做**输入输出设备**），是计算机与外界进行信息交互的设备。通常说来，前者是藏在机箱里看不见的，后者是我们能够直接看见的。

一个计算机的主机设备如图所示：

下面将会逐个介绍这些设备。



图 1.1: 计算机的主机设备

1.2.1.1 中央处理器（CPU）

CPU 是计算机的最核心部件，它从存储设备读取指令和数据，并且执行这些指令。尽管现代处理器对代码和数据会有不同的处理，但是其本质上并没有严格的区分。代码由一条一条的指令组成，CPU 按照顺序一条一条执行从存储设备中读取的指令（至少从软件和程序员等使用者的视角看是这样），指令可以是修改 CPU 的状态，进行运算，或者是从其他硬件读取信息或者输出信息。如果希望进一步学习 CPU 如何运作等相关知识，可以参考著名的教材《CSAPP》，也可以修习《计算机系统导论》（ICS）这门课。

1.2.1.2 内存（RAM）

内存是计算机的临时存储器，它用于存储正在运行的程序和数据。它能够被 CPU 直接访问，因此速度较快。对于程序员而言，内存可以被抽象为一堆连续的存储单元，每个存储单元都有一个唯一的地址；执行程序时，程序的一部分或者全部被放进内存中，CPU 就在内存中找寻需要的数据或者指令，如同在排列整齐的暑假上寻找需要的书籍。

现代计算机内存读写速度很快，但是已经跟不上 CPU 的速度，因此又引入了高速缓存来加速内存的读写速度。高速缓存是内存和 CPU 之间的小型存储器，它存储了最近使用的数据和指令，以便 CPU 可以更快地访问它们。在断电以后，内存中存储的数据会丢失，因此内存也被称为是易失性的存储器。

特别注意：上述文本中的“内存”指的是“随机存取存储器”（RAM）。这里的“随机”

指的是“随机存取”，也就是可以在任意时刻访问任意地址的存储器，而不是“顺序存取”的存储器（例如磁带）。同时，“内存”这个词在部分语境下有不同的含义，例如在 BIOS 语境下的“内存”指的是“只读存储器”（ROM），在移动设备（手机）等语境下的“内存”指的是“闪存”，这实际上是外存。

1.2.1.3 外存

外存是现代计算机的主要存储设备，用于存储操作系统、应用程序和数据等内容。其读写速度往往比内存慢得多，但是它的存储容量更大且往往是非易失的（相对内存而言）。

现代计算机的主要外存设备是硬盘。硬盘可以分为机械硬盘（HDD）和固态硬盘（SSD）。机械硬盘使用磁头在旋转的磁盘上读取和写入数据，而固态硬盘使用闪存芯片来存储数据。固态硬盘的读写速度比机械硬盘快得多，现在价格也便宜得多，但是使用寿命较短，且因为电荷流失等问题无法接受长期不通电等情况，不适宜作为长期存档介质（个人使用寿命和 HDD 无明显差异，基本都能用到彻底换机）。

除硬盘外，还有其他外部存储设备。例如：

- U 盘：一种小型的闪存存储设备，通常通过 USB 接口连接到计算机上。本质是一个弱化的 SSD，它可以临时地用于存储和传输数据。
- 光盘：一种使用激光读取和写入数据的存储介质。常见的光盘有 CD、DVD 和蓝光光盘。在大量存档的时候较为经济（大量购买比 HDD 便宜，可靠性比 SSD 高）。缺点是容易划伤和损坏，且信息密度低，读写速度慢。
- 磁带：一种使用磁性材料存储数据的介质，通常用于备份和存档。磁带的读写速度极为缓慢（和倒带速度成正比）且需要专门的设备来读写，设备价格昂贵，维护成本高。其优点是可靠性高，存储密度高。
- 软盘：一种老古董，使用磁性材料存储数据。Windows 计算机中盘符从 C 开始而不是从 A 开始，正是因为 AB 盘符是给软驱用的。现在软盘因为存储容量小、速度慢、易损坏等缺点，已经被淘汰了；但是硬盘盘符从 C 开始的传统保留了下来，成为 Windows 的一个标志性特征。

提醒：硬盘有价，数据无价。请务必定期备份数据，尤其是重要数据。

1.2.1.4 显卡

显卡是计算机的图形处理器，它用于处理图形和视频数据。显卡可以加速图形渲染，提高游戏和视频播放的性能。显卡通常有自己的内存，用于存储图形数据。

对于现在 AI 时代而言，显卡因为其良好的并行特性使得其成为了深度学习的首选硬件。显卡的计算能力通常用“浮点运算每秒”（FLOPS）来衡量，通常情况下，显卡在机器学习等需要大量并行的简单计算工作上，表现远好于 CPU。

1.2.1.5 主板

主板是一块电路板，将所有的硬件设备连接起来。主板上的芯片组负责协调各个硬件之间的通信。同时，主板还有一系列外部接口，用于连接外部设备。

1.2.1.6 电源

电源是计算机的电源供应器，它不参与数据存储与运算等操作，但能够为计算机的各个部件提供所需的稳定工作电压和电流。优质的电源能够避免计算机在运行过程中出现故障，延长计算机的寿命。

1.2.1.7 输入输出设备

输入输出设备指的是计算机与外界进行信息交互的设备。输入设备用于将用户的输入转换为计算机可以理解的格式，而输出设备则将计算机处理后的数据转换为用户可以理解的格式。

最古老的输入设备是拨插电缆，后来变成打孔纸带；现代常见的输入设备包括键盘、鼠标、扫描仪、麦克风等；现代常见的输出设备例如显示器、打印机、音响等。

1.2.2 计算机软件

计算机软件指的是计算机的程序和数据的集合。它可以分为系统软件和应用软件两大类。

1.2.2.1 操作系统

操作系统是计算机的核心软件，它负责管理计算机的硬件和软件资源。操作系统提供了一个用户界面，使用户可以与计算机进行交互。我们可以认为操作系统是连接现代软件和硬件的桥梁。目前，常见的操作系统有 Windows、macOS、Linux 等。

Windows 是目前占有市场份额最大的操作系统。它由微软公司开发，广泛应用于个人计算机。Windows 以其易用性和兼容性而闻名，广泛支持各种软件和硬件设备，但是缺点是不适宜用作开发工具。

macOS 是苹果公司开发的操作系统，专门用于苹果的计算机产品。macOS 以其优雅界面和强大的功能而闻名。

Linux 是一个开源的操作系统，它是一个类 Unix 操作系统。Unix 因为太大了而不适宜在个人计算机上使用，因此 Linus 等人开发了 Linux，但因为学习曲线陡峭，至今未能广泛应用于个人计算机上，服务器和嵌入式系统使用居多。Linux 的开源特性使得它可以被自由修改和分发，因此有很多不同的 Linux 发行版，例如 Ubuntu、Debian、Arch 等。

对于计算机新手，我们推荐使用 Windows 和 macOS 系统作为操作系统，这是因为它们提供了友好的用户界面和丰富的软件支持，适合初学者使用。对于希望深入学习计算机的初学同学，我们推荐使用 Linux 系统的发行版 Ubuntu，因为它具有和 Windows 与 macOS 类似的图形界面，并具有良好的社区支持和丰富的学习资源。对于希望进阶的同学，我们推荐使用 Arch Linux，它是一个轻量级的 Linux 发行版，具有高度的可定制性和灵活性。

1.2.2.2 驱动程序

驱动程序是操作系统和硬件之间的桥梁，它负责将操作系统的指令转换为硬件可以理解的语言。驱动程序通常由硬件制造商提供，并且在操作系统安装时自动安装。驱动程序的作用是使操作系统能够正确地识别和使用硬件设备。

驱动程序通常是特定于硬件的，因此不同的硬件设备需要不同的驱动程序。操作系统通常会自动检测硬件设备并安装相应的驱动程序，但是有时候需要手动安装驱动程序。我们可以到软件官网上下载最新的驱动程序，或者使用操作系统自带的驱动程序更新工具来更新驱动程序。不推荐使用“驱动精灵”等第三方驱动程序更新工具，因为它们可能会安装不必要的驱动程序，甚至可能会导致系统不稳定。

1.2.2.3 应用软件

应用软件指的是我们具体用于实现某一功能的工具。这类软件有很多，我们常用的通讯软件 QQ、微信等，浏览网页的 Chrome、Microsoft Edge 等，都是常规软件。我们下载软件的主要渠道有两种：通过官方渠道下载、通过包管理器（Winget, Homebrew, apt 等）下载。这两种渠道一般认为是最安全且问题最少的。

以安装 QQ 为例，在 Windows 上，我们需要在腾讯官网上找到 QQ 的下载页面，然后下载并安装之。在有包管理器的系统（如 Linux、macOS）上，我们可以通过包管理器下载，例如 `brew qq`（不保证该命令能够执行。你可能需要基于不同系统使用其他的一些命令。）。我们并不推荐在非官方渠道下载软件，这些非官方渠道往往以某某软件站、某某下载站、某某应用商店（Microsoft Store 这类系统自带的除外）等形式出现。通过上述方式下载的软件可能导致使用盗版、附带流氓插件甚至木马、病毒等问题，或者遇到一些其他各种问题。

1.2.2.4 北京大学正版软件

为了保护知识产权、方便学生节约资金，北京大学购买了一系列常用软件的正版授权，方便师生使用。我们可以登录[北京大学正版软件网站](#)，使用自己的北大账号和密码登录，下载和安装这些软件。

1.2.3 实用软件推荐

在学习和工作中，我们常常需要一些实用的软件来提高效率。以下是笔者个人推荐的一些实用软件，以供同学们参考。这些软件中有些是免费的，有些是收费的，具体使用时请注意软件的授权和使用条款。同时，为了防止功能冗余，我们非常建议每类软件只安装一个（尤其是播放器和杀毒软件！）。

- 下载器类

- **Internet Download Manager (IDM)**: 一个极为强大的收费下载软件，可以显著加速下载速度，并支持断点续传等功能。遗憾的是，它不支持磁力链接和 BT 下载。
- **Free Download Manager (FDM)**, 一个免费的下载软件，界面友好且现代，且支持磁力链接和 BT 下载。
- **比特彗星 (BitComet)**: 一个免费且经典的 BT 下载软件，支持磁力链接和 BT 下载。
- **qBittorrent**: 免费且开源的 BT 下载软件。

- 浏览器类

- **Google Chrome**: 一个免费的浏览器，基于 Chromium 内核。
- **Mozilla Firefox**: 一个免费的浏览器，基于 Gecko 内核。
- **油猴**: 一个浏览器扩展，可以让用户自定义网页的样式和功能。它可以通过安装脚本来实现各种功能，例如广告拦截、界面美化等。油猴支持多种浏览器，包括 Chrome、Firefox 等。这里推荐一个链接: **PKU-Art**，它可以给你一个风格现代、足够好看的教学网。

- 压缩与解压缩类

- **7-Zip**: 一个免费且强大的开源老牌压缩软件，支持多种压缩格式，包括 7z、zip、rar 等。它的压缩率高（7z 格式压缩号称全球第一压缩率），速度快，功能强大。
- **NanaZip**: 在 7-Zip 基础上提供更现代化的界面（Windows 11 风格），并增加对 ZStd、LZ4 等压缩算法的编解码支持。此外，它使用 MSIX 打包，因此可上架 Microsoft Store，且可以在 Windows 11 的默认右键菜单中直接使用，而无需打开扩展右键菜单。

- 播放器类

- **VLC Media Player**: 一个免费的开源播放器，支持众多音频和视频格式。
- **MPV**: 免费且开源的播放器，支持格式众多。可以使用命令行、脚本或着色器来精细地控制播放器行为，但上手难度较高。
- **PotPlayer**: 另一个免费的播放器。

- 杀毒软件类

- **Windows Defender**: Windows 系统自带的杀毒软件，功能强大，能够有效地保护计算机免受病毒和恶意软件的侵害。但是它的查杀率和误报率不如其他一些专业的杀毒软件。
- **火绒**: 一个免费的国产杀毒软件，功能强大，能够有效地保护计算机免受病毒和恶

意软件的侵害。它的查杀率和误报率都很低，界面友好，适合普通用户使用。

- 其他

- **Everything**: 一个免费的文件搜索工具，能够快速搜索计算机上的文件。它的搜索速度极快，支持多种搜索方式，包括模糊搜索、正则表达式搜索等。
- **Wallpaper Engine**: 一个收费的动态壁纸软件，能够让桌面变得更加美观。它支持多种动态壁纸，包括视频壁纸、动画壁纸等。
- **Rufus**: 一个免费的U盘制作工具，能够将ISO镜像文件写入U盘，制作成可启动的U盘。它支持多种操作系统的ISO镜像，包括Windows、Linux等。
- **UltraISO**: 一个收费的光盘镜像制作工具，能够创建、编辑和转换光盘镜像文件。它支持多种光盘格式，包括ISO、BIN、CUE等。
- **VMware/VirtualBox**: 两个免费的虚拟机软件，能够在计算机上创建虚拟机，运行其他操作系统，可以用于测试软件、学习操作系统等。
- **Cherry Studio**: 一个LLM管理器，能够帮助你使用各种LLM来简单地创建Agent，来辅助你的开发和生活。

1.2.4 怎样卸载软件

我们不推荐反复装卸软件，因为这可能会导致系统不稳定或者软件残留。但是有些时候，我们认为某个软件长期内不会再需要了，且磁盘空间告急，这时我们应该考虑将其卸载。

计算机小白最喜欢做的一件事是把桌面上的快捷方式移动到回收站，这是非常错误的做法。快捷方式只是指向软件的一个链接，删除快捷方式并不会卸载软件本身。对计算机半懂不懂的人喜欢找到软件的安装目录，直接删除软件的文件夹，这也是错误的做法。因为对于许多软件而言，这样做会导致软件的注册表项和其他配置文件残留在系统中，可能会导致系统不稳定或者软件无法正常工作。

正确的做法有两种：要么使用计算机自带的“程序与功能”界面删除软件，要么使用软件自带的卸载程序（通常命名为 `uninstall.exe` 或者类似名称）。某些软件可能会在安装时提供一个卸载程序，我们可以在开始菜单或者软件的安装目录中找到它。使用这些方法可以确保软件被完全卸载，留下的残留文件也较少。如要彻底删除残留文件，可以使用一些专业的卸载工具，例如 Geek 等。

对于 macOS 用户而言，卸载软件不需要上文叙述那么麻烦，只需要将应用程序拖到废纸篓中即可！

1.3 计算机间的通讯

计算机的通讯是指计算机之间或者计算机与其他设备之间进行信息交换的过程。目前计算机间的通讯主要是靠网络来实现的。网络是由许多计算机和其他设备通过通信协议连接在一起的系统，两大要素是**网络协议**（数据要依赖统一的协议传输）和**网络设备**（用于连接的

设备)。

1.3.1 网络基本名词及其解释

不论是修电脑，还是平常使用计算机联网工作，我们总会在一些场合听到一些网络名词，这些网络名词不乏有被误解的。下面我们将对一些常见的网络名词进行解释。

1.3.1.1 IP 地址和端口号

IP 地址是计算机在网络中的唯一标识符，类似于“门牌号”。目前有两种通行的 IP 地址：IPv4 和 IPv6。IPv4 地址是一个 32 位的二进制数，通常用点分十进制数表示（例如 192.168.1.1）。IPv6 地址是一个 128 位的二进制数，通常用冒分十六进制数表示。IPv6 地址的引入是为了应对 IPv4 地址耗尽的问题。

端口号是计算机在网络中用于区分不同应用程序的标识符，类似于“房间号”。端口号是一个 16 位的整数，范围从 0 到 65535。常见的端口号有 80 (HTTP)、443 (HTTPS)、22 (SSH) 等。我们假设要指向某一个计算机上的某一个应用程序，那么我们需要指定该计算机的 IP 地址和端口号。IP 地址和端口号一起构成了一个完整的网络地址，通常表示为“IP 地址: 端口号”（例如 0.0.0.0:8000）。

1.3.1.2 域名

域名是计算机在网络中的人类可读的标识符，类似于“网站名称”。域名由多个部分组成，通常用点分隔（例如 www.pku.edu.cn）。域名系统 (DNS) 将域名转换为 IP 地址，以便计算机可以通过 IP 地址进行通信。

1.3.1.3 子网掩码、网关

子网掩码是一个 32 位的二进制数，用于划分 IP 地址的网络部分和主机部分。它通常用点分十进制数表示，它与 IP 地址进行按位与运算后，可以得到网络地址。子网掩码的作用是将一个大的网络划分为多个小的子网，以提高网络的效率和安全性。

网关是计算机在网络中的出口，用于连接不同的网络。网关通常是一个路由器或者交换机等设备。

1.3.1.4 内网和外网

我们在日常生活中，常常会听到“内网”和“外网”这两个词。内网是指一个局域网内部的网络，通常用于家庭、学校或者公司等小范围的网络。内网中的计算机可以通过路由器或者交换机等设备连接到外网。外网是指互联网上的网络，通常用于连接不同的局域网和广域网。

内网和外网的 IP 地址往往是不同的。内网 IP 地址通常是私有的 IP 地址，仅在内网中有效（例如每一个地级市都可能有一个“二中”，但是在不同的市称呼“二中”指的是不是同一个学校）；而外网 IP 地址全球唯一，互联网可以访问（例如“东港二中”）。例如大名鼎鼎的 8.8.8.8 是 Google 的公共 DNS 服务器的 IP 地址，它是一个外网 IP 地址。如果在内网中访问该地址，则可能访问到的不是 Google 的 DNS 服务器，而是内网中的某个设备。

1.3.1.5 MAC 地址

MAC 地址是计算机网络接口的唯一标识符，类似于“身份证号码”。它是一个 48 位的二进制数，通常用冒分十六进制数表示（例如 00:1A:2B:3C:4D:5E）。MAC 地址用于在局域网中唯一标识一个设备。每个网络接口卡（NIC）都有一个唯一的 MAC 地址。MAC 地址通常由设备制造商分配，并且在设备的硬件中存储。

1.3.1.6 网络协议

网络协议是计算机之间进行通信的规则和约定。它定义了计算机如何发送和接收数据，以及如何处理错误和异常等情况。常见的网络协议有 TCP/IP、HTTP、FTP 等。不同的网络协议适用于不同的应用场景，例如 TCP/IP 协议适用于可靠的数据传输，而 HTTP/HTTPS 协议适用于 Web 应用程序的通信、UDP 适用于精度要求不太高的实时通信等。

1.3.1.7 网络设备

网络设备是用于连接计算机和其他设备的硬件设备。常见的网络设备有路由器、交换机、集线器等。**路由器**用于连接不同的网络，并且可以根据网络协议进行数据转发；**交换机**用于在同一局域网内连接多个设备，并且可以根据 MAC 地址进行数据转发；**集线器**用于将多个设备连接到同一个网络，但不具备智能转发功能。

调制解调器（Modem，猫）是用于将数字信号转换为模拟信号的设备，通常用于连接到电话线或者有线电视网络。调制解调器可以将计算机发送的数据转换为模拟信号，并且将接收到的模拟信号转换为数字信号。我们家用的宽带通常是通过调制解调器连接到互联网的。

目前家用的路由器往往集成了调制解调器和交换机的部分功能，因此我们往往不需要像以前一样购买一大堆设备了。

1.3.2 校内网络配置指南

1.3.2.1 计算机如何联网

数据通过网络的传输需要以“数据包”的形式进行。数据包是网络传输的基本单位，它包含了发送方和接收方的 IP 地址、端口号、数据等信息。数据包通过网络设备（如路由器、交换机等）进行转发，最终到达接收方。

一台具体的机器，进行联网的步骤如下：

1. **物理连接**：使用有线或者无线的方式，将计算机连接到网络设备（如路由器、交换机等）。
2. **IP 地址分配**：计算机通过 DHCP 协议自动获取 IP 地址、子网掩码、网关和 DNS 等信息。
3. **网络协议配置**：计算机根据网络协议栈（如 TCP/IP）配置网络协议，确保数据包的正确传输。
4. **应用层协议**：计算机通过应用层协议（如 HTTP、FTP 等）与其他设备进行通信。
5. **数据传输**：计算机通过网络设备和协议，将数据包发送到目标设备，并接收返回的数据包。

1.3.2.2 有线连接 PKU 校园网

我们可以通过有线连接 PKU 校园网的方式来联网。在宿舍或者有网线接口的地方，我们可以将网线插入计算机的接口，这样就可以直接连接到校园网和互联网。

校园网的网络安全比较一般，建议购买一个路由器，连接到网线接口上，然后通过路由器连接到计算机。这样可以提高网络的安全性和稳定性。

1.3.2.3 无线连接 PKU 校园网

你可以在支持无线网络的设备 WLAN 列表中找到以下和北京大学有关的无线网络：

- **PKU**：不安全，不建议使用。
- **PKU Secure**：采用 IEEE 802.1x 技术，能为用户提供较为安全的加密链路连接，一般情况下建议使用这个网络。
- **PKU Visitor**：访客网络，学生不需要使用这个。
- **My BJMU**：北大医学部的无线网络。
- **eduroam**：全球教育科研网络，该网络可以在全球许多高校使用。

具体怎样连接 PKU Secure 网络，请参考[PKU Secure 连接指南](#)；使用 eduroam 的同学也可以参考[eduroam 连接指南](#)。

1.3.2.4 校外连接北大内网

为方便北京大学校园网用户在校外（家中、出差或国外）访问校园网资源，计算中心提供了 VPN 服务，可安全地接入校园网，如同在校内一样方便地访问学校全部的内网资源与服务（如校内门户、电子期刊数据资源等）。

具体怎样操作是一件比较复杂的事情，建议参考[北京大学 VPN](#)的指南进行操作。

1.3.2.5 北大网盘

北大网盘是北京大学为师生提供的云存储服务，用户可以通过北大网盘存储、共享和管理文件。北大网盘提供了大容量的存储空间，并支持多种文件格式的上传和下载。

具体的使用可以参照[北大网盘指南](#)进行操作。

1.3.2.6 PKU 腾讯会议教育版

为了解决腾讯会议教育版资源紧张、预约繁琐的问题，计算中心上线了“腾讯会议预约申请”系统。

具体使用可以参照[腾讯会议教育版指南](#)进行操作。

腾讯会议教育版只用于学习和工作用途，临时、小规模或其他用途的会议请使用个人账号。

1.3.2.7 北大邮箱及其在第三方客户端的配置（以 Outlook 为例）

北京大学为每一个新生都提供了一个北大邮箱。新生的邮箱服务一般由网易提供，地址是 < 学号 >@stu.pku.edu.cn。北大邮箱可以用于接收学校的通知、邮件等信息。

为了方便起见，我们往往习惯于将邮箱配置到第三方客户端（如 Outlook、Thunderbird 等）上，以便于管理和使用。我将以 Outlook 为例，介绍如何配置北大邮箱。

首先，打开 Outlook，点击“文件”菜单，然后选择“添加账户”。在弹出的窗口中，选择手动进行配置。下文使用 IMAP-SMTP 协议进行配置。

pku.edu.cn 用户：

接收（IMAP）邮件服务器：imap.pku.edu.cn

发送（SMTP）邮件服务器：smtp.pku.edu.cn

在“发送邮件服务器”选项选定“我的服务器要求身份验证”。

SMTP SSL 安全连接使用 465 端口，IMAP SSL 安全连接使用 993 端口。

stu.pku.edu.cn 用户：

接收（IMAP）邮件服务器：imaphz.qiye.163.com

发送（SMTP）邮件服务器：smtphz.qiye.163.com

SMTP SSL 安全连接使用 994 端口，IMAP SSL 安全连接使用 993 端口。

在进行正确的配置之后，Outlook 会自动连接到北大邮箱服务器，此时会提示你输入用户名和密码。用户名一般是你的邮箱地址。对 pku.edu.cn 用户而言，密码是你在北大邮箱的登录密码；对 stu.pku.edu.cn 用户而言，密码应在网易邮箱客户端中获取（登录网易邮箱网页客户端后，设置 > 客户端设置 > 客户端授权密码）。输入正确的用户名和密码后，Outlook 会自动连接到北大邮箱服务器，并开始同步邮件。

自 2025 年 7 月 2 日始，北京大学邮箱服务建议使用 *pku.edu.cn* 域名的用户启用北京大学邮件系统二次验证和客户端专用密码功能。详见[通知原文](#)，其中也包含了如何启用二次验证和客户端专用密码的说明。

1.4 网安相关知识

1.4.1 网络的风险

虽然互联网的出现给我们带来了便利，但也带来了很大风险。部分人心术不正，使用互联网进行诈骗、盗窃、敲诈勒索等违法犯罪活动；而他们使用的主要手段是不定向的网络攻击，例如钓鱼、木马和病毒等。

钓鱼指的是使用伪造的网页、邮件等方式，诱骗用户输入个人信息，例如用户名、密码、银行卡号等。其目的通常是获取用户的私人信息，以方便对其进行后续的诈骗、勒索等活动。

木马这个词来源于神话中的“特洛伊木马”，原指在一只巨大的木马中藏匿士兵，诱骗敌人打开城门进而发动攻击。现在的木马指一种恶意软件，它伪装成合法的软件或者捆绑在合法软件中，诱骗用户安装。一旦安装，木马就可以在用户不知情的情况下，窃取用户的个人信息、打开端口等。例如，一种最古老且经典的木马是 FTP 木马，它会在用户的计算机上打开一个 FTP 端口，允许攻击者远程访问用户的计算机。

蠕虫指的是一种自我复制的恶意软件，它可以在计算机之间传播。蠕虫通常利用计算机系统的漏洞进行传播，一旦感染了一个计算机，就会自动复制自己并传播到其他计算机。蠕虫通常会消耗计算机的资源，导致计算机变得缓慢或者崩溃。一个很经典的蠕虫是“小邮差”，它通过发送带毒邮件进行传播，会占满计算机的网络带宽；一旦感染了计算机，就会自动发送带毒邮件给其他计算机。

病毒指的是一种恶意软件，它可以在计算机之间传播。病毒通常依附在合法的软件中进行传播，一旦感染了一个计算机，就会自动复制自己并传播到其他计算机。病毒通常会破坏计算机的文件、数据等，导致计算机无法正常工作。一个很经典的病毒是“CIH”，它会在每年的 4 月 26 日感染计算机，并破坏计算机上的所有文件。它与蠕虫的区别是，病毒不能自己执行，只能依附于其他软件执行；而蠕虫可以自己执行。

被上述恶意软件感染后，计算机变得不稳定、产生额外的资源开销，甚至导致计算机崩溃、数据破坏，造成经济或其他损失。君子爱财取之有道，我们应该遵纪守法，不要为了炫耀技术或者获取经济利益而制作这些恶意软件。

1.4.2 从根源断绝问题

为了防止计算机受到感染和破坏，最简单的方式是从根源上解决问题。我们在日常使用网络的时候，应该遵循以下原则：

不浏览不安全网页：在浏览网页的时候，若不能确定安全性，则尽量避免浏览不明链接、下载不明文件；如果确实需要下载软件，应该到官方网站上下载。

识别伪造网站：当一个网站需要你输入个人信息时，应该仔细检查该网站的安全性；钓鱼网站通常会伪装成合法网站，例如使用 HTTPS 协议、与合法网站相似的域名等。我们可以通过查看浏览器地址栏中的锁图标、检查网站的证书、观察域名是否正确等方式来判断网站的安全性。北京大学计算中心每年都会主动制作钓鱼网站，测试本校教职工和学生抗钓鱼的

能力。虽然被计算中心骗了是一件不太光彩的事情，也总比被其他人骗了好，至少不会损失钱财。

保持系统和软件的更新：系统和软件的更新有一大部分是安全更新。定期更新软件可以阻止恶意软件利用漏洞进行攻击。

保持杀毒软件的自动检测功能开启：这类检测功能可以帮助我们初步检测计算机上的恶意软件。虽然有时候存在令人诟病的误报，但是它们仍然是我们保护计算机的一道重要防线。

使用密钥代替密码：密钥是一种更安全的身份验证方式，它可以防止密码被窃取。除了常用的公钥-私钥对，密钥还可以是 USB 设备、手机等。使用密钥可以防止密码被窃取和破解。目前一些技术网站已经支持使用密钥登录，例如 GitHub、Google 等。而我们在登录远程服务器的时候，也建议使用密钥而非密码登录。

使用强密码并定期更换：如果不得不使用密码登录，建议使用复杂的密码，并定期更换密码。复杂的密码应该包含字母、数字和特殊字符，并且长度至少为 8 位。我们也可以使用密码管理器（例如 BitWarden）来生成和管理复杂的密码。

定期备份数据：定期备份数据可以防止数据丢失和损坏。数据备份有一个 321 原则：3 份数据，2 种介质，1 个异地备份。也就是说，我们应该至少有 3 份数据备份，其中 2 份存储在不同的介质上（例如移动硬盘和云存储），1 份存储在异地（例如云存储）。这样，即使我们的数据损坏了，也可以迅速恢复来减少损失。

善用沙箱：沙箱是一种虚拟化技术，可以将应用程序隔离在一个独立的环境中运行。这样可以一定程度上防止恶意软件对计算机造成损害。我们可以使用虚拟机、Docker 等工具来创建沙箱环境，对于不能确定安全性的程序可以在此类环境中运行。

1.4.3 亡羊补牢，为时未晚

如果发现自己被钓鱼，应该紧急冻结相关账户并迅速更改密码，防止进一步的经济损失。在接下来的一个月到数个月中务必慎之又慎，你的信息可能已经被泄露，招致电信诈骗的概率显著增高。

如果发现自己的计算机感染了木马、病毒、蠕虫等，此事比被钓鱼更加严重。你应该依次执行以下内容：

- **立即结束进程：**如果你能定位具体是哪一个软件正在搞破坏，可以使用任务管理器结束该进程。不过大多数情况下我们无法确定具体是哪一个进程出问题，此时忽略这一步即可。
- **立即断网：**这是一种负责任的行为，可以防止病毒进一步扩散。仅在软件层面上切断网络并不足够，如果你的计算机使用有线网络应该拔掉网线，以防止恶意进程重新连接网络。
- **立即查杀：**你可以运行杀毒软件的查杀功能，如是旧种类的病毒，杀毒软件应对起来不会太难。
- **立即上报：**如果杀毒软件查杀失败，应该立即上报相关部门。这可能意味着新型病毒的

出现，上报有关部门有利于他们做出迅速反应，可以减少损失。

- **立即重装系统：**彻底重新格式化硬盘并重装系统可以彻底地清除残留的病毒文件。这是没有办法的办法，但却是最有效的。

第二章 搜索和信息获取

在大学，上课和课本固然是一种重要的信息获取方式。但是课程和课本本身由于是静态的，往往无法及时更新最新的信息；然而对计算机科学等发展迅速、信息爆炸、对技术要求较高的科目，仅凭借课本等静态资源显然是远远不够的。因此，我们需要借助其他方式来获取信息。

2.1 搜索

2.1.1 搜索引擎的选择

国内最常见的搜索引擎是百度。但是当我们在百度搜索相关内容时，第一页往往会被大量的广告占据。这显然并不是我们想要的结果。其他的国内搜索引擎都或多或少有相关的问题，因此并不好用。

由于现在大多数新购整机都预装了 Windows 正版系统，因此基本上都自带一个内置浏览器 Microsoft Edge。Edge 的默认搜索引擎是必应（Bing），在搜索的时候我们可以在页面顶端发现“国内版”和“国际版”的选项。在使用国内版搜索时，仍然会出现少量的广告和不相关的内容，但是相对百度而言，必应的搜索结果要好得多。在使用国际版搜索的时候，必应的搜索结果会更好，但是由于网络问题，可能会出现无法访问的情况。

细心的同学可能会发现，我们从国内网络访问 Bing，无论是国内版还是国际版，网址都是 cn.bing.com。而真正的 Bing 的网址是 www.bing.com。有条件能够访问这一网址的同学可以使用这个 Bing。而 Google 作为全球最大的国际搜索引擎，搜索结果通常比必应还要准确、直接且全面。

因此，不使用特殊方式上网的情况下，如果我们要搜索的信息非中文社区独有，我们更推荐使用必应的国际版搜索引擎。在该课程中不将涉及任何特殊上网方式的教学。

2.1.2 搜索技巧

有时候我们搜索的时候无法搜索到想要的信息。这时候我们需要使用一些技巧。

关键词搜索是最常见的搜索技巧之一。我们使用完整句子进行搜索的时候，搜索引擎会利用语言模型将其拆分成多个关键词进行搜索，而语言模型总会导致一定的偏差。因此，我们可以一步到位，使用关键词进行搜索。例如，我们如果想要搜索“我怎样改善睡眠质量”，可以把它拆分成“改善睡眠方法”关键词进行搜索；如果需要进一步约束（例如我希望方法快速起效），可以搜索“改善睡眠 方法 快速”。

使用英文是另一个常见的搜索技巧。中文互联网的一大特点是信息向应用内部收缩，形成无法被搜索引擎检索到的“深网”，导致中文开放互联网的信息量小于英文开放互联网的信

息量。使用英文搜索的另一个原因是英语依然是世界上最通用的语言，尤其在技术、科学等领域，大部分的文献、资料、教程、说明等都是用英文写的；相关领域的研究材料往往也先以英文发表。因此我们在搜索的时候，使用英文搜索往往能够得到更好的结果。

即便英文水平一般的同学也不必担心。我们可以使用翻译软件（例如微软翻译、有道翻译等）将中文翻译成英文，然后再进行搜索。

使用高级搜索选项也是一种搜索技巧，最常见的高级搜索选项有：

- 使用引号将关键词括起来，这样搜索引擎就会强制将其视为一个整体进行搜索，而不是将其拆分成多个关键词。依然以改善睡眠为例，可以使用“如何改善睡眠质量”进行搜索；
- 使用减号将不需要的关键词排除在外。例如，我们想要改善睡眠，但是不想看到关于药物的信息，可以使用“改善睡眠 方法 快速 -药物”进行搜索，这样搜索引擎就会把含有药物的信息排除在外；
- 使用“site:”限制搜索范围。例如，我们如果想要搜索“如何改善睡眠质量”，但是只想看到来自知乎的信息，可以使用“改善睡眠 方法 快速 site:zhihu.com”进行搜索。

判断信息的可靠性虽然不属于搜索技巧，但是却是一个非常重要的技能。我们在搜索到信息的时候，往往需要判断其可靠性。我们可以从以下几个方面来判断信息的可靠性：

- 来源：信息的来源是否可靠？是否来自权威机构、专家或者知名网站？
- 时间：信息是否及时？是否过时？
- 评价：其他人对该信息的评价如何？是否有很多人认可？
- 完整性：信息是否完整？是否有遗漏？
- 可验证性：信息是否可以被验证？是否有相关的证据？

2.2 信息平台

除了使用搜索引擎在信息平台上搜索以外，我们还可以直接在著名的信息平台上面寻找相关信息。

2.2.1 官方文档、Wiki、论坛

如果我们希望获取某软件等的信息，最好的地方往往是其官方文档；对于类似于 Arch Linux 这种纯由社区维护的项目，其官方 Wiki 与论坛也是获取信息的最佳选择之一。官方文档虽然可能存在晦涩、难懂、省略等问题，但是往往依然是最权威、最全面的文档，这将会是你学习一门新技术的最佳选择。

如果你在请求问题的时候，遇到了诸如“RTFM”（Read The F**king Manual）的回应，这说明回答者认为你需要搜索官方文档和使用手册。当然在这种情况下，他大概率是对的，你应该去读一读。同样道理的还有 STFW（Search The F**king Web）和 RTFSC（Read The F**king Source Code）。而往往通过这种方式搜索信息，你能够学到的内容比直接告诉你答案要多得

多。

2.2.2 Stack Overflow

堆栈溢出（Stack Overflow）是一个程序员问答网站，专门用于解决编程和技术问题。它是一个社区驱动的网站，用户可以在上面提问和回答问题。堆栈溢出有一个强大的搜索功能，可以帮助用户快速找到相关的问题和答案。从我的个人使用体验而言，这东西有点像百度贴吧和知乎的结合体，且专业性比两者都要强得多。

2.2.3 GitHub

GitHub 是一个代码托管平台，用户可以在上面存储和分享代码。GitHub 上有很多开源项目，用户可以在上面找到相关的代码和文档。GitHub 还提供了一个强大的搜索功能，可以帮助用户快速找到相关的项目和代码。同时，GitHub 也是一个非常重要的开源社区，当你某个项目有疑问或者发现 Bug 的时候，你可以对该项目提出 Issue，只要项目没“死”，总会有人告诉你答案；当你想要为某一项目做出贡献的时候，你可以 Fork 该项目，然后提交 Pull Request。

2.2.4 Wikipedia

维基百科是一个自由的百科全书，用户可以在上面找到各种各样的信息。维基百科是一个社区驱动的网站，用户可以在上面编辑和修改条目。维基百科的内容是由志愿者编写和维护的，因此它的准确性和可靠性可能较低，不过它仍然是一个非常有用的信息来源。维基百科的搜索功能也很强大，可以帮助用户快速找到相关的条目。

2.2.5 其他著名博客和教程

W3Schools 提供了许多关于开发的教程和示例，适合初学者使用。它的内容覆盖了 HTML、CSS、JavaScript、SQL 等多个领域。国内也有类似的网站，例如菜鸟教程、W3School 等，只是内容丰富程度上较为逊色。

OI Wiki、**CTF Wiki**、**HPC Wiki** 是一些关于算法、数据结构、编程竞赛等方面的 Wiki，适合对这些领域感兴趣的同学使用。它们的内容覆盖了算法、数据结构、编程竞赛等多个领域。这些 Wiki 则是由在相关领域耕耘多年的选手前辈们维护的，内容质量较高。

CS 自学指南是由信科的一位学长发起、旨在帮助计算机专业的同学自学计算机科学的一个项目。它的内容覆盖了计算机科学的各个领域，包括计算机网络、操作系统、编译原理等。它的内容质量较高，适合对计算机科学感兴趣且希望自学的同学使用。

2.2.6 国内优质平台

我们一般认为国内能够算上优质平台的有：博客园、哔哩哔哩、知乎、简书。这些平台普遍是免费的，你可以找到许多关于技术、编程、科学等方面的文章和视频。它们的内容质量参差不齐，也不乏卖课的（例如我曾经在 B 站看到过“预测 2025 年将会淘汰的编程语言：C/C++、Java、C#、Golang、Python”等视频，当然这显然是胡扯），但是它们仍然是一个非常有用的信息来源。我们在接受信息的时候，仍然需要判断其可靠性。

特别说明：CSDN 上虽然也有不少信息，但是该平台质量较低，商业化程度较高。这导致在该平台寻找信息的时候，我们必须在海量的 AI 水文、抄袭博客、低质付费文字、商业广告等无用信息中找到夹缝中的少数高质量文章，这是一件极为痛苦的事情。虽然在少数情况下我们最终能够找到一些有用的信息，但是高质量的平台能节约鉴别信息的精力。

2.3 LLM、VLM 和提示词工程

现在，大语言模型（Large Language Model, LLM）已经广泛地投入了使用，无论是 ChatGPT、Claude、Gemini 等国外著名 LLM，还是国内的 DeepSeek、Kimi、通义千问等 LLM，都已经投入了广泛应用。LLM 使得我们获取信息的方法变得更加简单高效，我们可以把它们当作一个搜索引擎来使用。

部分 LLM 支持多模态，能够处理文本、图像等多种输入，这些 LLM 也被称作是 VLM。它们在处理图像和文本的时候，能够提供丰富的信息和更好的交互体验，有广泛的应用。

2.3.1 选择 LLM

市面上有许多 LLM 可供选择，不同的 LLM 在性能和擅长用途上都有所不同。为了帮助我们选择合适的 LLM，我们可以参考一些主流的 LLM 评测榜单。这些榜单通过标准化的测试来评估不同模型的性能，为我们提供了有价值的参考。

以下是一些推荐的适合初学者大致了解情况的 LLM 评测榜单：

- **综合性能榜单**：**Artificial Analysis LLM Leaderboard** 是一个综合性的 LLM 性能排行榜。它从多个维度评估模型解决问题的能力，包括数学能力、推理能力、知识水平、代码能力等，适合用于了解一个模型的综合实力。榜单中还包含了各项具体测试的分数和排名，帮助我们更好地比较不同场景下模型的性能。
- **模型幻觉评估**：**HHEM Leaderboard** (Hughes Hallucination Evaluation Model, HHEM) 是一个专门评估 LLM “幻觉”程度的榜单。LLM 的幻觉指的是模型会生成一些看似合理但实际上是错误的或者无中生有的信息。这个榜单对于那些对信息准确性要求较高的应用场景非常有参考价值。
- **上下文性能评估**：**Context Arena** 和 **Fiction.liveBench** 这两个榜单专注于评估模型处理长上下文的能力。如果你的应用需要处理大量的文本，例如长篇文档分析或者需要模型记住很长的对话历史，那么这两个榜单的结果会很有帮助。

需要强调的是，我们应该理性看待这些 LLM 榜单。榜单的排名是基于特定的测试集和评估方法得出的，不一定能完全反映模型在所有场景下的表现。因此，在选择 LLM 时，除了参考榜单，我们还应该结合自己的具体需求、应用场景和预算来进行综合考量。最好的方法是亲自试用几个排名靠前的模型，感受它们的实际表现。

2.3.2 使用 LLM 的基本原则

LLM 目前依然只是一个按照概率分布生成文本的模型，而不是一个真正“理解”语言的东西；它的输出是基于统计数据和模式，而不是基于对世界的真正理解。而这个概率除了受到语法、语义等语言学因素的影响和模型本身的影响以外，还受到输入的 Prompt（提示词）的影响，因此我们可以通过优化 Prompt 来在不改进模型性能的条件下尽量优化 LLM 的输出。这个领域被称为“提示词工程”（Prompt Engineering）。

具体来说，我们要遵循以下原则：

- **具体性**：使用 LLM 的时候提问应该极为具体，避免使用模糊、省略的语言或者关键字。例如我们如果想要获取改善睡眠质量的信息，应该使用“如何改善睡眠质量”而不是“改善睡眠”等关键字组合。
- **明确性**：在使用 LLM 的时候，我们的 Prompt 应该明确无歧义。这在 LLM 上面有一个专门的课题叫做 WSD（消歧）。例如“Have a friend for dinner”，我们应该明确地解释成“treat your friend to dinner”或者“Eat your friend”，而不是让 LLM 去猜。与之类似的是我们可以在 Prompt 中规定其输出格式，例如提供一个示例，这对获得期望的输出非常有效。
- **简单化**：目前的 AI 依然缺乏处理复杂问题的能力。当我们提出一个复杂的问题时，LLM 往往会混乱，进而得出错误答案。这时，我们可以采用分治思想，把一个大问题分成多个小问题，然后让 LLM 分别解决这些小问题，然后合并答案。

使用 LLM 非常利于我们学习计算机相关知识。但是，部分同学会将作业和代码一股脑的扔给 LLM 让其完成，然后自己甚至连理解一遍代码的含义都不去做。这样的行为不仅是对课程不负责任，也是对自己的工程能力不负责任。我们鼓励同学们多使用 LLM 学习计算机等相关知识，但是对于代码为主（而不是结果为主）的工程和作业，要尽可能地减少让 LLM 生成大段大段的代码。

2.3.3 LLM 的局限性

LLM 虽然强大，但是它仍然有一些局限性，主要集中在信息滞后与幻觉两大方面。

目前，LLM 依然使用的是 Transformer 架构，这使得它的知识库是静态的。LLM 在训练完成之后，其知识库就不会再更新了。这就导致了 LLM 无法获取最新的信息。虽然有些 LLM 会定期更新其知识库，但是更新的频率往往较低，且更新的内容也有限。因此，LLM 对于截止日期之后的信息往往无法提供准确的答案。而幻觉的来源也很简单：毕竟现在的 LLM 依然只是在做猜词游戏而已，犯错误非常正常。

所以说，虽然我们将 LLM 作为一个获取知识的渠道并把它当作一个“超级搜索引擎”来使用，但是我们依然要对其输出内容进行验证，尤其是在其知识库截止日期后的内容：LLM 对于一个它不知道的问题往往不会回答“不知道”，而是胡编一个答案出来，这在某些情况下是不可忍受的。

因此，我们在使用 LLM 的时候，仍然需要保持批判性思维。如果我们希望获取一些旧而笼统的信息，使用 LLM 能提高我们的搜索效率，并且答案往往是可信的；如果我们希望获取一些新信息或者较为精确的信息，使用 LLM 是显著不如搜索的。

2.3.4 提示词工程简介

提示词工程指的是优化 LLM 的输入提示词（Prompt）以获得更好的输出结果的过程。提示词工程的目标是通过调整输入提示词的内容、格式和结构，使得 LLM 能够更准确地理解用户的意图，从而生成更符合预期的输出。

提示词（Prompt）是指输入给 LLM 的文本。一般情况下，提示词分为系统级提示词和用户级提示词两种，系统级提示词规定了 LLM 的行为和输出格式等内容，而用户级提示词则是用户输入的具体问题和请求。一般情况下，系统级提示词我们是不可见也无法修改的，而用户级提示词则是我们可以修改的。因此，下文中提到的提示词，如非特别说明，均指用户级提示词。

2.3.4.1 角色-任务-约束-范式

我们通过明确角色、任务和约束来指导 LLM 生成更符合预期的输出。比如说：

你是一个科普作家。你要给初中学生科普“熵”是什么。你需要使用通俗易懂的语言，避免使用专业术语，并且要举例说明。请用不超过200字的篇幅来解释。

以上 Prompt 仅包含四句话，但是非常有效地指导了 LLM 的输出。我们可以看到，以上 Prompt 明确了角色（科普作家）、任务（给初中学生科普“熵”）、约束（通俗易懂、避免专业术语、举例说明、篇幅不超过 200 字）和范式（使用自然语言）。这种方法可以帮助 LLM 更好地理解用户的意图，从而生成更符合预期的输出。

我们一般把以上提示词称为“角色-任务-约束-范式”，简称 RTCP。在提示词工程上，上述方法满足“少样本提示”和“角色扮演提示”的定义。

2.3.4.2 思维链和零样本思维链

思维链指的是模型将问题分解成多个子问题，并逐步解决每个子问题的过程。思维链可以帮助模型更好地理解问题，并生成更符合预期的输出。举例：

小明有10个苹果，他吃掉了一个苹果，他又吃掉了2个苹果，他现在有几个苹果？

这样手动加入思维链可以显著增强模型的推理能力。另一种方式是零样本思维链，也就是说：

...正常的Prompt...

请逐步推理。

“请逐步推理”这五个字¹可以激活模型的内部推理能力。零样本思维链在 2022 年被发现，现在已经被广泛应用于提示词工程中。

2.3.4.3 自洽采样和反思提示

有时候，我们可以多次访问 LLM。这时候，我们可以使用自洽采样和反思提示的方法，来优化 LLM 的输出。

自洽采样指的是在同一个问题上反复询问多次，之后对多次回答进行统计分析，取其中相同结果最多的结果（或者结果的均值、中位数等统计量）为最终的结果。这样可以减少模型的随机性和不确定性，提高输出结果的可靠性。这比贪心采样（指取一次输出）效果好许多，有效提高了模型的输出质量。

反思提示指的是对于有上下文的模型，使用模型的输出作为下一次输入的一部分，并让模型反思它的输出是不是有不合理之处（例如“请分析以上证明过程有无循环论证？”）。这样可以在一定程度上规避错误，提高模型的输出质量。

2.3.4.4 上下文工程

上下文工程指的是利用 LLM 对上下文的处理方式来确定 LLM 的输出。有时候我们的 Prompt 非常长，或者包含了大量的信息，这时候 LLM 可能会忽略掉一些重要的信息，导致输出结果不符合预期。为了避免这种情况，我们可以使用上下文工程来优化 Prompt。我们最常见的两个方式是：

- 关键信息放开头：LLM 在处理信息的时候有着显著的首因效应（或者中间丢失效应）。我们可以尽可能地把关键信息放在提示词的开头，这样可以提高模型对关键信息的关注度和处理能力。
- 分块摘要：对于超级长的 Prompt，我们可以将其分成多个块，然后对每个块进行摘要。这样可以减少模型对信息的处理负担，提高模型的输出质量。一般以 32k Token 为界限进行分块，可以利用模型对每一块进行适当的摘要然后将摘要作为输入，或者在每一块的开头添加摘要信息。这样可以提高模型对信息的处理能力和输出质量。

¹Let's think step by step 也是五个词。真是巧合。

2.3.5 在 Cherry Studio 上使用 LLM

Cherry Studio 是一个 LLM 管理器，方便我们使用 LLM。它提供了一个简单易用的界面，可以让我们轻松地使用 LLM 进行各种任务。Cherry Studio 支持多种 LLM，包括 ChatGPT、Claude、Gemini 等。它还提供了一个强大的提示词编辑器，可以帮助我们优化提示词，提高输出质量。

为了使用 Cherry Studio，我们应当持有一个 API Key。API Key 通常需要在模型厂商处购买或者申请才能获得。在获得 API Key 之后，我们可以在 Cherry Studio 的设置界面下，找到 Key 对应的模型，然后启用模型并输入 Key 即可。Cherry Studio 默认启用硅基流动的模型，但是并不包含 Key，因此如果我们不使用硅基流动模型则可以将其关闭。

Cherry Studio 界面的左侧是 Agent 列表，可以使用其提供的 Agent 模板来快速构建自己的 Agent 并使用。在 Cherry Studio 中，Agent 的 System Prompt 对用户可见且可以修改，我们可以在这里添加自己的 RTCP 类提示词（你是一袋猫粮...）。右侧则和网页版的 LLM 界面极其相似。

2.4 提问的艺术

当上述方法全部失败的时候，我们还有最后一个方法：可以抱大佬大腿，或者说向有经验的前辈提问和讨教。

除了抱身边大佬大腿以外，一个最传统的方式是，你可以在上述提到的平台或者其他技术社群上提问相关内容。你可以得到来自不同人的回答，这样你就有概率能够得到更多的帮助。当然，收集到的信息也相对良莠不齐，信息的价值需要自行甄别。同时，你的贴子和问答也会被其他人看到，一定程度上也可造福后人。例如，你可以在 Stack Overflow 上提出相关技术问题。

另一个方法是在 GitHub 上发布相关的 Issue，这样项目的维护者就会看到你的问题，并提出相关的解答；有时候也有可能是项目本身的问题。这也能够帮助到以后的用户。

在提问的时候，应该遵照以下的原则：

- 礼貌与尊重：没有人有义务解答你的问题，解决问题也许会耗费不少的时间和精力，大多数人回答问题往往只是出于本能的善意。礼貌的表达不仅能促使他人更愿意帮助你，还能建立良好的沟通氛围。当下互联网环境下，其实这一点的重要性远超想象。
- 增加有用信息：缺乏相关信息会让帮助你的人有心无力。程序崩溃有许多可能情况，不同的情况往往对应着不同的解决方案。如果能够在问题描述中增添足够的有用信息（例如列出错误代码），就会为解决问题增添巨大的可能性。
- 减少无用信息：部分人在提问的时候总会无意识地强调与问题无关的东西。这种内容往往会显著地降低信息密度，招致人的反感与厌恶。一个更常见的例子是在社群中发送大段语音而不是文字。
- 明确化你的描述：有时，我们的描述会出现歧义或者不明确的现象，例如“直面天命”

这个短语对于没有关注或者没有游玩过《黑神话·悟空》的人而言容易导致迷惑。在这种情况下，使用更为具体的“游玩《黑神话·悟空》”等称谓更加合适。

- 列出你失败的尝试：这不仅表现出你为了自己解决自己遇到的问题所付出的努力，也能够显著地减少重复劳动与受到类似 STFW 等回复。

一个较好的提问例子是：

“我的电脑突然蓝屏了，我的蓝屏时候遇到的代码是 XXXXXXXX，是在游玩《黑神话·悟空》的时候突然蓝屏的。我上网搜索了代码相关的错误信息，尝试了网上可能有用的 A 方法和 B 方法，但都没有奏效。能麻烦你帮我看看吗？拜托了，非常感谢！”

第三章 初步使用计算机

我们已经讲过了计算机的基本构成和工作原理，现在我们来讲一些计算机的初步使用方法。

当然，我们不会涉及到计算机的使用细节，例如如何使用鼠标、如何使用键盘等。我们假设同学们已经具备了最基本的计算机操作能力。这里将会介绍一些科研学习中使用计算机的小技巧，以及一些常用的软件和工具。

3.1 维护你的系统

计算机的维护是一个非常重要的环节。我们需要定期对计算机进行清理和维护，以保证计算机的正常运行。

3.1.1 保持更新系统的习惯

计算机在运行过程中，操作系统和软件会不断地更新，以修复漏洞、提高性能和增加新功能。我们应该定期检查系统和软件的更新，并及时按需要安装它们。

对于一些重要的更新（例如安全更新等），我们应该立即安装，这是因为此类更新通常是为了修复一些新近发现的漏洞和问题，如果不及时安装，可能会导致计算机被攻击或者出现其他问题。而对于一些不重要的更新（例如功能更新等），我们可以根据自己的需要选择安装。

特别注意：虽然我们提倡保持更新，但是在生产类环境中，贸然更新可能会导致系统不稳定或软件不兼容。因此，在生产环境中，我们应该在更新之前进行充分的测试，确保更新不会影响系统的正常运行，或者使用虚拟机等隔离环境运行生产用代码。

3.1.2 定期备份数据

定期备份数据是保护计算机数据安全的重要措施。我们可以使用外部硬盘、云存储等方式备份数据，以防止信息泄露或者重要文件丢失。数据备份的频率可以根据数据的重要性和变化频率来决定。

数据备份有一个重要的原则：**3-2-1 备份法则**。即：至少保留三份数据备份，存储在两个不同的介质上，其中一份存储在异地。例如，我们可以在本地硬盘上存储一份数据备份，在外部硬盘上存储一份数据备份，并将另一份数据备份存储在云端。这样，即使其中一份甚至两份损坏或者丢失，我们也可以通过其他方式恢复数据。

3.1.3 定期清理系统

定期清理系统可以提高计算机的性能和安全性。我们可以使用一些系统清理工具，删除不必要的文件、缓存和临时文件等，以释放磁盘空间和提高系统性能。我们推荐使用系统自带的清理工具，例如 Windows 的磁盘清理工具、macOS 的存储管理工具等。如果较为富裕，也可以使用一些知名的清理软件，例如 CCleaner 等（免费版已经足够好用了）。出于众所周知的原因，我们不推荐使用 360 等软件。

除此之外，休眠文件、系统还原点等也会占用大量磁盘空间。我们可以根据自己的需要，选择是否保留这些文件。

特别注意的是，清理系统和减肥差不多，同样需要缓慢、谨慎、循序渐进地进行。

3.1.4 碎片整理

在计算机使用过程中，如果使用机械硬盘，文件的删除和修改会导致磁盘上的数据变得零散，从而影响计算机的性能。我们可以使用碎片整理工具，定期对磁盘进行碎片整理（即重排文件使其连续），以部分提高磁盘的读写速度。

直接使用 Windows 自带的碎片整理工具即可。对于固态硬盘，碎片整理并不会提高性能，反而会缩短使用寿命，因此不建议对 SSD 进行碎片整理。

3.2 善用快捷键

使用快捷键可以减少鼠标操作的频率。以下是来自 Windows 的常用快捷键：

- Ctrl+C：复制
- Ctrl+V：粘贴
- Ctrl+Z：撤销
- Ctrl+Y：重做
- Ctrl+A：全选
- Ctrl+Alt+Del：任务管理器
- Alt+F4：关闭窗口
- Win+R：打开运行窗口
- Win+E：文件资源管理器¹
- Win+D：显示桌面
- Win+L：锁定计算机
- Ctrl+S：保存
- Ctrl+P：打印
- Ctrl+F：查找替换

¹文件资源管理器是 Windows 系统中用于浏览和管理文件和文件夹的工具。

3.3 终端初步

终端是计算机与操作系统之间的一个交互界面。它允许用户通过命令行输入指令，与操作系统进行交互。

虽然终端是一个非常古老的工具，但是使用它依然可以提高工作效率，尤其是在处理大量文件或者进行复杂操作时。它还可以用于远程连接到其他计算机，进行远程管理和维护等操作。

一般情况下，一条命令满足以下结构：

```
<命令> [<选项>] [<参数>]
```

其中，命令是要执行的操作，选项是对命令的修改（例如 `-h` 表示帮助信息），参数是命令的输入（例如文件名）。选项和参数往往都是可选的。

对于 Linux 和 macOS，我们建议使用 Zsh 或 Fish 等更加友好的 shell，他们能够进行语法高亮、自动补全等操作。对于 Windows，我们建议使用 Windows PowerShell。PowerShell 的命令统一采用的是动词-名词的格式，和 Linux Shell 的简单缩写形式有很大的不同。这是因为 PowerShell 的设计理念是模仿 C# 的“对象”，而不是 Linux Shell 的文本流。不过也正因此，PowerShell 本身就是一门完备的语言，功能非常强大，在处理复杂的任务上更为简单。表 3.1 是一些常见的命令。

操作	Bash	Pwsh
创建文件	<code>touch</code>	<code>New-Item</code>
列出文件	<code>ls</code>	<code>Get-ChildItem</code>
复制文件	<code>cp</code>	<code>Copy-Item</code>
移动文件	<code>mv</code>	<code>Move-Item</code>
删除文件	<code>rm</code>	<code>Remove-Item</code>
创建目录	<code>mkdir</code>	<code>New-Item -Type Directory</code>
删除目录	<code>rmdir</code>	<code>Remove-Item -Recurse</code>
查看帮助	<code>man</code>	<code>Get-Help</code>

表 3.1: Bash 和 PowerShell 的常用命令对比

这仅仅是最基本的命令使用方法，实际上终端的命令系统非常复杂，功能也非常强大。感兴趣的同学可以自行查找相关资料进一步学习。

3.4 包管理器

包管理器是用于管理软件包的工具。它可以自动下载、安装、升级和卸载软件包，并且可以解决软件包之间的依赖关系。

在 Linux 和 macOS 中，包管理器是非常常用的工具。它可以帮助用户快速安装和管理软件包，避免手动下载和安装软件包的麻烦。比如我希望在 Arch Linux 下安装 GCC，只需要执行以下命令即可：

```
sudo pacman -S gcc
```

在 Windows 中，包管理器的使用不普遍。虽然有一个官方的包管理器 winget，但是支持的软件包较少，且无法自动管理依赖（但也基本够用）；还有一些例如 Chocolatey、Scoop 等第三方包管理器。除此以外，使用 MSYS2、Cygwin 等类 Unix 环境也可以从某种程度上当成包管理器使用。例如后文讲的安装 GCC 的过程，我们就是使用 MSYS2 来安装的，比下载预编译版本简单许多。

3.5 Git 初步

试想以下环境：我们正在写一项作业，开发工作已经基本完成，试运行也能够得到 90 分。此时我们希望进一步精进代码，使得分数达到 95 分以上；但是经过一通修改以后，发现程序再也运行不起来了。这时候距离 ddl 只有 1 小时，我们决定摆烂，提交能够得到 90 分的代码。然后我们根据记忆改回原来的代码的时候，发现我们再也想不起来旧代码是怎么写的了！这无疑令人极为懊恼的。

为了避免以上问题，我们引入了版本控制系统。目前最常用的版本控制系统是 Git。

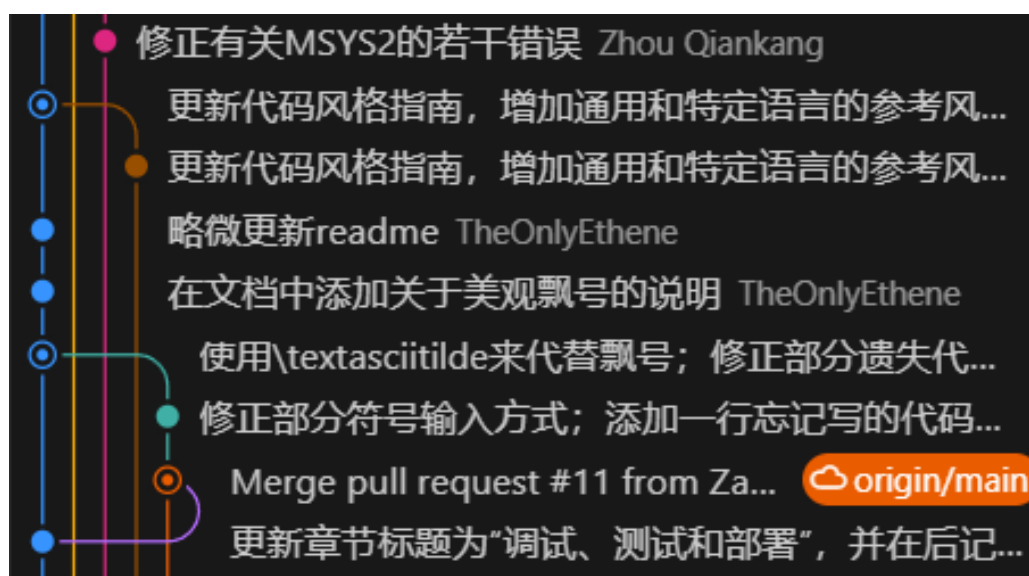


图 3.1: 一个典型的 Git 工作流程

3.5.1 Git 的工作原理

Git 有三个目录共同完成版本控制：工作区、暂存区、版本库。工作区是项目目录，暂存区是一个隐藏的文件夹.git，版本库是一个隐藏的文件夹.git/objects。工作区是我们平时使用的目录，暂存区是 Git 用来存储修改的地方，版本库是 Git 用来存储所有版本信息的地方。版本库有一个指针，指向当前版本的某一节点（一般指向最新的节点）。每个节点都有一个唯一的

哈希值，用来标识该节点。每个节点包含了该版本的所有文件和目录的信息，以及指向上一个版本的指针。Git 使用哈希值来标识每个版本，这样可以保证每个版本都是唯一的。

这样讲解很难以理解，我们不妨举例说明：现在，Git 中有一个版本为 X 的节点，包括文件 A 和文件 B 两个文件。这些文件存储在版本库中。此时，工作区为空，暂存区为空，指针指向 X。我现在希望对它们进行修改，这个修改遵循以下过程：

1. 我拿出了这些文件，并且对文件 A 进行修改。此时，工作区有 AB 两个文件，但是暂存区依然是空的。我们的任何修改都不会被暂存区记录，Git 也不会知道我对这些文件进行了修改。
2. 我觉得修改差不多了，现在把 A 放进暂存区。现在 Git 知道我对 A 进行了一些修改了。
3. 我又对 B 进行了类似的修改，此时 B 也进暂存区了。
4. 我觉得修改差不多了。我认为我应该永久保存目前的状态，于是就把暂存区提交到版本库。此时版本库多了一个 Y 节点，指针也指向 Y 节点，有修改过的 AB 两个文件。此时，暂存区又清空了，而工作区和版本库的 Y 版本一致。

3.5.2 下载 Git

一个最简单的方式是使用 Winget 包管理器：

```
winget install Microsoft.git
```

或者你也可以从官方网站上下载并安装之。同样，安装的时候一定要勾选“添加到 PATH”这一选项，否则你在命令行中无法使用 Git。

3.5.3 Git 信息设置

安装并使用 Git 的第一步是先编辑本地的一些信息。Git 的提交需要一个用户名和一个邮箱，来对应每次提交的作者。我们可以使用以下命令来设置这些信息：

```
git config --global user.name "Your_Name"  
git config --global user.email "email@example.com"
```

这样即可设置全局用户名和邮箱。如希望给某个特定仓库设置特定的用户名和邮箱，你需要在该仓库下重新执行上述命令，但是不写 `--global` 命令。

现代 Git 一般提倡使用 `main` 作为根分支的名称。而 Git 依然使用旧的 `master` 分支作为根分支，你可以使用以下命令修改为 `main`：

```
git config --global init.defaultBranch main  
# 这条命令会修改全局的默认分支名称
```

3.5.4 Git 的最基本使用

3.5.4.1 提交

要具体地在某一目录下进行版本控制，我们需要在命令行中进入到我们希望使用 Git 的目录下。然后我们可以使用以下命令来初始化一个 Git 仓库：

```
git init
```

如果你在视窗中开启了“显示隐藏文件”这类功能，你就会发现一个隐藏的文件夹.git 出现在了您当前的目录下。这个文件夹就是 Git 用来存储版本信息的地方。

然后你可以使用以下命令来添加文件到 Git 仓库中（这个命令的实际意义是把文件添加到暂存区）：

```
git add <filename>
```

如果我们忘记了当前状态下有哪些文件被修改了，我们可以使用以下命令来查看当前状态：

```
git status
```

如果你觉得修改差不多了，保存文件以后，你可以使用以下命令来提交文件到 Git 仓库中（这个命令的实际意义是把暂存区的文件提交到版本库中）：

```
git commit -m "commit_message"
```

上述内容中，-m 后面是提交信息。提交信息是对本次提交的简要描述。我们建议每次提交都写上简要的提交信息，这样可以帮助我们更好地理解代码的修改历史。

3.5.4.2 回退

如果出现了先前我们说的不小心写坏了的情况，这时候就可以进行版本回退了。我们可以使用以下命令来查看当前的版本信息：

```
git log # 例如版本库是a-b-c-d-e-f-g
```

找到你希望回退到的版本的哈希值（前几位即可），然后使用以下命令来回退到该版本（这个命令会把指针回退到指定的版本，丢弃之后的所有内容，然后丢弃暂存区和工作区的所有东西）：

```
git reset --hard <commit_hash>
```

请谨慎使用这一命令！该命令不会保留当前的修改！

如果你希望回退到某个版本，但是不想丢失当前的修改，你可以使用以下命令来回退到该版本（这个命令会把版本库后面的东西全部丢弃，清空暂存区，但是保留当前工作区）：

```
git reset --mixed <commit_hash>
```

我们更加推荐这个回退方式，`--mixed`可以省略，或者用`--soft`替代。

用`--soft`替代时，不会清空暂存区。

使用图解来表示一下：可以看到，回退操作虽然会把指针回退到指定的版本并丢弃之后

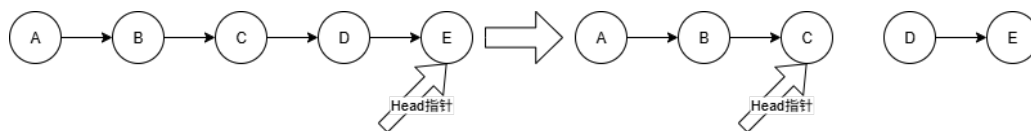


图 3.2: Git 的回退操作

的版本，但是之后的版本提交依然存在于版本库中，只是被从树上摘下来了。这些提交被称为“孤立提交”。如果希望恢复或者删除这些孤立提交，可以执行以下命令：

```
git fsck --lost-found # 查看孤立提交、孤立分支等
```

```
git checkout <commit_hash> # 进入分离头模式
```

```
git branch <branch_name> # 创建一个分支来恢复孤立提交
```

```
git gc --prune=now # 清理孤立提交
```

即使我们不使用 `git gc` 手动清理孤立提交，随着时间的推移（一般是 90 天提交记录过期），孤立提交也会被 Git 逐渐自动清理掉。

3.5.4.3 排除相关文件

有时候我们版本跟踪的时候不需要跟踪一些文件，例如具有敏感信息的文件（如密码），或者构建文件等。此时，我们可以创建一个文件 `.gitignore` 来阻止跟踪。例如，在 Linux 下，构建文件往往是 `*.o`。那么我们可以在上述文件中加入 `*.o`，之后 `git` 就会忽略这些文件。

关于 Git 版本控制的一些更加进阶的知识（例如分支管理等内容），欢迎查阅更多资料。我们在高阶课程 7.1 中会介绍一些 Git 的进阶用法。

第四章 购买计算机

本章可能含有消费建议，可能已经过时。

如你所见，本章将介绍如何购买一台计算机。

与其他章节不同，这一章我想了很久很久才动笔，因为我知道，计算机的购买涉及到很多方面的知识和经验。同时，我也有可能因为个人的偏见和不充分的经验，给出一些不够全面或不够客观的建议，因此引发部分人的反感。所以，我预先在此声明：**本章内容仅供参考。**主观上本章并不含有也不会含有任何商业广告。

4.1 获取计算机的途径

一般情况下，我们有两种情况获取一台计算机：要么直接购买整机（笔记本或者整机台式机），要么购买零部件组装一台计算机（往往是台式机）。前者的优点是简单方便，缺点是性价比低；后者的优点是性价比高且高度可定制，缺点是需要一定的组装技术与经验，并对计算机的基本硬件知识有一定的了解。

其实希望购买零配件组装整机的人往往是对计算机性能提出更高要求的人，尤其是在游戏、图形设计、视频编辑等领域。因此，实际上组装机往往确实价格上更贵。

4.2 买机器的原则

- **买新不买旧：**计算机的更新换代非常快，旧机器的性能往往无法满足新软件的需求，甚至可能无法运行新操作系统。旧机器的硬件也可能存在兼容性问题，导致无法使用最新的软件和驱动程序。一般只考虑近两年上架的产品。
- **确定型号和参数：**在购买之前，我们非常建议先确定好型号和参数，不能使用任何描述性的语言来作为购买依据，例如“Intel 12 代高性能处理器”是描述性的，实际上对应的型号可能是 i7-12700H 或者 N5095，这两个玩意性能差距有七倍。

4.2.1 笔记本电脑的简单分类

笔记本电脑可以简单分类为以下几类：

- **轻薄本：**轻薄本是指重量轻、厚度薄的笔记本电脑，通常用于日常办公、学习和娱乐。它们通常配备低功耗处理器，续航时间较长，但性能相对较弱。
- **游戏本：**游戏本是指专门为游戏设计的笔记本电脑，通常配备高性能处理器和独立显卡，能够运行大型游戏和图形密集型应用。它们通常较重，续航时间较短，但性能强大。很多人反映，背着这玩意去教室困难，请谨慎选择。

- **全能本**：全能本是指兼顾轻薄和性能的笔记本电脑，通常配备中高功耗处理器和独立显卡，能够满足日常办公、学习和娱乐的需求，同时也具备一定的游戏性能。

4.2.1.1 MacBook 系列

由于苹果系列产品的特殊性，这里单独介绍。

MacBook 在保持了轻薄的同时，拥有动辄十小时的超长续航和较强的性能。在习惯了 macOS 操作逻辑之后，使用 MacBook 会获得极流畅舒服的体验，如果你恰好拥有其他苹果设备构成生态，也会使得工作效率大幅提升。作为类 Unix 系统，macOS 在编程开发时配置环境较为容易。对于音视频编辑的工作，MacBook 也有较大优势。

对于学生党来说，苹果最大的缺点其实是贵，同时，游戏体验一般。且少数特定软件在苹果的 macOS 上支持不佳，因此选购 MacBook 前务必向学长学姐打听好软件支持。就统计来看，各种专业的绝大多数必备软件都是支持的，个别研究方向可能出现此问题。就信科来看，计算机专业的软件几乎都有 macOS 支持，电子专业则有部分软件不能运行在 MacBook 上。

推荐选择苹果官网作为 Macbook 的购买渠道，除了可以自定义配置以外，进行学生认证后可以获得优惠、并有礼品赠送（通常是耳机），价格几乎是全网最低。而且，即使是拆封激活后也可以七天无条件免费退货，有购物保障。

选购 Macbook 时主要的定制参数就是内存和硬盘，这里就涉及到苹果最大的问题：内存和硬盘很贵，由于内存和硬盘都无法扩展，建议至少选配 16GB 统一内存和 512GB 固态硬盘。如果提高配置之后预算超过上限，建议选购 Windows 本。

同时，M1、M2 芯片的 MacBook Air/Pro 均仅支持至多一块外接显示器，只有 M1 Pro/Max、M2 Pro/Max 才支持多块屏幕，如有相关需求需要在选购时注意。

4.2.2 奸商常见套路

奸商常见套路有以下几种：

- **模糊配置**：没有写明具体配置，尤其是采用“描述性语言”而不是具体型号，消费者完全无法判断性能与实际价格。
- **偷梁换柱**：跟你说是配置 A 的电脑，实际卖给你的是配置 B 的电脑，消费者不知不觉就上了套。
- **突然缺货**：等你咨询完准备下单之后，突然跟你说你要的那款没货，然后让你换成所谓同等价位实际却差很远的电脑。

4.2.3 专业测评

这里为大家推荐一个较为专业客观的公众号：**笔吧测评室**。

4.3 整机

4.3.1 品牌的选择

购买整机首先要面对的是品牌。联想、戴尔、惠普、华硕、宏碁、苹果、微软 Surface、华为、小米、荣耀、雷蛇、微星、技嘉、神舟、机械革命、机械师、雷神、炫龙、火影、吾空、未来人类……名字多得像超市货架上的零食。

一般有以下两个思路：

- **只看御三/四家：**联想、戴尔、惠普。它们的产品线丰富，覆盖了从入门级到高端级的各个价位段，售后服务也相对完善。以上三家市场占有率高，售后网点多，配套驱动更新及时且长期维护，虽然贵一些，但是适合不想折腾的人。苹果近年来逐渐加入了这个行列，虽然价格更贵且 macOS 兼容性较差，但是对于设计师和创作者来说，macOS 是一个很大的优势。
- **只看性价比：**神舟、机械革命、火影、吾空、未来人类，同配置常常比御三家便宜一两千，但售后依赖返厂，品控如同抽盲盒。

4.3.2 渠道：线上和线下

4.3.2.1 线上

线上购买渠道不少，主要有这四种：京东、天猫、官网、拼多多百亿补贴。一般认为，京东自营大于天猫旗舰店，约等于官网，大于拼多多百亿补贴。

百亿补贴便宜是真便宜，翻车是真翻车，水深得很。要是贸然入坑，一定要做好功课，到货以后也要录开箱视频 + 查 SN+ 七天无理由退货。

4.3.2.2 线下

如果你确实需要这本手册，那么我非常不建议你去线下任何门店购买任何计算机！

线下主要有品牌直营店、授权专卖店、电脑城、商超等。一般前两个渠道售后服务较好，但是价格往往比线上购买高一些，好处是能够当场验机并激活常用软件等。

电脑城水最深，包括并不限于转型机、展示机、矿机翻新、贴标内存，防不胜防。新手极不建议去趟浑水（可以去试试手感，但是不要买；熟人带路也不能买，坑的就是熟人！）。

4.3.3 验机、保修

无论线上还是线下，拿到机器后务必“开箱-插电-不联网”三步走：

- **开箱：**检查外包装是否完好，是否有明显的磕碰、划痕等。

- 插电：插上电源，检查电源适配器是否正常工作，电源指示灯是否亮起，电池是否充电。不要先进系统，进 BIOS 检查硬盘通电次数和电池循环次数，一般小于十次是可以接受的。
- 不联网：不要联网，也不要激活系统和软件；先检查屏幕是否有坏点、漏光、色偏等问题。可以使用 U 盘拷入 DisplayX、AIDA64 等工具进行检测。有部分品牌在激活 Office 或联网后就不给无理由退换了，所以尽可能检查清楚。

Win11 新机器默认情况下需要注册微软账户，不联网无法完成系统初始化。这时，可以在该初始化界面按下 *Shift+F10* 打开命令行，输入 *OOBE\BYPASSNRO*，然后重启电脑。这样就可以跳过联网注册微软账户的步骤。

发现问题尽早退换货，防止不必要的麻烦。

保修政策要看清：

- 全国联保 \neq 全球联保。留学生买美版 ThinkPad 回国，坏了要送香港修。
- 上门服务 \neq 免费上门服务。戴尔 ProSupport 可以第二天上门，但那是你多花 800 块买的。
- 意外险 \neq 全保。液体泼溅、跌落、电涌，有的意外险只赔一次，第二次自费。

4.4 组装机

组装机小白有一个误区：先选 CPU 再选别的。这并不合适。因为 CPU 的选择会影响主板的选择，而主板又会影响内存、显卡等其他部件的选择。因此，建议先确定自己的需求，再根据需求来选择合适的 CPU、主板、内存、显卡等。

正确的顺序是：先定需求，再定预算；先买核心配件（CPU、主板、内存、显卡），再买其他设备。

定需求：

- 办公学习：i3-14100 / R5-5600G + 核显。价格约 5000 元。
- 主流网游 / 轻度剪辑：i5-14400F / R5-7500F + RTX 4060 / RX 7600。价格约 8000 元。
- 2K 高刷 / 3A 大作：i7-14700KF / R7-7800X3D + RTX 4070 Ti SUPER。价格约 12000 元。
- 生产力 / 4K 剪辑 / AI：i9-14900K / R9-7950X3D + RTX 4080 SUPER 或 Ada 工作站卡。价格约 20000 元。

低于 5000 元的配置不建议购买组装机，建议用这个钱购买整机，需求肯定够了。

除了以上推荐的配置以外，我们还有其他需要注意的地方，例如需要做数据处理的同学们应该需要巨大的内存等。

然后定机箱体积，这个事实上决定了你机器的占地体积。这需要根据自己的桌子空间来定：全塔、中塔、MATX、ITX。ITX 溢价高，风道难做，慎入。

4.4.1 核心配件

4.4.1.1 板 U 套装

买板 U 套装一般便宜，且主板和 CPU 的兼容性有保障。如果不买板 U 套装，可以买散片 CPU，比盒装便宜得多。当然，散片风险也大一些。

目前电脑主流 CPU 主要有 Intel 和 AMD 两种，当前两者芯片势均力敌，都可以放心选择。Intel 名声较大，多年以来在市场占有率上有着巨大的优势；AMD 在 2017 年提出新架构以后，性能突飞猛进，“AMD 性能差”已成为错误认知。

- Intel 14 代：接口 LGA 1700；主板 B660/B670/Z790。注意区分内存插槽类型，DDR4 和 DDR5 的主板不兼容。
- AMD 7000 系列：接口 AM5；主板 B650/A620/X670。A620 不超频，适合游戏党。
- 老东西：Intel 12 i5-12490F + B660 仍然是性价比之王。

4.4.1.2 显卡

2025 的显卡市场依然混乱，预算充足的情况下无脑英伟达 40 系列即可。AMD 性价比高得多，但是光追和生产力依然不如英伟达。Intel Arc 驱动终于正常了，剪视频很不错，游戏党建议观望。

避坑：二手显卡风险巨大，30 系默认矿卡，40 系默认锁算力。（除非你认识买家！）另一方面，淘宝上所谓的“电竞显卡”店往往是丐版贴牌，慎入。

4.4.1.3 内存和存储

内存方面，DDR4 和 DDR5 的主板不兼容，必须注意。

25 年一般喜欢 DDR5，盯着 32GB 6000MHz CL30 的套装买就可以了。英特尔 14 代可以冲 7200MHz。（其实能做到这么好的厂家也就海力士了）

DDR4 已经白菜价了，对于老东西而言性价比超级高。

存储方面，SSD 的速度和容量是关键。QLC 便宜，但是掉速严重。系统盘建议 TLC，仓库盘可以选择 QLC。机械硬盘除非需要超大容量（4TB 以上），否则不建议买，一定要买的话盯着西数和希捷就行。叠瓦盘别碰。

4.4.2 其他配件

作者在这里踩过坑，总结出来的经验是：一定不能因为是非核心配件就只图便宜，盯着大牌子买总没问题。

4.4.2.1 电源

首先务必确定以上你买了些什么。一般的，瓦数至少是核心配件瓦数的 1.5 倍再加 100 瓦。盯着金牌全模组电源买就行，航嘉、长城、海韵、振华都可以。

4.4.2.2 散热

i5/R5 单塔 4 热管（利民 PA120）即可；i7/R7 双塔 6 热管（利民 FC140）；i9/R9 360 水冷起步（瓦尔基里 A360）。

4.4.2.3 机箱

这个没什么可说的，一般对于高性能计算机，前进风至少 3 风扇，后出风至少 1 风扇，侧面风扇和顶风扇可选。玻璃侧透谨慎购买，尤其是闷罐，可能导致箱内温度极高。

兼容性方面，显卡长度、CPU 散热器高度、主板尺寸（ATX/MATX/ITX）、电源长度等都要考虑。

4.4.3 显示器

显示器是一种输入输出设备，按理说应该属于“其他配件”，但是我还是单独把他拿出来讲了。这是因为，显示器对使用者而言完全就是整个计算机中最重要的部分之一，毕竟这玩意能真正直接影响到你的所有使用体验：一个糟糕的显示器完全能够抵消一块 4090 带来的快乐！

4.4.3.1 尺寸与分辨率

一般来说，显示器的尺寸以对角线长度来表示，单位是英寸。分辨率则是指屏幕上像素点的数量，常见的有 1080p、2K、4K 等。为了定义屏幕的清晰度，我们引入一个概念：PPI（Pixels Per Inch），即每英寸的像素点数。PPI 越高，屏幕越清晰。具体的公式这里不啰嗦了，我们可以使用这个[工具](#)来计算 PPI。

对角线	分辨率	近似 PPI	推荐视距	典型用途
24 英寸	1920×1080	92	60-70 cm	办公、网课、MOBA
27 英寸	2560×1440	109	65-75 cm	全能甜点
27 英寸	3840×2160	163	55-65 cm	代码、设计、4K 影音
32 英寸	3840×2160	138	70-80 cm	剪辑、影视后期
34 英寸	3440×1440	110	65-75 cm	带鱼屏游戏、股票

经验法则：1080p 别超过 24 英寸，否则 PPI 太低，内容将非常模糊，对眼睛完全就是一种折磨。27 英寸屏幕起步 2K；4K 最好 27-32 英寸，否则缩放比例尴尬。带鱼屏（21:9）优先 3440×1440，2560×1080 纵向 PPI 太低。

4.4.3.2 刷新率

刷新率指的是显示器每秒钟能够更新的画面数量，单位是赫兹（Hz）。一般的，人类看到的显示屏至少得 90Hz 以上，看起来才足够舒服。

- 60 Hz：办公、影音足够，不过现在已经很低端了。
- 75-100 Hz：轻度电竞、日常使用，预算友好，且非常流畅。
- 144-165 Hz：FPS 玩家黄金档，显卡吃到 RTX 4060 以上即可跑满。
- 240 Hz 及以上：CS2、APEX 职业选手专属，钱包与显卡双重考验。

注意：高刷必须搭配 DP1.4 或 HDMI 2.1 线，否则 1080p 240 Hz 只能缩到 144 Hz。

4.4.3.3 面板技术

面板技术指的是显示器使用什么类型的材料进行显示。主要有以下几种，其中 IPS、VA、TN 是液晶面板技术，OLED 是有机发光二极管技术，Mini-LED 是一种新型的背光技术。

面板	优点	缺点	适合人群
IPS	颜色准、可视角度大	普遍漏光	设计、办公、全能
Fast IPS	1 ms GTG、高刷	对比度一般	电竞、兼顾创作
VA	高对比度、色彩艳	响应慢、拖影	影音、单机 3A
TN	极快响应、便宜	可视角度渣	纯 FPS 硬核玩家
OLED	无限对比、极快响应	烧屏风险、贵	影音发烧、HDR 游戏
Mini-LED	高亮度、多分区背光	光晕、价高	HDR 剪辑、次旗舰电竞

避坑提醒：

“IPS 级别” ≠ IPS，可能是廉价 AHVA。

VA 曲面屏 27 英寸以下意义不大，32 英寸以上才显沉浸。

OLED 桌面静态 UI 易烧屏，建议隐藏任务栏 + 黑色壁纸。

4.4.3.4 色彩与亮度

色域、色准一般用户基本上不需要考虑。对于板绘画师、视频剪辑师等专业用户来说，色域和色准则是非常重要的，这边更推荐有相关爱好的同学们咨询业内大佬，我就不在这里班门弄斧了。

亮度比较重要，尤其是对画面有追求的用户而言，带 HDR 的显示器几乎是必备。一般 SDR 250 nit 起步，HDR400 认证只是“能亮”；真想 HDR 观影选 HDR600 以上 + 分区背光或 OLED。

4.4.3.5 接口与线材

- DP1.4：现代主流接口，支持 2K 165 Hz / 4K 144 Hz 10bit 无压缩。
- HDMI 2.1：主机党接 PS5/XSX 4K 120 Hz 必须。

- Type-C: 65-90 W 反向供电 + DP Alt-mode, 笔记本外接一条线搞定。
- USB-B 上行口: 老式 KVM 或显示器集成 USB Hub 时才会用到。

线材别图便宜:

2K 165 Hz 以上务必买 VESA 认证 DP1.4 线 (10-20 元杂线会黑屏); HDMI 2.1 认准超高速认证 (48 Gbps); Type-C 线看 E-Marker 芯片, 标 100 W 却只支持 60 Hz 的比比皆是。

4.4.3.6 验屏

收到屏幕之后, 建议对屏幕进行以下检查以规避问题。坏点指的是显示器上无法显示的像素点, 漏光指的是屏幕边缘或角落出现的光线泄漏现象。对于需求较高的用户而言, 也请验证刷新率和色域。

- 坏点: 使用纯色图片, 红绿蓝白黑共五张, 肉眼距离 50 cm 观察。国标允许 3 个以内坏点, 超过即退换。
- 漏光: 全黑图, 手机夜景模式拍照, 四角漏光若呈“黄雾”可接受, “白雾”则太严重。
- 刷新率、色域: UFO Test 网站跑 144/165/240 Hz, 看是否掉帧; DisplayCAL 校色仪验证色域覆盖与 ΔE 。

注意: 长时间盯着 UFO Test 网站可能导致眩晕, 建议晕车的同学谨慎使用。

至此, 显示器选购的坑与雷已悉数奉上。记住一句话: 屏幕是你每天盯得最久的部件, 预算再紧也别在这上面省过头。

4.5 捡垃圾

我们非常不推荐购买任何二手计算机, 因为二手计算机的风险极大。如果确实预算有限, 或者仅仅是想要体验组装电脑的乐趣, 可以考虑捡垃圾 (二手)。捡垃圾有两种方式: 一是从身边的朋友那里获取二手设备, 二是从网上的二手市场获取。对于这一方面, 我建议同学们查看较早一些的教程。

捡垃圾也有一定的底线:

- 电源永远不捡垃圾: 电源一炸, 整个机器全完。
- 机械硬盘永远不捡垃圾: SMART 重映射扇区大于 100 可以直接扔了。
- 当面交易优先: 二手市场水极深, CPU-Z、GPU-Z、HWiNFO、CrystalDiskInfo 等工具都要用起来。

自此, 购买计算机的方法和注意事项已经介绍完毕。当然, 在你的大学四年之间, 计算机的更新换代会极为迅速; 而且真买了一台超高配置的笔记本电脑也很难保证能够使用四年: 即使是没有出现故障, 笔记本电脑的性能也会因为电池老化、硅脂干涸等问题而逐渐下降; 笔记本电脑的维修和养护也是一个很大的难题。台式机可能会好一些, 至少能够方便地更换零部件, 但是也需要定期清灰、保养等。

希望同学们都能够买到一台称心如意的计算机, 享受计算机带来的乐趣和便利。

第二部分

大学计算机前置

第五章 正式踏入编程世界

对于计算机小白而言，编程的世界可能会显得有些陌生和复杂。要想在这个世界中游刃有余，我们需要掌握的两个重要要素是**工具**和**环境**。

工具，指的是我们用来编写和运行程序的东西。包括语言、编辑器等。它们帮助我们更高效地达成我们的目标，例如调试程序、管理项目等。

环境，指的是我们编写和运行代码所依赖的操作系统、编程语言版本以及相关的库和框架。一个好的编程环境可以大大提高我们的工作效率；同时，我们写出的代码，最终也是要运行在某个环境中的。编程语言本身就是为了让我们能够更方便地与计算机进行交流。

5.1 编程语言初探

5.1.1 编程语言的发展简史

编程语言的发展经历了几个重要阶段：

- **机器语言**：最早的编程语言，直接与计算机硬件对应，使用二进制代码。最早的机器语言是通过拔插电缆实现的，是一个体力活，非常不便。后来改为采用打孔纸带的形式，但仍然非常繁琐和不直观。同时，对于不同的硬件架构，机器语言也不兼容，导致了可移植性差的问题。
- **汇编语言**：在机器语言基础上发展而来，引入了助记符，使得编程更加人性化。同时，汇编语言与特定的指令集相关联，这大大增强了其可移植性，但仍然需要对硬件有一定的了解。汇编语言虽然比机器语言更易读，但仍然需要手动管理内存和硬件资源。
- **高级语言**：高级语言的出现使得编程过程更像说话，而不是在机器上进行什么精确控制硬件的操作，显著增强了其可维护性和可读性；同时同一个高级语言在不同的硬件平台上只需要在对应系统上有一个编译器或解释器就可以运行，这也大大增强了其可移植性。高级语言可以分为两类：
 - **编译型语言**：如 C 系语言，代码在运行前需要经过编译器转换为机器码，这样可以提高运行效率，但编译过程可能较慢。
 - **解释型语言**：如 Python，代码在运行时由解释器逐行解释执行，虽然运行速度可能较慢，但开发效率更高，调试更方便。

现代的编程语言通常结合了编译型和解释型的优点，提供了更高的抽象层次和更丰富的功能库，使得编程变得更加高效和便捷。微软的.NET 系就是一个典型的例子，它提供了一个统一的编程环境，支持多种语言，并且可以在不同的平台上运行。

5.1.2 编程语言的特点和选择

不同的编程语言往往有着自己的特点，选择合适的语言取决于项目需求和个人偏好。

一般情况下，涉及到底层硬件操作、性能要求高的项目，通常会选择编译型语言，如 C/C++、Rust 等。这些语言提供了对内存和硬件的直接控制，能够实现高效的性能优化；而对于快速开发、原型设计等任务，解释型语言如 Python 或 JavaScript 则更为合适。

除了这些通用性语言，还有一些语言能够对特定内容进行极好的支持，例如 C# 用于游戏开发，LaTeX 用于排版，SQL 用于数据库查询，MatLab 用于科学计算和数据分析，R 用于统计分析等。这些语言通常在特定领域内有着广泛的应用。

不过归根结底，编程语言只是工具。初学者在学习编程的时候，更应该关注的是编程的思想和方法，而不是具体的编程语言。每一门语言都有自己的长处和缺点，在实际使用的时候应该具体情况具体应对。

5.2 编程环境的搭建

编程环境的搭建是编程的第一步。一个好的编程环境可以大大提高我们的工作效率。

在搭建编程环境之前，我们应当先认识“环境变量”，它是操作系统中存储的变量，用于配置程序运行环境。环境变量可以影响程序的行为，例如指定编译器的路径、设置库文件的搜索路径等。

特别注意：系统，尤其是 Windows 系统是一个很玄学的玩意，有时候需要重启计算机（或者终端）才能使环境变量生效，有时候则不需要。

5.2.1 C 系编译器及其环境配置

对于 C 语言，有三个最常见的编译器：GCC、Clang 和 MSVC。它们各有特点，但都能满足大部分初学者的 C 语言开发的需求。这些编译器通常会与特定的 C 标准库实现（如 GNU 的 libstdc++、LLVM 的 libc++ 或 Microsoft 的 MSVC CRT）配合使用，不同的标准库之间存在细微差异。我们一般建议在 Linux 上使用 GCC，而在 Mac 上推荐使用 Clang。MSVC 工具链在 Windows 上非常流行，但是不跨平台且为闭源软件，有部分程序员可能因此不愿意使用。

Linux 和 Mac 用户可以通过包管理器安装特定的编译器，在暑假课程的《包管理器》一节有讲授，这里不再赘述。

对于 Windows 用户，有两种方式获得这一编译器：如使用 MSVC，则直接下载 Visual Studio 并安装 C++ 开发模块即可，Visual Studio 内置了 MSVC 工具链；如果使用 GCC，则需要通过其他渠道。比起手动下载和配置 GCC，我更推荐使用 MSYS2 或者 Cygwin 来安装 GCC。它们提供了一个完整的 Unix 环境，免去了在 Windows 上配置编译器的麻烦。

你需要在 [MSYS2 官网](#) 下载最新的安装包，并按照官网的说明进行安装。安装完成后，你可以通过 MSYS2 的包管理器 pacman 来安装 GCC。

你可以在 MSYS2 终端中运行以下命令来安装 GCC 和 GDB（建议使用 UCRT64 终端，不建议使用逐渐失去支持的 32 位以及 MINGW64 环境，也不建议新手使用 CLANG64 环境，该环境完全使用 Clang 代替 GCC）：

```
pacman -S mingw-w64-ucrt-x86_64-gcc
pacman -S mingw-w64-ucrt-x86_64-gdb
```

在 MSYS2 中安装完成后，用户如果想要在 Windows 终端中使用 GCC，则需要设置环境变量，以便在命令行中直接使用编译器命令。

一般情况下，用户需要将 MSYS2 的 bin 目录添加到系统的 PATH 环境变量中。具体步骤如下：

- 找到 MSYS2 的安装目录，通常是 C:\msys64。
- 将 C:\msys64\ucrt64\bin 添加到系统的 PATH 环境变量中。（请按照你的实际安装路径进行调整，下同）
- 在 PowerShell 或者 CMD 中运行以下命令来验证是否配置成功：gcc --version。如果输出了 GCC 的版本信息，则说明配置成功。
- 需要在 Windows 的 PowerShell 或者 CMD 中运行 POSIX 风格工具时，也可以将下列路径也添加到用户的 PATH 环境变量中：C:\msys64\usr\bin。但这样具有环境冲突风险，需要注意保证该变量的查找顺序在比 ucrt64 的 bin 目录更靠后，以避免冲突。

特别注意：有的同学可能不是按照上述推荐的方式安装 GCC 的，而是通过其他方式（例如直接下载预编译版本）安装的 GCC。如果是这种情况，务必记住 GCC 和非 ASCII 字符是死敌，因此请不要将 GCC 安装在包含非 ASCII 字符（如汉字、空格）的路径下！（最大的坑可能是你的用户名中包含非 ASCII 字符，例如汉字！）

5.2.2 虚拟环境及其配置

开发，尤其是生产类开发有一个重要的原则：不重复造轮子。以 Python 为例，作为一门流行的编程语言，它拥有丰富的第三方库和框架，可以帮助我们快速实现各种功能，并不需要从零开始开发。

因此，第三方库的安装和管理是开发中非常重要的一部分，而不同的开发往往需要不同的包，或者同一个包的不同版本。这些包有可能会产生冲突，如果用全局环境则会导致依赖混乱。这时，我们引入了虚拟环境，它是解决包冲突的有效手段。

虚拟环境可以理解为一个单独的盒子，包含了特定版本的编译器、解释器和所有依赖的包。用户可以在虚拟环境中自由安装和管理包，而不会影响全局的 Python 环境。一般有以下几种方法创建虚拟环境：

- **venv**：Python 内置的虚拟环境模块，适用于大多数场景。
- **virtualenv**：一个第三方库，提供了更多的功能和灵活性。
- **conda**：Anaconda 发行版提供的虚拟环境管理工具，非常简洁高效，逐渐成为目前开发的主流选择。

- **docker**: 容器化技术, 可以将应用及其所有依赖打包在一个容器中, 适用于生产环境, 但是由于比较复杂、笨重、资源开销大, 一般不推荐用于开发环境。

对于学生一般使用的是 `conda`, 或者其衍生高速版本 `mamba`。我们的讲述也是以 `conda` 为主。

`conda` 有两个发行版, 一个是 `Anaconda`, 另一个是 `Miniconda`。`Anaconda` 包含了大量的预装包, 适合初学者和数据科学家使用; 而 `Miniconda` 则是一个轻量级的发行版, 只包含 `conda` 和 `Python`, 适合需要自定义环境的用户。我们非常建议使用 `Miniconda`, 因为它更轻量, 安装速度更快, 并且可以根据需要安装所需的包。

特别注意的是, 以上两个发行版在 `Linux` 和 `Winget` 上都难以利用包管理器进行安装。因此, 我们一般都是直接从官网下载对应的安装包进行安装。安装完成后, 用户需要设置环境变量, 以便在命令行中直接使用 `conda` 命令。

接下来, 因为一些原因, 我们应当重启计算机 (或者终端), 以确保环境变量生效。

然后, 我们需要进行终端初始化, 例如在 `Windows` 上, 我们可以使用以下命令:

```
conda init powershell
```

你应该将上述命令替换为你所使用的终端类型, 例如在 `Linux` 上可以使用 `bash` 或 `zsh`。然后你需要重启终端, 以使初始化生效。

我们可以创建一个新的虚拟环境, 例如:

```
conda create -n myenv python=3.10
```

这样就可以创建一个名为 `myenv` 的虚拟环境, 并安装 `Python 3.10` 的尽可能新的版本。

下面我们需要激活虚拟环境:

```
conda activate myenv
```

这样就进入了虚拟环境, 你可以在终端上看到提示符前面有 `(myenv)`, 表示当前处于 `myenv` 虚拟环境中。如没有, 可能是终端配置失败或者其他原因, 安装 `OhMyPosh` 的部分主题也可能导致这个问题 (最大的可能是你的主题没有 `Python` 虚拟环境对应的 `section`)。

现在在 `Python` 中我们一般流行使用 `pip` 来安装包, 它从 `PyPI` 安装和管理第三方库。现在不流行直接使用 `conda` 管理包了。

我们可以使用以下命令安装一个包, 例如安装 `Numpy` 库:

```
pip install numpy
```

这将会在当前虚拟环境中安装 `Numpy` 库, 而不会影响全局的 `Python` 环境。

如果你需要安装多个包, 可以将它们写在一个文件中, 例如 `requirements.txt`, 然后使用以下命令安装:

```
pip install -r requirements.txt
```

以上命令也常用于项目的依赖管理, 可以方便地安装和更新项目所需的所有包; 同时也可以通过修改 `requirements.txt` 文件来管理项目的依赖版本, 适宜分发。

5.3 选择合适的 IDE 或者文本编辑器

IDE（集成开发环境）或者文本编辑器是编程的核心工具之一，用于编写代码、调试和测试等功能。选择合适的 IDE 或编辑器可以提高编程效率和代码质量。

聪明的懒人宁可使用一天时间来把环境配好来节省以后的时间用来摸鱼，而不是天天花时间来鼓捣环境。

5.3.1 选择编辑器

最 Geek 的一批程序员最喜欢使用命令行编辑器，例如 Vim、Emacs、NeoVim 等。这些编辑器通常具有强大的功能和高度的可定制性，适合喜欢命令行操作和自定义配置的用户。但是这些编辑器的使用难度极高，学习曲线陡峭，对于初学者来说极不友好。StackOverflow 上的一个非常著名的笑话是“Vim 是一个非常强大的编辑器，但是你需要先学会如何退出它”。

题外话：退出 Vim 的命令是 `:q`，如果你在编辑器中输入了内容并且想要保存，可以使用 `:wq` 命令；如果你不想保存，可以使用 `:q!` 命令强制退出。

正课一般推荐以下的几个 IDE：C++ IDE 是 Visual Studio 和 DevC++，Python IDE 是 PyCharm。它们各有各的优势，并且有一个最大的共同点：开箱即用，用户并不需要复杂的配置来进行编程。

但是它们的缺点非常明显：Visual Studio（一般简称 VS）和 PyCharm 都非常臃肿，尤其是前者如果安装全家桶需要大量的磁盘空间和内存资源。同时，它们更侧重于超大型项目的开发，这一“超大”往往动辄涉及数十万甚至上百万行代码，我们日常学习使用的代码量远远达不到这个级别，只能说是“杀鸡焉用牛刀”。VS 的另一个缺点是它实际上专精于 Windows 平台和微软的 .NET 生态系统，虽然它也支持 C++ 和 Python 等语言，但很笨重。至于 DevC++，它的功能少得可怜且只支持 C/C++。虽然比较适合初学者，但扩展性极差，完全无法满足更复杂的开发需求。

因此，我并不推荐初学阶段就使用这些 IDE。从长远来看，使用更加通用的编辑器会更有利于你在编程世界中游刃有余。我们强烈推荐同学们使用 Visual Studio Code（VSCode）完成大多数的任务。它是一款轻量级的编辑器，具有良好的扩展性和社区支持，可以满足不同用户的需求。VSCode 支持多种编程语言，并且有丰富的插件生态系统，可以根据需要安装各种插件来增强功能。它同时也为调试和版本控制功能添加了 GUI，非常适合初学者和中小型项目开发使用。

除此之外，还有一些语言仅在特定的编辑器中有很好的支持，例如 C# 之于 Visual Studio，SQL 之于 DBeaver 和 DataGrip，Java 之于 Eclipse 等。这些语言通常需要特定的 IDE 来提供更好的支持和功能，此时再去使用 VS Code 可能会有些不便。

5.3.2 安装 VS Code

我们应该上官网下载安装包进行安装。我们需要安装的是 System Installer 版本，而不是 User Installer 版本。因为 User Installer 版本会将 VS Code 安装在用户目录下，而 System Installer 版本会将 VS Code 安装在系统目录下，这样可以方便地在所有用户之间共享 VS Code，并且能够把它直接放在环境变量中。

安装完成后，我们需要设置环境变量，以便在命令行中直接使用 `code` 命令。不过如果你在安装时选择了“Add to PATH”选项，则不需要手动设置环境变量。

5.3.3 配置 VS Code

VS Code 是一个非常灵活的编辑器，可以通过安装插件来增强其功能。常用的插件包括：

- **Python**: 提供对 Python 的支持，包括语法高亮、代码补全、调试等功能。
- **C/C++**: 提供对 C/C++ 的支持，包括语法高亮、代码补全、调试等功能。
- **GitLens**: 增强版的 Git 支持，可以更好地查看版本历史和代码变更。
- **Chinese (Simplified) Language Pack for Visual Studio Code**: 提供中文界面支持。

此外，VS Code 还支持多种主题和图标包，可以根据个人喜好进行定制。你可以在 VS Code 的插件市场中搜索并安装这些插件和主题等。

5.3.3.1 在 VS Code 中配置 C/C++

如果你使用的是 GCC 编译器，则需要在 VS Code 中配置 GCC，以便能够编译和运行 C/C++ 代码。可以通过以下步骤进行配置：

1. 安装 C/C++ 插件：在 VS Code 的插件市场中搜索并安装 C/C++ 插件。建议直接安装微软提供的全家桶。
2. 配置 `tasks.json` 文件：在 VS Code 中创建一个 C++ 文件 `*.cpp` 或者 `*.cc`，然后随便输入一些什么代码，然后编译之。首次编译 C++ 代码时，VS Code 会提示你创建一个 `tasks.json` 文件。选择“C/C++: g++.exe 生成活动文件”，这将会在项目根目录下创建一个 `.vscode/tasks.json` 文件。（如果你创建的是 C 文件 `*.c`，那么你可以选择“C/C++: gcc.exe 生成活动文件”）
3. 配置 `launch.json` 文件：在 VS Code 中按下 F5，选择“C++ (GDB/LLDB)”，然后选择“g++.exe build and debug active file”。这将会在项目根目录下创建一个 `launch.json` 文件。（如果没有后一步，可以忽略之。）

如果你并不信任自动生成的配置文件或者需要更多的功能（例如开 `-O2` 优化），可以手动创建并修改 `tasks.json` 和 `launch.json` 文件。这两个文件都应该放在项目根目录下的 `.vscode` 文件夹中。

以下是一个简单的 `tasks.json` 文件示例（其实就是上面自动生成的那个，按照笔者的计算机环境稍微改了改）：

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "C/C++: g++.exe 生成活动文件",
      "type": "shell",
      "command": "g++",
      "args": [
        "-g",
        "${file}",
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "problemMatcher": ["$gcc"],
      "detail": "生成活动文件"
    }
  ]
}
```

以下是一个简单的 launch.json 文件示例:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "C++ Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
```

```

    {
      "description": "Enable pretty-printing for gdb",
      "text": "-enable-pretty-printing",
      "ignoreFailures": true
    }
  ],
  "preLaunchTask": "C/C++: g++.exe 生成活动文件",
  "miDebuggerPath": "C:\\msys64\\ucrt64\\bin\\gdb.exe"
}
]
}
```

如不想用 JSON 文件进行配置，我们还可以使用 UI 配置相关功能。

在 Code 中按下 `Ctrl + Shift + P`，找到“C/C++: Edit Configurations (UI)”选项。这样就可以通过图形界面来配置 C/C++ 的编译和调试选项。

一般地，我们需要更改以下内容：

- 配置名称：默认即可。
- 编译器路径：选择你要选用的编译器的路径。该选项一般会检测电脑上的编译器。
- 编译器参数：留空即可。如需要，可以添加一个 `-O2` 或者 `-O3` 来开启编译优化，或者添加一个 `-g` 来开启调试信息。但是，`-g` 会严重拖慢编译速度，因此建议只在调试时开启。
- IntelliSense 模式：根据你的系统、编译器、CPU 架构选择对应的模式。该选项会为你打开对应的代码补全、语法高亮、错误警示功能。
- 包含路径：不用动。
- 定义：不用动。
- C/C++ 标准：建议 C17 和 C++17，和 PKU 线上代码检查的标准一致。其他学校的学生按自己的学校要求来设置，例如 14 或 11。
- 高级：一个都不用动。

非常闹麻的一点是，调试配置没有 UI 配置选项，我们还得老实地手动编辑 `launch.json` 文件。当然默认调试已经够了，如果你不需要更复杂的调试功能，完全可以不修改。

5.3.3.2 在 VS Code 中配置 Python

在 VS Code 中配置 Python 非常简单。只需要安装微软提供的三个 Python 插件，然后在 VS Code 中打开一个 Python 文件。

你会在右下角看到一个黄色按钮“选择 Python 解释器”，点击它可以让你选择你想要使用的 Python 解释器。一般情况下，你可以选择“Python 3.x (conda)”或者“Python 3.x (venv)”等选项，这样 VS Code 就会自动识别你当前的虚拟环境。

当然，我们非常建议同学们趁早熟悉纯命令行运行 Python 的方式，例如 `python main.py`，这样可以更好地理解 Python 的运行机制，同时也更便于调试 (?)。

5.3.3.3 在 VS Code 中配置 Git

在 VS Code 中配置 Git 同样非常简单。只需要安装 Git，并确保 Git 的可执行文件在系统的 PATH 环境变量中。然后在 VS Code 中打开一个 Git 仓库，VS Code 会自动识别并启用 Git 功能。

5.4 美化你的终端

在 Windows 上，默认的终端是 cmd 或者 PowerShell，它们的界面比较简陋，不能显示很多信息。为了让终端更美观、更实用，我们可以使用一些终端美化工具，这里我推荐使用 **Oh My Posh**。

5.4.1 安装 Oh My Posh

我们建议使用 winget 来安装 Oh My Posh。你可以在 PowerShell 中运行以下命令来安装：

```
winget install JanDeDobbeleer.OhMyPosh
```

安装完成后，你需要在 PowerShell 中运行以下命令来初始化 Oh My Posh：

```
oh-my-posh init pwsh
```

在配置 Oh My Posh 的时候，很多的命令涉及到执行脚本。默认情况下，Windows PowerShell 会阻止执行脚本以保护系统安全。因此，你需要先修改执行策略来允许执行脚本。在 PowerShell 中运行以下命令：

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

就可以允许当前用户执行远程签名的脚本。

如果你使用的是其他终端，例如 cmd 或者 Git Bash，你可以在 Oh My Posh 的[安装文档](#)中找到相应的安装方法。

5.4.2 配置 Oh My Posh

首先，你应该安装 OMP 推荐使用的字体，例如 Nerd Font。这是因为 Oh My Posh 使用了一些特殊的图标，如果没有合适的字体，可能会导致图标无法正常显示。

你可以在[Nerd Fonts 官网](#)下载最新的字体包。安装完成后，你需要在终端中设置字体为 Nerd Font，以便能够正确显示 Oh My Posh 的图标。另一个办法是利用 OhMyPosh 安装这个字体：

```
oh-my-posh font install meslo
```

安装完成后，你需要在终端中设置字体为 Nerd Font。以 PowerShell 为例，你可以右键点击窗口标题栏，选择“属性”，然后在“字体”选项卡中选择 Nerd Font 即可。

接下来，你可以在 PowerShell 中运行命令来设置 Oh My Posh 的主题了。主题可以在 Oh My Posh 的[主题文档](#)中找到。

5.4.3 在 VS Code 中配置终端

VS Code 内置了终端功能，可以方便地在编辑器中运行命令。终端默认使用系统的命令行工具，例如在 Windows 上是 cmd，在 Linux 上是 bash。

你可以通过快捷键 `Ctrl + ``（反引号）打开终端，也可以通过菜单视图 > 终端来打开。终端打开后，你可以在其中输入命令，和在普通命令行中一样。

如果你希望在 VS Code 中使用 Oh My Posh，只需要把 Code 的终端字体设置为 Nerd Font 即可。你可以在 VS Code 的设置中搜索 `terminal.integrated.fontFamily`，然后将其值设置为你安装的 Nerd Font 的名称，例如 MesloLGS Nerd Font。

5.5 编写程序的基本素养

做了这么多操作，我们终于可以编写第一个能跑的程序了。我们将使用 C++ 和 Python 两个语言来演示怎么书写第一个程序，同时告诉大家编程新手应有的素养。

5.5.1 编写你的第一个程序

由于众所周知的原因，我们的第一个程序通常是“Hello, World!”程序。它的作用是让我们熟悉编程语言的基本语法和编译运行流程，同时也是一个传统。而第二个程序一般往往是写一个加法，让我们熟悉输入输出的基本操作。

5.5.1.1 C++

对于 C++，我们可以使用以下代码来编写第一个程序。你可以在 VS Code 中创建一个新的 C++ 文件，例如 `hello.cpp`（该文件的路径不能包含空格和中文！），然后输入以下代码：

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

然后如果配置得当，我们就可以通过按下 F5 键来编译并运行这个程序了。VS Code 会自动调用编译器进行编译，并在终端中显示输出结果。如果一切顺利，你应该会看到“Hello, World!”的输出。

在一些极端情况下（例如无 GUI 环境），你可能需要手动编译和运行程序。可以使用以下命令来编译和运行程序：

```
g++ -o hello hello.cpp
./hello
```

请使用类似的方式手敲、编译、运行以下代码：

```
#include <iostream>
int main() {
    int a, b;
    std::cout << "Enter the first integer: ";
    std::cin >> a;
    std::cout << "Enter the second integer: ";
    std::cin >> b;
    std::cout << "Their sum is: " << a + b << std::endl;
    return 0;
}
```

5.5.1.2 Python

对于 Python，我们同样可以使用以下代码来编写第一个程序：

```
print("Hello, World!")
```

同样，如果配置得当，我们就可以通过按下 F5 键来运行这个程序了。VS Code 会自动调用 Python 解释器运行，并在终端中显示输出结果。同样的，如果希望使用命令行来运行程序，可以使用以下命令：

```
python hello.py
```

请使用类似的方式手敲、编译、运行以下代码：

```
a = int(input("Enter the first integer: "))
b = int(input("Enter the second integer: "))
print("Their sum is:", a + b)
```

5.5.1.3 这两个语言有什么区别？

可以看到，使用命令行来执行程序的方式有所不同：C++ 需要两步，但是 Python 只需要一步。这是因为 C++ 是编译型语言，需要先将源代码编译成可执行文件，然后再运行；而 Python 是解释型语言，直接运行源代码即可。前者的好处是，一份需要被反复运行的代码只需要编

译一次，之后可以反复高效率运行。而后者的好处是，代码修改后可以立即运行，但是需要反复解释执行，运行速度（相对的）非常缓慢。

另一个区别是，C++ 中，我们可以看到定义 `a` 和 `b` 之前需要先声明它们的类型，而 Python 中则不需要。这说明，C++ 是强类型语言，变量的类型在编译时就确定了；而 Python 是动态类型语言，变量的类型在运行时才确定。

而这也导致了一个问题：编译器可以识别全部的语法错误和部分的语义错误，因此一份能够编译通过的 C++ 代码，通常代码本身是正确的，但是算法可能因为极端数据出现错误，例如除零等；而 Python 则无法检查语法错误和语义错误，解释器只会在按顺序运行代码，直到在出现问题的地方停止。Python 自身的动态类型系统与缺少编译器带来的静态查错系统，使得实际写出来的 Python 代码中经常包含大量的错误。

在 Python 的较新版本中引入了“类型注释”，例如 `func(para: int) -> int`。VS Code 的 *Pylance* 插件能够识别类型注释，并在编辑器中提供有限的类型检查。一些新生代程序员在编写程序时，会使用类型注释来帮助自己检查代码的正确性，防止出现错误。在 C++ 的较新版本中，也引入了“类型推断”，我们可以把部分变量声明为 `auto` 类型，例如 `for(auto item : items)`，其中 `items` 是一个列表。编译器能够自动推断变量的类型，从而减少了代码的冗余。由此可见，编程语言的发展是不断演进的，程序员们不断引入新的特性和语法，以提高代码的可读性和可维护性；同时，我们还可以看到，强类型语言和动态类型语言之间的界限正在逐渐模糊。

5.5.2 学会阅读错误信息

从上文中我们知道，代码中出现错误是不可避免的一件事情。有时候，我们会犯较为低级的语法错误，此时编辑器会自动指出问题；有时候，我们在只有在代码跑起来的时候才能发现程序错误、不能执行，此时编译器或解释器会给出错误信息，帮助我们定位问题所在；还有一些时候，程序自己运行时并没有因为致命错误而停止运行，但是输出的结果并不是我们期望的，此时我们只能通过调试来解决问题。

例如，以下是 C++ 初学者常见的错误：

```
#include<iostream>
using namespace std;
int mian()
{
    cout<<"Hello World!"<<endl;
    return 0;
}
```

而它的错误信息是在编译时报出：

```
> g++ example.cpp -o example.exe
```

```
ld.exe: *.a(lib64_libmingw32_a-crtexewin.o): in function `main':
C:/.../crtexewin.c:70: undefined reference to `WinMain'
collect2.exe: error: ld returned 1 exit status
```

虽然信息略显抽象，但我们还是可以看到很多有用的信息。ld 是 C++ 中的链接器，再往上看可以发现对 WinMain 的引用是未定义的。这提示我们去看 main 函数，从而发现这里将 main 函数写成了 mian，因此链接器无法找到 main 函数，从而引发错误。

而 Python 给出的错误信息则更为直观，例如以下代码：

```
def calc(numbers):
    total = sum(numbers)
    count = len(numbers)
    return total / count

numbers = [10, 20, 30, 40, 50]
print("Average:", calc(numbers))

numbers.append("60")
print("Updated Average:", calc(numbers))
```

其报错是：

```
Average: 30.0
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    print("Updated Average:", calc(numbers))
  File "example.py", line 2, in calc
    total = sum(numbers)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

可以看到，解释器对第 10 行和第 2 行进行了报错。第 10 行的报错是因为在调用 calc 函数时，传入的 numbers 列表中包含了一个字符串“60”，而 calc 函数期望的是一个数字列表，因此在计算平均值时出现了类型错误（TypeError）。而第 2 行的报错则是因为在计算总和时，无法将整数和字符串相加。于是我们发现了问题所在：在第 9 行，我们向 numbers 列表中添加了一个字符串“60”，而不是一个数字。我们可以通过将其改为 numbers.append(60) 来解决这个问题。

顺便一提，这段代码在 C++ 这种强类型语言中是无法通过编译的（List<int> 类型不能进行 append(string)），但 Python 的解释器还是运行代码直到遇到了具体的问题，在输出信息中可以看到第一个 print() 语句仍然被正常地执行。

5.5.3 学会调试

调试（技术人一般直接说 **debug**）是我们发现和修复代码中隐藏起来的错误的最有力工具。调试可以帮助我们理解代码的执行流程，从而**定位**问题所在。

调试有两种手段：静态调试和动态调试。前者一般是通过静态分析工具（例如反汇编器）来分析代码的结构和逻辑，寻找潜在的问题；后者则是通过运行代码并观察其行为来发现问题。静态调试通常用于编译型语言且难度极高，我们不会涉及；而动态调试则适用于所有语言，接下来的内容我们将主要介绍动态调试。

C 系有着自己的调试器：**GDB**（GNU Debugger），它是一个强大的调试工具，可以在命令行中使用。**GDB** 可以让我们逐行执行代码，查看变量的值，设置断点等。**VS Code** 也集成了 **GDB**，可以通过图形界面进行调试。**Python** 也有类似的调试器：**PDB**（Python Debugger），它同样可以在命令行中使用，也可以通过 **VS Code** 进行调试。

纯命令行调试的方式极为困难（尤其是 **GDB**，需要背诵大量的命令），我们在这里不做介绍。然而，**VS Code** 提供了一个非常友好的调试界面，可以通过图形化的方式进行调试。我们可以在代码中设置断点，逐行执行代码，查看变量的值等。这样可以大大提高调试效率。（当然这需要你安装 **GDB**，安装并配置的过程见 5.3.3.1。）

我们调试主要有以下几个手段：打日志、打断点、写测试代码。

5.5.3.1 打日志

打日志是指在代码中添加打印语句，以便在运行时输出某些特定变量的值，进而确定程序的执行流程。这样可以帮助我们理解代码的执行过程，定位问题所在。

新人常常不喜欢这种手段，因为它需要在代码中添加额外的打印语句，很丑陋、不优雅，且会影响代码的可读性和维护性。但是打日志是一个非常有效的调试手段，尤其是在工程量巨大、无法或者很难打断点的情况下。

例如我在调试某数万行的大型项目时，出现断言错误。于是本人在代码中添加了以下打印语句：



```
56 }
57 response = requests.post(url, headers=headers, data=json.dumps(data))
58 print(response.status_code)
59 # assert response.status_code == 200
60 data = response.json()
61 print(f'测试创建智能体: {data}; status_code: {response.status_code}')
62
63
64
```

图 5.1: 打日志的例子

这样我就知道了程序在执行到这里的时候，返回的不是预期的 200，而是 404。于是这让我顺藤摸瓜，排查可能会导致 404 的原因，最终发现是因为某个 API 的返回值发生了变化，导致程序无法正常运行。

这是打日志的一个典型例子。通过在代码中添加打印语句，我们可以快速定位问题所在，并进行修复。

5.5.3.2 打断点

在 VS Code 中，我们可以通过点击行号左侧的空白区域来设置断点。断点是调试过程中非常重要的工具，它可以让代码执行到特定的某行时暂停，从而查看当前的变量值和程序状态。

我们可以逐行执行代码，查看变量的变化，从而定位问题所在。

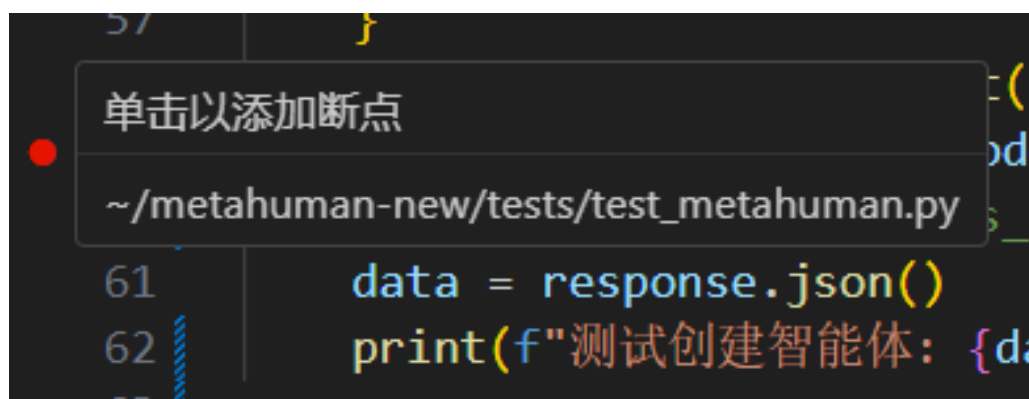


图 5.2: 打断点的例子

这样就可以打出一个断点。在调试过程中，当程序执行到断点所在的行时，程序会暂停，我们可以查看当前的变量值和程序状态。我们可以通过单步执行（Step Over）来逐行执行代码，或者通过单步进入（Step Into）来进入函数内部进行调试。对于小型项目或者单文件项目，打断点是一个非常有效的调试手段。

5.5.3.3 写测试代码

写测试代码是指编写一些专门用于测试的代码，以便在运行时验证程序的正确性。测试代码可以帮助我们发现潜在的问题，并确保程序的功能正常。这也是用于较大型项目的调试手段，但是小型项目也可以使用。我们可以在这些测试代码中模拟各种可能出现的情况（包括常规值、边界值、异常值等），从而验证程序的正确性和健壮性。

测试代码通常分为单元测试和集成测试两种。单元测试是对程序的最小可测试单元进行验证，通常是函数或方法；而集成测试则是对多个单元进行组合验证，确保它们能够正常协同工作。

5.5.3.4 小结

debug 最重要的一件事是缩小错误出现的范围，为达成这一目的我们通常会跟踪代码的行为，直到发现代码的行为与预期不符。实际上最棘手的情况是，代码只在特定的数据上出

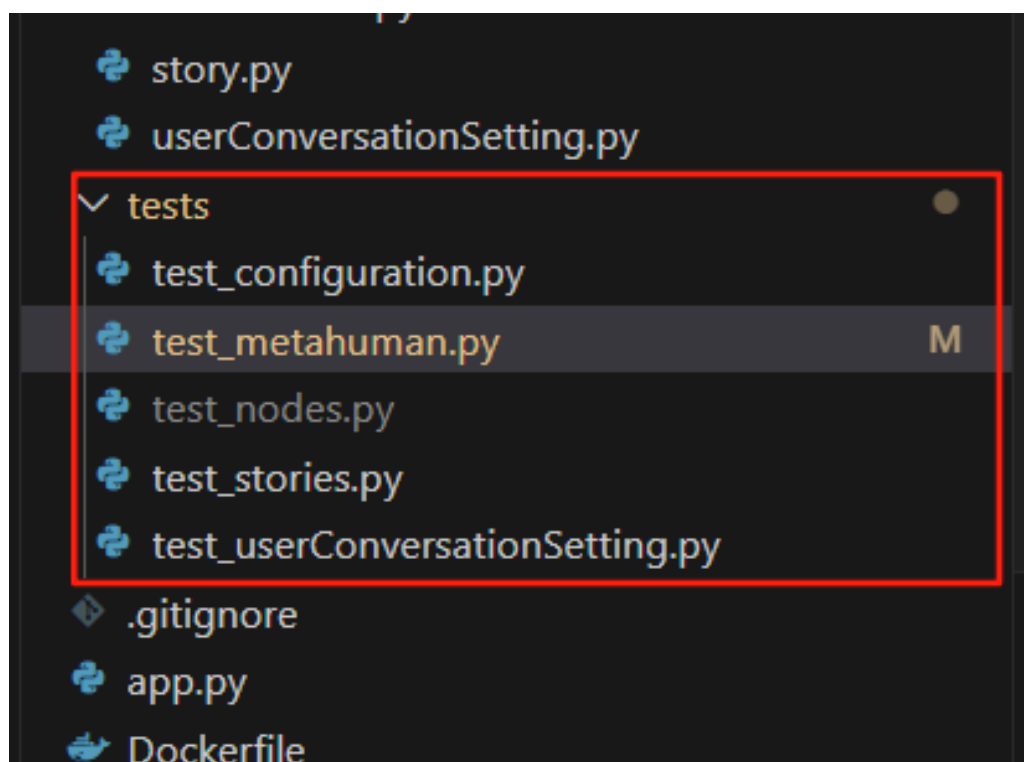


图 5.3: 我为本人实习的项目编写的集成测试

现错误，尤其是当我们无法获取程序执行日志的时候。这种情况尤见于我们在 POJ 上做题的时候：POJ 的测试数据是不可见的，只会告诉你结果是 WA、RTE 还是 TLE、MLE。

这时最应该做的是重新审视自己的预期（以及 OJ 题的题面），寻找是否遗漏了什么约束条件或关键信息。一份貌似运行正常的代码很有可能会在边界条件或复杂数据的情况下出问题，可以尝试手写一些处于边界条件之下的数据，或编写一个数据生成器来生成更复杂的数据。实在手足无措时，休息一下放空大脑也是很好的选择。实在走投无路之时，摇人求助也不是什么大不了的事情。debug 很可能会占用比编写代码更多的时间和精力，保持良好的心态才是 debug 的关键。

第六章 文字排版

文本编辑工具是我们表达思想、传递知识的重要手段。无论是在科研、写作还是展示中，排版都是重要且必要的内容；我们需要选择合适的排版工具，以大大提高作品质量。

目前最常用的排版工具或语言有 Microsoft Word、Markdown、LaTeX 和 Typst 等。每种工具都有其特点和适用场景：在日常生活中，Microsoft Word 因其简单易用、所见即所得的特点而广受欢迎；Markdown 因为其轻量级和易于学习的特点而在技术文档和博客中广泛使用；在科研写作中，又属 LaTeX 最为常用，因为它提供了强大的排版功能，尤其是在处理复杂的数学公式和图表时；而 Typst 则是近年来新兴的排版工具，旨在提供一种更直观、更易于使用的排版方式，正在年轻一代技术人中逐渐流行开来。

特性	MS Word	Markdown	LaTeX	Typst
安装	简单	简单	难	简单
语法复杂度	非常简单	简单	高	中
编译速度	较快	快	慢	快
排版能力	较强	一般	极强	较强
模板能力	几乎没有	中等	强	强
编程能力	无	无	强，风格古老	强，风格现代
方言	无	极多	有	较少

表 6.1: 不同排版工具的对比

6.1 Markdown

Markdown 是一种轻量级的标记语言，可用于在纯文本文档中添加格式化元素。和其他排版工具相比，它仅仅使用十几个记号进行排版。这使得它易于学习，使得使用者能够更专注于内容的同时，快速地进行美观大方的排版。

Markdown 无需安装，用户使用任何文本编辑器即可编写 Markdown 文档。一般的，VS Code 已经集成了 MD 的语法高亮和预览功能，用户只需要安装 Markdown 插件即可。

6.1.1 Markdown 的语法

Markdown 的内容输入和纯文本文件几乎一模一样，接下来我们将逐个介绍 Markdown 排版所用到的控制符号。不过，在此之前，请把你的输入法标点符号切换为半角，防止出现类似于 ¥¥ 的臭名昭著的错误。

值得注意的是，Markdown 的语法并不是固定的，不同的 Markdown 渲染器可能会有一些差异，这被称作方言性。我们在这里介绍的是最常见也最通用的 Markdown 语法。同时，你在不同的地方看到的 MarkDown 渲染结果可能会不同，这也是正常现象。

6.1.1.1 分段、换行、分割线

在 Markdown 中，必须通过空行来进行分段。也就是说，如果你想要对文件进行分段，需要在两段之间加入一个空行。特别注意，Markdown 不接受缩进或者首行缩进，所以不要使用 Tab 键或者空格进行缩进！（否则会编译为代码块）

而如果希望仅仅换行而不分段，则仅仅在行尾加入两个空格，然后另起一行，在新的行中书写；或者使用 HTML 标记，也就是 `
` 符号，该符号无需另起一行也可以进行换行操作。

对于分割线（你经常会在知乎看见这种分割线），请在单独一行上使用三个或多个星号（***）、连接号（---）或下划线（___），并且不能包含其他内容。为了兼容性考虑，请在该分割线前后加上空行。

6.1.1.2 标题

Markdown 使用井号（#）来表示标题。井号的数量表示标题的层级，例如：

```
# 一级标题
## 二级标题
### 三级标题
```

6.1.1.3 强调、删除

在 Markdown 中，可以使用星号（*）或下划线（_）来表示强调。单个星号或下划线表示斜体，两个星号或下划线表示粗体。同时，还可以使用波浪号（~）来表示删除线，例如：

```
*斜体文本*
**粗体文本**
***粗斜体文本***
~~删除线文本~~
```

6.1.1.4 转义

在 Markdown 中，如果需要输入特殊字符（例如星号、井号等），可以使用反斜杠（\）来进行转义。例如，\`*` 代表一个星号。

6.1.1.5 代码和代码块

在 Markdown 中，可以使用反引号来表示代码。单个反引号表示行内代码，三个反引号表示代码块。例如：

```
`行内代码`
```



```
```  
代码块
```
```

如果需要在代码块中打出反引号且防止此反引号被编译，只要保证用于包装的反引号数量比防止编译的反引号数量多就可以了。

6.1.1.6 引用

在 Markdown 中，可以使用大于号 (>) 来表示引用。引用块也是可以嵌套的，只需要在每一行的开头添加一个大于号即可。例如：

```
> 这是一段引用文本。  
> > 这是一段嵌套的引用文本。
```

6.1.1.7 列表

在 Markdown 中，可以使用减号 (-) 来表示无序列表。例如：

```
- 列表项 1  
- 列表项 2  
- 列表项 3
```

有序列表则使用数字加点的方式表示，例如：

```
2. 列表项 2  
3. 列表项 3
```

6.1.1.8 表格

在 Markdown 中，可以使用竖线 (|) 来表示表格。表格的第一行是表头，第二行是分隔线，后面的行是表格内容。例如：

```
| 列表项 1 | 列表项 2 | 列表项 3 |  
| ----- | ----- | ----- |  
| 内容 1 | 内容 2 | 内容 3 |
```

6.1.1.9 外部资源：链接、图片

在 Markdown 中，可以使用方括号和圆括号来表示链接。例如：

```
[链接文本](https://www.example.com)
```

在 Markdown 中，可以使用感叹号、方括号和圆括号来表示图片。例如：

```
![图片描述](https://www.example.com/image.png)
```

6.1.1.10 数学公式

在 Markdown 中，可以使用美元符号 (\$) 来表示行内公式。例如：

```
这是一个行内公式  $E=mc^2$  的示例。
```

如果需要输入块级公式，可以使用两个美元符号 (\$\$) 来包裹公式。例如：

```
这是一个块级公式的示例：
```

```
$$ E=mc^2 $$
```

大多数的 Markdown 编译器都会正确渲染 LaTeX 公式，但是不排除少数编译器不支持渲染 LaTeX。一般而言，不支持渲染的编译器将会原样显示公式内容。开始写作前试试编辑器能否正常渲染公式总是好的选择。

6.2 L^AT_EX

L^AT_EX 是一种基于 TeX 的排版系统，广泛用于学术论文、书籍和其他需要高质量排版的文档。与 Markdown 相比，L^AT_EX 提供了更强大的排版功能，尤其是在处理复杂的数学公式和图表时。

它的源文档与 Markdown 的简洁干净不同，而是充斥着许多反斜杠、大括号和宏。这表明如果直接使用 LaTeX 进行文本编辑的话会令人极度头大乃至效率降低；因此我个人建议同学们在使用上述工具时，最好是心中打好腹稿然后再进行工作。

6.2.1 LaTeX 的安装

虽然 LaTeX 功能强大，但是其安装过程非常缓慢且困难。对于不愿意在自己电脑上本地安装这东西的读者，笔者建议使用一些线上编译器，例如著名的 Overleaf 等。PKU LaTeX 也是一个线上编译器，由 LCPU 开发并维护，欢迎大家使用！

LaTeX 的安装冗长且复杂，通常需要安装一个完整的 TeX 发行版，例如 TeX Live。

6.2.1.1 Windows 机器

在安装新的 TexLive 之前，笔者建议彻底删除任何旧版的 CTeX 套装，同时检查环境变量中有没有 C:\Windows\System32。如无，请将上述路径添加回环境变量中去。

然后，检查自己的用户名是不是无空格的英文。如果不是，建议修改，这是一个一劳永逸的办法。另一个办法是执行以下命令（注意：PowerShell 用户请自行替换命令为正确的命令）：

```
mkdir C:\temp
set TEMP=C:\temp
set TMP=C:\temp
```

如无意外，用户可以从最近的 CTAN 源下载 TexLive 的相关镜像（这个镜像大小高达 6GB）。当然，官网下载过程是非常缓慢的，如果实在是无法忍受其速度，可以考虑改用其他镜像站。

由于未知原因，如果计算机上提前安装了 jdk、mingw 或 Cygwin，建议暂时先把以上软件从环境变量中剔除，等整个安装好了以后再加回去。2345 好压可能也会导致类似的错误，本人建议彻底卸载之，并从此以后不要碰相关的东西；笔者推荐使用 7z 这个压缩软件。

将下载下来的虚拟光驱镜像装载到虚拟光驱中，然后执行其中的批处理文件进行安装。安装过程中，建议选择“安装所有包”以防出现各种未知的错误。之后，在弹出的窗口中选择清华源（校外）或者北大源（校内，速度更快）并进行下载安装。安装过程可能需要较长时间，请耐心等待。

如果你不希望安装在默认的 C:\texlive 目录下，可以在安装过程中选择自定义安装路径。但是，该目录不应包含任何非 ASCII 字符，或者说不应包含空格或其他非英文特殊字符。安装完成后，建议将 TeX Live 的 bin 目录添加到系统的环境变量中，以便在命令行中直接使用 LaTeX 命令。我们不建议安装 TexLive 的 GUI 前端，因为它不易于使用。

6.2.1.2 Linux: 以 Ubuntu 为例

在安装前，建议将 Ubuntu 源更改至国内源以提高下载速度。建议直接去找清华源或者北大源提供的现成配置文件。

然后，下载光盘镜像，并进行装载。

```
sudo apt install fontconfig gedit
sudo mkdir /mnt/texlive
sudo mount ./texlive2025.iso /mnt/texlive
sudo /mnt/texlive/install-tl
```

之后，终端会弹出大量内容，我们可以按照提示进行操作。安装完毕后，将安装镜像卸载：

```
sudo umount /mnt/texlive
sudo rm -r /mnt/texlive # 删除临时挂载目录
```

在安装完毕后，安装程序会提示用户将一些目录添加到环境变量中。用户可以按照提示进行操作。

之后，我们应当配置字体。如果用户改变了安装路径，应将 path/ 改为自己的实际安装路径。

```
sudo cp path/texmf-var/fonts/conf/texlive-fontconfig.conf \
/etc/fonts/conf.d/09-texlive.conf
sudo fc-cache -fsv
```

6.2.2 LaTeX 在 VS Code 的配置

一般情况下，LaTeX 有两个编译器：pdfLaTeX 和 XeLaTeX。前者是传统的 LaTeX 编译器，适合简单的英文文档；后者则支持 Unicode 字符和 OpenType 字体，适合处理中文文档。我们建议通过手动配置来指定使用 XeLaTeX 编译器。

首先，我们应当下载并安装 VS Code 的 LaTeX Workshop 插件。该插件提供了 LaTeX 的语法高亮、自动补全、编译和预览等功能。之后，打开你的 Code 的用户设置 json 文件，并添加以下配置：

```
"latex-workshop.latex.tools": [
  {
    "name": "xelatex",
    "command": "xelatex",
    "args": [
      "-synctex=1",
      "-interaction=nonstopmode",
      "-file-line-error",
      "%DOC%"
    ]
  }
],
"latex-workshop.latex.recipes": [
  {
    "name": "xelatex",
    "tools": [
      "xelatex"
    ]
  }
],
```

然后，如果没有什么问题的话，VS Code 就会使用 XeLaTeX 编译器来编译你的 LaTeX 文档了。

我们非常建议关闭 LaTeX Workshop 的自动清理功能，因为它会在每次编译后删除所有的辅助文件，这会导致目录、参考文献等相关功能难以正常工作——这些工作往往要求连续编译两次，因此辅助文件是很必要的。为了关闭这一功能，我们可以在用户设置 json 文件中添

加以下配置:

```
"latex-workshop.latex.autoClean.run": "never",
```

如果我们不编译很长的文章的话,可以打开自动编译功能,这样每次保存文档时,VS Code 都会自动编译 LaTeX 文档。但是对于超长文档,自动编译会导致每次习惯性按下保存时都要等待许久。我们需要按需开启或关闭自动编译功能。可以在用户设置 json 文件中添加以下配置:

```
"latex-workshop.latex.autoBuild.run": "onSave",
```

这样每次保存文档时,VS Code 都会自动编译 LaTeX 文档。将"onSave"改为"never"则可以关闭自动编译功能。

6.2.3 LaTeX 的语法

一个 LaTeX 文档通常由以下几个部分组成:

- 文档类声明: 指定文档的类型和格式。
- 导言区: 加载宏包和设置文档的全局参数。
- 正文: 实际的内容部分。

一个简单的 LaTeX 文档示例如下:

```
\documentclass{article}
\usepackage{amsmath}

\begin{document}
\title{我的第一篇 LaTeX 文档}
\author{张三}
\date{\today}
\maketitle
\section{引言}
这是一篇 LaTeX 文档。我们可以在这里写一些数学公式:
\begin{equation}
E=mc^2
\end{equation}
\section{结论}
LaTeX 是一种强大的排版工具,适用于学术论文和书籍等文档的排版。
\end{document}
```

6.2.3.1 文档类声明

文档类声明是 LaTeX 文档的第一行,指定了文档的类型和格式。常用的文档类包括:

- **article**: 文章。最基本的文档类型。用于排版不太长（几页）的文字。
- **report**: 报告。用于排版较长（数十页）的文章。支持章节和附录。
- **book**: 书籍。用于排版更长（数百页或者更多）的文章或者书籍，支持章节、索引和目录等功能。
- **letter**: 信件。通常用于排版纸质的正式信件。
- **beamer**: 幻灯片（Beamer 是德语）。用于制作幻灯片演示文稿。
- **moderncv**: 现代英式简历。

要声明一个文档类，只需在文档的第一行使用 `\documentclass` 命令。例如，要声明一个文章类型的文档，可以使用以下命令：

```
\documentclass{article}
```

6.2.3.2 导言区

导言区是 LaTeX 文档的第二部分，用于加载宏包和设置文档的全局参数。宏包是 LaTeX 的扩展功能，可以提供额外的排版功能和样式。在导言区，可以使用 `\usepackage` 命令来加载宏包。例如，要加载 `amsmath` 宏包以支持数学公式，可以使用以下命令：

```
\usepackage{amsmath}
```

除此之外，还可以通过其他命令来设置文档的全局参数，例如字体、行距、页边距等。一些常用的全局参数设置命令可以在 6.2.5 中找到。

6.2.3.3 正文

正文是 LaTeX 文档的主要内容部分。在正文中，可以使用各种命令来添加标题、段落、列表、表格、图片等内容。正文部分被包裹在 `\begin{document}` 到 `\end{document}` 的块中间。

LaTeX 其实还有“注释”部分，这些部分仅在 tex 源码中存在，但是不会出现在最终的 pdf 文档中。注释部分以百分号（%）开头，本行后面的内容会被 LaTeX 忽略。例如：

```
\begin{document}
...正文...
% 这是一个注释
...正文...
\end{document}
```

6.2.4 LaTeX 的常用命令

6.2.4.1 标题

使用 `\section`、`\subsection` 和 `\subsubsection` 命令来添加标题，这三个命令分别可以添加一级标题、二级标题和三级标题，这些标题会自动标号。对于 `report` 和 `book` 文档类，还可以使用 `\chapter` 命令添加章节标题、`\part` 命令添加部分标题，这些标题也会编号。

我们还可以使用 `\title`、`\author` 和 `\date` 命令来设置文档的标题、作者和日期。然后使用 `\maketitle` 命令来生成标题页。同时，诸如 `\section*` 等命令可以生成无编号的标题。

6.2.4.2 段落

在 LaTeX 中，段落是通过空行来分隔的。要开始一个新段落，只需在上一段的末尾添加一个空行即可。

6.2.4.3 换行

如果需要在段落中换行，可以使用 `\newline` 命令，或者在行尾添加两个空格后再换行。也可以使用 `\\` 来换行，我们一般使用的是这种方式。

6.2.4.4 强制空格

对于纯英语的文本，LaTeX 会自动处理空格。但有时在英汉混写或纯汉语的文档中，代码中的空格不会编译，此时我们需要在文本中插入一个强制空格。可以使用 `\` 命令来实现，这个反斜杠前后都应该加上空格。例如：

这是一个强制 \ 空格的例子。

6.2.4.5 转义字符和反斜杠

在 LaTeX 中，反斜杠 (`\`) 是一个特殊字符，用于引入命令和转义。例如 `\$` 可以强制输出美元符号而不是将其视为数学公式的开始。如果需要在文本中输入反斜杠本身，不可以使用两个反斜杠 (`\\`)，而是需要使用 `\textbackslash` 命令来实际地打出一个反斜杠。

除此之外，LaTeX 支持飘号 `~`，但是默认的飘号极不美观。我们可以使用 `\textasciitilde` 命令来打出一个美观的飘号。

6.2.4.6 数学公式

LaTeX 对数学公式的支持非常强大。可以使用美元符号 (`$`) 来包裹行内公式，或者使用两个美元符号 (`$$`) 来包裹块级公式。例如：

这是一个行内公式 $E=mc^2$ 的示例。

这是一个块级公式的示例：

$$E=mc^2$$

除此之外，也可以使用 `\begin{equation}` 和 `\end{equation}` 来包裹数学公式，这样可以自动编号公式。例如：

```
\begin{equation}
E=mc^2
\end{equation}
```

对于多行公式¹，可以使用 `\begin{align}` 和 `\end{align}` 来包裹公式，并使用 `&` 符号来对齐公式。例如：

```
\begin{align}
E &= mc^2 \\
F &= ma
\end{align}
```

LaTeX 还支持各种数学符号和运算符，例如乘号 (`\times`)、除号 (`\div`)、积分符号 (`\int`) 等。可以通过查阅 LaTeX 的数学符号手册来了解更多数学符号的用法。

6.2.4.7 文本强调

可以使用 `\emph{文本}` 命令来强调文本，通常会将文本设置为斜体，实际上可能会根据文档类和宏包的不同而有所变化。除此之外：

- `\textbf{文本}`：粗体，**Bold**。
- `\textit{文本}`：斜体，*Italic*。
- `\texttt{文本}`：等宽字体，`Typewriter`。
- `\textsf{文本}`：无衬线字体，`Sans-serif`。
- `\textsc{文本}`：小型大写字母，`SMALL CAPS`。
- `\underline{文本}`：下划线，Underline。

6.2.4.8 列表

使用 `itemize` 环境来创建无序列表，使用 `enumerate` 环境来创建有序列表。这个列表也是可以嵌套的，例如：

```
\begin{itemize}
```

¹需要宏包 `amsmath`

```

\item 列表项 1
\item 列表项 2
\begin{itemize}
  \item 嵌套列表项 1
  \item 嵌套列表项 2
\end{itemize}
\item 列表项 3
\end{itemize}

```

6.2.4.9 表格

使用 `table` 环境来创建表格。以下是一个示例：

```

\begin{table}
  \centering
  \begin{tabular}{lr}
    列1 & 列2 \\
    \hline
    内容1 & 内容2 \\
    内容3 & 内容4 \\
  \end{tabular}
  \caption{一个简单的表格}
  \label{tab:simple-table}
\end{table}

```

上述代码创建了一个简单的表格，其中 `tabular` 环境用于定义表格的列格式，`l` 表示左对齐，`r` 表示右对齐；如果需要居中对齐，可以使用 `c`。上述 `lr` 意思是第一列左对齐，第二列右对齐，两列中间没有竖线，表格左右两边也没有竖线。如果需要添加竖线，可以使用 `|` 符号，例如 `|l|r|` 表示第一列左对齐，第二列右对齐，并且两列之间有一条竖线，表格的左右两边也有竖线。

表格中的每一行使用 `\\` 来分隔，列之间使用 `&` 来分隔（实际上是制表符）。用 `\hline` 命令来添加水平线。如果想要添加粗线，可以使用 `\hline\hline`。

`caption` 命令用于添加表格的标题，`label` 命令用于给表格添加标签²，以便在文档中引用。上述代码展示的标签是一个标签的习惯命名方式，不必严格按照这个格式来命名标签。

6.2.4.10 图片

使用 `graphicx` 宏包来插入图片。首先，在导言区加载这个宏包，然后使用 `figure` 环境来插入图片。例如：

²需要 `hyperref` 宏包

```

\usepackage{graphicx}

\begin{figure}
  \centering
  \includegraphics[width=0.5\textwidth]{example-image}
  \caption{一个示例图片}
  \label{fig:example}
\end{figure}

```

上述代码中，`includegraphics` 命令用于插入图片，`width` 参数用于设置图片的宽度，可以使用相对宽度（例如 `0.5\textwidth` 表示图片宽度为文本宽度的一半）或者绝对宽度（例如 `5cm`）。`caption` 和 `label` 命令的用法与表格类似。

6.2.4.11 超链接与标签引用

可以使用 `hyperref` 宏包来创建超链接。首先，在导言区加载这个宏包，然后使用 `\href{链接地址}{链接文本}` 命令来创建超链接。例如：

```
这是一个超链接：\href{https://www.example.com}{超链接文本}
```

标签引用则与超链接类似，可以使用 `\label{标签名}` 命令来创建标签，然后使用 `\ref{标签名}` 命令来引用标签。例如：

```

\section{引言}
\label{sec:introduction}

....（此处略去许多正文内容）....

```

在 `\ref{sec:introduction}` 节中，我们介绍了 L^AT_EX 的基本用法。

6.2.4.12 目录

使用 `\tableofcontents` 命令可以生成文档的目录。L^AT_EX 会自动根据文档中的标题生成目录，并在需要时自动更新。

目录也属于标签引用的一种，因此在使用目录之前，需要先编译一次文档以生成目录文件。之后再编译一次，目录就会出现在文档中。

6.2.4.13 脚注

可以使用 `\footnote{脚注内容}` 命令来添加脚注。

6.2.4.14 引用

可以使用 `\quote{引用内容}` 命令来添加引用。LaTeX 还支持使用 `\cite{文献标签}` 命令来引用文献，这需要在导言区加载 `biblatex`。

6.2.4.15 参考文献

可以使用 BibTeX 或 BibLaTeX 来管理参考文献。可以在导言区加载 `biblatex` 宏包，并使用 `\addbibresource{*.bib}` 命令来添加参考文献文件。

在正文中，可以使用 `\cite{文献标签}` 命令来引用文献。文献标签通常是一个唯一的标识符，用于在参考文献列表中查找对应的文献条目。

在文档末尾使用 `\printbibliography` 命令来生成参考文献列表。

`bib` 文件需要遵循一定的格式来编写，我们可以自行查找相关资料。

6.2.5 LaTeX 的宏包

LaTeX 的宏包是扩展 LaTeX 功能的重要工具。通过加载不同的宏包，可以实现各种排版效果和功能。除了我们刚刚提到的 `amsmath`、`graphicx`、`hyperref` 和 `biblatex` 等常用宏包外，还有许多其他有用的宏包。例如：

- `geometry`：用于设置页面布局和边距。
- `fontspec`：用于设置字体，特别是在 XeLaTeX 中。
- `tikz`：用于绘制图形和图表。
- `pgfplots`：用于绘制函数图像和数据可视化。
- `listings`：用于排版代码。
- `algorithm2e`：用于排版算法。

6.2.6 LaTeX 与汉语专题

非常遗憾的是，LaTeX 对于汉语、日语、朝鲜语等非字母语言的支持稍差。因此我们在使用 LaTeX 排版中文文档的时候，请确保使用 XeLaTeX 编译器，并且在导言区加载 `ctex` 宏包。这个宏包提供了对中文的支持，包括中文字体、段落格式等，这也是 LaTeX 主流的中文排版方式，其底层宏包是 `xeCJK`（C 指中，J 指日，K 指韩）。

LaTeX 的字体分三种：常规字体、无衬线字体和等宽字体。常规字体是指普通的衬线字体，通常用于正文；无衬线字体是指没有衬线的字体，通常用于标题或强调；等宽字体是指每个字符宽度相同的字体，通常用于代码或表格。对于 Windows 下的中文字体，默认情况下 LaTeX 使用的是宋体（SimSun）作为常规字体，微软雅黑（Microsoft YaHei）作为无衬线字体，等宽字体则使用仿宋（FangSong）。

需要指出的是，Windows 系统中，在 LaTeX 中对宋体（SimSun）进行加粗，实际上得到的是黑体（SimHei）；同理斜体是楷体（KaiTi）。而在 Linux 中，宋体（FandolSong-Regular）

的粗体是真粗体（FandolSong-Bold）。这和 MS Word 中的宋体加粗是不同的：MS Word 中的宋体加粗实际上是“伪粗体”，只是通过加粗字形来实现的。因此，同学们在使用 LaTeX 排版中文文档时，请注意这一点。

我们可以使用 `ctex` 宏包来设置中文文档的格式和字体。以下是一个简单的示例：

```
\documentclass{article}
\usepackage[UTF8, heading=true]{ctex}
% 这里的heading=true的意思是“采用中式标题”，没有的话会报错

\title{\zihao{0} 标题} % 这里的意思是用小初号标题

\ctexset{
  section = {
    format += \zihao{4}\kaishu\raggedright,
    % 这里的+=指的是在原有格式上增加以下内容,
    % \kaishu在XeCJK中指的就是\zhkai.
    name = {第,节},
    number = \arabic{section},
    aftername = \hspace{0.5em},
    beforeskip = 1.5ex plus .2ex minus .2ex,
    afterskip = 1.5ex plus .2ex minus .2ex
  }
}
% 这段代码的意思是把section的默认写法从一个数字变成：“第X节”，
% 并设置前后垂直间距等。
```

6.2.7 LaTeX 与多文件

我们在9.1中提到过，对于非常长的代码文件，建议将其拆分成多个文件进行管理。LaTeX 也不例外，虽然 `tex` 文件是以文本内容为主的，但是对于动辄几十页上万行的文档，拆分成多个文件也是必要的，LaTeX 也支持这种做法。

一些比较经典的手段是使用 `\input{文件名}` 或 `\include{文件名}` 命令来引入其他 LaTeX 文件。但是这两个命令风格比较老，不建议使用。

现代的 LaTeX，我们建议使用 `\subfile{文件名}` 命令来引入其他 LaTeX 文件，这个命令是由 `subfiles` 宏包提供的。这样能够较好地保证文件的独立性，不仅可以在主文件中编译，也可以在子文件中独立编译，使得调试更加方便。

举例说明：我们可以写一个主要文件 `main.tex`，内容如下：

```
\documentclass{article}
\usepackage{subfiles}
```

```

\begin{document}
\title{我的 LaTeX 文档}
\author{张三}
\date{\today}
\maketitle

\section{引言}
这是一个 LaTeX 文档的示例。

\subfile{section1.tex} % 引入 section1.tex 文件
\subfile{section2.tex} % 引入 section2.tex 文件

\end{document}

```

然后，我们可以创建两个子文件 section1.tex 和 section2.tex，内容如下：

```

\documentclass[main.tex]{subfiles} % 指定主文件是main.tex

\section{第一节}
这是第一节的内容。

\subsection{子节}
这是第一节的子节内容。

```

需要注意的是，采用多文件来管理文档的时候，我们在应对多文件引用时，需要多次编译才能得到正确的引用和目录，一般来说是三次。（而单文件文档只需要两次编译即可。）

6.3 Typst*

Typst 是一种新兴的排版语言，由著名的语言神 Rust 编写，旨在提供一种更直观、更易于使用的排版方式。它的完备性与 LaTeX 类似，但语法更简洁，易于上手，和 Markdown 类似。Typst 的设计理念是让用户能够专注于内容，而不是被复杂的语法和命令所困扰。

6.3.1 Typst 的安装

Typst 没有官方的包。但是，官方提供了 Typst 的 WebAPP，我们可以直接使用之，但是其中文字体和版本控制都不优秀。如果希望在本地使用，则可以在 VS Code 中安装插件 Tiny mist Typst。安装完成后，用户可以在 VS Code 中创建 .typ 文件，并使用 Typst 语法进行排版。

6.3.2 Typst 的语法

Typst 有两类语言模式：标记模式和脚本模式，而本质上都可以归结为脚本模式。

在默认情况下，Typst 使用标记模式进行排版，使用类似于 Markdown 的简单语法来编写文档。如果希望进入脚本模式，可以使用井号 # 来切换到脚本模式，例如 `#heading(strong([加粗]))` 是一个合法的语法。大段的脚本代码则可以使用花括号，例如 `#{1+1}`。

在脚本模式中，也可以使用方括号来进入标记模式，这将成为脚本模式的“内容块”或者标记元素。

6.3.2.1 标记模式

对于 Typst，所有的标记其实都是语法糖。这样能像 Markdown 一样做到内容和格式的彻底分离，也方便了对我们的控制。在标记模式中，用户可以使用以下语法来进行排版：

```
= 一级标题
上述文本等价于
#heading(level: 1, [一级标题])

== 二级标题
上述文本等价于
#heading(level: 2, [二级标题])

*加粗* 等价于 #strong[加粗]
_强调_ 等价于 #emph[强调]
需要注意，没有标记下划线，需要用#underline[下划线]来进行。
#strike[删除线]，没有标记删除线

- 无序列表
+ 有序列表
/术语：术语列表
上述文本等价于
#list.item[无序列表]
#enum.item[有序列表]
#terms.item[术语][术语列表]

$x^2/a^2 + y^2/b^2 = 1$ 公式两边不空空格是行内公式
$ \sum_{k=0}^n k$
   &= 1 + ... + n \
   &= (n(n+1)) / 2 $
公式两边空格是行间公式
```

你可能注意到了 Typst 公式与 \LaTeX 公式有差异。

```
```py
```

```
print("Hello, World!")
```

```
```
```

上述代码块等价于

```
#raw(lang: "py", block: true, "print("Hello, World!")")
```

还有一种特殊的语法糖：

```
#fn(ZZZ)[XXX][YYY] 是 #fn(ZZZ, XXX, YYY) 的语法糖。
```

6.3.2.2 脚本模式

在脚本模式中，用户可以使用类似于 Python 的语法来编写代码。主要有三个最重要的脚本：set、show 和 let。

set 可以设置样式，也就是“为参数设置默认值”的能力。例如：

```
#set heading(numbering: "1.")
```

上述代码设置了标题的编号。

show 用于全局替换和样式设计等，例如：

```
#show "114": "514"
```

上述代码可以把整篇文档中的所有“114”在编译出的文档都显示成“514”。

```
#show heading.where(level: 1): body =>{
  set align(center)
  body
}
```

这里的 heading.where() 有 sqlalchemy 库的影子，大家应该可以意会到它是一个选择器，选择标题中等级为 1 的所有元素。body 则是检索到的原始内容，这里的箭头则是一个函数，指的是接受箭头前面的内容，然后返回修改后的内容，也就是大括号内部的东西。在大括号内部，我们使用 set 调整了样式，使其居中。

let 用于定义变量和函数等，很像它在 JavaScript 中的亲戚。主要有以下几种使用方式：

```
// 存储基本值
#let x = 10
#let name = "Typst"
#let is-active = true
The value of x is #x. // 输出: The value of x is 10
```

```
// 存储内容块
#let warning = [**Warning:** This is important!]
#warning // 输出: **Warning:** This is important!

// 定义函数
#let add(a, b) = a + b
#let greet(name) = [Hello, #name!]
#add(5, 3) // 输出: 8
#greet("Alice") // 输出: Hello, Alice!

// 定义样式
#let emph-style = set text(red, weight: "bold")
#emph-style
This is emphasized text. // 红色加粗

// 多式综合
#let base-style = set text(font: "Helvetica", size: 12pt)
#let title-style = base-style.with(size: 16pt, weight: "bold")
// 这里的*.with()可以扩展或者覆盖原有的样式

#title-style
= This is a title // 使用 Helvetica, 16pt, 加粗
```

特别注意：默认情况下，该方法定义的变量是有类似 C 系的变量作用域的。let 和 show 结合使用，则可以制作出各种各样的模板。

对于更进一步的使用（例如 Touying 和 Pinit 等著名包的使用），我们不做更多介绍了，感兴趣的同学可以自行查找相关资料进行了解。函数文档可以去查看[Typst 官方文档](#)，已经发布的包可以在[Typst Universe](#)上查看，对于中文问题可以查看[Typst 中文社区导航](#)，对了还有 Typst 非官方中文交流群（QQ 群号：793548390），欢迎来交流。

第七章 计算机进阶

我们在初阶课程中讲述的所谓“进阶”内容还是过于基础了，以至于无法称之为真正的“进阶”。因此，在本章中，我们将介绍一些计算机领域的进阶知识和技能，帮助同学们更深入地理解计算机的工作原理和应用。

7.1 Git 进阶

在初阶课程3.5中，我们已经介绍了 Git 的基本使用方法，包括如何创建仓库、提交代码、查看历史记录等。但是，Git 还有许多高级功能和技巧，可以帮助我们更高效地管理代码和协作。

7.1.1 分支管理

有时候我们想同时开发新功能，并且调优以前的代码，这样可能就需要两条线进行开发。这时，分支相关的功能就会很有帮助。Git 的分支功能允许我们在同一个仓库中创建多个独立的开发线，每个分支可以独立地进行提交和修改。

我们可以做如下假设：已经有一个名为 `main` 的分支，并已经有了一系列提交记录 A、B、C。现在，我希望开发一个新的功能，但是不想影响到 `main` 分支上的代码。这时，我们可以创建一个新的分支，例如 `feature`，并在该分支上进行开发。

7.1.1.1 创建和切换分支

可以使用以下命令创建一个新的分支并切换到该分支：

```
git checkout -b feature
```

以上等价于执行

```
git branch feature <commit-hash of C>
git checkout feature
```

如果我现在想要回到 `main` 分支，可以使用以下命令：

```
git checkout main
```

7.1.1.2 分支变基

如果我们已经在 `feature` 分支上进行了多次提交 F、G，同时在 `main` 分支上也有了新的提交 D、E。现在想要将 `feature` 这些提交变基到 `main` 分支上，可以使用以下命令：

```
git rebase main
git checkout main
```

这样会把上述 `feature` 上的三个提交从 `C` 变基到 `E`，变成 `F'` 和 `G'`。我们可以用图解来理解这个过程：

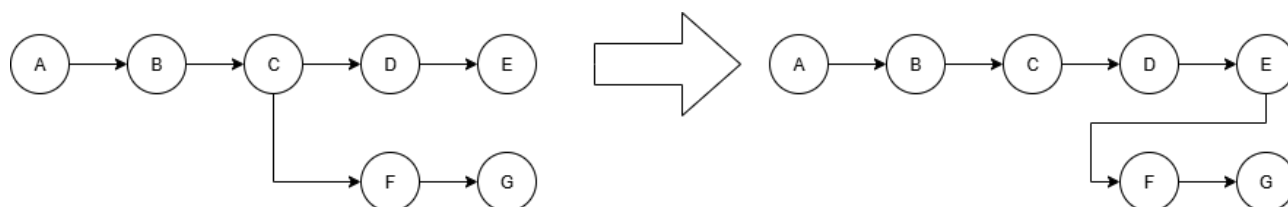


图 7.1: 分支变基示意图

特别注意：变基操作会改变提交的哈希值。

7.1.1.3 合并分支和冲突解决

如果我们想要将 `feature` 分支上的代码合并（不是变基）到 `main` 分支上，可以使用以下命令：

```
git checkout main
git merge feature
```

这时候我们在 `main` 分支上，并试图将 `E` 和 `G` 合并在一起。这时，会自动创建一个特殊的提交 `Merge`，它有两个父提交。之后的提交就会以 `Merge` 为父提交，而不是 `E` 或 `G` 中的任何一个。

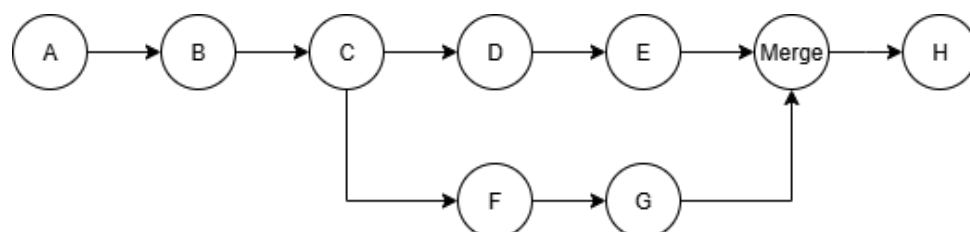


图 7.2: 分支合并示意图

如果这两个提交没有冲突，那么合并会自动完成。但是如果有冲突（例如两个分支涉及到同一行的修改），Git 会提示我们解决冲突。此时，我们不得不手动解决冲突。我们会看到以下内容（或者其英文版本）：

```
自动合并 example1.txt
冲突（内容）：合并冲突于 example1.txt
自动合并失败，修正冲突然后提交修正的结果。
```

此时，我们需要打开冲突的文件，手动解决冲突。Git 会在冲突的地方插入标记，例如：

```
<<<<<< HEAD
这是 main 分支上的内容。
=====
这是 feature 分支上的内容。
>>>>>> feature
```

我们需要手动编辑这个文件，删除这些标记，并保留我们想要的内容。

如果使用 Code 等编辑器，通常会有冲突解决的工具，可以帮助我们更方便地解决冲突。

解决完冲突后，我们需要使用以下命令来标记冲突已解决：

```
git add .
git merge --continue
```

7.1.1.4 删除分支

如果我们已经完成了 feature 分支上的开发，并且已经将其合并到 main 分支上，可以使用以下命令删除该分支：

```
git branch -d feature
```

一般不建议直接删除分支，而是使用 `-d` 选项来删除已经合并的分支。如果分支没有被合并，可以使用 `-D` 选项强制删除。

7.1.1.5 压缩提交

有时候，我们在开发过程中，可能会有很多小的提交，这些提交可能是一些临时的修改或者调试信息。为了保持代码和版本库的整洁，我们可以使用 Git 的压缩提交功能，将多个提交合并为一个提交。这个压缩功能被称作是 **Squash**，但是特别注意：没有 `git squash` 命令。

我们一般只在分支合并的时候使用压缩提交。可以使用以下命令中的一个来压缩提交：

```
git merge --squash feature
```

7.1.2 标签管理

标签（Tag）是 Git 中用于标记特定提交的功能。标签通常用于标记版本发布或重要的里程碑。与分支不同，标签是静态的，不会随着提交而移动。

7.1.2.1 创建标签

可以使用以下命令创建一个标签：

```
git tag v1.0
```

这将创建一个名为 v1.0 的标签，指向当前的提交。如果需要为特定的提交创建标签，可以在命令中指定提交的哈希值：

```
git tag v1.0 <commit-hash>
```

7.1.2.2 查看标签

可以使用以下命令查看所有标签：

```
git tag
```

7.1.2.3 删除标签

如果需要删除一个标签，可以使用以下命令：

```
git tag -d v1.0
```

7.1.3 “摘樱桃”

Cherry-Pick（摘樱桃）操作（也叫挑拣）是指从一些提交中选择一些特定的提交（修改），并将这些提交（修改）应用到当前分支上。这适用于当我们只想要一些特定的提交而不是整个分支的所有提交的时候。

一般，CherryPick 操作很难使用命令行来操作，其复杂程度过高。我们可以使用 VS Code 的自带 Git 视窗或者 GitLens 等工具来进行这个操作。

使用视窗进行挑拣非常方便，我们只需要在提交列表中选择需要的提交，然后右键点击“Cherry-Pick”（汉化应该是挑拣）即可。这样会将选中的提交应用到当前分支上。

7.1.4 远程仓库

很多项目无法只在一台机器上进行开发，往往都需要在远程部署一个仓库（例如 GitHub、GitLab 等，或者公司自建库），然后将本地的代码推送到远程仓库中。这样，我们就可以在不同的机器上从远程仓库中拉取代码，从而保证代码的一致性。

在本节，我们将使用 GitHub 作为远程仓库的示例，介绍如何将本地仓库与远程仓库进行关联、推送和拉取代码。

7.1.4.1 创建仓库

首先，我们需要在 GitHub 上创建一个新的仓库。创建完成后，GitHub 会提供一个远程仓库的 URL，例如：

```
https://github.com/YourName/example.git
```

我们可以在 GitHub 上的仓库页面中找到这个 URL，如图所示。

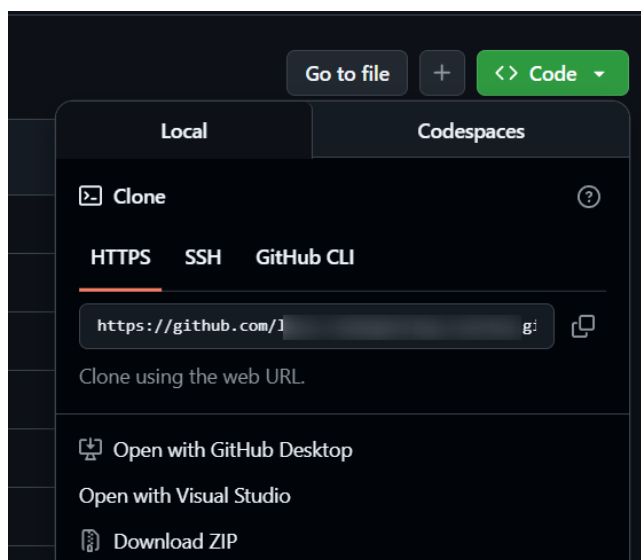


图 7.3: GitHub 上的仓库页面

接下来，我们需要将本地仓库与远程仓库关联。可以使用以下命令：

```
git remote add origin <your-repo-url>
```

这将把远程仓库的 URL 添加为名为 origin 的远程仓库。origin 是习惯上的远程仓库名称。

现在我们要将这个本地仓库的代码推送到远程仓库中。可以使用以下命令：

```
git push -u origin main
```

这里的 -u 选项表示将本地的 main 分支与远程的 main 分支关联起来，以后可以直接使用 git push 和 git pull 命令进行推送和拉取。

如果本地分支名称和远程有区别，（例如本地仓库主要分支是 master，而远程仓库的主要分支是 main），我们可以使用以下命令来推送代码：

```
git push -u origin master:main
```

这将把本地的 master 分支推送到远程的 main 分支。

7.1.4.2 使用仓库

如果你是仓库的使用者，想要从远程仓库中拉取代码（但是本地没有这个仓库），可以使用以下命令：

```
git clone <your-repo-url>
```

这样会在本地创建一个新的目录，并将远程仓库的代码克隆到该目录中。在克隆代码的时候，Git 会自动创建与远程同名的分支，并把它们与远程的 `main` 分支关联起来。

如果你已经有了本地仓库，并且想要将远程仓库的代码拉取到本地，经典的操作是以下命令：

```
git pull origin main
```

直接使用 `git pull` 命令也是可以的，因为我们之前已经使用 `-u` 选项将本地分支与远程分支关联起来了。然而，需要注意的是现代的拉取操作往往不推荐使用 `git pull` 命令，因为它会自动合并远程分支的代码到本地分支，这可能会导致冲突。更推荐的做法是先使用 `git fetch` 命令拉取远程仓库的代码，然后再手动合并：

```
git fetch origin  
git merge origin/main
```

这样可以更好地控制合并过程，避免自动合并带来的问题。

如果你在本地做出了一些修改，想要将这些修改推送到远程仓库，可以使用以下命令：

```
git push origin main
```

直接使用 `git push` 命令也可以。

如果存在某些提交在远程仓库中，而本地仓库没有这些提交，Git 会提示你先拉取远程仓库的代码，然后再推送本地的修改。这是因为 Git 不允许直接推送到远程仓库，除非本地仓库是最新的。如果你确定你不需要远程仓库的提交，可以使用以下命令强制推送本地的修改：

```
git push -f origin main
```

但是请注意，这样会覆盖远程仓库的代码，可能会导致其他工作丢失，因此请谨慎使用。

7.1.5 GitHub 指南

7.1.5.1 这些都是什么东西？

我们打开 GitHub 的一个仓库的时候，映入眼帘的类似这张图片内容。可以看到，这些图片中有很多不同的概念和功能。我们来逐一介绍一下。

在上面的一栏中，我们可以看到以 `code`、`issues`、`pull requests` 等为标题的选项卡。每个选项卡对应着一个功能模块。

- **Code:** 代码模块，显示仓库中的代码文件和目录结构。我们可以在这里浏览代码、下载代码、查看提交历史等。

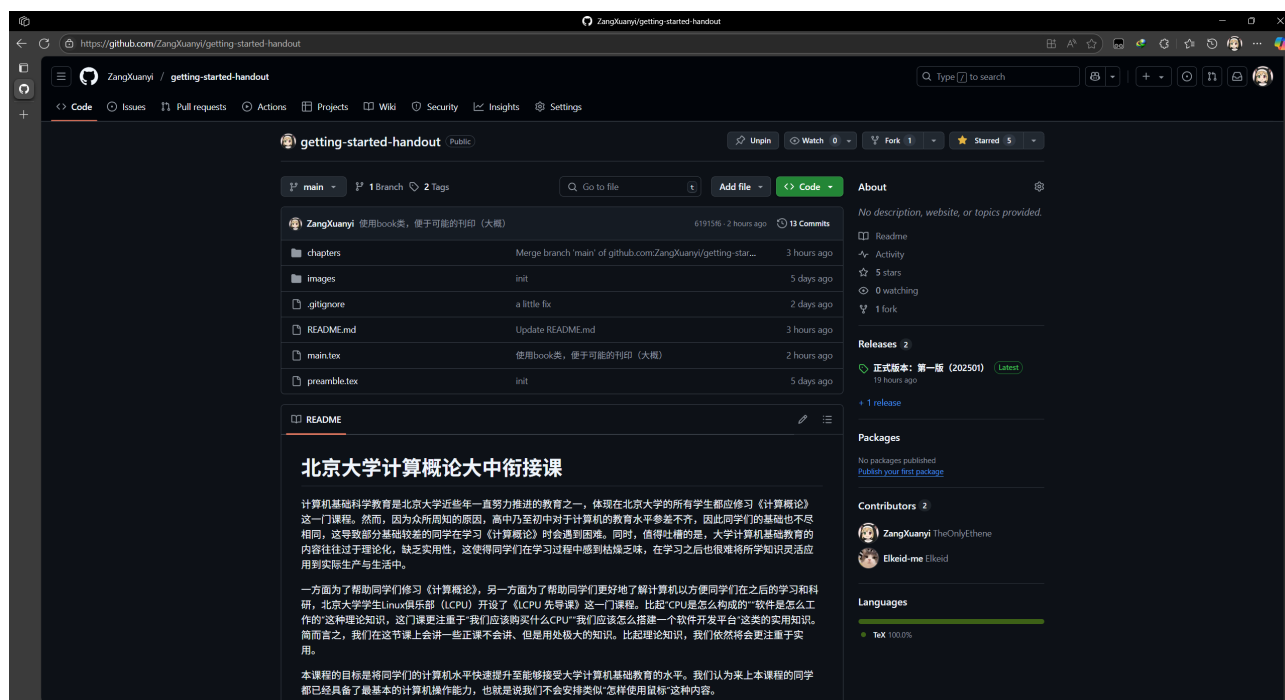


图 7.4: GitHub 仓库页面

- **Issues:** 问题模块，用于跟踪和管理项目中的问题和任务。我们可以在这里创建新的问题、查看已有的问题、评论和解决问题等。在 Gitea 上，问题模块被称为“工单”（Tasks），这与它常用于公司自建库的特点有关。
- **Pull Requests:** 合并请求模块，用于管理代码的合并和审查。我们可以在这里创建新的合并请求、查看已有的合并请求、评论和审查代码等。Pull Request（简称 PR）是 GitHub 和 GitLab 等平台提供的一种代码审查和合并的机制，具体内容可以参考 7.1.6.2 节。
- **Actions:** 自动化模块，用于管理项目的自动化 workflows。我们可以在这里创建新的 workflows、查看已有的 workflows、运行和调试 workflows 等。
- **Projects:** 项目模块，用于管理项目的进度和任务。我们可以在这里创建新的项目、查看已有的项目、添加任务和卡片等。
- **Wiki:** 维基模块，用于管理项目的文档和知识库。我们可以在这里创建新的页面、编辑已有的页面、添加图片和链接等。GitHub Wiki 是 GitHub 提供的一种文档管理工具，可以帮助我们编写和维护项目的说明文档。
- **Security:** 安全模块，用于管理项目的安全性和漏洞。我们可以在这里查看项目的安全报告、修复漏洞、配置安全策略等。
- **Insights:** 洞察模块，用于分析项目的活动情况，例如提交历史、问题和合并请求的统计信息等。我们可以在这里查看项目的活跃度、贡献者的统计信息、代码的质量和覆盖率等。
- **Settings:** 设置模块，用于管理项目的设置和配置。我们可以在这里修改项目的名称、描述、权限等属性。

靠下一行就是仓库的名称，右面是仓库的描述和一些操作按钮。Star 用来标记喜欢的仓库。

库，Fork 用来复制仓库到自己的账户下，Watch 用来关注仓库的更新。

再靠下一行，我们可以看到仓库的分支（Branch）和提交（Commit）信息。分支是代码的不同版本，提交是代码的历史记录。我们可以在这里切换分支、查看提交历史、比较不同分支的差异等。同一行的那个绿色的 Code 按钮是用来下载代码的，可以选择下载为 ZIP 文件或者使用 Git 克隆仓库。我们非常推荐使用 Git 克隆仓库，因为这样可以更方便地管理代码和提交。

页面的左下方部分，在文件目录之下，是仓库的 readme 文件内容。README 文件是仓库的说明文档，通常包含项目的介绍、安装和使用说明、贡献指南等信息。我们可以在这里查看项目的详细信息。

页面右侧的一列是仓库的统计信息，包括提交历史、分支、标签、贡献者等。我们可以在这里查看项目的活跃度、贡献者的统计信息、代码的质量和覆盖率等。同时，我们也可以在这里找到仓库的发行版等信息。

7.1.6 多人协作

成熟的项目往往是由多人协作完成的，因此需要一些规范来管理代码的提交和合并等。GitHub、GitLab 等提供了多种方式来支持多人协作，包括分支管理、代码审查、合并请求等。

7.1.6.1 Fork

Fork 是 GitHub 和 GitLab 等平台提供的一种代码复制和协作的机制。它允许用户将其他人的仓库复制到自己的账户下，从而可以在自己的仓库中进行修改和提交。这样可以使得修改更加方便（主要是防止权限不够），并且可以避免直接修改原仓库的代码。当然，权限足够的情况下，我们往往会直接在原仓库中创建新分支进行修改。

7.1.6.2 Pull Request

Pull Request（简称 PR）是 GitHub 和 GitLab 等平台提供的一种代码审查和合并的机制。它允许开发者在完成某个功能或修复某个问题后，将自己的代码提交到主分支（通常是 main 或 master）之前，先进行代码审查和讨论。

PR 的工作流程通常如下：

1. 开发者 fork（分叉）一个仓库，或者在原仓库中创建一个新的分支。
2. 开发者在自己的分支上进行开发，完成某个功能或修复某个问题。
3. 创建一个 PR，请求将自己的分支合并到主分支。PR 中可以包含对代码的描述、相关问题的链接等信息。
4. 其他开发者可以对 PR 进行代码审查，提出修改意见或建议。
5. 开发者根据审查意见修改代码，并更新 PR。

6. 当 PR 获得足够的审查和批准后，可以将其合并到主分支。通常会有一个维护者或项目负责人来执行这个操作。
7. 合并后，PR 会被关闭，相关的分支可以被删除；也可以保留，以便后续的开发和维护。

7.1.6.3 Lint

在多人协作中，代码风格和规范的一致性非常重要。Lint 工具可以帮助我们检查代码中的潜在问题和不符合规范的地方。常见的 Lint 工具有 ESLint（用于 JavaScript）、Pylint（用于 Python）等。

如果我们在仓库中包含了 Lint 工具的配置文件（例如 `.eslintrc.json` 或 `.pylintrc`），那么在提交代码时，Git 会自动运行 Lint 工具，对代码进行检查。如果代码不符合规范，Lint 工具会给出相应的错误或警告信息。

Lint 工具通常会在 PR 中自动运行，并将检查结果反馈给开发者。开发者可以根据检查结果修改代码，确保代码符合项目的规范。

7.1.6.4 成熟项目的分支管理策略

在成熟的项目中，一般会采用一些分支管理策略来规范分支的使用和合并等。一般说来，同一个仓库中会有以下几种分支：（以下是 Git Flow 的工作管理策略）

- **main/master**: 主分支，通常是代码的稳定版本。一般禁止直接提交代码，只能通过合并其他分支来进行更改。
- **develop/dev**: 开发分支，一般是集成了所有的新功能的基准分支，是开发的主要分支。该分支从 main 分出，最终也要进入 main 分支。对于一些较为轻量级的项目，有时候会直接使用 feature 分支来代替 develop 分支。
- **feature/feature-name**: 功能分支，每个新功能或改进都在独立的分支上进行开发。不同的开发者可以在不同的功能分支上工作，完成后再合并到 develop 分支。在功能完成开发后，通常会删除该分支。
- **hotfix/hotfix-name**: 热修复分支，一般是绕过开发流程，直接从 main 分支分出，修复完成后合并回 main 分支和 develop 分支。热修复分支通常用于修复生产环境中的紧急问题。
- **release/release-name**: 发布分支，一般用于准备发布新版本的代码。该分支从 develop 分出，经过测试和修复后再合并回 main 分支和 develop 分支。发布分支通常用于准备发布新版本的代码。在发布完成后，通常会删除该分支。

除了 Git Flow，还有其他一些分支管理策略，例如 GitHub Flow、GitLab Flow 等。GitHub Flow 是 GitHub 提出的分支管理策略，主要用于快速迭代和持续集成，其开发非常轻量级，一般只有 main/master 和 feature 分支。GitLab Flow 则是 GitLab 提出的分支管理策略，多出了产品分支和预发布分支等，分别用于生产环境和预发布环境。

7.2 密钥进阶

在初阶课程中，我们已经知道了密钥是什么东西，并且知道了在大多数的情况下要使用密钥而不是密码来进行身份验证。但是关于“怎么使用和管理”密钥，则没有进行详细介绍。在本节中，我将会详细地介绍密钥的使用和管理。

7.2.1 SSH 密钥的生成

在 Windows 上，我们需要安装系统功能 OpenSSH Client 来进行密钥的初步使用。在 Linux 和 Mac 上，OpenSSH 通常是预装的。如果没有安装，请自行查找相关资料进行安装。

在安装完成后，我们可以使用以下命令来生成密钥对：

```
ssh-keygen -t rsa -b 4096 -C "<你的邮箱地址>"
```

上述命令会生成一个 RSA 密钥对，密钥长度为 4096 位，并且会在密钥中添加一个注释（通常是你的邮箱地址）。执行该命令后，会提示你输入密钥的保存路径和密码。默认情况下，密钥对会保存在 `~/.ssh/id_rsa` 和 `~/.ssh/id_rsa.pub` 中。

RSA 密钥对是最常用的密钥对之一，不过因为 RSA 密钥对的安全性已经不如以前了，因此现在推荐使用 Ed25519 密钥对。可以使用以下命令生成 Ed25519 密钥对：

```
ssh-keygen -t ed25519 -C "<你的邮箱地址>"
```

生成密钥对后，我们需要将公钥（`id_rsa.pub` 或 `id_ed25519.pub`）添加到远程服务器或服务（例如 GitHub、GitLab 等）的 SSH 密钥列表中。**注意：私钥（`id_rsa` 或 `id_ed25519`）必须保密，绝对不能泄露给任何人！**

如果我们能够直接访问远程服务器，可以使用以下命令将公钥复制到远程服务器上：

```
ssh-copy-id user@remote-server
```

我们也可以手动将公钥复制到远程服务器的 `~/.ssh/authorized_keys` 文件中。我们可以使用记事本或者 code 等编辑器打开公钥文件，复制其中的内容，然后在远程服务器上使用以下命令将其添加到 `~/.ssh/authorized_keys` 文件中。以上方法适用于无法使用 `ssh-copy-id` 命令的情况，例如 Windows 系统。

为了保护私钥的安全，我们可以为私钥设置一个密码。这样，在使用私钥进行身份验证时，需要输入密码才能解锁私钥。可以在生成密钥对时设置密码，也可以在后续使用 `ssh-keygen` 命令修改密码。

设置密码的方式非常简单。在生成密钥对时，系统会提示你输入密码。如果你不想设置密码，可以直接按 Enter 键跳过。

如果你已经生成了密钥对，但没有设置密码，可以使用以下命令为私钥设置密码：

```
ssh-keygen -p -f ~/.ssh/id_rsa
```

实际上如果保密需求不是非常高的话，我们可以不设置密码。因为使用密钥除了安全性以外，最大的好处是可以免去每次连接远程服务器时输入密码的麻烦。而如果设置了密码，则每次连接远程服务器时都需要输入密码，这样就失去了使用密钥的便利性。

7.2.2 密钥的使用

在生成密钥对并将公钥添加到远程服务器或服务后，我们就可以使用密钥进行身份验证了。使用密钥进行身份验证的方式与使用密码类似，只不过需要指定私钥文件。

7.2.2.1 连接到远程服务器

可以使用以下命令连接到远程服务器：

```
ssh -i ~/.ssh/id_rsa user@remote-server
```

如果你使用的是 Ed25519 密钥对，则需要将 `id_rsa` 替换为 `id_ed25519`。

如果你已经将私钥添加到 SSH Agent（实际上这确实是更一般的情况）中，可以直接使用以下命令连接到远程服务器：

```
ssh user@remote-server
```

7.2.2.2 Git 托管

GitHub 的有两种托管代码的方式：HTTPS 和 SSH。HTTPS 是通过用户名和密码进行身份验证，而 SSH 是通过密钥进行身份验证。我们建议使用 SSH 进行身份验证，因为它更加安全和方便，且无需忍受网络代理的折磨。

我们需要将公钥添加到 GitHub 的 SSH 密钥列表中。可以在 GitHub 的设置页面中找到 SSH 密钥列表，然后点击“添加 SSH 密钥”按钮，将公钥粘贴到文本框中。

如果你使用的是 Windows 系统，可能需要将公钥转换为 OpenSSH 格式。可以使用以下命令将公钥转换为 OpenSSH 格式：

```
ssh-keygen -i -f ~/.ssh/id_rsa.pub
```

添加公钥后，我们就可以使用 SSH 进行身份验证了。在某些情况下，我们可能需要手动指定使用的哪一个密钥文件。可以使用以下命令将 SSH 密钥添加到 SSH Agent 中：

```
ssh-add ~/.ssh/id_rsa
```

这样可以免去每次连接远程服务器时指定密钥文件的麻烦。

7.2.3 使用 VS Code 建立 SSH 连接

除了使用终端建立 SSH 连接到远程服务器以外，还可以使用一些其他的工具来建立 SSH 连接。这时候我们还要请出那位大神：VS Code（怎么哪都有你）。

VS Code 提供了一个名为 Remote-SSH 的扩展，可以帮助我们通过 SSH 连接到远程服务器，并在远程服务器上进行开发。这样，可以在 SSH 连接中使用一个很方便的图形化界面，以进行和 Windows 相似的便捷操作。

安装 Remote-SSH 扩展后，我们可以在 VS Code 的界面找到远程连接的选项，一般是左下角的蓝色按钮，图标类似这个 \leq 数学符号。点击这个按钮后，会弹出一个菜单，点选“连接到主机”选项，会让你输入 `userhost` 类似的远程服务器地址。输入完成后，如果是一个新的远程服务器，Code 会让你把它加入到已知主机列表中，用户可以视情况添加到系统配置文件或者其他的配置文件中。

然后，Code 会弹出一个新的窗口，试图连接到远程服务器，可能会要求你输入远程服务器的密码和系统类型等信息。连接完成后，就可以在远程服务器上进行开发了。此时，Code 会在左侧的资源管理器中显示远程服务器的文件系统（当然你需要打开一个文件夹）。

在 Code 中，如果不是用终端而是用 Code 的图形界面来打开新的文件夹，那么每一次打开文件夹都会重新进行一次身份验证。如果你使用的是密码，则需要反复输入，非常麻烦。这时我们一定要尽可能地使用密钥进行登录。

7.3 Windows 的文件管理和自动化操作

在计算机上，我们经常需要对文件进行管理和自动化操作。文件管理包括文件的增删改查、移动复制等操作，而自动化操作则是通过脚本或工具来实现对文件的批量处理和自动化任务。

本段将以 Windows 为例，介绍一些常用的文件管理和自动化操作的相关知识。有关 Linux 文件系统的相关知识将会在下一章节介绍。

7.3.1 文件系统基础知识

文件系统是操作系统用来管理存储设备上数据的结构和方法。它定义了如何在存储设备上组织、存储和访问文件和目录。文件系统的主要功能包括文件的增删改查等操作。

对于一个文件而言，文件系统会为其分配一个唯一的标识符（通常是文件名），并将其存储在一个目录结构中。为了方便理解，我们可以将文件系统理解成一个巨大的档案馆，每个文件就像是档案馆中的一份档案，目录则有些像书架、文件夹等存储档案的工具或者设备，而目录结构则是档案馆中的分类系统。

对于任何文件或者目录，都有一个唯一的路径（Path）来标识它在文件系统中的位置。路径是由目录和文件名组成的字符串，通常使用斜杠（/）或反斜杠（\）作为分隔符。在 Windows

系统下的一个路径示例是：`D:\dir\file.txt`。这种从盘符（根）开始的路径结构叫做**绝对路径**。

在有些时候，使用绝对路径并不合适。同样以档案举例子，假设某档案要求参阅同一档案袋里的某一份其他档案。这时我们如果使用绝对路径记录的话，需要记录整个档案的路径，例如“辽宁-张三-身份证”依赖于“辽宁-张三-出生证明”。现在张三移居北京，这个档案袋也整体移到北京。如果张三的身份证依赖的出生证明路径依然是“辽宁-张三-出生证明”，这时就会找不到这个东西了。

为了解决这一问题，我们引入了**相对路径**的概念。相对路径是相对于当前目录的路径，不需要从根目录开始。例如，如果当前文件是 `D:\dir\file.txt`，那么文件 `D:\dir\file2.txt` 的相对路径就是 `file2.txt`，而文件 `D:\dir\dir1\file3.txt` 的相对路径就是 `dir1\file3.txt`。

在一些比较严谨的文档中，你也可以看到相对路径的写法是 `./file2.txt`，其中 `./` 表示当前目录。这在命令行中常用，尤其是在使用命令行执行可执行文件的时候只能这样写，防止你误执行了系统目录下的同名文件。

在 Python 中，你甚至可以看到相对路径的写法是 `.file2`，其中 `.` 表示当前目录。这种写法在某些情况下也可以使用，但是大多数时候这个点（`.`）开头的文件指的是隐藏文件，我们不推荐在除了 Python 之外的场合使用这种写法。

有时候相对路径需要跨目录访问，这需要“向上一级”操作。在这种情况下，我们可以使用两个点（`..`）来表示上一级目录。例如，如果当前文件是 `D:\dir\file.txt`，那么文件 `D:\dir2\file4.txt` 的相对路径就是 `..\dir2\file4.txt`。

7.3.2 Windows 的文件系统结构

Windows 使用的是 NTFS 文件系统，基于驱动器（Drive），每个驱动器都有一个盘符（例如 C:、D: 等）。每个驱动器既可以是一个物理存储设备，也可以是一个存储设备的某一分区（主分区或者逻辑分区）。Windows 的文件系统结构是树形结构，每个驱动器下都有一个根目录（例如 C:\），根目录下可以有多个子目录和文件。

在 Windows 下，路径使用反斜杠（\）作为分隔符。例如 `D:\dir\file.txt`。尽管如此，Windows 也支持使用正斜杠（/）作为分隔符，两者也可以混合使用。不过我们建议在 Windows 下使用反斜杠（\）作为分隔符，以保持一致性。

Windows 中有如下一些常用的文件夹：

- **C:\Windows**：Windows 操作系统的核心文件夹，包含系统文件和驱动程序。
- **C:\Program Files**：安装的应用程序的默认文件夹，通常包含 64 位应用程序。
- **C:\Program Files (x86)**：安装的 32 位应用程序的默认文件夹。
- **C:\Users**：用户文件夹，包含每个用户的个人文件和设置。
- **C:\Downloads**：下载文件夹，默认存储从互联网下载的文件。

7.3.3 文件的默认打开方式

在 Windows 中，每种文件类型都有一个默认的打开方式。当我们双击一个文件时，系统会根据文件的扩展名（例如.txt、.jpg 等）来确定使用哪个应用程序打开该文件。

一般情况下，除非你知道你在干什么，否则不要随便修改文件的默认打开方式。但是部分软件会改变文件的默认打开方式，例如安装了某个文本编辑器后，可能会将所有的文本文件（.txt）默认打开方式改为该编辑器。

我们非常建议同学们在安装软件时，仔细阅读安装向导中的选项，避免不必要的修改；同时，尽可能不要安装多个同类软件，以免造成文件默认打开方式的混乱。

7.3.4 高效整理与搜索文件

在 Windows 中，我们可以使用文件资源管理器（File Explorer）来浏览和管理文件。文件资源管理器提供了多种视图模式（例如列表视图、详细信息视图等），可以帮助我们更高效地浏览和整理文件。

我们可以使用右键菜单来对文件进行各种操作，例如复制、移动、重命名、删除等。除此之外，在文件夹菜单下可以找到排序方式、分组方式等选项，可以帮助我们更高效地整理文件。

我们在整理文件的时候，要遵循一定的命名规范。例如，文件名应该简洁明了，能够清楚地表达文件的内容；文件夹名应该具有层次性，能够清晰地表示文件的分类和组织结构。同时，由于各种各样的兼容原因，文件和文件夹名中不要包含特殊字符，并且尽量避免使用空格。

7.3.5 搜索工具

在 Windows 上，我们可以使用文件资源管理器的搜索功能来查找文件。只需要在文件资源管理器的搜索框中输入关键词，系统就会自动搜索当前文件夹及其子文件夹中的文件。这个搜索的问题是速度非常缓慢，尤其是在文件数量较多的情况下（例如全盘搜索）。

如果我们需要更高级的搜索功能，可以使用第三方搜索工具，例如 Everything、Listary 等。这些工具可以提供更快、更准确的搜索结果，并且支持多种搜索条件和过滤器。

7.3.6 自动化脚本

7.3.6.1 批量处理文件示例

一般情况下，整理相机里的照片时，我们会将照片按照日期和地点等信息进行分类。但是如果照片数量过多，手动逐个重命名会非常繁琐。此时，我们可以使用自动化脚本来批量处理文件。

在 Windows 上，我们可以使用批处理脚本（Batch Script）或 PowerShell 脚本来实现自动化操作。批处理脚本是一种简单的脚本语言，可以通过编写一系列命令来实现对文件的批量处理。不过 PowerShell 脚本很多新手并不熟悉，批处理脚本风格又太老，因此我们推荐使用 Python 的 `os` 库来实现自动化操作。

一个典型的示例是：

```
import os
for filename in os.listdir("."):
    if filename.endswith(".jpg"):
        os.rename(filename, "pic_" + filename)
```

使用以上脚本，我们可以将当前目录下所有的.jpg 文件批量重命名为 `pic_` + 原文件名的形式。对以上代码进行相关修改，则可以进行其他的批量处理操作，例如移动文件、删除文件、批量添加前后缀、按照规则排序等。

如果不想保留原文件名，而是使用“照片 (1)”这样的格式，就不必写脚本了。可以直接使用 Windows 的批量重命名功能。只需要选中所有的照片，右键点击，选择“重命名”，然后输入新的文件名（例如“照片”），系统会自动将所有选中的照片重命名为“照片 (1)”、“照片 (2)”等格式。

7.3.6.2 定时自动备份脚本示例

我们可以使用 Python 的 `shutil` 库来实现定时自动备份脚本。以下是一个简单的示例：

```
import shutil
import datetime

# 备份文件夹到指定位置
backup_dir = f"backup_{datetime.date.today()}"
shutil.copytree("我的文档", backup_dir)
```

以上脚本会将“我的文档”文件夹备份到当前目录下，并以“`backup_YYYY-MM-DD`”的格式命名备份文件夹。我们可以将该脚本设置为定时任务，定期自动执行备份操作。

7.3.6.3 更进阶的自动化

在 Windows 中，我们可以使用任务计划程序（Task Scheduler）来设置定时任务。任务计划程序允许我们创建和管理定时任务，可以设置任务的触发条件、执行时间等。

要创建一个定时任务，可以按照以下步骤操作：

1. 打开任务计划程序（可以在开始菜单中搜索“任务计划程序”）。
2. 点击“创建基本任务”。
3. 输入任务的名称和描述。

4. 设置触发条件，例如每天、每周等。
5. 设置操作，例如运行脚本或程序。
6. 完成设置，保存任务。

通过任务计划程序和自动化脚本，我们可以实现定时自动备份、定时清理临时文件等自动化操作。

7.3.6.4 跨设备同步

在现代的工作环境中，我们经常需要在多台设备之间同步文件。Windows 提供了多种方式来实现在跨设备同步，例如使用 OneDrive、Google Drive 等云存储服务。当然，以上云存储服务需要支付一定的费用才能获得更多的存储空间。

作为替代，我们可以使用 SyncThing 等开源工具来实现跨设备同步。SyncThing 是一个分布式的文件同步工具，可以在多台设备之间自动同步文件，而无需依赖云存储服务。

第八章 开玩 Linux

很多人是因为迫不得已而使用 Linux（最常见的是上 ICS）。更深入的想一想，还有没有其他使用 Linux 的原因呢？

对于 Windows 等带图形化界面操作系统，我们所访问的其实是设计者抽象出的交互逻辑。但对于高效的系统，自底向上的彻底理解和掌握是高效使用系统的必备途径。我们得以更深入洞悉文件和文件之间的联系，获得系统更高的主动权。同时，以最小化的人机接口访问能把足够多的资源投入至计算，获得最高的资源利用率。Linux 正是带着这样的思想而诞生的。

因此，学习 Linux，我们需要掌握：

1. 对这一套思想有充分理解、能顺利玩一些玩具
2. 使用现有的工具操作命令行、把他人准备好的软件运行起来
3. 创造新的工具

让我们开始吧！

8.1 获取 Linux

不管怎么说，Linux 最终还是一个操作系统。我们最终还是先要获取一个基于 Linux 的系统当玩具：不摸一摸，知道用起来舒不舒服，又了解它干啥呢？

8.1.1 CLab

我们在这一讲推荐大家直接去 [CLab](#) 注册一个账号，根据上面的指南连接到虚拟机。这样，你就可以在不破坏自己的系统的情况下，体验 Linux 的魅力了。

当然，这种情况下，我们还可以顺带着复习一下密钥的相关用法。

8.1.2 实机安装

如果我们有一台不怎么重要的机器和一个 U 盘，可以利用 U 盘在这台机器上面安装 Linux。你可以选择任意的发行版进行安装。

8.1.2.1 Ubuntu

对于 Linux 新手，我们建议安装 Ubuntu 系统，因为它有着和 Windows 类似的图形化界面，且有着丰富的社区支持和文档。以下是一个简要步骤：

1. 获取 Linux 发行版的 ISO 镜像文件。可以从（[Ubuntu 官网](#)）下载最新版本的 Ubuntu ISO 镜像。

2. 准备一个 U 盘，至少 8GB 容量。将这个镜像文件写入 U 盘。可以使用工具如 Ultra ISO、Rufus（Windows）或 Etcher（跨平台）来完成这个操作。
3. 将 U 盘插入目标机器，重启电脑并进入 BIOS 设置，将 U 盘设置为首选启动设备。
4. 保存设置并重启，系统将从 U 盘启动，进入 Ubuntu 安装界面，并按照提示进行安装。我们建议选择“安装 Ubuntu”选项，并在安装过程中选择“擦除磁盘并安装 Ubuntu”选项（注意，这将删除磁盘上的所有数据，请确保备份重要数据）。
5. 安装完成后，重启电脑，拔出 U 盘，系统将进入 Ubuntu 桌面环境。

对于硬盘容量较大且熟悉分区的学生，可以使用磁盘分区工具自己划出一个分区，并将 Ubuntu 安装在这个分区上。这样可以保留原有的操作系统，并在需要时切换到 Ubuntu。在这种情况下我们一般使用 grub 引导程序来管理多系统启动。

8.1.2.2 其他发行版

如果你对 Linux 有一定了解，或者想尝试其他发行版，可以选择其他发行版进行安装。对于此类学生，我们非常推荐使用 Arch Linux，因为它提供了一个非常灵活和可定制的环境，适合有一定 Linux 基础的用户。

同时，安装 Arch Linux 也是一个很好的学习 Linux 的机会，因为它的安装过程需要用户从头手动配置系统，这样可以更深入地了解 Linux 的工作原理。

8.1.3 使用虚拟机

使用虚拟机也是一个很好的选择。通过虚拟机，我们可以在现有的操作系统上运行 Linux 或者其它系统，而不需要重新安装或配置硬件。

一般我们使用的虚拟机软件有 VirtualBox、VMware 等。不同的虚拟机有着不同的特点和使用方法，但是总体而言在虚拟机中安装一个 Linux 发行版的步骤与在实机上安装类似（只是不需要设置 BIOS 和 U 盘启动等了）。

8.1.4 WSL

WSL 是在 Windows 上使用 Linux 的最佳方式之一。它允许用户在 Windows 上运行 Linux 发行版，并提供了与 Linux 相似的命令行环境。WSL 的安装和配置非常简单，只需要在命令行中运行以下命令即可：

```
wsl --install
```

WSL 提供了与 Linux 相似的命令行环境，并且可以直接访问 Windows 文件系统。例如：

```
cd /mnt/c/Users/YourUsername/Documents
```


上述命令会在 Linux 环境中查看 Windows 的文档目录。这种跨系统的文件访问方式非常方便，可以让用户在 Windows 和 Linux 之间无缝切换。这样，我们就可以在 Windows 上使用 Linux 的命令行工具和开发环境了。

8.2 Linux 的基本操作

得了，来了啥也别说，先跑个小火车：

```
sudo apt update
sudo apt install -y sl
sl
```

执行以上命令，我们就可以看到一个小火车在终端上跑过去了。

`sl` 是一个玩具软件，它的全称是 `Steam Locomotive`，是一个在终端上显示火车动画的程序。这不仅能够展示 ASCII 艺术的魅力，还能矫正打错的 `ls` 命令（列出目录内容）。

`apt` 是 Debian 及其衍生发行版（如 Ubuntu）中用于管理软件包的工具。它可以用来安装、更新和删除软件包。

于是我们就能看到，一个程序就这么跑起来了。

让我们看看上面内容是怎么发生的。我们刚刚输入的命令大概是这样的形式：

```
程序 子命令 [选项] [对象]
```

其中，除了子命令，选项和对象都是可选的。我们就用第二个命令来举例说说：`apt` 是程序，`install` 是子命令，`-y` 是选项，`sl` 是对象。

看起来很明显。

那么，我们怎么知道有什么程序，我们又怎么使用它们呢？对于第一个问题，我们可以通过搜索来解决；对于第二个问题，我们可以通过手册来解决。

- `<program> -h`：这是最简单的方式，直接查看程序的帮助信息。通常会列出所有可用的子命令和选项。
- `man <program>`：这是查看程序手册的方式。手册通常会提供更详细的信息，包括子命令的用法、选项的含义等。但是这个手册可能会比较长，需要耐心阅读。
- `tldr <program>`：这个命令会给出一些常用的命令示例和简要说明。当然，`tldr` 需要自己安装，当然，安装方法和 `sl` 类似。

8.3 Linux 的文件系统

好的，我们刚刚已经知道怎么使用 Linux 了。接下来，我们来看看 Linux 的文件系统。我们不会涉及到任何复杂的概念，只会介绍一些最基本的内容。

思考以下问题：我们刚刚确实输入了 `sl` 命令，但是我们并没有输入 `sl` 的路径。那这个 `sl` 到底在哪里呢？我们怎么知道它在哪里？（其实我们知不知道真无所谓）终端又怎么知道它在哪里？小火车又是怎么跑起来的？

为了解决这一问题，计算机前辈们发挥了聪明才智：只要把所有的东西都归纳为一个概念，那么不就可以方便的管理了吗？于是，文件系统就诞生了。

Unix 和 Linux 认为，所有的东西都是文件。文件系统就是用来管理这些文件的。这时候，我们就可以使用同样的方式来对所有的东西进行操作了。但是我们又出现了其他问题：

1. 文件怎么组织？
2. 文件是谁的？怎么反映不同类型的特性？
3. 文件怎么相互联系？

接下来我们将会逐个回答以上问题。

8.3.1 文件的组织

我们在上一章中提到，Windows 系统下，物理先于文件存在，所以有盘符一说。但是 Linux 不这么认为：Linux 认为什么都是文件。于是，Linux 的根目录/就成了所有文件的起点。所有的文件都在这个根目录下。

于是，我们就能够通过目录来解决这一问题。目录是一个特殊的文件，它可以包含其他文件和子目录。我们可以使用 `ls` 命令来查看当前目录下的文件和子目录，也可以使用 `cd` 命令来切换目录。

Linux 的目录结构和 Windows 极其相似，只是没有盘符，并且使用/作为分隔符，而不是\。我们可以使用/来表示根目录，使用..来表示上一级目录，使用./来表示当前目录。需要注意的是，Linux 的文件系统是区分大小写的，所以/home和/Home是两个不同的目录。

Linux 还有一个重要的目录：用户的家目录。每个用户都有一个家目录，通常位于/home/username。在这个目录下，用户可以存放一些配置文件。我们可以使用 `~` 来表示当前用户的家目录。我们不推荐把自己的杂七杂八文件放在根目录或者家目录下，而是放在家目录的子目录下。这样可以更好地组织文件，并且避免与系统文件冲突。

我们在上一章提到，处于安全性考虑，对于可执行文件，如果没有提供路径，系统会在一些特定的目录下查找。因此假如有一个可执行文件 `hello`，我们应当使用 `./hello` 来运行它，而不是直接使用 `hello`。

如果我们想要在任何地方都能运行这个程序，我们可以把它加入 PATH 环境变量中去。PATH 是一个环境变量，它包含了一些目录的路径，系统会在未提供路径时去这些目录下查找可执行文件。我们可以使用 `echo $PATH` 命令来查看当前的 PATH 变量。

8.3.2 文件是谁的，有什么属性

作为多机系统，Linux 中文件如果对每个访问者都相同，那就没有安全性可言了。Linux 的做法是，抽象出了“用户”这个实体（其实就是在 `/etc/passwd` 里面定义的一行 UID 和用

户名的对应而已)。为了方便用户的文件共享,同时抽象出了组(Group)的概念,代表一组互相信任的用户。

对文件的基本操作有读、写、执行三种。我们使用权限位 4、2、1 来表示这三种操作。每个文件都有三个权限位,分别对应所有者、组和其他用户。我们可以使用 `ls -l` 命令来查看文件的权限。

举例: 750, 表示所有者有读、写、执行权限 ($7=4+2+1$), 组有读和执行权限 ($5=4+1$), 其他用户没有任何权限 (0)。

我们可以使用 `chmod` 命令来修改文件的权限。例如, `chmod 777 file.txt` 将会把 `file.txt` 的权限设置为 777。

注意: 修改文件权限需要有相应的权限, 否则会报错。同时, 不要执行这类抽象的命令: `chmod 777 /`, 这会导致系统无法正常工作。

所以可执行文件并不是因为这个文件本身有什么特别, 而是这个文件被你赋予了可执行的性质。一个简单的文本文件也可以被加上可执行的权限, 也可以发挥操作其他文件的作用。

例如, 如果你会写 Python 的话, 写一个从输入读取 2 个数字, 输出他们和的程序, 输出结果到控制台。在本地跑起来这个程序之后, 把 `#!/usr/bin/env python3` 放在脚本的第一行 (这个特殊的一行叫 Shebang)。给这个脚本加上可执行的属性, 然后直接运行这个文本!

8.3.3 文件的联系

文件间的联系, 主要是通过文件系统的链接来实现的。Linux 中有两种链接: 硬链接和软链接。硬链接是指在文件系统中, 一个文件可以有多个文件名, 存在于多个位置, 但是文件系统中只有一份文件副本, 所有链接均指向这一副本。删除其中一个文件名并不会影响文件内容, 只有所有位置下的文件链接均被删除时, 此文件副本才会被最终移除。软链接是指一个文件名指向另一个文件名, 删除原文件名会影响软链接的有效性。

硬链接和软链接的区别在于, 硬链接是文件系统的一个特性, 而软链接是文件系统的一个单独的文件。硬链接只能在同一个文件系统下, 而软链接可以在不同文件系统下。硬链接不能链接目录, 而软链接可以链接目录。

8.4 Linux 的进一步使用

8.4.1 root 权限的配置

root 用户是超级用户, 拥有着 Linux 系统内最高的权限, 在终端内使用 `su` 命令即可以超级用户开启终端, root 用户的权限最高, 而其他账户则可能会有以能以超级用户身份执行命令的授权 (可以类比 Windows 中的管理员权限), 但即使是拥有授权的账户在终端输入的命令也不会以超级用户身份执行, 如果要以超级用户的身份运行则需要在此命令前加 `sudo`。

第一种情况, 如果你所选择的发行版在安装过程中没有设置 root 密码的环节 (如 Ubuntu), 则新创建的用户会拥有管理员权限, 一般不需要使用 root 账户, 直接使用 `sudo` 命令即可。

第二种情况，如果你所选择的发行版在安装过程中已经设置了 root 密码，但是自己的账户并没有管理员权限（如 Debian），为了用起来方便一般会用 root 账户给自己的账户添加管理员权限，具体操作如下（\$ 号后的为输入的命令）：

```
yourusername@yourcomputer$ su
root@yourcomputer$ /usr/sbin/usermod -aG sudo yourusername
```

前者表示切换至 root 账户，后者表示为你指定的账户添加管理员权限。有些发行版中 wheel 组表示有 sudo 权限的用户组（例如 Arch），也可以用 visudo 编辑 sudo 配置。

8.4.2 软件的安装及其源的配置

如果你的系统在安装的时候已经选择过了国内源则忽略，否则默认源来自于国外。从国外的服务器更新软件包会很慢，可以根据自己系统的版本自行搜索匹配的源并更换。具体参考[北大开源镜像站的帮助文档](#)。

以采用 apt 包管理器为例，更新源后需要重新更新软件索引，请执行以下操作：

```
sudo apt-get update
sudo apt-get upgrade # 如果需要升级软件包
```

如果你需要安装软件包，可以使用以下命令：

```
sudo apt-get install <package-name>
```

如果想要卸载软件包，可以使用以下命令：

```
sudo apt-get remove <package-name>
```

有时候，我们不得不使用一些其他安装方式，例如从 *.deb 包安装。对于这些情况，我们可以使用以下命令：

```
sudo dpkg -i <package-name>.deb # 安装 .deb 包
sudo apt-get install -f # 修复依赖问题
```

有时，软件自带安装脚本，我们直接运行这些脚本即可。

特别注意：我们请尽可能地使用包管理器来安装软件，而不是直接下载二进制文件或源码编译安装。包管理器可以自动处理依赖关系，并且可以方便地进行软件的更新和卸载。如果一定要手动安装软件，请确保你了解该软件的安装过程和依赖关系，并尽可能在虚拟环境或容器中进行测试，以避免对系统造成不必要的影响。

8.4.3 再临终端命令行

在表3.1中，我们初步认识了一些常用的命令。接下来我们会对它们进行一定的扩展。

8.4.3.1 列出类命令

`pwd` 命令用于显示当前工作目录的绝对路径。它可以帮助我们了解当前所在的位置。

`ls` 命令用于列出目录中的文件和子目录。我们可以使用一些选项来改变输出的格式：

- `-l`: 以长格式列出文件和目录的详细信息，包括权限、所有者、大小和修改时间等。
- `-a`: 列出所有文件和目录，包括以点号开头的隐藏文件。
- `-h`: 以人类可读的格式显示文件大小，例如将字节转换为 **KB**、**MB** 等。
- `-R`: 递归地列出子目录中的文件和目录。
- `-t`: 按修改时间排序，最新的文件排在最前面。

8.4.3.2 文件系统操作类

`cd` 命令用于切换当前工作目录。

`mkdir` 命令用于创建新目录。我们可以使用一些选项来改变创建目录的方式：

- `-p`: 递归地创建多级目录，如果上级目录不存在则一并创建。
- `-v`: 显示创建目录的详细信息。
- `-m`: 设置新目录的权限。
- `-Z`: 设置 SELinux 上下文。

`touch` 命令用于创建新文件或更新现有文件的修改时间。我们可以使用一些选项来改变创建文件的方式：

- `-a`: 只更新访问时间。
- `-m`: 只更新修改时间。
- `-c`: 如果文件不存在则不创建。
- `-t`: 设置文件的时间戳。
- `-r`: 使用指定文件的时间戳。

`rm` 命令用于删除文件或目录。我们可以使用一些选项来改变删除的方式：

- `-r`: 递归地删除目录及其内容。
- `-f`: 强制删除文件或目录，不提示确认。
- `-i`: 在删除前提示确认。
- `-v`: 显示删除的详细信息。

`rmdir` 命令用于删除空目录。

`cp` 命令用于复制文件或目录。我们可以使用一些选项来改变复制的方式：

- `-r`: 递归地复制目录及其内容。
- `-f`: 强制覆盖目标文件。
- `-i`: 在覆盖前提示确认。
- `-v`: 显示复制的详细信息。
- `-u`: 只在源文件比目标文件新时才进行复制。

`mv` 命令用于移动或重命名文件或目录。我们可以使用一些选项来改变移动的方式：

- **-f**: 强制覆盖目标文件。
- **-i**: 在覆盖前提示确认。
- **-v**: 显示移动的详细信息。
- **-u**: 只在源文件比目标文件新时才进行移动。

ln 命令用于创建链接。我们可以使用一些选项来改变链接的方式:

- **-s**: 创建软链接（符号链接）。
- **-f**: 强制覆盖目标链接。
- **-i**: 在覆盖前提示确认。
- **-v**: 显示链接的详细信息。
- **-T**: 将目标视为一个普通文件，而不是目录。

tar 命令用于打包和解包文件。我们可以使用一些选项来改变打包和解包的方式:

- **-c**: 创建一个新的归档文件。
- **-x**: 从归档文件中提取文件。
- **-f**: 指定归档文件的名称。
- **-v**: 显示详细的操作信息。
- **-z**: 使用 **gzip** 压缩或解压缩归档文件。
- **-j**: 使用 **bzip2** 压缩或解压缩归档文件。
- **-J**: 使用 **xz** 压缩或解压缩归档文件。
- **-p**: 保留文件的权限和时间戳。
- **-C**: 切换到指定目录后再进行打包或解包。

8.4.3.3 其他命令

cat 命令用于连接文件并打印到标准输出。我们可以使用一些选项来改变输出的方式:

- **-n**: 为每一行添加行号。
- **-b**: 为非空行添加行号。
- **-s**: 压缩连续的空行。
- **-E**: 在每行末尾显示 **\$** 符号。
- **-T**: 将制表符显示为 **^I**。
- **-v**: 显示不可见字符。
- **-A**: 显示所有不可见字符，包括空格和制表符。
- **-e**: 等同于 **-vE**，显示不可见字符并在行末添加 **\$** 符号。

echo 命令用于在终端输出文本。我们可以使用一些选项来改变输出的方式:

- **-n**: 不在输出末尾添加换行符。
- **-e**: 启用转义字符的解释，例如 **\n** 表示换行，**^I** 表示制表符。
- **-E**: 禁用转义字符的解释。
- **-c**: 不输出任何内容。

- **-C**: 将输出内容转换为大写字母。
 - **-l**: 将输出内容转换为小写字母。
 - **-a**: 将输出内容转换为首字母大写字母。
 - **-s**: 将输出内容转换为首字母小写字母。
 - **-p**: 将输出内容转换为首字母大写字母，并将其他字母转换为小写字母。
- ps** 命令用于显示当前运行的进程。我们可以使用一些选项来改变输出的方式:
- **-e**: 显示所有进程。
 - **-f**: 以全格式显示进程信息，包括父进程 **ID**、用户等。
 - **-l**: 以长格式显示进程信息。
 - **-u**: 显示指定用户的进程。
 - **-p**: 显示指定进程 **ID** 的进程。
 - **-o**: 自定义输出格式。
 - **-H**: 以树形结构显示进程之间的关系。
 - **-j**: 以作业控制格式显示进程信息。
 - **-x**: 显示所有进程，包括没有控制终端的进程。

第三部分

走向开发

第九章 实用主义编程

自本章开始，笔者默认同学们已经学会了至少一门编程语言的语法。

现在同学们的前置知识已经充分，是时候开始正式走向开发了。

对于一些同学而言，即使他们走向工作岗位或者科研岗位之后，他们写出的代码依然难以阅读，更难以进行长期维护。从某种程度上说，**代码是写给人看的**，机器只是顺便运行，按理说写成什么样子都可以；但是如果可读性太差的话，估计未来的自己都会抽自己几巴掌——完全看不懂。因此，我们将在这里介绍一下怎么才能真正写出来一些**真正可以交付**的代码。

9.1 写代码的基本素养

代码风格（码风）是指代码的书写规范和格式化方式。良好的代码风格可以提高代码的可读性和可维护性，使得其他人（包括未来的自己）能够更容易地理解和修改代码。一般情况下，我们会遵循一些通用的代码风格规范，例如 **Google C++ Style Guide** 或 **PEP 8**（**Python Enhancement Proposal 8**）等。而在团队协作的时候，我们则尽可能保证码风和团队的码风一致。

9.1.1 通用代码风格指南

尽管不同语言和项目有不同的风格，但以下几点是相对普适的、值得注意的基本素养：

- **缩进**：使用空格或制表符进行缩进，通常使用 2 个空格或 4 个空格或 1 个制表符。关键是 **不要混用空格和制表符**。
- **命名**：使用有意义的变量名（`int user_count`）和函数（`calc_total_price()`），不要使用单个字母（`a1`）、过度缩写（`cal()`）或无意义的名称（`tmp1`）。此外，在同一个项目中，对于同一种程序实体（例如，类、函数、变量），应当采用统一的命名风格。例如大驼峰（`CamelCase`）、小驼峰（`camelCase`）、下划线（`snake_case`）等。绝大多数时候，常量通常使用全大写字母和下划线分隔（例如 `MAX_VALUE`）。
- **注释**：在代码中添加适当的注释，解释代码的逻辑和意图。注释应该简洁明了，不要过于冗长。同时，注释应该与代码保持同步，避免出现过时的注释。避免使用 `#if 0` 和 `#endif` 来注释代码，这种方式风格很老，现在已经不推荐使用了。
- **空行**：适当使用空行来分隔代码块，以提高可读性。通常在逻辑相关的代码块之间、函数之间、类之间使用一到两个空行。
- **括号**：使用一致的括号风格，例如 **K&R 风格**（函数定义的左括号在同一行）或 **Allman 风格**（函数定义的左括号在新的一行）。
- **空格**：在运算符两边添加空格，例如 `a + b` 而不是 `a+b`。在逗号、分号等符号后面添加空格，例如 `a, b` 而不是 `a,b`。

- **行长度**：尽量保持每行代码的长度在 80-120 个字符之间，避免过长的行导致代码难以阅读。
- **文件长度**：尽量保持每个文件的长度在合理范围内（例如不超过 1500 行），避免过长的文件导致代码难以阅读和维护。

9.1.2 特定语言与社区风格指南

代码风格并非放之四海而皆准的真理。不同的编程语言、社区和公司都有自己独特的代码风格规范。无论是开源社区项目还是公司内部项目，当你参与时，都应该优先遵循其既有的代码风格。这有助于你更好地融入团队，并与他人高效协作。

以下是一些常见语言的知名代码风格指南，可供参考：

9.1.2.1 Python: PEP 8

Python 社区广泛遵循**PEP 8**（Python Enhancement Proposal 8）规范。它对命名约定、缩进、行长度、注释格式等都做了详细规定。PEP 8 推荐使用 4 个空格进行缩进，函数和变量名使用下划线命名法（snake_case）。

9.1.2.2 C: K&R, GNU, and Linux Kernel Style

作为一门历史悠久的系统编程语言，C 语言形成了多种常见的代码风格。

- **K&R 风格**：源自 C 语言的作者 Brian Kernighan 和 Dennis Ritchie 的著作《**The C Programming Language**》。它的特点是括号风格紧凑，左大括号跟在函数名或控制语句的同一行。这是许多后续风格的基础。
- **GNU 风格**：由**GNU 项目**推广。它对代码的布局和注释有非常详细的规定。其括号风格很独特，大括号需要单独成行并缩进。
- **Linux 内核风格**：由 Linus Torvalds 主导，用于**Linux 内核的开发**。它基于 K&R 风格，但有自己的独特规则，例如使用制表符（tab）进行缩进（且一个 tab 等于 8 个空格），并严格限制行长为 80 个字符。

9.1.2.3 C++: Google Style Guide & LLVM Style

C++ 社区存在多种代码风格。其中，**Google C++ Style Guide** 是一个非常著名且详尽的指南，被广泛应用于 Google 内部及其开源项目。它规定了包括文件命名、类设计、函数参数顺序在内的方方面面。另一个有影响力的风格是 **LLVM Coding Standards**，它在开源编译器社区中非常流行。

9.1.2.4 关注项目与社区习惯

即便是相对小众的语言，其社区也往往会形成自己的代码风格。例如，Vala 语言的社区就有一套流行的 **elementary 编码风格**，它在函数调用时推荐在函数名和括号间加空格（例如 `print ("Hello");`），这与许多其他语言的习惯可能不同。

当你加入一个新项目时，至关重要的第一步往往是花时间阅读并理解其代码风格指南。如果项目没有成文的规范，那么就通过阅读现有代码来学习和模仿其风格。**保持一致性是关键，不要将个人偏好随意带入项目中。**这不仅是对项目和其他贡献者的尊重，也能让你的代码更快地被他人接受。

9.1.2.5 自动化工具的重要性

我们自己写代码的时候虽然不能强制要求自己遵循某种风格，但是在团队协作中，保持一致的代码风格是非常重要的。我们可以使用一些工具来自动格式化代码，VS Code 的 C++ 插件就提供了代码格式化功能，可以通过快捷键（通常是 `Shift + Alt + F`）来自动格式化代码。至于 python，我们 `pip install black`，然后 `black .` 就可以自动格式化当前目录下的所有 python 文件了。

不要什么都手动缩进。人类不是打字机，机器比我们人打的整齐十倍甚至九倍。

除了这些以外，不同的岗位也有一些不同的码风需求，例如对于后端而言一个非常常见的差代码：

```
for(int i = 0; i<s.length(); i++){
    if(s[i] == 'a'){
        s[i] = 'b';
    }
}
```

以上代码的意图是将字符串中的所有字母 a 替换为 b，但是它的效率非常低下，因为每次替换都需要调用一遍 `s.length()`，而且每次替换都需要重新构建字符串，前端得等半天才能看到结果。而前端也有可能出现类似错误，前端的差代码可能是把 SQL 语句写进了 HTML 中，这直接导致了 SQL 注入漏洞，后端同学估计会直接气炸。

为了规避这些错误，同学们需要在实际项目中不断积累经验，才能写出更好的代码，我就不啰嗦了。

9.1.3 注释

很多人都不喜欢写注释，认为代码本身就是应该是自解释的，实则不然。当代码逻辑复杂或者涉及到一些特定的业务逻辑时，注释就显得尤为重要；要是码风再差一点，代码就更不能自解释了。

于是我们赌气一般地写了以下注释：

```
i += 1 # 增加 i 的值
```

对以上注释，我的看法是不如不写，因为只要是认识 `+=` 的人都知道这行代码的意思，这句话本质上是在重复代码本身，没有什么用处。

因此，注释的原则是：**注释应该解释代码的意图**，或者说，说清楚为什么这么写（在做什么）而不是“这是什么”。或者举个例子：

```
i += 1 # 跳过表头行，数据从下一行开始
```

这行注释就比上面的注释有用得多，后续在做代码评审的时候也很容易复现当时的思路。

当然，我们作为汉语使用者不喜欢注释非常正常，因为谁都不喜欢来回切换输入法，我也不喜欢大量的写注释。但是对于一些复杂的逻辑，虽然无法要求自己每一行都写注释，但是至少要在关键的地方写注释，例如某个非常复杂的算法，至少也要按照步骤写注释（这一部分是做什么，那一部分是做什么）。

部分公司做自建房的时候，会强制要求代码中注释达到一定的比例。这本意是好的，但是如果注释的质量不高，反而会导致代码难以阅读。部分人甚至导入数万字的网文来应付了事，这是非常不推荐的。

9.2 防御式编程

近年来防御式编程已经被解构成一个令人不忍直视的名词，例如向代码中添加大量的逆天处理（例如 `#define true false`）来防止自己被其他人取代或者被公司裁员。这完全是违背了“防御式编程”的初衷，这个名词来源于防御式驾驶，也就是你永远不知道其他司机会干出什么妨碍你的事来，所以要保持警惕。同理，在编程的时候也要保留着大量的警惕，防止别人或者自己在未来时犯错误；同时也在代码崩溃的时候把本不属于自己的错误优雅地甩锅给别人。

除了最常见的大量 `if-else` 以外，我们还可以使用一些其他的手段来更优雅地实现防御式编程，例如使用异常处理机制（`try-catch`）来捕获错误，或者使用断言（`assert`）来检查代码的前置条件和后置条件。

9.2.1 异常处理

程序员中经常流传着一首歌曲（尤其是 C# 程序员）：“死了都要 Try……”¹，说明了异常处理机制的重要性。异常处理机制可以帮助我们捕获和处理运行时错误，避免程序崩溃；换句话说，异常处理机制的思路是“晚崩溃，晚挨骂”。

一个经典的 C# 异常处理结构如下：

¹来自著名歌曲《死了都要爱》


```

try{
    // 可能有毛病的代码
    // 我们甚至可以人造异常
    // 例如throw new Exception("这是一个人造异常");
}
catch(Exception e){
    // 出了毛病就执行的代码，例如打印错误信息
}
finally{
    // 无论如何都会执行的代码
}

```

一般 `finally` 可以省略，在其他语言中语法也差不多。一个示例是：

```

try:
    user = User.get_by_id(user_id)
except UserNotFoundError:
    logger.error(f"用户{user_id}不存在")
    return {"error": "用户不存在"}

```

这种代码在查询的时候非常常见，能够有效地防止查到空对象并尽早暴露问题，同时还能防止程序崩溃，防止笨蛋甲方在酒吧点炒饭的时候不停地输入不存在的用户 ID 导致程序崩溃。

不要 *except Exception as e: pass*，除非你这个大笨蛋想被运维半夜叫醒。

9.2.2 断言

断言的意思是“我认为这个应该是对的”，如果不成立就抛出异常。断言通常用于检查代码的前置条件和后置条件，确保代码在运行时满足一定的条件。断言可以帮助我们在开发阶段发现问题，并且在生产环境中也可以用来捕获一些潜在的错误。断言的思路和异常处理的思路是相反的：早崩溃，早开心。

比方说以下代码：

```

def divide(a, b):
    return a / b

```

然后某在酒吧点炒饭的笨蛋甲方输入了 0 作为第二个参数，导致程序崩溃，然后甲方开骂，乙方只能默默挨骂。

这时候，我们可以使用断言来解决这个问题：

```

def divide(a, b):
    assert b != 0, "除数不能为零！"

```

```
return a / b
```

如果甲方输入了 0 作为第二个参数，程序就会抛出异常（一个“断言错误”），并且输出“除数不能为零！”的错误信息。这样就可以避免程序崩溃，并且可以更好地定位问题。（甲方估计也不会因为这个错误而开骂了）

这个代码如果用 `raise` 来写的话就会变成：

```
def divide(a, b):
    if b == 0:
        raise ValueError("除数不能为零！")
    return a / b
```

`raise` 是 Python 中手动抛出异常的语句，和 C# 的 `throw` 很像。上述代码和使用断言的区别有两点：一个是代码量变大了（多了一行，不够优雅），另一个是这个异常抛出的是“值错误”而不是“断言错误”。不过在实际操作中，这两个区别不大，且在查询等需要更多信息的场景中，使用 `raise` 会更好一些（生产环境中往往禁用断言，优雅不能当饭吃）。

在 C 系中，也有这样的断言，包括运行时断言和编译时断言。运行时断言指的是在运行时会检查某些条件是否成立，对编译进程没有影响；编译时断言则是在编译阶段就检查某些条件是否成立，如果不成立直接掐断编译。

下文是一个运行时断言：

```
#include <assert.h>
void divide(int a, int b) {
    assert(b != 0 && "除数不能为零！");
    printf("%d\n", a / b);
}
```

9.3 监控程序的运行情况

9.3.1 日志

日志能够帮助我们记录程序运行时的状态和错误信息，我们在 5.5.3 节中提到过的“打日志”调试方式就是一种使用日志的方式。

使用 `print` 语句是很初级的一种打日志手段，通常我们还会使用更高级的日志库，例如 Python 的 `logging` 模块或者 C++ 的 `spdlog` 库等。这些日志库可以提供更丰富的功能，例如日志级别、日志格式化、日志输出到文件等。使用这些日志库可以让我们的代码更加优雅，同时也能更好地管理日志信息。通常说来，日志有以下几个级别（严重性从低到高）：

- **DEBUG**：调试信息，通常用于开发阶段，记录一些调试信息。
- **INFO**：普通信息，记录一些程序运行的基本信息。
- **WARNING**：警告信息，记录一些可能导致问题的情况，但不影响程序的正常运行。

- **ERROR:** 错误信息，记录一些导致程序无法正常运行的错误。（这些错误往往不会导致程序崩溃）
- **CRITICAL:** 严重错误信息，记录一些导致程序崩溃的错误。

日志通常遵循一定的结构：时间-模块-级别-消息。例如：

```
2025-07-16 14:30:00 [user.py:45] ERROR 用户123登录失败：密码错误
```

日志的格式化可以使用一些工具来实现，例如 Python 的 `logging` 模块提供了丰富的格式化选项，可以自定义日志的输出格式。

9.3.2 其他监控手段

除了日志，我们还可以使用其他的监控手段来监控程序的运行情况，比方说打开任务管理器，查看 CPU、GPU、内存等资源的使用情况；或者使用一些性能分析工具，例如 Python 的 `cProfile` 模块、C++ 的 `gprof` 工具等，来分析程序的性能瓶颈；特定领域也有一些特定的性能分析工具，例如 `TensorBoard` 等。

一个例子：我们在机器学习相关的课程实验中经常会遇到训练过慢的问题。这个时候，不妨打开任务管理器，重点查看 GPU 和内存的使用情况。如果 GPU 使用率很低，要么是代码没有充分利用 GPU 的计算能力，应该增加并行能力（如提高批量大小）以加快代码运行速度；要么是代码没有在 GPU 上运行，这时候应该检查代码是否有从 CPU 到 GPU 的数据传输等。如果 GPU 使用率很高，说明代码可能存在性能瓶颈或者数据处理不当的问题，应该降低并行程度（但是一般这很难遇见，毕竟这种情况下可以堆卡）。如果内存使用率很高，说明代码可能存在内存泄漏或者数据处理不当的问题。通过这些信息，我们虽然很难直接定位代码的问题所在，但是也可以得到一些直接或者间接的线索，从而更好地优化代码。

对于一些使用 HTTP 服务的项目，我们还可以监控服务的请求和相应情况，其中最重要的数据应该是 QPS（每秒请求数）和响应时间。如果 QPS 很低或者降到 0，说明服务大概率出现了问题（另一种可能是真没有人使用这个服务）；如果响应时间很高，说明服务可能存在性能瓶颈或者数据处理不当的问题。我们可以使用一些工具来监控服务的请求和响应情况，例如 `Prometheus`、`Grafana` 等。前端的开发人员也可以使用浏览器自带的开发者工具来监控页面的加载时间、资源使用情况等。

9.4 常见的代码架构

在实际的开发中，为了便于组织代码，我们通常会遵循一定的代码架构，而不是把代码这碗面煮成一锅粥或者面疙瘩汤。在这里，我们向大家介绍几个常见的架构：

9.4.1 MVC 架构

这可以说是最简单的代码架构之一了。它由三个相对独立但是联系密切的部分组成：模型（Model）、视图（View）和控制器（Controller）。模型负责数据的存储和处理，视图负责展示用户界面，控制器负责处理用户输入、协调模型和视图之间的交互。

举个例子：假设我们在制作一个视觉小说游戏，那么我们的代码架构可以采取以上架构：

- 模型：负责存储游戏的状态、角色信息、剧情分支等数据。
- 视图：负责展示游戏的界面，包括角色立绘、背景、对话框等。
- 控制器：负责处理用户的输入，例如点击选项、输入文本等，并根据用户的选择更新模型和视图。

MVC 架构的优点是将代码分成了三个相对独立的部分，实现起来非常简单，尤其适用于中小型项目。缺点是三个部分之间虽然有着明确的职责划分且相对独立，但是耦合度仍然较高，如果需要修改某个部分的代码，经常会影响到其他部分的代码（你一改代码，别人都得跟着改），代码的可维护性比较低。

9.4.2 MVVM 架构

MVVM（Model-View-ViewModel）架构是一种常用于前端开发的架构，它将视图（View）和业务逻辑（ViewModel）分离开来，从而实现了更好的代码组织和可维护性。MVVM 架构通常用于前端框架，例如 Vue.js、Angular 等。笔者不是前端程序员，对 MVVM 了解甚少，于是不在这里误人子弟了。感兴趣的同学可以自行查阅相关资料。

9.4.3 洋葱架构（干净架构）

洋葱架构（也叫干净架构）是一种分层的架构，它的“层”之间有着严格的依赖关系。一般而言，洋葱架构分四层，外层依赖内层，但内层对外层一无所知，没有任何依赖。

一个典型的洋葱架构分四层：

- 实体层（Entity Layer）：最内层，包含业务逻辑和领域模型。它定义了系统的核心业务规则和数据结构。
- 用例层（Use Case Layer）：第二层，包含应用程序的用例和业务逻辑。它定义了系统的功能和行为，并调用实体层来实现业务逻辑。
- 接口层（Interface Layer）：第三层，包含与外部系统交互的接口和适配器。它定义了系统的输入和输出，并调用用例层来实现功能。
- 外部层（External Layer）：最外层，包含与外部系统交互的具体实现，例如数据库、Web 服务等。它依赖接口层来实现功能。

洋葱架构的优点是将代码分成了四个仅单侧依赖的部分，代码的可维护性和可扩展性都比较高（换 UI 不用动数据库格式；换数据库不用动业务逻辑）。缺点是实现起来比较复杂，因此比较适用于大型项目。如果同样拿刚刚的视觉小说游戏来举例，那么我们的代码架构可以

采取以上架构：

- 实体层：定义游戏的状态、角色信息、剧情分支等数据结构。
- 用例层：处理游戏的逻辑，例如实现判断玩家的选择、更新游戏状态、计算好感度等方法，但是不关心其他层怎么用这玩意。
- 接口层：与外部系统交互，例如接受点击的信息后，调用某个方法、返回某些数据，但不关心这数据具体是 Json 还是 SQL。
- 外部层：负责具体的实现。

这样，故事脚本永远在最甜的心里，就算明天把 Unity 改成 Godot、把这个立绘换成那个立绘，也只需要替换最外层，里面一点不用动。但是这样既不够直观，也不够简单，反而会让人觉得过于臃肿、没有必要。

但是如果我们做一个类似于微信的即时通讯软件，那么洋葱架构就非常适合了：

- 实体层：定义用户信息、聊天记录等数据结构，别的啥也不干。
- 用例层：基于这些数据结构，处理核心的业务逻辑，例如处理好友关系、群成员上限判断、雪花算法、端到端加密策略等，并不关心接口层用这些玩意干什么。
- 接口层：使用用例层给出的方法，处理用户输入和输出，例如收到“发送消息”事件 → 检查权限 → 端到端加密算法 → 发送到服务器 → 返回“发送成功”事件。它只关心如何使用用例层提供的功能，而不关心其他层具体怎么搞的。
- 外部层：负责具体的实现，包括并不限于在手机上、电脑上、车载系统上等不同平台的实现。

这样，把聊天核心逻辑（内两层）做成一个独立 SDK，外层壳子可以是微信本体、企业微信、微信 Mac 客户端，甚至车载微信，可移植性非常强。这样拆完，需求变更、团队并行、平台移植都变得像剥洋葱——泪流满面的是甲方，不是程序员。

9.4.4 微服务架构

与以上的架构不同，微服务架构并没有一个非常统一的部分或者层次划分；它的宗旨是将一个大型项目拆成许多小的、独立度极高的服务，使得每个服务都可以独立部署、独立扩展、独立维护。每个服务都可以使用不同的技术栈和编程语言来实现，从而实现了更好的灵活性和可扩展性，适合各类大中小型项目。

微服务架构的核心是 API（应用程序编程接口），每个服务都提供一组 API 供其他服务调用。服务之间通过 API 进行通信，通常使用 HTTP 或消息队列等方式。打个比方：一个校园，我们把它拆成了许多服务，例如教务、食堂等；我们学生（也是一个服务）可以调用各种 API（例如食堂提供的“吃饭”API、教务提供的“上课”API 等）来达到自己的目的。

微服务架构的缺点也很明显：服务之间的通信和协调比较复杂，需要使用一些工具来管理服务的依赖关系和通信（例如通信过多的时候就“暂时不能给你明确的答复”）；另一个问题是服务之间通信的延迟、网络问题会显著降低该架构的性能和可靠性；除此之外，它还有部署复杂、测试困难等问题。

第十章 调试、测试和部署

我们在前面的章节中已经知道，代码中出现错误是难免的事情。无论是语法错误、语义错误，还是不能称得上错误但是不符合预期的行为，我们都需要进行调试和测试来找出问题所在。

一些大公司专门有“测试工程师”这类岗位来负责代码正式上线之前的测试工作，以检查代码的正确性和可靠性；但是对于大多数小型项目而言，这些工作往往是由开发人员自己完成的。在本章节中，我们会介绍一些常用的调试和测试方法，帮助同学们更好地理解 and 解决代码中的问题。

总得说来，调试和测试就像诊治一个危重病人（值得庆幸的是，这个病人能够复活）一样。此时，我们需要四步走：先救命、再治病、再调养，最后购买医疗保险，然后让他去工作。顺序不要乱，一步都不能少。

10.1 先救命

最常见的情况是：代码崩溃了，程序完全无法执行。对于 Python 而言，它的报错信息非常详细，通常可以直接定位到出错的行数和错误类型，因此往往只需要根据报错信息进行修复即可。

而对于 C/C++ 这类语言而言，它们是静态的，运行时错误能过编译，而且它们的报错信息通常非常简洁，仅凭报错信息很难定位到具体的错误地点。因此，我们需要使用一些调试工具来帮助我们定位错误。

什么，你问我怎么确定编译错误？我认为编译错误应该由编译器来判断，而不是我们来判断；编译过一次以后，你的代码编辑器应该也能够显示哪里有编译错误！

10.1.1 使用调试器

我们在开发的过程中可以使用 VS Code 调试器，但是有时候我们无法获取程序源码。在这种情况下，我建议同学们使用那个最经典的调试工具——gdb。它是 GNU 项目的一部分，支持多种编程语言，包括 C、C++、Fortran 等。当然对于一些会读汇编的同学们而言，我觉得可以用 objdump 这类反汇编器来反汇编程序，并对这些汇编代码进行阅读；常见的反汇编器还有著名的 IDA Pro，它是一个商业软件，价格很贵，但是功能绝对对得起它的价格：它甚至能够对反汇编出来的东西进行自动分析！

```
objdump -d ./your_program > asm.s
```

当然大多数人是没这个能力也没这个毅力去读汇编的，因此 GDB 最终还是我们最常用的动态调试工具。例如，以下命令可以启动 GDB 并加载程序：


```
gdb ./your_program
```

在 GDB 中，我们可以使用以下命令来设置断点、运行程序、查看变量等：

- **break**: 设置断点，可以简写为 **b**，例如 `break main` 在 `main` 函数处设置断点。也可以在特定的某一行设置断点，例如 `break 42` 在第 42 行设置断点；也可以利用偏移量来设置断点，例如 `break +10` 在当前行的往前数 10 行设置断点。
- **run**: 运行程序。可以简写为 **r**。
- **next**: 执行下一行代码，可以简写为 **n**。如果当前行是函数调用，则不会进入函数内部，而是把这个函数视为一个整体往下执行一行。该命令有一个变体 **ni**，它是汇编级别的断点定位，也就是执行下一条汇编指令。
- **step**: 也是执行下一行代码，可以简写为 **s**。如果当前行是函数调用，则会进入函数内部，逐行执行函数内部的代码。**si** 是它的变体，表示汇编级别的单步执行。
- **continue**: 继续运行直到下一个断点，可以简写为 **c**。
- **print**: 打印变量的值，例如 `print variable`，可以简写为 `p variable`。如果变量是一个结构体或类的实例，可以使用 `print variable.field` 来打印某个字段的值。
- **backtrace**: 查看函数调用栈，可以简写为 **bt**。这对于定位程序崩溃时的调用路径非常有用。
- **layout**: 切换到图形界面模式，可以使用 `layout src` 来显示源代码，使用 `layout asm` 来显示汇编代码。
- **info**: 查看程序的状态，例如 `info breakpoints` 查看断点信息，`info registers` 查看寄存器状态，`info locals` 查看局部变量等。
- **watch**: 设置观察点，当某个变量的值发生变化时暂停执行，例如 `watch variable`。这对于调试复杂的逻辑错误非常有用。
- **set**: 强制设置变量的值，例如 `set variable = value`。这对于调试时修改变量的值非常有用。
- **list**: 查看源代码（带行号），可以简写为 **l**。
- **quit**: 退出 GDB。

当然，如果想要显示行号的话，我们编译代码的时候需要加上 `-g` 选项，例如：

```
gcc -g -o your_program your_program.c
```

也只有加上了 `-g` 选项，GDB 才能够显示源代码和行号，**s** 和 **n** 命令才能够逐行执行源代码，但是 **si** 和 **ni** 命令仍然能够正常执行。

以上命令其实很复杂，需要同学们多加练习才能熟练掌握。这里我推荐一个 GDB 的小练习：**CMU ICS Lab2: BombLab**。这个练习的目的是让同学们通过 GDB 来调试一个被加密的程序，找到正确的输入来“拆弹”。这个练习非常有趣，而且可以帮助同学们熟悉 GDB 的使用。

当然，GDB 的 TUI 界面还是太老了，我推荐装个插件 `pwndbg`，它可以让 GDB 的 TUI 现代化得多。

10.1.2 尸检

有时候程序确实崩了，救不活了，这时候我们需要“死后验尸”，确定程序真正的“死因”。当然，我们在解剖尸体之前，至少得对死因了解个大概，例如是段错误、内存泄漏、未定义行为，还是什么奇奇怪怪的东西。

10.1.2.1 段错误

段错误可以使用 `core dump` 来分析。`core dump` 是程序崩溃时操作系统生成的一个文件，包含了程序的内存状态和寄存器状态等信息。我们可以使用 GDB 来分析 `core dump` 文件。

`ulimit -c unlimited` 命令可以设置 `core dump` 文件的大小限制为无限制。然后运行崩溃的程序，等着程序再崩溃一次。然后运行：`gdb ./your_program core`。这会加载 `core dump` 文件，并且可以使用 `bt` 命令查看函数调用栈，使用 `info locals` 查看局部变量等。

10.1.2.2 内存泄漏

内存泄漏可以使用 Valgrind 来分析。Valgrind 是一个开源的内存调试工具，可以检测内存泄漏、未初始化内存读取等问题。使用 Valgrind 非常简单，只需要在运行程序时加上 `valgrind` 命令即可，例如：

```
valgrind --leak-check=full ./your_program
```

Valgrind 会输出内存泄漏的详细信息，包括泄漏的内存地址、大小、调用栈等信息。我们可以根据这些信息来定位内存泄漏的代码。

Valgrind 跑完以后别急着关，如果是开源项目或者协作项目，把 `definitely lost` 那行抄下来贴到 issue 上面，省得以后再犯同样的错误。

10.1.2.3 未定义行为

越界和未定义行为可以使用 ASan 和 UBSan 来分析。ASan (AddressSanitizer) 是一个内存错误检测工具，可以检测越界访问、使用后释放等问题。UBSan (UndefinedBehaviorSanitizer) 是一个未定义行为检测工具，可以检测整数溢出、空指针解引用等问题。

我们在编译文件的时候，可以加上 `-fsanitize=address` 和 `-fsanitize=undefined` 选项来启用 ASan 和 UBSan。

10.2 再治病

有些时候，程序没有崩溃的风险，但它的执行不符合预期且占用了过多的资源。这时候，我们需要进行性能调优和资源使用分析。以下是一些常见的性能问题和资源使用问题，以及相应的分析工具。

10.2.1 CPU 吃满但是程序显然不应该吃满 CPU

这种情况可以使用 `perf` 来分析问题，由 Linux 内核提供。只需要在运行程序时加上 `perf` 命令即可，例如：

```
perf record -g ./your_program && perf report
```

10.2.2 内存吃满但是程序显然不应该吃满内存

这种情况可以使用 `massif` 来分析问题，该工具由 Valgrind 提供。例如：

```
valgrind --tool=massif ./your_program
```

运行完毕后，可以使用 `ms_print` 命令来查看内存使用情况，例如：

```
ms_print massif.out.<pid>
```

10.2.3 IO 卡死

IO 卡死通常是因为程序在等待某个 IO 操作完成，例如网络请求、文件读写等。我们可以使用 `iostat` 来分析 IO 卡死问题，只需要：

```
sudo iotop -o
```

然后盯着它看，找谁疯狂 IO 就可以了。

当然，对于 Node.js 和 Python 服务，可以使用 `--inspect` 模式；然后打开 Chrome 浏览器，输入 `chrome://inspect`，火焰图和前端一样香。

10.3 再调养

经过以上的两步骤，我们总算是把程序的主要问题解决了个差不多。但是，为了保证程序的稳定性和可靠性，我们还需要进行一些额外的调养工作——测试。

10.3.1 测试第一步：隔离

我们最好是找个地方把需要测试的代码隔离开来，放到一个单独的地方，防止其他代码或者文件影响测试结果；对于服务类型的项目，则更是如此。我们可以使用 `tmux`、`screen` 或者 `Docker` 来隔离测试环境。

10.3.1.1 使用 tmux 和 screen

tmux 和 screen 是两个常用的终端复用工具，可以在一个终端中创建多个会话，方便我们进行隔离测试。使用方法非常简单，只需要在终端中输入 tmux 或 screen 即可进入一个新的会话。

在 tmux 中，我们可以使用以下命令来创建新的窗口、分割窗口等：

```
tmux new session -s session_name # 创建一个新的会话
tmux ls # 列出所有会话
tmux attach -t session_name # 连接到指定的会话
```

会这三个就行。在 screen 中的相同功能命令是：

```
screen -S session_name # 创建一个新的会话
screen -ls # 列出所有会话
screen -r session_name # 连接到指定的会话
```

10.3.1.2 使用 Docker

docker 和上述两个东西不太一样。上述两个东西只能做到“守护终端”，但是对于环境的变化无能为力。而使用 docker 可以做到隔离环境，甚至可以做到“守护进程”。我们可以使用 Docker 来创建一个隔离的测试环境，例如：

```
docker run -it --rm -v $(pwd):/app -w /app python:3.9 bash
```

以上代码的含义是创建一个 Python 3.9 的 Docker 容器，并将当前目录挂载到容器的/app 目录下，然后进入容器的 bash 终端。这样，我们就可以在隔离的环境中进行测试了。

我们也可以让测试代码在 docker 容器里面跑起来以后，再退出来，使用其他代码（例如测试代码）来访问这个容器。此时我们需要再 docker 容器中预留端口，也就是：

```
docker run -it --rm -p 8000:8000 -v $(pwd):/app -w /app python:3.9 bash
```

上述命令是做了一个端口映射，我们可以在本地的 8000 端口访问容器中的 8000 端口。

默认情况下，离开容器不会关闭容器，除非服务挂了。这时候我们可以使用 docker exec 命令来进入容器，例如：

```
docker exec -it container_name bash
```

如果我们不想要这个容器了，可以使用 docker rm 命令来删除它，例如：

```
docker rm container_name
```

10.3.2 单元测试

单元测试是对代码的最小可测试单元进行验证的过程。它通常是自动化的，可以帮助我们快速发现代码中的问题。Python 和 C/C++ 都有很好的单元测试框架。

对于 Python，我们可以使用 unittest 框架来编写单元测试。以下是一个简单的示例：

```
import unittest
class TestMyFunction(unittest.TestCase):
    def test_case_1(self):
        self.assertEqual(my_function(1, 2), 3)

    def test_case_2(self):
        self.assertRaises(ValueError, my_function, -1, 2)
if __name__ == '__main__':
    unittest.main()
```

在这个示例中，我们定义了一个测试类 `TestMyFunction`，它继承自 `unittest.TestCase`。我们在这个类中定义了两个测试用例，分别测试了 `my_function` 函数的正确性和异常处理。最后，我们使用 `unittest.main()` 来运行所有的测试用例。这是一个非常简单的单元测试示例，实际的单元测试可能会更加复杂，要涉及几乎所有的代码逻辑。因此测试工程师这个职业也不是什么轻松的工作。

在编写测试用例的时候，除了涉及到大多数的常规数据以外，也要尽可能地考虑一些特殊值（例如边界情况等），也需要故意混入一些显然错误的数据来测试程序的健壮性和异常处理能力。

10.3.3 集成测试

集成测试则指的是对整个系统进行测试，验证各个模块之间的交互是否正常。集成测试通常是手动进行的，一般是编写一些集成测试代码，然后用这个代码来测试整个系统的功能是否正常。

对于 Python，我们可以使用 pytest 框架来编写集成测试。以下是一个简单的示例：

```
import pytest
def test_my_function():
    assert my_function(1, 2) == 3
    assert my_function(-1, 2) == 1
    assert my_function(0, 0) == 0
    with pytest.raises(ValueError):
        my_function(-1, -2)
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
```

```
(-1, 2, 1),
(0, 0, 0),
])
def test_my_function_parametrized(a, b, expected):
    assert my_function(a, b) == expected
```

对于服务类项目，我们的测试代码可以是自动调用这个服务来做一些事情，看看能不能产生符合预期的结果。比如说，我们可以使用 `requests` 库来发送 HTTP 请求，验证服务的响应是否正确。

10.4 买保险

大多数时候，我们的程序经过一大堆调试和测试后，已经可以正常运行，并得到了充分的优化。但是为了保证程序确实得到了优化，我们还需要进行一些额外的测试工作。比方说，我们使用 Python 的 `pytest-benchmark` 装饰器来进行性能测试：

```
@pytest.mark.benchmark
def test_my_function(benchmark):
    result = benchmark(my_function, *args, **kwargs)
    assert result == expected_result
    return result
```

跑就完了。

对于所有程序，我们都可以使用 `hyperfine` 来进行性能测试：

```
hyperfine 'python_old.py' 'python_new.py' --warmup 5 --runs 50
```

加上这个选项把东西抄下来：

```
--export-markdown results.md
```

然后 PR 里面贴出“优化完成”和这张表，老板登时点赞。

10.5 去工作

代码调试完成了，现在该把代码部署到生产环境了。部署代码的方式有很多种，具体取决于项目的类型和规模。对于小型项目，我们可以直接将代码上传到服务器上运行；对于大型项目，我们可以使用 CI/CD 工具来自动化部署流程。

GitHub Actions 是一个常用的 CI/CD 工具，可以帮助我们自动化部署流程。我们可以编写一个 GitHub Actions 工作流文件，定义部署的步骤和条件。例如：

```
name: Deploy
```



```
on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.9'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run tests
        run: pytest
      - name: Deploy to server
        run: |
          scp -r . user@server:/path/to/deploy
          ssh user@server 'cd /path/to/deploy && ./deploy.sh'
```

在这个示例中，我们定义了一个名为 `Deploy` 的工作流，当代码推送到 `main` 分支时触发。工作流包含了几步：检出代码、设置 `Python` 环境、安装依赖、运行测试和部署到服务器。当然，具体的部署步骤需要根据项目的实际情况进行调整。

除了自动部署以外，`CI/CD` 工具还可以帮助我们进行代码质量检查、性能测试等工作。我们可以在工作流中添加相应的步骤来实现这些功能。

第十一章 常用构建工具链概览

讲完了开发的各个方面，现在该讲讲怎么把你的代码构建成可执行的文件（或包），以便于之后的发布和使用。

一个比较古老的构建方法是使用 `makefile` 构建，这是非常经典的构建方式，几乎所有的编程语言都可以使用。但是因为风格过老，现在已经不大流行。目前较为常见的构建方式是 `CMake` 和 `XMake`，它们风格现代，功能强大。

11.1 CMake

C 系语言功能非常强大是公认的不争事实。但是它的构建一直是一个巨大的痛点：使用 `include` 来链接库是非常麻烦的事情，文件一多，`gcc` 直接变成一种折磨（`gcc a.c b.c ... -o project`，还得管依赖关系，还得管编译顺序），困难得让人完全不想用。

在 2000 年左右，`CMake` 作为一个跨平台的构建工具被提出。它的主要目标是提供一个简单的方式来管理大型项目的构建过程。在当时，`autotools` 难写，`Makefile` 难以跨平台，而 `CMake` “描述意图”代替“描述过程”，使得构建过程变得更简单。

好风凭借力。随着 `CMake` 渐渐进入人们的视野，`KDE`、`LLVM`、`OpenCV`、`Qt6`、`ROS2` 等项目全都开始使用 `CMake` 作为构建工具，社区生态空前繁荣，直接将 `CMake` 推上了构建工具的顶峰。直到现在，`CMake` 依然是面向生产环境的“事实标准”，`VS`、`CLion`、`QtCreator`、`VS Code` 全部将 `CMake` 作为一级公民，`GH Actions`、`Docker`、`Conan`、`vcpkg` 无需装包，默认识别 `CMake` 项目。

因此，不学会 `CMake`，你在今日的开发世界中寸步难行：除非你愿意用 `Makefile` 这种玩意为每个 IDE 和平台写一大堆构建脚本！

11.1.1 最小运行实例

`CMake` 的文件名称默认是 `CMakeLists.txt`。除非特殊情况，以后不再提及文件名称。

11.1.1.1 单一文件

假设现在存在一个 `main.cpp` 文件（不管内容是什么了）。在同一目录下新建一个 `CMake` 文件，内容如下：

```
cmake_minimum_required(VERSION 3.10) # 声明最低CMake版本
project(hello LANGUAGES CXX) # 声明项目名称和使用的语言
add_executable(hello main.cpp) # 声明可执行文件hello，源文件为main.cpp
```

然后构建：

```
cmake -B build # 生成构建系统
cmake --build build # 构建
./build/hello # 运行可执行文件 (Linux和mac)
./build/Debug/hello.exe # 运行可执行文件 (Windows)
```

爽。

11.1.1.2 多个目录

假设现在有一个目录结构：

```
src/
- main.cpp
- math/
  - add.cpp
  - add.hpp
- io/
  - print.cpp
  - print.hpp
```

根目录 CMake 文件如下：

```
cmake_minimum_required(VERSION 3.10)
project(multidir)

add_subdirectory(src) # 告诉CMake: src里面还有东西
```

src/CMake 文件如下：

```
add_library(math STATIC math/add.cpp) # 声明静态库math, 源文件为add.cpp
add_library(io STATIC io/print.cpp) # 声明静态库io, 源文件为print.cpp
add_executable(app main.cpp) # 声明可执行文件app, 源文件为main.cpp
target_link_libraries(app PRIVATE math io) # 将math和io库链接到app
```

构建同单文件一样，不打第二遍。

我们看到一个问题：CMake 需要每个目录一个，职责清晰；同时，库和可执行文件都是“目标”。目标这个词在 CMake 中非常重要，几乎所有的操作都是针对目标进行的，后文会反复出现。

11.1.2 语法

11.1.2.1 目标和作用域

现代的 CMake 的语法基于目标，把属性贴在目标上，谁链接谁可见。比如说：

```
add_library(foo STATIC foo.cpp) # 告诉CMake，这有个库foo，是静态的
target_include_directories(foo PUBLIC include) # foo库的头文件在include目录下
target_compile_features(foo PUBLIC cxx_std_20) # foo库需要C++20特性
target_link_libraries(foo PUBLIC bar) # foo库需要链接bar库
```

这里的 PUBLIC 是一个作用域，表示这个属性对链接到 foo 的目标可见。作用域有三个：

- PRIVATE: 给自己用
- INTERFACE: 给下游用，自己不用
- PUBLIC: 自己和下游都用

一般情况下，除非显式传递，否则子目录自动继承父目录作用域；而对于一个特定的目标，属性自动跟随，跨目录也能传递。类似 link_directories 这样的命令是全局的命令，在现代 CMake 中非常不推荐使用。

11.1.2.2 变量、缓存、生成器

对于 CMake 而言，我们可以声明一些变量，使得 CMake 源码更简洁。例如：

```
set(SOURCE_FILES main.cpp math/add.cpp io/print.cpp) # 声明变量SOURCE_FILES
add_executable(app ${SOURCE_FILES}) # 使用变量
```

这样可以省去很多重复的代码。

CMake 的变量分为两种：缓存变量和普通变量。普通变量用户是修改不了的，而缓存变量可以通过命令行或 CMake GUI 修改。缓存变量通常用于配置选项，比如：

```
option(BUILD_TESTS "Build tests" ON) # 声明一个缓存变量BUILD_TESTS，默认值为ON
set(CMAKE_BUILD_TYPE "Release" CACHE STRING "Build type") # 声明一个缓存变量CMAKE_BUILD_TYPE，默认值为Release
```

用户可以修改这些缓存变量的值，以定制构建过程。

```
cmake -B build -DBUILD_TESTS=OFF -DCMAKE_BUILD_TYPE=Debug # 修改缓存变量，将
BUILD_TESTS设置为OFF，将CMAKE_BUILD_TYPE设置为Debug
```

生成器决定了 CMake 生成的构建系统类型。CMake 支持多种生成器，比如 Makefile、Ninja、Visual Studio 等。可以通过 -G 选项指定生成器。目前较为常见的生成器有：

- Ninja: 一个快速的构建系统，CMake 默认生成的构建系统。跨平台最快。
- Unix Makefiles: 传统的 Makefile 生成器，Linux 服务器默认。
- Visual Studio: Windows 下的 Visual Studio 生成器。

- Xcode: macOS 下的 Xcode 生成器。

使用这些生成器非常简单:

```
cmake -B build -G Ninja # 使用Ninja生成器
```

11.1.2.3 条件判断、执行命令、输出信息

在 CMake 中, 我们可以使用条件判断来控制构建过程。这在使用了缓存变量的时候非常有用:

```
if(BUILD_TESTS) # 如果BUILD_TESTS为真
    enable_testing() # 启用测试
    add_subdirectory(tests) # 添加tests目录
endif() # 结束条件判断
```

这个 if 的条件写法和 C++ 的 if 类似, 也可以写 `elseif()` (没有空格) 和 `else()`, 含义也相同; 但是不支持运算符。所有的运算符都应该用字符串表示:

- AND、OR、NOT: 御三家, 不用解释都知道这是什么
- EQUAL、LESS、GREATER: 比较运算符, 分别表示等于、小于、大于
- EXIST: 后面的东西 (绝对路径) 若存在, 则为真; 否则为假
- DEFINED: 后面的变量若已定义, 则为真; 否则为假
- COMMAND: 后面的命令 (宏、函数) 若存在并能执行, 则为真; 否则为假
- STREQUAL、STRLESS、STRGREATER: 字符串比较运算符, 分别表示等于、小于、大于。使用前提是字符串必须是有效的数字。

我们可以使用 `execute_process` 命令来执行外部命令:

```
execute_process(COMMAND echo "Hello, World!" OUTPUT_VARIABLE output RESULT_VARIABLE
    result) # 执行命令, 输出到output变量, 结果状态码到result变量
```

一次可以同时执行许多命令, 这些命令是并行的; 每一个子进程的标准输出都会映射到下一个进程的标准输入; 所有的子进程公用一个标准错误输出管道。除了 `COMMAND` 外, 其余关键字均可以省略。

有时候, 我们需要输出一些信息到控制台上 (例如错误信息等)。CMake 提供了 `message` 命令来输出信息:

```
message(STATUS "This is a status message") # 输出状态信息
message(WARNING "This is a warning message") # 输出警告信息
message(ERROR "This is an error message") # 输出错误信息
message(FATAL_ERROR "This is a fatal error message") # 输出致命错误信息并终止构建
```

11.1.2.4 列表

CMake 支持列表 (list)。列表一般需要使用 set 来定义。以下是一些常用的列表操作命令：

```
set(SOURCES main.cpp math/add.cpp io/print.cpp) # 定义一个列表SOURCES
list(APPEND SOURCES io/scan.cpp) # 向列表SOURCES添加一个元素
list(REMOVE_ITEM SOURCES io/scan.cpp) # 从列表SOURCES中移除一个元素
list(LENGTH SOURCES length) # 获取列表SOURCES的长度，存储到length
list(GET SOURCES 0 first) # 获取列表SOURCES的第一个元素，存储到first
list(SORT SOURCES) # 对列表SOURCES进行排序
list(FIND SOURCES main.cpp index) # 查找列表SOURCES中元素main.cpp的位置，存储到index
list(INSERT SOURCES 0 "io/scan.cpp") # 在列表SOURCES的开头插入元素io/scan.cpp
list(JOIN SOURCES " " joined) # 将列表SOURCES用逗号和空格连接成一个字符串，存储到
joined
```

基本上是所见即所得。可以看到，列表操作和 C++ STL 的 vector 类似。

11.1.2.5 调库

有时候我们需要调外部库，例如 OpenCV 等。我们把调库分为两种：系统库和第三方库。如果系统已经安装，使用 find_package 命令即可：

```
find_package(fmt REQUIRED) # 查找fmt库，REQUIRED表示必须找到
target_link_libraries(app PRIVATE fmt::fmt) # 将fmt库链接到app
```

上述提到的 fmt 库是一个常用的 C++ 格式化库，Linux 发行版可以通过包管理器安装（例如 Ubuntu 的 apt install libfmt-dev）。

如果系统没安装，则需要使用 FetchContent 来当场下载。FetchContent 是 CMake 3.11 引入的模块。示例代码如下：

```
include(FetchContent) # 引入FetchContent模块
FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG ? # 指定版本，这里没有指定，实际应当指定一个
)
FetchContent_MakeAvailable(googletest) # 下载并编译googletest库
```

FetchContent 会自动下载并编译 googletest 库，并将其链接到当前项目。这样构建的好处是不需要预装库，直接构建时下载并编译，适合 CI/CD 等场景。缺点：首次编译时间长。

除此之外，CMake 还支持使用 Conan、vcpkg 等包管理器来管理第三方库。Conan 和 vcpkg 都是非常流行的 C++ 包管理器，可以自动处理依赖关系和版本问题。


```

vcpkg install fmt # 使用vcpkg安装fmt库
cmake -B build -DCMAKE_TOOLCHAIN_FILE=$VCPKG_ROOT/scripts/buildsystems/vcpkg.
cmake # 使用vcpkg的CMake工具链文件

```

vcpkg 会将库链接到当前项目，例如上文会将 `fmt::fmt` 暴露给 `find_package`。

11.1.3 工具链

工具链文件（ToolChain 文件）是一段纯粹的 CMake 脚本，在 `project` 命令之前被 CMake 加载，用于通知当前线索的重要信息：目标平台是什么？使用什么交叉编译器？头文件什么的在哪？默认编译、链接 flags 是什么？

闲的没事的人可以使用 `cmake -D` 一个个传。这么做最大的问题是：太多了，先不说传错、传漏的可能性，光说脚本的可读性就够我们喝一壶了。

假设我们现在希望在 Linux 机器上编译一个 Win32 程序，那么工具链文件可以是这样的：

```

# 文件名: cross-mingw.cmake
# 1. 目标系统
set(CMAKE_SYSTEM_NAME Windows) # 告诉 CMake “我要生成 Windows PE”
set(CMAKE_SYSTEM_PROCESSOR i686) # 目标 CPU

# 2. 交叉编译器
set(CMAKE_C_COMPILER i686-w64-mingw32-gcc)
set(CMAKE_CXX_COMPILER i686-w64-mingw32-g++)

# 3. sysroot / 搜索根目录
set(CMAKE_FIND_ROOT_PATH /usr/i686-w64-mingw32)

# 4. 查找策略：头文件/库只在目标环境里找，可执行程序在宿主环境里找
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

```

之后编译：

```

cmake -B build-win -DCMAKE_TOOLCHAIN_FILE=cross-mingw.cmake
cmake --build build-win
./build-win/hello.exe # 运行可执行文件

```

上述代码会在 Linux 上编译一个 Win32 程序，但是可以直接拷贝到 Windows 上运行。

当然，这只是一个最基本的示例，实际上工具链的使用相当复杂，甚至超过了 CMake 本身的复杂度。但是，交叉编译的脏活累活全是它做的，正是工具链的使得我们无需使用一大

堆-D 参数来传递信息。因此将来如果有的同学有志于从事嵌入式开发等工作，建议还是去官方网站上学习一下工具链的使用。

11.1.4 安装、导出、打包

CMake 支持安装、导出和打包功能。安装功能可以将构建好的目标安装到指定目录，导出功能可以将目标导出为一个 CMake 包，打包功能可以将目标打包为一个可分发的文件。

安装功能使用 `install` 命令：

```
install(TARGETS foo EXPORT fooTargets # 安装foo目标，并导出为fooTargets
  RUNTIME DESTINATION bin # 可执行文件安装到bin目录
  LIBRARY DESTINATION lib # 动态库安装到lib目录
  ARCHIVE DESTINATION lib # 静态库安装到lib目录
)
install(DIRECTORY include/ DESTINATION include) # 头文件安装到include目录
```

导出功能使用 `export` 命令：

```
install(EXPORT fooTargets
  FILE foo-config.cmake
  NAMESPACE foo::
  DESTINATION lib/cmake/foo) # 导出fooTargets为foo-config.cmake文件，命名空间为foo
::
```

导出后下游项目可以直接 `find_package(foo REQUIRED)` 来使用 `foo` 库。

打包功能使用 `cpack` 命令：

```
set(CPACK_GENERATOR "DEB")
set(CPACK_PACKAGE_NAME "mylib")
include(CPack)
```

上述代码会生成一个 DEB 包，包名为 `mylib`。构建的时候，使用以下命令可以得到打包文件：

```
cmake --build build --target package # 构建并打包
```

11.1.5 常见坑

CMake 虽然强大，但也有一些常见的坑需要注意：

- 缓存残留：目录改了名，CMake 仍然记得旧值，这时候把整个 `build` 目录全删了就可以了。
- 找不到头文件：看看 `target_include_directories` 是否正确设置了作用域，八成是错写成了 `PRIVATE`。

- 找不到 dll (Windows 特供): 把 bin 目录加入 PATH, 或者运行以下命令后运行 `install/bin` 下的可执行文件:

```
cmake --install build --prefix install
```

- 生成器表达式看不懂: 在 `CMakeLists` 里 `message(STATUS "$<TARGET_FILE:foo>")` 打印调试。

以上内容仅仅是给 CMake 的一个概览。虽然可以让同学们理解 CMake 的基本用法, 但要深入使用 CMake, 还是需要阅读官方文档和相关教程。毕竟 CMake 的命令非常多 (就一个 `execute_process` 就有一大堆选项, 合起来能占满半个屏幕), 而且每个命令的用法也非常灵活。

这里帮各位把 **CMake 官方说明文档** 贴出来了, 感兴趣的同学可以去看看; 也可以看看这本电子书: **Professional CMake: A Practical Guide** (作者: Craig Scott), 这本书是 CMake 的权威指南, 内容非常全面, 适合深入学习 CMake。

11.2 XMake

不少同学可能会疑惑: 怎么又来一个 XMake?

这是因为, CMake 虽然已经很好了, 但是语法还是有些复杂, 有点像“外星语”。而 XMake 比 CMake 现代的多, 语法也更简单, 使用 lua 脚本而不是 CMake 特有的脚本, 更容易上手。同时, XMake 把 CMake 的 `configure`、`generate`、`build` 等步骤压成一步, 且自带包管理, 非常舒适。当然, 也因为太新了, 所以大多数情况下使用 CMake 已经可以满足需求了; 不过在这里我还是简单讲一下吧, 毕竟 PKU 的部分课程提供的引擎需要我们手写 XMake 脚本。

安装非常简单, 包管理器直接解决, 不管用的 `winget` 还是 `apt`, 都可以直接安装 XMake:

```
winget install xmake # Windows
apt install xmake # Ubuntu
brew install xmake # macOS
```

11.2.1 最小运行实例

XMake 的构建文件名称默认是 `xmake.lua`, 本章不再赘述。

运行以下命令看看模板:

```
xmake create -l c++ hello # 创建一个C++项目hello
xmake create -l c++ -t static hello_lib # 创建一个C++静态库项目hello_lib
```

我们发现当前目录多了个 `hello` 文件夹, 结构是这样的:

```
hello/
- xmake.lua # 构建文件
```

```
- src/
  - main.cpp # 源文件
```

如果希望构建并运行这玩意，直接运行以下命令：

```
cd hello # 进入hello目录
xmake # 构建
xmake run # 运行
xmake run -d # 调试运行
```

11.2.2 语法速通

11.2.2.1 最小示例

```
-- xmake.lua
set_project("hello")
set_version("1.0.0")
add_rules("mode.debug", "mode.release") -- 自动生成 debug/release 配置

target("app")
    set_kind("binary")
    add_files("src/*.cpp")
    set_languages("c++20")
    add_packages("fmt")
```

-- 一个目标
-- 可执行文件
-- 通配符
-- 标准
-- 依赖 fmt 头文件+库

我们发现，对比 CMake，XMake 的语法更加简洁，没有复杂的概念；无需手写类似于 `if (WIN32)` 这样的系统条件判断，XMake 会自动根据宿主处理；虽然也有目标的概念（`target` 块），但是这一个块就相当于 CMake 的 `add_executable`（或者 `add_library`）和 `target_*` 的组合。

在 XMake 中，目标也有作用域的概念。不过写起来非常简单，例如 `add_includedirs("include", public = true)` 表示这个头文件目录是公共的，链接到这个目标的其他目标也可以使用这个头文件目录。

11.2.2.2 语法速查（和 CMake 对比）

11.2.2.3 目标、规则

在 XMake 中，目标可以是以下种类当中的任意一种，我们可以使用 `set_kind()` 来设置目标类型。关于目标和作用域的讨论，XMake 和 CMake 如出一辙。

- **binary**: 可执行文件

| CMake | XMake | 说明 |
|---|--|----------------|
| <code>add_executable</code> | <code>target("app"), set_kind("binary")</code> | 声明一个可执行文件或库 |
| <code>add_library</code> | <code>target("lib"), set_kind("static")</code> | 声明一个静态库或动态库（sh |
| <code>target_sources</code> | <code>add_files("src/.cpp")</code> | 添加源文件 |
| <code>target_include_directories</code> | <code>add_includedirs("include")</code> | 添加头文件目录 |
| <code>target_compile_definitions</code> | <code>add_defines("F00")</code> | 添加编译选项 |
| <code>target_compile_options</code> | <code>set_cxflags("-Wall")</code> | 添加编译选项 |
| <code>target_link_libraries</code> | <code>add_links("bar")</code> | 添加链接库 |
| <code>find_package</code> | <code>add_packages("fmt")</code> | 查找并添加包 |
| <code>option()</code> | <code>option("BUILD_TESTS", true)</code> | 设置配置选项 |
| <code>install()</code> | <code>add_installfiles("bin")</code> | 安装目标 |

表 11.1: CMake 和 XMake 的常用命令对比

- `static`: 静态库
- `shared`: 动态库
- `phony`: 伪目标（不生成文件，只执行命令）
- `headeronly`: 头文件库（只包含头文件，没有源文件）

XMake 的规则（rules）是预定义的构建规则，可以通过 `add_rules()` 来添加。我们可以理解为“编译的流水线”，例如：

```
rule("embed")
    set_extensions(".txt")
    on_build_file(function(target, sourcefile)
        -- 把文本编译成 .o 里的字符数组
    end)
```

上述代码定义了一个名为 `embed` 的规则，作用是将文本文件编译成目标文件中的字符数组。XMake 提供了许多内置规则，例如 `mode.debug`、`mode.release` 等，可以直接使用。一般情况下，多个 `target` 可以共用一个规则。

11.2.2.4 调包

XMake 的包管理比 CMake 还要容易。我们可以非常简单地使用 `add_requires()` 来添加依赖包，非常方便。

XMake 的官方仓库有着九百多个常用的库。第一次编译的时候，XMake 会先查本地缓存有没有需要的包；如果没有，就自动下载预编译文件或者源码。然后，XMake 会自动设置 `include` 路径等必要的内容。

```
add_requires("fmt") -- 添加 fmt 依赖
target("app")
    set_kind("binary")
    add_files("src/*.cpp")
    add_packages("fmt") -- 链接 fmt 包
```

11.2.2.5 跨平台、交叉编译、远程编译

XMake 的跨平台做得很好，不需要手写 toolchain 文件等，只需要在命令行中指定平台和架构即可。例如：

```
$ xmake f -p windows -a x64 -m release # Windows 64-bit Release
$ xmake f -p linux -a arm64 --sdk=/opt/rpi
$ xmake
```

上述代码会在 Windows 上编译一个 64 位的 Release 版本，在 Linux 上编译一个 ARM64 的版本，并指定了 Raspberry Pi 的 SDK 路径。Raspberry Pi 是一个流行的单板计算机（“树莓派”，常用于前端开发和嵌入式开发等轻量级场景）。

另一方面，使用 XMake 运行远程编译功能也很容易：

```
$ xmake service --start          # 本机当编译服务器
$ xmake f --remote_build=y
$ xmake                          # 自动分发
```

上述代码会启动一个编译服务器，然后在本地编译时自动分发到远程服务器上进行编译。XMake 会自动处理远程编译的细节，用户只需要关注代码和配置即可。

11.2.3 打包发布

XMake 的打包发布也和 CMake 大同小异：

```
xmake package -f deb      # 生成 .deb
xmake package -f nsis     # Windows 安装向导
xmake package -f zip      # 绿色压缩包
```

11.2.4 常见坑

XMake 的常见坑和 CMake 类似：

- 缓存出错：`xmake f -c` 清理配置，比删 build 快。
- 多个版本包冲突：使用 `xmake require --info <package>` 查看已经缓存的包版本，使用 `xmake require --extra="{debug=true}" <package>` 强制重装。
- 调试脚本：使用 `xmake -vD` 输出详细日志。VS Code 安装 xmake 插件也能打断点。
- 懒惰（不想学 lua）：绝大多数场景只使用 6 个 API：`target`、`set_kind`、`add_files`、`add_includedirs`、`add_links`、`add_packages`。其他的现查都来得及。

11.2.5 从 CMake 迁移到 XMake

如果你已经有了一个 CMake 项目，想要迁移到 XMake。CMake 项目的结构：

```
project/  
- CMakeLists.txt  
- src/  
- tests/  
- thirdparty/fmt/
```

首先使用 `cmake2xmake` 工具粗略转换为 XMake 项目：

```
xmake create -P . -t cmake2xmake
```

然后手动调整生成的 `xmake.lua` 文件，主要是：

- 把 `add_subdirectory` 拆成多个 `target`；
- 把 `FetchContent` 转换为 `add_requires`；
- 把 `CMAKE_BUILD_TYPE` 转换为 `set_config("debug")` 或 `set_config("release")`；
- 把 `gtest_discover_tests` 转换为 `add_rules("test")` 等。

最后，保留旧的构建目录，并行验证新旧构建结果是否一致。

如果使用了 CI 等工具，需要将 CI 脚本中的相关命令从 CMake 转换为 XMake。

第四部分

附录

附录 A 迷你 ICS

本章节蒸馏自 CMU 与 PKU 的 ICS 课程，内容经过了大量的删减和改编，但尽可能地保留了原有的精华。

我在写第十章《用 GDB 调试》的时候，和 LCPU 的同学们在群聊里面吹了吹水。然后，我发现一个非常严重的问题：大多数人对于汇编、CPU、内存等概念都不甚了解，结果我写出来的一大堆东西全是鸡同鸭讲，堪比天书，这样的东西完全违背了我的本意。因此，我紧急中断了第十章的编写，决定先写一个 mini ics，来介绍一下计算机的许多基本原理。

刻意淡化理论知识的人最终又开始讲理论知识了吗，哈基臧，你这家伙……

我非常建议有时间的同学们阅读 CSAPP 这本经典教材，而且能读英文原版就读原版。这是一本非常好的教材，内容非常全面，值得一读。我这一章是蒸馏版，把 CSAPP 的内容浓缩成了五个问题，虽然说半小时就能看完、立刻就能用，但是毕竟是缺少了一大堆细节的。

A.1 信息怎么被表示？

在计算机上，信息是以二进制的形式被表示的，也就是 0 和 1。每一个数字都是一个二进制位（bit），8 个二进制位组成一个字节（byte）。于是我们就了解了：计算机上的数字只是约定好的 0-1 串，格式错了全完蛋。

为了表示方便，计算机上一般利用十六进制的两位来表示一个字节，例如十进制下的 11，写成二进制是 0b1011，写成十六进制是 0xB；其中 0b 表示后面这个是二进制数，0x 表示后面这个是 16 进制。有时候也可以看见八进制 013，这个开头 0 就表示后面这个是 8 进制数。

A.1.1 整数

整数在计算机中通常使用补码的形式来表示。对于一个 n 位（这个 n 指的是二进制位）的有符号整数，最高位是符号位，其他 $n-1$ 位用于表示数值；对于无符号整数，所有 n 位都用于表示数值。

例如一个四位的有符号整数，实际数值是各位数值相加，最高位表示 -2^3 ，剩下的三位表示 $2^2 + 2^1 + 2^0$ ，因此它的数值范围是 -8 （0b1000）到 7 （0b0111）。而一个四位的无符号整数，所有四位都表示数值，因此它的数值范围是 0 到 15 。

计算机中，常见的整数类型 `int` 指的通常是 32 位的整数（也就是 4 个字节）。于是我们就知道了，`int` 类型的最大值是 $2^{31} - 1$ ，最小值是 -2^{31} 。而无符号整数类型 `unsigned int`（以下简称为 `uint`）的最大值可以表示为 $2^{32} - 1$ ，最小值为 0 。在 C/C++ 中，`int` 的最大值有宏定义 `INT_MAX`，最小值有宏定义 `INT_MIN`，无符号整数的最大值有宏定义 `UINT_MAX`。

我们知道，十进制中的加法可能会产生进位。二进制加法也不例外，例如 $1+1=10$ 。但是在刚刚的讲解中，我们发现对于 `uint` 类的变量，最长只有 32 位，那么如果两个变量相加，可

能会超过 32 位，比如两个“1 后面跟 31 个 0”这样的整数相加，结果是 1 后面跟 32 个 0，超过了 32 位，这个时候就会发生溢出（overflow）。在 C/C++ 中，整数溢出是未定义行为（undefined behavior），因此我们一般不能依赖于溢出后的结果。

一般情况下，计算机对溢出行为的处理是取最低 32 位的结果。例如两个 `uint` 类型的变量相加，如果结果超过了 32 位，那么计算机会将结果的低 32 位作为最终结果返回。对于有符号整数（`int`），如果发生溢出，结果也是未定义行为，但通常情况下，计算机会将结果的低 32 位作为最终结果返回。（当然如果一开始就是两个更长的整数相加，例如两个 64 位的整数相加发生溢出时，就会返回低 64 位的结果。）

A.1.2 浮点数

浮点数是另一个表示实数的方式。浮点数在计算机中通常使用 IEEE 754 标准来表示。一个 32 位的浮点数（单精度）由三部分组成：符号位、指数位和尾数位。浮点数的本质是科学计数法的变种。

- 符号位（1 位）：表示数值的正负。
- 指数位（8 位）：表示数值的指数部分。
- 尾数位（23 位）：表示数值的有效数字部分。是一个不大于 1 的二进制小数。

浮点数的表示方式是这样的：数值 = 符号 \times (尾数 + 1) $\times 2$ 的指数次方。具体说来比较复杂，作为 Mini ICS，我们只需要知道浮点数有误差就可以了（这是因为将数字从 10 进制转换为 2 进制可能出现除不尽的状况）。因此，**不可以利用浮点数来进行货币运算。**

有不少算法依赖于浮点数有精度这一客观事实，例如最大流算法中 Ford-Fulkerson 算法在计算机上的有限终止性证明就利用了这一特性。

A.2 程序怎么跑起来？

我们这边只讲 C 系语言。

C 系语言写出的程序，一般情况下是由源代码（.c 或 .cpp 文件）编译成目标代码（.o 或 .obj 文件），然后链接成可执行文件（.exe 或 .out 文件）。这个过程通常分为三个步骤：预处理、编译和链接。

A.2.1 预处理

预处理是对源代码进行一些简单的文本替换和宏展开。预处理器会处理一些指令，例如 `#include`、`#define` 等。同时，预处理会除去源代码中的所有注释。从某种程度上来说，预处理后的代码和源代码完全等价。

A.2.2 编译和汇编

计算机通过编译器将预处理后的代码转换为汇编码（能读懂一部分），然后再利用汇编器把汇编码转变成机器码（二进制码，人几乎读不懂）。编译器会将源代码转换为目标代码（.o 或.obj 文件），这个目标代码是特定于某个操作系统和处理器架构的。

编译器会将源代码中的每个函数、变量等转换为机器码指令，并生成符号表（symbol table）来记录这些符号的地址。

编译器还会进行一些优化，例如常量折叠、循环展开等，以提高程序的执行效率。默认情况下，gcc 编译器会进行一些基本的优化，但如果需要更高的优化级别，可以使用 -O2 或 -O3 选项。优化可能导致编译出错，但是一般情况下不至于出现这种情况。

A.2.3 链接

链接是将多个目标代码文件和库文件合并成一个可执行文件的过程。链接器会将目标代码中的符号解析为实际的地址，并将它们合并成一个完整的可执行文件。链接也是一个比较复杂的过程，在这里我们不深入讨论。

A.2.4 常见汇编码

x86-64 架构是 Intel 的 64 位架构，在现代计算机非常常见。我们会利用该架构来简单解释汇编码的语法。

在深入介绍汇编码之前，我们要先了解一下 CPU 的寄存器。寄存器是 CPU 内部的高速存储器，用于存储临时数据和指令。x86-64 架构有 16 个通用寄存器（RAX、RBX、RCX、RDX、RSI、RDI、RBP、RSP、R8-R15），每个寄存器都是 64 位的。CPU 将指令和数据加载到寄存器中，利用控制单元 CU 来执行指令，利用算术逻辑单元 ALU 来进行计算。

x86-64 架构的汇编码通常由操作码（opcode）和操作数（operand）组成。操作码是指令的名称，操作数是指令的参数。以下是一些常见的汇编码指令：

- **mov**: 将数据从一个寄存器或内存位置移动到另一个寄存器或内存位置。
- **add**: 将两个寄存器或内存位置的值相加，并将结果存储在第一个寄存器或内存位置中。
- **sub**: 将一个寄存器或内存位置的值减去另一个寄存器或内存位置的值，并将结果存储在第一个寄存器或内存位置中。
- **jmp**: 无条件跳转到指定的标签。
- **call**: 调用函数，将返回地址压入栈中。
- **ret**: 从函数返回，弹出栈顶的返回地址。
- **cmp**: 比较两个寄存器或内存位置的值，并设置标志位。
- **je**、**jne**、**jg**、**jl** 等: 条件跳转指令，根据比较结果跳转到指定的标签。

在 x86-64 架构中，假如我们想要把某个整数从寄存器 RAX 移动到寄存器 RBX，可以使用以下指令：

```
mov rbx, rax
```

或者说我们想要 `call` 一个函数，假设函数名为 `foo`，可以使用以下指令：

```
call foo
```

这是最基本的一些汇编码指令。对于其他的机器（例如 `Arm` 架构），汇编码的语法和指令可能会有所不同，但基本原理是相似的。

A.3 内存怎么被管理？

在第一章中我们就提到过，计算机的 `CPU` 从内存取指令和数据，执行指令，然后把结果再存回内存。但是现在的问题是：对于一些用户，我们可能会在后台挂着 114514 个进程，这些进程都需要使用内存。但是这些进程所占用的内存可能远远大于实际物理内存的大小。那么，计算机到底怎么管理内存，使得每个进程都能正常运行？

A.3.1 虚拟内存

计算机使用虚拟地址空间来管理内存。每个进程都有自己的虚拟地址空间，都认为自己是从 0 号地址开始用内存的。操作系统通过虚拟内存技术，将虚拟地址映射到物理地址。这样，每个进程都可以独立地使用内存，而不需要关心其他进程的内存使用情况。

打个比方：某高度智能运行的图书馆给每一本书贴一个标签，标签上写着书的编号；但是读者不需要管实际上书放在哪里，只需要知道自己的书编号就行了。

A.3.2 磁盘交换区

当物理内存不足时，操作系统会将一些不常用的页面（`page`）从物理内存（快）中换出到磁盘上的交换区（`swap space`）（慢）。我们可以理解为，图书馆把常用的书放在书架上，而不常用的书放在仓库里。这样，当需要使用不常用的书时，图书馆可以从仓库中取出书来。

A.3.3 页面、页表、缺页异常

页面是虚拟内存的基本单位，通常是 4KB 或 8KB。操作系统使用页表（`page table`）来管理虚拟地址和物理地址之间的映射关系。如果我们查到了一个虚拟地址对应的物理地址，但是这个页面不在物理内存中，那么就会发生缺页异常（`page fault`）。操作系统会捕获这个异常，然后从磁盘上的交换区中加载相应的页面到物理内存中。同样利用图书馆打比方：图书馆有一本书的编号，但是这本书不在书架上，而是在仓库里。图书馆会去仓库里取出这本书，然后放到书架上。

如果最近的书架满了，怎么办呢？一个常见的策略是使用 LRU（Least Recently Used）算法，淘汰最近最少使用的页面。也就是图书馆会把最近很久没被借阅的书从书架上拿下来，腾出空间来放新书。

A.3.4 内存分配器

内存分配器（memory allocator）是操作系统或运行时库提供的，用于管理进程的内存分配和释放。常见的内存分配器有 `malloc`、`free` 等函数。内存分配器会维护一张空闲内存块的列表，当进程请求分配内存时，分配器会从空闲列表中找到合适的内存块，并将其分配给进程。

假如我们在 C 系语言用了 `malloc` 函数分配了许多字节的内存，这时候操作系统不会直接分配物理内存，而是分配虚拟内存。操作系统会在页表中记录这个虚拟地址和物理地址的映射关系。而真正给物理页，是“用到才给”，也就是当我们第一次访问这个虚拟地址时，操作系统会触发缺页异常，然后将对应的物理页加载到内存中。

这也可以解释为什么我 `malloc` 了 10GB 内存但是电脑依然流畅运行：还没真正分配呢。

A.3.5 一个例子

假如，我们打开了微信。这时候，操作系统给微信分配了 1GB 的虚拟地址；但是实际上只先分配很少数的物理内存来加载常用数据，剩下的全在磁盘交换区。然后，假设我们又切换到其他应用程序（例如去 B 站看视频），这时候 B 站会获得许多新的物理页，而微信的物理页会被换出去一部分。

现在老板给我发消息了，我打开微信，点击几下，这时操作系统触发一个缺页异常，然后微信数据又被拉回内存。如此反复，整个过程只在数十毫秒内完成，使得我们几乎感觉不到延迟。

因此以后谁再拿“某某手机/某某电脑真好，同时开十个 APP 也不卡”来宣传产品的时候，你可以跟他讲讲虚拟内存！

A.4 缓存怎么被管理？

有时候在做超大矩阵乘法的时候，我们发现仅将循环从按列换成按行，或者从按行换成按列，就能将程序的运行速度提升许多。这是为什么呢？

我们在第一章中提到过，现在的计算机内存的速度已经远远跟不上 CPU 的速度了，为了进一步压榨性能，计算机采用了缓存技术。本节就会简单介绍一下缓存的原理。

A.4.1 缓存的分级

CPU 的缓存 (cache)，又叫高速缓存，是一种小容量、高速度的存储器，用于存储经常使用的数据和指令。缓存通常分为三级：L1、L2 和 L3 缓存。

- L1 缓存：位于 CPU 内部，速度最快（1 纳秒级），但容量最小，通常为 32KB 或 64KB。
- L2 缓存：位于 CPU 内部或外部，速度较快（3 到 5 纳秒级），容量较大，通常为 256KB 或 512KB。
- L3 缓存：位于 CPU 外部，速度较慢（10 纳秒级），但容量最大，通常为 2MB 或更大。

再往后就轮到内存了，内存的速度大约是 100 纳秒级别。我们可以利用小卖部来理解，L1 缓存有点像学校每层楼都有的贩卖机，L2 有点像每栋楼都有的小超市，L3 有点像学校的大超市，而内存有点像学校外面的供货仓库。

A.4.2 缓存行和局部性原理

缓存是以缓存行为单位进行存储的。缓存行 (cache line) 是缓存中最小的存储单位，通常为 64 字节。当 CPU 访问内存时，如果访问的地址在某个缓存行内，那么这个缓存行就会被加载到缓存中。我们可以这么理解：当我们去贩卖机只会买一瓶饮料，但是贩卖机补货的时候是一补补一箱。只要把经常一起用的数据放在连续的一个缓存行上，就能一口气全带走，非常方便。

缓存的局部性原理是指程序在执行过程中，访问数据的地址往往具有一定的规律性。局部性分为时间局部性和空间局部性。时间局部性指的是最近访问的数据很可能会再次被访问；空间局部性指的是如果访问了某个地址，那么很可能会访问相邻的地址。

因此，我们在编写程序时，应该尽量利用局部性原理，将相关的数据放在一起，减少缓存未命中 (cache miss) 的情况。

A.4.3 组相联和标签

缓存通常采用组相联 (set-associative) 方式来存储数据。组相联缓存将缓存分为多个组，每个组包含多个缓存行。当 CPU 访问某个地址时，首先计算出该地址对应的组，然后在该组内查找是否有对应的缓存行 (way)。如果有，就命中 (hit)，否则就未命中 (miss)，需要从内存中加载数据。

每一个缓存行都会贴两个标签，一个是 tag 记录该缓存行对应的内存地址的高位部分，另一个是 valid 位记录该缓存行是否有效。在 CPU 要读一个地址的时候，CPU 会先计算出该地址对应的组，然后在该组内查找是否有有效的缓存行。如果有，就命中；如果没有，就未命中，需要从内存中加载数据。

A.4.4 未命中常见工作流程

当 CPU 访问的地址不在缓存中时，就会发生读不命中。这时，CPU 需要从内存中加载数据到缓存中。加载数据的过程通常分为以下几个步骤：

1. L1 缓存没有，去 L2 缓存查找；L2 缓存没有，去 L3 缓存查找；L3 缓存没有，去内存查找。
2. 如果找到了，就将数据加载到 L1 缓存中，并更新 L1 缓存的标签和有效位。
3. 如果 L1 缓存满了，就需要选择一个缓存行进行替换。通常使用 LRU (Least Recently Used) 算法来选择最近最少使用的缓存行进行替换。L2 和 L3 缓存也会进行类似的替换操作。

如果 CPU 试图往缓存中写入数据，而该缓存行已经被其他数据占用，那么就触发了写不命中。一般有一些策略来处理写不命中，例如写回 (write-back) 和直写 (write-through)。写回策略是将数据先写入缓存，等到缓存行被替换时再写回内存；直写策略是直接将数据写入内存。写分配和不写分配是指在写不命中时，是否将数据加载到缓存中。写分配会将数据加载到缓存中，而不写分配则不会。

A.4.5 怎么提高代码运行效率？

于是我们讲完了缓存，现在就可以来解释为什么有时候换个循环顺序就能提速一倍了。

一般情况下，一个二维数组，按行扫的时候，相邻的元素在内存连续，64 个字节一口气全都搬进 L1，命中率非常高；而按列扫的时候，相邻的元素在内存中并不连续，可能需要多次访问 L2 和 L3 缓存，命中率就会降低。

另一种方式就是手动对齐数据，例如利用结构体来对齐数据。这样可以防止诸如 double 等长数据类型被拆成好几个缓存行，手动对齐可以强制把这样的 64 位数据按进一个缓存行，速度至少翻倍。

简单地说，只要让常用数据挤在同一箱里，就能让小卖部永远有货。

A.5 系统怎么被调用？

有时候我们电脑死机了，或者程序崩溃了，终端报错“Segmentation Fault”（段错误）。这时候，操作系统到底做了什么？为什么会发生段错误？我们来分析一下“系统调用”就知道了。

A.5.1 为什么要有这个系统调用？

一般情况下，程序运行时仅会访问分配给自身的内存中的数据 and 指令。如果程序试图访问未分配的内存区域，或者试图修改只读内存区域，就会发生段错误。这是出于安全性和稳定性的考量：操作系统需要确保每个进程都只能访问自己的内存区域，不能访问其他进程的内存区域。这样可以有效防止恶意程序破坏系统的稳定性和安全性。

但是有些情况下，程序确实需要访问一些特殊的内存区域，例如访问硬件设备、操作系统内核等。为了解决这种问题，操作系统提供了系统调用（system call）来处理内存访问。

简单地说，可以把操作系统看成化学实验室管理员，管理危化品。把危化品直接扔给学生非常危险，学生必须先向管理员填表申请，管理员检查后再给学生。填的这个表就是系统调用。

A.5.2 系统调用长什么样？

以 Linux 为例，一个系统调用往往包括系统调用号（放在 RAX）、参数（放在 RDI、RSI、RDX 等寄存器，包括要干什么、干多少次、怎么干）、触发指令（syscall）和返回值（放在 RAX）。当程序需要进行系统调用时，会使用 syscall 指令来触发系统调用。系统调用的种类很多，例如 read、write、open、close 等，每个系统调用都有一个唯一的系统调用号。

以实验室为例，上述填表就包括：编号（系统调用号）、申请的危化品（参数）（包括要什么、要多少、放哪）、申请的指令（syscall），以及管理员的批复（返回值）。当学生需要使用危化品时，就会向管理员提交申请，管理员检查后返回批复。

A.5.3 系统调用的处理流程

当程序触发系统调用时，CPU 会将当前的执行状态保存到内核栈中，然后切换到内核态（kernel mode）。在内核态下，操作系统会根据系统调用号找到对应的系统调用处理函数，并执行相应的操作。处理完成后，操作系统会将结果返回给用户态（user mode），并恢复之前保存的执行状态。

以 printf("Hello") 为例，这个东西实际上是做了一次系统调用 write(1, buf, 5)。现在 glibc 把这玩意塞进寄存器触发 syscall 指令，然后 CPU 就切换到内核态。

之后，CPU 在内核态办事：检查文件描述符 1 是否可写，发现可以写，就把 Hello 这 5 个字节写入到文件描述符 1 对应的设备（通常是终端）。

写完后，CPU 会将结果（成功写入的字节数，在这里是 5）放回 RAX 寄存器，然后切换回用户态。然后代码就会继续执行了。

A.5.4 系统调用的代价与实践尝试

系统调用虽然显著提升了系统的安全性，但是也带来了巨大的性能损失。因为每次系统调用都需要切换到内核态，这个过程需要保存和恢复 CPU 的状态，涉及到上下文切换（context switch），会消耗大量的时间（比普通函数调用慢一百倍到一千倍）。因此，系统调用的次数越少，程序的性能就越好。

在代码实践中，我们最简单的优化方式就是尽可能减少系统调用的次数，例如禁止频繁的读写等；应该尽可能多的利用批量读写或者缓存以加快速度。

附录 B C/C++ 高速入门

本章会快速带领大家过一遍 C 系的基本语法和常用特性，除了用作预习材料以外，还可以在期末考试复习的时候来快速回顾其基本语法与常用的高级特性。

这里直接从 C++ 开始讲起，因为 C++ 是 C 的超集，C 的语法在 C++ 中完全可以使用。

由于 C++ 的语法和特性极多、语法也较为复杂，因此本章节可能会显得有些长、节奏也非常快。

让我们开始吧：

B.1 C++ 的基本语法

第一次写 C++ 的时候，我们只需要了解一些最基本的语法和特性。记住以下三件事：

程序有入口；先存再算；算完告诉外面。剩下的内容，都跟我们说话一样，只不过是使用 C++ 的语法来表达，而且我们说话的句号在 C++ 中是分号。

一个简单的 C++ 程序如下：

```
#include <iostream>
using namespace std;

int main() {
    int age = 18;
    cout << "I am " << age;
    return 0;
}
```

逐行拆解之：

- `#include <iostream>`：告诉编译器，我要用输入输出工具。
- `using namespace std;`：使用标准命名空间，这样我们就可以直接使用 `cout` 而不需要加上 `std::` 前缀。
- `int main()`：程序的入口函数，告诉电脑程序从这里开始执行。`int` 表示这个函数返回一个整数值。
- `int age = 18;`：跟电脑说，我要在内存找个地方放个整数，这个地方叫 `age`，放个 18 进去。
- `cout << "I am " << age;`：把东西一股脑送送到屏幕上。
- `return 0;`：返回 0，告诉操作系统：一切 OK。

以上就是骨架。接下来该往骨架里面填肉了：

B.1.1 基本变量及其运算

变量用来存储数据，可以变化；声明格式是“先写类型，再写名字”。

常见的变量类型有：

- `int`：整数类型，通常是 32 位（二进制位数）。
- `double`：双精度浮点数，通常是 64 位。
- `char`：字符类型，通常是 8 位整数，表示一个字符。也可以用于表示整数。
- `bool`：布尔类型，表示真或假。
- `string`：字符串类型，表示一串字符。

对于变量的运算就跟数学差不多，比方说

```
int a = 10;
int b = 20;
int c = a + b;
c = a * 2;
c += 5;
```

`int c = a + b` 的意思是“我要创建一个变量 `c`，把 `a+b` 的结果放进去”。可以看到，从这一行以后再提到 `c`，就不需要再写 `int` 了，因为电脑已经知道 `c` 是个什么东西了。

下一行 `c = a * 2` 的意思是“我要把 `a` 乘以 2 的结果放到 `c` 里面，`c` 以前不管是什么我都不要了”，而再下一行 `c += 5` 的意思是“我要把 `c` 加上 5”。在上述代码中，我们发现变量 `c` 的值会随着每一行代码的执行而变化，例如第三行代码执行后，`c` 的值变成了 30；第四行代码执行后，`c` 的值变成了 20；第五行代码执行后，`c` 的值变成了 25。所以说 `c` 是一个变量。

变量的值也可以在声明时不确定（初始化），例如 `int a`；这样也是可以的。如果在声明的时候不初始化变量的值，那么这个变量的初始值将会是一个随机值，这个值取决于内存中该位置之前存储的内容。

让我们看看常见的运算符：

- 四则运算：`+`（加）、`-`（减）、`*`（乘）、`/`（除）。注意，除法运算中，如果两个整数相除，结果仍然是整数，余数会被舍弃。
- 取模：`%`，表示取余数。例如 `5 % 2` 的结果是 1，因为 5 除以 2 的余数是 1。
- 自增和自减：`++`（自增）和`--`（自减）。例如，`a++` 表示将 `a` 的值加 1，`b--` 表示将 `b` 的值减 1。

不要纠结 `i++` 和 `++i` 的区别，初学者完全可以认为这两个和 `i += 1` 没有区别。一个饱受诟病的题目“`i = 3, i++ + i++ = ?`”答：这个题目是错误的，至少是不良定义的。不同的编译器对上述代码的处理方式不同。

我们非常不建议大家弄出这种代码：`a = b++`，这种在运算或者赋值中使用自增的代码令人恼火。如果你想要先用 `b` 的值再加 1，可以写成 `a = b; b++`；如果你想要先加 1 再用 `b` 的值，可以写成 `b++`；`a = b`；。

更现代的程序员往往使用 `i += 1` 来代替 `i++`。

是不是非常简单？

B.1.2 注释

注释是代码中的说明文字。它们会被编译器忽略，因此注释完全是给编写者和读者看的。

在 C++ 中，注释有两种方法来写：

- 单行注释：使用 `//`，例如 `// 这是一个单行注释`。注释符号后面的内容会被编译器忽略，直到行尾为止。
- 多行注释：使用 `/* ... */`，例如 `/* 这是一个多行注释 */`。两个注释符号之间的内容会被编译器忽略，可以跨越多行。

在阻止部分代码执行的时候，我们一般不习惯于直接删除这些代码，而是使用注释。这样做的好处是可以留痕，便于以后的恢复（解注释）；这就是程序员们常说的“注释掉”代码。在 VS Code 等编辑器中，常用的一键注释是 `Ctrl + /`，它会自动将光标所在的一行或多行代码注释掉。

B.1.3 常量

常量指的是一旦确定就不会改变的值，任何试图对常量进行更改的操作都会使得编译不通过。常量的值应当在声明时确定，可以通过赋值或者计算得到。

声明常量的方法和声明变量差不多，但是要在最前面加上 `const` 关键字，如：

```
const int MAX_VALUE = 100;
const int P = a + b; // 这里的a和b可以是变量
// P = 10 // 这行代码编译不通过，因此要注释掉
```

以上代码的意思是：我要创建一个常量 `MAX_VALUE`，它的值是 100。注意，常量的名字通常使用大写字母来表示，以便于和变量区分。

如果常量的值无需在运行时确定，可以使用编译时常量，这样的常量在编译的时候值就确定了，不过因此也需要在定义中就写明它的值。编译时常量的声明方法和常量类似，只是把 `const` 换成 `constexpr`。编译时常量也可以通过计算得到，计算在编译时进行，可以节省程序运行时间。但是，用于计算的值必须也都是编译时常量。下文是编译时常量的几个例子。

```
constexpr double E = 2.71828;
constexpr double PI = 3.14159;
constexpr double EPI = E * PI;
```

B.1.4 判断和循环

有时候，代码需要根据不同的条件或不同的输入来执行不同的操作；有时候，一段代码需要执行很多次，但是并不知道具体要执行多少次。C++ 提供了条件语句和循环语句来实现这些功能。

B.1.4.1 条件表达式

条件表达式是一个布尔表达式，它的值要么是 `true`（真），要么是 `false`（假）。在 C++ 中，条件表达式通常用于控制程序的执行流程。一般情况下，认为 `false` 等价于 0，`true` 等价于 1。

常见的一些条件表达式包括：

- `==`：等于运算符。
- `!=`：不等于运算符。
- `<`：小于运算符。
- `>`：大于运算符。
- `<=`：小于等于运算符。
- `>=`：大于等于运算符。
- `&&`：逻辑与运算符。如果前后两个条件都为真，则结果为真；有一个是假的话，结果为假。
- `||`：逻辑或运算符。如果前后两个条件至少有一个为真，则结果为真；如果两个都为假，结果为假。
- `!`：逻辑非运算符。如果后面跟着的条件是真的，那么结果为假；反之为假。

在 C++ 中，与或非运算符是有一定的运算顺序的。一般情况下，逻辑非运算符的优先级最高，其次是逻辑与运算符，最后是逻辑或运算符。不过笔者非常不建议同学们背诵这个顺序；实际在工程上不仅不建议大量嵌套使用这些运算符，而且遇事不决可以加括号——括号可比记运算顺序靠谱得多了！

B.1.4.2 条件语句

最常见的条件语句是 `if` 语句。它的基本格式如下：

```
if (条件) {  
    // 条件为真时执行的代码  
}  
else if (另一个条件) {  
    // 另一个条件为真时执行的代码  
}  
else {  
    // 以上条件全部为假时执行的代码  
}
```

以上代码可以有很多个 `else if` 分支，也可以没有 `else` 分支。意思是：我执行到 `if` 这一行的时候，检查后面的条件。如果条件为真，那么执行第一个大括号内的代码，剩下的全都不执行；如果条件为假，那么检查下一个 `else if` 的条件，如果为真就执行它的大括号内的代码，剩下的全不执行；如果所有的条件都为假，那么执行 `else` 大括号内的代码。

例子：

```
if (age < 18) {
    cout << "未成年";
}
else if (age < 60) {
    cout << "成年人";
}
else {
    cout << "老年人";
}
```

一目了然，不言而喻。这个 `age` 变量可以是前面提到的许多类型。

B.1.4.3 三元表达式

三元表达式也是一种条件表达式，只不过它可以在一行代码中完成条件判断和结果返回，因此显得更简洁。它通常用于简单的条件判断和赋值操作。它的基本格式如下：

条件 ? 真值 : 假值

以上代码的意思是：如果条件为真，整个表达式的值和真值一样；否则，整个表达式的值和假值一样。它非常适合简单的条件判断和赋值操作，但是我们不建议在复杂的条件判断中使用它或者者嵌套使用它，这样会大大降低代码的可读性。

比方说，我们可以用它来判断一个数是奇数还是偶数：

```
int n = 5;
string result = (n % 2 == 0) ? "偶数" : "奇数";
```

以上代码的意思是：如果 `n` 是偶数，就把字符串“偶数”赋值给 `result`；否则把字符串“奇数”赋值给 `result`。

如果使用 `if` 语句来实现同样的功能，可以写成：

```
int n = 5;
string result;
if (n % 2 == 0) {
    result = "偶数";
} else {
```

```
    result = "奇数";  
}
```

B.1.4.4 切换语句

有时候，我们需要根据一个变量的值来执行不同的操作。C++ 提供了 `switch` 语句来实现这个功能。它的基本格式如下：

```
switch (变量) {  
    case 值1:  
        // 当变量等于值1时执行的代码  
        break;  
    case 值2:  
        // 当变量等于值2时执行的代码  
        break;  
    default:  
        // 当变量不等于任何case的值时执行的代码  
}  

```

以上代码的意思是：检查变量的值，如果等于值 1，就执行第一个大括号内的代码；如果等于值 2，就执行第二个大括号内的代码；如果都不等于，就执行 `default` 大括号内的代码。可以有任意多个 `case` 分支，也可以没有 `default` 分支。

注意，`break` 语句用于跳出 `switch`，这个是必须的。

例子：

```
switch (day) {  
    case 1:  
        cout << "星期一";  
        break;  
    case 2:  
        cout << "星期二";  
        break;  
    case 3:  
        cout << "星期三";  
        break;  
    // .....其他的，基本一个写法  
}  

```

这也一目了然不言而喻了。

B.1.4.5 for 循环语句

for 循环是灵活度极高的循环语句。它的基本格式如下：

```
for (初始化; 条件; 更新) {  
    // 循环体  
}
```

以上代码的意思是：先执行初始化语句，然后检查条件是否为真。如果为真，就执行循环体内的代码，然后执行更新语句。接着再检查条件，如果为真就继续执行循环体，否则跳出循环。比方说，我们需要打印 1 到 10 的数字，可以这样写：

```
for (int i = 1; i <= 10; i++) {  
    cout << i << " ";  
}
```

这段代码的意思是：先初始化一个变量 *i* 为 1，然后检查 *i* 是否小于等于 10。如果是，就打印 *i* 的值，然后将 *i* 加 1。接着再检查 *i* 是否小于等于 10，如果是就继续打印，否则跳出循环。

这个 `int i = 1` 可以在其他地方声明过，那么这里就遵从“先声明后使用”的原则，不需要再写 `int` 了。

B.1.4.6 while 和 do-while 循环语句

while 循环是另一种常见的循环语句。它的基本格式如下：

```
while (条件) {  
    // 循环体  
}
```

以上代码的意思是：先检查条件是否为真。如果为真，就执行循环体内的代码，然后再次检查条件。如果条件仍然为真，就继续执行循环体，否则跳出循环。比方说，我们需要打印 1 到 10 的数字，可以这样写：

```
int i = 1;  
while (i <= 10) {  
    cout << i << " ";  
    i++;  
}
```

以上内容与 for 循环的例子是等价的。

do-while 循环与 while 循环类似，但它会先执行一次循环体，然后再检查条件。它的基本格式如下：

```
do {
```

```
// 循环体
} while (条件);
```

这样可以保证循环体至少执行一次。比方说，我们需要打印 1 到 10 的数字，可以这样写：

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 10);
```

B.1.5 break 和 continue 语句

有时候，我们需要在循环中提前跳出循环或者跳过当前的迭代。C++ 提供了 `break` 和 `continue` 语句来实现这些功能，它们也叫做循环控制语句。

`break` 语句用于跳出循环，通常用于满足某个条件时立即结束循环。例如：

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break; // 当i等于5时跳出循环
    }
    cout << i << " ";
}
```

上述代码如果不写 `break` 一句，那么会输出 1 到 10；如果写了 `break` 一句，那么会输出 1 到 4。

而 `continue` 语句只用于跳过当前循环，继续下一次迭代。例如：

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        continue; // 当i等于5时跳过当前迭代
    }
    cout << i << " ";
}
```

上述代码的输出应该是 1 2 3 4 6 7 8 9 10。因为当 `i` 等于 5 时，`continue` 语句会跳过当前迭代。

B.1.6 输入输出

在 C++ 中，我们建议使用更安全的输入输出流 `cin` 和 `cout` 来进行输入输出操作。它们分别用于从标准输入（通常是键盘）读取数据和向标准输出（通常是屏幕）打印数据。

`cin` 和 `cout` 的基本用法如下：


```

#include <iostream>
using namespace std;
int main() {
    int age;
    cout << "请输入你的年龄: "; // 输出提示信息
    cin >> age; // 从标准输入读取数据
    cout << "你输入的年龄是: " << age << endl; // 输出读取到的数据
    return 0;
}

```

以上代码的意思是：先输出提示信息“请输入你的年龄：”，然后从标准输入读取一个整数值并存储到变量 `age` 中。接着输出“你输入的年龄是：”以及读取到的年龄值。

C 风格的输入输出分别是 `printf` 和 `scanf`，它们的用法如下：

```

#include <cstdio>
// 也可以写成 #include <stdio.h>, 但不推荐
using namespace std;
int main() {
    int age;
    printf("请输入你的年龄: "); // 输出提示信息
    scanf("%d", &age); // 从标准输入读取数据
    printf("你输入的年龄是: %d\n", age); // 输出读取到的数据
    return 0;
}

```

以上代码中，`%d` 的意思是“这里要输出一个整数”，而 `&age` 的意思是“把 `age` 的地址传给 `scanf`”。前者叫做“格式化输出”，后者叫做“地址传递”。`printf` 和 `scanf` 的速度更快，因为不需要流操作；但是它们存在一些安全隐患，例如格式化字符串攻击和缓冲区溢出等问题。现代 C 编程中，更推荐使用 `scanf_s` 和 `printf_s` 来代替 `scanf` 和 `printf`，它们允许一个额外的参数来指定缓冲区的大小，从而避免缓冲区溢出的问题。

不过，虽然在做题的时候确实可以使用 `scanf` 和 `printf` 来压榨时间，但是我们仍然建议在 C++ 工程上使用更安全的 `cin` 和 `cout`。

实际上，`cin` 和 `cout` 和 `printf` 和 `scanf` 区别巨大。后者是一个函数，而前者是一个“流对象”（可以理解为一个“东西”而不是一个“手段”）。它们是 C++ 标准库中的流对象，真正负责输入输出的实际上是 `<istream>` 头文件中的 `istream::read` 和 `<ostream>` 头文件中的 `ostream::write` 方法，它们被封装进 `>>` 和 `<<` 这两个运算符（流运算符），和我们的加减乘除等运算符一样。这两个运算符必然是返回流对象的一个引用，因此可以链式调用。特别的，当输入失败的时候，会返回流对象的一个“失败”状态，因此可以通过 `cin.fail()` 来判断输入是否成功，也可以通过布尔上下文转换（例如 `while(cin>>n)`）来判断输入是否

成功。

实际上，流运算符也不是它们的原本样子。它们原本是右移和左移运算符：例如 $a \ll b$ 是对 a 进行左移操作，将 a 的二进制表示整体向左边移动 b 位，右边补 0；右移类似（只不过对于有符号整数最高位是 0 补 0，是 1 补 1；无符号整数默认补 0）。在 `<iostream>` 头文件中，这两个运算符被重载了，使得它们可以用于流对象，进而辅助执行输入输出操作；也正因此，我们需要引用上述头文件才能使用它们。不过值得庆幸的是，我们可能一辈子都不会用到它们的原本样子。

头文件 `<stdio.h>` 是 C 的头文件，而 `<cstdio>` 是这个头文件在 C++ 中的移植版本。两者内容完全一致，只不过 `<cstdio>` 使用了 C++ 的命名空间 (`std`)；但是由于 C++ 是 C 的超集，因此大多数实现也允许不套命名空间直接用 `printf` 等。在现代风格的 C++ 编程中，我们通常使用 `<iostream>` 或 `<cstdio>` 来进行输入输出操作，而不是使用 `<stdio.h>`。

这些看起来都非常简单。而以上内容就是 C++ 的全部基本语法了：是的，你已经学完了！让我们来做个小练：

B.1.7 基本语法小练

角谷猜想是一个有趣的数学问题：从任意整数开始，如果他是奇数就乘以 3 加 1，如果是偶数就除以 2，如此反复循环，最终一定会得到 1。目前还没有人证明这个猜想，但我们可以用 C++ 来验证一些具体值。

请编写一个 C++ 程序，输入一个整数 n ，然后输出这个整数经过角谷猜想的处理后，最终得到 1 的过程。要求输出每一步的结果。例如，输入 $n=6$ 时，输出应该是：6, 3, 10, 5, 16, 8, 4, 2, 1。

B.1.7.1 边写边说

一句一句地看：任意整数，奇数乘以 3 加 1，偶数除以 2。这一段代码写起来很方便。我们知道，整数的奇偶性可以通过取模来判断：如果 $n \% 2 == 0$ ，那么 n 是偶数；否则 n 是奇数。

因此仅仅这句话，从人的自然语言到代码语言的转换就非常简单了。

```
int n;
if(n % 2 == 0) {
    n /= 2; // 偶数除以2
} else {
    n = n * 3 + 1; // 奇数乘以3加1
}
```

或者使用三元表达式：

```
n = (n % 2 == 0) ? (n / 2) : (n * 3 + 1);
```

然后是下一句话：如此反复循环。这说明我们至少需要写一个循环语句。再看下一句：最终一定会得到 1。

这样我们就明确了：我们需要写一个循环语句，跳出循环的条件是 n 等于 1。于是我们可以写成：

```
while(n != 1){  
    // 处理 n 的代码  
}
```

再看下一句：输入一个整数 n ，输出每一步的结果。这说明我们需要一个输入语句和一个输出语句。输入语句可以用 `cin`，输出语句可以用 `cout`。

题目读完了，那么我们就可以把这些代码组合起来了：

```
int n;  
cin >> n; // 输入一个整数n  
cout << n; // 输出初始值  
while(n != 1) {  
    if(n % 2 == 0) {  
        n /= 2; // 偶数除以2  
    } else {  
        n = n * 3 + 1; // 奇数乘以3加1  
    }  
    cout << "□" << n; // 输出每一步处理后的结果  
}
```

这就是基本的代码框架。下一步，我们结合一开始说的话：程序要有入口，先存再算，算完告诉外面。于是我们可以真正地完成这段代码：

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int n;  
    cin >> n; // 输入一个整数n  
    cout << n; // 输出初始值  
    while(n != 1) {  
        if(n % 2 == 0) {  
            n /= 2; // 偶数除以2  
        } else {  
            n = n * 3 + 1; // 奇数乘以3加1  
        }  
        cout << "□" << n; // 输出每一步处理后的结果  
    }  
}
```

```

}
return 0; // 返回0，表示程序正常结束
}

```

这段代码就是一个完整的 C++ 程序了。同学们可以在自己的电脑上编译运行，看看效果！

非常简单，对不对？下面用三元表达式来改写一下这个程序试试看！

B.2 C++ 的进阶使用

B.2.1 更进阶的变量类型

C++ 提供了许多更进阶的变量类型和特性，可以帮助我们更好地组织代码和数据。以下是一些常见的进阶变量类型和特性：

B.2.1.1 数组（C 风格）

数组是一个可以存储多个同类型数据的变量。它的基本格式如下：

```
类型 数组名[大小];
```

以上代码的意思是：声明一个名为数组名的数组，它可以存储大小个同类型的数据。数组的索引从 0 开始。例如，我们可以声明一个整数数组来存储 10 个整数：

```
int arr[10];
```

以上的数组 `arr` 中的元素可以通过索引来访问，例如 `arr[0]` 表示第一个元素，`arr[1]` 表示第二个元素，以此类推，一直到 `arr[9]` 表示第十个元素。没有 `arr[10]`，访问这个会出错。我们不能像 Python 一样访问 `arr[-1]`，因为 C++ 不支持负索引。

全局数组（C 风格）的大小应当是常量，不能是变量；但是局部数组的大小可以是变量（需要较高 C++ 版本）。

B.2.1.2 字符串

C++ 风格的字符串类型是 `string`，它可以存储一串字符。字符串的基本格式如下：

```
#include <string> // 引入字符串库
string str = "Hello, World!";
```

引用字符串库是必要的，否则编译器可能会报错；这个库还提供了一些对字符串进行操作的方法，非常方便。

字符串的本质是一个数组，存储了一串字符（C 风格的字符串正是 `char[]`）。我们可以通过索引来访问字符串中的字符，例如 `str[0]` 表示第一个字符，`str[1]` 表示第二个字符，以此类推。

字符串的长度可以通过 `str.length()` 方法来获取。除此以外，还有很多字符串操作方法，例如 `str.substr()`（获取子串）、`str.find()`（查找子串）等。

字符串是一个复杂类，和以上提到的所有数据类型都有区别。具体为什么是“复杂类”，这涉及到 C++ 的面向对象编程（OOP）特性。我们会在后续章节中详细介绍。

B.2.1.3 结构体

结构体是一个可以存储多个不同类型数据的变量。它的基本格式如下：

```
struct 结构体名 {  
    类型 成员名1;  
    类型 成员名2;  
    // ...  
};
```

以上代码的意思是：声明一个名为结构体名的结构体，它可以存储多个不同类型的数据。结构体的成员可以是任意类型，包括基本类型、数组、字符串等。例如，我们可以声明一个表示学生的结构体：

```
struct Student {  
    string name; // 学生姓名  
    int age;     // 学生年龄  
    double gpa;  // 学生绩点  
};  
  
Student student1; // 声明一个学生变量  
student1.name = "Alice"; // 设置学生姓名  
student1.age = 20; // 设置学生年龄  
student1.gpa = 3.5; // 设置学生绩点  
cout << "Name:␣" << student1.name << ",␣Age:␣"  
      << student1.age << ",␣GPA:␣" << student1.gpa << endl;
```

以上内容很好地展示了怎么定义、声明、使用一个结构体。结构体的成员可以通过点（.）运算符来访问，例如 `student1.name` 表示学生 1 的姓名。

结构体可以帮助我们更好地组织数据，使得代码更易读。

B.2.2 联合体

联合体（union）是一个特殊的结构体，它的所有成员共享同一块内存空间。联合体的基本格式如下：

```
union 联合体名 {
```

```

    类型 成员名1;
    类型 成员名2;
    // ...
};

```

以上代码的意思是：声明一个名为联合体名的联合体，它可以存储多个不同类型的数据，但它们共享同一块内存空间。联合体的成员可以是任意类型，包括基本类型、数组、字符串等。

例如，我们可以声明一个表示数据的联合体：

```

union Data {
    int intValue;    // 整数值
    float floatValue; // 双精度浮点数值
};
Data data; // 声明一个数据变量
data.intValue = 42; // 设置整数值
cout << "Int_Value:_" << data.intValue << endl;

```

以上代码的意思是：声明一个名为 `Data` 的联合体，它可以存储整数值、双精度浮点数值和字符值。我们可以通过访问联合体的成员来获取数据。

当然，对于上述代码中使得 `data` 为 `int` 的值为 42 的情况，`data` 中的其他成员也已经随之确定：也就是把 `floatValue` 的二进制表示设定为 42 的二进制表示。但是，根据 Mini ICS 的知识我们知道，整数和浮点数的二进制表示是不同的，因此这个浮点数是一个确定的值，但它并不是 42。

B.2.3 指针

指针可以说是奠定 C 和 C++ 地位的最重要特性之一，它允许用户像汇编一样直接操作内存地址。

指针实际上也是一个变量，但是它并不是像前文所说的变量“存储数据”，而是“存储地址”。例如，我们 `int a = 10`，这个 `a` 确实是一片内存空间，但是我们没办法利用 `a` 访问这片内存空间的地址。指针可以做到这一点。比如说，`int* p = &a`，这个 `p` 就是一个指针，它存储了变量 `a` 的地址（`&a`）。我们可以通过 `*p` 来访问这个地址上的数据。

如果依然云里雾里，可以试着 `print` 一下 `p` 和 `*p` 的值。我们发现，前者输出的是一串十六进制数，而后者输出的是 10。

我们可以利用指针进行一些较为高级的控制。例如控制数组的访问、动态内存分配等。比方说：

```

int arr[5] = {1, 2, 3, 4, 5}; // 声明一个数组
int* p = arr; // 声明一个指针，指向数组的首元素
for (int i = 0; i < 5; i++) {
    cout << *(p + i) << "_"; // 通过指针访问数组元素
}

```



```
}

```

指针是一种非常强大的工具，但也需要小心使用。错误地使用指针可能会导致程序崩溃或内存泄漏，有时候也有可能会导致悬空指针（俗称“野指针”）的问题。悬空指针是指指向已经释放的内存空间的指针，这种指针无法访问有效的数据，可能会导致程序崩溃或产生不可预知的后果。

野指针不是空指针！空指针是指向空地址的指针，是安全的；野指针是指向已经释放的内存空间（现在可能已经装进去一些其他数据）的指针，是不安全的。

在 C++ 中，有一些比较高级的指针特性，例如智能指针（smart pointer）。它可以自动管理内存，避免内存泄漏和野指针等问题。常见的智能指针有 `std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`。

B.2.4 引用

引用是 C++ 中的一个重要特性，它允许我们创建一个变量的别名。引用的基本格式如下：

```
类型& 引用名 = 原变量名;
```

以上代码的意思是：声明一个名为引用名的引用，它是原变量名的别名。引用的作用是可以访问原变量。例如，我们可以声明一个整数的引用：

```
int a = 10; // 声明一个整数变量
int& ref = a; // 声明一个整数的引用
cout << "a: " << a << ", ref: " << ref << endl; // 输出a和ref的值
ref = 20; // 修改引用的值
cout << "a: " << a << ", ref: " << ref << endl; // 输出修改后的a和ref的值
```

以上代码的意思是：声明一个名为 `ref` 的引用，它是变量 `a` 的别名。我们可以通过 `ref` 来访问 `a`。当我们修改 `ref` 的值时，实际上也修改了 `a` 的值。

引用的本质其实也是一个指针，但是它的语法简洁得多。引用可以用于函数参数传递、返回值等场景，可以避免不必要的内存拷贝，提高程序性能。在 C++ 中，比起指针，我们更推荐使用安全、简洁的引用。

B.2.5 函数和变量的作用域

有时候，我们需要在这个地方使用一些代码，在另外一个地方也使用同样的代码。为了避免重复编写代码，我们可以将这些代码封装成一个函数。函数是一个可以重复调用的代码块，它可以接受参数并返回结果。

函数的基本格式如下：

```
返回类型 函数名(参数列表) {
    // 函数体
}
```

```

    return 返回值; // 如果返回类型不是void, 则需要返回一个值
}

```

以上代码的意思是：声明一个名为函数名的函数，它可以接受参数列表中的参数，并返回一个返回类型的值。函数体是函数的具体实现。

例如，我们可以声明一个计算两个整数和的函数：

```

int add(int a, int b) {
    return a + b; // 返回a和b的和
}

```

我们可以在其他函数中调用这个函数：

```

int main() {
    int x = 5;
    int y = 10;
    int sum = add(x, y); // 调用add函数
    cout << "Sum:_" << sum << endl; // 输出结果
    return 0;
}

```

函数可以有任意数量的参数，也可以没有参数。函数的返回类型可以是任意类型，包括基本类型、结构体、数组等。如果函数不需要返回值，可以将返回类型设置为 `void`。

我们发现，在上述方法 `add` 中定义的变量 `a` 和 `b` 只能在函数 `add` 内部使用，不能在其他地方使用。这是因为函数的作用域是局部的。这说明，变量具有一定的可访问范围，我们把这个可访问范围叫做“作用域”。一般来说，全局变量在任何位置都可以访问，而局部变量只能在它所在的函数或代码块中访问。

B.2.6 函数的递归调用

函数可以调用自己，这种调用方式叫做递归。递归函数通常用于解决一些具有重复结构的问题，例如计算阶乘、斐波那契数列等。递归函数的基本格式如下：

```

int foo(){
    if (base_case) {
        return base_value; // 基础情况，直接返回结果
    } else {
        return foo(); // 递归调用
    }
}

```

以上代码：在执行第一个 `foo` 的时候，会判断是不是基本情况，如果是则直接结束；如果不是，则会调用 `foo` 函数本身。这个过程会一直重复，直到满足基本情况为止。某种程度上，递归也是一种循环的形式。

需要注意的是，递归需要一个基础情况来跳出递归，否则则会产生无限递归错误。例如，我们都知道计算阶乘可以使用 $n! = n \times (n - 1)!$ ，但是只有这一个公式是不够的，不停地递归下去没有尽头。这时候，我们需要一个基础情况来结束递归： $0! = 1$ 。因此，我们可以写出递归公式： $factorial(n) = n \times factorial(n - 1)$ ，其中 $factorial(0) = 1$ 。然后，我们就可以用程序语言来描述这个数学语言：

```
int factorial(int n) {
    if (n == 0) {
        return 1; // 基础情况
    } else {
        return n * factorial(n - 1); // 递归调用
    }
}
```

B.2.7 函数的传参

刚刚说到，函数可以接受一些参数。一般情况下，有三个传参方式：值传递、引用传递和指针传递。

- 值传递：函数接收参数的副本，修改参数不会影响原变量。基本类型（如 `int`、`double` 等）默认使用值传递。
- 引用传递：函数接收参数的引用，修改参数会影响原变量。可以通过在参数类型前加 `&` 来实现。
- 指针传递：函数接收参数的指针，修改参数会影响原变量。可以通过在参数类型前加 `*` 来实现。

例如，我们可以使用引用传递来交换两个整数的值：

```
void swap(int& a, int& b) {
    int temp = a; // 使用临时变量交换
    a = b;
    b = temp;
}
```

使用传指针其实也可以实现同样的功能。但是，传值不能实现同样的功能：传值的本质是复制参数的值到函数内部，因此在函数内部修改参数不会影响原变量。至于引用和指针，则对应的可以理解为剪切，因此能够直接修改原变量的值。

B.2.8 小练

素数在数学中是一个非常重要的概念，它指的是只能被 1 和它本身整除的自然数。素数在密码学、数据加密等领域有着广泛的应用。一般我们可以使用筛法找到素数：在一系列整数中，先找到最小的素数（2），然后将它的倍数都去掉；然后再找到下一个最小的素数（3），再将它的倍数都去掉；如此反复，直到所有的数都被处理完。

请编写一个 C++ 程序，输入一些整数 n_1, n_2, \dots ，然后输出这些整数是不是素数。 n 的数值在 0 到 1000 之间。

B.2.8.1 边写边说

同样的，写程序就像说话一样。我们阅读题目：在一系列整数中，这个“一系列”整数提示我们可以使用数组来存储这些整数；先找到最小的素数，然后将它的倍数都去掉；然后再找到下一个最小的素数（3），再将它的倍数都去掉；如此反复，直到所有的数都被处理完，这句话提示我们需要使用循环来处理这些整数。

数组的索引天然就是自然数集合，因此我们可以使用索引表示整数。我们可以声明一个比较大的数组，来存储从 1 到 n 的整数。假设 n 不超过 1000，我们可以声明一个大小为 1000 的数组来存储这些整数，并将它们全部初始化为 0（表示未处理）。对于数组中的每一个元素，我们都可以将他的索引作为整数的值，而元素的值为 0 说明是素数，1 说明不是素数。

然后我们可以使用一个循环来遍历这个数组，找到素数。写成代码就是：

```
bool arr[1000] = {}; // 声明一个数组来存储0和1，并将全部数值初始化为0
arr[0] = 1; // 0不是素数
arr[1] = 1; // 1不是素数
for (int i = 2; i < 1000; i++){
    if (arr[i] == 0) { // 如果这个数是素数
        for (int j = 2; j < 1000; j++) { // 将它的倍数都标记为1
            arr[j * i] = 1;
        }
    }
}
```

接下来，我们需要输出素数。我们可以直接查询 n_1, n_2, \dots 是否在数组中对应的索引处的值为 0，如果是，则说明这个数是素数。写成代码就是：

```
if (arr[i] == 0) { // 如果这个数是素数
    cout << i << "是素数" << endl;
}
else {
    cout << i << "不是素数" << endl;
}
```

最后，我们需要将这些代码组合起来，形成一个完整的 C++ 程序。我们可以将输入 *n* 的部分放在 `main` 函数中，然后调用上面的代码来处理素数。写成代码就是：

```
#include <iostream>
using namespace std;

bool arr[1000] = {}; // 声明一个数组来存储0和1，并将全部数值初始化为0

int findPrimes() {
    arr[0] = 1; // 0不是素数
    arr[1] = 1; // 1不是素数
    for (int i = 2; i < 1000; i++) {
        if (arr[i] == 0) { // 如果这个数是素数
            for (int j = 2; j * i < n; j++) { // 将它的倍数都标记为1
                arr[j * i] = 1;
            }
        }
    }
    return 0;
}

int main() {
    int n;
    cin >> n; // 输入一个整数n
    findPrimes(1000); // 调用函数来筛出素数
    while(cin >> i){
        if (arr[i] == 0) { // 如果这个数是素数
            cout << i << "是素数" << endl;
        }
        else {
            cout << i << "不是素数" << endl;
        }
    }
    return 0; // 返回0，表示程序正常结束
}
```

上述代码中的 `while(cin >> i)` 可以在无法确定输入的数量情况下，帮我们自动处理输入。

思考：不使用筛法的情况下，还有没有其他的算法？

B.3 C++ 的高级特性

C++ 提供了许多高级特性，可以帮助我们更好地组织代码和数据。

B.3.1 命名空间

命名空间（namespace）是 C++ 中的一个重要特性，它允许我们将代码组织在不同的命名空间中，以避免名称冲突。命名空间的基本格式如下：

```
namespace 命名空间名 {  
    // 代码  
}
```

以上代码的意思是：声明一个名为命名空间名的命名空间，命名空间中的代码可以在这个命名空间内访问。我们可以在不同的命名空间中定义同名的变量、函数等，而不会发生冲突。

B.3.2 面向对象编程

面向对象编程是 C++ 的最重要特性之一。它允许我们将数据和操作数据的函数封装在一起，形成一个对象。对象是一个包含数据和方法的实体，它可以表示现实世界中的事物。同时，面向对象编程还提供了继承、多态等特性，可以帮助我们更好地组织代码和数据。

B.3.2.1 类和属性

类是面向对象编程的基本操作单位。如果不熟悉类，可以把类当成“超级 struct”来理解，这里面除了存储数据（C++ 叫“属性”）以外，还可以顺便把函数（C++ 叫“方法”）也打包进去。

```
class Point2D{  
public:  
    static const int DIMENSION = 2; // 类的常量属性  
    static int count; // 类的静态属性  
    int x, y;  
    void move(int dx, int dy) {  
        x += dx;  
        y += dy;  
    }  
    Point2D(int x = 0, int y = 0) : x(x), y(y) { count++; } // 构造函数  
    ~Point2D() { count--; } // 析构函数  
}
```

于是，这下变量和函数成了一家人：


```
Point2D p(1, 2); // 创建一个Point2D对象p, x=1, y=2
p.move(5, -3); // 移动点p, 它自己知道怎么动!
cout << Point2D::count << endl; // 输出类的静态属性
```

这就是“把数据和对数据的操作绑在一起”——面向对象的核心思想。

在类中，你可以看到我打了一个 `public`，这说明以下属性和方法是公开的，其他所有类或者类外的东西都可以访问它。如果你不打 `public`，那么默认是私有的（`private`），只有这个类内部可以访问；另一种访问权限是 `protected`，它允许子类访问，但不允许类外的东西访问。（至于什么是子类，请先收起疑问，往下看就懂了）

部分属性前面，你可以发现打了 `static` 符号。这说明这个属性是静态的，属于类本身，而不是类的实例（实例指的就是可操作的对象，例如上面的 `p`）。静态属性可以通过类名直接访问，例如 `Point2D::count`。静态属性在所有实例之间共享，因此它们的值是全局的。

B.3.2.2 自指

类可以包含指向自身的指针或引用，这种特性称为自指，用 `this` 可以访问当前对象的指针。自指允许我们在类中定义链表、树等数据结构。自指的基本格式如下：

```
class Node {
public:
    int data; // 节点数据
    Node* next; // 指向下一个节点的指针
    Node(int value) : data(value), next(nullptr) {} // 构造函数
};
```

B.3.2.3 构造、析构、拷贝和赋值

类的构造函数和析构函数是特殊的方法，用于对象的初始化和清理。构造函数在创建对象（也叫实例化）时自动调用，而析构函数在对象销毁时自动调用。构造函数的名称与类名相同，并且没有返回值；析构函数的名称是类名加上波浪号（`~`），也没有返回值。

构造函数可以有参数，用于初始化对象的属性；析构函数通常用于释放对象占用的资源。这是 C++ 的一个重要特性：**RAII (Resource Acquisition Is Initialization)**，资源获取在初始化中获取、在析构中释放。我们在 C++ 中不需要像 C 一样手动 `malloc` 和 `free` 内存，而是通过构造函数和析构函数来自动管理资源，代码更简洁也更安全。

一般情况下，类有着默认的构造和析构函数，它们不含有任何参数，且不执行任何操作。默认的构造函数会将所有属性初始化为默认值（例如整数为 0，布尔值为 `false` 等），析构函数则不会做任何事情。满足这种条件的类也叫做 **POD 类型, Plain Old Data**：没有自定义构造函数、析构函数和拷贝构造函数，它们的行为类似于 C 语言中的结构体。

如果我们写了自己的构造和析构函数，那么默认的就会被覆盖掉。比方说，上文 `Point2D` 类中，我们定义了一个带参数的构造函数和一个析构函数。这样，当我们创建一个 `Point2D` 对象时，就会调用这个构造函数来初始化对象的属性（将全局点数量增加 1）；当对象被销毁时，就会调用析构函数来干点别的（将全局点数量减 1），然后清理资源。

需要注意的一点是：由于在 C++ 中对象的资源管理由构造函数和析构函数自动完成，因此我们不要在构造函数中 `malloc`，也不要再在析构函数中 `free` 或者 `delete this`。前者有可能导致内存泄漏，后者可能会导致双重释放，这是非常危险的。

拷贝函数是一个特殊的构造函数，它用来复制对象。一般情况下，C++ 会自动生成一个拷贝构造函数，它会逐个复制对象的属性。但是，如果类中有指针或动态分配的资源，我们需要自定义拷贝构造函数来正确地复制对象。拷贝构造函数的参数是类本身的常量引用，而对方法本身没有什么要求。

一般拷贝分为浅拷贝和深拷贝。浅拷贝只是复制指针的值，而深拷贝则会复制指针指向的内容。对于包含指针的类，我们通常需要实现深拷贝，以避免多个对象的指针指向同一块内存空间，导致资源管理混乱。默认情况下，C++ 的拷贝形式是浅拷贝。

拷贝赋值运算符是一个特殊的运算符，用于将一个对象的值赋给另一个对象。它的基本格式如下：

```
类名& operator=(const 类名& other) {
    if (this != &other) { // 防止自赋值
        // 复制属性
    }
    return *this; // 返回当前对象的引用
}
```

由此，我们看到了我们对 `=` 进行了重载。这实际上是定义了一个赋值函数。

B.3.2.4 封装

封装是面向对象编程的一个重要特性，它允许我们将数据和方法封装在一起，形成一个对象。封装的目的是隐藏实现细节，只暴露必要的接口给外部使用。这样可以提高代码的可维护性和可重用性。

比方说：

```
class BankAccount {
private:
    int balance; // 私有属性，外部无法直接访问
public:
    BankAccount(int initialBalance) : balance(initialBalance) {} // 构造函数
    void deposit(int amount) { // 公共方法，允许外部调用
        if (amount > 0) {
```

```

        balance += amount; // 增加余额
    }
}
void withdraw(int amount) { // 公共方法，允许外部调用
    if (amount > 0 && amount <= balance) {
        balance -= amount; // 减少余额
    }
}
int getBalance() const { // 公共方法，允许外部查询余额
    return balance; // 返回余额
}
};

```

这样可以阻止外部直接修改余额，只能通过存款和取款方法来操作余额。问我为什么余额用 `int` 而不是 `double` 或者 `float` 的，建议重读 Mini ICS。

在 C# 中，对于封装有一个非常优雅的名词：*Getter* 和 *Setter*。*Getter* 是获取属性值的方法，*Setter* 是设置属性值的方法，同样是上述的代码我们在 C# 中可以写成 `public int Balance { get; private set; }`，意思是只有类内可以设置这个属性的值，而类内外可以获取这个属性的值。这样就实现了封装，同时又不失优雅。C++ 中没有这个优雅的语法，因此我们只能像上述代码中手动定义 *getter*。

B.3.2.5 继承

继承是面向对象编程的一个重要特性，它允许我们创建一个新的类（子类），它继承了另一个类（父类）的属性和方法。子类可以添加自己的属性和方法，也可以重写父类的方法。被重写的方法必须是**虚方法**，也就是在父类中声明为 `virtual` 的方法。

继承的基本格式如下：

```

class Shape { public: virtual double area() = 0; };
class Circle : public Shape { ... };

```

以上代码的意思是：声明一个名为 `Shape` 的类，它有一个纯虚函数 `area()`，表示这个类是一个抽象类。然后声明一个名为 `Circle` 的类，它继承了 `Shape` 类，并实现了 `area()` 方法。

在实际操作中，子类一般属于父类的一个特例，或者说**子类真包含于父类**。例如，我们要创建一个“大舅”类和一个“二舅”类，一个非常差的设计是让“二舅”继承自“大舅”，因为二舅并不是大舅的一个特例（或者说二舅不是大舅），反过来也一样。一个好的设计是让这两个类都继承自一个“舅舅”类（他大舅他二舅都是他舅），这样就可以避免这种问题。

B.3.2.6 多态

多态指的是同一个方法在不同的对象上有不同的表现。多态是通过继承和虚函数实现的。当我们调用一个虚函数时，实际调用的是子类中重写的方法，而不是父类中的方法。这种特性使得我们可以使用父类指针或引用来调用子类的方法。

以继承中涉及到的 `Shape` 和 `Circle` 类为例：

```
Shape* shape = new Circle(); // 创建一个Circle对象，并将其赋值给Shape指针
shape->area(); // 调用Circle类的area()方法
```

上述代码中会自动调用 `Circle` 类的 `area()` 方法，而不是 `Shape` 类的 `area()` 方法。这就是多态的体现：不用去关心具体的对象类型，省去了 `switch` 语句的麻烦。

B.3.3 重载

重载是 C++ 中的一个重要特性，它允许我们定义多个同名的函数或运算符，但它们的参数列表或返回类型不同。写一个例子就好了：

```
class Tensor{
    int x, y;
public:
    Tensor(int x, int y) : x(x), y(y) {}
    Tensor operator+(const Tensor& other) { // 重载加法运算符
        return Tensor(x + other.x, y + other.y);
    }
}

Tensor t1(1, 2);
Tensor t2(3, 4);
Tensor t3 = t1 + t2; // 调用重载的加法运算符
cout << t3.x << ", " << t3.y << endl; // 输出结果
```

以上代码就是重载的一个鲜活实例。我们重载了加法运算符，这使得我们能够对 `Tensor` 类的对象进行加法运算。合适的重载可以使代码更简洁、更易读。

除了重载运算符，还可以重载流运算符来实现自定义输入输出，重载函数实现对不同参数的处理等。重载的关键是参数列表的不同，返回类型可以相同或不同。

B.3.4 模板

模板也是一个很重要的特性，它允许我们编写通用的代码，可以处理不同类型的数据。模板可以分为函数模板和类模板。

比方说我们想写一个加法：

```
template <typename T>
T add(T a, T b) {
    return a + b; // 返回a和b的和
}

int main() {
    int x = 5, y = 10;
    cout << add(x, y) << endl; // 调用add函数，输出15
    double a = 3.14, b = 2.71;
    cout << add(a, b) << endl; // 调用add函数，输出5.85
    return 0;
}
```

这个函数就可以对任何类型的数据进行加法操作，只要这个类型支持加法运算符。对于不支持加法运算符的类型，编译器会报错（但是我们可以为这些类型重载加法运算符）。

类模板的语法类似，只不过是定义一个类而不是一个函数：

```
template <typename T>
class Box {
public:
    T value; // 存储一个值
    Box(T v) : value(v) {} // 构造函数
    T getValue() const { return value; } // 获取值的方法
};

Box<int> intBox(42); // 创建一个存储整数的Box对象
Box<double> doubleBox(3.14); // 创建一个存储双精度浮点数的Box对象
```

使用模板可以显著降低代码量，提高代码的可重用性。

B.3.5 类型推断

类型推断是 C++11 引入的一个特性，它允许编译器根据变量的初始值自动推断变量的类型。使用类型推断可以使代码更简洁、更易读。类型推断的基本语法是使用 `auto` 关键字：

```
auto x = 5; // 编译器推断x的类型为int
auto y = 3.14; // 编译器推断y的类型为double
auto str = "Hello, World!"; // 编译器推断str的类型为const char*
```

以上代码中，编译器会根据初始值自动推断变量的类型。但是我们并不推荐在定义变量的时候使用类型推断，因为这会使得代码的可读性降低，尤其是当变量的类型不明显时。

那么我为什么要讲这个？请看下一节，这一节将会是使得整个 C++ 成功获得“没有人能够真正精通”称号的关键特性。

B.3.6 类型别名

类型别名是 C 就有的一个特性，但是 C++11 对它进行了扩展。类型别名允许我们为现有的类型创建一个新的名称，使得代码更易读。

C++ 中可以使用 `using` 关键字来定义类型别名。

```
using ll = long long; // 定义一个长整型的别名
using IntVector = std::vector<int>; // 定义一个整型向量的别名
IntVector v = {1, 2, 3}; // 使用别名创建一个整型向量
```

如果使用 C 风格的语法，则是：

```
typedef long long ll; // 定义一个长整型的别名
typedef std::vector<int> IntVector; // 定义一个整型向量的别名
IntVector v = {1, 2, 3}; // 使用别名创建一个整型向量
```

`using` 和 `typedef` 几乎没有什么区别，只不过 `using` 的语法更加符合直觉（用这个作为这个的别名），类似于声明变量；而 `typedef` 则更像是定义一个宏（虽然实际上不是），阅读方向是反直觉的。

`using` 的另一个独特之处是可以用于模板类型的别名：

```
template <typename T>
using Matrix = std::vector<std::vector<T>>>; // 定义一个二维向量的别名
Matrix<int> m = {{1, 2}, {3, 4}}; // 使用别名创建一个二维向量
Matrix<double> dm = {{1.1, 2.2}, {3.3, 4.4}}; // 使用别名创建一个二维向量
```

`typedef` 就无法应用于模板类型别名。因此，在 C++ 中，我们推荐使用 `using` 来定义类型别名。

B.3.7 类型强转

有时候，在编程中我们需要将一个类型转化成另一个类型，以满足特定的需求。

C 语言就有类型强转的功能：

```
float f = 3.14;
int i = (int)f; // 将float类型强制转换为int
// 也可以写成 int i = int(f);

printf("i: %d\n", i); // 输出结果，结果是3
```

在变量前面加上括号和目标类型，就可以将变量强制转换为目标类型。

在 C++ 中，类型强转被拆成了四个方式（四大金刚）：

- **static_cast**: 编译期安全的强转，包括数值提升/截断，枚举/整型，指针向上转型、void* 转型等。它是最常用的类型转换方式，适用于大多数情况。

- **dynamic_cast**: 运行时安全的强转, 用于类的向下转型 (子类指针转父类指针)。它会在运行时检查类型安全, 如果转换不安全, 则返回 `nullptr`。它只能用于有虚函数的类。同时, 它是唯一一个在运行时检查强转安全性的转换方式。
- **const_cast**: 常变转换, 其他啥都不干。它是唯一一个能去 `const` 的转换方式。
- **reinterpret_cast**: 按位重解释, 用于 `int` 指针互转、`void` 指针互转、无关类指针互转等。它是危险的转换方式, 仅在编译期做极弱的检查。除非我们知道在干什么, 否则不要使用它。

举例说明:

```
#include <iostream>
using namespace std;

double d = 3.14;
int a = static_cast<int>(d); // 使用static_cast进行数值转换
int a = (int)d; // C风格的强转, 也行

const int c = 42;
int* p = const_cast<int*>(&c); // 使用const_cast去掉const属性
*p = 100; // 修改p指向的值, 即使C是const的也能改

class Base;
class Derived : public Base;
Base* b = new Derived();
Derived* d = dynamic_cast<Derived*>(b); // 使用dynamic_cast进行向下转型
Derived* d = static_cast<Derived*>(b); // 使用static_cast进行向下转型 (不安全, 但是能过编译)

uintptr_t ptr = reinterpret_cast<uintptr_t>(b); // 使用reinterpret_cast将指针转换为整数
```

那么有些同学可能会问: 为什么 C++ 要提供这么多种类型强转? 难道 C 风格的强转不行吗? 没错, 两种代码实际上都可以用。不过, C 风格的强转像个大锤, 一口气把任何东西都能砸成目标东西, 但是它可不带管安全性的; 而 C++ 强转四大金刚分别是四把精确的手术刀, 功能单一、语义明确, 编译器会帮助你把关, 要是危险或者出错了, 编译器给你兜底。这样就可以避免很多潜在的错误。

C 语言的强转实际上会先尝试常变转换, 再尝试数值转换, 要是不行就常变数值一起转, 还不行就按位重解释。所以说这玩意实际上是四合一, 不过也导致它隐形语义极为复杂、易于出错, 出错了也不容易搜索定位。

```
const volatile void* v = ...;
```

```
int* bad = (int*)v; // C风格的强转，实际上一口气把const和volatile都去掉了，顺便做了个按位重解释
```

所以说，我们非常建议优先使用 C++ 四大金刚做显式强转。我们非常不建议在 C++ 中使用旧式风格的强转，除非要做向下兼容等不这么做不行的事情。

volatile 是 C/C++ 中的一个关键字，表示变量可能会被外部因素改变，因此编译器不会对它进行优化。它通常用于多线程编程或硬件寄存器的访问等。这个东西和移位运算符一样绝大多数人一辈子都不会用到。

B.4 STL

STL (Standard Template Library) 是 C++ 的最重要特性，它提供了一组通用的模板类和函数，可以帮助我们更高效地处理数据结构和算法。STL 包含了许多常用的数据结构和算法，例如向量 (vector)、链表 (list)、集合 (set)、映射 (map) 等。

简单地说，STL 可以看作是：容器 + 迭代器 + 算法。容器把数据结构当变量类型用，迭代器把指针当普通函数用，算法把现成高复杂的轮子当函数用，这玩意能让你用三行代码完成 C 里三十行甚至三百行的工作，还自带内存管理和类型安全。

于是，C++ 开发就变成了：打开编辑器，敲下头文件，剩下的一律交给 STL。

B.4.1 容器

举个最常见的例子：

```
std::vector<int> v = {3,1,4}; // 自动扩容的数组
std::set<int> s = {3,1,4}; // 自动排序的红黑树
std::unordered_map<std::string,int> m; // 哈希表
```

以上代码中，我们使用了 STL 提供的向量 (vector)、集合 (set) 和映射 (unordered_map) 容器。它们都是模板类，可以存储任意类型的数据。使用它们非常容易：头文件即声明、自动管理内存、接口几乎全 STL 统一。

常见的容器有以下几种：（如果我没记错的话，C++ 正课会要求全部掌握这些容器，我只能说：祝你好运！）

- **vector**: 动态数组（向量），可以自动扩容，支持随机访问。
- **list**: 双向链表，支持高效的插入和删除操作，但不支持随机访问。
- **deque**: 双端队列，支持在两端高效地插入和删除操作。
- **set**: 集合，存储唯一元素，并自动排序。
- **map**: 映射，存储键值对，并根据键自动排序。
- **unordered_set**: 无序集合，存储唯一元素，不自动排序，查询效率高。
- **unordered_map**: 无序映射，存储键值对，不自动排序，查询效率高。

- `stack`: 栈, 后进先出 (LIFO)。
- `queue`: 队列, 先进先出 (FIFO)。
- `priority_queue`: 优先队列, 支持按优先级访问元素。
- `array`: 固定大小的数组, 类似于 C 风格的数组, 但提供了更多的功能。
- `bitset`: 位集合, 支持高效的位操作。
- `tuple`: 元组, 可以存储不同类型的多个值。
- `forward_list`: 单向链表, 类似于 `list`, 但只支持单向遍历。
- `unordered_multiset`: 无序多重集合, 存储可以重复的元素, 不自动排序。
- `unordered_multimap`: 无序多重映射, 存储可以重复的键值对, 不自动排序。

其实遇事不决的情况下, 我们可以按照需求选择容器:

- 速查: 如果需要快速查找元素 (建哈希表), 使用 `unordered_set` 或 `unordered_map`。
- 排序: 如果需要自动排序, `priority_queue` 是最好的选择, 其次是 `set` 和 `map`。
- 频繁插入删除: 如果需要频繁插入和删除元素, 使用 `list`。
- 只关心两头: 如果只关心两端的插入和删除, 使用 `deque`。如果能确定用的是栈或队列, 使用 `stack` 或 `queue`。
- 遇事不决: 如果不确定用什么容器, 使用 `vector`。

特别说明: 虽然我把 `stack` 和 `queue` 也当成容器、实际上在工程上也不怎么区分这东西, 但是这里我有必要提及: 这两个玩意实际上是容器适配器 (*container adapter*), 它们是基于其他容器实现的, 提供了栈和队列的接口。一般情况下, 默认参数是 `vector` 或者 `deque` (因此不必指明), 但是你也可以指定其他容器作为底层容器。

B.4.2 迭代器

迭代器可以认为是指针的语法糖。一个示例:

```
for(auto it=v.begin(); it!=v.end(); ++it) cout<<*it<<' ';
// 或者直接:
for(auto x : v) cout<<x<<' '; // auto最应该这么用!
```

所有容器风格完全一致, 完全不必关心装的是什麼玩意。一些常见的迭代器和方法:

- `begin()`: 返回容器的起始迭代器。
- `end()`: 返回容器的结束迭代器。
- `rbegin()`: 返回容器的反向起始迭代器。
- `rend()`: 返回容器的反向结束迭代器。
- `cbegin()`: 返回容器的常量起始迭代器。
- `cend()`: 返回容器的常量结束迭代器。
- `next(it)`: 返回迭代器 `it` 的下一个位置。
- `prev(it)`: 返回迭代器 `it` 的上一个位置。
- `distance(it1, it2)`: 返回迭代器 `it1` 和 `it2` 之间的距离。

迭代器也可以加减，例如 `it+1` 表示下一个元素，`it-1` 表示上一个元素。

B.4.3 算法

STL 提供了许多常用的算法，可以帮助我们更高效地处理数据，直接拿出来用就行：

```
std::sort(v.begin(), v.end());    // 快排
std::binary_search(v.begin(), v.end(), 4); // 二分
std::reverse(v.begin(), v.end()); // 原地翻转
```

以上代码中，我们使用了 STL 提供的排序（`sort`）、二分查找（`binary_search`）和翻转（`reverse`）算法。STL 的算法通常是模板函数，可以处理任意类型的数据。

除此之外，还有一些常用的算法：

- `std::find`：查找元素。
- `std::count`：统计元素出现的次数。
- `std::accumulate`：计算元素的累加和。
- `std::max_element`：找到最大元素。
- `std::min_element`：找到最小元素。
- `std::shuffle`：随机打乱元素顺序。
- `std::unique`：去除重复元素。
- `std::merge`：合并两个已排序的范围。
- `std::partition`：对元素进行分区。
- `std::transform`：对元素进行转换。
- `std::for_each`：对每个元素执行操作。
- `std::set_union`：计算两个集合的并集。
- `std::set_intersection`：计算两个集合的交集。
- `std::set_difference`：计算两个集合的差集。
- `std::set_symmetric_difference`：计算两个集合的对称差集。
- `std::nth_element`：找到第 `n` 小的元素。
- `std::lower_bound`：找到第一个不小于给定值的元素。
- `std::upper_bound`：找到第一个大于给定值的元素。

利用头文件 `<algorithm>` 可以使用这些算法。STL 的算法通常是模板函数，可以处理任意类型的数据；配合迭代器，算法和容器原地解耦。

以上，就是 C++ 的全部内容了（也不是全部内容，毕竟 C++14、C++17 等版本有越来越多的新特性，但是能掌握 C++11 的全部特性就已经不得了了）。C++ 的语法和特性非常丰富，学习曲线较陡，但一旦掌握，就可以编写高效、可维护的代码。

附录 C Python 高速入门

本章会快速带领大家过一遍 Python 的基本语法和常用特性。除了用作预习材料以外，还可以在期末考试复习的时候来快速回顾其基本语法与常用特性。

在 PKU，Python 主要是文科生学习较多，因此我这一章的节奏会比 C++ 的慢许多，也不会像 C++ 一样涉及那么多的名词（内存空间、指针、引用等）。

C.1 Python 的基本语法

我在 C++ 的章节中提过，写代码的本质是和计算机说话。如果说 C++ 更像正式信件，有信头、正文、落款，那么 Python 更像是口语化的对话。

比方说，我们写一个最基本的程序：

```
str = "Hello, world!"  
print(str)
```

执行以上代码，我们会看到输出：“Hello, world!”。

逐行拆解代码，第一行的意思是，我告诉计算机“我有一个变量叫做 str，它的值是 Hello, world!”；第二行，则告诉计算机“请把变量 str 的值打印出来”。

看起来非常简单。

C.1.1 Python 的变量

Python 的变量非常简单，并不需要遵从“先声明再使用”的规则，而是直接就可以拿出来用。同时，Python 的语法也非常宽松，对于变量并不需要指定其类型，一个变量可以是任何类型的值。

```
a = 10 # 整数  
a = 3.14 # 浮点数  
a = 1 + 2j # 复数  
a = "Hello, world!" # 字符串  
a = [1, 2, 3] # 列表  
a = (1, 2, 3) # 元组  
a = {1, 2, 3} # 集合  
a = {"name": "Alice", "age": 30} # 字典  
a = True # 布尔值  
a = None # 空值
```

这么直接拿出来就可以用。这里的等号 `=` 不是数学上的等号，它的意思是“把右边的值赋给左边的变量”。而且，Python 并不需要担心像 C++ 一样的溢出问题，Python 会自动处理大数。

一切都比 C++ 简单得多。

C.1.2 Python 的运算

有时候，我们需要计算机帮助我们执行一些运算。例如：

```
a = 10
b = 3
print(a + b) # 输出13
print(a - b) # 输出7
a += b # 相当于a = a + b
```

`a+b` 的意思就是“把 `a` 和 `b` 相加”，而 `a+=b` 的意思是“把 `b` 加到 `a` 上”。

Python 支持许多常见的运算符：

- 四则运算：加 `+`、减 `-`、乘 `*`、除 `/`（浮点除法）和取整除 `//`（整数除法）。
- 乘方：使用 `**` 表示乘方运算。
- 取余数：使用 `%` 表示取余数运算。

对于字符串类型的变量，使用加法 `+` 可以连接两个字符串，例如：

```
str1 = "Hello, "
str2 = "world!"
print(str1 + str2) # 输出"Hello, world!"
```

C.1.3 输入、输出

Python 的输入输出非常简单。我们可以使用 `print()` 函数来输出内容，而使用 `input()` 函数来获取用户输入。`input` 函数会暂停程序的执行，等待用户输入内容，并将输入的内容作为字符串返回。

`input` 里面的内容是提示用户输入的文本。

例如：

```
name = input("请输入你的名字：")
print("Hello, " + name + "！")
```

我们还可以使用一些格式化字符串的方式来输出内容，例如：

```
name = "Alice"
age = 30
print(f"Hello, {name}！You are {age} years old.")
```


这会输出 “Hello, Alice! You are 30 years old.”。这个 f+ 字符串的语法表示这是一个格式化字符串，可以直接在字符串中使用变量。

我们还可以使用一些参数来进一步格式化输出内容，例如：

```
print("Hello, World!", end="") # 不换行输出
print("Hello, 1", "Hello, 2", sep=", ") # 使用逗号分隔输出
print("Hello, World!", file=open("output.txt", "w")) # 输出到文件
```

上述代码中的第二个 print 函数使用了 sep 参数来指定输出内容之间的分隔符，默认是空格。它的输出将会是：“Hello, 1, Hello, 2”。

上述输出到文件的例子会将 “Hello, World!” 写入同目录下的 output.txt 文件中，这个 w 的意思是“写入模式”，如果文件不存在则会创建，如果存在则会覆盖原有内容。如果改成 a，则会以追加模式打开文件，即在文件末尾添加内容。

C.1.4 注释

注释是代码中用于解释说明的部分，它会被解释器忽略，不会影响程序的运行。Python 中的单行注释使用井号 # 开头，这一行后面的内容都是注释；或者使用三引号来框住注释内容，可以创建多行注释。

```
# 这是一个注释
print("Hello, world!") # 这也是一个注释

"""
这是一个多行注释
可以包含多行内容
"""
```

在阻止部分代码执行的时候，我们一般不习惯于直接删除这些代码，而是使用注释。这样做的好处是可以留痕，便于以后的恢复（解注释）；这就是程序员们常说的“注释掉”代码。在 VS Code 等编辑器中，常用的一键注释是 Ctrl + /，它会自动将光标所在的一行或多行代码注释掉。

C.1.5 类型强转

虽然 Python 是动态类型语言，同一个变量可以在不同的时间点上拥有不同的类型，但是在某一个确定的时刻，一个变量的类型是确定的。例如我们给 a 赋值 a = "12321"，那么这个时候 a 的类型就是字符串。如果对该变量进行和数的加减操作，代码将会无法执行。

有些时候，我们希望把一些变量的类型转换为其他类型，例如把字符串 “12321” 转换为整数 12321。我们可以使用一些函数来实现类型转换：

```
a = "12321"
print(a+1) # 报错，因为a是字符串，1是整数，不能直接相加
b = int(a) # 将字符串转换为整数，现在b是整数12321
print(b+1) # 能执行，输出12322
```

我们可以使用 `int()` 函数将字符串转换为整数，使用 `float()` 函数将字符串转换为浮点数，使用 `str()` 函数将其他类型转换为字符串等。

C.2 控制程序的执行流程

C.2.1 条件语句

有时候，我们需要让计算机根据条件来执行不同的操作。Python 提供了 `if` 语句来实现这一点。

例如，我们可以根据用户输入的年龄来判断是否成年：

```
age = int(input("请输入你的年龄："))
if age >= 18:
    print("你是成年人。")
else:
    print("你是未成年人。")
```

在这个例子中，`if` 语句后面跟着一个条件表达式（`age >= 18`），如果条件为真，则执行冒号后面的代码块；否则，执行 `else` 后面的代码块。

Python 使用**缩进**来表示代码块的层次结构，且对缩进要求极为严格。通常情况下，我们用一个制表符或者四个空格来表示一个缩进层级。要打出制表符，可以按下 `Tab` 键。

C.2.2 循环语句

有时候，我们需要让计算机重复执行某些操作。Python 提供了 `for` 和 `while` 两种循环语句。

比方说我们使用 `for` 循环来输出 1 到 10 的数字：

```
for i in range(1, 11):
    print(i)
```

在这个例子中，`range(1, 11)` 生成了一个从 1 到 10 的整数序列，`for` 循环会依次将每个数字赋值给变量 `i`，并执行代码块中的操作。

我们也可以使用 `while` 循环来实现类似的功能：

```
i = 1
```

```
while i <= 10:
    print(i)
    i += 1
```

在这个例子中，`while` 循环内的代码块会一直循环执行，直到条件 `i <= 10` 不再满足为止。

可以看到，我们在这个循环内部对 `i` 进行了增加操作。如果没有这个操作，循环将会无限进行下去，技术上一般叫做“死循环”。表现在程序上，上述程序会不断地输出 1，直到你强制终止程序。而一般 `for` 循环则不会出现这种情况，因为它会自动处理循环变量和循环条件。

有时候，我们在使用 `for` 循环的时候并不关心循环变量的值，只是想要重复执行某些操作。这时，我们可以使用下划线作为循环变量的占位符：

```
for _ in range(5):
    print("Hello, World!")
```

在这个例子中，循环会执行 5 次，但我们并不关心循环变量的值，只是简单地输出“Hello, World!”，于是使用下划线将其“丢弃”。

`break` 和 `continue` 语句可以用来控制循环的执行流程。使用 `break` 可以提前退出循环，而使用 `continue` 可以跳过当前迭代，继续下一次循环。例如：

```
for i in range(1, 11):
    if i == 5:
        break # 当i等于5时，退出循环
    print(i)
for i in range(1, 11):
    if i == 5:
        continue # 当i等于5时，跳过当前迭代
    print(i)
```

上述两个循环中，第一个循环会输出 1 到 4，然后退出循环；第二个循环会输出 1 到 4、6 到 10，但跳过 5。

C.3 复合数据类型

Python 提供了多种复合数据类型，用于存储多个值。最常用的有列表（list）、元组（tuple）、集合（set）和字典（dict）。

C.3.1 列表（list）

列表是一个有序的可变集合，可以存储任意类型的元素。我们可以使用方括号 `[]` 来创建一个列表，并使用索引来访问元素。索引从 0 开始，如果我们试图访问一个不存在的索引，会抛出 `IndexError` 异常。例如：

```
my_list = [1, 2, 3, "Hello", True]
print(my_list[0]) # 输出1
print(my_list[3]) # 输出"Hello"
print(my_list[-1]) # 输出True（负索引从后往前计数）
print(my_list[5]) # 抛出IndexError异常
```

我们可以使用 `append()` 方法添加元素，使用 `remove()` 方法删除元素，使用 `sort()` 方法对列表进行排序等。具体什么是“方法”，详见C.4节。例如：

```
my_list = [1, 2, 3, "Hello"]
my_list.append(4) # 添加元素4
my_list.remove("Hello") # 删除元素"Hello"
my_list.sort() # 对列表进行排序
print(my_list) # 输出[1, 2, 3, 4]
```

C.3.2 元组 (tuple)

元组和列表类似，但是它不可以被修改（不可变）。我们可以使用圆括号 `()` 来创建一个元组。元组的元素也可以通过索引访问。例如：

```
my_tuple = (1, 2, 3, "Hello", True)
print(my_tuple[0]) # 输出1
print(my_tuple[3]) # 输出"Hello"
print(my_tuple[-1]) # 输出True
print(my_tuple[5]) # 抛出IndexError异常
```

元组的元素不能被修改，但我们可以通过重新赋值来创建一个新的元组。例如：

```
my_tuple = (1, 2, 3)
my_tuple_1 = my_tuple + (4,) # 创建一个新的元组
print(my_tuple_1) # 输出(1, 2, 3, 4)
```

C.3.3 集合 (set)

集合是一个无序的可变集合，不能包含重复元素。我们可以使用花括号来创建一个集合。集合的元素也可以通过索引访问，但由于集合是无序的，所以集合没有索引这种东西。

集合也有类似于列表的添加和删除元素的方法，例如 `add()` 和 `remove()`。但是集合不支持排序：我们无法对一个本来就没有“顺序”这个定义的东西进行排序。

例如：

```
my_set = {1, 2, 3, "Hello", True}
```

```
print(my_set) # 输出{1, 2, 3, "Hello", True}
my_set.add(4) # 添加元素4
my_set.remove("Hello") # 删除元素"Hello"
print(my_set) # 输出{1, 2, 3, 4, True}
```

C.3.4 字典 (dict)

字典是一个无序的可变集合，存储键值对 (key-value pairs)。我们可以使用花括号来创建一个字典，并使用键来访问值。字典的键必须是不可变类型（如字符串、整数等），而值可以是任意类型。例如：

```
my_dict = {"name": "Alice", "age": 30, "is_student": False}
print(my_dict["name"]) # 输出"Alice"
print(my_dict["age"]) # 输出30
print(my_dict["is_student"]) # 输出False
my_dict["age"] = 31 # 修改键"age"对应的值
print(my_dict) # 输出{"name": "Alice", "age": 31, "is_student": False}
```

对于字典，我们可以使用 `keys()` 方法获取所有的键，使用 `values()` 方法获取所有的值，使用 `items()` 方法获取所有的键值对。每一个键值对都是一个元组。例如：

```
print(my_dict.keys()) # 输出dict_keys(['name', 'age', 'is_student'])
print(my_dict.values()) # 输出dict_values(['Alice', 31, False])
print(my_dict.items())
# 输出dict_items([('name', 'Alice'), ('age', 31), ('is_student', False)])
```

C.3.5 高级操作

我们可以使用 `for` 循环来对以上各种复合数据类型进行遍历：

```
for item in my_list:
    print(item) # 遍历列表
for item in my_tuple:
    print(item) # 遍历元组
for item in my_set:
    print(item) # 遍历集合
for key, value in my_dict.items():
    print(key, value) # 遍历字典
for value in my_dict.values():
    print(value) # 遍历字典的值
```

我们还可以使用对这些复合数据类型进行切片。切片的语法是 `start:end:step`，其中 `start` 是起始索引，`end` 是结束索引（不包含），`step` 是步长（可以省略）。例如：

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(my_list[2:5]) # 输出[3, 4, 5]
print(my_list[::2]) # 输出[1, 3, 5, 7, 9]（步长为2）
print(my_list[::-1]) # 输出[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]（反转列表）
```

我们还可以使用列表推导式（list comprehension）来创建新的列表。列表推导式是一种简洁的语法，可以在一行代码中创建一个新的列表。例如：

```
squares = [x**2 for x in range(1, 11)]
print(squares) # 输出[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens) # 输出[2, 4, 6, 8, 10]
```

以上代码中，列表推导式的通用语法是 `[thing for item in iterable if condition]`，其中 `thing` 是你想要的东西，`iterable` 是可迭代对象（如列表、元组等），`condition` 是可选的条件。

看起来真就像说话一样。

C.3.6 字符串

字符串指的是一串字符的序列。Python 中的字符串是不可变的，这意味着一旦创建，就不能修改其内容。

比方说一个字符串：

```
my_string = "Hello, world!"
```

我们可以使用索引来访问字符串中的字符，索引从 0 开始。例如：

```
print(my_string[0]) # 输出'H'
print(my_string[7]) # 输出'w'
print(my_string[-1]) # 输出'!'
print(my_string[13]) # 抛出IndexError异常
my_string[0] = 'h' # 抛出TypeError异常，因为字符串是不可变的
```

字符串可以看作是一个字符的元组，因此我们可以使用切片来获取字符串的子串：

```
print(my_string[0:5]) # 输出'Hello'
print(my_string[7:]) # 输出'world!'
print(my_string[:5]) # 输出'Hello'
print(my_string[::2]) # 输出'Hlo ol!'
print(my_string[::-1]) # 输出'!dlrow ,olleH'（反转字符串）
```


我们还可以使用字符串的各种特有方法来操作字符串，例如：

```
my_string.lower() # 'hello, world!' (转换为小写)
my_string.upper() # 'HELLO, WORLD!' (转换为大写)
my_string.strip() # 'Hello, world!' (去除首尾空格)
my_string.replace("world", "Python") # 'Hello, Python!' (替换子串)
my_string.split(",") # ['Hello', 'world!'] (按逗号分割字符串)
my_string.find("world") # 7 (查找子串的位置)
my_string.startswith("Hello") # True (检查字符串是否以指定子串开头)
my_string.endswith("!") # True (检查字符串是否以指定子串结尾)
my_string.count("o") # 2 (统计子串出现的次数)
```

除了使用双引号来定义字符串，我们还可以使用单引号来定义字符串。两者完全等价。当然，由双引号框起来的字符串中包含单引号是可行的，反过来也可行，这个可以用来避免转义字符的使用。

```
my_string = 'Hello, world!'
print(my_string) # 输出'Hello, world!'
```

我们还可以使用三引号（单引号或双引号）来定义多行字符串，这样就可以在字符串中包含换行符了：

```
my_string = """Hello, world!"""
print(my_string) # 输出'Hello, world!'
my_string = '''Hello,
world!'''
print(my_string) # 输出'Hello,\nworld!'
```

\n 指的是换行。

C.4 函数和模块

C.4.1 函数

有时候，我们有一个功能需要多次使用，这时我们可以将其封装成一个函数（也叫方法）。Python 使用 `def` 关键字来定义函数。简单地说，函数可以把套路打包成一句话，就像汉语中的成语。

例如，我们可以定义一个函数来计算两个数的和：

```
def add(a, b):
    return a + b
```

一个函数应该以 `def` 开头，后面跟着函数名和参数列表。函数体使用缩进来表示。一个函数应该包含一个返回值，使用 `return` 关键字来返回结果就可以了。

在定义函数之后，我们可以在任意地方调用它，只需要提供函数希望的参数即可。例如：

```
result = add(3, 5)
print(result) # 输出8
```

我们也可以在函数中使用默认参数，这样在调用函数时可以省略某些参数：

```
def greet(name="World"):
    print(f"Hello, {name}!")

greet() # 输出"Hello, World!"
greet("Alice") # 输出"Hello, Alice!"
```

函数还可以使用递归来解决问题，即函数在其内部调用自身。例如计算阶乘：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5)) # 输出120
```

递归从某种程度上说也可以认为是循环的一种形式。同样的，递归也要有终止条件，否则会导致无限递归。

有些时候，我们希望函数参数只能存入某种类型的数据，或者使得某些函数返回某类特定的值。这个时候，类型注释就派上了用场。类型注释一般遵循冒号 + 类型或者箭头 + 类型的形式，例如：

```
def add(a: int, b: int) -> int:
    return a + b
```

上述代码的意思是，函数 `add` 接受两个整数参数 `a` 和 `b`，并返回一个整数。类型注释可以帮助我们更好地理解函数的输入输出类型，也可以在 IDE 中提供更好的代码提示。

但是我们需要注意一个问题：**类型注释不会强制执行类型检查**，换句话说它实际上依然是个注释而已——是给人方便开发用的，不是给电脑看的！

C.4.2 模块

有时候，一些功能大家都在用。这时候，为了防止重复工作，程序员们把这些功能打包成一个模块，供大家使用；而我们使用者只需要使用 `import` 关键字来导入模块，就可以使用模块中的许多方便的方法了。安装模块的方法参见正文部分5.2.2。

例如，我们可以导入 Python 的内置模块 `math` 来使用数学模块：

```
import math
print(math.sqrt(16)) # 输出4.0（计算平方根）
print(math.pi) # 输出3.141592653589793（圆周率）
print(math.factorial(5)) # 输出120（计算阶乘）
```

有时候，我们只需要导入模块中的某个函数或类，可以使用 `from ... import ...` 语法：

```
from math import sqrt, pi
print(sqrt(16)) # 输出4.0
print(pi) # 输出3.141592653589793
print(factorial(5)) # 抛出NameError异常，因为factorial没有被导入
```

还有一些时候，模块名称太长（例如 `matplotlib`），我们可以使用 `as` 关键字来给模块起一个别名：

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6]) # 画一条线
plt.show() # 显示图形
```

一些特定的模块有着约定俗成的简称，例如 `numpy` 通常简称为 `np`，`pandas` 通常简称为 `pd`，`matplotlib.pyplot` 通常简称为 `plt` 等。如果我们希望写出大家都能读懂、易于维护的代码，最好遵循这些约定。

C.5 文件操作

有时候，我们需要将数据保存到文件中，或者从文件中读取数据。Python 提供了简单的文件操作接口。例如，我们可以使用 `open()` 函数打开一个文件，并使用 `read()` 方法读取文件内容：

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content) # 输出文件内容
with open("example.txt", "w") as file:
    file.write("Hello, world!") # 写入内容到文件
with open("example.txt", "a") as file:
    file.write("\nThis is a new line.") # 追加内容到文件
```

我们可以使用 `with` 语句来自动管理文件的打开和关闭，这样可以避免忘记关闭文件导致资源泄漏的问题。

C.6 文科生的 Python

对于文科生来说，Python 的语法和特性已经足够简单了。我们可以使用 Python 来处理文本、数据分析等。同时，Python 有着非常蓬勃的生态，同学们可以调用许多现成的库来完成各种任务。

一般而言，文科生的 Python 缺不了三件套：超级 Excel (Pandas)、数据可视化 (Matplotlib) 和自然语言处理 (jieba)。这三件套可以帮助文科生处理数据、分析数据和可视化数据。

C.6.1 Pandas

Pandas 是一个强大的数据分析库，可以帮助我们处理表格数据。

Pandas 比较喜欢的文件是 CSV（逗号分隔的值）文件，我们可以使用 `read_csv()` 函数来读取 CSV 文件，并将其转换为 DataFrame 对象。当然，Pandas 也支持其他格式的文件，如 Excel、JSON 等。

```
import pandas as pd
df = pd.read_csv("data.csv") # 读取CSV文件
print(df.head()) # 输出前5行数据
df.to_csv("output.csv", index=False) # 将DataFrame保存为CSV文件
```

然后，对于这个 DataFrame 对象，我们可以使用各种方法来处理数据，例如筛选、排序、分组等：

```
filtered_df = df[df["age"] > 18] # 筛选年龄大于18的数据
sorted_df = df.sort_values(by="name") # 按照姓名排序
df[df['朝代'] == '唐']['作者'].value_counts() # 输出唐代谁被提及最多
pd.merge(df1, df2, on='书名') # 合并两个表，随意拼接
```

C.6.2 Matplotlib

Matplotlib 是一个强大的数据可视化库，可以帮助我们绘制各种图表：折线图看朝代更替与词频变化，柱状图比不同译本字数，词云图让关键词自己“跳出”屏幕。

比方说：

```
import matplotlib.pyplot as plt
plt.plot(df['年份'], df['酒'])
plt.title('唐诗中“酒”字频率变化')
plt.show()
```

只需要四行代码，就可以轻轻松松绘制成一张折线图。

当然，`plt` 有两个常见的小坑：

- 中文乱码？调包下面加一行 `plt.rcParams['font.sans-serif'] = ['SimHei']` 完事。
- 颜色太理工？`plt.style.use('seaborn-v0_8-pastel')` 一键切换温柔色调。

C.6.3 jieba

`jieba` 是一个中文分词库，可以帮助我们对中文文本进行分词处理。它可以将一段连续的中文文本切分成一个个单独的词语。

比方说，我想要把一整段《红楼梦》切成“贾宝玉”“林黛玉”“葬花”等词语，并统计频次：

```
import jieba
with open("book.txt", "r", encoding="utf-8") as file:
    text = file.read()
words = jieba.cut("林黛玉葬花") # 分词，精确模式
jieba.add_word("林黛玉葬花") # 添加新词，这个词会被识别为一个整体
```

当然，三件套最好一起用：Pandas 读数据 → jieba 分词 → Pandas 统计 → Matplotlib 可视化。这样可以轻松地完成论文中最让文科生们头痛的“数据分析”部分了！不过以上三件套的使用方法只是冰山一角，文科生们可以通过查阅相关文档和教程来深入学习。

当然，Python 也是一门语言，所有的语言都需要大量的练习和实践才能掌握；仅仅是看完这一章可能只需要一天，但是真正熟练应用语法可能需要一周的时间，熟练玩转文科生三板斧可能需要一个学期甚至还要多的时间。不过，不用担心：路在脚下，行则将至，只要你坚持下去，就一定能骄傲地说出：我是文科生，但我 Python 用得也很好。

后记

恭喜同学们完成了本手册的阅读！

当下，人心浮躁：网文讲究的是浮光掠影，视频讲究的是短平快，已经很久没有人能够静下心来阅读这么长的一本手册了。所以说，能看到这里的同学都是有心人，都是愿意花时间去学习、去实践的同学。你们的坚持和努力值得赞赏，也是我在缺乏合作者的时光中不断推进进度的动力。

谢谢。

不过也正常，大家看书总归是看个乐，我相信大多数人不会故意去做一些自己厌恶的事情去折磨自己。而不同的人喜欢的东西又不一样，所以说看不完手册也是非常正常的事情。毕竟人最终还是要过得快乐一些。当然，大伙都是貔貅，光进不吐，这导致整本书的内容全都是我手敲的，真是令人遗憾。（不过敲字也是我的一个爱好——这也算是因祸得福了？）

不要因为我说了这几句话就不给我提 PR 和 Issue 了啊喂（#'0'）！

这份手册的前身是《计算概论衔接课》第一部分的讲义。后经过本人的思考、修改和扩充，最终形成了近百页的手册。其中，LCPU 和 PKUHub 的同学们为我提供了许多宝贵的意见和建议，帮助我完善了手册的内容；也有许多同学在暑假课提出了问题和建议，也踩过不少坑，帮助我在编写手册时细化了许多内容、避免了许多错误。应该说，本手册编写完成，离不开众多个人与组织的无私帮助与鼎力支持。在此，也谨向所有给予我们指导、鼓励与便利的朋友们致以最诚挚的谢意。

感谢以下为本手册提供过贡献的人们（按提交时间排序）：

- LCPU Getting Started 全体成员
- PKUHub 全体成员
- [wszqkzqk](#)为多个部分提供了极为宝贵的建议，指出了一些严重的错误
- [Elkeid-me](#)提供了一些常用软件的推荐
- [AsTonyshment](#)指出了手册的一个落后之处：pku.edu.cn 邮箱已启用二次验证和客户端专用密码功能
- [ICUlizhi](#)为手册推荐了现在所用的主题文件
- [ha0xing](#)为手册增添了一行遗漏的代码，给出了使用更美观飘号的建议
- [whcpumpkin](#)提供了细化 LLM 使用方法的意见并给出了大纲
- [yjdyamv](#)为手册充实了 Typst 部分的内容

最后，感谢每一位能够读到这里的同学。愿你们在代码与终端的世界里，既能脚踏实地，又能仰望星空；既能把系统玩得风生水起，也能把生活过得热气腾腾。

再次致谢！

臧炫懿

2025 年 7 月，在燕园