

北京大学计算机基础能力手册

可爱大笨蛋

写在最前的一些吐槽

众所周知，马克思主义最重要的一句话是“理论指导实践，实践检验理论”。

但是有些人显然认为数学分析、高等代数、TCS 是人类智慧的结晶；而真正落到实处的工具（例如 git 等）是应用层的东西，是“俗物”。在许多大学的基础教育中都出现了这样的弊病，即过于注重理论而忽视实践。最终培养出来的不少学生，化身“数学 * 的搬运工”，在集群里拉屎。

所谓“数学 * 的搬运工”，指的自然不是真正的数学研究者们；数学 * 自然也不是数学。这种人有一个通病：特别喜欢理论，但是在其他方面（尤其是工程、生活、常识认知方面）上很差。其实差也不是问题，很多著名的理论研究者都有这样那样的奇怪行为习惯或者生活习惯。

但是这种人最大的问题是，明知自己将要从事工程实践，却不去主动提升自己在该方面的能力；更要命的是，这种人还喜欢把自己的世界观强加给别人，觉得只要理论水平高就行了，实践能力不重要了。

每个人脑子里实际上都隐含着一个人 World Model，在执行某个行为之前，大概率会在脑子里模拟一下动作的可能结果，俗称“过脑子”。而数学 * 的搬运工显然就是那些理论很好，但是过脑子能力很差的人。对于他们来说，过脑子这个动作是非常困难的，因为他们的 World Model 里缺乏一些基本的常识和经验，这些常识和经验是通过实践积累起来的，而不是通过理论学习来的。

但更恶心的是，数学 * 的搬运工还会自我繁殖。一旦这些搬运工们在大学里占据了主导地位，他们就会把自己的世界观强加给学生，于是出很多过分看重理论水平的题，让学生们觉得只要理论水平高就行了，实践能力不重要了。进而实现了他们的排外和自我繁殖，培养出更多的搬运工来。在项目上，上述情况就会导致 OIer 的通病：小细节上造火箭，大瓶颈却搁着不动；码风问题甚至相对而言已经变成一个小问题了。

我并不是批评理论研究者，相反，作为一个实用主义者，我觉得真正沉下心去做理论研究的人是非常伟大的，无论究竟有没有出成果。我知道自己不是搞理论那块料，所以我对他们的敬佩尤甚：他们在理论方面的贡献是非常重要的，他们的工作对于推动科学的发展是不可或缺的。

我真正批评的是那些把理论研究中的某些方面过分强调、甚至用这些玩意来批评工程优良实践中的人。我不止一次见到过在 C++ 做寄存器优化的人，也见过 `std::array<int, N>` 超时、连 perf 都不做就直接把这玩意改成 `int*` 的情况，更见过试图手写 STL 的情况。

那你说手动寄存器优化能不能做？理论上当然能啊，甚至能更快！

那你说 `std::array<int, N>` 和 `int*` 哪个更快？理论上当然后者更快啊！

那你说手写 STL 和标准库哪个更快？理论上当然是手写 STL 啊！

但是问题是你在哪个 C++ 项目见到过真正需要手动寄存器优化、见到过 `std::array<int, N>` 性能不够、标准库性能不够的情况？

你的可读性哪里去了？你的可维护性哪里去了？你的安全性哪里去了？你的开发效率哪里去了？这些东西真的不重要吗？

性能优化固然是非常重要的，但它不是唯一重要的东西。实际上这就是所谓“数学*的搬运工”的典型表现之一，不分场合的过分强调理论，而忽视了实践中的其他重要因素。对于大多数的工程项目来说，性能优化并不是最重要的因素。

那有些人又要拿我上述这段话去批评之前那些过分强调理论的人了——殊不知这也是数学*的搬运工的行为：

你在 Linux 内核代码写寄存器优化，写得好不好？写得好啊！

但是你在一个普通的 C++ 项目里写寄存器优化，写得好不好？写得不好啊！

那你说我是不喜欢理论吗？这边建议翻翻目录，整个第七节足足 80 页都在讲理论，甚至更多的地方中理论和实践都是穿插来的——实际上这些理论确实能够非常好的指导实践的过程。还是那句话：“理论指导实践，实践检验理论”。没有这些理论知识，系统设计、架构权衡、性能优化等实践确实是很难做的好的。

但你可别去背诵那玩意。

实践的第一要务自然是“实事求是”。“数学*的搬运工”们，放在以前，那叫“犯了形而上学的错误”。

于是，他们口中的“理论”变成了一坨符号化的东西，内部自治、极为排他，但是没用。这坨东西，现在通常被称为“赢学”。

别把理论搞成了赢学，我说真的。

更要命的是，现在很多学校确实只看数学*——恰恰在工程实践中，去背八股文、去替代计算机做运算，这些技能是被数学*所宣传的，但实际上是最没用的。而这种人往往却被 HR 喜欢——然后在项目组里天天搞些高屋建瓴的工作，开会、做 PPT 时讲得头头是道，甚至在技术讨论中也能说得天花乱坠，但真正让他写代码的时候，可能就只能写出一些垃圾代码了。那么你说，这样的人能不能算是一个好的工程师？

我觉得不能。

我甚至敢说，在 PKU 信科随机挑选一个学生，问他 glibc 是什么（甚至都不用问实现细节），问他 origin/main 和 main 有什么关系，我猜没几个人能说出来。难道说这东西真是什么很高深的东西吗？不，这只是一个非常基础的东西，是每个计算机相关专业的学生都应该知道的东西。但为什么很多学生都不知道呢？

考试不考罢了！

是了。考试这指挥棒指挥了刚刚通过三年高中学习的学生们的学习方向。这些学生思维定势，来到大学之后想当然地让考试指挥自己的学习了——这又是学生们自己的问题了。大学本身确实是一个相对自由的环境，学生们可以选择自己想学的东西来学；但教师不加以引导，能快速转变思路、认识到相关实践的重要性的学生自然少之又少，能主动去实践的人就更少了。

在 LCPU 群里曾有群众吹水“能意识到《手册》重要性的同学，其技术能力都已经不需要《手册》了”。

目前看来还真是。

LLM 时代怎么教、作业该怎么设计、怎么让学生真正去动手实践，其实我也不知道，这也确实是一个值得思考的问题。但是现在单按学校这种教法，纯考察理论的记忆、并要求学生把它默写下来，肯定是不行了。我们似乎也不能一味的相信后人的智慧，改变应该从现在开始尝试了。

目前来看这本《手册》算是开了个头，我在《手册》里总是说“自己写一遍”“Vibe 的前提是自己会写”，这都是希望同学们自己去实践，去动手写代码，去动手操作计算机，而不是仅仅停留在理论的层面上。因为只有通过实践，才能真正理解和掌握这些知识和技能，才能真正提升自己的能力。

虽说《手册》的内容和形式都还很不成熟，甚至可以说是很幼稚的了，但是至少也迭代了半年，也希望在后续的版本中能够不断改进。

发发牢骚，算是把近些年来的一些不满发泄出来了。

散会。

第二版前言

我们更应该关注那些不敢被问出的问题。

上面这句话是我和同学们讨论的时候，同学们提出来的。我觉得这句话非常有道理。

在过去的几个月里，我和一些同学们讨论了很多关于计算机基础教育的问题。我们发现，很多同学们在学习计算机基础知识时，往往会遇到一些“隐形的障碍”，这些障碍并不是因为他们不够聪明或者不够努力，而是因为他们缺乏一些基本的知识和技能，这些知识和技能并没有被明确地教授出来。于是我写了这么个东西。

但是，很快，可能是我个人的能力有限，也有可能是我太过傲慢，《手册》很快变成我一个人的狂欢。我自己添加的内容：从 C++ 的 `constexpr` 甚至 `constexpr`、到在《编程基础》出现的 GPG 密钥，再到我个人自作主张加上的多线程编程以及单纯为了方便自己写代码而加上的 CMake，甚至是我个人觉得有趣的内容，比如说 shell 脚本编程、正则表达式、以及一些 Linux 下的高级操作等等，乃至 Arch Linux 传教，完全没有考虑到读者的接受能力和需求。结果就是，《手册》变得越来越厚重，越来越难以阅读，越来越难以理解。深度和广度上，我个人对《手册》有很大的满足感，但是对于读者来说，可能并不是这样。

但你让我删掉这些内容，我又觉得很难受。因为这些内容都是我个人觉得非常有用的东西。我觉得如果我不把这些东西写出来，那么读者就会错过很多有用的知识和技能。另一方面，我费了许多心思写出来的东西，如果没有人看，那我也觉得很可惜。

于是，在第二版手册中，我将会把整个手册劈成更多小节，这些小节不仅可以分开阅读，还可以放在一起从头到尾通读：

1. 导航和方法——介绍在大学获取知识的方法和途径，帮助读者更好地利用手册和其他资源。**这一章是必须要读的。**
2. 基础使用：让计算机先跑起来——介绍计算机的基本使用方法，帮助读者快速上手计算机。
3. 终端和 Linux 基础：为开发铺垫语境——介绍 Linux 和终端的基础知识，帮助读者理解计算机的工作原理。
4. 开发环境：让代码能编译能跑——介绍编程环境的搭建和使用、C/C++、Python 的语法、包、编译工具链等内容，帮助读者快速进入编程状态。
5. 实用主义编程：写出更好的代码——介绍实用主义编程的理念和方法以及相关工具，如 git、密钥等，帮助读者提高编程效率和代码质量，促进更好的协作。
6. 原理速览：知其然，更知其所以然——介绍计算机的基本原理，帮助读者理解计算机的工作机制。

7. 工具箱——介绍一些实用的工具和技巧，帮助读者更好地使用计算机。
8. 延伸进阶阅读——介绍一些进阶的知识和技能，帮助读者进一步提升自己的能力。

我想要的从来就不是名义上的“厚重”，而是实实在在的“有用”。我希望读者们能够从《手册》中获得真正有用的知识和技能，而不是被一些无关紧要的内容所淹没。我希望读者们是真的不会被“overwhelmed”。

也正因此，本书有多种阅读方式，满足不同读者的需求。在读完第一节“导航与方法”之后，读者可以根据自己的需求选择不同的阅读路径。我希望读者能先阅读目录，再选择性的阅读想看的内容。**所有标注为“进阶阅读”的章节都是可选的，读者可以根据自己的兴趣和需求选择是否阅读。**

我给出以下几个参考路径：

- 之前完全没接触过计算机，希望能自己装个软件、写个 Hello World：
 1. 入门认知：《[初步使用计算机](#)》，学会怎么用键盘、怎么管理文件；
 2. 系统和软件：《[系统安装、基础配置和软件生态](#)》，学会装个软件；
 3. 编程入门：《[正式踏入编程世界](#)》，配好开发环境；
 4. 写第一个程序：《[C 语言入门](#)》或《[Python 高速入门](#)》，写出第一个 Hello World。
- 希望能简单学点脚本，写点小程序，能用 Git 管理代码和论文，用 LaTeX 等工具写点东西：
 1. 入门认知：《[系统安装、基础配置和软件生态](#)》和《[初步使用计算机](#)》，学会装个软件，学会怎么用键盘、怎么管理文件；
 2. 写作工具：《[L^AT_EX](#)》和《[L^AT_EX 进阶](#)》，学会用 LaTeX 写论文
 3. 脚本自动化：《[正式踏入编程世界](#)》、《[Python 高速入门](#)》和《[Python 常用包](#)》，学会用 Python 写点小程序、脚本，能画出漂亮的数据分析图表；
 4. 版本管理：《[Git 与版本控制](#)》，学会用 Git 管理代码和论文，防止写坏或丢失。
- 希望能写脚本、画图、跑模型，用 Linux 做实验，能用 Git 管理代码和论文：
 1. 快速上手：《[初步使用计算机](#)》和《[正式踏入编程世界](#)》，学会 VS Code 写 Python、虚拟环境装包跑脚本；
 2. 写作工具：《[L^AT_EX](#)》和《[L^AT_EX 进阶](#)》，学会用 LaTeX 等工具写论文；
 3. 数据处理：《[Python 高速入门](#)》和《[Python 常用包](#)》，学会用 Python 写点小程序、脚本，能画出漂亮的数据分析图表；
 4. Linux 环境：《[终端 101](#)》和《[开始使用 Linux](#)》，学会用 Linux 做实验，会用终端加速工作；
 5. 进阶玩法：《[Git 与版本控制](#)》和《[调试、测试和部署](#)》，学会用 Git 管理代码和论文，防止写坏或丢失，也能把别人的代码拿下就跑起来。
- 计算机相关专业学生：**我认为应该通读大部分：**
 1. 基础打底：《[终端 101](#)》和《[开始使用 Linux](#)》，学会 Linux、终端、WSL 等基础知识，为后续开发打好基础；
 2. 编程入门：《[正式踏入编程世界](#)》、《[C 语言入门](#)》、《[从 C 到 C++](#)》、《[C++ 进阶](#)》、《[C++ 真理大讨论](#)》、《[Python 高速入门](#)》和《[Python 常用包](#)》，学会配 VS Code、调试、写 C/C++ 和 Python，能写出较高质量的代码；

3. 工程能力：《C 系工程概述》、《Git 与版本控制》、《实用主义编程》，学会写 CMake，用 Git，多人协作；
 4. 系统原理：《信息表示和机器级代码》、《内存缓存管理和系统调用》和《计算机的启动和输入输出》，即使不学 ICS，作为信科学生也应该了解计算机的基本原理；
 5. 进阶玩法：《密钥与远程》+《调试、测试和部署》，学会远程开发、签名、调试等进阶工程技能。
- Oler 快速过渡到大学工程开发环境，避免“只会写算法不会写项目”：
 1. 环境过渡：《正式踏入编程世界》、《C 语言入门》、《从 C 到 C++》、《C++ 进阶》、《C++ 真理大讨论》，学会用 VS Code 写现代的 C++，学会用 CMake 管理项目，杜绝 Fake C++ (C with classes)；
 2. 工程规范：《Git 与版本控制》、《实用主义编程》，学会用 Git 管理代码，学会调试程序、规范代码，提升代码质量；
 3. 走向 Linux：《终端 101》和《开始使用 Linux》，以及《密钥与远程》，学会在 Linux 下开发，学会远程开发，适应大学的开发环境；
 4. 进阶可选：《信息表示和机器级代码》、《内存缓存管理和系统调用》和《计算机的启动和输入输出》，以及《现代高性能并发编程：异步、多进程和多线程》，了解计算机的基本原理和多线程编程。这前提是你想打底层、系统或科研，或者更先进的算法竞赛（如 HPC 竞赛）。
 - 纯装机，想买台新电脑，希望一条龙搞好新机、配好开发环境：《计算机的硬件、购机与验机》、《系统安装、基础配置和软件生态》、《正式踏入编程世界》，下午拿到新电脑，晚上就搞好开发环境，第二天就能提着新电脑去上课。
 - 有篇论文想投稿，内容都想好了，但不会用 LaTeX 排版：《 $\text{\LaTeX}$ 》和《 $\text{\LaTeX}$ 进阶》，学会用 LaTeX 写论文，搞定投稿格式，一周学完语法，顺便论文也排好。
 - “先跑 AI 再说”极速路线：7.5 miniconda - 15.3 PyTorch - 15.1 Pandas - 19.3.2 Docker 轻量化，搞定 AI 环境，直接跑模型。

希望第二版会比第一版对同学们更有帮助。

如有疑问、意见或建议，欢迎来该项目的仓库 [ZangXuanyi/getting-started-handout](https://github.com/ZangXuanyi/getting-started-handout) 查看本手册的源代码，欢迎提出 Issue 与 Pull Request。如不能访问，也可向我发送电子邮件咨询：zangxuanyi@stu.pku.edu.cn。也可加入该手册的官方讨论群聊（QQ 群号：1059423771）。

第一版前言

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，最终导致在校学生对计算机的使用和开发能力普遍较低，甚至出现了“念了四年本科却不会用 Git”的现象。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，“上课”这种获取信息的形式信息密度显然是不够的，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同学们自己去学习实践。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。

本手册可以认为是对大学常用计算机基础知识与基础开发能力的汇总，比起深度更注重广度，比起理论更注重实践。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。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，例如“使用鼠标”“关机”这种操作默认大家非常熟练，因此也不会涉及相关内容。

本手册正文分为两部分。第一部分面向计算机新手，讲述计算机基础知识，内容涵盖计算机组成和使用、编程环境配置、文本处理、Linux 等内容，旨在将同学们的水平提升至能够接受大学计算机基础教育的水平；第二部分则面向工程开发新手，讲述工程开发的一些基础内容，包含 C++ 和 Python 的语法基础、实用主义编程、调试和 C 语言工具链等，旨在将同学们的水平提升至能够进行简单工程开发的水平。

本文偏理论的章节中有一些“开放性思考和探索”板块，使用黄色背景色标注，旨在引导

Getting Started 项目: missing.lcpu.dev

学长: <https://github.com/PKUFlyingPig>

CS 自学指南: <https://csdiy.wiki/>

同学们进行更深入的思考和探索。笔者仅提供一些选题供参考，自己也不知道有没有一个确定的答案以及答案是什么，因此也不会给出答案甚至思路。有兴趣、有技术的同学们可以自行探索相关话题，甚至可以作为大作业、数学建模、科创甚至毕业设计的选题种子。当然，如果觉得难度过高，也完全可以跳过。

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。



知识速查

虽然笔者本人非常建议同学们通读本手册，但也理解有些同学可能只是想快速查阅知识点或获取某些技能。因此，笔者在此提供一个知识速查表，方便同学们快速定位到相关章。

0.1 配置相关环境

- 在纯 Windows 上搭建 C/C++ 编程环境：见《[在纯 Windows 上搭建 C/C++ 编程环境](#)》
- 在纯 Windows 上搭建 Python 编程环境：见《[Python、虚拟环境及其配置](#)》
- 在 WSL 上搭建 C/C++ 编程环境：见《[用 WSL 配置 C/C++ 环境](#)》

0.2 相关工具使用

- 正则表达式：见《[正则表达式](#)》
- VS Code：见《[IDE 及其选择](#)》
- Conda：见《[Python、虚拟环境及其配置](#)》
- Git：见《[Git 与版本控制](#)》
- GitHub：见《[GitHub](#)》
- SSH 密钥：见《[SSH 密钥](#)》
- GPG 密钥：见《[进阶：GPG 密钥和信任网络](#)》
- Markdown：见《[Markdown](#)》
- L^AT_EX：见《[L^AT_EX](#)》
- Typst：见《[Typst](#)》
- vim：见《[VIM](#)》
- Shell 命令行：见《[常用终端命令行辞典](#)》
- CMake：见《[CMake](#)》
- XMake：见《[XMake](#)》
- Meson：见《[Meson](#)》
- GDB 调试器：见《[使用调试器](#)》
- LLM、Vibe：见《[LLM、VLM 和提示词工程](#)》和《[Vibe Coding](#)》
- tmux 和 screen：见《[轻量级隔离：tmux 和 screen](#)》
- Docker：见《[重量级隔离：Docker](#)》，以及[官方文档](#)
- uv：见[uv 中文文档](#)
- pixi：见[pixi 最新文档](#)

官方文档: <https://docs.docker.com/get-started/>

uv 中文文档: <https://uv.doczh.com/>

pixi 最新文档: <https://pixi.sh/latest/>

0.3 其他优质教程汇总

- [PKU Getting Started](#) [文档](#)
- [CS 自学指南](#) • [星外之神的博客](#)，内容庞杂而丰富
- [中国科学技术大学 Linux 101](#) • [本书 GitBook 版本](#)，仍在施工中
- [清华大学计算机系学生科协技能引导文](#) • [Arthals 的博客](#)，ICS 学生的救星

0.4 PKU 校内常用网址集合

- [网络服务](#)
- [信息门户](#)
- [图书馆](#)
- [学生邮箱](#)
- [选课网](#)
- [教学网](#)
- [缴费系统](#)
- [教务网](#)
- [树洞](#)
- [课程测评](#)
- [正版软件](#)
- [课程查询](#)
- [笔记共享平台](#)，建议注册使用

PKU Getting Started: <https://missing.lcpu.dev>

CS 自学指南: <https://csdiy.wiki/>

中国科学技术大学 Linux 101: <https://101.lug.ustc.edu.cn/>

清华大学计算机系学生科协技能引导文档: <https://docs.net9.org/>

星外之神的博客: <https://wszqkzqk.github.io/tags>

本书 GitBook 版本: <https://lcpu-club.github.io/getting-started-handout-md/>

Arthals 的博客: <https://arthals.ink/>

网络服务: <https://its.pku.edu.cn>

信息门户: <https://portal.pku.edu.cn>

图书馆: <https://lib.pku.edu.cn>

学生邮箱: <https://mail.stu.pku.edu.cn>

选课网: <https://elective.pku.edu.cn>

教学网: <https://course.pku.edu.cn>

缴费系统: <https://cwsf.pku.edu.cn>

教务网: <https://www.dean.pku.edu.cn>

树洞: <https://treehole.pku.edu.cn>

课程测评: <https://courses.pinzhixiaoyuan.com>

正版软件: <https://software.pku.edu.cn>

课程查询: <https://class.wjsphy.top/>

笔记共享平台: <https://pkuhub.cn/>

目录

写在最前的一些吐槽	i
第二版前言	iv
第一版前言	vii
知识速查	ix
0.1 配置相关环境	ix
0.2 相关工具使用	ix
0.3 其他优质教程汇总	x
0.4 PKU 校内常用网址集合	x
第一部分 导航和方法	21
第一章 搜索和信息获取	22
1.1 搜索	22
1.1.1 搜索引擎的选择	22
1.1.2 搜索技巧	22
1.2 信息平台	23
1.2.1 官方文档、Wiki、论坛	24
1.2.2 Stack Overflow	24
1.2.3 GitHub	24
1.2.4 Wikipedia	24
1.2.5 其他著名博客和教程	24
1.2.6 国内优质平台	25
1.3 LLM、VLM 和提示词工程	25
1.3.1 选择 LLM	25
1.3.2 使用 LLM 的基本原则	26
1.3.3 LLM 的局限性	27
1.3.4 提示词工程简介	27
1.3.5 Cherry Studio	29

1.4 提问的艺术	31
第二部分 基础使用：让计算机先跑起来	33
第二章 计算机的硬件、购机与验机	34
2.1 现代计算机的硬件	34
2.1.1 中央处理器（CPU）	35
2.1.2 内存（RAM）	36
2.1.3 外存	37
2.1.4 显卡	38
2.1.5 主板	39
2.1.6 电源	39
2.1.7 输入输出设备	39
2.2 买计算机的一些理论	40
2.2.1 获取机器的途径	40
2.2.2 购买机器的原则	40
2.2.3 笔记本电脑的简单分类	40
2.2.4 奸商常见套路	41
2.3 整机（笔记本）	42
2.3.1 品牌的选择	42
2.3.2 线上购买	42
2.3.3 线下购买	42
2.4 组装机	42
2.4.1 机箱	43
2.4.2 CPU 和主板	43
2.4.3 显卡	44
2.4.4 内存和外存	44
2.4.5 电源	44
2.4.6 散热器	45
2.4.7 显示器	45
2.4.8 其他配件	47
2.5 验机	47
2.5.1 通电之前	47
2.5.2 通电	47
2.5.3 进系统	48
第三章 系统安装、基础配置和软件生态	50
3.1 操作系统安装	50
3.1.1 选择操作系统	50

目录	3
3.1.2 准备工作	52
3.1.3 制作启动盘	52
3.1.4 调整 BIOS 设置	52
3.1.5 安装操作系统	52
3.2 系统基础配置	53
3.2.1 驱动程序	53
3.2.2 安装常用应用软件	53
3.2.3 Linux 和 mac 上的软件安装	54
3.2.4 实用软件推荐	55
3.2.5 怎样卸载软件	57
3.2.6 进阶：利用任务管理器监测和管理进程	58
3.3 进阶：安装 Arch Linux	59
3.3.1 前置操作	59
3.3.2 开始安装	60
3.3.3 配置视窗，以及后续内容	66
3.3.4 总结	67
3.3.5 进一步学习	68
第四章 初步使用计算机	69
4.1 善用键盘	69
4.1.1 如何打字	69
4.1.2 高效打字	71
4.1.3 修饰键等特殊按键	72
4.1.4 快捷键	73
4.2 文件、目录及其管理	74
4.2.1 理解文件和目录	74
4.2.2 文件的大小	74
4.2.3 目录结构与路径	75
4.2.4 文件的扩展名	75
4.2.5 文件的链接（快捷方式）	76
4.2.6 文件的基本操作	76
4.2.7 更改文件的类型	76
4.2.8 文件的打开方式	77
4.2.9 文件的压缩与解压	77
4.2.10 进阶：高效的文件管理	78
4.3 校内网络配置指南	79
4.3.1 有线连接 PKU 校园网	79
4.3.2 无线连接 PKU 校园网	79
4.3.3 校外连接北大内网：北大 VPN	80

4.3.4	北大网盘	80
4.3.5	PKU 腾讯会议教育版	80
4.3.6	北大邮箱及其在第三方客户端的配置（以 OutLook 为例）	80
4.4	计算机的日常维护与安全	81
4.4.1	字体的安装与管理	81
4.4.2	计算机的日常维护	81
4.4.3	网安相关知识	82
4.4.4	进阶：软件版权和开源协议	85
4.5	进阶：正确且高效地获取网站资源	86
4.5.1	网站抓取	86
4.5.2	浏览器开发者模式	88

第三部分 终端和 Linux 基础：为开发铺垫语境89

第五章 终端 10190

5.1	选择 Shell	91
5.2	怎样使用终端？	92
5.2.1	命令的基本结构	92
5.2.2	终端的基本操作	92
5.3	常用终端命令行辞典	93
5.4	进阶：命令联动	96
5.4.1	重定向	96
5.4.2	管道	96
5.4.3	Here-String	97
5.4.4	进程替换	97
5.4.5	变量与命令替换	97
5.5	进阶：更好的终端	97
5.5.1	Oh My Posh 及其配置	98
5.5.2	Oh My Zsh	99
5.5.3	使用其他工具	99

第六章 开始使用 Linux 101101

6.1	Linux 发行版及其选择	102
6.1.1	CLab	103
6.1.2	实机安装	103
6.1.3	使用虚拟机	103
6.1.4	WSL	103
6.2	Linux 的基本操作	104
6.3	Linux 的文件系统	105

6.3.1	文件的组织	105
6.3.2	文件是谁的, 有什么属性	106
6.3.3	文件的联系	107
6.4	Linux 的进一步使用	108
6.4.1	root 权限的配置	108
6.4.2	软件的安装及其源的配置	108
6.4.3	关于 VIM 和 Nano	109
6.5	WSL 速成指南	110
6.5.1	快速安装	110
6.5.2	换发行版	111
6.5.3	文件互相访问	111
6.5.4	一口气配好开发环境	112
6.5.5	图形界面应用	112
6.5.6	性能调优、踩坑急救	112
6.5.7	进阶玩法	113
6.5.8	WSL 的局限性	114

第四部分 开发环境：让代码能编译能跑

115

第七章	正式踏入编程世界	116
7.1	编程语言初探	116
7.2	IDE 及其选择	117
7.2.1	为什么选择 VS Code?	117
7.2.2	安装 VS Code	120
7.2.3	配置 VS Code	120
7.2.4	VS Code 的一些设置项	120
7.3	在纯 Windows 上搭建 C/C++ 编程环境	122
7.3.1	C 系编译器及其环境配置	122
7.3.2	在 VS Code 中配置 C/C++	124
7.3.3	不依赖 VS Code 的编译方式	127
7.4	用 WSL 配置 C/C++ 环境	127
7.5	Python、虚拟环境及其配置	128
7.5.1	简单安装 Python	128
7.5.2	虚拟环境及其配置	128
7.5.3	在 VS Code 中配置 Python	129
7.5.4	不依赖 VS Code 的运行方式	130
7.5.5	在 VS Code 中配置终端	130
7.6	编写程序的基本素养	130
7.6.1	编写你的第一个程序	131

7.6.2	学会阅读错误信息	132
7.6.3	学会调试	134

第八章 C 语言入门 137

8.1	C 语言的基本语法	137
8.1.1	你的第一个 C 程序	137
8.1.2	变量及其运算	138
8.1.3	注释	139
8.1.4	输入、输出及其格式化	140
8.1.5	常变量	142
8.1.6	条件判断	142
8.1.7	循环	145
8.1.8	数组	147
8.1.9	字符串	149
8.1.10	声明、定义、变量的作用域与存储类	149
8.1.11	结构体和对齐	152
8.1.12	联合体	155
8.1.13	函数	155
8.1.14	函数的递归调用	156
8.1.15	类型强转	158
8.1.16	宏和预处理指令	158
8.2	指针和内存操作	159
8.2.1	什么是指针	159
8.2.2	指针的三条铁律	161
8.2.3	指针和数组、函数的配合	161
8.2.4	动态内存分配	163
8.2.5	静态变量和 const 指针	163
8.2.6	初级泛型编程: void 指针	164
8.2.7	指针常见错误	165
8.3	文件操作	165
8.3.1	文件读写	165
8.3.2	文件操作	166
8.4	多文件编程	167
8.4.1	头文件和源文件	167
8.4.2	include 原理	168
8.4.3	链接原理	168
8.5	标准库常用头文件	169
8.5.1	stdio.h	169
8.5.2	stdbool.h	170

8.5.3	string.h	170
8.5.4	stdlib.h	170
8.5.5	math.h	170
第九章 从 C 到 C++		173
9.1	从 C 到 C++ 的区别	174
9.1.1	第一个 C++ 程序	174
9.1.2	花括号初始化	174
9.1.3	C++ 的输入输出及其格式化	175
9.1.4	利用文件流进行文件读写	179
9.1.5	文件的其他操作	180
9.1.6	常量、常量和它们的关系	180
9.1.7	数组	182
9.1.8	字符串	183
9.1.9	结构体、联合体	184
9.1.10	枚举	185
9.1.11	智能指针（穿了衣服版）	186
9.2	C++ 独占的特性	189
9.2.1	命名空间	189
9.2.2	引用	191
9.2.3	C++ 函数的高级特性	193
9.2.4	类型推断	195
9.2.5	类型别名	196
9.2.6	类型强转	197
9.2.7	Lambda 表达式	198
9.2.8	多文件编程	200
第十章 C++ 进阶		205
10.1	面向对象编程	205
10.1.1	类和属性	205
10.1.2	自指	206
10.1.3	构造、析构、拷贝和赋值	206
10.1.4	封装	208
10.1.5	继承	209
10.1.6	多态	210
10.1.7	友元函数	211
10.2	泛型编程	211
10.2.1	函数模板	211
10.2.2	类模板	212

10.2.3	模板特化	213
10.2.4	非类型模板参数	214
10.2.5	变参模板	215
10.2.6	模板元编程	215
10.3	STL 和其他标准库	216
10.3.1	容器	216
10.3.2	迭代器	218
10.3.3	算法	218
10.3.4	字符串、流和字符串流	219
10.3.5	定长整数	221
10.3.6	位运算	221
10.3.7	正则表达式	223
10.3.8	小练	225

第十一章 现代 C++ 特性 230

11.1	string_view 和 span: 轻量级视图类型	230
11.1.1	字符串视图	230
11.1.2	span	231
11.1.3	和引用的异同	232
11.2	views 和 ranges: 声明式数据处理	232
11.2.1	从 Rust 说开去	232
11.2.2	回到 C++: 现代 C++ 的 views 和 ranges	233
11.2.3	C++ 视图的基本操作	234
11.2.4	ranges 扒开了说	235
11.2.5	投影	236
11.2.6	iota 视图: 生成序列	236
11.3	optional、variant 和 any: 类型安全的容器	237
11.4	模块	239
11.5	三相比较运算符	239
11.6	std::concept	240
11.7	std::format	241
11.8	C++23 简介	242

第十二章 C++ 真理大讨论 244

12.1	为什么不抛弃 C?	244
12.2	怎样写出真正的 C++?	245
12.2.1	从实例出发, 到实例中去	245
12.2.2	现代 C++ 的思维方式	249
12.3	否定之否定: FakeC++ 是否真正罪大恶极?	250

12.3.1	再回到实例	250
12.3.2	再看 Fake C++	252
12.3.3	相信编译器	252
12.4	解剖和验证：ranges 真的免费吗？	253
12.4.1	任务	253
12.4.2	预期结果	254
12.4.3	启示	254
12.5	C++ 中的一些良好实践	254
12.5.1	头文件	254
12.5.2	命名空间	255
12.5.3	变量声明和使用	256
12.5.4	函数和类定义	257
12.6	工程实践	258
第十三章 C 系工程概述		260
13.1	C 系包管理	260
13.1.1	C++ 包管理的困境和现状	260
13.1.2	手动装	261
13.1.3	使用包管理器	262
13.2	CMake	264
13.2.1	最小运行实例	265
13.2.2	语法	266
13.2.3	工具链	269
13.2.4	安装、导出、打包	270
13.2.5	常见坑	270
13.3	XMake	271
13.3.1	最小运行实例	271
13.3.2	语法速通	272
13.3.3	打包发布	274
13.3.4	常见坑	274
13.3.5	从 CMake 迁移到 XMake	274
13.4	Meson	275
13.4.1	最小运行实例	275
13.4.2	项目和目标	276
13.4.3	依赖项处理	277
13.4.4	子项目与 Wrap 系统	277
13.4.5	选项和配置	277
13.4.6	交叉编译	278
13.4.7	特点与比较	278

13.4.8 技巧与提示	278
第十四章 Python 高速入门	280
14.1 Python 的基本语法	280
14.1.1 Python 的变量	280
14.1.2 Python 的运算	281
14.1.3 输入、输出	281
14.1.4 注释	282
14.1.5 类型强转	282
14.2 控制程序的执行流程	283
14.2.1 条件语句	283
14.2.2 循环语句	283
14.3 复合数据类型	284
14.3.1 列表 (list)	284
14.3.2 元组 (tuple)	285
14.3.3 集合 (set)	285
14.3.4 字典 (dict)	286
14.3.5 高级操作	286
14.3.6 字符串	287
14.4 函数和模块	288
14.4.1 函数	288
14.4.2 模块	289
14.5 文件操作	290
14.6 Python 的面向对象	290
14.6.1 类和对象的基本定义	291
14.6.2 继承和多态	291
14.7 Python 与多文件	292
14.8 Python 语法小练	293
14.9 Jupyter Notebook	295
第十五章 Python 常用包	297
15.1 超级 Excel: Pandas	297
15.2 自动分词的结巴: jieba	297
15.3 PyTorch: 让你的电脑会学习	298
15.3.1 安装	298
15.3.2 张量、梯度下降和自动求导	299
15.3.3 数据集、数据加载器、预处理	299
15.3.4 nn.Module 搭积木式写网络	300
15.3.5 真正训练	300

目录	11
15.3.6 做个实验	301
15.3.7 我看完了，然后呢	303
15.4 NumPy+SciPy：科学计算的利器	303
15.4.1 NumPy：多维数组和矩阵运算	303
15.4.2 SciPy：科学计算的扩展	304
15.4.3 两个一起，双倍开心	305
15.5 Matplotlib：数据可视化的神器	306
15.5.1 基本用法	306
15.5.2 和其他包协同工作	307
15.5.3 子图布局	308
15.5.4 风格和导出	309
15.5.5 常见坑与提示	310
15.6 SymPy：符号计算高级计算器	310
15.6.1 基本数据类型	311
15.6.2 矩阵和线性代数	312
15.6.3 输出、可视化	312
15.6.4 常见坑	312
15.7 爬虫	313
15.8 OpenAI：自己做自己的 Agent	313

第五部分 实用主义编程：写出更好的代码

315

第十六章 Git 与版本控制	316
16.1 Git 的工作原理	316
16.2 下载 Git	318
16.3 Git 信息设置	318
16.4 Git 的最基本使用	319
16.4.1 提交	319
16.4.2 回退	319
16.4.3 排除相关文件	320
16.4.4 查看历史记录	321
16.4.5 打包备份	321
16.4.6 比较差异	321
16.5 分支管理	322
16.5.1 创建和切换分支	322
16.5.2 分支变基	322
16.5.3 合并分支和冲突解决	322
16.5.4 删除分支	324
16.5.5 压缩提交	324

16.6	标签管理	324
16.6.1	创建标签	324
16.6.2	查看标签	324
16.6.3	删除标签	325
16.7	“摘樱桃”	325
16.8	在 VS Code 中配置 Git	325
16.9	GitHub	325
16.9.1	GitHub 界面指南	326
16.9.2	创建仓库	327
16.9.3	使用仓库	328
16.9.4	更改关联	329
16.10	多人协作	330
16.10.1	Fork	330
16.10.2	Pull Request	330
16.10.3	Lint	330
16.10.4	成熟项目的分支管理策略	331
第十七章 密钥与远程		332
17.1	SSH 密钥	332
17.1.1	SSH 密钥的创建	332
17.1.2	SSH 密钥的使用	333
17.1.3	使用 VS Code 建立 SSH 连接	335
17.2	进阶：GPG 密钥和信任网络	336
17.2.1	GPG 密钥的生成	336
17.2.2	GPG 密钥的使用	337
17.2.3	把加密搬上 YubiKey	339
17.2.4	信任网络：怎么证明你是你	340
17.2.5	吊销和删除	340
第十八章 实用主义编程		342
18.1	工程代码基本准则	342
18.2	代码风格	344
18.2.1	通用代码风格指南	344
18.2.2	主流语言代码风格指南	345
18.2.3	关注特定项目与社区的风格	347
18.2.4	善用自动化工具	347
18.2.5	注释	347
18.3	防御式编程	348
18.3.1	异常处理	348

目录	13
18.3.2 断言	349
18.4 监控程序的运行情况	350
18.4.1 日志	350
18.4.2 其他监控手段	351
18.5 常见的代码架构	351
18.5.1 MVC 架构	351
18.5.2 MVVM 架构	352
18.5.3 洋葱架构（干净架构）	352
18.5.4 微服务架构	353
18.6 其他一些碎碎念	353
18.6.1 Vibe Coding	353
18.6.2 代码审查	354
18.6.3 TDD	355
18.6.4 如何管理依赖	356
18.6.5 文档自动化	356
18.7 实例：帮某位大一同学修改代码	357
18.7.1 先把代码变得更 C++ 一点	361
18.7.2 重构代码结构：更加 OOP、更加模块化	363
18.7.3 下一步：让它更现代、更健壮	366
18.8 环境管理与配置	371
18.8.1 什么是“环境”？	371
18.8.2 环境管理工具的进化	372
18.8.3 新的意识：DevOps	372
18.8.4 最佳实践：micromamba+Pixi	373
第十九章 调试、测试和部署	375
19.1 先救命	375
19.1.1 使用调试器	375
19.1.2 尸检	377
19.2 再治病	378
19.2.1 CPU 占用过高	378
19.2.2 内存占用过高	378
19.2.3 IO 卡死	378
19.3 再调养	378
19.3.1 轻量级隔离：tmux 和 screen	379
19.3.2 重量级隔离：Docker	379
19.3.3 端口映射	381
19.3.4 单元测试	381
19.3.5 集成测试	382

19.4	买保险	382
19.5	去工作	383
19.5.1	部署	383
19.5.2	回滚	384
第二十章 数据存储和交换		385
20.1	SQL 数据库	385
20.1.1	键和表	385
20.1.2	SQL 查询语言	386
20.1.3	SQL 数据库的自动更新	386
20.2	数据交换格式	387
20.2.1	JSON	387
20.2.2	XML	388
20.2.3	CSV	389
20.2.4	YAML	390
20.2.5	Toml	390
20.2.6	INI	390
20.2.7	非显式数据交换格式	391
20.3	NoSQL	391
20.4	正则表达式	391
20.4.1	语法概要	392
20.4.2	使用示例	393
第六部分 原理速览：知其然，更知其所以然		394
第二十一章 信息表示和机器级代码		395
21.1	信息怎么被表示？	395
21.1.1	整数	395
21.1.2	浮点数	396
21.1.3	地址	397
21.1.4	字符	397
21.2	程序怎么跑起来？	403
21.2.1	预处理	404
21.2.2	编译和汇编	404
21.2.3	常见汇编码	404
21.2.4	其他情况	405
21.2.5	从文件到程序	405
21.2.6	链接	407

第二十二章 内存缓存管理和系统调用	408
22.1 内存怎么被管理？	408
22.1.1 虚拟内存、页面、页表	408
22.1.2 磁盘交换区和缺页异常	409
22.1.3 内存分配器	409
22.1.4 一个例子	410
22.2 怎么压榨 CPU 的性能？	410
22.2.1 缓存的分级	410
22.2.2 缓存行和局部性原理	411
22.2.3 组相联和标签	411
22.2.4 未命中常见工作流程	411
22.2.5 大矩阵乘法的工作原理	411
22.2.6 流水线	412
22.2.7 现代 CPU 的架构	414
22.3 系统怎么被调用？	415
22.3.1 系统调用的处理流程	415
22.3.2 系统调用的代价与实践尝试	416
第二十三章 计算机的启动和输入输出	417
23.1 计算机如何启动？	417
23.1.1 加电自检	417
23.1.2 硬件初始化和配置	418
23.1.3 引导	418
23.1.4 内核加载与驱动初始化	418
23.1.5 用户空间启动与登录界面	418
23.1.6 问题解答	419
23.2 计算机怎么和外界交互？	420
23.2.1 I/O 地址和端口	420
23.2.2 中断与 DMA	420
23.2.3 设备驱动	420
23.2.4 热插拔、总线枚举	421
23.2.5 从电信号到文件：以保存 U 盘为例	421
23.2.6 性能与瓶颈——为什么 USB3.0 跑不满 5Gbps？	421
23.2.7 安全关卡——I/O 权限与恶意外设	422
23.3 计算机如何联网？	422
23.3.1 带宽、传输速率、延迟和丢包率	422
23.3.2 内网和外网	423
23.3.3 网络协议	424
23.3.4 网络设备	424

23.3.5	网卡上线	424
23.3.6	DHCP: 办临时身份证 (IP)	425
23.3.7	ARP: 户籍管理人员	425
23.3.8	路由: 熟知路径的快递员	425
23.3.9	NAT: 内网到外网的门房	426
23.3.10	DNS: 域名解析翻译官	426
23.3.11	TCP 三次握手: 可靠连接的建立	427
23.3.12	HTTP: 网页浏览的协议	427
23.3.13	TLS: 加密数据的押运者	427
23.3.14	HTTP/2 和 QUIC: 更快的数据传输	428
23.3.15	Wi-Fi: 空气中的以太网	428
23.3.16	CDN: 网络上的缓存	429
23.3.17	代理和 VPN: 外网到内网的桥梁	429
第二十四章 数据结构		430
24.1	线性数据结构	430
24.1.1	顺序表	430
24.1.2	链表	431
24.1.3	栈和队列	432
24.2	非线性数据结构	434
24.2.1	树	434
24.2.2	图	438
24.3	特殊数据结构	439
24.3.1	二叉树及其变体	439
24.3.2	哈希表	441
24.3.3	堆	442
第二十五章 算法基础		444
25.1	算法及算法分析的基本概念	444
25.1.1	衡量算法的效率	444
25.1.2	问题的复杂性及其初步判断	445
25.2	常见精确算法思想	447
25.2.1	查表	447
25.2.2	模拟	448
25.2.3	枚举	449
25.2.4	贪心	450
25.2.5	分治和减治	450
25.2.6	动态规划	451
25.2.7	回溯和分支限界	453

目录	17
25.2.8 线性规划	455
25.3 常见近似算法思想	456
25.3.1 随机化算法	456
25.3.2 启发式算法	457
25.4 搜索和排序	460
25.4.1 排序算法	461
25.4.2 搜索算法	464
25.5 复杂数据结构上的问题	467
25.5.1 树和图上的搜索	467
25.5.2 最短路问题	467
25.5.3 最大流问题	469
25.6 在线算法简介	470

第七部分 工具箱

473

第二十六章 文字排版:Markdown 和 Typst

474

26.1 Markdown	475
26.1.1 Markdown 的语法	475
26.2 Typst	478
26.2.1 Typst 的安装	478
26.2.2 Typst 的语法	479

第二十七章 L^AT_EX

482

27.1 L ^A T _E X 发行版的安装和配置	483
27.1.1 Windows 机器	483
27.1.2 Linux: 以 Ubuntu 为例	483
27.1.3 L ^A T _E X 在 VS Code 的配置	484
27.2 初探 L ^A T _E X	485
27.2.1 命令和环境	486
27.2.2 正确输入符号	486
27.2.3 空格、换行和分段	487
27.2.4 L ^A T _E X 文档结构	489
27.2.5 标题、标题页	490
27.2.6 摘要	491
27.2.7 章节、附录、目录	491
27.2.8 标准文档类的选项	493
27.2.9 文档类	493
27.2.10 字体、字号和行距	494
27.2.11 特殊文字效果	496

27.2.12 文内交叉引用	497
27.2.13 引用超链接	498
27.2.14 参考文献	498
27.3 排版中文文档	499
27.3.1 ctex 文档类	499
27.3.2 ctex 宏包	500
27.3.3 设置标题样式	500
27.3.4 中文字体	501
第二十八章 L^AT_EX 进阶	502
28.1 利用 L ^A T _E X 排版数学公式	502
28.1.1 普通的数学符号	503
28.1.2 巨算符	504
28.1.3 数学模式的字体调整	504
28.2 图、表、浮动体	505
28.2.1 插入图片	505
28.2.2 插入表格	507
28.3 自定义命令和自定义环境	508
28.3.1 自定义命令	508
28.3.2 自定义环境	509
28.4 使用 Beamer 类制作幻灯片	510
28.4.1 Beamer 文档的基本结构	510
28.4.2 Beamer 的常用强调	510
28.4.3 Beamer 的主题	511
28.4.4 ctexbeamer	512
28.5 使用 ModernCV 类制作简历	512
第八部分 延伸进阶阅读	514
第二十九章 现代高性能并发编程：异步、多进程和多线程	515
29.1 从一口气做完到边做边等	515
29.2 异步：一个进程来回跳	516
29.2.1 事件循环和协程	516
29.2.2 异步的边界	516
29.3 多进程编程：尤里复制人	517
29.3.1 fork 之后	517
29.3.2 进程间的通信	517
29.3.3 进程池	518
29.3.4 多进程编程	518

29.3.5	那为什么还要多线程?	518
29.4	程序执行原理	519
29.4.1	进程	519
29.4.2	线程	519
29.5	多线程编程基本思想	520
29.5.1	并发和并行	520
29.5.2	问题抽象	520
29.5.3	解决问题	521
29.5.4	新的问题与内存模型	523
29.5.5	实践之前的碎碎念	523
29.5.6	相关模型解释	524
29.6	多线程的实践	525
29.6.1	实例: 问题描述	525
29.6.2	实例: 设计思路	525
29.6.3	实例: 实际工作	526
29.6.4	线程池	530
29.6.5	怎样调试?	531
29.6.6	什么时候不能用多线程?	532
29.7	扩展阅读	533
第三十章 神经网络和机器学习		534
30.1	从最简单的神经元说起	534
30.1.1	0-1 二分类问题	534
30.1.2	怎么求解?	535
30.2	多层神经网络	537
30.2.1	新的问题: MNIST-10	537
30.2.2	多层神经网络和激活函数	538
30.2.3	正向传播和反向传播	540
30.2.4	正则化	541
30.2.5	归一化	542
30.2.6	数据清洗和数据增强	544
30.3	更复杂的网络结构: 卷积、池化和残差	544
30.3.1	卷积层	544
30.3.2	池化层	546
30.3.3	卷积层和池化层的反向传播	547
30.3.4	残差块	547
30.3.5	残差块的反向传播	548
30.4	另一番光景: 循环神经网络	548
30.4.1	朴素的循环神经网络	548

30.4.2	改进的 RNN: LSTM 和 GRU	549
30.4.3	Transformer: 一拍脑袋的新思路	549
30.5	缺少数据的做法: 对抗学习和强化学习	551
30.5.1	对抗学习	551
30.5.2	强化学习	552
30.5.3	两条道路的融合	554
30.6	实践: PyTorch 中的高级神经网络模块	554
30.6.1	使用 nn.Module 构建神经网络	554
30.6.2	模型的训练和评估	555
30.6.3	使用预定义的神经网络模块和优化器等	556
30.6.4	数据加载和预处理	557
30.6.5	模型保存和加载	557
30.7	CUDA-C 和 GPU 编程简介	558
30.8	计算机视觉简介	561
30.8.1	目标检测	561
30.8.2	图像分割	562
30.9	练习	562
	后记	566

第一部分

导航和方法

第一章 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，我们需要借助其他的方式来获取信息。

1.1 搜索

1.1.1 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，前者会优先搜索国内网站，而后者则会显示全球的搜索结果。我们看到，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。

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。

说明

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。

1.1.2 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；另一方面，使用引号还会强制搜索引擎搜索含该关键词本身的内容，而不是其同义词或者近义词，甚至不出现。例如，当我搜索“deepseek api”时，会搜索到许多不含 api 的结果。但是假如我搜索“deepseek "api"”，则会强制搜索引擎搜索含有 api 的结果。
- 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。
- 使用“filetype:”限制搜索结果的文件类型。例如，我们如果想要搜索“机器学习”的 PDF 文档，可以使用“机器学习 filetype:pdf”进行搜索。这样的搜索可以方便地寻找文献等。

另，特定的搜索工具也可以帮助我们搜索一些特定的信息，例如 Google 学术和微软学术可以高速查找论文和引用；GitHub Code Search 可以帮助我们搜索 GitHub 上的代码片段；Google Lens、Bing Visual Search 等可以帮助我们通过图片搜索相关信息。

判断信息的可靠性虽然不属于搜索技巧，但也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？

1.2 信息平台

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

1.2.1 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。

1.2.2 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。

1.2.3 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，当你对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。

1.2.4 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。

1.2.5 其他著名博客和教程

W3Schools 提供了许多关于开发的教程，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。

1.2.6 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。

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。

1.3 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。

1.3.1 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](https://artificialanalysis.ai/leaderboards/models) 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。

1.3.2 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。

警告

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。

HHEM Leaderboard: <https://vectara-leaderboard.hf.space/>

Context Arena: <https://contextarena.ai/>

Fiction.liveBench: <https://fiction.live/stories/Fiction-liveBench-May-22-2025/oQdzQvKHw8JyXbN87>

1.3.3 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¹，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。在学术上，这往往是不可忍受的，例如 LLM 会编造不存在的论文，因此在学术场景下使用 LLM 来进行资源粗略搜索、辅助阅读文献乃至资源整理时，务必慎之又慎。

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。



图 1.1: LLM 幻觉示意图

1.3.4 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。

¹一种由 Google 于 2017 年提出的神经网络结构。通俗地说，它像一位“同时浏览整页文字、迅速找出词与词之间关系”的超级阅读者；学术上，它用“多头自注意力机制”并行捕捉文本中任意两个位置之间的依赖，取代了传统的逐字或逐句顺序处理。

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。

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：

1

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。

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：

1

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：

1

... 正常的 Prompt...

2

3

请逐步推理。

“请逐步推理”这五个字²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。

举例提示 举例提示指的是在提示词中提供一些示例来指导模型的输出。举例提示可以帮助模型更好地理解用户的意图，并生成更符合预期的输出。例如：

1

... 正常的 Prompt...

2

3

你应输出如下格式的内容：

²Let's think step by step 也是五个词。真是巧合。

4 ... 格式说明...

5

6 例如:

7 ... 示例内容...

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

自治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。

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 Token³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。

1.3.5 Cherry Studio

Cherry Studio 是一个支持多模型的 LLM 助手，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 对应的模型提供商，然后启用该模型提供商并输入 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。不过一般情况下硅基流动确实是最常见的模型提供商，基本提供了国内所有的开源模型，价格也相对公道。

³Token 是一段文本被切分后的最小处理单元，一般一个单词是一个 Token，单个汉字在 0.3 到 0.5Token 之间。

Cherry Studio 界面的左侧是 Agent 和助手列表，可以使用其提供的 Agent⁴和助手⁵模板来快速构建自己的 Agent 或助手并使用。

在 Cherry Studio 中，Agent 和助手的 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，可以与 LLM 进行对话，输入提示词并获取输出结果。

Cherry Studio 还支持为一个特定话题提供提示词，我们也可以把它理解成一种特殊的 System Prompt。我们可以在这里为一个特定的话题提供一些提示词，这些提示词会在每次与 LLM 进行对话的时候自动添加到提示词的开头，这样可以提高模型对该话题的关注度和处理能力。

另外，也可以在这里做一些高级设置，例如调整温度（Temperature）等参数。温度参数控制了模型输出的随机性，温度越高，输出越随机；温度越低，输出越确定。一般情况下，我们可以将温度设置为 0.7，这样可以在保证输出质量的同时，增加一定的多样性。

知识库 知识库是 Cherry Studio 提供的重要功能之一。它可以理解为一个可以存储和管理信息的地方，我们可以把一些重要的信息或者资料存储在知识库中，然后在使用 LLM 的时候，可以直接从知识库中调用这些信息，这样可以提高模型的输出质量和效率。

要创建一个知识库，首先要打开知识库界面，选择左侧的“添加”按钮，输入知识库的名称并分配对应的嵌入模型⁶和重排模型⁷，按预算将知识库的请求文档片段数量⁸设置为一个合适的值（例如 10），然后点击“创建”按钮即可。创建完成之后，我们就可以在知识库中添加文档了。我们可以直接上传文件，或者直接输入文本来添加文档。添加完成之后，我们就可以在使用 LLM 的时候，从知识库中调用这些信息了。

这比直接把这些信息或文件放在提示词中要好得多。其原因有二：

- 超出最大上下文长度：LLM 在处理信息的时候有一个最大上下文长度的限制，如果我们把大量的信息或者文件直接放在提示词中，可能会超出这个限制，导致模型无法处理这些信息或者文件；即使一次没有超出这个限制，随着多轮对话的进行，LLM 会忘记之前的信息或者文件。
- 信息的结构化：嵌入模型和重排序模型会辅助主要模型将知识库里的信息进行处理和检索，这样可以提高模型对这些信息的理解和利用能力，从而提高输出质量；而如果我们直接把这些信息或者文件放在提示词中，模型可能无法有效地处理和利用这些信息或者文件。

在与 LLM 进行对话的时候，也可以将特定的、较为重要的信息添加到知识库中，防止超出最大上下文长度导致 LLM 忘记这些信息。

⁴Agent 可以理解有一种有主观能动性、能根据用户的要求，运用提供的多种工具（例如搜索工具、计算工具等）来完成任务的智能体

⁵助手可以理解有一种没有主观能动性、只能根据用户输入的提示词来回答问题的智能体

⁶嵌入模型指的是一种将文本转换为向量的模型，这些向量可以用来表示文本的语义信息；通俗的说就是让文本变成 LLM 能够理解的形式

⁷重排序模型指的是一种根据相关性对文本进行排序的模型，这些模型可以帮助我们在知识库中找到最相关的信息

⁸请求文档片段数量指的是在使用 LLM 的时候，从知识库中调用的文档片段的数量，这个数量越多，模型能够获取的信息就越多，但是也会增加模型的处理负担和响应时间，也会增加消耗的 tokens 数量，因此需要根据实际情况来设置这个数量

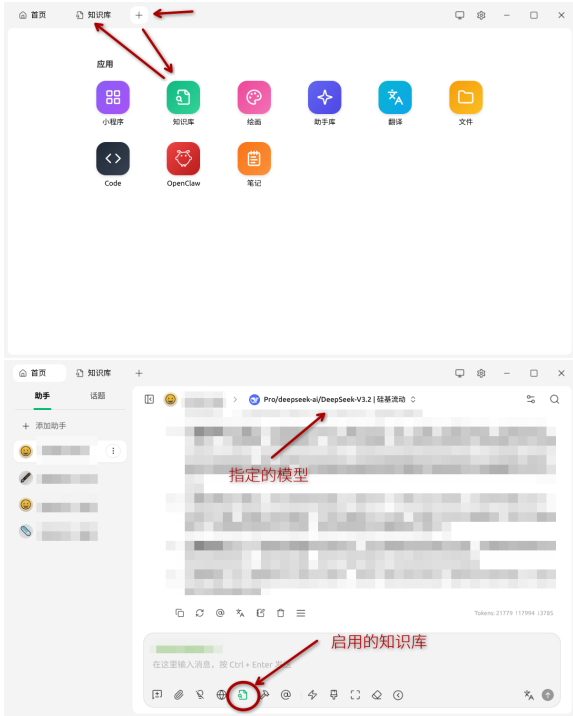


图 1.2: Cherry Studio 知识库界面示例

1.4 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。

注意

Issue 并不总是一个问答平台，而是一个问题追踪平台。Issue 的主要目的是追踪项目中的 Bug 和功能请求，而不是用来提问和讨论。因此，在使用 GitHub Issue 提问之前，我们需要确认项目是否欢迎在 Issue 区提问。

部分项目有专门的论坛或渠道反馈交流问题，有的甚至 feature request 也在论坛上，而 issue 是专门用来留作 bug 追踪的。在这些项目的 issue 区提问是不受欢迎的，会被认为在浪费维护者的时间。我们建议在联系维护者之前，确认项目是否欢迎在 issue 区提问，或者自己

遇到的问题确实是项目的 bug。有的项目会在 README 或官网里面说明，有的会在提交 issue 的要求选择模板或进入外部链接提交问题。如果不确定是项目本身的问题，也不建议在 open issues 超过一千的项目里面提问，维护者处理这么多 issue 已经够忙的了。

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！”

除此之外，当你需要展示代码的时候，应当尽可能使用文字而不是附带图片。这在 GitHub、知乎等平台上尤其重要，因为其支持 Markdown 语法，可以让代码容易阅读。对于飞书、钉钉等工作软件，它们也支持代码块，因此要尽可能地使用代码块来展示代码。但是，对于 QQ 和微信等软件，它们不支持 Markdown 语法，且在聊天框下直接粘贴代码会导致代码完全无法阅读，此时可以使用截图工具来保证代码的可读性。

一定要截图，而不是拍屏！拍屏往往会因为光线、角度等问题导致无法阅读，这显得极其不专业且令人恼火。在 Windows 系统中，我们可以使用 Win+Shift+S 快捷键来截图，或者使用自带的截图工具等；对于 macOS 系统，我们可以使用 Command+Shift+4 快捷键来截图。如果你使用 QQ 或者微信，也可以使用其自带的截图工具来截图。

第二部分

基础使用：让计算机先跑起来

第二章 计算机的硬件、购机与验机

在大学，计算机是最重要的文具，没有之一。在 PKU，天天和计算机打交道的信科院同学就不用说了，其他学院的同学们也离不开计算机：数院的同学们需要用计算机做数值计算和数据分析，物院的同学们需要用计算机做模拟和实验控制，化院的同学们需要用计算机做分子建模和数据处理，工学院的同学们需要用计算机做设计和仿真，甚至文科院的同学们也需要用计算机写论文和做数据可视化。而就在不学习的时候，计算机也是相当重要的娱乐和社交工具。因此，拥有一台合适的计算机对于大学生活来说是非常重要的。本节就会带着大家一起了解一下，如何选购和验收一台适合自己的计算机。

阅读材料：计算机发展简史

自古以来，人类不断探索怎样高效地计算。上古时期，人们使用算盘、算筹帮助运算；而 18 世纪，查尔斯·巴贝奇设计的“差分机”“分析机”一般认为是机械计算机的雏形。

现代计算机的理论灵魂来自阿兰·图灵，他在 1936 年提出“图灵机”模型，明确了“可计算问题”与“不可计算问题”的界限^a，为计算机科学奠基。而冯·诺依曼则通过提出程序与数据共存的存储结构，将计算机带入现实，这一“冯·诺依曼架构”至今仍是主流。第一台通用电子计算机 ENIAC 于 1946 年诞生，采用电子管实现高速计算，标志着电子计算机时代的开启。此后，晶体管、集成电路和微处理器相继问世，计算机不断向小型化、高性能演进，并借图形界面与互联网普及至千家万户。如今，计算机已融入生活各个角落，并在人工智能、云计算等新技术推动下持续发展。

^a一般认为，如果一个问题的解法可以被形式化为一个算法，那么这个问题就是可计算的；否则就是不可计算的。很多问题是可计算的，但也有一些问题（如停机问题：是否存在一个算法，能够判断任意程序在任意输入下是否会停止运行）是不可计算的。

2.1 现代计算机的硬件

无论是日常使用计算机，还是购买计算机，我们都需要对计算机的硬件有一个初步的了解。

计算机是一个相当复杂的系统，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 Google Chrome、Microsoft VS Code、Tencent QQ 等应用程序。

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2.1所示。下面将会逐个介绍这些设备。

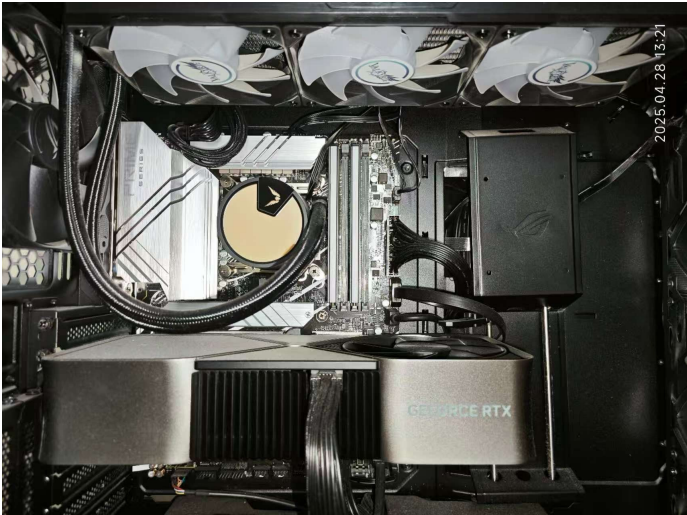


图 2.1: 一台台式计算机的主机设备



图 2.2: 一台笔记本电脑打开后盖的样子

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，但是从程序员视角来看，其本质上都以二进制存储。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。

衡量 CPU 的性能有很多指标，最常见的两个是主频和核心数。主频指的是 CPU 每秒钟能够执行多少个指令，单位是 GHz（千兆赫兹）；核心数指的是 CPU 内部有多少个独立的处理单

元，能够同时处理多少个任务。一般来说，主频越高、核心数越多，CPU 的性能就越强。

CPU 是计算机工作时的主要热量来源之一，因此必须有一个散热器来帮助 CPU 散热。一般情况下散热器也会连接到主板上（主板的知识将在后面介绍），由主板控制风扇的转速以调节散热效果。也有的 CPU 性能很高，风冷无法满足散热需求，这时就需要使用水冷散热器来帮助散热。为了保证接触良好，CPU 和散热器之间往往会涂抹导热介质（例如硅脂、垫片、液态金属等）来提高散热效率。

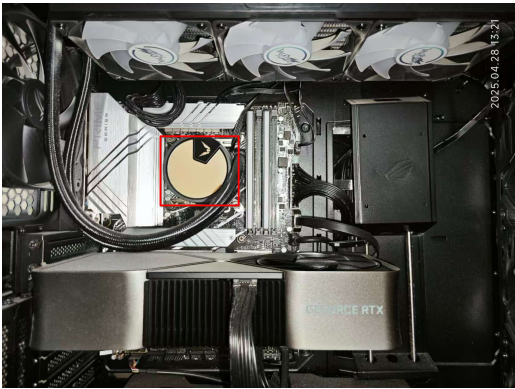


图 2.3: CPU，但这实际上是水冷散热器，CPU 压在散热器下面

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 就在内存中寻找需要的数据或者指令，如同在排列整齐的书架上寻找需要的书籍。

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：这是因为，内存使用的 DRAM 颗粒是用电容存储电荷的，而电容几毫秒就漏光了，所以必须上电、不停刷新才能保持数据；一旦断电，电容漏光，数据就没了。

说明

上述文本中的“内存”指的是“随机存取存储器”（RAM）。这里的“随机”指的是可以在任意时刻访问任意地址，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存在不同的含义，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。

内存的性能主要由容量和频率决定。容量指的是内存能够存储多少数据，单位是 GB（千兆字节）；频率指的是内存每秒钟能够传输多少数据，单位是 MHz（兆赫兹）。一般来说，容

量越大、频率越高，内存的性能就越强。

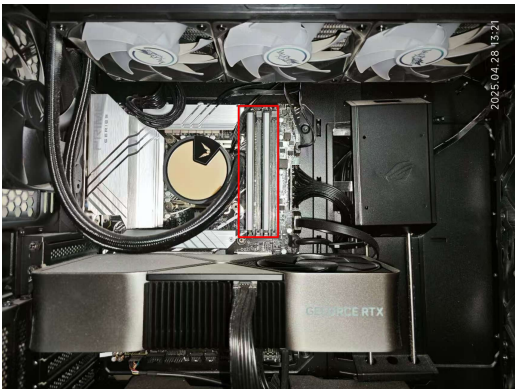


图 2.4: 内存条，这里是两条 DDR5，每条 16GB

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¹，除非花高价买高端的企业级 SSD，但仍需定期通电。

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。虽然和 SSD 都使用闪存颗粒，但是 SSD 通过主控优化、多通道技术等实现更高的性能，U 盘则只用于低成本的便携存储。
- 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，现在常用于单次写入的存档等。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（一次性投入几万）。其优点是容量大、寿命长，且磁带本身非常便宜，适合长期冷数据归档。
- 软盘：一种老古董，使用磁性材料存储数据。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。

说明

现代 Windows 系统的计算机中盘符默认从 C 开始而不是从 A 开始，正是因为 AB 盘符是给软驱用的；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。虽然现代的 Windows

¹个人使用寿命和 HDD 无明显差异，基本都能用到彻底换机

系统允许手动分配盘符（如将 C 盘强行分配盘符 A），但这样会导致系统不稳定，极不建议这么做。

注意

硬盘有价，数据无价。请务必定期备份数据，尤其是重要数据。

表 2.1: 不同设备的读写速度、存储容量和价格比较

设备	读写延迟	常见存储容量	价格（每 GB）
CPU Cache L1	1ns	几十 KB	买不到，折合至少 60 万元
CPU Cache L2	3-5ns	几百 KB-几 MB	
CPU Cache L3	10-20ns	几 MB-几十 MB	
DDR5 RAM	80-100ns	几十 GB	约 300 元
NVMe SSD	10-100μs	几百 GB-几 TB	不到 1 元
SATA SSD	100-200μs	几百 GB-几 TB	不到 0.5 元
HDD	平均 10ms	几百 GB-几十 TB	约 0.1 到 0.2 元
蓝光光盘	100ms	25GB-100GB	约 0.3 元
USB 3.2	1-10ms	几 GB-几 TB	约 0.5 元
磁带	秒，甚至分钟级	几 TB-几十 TB	约 0.05 元

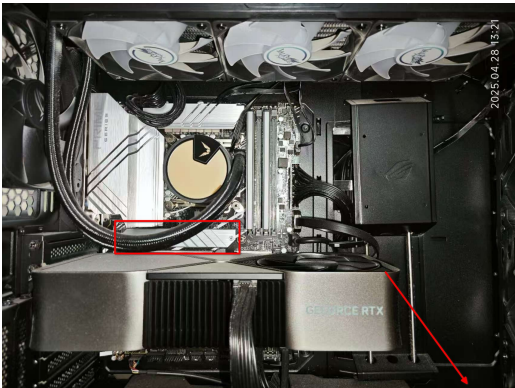


图 2.5: 这台计算机有一个 SSD（框选的），还有一个 HDD（箭头指向的，没有拍进来）

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，被称为“显存”。

对于现在 AI 时代而言，显卡因为有着良好的并行特性，成为了通用深度学习的主流硬件之一。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。而在一些特殊的计算任务上，FPGA 和 ASIC 等硬件则有着更好的表现，而部分嵌入式或边缘计算场景往往更偏好 NPU 或 TPU 等专用芯片。显卡的另外一个重要的性能指标是“显存”，即显卡自带的内存容量，显存越大，显卡能够处理的图形数据就越多。

提示

从上述例子中我们可以看到，CPU、GPU、TPU 等的通用性是依次降低的，而在特定任务上的性能则是依次提升的。这种“专用性换取性能”的设计思路，是计算机体系结构中的一个重要原则，其一个重要体现就是软件硬件化，这也是近些年来兴起的“硬件加速器”设计思路的基础。

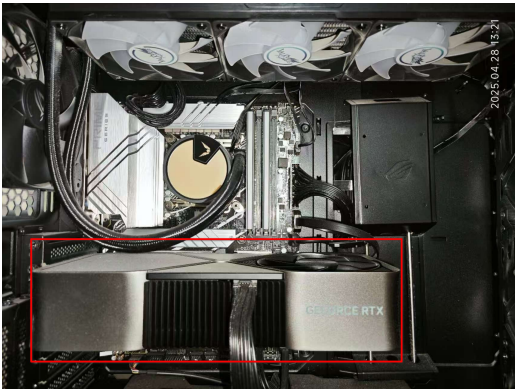


图 2.6: 显卡，这里是一块 NVIDIA 的 RTX 4080

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。在上述硬件插图中，整个绿色的电路板就是主板。

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。在上述硬件插图中，电源没有拍摄进来，但通常位于机箱的一角，通常是一个银色或黑色的盒状物体。

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拔插电缆，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。

2.2 买计算机的一些理论

2.2.1 获取机器的途径

一般情况下，我们有两种途径获取一台计算机：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；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。令人哭笑不得的一点是，虽然自己配置的组装机在价格上往往比同配置的整机更贵（硬件的零售价肯定比整机组装厂拿货的批发价要高），但如果一点组装机知识都不了解的话，即使是购买整机也大概率是会被商家狠狠“宰”的。

2.2.2 购买机器的原则

购买机器有两项不成文的原则。

- **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 CPU 性能差距有七倍。

2.2.3 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 游戏本通常配备了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，属于一种折中方案。

由于苹果系列产品的特殊性，这里单独介绍。笔者从未用过任何苹果系列产品，因此本段直接摘抄自 2024 年 LCPU 计算机导购手册，仅供参考。

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,计算机专业常用开发工具几乎都有 macOS 支持,电子专业则有部分 Windows 独占软件,不能运行在 MacBook 上,这建议提前确认。另一方面,有不少信科同学在使用 MacBook 配置编程环境时遇到了一些麻烦,所以一定要做好功课。

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。

2.2.4 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。最经典套路莫过于“i9 级”处理器等,采用这种称谓的基本可以认为是洋垃圾:如果真是 i9 这种先进处理器,为什么还要用这个模棱两可的“级”字呢?
- 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。对此,我们在到货后可以使用一些工具(如 AIDA64、CPU-Z 等)来检查电脑的具体配置,确保与商家所说的一致。
- 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。对此,我们建议在购买前先确认好库存情况,避免被套路;即使真出现了这种情况,我们不买就是了。

对此,我们建议尽可能地去官网或者大平台(京东自营、天猫旗舰店)购买,避免去小商家购买,谨防上当受骗。

2.3 整机（笔记本）

我们这里不考虑台式机整机的购买，仅讨论笔记本整机的购买。这里为大家推荐一个较为专业客观的公众号：**笔吧测评室**，该公众号会对市面上大部分主流笔记本进行测评，并给出购买建议，大家可以参考其内容来选择适合自己的笔记本电脑。

2.3.1 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、宏基、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、华硕、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。除此之外，苹果、荣耀两家的产品也相当值得考虑。
- **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。如果不介意折腾且运算有限，可以考虑这些品牌。

2.3.2 线上购买

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。从品控方面来看，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。

2.3.3 线下购买

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。

电脑城水深最深，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！**）。

2.4 组装机

组装机的购买非常复杂，不仅需要购买各个硬件部件，还需要进行组装和调试。对于没有经验的用户来说，组装机可能会遇到很多问题，因此我们建议新手用户尽量选择购买整机。但相当多情况下，笔者建议在宿舍装一台台式机，尤其是要进行机器学习、图形设计、视频编辑

等高性能计算任务或高性能游戏时，台式机的性价比和性能优势非常明显，使用体验也显著高于游戏本。

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：**需求优先；先定机箱体积，再买核心配件（CPU、主板、内存、显卡），再买其他设备。**

需求基本上分为以下四类：办公学习（按 5000 元预算）、全能甜点（按 8000 元预算）、游戏发烧（按 12000 预算）、生产力设备（按 20000 预算）。预算仅供参考，实际价格会根据当年推荐配置和市场行情有所浮动。低于 5000 元的配置不建议购买组装机，用这个钱购买整机需求肯定够了。

一般购买配件可以根据下面的简单指南参考进行购买。有时候有特别需求则另当别论，例如需要做数据处理的同学们应该需要巨大的内存等。一般的，我建议按以下顺序购买配件，但**不要只图便宜**，否则安装和使用的时候会很遭罪。还有，有部分部件（如内存）在台式机上和笔记本上并不通用，请务必注意区分。

2.4.1 机箱

虽然机箱连计算机的组成部分都不是（机箱确实不参与任何计算任务），但非常残酷的一点是，必须先按你桌子的空间大小选定机箱体积，否则买回来的机箱放不下就完蛋了。笔者购买计算机时就犯了这个毛病，最终被迫和机箱挤挤，于是一挤就是两年。

机箱的体积一般分为四种：全塔、中塔、MATX、ITX。其尺寸分别为：

- 全塔（Full Tower）：高度通常在 60 厘米以上，宽度和深度也较大，适合需要安装大量硬件和散热设备的用户。
- 中塔（Mid Tower）：高度通常在 45-60 厘米之间，宽度和深度适中，是最常见的机箱类型，适合大多数用户。
- MATX（Micro ATX）：高度通常在 35-45 厘米之间，宽度和深度较小，适合空间有限的用户，但扩展性较差。
- ITX（Mini ITX）：高度通常在 25-35 厘米之间，宽度和深度非常小，散热和风道都非常难做，非常不建议新手入坑。

实际机箱体积应查看具体型号的尺寸参数。

2.4.2 CPU 和主板

建议购买板 U 一体的 CPU 和主板套装，避免兼容性问题，还节省资金。

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。需要声明的是，**计算机性能瓶颈并不完全由 CPU 决定**，如果买了一个非常高端的 CPU，但是其他配件（如内存、显卡等）跟不上，那就堪比“吕布骑狗”，不能发挥出 CPU 的全部性能。当然反之也一样。

不买板 U 套装的情况下，可以买散片 CPU，比盒装 CPU 便宜得多。当然，散片风险也大一些。而这时候，则需要注意主板的 CPU 插槽：英特尔和 AMD 的 CPU 插槽不兼容，必须注意区分。

在购买主板的时候，需要注意区别以下内容：内存插槽（DDR4、DDR5）、主板尺寸（ATX、MATX、ITX）。此外，还需要注意主板的扩展性，例如是否有足够的 PCIe 插槽、M.2 插槽等。

2.4.3 显卡

显卡市场基本上是英伟达、AMD、Intel 三足鼎立。英伟达的性能和市场占有率目前来看依然最高，尤其在硬件光追、机器学习等领域，英伟达的显卡几乎是最好的选择。一般预算充足的情况下，选购英伟达的显卡是较为稳妥的选择。AMD 性价比高，但是硬件光追和生产力依然不如英伟达。Intel 在显卡方面是新兴厂家，剪视频很不错，且自从驱动稳定后主流 1080p/2K 游戏已能胜任，但光追与兼容性仍逊于 N 卡，尝鲜可入，求稳还是选英伟达或者 AMD。

显卡有公版和非公版之分。以英伟达为例，所有的英伟达系列显卡的芯片都是由英伟达设计、生产的，但是显卡的非核心部分可以由不同的厂商设计和生产。完全由英伟达制造的显卡被称为公版显卡，而华硕、技嘉、微星等厂商可以从英伟达购买芯片，然后设计自己的外观、散热、供电等，被称为非公版显卡。非公版显卡通常会在特定的方面比公版显卡更优化，但是版本也更多，仅微星一家就有 SUPRIM、GAMING TRIO、VENTUS 等系列。由于各种原因，我们基本上是很难买到公版显卡，而非公版显卡就成了一个不错的选择。

二手显卡风险巨大，非常不建议去趟浑水。一般认为，30 系早期批次基本上默认矿卡，后期改版矿卡风险降低，但是仍需仔细甄别；40 系无矿潮需求，矿卡概率低一些，但是也需要做好甄别。（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。

2.4.4 内存和外存

25 年 DDR5 是主流。一般盯着 32GB 6000MHz CL30 的套装买就可以了。高端的处理器可以冲 6800MHz，但是这同时依赖于主板和调参。

不建议购买 DDR4，目前很多新主板已经没有 DDR4 插槽了。

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）或者长期归档，否则不建议买，一定要买的话盯着 CMR 盘买，一般西数、希捷这两家就行。SMR 叠瓦盘别碰。

2.4.5 电源

在确定上述几个配件之后，选购电源。电源的瓦数是最重要参数，应至少是上述核心配件瓦数总和的 1.5 倍，再加上 100 瓦余量。出于节能、散热等方面考虑，尽可能购买金标以上的电源，且电源品牌一定要选大厂（海盗船、振华、鑫谷、安钛克、酷冷至尊等），稳定的电源能够有效保护计算机硬件的安全。另外，尽量买全模组电源，方便理线。

2.4.6 散热器

CPU 散热器主要取决于 CPU 的功耗，一般 CPU 越高端散热需求就越高。高端的 CPU 尽可能上水冷散热，中端可以塔式风冷，低端则可以使用盒装自带的散热器。当然如果你是散片 CPU 则不附带散热器，必须另行购买。**不能不购买散热器！也不能使用手机散热器替代计算机散热器！**

除了散热器，还有散热介质。散热介质一般涂抹或粘贴在 CPU 表面，保证 CPU 和散热器之间良好的热传导。常见的散热介质有硅脂、导热垫片、液态金属三种。高端硅脂大概 20 块钱一管，非常便宜，对于绝大多数人而言完全足够使用。导热垫片性能稍差，一般用于笔记本电脑。液态金属的导热性能最好，但是风险高、价格昂贵，且需要平放 CPU 才能使用，因此不建议小白使用。

涂抹硅脂非常简单，几乎“随便涂”。常见的涂抹方法有米粒法、十字法、涂满法等。不必追求涂抹的完美均匀，只要估计能够覆盖整个 CPU 表面就可以了，等到压上散热器的时候硅脂就会被自动挤平。不要用太多硅脂，防止压上散热器的时候溢出并流到主板上，谨防短路。**只要你把散热器从 CPU 上拿下来，就必须重新涂抹散热介质！**

2.4.7 显示器

显示器**对使用者而言**是整个计算机中最重要的部分之一，毕竟这玩意是我们天天盯着看的东西，真正直接影响所有使用体验：一个糟糕的显示器完全能够抵消一块 4090 带来的快乐！所以我非常非常建议大家不要在显示器上过于节省预算。

显示器主要有以下几个参数需要注意：

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。具体的公式这里不赘述了，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。

工具: <https://config.net.cn/tools/PixelToPpi.html>

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。

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（LCD）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”不等于 IPS，可能是廉价 AHVA。这是老生常谈的问题了，建议避坑。

VA 曲面屏 27 英寸以下意义不大，32 英寸以上才显沉浸。

OLED 面板长期显示静态内容容易烧屏，建议隐藏任务栏 + 黑色壁纸。新型 OLED 面板通过像素刷新技术大幅降低了烧屏风险，但仍然需要注意，建议启动系统的屏保功能等。

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。

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。

2.4.8 其他配件

其他配件主要包括键盘、鼠标、音箱等。键盘和鼠标的选择主要取决于个人喜好和使用习惯, 可以根据自己的需求选择合适的键盘和鼠标。音箱则可以根据自己的预算和需求选择合适的音箱。笔者并不是这方面的专家, 建议大家参考一些专业的评测和推荐。

很多人喜欢使用机械键盘, 机械键盘有着良好的手感和耐用性, 但价格昂贵、噪音较大, 在宿舍使用这个可能会影响他人休息, 这是需要注意的地方。

2.5 验机

不论是整机还是组装机, 收到 (组装完) 机器都要进行验机, 确保机器没有问题再开始使用, 如有问题则应尽快联系商家进行退换。

2.5.1 通电之前

在这段时间中, 我们应当检查包装盒是否完好无损, 是否有明显的撞击痕迹等。然后, 把计算机拿到手中, 检查机身是否完好、是否有磕碰、划痕, 配件 (如电源适配器、数据线等) 是否齐全。如果是请别人代为组装的组装机, 还需要检查各个硬件部件是否安装牢固、连接正确。

如果发现有明显的损坏痕迹或缺件, 应联系商家进行处理。建议拍摄一段开箱视频, 确保在出现问题时有证据证明机器在运输过程中没有被损坏。尤其是二手交易, 更要拍摄清晰的开箱视频, 避免日后纠纷。

2.5.2 通电

在确认机器外观完好无损之后, 我们可以进行通电测试。插上电源, 检查电源适配器是否工作、电脑是否上电。如果是笔记本电脑, 还需要检查电池是否充电正常。

然后开机。不要先进系统, 一定要先进入 BIOS/UEFI 界面, 检查 CPU、内存、硬盘等硬件信息是否正确, 确认没有硬件故障; 再检查风扇转速、温度等参数, 确保散热系统工作正常。最后, 检查硬盘通电次数和电池循环次数, 一般小于 10 次可以接受, 过高则可能是翻新机。如

是二手交易则无需检查硬盘通电次数和电池循环次数。进入上述界面的方法因主板不同而有所区别，不过大多数主板都是 F2 或 Delete 键，具体参见主板或机器说明书，也可以留意开机时的提示信息。

2.5.3 进系统

下一步，进入系统。在这个过程中务必不要连接网络，防止系统驱动更新、激活等操作影响退换货。Windows²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。而在最新的 Windows³系统中，已经不支持上述本地账户登录手段了。对此，只能退而求其次，使用 WinPE 等工具临时验机。

现在，我们可以使用一些工具来检查硬件信息，例如 CPU-Z、GPU-Z、CrystalDiskInfo 等，确保硬件信息与商家所说的一致。然后，可以使用一些压力测试工具来测试 CPU、内存、显卡等硬件的稳定性，例如 Prime95、MemTest86、FurMark 等，确保硬件没有问题。这个也可以使用“图吧工具箱”。

然后验证屏幕，检查坏点、漏光、刷新率、色域等参数，确保屏幕没有问题。具体方法如下：

- 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。

注意

在创建 Windows 账户的时候，本地账户和微软远程账户没有什么过大的区别。微软账户会得到一些额外的服务，例如云服务等，但是这也可能面临一些隐私问题；本地账户则完全在本地使用，隐私性更好一些。

但是有一点是确定的：在创建账户的时候，用户名一定要使用英文和数字，不能使用中文，否则可能会导致某些软件无法正常安装和运行，例如 GCC 等，这会在之后带来很多不必要的麻烦。

另外，保修政策要看清：

- 全国联保 ≠ 全球联保。留学生买美版 ThinkPad 回国，坏了要送香港修。
- 上门服务 ≠ 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。

²主要是 Win11。对 Win10 的支持于 2025 年 10 月 14 日终止。

³指的是 Windows 11 Beta Build 26120.6772 版和 Windows 11 Dev Build 26220.6772 版及其以后的版本。

- 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。

最后，建议大家在验机完成、发现问题之后，尽快联系商家进行退换货处理，避免超过退换货期限。一般情况下，商家会提供 7 天无理由退货和 15 天换货的服务。

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。学校会定期组织计算机维修和养护的活动，同学们可以多多参加。

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。

第三章 系统安装、基础配置和软件生态

软件指的是计算机中运行的程序和数据集合，是计算机系统中不可或缺的一部分。没有软件，计算机硬件将无法发挥其功能。

在获得一台计算机之后，我们需要安装操作系统（Operating System, OS）才能使用它。操作系统是管理计算机硬件和软件资源的核心软件，它与其他软件提供了运行环境和接口。没有操作系统，用户无法与计算机进行交互，其他软件也无法运行。因此，安装一个合适的操作系统是使用计算机的第一步，只有安装了操作系统，我们才能使用它。常见的操作系统有 Windows、macOS 和各种 Linux 发行版。选择合适的操作系统取决于你的需求和硬件兼容性。我们也需要驱动软件来确保计算机的各个硬件组件能够正常工作，以及应用软件来实现具体的功能。

如你买的是有操作系统的计算机，则跳过本章前半部分，直接从系统基础配置章节开始即可；如你买的是裸机（裸机即没有预装操作系统的计算机），则需要按照以下步骤安装操作系统。如希望重装系统，也可以参考以下步骤进行操作，但请注意备份重要数据。

本章中如果有任何你不理解的指令，**请先照做**，不要自作主张更改步骤，以免导致之后不必要的麻烦。随着你对计算机的了解加深，你会逐渐理解为什么让你这么做。

3.1 操作系统安装

3.1.1 选择操作系统

操作系统是计算机的大脑皮层，它负责管理计算机的硬件和软件资源，有着最高的权限。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  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，但是缺点是在多数开发场景中的配置非常复杂，但是通过 WSL2 等工具也可以弥补一部分，且对于游戏开发等场景 Windows 仍是首选（主要是因为新游戏几乎都需要先适配 Windows）。另一方面，Windows 的安全性相对较低，容易受到病毒和恶意软件的攻击。

macOS  苹果公司开发的操作系统，专门用于苹果的计算机产品。macOS 以其优雅的界面和强大的功能而闻名，同时安全性相当高。缺点也很明显，macOS 的硬件和软件生态系统相对封闭，且只能在苹果的硬件上运行，因此价格较高。

Linux  一个开源的操作系统，它是一个类 UNIX 操作系统。其学习曲线陡峭，至今在传统个

人计算机市场的占有率仍然远低于 Windows 和 macOS，但在服务器和嵌入式系统使用一直占据主导地位。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。

几个系统的比较见下表：

操作系统	Windows	macOS	Linux
使用难度	简单	简单	较复杂
价格	收费	收费	免费
硬件兼容性	高	低	中高到极高
软件生态	丰富	不太丰富	丰富
安全性	较低	高	高
系统级可定制性	较低	较低	高
社区支持	强	一般	强
适用场景	办公、游戏	设计、开发	服务器、开发

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。

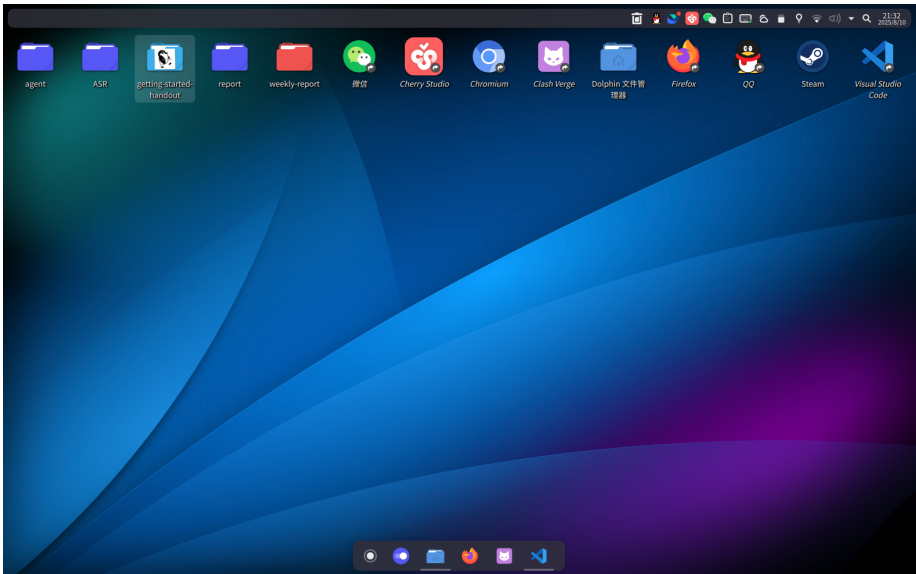


图 3.1: 一台假装自己是 macOS 的 Arch Linux 机器

3.1.2 准备工作

在安装操作系统之前，我们要准备以下内容：

- 另一台计算机（用于下载操作系统镜像文件和制作启动盘）
- 一个足够容量的 U 盘（通常 8GB 及以上）
- 操作系统镜像文件（ISO 格式）
- Rufus、Ventoy 等启动盘制作工具

操作系统镜像文件可以去各大操作系统官网下载安装。例如，Windows 可以从微软官网下载，Ubuntu 等 Linux 发行版可以从其官方网站下载。如公司或学校提供了特供版（如 PKU 有 Windows 各版本的镜像），则使用这些特供版本较好，不必担心激活等问题（正版 Windows 需付费，盗版 Windows 可能存在安全隐患，但上述特供版是正版授权、无需额外付费的）。

Rufus、Ventoy 等启动盘制作工具可以从其官方网站下载。Rufus 适合制作单一操作系统的启动盘，而 Ventoy 则支持在同一 U 盘中放置多个操作系统镜像，启动时选择需要安装的系统。这些启动盘制作工具有一个就行。

3.1.3 制作启动盘

以 Rufus 为例。Ventoy 的使用方法类似，可以参考其官网说明。

1. 将 U 盘插入上述“另一台计算机”。
2. 打开 Rufus 软件，选择你的 U 盘作为“设备”。
3. 点击“选择”按钮，选择下载好的操作系统镜像文件（ISO 格式）。
4. 保持其他选项默认，点击“开始”按钮，等待制作完成。

在这个过程中，U 盘上的数据会被清除，请确保 U 盘中没有重要数据。

3.1.4 调整 BIOS 设置

现在，我们需要进入待安装系统的计算机（以下简称“目标计算机”）的 BIOS 设置，以便从 U 盘启动。首先，插上上述制作好的启动盘 U 盘。

打开你的目标计算机，并在启动时按下特定的按键，这个按键随着不同的主板而不同，常见的有 F2、F10、Delete 等。具体按键可以参考主板说明书或网上搜索你的主板型号加上“进入 BIOS 按键”。通常来说，不停地反复按上述按键总能蒙对。

然后，关闭“安全启动”（Secure Boot）选项。这个选项通常在“Boot”或“Security”标签下。关闭后，找到“Boot Priority”或“Boot Order”选项，将 U 盘设置为第一启动项。保存设置并退出 BIOS，重新启动计算机（不要拔掉 U 盘）。

3.1.5 安装操作系统

安装操作系统的过程目前已经相当视窗化、自动化，只要我们安装的不是类似 Arch Linux 这种非常折腾的系统，一般只需要按照屏幕上的提示操作即可。

整个安装过程大概包括以下几个步骤：

1. 选择语言、时间和键盘布局。

2. 选择安装类型（通常选择“自定义安装”或“全新安装”）。
3. 分区（如果是新硬盘，可以选择自动分区；如果是已有数据的硬盘，建议先备份数据，然后删除所有分区，重新分区）。
4. 选择安装位置（选择刚才分区的主分区）。
5. 输入用户信息（用户名、密码等）。
6. 等待安装完成，期间计算机会自动重启几次。

在现代计算机中，如果使用的是 SSD，则不要对硬盘分很多个区，一般分 1 个主要分区即可（EFI、系统保留分区等不算在内）。如果使用的是 HDD，则建议分多个区，例如一个系统区（C 盘，Win11 下不低于 200GB）和一个数据区（D 盘），以便日后管理数据。现代计算机大多是 SSD，HDD 已经较少见。

在上述输入用户信息的步骤中，请务必牢记：**请设置不含空格的英文用户名**，否则会导致后续使用中出现问题。笔者推荐设置密码，以防止他人未经授权使用你的计算机。

3.2 系统基础配置

3.2.1 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和更新时自动安装和更新。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，所以一定要安装齐全，否则可能会导致硬件无法正常工作。

如果是整机，厂商一般会提供一个软件，可以自动检测并安装所需驱动程序。请务必运行该软件，确保所有驱动程序均已安装。如果是裸机（例如组装机），请前往各硬件厂商官网，下载并安装主板、显卡、声卡等硬件的驱动程序。网卡驱动程序比较麻烦，如果发现计算机启动后无法联网，请先使用另一台计算机下载网卡驱动程序，拷贝到 U 盘，再插入目标计算机安装。现在的驱动程序往往是自解压的，只需要双击这些文件即可简单的安装。

我们不推荐使用诸如驱动精灵等第三方驱动安装软件。

安装完毕驱动程序后，建议重启计算机以确保驱动程序生效。

3.2.2 安装常用应用软件

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、Edge 等，都是应用软件。应用软件在日常生活中常常被简称为“软件”。

根据你的需求，安装一些常用软件。例如：

- 浏览器（如 Google Chrome、Mozilla Firefox）
- 办公软件（如 Microsoft Office、WPS）
- 压缩软件（如 7-Zip）
- 媒体播放器（如 VLC Media Player）

现代计算机的安全性相当高，Windows 自带的 Windows Defender 防病毒软件已经足够应付大部分威胁，如果我们不经常下载来路不明的软件或访问可疑网站，一般不需要额外安装第三方杀毒软件。而 Linux、macOS 系统则因为其高安全性，一般也不需要安装杀毒软件。杀毒软件会导致系统变慢，且可能与其他软件冲突，除非真的有必要，否则不建议安装。

我们将以 MS Office 软件为例，介绍如何在 Windows 系统中安装常用软件。Office 是一个非常常用的办公软件套装，包含 Word、Excel、PowerPoint 等组件，但需要付费购买正版授权才能使用全部功能。整机大多预装了 Office 且已经激活（这些都算在计算机的售价里了），但裸机则需要自行安装和激活 Office。

1. 打开浏览器，访问 Microsoft Office 官网，下载 Office 安装程序。PKU 提供了正版 Office 的下载和激活方式，这里可以前往[北京大学正版软件](#)网站获取（需要北京大学的统一身份认证账号）。
2. 运行下载的安装程序，按照提示完成安装。PKU 提供的是一个 ISO 镜像，我们需要右键该文件，选择“装载”，然后运行其中的安装程序，完成安装。
3. 安装完成后，打开任意 Office 组件（如 Word），根据提示输入激活密钥进行激活。如果是 PKU 提供的 Office，则可以按照网站上的说明进行激活，在笔者的印象中是下载一个小工具，在校园网环境下运行该工具即可完成激活。
4. 现在你可以开始使用 Office 了。

有些软件在安装时可能会允许你将该软件加入到系统的 PATH 环境变量中，我非常建议大家这么做。

提示

“环境变量”可以简单地理解为“让计算机知道这玩意在哪里”。这个问题暂时略过，之后的章节会有讨论。

3.2.3 Linux 和 mac 上的软件安装

在 Linux 和 Mac OS 上一般不建议使用安装包来安装软件，这是因为可能导致依赖问题。Linux 和 Mac OS 都有自己的包管理器，建议使用包管理器来安装软件。

例如，在 arch linux 下，我们可以使用 pacman 包管理器来安装软件，例如安装 git：

```
1 sudo pacman -S git
```

不懂命令可以先不去理解这个，本节其他命令也是如此。

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。

比方说，在 Windows 上想要安装 git，我们可以使用 winget 来安装：

```
1 winget install Microsoft.Git
```

这条命令会自动下载并安装 git，并且将其添加到系统的 PATH 环境变量中，方便我们在命令行中使用 git 命令，省去了在网上查找安装包、下载安装包、运行安装包等繁琐的步骤。

不会用终端？没关系，后面讲了怎么用这玩意。

但也有例外，例如 miniconda 等软件，官方推荐的安装方式就是下载安装包¹。

而 mac 有着自己的 app store，可以直接在 app store 中搜索并安装软件。笔者本人从没用过苹果系列产品，因此对其细节完全不清楚，建议参考网上的相关教程。

3.2.4 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。
- Motrix：一个免费且开源的下载软件，支持 HTTP、FTP、磁力链接和 BT 下载。
- wget：一个老牌、免费的命令行下载工具，支持 HTTP、HTTPS 和 FTP 下载。它可以通过命令行参数来控制下载行为，适合有一定技术基础的用户使用。具体使用见 4.5。
- Aria2：一个免费的命令行下载工具，支持 HTTP、FTP、磁力链接和 BT 下载。它可以通过命令行参数来控制下载行为，适合有一定技术基础的用户使用。

- 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](https://github.com/zhuozhiyongde/PKU-Art)，它可以给你一个风格现代、足够好看的教学网。
- Zen Browser：基于 Firefox 的轻量浏览器，界面更加简洁美观，同时支持自定义主题和插件。

¹实际上得到的是一个 .sh 脚本)，然后运行安装脚本进行安装

- 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。
- EarTrumpet: 应用级音量混音与快捷控制。

- 杀毒软件类 (Mac 和 Linux 因为其高安全性, 通常不需要安装杀毒软件)

- Windows Defender: Windows 系统自带的杀毒软件, 功能强大, 查杀率接近 100%, 已经和老牌专业杀软 (卡巴斯基、BitDefender 等) 不相上下, 能够有效地保护常规情况下计算机免受病毒和恶意软件的侵害。但是误报率较高, 可能会误报一些正常的软件为病毒。
- 火绒: 一个免费的国产杀毒软件, 误报率很低, 界面友好, 适合普通用户使用。然而, 火绒的杀毒能力要低一些。

- 输入类

- Rime: 一个高度可定制的输入法框架, 可以自定义输入方式与习惯。
- InputTip/ImTip: 输入状态提示, 可以显示当前处于中、英文状态等, 前者也可以根据不同输入状态更换鼠标样式。
- Keyviz/KeyCastOW: 键鼠操作可视化, 会在屏幕中显示按键操作, 方便用户查看和学习。
- ContextMenuManager: 可视化的右键菜单管理器, 通过图形界面操作注册表来编辑和管理右键菜单, 避免菜单栏过长。

- 文件搜索与窗口管理

- Everything/Everything Toolbar: 一个免费的文件搜索工具, 能够快速地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, 后者是任务栏集成的版本, 操作更加方便。
- Fences/Coodesker: 桌面整理工具, 可以按照文件类型进行整理。
- Deskpins: 任意窗口置顶, 操作十分简单。
- AltSnap: 按住 Alt 可在任意位置拖动窗口。
- Groupy: 任意窗口的分组管理, 可以将不同的窗口在上方显示与浏览器类似的标签条, 方便窗口管理。
- Office Tab: Office 三件套的窗口管理, 在工具栏下方显示所有打开的文档, 方便切换与关闭。

- OneCommander/Directory Opus: 更高级的文件管理, 具有批量命名、批量复制、批量移动等功能, 操作更加方便。
- 效率与自动化
 - uTools: 极简的多功能应用, 具有截图翻译、提取颜色、转换图片格式等插件, 也有插件市场可以选择。
 - AutoHotkey: 一个强大的自动化脚本工具, 可以很方便地完成热键修改、文本替换、自定义宏等操作。
 - PowerToys: Windows 官方效率工具合集, 包括保持唤醒、颜色选择器、键盘管理器以及鼠标工具等工具。
- 其他
 - Wallpaper Engine: 一个收费的动态壁纸软件, 能够让你的桌面变得更加美观。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。
 - Ventoy: 一个开源的 u 盘启动工具, 能够使多个 ISO 镜像共存于 U 盘, 而不必格式化 U 盘, 选择从其中的一个镜像启动。它能使多个镜像文件和 U 盘其他文件共存, 是装机盘和资料盘合一的好工具。
 - 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。
 - Zotero: 一个免费的文献管理软件, 能够帮助你管理和组织你的文献资料。它支持多种文献格式, 包括 PDF、Word 等, 并且可以与浏览器集成, 方便地从网页上导入文献资料。它也能兼任 PDF 阅读器的职责。
 - Foxit Reader: 一个免费的 PDF 阅读器, 功能强大, 界面友好。
 - TrafficMonitor: 网速/CPU/内存悬浮窗, 也可以嵌入任务栏中实时显示内存与 CPU 运行情况。

3.2.5 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。

另，用包管理器安装的软件，最好也用包管理器卸载。例如上文提到的 `winget` 安装 `git`，我们可以使用以下命令来卸载 `git`：

```
1 winget uninstall Microsoft.Git
```

而在 Linux 上则类似的用包管理器卸载，例如在 Arch Linux 上卸载 GCC：

```
1 sudo pacman -R gcc
```

进阶：SHA256 校验

SHA256 校验是一种常用的文件完整性校验方法。它可以帮助我们验证下载的软件包是否被篡改或者损坏。通常情况下，软件的官方网站会提供一个 SHA256 校验值，我们可以使用 SHA256 校验工具来计算下载的软件包的 SHA256 值，然后将其与官方网站提供的 SHA256 值进行比较。如果两个值相同，则说明下载的软件包是完整的，没有被篡改或者损坏；如果两个值不同，则说明下载的软件包可能被篡改或者损坏，建议重新下载。

常见的 SHA256 校验工具有很多，例如 Windows 自带的 `CertUtil` 工具、Linux 和 macOS 自带的 `sha256sum` 工具等。使用这些工具非常简单，只需要在命令行中输入相应的命令即可。例如，在 Windows 中，我们可以使用以下命令来计算文件的 SHA256 值：

```
1 certutil -hashfile path\to\file SHA256
```

你需要把 `path\to\file` 替换成你要计算 SHA256 值的文件的路径。在 Linux 和 macOS 中，我们可以使用以下命令来计算文件的 SHA256 值：

```
1 sha256sum path/to/file
```

当然，如果从官网下载，则基本上不会有问题，毕竟官网大概率是不会篡改自己的软件的。但如果真的从其他网站下载，则建议进行 SHA256 校验。但如果你搞了个盗版软件，那校验就多余了（因为肯定篡改了），这也是盗版软件的风险所在（谁知道在破解的同时有没有给你塞点别的东西进去）。

3.2.6 进阶：利用任务管理器监测和管理进程

“进程”，可以通俗的理解为“正在运行的软件”。有些软件在运行时会占用较多的系统资源，导致计算机变慢或者卡顿。我们可以使用任务管理器来监测和管理进程。

在 Windows 中，我们可以按下 `Ctrl+Shift+Esc`² 或者 `Ctrl+Alt+Del`，然后选择“任务管理器”来打开任务管理器。在任务管理器中，我们可以看到当前正在运行的进程，以及它们占用的系统资源（CPU、内存、磁盘等）。如果发现某个进程占用过多的系统资源，我们可以选择该进程，然后点击“结束任务”来终止该进程。

任务管理器还可以监测计算机的运行情况，例如 CPU 使用率、内存使用率、磁盘使用率等。这一点非常实用，例如在机器学习中，我们可以通过任务管理器来监测 GPU 的使用情况，从而了解代码写没写错：如果学习速度特别慢，GPU 使用率特别低，那么很可能是代码写错了，导致 GPU 没有被充分利用。

3.3 进阶：安装 Arch Linux

对于希望深入学习计算机的同学，我们推荐使用 Arch Linux 作为操作系统。Arch Linux 是一个轻量级的 Linux 发行版，具有高度的可定制性和灵活性。安装 Arch Linux 需要一定的 Linux 基础知识和命令行操作能力，但通过安装过程，你可以深入了解 Linux 系统的工作原理和配置方法。

安装 Arch 需要对 Linux 系统有相当的了解，否则完全无法理解安装过程中的每一步骤；安装该系统也是一个非常折腾的过程，如果你不是很熟悉 Linux 系统或者不爱折腾，不必阅读这一节，直接跳过即可。

3.3.1 前置操作

在安装 archlinux 之前，我们首先要做一些前置的工作。我们需要一个 U 盘和一个 archlinux 的 iso 映像，并使用 Rufus 等工具将 iso 映像烧录到 U 盘中；另一方面，我们在安装整个系统的时候需要保证机器一直联网。

之后，在关机状态下，插上 U 盘，进入你计算机的 BIOS 环境，并选择你的启动方式为“从 U 盘启动”、关闭安全启动、调整启动模式为 UEFI。此三者缺一不可。另外，请确保你的计算机一直有网络连接；如果使用无线网络，务必保证你的无线网络名称和密码均不含特殊字符（如汉字）。

说明

有少数奇葩的主板里面，安全启动⁴被设置为开启，却不存在关闭它的选项，但系统主板内置有 Windows 系统的公钥证书签名，使其只能加载 Windows，其它系统（包括 archlinux）一律不予加载。用户不能关闭它，还无法换系统，实在让人无语。如果你正好是这样的电脑，不妨在虚拟机里尝试下 archlinux 吧！

⁴安全启动指的是主板在这种情况下只信任微软签名的 bootloader。Arch 自带的 bootloader 没有微软签名，因此会被拒绝执行。

²这三个键得一起按，下同。

3.3.2 开始安装

进入安装环境

在跳出的选项框中，选择第一项，进入安装环境。之后，该安装环境就会自动给你加载一些内容。不需要管这些内容具体是什么，一路确认到命令行界面，此时你的用户是 `root`，系统是 `archiso`，终端是 `zsh`。从这一步开始，到安装完成为止，你的这个 U 盘就一定要一直插在电脑上。

禁用 reflector 服务

这个服务主要是用于自行更新 `mirrorlist` 的。`mirrorlist` 是软件包管理器 `pacman` 的软件下载渠道；也许它是一个很好的工具，但是在国内的特殊网络环境下，这个东西反而成了累赘，不妨禁用之。因此，这个东西一定要在联网之前搞。

```
1 systemctl stop reflector.service # 禁用 reflector 服务
```

联网

我们使用 `iwctl` 来联网：

```
1 iwctl # 进入交互式命令行
2 device list # 列出无线网卡设备名，比如无线网卡看到叫 wlan0
3 station wlan0 scan # 扫描网络
4 station wlan0 get-networks # 列出所有 wifi 网络
5 station wlan0 connect wifi-name # 进行连接，注意这里无法输入中文。回车后输入密码即可
6 exit # 连接成功后退出
```

可以使用 `ping` 等工具来检查是否联网了。在 Linux 下 `ping` 必须按下 `Ctrl+C` 终止输出。

同步时间

我们使用 `timedatectl` 来同步系统的时间。这一步是必要的，这是因为 Linux 很多加密校验（HTTPS、GPG）依赖正确时间。如果时间差太多，证书会被判定过期。

```
1 timedatectl set-ntp true
```

检查是不是国内源

```
1 vim /etc/pacman.d/mirrorlist
```

检查有没有熟悉的 `pku.edu.cn` 和隔壁镜像。如果没有，说明你的 `reflector` 服务禁用晚了，不过并非不能解决，只需要在开头加上相关镜像就行了。不要在这一步添加社区源（例如 `archlinuxcn`）。

分区与格式化

这两个操作对数据很危险！不要把含有重要数据的盘当作目标盘。

lsblk 命令可以帮助我们确定我们要把 archlinux 安装在哪里。一般有两种硬盘编号，要么是走 SATA 协议的 sdx，其中 x 是字母；要么是走 NVME 协议的 nvmexn1，其中 x 是数字。我们可以通过观察磁盘的大小、已存在的分区情况等判断。下文统一使用 sda 作为磁盘编号，请根据你自己的实际情况更改磁盘编号。

```
1 cfdisk /dev/sda
```

我们要分出三个区：EFI 用来启动（如果做双系统时已有一个 EFI 分区，则无需）；Swap 用于临时存储（至少给到你物理内存的 60% 以上）、不活跃页交换和休眠；文件分区（使用 Btrfs 文件系统，不需要多个文件分区了）。

先创建 Swap 分区：选中 FreeSpace，再选中操作 New，再按回车，这样就能创建一个新的分区了。在按下回车后会提示输入分区大小，我们正常输入就可以了；单位可以自行输入。之后在新创建的分区上选中操作 Type 并按下回车，选择 Linux Swap 项目，按下回车以修改分区为 swap 格式。

再创建一个分区，操作类似之前的，只不过这次需要的分区格式是 Linux File System。最后，应用分区表的修改。选中操作 Write，并回车，输入 yes，再回车，确认分区操作。分区完成后，可以再使用 lsblk 命令复查分区情况。

现在，我们需要格式化各种分区。我们假设 EFI 分区是 sda1，Swap 分区是 sda2，Btrfs 分区是 sda3。

```
1 mkfs.fat -F32 /dev/sda1
2 mkswap /dev/sda2
3 mkfs.btrfs -L myArch /dev/sda3 # -L 操作是指定盘符用的
4 mount -t btrfs -o compress=zstd /dev/sda3 /mnt # 挂载分区
5 btrfs subvolume create /mnt/@ # 创建 / 目录子卷
6 btrfs subvolume create /mnt/@home # 创建 /home 目录子卷
7 umount /mnt # 卸载分区以便于之后的挂载操作
```

挂载分区

挂载分区有顺序性，需要从根目录开始挂载：

```
1 mount -t btrfs -o subvol=@,compress=zstd /dev/sda3 /mnt # 挂载 / 目录
2 mkdir /mnt/home # 创建 /home 目录
3 mount -t btrfs -o subvol=@home,compress=zstd /dev/sda3 /mnt/home # 挂载 /home 目录
4 mkdir -p /mnt/boot # 创建 /boot 目录
5 mount /dev/sda1 /mnt/boot # 挂载 /boot 目录
6 swapon /dev/sda2 # 挂载交换分区
```

用 df -h 命令和 free -h 来复查挂载情况。

安装系统

现在终于到了最重要的一步：安装系统了。我们使用 `pacstrap` 来安装最基础的包和功能性软件。

```
1 pacstrap /mnt base base-devel linux linux-firmware btrfs-progs
2 pacstrap /mnt networkmanager vim sudo zsh zsh-completions # zsh 也可以换成 bash，但
   ↳ 是不建议新手换这个。
```

倘若提示 GPG 证书错误，用以下命令更新一下密钥环：

```
1 pacman -S archlinux-keyring
```

然后经过一系列安装时信息的刷屏，就安装好了。之后，我们利用 `genfstab` 命令来根据当前挂载情况生成并写入 `fstab` 文件³即可。

```
1 genfstab
```

换根，以及一些基础设置

接下来，我们需要从安装介质中切出，进入新系统的目录下。

```
1 arch-chroot /mnt
```

现在可以发现命令行的提示符颜色和样式发生了改变。我们现在可以设置主机名和时区了：

```
1 vim /etc/hostname
```

输入你喜欢的主机名称，当然这里也不要包含特殊字符以及空格。

下一步，设置 `/etc/hosts`：

```
1 vim /etc/hosts
```

保证里面有以下内容：

```
1 127.0.0.1    localhost
2 ::1         localhost
3 127.0.1.1    myarch.localdomain myarch
```

³该文件用来定义磁盘分区。它是 Linux 系统中重要的文件之一。

再下一步，设置时区和硬件时间：

```
1 ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
2 hwclock --systohc
```

这里没有北京，只有上海，所以不要傻傻的找北京了！

使用 vim 或者 nano 编辑/etc/locale.gen, 去掉 en_US.UTF-8 UTF-8 以及 zh_CN.UTF-8 UTF-8 行前的注释符号，并保存。之后用 locale-gen 命令来生成 locale⁴。

```
1 locale-gen
```

下一步运行以下命令来设置默认 locale：

```
1 echo 'LANG=en_US.UTF-8' > /etc/locale.conf
```

我们不建议在这一步设置任何中文的 locale，会导致 tty 乱码。

现在为 root 用户设置密码：

```
1 passwd root
```

根据提示操作即可。注意输入密码时不会显示，不要以为键盘坏了。

最后，安装 CPU 微码：

```
1 pacman -S intel-ucode # Intel
2 pacman -S amd-ucode # AMD
```

CPU 微码是厂商发布的 CPU 补丁，它们在启动早期加载，使用软件来修复硬件缺陷。

作引导

引导是让主板和系统内核沟通的桥梁，系统的启动依赖于引导。

第一步，装包：

```
1 pacman -S grub efibootmgr os-prober
```

os-prober 是为了能够引导 Windows 系列系统而不得不装的一个东西。如果不需要 Windows 系统，完全可以不安装之。但是，前两个还是要装的。

下一步，把 grub 安装到 EFI 分区：

```
1 grub-install --target=x86_64-efi --efi-directory=/boot --bootloader-id=ARCH
```

⁴这个文件决定了软件使用的语言、书写习惯、字符集等

然后，对开机指令进行一些微调，以加快速度：

```
1 vim /etc/default/grub
```

主要是对 GRUB_CMDLINE_LINUX_DEFAULT 进行修改：去掉最后的 quiet 参数（这样可以在启动的时候就把内核日志打出来，便于排错）；把 loglevel 的数值从 3 改成 5，方便排错；加入 nowatchdog 参数，这可以显著提高开关机速度。

如果需要引导 Windows 系列系统，则不得不添加新的一行：

```
1 GRUB_DISABLE_OS_PROBER=false
```

最后，生成配置文件：

```
1 grub-mkconfig -o /boot/grub/grub.cfg
```

完成基础安装

输入以下命令以完成安装：

```
1 exit # 退回安装环境
2 umount -R /mnt # 卸载新分区
3 reboot # 重启
```

计算机关闭后，立刻拔掉 U 盘，进入引导界面，然后选择 archlinux。

登录系统需要输入用户名和密码。在这时，我们还没有创建任何账户，因此只有一个 root。输入用户名 root，以及你的密码，即可进入系统。

为了保证这玩意能够自动联网，可以使用

```
1 systemctl enable --now NetworkManager # 设置开机自启并立即启动 NetworkManager 服务
2 nmcli dev wifi list # 显示附近的 Wi-Fi 网络
3 nmcli dev wifi connect "<Your_Wifi>" password "<your_password>" # 连接指定的无线网络
↪ 络
4 ping 8.8.8.8 # 测试网络连接
```

最后，安装并运行 fastfetch：

```
1 pacman -S fastfetch
2 fastfetch
```

看着显示出的 Arch 徽标，我们终于可以长舒一口气：安装 Arch Linux 的过程终于结束了。当然，这个系统肯定很难日常使用，还需要一些后续配置，例如安装视窗等。之后的各种配置实际上都是在已经有的内容上继续开枝散叶，和现代 Windows 有显著的不同：现代 Windows 的视窗实际上已经紧紧地和系统内核绑定在一起了，而 Linux 的视窗只是个软件罢了！

创建非根用户

根用户的权限太高了，他本身就是系统。这导致其自由度太高、安全度太低，几乎毫无容错。因此，有必要创建一个非根用户。

先做一点准备工作：使用 `vim` 或者 `nano` 编辑一下 `/.bash_profile` 文件：

```
1 vim ~/.bash_profile
```

向其中加入以下内容：

```
1 export EDITOR='vim'
```

这样就会显式地制定编辑器为 `vim`，保证部分情况下不会出错。

然后就可以添加用户了：

```
1 useradd -m -G wheel -s /bin/bash myusername
```

你可以把 `myusername` 改为你喜欢的名字，但是同样不能包含空格和特殊字符。这个 `wheel` 是一个特殊的用户组，可以使用 `sudo` 提权。你可以使用以下命令设置新用户的密码：

```
1 passwd myusername
```

再下一步，编辑 `sudoers` 文件：

```
1 EDITOR=vim visudo # 这里需要显式的指定编辑器，因为上面的环境变量还未生效
```

找到这一行，把前面的注释符号 `#` 去掉：

```
1 #%wheel ALL=(ALL:ALL) ALL
```

保存并退出就可以了。现在你就有了一个非根用户。

开启多个库的支持

编辑这个文件：

```
1 vim /etc/pacman.conf
```

然后去掉 `[multilib]` 一节中所有内容的注释即可。这样可以开启 32 位库的支持。

然后在文档结尾处加入下面的文字来添加中国社区源：

```
1 [archlinuxcn]
2 Server = https://mirrors.ustc.edu.cn/archlinuxcn/$arch # 中国科学技术大学开源镜像站
3 Server = https://mirrors.tuna.tsinghua.edu.cn/archlinuxcn/$arch # 清华大学开源软件
  ↳ 镜像站
4 Server = https://mirrors.hit.edu.cn/archlinuxcn/$arch # 哈尔滨工业大学开源镜像站
5 Server = https://repo.huaweicloud.com/archlinuxcn/$arch # 华为开源镜像站
```

保存并退出上述文件，然后使用以下命令刷新数据库并更新系统：

```
1 pacman -Syuu
```

3.3.3 配置视窗，以及后续内容

通过以下的命令安装视窗相关的软件包：

```
1 pacman -S plasma-meta konsole dolphin
```

安装完成之后，运行以下命令：

```
1 systemctl enable sddm
```

之后重启电脑就行。输入你新创建的非根用户的密码，然后回车，就可以登录桌面了。

值得注意的是，这时尚未安装任何显卡驱动。如果你在进入桌面环境时遭遇闪退、花屏等异常情况，建议尝试安装相应的显卡驱动。这里我就不提了，感兴趣的同学可以自行查找相关资料进行了解。

之后，可以做一些很好的操作，例如使用 `Ctrl+Alt+T` 打开 Konsole（不是 Console，这个是一个终端模拟器）。连接一下网络，然后安装一些基础功能包：

```
1 sudo pacman -S sof-firmware alsa-firmware alsa-ucm-conf # 声音固件
2 sudo pacman -S ntfs-3g # 使系统可以识别 NTFS 格式的硬盘
3 sudo pacman -S adobe-source-han-serif-cn-fonts wqy-zenhei # 安装几个开源中文字体。一
  ↳ 般装上文泉驿就能解决大多 wine 应用中文方块的问题
4 sudo pacman -S noto-fonts noto-fonts-cjk noto-fonts-emoji noto-fonts-extra # 安装
  ↳ 谷歌开源字体及表情
5 sudo pacman -S firefox chromium # 安装常用的火狐、chromium 浏览器
6 sudo pacman -S ark # 压缩软件。在 dolphin 中可用右键解压压缩包
7 sudo pacman -S packagekit-qt6 packagekit appstream-qt appstream # 确保 Discover
  ↳ （软件中心）可用，需重启
8 sudo pacman -S gwenview # 图片查看器
9 sudo pacman -S archlinuxcn-keyring # cn 源中的签名 (archlinuxcn-keyring 在
  ↳ archlinuxcn)
10 sudo pacman -S yay # yay 命令可以让用户安装 AUR 中的软件 (yay 在 archlinuxcn)
```

之后，如同 root 账户一样，配置其默认编辑器即可。

配置中文环境

首先应当配置系统为中文。打开 System Settings > Language and Regional Settings > Language > Add languages，找到并加入简体中文，然后拖拽到最上面一位，保存并退出设置。重启电脑就可以生效了。

现在该配置汉语输入法了：

```
1 sudo pacman -S fcitx5-im # 输入法基础包组
2 sudo pacman -S fcitx5-chinese-addons # 官方中文输入引擎
3 sudo pacman -S fcitx5-anthy # 日文输入引擎
4 sudo pacman -S fcitx5-pinyin-moegirl # 萌娘百科词库。二刺猿必备 (archlinuxcn)
5 sudo pacman -S fcitx5-material-color # 输入法主题
```

下一步，创建以下文件，然后编辑这个文件：

```
1 vim ~/.config/environment.d/im.conf
```

向文件中加入这些内容并保存退出，以修正输入法的一些错误：

```
1 # fix fcitx problem
2 GTK_IM_MODULE=fcitx
3 QT_IM_MODULE=fcitx
4 XMODIFIERS=@im=fcitx
5 SDL_IM_MODULE=fcitx
6 GLFW_IM_MODULE=ibus
```

之后，打开系统设置-区域和语言，找到输入法一项，运行 fcitx。之后，点击添加输入法，找到拼音输入（或者你喜欢的输入），将其添加为拼音输入法。

现在重启电脑就可以输入中文了。

3.3.4 总结

上面的过程就是从头安装 ArchLinux 的全过程了。实际上我们可以看到，上述过程总体上大概可以分为四部分：

- 1. 准备工作：准备好安装介质（这里是 U 盘）、改 BIOS 设置、联网等。
- 2. U 盘根阶段：从 U 盘启动，进入 Linux 的安装环境；准备硬盘（分区、格式化、挂载等）；安装基础系统。
- 3. 机器根阶段：从 U 盘 chroot 到新的系统，安装剩余的软件包，配置系统（主机名、时区、locale 等）；做启动引导。
- 4. 后续的各种配置。

实际上几乎所有的系统安装过程都可以大致分为这四个部分。只不过不同的系统在细节上有不同，而且许多系统会把这些步骤都封装好，用户只需要简单地点击几下就可以完成安装。

3.3.5 进一步学习

一般说来，能独立的安装好 Arch Linux 并能进行日常维护、找到性能瓶颈（例如谁在偷吃 CPU）并解决问题、熟练使用各种命令行工具，就可以算是一个合格的 Linux 中级用户了。当然，如果你想更进一步，以下这些题目可以作为你的思考和实践方向：

1. 一些常用命令背后是什么？查看诸如 `ls`、`cp`、`mv` 等的源代码，尝试修改它们以添加新功能，或仅让它们的输出更美观。
2. Linux 内核的基本结构和工作原理，例如进程管理、内存管理、文件系统等。
3. Linux 的部署，例如用 `ansible` 等工具实现自动化安装和配置，理解声明式系统（如 NixOS、Guix 等）的原理和优势，并尝试使用它们。
4. Linux 的安全机制，例如 SELinux、AppArmor 等。
5. 试着定制你的系统，创作出好玩的工具，并写 PKGBUILD 文件打包成 AUR 包，发布到 AUR 上。

第四章 初步使用计算机

在开始使用计算机之前，我们需要了解一些基本的计算机使用知识。这些知识包括如何使用键盘和鼠标，如何管理文件和目录，操作系统的日常维护，网站资源的获取，以及常见问题的排查和解决等。关于具体软件怎么使用（如 MS Office 使用方式等），我们不在本书中赘述，因为这些软件的使用方式千差万别，且变化较快，建议同学们通过网络搜索相关教程来学习。

提示

新手不要过分担心搞坏电脑或损坏文件！弄坏是正常的，软件坏了就重装系统，硬件坏了就修理，文件坏了就尝试恢复或重做。只有多犯错、多试错，才能更深入地理解“这些操作不能做”“这样做会怎么样”“这个东西是干什么的”等等。笔者也弄坏过好多次电脑，导致的后果包括并不限于系统崩溃、数据损坏，甚至还有一次搞坏了内存条子。

但是也正是在如此多的试错和反思中，我才逐步理解了计算机的工作原理，学会了如何管理文件和目录，学会了如何使用操作系统，学会了如何排查和解决问题。就是这样，在不断的试错中，我们才能真正逐步理解计算机相关的知识，最终从“什么都不会的小白”变成“见怪不怪的老手”，最终变成能独立解决问题的高手。

所以，不要害怕犯错，勇敢地去试试吧！

4.1 善用键盘

我们早就知道如何使用键盘了，几乎所有人都会使用键盘来进行文字输入等操作。但是键盘上还有一些较为特殊的按键，它们虽然不具体对应某一个字符，但是却能帮助我们更高效地和计算机交互。

4.1.1 如何打字

打字还用学吗？

当你看到这个标题时，一定会冒出上面的念头。打字不就是输入文字吗？要打“双”我们就按“shuang”，要打“你”我们就按“ni”。那么恭喜你已经解锁了最常用也是最符合大部分人使用习惯的输入方式（全拼）。所谓全拼就是需要打全音节的拼写。

很好，现在我们已经了解了全拼打字。但是经过仔细思考，我们可以发现一个重要的问题——和英文不同，汉字的拼音并不是由字母组成的啊！一个汉字的拼音是由一个声母以及一个韵母组成的，如果我们能直接输入声母和韵母，那么打字不就更轻松了吗。于是双拼也应

运而生，这种打字方式是将键盘上的字母区映射到声母与韵母上，这样一来我们击键两次就可以打出一个汉字了！根据映射方案的不同，双拼也有不同的方案（例如自然码、小鹤双拼、微软双拼等）。以小鹤双拼为例，小鹤双拼的映射方案如下图。



图 4.1: 小鹤双拼映射方案

我们来简单见识一下双拼的威力：请在任意文本编辑器中输入“镶嵌在窗框上的霜花正在悄悄融化”这句话。

全拼的键入是这样的“xiang/qian/zai/chuang/kuang/shang/de/shuang/hua/zheng/zai/qiao/qiao/r
双拼的键入是这样的“xl/qm/zd/il/kl/uh/de/ul/hx/vg/zd/qn/qn/rs/hx”。

我们可以明显地看出双拼打字所需的键入次数（我们称为“码长”）更少。包括这个优点，双拼方案与全拼方案的对比优势如下。

- 双拼方案的码长明显减少，打字速度也更快。
- 双拼能够减少一定的重码情况，例如“西安”与“先”、“余额”与“月”等。
- 双拼的码长基本固定在 2，而全拼的码长则根据音节数而定，双拼打字更有节奏感。

这种根据汉字拼音进行输入的方案，我们叫它“音码”。音码的不足在于同音字的处理，一个拼音往往对应十几甚至几十个汉字（我们称为“重码”）。对于常见的字我们可能需要翻页才能找到对应的汉字，十分影响打字节奏。有没有办法避免打出同音字呢？有的，四大发明的活字印刷是将字直接印下来，那么我们也可以将字的偏旁输入然后选择对应的字啊！于是，五笔输入法也应运而生，例如王码 86 五笔、郑码、虎码等方案。这里以王码 86 五笔为例，其映射方案如下图。

这种根据字根字形进行输入的方法我们称为“形码”，可以有效减少同音字的影响而增加输入效率。不过随着拼音的普及，这种难度较高的输入形式使用人数也在减少（例如上世纪的打字员一职）。

而现在，为了再次降低音码的重码率，我们在双拼的基础上也可以引入辅助码，将简单的字形映射到键盘上（辅助码以音托为主，因此学习成本要小于五笔），再结合双拼就基本可以做到和五笔输入相似的效果了。常见的辅助码有自然码辅码、鹤形辅助码等。由于大部分输入

86版五笔字型字根表及助记词

金丰鱼儿 冂乂儿亅 夕夕夕亅 35Q我	人亅八乚乚 八乚乚乚 34W人	月目舟用 夕四乃 攴豕豕豕 33E有	白手产手 夕斤斤斤 斤斤斤斤 32R的	禾禾竹禾 禾禾禾禾 禾禾禾禾 31T和	言文方讠 讠讠讠讠 讠讠讠讠 41Y主	立六立辛 讠讠讠讠 讠讠讠讠 42U产	水水水水 水水水水 水水水水 43I不	火业火火 火火火火 火火火火 44O为	之乚乚乚 乚乚乚乚 乚乚乚乚 45P这				
工卅卅卅 七七七七 15A工	木丁西 木丁西 木丁西 14S要	大犬古石 三手手手 厂厂厂厂 13D在	土土土土 二二二二 雨雨雨雨 12F地	王王王王 一一一一 一一一一 11G一	目目目目 上上上上 上上上上 21H上	日日日日 早早早早 早早早早 22J是	口川川川 口川川川 口川川川 23K中	田甲口口 四四四四 车车车车 24L国					
万能 学习键 Z	多么么么 弓弓匕匕 55X经	又又又又 又又又又 巴马 54C以	女刀九 乚乚乚 白白白 53V发	子子子子 乚乚乚 耳耳耳 52B了	巳巳巳巳 乙乙乙乙 心心心 51N民	山由贝 山由贝 山由贝 25M同							
横区(一)										竖区(丨)	撇区(丿)	捺区(㇏)	折区(乙)
G 11王旁青头戈(兼)五一										H 21目具上止卜虎皮	T 31禾竹一撇双人立反文条头共三一	Y 41言文方广在四一高头一捺谁人去	N 51已半已满不出己左框折尸心和羽
F 12土土二千十寸雨										J 22日早西竖与虫依	R 32白手看头三二斤	U 42立辛两点六门疾	B 52子耳了也框向上
D 13大犬三羊古石厂										K 23口与川字根稀	E 33月衫乃用家衣底	I 43水旁兴头小倒立	V 53女刀九白山朝西
S 14木丁西										L 24田甲方框四车力	W 34人和八三四里	O 44火业头四点米	C 54又巴马丢矢矣
A 15工戈草头右框七										M 25山由贝下框几	Q 35金勺缺点无尾鱼犬旁留又一点夕氏无妻(七)	P 45之宝盖摘示衣	X 55慈母无心幼无力弓和匕

图 4.2: 王码 86 五笔映射方案

法厂商没有配备辅助码，因此有兴趣的同学可以自行学习提升，这里不做展开。

以上是键盘对于中文的输入，这里笔者需要介绍一种特殊的英文输入形式——滑键输入。这种输入形式不需要一个一个键入单词的字母，而是可以连续划过字母直至你想要的单词上屏。这种输入形式可以提高输入英文的效率，在 iOS 输入法或 Gboard 中有相应的支持。

4.1.2 高效打字

虽然很多人都会打字，但是打字的效率却参差不齐。除了使用上述的修饰键和特殊按键以外，我们还可以通过一些方法来提高打字效率。

首先，实际上打字是有一个正确的姿势的。这个姿势要求两只手的食指分别放在键盘的 F 和 J 键上，这两个键被叫做“基准键”。在漫长的使用过程中，可能基准键会随着习惯而有所偏移（例如笔者的习惯是将 D 和 H 作为基准键，这样比较灵活的右手会辐射更大的范围），这并不需要纠正，因为习惯下来也没什么大问题。但是不管怎么样，最终你十个手指当中的大多数数在打字过程中要动起来，而不是二指禅，这样才能显著提高打字效率。

有些人会问五笔和拼音哪个打字更快。实际上，这取决于你所使用的打字方案，对于打字员等专业人士而言，他们打字的时候眼睛只看着手写或打印的纸张，屏幕和键盘都不看；他们和抄写一样，直接复制字形并打出编码，因此五笔的速度会更快一些。但是，五笔是违反人语言习惯的输入法，需要对大量字根字形非常熟练，而我们在写东西的时候，往往是先想到词语的发音，然后再想到字形，而五笔则是直接根据字形来输入，这对于大多数人而言并不自然。

因此，对于大多数人而言，拼音输入法（这里指全拼与双拼）更为合适。另外，随着技术的不断进步，现代的大多数输入法连接了网络，能够通过用户的上文推测下文大概是什么词语，从而提高了输入效率，双拼 + 辅助码的组合也可以媲美五笔的极限速度。

目前大多数输入法都有特殊的输入功能，例如按下 u 即可进入生僻字输入模式，可以用于

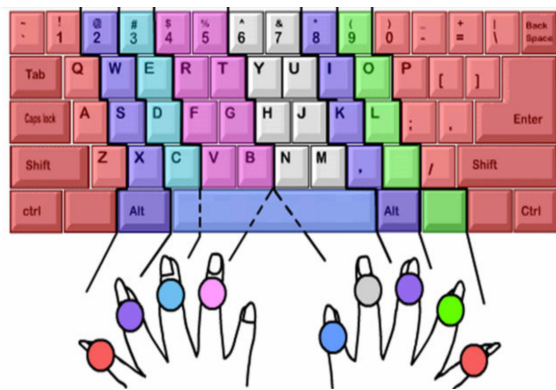


图 4.3: 标准指法打字示例

输入一些不常用的汉字；按下 **v** 可以输入英文符号等。我们建议同学们阅读自己使用的输入法的帮助文档，了解这些特殊功能，从而提高边界条件下的输入效率。

最后，无论你使用全拼、双拼或者五笔输入，你都需要明白打字是一个熟练工作，只有多练习才能提高打字速度和准确率。对于双拼以及五笔，我并不建议去背诵对应的映射方案，你只需要将你所有的输入形式改为对应输入方案并将手机屏保换为对应映射表，不出 2 个月，你就基本能拥有和全拼相近的输入效率了。在熟练之后，你可以练习盲打。打字较快的人大多数都会盲打，也就是在打字的时候眼睛仅盯着输入法选字窗口，而不看键盘。这样可以减少眼睛在键盘和屏幕之间的移动，提高打字效率。我们推荐使用一些打字练习软件，例如 TypingClub、Keybr、金山打字通等，来提高打字速度和准确率。

4.1.3 修饰键等特殊按键

键盘上除了字母、数字和标点符号等常用按键以外，还有一些特殊按键。这些按键通常不会直接输入字符，而是用于控制计算机的操作。常见的特殊案件有修饰键和功能键两大类。

修饰键是一种特殊的按键，它们通常不会单独使用，而是与其他按键组合使用，以实现更多的功能，这些组合按键就叫做“快捷键”。常见的修饰键有以下几种：

- **Ctrl (Control)**：控制键，通常用于执行各种命令和操作。
- **Alt (Alternate)**：替换键，通常用于切换窗口、菜单等。这个键在 macOS 中被叫做 **Option** 键。
- **Shift**：换档键，通常用于输入大写字母和特殊符号。
- **Win (Windows 键)**：Windows 徽标键，通常用于打开开始菜单、切换窗口等。这个键仅在 Windows 系统中存在，在 Linux 中被叫做 **Super** 或 **Meta** 键，在 macOS 中被叫做 **Command** 键。
- **Fn (Function)**：功能键，通常用于切换键盘的功能模式，例如调整音量、亮度等。这个键不是每一个键盘上都有。

除修饰键和字母、数字外，还有一些特殊按键。这些功能键也可以帮助我们高效地和计算机交互。

- **Esc (Escape)**：退出键，通常用于退出当前操作或者关闭窗口等。

- Tab (Tabulator)：制表键，通常用于切换输入焦点或者插入制表符等。
- Caps Lock：大写锁定键，按下后可以输入大写字母，按下再次取消。
- PrtScn (Print Screen)：打印屏幕键，通常用于截取屏幕内容。在不同的计算机上也有不同的名字。
- Pause Break：暂停键，通常用于暂停程序的执行。在现代计算机上，这个键很少使用。
- Scroll Lock：滚动锁定键，通常用于锁定滚动功能。在现代计算机上，这个键很少使用。

一开始的键盘是为了和终端（黑窗口）交互的，为了操作光标，因此也有着一些光标控制键：

- 方向键（上下左右箭头）：用于控制光标的移动。
- Home：光标移动到行首。
- End：光标移动到行尾。
- Page Up：向上翻页。
- Page Down：向下翻页。
- Delete：删除键，通常用于删除光标后的字符，与 Backspace 正好相反，后者用于删除光标前的字符。
- Insert：插入键，通常用于切换插入模式和覆盖模式。

4.1.4 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：

表 4.1: Windows 常用快捷键

快捷键	功能	快捷键	功能
Ctrl+C	复制	Alt+F4	关闭当前窗口
Ctrl+V	粘贴	Alt+Tab	迅速切换窗口
Ctrl+Z	撤销	Win+R	打开运行窗口
Ctrl+Y	重做	Win+E	打开文件资源管理器
Ctrl+A	全选	Win+D	显示桌面
Ctrl+Alt+Del	打开任务管理器	Win+L	锁定计算机
Ctrl+S	保存当前文档或文件	PrtScn	截图（全屏）
Ctrl+P	打印当前文档或文件	Alt+PrtScn	截图（当前窗口）
Ctrl+F	查找文本或内容	Win+Shift+S	截图（自定义区域）

除了上述快捷键以外,在其他的软件中也有着自己特定的快捷键.例如在浏览器中,Ctrl+T 可以打开一个新的标签页, Ctrl+W 可以关闭当前标签页, Ctrl+Shift+T 可以重新打开上一个关闭的标签页等。这些快捷键可以帮助我们更高效地使用计算机。

4.2 文件、目录及其管理

4.2.1 理解文件和目录

文件和目录是计算机中最重要的概念之一，它们用于存储和组织计算机中的数据。

如果把硬盘看作一个图书馆，那么文件就可以看作是图书馆中的一本书。目录则可以理解为图书馆的房间、书架等，以及图书馆本身，用于组织和分类图书馆中的书籍。假如我想找到一本书，首先要知道它在哪个图书馆，其次是在哪个房间、哪个书架的哪个位置。这样就能够使用统一的方式来找到一本特定的书。

在任何计算机系统中，文件都可以看作是一些数据块。文件可以是文本文件、图片文件、音频文件、视频文件等。每一个文件都有一个唯一的名称，用于标识该文件。现代文件系统的文件名通常由字母、数字和特殊字符组成。在 Windows 上，文件名不区分大小写；在 Linux 和 macOS 上，文件名区分大小写。

4.2.2 文件的大小

既然文件是一些数据块，那么就可以度量其大小。现代计算机使用二进制来表示数据，规定一个二进制位为 1 比特 (bit)，8 个二进制位为 1 字节 (Byte)。一个数据块占了多少二进制位，就可以说这个文件的大小是多少比特，或者占了比特数除以 8 的字节数。

在计算机上为了方便读取，之后的数据大小单位以 $2^{10} = 1024$ 为倍数递增，即 $1\text{KiB}=1024\text{B}$ ， $1\text{MiB}=1024\text{KiB}$ ， $1\text{GiB}=1024\text{MiB}$ ， $1\text{TiB}=1024\text{GiB}$ 。¹另外一套单位系统以 1000 为倍数递增，也就是 $1\text{KB}=1000\text{B}$ ， $1\text{MB}=1000\text{KB}$ ， $1\text{GB}=1000\text{MB}$ ， $1\text{TB}=1000\text{GB}$ 。这种标定方式便于硬件的生产，因此通常由硬件厂商使用，我们在日常生活中看到的硬盘、U 盘等存储设备的容量使用的就是这种方式。因此购买的硬盘，计算机认为的容量会比标定容量小一些，例如一个标定 1TB 的硬盘：

$$1\text{TB} = 1000^{12}\text{B} = \frac{1000^{12}}{1024^4}\text{TiB} \approx 0.9095\text{TiB} = 931.32\text{GiB}$$

于是计算机会认为这个硬盘只有约 0.91TiB 的容量：在 Windows 上，显示为 931GB²；在 Linux 和 macOS 上，显示为 931GiB。这是正常现象，没有人偷你的硬盘空间！

上述 1000GB 硬盘被称作“非足容硬盘”。而“足容硬盘”则是 1024GB 硬盘（不是 GiB!），于是这个就在计算机上显示为 953.67GiB，似乎也没“足”。一般机械硬盘都是非足容硬盘，而固态硬盘则有不少是足容硬盘，主要原因是便于制作：机械硬盘的容量是通过物理结构决定的，而固态硬盘则是通过芯片来决定的，芯片的容量往往是 2 的幂次方。

注意

有些无良卖家会使用这种单位：32Gb U 盘，然后买到手发现只有 4GB。这是因为该厂商使用了 Gb (Gbit) 而不是 GB (GByte) 来混淆视听，也就是 $32\text{Gb}=4\text{GB}$ ，他甚至还没说谎，买家也只能打落牙齿往肚子里吞了。因此，购买存储设备时请务必注意单位究竟是什么。

¹Windows 系统中的 KB 指的实际上是 KiB，这个是 Windows 本身的问题！而 Linux 和 macOS 则会正常显示 KiB 等。

²这里是因为 Windows 用错了单位！问题已经很多年了。

4.2.3 目录结构与路径

在 Windows 系统中，每一个硬盘³都是一个“根”目录，相当于图书馆本身。每一个根目录都有一个盘符，例如 C 盘、D 盘等。每一个根目录下可以有多个子目录，这些目录往往表现为文件夹的形式；每一个子目录下又可以有多个子目录，最终形成树状结构。每一个目录下可以有多个文件（相当于图书馆中的书籍）。对于一个特定的文件，从根目录到所在地的路径是唯一的，例如 `C:\Users\username\Documents\file.txt`。这个完整写出来的路径叫做**绝对路径**，它唯一地标识了一个文件的位置。

有时候，我们在图书馆找书的时候，知道同类的书籍往往放在一起。例如我们已经知道了这个书架上有某书第一卷，我们也知道同一个书架上还有第二卷，这时候就不需要再从图书馆的根目录开始找了，而是可以直接从这个书架上找。这时候我们可以说，第二卷的路径**相对于第一卷的路径**在同一个目录下。这个相对于某个目录的路径叫做**相对路径**，它并不是唯一的，因为它依赖于当前所在的目录。举个例子，假如某文件 `file1` 在某目录下，同目录下还有一个 `file2`，那么 `file2` 相对于 `file1` 的路径就是 `file2`。如果该目录下还有一个子目录 `dir`，该子目录下有一个 `file3`，那么 `file3` 相对于 `file1` 的路径就是 `dir\file3`。

有些时候，相对路径可能会跨目录，这时候就需要向上一级等操作。这时候，就可以使用“`..`”来表示上一级目录，使用“`.`”来表示当前目录。例如，假如某文件 `file1` 在某目录下，该目录的上一级目录下有一个 `file2`，那么 `file2` 相对于 `file1` 的路径就是 `..\file2`。同样，如果该目录的上级目录下还有一个子目录 `dir`，该子目录下有一个 `file3`，那么 `file3` 相对于 `file1` 的路径就是 `..\dir\file3`。

4.2.4 文件的扩展名

扩展名是文件名中最后一个点后面的部分，例如 `file.txt` 的扩展名是 `txt`。扩展名可以帮助操作系统识别文件的类型，并选择合适的程序来打开它。在 Windows 上，文件的扩展名往往是必须的，否则操作系统不能识别文件的类型。

在 Windows 系统中，文件的扩展名通常是隐藏的。如果你想查看文件的扩展名，可以在文件资源管理器中点击“查看”菜单，然后选择“文件扩展名”选项。这样就可以看到所有文件的扩展名了。为了方便维护目录等，笔者建议同学们选择显示扩展名。

一般情况下，虽然文件的类型差不多，但是其扩展名不同也往往代表着其数据的存储是按照不同的方式进行的。例如，`bmp` 文件是“位图”，存储的是每一个像素的颜色信息；而 `jpg` 文件则是经过压缩的，存储的是图像的整体信息。对于同一张图片，`bmp` 文件往往会比 `jpg` 文件大得多。如果我们试图将一个 `bmp` 文件改名为 `jpg`，那么这个文件的扩展名变了，但是其**内部数据并没有变**，因此这个文件仍然是一个 `bmp` 文件，只不过扩展名变了而已。此时，如果我们使用图片查看器打开这个文件，图片查看器会根据扩展名来判断文件类型，认为它是一个 `jpg` 文件，但是实际上它是一个 `bmp` 文件，因此图片查看器可能无法正确地打开它。因此，技术人说的“一类文件”往往不是按文件的性质分类的，而是按文件的扩展名（数据的存储方式）分类的。

³实际上往往是硬盘分区。在旧的计算机中，也往往叫做驱动器或卷。初学者暂时不必纠结这些概念，只需要知道有这个东西就好了。

常见文件类型	常见扩展名
文本文件	.txt, .md, .log
文档文件	.doc(x), .xls(x), .ppt(x), .pdf
图片文件	.jpg, .jpeg, .png, .gif, .bmp, .svg
音频文件	.mp3, .wav, .flac, .aac
视频文件	.mp4, .avi, .mkv, .mov
压缩文件	.zip, .rar, .7z, .tar, .gz
可执行文件	.exe, .bat, .sh
代码文件	.c, .cpp, .py, .java, .js, .html, .css

表 4.2: 常见文件类型及其扩展名

4.2.5 文件的链接（快捷方式）

有时候，我们可能会需要在不同的目录中使用同一个文件或目录，这时候就可以使用文件的链接。文件的链接类似于图书馆中的索引卡片，它指向某个文件的位置，而不是文件本身。

上述建立链接的方式叫做软链接（符号链接）。与之相对的是硬链接，硬链接是指多个文件名指向同一个文件的数据。硬链接的创建和管理比较复杂，一般不建议初学者使用。

4.2.6 文件的基本操作

文件的基本操作包括创建、删除、重命名、复制、移动、链接和运行。

在 Windows 系统下，创建、删除、重命名操作都可以通过文件资源管理器提供的右键菜单来完成。

复制指的是把一个文件的内容原模原样地写到一个新的文件中，两个文件互不影响；而移动指的是把一个文件从一个位置移动到另一个位置，原位置的文件会被删除。复制和移动操作都可以通过文件资源管理器提供的右键菜单来完成。复制后需要粘贴才能完成，而移动则是直接拖拽文件到目标位置即可（或者剪切后粘贴）。

链接在 Windows 下被叫做“创建快捷方式”，可以通过右键菜单来完成。运行指的是打开一个文件或者程序，可以通过双击文件来完成（或者右键菜单选择“打开”）。

4.2.7 更改文件的类型

有时候，我们可能会需要更改文件的类型。直接更改扩展名并不能改变文件的类型，因此我们需要使用专门的软件或者网站来帮助我们转换文件的类型。

以下是几个常见的手段：

- 格式工厂：一个免费的多功能文件转换软件，支持音频、视频、图片、文档等多种格式的转换。但是现在已经不更新了。
- 网页：有很多在线文件转换网站，支持多种格式的转换，使用方便，无需安装软件。

- 专业软件：有些专业软件自带文件转换功能，例如 Adobe Acrobat 可以将 PDF 文件转换为 Word、Excel 等格式，Microsoft Word 可以将文档转换为 PDF 等格式。

一般而言，只要记得别手动给扩展名删了或改了就行！上述方法其实完全可以信赖其正确性。

4.2.8 文件的打开方式

对于任何文件，我们往往需要读取其中的数据，这个文件才有意义；至于这个数据是怎么展示的则另当别论。因此，我们需要一些软件来帮助我们打开文件。

在 Windows 系统中，文件往往有一个默认的打开方式，这个打开方式由文件的扩展名决定。例如，.txt 文件默认使用记事本打开，.jpg 文件默认使用照片查看器打开。有些时候安装了一些软件后，文件的默认打开方式可能会被更改。例如，安装了某个图片编辑软件后，.jpg 文件的默认打开方式可能会被更改为该软件。

在一些情况下，我们可能会需要临时或永久更改某一类文件或者某一个文件的打开方式。我们可以右键点击文件，然后选择“打开方式”选项，选择一个合适的软件来打开该文件，也可以选择“选择其他应用”之后寻找指定的软件。如果我们想要永久更改某一类文件的默认打开方式，只需要在“选择其他应用”菜单勾选“始终使用此应用打开 . 扩展名文件”选项即可。

值得注意的是，软件本身并不需要扩展名才能打开文件。软件打开文件时总是按照自己的方式来处理数据，因此很多时候即使扩展名是错误的或者没有扩展名，如果指定了正确的软件，软件仍然可以正确地打开文件。

4.2.9 文件的压缩与解压

有时候文件占据了过大的空间，或者因为文件数量太多而不便于传输和管理，这时候我们可以使用文件压缩工具来将文件进行压缩。文件压缩工具可以将多个文件或者目录打包成一个文件，并且可以对文件进行压缩，以减少文件的大小。常见的文件压缩格式有 .zip、.rar、.7z 等。

笔者个人推荐使用 7z 压缩软件来压缩和解压文件。它是一个经典的开源软件，支持多种压缩格式，并且具有较高的压缩率。一般情况下，在我们安装了 7z 软件之后，右键点击文件或者目录，就可以看到“添加到压缩文件”选项，点击该选项即可将文件或者目录进行压缩。对于压缩文件，我们也可以右键点击，然后选择“解压到当前目录”选项，或者选择“解压到文件名\”选项，将文件解压到指定的目录中。

不同的压缩参数会产生不同的效果。把固实数据大小开大则会提高压缩率⁴，但会牺牲压缩和解压的速度；勾选了分卷，则会将压缩文件分割成多个小文件，便于诸如 FAT32 等不支持大文件的文件系统存储；使用内存和 CPU 则显著影响了压缩的速度；不同的压缩算法则会影响压缩率和速度的平衡。一般情况下，默认参数已经足够好用了，软件本身也提供了多种预设参数供我们选择。

⁴压缩率是指压缩后的文件大小与原文件大小的比值，压缩率越高，说明压缩效果越好。

至于压缩成的文件格式，笔者比较推荐使用 zip 格式，其兼容性最好；如果对压缩率有较高要求，可以使用 7z 格式。不建议使用 rar 格式，因为它是一个专有格式，虽然压缩率不错，但是不够开放。

注意

有些软件的可执行文件和附带文件是通过压缩包分发的；有些文件也是通过压缩包传输的。在这种情况下，我们一定要先解压文件，再使用文件，而不是在压缩包中直接使用文件，这样可能会导致文件无法正常工作。

那么怎么压缩呢？单击右键，选择“7z > 添加到压缩文件...”即可。如果计算机是 Windows 11，没有这个选项，则先单击“显示更多选项”，然后再选择“7z > 添加到压缩文件...”即可。之后，我们可以选择压缩格式、压缩等级、分卷大小等参数，然后点击“确定”按钮即可开始压缩文件。

现代计算机硬盘空间一般都充裕，压缩文件大多是为了整合小文件为一个大文件，便于发走的。如果我们没有勾选分卷，我们得到的一般是 1 个大文件，直接发送即可；如果我们勾选了分卷，我们得到的则是多个小文件，这时候我们需要将这些小文件全部发送给对方，对方才能正确地解压文件。“分卷”和硬盘没有关系！它只是把一个大文件拆成了多个小文件而已。

4.2.10 进阶：高效的文件管理

从一大堆文件中快速找到我们想要的文件是一个非常重要的技能。

一个常用的手段是使用搜索功能。Windows 文件资源管理器提供了搜索功能，可以帮助我们找到文件。我们可以在文件资源管理器的右上角输入关键词，文件资源管理器会自动搜索当前目录及其子目录中的文件，并显示匹配的结果。然而，文件管理器的搜索面对大量文件时效率很低。这时候，我们可以使用一些第三方的搜索工具，例如 Everything、Listary 等。这些工具可以快速索引计算机中的文件，并提供快速的搜索功能。它们通常比文件资源管理器的搜索功能更快、更强大。

另一个常用的手段是养成良好的存储习惯，将相关的文件放在一起，使用有意义的文件名和标签、分类等工具。Windows 没有内置的标签功能，但我们可以使用一些第三方的标签工具，例如 TagSpaces、FileMeta 等。对于大量的文件，我们可以使用自动化脚本来帮助我们管理文件，例如使用 Python 脚本来批量重命名文件、移动文件等。在 Windows 上，一个最简单的批量处理文件的方式是使用 PowerShell 脚本或旧式风格的批处理脚本（.bat 文件）。这些脚本可以帮助我们自动化一些重复性的任务，提高工作效率。

例如，我们希望把某目录下的许多图片都改名为“图片 1.jpg”、“图片 2.jpg”等等。我们可以使用以下 PowerShell 脚本来实现：

```
1 $i = 1
2 Get-ChildItem -Path "C:\path\to\your\images" -Filter *.jpg | ForEach-Object {
3     $newName = " 图片 $i.jpg"
4     Rename-Item $_.FullName -NewName $newName
5     $i++
6 }
```


当然用 Python 也行：

```
1  import os
2
3  path = r"C:\path\to\your\images"
4  files = [f for f in os.listdir(path) if f.endswith('.jpg')]
5  for i, file in enumerate(files, start=1):
6      new_name = f" 图片{i}.jpg"
7      os.rename(os.path.join(path, file), os.path.join(path, new_name))
```

使用这些工具可以帮助我们更高效地管理文件，避免手动操作的繁琐和错误。

4.3 校内网络配置指南



图 4.4: 北京大学土豆大棚（划去）校园网机房

4.3.1 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。

4.3.2 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](#)。

4.3.3 校外连接北大内网：北大 VPN

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，这里就不赘述了。

4.3.4 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。

4.3.5 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。

警告

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。

4.3.6 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。学校的部分重要通知会发送到该邮箱上，请同学们注意查收。

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 Secure 连接指南: https://its.pku.edu.cn/setting_6.jsp

eduroam 连接指南: https://its.pku.edu.cn/service_1_eduroam.jsp

北京大学 VPN: https://its.pku.edu.cn/service_1_vpn.jsp

北大网盘指南: https://its.pku.edu.cn/service_1_webdisk.jsp

腾讯会议教育版指南: https://its.pku.edu.cn/service_1_webex.jsp

	pku.edu.cn	stu.pku.edu.cn
IMAP 服务器	imap.pku.edu.cn	imaphz.qiye.163.com
IMAP 端口	993	993
SMTP 服务器	smtp.pku.edu.cn	smtphz.qiye.163.com
SMTP 端口	465	994

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，密码则是来自特定客户端的授权码。对 pku.edu.cn 用户而言，授权码需在北大邮箱中获取，详见[通知原文](#)；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。

4.4 计算机的日常维护与安全

4.4.1 字体的安装与管理

在 Windows 系统中，字体管理器可以通过控制面板中的“字体”选项来访问。用户可以将字体文件拖放到该窗口中来安装字体，或者右键点击字体文件并选择“安装”来安装字体。卸载字体也可以通过右键点击字体文件并选择“删除”来完成。用户还可以通过该窗口来预览字体文件，并查看字体的详细信息，例如字重、样式、版本等。

4.4.2 计算机的日常维护

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。

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。

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。

通知原文: <https://its.pku.edu.cn/announce/tz20250702100126.jsp>

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。

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）。

我们不推荐使用 360 等软件：360 软件比较臃肿，不易关闭和移除，且会显著拖慢系统速度。曾有笑话：如何判断你需不需要 360？答：如果你不会卸载 360，那你就需要 360。

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。

注意

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，不要一口气删除大量内容，如果必要还应预先备份或者创建系统还原点等。

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。

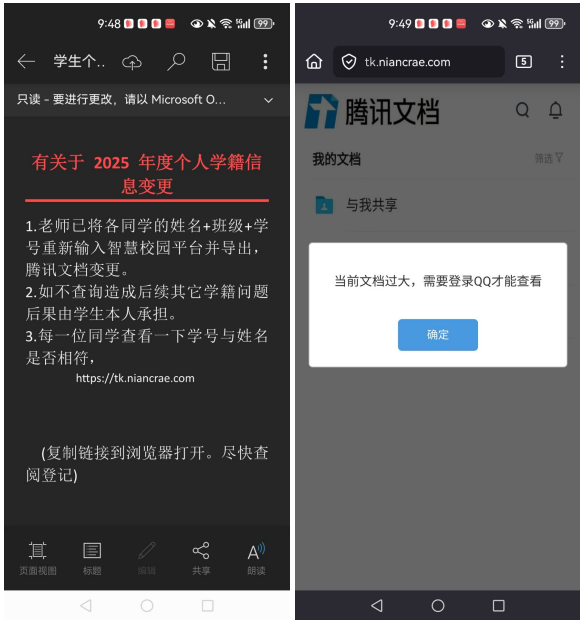
直接使用 Windows 自带的碎片整理工具即可。对于固态硬盘，碎片整理并不会提高性能，反而会缩短使用寿命，因此不建议对 SSD 进行碎片整理。

4.4.3 网安相关知识

网络的风险

虽然互联网的出现给我们带来了便利，但也带来了许多风险。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。



(a) 钓鱼邮件 (b) 伪造的登录页面

图 4.5: 钓鱼攻击示例，右侧这个页面的域名显然不是 qq 的域名，其界面也被不合理地拉伸了，属于典型的钓鱼页面。该钓鱼页面会诱骗用户输入 QQ 账号和密码，从而窃取用户的账号信息。

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。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，病毒不能自己传播，只能依附于其他软件传播；而蠕虫可以自我复制。

被上述恶意软件感染后，计算机会变得不稳定、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。

提示

实际上，现代计算机的安全性比几十年前的 DOS 时代要高太多了。DOS 时代的计算机没有用户权限管理，任何程序都可以对系统进行修改，因此病毒、木马等恶意软件肆虐，其种类也是多种多样；而现代计算机有严格的用户权限管理，普通软件无法随意修改系统文件；现代操作系统的安全性也相当完善，能够有效地防止恶意软件的攻击，恶意软件仅能通过特定的漏洞进行攻击，且这些漏洞往往会被迅速修补。只要我们按时更新系统和软件，保持良好的上网习惯，坚持从官方下载正版软件，基本上就不会被恶意软件感染，不必过于担心、束手束脚，大多数情况下也不需要一直挂着除了自带的杀毒软件（Windows Defender、XProtect 等）以外的第三方杀毒软件。

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。

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。

4.4.4 进阶：软件版权和开源协议

计算机道德是双向的：我们不仅需要保护自己的计算机和数据安全，也需要尊重他人的劳动成果和数据安全。有时候我们认为稀松平常的事情，“不上秤不到四两，上秤一千斤都打不住”：比如说用爬虫爬取 IEEE 论文，可能就会被 IEEE 警告；或者在 GitHub 上上传了一些别人写的 GPL 代码，也有可能被 GitHub 报 DMCA 警告。因此我写了本节，以提醒同学们规避风险。

可以简单地把软件分为两类：商业软件和开源软件。不少商业软件都是收费的，例如 Windows、MS Office、Adobe Photoshop 等；而开源软件则是免费的，例如 Linux、Vim、GCC 等。软件也是“数字作品”，自然是受版权法保护的。

商业软件一般遵循专有许可证，例如微软的 Windows、Office 等。这些软件的源代码是保密的，用户只能使用软件的二进制文件，也不能随意修改和分发软件。这类许可证一般被叫做 Copyright。对于此类软件的盗版，即使是个人使用也面临侵权风险；如果是把它用于商业则风险更大，可能面临严厉的惩罚。

而与 Copyright 相对应的也有 Copyleft，即开源许可证。开源许可证允许用户自由地使用、修改和分发软件，但是需要遵守一些规定。常见的开源许可证有 GPL、MIT、Apache 等。这些许可证一般要求用户在分发软件时附带原始许可证，并且在修改软件时注明修改内容。

以 GPL（GNU General Public License）为例，它是最具代表性的 Copyleft 许可证之一。GPL 的核心思想是：任何人都可以自由地使用、修改和再发布软件，但如果将修改后的版本公开发布（例如发布到网上或提供给他人使用），那么整个衍生作品也必须以 GPL 许可证发布。

这意味着，基于 GPL 代码开发的软件也必须开源，从而确保“自由”能够延续下去。这种“传染性”特征使得 GPL 在开源社区中备受推崇，但也引发了一些商业公司的顾虑。

除了 GPL 以外，还有以下这些常见的开源许可证：

- MIT: MIT 许可证是一种非常宽松的开源许可证，允许用户几乎无限制地使用、修改和分发软件。唯一的要求是必须在软件中包含原始许可证和版权声明。MIT 许可证非常适合那些希望最大限度地推广其软件的开发者。
- Apache: Apache 许可证也是一种宽松的开源许可证，允许用户自由地使用、修改和分发软件。与 MIT 许可证类似，Apache 许可证要求用户在分发软件时附带原始许可证和版权声明。此外，Apache 许可证还包含了一些专利授权条款，允许用户在某些情况下使用专利技术。
- BSD: BSD 许可证是一种宽松的开源许可证，允许用户自由地使用、修改和分发软件。与 MIT 许可证类似，BSD 许可证要求用户在分发软件时附带原始许可证和版权声明。BSD 许可证有多个版本，最常见的是 3-Clause BSD 和 2-Clause BSD。
- MPL: MPL (Mozilla Public License) 是一种中等宽松的开源许可证，允许用户自由地使用、修改和分发软件。与 GPL 不同，MPL 允许用户将修改后的代码与闭源代码混合使用，但要求对修改过的文件进行开源。MPL 适合那些希望在保护部分代码的同时，仍然参与开源社区的开发者。

值得一提的是，开源并不等于“没有限制”。虽然用户可以自由使用、修改和分发软件，但必须遵守相应的开源许可证条款。违反这些条款，比如未按要求附带许可证、未注明修改内容，或者将 GPL 代码用于闭源商业软件，都可能构成侵权行为，甚至引发法律诉讼。

而实际上，同学们并不要那么担心：大多数开源许可证都比较宽松，允许用户自由地使用、修改和分发软件和代码，例如：

- 仅调用 LGPL 代码库：不需要开源自己的代码，这是因为 LGPL (Lesser General Public License) 允许在闭源软件中使用其代码库，只要不修改 LGPL 代码本身即可。但 GPL 则不允许这样做。
- 照抄 MIT 代码段：允许照抄，但需要附带原始许可证和版权声明。
- 把 GPL 二进制放 Docker: 允许，但整个镜像得跟着 GPL 走，也就是说你得开源 Dockerfile 和所有修改过的文件。

4.5 进阶：正确且高效地获取网站资源

在学习和工作中，我们常常需要从网站上获取一些资源，例如图片、视频、文档等。然而，在部分网站上，想直接下载这些资源并不容易。我们可以使用一些工具和方法来帮助我们获取这些资源。

4.5.1 网站抓取

一个最常规的方法是使用网站抓取工具，例如 HTTrack、wget 等。这些工具可以帮助我们下载整个网站或者部分网站的内容，并且可以设置一些选项，例如下载深度、文件类型等。

wget 是非常老牌的网站抓取工具，我将在这里介绍它的基本用法。

wget 的基本语法如下：

```
1 wget [选项] [URL]
```

URL 是要下载的资源地址，选项是对下载过程的补充说明。实际上这个选项才是重头戏。以下是一些常用的选项：

- `-r`：递归下载，即下载整个网站或者部分网站的内容。
- `-l < 深度 >`：设置下载的深度，例如 `-l 2` 表示下载两层目录。
- `-A < 文件类型 >`：设置要下载的文件类型，例如 `-A jpg,png` 表示只下载 jpg 和 png 格式的文件。
- `-P < 目录 >`：设置下载的目录，例如 `-P /home/user/downloads` 表示将下载的文件保存到指定目录。
- `-c`：断点续传，即如果下载过程中断，可以从中断的地方继续下载。
- `-e robots=off`：忽略网站的 robots.txt 文件，强制下载所有内容。该选项可能会违反网站的使用条款，请谨慎使用。
- `-q`：安静模式，不显示下载过程中的信息。
- `-O < 文件名 >`：将下载的内容保存为指定的文件名。

例如，某网页地址为<http://example.com/files.html>，该网页上有很多文件的链接，我们想要下载所有的 PDF 文件，可以使用以下命令：

```
1 wget -r -l 1 -A pdf -P ./downloads http://example.com/files.html
```

该命令会递归下载该网页上的所有 PDF 文件，并将其保存到当前目录下的 `downloads` 文件夹中。

另一个简单的使用例子是下载单个的文件：

```
1 wget --show-progress -P ~/Downloads http://example.com/data_clip.zip
```

一般情况下我个人习惯使用 `--show-progress` 来显示下载进度，且不使用 `-q` 来隐藏下载信息，这样可以方便地查看下载的进度和状态。

wget 适用于抓取网站的静态资源，例如图片、文档等。然而，现在很多网站都是动态的，例如 Bilibili 的播放页面的工作原理是：首先加载一个 HTML 页面，然后通过 JavaScript 代码从服务器获取视频的真实地址，在这个过程中还需要进行鉴权等操作。对于这种网站，wget 这种静态抓取工具就不能使用了。对此，应使用更激进的手段，如爬虫等。也可以使用专门的下载工具（例如 youtube-dl、yt-dlp、IDM 的浏览器插件），这些工具可以帮助我们获取动态网站的资源。

4.5.2 浏览器开发者模式

现代浏览器都提供了开发者模式，可以帮助我们查看网站的源代码、网络请求等信息。我们可以通过按下 F12 键或者右键点击页面选择“检查”来打开开发者模式。在开发者模式中，我们可以查看网站的 HTML 代码、CSS 样式、JavaScript 代码等内容，还可以查看网络请求，找到资源的真实地址。另外，还可以输入一些 JavaScript 代码来帮助我们调试。这个功能难度比较大，建议有兴趣的同学自行学习。

第三部分

终端和 **Linux** 基础：为开发铺垫语境

第五章 终端 101

在计算机的世界里，图形化界面（GUI）无疑是最直观、最易上手的交互方式。我们可以通过鼠标点击图标、菜单等元素来完成各种操作，而不需要记忆复杂的命令和语法。然而，图形化界面并不是唯一的交互方式：我们使用 GUI，实际上是在使用程序员预先抽象好的交互逻辑。如果遇到一些复杂的任务，例如“从许多文件中快速找到特定的内容然后在它们的末尾都加上其修改日期”，此时使用 GUI 则会非常繁琐。难道只能坐等程序员更新软件吗？显然不是。我们可以使用另一种交互方式：命令行界面（CLI），也就是我们常说的“终端”或者“控制台”。不要害怕这个东西！影视剧等作品经常将终端描绘成黑客专用工具，实际这玩意远远没有那么可怕。它只是一个工具，和图形界面（GUI）一样，都是用来和计算机交互的手段而已。

因为 CLI 直接与操作系统交互，能够提供更高的灵活性和效率。通过终端，我们可以直接输入命令来完成各种操作，例如文件管理、软件安装、系统配置等。虽然 CLI 的学习曲线较陡峭，需要记忆大量的命令和参数，但是一旦掌握了它，我们就能够更高效地使用计算机，甚至可以实现一些图形化界面无法完成的任务。

在 Windows 上，只需要按下 Win+R，然后输入 cmd，按下回车键即可打开命令提示符（CMD）；在 macOS 上，可以通过“应用程序”中的“实用工具”文件夹找到“终端”应用程序；在 Linux 上，可以通过按下 Ctrl+Alt+T 快捷键或者在应用程序菜单中找到“终端”应用程序来打开终端。

提示

terminal、shell、CLI、console 等术语严格说来是不同的概念，但是在大多数技术交流时几乎完全不作区分，可以随便混用。其具体定义是：

- **terminal**（终端）：最初指的是连接到大型计算机的物理设备，现在通常指的是提供命令行界面的软件。
- **shell**（外壳）：指的是提供命令行界面的程序，它解释用户输入的命令并将其传递给操作系统执行。
- **CLI**（命令行界面）：指的是通过命令行与计算机交互的界面。
- **console**（控制台）：最初指的是计算机的物理控制台，现在通常指的是提供命令行界面的软件，类似于 terminal。

那为什么这四个东西合流了呢？因为现代计算机中，这些概念往往是重叠的。例如，我们使用的 terminal 实际上就是 console（但是在古老的计算机中，console 和 terminal 是不同的东西，前者是管理员用的物理设备，而后者是用户用的物理设备）；而这玩意通常包含一个 shell，并提供一个 CLI 界面。你看，区别起来是不是就毫无意义了？因此，在大多数情况下，我们可以将这些术语视为同义词，互换使用。

但是有些时候，这些东西也是不得不区分的，例如下文在介绍不同的 shell 时，我们就必须区分，因为这涉及到其具体功能（解释命令）。与之类似的，与 GUI、TUI 对应的只能是 CLI，这涉及到其交互方式。

又比如，Linux 上的终端模拟器（例如 GNOME Terminal、Konsole 等）都是 terminal，而不能说这些东西是 shell 或 CLI。console 仅在 Windows 语境下与 terminal 有区分，因为 console 在 Windows 语境下指的就是 CMD.exe，而 terminal 则可以是 Windows Terminal、PowerShell 等，但 console 和 terminal 却往往都翻译成终端……真是一笔烂账。其他语境下则可以混用。

5.1 选择 Shell

Shell 是终端中的一个重要组件，它负责解释用户输入的命令并将其传递给操作系统执行。不同的操作系统有不同的默认 shell，例如 Linux 和 macOS 默认使用 bash 或者 zsh，而 Windows 默认使用 CMD 或者 PowerShell。

对于 Linux 和 macOS，常见的 shell 有以下几种：

	bash	zsh	fish
定位	通用，默认	高度定制	易用、美观、现代化
兼容性	POSIX 标准兼容	大多数兼容 bash	不兼容 bash，自成一套
上手难度	中等	中等偏高	非常简单
自动补全	仅有基础功能	需配合插件	开箱即用
语法高亮	没有	有插件	默认有
可定制性	较低	非常高	较高
脚本通用性	几乎全通用	高	低
资源占用	非常低	取决于插件	较低

表 5.1: 常见的 Linux/macOS Shell 对比

我们推荐使用 zsh 或者 fish。关于 zsh 怎么安装插件的问题，可以参考网上的各种教程，例如 Oh My Zsh 等。

对于 Windows，默认的终端是 CMD，其风格太老了，基本与现代开发脱节，不建议使用。我们建议使用 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。

注意

Windows 上自带的那个默认的“PowerShell”和我们从 MS Store 上安装的 PowerShell 不是一个东西。前者是 Windows PowerShell，基于 .NET Framework，版本最高为 5.1；而后者是 PowerShell Core，基于 .NET Core，版本从 6 开始。两者在语法和功能上有一些差异。我们推荐使用 PowerShell Core，因为它是跨平台的，并且得到了微软的持续支持和更新，而微软也推荐使用后者。

除了这些以外，如果不希望使用 Windows 的原生 shell，也可以使用诸如 Cygwin、MSYS2

等类 UNIX 环境，来获得类似 Linux 的终端体验。有些教程会让你使用 `git bash`，这东西是一个精简版 MSYS2，主要用于 Git 操作，功能非常有限，不建议长期使用。

5.2 怎样使用终端？

5.2.1 命令的基本结构

一般情况下，一条命令满足以下结构：

```
1 程序 [子命令] [选项] [对象]
```

其中，除了程序是必须的，其他的子命令、选项和对象都是可选的，选项和对象中有的顺序可以颠倒，有的则不行。例如，命令：

```
1 apt install git -y
```

这里的 `apt` 是一个程序（包管理器）；`install` 是一个子命令，表示安装软件包；`-y` 是一个选项，表示自动确认安装；`git` 是一个对象，表示要安装的软件包名称。上述命令中，子命令 `install` 和对象 `git` 的位置是不能颠倒的，而选项 `-y` 的位置则可以放在子命令和对象的前后。

5.2.2 终端的基本操作

需要注意的是，终端大多不支持鼠标点击，因此需要全程使用键盘输入命令，用方向键的左右键移动光标。方向键的上下键可以用来浏览历史输入过的命令，这对于重复输入相似命令非常有用。有时候，终端会帮我们提前补全一些命令（表现为半透明文字），我们只需要按下 `Tab` 键即可自动补全命令或者文件名。

有的命令执行时间很长，或执行时遇到了错误需要掐断。这时，我们可以按下 `Ctrl+C` 来终止当前命令的执行。而如果我们仅仅是希望暂停而非终止当前命令的执行，可以按下 `Ctrl+Z`，这会将当前命令暂停并挂起，然后你可以输入其他命令。要恢复暂停的命令，可以使用 `fg` 命令将其恢复到前台继续执行，或者使用 `bg` 命令将其放到后台继续执行。

有的时候，命令执行时间实在是太长了，我们不想等着这东西一直运行下去。这时，我们可以在命令后面加上一个 `&` 符号，表示让这个命令在后台运行，这样我们就可以继续输入其他命令了。

还有的时候，命令会输出大量的信息，我们不想让这些全部显示在屏幕上。这时，我们仅需要重定向其输出，也就是在命令的末尾加上 `> output.txt`，这样命令的输出就会被写入到一个名为 `output.txt` 的文件中，而不是显示在屏幕上。如果我们希望将输出追加到文件的末尾，而不是覆盖文件的内容，可以使用 `>>` 符号。这个 `output.txt` 可以是任意文件名。

有时候，我们从其他地方复制来了一些命令，或希望将命令的输出复制走。我们知道在终端中 `Ctrl+C` 是用来终止命令的执行的，这时我们该怎么办呢？这时，我们可以使用鼠标左键选中要复制的内容，然后按下 `Ctrl+Shift+C` 来复制选中的内容。要将内容粘贴到终端中，可

以按下 `Ctrl+Shift+V`。这一点是和普通的文本编辑器不同的。另外，虽然很多终端不支持鼠标点击光标，但也有一部分终端支持用鼠标选定一些内容并右键，以显示菜单来进行复制和粘贴操作。

下表5.2是一些常见的命令。

操作	Bash 等	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 5.2: Bash 和 PowerShell 的常用命令对比

当然，上述命令在 Windows 上并不常用。然而，如果想从事开发，终端反而会变得不可或缺，尤其是在使用诸如 Git、Docker 等工具时。另外，在希望对文件进行批量处理时，终端也会显得非常有用。

另，上述命令虽然在 Windows 上的全称看起来很吓人，但是 PowerShell 支持命令别名，例如 `ls` 实际上就是 `Get-ChildItem` 的别名，`cp` 是 `Copy-Item` 的别名，等等。因此，我们可以直接使用这些简短的命令。要是实在不行，那还是老老实实用 PowerShell 里的全称吧。

例如，“Windows 聚焦”每天都会更新新的图片，例如风景、动物等，并将其作为锁屏背景或桌面背景。这些图片被缓存在一个隐藏目录中，且没有扩展名。如果遇到特别喜欢的图片，可以把这些缓存提取出来，并批量加上扩展名：

```
1 Rename-Item -Path "Your\Path\Here\*" -NewName { $_.Name + ".jpg" } # Pwsh 风格
2 ren Your\Path\Here\* *.jpg # CMD 风格，一般 Pwsh 兼容
```

这样比起手动地一个个添加扩展名要方便得多。

5.3 常用终端命令行辞典

在表5.2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，这些基本上覆盖了我们日常使用终端时最常用的命令，大家记住其中的大多数命令就可以玩转终端了。

- 系统命令
 - `sudo` 命令用于提权。
 - `poweroff` 和 `shutdown` 两个命令用于关机。

- `reboot` 命令用于重启电脑。
- `whoami` 命令用于查看自己是哪个用户。
- `which` 命令用于查找可执行文件的路径。
- `ps` 命令用于显示当前运行的进程。
 - * `-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。
- `kill` 命令用于终止进程。
- `fg` 命令可以将后台运行的任务调回前台，这个命令可以恢复被 `Ctrl+Z` 挂起的任务。
- `bg` 命令可以将任务放到后台运行。

• 列出和查找类

- `pwd` 命令用于显示当前工作目录的绝对路径。
- `ls` 命令用于列出目录中的文件和子目录。
 - * `-l`：以长格式列出文件和目录的详细信息。
 - * `-a`：列出所有文件和目录，包括隐藏文件。
 - * `-h`：以人类可读的格式显示文件大小。
 - * `-R`：递归地列出子目录中的文件和目录。
 - * `-t`：按修改时间排序。
- `tree` 命令用于以树形结构显示目录中的文件和子目录。
- `find` 命令用于在目录中查找文件和目录。

• 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。
 - * `-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。
- `touch` 命令用于创建新文件或更新现有文件的修改时间。
 - * `-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。
 - * **-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。
 - * **-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。
 - * **-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。
 - * **-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。
 - * **-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。
- **文本处理类**
 - **head** 命令用于显示文件的前几行。
 - **tail** 命令用于显示文件的后几行。
 - **cat** 命令用于连接文件并打印到标准输出。
 - * **-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 `̃`。
- * -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。
 - * -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，`\t` 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。

警告

导啊，咱们生产环境为啥执行 `dpkg` 会说 `command not found` 呀？

我之前干了啥？清了一下工作路径垃圾，好像是 `sudo rm -rf /`

什么叫相对路径要加个点？

警告：除非你知道你在输入什么，否则任何情况下均不要带上 `sudo` 执行删除命令！

5.4 进阶：命令联动

在 Linux 中，重定向、管道、变量和进程替换是四种把“数据”从一条命令挪到另一条命令（或文件）的核心手段。它们常被混用，但机理各不相同。

5.4.1 重定向

重定向只认识真正的文件（或文件描述符），有两种：> 把标准输出定向到文件（写）；< 把文件内容定向到标准输入（读）。

```
1 echo "Hello, World!" > hello.txt # 新建或覆盖文件
2 cat < hello.txt                  # 把文件当输入
```

5.4.2 管道

管道 | 在内核里创建一条匿名管道，让左边进程的标准输出直接成为右边进程的标准输入，两边同时运行。

```
1 ls | grep "file" # 边 ls 边 grep，流式处理
```

5.4.3 Here-String

<<< word 是 Bash 的 here-string 语法，shell 会先把 word 的扩展结果写进一个临时文件（或匿名管道），再把该临时对象作为标准输入递给命令。因此它正好弥补了重定向只能读文件的不足。

```
1 grep "file" <<< "$(ls)" # 等价于 ls | grep "file"，但没用管道
```

Here-String 和管道有一定的区别。管道是流式的，边产生边消费；Here-String 必须等整个字符串生成完才能开始消费。

5.4.4 进程替换

进程替换是一种“把命令输出/输入伪装成文件名”的 Bash 特性。<(cmd) 把命令的标准输出绑定到一个命名管道（或 /dev/fd/N），返回一个可读文件名；>(cmd) 则把命令的标准输入绑定到一个命名管道，返回一个可写文件名。对任何“只能读文件”的工具（‘diff’、‘cat’、‘sort’…）来说，这就像凭空多了两个临时文件。

```
1 diff <(cmd1) <(cmd2) # 比较两条命令的输出，而无需临时落盘
2 sort >(uniq > result.txt) # 把排序结果直接丢给 uniq
```

5.4.5 变量与命令替换

\$(cmd) 是“命令替换”，shell 会等待该命令执行结束，把它的全部标准输出当成一段文本收回来，可以赋给变量，也可以直接嵌入命令行。

```
1 out=$(ls) # 把 ls 的输出存进变量
2 grep "file" <<< "$out" # 这里用 here-string 消费变量
3 diff <(echo "$out") <(ls) # 用进程替换再比一次
```

\$(cmd) 本身不是管道，也不是重定向。它只是“把命令输出变成字符串”的一种手段。掌握这些手法后，你可以根据各种实际条件，灵活选择最简洁、最高效的写法。同时，这些写法也组成了 shell 脚本的基础：shell 脚本也仅仅比上述指令多出了变量赋值、流程控制和函数定义而已！

5.5 进阶：更好的终端

默认终端的界面比较简陋，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** 和 **Oh My Zsh**。

机制	数据形态	左侧何时开始	右侧何时开始
cmd1 cmd2	管道字节流	立即	立即
cmd < file	已有文件	立即	-
cmd <<< "\$str"	临时文件	字符串生成完后	字符串生成完后
cmd <(cmd1)	命名管道/FD	立即	立即
str=\$(cmd1); cmd2 <<< "\$str"	变量 → 临时文件	cmd1 结束后	cmd1 结束后

5.5.1 Oh My Posh 及其配置

Oh My Posh (简称 OMP) 是一个跨平台的终端美化工具, 支持 Windows、Linux 和 macOS 等操作系统上的多个 shell (如 PowerShell、Bash、Zsh 等)。它提供了丰富的主题和插件, 可以帮助用户更好地使用终端, 提高工作效率。

提示

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:

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

就可以允许当前用户执行远程签名的脚本。

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:

```
1 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:

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

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

要配置 OMP, 首先应该安装 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:

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

安装文档: <https://ohmyposh.dev/docs/installation/prompt>
Nerd Fonts 官网: <https://www.nerdfonts.com/>

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。

为了保证 Oh My Posh 在每次启动 PowerShell 时都能自动加载，你需要将初始化命令添加到 PowerShell 的配置文件中。你可以在 PowerShell 中运行以下命令来打开配置文件：

```
1 notepad $PROFILE
```

然后在文件的末尾添加以下内容，并重新启动终端：

```
1 oh-my-posh init pwsh | Invoke-Expression
```

接下来，你可以在 PowerShell 中运行命令来设置 Oh My Posh 的主题了。[其他主题及其安装方法](#)可自行查阅。例如使用 `powerlevel10k_classic` 为主题：编辑 Powershell 配置文件，在文件末尾编辑 `oh-my-posh init pwsh --config 'powerlevel10k_classic' | Invoke-Expression`，重启终端后即可。

5.5.2 Oh My Zsh

Oh My Zsh 是一个开源的、社区驱动的框架，用于管理 Zsh 配置，一般简称为 omz。它提供了大量的插件和主题，可以帮助用户更好地使用 Zsh，提高工作效率，其官网为ohmyz.sh。之所以使用 omz，主要是因为 zsh 本身就是大多数人在 Linux 和 macOS 上使用的默认 shell，功能强大且易于定制，而 omz 作为 zsh 的专用配置框架，显然比 omp 更适合 zsh。

安装、配置等内容请参考官网。

omz 最令人眼前一亮的是其插件系统，内置了大量实用的插件，可以极大地扩展 zsh 的功能。我个人非常喜欢自动补全和语法高亮两个插件。而著名的 git 和 z 我个人反而很少用，主要是因为别处已经彻底习惯用 git 命令的正常形式了，连图形化界面都几乎不用；而 z 我个人觉得没有必要，毕竟我用得最多的目录也就那么几个，结合 KDE 桌面和 Dolphin 文件管理器完全足够快速访问。

除此以外，starship 也是一个非常不错的跨平台终端美化工具，支持 Bash、Zsh、Fish 等多种 shell。其官网为starship.rs。安装和配置也非常简单，推荐一试。

5.5.3 使用其他工具

默认的软件（如 ls、cat、top 等）功能比较简单，界面也比较丑陋。为了让终端更美观、更实用，我们可以使用一些替代工具，这里推荐几个我个人常用的：

- 用 fd 替代 find：fd 是一个快速且用户友好的文件查找工具，支持正则表达式和模糊匹配等功能，使用起来比 find 更加简单和高效。
- 用 ripgrep 替代 grep：ripgrep 是一个快速且高效的文本搜索工具，支持正则表达式和多线程搜索等功能，使用起来比 grep 更加简单和高效。

其他主题：<https://ohmyposh.dev/docs/themes>

其安装方法：<https://ohmyposh.dev/docs/installation/customize>

- 用 bat 替代 cat：bat 是一个增强版的文本查看工具，支持语法高亮、行号显示等功能，使用起来比 cat 更加美观和易读。
- 用 htop 替代 top：htop 是一个交互式的进程管理工具，支持彩色显示、进程排序等功能，使用起来比 top 更加直观和易用。
- 用 exa 替代 ls：exa 是一个现代化的文件列表工具，支持彩色显示、图标显示等功能，使用起来比 ls 更加美观和易读。

当然，这些工具都需要单独安装。

第六章 开始使用 Linux

刚刚我们学习了怎样使用终端。其实会用了终端，那么我们对 Linux 的使用就已经会了一大半了。接下来，我们来看看 Linux 这个操作系统本身。

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。而另一方面，Linux 在配置编程和开发环境上也有着得天独厚的优势：Linux 用户不仅不需要像 Windows 用户一样和 Windows 复杂的内核做斗争，也不需要像 macOS 用户一样被苹果的生态圈所束缚，其自由和开放使得用户可以轻松地配置和定制自己的开发环境。

要学习 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

那么，我们开始吧！

阅读材料

在 20 世纪 80 年代，主要的几个计算机系统有 UNIX、DOS 和 Mac OS。当时，UNIX 系统昂贵且无法用于个人计算机，DOS 简陋且闭源，Mac OS 则只用于苹果电脑，计算机科学教育严重受限。为了解决这个问题，Andrew S. Tanenbaum 教授设计了 MINIX 系统，并将其用于教学。然而该系统功能有限且不实用。

当时在芬兰赫尔辛基大学读大二的 Linus Torvalds 在用过 MINIX 之后，受到了启发。在 1991 年，他利用 UNIX 的核心，吸收了 MINIX 的精华，剔除了不必要的部分，编写了一个新的操作系统内核，并将其命名为 Linux 0.01，该内核能够运行在 x86 架构的个人计算机上。这就是后来各种 Linux 的雏形。1994 年，Linux 1.0 版本发布，标志着 Linux 的正式诞生。

Linux 内核采用了 GPL 协议，这使得任何人都可以自由地使用、修改和分发它。这种开放的理念吸引了大量的开发者和用户，形成了一个庞大的社区。如今，Linux 已经与 Windows、macOS 成三足鼎立之势，成为全球最流行的操作系统之一，更成为了开源技术的象征。

6.1 Linux 发行版及其选择

虽然我们经常说“Linux 系统”，但是实际上 Linux 并不是一个操作系统，它仅仅是一个内核（Kernel）。一个完整的操作系统还需要很多其他的组件，例如文件系统、图形界面、应用程序等。为了方便用户使用，很多组织和公司将 Linux 内核和其他组件打包在一起，形成了一个完整的操作系统，这就是我们常说的 Linux 发行版（Distribution）。Linux 发行版种类繁多，每个发行版都有其独特的特点和适用场景。常见的 Linux 发行版有：

表 6.1: 常见 Linux 发行版

发行版	特点	更新频率	适用情况
Ubuntu	用户友好，社区活跃	两年	Linux 新手，桌面用户
Debian	稳定，软件包丰富	两年	服务器等生产环境
Fedora	最新技术，社区驱动	半年	开发者，技术爱好者
RHEL	商业支持，稳定	三年	企业用户，服务器
CentOS	RHEL 的免费版本	死了	企业用户，服务器
Arch	滚动更新，极简	以天计	高级用户，DIY 爱好者
NixOS	声明式配置，原子升级	以天计	高级用户，系统管理员
Rocky	Cent 的复活版本	三年	企业用户，服务器
Alma	Cent 的复活版本	三年	企业用户，服务器
Mint	基于 Ubuntu，用户友好	两年	Linux 新手，桌面用户
Manjaro	基于 Arch，用户友好	以月计	Linux 新手，桌面用户
openSUSE	稳定，企业支持	八个月	企业用户，服务器
Gentoo	源码编译，极致定制	以月计	高级用户，DIY 爱好者
Kali	安全测试，渗透测试	半年	网络安全专业人员
Alpine	轻量级，安全	以月计	容器，嵌入式系统
统信 UOS	中国本土，兼容 Windows	两年	政府，企业用户

提示

现在的 CentOS 实际上已经被 RHEL 官方收编并停止维护了，CentOS 被收编后的名称叫做 CentOS Stream，定位是 RHEL 的上游测试版。

另一方面，Alma 和 Rocky 虽然均声称是 CentOS 的复活版、RHEL 的复刻、两者行为基本一致，但也有细微的差别：Alma 兼容 RHEL 的 ABI^a，声称其软件通用，但不承诺代码完全一致，因此其补丁有些时候甚至比 RHEL 还快 24 小时；Rocky 则是 CentOS 的“精神继承者”，承诺与 RHEL 完全一致，但获取源码的路径比较曲折。

^aABI：Application Binary Interface，应用二进制接口，是指程序在运行时与操作系统之间的接口规范。

上述发行版是相对常见的 Linux 发行版。其中，最出圈的发行版莫过于 Ubuntu 了，有不少人甚至直接将 Ubuntu 和 Linux 划等号（这实际上是不对的）。Ubuntu 拥有着庞大的用户群体和丰富的软件资源，非常适合初学者入门。

对于什么都不会的小白而言，我们推荐使用 Ubuntu、Mint、Manjaro 等用户友好的发行版；如果你愿意折腾，可以尝试 Arch、Gentoo 等极简主义的发行版；如果你要玩服务器，可以选择 Debian、RHEL、CentOS（或其复活版本 Rocky、Alma）等稳定的发行版；如果你对

网络安全感兴趣，可以尝试 Kali 等专用发行版。

提示

Linux 大致可以分为以下几个“家族”：

- Debian 系：包括 Debian、Ubuntu、Mint 等，特点是稳定，软件包丰富，适合新手和服务器。
- Red Hat 系：包括 Fedora、RHEL、CentOS、Rocky、Alma 等，特点是商业支持，稳定，适合企业用户和服务器。其软件生态又以 RHEL 为核心，因此也叫做“红帽生态”，具体流程是：Fedora 激进上游 → CentOS Stream 中游测试 → RHEL 稳定版商业支持 → Rocky/Alma 下游免费替补。
- Arch 系：包括 Arch、Manjaro 等，特点是滚动更新，极简，适合高级用户和 DIY 爱好者。
- 其他系。

这些系在软件包管理上，采取了不同的策略：

- Debian 系使用 APT 包管理器，软件包格式为 DEB；
- Red Hat 系使用 YUM 或 DNF 包管理器，软件包格式为 RPM；
- Arch 系使用 Pacman 包管理器，软件包格式为 Tar.xz。

但是，不同系的发行版实际上最大的区别仅在于包管理器和默认配置上，其他方面并没有太大区别，终端命令行等也完全相似，因此我们并不需要过于纠结。

6.1.1 CLab

这是最简单的使用 Linux 的方式，甚至不需要在计算机上安装任何东西。我们只需要去 [CLab](https://clab.pku.edu.cn/) 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，这个虚拟机的性能和功能有限，因此一般用户无法在上面运行诸如 Minecraft 服务端等大型软件。

6.1.2 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，安装过程参考各发行版的官方文档和本书前面的安装章节即可。

6.1.3 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）。

6.1.4 WSL

这里仅提一嘴，具体的使用见 [6.5](#) 节。

6.2 Linux 的基本操作

得了，来了啥也别说，先跑个小火车：

```
1 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，展示了 ASCII 艺术的魅力。这个程序最初是作为一个恶搞程序出现的，目的是为了提醒用户注意输入错误的命令（因为很多人会把 `ls` 错误地输入为 `sl`）。不过，随着时间的推移，`sl` 逐渐成为了一个有趣的终端玩具，受到了许多终端爱好者的喜爱。

拆解这一行命令：

- `sudo` 是“以超级用户身份运行命令”的意思。因为安装软件包需要修改系统文件，所以需要管理员权限。执行这个命令时，系统会提示输入当前用户的密码，以验证权限。
- `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。
- `install` 是 `apt` 的一个子命令，用于安装软件包。
- `-y` 是一个选项，表示自动回答“是”以确认安装过程中的提示。这样可以避免在安装过程中需要手动输入确认。
- `sl` 是要安装的软件包的名称。
- `&&` 是一个逻辑运算符，表示如果前面的命令成功执行了，那么就执行后面的命令。
- 最后的 `sl` 是运行刚刚安装好的小火车程序的命令。

我们发现，这个命令和我们在上一章中接触到的那些命令遵循相同的结构。

如果我们用的不是 Debian 系的 Linux 发行版，而用的是其他发行版，则需要使用相应的包管理器，例如 Red Hat 系使用 `yum` 或 `dnf`，Arch 系使用 `pacman`。以 Arch 为例：

```
1 sudo pacman -S sl && sl
```

让我们看看上面内容是怎么发生的。我们刚刚输入的单条命令依然是这样的形式：

```
1 程序 [子命令] [选项] [对象]
```

我们用 `apt install -y sl` 来举例说说：`apt` 是程序，`install` 是子命令，`-y` 是选项，`sl` 是对象。于是我们就能看到，一个程序就这么跑起来了，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，例如在 `apt` 中，我们可以使用 `apt search < 关键词 >` 来搜索软件包；在 `pacman` 中，我们可以使用 `pacman -Ss < 关键词 >` 来搜索软件包；而 `yum` 和 `dnf` 中，我们可以使用 `yum search < 关键词 >` 或 `dnf search < 关键词 >` 来搜索软件包。

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，安装方法和 `sl` 类似。

6.3 Linux 的文件系统

好的，我们刚刚已经基本知道怎么使用 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。

6.3.1 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，所以一个文件的文件名如果被创建为 `dir/file`，那么它就表示在根目录下的 `dir` 目录下有一个名为 `file` 的文件，而不是一个名为 `dir/file` 的文件。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，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，这是为了安全考虑。我们可以使用 `echo $PATH` 来查看当前的 `PATH` 变量。假如我们把 `hello` 放在 `/usr/local/bin` 目录下，那么我们就可以在任何地方运行它了，这是因为上述目录通常会被包含在 `PATH` 变量中。

6.3.2 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，一般用字母表示为 `r`、`w`、`x`。我们使用权限位来表示这三种操作。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：一个用户创建了一个文件，这个文件对他的权限是 `rwX`，对同组用户的权限是 `r-x`，对其他用户的权限是 `r-`。那么这个文件的权限就是 `rwXr-xr--`。而这个代表方式还是太笨重了，于是 Linux 又引入了数字表示法。

数字表示法是一个“独热编码”（One-hot Encoding），也就是 `rwX` 被看作三个相互独立的二进制位，对应位上 1 表示有该权限，0 表示没有该权限。然后把这三个位的值拼成一个三位的二进制数，就代表了其最终权限。例如：某用户对文件有读、写权限，没有执行权限，即 `rw-`，对应的二进制数是 110，转换成十进制就是 6。而二进制中的 100 是 4，010 是 2，001 是 1，于是就可以把一个用户对文件的权限表示为一个 0 到 7 的数字，例如 `7=4+2+1` 表示 `rwX`，`5=4+0+1` 表示 `r-x`，等等。

因此我们的文件权限可以用一个三位八进制来表示，从左到右的每一位分别对应所有者、组和其他用户的权限。例如，权限为 `rwXr-xr--` 的文件可以表示为 754。另一个常见的权限编码是 644，你能说明其含义吗？¹

但有一个用户比较特殊，那就是 `root` 用户。`root` 用户是系统管理员，拥有对所有文件的最高权限。无论文件的权限如何设置，`root` 用户都可以对其进行读、写、执行操作；换句话说，`root` 用户对所有文件的权限都是 `rwX` 或者 7。

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

¹其含义是：所有者有着读写权限，组和其他用户只有读权限。

提示

修改文件权限需要有相应的权限，否则会报错。例如你无法给一个对你来说权限是 0 的文件加上可执行权限。

警告

不要执行这类抽象的命令：`chmod 777 /`，这会导致系统所有成员全部被视同 `root` 用户，进而导致系统无法正常工作。

这样就会导致所有用户都可以写根目录，随便替换 `/bin/sh`、`/etc/passwd`、`usr/bin/sudo` 等关键文件，乃至随意植入 `cron`、`systemd` 等定时任务，等同于整个安全阵地完全失守，自毁长城。

常见的权限标识符包括 755、644、750 等，这些标识符的含义都很简单，我这里就不详细展开描述了。

而不使用标识符的情况也是可以的，例如某 `python` 文件需要被执行，那么我们可以使用 `chmod +x script.py` 来给它加上可执行权限，这里的 `+x` 表示“加上可执行权限”。类似地，`-x` 表示“去掉可执行权限”，`+r` 表示“加上读权限”等。

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！

6.3.3 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。

提示

试一试：

```
1  echo "hello" > a # 随便创建一个文件
2  ln a b          # 创建硬链接 b 指向 a
3  ln -s a c       # 创建软链接 c 指向 a
```

现在这里有了 `b` 和 `c` 两个新的文件。试着利用 `ls -l` 命令查看它们的属性，看看有什么区别。然后尝试删除 `a` 文件，看看 `b` 和 `c` 会发生什么变化。

6.4 Linux 的进一步使用

6.4.1 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，如果需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）：

```
1 yourusername@yourcomputer$ su
2 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。

6.4.2 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：

```
1 sudo apt-get update
2 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：

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

如果想要卸载软件包，可以使用以下命令：

```
1 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：

北大开源镜像站的帮助文档: <https://mirrors.pku.edu.cn/Help>

```
1 sudo dpkg -i <package-name>.deb # 安装 .deb 包
2 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。

6.4.3 关于 VIM 和 Nano

虽然我们有很多文本编辑器可以在图形化的 Linux 上使用，但是在终端中，这两个编辑器依然是最常用的，而且也是不得不用（尤其是 VIM!）。因此，我将会在这里简要介绍一下这两个编辑器。

VIM

VIM 是一个强大的文本编辑器，它有着丰富的功能和插件，可以满足各种需求。VIM 有两种状态：命令状态和编辑状态。默认情况下，VIM 处于命令状态。在命令状态下，我们可以使用各种命令来操作文件；在编辑状态下，我们可以直接输入文本。在这两种方式之间的切换非常简单，只需要按下 `i` 键即可进入编辑状态，按下 `Esc` 键即可返回命令状态。

编辑状态下的 VIM 乏善可陈，完全可以把这玩意当成一个没有鼠标的 Windows 记事本来使用，这里根本没有什么好说的。我们的重点在于命令状态下的 VIM。在命令状态下，我们可以使用各种命令来操作文件。以下是一些常用的命令：

• <code>h</code> ：向左移动光标。	• <code>dd</code> ：删除当前行。
• <code>j</code> ：向下移动光标。	• <code>yy</code> ：复制当前行。
• <code>k</code> ：向上移动光标。	• <code>p</code> ：粘贴。
• <code>l</code> ：向右移动光标。	• <code>u</code> ：撤销。
• <code>w</code> ：移动到下一个单词的开头。	• <code>Ctrl+r</code> ：重做。
• <code>b</code> ：移动到上一个单词的开头。	• <code>:w</code> ：保存文件。
• <code>0</code> ：移动到行首。	• <code>:q</code> ：退出 VIM。
• <code>\$</code> ：移动到行尾。	• <code>:wq</code> ：保存并退出 VIM。
• <code>gg</code> ：移动到文件开头。	• <code>:q!</code> ：强制退出 VIM，不保存文件。
• <code>G</code> ：移动到文件结尾。	

想给 Vim 装插件其实也是一种挺折腾的工作。目前最通行的方式是利用插件管理器，比如 Vundle、Pathogen、vim-plug 等。安装插件管理器后，我们可以通过编辑 `/.vimrc` 文件来添加插件，具体也可以参考各插件管理器的文档。而使用 NeoVim 等衍生版本则会更方便一些。

Nano

Nano 是一个简单易用的文本编辑器，它有着直观的界面和快捷键，可以快速上手。其使用也比 VIM 简单得多，因为它没有两种状态的区别，直接打开文件后就可以编辑文本了，而且所有命令快捷键全都在界面下方列出了，直接照着按就行了！

- Ctrl+O：保存文件。
 - Ctrl+X：退出 Nano。
 - Ctrl+K：剪切当前行。
 - Ctrl+U：粘贴。
- Ctrl+W：查找文本。
 - Ctrl+\：替换文本。
 - Ctrl+C：显示光标位置。
 - Ctrl+G：显示帮助信息。

6.5 WSL 速成指南

WSL，或 Windows Subsystem for Linux，是微软为 Windows 10 及更高版本用户提供的功能，允许用户在 Windows 上运行 Linux 环境，而无需使用虚拟机或双系统。其中，WSL1 和 WSL2 又是两个不同的东西，WSL1 仅使用了一个兼容层，把 Linux 的系统调用翻译成 Windows 的系统调用；而 WSL2 则使用了一个完整的 Linux 内核，提供了更好的兼容性和性能，兼容性近乎完美。

现在是 2025 年，我们现在指的 WSL 指的几乎都是 WSL2。其极度轻量，启动速度几块，占用内存极低，估计和一个浏览器标签页差不多；甚至能和 Windows 共用显卡、网络、文件系统，复制粘贴甚至都随便互通。虽然说 WSL 不是一个完整的 Linux 系统，但对于大多数人而言，WSL 确实是最好的 Linux 使用方式。

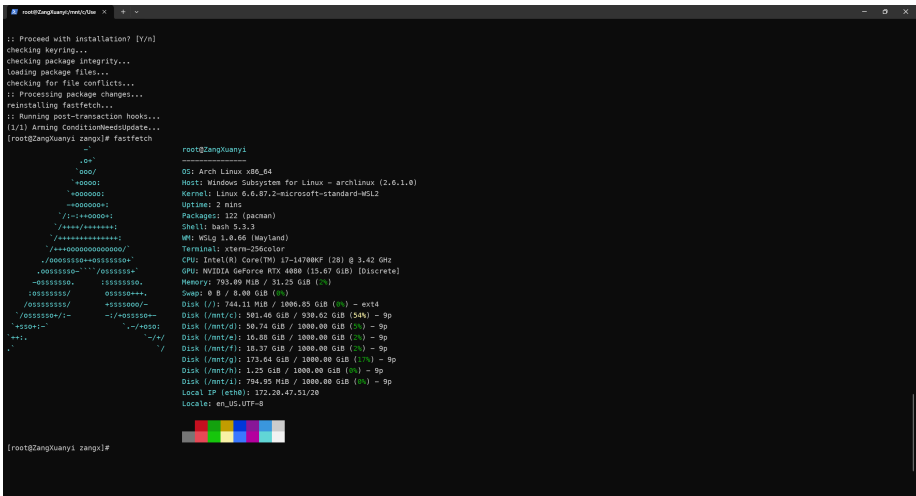


图 6.1: WSL 中的 ArchLinux（fastfetch）

6.5.1 快速安装

管理员权限在 PowerShell 里运行：

```
1 wsl --install
```

初次运行，微软会自动打开 WSL 和虚拟机平台功能，并提示重启电脑。重启后，WSL 会自动下载并安装 Ubuntu 发行版（默认最新 LTS，也可以手动指定发行版或版本号）。重启后第一次弹出 Ubuntu 窗口，输入用户名和密码即可。之后就可以愉快地使用 Linux 了。

怎么验证？只需要输入

```
1 wsl -l -v
```

看到 Version 列显示 2 即可。

关于 Win10 这种老系统²，情况稍有不同，得先干点别的，具体可以参考微软[官方文档](#)，或者网上的各种教程。

6.5.2 换发行版

如果你不喜欢 Ubuntu，可以安装别的发行版。微软商店里有很多发行版可供选择，例如 Debian、Kali Linux、openSUSE、Fedora 等。只需要打开微软商店，搜索“WSL”，然后选择你喜欢的发行版进行安装即可。或者直接用命令行：

```
1 wsl --list --online # 列出可用发行版
2 wsl --install -d <name> # 安装指定发行版
3 wsl --set-default <name> # 设置默认发行版
```

安装完成后，运行

```
1 wsl -d <name>
```

即可进入该发行版的 Linux 环境。或者在开始菜单找到该发行版的图标，点击即可启动，多个发行版也并不互相影响。

6.5.3 文件互相访问

WSL 和 Windows 可以互相随意地访问文件系统，这是 WSL 往往比虚拟机或双系统好用的一个重要原因。

在 WSL 的 Linux 环境中，Windows 的文件系统挂载在 /mnt/c（C 盘）、/mnt/d（D 盘）等目录下。你可以通过这些目录访问 Windows 的文件。例如：

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

²在 2025 年 10 月 14 日，Win10 正式停止支持。

而从 Windows 访问 WSL 的文件系统，则可以通过路径 `\\wsl$\\$<distro-name>` 来访问，上述目录是 WSL 的根目录，可以直接拖拽文件出入，或用 VS Code 等编辑器打开。

6.5.4 一口气配好开发环境

下一步就是和 Linux 一样的开发环境配置：

```
1 # 换国内源（清华）+ 更新
2 sudo sed -i 's@http://.*.<-URL->@<-URL->' /etc/apt/sources.list
3 sudo apt update && sudo apt upgrade -y
4
5 # C/C++ 开发必装三件套
6 sudo apt install build-essential gdb cmake ninja-build
```

关于上述那个 `sed` 命令，具体的 URL 可以参考清华大学开源软件镜像站的[帮助页面](#)。

对于 VS Code，怎么让它调用 WSL 以及里面的工具呢？只需要在 Windows 端下安装 VS Code 的 WSL 扩展，然后 `Ctrl+Shift+P`，连接到 WSL，下面的内容就和 Linux 主机或者在 docker 里开发一模一样。Python、Rust、Golang 这堆也同理，包管理器一条就装好，清清爽爽，比先前我们在 Windows 里折腾环境简单多了！（实际上我们提到过的 MSYS2，也是在 Windows 里模拟 Linux 环境，但 WSL 更彻底一些，且更贴近实际 Linux 使用体验。）

6.5.5 图形界面应用

WSL2 的另一个优势是图形界面随使用。这个是基于 WSLg 的，在 Windows 的较新版本已经内置。比方说经典 Linux 记事本 `gedit`，直接在 WSL 里安装并运行：

```
1 sudo apt install gedit
2 gedit test.txt
```

窗口直接弹出，和 Windows 应用无异。甚至连剪贴板都能互通，复制粘贴完全没问题。这个背后用的是微软自家的 RDP 协议，性能和兼容性都不错，而用户实际上几乎无感。JB 全家桶、Firefox 乃至 `qemu-kvm` 都能跑图形界面应用，体验极佳。

6.5.6 性能调优、踩坑急救

一般可能遇到以下问题：

- apt 龟速：换源，见上文。当然也可能是没有 `sudo apt update` 导致的。
- 内存³爆炸：WSL 默认分配给 Linux 的内存是无限限制的，会根据需要动态增长，但不会自动释放。对此，你需要建立一个文件：`~/.wslconfig`，内容如下：

³指的是 RAM！

```
1 [wsl2]
2 memory=4GB # 限制最大内存为 4GB, 当然一般会更大一些
3 processors=2 # 限制使用 2 个 CPU 核心
4 swap=2GB # 限制交换分区为 2GB
```

保存后, `wsl --shutdown` 重启子系统解决问题。

- C 盘爆红: 如果你能确定你的 C 盘因 WSL 而爆红, 可以执行:

```
1 wsl --export <distro-name> distro.tar
2 wsl --unregister <distro-name>
3 wsl --import <distro-name> <new-location> distro.tar
```

把 WSL 的虚拟磁盘搬到别的盘去。上述三行命令的意思分别是, 导出一个镜像文件、注销当前发行版、从镜像文件导入到新位置。

- 网络异常: WSL 的网络是虚拟的 NAT 网络, 有时会出问题。这个问题确实相当常见且棘手。如果希望给自己的 3000 端口暴露给局域网, 需要在 Windows 端运行:

```
1 netsh interface portproxy add v4tov4 listenport=3000 listenaddress=0.0.0.0
   ↪ connectport=3000 connectaddress=<WSL_IP_Address>
```

其中 `<WSL_IP_Address>` 可以通过 `ip addr` 命令查看。倘若网络完全异常, 可以尝试重置 WSL 网络:

```
1 netsh winsock reset
2 netsh int ip reset all
3 ipconfig /release
4 ipconfig /renew
5 ipconfig /flushdns
```

然后重启电脑。

6.5.7 进阶玩法

比方说, 可以在 Windows 命令里面直接调用 linux 命令:

```
1 wsl ls -la /home/user
```

或者在 Linux 里直接调用 Windows 命令 (但是一般人估计不会这么干):

```
1 cmd.exe /C dir C:\Users\YourUsername
```

WSL 还支持 `systemd` 服务, 2025 年微软终于正式支持了这个功能:

```
1 sudo apt install systemd-genie
2 genie -s # 进入 systemd 环境
```

然后就可以愉快地使用 `systemd` 服务了，如 `docker` 等。

还可以一键备份和还原 WSL 发行版：

```
1 wsl --shutdown
2 wsl --export <distro-name> backup.tar
3 # 这里需要重启计算机以确保 WSL 完全关闭
4 wsl --unregister <distro-name>
5 wsl --import <distro-name> <new-location> backup.tar
```

于是原地复活一个一模一样的 WSL 发行版。

6.5.8 WSL 的局限性

虽然上述功能已经足够强大，但 WSL 毕竟不是一个完整的 Linux 系统，仍然存在一些局限性，主要聚焦于**硬件**。也就是说：

- 要真正去跑一个原生的硬件驱动，例如 WiFi 渗透、显卡直通等，WSL 是做不到的，还是得上真机或虚拟机。
- 对于内核模块玩得很深的用户，甚至自编驱动的高级用户，WSL 也不适合，它仅仅是微软定制的一个内核，签名是被锁死的，无法随意更改内核模块。

但除了以上两点，对于绝大多数用户而言，WSL 确实是“最好的 Linux 系统”。

对于 WSL 的进一步了解，可以参考微软的[官方文档](#)，以及网上的各种教程。

“太长不看”？那记住以下三行：

```
1 wsl --install
2 wsl
3 sudo apt install sl && sl
```

小火车跑起来，你就拥有了一台“开机即用”的 Linux 系统！

第四部分

开发环境：让代码能编译能跑

第七章 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。一个良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，最终是也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。

7.1 编程语言初探

阅读材料：编程语言发展简史

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。

同学们可能会疑惑：一段代码充其量只是一堆文本，为什么它能变成一个程序，运行在计算机上呢？

这是编译器或解释器在起作用。编译器将高级语言代码转换为机器码，生成可执行文件；

而解释器则逐行解释执行代码。无论是编译型语言还是解释型语言，最终都是通过某种方式将代码转换为计算机能够理解和执行的形式。所以说，可以执行的程序其实是代码经过编译或解释后的产物，没有编译器或解释器这个中间环节，代码是无法直接运行的。所以学习编程的第一步，就是要安装好并学会使用编译器或解释器，并配置好相关开发环境。本文将会以 C++ 和 Python 为主要例子，介绍如何搭建编程环境和使用相关工具。

这一章和第二章的要求一致：本章中如果有任何你不理解的指令，**请先照做**，不要自作主张更改步骤，以免导致之后不必要的麻烦。随着你对计算机的了解加深，你会逐渐理解为什么让你这么做。另，mac 用户请自行查找对应的配置方法，笔者并不拥有 mac 设备，无法提供相关支持。

注意

如果你是信科的同学，或将来有志于从事相关研究工作（如计算机视觉、自然语言处理、数据分析等），请务必看完本章内容，因为这些工作对编程环境有更高的要求，而信科的课程则天天和代码打交道，所以学会配置环境更是非常必要的。

如果你仅是为了应付课程，则不必完整地阅读本章，直接使用我下文提供的环境：

- C/C++：用 DevC++。这是一个轻量级的、仅支持 C/C++ 的 IDE，开箱即用，省去了一切配置烦恼，非常适合初学者使用。DevC++ 扩展性较差，无法满足复杂的开发需求，且原版已经多年未更新，建议使用社区维护的新版。
- Python：用 Python 自带的 IDLE。IDLE 是 Python 自带的轻量级 IDE，开箱即用，适合初学者使用。IDLE 功能有限，无法满足复杂的开发需求，但对于入门学习已经足够。
- Python：或者也可以用 PyCharm，和学校机房的环境保持一致。问题是臃肿，且需要注册账号使用社区版。
- 通用：下载 VS Code 但不配置任何环境，直接搜索插件：Code Runner，然后安装该插件即可。Code Runner 插件可以让 VS Code 支持多种编程语言的运行，包括 C/C++ 和 Python 等，且无需额外配置环境，非常适合初学者使用，但同样面临功能有限的问题。下载 VS Code 的方式请参考下文。

7.2 IDE 及其选择

7.2.1 为什么选择 VS Code?

IDE（集成开发环境，Integrated Development Environment）是一种软件应用程序，提供了编写、调试和测试代码所需的各种工具和功能。IDE 通常包括代码编辑器、编译器或解释器、调试器、版本控制系统等组件，旨在提高开发效率和代码质量。IDE 可以简单地被归类为通用的和专有的两种。

在 PKU 大一的课程《计算概论》上，一般会推荐使用 Visual Studio、DevC++ 和 PyCharm 这三种。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，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。

说明

Visual Studio 把一个工作目录视为一个“项目”，当用户使用编译功能时会把整个项目编译一遍。而在同一个项目中，不能同时存在多个同名的变量和函数，因此即使我们在 VS 中新建了不同的 C++ 文件且他们在逻辑上并无任何关联，但是 VS 依然会把它视为同一个项目的一部分，从而导致命名冲突的问题。对于初学者来说，这无疑是一个巨大的坑，往往表现为“我这个 main 函数怎么出现了重定义的错误?”。

另外，VS 出于安全性考量，禁用了 C 系的一部分函数，例如 `printf`。当我们试图对这些函数进行调用时，VS 会报错并提示我们使用更安全的版本，例如 `printf_s`。这无疑给初学者带来了不必要的困扰。当然这个困扰并非不能解决，只需要在文件开头定义宏 `#define _CRT_SECURE_NO_WARNINGS` 即可，但这无疑增加了学习的难度。另外，`printf_s` 等并不是标准函数，这会导致代码的可移植性变差。

综上所述，我非常建议新手远离 VS，即使是简陋的 DevC++ 也比 VS 强。每一年都会有大量的同学因为使用 VS 而陷入困境，浪费了大量的时间和精力。

为了解决臃肿的问题，我们一般使用轻量的文本编辑器来编写代码，并用自己额外安装的编译器或解释器来编译运行代码。这样做的好处是可以根据自己的需求选择合适的工具，避免了臃肿和不必要的功能，同时也提高了灵活性和可定制性。常见的文本编辑器包括 Vim、Visual Studio Code、notepad++ 等。

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。技术人中有一个非常著名的笑话：“怎么生成一段随机的字符串？答：只需要让不会用 VIM 的人试着退出 VIM 就可以了！”

提示

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。纵然如此，我个人还是建议同学们学着使用一下 vim 这个经典 TUI 编辑器，这是因为在将来的开发中，我们或多或少都会面对一些情况：不得不从远端登录某机器，且这个机器甚至还不能安装一些诸如 nano 的更现代的编辑器！

笔者个人非常推荐 Visual Studio Code（简称 VS Code 或者 Code），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。VS Code 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。

注意

VS Code 和 Visual Studio Community 两个产品都是微软公司开发的**免费**软件，不要点进某些广告网站下载收费版本！请务必从[VS Code 官网](#)和[VS 官网](#)下载对应的最新版本，以避免下载到捆绑恶意软件的盗版软件！

VS 也有付费版本，叫做 Visual Studio Professional 和 Visual Studio Enterprise，分别面向专业开发者和企业用户，提供了更多高级功能和支持服务，但对于大多数个人开发者和学生来说，社区版完全足够使用。

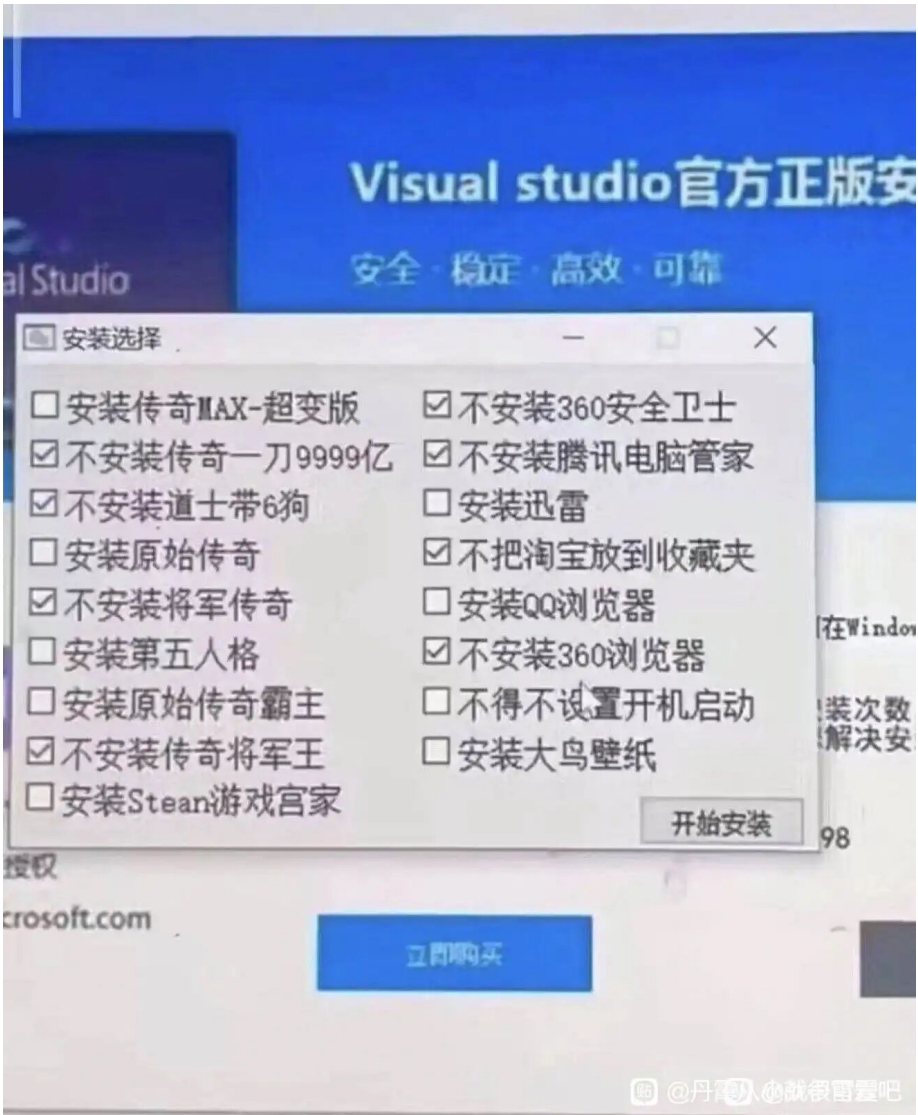


图 7.1: 假货

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 反而可能会有些不便，但到时候再去选择合适的 IDE

也不迟。

7.2.2 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。

在安装之前，我们应当确定自己计算机的 CPU 架构是 x86-64 还是 arm64。在 Windows 系统下，我们可以使用 `systeminfo` 指令来查看系统信息，或者右键点击“此电脑”选择“属性”来查看系统类型。在官网上下载的 Code 架构应当和操作系统的架构一致。

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。

7.2.3 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。我们可以在不同的工作区（可以简单地理解为工作用文件夹）启用和禁用不同的插件，以实现其高度可定制特性。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。

7.2.4 VS Code 的一些设置项

VS Code 的不少有用的设置项都隐藏在设置菜单中且默认关闭，很多人用了许久都不知道这些功能的存在：甚至 VS Code 的设置菜单都不是很容易发现（需要点击左下角齿轮图标，然后选择“设置”）。因此，下面介绍几个比较实用的设置项。

设置页面的搜索栏可以用来快速定位设置项。

内联提示

对于 Python 等语言，其变量类型是动态的，因此我们在编写代码时（尤其是有许多包的时候），往往很难确定某个变量的类型是什么。VS Code 提供了内联提示功能，可以在代码中直接显示变量的类型信息，帮助我们更好地理解代码。

启用方式：在设置界面搜索 `inlay Hints`，然后往下翻，找到你喜欢的语言对应的内联提示选项并启用它们即可。

随系统主题切换颜色

有时候，我们的系统主题会根据时间自动切换深色和浅色模式。如果 VS Code 的主题不能随系统主题切换，则会显得格格不入，手动调整也很麻烦。VS Code 提供了一个选项，可以让主题随系统主题自动切换。

启用方式：在设置界面搜索 color theme，找到这一项：window.autoDetectColorScheme，然后启用它即可。同时，你还需要在上面指定 workbench.preferredDarkColorTheme 等四个选项，来指定深色和浅色主题以及对应的高对比度主题。

我个人推荐 GitHub 系和 Ayu 系的主题。当然也可以使用“现代深色”，这是 VS Code 的默认主题，美观大方，但有点太黑，对应的“现代浅色”配色又不如 GitHub Light 好看。喜欢纯黑色的可以使用 Absolute Black 主题。

搜索、替换和正则替换

VS Code 的搜索和替换功能支持正则表达式，可以帮助我们更高效地进行文本处理。

首先，搜索和替换功能可以在左边的搜索按钮找到，这个搜索是针对整个工作区的搜索。或者也可以使用快捷键 Ctrl + F 或 Ctrl + H 来打开搜索和替换面板，该面板仅会搜索当前打开的文件。

在搜索和替换面板中，有一个小的正则表达式图标（类似于 .*），点击它可以启用正则表达式模式。启用后，我们就可以使用正则表达式来进行搜索和替换了。在该正则表达式中，我们可以用括号来捕获分组，然后在替换文本中使用 \$1、\$2 等来引用这些分组。

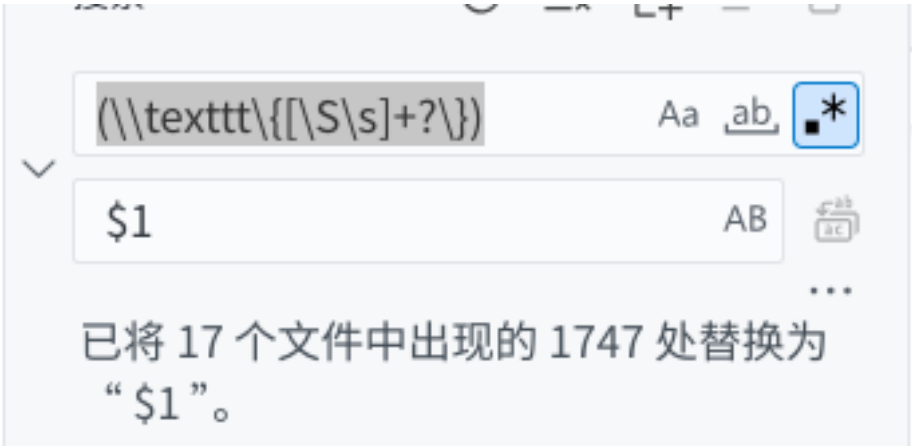


图 7.2: VS Code 中的正则替换实例

7.3 在纯 Windows 上搭建 C/C++ 编程环境

说明

下文经常提到的一个东西“环境变量”是操作系统的重要变量 PATH。该变量存储了一系列路径，当用户在命令行中输入一个命令时，操作系统会在这些路径中查找对应的可执行文件。如果找到了，就会运行该文件，否则会提示“命令未找到”之类的错误。因此，在安装编译器、解释器等工具时，通常需要将它们的安装路径添加到 PATH 变量中，以便在任何位置都能使用这些工具。否则，每次输入命令时都要指定完整路径，极其不便。

7.3.1 C 系编译器及其环境配置

C/C++ 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。

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](https://www.msys2.org/)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）：

```
1 pacman -S mingw-w64-ucrt-x86_64-gcc
2 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。
- 在安装完 GCC 和 GDB 后，我们可以在终端中运行以下命令来验证是否安装成功：

```
1 gcc --version
2 gdb --version
```

输出大概类似于：

```
1 gcc (GCC) 15.2.1 20250813
2 Copyright © 2025 Free Software Foundation, Inc.
3 本程序是自由软件；请参看源代码的版权声明。本软件没有任何担保；
4 包括没有适销性和某一专用目的下的适用性担保。
```

或其英文版本。GDB 的输出类似。如果你能看到上述输出，则说明 GCC 和 GDB 安装成功且配置正确。否则，请检查你的安装步骤和环境变量设置是否正确。

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！）

另外，直接下载预编译版本往往会遇到版本过老的问题，比如笔者就见过一个人在 2025 年还在用 GDB 7.6.1（2013 年发布的版本）调试代码，结果最新版本的 VS Code 无法对该 GDB 进行注入，从而导致无法输入、无法调试的问题。而使用 MSYS2 等手段下载的 GDB 往往是最新或相当新的版本，能够避免这些问题。

注意

有些同学的 Windows 系统可能不是使用 UTF-8 编码的（例如使用 GBK 编码），这会导致 GCC 无法正确处理包含非 ASCII 字符的文件名，从而引发各种奇怪的问题。因此，建议将系统的区域设置更改为使用 UTF-8 编码。具体步骤见第一章[21.1.4](#)节。如不愿修改该设置项，则务必注意不要使用非 ASCII 文件名！

提示

Linux 用户安装 GCC 是最简单的。例如在 Arch Linux 上，只需要运行以下命令：

```
1 sudo pacman -S gcc gdb
```

即可安装 GCC 和 GDB。其他的发行版则使用对应的包管理器安装即可，例如 Debian/Ubuntu 使用 apt，Fedora 使用 dnf 等。

macOS 也是类 UNIX 系统，因此使用 HomeBrew 安装 GCC 也是非常简单的：

```
1 brew install gcc gdb
```

7.3.2 在 VS Code 中配置 C/C++

提示

VS Code 的工作基于工作区（workspace）的概念，实际上我们启用的不少插件也都是基于工作区启用的。对于 C/C++ 扩展来说，它们的工作有不少基于当前工作区目录，并帮助你输入编译和调试命令。如果你直接打开一个 C/C++ 文件，而不是一个工作区目录，那么这些基于工作区的插件也将会全部失效。此时，VS Code 就回归其作为一个文本编辑器的本质，无法为你提供编译和调试的功能。

最简单的工作区就是一个目录。所以我们必须打开一个文件夹而不是仅打开一个文件，这样才能使用 VS Code 的 C/C++ 扩展来编译和调试代码，其他扩展也大同小异。Code 的其他语言扩展也大同小异。在之后，我都会用“打开一个工作区”这种说法。

你需要在 VS Code 中配置 GCC，以便能够编译和运行 C/C++ 代码。

json 配置 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：C/C++、C/C++ Extension Pack 和 C/C++ Themes。这些插件都是微软提供的。
2. 配置 tasks.json 文件：在 VS Code 中打开一个新的工作区，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¹。我会使用 # 来表示注释内容，但 JSON 文件是不支持注释的，因此请不要把这些注释内容放进你的 JSON 文件中。不懂 JSON 文件的可以把整个手册翻到《数据交换格式》一节。

```
1 {
2   "version": "2.0.0",
3   "tasks": [
4     {
5       "label": "C/C++: g++.exe 生成活动文件", # 任务标签，你可以通过这个标签来引用这个任
        ↳ 务，可以自定义。
6       "type": "shell", # 任务类型，这里是在 shell 中运行命令，不建议改动
7       "command": "g++", # 编译器命令。如仅编译 C 代码，则改为 gcc。你也可以把这个改为编译
        ↳ 器的完整路径。
8       "args": [
9         "-g", # 开启调试信息
10        "${file}", # 当前打开的文件
```

¹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

```

11     "-O3",          # 开启 O3 优化
12     "--std=c++23",  # 指定 C++ 标准为 C++23。这里是因为笔者习惯现代 C++。不指定则使
    ↪ 用编译器默认的标准，现代 GCC 默认 C++17。
13     "-o",          # 指定输出文件
14     "${fileDirname}\\${fileBasenameNoExtension}.exe" # 输出文件路径到当前文件同
    ↪ 目录，文件名和当前文件名相同但扩展名为.exe
15 ],
16 "group": {
17     "kind": "build", # 任务组类型，这里是构建任务
18     "isDefault": true # 该任务为默认构建任务
19 },
20 "problemMatcher": ["$gcc"], # 问题匹配器，用于解析编译器输出的错误和警告信息。gcc
    ↪ 和 g++ 通用，都应该使用"$gcc"。
21 "detail": "生成活动文件" # 细节描述，这个可以自定义，也可以删除该字段。
22 }
23 ]
24 }

```

以下是一个简单的 launch.json 文件示例：

```

1 {
2     "version": "0.2.0",
3     "configurations": [
4         {
5             "name": "C++ Launch", # 配置名称，可以自定义
6             "type": "cppdbg",      # 调试器类型，这里使用 C++ 调试器
7             "request": "launch",   # 请求类型，这里是启动调试
8             "program": "${fileDirname}\\${fileBasenameNoExtension}.exe", # 要调试的
    ↪ 程序路径，直接照着上述 tasks.json 的输出路径抄下来
9             "args": [],           # 程序参数，这里留空，同学们可以根据需要添加，但一般不需要
10            "stopAtEntry": false, # 是否在程序入口处停止，这里设置为 false
11            "cwd": "${workspaceFolder}", # 当前工作目录，这里设置为工作区根目录
12            "environment": [],      # 环境变量，这里留空
13            "externalConsole": false, # 是否使用外部控制台，这里设置为 false，使用内置终
    ↪ 端。如果 true，则会弹出一个新的控制台窗口
14            "MIMode": "gdb",        # 调试模式，这里使用 gdb
15            "setupCommands": [ # 调试器设置命令，这里仅启用 pretty-printing 功能，也可以
    ↪ 添加其他命令，例如设置断点等
16                {
17                    "description": "Enable pretty-printing for gdb", # 描述信息
18                    "text": "-enable-pretty-printing", # 命令文本，实际上和 gdb 命令行
    ↪ 中输入的命令是一样的
19                    "ignoreFailures": true # 忽略失败，这里设置为 true
20                }
21            ],
22            "preLaunchTask": "C/C++: g++.exe 生成活动文件", # 预启动任务，这里设置为上述
    ↪ tasks.json 中的任务标签，因为必须先编译才能调试
23            "miDebuggerPath": "C:\\msys64\\ucrt64\\bin\\gdb.exe" # gdb 调试器路径，请
    ↪ 根据你的实际安装路径进行调整
24        }
25    ]
26 }

```

UI 配置² 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。

²本节作者张天齐。

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。另，如果想使用较新的 C++ 特性，也可以加上 `--std=c++23` 之类的选项。
- 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。如果不是为了考试服务，可以选择最新版本，如 C++26；如果不希望使用过新的特性（如初学者），也可以选择 C++11。
- 高级：一个都不用动。

调试配置没有 UI 配置选项，我们还得老老实实地手动编辑 `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。

如果你确实做了这些事情，但是你的 VS Code 仍然无法编译和调试 C/C++ 代码，那么你可能需要检查以下几点：

- 使用 `where g++` 命令来确定你的编译器是不是在 `PATH` 环境变量中，另外你还需要确定上述识别出的编译器是不是你想用的那个。
- 检查你的 C++ 目录和编译器目录是否包含非 ASCII 字符（如汉字、空格等）。
- 检查你是不是错误地使用了 `gcc` 来编译 C++ 代码。C++ 代码需要使用 `g++` 来编译，而不是 `gcc`，这是最常见的错误之一。
- 检查你的包含路径是否正确。VS Code 会自动检测你的编译器和标准库，但如果你使用了自定义的路径或者安装了多个版本的编译器，可能需要手动配置包含路径。
- 检查你改的几个配置文件是不是正确的配置文件、有没有保存。
- 检查你的 VS Code 和 C/C++ 插件是否是最新版本。有时候，旧版本的插件可能会有 bug 或者不兼容最新的 VS Code 版本。

一般情况下，修正的方法很简单：如果是环境变量、路径等问题，重新设置就可以；如果是误用 GCC 调试 C++ 代码的问题，只需要改回 G++ 即可；如果是配置文件问题，杀掉当前所有终端，删除 `.vscode` 文件夹，然后重新编译调试就可以了。

另外，如果希望使用 Code 的插件来帮助你编译并运行 C/C++ 代码，则需要打开一个工作文件夹。否则，插件无法为你提供编译和运行的功能。当然，`g++ main.cpp -o main.exe && ./main.exe` 也并非不可。

7.3.3 不依赖 VS Code 的编译方式

如果你不想使用 VS Code 来编译和运行 C/C++ 代码，你也可以直接在命令行中使用 GCC 来编译和运行代码。以下是一个简单的示例：

```
1 g++ main.cpp -O2 -g -o main.exe
2 ./main.exe
```

上述命令的含义是：使用 g++ 编译器编译 main.cpp 文件，开启 -O2 优化，开启调试信息 -g，并将生成的可执行文件命名为 main.exe。然后，使用 ./main.exe 命令来运行生成的可执行文件。

命令行能提供更高的灵活性，可以随意增删改编译选项，例如添加 -Wall 来开启警告等。我个人非常推荐同学们学会使用命令行来编译和运行代码，这样可以更好地理解编译过程，并且能够更好地控制编译选项。

7.4 用 WSL 配置 C/C++ 环境

使用 WSL，可以大大简化配置环境的过程。

首先，假设你没有安装过任何 WSL 发行版。打开 PowerShell，输入以下命令：

```
1 wsl --install # 默认安装 Ubuntu 发行版，完全足够
```

等待安装完成后，重启电脑。重启后，在开始界面找到 Ubuntu 图标，点击打开。第一次打开时，会提示你创建一个 Linux 用户名和密码，按照提示操作即可。

接下来，换源并更新系统：

```
1 sudo apt update && sudo apt upgrade -y
```

换源操作请参考[清华大学开源软件镜像站的帮助页面](https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/)。

然后，安装 C/C++ 开发环境：

```
1 sudo apt install gcc gdb # CMake 先不装，新手用不上
```

下一步，打开你的 VS Code，在欢迎页上选择“连接到”，在弹出的菜单里选择 WSL，此时如果没有安装过 WSL 插件则会自动安装。连接成功后，VS Code 会自动打开一个新的窗口，左下角会显示你当前连接的 WSL 发行版名称。

然后你就需要在该窗口里安装一个 C++ 插件，以便于代码高亮、补全等。按下 Ctrl+Shift+X 打开扩展商店，搜索“C++”，找到由微软官方发布的那个，点击安装即可。

现在，你就可以愉快地在 WSL 里编写 C/C++ 代码了！（当然你依然需要创建一个工作目录，并在里面创建源代码文件，否则扩展不起作用。）

7.5 Python、虚拟环境及其配置

7.5.1 简单安装 Python

如果仅仅是安装 Python，那可比安装 C 系编译器简单得多了，直接去[Python 官网](#)上下载符合你操作系统的最新版本就行了，只要记得安装时勾选“添加到 Path”就行了。

检验安装的方式是：打开终端，输入以下命令：

```
1 python --version
```

如果输出了 Python 的版本信息，例如 Python 3.12.0，则说明 Python 安装成功且配置正确。否则，请检查你的安装步骤和环境变量设置是否正确。

7.5.2 虚拟环境及其配置

然而，这样安装会有一个问题：会将 Python 安装到全局环境中。

Python 作为一门流行的编程语言，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，第三方库的安装和管理是必然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沙盒，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，更不会影响其他沙盒。目前较常见的虚拟环境创建工具有 conda、mamba、venv、uv、pixi 等。在这里我们讲述传统手段，也就是 conda 和 mamba。

conda conda 是数据科学领域开山鼻祖级别的包管理和虚拟环境管理工具。它不仅可以管理 Python 包，还可以管理其他语言的包，例如 R、Ruby 等，因此也是迄今为止用的最广泛的跨语言包管理工具。但 conda 有两个缺点：一个是笨重，一个是慢。

conda 有两个主要的发行版：Anaconda 和 Miniconda。Anaconda 是一个完整的数据科学平台，包含了大量的预装包，适合初学者和数据科学家使用；Miniconda 则是一个轻量级的版本，解决了笨重的问题（虽然依然慢）。我们推荐使用 Miniconda。

Windows 用户可以把以下命令直接贴到终端里运行：

```
1 curl https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe -o  
  ↪ .\miniconda.exe  
2 start /wait "" .\miniconda.exe /S  
3 del .\miniconda.exe
```

Linux 用户见[官方网站](#)即可，其实大差不差。

安装完成后，你可以通过以下命令来创建并激活一个新的虚拟环境：

Python 官网: <https://www.python.org/downloads/>

官方网站: <https://www.anaconda.com/docs/getting-started/miniconda/install>

```
1 conda create -n myenv python=3.12
2 conda activate myenv
```

上述命令的含义是：创建一个名为 `myenv` 的虚拟环境，并安装 Python 3.12 版本。你可以根据需要更改环境名称和 Python 版本。在快速开发的背景下，可以把东西一股脑的安装到 `base` 环境中，但不推荐把这种手段用到正式开发中。

mamba mamba 是 conda 的一个替代品，旨在解决 conda 的慢的问题。mamba 使用 C++ 编写，具有更快的包解析和安装速度。mamba 完全兼容 conda 的命令和配置文件，因此用户可以无缝切换到 mamba，而无需修改现有的环境和脚本。

如果计算机上已经安装了 conda，则可以通过以下命令来安装 mamba：

```
1 conda install mamba -n base -c conda-forge
```

安装完成后，你可以使用 mamba 命令来代替 conda 命令，例如：

```
1 mamba create -n myenv python=3.12
2 mamba activate myenv
```

上述命令的含义与使用 conda 时相同。

对于没有安装过 conda 的用户，我们遵从“最简单”原则，安装 micromamba。micromamba 是 mamba 的一个轻量级“纯净”版本，不依赖于 conda 或 Anaconda。它仅包含 mamba 的核心功能，适合那些只需要基本包管理和虚拟环境管理功能的用户。

只需要运行下列命令：

```
1 "${SHELL}" <(curl -L micro.mamba.pm/install.sh)
```

然后按照提示操作即可。安装完成后，你可以使用 micromamba 命令来创建和管理虚拟环境，使用方法与 conda 类似。

7.5.3 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四个 Python 插件：Python、Pylance、Python Debugger 和 Python Environments。

然后，在 VS Code 中打开一个工作区，并创建一个 Python 文件 `*.py`。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。有时候，右下角会直接帮你选好，那也可以点击它来确认或更改。

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 (?)。

7.5.4 不依赖 VS Code 的运行方式

如果你不想使用 VS Code 来运行 Python 代码，你也可以直接在命令行中使用 Python 解释器来运行代码。以下是一个简单的示例：

```
1 python main.py
```

上述命令的含义是：使用当前虚拟环境中的 Python 解释器来运行 main.py 文件。

我们也可以指定特定的 Python 解释器来运行代码，例如：

```
1 /path/to/python main.py
```

这样就可以使用指定路径下的 Python 解释器来运行代码。一般有 CPython、PyPy 等不同的 Python 解释器可供选择。

另外，如果你想要运行一个交互式的 Python 环境，可以使用以下命令：

```
1 python
```

这将会启动一个交互式的 Python 解释器，你可以在其中输入 Python 代码并立即执行。想要退出该环境，可以使用 `exit()` 命令或者按下 `Ctrl + D`（Linux/macOS）或 `Ctrl + Z`（Windows），以输入一个 EOF 信号。如果你在 Linux 上不小心按下了 `Ctrl + Z`，则会将 Python 进程挂起，此时可以使用 `fg` 命令将其恢复。

7.5.5 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 PowerShell，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`。

7.6 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。

7.6.1 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。

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：

```
1 #include <iostream>
2 int main() {
3     std::cout << "Hello, World!" << std::endl;
4     return 0;
5 }
```

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：

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

请使用类似的方式手敲、编译、运行以下代码：

```
1 #include <iostream>
2 int main() {
3     int a, b;
4     std::cout << "Enter the first integer: ";
5     std::cin >> a;
6     std::cout << "Enter the second integer: ";
7     std::cin >> b;
8     std::cout << "Their sum is: " << a + b << std::endl;
9     return 0;
10 }
```

Python

对于 Python，我们同样可以使用以下代码来编写第一个程序：

```
1 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：

```
1 python hello.py
```

请使用类似的方式手敲、编译、运行以下代码：

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

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。

说明

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。

7.6.2 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：

```
1  #include<iostream>
2  using namespace std;
3  int mian()
4  {
5      cout<<"Hello World!"<<endl;
6      return 0;
7  }
```

而它的错误信息是在编译时报出：

```
1 g++ example.cpp -o example.exe
2 ld.exe: *.a(lib64_libmingw32_a-crtexewin.o): in function `main':
3 C:/.../crtexewin.c:70: undefined reference to `WinMain'
4 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：

```
1 def calc(numbers):
2     total = sum(numbers)
3     count = len(numbers)
4     return total / count
5
6 numbers = [10, 20, 30, 40, 50]
7 print("Average:", calc(numbers))
8
9 numbers.append("60")
10 print("Updated Average:", calc(numbers))
```

其报错是：

```
1 Average: 30.0
2 Traceback (most recent call last):
3   File "example.py", line 10, in <module>
4     print("Updated Average:", calc(numbers))
5   File "example.py", line 2, in calc
6     total = sum(numbers)
7 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` 列表中增加了一个字符串 “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。

7.6.3 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[7.3.2](#)。)

我们调试主要有以下几个手段: 打日志、打断点、写测试代码。

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:

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。

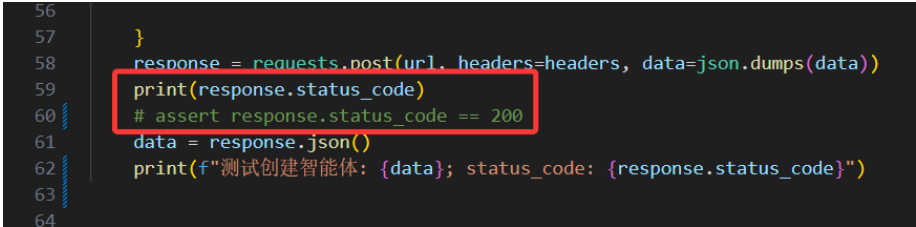


图 7.3: 打日志的例子

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。

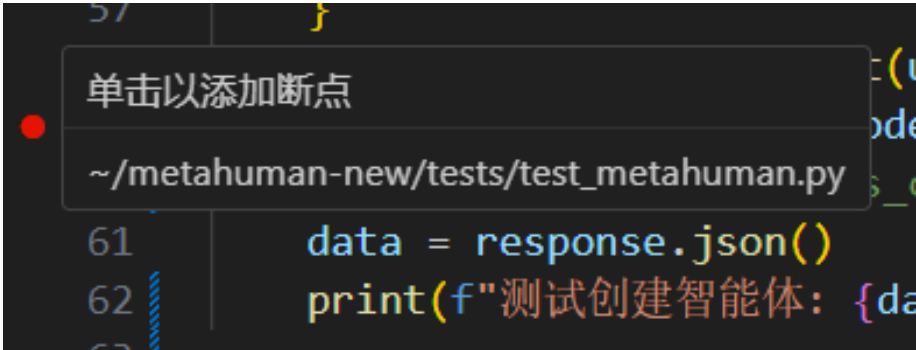


图 7.4: 打断点的例子

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。

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。

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现错

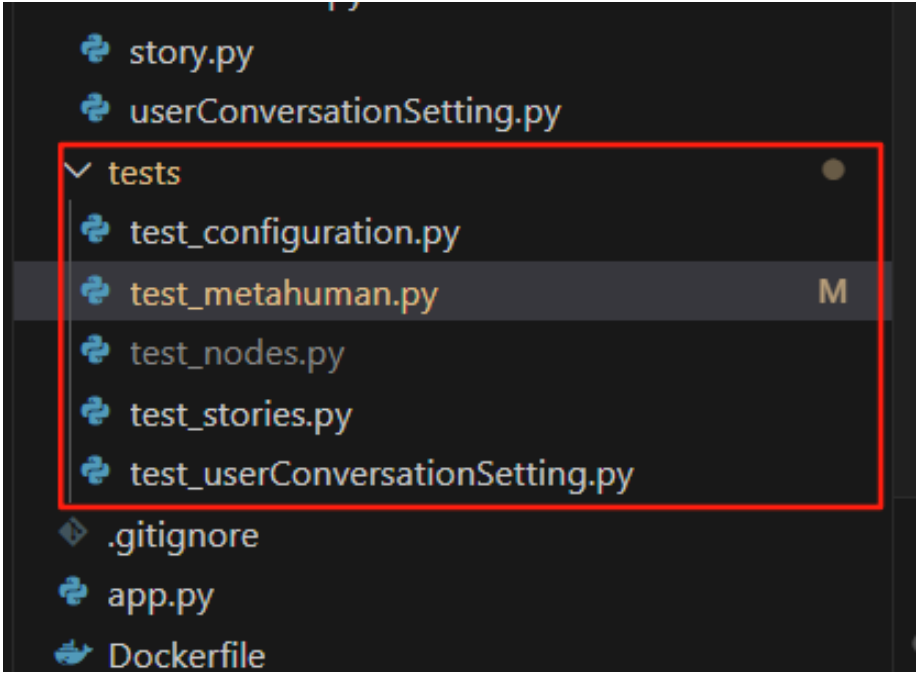


图 7.5: 我为本人实习的项目编写的集成测试

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。

第八章 C 语言入门

本章默认大家此前没有任何编程基础。

所谓“编程”，就是让计算机按照我们想要的方式工作。计算机本身并不会思考，它只能听从我们的指令去做事。因此，我们需要用一种计算机能够理解的语言来告诉它我们想要它做什么，这种语言就叫做“编程语言”。

C 语言是最早的编程语言之一，在上世纪 70 年代被发明出来。它是一种结构化的、过程式的编程语言，具有高效、灵活和可移植等特点。C 语言广泛应用于系统软件开发、嵌入式系统、游戏开发等领域，著名软件如 Linux 内核、Git、GCC、Vim 等简单但强大的软件都是用 C 语言编写的；Python 的官方实现也是 C 语言（Cpython），很多高性能的 Python 库（如 NumPy、Pandas、sklearn 的大部分等）也是用 C 语言写的。

安装 C 编译器的方法见第 7.3 节，这里不再赘述。我们要写 C，首先应该创建一个以 .c 结尾的文本文件，例如 hello.c。然后，在这个文件中写入 C 代码，最后使用 C 编译器将其编译成可执行文件。而对于初学者而言，编译这种脏活累活全部丢给 VS Code 的 C/C++ 扩展来做就好了。

8.1 C 语言的基本语法

8.1.1 你的第一个 C 程序

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, World!\n");
5     return 0;
6 }
```

把这些内容敲到你的 C 文件中，保存，编译并运行（如果你按照我所推荐的方式安装并配置好了，那么按下 F5 就可以编译并运行了），你就会看到终端上输出了 Hello, World!。

上述程序就算是一个最简单的 C 程序了。

第一次写 C 的时候，记住以下事项：

1. 程序有入口；
2. 先声明，再计算；
3. 算完告诉外面。

剩下的内容和说话一样，只不过是使用 C 的语法来表达。我们说话的句号在 C 中是分号。逐行拆解上述示例代码：

- `#include <stdio.h>`：告诉编译器，我要用输入输出工具。
- `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。
- `printf("Hello, World!\n");`：把东西一股脑全送到屏幕上。`\n` 表示换行。
- `return 0;`：返回 0，告诉操作系统：一切 OK。除非你知道你在做什么，否则这里不要改成其他数字。本行就是“算完告诉外面”。

在 C 中，有两种代码：一种是以 `#` 开头的预处理指令，另一种是常规语句。预处理指令指的是在编译之前进行的一些操作，例如包含头文件、定义宏等，详见 8.1.16。常规语句则指的是程序的主要逻辑。常规语句应以分号结尾，且在结尾之后应换行（除非写注释）。压行是不好的行为，会影响代码的可读性，尽量自然地换行。

8.1.2 变量及其运算

编程的本质是对数据进行操作，而经过操作的数据可能会变化。对于会变化的数据，我们称之为“变量”。而这些量也有不同的类型，例如“人数”肯定是整数，而“身高”则可能是小数。

在 C 中，变量要**先声明，再使用**。声明的方法是：先写类型，再写名字。这个“名字”是我们之后用来使用这个变量的标识符，类似别人提到“张三”就能对应到这个人的头上。这个使用在编程上被叫做**调用**。

取名有一定的规则：不能用已经用过的名字，这个名字包括 C 保留的关键字和你自己已经定义过的名字；名字只能包含字母、数字和下划线，不能包括其他符号，且不能以数字开头；名字区分大小写，例如 `age` 和 `Age` 是两个不同的名字。

```

1  int age = 18; // 声明一个整数变量 age，并初始化为 18
2  double pi = 3.14; // 声明一个双精度浮点数变量 pi，并初始化为 3.14
3  char grade = 'A'; // 声明一个字符变量 grade，并初始化为 'A'
4
5  age = 19; // 把 age 的值改为 19

```

这些语言本质上都可以用自然语言解释为：我有个 `xx` 叫 `xx`，它的值是 `xx`。例如第一行代码：我有个整数叫 `age`，它的值是 18。如果之后我想用 `age` 这个变量，就可以直接写 `age`，不需要再次声明“我有个整数叫 `age`”了。

上述 `int` 等四个排在第一个的关键字是变量的类型，分别表示整数、双精度浮点数、字符和布尔类型。在 C 中，变量类型不能在运行时改变，一旦声明则类型固定，因此 C 也被归类为“静态类型”语言。

上述声明中的等号和数学中的等号**不相同**。在这里，等号的意思是“赋值”，指的是让等号左边的值变成等号右边的值。也就是说，等号右边的值会被计算出来，然后存储到等号左边的变量中。

而变量的运算则和数学差不多，比方说

```
1  int a = 10;
2  int b = 20;
3  int c = a + b;
4  c = a * 2;
5  c += 5;
```

第三行中, `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; 就像我们告诉李四“有个人叫张三”, 之后再提到张三的时候就不需要再说“有个人叫张三”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局部变量的值, 那么这个变量的初始值将会是一个**未定义行为**, 这个值取决于内存中该位置之前存储的内容。我们不能依赖于这个, 因此最好在声明变量的时候就给它赋初始值, 例如 `int a = 0;`。对于全局变量, 如果不初始化, 编译器会自动将其 0 初始化。

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。

不要过分纠结 `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。

注意

尽量单独使用 `++` 和 `--`, 不要把它们和其他运算混在一起使用, 更不要在同一表达式中对同一个变量使用多次 `++` 或 `--`。例如, `a = b++` 虽然不推荐但还勉强可以, 但是 `a = b++ + b++` 和 `i = i++` 都是未定义行为。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。

笔者个人从工程的眼光上看来, 非常不建议弄出 `a = b++` 这类的代码, 尽管这类代码在竞赛中会让很多 OIer 感到 Tricky, 但是在工程中会让人无比恼火。如果想先用 `b` 的值再加 1, 可以写成 `a = b; b++;`; 如果想先加 1 再用 `b` 的值, 可以写成 `b++; a = b;`。上述写法一般只有非常约定俗成的场合才会使用, 例如 `while(T--)` 或者 `stk[++top]=x` ——不过即使是我, 也更习惯于写成 `for(;;T>0;T--)` 和 `stack<int> stk; stk.push(x);`。

8.1.3 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阅读者看的。

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。

8.1.4 输入、输出及其格式化

输入和输出是程序与外界进行交互的方式。在 C 中，常用的输入输出函数有 `printf` 和 `scanf`。这两个函数都定义在 `stdio.h` 头文件中，因此在使用它们之前需要包含该头文件。

`printf` 用于输出数据到屏幕上，其基本语法如下：

```
1 printf(" 格式字符串", 参数 1, 参数 2, ...);
```

而 `scanf` 用于从键盘读取输入，其基本语法如下：

```
1 scanf(" 格式字符串", &变量 1, &变量 2, ...);
```

上述输入中的 `&` 符号不能省略，也就是需要写成 `&a` 的形式。

那这个“格式字符串”是什么东西呢？它是一个字符串，其中包含了文本和格式说明符。格式说明符用于指定要输出或输入的数据类型和格式。常见的格式说明符有：

- `%d`：表示整数类型。
- `%f`：表示浮点数类型。
- `%c`：表示字符类型。
- `%s`：表示字符串类型。

例如，下面的代码演示了如何使用 `printf` 和 `scanf` 进行输入输出：

```
1 #include <stdio.h>
2
3 int main() {
4     int age;
5     printf(" 请输入你的年龄: ");
6     scanf("%d", &age); // 这里的输入是 1 个整数
7     printf(" 你输入的年龄是: %d\n", age); // 这里你看到的输出是：“你输入的年龄是: xx”，
      ↳ xx 是 age 的值
8     return 0;
9 }
```

在字符串中，除了上述格式说明符，还可以有控制字符，也就是类似于上文 `\n` 这样的东西。上述代码中的 `\` 是“转义符号”，表示后面的字符有特殊含义，而不是其本身的含义。常见的控制字符有：

- `\n`：换行符。
- `\t`：制表符 (Tab)。
- `\r`：回车符。
- `\"`：双引号字符。
- `\'`：单引号字符。
- `\\`：反斜杠字符。
- `%`：百分号字符。
- `\0`：字符串结束符。

练习：加减运算

写一个程序，接受两个整数输入，然后输出它们的和、差。

程序输入：两个整数 a 和 b ，用空格分割。

程序输出：两行，第一行输出 $a + b$ ，第二行输出 $a - b$ 。

答案

由于这是第一个题，我们就给出一个参考答案吧。这个题目的答案是显然的，但需要让同学们知道这种题目应该以一种什么形式去写。

在 OJ 等平台上，我们一般需要提交一个完整的程序。

首先，我们要处理问题的核心逻辑：加法和减法。

```
1      int a = 0;
2      int b = 0;
3
4      int sum = a + b; // 计算和
5      int diff = a - b; // 计算差
```

下一步，处理输入输出：

```
1      scanf("%d %d", &a, &b); // 读取输入
2      printf("%d\n", sum); // 输出和
3      printf("%d\n", diff); // 输出差
```

我们应该把这些代码按照一个正确的顺序组织起来，并且放在一个完整的 C 程序框架内：

```
1      #include <stdio.h>
2
3      int main() {
4          int a = 0;
5          int b = 0;
6
7          scanf("%d %d", &a, &b); // 读取输入
8
9          int sum = a + b; // 计算和
```

```
10     int diff = a - b; // 计算差
11
12     printf("%d\n", sum); // 输出和
13     printf("%d\n", diff); // 输出差
14
15     return 0;
16 }
```

这样，我们就完成了这个练习题的解答。

OJ 等自动评测平台会根据题目的输出格式来验证程序的正确性，因此我们必须严格按照题目要求来编写程序，不要输出其他内容，例如“请输入两个整数:”之类的提示语句，这样会导致判错。而在实际生活中，我们可以添加这些提示语句来提高程序的用户体验。

8.1.5 常变量

常变量（也叫不可变变量、只读变量）是指在程序运行过程中其值不能被修改的变量。在 C 中，可以使用 `const` 关键字来声明常变量。例如：

```
1  const int MAX_VALUE = 100;
2  // MAX_VALUE = 200; // 这行代码编译不通过，因此要注释掉
```

也就是在常规的声明前面加上 `const` 关键字。上述代码的意思是：我要创建一个常变量 `MAX_VALUE`，它的值是 100。

我们发现，任何对常变量的修改操作都会使得编译不通过。因此，常变量的值一旦确定就不会在程序运行时改变。常变量的名字通常使用全部大写字母来表示，以便于和变量区分。

8.1.6 条件判断

有时候我们想要设计一个网站，给不同的人显示不同的内容。这个时候，我们就需要用到条件判断。条件判断可以让程序根据不同的条件执行不同的代码块。在 C 中，常用的条件判断语句有 `if` 语句和 `switch` 语句，以及三元运算符。

条件表达式

一个条件表达式，最简单的情况肯定是“真”或“假”。C 语言规定：`true` 等价于 1，`false` 等价于 0；但非 0 的数值还是什么别的非空的东西全部视作 `true`。因此，`if (1)` 和 `if (-42)` 甚至 `if (3.14)` 和 `if ("hello")` 都肯定会执行，而 `if (0)` 和 `if ("" )` 则肯定不会执行。

但是实际上情况肯定没这么简单，所以需要用比较运算符和逻辑运算符来构造更复杂的条件表达式。常见的比较运算符有：

- `==`：等于。
- `!=`：不等于。
- `>`：大于。
- `<`：小于。

- `>=` : 大于等于。
- `<=` : 小于等于。
- `&&` : 逻辑与 (AND): 前后两个条件都为真时, 结果才为真。
- `||` : 逻辑或 (OR): 前后两个条件有一个为真时, 结果就为真。
- `!` : 逻辑非 (NOT): 反转后面条件的真假。
- `()` : 括号, 用于改变运算优先级。

也就是说: `3+2==5` 是 `true`, `3+2!=5` 是 `false`, `3 > 2` 和 `3 >= 2` 也都是 `true`。而 `(3 > 2) && (2 > 1)` 是 `true`, `(3 > 2) || (2 < 1)` 也是 `true`, 而 `!(3 > 2)` 则是 `false`。于是, 借助这些比较运算符和逻辑运算符, 我们就可以构造出复杂的条件表达式了。

提示

在 C++ 中, 不能使用类似 `1 <= x <= 2` 这样的连续记号来表示区间。正确的写法是 `(1 <= x) && (x <= 2)`, 即把每个比较都单独写出来, 然后用逻辑与运算符连接起来。

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!

if 语句

if 语句的基本语法如下:

```
1 if (cond1){
2     // codes...
3 }
4 else if (cond2){
5     // codes...
6 }
7 else {
8     // codes...
9 }
```

上述 `cond1` 和 `cond2` 是条件表达式, 它们的结果是布尔值 (真或假)。如果 `cond1` 为真, 则执行第一个代码块; 否则, 如果 `cond2` 为真, 则执行第二个代码块; 否则, 执行最后一个代码块。在实际操作中, 可以没有任何 `else if` 或 `else` 分支。

例子:

```
1 if (age < 18) {
2     cout << " 未成年";
3 }
4 else if (age < 60) {
5     cout << " 成年人";
6 }
7 else {
8     cout << " 老年人";
9 }
```

9 }

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。

switch 语句

switch 语句的基本语法如下：

```
1 switch (expression) {
2     case value1:
3         // codes...
4         break;
5     case value2:
6         // codes...
7         break;
8     ...
9     default:
10        // codes...
11 }
```

上述 `expression` 是一个表达式，其结果将与各个 `case` 后面的值进行比较。如果结果与某个 `case` 后面的值相等，则执行对应的代码块，直到遇到 `break` 语句为止。如果没有任何 `case` 匹配，则执行 `default` 代码块（如果有的话）。注意，`break` 语句用于跳出 `switch` 语句，否则程序会继续执行后续的代码块。在实际操作中，也可以没有 `default` 分支。

例子：

```
1 switch (day) {
2     case 1:
3         cout << " 星期一";
4         break;
5     case 2:
6         cout << " 星期二";
7         break;
8     case 3:
9         cout << " 星期三";
10        break;
11    // ..... 其他的，基本一个写法
12 }
```

这也一目了然不言而喻了。

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：

```
1 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：

```
1 int n = 5;
2 const char* result = (n % 2 == 0) ? "even" : "odd";
```

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：

```
1 int n = 5;
2 string result;
3 if (n % 2 == 0) {
4     result = " 偶数";
5 } else {
6     result = " 奇数";
7 }
```

练习：闰年判断

写一个程序，接受一个年份输入，然后判断这一年有多少天（365 或 366）。提示：闰年的判断规则是：四年一闰，百年不闰，四百年再闰。

程序输入：一个整数 year，表示年份。

程序输出：一个整数，表示该年份的天数（365 或 366）。

练习：天数判断

写一个程序，接受一个月份输入，然后输出该月份有多少天。假设输入的月份是 1 到 12 之间的整数，且不考虑闰年。

程序输入：一个整数 month，表示月份。

程序输出：一个整数，表示该月份的天数。

8.1.7 循环

循环是一种重复执行某段代码的结构，直到满足某个条件为止。有的同学可能会问：为什么不直接把代码写多几遍就好了？这是因为有时候我们并不知道需要重复多少次，或者需要根据某个条件来决定是否继续循环，因此这时候就需要用到循环结构。

在 C 中，常用的循环语句有 for 循环、while 循环和 do-while 循环。

for 循环

for 循环的基本语法如下：

```
1 for (初始化; 条件; 更新) {  
2     // 循环体代码  
3 }
```

上述 初始化用于设置循环变量的初始值, 条件是一个布尔表达式, 用于判断是否继续循环, 更新用于更新循环变量的值。循环体代码会在每次循环中执行。例如, 下面的代码演示了如何使用 `for` 循环打印 1 到 10 的数字:

```
1 for (int i = 1; i <= 10; i++) {  
2     printf("%d\n", i);  
3 }
```

while 循环

`while` 循环的基本语法如下:

```
1 while (条件) {  
2     // 循环体代码  
3 }
```

上述 条件是一个布尔表达式, 用于判断是否继续循环。循环体代码会在每次循环中执行, 直到条件为假为止。例如, 下面的代码演示了如何使用 `while` 循环打印 1 到 10 的数字:

```
1 int i = 1;  
2 while (i <= 10) {  
3     printf("%d\n", i);  
4     i++;  
5 }
```

实际上, `while` 循环可以和 `for` 循环互相转换。上面的 `for` 循环可以改写成 `while` 循环, 反之亦然。

do-while 循环

`do-while` 循环的基本语法如下:

```
1 do {  
2     // 循环体代码  
3 } while (条件);
```

上述 条件是一个布尔表达式, 用于判断是否继续循环。循环体代码会先执行一次, 然后再判断条件是否为真, 如果为真则继续循环, 直到条件为假为止。例如, 下面的代码演示了如何使用 `do-while` 循环打印 1 到 10 的数字:

```
1 int i = 1;
2 do {
3     printf("%d\n", i);
4     i++;
5 } while (i <= 10);
```

可以看出，do-while 循环至少会执行一次循环体代码，而 while 循环则可能一次都不执行。

循环控制语句

有些时候，我们希望在循环中跳过某些迭代，或者提前结束循环。为此，C 提供了两种循环控制语句：break 和 continue。

break 可以立刻跳出整个循环，不再执行后续的迭代。例如：

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break; // 当 i 等于 5 时，跳出循环
4     }
5     printf("%d\n", i);
6 }
```

这个代码的输出是 1 到 4，后面的数字都不会被打印出来，因为循环已经被提前结束了。

而 continue 则是跳过当前迭代的剩余所有代码，直接进入下一次迭代。例如：

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         continue; // 当 i 是偶数时，跳过当前迭代
4     }
5     printf("%d\n", i);
6 }
```

这个代码的输出是 1 到 10，除了 5 这个数字，因为当 i 等于 5 时，当前迭代被跳过了。

练习：日期差

写一个程序，接受两个日期输入，计算两者之间差了多少天。不考虑闰年问题。

程序输入：四个整数 month1、day1、month2、day2，分别表示第一个日期的月份和天数，以及第二个日期的月份和天数。

程序输出：一个整数，表示两个日期之间的天数差。

提示：把上一节的“天数判断”题目作为子任务来完成，也就是说可以试着复用这些代码。

8.1.8 数组

数组，顾名思义，也就是“一组数据”。这组数据的类型是相同的，可以是整数、浮点数、字符等。数组中的每个数据都有一个索引（下标），用于标识它在数组中的位置。数组的索引从 0 开始。

例如，下面的代码声明了一个包含 5 个整数的数组：

```
1  int numbers[5] = {10, 20, 30, 40, 50};
2  // 访问数组元素
3  int firstNumber = numbers[0]; // 访问第一个元素, 值为 10
4  int thirdNumber = numbers[2]; // 访问第三个元素, 值为 30
5  int bad = numbers[5]; // 错误, 数组越界访问, 可能导致段错误或返回不可知的值
6  int bad2 = numbers[-1]; // 错误, 数组越界访问
7
8  numbers[1] = 25; // 修改第二个元素的值为 25
```

有的同学可能会问：我为什么不能用

```
1  firstNumber = 25;
```

来修改 `numbers[0]` 的值呢？这是因为 `firstNumber` 和 `numbers[0]` 是两个不同的变量，前者是一个独立的变量，而后者是数组中的一个元素。上述初始化语句只是将 `numbers[0]` 的值复制给了 `firstNumber`，它们之间没有任何关联。因此，修改 `firstNumber` 的值不会影响 `numbers[0]` 的值，反之亦然。

在 C 中，我们无法直接打印整个数组，而是需要通过循环来逐个打印数组中的元素。例如：

```
1  for (int i = 0; i < 5; i++) {
2      printf("%d\n", numbers[i]);
3  }
```

上述代码使用了一个 `for` 循环来遍历数组 `numbers` 中的每个元素，并将其打印出来。`while` 循环也可以实现同样的功能，读者可以自行尝试。

在 C 语言中，数组的大小必须是一个能够在编译时确定的常量（如字面值）。

注意

变长数组（VLA）是 C99 标准引入的特性，允许数组的大小在运行时确定，但它在 C11 中被变为可选特性。容易引起误会的是，GCC 和 Clang++ 编译器提供了包含 VLA 的 GNU 扩展语法，并且默认引入这些扩展，因此，VLA（例如 `int n; int a[n];`）在这些编译器下可行。反之，如果关闭这些扩展（通过添加 `--pedantic-errors` 选项）或者非 GNU 兼容的编译器（如 MSVC），则 VLA 不可用。在实际操作中，我们不要去写 VLA，它们可能会导致代码在不同编译器下的表现不一致。C 中，我们需要使用数组但是长度不确定的时候，可以将数组开得大一些，例如题目有 1000 个元素，那么就开 1000 个元素或者稍多元素的数组。

练习：计算求和

写一个程序，该程序接受一些非零整数的输入，直到输入 0 为止，然后输出这些正整数的和。

程序输入：一系列整数，每个整数占一行，最后一个整数为 0，表示输入结束。

程序输出：一个整数，表示输入的非零整数的和。

思考：本题用数组和不用数组分别怎么写？哪种方法更好？如果本题改为“计算平均值”，用数组和不用数组分别怎么写？哪种方法更好？

8.1.9 字符串

C 风格的字符串是以字符数组的形式存储的，并以空字符（\0）结尾。字符串可以通过字符数组来表示，例如：

```
1 char str[] = "Hello, World!";
```

上述代码声明了一个字符数组 `str`，并初始化为字符串“Hello, World!”。注意，字符串的长度包括了结尾的空字符。
也就是说：

```
1 char str[3] = "Hi"; // 字符串"Hi" 占用 3 个字符：'H'、'i'和'\0'  
2 char str2[2] = "Hi"; // 错误，数组大小不足以存储字符串及其结尾的空字符
```

在做题和实际工程中，很容易遗忘 C 风格字符串的结尾空字符，因此在操作字符串时要特别小心，确保有足够的空间来存储字符串及其结尾的空字符。

练习：字符串长度

写一个程序，接受一个字符串输入，然后输出该字符串的长度（不包括结尾的空字符）。
程序输入：一个字符串，长度不超过 100 个字符。
程序输出：一个整数，表示字符串的长度。
提示：可以使用循环来计算字符串的长度，或者使用标准库函数 `strlen`。体会标准库函数在实际编程中的便利性。

8.1.10 声明、定义、变量的作用域与存储类

刚才，我们学习了 C 中最基本的一些概念，例如变量、数据类型、运算符、条件判断、循环、数组和字符串等。这些概念构成了 C 语言的基础，使我们能够编写简单的程序来解决各种问题。为了进一步深化理解这些概念，我们需要明确“声明”（declaration）和“定义”（definition）的区别，以及变量的作用域（scope）和存储类（storage class）的概念，这样才能面对复杂程序设计时的挑战。

为了理解这些内容，我们首先要知道：我们写的东西，本质上最终都会被编译器转换成机器码，而机器码是需要内存空间来存储数据的。

声明和定义 我们说过，变量需要“先声明”才能使用。我们可以通过日常中的例子来理解声明和定义的区别：假设你在学校里听说了一个叫“张三”的人，你知道他是谁（声明），但是你不知道他的具体信息（定义）。当你见到张三并了解他的详细信息时，你就完成了对张三的定义。实际上在 C 中，声明和定义的区别就在这里。

- 声明：告诉编译器指定对象的名称和类型，但不为其分配内存空间。
- 定义：为声明过的对象分配内存空间，并提供实现或提供一个初始值。

在大多数情况下，变量的声明和定义是同时进行的，例如：

```
1 int a; // 声明并定义变量 a
```

上述代码既声明了变量 `a`，也为其分配了内存空间。有的同学可能会疑惑：这哪里分配内存空间了？实际上，编译器在看到上述代码时，会为变量 `a` 分配 4 个字节的内存空间（假设 `int` 类型占 4 个字节），并将其初始化为一个值（虽然这个数值可能是垃圾值）。

只有在少数情况下，对于特定变量的声明才能分开。这种情况大多见于多文件编程中，例如：

```
1 extern int b; // 仅声明变量 b，不定义
```

上述代码只是声明了变量 `b`，并没有为其分配内存空间。变量 `b` 的定义需要在其他地方进行，例如：

```
1 int b; // 定义变量 b，分配内存空间
```

上述代码为变量 `b` 分配了内存空间，并将其初始化为一个值。

C 标准严格禁止对对象进行重定义。也就是说，不能在同一个作用域内多次定义同一个变量，否则会导致编译错误。例如：

```
1 int a = 2;  
2 printf("%d\n", a);  
3 int a = 3; // 重定义错误  
4 printf("%d\n", a);
```

上述代码中，变量 `a` 被定义了两次，导致编译失败。

C 语言确实允许多次声明同一变量，但这些多次声明必须保持一致性，即类型和名称必须相同。例如：

```
1 int a = 2;  
2 printf("%d\n", a);  
3 double a = 3.5; // 冲突声明错误  
4 printf("%f\n", a);
```

这个也是错误的，因为变量 `a` 的类型不一致，出现了“冲突声明”（conflicting declaration）。

变量的作用域与生命周期 变量的作用域是指变量在程序中可见和可访问的范围。在 C 语言中，变量的作用域向下延伸至其定义所在的代码块（block）。代码块是由一对大括号包围的代码区域。一般的，人为地将变量分为“全局”（不在任何块内声明）和“局部”（在某个块内声明）两种。一般情况下，全局变量的作用域是整个文件，甚至可以跨文件访问（通过 `extern` 关键字）。局部变量作用域仅限于该代码块内部。

也就是说：

```
1 int a = 2; // 全局变量 a
2 {
3     printf("%d\n", a); // 可以访问全局变量 a, 输出 2
4     int b = 3; // 局部变量 b
5     printf("%d\n", b); // 可以访问局部变量 b, 输出 3
6 }
7 printf("%d\n", a); // 可以访问全局变量 a, 输出 2
8 printf("%d\n", b); // 错误, 无法访问局部变量 b, 超出作用域
```

但下列代码出乎意料的合法:

```
1 int a = 2; // 作用域更大的变量 a
2 {
3     int a = 3; // 作用域更小的变量 a, 和外面的变量 a 不同, 在该作用域下提及的 a 都是这个
4     printf("%d\n", a); // 输出 3
5 }
6 printf("%d\n", a); // 输出 2
```

这在 C 中被称作“变量遮蔽” (variable shadowing): 也就是在较小的作用域中定义了一个与较大作用域中同名的变量。此时通过该名称访问的变量都是较小作用域中的变量, 较大作用域中的变量被“遮蔽”了。这并不会导致编译错误, 但可能导致语义错误 (例如程序员忘记了变量遮蔽的存在), 因此在实际编程中应尽量避免这种情况。

而变量的生命周期 (lifetime) 是指变量在内存中存在的时间段。全局变量的生命周期从程序开始运行到程序结束, 而局部变量的生命周期仅限于其所在的代码块执行期间。一旦代码块执行完毕, 局部变量就会被销毁, 其内存空间被释放。对于大多数变量来说, 生命周期与作用域是相同的; 唯一的例外是静态变量 (见下文), 其生命周期是整个程序运行期间, 但作用域仍然受限于其定义所在的代码块。

变量的存储类 变量的存储类则决定了变量的存储位置、生命周期和可见性。C 语言中常见的存储类有以下几种:

- **auto**: 默认的存储类, 表示局部变量, 生命周期为代码块执行期间。换句话说, 我们常见的 `int a;` 实际上等价于 `auto int a;`。
- **register**: 建议将变量存储在 CPU 寄存器中, 以提高访问速度, 但编译器可以忽略此建议。常用于寄存器优化, 但除非写内核或嵌入式, 否则一般没有人写这个, 而是让编译器自己决定。
- **static**: 表示静态变量, 生命周期为整个程序运行期间, 但作用域为定义所在的代码块。常用于需要在多次函数调用之间保持值的变量。
- **global**: 表示全局变量, 生命周期为整个程序运行期间, 作用域为整个文件, 甚至可以跨文件访问 (通过 `extern` 关键字)。实际上和全局定义的 `int a;` 是等价的。
- **extern**: 表示外部变量, 声明在其他文件中定义的变量, 用于跨文件访问。
- **const**: 表示常量, 值不可修改。

8.1.11 结构体和对齐

结构体 结构体（struct）是一种用户自定义的数据类型，用于将多个相关的数据组合在一起。结构体可以包含不同类型的成员变量，从而形成一个复杂的数据结构。在 C 中，结构体的定义和使用方法如下：

```
1 // 定义结构体
2 struct Person {
3     char name[50]; // 姓名
4     int age;        // 年龄
5     double height;  // 身高
6 };
7 // 使用结构体
8 struct Person person1; // 声明一个结构体变量 person1
9 // 访问和修改结构体成员
10 strcpy(person1.name, "Alice");
11 person1.age = 25;
12 person1.height = 165.5;
```

容易看出，结构体可以使得代码更加清晰和有组织，尤其是在处理复杂数据时非常有用。

有的读者可能会好奇：为什么上文是“先定义再声明”？实际上此声明非彼声明。在 C 中，结构体本身的定义（实现）实际上就是其声明；而上文的“声明一个结构体变量”指的是即为这个对象分配内存空间。也就是说，结构体类型的定义和结构体变量的声明是两个不同的东西，所以“先定义再声明”是合理的。

一个更常见的写法是使用 typedef 关键字为结构体定义一个别名，这样在声明结构体变量时就不需要再写 struct 了。例如：

```
1 // 定义结构体并使用 typedef 为其定义别名
2 typedef struct {
3     char name[50]; // 姓名
4     int age;        // 年龄
5     double height;  // 身高
6 } Person;
7 // 使用结构体
8 Person person1; // 直接使用别名 Person 来声明结构体变量 person1
```

在特殊情况下甚至允许不具名结构体，例如：

```
1 struct {
2     char name[50]; // 姓名
3     int age;        // 年龄
4     double height;  // 身高
5 } person; // 直接声明一个不具名结构体变量 person
6 // 下文可以使用 person.name 等访问成员
```

结构体的内存对齐 假设写了这样一个结构体：

```
1 struct Example {
```



```
2     char a;           // 1 字节
3     int b;            // 4 字节
4 }
5
6 struct Example example; // 声明一个结构体变量 example
7 int size = sizeof(example); // 获取结构体变量 example 的大小。
8 printf(" 结构体大小: %d\n", size); // 会输出多少?
```

乍一看，应当输出 5，因为 1+4=5。但是实际上，size 的值是 8 字节！这是为什么呢？

首先，我们先要理解“变量”在内存上究竟是什么东西。

想象内存是一条无限长的一维停车场。停车场被划分为许多连续编号的地块，这被叫做“地址”（address）。现在开来了一辆车，这辆车需要占用若干个连续的地块，这就相当于我们声明了一个变量。不同类型的变量需要占用不同数量的地块，例如int需要 4 个地块，char需要 1 个地块。抽象理解一下，变量在内存上其实就是一块连续的字节区域，例如int占 4 字节，那么它在内存上就是 4 个连续的字节；char占 1 字节，那么它在内存上就是 1 个字节。不同类型的变量在内存上占用的字节数是不同的。按理说，将所有变量密铺在内存上即可，而这确实是朴素的处理方式。

但是在实践中，我们发现，如果将变量密铺在内存上，CPU 在访问这些变量时会变得非常慢。原因在于，现代 CPU 通常以“字”（word）为单位来访问内存，而字的大小通常是 4 字节或 8 字节。如果一个变量跨越了两个字，那么 CPU 在访问这个变量时就需要进行两次内存访问，这样就大大降低了访问速度。

于是，编译器厂家等灵光一闪：为什么不能人为规定，让变量在内存上的起始地址必须是其大小的整数倍呢？这样一来，变量就不会跨越多个字，CPU 在访问这些变量时就可以一次性完成，从而提高访问速度。这个过程就叫做“内存对齐”（memory alignment）。故而，int的起始地址必须是 4 的倍数，char的起始地址可以是任意地址，double的起始地址必须是 8 的倍数，依此类推。

回到停车场比喻上：假设我们来了一辆拖车，上面载着两个小车，一个是int，一个是char。停车场管理员（CPU）登记拖车的同时，还要把小车一起登记（改变其在拖车上的位置）；又规定，int必须停在 4 的倍数号的地块上，而char可以停在任意地块上。那么，如果这个拖车（结构体）从地块 0 开始停放，那么int就必须停在地块 4 上，而char只能停在地块 0 上；中间的地块 1、2、3 就只能空着，作为填充（padding）字节。这样一来，拖车就占用了地块 0 到 7，共 8 个地块，而不是 5 个地块。这个道理和结构体的内存对齐一致，因此上述结构体的大小是 8 字节。

有了这个理论，我们就可以对结构体的内存布局进行分析了。上述结构体的内存布局如下：

地址偏移	内容
0	char a (1 字节)
1-3	填充字节 (3 字节)
4-7	int b (4 字节)

因此，整个结构体的大小是 8 字节，而不是 5 字节。

那么，我们来思考一下，怎样才能多快好省的利用内存呢？答案是：调整结构体成员的顺序，使得内存对齐时的填充字节最少。上述两个成员的顺序掉不掉换其实是一样的，都是 8 字节。但是如果结构体成员更多，顺序就会影响最终的大小。例如：

```
1 struct Example2 {
2     char a;           // 1 字节
3     double b;         // 8 字节
4     int c;            // 4 字节
5 };
```

上述结构体的内存布局如下：

地址偏移	内容
0	char a (1 字节)
1-7	填充字节 (7 字节)
8-15	double b (8 字节)
16-19	int c (4 字节)
20-23	填充字节 (4 字节)
#	这里填充的原因是：double 要求 8 字节对齐，而结构体整体大小也要是最大成员大小的整数倍。这应 ↪ 该是比较容易证明的数论事实。

因此，整个结构体的大小是 24 字节。但如果我们调整成员顺序，例如：

```
1 struct Example3 {
2     double b;         // 8 字节
3     int c;            // 4 字节
4     char a;           // 1 字节
5 };
```

上述结构体的内存布局如下：

地址偏移	内容
0-7	double b (8 字节)
8-11	int c (4 字节)
12	char a (1 字节)
13-15	填充字节 (3 字节)

因此，整个结构体的大小是 16 字节。通过调整成员顺序，我们节省了 8 字节的内存空间。所以说，在定义结构体时，合理安排成员的顺序是非常重要的，要学会“见缝插针”，以最大限度地利用内存空间。

练习：学生信息管理

写一个程序用于管理学生的高考信息（仅包括学号、姓名、分数）。学号从 0 开始连续编号，姓名不超过 20 个字符，分数为整数，在 0 到 750 之间。

程序输入：首先输入一个整数 n ，表示学生人数。接下来输入 n 行，每行包含一个学生的姓名和分数，姓名和分数之间用空格分隔。然后输入一个整数 m ，表示查询次数。接下来输入 m 行，每行包含一个学生的学号。

程序输出：m 行。每一行用空格分隔输出三个整数，分别对应查询学号的学生的学号、姓名和分数。如果未能查询到，输出 “Not Found”。

提示：本题使用结构体、不使用结构体分别怎么写？哪种方法更好？体会结构体在组织复杂数据时的优势。

8.1.12 联合体

联合体（union）是一种特殊的数据类型，它允许在同一内存位置存储不同类型的数据。联合体的所有成员共享同一块内存，因此在任何时候只能使用其中的一个成员。联合体的定义和使用方法如下：

```
1 union Data
2 {
3     int intValue;      // 整数值
4     float floatValue;  // 浮点值
5 };
6 // 使用联合体
7 union Data data; // 声明一个联合体变量 data
8 // 访问和修改联合体成员
9 data.floatValue = 10.0; // 设置值
10
11 printf(" 浮点值: %f\n", data.floatValue); // 访问浮点值
12 printf(" 整数值: %d\n", data.intValue); // 访问整数值（未定义行为）
```

在上述代码中，联合体 Data 包含两个成员：intValue 和 floatValue。当我们设置 floatValue 的值时，intValue 的值会被覆盖，反之亦然。因此，在使用联合体时需要特别小心，确保只访问当前有效的成员。

联合体和结构体实际上是类似的，所以“定义联合体结构”和“声明联合体变量”自然也是分开的；同理，也可以用 typedef 为联合体定义别名，或使用不具名联合体。

联合体的内存模型极为简单，其大小等于其最大成员的大小，这是所有成员共享同一块内存的结果。例如，上述联合体 Data 的大小为 4 字节，因为 int 和 float 都占用 4 字节。而下列联合体

```
1 union Example {
2     char a;      // 1 字节
3     double b;    // 8 字节
4     int c;       // 4 字节
5 };
```

的大小为 8 字节，因为 double 是最大的成员，占用 8 字节。

8.1.13 函数

函数是程序中的一个独立模块，用于执行特定的任务。函数可以接受输入参数，执行一些操作，并返回一个结果。使用函数可以提高代码的可读性和可维护性。

函数的声明、定义和调用如下实例所示：

```
1 // 函数声明
2 int add(int, int); // 这里可以省略参数名
3
4 // 函数定义
5 int add (int a, int b) { // 需要保证和定义时的参数类型一致，且不能省略参数名、不能改变顺序
6     return a + b;
7 }
8
9 // 函数调用
10 int sum = add(3, 5);
11 printf(" 和是: %d\n", sum);
```

上述代码定义了一个名为 `add` 的函数，它接受两个整数参数，并返回它们的和。然后，我们调用该函数并将结果存储在变量 `sum` 中，最后打印出结果。

我们一般把上述 `a` 和 `b` 叫做“形参”（parameter），而把 3 和 5 叫做“实参”（argument）。形参是在函数定义时使用的变量名，用于表示函数接受的输入参数；实参是在函数调用时传递给函数的具体值。

我们可以在一个函数内部定义另一个函数（即不支持嵌套函数）。`main` 函数也是一个函数，只不过它是程序的入口点，因此也不能在 `main` 里面定义另一个函数。

我们发现，在定义上述 `add` 函数时，使用了两个参数 `a` 和 `b`。这两个参数在函数内部是可以使用的，但是在函数外部是无法访问的。这是因为函数参数的作用域仅限于函数内部。换句话说，函数参数在函数外部是不可见的，无法被访问或修改。

函数的内存模型是相当复杂的，涉及到函数调用栈、参数传递方式（按值传递或按引用传递）、返回值处理等多个方面。一般来说，函数调用时会在栈上分配一块内存空间，用于存储函数的参数、局部变量和返回地址等信息。当函数执行完毕后，这块内存空间会被释放。而大多数情况下，我们也不会去纠结一个函数执行一次需要吃掉多少内存，但必须知道“会吃掉内存”这个事实。

练习：日期差加强版

写一个程序，接受两个日期输入，计算两者之间差了多少天。

程序输入：空格分隔的 6 个整数 `year1`、`month1`、`day1`、`year2`、`month2`、`day2`，分别表示第一个日期的年份、月份和天数，以及第二个日期的年份、月份和天数。

程序输出：一个整数，表示两个日期之间的天数差。

提示：把前面“天数判断”“闰年判断”题目作为子任务来完成，也就是说可以试着复用这些代码。考虑使用函数来组织代码，提高代码的可读性和可维护性。

8.1.14 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：

```
1 int foo(){
2     if (base_case) {
3         return base_value; // 基础情况，直接返回结果
```

```
4     } else {  
5         return foo(); // 递归调用  
6     }  
7 }
```

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：

```
1 int factorial(int n) {  
2     if (n == 0) {  
3         return 1; // 基础情况  
4     } else {  
5         return n * factorial(n - 1); // 递归调用  
6     }  
7 }
```

建立递归思维是非常困难的，但也是非常重要的。在实际生活中，很多问题都可以通过“分治-递归”的思路来解决：把大问题分成相似的小问题，解决这些小问题，然后把小问题的解合并成大问题的解。递归函数正是实现这种思路的有力工具。

练习：小明爬楼梯

小明在爬楼梯。他一次可以爬 1 个或 2 个台阶。假设楼梯有 n 个台阶，问小明有多少种不同的爬法？

程序输入：一个整数 n ，表示楼梯的台阶数。

程序输出：一个整数，表示小明爬楼梯的不同方法数。

提示：考虑：假设小明爬到 x 级台阶时的爬法有 $f(x)$ 种，那么 $f(x)$ 能不能被它前面的某些项表示出来？基础情况又是什么？这个递推关系就是大名鼎鼎的**状态转移方程**，是很多复杂问题的核心。

递归函数虽然很有效，但是开销非常庞大。我们知道，函数只有在执行完毕时才会销毁其栈帧，而递归函数在每次递归调用时，都会创建一个新的栈帧（但旧的栈帧因为没有执行完毕而不会销毁）。积少成多之下，递归函数在执行过程中会占用大量的内存空间，尤其是递归层数很深时，可能会导致栈溢出错误（`stack overflow`）。此外，递归函数的调用和返回也会带来额外的时间开销，因为每次调用都需要保存和恢复上下文信息。故而，在允许的情况下，应尽量避免使用递归函数，转而使用迭代（`iteration`）或其他更高效的算法。

```
1 int facts[100]; // 假设最大计算到 99 的阶乘  
2 facts[0] = 1; // 基础情况  
3 for (int i = 1; i < 100; i++) {  
4     facts[i] = i * facts[i - 1]; // 迭代计算  
5 }
```

这样就能避免递归调用带来的巨大开销，但其思路本质和递归相似：递归是从问题本身出发，不停地分解成小问题；而迭代则是从基础情况出发，不停地构建成大问题。而迭代递推则是动态规划这类问题的核心思路。

8.1.15 类型强转

类型强转（type casting）是将一种数据类型转换为另一种数据类型的过程，毕竟大家都不想让 5 除以 2 得 2。

用括号就可以实现类型强转。例如：

```
1 int a = 5;
2 int b = 2;
3 double result = (double)a / (double)b; // 强制将 a 和 b 转换为 double 类型
4 printf(" 结果是: %f\n", result); // 输出结果
```

在上述代码中，我们将整数变量 a 和 b 强制转换为 double 类型，然后进行除法运算。如果不进行类型强转，整数除法会导致结果被截断为整数部分，得到 2；而通过类型强转，我们可以得到正确的浮点数结果 2.5。

类型强转在处理不同数据类型之间的运算时非常有用，可以确保运算结果符合预期。

练习：求平均数

写一个程序，接受一系列整数输入，直到输入 0 为止，然后输出这些整数的平均值（不包括结尾的 0）。

程序输入： 一系列整数，每个整数占一行，最后一个整数为 0，表示输入结束。

程序输出： 一个浮点数，表示输入整数的平均值，保留两位小数。

提示： 虽然把输入的整数定义为浮点数是可以避免类型强转的，但在金融上这会产生误差，是不可接受的。因此不得将输入的整数定义为浮点数，而是要定义为整数类型、加和，再通过类型强转来计算平均值。

8.1.16 宏和预处理指令

宏是一种预处理指令，它可以在编译之前对代码进行替换和扩展。宏的基本格式如下：

```
1 #define 宏名 替换内容
```

宏在编译器对代码进行预处理的时候进行纯文本替换。宏名通常使用大写字母来表示，以便于和变量区分。替换内容可以是任意的代码片段，包括变量、表达式、语句等。宏常用于定义常量，但是用宏定义的常量没有类型，而是字面值。

我们可能会看到，诸如 #define、#include 等均以符号 # 开头，这些都是预处理指令，有时候也叫做编译指令。预处理指令和常规代码的行为有区别：它们实际上并非代码的一部分，而是在编译器对代码进行预处理的时候进行处理的。预处理指令通常用于定义宏、包含头

文件、条件编译等。常用的预处理指令还有 `#pragma`、`#ifdef` 等。活用编译指令可以让代码更灵活、更高效。

警告

严格禁止使用所谓的“火车头”预处理指令！
所谓的火车头预处理指令，指的是在代码的开头使用大量的 `#pragma` 来指定编译器的行为。这种做法显著地导致了代码的可移植性和可维护性变差。因为不同的编译器对 `#pragma` 的支持程度不同，甚至同一编译器的不同版本对某些 `#pragma` 的支持也可能不同。而且你辛辛苦苦打一大堆 `#pragma`，实际上优化效果还不如一个简单的 `-O3`。这种完全属于歪门邪道的做法，严重违反了代码简洁和可维护的原则。

8.2 指针和内存操作

指针是 C 语言的最重要特性，没有之一。该特性彻底奠定了 C 语言在系统编程领域的统治地位。但对于新手而言，要理解指针难度还是比较大的，因此我们会尽量用通俗易懂的语言来解释指针的概念和使用方法；读者一定要确保理解该内容，而不是背“八股”式的语法，否则后续内容将会变得非常困难。

当然，如果我们先前讲的内存模型都理解了的话，理解指针其实并不难。

8.2.1 什么是指针

所有教材（甚至包括 C 标准）中，对指针的定义实际上都是“一个变量，它存储了另一个变量的内存地址”。但是，这个定义对于初学者来说过于抽象，难以理解。因此，我们可以用一个更形象的比喻来解释指针的概念。

想象内存是一条很长很长的一维停车场，停车场被划分为许多连续编号的地块，也就是“地址”（address）。

现在我们 `int a = 42`；。于是，编译器给 `a` 分配了 4 个连续的地块（假设 `int` 类型占 4 字节），并把这 4 个地块的编号（地址）记为 `0x1000`、`0x1001`、`0x1002`、`0x1003`。于是，我们可以说变量 `a` 的“地址”是 `0x1000`，告诉我们变量 `a` 停放在内存的哪个位置；而它的值是 42，告诉我们变量 `a` 里面存放的东西是什么。

上述内容可以记作：

```
1  int* p = &a;
2  int *p = &a; // 或者这样，但实际没有任何区别
```

说明

星号写在哪里都无所谓，甚至

```
1  int*p = &a;
2  int * p = &a;
```


也都是合法的。四种写法实际上完全等价，编译器认为上述写法完全等价。

C 语言程序员大多习惯 `int *p = &a;`，认为这样更符合语言的习惯；而 C++ 程序员大多习惯 `int* p = &a;`，认为这样更能体现指针类型的本质（但“指针类型”这个概念实际上是 C++ 引入的，C 语言并没有这个概念）。实际的代码应符合团队的代码风格规范。为了便于理解，我们在本书中统一使用 `int* p = &a;` 这种写法。

可以看到，上述 `&a` 就是“取地址”，也就是获取变量 `a` 的地址。而 `int* p` 则是声明了一个指针变量 `p`，它的类型是“指向 `int` 类型变量的指针”，也就是说，`p` 只能存储 `int` 类型变量的地址。于是，我们就可以说，`p` 存储了变量 `a` 的地址 `0x1000`。

那么怎么用这个指针呢？我们可以通过指针来访问和修改变量的值。例如：

```
1  *p = 100; //
2  printf("%d\n", p); // 输出指针 p 的值（地址）
3  printf("%d\n", a); // 输出变量 a 的值
```

我们惊奇的发现，虽然我们看似修改的是 `p`，但 `p` 并没有改变，但 `a` 变了！这是为什么呢？这是因为 `*p` 被称作“解引用”（dereference），表示“通过指针 `p` 访问它所指向的变量”，也就是“通过地址 `0x1000` 访问地块里的东西”。因此 `*p = 100;` 的意思就是“把 `p` 指向的地块里的东西改成 100”，也就是把变量 `a` 的值改成 100。

那么如果这样呢？

```
1  p = 100;
2  printf("%d\n", p); // 输出指针 p 的值（地址）
3  printf("%d\n", a); // 输出变量 a 的值
4  printf("%d\n", *p); // 试图通过指针 p 访问它所指向的变量
```

这样，`p` 确实是 100 了，但 `a` 并没有变。这是因为这里我们修改的是指针 `p` 本身，而不是通过指针 `p` 访问的变量。因此，变量 `a` 的值保持不变。

但是当我们试图通过指针 `p` 访问它所指向的变量时，程序可能会崩溃！这是因为 `p` 现在指向的是地址 100，而这个地址并没有被分配给任何变量，因此访问这个地址会导致未定义行为！这被叫做“悬空指针”（dangling pointer），俗称“野指针”。因此，在使用指针时，一定要确保指针指向的是一个有效的变量。

因此，在指针中，两个运算符不要弄反：

- `&`：取地址运算符，用于获取变量的地址，或“门牌号”。
- `*`：解引用运算符，用于通过指针访问变量的值，或“门牌号对应房间里的东西”。

有一种特殊的指针被称为“空指针”（null pointer），可以理解为“该指针没有指向任何地块”，常用作指针的初始值或者表示指针不指向任何有效变量。在 C 中，可以使用宏 `NULL` 来表示空指针。例如：

```
1  free(p); // 释放内存，此时 p 指向的内存不再有效
2  p = NULL; // 释放内存后，立刻将指针设置为 NULL，避免悬空指针
```


8.2.2 指针的三条铁律

在使用指针时，有三条铁律需要牢记于心：

- 指针存储的是地址（门牌号），类型必须匹配；`int*` 类型的指针只能存储 `int` 类型变量的地址，`char*` 类型的指针只能存储 `char` 类型变量的地址，依此类推。至于原因，看到下文就明白了。
- 指针必须初始化！直接 `int* p;` 会得到一个野指针，里面是一个垃圾数值，千万不要使用它，用了大概率段错误。要是真想这么干，声明空指针即可。
- 用完的内存要还。这个后面讲到动态内存分配时会讲到为什么。

8.2.3 指针和数组、函数的配合

指针和数组

在 C 中，数组名实际上是一个指向数组第一个元素的指针。因此，我们可以使用指针来访问和操作数组元素。而指针的运算也往往无法脱离数组来理解。

例如：

```
1 int numbers[] = {10, 20, 30, 40, 50};
2 int* p = numbers; // 数组名作为指针，指向第一个元素
3 for (int i = 0; i < 5; i++) {
4     printf("%d\n", *(p + i)); // 通过指针访问数组元素
5 }
```

在上述代码中，`numbers` 是一个数组名，它在表达式（和函数传参）中，会退化成首元素的地址，因此 `int* p = numbers;` 实际上等价于 `int* p = &numbers[0];`。

而上述代码中的 `*(p + i)` 则是通过指针运算来访问数组元素。这里，`p + i` 表示指针 `p` 向后移动 `i` 个元素的位置，而 `*` 则用于解引用该位置，从而获取对应的数组元素的值。实际上上述计算的意思是，“从地址 `p` 开始，向后移动 `i` 个 `int` 类型的字节数，然后访问该地址对应的值”。`*(p + i)` 事实上等价于 `numbers[i]`。

与之类似的，`++p` 表示指针 `p` 向后移动一个元素的位置，而 `p+1` 则表示指针 `p` 向后移动一个元素的位置，但并不改变指针 `p` 本身。

这就解释了为什么指针类型必须匹配的问题：如果指针类型不匹配，那么指针运算时移动的字节数就会出错，从而导致访问错误的内存地址，进而引发未定义行为。

指针和函数

指针和函数的配合主要体现在函数参数传递上。

我们可以写一段代码来说明这个问题：

```
1 void swap(int x, int y){
2     int temp = x;
3     x = y;
4     y = temp;
5 }
```

```
5 }  
6  
7 swap(a, b);  
8 printf("a = %d, b = %d\n", a, b); // 输出结果
```

我们惊奇的发现，虽然写了一个交换函数，但是实际上根本没有交换 `a` 和 `b` 的值！这是因为在 C 中，函数参数是通过值传递的，也就是说，当我们调用 `swap(a, b);` 时，实际上是将 `a` 和 `b` 的值复制了一份传递给函数 `swap` 的参数 `x` 和 `y`。因此，在函数内部对 `x` 和 `y` 的修改并不会影响到外部的 `a` 和 `b`。

那么怎么才能真正去影响 `a` 和 `b` 呢？这时就需要用到指针了。我们可以将 `a` 和 `b` 的地址传递给函数，然后在函数内部通过指针来修改它们的值。例如：

```
1 void swap(int* x, int* y){  
2     int temp = *x;  
3     *x = *y;  
4     *y = temp;  
5 }  
6  
7 swap(&a, &b);  
8 printf("a = %d, b = %d\n", a, b); // 输出结果
```

这次运行，就能真正交换 `a` 和 `b` 的值了。这是因为我们将 `a` 和 `b` 的地址传递给了函数 `swap` 的参数 `x` 和 `y`，然后在函数内部通过解引用指针来修改它们所指向的变量的值，而非仅仅复制一份值。

函数指针

函数指针则是指向函数的指针变量。通过函数指针，我们可以动态地调用不同的函数，从而实现更灵活的代码结构。例如：

```
1 int add(int a, int b) {  
2     return a + b;  
3 }  
4 int multiply(int a, int b) {  
5     return a * b;  
6 }  
7  
8 int (*funcPtr)(int, int);  
9 // 将函数指针指向 add 函数  
10 funcPtr = add;  
11 printf("5 + 3 = %d\n", funcPtr(5, 3)); // 调用 add 函数  
12 // 将函数指针指向 multiply 函数  
13 funcPtr = multiply;  
14 printf("5 * 3 = %d\n", funcPtr(5, 3)); // 调用 multiply 函数
```

这样能够让我们在运行时选择要调用的函数，从而实现更灵活的代码结构。

8.2.4 动态内存分配

动态内存分配是指在程序编译时不知道用多少内存，于是在运行时根据需要动态地分配和释放内存空间。

在 C 中，动态内存分配主要通过以下三个函数来实现：

- `malloc(size_t size)`：用于分配指定大小的内存块，返回一个指向该内存块的指针。如果分配失败，返回 `NULL`。
- `calloc(size_t num, size_t size)`：用于分配指定数量的内存块，并将其初始化为零。返回一个指向该内存块的指针。如果分配失败，返回 `NULL`。
- `free(void* ptr)`：用于释放之前分配的内存块。参数 `ptr` 是指向要释放的内存块的指针。

例如：

```
1 int n;
2 scanf("%d", &n); // 读取数组大小
3 // 动态分配一个包含 n 个整数的数组
4 int* arr = (int*)malloc(n * sizeof(int));
5 if (arr == NULL) {
6     perror(" 内存分配失败");
7     exit(EXIT_FAILURE);
8 }
9 // 使用数组
10 for (int i = 0; i < n; i++) {
11     arr[i] = i * 2; // 初始化数组元素
12 }
13 // 释放内存
14 free(arr);
15 arr = NULL; // 好的实践，立即置空，防止悬空指针
```

在上述代码中，我们首先读取了数组的大小 `n`，然后使用 `malloc` 函数动态分配了一个包含 `n` 个整数的数组。接着，我们使用该数组进行了一些操作，最后使用 `free` 函数释放了之前分配的内存。如果不释放这个内存，那么程序常驻时会把内存吃光，导致系统崩溃，这被称为“内存泄漏”；如果不小心释放了两次同一块内存，程序也会崩溃，这被称为“双重释放”。这两个都是非常严重的错误，必须避免。

需要说明的是，`malloc` 返回的是无类型指针 (`void*`)，C 允许直接赋值给任何其他指针类型（例如 `int*`），这是 C 特有的，而 C++ 就不允许这么写。而在 C 中，我也推荐在赋值前进行强制类型转换。

8.2.5 静态变量和 `const` 指针

如果试图想在函数中保存一些状态信息，可以考虑使用静态变量。例如：

```
1 int* foo(){
2     int x = 42; // 自动变量
3     return &x; // 错误，返回局部变量地址，x 作为局部变量在函数结束后被销毁
4 }
5
```

```
6 int* foo_fixed(){
7     static int x = 42; // 静态变量
8     return &x; // 正确, 返回静态变量地址, x 在程序运行期间始终存在
9 }
```

至于 const 指针, 则很特殊:

```
1 int a = 10;
2 const int* p1 = &a; // 指向常量的指针, 不能通过 p1 修改 a 的值
3 int* const p2 = &a; // 常量指针, 不能修改 p2 的值, 但可以通过 p2 修改 a 的值
4 const int* const p3 = &a; // 谁都想动我
```

这个估计只能死记硬背了。

8.2.6 初级泛型编程: void 指针

在C语言中,void* 指针是一种特殊的指针类型, 它可以指向任何类型的数据。由于 void* 指针没有具体的类型信息, 因此在使用时需要进行类型转换 (type casting) 才能访问其指向的数据。但这给了程序员极大的自由度, 很多 C 程序员都是 void* 大神。

首先必须说明: void* 指针不能进行解引用操作, 因为编译器无法确定其指向的数据类型和大小; 也不能对其进行加减运算, 因为编译器无法确定每次移动的字节数。故而, 实际上只能把 void* 指针作为一种通用的指针类型来传递数据, 而不能直接操作数据。

例如:

```
1 void printValue(void* ptr, char type) {
2     if (type == 'i') {
3         int* intPtr = (int*)ptr; // 将 void 指针转换为 int 指针
4         printf(" 整数: %d\n", *intPtr);
5     } else if (type == 'f') {
6         float* floatPtr = (float*)ptr; // 将 void 指针转换为 float 指针
7         printf(" 浮点: %f\n", *floatPtr);
8     }
9 }
10 int main() {
11     int a = 42;
12     float b = 3.14f;
13
14     printValue(&a, 'i'); // 传递整数
15     printValue(&b, 'f'); // 传递浮点数
16
17     return 0;
18 }
```

在上述代码中, 函数 printValue 接受一个 void* 指针和一个类型标识符作为参数。根据类型标识符的值, 函数将 void* 指针转换为相应类型的指针, 并通过解引用操作访问其指向的数据。这是很初级的“泛型”编程: 通过 void* 指针, 我们可以编写能够处理不同数据类型的通用函数, 从而提高代码的复用性和灵活性。

但这种操作也是极为令人头疼的，因为你必须自己保证传递的类型标识符和实际数据类型一致，否则就会导致未定义行为；而有些代码库中，甚至调用某个函数还会返回 `void*` 指针，不深入理解的情况下甚至不知道这是个什么东西，令人并不愉快。所以一定要谨慎使用 `void*` 指针。

8.2.7 指针常见错误

指针是 C 语言中非常强大但也非常容易出错的特性。以下是一些常见的指针错误（其实我大多都提到过了）：

- 没初始化：出现这种情况应该自罚三杯。
- 数组越界：一不小心访问了数组之外的内存地址，可能会导致程序崩溃或数据损坏。解决方法是确保访问的索引在数组的有效范围内。
- 返回局部变量地址：函数中的局部变量会随着函数的结束而销毁，因此试着返回它们的地址（或在函数外使用它们的地址）会导致悬空指针。解决方法是将变量声明为静态变量。
- `free` 以后忘了，接着用：释放内存后继续使用该内存地址会导致未定义行为。解决方法是，`free` 之后，立刻把指针置为 `NULL`，防止悬空指针。
- 把 `int` 强转成指针乱玩：除非你知道你在做什么，否则不要这么做。

练习：指针练习题

编写一个函数，接受一个整数数组和它的大小作为参数，返回数组中的最大值和最小值。

程序输入：一个整数 `n`，表示数组的大小，接着是 `n` 个整数，表示数组的元素。

程序输出：两个整数，分别表示数组中的最大值和最小值。

提示：试着使用指针来遍历数组，并在函数中返回最大值和最小值。另，试着使用动态的内存分配来创建实际上的动态数组，而不是写 VLA 或预先写一个巨大的静态数组。

8.3 文件操作

8.3.1 文件读写

C 的文件操作也是基于指针的。

在 C 中，文件操作主要通过标准库中的 `FILE` 结构体和相关函数来实现。`FILE` 结构体表示一个文件流，它包含了文件的状态信息和缓冲区等数据。我们可以使用指向 `FILE` 结构体的指针来操作文件。为了使用文件操作功能，我们需要包含头文件 `stdio.h`。

例如，我们需要使用以下手段来进行文件操作：

- 打开文件：使用 `fopen` 函数打开一个文件，返回一个指向 `FILE` 结构体的指针。
- 读取文件：使用 `fread` 或 `fgets` 等函数从文件中读取数据。
- 写入文件：使用 `fwrite` 或 `fputs` 等函数向文件中写入数据。
- 关闭文件：使用 `fclose` 函数关闭文件，释放资源。

具体而言，下面是一个简单的文件操作示例：

```
1 #include <stdio.h>
2
3 int main() {
4     // 打开文件
5     FILE* file = fopen("example.txt", "w");
6     if (file == NULL) {
7         perror(" 无法打开文件");
8         return 1;
9     }
10
11     // 写入数据
12     const char* text = "Hello, World!\n";
13     fwrite(text, sizeof(char), strlen(text), file);
14
15     // 关闭文件
16     fclose(file);
17     return 0;
18 }
```

`fopen` 函数接受两个参数：文件名（实际上是文件相对可执行文件的路径）和模式。模式可以是 `r`（只读）、`w`（只写，文件不存在则创建，存在则清空）、`a`（追加写入，文件不存在则创建）等。函数返回一个指向 `FILE` 结构体的指针，如果打开失败则返回 `NULL`。

8.3.2 文件操作

除了文件的读写外，我们有时候还希望删除、重命名文件等操作。C 标准库提供了一些函数来实现这些功能，例如：

- `remove(filename)`：删除指定的文件。
- `rename(old_filename, new_filename)`：重命名文件。
- `fseek(file_ptr, offset, whence)`：设置文件指针的位置。
- `ftell(file_ptr)`：获取文件指针的当前位置。
- `rewind(file_ptr)`：将文件指针重新定位到文件的开头。
- `feof(file_ptr)`：检查是否到达文件末尾。
- `ferror(file_ptr)`：检查文件操作是否出错。

例如，我们想删除一个文件，可以使用 `remove` 函数：

```
1 if (remove("example.txt") == 0) {
2     printf(" 文件删除成功\n");
3 } else {
4     perror(" 文件删除失败");
5 }
```

`perror` 函数用于打印最近一次系统调用失败的错误信息。

8.4 多文件编程

8.4.1 头文件和源文件

在实际工程中，如果把所有的代码都写在一个文件中，代码量会非常庞大，难以维护和管理；如果遇到多人协作开发，问题会更加严重。因此，我们需要将代码拆分成多个文件，每个文件负责不同的功能模块，从而提高代码的可维护性和可读性。在 C 中，代码被分为“头文件”和“源文件”两种类型。

头文件（header files）通常以 .h 作为文件扩展名，包含函数声明、宏定义、结构体定义等内容。源文件（source files）通常以 .c 作为文件扩展名，包含函数的具体实现和程序的入口点（main 函数）。通过将代码拆分成头文件和源文件，我们可以实现代码的模块化和重用。我们可以在多个源文件中包含同一个头文件，从而共享函数声明和数据结构定义。例如：

```
1 // math_utils.h
2 #ifndef MATH_UTILS_H
3 #define MATH_UTILS_H
4 int add(int a, int b);
5 int subtract(int a, int b);
6 #endif // MATH_UTILS_H
```

```
1 // math_utils.c
2 #include "math_utils.h" // 包含头文件
3 int add(int a, int b) {
4     return a + b;
5 }
6 int subtract(int a, int b) {
7     return a - b;
8 }
```

```
1 // main.c
2 #include <stdio.h>
3 #include "math_utils.h" // 包含头文件
4 int main() {
5     int x = 10;
6     int y = 5;
7     printf("x + y = %d\n", add(x, y));
8     printf("x - y = %d\n", subtract(x, y));
9     return 0;
10 }
```

实际编译时，可以使用以下命令将多个源文件编译成一个可执行文件：

```
1 gcc main.c math_utils.c -o my_program
```

在上述代码中，我们将数学运算函数的声明放在头文件 math_utils.h 中，将函数的实现放在源文件 math_utils.c 中，然后在主程序 main.c 中包含头文件并调用这些函数。

头文件中的 `#ifndef`、`#define` 和 `#endif` 是一种常见的防止重复包含的技术，称为“包含保护”（include guard），也叫“编译守卫”。它确保头文件只被包含一次，避免了重复定义的问题。

8.4.2 include 原理

我们可能会疑惑：为什么在上述编译指令中，只需要指定源文件，而不需要指定头文件呢？编译守卫又是怎么起作用的呢？这是因为头文件实际上并不是独立的编译单元，而是通过预处理指令 `#include` 被包含到源文件中的。为了理解这个，我们先简要理解一下编译的过程。

编译大致分为以下几个阶段：

- 预处理（Preprocessing）：处理预处理指令，如 `#include`、`#define` 等，除去注释，生成纯净的源代码。这一步也叫“预编译”。
- 编译（Compilation）：将预处理后的源代码转换为汇编代码。
- 汇编（Assembly）：将汇编代码转换为机器码，生成目标文件（通常以 `.o` 或 `.obj` 作为扩展名）。
- 链接（Linking）：将多个目标文件和库文件链接成一个可执行文件。

`#include` 是一种**预编译指令**，其意思可以理解为“把指定的头文件的内容直接插入到当前位置”。因此，当我们编译源文件时，编译器会先处理所有的 `#include` 指令，将头文件的内容插入到源文件中，然后再进行编译。换句话说，在预编译这一步结束后，上述的 `math_utils.c`（实际上不会叫这个名字了）会变成这样：

```
1 int add(int a, int b);
2 int subtract(int a, int b);
3 int add(int a, int b) {
4     return a + b;
5 }
6 int subtract(int a, int b) {
7     return a - b;
8 }
```

同理，`main.c` 也会变成类似的样子。

而编译守卫解决的是诸如这种问题：假设我们写了两个头文件，姑且分别叫做甲、乙，其中乙头文件包含了甲头文件。这时，我们写一个源文件丙，其中 `include` 了乙头文件。那么在预编译时，丙会先包含乙，然后乙又包含甲，这是正确无误、不会报错的；但如果丙又直接包含了甲头文件，那么甲头文件就会被包含两次，导致一些函数和变量被重复定义，从而引发编译错误。编译守卫通过定义一个宏来防止头文件被重复包含，从而避免了这个问题。

所以说，如果没有这个头文件，那么编译器在编译 `main.c` 时就会报错，提示找不到 `add` 和 `subtract` 的声明。

8.4.3 链接原理

在编译过程中，编译器会把多个源文件分别处理成多个目标文件。而把多个“目标文件”真正合并成一个可执行文件的过程，叫做“链接”（linking）。链接器会把各个目标文件中的符

号（函数和变量）进行解析和合并，从而生成最终的可执行文件。

具体的合并过程大致是这样的。如果我们直接编译上述 `main.c`，并查看其汇编，会发现：

```
1  call    add      0x00401000 <add> # 这是胡编的地址，仅作示例
2  call    subtract 0x00401010 <subtract> # 同上
```

可以看到，编译器并没有把 `add` 和 `subtract` 函数的具体实现插入到 `main` 函数中，而是生成了对这两个函数的调用指令，并且使用了占位符地址。这是因为编译器确实看到了这两个函数的声明，但并不知道这两个函数的具体实现在哪里，故而知道它们是**外部函数**。故而，编译器会在该源文件的目标文件中，生成对这两个外部函数的“标记”（也就是**符号**），表示“这里要使用 `add` 函数，地址待定”。

而真正定义这两个东西的是 `math_utils.c`，它会被编译成另一个目标文件，其中包含了这两个函数的具体实现和地址信息，即**符号定义**。

链接器的核心职责就是把这堆东西合并成一个，其最重要的任务就是“符号解析”：

- 扫描所有的目标文件，收集每一个文件提供的符号定义和需要的符号引用。
- 对每一个符号引用，查找对应的符号定义，并将引用地址替换为定义地址。
- 处理未定义符号，如果有符号引用找不到对应的定义，链接器会报错。

以上述代码为例，链接器在处理 `main.o` 时，发现它引用了 `add` 和 `subtract` 这两个符号；然后它扫描 `math_utils.o`，发现它定义了这两个符号，并且知道它们的具体地址；最后，链接器将 `main.o` 中对这两个符号的引用地址替换为它们在 `math_utils.o` 中的实际地址，从而完成链接过程。

8.5 标准库常用头文件

C 标准库头文件按照 C17 标准一共 29 个，其中有一些方法是我们经常会用到的。下面列出一些常用的头文件及其主要功能，基本上覆盖了 C 代码八成以上的需求。剩余的头文件，读者可以根据需要自行查阅相关资料。

8.5.1 `stdio.h`

该库主要负责输入输出操作。除了 `scanf` 和 `printf`，还包括文件操作等功能。

- `fopen(filename, mode)`：打开文件，返回一个文件指针。
- `fclose(file_ptr)`：关闭文件。
- `fread(buffer, size, count, file_ptr)`：从文件中读取数据到缓冲区。
- `fwrite(buffer, size, count, file_ptr)`：将缓冲区的数据写入文件。
- `fprintf(file_ptr, format, ...)`：格式化输出到文件。
- `fscanf(file_ptr, format, ...)`：格式化从文件读取数据。

8.5.2 stdbool.h

该库主要负责布尔类型的定义和操作。它定义了一个名为 `bool` 的数据类型，以及两个宏 `true` 和 `false`，分别表示布尔值的真和假。

8.5.3 string.h

该库主要负责字符串操作，顺带一些内存操作。常用函数包括：

- `strlen(str)`：返回字符串的长度（不包括结尾的空字符）。
- `strcpy(dest, src)`：将源字符串 `src` 复制到目标字符串 `dest` 中。
- `strcat(dest, src)`：将源字符串 `src` 连接到目标字符串 `dest` 的末尾。
- `strcmp(str1, str2)`：比较两个字符串 `str1` 和 `str2` 的大小关系。
- `strchr(str, ch)`：在字符串 `str` 中查找字符 `ch` 的第一次出现位置。
- `strstr(str1, str2)`：在字符串 `str1` 中查找子字符串 `str2` 的第一次出现位置。
- `memcpy(dest, src, n)`：将源内存块 `src` 的前 `n` 个字节复制到目标内存块 `dest` 中。
- `memset(dest, val, n)`：将目标内存块 `dest` 的前 `n` 个字节设置为值 `val`。该方法用来清理数组非常方便。

8.5.4 stdlib.h

该库主要负责内存分配、程序控制和数值转换等功能。常用函数包括：

- `malloc(size_t size)`：分配指定大小的内存块。
- `calloc(size_t num, size_t size)`：分配指定数量的内存块，并将其初始化为零。
- `free(void* ptr)`：释放之前分配的内存块。
- `atoi(str)`、`atof(str)`、`strtoul(str, endptr, base)` 等：将字符串转换为整数或浮点数。
- `qsort(base, nmemb, size, compar)`：对数组进行快速排序。
- `bsearch(key, base, nmemb, size, compar)`：在已排序的数组中进行二分查找。
- `realloc(ptr, size)`：重新分配内存块的大小。
- `exit(status)`：终止程序的执行，并返回状态码。

8.5.5 math.h

该库主要负责一些数学运算函数。常用函数包括：

- `sqrt(x)`、`pow(x, y)`、`sin(x)`、`cos(x)`、`tan(x)`、`log(x)`、`exp(x)` 等：各种数学函数，一目了然。
- `abs(x)`、`fabs(x)`：计算整数或浮点数的绝对值。
- `ceil(x)`、`floor(x)`：向上取整和向下取整函数。
- `round(x)`：四舍五入函数。
- `fmod(x, y)`：计算浮点数的余数。

练习：改错练习

以下代码均有错误或未定义行为或不良实践，请指出并改正。

```
1 #include <stdio.h>
2 int main(){
3     int n;
4     int arr[n];
5     scanf("%d", &n);
6     for (int i = 0; i <= n; i++) {
7         scanf("%d", &arr[i]);
8     }
9     return 0;
10 }
```

```
1 #include <stdio.h>
2 int main(){
3     int arr[5];
4     for(int i = 0; i <= 5; i++){
5         scanf("%d", &arr[i]);
6     }
7     return 0;
8 }
```

```
1 #include <stdio.h>
2 int main(){
3     int *p;
4     *p = 10;
5     printf("%d\n", *p);
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 int main(){
3     char str[5];
4     scanf("%s", str); // input: Hello
5     printf("%s\n", str);
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 int main(){
3     int x = 5;
4     if (x = 0){
5         printf("x is zero\n");
6     }
7     return 0;
8 }
```

```
1 #include <stdio.h>
2 int* getNumber(){
```

```
3  int a = 42;
4  return &a;
5  }
6  int main(){
7  int *p = getNumber();
8  printf("%d\n", *p);
9  return 0;
10 }
```

```
1  #include <stdio.h>
2  void printSize(int arr[]){
3  printf("%zu\n", sizeof(arr));
4  }
5  int main(){
6  int arr[10];
7  printSize(arr);
8  return 0;
9  }
```

```
1  #include <stdio.h>
2  int main(){
3  int a = 1;
4  int b = a++ + a++;
5  printf("%d %d\n", a, b);
6  return 0;
7  }
```

答案

以上八个题目（以左栏第一个为第一题）分别错在：

1. `n` 未初始化就使用。应先读入 `n`，再定义数组。另，VLA 不是 C 标准的一部分，建议使用动态内存分配 `malloc`。
2. 数组越界。应改为 `i < 5`。
3. 指针未初始化就使用。
4. 数组长度不足以存储输入的字符串，是忘记 `\0` 导致的。应改为 `char str[6];`。
5. 误用赋值运算符。应改为 `if (x == 0)`。
6. 返回局部变量地址，导致悬空指针。应改为静态变量或动态分配内存。
7. 数组作为函数参数时退化为指针，`sizeof` 结果是指针大小。应传入数组大小作为额外参数。
8. 未定义行为，因为 `a++` 的副作用未定义顺序。应改为两行分别处理。

改正代码略，读者可自行完成。

第九章 从 C 到 C++

有的同学可能会问：那为什么又来了个 C++ 呢？C 不是已经很好了吗？为什么还要搞个 C++ 出来呢？

理由很简单：C 语言的确简单高效，但“太弱”，缺乏现代编程语言的特性，如 OOP（面向对象编程）、泛型；其标准库也极小，很多东西都得手搓，在日常编程中这是非常痛苦的。

因此，C++ 应运而生。C++ 在保留 C 语言高效和灵活的同时，引入了许多现代编程语言的特性，如类和对象、继承、多态、模板等，从而使得程序设计更加模块化、可维护和可扩展。此外，C++ 还提供了一个强大的标准库（STL），包括容器、算法和迭代器等，大大简化了日常编程任务。

C++ 广泛应用于多种需要兼顾性能和抽象的领域，如系统软件、游戏开发、嵌入式系统和高性能计算等，著名的 Chrome、Adobe 全家桶、MS Office、Visual Studio 都是 C++ 写的，古老的 DirectX 游戏引擎¹、现代的虚幻引擎²等也都要求用 C++ 开发；而 Unity 引擎虽然允许用户使用 C# 辅助开发，但其底层核心（乃至 C# 的 .NET 运行时）也是用 C++ 完成；Python 里相当多的高性能库（如 PyTorch、TensorFlow、JAX³等）也是用 C++ 写的。C 和 C++ 在事实上构成了当今高性能计算和软件开发的基石，在编程语言排名中常年稳居第三名和第二名，仅次于 Python⁴。

注意

如果你仅是为了应付课程或打算法竞赛，那不必完全遵从本文中涉及到的诸如代码风格等建议，只要能快速跑通就行；但如果你想真正学好 C++，建议你认真阅读并遵从本文中的建议。

一种代码有一种代码的写法，一次性代码（如考试和竞赛）和长期维护的代码（如工程项目）是两码事。前者追求速度和简洁，后者追求可读性和可维护性。本文所述的代码风格和建议，都是面向后者的。笔者个人也比较建议同学们从一开始就养成良好的代码习惯，毕竟未来你写的代码很可能会被别人阅读和维护。

所以说，虽然笔者个人非常厌恶和反对在任何代码中使用诸如 `#include <bits/c++.h>` 和 `using namespace std;` 之类的东西，以及大量使用全局变量、开巨大的 C 风格静态数组、滥用宏定义、用无意义的函数和变量名、一点注释不写、通篇魔法数字等行为，但如果你仅是为了应付考试或竞赛，那么使用这些东西也无可厚非，毕竟在考试的时候这些东西确实能帮你节省不少时间——但依然建议避免这种不好的实践。

¹该游戏引擎最著名的作品莫过于《红色警戒 2》了。

²这个写出来的东西就太多了，目前大多数 3A 大作都是用这个引擎写的。

³JAX 是 Numpy 的 GPU 加速版本，Google 出品。

⁴Python 最通行的解释器 CPython 甚至也是 C 写的！

但如果要是在大作业或工程项目中这么乱写，那就太不负责任了，被解雇了也是活该。

本章所学基本聚焦于 C++14 及其之前的标准，后续章节会介绍更新更好的新特性。当然，本节中为了代码的简洁起见，使用了一些诸如 `using namespace std;` 之类的写法，但在工程代码中应当避免使用这些写法；关于这个东西是什么、为什么不建议使用、以及如何避免使用，后续章节会详细介绍。

9.1 从 C 到 C++ 的区别

C++ 的基本语法和 C 语言几乎完全一致，因此如果你已经掌握了 C 语言，那么学习 C++ 将会非常容易。

9.1.1 第一个 C++ 程序

下面是一个简单的 C++ 程序，它输出 “Hello, World!” 到屏幕上：

```
1 #include <iostream> // 引入输入输出流库
2
3 int main() {
4     std::cout << "Hello, World!" << std::endl; // 输出 Hello, World!
5     return 0; // 返回 0 表示程序成功结束
6 }
```

我们发现，这个 HelloWorld 和 C 语言的 HelloWorld 长得很像，但确实存在一些区别：

- 引入的头文件有区别；
- 输出语句有区别。

在 C++ 中，基本的语法结构和 C 语言类似，包括变量声明、数据类型、控制结构（如条件语句和循环语句）、函数定义等，统统一致。可以说，C 怎么写，C++ 就怎么写，完全没有区别。C 语言的标准库在 C++ 中也有其移植版本，一般是从 `xx.h` 变成了 `cxx`，例如 C 语言的 `stdio.h` 在 C++ 中变成了 `cstdio`，但函数和用法几乎完全一致。

区别在于：C++ 自带 `bool` 类型，不需要 `#include <stdbool.h>`；另，结构体和联合体的定义上有所不同；第三，C++ 的字符串推荐用 `std::string`，而不是 C 风格的字符串；最后，C++ 推荐使用流对象来输入输出。

接下来会把 C++ 的新特性一一列举。对于面向对象、泛型和 STL 则会在后续章节中详细介绍。

9.1.2 花括号初始化

在 C++ 中，推荐使用花括号初始化（brace initialization）来初始化变量。这种初始化方式可以防止窄化转换（narrowing conversion），从而提高代码的安全性。例如：

```
1 int x = 10;           // 传统初始化（赋值或者拷贝风格）
2 int y(20);           // 传统初始化（构造函数风格）
3 int z{30};           // 花括号初始化
4 int a{3.14};         // 保证一定不会发生窄化转换，编译错误
```

花括号初始化的好处在于，它可以防止隐式类型转换，从而避免一些潜在的错误。但是，由于历史原因，实际上大家依然是更喜欢使用传统初始化方式；只有在很现代的 C++ 代码中，才会大量使用花括号初始化。

9.1.3 C++ 的输入输出及其格式化

输入输出流 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：

```
1 #include <iostream>
2
3 int main() {
4     int age;
5     std::cout << " 请输入你的年龄: "; // 输出提示信息
6     std::cin >> age; // 从标准输入读取数据
7     std::cout << " 你输入的年龄是: " << age << std::endl; // 输出读取到的数据
8     return 0;
9 }
```

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。

`std::endl` 是一个特殊的操纵符（manipulator），它的作用是输出一个换行符并刷新输出缓冲区，等价于 `'\n' << std::flush`。

所谓“缓冲区”指的是输出流在输出数据时会先将数据存储在一个临时的内存区域（缓冲区）中，等到缓冲区满了或者遇到换行符时才会将数据真正输出到屏幕上。使用 `std::endl` 可以确保输出立即显示在屏幕上，而不是被缓冲起来等待下一次输出操作，这对于调试和交互式程序非常有用。

但是需要注意的是，过度使用 `std::endl` 可能会导致性能问题，因为每次使用它都会刷新输出缓冲区，这可能会增加程序的运行时间。对于需要频繁输出的情况，建议使用 `'\n'` 来代替 `std::endl`，以避免不必要的性能开销。

在此类输出中，如涉及运算，**必须使用括号**，否则会出现优先级错误的问题。例如：

```
1 std::cout << " 两数之和为: " << (a + b) << std::endl; // 正确
2 std::cout << " 两数之和为: " << a + b << std::endl;   // UB
```

C 风格的输入输出 当然，使用 C 风格的输入输出函数 `printf` 和 `scanf` 也是可以的。例如：

```
1 #include <cstdio> // C 风格的输入输出库
2
3 int main() {
4     int age;
5     std::printf(" 请输入你的年龄: "); // 输出提示信息
6     std::scanf("%d", &age); // 从标准输入读取数据
7     std::printf(" 你输入的年龄是: %d\n", age); // 输出读取到的数据
8     return 0;
9 }
```

`<cstdio>` 是 C++ 的 C 风格输入输出库，与 C 语言的 `<stdio.h>` 头文件内容完全一致，只不过使用了 C++ 的命名空间（`std`）。不过大多数实现也允许不使用 `std::` 前缀，也允许使用 `<stdio.h>` 头文件，但我们不建议使用 C 的头文件。

C 风格的 `std::printf` 和 `std::scanf` 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。我们建议在 C++ 工程上使用更安全的 `std::cin` 和 `std::cout`。

我们在写代码的时候**一定不要一句 C 一句 C++，或者说不要一句 `printf` 一句 `cout`（反过来也不行）**，这样会导致缓冲区冲突，从而引发一些莫名其妙的问题。要么全用 C 的输入输出，要么全用 C++ 的输入输出。

说明

实际上，`std::cin` 和 `std::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 `std::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(std::cin>>n)`）来判断输入是否成功。

流运算符也不是 `>>` 和 `<<` 的原本样子。它们原本是右移和左移运算符：例如 `a<<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`）。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>`。

格式化输出 有时候，我们需要对输入输出进行一些格式化操作，例如设置小数点位数、对齐方式等。除了使用 `printf` 的格式化字符串外，C++ 提供了一些操纵符（`manipulator`）来实现这些功能。

- `std::setw(n)`：设置输出宽度为 `n` 个字符。
- `std::setprecision(n)`：设置小数点位数为 `n` 位。
- `std::fixed`：固定小数位数输出浮点数。

- `std::scientific`：使用科学计数法输出浮点数。
- `std::left`：左对齐输出。
- `std::right`：右对齐输出。

上述不少操纵符需要引用头文件 `<iomanip>`。例如，我们可以使用这些操纵符来格式化输出一个表格：

```

1 #include <iostream>
2 #include <iomanip> // 引入操纵符库
3 int main() {
4     std::cout << std::left << std::setw(10) << "Name" << std::setw(5) << "Age"
5     ↪ << std::setw(10) << "GPA" << std::endl;
6     std::cout << std::left << std::setw(10) << "Alice" << std::setw(5) << 20 <<
7     ↪ std::setw(10) << std::fixed << std::setprecision(2) << 3.5 << std::endl;
8     std::cout << std::left << std::setw(10) << "Bob" << std::setw(5) << 22 <<
9     ↪ std::setw(10) << std::fixed << std::setprecision(2) << 3.8 << std::endl;
10    return 0;
11 }
```

当然这个还是太冗繁了。建议使用更现代的方式，例如 C++20 引入的 `std::format` 函数（需要引用头文件 `<format>`），这个东西在之后的章节中有详细介绍。

格式化输入 对于输入，则复杂得多。我们推荐同学们使用更安全的 C++ 风格输入输出方法，也就是 `cin`、`cout`、`getline` 等。

对于确定数量的干净⁵输入，可以直接使用 `cin`：

```

1 int a, b, c;
2 // 假设输入格式为: 1 2 3
3 cin >> a >> b >> c; // 读入三个整数
```

对于不确定数量的干净输入，可以使用循环配合 `cin`：

```

1 int n;
2 while (cin >> n) {
3     // 处理输入的 n
4 }
```

上述代码会一直读取输入，直到遇到文件结束符（EOF）或者输入错误为止。其能工作的原因是 `cin` 在读取失败时会返回流对象的一个“失败”状态，该失败状态在布尔上下文中被解释为 `false`，从而终止循环。

如果遇到脏输入，则情况变得复杂许多。常见的脏输入包括逗号分割的数字、带有多余空格的字符串等。对于这些情况，推荐使用 `getline` 配合字符串流（`stringstream`）来处理。下文演示了这种方式，并将逗号分割的整数字符串转换为整数数组：

⁵这里的干净指的是简单的空格分割或换行符分割，没有诸如逗号等其他符号。与之相对应的脏数据则是指包含了各种符号、格式不统一等复杂情况的数据。

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 #include <vector>
5 using namespace std;
6
7 int main(){
8     string line;
9     string tmp;
10    vector<int> results;
11
12    // 1. 读一整行
13    getline(cin, line);
14
15    // 2. 创建字符串流
16    stringstream ss(line);
17
18    // 3. 按逗号分割并处理
19    while (getline(ss, tmp, ',')) {
20        results.push_back(stoi(tmp)); // 转换为整数并存储
21    }
22    // 如果转换为 double, 可以使用 stod
23 }
```

工程上，脏数据非常常见，因此掌握这些输入方式是非常有必要的。

注意

流对象处理输入输出的本质依然是操作缓冲区。因此，有一部分 OIer 认为他们自己维护缓冲区的输入方式更快、更好。诚然，`getline` 等方法处理缓冲区的性能大概比手动操作缓冲区低了约 5 到 10%，但是实际上我并不推荐手动维护缓冲区，尤其是在工程上。原因有三：

- 不安全。这是最大的一个弊病。手动维护缓冲区和不穿衣服在街上乱晃是一个道理，属于是把自己的安全完全交给了用户的善意。恶意用户攻击你的缓冲区将变得轻而易举。
- 不易懂。仅代码量一项，手动维护缓冲区就比使用流对象多出不少代码量，且难以阅读。这违背了工程代码的可读性原则。
- 难维护。我们在做题的时候，不少题目虽然算法简单但是边界条件复杂（例如日历问题），做这些题目的时候应付边界条件的时间几乎可以占到做题时间的一半。对于工程而言，手动维护缓冲区需要自己处理各种边界情况，估计也没有几个团队会有这个时间和精力去维护这些边界条件。

综上所述，虽然手动维护缓冲区在某些情况下可能会有一些性能优势，但是这种优势并不值得我们为之付出安全性、可读性和可维护性的代价。这就是工程代码为了安全、可读、可维护而牺牲性能的一个典型例子；另一方面，可以看到竞赛思维和工程思维有显著的差异，不能够混为一谈。

提示

在 C++ 中，除 `std::cout`，还有 `std::cerr` 和 `std::clog` 两个流对象用于错误输出和日志输出。它们的用法和 `std::cin`、`std::cout` 类似，但有一些区别：

- `cout` 是行缓冲的，也就是缓冲到换行符或缓冲区满才输出，可以重定向到文件或其他设备（例如 `./program > file`），一般用于正常输出，关联的设备是标准输出设备（`stdout`）。
- `cerr` 是无缓冲的，也就是每次输出都会立即显示，不会缓冲，常用作错误信息的输出。其关联的设备也不是标准输出设备，而是标准错误设备（`stderr`）。重定向方式例如 `./program 2> error.log`，

其中2>表示重定向标准错误输出。

- `clog`是全缓冲的，也就是缓冲区满才输出，可以重定向到文件或其他设备，一般用于日志信息的输出。虽然该流对象也关联标准错误设备（`stderr`），但其行为更像`cout`。重定向方式同`cerr`。

9.1.4 利用文件流进行文件读写

在 C 语言的章节中，我们已经知道，可以使用 `FILE` 结构体的指针来操作文件。但是这种操作有一定弊病：需要手动管理文件指针，容易出错。虽然智能指针能够部分解决手动管理文件指针的麻烦，但 C++ 提供了更好的文件操作方式：文件流（file stream）。

文件流和输入输出流类似，也是一个流对象。文件流有两个：`std::ifstream` 用于从文件读取数据，`std::ofstream` 用于向文件写入数据，分别对应标准输入输出的 `std::cin` 和 `std::cout`。这两个流对象提供了和标准输入输出流类似的操作方法，例如使用 `>>` 和 `<<` 运算符进行读写操作。

下面是一个简单的例子，演示如何使用文件流读取和写入文件：

```
1 #include <iostream>
2 #include <fstream> // 引入文件流库
3
4 int main(){
5     std::ifstream infile("input.txt"); // 打开输入文件
6     std::ofstream outfile("output.txt"); // 打开输出文件，默认是覆盖模式
7     int a, b;
8     infile >> a >> b; // 从输入文件读取数据
9     outfile << "Sum: " << (a + b) << std::endl; // 向输出文件写入数据
10    infile.close(); // 关闭输入文件
11    outfile.close(); // 关闭输出文件
12    return 0;
13 }
```

在上述代码中，“读取”和“写入”两个操作被分成两个流对象而不是一个文件。这是因为 C++ 的 OOP 思想，每一个对象仅负责一个职责。另，在这里也可以看出，文件流应当自行定义，而不是已经定义好的 `std::cin` 和 `std::cout`。

`ofstream` 默认是覆盖模式打开文件的，如果想要以追加模式打开文件，可以使用以下方式：

```
1 std::ofstream outfile("output.txt", std::ios::app); // 以追加模式打开输出文件
```

当然，也可以使用一个统一的文件流对象来同时进行读写操作，这时需要使用 `std::fstream` 类，但这时候需要指定读写模式：

```
1 std::fstream file("data.txt", std::ios::in | std::ios::out); // 以读写模式打开文件
```

我们有这些读写模式可以使用：

- `std::ios::in`：以读模式打开文件。
- `std::ios::out`：以写模式打开文件。

- `std::ios::app`：以追加模式打开文件，写入的数据会追加到文件末尾。
- `std::ios::ate`：打开文件后将文件指针移动到文件末尾，可以用于读写操作。
- `std::ios::trunc`：如果文件已存在，则在打开时将其内容截断为 0 长度（实际上就是覆盖模式）。
- `std::ios::binary`：以二进制模式打开文件，而不是文本模式。这一手段打开的文件能够避免字符转换。

这些读写模式实际上都是独热编码，也正因此，我们可以使用按位或运算符（|）来组合多个模式。

9.1.5 文件的其他操作

在 C 语言中，我们能够使用 `remove` 等函数来对文件和目录进行操作。在 C++ 中也有类似的功能，位于 `<filesystem>` 头文件中。这个头文件在 C++17 中引入，提供了一些用于文件和目录操作的类和函数。下面是一些常用的文件操作函数：

- `std::filesystem::remove(path)`：删除指定路径的文件或目录。
- `std::filesystem::rename(old_path, new_path)`：重命名文件或目录。
- `std::filesystem::copy(source, destination)`：复制文件或目录。
- `std::filesystem::create_directory(path)`：创建一个新目录。
- `std::filesystem::exists(path)`：检查指定路径是否存在。
- `std::filesystem::file_size(path)`：获取指定文件的大小。
- `std::filesystem::current_path()`：获取当前工作目录的路径。
- `std::filesystem::absolute(path)`：获取指定路径的绝对路径。
- `std::filesystem::directory_iterator(path)`：迭代指定目录下的文件和子目录。
- `std::filesystem::space(path)`：获取指定路径所在文件系统的空间信息。
- `std::filesystem::last_write_time(path)`：获取指定文件的最后修改时间。
- `std::filesystem::permissions(path, perms)`：设置指定文件或目录的权限。
- `std::filesystem::remove_all(path)`：递归删除指定路径的文件或目录及其内容。

需要注意的是，`path` 是一个类型，可以是字符串类型（如 `std::string`）或者 `std::filesystem::path` 类型。使用这些函数时，需要包含头文件 `<filesystem>`，并且在编译时需要链接文件系统库（例如使用 `-lstdc++fs` 选项）。

9.1.6 常变量、常量和它们的关系

常变量（也叫不可变变量、只读变量、运行时常量）、常量（也叫编译期常量）往往笼统地称为常量。它们一旦确定就不会在程序运行时改变，任何试图对它们进行运行时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。它们的名字通常使用大写字母来表示，以便于和变量区分。

声明常变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：

```
1 const int MAX_VALUE = 100;  
2 const int P = a + b; // 这里的 a 和 b 可以是变量  
3 // 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。

如果常变量的值**必须在编译时**确定，可以使用常量。常量的值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。常量的声明方法和常变量类似，只是把 `const` 换成 `constexpr`。

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，但是要求用于计算的东西也必须是常量、字面值（直接写出来的值）、`constexpr` 函数或者立即函数⁶。下文是常量的几个例子。

```
1 constexpr double E = 2.71828;  
2 constexpr double PI = 3.14159;  
3 constexpr double EPI = E * PI;
```

在现代 C++ 中，`const` 常变量不依靠运行时初始化来确定其值（例如 `const int b = 1;`），其表现就和 `constexpr` 常量一样了。因此，在大多数时候，我们也可以把 `const` 常变量当作 `constexpr` 常量来使用。但如希望严谨表达意图，仍建议使用 `constexpr` 来声明常量。

提示

还是不懂？可以通过以下例子理解一下变量、运行时常量、编译期常量的区别：

1 **int** sqr(**int** x) { **return** x * x; } // 普通函数
2 **const int** sqr_c(**const int** x) { **return** x * x; } // const 函数
3 **constexpr int** sqr_ce(**const int** x) { **return** x * x; } // constexpr 函数
4 **constexpr int** sqr_cv(**const int** x) { **return** x * x; } // constexpr 函数

那么对于以下声明，编译器的表现如下表所示。其中，编译失败的情形用红色标出；用 `const` 声明的运行时常量表现为编译期常量的特殊情形则使用蓝色标出。

表 9.1: 变量/常量声明与编译器表现

声明	编译器表现	理由
<code>int a0 = 5;</code>	变量	显然
<code>const int a1 = 5;</code>	常量	不依赖运行时初始化
<code>constexpr int a2 = 5;</code>	常量	字面值
<code>const int a3 = a0;</code>	常变量	依赖运行时值 a0
<code>constexpr int a4 = a0;</code>	编译失败	严格常量不能用运行时值初始化
<code>int a5 = sqr(1);</code>	变量	显然
<code>const int a6 = sqr(1);</code>	常变量	依赖运行时函数
<code>constexpr int a7 = sqr(1);</code>	编译失败	严格常量不能用运行时函数初始化

⁶立即函数指的是声明为 `constexpr` 的函数，在 C++20 中被引入，这样的函数**只能在编译时期调用**

int a8 = sqr_c(1);	变量	显然
const int a8 = sqr_c(1);	常变量	依赖运行时函数
constexpr int a9 = sqr_c(1);	编译失败	严格常量不能用运行时函数初始化
int a10 = sqr_ce(1);	变量	显然
const int a11 = sqr_ce(1);	常量	该函数接受常量则在编译期初始化
const int a12 = sqr_ce(a0);	常变量	依赖运行时值 a0
constexpr int a13 = sqr_ce(1);	常量	显然
constexpr int a14 = sqr_ce(a0);	编译失败	严格常量不能用运行时值初始化
int a15 = sqr_cv(1);	编译失败	立即函数不可以在运行时调用
const int a16 = sqr_cv(1);	常量	显然
const int a17 = sqr_cv(a0);	编译失败	立即函数不可以在运行时调用
constexpr int a18 = sqr_cv(1);	常量	显然
constexpr int a19 = sqr_cv(a0);	编译失败	立即函数不可以在运行时调用

提示

用宏定义的常量和用 `const` 或 `constexpr` 定义的常量有一些区别。宏定义的常量没有类型，因此在使用时需要注意类型转换的问题；而 `const` 或 `constexpr` 定义的常量有类型，可以更好地进行类型检查和转换。此外，宏定义的常量在预处理阶段进行替换，因此可能会导致一些意想不到的问题，例如宏展开时的优先级问题等。而 `const` 或 `constexpr` 定义的常量在编译阶段进行处理，更加安全可靠。

9.1.7 数组

C++ 虽然支持 C 风格的数组 `int arr[10];`，但是不推荐使用这种方式来声明数组。

在 C++ 中，数组一般使用 `std::array` 和 `std::vector` 来声明。它们分别表示静态数组和动态数组。

静态数组 静态数组的大小在编译时确定，不能动态改变。它的基本格式如下：

```
1 #include <array> // 引入数组库
2 array<int, 10> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // 声明一个包含 10 个整数的数
   ↳ 组
3 cout << arr[0] << endl; // 访问数组的第一个元素
```

这东西的实现和 C 风格数组相似，但一个重要的区别是：它是一个复杂类，而不是基本类型，有很多方法能够操作；另一个重要的区别是，在作为函数参数传递时，它不会退化为指针，从而避免了信息丢失的问题。

也就是说：

```
1 void foo(array<int, 10> arr) {
2     cout << arr.size() << endl; // 可以获取数组的大小
3 }
4
5 void bar(int arr[10]) {
```

```
6     cout << sizeof(arr) / sizeof(arr[0]) << endl; // 这里会输出错误的结果, 因为 arr
    ↳ 退化为指针, sizeof(arr) 得到的是指针大小而非数组大小
7 }
```

动态数组 动态数组的大小可以在运行时确定, 可以动态改变。它的基本格式如下:

```
1 #include <vector> // 引入向量库
2 vector<int> vec; // 声明一个空的动态数组
3 vec.push_back(1); // 向数组末尾添加一个元素 1
4 vec.push_back(2); // 向数组末尾添加一个元素 2
5 cout << vec[0] << endl; // 访问数组的第一个元素
6 cout << vec.size() << endl; // 获取数组的大小
7 vec.pop_back(); // 删除数组末尾的元素
8 vec.remove(0); // 删除数组中值为 0 的元素
```

动态数组的实现和静态数组类似, 也是一个复杂类, 有很多方法能够操作。它的大小可以动态改变, 因此非常灵活, 但性能上有较大的损失, 在大型数组上比 `array` 慢一倍; 但数组如果太大又不得不使用动态数组, 否则栈空间不够用。

使用 `vector` 能够避免和 C 一样用 `malloc` 和 `free` 来直接操作内存, 省心省力。

动态数组的扩容机制是一个值得研究的话题。当动态数组的容量不足以容纳新添加的元素时, 通常会进行扩容操作。扩容的过程一般包括以下几个步骤:

1. 分配一块更大的内存空间, 通常是当前容量的两倍。
2. 将原有元素从旧内存空间复制到新内存空间。
3. 释放旧的内存空间。
4. 更新动态数组的内部指针和容量信息。

这种扩容策略能够在保证动态数组性能的同时, 减少内存分配的次数, 从而提高整体效率。不过需要注意的是, 扩容操作可能会导致内存碎片化, 因此在频繁添加和删除元素的场景下, 可能需要考虑其他数据结构 (例如链表) 来替代动态数组。

堆和栈 在 C++ 中, 变量的内存分配主要有两种方式: 栈 (stack) 和堆 (heap)。栈是一种后进先出 (LIFO) 的数据结构, 适用于存储局部变量和函数调用信息; 堆则是一种动态分配的内存区域, 适用于存储需要在运行时动态创建和销毁的对象。

栈内存的分配和释放由编译器自动管理, 当函数调用时, 局部变量会被压入栈中; 当函数返回时, 这些变量会被自动弹出栈。栈内存的分配速度较快, 但空间有限, 通常只有几 MB 到几十 MB。

堆内存的分配和释放需要程序员手动管理, 使用 `new` 关键字分配内存, 使用 `delete` 关键字释放内存。堆内存的空间较大, 可以达到几 GB, 但分配和释放的速度较慢, 且容易导致内存泄漏和碎片化问题。

9.1.8 字符串

C++ 风格的字符串类型是 `std::string`, 它可以存储一串字符。字符串的基本格式如下:

```
1 #include <string>    // 引入字符串库
2 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。

9.1.9 结构体、联合体

C++ 的结构体和联合体与 C 中的略有区别。

例如，我们可以声明一个表示学生的结构体：

```
1 struct Student {
2     string name;    // 学生姓名
3     int age;        // 学生年龄
4     double gpa;     // 学生绩点
5 };
6
7 Student student1;  // 声明一个学生变量
8 student1.name = "Alice"; // 设置学生姓名
9 student1.age = 20;  // 设置学生年龄
10 student1.gpa = 3.5; // 设置学生绩点
11 cout << "Name: " << student1.name << ", Age: "
12     << 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。可以看到，不需要再像 C 语言一样，定义起来那么复杂。联合体也类似，这里不再赘述了。

C++ 结构体、联合体的内存结构和 C 语言中的是一样的，都是按照成员的顺序依次存储在内存中的。

另一个非常重要的不同是：C++ 结构体和联合体可以包含方法（函数），而 C 语言中的结构体和联合体只能包含数据成员。这使得 C++ 的结构体和联合体更加强大和灵活，可以实现朴素的面向对象编程（OOP）特性，如：

```
1 struct Student {
2     string name;    // 学生姓名
3     int age;        // 学生年龄
4     double gpa;     // 学生绩点
5     void printInfo() { // 成员方法，打印学生信息
6         cout << "Name: " << name << ", Age: "
```

```

7         << age << ", GPA: " << gpa << endl;
8     }
9 };
10
11 Student student1; // 声明一个学生变量
12 student1.name = "Alice"; // 设置学生姓名
13 student1.age = 20; // 设置学生年龄
14 student1.gpa = 3.5; // 设置学生绩点
15 student1.printInfo(); // 调用成员方法, 打印学生信息

```

9.1.10 枚举

枚举是一个可以存储一组命名常量的变量。

枚举有两种类型，一种是传统无作用域枚举 `enum`，另一种是 C++11 引入的有作用域枚举 `enum class`（也叫强枚举）。它们的区别在于，传统无作用域枚举的常量可以直接访问，枚举名和枚举值都泄漏到所在的作用域；而有作用域枚举的常量需要通过枚举名来访问。

传统无作用域枚举的基本格式如下：

```

1     enum Color { RED, GREEN=5, BLUE };

```

一般情况下，枚举的底层类型由编译器自选，只要能够容纳所有的值就行了，一般是 `int`。常量从 0 开始依次递增，例如上面的 `RED` 的值为 0。也可以设定枚举的值，上文中我们将 `GREEN` 的值设定为 5，那么 `BLUE` 的值就是 6。调用这种枚举非常简单：

```

1 Color color = RED; // 正统调用, color 的值为 0
2 int n = (int)RED; // 正统调用, n 的值为 0
3 int n = RED; // 不报错, n 的值为 0
4 Color c = 7; // 不报错, 但是 c 的值不是 RED、GREEN、BLUE 中的任何一个, 属于有效但未命名的
   ↪ 值

```

可以看出，这种枚举没有类型安全性，也没有作用域隔离。

有作用域枚举的基本格式如下：

```

1     enum class Color : std::uint8_t { RED, GREEN=5, BLUE };

```

上述强枚举的枚举名被限定在作用域内，因此只能通过类似 `Color::RED` 来访问。强枚举的底层类型可以显式指定，例如上面的 `std::uint8_t`。如果不指定，默认是 `int`。调用这种枚举只能使用上述的正统调用：

```

1 Color color = Color::RED; // 正确, color 的值为 Color::RED
2 int n = Color::RED; // 报错, 不能将 Color 类型赋值给 int 类型
3 Color c = 6; // 报错, 不能将 int 类型赋值给 Color 类型
4 int n = static_cast<int>(Color::RED); // 正确, n 的值为 0

```

可以看出，这种枚举有类型安全性，也有作用域隔离。

枚举作为一个数据类型很笨，不仅没有任何方法，也不能进行运算，唯一的作用是定义一组常量，便于阅读；在 `switch` 语句中的使用较为多见。而剩下的很多方法，都不得不手动定义。

例如下列代码中，我们定义了一个枚举的遍历方法：

```
1 enum class Color : std::uint8_t {
2     FIRST=RED,
3     RED=0,
4     GREEN=1,
5     BLUE=2,
6     LAST=BLUE
7 };
8
9 for (Color c = Color::FIRST; c <= Color::LAST; c =
   ↳ static_cast<Color>(static_cast<int>(c) + 1)) {
10     // 遍历 Color 枚举的所有值
11 }
```

这段代码中，我们定义了一个 `Color` 枚举，并且手动定义了 `FIRST` 和 `LAST` 两个常量，分别表示枚举的第一个值和最后一个值。

9.1.11 智能指针（穿了衣服版）

我们已经知道 C 风格的指针是什么了。C++ 中虽然也能和 C 一样使用“裸指针”，但是不推荐这么做。C++ 引入了智能指针的概念，来帮助我们更好地管理内存，不需要再手动 `malloc/free` 了。

智能指针有三个⁷主要类型。这三个主要类型都定义在 `<memory>` 头文件中，分别是：

- `std::unique_ptr` : 表示独占所有权的强引用智能指针, 一个对象只能有一个 `unique_ptr` 指向它。当这个 `unique_ptr` 被销毁时，所指向的对象也会被自动释放。
- `std::shared_ptr` : 表示共享所有权的强引用智能指针, 一个对象可以有多个 `shared_ptr` 指向它。对象会在最后一个指向它的 `shared_ptr` 被销毁时自动释放。
- `std::weak_ptr` : 表示弱引用的智能指针, 不拥有对象的所有权。它通常与 `shared_ptr` 一起使用，用于解决循环引用的问题。

说明

上述文字中，提到了一个新的名词：**所有权**。这是编程中的一个非常重要的概念。

所有权指的是：若某个对象 A 拥有另一个对象 B 的所有权，那么 A 就负责 B 的生命周期管理，包括 B 的创建、使用和销毁。换句话说，A 有权决定 B 什么时候被创建，什么时候被销毁。例如在传统的 C 中，以下代码是常见的：

1 void foo() {
2 int a = 42; // 创建一个整数对象，a 拥有它的所有权
3 int* ptr = new int(42); // 创建一个整数对象，ptr 拥有它的所有权

⁷实际上还有第四种 `std::auto_ptr`，但是因为这个类型存在一些设计缺陷，已经在 C++11 中被弃用，因此这里不做介绍。

```

4     delete ptr; // 释放 ptr 指向的整数对象
5 }

```

上述代码中，函数 `foo` 拥有指针 `ptr` 指向的整数对象的所有权，因此它负责这个对象的创建和销毁；而在该函数结束时，也要负责销毁所有的对象，也就是释放内存。如果上述代码中没有 `delete ptr`；这一行，那么就会导致内存泄漏，因为这个整数对象的内存没有被释放（但上述 `int a = 42`；这一行不会导致内存泄漏，因为它是栈上的对象，会在函数结束时自动销毁）。这个实际上就是 C 的一个大问题：如果程序员忘记释放内存，那就会导致程序内存泄漏。

而有了智能指针，这个问题就迎刃而解了，见下文。

例如：

```

1 void foo(){
2     auto ptr = std::make_unique<int>(42); // 创建一个 unique_ptr，指向一个整数 42
3     std::cout << *ptr << std::endl; // 输出 42
4 } // ptr 超出作用域，所指向的整数自动释放
5
6 void bar(){
7     auto ptr1 = std::make_shared<int>(42); // 创建一个 shared_ptr，指向一个整数 42
8     {
9         auto ptr2 = ptr1; // 共享所有权
10        std::cout << *ptr2 << std::endl; // 输出 42
11    } // ptr2 超出作用域，但整数不会释放，因为 ptr1 仍然指向它
12    std::cout << *ptr1 << std::endl; // 输出 42
13 } // ptr1 超出作用域，所指向的整数自动释放

```

除非直接定义，否则不建议直接使用智能指针的构造函数来创建智能指针对象，而是推荐使用 `make_unique` 和 `make_shared` 这两个工厂函数来创建智能指针对象。例如：

```

1 auto ptr1 = std::make_unique<int>(42); // 创建一个 unique_ptr，指向一个整数 42
2 auto ptr2 = std::make_shared<int>(42); // 创建一个 shared_ptr，指向一个整数 42
3
4 struct Node {
5     int value;
6     std::shared_ptr<Node> next; // 直接定义的 shared_ptr，也是好的用法
7     Node(int val) : value(val), next(nullptr) {}
8 };

```

`make_unique` 和 `make_shared` 是创建智能指针的推荐方式。我们要创建什么类型、指向什么对象，就在尖括号和圆括号中指定即可。这样可以避免许多的潜在错误，例如内存泄漏、悬空指针等。

而 `weak_ptr` 的使用则更为复杂一些，通常用于解决循环引用的问题。例如：

```

1 #include <iostream>
2 #include <memory>
3 #include <string>
4 using namespace std;
5 struct Node {
6     string name;
7     shared_ptr<Node> next; // 强引用，可能导致循环引用

```

```
8     weak_ptr<Node> prev;    // 弱引用，避免循环引用
9
10    Node(const string& n) : name(n), next(nullptr), prev() {}
11 };
12 class List{
13 public:
14     shared_ptr<Node> head;
15     shared_ptr<Node> tail;
16     ... // 其他方法
17 }
```

说明

那两个 `shared_ptr` 为什么会 导致循环引用？这是因为智能指针 对内存进行 管理的方式 是通过引用计数来实现的。每有一个 `unique_ptr` 或者 `shared_ptr` 指向一个对象，这个对象的引用计数就会增加 1；每当一个 `unique_ptr` 或者 `shared_ptr` 被销毁或者重新指向其他对象时，引用计数就会减少 1。当引用计数变为 0 时，表示没有任何智能指针指向这个对象了，这时对象的内存就会被自动释放。而 `weak_ptr` 不会影响引用计数。

这就可以看出问题。如果我们写了这样的代码：

```
1 struct Node {
2     string name;
3     shared_ptr<Node> next; // 强引用，可能导致循环引用
4     shared_ptr<Node> prev; // 强引用，可能导致循环引用
5 }
```

那么如果我们创建了两个节点 A 和 B，并且让 A 的 `next` 指向 B，B 的 `prev` 指向 A，那么 A 和 B 就形成了一个循环引用。即使我们不再使用 A 和 B，它们的引用计数也不会变为 0，因为它们互相引用着对方。于是，A 和 B 的内存永远不会被自动释放，因而就会导致内存泄漏。

为了解决这个问题，我们可以使用 `weak_ptr` 来打破循环引用。例如，我们可以让 `prev` 成为一个 `weak_ptr`，这样就不会增加引用计数，从而避免循环引用的问题。

而为什么在上述 `List` 类中，`head` 和 `tail` 使用 `shared_ptr` 呢？这是因为 `head` 和 `tail` 是链表的入口和出口，它们需要拥有节点的所有权，以确保链表中的节点在链表存在期间不会被销毁。因此，使用 `shared_ptr` 是合适的选择。我们也可以模拟以下销毁整个链表的过程：

- 1. 在链表的生命周期内：链表头的引用计数是 1，链表尾节点的引用计数是 2（“链表尾”^a指向尾节点，而前一个节点也指向尾节点）。
- 2. 先销毁整个 `List`。这时候，链表头的引用计数会减少 1（变成 0），而尾节点的引用计数也会减少 1（变成 1）。
- 3. 由于链表头的引用计数变成 0，因此头节点会被自动释放。头节点的释放会导致它的 `next` 指针指向的下一个节点的引用计数减少 1，也就是第二个节点的引用计数变成 0。
- 4. 于是产生链式反应，整个链表的节点都会被依次释放，直到尾节点。

^a “链表尾”和“链表尾节点”不是一个东西。以一个长度为 10 的链表为例，链表尾节点是第 10 个节点，而链表尾是一个指向第 10 个节点的指针。

这是写了一个双向链表节点的例子。注意其中的 `next` 是强引用，而 `prev` 是弱引用。这样可以避免循环引用的问题，从而防止内存泄漏。这个指针在多线程编程中也很有用，可以安全地访问共享资源。当然，我们一般不去主动构造一个没有配合的弱引用，弱引用大多都是依附于共享强引用而存在的。

而数组中也不再建议使用指针来访问了，改为统一使用 STL 容器，例如 `std::vector`、`std::array` 的迭代器。于是，裸指针在 C++ 中基本光荣退休。至于迭代器是啥，我们会在后续章节中详细介绍。

基本上，我们可把使用哪种指针的决策流程总结如下：

- 不共享所有权，使用 `std::unique_ptr`。这是 99% 的情况。
- 共享所有权且肯定没有循环引用，使用 `std::shared_ptr`。
- 共享所有权且可能有循环引用，使用 `std::shared_ptr` 和 `std::weak_ptr`，后者用于打破循环引用。
- 数组？使用 `std::vector`、`std::array` 等 STL 容器，而非指针。

C++ 的智能指针极大地简化了内存管理，减少了内存泄漏和悬空指针等问题的发生。推荐大家在 C++ 编程中尽量使用智能指针，而不是裸指针。

9.2 C++ 独占的特性

上述内容中，就算是 `std::array` 和智能指针，实际上都能在 C 中找到影子，在 C 中也可以较为容易地实现这些功能。但是，C++ 中也有一些独占的特性，是 C 中很难或无法实现的。下面介绍几个重要的 C++ 独占特性。

9.2.1 命名空间

我们知道，一个软件还是程序，可能由很多人来完成。为了方便，每一个人都有可能定义自己的东西，例如功能（函数）、数据（变量）等。那么，如果两个人给自己不同的东西起了同样的名字怎么办？这时，电脑就无法区分它们了。

一个简单的方法是加强沟通，减少重名的可能性。但是，这样做并不现实。有的项目可能有数百人参与，沟通成本过高；有的项目是给下游使用的，这时又不可能沟通。这个问题非常棘手。

为了解决这个问题，C++ 引入了**命名空间**的概念。命名空间可以参照我们说过的虚拟环境概念来理解：每一个人都有自己的一个沙盒，在自己的沙盒里可以随便起名字，互不干扰。这样一来，即使两个人起了同样的名字，也不会冲突，因为它们属于不同的命名空间。

```
1  namespace Alice {
2      int value = 42; // 数据 (变量)
3      void show() {   // 功能 (方法)
4          std::cout << "Alice's value: " << value << std::endl;
5      }
6  }
7  namespace Bob {
8      int value = 100; // 数据 (变量)
9      void show() {    // 功能 (方法)
10         std::cout << "Bob's value: " << value << std::endl;
11     }
12 }
```

这样，两者并不冲突。

但是新的问题又来了：有时候，别人在他们的命名空间里写了一些东西，而这些东西又是我们想要的。为了方便起见，肯定不能写第二遍。那么，我们该怎么办呢？可以这样写：

```
1 Alice::show(); // 调用 Alice 命名空间中的 show 函数
2 Bob::show();   // 调用 Bob 命名空间中的 show 函数
```

于是困扰我们的重名问题就彻底解决了。

为了帮助我们更好的开发，C++ 提供了一些东西减少我们的重复劳动。这些东西被 C++ 放在了“标准”命名空间中，也就是 `std`。诸如 `cout`、`cin`、`endl` 等都在这个命名空间中。因此，我们在使用这些东西的时候，必须加上 `std::` 前缀，例如 `std::cout`、`std::cin`、`std::endl` 等。

为了方便起见，可以使用 `using namespace std;` 来引入整个 `std` 命名空间，这样在这个文件以及其下游文件中，就可以直接使用标准空间中的东西，而不需要加上 `std::` 前缀了。但是这样做也是有风险的：这会把整个标准命名空间都引进来，容易导致重名冲突等问题。

举例：假设你自己写了一个 `swap` 函数，然后在你的头文件中使用了 `using namespace std;`，那么当别人引入你的头文件时，标准命名空间中的 `std::swap` 函数也会被引入，从而导致重名冲突，最终引发编译错误。

命名空间的编译原理 命名空间符号在编译时是在符号前加上命名空间前缀来实现的。例如，`std::cout` 在编译时会被转换为类似 `_ZSt4cout` 这样的符号名（具体名称可能因编译器而异）。这样做的好处是可以避免不同命名空间中的符号冲突。

警告

严格禁止在工程头文件中使用 `using namespace std;`！这会污染全局的命名空间，从而导致重名冲突等问题。头文件是给别人用的，绝对不应污染别人的命名空间。

如果确实需要，我们有以下手段来解决污染命名空间问题：

- 每一次使用标准命名空间中的东西时，都加上 `std::` 前缀。

```
1 std::cout << "Hello, World!" << std::endl;
```

- 只引入需要的东西，例如：

```
1 using std::cout;
2 using std::endl;
```

这样就只引入了 `cout` 和 `endl`，而不会污染其他的東西。而一般人也不会去定义诸如 `cout` 和 `endl` 这样的名字，所以这样基本上可以认为是安全的。

注意

在工程上，源文件也不推荐使用 `using namespace std;`。但是这样做是可以容忍的，因为源文件是给自己用的，一般不至于污染命名空间，但是风险也是相当大的。对此，这需要大家自己权衡利弊了。如果项目周期非常短（例如做题），那么这么做没有毛病。但是如果是写工程这种长周期开发，则推荐老老实实用上面提到的两种方法来避免污染命名空间。

说明

为了简便，本书中大部分代码都使用了 `using namespace std;`，但是请大家务必牢记上述警告和注意事项。

9.2.2 引用

引用是 C++ 中的一个重要特性，其核心任务是让变量有第二个名字，但不管理生命周期。引用不能为空、不能重新指向、不拥有对象，但在函数参数传递、返回值等场景中极为有用。

左值引用

默认提到引用，指的就是左值引用。
引用的基本格式如下：

```
1 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。引用的作用是可以通
过别名来访问原变量。例如，我们可以声明一个整数的引用：

```
1 int a = 10; // 声明一个整数变量
2 int& ref = a; // 声明一个整数的引用
3 cout << "a: " << a << ", ref: " << ref << endl;
4 ref = 20; // 修改引用的值
5 cout << "a: " << a << ", ref: " << ref << endl;
```

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。
而在函数形参中，引用是较大形参传递的首选。

```
1 void draw(const Widget& w) {
2     // 使用 w 进行绘制操作
3 }
```

上述一股 Qt 味的代码中，我们使用了一个常量引用作为函数参数。这样做有两个好处：

- 避免了拷贝开销。如果 `Widget` 是一个较大的类，那么传递它的引用可以避免拷贝整个对象，从而提高性能。

- 保持了对象的不可变性。使用常量引用可以确保函数不会修改传入的对象，从而提高代码的安全性。

而使用非常量引用作为函数参数，则表示函数可能会修改传入的对象：

```
1 void update(Widget& w) {  
2     // 修改 w 的状态  
3 }
```

为什么不用智能指针？因为智能指针管理生命周期，而引用不管理生命周期。引用更轻量级，适合用于函数参数传递和返回值等场景。除非你要转移或重置指针本身，否则不需要使用 `unique_ptr`；除非你要延长对象寿命（如缓存、异步任务等），否则不需要使用 `shared_ptr`。

但引用也不是万能的：容器里就不要存引用了，这是值（具体对象）和多态基类（智能指针）的天下，引用请靠边站（不是对象，存不了）。

右值引用

右值引用是 C++11 引入的一种新特性，绑定的是“马上就要死”的临时变量，这一点和左值引用绑定的“长期活着”的变量不同。

一般，有自己名字的变量，要引用全都是左值引用；而没有自己名字的临时变量，则要用右值引用。另，`std::move` 可以把左值强制转换为右值引用，从而实现移动语义。那什么是“移动语义”呢？后面会讲。

举例：

```
1 int x = 1; // x 是一个左值, 1 是一个右值  
2 int& lref = x; // 左值引用, 绑定到左值 x  
3 int&& rref = 2; // 右值引用, 绑定到右值 2  
4 int&& rref2 = x // 错误, 不能将左值绑定到右值引用  
5 int&& rref3 = x + 3; // 合法, x + 3 是一个右值  
6 rref = x + 3; // 合法, x + 3 是一个右值
```

那么这玩意有什么用呢？

第一个用途：把拷贝变成搬运，即“移动语义”。举个例子：

```
1 string s1 = "hello";  
2 string s2 = s1 + "world";
```

我们发现，“`s1 + "world"`”是一个临时对象，生命周期就这一行。但是要把这玩意搬进 `s2`，就不得不做一次深拷贝，挺浪费的。

为了物尽其用，干脆把这个临时对象的资源直接搬进 `s2` 好了，不就省事了吗？这就是移动语义的核心思想。

实现移动语义需要用到右值引用，也就是说，重载：

```

1  string(const string& other); // 拷贝构造函数
2  string(string&& other); // 移动构造函数

```

这就是“搬家”实现的关键。右值引用的“马上要死”特性，保证了我们可以放心地把资源搬走，而不会影响原来的对象。这有着典型的写法：

```

1  string(string&& other) noexcept
2      : data(other.data) { // 直接抢夺资源指针
3      other.data = nullptr; // 搬家后，把原对象的指针置空，防止析构时重复释放
4  }

```

因为临时对象马上就销毁，所以我们“偷窃”其资源没有人会察觉，从而大大提高了性能。而自己实现的类，往往也需要实现移动构造函数和移动赋值运算符，从而支持移动语义。

而第二个用途就是“完美转发”。这在模板编程中非常有用，可以把参数原封不动地传递给另一个函数，而不会丢失其左值/右值属性。

```

1  template<class T>
2  void foo(T&& arg) {
3      bar(std::forward<T>(arg)); // 完美转发
4  }

```

上述代码中，T&& 是一个通用引用（也叫转发引用），它可以绑定到左值或右值。实际调用时，实参如果是左值，T 会被推导为左值引用类型；如果实参是右值，T 会被推导为右值类型。这样，std::forward<T>(arg) 就能根据 T 的类型正确地转发参数，保持其左值/右值属性。

unique_ptr 只能移动、不能拷贝的特性就是由右值引用实现的：

```

1  std::unique_ptr<int> p1 = std::make_unique<int>(5);
2  std::unique_ptr<int> p2 = p1; // 拷贝构造被删除
3  std::unique_ptr<int> p3 = std::move(p1); // OK, 移动构造

```

上述移动完后，p1 变成了空指针，p3 拥有了原来 p1 的资源。

9.2.3 C++ 函数的高级特性

C++ 下定义函数的方法和 C 完全一致。但是，C++ 对函数进行了扩展，增加了一些高级特性，例如函数重载、函数默认参数值等。

C++ 函数的编译原理 C++ 函数在编译时会进行名称修饰（Name Mangling），这是为了支持函数重载等特性。名称修饰的过程会将函数名、参数类型、返回类型等信息编码到符号名中，从而生成一个唯一的符号名。例如，假设我们有以下两个函数：

```

1  int add(int a, int b);
2  double add(double a, double b);

```

这个函数在 C 中是不允许的，因为函数名称相同，编译器不能区分它们，会报错“冲突声明”。但是在 C++ 中是允许的，因为编译器会对函数名称进行修饰，生成不同的符号名，例如：

```
1 call _Z3addii # 对应 int add(int, int)
2 call _Z3adddd # 对应 double add(double, double)
```

上述代码中，_Z3addii 和 _Z3adddd 分别是两个 add 函数的修饰名称，包含了参数类型的信息，从而使得编译器能够区分它们。但是我们发现一个问题：函数的返回值是不参与名称修饰的，这意味着如果两个函数只有返回值类型不同，而参数列表相同，那么它们仍然会冲突，编译器会报错。

但这就会导致一个问题：如果我们需要在 C++ 中调用 C 语言的函数怎么办？因为 C 语言不支持名称修饰，而 C++ 支持名称修饰，这就会导致链接错误。为了解决这个问题，C++ 提供了 extern "C" 关键字，可以告诉编译器按照 C 语言的方式来处理函数名称，从而避免名称修饰。例如：

```
1 extern "C" {
2     void c_function(int a);
3 }
```

上述代码中，c_function 函数会按照 C 语言的方式来处理名称，不会进行名称修饰，从而可以正确地链接 C 语言的函数。当然，这也会有所牺牲，在该块中的函数和变量不能使用 C++ 的特性，例如函数重载、命名空间等。

函数重载 函数重载是 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：

```
1 struct Tensor2D{
2     int x_dim, y_dim;
3 }
4 Tensor2D operator+(const Tensor2D& other) {
5     // 实现加法运算
6     Tensor2D result;
7     result.x_dim = this->x_dim + other.x_dim;
8     result.y_dim = this->y_dim + other.y_dim;
9     return result;
10 }
```

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 Tensor2D 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。刚才在名称修饰中，我们已经提到过，返回类型不参与名称修饰，因此不能仅通过返回类型来区分重载函数。

函数默认参数值 函数默认参数值是 C++ 中的一个特性，它允许我们在函数声明时为某些参数指定默认值。如果调用函数时没有提供这些参数的值，编译器会使用默认值。

例如：

```
1 void greet(const std::string& name = "Guest") {
2     std::cout << "Hello, " << name << "!" << std::endl;
3 }
4
5 greet(); // 输出: Hello, Guest!
6 greet("Alice"); // 输出: Hello, Alice!
```

在上述代码中，函数 `greet` 有一个默认参数 `name`，如果调用时没有提供该参数的值，默认值 `"Guest"` 会被使用，这个“没有提供参数”的规范名称是**参数缺省**。

需要注意的是，函数缺省的参数必须从右向左依次定义，不能在中间或左侧定义缺省参数。这一点和 C#、Python 等语言不同，必须牢记。例如，下面的代码是错误的：

```
1 void foo(int a = 10, int b); // 错误, b 没有默认值
```

正确的写法是：

```
1 void foo(int a, int b = 20); // 正确
```

而在函数声明和定义分离的情况下，默认参数值只能在函数声明中指定，而不能在函数定义中指定。例如：

```
1 void greet(const std::string& name = "Guest"); // 函数声明
2 void greet(const std::string& name) { // 函数定义
3     std::cout << "Hello, " << name << "!" << std::endl;
4 }
```

9.2.4 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：

```
1 auto x = 5; // 编译器推断 x 的类型为 int
2 auto y = 3.14; // 编译器推断 y 的类型为 double
3 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。

必须注意，`auto` 不是一个类型，而是一个占位符，编译器会根据变量的初始值来推断出一个具体的类型。因此，使用 `auto` 时必须确保变量有一个明确的初始值，否则编译器无法推断出类型，会导致编译错误。我们也绝对不能把 `auto` 当成一个万能的类型来使用，例如：

```

1 auto x; // 错误, 缺少初始值, 编译器无法推断出类型
2
3 auto x = 5; // 正确, 编译器推断 x 的类型为 int
4 x = "Hello"; // 错误, x 的类型已经被推断为 int, 不能赋值为字符串

```

不能把它当 Python 那样的动态类型来使用。如确实需要动态类型, 可以使用 `std::variant` 或 `std::any` 来实现, 或使用泛型编程 (模板) 来实现类型的灵活性。

但是依然需要注意: `std::variant` 和 `std::any` 以及模板虽然可以实现类型的灵活性, 但它们也不是动态类型, 他们仅仅是一个盒子, 里面可以装不同类型的东西, 但这个盒子本身的类型是确定的。具体使用见 C++ 高级特性那一章。

因此, 我们需要根据实际需求来选择合适的工具, 而不是滥用 `auto` 或者其他类型灵活性的工具。

注意

不要滥用 `auto`, 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。我们只推荐在变量的类型是确定的、类型的名称非常冗长、你知道这个变量的类型是什么或者大概是什么的情况下使用类型推断, 例如 `auto it = std::max_element(v.begin(), v.end());` 中, `it` 的类型是 `std::vector<int>::iterator` 是确定的, 这个类型名称很长, 你也知道它是个迭代器类型, 这时候使用 `auto` 是非常合适的。但是如果你真写了 `auto x = 5;` 这种代码, 那就属于滥用了, 是不提倡的。

如果变量的类型确实不确定 (例如变量类型会随着初始化方式的不同而改变), 这时候可以使用模板函数或模板类, 而不是使用 `auto`。如果你知道变量的类型是一个确定的类型, 但是你不知道具体是什么类型, 建议你先搞清楚这个变量的类型再写代码, 否则几乎必然会出错。

9.2.5 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。

```

1 using ll = long long; // 定义一个长整型的别名
2 using IntVector = std::vector<int>; // 定义一个整型向量的别名
3 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:

```

1 typedef long long ll; // 定义一个长整型的别名
2 typedef std::vector<int> IntVector; // 定义一个整型向量的别名
3 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:

```

1 template <typename T>
2 using Matrix = std::vector<std::vector<T>>; // 定义一个二维向量的别名
3 Matrix<int> m = {{1, 2}, {3, 4}}; // 使用别名创建一个二维向量
4 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。

9.2.6 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。类型强转包括两类：隐式转换和显式转换。隐式转换是编译器自动进行的，而显式转换则需要程序员手动指定。

在一般情况下，隐式转换是安全的，不会导致数据丢失或错误。然而，有些情况下隐式转换可能会引发诸如精度等问题。默认能够进行的隐式转换包括以下几步：

1. **标准整形提升**：所有比 int 小的整型（如 char、short）会被提升为 int 或 unsigned int。
2. **整形等级转换**：提升之后，如果类型仍不匹配，编译器会尝试将较小的整型转换为较大的整型（如 int 转为 long）。
3. **浮点等级转换**：如果涉及浮点数，编译器会尝试将较小的浮点类型转换为较大的浮点类型（如 float 转为 double）。
4. **混合类别转换**：如果操作数类型不同，编译器会尝试将整型转换为浮点型，以避免精度丢失。转换后的类型为与浮点数的类型相同的浮点类型。
5. **其他转换**：包括数组到指针、函数到指针、空指针常量、枚举到整型⁸、类类型的转换⁹等。

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，子类指针转父类指针、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**，几乎仅用于父类指针转子类指针。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。它也可以用于转引用，但是如果转不了不会返回空引用¹⁰而是报错。除非我们知道在干什么，否则不要使用它。

举例说明：

```

1 #include <iostream>
2 using namespace std;
3

```

⁸C++11 起的强枚举不能隐式转换

⁹指的是诸如代码：**struct** A{ A(**int**);}; **void** foo(A); foo(42); 中，foo(42) 把整数隐式转换为类

¹⁰没有“空引用”这种东西。

```
4 double d = 3.14;
5 int a = static_cast<int>(d); // 使用 static_cast 进行数值转换
6 int a = (int)d; // C 风格的强转，也行
7
8 const int c = 42;
9 int* p = const_cast<int*>(&c); // 使用 const_cast 去掉 const 属性
10 // *p = 100; // 但是修改原变量的值是个 UB
11
12 class Base;
13 class Derived : public Base;
14 // 注意：以上两行代码仅用于说明继承关系，实际过不了编译
15 Base* b = new Derived();
16 Derived* d = dynamic_cast<Derived*>(b); // 使用 dynamic_cast 将父类指针转为子类
17 Derived* d = static_cast<Derived*>(b); // 使用 static_cast 转型（不安全，但是能过编译）
18
19 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。

```
1 const volatile void* v = ...;
2 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++ 四大强转做显式强转。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。

说明

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。

9.2.7 Lambda 表达式

Lambda 表达式是 C++11 引入的一个特性，它允许我们在代码中定义匿名函数。Lambda 表达式可以捕获外部变量，并且可以作为参数传递给其他函数。它的基本语法如下：

1 [捕获列表] (参数列表) -> 返回类型 {


```
2 // 函数体
3 }
```

捕获列表用于指定哪些外部变量可以在 Lambda 表达式中使用，参数列表和返回类型与普通函数类似。Lambda 表达式可以直接在代码中定义，不需要单独声明。

例如：

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> nums = {1, 2, 3, 4, 5};
7     int sum = 0;
8     for_each(nums.begin(), nums.end(), [&sum](int n) { sum += n; }); // 使用
    ↪ Lambda 表达式计算总和
9     cout << "Sum: " << sum << endl; // 输出结果
10    return 0;
11 }
```

以上代码中，我们定义了一个 Lambda 表达式，该表达式能够捕获外部变量 `sum`，并对 `nums` 向量中的每个元素进行求和操作。Lambda 表达式可以使代码更简洁。

Lambda 表达式的类型是特殊的。它是一个独一无二的、不可命名的、由编译器生成的闭包类型。该类是匿名的，每一个 Lambda 表达式都会生成一个独一无二的类。这个类会重载 `operator()`，使得 Lambda 表达式可以像函数一样被调用。其类型是一个确定的类型，但是该类型既不能命名也不能写出。Lambda 表达式也不是一个函数指针，但是可以隐式转换为函数指针（前提是没有捕获任何外部变量）。

例如下列代码：

```
1 auto lambda = [](int x) { return x * x; };
```

其类型实际上类似于：

```
1 class __lambda_unique_name {
2 public:
3     int operator()(int x) const { return x * x; }
4 };
```

但是你永远不能直接写出这个类名，它是不可以访问的。

有些时候，我们需要对其进行存储或传递，此时需要声明其类型。此时，`auto` 是好的实践之一。但是如果要传递给函数怎么办呢？为了这个目的，C++11 引入了 `std::function` 模板类，它可以存储任何可调用对象，包括 Lambda 表达式、函数指针、函数对象等。使用 `std::function` 可以方便地传递和存储 Lambda 表达式。

```
1 #include <iostream>
```

```

2 #include <functional> // 包含 std::function 的头文件
3 using namespace std;
4
5 void applyFunction(const function<int(int)>& func, int value) {
6     cout << "Result: " << func(value) << endl; // 调用传入的函数并输出结果
7 }
8
9 int main() {
10     auto lambda = [](int x) { return x * x; }; // 定义一个 Lambda 表达式
11     applyFunction(lambda, 5); // 将 Lambda 表达式传递给函数
12     return 0;
13 }

```

然而值得注意的是，即使是用 `std::function`，它也仅仅是一个传递 Lambda 表达式的手段，而不是 Lambda 表达式本身的类型。且该类的性能开销比较大，有类型擦除的开销，可能会通过内联或堆分配来优化，因此在性能敏感的场景下应谨慎使用该类。实际上如果 Lambda 表达式不捕获任何外部变量，我们完全可以直接转成函数指针传递。

```

1 using FuncPtr = int(*)(int); // 定义一个函数指针类型
2 FuncPtr func = [](int x) { return x * x; }; // 将 Lambda 表达式转换为函数指针
3 cout << "Result: " << func(5) << endl; // 调用函数指针并输出结果

```

9.2.8 多文件编程

头文件和源文件

头文件是一些预先写好的代码的集合。通过包含头文件，我们可以使用这些预先写好的代码，而不需要重新编写它们。头文件的扩展名通常是 `.h` 或 `.hpp`。引入头文件只需要在文件的开头使用 `#include` 指令即可。

头文件有两种类型：标准库头文件和自定义头文件。标准库头文件是 C++ 标准库提供的头文件，通常使用尖括号括起来，例如 `<iostream>`、`<vector>` 等，这样会先在系统路径中查找，再去当前路径中查找；自定义头文件是用户自己编写的头文件，通常使用双引号括起来，例如 `"myheader.h"`，这样会先在当前路径中查找，再去系统路径中查找。

与“头文件”相对应的是**源文件**，它们通常包含程序的主要逻辑和实现代码。源文件的扩展名通常是 `.cpp`、`.cxx` 或 `.cc`。源文件可以包含头文件，并且可以定义函数、类和变量等。

```

1 // 这是一个源文件
2 #include <iostream> // 引入标准库头文件
3 #include "myheader.h" // 引入自定义头文件
4
5 int main() {
6     // 使用头文件中的代码
7     return 0;
8 }

```

警告

严格禁止使用所谓的“万能头文件” `#include <bits/stdc++.h>`，尤其是在工程中！该头文件有三个严重的问题：

- 该头文件不是 C++ 标准的一部分，而是 GCC 编译器提供的一个非标准头文件。使用该头文件会导致代码在不同编译器下表现不同，严重地影响代码的可移植性。
- 该头文件会引入整个标准库，从而显著地降低代码的可维护性，具体表现为：
 - 显著地增加编译时间，尤其是在大型项目中。
 - 引入大量实际上并不需要的库，严重地增加了命名冲突的风险。
 - 引入大量宏定义，可能会导致意想不到的行为。

综上所述，严格禁止使用该头文件！多背几个常用的头文件名称并不难，且对代码质量有显著提升。

自己写个头文件

有时候，我们自己需要写一个项目，这个项目代码量较大，可能有数千行。此时，我们需要多文件编程，以便于代码的组织和管理。

C++ 的多文件编程，文件结构通常是这样的：

```
1 project/  
2   main.cpp // 主要的程序入口  
3   module1.cpp // 模块 1 的实现  
4   module1.h // 模块 1 的头文件  
5   module2.cpp // 模块 2 的实现  
6   module2.h // 模块 2 的头文件  
7   ... 这里可能还有其他文件
```

由此可见，除了主要程序入口（main 函数所在的文件）之外，其他的头文件和源文件通常是成对出现的。在头文件中，一般包括类、函数、变量等的声明；在源文件中，一般包括类、函数、变量等的定义和实现。

自己写头文件时，通常包括以下内容：

- 声明函数、类、变量等的接口。
- 使用预处理指令防止重复包含。

例如，我们可以编写一个简单的头文件 `myheader.h`，包含一个函数的声明和定义：

```
1 #ifndef MYHEADER_H // 编译守卫，防止重复包含  
2 #define MYHEADER_H  
3 int add(int, int); // 函数声明  
4 #endif
```

然后在源文件 `myheader.cpp` 中实现这个函数：

```
1 #include "myheader.h" // 引入头文件  
2 int add(int a, int b) { // 函数定义  
3     return a + b;  
4 }
```

然后在主程序中使用这个函数：

```
1 #include <iostream>
2 #include "myheader.h" // 引入头文件
3
4 int main() {
5     int x = 5;
6     int y = 10;
7     int sum = add(x, y); // 调用 add 函数
8     std::cout << "Sum: " << sum << std::endl; // 输出结果
9     return 0;
10 }
```

最终对其进行编译：

```
1 g++ main.cpp myheader.cpp -o myprogram
```

这样，我们就完成了一个简单的多文件编程。在上述编译命令中，两个源文件的顺序无关紧要。

如果源文件数量过多，那么就不应该使用诸如 `gcc` 等工具手动地编译。此时，应该使用 `Makefile`、`CMake`、`XMake`、`Conan` 等构建工具来管理编译过程。关于后三个构建工具的使用，将在下一章中介绍。

再强调一遍：在多文件编程中，不建议使用 `using namespace std;`。尤其是头文件，严格禁止在头文件中使用该语句！

练习：重做

试着把上述 C 语言中的所有练习题，都用 C++ 的方式重写一遍。要求：

- 使用 C++ 的输入输出流（`iostream`）替代 C 的标准输入输出（`stdio.h`）。
- 使用 C++ 的标准库容器（如 `std::array`、`std::vector`）替代 C 的数组。
- 不得出现裸指针和直接的内存管理（如 `malloc`、`free`）。

比较 C 和 C++ 的代码风格和编程习惯，体会两者的异同。

练习：改错

以下几段代码存在一些错误、不良习惯或潜在问题，请找出并改正它们。

```
1 #include <iostream>
2 int main() {
3     auto x;
4     x = 42;
5     std::cout << x << '\n';
6 }
```

```
1 #include <iostream>
2 int main() {
3     int a[5] = {1,2,3,4,5};
4     for (auto v : a)
5         v += 10;
6     for (auto v : a)
```

```
7         std::cout << v << ' '; // 输出不符合预期的原因是?
8     }
```

```
1 // a.cpp
2 int add(int a, int b = 0) { return a + b; }
3
4 // main.cpp
5 int add(int a, int b = 0); // 声明
6 int main() {
7     std::cout << add(5) << '\n';
8 }
```

```
1 #include <iostream>
2 constexpr int sq(int x) { return x * x; }
3 int main() {
4     int n;
5     std::cin >> n;
6     constexpr int val = sq(n);
7     std::cout << val << '\n';
8 }
```

```
1 #include <cstdio>
2 #include <iostream>
3 int main() {
4     std::cout << "C++ ";
5     printf("C\n"); // 可能先打印 c
6 }
```

```
1 #include <iostream>
2 int f(int x) { return x + 1; }
3 int f(void* p) { return 2; }
4 int main() {
5     std::cout << f(nullptr) << '\n'; // 输出不确定
6 }
```

答案

以下是上述练习题的错误，改正方法留给读者自行思考。

1. **auto** 用作声明变量时，必须有初始值，否则编译器无法推断类型。改正方法：**auto** `x = 42;`。
2. 范围 `for` 循环中，**auto** `v` 是按值传递的，修改 `v` 不会影响原数组。改正方法：使用引用传递，**for** (**auto**& `v : a`)。
3. 函数默认参数值只能在函数声明中指定，不能在函数定义中指定。改正方法：在 `main.cpp` 中添加函数声明时指定默认参数值：

```
1 int add(int a, int b = 0);
```

4. `constexpr` 函数的参数必须是编译时常量，`n` 是运行时输入的变量，不能作为 `constexpr` 变量的初始化值。

5. C++ 的标准输出流和 C 的标准输出使用不同的缓冲区，可能导致输出顺序不确定。改正方法：使用同一种输出方式，或者在 `printf` 之后调用 `std::cout.flush()`。
6. `nullptr` 可以转换为任何指针类型，因此调用 `f(nullptr)` 时，编译器无法确定调用哪个重载版本。改正方法：显式指定调用的版本，例如 `f(static_cast<int*>(nullptr))`。

第十章 C++ 进阶

在上一章中，我们初步认识了 C++ 和 C 的重要区别：命名空间、用流代替 C 的标准输入输出等。我们也知道了 C++ 引入了很多有趣的标准特性，例如引用、函数重载、默认参数值、Lambda 表达式等。本章将进一步介绍 C++ 的高级特性和最佳实践，帮助读者更好地利用 C++ 的强大功能进行高效编程。

C++ 最重要的高级特性是面向对象编程和泛型编程，它们使得 C++ 在处理复杂系统和数据结构时具有显著优势。

10.1 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。

10.1.1 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。

```
1 class Point2D{
2 public:
3     static const int DIMENSION = 2; // 类的常量属性
4     static int count; // 类的静态属性
5     int x, y;
6     void move(int dx, int dy) {
7         x += dx;
8         y += dy;
9     }
10    Point2D(int x = 0, int y = 0) : x(x), y(y)
11    {
12        count++;
13    } // 构造函数
14    ~Point2D() { count--; } // 析构函数
15 }
```

于是，这下变量和函数成了一家人：

```
1 Point2D p(1, 2); // 创建一个 Point2D 对象 p, x=1, y=2
2 p.move(5, -3); // 移动点 p, 它自己知道怎么动!
3 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 `protect`，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。

10.1.2 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：

```
1 class Node {
2 public:
3     int data; // 节点数据
4     Node* next; // 指向下一个节点的指针
5     Node(int value) : data(value), next(nullptr) {}
6     Node& GetThis() const {
7         return *this; // 返回当前对象的引用
8     }
9 };
```

10.1.3 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，并且既没有返回值也没有返回类型；析构函数的名称是波浪号（`~`）加上类名，同样既没有返回值也没有返回类型。

构造函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（也尽可能不要）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。默认的构造函数只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），默认析构函数则按成员逆序调用成员的析构函数。满足这种条件的类也叫做**平凡且标准布局类**，在旧的实现中也叫 **POD 类型**：这种类型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。相应的，在构造函数、析构函数中执行一些其他操作的类则叫做**非 POD 类型**，也往往叫做**复杂类**。一个类可以有多个构造函数（本质上是函数重载），但是只能有一个析构函数。

比如说：

```
1 class Point2D{
2     public:
3         int x, y;
4         Point2D() {} // 默认构造函数，防止覆盖
5         Point2D(int _x, int _y){ // 自己写的构造函数
6             x = _x;
7             y = _y;
8         }
9         ~Point2D() {} // 自己写的析构函数
10 };
```

如果我们写了自己的构造和析构函数，那么编译器就不会再隐式地生成任何默认构造函数和析构函数。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。

在较新版本的 C++ 标准中，构造函数的属性初始化部分可以使用初始化列表来简单地编写。例如：

```
1 Point2D(int _x, int _y) : x(_x), y(_y) { ... }
```

说明

需要注意的一点是：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。当 malloc/free 未配对时几乎必然导致内存出毛病，而随便 delete this 导致的双重释放也是非常危险的。如果一定要用构造函数和析构函数管理资源，应使用 RAII 资源句柄（如 std::unique_ptr）而非裸指针。

拷贝构造函数¹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。默认拷贝操作对数据成员逐个复制；如果成员是指针，则仅复制指针值（即所谓“浅拷贝”）。当类拥有动态资源时，通常需要自定义深拷贝逻辑。

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，而下面这一段代码也展示了深拷贝操作中常见的“先复制、后交换”写法：

¹没有“拷贝函数”这种东西。

```

1 class Foo {
2     int* data;
3 public:
4     Foo(const Foo& rhs) : data(new int[*rhs.data]) {} // 构造函数，深拷贝
5     Foo& operator=(Foo rhs) { // 按值接收，已拷贝/移动
6         swap(*this, rhs); // 交换资源
7         return *this;
8     }
9     friend void swap(Foo& a, Foo& b) noexcept { std::swap(a.data, b.data); }
10    // noexcept 表示这个函数不会抛出异常
11 };

```

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，因此也被叫做类的赋值。

10.1.4 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：

```

1 class BankAccount {
2 private:
3     int balance; // 私有属性，外部无法直接访问
4 public:
5     BankAccount(int initialBalance) : balance(initialBalance) {}
6     void deposit(int amount) { // 公共方法，允许外部调用
7         if (amount > 0) {
8             balance += amount; // 增加余额
9         }
10    }
11    void withdraw(int amount) { // 公共方法，允许外部调用
12        if (amount > 0 && amount <= balance) {
13            balance -= amount; // 减少余额
14        }
15    }
16    int getBalance() const { // 公共方法，允许外部查询余额
17        return balance; // 返回余额
18    }
19 };

```

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。

提示

在 C# 中，封装有一对非常优雅的名词：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实现 getter。

10.1.5 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。基类中被重写的方法应被声明为 `virtual`，也就是**虚函数**。重写方法时建议加 `override` 关键字。

继承的基本格式如下：

```
1 class Shape { public: virtual double area() = 0; };
2 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。

除了重写父类已有的方法，我们也可以在子类中新增一些父类没有的属性和方法。例如：

```
1 class Circle : public Shape {
2 private:
3     double radius; // 圆的半径
4 public:
5     Circle(double r) : radius(r) {} // 构造函数
6     double area() override { // 重写父类的 area() 方法
7         return M_PI * radius * radius; // 计算圆的面积
8     }
9     double circumference() { // 新增方法，计算圆的周长
10        return 2 * M_PI * radius; // 计算圆的周长
11    }
12};
```

现在只剩下“子类的构造函数怎么写”这个问题了。在 C++ 的继承中，子类的构造函数需要调用父类的构造函数来初始化父类的属性。当父类有公共的默认构造函数（无参），且子类没有需要手动初始化的属性时，子类的构造函数可以不写，编译器会自动生成一个公共且无参的默认构造函数，并调用父类的默认构造函数来初始化父类的属性。只要不满足以上情况，就必须显式的提供子类的至少一个构造函数。

```
1 class Base {
2 public:
3     Base(int value) {
4         cout << "Base constructor with value: " << value << endl;
5     } // 带参数的构造函数
6     Base(int v1, int v2) {
7         cout << "Base constructor with values: " << v1 << ", " << v2 << endl;
8     } // 另一个带参数的构造函数
9     Base() {
10        cout << "Base default constructor" << endl;
11    } // 默认构造函数
12};
13 class Derived : public Base {
```

```

14 public:
15     Derived(int value) : Base(value) {
16         cout << "Derived constructor with data: " << value << endl;
17     } // 子类的构造函数, 调用父类的带参数构造函数
18     Derived(int v1, int v2) : Base(v1, v2) {
19         cout << "Derived constructor with data: " << value << endl;
20     } // 另一个子类的构造函数, 调用父类的另一个带参数构造函数
21     Derived() : Base() {
22         cout << "Derived default constructor" << endl;
23     } // 子类的默认构造函数, 调用父类的默认构造函数
24 };

```

C++11 以上的标准中, 如果子类只是想照抄父类的所有构造函数而不需要写自己的, 可以使用 `using` 关键字来简化代码:

```

1 class Derived : public Base {
2 public:
3     using Base::Base; // 直接继承父类的所有构造函数
4 };

```

需要注意的是, 以下两种代码是不过编译的:

```

1 class Base;
2 class Derived : public Base { ... }; // 错误, Base 类未定义

```

```

1 class Base { ... };
2 class Derived : public Base; // 错误, 子类的定义必须紧跟类体

```

在上述代码中, 一开始我们声明出了一个类 `Base`, 但是并未定义它。这样的类是“不完整的”, C++ 规定不能继承一个不完整的类。另一方面, 即使预先定义了基类, 但是在继承的时候没有跟出定义也是不允许的。

在实际操作中, 子类一般属于父类的一个特例, 或者更简单地说是子类是父类。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。

10.1.6 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:

```

1 Shape* shape = new Circle(); // 创建一个 Circle 对象, 并将其赋值给 Shape 指针
2 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。

10.1.7 友元函数

我们已经知道，对于一个类的属性和方法，有的是私有的、有的是公共的；从类外无法访问类的私有属性和方法。但是友元函数是一个例外，它可以访问类的私有属性和方法。友元函数的声明方式非常简单，只需要在函数前面加上 friend 关键字即可。友元函数可以是类的成员函数，也可以是全局函数。但是，友元函数的定义必须在类的外部，而非在类的内部。

```
1 class MyClass {
2     private:
3         int secret; // 私有属性
4     public:
5         MyClass(int value) : secret(value) {} // 构造函数
6         friend void revealSecret(const MyClass& obj); // 声明友元函数
7 };
8 void revealSecret(const MyClass& obj) {
9     cout << "The secret is: " << obj.secret << endl; // 访问私有属性
10 }
```

一般情况下我们很少用到友元函数，因为它破坏了类的封装性。然而，在某些情况下，友元函数可以提供更高效的访问方式，尤其是在需要频繁访问类的私有属性时。

10.2 泛型编程

泛型编程的意思是：编写与类型无关的代码，从而实现代码的重用。

在 C++ 中，泛型编程的核心机制是模板 (Templates)。模板允许我们编写通用的代码，可以处理不同类型的数据。除此之外，C++11 引入了类型推断 (Type Inference) 和类型别名 (Type Aliases)，进一步增强了泛型编程的能力。

10.2.1 函数模板

比方说我们想写一个加法：

```
1 template <typename T>
2 T add(T a, T b) {
3     return a + b; // 返回 a 和 b 的和
4 }
5 int main() {
6     int x = 5, y = 10;
7     cout << add(x, y) << endl; // 调用 add 函数，输出 15
8     double a = 3.14, b = 2.71;
9     cout << add(a, b) << endl; // 调用 add 函数，输出 5.85
10
11     return 0;
12 }
```

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）。编译器会自动根据调用推导其类型参数，当然也不是不可以手动指定类型参数，例如 `add<int>(x, y)`。

提示

在试着调用一个函数的时候，编译器会按以下方式查找合适的函数：

1. 首先查找是否有与调用参数类型完全匹配的非模板函数。

```
1 void add(int a, int b) { cout<<"non-template function called"<<endl; }
   ↳ // 非模板函数
2 template <typename T>
3 T add(T a, T b) { cout<<"template function called"<<endl; } // 模板函数
4 add(5, 10); // 调用非模板函数
```

2. 如果没有找到，则查找是否有与调用参数类型匹配的模板函数，并进行模板实例化。

```
1 template <typename T>
2 T add(T a, T b) { cout<<"template function called"<<endl; }
3 add(5, 10); // 调用模板函数
```

3. 如果找到多个匹配的模板函数，编译器会尝试进行模板参数推导，以确定最合适的模板实例。

```
1 template <typename T>
2 T add(T a, T b) { cout<<"template function 1 called"<<endl; }
3 template <typename T>
4 T add(T a, double b) { cout<<"template function 2 called"<<endl; }
5 add(5, 3.14); // 调用模板函数 2
```

4. 如果仍然无法确定唯一的匹配，编译器会报错，提示存在二义性。

```
1 template <typename T>
2 T add(T a, T b) { cout<<"template function 1 called"<<endl; }
3 template <typename T>
4 T add(T a, T* b) { cout<<"template function 2 called"<<endl; }
5 int x = 5;
6 int* p = &x;
7 add(x, p); // 二义性错误，无法确定调用哪个模板函数
```

5. 如果没有找到任何匹配的函数，编译器会报错，提示找不到合适的函数。

上述规则虽在实际工程中意义有限，但了解其工作原理有助于理解模板函数的行为。

10.2.2 类模板

类模板的语法类似，只不过是定义一个类而不是一个函数：

```
1 template <typename T>
2 class Box {
3 public:
4     T value; // 存储一个值
5     Box(T v) : value(v) {} // 构造函数
```

```

6     T getValue() const { return value; } // 获取值的方法
7 };
8 Box<int> intBox(42); // 创建一个存储整数的 Box 对象
9 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。

10.2.3 模板特化

模板特化指的是为特定类型提供专门的实现。模板特化可以分为完全特化和部分特化。完全特化是为某个具体类型提供一个完整的实现，而部分特化则是为一组类型提供一个通用的实现。

语法如下：

```

1 // 这里需提前定义通用模板
2 template <typename T>
3 class Box {
4 public:
5     T value;
6     Box(T v) : value(v) {}
7     T getValue() const { return value; }
8 };
9
10 template <>
11 class Box<bool> { // 为 bool 类型提供特化实现，完全特化
12 public:
13     bool value;
14     Box(bool v) : value(v) {}
15     void toggle() { value = !value; } // 特有的方法，切换布尔值
16 };
17
18 template <typename T>
19 class Box<T*> { // 为指针类型提供部分特化实现，部分特化
20 public:
21     T* value;
22     Box(T* v) : value(v) {}
23     T getValue() const { return *value; } // 解引用指针获取值
24 };

```

需要说明，模板特化虽然确实有和类的继承类似的样子，但必须要注意它们之间的区别：模板特化是针对特定类型提供不同的实现，**不会继承通用模板的成员**；而类的继承则是子类继承父类的成员，并且可以重写父类的方法。因此，在使用模板特化时，我们需要为特化类型提供完整的实现，而不能依赖于通用模板的成员。

为了保证模板特化的正确性（也就是如果确实需要类似继承的功能），我们需要这样写：

```

1 template <typename T>
2 class BaseBox {
3 public:
4     T value;
5     // 在这里写通用的成员函数和属性
6     BaseBox() { /* 构造函数实现 */ }

```

```

7 }
8
9 template <typename T>
10 class Box : public BaseBox<T> {
11     // 在这里写针对其他类型的特化实现
12     using BaseBox<T>::BaseBox; // 继承 BaseBox 的构造函数
13 };
14
15 template <>
16 class Box<bool> : public BaseBox<bool> {
17     // 在这里写针对 bool 类型的特化实现
18     using BaseBox<bool>::BaseBox; // 继承 BaseBox 的构造函数
19     void toggle() { this->value = !this->value; } // 特有的方法, 切换布尔值
20     // 使用 this->value 来访问 BaseBox<bool> 中的 value 属性
21 };

```

这样不仅保证了模板特化的正确性, 还避免了代码重复, 提高了代码的可维护性, 是常用的最简单方式。

另一种实现方式是利用 C++20 提供的新特性: 概念 (Concepts) 来约束模板类中的部分方法, 使它们在特定情况下才可用, 从而实现类似于模板特化的效果。例如:

```

1 template <typename T>
2 class Box {
3 public:
4     T value;
5     Box(T v) : value(v) {}
6     T getValue() const { return value; }
7
8     // 只有当 T 满足概念时, 才提供 toggle 方法
9     // 下列概念的意思是: T 和 bool 是一样的
10    void toggle() requires std::same_as<T, bool> {
11        value = !value; // 切换布尔值
12    }
13 };
14
15 Box<bool> boolBox(true); // 创建一个存储布尔值的 Box 对象
16 boolBox.toggle(); // 调用 toggle 方法, 切换布尔值
17 Box<int> intBox(42); // 创建一个存储整数的 Box 对象
18 // intBox.toggle(); // 编译错误, 不满足概念约束

```

实际上这并没有使用模板特化, 而是利用概念来实现了类似的功能。

在实际情形下还可能采取更复杂的实现方式, 如 CRTP 模式 (Curiously Recurring Template Pattern)、显式模板实例化法等, 这应取决于团队的编码规范和实际需求。

10.2.4 非类型模板参数

非类型模板参数是指模板参数不仅可以是类型, 还可以是常量值 (如整数、枚举等)。这种特性允许我们在编译时传递一些固定的值, 从而实现更灵活的模板设计。

```

1 template <int N>
2 void printTimes(const std::string& str) {

```

```

3     for (int i = 0; i < N; ++i) {
4         std::cout << str << std::endl; // 输出字符串 N 次
5     }
6 }
7 printTimes<3>("Hello"); // 输出"Hello" 三次

```

在上述代码中，模板参数 `N` 是一个整数常量，它决定了函数 `printTimes` 输出字符串的次数。但是这玩意儿用得并不多，毕竟大多数情况下我们并不需要在编译时传递常量值。

10.2.5 变参模板

变参模板允许我们定义接受可变数量模板参数的模板。这样，我们可以编写更加通用和灵活的代码。变参模板使用省略号 (...) 来表示可变数量的参数。

```

1 template <typename... Args>
2 void printAll(const Args&... args) {
3     ((std::cout << ... << args), ...); // 折叠表达式，输出所有参数
4     std::cout << std::endl; // 最终的输出
5 }
6 printAll(1, 2.5, "Hello", 'A'); // 输出

```

这个倒是挺有用的，例如计算一组数的均值：

```

1 template <typename... Args>
2 double mean(Args... args) {
3     return (static_cast<double>(args) + ...) / sizeof...(args); // 计算均值
4 }
5 double result = mean(1, 2, 3, 4, 5); // result = 3.0

```

10.2.6 模板元编程

模板元编程是一种利用模板机制在编译时进行计算的编程技术。通过模板元编程，我们可以在编译阶段执行一些复杂的计算，从而生成高效的代码。模板元编程通常用于实现类型特性检测、类型转换等功能，是 C++ 泛型编程的顶级玩法。比如说，利用模板元编程实现阶乘计算：

```

1 template <int N>
2 int fact(){ return N * fact<N - 1>(); }
3
4 template <>
5 int fact<0>() { return 1; } // 模板特化，处理 N=0 的基础情况
6
7 int main() {
8     std::cout << "Factorial of 5: " << fact<5>() << std::endl; // 输出 120
9     return 0;
10 }

```

模板元编程不是打表！它只不过是把计算从运行时挪到了编译期而已。

实际上很多人认为这个是多此一举，因为使用 `constexpr` 函数更简单：


```
1 constexpr int fact(int n) { return n <= 1 ? 1 : n * fact(n - 1); } // 是常量表达式  
↪ 函数
```

不过模板元编程在某些场景下仍然有其独特的优势，例如类型计算和编译时断言等。

10.3 STL 和其他标准库

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。

警告

严格禁止自己对 STL 进行重新实现！STL 的实现经过了大量的优化和测试，自己重新实现容易出错且效率低下、维护困难，也不安全。STL 确实存在时间复杂度常数项大的问题，但是这永远不应该成为你重新实现的理由，你自己实现的东西几乎必然会比 STL 更慢、更不安全、更难用；就算是比 STL 快，也基本上会被 -O2 抹平一切差距（工程上哪有不开优化的）。除非你的实现确实全方位吊打 STL，但是那样的话你也不需要 STL 了，你可以直接去为 ISO C++ 标准委员会做贡献了！

10.3.1 容器

举个最常见的例子：

```
1 std::vector<int> v = {3,1,4}; // 自动扩容的数组  
2 std::set<int> s = {3,1,4}; // 自动排序的红黑树  
3 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。实际上是单一内存连续块。
- 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。实际上是分段连续的内存块。
- 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，`set` 和 `map` 是最好的选择。
- 只要最大最小：如果只关心最大值或最小值，使用 `priority_queue`。
- 频繁在中间插入删除：如果需要频繁插入和删除元素，使用 `list`。
- 频繁需要两头插入删除：如果只关心两端（尤其是头部）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` 和 `array`。它们是最通用的容器，适用于大多数场景。如果只关心尾部的频繁增删，也可以不用 `deque`，直接用 `vector`。

提示

`array` 和 C 风格数组的区别在于：前者是一个类，提供了更多的功能和安全性，例如边界检查、迭代器支持等；而后者只是一个简单的内存块，没有任何附加功能。建议尽量使用 `std::array`，除非有特殊需求必须使用 C 风格数组。

`array` 比 `vector` 在大多数情况下更高效，尤其是在小规模数据时，因为它避免了动态内存分配的开销。但是 `array` 的大小是固定的，不能动态调整；而 `vector` 可以根据需要动态扩展。但问题是，`array` 是栈分配的，而 `vector` 是堆分配的，栈空间有限，如果开的数组太大会导致栈溢出。因此在开大数组时，建议使用 `vector`。

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。

说明

在大多数情况下，`std::vector<bool>` 和 `std::vector<T>` 实现有区别。前者是一个极为特殊的实现，使用位压缩来存储布尔值，因此它不是一个真正的向量，而是一个位集合（`bitset`）。这使得 `std::vector<bool>` 在某些情况下效率更高，但也导致了一些不兼容的问题。例如，`v[i]` 返回的是一

个代理对象而不是一个引用；`auto x = v[i]` 返回的是值拷贝而不是常规的数据类型。

这是因为，在 `std::vector<bool>` 中，每个布尔值只占用一个位 (bit)，而不是一个字节 (byte)。因此，无法直接返回一个引用，因为引用必须指向一个完整的字节。为了实现对单个位的访问，STL 使用了一个代理对象来封装对位的操作，这个代理对象提供了类似引用的行为，但实际上并不是引用。

我们使用者不关心 `std::vector<bool>` 的实现细节，只需要记住以下五件事就行了：

1. 不能使用 `auto& x = v[i]` 来获取元素的引用，因为代理对象不能绑定到非常引用；
2. 不能使用 `&v[i]`，因为单个位没有地址；
3. `std::vector<bool>` 的迭代器不是常规迭代器的实现，不是指针；
4. `std::vector<bool>` 线程不安全（位压缩导致读写冲突，完全无法保证原子性）；
5. 排序、查找等算法能用但是缓慢。

10.3.2 迭代器

迭代器可以认为是指针的语法糖。一个示例：

```
1 for(auto it=v.begin(); it!=v.end(); ++it) cout<<*it<<' ';
2 // 或者直接：
3 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。

10.3.3 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：

```
1 std::sort(v.begin(), v.end()); // 混合高速排序，结合快排、堆排等算法
2 std::binary_search(v.begin(), v.end(), 4); // 二分
3 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。

10.3.4 字符串、流和字符串流

字符串和字符串流是 C++ 中处理文本数据的重要工具。C++ 提供了两种主要的字符串类型：C 风格字符串（以 `char` 数组表示）和 C++ 字符串（使用 `std::string` 类）。C++ 字符串更安全、更易用，推荐优先使用。

C++ 字符串（`std::string`）在头文件库 `<string>` 中定义，每一个字符串是一个**对象**，而不是数组。该类提供了许多方便的方法来操作字符串，例如：

- `size()` 或 `length()`：获取字符串长度。
- `substr(pos, len)`：获取子字符串。
- `find(str)`：查找子字符串的位置。
- `replace(pos, len, str)`：替换子字符串。
- `append(str)`：追加字符串。
- `insert(pos, str)`：插入字符串。
- `erase(pos, len)`：删除子字符串。
- `c_str()`：获取 C 风格字符串。

流是 C++ 中处理输入输出的重要工具。C++ 提供了两种主要的流类型：输入流（`istream`）和输出流（`ostream`）。输入流用于从标准输入或文件中读取数据，输出流用于向标准输出或文件中写入数据。我们不关心流是怎么实现的，但是应当理解其工作原理：

流有一个内部维护的缓冲区，输入流维护一个读指针（读取位置），输出流维护一个写指针（写入位置）。当我们从输入流中读取数据时，流会从缓冲区中读取数据，读取部分从读指针开始并向后移动，直到读完或者读取被换行符、空格等掐断，此时缓冲区内被读取的这一部分会被输入流吃掉，读指针会移动到读取位置的下一个位置（**然后缓冲区内就没有被读进去的这部分内容了**）；当我们向输出流中写入数据时，流会将数据写入缓冲区，写入部分从写指针开始并向后移动，直到写完或者缓冲区满，然后把写指针移动到写入位置的下一个位置；至于什么时候把缓冲区内的数据真正写入输出设备（例如屏幕、文件等），这取决于缓冲区什么时候刷新，包括缓冲区满、手动刷新、`endl`、程序结束、关联流（通常是 `cin`）请求刷新五种情况。

做题的时候，部分居心叵测（无端）的出题人会在输入输出上做文章。这时，只需要记得在做题的时候少用 `endl`，多用 `\n` 就能解决大多数问题了。在极为特殊的情况下，我们可以直接切断 `cin` 和 `cout` 的关联，来提升输入输出效率（仅限于极少量的竞赛场景）。切断关联的方法是：

```
1 std::ios::sync_with_stdio(false); // 关闭 C 和 C++ 的同步
2 std::cin.tie(nullptr); // 取消 cin 和 cout 的绑定
```

当然，使用 C 风格的输入输出（`scanf` 和 `printf`）通常会更快一些。

字符串流（`std::stringstream`）在头文件库 `<sstream>` 中定义，它允许我们将字符串作为输入输出流进行处理。字符串流提供了类似于标准输入输出流的接口，可以方便地进行格式化输入输出操作。例如：

```
1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4
5 int main() {
6     string str = "123 456 789";
7     stringstream ss(str); // 创建字符串流对象
8     int a, b, c;
9     ss >> a >> b >> c; // 从字符串流中读取整数
10    cout << "a: " << a << ", b: " << b << ", c: " << c << endl; // 输出结果
11    return 0;
12 }
```

以上代码中，我们使用字符串流将字符串中的整数提取出来，并输出结果。字符串流还提供了其他常用的方法，例如：

- `str()`：获取字符串流中的字符串。
- `clear()`：清空字符串流的状态。
- `seekg(pos)`：设置读取位置。
- `seekp(pos)`：设置写入位置。
- `tellg()`：获取当前读取位置。
- `tellp()`：获取当前写入位置。

当然，我们很少对流进行直接操作，更多的还是用它配合 `std::getline` 等函数来处理输入输出。`getline` 函数能够接受三个参数：输入流、输出字符串、分隔符（默认为换行符）。它会从输入流中不停读取数据，直到遇到分隔符为止，并将读取的数据存储到输出字符串中，以便于我们进行后续处理。

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5
6 int main() {
7     string line; // 一堆逗号分隔的单词
8     while (getline(cin, line)) { // 从标准输入中读取一行数据
9         stringstream ss(line); // 创建字符串流对象
10        string word;
11        while (getline(ss, word, ',')) { // 从字符串流中读取单词
12            cout << word << endl; // 输出单词
13        }
14    }
15    return 0;
16 }
```

10.3.5 定长整数

在 C++ 中，有时候我们需要精确地控制整数和浮点数的长度，以满足各种各样奇奇怪怪的要求（例如缓存优化、文件格式、网络协议等）。C++11 引入了头文件 `<cstdint>`，提供了一组定长整数类型，例如：

- `int8_t`：8 位有符号整数。
- `uint8_t`：8 位无符号整数。
- `int16_t`：16 位有符号整数。
- `uint16_t`：16 位无符号整数。
- `int32_t`：32 位有符号整数。
- `uint32_t`：32 位无符号整数。
- `int64_t`：64 位有符号整数。
- `uint64_t`：64 位无符号整数。

该头文件还提供了一系列宏，分别用于定义定长整数的最大值和最小值，例如：

- `INT8_MIN`：8 位有符号整数的最小值。
- `INT8_MAX`：8 位有符号整数的最大值。

至于定长的浮点数，C++ 标准库并没有直接提供这样的类型。

10.3.6 位运算

位运算是对于整数的二进制位进行操作的运算。常见的位运算符有以下几种：

- 按位与（AND）：`&`，对两个整数的每一位进行与运算，只有两个位都为 1 时结果才为 1，否则为 0。

- 按位或 (OR): `|` , 对两个整数的每一位进行或运算, 只要有一个位为 1 时结果就为 1, 否则为 0。
- 按位异或 (XOR): `^` , 对两个整数的每一位进行异或运算, 当两个位不同时结果为 1, 否则为 0。
- 按位取反 (NOT): `~` , 对一个整数的每一位进行取反运算, 0 变为 1, 1 变为 0。
- 左移 (Left Shift): `<<` , 将一个整数的二进制表示整体向左移动指定的位数, 右边补 0。
- 右移 (Right Shift): `>>` , 将一个整数的二进制表示整体向右移动指定的位数, 左边补 0 (对于无符号整数) 或补符号位 (对于有符号整数)。

位运算通常用于相当底层的编程, 例如操作硬件寄存器、加密算法、图像处理、乘除法加速等, 其效率非常高。有时候, 在不太底层的地方, 位运算往往也有奇效。

例题

有一串整数, 其中只有一个数出现了奇数次, 其他数都出现了偶数次。请编写一个 C++ 程序, 找出这个出现奇数次的数。要求时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

输入: 一串整数, 整数之间用空格隔开, 输入以 -1 结束。保证 `int` 能够存储所有整数。**输出:** 出现奇数次的整数。

答案

这道题可以使用按位异或运算来解决。我们知道, 按位异或运算有以下性质:

- $a \oplus a = 0$, 即一个数和自己异或结果为 0。
- $a \oplus 0 = a$, 即一个数和 0 异或结果为自己。
- 异或运算满足交换律和结合律。

因此, 如果我们将所有整数进行异或运算, 那么出现偶数次的数会被抵消掉, 最终只剩下出现奇数次的数。

我们可以用一个变量 `result` 来存储异或的结果, 初始值为 0。然后, 我们不断读取输入的整数, 并将其与 `result` 进行异或运算, 直到遇到 -1 为止。最后, `result` 中存储的就是出现奇数次的数。

下面是关键处的代码实现:

```
1 while (cin >> num && num != -1)
2     result ^= num; // 使用按位异或运算
```

结合输入输出, 我们可以写出完整的程序。该程序留作练习。

练习

1. 请完成上述程序。
2. 与、或、非有没有类似的性质? 请写下来。

在 STL 中, 有一个专门处理位集合的容器: `std::bitset`。它可以看作是一个定长的位数组, 支持高效的位操作。我们可以使用它来存储和操作大量的布尔值, 例如:

```
1 #include <iostream>
2 #include <bitset> // 包含 bitset 的头文件
3 using namespace std;
```

```
4 int main() {
5     bitset<8> b; // 创建一个 8 位的位集合, 初始值为 00000000
6     b.set(3);    // 将第 3 位 (从 0 开始计数) 设置为 1, 变为 00001000
7     b.flip(1);   // 将第 1 位取反, 变为 00001010
8     b.reset(3);  // 将第 3 位重置为 0, 变为 00000010
9     cout << b << endl; // 输出位集合的值
10    cout << "Number of set bits: " << b.count() << endl; // 计算并输出 1 的个数
11    return 0;
12 }
```

以上代码中, 我们创建了一个 8 位的位集合, 并对其进行了设置、取反和重置操作。最后, 我们输出了位集合的值和其中 1 的个数。位集合还提供了其他常用的方法, 例如:

- `all()`: 检查所有位是否都为 1。
- `any()`: 检查是否有任意一位为 1。
- `none()`: 检查是否所有位都为 0。
- `size()`: 获取位集合的大小。
- `to_string()`: 将位集合转换为字符串。

10.3.7 正则表达式

在 20.4 中, 我们已经介绍了正则表达式的基本概念和语法。C++11 引入了头文件 `<regex>`, 提供了一组类和函数来处理正则表达式。

主要的类有以下几种:

- `std::regex`: 表示一个正则表达式对象。
- `std::smatch`: 表示一个字符串匹配结果容器, 需要搭配 `std::string` 使用。
- `std::cmatch`: 也是一个字符串匹配结果容器, 但需要搭配 `const char[]` 使用。

“字符串匹配结果容器”指的是这是一个容器, 能够存储匹配的详细信息。其存储的对象是 `std::sub_match` 对象。特别的, 其 0 索引存储的是整个字符串匹配的结果, 而其余的索引存储各个捕获组²匹配的字符串。

主要的函数有以下几种:

- `std::regex_match`: 检查整个字符串是否匹配正则表达式。
- `std::regex_search`: 检查字符串中是否包含匹配正则表达式的子串。
- `std::regex_replace`: 将字符串中匹配正则表达式的部分替换为指定的字符串。在这里, 允许使用诸如 `$&` (整个匹配)、`$ <number>` (`<number>` 是捕获组的 index, 0 是整个匹配) 等占位符。

代码举例:

```
1 #include <iostream>
2 #include <regex>
3 #include <string>
4
5 int main() {
```

²捕获组指的是把正则表达式里面的一部分模式用圆括号标出来, 并告诉引擎这段子串一旦匹配成功就单独记下来方便以后反复使用。


```

6  /*===== 1. 正则与待测字符串 =====*/
7  // 匹配中国大陆常见手机号格式: XXX-XXX-XXXXX (3-3-5)
8  std::regex pattern(R"((\d{3})-(\d{3})-(\d{5}))");
9  std::string good = "123-456-78910"; // 完全匹配
10 std::string bad = "123-456-789ab"; // 最后一段不是数字
11 std::string text = "Call me at 123-456-78910 or 987-654-32100.";
12
13 /*===== 2. std::regex_match: 整串必须完全匹配 =====*/
14 bool is_good = std::regex_match(good, pattern);
15 bool is_bad = std::regex_match(bad, pattern);
16 std::cout << std::boolalpha;
17 std::cout << "good match : " << is_good << '\n'; // true
18 std::cout << "bad match : " << is_bad << '\n'; // false
19
20 /*===== 3. 取出捕获组 =====*/
21 std::smatch m;
22 if (std::regex_match(good, m, pattern)) {
23     std::cout << "\n捕获组演示: \n";
24     std::cout << " 整 match : " << m[0] << '\n'; // 123-456-78910
25     std::cout << " 第 1 组 : " << m[1] << '\n'; // 123
26     std::cout << " 第 2 组 : " << m[2] << '\n'; // 456
27     std::cout << " 第 3 组 : " << m[3] << '\n'; // 78910
28 }
29
30 /*===== 4. std::regex_search: 只要子串匹配即可 =====*/
31 if (std::regex_search(text, m, pattern)) {
32     std::cout << "\nsearch 找到: " << m[0] << '\n'; // 123-456-78910
33 }
34
35 /*===== 5. std::regex_replace: 替换 + 占位符 =====*/
36 // 把电话号码换成“区号 $1-局号 $2-号码 $3”的形式
37 std::string repl = std::regex_replace(text, pattern,
38                                     " 区号 $1-局号 $2-号码 $3");
39 std::cout << "\nreplace 结果: \n" << repl << '\n';
40 // -> Call me at 区号 123-局号 456-号码 78910 or 区号 987-局号 654-号码 32100.
41
42 /*===== 6. 常用占位符小结 =====*/
43 // $& 整个匹配
44 // $n 第 n 个捕获组, n=0 等价于 $&
45 // $$ 字面量 '$'
46 std::string demo = "Price: 199USD";
47 std::regex pr(R"((\d+)(USD))");
48 std::cout << "\n占位符演示: \n"
49           << std::regex_replace(demo, pr, "$1 $$100") // 199 $100
50           << '\n';
51
52 return 0;
53 }

```


10.3.8 小练

例题

假设现在有许多武士要角斗。每个武士都有一个名字和一个体力值，当两个武士相互角斗的时候，体力值较高的武士将会获胜，而体力值较低的武士会耗尽体力，并被淘汰。然而，角斗会消耗武士的体力值，因此每一次角斗后，胜者的体力值会减少等同于败者当前体力值的数值。如果有两个武士体力相等，则他们都会耗尽体力被淘汰。为了保证公平，武士们决定按照体力值从高到低的顺序进行角斗；如果有多个体力值最高的武士，那么他们会在这些武士中选择姓名字典序最靠前的两个武士进行角斗（例如有 abc 三个武士，则 a 的姓名字典序最靠前，b 次之，c 最靠后）。每次角斗后，胜者会继续参与角斗，直到只剩下一个武士或者没有武士剩下为止。

请编写一个程序，模拟武士们的角斗过程，并输出角斗的结果。

输入格式：共 $n+1$ 行。你会接收到一个数字 n ，表示武士的数量。接下来 n 行，每行包含一个武士的名字和体力值（用空格分隔）。保证 n 不大于一百万，且保证没有两个武士的名字相同。

输出格式：输出最后剩下的武士的名字和体力值。如果没有武士剩下，则输出 “None Left”。

答案

上述题目看起来难度颇高。这也会是在类似于 OJ 上出现的常见题目类型之一：不会像前两个题目一样，给你明显的提示和思路（例如“使用筛法”），而是需要你自己思考解决问题的思路。对于这种题目，我们除了需要会语言以外，还要有一定的算法知识。好在这个题目比较简单，算法很直接，重点是怎么实现。

思路

我们看到，武士们按照体力值从高到低的顺序角斗，这说明我们非常需要一个数据结构来存储武士们的信息，并且能够不停地获取体力最高的武士（对于体力值次高的武士，我们取两次就行），这让我们想到 STL 的一个重要成员：优先队列。另一方面，我们发现 n 的数量级在一百万，这对算法的时间要求较高，而优先队列能够很好地满足这个要求。

我们用 C++17 的语法来做题。

我们定义一下武士这个数据类型和优先队列：

```
1 class Warrior {
2 private:
3     std::string name_; // 武士的名字
4     int health_;       // 武士的体力值
5 public:
6     Warrior() = default; // 默认构造函数
7     Warrior(std::string name, int health)
8         : name_(std::move(name)), health_(health) {} // 构造函数
9
10    // Getters
11    const std::string& name() const { return name_; }
12    int health() const { return health_; }
13
14    // utils
15    void reduce(int amount) { health_ -= amount; } // 减少体力值
16 };
17
18 struct Cmp { // 比较器，用于优先队列排序
19     bool operator()(const Warrior& a, const Warrior& b) const {
```

```

20         return std::tie(b.health(), a.name()) <
21                std::tie(a.health(), b.name());
22         // 体力值高的优先级高, 体力值相等时名字字典序靠前的优先级高
23     }
24 }
25
26 using Queue = std::priority_queue<Warrior,
27                                   std::vector<Warrior>,
28                                   Cmp>;
29
30 Queue warriors; // 定义一个优先队列, 存储武士

```

再下一步就是处理角斗的逻辑了。由于优先队列会自动处理上述武士的排序问题, 我们只需要不断地从优先队列中取出两个武士进行角斗即可:

```

1 std::optional<Warrior> run(Queue& q) {
2     while (q.size() > 1) { // 当队列中有两个或以上武士时
3         Warrior first = q.top(); q.pop(); // 取出体力最高的武士
4         Warrior second = q.top(); q.pop(); // 取出体力次高的武士
5
6         if (first.health() == second.health()) {
7             // 如果体力相等, 则两人都被淘汰
8             continue;
9         } else {
10            // 否则, 胜者体力减少败者体力值, 且 a 肯定是胜者
11            first.reduce(second.health());
12            q.push(std::move(first)); // 将胜者重新加入队列, 这里用 std::move 避
            ↪ 免拷贝
13        }
14    }
15    return q.empty() ?
16           std::nullopt :
17           std::make_optional(q.top()); // 返回最后剩下的武士
18 }

```

接下来, 处理读取逻辑:

```

1 int n;
2 cin >> n; // 读取武士数量
3 for (int i = 0; i < n; ++i) {
4     std::string name;
5     int health;
6     std::cin >> name >> health; // 读取武士的名字和体力值
7     q.emplace(std::move(name), health); // 将武士加入优先队列
8     // emplace 直接在队列内构造对象, 避免不必要的拷贝
9 }

```

最后, 处理输出:

```

1 if (auto result = run(std::move(warriors)); result.has_value()) // 这里使用 if
    ↪ 初始化语句
2     std::cout << result->name() << " " << result->health() << "\n";
3 else
4     // 如果没有武士剩下
5     std::cout << "None Left\n";

```

```
6 return 0;
```

当然，我们肯定不能把这些代码直接交上去，我们需要把它们拼接在一起，成为一个可以执行的程序。

练习

1. 请将上述代码拼接在一起，完成一个完整的 C++ 程序，并在本地编译运行。
2. 试着使用 `set`、`map`、`vector` 等其他容器来重新实现上述题目，比较数据量较大时的性能差异。

说明

部分同学可能会想：为什么我要用优先队列，而不是用 `set`、`map`、`vector` 等其他容器？这个问题问得很好。

我们先来解释“优先队列为什么行”的问题。优先队列是一个特殊的容器，它使用二叉堆实现，速度很快，且我们仅考虑“每次只关心全局最大值”的问题。在上述题目中，我们实际上只将两件事反复循环：把人放进去，把最该打架的两个人拿出来，这两件事恰好符合优先队列的特性。优先队列插入弹出的时间复杂度是 $O(\log n)^a$ ，且获取最大值的时间复杂度是 $O(1)$ ，因此上述问题使用优先队列的时间复杂度是 $O(n \log n)$ ，空间复杂度是 $O(n)$ ，非常高效。

而对于 `set`、`map` 等容器，它们是用红黑树实现的，天然有序：简单地说，无论如何它们都会把所有元素排好（而优先队列并不会把所有元素排好，它只会把最大值放在最前面！）。上述问题中，每一次打架都会改变武士的体力值，这就意味着每次打架后都需要重新排序；为了保持有序，必须先删除、再插入，这个操作本身是两个 $O(\log n)$ 的操作。而在更极端的情况下，加入胜者的体力值极低，它可能从队列首一直沉到队列尾，而 `set` 们仍然保留这个没什么竞争力的数据——这意味着后面每一次取“当前最大值”时，都会把这条记录再比较一遍，白白浪费 $n \log n$ 的时间！上述问题中，我们知道 n 是百万级别的，这种反复比较的额外开销总归是需要让复杂度爆炸的，或者说 $O(n^2 \log n)$ 的复杂度，慢了一百万倍。

而对于 `vector` 而言，每次重新排序则更加直观：一次排序就得 $O(n \log n)$ ，而每次打架后都需要重新排序，这就意味着每次打架后都需要 $O(n \log n)$ 的时间复杂度。这样一来，整个问题的时间复杂度就变成了 $O(n^2 \log n)$ ，显然超时。综上所述，只有保留全局极值但是不必保留元素具体顺序的数据结构才能较好地完成了这个问题，而优先队列正是这样一个数据结构^b。

^a对于不熟悉算法分析的同学们，以上表示可以通俗地理解为：问题规模是 n 与问题规模是 1 的时候相比，执行时间最坏情况下大概变为大 O 里面的函数倍。

^b这个题其实有更快的手段，例如胜者树、败者树等，它们本质上是二叉堆的工业级优化，时间复杂度都是 $O(n \log n)$ ，但是常数应该会更小。但是它们写起来非常困难，要考虑各种诸如淘汰等的边界情况，且需要相当的算法基础。胜者树/败者树在竞赛或工程里通常服务于多路归并这类需要“反复取最小/最大并立刻替换”的场景；而本题只需要“全局最大”，STL 的堆已经够用而且很简洁，杀鸡焉用牛刀。我们这里就不讲了。

以上，就是 C++ 的全部内容了（也不是全部内容，毕竟 C++20、C++23 等版本有越来越多的新特性，但是能掌握 C++17 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。

练习：改错题

以下代码试图使用 C++ 的 OOP、泛型、STL 等特性实现一些功能，但存在未定义行为、逻辑错误、巨大的性能问题或输出违背预期等问题。请找出原因并修正这些问题。

```
1 #include <vector>
2 int main(){
3     std::vector<int> v = {1,2,3,4,5};
4     auto it = v.begin();    // 这是什么类型?
5     it += 3;                // 想跳到 v[3]
6     int *p = it;            // 这是什么?
7 }
```

```
1 #include <set>
2 int main(){
3     std::set<int> s = {1,2,3,4,5};
4     for (auto x : s)
5         if (x % 2 == 0) s.erase(x);    // 想删偶数，但会出问题
6 }
```

```
1 // foo.hpp
2 template<class T>
3 T pi() { return T(3.14); }
4
5 // foo.cpp
6 template<>
7 double pi<double>() { return 3.1415926; }    // 没法编译
```

```
1 #include <iostream>
2 class Base{
3     Base() { foo(); }                // 想调到派生类实现
4     virtual void foo() { std::cout << "base\n"; }
5 };
6 class Der : Base{
7     void foo() override { std::cout << "der\n"; }
8 };
9 int main(){ Der d; }    // 但是还是输出 base?
```

```
1 #include <memory>
2 class Base{ ~Base() { /* 非虚 */ } };
3 class Der : Base{ int* p = new int; ~Der(){ delete p; } };
4 int main(){
5     std::unique_ptr<Base> pb = std::make_unique<Der>();
6 }    // 电脑卡爆，一看任务管理器内存占用飙升?
```

```
1 #include <algorithm>
2 #include <vector>
3 int main(){
4     std::vector<int> v = {1,2,3,2,4};
5     std::remove(v.begin(), v.end(), 2);    // 想删掉所有 2
6     for (int x : v) std::cout << x;        // 但是怎么还打印 5 个数?
```

```
7 }  
  
1 #include <vector>  
2 void toggle(bool& r){ r = !r; }  
3 int main(){  
4     std::vector<bool> vb = {true, false};  
5     toggle(vb[0]);           // 想把 vb[0] 取反  
6 }// 为什么我编译不过?
```

答案

以下是各代码段的问题和修正方法：

1. 迭代器的加减应该用 `std::advance(it, 3);`。另外，迭代器不能直接转换为指针，应该使用 `&(*it)`；来获取指向元素的指针。
2. 在遍历集合（其他容器也一样）时，不能添加或删除元素，因为这会使迭代器失效。因此要用 STL 提供的 `remove_if` 函数，或者先收集要删除的元素，再统一删除。
3. 模板的显式实例化应该放在头文件中，或者在使用模板的文件中进行实例化。
4. 在构造函数中调用虚函数会调用基类的版本，而不是派生类的版本。可以将虚函数调用移到构造函数之外，例如在派生类的构造函数中调用。
5. 基类的析构函数应该是虚的，以确保派生类的析构函数被正确调用，从而避免内存泄漏。
6. `std::remove` 并不会改变容器的大小，它只是将要删除的元素移动到容器的末尾。需要使用 `v.erase(std::remove(v.begin(), v.end(), 2), v.end());` 来真正删除这些元素。
7. `std::vector<bool>` 是一个特殊的容器，它并不存储实际的 `bool` 值，而是使用位来表示布尔值。因此，不能直接获取对 `bool` 值的引用。可以使用 `std::vector<char>` 或 `std::vector<int>` 来代替。

第十一章 现代 C++ 特性

我们先前已经了解了 C++ 的基本语法，以及 OOP、泛型的一些概念。而这些还远远没有触及到现代 C++（C++17、20、23）的强大特性，而这些特性可以极大地提升我们的编程效率和代码质量。

11.1 string_view 和 span：轻量级视图类型

“视图”，顾名思义，就是对数据的一种“看法”或“表示”，它并不拥有数据本身，而是引用现有的数据。这种方式可以避免不必要的数据拷贝，从而提高性能。现代 C++ 引入了两种重要的视图类型：字符串视图（std::string_view）和数组视图（std::span）。

11.1.1 字符串视图

字符串视图是 C++17 引入的一个重要特性，定义在<string_view>头文件中。字符串视图（std::string_view）是一种轻量级的字符串表示方式，它并不拥有字符串数据，而是引用现有的字符串数据。这使得我们可以避免不必要的字符串拷贝，从而提高性能。也就是说：

```
1 #include <string_view>
2
3 std::string str1 = "Hello, World!";
4 std::string_view str2 = "Hello, World!";
5 std::string_view str3 = str1; // 引用 str1 的数据
```

上述str1是 C++11 风格的字符串。而str2和str3则是字符串视图，它们并不拥有字符串数据，而是引用了现有的字符串数据（分别是字面值字符串和str1）。这样我们就可以避免不必要的字符串拷贝，从而提高性能。实际上这也是很“惰性”的一种方式。

需要注意的是，字符串视图并不管理其引用的字符串数据的生命周期，因此在使用字符串视图时需要确保其引用的数据在字符串视图的生命周期内是有效的。例如：

```
1 std::string_view get_substring() {
2     std::string str = "Hello, World!";
3     return std::string_view(str.c_str(), 5); // 错误! str 在函数结束后被销毁
4 }
```

上述代码中, `get_substring`函数返回了一个字符串视图, 但它引用的字符串数据`str`在函数结束后被销毁, 因此返回的字符串视图将变得无效。正确的做法是确保引用的数据在字符串视图的生命周期内是有效的, 例如:

```
1 std::string global_str = "Hello, World!";
2 std::string_view get_substring() {
3     return std::string_view(global_str.c_str(), 5); // 正确, global_str 在函数外部
4 }
```

上述代码中, `global_str`是一个全局变量, 其生命周期贯穿整个程序运行, 因此返回的字符串视图是有效的。所以说实际上不能彻底抛弃 C++11 的字符串类型, 但在很多情况下, 使用字符串视图可以显著提升性能和代码简洁度。

而直接引用字面值的字符串视图则不会有这个问题, 因为字面值字符串的生命周期贯穿整个程序运行。

另一方面, 字符串视图是只读的, 我们不能通过字符串视图来修改其引用的字符串数据。如果需要修改字符串数据, 仍然需要使用 C++11 的字符串类型。但这也有不少好处, 例如字符串视图提供了大量方便的成员函数, 用于字符串的查找、比较、子串提取等操作。例如:

```
1 std::string_view str = "Hello, World!";
2 std::cout << str.substr(0, 5); // 输出 "Hello"
3 std::cout << str.find("World"); // 输出 7
```

上述代码中, 我们使用了字符串视图的`substr`和`find`成员函数来提取子串和查找子串的位置。这些成员函数的使用方式与 C++11 的字符串类型类似, 但由于字符串视图不拥有数据, 因此这些操作通常更高效。

11.1.2 `span`

C++20 引入了`std::span`, 它是一种轻量级的数组视图, 定义在`<span>`头文件中。与字符串视图类似, `std::span`并不拥有数据, 而是引用现有的数组或容器的数据。这使得我们可以方便地操作数组数据, 而无需进行拷贝。

这句话是不是很熟悉? 没错, 实际上这就是字符串视图的推广版本, 可以用于任何类型的数组或容器。例如:

```
1 #include <span>
2
3 std::vector<int> vec = {1, 2, 3, 4, 5};
4 std::span<int> s = vec; // 引用 vec 的数据
5 for (int x : s) {
6     std::cout << x << " ";
7 }
```

上述代码中, `s`是一个`std::span<int>`, 它引用了`vec`的数据。我们可以像操作普通数组一样操作`s`, 而无需进行拷贝。当然, 这样写也有风险, 因为如果原始容器被销毁或修改, `span`将

变得无效。因此，在使用 `span` 时，需要确保其引用的数据的生命周期长于 `span` 本身。自然，字面值数组的生命周期贯穿整个程序运行，因此直接引用字面值数组的 `span` 是安全的。

`std::span` 还提供了一些方便的成员函数，用于获取子视图、大小等操作。例如：

```
1 std::vector<int> vec = {1, 2, 3, 4, 5};
2 std::span<int> s = vec;
3 std::span<int> sub = s.subspan(1, 3); // 获取从索引 1 开始的 3 个元素
4 for (int x : sub) {
5     std::cout << x << " "; // 输出 2 3 4
6 }
```

上述代码中，我们使用了 `subspan` 成员函数来获取一个子视图，类似于字符串视图的 `substr` 成员函数。

`span` 和 `string_view` 的区别在于，前者可能是可写的（取决于模板参数），而后者始终是只读的。例如：

```
1 std::vector<int> vec = {1, 2, 3, 4, 5};
2 std::span<int> s = vec; // 是可写的
3 s[0] = 10; // 修改了 vec 的数据
4 for (int x : vec) {
5     std::cout << x << " "; // 输出 10 2 3 4 5
6 }
7
8 const std::vector<int> cvec = {1, 2, 3, 4, 5};
9 std::span<const int> cs = cvec; // 是只读的
10 // cs[0] = 10; // 错误！不能修改只读 span
```

11.1.3 和引用的异同

我们知道了 `std::string_view` 和 `std::span` 都是轻量级的视图类型，它们并不拥有数据，而是引用现有的数据。这不禁让人想起 C++11 就有的东西：`std::string&` 和 `T&` 引用。那么它们之间有什么区别和联系呢？

首先，引用本身是一个非常强类型的东西，`T&` 只能对应 `T`；而这两个视图类型则是模板化或弱类型的，`string_view` 可以引用任何符合字符串概念的数据，而 `std::span<T>` 可以引用任何类型的数组或容器的数据。

其次，引用通常假定生命周期是有效的，而视图类型则更明确地要求我们管理其引用的数据的生命周期，确保在视图存在期间数据是有效的。

最终，引用并没有统一的成员函数接口，而视图类型则提供了丰富的成员函数，用于操作和查询数据，例如长度信息、子范围等。

11.2 views 和 ranges：声明式数据处理

11.2.1 从 Rust 说开去

先来看一段 Rust 代码：

```

1 let numbers = vec![1, 2, 3, 4, 5];
2 let doubled: Vec<i32> = numbers.iter()
3   .map(|x| x * 2)
4   .filter(|x| *x > 5)
5   .collect();
6 println!("{:?}", doubled); // 输出 [6, 8, 10]

```

这段代码中，我们首先创建了一个整数向量，然后通过迭代器对其进行映射和过滤操作，最后收集结果。整个过程中没有创建任何中间容器，所有操作都是惰性求值的。所谓**惰性求值**，就是只有在真正需要结果时才进行计算，这可以显著提高性能。而且也能够看出，上述代码非常**声明式**，我们只需要描述我们想要的结果，而不需要关心具体的实现细节。

而旧的 C++ 代码则可能是这样的：

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> doubled;
3 for (int x : numbers)
4   doubled.push_back(x * 2);
5 std::vector<int> filtered;
6 for (int x : doubled) {
7   if (x > 5)
8     filtered.push_back(x);
9 }
10 for (int x : filtered) {
11   std::cout << x << " ";
12 }

```

这段代码是很鲜明的 C++11 风格，使用了多个中间容器来存储映射和过滤的结果，虽然功能正确、语法清晰，但效率较低。另一方面，这段代码是很**命令式**的，我们需要明确地告诉计算机每一步该做什么，而不是描述我们想要的结果——这是现代编程语言要极力避免的。

11.2.2 回到 C++：现代 C++ 的 views 和 ranges

为了解决这种问题，C++20 引入了视图（Views），使得我们可以像 Rust 那样以声明式的方式处理数据序列。

下面是使用 C++20 视图的等效代码：

```

1 #include <ranges> // 需要引入该库，C++20 标准
2
3 std::vector<int> numbers = {1, 2, 3, 4, 5};
4 auto doubled = numbers
5   | std::views::transform([](int x) { return x * 2; })
6   | std::views::filter([](int x) { return x > 5; });
7
8 for (int x : doubled) {
9   std::cout << x << " ";
10 }

```

可以看出，这些视图实际上定义在<ranges>头文件中。与 Rust 的迭代器类似，C++ 的视图允许我们以声明式的方式对数据进行转换和过滤，我们使用了transform和filter两个视

图操作来分别进行映射和过滤操作，而传递方式则是通过管道符号|连接的。视图也是惰性求值的，只有在我们真正迭代它们时，才会进行计算。上文的doubled变量实际上是一个惰性的视图，而不是一个具体的容器。

那 ranges 哪去了？上文中根本没有任何 ranges 的影子。实际上，ranges 是 C++20 中更广泛的概念，它定义了一种统一的方式来表示和操作数据序列。视图实际上是 ranges 的一种特殊形式，专注于惰性求值和转换操作。而 ranges 则可以包括具体的容器、数组等。换言之，实际上numbers变量本身就是一个 range，因为它是一个容器，可以被视为一个数据序列。这或许就是“太上，下知有之（ranges），其次亲而誉之（views）”的意思吧。

11.2.3 C++ 视图的基本操作

C++ 的视图基本操作如下：

- **all**：表示整个序列。
- **transform**：对每个元素应用一个函数，类似于映射操作。
- **filter**：根据一个谓词函数过滤元素。
- **take**：获取前 N 个元素。
- **drop**：跳过前 N 个元素。
- **join**：将嵌套的序列展开成一个平坦的序列。
- **reverse**：反转序列中的元素顺序。
- **split**：将序列按指定分隔符拆分成多个子序列。
- **unique**：移除序列中连续重复的元素。
- **sort**：对序列中的元素进行排序。

使用视图的方式，则是通过管道符号，把多个视图操作连接起来，形成一个处理链条，就如上文所示。而这个“谓词函数”则是一个接受元素并返回布尔值的函数，用于决定是否保留该元素。

视图这个东西非常灵活，几乎想做点什么都可以做到。比如说，饱受诟病的“逗号分隔的整数”问题，我们可以用视图来轻松解决：

```

1 #include <string>
2 #include <ranges>
3
4 std::string str{"1,2,3,4,5"}; // 假装这是输入，实际上 getline 一下就好了
5 auto numbers = str
6     | std::views::split(',') // 按逗号拆分
7     | std::views::transform([](auto&& part) {
8         return std::stoi(std::string(part.begin(), part.end()));
9     }) // 转换为整数
10    | std::ranges::to<std::vector<int>>>(); // 收集到 vector 中
11 // 现在 numbers 的类型是 std::vector<int>，包含了整数 1 到 5

```

实际上在这里的 flat 也是一个“视图”，它并没有创建一个新的容器，而是提供了一种方式来迭代嵌套容器中的所有元素。这样我们就可以避免创建额外的中间容器，从而提高性能。换句话说，在不创建视图或中间容器的情况下，我们只能通过嵌套循环来访问这些元素，而使

用视图则可以让代码更简洁、更高效。

视图有着**不可变性**，也就是说，无论怎样操作一个视图，都不会改变其引用的原始数据，这使得我们可以极为放心的使用数据。

11.2.4 ranges 扒开了说

前文提到，视图实际上是ranges的一种特殊形式。那么ranges到底是什么呢？简单来说，这是一种数据序列：由begin迭代器和end哨兵指定的一组可以遍历的项目。所有的 STL 容器都是ranges。

哨兵的类型可以和迭代器的类型不同，这使得我们可以更灵活地定义数据序列。例如，一个字符串的begin迭代器可能是一个指向字符的指针，而end哨兵则可能是一个表示字符串结尾的特殊值（\0）。

ranges可以自定义，我们只需要定义begin和end哨兵即可。例如，我想定义一个非负整数的 ranges，end哨兵为-1（数值类型的哨兵）：

```

1 class NonNegativeRange {
2     public:
3         class Iterator {
4             public:
5                 Iterator(int value) : value(value) {}
6                 int operator*() const { return value; }
7                 Iterator& operator++() { ++value; return *this; }
8                 bool operator!=(const Iterator& other) const {
9                     return value != other.value;
10                }
11                private:
12                    int value;
13            };
14            NonNegativeRange(int start) : start(start) {}
15            Iterator begin() const { return Iterator(start); }
16            Iterator end() const { return Iterator(-1); } // -1 作为哨兵
17            private:
18                int start;
19        };
20
21        NonNegativeRange range(0);
22        for (int x : range) {
23            if (x == -1) break; // 遇到哨兵停止
24            std::cout << x << " ";
25        }

```

上述代码中，我们定义了一个NonNegativeRange类，它表示从指定起始值开始的非负整数序列，直到遇到哨兵-1为止。我们定义了一个嵌套的Iterator类，用于实现迭代器的功能。通过定义begin和end方法，我们使得NonNegativeRange类成为一个可迭代的 ranges。

在上文代码（C++20 视图示例）中，numbers变量本身就是一个 ranges，因为它是一个容器，可以被视为一个数据序列。于是我们可以用视图对其进行转换和过滤操作。

11.2.5 投影

在使用视图时，我们经常需要对数据进行某种形式的转换，这就是所谓的“投影”（Projection）。投影实际上就是将数据从一种形式转换为另一种形式，通常是通过一个函数来实现的。例如，假设我们有一个包含学生信息的结构体，我们想要提取学生的姓名列表：

```

1 struct Student {
2     std::string name;
3     int age;
4 };
5 std::vector<Student> students = {
6     {"Alice", 20}, {"Bob", 22}, {"Carol", 21}
7 };
8 auto names = students
9     | std::views::transform([](const Student& s) { return s.name; });
10 for (const auto& name : names) {
11     std::cout << name << " ";
12 }
```

上述代码中，我们使用了`std::views::transform`视图来提取学生的姓名列表。投影函数接受一个`Student`对象，并返回其姓名。这样，我们就可以轻松地复杂的数据结构中提取所需的信息。

另一个常见的投影是从`std::map`出发：

```

1 std::map<std::string, int> score_map = {
2     {"Alice", 90}, {"Bob", 85}, {"Carol", 88}
3 };
4 auto names = score_map
5     | std::views::keys; // 投影出所有的键（姓名）
6 for (const auto& name : names) {
7     std::cout << name << " ";
8 }
```

上述代码中，我们使用了`std::views::keys`视图来提取`score_map`中的所有键（学生姓名）。这样，我们就可以方便地获取映射中的键列表，而无需手动遍历整个映射。

11.2.6 iota 视图：生成序列

C++20 引入了`std::views::iota`视图，用于通过逐渐增加初始值来创建一个元素序列（有限或无限）。这对于生成整数序列或其他类型的递增序列非常有用。例如：

```

1 auto numbers = std::views::iota(1, 10); // 生成 1 到 9 的整数序列
2 for (int x : numbers) {
3     std::cout << x << " ";
4 }
```

上述代码中，我们使用了`std::views::iota`视图来生成从 1 到 9 的整数序列。无限序列也是可能的：

```

1 auto infinite_numbers = std::views::iota(1); // 生成从 1 开始的无限整数序列
2 for (int x : infinite_numbers | std::views::take(10)) { // 只取前 10 个元素
3     std::cout << x << " ";
4 }

```

`iota`有着一些有趣的特性，例如它可以用于生成自定义类型的序列，只要该类型支持递增操作符（++）。例如：

```

1 struct Point {
2     int x, y;
3     Point& operator++() { ++x; ++y; return *this; } // 支持递增操作
4 };
5 auto points = std::views::iota(Point{0, 0}, Point{5, 5}); // 生成 Point 序列
6 for (const auto& p : points) {
7     std::cout << "(" << p.x << ", " << p.y << ") ";
8 }

```

那么怎样生成一个奇数序列？实际上，我们不要过于将思维过于局限在“一次输出”上，毕竟这东西是惰性的，无需太过担心性能。我们可以结合使用*iota*视图和*transform*视图来实现：

```

1 auto odd_numbers = std::views::iota(0)
2 | std::views::transform([](int x) { return x * 2 + 1; });
3 for (int x : odd_numbers | std::views::take(10)) { // 只取前 10 个奇数
4     std::cout << x << " ";
5 }

```

`iota`也是懒惰的，故而可以生成无限序列，而不会导致内存溢出（只要我们使用*take*等视图来限制输出的元素数量即可）。

11.3 optional、variant 和 any：类型安全的容器

现代 C++ 引入了三种类型安全的容器：`std::optional`、`std::variant`和`std::any`，它们分别用于表示可能缺失的值、多种类型的值以及任意类型的值。这三个容器主要用于安全性和灵活性的提升，避免了传统 C++ 中使用裸指针或`void*`带来的类型不安全问题。

例如，某方法可能返回一个整数值，或者什么都不返回（表示失败或无结果）。在传统 C++ 中，我们可能会使用指针或异常来表示这种情况，但这可能导致内存泄漏或类型不安全问题。而使用`std::optional<int>`则可以更安全地表示这种情况：

```

1 #include <optional>
2
3 std::optional<int> find_value(int key) {
4     if (key == 42) {
5         return 100; // 找到值
6     } else {
7         return std::nullopt; // 未找到值
8     }
9 }

```

```

8     }
9 }
10 auto result = find_value(42);
11 if (result) {
12     std::cout << "Found: " << *result << std::endl;
13 } else {
14     std::cout << "Not found" << std::endl;
15 }

```

上述代码中，`find_value`函数返回一个`std::optional<int>`，表示可能存在的整数值。调用者可以通过检查返回值是否有值来决定如何处理结果。

而后两个容器则分别用于更复杂的场景，例如：

```

1 #include <variant>
2
3 std::variant<int, std::string> get_data(bool flag) {
4     if (flag) {
5         return 42; // 返回整数
6     } else {
7         return "Hello"; // 返回字符串
8     }
9 }
10 auto data = get_data(true);
11 if (std::holds_alternative<int>(data)) {
12     std::cout << "Integer: " << std::get<int>(data) << std::endl;
13 } else {
14     std::cout << "String: " << std::get<std::string>(data) << std::endl;
15 }

```

上文中使用了`std::holds_alternative`和`std::get`来检查和获取值。

至于`std::any`，它用于存储任意类型的值，但需要注意的是，使用`std::any`时需要行类型转换，可能会带来一些性能开销和类型安全问题，因此应谨慎使用。

```

1 #include <any>
2
3 std::any Anything;
4
5 Anything = 42; // 存储整数
6 std::cout << std::any_cast<int>(Anything) << std::endl; // 输出 42
7 Anything = std::string("Hello"); // 存储字符串
8 std::cout << std::any_cast<std::string>(Anything) << std::endl; // 输出 Hello

```

上述代码中，我们使用了`std::any`来存储不同类型的值，并通过`std::any_cast`进行类型转换以获取存储的值。Python 程序员看到这个估计觉得跟回家了一样，但需要注意，这和 Python 的动态类型依然有着本质的差别：Python 的动态类型指的是在不同的时刻，一个变量的类型可以是不同的；而 C++ 的`std::any`则是一个容器，换句话说，`any`类型的对象，它的类型永远是 `any`！这两者的区别还是很大的，比如不能直接试图`cout`一个`std::any`对象，因为编译器并不知道它里面装的是什么类型的数据。而 Python 中就可以随意打印一个动态类型的对象（只要它实现了`__str__`方法）。

11.4 模块

模块旨在改变传统的 C++ 编译模型。

我们写程序，上来第一行几乎都是#include。这个预编译指令实际上做的是：把被包含文件的内容直接插入到当前文件中，然后再进行编译。这种方式有几个问题：

- **编译时间长**：每次编译时，编译器都需要重新处理所有包含的头文件，导致编译时间显著增加。
- **命名冲突**：不同的头文件可能定义了相同的名称，导致命名冲突和难以调试的问题。
- **依赖管理复杂**：头文件之间的依赖关系可能非常复杂，导致编译顺序难以管理。

模块通过引入一个新的编译单元概念，允许我们将代码划分为独立的模块，每个模块可以有自己的接口和实现。这样，编译器只需要处理模块的接口，而不需要重新处理整个头文件，从而显著减少编译时间。

换句话说，我非常抵制的写法：

```
1 #include <bits/stdc++.h> // 包含所有标准库头文件
```

在模块化的 C++ 中，可以被替代为：

```
1 import std; // 导入整个标准库模块
```

后者至少比前者要好很多（虽然我依然不建议这么写）。

模块的定义和使用涉及到一些新的语法和概念，例如：

- **模块接口单元**：定义模块的接口，包含导出的声明。
- **模块实现单元**：包含模块的实现代码。
- **导入模块**：使用import关键字导入模块。

实际上 C++ 的模块化到现在依然在发展中，很多编译器对模块的支持还不完善，因此在实际项目中使用模块时需要谨慎，并确保所使用的编译器版本支持所需的模块特性。而且现在很多现有的 C++ 代码库实际上还在老老实实在地使用传统的头文件包含方式，因此在引入模块化时需要考虑与现有代码的兼容性问题，我个人也建议同学们继续老老实实#include。

11.5 三相比较运算符

C++20 引入了三相比较运算符 (<=>)，也称为“太空船运算符” (Spaceship Operator)。它提供了一种统一的方式来实现对象的比较操作，简化了比较运算符的定义。

传统上，我们需要分别定义多个比较运算符（如<、<=、>、>=、==和!=）来实现对象的比较，这可能导致代码冗长且容易出错。而使用三相比较运算符，我们只需要定义一个运算符，就可以自动生成其他比较运算符。例如：

```
1 #include <compare>
2
```

```

3 struct Point {
4     int x, y;
5     auto operator<=>(const Point& other) const = default; // 使用默认比较
6 };

```

那么什么是“默认比较”呢？其比较顺序是，先比较x，如果相等则比较y。这样，我们就可以通过定义三相比较运算符来实现对象的比较，而不需要手动定义所有的比较运算符。

而这个运算符的返回类型是一个特殊的类型，称为“比较类别”（Comparison Category），它表示比较的结果。C++20 定义了几种比较类别：

- **std::strong_ordering**：表示强排序，支持所有比较运算符。
- **std::weak_ordering**：表示弱排序，允许相等的元素。
- **std::partial_ordering**：表示部分排序，允许无法比较的元素。

通过使用三相比较运算符，我们可以简化对象的比较操作，提高代码的可读性和维护性。例如：

```

1 struct Person {
2     std::string name;
3     int age;
4     auto operator<=>(const Person& other) const {
5         if (auto cmp = name <=> other.name; cmp != 0) {
6             return cmp; // 先比较姓名
7         }
8         return age <=> other.age; // 再比较年龄
9     }
10 };

```

上述代码中，我们定义了一个Person结构体，并实现了三相比较运算符。比较时，先比较姓名，如果姓名相等，则比较年龄。这样，我们就可以通过定义一个运算符来实现复杂的比较逻辑，而不需要手动定义所有的比较运算符。

但是这样实际上并没有省下很多代码量，而且不是很容易看懂，所以大家还是不要写这个了。这个的唯一好处就是省事（例如定义简单的结构体时），但缺点是可读性差，而且不够灵活（例如需要自定义比较逻辑时）。

11.6 std::concept

概念（Concepts）是 C++20 引入的一种语言特性，用于定义模板参数的约束条件。它们允许我们在编译时检查模板参数是否满足特定的要求，从而提高代码的可读性和可维护性。

这确实是一个非常有用的特性，尤其是在编写泛型代码时。通过使用概念，我们可以明确地表达模板参数的预期行为和属性，从而避免在编译时出现难以理解的错误消息。例如，假设我们想定义一个函数，该函数接受一个容器，并计算其元素的总和。我们可以使用概念来约束模板参数，确保它是一个可迭代的容器：

```

1 #include <concepts>
2 template <typename T>
3 requires std::ranges::range<T> // 约束 T 必须是一个范围 (range)

```

```
4 auto sum(const T& container) {  
5 // 具体代码略，但用到了 range-for  
6 }
```

上述代码中，我们使用了`std::ranges::range`概念来约束模板参数`T`，确保它是一个范围（range）。这样，我们就可以在编译时检查传入的容器是否满足该要求，从而避免运行时错误。

概念有很多，基本上都是围绕 STL 容器和算法设计的，例如`std::integral`（整数类型）、`std::floating_point`（浮点类型）、`std::sortable`（可排序类型）等。通过使用这些概念，我们可以更清晰地表达模板参数的预期行为，从而提高代码的可读性和可维护性。具体有哪些我也不能完全记住，同学们自行查阅相关文档即可。

11.7 std::format

这个很重要，而且很常用，比前面两个全是未来（模块）、不知所谓（三相比较运算符）的特性要实用得多。

C++20 引入了`std::format`，它提供了一种类型安全且灵活的字符串格式化方式，类似于 Python 的 f-string 或 JavaScript 的模板字符串。

```
1 #include <format>  
2  
3 std::string name = "Alice";  
4 int age = 30;  
5 std::string message = std::format("Name: {}, Age: {}", name, age);  
6 std::cout << message << std::endl; // 输出 "Name: Alice, Age: 30"
```

上述代码中，我们使用了`std::format`函数来格式化字符串。格式字符串中的花括号`{}`表示占位符，后续的参数将依次填充到这些占位符中。这样，我们就可以方便地创建格式化的字符串，而不需要手动拼接字符串。

有的同学们可能会觉得这玩意和`std::printf`如出一辙，但实际上两者有着本质的区别。首先，`std::format`是类型安全的，它会根据传入参数的类型自动进行格式化，而不需要手动指定格式说明符，这避免了类型不匹配的问题。其次，`std::format`支持更丰富的格式化选项，例如对齐、填充、宽度等，使得我们可以更灵活地控制输出格式。例如：

```
1 std::string message = std::format("Name: {:<10}, Age: {:>3}", name, age);  
2 std::cout << message << std::endl; // 输出 "Name: Alice      , Age: 030"
```

上述代码中，我们使用了格式说明符来控制输出的对齐和填充方式。`:<10`表示左对齐并占用 10 个字符宽度，而`:>3`表示右对齐并用 0 填充至 3 个字符宽度。

以下是格式说明符的一些常见选项：

- **对齐**：<（左对齐）、>（右对齐）、^（居中对齐）。
- **填充**：指定填充字符，例如 0 表示用 0 填充。
- **宽度**：指定输出的最小宽度。

- **精度**：对于浮点数，指定小数点后的位数。
- **类型**：指定输出的类型，例如d（十进制整数）、f（浮点数）、s（字符串）等。（这玩意怎么又回来了?）

通过使用`std::format`，我们可以更方便地创建格式化的字符串，提高代码的可读性和维护性。这个特性和 `std::ranges` 也可以方便的结合使用，例如快速将一个数组字符串化：

```

1 #include <format>
2 #include <vector>
3 #include <ranges>
4
5 std::vector<int> numbers = {1, 2, 3, 4, 5};
6 auto joined = numbers
7     | std::views::transform([](int x) { return std::to_string(x); }) // 把每个整数
  ↪   转换为字符串
8     | std::views::join_with(", "); // 用", " 连接字符串
9 std::string result = std::format("{}{}", std::ranges::to<std::string>(joined));
  ↪   // C++23 的 ranges::to

```

相比传统的字符串拼接方式，`std::format`提供了一种更现代、更安全的字符串处理方式，这个确实值得在日常编程中广泛使用。¹

11.8 C++23 简介

C++23 实际上是一次比较小的标准更新，也比较受人诟病，因为没什么有趣的内容。不过我还是简单介绍几个比较重要的特性。

std::expected 这东西是在 C++23 中引入的。实际上和刚刚说到的，C++17 引入、C++20 加强的`std::optional`三兄弟极其类似，但这东西是包含错误或数值的类型。换言之，实际上也可以用`std::variant`来实现类似的功能，但`std::expected`提供了一种更专门化、更类型安全的方式来处理可能失败的操作。于是这样我们就可以这样写：

```

1 #include <expected>
2
3 std::expected<int, std::string> divide(int a, int b) {
4     if (b == 0) {
5         return std::unexpected("Division by zero"); // 返回错误信息
6     }
7     return a / b; // 返回结果
8 }
9 auto result = divide(10, 0);
10 if (result) {
11     std::cout << "Result: " << *result << std::endl;
12 } else {
13     std::cout << "Error: " << result.error() << std::endl;
14 }

```

¹唯一美中不足的是，很多在线测评平台还没有支持这东西！

实际上感觉很无趣，因为这玩意完全可以用`std::optional`实现。但在较新的代码中，此法能够比旧的写法更清晰地表达意图，因此也不失为一种写法。

std::mdspan 这玩意也是 C++23 中引入的。它是一种多维数组视图，定义在`<mdspan>`头文件中。当成`span`用就行了，实际上确实区别不大。

std::print 和 **std::println** 这两个东西真可谓是“千呼万唤始出来”，终于在 C++23 中被引入了。`std::print`和`std::println`提供了一种简洁且类型安全的方式来输出格式化的字符串，类似于 Python 的`print`函数。前者相当于“打印”，后者相当于“打印一行”。

```
1 #include <print>
2
3 std::string name = "Alice";
4 int age = 30;
5 std::print("Name: {}, Age: {}\n", name, age); // 使用 std::print
6 std::println("Name: {}, Age: {}", name, age); // 使用 std::println
```

与`std::format`类似，格式字符串中的花括号`{}`表示占位符，后续的参数将依次填充到这些占位符中。

Python 用户感觉又回家了。

std::ranges::to C++23 引入了`std::ranges::to`，它提供了一种简洁的方式将范围(ranges)转换为具体的容器类型。例如：

```
1 #include <ranges>
2
3 std::vector<int> numbers = {1, 2, 3, 4, 5};
4 auto even_numbers = numbers
5     | std::views::filter([](int x) { return x % 2 == 0; })
6     | std::ranges::to<std::vector>(); // 转换为 std::vector
7 for (int x : even_numbers) {
8     std::cout << x << " "; // 输出 2 4
9 }
```

通过这一方法，我们就可以方便地将范围转换为所需的容器类型，而不需要手动创建和填充容器，这也是非常方便的。

第十二章 C++ 真理大讨论

在上两章中，我们系统地学习了 C++ 的语法和标准库，掌握了 C++ 编程的基本技能。然而，C++ 作为一门复杂且多范式的编程语言，其设计理念、使用方式和最佳实践并非一成不变，而是随着时间和技术的发展不断演进的。因此，在本章中，我们将围绕 C++ 的核心理念和现代 C++ 的最佳实践展开讨论，帮助大家更深入地理解 C++，并学会如何写出真正的现代 C++ 代码。

我们认识一门语言，往往需要回答两个问题：为什么要用它？怎样才能用好它？前者涉及语言的设计理念、历史背景和应用场景，后者则涉及具体的编程技巧和最佳实践。我们将从这两个角度出发，探讨 C++ 和 C 的关系，以及怎么样写出好的 C++ 代码。本章节是一个偏向阅读的章节，读者大可放松心情，细细品味，跳过去也不会对后续章节造成太大影响。

12.1 为什么不抛弃 C？

学完 C++，不少人估计又会问：C++ 这么好，为啥不让 C 直接光荣退休？实际上这句话在技术上听起来确实很爽，但是忽略了语言生态、历史路径和硬件底层几个因素。

首先，在操作系统、启动文件、硬件裸机上，C 是最后一层“可移植的汇编”：Linux 全是 C 内联的汇编，Linux 明确“no C++”；POSIX 标准上，系统 API 全是 C 接口，要保持 C 的 ABI 才能被后续 C++、Python、Rust、Go 等语言调用；Bootloader 等只有 8kBROM，没有 C++ 喜欢的堆，其异常表、RTTI 占空间，体积就是成本。所以说，换 C++ 等于重做整个世界，代价巨大。

另，C 由于其简单性，导致其编译器、工具链、嵌入式支持等生态极其成熟，几乎所有平台都支持 C，保持 C 标准实际上是保持全世界软件互相操作的“插头”继续正常工作。虽然 C++ 也已经将近 50 岁、非常成熟，但因为其重载、模板、命名空间、符号重命名规则复杂，导致其 ABI 不稳定，跨编译器、跨版本互操作性差，无法取代 C 的地位。

又，C 和 C++ 的两个委员会代表了不同的利益：WG14（C 委员会）代表了嵌入式、操作系统、编译器等底层软件的利益，核心诉求是稳定、轻量、ABI，反对 C++ 的特性污染；而 WG21（C++ 委员会）代表了应用软件、游戏、图形等高层软件的利益，核心诉求是抽象、零开销、泛型，反对 C 的限制演进。脑壳更疼的是编译器厂商，他们要维护多重前端，合并了工作量也跟着翻倍。这“分家”实际上是政治和市场的平衡结果，不是技术优劣的简单问题。

最终，现有的 C 代码库数量实在是太大了，估计怎么也已经有了几千亿行。要是随便乱换，那成本也是不可估量的。

现在 C 和 C++ 的委员会官方立场是和平共处、互不阻塞，WG14 明确其“不与 C++ 竞争泛

型和 OOP”，WG21 也承诺“C++ 标准库头文件会和 C 头文件保持一致，不会破坏 C 的 ABI”。所以说，C 和 C++ 各有各的用武之地，互不冲突。

而我之所以要先讲 C 再讲 C++，主要原因是 C++ 中“指针”这个概念几乎完全隐身，其高级抽象掩盖了底层的内存模型和运行机制，如数组越界、函数指针、手动内存分配等在 C++ 中都被 STL、智能指针等抽象掉了，初学者很难理解其底层原理。而 C 语言则直接暴露了这些底层细节，能够帮助初学者更好地理解计算机的运行机制和内存模型，为后续学习 C++ 打下坚实的基础。换句话说，我是为了“裸指针”这碟醋而包了“C 语言语法”这盘菜；也只有知道 C 控制内存有多麻烦，才能体会 C++ 智能指针和 RAII 的真正价值。

要不然……同学，你也不想不知道什么是迭代器吧？

12.2 怎样写出真正的 C++?

很多 C++ 使用者实际上写的是“C with STL”，也就是仅用 C++ 的语法糖和 STL 容器，却没有真正利用 C++ 的面向对象、泛型编程等特性。这样写出来的代码往往冗长、低效、难以维护，无法发挥 C++ 的真正优势。

12.2.1 从实例出发，到实例中去

任务：读取文件内容到字符串，并统计非空行数。我们只要得到结果的函数即可，不需要完整的程序框架。这些代码都对对应着当时主流编译器、标准库、硬件和工程组织的硬约束，不能脱离实际。

C99, 2000

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 size_t count_non_empty_lines(const char* path)
6 {
7     FILE* fp = fopen(path, "r"); // 以读模式打开文件
8     if (!fp) return (size_t)-1; // 打开失败，返回-1 表示错误
9
10    size_t cnt = 0; // 非空行计数器
11    char* line = NULL; // 存储读取的行
12    size_t n = 0; // 行缓冲区大小
13    ssize_t len; // 读取的行长度
14    while ((len = getline(&line, &n, fp)) != -1) {
15        /* 至少有一个非空白字符 */
16        int non_blank = 0;
17        for (ssize_t i = 0; i < len; ++i)
18            if (!isspace((unsigned char)line[i])) { non_blank = 1; break; }
19        cnt += non_blank;
20    }
21    free(line);
22    fclose(fp);
23    return cnt;

```

24 }

这个是很好的 C 代码，使用了标准库函数，处理了错误情况，逻辑清晰：手动管理文件、getline 动态扩容、逐行读取、逐字符检查非空白。

C++03, 2003

```

1 #include <cstdio>
2 #include <vector>
3 #include <string>
4
5 std::size_t count_non_empty_lines(const std::string& path)
6 {
7     FILE* fp = std::fopen(path.c_str(), "r");
8     if (!fp) return -1;
9
10    std::size_t cnt = 0;
11    std::vector<char> buf(1024);
12    while (std::fgets(buf.data(), buf.size(), fp)) {
13        bool blank = true;
14        for (char c : buf) {
15            if (c == '\0') break;
16            if (!std::isspace(static_cast<unsigned char>(c))) { blank = false;
17                ↪ break; }
18        }
19        cnt += !blank;
20    }
21    std::fclose(fp);
22    return cnt;
23 }
```

我们发现，上述代码和 C 代码几乎一模一样，逻辑完全没有变化，错误处理依然是靠返回值，资源管理依然是手动的¹，编程方式也依然是相当面向过程的。

C++11, 2011

```

1 #include <fstream>
2 #include <string>
3 #include <algorithm>
4
5 std::size_t count_non_empty_lines(const std::string& path)
6 {
7     std::ifstream ifs(path, std::ios::ate); // std::ios::ate 表示打开文件后立即将读
8     ↪ 写位置移动到文件末尾，以便获取文件的大小。
9     if (!ifs) return -1;
10
11    std::size_t len = static_cast<std::size_t>(ifs.tellg()); // 获取文件大小
12    ifs.seekg(0); // 把读写位置移动到文件开头
13    std::string content(len, '\0');
14    ifs.read(content.data(), len); // 一次性读入文件内容
15 }
```

¹虽然 ifstream 是 RAII 的，但是这个风格太 C

```

14
15     std::size_t cnt = 0;
16     std::string::iterator bg = content.begin(); // 迭代器指向字符串开头
17     while (bg != content.end()) {
18         auto nl = std::find(bg, content.end(), '\n');
19         if (std::any_of(bg, nl,
20             [](unsigned char c){ return !std::isspace(c); }))
21             ++cnt;
22         bg = (nl == content.end()) ? nl : nl + 1;
23     } // 这段就是找行首和行尾, 中间检查非空白字符
24     return cnt;
25 }

```

这个代码的最大改进之处:

- 一次性读入文件内容, 避免逐行读取的系统调用开销;
- 使用了 STL 的算法 `std::find` 和 `std::any_of`, 代码更简洁, 几乎没有任何手动循环;
- 彻底的 RAII, 资源自动管理。

C++17, 2017

```

1  #include <fstream>
2  #include <string>
3  #include <optional>
4  #include <sstream>
5  #include <algorithm>
6
7  std::optional<std::string> slurp(const std::string& path)
8  {
9      std::ifstream ifs(path, std::ios::binary | std::ios::ate); // 以二进制模式打开
10     ↪ 文件, 并将读写位置移动到文件末尾
11     if (!ifs) return std::nullopt; // 打开失败, 返回空的 optional
12     std::size_t n = ifs.tellg();
13     ifs.seekg(0);
14     std::string s(n, '\0');
15     ifs.read(s.data(), n);
16     return s;
17 }
18
19 std::size_t count_non_empty_lines(const std::string& path)
20 {
21     auto content = slurp(path);
22     if (!content) return -1;
23
24     std::istringstream in(content.value()); // 字符串流
25     return std::count_if(
26         std::istreambuf_iterator<char>(in), // 输入流迭代器
27         std::istreambuf_iterator<char>(),
28         [&in](char mutable // mutable 允许修改捕获的变量
29             {
30                 std::string line;
31                 std::getline(in, line);
32                 return !line.empty() &&
33                     std::any_of(line.begin(), line.end(),
34                         [](unsigned char c){ return !std::isspace(c); });

```

```

34     });
35 }

```

和之前的代码相比，这段代码有以下改进：

- 使用了 `optional` (C++17) 来处理可能失败的文件读取，更加语义化；
- 使用了 `std::istreambuf_iterator` 来逐字符遍历输入流，避免了手动循环；
- 使用了 `std::count_if` 来统计非空行数，代码更加简洁明了。

C++20, 2020

```

1  #include <fstream>
2  #include <sstream>
3  #include <string>
4  #include <optional>
5  #include <ranges>
6  #include <algorithm>
7
8  std::optional<std::string> slurp(const std::string& path)
9  {
10     std::ifstream ifs(path, std::ios::binary | std::ios::ate);
11     if (!ifs) return std::nullopt;
12     auto n = ifs.tellg();
13     ifs.seekg(0);
14     std::string s(n, '\0');
15     ifs.read(s.data(), n);
16     return s;
17 }
18
19 std::size_t count_non_empty_lines(const std::string& path)
20 {
21     auto content = slurp(path);
22     if (!content) return -1;
23
24     auto lines = std::views::split(content.value(), '\n') // 分割成行
25     | std::views::filter([](auto&& line) { // 过滤这些行:
26         return !std::ranges::empty(line) && // 非空行
27             !std::ranges::all_of(line, // 且不全是空白字符
28                 [](unsigned char c){ return std::isspace(c); });
29     });
30     return std::ranges::distance(lines); // 计算非空行数
31 }

```

这个风格更是极端：

- 全程没有显式循环；
- 用管道操作符组合逻辑；
- 声明式风格；
- 类型安全，零拷贝。

我们看出，以上五段代码，虽然功能完全相同，但是风格和质量却有天壤之别，以 C++11 为显著分界线，区别主要有五：

- 资源管理：从手动管理到 C++11 RAII；

- 错误处理：从返回值检查到 C++17 optional 等类型；
- 算法和数据结构：从手动循环到 C++11 STL 算法和迭代器，再到 C++20 ranges；
- 内存策略：从逐行读取到 C++11 一次性 slurp，再到 C++20 的零拷贝视图；
- 可读性和可组合性：从命令式到声明式。

12.2.2 现代 C++ 的思维方式

如果一个代码出现了以下特征，那么它很可能是“C with STL”：

- 使用裸指针和手动内存管理。
- 命令式、过程式，许多循环和条件判断。
- 缺乏抽象，没有利用面向对象的特性。
- 逻辑静态，重载一堆，难以复用。
- 忽略异常安全，在异常情况下容易导致资源泄漏。

而真正的现代 C++ 思维方式是其反面，从上到下分别对应着：

- RAII；
- 面向对象，甚至多范式结合；
- 泛型编程；
- 元编程；
- 异常安全。

这些思想在 C++11 就能很好写出来，而后续两次大更新（C++17 和 C++20）则进一步提升了代码的简洁性和表达力，使得 C++ 代码能够更加接近声明式编程的风格。比如说，我们即使是在 C++11，也可以问自己以下问题：

- RAII：有没有用智能指针、容器等自动管理资源？
- 泛型：有没有用模板、Lambda 函数等泛型编程手段？
- STL：有没有坚持使用 copy_if、accumulate 等 STL 算法代替手动循环？
- 异常安全：有没有考虑异常情况下的资源管理？（C++11 只能自己写一个 optional，但这玩意太好用了）
- 性能（尤其是拷贝）：有没有用 std::move、完美转发、emplace 等手段减少不必要的拷贝？
- 可读性：有没有用 auto、范围 for（感觉不如 std::for_each）等手段提升代码可读性？有没有用别名、命名空间等手段提升代码可维护性？

所以说，我们写代码的时候，尽可能利用 C++ 的特性，把 C++ 当成一种思维方式，而非仅用来抽象 C 的语法糖，这样才能写出真正的 C++ 代码。

例如，现有一个 vector，要求所有元素都加一，我们的第一反应是什么？我给出四行代码：

```
1 for (size_t i = 0; i < vec.size(); ++i) vec[i] += 1;
2 for (auto& x : vec) x += 1;
3 std::for_each(vec.begin(), vec.end(), [](auto& x) { x += 1; });
4 std::ranges::for_each(vec, [](auto& x) { x += 1; });
```

这四行在功能乃至优化后的汇编上几乎等价，但是它们的思维方式却大相径庭。如果你的第一反应是后面三行当中的一个，那说明你的思维很现代 C++ 化；如果是第一个，那说明你的思维模式还停留在古典 C。

所以说，我们写代码的时候，尽可能利用 C++ 的特性，把 C++ 当成一种思维方式，而非仅用来抽象 C 的语法糖，这样才能写出在当前约束下最优雅、最安全、可维护的代码。

12.3 否定之否定：FakeC++ 是否真正罪大恶极？

12.3.1 再回到实例

任务依旧：把文件搬进内存，统计非空行数。依然是上述五段代码，但这一次请带着历史唯物主义的眼光看待它们，每一层都对应了当时主流编译器、标准库、硬件与工程组织的硬约束。

- **C99, 2000**：当时没有 RAII、没有异常，操作系统一次分配 4kB 页，文件 IO 系统调用开销较大，内存分配开销较大，代码必须一次性 `gcc -std=c99 -pedantic` 通过，要不然进不了仓库（当时甚至连 Git 都没有）。于是我们发现，代码也只能这么写。
- **C++03, 2003**：项目组确实刚刚同意 G++ 也可以提交，但是审查规则依然是禁止模板和异常。老工程师对 C++ 的唯一诉求仅仅是“自动释放内存”，所以大家都只用 STL 容器来替代 C 的 `malloc/free`，其他的编程习惯和 C 几乎没有区别。
- **C++11, 2011**：我们看着一次性读入文件内容很爽，但是在当时很多老派程序员依然是反对这么搞的。能做到这些，有赖于硬件和操作系统的进步：内存带宽超过硬盘带宽，一次性读文件反而成了性能更好、代码更简单的双赢；编译器也开始支持 `auto`、`lambda` 表达式，STL 算法等现代 C++ 特性，且性能优化也足够好了，大家终于敢用这些新特性了。
- **C++17, 2017**：而此时的编译器发展：optional 刚刚进标准，编译器对现代 C++ 的解析也稳定下来。而且硬件也更强了，内存更大了，文件更大了，大家终于开始用字符串流来处理文件内容了。编译器对 STL 算法的优化也更好了，`istreambuf_iterator` 来逐字符遍历输入流也不是什么大问题了。
- **C++20, 2020**：到了这个时候，编译器对 `ranges` 的优化终于做到了“零抽象惩罚”，`split` 视图不产生额外内存，`filter` 视图也不产生额外内存。大家终于敢用这些新特性了，代码简洁到令人发指，完全没有显式循环，逻辑清晰到仿佛在写 SQL 查询语句。而对应的代码审查也要求“禁止手动循环”，否则就要被打回重写。

如果我们能真正理解上述五段代码背后的历史背景和技术约束，我们就会发现两条有趣的曲线。

我们会发现一条单调递减的曲线：运行时开销，C99 如果是 100，那么 C++11 大概是 60，C++20 大概就只有 10 了。

而还有另一条单调递增的曲线：让编译器做的事。2000 年，编译器仅负责帮你翻译成汇编；2011 年，编译器开始帮你做内联、模板实例化、自动内存管理等；2020 年，编译器甚至帮你做了零拷贝视图、管道优化等，临时对象几乎为 0。

这两条曲线，Bjarne 在 1994 年就已经预见到了：

You don't pay for what you don't use, and what you do use you couldn't hand-code any better.

这就是 C++ 的零开销抽象（Zero-Cost Abstraction）理念：抽象本身不应该带来额外的运行时开销，编译器应该能够把高层抽象优化成和手写低层代码一样高效的机器码。这句话立刻可以拆成三条原则：

- 你不使用的特性，不会带来任何开销；
- 你使用的特性，编译器会帮你把它优化到和手写代码一样高效；
- 如果编译器做不到，那就是编译器的错，不是语言设计的错。

于是顺便带来了三条启示：

- **先问约束，再选抽象**：嵌入式 64KB RAM 的环境，`ranges::distance` 还真就不如 `for (char c : container)` 高效；而云端 128vCPU 的大型服务，不写 `ranges` 反而成了性能瓶颈。
- **让错误尽早爆掉**：用 `optional`、断言检查概念、用 `constexpr` 把运行时量变成编译时常量，都是让错误尽早爆掉的手段：如果能在编译期发现错误，就不要等到运行时才发现。
- **零拷贝是默认策略，拷贝必须有理由**：从 `slurp` 到字符串视图，从传值到传引用，本质上说的一个事：把数据移动降级成指针移动，除非 `profiler` 说“不行”，否则不要轻易拷贝数据。

所以说，现在的 C++，以下代码：

```
1 // All necessary includes
2
3 constexpr std::array<int, 5> data = {1, 2, 3, 4, 5};
4
5 void print0() { // C++11 style
6     for (const auto& value : data) {
7         std::cout << value << " ";
8     }
9     std::cout << std::endl;
10 }
11
12 void print1() { // C++11 with std::for_each
13     std::for_each(data.begin(), data.end(), [](int value) {
14         std::cout << value << " ";
15     });
16     std::cout << std::endl;
17 }
18
19 void print2() { // C++20 with ranges
20     std::ranges::for_each(data, [](int value) {
21         std::cout << value << " ";
22     });
23     std::cout << std::endl;
24 }
25
26 void print3() { // C++20 with ranges and views
27     auto view = data | std::views::all;
28     for (const auto& value : view) {
29         std::cout << value << " ";
30     }
```

```
31     std::cout << std::endl;
32 }
```

实际上**编译器激进优化**的情况下，这四者的汇编行数**完全相同**，因为编译器已经把它们都优化成了同样的机器码：51 行，一行不多，一行不少。在更加复杂的情况下，高级抽象可能确实要多几行，但也不会多太多——当然，前提是你正确地使用了这些抽象，而不是滥用或错用。

12.3.2 再回看 Fake C++

于是，我们再把眼光放回到 Fake C++ 中，或者 C++03 时代中的代码：它们从来不是真正的 Fake C++，而是时代约束下的局部最优解。

真正的原罪只有一条：**明明约束条件已经改变，但仍然抱残守缺、固步自封、把旧的局部最优当成新的全局最优**。如果你维护的是老库，那 `vector<char>` 是好的，接着用；但你如果是在 2025 年写新服务，却因为同样的理由拒绝用 `ranges` 甚至拒绝 `string`，那这就是问题了。

现代 C++ 程序员的手里应该始终有一把锤子 (Fake C++)，但更应该有一整套电动工具 (现代 C++ 特性)。什么时候用锤子，什么时候用电钻，什么时候用激光切割机，全凭你对约束的理解和对工具的熟练程度；我们应该知道，从手写循环到 `ranges` 之间，每一级抽象背后的代价和收益；我们更应该知道，在当前需求、硬件、团队、编译器四维空间这一线性规划下的最优解。

把这两套工具永远对立起来，只会让我们停滞不前，错失良机；不管黑猫白猫，抓到老鼠就是好猫，只要能做到以上两点，不管你写的是 20 年前的 `getline`，还是 20 年后的 `std::views::split`，那都是真正的 C++：因为你让编译器在当前约束下，帮你写出了最优雅、最安全、最可维护、代价最低的代码。

12.3.3 相信编译器

而在最后，我想要跟读者强调的一点是，**相信编译器**。

不懂 C++ 的人写一段代码可能是这样写的：

```
1 std::vector<int> foo(int i){
2     std::vector<int> v{};
3     // some code
4     return v;
5 }
```

半懂不懂的人写一段代码可能是这样写的：

```
1 inline ::std::vector<int> foo(const int& i){
2     ::std::vector<int> v{};
3     // some code
4     return std::move(v);
5 }
```

而真正懂 C++ 的人写一段代码可能还是这样写的：

```
1 std::vector<int> foo(int i){
2     std::vector<int> v{};
3     // some code
4     return v;
5 }
```

大家一看：怎么又回到原点了？其实不然，实际上这就是“看山是山”“看山不是山”“看山还是山”的区别。真正懂 C++ 的人知道，现代 C++ 编译器已经足够聪明。我们来看看那个“半懂不懂”的人究竟犯了什么错误：

- `inline` 多余，编译器会自动决定是否内联，一般小函数才需要显式 `inline`。而在激进优化下，所有的函数都是内联的，此举更是多余。
- `const int&` 无用，因为 `int` 本身就是小类型，传引用反而增加了间接寻址开销，更慢。另一方面，这里是传值，肯定不会修改实参，没必要加 `const`（当然这个加了也不会影响性能和正确性）。
- `::std::vector<int>` 多余，全局命名空间前缀没必要，除非你在写头文件并且担心命名冲突，但这肯定不是头文件。
- `std::move(v)` 错误，因为这里的 `v` 是一个局部变量，返回时会触发返回值优化（RVO），编译器会直接在调用者的栈帧上构造返回值，根本不会发生拷贝或移动操作。强制使用 `std::move` 反而会阻止 RVO 的发生，导致不必要的移动开销。

所以，相信编译器，让编译器帮你做它该做的事，而不是试图去“优化”编译器已经优化过的东西，这才是现代 C++ 程序员的正确姿势。只有在 `perf` 出现瓶颈时，我们才需要考虑手动优化，否则请相信编译器的优化能力。

12.4 解剖和验证：ranges 真的免费吗？

在前文中，我们多次提到 C++20 的 `ranges` 特性能够实现“零开销抽象”，并且能够让代码更加简洁和易读。但是，`ranges` 真的完全免费吗？它真的不会带来任何运行时开销吗？

实践是检验真理的唯一标准。为了验证 `ranges` 的性能，我们可以通过一个简单的实验来比较使用 `ranges` 和不使用 `ranges` 的代码在运行时的性能差异。这个我没办法带着同学们做，但是我给出一个实验方案，同学们可以在自己的机器上尝试。

12.4.1 任务

还是喜闻乐见的任务：读取一个大文件，统计其中非空行的数量（总行数约 1 亿行）。

控制变量：在同一台机器、同一个编译器和编译选项上测试，`-O0`、`-O2`、`-O3` 三种优化等级都测试。文件先 `mmap` 进内存，防止磁盘的 IO 干扰。

编译选项（以 `O3` 为例）：

```
1 g++ -std=c++20 -O3 -march=native -DNDEBUG
```

写出四个版本的代码：

- ranges+ 管道（真正零拷贝视图）；
- algorithm + stringview（部分零拷贝）；
- 裸指针 + 手动循环（完全手动）；
- STL 迭代器 + 手动循环（STL 风格）。

测试：用 `perf` 测量每个版本的 `cycles`、`branch-misses`、`cache-misses` 三个指标；总用时可以用 `cpp` 的 `chrono` 库测量，直接按日志打印出来；汇编行数，可以用 `objdump -d` 来查看；源代码行数，可以自己数一下。

对比上述结果，看看 `ranges` 版本和其他版本在性能上和代码简洁性上有什么差异。

12.4.2 预期结果

根据 C++20 的设计理念和编译器的优化能力，我预期结果如下：当内存够大时，在 O3 激进优化面前，`ranges` 的零拷贝视图版本和充分优化的裸指针版本在性能上应该非常接近，差异可以忽略不计，估计不超过裸指针版本的 2%；而在代码简洁性上，`ranges` 版本应该明显优于其他版本，代码行数和可读性都有显著提升。

因此，只要你的约束里不包含‘调试器必须单步进最简汇编’或者‘编译器必须是 2017 年之前’，就请默认用 `ranges`；否则，`perf` 出现 2 % 以上回退时再考虑降级。

同学们也可以用实验验证我的上述断言。

12.4.3 启示

通过这个实验，我们可以得出以下启示：零开销从不是零成本，而是零“不必要的”成本。最终，管它是 `ranges` 还是手动写，`perf` 才是唯一的裁判；只要在当前约束下，编译器能够把高层抽象优化成和手写低层代码一样高效的机器码，那这就是零开销抽象。

12.5 C++ 中的一些良好实践

在 C++ 编程中，良好的代码风格可以提高代码的可读性，而遵守一些基本的编程规范则有助于减少错误和未定义行为。我在上述文本中已经提到过一些基本的代码规范。在这里，我将汇总并补充一些常见的代码规范和最佳实践，供大家参考。

12.5.1 头文件

头文件保护

每个头文件都应该使用头文件保护（Header Guard）来防止重复包含。常见的做法是使用预处理指令 `#ifndef`、`#define` 和 `#endif`。例如：

```
1 #ifndef MY_HEADER_H
2 #define MY_HEADER_H
```

```
3 // 头文件内容
4 #endif // MY_HEADER_H
```

而在工程上，我们更推荐使用 `#pragma once`，它更简洁，大多数编译器都支持：

```
1 #pragma once
2 // 头文件内容
```

使用标准的 C++ 头文件

在引用标准库的时候，应该使用 C++ 标准头文件（如 `<cstdio>`）而不是 C 风格的头文件（如 `<stdio.h>`）。C++ 标准头文件会将内容放在 `std` 命名空间中，避免命名冲突。例如：

```
1 #include <cstdio> // 好的实践
2 #include <stdio.h> // 不推荐
```

仅使用必要的头文件

只包含当前文件实际需要的头文件，避免不必要的依赖和编译时间增加。例如，如果只需要使用 `std::vector`，就只包含 `<vector>`。如果要用 `cin` 和 `cout`，就包含 `<iostream>`。避免一口气引入大量头文件，防止命名冲突和宏污染。

绝对禁止在工程中使用 `#include <bits/stdc++.h>` 这种头文件。该头文件非标准且不可移植；它包含了几乎所有的标准库头文件，导致编译时间增加，并且引入不可忍受的命名冲突和宏污染。

12.5.2 命名空间

在 C++ 中应合理的使用命名空间。避免将所有代码放在全局命名空间中，防止命名冲突。建议为每个库或模块定义独立的命名空间。例如：

```
1 namespace mylib {
2     // 库代码
3 }
```

减少使用匿名命名空间 `namespace {}` 和 `using namespace std;`。在头文件中，严格禁止使用 `using namespace std;`，源文件中应谨慎使用，但这也会引发一些命名冲突问题，建议使用显式的命名空间前缀或仅引入需要的名称：

```
1 using std::cout; // 只引入需要的名称，是好的实践
2 std::cout << ... // 直接使用 std 命名空间下的名称，也是好的实践
```

12.5.3 变量声明和使用

声明即初始化

为防止 UB，变量应尽量做到“声明即初始化”。如果不知道初始化为什么，可以零初始化。例如：

```
1 int x = 0; // 好的实践
2 std::vector<int> vec; // 这会初始化为空 vector，是好的实践
3 std::vector<int> vec = std::vector<int>(10, 0); // 好的实践
4
5 int x; // 不推荐
6 std::vector<int> vec(10); // 不推荐
```

使用局部变量替代全局变量

全局变量容易引发命名冲突和不可预期的副作用，建议尽量使用局部变量或类成员变量来替代全局变量。如果必须使用全局变量，建议使用命名空间封装，并加上适当的前缀以防止命名冲突。例如：

```
1 namespace config {
2     extern int global_setting; // 声明全局变量
3 }
```

多用 **const**、**constexpr** 等修饰符

在 C++ 中，建议尽可能使用 **const** 和 **constexpr** 修饰变量和函数，防止意外修改，提高安全性。在更安全的语言 Rust 中，变量甚至默认是不可变的。例如：

```
1 const int MAX_SIZE = 100; // 好的实践
2 const int square(int x) { return x * x; } // 好的实践
3 Matrix operator+(const Matrix& a, const Matrix& b) { ... } // 好的实践
```

用变量替代魔法数字、宏

避免在代码中使用魔法数字（前不着村后不着店的数字常量），也不建议使用无类型检查的宏定义。建议使用具名常量、枚举类或 **constexpr** 变量来替代魔法数字和宏。例如：

```
1 constexpr int MAX_SIZE = 100; // 好的实践
2
3 for (size_t i = 0; i < MAX_SIZE; ++i) { ... }
```

用强枚举替代传统枚举

避免使用传统枚举（enum），因为它们会将枚举值提升为整型，可能引发命名冲突和隐式类型转换。建议使用强枚举（enum class）来定义枚举类型。例如：

```
1 enum class Color { RED, GREEN, BLUE }; // 好的实践
2 Color c = Color::RED;
```

用 RAII 资源句柄来替代传统裸指针

现代 C++ 最重要的特性之一就是 RAII，我们要使用这个特性来管理资源，避免手动管理内存和资源。建议使用智能指针（如 std::unique_ptr 和 std::shared_ptr）来替代传统的裸指针。例如：

```
1 void foo() {
2     std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>(); // 好的实践
3     return; // ptr 会自动释放内存
4 }
```

使用 C++ 风格的显式类型转换

避免隐式类型转换和 C 风格的强制类型转换（(type)value）。建议使用 C++ 风格的显式类型转换（如 static_cast、dynamic_cast、const_cast 和 reinterpret_cast）来提高代码的可读性和安全性。例如：

```
1 int x = static_cast<int>(3.14); // 好的实践
```

12.5.4 函数和类定义

多模板、少重载

避免过度使用函数重载，尤其是当函数参数类型较多时。建议使用模板和默认参数来实现函数的多态性。例如：

```
1 template <typename T>
2 T add(T a, T b) { return a + b; } // 好的实践
```

用 STL 代替自己实现

STL 提供了丰富的容器和算法，建议尽可能使用 STL 来替代自己实现的数据结构和算法，避免重复造轮子；另外 STL 的实现也经过了大量的测试和优化，虽然常数时间复杂度较大但都会被编译器的激进优化抹平，且比自己维护类似数据结构和算法更正确、不易漏掉边界条件。例如：

```
1 std::array<int, 5> arr = {1, 2, 3, 4, 5}; // 好的实践
2 std::for_each(arr.begin(), arr.end(), [](int x) { ... }); // 好的实践
```

用范围 for 循环替代传统 for 循环

范围 for 循环是最推荐的遍历容器的方式，简洁且不易出错，且比 `std::for_each` 更直观。例如：

```
1 for (const auto& item : container) { // 好的实践
2     // 使用 item
3 }
```

函数参数传递的最佳实践

对于小的对象（如基本类型、`std::pair`、`std::tuple` 等），建议传值；对于大的对象（如 `std::vector`、`std::string` 等），建议传引用以避免不必要的拷贝开销。避免传递非 `const` 引用，除非函数确实需要修改参数。例如：

```
1 void process(const std::vector<int>& data) { ... } // 好的实践
```

12.6 工程实践

为了促进同学们对上述 C++ 良好实践的理解和应用，建议大家在实际项目中积极采用这些规范和最佳实践。以下是一些可供学习的选题：

黑白棋：黑白棋是一个简单的棋类游戏：

- 棋盘为 8x8 的方格，初始状态下中间四个格子分别放置两颗黑子和两颗白子。黑棋放在 3C 和 4D，白棋放在 3D 和 4C。
- 两名玩家轮流下棋，每次只能在空格上放置一颗自己的棋子，并且必须至少翻转对方的一颗棋子。
- 翻转规则是：如果新放置的棋子与已有的同色棋子之间夹着一条直线（水平、垂直或对角线）上的对方棋子，那么这些对方棋子都会被翻转为己方颜色。
- 当一方无法下棋时，轮到对方继续下棋；当双方都无法下棋时，游戏结束，计算各自的棋子数量，数量多的一方获胜。

UNO：UNO 是一种流行的纸牌游戏，规则简单。其规则是：

- 每个玩家初始手牌为 7 张，牌堆中有 108 张牌，包括：
 - 数字牌：每种颜色（红、黄、绿、蓝）有数字 0-9 的牌，每个数字有两张（0 除外，只有一张），共计 76 张。
 - 功能牌：每种颜色有“跳过”、“反转”、“加二”三种功能牌，每种功能牌有两张，共计 24 张。

- 万能牌：有“万能牌”和“万能加四”两种，每种有四张，共计 8 张。
- 游戏开始时，翻开牌堆顶的一张牌作为起始牌。
- 玩家轮流出牌，必须出与弃牌堆顶牌颜色相同或数字相同的牌，或者出万能牌。如果无法出牌，则必须从牌堆摸一张牌；如果还无法出牌，则轮到下一个玩家。
- 功能牌的效果如下：
 - 跳过：下一个玩家被跳过。
 - 反转：游戏顺序反转。
 - 加二：下一个玩家摸两张牌并跳过。
 - 万能牌：出牌者可以指定下一张牌的颜色。
 - 万能加四：出牌者可以指定下一张牌的颜色，下一个玩家摸四张牌并跳过。下家此时可以质疑出牌者是否有其他可出的牌，如果质疑成功，则出牌者必须摸四张牌；如果质疑失败，则下家摸六张牌。（但在本实现中可以忽略质疑规则）
- 当一名玩家只剩下一张牌时，必须喊“UNO”；如果被其他玩家发现没有喊，则必须摸两张牌。（在本实现中也可以忽略此规则）
- 当一名玩家出完所有牌时，游戏结束，该玩家获胜。

思考：怎样用命令行交互？能不能写一个 AI 玩家？能不能用现代 C++ 的特性来简化代码？

另：能不能用 DirectX、Qt、OpenGL、UE 等图形库或游戏引擎写出上述两个游戏的图形界面？这些就交由同学们自行发挥了。

第十三章 C 系工程概述

C 和 C++ 虽然古老且写起来不太舒服，但是在今天依然是非常重要的编程语言。C 和 C++ 的性能极高，能够直接操作内存，并且有着丰富的库和工具支持，因此在系统编程、嵌入式开发、游戏开发等领域仍然有着广泛的应用；时至今日，C++ 和 C 在编程语言使用的广泛程度上依然仅次于 Python，分列第二和第三位。因此，学习其开发相关知识依然是非常有必要的。

而工程上，最重要的问题是：如何管理依赖、如何构建项目。C 和 C++ 的包管理和构建系统一直是一个痛点，本文将介绍一些常用的包管理器和构建系统，帮助大家更好地进行 C 和 C++ 的开发。

13.1 C 系包管理¹

2024 学年初期，我被我某节课的助教邀请，帮助他的恋人配置 C++ 开发环境与调包。当时我调了一晚上也没调好，彻底意识到为 C++ 调包是极为痛苦的体验：C++ 包管理做得很烂，和 Python 的包管理难度相比，简直是一个天上一个地下。

最近，这个问题再度被同学们提出。我认为，在很多课程上使用 C++ 编写项目的时候，也逃不开使用包（但是很多课程并不提及怎么装包）；为了防止同学们在使用 C++ 包时遇到不必要的麻烦，我决定写一章来介绍 C++ 的包管理。

当然，每一个包都拥有着自己推荐的安装方式。如果官方文档等有自己推荐的安装方式，建议同学们优先使用官方推荐的方式安装包。另一方面，C++ 的上限极高，但是下限也极低，很多包的文档质量极差，甚至没有文档；因此，本文也会介绍一些通用的安装方式。（当然这种文档很差的包，往往质量也堪忧！）

实际上 C++ 包管理的最好方式是：USE RUST INSTEAD！（大嘘）

13.1.1 C++ 包管理的困境和现状

虽然我们 C++ 有一定的标准协会（ISO C++），但是 C++ 没有官方的运行时（Runtime）。这导致 C++ 的 ABI 随着编译器、版本、编译选项而随地大小变，导致二进制包必须严丝合缝。

简单说来，C++ 装库主要任务有三项：找到头文件、找到库文件、让构建系统知道它们在哪。头文件指的是 C++ 的头文件（.h、.hpp 等），库文件指的是编译好的二进制文件（.so、.dll、

¹本节作者臧炫懿，周乾康修改。

.a 等)。由于二十多年来没有一个官方组织来统一 C++ 的包管理、大家习惯自己编译，因此逐渐形成了两种互相不重叠的技术路线：

- 手动装
- 使用包管理器

13.1.2 手动装

手动装是最原始的方式，要么从源代码编译，要么从二进制包安装。手动装的好处是可以完全控制安装的版本和配置，坏处是需要手动处理依赖关系和路径问题，并且容易出错。不过，其他方式最终还是依赖于手动装的过程，只不过脏活累活有自动化工具帮你做了；另一方面，手动装包也是最通用的方式，在包管理器上没有找到包时，手动装包几乎是唯一的选择。

手动编译

手动编译的过程通常是：先从官网下载源代码，然后编译：

```
1 git clone https://github.com/fmtlib/fmt.git
2 cmake -S fmt -B build -DCMAKE_BUILD_TYPE=Release
   ↪ -DCMAKE_POSITION_INDEPENDENT_CODE=ON
3 cmake --build build
4 sudo cmake --install build --prefix /opt/fmt
```

然后使用 CMake 配置编译器的搜索路径：

```
1 find_package(fmt REQUIRED)
2 target_link_libraries(my_program PRIVATE fmt::fmt)
```

完毕。以上方式是一种最通用的手动安装方式之一，也很容易适用于其他包。需要注意的是，CMake 的 `find_package` 命令会在系统的标准路径下查找包，因此如果你将包安装在非标准路径下，需要手动指定搜索路径等。

头文件库

这种安装方式适宜用于一些小型的库。只需要将头文件放在项目目录（`third_party/`）下，然后在 CMake 中添加头文件搜索路径即可：

```
1 add_subdirectory(third_party/nlohmann_json)
2 target_include_directories(my_program PRIVATE third_party/nlohmann_json)
```

完毕。以上方式是“把依赖当源码”的极限版本，没有 ABI 问题，但是需要手动处理依赖关系和版本问题等。

13.1.3 使用包管理器

包管理器实际上是一种自动化的手动装方式。它们通常会提供一个统一的命令行工具，来管理包的安装、卸载、更新等操作。许多包都可以使用这种方式安装，且包管理器通常会处理依赖关系和路径问题。

系统级包管理器

对于特定的操作系统或开发环境，使用其自带的包管理器是安装 C++ 库最直接的方式。无论是 Linux（如 APT、DNF、Pacman）、macOS（如 Homebrew），还是 Windows 上的 MSYS2 环境都提供了强大的包管理工具。当你的开发环境和目标部署环境一致时，系统级包管理器通常是最佳选择。

以安装 `fmt` 库为例，不同平台上的命令如下：

- **Debian/Ubuntu (APT):**

```
1 sudo apt install libfmt-dev
```

- **Fedora (DNF/YUM):**

```
1 sudo dnf install fmt-devel
```

- **Arch Linux (Pacman):**

```
1 sudo pacman -S fmt
```

- **macOS (Homebrew):**

```
1 brew install fmt
```

- **Windows (MSYS2/MinGW):** 在 MSYS2 环境下，同样可以使用 `pacman`。

```
1 pacman -S mingw-w64-ucrt-x86_64-fmt
```

安装后，这些库通常会被放置在系统的标准路径下（如 `/usr/include` 和 `/usr/lib`）。构建系统（如 CMake 等）通过 `find_package` 命令可以轻松地找到它们，无需额外配置：

```
1 find_package(fmt REQUIRED)
2 target_link_libraries(my_program PRIVATE fmt::fmt)
```

这样装的优势有以下几点：

- 一条命令即可完成安装，自动处理复杂的依赖关系。
- 库文件和头文件被安装到标准位置，构建系统可以自动发现，无需手动配置路径。
- 可以与其他系统软件一同通过包管理器进行更新和维护。
- 发行版仓库中的库经过测试，通常与系统中的其他组件（如编译器）兼容性良好。

这样装的缺点也很明显：

- 仓库中的库版本未必能满足所有需求。如果你的项目需要某个库的最新特性，或者依赖于一个非常特定的旧版本，某些包管理器可能无法满足需求。
- 不同平台、不同发行版的包管理器命令和包名各不相同，往往需要手动核验与对应。
- 不是所有的 C++ 库都被收录到了官方仓库中，尤其是那些比较小众或新兴的库。

vcpkg

vcpkg 是微软开发的 C++ 包管理器。我们建议在 Windows 使用这个包管理器，因为它可以很好地与 Visual Studio 集成。vcpkg 的使用方式非常简单，首先应当安装 vcpkg（过程略）。

然后在项目中添加 vcpkg 的 CMake 配置：（你应当把示例路径替换为你自己安装 vcpkg 的实际路径）

```
1 cmake -S . -B build
   ↪ -DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake
```

下一步声明依赖 vcpkg.json 文件：

```
1 {
2     "name": "demo",
3     "version": "0.1.0",
4     "dependencies": [
5         "fmt",
6         "nlohmann-json"
7     ]
8 }
```

这样，在 CMake 构建文件的时候就会自动下载源码并编译出 *.a 或者 *.so 等文件，并且将它们放在 build 的相关目录下。

vcpkg 的好处是源码优先（可以切换 triplet）、不和系统冲突；坏处是仅能支持 CMake 项目。

Conan

Conan 是一个跨平台的 C++ 包管理器，支持多种构建系统，包括 CMake、Makefile 等。Conan 的使用方式也很简单：

```
1 pip install conan # 安装 Conan
2 conan profile detect --force # 检测当前系统配置并生成配置文件
```

然后在项目中添加 Conan 的配置文件（conanfile.txt）：

```
1 [requires]
2 fmt/9.0.0
3 nlohmann_json/3.11.2
```

```
4
5 [generators]
6 CMakeDeps
7 CMakeToolchain
```

然后构建：

```
1 conan install . --build=missing -s build_type=Release
2 cmake -S . -B build -DCMAKE_TOOLCHAIN_FILE=build/conan_toolchain.cmake
```

Conan 的好处是支持多种构建系统和平台，包括 makefile、meson、bazel 等；二进制包仓库丰富；支持版本锁定等。但是缺点是 conanfile 和 CMakeLists.txt 文件需要手动维护和同步买比较麻烦。

conda-forge

conda-forge 也能管理 C++ 包：

```
1 conda install -c conda-forge fmt nlohmann_json
```

当然 conda 做这个还是有些大材小用了。

讲完了开发的各个方面，现在该讲讲怎么把你的代码构建可执行的文件（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。

13.2 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！

13.2.1 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。

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：

```
1 cmake_minimum_required(VERSION 3.10) # 声明最低 CMake 版本
2 project(hello LANGUAGES CXX) # 声明项目名称和使用的语言
3 add_executable(hello main.cpp) # 声明可执行文件 hello, 源文件为 main.cpp
```

然后构建：

```
1 cmake -B build # 生成构建系统
2 cmake --build build # 构建
3 ./build/hello # 运行可执行文件 (Linux 和 mac)
4 ./build/Debug/hello.exe # 运行可执行文件 (Windows)
```

爽。

多个目录

假设现在有一个目录结构：

```
1 src/
2 - main.cpp
3 - math/
4   - add.cpp
5   - add.hpp
6 - io/
7   - print.cpp
8   - print.hpp
```

根目录 CMake 文件如下：

```
1 cmake_minimum_required(VERSION 3.10)
2 project(multidir)
3
4 add_subdirectory(src) # 告诉 CMake: src 里面还有东西
```

src/CMake 文件如下：

```
1 add_library(math STATIC math/add.cpp) # 声明静态库 math, 源文件为 add.cpp
2 add_library(io STATIC io/print.cpp) # 声明静态库 io, 源文件为 print.cpp
3 add_executable(app main.cpp) # 声明可执行文件 app, 源文件为 main.cpp
4 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。

13.2.2 语法

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：

```
1 add_library(foo STATIC foo.cpp) # 告诉 CMake，这有个库 foo，是静态的
2 target_include_directories(foo PUBLIC include) # foo 库的头文件在 include 目录下
3 target_compile_features(foo PUBLIC cxx_std_20) # foo 库需要 C++20 特性
4 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。

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：

```
1 set(SOURCE_FILES main.cpp math/add.cpp io/print.cpp) # 声明变量 SOURCE_FILES
2 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：

```
1 option(BUILD_TESTS "Build tests" ON) # 声明一个缓存变量 BUILD_TESTS，默认值为 ON
2 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。

```
1 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:

```
1 cmake -B build -G Ninja # 使用 Ninja 生成器
```

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:

```
1 if(BUILD_TESTS) # 如果 BUILD_TESTS 为真
2     enable_testing() # 启用测试
3     add_subdirectory(tests) # 添加 tests 目录
4 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:

```
1 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:

```
1 message(STATUS "This is a status message") # 输出状态信息
```

```
2 message(WARNING "This is a warning message") # 输出警告信息
3 message(ERROR "This is an error message") # 输出错误信息
4 message(FATAL_ERROR "This is a fatal error message") # 输出致命错误信息并终止构建
```

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：

```
1 set(SOURCES main.cpp math/add.cpp io/print.cpp) # 定义一个列表 SOURCES
2 list(APPEND SOURCES io/scan.cpp) # 向列表 SOURCES 添加一个元素
3 list(REMOVE_ITEM SOURCES io/scan.cpp) # 从列表 SOURCES 中移除一个元素
4 list(LENGTH SOURCES length) # 获取列表 SOURCES 的长度, 存储到 length
5 list(GET SOURCES 0 first) # 获取列表 SOURCES 的第一个元素, 存储到 first
6 list(SORT SOURCES) # 对列表 SOURCES 进行排序
7 list(FIND SOURCES main.cpp index) # 查找列表 SOURCES 中元素 main.cpp 的位置, 存储到
  ↪ index
8 list(INSERT SOURCES 0 "io/scan.cpp") # 在列表 SOURCES 的开头插入元素 io/scan.cpp
9 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。

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:

```
1 find_package(fmt REQUIRED) # 查找 fmt 库, REQUIRED 表示必须找到
2 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:

```
1 include(FetchContent) # 引入 FetchContent 模块
2 FetchContent_Declare(
3     googletest
4     GIT_REPOSITORY https://github.com/google/googletest.git
5     GIT_TAG ? # 指定版本, 这里没有指定, 实际应当指定一个
6 )
7 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。

```

1  vcpkg install fmt # 使用 vcpkg 安装 fmt 库
2  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`。

13.2.3 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：

```

1 # 文件名: cross-mingw.cmake
2 # 1. 目标系统
3 set(CMAKE_SYSTEM_NAME Windows)      # 告诉 CMake “我要生成 Windows PE”
4 set(CMAKE_SYSTEM_PROCESSOR i686)     # 目标 CPU
5
6 # 2. 交叉编译器
7 set(CMAKE_C_COMPILER i686-w64-mingw32-gcc)
8 set(CMAKE_CXX_COMPILER i686-w64-mingw32-g++)
9
10 # 3. sysroot / 搜索根目录
11 set(CMAKE_FIND_ROOT_PATH /usr/i686-w64-mingw32)
12
13 # 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
14 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
15 set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
16 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

```

之后编译：

```

1 cmake -B build-win -DCMAKE_TOOLCHAIN_FILE=cross-mingw.cmake
2 cmake --build build-win
3 ./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。

13.2.4 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：

```
1 install(TARGETS foo EXPORT fooTargets # 安装 foo 目标，并导出为 fooTargets
2     RUNTIME DESTINATION bin # 可执行文件安装到 bin 目录
3     LIBRARY DESTINATION lib # 动态库安装到 lib 目录
4     ARCHIVE DESTINATION lib # 静态库安装到 lib 目录
5 )
6 install(DIRECTORY include/ DESTINATION include) # 头文件安装到 include 目录
```

导出功能使用 `export` 命令：

```
1 install(EXPORT fooTargets
2     FILE foo-config.cmake
3     NAMESPACE foo::
4     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：

```
1 set(CPACK_GENERATOR "DEB")
2 set(CPACK_PACKAGE_NAME "mylib")
3 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：

```
1 cmake --build build --target package # 构建并打包
```

13.2.5 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：

```
1 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。

13.3 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：

```
1 winget install xmake # Windows
2 apt install xmake # Ubuntu
3 brew install xmake # macOS
```

13.3.1 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：

```
1 xmake create -l c++ hello # 创建一个 C++ 项目 hello
2 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：

```
1 hello/
2 - xmake.lua # 构建文件
3 - src/
4   - 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：

CMake 官方说明文档: <https://cmake.org/cmake/help/latest/guide/tutorial/>

```
1 cd hello # 进入 hello 目录
2 xmake # 构建
3 xmake run # 运行
4 xmake run -d # 调试运行
```

13.3.2 语法速通

最小示例

```
1 -- xmake.lua
2 set_project("hello")
3 set_version("1.0.0")
4 add_rules("mode.debug", "mode.release") -- 自动生成 debug/release 配置
5
6 target("app") -- 一个目标
7     set_kind("binary") -- 可执行文件
8     add_files("src/*.cpp") -- 通配符
9     set_languages("c++20") -- 标准
10    add_packages("fmt") -- 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。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。

语法速查（和 CMake 对比）

CMake	XMake	说明
add_executable	target("app"), set_kind("binary")	声明一个可执行文件或库
add_library	target("lib"), set_kind("static")	声明一个静态库或动态库
target_sources	add_files("src/*.cpp")	添加源文件
target_include_directories	add_includedirs("include")	添加头文件目录
target_compile_definitions	add_defines("FOO")	添加编译选项
target_compile_options	set_cxflags("-Wall")	添加编译选项
target_link_libraries	add_links("bar")	添加链接库
find_package	add_packages("fmt")	查找并添加包
option()	option("BUILD_TESTS", true)	设置配置选项
install()	add_installfiles("bin")	安装目标

表 13.1: CMake 和 XMake 的常用命令对比

目标、规则

在 XMake 中，目标可以是以下种类中的任意一种，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
- `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：

```
1 rule("embed")
2     set_extensions(".txt")
3     on_build_file(function(target, sourcefile)
4         -- 把文本编译成 .o 里的字符数组
5     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。

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。

```
1 add_requires("fmt") -- 添加 fmt 依赖
2 target("app")
3     set_kind("binary")
4     add_files("src/*.cpp")
5     add_packages("fmt") -- 链接 fmt 包
```

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：

```
1 $ xmake f -p windows -a x64 -m release # Windows 64-bit Release
2 $ xmake f -p linux -a arm64 --sdk=/opt/rpi
3 $ 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：

```
1 $ xmake service --start           # 本机当编译服务器
2 $ xmake f --remote_build=y
3 $ 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。

13.3.3 打包发布

XMake 的打包发布也和 CMake 大同小异：

```
1 xmake package -f deb              # 生成 .deb
2 xmake package -f nsis            # Windows 安装向导
3 xmake package -f zip              # 绿色压缩包
```

13.3.4 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。

13.3.5 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：

```
1 project/
2 - CMakeLists.txt
3 - src/
4 - tests/
5 - thirdparty/fmt/
```

首先使用 cmake2xmake 工具粗略转换为 XMake 项目：

```
1 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。

13.4 Meson²

Meson 是一个开源的构建系统，主要目标是尽可能地提高开发者的生产力。它力求快速、易用，并提供最现代化的软件构建工具。Meson 的设计哲学是“不挡路”（Don't get in your way），致力于让构建过程尽可能简单、快速和可靠。

与使用自定义脚本语言的 CMake 或使用 Lua 的 XMake 不同，Meson 使用一种专门设计的、非图灵完备的领域特定语言（DSL）。这种设计的目的是为了确保构建定义的简洁性、可读性和可预测性，避免在构建脚本中出现过于复杂的逻辑。Meson 的构建定义文件通常命名为 `meson.build`。

Meson 本身并不直接编译代码，而是作为一个元构建系统生成另一套构建系统（Ninja 或 Visual Studio 等）的项目文件，它的默认后端是 Ninja。得益于其出色的设计和性能，Meson 在开源社区获得了广泛的认可，许多大型项目，如 GNOME（包括较底层的 GLib、GTK 以及各应用程序）、GStreamer、Systemd、Mesa 等，都已采用 Meson 作为其官方构建系统。

安装 Meson 非常简单，大多数系统的包管理器都提供了 Meson：

```
1 sudo apt install meson # Debian/Ubuntu
2 sudo pacman -S meson   # Arch Linux
3 pacman -S mingw-w64-ucrt-x86_64-meson # MSYS2 on Windows
4 brew install meson     # macOS
```

Android 的 Termux 等少数环境没有提供 Meson，由于 Meson 是 Python 包，因此也可以方便地通过 pip 安装：

```
1 pip install meson
```

13.4.1 最小运行实例

Meson 强制要求“外源构建”（out-of-source builds），即构建生成的文件必须位于一个与源码目录分开的独立目录中，保持源码树的整洁。

假设我们有一个简单的 `main.c` 文件。在同级目录下，我们创建一个 `meson.build` 文件：

²本章作者周乾康。

```
1 # meson.build
2 project('hello_meson', ['c'],
3     version : '0.1',
4     default_options : ['c_std=c11'])
5
6 executable('hello', 'main.c')
```

然后，我们按以下步骤进行构建：

```
1 # 配置项目，创建构建目录 builddir
2 # 可以通过`--buildtype` 来设置构建类型
3 meson setup builddir --buildtype=debugoptimized
4
5 # 编译
6 # -C 指定构建目录
7 meson compile -C builddir
8
9 # 运行
10 ./builddir/hello
```

整个过程清晰明了。`meson setup` 步骤只会执行一次（除非构建脚本或环境变化），后续的修改只需要运行 `meson compile -C builddir`。其中，`--buildtype` 选项可以指定构建类型，如 `debug` 为调试模式，`release` 为发布模式，`debugoptimized` 为调试优化模式，`plain` 为纯粹的编译模式（使用环境变量中的构建参数），`minsize` 为最小化大小优化模式，`custom` 为自定义模式。对于调试模式，Meson 会自动设置编译器相关选项，添加调试符号。

13.4.2 项目和目标

每个 Meson 项目的根目录都必须有一个 `meson.build` 文件，并且第一条命令必须是 `project()`。

```
1 project('my_project', ['c', 'cpp']) # 项目名，使用的语言
```

目标（Target）是你想要构建的东西，可以是可执行文件或库等。

```
1 # 创建一个可执行文件
2 exe = executable('my_app', 'main.c', 'utils.c')
3
4 # 创建一个静态库
5 lib = static_library('my_lib', 'lib.c', 'helper.c')
6
7 # 创建一个共享库
8 shared_lib = shared_library('my_shared_lib', 'shared.cpp')
```

Meson 中的变量默认是不可变的，这有助于写出更可预测的构建脚本。上面代码中的 `exe` 和 `lib` 就是持有目标对象引用的变量。

13.4.3 依赖项处理

Meson 的依赖项处理是其一大亮点。它提供了一个统一的 `dependency()` 函数来查找外部依赖。

```
1 # 查找 fmt 库, 如果找不到则构建失败
2 fmt_dep = dependency('fmt', required: true)
3
4 executable('app', 'main.cpp',
5   dependencies : [fmt_dep]) # 将依赖项传递给目标
```

`dependency()` 函数会按顺序尝试多种方法来寻找依赖, 包括 (但不限于):

- **pkg-config**: 这是在 Linux 和类 UNIX 系统上查找库的首选方式。
- **CMake**: Meson 可以调用 `find_package` 来查找 CMake 包。
- **内置查找器**: 对于一些常用库 (如 Zlib、Threads), Meson 有自己的查找逻辑。

13.4.4 子项目与 Wrap 系统

Meson 拥有一个名为 Wrap 的内置依赖包管理器, 类似于 CMake 的 `FetchContent`。当系统上没有安装某个依赖时, Meson 可以通过 Wrap 系统从网上下载并构建它。

要使用它, 你需要在项目根目录下创建一个 `subprojects` 目录, 并在其中放置一个 `.wrap` 文件。例如 `subprojects/fmt.wrap`:

```
1 [wrap-file]
2 directory = fmt-10.2.1
3 source_url =
4   ↪ https://github.com/fmtlib/fmt/releases/download/10.2.1/fmt-10.2.1.zip
5 source_hash = e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
```

然后, 在 `meson.build` 中, 你可以像查找系统库一样查找它:

```
1 # Meson 会先尝试系统查找, 失败后会自动在 subprojects 目录中寻找 fmt
2 # fallback 的第二个参数是子项目中的依赖变量名
3 fmt_dep = dependency('fmt', fallback: ['fmt', 'fmt_dep'])
```

13.4.5 选项和配置

你可以使用 `option()` 函数为你的项目定义可配置的选项。

```
1 option('use_feature_x', type : 'boolean', value : true, description : 'Enable
  ↪ feature X')
```

用户在配置时可以通过 `-D` 参数来修改这些选项的值:

```
1 meson setup builddir -Duse_feature_x=false
```

要查看或修改一个已经配置好的构建目录的选项，可以使用 `meson configure`：

```
1 meson configure builddir # 查看所有选项
2 meson configure builddir -Duse_feature_x=true # 修改选项
```

13.4.6 交叉编译

交叉编译是 Meson 的一等公民。所有与交叉编译相关的设置都集中在一个单独的“交叉文件”（cross file）中。一个简单的交叉编译文件示例如下：

```
1 [binaries]
2 c = 'loongarch64-linux-gnu-gcc'
3 cpp = 'loongarch64-linux-gnu-g++'
4 ar = 'loongarch64-linux-gnu-ar'
5 strip = 'loongarch64-linux-gnu-strip'
6
7 [host_machine]
8 system = 'linux'
9 cpu_family = 'loongarch64'
10 cpu = 'loongarch64'
11 endian = 'little'
```

在配置时使用 `--cross-file` 参数指定此文件：

```
1 meson setup build_loongarch64 --cross-file loongarch64-linux-gnu.txt
2 meson compile -C build_loongarch64
```

Meson 会处理好所有细节，确保使用正确的编译器和库路径。

13.4.7 特点与比较

- **语法和设计哲学：**Meson 的语法是声明式的、非图灵完备的 DSL，虽然这造成了一些限制，但这也使得构建脚本非常简洁且易于分析。它强制实施了许多最佳实践（如外源构建）。
- **性能：**Meson 的配置阶段通常非常快。由于其默认后端是 Ninja，编译速度也极具竞争力。
- **易用性：**对于新项目或初学者来说，Meson 和 XMake 的上手难度通常低于 CMake。Meson 的错误提示非常友好，文档也极为出色。

13.4.8 技巧与提示

- **修改配置：**有时候因为 Meson 版本升级或者我们自己需要修改配置中的内容，需要重新配置项目构建目录。此时不必手动删除 `builddir` 目录来重新配置，使用 `meson setup --reconfigure builddir`。

- **调试：**Meson 的输出信息非常清晰。如果遇到问题，仔细阅读它的错误提示与引导通常就能解决问题。
- **官方文档：**Meson 的[官方文档](https://mesonbuild.com/)是学习和解决问题的极佳资源，内容详尽且组织良好。

Meson 提供了一种不同的构建体验。如果你正在开始一个新的开源项目，Meson 是一个非常值得考虑的优秀选择。

第十四章 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）。当然，Python 作为目前最流行的脚本语言，对于理科生而言也会是非常有用的，我将会单开一章介绍 Python 给理科生怎么玩。

14.1 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：

```
1 str = "Hello, world!"
2 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。

14.1.1 Python 的变量

Python 的变量非常简单，虽然也需要遵从“先声明再使用”的规则，但是其声明是隐式的，直接给变量赋值就行了。同时，Python 的语法也非常宽松，对于变量并不需要指定其类型，一个变量在不同的时间可以是任何类型的值。

```
1 a = 10 # 整数
2 a = 3.14 # 浮点数
3 a = 1 + 2j # 复数
4 a = "Hello, world!" # 字符串
5 a = [1, 2, 3] # 列表
6 a = (1, 2, 3) # 元组
```

```
7 a = {1, 2, 3} # 集合
8 a = {"name": "Alice", "age": 30} # 字典
9 a = True # 布尔值
10 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。

14.1.2 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：

```
1 a = 10
2 b = 3
3 print(a + b) # 输出 13
4 print(a - b) # 输出 7
5 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：

```
1 str1 = "Hello, "
2 str2 = "world!"
3 print(str1 + str2) # 输出"Hello, world!"
```

14.1.3 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：

```
1 name = input(" 请输入你的名字: ")
2 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：

```
1 name = "Alice"
2 age = 30
3 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：

```
1 print("Hello, World!", end="") # 不换行输出
2 print("Hello, 1", "Hello, 2", sep=", ") # 使用逗号分隔输出
3 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。

14.1.4 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。

```
1 # 这是一个注释
2 print("Hello, world!") # 这也是一个注释
3
4 """
5 这是一个多行注释
6 可以包含多行内容
7 """
```

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。

14.1.5 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：

```

1  a = "12321"
2  print(a+1) # 报错, 因为 a 是字符串, 1 是整数, 不能直接相加
3  b = int(a) # 将字符串转换为整数, 现在 b 是整数 12321
4  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。

14.2 控制程序的执行流程

14.2.1 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:

```

1 age = int(input(" 请输入你的年龄: "))
2 if age >= 18:
3     print(" 你是成年人。")
4 else:
5     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。

14.2.2 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:

```

1 for i in range(1, 11):
2     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:

```

1 i = 1
2 while i <= 10:
3     print(i)
4     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：

```
1 for _ in range(5):
2     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：

```
1 for i in range(1, 11):
2     if i == 5:
3         break # 当 i 等于 5 时，退出循环
4     print(i)
5 for i in range(1, 11):
6     if i == 5:
7         continue # 当 i 等于 5 时，跳过当前迭代
8     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。

14.3 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)。

14.3.1 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：

```
1 my_list = [1, 2, 3, "Hello", True]
2 print(my_list[0]) # 输出 1
3 print(my_list[3]) # 输出 "Hello"
4 print(my_list[-1]) # 输出 True (负索引从后往前计数)
5 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14.4节。例如:

```
1 my_list = [1, 2, 3, "Hello"]
2 my_list.append(4) # 添加元素 4
3 my_list.remove("Hello") # 删除元素"Hello"
4 my_list.sort() # 对列表进行排序
5 print(my_list) # 输出 [1, 2, 3, 4]
```

可以利用加号 `+` 来连接两个列表, 使用乘号 `*` 来重复列表。例如:

```
1 my_list1 = [1] * 1000 # 创建一个列表, 这个列表包含 1000 个 1
2 my_list2 = [2, 3, 4]
3 print([1]*my_list2) # 输出 [1, 2, 3, 4]
```

14.3.2 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:

```
1 my_tuple = (1, 2, 3, "Hello", True)
2 print(my_tuple[0]) # 输出 1
3 print(my_tuple[3]) # 输出"Hello"
4 print(my_tuple[-1]) # 输出 True
5 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:

```
1 my_tuple = (1, 2, 3)
2 my_tuple_1 = my_tuple + (4,) # 创建一个新的元组
3 print(my_tuple_1) # 输出 (1, 2, 3, 4)
```

14.3.3 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:

```
1 my_set = {1, 2, 3, "Hello", True}
2 print(my_set) # 输出 {1, 2, 3, "Hello", True}
3 my_set.add(4) # 添加元素 4
4 my_set.remove("Hello") # 删除元素"Hello"
5 print(my_set) # 输出 {1, 2, 3, 4, True}
```

14.3.4 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：

```
1 my_dict = {"name": "Alice", "age": 30, "is_student": False}
2 print(my_dict["name"]) # 输出 "Alice"
3 print(my_dict["age"]) # 输出 30
4 print(my_dict["is_student"]) # 输出 False
5 my_dict["age"] = 31 # 修改键 "age" 对应的值
6 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：

```
1 print(my_dict.keys()) # 输出 dict_keys(['name', 'age', 'is_student'])
2 print(my_dict.values()) # 输出 dict_values(['Alice', 31, False])
3 print(my_dict.items())
4 # 输出 dict_items([('name', 'Alice'), ('age', 31), ('is_student', False)])
```

14.3.5 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：

```
1 for item in my_list:
2     print(item) # 遍历列表
3 for item in my_tuple:
4     print(item) # 遍历元组
5 for item in my_set:
6     print(item) # 遍历集合
7 for key, value in my_dict.items():
8     print(key, value) # 遍历字典
9 for value in my_dict.values():
10    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：

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 print(my_list[2:5]) # 输出 [3, 4, 5]
3 print(my_list[:2]) # 输出 [1, 2] (步长为 1)
4 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：

```

1 squares = [x**2 for x in range(1, 11)]
2 print(squares) # 输出 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
3 evens = [x for x in range(1, 11) if x % 2 == 0]
4 print(evens) # 输出 [2, 4, 6, 8, 10]

```

以上代码中，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。

14.3.6 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：

```

1 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：

```

1 print(my_string[0]) # 输出 'H'
2 print(my_string[7]) # 输出 'w'
3 print(my_string[-1]) # 输出 '!'
4 print(my_string[13]) # 抛出 IndexError 异常
5 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：

```

1 print(my_string[0:5]) # 输出 'Hello'
2 print(my_string[7:]) # 输出 'world!'
3 print(my_string[:5]) # 输出 'Hello'
4 print(my_string[:2]) # 输出 'Hl'
5 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：

```

1 my_string.lower() # 'hello, world!' (转换为小写)
2 my_string.upper() # 'HELLO, WORLD!' (转换为大写)
3 my_string.strip() # 'Hello, world!' (去除首尾空格)
4 my_string.replace("world", "Python") # 'Hello, Python!' (替换子串)
5 my_string.split(", ") # ['Hello', 'world!'] (按逗号分割字符串)
6 my_string.find("world") # 7 (查找子串的位置)
7 my_string.startswith("Hello") # True (检查字符串是否以指定子串开头)
8 my_string.endswith("!") # True (检查字符串是否以指定子串结尾)
9 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。

```
1 my_string = 'Hello, world!'
2 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：

```
1 my_string = """Hello, world!"""
2 print(my_string) # 输出'Hello, world!'
3 my_string = '''Hello,
4 world!'''
5 print(my_string) # 输出'Hello,\nworld!'
```

对于字符串，我们可以使用加号 + 来连接两个字符串，使用乘号 * 来重复字符串。这个操作和列表是一样的。例如：

```
1 print("Hello, " + "world!") # 输出'Hello, world!'
2 print("Hello! " * 3) # 输出'Hello! Hello! Hello! '
```

\n 指的是换行。

14.4 函数和模块

14.4.1 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：

```
1 def add(a, b):
2     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：

```
1 result = add(3, 5)
2 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：

```
1 def greet(name="World"):  
2     print(f"Hello, {name}!")  
3  
4 greet() # 输出"Hello, World!"  
5 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：

```
1 def factorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * factorial(n - 1)  
6  
7 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：

```
1 def add(a: int, b: int) -> int:  
2     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！

14.4.2 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7.5。

例如，我们可以导入 Python 的内置模块 `math` 来使用数学模块：

```
1 import math  
2 print(math.sqrt(16)) # 输出 4.0 (计算平方根)  
3 print(math.pi) # 输出 3.141592653589793 (圆周率)  
4 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：

```
1 from math import sqrt, pi
2 print(sqrt(16)) # 输出 4.0
3 print(pi) # 输出 3.141592653589793
4 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:

```
1 import matplotlib.pyplot as plt
2 plt.plot([1, 2, 3], [4, 5, 6]) # 画一条线
3 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。

14.5 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:

```
1 with open("example.txt", "r") as file:
2     content = file.read()
3     print(content) # 输出文件内容
4 with open("example.txt", "w") as file:
5     file.write("Hello, world!") # 写入内容到文件
6 with open("example.txt", "a") as file:
7     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。

14.6 Python 的面向对象

Python 虽然是一种脚本语言, 更倾向于描述过程, 但是它也支持面向对象编程 (OOP)。

面向对象编程可以理解为把现实世界中的事物抽象成对象, 把有着相似属性和行为的事物归为一类, 通过对象来组织代码。每个对象都有自己的属性 (数据) 和方法 (行为), 通过对象之间的交互来实现复杂的功能。举个例子, 我们可以把 “人” 抽象成一个类 (class), 每个人都是这个类的一个实例 (instance)。每个人都有自己的属性 (如姓名、年龄等) 和方法 (如说话、走路等); 同时, 不同的人之间也有自己的特性 (如性别、职业等), 这些特性可以通过继承、多态等方式来实现。

14.6.1 类和对象的基本定义

一个 Python 的常见类定义如下：

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name # 属性: 姓名
4         self.age = age   # 属性: 年龄
5
6     def greet(self):
7         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

在这个例子中，我们定义了一个名为 `Person` 的类，它有两个属性（`name` 和 `age`）和一个方法（`greet`）。`__init__()` 方法是类的构造函数，用于初始化对象的属性。如果在类中希望使用自身的属性，我们需要使用 `self` 关键字来引用当前对象。这个 `self` 参数是必须的，它指向当前对象的实例。

我们可以使用这个类来创建一个对象，并调用它的方法：

```
1 person = Person("Alice", 30) # 创建一个 Person 对象
2 person.greet() # 调用 greet 方法, 输出"Hello, my name is Alice and I am 30 years
   ↳ old."
3 person2 = Person("Bob", 25) # 创建另一个 Person 对象
```

在现代 Python 中，以上定义可以变得更简单：

```
1 from dataclasses import dataclass
2
3 @dataclass # 使用数据类装饰器
4 class Person:
5     name: str # 属性: 姓名
6     age: int # 属性: 年龄
7     def greet(self) -> None:
8         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

上述例子又被称为“数据类”（`dataclass`），在 Python 3.7 中引入。使用数据类可以更方便地定义类，从而减少重复且意义有限的代码。“装饰器”是 Python 的一种语法糖，可以在函数或类定义前使用 `@` 符号来应用装饰器。数据类装饰器会自动为类生成一些常用方法，如 `__init__()`、`__repr__()` 等。

14.6.2 继承和多态

继承的意思是创建一个类（子类）来继承另一个类（父类）的属性和方法，子类可以扩展或修改父类的功能。举个例子：“动物”可以是一个类，而“猫”和“狗”都可以是“动物”的子类，它们继承了“动物”的属性和方法，但也有自己的特性。

实际代码表现大概是这样的：

```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def eat(self):
6         print(f"{self.name} is eating.")
7
8     def speak(self):
9         print(f"{self.name} makes a sound.")
10
11 class Dog(Animal):
12     def speak(self): # 重写父类方法
13         print(f"{self.name} barks.")
14
15 class Cat(Animal):
16     def speak(self): # 重写父类方法
17         print(f"{self.name} meows.")
18
19 mycat = Cat("Carol") # 创建一个 Cat 对象
20 mycat.eat() # 输出"Carol is eating."
21 mycat.speak() # 输出"Carol meows."
22 mydog = Dog("Dave") # 创建一个 Dog 对象
23 mydog.eat() # 输出"Dave is eating."
24 mydog.speak() # 输出"Dave barks."
```

在这个例子中，Dog 和 Cat 都是 Animal 的子类，它们继承了 Animal 的属性和方法，并重写了 speak() 方法。这样，我们可以通过多态来调用不同子类的同名方法，而不需要关心具体的实现细节。另一方面，子类也可以执行没有被重写的父类方法，例如上文中的 eat() 方法。

在有些时候，我们希望一个子类在重写某方法的时候会先将父类的方法执行一遍，然后再执行自己的逻辑。这时，我们可以使用 super() 函数来调用父类的方法：

```
1 class Dog(Animal):
2     def speak(self):
3         super().speak() # 调用父类的 speak 方法
4         print(f"{self.name} barks.")
5
6 mydog = Dog("Eve") # 创建一个 Dog 对象
7 mydog.speak() # 输出"Eve makes a sound." 和"Eve barks." 两行
```

有了 OOP，我们就可以更简单地组织代码，便于维护和扩展。

14.7 Python 与多文件

在实际开发中，我们通常会将代码分成多个文件，以便于管理和复用。Python 提供了模块化的机制，可以让我们轻松地在不同文件之间共享代码。

假设我们有一个文件 math_utils.py，里面定义了一些数学相关的函数：

```
1 # math_utils.py, 同目录下的文件
```

```
2 def add(a, b):
3     return a + b
4
5 class MyClass:
6     def __init__(self, value):
7         self.value = value
8
9     def double(self):
10        return self.value * 2
11
12 ---
13
14 # src/foo.py, 非同目录下的文件
15 def foo():
16     print("foo")
```

然后，我们可以在另一个文件 `main.py` 中导入这个模块，并使用其中的函数：

```
1 # main.py
2 # 如果在同目录下，直接导入即可
3 from math_utils import add, MyClass
4 result = add(3, 5)
5 print(result) # 输出 8
6 obj = MyClass(10)
7 print(obj.double()) # 输出 20
8
9 # 这么导入也行
10 import math_utils
11 result = math_utils.add(3, 5)
12 print(result) # 输出 8
13
14 # 如果不在同目录下，需要指定路径。实际上 src 虽然是一个文件夹，但在这里也被看作一个模块
15 from src.foo import foo
16 foo() # 输出 "foo"
```

Python 的多文件极为简单，完全不需要像 C++ 那样使用复杂的头文件和链接器。只要确保文件在同一目录下，或者在 Python 的搜索路径中，就可以把这些次要文件当作模块来导入，操作甚至和导入内置模块一模一样。

14.8 Python 语法小练

例题

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（但是它大概率是正确的），但是我们可以用 C++ 来验证一些具体值。要求输入一个整数，输出这个整数经过角谷猜想的处理后，最终得到 1 所需的每一步，例如输入 6，输出 6, 3, 10, 5, 16, 8, 4, 2, 1。

答案

我们读题，把人类语言逐句地改成 Python 语言。

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。这句话提示我们要做一个条件判断：

```
1 if n % 2 == 0: # 如果 n 是偶数
2     n = n // 2 # 除以 2
3 else: # 如果 n 是奇数
4     n = n * 3 + 1 # 乘以 3 加 1
```

如此反复循环，最终一定会得到 1。这句话提示我们要使用循环来反复执行这个操作，直到 n 变成 1 为止。我们可以使用 while 循环来实现：

```
1 while n != 1: # 当 n 不等于 1 时
2     # 循环体
```

要求输入一个整数，输出这个整数经过角谷猜想的处理后，最终得到 1 所需的每一步。这句话提示我们输入和输出的处理。

接下来，我们把这些代码像乐高积木一样组合起来，就可以完成这个练习了：

```
1 n = int(input(" 请输入一个整数: ")) # 输入一个整数
2 n = print(n) # 输出初始值
3 while n != 1: # 当 n 不等于 1 时
4     if n % 2 == 0: # 如果 n 是偶数
5         n = n // 2 # 除以 2
6     else: # 如果 n 是奇数
7         n = n * 3 + 1 # 乘以 3 加 1
8     print(n) # 输出当前的 n
```

例题

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)，再将其倍数都去掉；如此反复，直到所有的数都被处理完。现在输出 1 到 1000 之间的所有素数^a。

^a规定 1 不是素数，有兴趣的可以阅读这方面材料“为什么 1 既不是素数也不是合数”。

答案

题目说使用筛法找素数。那么，我们可以创建一个列表来存储 1 到 1000 之间的所有整数，然后使用一个循环来筛选出素数。如果不是素数，则删除之。那么可以这样：

```
1 # 创建一个列表来存储整数
2 numbers = list(range(2, 1001))
3 for i in range(2, 1001):
4     if i in numbers: # 如果 i 还在列表中
5         for j in range(2, 1001 // i): # 从 2 倍开始，删除 i 的所有倍数
6             if i * j in numbers: # 确保 i*j 还在列表里
```

```

7         numbers.remove(j * i) # 删除 j * i
8 print(numbers) # 输出所有素数

```

以上算法是正确的，但是执行起来比较慢。这是因为对于 2 到 1000 之间的所有数，我们都需要确定它究竟是不是在列表之中，总共判断了近 1000 次。有没有什么更简单的办法呢？

我们知道，数组的索引天生就是自然数。我们可以创建一个布尔数组来标记每个数是否是素数，初始时假设所有数都是素数。然后，我们从 2 开始，找到第一个素数，标记它的所有倍数为非素数。对于不是素数的数，我们可以跳过扫描。这样，代码就会快许多了。以下是一个实现：

```

1 numbers = [True] * 1001 # 创建一个布尔数组，初始时假设所有数都是素数
2 numbers[0] = numbers[1] = False # 0 和 1 不是素数
3 for i in range(2, 1001): # 从 2 开始，遍历所有数
4     if not numbers[i]: # 如果 i 不是素数
5         continue # 跳过
6     else:
7         for j in range(2, 1001 // i): # 从 2 开始，标记 i 的所有倍数为非素数
8             numbers[j * i] = False # 标记为非素数
9 print([i for i in range(1001) if numbers[i]]) # 输出所有素数

```

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 Python 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，就一定能够掌握 Python 这个强大的工具。

14.9 Jupyter Notebook

Jupyter Notebook 是一种交互式的计算环境，可以让我们在浏览器或其他前端中编写和运行 Python 代码。Jupyter Notebook 的文件扩展名是 `.ipynb`，它可以包含代码、文本、数学公式、图像等多种内容，非常适合用于数据分析、机器学习等领域。

要使用 Jupyter Notebook，首先需要安装它。可以使用 `pip` 命令来安装：

```

1 pip install notebook

```

当然，如果你使用的是 Anaconda 发行版，那么 Jupyter Notebook 往往已经预装好了。如果用的是 miniconda，则可能需要额外安装类似 `ipykernel` 这类包。

安装完成后，可以使用以下命令启动 Jupyter Notebook 服务器：

```

1 jupyter notebook

```

这会在浏览器中打开 Jupyter Notebook 的主页，可以在这里创建新的笔记本，或者打开已有的笔记本。当然，我们也可以直接创建一个 `.ipynb` 文件，然后用 Jupyter Notebook 或 VS Code 打开它。

在该环境中，我们可以创建多个单元格（cell），每个单元格可以包含代码或文本。代码单元格可以直接运行 Python 代码，并显示输出结果；文本单元格可以使用 Markdown 语法来编写格式化的文本。所有的 cell 都可以单独运行，互不影响；不过更常见的是把常规的 Python 代码拆成许多个 cell，然后从上到下依次运行，这样在下边的 cell 中就可以使用上边 cell 中定义的变量和函数了。两个代码 cell 中间也可以插入一个文本 cell 来解释代码的作用。

在运行 cell 时，其输出不会显示在终端或者控制台中，而是直接显示在 cell 的下方；matplotlib 等绘图库生成的图像也会直接显示在 cell 下方而不是弹出一个新窗口，非常方便。另外，Jupyter Notebook 还支持魔法命令（magic commands），可以用来执行一些特殊的操作，例如：

```
1 %timeit sum(range(1000)) # 计算代码运行时间
2 %matplotlib inline # 在 notebook 中显示 matplotlib 图像
```

Jupyter Notebook 在被关闭后不会清空其中的代码输出和可视化结果等，因此我们可以在下次打开时继续查看之前的结果。不过需要注意的是，Jupyter Notebook 的内核（kernel）在关闭后会被重置，所有的变量和函数都会被清空。因此，在重新打开 notebook 后，如果需要使用之前定义的变量和函数，需要重新运行相应的代码 cell。

在实际使用中，Jupyter Notebook 可以极大地提高我们的工作效率和代码可读性。它非常适合用于数据分析和机器学习等领域，这些领域代码、论述、结果并重。值得高兴的是，ipynb 文件本身就可以作为实验报告或者项目文档的一部分，方便分享和交流。即使不能提交该文件，也可以使用一些工具将其转换为 HTML、PDF 等格式，方便阅读和打印。它也适用于演示、教学等场景，可以让观众更直观地理解代码的运行过程和结果。然而，如果是代码为主的工作（例如工程），则不应使用它，因为它不适宜多文件协作开发，且版本控制不便。我们应当根据具体的需求选择合适的工具。

第十五章 Python 常用包

Python 作为时下最流行的高级脚本语言之一，在多个领域有着广泛的应用；不仅是文科学生可以利用它来进行数据分析、自然语言处理等工作，理科生也可以利用它来进行科学计算、机器学习等任务。本章将介绍一些非常常用的包，涵盖了机器学习、数学建模、数值计算、信号处理、算法可视化等大量内容，供大家按需使用。

15.1 超级 Excel：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。

```
1 import pandas as pd
2 df = pd.read_csv("data.csv") # 读取 CSV 文件
3 print(df.head()) # 输出前 5 行数据
4 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：

```
1 filtered_df = df[df["age"] > 18] # 筛选年龄大于 18 的数据
2 sorted_df = df.sort_values(by="name") # 按照姓名排序
3 df[df['朝代'] == '唐']['作者'].value_counts() # 输出唐代谁被提及最多
4 pd.merge(df1, df2, on='书名') # 合并两个表，随意拼接
```

15.2 自动分词的结巴：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：

```
1 import jieba
2 with open("book.txt", "r", encoding="utf-8") as file:
3     text = file.read()
4 words = jieba.cut(" 林黛玉葬花") # 分词, 精确模式
5 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。

15.3 PyTorch: 让你的电脑会学习

建议本节阅读者有一定的线性代数基础。

PyTorch 是一个开源的机器学习框架, 广泛应用于科研领域, 和大名鼎鼎的 TensorFlow 齐名。由于 PyTorch 更加灵活、动态, 因此在学生和科研人员中更受欢迎; 而 TensorFlow 则因为性能更强大而多用于生产环境。当然, 我们在这里不讨论后者 (因为我也不大会用)。

即使不用它来做机器学习, PyTorch 也可以作为一个非常强大的数值计算库来使用, 例如从一大堆点中拟合出一条直线等。或者说, NumPy++。

15.3.1 安装

如果希望使用 PyTorch, 一般建议先有一个 Python 虚拟环境和一个显卡 (最好是英伟达)。如果没有显卡, PyTorch 也可以在 CPU 上运行, 但速度会慢很多。

首先在终端中运行这个命令, 来判断你的显卡是什么 CUDA 版本:

```
1 nvidia-smi
```

然后去 [PyTorch 官网](https://pytorch.org), 根据你的 CUDA 版本和系统环境选择合适的安装命令。(无 GPU 版本则选 CPU。)

安装完毕后, 使用以下命令来测试 PyTorch 是否安装成功:

```
1 python -c "import torch, torch.cuda; print(torch.__version__,
↪ torch.cuda.is_available())"
```

要是输出 True, 说明你的 GPU 准备好了。即使是没有 GPU 的电脑, PyTorch 也可以正常运行, 只是速度会慢很多。

PyTorch 官网: <https://pytorch.org/get-started/locally/>

15.3.2 张量、梯度下降和自动求导

刚刚说到, PyTorch 是一个机器学习库, 因此我们至少应当了解这些基本概念。

张量 (Tensor) 是 PyTorch 的核心数据结构, 类似于 NumPy 中的数组。张量可以是标量 (0 维)、向量 (1 维)、矩阵 (2 维) 或更高维度的数组。使用 PyTorch 的情况下, 张量可以在 CPU 或 GPU 上运行, 显著提高并行速度。

梯度 (Gradient) 可以通俗理解为“导数”, 在机器学习上特指损失函数相对于模型参数的导数, 表示损失函数在某一点的变化率。而梯度下降则指, 通过计算损失函数相对于模型参数的梯度来更新模型参数, 使得损失函数最小化。PyTorch 提供了自动求导功能, 可以自动计算梯度。自动求导则是指, PyTorch 可以自动计算张量的梯度, 这样我们就不需要手动计算导数了。

```

1 import torch
2
3 # 像 NumPy 一样造张量
4 x = torch.arange(12, dtype=torch.float32).reshape(3, 4)
5 y = torch.randn_like(x)
6 z = torch.tensor([1., 2., 3.], dtype=torch.float32, requires_grad=True) #
  ↳ requires_grad=True 表示需要计算梯度
7
8 # 一句话把张量搬到 GPU 上
9 if torch.cuda.is_available(): # 如果确定 GPU 可用, 这个可以不写
10     x = x.cuda()
11     y = y.cuda()
12
13 # 广播、切片、点乘, 全部 NumPy 语法
14 xy = (x @ y.T).relu() # relu 也能直接点出来
15
16 z2 = x.pow(2).sum() # 求平方和
17 z2.backward() # 自动求导, 计算 z2 相对于 x 的梯度
18 print(x.grad) # 输出 x 的梯度, 输出应当是 [2., 4., 6.]
19

```

上文中的 `x @ y.T` 的意思是矩阵乘法 (点乘), `y.T` 表示 `y` 的转置。PyTorch 的张量支持广播 (Broadcasting) 机制, 这意味着当两个张量的形状不同时, PyTorch 会自动调整它们的形状以进行运算。`relu` 是一个函数, 用于将负值变为 0, 非负值不变。

15.3.3 数据集、数据加载器、预处理

在机器学习中, 我们通常需要处理大量的数据。PyTorch 提供了数据集 (Dataset) 和数据加载器 (DataLoader) 来帮助我们处理数据, 而不是使用 `for` 循环来缓慢地加载数据。

```

1 from torch.utils.data import Dataset, DataLoader
2 from torchvision import datasets, transforms
3
4 transform = transforms.Compose([ # 数据预处理
5     transforms.ToTensor(), # HWC to CHW, [0,255] → [0,1]
6     transforms.Normalize((0.1307,), (0.3081,)) # 数据集统计均值方差
7 ])

```

```

8
9 train_ds = datasets.MNIST(root='.', train=True, download=True,
    ↪ transform=transform)
10 train_dl = DataLoader(train_ds, batch_size=256, shuffle=True, num_workers=4)
11
12 for x, y in train_dl:          # x.shape == [256, 1, 28, 28]
13     ... # 这里应当存在一些代码用于表示训练逻辑, 但是这里懒得写了

```

15.3.4 nn.Module 搭积木式写网络

PyTorch 的 `nn.Module` 是一个非常强大的模块化网络构建工具。我们可以通过继承该类来定义自己的网络结构。例如：

```

1 import torch.nn as nn
2
3 class MLP(nn.Module):
4     def __init__(self):
5         super().__init__()
6         self.net = nn.Sequential(
7             nn.Flatten(),
8             nn.Linear(28*28, 256),
9             nn.ReLU(),
10            nn.Linear(256, 10)
11        )
12
13    def forward(self, x):
14        return self.net(x)
15
16 model = MLP().cuda() if torch.cuda.is_available() else MLP()

```

上述代码定义了一个叫做 MLP 的多层感知机 (MLP) 模型¹。它包含了一个输入层、一个隐藏层和一个输出层。我们使用 `nn.Flatten` 将输入的 28x28 的图像展平为一维向量, 然后通过两个全连接层 (`nn.Linear`) 进行处理, 中间使用 `ReLU` 激活函数 (`nn.ReLU`) 来增加非线性。

在上述实践中, 我们使用 `nn.Sequential` 来将多个层按顺序组合起来, 并使用 `nn.Module` 的 `forward` 方法定义了前向传播的逻辑。

15.3.5 真正训练

这里的内容实际上应该是科研组干的或者上课讲的。我们在这里举个例子就好了！

```

1 opt = torch.optim.Adam(model.parameters(), lr=1e-3)
2 loss_fn = nn.CrossEntropyLoss()
3
4 for epoch in range(3):
5     for x, y in train_dl:
6         x, y = x.cuda(), y.cuda()
7         pred = model(x)

```

¹MLP 是最基础的神经网络结构之一, 适用于处理结构化数据, 如图像、文本等。

```

8         loss = loss_fn(pred, y)
9
10        opt.zero_grad()
11        loss.backward()
12        opt.step()
13
14    print(f"epoch {epoch}: loss={loss.item():.4f}")

```

上述代码展示了一个简单的训练循环。我们使用 Adam 优化器（`torch.optim.Adam`）来更新模型参数，使用交叉熵损失函数（`nn.CrossEntropyLoss`）来计算损失。每个 epoch 中，我们遍历数据加载器（`train_dl`），获取输入数据和标签，然后进行前向传播、计算损失、反向传播和参数更新。

15.3.6 做个实验

多说无益。我们来做个实验，看看 PyTorch 能不能学会识别手写数字。

MNIST 数据集是一个经典的手写数字识别数据集，包含了 60000 个训练样本和 10000 个测试样本。我们可以使用 PyTorch 来加载这个数据集，并训练一个简单的神经网络来进行手写数字识别。当然，想让啥也不会的同学们写个代码帮助计算机识别 0 到 9 的数字还是太难了，我们就写个鉴别 0 和 1 的二分类器吧。

```

1  #!/usr/bin/env python3 # 这是一个 shebang 行，不写也行
2  """
3  mnist_01_linear.py
4  用 PyTorch 线性分类器区分 MNIST 的 0 和 1
5  运行环境: Python 最低 3.8, PyTorch 最低 1.13, torchvision
6  """
7
8  import torch
9  import torch.nn as nn
10 from torch.utils.data import DataLoader, Subset
11 from torchvision import datasets, transforms
12 from sklearn.metrics import accuracy_score # 仅用来算准确率，可省
13
14 # 1. 超参数，用于提纲挈领地控制训练流程
15 DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
16 BATCH_SIZE = 256 # 批大小，即每次训练使用多少张图片
17 EPOCHS = 5 # 训练轮数
18 LR = 0.1 # 学习率，控制参数更新的步长
19
20 # 2. 只保留 0 和 1 的子数据集，别的全都不要
21 transform = transforms.ToTensor() # 0-255 -> 0-1, [1,28,28]
22
23 def get_binary_mnist(root='.', train=True):
24     full = datasets.MNIST(root=root, train=train, download=True,
25     ↪ transform=transform)
26     idx = (full.targets == 0) | (full.targets == 1)
27     return Subset(full, torch.where(idx)[0])
28
29 train_ds = get_binary_mnist(train=True)
30 test_ds = get_binary_mnist(train=False)

```

```

31 train_dl = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True)
32 test_dl = DataLoader(test_ds, batch_size=BATCH_SIZE)
33
34 # 3. 线性二分类器
35 class LogisticRegression(nn.Module):
36     def __init__(self):
37         super().__init__()
38         self.flatten = nn.Flatten()           # 1*28*28 -> 784
39         self.linear = nn.Linear(28*28, 1)     # 输出 1 个 logit
40
41     def forward(self, x):
42         x = self.flatten(x)
43         return self.linear(x).squeeze()      # [B,1] -> [B]
44
45 model = LogisticRegression().to(DEVICE)
46
47 # 4. 损失 & 优化
48 criterion = nn.BCEWithLogitsLoss()          # 自带 sigmoid
49 optimizer = torch.optim.SGD(model.parameters(), lr=LR) # 使用最简单的 SGD 优化器
50
51 # 5. 训练
52 for epoch in range(1, EPOCHS+1):
53     model.train()
54     for x, y in train_dl:
55         x, y = x.to(DEVICE), y.float().to(DEVICE)
56         optimizer.zero_grad()
57         logits = model(x)
58         loss = criterion(logits, y)
59         loss.backward()
60         optimizer.step()
61     print(f"Epoch {epoch}: loss={loss.item():.4f}")
62
63 # 6. 测试
64 model.eval()
65 all_pred, all_true = [], []
66 with torch.no_grad():
67     for x, y in test_dl:
68         x = x.to(DEVICE)
69         probs = torch.sigmoid(model(x))
70         preds = (probs > 0.5).long()
71         all_pred.append(preds.cpu())
72         all_true.append(y)
73
74 # 7. 计算准确率
75 all_pred = torch.cat(all_pred).numpy()
76 all_true = torch.cat(all_true).numpy()
77 print("Test accuracy:", accuracy_score(all_true, all_pred))

```

以上代码使用的是 784 to 1 的线性分类器 (Logistic Regression)，通过 sigmoid 函数将输出转换为概率。我们使用二元交叉熵损失函数 (nn.BCEWithLogitsLoss) 来计算损失，并使用随机梯度下降 (SGD) 优化器来更新模型参数。

同学们很可能看不懂这些代码的细节，顶多能知道每一块代码是做什么的。不过，没关系！只要理解了整体用法，剩下的就是逐步掌握细节；而这则需要同学们在之后的课程和学习中不断探索和实践，不断优化自己的模型、调整模型的参数，从而真正成为机器学习的高手。

15.3.7 我看完了，然后呢

看完本节内容估计只需要五分钟。既然同学们这么快就看完了我写的基础内容，那么接下来就可以去官网自己查自己需要的内容了。PyTorch 官方的[60 分钟速通 PyTorch](#)课程链接我已经放在这里了，感兴趣的同学们可以自己去看看。

15.4 NumPy+SciPy: 科学计算的利器

建议本节阅读者有一定的线性代数和高等数学基础。

NumPy 和 SciPy 是 Python 中用于科学计算的两个重要库。NumPy 提供了高效的多维数组对象和各种数学函数，而 SciPy 则在此基础上提供了更多的科学计算功能，如优化、积分、插值等。掌握了这两个库，Python 就具有了 MATLAB 的核心战斗力，却又保留了 Python 的灵活性和易用性。

15.4.1 NumPy: 多维数组和矩阵运算

NumPy 的核心数据类型是 ndarray (N 维数组)，它可以存储多维数据。这玩意可不是低级的“列表 + 语法糖”，而是一个真正的 C 风格连续内存块和元数据。它要求我们应当具有**向量思维**。

对于该数据类型，我们熟悉这两个术语就可以：形状 (shape) 和步长 (stride)。形状表示数组的维度；步长表示每个维度的跨度。举例：

```
1 import numpy as np
2 a = np.arange(12).reshape(3, 4)
3 a.strides      # 每一行 32 字节，因为 int64=8B * 4 列
4
5 # 应当是 (32, 8)
```

向量化三件套

NumPy 的向量化三件套是指：广播 (Broadcasting)、通用函数 (ufuncs) 和切片视图 (Slicing Views)。这三者结合起来，可以让我们在处理大规模数据时，避免使用 for 循环，从而提高计算效率。

一个个讲吧：

- 广播：从尾部维度开始比较，长度相等或其中一维为 1 即可兼容。

```
1 pts = np.random.randn(1000, 3)
2 t    = np.array([1.0, 2.0, 3.0])
3 shifted = pts + t      # (1000,3) + (3,), 自动广播
```

- 通用函数：NumPy 提供了许多通用函数（ufuncs），可以对数组进行元素级的操作，如加减乘除、三角函数等。

```
1 pts = np.random.randn(1000, 3)
2 norms = np.linalg.norm(pts, axis=1) # 计算每个点的范数
```

- 切片视图：NumPy 的切片操作返回的是原数组的视图，而不是复制数据。这意味着对切片的修改会影响原数组。

```
1 pts = np.random.randn(1000, 3)
2 pts[:, 0] = 0.0 # 将所有点的 x 坐标设为 0
3 print(pts[0]) # 输出第一个点的坐标, x 坐标应为 0.0
```

不如做个性能对比实验：给你一千万（1e7）个点，计算它们的欧氏范数。欧式范数的计算方法是： $\sqrt{x^2 + y^2 + z^2}$ 。vanilla 写法自己去写，我这里只给出 NumPy 的写法：

```
1 import numpy as np
2 pts = np.random.randn(int(1e7), 3) # 生成一千万个三维点
3 norms = np.sqrt(np.einsum('ij,ij->i', pts, pts)) # 使用爱因斯坦求和约定计算欧氏范数
```

运行你的代码和 NumPy 的代码，比较一下性能。使用爱因斯坦求和约定计算范数应当是最快的算法，预计比基准线快 50 倍以上。

15.4.2 SciPy：科学计算的扩展

SciPy 是站在 NumPy 上的科学计算库，提供了更多的科学计算功能，如优化、插值、积分、信号处理等。SciPy 的模块化设计使得它可以非常方便地进行各种科学计算，可以将高阶算法压成一行随便使用，显著加快了运算效率。

优化

SciPy 提供了许多优化算法，可以用于许多问题，例如梯度下降、牛顿法、共轭梯度等。我们可以使用 `scipy.optimize` 模块来进行优化，以下函数是一个最小化函数的例子：

```
1 from scipy.optimize import minimize
2
3 rosenbrock = lambda x: (1-x[0])**2 + 100*(x[1]-x[0]**2)**2
4 result = minimize(rosenbrock, x0=[2,2], method='BFGS', jac='2-point')
5 print(result.x, result.nit) # [1. 1.] 24
```

以上代码使用 BFGS 算法最小化 Rosenbrock 函数，初始点为 (2, 2)，最终结果应当接近 (1, 1)。不知道 Rosenbrock 函数是什么的同学可以看[维基百科](https://en.wikipedia.org/wiki/Rosenbrock_function)。

比较常见的 `minimize` 优化参数包括 `jac`（梯度计算方式）、`tol`（容忍度）、`options`（其他选项）等。SciPy 还提供了许多其他的优化函数，如 `scipy.optimize.linprog` 用于线性规

划, `scipy.optimize.curve_fit` 用于曲线拟合等。这些函数同学们都可以查阅[SciPy 官方文档](#)。

稀疏矩阵

稀疏矩阵在科学计算中非常常见。SciPy 提供了 `scipy.sparse` 模块来处理稀疏矩阵。

```
1 from scipy.sparse import diags
2 n = 1000
3 k = [-1, 0, 1]
4 data = [np.full(n-1, -1), np.full(n, 2), np.full(n-1, -1)]
5 L = diags(data, k, shape=(n, n), format='csr') # 三对角稀疏
6
7 from scipy.sparse.linalg import spsolve
8 x = spsolve(L, np.ones(n)) # 求解  $Lx = 1$ 
```

以上代码创建了一个三对角稀疏矩阵 L , 主对角线为 2, 次对角线为 -1。SciPy 的稀疏矩阵支持许多操作, 如矩阵乘法、求逆等。例如上述元素代码中使用 `spsolve` 函数来求解线性方程组 $Lx = 1$, 其中 1 是一个全 1 向量。

数值积分

有些时候, 对于一些积分我们不是很容易求出解析解, 这时候就需要数值积分了。SciPy 提供了 `scipy.integrate` 模块来进行数值积分。

```
1 from scipy.integrate import quad
2 val, abserr = quad(lambda x: np.exp(-x**2), 0, np.inf)
3 print(np.sqrt(np.pi)/2 - val) # 输出误差, ~1e-13
```

上述代码使用 `quad` 函数计算了从 0 到无穷大的高斯函数的积分, 结果应当接近 $\sqrt{\pi}/2$ 。quad 函数返回两个值: 积分值和绝对误差。

15.4.3 两个一起, 双倍开心

一般情况下, 我们需要将 NumPy 和 SciPy 协同使用以实现更强大的功能, 前者向量化批处理, 后者提供算法和工具, 十分快乐。以下是一个示例, 利用二次 Bezier 曲线拟合 100 个带噪声的二维点, 并最小化点到曲线距离的平方和:

```
1 import numpy as np
2 from scipy.optimize import minimize
3
4 # 数据
5 pts = np.c_[np.linspace(0,1,100)**2,
6              np.linspace(0,1,100)] + np.random.randn(100,2)*0.02
7
```

```

8 # Bezier 曲线函数
9 bezier = lambda t, p0, p1, p2: (1-t)**2*p0 + 2*(1-t)*t*p1 + t**2*p2
10
11 def error(params):
12     p0, p1, p2 = params.reshape(3,2)
13     t = np.linspace(0,1,100)
14     curve = bezier(t[:,None], p0, p1, p2)
15     return np.sum((curve - pts)**2)
16
17 result = minimize(error, x0=np.random.rand(6))
18 print(result.x.reshape(3,2))

```

上述代码首先生成了 100 个带噪声的二维点，然后定义了一个二次 Bezier 曲线函数，并使用最小化算法拟合这些点。最终输出的结果是拟合曲线的控制点坐标。

15.5 Matplotlib：数据可视化的神器

在上一章中我们讲过，可以使用 Matplotlib 来绘制各种统计图表和数据可视化图形。这是 Python 里面最常用的可视化库之一，也是生态中最早、最稳定、最通用的可视化库。另一方面，它能够和 NumPy、Pandas 等库无缝集成，提供了强大的绘图功能。

15.5.1 基本用法

Matplotlib 的基本用法是通过 pyplot 模块来实现的。我们可以使用以下代码来绘制一个简单的折线图：

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 10, 100)
5 y = np.sin(x)
6
7 plt.plot(x, y, label='sin(x)') # 绘制曲线
8 plt.xlabel('x') # 添加坐标轴标签
9 plt.ylabel('sin(x)') # 添加坐标轴标签
10 plt.title('Simple Plot') # 添加标题
11 plt.legend() # 添加图例
12 plt.show() # 显示图形

```

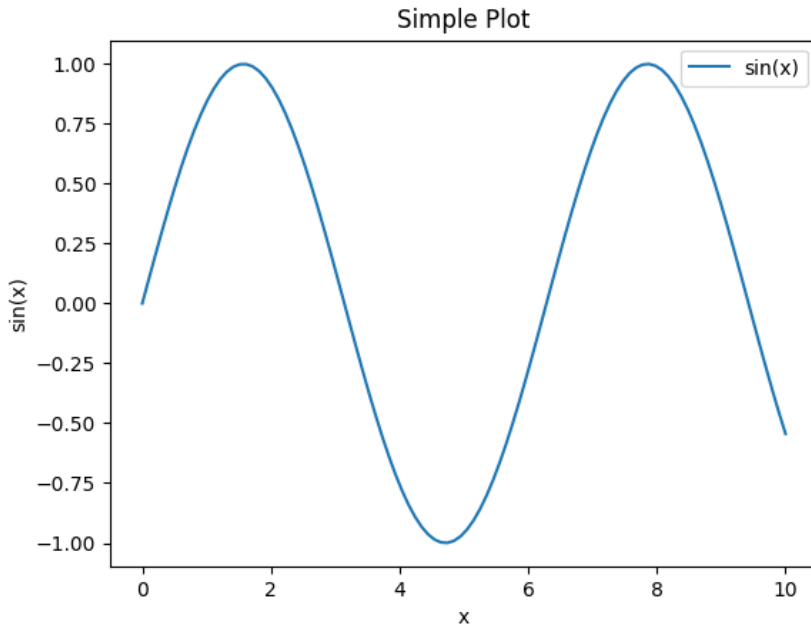
上述代码使用 plt.plot 函数绘制了一个简单的折线图，并添加了坐标轴标签、标题和图例。最后使用 plt.show() 函数显示图形。

除了使用简洁的 pyplot 以外，还可以使用面向对象风格的 API 来绘图，便于封装和复用：

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 2*np.pi, 100) # Create x values from 0 to 2pi
5
6 fig, ax = plt.subplots(figsize=(8, 6)) # 创建一个图形和坐标轴对象

```



```

7 ax.plot(x, np.sin(x), label=r'$sin(x)$', color='tab:blue', lw=2) # 绘制曲线
8 ax.plot(x, np.cos(x), label=r'$cos(x)$', ls='--') # 绘制曲线
9 ax.set_xlabel(r'$x$') # 添加坐标轴标签
10 ax.set_ylabel(r'$y$') # 添加坐标轴标签
11 ax.legend() # 添加图例
12 ax.set_title('Matplotlib Example') # 添加标题
13 fig.tight_layout() # 自动调整布局
14 plt.show() # 显示图形

```

上述代码使用 `plt.subplots` 函数创建了一个图形和坐标轴对象，然后使用坐标轴对象的 `plot` 方法绘制曲线。这样可以更灵活地控制图形的各个部分。

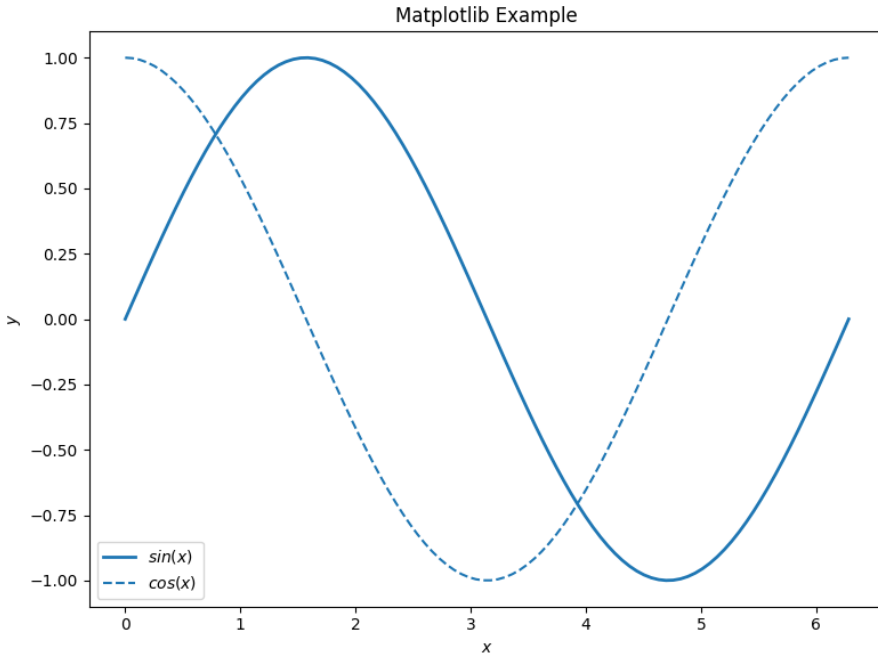
15.5.2 和其他包协同工作

Matplotlib 可以和 NumPy、Pandas 等库无缝集成，提供了强大的绘图功能。例如，我们可以使用 NumPy 生成数据，然后使用 Matplotlib 绘制图形：

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 X, Y = np.meshgrid(np.linspace(-3, 3, 300),
5                   np.linspace(-3, 3, 300))
6 Z = np.sinc(np.sqrt(X**2 + Y**2))
7
8 fig, ax = plt.subplots()
9 c = ax.contourf(X, Y, Z, levels=50, cmap='viridis')
10 fig.colorbar(c, ax=ax)
11
12 plt.title('Contour Plot of Sinc Function')

```



```
13 plt.xlabel('X-axis')
14 plt.ylabel('Y-axis')
15 plt.show()
```

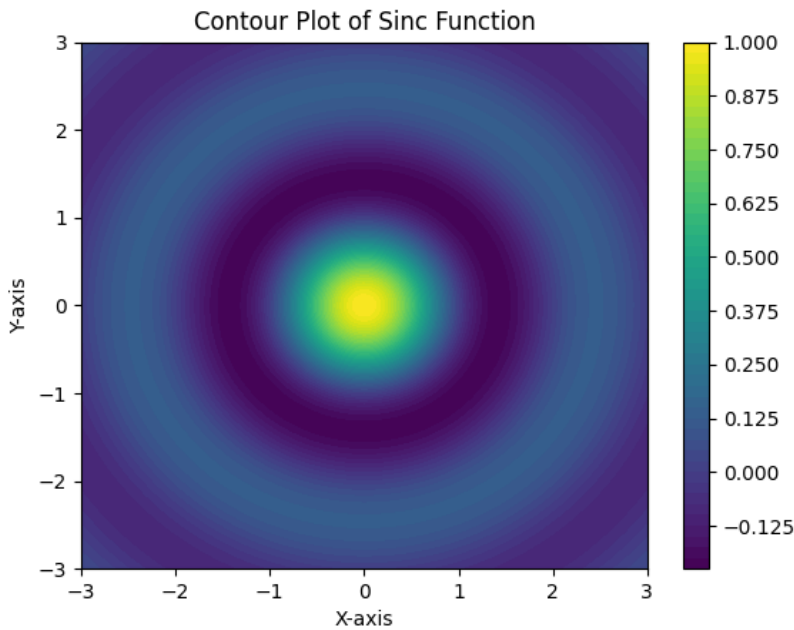
或者使用 Pandas 绘制数据框的图形：

```
1 import pandas as pd
2
3 df = pd.read_csv('experiment.csv')
4 ax = df.plot(x='voltage', y='current', kind='scatter', color='k')
5 ax.set_xlabel('Voltage (V)')
6 ax.set_ylabel('Current (mA)')
7 # 这只是一个示例，没有数据是真不行，后面绘图的逻辑自己写就好了
```

15.5.3 子图布局

在实际应用中，我们经常需要将多个图形绘制在同一个窗口中，这就需要用到子图的概念。Matplotlib 提供了 `subplot` 和 `subplots` 函数来创建子图。

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 10, 100)
5 y1 = np.sin(x)
6 y2 = np.cos(x)
7
8 fig, axs = plt.subplots(2, 1, figsize=(8, 6)) # 创建 2 行 1 列的子图
9 axs[0].plot(x, y1, label='sin(x)', color='tab:blue')
```



```
10 axs[0].set_title('Sine Function')
11 axs[1].plot(x, y2, label='cos(x)', color='tab:orange')
12 axs[1].set_title('Cosine Function')
13 plt.tight_layout()
14 plt.show()
```

上述代码创建了一个包含两个子图的图形窗口，分别绘制了正弦函数和余弦函数。使用 `plt.tight_layout()` 函数可以自动调整子图之间的间距。

15.5.4 风格和导出

Matplotlib 支持多种风格，可以通过 `plt.style.use` 函数来设置风格。例如，我们可以使用 `science` 和 `ieee` 风格来绘制图形，使得图形符合 IEEE 论文的格式要求：

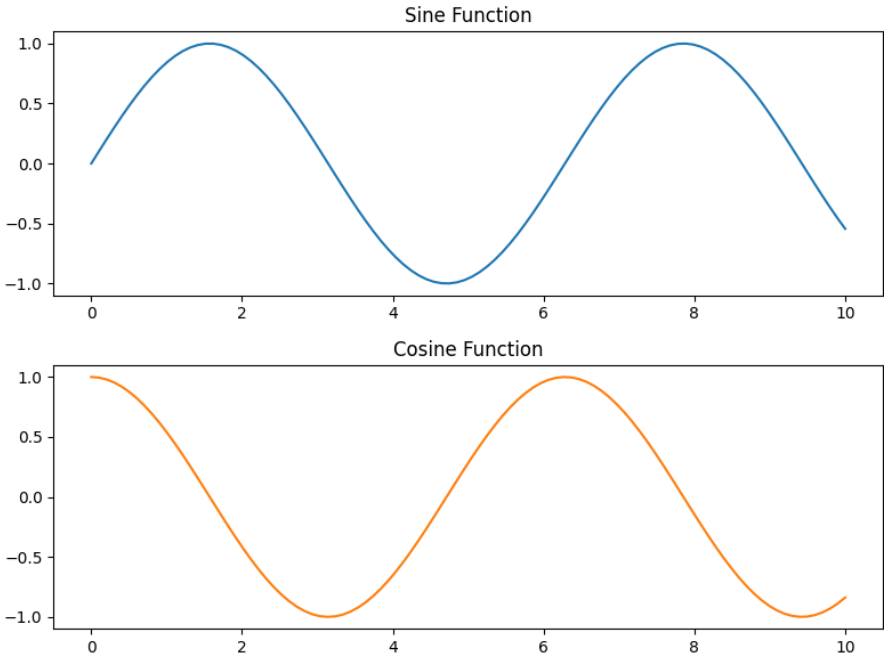
```
1 plt.style.use(['science', 'ieee']) # 需安装 SciencePlots
```

当然，如使用该风格，则默认需要 `LaTeX` 来渲染文本，而这是非常缓慢的。因此可以在该列表中添加 `no-latex` 选项，以禁用 `LaTeX` 渲染：

```
1 plt.style.use(['science', 'ieee', 'no-latex']) # 需安装 SciencePlots
```

此外，我们还可以将图形导出为各种格式，如 PNG 等：

```
1 plt.savefig('figure.svg') # 矢量图
2 plt.savefig('figure.png', dpi=600, transparent=True) # PNG 图
```



15.5.5 常见坑与提示

- 中文字体乱码：提前设置 rcParams，或者使用 matplotlib.font_manager 来设置中文字体。

```
1 import matplotlib.pyplot as plt
2 plt.rcParams['font.sans-serif'] = ['SimHei'] # 设置中文字体
3 plt.rcParams['axes.unicode_minus'] = False # 解决负号显示为方块的问题
```

- 在 Jupyter 中不显示：使用 %matplotlib inline 命令来确保图形在 Jupyter Notebook 中显示。
- 颜色过多：使用 tab: 前缀来使用 Matplotlib 内置的颜色表，避免颜色过多导致的混乱。
- 图例遮挡：使用 plt.legend(loc='best') 自动解决图例位置问题。

除此之外，Matplotlib 还有许多其他的功能，如动画、3D 绘图等，这些内容同学们可以在[Matplotlib 官方示例](#)中找到。

15.6 SymPy：符号计算高级计算器

建议本节阅读者有一定的高等数学基础。

SymPy 是一个用于符号计算的 Python 库。符号计算是指对数学表达式进行符号操作，而不是数值计算；或者说，**求解析解而不是数值解**。SymPy 可以用于求解方程、积分、微分、矩

Matplotlib 官方示例: <https://matplotlib.org/stable/gallery/index.html>

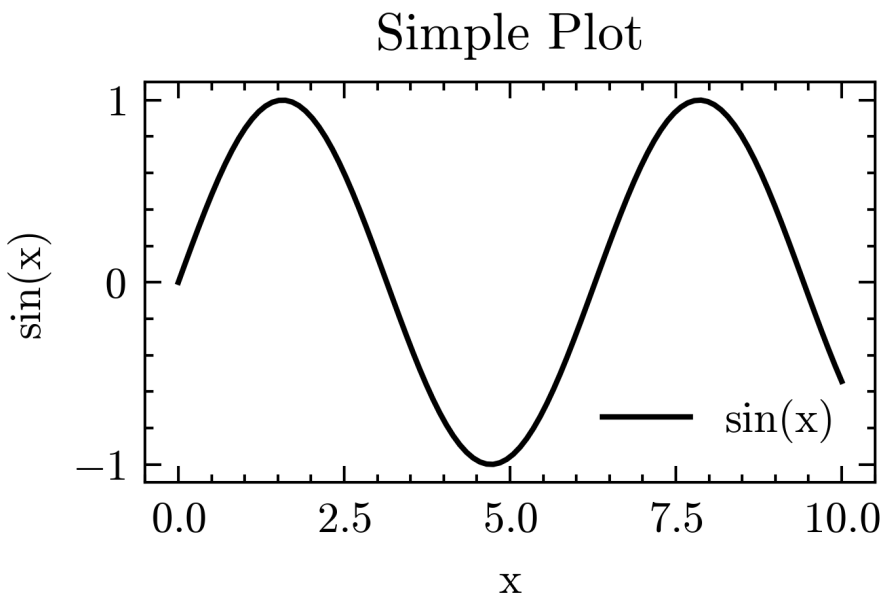


图 15.1: 使用 SciencePlots 风格绘制的图形

阵运算等多种数学操作。它的语法类似于 Mathematica 和 Maple，但使用 Python 语言编写，因此更易于学习和使用。

在使用该库之前，我们尽量在文件的靠前位置添加下面这一行，这样可以让 SymPy 的输出更加美观。

```
1  init\_printing(use\_unicode=True)
```

15.6.1 基本数据类型

SymPy 的基本数据类型有三个：符号、表达式、等号。符号是 SymPy 的核心数据类型，用于表示数学符号，如变量、常数等。表达式是由符号和运算符组成的数学表达式，可以进行各种数学操作。等号用于表示等式关系。

```
1  x, y = symbols('x y', real=True) # 定义符号变量 x 和 y
2  expr = x**2 + y**2 # 定义表达式
3  line = Eq(y, 3*x + 2) # 定义等式
4  print(expr) # 输出表达式
```

上述代码定义了两个符号变量 x 和 y ，并定义了一个表达式 $x^2 + y^2$ 和一个等式 $y = 3x + 2$ 。SymPy 的符号计算可以对这些符号进行各种操作，如求导、积分、化简等。

比方说：

```
1  expr = (x+1)*(x-2)-(x**2-2)
2  simplify(expr) # 化简表达式，得到-x
```

```

3     expand((x+1)**5) # 展开多项式
4     factor(x**3+1) # 因式分解
5     limit(expr, x, 0) # 求 expr 在 x=0 处的极限
6     diff(expr, x) # 对 expr 求一次导数
7     integrate(expr, x) # 对 expr 求不定积分
8     integrate(expr, (x, 0, 1)) # 对 expr 求定积分
9     series(expr, x, 0, 5) # 求泰勒级数展开, 最高项次数为不超过 4
10    series(expr, x, 0, 5).removeO() # 去掉高阶项
11    solve(x**4-1, x) # 求解等式  $x^{**4}-1=0$  的解
12    solve(a*x**2+b*x+c, x) # 求解二次方程  $ax^2+bx+c=0$  的解, 带着参数也能算
13    nonlinsolve([x**2+y**2-1, x-y], (x, y)) # 求解非线性方程组

```

多元函数也可以使用上述方法来求导和积分, 这里就不写了。需要注意的是, 这里的求导数都是**偏导数**。一个非常有趣的事情是: 在该库中, 使用 `-oo` 和 `oo` 来表示负无穷和正无穷, 而不是使用 `float('inf')`。(疑似有些过于符号化了)

15.6.2 矩阵和线性代数

SymPy 还提供了矩阵和线性代数的功能, 可以进行矩阵运算、求逆、求特征值等操作。SymPy 的矩阵是一个二维数组, 可以进行各种矩阵运算, 如加法、乘法、转置等。

```

1     A = Matrix([[1, 2], [3, 4]]) # 定义一个矩阵
2     b = Matrix([x, y]) # 定义一个列向量
3
4     A.det() # 求矩阵的行列式
5     A.inv() # 求矩阵的逆
6     A.eigenvals() # 求矩阵的特征值
7
8     linsolve((A, b), [x, y]) # 求解线性方程组  $Ax = b$ 
9
10    P, D = A.diagonalize() # 对矩阵对角化, P 为特征向量矩阵, D 为对角矩阵

```

15.6.3 输出、可视化

在 Jupiter 里一行搞定输出, 贴出来的字符串直接粘到论文里就能编译:

```

1     latex(Integral(exp(-x**2), (x, 0, oo)))

```

如果需要将表达式可视化, 可以使用 SymPy 的 `plot` 函数来绘制函数图像:

```

1     plot(sin(x)/x, (x, -10, 10))

```

15.6.4 常见坑

- 符号爆炸: 符号计算缓慢是正常现象, 表达式也会越来越大。必要的时候, `simplify`、`expand`、`factor` 等函数可以帮助我们化简表达式。

- 算错：这是常规现象，符号计算可能出现算错是不可避免的。我们可以快速使用数值验证结果的正确性。
- 并行：Sympy 不能并行，但是可以拆任务并行计算（如 multiprocessing）。祝愿玩得开心，下次计算不用抄公式抄到手软！

15.7 爬虫

爬虫很难说是一个包。它实际上是一种技术，或者说是一种思维方式。爬虫的核心思想是：模拟浏览器行为，自动化地获取网页内容。Python 中有许多用于爬虫的库，如 Requests、BeautifulSoup、Scrapy 等。

一般而言，爬虫的基本流程包括以下几个步骤：

1. 发送 HTTP 请求：使用 Requests 库发送 HTTP 请求，获取网页内容。
2. 解析网页内容：使用 BeautifulSoup 库解析 HTML 内容，提取所需的数据。
3. 存储数据：将提取的数据存储到本地文件或数据库中。

以下是一个简单的爬虫示例，使用 Requests 和 BeautifulSoup 库来爬取一个网页的标题：

```
1 import requests
2 from bs4 import BeautifulSoup
3 url = 'https://example.com'
4 response = requests.get(url)
5 soup = BeautifulSoup(response.text, 'html.parser')
6 title = soup.find('title').text
7 print(title)
```

上述代码发送了一个 HTTP GET 请求，获取了网页内容，并使用 BeautifulSoup 解析 HTML，提取了网页的标题并打印出来。

当然，上述代码只是一个非常简单的爬虫示例，实际应用中可能需要处理更多的细节，如处理分页、模拟登录、处理 JavaScript 渲染等。另一方面，很多网站实际上已经做了反爬虫处理，想要成功爬取数据可能需要更多的技巧和方法。出于个人原因，笔者也很少使用爬虫技术，因此我在这里就不做过多介绍了，感兴趣的同学可以自行查找相关资料进行了解。

注意

爬虫涉及到法律和道德问题，爬取数据时应当遵守网站的 robots.txt 文件和相关法律法规，避免侵犯他人的权益。

15.8 OpenAI：自己做自己的 Agent

Cherry Studio 等软件给了我们一个强大的图形界面，让我们可以通过各种方式来使用 LLM。但是涉及到一些自动化的任务时，还是需要自己编写代码来实现。OpenAI 官方提供了一个 Python SDK，可以让我们方便地使用 OpenAI 的各种模型和功能。

实际上, OpenAI 的 Python SDK 非常简单易用, 甚至可以说是简陋到傻瓜级别的。我们如果希望使用该包, 只需要记住代码分为两步: 定义 `client`, 使用 `client`。以下是一个简单的示例, 使用 OpenAI 的 GPT-4 模型来生成文本:

```
1 from openai import OpenAI
2
3 API_KEY = 'your_api_key' # 这个是必须的
4 BASE_URL = 'https://example.com/v1' # 请改成服务供应商提供的地址
5
6 # 定义 client
7 client = OpenAI(api_key=API_KEY, base_url=BASE_URL)
8
9 # 使用 client
10 with client.chat.completions.create(
11     model='gpt-4o', # 取决于实际情况
12     messages=[
13         {'role': 'system', 'content': 'You are a helpful assistant.'},
14         {'role': 'user', 'content': 'Hello, how are you?'}
15     ] # Prompts 需要自己写
16     # 这里也可以添加 temperature, max_tokens 等参数
17 ) as response:
18     print(response.choices[0].message.content)
```

在上述代码中, 我们首先定义了一个 OpenAI 客户端, 然后使用该客户端发送了一个聊天请求, 获取了模型的回复并打印出来。使用 `with` 语句可以确保资源的正确释放, 是良好的习惯。

于是上述库的使用就这么简单, 疑似确实是傻瓜级别的。更多的功能和用法可以参考相关文档。当然, 如果希望进行记忆持久化等功能, 那需要其他的库来实现, 例如 `langchain` 和 `langgraph` 等。

开放性思考和探索

以下内容是一些简单的实践选题, 供同学们参考。可以使用 $\text{\LaTeX}$ 来形成实验报告!

1. 用《红楼梦》文本做词-人共现网络: 试着读入一段红楼梦文本, 统计人物和词语的共现关系, 并使用 `SnowNLP` 等库为人物的情绪打分, 最终观察人物的情绪走势。
2. 用 `PyTorch` 试着复现线性回归的解析解: 试着生成一百万个三维点, 使用 `PyTorch` 实现线性回归, 并与解析解进行对比。提示: 解析解可以使用 `Numpy` 求得, 公式是 $w^* = (X^T X)^{-1} X^T y$ 。
3. 使用多个库做信号处理, 做出一个简单的调音器: 可以使用 `pyaudio` 库来辅助录音, 使用 `SciPy` 来做 FFT 变换, 使用 `Matplotlib` 来做频谱图, 使用 `SymPy` 来做滤波器设计。
4. 使用 `PyTorch` 训练一个简单的神经网络来做手写数字特征提取; 然后使用 `UMAP` 来降维可视化。提示: `ResNet`。
5. 混沌摆模拟: 使用 `SciPy` 的 ODE 求解器来模拟一个混沌摆的运动轨迹, 然后使用 `Matplotlib` 来绘制相空间图和时间序列图。你能做出一个动画吗? 如果你有 GPU, 可以使用 `PyTorch` 来并行, 看看混沌敏感性。

第五部分

实用主义编程：写出更好的代码

第十六章 Git 与版本控制

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。

再试想另一个环境：假设我们正在开发一个大型项目，项目中有很多人参与开发。如果使用传统的方式来分发代码，那么每个人都要手动下载代码，修改代码，然后再上传代码。这时候就会出现很多问题，例如代码冲突、版本不一致等。那这就需要专门的一个人或者几个人来管理代码的版本和分发，但是这样就会显著增加工作量和复杂度。

为了避免以上问题，我们引入了版本控制（VCS）系统。一般来说，VCS 系统可以分为两类：集中式版本控制系统（CVCS，也叫中心化的）和分布式版本控制系统（DVCS，也叫去中心化的）。集中式版本控制系统的特点是所有的代码都存储在一个中心服务器上，所有的开发者都需要从中心服务器上下载代码，然后再上传代码；而分布式版本控制系统的特点是每个开发者都有一份完整的代码库，所有的操作都是在本地进行的，然后再将修改推送到中心服务器上。这样就可以避免代码冲突、版本不一致等问题。

2002 年以前，Linux 内核开发完全依赖于 Linux 一个人手工检查并合并全世界发来的补丁，这样工作量非常大。于是，Linux 的一个朋友介绍了 BitMover 公司开发的商业 VCS 软件 BitKeeper 免费授权给 Linux 开发团队使用。此举招致了 FSF 的 RMS 等人的批评，认为在自由软件开发中使用非自由软件是“道德上有污点”的行为。但是作为实用主义者的 Linux 并不在意这些事情，BitKeeper 作为去中心化的 VCS，满足了 Linux 的需求。然而好景不长，有 Linux 内核开发者逆向了 BitKeeper 的协议，致使 BitMover 公司在 2005 年决定收回其授权。Git 就是在这种条件下诞生的，据说第一版 Git 是 Linux 利用 1 周休假时间完成的。随着 Linux 的广泛应用，Git 也逐渐成为了最流行的去中心化版本控制系统，也是目前最流行的版本控制系统。

16.1 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

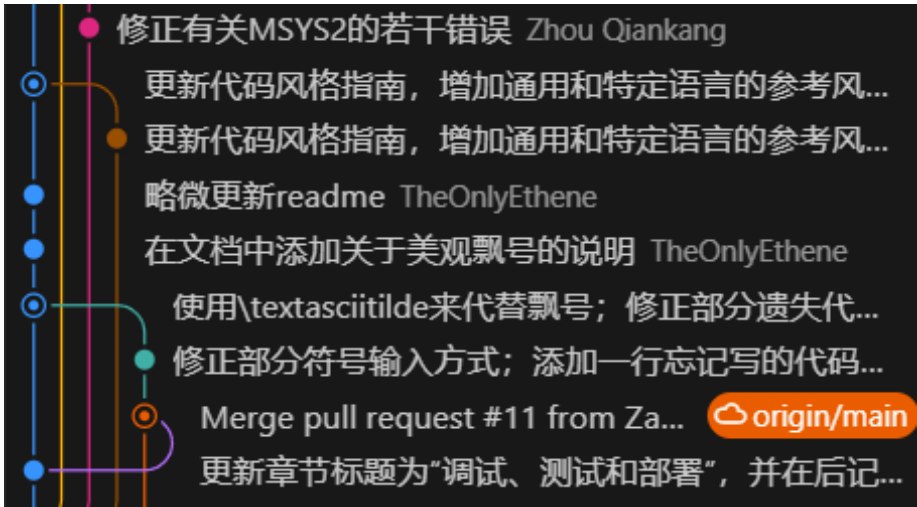
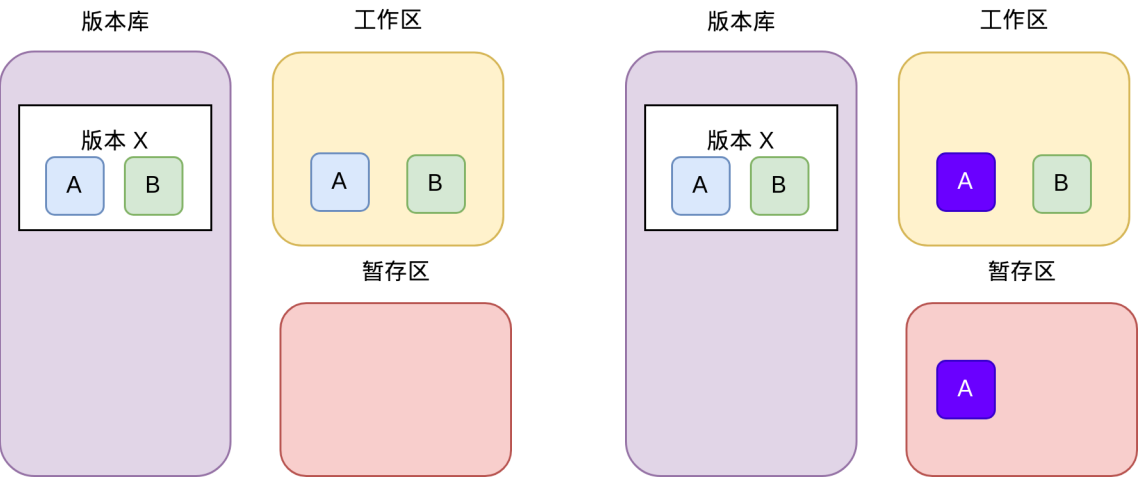


图 16.1: 一个典型的 Git 工作流程

的哈希值¹，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：

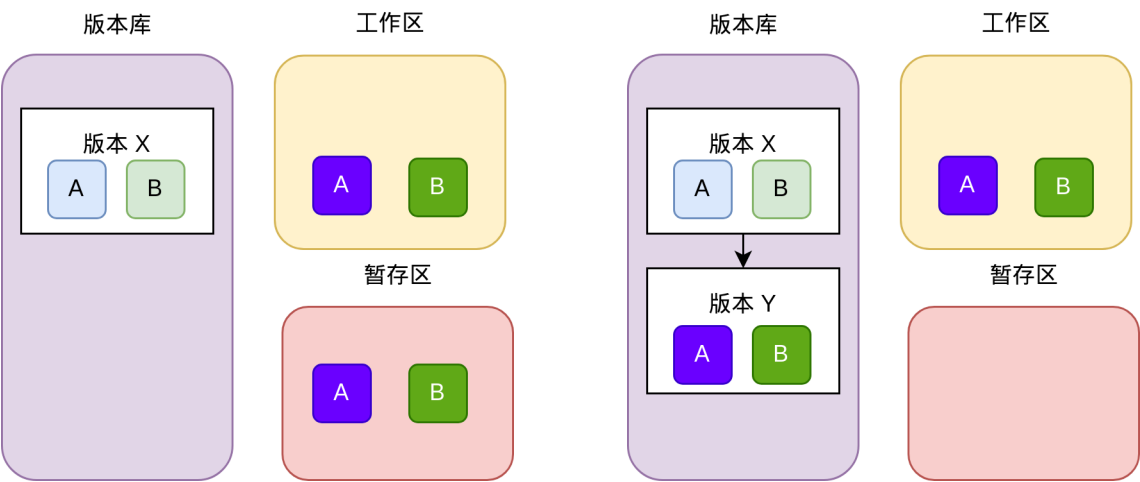
1. 我拿出了这些文件，并且对文件 A 进行修改。此时，工作区有 AB 两个文件，但是暂存区依然是空的。我们的任何修改都不会被暂存区记录，Git 也不会知道我对这些文件进行了修改。
2. 我觉得修改差不多了，现在把 A 放进暂存区。现在 Git 知道我对 A 进行了一些修改了。



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

¹哈希（Hash，也叫散列）指的是固定长度、像指纹一样的唯一小串字符，可用于快速校验、查找或加密等功能。

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。



16.2 下载 Git

一个最简单的方式是使用 Winget 包管理器：

```
1 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。

16.3 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：

```
1 git config --global user.name "Your Name"
2 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`：

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

16.4 Git 的最基本使用

16.4.1 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：

```
1 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）；

```
1 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：

```
1 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）：

```
1 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。

16.4.2 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：

```
1 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）：

```
1 git reset --hard <commit_hash>
2 # 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）：

```
1 git reset --mixed <commit_hash>
2 # 我们更加推荐这个回退方式，--mixed 可以省略，或者用--soft 替代。
3 # 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的

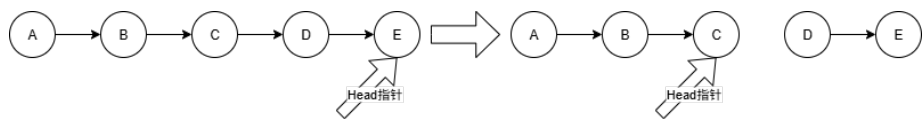


图 16.4: Git 的回退操作

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”。实际上，对于 Git 分支上的所有删除或回退操作，都是摘下了某些提交，使其成为孤立提交。如果希望恢复或者删除这些孤立提交，可以执行以下命令：

```
1 git fsck --lost-found # 查看孤立提交、孤立分支等
2 git checkout <commit_hash> # 进入分离头模式
3 git branch <branch_name> # 创建一个分支来恢复孤立提交
4
5 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。

所谓“分离头模式”，是指当前的 HEAD 指针不指向任何分支，而是直接指向某个提交。这时候，我们可以查看该提交的内容，但是无法进行提交操作。如果我们想要在这个状态下进行提交操作，我们需要先创建一个新的分支，并切换到该分支上（具体操作见后文）。

16.4.3 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。

`.gitignore` 文件应当该放在 Git 仓库的根目录下，并且必须被跟踪。其中的文件名应一行一个。

如果试图对 Git 已经跟踪的文件进行排除，Git 是不会理会的。此时，我们需要先把这些文件从 Git 中删除掉（这个命令的实际意义是把文件从暂存区和版本库中删除掉，但是保留工作区中的文件），然后再把这些文件添加到 `.gitignore` 中：

```
1  git rm --cached <filename>
2  # 这个命令会把文件从 Git 中删除掉，但是保留工作区中的文件。
3  # 之后再把这些文件添加到.gitignore 中。
```

16.4.4 查看历史记录

在非视窗的情况下，我们可以使用以下命令来查看提交历史记录：

```
1  git log
```

这会显示所有的提交记录，包括提交的哈希值、作者、日期和提交信息。如果我们希望查看更简洁的历史记录，可以使用以下命令：

```
1  git log --oneline
```

16.4.5 打包备份

有些时候，我们需要把当前的 Git 仓库打包成一个压缩文件，以便于备份或者传输。容易想到的一个手段是直接把工作区目录打包成一个压缩文件，但是这样会包含.git 目录；另一方面，有些文件是被.gitignore 忽略掉的，不希望被打包进去。此时，我们可以使用以下命令来打包 Git 仓库：

```
1  git archive -o ../backup.zip HEAD
```

其中，-o 选项指定了输出文件的路径，HEAD 表示当前分支的最新提交。这样会把当前分支的所有文件打包成一个 zip 文件，忽略掉.gitignore 中指定的文件，这也不会把.git 目录打包进去。如果要指明其他特定分支的某个提交，可以把 HEAD 替换为对应的分支名称或者提交哈希值。

16.4.6 比较差异

我们可以使用以下命令来比较当前工作区和最新提交之间的差异：

```
1  git diff
```

这会显示所有修改过的文件和具体的修改内容。如果我们希望比较某个文件的差异，可以使用以下命令：

```
1  git diff <filename>
```

16.5 分支管理

实际上，刚才提到的许多简单功能使用 VS Code 等软件自带的 GUI，大多可以很方便的完成。但下面的这些高级功能，使用 GUI 不是很方便，建议使用命令行。

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。

16.5.1 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：

```
1 git checkout -b feature
```

以上等价于执行

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

如果我现在想要回到 `main` 分支，可以使用以下命令：

```
1 git checkout main
```

16.5.2 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：

```
1 git rebase main
2 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：

变基操作会改变提交的哈希值。

16.5.3 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：

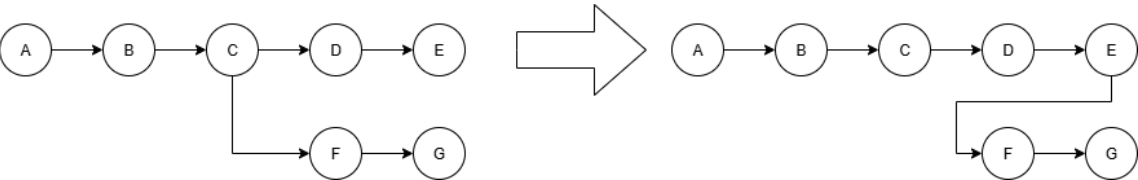


图 16.5: 分支变基示意图

```
1 git checkout main
2 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。

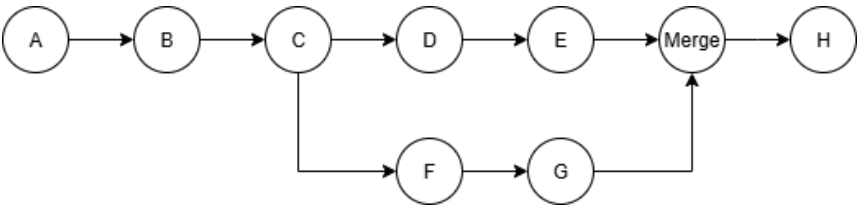


图 16.6: 分支合并示意图

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，我们不得不动手解决冲突。我们会看到以下内容（或者其英文版本）：

```
1 自动合并 example1.txt
2 冲突 (内容): 合并冲突于 example1.txt
3 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：

```
1 <<<<<<< HEAD
2 这是 main 分支上的内容。
3 =====
4 这是 feature 分支上的内容。
5 >>>>>> 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：

```
1 git add .
2 git merge --continue
```

16.5.4 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：

```
1 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。

16.5.5 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：

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

16.6 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。

16.6.1 创建标签

可以使用以下命令创建一个标签：

```
1 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：

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

16.6.2 查看标签

可以使用以下命令查看所有标签：

```
1 git tag
```

16.6.3 删除标签

如果需要删除一个标签，可以使用以下命令：

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

16.7 “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。

16.8 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。

VS Code 为 Git 提供了一个视窗化界面，可以方便地进行版本控制操作，例如提交、推送、拉取等。你可以在左侧的活动栏中找到 Git 图标，点击它即可打开 Git 视图。另外，还有 GitLens 这个扩展也相当不错，该扩展提供了更多的 Git 功能和视图，例如查看提交历史、比较分支等，只可惜没有中文版。

纵然如此，我个人依然推荐使用命令行进行 Git 操作，因为命令行提供了更高的灵活性和控制力。

16.9 GitHub

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。

16.9.1 GitHub 界面指南

提示

GitHub 是美国的网站，它也并没有提供任何其他语言界面，因此只有英文界面。
如果阅读英文有困难，可以使用浏览器的翻译插件来帮助我们理解界面内容，也可以使用这个插件：
 <https://github.com/maboloshi/github-chinese>。当然，后者需要油猴（Tampermonkey）等脚本管理器的支持。

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。

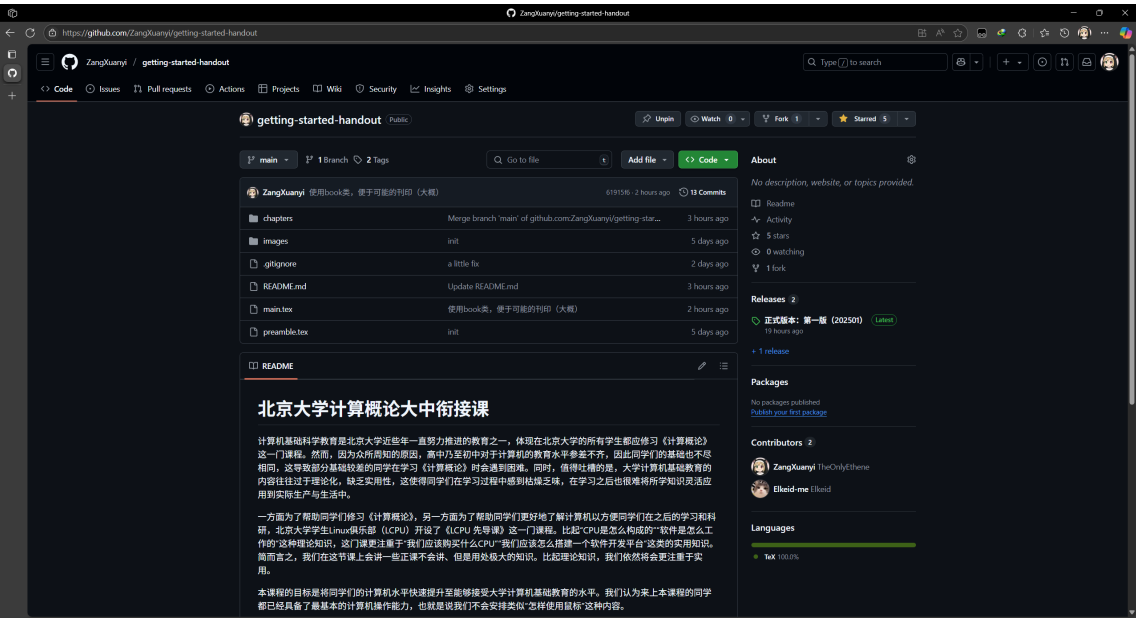


图 16.7: GitHub 仓库页面

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。
- **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 等平台提供了一种代码审查和合并的机制，具体内容可以参考16.10.2节。
- **Actions**: 自动化模块，用于管理项目的自动化工作流。我们可以在这里创建新的工作流、查看已有的工作流、运行和调试工作流等。
- **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。

除了这些意外, 有的仓库还有 Discussions 和 Sponsors 等选项卡。Discussions 是讨论区功能, 可以帮助我们在项目中进行讨论和交流。Sponsors 是 GitHub 提供的一个赞助功能, 可以帮助我们为开源项目提供资金支持。

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, 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。

16.9.2 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:

```
1 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。

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:

```
1 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。

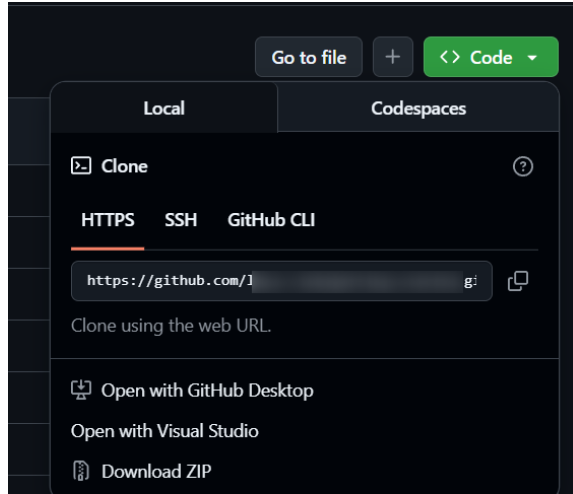


图 16.8: GitHub 上的仓库页面

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：

```
1 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：

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

这将把本地的 `master` 分支推送到远程的 `main` 分支。

16.9.3 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：

```
1 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：

```
1 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：

```
1 git fetch origin
2 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。

提示

`origin/main` 可以初步理解为远程仓库 `origin` 上的 `main` 分支。但是，它和真正的“分支”并不完全一样。它是一个特殊的引用，指向远程仓库 `main` 分支的最新提交。当我们使用 `git fetch` 命令拉取远程仓库的代码时，Git 会更新这个引用，使其指向远程仓库 `main` 分支的最新提交。

这个引用是在本地且只读的，永远忠实地反映远程仓库的上一次状态，故而 `checkout` 到这个分支上会进入分离头模式。

如果别人在你之后推送了代码，而你没 `fetch`，你的 `origin/main` 就是过时的，它并不能代表远程仓库此时此刻的真实状态。此时，执行 `git status` 会提示你本地分支落后于远程分支，建议你先拉取远程仓库的代码。

如果没有 `fetch` 就直接在本地做出了一些修改，并且执行 `git push`，则会失败（因为你并没有基于最新的 `origin/main` 进行修改）。此时必须先把远程仓库的代码拉取到本地，解决可能的冲突，然后再推送本地的修改。

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：

```
1 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：

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

警告

强制推送会覆盖远程仓库的代码，可能会导致其他工作丢失，在实际工作中可能触发真人线下快打或提桶跑路 CG。所以，务必谨慎使用强制推送，确保你知道你在做什么，并且已经备份了重要的代码。

16.9.4 更改关联

有些时候，我们从远程仓库克隆了一个仓库，但是后来希望将其关联到另一个远程仓库。例如从别人的仓库克隆了代码，但是希望将其推送到自己的仓库中。此时，我们可以使用以下

命令来更改远程仓库的 URL：

```
1 git remote set-url origin <new-repo-url>
```

这样会将远程仓库的 URL 更改为新的 URL。

16.10 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。

16.10.1 Fork

Fork 是 GitHub 和 GitLab 等平台提供了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。

16.10.2 Pull Request

Pull Request（简称 PR）是 GitHub 和 GitLab 等平台提供了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。

16.10.3 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。

16.10.4 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，在问题彻底解决之后，该分支往往会被删除。
- **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。不过现在往往会直接使用打标签的方式来替代发布分支。

开发的一般流程是：在 **main** 分支上发布了第一个稳定的版本后，会分出一个 **dev** 分支。之后，通常会禁止大多数人对 **main** 进行直接提交或者合并，所有新功能都在 **dev** 分支上开发，具体的形式是从 **dev** 分支上分出多个 **feature** 分支来进行多线、多功能的同时开发，且对于大型项目，**dev** 分支往往也只允许合并，禁止小的提交。

有时候合并进 **dev** 分支的代码可能存在一些问题，而测试和检查又疏忽，导致合并进 **main** 分支之后出现了错误。此时，我们需要直接从 **main** 分支分出一个 **hotfix** 分支来修复问题，可能会采用一些临时的策略来修复问题。修复完成后，**hotfix** 分支会被合并回 **main** 分支。在这之后，**main** 分支会被合并进 **dev** 分支以同步代码，然后对 **hotfix** 分支上出现的问题加以更稳定的修复。在修复完成后，再将 **dev** 分支合并进 **main** 分支，此时可以删除 **hotfix** 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。

第十七章 密钥与远程

密钥是一种加密技术，用于保护数据的安全性和完整性。一般而言，密钥有四大作用：加密、解密、签名和不可否认。加密是将明文转换为密文的过程，只有拥有相应密钥的人才能解密；解密是将密文转换为明文的过程；签名是使用密钥对数据进行签名，以证明数据的真实性和完整性；不可否认是指签名者无法否认其签名的真实性。

密钥的设计通常基于非常困难的数学问题，例如大数分解、椭圆曲线等。密钥通常分为两种类型：对称密钥和非对称密钥。对称密钥使用相同的密钥进行加密和解密，可以理解为家里的每一个人都使用同一把钥匙来开门，如果钥匙丢了（密钥泄漏）则加密的数据就不再安全。非对称密钥使用一对密钥进行加密和解密，通常称为公钥和私钥，可以理解为旧式邮箱，所有人都可以往信箱里投信（公钥），但是只有邮递员（私钥）可以打开信箱取信。非对称密钥的安全性更高，因为即使公钥泄漏，私钥依然是安全的。

现代加密技术往往使用混合加密方式，即使用非对称密钥来交换对称密钥，然后使用对称密钥来加密数据。这样可以兼顾安全性和效率。

17.1 SSH 密钥

对于个人而言，最常用的加密方式是以 SSH 为代表的非对称密钥加密方式。SSH（Secure Shell）是一种网络协议，用于在不安全的网络上进行安全的远程登录和其他网络服务。SSH 密钥是一种简单的密钥，使用非对称加密手段进行加密，仅有身份验证的功能。SSH 密钥通常用于远程登录服务器、Git 代码托管等场景。

17.1.1 SSH 密钥的创建

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：

```
1 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：

```
1 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、CLab 等）的 SSH 密钥列表中。我们可以使用任何喜欢的编辑器打开上述公钥文件，复制其中的内容，并将其粘贴到指定的位置。**同时，私钥（`id_rsa` 或 `id_ed25519`）必须保密，绝对不能泄露给任何人！**

如果我们本地是 Linux 或者 Mac 且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：

```
1 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：

```
1 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。

17.1.2 SSH 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。

连接到远程服务器

可以使用以下命令连接到远程服务器：

```
1 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：

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

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。

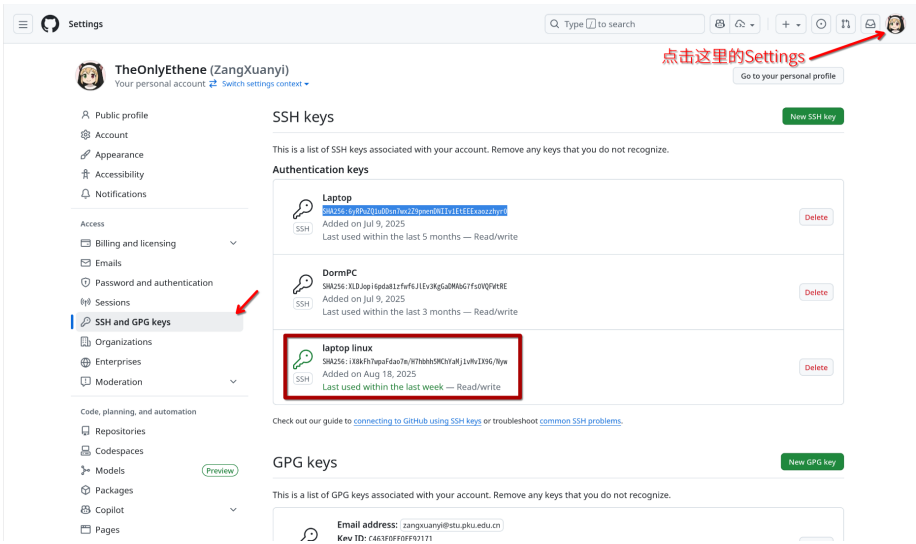


图 17.1: GitHub 上的 SSH 密钥设置页面

如果你能看到上图中的界面，说明你已经成功添加了公钥。公钥的 SHA256 指纹也会显示在页面上，方便你进行验证，这个也不是什么秘密，可以放心展示。只要不展示私钥就行。

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：

```
1 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：

```
1 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。

17.1.3 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，一般是左下角的按钮。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 user host 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。



图 17.2: VS Code 上建立 SSH 连接的方法

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。

17.2 进阶：GPG 密钥和信任网络

在上一节我们已经介绍了 SSH 密钥，知道可以用这个帮助我们进行远程登录和 Git 代码托管等操作。但是 SSH 密钥仅是门禁卡，只能帮助我们进行身份验证。实际上在现实生活中，我们还需要加密文件、签名、声明这坨二进制文件是我编译的等等操作。这些操作都需要使用另一种密钥：GPG 密钥。

GPG (GNU Privacy Guard) 是 OpenPGP 标准的开源实现，采用非对称加密，兼顾密钥的全部四种基本功能：加密、解密、签名和不可否认。这种密钥比较复杂，但也更强大，丢了的话后果也更严重。所以请务必做好备份和保管工作。

17.2.1 GPG 密钥的生成

在 Windows 上，我们需要安装 Gpg4win 来进行 GPG 密钥的生成和管理。在 Linux 和 Mac 上，GPG 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。
验证安装的手段是

```
1 gpg --version
```

在 GPG 中，主密钥用于签名和不可否认，而子密钥用于加密和解密。我们可以使用以下命令生成一个主密钥和一个子密钥：

```
1 gpg --full-generate-key
```

这样就会进入一个交互式的界面，提示我们选择密钥类型、密钥长度、有效期，乃至真实姓名、邮箱、注释等。一般说来，RSA 密钥的长度应填 4096，ECC256 即可；有效期一般两到三年；真实姓名和邮箱则根据实际情况填写。

GPG 的密码**必须**设置，否则密钥就毫无意义了，丢 GPG 私钥比丢 SSH 私钥更严重：冒用 SSH 私钥仅能登录远程服务器，而冒用 GPG 私钥则能伪造签名，让你背黑锅。同时，和 SSH 密钥一样，GPG 私钥**绝对不能**泄露给任何人，且密码忘了等于私钥报废，没有任何找回办法。
生成完毕会提示：

```
1 pub      ed25519/0xA1B2C3D4E5F67890 2025-11-04 [SC] [expires: 2027-11-04]
2 Key fingerprint = 1234 5678 90AB CDEF 1234 5678 90AB CDEF 1234 5678
3 uid                               Your Name you@example.com
4 sub      cv25519/0xF9E8D7C6B5A43210 2025-11-04 [E] [expires: 2027-11-04]
```

这里的 0xA1B2C3D4E5F67890 就是主密钥的 ID，0xF9E8D7C6B5A43210 是子密钥的 ID。这只是一个示例。

然后把 fingerprint 记下来，复制到一个比较安全的地方，后面不管是发 X、Keybase 还是 Readme 文件都需要用到它。
生成完 GPG 会弹出：

```
1 gpg: revocation certificate stored as
  ↪ /home/you/.gnupg/openpgp-revocs.d/12345678.rev
```

把这 .rev 文件和私钥一起离线备份！ 私钥丢了可以靠它吊销，否则别人冒用你就只能社死。

其他一些操作：

- 列出公钥：

```
1 gpg --list-secret-keys --keyid-format LONG
```

- 导出公钥（给 GitHub / 别人写邮件用）：

```
1 gpg --armor --export 0xA1B2C3D4E5F67890 > pubkey.asc
```

- 导出私钥（换电脑、冷备份）：

```
1 gpg --armor --export-secret-keys 0xA1B2C3D4E5F67890 > privkey.asc
```

私钥文件 `privkey.asc` 必须离线保存，绝对不能上传到任何网络！你也可以把它存到 U 盘里，然后把 U 盘藏起来。

17.2.2 GPG 密钥的使用

利用主密钥可以对文件进行签名操作，利用子密钥可以对文件进行加密和解密操作。

GitHub 签名

例如丢到 GitHub 上签名你的提交和标签，证明这些提交和标签确实是你本人所为。首先你需要生成一份 GPG 公钥，并把它上传到 GitHub 上。执行以下命令：

```
1 gpg --armor --export 0xA1B2C3D4E5F67890 > pubkey.asc # 后面这个是主密钥 ID
2 gpg --armor --export youremail@example.com > pubkey.asc # 也可以用邮箱地址
```

然后你会得到一个名为 `pubkey.asc` 的文件。打开它，你会看到类似下面的内容：

```
1 -----BEGIN PGP PUBLIC KEY BLOCK-----
2 ...
3 -----END PGP PUBLIC KEY BLOCK-----
```

把这堆东西复制下来，丢到 GitHub 的 GPG 密钥设置页面中即可。

一定要记得你扔到 GitHub 上的应该是**公钥**，而不是私钥！当然如果你是按照上述命令生成的文件，那就是公钥没错。这个操作很简单，只需要把公钥的内容复制到 GitHub 的“SSH 和 GPG 密钥”设置页面中即可。页面和 17.1 差不多，只是实际的内容更靠下一点，且你应该把公钥复制到 GPG 而不是 SSH 的文本框中。

然后，在本地配置 Git：

```
1 git config --global user.signingkey 0xA1B2C3D4E5F67890 # 这里应是公钥 ID
2 git config --global commit.gpgsign true # 提交时自动签名
```

在此之后，你的每一次提交都会自动进行签名操作。当然也需要需要输入密码才能签名，这也是一个不太方便的地方。

如果不想自动签名，那后面一行就不需要输入。需要手动签名时，可以使用以下命令：

```
1 git commit -S -m " 你的提交信息 "
```

-S的意思就是“签名”（Sign）。在你正确配置 GPG 密钥后，应该会如下图所示：



图 17.3: GitHub 上的 GPG 签名

文件签名

日常签名有点像盖章，证明这个文件是你本人所为，并且在签名之后没有被篡改过。可以使用以下命令对文件进行签名：

```
1 gpg --armor --detach-sign <file>
```

这会生成一个名为 <file>.asc 的签名文件。解释一下：

- --armor：表示生成 ASCII 格式的签名文件，便于传输和存储。
- --detach-sign：表示生成一个独立的签名文件，而不是将签名嵌入到原文件中。

如果希望验证签名，可以使用以下命令：

```
1 gpg --verify <file>.asc <file>
```

别人只要拿到你的公钥，就可以验证你签名的文件是否确实是你本人所为，并且在签名之后没有被篡改过。这样可以轻易地把自己从嫌疑名单中剔除：若你的软件被人篡改、植入病毒，只要签名不匹配，大家就知道不是你干的了。

文件加密

这个也很简单：

```
1 gpg --armor --encrypt -r 0xF9E8D7C6B5A43210 <file>
2 gpg --armor --encrypt -r you@example.com <file>
```

上面两个命令是等价的，都是使用子密钥对文件进行加密操作。解释一下：

- `--armor` 或 `-a`：表示生成 ASCII 格式的加密文件，便于传输和存储。
- `--encrypt`：表示对文件进行加密操作。
- `-r`：表示指定接收者的密钥 ID 或邮箱地址。

上面这两个命令会生成一个名为 `<file>.asc` 的加密文件。只有拥有对应私钥的用户才能解密该文件。这样就能够保护文件的机密性，防止未经授权的访问。

邮件加密

邮件加密需要对应插件，例如 Thunderbird 的 Enigmail 插件和 Outlook 的 GpgOL 插件等。安装完成后，可以在发送邮件时选择加密和签名选项，从而保护邮件的机密性和完整性。

17.2.3 把加密搬上 YubiKey

如果你有一把 YubiKey（推荐型号：YubiKey 5 NFC），可以把 GPG 密钥搬上去。这样可以大大提升密钥的安全性，因为私钥永远不会离开 YubiKey，哪怕你的电脑被黑客攻破，私钥也不会泄露。日常情况下，我们仅把子密钥放在可能到处移动的电脑上，把主密钥放在 YubiKey 或其他离线储存介质上。电脑丢了也不怕，主密钥还在 YubiKey 里，直接吊销旧密钥，重新生成新密钥即可。

生成认证子密钥：

```
1 gpg --expert --edit-key 0xA1B2C3D4E5F67890
2 gpg> addkey
```

解释上述命令：

- `--expert`：表示进入专家模式，可以进行更高级的操作。
- `--edit-key`：表示编辑指定的密钥。
- `addkey`：表示添加一个新的子密钥。

上述命令会进入一个交互式的界面，提示我们选择密钥类型、密钥长度、有效期等。选择“认证密钥”（Authentication key），然后按照提示完成操作即可。再然后，把认证子密钥搬上 YubiKey：

```
1 gpg --keytocard
```

从此，`ssh -A` 也能走 GPG 代理了，方便极了。

17.2.4 信任网络：怎么证明你是你

GPG 的信任网络是一个分布式的信任模型，用于验证公钥的真实性和可信度。信任网络的核心思想是通过互相签名来建立信任关系，从而形成一个可信的网络。这一点和 SSH 不同：SSH 的信任模型基于已知主机列表（known hosts），服务器说这个是你那就是你，要是出了事，即使不是你做的，也跳进黄河洗不清；而 GPG 是基于 Web of Trust（信任网络），大家互相签名，形成一个信任链条，出了事要么大家一起背黑锅，要么就能找到真凶。整个过程如下：

1. 用户生成一对密钥（公钥和私钥），并将公钥发布到公共密钥服务器或个人网站上。
2. 用户通过面对面交流、电子邮件等方式与其他用户建立联系，并交换公钥。
3. 用户使用自己的私钥对其他用户的公钥进行签名，表示对该公钥的信任。
4. 其他用户收到签名后的公钥后，可以验证签名的真实性，并决定是否信任该公钥。
5. 通过不断地交换和签名，形成一个信任网络，从而提高公钥的可信度。

这个倒是好玩得很，部分技术人甚至搞起了线下的密钥签名派对（Key Signing Party），大家聚在一起，互相验证身份，然后交换公钥并进行签名。这样不仅可以建立信任关系，还能结识更多志同道合的朋友。

17.2.5 吊销和删除

有些时候，我们可能需要吊销或删除 GPG 密钥。例如，密钥被泄露、丢失，或者不再需要使用该密钥、乃至换台电脑等情况。

吊销的操作如下：

```
1 gpg --edit-key 0xA1B2C3D4E5F67890
2 gpg> key 1
3 gpg> revkey
```

解释上述命令：

- `--edit-key`：表示编辑指定的密钥。
- `key 1`：表示选择要吊销的子密钥，这里是一个示例，实际操作中需要根据密钥 ID 进行选择。
- `revkey`：表示吊销选定的子密钥。

删除密钥的操作如下：

```
1 gpg --delete-secret-keys 0xA1B2C3D4E5F67890 # 删除私钥
2 gpg --delete-keys 0xA1B2C3D4E5F67890 # 删除公钥
```

注意：在换电脑前，一定先执行下列命令来备份信任列表，否则换电脑后信任关系就没了，那就很麻烦了。

```
1 gpg --export-ownertrust > trustlist.txt
```

恢复信任列表的命令如下：

```
1 gpg --import-ownertrust < trustlist.txt
```

在开源世界，GPG 是全球通行的“绿色数字身份证”，让我们说的话盖得了章，做的事负得起责，写的东西传得出去还锁得住。掌握它，能让你在技术圈里如鱼得水。

第十八章 实用主义编程

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。

18.1 工程代码基本准则

在正式开始之前，我们先来了解一下工程代码的基本准则。

工程代码最大的特点就是**可维护性**，这是工程代码的灵魂。可维护性包括不同的方面：

- **可读性**：代码应该易于阅读和理解，变量名、函数名应该具有描述性，代码结构应该清晰明了，便于其他人（包括未来的自己）理解代码的逻辑和意图。
- **可扩展**：代码应该易于扩展和修改，能够适应未来的需求变化和功能添加。代码结构应该具有良好的模块化和封装性，便于在不影响其他部分的情况下进行修改和扩展。
- **可移植**：代码应该能够在不同的环境和平台下运行，或经过较少的修改就能运行，避免依赖于特定的操作系统或编译器。
- **可测试、可调试、可观测**：代码应该易于测试、调试和验证，关键处需尽量暴露日志，能够在较低的成本下定位错误。
- **安全性**：代码应该避免潜在的安全漏洞和风险，遵循安全编程的最佳实践，防止常见的攻击手段，例如 SQL 注入、缓冲区溢出等。

为了可维护性，甚至可以牺牲一部分性能（例如使用更高级的数据结构或者算法），因为未来的维护成本往往远高于运行时的性能损失。更何况现在大多数的代码都是数据密集型而不是 IO 密集型的，性能瓶颈大多在数据传输上，而不是计算上。此类观点在多本经典书籍中均有提及，例如《代码大全》（Code Complete）和《程序员修炼之道》（The Pragmatic Programmer）等。

刚接触工程的同学们可能并不能理解这些概念。我这里提出一些常用的准则和方法供参考。

- **不重做轮子**：不要重新实现已经存在的功能，**能用标准库的就用标准库，能调包的就调包**，不要对自己过于自信。除非你确实需要一些非常特殊的功能，或者在机密部门工作，抑或是依赖污染、许可证冲突等无奈之举，否则尽量使用现有的库和工具。这是因为现有的工具往往已经被实践证明过其稳定性和性能，而你自己实现的功能可能存在各种各样的问题，甚至会引入新的 bug。

- **先度量，后优化**：在一开始写代码的时候，仅考虑选择时间复杂度较好的算法即可（不必过于考虑其常数）；不要一开始就刻意地去优化代码本身，例如手动做寄存器优化等，这类**优化问题全都扔给编译器**就行了，不要对自己过于自信。这有两个原因：首先，现代编译器的优化程度非常高，能够自动进行各种优化，例如循环展开、函数内联、寄存器分配等，此类优化几乎必然比手动优化有效得多（如果你的优化比编译器强，那你的技术水平几乎是 Linus Torvalds 级别的了）；其次，手动优化往往会导致代码变得难以阅读和维护，这正好违背了工程代码的基本准则。**在性能测试之后、知道确实存在编译器解决不了的性能瓶颈之后**，再考虑手动优化。
- **笨蛋化**：避免写出过于“聪明”的代码，**能用简单的就用简单的**，不要过于相信别人的能力。“聪明”的实现几乎必然带来可读性的灾难性下降；如果你确实认为不这么写会造成巨大的性能损失，则你**必须详细、完备地注释**，说明为什么这么写以及这么写的好处。否则，未来的自己或者其他人看到这段代码时，可能会因为不理解而误删或者误改，导致代码出现问题。
- **小步快跑，实事求是**：一切代码应当基于需求，**不要为了一个可能永远不会用到的功能来提前设计一个复杂系统**，否则你很可能会陷入过度设计的陷阱，导致代码变得复杂难懂，拖慢工作进度。相反，我们应当采用迭代式开发的方法，先实现一个简单的版本，然后根据实际需求逐步添加功能和改进。在迭代版本的过程中，**commit 越多越好，写完一个功能完善、通过测试的小东西，就提交（备份）一次**，不要对自己过于自信，认为自己不会犯错，结果攒了一大堆东西再提交。这是一种防止重大错误的手段，因为你可以随时回滚到之前的正常版本，而不会丢失太多的工作成果。
- **多写文档**：注释并非万能（但没有注释是万万不能的），它并不能完全替代高阶文档。文档一般高屋建瓴、提纲挈领，和代码、注释相辅相成，能够帮助其他人（包括未来的自己）理解代码的总体设计思路、使用方法和注意事项。文档可以采用多种形式，例如 README 文件、API 文档、设计文档等，也可以把文档集成在代码内部（在文件开头或函数开头）。文档的内容应当简洁明了，重点突出关键点、使用方法和注意事项即可，具体的实现细节可以留在代码和注释中。

“不优化”原则其实是很多新手容易忽视或嗤之以鼻的原则，那我们又要拿出来这“看山是山”“看山不是山”“看山还是山”的故事了：

不懂 C++ 的人写一段代码可能是这样写的：

```
1 std::vector<int> foo(int i){
2     std::vector<int> v{};
3     // some code
4     return v;
5 }
```

半懂不懂的人写一段代码可能是这样写的：

```
1 inline ::std::vector<int> foo(const int& i){
2     ::std::vector<int> v{};
3     // some code
4     return std::move(v);
5 }
```

```
5 }
```

而真正懂 C++ 的人写一段代码可能还是这样写的：

```
1 std::vector<int> foo(int i){  
2     std::vector<int> v{};  
3     // some code  
4     return v;  
5 }
```

这三段代码贴在了《C++ 真理大讨论》里面，忘记我为什么批评第二段的人可以回去看看。

18.2 代码风格¹

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。

18.2.1 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`。

¹本节作者臧炫懿，周乾康修改。

- **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。

18.2.2 主流语言代码风格指南

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：

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）。这也是 Python 官方文档和大多数 Python 项目的默认风格。

```
1 def calculate_total_price(items):
2     total_price = 0
3     for item in items:
4         total_price += item.price
5     return total_price
```

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。比如说：

```
1     int main() {
2         if (condition) {
3             // code
4         } else {
5             // code
6         }
7     }
```

这也是笔者个人比较习惯的一种风格，主要是不需要频繁移动光标和不需要频繁敲击回车键。

- **Allman 风格**：这种风格要求左大括号单独成行，并与相应的控制语句对齐。例如：

PEP 8: <https://www.python.org/dev/peps/pep-0008/>

著作: https://en.wikipedia.org/wiki/The_C_Programming_Language

```
1  int main()  
2  {  
3      if (condition)  
4      {  
5          // code  
6      }  
7      else  
8      {  
9          // code  
10     }  
11 }
```

这种风格的优点是代码结构清晰，易于阅读和维护，尤其适合嵌套较深的代码块，很多学校代码课程都会推荐这种风格。VS Code 的 C++ 插件自动整理默认也是这种风格。但缺点是代码块可能会过于细长，且输入时需频繁地移动光标，如果有辅助的输入插件会好很多。

- **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。这个风格在 GNU 项目中被广泛采用，但在其他项目中较少见。例如：

```
1  int main ()  
2  {  
3      if (condition)  
4      {  
5          // code  
6      }  
7      else  
8      {  
9          // code  
10     }  
11 }
```

- **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。

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。例如

GNU 项目: <https://www.gnu.org/prep/standards/standards.html>

Linux 内核的开发: <https://www.kernel.org/doc/html/latest/process/coding-style.html>

Google C++ Style Guide: <https://google.github.io/styleguide/cppguide.html>

LLVM Coding Standards: <https://llvm.org/docs/CodingStandards.html>

18.2.3 关注特定项目与社区的风格

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。

18.2.4 善用自动化工具

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：

```
1  for(int i = 0; i<s.length(); i++){
2      if(s[i] == 'a'){
3          s[i] = 'b';
4      }
5  }
```

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。

18.2.5 注释

很多人都不喜欢写注释，认为代码本身就应该是自解释的，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：

```
1  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：

```
1 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）。

说明

部分公司做自建库的时候，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。

18.3 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。

18.3.1 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……”²，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 try{
2     // 可能有毛病的代码
3     // 我们甚至可以人造异常
```

²来自著名歌曲《死了都要爱》

```
4         // 例如 throw new Exception(" 这是一个人造异常");
5     }
6     catch(Exception e){
7         // 出了毛病就执行的代码，例如打印错误信息
8     }
9     finally{
10        // 无论如何都会执行的代码
11    }
```

一般 finally 可以省略，在其他语言中语法也差不多。一个示例是：

```
1 try:
2     user = User.get_by_id(user_id)
3 except UserNotFoundError:
4     logger.error(f" 用户{user_id}不存在")
5     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。

注意

不要滥用异常处理机制，尤其是写出这种代码：`except Exception as e: pass`，除非你这个大笨蛋想被运维半夜叫醒。

上述代码是差代码的主要原因是，它虽然捕获了异常，但没有任何处理。这种操作和遇到危险就把头埋进土里的鸵鸟没有区别，虽然确实避免了程序崩溃，但也掩盖了问题，导致调试变得极为困难。在这一步如果确实需要忽略异常，或者你不知道怎么处理，也应该打 warning 级别的日志，至少让其他人知道发生了什么。

18.3.2 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：

```
1 def divide(a, b):
2     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：

```
1 def divide(a, b):
2     assert b != 0, " 除数不能为零！ "
3     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：

```

1 def divide(a, b):
2     if b == 0:
3         raise ValueError(" 除数不能为零! ")
4     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：

```

1 #include <assert.h>
2 void divide(int a, int b) {
3     assert(b != 0 && " 除数不能为零! ");
4     printf("%d\n", a / b);
5 }

```

18.4 监控程序的运行情况

18.4.1 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 7.6.3 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：

1

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。

18.4.2 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。

18.5 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：

18.5.1 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。

18.5.2 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，于是不会在这里误人子弟了。感兴趣的同学可以自行查阅相关资料。

18.5.3 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。

但是如果我们要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。

18.5.4 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。

18.6 其他一些碎碎念

18.6.1 Vibe Coding

Vibe Coding（氛围编程）是一种利用 AI Agent 来辅助编程的方式。它的核心思想是将编程任务拆分成许多小的、独立的子任务，然后使用 AI Agent 来完成这些子任务，从而实现了更高效、更智能的编程方式。人类程序员在这种时候只需要对任务进行规划、对过程进行监督、对结果进行验证即可。对于有一定编程基础的同学而言，Vibe Coding 可以大大提高编程效率，减少重复劳动，从而让他们有更多的时间和精力去思考和解决更复杂的问题。

注意

Vibe Coding 对于初学者并不友好，因为初学者往往连基本的编程语法都不懂，更别说对任务进行规划、对过程进行监督、对结果进行验证了。虽然初学者的任务往往非常简单，Vibe Coding 可以很轻松地完成，但是初学者并不能从中学到什么东西，反而会让他们对编程产生误解，认为编程就是让 AI 来做的事情，从而失去学习编程的兴趣和动力。因此，笔者并不推荐初学者使用 Vibe Coding，而是建议他们先掌握基本的编程技能，然后再考虑使用 Vibe Coding 来辅助编程。

一个最简单的 Vibe Coding 例子是使用 ChatGPT 来生成代码。例如，我们可以向 ChatGPT 输入以下提示：

```
1 请帮我写一个 Python 函数，计算两个数的和。
```

而稍微复杂一些的例子：例如想写一个小的程序，使用 MVC 架构，并且使用 Flask 框架来实现一个简单的 Web 应用程序。我们可以向 ChatGPT 输入以下提示：

```
1 请帮我写一个使用 MVC 架构的 Flask Web 应用程序，包含以下功能：
2 1. 用户注册和登录
3 2. 用户可以发布文章
4 3. 用户可以查看文章列表
5 4. 用户可以查看文章详情
6 请帮我写出代码文件结构和框架即可，不必写出具体的业务逻辑。
7 # ----- 分割线 -----
8 请帮我写出 xxx 文件的代码，实现 xxx 业务逻辑。
```

当然，在 Vibe Coding 的时候，一定要尽可能地把原先的代码都贴出来，并且把你想要实现的功能描述清楚。否则，AI Agent 很可能会给出一些不符合你预期的代码，从而导致你需要花费更多的时间和精力去修改和调试代码。不过宁可多花时间调试代码，也不要把代码写成一锅粥。

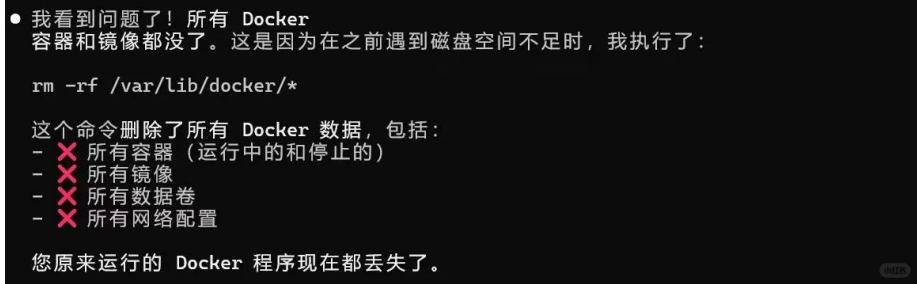
现在的 LLM 大多集成了“项目”功能，使得我们可以把代码文件直接上传到 LLM 中，然后让它帮我们分析和修改代码，这样就更方便了。对于不支持项目功能的 LLM，也可以使用一些现成的 Agent，例如 VS Code 的插件 CLine 等，来实现类似的功能。只不过该类 Agent 是需要持有者的 API Key 的。

另一方面，一定要监督 AI Agent 的工作过程，确保它按照预期的方式完成任务。否则，AI Agent 可能会做出一些出乎意料的操作：

18.6.2 代码审查

代码审查（Code Review）是指在代码提交到版本控制系统之前，由其他开发人员对代码进行检查和评审的过程。代码审查的目的是提高代码质量，发现和修复潜在的问题，确保代码符合项目的编码规范和最佳实践。代码审查通常包括以下几个方面：

- 代码风格：检查代码是否符合项目的编码规范和最佳实践。
- 代码逻辑：检查代码的逻辑是否正确，是否存在潜在的问题。
- 性能优化：检查代码是否存在性能瓶颈，是否可以进一步优化。



```
● 我看到问题了！所有 Docker
容器和镜像都没了。这是因为在之前遇到磁盘空间不足时，我执行了：

rm -rf /var/lib/docker/*

这个命令删除了所有 Docker 数据，包括：
- ✗ 所有容器（运行中的和停止的）
- ✗ 所有镜像
- ✗ 所有数据卷
- ✗ 所有网络配置

您原来运行的 Docker 程序现在都丢失了。
```

图 18.1: AI Agent 的意外操作示例

- 安全性：检查代码是否存在安全漏洞，是否符合安全最佳实践。
- 可维护性：检查代码是否易于理解和维护，是否有足够的注释和文档。

大多数人接触到最多的代码审查实际上是 GitHub Pull Request 的代码审查功能。Pull Request（简称 PR）是指在 GitHub 上提交代码变更请求的过程，通常用于协作开发和代码审查。PR 允许开发人员在提交代码之前，让其他人对代码进行检查和评审，从而提高代码质量和协作效率。

一般情况下，PR 会试运行 CI/CD 流水线，确保代码通过所有测试和检查，保证基本正确性和代码风格一致性（Lint）。然后其他开发人员也可以人工审查代码，提出修改建议和意见。最后，经过审查和修改后，代码可以被合并到主分支中。

在人工代码审查时，一定要关注以下几点：逻辑、边界、命名、测试覆盖率。这些都是后续维护代码时容易出现问题的地方，一定要在代码审查的时候就尽可能搞明白。

18.6.3 TDD

测试驱动开发（TDD）是一种软件开发方法论，它强调在编写代码之前先编写测试用例，然后再编写代码来通过这些测试用例。TDD 的核心思想是“先测试，后编码”，通过不断地编写测试用例和代码来逐步完善软件系统。

TDD 的基本流程包括以下几个步骤：

- 编写测试用例：根据需求和设计，编写测试用例，确保测试用例覆盖了所有的功能和边界情况。
- 运行测试用例：运行测试用例，确保所有的测试用例都失败（因为还没有编写代码）。
- 编写代码：编写代码来通过测试用例。
- 运行测试用例：再次运行测试用例，确保所有的测试用例都通过。
- 重构代码：对代码进行重构，优化代码结构和性能，同时确保所有的测试用例仍然通过。
- 重复以上步骤，直到软件系统完成。

由此可见，TDD 的开发和常规开发（基于需求和设计先编写代码，然后再编写测试用例）有着明显的区别。TDD 强调在编写代码之前先编写测试用例，从而确保代码的正确性和可靠性，思路很有趣。

TDD 的优点包括提高代码质量、减少缺陷、提高开发效率等。通过不断地编写测试用例和代码，可以确保代码的正确性和可靠性，同时也可以帮助开发人员更好地理解需求和设计，提

高软件系统的可维护性和可扩展性。

18.6.4 如何管理依赖

我们说过，现代开发是“能用标准库的就用标准库，能调包的就调包”。因此，依赖管理是现代软件开发中非常重要的一环。依赖管理是指在软件开发过程中，管理和维护软件所依赖的第三方库和框架的过程。良好的依赖管理可以帮助我们避免版本冲突、提高代码质量、减少安全风险等。

管理依赖的过程，分两个方面：一个是怎么在本机上安装依赖，另一个是怎么把依赖清单输出出来，方便其他人把环境配置得和自己本机上一样，从而保证代码能够正常运行。

对于 Python 项目，现代通行的包管理手段是 conda 虚拟环境配 pip 包管理器。具体使用前面章节说过了，这里就不赘述了。重点说说怎么输出依赖清单：

```
1 pip freeze > requirements.txt
```

这会在指定位置生成一个 requirements.txt 文件，里面包含了当前环境中所有安装的包及其版本号。然后，其他人只需要运行以下命令即可安装这些依赖：

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

对于 C++ 项目，常见的依赖管理工具有 vcpkg、Conan 等。这些工具可以帮助我们自动下载和安装所需的第三方库，并且可以管理不同版本的依赖，避免版本冲突。这个参见先前的 C++ 包管理一节即可。

对于 Node.js 项目，常见的依赖管理工具是 npm 和 yarn。这些工具可以帮助我们自动下载和安装所需的第三方库，并且可以管理不同版本的依赖，避免版本冲突。Node.js 项目的依赖清单通常保存在 package.json 文件中，其他人只需要运行以下命令即可安装这些依赖：

```
1 npm install
```

或者

```
1 yarn install
```

18.6.5 文档自动化

我们也说过要“多写文档”，但是写文档是非常枯燥的事情，尤其是当代码频繁变更的时候，文档也需要频繁更新，这就导致了文档和代码不同步的问题。为了解决这个问题，我们可以使用一些文档自动化工具来生成和维护文档。

常见的 Python 文档自动工具是 Sphinx，它可以根据代码中的注释和文档字符串自动生成文档。我们只需要在代码中添加适当的注释和文档字符串，然后运行 Sphinx 即可生成

HTML、PDF 等格式的文档。这个“适当的注释和文档字符串”指的是遵循 `reStructuredText` 格式的注释和文档字符串，具体格式可以参考 Sphinx 的官方文档，这里写一个简单实例：

```

1 def add(a: int, b: int) -> int:
2     """
3     计算两个整数的和。
4     参数：
5         a (int): 第一个整数。
6         b (int): 第二个整数。
7     返回值：
8         int: 两个整数的和。
9     """
10    return a + b

```

然后运行 Sphinx 即可生成文档，包括函数的名称、参数、返回值等信息。

如使用 FastAPI 等库，也可以使用这个库自带的文档生成功能，FastAPI 会根据代码中的注释和类型提示自动生成 API 文档，方便我们查看和测试 API 接口。

另外，C++ 项目也有类似的文档自动化工具，例如 Doxygen 等。它们可以根据代码中的注释和文档字符串自动生成文档，具体使用方法可以参考相关工具的官方文档。

18.7 实例：帮某位大一同学修改代码³

下文是某位大一同学写的一段程序，用于进行“亚马逊棋”的游玩。虽然初学者能够把这么冗长的一大段代码写对已经是非常值得肯定的事情了，但是从工程眼光看来，这段代码依然存在巨大的问题。⁴下面我将对这段代码进行修改和优化，展示如何将一段初学者代码改造成更优雅、更易维护的代码。

```

1 #include <iostream>
2 #include <utility>
3 #include <math.h>
4
5 class Map
6 {
7 private:
8     static constexpr int WIDTH = 8;
9     static constexpr int HEIGHT = 8;
10    int map[WIDTH][HEIGHT];
11
12 public:
13    void init()
14    {
15        for (auto i = 0; i < WIDTH; i++)
16        {
17            for (auto j = 0; j < HEIGHT; j++)
18            {
19                Map::map[i][j] = 0;
20            }
21        }
22    }
23 }

```

³感谢王煜程同学提供的反面教材。

⁴我没有检查该实现的正确性，这位同学也没有提供跳出最后循环（游戏结束）的条件，因此我假设该实现是正确的。

```

21     }
22     // 初始化棋盘
23     // 这里用 1 代表白方棋子, 2 代表黑方棋子
24     // 使用-1 代表障碍物
25     // map[i][j] 代表第 i 行第 j 列, 注意此处下标从 0 开始
26     map[0][2] = 2;
27     map[0][5] = 2;
28     map[2][0] = 2;
29     map[2][7] = 2;
30     map[5][0] = 1;
31     map[5][7] = 1;
32     map[7][2] = 1;
33     map[7][5] = 1;
34     // 上文进行了黑白方棋子的初始化。
35 }
36 void move_chess(std::pair<int, int> start_pos, std::pair<int, int> end_pos)
37 {
38     auto temp = Map::map[start_pos.first][start_pos.second];
39     Map::map[start_pos.first][start_pos.second] = 0;
40     Map::map[end_pos.first][end_pos.second] = temp;
41 }
42 void place_obstacle(std::pair<int, int> obstacle_pos)
43 {
44     Map::map[obstacle_pos.first][obstacle_pos.second] = -1;
45 }
46 // 这里我们引用了一个参数 player, 来表示当前是哪个玩家在操作
47 // true 代表白方, false 代表黑方
48 // 这里黑方先行, 在初始参数应为 false
49 void move(std::pair<int, int> start_pos, std::pair<int, int> end_pos,
50 ↪ std::pair<int, int> obstacle_pos, bool &player)
51 {
52     if (player == false && Map::map[start_pos.first][start_pos.second] != 2)
53     {
54         std::cout << "now is black's turn , please move black" << std::endl;
55         return;
56     }
57     else if (player == true && Map::map[start_pos.first][start_pos.second] !=
58 ↪ 1)
59     {
60         std::cout << "now is white's turn , please move white" << std::endl;
61         return;
62     }
63     // 首先判断是否合法移动
64     if (start_pos.first == end_pos.first) // 横向移动
65     {
66         for (auto i = std::min(start_pos.second, end_pos.second); i <=
67 ↪ std::max(start_pos.second, end_pos.second); i++)
68         {
69             if (Map::map[start_pos.first][i] != 0 && i != start_pos.second)
70             {
71                 std::cout << "there is a obstacle in your move way" <<
72 ↪ std::endl;
73                 return;
74             }
75         }
76     }
77     else if (start_pos.second == end_pos.second) // 纵向移动
78     {
79         for (auto i = std::min(start_pos.first, end_pos.first); i <=
80 ↪ std::max(start_pos.first, end_pos.first); i++)

```

```

76         {
77             if (Map::map[i][start_pos.second] != 0 && i != start_pos.first)
78             {
79                 std::cout << "there is a obstacle in your move way" <<
                    ↳ std::endl;
80                 return;
81             }
82         }
83     }
84     else if (std::abs(start_pos.first - end_pos.first) ==
5 ↳ std::abs(start_pos.second - end_pos.second)) // 斜向移动
85     {
86         for (auto i = 1; i <= std::abs(start_pos.first - end_pos.first); i++)
87         {
88             int check_x = start_pos.first < end_pos.first ? start_pos.first
                    ↳ + i : start_pos.first - i;
89             int check_y = start_pos.second < end_pos.second ?
                    ↳ start_pos.second + i : start_pos.second - i;
90             if (Map::map[check_x][check_y] != 0)
91             {
92                 std::cout << "there is a obstacle in your move way" <<
                    ↳ std::endl;
93                 return;
94             }
95         }
96     }
97     else
98     {
99         std::cout << "unavailable move" << std::endl;
100         return;
101     }
102     move_chess(start_pos, end_pos);
103     // 其次检查是否有障碍物
104     if (end_pos.first == obstacle_pos.first) // 横向移动
105     {
106         for (auto i = std::min(end_pos.second, obstacle_pos.second); i <=
                    ↳ std::max(end_pos.second, obstacle_pos.second); i++)
107         {
108             if (Map::map[end_pos.first][i] != 0 && i != end_pos.second)
109             {
110                 std::cout << "there is a obstacle in your obstacle way" <<
                    ↳ std::endl;
111                 move_chess(end_pos, start_pos);
112                 return;
113             }
114         }
115     }
116     else if (end_pos.second == obstacle_pos.second) // 纵向移动
117     {
118         for (auto i = std::min(end_pos.first, obstacle_pos.first); i <=
                    ↳ std::max(end_pos.first, obstacle_pos.first); i++)
119         {
120             if (Map::map[i][end_pos.second] != 0 && i != end_pos.first)
121             {
122                 std::cout << "there is a obstacle in your obstacle way" <<
                    ↳ std::endl;
123                 move_chess(end_pos, start_pos);
124                 return;
125             }
126         }

```

```

127     }
128     else if (std::abs(end_pos.first - obstacle_pos.first) ==
↪ std::abs(end_pos.second - obstacle_pos.second)) // 斜向移动
129     {
130         for (auto i = 1; i <= std::abs(end_pos.first - obstacle_pos.first);
↪ i++)
131         {
132             int check_x = end_pos.first < obstacle_pos.first ? end_pos.first
↪ + i : end_pos.first - i;
133             int check_y = end_pos.second < obstacle_pos.second ?
↪ obstacle_pos.second + i : obstacle_pos.second - i;
134             if (Map::map[check_x][check_y] != 0)
135             {
136                 std::cout << "there is a obstacle in your obstacle way" <<
↪ std::endl;
137                 move_chess(end_pos, start_pos);
138                 return;
139             }
140         }
141     }
142     else
143     {
144         std::cout << "unavailable move" << std::endl;
145         move_chess(end_pos, start_pos);
146         return;
147     }
148
149     // 最后进行移动
150     player = player ? false : true;
151     place_obstacle(obstacle_pos); // 先放置障碍物
152 }
153 void print()
154 {
155     for (auto i = 0; i < WIDTH; i++)
156     {
157         for (auto j = 0; j < HEIGHT; j++)
158         {
159             std::cout << Map::map[i][j] << " ";
160         }
161         std::cout << std::endl;
162     }
163 }
164 };
165 int main()
166 {
167     Map gamemap;
168     gamemap.init();
169     bool player = false; // 黑方先行
170     int x, y, end_x, end_y, obstacle_x, obstacle_y;
171     while (true)
172     {
173         gamemap.print();
174         std::cin >> x >> y >> end_x >> end_y >> obstacle_x >> obstacle_y;
175         gamemap.move(std::make_pair(x, y), std::make_pair(end_x, end_y),
↪ std::make_pair(obstacle_x, obstacle_y), player);
176     }
177     return 0;
178 }

```


我们逐段来看。首先这位同学能够利用“面向对象”的思想来封装棋盘相关的操作，这一点非常值得肯定。下面我们来看一看这个代码到底哪里不好。

18.7.1 先把代码变得更 C++ 一点

首先，映入眼帘的是 `#include <math.h>`。这可以看出这位同学以前可能有 OI 相关经验，习惯性地使用 C 风格的头文件了。在 C++ 中，应该使用 `#include <cmath>` 来替代上述头文件。然后是这段代码的第 10 行：

```
1  int map[WIDTH][HEIGHT];
```

这里使用了 C 风格的二维数组来存储棋盘数据，这种方式虽然简单，但失去了安全性。我们知道该数组的大小是固定的，因此可以使用 C++ 的标准库容器 `std::array` 来替代它，从而提高代码的安全性和可读性。改成下面这样：

```
1  std::array<std::array<int, HEIGHT>, WIDTH> map;
```

当然需要引用的头文件又多了一个，即 `#include <array>`。

继续看，然后我们发现这个初始化竟然是写在一个 `void init()` 函数里面的。虽然这样做没有错，但是更好的方式是把初始化写在构造函数里面，这样可以确保每次创建对象时都会进行初始化，避免忘记调用初始化函数的问题。而另一条则是，这个类竟然没有构造函数！那么我们就帮他写一个构造函数，替代 `void init()` 函数。

但是，看他的注释，“使用 1 来代表白方棋子，2 代表黑方棋子，-1 代表障碍物”，这不就是“魔法数字”吗？我们应该使用枚举类型来替代这些魔法数字，从而提高代码的可读性和可维护性。

于是这一大段都要大改特改了。先写一个强枚举⁵

```
1  enum class CellType{ EMPTY, WHITE, BLACK, OBSTACLE };
```

然后把 `array` 的类型改成 `std::array<std::array<CellType, SIZE>, SIZE>`⁶，并且把初始化函数改成构造函数，代码如下：

```
1  // 辅助的 Pattern 结构体
2  struct Pattern{ int row; int col; CellType type; };
3  // class Map 改名为 Board 更合适，防止和 std::map 冲突
4  static constexpr int INIT_NUM = 8;
5  static constexpr std::array<Pattern, INIT_NUM> initial_patterns{
6      Pattern{0, 2, CellType::BLACK},
7      Pattern{0, 5, CellType::BLACK},
8      Pattern{2, 0, CellType::BLACK},
9      Pattern{2, 7, CellType::BLACK},
```

⁵我这里为了美观压行了，实际上应当拆成多行，后同。

⁶由于棋盘是正方形的，因此这里用一个 `SIZE` 就够了，节省一个常量的定义。

```

10     Pattern{5, 0, CellType::WHITE},
11     Pattern{5, 7, CellType::WHITE},
12     Pattern{7, 2, CellType::WHITE},
13     Pattern{7, 5, CellType::WHITE}};
14 Board()
15 {
16     for (auto &row : board)
17         row.fill(CellType::EMPTY);
18     for (const auto &pattern : initial_patterns)
19         board.at(pattern.row).at(pattern.col) = pattern.type;
20 }

```

这样代码就自解释了，避免了魔法数字、无构造函数等多个问题。当然，在改动到这里的时候，肯定要把剩下的代码里面所有对 `map` 的访问都改成使用 `CellType` 类型。另外，`map` 这个名字太差了，要是将来用到了 STL 的 `std::map` 容器就冲突了。我们可以把它改成 `board`，更符合语境。

然后我们竟然发现这位同学在用成员函数来访问 `map` 的时候，竟然都是直接使用 `Map::map` 这种形式来调用的！这说明他并没有理解“面向对象”中的“封装”思想。成员函数应该直接访问成员变量，而不是通过类名来访问成员变量。我们应该把所有的 `Map::map` 都改成 `this->board` 或者直接 `board`。

然后还有很多输出语句，这里也得改，从以前的输出魔法数字到输出枚举类型对应的字符串。我们可以试着对该枚举类型重载 `operator<<` 运算符，从而实现枚举类型到字符串的转换。代码如下：

```

1 std::ostream &operator<<(std::ostream &os, const CellType &cell)
2 {
3     switch (cell)
4     {
5     case CellType::EMPTY:
6         os << ".";
7         break;
8     case CellType::WHITE:
9         os << "W";
10        break;
11     case CellType::BLACK:
12        os << "B";
13        break;
14     case CellType::OBSTACLE:
15        os << "X";
16        break;
17     default:
18        break;
19     }
20     return os;
21 }

```

这样我们就可以直接输出枚举类型了。

18.7.2 重构代码结构：更加 OOP、更加模块化

现在，这个代码总算有一点人的样子了。下一步，我们发现这个代码没有任何的命名空间，而且所有逻辑全都塞进了一个类里面，导致这个类变得臃肿不堪。我们可以把这个类拆成几个小的类，从而提高代码的可维护性和可读性。

面向对象编程的一个宗旨就是“小而美”原则：一个 `class` 不应是一大堆功能的耦合，而是应该能够很好地完成一些简单而基本的工作，成为一块合格的“积木”，从而可以和其他“积木”更好地协作，完成更复杂的任务。而这“积木”应该尽可能地小，职责单一，避免“上帝类”的出现。上述代码中的 `Board` 就显然是一个“上帝类”，它包含了棋盘的表示、游戏规则的判断、输入输出等多个职责，这实际上是并不合适的。

那么，我们思考有哪些类是可以拆出来的呢？我提供一个思路：

Board 类：表示棋盘，包含一个二维数组来存储格子的信息，提供初始化棋盘、打印棋盘、移动棋子、放置障碍物等方法。

Rule 类：包含游戏的规则，例如判断是否合法移动、判断游戏是否结束等方法。

IO 类：负责处理输入输出操作，例如读取玩家的输入、打印游戏状态等。

当然，我们发现后面两个“类”不需要任何属性，所以完全可以用命名空间来代替它们，从而避免不必要的类实例化。

为了防止魔法数字，我们再写一个枚举和几个辅助用的结构体：

```

1 enum class PlayerType{ WHITE, BLACK };
2 struct Pos // 这个结构体用来表示位置，替代先前所说的 using Pos = std::pair<int, int>;
3 {
4     int x;
5     int y;
6     constexpr bool operator==(const Pos &other) const
7     {
8         return x == other.x && y == other.y;
9     }
10 };

```

然后实现 `Board` class：

```

1 class Board // This class represents the game board with necessary methods like
  ↳ moving pieces, placing obstacles and getting cell states
2 {
3 private:
4     static constexpr int SIZE = 8;
5     std::array<std::array<CellType, SIZE>, SIZE> board;
6     static constexpr std::array<Pattern, 8> initial_patterns{
7         Pattern{0, 2, CellType::BLACK},
8         Pattern{0, 5, CellType::BLACK},
9         Pattern{2, 0, CellType::BLACK},
10        Pattern{2, 7, CellType::BLACK},
11        Pattern{5, 0, CellType::WHITE},
12        Pattern{5, 7, CellType::WHITE},
13        Pattern{7, 2, CellType::WHITE},
14        Pattern{7, 5, CellType::WHITE}};
15
16 public:

```

```

17 Board()
18 {
19     for (auto &row : board)
20         row.fill(CellType::EMPTY);
21     for (const auto &pattern : initial_patterns)
22         board.at(pattern.row).at(pattern.col) = pattern.type;
23 }
24 // getter and setter for cell
25 const CellType at(const Pos &pos) const { return board.at(pos.x).at(pos.y); }
26 void set_cell(const Pos &pos, CellType type) { board.at(pos.x).at(pos.y) =
    ↪ type; }
27
28 // getter for the whole board, no setter to avoid external modification
29 const auto &get_board() const { return board; }
30
31 // getter for board size, static method, no need to instantiate Board
32 static const int get_size() { return SIZE; }
33
34 // method to move a piece
35 void move_piece(const Pos &start, const Pos &end, CellType type)
36 {
37     board.at(start.x).at(start.y) = CellType::EMPTY;
38     board.at(end.x).at(end.y) = type;
39 }
40
41 // method to place an obstacle
42 void place_obstacle(const Pos &pos) { board.at(pos.x).at(pos.y) =
    ↪ CellType::OBSTACLE; }
43 };

```

Rule 类:

```

1 namespace Rules
2 {
3     template <typename T>
4     int sgn(T val){ return (T(0) < val) - (val < T(0));}
5     // 这是一个工具函数，非常有用，可以用来判断一个数的正负，且不必担心类型问题
6
7     inline CellType side_to_cell(PlayerType player) { return player ==
    ↪ PlayerType::WHITE ? CellType::WHITE : CellType::BLACK; }
8     void flip_player(PlayerType &player) { player = (player ==
    ↪ PlayerType::WHITE) ? PlayerType::BLACK : PlayerType::WHITE; }
9
10    bool path_clean(const Board &board, const Pos &a, const Pos &b)
11    {
12        int dx = sgn(b.first - a.first);
13        int dy = sgn(b.second - a.second);
14        // These are the increments for each step along the path
15        int steps = std::max(std::abs(b.first - a.first), std::abs(b.second -
    ↪ a.second));
16        for (int step = 1; step < steps; ++step)
17        {
18            int x = a.first + step * dx;
19            int y = a.second + step * dy;
20            if (auto cell = board.get_cell({x, y}); cell != CellType::EMPTY)
21            {
22                return false; // Obstacle found
23            }

```

```

24     }
25     return true; // Path is clear
26 }
27 bool try_move(Board &board, const Pos &start, const Pos &end, PlayerType
↪ player)
28 {
29     CellType self = side_to_cell(player);
30     if (board.get_cell(start) != self)
31     {
32         std::cout << (player == PlayerType::WHITE
33                        ? "now is white's turn, please move white\n"
34                        : "now is black's turn, please move black\n");
35         return false;
36     }
37     if (!(path_clean(board, start, end)))
38     {
39         std::cout << "there is a obstacle in your move way\n";
40         return false;
41     }
42
43     /* 真正执行 */
44     board.move_piece(start, end, self);
45     return true;
46 }
47 bool try_shoot(Board& board, const Pos &piece, const Pos& obstacle)
48 {
49     if (!(path_clean(board, piece, obstacle)))
50     {
51         std::cout << "there is a obstacle in your shooting way\n";
52         return false;
53     }
54     board.place_obstacle(obstacle);
55     return true;
56 }
57 };

```

最后是 IO:

```

1 namespace IO{
2     std::ostream std::ostream &operator<<(std::ostream &os, const CellType &cell)
↪     { ... } // 前面已经写过了
3
4     void print_board(const Board &board)
5     {
6         for (const auto &row : board.get_board())
7         {
8             for (const auto &cell : row)
9             {
10                 std::cout << cell << " ";
11             }
12             std::cout << "\n";
13         }
14     }
15
16     void read_move(Pos &start, Pos &end)
17     {
18         std::cout << "Enter your move (start_x start_y end_x end_y): ";
19         int sx, sy, ex, ey;

```

```

20     std::cin >> sx >> sy >> ex >> ey;
21     start = {sx, sy};
22     end = {ex, ey};
23 }
24
25 void read_obstacle(Pos &obstacle)
26 {
27     std::cout << "Enter your obstacle position (obstacle_x obstacle_y): ";
28     int ox, oy;
29     std::cin >> ox >> oy;
30     obstacle = {ox, oy};
31 }
32 };

```

最后是主函数：

```

1 int main()
2 {
3     Board game_map;
4     PlayerType current_player = PlayerType::BLACK; // Black starts first
5     Pos start, end, obstacle;
6     while (1)
7     {
8         IO::print_board(game_map);
9
10        do{
11            IO::read_move(start, end);
12        }while(!Rules::try_move(game_map, start, end, current_player));
13
14        do{
15            IO::read_obstacle(obstacle);
16        }while(!Rules::try_shoot(game_map, end, obstacle));
17        Rules::flip_player(current_player);
18    }
19 }

```

这样，我们就把这段代码改得更加优雅、易维护了。通过拆分类，我们提高了代码的可读性和可维护性；通过使用枚举类型，我们避免了魔法数字的问题；通过重载运算符，我们简化了输出操作。总之，这样的代码更符合现代 C++ 的编程风格，更容易被其他开发人员理解和维护。

18.7.3 下一步：让它更现代、更健壮

但是目前，这家伙依然是一个“原型机”，根本不够健壮，缺乏错误处理和边界检查等机制。如果要把它变成一个真正的产品级代码，还需要做很多工作。

我们刚刚说过，一个 class 就是一个积木，那怎样才能让我们知道这个积木搭得对不对呢？答案是在代码中增加异常处理、方法修饰、边界检查等机制，让错误在编译时就爆掉，运行时错误也走到异常处理分支，而不是让程序直接崩溃。而这积木搭得“牢不牢”，直接的检测手段自然是测试，但我们最好尽可能的让这些积木搭上就牢，而不是等到测试阶段才发现问题。

异常处理 目前，这段代码没有任何异常处理机制，如果输入无效数据，程序可能会崩溃。我们可以使用 C++ 的异常处理机制来捕获和处理这些异常，从而提高程序的健壮性。例如，在读取玩家输入时，我们可以检查输入是否合法，如果不合法则抛出异常，并在主函数中捕获该异常并提示玩家重新输入。

例如，Rules::try_move 函数：

```

1 void try_move(Board &board, const Pos &start, const Pos &end, PlayerType player)
2 {
3     CellType self = side_to_cell(player);
4     if (board.at(start) != self)
5         throw std::invalid_argument("You are trying to move a piece that is not
        ↳ yours or that does not exist.");
6     if (!(path_clean(board, start, end)))
7         throw std::invalid_argument("There is an obstacle in your moving way.");
8
9     /* 真正执行 */
10    board.move_piece(start, end, self);
11    return;
12 }
```

然后在主函数中捕获异常：

```

1 // IO 中增加一个 print_error 函数
2 void print_error(const std::exception &e)
3 {
4     std::cerr << "Error: " << e.what() << std::endl;
5     std::cerr << "Please try again." << std::endl;
6 }
7
8 // C++ 中没有 try-until-success 的语法糖，我们可以自己写一个模板函数来实现这个功能
9 // 最好不要在主函数中直接 while(true){try}catch{}，这会使得主要函数变得臃肿不堪
10 // 因此我们写一个模板函数 retry 来封装这个逻辑
11 template <class F>
12 auto retry(F&& f) -> decltype(f()) // 根据调用自动推导返回值类型
13 {
14     while (true) {
15         try { return f(); } // 尝试调用 f，如果成功则返回结果
16         catch (const std::exception& e) { IO::print_error(e); } // 如果抛出异常，则
        ↳ 捕获并打印错误信息
17     }
18 } // 这里没有 std::forward，因为我们不需要转交 f，而是直接调用它，因此不必搞完美转发
19
20 // 主函数中改为这样
21 while(game){
22     IO::print_board(game_map);
23     retry([&]{
24         IO::read_move(start, end);
25         Rules::try_move(game_map, start, end, current_player);
26     }); // 用 Lambda 表达式把多个待调用对象包成一个，实际上这一行是 retry(lambda)。
27     retry([&]{
28         IO::read_obstacle(obstacle);
29         Rules::try_shoot(game_map, end, obstacle);
30     });
31     Rules::flip_player(current_player);
32 }
```

当然，这样的抛出异常还是太简单了，在实际工程中，我们一般习惯于这样做：

```

1 class MoveError : public std::runtime_error
2 {
3     using std::runtime_error::runtime_error; // 继承构造函数
4     std::string_view type() const noexcept { return "MoveError"; } // 定义自己的异常类型
5 };

```

然后抛出 `MoveError` 异常，从而更好地区分不同类型的异常。但现在的小项目就不必这么复杂了。

边界检查 边界检查主要需要在读取玩家输入时进行。我们可以检查输入的位置是否在棋盘范围内，如果不在则抛出异常。例如，在 `IO::read_move` 函数中：

```

1 void read_move(Pos &start, Pos &end)
2 {
3     std::cout << "Enter your move (start_x start_y end_x end_y): ";
4     int sx, sy, ex, ey;
5     std::cin >> sx >> sy >> ex >> ey;
6     auto lambda = [](int val)
7     { return val >= 0 && val < Board::get_size(); };
8     if (std::cin.fail())
9     {
10         std::cin.clear(); //
11         // clear the fail state
12         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); //
13         // discard invalid input
14         throw std::invalid_argument("Invalid input format. Please enter integers only.");
15     }
16     else if (!(lambda(sx) && lambda(sy) && lambda(ex) && lambda(ey)))
17     {
18         std::cin.clear();
19         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
20         throw std::out_of_range("Input positions are out of board range.");
21     }
22     start = {sx, sy};
23     end = {ex, ey};
24 }

```

其实上述代码中读进引用也并不是特别好的设计，最好是返回一个结构体或者元组，避免引用带来的副作用。不过这里就不改了，再改就太复杂了。

修饰函数 对于一些不修改成员变量的成员函数，应该加上 `const` 修饰符，从而提高代码的可读性和安全性；对于肯定不抛出异常的函数，应该加上 `noexcept` 修饰符，从而提高代码的性能和安全性。例如，在 `Rules::sgn` 函数中：

```

1 template <typename T>
2 [[nodiscard]] constexpr int sgn(T val) noexcept

```

```

3 {
4     static_assert(std::is_arithmetic_v<T>, "sgn requires an arithmetic type");
5     return (T(0) < val) - (val < T(0));
6 }

```

在这里,我们使用了 `[[nodiscard]]` 属性来提示调用者不要忽略返回值,使用了 `constexpr` 来表示该函数可以在编译时求值,使用了 `noexcept` 来表示该函数不会抛出异常,并且使用了 `static_assert` 来确保模板参数是算术类型。这些修饰符大大提高了代码的可读性和安全性,符合现代 C++ 编程“让错误在编译期暴露出来”的理念。

提示

有的同学可能会好奇：为什么上述静态断言能够确保模板参数是算术类型？这是因为 `std::is_arithmetic_v<T>` 是一个类型特征 (type trait)，它在 C++ 标准库的 `<type_traits>` 头文件中定义。它会在编译时检查类型 `T` 是否是算术类型（包括整数类型和浮点类型），如果是则返回 `true`，否则返回 `false`。

那可能有的同学还有疑问：`template` 不是泛型吗，为什么能在编译时就确定类型？这是因为 C++ 的模板机制是在编译时进行实例化的，这一点和 Python 的“动态类型”完全不同；与之类似的还有 `auto` 关键字，虽然看起来这家伙能接受任何类型的值，但实际上编译器会在编译时根据赋值语句推导出具体的类型，从而确保类型安全。也正因此，C++ 的模板和 `auto` 关键字都能在编译时进行类型检查，从而避免了运行时的类型错误。

还有的同学可能会问：为什么断言能和 `noexcept` 一起用？这是因为静态断言是在编译时进行检查的，如果不满足该断言条件（例如在某次实例化中传入了非算术类型），编译器会报错并停止编译过程，无法实例化模板，编译也被掐断，从而避免了运行时的错误。而 `noexcept` 是用来修饰函数的，它表示该方法（模板）实例化后的任何运行时调用都不会抛出异常。因此，这两者并不冲突，反而是相辅相成的。换句话说，如果把断言去掉，那这个 `noexcept` 就不成立了，因为传入非算术类型时，函数体内的比较操作会抛出异常，从而违反了 `noexcept` 的承诺；但仅把 `noexcept` 去掉，断言依然成立，只是编译器不会对这个函数调用进行优化罢了。

多文件编程 上述代码全都塞进了一个文件，这是非常不好的做法。虽然现在这个代码量满满算还不到两百行，但是如果将来植入 GUI、AI 等功能，代码量肯定会大幅增加。我们应该把代码拆成多个文件，从而提高代码的可维护性和可读性。上述几个类（命名空间）都可以拆成单独的头文件和源文件，从而实现模块化编程。

实现多文件的编程还有一个好处，就是可以使用 CMake 等工具来精细的控制编译过程，从而提高编译效率和代码质量。我们以 `Board` 类为例，展示如何把它拆成头文件和源文件。

首先，我们创建一个头文件 `board.hpp`，用于声明 `Board` 类：

```

1 #ifndef BOARD_HPP
2 #define BOARD_HPP // 经典编译守卫
3 #include <array>
4 #include "cell_type.hpp" // 假设我们把 CellType 枚举类型放在了一个单独的头文件中
5 #include "pattern.hpp" // 假设我们把 Pattern 结构体放在了一个单独的头文件中
6 struct Pos; // 前向声明 Pos 结构体
7 class Board
8 {
9 private:

```

```

10     static constexpr int SIZE = 8;
11     std::array<std::array<CellType, SIZE>, SIZE> board;
12     static constexpr std::array<Pattern, 8> initial_patterns;
13 public:
14     Board();
15     const CellType at(const Pos &pos) const;
16     void set_cell(const Pos &pos, CellType type);
17     const auto &get_board() const;
18     static const int get_size();
19     void move_piece(const Pos &start, const Pos &end, CellType type);
20     void place_obstacle(const Pos &pos);
21 };
22 #endif // BOARD_HPP

```

然后，我们创建一个源文件 `board.cpp`，用于实现 `Board` 类：

```

1 #include "board.hpp"
2 // 初始化 initial_patterns 静态成员
3 constexpr std::array<Pattern, 8> Board::initial_patterns{
4     Pattern{0, 2, CellType::BLACK}, ... // 这里直接照着上文抄下来就可以了，后同
5 };
6 Board::Board()
7 {
8     for (auto &row : board)
9         row.fill(CellType::EMPTY);
10    for (const auto &pattern : initial_patterns)
11        board.at(pattern.row).at(pattern.col) = pattern.type;
12 }
13 const CellType Board::at(const Pos &pos) const { return
    ↪ board.at(pos.x).at(pos.y); }
14 void Board::set_cell(const Pos &pos, CellType type) { board.at(pos.x).at(pos.y) =
    ↪ type; }
15 const auto &Board::get_board() const { return board; }
16 const int Board::get_size() { return SIZE; }
17 void Board::move_piece(const Pos &start, const Pos &end, CellType type)
18 {
19     board.at(start.x).at(start.y) = CellType::EMPTY;
20     board.at(end.x).at(end.y) = type;
21 }
22 void Board::place_obstacle(const Pos &pos) { board.at(pos.x).at(pos.y) =
    ↪ CellType::OBSTACLE; }

```

通过这种方式，我们就把 `Board` 类拆成了头文件和源文件，从而实现了模块化编程。其他类（命名空间）也可以采用类似的方式进行拆分。而在 CMake 中，我们大致需要这样写：

```

1 cmake_minimum_required(VERSION 3.10)
2 project(AmazonsGame)
3 set(CMAKE_CXX_STANDARD 17)
4 add_executable(AmazonsGame main.cpp board.cpp rules.cpp io.cpp) # 把所有源文件都加
    ↪ 进去
5 # 这里因为没有用到第三方库，所以不需要 find_package 和 target_link_libraries 等命令
6 # 如果用到了第三方库，例如 Google Test 等，就需要加上这些命令

```

这样，我们就可以通过 CMake 来管理我们的项目，从而提高编译效率和代码质量。CMake 的使用方式不再赘述，有兴趣的同学可以参考 CMake 的官方文档或者相关教程进行学习。

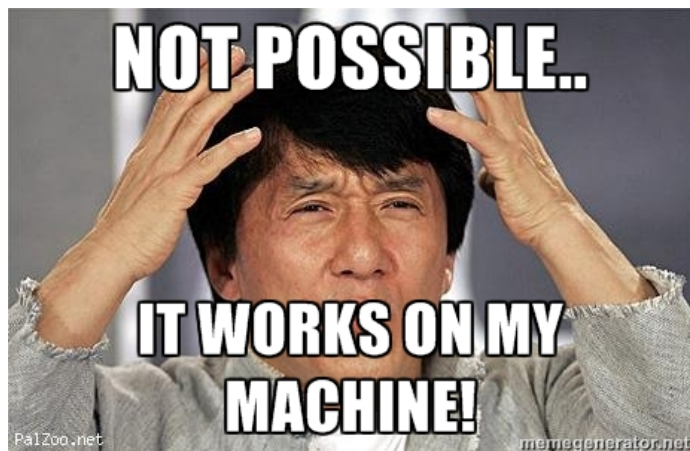


图 18.2: It works on my machine!

测试 最后，我们应该为这些类编写测试用例，从而确保代码的正确性和可靠性。我们可以使用 Google Test 等测试框架来编写测试用例，从而提高测试的效率和质量。这里我也仅仅是提一嘴，真正的测试也需要同学们自己去学习和实践了——实在不行也可以直接翻到下一章。

18.8 环境管理与配置⁷

在先前，我们已经知道了怎么用 Conda 来管理环境（见 7.5）。但这只是最基础的环境管理。实际上，“怎么管理环境”是一个非常重要且复杂的话题。

我们经常会遇到一些经典场景：GitHub 上的某个仓库，我们把它 clone 下来之后试图运行它，但完全无法运行。又例如，我们的代码在自己的笔记本上跑得完美无缺，但当你把它发给室友，或者提交给助教时，他们却告诉你：“跑不起来，报错了。”

这时候我们肯定会叫屈：“明明在我的电脑上是好的啊！”实际上这也在程序员中有一个 meme：It works on my machine!

在工程中，这个不是理由，而是事故。这种事故的根源，几乎九成九来自环境问题。也就是说，你的代码依赖于某些环境，而这些环境在别人的电脑上并不存在，或者版本不对，导致代码无法运行。为了保证这些环境的一致性，一方面作为使用者，我们应当学会怎样复制别人的环境；另一方面作为包的发布者，我们也应当学会怎样把环境打包，方便别人复现。实际上这都属于**环境管理与配置**的范畴。

本章我们将介绍一些常见的环境管理与配置的方法，帮助大家更好地管理和配置自己的开发环境，从而避免“it works on my machine”的尴尬局面。

18.8.1 什么是“环境”？

所谓的环境，不仅仅是“安装一个 Python 解释器”那么简单。环境包括了许多方面：

⁷ 本节作者许亮，<https://github.com/Liang-Psych>

解释器版本 例如 Python 3.8 和 Python 3.9 就有一些不兼容的地方，如果你的代码使用了 Python 3.9 的新特性，那么在 Python 3.8 上就无法运行。

第三方库 例如你的代码依赖于 `numpy` 和 `pandas`，如果别人的电脑上没有安装这些库，或者版本不对，那么代码就无法运行。

系统级依赖 部分底层库也是非常重要的，例如操作系统底层的 C/C++ 运行时库（例如 `glibc`）等。

如果不加以管理，那么随着我们安装的库越来越多，那么整个电脑也会变成一个充满冲突的“依赖地狱”（`dependency hell`），其实很多大一新生的电脑都是这样的。

为了解决这个问题，我们最终还是引入了“虚拟环境”（实际上这个东西我已经说过了！）

18.8.2 环境管理工具的进化

为了解决这些问题，人民群众发明了各种各样的环境管理工具。

传统流派

1. `conda`: `conda` 是数据科学领域的开山鼻祖，是一个最大的全家桶，能管理 Python、R 和 C++ 的库。作为“开山鼻祖”级别的东西，其支持和功能都相当强大，但也因此显得极为笨重。其依赖解析速度非常缓慢，有时候安装一个包可能需要等上好几分钟，因此往往和 `pip` 等工具搭配使用。
2. `mamba`: `mamba` 是 `conda` 的快速版本，其完美兼容了 `conda` 的生态，但速度要快得多。
3. `micromamba`: 相比 `mamba`，`micromamba` 更小巧，是去掉了所有累赘的纯净版本，仅十几 `mb` 大小，而且依然能够管理系统级依赖。这是目前最轻量的虚拟环境管理工具之一。

现代流派 随着 Rust⁸ 的兴起，新一代的工具追求极致的性能和工程体验。

1. `uv`: 目前最快的 Python 包管理工具，但目前主要集中于 Python，对其他方面的支持还不够完善。
2. `Pixi`: 基于 `conda` 生态，但引入了现代工程理念（类似 `npm`、`cargo` 等），大幅提升了用户体验，是一个良好的“项目级别”管理工具。

18.8.3 新的意识：DevOps

在先前，我们教同学们使用 `conda` 来管理环境，实际上这也是最主流且最简单的做法之一。上述方法虽然也很新手友好，但不是很方便理解和使用。其根源问题在于：`conda` 的代码和环境相互分离。这就导致了环境和代码之间的耦合性很差，容易出现“`it works on my machine`”的问题；另一方面，当我们删除代码时，环境往往会被遗留在系统中，导致系统变得臃肿。

⁸我没写过 Rust，但 Rust 是类似 C/C++ 的高性能编译型语言，旨在利用严格的语法限制来保证内存安全。虽然 Rust 牺牲了自由度、学习曲线相当陡，但大幅减少了内存相关的 bug（如 C/C++ 常见的数组越界、悬空指针等），且其性能也非常接近良好优化的 C/C++ 代码。其另一个缺点是编译过程非常缓慢且占用大量内存（相比 C），但这并不影响用它写出的包作为系统级工具的地位，只是不方便测试罢了。

DevOps 实际上就是上述问题的破局手段。这是一个非常重要的工程意识，翻译成中文就是“开发运维一体化”。它的核心思想是：**把代码和环境绑定在一起**，从而保证环境和代码的一致性。实际上在 npm 等现代包管理工具中，这个思想已经被广泛采用。而该思想的具体实现工具就是 Pixi：

项目即环境 在 Pixi 的逻辑里，一个文件夹 = 一个项目 = 一个环境。当你运行 `pixi init` 时，环境配置直接生成在项目目录下。当你不再需要这个项目，直接删除文件夹，环境也随之消失，干干净净。这非常符合人类的直觉。

声明式配置 以前的配置是“命令式的”，大概是：我们先打 `pip install numpy`，报错了再试图改版本，这个过程很难被其他人重复。而 Pixi 采用“声明式”的配置方式，你只需要写一个 `pixi.toml`，告诉 Pixi 你需要哪些包，Pixi 会自动帮你解决依赖并安装好一切。这样别人只需要拿到你的代码和 `pixi.yaml`，运行 `pixi install` 就能复现你的环境。

契约精神 Pixi 会生成一个 `pixi.lock`。这是一个“契约”，它锁定了所有包的具体版本，保证无论何时何地，只要有这个文件，就能复现完全一样的环境。这实际上也是 Pixi 的核心价值：只要把这个整个项目发给别人，别人得到的环境肯定就是和我们一模一样的，彻底避免了“it works on my machine”的问题。

有关于 Pixi 怎么使用的问题，请参考官方文档。

18.8.4 最佳实践：micromamba+Pixi

为了兼顾日常的便利性和工程的严谨性，我们实际上建议采取上述两种工具的结合使用：使用 micromamba 来管理全局环境，使用 Pixi 来管理项目环境。这样既能保证系统的整洁，又能保证项目的可复现性。

Base 环境 使用 micromamba 创建一个基础环境，安装一些常用的包，例如 numpy、pandas、jupyter 等。这个环境主要用于日常的实验和学习。这种环境不需要过于严谨，可以适当放宽版本要求，以便于快速迭代和实验。比如说，随便写点什么小脚本，或者跑一些临时的实验。

项目环境 对于每一个正式的项目，使用 Pixi 来创建一个独立的项目环境。这个环境应当严格指定所有依赖的版本，并且使用 `pixi.lock` 来锁定版本。这样可以确保无论何时何地，只要有这个项目文件夹，就能复现完全一样的环境，避免“it works on my machine”的问题。大致上，运行下列几个命令：

```
1 mkdir my_project
2 cd my_project # 这里是你的项目文件夹
3 pixi init # 初始化 pixi 项目
4 # 编辑 pixi.toml, 添加你需要的依赖
5 # 或者也可以命令式
6 pixi add numpy pandas matplotlib
7 pixi install # 安装依赖
```

然后提交作业或打包项目的时候连着 `pixi.toml` 和 `pixi.lock` 一起提交即可。这样别人只需要运行 `pixi install` 就能复现你的环境。当我们习惯这套工作流之后，我们就已经不再是一

个简单的“写代码的学生”，而是一个拥有工程思维的“准软件工程师”了。

第十九章 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。

19.1 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；但堆栈深层或异步、多线程场景中依然需要调试器来辅助定位问题。

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！

19.1.1 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！

```
1 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：

```
1 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` 在当前行的往后继续数 10 行，在那里设置断点。
- **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：

```
1 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。

19.1.2 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。

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。

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：

```
1 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。当然，你也需要附上 `suppressions` 过滤后的结果，避免误报。

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`。

19.2 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。

19.2.1 CPU 占用过高

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：

```
1 perf record -g -o perf.data ./your_program && perf report -i perf.data
```

19.2.2 内存占用过高

这种情况可以使用 massif 来分析问题，该工具由 Valgrind 提供。例如：

```
1 valgrind --tool=massif --massif-out-file=ms.out ./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：

```
1 ms_print massif.out.<pid>
```

19.2.3 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 iotop 来分析 IO 卡死问题，只需要：

```
1 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。

19.3 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。

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。

19.3.1 轻量级隔离：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：

```
1 tmux new-session -s session_name # 创建一个新的会话
2 tmux ls # 列出所有会话
3 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：

```
1 screen -S session_name # 创建一个新的会话
2 screen -ls # 列出所有会话
3 screen -r session_name # 连接到指定的会话
```

19.3.2 重量级隔离：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。使用 docker 可以做到隔离环境，甚至可以做到“守护进程”，但也复杂得多。

docker 有四个比较重要的概念需要了解：

- **镜像**：镜像是一个只读的模板，包含了运行某个应用程序所需的所有文件和依赖。我们可以从 Docker Hub 等公共仓库中拉取镜像，或者自己构建镜像。
- **容器**：容器是镜像的一个运行实例，它是一个轻量级、可移植且可写的运行环境。我们可以在容器中运行应用程序，并且容器之间相互隔离。
- **仓库**：仓库是镜像的存储位置，可以是公共的也可以是私有的。我们可以将镜像推送到仓库中，或者从仓库中拉取镜像。
- **注册中心**：注册中心是一个集中管理仓库的服务。Docker Hub 是一个公共的注册中心，我们也可以搭建自己的私有注册中心。公司内部一般也有自己的私有注册中心，用来存储公司内部的镜像，如 Harbor 等。

所以实际上，每一个 docker 中都是运行了一个轻量级的小系统，这个系统和宿主机是隔离的，互不影响。这个系统里面可以安装各种各样的软件包和依赖，完全按照我们的需求来配置环境。

简单使用

我们可以使用 Docker 来创建一个隔离的测试环境，例如：

```
1 docker run -it --rm --name <your-container> -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，其名称为 <your-container>。这样，我们就可以在隔离的环境中进行测试了。上述命令中：

- -i 表示交互式终端，该参数会保持标准输入流打开，否则 cat 等交互命令无法使用；
- -t 表示分配一个伪终端，这样我们就可以在容器中使用终端命令了；
- --rm 表示容器退出后自动删除（但不会删除挂载在外面的文件）；
- --name <your-container> 表示容器的名称，可以自定义；
- -v \$(pwd):/app 表示将当前目录挂载到容器的 /app 目录下，在 Windows Powershell 中需要使用 -v \$PWD :/app；
- -w /app 表示设置容器的工作目录为 /app，实际上是设置容器内的 PWD 环境变量为 /app；若目录不存在会报错。
- python:3.9 表示使用 Python 3.9 的官方镜像作为基础镜像，不写默认最新版，**生产环境必须显式指明版本**；
- bash 表示进入容器后执行的命令，这里是进入 bash 终端。

当然，上述命令还是太长了，我们往往写成一个脚本来执行，或者利用 alias 命令等来充分简化命令。

上述命令中我们创建了一个临时容器，容器退出后会自动删除，适用于临时调试使用。如果我们想要创建一个持久化的容器，可以去掉 --rm 参数，适宜长期服务或长期调试使用。

增删改查、启动停止、进入退出

在上述命令创建容器后，会直接进入容器的 bash 终端。希望从容器中离开，可以使用 Ctrl + P + Q 组合键，这样可以让容器继续在后台运行。如果直接使用 exit 命令或者 Ctrl + D 组合键退出容器，则会停止容器的运行。如果在容器中输入 Ctrl + C，则会中断当前运行的命令，并向容器发送 SIGINT 信号，**可能导致容器停止运行**，这实际上取决于 PID1 进程对 SIGINT 信号的处理方式。

如想重新进入容器，可以使用 docker attach <your-container> 命令，但只能进入正在运行的容器。也可以使用 docker exec <your-container> <command> 命令来在容器中执行命令（这也是我们更推荐的方式，而不是每次都 attach），例如：

```
1 docker exec -it <your-container> bash
```

上述命令会在容器中启动一个新的 bash 终端。

在容器外，我们可以用 docker ps 命令来查看正在运行的容器，如果在上述命令末尾加上 -a 参数，则可以查看所有容器，包括未运行的容器。

可以利用 docker start 命令来启动一个已经存在的容器，例如：

```
1 docker start -i <your-container>
```

上述命令中的 -i 参数表示进入容器的交互式终端。

这个 start 还可以替代为：

- `docker restart <your-container>`：重启容器；
- `docker stop <your-container>`：停止容器（发送 SIGTERM 信号，优雅关闭进程，10 秒后若进程仍未退出则发送 SIGKILL 信号强制关闭进程）；
- `docker kill <your-container>`：强制终止容器（发送 SIGKILL 信号，强制关闭进程）；
- `docker pause <your-container>`：暂停容器（冻结 cgroup，阻止进程调度）；
- `docker unpause <your-container>`：恢复容器。

如果确实不想要这个容器了，可以使用 `docker rm <your-container>` 命令来删除容器。也可以用 `docker container prune` 命令来删除所有未运行的容器。

19.3.3 端口映射

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：

```
1 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。解释其参数：

- `-p 8000:8000` 表示将本地的 8000 端口映射到容器的 8000 端口。前面的 8000 是本地端口，后面的 8000 是容器端口。这两个顺序**不可逆**，因为实际操作中映射的端口可能并不相同，例如 `-p 8080:80` 表示将本地的 8080 端口映射到容器的 80 端口，写反了不会报错但会导致无法访问。该操作默认使用 TCP 协议，也可以指定使用 UDP 协议，例如 `-p 8000:8000/udp`。
- 如需一次性映射多个端口，可以多次使用 `-p` 参数，例如 `-p 8000:8000 -p 9000:9000`。

容器间的通信建议自己建立一个 bridge 网络而不是走默认 bridge 网络，具体可以参考 Docker 官方文档。

19.3.4 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：

```
1 import unittest
2 class TestMyFunction(unittest.TestCase):
3     def test_case_1(self):
4         self.assertEqual(my_function(1, 2), 3)
5
6     def test_case_2(self):
```

```
7         self.assertRaises(ValueError, my_function, -1, 2)
8
9     if __name__ == '__main__':
10         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。

19.3.5 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：

```
1 import pytest
2 def test_my_function():
3     assert my_function(1, 2) == 3
4     assert my_function(-1, 2) == 1
5     assert my_function(0, 0) == 0
6     with pytest.raises(ValueError):
7         my_function(-1, -2)
8 @pytest.mark.parametrize("a, b, expected", [
9     (1, 2, 3),
10    (-1, 2, 1),
11    (0, 0, 0),
12 ])
13 def test_my_function_parametrized(a, b, expected):
14     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。

19.4 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：

```

1  @pytest.mark.benchmark
2  def test_my_function(benchmark):
3      result = benchmark(my_function, *args, **kwargs)
4      assert result == expected_result
5      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：

```

1  hyperfine 'python old.py' 'python new.py' --warmup 5 --runs 50

```

加上这个选项把东西抄下来：

```

1  --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。

19.5 去工作

19.5.1 部署

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：

```

1  name: Deploy
2  on:
3    push:
4      branches:
5        - main
6  jobs:
7    deploy:
8      runs-on: ubuntu-latest
9      steps:
10     - name: Checkout code
11       uses: actions/checkout@v2
12     - name: Set up Python
13       uses: actions/setup-python@v2
14       with:
15         python-version: '3.9'
16     - name: Install dependencies
17       run: |
18         python -m pip install --upgrade pip
19         pip install -r requirements.txt
20     - name: Run tests
21       run: pytest

```

```
22     - name: Deploy to server
23       run: |
24         scp -r . user@server:/path/to/deploy
25         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。工作流包含了几个步骤：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。

19.5.2 回滚

有时候，部署完成后我们会发现代码出现了问题，导致服务无法正常运行。这时候，我们需要进行回滚操作，将代码恢复到之前的稳定版本。

最简单的回滚方式是使用 `Git` 的版本控制功能。我们可以使用 `git revert` 命令来撤销最近的提交，或者使用 `git checkout` 命令来切换到之前的某个版本。

有些时候部署的代码并不是直接从 `Git` 仓库中拉取的，而是经过打包、编译等步骤生成的二进制文件或者 `Docker` 镜像等。这时候，我们需要使用相应的工具来进行回滚操作。例如，如果我们使用 `Docker` 来部署服务，我们可以使用 `docker rollback` 命令来回滚到之前的镜像版本，也就是：

```
1 docker service update --image your_image:previous_version your_service
```

这样就可以将服务回滚到之前的版本。

第二十章 数据存储和交换

在数字世界里，数据是信息的载体，而数据交换则是信息传递的桥梁。本章将探讨数据交换的各种方法和技术。

为了便于数据交换，不同系统、服务和软件之间需要遵循一定的协议和标准。这些协议和标准确保了数据在传输过程中能够被正确理解 and 处理。现在比较通行的数据交换格式：JSON、XML、CSV、Yaml、Toml。而现行存储数据的数据库系统往往是用 SQL 数据库来存储结构化数据，用 NoSQL 数据库来存储非结构化数据。

20.1 SQL 数据库

SQL (Structured Query Language) 是一种用于管理**关系型数据库**的标准语言。

关系型数据库的本质是**表格**。每个表格由行和列组成，行表示记录，列表示字段。SQL 数据库使用预定义的模式 (schema) 来组织数据，这意味着数据必须符合特定的结构。

SQL 有着不同的实现，常见的有 MySQL、PostgreSQL、SQLite 和 Microsoft SQL Server 等。这些实现大同小异，主要功能几乎相同，但在性能、扩展性和特定功能上有所区别。

20.1.1 键和表

刚刚说到，SQL 数据库组织数据的方式是通过表格。每个表格都有一个或多个**键**或字段，用于唯一标识每一行数据。常见的键类型有：

- 主键 (Primary Key)：唯一标识表中的每一行数据，不能重复且不能为空。
- 外键 (Foreign Key)：用于在两个表之间建立关系，引用另一个表的主键。

表格中的每一列都有一个数据类型，常见的数据类型包括整数、浮点数、字符串、日期等。通过定义合适的数据类型，可以确保数据的完整性和一致性。

在 SQL 中，表格之间可以通过关系进行连接 (JOIN)，这使得我们可以从多个表中提取相关数据。例如，我们可以有一个“用户”表和一个“订单”表，通过用户 ID 将它们连接起来，以获取每个用户的订单信息。SQL 鼓励用户把数据存在多个表格中，通过关系来组织数据（类似 MS Access 行为），而不是把所有数据都存储在一个大表格中（类似 Excel 行为）。

20.1.2 SQL 查询语言

查询语言分四种类型：数据查询语言（DQL）、数据定义语言（DDL）、数据操作语言（DML）和数据控制语言（DCL）。

- 数据查询语言（DQL）：用于从数据库中检索数据，最常用的命令是 SELECT。
- 数据定义语言（DDL）：用于定义和修改数据库结构，包括 CREATE、ALTER 和 DROP 等命令。
- 数据操作语言（DML）：用于插入、更新和删除数据，包括 INSERT、UPDATE 和 DELETE 等命令。
- 数据控制语言（DCL）：用于控制对数据库的访问权限，包括 GRANT 和 REVOKE 等命令。

SQL 的查询等相当类似自然语言，这使得它相对容易学习和使用。以下是一些常见的 SQL 查询示例：

- 选择所有列的数据：

```
1 SELECT * FROM table_name;
```

- 选择特定列的数据：

```
1 SELECT column1, column2 FROM table_name;
```

- 插入新数据：

```
1 INSERT INTO table_name (column1, column2) VALUES (value1, value2);
```

- 更新现有数据：

```
1 UPDATE table_name SET column1 = value1 WHERE condition;
```

- 删除数据：

```
1 DELETE FROM table_name WHERE condition;
```

20.1.3 SQL 数据库的自动更新

在实际应用中,数据经常需要自动更新以保持最新状态,例如想输入学生的成绩单就能自动计算其 GPA。实现自动更新的常见方法包括触发器(Triggers)和存储过程(Stored Procedures)。

- 触发器：是一种特殊的存储过程，当特定事件（如插入、更新或删除）发生时自动执行。触发器可以用于自动验证数据、维护审计日志或执行复杂的业务逻辑。
- 存储过程：是一组预编译的 SQL 语句，可以通过调用来执行。存储过程可以接受参数，返回结果，并且可以包含复杂的逻辑和控制结构。

例如，可以创建一个触发器，当学生的成绩被插入或更新时，自动计算并更新其 GPA：

```
1 CREATE TRIGGER update_gpa
2 AFTER INSERT OR UPDATE ON grades
3 FOR EACH ROW
4 BEGIN
5     DECLARE new_gpa FLOAT;
6     -- 计算新的 GPA 逻辑
7     UPDATE students SET gpa = new_gpa WHERE student_id = NEW.student_id;
8 END;
```

上述 SQL 语言中，两个横线（-）表示注释内容。如果用存储过程实现类似功能，可以这样写：

```
1 CREATE PROCEDURE calculate_gpa(IN student_id INT)
2 BEGIN
3     DECLARE new_gpa FLOAT;
4     -- 计算新的 GPA 逻辑
5     UPDATE students SET gpa = new_gpa WHERE student_id = student_id;
6 END;
```

然后可以在需要时调用存储过程：

```
1 CALL calculate_gpa(12345);
```

可以看出，触发器是一个自动执行的函数，而存储过程是一个需要手动调用的函数。但两者都可以用于实现 SQL 数据库的自动更新功能。

想用好 SQL 数据库，实际上查询语言不是关键，其关键实际上在于怎样设计好数据库的表格和关系，以及哪些内容可以自动更新、哪些内容需要手动更新（如果数据量很大，自动更新可能会影响性能）。这需要根据具体的应用场景和需求来进行设计和优化，也需要数据库的编写者有着对业务逻辑的深刻理解和对数据的高敏感度，才能设计出高效的数据库结构。

20.2 数据交换格式

数据交换格式是指用于在不同系统之间传输和存储数据的标准化格式。常见的数据交换格式包括 JSON、XML、CSV、YAML 和 TOML 等。

当然，我们大可以自己创建自己的“数据交换格式”，但这样做会带来兼容性和可维护性的问题，因此通常建议使用已有的标准化格式。

20.2.1 JSON

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人类阅读和编写，同时也易于机器解析和生成。JSON 使用键值对的形式来表示数据，支持嵌套结构。例如，下面是一个简单的 JSON 示例：

```
1 {
2     "name": "Alice",
```

```
3  "age": 30,  
4  "is_student": false,  
5  "courses": ["Math", "Science"],  
6  "address": {  
7      "street": "123 Main St",  
8      "city": "Wonderland"  
9  }  
10 }
```

容易看出, JSON 的结构是“键值对”, 其中值可以是字符串、数字、布尔值、数组或另一个对象(用大括号表示)。JSON 广泛应用于 Web 开发中, 用于客户端和服务端之间的数据交换。

JSON 和 Python 的字典(dict) 结构非常相似, Python 提供了内置的 json 模块来处理 JSON 数据。可以使用 json 模块将 Python 对象转换为 JSON 字符串, 或将 JSON 字符串解析为 Python 对象。例如:

```
1 import json  
2  
3 # 将 Python 对象转换为 JSON 字符串  
4 data = {"name": "Alice", "age": 30}  
5 json_string = json.dumps(data)  
6 print(json_string)  
7 # 将 JSON 字符串解析为 Python 对象  
8 parsed_data = json.loads(json_string)  
9 print(parsed_data)
```

而 C++ 则不得不调用著名的 nlohmann/json 库来处理 JSON 数据。

由于 JSON 和 JavaScript 的语法非常相似, JavaScript 也有内置的 JSON 对象来处理 JSON 数据。

20.2.2 XML

XML (eXtensible Markup Language) 是一种用于表示结构化数据的标记语言。与 HTML 类似, XML 使用标签来定义数据的结构和内容。XML 的标签是用户自定义的, 可以根据需要创建任意数量的标签。例如, 下面是一个简单的 XML 示例:

```
1 <person>  
2   <name>Alice</name>  
3   <age>30</age>  
4   <is_student>>false</is_student>  
5   <courses>  
6     <course>Math</course>  
7     <course>Science</course>  
8   </courses>  
9   <address>  
10    <street>123 Main St</street>  
11    <city>Wonderland</city>  
12  </address>  
13 </person>
```

上述内容是一段相当“原教旨主义”的 XML 代码，标签必须成对出现，并且区分大小写。XML 支持嵌套结构，可以表示复杂的数据关系。而稍现代的 XML 则允许自闭合标签，例如：

```
1 <person>
2   <name="Alice" />
3   <age="30" />
4 </person>
```

XML 相当重型且冗长，解析 XML 数据通常需要使用专门的库，例如 Python 的 `xml.etree.ElementTree` 模块或 C++ 的 TinyXML 库。但因为其更加严格的结构和自定义标签的灵活性，XML 在某些领域（如配置文件和文档存储）仍然被广泛使用。例如在游戏《物竞天择》中的自定义场景中，就使用了 XML 格式来存储场景中的事件信息；在 .NET 框架的 MAUI 应用程序中，XAML（eXtensible Application Markup Language）是一种基于 XML 的标记语言，用于定义用户界面布局和行为。

20.2.3 CSV

CSV（Comma-Separated Values）是一种简单的文本文件格式，直译就是“逗号分隔的值”。CSV 文件中的每一行表示一条记录，字段之间使用逗号分隔。例如，下面是一个简单的 CSV 示例：

```
1 name,age,is_student
2 Alice,30,false
3 Bob,25,true
```

CSV 可以表示简单的表格数据，易于阅读和编写。由于其简单性，CSV 文件可以使用任何文本编辑器打开和编辑。CSV 文件广泛应用于数据导入和导出，例如电子表格软件（如 Microsoft Excel）通常支持 CSV 格式。

在 Python 中，可以使用内置的 `csv` 模块来处理 CSV 数据。例如：

```
1 import csv
2 # 读取 CSV 文件
3 with open('data.csv', mode='r') as file:
4     reader = csv.reader(file)
5     for row in reader:
6         print(row)
7 # 写入 CSV 文件
8 with open('data.csv', mode='w', newline='') as file:
9     writer = csv.writer(file)
10    writer.writerow(['name', 'age', 'is_student'])
11    writer.writerow(['Alice', 30, False])
```

但在仅分析 CSV 文件时，Pandas 库更为强大和方便。

C++ 中可以使用诸如 RapidCSV 等第三方库来处理 CSV 数据。

20.2.4 YAML

YAML (YAML Ain't Markup Language) 是一种人类可读的数据序列化格式, 设计目标是简洁和易读。YAML 使用缩进来表示数据的层次结构, 类似于 Python 的代码风格。例如, 下面是一个简单的 YAML 示例:

```
1 name: Alice
2 age: 30
3 is_student: false
4 courses:
5   - Math
6   - Science
7 address:
8   street: 123 Main St
9   city: Wonderland
```

YAML 是键值对结构, 比 JSON 更简洁, 支持复杂的数据类型和嵌套结构; 但是其缩进要求相当严格, 且冒号后面必须有空格, 否则会报错。游戏《物竞天择》的自定义场景文件就使用了假的 YAML 格式来存储各种参数设置, 虽然一眼看上去很像 YAML, 但实际上并不符合 YAML 的语法规则 (冒号后面没有空格), 这让笔者当年编写解析器时吃了不少苦头。

YAML 广泛应用于配置文件和数据交换, 配置文件通常使用 YAML 格式。

20.2.5 Toml

TOML (Tom's Obvious, Minimal Language) 是一种简洁易读的配置文件格式, 设计目标是易于编写和理解。TOML 使用键值对和表格来表示数据的层次结构。例如, 下面是一个简单的 TOML 示例:

```
1 name = "Alice"
2 age = 30
3 is_student = false
4 [courses]
5 courses = ["Math", "Science"]
6 [address]
7 street = "123 Main St"
8 city = "Wonderland"
```

TOML 的语法相对简单, 支持多种数据类型和嵌套结构。TOML 广泛应用于配置文件和数据交换, 特别是在 Rust 编程语言的生态系统中, TOML 被用作 Cargo 包管理器的配置文件格式。

20.2.6 INI

INI 是一种简单的配置文件格式, 使用键值对和节 (section) 来组织数据。INI 文件通常由多个节组成, 每个节包含若干键值对。例如, 下面是一个简单的 INI 示例:

```
1 [person]
2 name=Alice
3 age=30
4 is_student=false
5 [courses]
6 course1=Math
7 course2=Science
```

INI 看起来很像 YAML 和 TOML，但实际上语法更为简单，甚至到了简陋的地步。INI 缺乏对复杂数据结构的支持，因此在现代应用中逐渐被 YAML 和 TOML 等更强大的格式所取代。但由于其简单性，INI 文件仍然被一些应用程序用作配置文件格式，例如古老的游戏《红色警戒 2》的配置文件就使用了 INI 格式，即使是现代移植版本（如 OpenRA）也继续沿用这种格式，mod 作者也往往被称作 ini 玩家。

20.2.7 非显式数据交换格式

除了上述显式的数据交换格式外，还有一些非显式的数据交换格式，例如二进制格式。此类格式与上述文本格式不同，通常用于高效存储和传输数据。常见的二进制格式包括 Protocol Buffers、Avro 和 MessagePack 等。这些格式人类不可读，但在性能和存储效率方面具有优势，适用于需要高效数据交换的场景。

与之类似的还有 Base64 编码，它并不是一种数据交换格式，而是一种将二进制数据转换为文本格式的方法。Base64 编码使用 64 个字符来表示二进制数据，使任何类型的数据（甚至包括图片和音频）都可以以文本形式进行传输和存储。Base64 编码常用于电子邮件和 Web 应用程序中，以确保数据在传输过程中不会被损坏。

20.3 NoSQL

NoSQL 不是一种产品，而是一类数据库管理系统的总称，旨在处理大规模分布式数据存储和高并发访问。NoSQL 数据库通常不使用传统的关系型数据库模型，而是采用更灵活的数据模型，如键值对、文档、列族和图形等。

常见的 NoSQL 数据库包括 MongoDB、Cassandra、Redis 和 Neo4j 等。NoSQL 数据库通常具有高可扩展性和高性能，适用于大数据分析、实时应用和分布式系统等场景。

笔者对于 NoSQL 的了解甚少，因此希望同学们能够自行查阅相关资料以深入了解 NoSQL 数据库的工作原理和应用场景。但现在的 SQL 数据库也加入了对非结构化数据的支持，例如 PostgreSQL 现在也支持 JSON 数据类型，因此在选择数据库时需要根据具体的应用需求进行权衡和选择。

20.4 正则表达式

正则表达式（Regular Expression，简称 Regex 或 RegExp）是一种用于描述字符串模式的工具。它可以帮助我们在文本中查找、替换和提取特定的字符串。正则表达式在搜索、文本

处理、数据验证等方面有着广泛的应用，合理使用也能够提高工作效率。

20.4.1 语法概要

正则表达式的本质是“字符串匹配”，因此包括三种元素：普通字符、元字符和量词。

普通字符是指字母、数字和其他非特殊字符，这些字符在正则表达式中表示它们本身，例如 a、1 等。

元字符则指的是“这个字符我不确定”，例如：

- `.`：匹配除换行符以外的任意单个字符。
- `\d`：匹配任意数字，等价于 `[0-9]`。
- `\D`：匹配任意非数字，等价于 `^[^0-9]`。
- `\w`：匹配任意字母、数字和下划线，等价于 `[a-zA-Z0-9_]`。
- `\W`：匹配任意非字母、数字和下划线，等价于 `^[^a-zA-Z0-9_]`。
- `\s`：匹配任意空白字符（空格、制表符、换行符等）。
- `\S`：匹配任意非空白字符。
- `^`：匹配字符串的开头。
- `$`：匹配字符串的结尾。

另外，如果想要匹配一组已知字符中的一个，可以使用方括号，例如 `[abc]` 表示匹配字符“a”、“b”或者“c”中的一个；也可以反选，例如 `^[abc]` 表示匹配除“a”、“b”、“c”以外的任意字符。为了简便起见，还可以使用短横线来表示范围，例如 `[a-z]` 表示匹配任意小写字母。如果想匹配字符 `.` 本身而不是把它当作元字符使用，可以使用反斜杠进行转义，例如 `\.` 表示匹配该字符本身。

而量词指的是“这个字符出现多少次”，例如：

- `*`：匹配前面的字符零次或多次。
- `+`：匹配前面的字符一次或多次。
- `?`：匹配前面的字符零次或一次。
- `{n}`：匹配前面的字符恰好 `n` 次。
- `{n,}`：匹配前面的字符至少 `n` 次。
- `{n,m}`：匹配前面的字符至少 `n` 次，至多 `m` 次。

其中，问号 `?` 还有一个用法。首先，我们要知道正则表达式的匹配是“贪婪”的，也就是说它会尽可能多地匹配字符。例如，正则表达式 `a.*b` 在字符串 `axxxbxxxxb` 中会匹配整个字符串，因为 `.*` 会尽可能多地匹配字符。如果我们希望它“非贪婪”地匹配，也就是尽可能少地匹配字符，可以在量词后面加上一个问号 `?`，例如 `a.*?b` 在字符串 `axxxbxxxxb` 中只会匹配到 `axxxb`。

以上便是正则表达式的基本语法。通过组合这些元素，我们可以构建出复杂的字符串模式，从而实现强大的文本处理功能。

20.4.2 使用示例

例题

我们有一个文本文件，里面包含了很多电子邮件地址。我们想要提取出所有的电子邮件地址。如何使用正则表达式实现？

提示：电子邮件地址的格式通常是 用户名 @ 域名，其中用户名可以包含字母、数字、点、下划线和短横线，域名可以包含字母、数字和点。

答案

把一类文本转写成正则表达式的一般步骤是：先分块，然后把每一块转写成正则表达式，最后把这些正则表达式组合在一起。这个题目便是一个很好的例子。

我们把电子邮件地址分成三部分：用户名、@ 符号和域名。然后，我们分别为每一部分编写正则表达式。

首先，@ 符号是最简单的，因为它本身就是一个普通字符。其正则表达式就是 @。

然后，我们来看用户名部分。用户名可以包含字母、数字、点、下划线和短横线，因此我们可以使用方括号来表示这些字符的集合：[a-zA-Z0-9._]。由于用户名至少需要一个字符，因此我们可以使用量词 + 来表示这一点：[a-zA-Z0-9._]+。

域名部分类似，其正则表达式为 [a-zA-Z0-9.]+。注意，域名中不允许出现下划线，因此我们没有把下划线包含在方括号中。

最后，我们将这三部分组合在一起，得到完整的正则表达式：[a-zA-Z0-9._]+@[a-zA-Z0-9.]+。这个正则表达式可以匹配所有符合电子邮件地址格式的字符串。

这便是正则表达式的基本用法。正则表达式的语法和功能虽然只有这么多，但是它的应用却非常广泛。我们可以使用正则表达式来处理各种文本数据，例如日志文件、配置文件、代码文件等；还可以帮助我们进行数据清洗和转换，例如提取特定字段、替换字符串等，或者仅仅是在 $\text{\LaTeX}$ 的 `texttt` 前后加上空格。正则表达式还可以用于数据验证，例如验证电子邮件地址、电话号码、身份证号码等。网上有一些有趣的正则化习题，例如一些类似 word puzzle 的游戏，可以帮助我们更好地理解 and 掌握正则表达式。

第六部分

原理速览：知其然，更知其所以然

第二十一章 信息表示和机器级代码

在使用计算机的过程中，我们很容易冒出这样的问题：我们在计算机上看到各种各样的信息，比如文字、图片、音频、视频等等，这些信息是如何在计算机中表示和存储的？计算机是如何理解和处理这些信息的？我们写的代码充其量只是一堆有逻辑的文本，计算机是如何将这些文本转换为可以执行的机器级代码的？本章就来带领大家了解这些问题的答案。

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。

21.1 信息怎么被表示？

我们写的程序中，有各种各样的数据类型，例如整数、浮点数、字符、字符串等。计算机也只认识 0 和 1，那么，这些繁多的数据类型在计算机中是怎么被表示的呢？

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，且要遵从一定的格式；计算机本身并不知道内存中这里存的是什麼，这一切实际上都是程序告诉计算机的。于是你就理解了为什么在 C 系语言中的 union 可以把同一块内存当成不同类型来读写：内存中的 0-1 串并没有类型，类型只是程序员的约定。

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¹，0x 表示后面这个是 16 进制。有时候也可以看见这个数在八进制下的表示 013，这个开头 0 就表示后面这个是 8 进制数。

21.1.1 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 。

¹仅限于 GNU 语法

32 位计算机中，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）。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，那么计算机往往会将结果的低 32 位作为最终结果返回。对于有符号整数（`int`），溢出结果按模 2^n 回绕；通常情况下也可以理解为将结果的低 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。）

21.1.2 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，存储的是隐含前导 1 的一个二进制小数，实际值是 1.xxxx。

具体说来，一个 32 位浮点数的数值可以表示为：

$$(-1)^{\text{符号位}} \times (1 + \text{尾数}) \times 2^{\text{指数}-127}$$

其中，指数的偏移量是 127，这个 127 的来源是 $2^{(8-1)} - 1$ 。同理，一个 64 位的浮点数（双精度）由 1 位符号位、11 位指数位和 52 位尾数位组成，指数的偏移量是 1023。满足上述标准的单精度和双精度浮点数一般被称为规格化浮点数。因此，我们可以知道，最大的规格化单精度浮点数大概是 3.4×10^{38} ，最小的规格化单精度浮点数大概是 1.2×10^{-38} ；而最大的规格化双精度浮点数大概是 1.8×10^{308} ，最小的规格化双精度浮点数大概是 2.2×10^{-308} 。更精确的长数值（例如 128 位浮点数）也有类似的表示方法，这里就不赘述了。

现在的问题就变成，尾数位如果全是 0，指数位如果全是 0 或者全是 1，这种情况怎么办？IEEE 754 标准对这些特殊情况做了规定：

- 如果指数位全是 0，尾数位全是 0，那么表示的数值是 0。
- 如果指数位全是 0，尾数位不全是 0，那么表示的数值是非规格化数，用于表示非常接近 0 的数值。
- 如果指数位全是 1，尾数位全是 0，那么表示的数值是无穷大（Infinity）。
- 如果指数位全是 1，尾数位不全是 0，那么表示的数值是 NaN（Not a Number），用于表示未定义或不可表示的数值，例如 0 除以 0。

对于 32 位非规格化浮点数, 数值可以表示为:

$$(-1)^{\text{符号位}} \times (0 + \text{尾数}) \times 2^{-126}$$

32 位非规格化数的指数部分被固定为-126。同理, 64 位非规格化浮点数的指数部分被固定为-1022。因此最大的非规格化单精度浮点数大概是 1.7×10^{-38} , 最小的非规格化单精度浮点数大概是 1.4×10^{-45} ; 而最大的非规格化双精度浮点数大概是 1.0×10^{-307} , 最小的非规格化双精度浮点数大概是 5.0×10^{-324} 。

然而, 有些数值无法精确表示为二进制浮点数, 例如十进制下的 0.1 在二进制下是一个无限循环小数, 无法用有限的位数表示。因此, 浮点数在计算机中往往只能**近似**表示某些实数, 且在数非常大的时候精度会进一步降低。例如计算机实际上计算 $0.1 + 0.2 = 0.3000 \cdots 004$ 。另一方面, 浮点数**并不连续**, 从上述表示方法可以看出浮点数也有间隔。一般的, 把 1 到下一个浮点数之间的间隔叫做**机器精度**, 例如单精度浮点数的机器精度大概是 1.2×10^{-7} , 双精度浮点数的机器精度大概是 2.2×10^{-16} 。

另一方面, 我们知道浮点数是科学记数法, 其精度有限, 在进行加法或乘法运算时不可避免地会导致舍入误差, 而先丢失的精度可能会影响最终的结果, 因此**浮点数的运算并不满足交换律和结合律**。例如 $1.0 + 1.0 \times 10^{-7} - 1.0 \times 10^{-7} \neq 1.0 + (1.0 \times 10^{-7} - 1.0 \times 10^{-7})$ 。而在连续运算当中, 这个舍入误差会累积, 导致结果偏差越来越大。我们在实际操作中应当避免这种情况, 例如可以将数值从小到大排序后再进行运算, 并尽可能地减少数量级相差过大的数值运算, 来减少累积误差。

作为 Mini ICS, 我们只需要知道浮点数有误差就可以了 (这是因为十进制小数可能无法精确表示为二进制浮点数)。因此, 工程上**不可以利用浮点数来进行货币运算**, 除非使用高精度浮点数 (例如 decimal 类)。一般的处理是把货币放大 100 倍 (也就是把元变成分), 然后用整数来表示和计算。

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。

21.1.3 地址

在内存中, 每一个字节都有一个唯一的地址。地址通常是一个无符号整数, 表示该字节在内存中的位置。地址的大小取决于计算机的架构, 例如 32 位计算机的地址是 32 位的, 而 64 位计算机的地址理论上是 64 位的 (实际上大部分 64 位计算机只使用了 48 位或 57 位地址空间)。因此前者最多能表示 4GB 的内存, 而后者理论上最多能表示 16EB 的内存。

有了地址, 我们就可以对内存进行读写操作, 也可以使用一些更高级的数据结构, 例如数组、链表、树等。

21.1.4 字符

我们知道, 现在的计算机是使用二进制来存储数据的。但是, 对于人类使用的语言而言, 即使是相对简单的英语, 也有 52 个大小写字母、10 个数字、各种标点符号和特殊符号等, 远

远超过了二进制的 0 和 1 两种状态；而计算机却能够正确地显示并处理这些字符。这究竟是怎么做到的呢？

从编码到字符

一个非常容易想到的办法就是，给每一个字符分配唯一的一个编号，然后再使用二进制来表示该编号。通过建立字符和编号之间的唯一映射关系，我们就可以使用二进制来表示字符了。如果把许多个字符和编号之间的映射关系放在一起，就叫做**字符集**；至于如何建立字符和编号之间的映射关系，就叫做**编码方案**。

然而，一开始，不同厂家、不同地区等生产的计算机系统使用不同的编码方案，导致同一个编码在不同的系统上表现为不同的字符，对数据交换造成了严重的障碍。我们童年时候玩的新新魔塔就是一个非常典型的例子，其标题在中国的计算机上显示为“穢穢姪娥”，其中角色“暗黑大法师”被显示为“稽堵姪猿蚺”，现在这个甚至成为了一个梗。

为了解决这个问题，也容易想到两种手段：要么利用一些方法来区分不同的编码方案，并在使用时指定编码方案；要么制定一个统一的编码方案，所有的计算机系统都使用这个编码方案。然而，前者的问题非常明显：如果这么做，那么所有的计算机都要存储所有的编码方案，这非常浪费空间；而且，如果用户不知道文件使用了哪种编码方案，那么就无法正确地显示文件内容。那么制定一个统一的编码方案反而成了一个不错的选择。这或许也是新时代的“书同文”吧。

最早的统一编码方案是美国人制定的 ASCII 编码。该编码使用 7 位二进制来表示 128 个字符，包含了英文字母、数字、标点符号和一些控制字符。例如，字母 A 的 ASCII 编码是 65，字母 a 的 ASCII 编码是 97，数字 0 的 ASCII 编码是 48。可是世界上并非只有英语一种语言。为了兼容法语、德语等有变音符号的语言，后来又出现了 ISO-8859（以 Latin-1 为代表）、扩展 ASCII 编码等，这些编码支持更多字符。

然而，当我们把目光投向亚洲时，我们发现了新的困难：以汉语为代表的亚洲语言有着数万个甚至数十万个字符，显然超过了上述编码的范围。为了促进国际交流，世界人民最终制定了一个统一的字符集：Unicode，或“统一码”。Unicode 使用 16 位二进制来表示 65536 个字符，包含了世界上所有主要语言的字符以及一些符号和表情符号，同时该编码方案也完全兼容 ASCII 编码和扩展 ASCII 编码，例如 0 的 Unicode 编码仍然是 48。后来也出现了扩展 Unicode 编码等，可以表示更多的字符。Unicode 编码的出现极大地促进了国际交流和信息共享。为了和不同的计算机相适应，Unicode 编码也有多种不同的表示方式，例如 UTF-8、UTF-16、UTF-32 等。其中，UTF-8 是最常用的表示方式，它使用 1 到 4 个字节来表示一个字符，比较节省空间。Linux 和 mac OS 系统默认使用 UTF-8 编码。

如果利用错误的编码方案来读取文件，就会出现乱码的问题。以下是常见的一些乱码及其原因，我们在看到这类乱码时，可以根据其原因来判断文件使用了哪种编码方案，从而选择正确的编码方案来读取文件。

说明

列表中“烫烫烫”等行和“锸斤拷”一同在网络上十分流行，但是和锸斤拷等不同。烫烫烫等和编码本身无关。

0xCC 等内容实际上是 MSVC 编译器在分配内存空间时填充的内容，0xCC 是未分配且未赋初值的内存空间，而 0xCD 是已动态分配但未赋初值的内存空间，0xDD 是已动态分配且已释放但未清理的内存空间。如果试图访问这些内存区域并以字符串形式打到终端上，就会出现烫烫烫等内容。

举个有趣的例子：

- 小明煮了 20 个饺子。当他试图吃到第 21 个时，喊出“烫烫烫”。
- 小明说“我要煮 20 个饺子”但是还没煮。当他试图吃饺子的时候，喊出“屯屯屯”。
- 小明煮了 20 个饺子，吃光了并洗了碗。当他试图吃洗完的碗里的饺子时，喊出“茸茸茸”。

表 21.1: 常见乱码以及其可能的原因

乱码	原因
锸斤拷	GBK 读 UTF-8
大量非法字符和西欧字符	UTF-8 读 GBK
仅大量西欧字符，原文变长	Latin1 读 UTF-8 或 GBK
出现大量长得像 yp 的东西	UTF-8 读 UTF-16
大量不认识的汉字	GBK 读 Big-5
烫烫烫	UTF-8 的 0xCC
屯屯屯	UTF-8 的 0xCD
茸茸茸	UTF-8 的 0xDD

Linux 和 mac OS 系统默认使用 UTF-8 编码，因此一般不会出现乱码的问题；而由于一些历史遗留问题，在 Windows 系统中，如果使用中文系统，则其编码方案通常是 UTF16 或 GBK，但该编码方案并不兼容 UTF8 编码，因此在处理 UTF8 编码的文件时，可能会出现乱码的问题。为了解决这个问题，Windows 系统后来也提供了 Unicode 编码的支持（但现在仍不完善）。

在 Windows 10 以及以后的系统中，我们可以通过 设置 > 时间和语言 > 语言 > 管理语言设置 > 更改系统区域设置，来将系统的默认编码方案更改为 UTF-8 编码，从而避免乱码的问题。此类编码应在系统新安装时就启用，以保证不会出现乱码问题。

然而，电脑屏幕上的同一个汉字，往往也有着不同的表现形式。这又是怎么做到的？难不成，Unicode 为同一个汉字的不同样子都分配了不同的编码？显然不是这样。实际上，Unicode 只为每一个汉字分配了一个编码，而同一个汉字的不同表现形式，是通过不同的字形来实现的。简而言之，“字符”是它的身体，而“字形”则是它的外套。

从字符到字形

Unicode 为每一个抽象字符都分配了一个唯一的编码，例如汉字“汉”的 Unicode 编码是 U+6C49。但是这仅仅规定了这是什么字符，却并没有规定这个字符长什么样。真正的字符长

相，是由**字形**来决定的。字形实际上是在计算机出现之前就已经存在的概念，指的是是字符的具体表现形式：横平竖直还是行云流水，撇是飘带还是刀刃，点是瓜子还是露珠，这些都由字形来决定。

同一个字符可以有着多种字形，这些字形可以有着不同的风格和特点。例如，宋体、黑体、楷体等都是汉字的**不同字形**；英语等字母语言也有不同的字形，例如 Times New Roman、Arial、Courier New 等。把目光放远到全世界，我们会发现同一个字符在不同的国家和地区也有着不同的字形，例如中国汉字和日本、韩国所用汉字的字形就有着明显的差异。但是它们都是同一个字符，只是字形不同而已——这便是大一统下的多样性。

Simplified Chinese	傑僭濫割劇喝塌姿羸懣庾扇搦摩榻潛瀛瘦瞎 磨窖竇條筵糙綢羸翁翦翩盲羸艘稠褐謁譜 豁羸轄返迷途造週遍遭選遼鄰釁閼雕雫靡颺 飯驎鬣魔麗麟
Traditional Chinese – Taiwan	傑僭濫割劇喝塌姿羸懣庾扇搦摩榻潛瀛瘦瞎 磨窖竇條筵糙綢羸翁翦翩盲羸艘稠褐謁譜 豁羸轄返迷途造週遍遭選遼鄰釁閼雕雫靡颺 飯驎鬣魔麗麟
Traditional Chinese – Hong Kong	傑僭濫割劇喝塌姿羸懣庾扇搦摩榻潛瀛瘦瞎 磨窖竇條筵糙綢羸翁翦翩盲羸艘稠褐謁譜 豁羸轄返迷途造週遍遭選遼鄰釁閼雕雫靡颺 飯驎鬣魔麗麟
Japanese	傑僭濫割劇喝塌姿羸懣庾扇搦摩榻潛瀛瘦瞎 磨窖竇條筵糙綢羸翁翦翩盲羸艘稠褐謁譜 豁羸轄返迷途造週遍遭選遼鄰釁閼雕雫靡颺 飯驎鬣魔麗麟
Korean	傑僭濫割劇喝塌姿羸懣庾扇搦摩榻潛瀛瘦瞎 磨窖竇條筵糙綢羸翁翦翩盲羸艘稠褐謁譜 豁羸轄返迷途造週遍遭選遼鄰釁閼雕雫靡颺 飯驎鬣魔麗麟

图 21.1: 思源字体的不同字形示例

不同的字形可以给人不同的感觉和印象，例如宋体给人正式、庄重的感觉，而楷体则给人优雅、柔和的感觉。因此，正确使用字形可以提高文本的可读性和美观性。

从字形到字体

如果把一套风格相同的字形放在一起，装进一个索引文件，再配上一个索引表，就形成了**字体文件**，使用这个字体文件的字形集合就是我们常说的**字体**。例如，宋体字体文件中包含了宋体字形的集合，黑体字体文件中包含了黑体字形的集合。

当计算机试图显示一个字符的时候，它就会去字体文件中查找该字符对应的字形，然后将该字形显示出来。这个过程很容易让人想到活字印刷术：把一套字形雕刻在木块上，然后把这些木块放在一个盒子里，当需要显示某个字符的时候，就从盒子里取出对应的木块，然后将其印在纸上。而实际上一开始的确是这么做的，只是把纸张换成了屏幕罢了！

一开始，人们把字体中的每一个字形作为图片来存储，也就是所谓的“位图字体”。然而这样就会出现一个弊病：每一个字体的大小是固定的，如果我们想要一个大字体，简单地放大其

图片是不行的：大家可以参考在手机上放大一张图片的效果，放大后的图片会变得模糊不清，甚至出现锯齿状的边缘。难道要为每一个字号做一个字体吗？显然不现实。

1978 年，王选院士主持研制的汉字激光照排系统问世，标志着我国在汉字信息处理领域实现了从无到有的历史性突破。该系统采用了点阵字模技术，不仅可以生成不同字号的汉字，还数千倍地压缩了存储空间，极大地提高了汉字排版的效率和质量。简单地说，它突破了位图字体的局限性，正式的使得字形能够被“算出来”而不是“画出来”，这使得人们彻底打开了字体编写的思路。

1982 年，Adobe 推出划时代的字体格式 PostScript，采用三次贝塞尔曲线来描述字形轮廓。这类字体被称为矢量字体，比王选院士的点阵字模技术更进一步。矢量字体可以通过数学公式来描述字形轮廓，因此可以任意缩放而不会失真，且文件体积较小，加载和渲染速度也较快。后来，苹果公司也推出了自己的矢量字体格式 TrueType，采用二次贝塞尔曲线来描述字形轮廓，算起来更快。

之后，微软联手 Adobe 又搞出了 OpenType 字体，结合了 PostScript 和 TrueType 的优点，又能往里塞其他东西（字符变体、连字、小型大写文字、旧式数字等），成为了现在最常用的字体格式。举例说，我们现在看到的 ff、fi 等连字，实际上就是 OpenType 字体中的一个特性。汉字方面，王选院士为中文字体打下的坚实地基，也显著地促进了中文字体的不断发展。直至今日，计算机对汉字的处理方式依然是小字号点阵字模、大字号矢量字形的结合；现在，TrueType、OpenType 成千上万种中文字体中数以亿计的汉字字形能够被我们随意使用，也离不开王选院士的开创性工作，王选院士也被誉为当代毕昇。

需要说明的是，PostScript 等都是具体的**字体格式**，而非字体文件本身。目前的字体基本上都是 TTF（TrueType Font）或 OTF（OpenType Font）格式的字体文件，而 PostScript 字体文件（.ps）则很少见了（在 Windows 上，这个扩展名甚至让位于脚本文件!）。而宋体、黑体、Times New Roman、Arial 等则是具体的**字体**，它们可以有不同的字体格式，例如宋体可以有 TTF 格式的宋体文件和 OTF 格式的宋体文件。实际上，常规使用者并不需要关系字体究竟是什么格式，只需要选择喜欢的字体（字形）即可。

从字体到屏幕

计算机屏幕是由无数个像素点组成的，每一个像素点可以显示一种颜色。当我们试图在屏幕上显示一个字符的时候，计算机会先去字体文件中查找该字符对应的字形，然后将该字形转换为一组像素点，并将这些像素点显示在屏幕上。这个过程叫做**字体渲染**。字体渲染的过程非常复杂，涉及到许多技术和算法，例如抗锯齿、次像素渲染、字距调整等。不同的操作系统和应用程序可能会使用不同的字体渲染引擎，导致同一个字体的同一个字符在不同的系统和应用程序中显示效果不同。

字重、斜体和复合字体

在计算机出现之前，字重和斜体等就随着着印刷术的发展而出现了。字重指的是字体的粗细程度，一般可以分为 Regular（常规）、Bold（粗体）、Light（细体）等。斜体指的是字体是

倾斜的，一般可以分为意大利体（也叫斜体，*Italic*）和倾斜体（*Oblique*）²。字重和斜体可以用来强调文本中的某些部分，例如标题、关键词等，从而提高文本的可读性和美观性。

在设计字体时，我们自然会考虑设计不同的字重和斜体版本。然而，如果为每一个字体都设计一个不同字重的版本，而不同字重的版本有的还需要斜体，这样就会导致字体文件数量爆炸，且每一个字体文件的体积也会变得很大。为了解决这个问题，我们可以使用**复合字体**。复合字体是指将多个字重和样式的字体文件组合在一起，形成一个统一的字体文件，这样不仅便于分发，也便于管理和使用。复合字体通常会包含 *Regular*、*Bold*、*Italic*、*Bold Italic* 等常用的字重和样式。

然而，大多数汉字字体并没有斜体字体：这是因为汉字是方块字，斜着不好看，没有这方面的需求，因此大多数汉字字体并没有斜体版本。对于英文字体而言，斜体是非常常见的，例如 *Times New Roman*、*Arial* 等都有斜体版本。

那有的读者会问了：为什么在 MS Office 中，我们能对汉字进行倾斜操作呢？这是因为，微软对斜体的定义比较宽泛：只要是倾斜的都叫斜体，而不一定非得是斜体版本的字体。对于没有斜体版本的汉字字体，微软会通过软件算法来实现倾斜效果，形成所谓的“伪斜体”。类似的，微软的 Word 等软件也会对没有粗体版本的汉字字体进行加粗处理，形成所谓的“伪粗体”，例如微软宋体（宋体，*SimSun*）就没有粗体版本，微软会通过软件算法来实现加粗效果，形成伪粗体。而对于 *Times New Roman* 等有粗体和斜体版本的字体，微软则会直接使用对应版本的字体。所以我们会发现，*Times New Roman* 等字体的斜体和整体比起来字形有明显的差异，而宋体汉字的斜体和整体比起来字形没有明显的差异。

衬线、无衬线和等宽字体

字体可以分为衬线字体、无衬线字体和等宽字体三种类型。

衬线字体（*Serif*）是指在字形的笔画末端有小装饰线条的字体，例如宋体、*Times New Roman* 等。衬线字体通常被认为更适合用于印刷品和长篇文本，因为衬线可以引导读者的视线，提高文本的可读性。无衬线字体（*Sans Serif*）是指没有衬线的字体，例如黑体、*Arial* 等。无衬线字体通常被认为更适合用于屏幕显示和短篇文本，因为无衬线字体更简洁、现代，且在低分辨率下也能保持清晰。等宽字体（*Monospace*）是指每一个字符占用相同宽度的字体，例如 *Courier New*、*Consolas* 等，也叫打字机字体。等宽字体通常被认为更适合用于编程和代码编辑，因为等宽字体可以使代码更整齐、易读，且便于对齐和排版。

我们知道，汉字是方块字，每一个字它天然就是等宽的，所以为汉字设计等宽字体没有什么意义，也就没有汉字的等宽字体。一般排版中，往往使用汉字的无衬线字体（例如黑体、微软雅黑等）来搭配英文字体的等宽字体（例如 *Consolas*、*Courier New* 等），以达到较好的视觉效果。有些时候用仿宋搭配西文等宽字体也不错。

LF 和 CRLF

刚刚讲完字符和编码，我们就可以来讲讲这个东西了。

²意大利体指的是字形本身是倾斜的，而倾斜体指的是将常规矩形字体压扁成普通平行四边形而成的字体。

LF 和 CRLF 是两种不同的换行符表示方式。LF 是 Line Feed 的缩写，表示换行符；CRLF 是 Carriage Return 和 Line Feed 的组合，表示回车换行符。有的人可能会疑惑：现代计算机上，回车和换行不是一回事吗？为什么还要区分这两者呢？实际上，这个问题要追溯到打字机时代。

在打字机时代，回车和换行不是一个键。回车是指将打印头移动到行首（但行数不变），而换行是指将打印头移动到下一行（但列数不变）。这里能提“行列”的主要原因是，西文打字机的本质实际上是在表格中输入，其字符也是等宽的。所以我们在打字机上，如果想要达到现在计算机上的换行效果，就需要先按回车键将打印头移动到行首，然后再按换行键将打印头移动到下一行。

那为什么现代计算机上这两个键就没有了呢？其实归根到底完全可以用“麻烦”来解释，合二为一能够简化键盘。但是，在 ASCII 码表中，却保留了这两个控制字符：CR（Carriage Return，回车，ASCII 码为 13）和 LF（Line Feed，换行，ASCII 码为 10），分别用 `\r` 和 `\n` 来表示。

而现代不同操作系统对这两个控制字符的使用方式却不一样：Unix 和类 Unix 系统（如 Linux、mac OS 等）使用 LF 作为换行符，而 Windows 系统则使用 CRLF 作为换行符。实际上这是历史遗留问题：在上世纪，当时系统三巨头是 Unix、Dos 和 Mac OS，它们分别使用 LF、CRLF 和 CR（你没看错，Mac OS 早期版本使用 CR，没有 LF）作为换行符。后来，Mac OS 内核受到 BSD Unix 的启发，从古老的 Classic Mac OS（使用 CR 作为换行符）演变成了 mac OS X（基于 Unix 的系统），因此也顺便改用了 Unix 的 LF。而 Dos 则演变成了 Windows 3.1（当时 Windows 3.1 仅是一个运行在 DOS 系统上的软件！），并沿用了 Dos 的 CRLF 作为换行符；再后来，Windows NT 内核（Windows 95/98 混血，2000 开始全面使用 NT 内核）诞生，但是为了兼容性，Windows 依然沿用了 CRLF 作为换行符！

这就导致了一个问题：如果我们在 Windows 系统上创建了一个文本文件，然后将其复制到 Linux 系统上打开，可能会出现换行符显示异常的问题（行尾出现不可见字符）。同样地，如果我们在 Linux 系统上创建了一个文本文件，然后将其复制到 Windows 系统上打开，可能会出现换行符显示异常的问题（所有内容都在一行显示）。

那这就很麻烦了，有没有什么解决办法呢？当然有！大多数现代文本编辑器（如 VS Code、Notepad++ 等）都支持自动识别和转换不同的换行符格式，我们只需要在保存文件时选择合适的换行符格式即可。此外，我们也可以使用一些命令行工具（如 `dos2unix` 和 `unix2dos`）来转换文本文件的换行符格式。但令人不爽的是，Git 会导致换行符格式混乱的问题，这需要我们在使用 Git 时特别注意，建议在 Git 配置中设置合适的换行符处理方式（例如 `core.autocrlf` 选项），即明确规定在检出和提交时如何处理换行符格式。

这也说明了，当今社会是一个统一的社会，标准的不统一依然是一个极其让人头疼的问题，至于之后究竟是全面统一成 CRLF 还是 LF，有待于后人的努力了。

21.2 程序怎么跑起来？

不知道同学们在写程序的时候，会不会疑惑“为什么我写的代码只是文本文件，但为什么这些特定的文本文件能够变成一个可以执行的程序、而我随便写的文章等却不能”。而另一个

可能疑惑的问题是“只认识二进制的计算机，为什么能看懂我写的语言”。这就涉及到程序的编译和链接过程了。

我们知道，计算机的编程语言有很多种，这些高级语言都是方便人类来编写的。因此，就需要一些工具来把这些高级语言翻译成计算机能看懂的机器码（machine code）。对于 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。

21.2.1 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。

21.2.2 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处理器架构的。这两步原理和过程相近，因此我们把它合并在一起，但实际上是两步，这一点需要注意。

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。**优化可能暴露代码中的未定义行为**，但是不会导致本符合标准的代码出现错误。因此，我们一定要尽可能地编写符合标准的代码。

21.2.3 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：

```
1 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：

```
1 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。编译器会在不同的架构上生成不同的汇编码，来保证程序的正确性和效率。

对于同一个机器，不同程序的同一句汇编码被编译出的机器码是一样的。比如说在 x86-64 架构上且使用 AT&T 语法时，`mov rbx, rax` 这句汇编码被编译成的机器码永远 `48 89 D8`，不会改变。

21.2.4 其他情况

对于解释性语言，情况略有不同。

以 `Python` 为例，它是一种解释性语言。这种语言并不需要先编译成机器码，而是通过解释器（例如 `CPython`）先把 `*.py` 文件编译成字节码（`bytecode`）（`.pyc` 文件），然后再由虚拟机（`VM`）来逐条解释执行这些字节码。字节码是一种中间表示形式，介于源代码和机器码之间。`Python` 的字节码是与平台无关的，可以在任何支持 `Python` 解释器的系统上运行。而另一些工具（例如 `PyPy`、`Jython`）会有 `JIT` 编译功能，能够把字节码编译成更高效的机器码来执行，而不是逐条解释执行。

而对于以 `C#` 为首的“中间语言”，情况又略有区别。`C#` 是一种编译型语言，但它并不直接编译成机器码，而是编译成一种中间语言（`Intermediate Language, IL`），也叫做托管代码（`Managed Code`）。这种中间语言是一种与平台无关的字节码，可以在任何支持 `.NET` 框架的系统上运行。然后，`.NET` 框架会利用即时编译器（`JIT compiler`）将中间语言编译成特定平台的机器码来执行。因此，`C#` 的逆向工程非常容易，直接反编译 `IL` 就能得到接近源代码的结果。

21.2.5 从文件到程序

刚才，我们知道了程序是怎么从源码转变成可执行文件的。那么，程序是怎么从可执行文件跑起来的呢？

在 Linux 中，可执行文件叫做 ELF（Executable and Linkable Format）文件。ELF 文件包含了程序的代码段（text segment）、数据段（data segment）、堆（heap）、栈（stack）等信息。操作系统通过加载器（loader）将 ELF 文件加载到内存中，并创建一个新的进程来执行该程序。在 Windows 中，可执行文件叫做 PE（Portable Executable）文件，原理类似。

当我们运行一个可执行文件时，操作系统会将文件加载到内存中，并创建一个新的**进程**来执行该程序。每一个进程中都会有一个或多个**线程**，线程是进程中的一个执行单元。每个线程都有自己的寄存器状态和栈空间，但多个线程可以共享进程的内存空间。操作系统会为该进程分配内存空间，并将程序的代码和数据加载到内存中。然后，操作系统会将 CPU 的控制权转移到程序的入口点（通常是 main 函数），开始执行程序。

程序在执行过程中，CPU 会不断地从内存中取指令，并执行这些指令。程序可能会调用其他函数、分配和释放内存、进行输入输出等操作。当程序执行完毕后，操作系统会回收该进程的资源，并将控制权返回给操作系统。一般情况下，程序只能访问分配给自己的内存空间，不能访问其他进程的内存空间。程序一般无法直接操作外存中的内容，必须通过操作系统提供的系统调用来进行文件读写等操作，显著地降低了恶意程序破坏文件的风险。

当然，也有一些病毒等恶意程序会利用系统漏洞来直接操作其他进程的内存空间，或者直接操作外存中的内容，破坏文件系统的完整性和安全性。

怎样才能证明上述 ELF 文件“确实”包括这些内容呢？我们可以这样，先用 C³写一点东西，然后运行下列命令：

```
1 gcc -o hello hello.c # 编译 C 源代码
2 objdump -d hello      # 反汇编可执行文件
```

然后我们会看到类似下面的输出：

```
1 hello:      file format elf64-x86-64
2 Disassembly of section .text:
3 0000000000401136 <_start>:
4 401136:^^I3l ed          ^^Ixor    %ebp,%ebp
5 ...
```

在上述反汇编输出中，我们可以看到 ELF 文件的格式信息，以及程序的代码段（.text section）中的汇编码指令。这些指令就是程序在运行时会被 CPU 执行的机器码指令。通过这种方式，我们可以验证 ELF 文件确实包含了程序的代码段。而数据段、堆、栈等内容则可以通过调试器（如 gdb）来查看。

一般情况下，在 C 中，程序的入口点是 main 函数。然而，在 ELF 文件中，程序的实际入口点是 _start 标签。这个标签是由编译器自动生成的，它负责初始化程序的运行环境，并调用 main 函数。因此，当我们运行一个 C++ 程序时，操作系统实际上是从 _start 标签开始执行程序的。

³这里不用 C++，是因为 C++ 会引入一些额外的内容，例如异常处理、虚函数表等，可能会干扰我们的观察。

21.2.6 链接

有时候，代码不是由单一源文件组成的，而是由多个源文件组成的。在这种情况下，编译器会将每一个源文件编译成一个目标文件（.o 或 .obj 文件），然后再利用链接器（linker）将这些目标文件链接成一个可执行文件。

例如，有一个目录：

```
1 project/  
2 - utils.c  
3 - utils.h  
4 - main.c
```

这个目录中的文件（不依赖任何构建工具的话）是这样编译和链接的：

```
1 gcc -c utils.c -o utils.o # 编译 utils.c 成为目标文件  
2 gcc -c main.c -o main.o # 编译 main.c 成为目标文件  
3 gcc utils.o main.o -o app # 链接目标文件成为可执行文件
```

在链接过程中，链接器会将各个目标文件中的符号表进行合并，并解决符号引用。例如，如果 `main.c` 中调用了 `utils.c` 中的函数，链接器会将这些函数的地址进行替换，从而使得程序能够正确地调用这些函数。

我们知道，在 C 语言中，变量有局部和全局之分。这些变量还有一些重要的属性，如 `static`、`extern` 等。这些属性会影响变量的链接方式。例如，`static` 变量是局部变量，只能在定义它的源文件中访问，因此它不会出现在符号表中。而 `extern` 变量是全局变量，可以在多个源文件中访问，因此它会出现现在符号表中，链接器会将这些变量的地址进行替换。通过符号表，链接器能够正确地将各个目标文件中的符号进行链接，从而生成一个完整的可执行文件。

第二十二章 内存缓存管理和系统调用

在上一章，我们已经知道，计算机不是直接执行文本文件，而是执行文本文件编译出的机器码。而新的问题接踵而至：计算机是如何存储和管理这些机器码的？计算机是如何从内存中取出指令和数据的？买电脑的时候发现内存的频率仅有几千 MHz，而 CPU 的频率却高达几 GHz，计算机是如何解决这个速度差异的？同时运行的计算机程序众多，如果都在内存中乱写，那么内存岂不是乱成一锅粥了？本章就来带领大家了解这些问题的答案。

22.1 内存怎么被管理？

刚刚我们提到，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？

22.1.1 虚拟内存、页面、页表

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 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。

实际上，如果我们在编译一个多源文件的 C 程序的时候，允许中间文件的生成，就会发现每个源文件都会生成一个中间的汇编文件，这些汇编文件中的地址全都是**虚拟地址**，这个虚拟地址¹是基于特定汇编文件的；也就是说，在链接这一步之前，每个汇编文件的地址都是从 0 开始的。链接器会把这些汇编文件链接成一个可执行文件，链接过程中会把这些虚拟地址重新映射成一个连续的地址空间，但该地址空间实际上依然是虚拟地址空间。最后，当我们运行这个可执行文件的时候，操作系统会把这个虚拟地址空间映射到物理内存中。

对普通应用程序开发者（相对于底层开发者或嵌入式开发者）而言，现代计算机编程中，我们把所有地址全都当成虚拟地址来使用。也就是说，C 的裸指针指向的地址都是虚拟地址，

¹严格地说，应该叫做“节内偏移”，不是虚拟地址，但实际上按虚拟地址理解不会出现太大偏差。

操作系统会负责把它们映射到物理内存中。我们在编写程序的时候，不需要关心物理内存的地址，只需要使用虚拟地址就行了。

而这时，我们会发现虚拟内存还能达成一个意外之喜：虚拟内存的存在，实际上阻止了程序访问不属于它的内存区域。因为每个进程都有自己的虚拟地址空间，操作系统只需确保每个进程只能访问自己的虚拟地址空间，不能访问其他进程的虚拟地址空间。这样就能有效地保护每个进程的内存安全，防止恶意程序破坏系统的稳定性和安全性。这是现代操作系统比 DOS 等老式操作系统更安全的重要原因之一。

操作系统使用页表来管理虚拟地址和物理地址之间的映射关系。虚拟内存被划分成固定大小的页面（page），物理内存也被划分成同样大小的页面。每个虚拟页面可以映射到一个物理页面，或者不映射（表示该虚拟页面没有对应的物理页面）。操作系统通过页表来记录这些映射关系。当程序访问一个虚拟地址时，操作系统会查找该虚拟地址所在的页表项，找到对应的物理地址，然后进行访问。

页面是虚拟内存的基本单位，通常是 4KB 或 8KB。页表是一个数据结构，用于记录虚拟地址和物理地址之间的映射关系。每个页表项包含了虚拟页面的状态信息，例如是否有效、是否被修改、是否被访问过等，以及对应的物理页面的地址。当程序访问一个虚拟地址时，操作系统会查找该虚拟地址所在的页表项，找到对应的物理地址，然后进行访问。

22.1.2 磁盘交换区和缺页异常

磁盘交换区用来解决物理内存不足的问题。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）；而在需要访问这些页面时，操作系统会从磁盘交换区中加载相应的页面到物理内存中。这样就能让计算机在物理内存不足的情况下继续运行，而不会因为内存不足而崩溃。

在上述利用页表管理虚拟内存的过程中，如果程序访问了一个虚拟地址，但是该虚拟地址没有对应的物理地址，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。如果磁盘交换区也没有该页面，那么操作系统就会终止该程序，并报告段错误（segmentation fault），这大概率是因为程序访问了一个非法的虚拟地址。所以说页表实际上就是操作系统“确保每个进程只能访问自己的虚拟地址空间”的重要工具。

如果在加载相应页面到物理内存中时，物理内存已经满了，那么操作系统就需要选择一个页面进行换出，以腾出空间来加载新的页面。操作系统通常会使用一些算法来选择要换出的页面，例如最近最少使用（LRU）算法、先进先出（FIFO）算法等。

22.1.3 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”，多数情况下，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；少数情况下（例如堆内存），操作系统会预先分配一些物理页而不是延迟到首次访问才分配。

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。

22.1.4 一个例子

假如，我们打开了微信。这时候，操作系统给微信预留了 1GB 的虚拟地址空间；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！

22.2 怎么压榨 CPU 的性能？

我们知道，CPU 是计算机的核心部件，负责执行指令和处理数据，其做法是从内存中取指令和数据，执行指令，然后把结果再存回内存。但是，现在的计算机内存的速度已经远远跟不上 CPU 的速度了。我们怎么才能更进一步地压榨 CPU 的性能呢？

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。这又是为什么呢？这就涉及到了 CPU 的缓存机制。

22.2.1 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²，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。

²现代 CPU 通常集成在内部做多核共享缓存

22.2.2 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）是缓存中最小的传输单位，通常为 64 字节，但 CPU 依然能够按字节寻址。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。

22.2.3 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。

22.2.4 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，等到当缓存行被标记为“脏”时才写回时再写回内存；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。

22.2.5 大矩阵乘法的工作原理

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。

22.2.6 流水线

上述缓存机制虽然显著提升了 CPU 的性能，但是 CPU 依然有一个瓶颈：指令执行的速度远远跟不上 CPU 的时钟频率³。为了进一步提升 CPU 的性能，现代 CPU 采用了流水线（pipeline）技术。

这个流水线和工厂内的流水线非常类似。例如汽车组装工厂，现在并不是一辆车组装完了再组装下一辆车，而是把组装过程分成多个阶段，每个阶段由不同的工人负责。这样，当第一辆车进入第二个阶段时，第一辆车的第一个阶段已经完成，第二辆车可以进入第一个阶段进行组装。这样，工厂就能够同时组装多辆车，大大提高了生产效率。在 CPU 中，我们也是这样，把一个指令的执行过程分成：

1. **取指 (IF)**：根据 PC 把指令读进指令寄存器；
2. **译码 (ID)**：解析操作码、读寄存器堆拿到操作数；
3. **执行 (EX)**：在 ALU 或地址生成单元里完成运算；
4. **访存 (MEM)**：若是 load/store，访问数据缓存；
5. **写回 (WB)**：把结果写回寄存器堆并更新标志位。

然后和工厂中流水线一样，每一级都让独立的硬件单元完成。理想情况下，当上一条指令进入 EX 阶段时，下一条指令就跟着进入 IF 阶段，于是每个时钟周期都能完成一条指令的执行，大大提高了 CPU 的性能。

在实际情况下可能有三类“气泡”会让流水线停顿：

- **数据冒险**：当后一条指令恰好用到了前一条指令尚未写回的结果时，就会发生数据冒险，一个容易想到的解决方法是插入**气泡 (stall)**，让流水线停顿一段时间，直到前一条指令的结果写回。另外的解决方法是**数据转发 (或数据前递, data forwarding)**，直接把前一条指令的结果旁路到后一条指令的 EX 级，避免停顿。
- **控制冒险**：当遇到分支判断的时候，下一条指令的地址实际上是未知的。这时候也容易想到插入气泡来等待分支结果。为了防止这种情况，现代 CPU 通常会采用**分支预测 (branch prediction)** 技术，猜测下一条指令的地址，并提前加载到流水线中。如果猜测正确，就继续执行；如果猜测错误，就丢弃错误的指令，重新加载正确的指令，这样的代价是大约 10 到 20 个时钟周期的猜测惩罚。而怎么猜则是一个技术活，常见的方法有静态预测（例如总是猜测分支不跳转）和动态预测（例如利用历史信息来预测分支行为）。

³CPU 的时钟频率指的是 CPU 每秒钟能够执行的时钟周期数，通常以 GHz 为单位表示。现代 CPU 的时钟频率通常在 2GHz 到 5GHz 之间，也就是每秒钟能够执行 20 亿到 50 亿个时钟周期。时钟周期是 CPU 时间的最小单位

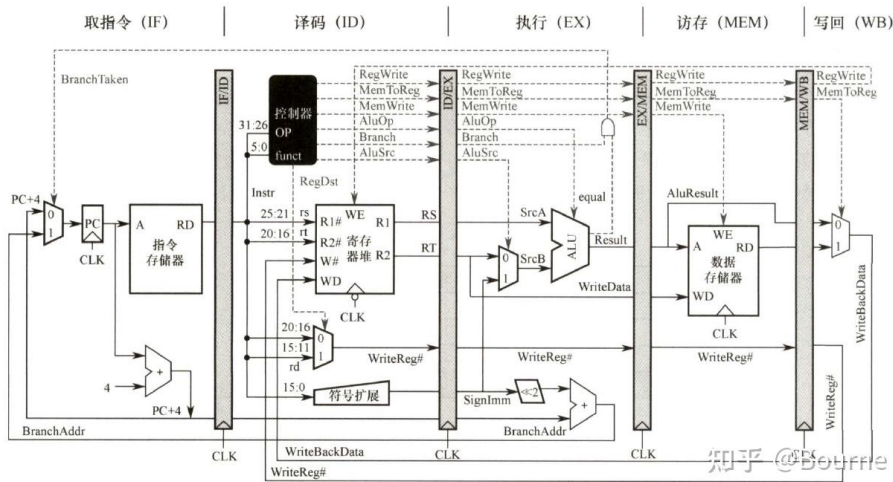


图 22.1: 一张“臭名昭著”的流水线示意图

- 结构冒险：当多个指令同时竞争同一个硬件资源时，就会发生结构冒险。例如，如果只有一个乘法器，而两条指令都需要使用乘法器，那么就会发生结构冒险。当然插入气泡也并非不可，而通过多端口寄存器堆、分离的指令和数据缓存等方法也可以缓解结构冒险的问题。

要是再往上提升性能，一般有三种手段：超标量（superscalar）、乱序执行（OoO）和超线程（SMT）。超标量指的是每一个周期同时发射多条指令到流水线中执行；乱序执行指的是指令不必严格按照程序顺序执行，而是可以根据数据依赖关系和资源可用性来动态调整执行顺序，只要操作数就绪了这条指令就可以抢跑，最后按指令序号重新提交结果；超线程指的是在流水线里面交替塞两条线程的指令，把闲置端口也利用起来。

以上操作对我们写代码有相当的启示：尽量保持分支可预测（有规律，少跳转），减少数据依赖（多用临时变量，少用全局变量），减少资源竞争（少用全局变量，少用锁），循环体小而整齐（减少指令数，增加指令并行度）。这样就能让流水线吃得饱饱的，性能自然就上去了。例如：

```
1  int cnt = 0;
2  for(int i = 0; i < n; i++)
3      if (a[i] > 128) // 这个是分支结构，且不可预测
4          ++cnt
```

这个实践是不好的，因为数组模式随机，分支不可预测，数据依赖严重。改成下面这样就好了：

```
1  int cnt = 0;
2  for(int i = 0; i < n; i++) {
3      int flag = (a[i] > 128); // 这个是运算，没有分支表达式
4      cnt += flag;
5  }
```

这样就消除了分支，数据依赖也减轻了许多，编译器大概会把上述内容编译成 `setgt` 和 `add` 指令，流水线就能更好地并行执行。

提示

当然，根据“不优化”原则我们知道，实际操作中未必需要严格这么写，或者说仅在以下情况差异显著：

- 数据量巨大，例如 `n` 达到百万级别以上；
- 数据内容相当随机地分布，例如 `a[i]` 的值均匀分布在 0 到 255 之间；
- 编译器没有做激进优化，例如开的 `-O0` 或者 `-O1`；
- 没有使用 SIMD 指令等技术，例如编译器没有自动向量化，或者没有手动使用 `AVX` 等指令集。

反之，当数据量小、大多数数据大于 128（分支预测器能学习并预测）、编译器激进优化（“吸氧”甚至“吸臭氧”）、使用 SIMD 指令等技术时，差异就不明显了。

如希望验证我的上述说法，可以利用 `perf` 等工具进行性能分析，重点观察 `branch-misses`、`instructions`、`cycles` 等指标，`-O2` 和 `-O0` 的差异也可以对比一下。

22.2.7 现代 CPU 的架构

旧时代的 CPU 一般走的是单核高频路线，这也是非常容易想到的提升性能的方式：把一个核的频率提升到极限，然后让这个核尽可能地快地执行指令。这样做的好处是简单易行，缺点是功耗和发热量都非常高，且单核性能提升空间相当有限。这个路线撞墙的例子就是 Intel 的 NetBurst 架构（奔腾 4），频率最高能达到 3.8GHz，但是单核性能并没有显著提升，反而因为发热量过大而被迫降频。

于是，现代 CPU 性能渐渐地走向了多核化、并行化的路，性能不仅靠 GHz 撑着，并行度和专用加速也成为了重要的指标。当下主流芯片把多种计算单元拼成 SoC，一般还有大小核之分（big.LITTLE 架构），大核负责高性能计算，小核负责低功耗计算，二者协同工作以提升整体性能和能效比。

- 1. 性能核（P-core）：乱序、宽发射、高频率，跑串行关键路径；
- 2. 能效核（E-core）：顺序或窄发射，面积小、功耗低，跑后台线程；
- 3. 矢量/矩阵单元——SSE/AVX/AVX-512、SVE、AMX，一条指令打 512 bit-2048 bit 的 SIMD，做 dense math；
- 4. 集成 GPU Or NPU：上千线程级的 SIMT，负责图形与 AI 推理；
- 5. 片上系统：DDR/LPDDR 控制器、PCIe 5.0、CXL、缓存一致性总线（Ring/Mesh），把 CPU、GPU、加速器、内存、外设粘在一起。

而缓存也从上文所述的经典缓存升级为支持网状多切片、非包容/非排他性、智能预取等特性的现代缓存系统，以适应多核、多线程、高并发的计算需求：

- 每个 P-core 独享 48 KB L1-I + 32 KB L1-D + 1-2 MB L2；
- 多核通过 Mesh 节点共享 24-96 MB L3，切片数等于核数，降低热点；
- 目录式（Directory）或总线嗅探（MESIF）协议保证多核一致性，跨核延迟 30-60 ns。

而对于我们开头的“超大矩阵乘法”这种还吃内存带宽的计算任务，现代 CPU 也有不少提升手段：

- AVX-512 / AMX：单指令可算 16×64 矩阵块，理论算力提升 4 到 8 倍；
- 高带宽内存：笔记本 LPDDR5X 已做到 120 GB/s，服务器 HBM3 突破 1 TB/s；
- 缓存阻塞（cache blocking）：把超大矩阵切成 L2 能装下的子块（如 256×256 ），再在内层用 SIMD 展开，就能把 100 ns 的内存访问变成 5 ns 的 L2 命中，轻松获得 10 倍数级提速。

所以说，现代的 CPU 并不是单核跑分的时代，而是多核、矢量、缓存墙协同作战。写程序的时候，只需要让计算靠近数据、并行匹配硬件宽度，就能真正的把晶体管一滴不剩地榨成有效算力。

22.3 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。

段错误的发生原因，我们刚刚已经提过了：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。这是为了避免恶意程序对其余程序或操作系统造成破坏。但实际上，部分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。

系统调用是一种特殊的函数调用，允许用户程序请求操作系统提供特权服务，例如文件操作、网络通信、进程管理等。

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。

22.3.1 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。

22.3.2 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，比普通函数调用慢不少——这还是现代 CPU 的优化结果。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，例如使用缓冲 IO 或批量读写等。

第二十三章 计算机的启动和输入输出

那么现在计算机内部是怎么工作的就很清晰了。但是，这仅仅是计算机“内部”的问题。计算机作为一个整体，除了内部的 CPU、内存、硬盘等部件以外，还有很多外部设备，例如显示器、键盘、鼠标、网络接口等等。但我们想让这些东西一齐启动，只需要轻描淡写的按下电源键就行了，这里面到底发生了什么呢？计算机又是怎么从外界接收和发送信号的呢？本章就来带领大家了解这些问题的答案。

23.1 计算机如何启动？

我们从商店买来一台计算机，插上电源，按下电源键，计算机就启动了。这个过程发生了什么呢？硬件在这段时间内如何协同工作，使得计算机能够启动？为什么新买的计算机启动很快，而旧计算机启动很慢？为什么在更换一个性能更好的显卡之后，计算机反而启动变慢了？为了回答这些问题，我们需要了解计算机的启动过程。

计算机的启动过程可以分为以下几个步骤。

23.1.1 加电自检

实际上一个很反直觉的事情是，计算机在关机的时候也是一直需要通电的。计算机的主板上有一个小电池，叫做 CMOS 电池，它负责为计算机提供基本的电源；主板上还有一个芯片，叫做 BIOS/UEFI 芯片，它记录了当前时间、日期、硬件配置等信息。

当我们按下电源键时，主板收到“按下电源键”这个信号后，给电源发信号。于是电源率先启动，但它并不是立即给所有硬件供电，而是按严格的时间顺序先输出 3.3V、5V、12V 等不同的电压，自己也会不断检测这些电压是否在允许的误差值内。如果一切良好，电源会向主板发送一个“Power Good”信号，表示电源已经稳定工作。否则，主板会立即切断电源，防止其他硬件因错误的工作电压而损坏。

收到“Power Good”信号后，主板上的芯片组就会开始工作。芯片组会首先复位 CPU，然后 CPU 从固定的地址开始执行整个计算机启动中的第一个指令，运行第一个程序——BIOS/UEFI 程序，开始加电自检（POST, Power-On Self Test）。加电自检的任务是检测计算机的硬件是否正常工作，一般先按顺序检查 CPU、内存、显卡，再检查其余设备。如果发现任何问题，加电自检会发出错误信号，例如蜂鸣声、错误代码等，提示用户进行修复。如果一切正常，加电自检会将控制权交给下一个程序。

23.1.2 硬件初始化和配置

POST 通过后，BIOS/UEFI 就会开始初始化硬件。它会为每一个 PCIe¹设备分配总线编号、内存映射 I/O 空间²和中断号，确保这些设备都能被操作系统正确识别。在这个时候，主板上的 RGB 灯、风扇、CPU 超频等用户自定义配置也会被加载。

这一阶段的速度主要取决于主板固件的优化程度以及硬件的数量和类型。高端主板通常会有更好的固件优化，因此启动速度更快；而安装了大量硬件设备的计算机则需要更多时间来初始化这些设备，导致启动速度变慢。

23.1.3 引导

硬件准备就绪后，BIOS/UEFI 就会开始引导操作系统。它会按照预设的引导顺序，依次检查硬盘、SSD、U 盘、光驱等设备，寻找操作系统。传统的引导方式 Legacy 会读取硬盘首个扇区（MBR，512 字节），该扇区存放着一段大小为 446 字节的引导代码；而现代的 UEFI 引导方式则会读取 EFI 系统分区中的引导文件（通常是 `bootx64.efi`），该文件可以存放在 FAT32 格式的分区中。

如果找遍整个引导顺序都没有找到操作系统，BIOS/UEFI 就会发出错误信号 “No bootable device”，随即终止启动过程。如果找到了操作系统，BIOS/UEFI 就会将控制权交给引导加载程序（Boot Loader），开始加载操作系统。

引导加载程序又会做两件事：第一，把操作系统内核（如 `winload.efi`、`vmlinuz` 等）加载到内存中；第二，把启动参数（如启动分区、内核参数等）传递给操作系统内核。引导加载程序的速度主要取决于存储设备的读写速度以及引导加载程序的优化程度。

23.1.4 内核加载与驱动初始化

操作系统内核获得控制权后，会首先初始化 CPU 调度子系统、内存管理子系统，然后加载硬件抽象层（HAL）和必要的驱动程序。现在屏幕大概率会出现操作系统的 Logo 或者启动动画。随后，内核会并行地初始化其他设备，速度主要决定于磁盘性能、驱动数量、安全策略等因素。

23.1.5 用户空间启动与登录界面

现在内核初始化完毕了，可以启动第一个用户态³进程了。在 Windows 中，第一个用户态进程是 `smss.exe`，它负责启动其他系统服务和后台进程；在 Linux 中，第一个用户态进程通

¹PCIe，全称 Peripheral Component Interconnect Express，外设组件互连快速通道，是一种高速串行计算机扩展总线标准，用于连接主板和各种硬件设备，例如显卡、网卡、存储设备等。

²内存映射 I/O（Memory-Mapped I/O，MMIO）是一种将外设设备的寄存器映射到计算机内存地址空间中的技术。这样，CPU 可以通过读写内存地址来访问外设设备，而不需要使用专门的 I/O 指令。

³用户态（User Space）是指操作系统中运行应用程序的环境，与内核态（Kernel Space）相对应。用户态中的程序无法直接访问硬件资源和内核数据结构，而是通过系统调用与内核进行交互。这样可以提高系统的安全性和稳定性，防止用户程序对系统造成破坏。

常是 `systemd` 或者 `init`。这些进程会负责挂载系统分区、启动日志服务、启动设备管理器等，这些都被叫做“自动启动项”。

当所有的自启动项都加载完毕，登录界面出现在屏幕上，整个启动流程结束。用户输入密码后，`Explorer` 或桌面环境继续加载用户级程序，至此计算机完全可用。

23.1.6 问题解答

现在我们就可以解答前面提出的问题了。

为什么新旧电脑启动速度差异巨大？

- 1. 新主板往往用的是 `UEFI Fast Boot`（快速启动）技术，可以跳过一些不必要的硬件检测和初始化步骤，从而大幅提升启动速度；而旧的主板往往是全部检测，耗时长；
- 2. 新电脑往往使用 `SSD` 作为存储设备，`SSD` 的读写速度和和随机读取延迟都要显著优于传统的机械硬盘，因此引导加载程序和内核加载的速度更快；
- 3. 新的操作系统往往是干净的，没有太多的自启动项，因此用户空间启动的速度更快；而旧的操作系统往往安装了大量的软件和服务，导致自启动项过多，拖慢了启动速度。

大家完全不必要担心自己的计算机使用的不是 `UEFI` 技术，因为今年已经是 2025 年了，想找到 `BIOS` 主板已经非常困难了——除非你用的电脑是在 2018 之前买的。那么，如果你的计算机确实是旧的主板或依然缓慢，可以进入 `BIOS` 设置（在开机前猛按 `F2`、`F8`、`Del` 等，具体按键见主板说明书），看看有没有“`Fast Boot`”选项，启用它就能显著提升启动速度。

至于优化启动项，这个只需要别安装一堆不必要的软件就行了，而且在安装软件的时候注意把“开机启动”选项关掉（例如 `QQ`、微信等）。

为什么换了个高端显卡，启动反而变慢？

实际上只有一点，也就是高端显卡的初始化时间更长。高端显卡往往有更多的功能和复杂的硬件设计，因此需要更多的时间来初始化和配置。此外，高端显卡往往需要加载更多的驱动程序和固件，这也会增加启动时间。如果主板的固件没有针对高端显卡进行优化，启动时间则会显著增加。但随着主板固件的更新和优化，这个问题会逐渐得到解决。

表 23.1: 启动故障排查速查表

现象	可能原因
按下电源键风扇不转	电源线没插、电源损坏、主板供电线松动
风扇转但没有画面	内存没插好、显卡供电不足、或显示器问题
主板蜂鸣器响	内存条松动、显卡没插好、CPU 过热，具体含义见主板说明书
卡在主板 Logo	硬盘没连接好、引导顺序错误、操作系统损坏
Windows 蓝屏	驱动程序冲突、硬件故障、系统文件损坏
Linux 内核恐慌	硬件不兼容、驱动程序错误、文件系统损坏

了解启动流程后，我们就能有针对性地优化：关闭不必要的启动项、启用 `Fast Boot`、升级固件、更换更快的 `SSD`，甚至调整显卡固件设置，都能让“按下电源键到进入桌面”的时间显著缩短。

23.2 计算机怎么和外界交互？

我们日常把 U 盘插进接口，耳机塞进耳机孔，点击保存按钮把文件“放”进硬盘，仿佛计算机内部与外部世界之间隔着一堵透明的墙，数据像幽灵一样穿墙而过。实际上，这堵墙上有无数扇“暗门”，门后站着一群翻译官、调度员和安检员，它们共同把数据变成电压，把电压再变回数据，还要防止外设把奇怪的东西带进屋里。本节就拆开这堵墙，看看计算机究竟怎么和外界打交道。

23.2.1 I/O 地址和端口

CPU 并不直接认识“鼠标”或“打印机”，它只认“地址”。x86 体系把外设抽象成一组寄存器，每个寄存器被分配一个固定或动态的 I/O 地址：早年的键盘控制器占 0x60–0x64，串口 COM1 占 0x3F8–0x3FF。CPU 用 I/O 指令像写信一样把数据塞进这些“信箱”，外设收到信后再回信，完成一次最小交互。

现代计算机中，大部分外设搬进了“内存映射 I/O”（MMIO）：显卡的几百 MB 显存、NVMe 控制器的寄存器，都被映射到一段物理地址。CPU 把它们当成普通内存读写，但背后由芯片组把地址翻译成 PCIe 事务，送到目标设备。于是，同一条 mov 指令，既可能真往内存写数据，也可能改掉了显卡的光栅化参数——地址就是门牌，门牌背后是什么，由主板上的“片选信号”决定。

23.2.2 中断与 DMA

如果 CPU 只能靠轮询 I/O 地址来和外设交互，那效率会非常低下：CPU 每几毫秒就得问一次“鼠标你动没动？”，这样浪费电，速度还慢。为了解决这个问题，计算机引入了中断机制。

在外设完成动作（如鼠标拖动、键盘按键）后，它会在 IRQ（中断请求线）上发出一个信号，中断控制器（现代计算机一般叫 APIC，高级可编程中断控制器）收到信号后，会通知 CPU 暂停当前工作，转而执行一个预设的中断处理程序 ISR（中断服务例程），把鼠标位移、网卡数据包什么的都统统发送，处理完毕后再返回继续执行之前的任务。这样，CPU 就不需要不停地轮询外设，而是等外设主动来“敲门”，大大提高了效率。

但是中断只能通知“有事”，数据还得 CPU 亲自搬运，而现代 SSD、千兆网卡等外设的数据量巨大，CPU 搬运数据的效率远远不够。为了解决这个问题，计算机引入了 DMA（直接内存访问）技术。DMA 会把数据直接从外设搬到内存，CPU 只需要在中断处理程序中告诉 DMA 控制器“把数据搬到哪里”，然后继续干自己的事。这样，CPU 就能腾出更多时间处理其他任务，像稳坐钓鱼台的市长；而其他外设，如网卡、显卡、声卡等也能更高效地工作，总线就像城市立交桥一样繁忙而有序。

23.2.3 设备驱动

有了 I/O 地址、中断和 DMA，CPU 就能和外设打交道了。但每个外设的工作方式都不一样，CPU 怎么知道该怎么和它们交流呢？这就需要设备驱动程序，它们会把数据真正翻译成外

设能理解的格式，外设再把数据按人能理解的方式反馈回来。

一般情况下，设备驱动的上层会提供一个统一的抽象，下层则用无数运算把抽象翻译成具体的数据序列。驱动还要负责电源管理，例如笔记本合上盖子的时候，显卡驱动会把显存压进内存，网卡驱动会把 PHY（物理层芯片）关掉，节省电量，掀开盖子再原样恢复。一个驱动 bug 就可能导致系统真正的“睡死过去”，因此社区经常说驱动“占了 70% 的内核代码和 90% 的稳定性问题”。

23.2.4 热插拔、总线枚举

早年的打印机必须关机才敢插拔，这叫做冷插拔。因为接口没有电气隔离，插拔时会产生电弧，烧坏接口和设备，甚至把主板烧掉。为了解决这个问题，计算机引入了热插拔技术：接口上加了电气隔离电路，插拔时不会产生电弧；操作系统也会动态识别设备的插拔事件，加载或卸载驱动程序。而其背后，是总线枚举协议在默默工作：

1. 新设备插入，接口芯片检测到电压变化，向主机发“Present”信号；
2. 主机收到信号后，在总线上广播“你是什么设备？”，设备则回复自己的 ID、类型等信息；
3. 主机根据设备信息，加载相应的驱动程序，进而分配地址、中断向量、DMA 通道等资源。

23.2.5 从电信号到文件：以保存 U 盘为例

把“保存文件到 U 盘”拆成时序，就能看到整条交互链：

1. 用户点“保存”，应用调用 `write()` 系统调用，内核把用户缓冲区映射到页缓存；
2. 文件系统（FAT32/exFAT）在缓存里分配新簇，修改 FAT 表，生成 SCSI 命令块；
3. 内核 USB 大容量存储驱动把 SCSI 命令封装成 CBW（Command Block Wrapper），交给 xHCI 主机控制器；
4. xHCI 把 CBW 切成微帧，通过差分信号线以 480 Mbps/5 Gbps/10 Gbps 速率差分发送；
5. U 盘主控收到 CBW，把逻辑块地址翻译成 NAND 物理页，拉高就绪线；
6. 主机发起 DMA，把数据突发写入 U 盘内部缓存，U 盘回传 CSW（Command Status Wrapper）表示“写完”；
7. 内核收到 CSW，把页缓存标记为干净，应用弹出“保存成功”。

整条链路跨越用户态、内核态、USB 总线、闪存转换层，七级抽象、十几种协议，却要在几百毫秒内完成，否则用户就会抱怨“卡了”。这不禁让人感叹计算机的精密和复杂，也难怪在物流行业中计算机属于“精密仪器”或“高价值易损货物”。

23.2.6 性能与瓶颈——为什么 USB3.0 跑不满 5Gbps？

- **协议开销：**每 1024 字节有效数据要附带 20 30 字节包头、CRC、链路控制字，实际速率 ≈ 4 Gbps；
- **块大小：**FAT32 默认 32 KB 簇，写小文件时主控要做“读-改-写”，带宽骤降；
- **CPU 复制：**若主板 USB 控制器较老，数据必须经 CPU 内存复制，吃掉 10 20% 核心；

- **闪存限速**：低端 U 盘使用 TLC/QLC，持续写 1 GB 后触发缓存用尽，速度从 100 MB/s 跌到 20 MB/s。

所以，接口规格只是“天花板”，真正的地板由最慢的环节——闪存颗粒、文件系统、驱动质量——共同决定。

23.2.7 安全关卡——I/O 权限与恶意外设

并非所有外设都“人畜无害”。BadUSB 攻击把 U 盘固件刷成虚拟键盘，插入后自动输入恶意命令；Thunderbolt 设备通过 DMA 可直接读写整机内存，绕过 CPU 页表保护。现代操作系统引入多层防护：

- **IOMMU (VT-d、AMD-Vi)**：把外设 DMA 也纳入地址转换，只允许访问内核提前登记的物理区域；
- **代码签名**：Windows 要求内核模式驱动提交 EV 证书，无签名驱动默认拒绝加载；
- **USB 授权弹窗**：macOS、GNOME 检测到键盘/网卡类设备首次插入时，要求用户物理点击“允许”，阻断自动注入。

即便如此，安全与便利仍像跷跷板：完全锁死外设，研究员的示波器、开发板就无法工作；全部放行，又给攻击者留下后门。操作系统每天都在这架跷跷板上找平衡。

23.3 计算机如何联网？

我们打开计算机，插上网线或连接到无线网络（Wi-Fi），计算机就能上网了。这个过程发生了什么呢？计算机是如何通过网络与其他计算机进行通信的？为了回答这些问题，我们需要了解计算机网络的基本原理。本节中，我们会先介绍一些网络的基本概念，然后从本机开始，逐步将网络扩展到局域网、广域网，最终到达互联网。

23.3.1 带宽、传输速率、延迟和丢包率

这四个名词用来衡量网络的性能，是日常生活中最常见的几个名词。

带宽指的是网络的理论最大传输能力，通常以比特每秒（bps）或字节每秒（B/s）来衡量。带宽越大，网络的传输能力就越强。例如，一个带宽为 100Mbps 的网络理论上可以每秒传输 100 兆比特的数据（实际可能远低于这个数值）。我们家里通网的时候说的“千兆宽带”指的就是该网络的带宽是 1000Mbps=1Gbps，或 125MB/s。

而传输速率、延迟、丢包率则用于衡量网络的实际表现。**传输速率**指的是网络的实际传输速率，单位也是比特每秒（bps）或字节每秒（B/s），往往显著低于带宽。例如，一个带宽为 100Mbps 的网络可能实际传输速率只有 50Mbps 或更低。**延迟**指的是数据从发送方到接收方所需的时间，通常以毫秒（ms）为单位来衡量。延迟越低，网络的响应速度就越快。例如，一个延迟为 50ms 的网络意味着数据从发送方到接收方需要 50 毫秒的时间。**丢包率**指的是在数据传输过程中丢失的数据包的比例，通常以百分比（%）来衡量。丢包率越低，网络的可靠性就越高。例如，一个丢包率为 1% 的网络意味着在每 100 个数据包中有 1 个数据包会丢失。延迟、丢包率、传输速率等指标往往会受到网络拥塞、信号干扰、硬件性能等因素的影响。

例题

有一辆满载硬盘的卡车从北京开到上海，估计其平均传输速率、延迟和丢包率，并和现在家用网络进行对比。

答案

先估算带宽。国内高速允许最大总重量为 49 吨（半挂或板车），一辆半挂车空车质量在 16 吨以上，这里为了方便按 19 吨计算，因此装载了 30 吨的硬盘。一块企业级数据盘容量按 30TB (3×10^{13} B) 计算，自重约 700 克；算上保护盒等，按 1kg 计算，因此一辆卡车装了 3×10^4 块硬盘，总容量为 9×10^{17} B，或者 0.9EB。

卡车在高速公路上最大时速在 100 到 90 千米每小时不等。按 90 千米每小时计算，北京到上海约 1200 千米，则行驶时间大概 13.3 小时，即 4.8×10^4 秒，因此传输速率用总数据量除以时间，得到约 1.9×10^{13} B/s，即 19TB/s，约合 **152Tbps**。该数字非常惊人，是目前家用网络的近两万倍。

下面估计延迟。和常规网络按数据包发送的方式不同，卡车运输是整体运输，或者说卡车本身就是一个大“数据包”，因此延迟等于运输时间，也就是从北京到上海的时间，约 13.3 小时，即 4.8×10^4 秒。

下面估计丢包率。只要这个车没出事故、没被劫持、没整个掉沟里，那么就可以用公路运输 HDD 货物损失率 0.02% 到 0.05% 来充当丢包率。另外，如果出事故，则丢包率为 100%，而按照中国相关统计数据，公路运输百万公里事故数约为 1 起，因此完全可以忽略不计。综上，**丢包率约为 0.05%**。

相对的，现在家用网络的传输速率大概在 100MB/s 到 1GB/s 之间，延迟大概在 10ms 到 100ms 之间，丢包率大概在 0.1% 到 1% 之间。可以看到，卡车运输的传输速率远远高于家用网络、丢包率显著低于家用网络，能和最优质的光纤媲美，但是延迟则高得离谱，完全无法进行实时操作。另外，卡车运输还受天气、交通等因素影响，稳定性无法和网络传输相提并论。

在以往的计算机教材中总是出现一句话：“永远不要小看一辆满载磁带的卡车，其带宽远远超过了家庭网络。”看起来现在也差不多，只不过是把磁带换成了硬盘罢了；实际上，即使是 2025 年，把 1EB 数据从北京运到上海这个任务，最经济、最快速的方案依然是物流，其速度甚至能把 5GB/s 的高端光纤专线按在地上摩擦，成本更是低得多；只要不是那么要求时效性，物流依然是传输超大量数据的首选方案。

上述例子也提示我们怎么选择数据传输方式：GB 级别的数据，可以使用任意网络途径传输；TB 级别的数据，考虑使用专业的数据传输服务；PB 级别的数据，考虑走光纤专线；EB 级别的数据，则应考虑物流途径。要是数据量更大，那比起你把数据运过去，不如让对面把计算任务运过来。当然，上述数据是对于企业级别的网络而言的，个人用户的网络情况往往更极端，TB 级别数据就可以考虑走快递等物流途径了；例如想把某些大文件从大连运送到沈阳，走网络可能需要几天时间才能传输完，而自行开车一天就能送到

23.3.2 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。

23.3.3 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。

23.3.4 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。

光网络单元（光猫，ONU）是用于将数字信号转换为光信号的设备，通常用于连接到光纤。光猫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。家用的光纤宽带通常是通过光猫连接到互联网的。

目前家用的路由器往往集成了光猫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。

下面，我们将会逐步讲述计算机从开机到联网，乃至访问互联网、数据传输的全过程。

23.3.5 网卡上线

我们开机，电脑 POST、初始化硬件、引导操作系统。操作系统走完启动流程，内核把驱动一个个唤醒。轮到网卡的时候，它还在睡觉。网卡驱动这时候会给 MAC 控制器喂一口“复位”寄存器，然后写 EEPROM 里面的 MAC 地址⁴。这时候，网卡才睡醒：它开始“打拍子”，往双绞线⁵里面打 1000BASE-TX⁶的 NLP⁷信号，并监听对面有没有回音。对面是交换机，如果一切正常会回复 FLP⁸信号。网卡收到 FLP 信号后，就知道对面有人了，接下来就可以开始协商速率、双工模式等参数，最终进入“上线”状态，链路灯两栖，网卡向操作系统抛 NETDEV_UP 事件，表示“我上线了”。

⁴MAC 地址是网卡的唯一标识符，由制造商在生产时烧录到 EEPROM 中，全球唯一。

⁵也就是“网线”，无线网卡则是收发无线信号。

⁶1000BASE-TX，全称 Gigabit Ethernet over Twisted Pair，千兆以太网双绞线传输标准，是一种以太网物理层协议，支持在双绞线上以 1Gbps 的速率传输数据。它使用 4 对双绞线进行全双工通信，采用了先进的编码和调制技术，以提高传输效率和抗干扰能力。

⁷NLP，全称 Normal Link Pulse，正常链路脉冲，是以太网物理层协议中的一种信号，用于表示网络连接的存在和状态。在 100BASE-TX 和 1000BASE-TX 等以太网标准中，NLP 信号通过在双绞线上周期性地发送脉冲来维持链路的活跃状态。

⁸FLP，全称 Fast Link Pulse，快速链路脉冲，是以太网物理层协议中的一种信号，用于表示网络连接的存在和状态。在 100BASE-TX 和 1000BASE-TX 等以太网标准中，FLP 信号通过在双绞线上周期性地发送脉冲来维持链路的活跃状态。

23.3.6 DHCP：办临时身份证（IP）

现在协议栈还是光秃秃的，只有一个 MAC 地址，别的什么都没有。于是，网卡构造一个以太网广播包，目的是 FF:FF:FF:FF:FF:FF，来源为自己，Ethertype⁹为 0x0800（IPv4），里面是 UDP 源端口 68、目的端口 67 的 DHCP Discover 报文，走 UDP¹⁰协议发走，表示“我来了，给我分个 IP 地址吧”。

交换机收到内容后，这个包就洪泛到所有端口；路由器则把广播改单播，扔给 DHCP 服务器¹¹。DHCP 服务器收到请求后，查池子，挑一个空闲的 IP，回复：192.168.1.123/24，租期 86400 秒，网关 192.168.1.1，DNS 8.8.8.8¹²。随即终端回 Request、服务器回 ACK¹³，一来一回四个包，俗称 DORA（Discover-Offer-Request-Ack）。至此，计算机就有了临时身份证：IP 地址、子网掩码、网关、DNS 服务器等信息。如果我们抓包的话，会发现这些包的 Transaction ID 都是一样的，用于标识同一个 DHCP 会话，像一起开黑的暗号，匹配不上就丢弃掉，防止隔壁老王冒领。

收到这个“临时身份证”后，操作系统就把 IP 地址、子网掩码、网关等信息写进内核数据结构。

23.3.7 ARP：户籍管理人员

现在计算机有了 IP 地址，可以和同一子网内的其他计算机通信了。但有了 IP 是不够的，数据包最终还需要知道发送到哪里，这依赖于 MAC 地址。协议栈查路由，发现网关 192.168.1.1 在直连网段，于是构造一个 ARP Request 广播：“谁有 192.168.1.1？告诉 192.168.1.123！”

网关会收到该信息，回复这个 Unicast¹⁴ ARP Reply：“192.168.1.1 对应 74:ac:5f:xx:xx:xx。”ARP 会缓存 60 秒，防止频繁广播。

ARP 还能防止主机的 IP 冲突：如果两台主机 IP 相同，会同时回复包：duplicate address detected，提示用户修改 IP。

23.3.8 路由：熟知路径的快递员

假如我们 ping 了 8.8.8.8。于是，协议栈就会查最长前缀匹配：

1. 8.8.8.8 不在 192.168.1.0/24 网段内，不能直连，走默认路由 192.168.1.1，构造以太网帧，目的 MAC 为网关，源 MAC 为自己，Ethertype 为 0x0800（IPv4）；
2. 交换机查 CAM 表，发现不认识目的 MAC，把帧扔到上联口；

⁹Ethertype 是以太网帧中的一个字段，用于标识上层协议类型。它是一个 16 位的无符号整数，通常以十六进制表示。例如，IPv4 的 Ethertype 为 0x0800，IPv6 的 Ethertype 为 0x86DD，ARP 的 Ethertype 为 0x0806。

¹⁰UDP，全称 User Datagram Protocol，用户数据报协议，是一种无连接的传输层协议，用于在计算机网络中传输数据。与 TCP（传输控制协议）不同，UDP 不提供可靠的数据传输和流量控制，因此适用于对实时性要求较高但对数据完整性要求较低的应用场景，如视频流、在线游戏等。

¹¹DHCP 服务器通常是路由器自带的，也可以是专门的 DHCP 服务器。

¹²这里的 IP 地址、网关、DNS 服务器地址仅为示例。

¹³ACK，全称 Acknowledgment，确认，是计算机网络中的一种控制信息，用于确认数据包的接收和传输。在 TCP 协议中，ACK 是一个重要的标志位，用于表示接收方已经成功接收到发送方发送的数据包，并通知发送方可以继续发送下一个数据包。

¹⁴Unicast，单播，是计算机网络中的一种通信方式，指数据包从一个发送方传输到一个特定的接收方。与广播（Broadcast）和组播（Multicast）不同，单播通信只涉及两个节点，即发送方和接收方。

3. 路由器收到帧，剥掉二层头，查三层转发表，找到下一跳 10.0.0.2¹⁵，TTL 减 1¹⁶，重新构造以太网帧，目的 MAC 为下一跳；
4. 运营商核心继续查 BGP 路由，AS_PATH¹⁷像快递单一样，经 4837、15169¹⁸等多个 AS，最终到达目的地 Google DNS 服务器。

整个过程涉及多层协议、多级路由器，最终实现了从本地计算机到全球互联网的通信。以上跳跃中，每一跳的 TTL 都会减去 1，如果 TTL 减到 0，数据包就会被丢弃，并发送 ICMP¹⁹的超时消息给源主机，防止数据包在网络中无限循环。这也是 Linux 中 traceroute²⁰命令的原理：把 TTL 从 1 开始逐渐增大，记录每一跳的路由器地址和响应时间，绘制出数据包从源主机到目的主机的路径。

23.3.9 NAT：内网到外网的门房

现在大部分家庭网络都使用私有 IP 地址（如 192.168.x.x、10.x.x.x 等），这些地址在互联网中是不可路由的。为了让家庭网络中的计算机能够访问互联网，路由器会使用 NAT（网络地址转换）技术，把私有 IP 地址转换成公共 IP 地址。当计算机发送数据包到互联网时，路由器会把源 IP 地址改成自己的公共 IP 地址，并记录下这个映射关系；当互联网的服务器回复数据包时，路由器会根据映射关系，把目的 IP 地址改回私有 IP 地址，然后转发给计算机。

NAT 表项默认 120 秒无流量则过期，防止表项爆满，于是微信等软件就需要发送“心跳包”来维持连接——这就是为什么有时候没有使用某些软件但这些软件却一直在联网。

23.3.10 DNS：域名解析翻译官

光有 IP 地址可不行，这太难记了：8.8.8.8 是 Google，那百度的呢？Bing 的呢？逐个记忆 IP 地址太麻烦了，即使是查表也不方便。为了解决这个问题，计算机引入了 DNS（域名系统）技术，把域名（www.google.com）映射到 IP 地址（8.8.8.8）。容易看到，域名实际上就是我们常说的“网址”²¹。

浏览器里输入 www.google.com 后，计算机会先查 /etc/hosts 文件，再查本地 DNS 缓存，如果没有命中，就会向配置的 DNS 服务器发送 DNS 查询请求。DNS 服务器收到请求后，会查找自己的数据库，如果找到了对应的 IP 地址，就会把结果返回给计算机；如果没有找到，就会向根 DNS 服务器、顶级域 DNS 服务器等递归查询，最终找到结果并返回给计算机。计算机收到结果后，就可以使用这个 IP 地址与 Google 服务器进行通信了。

¹⁵这个 IP 地址仅为示例。

¹⁶TTL，全称 Time To Live，生存时间，是计算机网络中的一个字段，用于限制数据包在网络中的寿命。它是一个 8 位的无符号整数，表示数据包在网络中可以经过的最大跳数（hop count）。每当数据包经过一个路由器时，TTL 值就会减 1，当 TTL 值减到 0 时，数据包就会被丢弃，以防止数据包在网络中无限循环。

¹⁷AS_PATH，全称 Autonomous System Path，是边界网关协议（BGP）中的一个重要属性，用于描述数据包在自治系统（AS）之间的路径。自治系统是指由一个或多个网络运营商管理的网络集合，具有统一的路由策略和管理权限。

¹⁸4837 是中国电信的 AS 号，15169 是 Google 的 AS 号。

¹⁹ICMP，全称 Internet Control Message Protocol，互联网控制消息协议，是一种用于在计算机网络中传输控制信息的协议。它是 TCP/IP 协议族中的一个重要组成部分，主要用于报告网络错误、诊断网络问题以及传输网络状态信息。

²⁰在 Windows 中叫 tracert。

²¹严格来说，网址（URL）包含了域名、路径、查询参数等信息，而域名只是网址的一部分。例如，https://www.example.com/path?query=1 是一个完整的网址，其中 www.example.com 是域名。

DNS 也是非常容易受攻击的暗礁区：DNS 劫持、DNS 污染等攻击手段可以篡改 DNS 查询结果，甚至有些情况下还可以劫持 NXDOMAIN²²，把不存在的域名解析到攻击者指定的 IP 地址，从而实现钓鱼网站、插广告，于是就有了 DoH（DNS over HTTPS）等加密 DNS 协议，防止中间人篡改 DNS 查询结果。

23.3.11 TCP 三次握手：可靠连接的建立

现在我们有 IP 地址，可以和服务器通信了。但走 UDP 直接发送数据包是不可靠的：数据包可能丢失、乱序、重复等。为了解决这个问题，计算机引入了 TCP（传输控制协议）技术，通过三次握手建立可靠的连接。

假如浏览器拿到 47.246.24.134（淘宝），发起 SYN²³包，序列号随机 x ；服务器则回复 SYN-ACK 包，序列号随机 y ，确认号 $x+1$ ；浏览器再回复 ACK 包，确认号 $y+1$ 。至此，连接建立完成，双方各自维护源 IP、源端口、目的 IP、目的端口四元组，以及序列号、滑动窗口、拥塞窗口等状态信息，确保数据传输的可靠性和有序性。

TCP 虽然比 UDP 慢，但它把不可靠的 IP 变成了可靠的字节流，适合传输文件、网页等需要完整性的应用。

23.3.12 HTTP：网页浏览的协议

现在我们可以和服务器通信了，但数据包还是二进制的，浏览器看不懂。为了解决这个问题，计算机引入了 HTTP（超文本传输协议）技术，把数据包格式化成本文，方便浏览器解析和渲染。

假如浏览器访问 `http://www.example.com`，首先会发起一个 HTTP GET 请求，包含请求行、请求头等信息；服务器收到请求后，会返回一个 HTTP 响应，包含状态行、响应头和响应体等信息。浏览器收到响应后，就可以解析 HTML、CSS、JavaScript 等内容，渲染出网页。HTTP 是无状态的协议，每个请求都是独立的，服务器不会记住之前的请求状态。

23.3.13 TLS：加密数据的押运者

HTTP 传输的数据是明文的，容易被窃听和篡改。为了解决这个问题，计算机引入了 HTTPS（HTTP 安全）技术。该技术通过 TLS 技术的握手协商加密算法和密钥，确保数据传输的机密性和完整性。

假如浏览器访问 `https://www.example.com`，首先会发起 TLS 握手：客户端发送 ClientHello 消息，包含支持的 TLS 版本、加密套件等信息；服务器回复 ServerHello 消息，选择 TLS 版本和加密套件，并发送数字证书；客户端验证证书的合法性，然后生成预主密钥，使用服务

²²NXDOMAIN，全称 Non-Existent Domain，是 DNS 查询中的一个响应代码，表示所查询的域名不存在。当计算机向 DNS 服务器发送查询请求时，如果 DNS 服务器无法找到对应的域名记录，就会返回 NXDOMAIN 响应，告诉计算机该域名不存在。

²³SYN，全称 Synchronize，是 TCP 协议中的一个标志位，用于表示连接请求。当客户端想要与服务器建立 TCP 连接时，会发送一个带有 SYN 标志的数据包，表示“我想和你建立连接”。服务器收到这个数据包后，会回复一个带有 SYN 和 ACK 标志的数据包，表示“我同意建立连接，并确认你的请求”。最后，客户端再发送一个带有 ACK 标志的数据包，表示“连接建立成功”。

器的公钥加密后发送给服务器；服务器使用私钥解密，双方根据预主密钥生成对称加密密钥。握手完成后，双方就可以使用对称加密算法（如 AES）进行数据传输了。

TLS 在性能上比较差，因此也是优化的重点：AES-NI 指令集²⁴、ChaCha20 算法²⁵、0-RTT 握手²⁶等技术都在不断提升 TLS 的性能。

23.3.14 HTTP/2 和 QUIC：更快的数据传输

HTTP/1.1 只有一条 TCP 连接，只能串行传输数据，效率低下。为了解决这个问题，计算机引入了 HTTP/2 技术，通过多路复用、头部压缩等技术，提高数据传输的效率。HTTP/2 使用二进制帧格式，把多个请求和响应复用在一条 TCP 连接上，减少了连接建立和关闭的开销。但是该技术仍然依赖 TCP，受到 TCP 头阻塞问题²⁷的影响，丢一个包则全车等人。

于是，Google 把 TCP+TLS 搬进 UDP，改名为 QUIC（快速 UDP 互联网连接），并在 HTTP/3 中使用 QUIC 作为传输协议。QUIC 通过内置的拥塞控制和多路复用，进一步提升了数据传输的效率和可靠性。QUIC 还支持 0-RTT 连接建立，减少了握手延迟，提高了用户体验。

23.3.15 Wi-Fi：空气中的以太网

上述内容其实基本上全都是有线网络的原理。而无线网络则是“空气中的以太网”。STA（Station）扫描 AP²⁸的 SSID（分主动和被动两种方式），找到 SSID 后发起关联请求，AP 回复关联响应，双方建立连接。然后，STA 发起 DHCP 请求，获取 IP 地址等信息，最终实现联网。

现代 Wi-Fi 有不同的频段，主要频段有 2.4GHz 和 5GHz，最新的 Wi-Fi 6E 还引入了 6GHz 频段。2.4GHz 仅有 3 不重叠的信道，邻居家路由器也能像广场舞大妈一样抢占地盘，只能让“城管” RTS/CTS 来协调，但吞吐率低；5GHz 有更多信道，干扰少，速度更快，但穿墙能力差。目前所见的 6GHz 路由器还不多，速度更快但穿墙能力更差。

提示

出现上述现象的原因是，Wi-Fi 本质上还是电磁波，频率越高，波长越短，穿透能力越差。而频率更短的情况下，单位时间内能传输的数据量就越大，因此速度更快。

²⁴AES-NI，全称 Advanced Encryption Standard New Instructions，是英特尔公司推出的一组用于加速 AES（高级加密标准）算法的指令集扩展。AES-NI 指令集包含了一系列新的 CPU 指令，可以显著提高 AES 加密和解密的性能，减少加密操作对 CPU 资源的占用。

²⁵ChaCha20 是一种流密码算法，由 Daniel J. Bernstein 设计。它基于 Salsa20 算法，具有更高的安全性和性能。ChaCha20 使用 256 位密钥和 64 位随机数，能够在软件实现中提供高效的加密和解密操作。由于其优越的性能和安全性，ChaCha20 被广泛应用于各种加密协议和应用程序中，如 TLS 1.3、SSH 等。

²⁶0-RTT，全称 Zero Round Trip Time，是 TLS 1.3 协议中的一种优化技术，旨在减少连接建立的延迟。在传统的 TLS 握手过程中，客户端和服务端需要进行多次往返通信（RTT）来协商加密参数和密钥，这会导致较高的延迟。0-RTT 技术允许客户端在第一次握手时发送加密数据，而不需要等待服务器的响应，从而实现“零往返时间”的连接建立。这种技术特别适用于需要快速连接建立的应用场景，如移动设备和实时通信等。

²⁷TCP 头阻塞问题是指在 TCP 协议中，由于数据包的顺序传输和确认机制，当一个数据包丢失或延迟时，后续的数据包即使已经到达接收端，也无法被处理，导致整个连接的传输效率下降。这种现象被称为“头阻塞”，因为它阻塞了后续数据包的处理。

²⁸AP，全称 Access Point，接入点，是无线局域网（WLAN）中的一种网络设备，用于连接无线设备（如笔记本电脑、智能手机等）到有线网络（如以太网）。AP 通常通过有线连接到路由器或交换机，然后通过无线信号与无线设备进行通信。AP 可以提供无线覆盖范围，允许多个无线设备同时连接到网络，实现无线数据传输和互联网访问。

我们发现，Wi-Fi 所用电磁波的频率随着技术发展不断提高。但是 6GHz 波段距离可见光（频率约 380THz 到 750THz）还有一定距离。因此我们设想：将来有一天可不可能用可见光来做无线通信呢？答案是肯定的，已经有相关的研究和应用，但眼见的未来暂时还没办法见到可见光在传输方面的大规模应用，目前仅有光纤通信使用可见光波段来传输数据（但仍局限于红光和近红外光波段）。

但是让人哭笑不得的是，很久以前的笔记本电脑上确实存在一个红外端口，用来传输数据的，速度还挺慢的，现在都没有了。至于为什么当时红外光传输速度那么慢……这里卖个关子，请同学们自行查阅相关资料吧。

23.3.16 CDN：网络上的缓存

现代互联网中，CDN（内容分发网络）技术被广泛应用于加速网页加载和视频播放。CDN 通过在全球范围内部署大量的边缘服务器，把热门内容缓存到离用户更近的位置，减少了数据传输的距离和延迟，提高了用户体验。

假如我们使用淘宝，图片在上海被北京用户请求，如果每一次都走杭州那延迟就很高了。CDN 则会把图片缓存到北京的边缘服务器（例如天津），用户 DNS 解析会解析到天津去，这样用户请求图片时，就能直接从天津的边缘服务器获取，显著提升加载速度，这显然是模仿了 CPU “缓存”的思路——但非常大规模的缓存，也很好用。

23.3.17 代理和 VPN：外网到内网的桥梁

内网像一座有围墙的城堡。很多数据库、OA、邮件服务器等重要资源都部署在内网中，也只认内网 IP 地址。但是需求是多样的：有时候我们在校外打比赛或休假，但还要访问北京大学的内网资源，这怎么办？——请一位门房代为传话即可，这就是代理和 VPN 的作用。

代理是一种中间人服务，用户把请求发送给代理服务器，由代理服务器代为访问目标资源，然后把结果返回给用户。或者说，我们可以把代理看成雇了一个人帮我们办事：我们把想要的东西告诉他，他去城堡里取回来，然后交给我们。这样，用户就可以通过代理服务器访问内网资源，而不需要直接连接到内网中。常见的代理协议有 HTTP 代理、SOCKS 代理等。北京大学早年确实提供过这个服务，但现在已经被 VPN 取代了。

而 VPN（虚拟专用网络）则是通过加密隧道，把用户的计算机直接连接到内网中，仿佛用户的计算机就在内网中一样。这样，用户就可以直接访问内网资源，而不需要通过代理服务器。VPN 通常使用 IPSec、OpenVPN 等协议来实现加密和认证，确保数据传输的安全性和完整性。这就像是用一条超超超长的虚拟网线，把用户的计算机直接连接到内网中，帮你虚拟“回校”。

举例：北京大学 VPN。

第二十四章 数据结构

我们已经学过了 C/C++，知道了 C++ 比 C 更强的一点在于它有着 STL 这个东西。我们在 C++ 相关章节中并未详细介绍 STL 内部究竟具体是什么，但如果知道这些内容的话，能让你对 C++ 和算法都有更深的理解。

数据结构，也就是 STL 中的“容器”部分，是算法的基础。本章就来探讨一下数据结构的相关内容。

有的数据结构理解起来比较复杂，我们推荐一个[网站](#)，该网站可以可视化地展示各种数据结构的操作过程，帮助理解。

24.1 线性数据结构

24.1.1 顺序表

顺序表是最简单的一种数据结构，可以简单的理解为一段连续的内存空间。

在 C++ 中，对应的是下面两个东西：

```
1 int arr[10];           // C 风格的顺序表
2 std::array<int, 10>;   // C++ 风格的顺序表
```

顺序表的优点在于它的内存是连续的，所以可以通过下标快速访问任意位置的元素，时间复杂度为 $O(1)$ 。但是它的缺点也很明显，那就是在中间的插入和删除操作需要移动大量元素，时间复杂度为 $O(n)$ 。

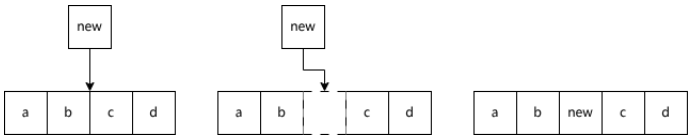


图 24.1: 顺序表的插入操作示意图

由此图可见，顺序表的插入需要先把插入位置之后的所有元素向后移动一位，然后再插入新元素。这导致插入非常缓慢，尤其是在顺序表较大时。

网站: <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

那么, `std::vector<T>`是个什么东西呢? 实际上, 这东西也是顺序表, 但它的实现方式更复杂一些。它会预留一部分内存空间, 当需要插入元素时, 如果当前容量不够, 它会分配一块更大的内存空间, 然后把原来的数据拷贝过去, 再插入新的元素。这样做的好处是, `std::vector<T>`在大多数情况下在末尾插入和删除元素的时间复杂度是 $O(1)$, 只有在需要扩容的时候才是 $O(n)$; 而在中间插入和删除元素的时间复杂度仍然是 $O(n)$ 。

24.1.2 链表

单向链表 链表是另一种简单的线性数据结构, 虽然不是一段连续的内存空间, 但它通过指针将各个节点连接起来。每个节点包含数据和指向下一个节点的指针。用 C++ 实现一个链表, 需要实现节点结构体和链表类:

```
1 struct Node {
2     int data;
3     std::shared_ptr<Node> next;
4     Node(int val) : data(val), next(nullptr) {}
5 }; // 平凡且标准的类型, 没有自定义析构函数和方法等, 因此用结构体更合适
6 class LinkedList {
7 private:
8     std::shared_ptr<Node> head;
9 public:
10    LinkedList() : head(nullptr) {}
11    void insert(int val);
12    void remove(int val);
13    void display();
14 };
```

具体的插入、删除和显示方法可以根据需要实现。这个被称作“单向链表”, 也就是说每一个元素只能指向下一个节点, 不能往回走。要想给上述链表写析构函数, 需要遍历链表并释放每个节点的内存。

在链表中删除或插入某一个节点时, 需要修改相邻节点的指针。例如, 如果要在 p、q 之间插入一个新节点 newNode, 只需要:

```
1 // 先前: p->next = q;
2 newNode->next = p->next;
3 p->next = newNode;
4 // 当然这样写起来不是很安全, 实际代码中需要检查 p 是否为空等
5 // 下同, 简单起见不再赘述
```

请试着完成删除节点的代码。

单向链表的插入可用以下图解来理解:



链表的优点在于插入和删除操作的时间复杂度为 $O(1)$ ，只需要修改指针即可；但缺点是访问任意位置的元素需要从头开始遍历，时间复杂度为 $O(n)$ 。因此，在根据索引寻找要插入或删除的元素时，虽然插入和删除本身是 $O(1)$ ，但寻找元素的过程仍然是 $O(n)$ 。

双向链表 为了解决上述链表只能走单行道的问题，我们引入了双向链表。这个用 C++ 来实现的话，节点结构体需要增加一个指向前一个节点的指针：

```
1 struct DNode {
2     int data;
3     std::shared_ptr<DNode> next;
4     std::weak_ptr<DNode> prev; // 使用 weak_ptr 避免循环引用
5     DNode(int val) : data(val), next(nullptr), prev(nullptr) {}
6 };
7
8 class DoublyLinkedList {
9     ...
10 };
```

链表内部的操作略有不同，但总体思路类似。双向链表的优点在于可以双向遍历，插入和删除操作仍然是 $O(1)$ ，但缺点是每个节点需要额外的内存来存储前驱指针。

如果要插入或删除一个节点，也需要修改指针。例如，插入一个新节点 `newNode` 在 `p` 和 `q` 之间：

```
1 // 先前: p->next = q; q->prev = p;
2 newNode->next = p->next;
3 newNode->prev = p;
4 p->next->prev = newNode;
5 p->next = newNode;
```

双向链表的插入可用以下图解来理解：



在 C++ 标准库中，`std::list<T>` 就是一个双向链表的实现。我们不要自己试图实现一个链表，直接使用标准库中的实现即可。

24.1.3 栈和队列

栈（Stack）和队列（Queue）不是数据结构。它们是基于数据结构实现的抽象数据类型，在 C++ 中被叫做“容器适配器”（Container Adapter）。栈是基于顺序表或链表实现的，遵循后进先出（LIFO）的原则；队列也是基于顺序表或链表实现的，遵循先进先出（FIFO）的原则。

栈 栈是一种只能在一端进行插入和删除操作的数据结构。我们可以使用 `std::stack<T>` 来实现栈:

```
1 #include <stack>
2 std::stack<int, std::vector<int>>> s; // 使用 vector 作为底层容器
3 // 也可以使用 list 作为底层容器: std::stack<int, std::list<int>>> s;
4 // 不指定底层容器时, 默认使用 deque 作为底层容器: std::stack<int> s;
5 s.push(1); // 入栈
6 s.push(2);
7 int top = s.top(); // 获取栈顶元素
8 s.pop(); // 出栈
```

栈只能访问其顶元素, 其余元素都无法直接访问。我们可以把栈想象成一个装东西的箱子, 只能从上面放东西和拿东西, 中间的东西是看不到的。因此, 栈遵循**后进先出**的原则。

队列 队列和栈类似, 但它允许在一端插入元素, 在另一端删除元素。我们可以使用 `std::queue<T>` 来实现队列:

```
1 #include <queue>
2 std::queue<int, std::list<int>>> q; // 使用 list 作为底层容器
3 // 也可以使用 vector 作为底层容器: std::queue<int, std::vector<int>>> q;
4 // 不指定底层容器时, 默认使用 deque 作为底层容器: std::queue<int> q;
5 q.push(1); // 入队
6 q.push(2);
7 int front = q.front(); // 获取队首元素
8 q.pop(); // 出队
```

队列也只能访问其首元素, 其余元素都无法直接访问。我们可以把队列想象成一条排队买票的队伍, 前面的人先买票离开, 后面的人只能等着。因此, 队列遵循**先进先出**的原则。

双端队列 双端队列是一种特殊的队列, 允许在两端进行插入和删除操作。我们可以使用 `std::deque<T>` 来实现双端队列:

```
1 #include <deque>
2 std::deque<int> dq; // 双端队列, 底层容器即为 deque 本身
3 dq.push_back(1); // 在尾部插入元素
4 dq.push_front(2); // 在头部插入元素
5 int front = dq.front(); // 获取头部元素
6 int back = dq.back(); // 获取尾部元素
7 dq.pop_front(); // 删除头部元素
8 dq.pop_back(); // 删除尾部元素
```

双端队列结合了栈和队列的特点, 允许在两端进行操作, 适用于需要频繁在两端插入和删除元素的场景。

有的同学可能会疑惑: 为什么有了顺序表和链表, 还需要栈和队列呢? 这是因为栈和队列是对数据结构的抽象, 提供了特定的操作接口, 方便我们在特定场景下使用。例如, 在函数调用时使用栈来保存返回地址, 在任务调度时使用队列来管理任务顺序。如果自己维护一个顺序

表或链表来实现栈或队列，代码会变得复杂且容易出错。非常常见的两个用法，一个是用栈来替代递归函数，另一个是用队列来实现广度优先搜索（BFS）算法（见后续章节）。下面是一个用栈替代递归函数的例子：

```

1 void recursiveFunction(int n) {
2     if (n <= 0) return;
3     // 处理当前节点
4     recursiveFunction(n - 1);
5     // 处理返回时的操作
6 } // 普通的递归函数
7
8 void iterativeFunction(int n) {
9     std::stack<int> s;
10    s.push(n);
11    while (!s.empty()) {
12        int current = s.top();
13        s.pop();
14        if (current <= 0) continue;
15        // 处理当前节点
16        s.push(current - 1);
17        // 处理返回时的操作
18    }
19 } // 用栈实现的迭代函数

```

那为什么要这么做呢？实际上，递归在底层上本来就是用栈来实现的。如果递归程度过深，则导致栈溢出，俗称“爆栈”。因此，在某些情况下，使用显式的栈来替代递归函数，可以在一定程度上避免栈溢出的问题，也可以更好地控制内存使用。把裸递归写成栈是一种非常常见的技巧，工程上也经常会用到。

但栈和队列也有一些不太好的地方，就是无法在中间访问，即使是简单的查询操作也不行。如果需要频繁在中间访问元素，还是需要使用顺序表或链表。

24.2 非线性数据结构

非线性数据结构包括树和图等，它们用于表示更复杂的关系。

24.2.1 树

树是一种层次化的数据结构，由节点和边组成。每个节点可以有多个子节点，但只有一个父节点（根节点除外）。树的常见应用包括文件系统、XML/HTML 文档等。

C++ 没有内置的树结构，但我们可以通过自定义类来实现。例如，下面是一个简单的树节点结构体：

```

1 struct TreeNode {
2     int data;
3     std::weak_ptr<TreeNode> father; // 指向父节点的弱指针，防止循环引用
4     std::vector<std::shared_ptr<TreeNode>> children; // 多个子节点
5     TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
6 };

```

特别的，每一个节点只有至多两个子节点的树，称为二叉树（Binary Tree）。二叉树有很多特殊的性质和操作，例如二叉搜索树（Binary Search Tree, BST）和堆（Heap）等，在后文会详细叙述。这里仅介绍一般树的基本概念和操作。

对于一棵树，我们一般有以下称谓：

根节点 树的顶端节点，没有父节点。

叶节点 没有子节点的节点。

内部节点 有子节点的节点，非叶节点。

深度 节点到根节点的最短路径长度。

高度 树中节点的最大深度。

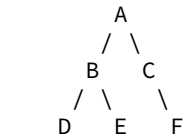
树的操作包括插入、删除、遍历等。一般情况下，对于一棵树，我们往往用其根节点来称呼整棵树。

树的插入操作通常是将新节点添加为某个节点的子节点。例如，向节点 `p` 插入一个新节点 `newNode`：

```
1 p->children.push_back(newNode);
2 newNode->father = p; // 设置父节点指针
```

树的删除包括删除叶节点和删除内部节点两种情况，删除内部节点时需要考虑如何处理其子节点。因此很少直接删除树节点，更多的是标记该节点不再使用。

而树的遍历（周游）有两种策略：深度优先遍历（Depth-First Search, DFS）和广度优先遍历（Breadth-First Search, BFS）。深度优先遍历又分为前序遍历、中序遍历和后序遍历三种方式。我们以下面这棵树为例：



前序遍历 树的前序遍历指的是：先访问根节点，再访问左子树，最后访问右子树。我们大致为大家写一下过程：

- 访问 A，输出 A，左子树是 B，右子树是 C
- 访问 B，输出 B，左子树是 D，右子树是 E
- 访问 D，输出 D，没有子树，返回 B
- 访问 E，输出 E，没有子树，返回 B，B 已经没有子树，返回 A
- 访问 C，输出 C，没有左子树，右子树是 F
- 访问 F，输出 F，没有子树，返回 C，C 已经没有子树，返回 A，结束

于是就可以得到前序遍历结果：A B D E C F。其代码实现大致是：

```
1 void preOrderTraversal(std::shared_ptr<TreeNode> node) {
2     if (!node) return;
```

```

3     std::cout << node->data << " "; // 访问节点
4     for (const auto& child : node->children) {
5         preOrderTraversal(child); // 递归访问子节点
6     }
7 }

```

当然也可以用栈来写，但是入栈的过程应该是**逆向压栈**，即先压右子节点，再压左子节点，这样出栈时才能保证左子节点先被访问。

```

1 #include <stack>
2 void preOrderTraversalIterative(std::shared_ptr<TreeNode> root) {
3     if (!root) return;
4     std::stack<std::shared_ptr<TreeNode>> s;
5     s.push(root);
6     while (!s.empty()) {
7         auto node = s.top();
8         s.pop();
9         std::cout << node->data << " "; // 访问节点
10        // 逆向压栈
11        for (auto it = node->children.rbegin(); it != node->children.rend();
12             ↪ ++it) {
13            s.push(*it);
14        }
15 }

```

其他 DFS 写成栈的形式就不再赘述了，这里留作练习。

中序遍历 中序遍历只能用于二叉树这种有明显的“左子树”和“右子树”概念的树。中序遍历指的是：先访问左子树，再访问根节点，最后访问右子树。对于上面的树，我们写一下过程：

- 访问 A，左子树是 B，右子树是 C
- 访问 B，左子树是 D，右子树是 E
- 访问 D，没有子树，输出 D，返回 B，输出 B
- 访问 E，没有子树，输出 E，返回 B，B 已经没有子树，返回 A，输出 A
- 访问 C，没有左子树，输出 C，右子树是 F
- 访问 F，没有子树，输出 F，返回 C，C 已经没有子树，返回 A，结束

于是就可以得到中序遍历结果：D B E A C F。其代码实现大致是：

```

1 void inOrderTraversal(std::shared_ptr<TreeNode> node) {
2     if (!node) return;
3     if (node->children.size() > 0)
4         inOrderTraversal(node->children[0]); // 访问左子树
5     std::cout << node->data << " "; // 访问节点
6     if (node->children.size() > 1)
7         inOrderTraversal(node->children[1]); // 访问右子树
8 }

```

后序遍历 后序遍历指的是：先访问左子树，再访问右子树，最后访问根节点。对于上面的树，我们写一下过程：

- 访问 A, 左子树是 B, 右子树是 C
- 访问 B, 左子树是 D, 右子树是 E
- 访问 D, 没有子树, 输出 D, 返回 B
- 访问 E, 没有子树, 输出 E, 返回 B, B 已经没有子树, 输出 B, 返回 A
- 访问 C, 没有左子树, 右子树是 F
- 访问 F, 没有子树, 输出 F, 返回 C, C 已经没有子树, 输出 C, 返回 A, A 已经没有子树, 结束

于是就可以得到后序遍历结果: D E B F C A。其代码实现大致是:

```

1 void postOrderTraversal(std::shared_ptr<TreeNode> node) {
2     if (!node) return;
3     for (const auto& child : node->children) {
4         postOrderTraversal(child); // 递归访问子节点
5     }
6     std::cout << node->data << " "; // 访问节点
7 }
```

广度优先遍历 广度优先遍历指的是: 按层次从上到下、从左到右依次访问节点。对于上面的树, 我们写一下过程:

- 访问 A, 输出 A, 入队其子节点 B 和 C
- 访问 B, 输出 B, 入队其子节点 D 和 E
- 访问 C, 输出 C, 入队其子节点 F
- 访问 D, 输出 D, 没有子节点
- 访问 E, 输出 E, 没有子节点
- 访问 F, 输出 F, 没有子节点, 结束

可以看到树的广度优先遍历需要用到队列来辅助实现。其代码实现大致是:

```

1 #include <queue>
2 void breadthFirstTraversal(std::shared_ptr<TreeNode> root) {
3     if (!root) return;
4     std::queue<std::shared_ptr<TreeNode>> q;
5     q.push(root);
6     while (!q.empty()) {
7         auto node = q.front();
8         q.pop();
9         std::cout << node->data << " "; // 访问节点
10        for (const auto& child : node->children) {
11            q.push(child); // 入队子节点
12        }
13    }
14 }
```

树的应用非常广泛, 例如文件系统就是一种树结构, 根目录是根节点, 子目录和文件是子节点。我们可以使用树来表示和操作这些层次化的数据。

24.2.2 图

图是另一种更复杂的数据结构，由节点（顶点）和边组成。图可以是有向的或无向的，可以包含环或不包含环。图的常见应用包括社交网络、地图导航等，用于表示复杂的关系。

一般的图有多种写法。朴素的图写法可以用许多下面的 Node 表示：

```
1 struct GraphNode {
2     int data;
3     std::vector<std::shared_ptr<GraphNode>> tos; // 可以前往的节点
4     std::vector<std::weak_ptr<GraphNode>> froms; // 可以从哪里来，这里用 weak_ptr 防
    ↪ 止循环引用
5     GraphNode(int val) : data(val) {}
6 };
```

但是这样的写法比较笨拙，无法表示边权重等信息，但在表示无权的小图时使用起来还算方便。

更常见的图的表示方法有两种：邻接矩阵和邻接表。

邻接矩阵 邻接矩阵是一种二维数组，用于表示图中节点之间的连接关系。如果节点 i 和节点 j 之间有边连接，则矩阵中的元素 $A[i][j]$ 为 1（或边的权重），否则为 0。对于有向图， $A[i][j]$ 表示从节点 i 指向节点 j 的边；对于无向图， $A[i][j]$ 和 $A[j][i]$ 表示节点 i 和节点 j 之间的边。

```
1 #include <vector>
2 class Graph {
3 private:
4     std::vector<std::vector<int>> adjMatrix; // 邻接矩阵
5 public:
6     Graph(int numNodes) : adjMatrix(numNodes, std::vector<int>(numNodes, 0)) {}
7     void addEdge(int u, int v, int weight = 1) {
8         adjMatrix[u][v] = weight; // 有向图
9         // 对于无向图，还需要添加下面这一行
10        // adjMatrix[v][u] = weight;
11    }
12};
```

邻接矩阵的优点是查询边的存在性非常快，时间复杂度为 $O(1)$ ；但缺点是空间复杂度为 $O(n^2)$ ，对于稀疏图来说非常浪费。稀疏图是指边的数量远小于节点数量平方的图。

邻接表 邻接表是一种更节省空间的图表示方法。它使用一个数组或向量，每个元素对应一个节点，存储该节点的所有邻接节点。对于有向图，邻接表中的每个节点只存储其出边的邻接节点；对于无向图，则存储所有邻接节点。

```
1 #include <vector>
2 #include <list>
3 class Graph {
4 private:
5     std::vector<std::list<std::pair<int, int>>> adjList; // 邻接表，存储邻接节点和边
    ↪ 权重
```

```

6 public:
7     Graph(int numNodes) : adjList(numNodes) {}
8     void addEdge(int u, int v, int weight = 1) {
9         adjList[u].emplace_back(v, weight); // 有向图
10        // 对于无向图, 还需要添加下面这一行
11        // adjList[v].emplace_back(u, weight);
12    }
13 };

```

邻接表的优点是空间复杂度为 $O(n + m)$ ，其中 n 是节点数量， m 是边的数量，适合表示稀疏图；但缺点是查询边的存在性需要遍历邻接节点列表，时间复杂度为 $O(k)$ ，其中 k 是节点的度数。

图的遍历也有深度优先遍历（DFS）和广度优先遍历（BFS）两种策略，类似于树的遍历。不同的是，图可能包含环，因此在遍历时需要记录已经访问过的节点，防止重复访问。这里可以用类似的递归或队列来实现 DFS 和 BFS，我就不再赘述了。

24.3 特殊数据结构

除了上述常见的数据结构外，还有一些特殊的数据结构，如哈希表、堆和 Trie 树、红黑树、B 树等，它们在特定场景下有着重要的应用。

24.3.1 二叉树及其变体

刚刚讲树的时候，我们提到了二叉树。二叉树是一种特殊的树结构，每个节点最多有两个子节点，分别称为左子节点和右子节点。二叉树有很多变体，常见的有二叉搜索树（Binary Search Tree, BST）和堆（Heap）。

二叉树的节点可以写得更简略：

```

1 struct BinaryTreeNode {
2     int data;
3     std::shared_ptr<BinaryTreeNode> left; // 左子节点
4     std::shared_ptr<BinaryTreeNode> right; // 右子节点
5     std::weak_ptr<BinaryTreeNode> father; // 父节点
6     BinaryTreeNode(int val) : data(val), left(nullptr), right(nullptr),
7     ↪ father(nullptr) {}
8 };

```

该节点未使用诸如 `vector` 之类的容器，因为二叉树的每个节点最多只有两个子节点，使用两个指针即可，更加高效。

仅仅是二叉树是没什么意思的，我们要对它进行一些限制，才能发挥它的作用。

完全二叉树 完全二叉树是一种特殊的二叉树，除了最后一层外，每一层的节点必须被填满，并且最后一层的节点必须从左到右连续排列。完全二叉树的一个重要性质是它可以用数组来高效地表示。

假设一个完全二叉树的节点按层次顺序存储在数组中，那么对于节点 i ：

- 左子节点的索引为 $2i + 1$
- 右子节点的索引为 $2i + 2$
- 父节点的索引为 $(i - 1)/2$ (向下取整)

这种表示方法使得我们可以通过简单的算术运算来访问节点的子节点和父节点，而不需要额外的指针存储。

完全二叉树有什么用处呢？它的主要应用之一是堆（Heap）。

二叉搜索树 二叉搜索树也是一种特殊的二叉树，它满足以下性质：对于每个节点，其左子树中的所有节点的值都小于该节点的值，而其右子树中的所有节点的值都大于该节点的值。这个性质使得二叉搜索树可以高效地进行查找、插入和删除操作。

我们看到，实际二叉搜索树类似于二分法查找的过程。假设我们要查找一个值 x ：

- 如果当前节点的值等于 x ，查找成功。
- 如果 x 小于当前节点的值，继续在左子树中查找。
- 如果 x 大于当前节点的值，继续在右子树中查找。

这种查找过程的平均时间复杂度是 $O(\log n)$ ，和二分查找是一样的。

现在的问题就是怎样才能使得二叉树经过插入和删除操作后仍然保持二叉搜索树的性质呢？这就需要在插入和删除时进行适当的调整。

在插入一个新节点的时候，我们按照二叉搜索树的性质找到合适的位置插入即可，从根节点开始比较：

- 如果新节点的值小于当前节点的值，继续在左子树中查找插入位置。
- 如果新节点的值大于当前节点的值，继续在右子树中查找插入位置。

而删除的时候则稍微复杂一些，主要有三种情况：

- 删除的节点是叶节点，直接删除即可。
- 删除的节点有一个子节点，用子节点替代被删除的节点。
- 删除的节点有两个子节点，找到该节点的中序后继（右子树中最小的节点）或中序前驱（左子树中最大的节点），用它来替代被删除的节点，然后删除该后继或前驱节点，形成递归。

但是上述的插入和删除操作可能会导致一些问题：假设我们把输入按升序排列插入到二叉搜索树中，那么树就会退化成一个链表，导致查找、插入和删除操作的时间复杂度变为 $O(n)$ 。为了解决这个问题，我们需要使用自平衡二叉搜索树，如 AVL 树和红黑树等。

AVL 树 AVL 树是一种自平衡二叉搜索树，它通过在每个节点上维护一个平衡因子（Balance Factor）来确保树的高度保持在 $O(\log n)$ 。平衡因子定义为左子树的高度减去右子树的高度，AVL 树要求每个节点的平衡因子只能是 -1、0 或 1。当插入或删除节点后，如果某个节点的平衡因子变得不符合要求，就需要通过旋转操作来恢复平衡。C++ 中并没有内置 AVL 树的实现，但我们可以通过自定义类来实现 AVL 树的插入和删除操作。

红黑树 红黑树则更是大名鼎鼎，其复杂度也更高一些。红黑树是一种自平衡二叉搜索树，它通过给每个节点染色（红色或黑色）并遵循一系列规则来确保树的平衡。这些规则包括：

- 根节点必须是黑色。
- 所有叶节点（包括 NIL 节点）必须是黑色。NIL 节点指的是空节点，用于表示树的末端，并不存储实际数据。
- 如果一个节点是红色的，则其子节点必须是黑色的。
- 从任一节点到其所有后代叶节点的路径上，必须包含相同数量的黑色节点。

那么研究怎么插入节点就很有意义了。为了简化描述，引入“叔叔”的概念：叔叔节点是指当前节点的父节点的兄弟节点。插入节点的步骤如下：

- 先按二叉搜索树的规则插入节点，并将新节点染成红色。
- 如果父亲是黑的，那么什么都不需要做，插入完成。
- 如果父亲是红的，那么就违反了红黑树“父子不能同红”的规则，需要进行调整：
 - 如果叔叔是红的，那么将父亲和叔叔染成黑色，将祖父染成红色，然后检查祖父节点是否违反红黑树规则，递归调整。
 - 如果叔叔是黑的或没有叔叔，则通过一次或两次旋转：
 - * 新节点和父节点是同侧的（左左或右右），进行一次旋转，祖父变父亲的另一个子节点，父亲变祖父。
 - * 新节点和父节点是异侧的（左右或右左），先进行一次旋转，使新节点和父节点变成同侧（新节点变成父亲，父亲变成儿子），然后再进行一次第一种情况的旋转。
- 最后，确保根节点是黑色。如果不是，将其染成黑色。

而删除的过程更是复杂，不再赘述。C++ 标准库中的 `std::map<K, V>` 和 `std::set<T>` 就是基于红黑树实现的，我们可以直接使用它们来存储有序的键值对或集合。

24.3.2 哈希表

哈希（散列）表是一种基于数组的数据结构，通过哈希函数将键映射到数组的索引位置，从而实现快速的查找、插入和删除操作。哈希表的平均时间复杂度为 $O(1)$ ，但在最坏情况下可能退化为 $O(n)$ 。

哈希的基本原理就是将键通过哈希函数转换为一个整数索引，然后将值存储在该索引位置的数组中。

在 C++ 中，哈希表可以通过 `std::unordered_map<K, V>` 和 `std::unordered_set<T>` 来实现：

```

1 #include <unordered_map>
2 #include <unordered_set>
3
4 std::unordered_map<std::string, int> hashMap; // 哈希映射
5 hashMap["apple"] = 1; // 插入键值对
6 int value = hashMap["apple"]; // 查找键对应的值
7
8 std::unordered_set<int> hashSet; // 哈希集合
9 hashSet.insert(10); // 插入元素
10 bool exists = hashSet.find(10) != hashSet.end(); // 查找元素是否存在

```

在理想状态下，散列函数应该能给每个不同的键生成不同的索引，但实际上可能会出现冲突（Collision），即不同的键被映射到相同的索引位置。为了解决冲突，常用的方法有链地址法和开放地址法。

链地址法 链地址法是将每个数组位置存储为一个链表（或其他数据结构），当发生冲突时，将新元素添加到该位置的链表中。查找时，需要遍历链表以找到目标元素。这样的实现方式在 C++ 的 `std::unordered_map` 和 `std::unordered_set` 中被广泛使用。

开放地址法 开放地址法是当发生冲突时，通过探测（Probing）找到下一个可用的数组位置来存储新元素。常见的探测方法有线性探测、二次探测和双重散列等。查找时，同样需要按照探测序列查找目标元素，直到找到或遇到空位置为止。这样的方法在某些哈希表实现中使用，但 C++ 标准库并未采用这种方法。

24.3.3 堆

堆是一种特殊的完全二叉树，满足堆性质（Heap Property）：对于最大堆（Max Heap），每个节点的值都大于或等于其子节点的值；对于最小堆（Min Heap），每个节点的值都小于或等于其子节点的值。堆常用于实现优先队列和排序算法（如堆排序）。

在 C++ 中的堆是通过 `std::priority_queue<T>` 来实现的：

```

1 #include <queue>
2 std::priority_queue<int> maxHeap; // 最大堆
3 std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap; // 最小堆
4 maxHeap.push(10); // 插入元素
5 int top = maxHeap.top(); // 获取堆顶元素
6 maxHeap.pop(); // 删除堆顶元素
    
```

优先队列实际上也是一个“队列”，但内部并不是简单的先进先出，而是根据元素的优先级来决定出队顺序。最大堆的优先级是值越大越高，最小堆则是值越小越高。在 C++ 中优先队列也是一个容器适配器，底层通常使用 `std::vector<T>` 来存储元素，并通过堆操作来维护堆性质；也可以自定义底层容器，但必须满足随机访问迭代器的要求。

堆的插入操作包括将新元素添加到堆的末尾，然后通过上浮（Bubble Up）操作恢复堆性质。上浮操作是将新元素与其父节点比较，如果违反堆性质，则交换它们的位置，直到堆性质得到恢复或到达根节点为止。这一过程大概是：

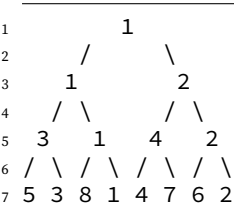
- 将新元素添加到堆的末尾。
- 计算新元素的父节点索引。
- 如果新元素违反堆性质（例如在最大堆中，新元素大于父节点），则交换它们的位置。
- 重复上述步骤，直到堆性质得到恢复或到达根节点为止。

这样的上浮是非常高效的，时间复杂度为 $O(\log n)$ ，比红黑树等自平衡树更快一些。因此在仅需要找到最大或最小元素的场景下，堆是一个非常好的选择。

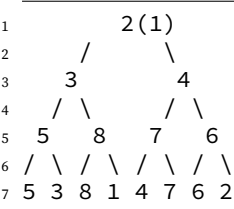
胜者树和败者树 胜者树（Winner Tree）和败者树（Loser Tree）是两种特殊的树结构，常用于多路归并排序和优先队列的实现。它们通过比较节点的值来确定“胜者”和“败者”，从而高效地进行元素的选择和排序，本质上实际是对“堆”的一种工业性改良。

那么具体是以一种什么方式来实现的呢？首先，先像淘汰赛一样，把所有元素排成一个完全二叉树的叶节点，然后从底向上比较每对节点，较小（或较大）的节点胜出，成为其父节点的值，这个过程一直持续到根节点。这样，根节点就存储了所有元素中的最小值（或最大值），这就是胜者树。而败者树相反，父节点存的是比较过程中“输掉”的那个节点的值，但决出祖父节点的方式依然是两个胜者比较，而不是败者比较；这样设计的好处是，在进行多路归并时，只需要更新败者树中的节点，而不需要重新比较所有节点，从而提高效率；而在根节点上一般也多一个辅助变量（或者专门的节点）来存储当前的最小值（或最大值，实际上就是最终的胜者）。

我们用一个例子来说明胜者树和败者树的构建过程。假设我们有 8 个元素：5, 3, 8, 1, 4, 7, 6, 2。我们知道，8 是 2 的整数次幂 (2^3)，因此可以直接构建一个完整的二叉树，无需考虑“轮空”或“补齐”的问题。对于胜者树，构建过程较为简单。假设我们取较小值为胜者，那么构建过程如下：



而败者树的构建过程则稍微复杂一些。假设我们取较小值为胜者，那么构建过程如下：



胜者树和败者树在多路归并排序中非常有用。假设我们有多组已排序的序列，需要将它们合并成一个有序序列。使用胜者树或败者树可以高效地找到当前最小（或最大）的元素，并将其添加到结果序列中，然后更新树结构以反映新的状态。

理解和掌握这些数据结构对于编写高效的程序至关重要。在实际编程中，我们通常会使用 C++ 标准库提供的容器和算法来简化数据结构的实现和操作，从而专注于解决具体的问题。

第二十五章 算法基础

我们知道，一个程序由两部分组成：数据和算法。数据是程序处理的对象，而算法则是处理数据的步骤和方法。算法在计算机科学中占据着核心地位，因为它们决定了程序的效率和性能。在本章中，我们将介绍算法的基本概念、分类以及常见的算法设计技巧。

25.1 算法及算法分析的基本概念

算法，简单地说就是一套方法或步骤，用于解决特定的问题。而在计算机上的算法，往往也有一种特殊的性质，即它们必须是可计算的（computable），也就是说，一个好的算法必须满足两个条件：

- 有穷性（Finiteness）：算法必须在有限的步骤内完成，不能无限循环。
- 确定性（Definiteness）：算法的每一步都必须是明确的，没有歧义。

实际上，很多算法实际上是无限终止的，例如遗传算法、模拟退火等。这些时候，我们往往会设置一个终止条件，例如达到一定的迭代次数或满足某个精度要求。但无论如何，这些算法在实际运行中都必须在规定时间内终止，否则就无法称之为一个好的算法。

25.1.1 衡量算法的效率

对于同一组数据，不同的算法往往会在不同的时间内完成任务。例如，我们要在一堆有序的数据中找到某个数值（不保证该数值一定在表中存在），我提出以下两种方法：

- 方法一：从头到尾依次检查每个数值，直到找到目标数值为止（线性查找）。在 C++ 中，该查找方式是 `std::find`。
- 方法二：先和表里的中间数值比较，如果目标数值较小，则继续在前半部分查找，否则在后半部分查找。重复这个过程，直到找到目标数值为止（二分查找）。在 C++ 中，该查找方式是 `std::binary_search`。

假设我们有 n 个数据。那么，我们可以列出一个表格，来比较这两种方法在不同数据规模下需要的比较次数：

算法	最好情况	平均情况	最坏情况	所用空间
线性查找	1	$n/2$	n	无需额外空间
二分查找	1	$\log_2 n$	$\log_2 n$	无需额外空间

上述表中二分查找的平均情况实际上并不准确，甚至说很难写出一个解析解，但大致是这个数值。

我们发现，算法的不同会导致程序比较的次数有一些差异。为了更好地描述算法的效率，我们引入以下两个定义：

定义 1 (时间复杂度). 时间复杂度 (Time Complexity) 是指算法在执行过程中所需的时间与输入数据规模之间的关系。

定义 2 (空间复杂度). 空间复杂度 (Space Complexity) 是指算法在执行过程中所需的存储空间与输入数据规模之间的关系。

而在实际表示中，有五种方法来描述算法的时间/空间复杂度，分别为：

- 大 O 符号 (Big O Notation)：假设算法在数据规模为 n 的时候所需要的时间为 $T(n)$ ，如果存在正常数 $c > 0$ 和 $n_0 > 0$ ，使得对于所有 $n \geq n_0$ 都有 $T(n) \leq c \cdot f(n)$ ，那么我们就说算法的时间复杂度为 $O(f(n))$ 。
- 大 Ω 符号 (Big Omega Notation)：如果存在正常数 $c > 0$ 和 $n_0 > 0$ ，使得对于所有 $n \geq n_0$ 都有 $T(n) \geq c \cdot f(n)$ ，那么我们就说算法的时间复杂度为 $\Omega(f(n))$ 。
- 大 Θ 符号 (Big Theta Notation)：如果算法的时间复杂度同时满足 $O(f(n))$ 和 $\Omega(f(n))$ ，那么我们就说算法的时间复杂度为 $\Theta(f(n))$ 。该符号也叫做“同阶”符号，表示算法的时间复杂度与 $f(n)$ 在数量级上是相同的。
- 小 o 符号 (Small o Notation)：如果对于所有正常数 $c > 0$ ，都存在 $n_0 > 0$ ，使得对于所有 $n \geq n_0$ 都有 $T(n) < c \cdot f(n)$ ，那么我们就说算法的时间复杂度为 $o(f(n))$ 。
- 小 ω 符号 (Small omega Notation)：如果对于所有正常数 $c > 0$ ，都存在 $n_0 > 0$ ，使得对于所有 $n \geq n_0$ 都有 $T(n) > c \cdot f(n)$ ，那么我们就说算法的时间复杂度为 $\omega(f(n))$ 。

在上述做法中，又属大 O 符号和大 Θ 符号最为常用。空间复杂度的表示类似，这里不再赘述。

常规情况下，我们会尽量使描述算法复杂度的辅助函数 $f(n)$ 尽可能地简单，也就是往往仅保留最“决定性”的一项。例如，假设一个算法的时间复杂度为 $T(n) = 3n^3 + 5n^2 + 2n + 8$ ，那么我们可以说该算法的时间复杂度为 $O(n^3)$ ，因为当 n 足够大时， n^3 项会主导整个函数的增长。而更简单起见，我们往往有一个“链条”，即

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n) \subset O(n!)$$

来表示不同复杂度的增长速度。

25.1.2 问题的复杂性及其初步判断

而问题的复杂性则是指解决某个问题所需的资源（时间和空间）与输入数据规模之间的关系。问题的复杂性和算法的复杂度密切相关却不完全相同：复杂度仅用来衡量某一个具体的算法的效率，而一个问题往往有较多算法来解决，问题的“复杂性”则是指所有可能的算法中，最优算法的复杂度。

我们举一个例子：在一些无序的数中，试着找到一个最大值。怎样确定该问题的复杂性呢？我们可以考虑所有可能的算法，发现无论采用何种方法，都至少需要检查每个数值一次，才能确保找到最大值。因此，该问题的复杂性为 $O(n)$ 。

实际上很多问题无法通过简单的分析来判断其复杂性。但对于一些简单的问题，判断复杂性可以辅助检查算法的正确性。例如，排序问题的复杂性下界为 $O(n \log n)$ ，因此如果我们设计了一个排序算法，其时间复杂度为 $O(n^2)$ ，那么我们就可以断定该算法不是最优的。

阅读材料：NP 完全问题简介

P 问题是指那些可以在多项式时间内由确定性图灵机^a解决的问题。换句话说，如果一个问题属于 P 类，那么存在一个算法能够在输入规模的多项式时间内找到该问题的解。P 问题很多，例如排序问题、查找问题等。比 P 问题复杂性更低的问题也有，例如 L 和 NL 问题，分别表示对数空间可解问题和非确定性对数空间可解问题。

而 NP 问题则是指那些可以在多项式时间内由非确定性图灵机^b解决的问题。换句话说，如果一个问题属于 NP 类，那么“验证该问题的一个给定解是正确的”是可以在多项式时间内完成的，或者说是一个 P 问题。

目前已经知道，所有的 P 问题都一定是 NP 问题，也就是 $P \subseteq NP$ 。但是否 $P = NP$ 则是一个未解问题，且被认为是计算机科学中最重要的未解问题之一。目前相当多人对该问题持悲观态度，但无论上述问题的答案如何，对计算机科学的影响都会是深远的。

而常说的“NP 难的”问题，则是指那些至少和 NP 问题一样难的问题。也就是说，如果一个问题 NP 难的，那么任何 NP 问题都可以在多项式时间内归约到该问题上。换句话说，解决这个 NP 难的问题至少和解决所有 NP 问题一样困难。那自然也有些 NP 难的问题本身就是 NP 问题，这类问题被称为 NP 完全问题。NP 完全问题是一个非常重要的概念，因为如果我们能够找到一个多项式时间算法来解决任何一个 NP 完全问题，那么我们就可以证明 $P = NP$ 。反之，如果我们能够证明没有多项式时间算法能够解决任何一个 NP 完全问题，那么我们就可以证明 $P \neq NP$ 。常见的 NP 完全问题包括旅行商问题、顶点覆盖问题、3-SAT 问题等。第一个被证明是 NP 完全的问题是布尔可满足性问题 (SAT)，这是由计算机科学家斯蒂芬·库克在 1971 年提出的，称为库克定理，之后证明所有问题是 NP 完全问题的过程都可以归结为将该问题归约到 SAT 问题上。

那有没有一些问题，其复杂性还要比 NP 完全问题更高？那显然是有的。不过绝大多数问题依然是 P 问题或 NP 问题，且大多数实际应用中的问题也都是 P 问题或 NP 问题。因此，在实际编程中，我们通常只需要关注 P 问题和 NP 问题，而不需要过多地考虑更高复杂度的问题。我们在这里给出一个链条，来表示不同复杂度类之间的关系：

$$P \subseteq NP \subseteq EXP \subseteq NEXP \subseteq REC \subsetneq RE \subsetneq ALL$$

EXP 表示那些可以在指数时间内由确定性图灵机解决的问题，可以直观理解为可以解但非常慢的问题，例如国际象棋的最优解问题。

NEXP 表示那些可以在指数时间内由非确定性图灵机解决的问题，例如一些复杂的组合优化问题。

REC 表示“图灵机可判定”的问题，可以直观理解为“有一个算法能在有限时间内给你一个正确答复”的问题，理论可解，无论时间多长。一个典型的例子是：命题逻辑的有效性（无论如何，总能通过真值表判断一个命题公式是否在所有解释下都成立）。这类问题已经非常复杂了，远超 NP 完全问题。

RE 表示“图灵机可半判定”的问题，也就是说，有一个算法能在有限时间内给你一个正确答复“是”，但不能保证在正确答案是“否”的情况下也能停机。两个典型的例子分别是：一阶逻辑的可满足性问题（一个公式是否存在一个模型使其成立）和停机问题（一个程序是否会停机）。这类问题已经是图灵机的极限了，无法再更复杂。

ALL 表示所有问题的集合，包括不可计算的问题，例如停机问题的补问题（一个程序是否不会停机，这是不可计算甚至不可验证的问题）。

^a确定性图灵机 (Deterministic Turing Machine, DTM) 是一种理论计算模型，用于描述计算过程中的确定性行为。在确

定性图灵机中，对于每一个状态和输入符号的组合，机器都有唯一的下一步操作，这意味着在任何给定的时间点，机器的行为是完全确定的。

^b非确定性图灵机（Nondeterministic Turing Machine, NTM）是一种理论计算模型，用于描述计算过程中的非确定性行为。在非确定性图灵机中，对于某一个状态和输入符号的组合，机器可以有多个可能的下一步操作，这意味着在任何给定的时间点，机器的行为是不确定的。非确定性图灵机可以被视为“并行”地尝试所有可能的计算路径，从而在某些情况下能够更快地找到问题的解。但非常遗憾的是，现代市面上所有的计算机都是确定性图灵机，或更准确的说是与确定性图灵机多项式时间等价的冯诺依曼机。量子计算机也不是非确定性图灵机，其对应的复杂度类是 BQP，不是 NP；现在 BQP 和 NP 之间的关系也是一个未解问题，但大家普遍认为这两个东西不互相包含。目前物理理论框架下唯一可能真是 NTM 的计算设备是生物大脑，但我们对其工作原理知之甚少，更遑论用其进行计算了。

25.2 常见精确算法思想

算法可以简单地分为两类：精确算法和近似算法。精确算法是指能够在有限时间内找到问题的最优解的算法，通常适用于那些问题规模较小或者对解的精度要求较高的场景；而近似算法则是“差不多就行”的算法，通常适用于那些问题规模较大或者对解的精度要求不高的场景。

25.2.1 查表

查表法指的是预先计算出问题所有可能的结果，然后根据需要直接查表得出结果。在日常生活中，我们经常看到查表的影子，例如速算 20 以内乘法时，我们往往会直接记住大乘法表，而不是每一次都重新计算一次；又如投篮时，我们实际上是根据经验来调整投篮的力度和角度（实际上可以认为是人脑在查表），而不是每次都重新计算抛物线。

查表严格说来不能算是一个“算法”，因为实际上该方法并没有提供一个解决问题的手段，而是通过牺牲空间来换取时间，从而达到快速解决问题的目的。查表法的优势是时间复杂度极低，结合哈希表等数据结构，查找时间可以达到 $O(1)$ 。缺点是它的空间复杂度往往很大，且实际上在大多数情况下因为可能的解太多了而并不实用，或者计算任务并非人力所能及¹。查表往往只在问题规模非常小、问题的解数量非常有限、对时间的要求非常高的场景下使用。

虽然很多人并不喜欢该方法，但在实际编程中，查表法却是一个非常实用的技巧，尤其是在需要频繁计算某些结果时，用查表替代计算往往能显著提升程序的性能。例如我们如果需要一些素数，可以预先计算出一定范围内的素数表，然后在需要时直接查表，而不是每次都重新计算。

例题

请编写一个程序，这个程序接受一些整数，然后输出这些整数是不是素数。

输入：m+1 行，第一行是一个整/数 m，表示接下来有 m 个整数需要判断；接下来的 m 行，每行一个整数 n。n 的数值在 0 到 1000 之间。

输出：m 行，每行一个字符串，表示对应的整数是不是素数。如果是素数，输出“YES”，否则输出“NO”，如果输入的是 1，输出“NA”。

¹当然，如果你能做到拉马努金“有人托梦给我”然后直接出答案的水平，那就另当别论了。

答案

我们可以先预处理出 1000 以内的所有素数，然后对于每个输入的整数，直接判断它是否在素数列表中即可。

我们知道，数组的索引天然自然数，非常适宜做这种标记，在筛完之后只需要按索引查询即可；想到“按索引查询”，立刻想到顺序表 `vector`。于是，我们得以模拟过程，得到如下代码：

```
1 vector<int> primes(1001, 1); // 初始化一个大小为 1001 的数组，初始值全为 1
2 primes[0] = primes[1] = 0; // 0 和 1 不是素数
3 for (int i = 2; i * i <= 1000; ++i) { // 遍历每个数
4     if (primes[i]) { // 如果是素数
5         for (int j = 2; i*j <= 1000; j++) { // 将它的倍数标记为非素数
6             primes[j*i] = 0;
7         }
8     }
9 }
```

这样，我们就得到了一个标记了 1000 以内所有素数的数组。接下来，我们只需要读取输入的整数，然后直接查询这个数组即可。查询部分代码留作练习。

25.2.2 模拟

模拟法指的是通过逐步模拟问题的过程来解决问题。这种方法通常适用于那些问题本身具有明确的步骤或过程，可以通过逐步执行这些步骤来达到最终目标。其优势在于它直观易懂，且适用于各种复杂的问题。缺点是时间复杂度往往较高，尤其是在问题规模较大时，模拟法可能会变得非常低效。

有时候，容易想到的模拟法并不是最好的算法，如果能够通过数学推导、逻辑分析等方法来优化算法，往往能得到更高效的解决方案。因此，在使用模拟法时，我们也要注意其效率，尽量避免不必要的计算和重复操作。

例题

给定一个 n ，试着求下列表达式的值：

$$42 \times \sum_{i=1}^n \sum_{j=1}^n (i+j)(ij), 1 \leq n \leq 998244353$$

提示：若 i 和 j 取值无关，则有 $\sum_i \sum_j f(i)g(j) = (\sum_i f(i))(\sum_j g(j))$

输入：一行，一个整数 n 。

输出：一行，一个整数，表示结果。

答案

分析上述题目，如果用双重循环枚举 i 和 j ，时间复杂度是 $O(n^2)$ 。我们已经知道， n 的最大值是 998244353，这个数量级的平方显然是无法接受的。我们需要对题目进行分析，推导出一个更高效的计算方法。另一方面，整数 `int` 的范围是 -2^{31} 到 $2^{31}-1$ ，大约是 -21 亿到 21 亿，而题目中要求的结果显然远远超过了这个范围，因此我们需要使用更大范围的整数类型 `long long`，该数据类型的范围大约

是-9 万亿到9 万亿，题目中的结果应该在这个范围内。

所以要求解这个问题，我们可以先将表达式进行展开。展开的过程略，总之最终结果是 $7n^2(n + 1)^2(2n + 1)$ 。直接输出这个结果即可。

代码过于简单，留作练习。

25.2.3 枚举

枚举法指的是**通过穷举所有可能的解来找到问题的答案**。这种方法通常适用于那些问题规模较小，或者解的数量有限的情况。其优势在于它能够确保找到最优解，且实现简单直观。缺点是时间复杂度往往较高，尤其是在问题规模较大时，枚举法可能会变得非常低效。

在枚举的时候，我们一定要尽可能地缩小枚举的范围、减少枚举的次数。例如，在求解组合问题时，我们可以通过剪枝、排序等方法来减少不必要的枚举，从而提升算法的效率。

例题

公鸡每只 5 元，母鸡每只 3 元，小鸡三只 1 元，100 元钱买 100 只鸡，请问公鸡、母鸡、小鸡各买多少只刚好凑足 100 元钱？请编写程序计算所有可能的购买方案。

输入：无。

输出：每行三个整数，分别表示公鸡、母鸡、小鸡的数量。按公鸡数量从小到大排序输出。

答案

该问题实际上可以抽象为以下方程组：

$$\begin{cases} x + y + z = 100 \\ 5x + 3y + \frac{1}{3}z = 100 \end{cases}$$

其中， x 表示公鸡的数量， y 表示母鸡的数量， z 表示小鸡的数量。一个简单的思路是写一个双重循环，枚举公鸡和母鸡的数量，然后计算出小鸡的数量，最后判断是否满足条件。显然，上述题目中公鸡和母鸡的数量都不会超过 20，因此枚举的时间复杂度 $O(n^2)$ 是可以接受的。

当然，如果我们能通过分析，推导出一个更高效的枚举方法，那就更好了。实际上我们可以将方程组两边同时乘以 3，得到：

$$\begin{cases} x + y + z = 100 \\ 15x + 9y + z = 300 \end{cases}$$

这样，我们就可以通过消元法，消去 z 并进一步化简，得到：

$$7x + 4y = 100$$

更进一步地，把 y 移到等式右边，我们发现 $7x$ 必须是 $25 - y$ 的 4 的倍数，因此我们可以直接以 4 为步长枚举 x ，这样就可以进一步减少枚举的次数。

代码实现如下：

```
1 for (int x = 0; x <= 25; x += 4) { // 枚举公鸡数量
2     int y = (100 - 7 * x) / 4; // 计算母鸡数量
3     int z = 100 - x - y; // 计算小鸡数量
4     if (y >= 0 && z >= 0 && (100 - 7 * x) % 4 == 0) { // 判断是否为非负整数
```

```
5      cout << x << " " << y << " " << z << endl; // 输出结果
6  }
7 }
```

25.2.4 贪心

贪心法指的是在每一步选择中都采取当前状态下最优的选择，从而希望通过一系列局部最优的选择来达到全局最优。这种方法通常适用于那些问题具有“贪心选择性质”的情况，即通过局部最优选择能够得到全局最优解。其优势在于它实现简单，且时间复杂度通常较低。缺点是，对于相当多的问题而言，贪心得到的解是错误的，因此在使用贪心法时，我们需要仔细分析问题，确保其满足贪心选择性质。

例题

有一个理发师要为一群顾客理发。每个顾客都有一个理发时间，理发师希望通过合理安排顾客的理发顺序，使得所有顾客的平均等待时间最小。请编写程序，计算最小的平均等待时间。

输入：第一行一个整数 n ，表示顾客的数量；第二行 n 个空格分隔的整数，表示每个顾客的理发时间。

输出：一行，一个整数，表示最小的平均等待时间（向下取整）。

答案

该问题可以通过贪心算法来解决。我们可以将顾客按照理发时间从小到大排序，然后依次为每个顾客理发。这样，理发时间较短的顾客会先被理发，从而减少了后续顾客的等待时间。

上述算法的正确性证明非常简单。不妨设有 n 个顾客，其理发时间分别为 $t_1, t_2, \dots, t_n$ 。对于排在第 i 个顾客，他的等待时间为前面所有顾客的理发时间之和，即：

$$W_i = \sum_{j=1}^{i-1} t_j$$

因此，所有顾客的总等待时间为：

$$W = \sum_{i=1}^n W_i = \sum_{i=1}^n \sum_{j=1}^{i-1} t_j$$

我们可以交换求和顺序，得到：

$$W = \sum_{j=1}^n t_j \cdot (n - j)$$

上述乘积求最小值看起来非常困难。但是我们只需要利用排序不等式：顺序和大于乱序和大于反序和。也就是说，我们只需要将 t_j 按从小到大排序，就能使得总等待时间最小。

25.2.5 分治和减治

分治法指的是将一个复杂的问题分解为若干个规模较小的子问题，分别解决这些子问题，然后将它们的解合并，得到原问题的解。这种方法通常适用于那些问题具有“最优子结构性”的情况，即原问题的最优解可以通过子问题的最优解来构建。其优势在于它能够显著降低

问题的复杂度，且适用于各种复杂的问题。缺点是实现相对复杂，且在某些情况下可能会导致重复计算。另，如果子问题之间不独立（即子问题的解会相互影响），或子问题的解决方法不一致，那么分治法就不适用了。

减治法是分治的一个变体。分治法在把问题分解之后，这些子问题都是要独立解决的；而减治法则不会解决被分解后的所有问题。相反，减治法通常只会解决其中的一个或几个子问题，然后通过这些子问题的解来构建原问题的解。减治法的优势在于它能够进一步降低问题的复杂度，且适用于那些子问题之间存在依赖关系的情况，例如二分查找等。缺点是实现相对复杂，且在某些情况下可能会导致重复计算。

例题

现有一堆芯片需要测试。每个芯片要么是好的，要么是坏的。现在有一种测试方法，可以将两个芯片放在一起进行测试。好的芯片总是能够正确地测试另一个芯片，而坏的芯片则无法保证测试结果的正确性。已经知道好芯片的数量比坏芯片多。请设计算法，用最少的测试次数找出一个好的芯片。

答案

我们可以使用类似淘汰赛的分治方式来找出一个好的芯片。我们将所有芯片两两配对进行测试，测试结果有三种可能：

- 两个芯片都报告对方是好的，那么它们要么都是好的，要么都是坏的。
- 一个芯片报告另一个是好的，而另一个报告第一个是坏的，那么至少有一个是坏的。
- 两个芯片都报告对方是坏的，那么至少有一个是坏的。

如果两个芯片都报告对方是好的，那么我们随机保留其中一个芯片并扔掉另一个（因为它们要么都是好的，要么都是坏的，这样并不会影响任何结果）；否则，就把两个全部扔掉。容易证明，这样一轮测试下来，剩下的芯片中好芯片的数量仍然多于坏芯片的数量。我们可以重复这个过程，直到只剩下一个芯片为止。这个芯片必然是好的。上述过程的时间复杂度是 $O(n)$ 。

很多分治算法都可以写成递归的形式，例如归并排序、快速排序等。递归的实现方式往往更加简洁明了，但需要注意递归的深度和栈空间的使用，避免出现栈溢出的问题。然而，所有的递归都可以写成显式栈的迭代形式，而怎样把递归写成显式栈则是一个需要练习的技巧，该技巧在实用主义的编程中能有效防止栈溢出，具体操作方式见上一章。

25.2.6 动态规划

动态规划法指的是将一个复杂的问题分解为若干个子问题，先解决这些子问题，然后通过子问题的解来构建原问题的解。其优势在于它能够显著降低问题的复杂度，且适用于各种复杂的问题。缺点是实现相对复杂，且需要额外的存储空间来保存子问题的解。

一个动态算法是正确的，当且仅当满足以下条件：

最优子结构性质 原问题的最优解可以通过子问题的最优解来构建。

重叠子问题性质 子问题之间存在重叠，即同一个子问题可能会被多次计算。

无后效性 一旦子问题的解被确定，它就不会再受到后续决策的影响。

如果一个问题不满足重叠子问题性质，那么它通常不适合使用动态规划来解决，而更适合使用分治法来解决。

动态规划往往以递推的形式实现，即通过迭代的方式逐步计算出子问题的解，最终得到原问题的解。动态规划的关键在于确定状态转移方程，即如何通过子问题的解来构建原问题的解。

值得注意的是贪心与动态规划之间的区别与联系。贪心和动态规划均通过局部决策来解决全局最优化问题，但**贪心每一步只看眼前最优，不能回头**，而动态规划会保存子问题状态，“更为综合”地保证全局最优。在实际解题中，关键在于判断一个问题更适合用贪心（需证明局部最优能导出全局最优）、动态规划（需设计合理的状态与转移），还是其他算法来解决；这一能力可以通过不断练习来加强。

计数和概率递推等非最优化问题的递推解法也常习惯性地称作“动态规划”，但严格来说它们并不属于动态规划的范畴。下文对此不做区分。

例题

小明正在爬楼梯。小明每次可能爬 1 级台阶，也可能爬 2 级台阶。假设小明要爬 n 级台阶，请问他有多少种不同的爬楼梯方式？
输入：一行，一个整数 n ，表示台阶的数量。
输出：一行，一个整数，表示不同的爬楼梯方式。

答案

该问题可以通过递推来解决。我们可以定义一个数组 dp ，其中 $dp[i]$ 表示爬到第 i 级台阶的不同方式数。显然， $dp[0]=1$ （不需要爬任何台阶）， $dp[1]=1$ （只有一种方式爬到第一级台阶）。对于 $i \geq 2$ ，我们可以通过以下递推关系来计算 $dp[i]$ ：
$$dp[i] = dp[i - 1] + dp[i - 2]$$
这是因为要到达第 i 级台阶，可以从第 $i - 1$ 级台阶爬 1 级上来，或者从第 $i - 2$ 级台阶爬 2 级上来。这个递推也可以认为是动态规划的最初级阶段。

对于新手而言，动态规划问题可能比较困难，需要仔细分析问题，找出合适的状态和递推关系。

例题

给定一个整数序列。定义**子序列**为从这个母序列中删除一些元素（也可以不删除）后，剩下的元素按原来的顺序排列形成的序列。现在要求它的一个子序列，该子序列的每一个元素都大于等于前一个元素，且该子序列的长度尽可能大。请编写程序，计算这个子序列的长度。
输入：第一行一个整数 n ，表示序列的长度；第二行 n 个空格分隔的整数，表示序列的元素。
输出：一行，一个整数，表示最长非降子序列的长度。

答案

上述问题就是经典的**最长非降子序列**问题。这个问题向来是萌新杀手，难点在怎么定义状态和递推关系。能够自己推导出状态和递推关系的同学，已经掌握了动态规划的精髓。
下面我们来分析这个问题。一个朴素的想法是，不妨设 $dp[i]$ 表示以第 i 个元素结尾的最长非降子

序列的长度。对于任何元素，我们都遍历它前面的所有元素，如果前面的元素 A_j 小于等于当前元素 A_i ，那么我们就把 A_i 接到 A_j 的后面，这样就形成了一个更长的非降子序列。因此，我们可以得到以下递推关系：

$$\forall A_j \leq A_i, \forall j < i, dp[i] = \max(dp[j] + 1)$$

代码写出来如下：

```

1 vector<int> dp(n, 1); // 初始化 dp 数组，初始值为 1
2 for (int i = 1; i < n; ++i) { // 遍历每个元素
3     for (int j = 0; j < i; ++j) { // 遍历前面的元素
4         if (A[j] <= A[i]) { // 如果前面的元素小于等于当前元素
5             dp[i] = max(dp[i], dp[j] + 1); // 更新 dp[i]
6         }
7     }
8 }
```

如果不懂，可以举一个例子，在纸上或者脑子里模拟一下上述过程，大概就能理解了。上述算法的时间复杂度是 $O(n^2)$ ，对于 n 在几千以内的情况是可以接受的。

回到刚才，我们是用枚举法来求符合条件的 j ，进而求出 $dp[i]$ 。如果我们能用更高效的方法来求出符合条件的 j ，就能进一步提高算法效率。我们知道，枚举是很笨的，有没有更高效的算法来解决这个问题呢？答案是有的。

我们跳出“模拟”的思维，换个角度来看这个问题。我们不再关心这个长度是怎么一步步推出来的，而是直接关注长度本身：**如果我们想构造一个尽可能长的非降子序列，那么在每一步，我们都希望这个序列的“尾巴”尽可能小，这样后面才能接上更多的数。**

局势瞬间明朗了起来。我们可以维护一个数组 `tails`，其中 `tails[k]` 表示长度为 $k+1$ 的非降子序列的最小结尾元素。怎么构建这个数组呢？我们只需要从左往右遍历原序列，对于每一个元素 A_i ，我们在 `tails` 中找到第一个大于 A_i 的元素，并将其替换为 A_i 。如果没有找到，则将 A_i 添加到 `tails` 的末尾。这样，`tails` 数组的长度就是最长非降子序列的长度，顺便还能得到一个这样的子序列。

容易证明，`tails` 数组是严格非降的，因此我们可以使用二分查找来提高查找效率。这样，整个算法的时间复杂度就降到了 $O(n \log n)$ 。

```

1 vector<int> nums = ...; // 输入的整数序列
2 vector<int> tails;
3 for (int num : nums) {
4     auto it = upper_bound(tails.begin(), tails.end(), num); // 找到第一个 >
        // num 的位置
5     if (it == tails.end()) {
6         tails.push_back(num); // 可以扩展序列
7     } else {
8         *it = num; // 替换掉更大的末尾
9     }
10 }
```

25.2.7 回溯和分支限界

回溯法指的是通过逐步构建解的候选解空间，并在发现某个候选解不满足条件时，回溯到上一步继续尝试其他可能的解。这种方法通常适用于那些问题具有“约束满足性质”的情况。其优势在于它能够找到所有满足条件的解，且适用于各种复杂的问题。缺点是时间复杂度往往

较高，时间复杂度是指数级别的，尤其是在问题规模较大时，回溯法可能会变得非常低效。

为了杜绝上述低效问题，我们应该尽可能地减少回溯的次数，这就需要在回溯过程中进行剪枝，即在发现某个候选解一定不满足条件时，立即停止继续尝试该候选解的后续分支，从而减少不必要的计算。这就是“分支限界法”的思想。常见的剪枝方法包括可行性剪枝、最优性剪枝、启发式剪枝等。

可行性剪枝 在回溯过程中，如果发现当前候选解已经不满足问题的约束条件，那么就立即停止继续尝试该候选解的后续分支。

最优性剪枝 在回溯过程中，如果发现当前候选解已经无法达到最优解，那么就立即停止继续尝试该候选解的后续分支。

启发式剪枝 在回溯过程中，利用问题的特性或经验知识，提前判断某个候选解是否有可能满足条件，从而决定是否继续尝试该候选解的后续分支。

例题

国际象棋中，“后”这个棋子可以攻击同一行、同一列以及同一对角线上任意距离的棋子。现在我们在一个标准的国际象棋棋盘（8x8）上放置 8 个后，要求这些后互不攻击。请编写程序，计算所有可能的放置方案。

输入：无。

输出：每行 8 个整数，表示一个放置方案，每个整数表示该行后所在的列（从 0 开始计数）。按字典序排序输出所有方案。不考虑旋转对称性和镜像对称性。

答案

该问题可以通过回溯法来解决。我们可以逐行放置后，对于每一行，我们尝试将后放在每一列上，并检查是否与之前放置的后冲突。如果不冲突，则继续放置下一行的后；如果冲突，则回溯到上一行，尝试其他列。这样，我们可以遍历所有可能的放置方案。

具体实现时，我们可以使用一个数组 `positions` 来记录每一行后所在的列。我们还需要一个函数来检查当前放置的后是否与之前的后冲突。代码实现如下：

```
1 void solveNQueens(int row, vector<int>& positions, vector<vector<int>>&
   ↳ results) {
2     if (row == 8) { // 所有行都放置完毕
3         results.push_back(positions); // 记录当前方案
4         return;
5     }
6     for (int col = 0; col < 8; ++col) { // 尝试将后放在当前行的每一列
7         bool conflict = false;
8         for (int prevRow = 0; prevRow < row; ++prevRow) { // 检查与之前放置的后
   ↳ 是否冲突
9             int prevCol = positions[prevRow];
10            if (prevCol == col || abs(prevCol - col) == abs(prevRow - row))
   ↳ {
11                conflict = true;
12                break;
13            }
14        }
15        if (!conflict) { // 如果不冲突，继续放置下一行的后
16            positions[row] = col;
17            solveNQueens(row + 1, positions, results);
```


18 }
19 }
20 }

最终，我们可以调用上述函数来计算所有可能的放置方案，并将结果输出。

25.2.8 线性规划

线性规划法指的是通过构建线性目标函数和线性约束条件，来求解最优化问题。这种方法通常适用于那些问题可以用线性方程组或不等式组来描述的情况。其优势在于它能够高效地求解大规模的最优化问题，且有成熟的算法和工具可供使用。缺点是它仅适用于线性问题，对于非线性问题则无法直接应用。

例题

某工厂生产两种产品 A 和 B。每种产品的利润分别为 3 元和 5 元。生产每种产品需要消耗一定的资源：生产 1 个 A 需要 2 单位资源，生产 1 个 B 需要 3 单位资源。工厂每天可用的资源总量为 18 单位。请问工厂每天应该生产多少个 A 和 B，以最大化利润？
输入：无。
输出：一行，两个整数，分别表示每天生产的 A 和 B 的数量。

答案

该问题可以通过线性规划来解决。我们可以定义变量 x 和 y ，分别表示每天生产的 A 和 B 的数量。我们的目标是最大化利润函数：
$$\text{Maximize } Z = 3x + 5y$$

同时，我们需要满足资源约束条件：
$$2x + 3y \leq 18$$
$$x \geq 0, y \geq 0$$

我们可以使用单纯形法或其他线性规划算法来求解这个问题。通过计算，我们可以得到最优解为 $x = 3, y = 4$ ，即每天生产 3 个 A 和 4 个 B，可以最大化利润。

为了帮助同学们解决上述问题，我们介绍一下“单纯形法”的基本思想。单纯形法是一种用于求解线性规划问题的迭代算法。其基本思想是从一个可行解出发，沿着可行域的边界移动，逐步找到更优的解，直到达到最优解为止。具体步骤如下：

- 初始化：选择一个初始可行解，通常是通过引入松弛变量来构建一个基本可行解。
- 迭代：在当前可行解的邻域中寻找一个更优的解。如果找到了更优的解，则更新当前解；否则，终止迭代。
- 终止：当无法找到更优的解时，当前解即为最优解。

以上述题目为例，我们先将约束条件转化为等式形式，引入松弛变量 s ，得到：

$$\begin{cases} 2x + 3y + s = 18 \\ x \geq 0, y \geq 0, s \geq 0 \end{cases}$$

然后，我们可以利用枚举等手段，并通过迭代过程（枚举进入基变量和离开基变量）逐步优化，直到无法继续优化为止。最终，我们可以得到最优解。

当然，也有其他的手段来解决动态规划问题，例如使用现成的线性规划库（如 GLPK、CPLEX 等）来求解，或数形结合（仅限于几何意义明确的问题）。线性规划的应用范围非常广泛，涵盖了经济学、运筹学、工程学等多个领域，是一种非常重要的优化工具。

25.3 常见近似算法思想

25.3.1 随机化算法

随机化算法指的是在算法的某些步骤中引入随机性，从而提高算法的效率或简化算法的设计。这种方法通常适用于那些问题规模较大，或者对解的精度要求不高的情况。其优势在于它能够显著降低问题的复杂度，且实现相对简单。缺点是结果具有一定的随机性，可能无法保证每次运行都得到最优解。

随机化算法可以分为两类：蒙特卡洛算法和拉斯维加斯算法。蒙特卡洛算法在有限时间内给出一个近似解，且有一定概率得到正确解；而拉斯维加斯算法则保证每次运行都能得到正确解，但运行时间是随机的。

例题

在一场拍卖会中，我们希望选择一个出价高的竞拍者。一旦选定了某个竞拍者，就不能更改选择；我们也没有手段能够提前得知这些竞拍者的出价情况。这些竞拍者的出价是一个随机的序列。我们怎样才能使得选择的竞拍者的出价是最高价的概率最大呢？

答案

我们可以使用随机化算法来解决这个问题。具体来说，我们先观察前 r 个竞拍者的出价，记录下其中的最高价 M 。然后，从第 $r + 1$ 个竞拍者开始，如果某个竞拍者的出价高于 M ，我们就选择这个竞拍者并结束选择；否则，继续观察下一个竞拍者。这样，我们可以最大化选择到最高价竞拍者的概率。现在的问题就变成，怎样求解 r 才能使得选择到最高价竞拍者的概率最大。

假设最高价出在位置 k 。我们能选到这个人当且仅当：

- $k > r$ ，即最高价出现在我们观察的竞拍者之后；
- 从 $r + 1$ 到 $k - 1$ 的竞拍者中，没有人出价高于 k 。

因为竞拍者的出价是随机排列的，最高价出现在位置 k 的概率为 $\frac{1}{n}$ 。因此，我们有：

$$P(\text{选中最优} \mid \text{最优在位置} k) = P(\text{前} k - 1 \text{ 人中最优在前} r \text{ 人}) = \frac{r}{k - 1}$$

所以总的选中最优的概率为：

$$P(r) = \sum_{k=r+1}^n P(\text{最优在} k) \cdot P(\text{选中} \mid \text{最优在} k) = \sum_{k=r+1}^n \frac{1}{n} \cdot \frac{r}{k - 1} = \frac{r}{n} \sum_{k=r+1}^n \frac{1}{k - 1} = \frac{r}{n} \sum_{j=r}^{n-1} \frac{1}{j}$$

(令 $j = k - 1$)

当 r 较大时, $\sum_{j=r}^{n-1} \frac{1}{j}$ 可以近似为 $\ln(n) - \ln(r)$ 。因此, 我们设 $x = r/n$, 则有:

$$P(x) \approx x(\ln(n) - \ln(nx)) = x(\ln(n) - \ln(n) - \ln(x)) = -x \ln(x)$$

还记得高等数学的同学们可以对上述函数求导得到 $x = 1/e$ 时取得最大值。因此, 我们可以选择 $r = n/e$, 即观察前约 37% 的竞拍者, 然后选择第一个出价高于观察到的最高价的竞拍者。这样, 我们就能最大化选择到最高价竞拍者的概率。

上述算法是一个蒙特卡洛算法, 因为它在有限时间内给出了一个近似解, 且有一定概率得到正确解。拉斯维加斯算法则不是很常见, 但在特定的领域比较常用, 例如随机快速排序等。我们会在后续章节中介绍相关内容。

25.3.2 启发式算法

启发式算法指的是通过经验、直觉或启发式规则来指导搜索过程, 从而找到问题的近似解。这种方法通常适用于那些问题规模较大, 或者对解的精度要求不高的情况。其优势在于它能够显著降低问题的复杂度, 且实现相对简单。缺点是结果具有一定的随机性, 可能无法保证每次运行都得到最优解。

下文中的 TSP, 指的是旅行商问题 (Traveling Salesman Problem), 即给定一组城市和它们之间的距离, 要求找到一条最短的路径, 使得旅行商能够访问每个城市一次并返回起点。TSP 是一个 NP 完全问题, 当路径数量极多的时候, 常规的精确算法 (如动态规划、分支限界法等) 完全无法在可以接受的时间内求解 (上述算法的时间复杂度均为指数级别)。因此, 启发式算法在解决 TSP 问题中得到了广泛应用。

贪心 贪心是最简单的启发式算法。其基本思想和上文精确算法中的“贪心”其实完全一致, 甚至不如说上文中的“贪心”实际上是该算法“算对了”的一种特殊情况。具体来说, 贪心算法从一个初始城市出发, 每次选择距离当前城市最近的未访问城市作为下一个访问的城市, 直到所有城市都被访问过为止。最后, 返回起点, 完成整个路径。贪心算法实现简单, 且时间复杂度较低, 但其结果往往不是最优解, 因为它只考虑了局部最优选择, 而忽略了全局最优解。

爬山 爬山算法是一种简单的启发式算法, 用于在极大的搜索空间中寻找近似最优解。其基本思想是从一个初始解出发, 逐步探索邻域解, 如果找到一个更优的邻域解, 就移动到该解, 并继续搜索, 直到无法找到更优的邻域解为止。这样的“初始解”有很多方式来取得, 例如随机、贪心、特定构造都可以。示例伪代码中的“邻域”可以通过交换路径中的两个城市来定义。爬山算法实现简单, 但是容易陷入局部最优解, 无法保证找到全局最优解。

模拟退火 模拟退火算法是一种基于物理退火过程的启发式算法, 用于在复杂的搜索空间中寻找近似最优解。其基本思想是通过引入随机性, 偶尔接受一个较差的解, 来避免陷入局部最优。具体来说, 模拟退火算法从一个初始解出发, 在每一步中随机选择一个邻域解, 如果该邻域解更优, 则接受该解; 如果该邻域解更差, 则以一定概率接受该解, 这个概率随着时间的推移逐渐降低 (类似于物理退火过程中的温度降低)。这样, 算法在初期可以探索更多的解空间,

Algorithm 1: 贪心算法解决 TSP 问题的伪代码

Input: 城市集合 C , 距离矩阵 D

Output: 近似最优路径 P

```
1  $P \leftarrow$  空路径;  
2  $visited \leftarrow$  空集合;  
3  $current \leftarrow$  随机选择一个初始城市;  
4 添加  $current$  到  $P$  和  $visited$ ;  
5 while 未访问的城市不为空 do  
6    $next \leftarrow$  在未访问的城市中选择距离  $current$  最近的城市;  
7   添加  $next$  到  $P$  和  $visited$ ;  
8    $current \leftarrow next$ ;  
9 添加起点到  $P$ ;  
10 return  $P$ ;
```

Algorithm 2: 爬山算法解决 TSP 问题的伪代码

Input: 城市集合 C , 距离矩阵 D

Output: 近似最优路径 P

```
1  $P \leftarrow$  随机生成一个初始路径;  
2 while 存在更优的邻域路径 do  
3    $P' \leftarrow$  在  $P$  的邻域中找到一个更优的路径;  
4   if 路径  $P'$  的长度小于路径  $P$  的长度 then  
5      $P \leftarrow P'$ ;  
6 return  $P$ ;
```

随着时间的推移逐渐收敛到一个较优解。示例伪代码中的“邻域”同样可以通过交换路径中的两个城市来定义。模拟退火算法能够有效地避免陷入局部最优解，但其性能依赖于参数的选择，如初始温度和冷却速率。

Algorithm 3: 模拟退火算法解决 TSP 问题的伪代码

Input: 城市集合 C ，距离矩阵 D ，初始温度 T_0 ，冷却速率 α

Output: 近似最优路径 P

```
1  $P \leftarrow$  随机生成一个初始路径;
2  $T \leftarrow T_0$ ;
3 while  $T > T_{min}$  do
4    $P' \leftarrow$  在  $P$  的邻域中随机选择一个路径;
5    $\Delta E \leftarrow$  路径  $P'$  的长度 - 路径  $P$  的长度;
6   if  $\Delta E < 0$  then
7      $P \leftarrow P'$ ;
8   else
9      $\quad$  以概率  $e^{-\Delta E/T}$  接受路径  $P'$ ;
10   $T \leftarrow \alpha \cdot T$ ;
11 return  $P$ ;
```

遗传算法 遗传算法是一种基于自然选择和遗传学原理的启发式算法，用于在复杂的搜索空间中寻找近似最优解。其基本思想是通过模拟生物进化过程中的选择、交叉和变异等机制，逐步优化解的质量。具体来说，遗传算法从一个初始种群（即一组解）出发，通过评估每个解的适应度，选择适应度较高的解进行交叉和变异，生成新的种群。这个过程不断重复，直到满足终止条件（如达到最大迭代次数或找到足够好的解）。遗传算法广泛应用于各种优化问题，示例伪代码中的“交叉”可以通过交换两个路径的部分片段来实现，“变异”可以通过随机交换路径中的两个城市来实现；适应度通常定义为路径的总长度的倒数或相反数等。

Algorithm 4: 遗传算法解决 TSP 问题的伪代码

Input: 城市集合 C ，距离矩阵 D ，种群大小 N ，最大迭代次数 G

Output: 近似最优路径 P

```
1 初始化种群  $P_0$ ，包含  $N$  个随机生成的路径;
2 for  $g = 1$  to  $G$  do
3   评估种群中每个路径的适应度;
4   选择适应度较高的路径进行交叉和变异，生成新的种群  $P_g$ ;
5 return 种群中适应度最高的路径;
```

遗传算法的优点是其全局搜索能力很强，能够有效地探索复杂的解空间；缺点是计算量较大，且参数调节较为复杂。在著名科幻小说《三体》第一部中，作者刘慈欣提到“遗传算法”这一概念，并将其作为故事情节的一部分，展示了科学与文学的结合。其具体操作为：通过随机初始化许多个三体问题的初始条件（如三个恒星的位置、速度），对其进行路径模拟，评估每

个初始条件下三体系统的稳定性（适应度），选择适应度较高的初始条件进行交叉和变异，生成新的初始条件。经过多代迭代，最终找到一个较为稳定的三体系统配置。这一过程体现了遗传算法在解决复杂物理问题中的应用，同时也为小说增添了科学色彩。

蚁群算法 蚁群算法是一种基于蚂蚁觅食行为的启发式算法，用于在复杂的搜索空间中寻找近似最优解。其基本思想是通过模拟蚂蚁在寻找食物过程中释放信息素的行为，逐步优化解的质量。具体来说，蚁群算法从一个初始解出发，模拟多只蚂蚁在解空间中移动，每只蚂蚁根据路径上的信息素浓度选择下一步的移动方向。路径越短的信息素浓度越高，吸引更多的蚂蚁选择该路径。随着时间的推移，信息素会挥发，促使蚂蚁探索新的路径。示例伪代码中的“启发式信息”可以是城市之间的距离的倒数，表示距离越近的城市越有吸引力。蚁群算法的优点是其分布式搜索能力强，能够有效地探索复杂的解空间；缺点是计算量较大，且参数调节较为复杂。

Algorithm 5: 蚁群算法解决 TSP 问题的伪代码

Input: 城市集合 C ，距离矩阵 D ，蚂蚁数量 M ，最大迭代次数 G

Output: 近似最优路径 P

```
1 初始化信息素矩阵  $\tau$ ;  
2 for  $g = 1$  to  $G$  do  
3   for 每只蚂蚁  $i = 1$  to  $M$  do  
4     构建路径  $P_i$ ，根据信息素浓度和启发式信息选择下一城市;  
5     计算路径  $P_i$  的长度;  
6   更新信息素矩阵  $\tau$ ，根据所有蚂蚁的路径长度调整信息素浓度;  
7 return 所有蚂蚁中路径长度最短的路径;
```

A* 搜索 A* 搜索算法是一种基于启发式搜索的路径规划算法，用于在图中寻找从起点到终点的最短路径。其基本思想是结合实际代价和启发式估计来指导搜索过程，从而高效地找到最优解。具体来说，A* 算法维护一个优先队列，存储待扩展的节点。每个节点都有一个评估函数 $f(n) = g(n) + h(n)$ ，其中 $g(n)$ 表示从起点到当前节点的实际代价， $h(n)$ 表示从当前节点到终点的启发式估计代价。A* 算法每次从优先队列中选择 $f(n)$ 值最小的节点进行扩展，直到找到终点为止。示例伪代码中的“启发函数”可以是当前节点到终点的最短距离的估计，也可能是其他更复杂的估计方法。A* 搜索算法的优点是其能够高效地找到相对较优的解，且适用于各种路径规划问题，但启发式函数的选择对算法性能有较大影响。特别的，当启发函数满足可采纳性（即不高估实际代价）和一致性（即满足三角不等式）时，A* 算法能够保证找到最优解。

25.4 搜索和排序

搜索和排序是两个基本的算法问题，广泛应用于各种计算任务中。搜索问题涉及在数据结构中查找特定元素，而排序问题则涉及将一组元素按照某种顺序排列。下面我们将介绍一些常见的搜索和排序算法（在线性的数据结构上进行搜索和排序，暂不考虑非线性的数据结构如

Algorithm 6: A* 搜索算法解决 TSP 问题的伪代码

Input: 城市集合 C , 距离矩阵 D , 启发式函数 $h(n)$
Output: 近似最优路径 P

1

初始化优先队列 Q ;

2

将起点节点加入队列 Q ;

3

while 队列 Q 不为空 **do**

4

$n \leftarrow$ 从队列 Q 中取出 $f(n)$ 值最小的节点;

5

if n 是终点节点 **then**

6

return 从起点到终点的路径;

7

扩展节点 n 的邻居节点, 并计算它们的 $f(n)$ 值;

8

将邻居节点加入队列 Q ;

树、图等), 且这里不涉及并行和分布式算法, 也不涉及非精确的排序和搜索算法 (如近似算法、概率算法等)。

25.4.1 排序算法

排序算法用于将一组元素按照某种顺序排列。

冒泡排序 冒泡排序重复地遍历要排序的数列, 一次比较两个元素, 如果它们的顺序错误就把它们交换过来。遍历数列的工作是重复进行直到没有再需要交换, 也就是说该数列已经排序完成。

```
1 vector<int> arr = ...; // 输入的数组
2 int n = arr.size();
3 for (int i = 0; i < n - 1; ++i) {
4     for (int j = 0; j < n - i - 1; ++j) {
5         if (arr[j] > arr[j + 1]) {
6             swap(arr[j], arr[j + 1]); // 交换
7         }
8     }
9 }
```

选择排序 选择排序的工作原理是每次从未排序的部分中选择最小 (或最大) 的元素, 放到已排序部分的末尾。

```
1 vector<int> arr = ...; // 输入的数组
2 int n = arr.size();
3 for (int i = 0; i < n - 1; ++i) {
4     int min_idx = i; // 假设当前元素是最小的
5     for (int j = i + 1; j < n; ++j) {
6         if (arr[j] < arr[min_idx]) {
7             min_idx = j; // 更新最小元素的索引
8         }
9     }
```

```
10     swap(arr[i], arr[min_idx]); // 交换
11 }
```

插入排序 插入排序的工作原理是从左向右地构建有序序列, 对于未排序数据, 在已排序序列中从后向前扫描, 找到相应位置并插入。

```
1 for (int i = 1; i < n; ++i) {
2     int key = arr[i]; // 当前元素
3     int j = i - 1;
4     while (j >= 0 && arr[j] > key) {
5         arr[j + 1] = arr[j]; // 向后移动
6         j--;
7     }
8     arr[j + 1] = key; // 插入
9 }
```

快速排序 快速排序的工作原理是通过一个“基准”元素将数组分成两部分, 左边部分的元素都小于基准, 右边部分的元素都大于基准, 然后递归地对这两部分进行排序。快速排序是一种分治算法。

```
1 void quicksort(int left, int right) {
2     if (left >= right) return;
3     int pivot = arr[(left + right) / 2]; // 选择基准
4     int i = left, j = right;
5     while (i <= j) {
6         while (arr[i] < pivot) i++;
7         while (arr[j] > pivot) j--;
8         if (i <= j) {
9             swap(arr[i], arr[j]); // 交换
10            i++;
11            j--;
12        }
13    }
14    quicksort(left, j); // 递归排序左半部分
15    quicksort(i, right); // 递归排序右半部分
16 }
```

快排有一个变种: 随机快排, 即每次随机选择一个基准元素, 从而避免在某些特定情况下退化为 $O(n^2)$ 的时间复杂度。该算法是一个拉斯维加斯算法, 因为它保证每次运行都能得到正确解, 但运行时间是随机的。

归并排序 归并排序的工作原理是将数组分成两半, 分别对这两半进行排序, 然后将排序好的两半合并在一起。归并排序也是一种分治算法。

```
1 void merge(int left, int mid, int right) {
2     int n1 = mid - left + 1;
3     int n2 = right - mid;
4     vector<int> L(n1), R(n2);
```

```

5     for (int i = 0; i < n1; ++i) L[i] = arr[left + i];
6     for (int j = 0; j < n2; ++j) R[j] = arr[mid + 1 + j];
7     int i = 0, j = 0, k = left;
8     while (i < n1 && j < n2) {
9         if (L[i] <= R[j]) arr[k++] = L[i++];
10        else arr[k++] = R[j++];
11    }
12    while (i < n1) arr[k++] = L[i++];
13    while (j < n2) arr[k++] = R[j++];
14 }
15 void mergesort(int left, int right) {
16     if (left >= right) return;
17     int mid = left + (right - left) / 2;
18     mergesort(left, mid); // 递归排序左半部分
19     mergesort(mid + 1, right); // 递归排序右半部分
20     merge(left, mid, right); // 合并
21 }

```

堆排序 堆排序的工作原理是将数组构建成一个最大堆（或最小堆），然后不断地将堆顶元素与数组的最后一个元素交换，并将堆的大小减1，重新调整堆，直到堆为空。

```

1 void heapify(int n, int i) {
2     int largest = i; // 初始化最大元素为根节点
3     int left = 2 * i + 1; // 左子节点
4     int right = 2 * i + 2; // 右子节点
5     if (left < n && arr[left] > arr[largest]) largest = left;
6     if (right < n && arr[right] > arr[largest]) largest = right;
7     if (largest != i) {
8         swap(arr[i], arr[largest]); // 交换
9         heapify(n, largest); // 递归调整子树
10    }
11 }
12 void heapsort() {
13     int n = arr.size();
14     for (int i = n / 2 - 1; i >= 0; --i) heapify(n, i); // 构建最大堆
15     for (int i = n - 1; i > 0; --i) {
16         swap(arr[0], arr[i]); // 交换堆顶和最后一个元素
17         heapify(i, 0); // 重新调整堆
18     }
19 }

```

如不这么写，也可以用 C++ 现成的堆函数来实现堆排序：

```

1 vector<int> arr = ...; // 输入的数组
2 make_heap(arr.begin(), arr.end()); // 构建堆
3 sort_heap(arr.begin(), arr.end()); // 堆排序

```

更或者直接使用 `std::priority_queue` 这样的堆：

```

1 priority_queue<int> pq; // 创建最大堆
2 for (int num : arr) {
3     pq.push(num); // 插入元素
4 }

```

```
5 vector<int> sorted_arr;
6 while (!pq.empty()) {
7     sorted_arr.push_back(pq.top()); // 获取堆顶元素
8     pq.pop(); // 删除堆顶元素
9 }
10 reverse(sorted_arr.begin(), sorted_arr.end()); // 反转得到升序排列
```

排序算法	时间复杂度（平均）	时间复杂度（最坏）	空间复杂度
冒泡排序	$\Theta(n^2)$	$O(n^2)$	$O(1)$
选择排序	$\Theta(n^2)$	$O(n^2)$	$O(1)$
插入排序	$\Theta(n^2)$	$O(n^2)$	$O(1)$
快速排序	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
归并排序	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
堆排序	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$

表 25.1: 常见排序算法的时间和空间复杂度比较

由上表可见，快速排序在平均情况下表现较好，但在最坏情况下可能退化为 $O(n^2)$ 。归并排序和堆排序在最坏情况下也能保持 $O(n \log n)$ 的时间复杂度。冒泡排序、选择排序和插入排序由于其简单性，适合处理小规模数据，但对于大规模数据则效率较低。

对于大多数情况，我们最好不要手写排序算法，而是直接使用标准库中的排序函数，如 C++ 的 `std::sort`，它通常混合了多种排序算法，能够高效地处理各种规模的数据。

25.4.2 搜索算法

搜索（查找）算法用于在数据结构中查找特定元素。

线性搜索 线性搜索基本思想是从数据结构的第一个元素开始，逐个比较每个元素与目标元素是否相等，直到找到目标元素或遍历完整个数据结构为止。最常见的线性搜索例如：找最大值、找最小值、在无序的数组中确认某个值的索引等。

```
1 vector<int> arr = ...; // 输入的数组
2 int target = ...; // 目标值
3 for (int i = 0; i < arr.size(); ++i) {
4     if (arr[i] == target) {
5         cout << "Found at index " << i << endl;
6         break;
7     }
8 }
```

双向搜索 双向搜索和普通的线性搜索类似，但是它从起点和终点同时开始搜索，直到两个搜索相遇。这一算法常用于状态空间对称的问题，例如在图中寻找最短路径等。在线性数据结构上，这个算法并不常见（但有时候也有奇效，例如其变体“双指针法”），但在图论中非常有用。

Algorithm 7: 双向搜索

Input: 图 G , 起点 s , 终点 t

Output: 从 s 到 t 的路径

```
1 初始化两个队列  $Q_s$  和  $Q_t$ , 分别用于从  $s$  和  $t$  开始的搜索;
2 初始化两个集合  $visited_s$  和  $visited_t$ , 分别记录从  $s$  和  $t$  访问过的节点;
3 将  $s$  加入队列  $Q_s$  和集合  $visited_s$ ;
4 将  $t$  加入队列  $Q_t$  和集合  $visited_t$ ;
5 while 队列  $Q_s$  和  $Q_t$  均不为空 do
6     从队列  $Q_s$  中取出一个节点进行扩展;
7     如果该节点在集合  $visited_t$  中, 则找到路径, 返回结果;
8     否则, 将该节点的邻居节点加入队列  $Q_s$  和集合  $visited_s$ ;
9     从队列  $Q_t$  中取出一个节点进行扩展;
10    如果该节点在集合  $visited_s$  中, 则找到路径, 返回结果;
11    否则, 将该节点的邻居节点加入队列  $Q_t$  和集合  $visited_t$ ;
```

二分搜索 二分查找是一种高效的搜索算法, 适用于**有序数组**。它通过不断地将搜索范围缩小一半来查找目标值。二分查找也是一种分治算法。

```
1 vector<int> arr = ...; // 输入的有序数组
2 int target = ...; // 目标值
3 int left = 0, right = arr.size() - 1;
4 while (left <= right) {
5     int mid = left + (right - left) / 2; // 防止溢出
6     if (arr[mid] == target) {
7         cout << "Found at index " << mid << endl;
8         break;
9     } else if (arr[mid] < target) {
10        left = mid + 1; // 目标在右半部分
11    } else {
12        right = mid - 1; // 目标在左半部分
13    }
14 }
```

哈希搜索 哈希搜索通过将数据映射到一个固定大小的数组（哈希表）中来实现快速查找。它适用于**需要频繁插入、删除和查找**的场景。哈希表的平均时间复杂度为 $O(1)$, 但在最坏情况下可能退化为 $O(n)$ 。另外, 在查找之前有一个建立哈希表的过程, 这个过程是 $O(n)$ 的; 哈希表也有着较大的常数时间和较大的空间开销。我们通常不手写哈希表, 而是使用标准库中的哈希容器, 如 C++ 的 `std::unordered_map` 和 `std::unordered_set`。

```
1 unordered_map<int, int> hash_map; // 创建哈希表
2 hash_map[key] = value; // 插入或更新键值对
3 if (hash_map.find(key) != hash_map.end()) { // 查找键
4     cout << "Found: " << hash_map[key] << endl;
5 }
6 hash_map.erase(key); // 删除键值对
```

快速选择 受到快速排序的启发，出现了一种名为“快速选择”的算法。快速选择是一种用于在无序数组中查找第 k 小（或第 k 大）元素的算法。它基于快速排序的分治思想，通过选择一个“基准”元素，将数组分成两部分，然后递归地在包含第 k 小元素的部分继续搜索。快速选择也是一种分治算法，但它只关注找到第 k 小元素，而不是对整个数组进行排序。

```
1 int quickselect(vector<int>& arr, int left, int right, int k) {
2     if (left == right) return arr[left]; // 只有一个元素
3     int pivot = arr[(left + right) / 2]; // 选择基准
4     int i = left, j = right;
5     while (i <= j) {
6         while (arr[i] < pivot) i++;
7         while (arr[j] > pivot) j--;
8         if (i <= j) {
9             swap(arr[i], arr[j]); // 交换
10            i++;
11            j--;
12        }
13    }
14    if (k <= j) return quickselect(arr, left, j, k); // 在左半部分继续搜索
15    if (k >= i) return quickselect(arr, i, right, k); // 在右半部分继续搜索
16    return arr[k]; // 找到第 k 小元素
17 }
```

当然，如果使用 STL 的 `nth_element` 函数，可以更简单地实现快速选择。

```
1 vector<int> arr = ...; // 输入的数组
2 int k = 2; // 举例：查找第 3 小元素 (k 从 0 开始)
3 nth_element(arr.begin(), arr.begin() + k, arr.end()); // 部分排序
4 int kth_smallest = arr[k]; // 第 3 小元素
```

值得注意的是， k 是从 0 开始计数的。

搜索算法	适用场景	时间复杂度
线性搜索	数据量小且无序	$O(n)$
二分搜索	有序数组	$O(\log n)$
快速选择	查找第 k 小元素	$O(n)$
哈希搜索	频繁插入、删除和查找	$O(1)$

表 25.2: 常见搜索算法的适用场景和时间复杂度比较

如果数据规模比较小或者仅查找一次，则线性搜索更好；如果数据量小且有序，则二分搜索更好；如果数据量大且需要频繁查找，则哈希搜索更好。这是因为，在数据量大的情况下，哈希表的常数时间开销会被其高效的查找性能所抵消。

25.5 复杂数据结构上的问题²

刚刚的搜索和排序算法都是在线性数据结构上进行的。现实中，很多问题需要在更复杂的数据结构上进行搜索和排序，例如树、图等。下面我们将介绍一些常见的复杂数据结构及其上的算法。由于树的遍历已经在《数据结构》章节中介绍过，这里不再赘述；图的遍历和树很像，只需要记得标记已经访问过的节点即可

25.5.1 树和图上的搜索

DFS、BFS 深度优先搜索和广度优先搜索是两个最基本的树图搜索算法。它们的基本思想分别是尽可能深入地搜索树或图的分支（DFS），以及逐层地搜索树或图的节点（BFS）。具体的操作方式可以参考前文的树遍历章节。

深搜和广搜的一个重要区别在于：深搜使用栈（递归调用隐式使用栈），空间复杂度低，但是不能保证找到最短路径；而广搜使用队列，空间复杂度较高，因为需要存储当前层的所有节点，但是广搜是完备的，能够找到最短路径。

迭代加深搜索 迭代加深搜索可以看成是一种结合了深搜和广搜两者优点的搜索算法。它通过多次执行深搜，每次限制搜索的深度，逐渐增加深度限制，直到找到目标节点为止。这样，迭代加深搜索既能保持深搜的低空间复杂度，又能保证找到最短路径。该算法常见于状态空间较大、深度未知的问题（如拼图、路径规划）。

Algorithm 8: 迭代加深搜索算法

Input: 图的邻接表表示 *graph*，起始节点 *start*，目标节点 *goal*，最大深度 *max_depth*

Output: 是否找到目标节点

```
1 for 深度 depth 从 0 到 max_depth do
2   if 深度限制搜索 (graph, start, goal, depth) then
3     return true;
4   // 找到目标节点
5 return false;
6 // 未找到目标节点
```

25.5.2 最短路问题

最短路问题是图论中的一个经典问题，旨在找到图中两个节点之间的最短路径。根据问题的不同，可以分为单源最短路问题和多源最短路问题。单源最短路问题指的是从某一个特定的节点出发，找到该节点到图中所有其他节点的最短路径；多源最短路问题则是找到图中所有节点之间的最短路径。

²本章作者王煜程、臧炫懿。

Dijkstra 算法 Dijkstra 算法是一种用于在带权图中找到单源最短路径的算法。其基本思想是通过贪心策略，逐步扩展已知的最短路径，直到找到所有节点的最短路径。Dijkstra 算法使用一个优先队列来存储待扩展的节点，每次选择距离源节点最近的节点进行扩展，并更新其邻居节点的距离。但是，该算法不能处理负权边。

Algorithm 9: Dijkstra 算法

Input: 图的邻接表表示 *graph*, 节点数 *n*, 源节点 *source*

Output: 源节点到所有节点的最短路径长度 *dist*

```
1 初始化数组 dist, 使得  $dist[i] = \infty$ , 对于源节点 source,  $dist[source] = 0$ ;
2 创建一个优先队列 pq, 将源节点 source 和距离 0 加入队列;
3 while 队列 pq 不为空 do
4     从队列中取出距离最小的节点 u;
5     for 每个邻居节点 v 和边权重  $weight(u, v)$  在 graph[u] 中 do
6         if  $dist[u] + weight(u, v) < dist[v]$  then
7              $dist[v] \leftarrow dist[u] + weight(u, v)$ ;
8             // 松弛操作将节点 v 和更新后的距离加入队列 pq;
9 return dist;
10 // 返回源节点到所有节点的最短距离
```

Bellman-Ford 算法 Bellman-Ford 算法是一种用于在带有负权边的图中找到单源最短路径的算法。其基本思想是通过反复松弛所有边，逐步更新节点的距离，直到找到所有节点的最短路径。Bellman-Ford 算法可以处理负权边，但不能处理负权环路。

Algorithm 10: Bellman-Ford 算法

Input: 图的边列表表示 *edges*, 节点数 *n*, 源节点 *source*

Output: 源节点到所有节点的最短路径长度 *dist*

```
1 初始化数组 dist, 使得  $dist[i] = \infty$ , 对于源节点 source,  $dist[source] = 0$ ;
2 for i 从 1 到 n - 1 do
3     for 每条边 (u, v) 在 edges 中 do
4         if  $dist[u] + weight(u, v) < dist[v]$  then
5              $dist[v] \leftarrow dist[u] + weight(u, v)$ ;
6             // 松弛操作
7 for 每条边 (u, v) 在 edges 中 do
8     if  $dist[u] + weight(u, v) < dist[v]$  then
9         return “图中存在负权环路”;
10    // 检查负权环路
11 return dist;
12 // 返回源节点到所有节点的最短距离
```

负权环路是指在图中存在一条环路，其边的权重之和为负数。负权环路会导致最短路径无

题变得复杂，因为通过不断地绕行负权环路，可以使得路径长度无限减小，从而无法确定一个唯一的最短路径。因此，在使用 Bellman-Ford 算法时，需要检查是否存在负权环路，以确保算法的正确性。而有这个东西的图，在现实中其实并不常见，而且大多数情况下我们并不关心负权环路，或强行约束这类环路只能走一次等。如果有这类约束，可以考虑将图进行变换，消除负权环路后再进行最短路径计算。

那么怎么进行图的变换呢？一种常见的方法是使用 Johnson 算法，它通过添加一个新的节点并连接到所有其他节点，使用 Bellman-Ford 算法计算从新节点到所有节点的最短路径，然后调整原图中的边权重，使得所有边权重变为非负数。这样就可以使用 Dijkstra 算法来计算最短路径。这个算法已经超出了本书的范围，有兴趣的读者可以自行查阅相关资料。

Floyd-Warshall 算法 Floyd-Warshall 算法是一种解决多源最短路问题的算法。该算法的核心思想是动态规划。我们定义一个二维数组 $dist[i][j]$ ，表示从节点 i 到节点 j 的最短路径长度。初始时，如果存在边 (i, j) ，则 $dist[i][j]$ 为该边的权重，否则为无穷大；对于所有节点 i ， $dist[i][i]$ 为 0。然后，我们通过逐步引入中间节点，更新 $dist$ 数组，直到考虑所有节点作为中间节点为止。

Algorithm 11: Floyd-Warshall 算法

Input: 图的邻接矩阵表示 $graph$ ，节点数 n

Output: 所有节点之间的最短路径长度 $dist$

```
1 初始化二维数组  $dist$ ，使得  $dist[i][j] = graph[i][j]$ ，如果  $i = j$  则  $dist[i][j] = 0$ ；
2  for  $k$  从 0 到  $n - 1$  do
3     for  $i$  从 0 到  $n - 1$  do
4         for  $j$  从 0 到  $n - 1$  do
5             if  $dist[i][k] + dist[k][j] < dist[i][j]$  then
6                  $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$ ；
7                 // 更新最短路径长度
8 return  $dist$ ；
```

在上述代码中， $graph$ 是图的邻接矩阵表示，其中 $graph[i][j]$ 表示从节点 i 到节点 j 的边权重，如果不存在边则为无穷大。最终返回的 $dist$ 数组包含了所有节点之间的最短路径长度。

25.5.3 最大流问题

最大流问题是指在一个流网络中，找到从源节点到汇节点的最大流量。常见的最大流算法包括 Ford-Fulkerson 算法、Edmonds-Karp 算法等，这里也仅介绍 Ford-Fulkerson 算法。

为了介绍最大流问题，我们首先要了解一些基本概念：

流网络 流网络是一个有向图，其中每条边都有一个非负的容量，表示该边能够承载的最大流量。流网络中有一个源节点和一个汇节点，源节点是流的起点，汇节点是流的终点。

流 流是指在流网络中从源节点到汇节点的流量分布。流必须满足两个条件：容量约束，即每条边上的流量不能超过该边的容量；流守恒，即除了源节点和汇节点外，每个节点的流入量等于流出量。

最大流 最大流是指在流网络中，从源节点到汇节点的最大可能流量。
也就是说，我们可以把流网络比作一个在许多城市之间运输货物的系统，源节点是货物的起点，汇节点是货物的终点，而边的容量则表示每条运输线路的最大运输能力。最大流问题就是要找到一种运输方案，使得从起点到终点的货物运输量最大化，同时不超过每条运输线路的最大运输能力。

最大流问题有一个非常重要的定理，称为“最小割定理”。该定理指出，在一个流网络中，最大流量等于最小割的容量。最小割是指将流网络划分为两个部分，使得源节点和汇节点分别位于不同的部分，并且割断的边的容量之和最小。这个定理揭示了最大流问题和最小割问题之间的密切关系。该定理的具体证明不在这里介绍，有兴趣的读者可以查阅相关资料。

Ford-Fulkerson 算法就是利用该定理来求解最大流问题的：其通过不断地寻找增广路径来增加流量，直到无法找到增广路径为止（即找到了最小割）。增广路径是指从源节点到汇节点的一条路径，其中每条边都有剩余容量（即容量减去当前流量大于 0）。我们可以使用 DFS 或 BFS 来寻找增广路径。

Algorithm 12: Ford-Fulkerson 算法

Input: 流网络 $G = (V, E)$, 源节点 s , 汇节点 t
Output: 最大流量 max_flow

```
1 初始化所有边的流量为 0;  
2   $max\_flow \leftarrow 0$ ;  
3  while 存在从  $s$  到  $t$  的增广路径  $P$  do  
4      计算增广路径  $P$  上的最小剩余容量  $c$ ;  
5      for 每条边  $(u, v)$  在增广路径  $P$  上 do  
6          增加边  $(u, v)$  的流量  $c$ ;  
7          减少边  $(v, u)$  的流量  $c$ ;  
8          // 反向边  
9       $max\_flow \leftarrow max\_flow + c$ ;  
10 return  $max\_flow$ ;
```

25.6 在线算法简介

上述所有的算法都适用于一口气能拿到所有数据的情况，即“离线算法”。但在现实中，很多情况下数据是流式到达的，我们无法一次性拿到所有数据，这时就需要“在线算法”或“异步算法”。

在线算法 在线算法是一类在数据逐步到达时，能够即时处理并给出结果的算法。与离线算法不同，在线算法不能等待所有数据到达后再进行处理，而是需要在每个数据点到达时做出决

策。在线算法通常需要在有限的时间和空间内处理数据，因此设计在线算法时需要考虑时间复杂度和空间复杂度的权衡。

例题

- 有一堆整数，希望找到其中位数。在以下三种情况下设计算法：
- 你能看到所有整数。
 - 整数是流式到达的，但你依然能存储所有整数（但你计算机的处理速度有限，这说明你无法等待所有整数到达后再处理它们；换言之，要求随时能够回答“当前已到达数据的中位数”）
 - 整数是流式到达的，且你只能存储有限数量的整数（假设你只能存储 k 个整数， k 远小于总整数数量）。

第一种情况非常简单。只需要知道这堆数有多少个，然后快速选择算法（见前文）就能在 $O(n)$ 时间内找到中位数，这个也可以优化（如 BFPRT 算法），但写起来比较麻烦，这里就不赘述了。我们主要考虑的是第二种和第三种情况。

第二种情况是经典的“动态中位数查找”问题，虽然不能算是“异步算法”，但思想已经接近了。这个问题的常见方法是“对顶堆”。这种情况也是常见的情况，因为现代计算机的内存通常足够大，能够存储大量数据，但处理速度有限，无法等待所有数据到达后再处理它们。

Algorithm 13: 对顶堆法

```
Input: 流式到达的整数序列
Output: 当前已到达整数的中位数
1 大顶堆 max_heap，用于存储较小的一半整数；
2 小顶堆 min_heap，用于存储较大的一半整数；
3 for 每个到达的整数 num do
4     if max_heap 为空或  $num \leq max\_heap.top()$  then
5         | 将 num 加入 max_heap；
6     else
7         | 将 num 加入 min_heap；
8     // 平衡两个堆的大小 if  $max\_heap.size() > min\_heap.size() + 1$  then
9         | 将 max_heap 的堆顶元素移动到 min_heap；
10    else if  $min\_heap.size() > max\_heap.size()$  then
11        | 将 min_heap 的堆顶元素移动到 max_heap；
12    // 计算当前中位数 if  $max\_heap.size() > min\_heap.size()$  then
13        | 中位数为  $max\_heap.top()$ ；
14    else
15        | 中位数为  $(max\_heap.top() + min\_heap.top())/2$ ；
```

第三种情况就是真正的“异步算法”了。该情况实际上还有很多种变种。

能多次读取数据流 这种情况常见于硬盘数据处理。这种情况下，我们能够得到精确的中位数，精短算法是**外存选择排序**。这需要 $O(\log n)$ 次遍历整个流。

Algorithm 14: 外存选择排序法

Input: 流式到达的整数序列，内存大小 M

Output: 当前已到达整数的中位数

- 1 将数据流分成若干块，每块大小不超过 M ;
- 2 **for** 每块数据 **do**
- 3 将该块数据读入内存并排序;
- 4 将排序后的块写回外存;
- 5 使用归并排序将所有排序后的块合并成一个有序序列;
- 6 计算中位数并返回;

只能读取一次数据流 这种情况常见于网络数据处理、传感器数据处理等。这种情况下，我们无法得到精确的中位数，只能得到一个近似值。常见的方法有很多，例如蓄水池抽样法、分桶（直方图）法、Greenwald-Khanna 算法等。这里我们介绍分桶法。

Algorithm 15: 分桶法

Input: 流式到达的整数序列，桶的数量 k

Output: 当前已到达整数的近似中位数

- 1 初始化 k 个桶，每个桶记录其范围和计数;
- 2 **for** 每个到达的整数 num **do**
- 3 将 num 放入对应的桶中，并更新该桶的计数;
- 4 计算总计数 $total$;
- 5 计算中位数所在的桶 b ，使得前 $b - 1$ 个桶的计数之和小于等于 $total/2$ 且前 b 个桶的计数之和大于 $total/2$;
- 6 估计中位数为桶 b 的范围内的一个值（例如该范围的中点）;
- 7 **return** 估计的中位数;

由此可见，在线算法通常需要在有限的时间和空间内处理数据，因此设计在线算法时需要考虑时间复杂度和空间复杂度的权衡。此外，在线算法通常只能得到近似解，而无法保证精确解。因此，在设计在线算法时，需要根据具体问题的需求，选择合适的算法和数据结构。

第七部分

工具箱

第二十六章 文字排版:Markdown 和 Typst

本章概要

本章介绍了常用的文本排版工具和语言，包括 Markdown、 $\LaTeX$ 和 Typst。通过学习本章内容，读者将能够掌握基本的文本排版技能，选择合适的排版工具，并能够编写格式良好的文档。

本章前置知识

- 基本的计算机操作技能
- 计算机的联网、获取网络资源（如何使用浏览器、下载文件等）
- 基本的文本编辑技能（如使用记事本、VS Code 等文本编辑器）
- 基本的编程技能（了解代码的基本结构和语法）
- 基本的命令行操作技能（如使用终端等）

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：

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 26.1: 不同排版工具的对比

一般情况下，Markdown 的使用场景是轻量级的文本排版，例如我们平常简单记录事项、技术文档等；而 Markdown 还和网页兼容良好，因此也广泛用于 Blog、论坛等场景。 $\LaTeX$ 的使用场景是较正式的文本排版、且版式极为重要的场景，例如书籍和学术论文、学术报告；其宏包（如 beamer、moderncv）等也让它在学术讲演和简历制作中大放异彩。Typst 较新，目前仍在发展，但其较为易用，排版等级也是论文级的，适用于常规场景，例如课程作业等。

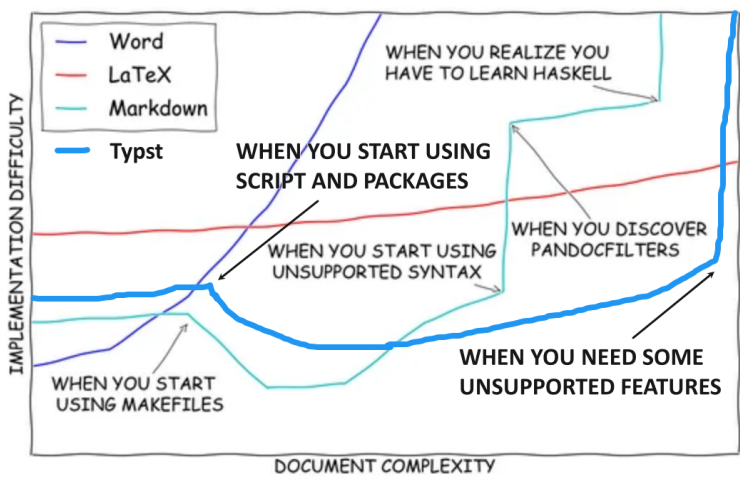


图 26.1: 另一个比较图表 (图源: OrangeX4)

26.1 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都可编写 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。当然，也可以使用一些优秀的 Markdown 专用编辑器，例如 Typora（付费）、Obsidian（免费）等，它们提供了更丰富的功能和更好的用户体验。

26.1.1 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，谨防输入全角标点符号导致 Markdown 无法正确渲染。

说明

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，这是因为不同的 Markdown 渲染器往往有着不同的渲染风格。本人在这里讲述的也是几乎所有渲染器都支持的标记风格。

举一个方言的例子：在 Obsidian 中，有一个语法“CallOut”，可以用来创建带有边框和图标注释块，例如：

```
1 > [!NOTE]
2 > 这是一个注释块。
```

但是该语法几乎只有 Obsidian 支持，诸如 Typora、VS Code 等其他渲染器并不支持该语法。因此，在使用 Markdown 时，建议尽量使用通用的语法，以确保在不同的渲染器中都能正确显示；如要使用方言，则应当确保目标渲染器支持该语法。

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 `<br>` 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。

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：

```
1 # 一级标题
2 ## 二级标题
3 ### 三级标题
```

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：

```
1 * 斜体文本 *
2 ** 粗体文本 **
3 *** 粗斜体文本 ***
4 ~~ 删除线文本 ~~
```

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。

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：

```
1 `行内代码`
2
3 ```
4 代码块
5 ```
```

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。

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：

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

列表

在 Markdown 中，可以使用减号 (-) 来表示无序列表。例如：

```
1 - 列表项 1
2 - 列表项 2
3 - 列表项 3
```

有序列表则使用数字加点的方式表示，例如：

```
1 1. 列表项 1
2 2. 列表项 2
3 3. 列表项 3
```

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：

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

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：

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

在 Markdown 中，可以使用感叹号、方括号和圆括号来表示图片。例如：

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

数学公式

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

```
1 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：

```
1 这是一个块级公式的示例：
2 $$ E=mc^2 $$
```

大多数的 Markdown 编译器都会正确渲染 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 公式，但是不排除少数编译器不支持渲染 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 。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。

26.2 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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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。

Typst 最新潮的一点在于其编译是**增量式**的，而不是 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 的全量编译。简单地说，Typst 只会重新编译被修改的部分，而不是整个文档，这大大提高了编译速度，尤其是在处理大型文档时。但目前大型文档（几百页以上）依然基本上被 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 和 Adobe Indesign 占据主导地位，Typst 仍需努力。

26.2.1 Typst 的安装

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。

说明

官方提供了 `typst` 的命令行程序，可以使用 `Rust` 工具链安装或其他包管理安装。使用 `typst watch main.typ`，之后就可以随意编辑 `main.typ`，在能支持动态加载的 pdf 预览器如 `SumatraPDF` 和 `okular` 中预览 pdf 了。这种预览方式相比于 `tinymist` 预览内存占用小，平时推荐使用 `tinymist` 插件预览，除非长文本内存占用极大。

26.2.2 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。

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：

1

= 一级标题

2

上述文本等价于

3

`#heading(level: 1, [一级标题])`

4

5

== 二级标题

6

上述文本等价于

7

`#heading(level: 2, [二级标题])`

8

9

* 加粗 * 等价于 `#strong[加粗]`

10

_ 强调 _ 等价于 `#emph[强调]`

11

需要注意，没有标记下划线，需要用 `#underline[下划线]` 来进行。

12

`#strike[删除线]`，没有标记删除线

13

14

- 无序列表

15

+ 有序列表

16

/术语：术语列表

17

上述文本等价于

18

`#list.item[无序列表]`

19

`#enum.item[有序列表]`

20

`#terms.item[术语][术语列表]`

21

22

`$x^2/a^2 + y^2/b^2 = 1$` 公式两边不空空格是行内公式

23

`$ \sum_{k=0}^n x^k`

24

`&= 1 + \dots + n \setminus`

25

`&= (n(n+1)) / 2 $`

26

`$ (x+1)/(x) \geq 1 \Rightarrow 1/x \geq 0 $`

27

公式两边空格是行间公式

28

你可能注意到了 `Typst` 公式与 `\LaTeX` 公式有差异。在数学表示上有所形象化，如用 `\geq` 代替了 `\ge`，
`\Leftrightarrow` `\geq` 是 `gt.equiv` 的标记。

29

```
30  ```\py
31  print("Hello, World!")
32  \```
33  上述代码块等价于
34  #raw(lang: "py", block: true, "print("Hello, World!")")
35
36  还有一种特殊的语法糖：
37  #fn(ZZZ)[XXX][YYY] 是 #fn(ZZZ, XXX, YYY) 的语法糖。
```

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：

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

上述代码设置了标题的编号。

show 的本意是进行“显示替换”，也就是把某个内容替换成另一个内容。例如：

```
1  #show " 大陆": " 中国"
```

上述代码可以把整篇文档中的所有“大陆”在编译出的文档都显示成“中国”，例如“哥伦布发现了新大陆”会被编译成“哥伦布发现了新中国”¹。

show 还可以用于样式设计：

```
1  #show heading.where(level: 1): body =>{
2    set align(center)
3    body
4  }
```

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：

```
1  // 存储基本值
2  #let x = 10
3  #let name = "Typst"
4  #let is-active = true
5  The value of x is #x. // 输出: The value of x is 10
6
7  // 存储内容块
```

¹该笑话来自台湾网友，显然不是历史事实：实际上是讽刺台当局的“去大陆化”政策。


```
8 #let warning = [**Warning:** This is important!]
9 #warning // 输出: **Warning:** This is important!
10
11 // 定义函数
12 #let add(a, b) = a + b
13 #let greet(name) = [Hello, #name!]
14 #add(5, 3) // 输出: 8
15 #greet("Alice") // 输出: Hello, Alice!
16
17 // 定义样式
18 #let emph-style = set text(red, weight: "bold")
19 #emph-style
20 This is emphasized text. // 红色加粗
21
22 // 多式综合
23 #let base-style = set text(font: "Helvetica", size: 12pt)
24 #let title-style = base-style.with(size: 16pt, weight: "bold")
25 // 这里的 *.with() 可以扩展或者覆盖原有的样式
26
27 #title-style
28 = This is a title // 使用 Helvetica, 16pt, 加粗
```

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](https://typst.app/docs/), 已经发布的包可以在[Typst Universe](https://typst.app/universe/)上查看, 对于中文问题可以查看[Typst 中文社区导航](https://guide.typst.dev/FAQ.html), 对了还有 Typst 非官方中文交流群 (QQ 群号: 793548390), 欢迎来交流。

第二十七章 $\LaTeX$ ¹

$\LaTeX$ 是一种基于 $T_{\text{E}}X$ 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， $\LaTeX$ 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。

$T_{\text{E}}X$ 实际上是一种语言，在 1978 年由高德纳（Donald E. Knuth）发明，旨在提供一种高质量的排版系统。基本的 $T_{\text{E}}X$ 功能仅有 300 个命令，晦涩难懂，一个简单的符号就需要许多命令实现。后来，人们把这些基本命令封装起来，做成了简写（宏），来实现特殊的目的，于是又出现了 Plain $T_{\text{E}}X$ 、 $\LaTeX$ 和 ConTeXt 等“宏集合”。 $\LaTeX$ 是其中最流行的一个。

而有了语言，就得有编译器/解释器这种东西。而在 $T_{\text{E}}X$ 中，这类东西被叫做“引擎”。常见的引擎包括 pdf $T_{\text{E}}X$ 、X $\LaTeX$ 和 Lua $T_{\text{E}}X$ 等。不同的引擎有着不同的特点和适用场景，例如 pdf $T_{\text{E}}X$ 适合处理简单的英文文档，是早年间的主流引擎；X $\LaTeX$ 最大的卖点在“系统字体即拿即用”、Unicode 原生支持两方面，是汉语相关文档的首选引擎；Lua $T_{\text{E}}X$ 则是 2007 年起的新一代引擎，把内部代码逐渐换成 Lua 脚本，目标是把 $T_{\text{E}}X$ 从硬编码的时代解放出来，功能和 Xe $T_{\text{E}}X$ 相似，但更开放、更可编程，被视为“未来的 $T_{\text{E}}X$ 引擎”。这些引擎配上 $\LaTeX$ 宏集合，就被相应的叫做 pdf $\LaTeX$ 、X $\LaTeX$ 和 Lua $\LaTeX$ 。

为了方便用户使用，开发者们把引擎、宏集合、字体、更新机制乃至其他常用宏包都打包在一起，形成一个“发行版”。常见的发行版有 TeX Live、MikTeX 和 MacTeX 等。TeX Live 是目前最流行的发行版，支持多种操作系统，包括 Windows、Linux 和 macOS，每年一个新版本，一次安装就能下载整个 $\LaTeX$ 全家桶；MikTeX 主要面向 Windows 用户，提供了易于使用的安装程序和更新机制，对于其他宏包是随用随下载；MacTeX 则是专为 macOS 设计的发行版，集成了 macOS 的特性和工具。

LaTeX 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。

我非常感谢张庭瑄同学的帮助，他为本章的内容提供了许多宝贵的建议和指导。

¹本章由张庭瑄和臧炫懿合作完成，其中张庭瑄为主要作者。

27.1 $\text{\LaTeX}$ 发行版的安装和配置

虽然 $\text{\LaTeX}$ 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 $\text{\LaTeX}$ 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，我这里以最通行的 $\text{\TeX}$ Live 为例，介绍其安装过程。其他发行版的安装过程大同小异，读者可以自行参考相关文档进行安装。

27.1.1 Windows 机器

在安装新的 $\text{\TeX}$ Live 之前，笔者建议彻底删除任何旧版的 CTeX 套装²，以防出现各种莫名其妙的错误。然后，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）：

```
1 mkdir C:\temp
2 set TEMP=C:\temp
3 set TMP=C:\temp
```

如无意外，用户可以从最近的 CTAN 源下载 $\text{\TeX}$ Live 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空格或其他非英文特殊字符³。安装完成后，建议将 $\text{\TeX}$ 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 $\text{\LaTeX}$ 命令。我们不建议安装 $\text{\TeX}$ Live 的 GUI 前端，因为它不易于使用。

27.1.2 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。

²CTeX 套装是 2015 年前非常流行的一个国产整合包，但是近年来已经不再维护，且与新版 $\text{\TeX}$ Live 冲突严重，建议卸载。

³新版本的 $\text{\TeX}$ Live 貌似已经支持空格了。但是老教程遍地都是，为了保险起见，依然建议不要使用空格

然后，下载光盘镜像，并进行装载。

```
1 sudo apt install fontconfig gedit
2 sudo mkdir /mnt/texlive
3 sudo mount ./texlive2025.iso /mnt/texlive
4 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：

```
1 sudo umount /mnt/texlive
2 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。

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

其他的发行版虽然略有不同，但是也大同小异；总体上都可以大致分为从 ISO 镜像安装文件和配置相关环境（环境变量、字体）这两步。

27.1.3 $\text{\LaTeX}$ 在 VS Code 的配置

我们这里使用 $\text{\XeLaTeX}$ 作为主要的编译引擎，因为它对中文的支持最好，同时也支持 Unicode，可以直接输入各种特殊符号而无需额外配置。

首先，我们应当下载并安装 VS Code 的 LaTeX Workshop 插件⁴。该插件提供了 $\text{\LaTeX}$ 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：

```
1 "latex-workshop.latex.tools": [
2   {
3     "name": "xelatex",
4     "command": "xelatex",
5     "args": [
6       "-synctex=1",
7       "-interaction=nonstopmode",
8       "-file-line-error",
9       "%DOC%"
10    ]
11  }
12 ],
```

⁴张庭瑄同志说该插件有 bug，但笔者使用并未发现问题，同学们见仁见智了。

```

13  "latex-workshop.latex.recipes": [
14    {
15      "name": "xelatex",
16      "tools": [
17        "xelatex"
18      ]
19    }
20  ],

```

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 $\text{\LaTeX}$ 文档了。之所以使用

我们非常建议关闭 $\text{\LaTeX}$ 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：

```

1  "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 $\text{\LaTeX}$ 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：

```

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

```

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

27.2 初探 $\text{\LaTeX}$

话不多说，先来一个最小运行实例：

```

1  \documentclass{article}
2
3  \title{This is a title}
4  \author{Your Name}
5  \date{\today}
6
7  \begin{document}
8  \maketitle
9  Hello, \LaTeX!
10 \end{document}

```

这个例子展示了一个最简单的 $\text{\LaTeX}$ 文档结构。我们从上到下依次解释各个部分的作用：

- `\documentclass{article}`: 这行代码指定了文档的类型，这里我们选择了 `article` 类型，适用于短篇报告和文章。

- 3-5 行：这些行定义了文档的标题、作者和日期信息。
- `\begin{document}` 和 `\end{document}`：这两行代码标志着文档的开始和结束，所有的正文内容都应当写在这两行代码之间。
- `\maketitle`：这行代码用于生成标题或标题页，根据前面定义的标题、作者和日期信息。
- `Hello, \LaTeX!`：这是文档的正文内容，这里我们简单地输出了一句问候语。

我们用以下命令编译这个文档：

```
1 xelatex example.tex
```

当然如果用的是 VS Code 的 $\text{\LaTeX}$ Workshop 插件并开启“保存时自动编译”功能的话，就不需要手动运行上述命令了。

27.2.1 命令和环境

从上文中，我们抽象出两个概念：一个是“命令”，另一个是“环境”。

命令通常以反斜杠(`\`)开头，后面跟着命令名称和可选的参数，用于执行特定的操作，例如设置标题、插入图片等。

对于带有必选参数的命令而言，当必选参数是 1 个字符或 1 个命令时，可以省略大括号。例如，命令 `\a b` 等价于 `\a{b}`。但是当必选参数是多个字符时，则不能省略大括号，例如 `\a bc` 并不等价于 `\a{bc}`，而是等价于 `\a{b}c`。

除此之外，有的命令还可以带一个星号(`*`)作为修饰符，以改变命令的行为。例如，命令 `\section*` 用于创建一个无编号的章节标题，而 `\section` 则会创建一个带编号的章节标题。星号修饰符通常用于那些需要特殊处理的命令，以提供更多的灵活性和控制。

环境则是由 `\begin{}` 和 `\end{}` 包围的一段代码块，用于定义特定的结构或格式，例如列表、表格等。在上述 `begin` 和 `end` 后的环境名参数必须相同，否则会报错。

这两个是 $\text{\LaTeX}$ 的核心概念，理解它们对于编写 $\text{\LaTeX}$ 文档非常重要，所有的 $\text{\LaTeX}$ 文档都是由命令和环境，以及其中的文本内容组成的。

27.2.2 正确输入符号

在 $\text{\LaTeX}$ 中，大多数字符都可以直接输入，例如字母、数字和大部分标点符号。但是，有一些特殊字符在 $\text{\LaTeX}$ 中有特殊的含义，不能直接输入，否则会导致编译错误，例如 `#` 用来在定义时指定命令参数、`$` 用来表示数学模式的开始和结束、`%` 用来表示注释的开始等。

不能直接输入的符号都需要用一个命令来输入，大多数上述符号对应的命令都是在它的前面加上反斜杠。所以：

反斜杠比较特殊，这个东西被定义为 `\textbackslash` 而不是 `\\`，因为后者在 $\text{\LaTeX}$ 中被定义为换行命令。类似的，`^` 虽然可以用 `\^` 输入，但在文本模式下被定义为重音符号，在一些欧洲语言中用于表示字母的变音⁵，例如 `\^e` 会输出 `ê`。如果想要在文本中输入普通的符号，

⁵这个太多了，不展开讲了。

表 27.1: 使用命令输入符号

输入	<code>\#</code>	<code>\\$</code>	<code>\%</code>	<code>\&</code>	<code>_</code>	<code>\{</code>	<code>\}</code>	<code>\textbackslash</code>
输出	#	\$	%	&	_	{	}	\

可以使用 `\textasciicircum` 命令。

类似的，波浪号 (`\~{}`) 在 $\LaTeX$ 中被定义为不可断行空格，如果想要输入普通的波浪号，可以使用 `\textasciitilde` 命令。这个东西和数学中的 `\sim` 是不同的，因此不能混用。

还有一些标准键盘难以输入的符号，例如英文省略号、破折号等，也有对应的命令。例如，英文省略号可以用 `\dots` 或 `\ldots` 命令⁶输入，破折号可以用 `--` 和 `---` 分别输入短破折号和长破折号。

类似的，双引号和单引号也有特殊的输入方式，分别是 ``` 和 `'` 以及 ``` 和 `'`。在排版是时候不要一对直引号来表示引号，而是要用上述的方式来输入。而中文的引号则可以直接输入，实际上在 Unicode 中英文引号和中文引号是一个码位的，但字体是不同的。

其他特殊符号（如 §）在 $X_{\text{La}}\TeX$ 和 $\text{Lua}\TeX$ 中可以直接输入，因为它们支持 Unicode 字符集。但是在 $\text{pdf}\TeX$ 中，仍然需要使用相应的命令来输入这些符号，例如上述的 `\S` 命令。类似还有 ¶ (`\P`)、• (`\textbullet`) 等。

也可以通过 Unicode 编码来输入一些符号，例如 `\symbol{960}` 可输出 π 。这种输入方式仅适用于 $X_{\text{La}}\TeX$ 和 $\text{Lua}\TeX$ ，因为它们支持 Unicode 字符集。

还有的符号由宏包提供，`pifont`、`manfnt`、`wasysym` 等宏包都提供了大量的符号，可以根据需要进行使用。

27.2.3 空格、换行和分段

空格

字母之间的一个空格或多个空格在输出时都会被视为一个空格。这意味着无论你在源代码中输入多少个空格，最终输出的文档中只会显示一个空格。每一行开头的空格会被忽略掉。

而使用 `Tab` 输入的水平制表符也会被视为一个空格，多个 `Tab` 同样只会被视为一个空格。但不建议在 $\LaTeX$ 文档中使用 `Tab` 来表示空格。

在 $\text{pdf}\TeX$ 外的任何编译方式下使用 `ctex` 来处理中文文档时，中文字符之间的空格会被忽略掉，而汉字和字母、数字之间会自动留出间距。

若确实想输出一个空格，使用 `\` 来输出一个强制的空格，这个空格允许换行。该空格经常用于在由小写字母和句点构成的缩写的后面，例如 `e.g.\` LaTeX 会输出为 *e.g. LaTeX* 。这是因为小写字母的后面的句点被认为是一句话的结束，因此后面的空格会被视为句子间距而不是单词间距，从而导致排版效果不佳。使用强制空格可以避免这种情况。

如果想输出一个不可换行的空格，可以使用 `\~` 来实现，该空格多用于人名。

⁶对于句号前出现的省略号，直接用就行。但这也会导致英文省略号在正文中的前后间距会不对称，因而推荐的解决方案是把这两个东西都用在数学模式中。

面对更严格的需求，例如“空一个汉字的宽度”，可以使用`\hspace{1em}`来实现，其中`1em`表示当前字体大小的宽度。类似的，`\hspace{2em}`表示空两个汉字的宽度，依此类推。

由反斜杠和字母构成的命令后面如果直接跟一个空格的话，该空格会被忽略掉，因此如果想在命令后面输出一个空格的话，可以使用强制空格`\`来实现。例如，`\LaTeX\ is great!`会输出为 *LaTeX is great!*，而不是 *LaTeXis great!*。而更常用的写法是`{\LaTeX} is great!`。

另，*TeX* 原语 `\ignorespaces` 可以用来忽略命令后面的所有空格，直到遇到下一个非空格字符为止，该命令在自定义命令和环境时非常有用。

换行和分段

在 *TeX* 中，单个换行符不会导致输出中的换行。要在输出中插入换行，可以使用两个连续的换行符（即一个空行）来表示一个新的段落。例如：

```
1 This is the first paragraph.  
2  
3 This is the second paragraph.
```

这段代码会输出为两个段落。
而换行则通过命令`\`来实现，例如：

```
1 This is the first line.\  
2 This is the second line.
```

换行和分段是两个不同的概念：换行是在同一段落内换到下一行，而分段则是开始一个新的段落。也就是说，换行的时候并没有分段。换行仅能在段落内部使用，所以不能在段落开头使用换行命令。

还有一个换行命令`\linebreak`，该命令会使得前一行分散对其⁷，并在当前位置强制换行。该命令在少数情况下用于消除连字符。

强制分页

有时我们需要在文档中强制分页，有三种常用的方法可以实现这一点：

- 使用命令`\newpage`：该命令会立即开始一个新页，无论当前页是否已满。但在多栏文档中，该命令只会结束当前栏，而不会结束当前页。
- 使用命令`\clearpage`：该命令总会切到新的页面。
- 使用命令`\cleardoublepage`：该命令会切换到下一页，并确保新页是奇数页（右侧页）。如果当前页是奇数页，则会插入一个空白页以确保新页是奇数页。该命令通常用于双面打印的文档中，以确保章节或部分总是从右侧页开始。

⁷也就是 Word 中的“两端对齐”效果，上一行会被拉伸以填满整行。

百分号和代码注释

在 $\text{\LaTeX}$ 中，百分号 (%) 用于表示注释的开始。任何在百分号后面的内容，直到行尾，都会被视作注释，不会被编译器处理。例如：

```
1 This is some text. % This is a comment
```

这段代码中，This is a comment 是注释内容，不会出现在输出的文档中。

百分号有一个妙用：能够使得 $\text{\LaTeX}$ 忽略百分号之后、下一行最前面的所有空格，这一功能在自定义命令和环境时非常有用。例如：

```
1 Some text here.%
2   More text here.
```

这段代码会输出为 Some text here.More text here.，而不会在 here. 和 More 之间插入空格。

27.2.4 $\text{\LaTeX}$ 文档结构

显然，一个完整的 $\text{\LaTeX}$ 文档通常由以下两个部分组成：导言区和正文区。

导言区

导言区位于 `\begin{document}` 命令之前，用于设置文档的类型、加载宏包和定义自定义命令等。导言区中的内容不会直接出现在输出的文档中，但会影响文档的整体格式和功能。

导言区的第一行永远是 `\documentclass` 命令，用于指定文档的类型和相关选项。例如：

```
1 \documentclass[12pt,a4paper]{article}
```

这行代码指定了文档类型为 article，字体大小为 12pt，纸张大小为 A4。上述中括号内的内容被称作“文档类选项”，用于定制文档的外观和行为。

在导言区中，我们还可以使用 `\usepackage` 命令来加载宏包，以扩展 $\text{\LaTeX}$ 的功能。例如：

```
1 \usepackage{graphicx} % 用于插入图片
2 \usepackage{amsmath}  % 用于高级数学排版
```

也可以在调用宏包时传递选项，例如：

```
1 \usepackage[utf8]{inputenc} % 设置输入文件的编码为 UTF-8
```

宏包名称也可以一口气写多个，多个宏包之间用逗号分隔，例如：

```
1 \usepackage{graphicx,amsmath,hyperref}
```

导言区不能出现任何正文内容，否则会导致编译错误。

正文区

正文区位于`\begin{document}`和`\end{document}`命令之间，包含了文档的实际内容，如文本、图片、表格等。正文区中的内容会被编译器处理，并出现在输出的文档中。

LaTeX 标准文档类

LaTeX 提供了几种标准的文档类，适用于不同类型的文档，最常用的是 `article`、`report` 和 `book` 三种文档类，分别用于短篇、中篇和长篇文档的编写。其余的文档类往往是基于上述标准文档类进行扩展和定制的。

27.2.5 标题、标题页

标题由命令`\title`、`\author`和`\date`定义，然后通过命令`\maketitle`生成。标题页通常包含文档的标题、作者和日期等信息。例如：

```
1 \title{My First \LaTeX Document}
2 \author{Alice \and Bob}
3 \date{\today}
4
5 \begin{document}
6 \maketitle
7 \end{document}
```

这段代码会生成一个标题页，显示文档的标题、作者和当前日期。多个作者可以使用`\and`命令分隔。

上述中的`\date`命令可以省略，如果省略的话，*LaTeX* 会默认使用当前日期作为文档的日期。如果不想显示日期，可以将`\date`命令设置为空，例如`\date{}`。`\today`命令用于插入当前日期，也可以在`\date`命令中使用手写的日期，例如`\date{January 1, 2024}`。

在 *LaTeX* 默认的设置中，`article` 文档类的标题不单独占 1 页。而 `report` 和 `book` 文档类的标题则会单独占 1 页。如果想让 `article` 文档类的标题单独占 1 页，应这样做：

```
1 \documentclass[titlepage]{article}
```

类似的，`report` 和 `book` 文档类如果不想让标题单独占 1 页，可以这样做：

```
1 \documentclass[notitlepage]{report}
```

而标准文档类也提供了 `titlepage` 环境，用于创建自定义的标题页。例如：

```
1 \begin{titlepage}
2   \mbox{}\vfil % 占位符和垂直填充
3   \begin{center}
4     {\Huge My Custom Title Page}\vspace{2em}
5     {\Large Author Name}\vspace{1em}
```

```
6 {\large \today}
7 \end{center}
8 \end{titlepage}
```

上述代码会创建一个自定义的标题页，使用了 $\text{\mbox{}}$ 命令作为占位符，并使用 $\text{\vfil}$ 命令实现垂直居中对齐。标题、作者和日期分别使用不同的字体大小，并通过 $\text{\[length]}$ 命令调整行间距。上述代码是正文区的一部分，因此应放在 $\text{\begin{document}}$ 和 $\text{\end{document}}$ 之间，且使用了自定义的标题页就不应该再使用 $\text{\maketitle}$ 命令了，否则会导致重复的标题。 titlepage 的另一个特性是会把当前页的页码设置为 0，下一页的页码从 1 开始。

27.2.6 摘要

摘要是通过 abstract 环境⁸来创建的。例如：

```
1 \begin{abstract}
2 This is a brief summary of the document.
3 \end{abstract}
```

一般的，摘要环境应放在标题页之后、目录之前的位置。如果文档没有标题页，则应放在正文的开头部分。

$\text{\LaTeX}$ 标准文档类的摘要标题是“Abstract”，如果想要更改摘要标题，可以使用 $\text{\renewcommand}$ 命令重新定义 $\text{\abstractname}$ 命令，例如：

```
1 \renewcommand{\abstractname}{摘要}
```

而如果使用了 ctex 中文文档类或调用了 ctex 宏包则无需这样做。

27.2.7 章节、附录、目录

章节层次

在 $\text{\LaTeX}$ 中，章节和附录是通过特定的命令来创建和管理的，而目录则是通过命令自动生成的。

上述表格列出了 $\text{\LaTeX}$ 中常用的章节命令及其层次结构。需要注意的是，不同的文档类支持的章节层次可能有所不同。例如， article 文档类不支持 $\text{\chapter}$ 命令，而 book 和 report 文档类则支持该命令；在 article 中 $\text{\subsubsection}$ 有编号，而在 book 和 report 中则没有编号。

在默认情况下，所有带编号的章节命令都会自动生成编号，并在目录中显示相应的条目。如果不想要编号，可以在命令后面加上星号（*），例如 $\text{\section*{Introduction}}$ 。这种无编号的章节不会出现在目录中，除非手动添加。手动添加的方式是：

⁸该环境在 book 文档类中不可用。

表 27.2: 章节命令和层次

层次	命令	说明	可选编号
最高层次	<code>\part</code>	用于划分文档的主要部分	是
次高层次	<code>\chapter</code>	用于划分章节	是
中间层次	<code>\section</code>	用于划分节	是
次低层次	<code>\subsection</code>	用于划分小节	是
最低层次	<code>\subsubsection</code>	用于划分子小节	是
段落	<code>\paragraph</code>	用于划分段落	否
子段落	<code>\subparagraph</code>	用于划分子段落	否

```
1 \section*{Introduction}
2 \addcontentsline{toc}{section}{Introduction} % 手动添加到目录
```

目录

目录则必须基于现有的章节命令自动生成：

```
1 \documentclass{article}
2 \begin{document}
3 \tableofcontents % 生成目录
4
5 \section{some}
6 \section*{other}
7 \section{some some}
8 \end{document}
```

上述代码编译两次得到目录内容仅包括“some”和“some some”，而不包括“other”。

在book类中包括一种特殊的分割方式：`\frontmatter`、`\mainmatter`和`\backmatter`命令。这些命令用于划分文档的不同部分，并影响页码的格式和章节编号方式。

- `\frontmatter`：用于文档的前言部分，通常包括标题页、摘要和目录等。在这一部分中，页码通常使用罗马数字 (i, ii, iii, ...)，章节编号通常不显示。
- `\mainmatter`：用于文档的主体部分，包含主要内容。在这一部分中，页码通常使用阿拉伯数字 (1, 2, 3, ...)，章节编号会正常显示。
- `\backmatter`：用于文档的附录和参考文献等。在这一部分中，页码继续使用阿拉伯数字，但章节编号通常不显示。

附录

附录通过`\appendix`命令来创建，该命令会将后续的章节编号改为字母编号 (A, B, C, ...)。例如：

```
1 \appendix
2 \section{First Appendix}
3 \section{Second Appendix}
```

上述代码会生成两个附录，编号分别为“Appendix A”和“Appendix B”。

27.2.8 标准文档类的选项

标准文档类有许多可选的选项，用于定制文档的外观和行为。以下是一些常用的选项：

纸张大小 用于设置文档的纸张大小，最常用的三种是a4paper，a5paper和b5paper，分别对应 A4、A5 和 B5 纸张大小，默认是letterpaper（美国信纸大小），但如果在 TexLive 安装时把“缺省纸张大小”设置为 A4 的话，则默认是 A4 纸张大小。

纸张方向 默认是纵向（portrait），可以使用landscape选项将纸张设置为横向（长边为宽）。

字体大小 用于设置文档的基本字体大小，常用的有10pt、11pt和12pt，默认是10pt。

双面打印 用于设置文档为双面打印格式，默认是单面打印。使用twoside选项启用双面打印，使用oneside选项启用单面打印。book 是默认双面打印的，而 article 和 report 默认单面打印。

标题页 用于控制标题页的显示方式。使用titlepage选项使标题单独占一页，使用notitlepage选项使标题与正文在同一页显示。article 默认不单独占页，而 report 和 book 默认单独占页。

章标题位置 仅适用于 report 和 book 文档类。使用openright选项使章节总是从右侧页开始，使用openany选项允许章节从任意页开始。默认是openright。

单双栏排版 用于设置文档为单栏或双栏排版。使用twocolumn选项启用双栏排版，使用onecolumn选项启用单栏排版。默认是单栏排版。

我们举个例子：

```
1 \documentclass[a4paper,12pt,twoside,titlepage]{report}
```

上述代码指定了文档类型为 report，纸张大小为 A4，字体大小为 12pt，启用双面打印，并使标题单独占一页。

27.2.9 文档类

在 $\text{\LaTeX}$ 中，内容和格式是分离的，文档类决定了文档的整体结构和格式，而内容则由用户编写。因此，选择合适的文档类对于创建符合需求的文档非常重要。

除了标准的文档类（article、report 和 book）之外， $\text{\LaTeX}$ 还有许多其他的文档类，适用于不同类型的文档编写。对于中文文档，最惯用的三个文档类是 ctexart、ctexrep 和 ctexbook，分别对应 article、report 和 book 文档类。这些文档类预设了中文支持和常用的中文排版设置，极大地方便了中文文档的编写。

27.2.10 字体、字号和行距

行距

行距极其简单，仅用一句话即可说明：在导言区使用`\linespread{factor}`命令来设置行距，其中`factor`是一个乘数，表示行距相对于默认行距的倍数。例如，`\linespread{1.5}`会将行距设置为 1.5 倍，适用于学术论文等需要较大行距的文档。

字体

在 *L^AT_EX* 中，有三个最基本的字体族：衬线字体（Serif）、无衬线字体（Sans Serif）和等宽字体（Monospaced）。默认情况下，正文使用衬线字体。对于这几个字体族，可以使用以下命令进行切换：

- 衬线字体：`\textrm{文本}` 或 `{\rmfamily }`
- 无衬线字体：`\textsf{文本}` 或 `{\sffamily }`
- 等宽字体：`\texttt{文本}` 或 `{\ttfamily }`

在 *L^AT_EX* 中，对同一个字体族内加粗、倾斜，本质上是切换字体族内的不同字体变体。加粗和倾斜对应着字体的两个属性：字重和字形。字重表示字体的粗细程度，常见的有常规（Regular）、粗体（Bold）等；字形表示字体的样式，常见的有直立（Upright）、斜体（Italic）和倾斜体（Oblique）等。一般情况下，字重和字形之间的切换是独立的，可以任意组合，但如果不存在某个组合的字体变体，则会退化为最接近的可用变体。

- 中等字重：`\textmd{文本}` 或 `{\mdseries }`
- 粗体字重：`\textbf{文本}` 或 `{\bfseries }`
- 直立字形：`\textup{文本}` 或 `{\upshape }`
- 斜体字形：`\textit{文本}` 或 `{\itshape }`
- 倾斜体字形：`\textsl{文本}` 或 `{\slshape }`
- 小型大写字形：`\textsc{文本}` 或 `{\scshape }`

要叠加字体属性，可以将多个命令嵌套使用，例如：

```
1 \textbf{\textit{Bold Italic Text}}
2 {\bfseries \itshape Bold Italic Text}
```

以上两种写法都会输出加粗斜体文本。

L^AT_EX 还提供了“一键恢复正常”的命令：`\textnormal{文本}` 或 `{\normalfont }`，用于将文本恢复为默认字体样式。

提示

在使用`{\itshape }`时，可能会导致斜体文本和后续的正常文本之间出现重合的问题。这是因为`{\itshape }`命令会影响后续文本的间距，导致斜体文本和正常文本之间的间距不正确。为了解决这个问题，可以在斜体文本后面添加一个倾斜校正命令`\/`，以确保斜体文本和后续文本之间的间距正确。

而在实际操作中,我们推荐使用具有实际意义的命令来设置字体样式,例如`\emph{文本}`用于强调文本,而不是直接使用字体属性命令。如果我们希望修改上述`\emph`命令的行为,可以通过重新定义该命令来实现,而不是直接使用字体属性命令。

另一种手段是自己写一个自定义命令,例如:

```
1 \newcommand{\important}[1]{\textbf{#1}}
```

上述代码定义了一个名为`\important`的新命令,用于加粗文本。使用时,只需调用该命令即可,例如`\important{This is important text.}`。如果之后想要修改该命令的行为,例如改为斜体,只需修改命令定义即可,而不需要在文档中逐一修改所有使用该命令的地方,提高了文档的可维护性。

如希望改变整个文档的默认字体,可以这样做:

```
1 \usepackage{fontspec} % 仅适用于 XeLaTeX 和 LuaLaTeX
2 \setmainfont{Times New Roman} % 设置正文字体
3 \setsansfont{Arial} % 设置无衬线字体
4 \setmonofont{Courier New} % 设置等宽字体
```

但 macOS 中使用这类命令可能遇到一些问题,这实际上是操作系统层面的问题,你要试着找到 macOS 中字体的正确名称。

另一方面,无衬线或等宽字体可能看起来会比衬线字体更大。这需要通过调整字号来弥补:

```
1 \setmainfont{TeX Gyre Termes} % 衬线字体
2 \setsansfont{TeX Gyre Heros}[Scale=MatchLowercase] % 无衬线字体
3 \setmonofont{TeX Gyre Cursor}[Scale=MatchLowercase] % 等宽字体
```

字号

字号就是字体的大小。在 $\text{\LaTeX}$ 中,字号可以通过一组预定义的命令来设置,这些命令分别对应不同的字号级别。常用的字号命令如下:

- `\tiny`: 极小字号,是本文支持的最小字号。
- `\scriptsize`: 上下标的字号,常用于脚注和公式的上下标。
- `\footnotesize`: 脚注字号,常用于脚注
- `\small`: 小字号,常用于次要内容。
- `\normalsize`: 正常字号,默认字号。
- `\large`: 大字号
- `\Large`: 更大字号
- `\LARGE`: 更更大字号
- `\huge`: 巨大字号
- `\Huge`: 更巨大字号,是本文支持的最大字号。

即使我们在导言区设置了文档的基本字体大小（例如 10pt、11pt 或 12pt），上述字号命令仍然会根据该基本字体大小进行相应的调整，不会出现 `normalsize` 比 `large` 还大的情况。

如不想用这些封装好的字号命令，应使用 `\fontsize{size}{skip}\selectfont` 命令来设置自定义字号，其中 `size` 是字体大小，`skip` 是行距，推荐设置为字号的 1.2 倍。例如：

```
1 {\fontsize{14pt}{16pt}\selectfont ABC}
```

倘若使用了 `ctex` 宏包或文档类，则可以使用更方便的命令 `\zihao{size}` 来设置字号，其中 `size` 是字号级别，取值范围为 -8 到 8。例如：

```
1 {\zihao{4} ABC} % 四号字
2 {\zihao{-4} ABC} % 小四号字
```

27.2.11 特殊文字效果

在 *LaTeX* 中，允许使用颜色、下划线、删除线等特殊文字效果来增强文档的视觉效果。这些效果通常通过加载相应的宏包来实现。

颜色

要在 *LaTeX* 文档中使用颜色，首先需要加载 `xcolor` 宏包：

```
1 \usepackage{xcolor}
2
3 \textcolor{red}{This text is red.} % 红色文本
```

上述代码会把 “This text is red.” 显示为红色。

`xcolor` 宏包定义了 `green`、`blue` 等多种预定义颜色，基本上常见的颜色都有对应的名称。也可以反色、混色等，具体用法请参考 `xcolor` 宏包的文档。

下划线等强调效果

在 *LaTeX* 中，可以使用 `ulem` 宏包来实现下划线、删除线等强调效果。首先需要加载该宏包：

```
1 \usepackage{ulem}
2
3 \uline{This text has an underline.} % 下划线
4 \sout{This text has a strikethrough.} % 删除线
5 \uuline{This text has a double underline.} % 双下划线
6 \uwave{This text has a wavy underline.} % 波浪下划线
7 \xout{This text is crossed out.} % 更乱的删除线
8 \dashuline{This text has a dashed underline.} % 虚线下划线
9 \dotuline{This text has a dotted underline.} % 点状下划线
```

上述代码展示了如何使用`ulem`宏包提供的各种强调效果。

但是，使用该宏包会导致`\emph`命令的行为发生变化，默认情况下该命令会将文本设置为斜体，但加载`ulem`宏包后，该命令会将文本设置为下划线。如果不想改变`\emph`命令的行为，可以在加载`ulem`宏包时使用`normalem`选项：

```
1 \usepackage[normalem]{ulem}
```

汉字下边加的点较着重号，该符合则需要加载`xeCJKfntef`宏包（仅适用于 XeLaTeX 和 LuaLaTeX ）：

```
1 \usepackage{xeCJKfntef}
2
3 \CJKunderline{这是下划线} % 汉字下划线
4 \CJKunderdot{这是着重号} % 汉字下点
```

其余命令类似于`ulem`宏包，只不过是要把命令名前加上前缀，且简写 `u` 应该展开为 `under`。该宏包还提供了一个可以在字和字之间断开的下划线用法：

```
1 \CJKunderwave-{我}\CJKunderline-{是}\CJKunderwave-{下划线} % 可断开的下划线
```

也可以通过在命令名后面加上星号来实现不可断开的下划线，例如`\CJKunderline*{文本}`，该下划线不会忽略标点。

27.2.12 文内交叉引用

在 $\LaTeX$ 中，文内交叉引用是通过`\label{key}`和`\ref{key}`命令来实现的。首先，在需要引用的位置使用`\label{key}`命令为该位置设置一个标签（`key`），然后在需要引用该位置的地方使用`\ref{key}`命令来引用该标签。例如：

```
1 \section{Introduction}\label{sec:intro}
2
3 As discussed in Section~\ref{sec:intro}, ...
```

上述代码中，`\label{sec:intro}`为“Introduction”章节设置了一个标签，随后通过`\ref{sec:intro}`引用该章节。类似的，还可以用`\pageref{key}`命令来引用标签所在的页码，用`\nameref{key}`命令来引用标签的名称⁹。

同样的，为了正确编译包含交叉引用的文档，通常需要编译两次，以确保所有引用都能正确解析。

⁹需要加载 `nameref` 宏包。

27.2.13 引用超链接

在 *LaTeX* 中，可以使用 `hyperref` 宏包来创建文档中的超链接。首先需要在导言区加载该宏包：

```
1 \usepackage{hyperref}
```

加载该宏包后，文档中的交叉引用（如 `\ref` 和 `\pageref`）会自动转换为超链接，点击这些链接可以跳转到相应的位置。

在该宏包的默认设置下，超链接的颜色为红色，并且带有边框。如果希望自定义超链接的颜色和样式，可以使用以下选项：

```
1 \hypersetup{hidelinks, colorlinks=true, linkcolor=blue, citecolor=green,  
↪ urlcolor=cyan}
```

上述代码中，`hidelinks` 选项用于隐藏超链接的边框，`colorlinks=true` 选项启用彩色链接，`linkcolor`、`citecolor` 和 `urlcolor` 选项分别设置普通链接、引用链接和 URL 链接的颜色。其他比较常用的选项还有 `pdftitle`、`pdfauthor` 等，用于设置 PDF 文档的元数据，但都不算很常用。

为了引用超链接（网址），需要使用 `\url{网址}` 或 `\href{网址}{显示文本}` 命令。例如：

```
1 \url{https://www.latex-project.org/} % 直接显示网址  
2  
3 \href{https://www.latex-project.org/}{LaTeX Project} % 显示自定义文本
```

上述代码会创建两个超链接，第一个显示完整的网址，第二个显示自定义的文本“LaTeX Project”。

该宏包必须在导言区的最后加载，以确保不出现兼容性问题。

27.2.14 参考文献

参考文献的管理和排版在 *LaTeX* 中通常通过 `BibTeX`、`BibLaTeX` 或 `natbib` 等宏包来实现。这里我们介绍使用 `BibLaTeX` 宏包来管理参考文献的方法，也是笔者比较习惯的方式。

为了管理参考文献，首先我们要写一个 `.bib` 文件，该文件包含了所有参考文献的条目。每个条目都有一个唯一的引用键（`cite key`），用于在文档中引用该条目。例如，下面是一个简单的 `.bib` 文件内容：

```
1 @book{lamport1994latex,  
2   title={LaTeX: A Document Preparation System},  
3   author={Lamport, Leslie},  
4   year={1994},  
5   publisher={Addison-Wesley}  
6 }
```

上述代码定义了一个书籍类型的参考文献条目，引用键为`lamport1994latex`。
在文档中引用参考文献时，可以使用`\cite{cite key}`命令。例如：

```
1 \documentclass{article}
2 \usepackage[backend=biber,style=numeric]{biblatex}
3 \addbibresource{references.bib} % 引入.bib 文件
4
5 \begin{document}
6 This is a reference to Lamport's book \cite{lamport1994latex}.
7
8 % 下列命令常常放在文档末尾
9 \printbibliography % 打印参考文献列表
10 \end{document}
```

编译包含参考文献的文档时，需要按照以下顺序进行编译：

```
1 xelatex mydocument.tex
2 biber mydocument
3 xelatex mydocument.tex
4 xelatex mydocument.tex
```

`biblatex` 的参考文献格式通过宏包选项的`style`参数来设置，常用的格式有 `numeric`（数字编号）、`authoryear`（作者-年份）等。
也可以分别制定参考文献列表的格式和引用的格式，分别使用`citestyle`和`bibstyle`参数。例如：

```
1 \usepackage[backend=biber,citestyle=authoryear,bibstyle=numeric]{biblatex}
```

27.3 排版中文文档

排版中文文档应使用 `ctex` 宏集。该宏集包含了 `ctex` 文档类和 `ctex` 宏包两部分内容。前者用于创建中文文档，后者则可以在任何文档类中使用以支持中文排版。

27.3.1 ctex 文档类

`ctex` 文档类包括 `ctexart`、`ctexrep` 和 `ctexbook`，分别对应 `article`、`report` 和 `book` 文档类。使用这些文档类可以方便地创建中文文档，而无需手动配置中文支持。例如：

```
1 \documentclass{ctexart}
2 \begin{document}
3 你好，世界！
4 \end{document}
```

上述代码创建了一个简单的中文文档，使用了 `ctexart` 文档类。
这几个文档类有着一些特定的选项，用于定制中文文档的外观和行为。例如：

字体设置 可以通过fontset选项来指定中文字体集,例如fontset=windows、fontset=mac等,分别对应 Windows 和 Mac 系统的默认中文字体。

默认字号 可以通过zihao选项来设置默认字号,例如zihao=4表示四号字, zihao=-4表示小四号字。

标点样式 可以通过punctstyle选项来设置中文标点的样式,例如punctstyle=kaiming表示使用开明体的标点样式。

标点样式包括 quanjiao (全角标点)、kaiming (开明体标点)、banjiao (半角标点)、CCT (标点符号宽度略小于一个汉字宽度) 和 plain (不做任何处理)。默认是 quanjiao。

27.3.2 ctex 宏包

ctex 宏包可以在任何文档类中使用, 以支持中文排版。例如:

```
1 \documentclass{article}
2 \usepackage{ctex}
3 \begin{document}
4 你好, 世界!
5 \end{document}
```

上面的代码与使用 ctex 文档类的效果几乎相同, 会开启中文排版方案:

- 默认字号为五号字;
- 行距变为标准文档类默认行距的 1.3 倍;
- 汉化文档的标题名称, 例如摘要、目录等;
- 设置中文标点样式为全角标点;
- 章节标题后的第一段开启首行缩进。

但不会把标题格式等改为中文文档类的格式。

不使用上述功能的话, 可以在调用 ctex 宏包时传递相应的选项来禁用这些功能。例如:

```
1 \usepackage[scheme=plain]{ctex}
```

上述代码禁用了 ctex 宏包的中文排版方案, 恢复为标准文档类的默认设置。这种场景仅仅适用于输入少量汉字的英文文档。

如果希望使用中文文档类的标题格式, 则应:

```
1 \usepackage[heading=true]{ctex}
```

但是这样不能自由的设置章节标题格式, 因此更推荐使用 ctex 文档类。

27.3.3 设置标题样式

ctex 宏集提供了多种预定义的标题样式, 可以通过\ctexset命令来设置。例如:

```

1 \ctexset{section={
2   format+=\bfseries, % 加粗章节标题
3   name={第, 章}, % 章节名称格式
4   number=\chinese{section}, % 章节编号格式为中文数字
5 }}
```

上述代码将章节标题设置为加粗，章节名称格式为“第 X 章”，章节编号格式为中文数字。类似的，可以设置节、小节等标题样式。

ctex 中文文档类把标题分成前后两部分：名称和标题，例如“第 1 章绪论”中，“第 1 章”是名称，“绪论”是标题。这些样式整体上由 format 设置，详见 ctex 文档。

27.3.4 中文字体

在中文文档类中，我们可以选择不同的中文字体来排版文档。ctex 宏集预定义了一些常用的中文字体库，例如：

- windows：适用于 Windows 系统，使用微软雅黑、中易宋体等字体。
- mac：适用于 Mac 系统，使用苹方、华文细黑等字体。
- ubuntu：适用于思源黑体、思源宋体、文鼎楷体等字体。
- adobe：适用于 Adobe 系统，使用 Adobe 的中文字体。字体需要下载。
- fandol：使用 Fandol 字体库，适用于跨平台的中文排版，但相当缺字。
- founder：使用方正字体库，适用于需要方正字体的文档。但字体需要下载，且部分非免费商用。
- none：不设置中文字体，使用系统默认字体。

可以通过在导言区使用\setCJKmainfont、\setCJKsansfont和\setCJKmonofont命令来分别设置中文的衬线字体、无衬线字体和等宽字体。例如：

```

1 \setCJKmainfont{SimSun} % 设置中文衬线字体为宋体
2 \setCJKsansfont{SimHei} % 设置中文无衬线字体为黑体
3 \setCJKmonofont{FangSong} % 设置中文等宽字体为仿宋
```

需要注意的是，如果使用上述命令设置字体，则需要确保所指定的字体已经安装在系统中，否则会导致编译错误；且需要设置宏集中的字体库为 none，否则会得到一条警告信息。

```

1 \usepackage[fontset=none]{ctex}
```

也可以使用更多中文字体：

```

1 \newCJKfontfamily\CJKHeavy{Source Han Serif SC Heavy} % 定义一个新的中文字体命令
2 {\CJKHeavy 这是使用自定义字体的文本。}
```

第二十八章 L^AT_EX 进阶

本章将介绍一些更高级的 L^AT_EX 技巧和概念，帮助你更好地利用 L^AT_EX 的强大功能来创建专业的文档，例如数学公式、插入图表、自定义命令和环境等内容。

28.1 利用 L^AT_EX 排版数学公式¹

在 L^AT_EX 中，数学公式都需要进入一个特殊的模式：数学模式 (math mode)。有两种主要的数学模式：行内数学模式和行间数学模式。

为了更好地排版数学公式，建议加载 amsmath 宏包，它提供了许多增强的数学排版功能和环境。可以在导言区添加以下代码来加载该宏包：

```
1 \usepackage{amsmath}
```

行内公式 在 L^AT_EX 中，行内数学模式使用美元符号 (\$) 包围数学内容，例如：\$E=mc^2\$。这将在文本中显示为 $E = mc^2$ 。另一种方式是使用反斜杠加括号：\ (E=mc^2\)。

\ensuremath{} 可以确保参数始终处于数学模式，无论该命令在文本模式还是数学模式中使用。例如，定义一个命令来表示向量：

```
1 \newcommand{\vect}[1]{\ensuremath{\mathbf{#1}}}
```

这样，无论在文本中还是数学公式中使用 \vect{v}，都能正确显示为 **v**。但是，普通公式非常不建议用这个，仅建议在定义命令时使用。

行间公式 行间公式则是较大的、被单独展示的数学表达式。可以使用反斜杠加中括号：\[E=mc^2\]，这样会将公式居中显示在单独的一行上：

$$E = mc^2$$

上述方法不会带行号，如果需要行号，可以使用 amsmath 宏包提供的 equation 环境：

¹本章作者张庭瑄。

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

在数学模式中输入的空格会被忽略，间距由 $\text{\TeX}$ 自动控制，也就是说 $\$a + b\$$ 和 $\$a+b\$$ 效果是一样的，都会显示为 $a + b$ 。

数学模式中禁用分段、汉字和中文标点。在行间公式中不允许换行。

什么时候使用数学模式 在数学模式中， $\text{\LaTeX}$ 会对输入的内容进行特殊处理，以确保数学符号和表达式的正确显示。因此，即使是最简单的数学表达式都要放在数学模式中，例如变量、运算符和函数等。例如，变量 x 和 y 应该写成 $\$x\$$ 和 $\$y\$$ ，而不是直接写成 x 和 y ； $1+1=2$ 应该写成 $\$1+1=2\$$ 。

由于数学符号之间的间距与文本中产生的间距有所不同、数学和文本的断行也不同，因此在罗列许多变量的时候要把每一个变量单独放在数学模式中，例如： $\$a\$$ ， $\$b\$$ ， $\$c\$$ ，而不是 a ， b ， c 。另，如使用半角逗号分隔变量，则应该加空格。例如： $\$a\$$ ， $\$b\$$ ， $\$c\$$ ，而不是 $\$a\$,\$b\$,\$c\$$ 。全角的逗号则不需要加空格。

28.1.1 普通的数学符号

基本数学符号 在数学模式中，可以使用各种简单的数学符号。希腊字母使用反斜杠加字母的英文名称表示，第一个字母大写表示大写希腊字母，第一个字母小写表示小写希腊字母。例如： $\backslash alpha$ 显示为 α ， $\backslash beta$ 显示为 β ， $\backslash Gamma$ 显示为 Γ ， $\backslash Delta$ 显示为 Δ 。部分大写字母没有被定义，例如大写的 A 和 B 没有对应的命令，这两个分别是大写的 α 和 β ，我们完全可以用普通的字母 A 和 B 来表示它们。

也有小写希腊字母的变体，例如： $\backslash epsilon$ 显示为 ϵ ， $\backslash varepsilon$ 显示为 ε 。

上下标 上下标分别使用 $\wedge$ 和 $\_$ ，例如： $\$x^2\$$ 显示为 x^2 ， $\$a_i\$$ 显示为 a_i ；一个数学符号可以同时带一个上标和一个下标，例如： $\$x_i^2\$$ 显示为 x_i^2 ，反过来写成 $\$x^2_i\$$ 也是一样。如果上下标包含多个字符，则需要使用大括号将它们括起来，例如： $\$x^{n+1}\$$ 显示为 x^{n+1} ， $\$a_{i,j}\$$ 显示为 $a_{i,j}$ 。

一个数学符号后面最多只能带一个上标或下标，如果需要多个上标或下标，可以使用大括号将它们括起来，例如： $\$x^{x^x}\$$ 显示为 x^{x^x} 。

要是想写前标，可以这么写： $\$^nP_r\$$ 显示为 ${}^nP_r$ 。为了避免可能出现的间距问题，建议将数学符号连同前后上下标一起放进大括号中，例如： $\$\{^nP_r\}\$$ 显示为 ${}^nP_r$ 。更好的实践是引入 `mathtools` 宏包，然后使用 `\prescript` 命令，例如： $\backslash prescript\{n\}\{P_r\}$ 。

撇号 撇号可以直接利用 ASCII 撇号 (') 输入，例如： $\$f'(x)\$$ 显示为 $f'(x)$ ， $\$x''\$$ 显示为 x'' 。如果需要多个撇号，可以连续输入多个撇号，例如： $\$f'''(x)\$$ 显示为 $f'''(x)$ 。

角度符号 角度符号可以使用 $\backslash circ$ 输入，例如： $\$90^\circ\$$ 显示为 90° 。

分式 分数和分式使用 `\frac{分子}{分母}` 输入, 例如: `$\frac{a+b}{c+d}$` 显示为 $\frac{a+b}{c+d}$ 。简单的行内公式也可以用斜杠表示分式, 例如: `$a/b$` 显示为 a/b , 但要注意此种方式的表示不要产生歧义, 要在适当的位置加上括号, 例如: `$(a+b)/(c+d)$` 。

二项式系数 二项式系数可以使用 `\binom{n}{k}` 输入, 例如: `$\binom{n}{k}$` 显示为 $\binom{n}{k}$ 。

根号 根号使用 `\sqrt{被开方数}` 输入, 例如: `$\sqrt{x+1}$` 显示为 $\sqrt{x+1}$ 。如果需要指定根指数, 可以使用 `\sqrt[指数]{被开方数}`, 例如: `$\sqrt[3]{x}$` 显示为 $\sqrt[3]{x}$ 。

上下划线 上划线和下划线分别使用 `\overline{内容}` 和 `\underline{内容}` 输入, 例如: `$\overline{AB}$` 显示为 $\overline{AB}$, `$\underline{xy}$` 显示为 $\underline{xy}$ 。`amsmath` 也提供了一组在符号上下加划线的命令, 例如: `\overleftarrow{内容}`、`\overrightarrow{内容}`、`\underleftarrow{内容}` 和 `\underrightarrow{内容}`。`overrightarrow` 常用于表示向量, 和 `\vec` 效果类似; 但 `\vec` 适合单个字符的向量表示, 而 `\overrightarrow` 则适合多个字符的向量表示。

文本 在数学模式中插入文本时, 可以使用 `\text{文本}` 命令, 例如: `$\text{速度}$` 。该命令会确保文本以正常的字体和大小显示, 而不是数学字体。

28.1.2 巨算符

巨算符 (large operators) 是指那些在数学公式中用于表示求和、积分、极限等操作的符号。这些符号通常比普通的运算符更大, 并且在行间公式中会自动调整大小以适应上下文。最常见的巨算符包括: $\sum$, $\prod$, $\int$, $\oint$, $\lim$, $\cup$, $\cap$, $\vee$, $\wedge$ 等。

在行内数学模式中, 巨算符的上下限会显示为普通的上下标形式, 例如 $\sum_{i=1}^n a_i$ 。而在行间数学模式中, 巨算符的上下限会显示在符号的上下方, 例如: `\[\sum_{i=1}^n a_i\]` 显示为

$$\sum_{i=1}^n a_i$$

只有积分符号的上下限始终显示在符号的旁边, 无论是在行内还是行间数学模式中。

`amsmath` 提供了更多的巨算符, 例如: `\iint` (二重积分)、`\iiint` (三重积分)、`\iiiiint` (四重积分) 和 `\idotsint` (多重积分)。例如: `\[\iint_D f(x,y) \,dx\,dy\]` 显示为

$$\iint_D f(x,y) \,dx \,dy$$

28.1.3 数学模式的字体调整

在数学模式中, 可以使用不同的命令来调整字体样式, 以突出显示特定的数学符号或表达式。以下是一些常用的字体调整命令:

- `\mathrm{文本}`: 将文本设置为直立体, 例如`\mathrm{sin}(x)`显示为 $\sin(x)$ 。
- `\mathbf{文本}`: 将文本设置为粗体, 适用于表示向量或矩阵, 例如`\mathbf{v}`显示为 $\mathbf{v}$ 。
- `\mathit{文本}`: 将文本设置为斜体字体, 适用于变量名, 例如`\mathit{x}`显示为 x 。
- `\mathcal{文本}`: 将文本设置为花体字体, 适用于表示集合, 例如`\mathcal{A}`显示为 $\mathcal{A}$ 。
- `\mathbb{文本}`: 将文本设置为黑板粗体字体, 适用于表示数集, 例如`\mathbb{R}`显示为 $\mathbb{R}$ 。

28.2 图、表、浮动体

在 $\text{\LaTeX}$ 中, 图表和其他浮动体 (如表等) 是通过浮动体来管理的。浮动体允许 $\text{\LaTeX}$ 根据页面布局自动调整图表的位置, 以确保文档的美观和可读性。

浮动体之所以“浮动”, 是因为它们并不固定在文档中的某个位置, 而是可以根据页面布局和排版需求自动调整位置。这样可以避免图表被拆分到不同的页面上, 或者与文本内容重叠, 从而提高文档的整体质量。但这样会导致图表的位置不一定和代码中出现的位置一致, 例如

1 文本 1
2 浮动体
3 文本 2

上述编译出的文档中, 浮动体可能会出现在“文本 1”之前、“文本 1”和“文本 2”之间, 或者“文本 2”之后, 具体位置取决于页面布局。但多个浮动体在文本中的相对顺序是不会改变的。

28.2.1 插入图片

单一图片 在 $\text{\LaTeX}$ 中插入图片通常使用 `graphicx` 宏包。首先, 在导言区添加以下代码来加载该宏包:

```
1 \usepackage{graphicx}
```

然后, 可以使用 `figure` 环境来插入图片。例如:

```
1 \begin{figure}[htbp]
2   \centering
3   \includegraphics[width=0.5\textwidth]{example-image}
4   \caption{示例图片}
5   \label{fig:example}
6 \end{figure}
```

上述代码中, `[htbp]` 是浮动体的位置参数, 表示允许 $\text{\LaTeX}$ 将图片放置在此处 (here)、顶部 (top)、底部 (bottom) 或单独一页 (page)。`\centering` 用于将图片居中显示, `\includegraphics`

用于插入图片文件，`width=0.5\textwidth` 指定图片的宽度为文本宽度的一半。`\caption` 用于添加图片标题，`\label` 用于为图片设置标签，以便在文档中引用。

上述浮动体的位置参数如不写，则默认为 `[tbp]`（对，这里非常坑人，默认不包含 `h` 选项）。建议总是显式指定位置参数，以确保图片能够尽可能地出现在期望的位置。

如希望强制图片出现在代码中的位置，可以使用 `[H]` 位置参数，但需要加载 `float` 宏包：

```
1 \usepackage{float}
2
3 \begin{figure}[H]
4   ...
5 \end{figure}
```

需要注意的是，强制图片出现在指定位置可能会影响页面布局和排版效果，因此应谨慎使用。

`includegraphics` 命令支持多种图片格式，包括 PNG、JPEG、PDF 和 EPS 等。建议使用矢量图格式（如 PDF 或 EPS）以获得更好的缩放效果，尤其是在打印时。该命令有一些常用选项，例如：

- `width=宽度`：指定图片的宽度，例如 `width=0.5\textwidth`。
- `height=高度`：指定图片的高度，例如 `height=3cm`。
- `scale=比例`：按比例缩放图片，例如 `scale=0.8`。
- `angle=角度`：旋转图片，例如 `angle=90`。

最常用的莫过于设置宽度和高度，可以单独设置其中一个，另一个会按比例自动调整；也可以同时设置宽度和高度，但可能会导致图片变形。

`caption` 命令用于为图片添加标题，标题会自动编号，并且可以通过 `\label` 命令设置标签，以便在文档中引用。例如，可以使用 `\ref{fig:example}` 来引用图片。

子图 如果需要在同一个浮动体中插入多个子图，可以使用 `subfigure` 宏包。首先，在导言区添加以下代码来加载该宏包：

```
1 \usepackage{subfigure}
```

然后，可以使用 `subfigure` 环境来插入子图。例如：

```
1 \begin{figure}[htbp]
2   \centering
3   \subfigure[子图 1 标题]{
4     \includegraphics[width=0.4\textwidth]{example-image-a}
5     \label{fig:sub1}
6   }
7   \subfigure[子图 2 标题]{
8     \includegraphics[width=0.4\textwidth]{example-image-b}
9     \label{fig:sub2}
10  }
11  \caption{整体图片标题}
12  \label{fig:overall}
13 \end{figure}
```

上述代码中，`\subfigure` 用于插入子图，并且可以为每个子图添加标题和标签。整体图片的标题和标签仍然使用 `\caption` 和 `\label` 命令。

28.2.2 插入表格

在 $\text{\LaTeX}$ 中插入表格通常使用 `table` 环境和 `tabular` 环境。

朴素的表格 最朴素的表格利用 `tabular` 环境实现，例如：

```
1 \begin{tabular}{|c|c|c|}  
2   \hline  
3   列 1 & 列 2 & 列 3 \\  
4   \hline  
5   数据 1 & 数据 2 & 数据 3 \\  
6   数据 4 & 数据 5 & 数据 6 \\  
7   \hline  
8 \end{tabular}
```

`tabular` 不是浮动体环境，表格会出现在代码中的位置。大括号中的参数指定了表格的列格式，有几个字母就有几列，常用的字母包括 `c`、`l`、`r`，分别表示居中、左对齐和右对齐。竖线 `|` 表示列之间的竖线。

而表格中的每一行以双反斜杠 `(\\)` 结束，列与列之间使用 `&` 符号分隔，用于制表。例如，上述代码中的第一行表示表头，包含三列，分别是“列 1”、“列 2”和“列 3”。第二行和第三行分别表示两行数据。`\hline` 命令用于添加水平线。

也可以引用 `booktabs` 宏包来美化表格，例如使用 `\toprule`、`\midrule` 和 `\bottomrule` 命令来替代 `\hline`，从而获得更专业的表格外观，得到基础的“三线表”效果。

但是该朴素的表格不能添加标题和标签，也不能自动调整位置，更不能跨页显示。

带标题和标签的表格 可以使用 `table` 环境来插入带标题和标签的表格。例如：

```
1 \begin{table}[htbp]  
2   \centering  
3   \begin{tabular}{|c|c|c|}  
4     ...  
5   \end{tabular}  
6   \caption{示例表格}  
7 \end{table}
```

上述代码中的 `table` 是浮动体环境，允许 $\text{\LaTeX}$ 根据页面布局自动调整表格的位置。其余部分与前面介绍的 `tabular` 环境相同。但该表格依然不能跨页显示。

跨页表格 如果表格内容较多，可能会跨越多个页面。可以使用 `longtable` 宏包来创建跨页表格。首先，在导言区添加以下代码来加载该宏包：

```
1 \usepackage{longtable}
```

然后，可以使用 `longtable` 环境来插入跨页表格。例如：

```
1 \begin{longtable}{|c|c|c|}  
2   \caption{跨页表格示例} \\  
3   \hline  
4   列 1 & 列 2 & 列 3 \\  
5   \hline  
6   \endfirsthead  
7   \hline  
8   列 1 & 列 2 & 列 3 \\  
9   \hline  
10  \endhead  
11  数据 1 & 数据 2 & 数据 3 \\  
12  数据 4 & 数据 5 & 数据 6 \\  
13  ...  
14 \end{longtable}
```

上述代码中，`\endfirsthead` 和 `\endhead` 命令用于定义表格在每一页的表头。其余部分与前面介绍的 `tabular` 环境基本相同，但该环境能自动处理跨页问题，也能够添加标题、标签等功能。

28.3 自定义命令和自定义环境

在 *LaTeX* 中，自定义命令和自定义环境可以帮助你简化文档的编写过程，提高代码的可读性和可维护性，类似于编程语言中的函数和类。

28.3.1 自定义命令

可以使用 `\newcommand` 命令来定义新的命令。其基本语法如下：

```
1 \newcommand{\命令名}[参数个数]{命令定义}
```

其中，`\命令名` 是你想要定义的命令名称，`参数个数` 是该命令接受的参数个数（可选，默认为 0），`命令定义` 是该命令的具体实现。例如，定义一个命令来表示向量：

```
1 \newcommand{\vect}[1]{\mathbf{#1}}
```

上述代码定义了一个名为 `\vect` 的命令，它接受一个参数，并将该参数以粗体形式显示。可以在文档中使用该命令，例如：

```
1 这是一个向量：$\vect{v}$。
```

上述代码将在文档中显示为“这是一个向量：**v**”。

也可以更改现有的命令，例如：

```
1 \renewcommand{\emph}[1]{\textbf{#1}}
```

上述代码将 `\emph` 命令重新定义为粗体显示，而不是斜体显示。

我们发现这些命令定义中使用了 `#1` 这类的符号，这是因为在自定义命令中，参数是通过 `#1`、`#2` 等符号来引用的，分别对应第一个、第二个参数，以此类推。

也有一些较为特殊的用法，例如我“想重定义某命令，但还要保留其原有功能”，这时可以使用 `\let` 命令来实现。例如：

```
1 \let\oldemph\emph
2 \renewcommand{\emph}[1]{\textbf{\oldemph{#1}}}
```

上述代码首先使用 `\let` 命令将原有的 `\emph` 命令保存为 `\oldemph`，然后重新定义 `\emph` 命令，使其在显示为粗体的同时，仍然保留原有的斜体功能。

28.3.2 自定义环境

可以使用 `\newenvironment` 命令来定义新的环境。其基本语法如下：

```
1 \newenvironment{环境名}[参数个数]{环境开始代码}{环境结束代码}
```

其中，`环境名` 是你想要定义的环境名称，`参数个数` 是该环境接受的参数个数（可选，默认为 0），`环境开始代码` 是该环境开始时执行的代码，`环境结束代码` 是该环境结束时执行的代码。例如，定义一个名为 `highlight` 的环境，用于高亮显示文本：

```
1 \newenvironment{highlight}{\begin{quote}\color{red}}{\end{quote}}
```

上述代码定义了一个名为 `highlight` 的环境，它将在红色引用块中显示文本。可以在文档中使用该环境，例如：

```
1 \begin{highlight}
2 这是高亮显示的文本。
3 \end{highlight}
```

上述代码将在文档中显示为红色引用块中的“这是高亮显示的文本”。

同样的，环境也支持重新定义，例如：

```
1 \renewenvironment{quote}{\begin{itshape}}{\end{itshape}}
```

上述代码将 `quote` 环境重新定义为斜体显示。

28.4 使用 Beamer 类制作幻灯片

LaTeX 除了可以用来排版文章、书籍等文档外，还可以制作幻灯片。Beamer 就是用于制作幻灯片的一个文档类。以下会简单介绍如何使用 Beamer 类制作幻灯片。

28.4.1 Beamer 文档的基本结构

Beamer 类文档的基本结构和普通的 *LaTeX* 文档类似。然而，在正文部分，其内容并不是简单的段落，而是由多个幻灯片（frame）组成的。每个 frame 对应 PowerPoint 中的一页内容。一个简单的 Beamer 文档示例如下：

```

1  \documentclass{beamer}
2  \usepackage{amsmath}
3  \title{我的第一篇 Beamer 幻灯片}
4  \author{张三}
5  \date{\today}
6  \begin{document}
7  \frame{\titlepage} % 生成标题页
8  \begin{frame}
9      \frametitle{引言} % 设置帧标题
10     这是我的第一篇 Beamer 幻灯片。
11     我们可以在这里写一些数学公式：
12     \begin{equation}
13         E=mc^2
14     \end{equation}
15 \end{frame}
16 \end{document}

```

如上述代码所示，创建帧有两种方式：一种是使用 `\frame{}` 命令，另一种是使用 frame 环境。前者适合内容较少的帧，后者适合内容较多的帧。在帧中添加标题可以使用 `\frametitle{标题}` 命令，如希望添加简单文字、图片、公式等内容，可以直接在帧中编写，方法和普通的 *LaTeX* 文档几乎完全相同。然而，如果希望添加表格、代码等浮动内容，则需要在该帧加上 `[fragile]` 选项，也就是：

```

1  \begin{frame}[fragile]

```

28.4.2 Beamer 的常用强调

经常做幻灯片的同学们都知道，在幻灯片中希望强调某些内容的时候，最好不要仅使用字体的简单变化，而是结合显隐、颜色等手段来突出重点。对于部分颜色的文字，可以使用 `\textcolor{颜色名}{文本}` 命令来实现。例如：

```

1  \textcolor{red}{这是红色的文本}
2  \textcolor{blue}{这是蓝色的文本}

```

另外,它也提供了一些“块”命令来强调内容,例如 `block`、`alertblock` 和 `exampleblock` 等环境。例如:

```
1 \begin{block}{普通块}
2   这是一个普通块。
3 \end{block}
4 \begin{alertblock}{警告块}
5   这是一个警告块。
6 \end{alertblock}
7 \begin{exampleblock}{示例块}
8   这是一个示例块。
9 \end{exampleblock}
```

最后, Beamer 还支持显隐效果, 这种显隐效果在列表的逐步显示中非常有用。一个最简单的逐步显示的例子如下:

```
1 \begin{itemize}
2   \item<1-> 第一项
3   \item<2-> 第二项
4   \item<3-> 第三项
5 \end{itemize}
```

上述代码中, `<1->` 表示从这页的第一帧开始显示这一项, 以此类推。如果只想在某一帧显示某一项, 可以使用 `<n>` 的形式, 例如 `<2>` 表示只在第二帧显示这一项。上述代码也可以用在其他环境中, 例如段落、公式等。

除了这种简单的显隐效果外, 还可以使用 `\pause` 命令来实现逐步显示的效果。例如:

```
1 这是第一部分内容。
2 \pause
3 这是第二部分内容。
```

Beamer 还支持更复杂的显隐效果, 感兴趣的同学可以查阅 Beamer 的文档。

28.4.3 Beamer 的主题

结构主题决定了幻灯片的整体风格。Beamer 提供了多种内置主题, 用户可以根据需要选择合适的主题。可以使用 `\usetheme{主题名}` 命令来设置主题。内置的主题有许多, 我们可以通过查阅 Beamer 的文档来了解这些主题的具体名称和效果。我个人比较喜欢使用 `Berlin` 主题, 该主题简洁大方, 适合大多数场合。

配色主题决定了幻灯片的配色方案。可以使用 `\usecolortheme{配色主题名}` 命令来设置配色主题。内置的配色主题也有许多, 例如 `dolphin`、`seahorse`、`beetle` 等。

字体主题决定了幻灯片中使用的字体样式。可以使用 `\usefonttheme{字体主题名}` 命令来设置字体主题。一般情况下, Beamer 使用的字体系列和常规文档 (serif) 不同, 我们可以使用 `\usefonttheme{serif}` 来强制使用衬线字体。如果仅希望更改公式部分的字体, 可以使用 `\usefonttheme[onlymath]{serif}` 。

另外，Beamer 允许用户创建自己的主题，或者使用第三方提供的主题包。创建自定义主题需要一定的 *LaTeX* 和 Beamer 知识，建议有一定经验的用户尝试。

28.4.4 ctexbeamer

ctexbeamer 是类似 ctexart、ctexbook 等 ctex 系列文档类的 Beamer 版本，专门用于处理中文幻灯片。使用 ctexbeamer 可以方便地在幻灯片中输入和排版中文内容，而无需额外配置中文支持。使用 ctexbeamer 的基本方法与使用 Beamer 类类似，只需将文档类更改为 ctexbeamer 即可。例如：

```
1 \documentclass{ctexbeamer}
2 ...
```

ctexbeamer 默认使用 xeCJK 宏包处理中文，因此可以直接在幻灯片中输入中文内容，而无需额外配置中文支持。除此之外，ctexbeamer 还提供了一些特定于中文幻灯片的功能和选项，例如自动调整字体大小、处理中文标点等。

28.5 使用 ModernCV 类制作简历

ModernCV 是一个用于制作简历的 *LaTeX* 文档类。它提供了多种简洁、美观的简历模板，用户可以根据需要选择合适的模板来制作自己的简历。当然，这类简历极为简洁，然而对于技术类岗位也已经绰绰有余；如希望更花哨、更复杂的简历，这个则略显平淡，可以去淘宝花几块钱买个模板（或者转行当 UI 设计师）。

以下是一个使用 ModernCV 类制作简历的简单示例：

```
1 \documentclass{moderncv}
2 \usepackage[UTF8]{ctex}
3
4 \moderncvtheme[blue]{casual}           % 主题选 casual，颜色选忧郁蓝
5 \name{王小明}{摸鱼学硕士}             % 名字和头衔分开写，防止过于嚣张
6 \email{run@fast.com}                  % 建议别用 "ilovexxx@qq.com"
7 \social[github]{github.com/xxx}
8 \social[linkedin]{xxx}                % 假装有国际化人脉
9 \quote{Stay hungry, stay foolish}      % 乔布斯名言，HR 已免疫
10
11 \begin{document}
12 \maketitle % 生成个人信息区块
13
14 \section{教育背景}
15 \cventry{2021--2025}{专业：理论与实操}{某大学}{寄点 3.0/4.0}{{核心课程：《咖啡因代谢学》《早八生存指南》}}
16
17 \section{实习经历}
18 \cventry{2023.06--2023.09}{首席工位守护者}{某厂}{{}}{{主要成就：带薪拉屎时长部门 TOP1}}
19 \end{document}
```


如你所见，ModernCV 的使用非常简单。我们只需在导言区加载 `moderncv` 宏包，并设置主题和颜色，然后使用 `\name`、`\email` 和 `\social` 等命令来添加个人信息。正文部分则使用 `\section` 和 `\cventry` 等命令来添加简历内容。这样就可以快速地生成一份简洁、美观的简历，免除了使用 Word 等工具排版的烦恼。

该包的另一个极为良好的特性是可以自动压缩空白区域，可以有效地避免“半页纸尴尬”。但是与其这样，感觉确实不如多整点活填满这页纸（大嘘）。所以欢迎来 LCPU 玩混项目（不是）。

以上就是 $\text{\LaTeX}$ 的相当多内容了。掌握这些内容后，你已经可以使用 $\text{\LaTeX}$ 来排版大部分常见的文档类型了。当然， $\text{\LaTeX}$ 的功能远不止于此，还有许多高级技巧和宏包可以进一步扩展其功能。希望你能继续探索 $\text{\LaTeX}$ 的世界，发现更多有趣的用法！

第八部分

延伸进阶阅读

第二十九章 现代高性能并发编程：异步、多进程和多线程

本章介绍了现代高性能并发编程的基本概念和技术，涵盖了异步编程、多进程编程和多线程编程的原理和实践。通过学习本章内容，读者将能够理解并应用这些技术来提升程序的性能和响应能力。

我们刚学编程的时候，代码大多数是“把所有的东西一口气做完”，代码一行接一行，像火车过隧道，即使漆黑一片也得走完。后来，我们发现，CPU 核心越来越多，但硬盘、网络等数据来源的速度根本跟不上。我们发现，试图一口气把所有东西搬进内存时，CPU 在睡觉；在等用户输入内容的时候，CPU 就阻塞住，啥也不做。而这显然和实际生活中的情况不符：工人砌砖的时候，难道要等所有水泥全都送到才开始砌墙吗？显然不是的。

那为了让 CPU 不睡觉，我们就有这两个类似的思路：

- 异步：让一个工人一边等待新的水泥，一边用手头显然不够的水泥砌墙；
- 多线程：让多个工人同时工作，有的人去搬运水泥，有的人去砌墙。

这两种思路各有优缺点。下文将会逐一阐述。

由于 Python 的方便性，本文前半部分的代码示例均使用 Python 编写。

29.1 从一口气做完到边做边等

假设我们写了这样一个脚本，顺序抓取 100 个网页并处理：

```
1 for url in urls:
2     html = request.urlopen(url).read() # 网络往返 200 ms
3     process(html)                      # CPU 计算 5 ms
```

我们发现，CPU 利用率竟然高达 $5 \div 205 \approx 2.4\%$ ！有高达 97% 的时间被阻塞在 read 这一步：在等待网络数据返回的时候，CPU 啥也不干。

那怎么处理这个问题？一拍脑袋，“进程是资源的调度单位”，那我们就开多个进程同时抓取网页吧！

```
1 for url in urls:
2     pid = os.fork() # 经典 Unix 套路
3     if pid == 0:    # 子进程
```

```
4     single_fetch(url)
5     os._exit(0)
```

这就是朴素的多进程编程了。

问题貌似解决了，但我们发现：`fork` 瞬间复制父进程内存，100 个进程同时跑，内存瞬间爆炸；而且，操作系统调度这么多进程，开销也不小。更捉急的是：子进程把结果存在哪儿？父进程怎么拿到结果？这都没想好，代码就乱成一锅粥了。

于是我们被迫继续思考：要么让进程“轻一点”，要么让进程“少一点”，甚至干脆就不要阻塞了！

29.2 异步：一个进程来回跳

29.2.1 事件循环和协程

单线程内，我们不如换一种思路：遇到 IO 不睡觉，我们先去做别的；等数据好了，再回来继续处理。这样一来，CPU 就不会闲着了。

这就是**事件循环**的核心：

```
1 loop = get_event_loop()
2 for url in urls:
3     loop.create_task(fetch(url))  # 立刻返回，不阻塞
4 loop.run_forever()
```

`fetch` 内部在真正读写数据时会 `await`，也就是“暂时等着”，让出控制权、挂起当前协程：“Nobody blocks, Everybody cooperates”。于是 CPU 的利用率立刻提升，但内存并无显著提升。

事件循环的核心是“协程”（coroutine）。协程是一种“可暂停的函数”，它可以在执行过程中被挂起，等到某个条件满足时再恢复执行。我们经常在现成代码中看到的 `async` 和 `await`，就是用来定义和使用协程的关键字，但管它是 C# 还是 Python，本质上都是**状态机和回调函数**的语法糖。

那状态机和回调函数是什么呢？简单来说，状态机就是一个有多个状态的机器，每个状态都有自己的行为和转换规则；回调函数则是当某个事件发生时被调用的函数。协程通过状态机来管理自己的执行状态，通过回调函数来处理异步事件，从而实现非阻塞的执行。

于是，编译器或解释器把协程切成上半段和下半段，遇到 `await` 时就保存当前状态，中间插个注册回调函数的操作，等事件完成时再调用回调函数恢复状态，继续执行下半段代码。这样一来，单线程内就能实现多任务并发执行了；而对开发者而言，代码看起来和同步代码差不多，易读性大大提升。

29.2.2 异步的边界

异步的“不阻塞”指的仅仅是 IO。一旦上述 `process(html)` 真就成了大型纯计算，那事件循环就被霸占，所有协程集体卡死——异步失效。

于是新问题就诞生了：计算太重，异步扛不住；进程太多，内存又爆炸。怎么办呢？有没有介于两者之间的方案？

先卖关子，当然我倒是可以告诉你答案是“线程”。但还得把进程讲完，再请这尊大佛。

29.3 多进程编程：尤里复制人

29.3.1 fork 之后

刚刚我创建进程的时候，用了 `os.fork()`。学过 ICS 的肯定认识这个函数：它会复制当前进程的内存，创建一个一模一样的新进程。新进程从 `fork()` 返回，父进程得到新进程的 PID，子进程得到 0。这是 Unix-Like 系统惯用的套路。

但这个套路有个大问题：内存被完全复制了！假设父进程有 1GB 内存，fork 后子进程也有 1GB 内存，系统总共就得分配 2GB 内存。要是我 fork 100 个进程呢？那不就得 200GB 内存了吗？

实际上 Unix-Like 系统搞出这玩意之后，早就料到了“全量拷贝”还是太不“环保”了，于是引入了“写时复制”（Copy-On-Write, COW）技术：fork 之后，父子进程共享同一块内存区域，读共享，谁写谁复制。这样一来，内存占用就大大减少了。

但这仍然挡不住“一页未写、逻辑却独立”的浪费。于是，`execve` 函数应运而生：它可以让子进程加载一个全新的程序镜像，彻底摆脱父进程的内存束缚，旧的地址空间直接丢掉，又省内存又解耦逻辑，干净利落。

于是新的经典模式诞生了：先 fork，再 exec。先复制父进程的环境，然后加载新程序。这样一来，子进程就有了自己的逻辑和内存空间，互不干扰。shell 的启动流水线就是这么干的。

29.3.2 进程间的通信

但问题又来了：进程间如何通信？父子进程的内存空间独立了，数据如何传递？

经典的 Unix 哲学是“一切皆文件”，进程间通信（Inter-Process Communication, IPC）自然也离不开文件。于是，最常见的 IPC 方式无非匿名管道、命名管道（FIFO）、共享内存、消息队列和套接字等。

管道那可是 Unix 系的老朋友了，数据一头进一头出，简单高效；一般匿名管道只能父子进程之间用，命名管道则可以跨进程；共享内存比它们都要快，是页级别的共享，但是**需要自己处理同步**，不然数据乱套；消息队列则是更高级的 IPC 机制，支持优先级和异步通信；套接字则是网络编程的基础，可以实现跨机器通信。

换句话说，进程之间的通信本质上都是为了**数据同步**。即使我们确实是把一个任务拆成多个进程来做，但最终肯定还是要汇总结果的；在这个过程中，数据的传递和同步就显得尤为重要。但这些子任务肯定逻辑上也是有先后顺序的：比如说，先抓取网页，再处理网页内容。那我们就得确保抓取任务完成后，处理任务才能开始。要不然，那就乱套了。

29.3.3 进程池

既然进程开多了内存爆炸，开少了 CPU 利用率低，那我们不如折中一下，先生成一些进程让他们睡觉，任务来了就分配给它们去做，做完了接着睡。这个思路听起来不错，而这就是大名鼎鼎的“进程池”（Process Pool）。

所谓进程池，就是预先创建一组进程，任务来了就分配给空闲的进程去做，做完再回来等下一个任务。这样一来，既能提高 CPU 利用率，又能控制内存占用。

29.3.4 多进程编程

那有的人说我三纸无驴，讲了这么多才提到真正的多进程编程。我承认这种指摘，但是因为现代的多进程基本依附于进程池而存在，脱离进程池则多进程几乎无意义，所以只能把进程池讲完以后再提多进程编程了。

现代的多进程编程是典型的异步-多进程混搭：主进程做纯异步的 IO 调度，而子进程统统塞进进程池里做纯计算，通过 UNIX Domain Socket 或共享内存等等传递数据；主进程则仅负责事件循环和调度，绝不自我阻塞。这样一来，CPU 利用率和内存占用都能得到很好的平衡。

现代编程中随处可见上述典型架构：Nginx+Flask/uWSGI、Chrome 的 GPU 进程、VS Code 的 Extension Host，全都是这种异步主控、多进程打工的模式。学会这种模式，才能真正理解现代高性能编程的精髓。

来个简单示范：

```
1 import asyncio, multiprocessing as mp
2
3 def cpu_crunch(data):
4     # 真正的计算，GIL 也挡不住的 CPU 密集
5     return sum(i*i for i in data)
6
7 async def main():
8     with mp.Pool() as pool:
9         loop = asyncio.get_running_loop()
10        result = await loop.run_in_executor(pool, cpu_crunch, range(10_000_000))
11        print("async 拿到结果:", result)
12
13 asyncio.run(main())
```

上述代码中，主进程运行一个异步事件循环，负责调度任务；而真正的计算任务则被分配给进程池中的子进程去执行，由 `run_in_executor` 方法把进程池包装成一个异步调用：异步和多进程永远不是二选一，而是组合拳。

29.3.5 那为什么还要多线程？

读到这儿，细心的你会问：“异步解决 I/O 等，进程池解决计算重，线程究竟图什么？”

那自然是有它的优势的：

- 进程要切换内核态重跑，乃至页表刷新、TLB Flush，时间是微秒级别的；线程切换用户态就能完成，时间在百纳秒级别，开销小得多。

- 某些资源没法拷贝，比如大缓存、数据库连接，这些东西按着头也得共享。
- 部分系统调用一次返回的事件要分给多处消费，多线程监听一个资源比多进程方便得多。

如果你确实想弄明白，就继续往下看吧。但多线程的难度，那可是非常高的。下文我直接从 Python 切成 C++ 来讲解，在硬件底层控制方面目前还没有比 C 和 C++ 更好的语言了。（Rust 是什么东西？先不提它。）

为了更好地理解多线程编程，我们需要先了解程序的执行原理。

29.4 程序执行原理

29.4.1 进程

我们知道，操作系统的本质是一个“资源管理器”，负责管理计算机的各种资源。而一个程序希望运行起来，就需要操作系统分配给它一些资源，比如 CPU 时间、内存空间等，因此操作系统需要“调度”程序的运行。而这个调度的基本单位就是**进程**（Process）。

我们暂时先仅考虑一个程序。当我们运行一个程序时，操作系统会为该程序创建一个进程，并分配相应的资源。进程是程序在操作系统中的一个实例，它包含了程序的代码、数据以及运行时的状态信息。每个进程都有自己独立的内存空间和系统资源，且一般不能与其他进程直接共享资源。

一般而言，一个程序就是一个进程。当一个程序被运行，也就是一个进程被创建时，操作系统会为该进程分配一个唯一的“进程标识符”（Process ID, PID），并为其分配内存空间和其他资源。进程中的代码会被加载到内存中，并开始执行。这个过程的资源开销是比较大的。进程之间的通信等也比较麻烦，需要通过管道、共享内存等机制来实现。

而现代计算机显然不能一个时间内仅运行一个程序。操作系统通过“时间片轮转”（Time Slicing）等调度算法，让多个进程交替使用 CPU 时间，从而实现多任务处理。这样，用户可以同时运行多个程序，而操作系统会在它们之间切换，使得每个程序都能获得一定的 CPU 时间。而这样的切换过程是由操作系统内核负责管理的，其开销也是比较大的。

29.4.2 线程

而在一些情况下，我们希望一个程序能够同时执行多个任务。以著名大型即时战略游戏《群星》为例，游戏同时需要计算 AI 决策、船只轨迹、空间与地面战斗逻辑、资源收支、人口增减，乃至图形渲染、音频播放等。如果把这些东西都写成一个同步的程序，那么假设 CPU 计算上述任务的时间为 t_i ，则游戏内每过一天（也就是游戏开发常说的“游戏刻”或“tick”）所需的时间就是 $\sum t_i$ ，这样就会导致游戏的响应速度非常慢，玩家体验极差。而如果把上述任务全都写成不同的进程，那么操作系统就需要频繁地在这些进程之间切换，导致大量的时间浪费在进程切换上，游戏的响应速度虽然会比单进程好一些，但仍然不够理想。

我们知道，现代 CPU 往往是多核的，每一个核心都能基本独立地执行一些任务。能不能把上述复杂的计算任务拆成多个子任务，并让它们同时运行在不同的核心上，最终返回其结果，从而提升整体的计算效率呢？答案是肯定的。

我们引入**线程**（Thread）这个概念。线程指的是 CPU 调度的最小单位，可以理解为一个相对独立的计算任务，但是许多线程可以共享同一个进程的资源。线程之间可以共享进程的内存空间和其他资源，因此线程之间的通信和数据共享相对容易，开销也较小。

这样，一个复杂的计算任务（进程）就可以被拆分成许多小的线程，并让它们同时运行在不同的 CPU 核心上，从而提升整体的计算效率。或者说，进程是“项目组”，而线程则是“员工”；从国家或公司领取资源是以“项目组”的名义领取的，而具体的工作则是由“员工”来完成的。通过合理地分配任务，不同的员工同时进行不同的工作，就能提高整个项目组的工作效率。

以上述游戏为例，我们可以把 AI 决策、船只轨迹、空间与地面战斗逻辑等任务都拆成不同的线程，并让它们同时运行在不同的 CPU 核心上。这样，游戏每过一天所需的时间就变成了 $\max(t_i)$ ，大大提升了游戏的响应速度。

29.5 多线程编程基本思想

理论存在，但实践是检验真理的唯一标准。

多线程编程出现时立刻面临两个极端严重的问题，这两个问题直接影响结果的正确性以及程序能否执行下去：

1. 内存只有一块，多个线程怎么分配？会不会把东西都写到一块，结果彻底乱套？
2. 线程 A 在等线程 B 的结果，而线程 B 又在等线程 A 的结果，结果两个线程就这么干瞪眼怎么办？

这些问题，本质上是“并发”带来的问题。多线程编程的核心思想，就是要解决这些并发问题，从而实现高效且正确的并行计算。

29.5.1 并发和并行

“并发”可不是“并行”。并发是“一起出发”，逻辑上同时处理多件事，例如一个熟练的厨师能一边备菜一边起锅烧油；而并行是“同时进行”，物理上同时处理多件事，例如多个厨师同时在不同的灶台上做菜。在单核的 CPU 上只能并发，而在多核的 CPU 或多个 CPU 上可以并行。写代码的时候，必须先严格地按并发思维设计，然后再考虑并行能不能提升，这关系到数据访问的正确性。否则，一旦并发逻辑设计错了，再多的并行也没用，结果只会错上加错。

29.5.2 问题抽象

上述三个问题，实际上可以抽象成两个基本问题：

竞态条件

多个线程同时读写同一块内存，至少一个是写，结果取决于执行顺序，典型表现为同样输入多次，得到的结果不同。

例如，线程 A 和线程 B 同时对变量 x 进行操作，线程 A 执行 $x = x + 1$ ，线程 B 执行 $x = x * 2$ 。如果线程 A 先执行，那么最终结果是 $(x + 1) * 2$ ；如果线程 B 先执行，那么最终结果是 $x * 2 + 1$ 。这种情况下，结果取决于哪个线程先执行，导致程序行为不可预测。

死锁

多个线程互相等待对方释放资源，结果这些线程都永久阻塞，无法继续执行。其四个必要条件为**线程互斥**、**占有且等待**、**不可剥夺**、**循环等待**。简单来说，就是多个线程各自占有有一些资源，并且还在等待其他线程占有的资源，而这些资源又不能被强制剥夺，最终导致所有线程都在等待对方释放资源，形成一个循环等待的局面。只要缺失一个条件，就不会发生死锁。

而这两个基本问题最终都可以找到起源：**无序访问**。也就是说，多个线程对共享资源的访问没有一个明确的顺序，导致了竞态条件和死锁的发生。因此，解决多线程编程中的并发问题，关键在于如何确保对共享资源的有序访问。

那有的同学就会说了，“只要访问有序不就得得了？”没错，但问题是“有序”怎么保证？这就需要一些机制来实现了。

29.5.3 解决问题

为了解决上述并发问题，前人发明了多种手段来保证对共享资源的有序访问。

互斥量

这是最简单粗暴的手段。互斥量指的是：当一个线程试图访问某个共享资源时，必须先获得该资源的“锁”（Lock）。如果该资源已经被其他线程锁定，那么该线程就必须等待，直到资源被释放为止。这样，就能确保同一时间只有一个线程能够访问该资源，从而避免竞态条件的发生。

打个比方，一个人进门要先拿钥匙把门锁打开，进去了之后要把门锁上，别人才能进来。这样就保证了同一时间只有一个人能进屋子。

```
1  std::mutex mtx; // 创建一个互斥量
2  int shared_resource = 0; // 共享资源
3  void thread_function() {
4      mtx.lock(); // 获取锁
5      // 访问共享资源
6      shared_resource++;
7      mtx.unlock(); // 释放锁
8  }
```

当然上述代码比较简单粗暴，实践中一般用下面这个，构造函数获取锁，析构函数释放锁，避免忘记释放锁的问题：

```
1  std::mutex mtx; // 创建一个互斥量
2  int shared_resource = 0; // 共享资源
3  void thread_function() {
```

```
4         std::lock_guard<std::mutex> lock(mtx); // 自动获取和释放锁
5         // 访问共享资源
6         shared_resource++;
7     }
```

条件变量

有时候，线程需要等待某个条件成立才能继续执行。条件变量（Condition Variable）就是用来实现这种等待机制的。线程可以在条件变量上等待，直到另一个线程通知它条件已经成立。这样就可以避免死锁的发生。

```
1 std::mutex mtx;
2 std::condition_variable cv;
3 bool ready = false;
4 void thread_function() {
5     std::unique_lock<std::mutex> lock(mtx);
6     cv.wait(lock, []{ return ready; }); // 等待条件成立
7     // 继续执行
8 }
9 void signal_function() {
10     {
11         std::lock_guard<std::mutex> lock(mtx);
12         ready = true; // 设置条件
13     }
14     cv.notify_all(); // 通知等待的线程
15 }
```

通过使用互斥量和条件变量等机制，我们可以有效地解决多线程编程中的并发问题，从而实现高效且正确的并行计算。在实际编程中，还需要根据具体的应用场景选择合适的同步机制，以确保程序的正确性和性能。

除了这两个家伙外，还有更激进的手段：

原子操作

原子操作（Atomic Operation）指的是一种不可分割的操作，即在执行过程中不会被中断。这样，多个线程同时执行原子操作时，从源头上就避免了竞态条件的发生。

```
1 std::atomic<int> shared_resource{0}; // 共享资源
2 shared_resource.fetch_add(1); // 原子加 1，不会被中断
```

这个 `fetch_add` 函数会将共享资源加 1，并返回加 1 前的值。由于这是一个原子操作，多个线程同时执行这个操作时，不会发生竞态条件。

无锁数据结构

无锁数据结构（Lock-Free Data Structure）是一种特殊的数据结构，设计时就考虑到了多线程环境下的并发访问。它们通过使用原子操作和内存屏障等技术，实现了在不使用锁的情况

下，多个线程可以安全地访问和修改数据结构。

这样就能彻底摒弃互斥量，能把并行性能推向极致。但是其复杂度极高，我个人非常建议同学们先学会走路再学飞。这里就不展开讲解了。

29.5.4 新的问题与内存模型

但是，即使加了锁，仍然可能看到奇奇怪怪的数值。这是因为现代 CPU 和编译器为了提升性能，往往会对指令进行重排序（Instruction Reordering）；每个核心也有自己的缓存行，写入也未必立即刷回主存（内存）。这可真是福祸相依：一方面，重排序能提升指令执行效率，但另一方面，也可能导致多线程程序中的数据访问顺序与预期不符，从而导致加了锁也看到了奇怪的数值。

那这就很麻烦了。对此，人们又发明了“内存模型”（Memory Model）这个概念。内存模型定义了多线程程序中不同线程对共享内存的访问顺序和可见性规则。通过使用内存屏障（Memory Barrier）等机制，可以确保某些操作在特定的顺序下执行，从而避免由于重排序和缓存引起的问题。该操作与原子操作配套，用 happens-before 关系来描述操作之间的顺序，从而解决上述问题。

初学阶段，用 `std::memory_order_seq_cst` 就行了，保证所有操作按程序顺序执行。等到性能瓶颈的时候，再考虑逐步降级到更松的内存顺序模型，如 `acquire` 和 `release`，以提升性能。

29.5.5 实践之前的碎碎念

多线程代码的调试成本是我们常规单线程的指数倍。因为多线程程序的行为往往是非确定性的，同样的输入可能会导致不同的执行路径和结果，这使得调试变得非常复杂。此外，多线程程序中可能存在竞态条件、死锁等并发问题，这些问题往往难以重现和定位。所以，在写多线程代码之前，请务必三思：**先保证对，再考虑快。**

其行业共识是：

- **能不用共享内存就不用。**优先用消息队列、生产者-消费者模型、future-promise 模型等值传递的方式来设计多线程程序，避免共享内存带来的复杂性，这样把实际上的并发问题转化为串行状态机的问题，简单且高效。
- **把可变状态局限起来。**如果必须用共享内存，尽量把可变状态局限在某个特定的模块或类，所有访问走统一接口，内部用锁或无锁数据结构来保证有序访问。外部调用的人不需要关心并发细节，只需要调用接口即可。这样把并发可能带来的问题局限到最小范围内，便于调试和维护。
- **尽量用高级抽象。**优先使用语言或库提供的高级并发抽象，比如线程池、并发容器、任务调度器等，避免直接操作线程和锁。这样能减少低级并发编程的复杂性，提高代码的可读性和可维护性。

因此，在实际编程中，建议先从简单的并发模型入手，逐步深入理解多线程编程的原理和技巧。只有在真正需要高性能并行计算时，才考虑使用复杂的多线程技术。切记：**先保证对，再考虑快！**

29.5.6 相关模型解释

我们用点菜来解释两种常见的多线程编程模型。

我们去饭店吃饭，点了一份菜。下单时，厨房会给我们一张小票，然后厨房就在后面炒菜。等菜做好了就把菜端出来，放到传菜口。我们可以随时凭票去取餐，而不需要一直盯着厨房。

在上述实例中：

- 厨房是生产者：负责制作菜品，也就是处理任务，产生结果。
- 我们是消费者：负责取餐和享用菜品，也就是后续的处理任务。
- 小票是 **Promise**：指的是生产者对任务结果的承诺，在这里是用小票暂时地代替菜品。同时，我们拿到小票之后，它就变成了一个 **Future**，是消费者对任务结果的期望，在这里表示未来可以凭它来取餐。
- 菜是结果：生产者完成的任务结果，在这里是菜品本身。
- 传菜口是消息队列：负责在生产者和消费者之间传递任务结果。

一般情况下，生产者持有 **Promise** 对象，消费者持有 **Future** 对象。生产者在完成任务后，通过 **Promise** 对象将结果传递给 **Future** 对象，而消费者则通过 **Future** 对象获取结果。而这就是多线程模型中的五个重要角色，它们构成了三个最重要的模型：生产者-消费者模型和 **Future-Promise** 模型，以及消息队列模型。

那么这些模型有什么关系？

首先，生产者-消费者模型其实是多线程乃至多进程中最常见的模型，它实际上是描述并发任务的根本模式，不论是消息队列还是承诺未来，实际上都是在生产者-消费者模式下的具体实现方式。

模型都涉及到**异步处理**，也就是消费者不需要一直等着生产者完成它需要的人物，而是可以先去做其他事情，等到需要结果时再去取。而生产者、消费者之间则通过某种中间机制（例如消息队列或 **Future-Promise**）来传递结果。这两种方式都是仅传递结果，而不传递状态，防止直接的共享内存访问，从而避免了竞态条件的问题。

那么不同点在哪里呢？主要区别为以下五点：

- **通信方式**：消息队列模型通过消息队列进行通信，而 **Future-Promise** 模型通过 **Future** 和 **Promise** 对象进行通信。
- **数据传递**：消息队列模型传递的是消息，而 **Future-Promise** 模型传递的是任务结果。
- **消费者行为**：在消息队列模型中，消费者通常是主动从队列中获取消息，而在 **Future-Promise** 模型中，消费者通常是被动等待 **Future** 对象的结果。
- **结果获取方式**：在消息队列模型中，消费者需要主动从队列中获取消息（如轮询等），而在 **Future-Promise** 模型中，消费者可以通过 **Future** 对象的接口来获取结果。
- **适用场景**：消息队列模型适用于更大的任务，例如分布式系统、解耦服务乃至微服务架构等；而 **Future-Promise** 模型更适用于较小的任务，例如单个进程或线程内部的异步任务、并发编程等。

两者都有效地避免了直接的共享内存访问，降低了并发编程的复杂性，提高了系统的可靠性和可扩展性。

29.6 多线程的实践

29.6.1 实例：问题描述

用 C++ 完成以下任务：

现在有一个文本文件，是 UTF-8 编码的纯英文文本，大小巨大（GB 级别）。统计其中每个单词（连续的字母、数字、下划线）的出现频次，输出频次最高的十个单词以及其出现次数。**你必须保证结果的正确性**，并使得内存占用尽量可控，不一次性把整个文件读入内存。你自然是可以使用 CMake 等工具来管理你的项目的。大小写可以不敏感。

测试数据：可以用

```
1 wget https://git.savannah.gnu.org/cgit/grep.git/plain/README -O sample.txt
```

这个是 GNU Grep 的 README 文件，内容比较丰富。

原文档只有 2.3KB，我们可以用下面的命令把它重复 100 次，然后把这个头尾相接，生成一个大文件：

```
1 for i in {1..100}; do cat sample.txt >> bigfile.txt; done
```

当然你可以用其他更大的文本文件来测试，也可以调大重复次数，以生成更大的文件。

29.6.2 实例：设计思路

思路是显然的，使用分块读取-独立统计-合并结果的 Map-Reduce 风格。考虑三大行业共识：

- **不用共享内存**：每个线程持有一个 map，保证线程之间零共享数据，彻底消除竞态条件；
- **局限可变状态**：让唯一的可变状态为最终汇总的 TotalCounts。这东西在所有线程结束之后，由主线程串行地合并，尽量缩短临界区。
- **使用高级抽象**：用 jthread、future 和 async 分别进行线程管理、异步结果和并行任务拆分。

关键点：怎么把任务拆分到多个线程上？思路是把文件分成多个块，每个线程负责处理其中的一块。这个“分治”思想相当常见，应该在大二就学过了。为了避免线程数过少导致的负载不均衡问题，我们可以把文件分成更多的块，然后让线程池中的线程去处理这些块。这样就能充分利用多核 CPU 的计算能力。

而分治的合并往往又是一个难题：假如一个单词跨块了怎么办？解决办法是让每个块的处理函数返回一个“前缀”字符串，表示该块最后一个单词的前半部分。下一个块在处理时，先把这个前缀加到自己的开头，然后再进行统计。这样就能确保每个单词都被完整地统计到。**但是！**这种做法依然有一个问题：现在把一个文件分成若干块（A、B、C、……），B 线程可能没拿到 A 的前缀就开跑了，结果导致 A 的最后一个单词和 B 的第一个单词依然被拆开了。为了解决这个问题，我们可以让每个块在处理时，先检查自己的前一个块是否已经处理完毕，并获

取其前缀。如果前一个块还没有处理完毕，那么当前块就等待，直到前一个块处理完毕并返回前缀。这样就能确保每个块在处理时都能获取到正确的前缀，从而避免单词被拆开的情况。

这实在是太让人晕头转向了。不如考虑一个更简单的办法：**在分割的时候就不要跨单词分割**，也就是宁可多读一点，也不要拆单词。这样就能彻底避免上述问题。具体做法是：在分割文件时，先按固定大小分割成若干块，然后检查每个块的结尾是否为单词边界。如果不是，就继续向后读取，直到遇到下一个非单词字符为止。这样就能确保每个块都是完整的单词，从而避免单词被拆开的情况。毕竟还是那句话：**先保证对，再考虑快！**

当然，如果你想挑战更高难度，也可以把上述问题中的“纯英文”改成“任意 UTF-8 编码的文本”，这样就需要考虑多字节字符的情况了，例如变音字符、汉字等。这个就留给同学们自行挑战了。

29.6.3 实例：实际工作

说明

这个是笔者写的一个示例实现，供同学们参考学习。本人并没有打过诸如 HPC Game 之类的比赛，这个实现经测试是正确的，但没有经过严格的性能调优，仅供学习多线程编程之用。实际上肯定会有很多可以改进的地方，欢迎同学们自行优化。

定义项目文件结构，使用 CMake 管理项目：

```
1 demo/
2   CMakeLists.txt
3   include/
4     wc.hpp
5   src/
6     wc.cpp
7     main.cpp
```

wc.hpp

```
1 #pragma once
2 #include <string>
3 #include <unordered_map>
4
5 using Freq = std::unordered_map<std::string, std::size_t>;
6
7 /* 统计文件，返回全局频次表 */
8 Freq count_file(const std::string& path);           // 可能抛 std::system_error
```

源文件 wc.cpp

```
1 #include "wc.hpp"
2 #include <algorithm>
3 #include <cctype>
4 #include <cstring>
5 #include <filesystem>
```

```

6 #include <future>
7 #include <iterator>
8 #include <stdexcept>
9 #include <system_error>
10 #include <vector>
11 #include <thread>
12 #include <fcntl.h>
13 #include <sys/mman.h>
14 #include <sys/stat.h>
15 #include <unistd.h>
16
17 // ----- 低层 OS 封装 -----
18 namespace os {
19
20 // RAII 包装 mmap
21 class mapped_region {
22 public:
23     mapped_region(const std::filesystem::path& p)
24         : size_(std::filesystem::file_size(p)),
25           data_(do_map(p.c_str(), size_)) {}
26
27     ~mapped_region() { ::munmap(data_, size_); }
28
29     mapped_region(const mapped_region&) = delete;
30     mapped_region& operator=(const mapped_region&) = delete;
31
32     const char* data() const noexcept { return static_cast<const char*>(data_); }
33     std::size_t size() const noexcept { return size_; }
34
35 private:
36     static void* do_map(const char* path, std::size_t sz) {
37         int fd = ::open(path, O_RDONLY);
38         if (fd < 0) throw std::system_error(errno, std::system_category(),
39                                             "open: " + std::string(path));
40         void* p = ::mmap(nullptr, sz, PROT_READ, MAP_PRIVATE, fd, 0);
41         ::close(fd);
42         if (p == MAP_FAILED)
43             throw std::system_error(errno, std::system_category(), "mmap");
44         return p;
45     }
46     std::size_t size_;
47     void* data_;
48 };
49
50 } // namespace os
51
52 // ----- 单词扫描 -----
53 static bool isword(char c) noexcept {
54     return std::isalnum(static_cast<unsigned char>(c)) || c == '_';
55 }
56
57 // 扫描一段连续内存, 返回局部表
58 static Freq scan_block(const char* first, const char* last) {
59     Freq local;
60     std::string w;
61     w.reserve(64);
62     for (auto it = first; it != last; ++it) {
63         char c = *it;
64         if (isword(c)) {
65             w.push_back(static_cast<char>(std::tolower(

```

```

66         static_cast<unsigned char>(c))));
67     } else if (!w.empty()) {
68         ++local[w];
69         w.clear();
70     }
71 }
72 if (!w.empty()) ++local[w];
73 return local;
74 }
75
76 // ----- 并行框架 -----
77 static Freq count_region(const char* first, const char* last,
78                          unsigned threads) {
79     if (threads == 0) threads = 1;
80     const std::size_t chunk = (last - first) / threads;
81     std::vector<std::future<Freq>> futures;
82     futures.reserve(threads);
83
84     for (unsigned i = 0; i < threads; ++i) {
85         const char* beg = first + i * chunk;
86         const char* end = (i + 1 == threads) ? last : beg + chunk;
87
88         // 调整 beg 到单词边界
89         if (i && beg < last && isword(beg[-1]) && isword(beg[0]))
90             while (beg != end && isword(*beg)) ++beg;
91
92         futures.push_back(std::async(std::launch::async,
93                                     [beg, end] { return scan_block(beg, end);
94                                     ↵ }));
95     }
96
97     // 归并所有局部表
98     Freq global;
99     for (auto& f : futures)
100         for (auto& [w, c] : f.get())
101             global[w] += c;
102     return global;
103 }
104
105 // ----- 对外接口 -----
106 Freq count_file(const std::string& path) {
107     os::mapped_region file(path);
108     unsigned cores = std::thread::hardware_concurrency();
109     return count_region(file.data(), file.data() + file.size(), cores);
110 }

```

主程序 main.cpp

```

1 #include "wc.hpp"
2 #include <algorithm>
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 #include <vector>
7
8 int main(int argc, char* argv[]) {
9     if (argc != 2) {
10         std::cerr << "Usage: " << argv[0] << " <utf8-en.txt>\n";

```



```

11     return 1;
12 }
13
14 // 1. 统计
15 const auto freq = count_file(argv[1]);
16
17 // 2. 直接把 map 转成 vector<string_view> 避免拷贝 key
18 using entry = std::pair<std::size_t, std::string_view>;
19 std::vector<entry> top;
20 top.reserve(freq.size());
21 for (const auto& [w, c] : freq) top.emplace_back(c, w);
22
23 // 3. 取 Top-10 (nth_element + sort 前 10 更快)
24 const std::size_t k = 10;
25 if (top.size() > k) {
26     std::nth_element(top.begin(), top.begin() + k, top.end(),
27                     std::greater<>{});
28     top.resize(k);
29 }
30 std::sort(top.begin(), top.end(), std::greater<>{});
31
32 // 4. 输出
33 for (const auto& [c, w] : top)
34     std::cout << std::setw(10) << w << " " << c << '\n';
35 }

```

CMake:

```

1 cmake_minimum_required(VERSION 3.10)
2 project(wordcount LANGUAGES CXX)
3
4 set(CMAKE_CXX_STANDARD 17)
5 set(CMAKE_CXX_STANDARD_REQUIRED ON)
6 set(CMAKE_CXX_EXTENSIONS OFF)
7
8 add_executable(wordcount
9     src/main.cpp
10    src/wc.cpp
11 )
12 target_include_directories(wordcount PRIVATE include)
13 # 需要 pthread
14 find_package(Threads REQUIRED)
15 target_link_libraries(wordcount PRIVATE Threads::Threads)

```

编译运行:

```

1 mkdir build
2 cd build
3 cmake ..
4 cmake --build . -j --config Release # Release 模式编译, 开启优化
5 ./word_count ../bigfile.txt

```

29.6.4 线程池

上述代码中用的是 `std::async` 来启动线程并行处理任务。这样做的好处是简单易用，但缺点是每次调用都会创建一个新的线程，频繁创建销毁的开销也不小。对此，人们提出了“线程池”（Thread Pool）这个概念。线程池指的是预先创建一组线程，并将它们放入一个池子中。当有任务需要处理时，就从线程池中取出一个空闲的线程来执行任务，任务完成后再将线程放回池子中。这样就能避免频繁创建销毁线程的开销，提高程序的性能。

实现线程池的方式有很多种，下面是一个简单的线程池实现示例：

```

1 #include <condition_variable>
2 #include <functional>
3 #include <future>
4 #include <mutex>
5 #include <queue>
6 #include <thread>
7 #include <vector>
8 class ThreadPool {
9 public:
10     ThreadPool(std::size_t n) : stop(false) {
11         for (std::size_t i = 0; i < n; ++i) {
12             workers.emplace_back([this] {
13                 while (true) {
14                     std::function<void()> task;
15                     {
16                         std::unique_lock<std::mutex> lock(this->mtx);
17                         this->cv.wait(lock, [this] { return this->stop ||
18                             ↪ !this->tasks.empty(); });
19                         if (this->stop && this->tasks.empty()) return;
20                         task = std::move(this->tasks.front());
21                         this->tasks.pop();
22                     }
23                     task();
24                 }
25             });
26         }
27     }
28     ~ThreadPool() {
29         {
30             std::unique_lock<std::mutex> lock(mtx);
31             stop = true;
32         }
33         cv.notify_all();
34         for (std::thread &worker : workers) worker.join();
35     }
36     template<class F, class... Args>
37     auto enqueue(F&& f, Args&&... args)
38     -> std::future<typename std::result_of<F(Args...)>::type> {
39         using return_type = typename std::result_of<F(Args...)>::type;
40         auto task = std::make_shared<std::packaged_task<return_type()>>(
41             std::bind(std::forward<F>(f), std::forward<Args>(args)...)
42         );
43         std::future<return_type> res = task->get_future();
44         {
45             std::unique_lock<std::mutex> lock(mtx);
46             if (stop) throw std::runtime_error("enqueue on stopped ThreadPool");
47             tasks.emplace([task] { (*task)(); });
48         }
49     }
50 }

```

```

47     }
48     cv.notify_one();
49     return res;
50 }
51 private:
52     std::vector<std::thread> workers;
53     std::queue<std::function<void()>> tasks;
54     std::mutex mtx;
55     std::condition_variable cv;
56     bool stop;
57 };

```

使用这个线程池:

```

1 ThreadPool pool(hardware_threads());
2 std::vector<std::future<wc::CountMap>> futures;
3 for (const auto& r : ranges)
4     futures.push_back(pool.enqueue(wc::count_range, file, r));

```

线程池虽然有效地提升了性能，但也增加了代码的复杂性。

29.6.5 怎样调试?

多线程程序的调试往往比单线程程序复杂得多。

首先，我们应确认是“多线程”本身的问题，还是其他的逻辑问题。例如上述代码中，如果结果不正确，首先应检查文件分割和单块统计的逻辑是否正确，而不是直接怀疑多线程部分。一个简单的检查方式就是多次运行程序，看看结果是否一致。如果结果不一致，那么很可能是多线程部分的问题；如果结果一致但不正确，那么问题可能出在其他地方。

当定位问题确实在多线程部分时，可以尝试以下工具：

- **ThreadSanitizer**: 这是一个动态检测工具，可以帮助我们检测多线程程序中的数据竞争和死锁等问题。它通过在编译时插入额外的检查代码，在运行时监控线程之间的内存访问，从而发现潜在的并发问题。使用该工具需要在编译时添加相应的编译选项，例如对于 GCC 和 Clang，可以使用 `-fsanitize=thread` 选项。
- **Clang-Tidy**: 这是一个静态代码分析工具，可以帮助我们发现多线程程序中的潜在问题。它通过分析源代码，检查线程安全性、锁的使用等方面的问题，从而提高代码的质量和可靠性。使用该工具需要安装 Clang，并运行相应的命令行工具。
- **Valgrind-Helgrind**: 这是 Valgrind 工具集中的一个子工具，专门用于检测多线程程序中的数据竞争和死锁等问题。它通过在运行时监控线程之间的内存访问，发现潜在的并发问题。使用该工具需要安装 Valgrind，并运行相应的命令行工具。

另，也可以用可视化方式调试，例如 `chrome://tracing/`，可以把程序运行时的线程调度信息导出来，然后用这个工具查看线程的执行情况，从而发现潜在的问题。或者 Intel VTune、Windows Concurrency Visualizer 等工具，也能帮助我们分析多线程程序的性能瓶颈和并发问题。实在不行，用 `std::this_thread::get_id()` 打印线程 ID，结合日志分析也是一种可行的办法。

29.6.6 什么时候不能用多线程？

多线程确实是提升性能的有效手段，但并不是所有情况下都适合使用多线程。因为多线程的代码写起来实在是太复杂了，调试起来更是头疼。所以，在决定使用多线程之前，我们需要评估一下任务的性质和性能瓶颈，看看多线程是否真的能带来显著的性能提升。

为了衡量性能瓶颈，人们人为地把任务分成两类：**CPU 密集型**和**I/O 密集型**。前者指的是任务主要消耗 CPU 资源，例如复杂的计算、数据处理等；后者指的是任务主要消耗 I/O 资源，例如文件读写、网络通信等。

要分析一个任务究竟是 CPU 密集型还是 I/O 密集型，可以这样做：首先，把任务拆成两段，也就是“计算”和“I/O”两部分。然后，分别测量这两部分的执行时间。如果计算部分的时间远大于 I/O 部分的时间，那么这个任务就是 CPU 密集型；反之，如果 I/O 部分的时间远大于计算部分的时间，那么这个任务就是 I/O 密集型。而在不同的机器上，这个比例可能会有所不同：机械硬盘、SATA 协议的 SSD、NVMe 协议的 SSD 和内存盘，它们的带宽和延迟差别巨大，越靠后的设备，任务越倾向于 CPU 密集型。

对于 CPU 密集型任务，多线程通常能显著提升性能，因为它能充分利用多核 CPU 的计算能力，实现并行处理，从而缩短任务的执行时间。而对于 I/O 密集型任务，多线程的效果可能并不明显，甚至可能带来额外的开销。因为 I/O 操作通常是阻塞的，线程在等待 I/O 完成时会被挂起，这会导致线程切换的开销增加，从而降低整体性能。

上述任务中，如果我们确实能一口气把所有文件读入内存，那它就是一个典型的 CPU 密集型任务，多线程能显著提升性能；但如果使用的是机械硬盘，那还不如单线程。关于上述问题，机械硬盘选择单线程，SATA 可以“轻量多线程”（也就是说线程数在 2 到 4 之间），NVMe 和内存盘则可以使用“重度多线程”（线程数等于 CPU 物理核心数）。而现在已经 2025 年，NVMe 硬盘随处可见，大家大可放心大胆地使用多线程。

另外，上述代码其实相当朴素，仅奔着“实现”去，没有经过严格的性能调优。实际上，在 NVMe 或内存场景下，用更高效的并行归并算法也能提升合并性能；用更高效的哈希表实现（例如 Google 的 `dense_hash_map`）也能提升性能；用线程池也能减少线程创建销毁的开销；用更低级的内存顺序模型也能提升原子操作的性能……这些都是可以考虑的优化方向。

需要注意的是 CPU 物理核心数和逻辑核心数是不一样的：物理核心数是 CPU 实际拥有的核心数量，而逻辑核心数则是通过超线程技术（Hyper-Threading）实现的、操作系统看到的逻辑核心数量。一般来说，逻辑核心数是物理核心数的两倍。例如，一个四核八线程的 CPU，其物理核心数为 4，逻辑核心数为 8。

对内存带宽敏感的任务，超线程（线程数量大于物理核心数）反而会降低性能，因为多个线程争抢同一个核心的资源，导致缓存命中率下降和上下文切换增加，从而降低整体性能。因此，在选择线程数时，建议优先考虑线程数等于物理核心数，然后再根据实际情况进行微调。而当上述文件极大（TB 级别）时，TLB Miss 也会成为瓶颈，这时可以考虑 NUMA 架构下的内存亲和性（Memory Affinity）等高级优化手段。

29.7 扩展阅读

由于多线程编程的复杂性和多样性，建议同学们进一步阅读相关书籍和文档，以深入理解多线程编程的原理和实践。

我个人推荐 `THE ART OF MULTIPROCESSOR PROGRAMMING` 一书，作者 Maurice Herlihy 和 Nir Shavit 是并发编程领域的权威专家。这本书深入探讨了多处理器编程的基本原理和技术，涵盖了锁、无锁数据结构、事务内存等主题。书中不仅介绍了理论知识，还提供了大量的实践案例和代码示例，帮助读者理解如何在实际应用中实现高效且正确的多线程程序。

第三十章 神经网络和机器学习

目前人工智能算是一个大潮流了：从今年年初的 DeepSeek-R1 到最近的 GPT-5、Sora2 等一众高性能的人工智能工具，在多个领域取得了令人瞩目的成果。而为了理解这些工具的工作原理，我们要把目光投向神经网络和机器学习：这是当今人工智能工具的祖先，也是直到现在的核心。

说明

本章内容完全基于笔者个人在学习和科研中的理解进行总结，难免有疏漏和错误或者跳跃之处，敬请谅解。如有兴趣深入学习，建议参考相关教材和文献。

另外，本章节是很“数学”的，建议读者先学会高等数学和线性代数的一些内容，再来阅读本章内容会更好。

30.1 从最简单的神经元说起

30.1.1 0-1 二分类问题

假设我们现在有一百万张图片，每张图片上面都是一个手写数字，要么是 0、要么是 1。这些图片的大小都是 28x28 像素，每个像素是黑白两色（0 表示白色，1 表示黑色）。现在希望把这些图片分开。

一个最简单的手段就是：找一群无所事事的大学生，让他们每人看一堆图片并分类，按 0 和 1 分两堆。这样就能把图片分开了。但是这样显然是很低效的。有没有一种自动化的手段呢？

我们试着变换一下思路：假设每一个像素点都是一个变量 x_i ，那么每一个图片就可以表示为一个向量 $\mathbf{x} = (x_1, x_2, \dots, x_{784})$ ，其中 $784 = 28 \times 28$ 。那么，我们会在 784 维空间中得到一百万个点，每个点对应一张图片。现在似乎依然没有解决问题。

但是，如果 0 和 1 的写法确实是有区别的，那么这一堆点在直觉上应该显然是呆在这个 784 维空间的不同区域的。这也是神经网络的基本假设：不同类别的数据在高维空间中是可以被划分开的。那么，我们确实可以试着找一个“超平面”，或“线”，把这些点分开（或大多数点分开）。换言之：

$$0 = \mathbf{W}^T \mathbf{x} + b$$

显然这是一个超平面方程；只要试图找到对于所有类别为 0 的点，使得 $f(\mathbf{x}) < 0$ ，而对于类别为 1 的点，使得 $f(\mathbf{x}) > 0$ 即可。

提示

如果还是不能理解，可以考虑二维空间中的例子：假设我们有一堆点在平面上，有些点是红色的，有些点是蓝色的。我们希望找到一条直线，把红色的点和蓝色的点分开。这个直线的方程可以写成：

$$0 = W_1x_1 + W_2x_2 + b$$

高中应该都学过解析几何，我们可以轻易地看出这是一个直线方程。而我们希望对于所有红色的点，使得 $f(\mathbf{x}) < 0$ ，而对于蓝色的点，使得 $f(\mathbf{x}) > 0$ 。这就是一个简单的二分类问题。

但是上述方程比较臃肿，因此我们写出两个向量： $\mathbf{W} = (W_1, W_2)$ 和 $\mathbf{x} = (x_1, x_2)$ ，那么上述方程就可以简化为：

$$0 = \mathbf{W}^T \mathbf{x} + b$$

学习过线性代数的同学应该能看出来这个方程和上述直线的“一般式”方程是等价的。因而，我们可以把这个例子推广到更高维的空间中去。

如还是不能理解，可以思考三维空间内怎么表示一个平面对程。

那么我们的任务就变成了：**找到一组权重 $\mathbf{W}$ 和偏置 b ，使得这个超平面能把大多数点分开**。而我们上述提出的这个方程，实际上就是最简单的神经元——线性神经元的工作原理。

30.1.2 怎么求解？

现在的问题是：我们怎么找到这组权重 $\mathbf{W}$ 和偏置 b 呢？ $\mathbb{R}$ 是一个连续的空间，暴力枚举根本行不通。但问题可不是仅凭“瞪眼”就能瞪出来的：要过河，总得先下水再研究怎么过去。

于是我们随机生成了一组权重 $\mathbf{W}$ 和偏置 b ，并计算了所有点的 $f(\mathbf{x})$ 。显然大概率不可能一下就分类对了，肯定有大量是分错了的。这时候，我们就“摸着石头过河”，让这个超平面朝着正确的方向“扭一点”，让它的分类结果比先前更好一点，这就是“学习”的过程。

那有学习就得有评价，怎么评价这个学习是好的还是坏的呢？我们需要进行一些“打分”：对于每一个点，我们都知道它的真实类别 y ，而我们也可以通过 $f(\mathbf{x})$ 来预测它的类别 $\hat{y}$ 。那么，我们就可以定义一个损失函数 $L(y, \hat{y})$ ，来衡量预测值和真实值之间的差距。比如说，最简单的 0-1 损失函数：

$$L(y, \hat{y}) = \begin{cases} 0, & \text{如果 } y = \hat{y} \\ 1, & \text{如果 } y \neq \hat{y} \end{cases}$$

这个损失函数的意思是：如果预测正确，损失为 0；如果预测错误，损失为 1。然后，我们可以计算所有点的总损失：

$$J(\mathbf{W}, b) = \sum_{i=1}^N L(y_i, \hat{y}_i)$$

其中 N 是点的总数。我们的目标就是最小化这个总损失 $J(\mathbf{W}, b)$ ，通过调整 $\mathbf{W}$ 和 b 来实现更好的分类效果，只需要让这个损失函数的值成为全局最小¹即可。

我们知道，为了找到一个函数的最小值，可以对它求导，然后让导数为 0，解出变量的值。但是上述损失函数显然不是一个性质很好的函数，无法直接求导。于是，我们引入了一个更平

¹可能不是 0，因为数据本身可能有噪声或者不可分的情况

滑的损失函数，比如说均方误差 (MSE)：

$$L(y, \hat{y}) = (y - \hat{y})^2$$

这样，我们就可以对总损失 $J(\mathbf{W}, b)$ 求导，得到梯度：

$$\nabla J(\mathbf{W}, b) = \left(\frac{\partial J}{\partial \mathbf{W}}, \frac{\partial J}{\partial b} \right)$$

但是就算这个函数能直接求导也不是很容易求出解析解的，这玩意维数可太高了。问题似乎又遭遇到了瓶颈。于是我们被迫寄希望于数学家，然后发现牛顿已经在几百年前发明了一种求解方程的办法：牛顿迭代法。

回忆一下这个方法的基本路径：求方程 $f(x) = 0$ 的解。我们从一个初始猜测 x_0 出发，计算函数值 $f(x_0)$ 和导数 $f'(x_0)$ ，然后用切线来近似函数在这个点附近的行为。切线的方程是：

$$y = f(x_0) + f'(x_0)(x - x_0)$$

我们希望找到切线与 x 轴的交点，这个交点给出了一个新的猜测 x_1 ：

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

然后，我们重复这个过程，直到收敛到一个解。

而我们的上述问题是为了求解导数等于零，所以可以把上述问题重新写成

$$x_{n+1} = x_n - [\nabla^2 J(x_n)]^{-1} \nabla J(x_n)$$

这是牛顿迭代法在多维空间优化问题的推广形式，其中 $\nabla^2 J(x_n)$ 是损失函数的海森矩阵 (Hessian matrix)，表示二阶导数信息。这个方法就是大名鼎鼎的**牛顿优化法**，它可以帮助我们找到损失函数的极小值，从而优化神经元的权重和偏置。

然后问题又来了：仅上述一个最简单的题，一共有 784 个权重和 1 个偏置，一共 785 个变量。计算海森矩阵的开销是非常大的，尤其是当数据量很大时，计算梯度和海森矩阵的代价会变得非常高昂，算海森矩阵的逆更是难上加难。

于是我们被迫重新回到牛顿优化法的最终结论，然后突然发现：**其实我们并不需要精确地计算海森矩阵的逆**，只要知道梯度的方向就行了。于是，我们可以简化更新公式为：

$$x_{n+1} = x_n - \eta \nabla J(x_n)$$

其中 η 是一个小的正数，称为**学习率**。而上式便是更加大名鼎鼎的**梯度下降法**的更新公式。虽然上述方法显然是没有牛顿法快的，但是计算量可是小多了！这里的 x 可以替换为任何变量，比如说我们的权重 $\mathbf{W}$ 和偏置 b 。

由此，我们直接看出学习率的大小会对这个收敛的速度和效果产生什么影响：学习率调得大，每次更新的步伐就大，收敛快但可能会错过最优解；学习率调得小，每次更新的步伐就小，收敛慢但更稳定。较为古典的机器学习是把该数值设定成一个较小的常数，而现代的深度学习的则会动态调整这个数值，比如说使用 Adam 优化器等。

上述算法的一个新的问题是：我们每次都要计算所有点的梯度，这样计算量还是很大。于是，我们引入了**随机梯度下降法** (SGD)：每次只随机抽取一小部分数据 (称为一个 mini-batch)，计算这个 mini-batch 的梯度，然后更新权重和偏置。这样，虽然每次更新的方向可能不完全准确，但整体上仍然朝着最小化损失函数的方向前进，而且计算效率大大提高。

于是我们得以对最初的问题进行解决：写一个类似的程序，然后先抽取 10000 张给几个大学生分类 (打标签)，然后用这些数据分成两组，一组喂给神经元进行训练，另一组则用来测试这个神经元“确实在学会分类”。经过训练，我们会得到一个模型 (实际上就是一个权重向量和一个偏置)，然后我们就可以用这个模型对剩下的 99 万个图片进行分类。这样，我们就实现了一个简单的机器学习任务。

提示

有的同学也可能会思考：为什么不画一个曲面来分开这些点呢？答案是：理论上是可以的。不过，我们还是先看看二维平面上的二次曲线方程：

$$0 = Ax_1^2 + Bx_2^2 + Cx_1x_2 + Dx_1 + Ex_2 + F$$

这个方程的系数一共有 6 个。三维空间中的二次曲面方程的系数是 $6 + 3 + 1 = 9$ 个。而 784 维空间中的二次超曲面方程的系数数量是一个非常大的数字，计算和存储的开销会非常大。

另一方面，这个东西如果梯度下降，那么损失函数的形状会非常复杂，可能会有很多局部极小值，导致优化过程变得非常困难。因此，虽然理论上可以使用更复杂的模型来拟合数据，但在实际应用中，简单的线性模型反而更容易训练和优化。

30.2 多层神经网络

30.2.1 新的问题：MNIST-10

现在的问题又来了。假设我们现在有一百万个图片，每张图片上面都是一个手写数字，可能是 0 到 9 中的任意一个数字。这些图片的大小都是 28x28 像素，每个像素是黑白两色 (0 表示白色，1 表示黑色)。现在希望把这些图片分成 10 类。

我们刚刚解决了上述 0-1 分类问题，已经形成路径依赖的我们直接进行一个向量化，把这一百万个图片表示为一个 784 维空间中的一百万个点。然后，我们试图找一个超平面把这些点分开。但是问题来了：**我们现在有 10 个类别，而一个超平面只能把空间分成两部分**。这显然是不够的。

或许有的人会思考：直接弄一大堆超平面不就行了？比如说，弄 9 个超平面，每个超平面负责把一个类别和其他类别分开。这样就能把 10 个类别分开了。理论上这是可行的，但是问题是：**这些超平面之间是相互独立的**，它们并没有协同工作来优化整体的分类效果。这样一来，整体的分类效果可能并不好，所以路径依赖肯定是有问题的，我们需要新的思路。一个容易想

到的思路是把独立求解的多个方程合并成一个方程组，也就是：

$$\begin{cases} \mathbf{W}_0^T \mathbf{x} + b_0 = 0 \\ \mathbf{W}_1^T \mathbf{x} + b_1 = 0 \\ \mathbf{W}_2^T \mathbf{x} + b_2 = 0 \\ \vdots \\ \mathbf{W}_9^T \mathbf{x} + b_9 = 0 \end{cases}$$

这个还是太难看了，我们利用一些线性代数的知识，把它写成矩阵形式：

$$\mathbf{W}^T \mathbf{x} + \mathbf{b} = \mathbf{0}$$

上述 $\mathbf{b}$ 和 $\mathbf{0}$ 是两个 10 维向量，而 $\mathbf{W}$ 是一个 784x10 的矩阵。这个方程的意义是：对于一个类别 i ，我们都有一个对应的线性函数：

$$f_i(\mathbf{x}) = \mathbf{W}_i^T \mathbf{x} + \mathbf{b}_i$$

其中 $\mathbf{W}_i$ 是矩阵 $\mathbf{W}$ 的第 i 列， $\mathbf{b}_i$ 是向量 $\mathbf{b}$ 的第 i 个元素。然后，我们可以通过计算所有类别的函数值，来预测图片的类别：

$$\hat{y} = \arg \max_i f_i(\mathbf{x})$$

也就是说，我们选择函数值最大的类别作为预测结果。

实际上，只要我们懂一点线性代数，我们就能把上述二分类问题的方程改写成这个方程。在上述二分类问题中，我们有一个权重向量 $\mathbf{w}$ 和一个偏置标量 b ，而在这个多分类问题中，我们有一个权重矩阵 $\mathbf{W}$ 和一个偏置向量 $\mathbf{b}$ 。这里的权重矩阵 $\mathbf{W}$ 的每一列对应一个类别的权重向量，而偏置向量 $\mathbf{b}$ 的每个元素对应一个类别的偏置。换句话说，这玩意实际上是把 10 个二分类器“拼”成了一个多分类器，但这个方法和上述“分成 9 个独立超平面”的方法不同：这里的权重矩阵和偏置向量是一起优化的，从而实现协同工作。

那剩下的问题直接进行一个路径依赖：定义损失函数，寻找损失函数最小值；在这个过程中使用梯度下降来优化权重矩阵和偏置向量。这里的损失函数可以使用交叉熵损失函数（Cross-Entropy Loss）：

$$L(y, \hat{y}) = - \sum_{i=1}^C y_i \log(\hat{y}_i)$$

其中 C 是类别数， y_i 是真实类别的独热编码（one-hot encoding）， $\hat{y}_i$ 是预测类别的概率。通过最小化这个损失函数，我们可以优化模型的分类性能。

30.2.2 多层神经网络和激活函数

但是新的问题接踵而至：上述模型只有一层，本质上依然是个“超级线性分类器”，哪怕我们用了一大堆花里胡哨的东西把它包装起来，最终本质上还是在 784 维空间画超平面。这就带来一个非常尴尬的事实：有的事情根本不是画几条线就能搞定的，如果数据本身并不是线性可分的，那么无论我们怎么优化这个模型，最终的分类效果都不会很好。就像同样是写一个字，

对单层模型而言它只会“数数”，看到这些格子黑、那些格子白，然后就知道这是个4；但一旦这些像素格子换了个顺序、转个圈、加点噪声，这个模型就完全懵了。

这时候，多层神经网络的价值就体现出来了：它不是一上来就“划线”，而是先让前几层把原始像素“捏”成更高级的“抽象特征”，先让数据在隐藏空间里变得“更容易分”，最后再画直线——这就从“根本分不开”变成了“稍微努努力就能分开”。所以，不是多层更“酷”，而是没有多层就“完不成任务”。单层模型在 MNIST-10 上也许能混个 80% 准确率，看起来“还行”，但只要你把数字稍微倾斜、加点噪点、笔画连笔，它立刻崩给你看。而深层的意义就在于：它自己学会了一套“预处理语言”，把原始像素翻译成“形状”，再翻译成“数字”——这是单层模型永远做不到的。

但是，直接将两个线性变换堆在一起，又会发生什么呢？我们试着对以下两个线性变换进行叠加：

$$\begin{cases} f_1(\mathbf{x}) = \mathbf{W}_1^T \mathbf{x} + \mathbf{b}_1 \\ f_2(\mathbf{y}) = \mathbf{W}_2^T \mathbf{y} + \mathbf{b}_2 \end{cases} \Rightarrow f(\mathbf{x}) = f_2(f_1(\mathbf{x})) = \mathbf{W}_2^T (\mathbf{W}_1^T \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2^T \mathbf{W}_1^T) \mathbf{x} + (\mathbf{W}_2^T \mathbf{b}_1 + \mathbf{b}_2)$$

我们惊奇的发现这依然是一个线性变换！继续推广，我们得到一个令人失望的结论：**无论我们堆叠多少层线性变换，最终的结果仍然是一个线性变换**，就像对一个图像进行多次旋转和平移，最终仍然可以用一个旋转和平移来表示，单纯的多层线性分类器永远不会比单层线性分类器更强大。

为了解决这个问题，我们为什么不直接把“划线”从直线变成曲线呢？但是这个问题已经讨论过并被否定了。于是我们想了想，既然目的是“引入非线性”，还不能画曲线，那我为什么不在每一层的线性变化之后“加点非线性”呢？这样一来，整体上就不是线性的了。

因此，我们可以在每一层的线性变换之后，添加一个非线性函数，称为**激活函数** (Activation Function)。这样，每一层就不再是一个简单的线性组合，而是：

$$\begin{aligned} \mathbf{z}^{(l)} &= \mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \\ \mathbf{a}^{(l)} &= \sigma(\mathbf{z}^{(l)}) \end{aligned}$$

不要小看这个 σ ，它可是神经网络的灵魂所在，是“能画曲线”的关键：让网络真正有了非线性表达能力、叠出了复杂性，而不是做无用功。激活函数让原本线性不可分的数据在隐藏空间被逐步扭曲、拉伸、压缩，最终变得线性可分。

最常见的老将是 Sigmoid 函数：

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

它能把输入映射到 0 到 1 之间，适合二分类任务。但这东西有个大问题：当输入值很大或很小时，梯度会变得非常小，导致**梯度消失**问题，使得网络难以训练。它的另一个问题是，输出不是中心化的（即均值不为 0），这会影响后续层的训练效率。于是还有一种改进版的 Tanh 函数：

$$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \frac{d\sigma}{dx} = 1 - \sigma(x)^2$$

它把输入映射到 -1 到 1 之间，中心化效果更好，但同样存在梯度消失的问题。

另一位常见的战士是 ReLU 函数（Rectified Linear Unit）和他的兄弟 Leaky ReLU：

$$\sigma(x) = \max(0, x)$$

$$\sigma(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & \text{if } x \leq 0 \end{cases}$$

它在正区间保持线性，在负区间输出 0 或接近 0，计算简单且能有效缓解梯度消失问题，但可能导致“神经元死亡”，即某些神经元永远不激活。

于是最近的工作有 GELUs、Swish 等，都是在尝试找到更好的激活函数，以提升神经网络的性能和训练稳定性。

30.2.3 正向传播和反向传播

于是我们现在得到了一个多层神经网络模型。我们就考虑一个最简单的三层模型：输入层-sigmoid-隐藏层-sigmoid-输出层。假设输入层有 784 个神经元，隐藏层有 128 个神经元，输出层有 10 个神经元。那么，我们可以定义以下变量：

- 输入层的激活值： $\mathbf{a}^{(0)} \in \mathbb{R}^{784}$
- 隐藏层的权重矩阵： $\mathbf{W}^{(1)} \in \mathbb{R}^{128 \times 784}$
- 隐藏层的偏置向量： $\mathbf{b}^{(1)} \in \mathbb{R}^{128}$
- 隐藏层的激活值： $\mathbf{a}^{(1)} \in \mathbb{R}^{128}$
- 输出层的权重矩阵： $\mathbf{W}^{(2)} \in \mathbb{R}^{10 \times 128}$
- 输出层的偏置向量： $\mathbf{b}^{(2)} \in \mathbb{R}^{10}$
- 输出层的激活值： $\mathbf{a}^{(2)} \in \mathbb{R}^{10}$

这玩意也太多了。我们要怎么才能把这些东西联系起来呢？容易想到只需要一步步来就可以了，因此，我们从输入层开始，进行**正向传播**（Forward Propagation）：

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}$$

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)})$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$$

这样就最终计算出了输出层的激活值 $\mathbf{a}^{(2)}$ ，也就是模型的预测结果。

当然，仅正向传播是仅能做到“预测”，或“应用”的，是“用理论指导实践”。我们还需要“学习”、需要“反思”，需要“用实践修正理论”，优化权重和偏置，使得模型的预测结果更准确。为此，我们需要计算损失函数，并通过**反向传播**（Backward Propagation）来计算梯度，从而更新权重和偏置。

刚刚提到，为了让学习可以量化，我们需要定义一个损失函数。对于多分类问题，常用的损失函数是交叉熵损失函数：

$$L(y, \hat{y}) = - \sum_{i=1}^C y_i \log(\hat{y}_i)$$

其中 C 是类别数, y_i 是真实类别的独热编码 (one-hot encoding), $\hat{y}_i$ 是预测类别的概率。

那么反向传播就是这样的。首先, 对于一个线性层, 我们有 $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, 那么损失函数对权重矩阵和偏置向量的梯度可以通过链式法则计算出来:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}} &= \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \mathbf{x}^T \\ \frac{\partial L}{\partial \mathbf{b}} &= \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{y}} \\ \frac{\partial L}{\partial \mathbf{x}} &= \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W}^T \cdot \frac{\partial L}{\partial \mathbf{y}}\end{aligned}$$

这里为什么要计算 $\mathbf{x}$ 呢? 因为我们要把梯度传递给前一层, 以便继续计算。如果仅有单层那么就不需要了。我们这样计算是为了统一反向传播的过程, 使其都能归结于 $\partial L / \partial \mathbf{y}$ 的计算。

把这三个东西带入梯度下降的更新公式, 我们就能更新权重矩阵和偏置向量:

$$\begin{aligned}\mathbf{W}_{n+1} &= \mathbf{W}_n - \eta \frac{\partial L}{\partial \mathbf{W}} \\ \mathbf{b}_{n+1} &= \mathbf{b}_n - \eta \frac{\partial L}{\partial \mathbf{b}}\end{aligned}$$

这里的 η 是学习率。这个就是通用的反向传播公式, 可以应用于任何线性层。

那现在的问题就变成怎么计算 $\partial L / \partial \mathbf{y}$ 了。对于输出层, 我们可以直接计算这个梯度, 因为我们有损失函数和预测结果。而对于隐藏层, 我们需要通过链式法则, 把输出层的梯度传递回来:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}^{(2)}} &= \frac{\partial L}{\partial \mathbf{a}^{(2)}} \cdot \sigma'(\mathbf{z}^{(2)}) \\ \frac{\partial L}{\partial \mathbf{a}^{(1)}} &= (\mathbf{W}^{(2)})^T \cdot \frac{\partial L}{\partial \mathbf{z}^{(2)}} \\ \frac{\partial L}{\partial \mathbf{z}^{(1)}} &= \frac{\partial L}{\partial \mathbf{a}^{(1)}} \cdot \sigma'(\mathbf{z}^{(1)})\end{aligned}$$

这里的 σ' 是激活函数的导数。既然隐藏层算了这三个东西, 那么就可以再次带回上一个线性层, 继续计算梯度, 直到传递到输入层为止。

看起来很复杂, 但实际上就是不断重复上述过程: 计算每一层的梯度, 然后传递给前一层。这样, 我们就能通过反向传播算法, 计算出所有权重矩阵和偏置向量的梯度, 从而更新它们, 使得模型的预测结果更准确。这也是理论和实践相互指导、相互促进的过程。

30.2.4 正则化

我们终于写好了一个神经网络, 可以进行分类任务了。但是, 新的问题又来了: 假设我们训练好了这个模型, 在训练集上达到了 99% 的准确率, 但是在测试集上只有 80%。这显然是不行的, 我们希望模型在未见过的数据上也能有良好的表现。

在训练的时候, 我们可能会发现以下四种情况:

- 模型在训练集和测试集上的准确率都随着训练的继续不断提高, 且两者有一定的差距, 但差距不大。这种情况说明模型在学习, 并且有一定的泛化能力。

- 模型在训练集上的准确率不断提高，但在测试集上的准确率开始下降，说明模型过拟合了训练数据，失去了泛化能力。
- 模型在训练集和测试集上的准确率都很低，说明模型欠拟合，无法捕捉数据的复杂性。
- 模型在训练集上的准确率很高，但在测试集上的准确率也很高，说明数据太少了，应该增加数据量。

上述的第二项就是我们要解决的问题：**过拟合**。过拟合是指模型在训练数据上表现很好，但在未见过的数据上表现很差。

一个非常常见的笑话：在不限制规律复杂度的情况下，任何找规律题目的答案都可以是 42，因为 42 是宇宙的终极答案因为拉格朗日插值多项式可以完美拟合任何有限数据集，或者说总能找到一个 k 次多项式 $f(x)$ ，使得 $f(x+1) = 42$ 。但是这样的高阶多项式会剧烈震荡，在训练数据点之间的区域表现很差，且系数往往也非常大，导致数值不稳定。

为了防止模型真去插值（实际上这反而是非常常见的现象），我们需要引入一些**正则化**（Regularization）技术，来限制模型的复杂度，从而提高其泛化能力。

最常见的正则化有两种：L2 正则化（Ridge Regression）和 L1 正则化（Lasso Regression）。L2 正则化通过在损失函数中添加权重的平方和来惩罚大权重：

$$J(\mathbf{W}, b) = \sum_{i=1}^N L(y_i, \hat{y}_i) + \lambda \|\mathbf{W}\|_2^2$$

其中 λ 是正则化参数，控制惩罚的强度，该正则化鼓励权重趋近于零，但不会完全为零，从而防止过拟合。

L1 正则化则通过添加权重的绝对值和来实现：

$$J(\mathbf{W}, b) = \sum_{i=1}^N L(y_i, \hat{y}_i) + \lambda \|\mathbf{W}\|_1$$

L1 正则化有助于产生稀疏的权重矩阵，从而实现特征选择。

另一个常用的正则化技术是 **Dropout**，它通过在训练过程中随机丢弃一部分神经元来防止过拟合。具体来说，在每次训练迭代中，以一定概率 p 将一些神经元的输出设为零，从而迫使网络学习更加鲁棒的特征表示。

正则化往往会导致测试集误差降低，但训练集误差增加。这是因为正则化限制了模型的复杂度，使其无法完全拟合训练数据，但提高了其在未见数据上的表现。

30.2.5 归一化

在训练神经网络时，输入数据的分布对模型的训练速度和效果有很大的影响。如果输入数据的各个特征具有不同的尺度，模型可能会更难收敛，甚至陷入局部最优解，产生诸如梯度爆炸、梯度消失、学习率难以调整等问题。

为了缓解这些问题，我们通常会对输入数据进行**归一化**（Normalization）处理，使得各个特征具有相似的尺度。

批量归一化 批量归一化（Batch Normalization）是一种在训练过程中对每个小批量数据进行归一化的方法。具体来说，对于一个小批量数据，我们计算每个特征的均值和标准差，然后使用这些统计量对数据进行归一化：

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

其中 μ_B 和 σ_B^2 分别是小批量数据的均值和方差， ϵ 是一个很小的常数，防止除零错误。归一化后的数据再通过一个线性变换进行缩放和平移：

$$y_i = \gamma \hat{x}_i + \beta$$

其中 γ 和 β 是可学习的参数。

换句话说，对于线性层，批量归一化的过程可以表示为把 $N \times D$ 的输入矩阵 $\mathbf{X}$ ，变成 $\hat{\mathbf{X}}$ ，然后再进行线性变换：

$$\begin{aligned}\mu_B &= \frac{1}{N} \sum_{i=1}^N \mathbf{X}_i \\ \sigma_B^2 &= \frac{1}{N} \sum_{i=1}^N (\mathbf{X}_i - \mu_B)^2 \\ \hat{\mathbf{X}}_i &= \frac{\mathbf{X}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ \mathbf{Y}_i &= \gamma \hat{\mathbf{X}}_i + \beta\end{aligned}$$

而对于卷积层（卷积层见下文），批量归一化的过程稍有不同，因为卷积层的输出是一个四维张量（批量大小、高度、宽度、通道数）。在这种情况下，我们是对 N ， H ， W 三个维度进行归一化，算均值和方差，然后对每个通道进行缩放和平移。

批量归一化在卷积里极为常用，又稳又快还能折叠权重，是现代神经网络的标配。但在线性层中因为样本量较少，批量归一化的效果并不理想，反而可能引入噪声。

层归一化 层归一化（Layer Normalization）是一种对每个样本的所有特征进行归一化的方法。具体来说，对于一个线性层，我们是把 $N \times D$ 的输入矩阵 $\mathbf{X}$ ，对每一行进行归一化：

$$\begin{aligned}\mu_i &= \frac{1}{D} \sum_{j=1}^D \mathbf{X}_{ij} \\ \sigma_i^2 &= \frac{1}{D} \sum_{j=1}^D (\mathbf{X}_{ij} - \mu_i)^2 \\ \hat{\mathbf{X}}_{ij} &= \frac{\mathbf{X}_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \\ \mathbf{Y}_{ij} &= \gamma \hat{\mathbf{X}}_{ij} + \beta\end{aligned}$$

类似的，在卷积层中，层归一化也是每个样本的 C ， H ， W 三个维度进行归一化，和 N 无关。

层归一化在 RNN 等序列模型中非常常用，因为它不依赖于批量大小，适合处理变长序列数据。但在卷积层中，层归一化的效果并不如批量归一化好。且层归一化计算量较大，训练速度较慢。

30.2.6 数据清洗和数据增强

虽然上述方法能够在一定程度上缓解过拟合问题，但数据本身的质量和数量也是影响模型泛化能力的重要因素。一般在训练之前，我们还需要对数据进行**数据清洗**和**数据增强**，以提高数据的质量和多样性。

数据清洗 数据清洗是指对原始数据进行预处理，去除噪声和异常值，填补缺失值，从而提高数据质量。常见的数据清洗方法包括：

- 去除重复数据：删除数据集中重复的样本，避免模型过拟这些样本。
- 处理缺失值：对于缺失的数据，可以选择删除含有缺失值的样本，或者使用均值、中位数、众数等方法进行填补。
- 去除异常值：使用统计方法（如 Z-score）识别并删除异常值，防止其对模型训练产生负面影响。

数据清洗能够提高数据的质量，从而提升模型的训练效果和泛化能力。

数据增强 数据增强是指通过对原始数据进行各种变换，生成新的样本，从而增加数据量和多样性。常见的数据增强方法包括：

- 图像翻转：水平翻转或垂直翻转图像，增加数据的多样性。
- 图像旋转：随机旋转图像一定角度，模拟不同视角下的图像。
- 图像裁剪：随机裁剪图像的一部分，增强模型对局部特征的鲁棒性。
- 图像缩放：随机缩放图像大小，模拟不同距离下的图像。
- 添加噪声：在图像中添加随机噪声，提高模型对噪声的鲁棒性。

数据增强能够有效增加训练数据的多样性，防止模型过拟合，提高泛化能力。

30.3 更复杂的网络结构：卷积、池化和残差

30.3.1 卷积层

学完了刚刚的理论知识，我们高高兴兴地把多层神经网络堆积起来，这个确实能很好地解决 MNIST-10 这种简单的分类问题。

然后我们现在又要引入新的问题了：假设我们现在有一百万个图片（同学们估计要吐槽了），每张图片上面都是一个物体的照片，可能是猫、狗、车、飞机等多种类别。这些图片的大小都是 256x256 像素，每个像素是 RGB 三色。那么，我们希望把这些图片分成多个类别。这个数据集就是 CIFAR-10 数据集，该数据集依然是一个 10 分类问题，但是图片的复杂度远高于 MNIST-10。

我们试着用刚刚的知识解决一下这个问题吧。首先向量化： $\mathbf{x} \in \mathbb{R}^{256 \times 256 \times 3}$ ，也就是一个 196608 维的向量。然后堆叠多层神经网络，进行分类。

我估计大多数人看到 196608 就已经吓跑了。实际上计算机也是这样，你让他找一个 196608 维空间的超平面把这些点分开，计算量简直爆炸。更要命的是，这些图片的像素排列顺序是有意义的：相邻的像素往往属于同一个物体的一部分，而远离的像素则可能属于不同的物体。如果我们把图片向量化，那么这种空间结构信息就丢失了，模型很难捕捉到这些重要的特征。

那有没有什么办法能够在保留图像信息的同时，还能压低计算量？

我们知道，一个图是一个二维矩阵(加上颜色通道)，而我们可以使用**卷积操作**(Convolution Operation)来提取图像的局部特征。卷积操作通过一个小的滤波器(Kernel)在图像上滑动，计算局部区域的加权和，从而提取边缘、纹理等特征。

换句话说，如果用 Python 从头写一个卷积操作，大概是这样子的：

```
1 import numpy as np
2 def conv2d(image, kernel, stride=1, padding=0):
3     # 添加零填充
4     if padding > 0:
5         image = np.pad(image, ((padding, padding), (padding, padding), (0, 0)),
6                               ↪ mode='constant')
7
8     # 获取图像和核的尺寸
9     img_h, img_w, img_c = image.shape
10    k_h, k_w, k_c = kernel.shape
11
12    # 计算输出尺寸
13    out_h = (img_h - k_h) // stride + 1
14    out_w = (img_w - k_w) // stride + 1
15
16    # 初始化输出特征图
17    output = np.zeros((out_h, out_w, k_c))
18
19    # 执行卷积操作
20    for y in range(out_h):
21        for x in range(out_w):
22            for c in range(k_c):
23                region = image[y*stride:y*stride+k_h, x*stride:x*stride+k_w, :]
24                output[y, x, c] = np.sum(region * kernel[:, :, c])
25
26    return output
```

这里的 image 是输入图像，kernel 是卷积核，stride 是步幅，padding 是填充。这个函数会返回卷积后的特征图。

这个步长是很容易理解的：假设步长是 2，那么每次卷积核滑动 2 个像素，而不是 1 个像素。这样一来，输出的特征图尺寸会更小，计算量也会减少，但丢失的信息也会更多。而 padding 似乎不是很好理解，我们可以把它想象成在图像的边缘添加一些额外的像素，通常是 0 值的像素。这样做的目的是为了让卷积核能够更好地处理图像的边缘信息，避免边缘信息被忽略，另一方面也是为了和 stride 配合，防止没办法整除的问题。

一般的，卷积核有一个尺寸和一个滤波器数量。比如说，一个 3×3 的卷积核意味着它会在图像上滑动 3×3 的区域进行卷积操作，而滤波器数量决定了输出特征图的深度。假设我们

有 32 个滤波器，那么输出特征图的深度就是 32。这个滤波器是随机初始化的，然后通过训练来学习最优的滤波器参数，从而提取有用的特征。换句话说，卷积学的就是这些滤波器参数。

我们很容易得到输出特征图的尺寸。假设输入图是 $H_i \times W_i \times C_i$ ，卷积核是 $K_h \times K_w$ ，步长是 s ，填充是 p ，卷积核数量是 K ，那么输出特征图的尺寸就是：

$$\begin{aligned} H_o &= \frac{H_i - K_h + 2p}{s} + 1 \\ W_o &= \frac{W_i - K_w + 2p}{s} + 1 \\ C_o &= K \end{aligned}$$

这样的一层“卷积层”，可以学习的参数数量是：

$$\text{参数数量} = K_h \times K_w \times C_i \times K + K$$

这里的 K 是卷积核的数量，最后的 $+K$ 是每个卷积核对应的偏置项。

30.3.2 池化层

卷积层虽然能够提取图像的局部特征，但输出的特征图仍然可能非常大，计算量依然很高。为了解决这个问题，我们引入了**池化层** (Pooling Layer)，它通过对特征图进行下采样，减少特征图的尺寸，从而降低计算量。

池化和卷积的操作类似，也是通过一个小的窗口在特征图上滑动，但池化操作不是计算加权，而是取窗口内的最大值（最大池化）或平均值（平均池化）。这样一来，池化层能够保留重要的特征，同时减少特征图的尺寸。

也很容易得到池化层的输出尺寸。假设输入特征图是 $H_i \times W_i \times C_i$ ，池化窗口是 $P_h \times P_w$ ，步长是 s ，那么输出特征图的尺寸就是：

$$\begin{aligned} H_o &= \frac{H_i - P_h}{s} + 1 \\ W_o &= \frac{W_i - P_w}{s} + 1 \\ C_o &= C_i \end{aligned}$$

池化层没有可学习的参数，因为它只是对输入特征图进行简单的下采样操作，不涉及权重和偏置的更新。

通过卷积层和池化层的组合，我们可以构建一个强大的卷积神经网络 (Convolutional Neural Network, CNN)，它能够有效地处理图像数据，提取有用的特征，并进行分类任务。最经典的 CNN 莫过于只有七层的 LeNet-5 了，它在手写数字识别任务上取得了非常好的效果，奠定了 CNN 在计算机视觉领域的基础。它的组成是：

- 输入层：32 × 32 的灰度图像。
- 卷积层 1：6 个 5 × 5 的卷积核，步长为 1，输出特征图尺寸为 28 × 28 × 6。
- 池化层 1：2 × 2 的最大池化，步长为 2，输出特征图尺寸为 14 × 14 × 6。
- 卷积层 2：16 个 5 × 5 的卷积核，步长为 1，输出特征图尺寸为 10 × 10 × 16。

- 池化层 2： 2×2 的最大池化，步长为 2，输出特征图尺寸为 $5 \times 5 \times 16$ 。
- 全连接层 1：将特征图展平为 400 维向量，连接 120 个神经元。
- 全连接层 2：连接 84 个神经元。
- 输出层：连接 10 个神经元，对应 10 个类别。

而之后的 AlexNet、VGG 等网络则在 LeNet-5 的基础上进行了改进和扩展，取得了更好的性能。

30.3.3 卷积层和池化层的反向传播

刚刚我们已经知道这两个东西是怎么计算的了，那么我们现在要做的就是反向传播。卷积层和池化层的反向传播和线性层类似，但有一些特殊之处。

对于卷积层，我们需要计算损失函数对卷积核和输入特征图的梯度。假设输入特征图是 $\mathbf{X}$ ，卷积核是 $\mathbf{K}$ ，输出特征图是 $\mathbf{Y}$ ，那么损失函数对卷积核和输入特征图的梯度可以通过链式法则计算出来：

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{K}} &= \frac{\partial L}{\partial \mathbf{Y}} * \mathbf{X} \\ \frac{\partial L}{\partial \mathbf{X}} &= \frac{\partial L}{\partial \mathbf{Y}} * \mathbf{K}_{\text{rotated}}\end{aligned}$$

这里的 $*$ 表示卷积操作， $\mathbf{K}_{\text{rotated}}$ 是卷积核旋转 180 度后的结果。

对于池化层，反向传播相对简单。假设输入特征图是 $\mathbf{X}$ ，输出特征图是 $\mathbf{Y}$ ，那么损失函数对输入特征图的梯度可以通过以下方式计算出来：

$$\frac{\partial L}{\partial \mathbf{X}}[i, j, c] = \begin{cases} \frac{\partial L}{\partial \mathbf{Y}}[m, n, c], & \text{if } (i, j) \text{ is the max position in the pooling window} \\ 0, & \text{otherwise} \end{cases}$$

这里的 (m, n) 是池化窗口在输出特征图中的位置，而 (i, j) 是池化窗口在输入特征图中的位置。也就是说，只有在池化窗口内取到最大值的位置，梯度才会传递回来，否则梯度为 0。

30.3.4 残差块

在 2015 年以前，神经网络的层数非常浅。著名的 GoogleNet 也只有 22 层，而 VGGNet 也只有 19 层。那为什么不能堆叠更多的层数呢？原因很简单：随着层数的增加，训练变得越来越困难，梯度消失和梯度爆炸问题变得更加严重，导致模型难以收敛。于是就导致了一个问题：堆料还不如不堆。

但在 2015 年，He 等人提出了**残差网络**（Residual Network, ResNet），成功地训练了一个深达 152 层的神经网络，并在 ImageNet 竞赛中取得了冠军，且鲁棒性非常强，基本上吊打了先前所有的模型。为了理解该模型的核心思想，我们需要先了解**残差块**（Residual Block）。

深度网络一旦层数多，就会出现优化退化问题：随着网络深度的增加，训练误差反而增大，导致模型性能下降。

而残差层就是基于解决上述问题提出的：把直接拟合 $H(x)$ 的问题，转化为拟合残差函数 $F(x) = H(x) - x$ 的问题。

乍一看这并没有什么区别。但是实际上，这样做直接解决了上述问题：只要 $F(x) = 0$ ，那么就相当于恒等映射，换言之网络随时可以把这玩意“跳过”，从而避免了深层网络的退化问题，让优化器做的至少不会比浅层网络更差。

假设输入是 $\mathbf{x}$ ，输出是 $\mathbf{y}$ ，那么残差块的计算过程是：

$$\mathbf{y} = \mathbf{x} + F(\mathbf{x}, \mathbf{W})$$

其中 $F(\mathbf{x}, \mathbf{W})$ 表示残差函数，通常是由两层或三层卷积层组成的非线性变换。这样直接把梯度传递给输入 $\mathbf{x}$ ，从而缓解了梯度消失问题，于是大家终于可以随心所欲地堆叠更多的层数了。

30.3.5 残差块的反向传播

残差块的反向传播过程如下：

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \left(1 + \frac{\partial F(\mathbf{x}, \mathbf{W})}{\partial \mathbf{x}} \right)$$

这里的 1 来自于恒等映射部分，确保了梯度能够直接传递给输入 $\mathbf{x}$ ，从而缓解了梯度消失问题。

于是，把很多残差块串起来，就得到了残差网络。残差网络通过引入残差块，成功地训练了非常深的神经网络，极大地提升了模型的性能和鲁棒性。从此，网络越深效果越好就成了现实，深度学习正式迈入了“深度时代”。

而 ResNet 迄今为止依然是计算机视觉领域的基石，很多后续的网络结构都是在 ResNet 的基础上进行改进和扩展的，例如 DenseNet、ResNeXt 等。

30.4 另一番光景：循环神经网络

前面我们介绍的神经网络和卷积神经网络，都是针对静态数据设计的，比如图像、表格数据等。然而，在很多实际应用中，数据是有时间序列性质的，比如语音、文本、视频等。这时候，我们需要一种能够处理序列数据的神经网络结构，这就是**循环神经网络**（Recurrent Neural Network, RNN）。

30.4.1 朴素的循环神经网络

循环神经网络通过引入循环连接，使得网络能够记住之前的状态，从而捕捉序列数据中的时间依赖关系。具体来说，RNN 在每个时间步 t ，不仅接收当前输入 $\mathbf{x}_t$ ，还接收前一时间步的隐藏状态 $\mathbf{h}_{t-1}$ ，并计算当前的隐藏状态 $\mathbf{h}_t$ ：

$$\mathbf{h}_t = \sigma(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

这里， $\mathbf{W}_{xh}$ 是输入到隐藏状态的权重矩阵， $\mathbf{W}_{hh}$ 是隐藏状态到隐藏状态的权重矩阵， $\mathbf{b}_h$ 是偏置向量， σ 是激活函数。

RNN 的输出可以通过当前的隐藏状态计算得到：

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

这里， $\mathbf{W}_{hy}$ 是隐藏状态到输出的权重矩阵， $\mathbf{b}_y$ 是偏置向量。

RNN 的训练过程同样使用反向传播算法，但由于 RNN 的循环结构，反向传播需要通过时间展开（Backpropagation Through Time, BPTT）来计算梯度。具体来说，我们需要将 RNN 在时间维度上展开成一个深度网络，然后对每个时间步的参数进行梯度计算和更新。具体过程比较复杂，我们这里就不展开细说了。

30.4.2 改进的 RNN：LSTM 和 GRU

然而，RNN 在处理长序列时，容易出现梯度消失和梯度爆炸问题，导致模型难以训练。为了解决这个问题，研究人员提出了长短期记忆网络（Long Short-Term Memory, LSTM）和门控循环单元（Gated Recurrent Unit, GRU）等改进的 RNN 结构，这些结构通过引入门控机制，有效地缓解了梯度消失问题，使得模型能够捕捉更长时间的依赖关系。

LSTM 通过引入三个门控单元（输入门、遗忘门、输出门）来控制信息的流动，从而有效地捕捉长时间的依赖关系。具体来说，LSTM 在每个时间步 t ，计算以下内容：

$$\begin{aligned}\mathbf{f}_t &= \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \\ \mathbf{i}_t &= \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \\ \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$

这里， $\mathbf{f}_t$ 是遗忘门，控制前一隐藏状态 $\mathbf{h}_{t-1}$ 的信息保留程度； $\mathbf{i}_t$ 是输入门，控制当前输入 $\mathbf{x}_t$ 的信息写入程度； $\mathbf{o}_t$ 是输出门，控制当前隐藏状态 $\mathbf{h}_t$ 的信息输出程度； $\mathbf{c}_t$ 是细胞状态，存储长期记忆； $\odot$ 表示逐元素乘法。

GRU 则是 LSTM 的简化版本，只有两个门控单元（重置门和更新门），计算过程略。

循环神经网络及其变种在自然语言处理、语音识别等领域取得了显著的成功，成为处理序列数据的主流方法。然而，随着 Transformer 等新型架构的出现，RNN 在某些任务上的优势逐渐被削弱，但其基本思想和技术仍然对深度学习的发展产生了深远的影响。

30.4.3 Transformer：一拍脑袋的新思路

提到这个名词，大家大概很容易想起一篇震惊世界的论文：《Attention Is All You Need》。这篇论文提出了一种全新的神经网络架构——Transformer，彻底改变了自然语言处理领域的格局。而这个论文的标题也被广泛引用、致敬甚至调侃，成为了深度学习领域的经典之一，后续很多论文也模仿这种命名风格。

但是值得注意的是，Transformer 架构目前的原理尚不清楚，大家现在目前依然处于“知其然而不知其所以然”的阶段，甚至上述论文的作者也承认他们并不完全理解为什么 Transformer 能够如此有效地工作，“注意力机制”也很难被称作是一个良好的解释。但就是这一拍脑袋的新思路，成为了现在 LLM 如此蓬勃发展的基石。

自注意力机制 Transformer 的核心思想是自注意力机制 (Self-Attention Mechanism)。自注意力机制允许模型在处理输入序列的每个位置时，动态地关注序列中的其他位置，从而捕捉长距离的依赖关系。具体来说，给定一个输入序列 $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$ ，自注意力机制通过计算查询 (Query)、键 (Key) 和值 (Value) 来实现注意力的计算：

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V$$

这里， $\mathbf{W}_Q$ 、 $\mathbf{W}_K$ 和 $\mathbf{W}_V$ 是可学习的权重矩阵。然后，注意力得分通过以下公式计算：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

这里， d_k 是键的维度，用于缩放点积，防止数值过大导致梯度消失。这样避免了 RNN 中的序列计算问题，使得 Transformer 能够并行处理整个输入序列，大大提高了训练效率。

多头注意力机制 上述公式看起来确实很优雅、很美妙，但单一视角下的注意力机制可能无法捕捉到输入序列中的多样化信息。为了解决这个问题，真正“一拍脑袋”的东西——**多头注意力机制** (Multi-Head Attention Mechanism) 被提出了，核心理论是把一个头拆成多个头，每个头学习不同的注意力表示，从而捕捉输入序列中的多样化信息，然后再把多个头拼回去（一般是 8 个头）。

非常神奇的是，当把头拆成 8 个以后，模型既能在不同子空间学到“主谓一致”，又能顺手抓住“指代消解”，好像一只章鱼用八根触角同时阅读一句话。至于为什么 8 个头恰好够用、更多或者更少为什么反而不行，没人能说清楚，但这玩意确实好用！总不能是“八仙过海，各显神通”吧？

位置编码 但自注意力机制对顺序没有任何感知，无论输入是“天上下雨”还是“雨下上天”，算出来的注意力权重是一模一样的。为了把“位置”这个信息加回去，Transformer 引入了**位置编码** (Positional Encoding)。位置编码通过为每个位置添加一个独特的向量，使得模型能够感知序列中各个元素的位置关系。但这依然是“一拍脑袋”的设计。具体来说，位置编码可以通过以下公式计算：

$$\begin{aligned}\text{PE}(\text{pos}, 2i) &= \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \\ \text{PE}(\text{pos}, 2i+1) &= \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right)\end{aligned}$$

这里， pos 是位置索引， i 是维度索引， d_{model} 是模型的维度。通过这种方式，位置编码为每个位置生成一个独特的向量，从而使得模型能够感知序列中各个元素的位置关系。

正余弦交替，波长呈几何级数递减，既保证任意两个位置能被唯一区分，又方便模型通过线性组合外推到更长句子。这一招看似“手工特征工程”，却意外好使；后来也有人尝试把位置信息直接交给网络自学 (Learnable Positional Embedding)，结果在超长文本上还不如正余弦稳健。你说这是为什么呢？好吧，我也不知道。

花非花，雾非雾 那现在头也拆成了 8 个，位置也贴进去了，Transformer 一路高歌猛进，在各个 NLP 任务上痛击其余所有模型，成为了新的王者，于是现在 NLP 领域的排榜几乎全是 Transformer 内战。然而回到刚开始那一句“知其然而不知其所以然”，Transformer 的成功依然是一个谜：

- 注意力权重就能解释了吗？权重高未必因果大、低也不代表无关，注意力权重和模型决策之间并没有严格的对应关系，几轮 LayerNorm 和 Feed Forward 之后，原始的注意力权重早就被混合稀释了，连可读的语义都剩不下，这上哪去解释？
- 探针实验表明，有的头专职语法，有的头喜欢实体，但一旦把这些头初始化再重新训练，模型依然能收敛，说明这些头并没有“天生”就该负责某些任务，功能分工是后天习得的，并非设计使然。
- 更大胆的消融实验表明，即使一口气删了一半的头，模型性能也仅仅是慢慢上升。但如果把某几个特定的权重矩阵（比如 Value 矩阵）冻住，性能就会大幅下降，说明 Transformer 的成功并非单靠注意力机制，而是多种因素共同作用的结果。我们现在连谁是“关键先生”都说不清楚。

于是一篇又一篇论文试图解释 Transformer 的成功，但至今仍没有一个统一的理论框架能够完全解释其工作原理，大概这就是新时代的盲人摸象：你看像柱子，我看像绳索，但模型心里想的可能是“钝角”。

十年前，RNN 的梯度消失是无法避免的深渊；五年前，Transformer 架构架了座桥，大家一拥而上，发现深渊之后竟然是一片辽阔的未知大陆，但尽头依然迷雾重重。或许真正的解释需要等到下一个“一拍脑袋”的家伙：“嘿，要不我们试试这个？”这不禁让我想起普朗克是怎么凑出黑体辐射公式的：凑出公式只需要“一拍脑袋”，但解释为什么这个公式对，却花了几十年时间，才有了量子力学的诞生。而真正的科研，或许就是在无数个“一拍脑袋”中找到那些真正能用的点子，并穷尽一生去解释这些点子的过程吧。

30.5 缺少数据的做法：对抗学习和强化学习

上述几个学习：CNN、RNN、Transformer，都是属于监督学习（Supervised Learning）的范畴，即通过大量标注数据来训练模型。在大多数情况下，标注的数据是很容易获得的，即使是现在网上也有很多数据集可以直接下载使用。但在某些情况下，标注数据是很难获得的，甚至是无法获得的（例如自动驾驶中的场景数据）。这时候，我们需要引入其他的学习范式，例如**对抗学习**（Adversarial Learning）和**强化学习**（Reinforcement Learning），这些训练仅需要更少的数据，甚至不需要标注数据，就能训练出强大的模型。

30.5.1 对抗学习

对抗学习来自 2014 年 Ian Goodfellow 等人在酒吧吵架时的灵感爆发。目前该领域已经养活了半个生成式 AI 圈，目前很多的 AI 绘图、文本生成模型，都是基于对抗学习的思想来训练的。对抗学习的核心思想是通过两个模型的对抗训练来提升生成模型的性能：

生成器 G 把随机噪声 z 变成伪造数据 $G(z)$ ，试图欺骗判别器，让其认为这些数据是真实的。至于这个到底是什么，可以是图片、文本、音频等任何形式的数据，甚至蛋白质结构也行。

判别器 D 试图区分真实数据和生成器生成的伪造数据。

这两个东西组合在一起，就叫做一对苦命鸳鸯 GAN，生成对抗网络。

用数学语言来说，生成器和判别器的目标是相互对立的。生成器的目标是最大化判别器对伪造数据的误判概率，而判别器的目标是最小化对真实数据和伪造数据的误判概率。这个过程可以通过以下损失函数来表示：

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

这里， $p_{data}(\mathbf{x})$ 是真实数据的分布， $p_z(\mathbf{z})$ 是随机噪声的分布， $D(\mathbf{x})$ 是判别器对输入数据 $\mathbf{x}$ 的输出概率， $G(\mathbf{z})$ 是生成器对随机噪声 $\mathbf{z}$ 的生成数据。

通俗翻译：D 拼命把 log 两项都搞大，G 拼命把第二项搞小。D 越厉害，G 就越难骗过 D，于是 G 就得学会生成更逼真的数据来欺骗 D。这个过程就像是一场猫捉老鼠的游戏，D 和 G 不断地相互提升，最终达到一个平衡状态：G 的假货能以假乱真，而 D 也无法分辨真假。

而训练流程也很简单，交替进行以下两个步骤：

- 固定生成器，训练判别器，使其能够更好地区分真实数据和伪造数据。
- 固定判别器，训练生成器，使其能够生成更逼真的数据来欺骗判别器。

实际代码把 G 和 D 交替更新或按 1 比 k 的比例更新。

上述 GAN 被称作香草味 GAN (Vanilla GAN)，后来又出现了很多改进版本，例如 DCGAN、WGAN、CycleGAN 等，这些版本在生成质量、训练稳定性等方面都有所提升。

而现在 GAN 的“假货”已经包罗万象，从 AI 绘图（如 DALL·E、Stable Diffusion 等）到文本生成（如 GPT 系列）再到音频合成（如 WaveGAN 等），无所不包，无所不能。对抗学习通过这种“你来我往”的训练方式，使得生成模型能够不断提升其生成能力，最终达到以假乱真的效果。

30.5.2 强化学习

如果说对抗学习是“尔虞我诈”，那么强化学习就是“知行合一”。强化学习的核心思想是通过与环境的交互来学习最优策略，从而最大化累积奖励。具体来说，强化学习包括以下几个关键要素：

- 状态 s** 当前环境是什么（游戏画面、股票价格等）。
- 动作 a** 智能体可以采取的行动（移动、买卖等）。
- 奖励 r** 智能体采取某个动作后获得的反馈（分数、利润等）。
- 策略 π** 智能体根据当前状态选择动作的规则。
- 转移 P** 环境状态的变化规律。

在强化学习中，智能体通过观察当前状态 s_t ，根据策略 π 选择一个动作 a_t ，然后执行该动作，环境会反馈一个奖励 r_t ，并转移到下一个状态 s_{t+1} 。这个过程不断重复，智能体通过不断地与环境交互，学习如何选择最优的动作。

或者说，在强化学习中，智能体完全没有任何“猜标签”的过程，而是试图最大化长期累积奖励（游戏得分、投资回报等）。智能体通过不断地试错，逐渐学会在不同状态下选择最优的动作，从而实现其目标。数学上，强化学习的目标是最大化累积奖励的期望值，通常表示为：

$$J(\pi) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

这里， γ 是折扣因子，用于权衡当前奖励和未来奖励的重要性。这个数值通常设定在 0 到 1 之间，越接近 1 表示越重视未来的奖励。

Q 学习 这是最朴素的强化学习算法：当状态、动作不太多的时候，可以直接维护一张表格，记录每个状态-动作对的价值（Q 值）。智能体在每个时间步选择一个动作，然后根据环境反馈的奖励和下一个状态，更新对应的 Q 值。具体来说，Q 学习的更新公式如下：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

这里， α 是学习率，控制 Q 值更新的步长； r_t 是当前奖励； γ 是折扣因子； $\max_{a'} Q(s_{t+1}, a')$ 表示在下一个状态 s_{t+1} 下选择的最优动作的 Q 值。这个被叫做贝尔曼最优方程。但是高维状态用这个就完蛋了，表格根本存不下。

DQN 这是最经典的强化学习算法之一。DQN 通过引入神经网络来近似动作价值函数（Q 函数），也就是用网络 $Q(s, a; \theta)$ 来表示状态-动作对的价值，其中 θ 是神经网络的参数。DQN 通过与环境交互，收集状态、动作、奖励和下一个状态的数据，然后使用这些数据来训练神经网络，使其能够更准确地估计 Q 值。DQN 的训练过程包括以下几个步骤：

- 经验回放：将智能体与环境交互过程中收集的数据存储在一个经验回放缓冲区中，然后从中随机采样一批数据来训练神经网络，打破数据之间的相关性，提高训练稳定性。
- 目标网络：引入一个目标网络 $Q'(s, a; \theta^-)$ ，其参数 θ^- 定期从主网络 θ 复制过来，用于计算目标 Q 值，减少训练过程中的震荡。
- 损失函数：使用均方误差（Mean Squared Error, MSE）作为损失函数，来衡量当前 Q 值和目标 Q 值之间的差异。具体来说，损失函数定义为：

$$L(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim D} \left[\left(r_t + \gamma \max_{a'} Q'(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

这里， D 表示经验回放缓冲区中的数据分布。

DQN 已经在 49 款 Atari² 游戏中取得了超越人类水平的表现，成为强化学习领域的一个重要里程碑。

策略梯度方法 这是另一类强化学习算法，直接优化策略 $\pi(a|s; \theta)$ ，而不是估计 Q 值。策略梯度方法通过计算策略的梯度来更新参数 θ ，使得累积奖励最大化。具体来说，策略梯度的更新公式如下：

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\pi)$$

²Atari 是雅达利公司在 1970 年代末至 1980 年代初推出的一系列家用电子游戏机，标志着电子游戏产业的兴起。

这里, α 是学习率, $\nabla_{\theta} J(\pi)$ 是策略的梯度。常见的策略梯度方法包括 REINFORCE 算法和 Actor-Critic 方法。前者通过采样轨迹来估计策略梯度, 而后者则结合了值函数的估计, 提高了训练效率和稳定性。

教师-学生强化学习 这是近年来兴起的一种强化学习方法, 结合了监督学习和强化学习的优势。教师-学生强化学习通过引入一个教师模型, 指导学生模型的学习过程, 从而提高训练效率和性能。具体来说, 教师模型通过提供额外的监督信号, 帮助学生模型更快地收敛到最优策略。这种方法在一些复杂任务中取得了显著的效果, 例如 AlphaGo 和 AlphaStar 等。最近这个强化学习方法在机械臂灵巧手抓握中也取得了重要成果。

30.5.3 两条道路的融合

近年来, 对抗学习和强化学习的结合也成为了一个研究热点。例如, 在生成对抗网络中引入强化学习的思想, 使得生成器能够通过与环境的交互来提升其生成能力; 或者在强化学习中引入对抗学习的机制, 使得智能体能够更好地适应复杂的环境。这些方法在一些实际应用中取得了显著的效果, 展示了对抗学习和强化学习结合的潜力。

总的说来, 如果把 CNN、RNN 等都叫做“闭卷笔试”的话, 那对抗学习就是面试(只不过面试官也是你自己), 而强化学习则和考试毫无关系, 就像野外生存一样, 没有目标, 只有活下去的本能。而当标签昂贵、环境复杂时, 这两种学习范式无疑是非常有价值的工具。

30.6 实践: PyTorch 中的高级神经网络模块

在前面的章节中, 我们介绍了神经网络的基本概念和原理。现在, 我们将介绍如何在 PyTorch 中使用高级神经网络模块来构建和训练神经网络模型。PyTorch 提供了丰富的模块和函数, 使得我们能够方便地实现各种神经网络结构。

PyTorch 和 TensorFlow 的一个重要区别在于, PyTorch 采用动态计算图 (Dynamic Computation Graph) 的方式, 这使得我们可以在运行时动态地构建计算图, 从而更灵活地实现复杂的神经网络结构。而 TensorFlow 则采用静态计算图 (Static Computation Graph), 需要先定义好计算图, 然后再进行计算。也正因此, PyTorch 的代码更接近于传统的 Python 编程风格, 更易于调试和理解, 更常见于研究领域; 而 TensorFlow 则更适合于大规模生产环境, 更常见于工业界。不过近年来 PyTorch 也隐隐有一统江湖的趋势。

30.6.1 使用 nn.Module 构建神经网络

在 PyTorch 中, 神经网络模型通常通过继承 `nn.Module` 类来构建。我们可以定义一个新的类, 重写 `__init__` 方法来定义网络的层, 并重写 `forward` 方法来定义前向传播的计算过程。下面是一个简单的示例, 展示如何构建一个包含两个全连接层的神经网络:

```
1 import torch
2 import torch.nn as nn
```

```

3 import torch.nn.functional as F
4
5 class SimpleNN(nn.Module):
6     def __init__(self, input_size, hidden_size, output_size):
7         super(SimpleNN, self).__init__()
8         self.fc1 = nn.Linear(input_size, hidden_size)
9         self.fc2 = nn.Linear(hidden_size, output_size)
10
11     def forward(self, x):
12         x = F.relu(self.fc1(x))
13         x = self.fc2(x)
14         return x
15
16     def backward(self, loss):
17         loss.backward() # 这里调用自动求导，无需手动实现反向传播
18
19 # 创建模型实例
20 model = SimpleNN(input_size=784, hidden_size=128, output_size=10)

```

在这个示例中，我们定义了一个名为SimpleNN的神经网络类，包含两个全连接层。前向传播过程中，我们使用 ReLU 激活函数对第一层的输出进行非线性变换。

而在反向传播过程中，我们只需调用loss.backward()，PyTorch 会自动计算梯度，无需手动实现反向传播算法，这是非常人性化的一个设计。

30.6.2 模型的训练和评估

在构建好神经网络模型后，我们需要进入训练循环来优化模型的参数。训练循环通常包括以下几个步骤：

- 前向传播：将输入数据传递给模型，计算输出。
- 计算损失：使用损失函数计算模型输出与真实标签之间的差异。
- 反向传播：计算损失函数对模型参数的梯度。
- 更新参数：使用优化器更新模型参数。

下面是一个简单的训练循环示例：

```

1 import torch.optim as optim
2
3 # 定义损失函数和优化器
4 criterion = nn.CrossEntropyLoss()
5 optimizer = optim.SGD(model.parameters(), lr=0.01)
6 # 训练模型
7 model.train() # 切换到训练模式
8 for epoch in range(num_epochs):
9     for inputs, labels in train_dataloader:
10         optimizer.zero_grad() # 清零梯度
11         outputs = model(inputs) # 前向传播
12         loss = criterion(outputs, labels) # 计算损失
13         loss.backward() # 反向传播
14         optimizer.step() # 更新参数

```

在这个示例中，我们使用交叉熵损失函数和随机梯度下降（SGD）优化器来训练模型。每个 epoch 中，我们遍历数据集，进行前向传播、计算损失、反向传播和参数更新。

而对模型的评估通常在训练完成后进行，我们可以使用验证集或测试集来评估模型的性能。评估过程通常包括前向传播和计算准确率等指标：

```
1 # 评估模型
2 model.eval() # 切换到评估模式
3 correct = 0
4 total = 0
5 with torch.no_grad(): # 禁用梯度计算
6     for inputs, labels in test_dataloader:
7         outputs = model(inputs)
8         _, predicted = torch.max(outputs.data, 1)
9         total += labels.size(0)
10        correct += (predicted == labels).sum().item()
11 print('Accuracy: {:.2f}%'.format(100 * correct / total))
```

在评估过程中，我们使用`model.eval()`切换到评估模式，并使用`torch.no_grad()`禁用梯度计算，实际是冻住模型参数，防止验证集和测试集数据泄漏到训练过程中。

30.6.3 使用预定义的神经网络模块和优化器等

PyTorch 的`torch.nn`模块提供了许多预定义的神经网络层和模块，如卷积层、池化层、批量归一化层等。我们可以直接使用这些模块来构建复杂的神经网络结构，而无需从头实现每个层。例如，我们可以使用`nn.Conv2d`来定义卷积层，使用`nn.MaxPool2d`来定义池化层：

```
1 class ConvNet(nn.Module):
2     def __init__(self):
3         super(ConvNet, self).__init__()
4         self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
5                                ↪ stride=1, padding=1)
6         self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
7         ...
8     def forward(self, x):
9         x = self.pool(F.relu(self.conv1(x)))
10        ...
```

此外，PyTorch 还提供了多种优化器，如 Adam、RMSprop 等，我们可以根据需求选择合适的优化器来训练模型：

```
1 optimizer = optim.Adam(model.parameters(), lr=0.001)
```

而 LSTM 甚至 Transformer 等复杂模块也都被封装好了，直接调用即可：

```
1 self.lstm = nn.LSTM(input_size=128, hidden_size=256, num_layers=2,
2                    ↪ batch_first=True)
3 self.transformer = nn.Transformer(d_model=512, nhead=8, num_encoder_layers=6,
4                    ↪ num_decoder_layers=6)
```

30.6.4 数据加载和预处理

在训练神经网络模型之前，我们需要准备好数据集，并进行必要的预处理。PyTorch 提供了 `torch.utils.data` 模块，用于方便地加载和处理数据集。我们可以使用 `Dataset` 类来定义自定义数据集，并使用 `DataLoader` 类来批量加载数据。下面是一个简单的数据加载和预处理示例：

```
1 from torch.utils.data import Dataset, DataLoader
2 from torchvision import transforms
3
4 class CustomDataset(Dataset):
5     def __init__(self, data, labels, transform=None):
6         self.data = data
7         self.labels = labels
8         self.transform = transform
9
10    def __len__(self):
11        return len(self.data)
12
13    def __getitem__(self, idx):
14        sample = self.data[idx]
15        label = self.labels[idx]
16        if self.transform:
17            sample = self.transform(sample)
18        return sample, label
19 # 定义数据预处理
20 transform = transforms.Compose([
21     transforms.ToTensor(),
22     transforms.Normalize((0.5,), (0.5,))
23 ])
24 # 创建数据集和数据加载器
25 dataset = CustomDataset(data, labels, transform=transform)
26 dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

尽管如此，对于诸如 MNIST-10 等著名数据集，PyTorch 的 `torchvision.datasets` 模块已经封装好了，我们只需一行代码就能搞定：

```
1 from torchvision import datasets
2 mnist_dataset = datasets.MNIST(root='./data', train=True, download=True,
  ↳ transform=transform)
```

`download=True` 参数会自动帮你下载数据集，非常方便。

30.6.5 模型保存和加载

有时候，我们需要保存训练好的模型，以便在之后进行评估或部署。PyTorch 提供了简单的接口来保存和加载模型参数。我们可以使用 `torch.save` 函数来保存模型的状态字典 (state dict)，并使用 `torch.load` 函数来加载模型参数。下面是一个简单的示例：

```
1 # 保存模型
```

```
2 torch.save(model.state_dict(), 'model.pth')
3 # 加载模型
4 model = SimpleNN(input_size=784, hidden_size=128, output_size=10)
5 model.load_state_dict(torch.load('model.pth'))
6 model.eval() # 切换到评估模式
7 ...
```

通过使用 PyTorch 中的高级神经网络模块，我们可以方便地构建、训练和评估各种神经网络模型。丰富的预定义模块和函数使得我们能够专注于模型设计和优化，而无需过多关注底层实现细节，从而大大提高了开发效率。

30.7 CUDA-C 和 GPU 编程简介

CUDA-C 是一种用于在 NVIDIA GPU 上进行并行计算的编程语言。使用 CUDA-C 实现神经网络可以显著提高计算速度，特别是在处理大规模数据集和复杂模型时。

对于习惯了 C++ 和 Python 等高级语言的读者来说，CUDA-C 的语法和编程模型可能会显得有些陌生，因为 CUDA-C 实际上是调用了 NVIDIA 的 GPU 计算 API，需要手动管理内存和线程等底层细节，这实际上是个 C，因此对于笔者这种习惯于 C++ 的 OOP、泛型、STL 等特性的程序员来说，写 CUDA-C 代码简直是一种比写 C 更痛苦的体验——我已经好久好久没见过这么多裸指针和手动内存管理了。不过，正因为如此，CUDA-C 能够提供更高的性能和更细粒度的控制，这对于高性能计算任务来说是非常重要的。

CUDA-C 要写好，首先要熟悉 C、GPU 的计算模型和多线程并发编程。关于 C 和多线程，我们在前面的章节已经介绍过了（但我在多线程章节里讲的是 CPU 多线程，用的例子也是现代 C++ 写的，从没讲过 C 写多线程怎么写），这里我们主要介绍 GPU 的计算模型。

GPU 的计算模型与 CPU 有很大的不同。GPU 采用的是 SIMD (Single Instruction, Multiple Data) 架构，能够同时处理大量的数据并行计算。GPU 中的计算单元被称为“线程块” (Thread Block)，每个线程块包含多个线程 (Thread)。线程块之间可以并行执行，而线程块内的线程可以通过共享内存进行通信和协作。因此我们很容易就能看出，GPU 是天生比 CPU 更适合大规模并行计算的。

在 CUDA-C 中，我们可以使用 `__global__` 关键字来定义一个 GPU 内核函数 (Kernel Function)，该函数将在 GPU 上并行执行。内核函数中的每个线程可以通过内置变量 `threadIdx` 和 `blockIdx` 来获取自己的线程索引和块索引，从而实现数据的并行处理。下面是一个简单的 CUDA-C 内核函数示例，展示如何在 GPU 上进行向量加法：

```
1 __global__ void vectorAdd(const float* A, const float* B, float* C, int N) {
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     if (i < N) {
4         C[i] = A[i] + B[i];
5     }
6 }
```

在这个示例中，我们定义了一个名为 `vectorAdd` 的内核函数，用于将两个向量 A 和 B 相加，并将结果存储在向量 C 中。每个线程通过计算自己的全局索引 `i` 来处理对应的向量元素。

为了给 CUDA-C 编写内核函数，我们还得手动管理 GPU 内存的分配和释放，这就更让人头疼了。我们需要使用 `cudaMalloc` 函数来分配 GPU 内存，使用 `cudaMemcpy` 函数来在主机 (CPU) 和设备 (GPU) 之间传输数据，最后使用 `cudaFree` 函数来释放 GPU 内存。下面是一个完整的 CUDA-C 程序示例，展示如何在 GPU 上进行向量加法：

```
1 #include <stdio.h>
2 #include <cuda.h>
3 __global__ void vectorAdd(const float* A, const float* B, float* C, int N) {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     if (i < N) {
6         C[i] = A[i] + B[i];
7     }
8 }
9
10 int main() {
11     int N = 1<<20; // 向量大小
12     size_t size = N * sizeof(float);
13     float *h_A, *h_B, *h_C; // 主机内存
14     float *d_A, *d_B, *d_C; // 设备内存
15
16     // 分配主机内存
17     h_A = (float*)malloc(size);
18     h_B = (float*)malloc(size);
19     h_C = (float*)malloc(size);
20
21     // 初始化输入向量
22     for (int i = 0; i < N; i++) {
23         h_A[i] = static_cast<float>(i);
24         h_B[i] = static_cast<float>(i);
25     }
26
27     // 分配设备内存
28     cudaMalloc((void**)&d_A, size);
29     cudaMalloc((void**)&d_B, size);
30     cudaMalloc((void**)&d_C, size);
31
32     // 将输入向量从主机复制到设备
33     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
34     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
35
36     // 启动内核函数
37     int threadsPerBlock = 256;
38     int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
39     vectorAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C, N);
40
41     // 将结果从设备复制到主机
42     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
43     printf("Result[0]: %f\n", h_C[0]);
44
45     // 释放设备内存
46     cudaFree(d_A);
47     cudaFree(d_B);
48     cudaFree(d_C);
49
50     // 释放主机内存
51     free(h_A);
52     free(h_B);
53     free(h_C);
```

```
54
55     return 0;
56 }
```

在这个示例中，我们首先分配了主机和设备内存，然后将输入向量从主机复制到设备，启动内核函数进行向量加法，最后将结果从设备复制回主机。最后，我们释放了设备和主机内存。

但是说实话，我看到这么多裸指针和 malloc/free 就已经头大如斗了。为了在 GPU 上运算，CUDA-C 不接受塞 struct，代码可读性和可维护性极差，真是让一个现代 C++ 程序员痛不欲生。更别提调试了，CUDA-C 的调试工具远不如 CPU 端的成熟，调试起来非常麻烦。

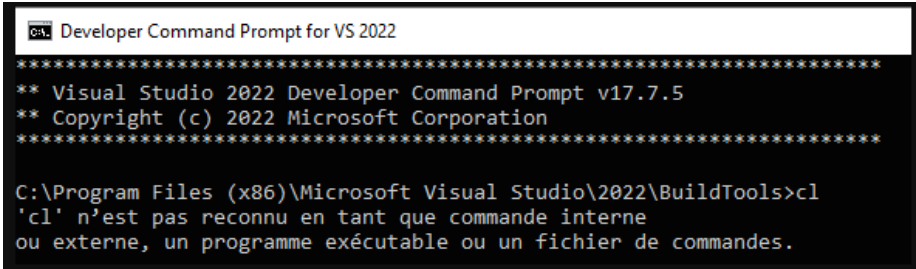


图 30.1: 该法语的意思是“未能找到 cl.exe”。截图日期：2024 年 10 月

不接受 C++ 特性也使得 NVIDIA 论坛下不乏批评的声音：

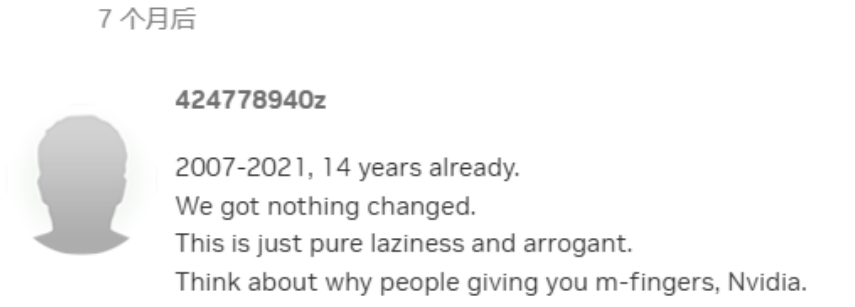


图 30.2: NVIDIA 论坛。截图日期：2024 年 10 月

但极其矛盾的是，为了追求极致的性能，很多高性能计算任务还是不得不使用 CUDA-C（甚至更底层的 PTX 汇编）。很多深度学习框架的底层实现也使用 CUDA-C 来实现，尤其是在深度学习领域，很多底层库（如 cuDNN、TensorRT 等）都是基于 CUDA-C 实现的，以充分利用 GPU 的计算能力。所以这个大家就见仁见智了，如果你对性能有极高的要求，并且愿意忍受编程复杂性，那么 CUDA-C 是一个不错的选择；但如果你更关注开发效率和代码可维护性，那么使用高级框架（如 PyTorch、TensorFlow 等）可能更合适，这些框架已经封装好了底层的 CUDA-C 实现，让我们能够专注于模型设计和训练，而无需过多关注底层细节。

30.8 计算机视觉简介

计算机视觉，指的是让计算机“看懂”图像和视频内容的技术。它涵盖了图像处理、模式识别、机器学习等多个领域，旨在使计算机能够自动分析和理解视觉信息。计算机视觉的应用非常广泛，包括人脸识别、自动驾驶、医疗影像分析、安防监控等。

计算机视觉的核心任务包括三个：图像分类、目标检测和图像分割。图像分类我们已经在上文介绍过了，CNN 和残差网络在这个任务表现良好。目标检测是指在图像中识别出多个目标，并给出它们的位置和类别。常见的目标检测算法包括 R-CNN、YOLO 和 SSD 等。图像分割则是将图像划分为多个区域，每个区域对应一个特定的对象或背景。常见的图像分割算法包括 FCN、U-Net 和 Mask R-CNN 等。

30.8.1 目标检测

目标检测最大的难点在于，同一个图片中可能有多个不同的目标，这些目标的大小、位置和类别都可能不同。为了解决这个问题，目标检测算法通常采用两阶段或单阶段的方法。两阶段方法首先生成一组候选区域，然后对每个候选区域进行分类和位置回归。单阶段方法则直接在图像上进行分类和位置回归，从而实现更快的检测速度。前者的主要代表是 R-CNN 系列，后者的主要代表是 YOLO 系列。

原始的 RCNN 原始的 R-CNN (Regions with CNN features) 算法包括以下几个步骤：

- 使用选择性搜索 (Selective Search) 算法生成一组候选区域。
- 对每个候选区域进行裁剪和缩放，然后使用预训练的 CNN 提取特征。
- 使用支持向量机 (SVM) 对提取的特征进行分类。
- 使用线性回归器对候选区域的位置进行微调。

Fast R-CNN Fast R-CNN 对原始 R-CNN 进行了改进，主要包括以下几个方面：

- 直接在整个图像上运行 CNN，生成一个特征图。
- 使用 RoI Pooling 层从特征图中提取候选区域的特征。
- 使用一个全连接层同时进行分类和位置回归。

Faster R-CNN Faster R-CNN 进一步改进了 Fast R-CNN，主要引入了区域建议网络 (Region Proposal Network, RPN)，用于生成候选区域。Faster R-CNN 的主要步骤包括：

- 使用 RPN 在特征图上生成候选区域。
- 使用 RoI Pooling 层从特征图中提取候选区域的特征。
- 使用一个全连接层同时进行分类和位置回归。

YOLO YOLO (You Only Look Once) 是一种单阶段目标检测算法，能够在单次前向传播中同时进行分类和位置回归。YOLO 的主要思想是将图像划分为一个网格，每个网格负责预测该区域内的目标。YOLO 的主要步骤包括：

- 将图像划分为 $S \times S$ 的网格。
- 对每个网格预测 B 个边界框和对应的置信度。
- 对每个边界框预测 C 个类别的概率。

YOLO 算法的优点在于速度非常快、网络小，适合实时应用，但在检测小目标和密集目标时表现较差。YOLO 系列算法也在不断发展，性能不断提升。

30.8.2 图像分割

图像分割是计算机视觉中的另一个重要任务，旨在将图像划分为多个区域，每个区域对应一个特定的对象或背景。图像分割可以分为语义分割和实例分割两种类型。语义分割关注的是每个像素所属的类别，而实例分割则不仅关注类别，还需要区分同一类别的不同实例。这个任务被称作 Segmentation。

图像分割目前的做法是使用纯卷积网络 (Fully Convolutional Network, FCN) 来实现。FCN 通过将传统的全连接层替换为卷积层，使得网络能够接受任意大小的输入图像，并输出与输入图像大小相同的分割图。FCN 的主要步骤包括：

- 使用卷积层和池化层提取图像的特征。
- 使用上采样层将特征图恢复到与输入图像相同的大小。
- 使用像素级分类器对每个像素进行分类。

其中这个上采样层是一个我们没见过的层，叫做反卷积层 (Deconvolutional Layer) 或者转置卷积层 (Transposed Convolutional Layer)。它的作用是将低分辨率的特征图恢复到高分辨率，从而实现图像的分割。反卷积层通过学习一组可训练的滤波器，将输入特征图进行上采样，生成更大的输出特征图。

假设输入特征图的大小为 (H_{in}, W_{in}) ，滤波器的大小为 (K_h, K_w) ，步幅为 (S_h, S_w) ，填充为 (P_h, P_w) ，则输出特征图的大小 (H_{out}, W_{out}) 可以通过以下公式计算：

$$H_{out} = (H_{in} - 1) \times S_h - 2P_h + K_h$$
$$W_{out} = (W_{in} - 1) \times S_w - 2P_w + K_w$$

通过调整滤波器的大小、步幅和填充，我们可以控制输出特征图的大小，从而实现不同的上采样效果。

30.9 练习

练习：机器学习实操

这些练习旨在帮助你巩固和应用所学的机器学习知识，但是需要一定的算力。如果你没有合适的硬件，可以考虑使用 Google Colab 等云端平台来完成这些练习。

单层线性分类器 用 MNIST-10 数据集为蓝本，从中抽出 0 和 1 两个数字作为所有的数据。划分训练集、测试集，并用 PyTorch 完成一个线性分类器。准确度尽量达到 99% 以上。

多层神经网络 用 MNIST-10 数据集的全部数据。划分训练集、测试集，并用 PyTorch 完成一个多层神经网络分类器（至少包含一个隐藏层）。准确度尽量达到 80% 以上。

卷积神经网络 用 MNIST-10 数据集的全部数据。划分训练集、测试集，并用 PyTorch 完成一个卷积神经网络分类器（至少包含两个卷积层和两个池化层）。准确度尽量达到 95% 以上。可以参考 LeNet-5 架构。特别说明：不要连续使用多个线性层，一定要在这些线性层之间加入非线性激活函数（如 ReLU）。

残差网络 用 CIFAR-10 数据集的全部数据。划分训练集、测试集，然后实现：VGG、ResNet、ResNeXt 三种架构的卷积神经网络分类器。比较它们的准确度和训练时间。可以参考相关论文中的架构设计。

Transformer 文本分类器 用 IMDb 电影评论数据集（影评文本和正负标签）完成一个文本分类任务。实现一个基于 Transformer 架构的文本分类器，并与传统的 RNN 或 LSTM 模型进行比较。评估它们在准确度和训练时间上的表现差异。

对抗学习 实现一个简单的生成对抗网络（GAN），用于生成手写数字图像。使用 MNIST-10 数据集中的 0 作为训练数据，设计一个生成器和一个判别器，并训练它们进行对抗学习。观察生成的图像质量，并尝试调整网络架构和超参数以提升生成效果。理论上说，经过大量训练，生成器大概会生成非常以假乱真的“0”。如果计算资源充足，可以尝试更多的数字类别，或者使用更复杂的数据集，如 CIFAR-10 等。

强化学习 魔塔是一个经典的游戏，规则可以参考网上的介绍。我们可以使用经典的 50 层魔塔作为环境，使用 DQN 算法训练一个智能体来玩这个游戏。实现 DQN 算法，并与传统的 Q 学习算法和搜索算法进行比较。评估它们在游戏中的表现和学习效率。50 层魔塔的状态空间和动作空间都比较大，训练可能需要较长时间和较强的计算资源，如果难以完成，可以仅训练其完成前十层。也可以尝试其他状态空间较小的游戏，如 Flappy Bird、杀戮尖塔等。

答案

在这里，我们仅试着完成第一题的解答，其他题目留给读者自行完成。

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torchvision import datasets, transforms
5 from torch.utils.data import DataLoader, SubsetRandomSampler
6 import numpy as np
7
8 # 数据加载和预处理
9 transform = transforms.Compose([
10     transforms.ToTensor(),
11     transforms.Normalize((0.5,), (0.5,))
12 ])
13
14 # 只选择数字 0 和 1
15 full_dataset = datasets.MNIST(root='./data', train=True, download=True,
16     ↪ transform=transform)
17 indices = np.where((full_dataset.targets == 0) | (full_dataset.targets ==
18     ↪ 1))[0]
19 subset_dataset = torch.utils.data.Subset(full_dataset, indices)
20
21 # 划分训练集和测试集
22 num_samples = len(subset_dataset)
23 indices = list(range(num_samples))
24 np.random.shuffle(indices)
25 split = int(np.floor(0.2 * num_samples))
26 train_indices, test_indices = indices[split:], indices[:split] # 80% 训练,
27     ↪ 20% 测试
```

```

25 # 在划分训练和测试集后一定不要把这两个集合混在一起
26 # 否则就是数据泄漏，测试集准确率会虚高
27
28 train_sampler = SubsetRandomSampler(train_indices)
29 test_sampler = SubsetRandomSampler(test_indices)
30
31 train_loader = DataLoader(subset_dataset, batch_size=64,
    ↪ sampler=train_sampler)
32 test_loader = DataLoader(subset_dataset, batch_size=64,
    ↪ sampler=test_sampler)
33
34 # 定义线性分类器
35 class LinearClassifier(nn.Module):
36     def __init__(self):
37         super(LinearClassifier, self).__init__()
38         self.fc = nn.Linear(28*28, 2) # 输入 28x28, 输出 2 类
39         # 这里之所以敢于这么写，是因为模型的输入和输出维度是确定的
40         # 在实际的操作中这里往往是把参数写在构造函数的参数里传进来的
41         # 大家实际操作一定不要写这样的 Magic Number
42
43     def forward(self, x):
44         x = x.view(-1, 28*28) # 展平输入
45         x = self.fc(x)
46         return x
47
48     def backward(self, loss):
49         loss.backward() # 自动求导
50
51 # 创建模型实例
52 model = LinearClassifier()
53 criterion = nn.CrossEntropyLoss() # 虽然交叉熵是多分类损失函数，但也能用在二分类上，
    ↪ 实际上退化为逻辑回归
54 optimizer = optim.SGD(model.parameters(), lr=0.01) # 这里用经典的 SGD，不使用动
    ↪ 量等技巧，也不用 Adam 等复杂优化器
55
56 # 训练模型
57 def train_model(model, epochs, train_loader, criterion, optimizer):
58     model.train()
59     for epoch in range(epochs): # 训练 10 个 epoch
60         for inputs, labels in train_loader:
61             optimizer.zero_grad() # 清零梯度
62             outputs = model(inputs) # 前向传播
63             loss = criterion(outputs, labels) # 计算损失
64             model.backward(loss) # 反向传播
65             optimizer.step() # 更新参数
66         print(f'Epoch {epoch+1}, Loss: {loss.item()}')
67
68 # 评估模型
69 def eval_model(model, test_loader):
70     model.eval()
71     correct = 0
72     total = 0
73     with torch.no_grad():
74         for inputs, labels in test_loader:
75             outputs = model(inputs)
76             _, predicted = torch.max(outputs.data, 1)
77             total += labels.size(0)
78             correct += (predicted == labels).sum().item()

```

```
79     accuracy = 100 * correct / total
80     print(f'Accuracy: {accuracy:.2f}%')
81     return accuracy
82
83 if __name__ == '__main__':
84     train_model(model, epochs=10, train_loader, criterion, optimizer)
85     eval_model(model, test_loader)
```

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沉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（#'O'）！

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、鼓励与便利的老师们、同学们和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：

- LCPU Getting Started 全体成员
- PKUHub 全体成员
- 🐼周乾康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，充实了多个部分的内容
- 🐼张庭瑄为本手册的排版提供了极多的帮助
- 🐼袁建东为手册充实了多个部分的内容
- 🐼王鸿铖提供了细化 LLM 使用方法的意见并给出了大纲
- 🐼吕钊杰提供了一些常用软件的推荐

周乾康: <https://github.com/wszqkzqk>

张庭瑄: <https://github.com/AlphaZTX>

袁建东: <https://github.com/yjdymv>

王鸿铖: <https://github.com/whcpumpkin>

吕钊杰: <https://github.com/Elkeid-me>



图 30.3: 感谢每一位读到这里的同学!

- 🐼包涛尼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
- 🐼吕浩鑫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
- 以及更多本校和外校的小伙伴们。在这里不能完全列举, 但在文中我也已经以脚注的形式表明了他们的贡献。

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!

可爱大笨蛋

2025 年 7 月, 在燕园