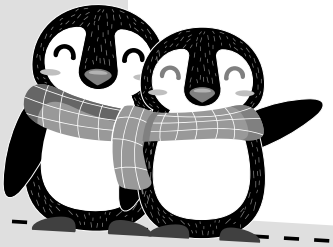

Linux

网络编程（第3版）

宋敬彬◎编著

清華大學出版社
北京

内 容 简 介

本书是获得大量读者好评的“Linux 典藏大系”中的一本。本书第 1、2 版出版后得到了大量读者的好评，曾经多次印刷并得到了 ChinaUnix 技术社区的推荐。本书全面、系统、深入地介绍 Linux 网络编程的相关知识，涉及面很广，从编程工具和环境搭建，到高级技术和核心原理，再到项目实战，几乎涵盖 Linux 网络编程的所有重要知识点。**本书提供教学视频、思维导图、教学 PPT 和习题参考答案等超值配套资料，可以帮助读者高效、直观地学习。**

本书共 20 章，分为 4 篇。第 1 篇“Linux 网络开发基础知识”，涵盖 Linux 操作系统概述、Linux 编程环境、文件系统概述，以及程序、进程和线程等相关知识；第 2 篇“Linux 用户层网络编程”，涵盖 TCP/IP 族概述、应用层网络服务程序概述、TCP 网络编程基础知识、服务器和客户端信息获取、数据的 I/O 及其复用、基于 UDP 接收和发送数据、高级套接字、套接字选项、原始套接字、服务器模型、IPv6 基础知识等；第 3 篇“Linux 内核网络编程”，涵盖 Linux 内核层网络架构和 netfilter 框架的报文处理；第 4 篇“综合案例”，介绍 3 个网络编程综合案例的实现，包括一个简单的 Web 服务器 SHTTPD 的实现、一个简单的网络协议栈 SIP 的实现和一个简单的防火墙 SIPFW 的实现。

本书内容丰富，讲解深入，适合想全面、系统、深入学习 Linux 网络编程的人员阅读，尤其适合 Linux 网络开发工程技术人员和基于 Linux 平台的网络程序设计人员作为参考读物。

图书在版编目（CIP）数据

Linux 网络编程 / 宋敬彬编著. —3 版. —北京：清华大学出版社，2024.4
（Linux 典藏大系）
ISBN 978-7-302-66051-4

Ⅰ. ①L… Ⅱ. ①宋… Ⅲ. ①Linux 操作系统－程序设计 Ⅳ. ①TP316.89

中国国家版本馆 CIP 数据核字（2024）第 070816 号

责任编辑：王中英
封面设计：欧振旭
责任校对：徐俊伟
责任印制：杨 艳

出版发行：清华大学出版社
网 址：https://www.tup.com.cn，https://www.wqxuetang.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-83470000 邮 购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京联兴盛业印刷股份有限公司
经 销：全国新华书店
开 本：185mm×260mm **印 张**：40.25 **字 数**：1057 千字
版 次：2010 年 1 月第 1 版 2024 年 4 月第 3 版 **印 次**：2024 年 4 月第 1 次印刷
定 价：159.00 元

产品编号：101193-01

前言

当前，Linux 已经成为非常流行的开源操作系统，在服务器和嵌入式系统等领域有广泛的应用，而且正在逐步应用于个人计算机的桌面操作系统上。Linux 网络程序设计在服务器和嵌入式领域有着广泛的应用。例如，Web 服务器、P2P 应用、嵌入式网络机顶盒、IPTV 机顶盒和手持设备等产品很多都采用开源的 Linux 操作系统。因此，能够熟练编写网络程序并构建自己的网络架构程序，对于程序开发人员是十分重要的。

本书是获得大量读者好评的“Linux 典藏大系”中的一本。本书全面、系统、深入地介绍 Linux 网络编程涉及的相关技术，涉及面很广，从编程工具和环境搭建，到核心原理和高级技术，再到项目实战，几乎涵盖 Linux 网络编程的所有重要知识点。其中，结合实例重点介绍 Linux 应用层网络设计、网络协议栈的实现原理和 Linux 内核防火墙技术。通过阅读本书，读者可以全面掌握 Linux 网络编程方方面面的技术，具备开发较为复杂网络项目的能力。

关于“Linux 典藏大系”

“Linux 典藏大系”是专门为 Linux 技术爱好者推出的系列图书，涵盖 Linux 技术的方方面面，可以满足不同层次和各个领域的读者学习 Linux 的需求。该系列图书自 2010 年 1 月陆续出版，上市后深受广大读者的好评。2014 年 1 月，创作者对该系列图书进行了全面改版并增加了新品种。新版图书一上市就大受欢迎，各分册长期位居 Linux 图书销售排行榜前列。截至 2023 年 10 月底，该系列图书累计印数超过 30 万册。可以说，“Linux 典藏大系”是图书市场上的明星品牌，该系列中的一些图书多次被评为清华大学出版社“年度畅销书”，还曾获得“51CTO 读书频道”颁发的“最受读者喜爱的原创 IT 技术图书奖”，另有部分图书的中文繁体版在中国台湾出版发行。该系列图书的出版得到了国内 Linux 知名技术社区 ChinaUnix（简称 CU）的大力支持和帮助，读者与 CU 社区中的 Linux 技术爱好者进行了广泛的交流，取得了良好的学习效果。另外，该系列图书还被国内上百所高校和培训机构选为教材，得到了广大师生的一致好评。

关于第 3 版

随着技术的发展，本书第 2 版与当前 Linux 的几个流行版本有所脱节，这给读者的学习带来了不便。应广大读者的要求，笔者结合 Linux 技术的新近发展对第 2 版图书进行全面的升级改版，推出第 3 版。相比第 2 版图书，第 3 版在内容上的变化主要体现在以下几个方面：

- Linux 系统更换为 Ubuntu 22.04；
- 对 Linux 内核的介绍增加 5.*系列；
- 对 IT 业界的动态信息进行更新；
- 对 GCC 软件包进行更新；
- 修订第 2 版中的一些疏漏，并对一些表述不够准确的内容重新表述；

- 对涉及的一些函数及其格式进行修改；
- 新增思维导图和课后习题，以方便读者梳理和巩固所学知识。

本书特色

1．提供配套教学视频，学习效果好

为了帮助读者更加高效、直观地学习，笔者专门针对书中的一些重点和难点内容录制配套教学视频，手把手带领读者进行学习。

2．内容由浅入深，讲解循序渐进

本书按照“基础知识→高级技术→进阶实战”的思路讲解，首先介绍 Linux 的基础知识与开发环境，然后介绍基本的 Linux 网络程序设计方法，接着介绍 Linux 内核网络编程方法，最后通过 3 个案例综合运用所介绍的知识，让读者更加深刻地理解 Linux 网络编程技术。

3．内容充实，涵盖面广

本书几乎涵盖 Linux 网络程序设计会用到的所有重要知识点，尤其对高级网络编程和原始套接字等用户层网络程序设计结合丰富的示例进行全面的讲解，另外对内核网络程序设计进行深入的剖析，还对 netfilter 框架进行详细的讲解，并给出一个全面使用 netfilter 框架的案例，以方便读者深入学习。

4．对比分析，讲解深入

本书在介绍多个主要函数时对用户空间和内核空间进行对比分析，让读者不但了解如何使用这些函数，而且能更加深入地理解为何这样用，做到所谓“知其然并知其所以然”。

5．案例精讲，提高实际开发水平

本书通过精讲 3 个典型案例，帮助读者更加深入地理解前面章节介绍的 Linux 网络编程的重要知识点，从而提高读者的实际开发水平。

6．提供习题、源代码、思维导图和教学 PPT

本书特意在每章后提供多道习题，用以帮助读者巩固和自测该章的重要知识点，另外还提供源代码、思维导图和教学 PPT 等配套资源，以方便读者学习和教师教学。

本书内容

第 1 篇　Linux 网络开发基础知识

本篇涵盖第 1～4 章，主要包括 Linux 操作系统概述、Linux 编程环境、文件系统概述，以及程序、进程和线程等相关知识。通过学习本篇内容，读者可以初步掌握 Linux 网络程序设计的基础知识，并了解 Linux 编程环境的相关知识。

第 2 篇　Linux 用户层网络编程

本篇涵盖第 5～15 章，主要包括 TCP/IP 族概述、应用层网络服务程序概述、TCP 网络编程基础知识、服务器和客户端信息获取、数据的 I/O 及其复用、基于 UDP 接收和发送数据、高级套接字、套接字选项、原始套接字、服务器模型、IPv6 基础知识等。通过学习本篇内容，读者可以全面、系统、深入地掌握 Linux 网络程序设计的大部分知识。

第 3 篇　Linux 内核网络编程

本篇涵盖第 16、17 章，主要包括 Linux 内核层网络架构和 netfilter 框架的报文处理。通过学习本篇内容，读者可以初步掌握 Linux 内核网络编程的相关知识。

第 4 篇　综合案例

本篇涵盖第 18～20 章，主要介绍 3 个网络编程综合案例的实现，包括一个简单的 Web 服务器 SHTTPD 的实现、一个简单的网络协议栈 SIP 的实现和一个简单的防火墙 SIPFW 的实现。通过学习本篇内容，读者可以掌握如何编写一个完整、可用的 Linux 网络程序。

阅读建议

- 对于没有基础的读者，尽量从前到后按顺序阅读，不要随意跳跃；
- 书中给出的示例和案例需要读者亲自上机动手实践，这样学习效果更好；
- 第 4 篇偏重于实战，这部分内容初期不需要读者全面掌握，只要理解基本的开发思路即可，等有了较丰富的开发经验后可进一步研读。

读者对象

- 想全面学习 Linux 网络编程的人员；
- Linux 网络编程从业人员；
- Linux 网络编程爱好者；
- 高等院校相关专业的学生；
- 培训机构的学员；
- 需要一本案头必备手册的开发人员。

配书资源获取方式

本书涉及的配套资源如下：

- 示例和案例源代码；
- 配套教学视频；
- 高清思维导图；
- 习题参考答案；
- 配套教学 PPT；
- 书中涉及的工具。

上述配套资源有以下 3 种获取方式：

- 关注微信公众号“方大卓越”，然后回复数字“17”，可自动获取下载链接；
- 在清华大学出版社网站（www.tup.com.cn）上搜索到本书，然后在本书页面上找到“资源下载”栏目，单击“网络资源”按钮进行下载；
- 在本书技术论坛（www.wanjuanchina.net）上的 Linux 模块进行下载。

技术支持

虽然笔者对书中所述内容都尽量予以核实，并多次进行文字校对，但是因时间所限，可能还存在疏漏和不足之处，恳请读者批评与指正。

读者在阅读本书时若有疑问，可以通过以下方式获得帮助：

- 加入本书 QQ 交流群（群号：302742131）进行提问；
- 在本书技术论坛（见上文）上留言，会有专人负责答疑；
- 发送电子邮件到 book@wanjuanchina.net 或 bookservice2008@163.com 获得帮助。

编　者

目录

第 1 篇　Linux 网络开发基础知识

第 2 篇　Linux 用户层网络编程

第 3 篇 Linux 内核网络编程

第 4 篇 综合案例

第 1 篇
Linux 网络开发基础知识

第 1 章　Linux 操作系统概述

Linux 操作系统是目前发展最快的操作系统之一，其在服务器和嵌入式等方面得到了迅速的发展，并在个人操作系统方面有广泛的应用，这主要得益于其开放性。本章的主要内容如下：

- 以时间为主线对 Linux 的发展历史进行介绍。
- 分析 Linux 和 UNIX 操作系统的异同。
- 介绍常用的几种 Linux 发行版本的特点。
- 对 Linux 操作系统架构进行简单的介绍。
- 介绍 GNU 通用公共许可证及其特点。

通过对本章的学习，读者可以对 Linux 的发展历史和 Linux 操作系统的基本特点有简单的认识。

1.1　Linux 的发展历史

Linux 操作系统于 1991 年诞生，目前已经成为主流的操作系统之一。其版本从开始的 Linux 0.01 版到目前的 Linux 6.0 版经历了 30 多年的发展，目前在服务器、嵌入式系统和个人计算机等多个方面得到了广泛应用。

1.1.1　Linux 的诞生和发展

Linux 的诞生和发展与个人计算机的发展历程是紧密相关的。在 1981 年之前没有个人计算机，计算机是大型企业和政府部门才能使用的昂贵设备。IBM 公司在 1981 年推出了个人计算机 IBM PC，从而促进了个人计算机的发展和普及。刚开始，微软帮助 IBM 公司开发的 MS-DOS 操作系统在个人计算机中占据统治地位。随着 IT 行业的发展，个人计算机的硬件价格虽然逐年在下降，但是软件价格特别是操作系统的价格一直居高不下。因此，计算机用户迫切希望有一款免费、开源的计算机系统。

1.1.2　Linux 名称的由来

Linux 操作系统最初并不称作 Linux。其开发者 Linus 给他的操作系统取的名字是 Freax，这个单词是怪诞的、怪物、异想天开的意思。当 Torvalds 将他的操作系统上传到服务器 ftp.funet.fi 上时，服务器管理员 Ari Lemke 对 Freax 这个名称很不赞成，因此将操作系统的名称改为了 Linus 的谐音——Linux，于是这个操作系统的名称就以 Linux 流传下来了。

1.2　Linux 的发展要素

Linux 是一套免费使用和自由传播的类 UNIX 的系统。Linux 诞生之后，借助 Internet，在全世界计算机爱好者的共同努力下，其成为目前世界上使用者最多的一种类 UNIX 操作系统。在 Linux 操作系统诞生、成长和发展的过程中，起到重要作用的有 5 个方面，分别是 UNIX 操作系统、Minix 操作系统、POSIX 标准、GNU 计划和 Internet 网络，下面具体介绍。

1.2.1　UNIX 操作系统

UNIX 操作系统于 1969 年在 Bell 实验室诞生，它是美国贝尔实验室的 Ken.Thompson 和 Dennis Ritchie 在 DEC PDP-7 小型计算机系统上开发的一种分时操作系统。

UNIX 是一个功能强大、性能优良、多用户和多任务的分时操作系统，在巨型计算机和普通 PC 等多种平台上都有十分广泛的应用。

在通常情况下，大型的系统应用如银行系统和电信系统，一般都采用固定机型的 UNIX 解决方案：在电信系统中以 Sun 公司（已经被 Oracle 公司收购）的 UNIX 系统方案居多，在民航系统里以 HP 公司的系统方案居多，在银行系统里以 IBM 系统的系统方案居多。

Linux 可以视为 UNIX 的一个复制版本，二者采用了几乎一致的系统接口，特别是在网络方面，二者接口的应用程序几乎完全一致。

1.2.2　Minix 操作系统

Minix 是一种基于微内核架构的类 UNIX 操作系统，它由荷兰的阿姆斯特丹自由大学的著名教授 Andrew S.Tanenbaum 于 1987 年开发完成。Minux 操作系统主要用于学生学习操作系统的原理。当时 Minix 操作系统在大学中是免费使用的，如果用于其他用途则需要收费。目前，Minix 操作系统全部是免费的，可以从许多 FTP 上下载，其中，Minix 3 是主流版本。

1.2.3　POSIX 标准

POSIX（Portable Operating System Interface for Computing Systems）是由 IEEE 和 ISO/IEC 开发的一套标准。POSIX 标准是对 UNIX 操作系统经验和实践的总结，对操作系统调用的服务接口进行了标准化，保证所编制的应用程序在源代码一级可以在多种操作系统上进行移植。

20 世纪 90 年代初，POSIX 标准的制定处于最后确定的投票阶段，而 Linux 正处于诞生时期。作为一个指导性的纲领性标准，Linux 的接口与 POSIX 相兼容。

1.3　Linux 与 UNIX 的异同

Linux 是一个类 UNIX 系统，没有 UNIX 就没有 Linux。但是，Linux 和传统的 UNIX 有很大的不同，二者的最大区别表现在版权方面：Linux 是开放源代码的自由软件，而 UNIX 是对

源代码实行知识产权保护的传统商业软件。此外，二者还存在如下区别：

- UNIX 操作系统大多数是与硬件配套的，操作系统与硬件进行了绑定；而 Linux 操作系统则可以运行在多种硬件平台上。
- UNIX 操作系统是一款商业软件（授权费大约为 5 万美元）；而 Linux 操作系统则是一款自由软件，是免费的，并且公开源代码。
- UNIX 的历史要比 Linux 悠久，但 Linux 操作系统吸取了其他操作系统的优点，其设计思想虽然源于 UNIX 但是要优于 UNIX。
- 虽然 UNIX 和 Linux 都是操作系统的名称，但是 UNIX 除了是一种操作系统的名称外，作为商标，它归 SCO 所有。
- Linux 的商业化版本有 Red Hat Linux、SuSe Linux、Slakeware Linux 和我国的红旗 Linux 等，还有 Turbo Linux；UNIX 主要有 Oracle 的 Solaris、IBM 的 AIX、HP 的 HP-UX，以及基于 x86 平台的 SCO UNIX/UNIXWare。
- Linux 操作系统的内核是免费的，而 UNIX 的内核并不公开。
- 在对硬件的要求上，Linux 操作系统比 UNIX 的要求低，并且没有 UNIX 对硬件要求那么苛刻；在系统安装的难易程度上，Linux 比 UNIX 容易得多；在使用上，Linux 没有 UNIX 那么复杂。

总体来说，Linux 操作系统无论在外观还是性能上都比 UNIX 好，但是 Linux 操作系统不同于 UNIX 源代码，在功能上，Linux 仿制了 UNIX 的一部分功能，与 UNIX 的 System V 和 BSD UNIX 相兼容。一般情况下，在 UNIX 上可以运行的源代码，在 Linux 上重新编译后就可以运行，甚至 BSD UNIX 的执行文件可以在 Linux 操作系统上直接运行。

1.4　常见的 Linux 发行版本和内核版本的选择

要在 Linux 环境中进行程序设计，首先要选择一款适合自己的 Linux 操作系统。本节介绍 Linux 常用的发行版本和内核，并简要介绍如何定制自己的 Linux 操作系统。

1.4.1　常见的 Linux 发行版本

Linux 的发行版本众多，曾有人收集过 300 多种发行版本。常用的 Linux 发行版本如表 1.1 所示。本书所使用的 Linux 版本为 Ubuntu。

表 1.1　常用的Linux发行版本

版本名称	网　　址	特　　点	软件包管理器
Debian Linux	www.debian.org	开放的开发模式，并且易于进行软件包升级	Apt
Fedora Core	www.redhat.com	拥有数量庞大的用户群体，有优秀的社区提供技术支持，并且有许多创新	Up2date（RPM），YUM（RPM）
CentOS	www.centos.org	CentOS是一种对RHEL（Red Hat Enterprise Linux）源代码再编译的产物，由于Linux是开放源代码的操作系统，并不排斥基于源代码的再分发，CentOS就是将商业化的Linux操作系统RHEL进行源代码再编译后分发，并在RHEL的基础上修正了不少已知的漏洞	RPM

续表

版本名称	网　　址	特　　点	软件包管理器
SUSE Linux	www.suse.com	专业的操作系统，易用的YaST软件包管理系统	YaST（RPM）和第三方Apt（RPM）软件库（Repository）
Mandriva	www.mandriva.com	操作界面友好，使用图形配置工具，有庞大的社区进行技术支持，支持NTFS分区大小的变更	RPM
KNOPPIX	www.knoppix.com	可以直接以光盘运行，具有优秀的硬件检测和适配能力，可作为系统的急救盘使用	Apt
Gentoo Linux	www.gentoo.org	高度的可定制性，使用手册较完整	Portage
Ubuntu	www.ubuntu.com	优秀、易用的桌面环境，基于Debian构建	Apt

1.4.2　内核版本的选择

内核是 Linux 操作系统最重要的部分，从最初的 Linux 0.95 版本到目前的 Linux 6.0 版本，Linux 内核开发经过了 30 多年的时间，其架构已经十分稳定。Linux 内核的编号采用如下形式：

```
主版本号.次版本号.主补丁号-次补丁号
```

例如，5.15.0-46 各数字的含义如下：

- 第 1 个数字（5）是主版本号，表示第 5 大版本。
- 第 2 个数字（15）是次版本号，偶数表示稳定版本，奇数表示开发中的版本。
- 第 3 个数字（0）是错误修补的次数。
- 第 4 个数字（46）是发型版本的补丁版本。

注意：除了前面的版本号之外，还有多种表示形式，如 5.15.0-46-generic，其中 generic 表示当前的内核版本为通用版本。

目前，Linux 内核的开发源码比较通用的是 5.x 的版本，本书中安装的环境对 Linux 内核没有特殊要求，读者在选择内核版本时不需要重新编译内核，使用操作系统自带的内核就可以满足需要。笔者的操作系统内核为 Linux-5.15.0-46-generi。

1.5　Linux 系统架构

Linux 系统从应用角度而言分为内核空间和用户空间两部分。内核空间是 Linux 操作系统的主要部分，但是仅有内核的操作系统是不能完成用户任务的，还需要丰富且功能强大的应用程序包。

1.5.1　Linux 内核的主要模块

Linux 内核主要由 5 个子系统组成，分别是进程调度、内存管理、虚拟文件系统、网络接口和进程间通信。下面依次介绍这 5 个子系统。

1．进程调度

进程调度（SCHED）指的是系统对进程的多种状态之间转换的策略。Linux 的进程调度有 3 种策略，分别是 SCHED_OTHER、SCHED_FIFO 和 SCHED_RR。

- SCHED_OTHER：针对普通进程的时间片轮转调度策略。在这种策略下，系统给所有处于运行状态的进程分配时间片。当前进程的时间片用完之后，系统从优先级最高的进程中选择进程开始运行。
- SCHED_FIFO：针对实时性要求比较高、运行时间短的进程调度策略。在这种策略中，系统按照进入队列的先后顺序进行进程的调度，在没有更高优先级进程到来或者当前进程没有因为等待资源而阻塞的情况下会一直运行。
- SCHED_RR：针对实时性要求比较高、运行时间比较长的进程调度策略。这种策略与 SCHED_OTHER 策略类似，只不过 SCHED_RR 进程的优先级要高得多。系统分配给 SCHED_RR 进程时间片，然后轮循运行这些进程，将时间片用完的进程放入队列的末尾。

由于存在多种调度方式，Linux 进程调度采用的是“有条件可剥夺”的调度方式。普通进程采用的是 SCHED_OTHER 的时间片轮循方式，实时进程的优先级高于普通进程。如果普通进程在用户空间运行，则普通进程立即停止运行，将资源让给实时进程；如果普通进程在内核空间运行，则需要等系统调用返回用户空间后方可释放资源。

2．内存管理

内存管理（MMU）是多个进程间的内存共享策略。在 Linux 系统中，内存管理的主要概念是虚拟内存。

虚拟内存可以让进程拥有比实际的物理内存更大的内存，可以是实际内存的很多倍。每个进程的虚拟内存有不同的地址空间，多个进程的虚拟内存不会发生冲突。

虚拟内存的分配策略是每个进程都可以公平地使用虚拟内存。虚拟内存的大小通常设置为物理内存的两倍。

3．虚拟文件系统

虚拟文件系统是 Linux 系统文件中一个抽象的软件层，在 Linux 系统中支持多种文件系统，如 ext2、ext3、Minix、VFAT、NTFS 和 proc 等。目前，Linux 系统中常用的文件类型是 ext2 和 ext3。ext2 为多个 Linux 发行版本的默认文件系统是 ext 文件系统的扩展。ext3 文件系统是在 ext2 的基础上增加日志功能后的进行的扩展，它兼容 ext2。这两种文件系统可以互相转换，ext2 不用格式化就可以转换为 ext3 文件系统，而 ext3 转换为 ext2 文件系统也不会丢失数据。

4．网络接口

Linux 是在 Internet 飞速发展的时期成长起来的，因为 Linux 支持多种网络接口和协议。网络接口分为网络协议和驱动程序，网络协议是一种网络传输的通信标准，而网络驱动则是硬件设备的驱动程序。Linux 支持的网络设备多种多样，几乎所有的网络设备都需要驱动程序。

5．进程间通信

Linux 操作系统支持多进程，进程之间需要进行数据交流才能完成控制和协同工作等功能，Linux 的进程间通信是从 UNIX 系统继承过来的，进程间的通信方式主要有管道、信号、消息

队列、共享内存和套接字等。

1.5.2　Linux 的文件结构

Linux 不使用磁盘分区符号来访问文件系统，而是将整个文件系统以树状结构展现出来，Linux 系统每增加一个文件系统，都会将其加入这个树中。

操作系统文件结构的开始只有一个单独的顶级目录结构，叫作根目录。一切都从“根”开始，用“/”表示，并且延伸到子目录。DOS/Windows 下的文件系统按照磁盘分区的概念分类，目录都存于分区上。Linux 则通过“挂接”的方式把所有分区都放置在“根”下的各个目录里。一个 Linux 系统的文件结构如图 1.1 所示。

不同的 Linux 发行版本的目录结构和具体实现的功能存在一些细微的差别。但是主要功能都是一致的。一些常用目录的作用如下：

- /etc：包括大多数 Linux 系统引导所需要的配置文件，系统引导时读取配置文件，按照配置文件的选项进行不同情况的启动，如 fstab 和 host.conf 等。
- /lib：包含 C 编译程序需要的函数库（如 glibc 等），是一组二进制文件。
- /usr：包括所有系统的默认软件。Linux 的内核就在 /usr/src 下。其下有子目录/bin，用于存放所有安装语言的命令，如 GCC 和 Perl 等。
- /var：包含系统定义表（如 cache），以便在系统运行改变时可以只备份该目录。
- /tmp：用于临时性的存储。
- /bin：大多数命令都存放在这里。
- /home：主要存放用户账号，并且可以支持 FTP 的用户管理。当系统管理员增加用户时，系统将在 home 目录下创建与用户同名的目录，此目录下一般默认有 Desktop 目录。
- /dev：用于存放一种设备文件的特殊文件，如 fd0 和 loop0 等。
- /mnt：专门供外挂的文件系统使用。

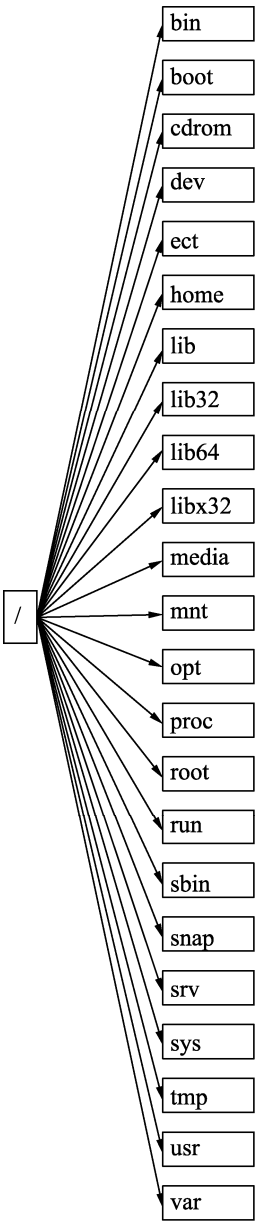

图 1.1　Linux 文件系统结构示意

1.6　GNU 通用公共许可证

GNU 通用公共许可证（简称为 GPL）是由自由软件基金会发行的用于计算机软件的一种许可证制度。GPL 最初是由 Richard Stallman 为 GNU 计划而撰写的。目前，GNU 通行证被绝

大多数的 GNU 程序和超过半数的自由软件所采用。此许可证的最新版本为“版本 3”，于 2007 年发布。GNU 宽通用公共许可证（Lesser General Public License，LGPL）是由 GPL 衍生出的许可证，被用于一些 GNU 程序库。

1.6.1　GPL 许可证的发展历史

GPL 是由 Richard Stallman 为了 GNU 计划，以 GNU 的 Emacs、GDB 和 GCC 的早期许可证为蓝本而撰写的。GPL 的版本 1，在 1989 年 1 月诞生。GPL 的版本 2 于 1991 年 6 月发布，同时另一个许可证——库通用许可证也随之发布，并记为 LGPL 版本 2，以示对 GPL 的补充。LGPL 版本 2.1 发布时与 GPL 版本不再对应，而 LGPL 也被重命名为 GNU 宽通用公共许可证。GPLv3 在 2007 年 6 月开始使用。

1.6.2　GPL 的自由理念

软件的版权保护机制在保护发明人权益的同时，也影响软件的发展。由于软件用户只具有软件的使用权，不具有对软件进一步开发的权利，因此软件的发展只能依靠软件开发公司或开发者，而不能依靠广大的用户。与此对应，GPL 授予程序的接受方（包括使用方和二次开发人员）下述权利，即 GPL 所倡导的“自由”权利：

- 可以以任何目的运行所购买的程序。
- 在得到程序代码的前提下，可以以学习为目的对源程序进行修改。
- 可以对复制件进行再发行。
- 可以对所购买的程序进行改进并公开发布。

1.6.3　GPL 的基本条款

GPL 许可证作为 Linux 平台软件的主要许可证，有很多独特的地方，其基本条款包括权利授予和 Copyleft。

1．权利授予

只要遵循 GPL 条款，不论收费软件还是免费的软件，其使用者都会获得作品的修改、复制、再发行此作品或者演绎此作品的权利，软件的使用者也可以通过上述行为收取费用。

一般的 GPL 分发软件的盈利模式是采用服务的方式，即如果使用者想更好地使用此软件，则需要向分发者提供服务报酬，分发者通过对使用者的软件进行优化或者进行人员培训等方式为使用者提供服务。

GPL 授权的另一层含义是要求分发者提供源代码，防止软件开发商对软件进行锁定，以限制用户的某些行为。如果用户获得了源代码，在分析源代码的基础上可以修改某些设置，从而对软件进行功能开发。

2．Copyleft

GPL 许可证要求接受人在进行软件再次发布的时候必须公开源代码，同时允许对再发行软件进行复制、发行和修改等，即再发行的软件必须为 GPL 许可证。

上述要求称为 Copyleft，GPL 由此被称为“被黑的版权法”。因为 GPL 的基础是承认软件是拥有版权的，即作品版权在法律上归发行者所有。由于软件的版权由发行者所有，所以发行者可以对软件的发行规定进行设置，GPL 就是发行者对版权进行的一些规定，如果某个再发行的版本不遵循 GPL 许可证，由于原作者拥有作品的版权，因此就有可能被原作者起诉。

GPL 的 Copyleft 仅仅在程序的再发行时起作用，如果接受者对软件进行修改后并没有发行，那么是可以不用开放源代码的。Copyleft 只对发行的软件起作用，对于软件的输出或者工作成果不起作用。

1.6.4　关于 GPL 许可证的争议

使用 GPL 许可证造成了很多争议，主要是对软件版权方面的界定、GPL 的软件传染性和商业开发方面的困扰等。比较有代表性的争议是对 GPL 软件产品链接库使用的产品版权的界定，即非 GPL 软件是否可以链接 GPL 的库程序。

基于 GPL 开放的源代码进行修改的产品遵循 GPL 的授权规定是很明确的，但是对于使用 GPL 链接库的产品是否需要遵循 GPL 许可证就存在分歧，有的专家并不认同这种观点，由此分成了自由和开放源代码社区两派。这个问题其实不是技术问题，而是一个法律问题，需要相关案例来例证。

1.7　Linux 软件开发的可借鉴之处

在 Linux 的发展过程中形成了一种独特的成功模式，包含软件的开发模式。例如，在《大教堂与集市》一书中对 Linux 开发模式进行了比较详细的分析，主要包含如下几个方面。

- 使用集市模式进行软件开发应该有一个基本成型的软件原型，使后来的参与者能够对此进行改进。
- 集市模式的开发把软件的使用者作为开发的协作者，而不仅仅是一个简单的用户，这样开发者和使用者能够共同对作品进行快速的代码迭代和高效率的调试。
- 集市模式开发使用早发布、常发布的方法，方便听取客户的建议，对软件进行改进。
- 集市开发模式验证了一个成功的假设：如果参与软件 Beta 版测试的人员足够多，那么软件中存在的所有问题几乎都能够被迅速地找出并进行纠正。
- 对于集市开发模式的项目来说，比技能和设计能力更重要的是项目协调人员必须具有良好的交流能力。

从 Linux 社区中还可以获得更多睿智的经验或者知识，例如 Linus 所持的一种观点：使用聪明的数据结构和笨拙的代码的搭配方式要比相反的搭配方式更好，可以作为软件开发的一种基本常识。

1.8　小　　结

本章对 Linux 的形成历史进行了简单的介绍，并对其发展过程中起重要作用的几个要素进行了解释。Linux 的发行版本数以百计，其中，Debian、Fedora Core、openSUSE 及 Ubuntu 是

比较有代表性的几种，本书均以 Ubuntu 为例进行介绍。本章还介绍了 Linux 的系统架构及其内核模块之间的关系，并对 GNU 的通用公共许可证进行了介绍，特别是 GNU 的 Copyleft 概念；最后介绍了 Linux 开发模式的成功之处并对集市开发模式进行了简单的介绍。

1.9　习　　题

一、填空题

1．Linux 的诞生和发展与_____的发展历程是紧密相关的。
2．UNIX 操作系统于 1969 年在_____实验室诞生。
3．MINIX 操作系统由荷兰的阿姆斯特丹大学的著名教授_____于 1987 年开发完成。

二、选择题

1．以下不是 UNIX 特点的选项是（　　）。
A．功能强大　　B．性能单一　　C．多用户　　D．多任务
2．以下对 Linux 文件描述错误的是（　　）。
A．/etc 包括大多数 Linux 系统引导所需要的配置文件
B．/var 包含系统定义表
C．/lib 包含 C 编译程序需要的函数库
D．/tmp 中放置了大多数的命令
3．以下不是说明 Linux 与 UNIX 异同的选项是（　　）。
A．UNIX 操作系统大多数是与硬件配套的，而 Linux 则可以运行在多种硬件平台上
B．Linux 的历史要比 UNIX 悠久
C．UNIX 操作系统是一种商业软件，而 Linux 操作系统则是一种自由软件，它是免费的
D．其他

三、判断题

1．GPL 的“版本 1”在 1989 年 3 月诞生。　　（　　）
2．Linux 的内核主要由 3 个子系统组成，分别是进程调度、内存管理和虚拟文件系统。
（　　）
3．Linux 通过“挂接”的方式把所有分区都放置在“根”下的各个目录里。　　（　　）

第 2 章　Linux 编程环境

在 Linux 环境中进行程序开发时，除了需要有一个可运行的 Linux 环境，还需要了解的基本知识有 Linux 命令行的环境和登录方式；Bash Shell 的使用。本章对 Linux 的编程环境进行介绍，通过本章的学习，读者将能够在 Linux 环境中编写、编译和调试程序。本章的主要内容如下：

- 如何使用 Linux 常用的编辑器（主要对 Vim 进行介绍）。
- 如何使用 GCC 编译程序，并进行优化和修改代码。
- 如何编写 Makefile 文件。
- 了解程序编译和执行的过程。
- 在 Linux 环境中如何使用 GDB 调试程序。

2.1　编　辑　器

在 Linux 环境中有很多编译器，例如，基于行的编辑器 Ed 和 Ex，基于文本的编辑器 Vim 和 Emacs 等。使用文本编辑器可以帮助用户进行翻页、移动光标、查找字符、替换字符和删除等操作。本节对 Vim 编辑器进行详细介绍，同时还会对其他编辑器进行简单的介绍。

2.1.1　Vim 简介

Vi 是 Visual Editor 的简写，发音为[vi'ai]，是 UNIX 系统通用的文本编辑器。Vi 不是一个所见即所得的编辑器，如果要复制和格式化文本，则需要手动输入命令进行操作。安装好 Linux 操作系统后，一般已经默认安装了 Vi 编辑器。为了使用方便，建议安装 Vi 的扩展版本 Vim，它比 Vi 更强大，更加适合初学者使用。

1．Vim的安装

在介绍如何使用 Vim 编译器之前，需要先安装 Vim 软件包，如果没有安装 Vim，可以使用如下命令进行安装。

```
#apt-get install vim                                    (使用 apt-get 命令安装 Vim)
```

在 Ubuntu 中，可以使用 apt-get 工具对系统的软件包进行管理。install 命令会自动查找和安装指定的软件包，其命令格式如下：

```
apt-get install 软件包的名字
```

2．Vim编辑器的模式

Vim 主要分为普通模式和插入模式。普通模式是命令模式，插入模式是编辑模式。

在插入模式下可以输入字符，输入的字符会显示在编辑框中。普通模式是进行命令操作的，输入的值代表一个命令。例如，在普通模式下按 h 键，光标会向左移动一个字符的位置。

插入模式和普通模式的切换分别为按 i 键和 Esc 键。在普通模式下按 i 键，会切换为插入模式；在插入模式下按 Esc 键则切换为普通模式。当用户进入 Vim 还没有进行其他操作时，操作模式是普通模式。

注意：在 Vim 编辑中输入的命令是区分大小写的。

2.1.2　使用 Vim 建立文件

Vim 的命令行格式为“vim 文件名”，文件名是所要编辑的文件名。例如，要编辑一个 hello.c 的 C 文件，可以按照以下步骤进行操作。

1．建立文件

使用 Vim 建立一个新文件的命令格式为“vim 文件名”。使用如下命令可以建立一个 hello.c 的 C 语言源文件，并同时将文件打开。

```
$vim hello.c
```

2．进入插入模式

打开文件后，默认是普通模式。按 i 键，进入插入模式，Vim 会在窗口的底部显示“--INSERT--”（中文模式下显示的是“--插入--”），表示当前模式为插入模式。

在输入文本的时候，最下边有一个指示框，告诉用户正在编辑的文件的一些信息，例如：

```
-- INSERT --                                          6,11          All
```

表示当前模式为插入模式，光标在第 6 行第 11 个字符位置上。

刚接触 Vim 的读者常常会不知道自己在什么模式下，或者不小心输入了错误的指令或进行了错误的操作。如果遇到这种情况，无论在什么模式下，要回到普通模式，只需按 Esc 键即可。有时需要按两次 Esc 键，当 Vim 发出“嘀”的一声，就表示 Vim 已经处于普通模式了。

3．文本输入

在编辑区输入如下文本：

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5  printf("Hello World!\n");
6  return 0;
7 }
```

输入第一行后，按 Enter 键开始一个新行。

4．退出Vim

编译完成后，按 Esc 键退出插入模式回到普通模式，输入“：wq”退出 Vim 编辑器。运行命令 ls：

```
$ls                                                    （查看当前目录下的文件）
hello.c
```

会发现当前目录下已经存在一个名为 hello.c 的文件。输入的 wq 是“保存后退出”的意思，其中，q 表示退出，w 表示保存。当不想保存所做的修改时，输入“:”键后，再输入“q!”，Vim 会直接退出，不保存修改。q!是强制退出的意思。

2.1.3　使用 Vim 编辑文本

Vim 的编辑命令有很多，本节选取经常使用的几个命令进行介绍。例如，如何在 Vim 中移动光标，进行字符的删除、复制、查找和转跳等操作。

1．移动光标命令h、j、k和l

Vim 在普通模式下，移动光标需要按特定的键，进行左、下、上、右光标移动操作的字符分别为 h、j、k 和 l，这 4 个字符的含义如下：

```
h       左 ←
j       下 ↓
k       上 ↑
l       右 →
```

按 h 键，光标左移一个字符的位置；按 l 键，光标右移一个字符的位置；按 k 键，光标上移一行；按 j 键，光标下移一行。

当然，还可以用方向键移动光标，但操作者必须将手从字母键位置上移动到方向键上，这样会减慢输入速度。有一些键盘是没有方向键的，需要特殊的操作才能使用方向键（如必须使用组合键）。因此，用 h、j、k、l 字符移动光标是很有帮助的。

2．删除字符命令x、dd、u和Ctrl+r

要删除一个字符，可以使用 x 键，在普通模式下，将光标移到需要删除的字符上然后按 x 键。例如，hello.c 的第一行输入有一个错误：

```
#Include <stdio.h>
```

将光标移动到 I 上，然后按 x 键，切换输入模式为插入模式，输入 i，对 include 的修正完成了。

要删除一整行，可以使用 dd 命令，删除一行后，它后面的一行内容会自动向上移动一行。使用这个命令时要注意输入两个 d 才是正确的，在实际使用过程中经常有用户只输入了一个 d。

恢复删除，可以使用 u 键。如果删除了不应该删除的信息，u 命令可以取消之前的删除。例如，用 dd 命令删除一行，再按 u 键，可以恢复被删除的该行字符。

Ctrl+r 是一个特殊的命令，作为取消一个命令，可以使用它弥补 u 命令造成的后果。例如，使用 u 命令撤销了之前的输入，重新输入字符是很麻烦的，此时使用 Ctrl+r 命令可以方便地将之前使用 u 命令撤销输入的字符重新找回。

3．复制和粘贴命令y、p

Vim 的粘贴命令是字符 p，它的作用是将内存中的字符中复制到当前光标的后面。使用 p 命令的前提是内存中有合适的字符串可以复制，例如，要将一行字符串复制到某个地方，可以用 dd 命令删除它，然后使用 u 命令恢复，这时候内存中就存在一个 dd 命令删除的字符串。将光标移动到需要插入的行之前，使用 p 命令可以把内存中的字符串复制后放置在选定的位置。

y 命令（即 yank）是复制命令，可以将指定的字符串复制到内存中，yw 命令（即 yank words）用于复制单词，可以指定复制的单词数量，y2w 可以复制两个单词。例如下面的代码：

```
1 #include <stdio.h>
```

光标位于此行的头部，当输入 y2w 时，字符串#include 就被复制到内存中，按 p 键后，此行代码如下：

```
1 ##include include <stdio.h>
```

注意：y 命令在进行字符串复制的时候包含末尾的空格。当按行复制字符串时，使用 dd 命令复制的方式比较麻烦，可以使用 yy 命令进行复制。

4．查找字符串命令“/”

查找字符串的命令是“/xxx”，其中的 xxx 代表要查找的字符串。例如，查找当前文件的 printf 字符串，可以输入以下命令进行查找。

```
":/printf"
```

按 Enter 键后，如果找到匹配的字符串，光标就停在第一个匹配的字符串上。如果要查找其他匹配的字符串，可以输入字符 n 光标会移动到下一个匹配的字符串上，输入字符 N 则光标会移动到上一个匹配的字符串上。

5．跳到某一行命令g

在编写程序或者修改程序的过程中，经常需要转跳到某一行（在编译程序出错，进行修改程序的时候经常会遇到，因为 GCC 编译器的报错信息会提示某行出错）。命令“:n”可以让光标转到某一行，其中，n 代表要跳转到的行数。例如，要跳到第 5 行，可以输入“:5”，然后按 Enter 键，光标会跳到第 5 行的头部。还有一种实现方式，即使用 nG 命令，其中，n 为要转跳的行数，如 5G 是转跳到第 5 行的命令，注意 G 为大写。

2.1.4　Vim 的格式设置

在 Vim 中可以设置很多种格式，这里仅对经常使用的格式进行介绍，如设置缩进格式、设置 Tab 键的宽度及设置行号等。

1．设置缩进格式

合理的缩进会使程序更加清晰，Vim 提供了多种方法来简化这项工作。要对 C 语言程序设置缩进格式，需要设定 cindent 选项；如果需要设置下一行的缩进长度，可以设置 shiftwidth 选项。例如，下面的命令实现 4 个空格的缩进。

```
:set cindent shiftwidth=4
```

设定好之后，当输入一行语句时，Vim 会自动在下一行进行缩进。例如，设置在 if(x)一行后面的代码自动向下一级缩进。

```
                          if (a==b)
自动缩进          --->        do_equal();
自动取消缩进      <--     if (a>b) {
自动缩进          --->        do_lt();
自动取消缩进      <--     }
```

自动缩进还能提前发现代码中的错误。例如，在输入了一个“}”后，如果发现比预想中的缩进多，那么可能是缺少了一个“}”。可以使用%命令查找与“}”相匹配的“{”。

2．设置Tab键的宽度

在进行文本编辑的时候，Tab 键可以移动一段较大的距离，不同的文本编辑器对 Tab 键移动距离的解释是不同的。在 Vim 编辑器中，Tab 键默认的移动距离为 8 个空格，当需要对这个值进行更改的时候，需要设置 tabstop 选项的值。使用命令“:set tabstop=n”可以设置 Tab 键对应空格的数量，例如，“:set tabstop=2”，表示将 Tab 键的宽度设置为 2 个空格。

3．设置行号

在程序中设置行号可以使程序更加一目了然，设置行号的命令是“:set number”，然后按 Enter 键，程序代码的头部会有一个行号数值。

2.1.5　Vim 的配置文件.vimrc

Vim 启动的时候会根据~/.vimrc 文件配置 Vi 的设置，可以修改文件.vimrc 来定制 Vim。例如，可以使用 shiftwidth 设置缩进的宽度、使用 tabstop 设置 Tab 键的宽度、使用 number 设置行号等格式来定义 Vim 的使用环境。例如，按照如下的情况对.vimrc 文件进行修改：

```
set shiftwidth=2                    #设置缩进宽度为 2 个空格
set tabstop=2                       #设置 Tab 键的宽度为 2 个空格
set number                          #显示行号
```

再次启动 Vim，对缩进宽度、Tab 键的宽度都进行了设定，并且会自动显示行号。

2.1.6　使用其他编辑器

在 Linux 中还可以使用其他编辑器，如 Gvim（Gvim 是 Vim 的 GNOME 版本）、CodeBlocks（严格来说是一个 IDE 开发环境）。

在 Linux 中进行开发并不排斥使用 Windows 环境中的编辑器，如写字板、UltraEdit、VC 的 IDE 开发环境等，保存时要注意保存为 UNIX 格式。在 Windows 环境中，换行为“\r\n”，而在 UNIX 环境中，换行为“\n”，因此，如果在 Linux 环境中用 Vim 查看 Windows 创建的文件会发现，每行的末尾有一个很奇怪的符号“～”。为了避免这个问题，可以将 Windows 编辑器创建的文件保存为 UNIX 格式，或者在 Linux 中用 dos2UNIX 命令对文件格式进行转换。例如，使用 Windows 编辑器保存的文件 hello.c 中会有“～”符号使用命令，将其转换为 UNIX 格式：

```
$dos2UNIX hello.c
```

再次查看文件 hello.c，“～”符号已经消失了。

2.2　GCC 编译器工具集

2.1 节介绍了如何使用 Linux 环境中的编辑器编写程序，并编写了一个 hello.c 程序。要使编写的程序能够运行，需要进行程序编译。本节介绍在 Linux 环境中编译器 GCC 的使用方式。

2.2.1　GCC 简介

GCC 是 Linux 中的编译工具集，是 GNU Compiler Collection 的缩写，包含 GCC、g++等编译器。这个工具集不仅包含编译器，还包含其他工具集，如 ar 和 nm 等。GCC 的 C 编译器是 GCC，其命令格式如下：

```
Usage: gcc [options] file…
```

GCC 支持默认扩展名策略，表 2.1 是 GCC 默认的文件扩展名。

表 2.1　GCC默认的文件扩展名

文件扩展名	GCC所理解的含义
*.c	该类文件为C语言的源文件
*.h	该类文件为C语言的头文件
*.i	该类文件为预处理后的C文件
*.C	该类文件为C++语言的源文件
*.cc	该类文件为C++语言的源文件
*.cxx	该类文件为C++语言的源文件
*.m	该类文件为Objective-C语言的源文件
*.s	该类文件为汇编语言的源文件
*.o	该类文件为汇编后的目标文件
*.a	该类文件为静态库
*.so	该类文件为共享库
a.out	该类文件为链接后的输出文件

GCC 有很多编译器，可以支持 C 和 C++等多种语言，表 2.2 是 GCC 常用的几个编译器。

表 2.2　GCC常用的编译器

GCC编译器命令	含　义	GCC编译器命令	含　义
cc	C语言编译器	gcc	C语言编译器
cpp	预处理编译器	g++	C++语言编译器

2.2.2　编译程序基础知识

GCC 编译器对程序的编译过程如图 2.1 所示，主要分为 4 个阶段：预编译、编译和优化、汇编和链接。GCC 的编译器可以通过指定的命令将这 4 个步骤合并成一个操作来执行。

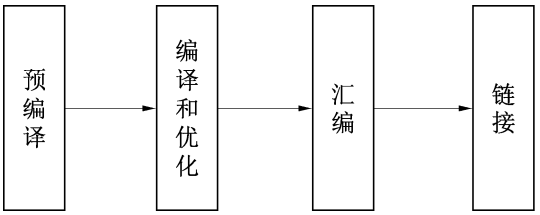

图 2.1　GCC 对程序的编译过程

源文件、目标文件和可执行文件是编译过程中经常涉及的名词。源文件通常指存放可编辑代码的文件，如存放 C、C++和汇编语言的文件。目标文件是指经过编译器编译生成的 CPU 可识别的二进制代码，但是目标文件一般不能执行，因为其中的一些函数过程没有相关的指示和说明。可执行文件就是目标文件与相关的库链接后的文件，它是可以执行的。

预编译过程是将程序中引用的头文件包含进源代码中，并对一些宏进行替换。

编译过程是将用户可识别的语言翻译成一组处理器可识别的操作码并生成目标文件，通常翻译成汇编语言，而汇编语言通常和机器操作码是一种一对一的关系。GNU 中有 C/C++编译器 GCC 和汇编器 AS。

所有的目标文件必须用某种方式组合起来才能运行，这就是链接的作用。在目标文件中通常仅对文件内部的变量和函数进行了解析，对于引用的函数和变量还没有解析，这需要将其他已经编写好的目标文件引用进来，将没有解析的变量和函数进行解析，通常引用的目标是库。链接完成后会生成可执行文件。

2.2.3　将单个文件编译成可执行文件

在 Linux 中使用 GCC 编译器编译单个文件十分简单，直接使用 gcc 命令后面加上要编译的 C 语言的源文件，GCC 会自动生成文件名为 a.out 的可执行文件。自动编译的过程包括头文件扩展、目标文件编译，链接默认的系统库生成可执行文件，最后生成系统默认的可执行程序 a.out。

下面是一个程序的源代码，代码的作用是在控制台输出"Hello World!"字符串。

```
/*hello.c*/
#include <stdio.h>                                  /*头文件包含*/
int main(void)
{
  printf("Hello World!\n");                         /*打印"Hello World!"*/
  return 0;
}
```

将代码存入 hello.c 文件，运行如下命令将代码直接编译成可以执行文件。

```
$gcc hello.c
```

在 2.2.1 小节中列出了 GCC 编译器可以识别的默认文件扩展名，通过检查 hello.c 文件的扩展名，GCC 知道这是一个 C 文件。

使用上面的编译命令进行编译时，GCC 先进行扩展名的判断，然后选择编译器。由于 hello.c 的扩展名为.c，GCC 认为这是一个 C 文件，会选择 GCC 编译器来编译 hello.c 文件。

GCC 将采取默认步骤，先将 C 文件编译成目标文件，然后将目标文件链接成可执行文件，最后删除目标文件。上述命令没有指定生成执行文件的名称，GCC 将生成默认的文件名 a.out。

运行结果如下：

```
$./a.out                                          （执行 a.out 可执行文件）
Hello World!
```

如果希望生成指定的可执行文件名，可以使用-o 选项。例如，将上述程序编译输出一个名称为 test 的执行程序：

```
$gcc -o test hello.c
```

上述命令把 hello.c 源文件编译成可执行文件 test。运行可执行文件 test，向终端输出"Hello World!"字符串。运行结果如下：

```
$./test
Hello World!
```

2.2.4　生成目标文件

目标文件是指经过编译器的编译生成的 CPU 可识别的二进制代码，因为其中一些函数过程没有相关的指示和说明，目标文件不能执行。

2.2.3 小节介绍了直接生成可执行文件的编译方法，在这种编译方法中，中间文件作为临时文件，在可执行文件生成后会被删除。在很多情况下需要生成中间的目标文件，用于不同的编译目标。

GCC 的-c 选项用于生成目标文件，这个选项将源文件生成目标文件，而不是生成可执行文件。默认情况下，生成的目标文件的文件名和源文件的名称一样，只是扩展名为.o。例如，下面的命令会生成一个名称为 hello.o 的目标文件：

```
$gcc -c hello.c
```

如果需要生成指定的文件名，可以使用-o 选项。下面的命令将源文件 hello.c 编译成目标文件，文件名为 test.o。

```
$gcc -c -o test.o hello.c
```

可以用一条命令编译多个源文件，然后生成目标文件，这种方式通常用于编写库文件或者一个项目中包含多个源文件的情况。例如，一个项目包含 file1.c、file2.c 和 file3.c，下面的命令可以将源文件生成 file1.o、file2.o 和 file3.o 3 个目标文件。

```
$gcc -c file1.c file2.c file3.c
```

2.2.5　多文件编译

GCC 可以自动编译链接多个文件，不论目标文件还是源文件，都可以使用同一个命令编译到一个可执行文件中。例如，一个项目包含 string.c 和 main.c 这两个文件，在 string.c 文件中有一个函数 StrLen()用于计算字符串的长度，而在 main.c 文件中可以调用这个函数将计算结果显示出来。

1．源文件string.c

文件 string.c 的内容如下。该文件主要包含用于计算字符串长度的函数 StrLen()。StrLen()函数的作用是计算字符串的长度，输入参数为字符串的指针，输出数值为字符串长度的计算结果。StrLen()函数将字符串中的字符与'\0'比较并进行字符长度计数，从而获得字符串的长度。

```
01  /*string.c*/
02  #define ENDSTRING '\0'              /*定义字符串*/
03  int StrLen(char *string)
04  {
05    int len = 0;
06
07    while(*string++ != ENDSTRING)     /*当*string 的值为'\0'时,停止计算*/
08      len++;
09    return len;                       /*返回此值*/
10  }
```

2．源文件main.c

在 main.c 文件中保存的是 main()函数的代码。main()函数调用 Strlen()函数计算字符串 Hello Dymatic 的长度，并将字符串的长度打印出来。代码如下：

```
01  /*main.c*/
02  #include <stdio.h>
03  extern int StrLen(char* str);       /*声明 Strlen()函数*/
04  int main(void)
05  {
06    char src[]="Hello Dymatic";       /*字符串*/
      /*计算 src 的长度并将结果打印出来*/
07    printf("string length is:%d\n",StrLen(src));
08    return 0;
09  }
```

3．编译运行

下面的命令将两个源文件中的程序编译成一个执行文件，文件名为 test。

```
$gcc -o test string.c main.c
```

执行编译的可执行文件 test，程序运行结果如下：

```
$./test
string length is:13
```

当然，也可以先将源文件编成目标文件，然后进行链接。例如，下面的代码先将 string.c 和 main.c 源文件编译成目标文件 string.o 和 main.o，然后将 string.o 和 main.o 链接生成 test。

```
$gcc -c string.c main.c
$gcc -o test string.o main.o
```

2.2.6　预处理

在 C 语言程序中，通常需要包含头文件并定义一些宏。在预处理过程中，将源文件中指定的头文件导入源文件，并且将文件中定义的宏进行扩展。

在编译程序时，选项-E，用于预编译操作。例如，下面的命令将文件 string.c 的预处理结果显示在计算机屏幕上。

```
$gcc -E string.c
```

如果需要指定源文件预编译后生成的中间结果文件名，需要使用选项-o。例如，下面的代码将文件 string.c 进行预编译，生成文件 string.i。string.i 的内容如下：

```
$gcc -o string.i -E string.c
# 0 "string.c"
```

```
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "string.c"


int StrLen(char *string)
{
  int len = 0;

  while(*string++ != '\0')
    len++;
  return len;
}
```

可以发现，之前定义的宏 ENDSTRING 已经被替换成“\0”。

2.2.7　编译成汇编语言

编译过程是将用户可识别的语言翻译成一组处理器可识别的操作码，通常翻译成汇编语言。汇编语言和机器操作码是一对一的关系。

生成汇编语言的 GCC 选项是-S，默认情况下生成的文件名和源文件一致，扩展名为.s。例如，下面的命令将 C 语言源文件 string.c 编译成汇编语言，文件名为 string.s。

```
$gcc -S string.c
```

下面是编译后的汇编语言文件 string.s 的内容。其中，第 1 行是 C 语言的文件名，第 3 行和第 4 行是函数描述，标签 StrLen 之后的代码用于实现字符串长度的计算。

```
01      .file   "string.c"
02      .text
03      .globl  StrLen
04      .type   StrLen, @function
05  StrLen:
06  .LFB0:
07      .cfi_startproc
08      endbr64
09      pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      movq    %rdi, -24(%rbp)
15      movl    $0, -4(%rbp)
16      jmp .L2
17  .L3:
18      addl    $1, -4(%rbp)
19  .L2:
20      movq    -24(%rbp), %rax
21      leaq    1(%rax), %rdx
22      movq    %rdx, -24(%rbp)
23      movzbl  (%rax), %eax
24      testb   %al, %al
25      jne .L3
26      movl    -4(%rbp), %eax
27      popq    %rbp
28      .cfi_def_cfa 7, 8
```

```
    ret
    .cfi_endproc
.LFE0:
    .size   StrLen, .-StrLen
    .ident  "GCC: (Ubuntu 11.2.0-19ubuntu1) 11.2.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
0:
    .string "GNU"
1:
    .align 8
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 8
4:
```

2.2.8　生成并使用静态链接库

静态库是 OBJ 文件的一个集合，静态库通常以“.a”作为后缀。静态库由程序 ar 生成，现在静态库已经不像以前那么普遍了，原因是程序都在使用动态库。

静态库的优点是可以在不用重新编译程序库代码的情况下，进行程序的重新链接，这种方法节省了编译的时间。静态库的另一个优势是开发者可以提供库文件给使用的人员，不用开放源代码，这是库函数提供者经常采用的手段。理论上，静态库的执行速度比共享库和动态库快 1%～5%。

1．生成静态链接库

生成静态库，或者将一个 OBJ 文件加到已经存在的静态库的命令为“ar 库文件 OBJ 文件 1 OBJ 文件 2”。创建静态库的基本步骤是生成目标文件，这一点前面已经介绍过。然后使用 ar 命令对目标文件进行归档。ar 命令的-r 选项可以创建库，并把目标文件插入指定库。例如，将 string.o 打包为库文件 libstr.a 的命令如下：

```
$ar -rcs libstr.a string.o
```

2．使用静态链接库

在编译程序的时候经常需要使用函数库，如 C 标准库等。GCC 链接时使用库函数和一般的 OBJ 文件的形式是一致的，例如对 main.c 进行链接的时候，需要使用之前已经编译好的静态链接库 libstr.a，命令格式如下：

```
$gcc -o test main.c libstr.a
```

也可以使用命令“-l 库名”，库名是不包含函数库和扩展名的字符串。例如，编译 main.c 链接静态库 libstr.a 的命令可以修改为：

```
$gcc -o test main.c -lstr
```

上面的命令将在系统默认的路径下查找 str 库函数，并把它链接到要生成的目标程序上。

可能系统会提示无法找到 str 库函数，这是由于 str 库函数没有在系统默认的查找路径下，需要显式指定库函数的路径，例如，库函数和当前编译文件在同一目录下：

```
$gcc -o test main.c -L./ -lstr
```

注意：在使用-l 选项时，-o 选项的目标名称要在-l 链接的库名称之前，否则 GCC 会认为-l 是生成的目标而出错。

2.2.9　生成动态链接库

动态链接库是程序运行时加载的库，在动态链接库正确安装后，所有的程序都可以使用动态库来运行程序。动态链接库是目标文件的集合，目标文件在动态链接库中的组织方式是按照特殊方式形成的。动态链接库中的函数和变量的地址是相对地址，不是绝对地址，其真实地址是在调用动态库的程序加载时形成的。

动态链接库的名称分为别名（soname）、真名（realname）和链接名（linkername）。别名由一个前缀 lib 加库的名称再加上一个后缀“.so”构成。真名是动态链接库的真实名称，一般是在别名的基础上加上一个小版本号和发布版本等。除此之外，还有一个链接名，即程序链接时使用的库的名称。在安装动态链接库的时候，先复制库文件到某个目录下，然后用一个软链接生成别名，在库文件进行更新的时候，仅仅更新软链接即可。

1．生成动态链接库

生成动态链接库的命令很简单，使用-fPIC 选项或者-fpic 选项即可。-fPIC 和-fpic 选项的作用是使得 GCC 生成的代码是与位置无关的。例如，下面的命令将 string.c 编译生成动态链接库：

```
$gcc -shared -Wl,-soname,libstr.so -o libstr.so.1 string.c
```

其中：-soname,libstr.so 选项表示生成动态库的别名是 libstr.so；-o libstr.so.1 选项表示生成名称为 libstr.so.1 的实际动态链接库文件；-shared 是告诉编译器生成一个动态链接库。

生成动态链接库之后，一个很重要的问题就是安装，一般情况下是将生成的动态链接库复制到系统默认的动态链接库的搜索路径下，通常为/lib、/usr/lib、/usr/local/lib，放到其中的任何一个目录下都可以。

2．动态链接库的配置

动态链接库不能随意使用，如果要在运行的程序中使用动态链接库，则需要指定系统的动态链接库搜索的路径，让系统找到运行所需的动态链接库才可以。系统中的配置文件/etc/ld.so.conf 是动态链接库的搜索路径配置文件。在这个文件内，存放着可被 Linux 共享的动态链接库所在目录的名称（系统目录/lib、/usr/lib 除外），多个目录名之间以空白字符（空格、换行等）、冒号或逗号分隔。下面的命令是查看系统中的动态链接库配置文件的内容：

```
$ cat /etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
```

Ubuntu 的配置文件将目录/etc/ld.so.conf.d 中的配置文件包含进来，下面的命令查看这个目录下的文件：

```
$ ls  /etc/ld.so.conf.d/
libc.conf  x86_64-linux-gnu.conf
```

3．动态链接库管理命令

为了让新增加的动态链接库能够被系统共享，需要运行动态链接库的管理命令 ldconfig。ldconfig 命令的作用是在系统的默认搜索路径和动态链接库配置文件列出的目录里搜索动态链接库，创建动态链接装入程序所需要的链接和缓存文件。搜索完毕后，将结果写入缓存文件/etc/ld.so.cache，在该文件中保存的是已经排好序的动态链接库名称列表。ldconfig 命令行的用法如下，其各选项的含义参见表 2.3。

```
ldconfig [-v|--verbose] [-n] [-N] [-X] [-f CONF] [-C CACHE] [-r ROOT] [-l]
[-p|--print-cache] [-c FORMAT] [--format=FORMAT] [-V] [-?|--help|--usage]
path…
```

表 2.3　ldconfig命令的选项及其含义

选　　项	含　　义
-v	打印ldconfig的当前版本号，显示扫描的每个目录和动态链接库
-n	处理命令行指定的目录，不对系统的默认目录/lib、/usr/lib进行扫描，也不对配置文件/etc/ld.so.conf中指定的目录进行扫描
-N	不会重建缓存文件
-X	不更新链接
-f CONF	使用用户指定的配置文件代替默认文件/etc/ld.so.conf
-C CACHE	使用用户指定的缓存文件代替系统默认的缓存文件/etc/ld.so.cache
-r ROOT	改变当前应用程序的根目录
-l	手动链接单个动态链接库
-p或--print-cache	打印缓存文件中共享库的名称

如果想知道系统中有哪些动态链接库，可以使用 ldconfig 命令的-p 选项来列出缓存文件中的动态链接库列表。下面的命令表明在系统缓存中共有 893 个动态链接库。

```
$ ldconfig -p                                          （列出当前系统中的动态链接库）
在缓存“/etc/ld.so.cache”中找到 893 个库          （缓存中的动态链接库的数目）
    libzstd.so.1 (libc6,x86-64) => /lib/x86_64-linux-gnu/libzstd.so.1
    libz.so.1 (libc6,x86-64) => /lib/x86_64-linux-gnu/libz.so.1
    libyelp.so.0 (libc6,x86-64) => /lib/x86_64-linux-gnu/libyelp.so.0
    libyaml-0.so.2 (libc6,x86-64) => /lib/x86_64-linux-gnu/libyaml-0.so.2
    libyajl.so.2 (libc6,x86-64) => /lib/x86_64-linux-gnu/libyajl.so.2
    …
```

使用 ldconfig 命令，默认情况下并不会输出扫描的结果。使用-v 选项可以将 ldconfig 命令在执行过程中扫描到的目录和共享库信息输出到终端。执行 ldconfig 命令后，将刷新缓存文件/etc/ld.so.cache。

```
$ ldconfig -v
/lib/x86_64-linux-gnu: (from /etc/ld.so.conf.d/x86_64-linux-gnu.conf:3)
    libnss_mdns_minimal.so.2 -> libnss_mdns_minimal.so.2
    libVkLayer_MESA_device_select.so -> libVkLayer_MESA_device_select.so
    …
```

当用户没有在系统动态链接库配置文件/etc/ld.so.conf 中指定目录时，可以使用 ldconfig 命令显式指定要扫描的目录，将用户指定目录中的动态链接库放入系统中共享。命令格式如下：

```
ldconfig 目录名
```

上面是使用 ldconfig 命令将指定的目录名称中的动态链接库放入系统的缓存文件/etc/ld.so.cache 中，从而可以被系统共享使用。下面的代码是扫描当前用户的 lib 目录，将其中的动态链接库加入系统：

```
$ ldconfig ~/lib
```

注意：执行上述命令后，如果再次执行 ldconfig 命令而没有加参数，那么系统会将/lib、/usr/lib 及/etc/ld.so.conf 下指定目录中的动态库加入缓存，这时候上述代码中的动态链接库就不能被系统共享了。

4．使用动态链接库

在编译程序时，链接动态链接库和静态链接库的方式是相同的，使用“-l 库名”的方式，在生成可执行文件的时候会链接库文件。例如，下面的命令将源文件 main.c 编译成可执行文件 test，并链接库文件 libstr.a 或者 libstr.so：

```
$gcc -o test main.c -L./ -lstr
```

其中，-L 指定链接动态链接库的路径，-lstr 表示链接库函数 str。

注意：如果在系统的搜索路径下同时存在静态链接库和动态链接库，则默认情况下会链接动态链接库。如果需要强制链接静态链接库，则需加上-static 选项。上面的编译可改为如下方式：

```
$gcc -o test main.c -static -lstr
```

2.2.10　动态加载库

动态加载库和一般的动态链接库的区别是，一般的动态链接库在程序启动的时候就要寻找动态库，找到库函数；而动态加载库可以用程序的方法来控制什么时候加载。动态加载库的主要函数有 dlopen()、dlerror()、dlsym()和 dlclose()。

1．打开动态链接库函数dlopen()

dlopen()函数按照用户指定的方式打开动态链接库，其中，参数 filename 为动态链接库的文件名，flag 为打开方式，一般为 RTLD_LASY，该函数的返回值为库的指针。dlopen()函数的原型如下：

```
void * dlopen(const char *filename, int flag);
```

例如，下面的代码使用 dlopen()函数打开当前目录下的动态链接库 libstr.so。

```
void *phandle = dlopen("./libstr.so", RTLD_LAZY);
```

2．获得函数指针的函数dlsym()

使用动态链接库的目的是调用其中的函数并完成特定的功能。dlsym()函数可以获得动态链接库中指定的函数指针，然后可以使用这个函数指针进行操作。dlsym()函数的原型如下：

```
void * dlsym(void *handle, const char *symbol);
```

其中，参数 handle 为 dlopen()打开动态库后返回的句柄，参数 symbol 为函数的名称，返回

值为函数指针。

3．动态加载库的使用示例

下面是一个动态加载库的使用例子。首先使用 dlopen()函数打开动态链接库，可以使用 dlerror()函数判断是否正确打开。如果上面的过程正常，则使用 dlsym()函数获得动态链接库中的某个函数，并可以使用这个函数来完成某些功能。代码如下：

```
01  /*动态加载库示例*/
02  #include <dlfcn.h>                              /*动态加载库库头*/
03  int main(void)
04  {
05    char src[]="Hello Dymatic";                   /*要计算的字符串*/
06    int (*pStrLenFun)(char *str);                 /*函数指针*/
07    void *phandle = NULL;                         /*库句柄*/
08    char *perr = NULL;                            /*错误信息指针*/
      /*打开 libstr.so 动态链接库*/
09    phandle = dlopen("./libstr.so", RTLD_LAZY);
10    /*判断是否正确打开*/
11    if(!phandle)                                  /*打开错误*/
12    {
13      printf("Failed Load library!\n");           /*打印库不能加载信息*/
14    }
15    perr = dlerror();                             /*读取错误值*/
16    if(perr != NULL)                              /*存在错误*/
17    {
18      printf("%s\n",perr);
19      return 0;                                   /*正常返回*/
20    }
21
22    pStrLenFun = dlsym(phandle, "StrLen");        /*获得函数 StrLen 的地址*/
23    perr = dlerror();                             /*读取错误信息*/
24    if(perr != NULL)                              /*存在错误*/
25    {
26      printf("%s\n",perr);                      /*打印错误函数获得的错误信息*/
27      return 0;                                   /*返回*/
28    }
29
      /*调用函数 pStrLenFunc 计算字符串的长度*/
30    printf("the string length is: %d\n",pStrLenFun(src));
31    dlclose(phandle);                             /*关闭动态加载库*/
32    return 0;
33  }
```

编译上述文件的时候需要链接动态库 libdl.so，使用如下命令将上述代码（main.c 文件）编译成可执行文件 testdl。并链接动态链接库 libdl.so。

```
$gcc -o testdl main.c libstr.so -ldl
```

执行文件 testdl 的结果如下：

```
$./testdl
string length is:13
```

使用动态加载库和动态链接库的结果是一样的。

2.2.11　GCC 的常用选项

除了前面介绍的基本功能外，GCC 的选项配置也很重要，如头文件路径、加载库路径、警告信息及调试等。本小节对 GCC 的常用选项进行介绍。

1．-DMACRO选项

在多种预定义的程序中经常需要定义一个宏。以下代码根据系统是否定义 Linux 宏来执行不同的代码段。使用-D 选项可以选择不同的代码段，例如，-DOS_LINUX 选项将执行代码段①。

```
#ifdef OS_LINUX
…代码段①
#else
…代码段②
#endif
```

- -Idir：将头文件的搜索路径扩大，包含 dir 目录。
- -Ldir：将链接时使用的链接库搜索路径扩大，包含 dir 目录。GCC 都会优先使用共享程序库。
- -static：仅选用静态程序库进行链接，如果在一个目录中静态库和动态库都存在，则仅选用静态库。
- -g：包括调试信息。
- -On：优化程序，程序优化后执行速度会更快，程序的占用空间会更小。通常，GCC 会进行很小的优化，优化的级别可以选择，最常用的优化级别是 2。
- -Wall：打开所有 GCC 能够提供的、常用的警告信息。

2．GCC的常用选项及其含义

表 2.4 是 GCC 的常用选项及其含义，可以在编译程序的时候对 GCC 的选项进行设置，以编写质量高的代码。

表 2.4　GCC常用的编译选项及其含义

GCC的警告选项		含　义
-Wall选项集合	-Wchar-subscripts	针对数组的下标值，如果下标值是char类型则给出警告
	-Wcomment	针对代码中的注释，如果出现不合适的注释格式则会出现警告
	-Wformat	针对输入/输出格式，检查字符串与参数类型的匹配情况
	-Wimplicit	针对函数的声明
	-Wmissing-braces	针对结构类型或者数组初始化时的不合适格式
	-Wparentheses	针对多种优先级的操作符在一起或者代码结构难以看明白的操作
	-Wsequence-point	针对顺序点，对代码中使用了可能会引起顺序点变化的语句给出警告
	-Wswitch	针对switch语句，如果没有default条件会给出警告
	-Wunused	针对代码中没有用到的变量、函数、值、转跳点等
	-Wunused-parameter	针对函数参数，函数的参数在函数实现中没有用到会给出警告
	-Wuninitialized	针对初始化变量，局部变量使用之前没有初始化会给出警告

续表

GCC的警告选项		含　义
非-Wall警告选项	-Wflot-equal	针对浮点值相等的判定，出现在相等判定的表达式中给出警告
	-Wshadow	判断局部变量作用域内是否有同名变量，如果有则给出警告
	-Wbad-function-cast	针对函数的返回值，当函数的返回值赋予不匹配的类型时给出警告
	-Wsign-compare	针对有符号数和无符号数的比较
	-Waggregate-return	针对结构类型的函数返回值，当返回值为结构、联合等类型时给出警告
	-Wmultichar	针对字符类型变量的错误赋值，如果赋值错误则给出警告
	-Wunreachable-code	针对冗余代码，如果在代码中有执行不到的代码则给出警告
其他	-Wtraditional	选项traditional试图支持传统C编译器的某些功能
ANSI兼容	-ansi	与ANSI的C语言兼容
	-pedantic	允许发出ANSI/ISO C标准列出的所有警告
	-pedantic-errors	允许发出ANSI/ISO C标准列出的所有错误
编译检查	-fsyntax-only	仅进行编译检查而不实际编译程序

注意：在编写代码时，不好的习惯会使程序在执行过程中发生错误。好的习惯是使用编译选项将代码的警告信息显示出来，并对代码进行改正。例如，使用编译选项-Wall 和-W 显示所有的警告信息，甚至可以更严格一些，使用-Werror 选项将编译时的警告信息作为错误信息来处理，中断编译。

2.2.12　搭建编译环境

在安装 Ubuntu 后，可以使用 which 命令查看系统中是否已经安装了 GCC。

```
$which gcc
```

如果不存在，使用 apt-get 命令可以获得 gcc 包然后进行安装：

```
$apt-get install gcc
```

如果读者对 C++感兴趣，可以安装 g++。在编译器安装完毕后，可以使用 GCC 进行程序的编译。

2.3　Makefile 文件简介

使用 GCC 的命令行进行程序编译对单个文件是比较方便的，当工程中的文件逐渐增多，甚至变得十分庞大时，使用 GCC 命令编译就会变得力不从心。Linux 中的 make 工具提供了一种管理工程的功能，可以方便地进行程序编译，也可以对更新的文件进行重新编译。

2.3.1　多文件工程实例

有一个工程文件的目录结构，如图 2.2 所示。工程共有 5 个文件，在 add 目录下有 add_int.c

和 add_float.c 两个文件，分别用于计算整型和浮点型数值的相加；在 sub 目录下有 sub_int.c 和 sub_float.c 两个文件，分别用于计算整型和浮点型数值的相减；顶层目录下的 main.c 文件负责整个程序。

工程代码分别存放在 add/add_int.c、add/add_float.c、add/add.h、sub/sub_int.c、sub/sub_float.c、sub/sub.h 和 main.c 下。

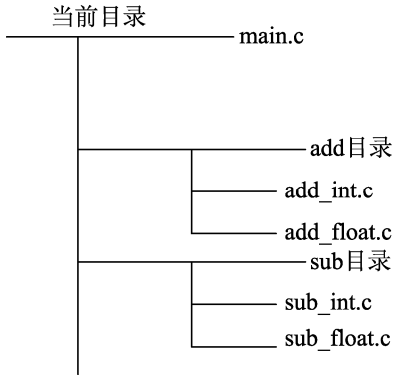

图 2.2　工程文件的目录结构

1．main.c文件

main.c 文件的代码如下。在 main()函数中调用整型和浮点型数值的加减运算函数进行计算。

```
01  /*main.c*/
02  #include <stdio.h>
03  /*需要包含的头文件*/
04  #include "add.h"
05  #include "sub.h"
06  int main(void)
07  {
08      /*声明计算所用的变量，*a、b 为整型，x、y 为浮点型*/
09      int a = 10, b = 12;
10      float x= 1.23456,y = 9.87654321;
11
12      /*调用函数并将计算结果打印出来*/
13      printf("int a+b IS:%d\n",add_int(a,b));      /*计算整型数值相加*/
14      printf("int a-b IS:%d\n",sub_int(a,b));      /*计算整型数值相减*/
15      printf("float x+y IS:%f\n",add_float(x,y));  /*计算浮点型数值相加*/
16      printf("float x-y IS:%f\n",sub_float(x,y));  /*计算浮点型数值相减*/
17      return 0;
18  }
```

2．相加操作

add.h 文件的代码如下，包含整型和浮点型数值的求和函数声明。

```
01  /*add.h*/
02  #ifndef __ADD_H__
03  #define __ADD_H__
04  /*整型和浮点型数值相加的声明*/
05  extern int add_int(int a, int b);
06  extern float add_float(float a, float b);
07  #endif /*__ADD_H__*/
```

add_float.c 文件的代码如下，add_float()函数进行浮点型数值的相加计算。

```
01  /*add_float.c*/
02  /*浮点数求和函数*/
03  float add_float(float a, float b)
04  {
05      return a+b;
06  }
```

add_int.c 文件的代码如下，add_int()函数进行整型数值的相加计算。

```
01  /*add_int.c*/
02  /*整数求和函数*/
```

```
03  int add_int(int a, int b)
04  {
05      return a+b;
06  }
```

3．相减操作

sub.h 文件的代码如下，包含整型和浮点型数值的相减函数声明：

```
01  /*sub.h*/
02  #ifndef __SUB_H__
03  #define __SUB_H__
04  /*整型和浮点型数值相减的声明*/
05  extern float sub_float(float a, float b);
06  extern int sub_int(int a, int b);
07  #endif /*__SUB_H__*/
```

sub_int.c 文件的代码如下，sub_int()函数进行整型数值的相减计算。

```
01  /*sub_int.c*/
02  /*整型数值相减函数*/
03  int sub_int(int a, int b)
04  {
05      return a-b;
06  }
```

sub_float.c 文件的代码如下，sub_float()函数进行浮点型数值的相减计算。

```
01  /*sub_float.c*/
02  /*浮点型数值相减函数*/
03  float sub_float(float a, float b)
04  {
05      return a-b;
06  }
```

2.3.2　多文件工程的编译

将 2.3.1 小节中的多文件工程编译成可执行文件有两种方法，一种是命令行操作，手动输入命令将源文件编译为可执行文件；另一种是编写 Makefile 文件，通过 make 命令将多个文件编译为可执行文件。

1．通过命令行编译程序

要将文件编译为可执行文件 cacu，使用 GCC 进行手动编译是比较麻烦的。例如，下面的编译方式是每行编译一个 C 文件，生成目标文件，最后将 5 个目标文件编译成可执行文件。

```
$gcc -c add/add_int.c -o add/add_int.o       （生成 add_int.o 目标函数）
$gcc -c add/add_float.c -o add/add_float.o   （生成 add_float.o 目标函数）
$gcc -c sub/sub_int.c -o sub/sub_int.o       （生成 sub_int.o 目标函数）
$gcc -c sub/sub_float.c -o sub/sub_float.o   （生成 sub_float.o 目标函数）
$gcc -c main.c -o main.o                     （生成 main.o 目标函数）
$gcc -o cacu add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o
main.o（链接生成 cacu）
```

或者使用 GCC 的默认规则，使用一条命令直接生成可执行文件 cacu：

```
$gcc -o cacu add/add_int.c add/add_float.c sub/sub_int.c sub/sub_float.c
main.c
```

2．适用于多文件的Makefile方法

利用上面的命令直接产生可执行文件的方法是比较容易的。但是当频繁修改源文件或者项目中的文件比较多、关系比较复杂时，用 GCC 直接编译就会变得十分困难。

使用 make 命令进行项目管理，需要一个 Makefile 文件，make 命令在编译的时候，从 Makefile 文件中读取设置情况，进行解析后运行相关的规则。使用 make 命令查找当前目录下的文件 Makefile 或者 makefile，按照其规则运行。例如，建立一个如下规则的 Makefile 文件。

```
#生成 cacu，":"右边为目标
cacu:add_int.o add_float.o sub_int.o sub_float.o main.o
    gcc -o cacu add/add_int.o add/add_float.o \
           sub/sub_int.o sub/sub_float.o main.o
#生成 add_int.o 的规则，将 add_int.c 编译成目标文件 add_int.o
add_int.o:add/add_int.c add/add.h
    gcc -c -o add/add_int.o add/add_int.c
#生成 add_float.o 的规则
add_float.o:add/add_float.c add/add.h
    gcc -c -o add/add_float.o add/add_float.c
#生成 sub_int.o 的规则
sub_int.o:sub/sub_int.c sub/sub.h
    gcc -c -o sub/sub_int.o sub/sub_int.c
#生成 sub_float.o 的规则
sub_float.o:sub/sub_float.c sub/sub.h
    gcc -c -o sub/sub_float.o sub/sub_float.c
#生成 main.o 的规则
main.o:main.c add/add.h sub/sub.h
    gcc -c -o main.o main.c -Iadd -Isub

#清理的规则
clean:
    rm -f cacu add/add_int.o add/add_float.o \
           sub/sub_int.o sub/sub_float.o main.o
```

3．多文件的编译

编译多文件的项目，在上面的 Makefile 文件编写完毕后运行 make 命令：

```
$make
gcc -c -o add/add_int.o add/add_int.c
gcc -c -o add/add_float.o add/add_float.c
gcc -c -o sub/sub_int.o sub/sub_int.c
gcc -c -o sub/sub_float.o sub/sub_float.c
gcc -c -o main.o main.c -Iadd -Isub
gcc -o cacu add/add_int.o add/add_float.o \
                    sub/sub_int.o sub/sub_float.o main.o
```

在 add 目录下生成了 add_int.o 和 add_float.o 两个目标文件，在 sub 目录下生成了 sub_int.o 和 sub_float.o 两个文件，在主目录下生成了 main.o 目标文件，并生成了 cacu 最终文件。

默认情况下会执行 Makefile 中的第一个规则，即 cacu 相关的规则，而 cacu 规则依赖于多个目标文件 add_int.o、add_float.o、sub_int.o、sub_float.o、main.o。编译器先生成上述目标文件，然后执行下面的命令：

```
$(CC) -o $(TARGET) $(OBJS) $(CFLAGS)
```

上面的命令将多个目标文件编译成可执行文件 cacu，即：

```
gcc -o cacu add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o
```

```
main.o -Iadd -Isub -O2
```

命令 make clean 会调用 clean 相关的规则，清除编译出来的目标文件及 cacu。例如：

```
$make clean
rm -f cacu add/add_int.o add/add_float.o \
                    sub/sub_int.o sub/sub_float.o main.o
```

clean 规则会执行-(RM)(TARGET)$(OBJS)命令，将定义的变量扩展：

```
rm -f cacu add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o
main.o
```

2.3.3　Makefile 的规则

Makefile 框架是由规则构成的。make 命令执行时先在 Makefile 文件中查找各种规则，对各种规则进行解析后运行规则。规则的基本格式如下：

```
TARGET… : DEPENDEDS…
   COMMAND
   …
   …
```

- TARGET：规则所定义的目标。通常，规则是最后生成的可执行文件的文件名或者为了生成可执行文件而依赖的目标文件的文件名，也可以是一个动作，称为“伪目标”。
- DEPENDEDS：执行此规则的必要依赖条件，如生成可执行文件的目标文件。DEPENDEDS 也可以是某个 TARGET，这样就形成了 TARGET 之间的嵌套。
- COMMAND：规则所执行的命令，即规则的动作，如编译文件、生成库文件、进入目录等。动作可以是多个，每个命令占一行。

规则的形式比较简单，要写好一个 Makefile 需要注意一些细节，并且对执行过程要有所了解。

1．规则的书写

在书写规则时，为了使 Makefile 更加清晰，要用反斜杠（\）将较长的行分解为多行。例如，将“rm-f cacu add/add_int.o add/add_float.o\sub/sub_int.o sub/sub_float.o main.o”分解为了两行。

命令行必须以 Tab 键开始，make 程序把出现在一条规则之后的所有连续的以 Tab 键开始的行都作为命令行来处理。

注意：规则书写时要注意 COMMAND 的位置，COMMAND 前面的空白是一个 Tab 键，不是空格。Tab 键告诉 make 程序这是一个命令行，make 程序执行相应的动作。

2．目标

Makefile 的目标可以是具体的文件，也可以是某个动作。例如，目标 cacu 就是生成 cacu 的规则，有很多的依赖项及相关的命令动作，而 clean 是清除当前生成文件的一个动作，不会生成任何目标项。

3．依赖项

依赖项是生成目标必须满足的条件。例如，生成 cacu 需要依赖 main.o，main.o 必须存在才

能执行生成 cacu 的命令，即依赖项的动作在 TARGET 的命令之前执行。依赖项之间的顺序按照自左向右的顺序检查或者执行。例如下面的规则：

```
main.o:main.c add/add.h sub/sub.h
    gcc -c -o main.o main.c -Iadd -Isub
```

main.c、add/add.h 和 sub/sub.h 必须都存在才能执行动作“gcc-c-o main.o main.c- Iadd-Isub”。当 add/add.h 不存在时，是不会执行规则的命令动作的，而且也不会检查 sub/sub.h 文件是否存在，当然，由于 main.c 在 add/add.h 依赖项之前，会先确认此项没有问题。

4．规则的嵌套

规则之间是可以嵌套的，这通常借助依赖项来实现。例如，生成 cacu 规则依赖于很多的.o 文件，而每个.o 文件又是一个规则。要执行 cacu 规则，必须先执行它的依赖项，即 add_int.o、add_float.o、sub_int.o、sub_float.o、main.o，这 5 个依赖项生成之后或者已经存在，才执行 cacu 的命令动作。

5．文件的时间戳

make 命令根据文件的时间戳判定是否执行相关的命令，以及依赖于此项的规则。例如，对 main.c 文件进行修改后保存，此时 main.c 对应的目标文件 main.o 的生成日期就早于 main.c 文件。当再次调用 make 命令编译时，就会重新生成 main.c 的目标文件，否则不会编译 main.c 文件。

6．执行的规则

当调用 make 命令编译程序时，make 程序会查找 Makefile 文件中的第 1 个规则，分析并执行相关的动作。例子中的第 1 个规则为 cacu，因此 make 程序执行 cacu 规则。由于其依赖项有 5 个，第 1 个为 add_int.o，分析其依赖项，如果 add/add_int.c add.h 存在，则执行如下命令动作：

```
gcc -c -o add/add_int.o add/add_int.c
```

命令执行完毕后，会按照顺序执行第 2 个依赖项，生成 add/add_flaot.o。当第 5 个依赖项满足时，即 main.o 生成的时候，会执行 cacu 的命令，链接生成执行文件 cacu。

当把规则 clean 放到第一个规则的位置上时，再执行 make 命令不是生成 cacu 文件，而是清理文件。要生成 cacu 文件，需要使用如下的 make 命令。

```
$make cacu
```

7．模式匹配

在上面的 Makefile 中，main.o 规则的书写方式如下：

```
main.o:main.c add/add.h sub/sub.h
    gcc -c -o main.o main.c -Iadd -Isub
```

有一种简便的方法可以实现与上面的书写方式相同的功能：

```
main.o:%o:%c
    gcc -c $< -o $@
```

在规则 main.o 中，依赖项中的“%o:%c”的作用是将 TARGET 域的.o 的扩展名替换为.c，即将 main.o 替换为 main.c。命令行中的$<表示依赖项的结果，即 main.c；$@表示 TARGET 域的名称，即 main.o。

2.3.4　在 Makefile 中使用变量

在 2.3.2 小节的 Makefile 中，生成的 cacu 规则如下：

```
cacu:add_int.o add_float.o sub_int.o sub_float.o main.o
    gcc -o cacu add/add_int.o add/add_float.o \
           sub/sub_int.o sub/sub_float.o main.o
```

生成 cacu 规则时，多次使用了同一组.o 目标文件：在 cacu 规则的依赖项中出现了一次，在生成 cacu 执行文件时又出现了一次。直接使用文件名的方法不仅书写麻烦，而且增加或者删除文件时容易遗忘。例如，增加一个 mul.c 文件，需要修改依赖项和命令行两个部分。

1．Makefile中的用户自定义变量

使用 Makefile 进行规则定义的时候，用户可以定义自己的变量，称为用户自定义变量。例如，可以用变量来表示上述文件名，定义 OBJS 变量表示目标文件：

```
OBJS = add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o main.o
```

在调用 OBJS 时前面加上$，并且将变量的名称用括号括起来。例如，使用 GCC 的默认规则进行编译，cacu 规则可以采用如下形式：

```
cacu:
    gcc -o cacu $(OBJS)
```

用 CC 变量表示 GCC，用 CFLAGS 表示编译的选项，RM 表示 rm-f，TARGET 表示最终的生成目标 cacu。

```
CC = gcc                                          (CC 定义成 GCC)
CFLAGS = -Isub -Iadd                              (加入头文件搜索路径 sub 和 add)
TARGET = cacu                                     (最终生成目标)
RM = rm -f                                        (删除的命令)
```

之前冗长的 Makefile 可以简化成如下方式：

```
CC = gcc
CFLAGS = -Iadd -Isub -O2                          (O2 为优化)
OBJS = add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o main.o
TARGET = cacu
RM = rm -f
$(TARGET):$(OBJS)
    $(CC) -o $(TARGET) $(OBJS) $(CFLAGS)
$(OBJS):%.o:%.c          (将 OBJS 中所有扩展名为.o 的文件替换成扩展名为.c 的文件)
    $(CC) -c $(CFLAGS) $< -o $@                   (编译生成目标文件)
clean:
    -$(RM) $(TARGET) $(OBJS)
```

执行命令的情况如下：

```
$ make
#编译 add_int.c 为目标文件
gcc -c -Iadd -Isub -O2 add/add_int.c -o add/add_int.o
#编译 add_float.c 为目标文件
gcc -c -Iadd -Isub -O2 add/add_float.c -o add/add_float.o
#编译 sub_int.c 为目标文件
gcc -c -Iadd -Isub -O2 sub/sub_int.c -o sub/sub_int.o
#编译 sub_float.c 为目标文件
gcc -c -Iadd -Isub -O2 sub/sub_float.c -o sub/sub_float.o
```

```
gcc -c -Iadd -Isub -O2 main.c -o main.o          #编译 main.c 为目标文件
#将文件 add_int.o、add_float.o、sub_int.o、sub_float.o、main.o 链接成 cacu 可执
 行文件，并指定默认的头文件搜索目录 add 和 sub，编译的优化选项为 O2
gcc -o cacu add/add_int.o add/add_float.o sub/sub_int.o sub/sub_float.o
main.o -Iadd -Isub -O2
```

执行 make 命令查找到第一个执行的规则为生成 cacu，但是 main.o 等 5 个文件不存在，因此 make 命令按照默认的规则生成 main.o 等 5 个目标文件。

2．Makefile中的预定义变量

在 Makefile 中有一些已经定义的变量，用户可以直接使用这些变量，无须进行定义。在进行编译时，在某些条件下 Makefile 会使用这些预定义变量的值进行编译。在 Makefile 中经常使用的预定义变量如表 2.5 所示。

表 2.5　在Makefile中经常使用的预定义变量及其含义

变　量　名	含　　义	默　认　值
AR	生成静态库库文件的程序名称	ar
AS	汇编编译器的名称	as
CC	C语言编译器的名称	cc
CPP	C语言预编译器的名称	$(CC) -E
CXX	C++语言编译器的名称	g++
FC	FORTRAN语言编译器的名称	f77
RM	删除文件程序的名称	rm -f
ARFLAGS	生成静态库库文件程序的选项	无默认值
ASFLAGS	汇编语言编译器的编译选项	无默认值
CFLAGS	C语言编译器的编译选项	无默认值
CPPFLAGS	C语言预编译的编译选项	无默认值
CXXFLAGS	C++语言编译器的编译选项	无默认值
FFLAGS	FORTRAN语言编译器的编译选项	无默认值

在 Makefile 中经常用变量 CC 表示编译器，其默认值为 cc，即使用 cc 命令进行 C 语言程序的编译；当进行程序删除时，经常使用的命令是 RM，它的默认值为 rm -f。

另外，CFLAGS 等默认值是调用编译器时的默认选项配置。例如，修改后的 Makefile 生成 main.o 时，没有指定编译选项，make 程序自动调用了文件中定义的 CFLAGS 选项-Iadd-Isub-O2 来增加头文件的搜索路径，并在所有的目标文件中都采用了此设置。经过简化之后，之前的 Makefile 可以采用如下形式：

```
CC = gcc
CFLAGS = -Iadd -Isub -O2                          #编译选项
OBJS = add/add_int.o add/add_float.o \            #目标文件
          sub/sub_int.o sub/sub_float.o main.o
TARGET = cacu                                     #生成的可执行文件
RM = rm -f
$(TARGET):$(OBJS)                                 #TARGET 目标，需要先生成 OBJS 目标
    $(CC) -o $(TARGET) $(OBJS) $(CFLAGS)          #生成可执行文件
clean:                                            #清理
    -$(RM) $(TARGET) $(OBJS)                      #删除所有的目标文件和可执行文件
```

在上面的 Makefile 中，clean 目标中的(RM)(TARGET)$(OBJS)前面的符号“-”表示当操作失败时不报错，命令继续执行。如果当前目录不存在 cacu，则会继续删除其他目标文件。例如，下面的 clean 规则在没有 cacu 文件时会报错。

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

执行 clean 命令：

```
$make clean
rm cacu
rm: 无法删除 'cacu': 没有那个文件或目录        （删除 cacu 文件失败）
make: *** [Makefile:10: clean] 错误 1
```

3．Makefile中的自动变量

Makefile 中的变量除了用户自定义变量和预定义变量外，还有一类自动变量。在 Makefile 编译语句中经常会出现目标文件和依赖文件，自动变量代表这些目标文件和依赖文件。表 2.6 是在 Makefile 中常见的一些自动变量。

表 2.6　Makefile中常见的自动变量及其含义

变　量	含　义
$*	目标文件的名称，不包含目标文件的扩展名
$+	所有的依赖文件，这些依赖文件之间以空格分开，以出现的先后次序为顺序，其中可能包含重复的依赖文件
$<	依赖项中第一个依赖文件的名称
$?	依赖项中，所有目标文件时间戳晚的依赖文件，依赖文件之间以空格分开
$@	目标项中目标文件的名称
$^	依赖项中，所有不重复的依赖文件，这些文件之间以空格分开

按照表 2.6 中的说明对 Makefile 进行重新编写，代码如下：

```
CFLAGS = -Iadd -Isub -O2                          #编译选项
OBJS = add/add_int.o add/add_float.o \            #目标文件
          sub/sub_int.o sub/sub_float.o main.o
TARGET = cacu                                     #生成的可执行文件
$(TARGET):$(OBJS)                                 #TARGET 目标，需要先生成 OBJS 目标
    $(CC) $^ -o $@ $(CFLAGS)                      #生成可执行文件
$(OBJS):%.o:%.c                                   #目标文件的选项
    $(CC) $< -c $(CFLAGS) -o $@                   #采用 CFLAGS 指定的选项生成目标文件
clean:                                            #清理
    -$(RM) $(TARGET) $(OBJS)                      #删除所有的目标文件和可执行文件
```

在重新编写后的 Makefile 中，生成 TARGET 规则的编译选项使用$@来表示依赖项中的文件名称，使用$<表示目标文件的名称。下面的命令：

```
$(TARGET):$(OBJS)                                 #TARGET 目标，需要先生成 OBJS 目标
    $(CC) $^ -o $@ $(CFLAGS)                      #生成可执行文件
```

与下面的命令的效果是一样的。

```
$(TARGET):$(OBJS)                                 #TARGET 目标，需要先生成 OBJS 目标
    $(CC) -o $(TARGET) $(OBJS) $(CFLAGS)          #生成可执行文件
```

2.3.5　搜索路径

在大的系统中，通常存在很多目录，手动添加目录的方法不仅十分笨拙而且容易出错。make 的目录搜索功能提供了一个解决此问题的方法，指定需要搜索的目录，make 会自动找到指定文件的目录，使用 VPATH 变量可以指定其他搜索路径。VPATH 变量的使用方法如下：

```
VPATH=path1:path2:…
```

VPATH 右边是冒号（:）分隔的路径名称，例如下面的指令：

```
VAPTH=add:sub（加入 add 和 sub 搜索路径）
add_int.o:%o:%c
    $(CC) -c -o $@ $<
```

make 的搜索路径包含 add 和 sub 目录。add_int.o 规则自动扩展成如下代码：

```
add_int.o:add/add_int.c
    cc -c -o add_int.o add/add_int.c
```

将路径名去掉以后可以重新编译程序，但是会发现目标文件都放到了当前目录下，这样不利于文件的规范化。可以将输出的目标文件放到同一个目录下来解决此问题，重新修改上面的 Makefile 代码如下：

```
CFLAGS = -Iadd -Isub -O2
OBJSDIR = objs
VPATH=add:sub:.
OBJS = add_int.o add_float.o sub_int.o sub_float.o main.o
TARGET = cacu
$(TARGET):$(OBJSDIR) $(OBJS)     （要执行 TARGET 的命令，先查看 OBJSDIR 和 OBJS 依
                                  赖项是否存在）
    $(CC) -o $(TARGET) $(OBJSDIR)/*.o $(CFLAGS)（将 OBJSDIR 目录中所有的.o 文
                                                 件链接成 cacu）
$(OBJS):%.o:%.c                  （将扩展名为.o 的文件替换成扩展名为.c 的文件）
    $(CC) -c $(CFLAGS) $< -o $(OBJSDIR)/$@
$(OBJSDIR):                      （生成目标文件，存放在 OBJSDIR 目录下）
    mkdir -p ./$@                （建立目录，-p 选项可以忽略父目录不存在的错误）
clean:
    -$(RM) $(TARGET)             （删除 cacu）
    -$(RM) $(OBJSDIR)/*.o        （删除 OBJSDIR 下的所有.o 文件）
```

这样，目标文件都放到了 objs 目录下，只有最终的执行文件 cacu 放在当前目录下，执行 make 命令的结果如下：

```
$make cacu
mkdir -p ./objs
cc -c -Iadd -Isub -O2 add/add_int.c -o objs/add_int.o
cc -c -Iadd -Isub -O2 add/add_float.c -o objs/add_float.o
cc -c -Iadd -Isub -O2 sub/sub_int.c -o objs/sub_int.o
cc -c -Iadd -Isub -O2 sub/sub_float.c -o objs/sub_float.o
cc -c -Iadd -Isub -O2 main.c -o objs/main.o
cc -o cacu objs/*.o -Iadd -Isub -O2
```

编译目标文件时会自动加上路径名，例如，add_int.o 所依赖的文件 add_int.c 自动变成了 add/add_int.c，并且当 objs 不存在时会创建此目录。

2.3.6　自动推导规则

使用 make 命令编译扩展名为.c 的 C 语言文件时，源文件的编译规则无须明确给出。这是因为使用 make 命令进行编译时会使用一个默认的编译规则，按照默认规则完成对.c 文件的编译并生成对应的.o 文件。Makefile 执行命令 cc -c 来编译.c 源文件。在 Makefile 中只要给出需要重建的目标文件名（一个.o 文件），make 会自动为这个.o 文件寻找合适的依赖文件（对应的.c 文件），并且使用默认的命令来构建这个目标文件。

例如前面的例子，默认规则是使用命令 cc -c main.c-o main.o 来创建文件 main.o。对一个目标文件是“文件名.o”，依赖文件是“文件名.c”的规则，可以省略其编译规则的命令行，由 make 命令决定如何使用编译命令和选项。此默认规则称为 make 的隐含规则。

这样，在书写 Makefile 时就可以省略描述.c 文件和.o 依赖关系的规则，只需要给出那些特定的规则描述（.o 目标所需要的.h 文件）。因此前面的例子可以使用更加简单的方式来书写，Makefile 文件的内容如下：

```
CFLAGS = -Iadd -Isub -O2
VPATH=add:sub
OBJS = add_int.o add_float.o sub_int.o sub_float.o main.o
TARGET = cacu
$(TARGET):$(OBJS)                                    (OBJS 依赖项的规则自动生成)
    $(CC) -o $(TARGET) $(OBJS) $(CFLAGS)             (链接文件)
clean:
    -$(RM) $(TARGET)
    -$(RM) $(OBJS)
```

在此 Makefile 中，不用指定 OBJS 的规则，make 自动会按照隐含规则形成一个规则来生成目标文件。

2.3.7　递归调用

如果有多位开发者在多个目录下进行程序开发，并且每个人负责一个模块，而文件在相对独立的目录中，这时由同一个 Makefile 维护代码的编译就会十分不便，因为开发者对自己目录下的文件进行增减都要修改此 Makefile，这通常会给项目维护带来不便。

1．递归调用的方式

make 命令有递归调用的作用，它可以递归调用每个子目录中的 Makefile。例如，在当前目录下有一个 Makefile，而目录 add 和 sub 及主控文件 main.c 由不同的开发者进行维护，可以用如下方式编译 add 中的文件：

```
add:
    cd add && $(MAKE)
```

它等价于：

```
add:
    $(MAKE) -C add
```

上面两个例子都是先进入子目录 add 下，然后执行 make 命令。

2．总控Makefile

调用$(MAKE) -C 的 Makefile 叫作总控 Makefile。如果总控 Makefile 中的一些变量需要传递给下层的 Makefile，可以使用 export 命令。例如，需要向下层的 Makefile 传递目标文件的导出路径：

```
export OBJSDIR=./objs
```

例如，前面的文件布局，需要在 add 和 sub 目录下分别编译，总控 Makefile 的代码如下：

```
CC = gcc
CFLAGS = -O2
TARGET = cacu
export OBJSDIR = ${shell pwd}/objs          （生成当前目录的路径字符串并赋值给
                                            OBJSDIR，外部可调用）
RM = rm -f
$(TARGET):$(OBJSDIR)   main.o
    $(MAKE) -C add                          （在目录 add 下递归调用 make）
    $(MAKE) -C sub                          （在目录 sub 下递归调用 make）
    $(CC) -o $(TARGET) $(OBJSDIR)/*.o       （生成 main.o 放到 OBJSDIR 中）
main.o:%.o:%.c                              （main.o 规则）
    $(CC) -c $< -o $(OBJSDIR)/$@ $(CFLAGS) -Iadd -Isub
$(OBJSDIR):
    mkdir -p $(OBJSDIR)
clean:
    -$(RM) $(TARGET)
    -$(RM) $(OBJSDIR)/*.o
```

CC 编译器变量由总控 Makefile 统一指定，下层的 Makefile 直接调用 CC 编译器变量即可。生成的目标文件都放到./objs 目录下，输出一个变量 OBJSDIR。其中，${shell pwd}是执行一个 shell 命令 pwd 获得总控 Makefile 的当前目录。

生成 cacu 的规则是先建立目标文件的存放目录，再编译当前目录下的 main.c 为 main.o 目标文件。在命令中，递归调用 add 和 sub 目录下的 Makefile 生成目标文件并存放到目标文件所在的路径下，最后的命令是将目标文件全部编译生成执行文件 cacu。

3．子目录Makefile的编写

add 目录下的 Makefile 文件如下：

```
OBJS = add_int.o add_float.o
all:$(OBJS)
$(OBJS):%.o:%.c
    $(CC) -c $< -o $(OBJSDIR)/$@ $(CFLAGS)
                                        （CC 和 OBJSDIR 在总控 Makefile 声明）
clean:
    $(RM) $(OBJS)
```

这个 Makefile 文件很简单，编译 add 目录下的两个 C 文件，并将生成的目标文件放置在总控 Makefile 传入的目标文件的存放路径下。

sub 目录下的 Makefile 与 add 目录下一致，也是将生成的目标文件放置在总控 Makefile 指定的路径中。

```
OBJS = sub_int.o sub_float.o
all:$(OBJS)
$(OBJS):%.o:%.c
    $(CC) -c $< -o $(OBJSDIR)/$@ $(CFLAGS)（CC 和 OBJSDIR 在总控 Makefile 中的声明）
```

```
clean:
    $(RM) $(OBJS)
```

2.3.8　Makefile 中的函数

在比较大的工程中，经常需要一些匹配功能或者自动生成规则的功能，这些功能可以通过函数来实现，本节将对 Makefile 中经常使用的函数进行介绍。

1．获取匹配模式的文件名函数wildcard

wildcard 函数的功能是查找当前目录下所有符合模式 PATTERN 的文件名，其返回值是以空格分隔的、当前目录下的所有符合模式 PATTERN 的文件名列表。函数原型如下：

```
$(wildcard PATTERN)
```

例如，返回当前目录下所有扩展名为.c 的文件列表，代码如下：

```
$(wildcard *.c)
```

2．模式替换函数patsubst

patsubst 函数的功能是查找字符串 text 中按照空格分开的单词，将符合 pattern 模式的字符串替换成 replacement。在 pattern 模式中可以使用通配符，%代表 0 个到 *n* 个字符，当 pattern 和 replacement 中都有%时，符合条件的字符将被 replacement 参数替换。patsubst 函数的返回值是替换后的新字符串，其原型如下：

```
$(patsubst pattern,replacement,text)
```

例如，需要将 C 文件替换为.o 的目标文件，可以使用如下模式：

```
$(patsubst %.c,%.o, add.c)
```

上面的模式将 add.c 字符串作为输入，当扩展名为.c 时符合模式%.c，其中，“%”在这里代表 add，替换为 add.o，并作为输出字符串。

```
$(patsubst %.c,%.o, $(wildcard *.c))
```

输出的字符串将当前扩展名为.c 的文件替换成扩展名为.o 的文件列表。

3．循环函数foreach

foreach 函数的原型如下：

```
$(foreach VAR,LIST,TEXT)
```

foreach 函数的功能是将 LIST 字符串中以一个空格分隔的单词，先传给临时变量 VAR，然后执行 TEXT 表达式，TEXT 表达式处理结束后输出结果。foreach 函数的返回值是空格分隔表达式 TEXT 的计算结果。

例如，对于存在 add 和 sub 的两个目录，设置 DIRS 为“add sub ./”包含目录 add、sub 和当前目录。表达式$(wildcard$(dir)/*.c)可以取出目录 add、sub 及当前目录下的所有扩展名为.c 的 C 语言源文件。

```
DIRS = sub add ./            (DIRS 字符串的值为目录 add、sub 和当前目录)
(查找所有目录下扩展名为.c 的文件，赋值给变量 FILES)
FILES = $(foreach dir, $(DIRS),$(wildcard $(dir)/*.c))
```

利用上面几个函数对原有的 Makefile 文件进行重新编写，使新的 Makefile 可以自动更新各

个目录下的 C 语言源文件：

```
CC = gcc
CFLAGS = -O2 -Iadd -Isub
TARGET = cacu
DIRS = sub add .                          (DIRS 字符串的值为目录 add、sub 和当前目录)
(查找所有目录下扩展名为.c 的文件并赋值给变量 FILES)
FILES = $(foreach dir, $(DIRS),$(wildcard $(dir)/*.c))
OBJS = $(patsubst %.c,%.o,$(FILES)) (替换字符串，将扩展名为.c 替换成扩展名为.o)
$(TARGET):$(OBJS)                         (OBJS 依赖项规则是默认生成的)
    $(CC) -o $(TARGET) $(OBJS)            (生成 cacu)
clean:
    -$(RM) $(TARGET)
    -$(RM) $(OBJS)
```

编译程序，输出结果如下：

```
# make
gcc -O2 -Iadd -Isub   -c -o sub/sub_float.o sub/sub_float.c
gcc -O2 -Iadd -Isub   -c -o sub/sub_int.o sub/sub_int.c
gcc -O2 -Iadd -Isub   -c -o add/add_float.o add/add_float.c
gcc -O2 -Iadd -Isub   -c -o add/add_int.o add/add_int.c
gcc -O2 -Iadd -Isub   -c -o main.o main.c
gcc -o cacu sub/sub_float.o sub/sub_int.o add/add_float.o add/add_
int.o ./main.o
```

注意：Windows 下的 nmake 是和 make 类似的工程管理工具，只是由于 Visual Studio 系列的强大 IDE 开发环境通常被忽略。所有平台的 Makefile 都是相似的，大概有 80%的相似度，不同之处主要是函数部分和外部命令，而最核心的规则部分则是一致的，都是基于目标、依赖项和命令的方式进行解析。

2.4　GDB 调试工具

前几节主要对 Linux 的编程环境进行了介绍。要使程序能够正常运行，跟踪代码、调试漏洞是不可缺少的。Linux 有一个很强大的调试工具 GDB（GNU Debuger），可以用它来调试 C 和 C++程序。GDB 提供了以下功能：

- 在程序中设置断点，当程序运行到断点处暂停。
- 显示变量的值，可以打印或者监视某个变量，将变量的值显示出来。
- 单步执行，GDB 允许用户单步执行程序，可以跟踪进入函数和从函数中退出。
- 运行时修改变量的值，GDB 允许在调试状态下修改变量的值，此功能在测试程序的时候是十分有用的。
- 路径跟踪，GDB 可以将代码的路径打印出来，方便用户跟踪代码。
- 线程切换，在调试多线程时，线程切换功能是必不可少的。

除了以上功能之外，GDB 还可以显示程序的汇编代码、打印内存的值等。

2.4.1　编译可调试程序

GDB 是一套字符界面的程序集，可以使用 gdb 命令加载要调试的程序。例如，输入 gdb 后显示 GDB 的版权声明（以 Ubuntu 为例）：

```
$ gdb
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

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

在此状态下输入 q，退出 GDB。

要使用 GDB 进行调试，在编译程序的时候需要加入-g 选项。例如，编译如下代码：

```
/*文件名：gdb-01.c*/
#include <stdio.h>                          /*用于 printf*/
#include <stdlib.h>                         /*用于 malloc*/
/*声明函数 sum()为 static int 类型*/
static int sum(int value);
/*用于控制输入、输出的结构*/
struct inout{
    int value;
    int result;
};
int main(int argc, char *argv[]){
    /*申请内存*/
    struct inout *io = (struct inout*)malloc(sizeof(struct inout));

    if(NULL == io)                          /*判断是否成功*/
    {
        printf("申请内存失败\n");           /*打印失败信息*/
        return -1;                          /*返回-1*/
    }
    if(argc !=2)                            /*判断输入参数是否正确*/
    {
        printf("参数输入错误!\n");          /*打印失败信息*/
        return -1;                          /*返回-1*/
    }
    io->value = *argv[1]-'0';               /*获得输入的参数*/
    io->result = sum(io->value);            /*对 value 进行累加求和*/
    printf("你输入的值为: %d,计算结果为: %d\n",io->value,io->result);
    return 0;
}
/*累加求和函数*/
static int sum(int value){
    int result = 0;
    int i = 0;
    for(i=1;i<value;i++)                    /*循环计算累加值*/
        result += i;

```

```
37      return result;                                  /*返回结果*/
38  }
```

上面的代码是进行累加求和，例如，向 sum 中输入 3，其结果应该是 1+2+3=6。编译代码如下：

```
$gcc -o test gdb-01.c -g
```

生成了 test 的可执行文件。运行此程序：

```
$./test 3
你输入的值为：3，计算结果为：3
```

GDB 之所以能够调试程序，是因为进行编译时的-g 选项，如果设置了这个选项，那么 GCC 会向程序中加入“楔子”，GDB 能够利用这些楔子与程序交互。

2.4.2　使用 GDB 调试程序

在 2.4.1 小节中，我们将源文件 gdb-01.c 编译成目标文件 test。在编译时加入了-g 选项，可以使用 GDB 对可执行文件 test 进行调试。下面利用 GDB 调试 test，查找 test 计算错误的原因。

1．加载程序

使用 GDB 加载程序时，需要先将程序加载到 GDB 中。加载程序的命令格式为“gdb 要调试的文件名”。例如，下面的命令将可执行文件 test 加载到 GDB 中。

```
$gdb test                                          （gdb 命令+调试的可执行文件）
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

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

2．设置输入参数

通常，可执行文件在运行的时候需要输入参数，在 GDB 中向可执行文件输入参数的命令格式为“set args 参数值 1 参数值 2 …”。例如，下面的命令 set args 3 表示向可执行文件输入的参数设为 3，即传给 test 程序的值为 3。

```
(gdb) set args 3                                   （设置参数 args 为 3）
(gdb)
```

3．打印代码内容

命令 list 用于列出可执行文件对应源文件的代码，命令格式为“list 开始的行号”。例如，下面的命令 list 1，从第一行开始列出代码，每次按 Enter 键后顺序列出后面的代码。

```
(gdb) list 1
01  /*文件名：gdb-01.c*/
02  #include <stdio.h>                              /*用于 printf*/
03  #include <stdlib.h>                             /*用于 malloc*/
04  /*声明函数 sum()为 static int 类型*/
05  static int sum(int value);
06  /*用于控制输入、输出的结构*/
07  struct inout{
08      int value;
09      int result;
10  };
(gdb)（按 Enter 键）
11  int main(int argc, char *argv[]){
12      /*申请内存*/
13      struct inout *io = (struct inout*)malloc(sizeof(struct inout));
14      if(NULL == io)                              /*判断是否成功*/
15      {
16          printf("申请内存失败\n");               /*打印失败信息*/
17          return -1;                              /*返回-1*/
18      }
(gdb)（按 Enter 键）
19      if(argc !=2)                                /*判断输入参数是否正确*/
20      {
21          printf("参数输入错误!\n");              /*打印失败信息*/
22          return -1;                              /*返回-1*/
23      }
24      io->value = *argv[1]-'0';                   /*获得输入的参数*/
25      io->result = sum(io->value);                /*对 value 进行累加求和*/
26      printf("你输入的值为: %d,计算结果为: %d\n",io->value,io->result);
27      return 0;
(gdb)（按 Enter 键）
28  }
29  /*累加求和函数*/
30  static int sum(int value){
31      int result = 0;
32      int i = 0;
33      for(i=0;i<value;i++)                        /*循环计算累加值*/
34          result += i;
35
36      return result;                              /*返回结果*/
(gdb)（按 Enter 键）
37  }
(gdb)（按 Enter 键）
38  Line number 38 out of range; gdb-1.c has 37 lines.
(gdb)
```

4．设置断点

b 命令可以在某一行设置断点，程序运行到断点的位置会中断，等待用户的下一步操作指令。

```
(gdb) b 34
Breakpoint 1 at 0x1254: file gdb-01.c, line 34.
(gdb)
```

5．运行程序

GDB 在默认情况下是不会让可执行文件运行的，此时，程序并没有真正运行起来，只是装载进了 GDB 中。要使程序运行，需要输入 run 命令。

```
(gdb) run 3
Starting program: /home/linux-c/Linux_net/02/2.4.1/test 3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, sum (value=3) at gdb-1.c:34
34          result += i;    （此处遇到断点）
(gdb)
```

6．显示变量

当程序运行到设置的第 34 行断点的时候，程序会中断运行，等待下一步的指令。这时可以进行一系列的操作，其中，命令 display 可以显示变量的值。

```
(gdb)display i（每次停止时显示变量 i 的值）
1: i = 0
(gdb)display result（每次停止时显示变量 result 的值）
2: result = 0
(gdb)c（继续运行）
Continuing.

Breakpoint 1, sum (value=3) at gdb-1.c:34
34          result += i;
1: i = 1
2: result = 0
(gdb)c（继续运行）
Continuing.

Breakpoint 1, sum (value=3) at gdb-1.c:34
34          result += i;
1: i = 2
2: result = 1
(gdb)
Continuing.
你输入的值为：3，计算结果为：3
[Inferior 1 (process 11647) exited normally]
(gdb)
```

通过上面的跟踪，已经可以判断出问题出在 for(i=0;i<value;i++);此行代码上，可以将其修改为 for(i=1;i<=value;i++)，或者修改 result+=i 为 result+=(i+1)。

7．修改变量的值

要在 GDB 中修改变量的值，可以使用 set 命令。例如，修改 result 的值为 6：

```
(gdb) set result=6                          （修改 result 的值为 6）
(gdb)c                                      （继续运行）
Continuing.
你输入的值为：3，计算结果为：12
[Inferior 1 (process 9563) exited normally]
```

8．退出GDB

在调试完程序后，可以使用 q 命令退出 GDB。

```
(gdb) q                                                    （退出）
$
```

2.4.3　GDB 的常用命令

在 2.4.2 小节中举了一个简单的例子演示了 GDB 的使用，本小节将详细介绍 GDB 的常用命令。GDB 的常用命令参见表 2.7，主要包含信息获取、断点设置、运行控制和程序加载等常用命令，这些命令可以进行调试时的程序控制和程序的参数设置等。

表 2.7　GDB的常用命令

GDB的命令	格　　式	含　　义	简　写
list	list [开始,结束]	列出文件的代码清单	l
print	printf p	打印变量内容	p
break	break [行号|函数名称]	设置断点	b
continue	continue [开始,结束]	继续运行	c
info	info para	列出信息	i
next	next	下一行	n
step	step	进入函数	S
display	display para	显示参数	
file	file path	加载文件	
run	run args	运行程序	r

1．执行程序

用 GDB 执行程序可以使用 gdb program 的方式，program 是程序的程序名。当然，此程序编译的时候要使用-g 选项。如果在启动 GDB 的时候没有选择程序名称，可以在 GDB 启动后使用 file program 方法启动。例如：

```
(gdb)file test
```

如果要运行准备好的程序，可以使用 run 命令，在它后面是传递给程序的参数，例如：

```
(gdb)run 3
Starting program: /home/linux-c/Linux_net/02/2.4.1/test 3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
你输入的值为：3，计算结果为：6
[Inferior 1 (process 11685) exited normally]
```

如果使用不带参数的 run 命令，GDB 就会再次使用前一条 run 命令的参数。

2．参数设置和显示

使用 set args 命令可以设置发送给程序的参数；使用 show args 命令可以查看其默认的参数。

```
(gdb)set args 3
(gdb)show args
```

```
Argument list to give program being debugged when it is started is "3".
(gdb)                                          （按 Enter 键）
Argument list to give program being debugged when it is started is "3".
```

如果按 Enter 键，GDB 默认执行上一个命令。例如，上面的例子中，执行命令 show args 后，按 Enter 键会接着执行 show args 命令。

3．列出文件清单

打印文件代码的命令是 list，简写为 l。list 的命令格式如下：

```
list line1,line2
```

上面的命令是打印 line1 到 line2 之间的代码。如果不输入参数，则从当前行开始输出。例如，打印第 2 行到第 5 行之间的代码：

```
(gdb) l 2,5
2   #include <stdio.h>                  /*用于 printf*/
3   #include <stdlib.h>                 /*用于 malloc*/
4   /*声明函数 sum()为 static int 类型*/
5   static int sum(int value);
(gdb)
```

4．打印数据

打印变量或者表达式的值可以使用 print 命令，简写为 p。它是一个功能很强大的命令，可以打印任何有效表达式的值。除了可以打印程序中的变量值之外，还可以打印其他合法的表达式。print 命令的使用方式如下：

```
(gdb)print var                                    （var 为参数）
```

print 命令可以打印常量表达式的值。例如，打印 2+3 的结果：

```
(gdb) p 2+3
$7= 5
```

print 命令可以计算函数调用的返回值。例如，调用函数 sum()对 3 求和：

```
(gdb) p sum(3)
$8 = 6
```

print 命令可以打印一个结构中各个成员的值。例如，打印上面代码中 io 结构中的各个成员的值：

```
(gdb) p *io
$9 = {value = 3, result = 6}
```

在 GDB 系统中，之前打印的历史值保存在全局变量中。例如，在$9 中保存了结构 io 的值，用 print 可以打印历史值。

```
(gdb) p $9
$3 = {value = 3, result = 6}
```

利用 print 命令可以打印构造数组的值，给出数组的指针头并且设定要打印的结构数量，print 命令会依次打印各个值。打印构造数组的格式如下：

```
基地址@个数
```

例如，*io 是结构 ioout 的头，要打印从 io 开始的两个数据结构（当然最后一个是非法的，这里只是举例）。

```
(gdb) p *io@2
```

```
$13 = {{value = 3, result = 6}, {value = 0, result = 135153}}
```

5．断点

设置断点的命令是 break，简写为 b。设置断点有如下 3 种形式，注意 GDB 的停止位置都是在执行程序之前。

- break 行号：程序停止在设定的行之前。
- break 函数名称：程序停止在设定的函数之前。
- break 行号或者函数+if 条件：这是一个条件断点设置命令，如果条件为真，则程序在到达指定行或函数时停止。

（1）设置断点。如果程序由很多的文件构成，在设置断点时则要指定文件名。例如：

```
(gdb) b gdb-01.c:34          （在文件 gdb-01.c 的第 34 行设置断点）
Breakpoint 5 at 0x804849c: file gdb-01.c, line 34.
(gdb) b gdb-01.c:sum         （在文件 gdb-01.c 的函数 sum 处设置断点）
Breakpoint 6 at 0x8048485: file gdb-01.c, line 25.
```

要设置一个条件断点，可以利用 break if 命令，在调试循环代码段时该命令比较方便，省略了大量的手动调试工作。例如，在 sum()函数中，当 i 为 2 时设置断点如下：

```
(gdb) b 38 if i==2
Breakpoint 8 at 0x804849c: file gdb-01.c, line 38.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/linux-c/Linux_net/02/2.4.1/test 3

Breakpoint 3, sum (value=3) at gdb-01.c:38
38          result += i;
```

（2）显示所有 GDB 的断点信息。使用 info breakpoints 命令可以显示所有断点的信息，例如，显示所有断点的信息：

```
(gdb) info breakpoints
Num     Type          Disp Enb Address    What
2       breakpoint     keep y   0x080484bc in main at gdb-01.c:29
3       breakpoint     keep y   0x08048502 in sum at gdb-01.c:38
    stop only if i=2
    breakpoint already hit 1 time
```

信息分为 6 类：Num 是断点编号，Type 是信息的类型，Disp 是描述，Enb 是断点是否有效，Address 是断点在内存中的地址，What 是对断点在源文件中的位置描述。

第 3 个断点的停止条件为“当 i==2 时”，已经命中了 1 次。

（3）删除指定的某个断点。删除某个指定的断点使用命令 delete，命令格式为“delete 断点编号”。例如，下面的命令 delete b 3 会删除第 3 个断点。

```
(gdb) delete 3
(gdb) info b
Num     Type          Disp Enb Address    What
2       breakpoint     keep y   0x080484bc in main at gdb-01.c:29
```

上面的命令会删除断点编号为 3 的断点。如果不带编号参数，则会删除所有的断点。

（4）禁止断点。禁止某个断点使用命令 disable，命令格式为“disable 断点编号”。将某个断点禁止后，GDB 进行调试时，在断点处程序不再中断。例如，下面的命令 disable 2 将禁止使用断点 2，即程序运行到断点 2 时不会停止，同时断点信息的使能域将变为 n。

```
(gdb) disable 2
```

（5）允许断点。允许某个断点，使用命令 enable，命令格式为“enable 断点编号”。该命令将禁止的断点重新启用，GDB 会在启用的断点处重新中断。例如，下面的命令 enable 2，将允许使用断点 2，即程序运行到断点 2 时会停止，同时断点信息的使能域将变为 y。

```
(gdb) enable 2
```

（6）清除断点。一次性地清除某行处的所有断点信息使用命令 clear，命令格式为“clear 源代码行号”。将某行的断点清除后，GDB 不再保存这些信息，此时不能使用 enable 命令重新允许断点生效。如果想重新在某行处设置断点，则必须重新使用命令 breakpoint 进行设置。例如，下面的命令 clear 29，将清除在源代码文件中第 29 行所设置的断点。

```
(gdb)clear 29
```

6．变量类型检测

在调试过程中有需要查看变量类型的情况，可以使用的命令有 whatis 和 ptype 等。

- whatis：打印数组或者变量的类型。要查看程序中某个变量的类型，可以使用命令 whatis。whatis 命令的格式为“whatis 变量名”，其中的变量名是要查看的变量。例如，查看 io 和 argc 的变量类型，io 为 struct inout 类型，argc 为 int 类型。

```
(gdb) whatis *io
type = struct inout
(gdb) whatis argc
type = int
```

- ptype：查看变量的详细信息。当使用 whatis 命令查看变量的类型时，只能获得变量的类型名称，不能得到变量的详细信息。如果想要查看变量的详细信息，需要使用命令 ptype。例如，查看 io 的类型，最后的*表明 io 是一个指向 struct inout 类型的指针。

```
(gdb) ptype io
type = struct inout {
    int value;
    int result;
} *
```

7．单步调试

在调试程序的时候经常需要单步跟踪，并在适当的时候进入函数体内部继续跟踪。GDB 的 next 命令和 step 命令提供了这种功能。next 是单步跟踪的命令，简写为 n；step 是可以进入函数体的命令，简写为 s。如果已经进入某个函数，又想退出函数的运行，返回到调用的函数中，那么可以使用 finish 命令。例如，在代码的第 28 行设置断点，跟踪程序，在第 34 行进入 sum() 函数的内部继续跟踪。

```
(gdb) b 28                                          （在 28 行处设置断点）
Breakpoint 5 at 0x80484a7: file gdb-01.c, line 28.
(gdb) run 3                                         （运行程序，输入参数为 3）
（在 28 行断点处停下）
Starting program: /home/linux-c/Linux_net/02/2.4.1/test 3
Breakpoint 5, main (argc=2, argv=0xbffff244) at gdb-01.c:28
28      io->result = sum(io->value);        /*对 value 进行累加求和*/
(gdb) s                                             （进入函数体 sum()内部）
sum (value=3) at gdb-01.c:34
34      int result = 0;
(gdb) n                                             （单步执行）
```

```
35      int i = 0;
(gdb) n                                          (单步执行)
37      for(i=0;i<=value;i++)                   /*循环计算累加值*/
(gdb) finish                                     (执行完函数 sum())
Run till exit from #0  sum (value=3) at gdb-01.c:37
0x080484b5 in main (argc=2, argv=0xbffff244) at gdb-01.c:28
28      io->result = sum(io->value);            /*对 value 进行累加求和*/
Value returned is $9 = 6                         (函数 sum()执行完毕，结果为 6)
(gdb) n                                          (单步执行)
29      printf("你输入的值为：%d,计算结果为：%d\n",io->value,io->result);
(gdb) c                                  (继续执行直到程序结束或者遇到下一个断点)
Continuing.
你输入的值为：3，计算结果为：6
[Inferior 1 (process 9772) exited normally]
(gdb)
```

8．设置监测点

display 命令可以显示某个变量的值，在程序结束或者遇到断点时，可以将设置变量的值显示出来。当然是否显示，还要看变量的作用域，display 命令只显示作用域内变量的值。例如，将 io 和 sum 中的 result 设置为显示，设置的断点在第 27 行、第 29 行和第 38 行，进行调试的情况如下：

```
$ gdb test                                       (运行 GDB，加载程序 test)
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"…
Reading symbols from test…
(gdb) b 27                                       (在第 27 行设置断点)
Breakpoint 1 at 0x8048490: file gdb-01.c, line 27.
(gdb) b 29                                       (在第 29 行设置断点)
Breakpoint 2 at 0x80484bc: file gdb-01.c, line 29.
(gdb) b 38                                       (在第 38 行设置断点)
Breakpoint 3 at 0x8048502: file gdb-01.c, line 38.
(gdb) run 3                  (运行程序 test，输入参数 3，将在第 27 行断点处停下)
Starting program: /home/linux-c/Linux_net/02/2.4.1/test 3

Breakpoint 1, main (argc=2, argv=0xbffff244) at gdb-01.c:27
27      io->value = *argv[1]-'0';               /*获得输入的参数*/
(gdb) display *io                                (设置显示参数*io)
1: *io = {value = 0, result = 0}                 (显示当前值)
(gdb) c                                          (继续运行，将在第 46 行断点处停下)
Continuing.

Breakpoint 3, sum (value=3) at gdb-01.c:38
```

```
38          result += i;                          (在第 38 行断点处停下)
(gdb) display result                              (设置显示参数 result)
2: result = 0
(gdb) n                                  (单步执行，到下一行停下并显示 result 的值)
37      for(i=0;i<=value;i++)                    /*循环计算累加值*/
2: result = 0                                     (当前值为 0)
(gdb) c                                           (继续执行，到第 38 行断点处停下)
Continuing.

Breakpoint 3, sum (value=3) at gdb-01.c:38
38          result += i;
2: result = 0                                     (result 参数的当前值为 0)
(gdb) c                                           (继续执行，到第 38 行断点处停下)
Continuing.

Breakpoint 3, sum (value=3) at gdb-01.c:38
38          result += i;
2: result = 1                                     (result 参数的当前值为 1)
(gdb) c                                           (继续执行，到第 38 行断点处停下)
Continuing.

Breakpoint 3, sum (value=3) at gdb-01.c:38
38          result += i;
2: result = 3                                     (result 参数的当前值为 3)
(gdb) c                                           (继续执行，到第 29 行断点处停下)
Continuing.

Breakpoint 2, main (argc=2, argv=0xbffff244) at gdb-01.c:29
29      printf("你输入的值为: %d,计算结果为: %d\n",io->value,io->result);
1: *io = {value = 3, result = 6}    (在自己的作用域内显示*io 的当前值)
(gdb) c                                  (继续执行)
Continuing.
你输入的值为：3，计算结果为：6
[Inferior 1 (process 9810) exited normally]
(gdb)
```

9．调用路径

backtrace 命令可以打印函数的调用路径并提供向前跟踪功能，该命令对跟踪函数很有帮助。backtrace 命令可以打印一个顺序列表，显示函数从最近到最远的调用过程，包含调用函数及其参数。backtrace 命令简写为 bt。例如，在第 38 行设置断点，然后打印调用过程：

```
(gdb) bt
#0  sum (value=3) at gdb-01.c:38
#1  0x080484b5 in main (argc=2, argv=0xbffff244) at gdb-01.c:28
```

10．获取当前命令的信息

info 命令可以获得当前命令的信息，如获得断点信息及参数的设置等。

11．多线程thread

多线程是现代程序中经常采用的编程方法，而多线程由于其在执行过程中的调度随机性，所以不好调试。多线程调试主要有两步：先获得线程的 ID 号，然后转到该线程进行调试。

info thread 命令用于列出当前进程中的线程号，其中，最前面的为调试用的 ID。使用 thread

id 命令可以进入需要调试的线程。

12．汇编disassemble

使用 disassemble 命令可以打印指定函数的汇编代码，例如 sum()函数的汇编代码如下：

```
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x080484e5 <+0>:     push   %ebp
   0x080484e6 <+1>:     mov    %esp,%ebp
   0x080484e8 <+3>:     sub    $0x10,%esp
   0x080484eb <+6>:     movl   $0x0,-0x8(%ebp)
   0x080484f2 <+13>:    movl   $0x0,-0x4(%ebp)
   0x080484f9 <+20>:    movl   $0x0,-0x4(%ebp)
   0x08048500 <+27>:    jmp    0x804850c <sum+39>
=> 0x08048502 <+29>:    mov    -0x4(%ebp),%eax
   0x08048505 <+32>:    add    %eax,-0x8(%ebp)
   0x08048508 <+35>:    addl   $0x1,-0x4(%ebp)
   0x0804850c <+39>:    mov    -0x4(%ebp),%eax
   0x0804850f <+42>:    cmp    0x8(%ebp),%eax
   0x08048512 <+45>:    jle    0x8048502 <sum+29>
   0x08048514 <+47>:    mov    -0x8(%ebp),%eax
   0x08048517 <+50>:    leave
   0x08048518 <+51>:    ret
End of assembler dump.
```

13．GDB的帮助信息

在使用本例时，读者可能会遇到一些困扰，如命令 c 是什么意思、display 是什么等。这些问题在 GDB 中可以利用 help 命令来解决。例如：

```
(gdb) help
List of classes of commands:

aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

help 命令可以罗列出 GDB 支持的命令，如断点（Breakpoints）、环境数据（Data）等。对命令 c 等的疑问可以使用 help 命令获得帮助：

```
 (gdb) help c
continue, fg, c
```

```
Continue program being debugged, after signal or breakpoint.
Usage: continue [N]
If proceeding from breakpoint, a number N may be used as an argument,
which means to set the ignore count of that breakpoint to N - 1 (so that
the breakpoint won't break until the Nth time it is reached).

If non-stop mode is enabled, continue only the current thread,
otherwise all the threads in the program are continued.  To
continue all stopped threads in non-stop mode, use the -a option.
Specifying -a and an ignore count simultaneously is an error.
```

help c 命令用于继续执行程序，并且可以设置执行的行数，其中，c 是 Continue 的简写。

2.4.4　其他 GDB 程序

除了基于命令行的 GDB 调试程序，在 Linux 中还有很多基于 GDB 的程序，如 xxgdb 和 insight 等。

1．xxgdb程序

xxgdb 程序对命令行的 GDB 进行简单的包装，将 GDB 的输入、命令和输出分为几个窗口，并且将命令用多个按钮来表示，方便使用。

2．Emacs程序

Emacs 程序集成了 GDB 的调试功能，可以用下面的命令启动 GDB。

```
M-x gdb
```

在 Emacs 程序中有一种多窗口调试模式，可以把窗口划分为 5 个窗格，同时显示 GDB 命令窗口、当前局部变量、程序文本、调用栈和断点。

Emacs 对 GDB 的命令和快捷键进行了绑定。对于常用的命令，使用快捷键比较方便。例如，Ctrl+C 和 Ctrl+N 快捷键对应的是 next line 命令，Ctrl+C 和 Ctrl+S 快捷键对应的是 step in 命令，在调试过程中用得最多的快捷键就是这两个。

注意：Ctrl+C 和 Ctrl+N 共同构成 next line 命令对应的快捷键。其中，Ctrl+C 作为前缀，Ctrl+N 表示实际的功能。

2.5　小　　结

本章介绍了在 Linux 环境中进行编程的基本知识，包括 Vi 编辑器、GCC 编译器、Makefile 的编写、使用 GDB 进行程序调试。

Vi 编辑器是进行 Linux 开发的常用编辑器，它的功能非常强大，本章介绍了 Vim 的使用方法。

- GCC 编译器是进行编程必须了解的工具，本章仅介绍了使用 GCC 进行程序编译的简单方法。
- GDB 是在 Linux 中进行程序调试的首选，并且现在已经有很多图形客户端，使用起来更加方便。有一些开发环境集成了 GDB 的调试环境。

- Makefile 是进行程序编译经常使用的编译配置文件，本章对 Makefile 进行了详细介绍。

注意：在进行较大的工程构建时，经常会用到 Libtools 工具。在这个工具的帮助下进行一些修改，就可以构建自己的项目了。目前，大多数项目都是采用 Libtools 工具构建的。与 Libtools 相比，CMake 的优势更明显，不论从速度还是可读性上，都比 Libtools 好很多，KDE 就采用了 CMake 工具来构建项目。

2.6 习　　题

一、填空题

1．Vi 是_____的简写。
2．GCC 的 C 编译器是_____。
3．Makefile 的框架是由_____构成的。

二、选择题

1．在 Vim 中光标左移一个字符的位置使用的键是（　　）。
A．j　　B．h　　C．k　　D．i
2．在 GCC 编译器中编译程序时，下列可以告诉编译器进行预编译操作的选项是（　　）。
A．-E　　B．-H　　C．-A　　D．前面三项都不正确
3．在 GDB 中可以列出文件的代码清单的命令是（　　）。
A．list　　B．info　　C．file　　D．前面三项都不正确

三、判断题

1．安装好 Linux 操作系统后，默认安装了 Vim 编辑器。（　　）
2．GCC 不可以自动编译链接多个文件。（　　）
3．使用 GDB 加载程序的时候，需要先将程序加载到 GDB 中。（　　）

四、操作题

1．使用 Vim 创建一个 linux.c 文件，在其中输入以下代码：

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

2．使用 GCC 编译器对 linux.c 文件进行编辑并生成可执行文件。