Apress®

C/C++技术丛书

高级 C/C++ 编译技术

Advanced C and C++ Compiling

[美] 米兰·斯特瓦诺维奇（Milan Stevanovic） 著 卢誉声 译

机械工业出版社
China Machine Press

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明:

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。

学习如何编写C/C++代码仅仅是个开始。如果你希望从事系统底层开发，或想深入理解操作系统、编译器及它们之间内在的关联，成为“编程专家”，那么就必须对编译器生成的二进制文件（目标文件、静态库、动态库和可执行文件）的作用和结构进行充分的了解。开源已经在许多方面从根本上改变了软件的原有面貌，越来越多的系统开始采用或集成开源代码，因此对每位开发人员来说，学习和理解这些底层技术也变得十分重要。

本书深入浅出地讲解了构建过程（编译、链接）中的各种细节，从多个角度展示了程序与库文件或代码的集成方法，提出了面向代码复用和系统集成的软件架构设计方法，以及系统开发过程中疑难问题的解决方案。同时，本书也是一本C/C++二进制文件方面的软件工程指南，涵盖中级和专家程序员所需的各方面内容和信息。

通过阅读本书，你将学到：

- 构建过程（编译和链接）与装载过程的内部原理
- 静态库、动态库和可执行文件的内部工作机制
- 面向代码复用和系统集成的软件架构设计方法
- 编译、链接与运行时出现问题的排查技巧
- 在Linux和Windows平台下利用二进制文件分析工具进行分析

Apress®

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机\程序设计
ISBN 978-7-111-49618-2
9 787111 496182 >
定价：69.00元

高级 C/C++ 编译技术

Advanced C and C++ Compiling

[美] 米兰·斯特瓦诺维奇（Milan Stevanovic） 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

高级 C/C++ 编译技术 /（美）斯特瓦诺维奇（Stevanovic，M.）著；卢誉声译．—北京：机械工业出版社，2015.3
（C/C++ 技术丛书）
书名原文：Advanced C and C++ Compiling

ISBN 978-7-111-49618-2

I. 高… II. ①斯… ②卢… III. C 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2015）第 048520 号

本书版权登记号：图字：01-2014-8120

Milan Stevanovic: Advanced C and C++ Compiling (ISBN: 978-1-4302-6667-9).

Original English language edition published by Apress L. P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA. Copyright © 2014 by Apress L. P. Simplified Chinese-language edition copyright © 2015 by China Machine Press. All rights reserved.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由 Apress 出版社出版。

本书简体字中文版由 Apress 出版社授权机械工业出版社独家出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括中国香港、台湾、澳门地区）销售发行，未经授权的本书出口将被视为违反版权法的行为。

高级 C/C++ 编译技术

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2015 年 4 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：17.5
书　　号：ISBN 978-7-111-49618-2　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

The Translator's Words 译者序

虽说计算机软件编程领域的技术发展日新月异，但我始终坚信所有程序开发技术演进都万变不离其宗，其本质自始至终从未发生过改变。就编程语言来说，无论抽象到何种层次，使用何种风格，最终都会回归到本质——机器代码。而在我们常用的高级语言中，最贴近机器及操作系统底层的恐怕就是 C 语言了。

从一定程度上来讲，C 语言堪称一门革命性的语言，它完美平衡了语言中机器相关与机器无关的部分，使得我们可以用机器无关的方式来处理程序逻辑，但必要时又可以直接控制底层硬件，C 语言被广泛运用在操作系统开发中正是这一点的绝佳例证。同时，C 语言的核心是非常简单的，一切细节都暴露在程序员面前，不会因为某种语法构造而导致隐藏的性能消耗。这使得 C 语言成为程序员在追求程序效率时的一个绝佳选择。C 语言为了保证核心语言的简单（包括标准库的简单）与高效，没有将很多必要的现代语言特性融入标准中，比如 C 语言自身没有提供任何反射机制。这是一把双刃剑，也就是说，如果想在 C 语言中实现任何动态语言特性，必须使用和系统相关的特性，尤其是在动态库、共享库、插件等概念日益重要的今天，这给 C 程序员带来很多麻烦。虽说现在各种主流操作系统中都很好地支持动态库或共享库，也提供了相应接口，但是不同平台中的概念都会有细微差别，而且在具体技术细节上也会不同，这使得程序员在处理跨平台问题时会遇到很多问题。

C++ 的出现则让这一切变得更加错综复杂。

C++ 是一门多范式语言，在提供了面向对象和泛型编程的同时，为了保证效率，依然坚持不加入任何动态特性，或者提供某些非常有局限性的动态特性（比如 RTTI），为了实现多态则采用了虚函数表这种折中方法。除了 C++ 语法给编译器实现带来的复杂性之外，受到更大挑战的则是 C++ 的链接器和装载器实现。符号名称修饰、全局对象管理、模板支持等各种特性使得链接器和装载器为了支持 C++ 需要付出更多的努力。

同时，C++ 程序员处理动态库或共享库时也同样需要更加细心、耐心。很多时候，我们一不留意就会被各种链接器或者装载器错误提示搅得心如乱麻，而查遍代码也找不出问题所在。编译器优化的不确定因素更加剧了这种情况。另一方面，现在的计算机教育中，包括程序设计语言课程、编译原理课程、操作系统课程，但并没有一门关于链接和装载的课程，链接和装载往往只是在这些课程中一带而过。或许这是由于链接和装载是非常纯粹的技术性话题，并不像其他课程那样需要传授各种理论，但这导致很多人对链接和装载一知半解，根本没有系统认识。

这些因素叠加起来的后果是，许多初级 C/C++ 程序员遇到链接和装载问题时感到束手无策，根本不知问题出在何处。如果说 Windows 平台下的程序员得益于 Visual Studio 这种强大的 IDE，在处理动态链接库时颇为方便，那么对 Linux 程序员来说，这便是一场噩梦。他们需要手动集成各种第三方库，有的是开源的，有的是非开源的，有的只提供了源代码，有的只提供了二进制文件，有的文档详尽，有的文档匮乏……他们永远不知道会遇到什么情况。即便是使用 IDE 的初级程序员，有时也不得不手动解决各种链接问题，但往往由于他们满足于 IDE 的良好封装，而忽略其底层编译链接的细节，因而在处理这些问题时更加盲目。在这种情况下，对链接和装载（其实也包括 C/C++ 语言本身以及编译过程）认识并不深刻的那些程序员在日常工作中很容易在处理这类问题时“触礁”。由于缺乏系统认知，他们需要在一次又一次的失败实践中总结，这会走许多弯路。

由于我本人在大规模分布式实时系统研发方面具有一定的经验，因此许多人在 C/C++ 开发中遇到问题时会咨询我，其中一部分问题出在 C++ 语法上，而更多的问题则出在链接和装载上。本书在这些方面进行了翔实总结和讨论。与纯粹讲解理论与技术细节的书不同，本书一方面阐述基本的理论，另一方面则聚焦于 C/C++ 使用静态库和动态库的一些注意事项，并举例说明如何解决实际的链接与装载问题。此外，本书尽量使用通俗易懂的语言来阐述这些知识，并补充了大量示例，避免让读者纠结枯燥的理论。

相信你会和我一样，在本书中发掘出大量有价值的资料，以便在日后的工作中游刃有余，并在处理 C/C++ 相关问题的时候厘清思路，排除障碍。

在翻译本书的过程中，我不仅查阅了大量国内外的相关资料，还与英文原著作者进行了深入沟通，力求做到专业词汇准确权威，书本内容正确，意译部分既无偏差又不失原著意境。在翻译过程中得到了很多人的帮助，这里一一感谢。感谢我的家人，他们是我学习和前进的动力。感谢鲁昌华教授，他在我的成长道路上给予了很大的支持和鼓励。感谢我在思科系统（中国）研发的同事们，他们在我的学习、工作中给予了很大帮助。感谢我的好友金柳颀，感谢他在我翻译本书过程中与我的通力合作。还要感谢机械工业出版社的陈佳媛编辑

对我的信任。

现在我怀着期盼和忐忑的心情将这本译著呈献给大家，我渴望得到你的认可，更渴望和你成为朋友，如果你有任何问题或建议，请与我联系（samblg@me.com），让我们一起探讨、共同进步。

卢誉声

前　言 Preface

我花了相当长的时间才认识到，计算机编程艺术与烹饪艺术之间存在着惊人的相似性。

说到烹饪和编程的比较，我的脑海中首先浮现出来的是烹饪专家和程序员的工作目标非常相似：都是为了满足特定对象的需求。对一个厨师来说，他的服务对象是食客，他需要使用大量的食材，一方面满足人们的温饱与营养需求，另一方面则要让人们享受到美食带来的快乐。而对一个程序员来说，其服务对象是微处理器，程序员使用大量不同的程序来为其提供代码，不仅要让微处理器产生有意义的动作，还需要以最优的形式将代码交付给微处理器。

虽说这种对比看起来有些牵强而且过于简单，但我们会在本书中列出一些更加合适和更具说服力的对比。

介绍烹饪方法的食谱的数量繁多。几乎所有的流行杂志都会开设专栏来介绍形形色色的美食和烹饪方法，无论是快餐式的食谱还是精致复杂的食谱、注重食材的食谱还是注重搭配稀有食材的食谱，你都能够找到。

但如果你希望自己成为烹饪大师，就会发现很难找到可供参考的资料，比如食品行业经营（批量生产、饭店或餐饮企业的经营）、食品生产的供需管理、选料和食材保鲜方面的指南或资料。很显然，这就是业余烹饪爱好者和专业食品企业之间的区别。

这种情况与程序设计其实非常相似。

我们可以轻松地从成千上万的书籍、杂志、文章、网络论坛和博客中搜集到编程语言方面的各类信息，无论是入门教程，还是“谷歌编程面试指南”这样的技巧。

但是，这类主题所涵盖的内容只能满足成为专业软件工程师一半的要求。我们不能一直沉浸在因程序实际执行（且执行正确）而带来的喜悦当中，而需要着重考虑接下来的问题：如何组织代码结构以便将来修改、如何从功能模块中提取出可重用代码，以及如何让程序能够适应不同的运行环境（无论是不同的人类语言和字符，还是在不同的操作系统环境中

运行）。

相较于其他编程主题来说，人们很少讨论这类问题。时至今日，这类问题变成了只有计算机科学专业人士（绝大多数软件架构师和构建工程师）和大学课堂上讲解编译器、链接器设计时，才会了解的“黑科技”。

由于 Linux 市场份额增加，而且越来越多的人都将 Linux 作为其编程环境，这促使开发人员开始关注 Linux 编程的相关问题。与在一些封装良好的平台（在 Windows 和 Mac 平台上利用 IDE 和 SDK 将程序员从一些特定的编程细节问题中解放出来）开发软件的开发人员不同，Linux 开发人员在日常工作中需要将来自不同项目且编码风格迥异的代码组合起来，这需要开发人员充分理解编译器、链接器的内部工作机制和程序装载机制，以及不同库的设计细节和使用方法。

本书将许多零碎的知识点进行汇总，并讨论其中那些有价值的内容，这些内容则通过一系列精心设计的简单示例进行验证。需要注意的是，本书的作者并非计算机科学科班出身。20 世纪 90 年代末至今的数字革命中，作者作为电气工程师供职于硅谷的一家多媒体行业高新技术企业，并因此掌握了相关领域的知识。希望本书的主题和内容能够让更多读者受益。

读者对象

作为一名软件设计实践顾问（虽然很忙，但我还是非常自豪的），我经常会与不同专业背景和资历的人群沟通。我在工作周中需要经常在不同的办公环境中工作，因此接触了许多开发人员（绝大多数来自硅谷），这也让我更加了解了本书的受众群体。其中包括以下几类人群。

- 第一类受众群体是来自不同工程领域的 C/C++ 开发人员（电气工程、机械、机器人技术和系统控制、航天、物理和化学等领域），这类人需要在日常工作中通过编程来解决问题。对缺乏正规计算机科学课程和理论教学的人来说，本书所提供的资料弥足珍贵。
- 第二类受众群体是具有计算机科学教育背景的初级程序员。本书能够帮助大家将主修课程中学到的知识具体化，并注重实践。将第 12 章 ~ 第 14 章的内容作为手册查阅，对资深工程师而言，也会有所受益。
- 第三类受众群体是操作系统集成和定制的爱好者。理解二进制文件及其内部工作机制将有助于在解决问题的过程中扫除障碍。

关于本书

我最初并没有计划去写这么一本书，甚至都没有打算写一本计算机科学领域的书。（我有

可能会去编写信号处理或程序设计艺术方面的书，但编写计算机科学方面的书？不……）

在职业生涯中，我经常处理当时我认为别人已经解决好的一些问题，而实际上这些问题并没有得到根本性解决，这是我写作此书的唯一原因。

很久以前，我决定成为一名从事高科技领域的“刺客”，将许多看似平静且体面的高科技公司从“恐怖分子”——复杂多媒体设计问题和大量严重缺陷所造成的破坏中解救出来。选择这样一个职业的结果就是，我并没有多少时间处理自己的生活，比如说孩子们想吃鸡肉而不是豌豆，我很难满足他们的这些需求。虽然我更倾向于使用傅里叶变换算法、小波、Z 变换、FIR 和 IIR 滤波器、倍频程、半音程、插值和抽取算法来解决问题（与 C/C++ 编程一起使用），但我还是要解决那些我并不喜欢解决的问题。总要有人去做这些事情吧。

出乎意料的是，在搜索一些非常简单明了的问题的答案时，我只能找到一些散乱的网络文章，而且绝大多数都只是泛泛而谈。我很耐心地把这些散乱的内容组织到一起，不仅完成了我手头的设计任务，而且学习总结了很多资料。

在一个天朗气清的日子里，我开始整理设计笔记（记录我工作中经常遇到的一些问题和解决方案）。但在整理工作完成的时候，这些笔记看起来就像……嗯……就像一本书——就是这本书。

不管怎么说……

就目前就业市场的情况而言，我认为（自 2005 年左右开始）熟悉 C/C++ 语言的复杂性，甚至是算法、数据结构和设计模式，对于找到一份好工作都是远远不够的。

在开源盛行的今天，专业开发人员在日常工作中所编写的代码越来越少，取而代之的是将现有代码集成到项目中。这不仅要求开发人员能够读懂其他人编写的代码（使用不同的代码风格和实践），还需要了解如何才能以最好的方式将现有的包（绝大多数以二进制文件（库）和导出头文件的形式提供）集成到代码中。

我希望本书能够兼具教学（对亟须这些知识的读者而言）和快速查询的功能（对分析 C/C++ 二进制文件相关工作的工程师而言）。

为何采用 Linux 进行演示？

选择 Linux 并非我个人的偏好。实际上了解我的人都知道，我过去是多么喜欢使用 Windows 作为开发环境（原本这是我首选的设计平台），原因是 Windows 平台具有编写良好的文档、完善的支持和符合规范的认证组件。我设计过许多专业化软件（曾为 Palm 公司设计开发了 Windows Mobile 平台的 GraphEdit，其中包含了许多最为复杂的功能，随后又开发了多个媒体格式和 DSP 分析软件），在当时我对 Windows 的技术了如指掌，并感叹 Windows 相

关技术所带来的改变。

与此同时，Linux 的时代到来了。有关 Linux 的技术随处可见，而对开发人员来说，也必须顺应这种趋势去学习和使用它。

Linux 软件开发环境具有开放、透明和简单明了的特点。在 Linux 中，我们可以对每个程序设计阶段进行控制。同时，Linux 提供了完善的文档，再加上网络上提供的资源，就可以轻松地使用 GNU 工具链。

实际上，由于 Linux C/C++ 开发经验可以直接适用于 Mac OS 平台的底层开发，因此我最终决定选用 Linux/GNU 作为本书所涵盖的主要开发环境。

别急！Linux 与 GNU 完全是两回事！

实际上，Linux 是内核，而 GNU 中包含了 Linux 内核之上的所有软件。除了 GNU 编译器（可以在其他操作系统上使用，比如 Windows 上的 MinGW）以外，在绝大多数情况下，GNU 与 Linux 的关系其实非常紧密。为了简单起见，同时为了符合一般开发人员对开发场景的认识，特别是为了将 Linux 与 Windows 进行对比，本书将 GNU 与 Linux 作为一个整体，简称为“Linux”。

章节概览

第 2 章 ~ 第 5 章讲解的内容主要为后续内容做铺垫。拥有计算机科学背景的读者可以快速阅读这些章节（幸运的是，这些章节的内容并不长）。实际上，任何计算机科学方面的教科书都会对这些内容进行类似介绍，而且内容会更为详细。我个人推荐由 Bryant 和 O’Hallaron 编写的《深入理解计算机系统》[⊖]（《Computer Systems—A Programmer’s Perspective》）一书，原因是本书对很多主题都进行了非常有条理的梳理和总结。

第 6 章 ~ 第 11 章是本书的核心章节。为求整体内容简洁明了，我花费了相当大的精力，并尝试使用一些日常生活中常见事物的文字和图片，来阐述那些最为重要的核心概念。如果你不是计算机科学科班出身，那么有必要先理解这些内容。其实这些章节是本书主题的要点。

第 12 章 ~ 第 14 章主要概括了一些实践方面的内容，便于读者快速查找相关的概念。这些章节针对一些特定平台的二进制文件分析工具进行了总结，然后在实践部分涵盖了完成独立任务的方法。

⊖ 本书中文版由机械工业出版社引进并出版，ISBN：978-7-111-32133-0。

目　录 *Contents*

译者序
前言

第 1 章　多任务操作系统基础 …… 1

1.1　一些有用的抽象概念 …… 1
1.2　存储器层次结构与缓存策略 …… 2
1.3　虚拟内存 …… 3
1.4　虚拟地址 …… 5
1.5　进程的内存划分方案 …… 5
1.6　二进制文件、编译器、链接器与装载器的作用 …… 6
1.7　小结 …… 7

第 2 章　程序生命周期阶段基础 …… 8

2.1　基本假设 …… 8
2.2　编写代码 …… 9
2.3　编译阶段 …… 11
2.3.1　基本概念 …… 11
2.3.2　相关概念 …… 11
2.3.3　编译的各个阶段 …… 12
2.3.4　目标文件属性 …… 23
2.3.5　编译过程的局限性 …… 24
2.4　链接 …… 26
2.4.1　链接阶段 …… 26
2.4.2　链接器视角 …… 31
2.5　可执行文件属性 …… 33
2.5.1　各种节的类型 …… 34
2.5.2　各种符号类型 …… 36

第 3 章　加载程序执行阶段 …… 37

3.1　shell 的重要性 …… 37
3.2　内核的作用 …… 39
3.3　装载器的作用 …… 39
3.3.1　装载器视角下的二进制文件（节与段） …… 39
3.3.2　程序加载阶段 …… 40
3.4　程序执行入口点 …… 43
3.4.1　装载器查找入口点 …… 43
3.4.2　_start() 函数的作用 …… 43
3.4.3　__libc_start_main() 函数的作用 …… 44
3.4.4　栈和调用惯例 …… 44

第 4 章　重用概念的作用 …… 46

4.1　静态库 …… 46

4.2 动态库 …… 48
4.2.1 动态库和共享库 …… 49
4.2.2 动态链接详解 …… 51
4.2.3 Windows 平台中动态链接的特点 …… 54
4.2.4 动态库的特点 …… 56
4.2.5 应用程序二进制接口（ABI）…… 56
4.3 静态库和动态库对比 …… 57
4.3.1 导入选择条件的差异 …… 57
4.3.2 部署难题 …… 59
4.4 一些有用的类比 …… 61
4.5 结论：二进制复用概念所产生的影响 …… 63

第 5 章 使用静态库 …… 64
5.1 创建静态库 …… 64
5.1.1 创建 Linux 静态库 …… 64
5.1.2 创建 Windows 静态库 …… 65
5.2 使用静态库 …… 65
5.3 静态库设计技巧 …… 66
5.3.1 丢失符号可见性和唯一性的可能性 …… 66
5.3.2 静态库使用禁忌 …… 67
5.3.3 静态库链接的具体规则 …… 68
5.3.4 将静态库转换成动态库 …… 68
5.3.5 静态库在 64 位 Linux 平台上的问题 …… 68

第 6 章 设计动态链接库：基础篇 …… 70
6.1 创建动态链接库 …… 70
6.1.1 在 Linux 中创建动态库 …… 70
6.1.2 在 Windows 中创建动态链接库 …… 72
6.2 设计动态库 …… 75
6.2.1 设计二进制接口 …… 75
6.2.2 设计应用程序的二进制接口 …… 79
6.2.3 控制动态库符号的可见性 …… 82
6.2.4 完成链接需要满足的条件 …… 94
6.3 动态链接模式 …… 94
6.3.1 加载时动态链接 …… 95
6.3.2 运行时动态链接 …… 95
6.3.3 比较两种动态链接模式 …… 98

第 7 章 定位库文件 …… 99
7.1 典型用例场景 …… 99
7.1.1 开发用例场景 …… 99
7.1.2 用户运行时用例场景 …… 100
7.2 构建过程中库文件的定位规则 …… 101
7.2.1 Linux 平台构建过程中的库文件定位规则 …… 101
7.2.2 Windows 构建过程中的库文件定位规则 …… 105
7.3 运行时动态库文件的定位规则 …… 109
7.3.1 Linux 运行时动态库文件的定位规则 …… 110
7.3.2 Windows 运行时动态库文件的定位规则 …… 114
7.4 示例：Linux 构建时与运行时的库文件定位 …… 115

第 8 章 动态库的设计：进阶篇 …… 119
8.1 解析内存地址的必要性 …… 119

8.2 引用解析中的常见问题 …… 120
8.3 地址转换引发的问题 …… 122
8.3.1 情景 1：客户二进制程序需要知道动态库符号地址 …… 122
8.3.2 情景 2：被装载的库不需要知道其自身符号地址 …… 123
8.4 链接器 – 装载器协作 …… 124
8.4.1 总体策略 …… 125
8.4.2 具体技术 …… 126
8.4.3 链接器重定位提示概述 …… 127
8.5 链接器 – 装载器协作实现技术 …… 128
8.5.1 装载时重定位（LTR）…… 129
8.5.2 位置无关代码（PIC）…… 129

第 9 章 动态链接时的重复符号处理 …… 134

9.1 重复的符号定义 …… 134
9.2 重复符号的默认处理 …… 137
9.3 在动态库链接过程中处理重复符号 …… 140
9.3.1 处理重复符号问题的一般策略 …… 142
9.3.2 链接器解析动态库重复符号的模糊算法准则 …… 143
9.4 特定重复名称案例分析 …… 144
9.4.1 案例 1：客户二进制文件符号与动态库 ABI 函数冲突 …… 144
9.4.2 案例 2：不同动态库的 ABI 符号冲突 …… 147
9.4.3 案例 3：动态库 ABI 符号和另一个动态库局部符号冲突 …… 151
9.4.4 案例 4：两个未导出的动态库符号冲突 …… 153
9.5 小提示：链接并不提供任何类型的命名空间继承 …… 161

第 10 章 动态库的版本控制 …… 162

10.1 主次版本号与向后兼容性 …… 162
10.1.1 主版本号变更 …… 162
10.1.2 次版本号变更 …… 163
10.1.3 修订版本号 …… 163
10.2 Linux 动态库版本控制方案 …… 163
10.2.1 基于 soname 的版本控制方案 …… 163
10.2.2 基于符号的版本控制方案 …… 169
10.3 Windows 动态库版本控制 …… 190
10.3.1 DLL 版本信息 …… 191
10.3.2 指定 DLL 版本信息 …… 192
10.3.3 查询并获取 DLL 版本信息 …… 193

第 11 章 动态库：其他主题 …… 202

11.1 插件 …… 202
11.1.1 导出规则 …… 203
11.1.2 一些流行的插件架构 …… 204
11.2 提示和技巧 …… 204
11.2.1 使用动态库的实际意义 …… 204
11.2.2 其他主题 …… 205

第 12 章 Linux 工具集 …… 211

12.1 快速查看工具 …… 211

12.1.1 file 实用程序 …… 211
12.1.2 size 实用程序 …… 212
12.2 详细信息分析工具 …… 212
12.2.1 ldd …… 212
12.2.2 nm …… 214
12.2.3 objdump …… 215
12.2.4 readelf …… 223
12.3 部署阶段工具 …… 229
12.3.1 chrpath …… 229
12.3.2 patchelf …… 230
12.3.3 strip …… 231
12.3.4 ldconfig …… 231
12.4 运行时分析工具 …… 232
12.4.1 strace …… 232
12.4.2 addr2line …… 233
12.4.3 gdb（GNU 调试器）…… 233
12.5 静态库工具 …… 234

第 13 章 平台实践 …… 238

13.1 链接过程调试 …… 238
13.2 确定二进制文件类型 …… 239
13.3 确定二进制文件入口点 …… 240
13.3.1 获取可执行文件入口点 …… 240
13.3.2 获取动态库入口点 …… 240
13.4 列出符号信息 …… 241
13.5 查看节的信息 …… 242
13.5.1 列出所有节的信息 …… 242
13.5.2 查看节的信息 …… 242
13.6 查看段的信息 …… 243
13.7 反汇编代码 …… 244
13.7.1 反汇编二进制文件 …… 244
13.7.2 反汇编正在运行的进程 …… 244
13.8 判断是否为调试构建 …… 244
13.9 查看加载时依赖项 …… 245
13.10 查看装载器可以找到的库文件 …… 245
13.11 查看运行时动态链接的库文件 …… 245
13.11.1 strace 实用程序 …… 245
13.11.2 LD_DEBUG 环境变量 …… 246
13.11.3 /proc/<ID>/maps 文件 …… 246
13.11.4 lsof 实用程序 …… 247
13.11.5 通过编程方式查看 …… 248
13.12 创建和维护静态库 …… 251

第 14 章 Windows 工具集 …… 252

14.1 库管理器（lib.exe）…… 252
14.1.1 使用 lib.exe 处理静态库 …… 253
14.1.2 使用 lib.exe 处理动态库（导入库生成工具）…… 257
14.2 dumpbin 实用程序 …… 258
14.2.1 确定二进制文件类型 …… 258
14.2.2 查看 DLL 的导出符号 …… 258
14.2.3 查看节的信息 …… 259
14.2.4 反汇编代码 …… 262
14.2.5 确定是否使用了调试模式构建 …… 263
14.2.6 查看加载时依赖项 …… 265
14.3 Dependency Walker 工具 …… 265

第 1 章 Chapter 1

多任务操作系统基础

构建可执行文件的精髓在于实现对程序执行过程的最大化控制。为了能够真正理解可执行文件结构中各部分的功能和含义，我们有必要对程序执行过程中的操作进行详细了解，操作系统内核和可执行文件中信息的相互作用在这里扮演了重要的角色。特别是在程序启动的最初阶段，在运行时环境数据（如用户配置和各种运行时事件等）对程序产生影响以前，这种相互作用通常会非常频繁。

为了能够真正理解可执行文件结构中各部分的功能和含义，首先要理解与程序运行相关的一些技术细节。本章将为读者着重阐述现代多任务操作系统的功能。

现代多任务操作系统在重要功能性方面的实现方法非常相似。因此，本章会把主要精力放在阐述与平台无关的一些概念上。除此之外，我们还将着重研究和分析一些特定平台的复杂性（将无所不在的 Linux 和 ELF 格式与 Windows 对比）。

1.1 一些有用的抽象概念

计算机技术领域的变化日新月异。集成电路技术带来的元件不仅种类繁多（光学元件、电磁元件和半导体元件）而且在功能性方面不断改进。按照摩尔定律，集成电路上可容纳的晶体管数目大约每隔两年便会增加一倍。而与晶体管数量密切相关的处理能力也将提升一倍。

经验告诉我们，若想要完全应对这种快速变化，唯一的方法就是在经常变动的实现层次之上，利用抽象和泛化的方法为计算机系统定义全局目标与体系结构。这种方法的核心在于描述抽象的方式，该方式要确保在去除相对无关的实现细节后，任何新的实现与核心定义都能够保持一致。整个计算机体系结构可以用图 1-1 展示的一组结构化抽象表示。

图 1-1 计算机体系结构抽象

在抽象的最底层，我们利用典型的字节流方式来处理各种类型的 I/O 设备（鼠标、键盘、控制杆、轨迹球、光笔、扫描仪、条形码扫描器、打印机、绘图仪、数码相机和网络摄像头）的输入和输出。事实上，在忽略不同设备用途、实现方法和功能性的情况下，对计算机系统设计而言，我们只需关注这些设备产生或接收（或两者皆有）的字节流。

下一个抽象层次是虚拟内存，它用来表示系统中多种不同的存储资源。虚拟内存是本书将要讨论的重要主题之一。实际上这种特定抽象的方式代表了多种不同的物理存储设备，这不仅影响了实际的软硬件的设计，而且是设计编译器、链接器和装载器的基础。

下一个抽象层次是指令集，它用来对物理 CPU 进行抽象。虽然高级程序员一定对指令集所具备的功能及其带来的处理性能十分感兴趣，但就本书所希望讨论的主题来看，这部分抽象的细节不是本书详细讨论的重点。

最后一个抽象层次是操作系统的复杂性。通常来说，操作系统在某些方面的设计（主要是多任务机制）对软件的体系结构有着决定性的影响。对多方都需要访问的共享资源来说，需要有完善的实现方法才可以避免重复的无用代码问题，而这个问题直接为我们引出了共享库设计的概念。

在接着分析整个计算机系统的复杂性之前，我们先了解一下与存储器使用相关的一些关键问题。

1.2 存储器层次结构与缓存策略

在计算机系统中，有一些和存储器相关的趣事：

- 人们对存储器容量的需求总是无法满足，而且存储器容量总是供不应求。每当存储器技术在容量（更快的存储器的容量）方面取得重大飞跃的时候，就会发现其实有一些技术已经为此等候多时。这些技术在理论上完全可行，只是需要等到拥有足够数量的物理存储器供其使用时才能够被真正实现出来。
- 存储器技术似乎是导致处理器性能障碍的主要原因。这被称为“处理器与存储器之间

的速度鸿沟”(the processor-memory gap)。

- 存储器的访问速度与其存储容量成反比。通常来说，容量最大的存储设备的访问时间要比容量最小的存储设备慢上好几个数量级。

现在，我们从程序员、设计师和工程师的视角来简单地看一下整个系统。在理想情况下，我们希望系统能够以最快的速度访问所有可用的存储器，但其实我们都知道，这是不可能实现的。紧接着的一个问题就是：我们能否做些什么来改善这种情况呢？

一个细节让我们大可放心，那就是在实际情况下，系统并不总是使用所有的存储器，而仅仅是在某些时段内使用某一部分存储器。在这种情况下，我们只需为立即需要执行的程序预留最快的存储器，而让那些并非立即执行的代码或数据使用较慢的存储设备。当 CPU 从最快的存储器中获取计划立即需要执行的指令时，硬件设备会预测接下来会执行程序的哪一部分，并将这部分代码交给较慢的存储器，然后等待执行。在执行到存储在较慢存储器上的代码之前，这些代码会转存到较快的存储器中。这种策略称为“缓存”。

我们可以把缓存比作现实生活中一般家庭的食物供给。除非是生活在荒无人烟的地方，否则我们一般不会采购一整年所需的食物带回家。相反，我们通常会在家中安置一些比较大的存储设备（比如冰箱、食品储藏室和食品架），用来存放未来一到两周的食物。当意识到食物快吃完的时候，我们就会去食品杂货店采购恰好能够填满家中存储设备的食物。

程序的执行通常会受到一些外部因素的影响（用户设置只是其中之一），这让缓存机制的实现变得十分困难。程序执行流程（通过跳转和中断等指令的数量来衡量）的可确定性越高，缓存机制工作得越好。相反，若程序的执行流程发生变化，那么之前缓存的指令也就不再有效，因此这部分缓存的指令会被丢弃，而程序所需的那部分指令则需要重新从较慢的存储器中获取。

缓存策略的实现无处不在，横跨多个级别的存储器，如图 1-2 所示。

图 1-2　存储器的缓存层次结构原理图

1.3　虚拟内存

存储器缓存的通用方法在下一个体系结构层次中得到了具体实现，我们在这一架构层中用名为“进程”的抽象概念来表示正在运行的程序。

现代多任务操作系统的设计允许一个或多个用户并发地运行多个程序。对一个普通用户来说，同时运行多个应用程序（比如 Web 浏览器、编辑器、音乐播放器和日历）再正常不过了。

通过虚拟内存的概念，可以很好地解决内存需求与有限的内存容量之间的矛盾，我们可以把这个概念概括为以下几条原则：

- 实际上所有程序的内存配额是固定、均等和显式的。
  通常来说，操作系统会允许程序（进程）使用的内存容量大小为 2^N 字节，其中 N 就目前来说是 32 或者 64。该值是固定的，而且与系统中的物理内存大小无关。
- 物理内存的数量可能会发生变化。通常来说，可用内存的数量会比声明的进程地址空间小很多。而供运行中程序使用的物理内存数量通常也各不相同。
- 运行时的物理内存会被划分成数个小的分段（页），而每一个页都可用来同步执行程序。
- 正在运行的程序的完整内存布局会被保存在低速存储器（硬盘）中。只有那些当前即将被执行的一部分内存（代码和数据）才会被加载到物理内存的页中。

要实现真正的虚拟内存概念，我们需要借助多个系统资源的配合，比如硬件资源（硬件异常、硬件地址翻译）、硬盘（交换文件）以及最底层的操作系统软件（内核）。虚拟内存的概念如图 1-3 所示。

图 1-3　虚拟内存概念的实现

1.4　虚拟地址

虚拟地址的概念是虚拟内存实现中最为基础的一个部分，并在很多方面会对编译器和链接器的设计产生十分重要的影响。

一般来说，程序设计者完全不用关心程序运行时所占用的地址范围（至少，对绝大多数用户空间的应用程序来说确实如此，但内核模块是个例外）。取而代之的是，编程模型假定地址空间的范围在 0 到 2^N（虚拟地址范围）之间，而且这个规则对所有程序适用。

在代码开发过程中，为所有程序采用简单且统一的地址模式能够带来许多好处。采用这一模式的好处如下：

- 简化链接过程。
- 简化加载过程。
- 实现运行时进程间共享。
- 简化内存分配机制。

操作系统利用地址翻译机制，对固定地址范围内的程序内存进行实际的运行时分配。该实现通过名为内存管理单元（MMU）的硬件模块完成，不需要程序本身的任何介入。

在图 1-4 中，对虚拟地址机制和简单的物理地址方案（该方案现如今用于简单微控制系统领域）进行了比较。

图 1-4　物理地址和虚拟地址

1.5　进程的内存划分方案

在 1.4 节中，我们说明了为绝大多数程序设计者提供相同内存映射的可行性。本节的主题是讨论进程内存映射的内部组织细节。我们假定程序地址（程序员观察得到的地址）的范围在 0 到 2^N 之间，其中 N 是 32 或者 64。

不同的多任务或多用户操作系统拥有不同的内存映射布局。对于 Linux 进程的虚拟内存映射来说，其遵循如图 1-5 所示的映射方案。

无论平台的进程内存划分方案有多么特殊，下面几个内存映射的节（section）都是必须支持的：

- 代码节：该段包含了供 CPU 执行的机器码指令（.text 节）。
- 数据节：该段包含了供 CPU 操作的数据。通常来说，初始化数据（.data 节）、未初始化数据（.bss 节）和只读数据（.rdata 节）会保存在分离的节中。

- 堆：动态内存分配的区域。
- 栈：为各个函数提供独立的存储空间。
- 最上层部分属于内核区域，特定进程的环境变量就存放在该区域。

Gustavo Duarte 撰写了一篇文章来详细讨论该主题，读者可以从下面的网址中找到这篇文章：http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory。

图 1-5　Linux 进程的内存映射布局

1.6　二进制文件、编译器、链接器与装载器的作用

在前面的章节中，我们弄清楚了程序运行过程中的内存映射。随之而来的一个重要问题就是：程序运行过程中的内存映射是如何在运行时创建的？我们会在本节针对这一问题进行简单的解释。

粗略地讲：

- 程序的二进制文件中包含了程序运行过程中的内存映射布局的细节。
- 链接器创建了二进制文件的整体框架。要实现这项功能，链接器要对编译器生成的二进制文件进行合并，然后向各个内存映射节填充信息（代码和数据等信息）。
- 进程内存映射的初始化建立工作是由程序装载器这一系统工具完成的。在最简单的情况下，装载器会打开二进制可执行文件，读取节的相关信息，然后将这些信息载入进程内存映射结构中。

所有现代操作系统都是按照这种角色分离的方式设计的。

需要注意的是，本节中的这些简单描述离完整描述还差得很远。读者应该将本节的内容作为针对后续详细讨论的一个简要介绍，我们会在后面继续深入讨论特定主题时，对有关二进制文件和进程加载方面的内容进行更加细致的讲解。

1.7　小结

本章对现代多任务操作系统设计上的一些最基本的概念进行了概述。虚拟内存和虚拟地址这些基本概念不仅对程序的执行产生影响（会在下一章中详细讨论），还直接影响程序可执行文件的构建过程（稍后会详细讲解）。

Chapter 2

第 2 章

程序生命周期阶段基础

在前一章中，读者应该已经对现代多任务操作系统在程序执行中所起到的作用有了大致的了解。那么开发人员会很自然地想到下一个问题：若想让程序能够执行，需要做什么，如何做以及为何这样做？

就像一只蝴蝶的生命周期是从毛毛虫阶段开始的一样，程序的生命周期极大地依赖于二进制文件的内部构造。操作系统装载器会加载、解包这些数据，并将这些数据用于程序的执行阶段。本章随后即将讨论的内容将围绕准备数据和嵌入数据到二进制可执行文件结构体的方法展开，读者应该不会对此感到意外。我们假定程序是用 C/C++ 编写的。

为了完全理解整个生命周期过程，我们会对程序生命周期的剩余细节进行详细的分析，包括程序的加载与执行阶段。本书将会重点围绕以下几个程序生命周期进行深入的讨论：

1）编写源代码。

2）编译。

3）链接。

4）装载。

5）执行。

实际上，相比链接、装载和执行阶段来说，本章的内容侧重于讲解编译阶段的有关细节。本章也会稍微提及其余一些阶段（特别是链接阶段）的内容，但是你仅仅会了解到这些内容的冰山一角。在介绍完大多数和链接相关的基础知识之后，本书的剩余章节将会针对装载和执行阶段进行详细的阐述。

2.1 基本假设

尽管大多数读者很可能属于高级或专家级开发人员，但我们在一开始依然会使用一些相

当基础和简单的例子。在本章中，主要讨论一个非常简单且易于讲解的例子。这个演示项目由两个简单的源代码文件组成，我们首先分别编译这两个文件，然后将其链接到一起。该项目的代码可以确保编译和链接的复杂度是最简单合理的。

比较特别的是，在本例中不会链接外部库，尤其是不会进行动态链接。唯一的例外是会链接 C 运行时库（对于绝大多数用 C 编写的程序来说都会链接这个库）。正因为在 C 程序执行的生命周期中使用 C 运行时库是如此普遍，因此为了简单起见，我们会有目的性地忽略链接 C 运行时库的一些特定细节，并假定所有 C 运行时库代码都被“自动”地嵌入程序内存映射结构体中。

通过这种方法，我们就能用最为简单且清晰的方式，对程序构建中的一些主要问题进行详细的阐述。

2.2　编写代码

考虑到本书讨论的主题是程序的构建过程（也就是编写完代码之后的过程），所以我们不准备在编写源代码上花费太长的时间。

除了在极少数情况下使用脚本来生成源代码的情况外，本书均假定用户使用编辑器输入代码，这些代码是一些使用 ASCII 字符编写的满足所选编程语言语法规则的语句（我们这里用的是 C 和 C++）。你可以选择多种不同的编辑器，从最简单的 ASCII 文本编辑器到高级的集成开发环境（IDE）工具。考虑到本书的绝大多数读者都是富有经验的开发人员，所以在选择编辑器的问题上也没有必要花费那么多时间去一一讲解了。

但是，这里有一项编程实践会对接下来的构建过程中产生重要影响，因此需要我们额外关注。为了更好地对源代码进行组织，开发人员通常会遵循以下编程实践方法：将具有不同功能性的代码置于不同的文件中，这使得一个项目通常包含许多不同的源代码和头文件。

人们从早期微处理器的开发环境出现时就已经开始采用这项编程实践。从那时起，开发人员就把这个编程实践作为一项开发过程中的固定设计方案，事实证明这样可以很好地组织代码，而且使得代码维护工作变得轻松许多。

这项实用的编程实践会对开发过程产生深远的影响。在接下来的构建过程阶段中，你很快就会了解到这项实践会导致一些不可预测性，我们需要花些心思来解决这些问题。

概念解释：基于演示项目

为了更好地说明编译过程的复杂性，同时也让读者能够快速拥有一次动手实践的机会，本书提供了一个简单的演示项目。该项目的代码非常简单，只包含了一个头文件和两个源代码文件。虽说如此，但为了让读者理解整个过程，这个经过精心设计的演示项目也足以说明程序生命周期中的各个重点。

该项目由以下几个文件组成：

- 源代码文件 main.c，其中包含 main() 函数。
- 头文件 function.h，其中声明了 main() 函数中所调用的函数和使用的数据。
- 源代码文件 function.c，其中包含的代码实现 main() 函数引用的函数，并初始化 main() 函数引用的数据。

演示项目是在 Linux 环境中使用 gcc 编译器构建的。代码清单 2-1 到代码清单 2-3 列出了演示项目中用到的代码。

代码清单 2-1　function.h

```
#pragma once

#define FIRST_OPTION
#ifdef FIRST_OPTION
#define MULTIPLIER (3.0)
#else
#define MULTIPLIER (2.0)#endif

float add_and_multiply(float x, float y);
```

代码清单 2-2　function.c

```
int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= MULTIPLIER;
    return z;
}
```

代码清单 2-3　main.c

```
#include "function.h"
extern int nCompletionStatus = 0;
int main(int argc, char* argv[])
{
    float x = 1.0;
    float y = 5.0;
    float z;

    z = add_and_multiply(x,y);
    nCompletionStatus = 1;
    return 0;
}
```

2.3　编译阶段

在编写完源代码后，就可以开始进行代码的构建了，其中第一步就是编译。在涉及编译的复杂性之前，我们先来简单了解一些基本术语。

2.3.1　基本概念

从广义上来讲，编译其实就是将某种编程语言编写的源代码转换成另一种编程语言描述的源代码。下面罗列出的一些基本概念将有助于你理解整个编译过程：

- 编译器负责编译程序。
- 编译器的输入是一个编译单元。编译单元通常是一个包含源代码的文本文件。
- 一个程序通常会包括多个编译单元。尽管我们完全可以把整个工程的源代码放在单个文件中，但是我们有充足的理由（原因已在上一节中解释过）不这么做。
- 编译过程的输出是一系列二进制目标文件的集合，其中每一个目标文件对应一个作为输入的编译单元。
- 若想让程序能够执行，这些目标文件还需要经过另一个程序构建阶段的处理：链接。

图 2-1 展示了编译的概念。

图 2-1　编译阶段

2.3.2　相关概念

我们通常会遇到以下几种与编译器相关的用例：

- 编译：从严格意义上讲，编译指的是将高级语言编写的源代码翻译成低级语言（通常是指汇编语言，甚至是机器代码）描述的代码的过程。
- 交叉编译：如果代码在同一个平台（相同 CPU 或操作系统）上进行编译，生成的代码可以在其他平台（不同 CPU 或操作系统）上执行，那么这种编译过程就称为交叉编

译。通常会利用交叉编译在桌面操作系统（如 Linux 或 Windows）上生成嵌入式设备或移动设备的代码。

- 反编译（反汇编）：是指将低级语言编写的源代码转换成高级语言描述的源代码的过程。
- 语言翻译：是指将一种编程语言编写的源代码转换成另一种具有相同语言等级和复杂度的语言的过程。
- 语言重写：是指将语言表达式转换成另一种形式，以更好地适应特定任务的过程（比如优化）。

2.3.3 编译的各个阶段

编译过程本身不是一蹴而就的。事实上，该过程可以粗略地划分为几个阶段（预处理阶段、语言分析阶段、汇编阶段、优化阶段和代码生成阶段），接下来我们会针对这几个阶段进行详细的讨论。

1. 预处理阶段

编译过程的第一步是使用“预处理程序”这个特殊文本处理程序来处理源代码文件。预处理程序会执行下面列出的一项或多项操作：

- 将 #include 关键字标示的含有定义的文件（包含文件或头文件）包含到源代码文件中。
- 将 #define 语句指定的值转换成常量。
- 在代码中调用宏的位置将宏定义转换成代码。
- 根据 #if、#elif 和 #endif 指令的位置包含或排除特定部分的代码。

预处理程序输出的内容就是转换完成的 C/C++ 代码，这些输出结果会供下一编译阶段——语法分析使用。

演示项目：预处理阶段示例

gcc 编译器提供了一种模式，可以仅仅对输入文件执行预处理操作：

```
gcc -i <input file> -o <output preprocessed file>.i
```

除非额外指定参数，否则预处理器的输出文件名会和输入文件名相同，区别在于扩展名是 .i。对文件 function.c 执行预处理后的结果如代码清单 2-4 所示。

代码清单 2-4　function.i

```
# 1 "function.c"
# 1 "
# 1 "
# 1 "function.h" 1

# 11 "function.h"
float add_and_multiply(float x, float y);
# 2 "function.c" 2

int nCompletionStatus = 0;
```

```
float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= MULTIPLIER;
    return z;
}
```

如果向 gcc 传递一些额外参数，我们就能够得到更为简洁明了的预处理输出结果，比如增加以下参数：

```
gcc -E -P -i <input file> -o <output preprocessed file>.i
```

在代码清单 2-5 中，可以看到预处理文件的结果：

代码清单 2-5　function.i（精简版本）

```
float add_and_multiply(float x, float y);
int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= 3.0;
    return z;
}
```

我们可以清楚地看到，预处理程序利用 FIRST_OPTION[⊖]变量定义范围内的宏，将 MULTIPLIER 符号替换成了对应的 3.0 这个值。

2. 语言分析阶段

在这个阶段中，编译器会先将 C/C++ 代码转换成更易于处理的形式（删去注释和不必要的空格，以及从文本中提取符号等操作）。通过词法分析，就可以得到这种优化和精简后的源代码形式，词法分析的目的在于检查程序是否满足编程语言的语法规则。编译器会在检查到不满足语法规则错误的时候报告错误或发出警告。编译错误会导致编译过程中断，而编译警告则不一定。对编译警告来说，是否中断编译过程取决于用户的设置。

深入了解编译，我们可以将该过程细分为以下三个不同的阶段：

⊖ 作者原文为 USE_FIRST_OPTION，但代码里实际是 FIRST_OPTION，此处疑为作者笔误。——译者注

- 词法分析：将源代码分割成不可分割的单词。
- 语法分析：将提取出来的单词连接成单词序列，并根据编程语言规则验证其顺序是否合理。
- 语义分析：目的是发现符合语法规则的语句是否具有实际意义。比如，将两个整数相加并把结果赋值给一个对象的语句，虽然能通过语法规则的检查，但可能无法通过语义检查（除非该对象重载了赋值操作符）。

在语言分析阶段中，编译器可能更像一个“喜欢抱怨的人”，因为相比真正编译代码而言，编译器更倾向于检查处理过程中遇到的拼写错误或者其他错误。

3. 汇编阶段

当源代码经过校验，其中不包含任何语法错误后，编译器才会执行汇编阶段。在这个阶段中，编译器会将标准的语言集合转换成特定 CPU 指令集的语言集合。不同的 CPU 包含不同的功能性需求，通常会包含不同的指令集、寄存器和中断，这就解释了为什么不同的处理器要有不同的编译器对其支持。

演示项目：汇编阶段示例

gcc 编译器支持一种模式：将输入文件的源代码转换成对应 ASCII 编码的文本文件，其中包含了特定芯片或操作系统所对应汇编指令集的代码行。

```
$ gcc -S <input file> -o <output assembler file>.s
```

除非特别指定预处理程序的输出文件名，否则输出文件名和输入文件名是相同的，只不过扩展名变成了 .s。

这里生成的文件还不能供我们直接执行，该文件只是一个包含了人们可读的汇编指令助记符的文本文件，这些内容有助于开发者更好地理解编译过程的内部工作细节。

对于 x86 处理器体系结构来说，编译器支持两种指令格式，汇编代码会遵循其中一种：

- AT&T 格式。
- Intel 格式。

通过向 gcc 汇编器传递额外的命令行参数，我们可以选择特定的指令格式。具体选择哪种格式主要依据开发者的个人喜好而定。

AT&T 汇编格式示例

当通过下面的命令将 function.c 文件汇编成 AT&T 格式时：

```
$ gcc -S -masm=att function.c -o function.s
```

生成汇编输出文件的内容如代码清单 2-6 所示。

代码清单 2-6　function.s（AT&T 汇编格式）

```
        .file     "function.c"
        .globl    nCompletionStatus
```

```
        .bss
        .align 4
        .type     nCompletionStatus, @object
        .size     nCompletionStatus, 4
nCompletionStatus:
        .zero     4
        .text
        .globl    add
        .type     add, @function
add:
.LFB0:
        .cfi_startproc
        pushl     %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl      %esp, %ebp
        .cfi_def_cfa_register 5
        subl      $20, %esp
        flds      8(%ebp)
        fadds     12(%ebp)
        fstps     -4(%ebp)
        movl      -4(%ebp), %eax
        movl      %eax, -20(%ebp)
        flds      -20(%ebp)
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size     add, .-add
        .globl    add_and_multiply
        .type     add_and_multiply, @function
add_and_multiply:
.LFB1:
        .cfi_startproc
        pushl     %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl      %esp, %ebp
        .cfi_def_cfa_register 5
        subl      $28, %esp
        movl      12(%ebp), %eax
        movl      %eax, 4(%esp)
        movl      8(%ebp), %eax
        movl      %eax, (%esp)
        call      add
        fstps     -4(%ebp)
        flds      -4(%ebp)
        flds      .LC1
        fmulp     %st, %st(1)
        fstps     -4(%ebp)
        movl      -4(%ebp), %eax
        movl      %eax, -20(%ebp)
        flds      -20(%ebp)
        leave
        .cfi_restore 5
```

```
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE1:
        .size     add_and_multiply, .-add_and_multiply
        .section          .rodata
        .align 4
.LC1:
        .long     1077936128
        .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section          .note.GNU-stack,"",@progbits
```

Intel 汇编格式示例

当通过下面的命令将同一个 function.c 文件汇编成 Intel 格式时：

```
$ gcc -S -masm=intel function.c -o function.s
```

生成汇编输出文件的内容如代码清单 2-7 所示。

代码清单 2-7　function.s(Intel 汇编格式)

```
        .file     "function.c"
        .intel_syntax noprefix
        .globl    nCompletionStatus
        .bss
        .align 4
        .type     nCompletionStatus, @object
        .size     nCompletionStatus, 4
nCompletionStatus:
        .zero     4
        .text
        .globl    add
        .type     add, @function
add:
.LFB0:
        .cfi_startproc
        push      ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov       ebp, esp
        .cfi_def_cfa_register 5
        sub       esp, 20
        fld       DWORD PTR [ebp+8]
        fadd      DWORD PTR [ebp+12]
        fstp      DWORD PTR [ebp-4]
        mov       eax, DWORD PTR [ebp-4]
        mov       DWORD PTR [ebp-20], eax
        fld       DWORD PTR [ebp-20]
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size     add, .-add
```

```
        .globl    add_and_multiply
        .type     add_and_multiply, @function
add_and_multiply:
.LFB1:
        .cfi_startproc
        push      ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov       ebp, esp
        .cfi_def_cfa_register 5
        sub       esp, 28
        mov       eax, DWORD PTR [ebp+12]
        mov       DWORD PTR [esp+4], eax
        mov       eax, DWORD PTR [ebp+8]
        mov       DWORD PTR [esp], eax
        call      add
        fstp      DWORD PTR [ebp-4]
        fld       DWORD PTR [ebp-4]
        fld       DWORD PTR .LC1
        fmulp     st(1), st
        fstp      DWORD PTR [ebp-4]
        mov       eax, DWORD PTR [ebp-4]
        mov       DWORD PTR [ebp-20], eax
        fld       DWORD PTR [ebp-20]
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE1:
        .size     add_and_multiply, .-add_and_multiply
        .section          .rodata
        .align 4
.LC1:
        .long     1077936128
        .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section          .note.GNU-stack,"",@progbits
```

4. 优化阶段

当由源代码文件生成最初版本的汇编代码后，优化过程就开始了，这可以将程序的寄存器使用率最小化。此外，通过分析能够预测出实际上不需要执行的部分代码，并将其删去。

5. 代码生成阶段

最后到生成编译输出目标文件的阶段，其中每一个目标文件对应一个编译单元。汇编指令（用可读的 ASCII 码编写）会在此阶段转换成对应机器指令（操作码）的二进制值，并写入目标文件的特定位置。

我们创建的目标文件还不足以让处理器执行。究其原因其实就是本书将要讨论的主题。而现在，我们亟须对单个目标文件进行分析。

作为二进制文件的目标文件，与预处理阶段和汇编阶段生成的可读的 ASCII 码输出文件有着本质区别。当我们想要更深入地研究目标文件内容时，这种差别就变得尤其明显。

若想深入了解目标文件的内容，除了直接使用十六进制编辑器（除非你以编写编译器为生，否则帮助也不大）以外，还可以利用名为反汇编的过程来帮助你进行分析。

编译是一种将基于 ASCII 码编写的文件转换成适合特定机器运行的二进制文件的过程，而进行反汇编则是将二进制文件转换成便于软件开发者阅读的基于 ASCII 码的文件，这么做看起来像是在走回头路。不过幸运的是，这种走回头路的方式仅仅是为了方便开发者阅读和理解目标文件的内容，通常情况下我们是不会进行反汇编操作的。

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  6c 01 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |l.......4.....(.|
00000030  0d 00 0a 00 55 89 e5 83  ec 14 d9 45 08 d8 45 0c  |....U......E..E.|
00000040  d9 5d fc 8b 45 fc 89 45  ec d9 45 ec c9 c3 55 89  |.]..E..E..E...U.|
00000050  e5 83 ec 1c 8b 45 0c 89  44 24 04 8b 45 08 89 04  |.....E..D$..E...|
00000060  24 e8 fc ff ff ff d9 5d  fc d9 45 fc d9 05 00 00  |$......]..E.....|
00000070  00 00 de c9 d9 5d fc 8b  45 fc 89 45 ec d9 45 ec  |.....]..E..E..E.|
00000080  c9 c3 00 00 00 00 40 40  00 47 43 43 3a 20 28 55  |......@@.GCC: (U|
00000090  62 75 6e 74 75 2f 4c 69  6e 61 72 6f 20 34 2e 36  |buntu/Linaro 4.6|
000000a0  2e 33 2d 31 75 62 75 6e  74 75 35 29 20 34 2e 36  |.3-1ubuntu5) 4.6|
000000b0  2e 33 00 00 14 00 00 00  00 00 00 00 01 7a 52 00  |.3...........zR.|
000000c0  01 7c 08 01 1b 0c 04 04  88 01 00 00 1c 00 00 00  |.|..............|
000000d0  1c 00 00 00 00 00 00 00  1a 00 00 00 00 41 0e 08  |.............A..|
000000e0  85 02 42 0d 05 56 c5 0c  04 04 00 00 1c 00 00 00  |..B..V..........|
000000f0  3c 00 00 00 1a 00 00 00  34 00 00 00 00 41 0e 08  |<.......4....A..|
00000100  85 02 42 0d 05 70 c5 0c  04 04 00 00 00 2e 73 79  |..B..p........sy|
00000110  6d 74 61 62 00 2e 73 74  72 74 61 62 00 2e 73 68  |mtab..strtab..sh|
00000120  73 74 72 74 61 62 00 2e  72 65 6c 2e 74 65 78 74  |strtab..rel.text|
00000130  00 2e 64 61 74 61 00 2e  62 73 73 00 2e 72 6f 64  |..data..bss..rod|
00000140  61 74 61 00 2e 63 6f 6d  6d 65 6e 74 00 2e 6e 6f  |ata..comment..no|
00000150  74 65 2e 47 4e 55 2d 73  74 61 63 6b 00 2e 72 65  |te.GNU-stack..re|
00000160  6c 2e 65 68 5f 66 72 61  6d 65 00 00 00 00 00 00  |l.eh_frame......|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000190  00 00 00 00 1f 00 00 00  01 00 00 00 06 00 00 00  |................|
000001a0  00 00 00 00 34 00 00 00  4e 00 00 00 00 00 00 00  |....4...N.......|
000001b0  00 00 00 00 04 00 00 00  00 00 00 00 1b 00 00 00  |................|
000001c0  09 00 00 00 00 00 00 00  00 00 00 00 68 04 00 00  |............h...|
000001d0  10 00 00 00 0b 00 00 00  01 00 00 00 04 00 00 00  |................|
000001e0  08 00 00 00 25 00 00 00  01 00 00 00 03 00 00 00  |....%...........|
000001f0  00 00 00 00 84 00 00 00  00 00 00 00 00 00 00 00  |................|
00000200  00 00 00 00 04 00 00 00  00 00 00 00 2b 00 00 00  |............+...|
00000210  08 00 00 00 03 00 00 00  00 00 00 00 84 00 00 00  |................|
00000220  04 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |................|
00000230  00 00 00 00 30 00 00 00  01 00 00 00 02 00 00 00  |....0...........|
00000240  00 00 00 00 84 00 00 00  04 00 00 00 00 00 00 00  |................|
00000250  00 00 00 00 04 00 00 00  00 00 00 00 38 00 00 00  |............8...|
00000260  01 00 00 00 30 00 00 00  00 00 00 00 88 00 00 00  |....0...........|
00000270  2b 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  |+...............|
00000280  01 00 00 00 41 00 00 00  01 00 00 00 00 00 00 00  |....A...........|
00000290  00 00 00 00 b3 00 00 00  00 00 00 00 00 00 00 00  |................|
000002a0  00 00 00 00 01 00 00 00  00 00 00 00 55 00 00 00  |............U...|
000002b0  01 00 00 00 02 00 00 00  00 00 00 00 b4 00 00 00  |................|
000002c0  58 00 00 00 00 00 00 00  00 00 00 00 04 00 00 00  |X...............|
000002d0  00 00 00 00 51 00 00 00  09 00 00 00 00 00 00 00  |....Q...........|
000002e0  00 00 00 00 78 04 00 00  10 00 00 00 0b 00 00 00  |....x...........|
```

图 2-2　以二进制方式显示目标文件的内容

演示项目：编译示例

gcc 编译器可以用于完整的编译（其中包括预处理、汇编和编译阶段），该过程中会生成二进制目标文件（标准扩展名是 .o），其结构遵循 ELF 格式规范。

除了一些常用的描述信息（头、表等信息）以外，目标文件中还包含所有相关的节（.text、.code、.bss）。如果只需执行编译过程（不执行链接），可以使用下面的命令行：

```
$ gcc -c <input file> -o <output file>.o
```

除非额外指定文件名，否则输出文件名和输入文件名是相同的，区别在于扩展名是 .o。

生成好的目标文件的内容并不能直接用文本编辑器查看。如果用十六进制编辑器或阅读器就会稍微好一点，至少它们能够区分不可见字符和缺失的换行符。图 2-2 展示了以二进制方式显示的目标文件 function.o 的内容，该文件是编译演示项目的文件 function.c 后生成的。

很显然，如果我们只去看目标文件的十六进制值，是很难看出其中包含哪些内容的。但通过反汇编，我们可以了解到其中的更多细节。

objdump（在常用的 binutils 包中）是一款专门用于反汇编二进制文件的 Linux 工具，除此之外它还有很多其他功能。该工具不仅能转换目标平台的二进制机器指令序列，还会指出指令所在的地址。

我们应该不会感到意外，该工具同时支持 AT&T(默认）和 Intel 两种风格汇编代码的输出。

通过简单执行 objdump 命令：

```
$ objdump -D <input file>.o
```

你将会在终端屏幕上得到以下输出内容：

function.o 文件的反汇编输出内容 (AT&T 汇编格式)

```
function.o:     file format elf32-i386

Disassembly of section .text:

00000000 <add>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  83 ec 14                sub    $0x14,%esp
   6:  d9 45 08                flds   0x8(%ebp)
   9:  d8 45 0c                fadds  0xc(%ebp)
   c:  d9 5d fc                fstps  -0x4(%ebp)
   f:  8b 45 fc                mov    -0x4(%ebp),%eax
  12:  89 45 ec                mov    %eax,-0x14(%ebp)
  15:  d9 45 ec                flds   -0x14(%ebp)
  18:  c9                      leave
  19:  c3                      ret

0000001a <add_and_multiply>:
  1a:  55                      push   %ebp
  1b:  89 e5                   mov    %esp,%ebp
  1d:  83 ec 1c                sub    $0x1c,%esp
  20:  8b 45 0c                mov    0xc(%ebp),%eax
  23:  89 44 24 04             mov    %eax,0x4(%esp)
```

```
  27:   8b 45 08                mov    0x8(%ebp),%eax
  2a:   89 04 24                mov    %eax,(%esp)
  2d:   e8 fc ff ff ff          call   2e <add_and_multiply+0x14>
  32:   d9 5d fc                fstps  -0x4(%ebp)
  35:   d9 45 fc                flds   -0x4(%ebp)
  38:   d9 05 00 00 00 00       flds   0x0
  3e:   de c9                   fmulp  %st,%st(1)
  40:   d9 5d fc                fstps  -0x4(%ebp)
  43:   8b 45 fc                mov    -0x4(%ebp),%eax
  46:   89 45 ec                mov    %eax,-0x14(%ebp)
  49:   d9 45 ec                flds   -0x14(%ebp)
  4c:   c9                      leave
  4d:   c3                      ret
Disassembly of section .bss:

00000000 <nCompletionStatus>:
   0:   00 00                   add    %al,(%eax)
        ...

Disassembly of section .rodata:

00000000 <.rodata>:
   0:   00 00                   add    %al,(%eax)
   2:   40                      inc    %eax
   3:   40                      inc    %eax

Disassembly of section .comment:

00000000 <.comment>:
   0:   00 47 43                add    %al,0x43(%edi)
   3:   43                      inc    %ebx
   4:   3a 20                   cmp    (%eax),%ah
   6:   28 55 62                sub    %dl,0x62(%ebp)
   9:   75 6e                   jne    79 <add_and_multiply+0x5f>
   b:   74 75                   je     82 <add_and_multiply+0x68>
   d:   2f                      das
   e:   4c                      dec    %esp
   f:   69 6e 61 72 6f 20 34    imul   $0x34206f72,0x61(%esi),%ebp
  16:   2e 36 2e 33 2d 31 75    cs ss xor %cs:%ss:0x75627531,%ebp
  1d:   62 75
  1f:   6e                      outsb  %ds:(%esi),(%dx)
  20:   74 75                   je     97 <add_and_multiply+0x7d>
  22:   35 29 20 34 2e          xor    $0x2e342029,%eax
  27:   36 2e 33 00             ss xor %cs:%ss:(%eax),%eax

Disassembly of section .eh_frame:

00000000 <.eh_frame>:
   0:   14 00                   adc    $0x0,%al
   2:   00 00                   add    %al,(%eax)
   4:   00 00                   add    %al,(%eax)
   6:   00 00                   add    %al,(%eax)
   8:   01 7a 52                add    %edi,0x52(%edx)
   b:   00 01                   add    %al,(%ecx)
   d:   7c 08                   jl     17 <.eh_frame+0x17>
   f:   01 1b                   add    %ebx,(%ebx)
  11:   0c 04                   or     $0x4,%al
```

```
  13:  04 88                    add    $0x88,%al
  15:  01 00                    add    %eax,(%eax)
  17:  00 1c 00                 add    %bl,(%eax,%eax,1)
  1a:  00 00                    add    %al,(%eax)
  1c:  1c 00                    sbb    $0x0,%al
  1e:  00 00                    add    %al,(%eax)
  20:  00 00                    add    %al,(%eax)
  22:  00 00                    add    %al,(%eax)
  24:  1a 00                    sbb    (%eax),%al
  26:  00 00                    add    %al,(%eax)
  28:  00 41 0e                 add    %al,0xe(%ecx)
  2b:  08 85 02 42 0d 05        or     %al,0x50d4202(%ebp)
  31:  56                       push   %esi
  32:  c5 0c 04                 lds    (%esp,%eax,1),%ecx
  35:  04 00                    add    $0x0,%al
  37:  00 1c 00                 add    %bl,(%eax,%eax,1)
  3a:  00 00                    add    %al,(%eax)
  3c:  3c 00                    cmp    $0x0,%al
  3e:  00 00                    add    %al,(%eax)
  40:  1a 00                    sbb    (%eax),%al
  42:  00 00                    add    %al,(%eax)
  44:  34 00                    xor    $0x0,%al
  46:  00 00                    add    %al,(%eax)
  48:  00 41 0e                 add    %al,0xe(%ecx)
  4b:  08 85 02 42 0d 05        or     %al,0x50d4202(%ebp)
  51:  70 c5                    jo     18 <.eh_frame+0x18>
  53:  0c 04                    or     $0x4,%al
  55:  04 00                    add    $0x0,%al
       ...
```

与上面执行的命令类似，我们可以指定 Intel 汇编风格：

```
$ objdump -D -M intel <input file>.o
```

你将会在终端屏幕上得到以下输出内容：

function.o 文件的反汇编输出内容（Intel 汇编格式）

```
function.o:     file format elf32-i386

Disassembly of section .text:

00000000 <add&gt:
   0:  55                       push   ebp
   1:  89 e5                    mov    ebp,esp
   3:  83 ec 14                 sub    esp,0x14
   6:  d9 45 08                 fld    DWORD PTR [ebp+0x8]
   9:  d8 45 0c                 fadd   DWORD PTR [ebp+0xc]
   c:  d9 5d fc                 fstp   DWORD PTR [ebp-0x4]
   f:  8b 45 fc                 mov    eax,DWORD PTR [ebp-0x4]
  12:  89 45 ec                 mov    DWORD PTR [ebp-0x14],eax
  15:  d9 45 ec                 fld    DWORD PTR [ebp-0x14]
  18:  c9                       leave
  19:  c3                       ret
0000001a <add_and_multiply>:
  1a:  55                      push   ebp
  1b:  89 e5                   mov    ebp,esp
  1d:  83 ec 1c                sub    esp,0x1c
```

```
  20:   8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
  23:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
  27:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
  2a:   89 04 24                mov    DWORD PTR [esp],eax
  2d:   e8 fc ff ff ff          call   2e <add_and_multiply+0x14>
  32:   d9 5d fc                fstp   DWORD PTR [ebp-0x4]
  35:   d9 45 fc                fld    DWORD PTR [ebp-0x4]
  38:   d9 05 00 00 00 00       fld    DWORD PTR ds:0x0
  3e:   de c9                   fmulp  st(1),st
  40:   d9 5d fc                fstp   DWORD PTR [ebp-0x4]
  43:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
  46:   89 45 ec                mov    DWORD PTR [ebp-0x14],eax
  49:   d9 45 ec                fld    DWORD PTR [ebp-0x14]
  4c:   c9                      leave
  4d:   c3                      ret

Disassembly of section .bss:

00000000 <nCompletionStatus>:
   0:   00 00                   add    BYTE PTR [eax],al
        ...

Disassembly of section .rodata:

00000000 <.rodata>:
   0:   00 00                   add    BYTE PTR [eax],al
   2:   40                      inc    eax
   3:   40                      inc    eax

Disassembly of section .comment:

00000000 <.comment>:
   0:   00 47 43                add    BYTE PTR [edi+0x43],al
   3:   43                      inc    ebx
   4:   3a 20                   cmp    ah,BYTE PTR [eax]
   6:   28 55 62                sub    BYTE PTR [ebp+0x62],dl
   9:   75 6e                   jne    79 <add_and_multiply+0x5f>
   b:   74 75                   je     82 <add_and_multiply+0x68>
   d:   2f                      das
   e:   4c                      dec    esp
   f:   69 6e 61 72 6f 20 34    imul   ebp,DWORD PTR [esi+0x61],0x34206f72
  16:   2e 36 2e 33 2d 31 75    cs ss xor ebp,DWORD PTR cs:ss:0x75627531
  1d:   62 75
  1f:   6e                      outs   dx,BYTE PTR ds:[esi]
  20:   74 75                   je     97 <add_and_multiply+0x7d>
  22:   35 29 20 34 2e          xor    eax,0x2e342029
  27:   36 2e 33 00             ss xor eax,DWORD PTR cs:ss:[eax]
Disassembly of section .eh_frame:

00000000 <.eh_frame>:
   0:   14 00                   adc    al,0x0
   2:   00 00                   add    BYTE PTR [eax],al
   4:   00 00                   add    BYTE PTR [eax],al
   6:   00 00                   add    BYTE PTR [eax],al
   8:   01 7a 52                add    DWORD PTR [edx+0x52],edi
   b:   00 01                   add    BYTE PTR [ecx],al
```

```
 d:  7c 08                  jl     17 <.eh_frame+0x17>
 f:  01 1b                  add    DWORD PTR [ebx],ebx
11:  0c 04                  or     al,0x4
13:  04 88                  add    al,0x88
15:  01 00                  add    DWORD PTR [eax],eax
17:  00 1c 00               add    BYTE PTR [eax+eax*1],bl
1a:  00 00                  add    BYTE PTR [eax],al
1c:  1c 00                  sbb    al,0x0
1e:  00 00                  add    BYTE PTR [eax],al
20:  00 00                  add    BYTE PTR [eax],al
22:  00 00                  add    BYTE PTR [eax],al
24:  1a 00                  sbb    al,BYTE PTR [eax]
26:  00 00                  add    BYTE PTR [eax],al
28:  00 41 0e               add    BYTE PTR [ecx+0xe],al
2b:  08 85 02 42 0d 05      or     BYTE PTR [ebp+0x50d4202],al
31:  56                     push   esi
32:  c5 0c 04               lds    ecx,FWORD PTR [esp+eax*1]
35:  04 00                  add    al,0x0
37:  00 1c 00               add    BYTE PTR [eax+eax*1],bl
3a:  00 00                  add    BYTE PTR [eax],al
3c:  3c 00                  cmp    al,0x0
3e:  00 00                  add    BYTE PTR [eax],al
40:  1a 00                  sbb    al,BYTE PTR [eax]
42:  00 00                  add    BYTE PTR [eax],al
44:  34 00                  xor    al,0x0
46:  00 00                  add    BYTE PTR [eax],al
48:  00 41 0e               add    BYTE PTR [ecx+0xe],al
4b:  08 85 02 42 0d 05      or     BYTE PTR [ebp+0x50d4202],al
51:  70 c5                  jo     18 <.eh_frame+0x18>
53:  0c 04                  or     al,0x4
55:  04 00                  add    al,0x0
     ...
```

2.3.4　目标文件属性

编译阶段的输出是一个或多个目标文件，因此接下来我们会分析这些目标文件的结构。目标文件的结构中还有很多内容需要介绍，这些内容有助于你对整个编译过程有更深入的了解。

概括来讲：

- 目标文件是通过其对应的源代码翻译得到的。由于一个项目中有许多的源代码文件，因此编译的结果将是一组目标文件。

  当编译完成后，后续的构建过程将基于目标文件进行。
- 符号（symbol）和节（section）是目标文件的基本组成部分，其中符号表示的是程序中的内存地址或数据内存。

  绝大多数的目标文件中都包含代码节（.text）、初始化数据节（.data）、未初始化数据节（.bss）以及一些特殊节（比如调试信息等）。
- 构建程序的目的在于：将编译的每个独立的源代码文件生成的节拼接到一个二进制可执行文件中。

  最终生成的二进制文件中包含了多个相同类型的节（.text、.data 和 .bss 节等），而这些

节是从每个独立的目标文件中拼接得到的。打比方，我们可以把一个目标文件看成一块简单的拼贴，进程内存映射看作一幅巨幅镶嵌画，而生成二进制文件的过程就是将这块拼贴放到镶嵌画中恰当位置的过程。

- 目标文件中独立的节都有可能包含在最终的程序内存映射中，因此目标文件中每个节的起始地址都会被临时设置成 0，等待链接时调整。

  在程序构建过程的后续阶段（链接阶段）中会确定程序内存映射中每个独立节的实际地址范围。
- 在将目标文件的节拼接到程序内存映射的过程中，其中唯一重要的参数是节的长度，准确地说是节的地址范围。
- 目标文件中不包含专门的节会影响堆与栈中的数据。内存映射中的堆与栈内容完全在运行时确定，除了需要指定堆与栈的默认长度以外，并不需要程序指定任何其他初始化设置。
- 目标文件只包含了程序 .bss（未初始化数据）节的基本信息，而 .bss 节本身也仅仅只有字节长度信息。装载器会利用这些有限的数据为 .bss 节建立足够其数据存储的内存空间。

通常来讲，目标文件中的信息是根据一组特定的二进制格式规范集合进行存储的，其中规范定义了多种不同平台的细节信息（Windows 和 Linux、32 位和 64 位、x86 和 ARM 处理器家族）。

二进制格式规范的设计通常是为了支持 C/C++ 语言结构并帮助其解决实现问题。二进制格式规范常常会涵盖各种各样的文件类型，比如可执行文件、静态库和动态库等。

在 Linux 上，可执行和可链接格式（Executable and Linkable Format，ELF）已经得到了普遍运用。而在 Windows 上，二进制文件通常遵循 PE/COFF 格式规范。

2.3.5 编译过程的局限性

通过对程序构建过程逐步分析，我们已经逐渐对整个构建过程有了一个基本的认识。到目前为止，你已经了解了编译过程：将 ASCII 源代码文件翻译成对应二进制目标文件集合。每一个目标文件都会包含多个节，而这些节最终又会成为整个程序内存映射中的一部分，如图 2-3 所示。

现在只需将存储在单个目标文件中的节拼接到程序内存映射中。我们在前面已经提到，该程序构建过程中剩下的阶段就是链接。

如果你观察仔细，可能不禁想问（在了解链接过程以前）：为什么在构建过程中，我们还需要另一个全新的阶段？或者更准确地说，为什么编译过程不能完成目标文件的拼接任务呢？我们有充分的理由去划分构建过程，而本章剩下的内容将对这个决定做出一些合理的解释。

简言之，我们可以用一两句话来解释这个问题。首先，拼接节（特别是代码节）的操作并不总是一件轻松的事情。这是一个很重要的原因，但是这个理由还不够充分，因为有很多

编程语言的构建过程可以用一步完成（换句话说，这些编程序言不需将构建过程划分成两个阶段）。

图 2-3　将独立的节拼接到最终的程序内存映射中

其次，程序构建过程中需要支持代码复用（以及拼接来自不同项目二进制文件的能力），这明确要求在实现 C/C++ 构建时分为两个（编译和链接）阶段。

是什么原因让节的拼接如此复杂

绝大多数情况下，将源代码翻译成目标文件的过程十分简单：将代码行翻译成特定处理平台运行的机器代码指令，或者为初始化变量预留空间并对这些变量初始化，或者为未初始化变量预留空间并用 0 填充等。

但这个过程会导致一些问题：虽然源代码被分成多个源代码文件存放，但要组成一个程序，那么它们之间必然存在一定的关联。事实上这些单独存放的代码一般会通过下面方式之一来建立联系：

- 功能独立的代码之间的函数调用：

  比如，一个聊天应用程序的 GUI 相关代码文件可能会调用 TCP/IP 网络功能源代码文件中的函数，也可能会调用加密功能源代码文件中的函数。

- 外部变量：

  在 C 编程语言的领域中（在 C++ 领域中很少），通常会采取预留全局变量的方式来维护不同部分代码之间的状态。对需要在多个源代码文件中使用的变量来说，我们会在一个源代码文件中将其定义成全局变量，并在其他源代码文件中作为外部变量进行引用。一个典型的例子就是 C 标准库中的 errno 变量，该变量用于保存最后一次出错的值。

为了访问变量或者函数（通常称为符号），就必须知道它们的地址（更准确地说，是程序内存中的函数地址和数据内存中的全局变量地址）。

但是，在单个节组装到程序的节中前（即在拼接节完成前），函数与变量的实际地址是无法确定的。在此之前，我们无法为函数调用及外部变量引用建立有意义的关联，此时函数调用和外部变量称为未解析引用。需要注意的是，当我们引用同一文件中定义的函数或者全局变量时就不会出现这种问题。在这种情况下，函数、函数调用者、变量、变量引用者都在同一个节内，它们的相对偏移在链接之前就已经确定。在拼接完成后，相对内存地址就确定了。

前文曾提到，如果只是为了解决这个问题，我们并不需要将构建过程划分为两个独立的阶段。实际上很多不同语言成功实现了一阶段式构建过程。但是，我们要在程序构建阶段支持复用（此处指的是二进制级别的复用），这才是我们将程序构建划分为两个阶段（编译和链接阶段）的主要原因。

2.4 链接

程序构建过程的第二个阶段就是链接。链接过程的输入是目标文件的集合，其中由编译阶段预先生成。每个目标文件可以被看作单个源代码文件的二进制存储版本，需要为程序内存映射提供各种各样的节（代码、初始化数据、非初始化数据、调试信息等）。链接器的最终任务是将独立的节组合成最终的程序内存映射节，与此同时解析所有的引用。需要注意的是，由于虚拟内存的概念允许链接器将需要填入的程序内存映射假想成从 0 开始的地址范围，而且每个程序的地址范围是相同的，而不需要考虑操作系统运行时到底为进程分配了多少地址范围，因此简化了链接任务。

为了简单起见，我们将会使用最简单的例子，该例子中所有需要拼接的内存映射节都来自同一个项目的文件。在实际情况中，由于代码复用，这种情况并不一定成立。

2.4.1 链接阶段

链接过程包含了一系列阶段（重定位、解析引用），接下来我们就详细介绍这些阶段。

1. 重定位

链接过程的第一个阶段仅仅进行拼接，其过程是将分散在单独目标文件中不同类型的节拼接到程序内存映射节中（见图 2-4）。为了完成该任务，需要将之前预留的空间，也就是节

中从 0 开始的地址范围转换成最终程序内存映射中更具体的地址范围。

图 2-4　重定位：链接的第一阶段

我们使用“更具体”这个词来强调一个事实：由链接器创建的最终程序映像（image）其实依然留下了一些空间。请记住，虚拟内存机制使得每个程序都拥有相同、一致且简单的程序地址空间视图（地址范围为 0 到 2^N），但是程序执行时的实际物理内存地址是由操作系统在运行时决定的，不过这对程序和程序员来说是透明的。

当重定位阶段完成之后，绝大部分（但不是全部）的程序内存映射就已经创建完成了。

2. 解析引用

现在我们来看链接过程中最难的部分。将节的地址范围线性地转换成程序内存映射地址范围，这算得上是一项非常简单的任务。相比来说，更艰巨的任务在于为不同部分的代码间建立关联，使得程序成为一个整体。

假定（由于演示程序非常简单，理应如此）之前所有的构建阶段（完成编译和节的重定位）都已经成功完成。现在来看一下在最后的链接阶段中仍需解决的问题。

前文曾提到，引发链接问题的根本原因其实很简单：代码片段在不同的编译单元中（源代码文件），它们之间尝试相互引用，但在将目标文件拼接成程序内存映射之前，又无从知晓引用对象的内存地址。这部分代码会引发问题往往是因为引用了其他的程序内存（函数入口点）或者数据内存（全局数据、静态数据、外部数据）中的地址。

在该代码示例中，你会遇到以下几种情况：

- dd_and_multiply 函数调用 add 函数，这两个函数在同一个源代码文件中（即同一个目标文件的同一个编译单元）。这种情况下，函数 add 的内存映射地址是一个已知量，会被扩展成其相对于 function.o 中代码节起始地址的相对偏移。
- main 函数会调用 add_and_multiply 函数，并同时引用外部变量 nCompletionStatus，这个时候问题就出现了——我们不知道它们的实际程序内存地址。实际上，编译器会假定这些符号未来会在进程内存映射中存在。但是，直到生成完整内存映射之前，这两项引用会一直被当成未解析引用。

该问题可以用图 2-5 来描述。

图 2-5 未解析引用问题的基本形式

为了解决这类问题，我们需要在链接阶段就对这些引用进行解析。此时链接器需要：

- 检查拼接到程序内存映射中的节。
- 找出哪些部分代码产生了外部调用。
- 计算该引用的精确地址（在内存映射中的地址）。
- 最后，将机器指令中的伪地址替换成程序内存映射的实际地址，这样就完成了引用的解析。

当链接器完成其工作之后，情况如图 2-6 所示。

3. 演示项目：链接示例

我们可以通过两种方法来编译和链接演示项目，用于创建可执行文件。

如果采用按部就班的方式，你首先需要编译这两个源代码文件，生成目标文件；然后将

两个目标文件链接输出可执行文件。

图 2-6　已解析引用

```
$ gcc -c function.c main.c
$ gcc function.o main.o -o demoApp
```

如果想一步到位，你可以使用一条命令调用编译器和链接器来完成相同的工作。

```
$ gcc function.c main.c -o demoApp
```

为了便于演示，我们采用按部就班的方法进行编译，这样将会生成目标文件 main.o，我们会针对其中的一些重要内容进行详细讲解。

反汇编 main.o 文件：

```
$ objdump -D -M intel main.o
```

该命令的输出中包含了未解析的引用。

main.o 文件的反汇编输出结果（Intel 汇编格式）

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明:

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。