

Modern CMake for C++

Discover a better approach to building, testing,
and packaging your software

Rafał Świdziński



Modern CMake for C++

Discover a better approach to building, testing, and packaging your software

构建、测试和打包软件

作者: Rafał Świdziński

译者: 陈晓伟

本书概述

创建一流的软件非常困难，开发人员很难确定哪些建议是最新的，哪些方法已经可以用更简单、更好的实践所取代。与此同时，大多数在线资源提供的解释有限，也缺乏相应的上下文。

本书提供了一种更简单、更全面的体验，介绍了如何构建 C++ 解决方案。Modern CMake for C++ 是一个端到端的任务自动化指南，包括构建、测试和打包。不仅可以了解如何在项目中使用 CMake 语言，还可以了解如何使它们可维护，优雅和干净。本书还关注源目录、构建目标和包的结构。随着了解的深入，将学习如何编译和链接可执行文件和库，这些过程如何工作，以及如何优化 CMake 中的构建得最佳结果。还将了解如何在项目中使用外部依赖项——第三方库、测试框架、程序分析工具和文档生成器。最后，导出内部和外部目标，以及安装和打包。

读完这本书，就能够自信地使用 CMake 了。

关键特性

- 理解并自动化 CMake 编译和链接
- 管理内部和外部依赖关系
- 添加质量检查和测试作为构建步骤

作者简介

Rafał Świdziński 在 Google 公司担任工程师，具有超过 10 年专业经验的全栈开发人员，了解大量的编程语言和技术，一直在自己的公司和包括 Cisco Meraki、Amazon 和 Ericsson 在内的公司开发软件。他来自波兰的罗兹 (Łódź)，现在生活在英国伦敦，在那里经营一个 YouTube 频道 “Smok”，讨论与软件开发相关的话题。他很喜欢处理技术问题，包括该领域的挑战。在工作中，他了解各种技术概念，并揭开了软件工程师角色背后的的艺术和科学的神秘面纱。他的主要关注代码质量和编程技巧。

本书相关

- Github 地址：
<https://github.com/xiaoweiChen/Modern-CMake-For-CXX>

前言

创造优秀的软件并非易事。开发人员经常会遇到无法确定哪些建议是最新的，以及哪些方法已经有更新或更好的实践的问题。同时，大多数资源在没有交代背景、上下文和结构的情况下，解释起来也很难。

本书提供了全面构建 C++ 的端到端解决方案，提供了更简单的方法。了解如何使用 CMake，如何使其可维护、优雅和干净。将使用自动化的方式帮助您完成项目中出现的许多复杂任务，比如构建、测试和打包。

本书会指导您如何生成源目录，以及如何构建目标和包。随着学习的深入，将了解如何编译和链接可执行文件和库，以及如何优化各种步骤以获得最佳结果。还将了解如何向项目添加外部依赖项：第三方库、测试框架、程序分析工具和文档生成器。最后，导出内部和外部目标，以及安装和打包的解决方案。

阅读完本书后，相信您就能够在非常专业地使用 CMake 了。

适宜读者

这本书是为具有 C/C++ 编程知识的工程师和软件开发人员所著，从而可以学习 CMake，以了解自动化构建小型和大型软件的解决方案。若刚开始使用 CMake，并长期使用 GNU Make，或者只是想复习一下最新的最佳实践，那么本书也非常适合您。

本书内容

第 1 章，初识 CMake，如何安装和使用 CMake 命令行，以及如何创建相应的项目文件。

第 2 章，CMake 语法，介绍了关键的代码信息：注释、命令调用和参数、变量、列表和控制结构。

第 3 章，CMake 项目，介绍了项目的基本配置、所需的 CMake 版本、项目元数据、文件结构，以及如何设置工具链。

第 4 章，使用目标，为可执行程序和库生成构建目标。

第 5 章，CMake 编译 C++，编译过程的工作方式，以及如何在 CMake 项目中进行控制。

第 6 章，进行链接，关于链接、静态和动态库。解释了如何构建一个项目，以便对其进行测试。

第 7 章，管理依赖关系，现代 CMake 中管理依赖关系的方法。

第 8 章，测试框架，如何将测试框架添加到项目中，以及如何使用 CMake 提供的 CTest。

第 9 章，分析工具，如何在项目中自动格式化，以及如何进行静态和动态分析。

第 10 章，生成文档，如何使用 Doxygen 根据 C++ 源代码生成手册。

第 11 章，安装和打包，展示如何准备将项目用于其他项目或安装到系统上，还有如何使用 CPack。

第 12 章，创建完整的项目，如何将了解的所有知识整合到一个完整的项目中。

附录：其它指令，提供了相应指令的快速引用：string()、list()、file() 和 math()。

编译环境

阅读本书的读者需要对 C++ 和类 Unix 系统有基本的了解。虽然这不是一个严格的要求，但有了这些基础，能够更加的理解本书给出的例子。

本书的目标是 CMake 3.20，但是描述的大多数技术应该从 CMake 3.15 后出现的（后面添加的特性通常会突出显示）。所有示例都在 Debian 上进行了测试，并安装了以下软件包：

```
clang-format clang-tidy cppcheck doxygen g++ gawk git graphviz lcov  
libpqxx-dev libprotobuf-dev make pkg-config protobuf-compiler tree  
valgrind vim wget
```

这里，建议使用 Docker 镜像。

如果您正在使用这本书的数字版本，建议您自己输入代码或从本书的 **GitHub** 存储库访问代码（下一节有链接）。这样做将帮助您避免与复制和粘贴代码相关的错误。

下载示例

可以从 GitHub 上下载这本书的示例代码文件 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp>。若代码有更新，会在 GitHub 存储库中更新。

还可以从丰富的书籍和视频目录中获得其他代码包<https://github.com/PacktPublishing/>。

下载彩图

我们还提供了一个 PDF 文件，其中有本书中使用的屏幕截图和图表的彩图。下载地址：https://static.packt-cdn.com/downloads/9781801070058_ColorImages.pdf。

目录

第一部分：基础知识	12
第 1 章 初识 CMake	13
1.1. 相关准备	13
1.2. 基础知识	14
1.2.1 CMake?	14
1.2.2 如何工作?	15
1.3. 不同平台的安装	18
1.3.1 Docker	18
1.3.2 Windows	19
1.3.3 Linux	19
1.3.4 macOS	20
1.3.5 使用源码构建	20
1.4. 使用命令行	21
1.4.1 CMake	21
1.4.2 CTest	32
1.4.3 CPack	32
1.4.4 CMake 用户界面	32
1.4.5 CCMake	33
1.5. 项目文件	34
1.5.1 源码树	34
1.5.2 构建树	35
1.5.3 文件列表	35
1.5.4 CMakeLists.txt	35
1.5.5 CMakeCache.txt	36
1.5.6 包配置文件	37
1.5.7 cmake_install.cmake, CTestTestfile.cmake 和 CPackConfig.cmake	37
1.5.8 CMakePresets.json 和 CMakeUserPresets.json	37
1.5.9 设置 Git 忽略文件	40
1.6. 脚本和模块	41
1.6.1 脚本	41
1.6.2 实用工具模块	42

1.6.3 查找模块	42
1.7. 总结	43
1.8. 扩展阅读	43
第 2 章 CMake 语法	44
2.1. 相关准备	44
2.2. 基本语法	44
2.2.1 注释	45
2.2.2 执行指令	46
2.2.3 指令参数	47
2.3. 变量	50
2.3.1 引用变量	51
2.3.2 环境变量	52
2.3.3 缓存变量	53
2.3.4 如何正确使用变量作用域	54
2.4. 列表	56
2.5. 控制结构	57
2.5.1 条件块	57
2.5.2 条件指令的语法	58
2.5.3 循环	60
2.5.4 定义指令	62
2.6. 实用指令	66
2.6.1 message() 指令	66
2.6.2 include() 指令	68
2.6.3 include_guard() 指令	68
2.6.4 file() 指令	68
2.6.5 execute_process() 指令	69
2.7. 总结	69
2.8. 扩展阅读	70
第 3 章 CMake 项目	71
3.1. 相关准备	71
3.2. 指令和命令	72
3.2.1 指定最低的 CMake 版本——cmake_minimum_required()	72
3.2.2 定义语言和元数据—project()	72
3.3. 划分项目	73
3.3.1 作用域的子目录	75
3.3.2 嵌套项目	76
3.3.3 外部项目	77
3.4. 项目结构	77
3.5. 环境范围	81

3.5.1 识别操作系统	81
3.5.2 交叉编译——主机系统和目标系统?	82
3.5.3 简化变量	82
3.5.4 主机系统信息	82
3.5.5 平台是 32 位还是 64 位架构?	84
3.5.6 系统的端序	84
3.6. 配置工具链	84
3.6.1 设定 C++ 标准	84
3.6.2 坚持支持标准	85
3.6.3 特定于供应商的扩展	85
3.6.4 过程间优化	86
3.6.5 检查支持的编译器特性	86
3.6.6 编译测试文件	86
3.7. 禁用内构建	88
3.8. 总结	89
3.9. 扩展阅读	89
第二部分：进行构建	90
第 4 章 使用目标	91
4.1. 相关准备	91
4.2. 目标的概念	91
4.2.1 依赖图	93
4.2.2 可视化的依赖性	94
4.2.3 目标属性	95
4.2.4 可传递需求	96
4.2.5 处理冲突的传播属性	98
4.2.6 实现伪目标	99
4.2.7 构建目标	101
4.3. 编写自定义命令	101
4.3.1 使用自定义命令作为生成器	102
4.3.2 使用自定义命令作为目标钩子	103
4.4. 生成器表达式	104
4.4.1 一般语法	105
4.4.2 计算类型	106
4.4.3 可以尝试的例子	113
4.5. 总结	116
4.5.1 扩展阅读	117
第 5 章 CMake 编译 C++	118
5.1. 相关准备	118
5.2. 编译基础	118

5.2.1 编译工作	119
5.2.2 初始配置	120
5.2.3 管理目标源	121
5.3. 预处理配置	122
5.3.1 提供包含文件的路径	122
5.3.2 预处理宏定义	122
5.3.3 配置头文件	125
5.4. 配置优化器	126
5.4.1 优化级别	127
5.4.2 函数内联	128
5.4.3 循环展开	129
5.4.4 循环向量化	130
5.5. 编译过程	130
5.5.1 减少编译时间	130
5.5.2 查找错误	134
5.6. 总结	138
5.6.1 扩展阅读	138
第 6 章 进行链接	140
6.1. 相关准备	140
6.2. 掌握正确的链接方式	141
6.3. 构建不同类型的库	144
6.3.1 静态库	144
6.3.2 动态库	144
6.3.3 模块库	145
6.3.4 位置无关的代码	145
6.4. 用定义规则解决问题	146
6.4.1 动态链接的重复符号	148
6.4.2 使用命名空间——不要依赖链接器	150
6.5. 连接顺序和未定义符号	150
6.6. 分离 main() 进行测试	152
6.7. 总结	153
6.8. 扩展阅读	154
第 7 章 管理依赖关系	155
7.1. 相关准备	155
7.2. 如何查找已安装的软件包	155
7.3. 使用 FindPkgConfig	160
7.4. 编写自己的查找模块	162
7.5. 使用 Git 库	166
7.5.1 通过 Git 子模块提供外部库	166

7.5.2 不使用 Git 的项目克隆依赖项	169
7.6. 使用 ExternalProject 和 FetchContent 模块	170
7.6.1 ExternalProject	170
7.6.2 FetchContent	175
7.7. 总结	178
7.8. 扩展阅读	178
第三部分：自动化	180
第 8 章 测试框架	181
8.1. 相关准备	181
8.2. 为什么自动化测试值得这么麻烦?	181
8.3. 使用 CTest 来标准化 CMake 中的测试	182
8.3.1 构建和测试模式	183
8.3.2 测试模式	184
8.4. 为 CTest 创建最基本的单元测试	188
8.4.1 为测试构建项目	192
8.5. 单元测试框架	195
8.5.1 Catch2	196
8.5.2 GTest	198
8.5.3 GMock	200
8.6. 生成测试覆盖报告	205
8.6.1 避免 SEGFAULT 问题	209
8.7. 总结	209
8.8. 扩展阅读	210
第 9 章 分析工具	211
9.1. 相关准备	211
9.2. 格式化	211
9.3. 静态检查	215
9.3.1 Clang-Tidy	217
9.3.2 Cpplint	218
9.3.3 Cppcheck	218
9.3.4 include-what-you-use	218
9.3.5 link-what-you-use	218
9.4. Valgrind 的动态分析	219
9.4.1 Memcheck	219
9.4.2 Memcheck-Cover	223
9.5. 总结	225
9.6. 扩展阅读	225
第 10 章 生成文档	227
10.1. 相关准备	227

10.2. 添加 Doxygen	227
10.3. 生成好看的文档	232
10.4. 总结	234
10.5. 扩展阅读	234
10.5.1 其他文档生成工具	234
第 11 章 安装和打包	235
11.1. 相关准备	235
11.2. 只导出, 不安装	235
11.3. 在系统上安装	238
11.3.1 安装逻辑目标	239
11.3.2 底层安装	242
11.3.3 安装期间使用脚本	248
11.4. 创建可重用的包	249
11.4.1 理解可重定位目标	250
11.4.2 安装目标导出文件	251
11.4.3 编写配置文件	252
11.4.4 创建高级配置文件	254
11.4.5 生成包版本文件	257
11.5. 定义组件	259
11.5.1 在 <code>find_package()</code> 中使用组件	259
11.5.2 在 <code>install()</code> 指令中使用组件	259
11.6. 使用 CPack 打包	261
11.7. 总结	264
11.8. 扩展阅读	264
第 12 章 创建完整的项目	266
12.1. 相关准备	266
12.2. 规划工作	266
12.3. 项目布局	269
12.3.1 对象库	269
12.3.2 比较动态库与静态库	270
12.3.3 目录结构	270
12.4. 构建和管理依赖项	272
12.4.1 构建 Calc 库	273
12.4.2 构建 Calc 终端可执行文件	274
12.5. 测试和程序分析	278
12.5.1 准备覆盖率模块	280
12.5.2 准备 Memcheck 模块	281
12.5.3 应用测试场景	282
12.5.4 添加静态分析工具	284

12.6. 安装和打包	286
12.6.1 库的安装	286
12.6.2 可执行文件的安装	288
12.6.3 使用 CPack 打包	288
12.7. 提供文档	289
12.7.1 自动生成文档	289
12.7.2 非技术性文件	290
12.8. 总结	293
12.9. 扩展阅读	293
附录: 其它指令	295
string() 指令	295
list() 指令	298
file() 指令	299
math() 指令	301

第一部分：基础知识

了解基础知识，对于理解高级功能和避免错误至关重要。这就是大多数 CMake 用户遇到麻烦的原因：不了解基础，所以很难得到正确的结果。通常，人们很容易跳过介绍性的内容，直接进入真正的内容，以便快速完成任务。本节中，将通过解释 CMake 的核心主题，并通过组合几行代码来展示最简单的项目是什么样子，来解决这些问题。

为了建立一个合适的思考环境，将解释 CMake 到底是什么，如何工作的，以及命令行是什么样的。会讨论不同的构建阶段，并学习用于生成构建系统的语言。还将讨论 CMake 项目：包含哪些文件，如何处理目录结构，以及主要配置。

- 第 1 章，启用 CMake
- 第 2 章，CMake 的最佳使用方式
- 第 3 章，创建 CMake 项目

第 1 章 初识 CMake

将源码转换为可工作应用会比较神奇。不仅是效果本身(即设计并赋予生命的工作机制),而且是将理念付诸于过程的行为本身。

作为开发者的工作流程基本是:设计、编码和测试。我们使用编译器能理解的语言来编写代码,并检查写出来的东西是否如预期的那样工作。为了从源码中创建高质量的应用,需要耐心地重复流程,并检查容易出错的任务:使用正确的命令、检查语法、链接二进制文件、运行测试、产生报告问题等。

每一次的进步都不容易,但我们更想把重点放在编码上,并将其他所有事情委托给自动化工具。理想情况下,这个过程应该从一个简单地按钮开始。在更改了代码之后,工具将以智能的、快速的、可扩展的方式,在不同的操作系统和环境中以相同的方式工作。支持多个集成开发环境(IDE)和持续集成(CI)流水,这些流水在更改提交到代码库后,就能对代码进行测试。

CMake 是许多此类需求一种答案,并且要正确配置和使用需要一定的工作量。这并不是因为 CMake 复杂,而是因为需求很复杂。请放心,我们会有条不紊地进行学习。完全了解这个流程后,您将成为一个软件构建大师。

可能有的读者会急于开始编写自己的 CMake 项目,我对您的态度表示赞赏。由于您的项目将主要面向用户(包括您自己),因此理解如何使用同样重要。

那么先从成为 CMake 的用户开始吧。先介绍一些基础知识:这个工具是什么,如何工作,如何安装。然后,将深入研究命令行和操作模式。最后,将总结项目中不同文件的用途,并了解如何在不创建项目的情况下使用 CMake。

本章中,我们将讨论以下主题:

- 基础知识
- 不同平台的安装
- 使用命令行
- 项目文件
- 脚本和模块

1.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter01>。

构建本书中提供的示例,推荐的命令:

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 <build tree> 和 <source tree>。注意:构建树是目标/输出目录的路径,源代码树是源码所在的路径。

1.2. 基础知识

C++ 源码的编译过程很简单:

```
1 // chapter-01/01-hello/hello.cpp
2
3 #include <iostream>
4 int main() {
5     std::cout << "Hello World!" << std::endl;
6     return 0;
7 }
```

现在，要获得可执行文件，只需要运行一个命令。用文件名作为参数使用编译器:

```
$ g++ hello.cpp -o a.out
```

代码正确，因此编译器将生成一个机器可以理解的二进制文件。可以通过文件名来运行程序:

```
$ ./a.out
Hello World!
$
```

然而，随着项目的增长，无法将所有内容都保存在一个文件中。务实的代码实践建议，文件应该较小，并采用组织良好的结构，并且手动编译每个文件是一个令人厌烦和脆弱的过程。

1.2.1 CMake?

假设通过编写一个遍历项目树，并编译所有内容的脚本来自动化构建。为了避免不必要的编译，脚本将检测源码(脚本)是否修改过。现在，我们希望有一种方法来管理传递给每个文件编译器的参数——最好是可配置的。此外，脚本应该知道如何链接已编译的文件，或者构建整个解决方案，以便在更大的项目中重用或作为模块。

添加的特性越多，就越有可能得到一个成熟的解决方案。构建软件是一个通用的过程(其中有一些步骤可以跳过):

- 编译可执行程序和库
- 管理依赖关系
- 测试
- 安装
- 打包
- 生成文档
- 测试更多功能

要做出一个真正模块化的、功能强大的C++构建应用来满足各种需求很难，但CMake确实做到了。Kitware的Bill Hoffman在20多年前实现了CMake的第一个版本，现在有了更多的功能，并得到了社区的支持。目前，CMake的开发很活跃，并已成为C和C++开发人员的行业标准。

以自动化的方式构建代码的问题比 CMake 出现的要早得多，所以会有很多选择:Make、Autotools、SCons、Ninja、Premake 等。但为什么 CMake 可以来居上呢？

关于 CMake，有几件事我觉得（主观地说）很重要：

- 专注于支持现代编译器和工具链。
- CMake 是真正的跨平台——支持 Windows、Linux、macOS 和 Cygwin 的构建。
- 为主流 IDE 生成项目文件:Microsoft Visual Studio, Xcode 和 Eclipse CDT。此外，也是其他项目的模型，如 CLion。
- CMake 操作在合适的抽象级别上——允许将文件分组到可重用的目标和项目中。
- 有很多用 CMake 构建的项目，其提供了一种简单的方法将它们包含到自己的项目中。
- CMake 将测试、打包和安装视为构建过程的固有组成。
- 弃用旧的、未使用的特性，从而保持 CMake 的精简。

CMake 提供了统一的、流线型的体验。不管是在 IDE 中构建，还是直接从命令行构建，还照顾到构建后阶段。即使前面所有的环境都不同，持续集成/持续部署 (CI/CD) 流水也可以轻松地使用相同的 CMake 配置，并使用单一标准构建项目。

1.2.2 如何工作？

可能有这样的印象：CMake 是在一端读取源代码，在另一端生成二进制文件的工具——虽然原则上没问题，但这并不是全部。

CMake 自己不能构建任何东西——它依赖于系统中的其他工具来执行实际的编译、链接和其他任务。可以将它看作是构建过程的协调器，知道需要完成哪些步骤，最终目标是什么，以及如何为工作找到合适的工人和材料。

这个过程有三个阶段：

- 配置
- 生成
- 构建

配置阶段

这个阶段会读取存储在源码树目录中的项目信息，并为生成阶段准备输出目录或构建树。CMake 首先创建一个空的构建树，并收集工作环境的信息，例如：架构、编译器、链接器和存档打包器。此外，还要检查编译器是否能编译简单的测试程序。

接下来，执行 CMakeLists.txt 项目配置文件（CMake 项目是用 CMake 的编码语言配置的）。这个文件是 CMake 项目的基础部分（源文件可以稍后添加），它告诉 CMake 有关项目结构、目标和依赖项（库和其他 CMake 包）的信息。此过程中，CMake 将收集的信息存储在构建树中，例如：系统详细信息、项目配置、日志和临时文件，这些信息将用于下一阶段，会创建一个 CMakeCache.txt 文件来存储变量（例如编译器和其他工具的路径），节省下次配置期间的时间开销。

生成阶段

读取项目配置之后，CMake 将为当前工作环境生成一个构建系统。构建系统只是其他构建工具的简化配置文件(例如，GNU Make 或 Ninja 的 Makefiles 和 Visual Studio 的 IDE 项目文件)，CMake 仍然可以通过生成器表达式对构建配置应用进行一些修改。

Note

生成阶段在配置阶段之后自动执行，在提到构建系统的“配置”或“生成”时，本书和其他资源经常提到这两个阶段。要显式地分段运行配置阶段，可以使用 `cmake-gui`。

构建阶段

为了生成项目中指定的工件，必须运行适当的构建工具。可以通过 IDE 直接调用，也可以使用 CMake 命令行。这些构建工具都会执行构建步骤，使用编译器、链接器、静态和动态分析工具、测试框架、报告工具，以及其他工具来生成目标。

这个解决方案的美妙之处在于，能够用单一配置(即相同的项目文件)为每个平台按需生成构建系统：

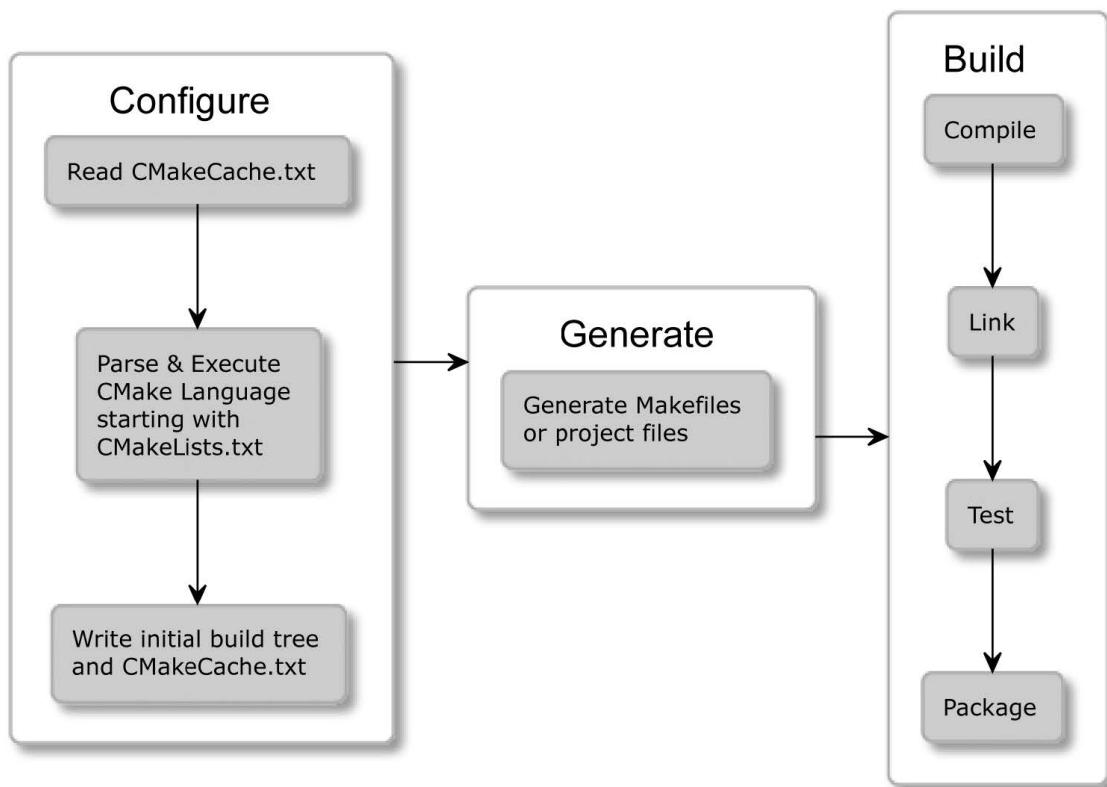


图 1.1 CMake 的各个阶段

还记得 `hello.cpp` 吗？CMake 可以很容易的对其进行构建。所需要的就是下面的 `CMakeLists.txt` 文件和两个简单的命令，`cmake -B buildtree` 和 `cmake --build buildtree`：

```
# chapter01/01-hello/CMakeLists.txt: Hello world in the CMake language

cmake_minimum_required(VERSION 3.20)
project(Hello)
```

```
add_executable(Hello hello.cpp)
```

下面是 Dockerized Linux 系统的输出:

```
root@5f81fe44c9bd:/root/examples/chapter01/01-hello# cmake  
-B buildtree.  
-- The C compiler identification is GNU 9.3.0  
-- The CXX compiler identification is GNU 9.3.0  
-- Check for working C compiler: /usr/bin/cc  
-- Check for working C compiler: /usr/bin/cc -- works  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Check for working CXX compiler: /usr/bin/c++  
-- Check for working CXX compiler: /usr/bin/c++ -- works  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /root/examples/  
chapter01/01-hello/buildtree  
root@5f81fe44c9bd:/root/examples/chapter01/01-hello# cmake  
--build buildtree/  
Scanning dependencies of target Hello  
[ 50%] Building CXX object CMakeFiles>Hello.dir/hello.cpp.o  
[100%] Linking CXX executable Hello  
[100%] Built target Hello
```

运行:

```
root@68c249f65ce2:~# ./buildtree/Hello  
Hello World!
```

我们生成了一个在 `buildtree` 目录中的构建系统。执行构建阶段，并生成能够运行的最终二进制文件。

现在了解了最终结果的样子，我相信你会充满疑问: 这个过程的先决条件是什么? 这些命令是什么意思? 为什么我们需要两个? 如何编写自己的项目文件?——这些问题将在后续的章节中回答。

寻求帮助

本书将提供与 CMake 当前版本相关的最重要的信息(撰写本文时是 3.20)。为了提供最好的建议，不会出现不推荐的功能。推荐至少使用 3.15 版本，从这个版本起才算是“现代的 CMake”。若需要了解更多信息，可以参考官方在线文档：<https://cmake.org/cmake/help/>。

1.3. 不同平台的安装

CMake 是用 C++ 编写的跨平台开源软件，可以自行编译它，但不推荐这样做。可以从官方网页下载预编译的二进制文件，网址是<https://cmake.org/download/>。

基于 Unix 的系统可以直接使用命令行进行安装。

Note

CMake 并不附带编译器。若系统中还没有安装编译器，需要在使用 CMake 之前安装或提供它们的路径。确保将可执行文件的路径添加到 PATH 环境变量中，以便 CMake 能够找到它们。

为了避免在阅读本书时遇到的工具和依赖问题，建议使用 Docker。

来了解一下可以使用 CMake 的环境。

1.3.1 Docker

Docker (<https://www.docker.com/>) 是一个跨平台工具，提供操作系统级虚拟化，允许应用程序以完整(称为容器)的形式发布。这些是捆绑包，包含软件及其运行所需的所有库、依赖项和工具。Docker 可在隔离的轻量级环境中运行容器。

这个概念使得共享工具链非常方便，这些工具链是必需的，并且是配置好的。

Docker 平台有一个镜像的公共存储库<https://registry.hub.docker.com/>，提供了数百万个现成的镜像。

方便起见，我发布了两个 Docker 镜像：

- `swidzinski/cmake:toolchain`: 包含了使用 CMake 构建所必需的策划工具和依赖项。
- `swidzinski/cmake:examples`: 包含了前面的工具链，以及本书中的所有项目和示例。

第一个选项是为那些只想要一个干净镜像的读者；第二个选项可以在阅读本章时，使用示例进行实践。

可以按照 Docker 官方文档的说明安装 Docker(请参考docs.docker.com/get-docker)。然后，在终端中执行以下命令下载镜像并启动容器：

```
$ docker pull swidzinski/cmake:examples
$ docker run -it swidzinski/cmake:examples
root@b55e271a85b2:root@b55e271a85b2:#
```

注意，所有的例子都在这样格式的目录中：`/root/examples/examples/chapter-<n>/<m>-<title>`。

1.3.2 Windows

Windows 上安装很简单——只需下载 32 位或 64 位的版本，可以为 Windows 安装程序选择便携 ZIP 或 MSI 包。

对于 ZIP 包，必须将 CMake 的 bin 目录添加到 PATH 环境变量中，这样就可以在目录中使用 CMake：

```
'cmake' is not recognized as an internal or external command,  
operable program or batch file.
```

若更喜欢简单的方式，就使用 MSI 安装：

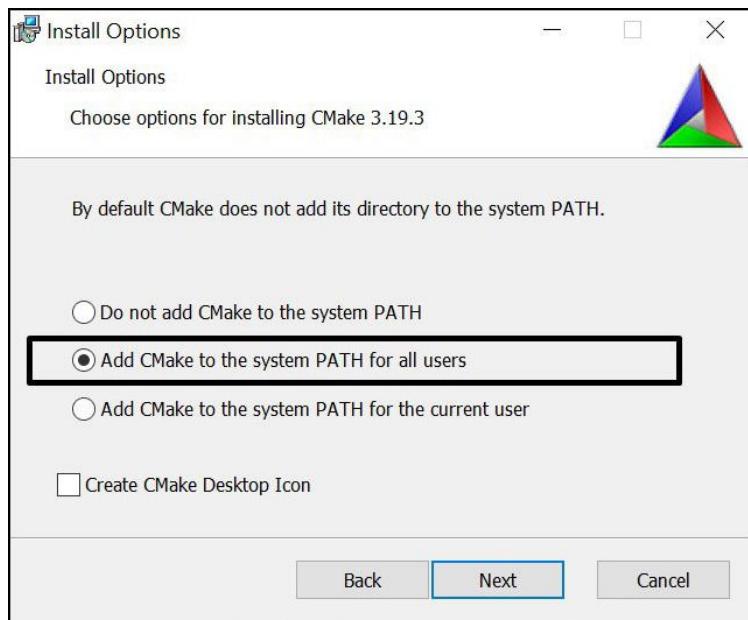


图 1.2 安装向导可以设置 PATH 环境变量

这是一个开源软件，所以可以自己构建 CMake。首先，必须在系统上获得 CMake 的二进制副本。那么，若有自己的构建工具，为什么还要使用其他构建工具呢？CMake 的代码贡献者可以使用此方式生成新版本的 CMake。

在 Windows 上，还需要一个构建工具来完成由 CMake 启动的构建过程。这里的选择是 Visual Studio，社区版可以从微软的网站上免费获得：<https://visualstudio.microsoft.com/downloads/>。

1.3.3 Linux

Linux 上获得 CMake 与获得其他包一样，只需在终端使用包管理器即可。软件包的版本不保证是最新的，若想要最新版本，可以下载安装脚本：

Linux x86_64 的安装脚本

```
$ wget -O - https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0-linux-x86_64.sh | bash
```

Linux aarch64 的安装脚本

```
$ wget -O - https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0-Linux-aarch64.sh | bash
```

Debian/Ubuntu 的安装包

```
$ sudo apt-get install cmake
```

Red Hat 的安装包

```
$ yum install cmake
```

1.3.4 macOS

这个平台也得到了 CMake 开发人员的大力支持。可以通过 MacPorts 安装:

```
$ sudo port install cmake
```

或者，使用 Homebrew:

```
$ brew install cmake
```

1.3.5 使用源码构建

若以上方法都失败了——或者是在特殊的平台上——可以从官方网站下载源代码，自行编译:

```
$ wget https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0.tar.gz  
$ tar xzf cmake-3.20.0.tar.gz  
$ cd cmake-3.20.0  
$ ./bootstrap  
$ make  
$ make install
```

从源码构建相对较慢，需要更多步骤。但通过这种方式，可以保证使用最新版本的 CMake。与 Linux 可用的包相比，这一点尤其明显：系统版本越老，得到的更新就越少。

我们已经安装了 CMake，现在来了解如何使用它吧！

1.4. 使用命令行

本书的大部分内容将展示如何为用户准备 CMake 项目，要彻底了解用户在不同场景下如何与 CMake 交互。将允许测试项目文件，并确保它们工作正常。

CMake 是一个工具集，由五个可执行程序组成：

- `cmake`: 配置、生成和构建项目的主要可执行文件。
- `ctest`: 用于运行和报告测试结果的测试驱动程序。
- `cpack`: 用来生成安装程序和源包的打包程序。
- `cmake-gui`: `cmake` 的图形界面。
- `ccmake`: `cmake` 基于控制台的图形界面。

1.4.1 CMake

提供了一些操作模式（也称为动作）：

- 生成项目构建系统
- 构建项目
- 安装项目
- 运行脚本
- 运行命令行工具
- 获得帮助

生成项目构建系统

这是构建项目的第一步，关于如何执行 CMake 构建操作的选项：

```
# 生成模式的语法

cmake [<options>] -S <path-to-source> -B <path-to-build>
cmake [<options>] <path-to-source>
cmake [<options>] <path-to-existing-build>
```

我们将在接下来的部分中讨论这些选项。现在，让我们集中精力选择正确的命令形式。CMake 的重要特性是支持源外构建或在单独目录中生成工件。与 GNU Make 等工具相比，这确保源目录与构建相关的文件保持分离，并避免不必要的文件污染版本控制系统 (VCS) 或忽略指令。这就是为什么最好使用生成模式的第一种形式的命令：用-S 选项指定源树的路径，然后用-B 选项指定生成的构建系统的目录：

```
cmake -S ./project -B ./build
```

上面的命令将从./project 目录中的源代码中在./build 目录中生成一个构建系统（如果没有就创建）。

可以跳过其中一个参数，cmake 将“猜测”我们打算使用当前目录。跳过这两者将形成内源代码构建，这很麻烦。

不推荐

不要使用 cmake <directory> 命令的第二种或第三种形式，因为它会产生一个混乱的内源代码构建（我们将在第 3 章中学习如何阻止它）。正如语法片段中所示，若 <directory> 中已经存在先前的构建，则相同的命令的行为不同，将使用到源的缓存路径并从那里重新构建。由于经常使用终端命令历史中运行相同的命令，因此在这里可能会遇到麻烦。使用此方式之前，请检查 shell 当前是否在正确的目录下。

示例

在当前目录中构建，但从一个目录中获取源代码 (-S 是可选的)：

```
cmake -S ..
```

在./build 目录中编译，并使用当前目录中的源代码：

```
cmake -B build
```

生成器的选项

可以在生成阶段指定一些选项。选择和配置生成器决定了将使用哪个构建工具进行构建，构建文件是什么样子的，以及构建树的结构。

CMake 确实支持许多平台上的多个原生构建系统，除非你同时安装了几个，否则 CMake 会正确地选择。选项可以使用 `CMAKE_GENERATOR` 环境变量覆盖，或者直接在命令行上指定生成器：

```
cmake -G <generator-name> <path-to-source>
```

一些生成器（如 Visual Studio）支持更深入的工具集（编译器）和平台（编译器或 SDK）规范。此外，各自有覆盖默认值的环境变量：`CMAKE_GENERATOR_TOOLSET` 和 `CMAKE_GENERATOR_PLATFORM`。也可以直接指定：

```
cmake -G <generator-name>
      -T <toolset-spec> -A <platform-name>
      <path-to-source>
```

Windows 用户通常希望为他们喜欢的 IDE 生成一个构建系统。在 Linux 和 macOS 上，使用 Unix Makefiles 或 Ninja 生成器比较常见。

要检查系统上有哪些生成器可用，使用以下命令：

```
cmake --help
```

帮助信息输出的最后，应该可以看到一个完整的列表：

```
# Windows 10上有很多可用的生成器

The following generators are available on this platform:
Visual Studio 16 2019
Visual Studio 15 2017 [arch]
Visual Studio 14 2015 [arch]
Visual Studio 12 2013 [arch]
Visual Studio 11 2012 [arch]
Visual Studio 10 2010 [arch]
Visual Studio 9 2008 [arch]
Borland Makefiles
NMake Makefiles
NMake Makefiles JOM
```

```
MSYS Makefiles
MinGW Makefiles
Green Hills MULTI
Unix Makefiles
Ninja
Ninja Multi-Config
Watcom Wmake
CodeBlocks - MinGW Makefiles
CodeBlocks - NMake Makefiles
CodeBlocks - NMake Makefiles JOM
CodeBlocks - Ninja
CodeBlocks - Unix Makefiles
CodeLite - MinGW Makefiles
CodeLite - NMake Makefiles
CodeLite - Ninja
CodeLite - Unix Makefiles
Eclipse CDT4 - NMake Makefiles
Eclipse CDT4 - MinGW Makefiles
Eclipse CDT4 - Ninja
Eclipse CDT4 - Unix Makefiles
Kate - MinGW Makefiles
Kate - NMake Makefiles
Kate - Ninja
Kate - Unix Makefiles
Sublime Text 2 - MinGW Makefiles
Sublime Text 2 - NMake Makefiles
Sublime Text 2 - Ninja
Sublime Text 2 - Unix Makefiles
```

缓存选项

CMake 在配置阶段向系统查询各种信息，这些信息缓存在构建树目录中的 `CMakeCache.txt` 中。可以想到的第一件事是预填充缓存信息：

```
cmake -C <initial-cache-script> <path-to-source>
```

可以提供到 CMake 脚本的路径，该脚本 (仅) 包含 `set()` 指令的列表，以指定将用于初始化空构建树的变量。

现有缓存变量的初始化和修改可以用另一种方式完成(例如, 创建文件时只设置几个变量有点多)。可以简单地在命令行中设置它们:

```
cmake -D <var>[:<type>]=<value> <path-to-source>
```

:<type> 部分是可选的(用在 GUI 端), 可以使用 BOOL、FILEPATH、PATH、STRING 或 INTERNAL。若省略了类型, 将设置为一个已经存在的变量的类型; 否则, 将设置为 UNINITIALIZED。

一个特别重要的变量包含构建的类型: 例如, Debug 和 Release。许多 CMake 项目将在许多场景下运行, 以决定符号信息的冗长程度、调试信息的存在程度, 以及所创建工件的优化级别。

对于单配置生成器(如 Make 和 Ninja), 需要在配置阶段使用 CMAKE_BUILD_TYPE 变量指定构建类型, 并为每种类型的配置生成单独的构建树:Debug、Release、MinSizeRel 或 RelWithDebInfo。

这有一个例子:

```
cmake -S . -B build -D CMAKE_BUILD_TYPE=Release
```

注意, 多配置生成器是在构建阶段配置的。

可以用-L 选项列出缓存变量:

```
cmake -L[A] [H] <path-to-source>
```

这样的列表将包含未标记为 ADVANCED 的缓存变量, 可以通过参数 A 进行改变。要打印带有变量的帮助消息——需要添加 H 参数。

使用-D 选项手动添加的自定义变量将不可见, 除非指定受支持的类型之一。

删除一个或多个变量可以通过以下选项完成:

```
cmake -U <globbing_expr> <path-to-source>
```

这里, 通配符表达式支持 * 通配符和任何? 字符, 使用时需要倍加小心。

-U 和-D 选项都可以重复多次使用。

用于调试和跟踪的选项

CMake 可以通过许多选项来运行, 这些选项可以让您了解内部发生了什么。要获取关于变量、命令、宏和其他设置的一般信息, 运行以下命令:

```
cmake --system-information [file]
```

可选的 file 参数允许将输出存储在文件中。在构建树目录中运行, 将从日志文件中打印有关缓存变量和构建消息的信息。

项目中，将使用 `message()` 报告构建过程的细节，CMake 根据当前日志级别（默认情况下是 `STATUS`）过滤这些日志的输出。下面的命令行指定了感兴趣的日志级别：

```
cmake --log-level=<level>
```

这里的级别可以是其中一个：`ERROR`、`WARNING`、`NOTICE`、`STATUS`、`VERBOSE`、`DEBUG` 或 `TRACE`。可以在 `CMAKE_MESSAGE_LOG_LEVEL` 缓存变量中永久地保留这个设置。

另一个有趣的选项允许显示每个 `message()` 的日志上下文。要调试非常复杂的项目，可以像使用堆栈一样使用 `CMAKE_MESSAGE_CONTEXT` 变量。当代码进入特定上下文时，可以向堆栈添加描述性名称，并在离开时删除它。通过这样做，消息将用当前的 `CMAKE_MESSAGE_CONTEXT` 变量装饰：

```
[some.context.example] Debug message.
```

启用这种日志输出的选项如下：

```
cmake --log-context <path-to-source>
```

我们将在第 2 章中更详细地讨论。

若所有这些都失败了——需要使用大型武器——跟踪模式。这将打印每个命令的文件名和调用的确切行号及其参数。可以通过以下方式启用：

```
cmake --trace
```

预置的选项

用户可以指定许多选项来从您的项目生成构建树。在处理构建树路径、生成器、缓存和环境变量时，很容易混淆或遗漏某些内容。开发人员可以简化用户与项目交互的方式，并提供 `CMakePresets.json` 文件，其指定了一些默认值。

要列出所有可用的预设，可以执行以下命令：

```
cmake --list-presets
```

使用预设的方式如下所示：

```
cmake --preset=<preset>
```

这些值覆盖系统默认值和环境。与此同时，也可以显式地在命令行上传递的参数所覆盖：

Variable Precedence

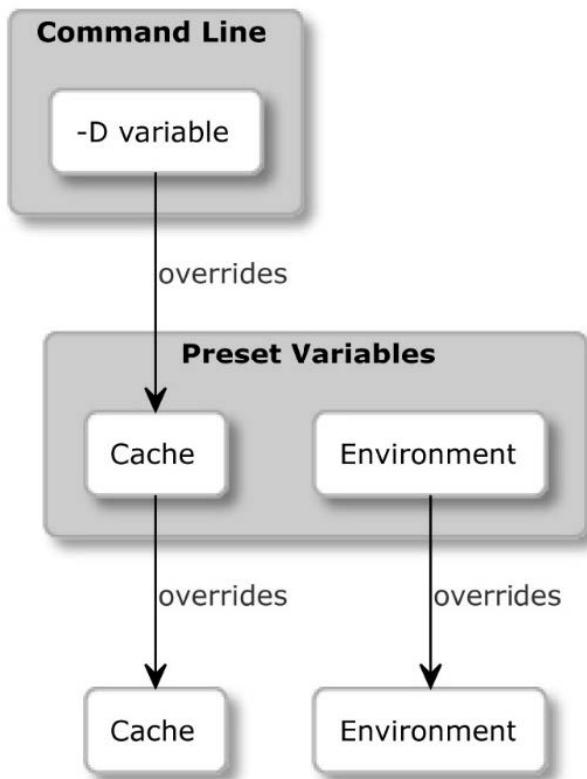


图 1.3 预设如何覆盖 CMakeCache.txt 和系统环境变量

构建项目

生成构建树后，就可以进入下一个阶段：运行构建器工具。CMake 不仅知道如何为许多不同的构建器生成输入文件，还可以使用特定于项目参数运行。

不推荐

许多在线资源建议在生成阶段之后直接运行 GNU Make: `make`。这是 Linux 和 macOS 的默认生成器，可以正常工作。但更推荐本节中描述的方法，与生成器无关，并且在所有平台上都得到支持。因此，不需要担心应用程序的每个用户的确切环境。

```
# 构建模式的语法
```

```
cmake --build <dir> [<options>] [-- <build-tool-options>]
```

大多数情况下，只要提供最基本的内容就可以获得成功的构建：

```
cmake --build <dir>
```

CMake 需要知道构建树在哪里，与在生成阶段使用 `-B` 参数传递的路径相同。

通过提供一些选项，CMake 允许指定适用于每个构建器的关键构建参数。若需要为所选的本机构建器提供特殊参数，请在命令末尾--后传递：

```
cmake --build <dir> -- <build-tool-options>
```

并行构建的选项

默认情况下，许多构建工具将使用多个并发进程并行编译源代码。构建者知道项目的依赖关系，因此可以同时处理满足依赖关系的步骤，从而节省用户的时间。

若是在功能强大的机器上构建(或强制使用单线程构建进行调试)，则可能需要覆盖该设置。只需使用以下选项之一，并指定作业数量：

```
cmake --build <dir> --parallel [<number-of-jobs>]  
cmake --build <dir> -j [<number-of-jobs>]
```

另一种方法是使用 CMAKE_BUILD_PARALLEL_LEVEL 环境变量来设置。和往常一样，可以使用前面的选项来覆盖变量。

目标的选项

我们将在本书的第二部分讨论目标。现在，假设每个项目都是由一个或多个目标组成。通常情况下，我们想要构建所有目标；有时，可能需要跳过某些目标或显式地构建一个(排除在正常构建之外)的目标感兴趣。可以这样做：

```
cmake --build <dir> --target <target1> -t <target2> ...
```

可以通过重复-t 参数来指定多个目标。

一个通常不建造的目标是 clean。这个目标将删除构建目录中的所有构件。可以这样使用：

```
cmake --build <dir> -t clean
```

此外，若想先清理，然后实现一个正常的构建，CMake 提供了一个方便的方法：

```
cmake --build <dir> --clean-first
```

多配置生成器的选项

我们已经对生成器有了一些了解。其中一些提供了比其他更多的特性，这个特性就是在单个构建树中构建 Debug 和 Release 构建类型的能力。

支持该功能的生成器包括 Ninja Multi-Config、Xcode 和 Visual Studio。其他生成器都是单配置生成器，为此需要单独的构建树。

选择 Debug, Release, MinSizeRel 或 RelWithDebInfo，并进行指定：

```
cmake --build <dir> --config <cfg>
```

否则，CMake 将使用 Debug 作为默认值。

调试选项

当出现问题时，应该做的第一件事是检查输出消息。经验丰富的开发人员知道，总是打印所有的细节信息不是明智之举，所以通常默认隐藏它们。当需要查看内部情况时，可以让 CMake 的信息更加详细：

```
cmake --build <dir> --verbose  
cmake --build <dir> -v
```

同样的效果可以通过设置 CMAKE_VERBOSE_MAKEFILE 缓存变量来实现。

安装项目

构建工件后，用户可以将它们安装到系统中。这会将文件复制到正确的目录中，安装库，或者从 CMake 脚本运行一些自定义安装逻辑。

```
# 安装模式的语法  
  
cmake --install <dir> [<options>]
```

与其他操作模式一样，CMake 需要一个到生成构建树的路径：

```
cmake --install <dir>
```

多配置生成器的选项

就像在构建阶段一样，可以指定希望安装使用哪种构建类型（有关更多细节，请参阅构建项目部分）。可用的类型包括 Debug、Release、MinSizeRel 和 RelWithDebInfo：

```
cmake --install <dir> --config <cfg>
```

组件的选项

作为开发人员，可能会选择将项目拆分为可以独立安装的组件。我们将在第 11 章中进一步详细讨论组件的概念。现在，假设它们代表解的不同部分。这可能是应用程序、文档和工具之类的东西。

安装单个组件时，可以使用以下方式：

```
cmake --install <dir> --component <comp>
```

权限的选项

若安装是在类 Unix 平台上进行的，可以为安装目录指定默认权限，使用以下选项，格式为 u=rwx,g=rx,o=rx：

```
cmake --install <dir>
--default-directory-permissions <permissions>
```

安装目录的选项

可以在项目配置中指定的安装路径前面，加上已经选择的前缀（当我们对某些目录有限制的写访问权限时）。以“/home/user”为前缀的“/usr/local”路径变成“/home/user/usr/local”：

```
cmake --install <dir> --prefix <prefix>
```

注意，这在 Windows 上不起作用，因为该平台上的路径通常以驱动器号开始。

调试的选项

类似地，对于构建阶段，也可以选择查看安装阶段的详细输出。可以使用以下任何一种方法：

```
cmake --build <dir> --verbose
cmake --build <dir> -v
```

若设置了 VERBOSE 环境变量，也可以达到同样的效果。

运行脚本

使用 CMake 的自定义语言配置 CMake 项目。为什么不用于其他任务呢？当然，可以编写独立的脚本（将在本章的最后讨论）。

CMake 可以像这样运行脚本：

```
# 脚本模式的语法

cmake [ {-D <var>=<value>}... ] -P <cmake-script-file>
[-- <unparsed-options>...]
```

运行脚本不会运行任何配置或生成阶段。此外，不会影响缓存。有两种方法可以将值传递给这个脚本：

- 通过使用-D 选项定义的变量。
- 通过可在--后传递的参数。CMake 将为传递给脚本的所有参数(包括--) 创建 CMake_ARGV<n> 变量。

运行命令行工具

极少数情况下，可能需要以与平台无关的方式运行单个命令——可能是复制一个文件或计算一个校验和。并不是所有的平台都一样，所以并不是所有的命令都可以在每个系统中使用，或者它们有不同的名称。

CMake 提供了一种模式，可以跨平台以相同的方式执行一些常见的方法：

```
# 命令行工具模式的语法

cmake -E <command> [<options>]
```

由于这种特定模式的使用是相当有限的，不会对其深入讨论。若对细节感兴趣，建议使用 cmake -E 列出所有可用的命令。为了简单地了解一下所提供的功能，CMake 3.20 支持以下命令：

capabilities, cat, chdir, compare_files, copy, copy_directory, copy_if_different, echo, echo_append, env, environment, make_directory, md5sum, shalsum, sha224sum, sha256sum, sha384sum, sha512sum, remove, remove_directory, rename, rm, server, sleep, tar, time, touch, touch_nocreate, create_symlink, create_hardlink, true 和 false。

若缺少想要使用的命令，或者需要更复杂的行为，可以考虑将其包装到脚本中并以-P 模式运行。

获得帮助

CMake 通过命令行提供帮助信息。

```
# 帮助模式的语法

cmake --help[-<topic>]
```

1.4.2 CTest

为了产生和维护高质量的代码，自动化测试非常重要。这就是为什么花了一整个章节来讨论这个主题（请参考第 8 章），在那里我们将深入研究 CTest 的用法。它是可用的命令行工具之一，所以先简单的介绍一下。

CTest 是将 CMake 包装在一个更高的抽象层中，构建阶段只是开发软件过程中的一个垫脚石。CMake 可以为我们完成的其他任务包括更新、运行各种测试、向外部仪表板报告项目状态，以及运行用 CMake 语言编写的脚本。

更重要的是，CTest 标准化了使用 CMake 构建的解决方案的运行测试和报告，所以用户不需要知道项目正在使用哪个测试框架或如何运行。CTest 提供了一个方便的方式来列出、筛选、打乱、重试和时间框测试运行。此外，若需要构建，也可以为调用 CMake。

为已构建项目运行测试的最简单方法，就是在生成构建树中使用 `ctest`:

```
$ ctest
Test project C:/Users/rapha/Desktop/CMake/build
Guessing configuration Debug
Start 1: SystemInformationNew
1/1 Test #1: SystemInformationNew ..... Passed 3.19 sec

100% tests passed, 0 tests failed out of 1
Total Test time (real) = 3.24 sec
```

1.4.3 CPack

构建并测试了软件之后，我们就可以与世界进行分享。高级用户完全可以接受源代码，这就是他们想要的。然而，由于方便和节省时间，世界上绝大多数人都在使用预编译的二进制文件。

CMake 不会把你困在这里，构建 CPack 的确切目的是为不同的平台创建包：压缩包、可执行安装程序、安装向导、NuGet 包、macOS 包、DMG 包、rpm 等。

CPack 的工作方式与 CMake 相似：使用 CMake 语言进行配置，并有许多包生成器可供选择（只是不要将它们与 CMake 构建系统生成器混淆）。我们将在第 11 章中详细介绍所有的具体细节，因为这是一个非常强大的工具，用于 CMake 项目的最后阶段。

1.4.4 CMake 用户界面

Windows 的 CMake 提供了一个 GUI 版本，用于配置之前准备好的项目的构建过程。对于类 Unix 平台，有一个使用 QT 库构建的版本。Ubuntu 在 `cmake-qt-gui` 包中提供了它。

要使用 CMake 用户界面，需要运行 `cmake-gui` 可执行文件：

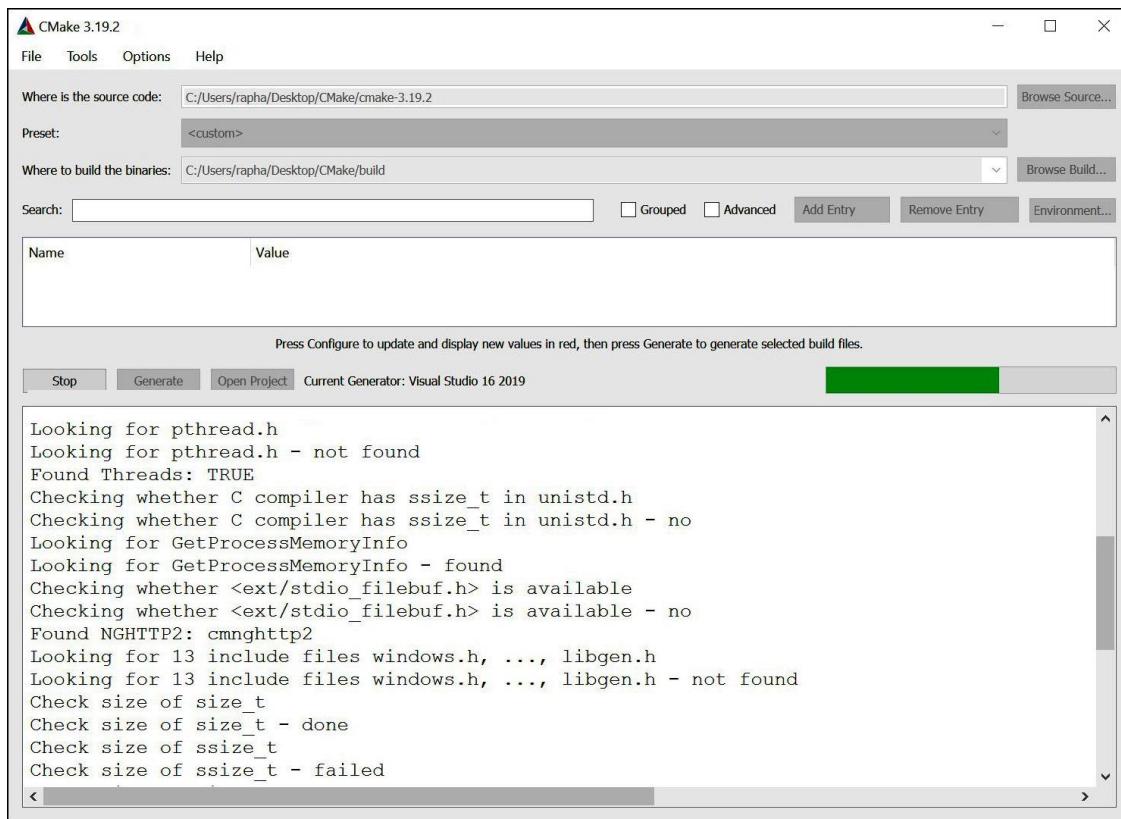


图 1.4 CMake GUI —— 使用 Visual Studio 2019 生成器构建系统的配置阶段

GUI 应用程序对于用户来说是一个方便的工具，因为其中的选项相当有限。对于那些不熟悉命令行而更喜欢基于窗口的界面的人来说，它可能很有用。

不推荐

我会向终端用户推荐 GUI，作为一名程序员，避免引入任何手册，阻止每次构建程序时都需要单击表单的步骤。这对于 CI 流水中的构建自动化尤其重要。这些工具需要无头软件，这样构建就可以在没有任何用户交互的情况下执行。

1.4.5 CCMake

ccmake 可执行文件是 CMake 面向类 Unix 平台的接口（它不适用于 Windows）。它不是 CMake 包的一部分，所以用户必须单独安装。

Debian/Ubuntu 系统的命令如下：

```
$ sudo apt-get install cmake-curses-gui
```

注意，可以通过此 GUI 以交互方式指定项目配置设置。当程序运行时，终端底部会给出简短的说明：

```
# CCMake命令的语法

ccmake [<options>]
ccmake {<path-to-source> | <path-to-existing-build>}
```

CCMake 使用与 cmake 具有相同的选项集:

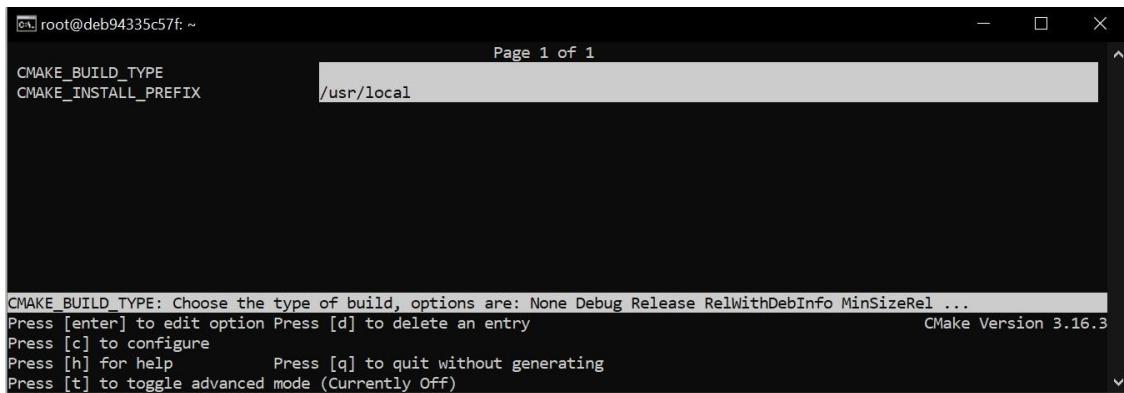


图 1.5 ccmake 的配置阶段

与图形用户界面 (GUI) 一样，这种模式具有相当的局限性，仅供经验不足的用户使用。

以上就是对 CMake 命令行的基本介绍。接下来，来了解一下 CMake 项目的结构。

1.5. 项目文件

CMake 使用文件来管理项目。修改文件内容之前，尝试对每个文件的作用有一个大致的了解，即使文件包含 CMake 语言命令，也不能确定它是为开发人员编辑而设计的。生成一些文件供后续工具使用，对这些文件的修改都将在某个阶段重写。高级用户根据自己的需要，使用其他文件对项目进行调整。最后，还有一些临时文件在特定的上下文中可以提供有价值的信息。本节还将说明它们中的哪些不应该位于版本控制系统中。

1.5.1 源码树

这是项目所在的目录(也称为项目根目录)，包含了所有的 C++ 源码和 CMake 项目文件。

以下是这个目录的几个关键点:

- 需要在其顶部目录中提供一个 CMakeLists.txt 配置文件。
- 使用 VCS(例如 git) 来管理它。
- 该目录的路径可有使用 cmake 命令的 -S 参数给定。
- 不要在 CMake 代码中硬编码源树的绝对路径——软件的用户可以将项目存储在不同的路径下。

1.5.2 构建树

CMake 使用这个目录来存储构建过程中生成的所有内容: 项目的构件、配置、缓存、构建日志, 以及本地构建工具将创建的文件。此目录的可选名称包括构建树。

以下是这个目录的几个关键点:

- 二进制文件将在这里创建, 例如可执行文件和库, 以及用于最终链接的目标文件和存档。
- 不要将这个目录添加到 VCS——其特定于系统。
- CMake 推荐源外构建或在与所有源文件分离的目录中生成工件的构建, 可以避免使用临时的、系统特定的文件(或源内构建)污染项目的源码树。
- 可由-B 指定, 若提供了到源码的路径, 则作为 cmake 命令的最后一个参数, 例如, `cmake -S .. /project ./`。
- 建议项目包含安装阶段, 可以将最终工件放在系统中正确的位置, 这样所有用于构建的临时文件都可以删除。

1.5.3 文件列表

包含 CMake 语言的文件称为文件列表, 可以通过使用 `include()` 和 `find_package()`, 或间接使用 `add_subdirectory()` 来将其包含在另一个文件中:

- CMake 不会强制对这些文件进行一致的命名, 通常的扩展名为.cmake。
- 主 CMakeLists.txt 的文件非常重要, 在配置阶段执行的第一个文件, 需要放在源树的顶层。
- 当 CMake 遍历源码树, 并包含不同的文件列表时, 内置了以下变量:CMAKE_CURRENT_LIST_DIR、CMAKE_CURRENT_LIST_FILE、CMAKE_PARENT_LIST_FILE 和 CMAKE_CURRENT_LIST_LINE。

1.5.4 CMakeLists.txt

CMake 项目配置为 CMakeLists.txt 列表文件, 需要在源码树根部提供。这样的顶层文件是在配置阶段首先执行的, 至少要包含两个命令:

- `cmake_minimum_required(VERSION <x.xx>)`: 设置 CMake 的期望版本(并隐式地告诉 CMake 要对遗留行为使用哪些策略)。
- `project(<name> <OPTIONS>)`: 这用于命名项目(提供的名称将存储在 PROJECT_NAME 中)并指定配置它的选项(将在第 2 章中进一步讨论这个问题)。

随着软件的发展, 可能希望将其划分为更小的单元, 以便分别进行配置和推理。CMake 通过子目录和其自身的 CMakeLists.txt 文件来完成划分, 项目结构可能如下所示:

```
CMakeLists.txt  
api/CMakeLists.txt  
api/api.h  
api/api.cpp
```

可以使用简单的 CMakeLists.txt 文件将它们组合在一起:

```
cmake_minimum_required(VERSION 3.20)
project(app)
message("Top level CMakeLists.txt")
add_subdirectory(api)
```

项目的主要方面包含在顶层文件中: 管理依赖项、声明需求和检测环境。文件中, 还有一个 add_subdirectory(api) 指令来包含另一个 CMakeListsts.txt。执行特定于应用 API 部分的指令。

1.5.5 CMakeCache.txt

当配置阶段第一次运行时, 缓存变量将从列表文件中生成并存储在 CMakeCache.txt 中。该文件位于构建树的根目录中:

```
# This is the CMakeCache file.
# For build in directory:
c:/Users/rapha/Desktop/CMake/empty_project/build
# It was generated by CMake: C:/Program
Files/CMake/bin/cmake.exe
# You can edit this file to change values found and used by
cmake.
# If you do want to change a value, simply edit, save, and
exit the editor.
# The syntax for the file is as follows:
# KEY:TYPE=VALUE
# KEY is the name of a variable in the cache.
# TYPE is a hint to GUIs for the type of VALUE, DO NOT EDIT
TYPE!.
# VALUE is the current value for the KEY.
#####
# EXTERNAL cache entries
#####
//Flags used by the CXX compiler during DEBUG builds.
CMAKE_CXX_FLAGS_DEBUG:STRING=/MDd /Zi /Ob0 /Od /RTC1
// ... more variables here ...
#####
# INTERNAL cache entries
#####
//Minor version of cmake used to create the current loaded
cache
CMAKE_CACHE_MINOR_VERSION:INTERNAL=19
```

```
// ... more variables here ...
```

从注释中观察到的，EXTERNAL 部分中的缓存项是供用户修改的，而 INTERNAL 部分是由 CMake 管理的。注意，不建议手动更改它们。

以下是需要记住的几个要点：

- 可以通过调用 `cmake`(请参阅命令行部分中的缓存选项) 或通过 `ccmake/cmake-gui` 管理该文件。
- 可以通过删除该文件将项目重置为默认配置，将从列表文件中重新生成。
- 缓存变量可以从列表文件中读写。有时候，变量参考的评估有点复杂，将在第 2 章中更详细地讨论它。

1.5.6 包配置文件

CMake 生态系统包括项目可以依赖的外部包，允许开发人员以无缝的、跨平台的方式使用库和工具。支持 CMake 的包应该提供一个配置文件，以便 CMake 理解如何使用它们。

我们将在第 11 章中学习如何编写这些文件。同时，有一些知识点需要了解：

- 配置文件包含关于如何使用库二进制文件、头文件和辅助工具的信息。有时，它们会公开 CMake 宏，以便在项目中使用。
- 使用 `find_package()` 指令来包含包。
- 描述包的 CMake 文件命名为 `<PackageName>-config.cmake` 和 `<PackageName>Config.cmake`。
- 使用包时，可以指定需要的包的版本。CMake 将检查相关的 `<Config>Version.cmake` 文件。
- 配置文件由支持 CMake 生态系统的包供应商提供。若供应商不提供这样的配置文件，可以用查找模块替换。
- CMake 提供了包注册表，用于在系统范围内为每个用户存储包。

1.5.7 `cmake_install.cmake`, `CTestTestfile.cmake` 和 `CPackConfig.cmake`

这些文件是由生成阶段的 `cmake` 可执行文件在构建树中生成的，不应该通过手动编辑。CMake 使用它们作为 CMake 安装操作、CTest 和 CPack 的配置。若正在实现一个内源代码构建(不推荐)，则将它们添加到 VCS 的忽略文件中可能是个好主意。

1.5.8 `CMakePresets.json` 和 `CMakeUserPresets.json`

当需要明确缓存变量、所选择的生成器、构建树的路径等时，项目的配置可能会成为一项相对繁忙的任务——特别是当有多种构建项目的方法时。这就处于预设的用武之地了。

用户可以通过 GUI 选择预置，也可以使用命令行`--listpresets`，并使用`--preset=` 选项为构建系统选择预置。

预设值以相同的 JSON 格式存储在两个文件中：

- `CMakePresets.json`: 项目作者提供的预设。
- `CMakeUserPresets.json`: 根据自己的偏好定制项目配置用户准备的(可以将其添加到 VCS 的忽略文件中)。

预设是项目文件，在项目中并不是必需的，只有完成初始设置时，预设才会有用。所以，若不需要的话，可以跳到下一节：

```
# chapter-01/02-presets/CMakePresets.json
{
  "version": 1,
  "cmakeMinimumRequired": {
    "major": 3, "minor": 19, "patch": 3
  },
  "configurePresets": [ ],
  "vendor": {
    "vendor-one.com/ExampleIDE/1.0": {
      "buildQuickly": false
    }
  }
}
```

CMakePresets.json 指定以下根字段：

- Version: 这是必需的，总是 1。
- cmakeMinimumRequired: 可选的。以带有三个字段的散列形式指定 CMake 版本:major、minor 和 patch。
- vendor: IDE 可以使用这个可选字段来存储它的元数据，其是一个以供应商域和斜杠分隔的路径为键的映射。CMake 会忽略了这个字段。
- configurePresets: 一个可选的可用预设数组。

在 configurePresets 数组中添加两个预设值：

```
# chapter-01/02-presets/CMakePresets.json : my-preset
{
  "name": "my-preset",
  "displayName": "Custom Preset",
  "description": "Custom build - Ninja",
  "generator": "Ninja",
  "binaryDir": "${sourceDir}/build/ninja",
  "cacheVariables": {
    "FIRST_CACHE_VARIABLE": {
      "type": "BOOL", "value": "OFF"
    },
    "SECOND_CACHE_VARIABLE": "Ninjas rock"
  },
  "environment": {
    "MY_ENVIRONMENT_VARIABLE": "Test",
```

```

    "PATH": "$env{HOME}/ninja/bin:$env{PATH}"
},
"vendor": {
    "vendor-one.com/ExampleIDE/1.0": {
        "buildQuickly": true
    }
}
},

```

该文件支持树状结构，其中子预设从多个父预设继承属性。可以创建前面预设的副本，并且只覆盖我们需要的字段。下面是一个子预设的例子：

```

# chapter-01/02-presets/CMakePresets.json : my-preset-multi
{
    "name": "my-preset-multi",
    "inherits": "my-preset",
    "displayName": "Custom Ninja Multi-Config",
    "description": "Custom build - Ninja Multi",
    "generator": "Ninja Multi-Config"
}

```

Note

CMake 文档只标记了一些明确需要的字段，还有一些其他的字段标记为可选的，这些字段必须在预设中提供，或者从其父字段继承。

预设定义为具有以下字段的映射：

- **name:** 标识预设的必需字符串，必须是机器友好，并在两个文件中具有唯一名称。
- **Hidden:** 一个可选的布尔值，从 GUI 和命令行列表中隐藏预设值。这样的预设可以是另一个的父级，只需要提供名称。
- **displayName:** 一个可选的字符串，提供人类友好的名称。
- **description:** 一个描述预设的可选字符串。
- **Inherits:** 这是要继承的预置名称的可选字符串或数组。在冲突的情况下，来自之前预设的值将是首选，并且每个预设都可以覆盖继承的字段。此外，CMakeUserPresets.json 可以从项目预设继承，但不能从项目预设继承。
- **Vendor:** 这是特定于供应商的值，遵循与供应商字段相同的约定(可选)。
- **Generator:** 一个必需的或继承的字符串，用于指定要用于预设的生成器。
- **architecture** 和 **toolset:** 这些是用于配置支持这些选项的生成器的可选字段(在生成项目构建系统部分中提到)。每个字段可以只是一个字符串或带有值和策略字段的哈希，其中策略是设置的或外部的。配置为 set 的 strategy 字段将设置该值，若生成器不支持该字段，则会产生错误。配置为“外部”时，是为外部 IDE 设置了字段值，CMake 应该忽略它。

- `binaryDir`: 一个必需的或继承的字符串，提供到构建树目录的路径(该目录是绝对的或相对于源树的)，支持宏展开。
- `cacheVariables`: 一个可选的缓存变量映射，其中键表示变量名。可接受的值包括 `null`、”`TRUE`”、”`FALSE`”、字符串值或带有可选类型字段和必选值字段的哈希。`value` 可以是 “`TRUE`” 或 “`FALSE`” 的字符串值。缓存变量是通过联合操作继承的，除非将值指定为 `null`，否则将保持未设置。字符串值支持宏展开。
- `Environment`: 一个可选的环境变量映射，其中键表示变量名。接受的值包括空值或字符串值。除非将环境变量的值指定为 `null`，否则将使用联合操作继承环境变量。字符串值支持宏展开，只要没有循环引用，变量可以以任何顺序相互引用。

识别和展开下列宏：

- `${sourceDir}`: 源码树的路径。
- `${sourceParentDir}`: 源树的父目录的路径。
- `${sourceDirName}`: `${sourceDir}` 的最后一个文件名组件。
例如，对于`/home/rafal/project`，其将是 `project`。
- `${presetName}`: 预设名称字段的值。
- `${generator}`: 预设的生成器字段的值。
- `${dollar}`: 一个美元符号 (`$`)。
- `${env}{<variable-name>}`: 一个环境变量宏。若定义了，将从预设值返回变量的值；否则，将从父环境返回值。预设中的变量名是区分大小写的(与 Windows 环境不同)。
- `${penv}{<variable-name>}`: 这个选项类似于 `${env}`，但总是返回来自父环境的值。这可以解决在预设环境变量中不允许使用循环引用的问题。
- `${vendor}{<macro-name>}`: 供应商自己的宏。

1.5.9 设置 Git 忽略文件

目前有许多版本控制工具，其中最流行的一种是 Git。每当开始一个新项目时，最好确保只签入需要存在的存储库文件。若只是向`.gitignore` 文件中添加生成的、用户的或临时文件，那么保持项目健康就会更容易，Git 知道在构建新的提交时会自动跳过它们。以下是我使用的`.gitignore` 文件：

```
# chapter-01/01-hello/.gitignore

# If you put build tree in the source tree add it like so:
build_debug/
build_release/

# Generated and user files
**/CMakeCache.txt
**/CMakeUserPresets.json
**/CTestTestfile.cmake
```

```
**/CPackConfig.cmake  
**/cmake_install.cmake  
**/install_manifest.txt  
**/compile_commands.json
```

项目中使用上述文件将为您和其他贡献者和用户提供更大的灵活性。

项目文件的未知领域现在已经逐渐显露。有了这个映射，就以编写自己的列表文件、配置缓存、准备预置等等。开始项目编写之前，来看看还可以用 CMake 创建哪些其他类型的自包含单元。

1.6. 脚本和模块

CMake 的工作主要集中在构建的项目和生成生产上，例如 CI/CD 流水和测试平台，或者部署到机器或工件存储库。但 CMake 的另外两个概念使使用者可以使用其语言进行创建和复用：脚本和模块。

1.6.1 脚本

为了配置项目构建，CMake 提供了一种平台无关的编程语言。这附带了许多有用的命令。可以使用此工具编写项目附带的，或完全独立的脚本。

可以把它看作是进行跨平台工作的一致方式：可以使用一个版本，而不是在 Linux 上使用 bash 脚本，在 Windows 上使用批处理或 PowerShell 脚本。也可以引入外部工具，如 Python、Perl 或 Ruby 脚本，但这种依赖将增加 C/C++ 项目的复杂性。有时这是唯一可以完成工作的方法，但通常情况下，可以用更简单的方法来规避。

我们已经知道，可以使用-P 选项执行脚本:-P script.cmake。但是所提供的脚本文件的需求是什么？没那么复杂：脚本可以非常复杂，也可以是一个空文件。但建议在脚本的开头使用 cmake_minimum_required() 指令，从而告诉 CMake 哪些策略应该应用到这个项目的后续指令中（更多细节见第 3 章）。

```
# chapter-01/03-script/script.cmake

# An example of a script
cmake_minimum_required(VERSION 3.20.0)
message("Hello world")
file(WRITE Hello.txt "I am writing to a file")
```

当运行脚本时，CMake 不会执行任何通常的阶段（比如配置或生成），也不会使用缓存。由于在脚本中没有源代码/构建树的概念，通常持有这些路径引用的变量将包含当前工作目录：CMAKE_BINARY_DIR, CMAKE_SOURCE_DIR, CMAKE_CURRENT_BINARY_DIR 和 CMAKE_CURRENT_SOURCE_DIR。

1.6.2 实用工具模块

CMake 项目可以使用外部模块来增强它们的功能。模块是用 CMake 语言编写的，包含宏定义、变量和执行各种功能的命令。范围从相当复杂的脚本 (CPack 和 CTest 也提供模块!) 到相当简单的脚本，如 AddFileDependencies 或 TestBigEndian。

CMake 发行版包含了近 90 个不同的实用程序模块，还可以通过浏览列表从网上下载更多，比如在<https://github.com/onqtam/awesome-cmake>上找到的列表，或者自己编写一个模块。

要使用实用程序模块，需要使用 `include(<MODULE>)` 指令。下面是一个简单的项目：

```
# chapter-01/04-module/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)

project(ModuleExample)
include(TestBigEndian)

TEST_BIG_ENDIAN(IS_BIG_ENDIAN)

if(IS_BIG_ENDIAN)
  message("BIG_ENDIAN")
else()
  message("LITTLE_ENDIAN")
endif()
```

我们将了解哪些模块是可用的，因为其与当前的主题相关。若对其他模块感兴趣，可以在<https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html>找到一个完整的绑定模块列表。

1.6.3 查找模块

包的配置文件，提到了 CMake 有一种机制，可以查找属于不支持 CMake 和不提供 CMake 配置文件 (或没有提供) 的外部依赖项的文件。这就是查找模块的作用。CMake 提供了 150 多个模块，这些模块能够定位系统中不同的包。就像实用模块一样，线上有很多查找模块，另一种选择是编写自己的模块。

可以通过使用 `find_package()` 指令，并提供相关包的名称来使用它们。这样的查找模块将玩一个捉迷藏的小游戏，并检查它正在寻找的软件的所有已知位置，然后定义变量 (如模块手册中指定的那样)，可以根据该依赖项进行构建。

例如，FindCURL 模块搜索一个流行的客户端 URL 库，并定义以下变量: `CURL_FOUND`、`CURL_INCLUDE_DIRS`、`CURL_LIBRARIES` 和 `CURL_VERSION_STRING`。

我们将在第 7 章中更深入地讨论查找模块。

1.7. 总结

现在了解了 CMake 是什么，以及它是如何工作的；学习了 CMake 工具系列的关键组件，以及如何在各种系统上安装它们。像一个真正的高级用户一样，知道通过命令行运行 CMake 的所有方法：构建系统生成、构建项目、安装、运行脚本、命令行工具和打印帮助。了解 CTest、CPack 和 GUI 应用程序。然后，以正确的视角为用户和其他开发人员创建项目。此外，还了解了项目的组成部分：目录、列表文件、配置、预置和帮助文件，以及在 VCS 中应该忽略的内容。最后，还了解了其他非项目文件：脚本和模块。

下一章中，将深入研究 CMake 的编程语言。可以来编写自己的 CMake 文件了，并为创建第一个脚本、项目和模块打开大门。

1.8. 扩展阅读

有关更多信息，可以参考以下资源：

- 官方的 CMake 网页和文档: <https://cmake.org/>
- 单配置生成器: <https://cgold.readthedocs.io/en/latest/glossary/single-configuration.html>
- CMake GUI 的分离阶段: <https://stackoverflow.com/questions/39401003/why-there-are-two-buttons-in-guiconfigure-and-generate-when-cli-does-all-in-one>

第 2 章 CMake 语法

用 CMake 语言编写有点棘手，第一次阅读 CMake 文件时，可能会有这样的印象：其中的语言非常简单。接下来的工作通常是在没有完全理解代码如何工作的情况下，尝试引入更改并对代码进行试验。开发者通常很忙，并且过分热衷于用很少的投资解决与构建相关的问题。所以倾向于做出基于直觉的改变，希望能奏效，这种解决技术问题的方法称为“巫毒编程”。

CMake 语言看起来很简单：在完成了添加、修复或修改或添加一行代码后，意识到有些东西无法工作，花在调试上的时间通常比花在实际研究主题上的时间要长。不过，本章将介绍在实践中使用 CMake 语言所需的绝大多数关键性知识。

本章中，不仅将学习 CMake 语言的构建模块——注释、指令、变量和控制结构——而且还将给出必要的背景知识，并在一个干净而现代的 CMake 示例中进行尝试。CMake 将开发者置于一个独特的位置：一方面，扮演构建工程师的角色——需要理解编译器、平台以及两者之间的所有复杂之处；另一方面，你是一个开发人员——需要编写生成构建系统的代码。编写好的代码是困难的，需要同时从多个维度进行思考——它应该能够工作并易于阅读，但也应该易于分析、扩展和维护。

最后，将介绍 CMake 中一些最有用和最常用的指令。不经常使用的指令将放在附录部分（这将包括字符串、列表和文件操作命令的完整参考指南）。

本章中，我们将讨论以下主题：

- 基本语法
- 变量
- 列表
- 控制结构
- 实用指令

2.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter02>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

2.2. 基本语法

编写 CMake 代码类似于用其他命令式语言编写代码：行从上到下、从左到右执行，偶尔会进入包含的文件或调用的函数。根据模式的不同（参见第 1 章），执行从源树的根文件 (`CMakeLists.txt`) 或者作为参数传递给 CMake 的 `.cmake` 脚本开始。

脚本支持大多数 CMake 语言(排除任何与项目相关功能), 所以是练习 CMake 语法的一个好方法。熟练地编写基本的列表文件之后, 将在下一章准备实际的项目文件, 可以使用以下命令运行脚本:

```
cmake -P script.cmake
```

Note

CMake 支持 7 位 ASCII 文本文件, 以实现跨平台的可移植性。可以同时使用 \n 或 \r\n 作为换行符。3.0 以上的 CMake 版本支持带有可选字节顺序标记(BOM)的 UTF-8, 3.2 以上的 CMake 版本支持 UTF-16。

CMake 文件中的所有内容不是指令调用就是注释。

2.2.1 注释

像 C++ 一样, 有两种注释——单行注释和括号(多行)注释。但与 C++ 不同的是, 括号注释可以嵌套:

```
# single-line comments start with a hash sign "#"
# they can be placed on an empty line
message("Hi"); # or after a command like here.

# [= [
bracket comment
# [[
    nested bracket comment
# ]]
# ]=]
```

多行注释的名称来自于它们的符号——以一个方括号([)、任意数量的等号(=)和另一个方括号[=开始。要关闭一个括号注释, 请使用相同数量的等号, 并像这样反转括号:]=]。

使用 # 在左括号标记前加上前缀是可选的, 并且可以通过在括号注释的第一行添加另一个# 来快速禁用多行注释, 就像这样:

```
## [= this is a single-line comment now
no longer commented
# [[
    still, a nested comment
# ]]
#=] this is a single-line comment now
```

这是一个巧妙的技巧, 但如何在 CMake 文件中使用注释呢? 由于编写列表文件本质上是编程, 因此将编码实践应用到文件中是一个好主意。遵循这种实践的代码通常认为是整洁的——这个术语

由软件开发大师使用，如 Robert C. Martin、Martin Fowler。什么是有益的或有害的，经常会引起激烈的争论，注释也一样。

每件事都应该在个案的基础上进行判断，但通常的指导方针认为，好的注释至少提供以下其中之一：

- 信息：可以解决诸如正则表达式模式或格式化字符串等复杂问题。
- 意图：当代码的意图在实现或接口中不明显时，可以解释代码的意图。
- 澄清：可以解释那些不容易重构或更改的概念。
- 结果警告：可以提供警告，特别是针对可能破坏其他内容的代码。
- 强调：可以强调难以用代码表达的想法的重要性。
- 法律条款：可以添加这个必要的法律条款，这通常不是程序员所擅长的领域。

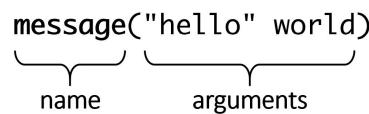
若想避免添加注释，可以采用更好的命名实践，或者重构或修改代码。可以避免添加以下类型的注释：

- 强制：强制添加这些注释是为了完整性，但不真正重要。
- 冗余：重复代码中已经明确写好的内容。
- 误导：若不跟随代码更改，则可能过时或不正确。
- 日志：记录更改的内容和时间（使用 VCS 替代）。
- 标记：仅用来进行标记。

编写没有注释的优雅代码非常困难，注释改善了读者的体验。因为我们花在读代码上的时间比写代码多，所以应该编写可读性好的代码，而不是仅仅试图快速地编写代码。建议查看本章末尾的扩展阅读部分，以获得一些关于干净代码的参考。若对注释特别感兴趣，可以找到我在 YouTube 上关于这个主题的视频链接。

2.2.2 执行指令

执行指令是 CMake 列表文件的基本功能，必须提供它的名称，后面跟着圆括号，在圆括号中可以包含一个以空格分隔的指令参数列表。



```
message("hello" world)
```

name arguments

图 2.1 指令示例

指令名不区分大小写，但在 CMake 社区中有一个约定，即在指令名中使用 `snake_case`（即小写单词与下划线连接）。

也可以定义自己的指令，会在“控制结构”一节中介绍这些指令。

与 C++ 相比，CMake 中的指令调用不是表达式。不能为调用指令提供另一个指令作为参数，因为括号之间的所有内容都会解释为该指令的参数。

CMake 指令不需要在调用结束时使用分号。这可能是因为每行源代码最多可以包含一个命令调用，后面跟着一个可选的单行注释。或者，整行必须是括号注释的一部分。所以，这些是唯一允许的格式：

```
command(argument1 "argument2" argument3) # comment  
[[ multiline comment ]]
```

不允许将指令放在括号注释后面:

```
[[ bracket  
]] command()
```

删除所有注释、空格和空行之后，得到一个指令列表。CMake 语法真的很简单，但这是一件好事吗？怎么处理变量呢？或者，如何指导执行的流程？

CMake 为这些操作提供了命令和更多的功能。为了使事情更简单，将在介绍不同示例时介绍相关指令，可以分为三类：

- 脚本指令：脚本指令可用，会改变指令处理器、访问变量的状态，并影响其他指令和环境。
- 项目指令：这些指令在项目中可用，操纵项目状态并构建目标。
- CTest 指令：这些指令在 CTest 脚本中可用，管理测试。

我们将在本章中介绍脚本指令（因为它们在项目中也很有用）。项目和 CTest 指令将在接下来的章节中讨论，将介绍与构建目标（第 3 章）和测试框架（第 8 章）相关的概念。

实际上，每个指令都依赖于该语言的其他元素来发挥作用：变量、条件语句，以及最重要的指令参数。来看看如何使用这些参数。

2.2.3 指令参数

许多指令需要用空格分隔的参数来参数化它们的行为。正如在图 2.1 中看到的，参数周围的引号发生了一些奇怪的事情。有些论点有引用，有些没有，这是怎么回事？

在底层，CMake 识别的唯一数据类型是字符串。这就是为什么每个命令的参数都期望 0 个或多个字符串。但是普通的静态字符串并不是很有用，特别是不能嵌套指令调用时。这就是参数发挥作用的地方——CMake 将计算静态字符串的每个参数，然后传递到命令中。求值意味着字符串插值，或者用另一个值替换字符串的部分。这可以替换转义序列、扩展变量引用（也称为变量插值）和解包列表。

根据上下文的不同，可以根据需要启用这种计算。因此，CMake 提供了三种类型的参数：

- 方括号参数
- 引号参数
- 非引号参数

每种参数类型提供不同的求值级别，并有一些小特点。

方括号参数

方括号参数不会进行求值，并用于将多行字符串作为单个参数逐字传递给命令。所以将包括制表符和换行符形式的空白。

这些参数的结构与注释完全相同——以 [=开始，以=] 结束，其中开始和结束标记中的等号数量必须匹配（跳过等号也可以，但必须匹配）。与注释的唯一区别是不能嵌套括号参数。

下面是 message() 指令使用这种参数的示例，将所有传递的参数打印到屏幕上：

```
# chapter02/01-arguments/bracket.cmake

message([ [multiline
bracket
argument
]])

message( [==[
because we used two equal-signs "==""
following is still a single argument:
{ "petsArray" = [[ "mouse", "cat" ], [ "dog" ] ] }
]==])
```

上面的例子中，可以看到不同形式的括号参数。第一个跳过了等号，将结束标记放在单独的行上，在输出中显示为空行：

```
$ cmake -P chapter02/01-arguments/bracket.cmake
multiline
bracket
argument

because we used two equal-signs "==""
following is still a single argument:
{ "petsArray" = [[ "mouse", "cat" ], [ "dog" ] ] }
```

第二种形式在传递包含双括号 ([]) (在代码片段中突出显示) 的文本时很有用，不会解释为标记参数的结束。

这类括号参数的用途有限——通常用于包含较长的文本块。大多数情况下，需要一些更动态的东西，比如引号参数。

引号参数

带引号的参数类似于常规的 C++ 字符串——这些参数将多个字符组合在一起，包括空格，它们将展开转义序列。与 C++ 字符串一样，开始和结束都使用双引号 ("")，因此要在输出字符串中包含一个引号字符，必须使用反斜杠对其进行转义 (\")。

也支持其他转义序列：\\表示字面反斜杠，\t 是一个制表符，\n 是换行符，和\r 是一个回车符。

这就是与 C++ 字符串的相似之处。带引号的参数可以跨越多行，将插入变量引用。可以把它们想象成拥有来自 C 语言的内置 sprintf 函数，或者 C++20 的 std::format 函数。要将变量引用插入到参数中，可以将变量名包装在字符串中，就像这样: \${name}。

我们将在“变量”一节中更多地讨论变量引号。

```
# chapter02/01-arguments/quoted.cmake

message("1. escape sequence: \" \n in a quoted argument")
message("2. multi...
line")
message("3. and a variable reference: ${CMAKE_VERSION}")
```

能猜到前面脚本的输出中有多少行吗？

```
$ cmake -P chapter02/01-arguments/quoted.cmake
1. escape sequence: "
in a quoted argument
2. multi...
line
3. and a variable reference: 3.16.3
```

没错——有一个转义引号字符、一个转义换行符和一个字面换行符。所有这些都将打印在输出中。还访问了一个内置的 CMAKE_VERSION 变量，可以看到在最后一行正确插入了该变量。

非引号参数

最后一种类型的参数在编程世界中肯定有点罕见。我们已经习惯了字符串必须以这样或那样的方式分隔的事实，例如：单引号、双引号或反斜杠。CMake 背离了这一惯例，引入了不加引号的实参。删除分隔符使代码更容易阅读，就像跳过分号一样。真是这样吗？

不加引号的实参计算转义序列和变量引用，但要小心使用分号 (;)，就像在 CMake 中一样，其会当作分隔符对待。CMake 会将包含它的参数拆分为多个参数，可以使用反斜杠对其进行转义 (\;)。这就是 CMake 管理列表的方式。

可能会发现这些参数是最复杂的，所以这里有一个例子可用来表明如何划分这些参数：

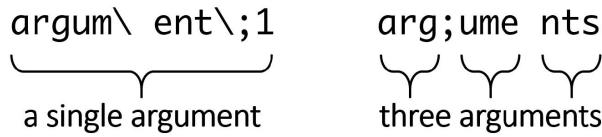


图 2.2 转义字符将使后续的字符解释为单个参数

问题

为什么“一个值是作为一个参数传递，还是作为多个参数传递”很重要？一些 CMake 指令需要特定数量的参数，并忽略开销。若参数分离，就会出现难以调试的错误。

不加引号的参数不能包含未转义的引号 (')、井号 (#) 和反斜杠 (\)，只有当它们形成正确的、匹配的对时才允许使用括号 (())。从一个左括号开始，然后在关闭指令参数列表之前关闭它。

来看看以上规则的例子:

```
# chapter02/01-arguments/unquoted.cmake

message(a\ single\ argument)
message(two arguments)
message(three;separated;arguments)
message(${CMAKE_VERSION}) # a variable reference
message((()()()) # matching parentheses
```

上面的输出是什么?一起来看看:

```
$ cmake -P chapter02/01-arguments/unquoted.cmake
a single argument
twoarguments
threeseparatedarguments
3.16.3
()()()
```

即使是像 message() 这样的简单指令，也非常讲究参数之间不加引号的分隔:

- 显式转义单个参数中的空格时，正确打印空格。
- message() 不会添加任何空格，所以两个参数和三个单独的参数会粘在一起。

已经了解了如何处理 CMake 参数的复杂性和特殊之处，就可以学习下一个有趣的主题——变量。

2.3. 变量

CMake 中的变量是一个复杂的主题。有三类变量——普通变量、缓存变量和环境变量——而且它们还在不同的作用域中，并对一个作用域如何影响另一个作用域具有特定的规则。通常情况下，对所有这些规则的不理解会成为 bug 和头痛的根源。我建议仔细学习这一节，确保在继续学习之前理解了这些概念。

先从 CMake 中关于变量的关键点开始:

- 变量名区分大小写，可以包含任何字符。
- 变量都在内部作为字符串存储，有些智力可以解释为其他数据类型的值(甚至是列表!)
- 基本的变量操作指令是 set() 和 unset()，但还有其他指令可以影响变量，如 string() 和 list()。

要设置变量，只需使用 set()，提供名称和值:

```
# chapter02/02-variables/set.cmake

set(MyString1 "Text1")
```

```
set([[My String2]] "Text2")
set("My String 3" "Text3")
message(${MyString1})
message(${My\ String2})
message(${My\ String\ 3})
```

使用括号和引号参数允许在变量名中包含空格。但当以后引用时，必须使用反斜杠来转义空格(\)。因此，建议在变量名中只使用字母数字字符、减号(-)和下划线(_)。

还要避免使用任何字符开头的保留名称(上、下或混合大小写):CMAKE_、_CMAKE_或下划线(_)，后面跟着 CMake 指令的名称。

Note

set() 指令接受纯文本变量名作为第一个参数，但 message() 使用包装在 \${} 语法中的变量引用。若向 set() 提供包装在 \${} 语法中的变量，会发生什么？要回答这个问题，需要更好地理解引用变量。

要取消变量的设置，可以使用: unset(MyString1)。

2.3.1 引用变量

已经在指令参数部分提到了引用，这是为引号和非引号参数进行计算的。创建对已定义变量的引用，需要使用 \${} 语法:message(\${MyString1})。

求值时 CMake 将遍历作用域堆栈，并将 \${MyString1} 替换为值，若没有找到变量则替换为空字符串(CMake 不会产生错误)。这个过程称为变量求值、展开或插值。

这样的插值是由内而外的方式执行的:

- 若遇到以下引用——\${MyOuter}\${MyInner}——CMake 将首先尝试求值 MyInner，而不是搜索名为 MyOuter\${MyInner} 的变量。
- 若 MyInner 变量成功展开，CMake 将重复展开过程，直到不能再展开为止。

考虑包含以下变量的例子:

- MyInner 的值是 Hello
- MyOuter 的值是 \${My

若使用 message("\${MyOuter}Inner} World")，输出将是 Hello World，这是因为 \${My 替换了 \${MyOuter}，当与顶层值 Inner} 结合时，会创建另一个变量引用——\${MyInner}。

CMake 执行这个扩展，只有这样它才会将结果值作为参数传递给命令。这就是为什么使用 set(\${MyInner} "Hi") 时，实际上不会更改 MyInner 变量，而是更改 Hello 变量。CMake 将 \${MyInner} 展开为 Hello，并将该字符串作为第一个参数传递给 set()，以及一个新值 Hi。很多时候，这并不是我们想要的。

当涉及到变量类别时，变量引用的工作方式有点奇怪。以下是通常情况适用的方式:

- \${} 用于引用普通变量或缓存变量。
- \$ENV{} 用于引用环境变量。

- `$CACHE{}` 用于引用缓存变量。

没错，使用 `${}{}` 以从一个类别或另一个类别获得值，后续章节会有详解。先介绍一些其他类型的变量，以便了解它们是什么。

Note

可以通过命令行在--标记之后将参数传递给脚本。值将存储在 `CMAKE_ARGV<n>` 变量中，传递的参数的计数将存储在 `CMAKE_ARGC` 变量中。

2.3.2 环境变量

这是最简单的一种变量。CMake 生成环境中用于启动 CMake 进程的变量的副本，并使它们在单一的全局作用域中可用。要引用这些变量，使用 `$ENV{<name>}`。

CMake 还允许设置变量 (`set()`) 和取消设置变量 (`unset()`)，但更改只会在运行的 CMake 进程中对本地副本进行，而不会对实际的系统环境进行更改；此外，这些更改对于构建或测试的后续运行不可见。

要修改或创建一个变量，使用 `set(ENV{<variable>} <value>)` 指令：

```
set(ENV{CXX} "clang++")
```

要清除环境变量，使用 `unset(ENV{<variable>})`：

```
unset(ENV{VERBOSE})
```

注意，有几个环境变量会影响 CMake 行为的不同方面。`CXX` 变量就是其中之一——指定编译 C++ 文件时，使用什么可执行文件。我们将讨论其他环境变量，因为它们与本书相关。完整的列表可以在文档中找到：

<https://cmake.org/cmake/help/latest/manual/cmake-envvariables.7.html>

若使用 `ENV` 变量作为指令的参数，这些值将在生成构建系统期间插入，并且会将其嵌入到构建树中。

例子：

```
# chapter02/03-environment/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Environment)
message("generated with " $ENV{myenv})
add_custom_target(EchoEnv ALL COMMAND echo "myenv in build
is" $ENV{myenv})
```

前面的示例有两个步骤：将在配置期间打印 `myenv` 环境变量，并通过 `add_custom_target()` 添加一个构建阶段，该阶段作为构建过程的一部分响应相同的变量。可以测试 `bash` 脚本会发生什么，在配置阶段使用一个值，在构建阶段使用另一个值：

```
# chapter02/03-environment/build.sh
```

```
#!/bin/bash
export myenv=first
echo myenv is now $myenv
cmake -B build .
cd build
export myenv=second
echo myenv is now $myenv
cmake --build .
```

运行上面的代码，可以清楚地看到在配置过程中，设置的值会保留在生成的构建系统中：

```
$ ./build.sh | grep -v "\-\-"
myenv is now first
generated with first
myenv is now second
Scanning dependencies of target EchoEnv
myenv in build is first
Built target EchoEnv
```

2.3.3 缓存变量

它们是存储在构建树中的 CMakeCache.txt 文件中的变量，包含在项目配置阶段收集的信息，包括从系统 (到编译器、链接器、工具和其他的路径) 和通过 GUI 从用户收集的信息。缓存变量在脚本中不可用 (因为没有 CMakeCache.txt 文件)——其只存在于项目中。

缓存变量可以通过 `$CACHE{<name>}` 语法来引用。

要设置一个缓存变量，使用 `set()`:

```
set(<variable> <value> CACHE <type> <docstring> [FORCE])
```

这里有一些必需参数 (与用于普通变量的 `set()` 指令相比)，还引入了第一个关键字: CACHE 和 FORCE。

将 CACHE 指定为 `set()` 参数可以更改在配置阶段提供的内容，要求提供变量 `<type>` 和 `<docstring>` 值。这是因为用户可以配置这些变量，GUI 需要知道如何显示它们。通常接受以下类型：

- BOOL: 一个布尔的开/关值。GUI 将显示一个复选框。
- FILEPATH: 磁盘上文件的路径。GUI 将打开一个文件对话框。
- STRING: 一行字符串。GUI 提供了一个要填充的文本字段，可以通过下拉控件选择替换。

```
set_property(CACHE <variable> STRINGS <values>)
```

- INTERNAL: 一行字符串。GUI 跳过内部条目。内部条目可以用于跨运行持久化存储变量。使用此类型会隐式添加 FORCE 关键字。

<doctring> 值只是一个标签，将由 GUI 显示在字段旁边，以便向用户提供关于该设置的更多详细信息。即使是 INTERNAL 类型也需要它。

设置缓存变量遵循与环境变量相同的规则——仅在当前执行 CMake 时覆盖值：

```
set(FOO "BAR" CACHE STRING "interesting value")
```

若变量存在于缓存中，上述调用不会产生永久影响。然而，若该值在缓存中不存在或指定了可选的 FORCE 参数，则该值将固定：

```
set(FOO "BAR" CACHE STRING "interesting value" FORCE)
```

设置缓存变量有一些不明显的含义，具有相同名称的普通变量都将删除。我们将在下一节中找出原因。

需要提醒的是，缓存变量也可以在命令行端进行管理。

2.3.4 如何正确使用变量作用域

变量作用域可能是 CMake 语言整个概念中最难的部分。这可能是因为我们太习惯于在更高级语言中做事情的方式。CMake 没有这些机制，所以它用自己的方式处理这个问题。

需要说明的是，变量作用域作为一个通用概念是为了分离不同的抽象层，以便在调用用户定义的函数时，该函数中设置的变量是局部的。这些局部变量不会影响全局作用域，即使局部变量的名称与全局变量的名称完全相同。若显式需要，函数也应该具有对全局变量的读/写访问权。这种变量（或作用域）分离必须在多个层面上工作——当一个函数调用另一个函数时，分离规则同样适用。

CMake 有两个作用域：

- 函数作用域：用于执行用 function() 定义的自定义函数
- 目录作用域：当从 add_subdirectory() 指令执行嵌套目录中的 CMakeLists.txt 文件时

将在本书后面介绍上述指令。首先，需要知道变量作用域的概念如何实现。当创建嵌套作用域时，CMake 只需用来自当前作用域的所有变量的副本填充。后续命令将影响这些副本。但若完成了嵌套作用域的执行，所有的副本都会删除，而原始的父作用域将恢复。

考虑以下场景：

1. 父作用域将 VAR 变量设置为 ONE。
2. 开始嵌套作用域，并将 VAR 输出到控制台。
3. VAR 变量设置为 TWO，并且 VAR 输出到控制台。
4. 嵌套的作用域结束，VAR 再输出到控制台。

控制台的输出应该是：ONE, TWO, ONE。这是因为在嵌套作用域结束后，复制的 VAR 变量将丢弃。

CMake 中作用域的概念如何工作，若在嵌套作用域中执行时取消设置 (unset()) 父作用域中创建的变量，只会在嵌套作用域中消失。当嵌套作用域完成时，变量将恢复到以前的值。

这就引出了变量引用的行为和 \${} 语法。每当尝试访问普通变量时，CMake 将从当前作用域搜索变量，若定义了具有这样一个名称的变量，它将返回其值，但当 CMake 找不到具有该名称的变量时（例如，若它不存在或未设置 (unset())），将搜索缓存变量，若找到匹配则从那里返回一个值。

若有一个使用 unset() 的嵌套作用域，这可能是个问题。根据引用该变量的位置——内部还是外部作用域——将访问缓存或原始值。

若真的需要更改调用(父)作用域中的变量，该怎么办呢？

CMake 有一个 PARENT_SCOPE 标志，可以在 set() 和 unset() 指令的末尾添加：

```
set(MyVariable "New Value" PARENT_SCOPE)
unset(MyVariable PARENT_SCOPE)
```

这种解决方法有一定的局限性，因为其不允许访问超过一个级别的变量。另一件值得注意的事情是，使用 PARENT_SCOPE 不会改变当前作用域中的变量。

来看看变量作用域在实践中是如何工作的：

```
# chapter02/04-scope/CMakeLists.txt

function(Inner)
    message(" > Inner: ${V}")
    set(V 3)
    message(" < Inner: ${V}")
endfunction()

function(Outer)
    message(" > Outer: ${V}")
    set(V 2)
    Inner()
    message(" < Outer: ${V}")
endfunction()

set(V 1)
message("> Global: ${V}")
Outer()
message("< Global: ${V}")
```

将全局变量 V 设为 1，然后调用 Outer 函数；然后将 V 设置为 2 并调用 Inner 函数，然后将 V 设置为 3。每一步后面，都将变量打印到控制台：

```
> Global: 1
> Outer: 1
    > Inner: 2
    < Inner: 3
    < Outer: 2
< Global: 1
```

深入函数时，变量值会复制到嵌套作用域，但当退出作用域时，原始值将恢复。

若改变 Inner 函数的 set() 来操作父作用域:set(V 3 PARENT_SCOPE)，输出会是什么？

```
> Global: 1
> Outer: 1
> Inner: 2
< Inner: 2
< Outer: 3
< Global: 1
```

我们影响了外部函数的作用域，但没有影响内部函数的作用域或全局作用域！

CMake 文档还提到 CMake 脚本在一个目录范围内绑定变量（这有点多余，因为唯一有效创建目录范围的命令 add_subdirectory() 在脚本中是不允许的）。

由于所有变量都存储为字符串，CMake 必须采用更有效的方法来处理更复杂的数据结构，如列表。

2.4. 列表

要存储列表，CMake 会将所有元素连接成一个字符串，使用分号 (;) 分隔:a;list;of;5;elements。可以用反斜杠转义元素中的分号，像这样: a\;single\;element。

要创建一个列表，可以使用 set() 指令:set(myList 一个包含五个元素的列表)。由于列表的存储方式不同，下面的命令将具有完全相同的效果：

- set(myList "a;list;of;five;elements")
- set(myList a list "of;five;elements")

CMake 会自动解包未加引号的参数中的列表。通过传递一个不加引号的 myList 引用，可以有效地向指令传递更多的参数：

```
message("the list is:" ${myList})
```

message() 指令将在这里接收 6 个参数：“list is:”，“a”，“list”，“of”，“five”，“elements”。这可能会产生意想不到的后果，因为输出将在参数之间没有任何空格的情况下打印出来：

```
the list is:alistoffiveelements
```

这是一个非常简单的机制，请谨慎使用。

CMake 提供了一个 list() 指令，提供了大量子命令来读取、搜索、修改和排序列表。下面是其中一部分：

```
list(LENGTH <list> <out-var>)
list(GET <list> <element index> [<index> ...] <out-var>)
list(JOIN <list> <glue> <out-var>)
```

```

list(SUBLIST <list> <begin> <length> <out-var>)
list(FIND <list> <value> <out-var>)
list(APPEND <list> [<element>...])
list(FILTER <list> { INCLUDE | EXCLUDE } REGEX <regex>)
list(INSERT <list> <index> [<element>...])
list(POP_BACK <list> [<out-var>...])
list(POP_FRONT <list> [<out-var>...])
list(PREPEND <list> [<element>...])
list(REMOVE_ITEM <list> <value>...)
list(REMOVE_AT <list> <index>...)
list(REMOVE_DUPLICATES <list>)
list(TRANSFORM <list> <ACTION> [...])
list(REVERSE <list>)
list(SORT <list> [...])

```

大多数时候，并不需要在项目中使用列表。但会发现在极少数情况下这个概念是方便的，可以在附录部分中找到 `list()` 指令的更多信息。

现在已经知道了如何使用各种类型的列表和变量，我们将注意力转移到控制执行流上，并学习 CMake 中可用的控制结构。

2.5. 控制结构

若没有控制结构，CMake 语言就不完整！和其他所有东西一样，以命令的形式提供，分为三类：条件块、循环和定义指令。控制结构在脚本中执行，并在项目的构建系统生成过程中执行。

2.5.1 条件块

CMake 中支持的条件块是不起眼的 `if()` 指令。所有条件块必须用 `endif()` 关闭，可以有任意数量的 `elseif()` 和一个可选的 `else()`，顺序如下：

```

if(<condition>
  <commands>
elseif(<condition>) # optional block, can be repeated
  <commands>
else() # optional block
  <commands>
endif()

```

与许多其他命令式语言一样，`if()-endif()` 块控制将执行哪些指令集：

- 若满足 `if()` 指令中指定的 `<condition>` 表达式，则执行第一部分。
- 否则，CMake 将在属于该块中满足条件的第一个 `elseif()` 指令节中执行命令。
- 若没有这样的命令，CMake 将检查是否提供了 `else()`，并执行该部分代码中的指令。

- 若以上条件都不满足，则在 `endif()` 之后继续执行。
- 提供的 `<condition>` 表达式根据非常简单的语法求值。

2.5.2 条件指令的语法

同样的语法也适用于 `if()`、`elseif()` 和 `while()` 指令。

逻辑运算符

`if()` 条件支持 NOT、AND 和 OR 逻辑操作符：

- NOT `<condition>`
- `<condition> AND <condition>`
- `<condition> OR <condition>`

条件的嵌套也可以通过匹配的括号 `()` 来实现。和所有体面的语言一样，CMake 语言尊重求值的顺序，从最里面的括号开始：

- `(<condition>) AND (<condition> OR (<condition>))`

字符串和变量的求值

由于历史原因（因为变量引用 `(${})` 语法并不总是存在），CMake 将尝试计算未加引号的参数，就像是变量引用一样。换句话说，在条件中使用普通变量名（例如 `VAR`）等于写入 `${VAR}`。这里有一个例子：

```
set(VAR1 FALSE)
set(VAR2 "VAR1")
if(${VAR2})
```

`if()` 条件在这里的工作方式有点复杂——首先， `${VAR2}` 求值为 `VAR1`。`VAR1` 是一个可识别的变量，而这个变量的值为 `FALSE` 字符串。只有当字符串等于以下任何一个常量时（这些比较不区分大小写），才认为是布尔值 `true`：

- ON, Y, YES 或 TRUE
- 一个非零的数

我们得出结论，上例中的条件将计算为 `false`。

这里还有另一个问题——若条件的参数没有加引号，且变量的名称包含一个值，如 `BAR`，那么该如何求值呢？考虑下面的代码示例：

```
set(FOO BAR)
if(FOO)
```

根据我们到目前为止所说的，它将是 `false`，因为 `BAR` 字符串不满足计算布尔值 `true` 值的标准。但情况并非如此，因为 CMake 在涉及到未加引号的变量引用时会出现例外。与带引号的参数不同，`FOO` 不会通过 `BAR` 求值，以生成 `if("BAR")` 语句（这将是 `false`）。相反，CMake 只会在 `if(FOO)` 为 `false` 时计算它是以下任何一个常量（这些比较不区分大小写）：

- OFF, NO, FALSE, N, IGNORE, NOTFOUND
- 以-NOTFOUND 结尾的字符串
- 一个空字符串
- 零

因此，未定义的变量将赋值为 false:

```
if (FOO)
```

但先定义一个变量会改变这种情况，并且计算为 true:

```
set(FOO "FOO")
if (FOO)
```

Note

若认为不加引号的参数的行为令人困惑，请将变量引用包装在加引号的参数中:`:if("${FOO}")`。这将导致在提供的参数传递到 if() 指令之前进行参数求值，并且行为将与字符串的求值一致。

所以，CMake 假设用户正在询问是否定义了变量（并且不显式为 false）。可以显式检查这个事实（而不用担心值）：

```
if(DEFINED <name>)
if(DEFINED CACHE{<name>})
if(DEFINED ENV{<name>})
```

比较

以下操作符支持比较操作：

EQUAL, LESS, LESS_EQUAL, GREATER 和 GREATER_EQUAL

可以用来比较数值：

```
if (1 LESS 2)
```

Note

CMake 文档声明，若其中一个操作数不是数字，则该值将为 false。实际实验表明，以数字开头的字符串的比较是正确的：`:if(20 EQUALS "20 GB")`。

可以比较 [main.[.minor[.patch]]] 之后的版本，可以在操作符前添加一个 VERSION_ 前缀来调整格式：

```
if (1.3.4 VERSION_LESS_EQUAL 1.4)
```

省略的组件视为零，非整数版本的组件在该点截断比较字符串。

对于字典字符串比较，需要在操作符前加上 STR 前缀（注意没有下划线）：

```
if ("A" STREQUAL "${B}")
```

通常需要比简单的相等比较更高级的机制。

CMake 还支持 POSIX 正则表达式匹配 (CMake 文档暗示了 ERE 风格，但没有提到对特定正则表达式字符类的支持)。

可以这样使用 MATCHES 操作符:`<variable|string> MATCHES <regex>` 匹配的组都在 CMAKE_MATCH_<n> 变量中。

简单的检查

已经提到了一个简单的检查，DEFINED。但是还需要其他检查，若满足条件则返回 true。

可以检查以下内容：

- 若值在列表中:`<variable|string> in _LIST <variable>`
- 若指令可用:`:command <command-name>`
- 若 CMake 策略存在:`:POLICY <policy-id>`(这将在第 3 章中介绍)
- 若使用 `add_test()` 添加 CTest 测试:`:test <test-name>`
- 若定义了构建目标:`:target <target-name>`

我们将在第 4 章中了解构建目标。现在，只说目标是项目中构建过程的逻辑单元，该项目使用 `add_executable()`、`add_library()` 或 `add_custom_target()` 指令创建，这些指令我们已经使用过了。

文件系统检查

CMake 提供了许多处理文件的方法。很少需要直接操作它们，通常会使用高级方法，本书将在附录部分提供与文件相关的命令的简短列表。大多数情况下，只需要以下操作符(行为只对绝对路径有很好的定义)：

- EXISTS `<path-to-file-or-directory>`: 检查文件或目录是否存在
这将解析符号链接(若符号链接的目标存在，则返回 true)。
- `<file1> IS_NEWER_THAN <file2>`: 检查哪个文件更新
如果 file1 比(或等于)file2 更新，或者两个文件中有一个不存在，则返回 true。
- IS_DIRECTORY `path-to-directory`: 检查路径是否为目录
- IS_SYMLINK `file-name`: 检查路径是否为符号链接
- IS_ABSOLUTE `path`: 检查路径是否为绝对路径

2.5.3 循环

CMake 中的循环相当简单——可以使用 `while()` 或 `foreach()` 重复执行同一组命令。这两个命令都支持循环控制机制：

- `break()` 循环停止剩余块的执行，并从封闭循环中断开。
- `continue()` 循环停止当前迭代的执行，并开启下一个迭代。

While

循环块用 `while()` 指令创建，用 `endwhile()` 关闭。只要 `while()` 中提供的 `<condition>` 表达式为 true，其后续的指令都会执行。条件语句的语法与 `if()` 指令相同：

```
while (<condition>
    <commands>
endwhile ())
```

通过一些额外的变量——while 循环可以替换 for 循环。实际上，使用 foreach() 循环要容易得多。

Foreach

foreach 块有几个变体，为每个值执行附带的指令，它有打开和关闭指令：foreach() 和 endforeach()。foreach() 最简单的形式是为了提供 C++ 风格的 for 循环：

```
foreach (<loop_var> RANGE <max>
    <commands>
endforeach ())
```

CMake 将从 0 迭代到 <max>(包括)。若需要更多的控制，可以使用第二个变量，提供 <min>、<max> 和 <step>(可选)。所有参数必须是非负整数。同时，<min> 必须小于 <max>：

```
foreach (<loop_var> RANGE <min> <max> [<step>])
```

然而，foreach() 在处理列表时展示了它的真面目：

```
foreach (<loop_variable> IN [LISTS <lists>] [ITEMS <items>])
```

CMake 将从所有提供的 <lists> 列表变量中获取元素，然后是所有显式声明的 <items> 值，并将它们存储在 <loop_variable> 中，对每个项逐个执行 <commands>。可以选择只提供列表，只提供值，或者两者都提供：

```
# chapter02/06-loops/foreach.cmake

set (MY_LIST 1 2 3)
foreach (VAR IN LISTS MY_LIST ITEMS e f)
    message (${VAR})
endforeach ()
```

上述代码将打印以下内容：

```
1
2
3
e
f
```

或者，使用一个简短的版本(跳过 IN 关键字)来获得相同的结果：

```
foreach(VAR 1 2 3 e f)
```

从 3.17 版本开始，`foreach()` 已经学会了如何压缩列表 (ZIP_LISTS):

```
foreach(<loop_var>... IN ZIP_LISTS <lists>)
```

压缩列表可以遍历多个列表并处理具有相同索引的各自项:

```
# chapter02/06-loops/foreach.cmake

set(L1 "one;two;three;four")
set(L2 "1;2;3;4;5")
foreach(num IN ZIP_LISTS L1 L2)
  message("num_0=${num_0}, num_1=${num_1}")
endforeach()
```

CMake 将为每个提供的列表创建一个 `num_<N>` 变量，用每个列表中的项填充该变量。可以传递多个 `<loop_var>` 变量名 (每个列表一个)，每个列表将使用单独的变量来存储:

```
foreach(word num IN ZIP_LISTS L1 L2)
  message("word=${word}, num=${num}")
```

若列表之间的项数不同，CMake 将不会为较短的列表定义变量。

以上就是关于循环的所有内容。

2.5.4 定义指令

有两种方法可以定义自己的命令: 可以使用 `macro()` 或 `function()`。要解释这两个指令间的区别，最简单的方法是将它们与 C 风格的宏和实际的 C++ 函数进行比较:

- `macro()` 的工作方式更像是查找和替换指令，而不是像 `function()` 这样的实际子例程调用。与函数相反，宏不会在调用堆栈上创建单独的条目。所以宏中调用 `return()` 将比在函数中返回调用语句的级别高一级 (若已经在顶层作用域中，可能会终止执行)。
- `function()` 为本地变量创建一个单独的作用域，这与 `macro()` 命令不同，后者在调用者的变量作用域中工作。这可能会导致令人困惑的结果。让我们在下一节中讨论其中细节。

这两个方法都接受可以在命令块中命名和引用的参数。此外，CMake 允许通过以下引用访问命令调用中传递的参数:

- `${ARGC}`: 参数的数量
- `${ARGV}`: 所有参数的列表
- `${ARG0}, ${ARG1}, ${ARG2}`: 特定索引处的实参值
- `${ARGN}`: 最后一个预期参数之后，由调用者传递的匿名参数列表

使用 `ARGC` 边界外的索引访问数值参数会产生未定义行为。

若确定需要定义一个带有命名参数的指令，则每次调用都必须传递所有这些参数，否则将无效。

宏

定义宏类似于任何其他块:

```
macro (<name> [<argument>⋯⋯])
  <commands>
endmacro ()
```

此声明之后，可以通过调用宏的名称来执行宏(函数调用不区分大小写)。

下面的例子强调了与宏中变量作用域相关的问题:

```
# chapter02/08-definitions/macro.cmake

macro (MyMacro myVar)
  set(myVar "new value")
  message("argument: ${myVar}")
endmacro()

set(myVar "first value")
message("myVar is now: ${myVar}")
MyMacro("called value")
message("myVar is now: ${myVar}")
```

下面是这个脚本的输出:

```
$ cmake -P chapter02/08-definitions/macro.cmake
myVar is now: first value
argument: called value
myVar is now: new value
```

发生了什么事? 尽管显式地将 myVar 设置为新值, 但并不影响 message("argument:\${myVar}")! 这是因为传递给宏的参数没有视为真正的变量, 而是作为常量查找并替换指令。

另一方面, 全局作用域中的 myVar 变量从第一个值更改为新值。这种行为称为副作用, 是一种糟糕的实践, 因为在不了解宏的情况下, 很难判断哪些变量可能会受到这种宏的影响。

建议尽可能使用函数, 可以避免许多令人头疼的事情。

函数

将指令声明为函数, 可以使用以下语法:

```
function (<name> [<argument>⋯⋯])
  <commands>
endfunction ()
```

函数需要一个名称，并可选地接受预期参数的名称列表。

若函数调用传递的参数多于声明的参数，多余的参数将解释为匿名参数并存储在 ARGN 变量中。

如前所述，函数开放自己的作用域。可以调用 set()，提供函数的一个命名参数，任何更改都将是函数的局部更改 (除非指定了 PARENT_SCOPE)。

函数遵循调用堆栈的规则，允许使用 return() 返回调用作用域。

CMake 为每个函数设置以下变量 (从 3.17 版本开始提供):

- CMAKE_CURRENT_FUNCTION
- CMAKE_CURRENT_FUNCTION_LIST_DIR
- CMAKE_CURRENT_FUNCTION_LIST_FILE
- CMAKE_CURRENT_FUNCTION_LIST_LINE

来看看这些函数变量在实践中的情况:

```
# chapter02/08-definitions/function.cmake

function(MyFunction FirstArg)
    message("Function: ${CMAKE_CURRENT_FUNCTION}")
    message("File: ${CMAKE_CURRENT_FUNCTION_LIST_FILE}")
    message("FirstArg: ${FirstArg}")
    set(FirstArg "new value")
    message("FirstArg again: ${FirstArg}")
    message("ARGV0: ${ARGV0} ARGV1: ${ARGV1} ARGC: ${ARGC}")
endfunction()

set(FirstArg "first value")
MyFunction("Value1" "Value2")
message("FirstArg in global scope: ${FirstArg}")
```

输出如下:

```
Function: MyFunction
File: /root/examples/chapter02/08-definitions/function.cmake
FirstArg: Value1
FirstArg again: new value
ARGV0: Value1 ARGV1: Value2 ARGC: 2
FirstArg in global scope: first value
```

函数的一般语法和概念非常类似于宏，但这一次是有效的。

CMake 中的过程范式

想象一下，我们用 C++ 编写程序的相同方式编写一些 CMake 代码。将创建一个 CMakeLists.txt 列表文件，将调用三个定义的指令，这些指令可能调用它们自己定义的指令：

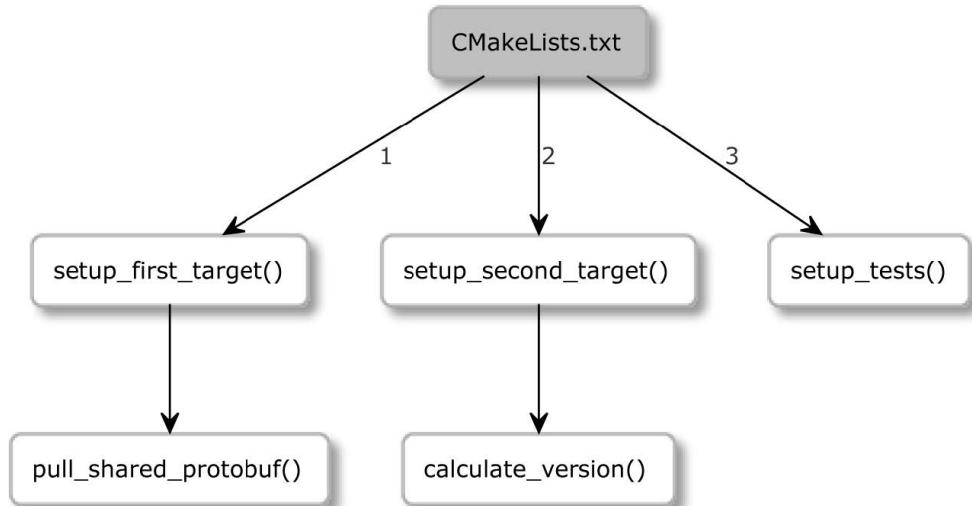


图 2.3 调用图

CMake 中，用这种过程风格编写代码有点麻烦——需要提前提供计划使用的命令定义：

```
cmake_minimum_required(...)
project(Procedural)
function(pull_shared_protobuf)
function(setup_first_target)
function(calculate_version)
function(setup_second_target)
function(setup_tests)

setup_first_target()
setup_second_target()
setup_tests()
```

真是恶梦一场！这段代码非常难以阅读，因为最微小的细节都在文件的顶部。一段结构正确的代码在第一个子例程中列出最通用的步骤，之后提供稍微详细一些的子例程，并将最详细的步骤推到文件的最后。

对于这个问题有一些解决方案：将命令定义移动到其他文件和跨目录分区作用域（作用域目录将在第 3 章中详细解释）。但也有一个简单而优雅的解决方案：在文件的顶部声明一个入口点宏，并在文件的最后调用它：

```
macro(main)
function(... # key steps
function(... # details
function(... # fine details
main()
```

通过这种方法，代码的编写范围逐渐缩小，因为直到最后我们才真正调用 `main()` 宏，CMake 不会抱怨未定义命令的执行！

最后一个问题是——为什么在推荐的函数上使用宏？可以无限制地访问全局变量，因为没有向 `main()` 传递参数，所以通常不需要担心这里的警告。

可以在本书的 GitHub 存储库中的 `chapter-02/09-procedural/CMakeLists.txt` 列表文件中找到这个简单例子。

关于命名约定

软件开发中，命名非常困难，但维护易于阅读和理解的解决方案非常重要。当涉及到 CMake 脚本和项目时，应该遵循干净代码方法的规则，就像对待软件开发解决方案一样：

- 遵循一致的命名风格 (`snake_case` 是 CMake 社区接受的标准)。
- 使用简短但有意义的名称（例如，避免 `func()`、`f()` 等）。
- 避免双关语和难懂的命名。
- 使用容易发音、容易搜索的名字，不需要人为映射。

现在已经知道了如何使用正确的语法正确地调用指令，让我们从使用指令开始研究。

2.6. 实用指令

CMake 提供了许多脚本命令，允许使用变量和环境。其中一些在附录部分有详细介绍，例如 `list()`、`string()` 和 `file()`。其他的，比如 `find_…()`，更适合在讨论依赖性管理的章节使用。

2.6.1 message() 指令

我们已经了解 `message()` 指令，将文本打印到标准输出，这其中的奥秘远比我们看到的要多得多。通过提供 `MODE` 参数，可以自定义输出的样式，并且在出现错误的情况下，可以停止代码：`:message(<mode> ”text”)` 的执行。

可选的模式如下：

- `FATAL_ERROR`: 将停止处理和生成。
- `SEND_ERROR`: 将继续处理，但跳过生成。
- `WARNING`: 继续处理。
- `AUTHOR_WARNING`: CMake 警告。继续处理。
- `DEPRECATION`: 若启用了 `CMAKE_ERROR_DEPRECATED` 或 `CMAKE_WARN_DEPRECATED` 变量，将做出相应处理。
- `NOTICE` 或省略模式（默认）: 将向 `stderr` 输出一条消息，以吸引用户的注意。
- `STATUS`: 将继续处理，建议用于用户的主要消息。
- `VERBOSE`: 将继续处理，用于通常不是很有必要的更详细的信息。
- `DEBUG`: 将继续处理，并包含在项目出现问题时可能有用的详细信息。
- `TRACE`: 将继续处理，并建议在项目开发期间打印消息。通常，在发布项目之前，将这些类型的消息删除。

下面的例子在第一条消息之后停止执行:

```
# chapter02/10-useful/message_error.cmake

message(FATAL_ERROR "Stop processing")
message("Won't print this.")
```

根据当前日志级别(默认为 STATUS)打印消息, 在前一章的调试和跟踪选项部分讨论了如何更改这一点, 这里就使用 CMAKE_MESSAGE_CONTEXT 进行调试。我们已经了解了调试的三个重要部分: 列表、作用域和函数。

当启用命令行标志 cmake --log-context 时, 消息将装饰为点分隔的上下文, 并存储在 CMAKE_MESSAGE_CONTEXT 列表中。考虑下面的例子:

```
# chapter02/10-useful/message_context.cmake

function(foo)
    list(APPEND CMAKE_MESSAGE_CONTEXT "foo")
    message("foo message")
endfunction()

list(APPEND CMAKE_MESSAGE_CONTEXT "top")
message("Before `foo`")
foo()
message("After `foo`")
```

前面脚本的输出如下所示:

```
$ cmake -P message_context.cmake --log-context
[top] Before `foo`
[top.foo] foo message
[top] After `foo`
```

函数的初始作用域是从父作用域复制的, foo 中的第一个命令在 CMAKE_MESSAGE_CONTEXT 中添加了一个 foo 函数名的新项。打印消息, 函数作用域结束, 丢弃本地复制的变量, 并恢复之前的作用域(不含 foo)。

这种方法对于复杂的目录中的嵌套函数非常有用。希望永远都不需要它, 但这的确是一个好例子, 说明了函数作用域在实践中是如何工作的。

message() 的另一个很酷的技巧是在 CMAKE_MESSAGE_INDENT 列表中添加缩进(与 CMAKE_MESSAGE_CONTEXT 的方法完全相同):

```
list(APPEND CMAKE_MESSAGE_INDENT " ")
```

这样脚本的输出看起来就更清晰了:

```
Before `foo`  
foo message  
After `foo`
```

由于 CMake 没有提供带有断点或其他工具的真正调试器，当事情没有完全按计划进行时，干净的日志就非常重要了。

2.6.2 include() 指令

可以将 CMake 代码划分到单独的文件中，以保持内容的有序和独立性。然后，可以通过 include() 从父列表文件引用：

```
include(<file>|module> [OPTIONAL] [RESULT_VARIABLE <var>])
```

若提供文件名（一个扩展名为.cmake），CMake 将尝试打开并执行它。这里不会创建嵌套的、单独的作用域，因此对该文件中变量的修改会影响调用作用域。

若文件不存在，CMake 将抛出一个错误，除非用 optional 关键字指定为可选。若需要知道 include() 是否成功，可以提供一个带有变量名的 RESULT_VARIABLE 关键字。若成功，则用包含的文件的完整路径填充，失败则用未找到(NOTFOUND) 填充。

脚本模式下运行时，将从当前工作目录解析相对路径。要强制搜索与脚本本身相关的内容，请提供绝对路径：

```
include("${CMAKE_CURRENT_LIST_DIR}/<filename>.cmake")
```

若不提供路径，但提供了模块的名称（没有.cmake 或其他），CMake 将尝试找到一个模块并包含它。然后，CMake 将在 CMake 模块目录 CMAKE_MODULE_PATH 中，搜索名称为<module>.cmake 的文件。

2.6.3 include_guard() 指令

包含有副作用的文件时，可能想要限制它们，以便它们只包含一次。这就是 include_guard([DIRECTORY|GLOBAL]) 的作用。

将 include_guard() 放在包含的文件的顶部。当 CMake 第一次遇到它时，将在当前作用域中进行记录。若文件再次包含（可能是因为没有控制项目中的所有文件），将不会处理。

若想要防止在不相关的函数作用域中包含不会彼此共享变量的函数，应该提供 DIRECTORY 或 GLOBAL 参数。顾名思义，DIRECTORY 关键字将在当前目录及其以下应用保护，而 GLOBAL 关键字将对整个构建应用保护。

2.6.4 file() 指令

为了了解 CMake 脚本可以做什么，快速的浏览一下文件操作命令：

```
file(READ <filename> <out-var> [...])
```

```
file({WRITE | APPEND} <filename> <content>...)
file(DOWNLOAD <url> [<file>] [...])
```

简而言之，`file()` 指令会以一种与系统无关的方式读取、写入和传输文件，并使用文件系统、文件锁、路径和存档。详情请参阅附录部分。

2.6.5 `execute_process()` 指令

有时需要使用系统中可用的工具(毕竟，CMake 主要是一个构建系统生成器)，CMake 为此提供了指令:`execute_process()` 可以用来运行其他进程，并收集它们的输出。这个命令非常适合脚本，也可以在配置阶段的项目中使用。下面是命令的一般形式：

```
execute_process(COMMAND <cmd1> [<arguments>] ... [OPTIONS])
```

CMake 将使用操作系统的 API 来创建子进程(因此，诸如 `&&`、`||` 和 `>` 等 shell 操作符将不起作用)。可以通过不止一次地提供 `COMMAND <cmd> <arguments>` 参数来连接命令，并将一个命令的输出传递给另一个命令。

若进程没有在要求的限制内完成任务，可以选择使用 `TIMEOUT <seconds>` 参数来终止进程，并且可以根据需要设置 `WORKING_DIRECTORY <directory>`。

通过 `RESULTS_VARIABLE <variable>` 参数，可以在列表中收集所有任务的退出代码。若只对最后执行命令的结果感兴趣，请使用单数形式:`RESULT_VARIABLE <variable>`。

为了收集输出，CMake 提供了两个参数:`OUTPUT_VARIABLE` 和 `ERROR_VARIABLE`(以类似的方式使用)。若想合并 `stdout` 和 `stderr`，请对两个参数使用相同的变量。

记住，在为其他用户编写项目时，应该确保命令在相应的平台上可用。

2.7. 总结

本章开启了使用 CMake 进行实际编程的大门——现在能够编写出色的、信息丰富的注释和调用内置指令，并且了解如何正确地为它们提供各种参数。仅这些知识就可以帮助您理解在其他项目中可能见过的 CMake 列表文件的不寻常语法。

接下来，介绍了 CMake 中的变量——如何引用、设置和取消设置普通变量、缓存变量和环境变量。我们深入研究了目录和函数作用域的工作方式，并讨论了与嵌套作用域相关的问题(及其解决方法)。

还讨论了列表和控制结构，讨论了条件的语法、逻辑操作、无引号求值，以及字符串和变量。学习了如何比较值，进行简单的检查，以及检查系统中文件的状态。这样就编写条件块和 `while` 循环。当我们讨论循环的时候，也掌握了 `foreach` 循环的语法。

我相信，知道如何用宏和函数语句定义自己的命令，将有助于您以更过程化的风格编写更清晰的代码。我们还分享了一些关于如何更好地构造我们的代码，并提出更可读的名称的想法。

最后，正式介绍了 `message()`，及其多个日志级别。还研究了如何划分和包含列表文件，并发现了一些其他有用的命令。了解了这些，就可以进入下一章了，在 CMake 中编写我们的第一个项目。

2.8. 扩展阅读

有关本章所涵盖主题的更多信息，请参阅以下网页：

- 代码整洁之道: 敏捷软件工艺手册 (Robert C. Martin): <https://amzn.to/3cm69DD>
- 重构: 改进现有代码的设计 (Martin Fowler): <https://amzn.to/3cmWk8o>
- 代码中的好注释 (Rafał Świdzinski): <https://youtu.be/4t9bpo0THb8>
- 设置和使用变量的 CMake 语法是什么?(StackOverflow): <https://stackoverflow.com/questions/31037882/whats-the-cmakesyntax-to-set-and-use-variables>

第 3 章 CMake 项目

现在开始讨论 CMake 的核心功能: 构建项目。CMake 中, 一个项目包含所有必要的源文件和配置, 以管理将解决方案变为现实的过程。

配置从执行所有检查开始: 是否支持目标平台, 是否具有所有必要的依赖项和工具, 所提供的编译器是否工作并支持所需的特性。

完成后, CMake 将选择的构建工具生成一个构建系统并运行它。源文件将编译并链接, 以产生输出工件。

开发者可以在内部使用项目来生成包, 用户可以通过包管理器将这些包安装到他们的系统上, 也可以使用提供的单可执行安装程序。项目还可以在开源存储库中共享, 这样用户就可以使用 CMake 在自己的机器上编译项目并安装。

充分利用 CMake 项目的潜力将改善开发体验和生成代码的质量, 可以自动化许多枯燥的任务, 例如在构建后运行测试, 检查代码覆盖率, 格式化代码, 以及使用 Linter 和其他工具检查源码。

为了解锁 CMake 项目的强大功能, 先讨论一些关键的决策——这些是如何将项目作为一个整体正确配置, 如何对其进行分区和设置源树, 以便所有文件都整齐地组织在正确的目录中。

然后学习如何查询构建项目的环境——例如, 它是什么架构? 可用的工具是什么? 支持哪些特性? 使用的是什么标准的语言? 最后, 将学习如何编译一个测试 C++ 文件, 以验证所选编译器是否满足项目的需求标准。

本章中, 我们将讨论以下主题:

- 指令和命令
- 划分项目
- 项目结构
- 环境范围
- 配置工具链
- 禁用内构建

3.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter03>。

构建本书中提供的示例, 推荐的命令:

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意: 构建树是目标/输出目录的路径, 源代码树是源码所在的路径。

3.2. 指令和命令

第1章中，已经了解了简单的项目定义。它是一个包含 CMakeLists.txt 文件的目录，其中包含一些配置语言处理器的命令：

```
# chapter01/01-hello/CMakeLists.txt: Hello world in CMake language

cmake_minimum_required(VERSION 3.20)
project(Hello)
add_executable(Hello hello.cpp)
```

3.2.1 指定最低的 CMake 版本——`cmake_minimum_required()`

这不是严格意义上特定于项目的指令，因为它也应该用于脚本。`cmake_minimum_required()` 将检查系统是否有正确的 cmake 版本，还将隐式调用另一个指令 `cmake_policy(version)`，告诉 cmake 该项目使用什么正确的策略。这些策略是什么？

在过去 20 年的 CMake 开发过程中，指令的行为方式发生了许多变化，CMake 支持的语言也在不断发展。为了保持语法的干净和简单，CMake 的团队决定引入一些策略来反映这些变化。每当引入一个向后不兼容的更改时，都会附带一个启用新行为的策略。

通过调用 `cmake_minimum_required()`，告诉 cmake 需要将策略应用到参数中提供的版本为止。当 CMake 使用新策略进行升级时，因为新策略不会启用，所以不需要担心破坏项目。用最新版本测试项目，若对结果满意，就可以将更新后的项目发送给用户了。

策略可以影响 CMake 的每一个方面，包括其他重要的指令，如 `project()`。因此，通过设置正在使用的版本来启动 CMakeLists.txt 文件是很重要的。否则，将得到警告和错误。

每个版本都引入了相当多的策略——描述它们没有实际价值，除非将遗留项目升级到最新的 CMake 版本时遇到了问题，请参考关于策略的官方文档获得更多信息：<https://cmake.org/cmake/help/latest/manual/cmake-policies.7.html>。

3.2.2 定义语言和元数据—`project()`

CMake 不需要 `project()`，包含 CMakeLists.txt 文件的目录都将在项目模式下进行解析。CMake 隐式地将该指令添加到文件的顶部。但我们需要从指定最低版本开始，所以最好不要忘记使用 `project()`。我们可以使用的两种形式之一：

```
project(<PROJECT-NAME> [<language-name>...])
project(<PROJECT-NAME>
[VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
[DESCRIPTION <project-description-string>]
[HOMEPAGE_URL <url-string>]
[LANGUAGES <language-name>...])
```

需要指定 `<PROJECT-NAME>`，其他参数可选。调用此指令将隐式设置以下变量：

- PROJECT_NAME
- CMAKE_PROJECT_NAME (只有在顶层 CMakeLists.txt 中)
- PROJECT_SOURCE_DIR, <PROJECT-NAME>_SOURCE_DIR
- PROJECT_BINARY_DIR, <PROJECT-NAME>_BINARY_DIR

支持哪些语言? 不少。这里有一个语言关键字列表, 可以用来配置项目:C, CXX(C++), CUDA, OBJC(Objective-C), OBJCXX (Objective-C++) , Fortran, ISPC, ASM, 以及 CSharp(C#) 和 Java。

CMake 默认启用 C 和 C++, 因此可能希望为 C++ 项目显式地仅指定 CXX。为什么? project() 将为所选语言检测和测试可用的编译器, 因此选择正确的编译器, 可以在配置阶段跳过对未使用语言的任何检查, 从而节省时间。

指定 VERSION 会自动定义以下变量:

- PROJECT_VERSION, <PROJECT-NAME>_VERSION
- CMAKE_PROJECT_NAME (只有在顶层 CMakeLists.txt 中)
- PROJECT_VERSION_MAJOR, <PROJECT-NAME>_VERSION_MAJOR
- PROJECT_VERSION_MINOR, <PROJECT-NAME>_VERSION_MINOR
- PROJECT_VERSION_PATCH, <PROJECT-NAME>_VERSION_PATCH
- PROJECT_VERSION_TWEAK, <PROJECT-NAME>_VERSION_TWEAK

上述变量对于配置包或传递到已编译文件, 在可执行文件中提供版本非常有用。可以以同样的方式设置 DESCRIPTION 和 HOMEPAGE_URL。

CMake 还可以使用 enable_language(<lang>) 来修改所使用的语言, 这将不会创建元数据。

上面的命令可以创建一个基本的列表文件, 并初始化一个空项目。现在, 可以开始添加要构建的东西。对于我们在示例中使用的小型单文件项目, 结构并不重要。但当代码更多时会发生什么呢?

3.3. 划分项目

随着解决方案的行数和文件数的增长, 慢慢地不可避免的事情即将到来: 要么开始对项目进行区分, 要么淹没在代码行和大量文件中。可以通过两种方式解决这个问题: 分配 CMake 代码和将源文件移动到子目录中。这两种情况下, 目标都是遵循称为关注点分离的设计原则, 就是将代码分成块, 将功能密切相关的代码分组, 同时将其他代码分离, 以创建强大的边界。

在第 1 章中讨论列表文件时, 讨论了一些关于分区 CMake 代码的内容。讨论了 include() 指令, 该指令允许 CMake 从外部文件执行代码。调用 include() 不会引入文件中没有定义的作用域(若所包含的文件包含函数, 则在调用时将正确处理它们的作用域)。

这种方法有助于分离关注点, 但是只有一些专门的代码提取到单独的文件中, 甚至可以在不相关的项目之间共享, 若作者不小心的话, 仍然会用其内部逻辑污染全局变量作用域。编程中有一个古老的真理: 最差的机制好过最好的意图。

稍后将学习如何处理这个问题, 现在让我们将注意力转移到源代码上。

假设有一个支持小型汽车租赁公司的软件示例——将有许多源文件定义软件的不同方面: 管理客户、汽车、停车位、长期合同、维护记录、员工记录等。若要把所有这些文件放在一个目录中, 查找文件肯定会是一场噩梦。因此, 在项目的主目录中创建了许多目录, 并将相关文件移动到其中。我们的 CMakeLists.txt 文件看起来类似这样:

```
# chapter03/01-partition/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Rental CXX)
add_executable(Rental
    main.cpp
    cars/car.cpp
    # more files in other directories
)
```

这一切都很好，但我们仍然在顶级文件中拥有来自嵌套目录的源文件列表！为了增加关注点的分离，可以将源列表放在另一个列表文件中，并使用前面提到的 `include()` 指令和 `cars_sources` 变量：

```
# chapter03/02-include/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Rental CXX)
include(cars/cars.cmake)
add_executable(Rental
    main.cpp
    ${cars_sources}
    # ${more_variables}
)
```

新的嵌套列表文件将包含源代码：

```
# chapter03/02-include/cars/cars.cmake

set(cars_sources
    cars/car.cpp
    # cars/car_maintenance.cpp
)
```

CMake 可以有效地将 `cars_sources` 设置在与 `add_executable` 相同的范围内，用所有文件填充变量。但这个解决方案有一些缺陷：

- 来自嵌套目录的变量会污染顶层作用域（反之亦然）：

在简单的示例中这不是一个问题，但在更复杂的、在流程中使用多个变量的多级树中，其很快就会成为一个难以调试的问题。

- 所有的目录将共享相同的配置：

随着项目多年来的成熟，这个问题会更加的明显。若没有粒度度的存在，必须将每个翻译单元视为相同的，并且不能指定不同的编译标志，不能为代码的某些部分选择较新的语言版本，

也不能在选定的代码区域设置静默警告。所有内容都是全局的，需要同时对所有源文件进行更改。

- 有一些共享编译触发器：

对配置的更改必须重新编译所有文件，即使更改对其中一些文件并没有意义。

- 所有的路径都相对于顶层：

注意 `cars.cmake` 中，必须提供 `cars/car.cpp` 文件的完整路径。这将导致大量重复的文本破坏可读性，并违背的不要自我重复 (DRY) 原则。并且，重命名一个目录会很困难。

另一种方法是使用 `add_subdirectory()` 指令，引入了一个变量范围等。

3.3.1 作用域的子目录

按照文件系统的自然结构来构造项目是一种常见的做法，其中嵌套的目录表示应用程序的离散元素：业务逻辑、GUI、API 和报告，最后，用测试、外部依赖项、脚本和文档分隔目录。为了支持这个概念，CMake 提供了以下指令：

```
add_subdirectory(source_dir [binary_dir]
[EXCLUDE_FROM_ALL])
```

这将向构建添加一个源目录，也可以提供一个写入构建文件的路径 (`binary_dir`)。`EXCLUDE_FROM_ALL` 关键字将禁用在子目录中定义的目标的默认构建 (将在下一章讨论目标)。这对于分离核心功能不需要的项目部分 (例如，示例和扩展) 可能很有用。

这个指令将在 `source_dir` 路径中查找 `CMakeLists.txt` 文件 (相对于当前目录求值)。这个文件将在目录作用域中进行解析，所以前面方法中提到的所有缺陷都不存在：

- 变量更改与嵌套作用域隔离。
- 可以随心所欲地配置嵌套工件。
- 更改嵌套的 `CMakeLists.txt` 文件中不需要构建和不相关的目标。
- 路径是目录的本地路径，若需要，可以添加到父 `include` 路径。

来看一个 `add_subdirectory()` 的项目：

```
chapter03/03-add_subdirectory# tree -A
.
├── CMakeLists.txt
└── cars
    ├── CMakeLists.txt
    ├── car.cpp
    └── car.h
└── main.cpp
```

这里，有两个 `CMakeLists.txt` 文件。顶层文件将使用嵌套目录 `cars`：

```
# chapter03/02-add_subdirectory/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.20.0)
project(Rental CXX)

add_executable(Rental main.cpp)

add_subdirectory(cars)
target_link_libraries(Rental PRIVATE cars)
```

最后一行用于将 cars 目录中的构件链接到 Rental 可执行文件。这是一个特定于目标的指令，我们将在下一章深入讨论。来看看嵌套的列表文件是什么样的：

```
#chapter03/02-add_subdirectory/cars/CMakeLists.txt

add_library(cars OBJECT
    car.cpp
# car_maintenance.cpp
)
target_include_directories(cars PUBLIC .)
```

使用 add_library() 生成了全局可见的目标 cars，并使用 target_include_directories() 将 cars 目录添加到其公共包括目录中。这允许 main.cpp 包含 cars.h 文件而不提供相对路径：

```
| #include "car.h"
```

可以在嵌套的列表文件中看到 add_library() 指令，示例中是否开始使用库了呢？事实上没有，因为我们使用了 OBJECT 关键字，这表示只对生成目标文件感兴趣（正如在前面的示例中所做的那样），只是将它们归类到一个逻辑目标（汽车）下。

3.3.2 嵌套项目

上一节中，简要地提到了 add_subdirectory() 中使用的 EXCLUDE_FROM_ALL 参数。CMake 文档建议，若在源码树中有这样的部件，应该在它们的 CMakeLists.txt 文件中有自己的 project() 指令，这样就可以生成自己的构建系统，并且可以独立构建。

其他情况下这是是否有用？当然。例如，正在处理在一个 CI/CD 流水中构建的多个 C++ 项目（构建一个框架或一组库时）。或者，也许正在从遗留解决方案中移植构建系统，例如使用普通 makefile 的 GNU Make，可能需要一个选项来将事情慢慢地分解为更独立的部分——将它们放在单独的构建流水中，或者只是在更小的范围上工作，可以由 CLion 等 IDE 加载。

可以通过向嵌套目录中的列表文件添加 project() 来实现，不要忘记在前面加上 cmake_minimum_required()。

既然支持项目嵌套，是否可以以某种方式将一起构建的相关项目连接起来？

3.3.3 外部项目

从一个项目到另一个项目在技术上存在，CMake 将在一定程度上支持这一点。甚至还有 `load_cache()` 指令允许从另一个项目的缓存中加载值。这不是一个常规的或推荐的用例，其将导致周期性依赖和项目耦合的问题，最好避免使用这个指令。所以，相关项目是应该嵌套，通过库连接，还是合并到单个项目中？

我们可以使用这些工具进行的分区：包括列表文件、添加子目录和嵌套项目。但应该如何使用它们，使我们的项目保持可维护性、易于导航和扩展呢？要做到这一点，需要一个定义良好的项目结构。

3.4. 项目结构

随着项目的发展，在列表文件和源代码中查找内容变得越来越困难。因此，从一开始就保持项目的健康非常重要。

想象一下，您需要交付一些重要的、时间敏感的更改，而它们在项目的两个目录中都不适合。现在，需要快速地提交一个清理提交，该提交为文件引入更多的目录和另一层层次结构，以便更改可以有一个合适的位置。或者（更糟糕的是），你决定把它们扔到其他地方，然后手写一张便签，稍后再处理这个问题。

随着时间的推移，这些便签会累积起来，技术债务也会增加，维护代码的成本也会增加。当运行的系统中存在需要快速修复的严重缺陷时，当不熟悉代码库的开发者需要更改时，这就变得非常棘手。

所以，一个好的项目结构意味着什么呢？我们可以从软件开发的其他领域（例如，系统设计）借鉴一些规则。

项目应具备以下特点：

- 易导航和扩展
- 自包含——例如，特定于项目的文件应该在项目目录中，而不是在其他目录中。
- 抽象层次结构应该通过可执行文件和二进制文件来表示。

这里没有统一的解决方案，但在网上有很多可用的项目结构模板，我建议采用这个模板，因为其简单且易扩展：

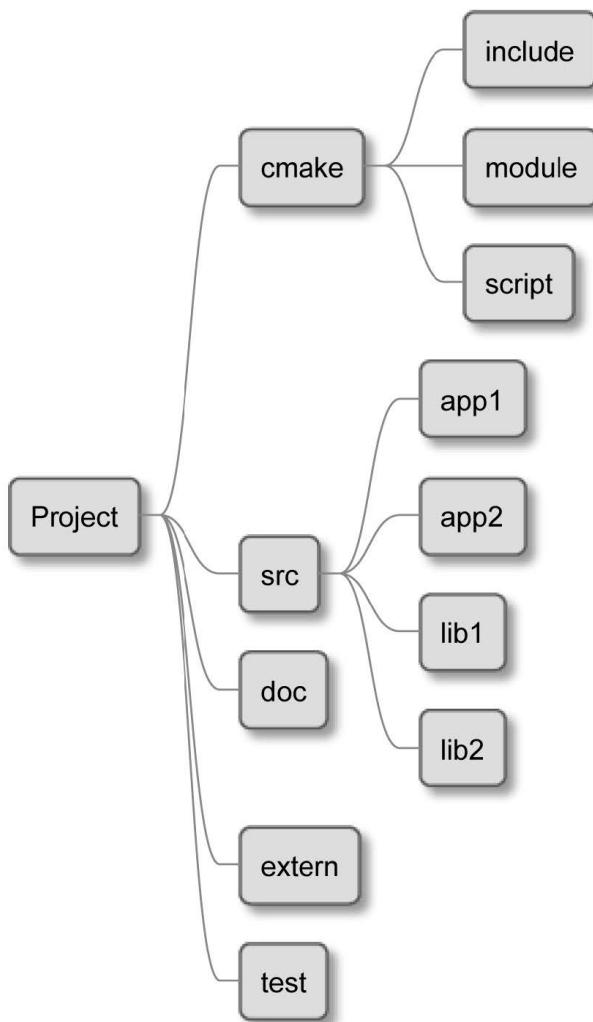


图 3.1 项目结构示例

本项目具有以下组件的目录:

- **cmake**: 宏和函数, 查找模块和一次性脚本
- **src**: 将存储的二进制文件和库的源代码
- **doc**: 用于构建文档
- **extern**: 从源代码构建的外部项目的配置
- **test**: 包含自动测试的代码

这个结构中, CMakeLists.txt 文件应该存在于以下目录中: 顶级项目目录、src、doc、extern 和 test。主列表文件不应该自己声明构建步骤, 应该使用 `add_subdirectory()` 指令来执行嵌套目录中的所有列表文件。反之, 可能会把这项工作委派给更深层。

Note

一些开发人员建议将可执行文件从库中分离出来, 创建两个顶级目录而不是一个:src 和 lib。CMake 对这两个工件一视同仁, 在这个级别上的分离并不重要。

src 目录中拥有多个目录对于更大的项目来说非常方便, 若只是构建一个可执行文件或库, 可以跳过它们, 直接将源文件存储在 src 中。因此, 需要在那里添加一个 CMakeLists.txt 文件, 并执行

嵌套的列表文件。

这是文件树寻找单个目标的方式:

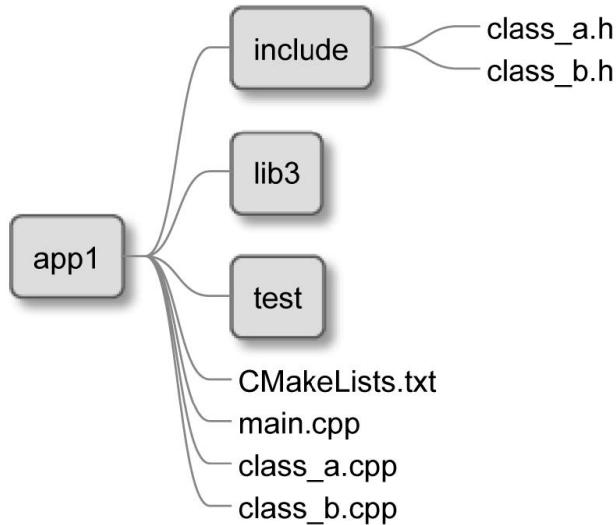


图 3.2 可执行文件的目录结构

在 app1 目录的根目录中看到一个 CMakeLists.txt 文件——配置关键项目设置，并包括来自嵌套目录的所有列表文件。src 目录包含另一个 CMakeLists.txt 文件和.cpp 实现文件: 两个类和带有可执行文件入口点的主文件。CMakeLists.txt 文件应该定义一个使用这些源文件构建可执行文件的目标——将在下一章学习如何做到这一点。

头文件在 include 目录中——.cpp 实现文件使用这些文件来声明来自其他 C++ 编译单元的符号。

我们有一个 test 目录来存放自动化测试的源码，还有 lib3，只包含一个特定于这个可执行文件的库(在项目的其他地方使用的库或在它之外导出的库应该位于 src 目录中)。

这种结构非常有表现力，可以对项目进行许多扩展。随着类添加越来越多，可以很容易地将它们分组到库中，以加快编译过程。来看看库的目录结构:

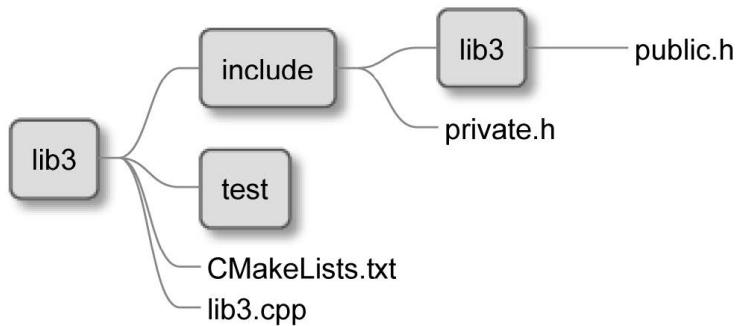


图 3.3 库的目录结构

库遵循与可执行文件相同的结构，只有很小的区别:include 目录中有一个可选的 lib3 目录。只有当从项目外部使用库时，才会出现这种情况，其提供了其他项目在编译期间将使用的公共头文件。将在第 5 章开始构建自己的库时回到这个主题。

因此，已经讨论了如何在目录结构中布局文件。现在，来看看各个 CMakeFiles.txt 文件是如何组合在一起形成单个项目的，以及其在更大的场景中扮演什么角色。

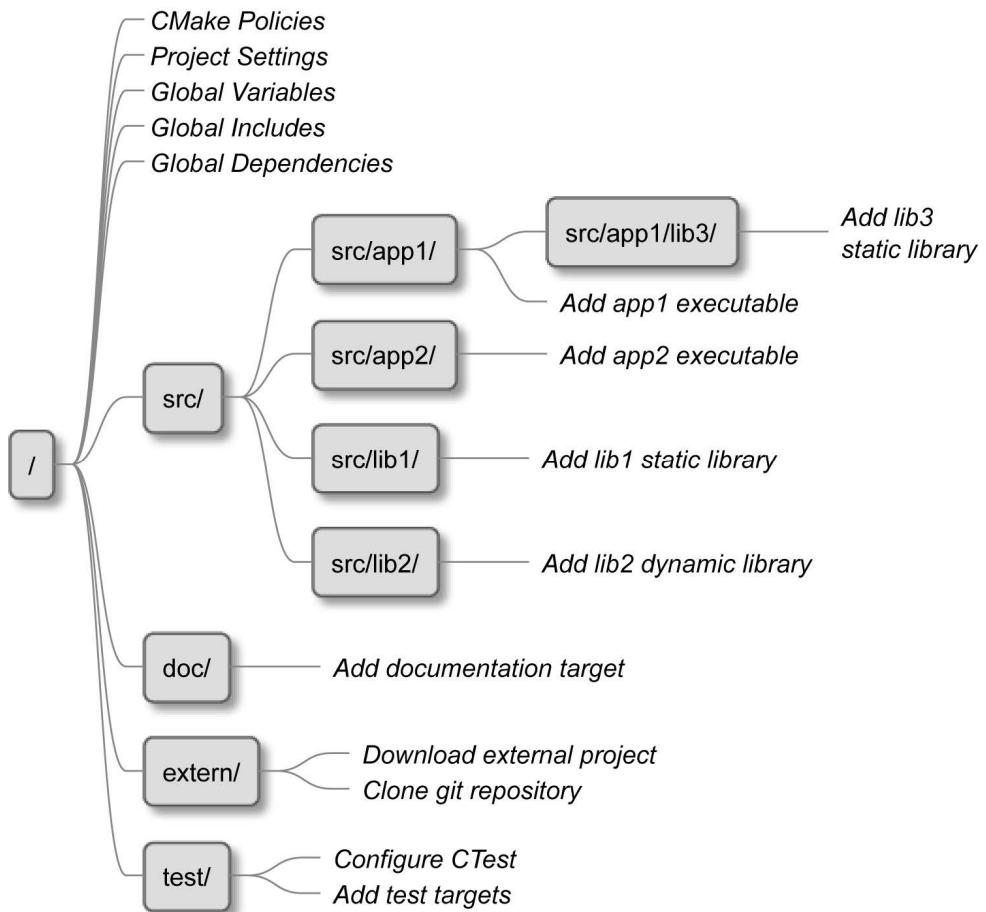


图 3.4 CMake 如何在单个项目中合并列表文件

图 3.4 中，每个框表示驻留在给定目录中的 CMakeLists.txt 列表文件，而文本中的标签表示每个文件执行的操作（从上到下）。让我们再从 CMake 的角度来分析一下这个项目：

1. 执行从项目的根开始——从源码树中的列表文件开始。该文件将使用适当的策略设置最低要求的 CMake 版本，设置项目名称、支持的语言、全局变量，并包括来自 CMake 目录的文件，以便内容全局可用。
2. 下一步是通过使用 `add_subdirectory(src bin)` 指令进入 `src` 目录的范围（将编译好的工件放在 `<binary_tree>/bin`，而不是 `<binary_tree>/src`）。
3. CMake 读取 `src/CMakeLists.txt` 文件，发现其唯一用途是添加四个嵌套子目录：`app1`、`app2`、`lib1` 和 `lib2`。
4. CMake 进入 `app1` 的变量作用域，了解另一个嵌套库 `lib3`，有自己的 CMakeLists.txt 文件；然后输入 `lib3` 的作用域。
5. `lib3` 库添加了一个具有相同名称的静态库目标。CMake 返回 `app1` 的父范围。
6. `app1` 子目录添加了一个依赖于 `lib3` 的可执行文件。CMake 返回 `src` 的父范围。
7. CMake 将继续进入剩下的嵌套作用域并执行它们的列表文件，直到完成所有 `add_subdirectory()`。
8. CMake 返回到顶级作用域并执行剩下的三个指令：`add_subdirectory(doc)`、`add_subdirectory(extern)` 和 `add_subdirectory(test)`。CMake 每次都进入新的作用域，并从适当的列表文件执行指令。

9. 收集所有的目标并检查其正确性，CMake 现在拥有生成构建系统所需的所有信息。

前面步骤的发生顺序与列表文件中写入命令的顺序完全一致。有时这很重要，而另一些时候则不是那么重要。我们将在下一章深入探讨。

那么，什么时候是创建目录以包含项目的所有元素的最佳时间呢？应该从一开始就做正确的事情——创建未来需要的所有东西，并保持目录为空——还是等到我们真正有了需要归入自己类别的文件？这是一种选择——可以遵循极限编程规则 YAGNI(不需要它)，或者可以尝试让项目具有前瞻性，为接纳新开发人员的到来奠定良好的基础。

试着在这些方法之间取得良好的平衡——若怀疑项目有一天可能需要 `extern` 目录，那么就添加它（可能需要创建一个空的`.keep` 文件来将目录检入存储库）。为了帮助其他人知道在哪里放置他们的外部依赖项，创建一个自述文件，并为未来将走上这条路的缺乏经验的程序员铺平道路。您可能已经注意到：开发人员不愿意创建目录，特别是在项目的根目录中。若提供一个好的项目结构，人们会倾向于遵循它。

有些项目可以在几乎所有的环境中构建，而另一些项目则对其细节非常挑剔。顶层列表文件是评估如何进行项目的最佳位置，这取决于可用的内容。

3.5. 环境范围

CMake 通过 `CMAKE_` 变量、`ENV` 变量和特殊命令提供了多种查询环境的方法。例如，收集的信息可以用于支持跨平台脚本。这些机制可避免使用特定于平台的 shell 命令，这些命令可能不容易移植，或者在不同的环境中命名不同。

对于性能关键型应用程序，了解目标平台的所有特性（例如，指令集、CPU 核心数等）将非常有用。然后，可以将此信息传递给已编译的二进制文件，以便对期进行优化（将在下一章学习如何做到这一点）。先来看看原生 CMake 中有哪些信息可用。

3.5.1 识别操作系统

很多情况下，了解目标操作系统是很有用的。即使是像文件系统这样普通的东西，Windows 和 Unix 在大小写敏感、文件路径结构、扩展名、特权等方面也有很大的不同。系统上的大多数命令在另一个系统上是不可用的，或者可以以不同的方式命名（即使是用一个字母命名——例如，`ifconfig` 和 `ipconfig` 命令）。

若需要用一个 CMake 脚本支持多个目标操作系统，只要检查 `CMAKE_SYSTEM_NAME` 变量，就可以采取相应的行动。这里有一个简单的例子：

```
if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
    message STATUS "Doing things the usual way"
elseif(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
    message STATUS "Thinking differently"
elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
    message STATUS "I'm supported here too."
elseif(CMAKE_SYSTEM_NAME STREQUAL "AIX")
    message STATUS "I buy mainframes."
```

```
else()
  message(STATUS "This is ${CMAKE_SYSTEM_NAME} speaking.")
endif()
```

有一个包含操作系统版本的变量:CMAKE_SYSTEM_VERSION。然而，建议是尽量使解决方案与系统无关，并使用内置的 CMake 跨平台功能。特别是对于文件系统上的操作，应该使用 file() 指令。

3.5.2 交叉编译——主机系统和目标系统?

在一台机器上编译要在另一台机器上运行的代码，称为交叉编译。可以(使用正确的工具集)通过在 Windows 机器上运行 CMake 来编译 Android 应用程序。交叉编译不在本书的讨论范围内，但是理解它如何影响 CMake 的某些部分是很重要的。

允许交叉编译的必要步骤之一是将 CMAKE_SYSTEM_NAME 和 CMAKE_SYSTEM_VERSION 变量设置为适合为目标编译的操作系统的值(CMAKE 文档将其称为目标系统)。用于执行构建的操作系统称为主机系统。

无论如何配置，主机系统上的信息总是可以在名称中带有 HOST 关键字的变量中访问:CMAKE_HOST_SYSTEM_NAME、CMAKE_HOST_SYSTEM_PROCESSOR 和 CMAKE_HOST_SYSTEM_VERSION。

还有一些变量的名称中带有 HOST 关键字，因此请记住它们显式引用主机系统。否则，所有变量都引用目标系统(通常是宿主系统，除非交叉编译)。

若有兴趣阅读更多关于交叉编译的内容，建议您参考 CMake 文档: <https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html>。

3.5.3 简化变量

CMake 将预定义一些变量，这些变量将提供关于主机和目标系统的信息。若使用特定的系统，则将适当的变量设置为非 false 值(即 1 或 true):

- ANDROID, APPLE, CYGWIN, UNIX, IOS, WIN32, WINCE, WINDOWS_PHONE
- CMAKE_HOST_APPLE, CMAKE_HOST_SOLARIS, CMAKE_HOST_UNIX,
- CMAKE_HOST_WIN32

对于 32 和 64 位版本的 Windows 和 MSYS，WIN32 和 CMAKE_HOST_WIN32 变量将为真(此值为遗留原因保留)。此外，UNIX 将适用于 Linux、macOS 和 Cygwin。

3.5.4 主机系统信息

CMake 可以提供更多变量，但为了节省时间，不会查询环境中很少需要的信息，比如处理器是否支持 MMX 或总物理内存是多少。这并不意味着这个信息不可用——只需要用下面的命令显式地进行:

```
cmake_host_system_information(RESULT <VARIABLE> QUERY <KEY>...)
```

需要提供一个目标变量和感兴趣的键的列表。若只提供一个键，则变量将只包含一个值；否则，它将是一个值列表。可以询问很多关于环境和操作系统的细节：

键值	返回值
HOSTNAME	主机名
FQDN	完全限定的 dimian 名称
TOTAL_VIRTUAL_MEMORY	总虚拟内存 (MiB)
AVAILABLE_VIRTUAL_MEMORY	可用的虚拟内存 (MiB)
TOTAL_PHYSICAL_MEMORY	总物理内存 (MiB)
AVAILABLE_PHYSICAL_MEMORY	可用的物理内存 (MiB)
OS_NAME	如果该命令存在，则 uname -s 的输出；若没有，则 uname -s 是 Windows、Linux 或 Darwin 中的一个
OS_RELEASE	OS 子类型，例如 Windows Professional
OS_VERSION	操作系统构建 ID
OS_PLATFORM	在 Windows 上，返回的是 PROCESSOR_ARCHITECTURE 环境变量。在 Unix 和 macOS 上，该值包含 uname -m 命令的输出。

可以查询特定于处理器的信息：

键值	返回值
NUMBER_OF_LOGICAL_CORES	逻辑核数
NUMBER_OF_PHYSICAL_CORES	物理核数
HAS_SERIAL_NUMBER	若处理器有序列号，则为 1
PROCESSOR_SERIAL_NUMBER	处理器序列号
PROCESSOR_NAME	可读的处理器名称
PROCESSOR_DESCRIPTION	可读的完整处理器描述
IS_64BIT	若处理器是 64 位，则为 1
HAS_FPU	处理器有浮点单元，则返回 1
HAS_MMX	若处理器支持 MMX 指令，则返回 1
HAS_MMX_PLUS	若处理器支持 Ext. MMX 指令，则返回 1
HAS_SSE	若果处理器支持 SSE 指令，则返回 1
HAS_SSE2	若处理器支持 SSE2 指令，则返回 1
HAS_SSE_FP	若处理器支持 SSE FP 指令，则返回 1
HAS_SSE_MMX	若处理器支持 SSE MMX 指令，则返回 1
HAS_AMD_3DNow	若处理器支持 3DNow 指令，则返回 1
HAS_AMD_3DNow_PLUS	若处理器支持 3DNow+ 指令，则返回 1
HAS_IA64	若是模拟 x86 的 IA-64 处理器，则返回 1

3.5.5 平台是 32 位还是 64 位架构?

64 位架构中，内存地址、处理器寄存器、处理器指令、地址总线和数据总线都是 64 位宽的。虽然这是一个简化的定义，但说明了 64 位平台与 32 位平台的不同之处。

C++ 中，不同的架构意味着一些基本数据类型 (int 和 long) 和指针的位宽不同。CMake 利用指针的大小来收集关于目标计算机的信息。这个信息可以通过 CMAKE_SIZEOF_VOID_P 变量获得，它将包含 64 位的值 8(因为指针是 8 字节宽的) 和 32 位的值 4(4 字节):

```
if(CMAKE_SIZEOF_VOID_P EQUAL 8)
  message(STATUS "Target is 64 bits")
endif()
```

3.5.6 系统的端序

架构可以是大端的，也可以是小端的。端序是字的字节顺序或处理器的自然数据单位。大端存储系统将最高位字节存储在最低内存地址，将最低位字节存储在最高内存地址。小端系统正好相反。

大多数情况下，字节顺序并不重要，但当编写需要可移植的位代码时，CMake 将为你提供一个 BIG_ENDIAN 或 LITTLE_ENDIAN 值存储在 CMAKE_<lang>_BYTE_ORDER 变量中，其中 <lang> 为 C、CXX、OBJC 或 CUDA。

知道了如何查询环境，现在让我们将注意力转移到项目的关键设置上。

3.6. 配置工具链

对于 CMake 项目，工具链包含构建和运行应用程序时使用的所有工具——例如，工作环境、生成器、CMake 可执行程序本身和编译器。

当构建因一些编译和语法错误而停止时，缺乏经验的用户会有什么感受。他们必须深入研究源代码，并试图理解发生了什么。经过一个小时的调试后，发现正确的解决方案是更新编译器。我们能否为用户提供更好的体验，并在开始构建之前检查编译器中是否存在所有所需的函数？

当然！有几种方法可以指定这些需求。若工具链不支持所有必需的特性，CMake 将提前停止并显示发生了什么事情的明确消息，要求用户介入。

3.6.1 设定 C++ 标准

若用户想要构建我们的项目，可能要做的第一件事是设置编译器需要支持的 C++ 标准。对于新项目，这应该至少是 C++14，但最好是 C++17 或 C++20。CMake 还支持将标准设置为实验性的 C++23，但这目前只是一个草案。

Note

从 C++11 正式发布到现在已经 10 年了，它不再认为是现代 C++ 标准。不建议使用此版本启动项目，除非目标环境非常旧。

坚持旧标准的另一个原因是，若正在构建难以升级的历史项目。然而，C++ 委员会非常努力地保持向后兼容。通常，升级标准不会遇到任何问题。

CMake 支持在每个目标的基础上设置标准，所以可以根据偏好的粒度使用。我认为最好在整个项目中采用单一标准。这可以通过设置 CMAKE_CXX_STANDARD 变量为以下值之一来实现：98、11、14、17、20 或 23（自 CMake 3.20 以来）。这将是所有随后定义的目标的默认值（因此最好将其设置在根列表文件顶部附近）。可以在每个目标的基础上覆写：

```
set_property(TARGET <target> PROPERTY CXX_STANDARD <standard>)
```

3.6.2 坚持支持标准

上一节中提到的 CXX_STANDARD 属性不会阻止 CMake 继续构建，即使编译器不支持所需的版本——视为首选项。CMake 不知道代码是否真的使用了以前编译器中没有的全新功能，它将尝试使用现有的功能。

若确定这不会成功，可以设置另一个默认标志（它可以以与前一个相同的方式被覆盖），并显式要求目标的标准：

```
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

若系统中没有最新的编译器（GNU GCC 11），用户只会看到以下消息，构建将停止：

```
Target "Standard" requires the language dialect "CXX23" (with
compiler extensions), but CMake does not know the compile flags
to use to enable it.
```

即使是在现代环境中，要求使用 C++23 也可能有点过分。但 C++14 应该没问题，因为自 2015 年以来 GCC/Clang 已经完全支持它了。

3.6.3 特定于供应商的扩展

根据组织中实现的策略，可能会对允许或禁用特定于供应商的扩展感兴趣。这些是什么？这么说吧，对于一些编译器生产者的需求，C++ 标准的发展有点慢，所以他们决定在语言中添加他们自己的增强——也可以叫插件。为了实现这一点，CMake 会将 -std=gnu++14，而不是 -std=c++14 添加到编译行。

一方面，因为它允许一些方便的功能，这可能是需要的。但另一方面，若切换到另一个编译器（或者您的用户切换到另一个编译器！），代码可能无法构建。

这也是一个每个目标的属性，它有一个默认变量 CMAKE_CXX_EXTENSIONS。CMake 在这里更加自由，并且允许扩展，除非我们告诉它不要这样做：

```
set(CMAKE_CXX_EXTENSIONS OFF)
```

若可能的话，我建议这样做，因为这个选项将坚持使用与供应商无关的代码。这样的代码不会对用户施加任何不必要的要求。以类似的方式，可以使用 set_property() 在每个目标的基础上更改这个值。

3.6.4 过程间优化

通常，编译器在单个翻译单元的级别上优化代码，所以.cpp 文件会进行预处理、编译，然后进行优化。稍后，这些文件将传递给链接器以构建一个二进制文件。现代编译器可以在链接之后执行优化(这称为链接时优化)，以便所有编译单元都可以作为单个模块进行优化。若编译器支持过程间优化，那么这可能是个好主意。

我们将遵循与前面相同的方法。需要设置的默认变量叫做 CMAKE_INTERPROCEDURAL_OPTIMIZATION。在设置它之前，需要确保支持，以避免错误：

```
include(CheckIPOSupported)
check_ipo_supported(RESULT ipo_supported)
if(ipo_supported)
    set(CMAKE_INTERPROCEDURAL_OPTIMIZATION True)
endif()
```

必须包含一个内置模块来访问 check_ipo_supported() 指令。

3.6.5 检查支持的编译器特性

若构建失败，最好尽早失败，这样就可以向用户提供明确的反馈消息。我们特别感兴趣的是衡量支持哪些 C++ 特性(哪些不支持)。CMake 会在配置阶段询问编译器，并在 CMAKE_CXX_COMPILE_FEATURES 变量中存储可用特性的列表。可以写一个非常具体的检查脚本，询问是否有某种特性可用：

```
# chapter03/07-features/CMakeLists.txt

list(FIND CMAKE_CXX_COMPILE_FEATURES
cxx_variable_templates result)
if(result EQUAL -1)
    message(FATAL_ERROR "I really need variable templates.")
endif()
```

使用的每个功能编写一个文档是一项艰巨的任务。甚至 CMake 的作者也建议只检查是否存在某些高级元特性:cxx_std_98、cxx_std_11、cxx_std_14、cxx_std_17、cxx_std_20 和 cxx_std_23，每个元特性都表明编译器支持特定的 C++ 标准。

CMake 功能的完整列表可以在文档中找到:https://cmake.org/cmake/help/latest/prop_gbl/CMAKE_CXX_KNOWN_FEATURES.html。

3.6.6 编译测试文件

用 GCC 4.7.x 编译一个应用程序时，我想到了一个特别有趣的场景。我已经在编译器的参考中手动确认，使用的所有 C++11 特性都得到了支持。然而，解决方案仍然不能正常工作。代码静静地忽略了对标准 <regex> 头文件。结果是 GCC 4.7.x 有一个 bug，正则表达式库没有实现。

没有检查可以保护您不受这种错误的影响，但是可以通过创建一个测试文件来减少这种行为，该文件可以填充您想要检查的所有特性。CMake 提供了两个配置时间命令,try_compile() 和 try_run(), 以验证目标平台上支持所需的一切。

第二个命令提供了更多的自由，因为可以确保代码不仅在编译，而且在正确地执行(可以测试 regex 是否在工作)。当然，这不适用于交叉编译场景(因为主机无法运行为不同目标构建的可执行文件)。这个检查的目的是在编译工作时向用户提供一个快速反馈，所以并不需要运行单元测试或其他复杂的东西。

像这样:

```
1 //chapter03/08-test_run/main.cpp
2
3 #include <iostream>
4 int main()
5 {
6     std::cout << "Quick check if things work." << std::endl;
7 }
```

使用 test_run() 一点也不复杂。首先设置所需的标准，然后调用 test_run() 并将收集到的信息打印给用户:

```
# chapter03/08-test_run/CMakeLists.txt

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

try_run(run_result compile_result
        ${CMAKE_BINARY_DIR}/test_output
        ${CMAKE_SOURCE_DIR}/main.cpp
        RUN_OUTPUT_VARIABLE output)

message("run_result: ${run_result}")
message("compile_result: ${compile_result}")
message("output:\n" ${output})
```

这个指令有很多可选的字段要设置，乍一看可能会让人不知所措，但当我们阅读并将其与示例中的调用进行比较时，就会发现所有内容都在一起:

```
try_run(<runResultVar> <compileResultVar>
        <bindir> <srcfile> [CMAKE_FLAGS <flags>...]
        [<COMPILE_DEFINITIONS> <defs>...]
        [<LINK_OPTIONS> <options>...]
        [<LINK_LIBRARIES> <libs>...]
        [<COMPILE_OUTPUT_VARIABLE> <var>]
```

```
[RUN_OUTPUT_VARIABLE <var>]
[OUTPUT_VARIABLE <var>]
[WORKING_DIRECTORY <var>]
[ARGS <args>...])
```

编译和运行一个非常基本的测试文件只需要几个字段，还使用可选的 RUN_OUTPUT_VARIABLE 关键字从 stdout 收集输出。

下一步是使用一些更现代的 C++ 特性来扩展这个简单的文件，将在整个实际项目中使用这些特性——可能是添加一个可变变量模板，看看目标机器上的编译器是否能够消化它。每次实际项目中引入一个新特性时，可以在测试文件中放入一个相同特性的小样本，确定检查编译是否在尽可能短的时间内工作。

最后，可以检入条件块，若收集的输出满足期望，并且当有些东西不正确时打印消息 (SEND_ERROR)。记住 SEND_ERROR 将继续通过配置阶段，但不会生成。这有助于在中止构建之前显示遇到的所有错误。

3.7. 禁用内构建

在第 1 章中，讨论了源代码内构建，以及如何建议始终将构建路径指定为源代码外的。这不仅可以具有更干净的构建树和更简单的.gitignore 文件，而且还减少了不小心覆盖或删除源码文件的可能。

在网上搜索解决方案时，可能会发现 StackOverflow 中有相同的问题:<https://stackoverflow.com/q/1208681/6659218>。作者注意到，无论做什么，CMake 似乎仍然会创建一个 CMakeFiles 目录和一个 CMakeCache.txt 文件。一些答案建议使用无文档变量，确保用户不能在源目录中写入：

```
# add this options before PROJECT keyword
set(CMAKE_DISABLE_SOURCE_CHANGES ON)
set(CMAKE_DISABLE_IN_SOURCE_BUILD ON)
```

我想说，在使用任何软件的无文档特性时都要谨慎，因为它们可能会在没有警告的情况下消失。CMake 3.20 中设置上述变量会终止构建并产生一个相当难看的错误：

```
CMake Error at /opt/cmake/share/cmake-3.20/Modules/
CMakeDetermineSystem.cmake:203 (file):
  file attempted to write a file:
  /root/examples/chapter03/09-in-source/CMakeFiles/CMakeOutput.
  log into a source directory.
```

然而，它仍然创建了上述文件！因此，建议是使用一个更老的——但得到充分支持的——机制：

```
# chapter03/09-in-source/CMakeLists.txt
```

```

cmake_minimum_required(VERSION 3.20.0)
project(NoInSource CXX)
if(PROJECT_SOURCE_DIR STREQUAL PROJECT_BINARY_DIR)
    message(FATAL_ERROR "In-source builds are not allowed")
endif()
message("Build successful!")

```

若 Kitware (CMake 背后的公司) 决定正式支持 CMAKE_DISABLE_SOURCE_CHANGES 或 CMAKE_DISABLE_IN_SOURCE_BUILD，那么可以切换到那种解决方案。

3.8. 总结

本章中介绍了许多概念，这将为我们前进和构建坚固的、经得起考验的项目提供坚实的基础。我们讨论了如何设置 CMake 的最低版本以及如何配置项目的关键方面——即名称、语言和元数据字段。

打下良好的基础将有助于确保我们的项目能够快速发展，这就是项目划分的原因。使用 `include()` 分析了原生代码分区，并将其与 `add_subdirectory()` 进行了比较。至此，了解了管理变量的目录作用域的好处，并探讨了使用更简单的路径和增加的模块化。当需要将代码分解成更独立的单元时，拥有创建嵌套项目并单独构建它的选项非常有用。

概述了可以使用的分区机制之后，探讨了如何使用——例如，如何创建透明的、弹性的和可扩展的项目结构。具体来说，分析了 CMake 如何遍历列表文件，以及不同配置步骤的正确顺序。

接下来，研究了如何确定目标计算机和主机计算机的环境范围、它们之间的区别，以及通过不同的查询可以获得关于平台和系统的哪些信息。

最后，了解了如何配置工具链——例如，如何指定所需的 C++ 版本，如何解决特定于供应商的编译器扩展问题，以及如何启用重要的优化。最后，发现了如何测试编译器所需的特性和编译测试文件。

虽然这是一个项目在技术上需要的全部内容，但它仍然不是一个非常有用的项目，我们需要引入目标。到目前为止，已经在这里或那里提到了它们，但在了解了一些一般概念之前，我试图避开这个主题。现在完成了这些铺垫，接下来将详细研究“目标”。

3.9. 扩展阅读

有关本章所涵盖主题的更多信息，请参阅以下连接：

- 关注点分离: <https://nalexn.github.io/separation-of-concerns/>
- 完整的 CMake 变量引用: <https://cmake.org/cmake/help/latest/manual/cmake-variables.7.html>
- 尝试编译和尝试运行描述:
 - https://cmake.org/cmake/help/latest/command/try_compile.html
 - https://cmake.org/cmake/help/latest/command/try_run.html

第二部分：进行构建

我们已经掌握了最基本的技能，是时候开始深入研究了。下一部分将解决在 CMake 中构建项目时遇到的大多数问题。

我们有意地专注于现代、优雅的实践，避免将太多的历史问题带入其中。具体来说，我们将处理逻辑构建目标，而不是操作单个文件。

接下来，将详细解释工具链从目标构建二进制工件所采取的所有步骤。这是许多关于 C++ 的书籍所遗漏的部分：如何正确配置和使用预处理器、编译器和链接器，以及如何优化其行为。

最后，将介绍 CMake 提供的管理依赖关系的所有不同方法，并将解释如何选择最佳方式。

- 第 4 章，使用目标
- 第 5 章，CMake 编译 C++
- 第 6 章，进行链接
- 第 7 章，管理依赖关系

第 4 章 使用目标

在 CMake 中最简单的目标是创建单个二进制可执行文件，其可由一段源码组成，例如经典的 `helloworld.cpp`。也可以是一些复杂的东西——由数百甚至数万个文件构建。这就是许多初级项目看起来的样子——用一个源文件创建一个二进制文件，添加另一个，然后，所有内容都链接到单个二进制文件。

作为软件开发者，我们故意划定边界并指定组件来分组一个或多个翻译单元 (`.cpp` 文件)。这样做的原因有很多：增加代码可读性，管理耦合和连接，加快构建过程，最后提取可重用组件。每个足够大的项目都需要引入某种形式的分区。

CMake 中的目标就是这个问题的解决方案——形成单一目标的逻辑单元。一个目标可能依赖于其他目标，并且可以以声明的方式产生的。CMake 负责确定构建目标的顺序，然后逐个执行必要的步骤。作为一般规则，构建目标将产生一个工件，该工件将作为其他目标的依赖，或作为构建的最终产品交付。

“工件”这个不准确的词，是因为 CMake 不限制只生成可执行文件或库。实际上，可以使用生成的构建系统来创建多种类型的输出：更多的源文件、头文件、目标文件、存档和配置文件——可以是任何东西。所需要的只是一个命令行工具（如编译器）、可选的输入文件和一个输出路径。

目标是一个非常强大的概念，极大地简化了项目的构建。理解其如何工作，以及如何以最优雅和干净的方式配置它们非常重要。

本章中，我们将讨论以下主题：

- 目标的概念
- 编写自定义命令
- 生成器表达式

4.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter04>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

4.2. 目标的概念

若曾经使用过 GNU Make，应该已经了解过目标的概念。本质上，它是构建系统用来将文件列表编译为另一个文件的一个方式。它可以是一个编译成 `.o` 对象文件的 `.cpp` 实现文件，一组打包成 `.o` 静态库的 `.o` 文件，以及许多其他组合。

然而, CMake 可以节省时间, 跳过这些食谱的中间步骤, 其可以在更高的抽象级别上工作。CMake 会理解如何直接从源文件构建可执行文件。因此, 不需要编写显式的配方来编译任何目标文件。所需要的只是一个 `add_executable()` 指令, 该指令带有可执行目标的名称和将成为其元素的文件列表:

```
add_executable(app1 a.cpp b.cpp c.cpp)
```

我们已经在前面的章节中使用了这个指令, 并且已经知道了在实践中如何使用可执行目标——生成步骤中, CMake 将创建一个构建系统, 并将其填充为方案编译每个源文件, 并将它们链接到单个二进制可执行文件中。

在 CMake 中, 可以使用以下指令创建目标:

- `add_executable()`
- `add_library()`
- `add_custom_target()`

前两个不言自明, 在前几章中已经简要地使用过它们来构建可执行程序和库(我们将在第 5 章中深入讨论)。但是这些定制目标是什么呢?

其允许你指定自己的命令行, 在不检查输出是否最新的情况下执行, 例如:

- 计算其他二进制文件的校验和。
- 运行代码消毒程序并收集结果。
- 向数据处理管道发送编译报告。

下面是 `add_custom_target()` 指令的完整签名:

```
add_custom_target(Name [ALL] [command1 [args1...]]  
                  [COMMAND command2 [args2...] ...]  
                  [DEPENDS depend depend depend ... ]  
                  [BYPRODUCTS [files...]]  
                  [WORKING_DIRECTORY dir]  
                  [COMMENT comment]  
                  [JOB_POOL job_pool]  
                  [VERBATIM] [USES_TERMINAL]  
                  [COMMAND_EXPAND_LISTS]  
                  [SOURCES src1 [src2...]])
```

不会在这里详述每个选项, 但定制目标不一定要以文件的形式产生工件。

定制目标的一个好的用例, 可能是需要在每个构建中删除特定的文件——例如, 以确保代码覆盖率报告不包含过时的数据。需要做的就是像这样定义一个自定义目标:

```
add_custom_target(clean_stale_coverage_files  
                  COMMAND find . -name "*.gcda" -type f -delete)
```

上面的命令将搜索所有扩展名为.gcda 的文件, 并删除它们。不过有一个问题; 与可执行目标和库目标不同, 自定义目标只有添加到依赖关系图中才会构建。

4.2.1 依赖图

成熟的应用通常是由许多组件构建，这里指的不是外部依赖关系。具体来说是内部库。从结构的角度来看，将它们添加到项目中是有用的，可以将相关的东西打包在一个逻辑实体中，还可以链接到其他目标——另一个库或可执行文件。当多个目标使用同一个库时，这尤其方便。请看图 4.1，其描述了一个示例依赖性图：

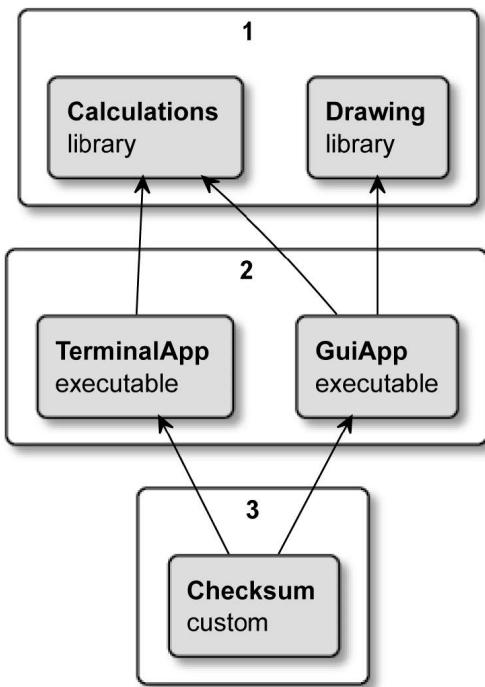


图 4.1 BankApp 项目中构建依赖的顺序

这个项目中，有两个库、两个可执行程序和一个自定义目标。这里的用例是为用户提供一个银行应用，该应用程序具有一个漂亮的 GUI(GuiApp)，以及一个命令行版本，可作为自动化脚本(TerminalApp)的一部分使用。两个可执行程序都依赖于相同的计算库，但只有一个需要绘图库。为了确保应用程序下载正确，将计算一个校验和，将其存储在一个文件中，并通过单独的安全通道分发它。CMake 在为这样的解决方案编写列表文件时非常灵活：

```
# chapter04/01-targets/CMakeLists.txt

cmake_minimum_required(VERSION 3.19.2)
project(BankApp CXX)

add_executable(terminal_app terminal_app.cpp)
add_executable(gui_app gui_app.cpp)

target_link_libraries(terminal_app calculations)
target_link_libraries(gui_app calculations drawing)

add_library(calculations calculations.cpp)
```

```
add_library(drawing drawing.cpp)

add_custom_target(checksum ALL
    COMMAND sh -c "cksum terminal_app>terminal.ck"
    COMMAND sh -c "cksum gui_app>gui.ck"
    BYPRODUCTS terminal.ck gui.ck
    COMMENT "Checking the sums..."
)
```

使用 `target_link_libraries()` 指令将库与可执行文件连接起来。若没有它，可执行文件的编译将会因为未定义的符号而失败。我们在实际声明库之前调用了这个命令。当 CMake 配置项目时，其会收集关于目标，及其属性的信息——名称、依赖项、源文件和其他信息。

解析所有文件之后，CMake 将尝试构建一个依赖图。和所有有效的依赖图一样，它们是有向无环图。有着一个明确的方向，确定哪个目标依赖于哪个目标，不过这样的依赖不能形成环。

当在构建模式下执行 `cmake` 时，生成的构建系统将检查已经定义的顶级目标并递归构建它们的依赖项。考虑一下图 4.1 中的例子：

1. 从顶部开始，构建第 1 组中的两个库。
2. 计算和绘图库完成后，构建组 2 - GuiApp 和 TerminalApp。
3. 构建一个校验和目标，运行指定的命令行生成校验和 (`cksum` 是一个 Unix 校验和工具)。

有一个小问题——前面的解决方案并不能保证在可执行文件之后构建校验和目标。CMake 不知道校验和依赖于当前的可执行二进制文件，所以可以自由地开始构建。为了解决这个问题，可以把 `add_dependencies()` 指令放在文件的末尾：

```
add_dependencies(checksum terminal_app gui_app)
```

这将确保 CMake 理解 Checksum 目标和可执行程序之间的关系。

不过 `target_link_libraries()` 和 `add_dependencies()` 之间有什么区别呢？第一个用于与实际库一起使用，并允许控制属性传播。第二种方法只能用于顶层目标，以设置它们的构建顺序。

随着项目变得越来越复杂，依赖树变得越来越难以理解。如何简化这个过程？

4.2.2 可视化的依赖性

即使是小项目也很有难和与其他开发人员直接分享。最简单的方法是通过一个漂亮的图表。毕竟，一张图片胜过千言万语。我们可以自己做这个工作并画一个图表，就像图 4.1 中所做的那样。但这很没意思，而且需要不断更新。幸运的是，CMake 有一个很好的模块来生成点/graphviz 格式的依赖关系图。它支持内部和外部依赖关系！

要使用它，可以简单地执行以下命令：

```
cmake --graphviz=test.dot .
```

该模块将生成一个文本文件，可以将该文本文件导入 Graphviz 可视化软件，该软件可以渲染图像或生成 PDF 或 SVG 文件，这些文件可以作为软件文档的一部分存储。每个人都喜欢好的文档，但是几乎没有人喜欢创建文档——现在，不需要自己来做了！

若赶时间，甚至可以直接从浏览器运行 Graphviz，地址如下：

<https://dreampuf.github.io/GraphvizOnline/>

重要的 Note

自定义目标在默认情况下是不可见的，需要创建一个特殊的配置文件 CMakeGraphVizOptions.cmake，可以自定义图形。设置了一个方便的定制命令 (GRAPHVIZ_CUSTOM_TARGETS TRUE)；将其添加到特殊的配置文件中，以支持报告图中的自定义目标。可以在该模块的文档中找到更多选项。

所需要做的就是复制和粘贴测试的内容。点文件到左边的窗口，项目将会可视化。很方便，不是吗？

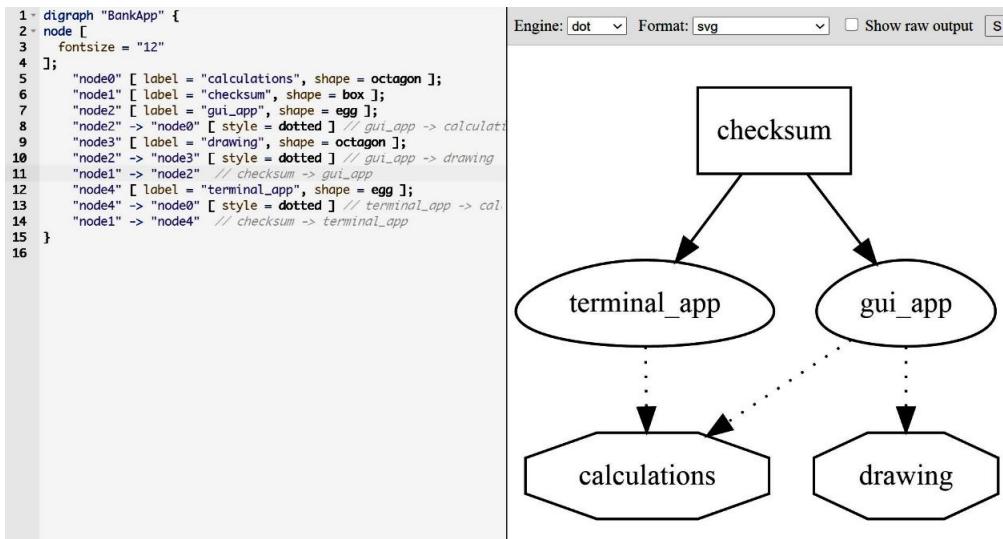


图 4.2 Graphviz 中 BankApp 示例的可视化

为了清晰起见，从上图中删除了自动生成的图例部分。

使用这种方法，可以快速地看到所有显式定义的目标。现在有了这个全局视角，让我们深入研究一下如何配置。

4.2.3 目标属性

目标具有类似于 C++ 对象字段的工作方式的属性。可以修改其中一些属性，而其他属性只读。CMake 定义了一个很大的“已知属性”列表（参见扩展阅读部分），这些属性取决于目标的类型（可执行、库或自定义），还可以添加自己的属性。使用以下命令操作目标器的属性：

```

get_target_property(<var> <target> <property-name>)
set_target_properties(<target1> <target2> ...
    PROPERTIES <prop1-name> <value1>
    <prop2-name> <value2> ...)

```

要在屏幕上打印目标属性，需要将其存储在 `<var>` 变量中，然后 `message()` 将其发送给用户，需要一个个地读取。另外，在目标上设置属性可以在多个目标上同时指定多个属性。

属性的概念并非目标所独有。`CMake` 还支持设置其他作用域的属性:GLOBAL、DIRECTORY、SOURCE、INSTALL、TEST 和 CACHE。

要操作所有类型的属性，有 `get_property()` 和 `set_property()` 指令。可以使用这些低层指令来做 `set_target_properties()` 指令做的事情:

```
set_property(TARGET <target> PROPERTY <name> <value>)
```

最好尽可能多地使用高级命令。`CMake` 提供了更多这样的功能，范围甚至更窄，比如在目标上设置特定的属性。例如，`add_dependencies(<target> <dep>)` 将依赖项附加到 `MANUALLY_ADDED_DEPENDENCIES` 目标属性中。可以使用 `get_target_property()` 查询，就像使用其他属性一样。然而，不能使用 `set_target_property()` 来更改 (只读)，`CMake` 中使用 `add_dependencies()` 指令只能进行追加操作。

接下来的章节中讨论编译和链接时，将介绍更多的属性设置命令。同时，我们会了解到一个目标的属性如何转换到另一个目标。

4.2.4 可传递需求

命名真的很难，有时会得到一个很难理解的结果。不幸的是，“传递性使用需求”是 `CMake` 文档中的神秘的主题之一。让我们来了解一下这个奇怪的名字，也许可以想出一个更容易理解的术语。

我将从解释这个谜题的中间部分开始，一个目标可能依赖于另一个目标。`CMake` 文档有时将这种依赖关系称为“使用”，例如在一个目标中使用另一个目标。这个很简单，我们继续下一个。

某些情况下，这样的已用目标具有使用目标必须满足的特定需求: 链接一些库，包含一个目录，或要求特定的编译特性。这些实际上都是需求，因此文档在某种意义上是正确的。问题是在文档的其他上下文中都不称其为需求。当为单个目标指定相同的需求时，可以设置属性或依赖项。因此，名称的最后一部分可能应该简单地称为“属性”。

最后一部分是传递，我相信这一点是正确的(也许有点太聪明了)。`CMake` 将使用目标的一些属性/需求附加到使用它们的目标的属性中。可以说某些属性可以隐式地跨目标转换(或简单地传播)，因此更容易表示依赖关系。

简化整个概念，将其视为源目标(使用的目标)和目标目标(使用其他目标的目标)之间的传播属性。

来看一个具体的例子，来理解其为什么会存在，以及如何工作的:

```
target_compile_definitions(<source> <INTERFACE|PUBLIC|PRIVATE>  
[items1...])
```

这个目标命令将填充 `<source>` 目标的 `COMPILE_DEFINITIONS` 属性。编译定义只是传递给配置 C++ 预处理器定义的编译器的-Dname= 定义标志(我们将在第 5 章中讨论)。有趣的是第二个论点，需要指定三个值 INTERFACE、PUBLIC 或 PRIVATE 中的一个，以控制应该将属性传递给哪个目标。现在，不要将这些与 C++ 访问说明符混淆。

传递关键字的工作原理如下:

- PRIVATE 设置源目标的属性。
- INTERFACE 设置相关目标的属性。
- PUBLIC 设置源和相关目标的属性。

当不将属性传递到其他目标时，可以将其设置为 PRIVATE。

当需要这样的传递时，使用 PUBLIC。若源目标在其实现 (.cpp 文件) 中不使用该属性，而只在头文件中使用该属性，这些属性会传递给消费者目标，那么就使用 INTERFACE。

这些是如何工作的？为了管理这些属性，CMake 提供了一些指令，比如前面提到的 target_compile_definitions()。当指定 PRIVATE 或 PUBLIC 关键字时，CMake 将在与指令匹配的目标的属性中存储提供的值——本例中是 COMPILE_DEFINITIONS。此外，若关键字是 INTERFACE 或 PUBLIC，将用 INTERFACE_ 前缀——INTERFACE_COMPILE_DEFINITIONS 将值存储在属性中。在配置阶段，CMake 将读取源目标的接口属性，并将其内容附加到目标目标。就传播属性，或传递了使用需求。

在 CMake 3.20 中，可以使用 target_link_options() 或直接使用 set_target_properties() 指令管理 12 个这样的属性：

- AUTOUIC_OPTIONS
- COMPILE_DEFINITIONS
- COMPILE_FEATURES
- COMPILE_OPTIONS
- INCLUDE_DIRECTORIES
- LINK_DEPENDS
- LINK_DIRECTORIES
- LINK_LIBRARIES
- LINK_OPTIONS
- POSITION_INDEPENDENT_CODE
- PRECOMPILE_HEADERS
- SOURCES

我们将在下面的页面中讨论这些选项中，但这些选项都在 CMake 手册中进行了描述。可以在 URL 下找到它们的页面（可以用感兴趣的属性替换 <PROPERTY>）：

https://cmake.org/cmake/help/latest/prop_tgt/<PROPERTY>.html

下一个问题是这种传播能有多远。属性是在第一个目标上设置的，还是发送到依赖关系图的最顶端？实际上，由使用者来决定。

要在目标之间创建依赖关系，可以使用 target_link_libraries() 指令。此指令的完整签名有一个 propagation 关键字：

```
target_link_libraries(<target>
    <PRIVATE | PUBLIC | INTERFACE> <item>...
    [<PRIVATE | PUBLIC | INTERFACE> <item>... ] ...)
```

此签名还指定了传播关键字，但此签名控制源目标的属性存储在目标目标中的何处。图 4.3 显示了在生成阶段（配置阶段完成后）传播属性的变化：

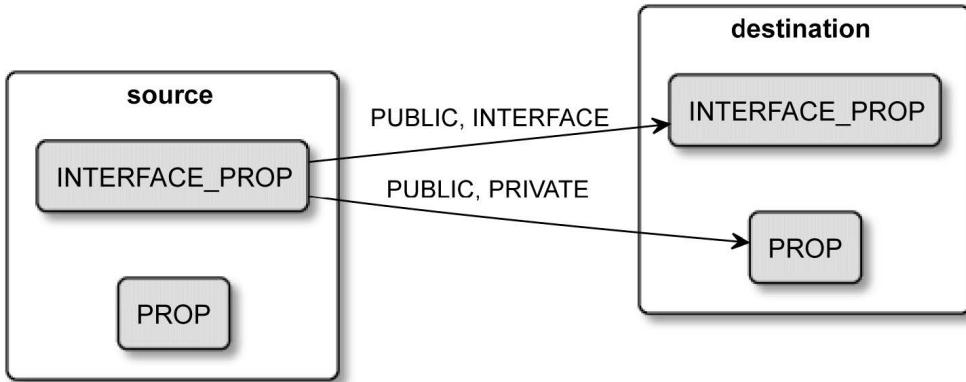


图 4.3 属性如何传播到目标

`propagation(传播)` 关键字的工作原理如下:

- PRIVATE 将源附加到目标的私有属性。
- INTERFACE 将源附加到目标的接口属性。
- PUBLIC 包括以上两种属性的特点。

接口属性仅用于在链中进一步传播属性，而目标不会在其构建过程中使用它们。

之前使用的基本 `target_link_libraries(<target> <item>...)` 指令隐式指定 PUBLIC 关键字。

若正确地为源目标设置了 `propagation(传播)` 关键字，属性将自动传递到目的目标上——除非有冲突……

4.2.5 处理冲突的传播属性

当目标依赖于多个其他目标时，可能会出现传播属性彼此完全冲突的情况。假设一个使用的目
标指定 `POSITION_INDEPENDENT_CODE` 属性为 True，另一个为 False。CMake 将此理解为冲突，并输出类似如下的错误：

```
CMake Error: The INTERFACE_POSITION_INDEPENDENT_CODE property
of "source_target2" does not agree with the value of POSITION_
INDEPENDENT_CODE already determined for "destination_target".
```

接收这样的消息很有用，因为明确地知道引入了这个冲突，并且需要解决它。CMake 有它自己的属性，这些属性必须在源目标和目标目标之间“一致”。

极少数情况下，这可能会变得很重要——例如，若在多个目标中使用相同的库构建软件，然后链接到单个可执行文件。若这些源目标使用同一个库的不同版本，可能会遇到问题。

为了确保只使用相同的特定版本，可以创建一个自定义接口属性 `INTERFACE_LIB_VERSION`，并将版本存储在那里。这还不足以解决问题，因为 CMake 在默认情况下不会传播自定义属性，必须显式地将自定义属性添加到“兼容”属性列表中。

每个目标都有 4 个这样的列表：

- `COMPATIBLE_INTERFACE_BOOL`
- `COMPATIBLE_INTERFACE_STRING`

- COMPATIBLE_INTERFACE_NUMBER_MAX
- COMPATIBLE_INTERFACE_NUMBER_MIN

将属性附加到其中一个，将触发传播和兼容性检查。BOOL 列表将检查传播到目标目标的所有属性是否计算为相同的布尔值。类似地，STRING 将求值为字符串。NUMBER_MAX 和 NUMBER_MIN 有点不同——传播值不需要匹配，但目标只会接收最高或最低的值。

这个例子将帮助我们理解如何在实践中应用：

```
# chapter04/02-propagated/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(PropagatedProperties CXX)

add_library(source1 empty.cpp)
set_property(TARGET source1 PROPERTY INTERFACE_LIB_VERSION
  4)
set_property(TARGET source1 APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING LIB_VERSION
)
add_library(source2 empty.cpp)

set_property(TARGET source2 PROPERTY INTERFACE_LIB_VERSION
  4)
add_library(destination empty.cpp)
target_link_libraries(destination source1 source2)
```

这里创建了三个目标，所有文件都使用相同的空源文件。两个源目标上，用 INTERFACE_ 前缀指定自定义属性。将它们设置为相同的匹配库版本。两个源目标都链接到目标目标。最后，指定一个 STRING 兼容性要求作为 source1 的属性 (没有在这里添加 INTERFACE_ 前缀)。

CMake 将这个自定义属性传播到目标目标，并检查所有源目标的版本是否完全匹配 (compatibility 属性可以只在一个目标上设置)。

既然了解了目标是什么，就看看其他看起来像目标，闻起来像目标，有时也像目标 (鸭子定律)。但事实证明，其并不是真正的目标。

4.2.6 实现伪目标

目标的概念非常有用，若其一些行为可以借鉴，那就太好了。具体来说，这是指不表示构建系统的输出而是表示输入的东西——外部依赖项、别名等。这些是伪目标，或者没有到达生成的构建系统的目标。

导入目标

若浏览了目录，就会知道将讨论 CMake 如何管理外部依赖项——其他项目、库等。导入的目标基本上是这个过程的产物。CMake 可以定义它们作为 `find_package()` 指令的结果。

可以调整这样一个目标的目标属性：编译定义、编译选项、包含目录等——甚至将支持可传递的使用需求。但应该将它们视为不可变的，不要改变它们的来源或依赖关系。

导入目标的范围可以是全局的，也可以是本地目录（在子目录中可见，但在父目录中不可见）。

别名目标

别名目标完全符合期望——不同的名称下创建对目标的另一个引用。可以使用以下指令为可执行文件和库创建别名目标：

```
add_executable(<name> ALIAS <target>
add_library(<name> ALIAS <target>)
```

别名目标的属性只读，并且不能安装或导出别名（在生成的构建系统中不可见）。

那么，使用别名的理由到底是什么呢？当项目的某些部分（如子目录）需要具有特定名称的目标时，就很方便了，而实际的实现可能根据情况在不同的名称下可用。例如，可能希望构建一个随解决方案一起提供的库，或者根据用户的选择导入。

接口库

这是一个有趣的构造——一个库不编译任何东西，而是作为一个实用工具目标。其整个概念是围绕传播属性（传递使用需求）构建的。

接口库有两个主要用途——纯头文件库和将一堆传播的属性捆绑到一个逻辑单元中。

使用 `add_library(INTERFACE)` 创建纯头文件库相当容易：

```
add_library(Eigen INTERFACE
src/eigen.h src/vector.h src/matrix.h
)
target_include_directories(Eigen INTERFACE
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src>
$<INSTALL_INTERFACE:include/Eigen>
)
```

前面的代码片段中，创建了一个具有三个头文件的特征接口库。接下来，使用生成器表达式，在导出目标时将其 `include` 目录设置为 `${CMAKE_CURRENT_SOURCE_DIR}/src`，在安装目标时设置为 `include/Eigen`。

要使用这样的库，只需要链接即可：

```
target_link_libraries(executable Eigen)
```

这里没有实际的链接，但是 CMake 将此命令理解为将所有 `INTERFACE` 属性传播到可执行目标的请求。

第二个用例使用了完全相同的机制，但目的不同——其创建了一个逻辑目标，可以作为传播属性的占位符。然后，可以将这个目标用作其他目标的依赖项，并以一种简洁、方便的方式设置属性。

这里有一个例子:

```
add_library(warning_props INTERFACE)
target_compile_options(warning_props INTERFACE
    -Wall -Wextra -Wpedantic
)
target_link_libraries(executable warning_props)
```

`add_library`(INTERFACE) 指令创建一个逻辑 `warning_props` 目标，用于在可执行目标上设置第二个命令中指定的编译选项。推荐使用这些 INTERFACE 目标，这提高了代码的可读性和可重用性。可以把它看作是将一堆神奇的值重构为一个命名良好的变量，还建议使用`_props` 后缀将接口库与常规库区分开来。

伪目标是否耗尽了目标的概念？当然不是！我们仍然需要理解，如何将这些目标转化为生成的构建系统。

4.2.7 构建目标

目标这个词有点夸张，在项目的上下文中和生成的构建系统的上下文中意味着不同的东西。当 CMake 生成一个构建系统时，会将 CMake 语言中的列表文件“编译”为所选构建工具的语言，也许为 GNU Make 创建了一个 Makefile。这样的 Makefile 有它们自己的目标——其中一些是列表文件目标的直接转换，另一些是隐式创建的。

这样的构建系统目标是 ALL，CMake 在默认情况下生成它来包含所有顶层列表文件目标，比如可执行文件和库（不一定是自定义目标）。ALL 是在运行 `cmake --build <build tree>` 而不选择具体目标时生成的，可以通过向前面的命令添加`--target <name>` 参数来选择一个。

有些可执行程序或库可能不是在每个构建中都需要，但希望将它们作为项目的一部分，以便在有用的时候使用。为了优化默认构建，可以像这样从 ALL 目标中排除它们：

```
add_executable(<name> EXCLUDE_FROM_ALL [<source>...])
add_library(<name> EXCLUDE_FROM_ALL [<source>...])
```

自定义目标则相反——默认情况下，排除在 ALL 目标之外，除非使用 ALL 关键字显式地定义它们，就像 BankApp 示例中那样。

另一个隐式定义的构建目标是干净的，只是从构建树中删除所生成的工件。用它来清除所有旧文件，从头开始构建一切。重要的是要理解它并不是简单地删除构建目录中的所有内容。所以要使 clean 正确工作，需要手动指定自定义目标可能创建为副产品的文件（请参阅 BankApp 示例）。

还有一种有趣的非目标机制可以创建可在所有实际目标中使用的自定义工件——自定义命令。

4.3. 编写自定义命令

使用自定义目标有一个缺点——添加到 ALL 目标或开始为其他目标依赖于它们，每次都进行构建（可以在 if 块中启用它们以限制）。有时，这是你想要的，但定制行为是必要的，以产生不应该无故重新创建的文件：

- 生成另一个目标所依赖的源代码文件

- 将另一种语言翻译成 C++
- 紧接在构建另一个目标之前或之后执行自定义操作

自定义命令有两个签名。第一个是 `add_custom_target()` 的扩展版本:

```
add_custom_command(OUTPUT output1 [output2 ...]
    COMMAND command1 [ARGS] [args1...]
    [COMMAND command2 [ARGS] [args2...] ...]
    [MAIN_DEPENDENCY depend]
    [DEPENDS [depends...]]
    [BYPRODUCTS [files...]]
    [IMPLICIT_DEPENDS <lang1> depend1
        [<lang2> depend2] ...]
    [WORKING_DIRECTORY dir]
    [COMMENT comment]
    [DEPFILE depfile]
    [JOB_POOL job_pool]
    [VERBATIM] [APPEND] [USES_TERMINAL]
    [COMMAND_EXPAND_LISTS])
```

自定义命令并不创建逻辑目标，但与自定义目标一样，其必须添加到依赖关系图中。有两种方法可以做到这一点——使用其输出工件作为可执行文件(或库)的源，或者显式地将其添加到自定义目标的 `DEPENDS` 列表中。

4.3.1 使用自定义命令作为生成器

诚然，不是每个项目都需要从其他文件生成 C++ 代码，也可能是通过 Google 的协议缓冲区(Protobuf).proto 文件的编译生成的。protobuf 是一个平台无关的结构化数据二进制序列化器。为了跨平台和执行效率，Google 工程师发明了他们自己的 protobuf 格式，可以在.proto 文件中定义模型，例如：

```
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
}
```

这样的文件可以跨多种语言共享——C++、Ruby、Go、Python、Java 等。Google 提供了读取.proto 文件并输出对所选语言有效的结构和序列化代码的编译器。聪明的工程师不会将这些编译好的文件检入存储库，而是会使用原始的 protobuf 格式，并将其添加到构建链中。

我们还不知道如何检测 protobuf 编译器在目标主机上是否可用(以及在哪里可用)。因此，现在假设编译器的 `protoc` 命令位于系统已知的位置。我们知道 protobuf 编译器将输出 `person.pb.h` 和 `person.pb.cc` 文件。下面是如何定义一个自定义命令进行编译：

```
add_custom_command(OUTPUT person.pb.h person.pb.cc
    COMMAND protoc ARGS person.proto
    DEPENDS person.proto
)
```

然后，为了在可执行文件中序列化，只需向源文件添加输出文件：

```
add_executable(serializer serializer.cpp person.pb.cc)
```

假设正确地处理了头文件的包含和 protobuf 库的链接，当对.proto 文件进行更改时，所有内容都将自动编译和更新。

一个简单的(而且不太实用的)例子是从另一个位置复制创建必要的头文件：

```
# chapter04/03-command/CMakeLists.txt

add_executable(main main.cpp constants.h)
target_include_directories(main PRIVATE
    ${CMAKE_BINARY_DIR})
add_custom_command(OUTPUT constants.h
    COMMAND cp
    ARGS "${CMAKE_SOURCE_DIR}/template.xyz" constants.h)
```

本例中，“编译器”是 cp 命令，通过在构建树根中创建 constants.h 文件来实现对主目标的依赖，只需从源树中复制即可。

4.3.2 使用自定义命令作为目标钩子

add_custom_command() 指令的第二个版本引入了在构建目标之前或之后执行命令的机制：

```
add_custom_command(TARGET <target>
    PRE_BUILD | PRE_LINK | POST_BUILD
    COMMAND command1 [ARGS] [args1...]
    [COMMAND command2 [ARGS] [args2...] ...]
    [BYPRODUCTS [files...]]
    [WORKING_DIRECTORY dir]
    [COMMENT comment]
    [VERBATIM] [USES_TERMINAL]
    [COMMAND_EXPAND_LISTS])
```

第一个参数中指定使用新行为“增强”的目标，并在以下条件下执行：

- PRE_BUILD 将在此目标的任何其他规则之前运行(仅 Visual Studio 生成器；对于其他的生成器，其行为像 PRE_LINK)。

- `PRE_LINK` 绑定要在编译所有源之后但在链接(或存档)目标之前运行的命令。不适用于自定义目标。
- `POST_BUILD` 将在为该目标执行所有其他规则之后运行。

使用这个版本的 `add_custom_command()`, 可以从前面的 BankApp 例子中复制校验和的生成方式:

```
# chapter04/04-command/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Command CXX)

add_executable(main main.cpp)
add_custom_command(TARGET main POST_BUILD
    COMMAND cksum
    ARGS "$<TARGET_FILE:main>" > "main.ck")
```

主可执行文件的构建完成后, CMake 将使用提供的参数执行 `cksum`。第一个参数中发生了什么? 它不是一个变量, 因为不是包装在花括号 (`{}$`) 中, 而是在尖括号 (`$<>`) 中, 而这是一个生成器表达式, 计算到目标二进制文件的完整路径。这种机制在许多目标属性的上下文中非常有用。

4.4. 生成器表达式

CMake 通过三个阶段构建解决方案——配置、生成和运行构建工具, 在配置阶段就有了所有所需的数据。但每隔一段时间, 就会遇到“先有鸡, 还是先有蛋”的问题。举一个前一节的例子——一个目标需要知道另一个目标的二进制工件的路径。但是, 只有在解析所有列表文件并完成配置阶段之后, 这些信息才可用。

该如何处理这种问题呢? 可以为该信息创建一个占位符, 并将其评估推迟到下一个阶段——生成阶段。

这就是生成器表达式 (有时称为 `genex`) 所做的事, 其是围绕目标属性构建的, 如 `LINK_LIBRARIES`、`INCLUDE_DIRECTORIES`、`COMPILE_DEFINITIONS`、传播属性和其他属性, 但不是全部。遵循类似条件语句和变量求值的规则。

值得注意的是, 表达式通常是在使用的目标上下文中求值的 (除非另有明确说明)。

重要的 Note

生成器表达式将在生成阶段计算 (当配置完成并创建构建系统时), 所以不能将其输出捕获到变量中, 并打印到控制台。要调试可以使用以下方法:

- 将其写入一个文件 (file() 指令的这个特定版本支持生成器表达式):

```
file(GENERATE OUTPUT filename CONTENT "$<...>")
```

- 添加一个自定义目标, 并根据命令行显式构建:

```
custom_target(gendbg COMMAND ${CMAKE_COMMAND} -E echo "$<...>")
```

4.4.1 一般语法

举一个最简单的例子:

```
target_compile_definitions(foo PUBLIC  
BAR=$<TARGET_FILE:foo>)
```

上面的命令在编译器的参数中添加了一个-D 定义标志 (暂时忽略 PUBLIC), 将 BAR 预处理器定义设置为 foo 目标的二进制工件的路径。

生成器表达式如何形成的?

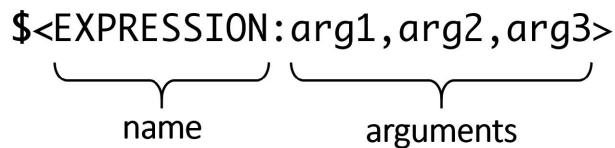


图 4.4 生成器表达式的语法

如图 4.4 所示, 该结构看起来相当简单且可读:

- 以 \$ 和括号 (\$<) 开头
- 添加 EXPRESSION 名称
- 若表达式需要参数, 则添加冒号 (:) 并提供 arg1、arg2 和 arg3 值, 用逗号分隔。
- 用 > 关闭表达式。

甚至有些表达式不需要任何参数, 例如 \$<PLATFORM_ID>。然而, 在使用生成器表达式更高级的特性时, 生成器表达式很快就会变得非常混乱和复杂。

嵌套

将通用表达式作为参数传递给另一个表达式的能力开始。换句话说, 就是通用表达式的嵌套:

```
$<UPPER_CASE:$<PLATFORM_ID>>
```

这不是一个非常复杂的示例, 但很容易想象当增加嵌套级别并使用使用多个参数的命令时会发什么。

若这还不够，可以在组合中添加一个变量展开：

```
$<UPPER_CASE:${my_variable}>
```

在配置阶段将展开一个变量，生成阶段将展开一个生成表达式。这个功能有一些罕见的用法，但不建议使用。

条件表达式

生成器表达式支持布尔逻辑，但是由于历史原因，它的语法不一致，很难阅读。它有两种形式。第一种形式支持成功和失败的路径：

```
$<IF:condition,true_string,false_string>
```

这里的语法与所有其他表达式对齐，并且与所有表达式一样，允许嵌套。因此，可以用另一个表达式替换任何参数，并生成一些非常复杂的计算——甚至可以将一个条件嵌套在另一个条件中。这种形式恰好需要三个参数，因此不能省略任何内容。未满足条件的情况下，跳过值的最佳选项：

```
$<IF:condition,true_string,>
```

第二种形式是前一种的缩写；只有满足以下条件，才会展开为字符串：

```
$<condition:true_string >
```

这打破了提供 EXPRESSION 名称作为第一个标记的惯例。这里的意图是缩短表达并跳过这三个宝贵的字符，但结果真的很难合理解释。下面是 CMake 文档中的一个例子：

```
$<$<AND:$<COMPILE_LANGUAGE:CXX>,$<CXX_COMPILER_ID:AppleClang>>>:COMPILING_CXX_WITH_CLANG>
```

希望语法与常规 if 指令的条件一样，但遗憾的是事实并非如此。

4.4.2 计算类型

生成器表达式可以计算成两种类型——布尔或字符串。布尔值由 1(真) 和 0(假) 表示。其他的都只是一个字符串。

条件表达式中作为条件传递的嵌套表达式，显式为布尔值。

显式逻辑运算符可以将字符串转换为布尔类型，但布尔类型可以隐式转换为字符串。

现在知道了基本的语法，来看看能用来做什么。

布尔型

在前一节开始讨论条件表达式。这里先把整个概念讲完，这样以后就不再回头讲了。有三种类型的表达式的值为布尔值。

逻辑运算符

逻辑运算符有 4 种：

- `$<NOT:arg>`

否定布尔参数。
 - `$<AND:arg1,arg2,arg3...>`

若所有参数都为 1，则返回 1。
 - `$<OR:arg1,arg2,arg3...>
returns 1 if any of the arguments is 1.`

若所有参数有 1，则返回 1。
 - `$<BOOL:string_arg>`

将参数从字符串转换为布尔类型。
- 若这些条件都不满足，字符串转换将计算为 1:
- 字符串为空。
 - 字符串不区分大小写，相当于 0、FALSE、OFF、N、NO、IGNORE 或 NOTFOUND。
 - 字符串以-NOTFOUND 后缀结尾(区分大小写)。

字符串比较

若满足条件比较的值为 1，否则为 0:

- `$<STREQUAL:arg1,arg2>`

是区分大小写的字符串比较。
- `$<EQUAL:arg1,arg2>`

将字符串转换为数字并比较是否相等。
- `$<IN_LIST:arg,list>`

检查 arg 元素是否在列表列表中(区分大小写)。
- `$<VERSION_EQUAL:v1,v2>, $<VERSION_LESS:v1,v2>,
$<VERSION_GREATER:v1,v2>, $<VERSION_LESS_EQUAL:v1,v2>,
$<VERSION_GREATER_EQUAL:v1,v2>`

版本的比较。

查询变量

有许多变量包含布尔类型的值。若满足条件，则求值为 1，否则为 0。

简单的查询:

- `$<TARGET_EXISTS:arg>`

arg 目标是否存在?

有多个查询扫描传递的参数以获取特定值:

- `$<CONFIG:args>`

是 args 中的当前配置 (Debug, Release 等)(不区分大小写)。

- `$<PLATFORM_ID:args>`

是 args 中的当前平台 ID。

- `$<LANG_COMPILER_ID:args>`

是 CMake 的 LANG 编译器 ID 在 args 中, 其中 LANG 是 C、CXX、CUDA、OBJC、OBJCXX、Fortran 或 ISPC 中的一种。

- `$<LANG_COMPILER_VERSION:args>`

是 CMake 的 LANG 编译器的 args 版本, 其中 LANG 是 C、CXX、CUDA、OBJC、OBJCXX、Fortran 或 ISPC 中的一种。

- `$<COMPILE_FEATURES:features>`

若编译器支持此目标的特性, 则返回 true。

- `$<COMPILE_LANG_AND_ID:lang,compiler_id1,compiler_id2...>`

这个 LANG 是目标的语言, 在 compiler_ids 列表中为这个目标使用的编译器。这个表达式对于指定特定编译器的配置很有用:

```
target_compile_definitions(myapp PRIVATE
    $<$<COMPILE_LANG_AND_ID:CXX,AppleClang,Clang>:CXX_CLANG>
    $<$<COMPILE_LANG_AND_ID:CXX,Intel>:CXX_INTEL>
    $<$<COMPILE_LANG_AND_ID:C,Clang>:C_CLANG>
)
```

在前面的例子中, 若用 AppleClang 或 Clang 编译 CXX 编译器, 则会设置-DCXX_CLANG 定义。对于来自 Intel 的 CXX 编译器, 将设置-DCXX_INTEL 标志。最后, 对于 C 和 Clang 编译器, 将得到-DC_CLANG 的定义。

- `$<COMPILE_LANGUAGE:args>`

若使用一种语言在 args 中编译此目标。可以用来为编译器提供特定于语言的标志:

```
target_compile_options(myapp
    PRIVATE $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>
)
```

- 若编译 CXX，编译器将使用-fno-exceptions 标志

```
$<LINK_LANG_AND_ID:lang,compiler_id1,compiler_id2...>
```

与 COMPILE_LANG_AND_ID 的工作原理类似，但会检查链接步骤使用的语言。使用此表达式指定特定语言的链接库、链接选项、链接目录和链接依赖项，以及目标中的链接器组合。

- \$<LINK_LANGUAGE:args>

args 中用于链接步骤的语言。

字符串求值

有很多表达式最后求值为字符串。可以直接输出到目标的占位符，也可以作为另一个表达式的参数使用。我们已经了解了单条件表达式的计算结果是字符串。还有什么可选的？

的查变量

表达式在生成阶段会求出特定的值：

- \$<CONFIG>

配置 (Debug 和 Release) 名称。

- \$<PLATFORM_ID>

当前系统的 CMake 平台 ID (Linux、Windows 或 Darwin)。

- \$<LANG_COMPILER_ID>

使用的 LANG 编译器的 CMake 编译器 ID，其中 LANG 是 C、CXX、CUDA、OBJC、OBJCXX、Fortran 或 ISPC 中的一种。

- \$<LANG_COMPILER_VERSION>

所使用的 LANG 编译器的 CMake 编译器版本，其中 LANG 是 C、CXX、CUDA、OBJC、OBJCXX、Fortran 或 ISPC 中的一种。

- \$<COMPILE_LANGUAGE>

计算编译选项时源文件的编译语言。

- \$<LINK_LANGUAGE>

计算链接选项时目标的链接语言。

查询依赖的目标

通过以下查询，可以计算可执行文件或库目标的属性。注意，从 CMake 3.19 开始，对于大多数表达式来说，在另一个目标的上下文中查询一个目标不再在这些目标之间创建自动依赖关系（就像在 3.19 之前那样）：

- `$<TARGET_NAME_IF_EXISTS:target>`
若目标存在，则为目标的目标名称；否则为空字符串。
- `$<TARGET_FILE:target>`
目标二进制文件的完整路径。
- `$<TARGET_FILE_NAME:target>`
目标文件名。
- `$<TARGET_FILE_BASE_NAME:target>`
目标的基名或
- `$<TARGET_FILE_NAME:target>`
没有前缀和后缀。比如：libmylib.so 的基名是 mylib。
- `$<TARGET_FILE_PREFIX:target>`
目标文件名 (lib) 的前缀。
- `$<TARGET_FILE_SUFFIX:target>`
目标文件名的后缀 (或扩展名)(.so, .exe)。
- `$<TARGET_FILE_DIR:target>`
目标二进制文件的目录
- `$<TARGET_LINKER_FILE:target>`
链接到目标目标时使用的文件。通常，目标表示的是库 (.a, .lib, .so) 在具有动态链接库 (DLL) 的平台上；对于动态库，将是.lib 导入库。
TARGET_LINKER_FILE 提供了与常规 TARGET_FILE 表达式相同的表达式群：
- `$<TARGET_LINKER_FILE_NAME:target>, $<TARGET_LINKER_FILE_BASE_NAME:target>, $<TARGET_LINKER_FILE_PREFIX:target>, $<TARGET_LINKER_FILE_SUFFIX:target>, $<TARGET_LINKER_FILE_DIR:target>`
- `$<TARGET SONAME FILE:target>`
带有 soname(.so.3) 的文件的完整路径。
- `$<TARGET SONAME FILE NAME:target>`
带有 soname 的文件名。

- `$<TARGET__SONAME__FILE__DIR:target>`

带有 soname 的文件的目录。

- `$<TARGET__PDB__FILE:target>`

链接器为目标生成的程序数据库文件 (.pdb) 的完整路径。

PDB 文件提供与常规 TARGET_FILE 相同的表达式:

```
$<TARGET__PDB__FILE__BASE__NAME:target>, $<TARGET__PDB__FILE__NAME:target>,  
$<TARGET__PDB__FILE__DIR:target>
```

- `$<TARGET__BUNDLE__DIR:target>`

BUNDLE(Apple 特定的包) 目录的完整路径 (my.app, my.framework 或 my.bundle)。

- `$<TARGET__BUNDLE__CONTENT__DIR:target>`

目标的包内容目录的完整路径。在 macOS 上, 它是 my.app/Contents, my.framework 或 my.bundle/Contents。其他软件开发包 (SDK)(如 iOS) 的包结构是扁平的——my.app, my.framework 或 my.bundle。

- `$<TARGET__PROPERTY:target,prop>`

目标的 prop 值。

- `$<TARGET__PROPERTY:prop>`

要对表达式求值目标的 prop 值。

- `$<INSTALL__PREFIX>`

当目标用 install(EXPORT) 导出, 或者当在 INSTALL_NAME_DIR 中求值时的安装目录, 若不存在为空。

特殊用法

极少数情况下, 需要将一个字符传递给具有特殊含义的生成器表达式。要避免此行为, 使用以下表达式:

- `$<ANGLE-R>`

字面值 > 符号 (比较包含 > 的字符串)

- `$<COMMA>`

字面值, 符号 (比较包含, 的字符串)

- `$<SEMICOLON>`

字面值，符号 (避免对参数进行展开)

字符串转换

使用表达式可以在生成器阶段处理字符串：

- ```
$<JOIN:list, d>
```

使用 `d` 分隔符连接以分号分隔的列表。

- ```
$<REMOVE_DUPLICATES:list>
```

对没有排序列表的删除重复的元素。

- ```
$<FILTER:list, INCLUDE | EXCLUDE, regex>
```

使用正则表达式从列表中包含/排除项。

- ```
$<LOWER_CASE:string>, $<UPPER_CASE:string>
```

将字符串转换为另一种情况。

- ```
$<GENEX_EVAL:expr>
```

将 `expr` 字符串作为当前目标上下文中的嵌套表达式求值。当嵌套表达式的求值返回另一个表达式(它们不是递归求值)时，这很有用。

- ```
$<TARGET_GENEX_EVAL:target, expr>
```

与 `GENEX_EVAL` 转换类似，但这是在目标的上下文中。

输出相关的表达式

CMake 文档未能很好地解释什么是“与输出相关的表达式”。这会让我们有点困惑，其与产出有什么关系？

根据 v3.13 文档(在更新的版本中删除了)，“这些表达式生成输出，在某些情况下取决于输入。”

有些是简易条件表达式的历史版本，其他的只是一个字符串转换表达式。

若满足特定条件，以下表达式将返回它们的第一个参数，否则返回空字符串：

- ```
$<LINK_ONLY:deps>
```

使用 `target_link_libraries()` 隐式设置以存储 PRIVATE `deps` 链接依赖，这不会作为使用需求进行传播

- ```
$<INSTALL_INTERFACE:content>
```

若与 `install(EXPORT)` 一起使用，则返回相应的内容

- ```
$<BUILD_INTERFACE:content>
```

若与 `export()` 一起使用或由同一构建系统中的另一个目标使用，则返回内容

以下输出表达式将对其参数的执行字符串进行转换:

- `$<MAKE_C_IDENTIFIER:input>`

转换为 C 标识符，遵循与字符串相同的行为 (MAKE\_C\_IDENTIFIER)。

- `$<SHELL_PATH:input>`

将绝对路径 (或路径列表) 转换为与目标操作系统匹配的 shell 路径样式。在 Windows shell 中斜杠转换为反斜杠，在 MSYS shell 中驱动器号转换为 POSIX 路径。

最后，有一个变量中查询表达式:

- `$<TARGET_OBJECTS:target>`

从目标对象库返回对象文件的列表

#### 4.4.3 可以尝试的例子

有一个很好的实际例子来支持理论时，一切都更容易理解。下面是生成器表达式的一些用法:

##### 构建配置

第 1 章中，讨论了构建类型，以指定要构建的配置——Debug、Release 等。某些情况下，可能会根据所创造的构建类型而采取不同的行动。一个简单易行的方法是使用 `$<CONFIG>` 生成器表达式:

```
target_compile_options(tgt $<$<CONFIG:DEBUG>:-ginlinepoints>)
```

上面的例子检查 CONFIG 是否等于 DEBUG；若是这种情况，嵌套表达式的值为 1。外部的简写 IF 表达式变成 true，-ginline-points 调试标志将添加到选项中。

##### 特定于系统的单行代码

生成器表达式还可以将冗长的 IF 命令压缩成简洁的一行:

```
if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
 target_compile_definitions(myProject PRIVATE LINUX=1)
endif()
```

若这是目标系统，其告诉编译器在参数中添加-DLINUX=1。虽然这不是很长，但可以很容易地替换为一个优雅的表达式:

```
target_compile_definitions(myProject PRIVATE
 ${${CMAKE_SYSTEM_NAME}:LINUX}:LINUX=1)
```

这样的代码工作得很好，但是在生成器表达式变得难以阅读之前，可以将多少内容放入生成器表达式中是有限制的。这种情况下，最好坚持使用条件块。

## 带有编译器特定标志的接口库

正如本章前面所讨论的，接口库可以用来提供与编译器匹配的标志：

```
add_library(enable_rtti INTERFACE)
target_compile_options(enable_rtti INTERFACE
$<$<OR:$<COMPILER_ID:GNU>,$<COMPILER_ID:Clang>>:-rtti>
)
```

即使在这样一个简单的例子中，已经可以看到嵌套过多生成器表达式时会发生什么，有时这是达到预期效果的唯一方法：

- 检查 COMPILER\_ID 是否为 GNU；若是这样，OR 值为 1。
- 若不是，就检查 COMPILER\_ID 是否为 Clang，并将 OR 计算为 1。否则，OR 值为 0
- 若 OR 赋值为 1，在 enable\_rtti 编译选项中添加-rtti。否则，什么都不做。

接下来，可以用 enable\_rtti 接口库链接库和可执行程序。若编译器支持，CMake 将添加-rtti 标志。

## 嵌套生成器表达式

在生成器表达式中嵌套元素时，会发生什么并不明显。可以通过向调试文件生成测试输出来调试表达式。

下面的例子，看看会发生什么：

```
chapter04/04-genex/CMakeLists.txt (fragment)

set(myvar "small text")
set(myvar2 "small > text")

file(GENERATE OUTPUT nesting CONTENT
"1 $<PLATFORM_ID>
2 $<UPPER_CASE:$<PLATFORM_ID>>
3 $<UPPER_CASE:hello world>
4 $<UPPER_CASE:${myvar}>
5 $<UPPER_CASE:${myvar2}>
")
```

输出结果如下：

```
cat nesting
1 Linux
2 LINUX
3 HELLO WORLD
4 SMALL TEXT
5 SMALL text>
```

每一行是如何工作的:

1. PLATFORM\_ID 的输出值是普通情况下 Linux。
2. 嵌套值的输出将转换为大写 LINUX。
3. 可以转换普通字符串。
4. 可以转换配置阶段变量的内容。
5. 变量首先进行插值，右尖括号 (>) 将解释为生成器表达式的一部分，因为只有字符串的一部分大写。

注意变量的内容可能会影响基因扩展的行为。若需要在变量中使用尖括号，请使用 \$<ANGLE-R>。

### 条件表达式与 BOOL 运算符求值的区别

计算布尔类型到字符串的值时，生成器表达式可能有点令人困惑。重要的是要了解它们与常规条件表达式的区别，首先是 IF 关键字:

```
chapter04/04-genex/CMakeLists.txt (fragment)

file(GENERATE OUTPUT boolean CONTENT
 "1 $<0:TRUE>
2 $<0:TRUE,FALSE> (won't work)
3 $<1:TRUE,FALSE>
4 $<IF:0,TRUE,FALSE>
5 $<IF:0,TRUE,>
")
```

这会生成一个如下所示的文件:

```
cat boolean
1
2 (won't work)
3 TRUE, FALSE
4 FALSE
5
```

检查每一行的输出：

1. 这是一个布尔展开，其中 BOOL 为 0；因此，TRUE 字符串不会写入。
2. 这是一个典型的错误——作者打算根据 BOOL 值打印 TRUE 或 FALSE，但由于它也是一个布尔值的假展开，两个参数会视为一个，并且而不打印。
3. 对于反转值也是同样的错误——是一个布尔真展开，将两个参数都写在一行中。
4. 以 IF 开头的适当条件表达式，输出 FALSE，因为第一个参数是 0。
5. 条件表达式的错误用法——当不需要为布尔值 FALSE 赋值时，应该使用第一种形式。

生成器表达式以其复杂的语法而闻名。本例中提到的差异甚至会让有经验的开发者感到困惑。可以将这样的表达式复制到另一个文件中，并使用添加的缩进和空格将其拆分，以便更好地进行理解。

## 4.5. 总结

理解目标对于编写干净、现代的 CMake 项目至关重要。本章中，不仅讨论了目标的构成，以及目标之间如何相互依赖，还讨论了如何使用 Graphviz 模块在图表中显示这些信息。

有了这些基础，就能够了解目标的关键特征——属性（各种属性）。不仅使用了一些命令在目标上设置常规属性，还解决了传递使用需求或传播属性的问题。这是一个很难解决的问题，因为不仅需要控制传播哪些属性，还需要控制如何将它们可靠地传播到选定的进一步目标。此外们还发现了当这些传播的属性来自多个源时，如何保证兼容。

然后简要地讨论了伪目标——导入目标、别名目标和接口库。其在项目中都会派上用场，特别是当知道如何将它们与传播的属性连接起来时。然后，讨论了生成的构建目标，以及如何在配置阶段对操作产生直接影响。之后，了解了自定义命令（如何生成用于其他目标的文件、编译、翻译等）和钩子函数——在构建目标时执行额外的步骤。

本章的最后一部分专门讨论了生成器表达式的概念，或者简称为 genex。我们解释了它的语法、嵌套以及它的条件表达式如何工作。然后，进行了两种类型的求值——对布尔值和对字符串值。每一种都有自己的一套表达，对此进行了详细的探讨和评论。此外，还介绍了一些使用示例，并阐明了其在实践中的工作原理。

有了这样一个坚实的基础，就为下一个主题做好了准备——将 C++ 源代码编译为可执行程序和库。

#### 4.5.1 扩展阅读

有关更多信息，请浏览以下连接：

- Graphviz 模块文档：

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/Graphviz>

<https://cmake.org/cmake/help/latest/module/CMakeGraphVizOptions.html>

- Graphviz 软件：

<https://graphviz.org>

- CMake 目标属性：

<https://cmake.org/cmake/help/latest/manual/cmakeproperties.7.html#properties-on-targets>

- 传递性使用的要求：

<https://cmake.org/cmake/help/latest/manual/cmakebuildsystem.7.html#transitive-usage-requirements>

# 第 5 章 CMake 编译 C++

简单的编译通常由工具链的默认配置处理，或者由 IDE 直接提供。但在专业环境中，业务需求通常需要更高级的东西。这可能是对更高的性能、更小的二进制文件、更强的可移植性、需要支持测试或调试等。以简单的方式管理所有这些，很快就让工程会变得复杂而混乱（特别是需要支持多个平台时）。

C++ 中对编译过程的解释往往不够充分（像虚基类这样的主题似乎更有趣）。这一章中，将了解编译是如何工作的，其内部阶段是什么，以及如何影响二进制的输出。

之后，将关注先决条件——将讨论可以使用什么命令来调整编译，如何要求编译器提供特定的特性，以及如何向编译器提供它必须处理的输入文件。

然后，关注编译的第一个阶段——预处理器。我们将为包含的头文件提供路径，并将研究如何使用预处理器定义插入来自 CMake 和环境的变量。我们将介绍一些有趣的用例，并学习如何将 CMake 变量批量公开给 C++ 代码。

之后讨论优化器，以及不同的标志如何影响性能。可能会痛苦地意识到优化的成本——调试损坏的代码有多么困难。

最后，将解释如何管理编译过程，使用预编译头文件和统一构建减少编译时间，为发现错误做准备，调试构建，并在最终的二进制文件中存储调试信息。

本章中，我们将讨论以下主题：

- 编译基础
- 预处理配置
- 配置优化器
- 编译过程

## 5.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter05>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 5.2. 编译基础

编译可以大致描述为将用高级编程语言编写的指令，转换为低级机器代码的过程。可以使用抽象概念（如类和对象）创建应用程序，而不必费心处理特定于处理器的汇编语言的细节。从而不需要

直接使用 CPU 寄存器，不需要考虑长短跳，也不需要管理堆栈帧。编译语言表达能力更强、可读性更强、更安全，并培养更易维护的代码（但仍然尽可能保持性能）。

C++ 依赖于静态编译——整个程序在执行之前必须翻译成本机代码。这是 Java 或 Python 等语言的另一种替代方法，后者在用户每次运行程序时，都会使用一个特殊的、单独的解释器动态编译程序。每种方法都有自己的优点。C++ 的策略是提供尽可能多的高级工具，同时仍然能够在一个完整的、自包含的应用程序中，为所有的架构提供原生性能。

创建和运行 C++ 程序需要几步：

1. 设计应用程序，并仔细编写源码。
2. 将单个的.cpp 实现文件（称为翻译单元）编译为目标文件。
3. 将目标文件链接到一个可执行文件中，并添加所有其他依赖项——动态库和静态库。
4. 为了运行该程序，操作系统将使用一个名为加载器的工具，将其机器码和所有必需的动态库映射到虚拟内存。然后加载器读取头文件以检查程序从哪里开始，并将控制权移交给代码。
5. C++ 运行时启动。执行一个 `special_start` 函数来收集命令行参数和环境变量。启动线程，初始化静态符号，并注册清理回调。这样，才能调用 `main()`（其中代码由开发者书写）。

相当多的工作发生在幕后，本章主要关注于列表中的第二步。通过全面考虑，可以更好地理解决一些可能的问题来自哪里。毕竟，软件中并没有什么黑魔法（即使难以逾越的复杂性让它看起来是这样）。每件事都有一个解释和理由。程序运行期间，由于编译的方式，可能会失败（即使编译步骤本身已经成功通过）。编译器不可能在工作过程中检查所有的边界情况。

### 5.2.1 编译工作

编译是将高级语言转换为低级语言的过程——是通过以特定于给定平台的二进制目标文件格式生成机器代码（特定处理器可以直接执行的指令）。在 Linux 上，最流行的格式是可执行和可链接格式（ELF）。Windows 使用 PE/COFF 格式规范。macOS 使用的是 Mach 对象（Mach-O 格式）。

目标文件是单个源文件的直接翻译。其中的每一个都必须单独编译，然后通过链接器连接到一个可执行程序或库中。所以，当修改代码时，可以通过只重新编译刚修改的文件来节省时间。

编译器必须执行以下步骤来创建一个目标文件：

1. 预处理
2. 语言分析
3. 汇编
4. 优化
5. 生成二进制文件

预处理（由大多数编译器自动调用）是实际编译前的一步。作用是以非常基本的方式操作源代码，可执行 `#include` 指令，用定义的值替换标识符（`#define` 指令和 `-D` 标志），使用简单的宏，并有条件地包含或排除基于 `#if`、`#elif` 和 `#endif` 指令的部分代码。预处理器并不了解 C++ 代码，所以其只是一个高级点的查找和替换工具。尽管如此，其工作在构建高级的程序方面还是至关重要的；将代码分解成多个部分，并跨多个翻译单元共享声明的能力，是代码重用的基础。

接下来是语言分析。编译器将逐个字符扫描文件（包含预处理器包含的所有头文件）并执行词法分析，将其分组为有意义的标记——关键字、操作符、变量名等。然后，将记号分组到记号链中，

并验证它们的顺序和存在是否遵循 C++ 的规则——这个过程称为语法分析或解析 (通常，这是打印错误方面最引人注目的部分)。最后，执行语义分析——编译器试图检测文件中的语句是否真正有意义。例如，必须满足类型正确性检查 (不能将整数赋值给字符串变量)。

汇编是根据平台可用的指令集，将这些记号转换为特定于 CPU 的指令。有些编译器实际上创建一个汇编输出文件，该文件稍后可以传递给特定的汇编程序，以生成 CPU 可以执行的机器代码。其他的则直接从内存生成相同的机器代码。通常，这样的编译器包含一个选项，可以生成人类可读的汇编文本 (这样做可能不值得)。

优化发生在整个编译过程中，在每个阶段都有。生成第一版汇编后有一个显式的阶段，该阶段负责最小化寄存器的使用并删除不使用的代码。一个有趣且重要的优化是内联扩展或内联。编译器将“剪切”函数体并“粘贴”它，而不是调用 (标准没有定义在什么情况下会发生这种情况——这取决于编译器的实现)。这个过程加快了执行速度并减少了内存的使用，但不利于调试 (执行的代码不再位于原来的行)。

生成二进制文件由根据目标平台指定的格式将优化的机器代码写入目标文件组成。这个目标文件还没有准备好执行——必须传递给下一个工具，链接器，其将适当地重新定位我们的目标文件的部分，并解析对外部符号的引用。这是从 ASCII 源代码转换为处理器可识别的二进制目标文件。

这些阶段中的每一个都是重要的，可以配置以满足我们的特定需求。看看如何使用 CMake 管理这个过程。

### 5.2.2 初始配置

CMake 提供了多个指令来影响每个阶段：

- `target_compile_features()`: 需要具有特定功能的编译器来编译此目标。
- `target_sources()`: 向已定义的目标添加源。
- `target_include_directories()`: 设置预处理器的包含路径。
- `target_compile_definitions()`: 设置预处理定义。
- `target_compile_options()`: 特定于编译器的选项。
- `target_precompile_headers()`: 预编译头文件。

以上所有指令都接受类似的参数：

```
target_...(<target name> <INTERFACE | PUBLIC | PRIVATE>
<value>)
```

所以，其支持属性传播，并且可以用于可执行程序和库。另外，还要提醒一下——所有指令都支持生成器表达式。

#### 要求编译器提供特定的特性

正如第 3 章中所讨论的那样，需要为出错做好准备，并致力于为软件的用户提供明确的信息——可用的编译器 X 并没有提供所需的特性 y。这比用户在不兼容的工具链中所产生的错误要好得多。我们不希望用户认为这是代码的问题，而不是他们的环境出了问题。

下面的指令允许指定目标构建所需的所有特性：

```
target_compile_features (<target> <PRIVATE|PUBLIC|INTERFACE>
 <feature> [...])
```

CMake 很了解 C++ 标准，并支持这些 compiler\_ids 的编译器特性：

- AppleClang: Apple Clang for Xcode versions 4.4+
- Clang: Clang Compiler versions 2.9+
- GNU: GNU Compiler versions 4.4+
- MSVC: Microsoft Visual Studio versions 2010+
- SunPro: Oracle Solaris Studio versions 12.4+
- Intel: Intel Compiler versions 12.1+

#### 重要的 Note

当然，可以使用 CMAKE\_CXX\_KNOWN\_FEATURES 变量，但建议使用通用 C++ 标准——cxx\_std\_98, cxx\_std\_11, cxx\_std\_14, cxx\_std\_17, cxx\_std\_20, 或 cxx\_std\_23。查看扩展阅读部分了解更多细节。

### 5.2.3 管理目标源

已经知道如何告诉 CMake 哪些源文件组成了单个目标——可执行文件或库。当使用 add\_executable() 或 add\_library() 时，需要提供文件列表。

随着解决方案的增长，每个目标的文件列表也会增长。最后可能会得到一些非常冗长的 add\_…() 指令。应该如何应对呢？一个诱惑可能是在 GLOB 模式下使用 file() 指令——可以从子目录收集所有文件并将它们存储在一个变量中。可以将它作为参数传递给目标声明，而不必再费心于列表文件：

```
file(GLOB helloworld_SRC "*.h" "*.cpp")
add_executable(helloworld ${helloworld_SRC})
```

但不建议这种方法。CMake 根据列表文件中的更改生成构建系统，所以若不做更改，构建可能会在没有任何警告的情况下中断（从长时间的调试经验中我们知道，这是最糟糕的中断类型）。除此之外，没有在目标声明中列出所有的源代码将破坏诸如 CLion (CLion 只解析一些命令来理解项目) 等 IDE 的代码检查。

若不建议在目标声明中使用变量，如何有条件地添加源文件，例如：当处理特定平台的实现文件，如 gui\_linux.cpp 和 gui\_windows.cpp？

可以使用 target\_sources() 指令追加源文件到之前创建的目标：

```
chapter05/01-sources/CMakeLists.txt

add_executable(main main.cpp)
if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
 target_sources(main PRIVATE gui_linux.cpp)
```

```
elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
target_sources(main PRIVATE gui_windows.cpp)
endif()
```

这样，每个平台都可以获得自己的一组兼容文件，但是长长的文件列表应该如何处理呢？好吧，我们只能接受有些东西还不完美，并继续手动添加它们。

既然已经了解了关于编译的关键知识点，接下来让我们更深入地了解第一步——预处理。

## 5.3. 预处理配置

预处理器在构建过程中起着巨大的作用，但其功能却又如此简单和有限。在接下来的小节中，将介绍如何提供包含文件的路径，以及如何使用预处理器定义。还将解释如何使用 CMake 配置包含的头文件。

### 5.3.1 提供包含文件的路径

预处理器最基本的特性是能够使用 `#include` 指令包含.h/.hpp 头文件。它有两种形式：

- `#include <path-spec>`: 尖括号式
- `#include "path-spec"`: 引号式

预处理器将用指定路径中的文件内容替换这些指令。找到这些文件可能是个问题。我们按什么顺序搜索哪些目录？但 C++ 标准并没有明确规定这一点，这需要查看编译器的手册。

通常，尖括号形式将检查标准包含目录，包括标准 C++ 库和标准 C 库头文件存储在系统中的目录。

引号式将开始在当前文件的目录中搜索包含的文件，然后在目录中查找带尖括号的目录。

CMake 提供了一个指令来操作头文件的搜索路径，以找到需要包含的头文件：

```
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
<INTERFACE|PUBLIC|PRIVATE> [item1...]
[<INTERFACE|PUBLIC|PRIVATE> [item2...] ...])
```

可以添加希望编译器检查的自定义路径，CMake 会将它们添加到生成的构建系统中的编译器调用中，这将提供一个特定于编译器的标志（通常是-I）。

使用 BEFORE 或 AFTER 确定路径是应该添加到目标 `INCLUDE_DIRECTORIES` 属性的前面还是后面，这仍然由编译器决定是在默认目录之前还是之后检查这里提供的目录（通常是在之前）。

`SYSTEM` 关键字通知编译器所提供的目录是标准的系统目录（与尖括号形式一起使用）。对于许多编译器，这个值将作为`-system` 标志提供。

### 5.3.2 预处理宏定义

还记得在描述编译阶段时提到的预处理器的 `#define` 和 `#if`、`#elif` 和 `#endif` 指令吗？看看下面的例子：

```
1 // chapter05/02-definitions/definitions.cpp
2
3 #include <iostream>
4 int main() {
5 #if defined(ABC)
6 std::cout << "ABC is defined!" << std::endl;
7 #endif
8
9 #if (DEF < 2*4-3)
10 std::cout << "DEF is greater than 5!" << std::endl;
11 #endif
12 }
```

实际上，这个例子什么也没有做；既没有定义 ABC 也没有定义 DEF（在本例中 DEF 默认为 0）。可以通过在这段代码的顶部添加两行代码来轻松地改变这一点：

```
1 #define ABC
2 #define DEF 8
```

编译并执行这段代码后，可以在控制台中看到这两条信息：

```
1 ABC is defined!
2 DEF is greater than 5!
```

这看起来很简单，但若我们想根据外部因素（如操作系统、体系结构或其他东西）来调整这些部分，会发生什么情况呢？好消息！可以将值从 CMake 传递到 C++ 编译器，这一点也不复杂。

`target_compile_definitions()` 指令就可以做到这一点：

```
chapter05/02-definitions/CMakeLists.txt

set(VAR 8)
add_executable(define definitions.cpp)
target_compile_definitions(define PRIVATE ABC
 "DEF=${VAR}")
```

前面的代码将完全类似于两个 `#define` 语句，可以自由使用 CMake 的变量和生成器表达式，并且可以将命令放在条件块中。

#### 重要的 Note

这些定义通常以-D 标志——`-DFOO=1`——传递给编译器，一些开发者仍然在这个命令中使用这个标志：

```
target_compile_definitions(hello PRIVATE -DFOO)
```

CMake 可以识别这一点，并将删除任何前导-D 标志。也会忽略空字符串，所以可以这样写：

```
target_compile_definitions(hello PRIVATE -D FOO)
```

`-D` 是一个单独的参数，在删除后成为一个空字符串，然后被忽略。

## 单元测试私有类的常见问题

为了单元测试的目的，一些在线资源建议使用特定的-D 定义和 #ifdef/ifndef 指令的组合。最简单的方法是将访问说明符包装在条件包含中，并在定义 UNIT\_TEST 时忽略它们：

```
1 class X {
2 #ifndef UNIT_TEST
3 private:
4 #endif
5 int x_;
6 }
```

虽然这个用例非常方便（允许测试直接访问私有成员），但代码不干净。单元测试应该只测试公共接口中的方法是否按预期工作，并将底层实现视为黑盒机制。这里建议只将此作为压箱底的手段。

## 使用 Git 进行编译版本跟踪

从了解环境或文件系统的细节中受益的用例，一个很好的例子可能是传递用于构建二进制文件的修订或提交 SHA：

```
chapter05/03-git/CMakeLists.txt

add_executable(print_commit print_commit.cpp)
execute_process(COMMAND git log -1 --pretty=format:%h
 OUTPUT_VARIABLE SHA)
target_compile_definitions(print_commit PRIVATE
 "SHA=${SHA}")
```

然后可以在应用中使用：

```
1 // chapter05/03-git/print_commit.cpp
2
3 #include <iostream>
4 // special macros to convert definitions into c-strings:
5 #define str(s) #s
6 #define xstr(s) str(s)
7 int main()
8 {
9 #if defined(SHA)
10 std::cout << "GIT commit: " << xstr(SHA) << std::endl;
11 #endif
12 }
```

当然，前面的代码要求用户在他们的 PATH 中安装了 Git。当在生产主机上运行的程序来自持续集成/部署流水时，这很有用。若软件有问题，也可以快速检查到底是哪一个 Git 提交构建了有问题的产品。

跟踪准确的提交对于调试非常有用。对于单个变量来说，这不是很多工作，但是当有几十个变量要传递给头文件时会发生什么呢？

### 5.3.3 配置头文件

若有很多个变量，可以通过 `target_compile_definitions()` 传递定义可能会有一点开销。但不能只提供一个头文件，其中包含引用各种变量的占位符，然后让 CMake 填充它们吗？

可以使用 `configure_file(<input> <output>)` 指令，可以从这样的模板生成新的文件：

```
1 //chapter05/04-configure/configure.h.in
2
3 #cmakedefine FOO_ENABLE
4 #cmakedefine FOO_STRING1 "@FOO_STRING@"
5 #cmakedefine FOO_STRING2 "${FOO_STRING}"
6 #cmakedefine FOO_UNDEFINED "@FOO_UNDEFINED@"
```

然后，可以像这样使用：

```
chapter05/04-configure/CMakeLists.txt

add_executable(configure configure.cpp)
set(FOO_ENABLE ON)
set(FOO_STRING1 "abc")
set(FOO_STRING2 "def")
configure_file(configure.h.in configured/configure.h)
target_include_directories(configure PRIVATE
 ${CMAKE_CURRENT_BINARY_DIR})
```

可以让 CMake 生成一个输出文件，像这样：

```
1 //chapter05/04-configure/<build_tree>/configure.h
2
3 #define FOO_ENABLE
4 #define FOO_STRING1 "abc"
5 #define FOO_STRING2 "def"
6 /* #undef FOO_UNDEFINED "@FOO_UNDEFINED@" */
```

`@VAR@` 和  `${VAR}` 变量占位符被 CMake 列表文件中的值所替换。对于已定义的变量，`#cmakedefine` 替换为 `#define`，而对于未定义的变量使用`/* #undef VAR */`替换。

若需要显式的 `#define 1` 或 `#define 0` 为 `# If` 块，可以使用 `#cmakedefine01` 代替。

如何在应用中使用这样的配置头文件呢？可以将它包含在实现文件中：

```
1 //chapter05/04-configure/configure.cpp
2
3 #include <iostream>
4 #include "configured/configure.h"
5 // special macros to convert definitions into c-strings:
6 #define str(s) #s
7
8 #define xstr(s) str(s)
9 using namespace std;
10 int main()
```

```
11 {
12 #ifdef FOO_ENABLE
13 cout << "FOO_ENABLE: ON" << endl;
14 #endif
15 cout << "FOO_ENABLE1: " << xstr(FOO_ENABLE1) << endl;
16 cout << "FOO_ENABLE2: " << xstr(FOO_ENABLE2) << endl;
17 cout << "FOO_UNDEFINED: " << xstr(FOO_UNDEFINED) << endl;
18 }
```

并且这里已经用 `target_include_directories()` 指令将构建树目录添加到 `include` 路径中，所以可以编译这个例子并运行：

```
FOO_ENABLE: ON
FOO_ENABLE1: FOO_ENABLE1
FOO_ENABLE2: FOO_ENABLE2
FOO_UNDEFINED: FOO_UNDEFINED
```

`configure_file()` 指令还有许多格式化和文件权限选项。若感兴趣请查看在线文档以获取详细信息 (链接在扩展阅读部分)。

准备好头文件和源文件的完整组合之后，就可以讨论在接下来的步骤中如何形成输出代码。由于不能直接影响语言分析或汇编 (这些步骤遵循严格的标准)，所以肯定可以访问优化器的配置。让我们来学习这是如何影响最终结果的吧。

## 5.4. 配置优化器

优化器将分析前一阶段的输出，并使用大量的技巧，开发者可能会认为这些技巧很脏，因为它们不遵守干净代码的原则。这没关系——优化器的关键作用是提高代码的性能 (使用较少的 CPU 周期、较少的寄存器和较少的内存)。当优化器遍历源代码时，将对其进行大量转换，以至于几乎无法阅读，会成为目标 CPU 专门准备的版本。优化器不仅将决定哪些函数可以删除或压缩，还会移动代码，甚至复制代码！若能够完全确定某些代码行没有意义，其将从一个重要函数的中间删除 (开发者不太会注意)。它将重用内存，因此许多变量可以在不同的时间段占用相同的插槽。若可以在这里或那里减少一些指令周期，把相应的控制结构转变成完全不同的结构。

这里描述的技术，若由程序员手动应用到源代码中，将会造成源代码变成可怕的、不可读的混乱符号集。很难阅读的同时，这些符号也很难写，也就更难于人为推演了。

另一方面，若编译器应用这些代码，其将完全按照所写的顺序进行。优化器是一头无情的野兽，只服务于一个目的：使执行快速，无论输出将如何混乱。若在测试环境中运行这样的输出，那么可能包含一些调试信息，也可能不包含，使未经授权的人难以修改。

每个编译器都有自己的技巧，与所遵循的平台和原理保持一致。我们将看一看最常见的，在 GNU GCC 和 LLVM Clang 中可用的优化器，从而了解什么是有用的，什么是可能的。

许多编译器默认情况下不会启用任何优化 (包括 GCC)，但在其他情况下就不是这样了。能快走为什么要慢走？要更改这些内容，可以使用 `target_compile_options()` 指令，并确切地指定想从编译器获得的内容。

该指令的语法与本章其他指令类似:

```
target_compile_options(<target> [BEFORE]
<INTERFACE|PUBLIC|PRIVATE> [items1...]
[<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

我们提供要添加的目标命令行选项，并指定传播关键字。指令执行时，CMake 将把给定的选项附加到目标的适当的 COMPILE\_OPTIONS 变量中。可选的 BEFORE 关键字可以用来指定把相应的选项放在前面。

#### 重要的 Note

target\_compile\_options() 是一个通用指令。还可以用来为类似编译器的-D 定义提供其他参数，为此 CMake 还提供了 target\_compile\_definition() 指令。总是建议尽可能使用 CMake 指令，因为其在所有支持的编译器上以相同的方式工作。

后面的部分将介绍可以在大多数编译器中启用的各种优化。

### 5.4.1 优化级别

优化器的所有不同行为，都可以通过编译选项传递的特定标志进行配置。要了解所有这些方法挺花时间的，并且需要了解大量关于编译器、处理器和内存内部工作原理的知识。若只想要在大多数情况下运行良好的最佳可能场景，能做什么呢？可以找到一个通用的解决方案——优化级说明符。

大多数编译器提供了从 0 到 3 的四个基本优化级别，用-O<level> 选项来指定。-O0 表示不进行优化，通常这是编译器的默认级别。另一方面，-O2 认为是完全优化，生成高度优化的代码，但以最慢的编译时间为代价。

有一个介于两者之间的-O1 级别，这（取决于需要）可能是一个很好的折衷方案——其支持合理数量的优化机制，而不会大幅降低编译速度。

可以使用-O3，这是完全优化，就像-O2 一样，但使用更积极的方法实现子程序内联和循环向量化。

还有一些优化变量会优化生成文件的大小（不一定是速度），-Os。有一个超级激进的优化，-Ofast，不严格遵守 C++ 标准的-O3 优化。最明显的区别是-fast-math 和-ffinite-math 标志的使用，若程序是关于精确计算的（大多数都是这样），最好望不使用这个参数。

CMake 了解不是所有的编译器都是相同的。因此，其通过为编译器提供一些默认标志来标准化开发人员的体验，存储在所用语言（CXX 用于 C++）和构建配置（DEBUG 或 RELEASE）的系统范围（不是特定于目标的）变量中：

- CMAKE\_CXX\_FLAGS\_DEBUG 等于 -g
- CMAKE\_CXX\_FLAGS\_RELEASE 等于 -O3 -DNDEBUG

Debug 配置不支持任何优化，发布配置直接针对 O3。可以使用 set() 指令直接修改，或者添加一个目标编译选项，覆盖默认行为。另外两个标志（-g, -DNDEBUG）与调试相关——将在为调试器提供信息一节中讨论。

诸如 CMAKE\_<LANG>\_FLAGS\_<CONFIG> 这样的变量是全局变量——适用于所有目标。建议通过属性和 target\_compile\_options() 等指令配置目标，而不是依赖全局变量。通过这种方式，可

以更细粒度的控制目标。

通过使用`-O<level>`选择优化级别，间接地设置了一长串标志，每个标志控制特定的优化行为。然后，可以通过添加更多标志来微调优化：

- 使用`-f`选项启用它们：`-finline-functions`。
- 使用`-fno`选项禁用它们：`-fno-inline-functions`。

其中一些标志需要花点时间了解，因为它们会影响程序的工作方式和调试方式。

### 5.4.2 函数内联

代码中，可以通过在类声明块中定义函数或显式使用`inline`关键字来提示编译器内联一些函数：

```
1 struct X {
2 void im_inlined() { cout << "hi\n"; };
3 void me_too();
4 };
5 inline void X::me_too() { cout << "bye\n"; };
```

由编译器决定函数是否内联。若启用了内联，并且函数只在一个地方使用（或者是在几个地方使用的相对较小的函数），那么很可能会触发内联。

这是一种非常有趣的优化技术。其工作原理是，从有问题的函数中提取代码，并将其放在调用函数的所有位置，替换原始调用并节省宝贵的CPU周期。

考虑以下类：

```
1 int main() {
2 X x;
3 x.im_inlined();
4 x.me_too();
5 return 0;
6 }
```

若没有内联，代码将在`main()`中执行，直到方法调用。然后，为`im_inlined()`创建一个新函数，在单独的作用域中执行，并返回`main()`。`me_too()`也是同样的情况。

当内联时，编译器将替换调用：

```
1 int main() {
2 X x;
3 cout << "hi\n";
4 cout << "bye\n";
5 return 0;
6 }
```

这不是一种精确的表示，因为内联发生在汇编或机器码级别（而不是源码级别）。编译器这样做是为了节省时间，不需要创建和删除一个新的调用作用域，也不需要查找要执行（和返回）的下一条指令的地址，而且可以更好地缓存附近的指令。

当然，内联有一些重要的副作用；若函数多次使用，则必须将其复制到所有位置（意味着更大的文件大小和更多的内存使用）。现在，这可能不像过去那么重要了，但仍是相关的，因为不断开发的软件需要运行在没有太多RAM的低端设备上。

除此之外，当调试代码时，会对我们产生严重影响。内联代码不再是其最初编写的行号，因此不容易（有时甚至不可能）跟踪。这正是，放置在内联函数中的断点从未命中的原因（尽管代码仍然以某种方式执行）。为了避免这个问题，需要为调试版本禁用内联（代价是不能测试与发布版本完全相同）。

可以通过为目标指定-O0 级别，或者直接在负责的标志后面执行：

- -finline-functions-called-once: 只有 GCC 可用
- -finline-functions: Clang 和 GCC 都可用
- -finline-hint-functions: 只有 Clang 可用
- -finline-functions-called-once: 只有 GCC 可用

可以使用-fno-inline-...，显式禁用内联。有关详细信息，请参阅编译器的特定版本的文档。

### 5.4.3 循环展开

循环展开是一种优化技术，方法是将循环转换为一组语句，以达到相同的效果。可以用程序的大小来换取执行速度，将减少或消除控制循环指针算术或循环结束测试的指令。

考虑下面的例子：

```
1 void func() {
2 for(int i = 0; i < 3; i++)
3 cout << "hello\n";
4 }
```

前面的代码将转换成这样：

```
1 void func() {
2 cout << "hello\n";
3 cout << "hello\n";
4 cout << "hello\n";
5 }
```

结果相同，但是不再需要分配 i 变量，增加它的值，或者将它与值 3 进行三次比较。若在程序的生命周期中调用 func() 的次数足够多，那么展开即使是这样一个短小的函数也会产生显著的差异。

然而，重要的是要了解两个限制因素。循环展开只有在编译器知道或能够有效估计迭代量的情况下才会有用。其次，循环展开可能会对现代 CPU 产生不良影响，因为增加的代码大小可能会消耗有效的缓存。

每个编译器提供的该标志版本略有不同：

- -floop-unroll: GCC
- -funroll-loops: Clang

若有疑问，请测试此标志是否影响程序，并显式启用或禁用它。注意，在 GCC 上是用-O3 会隐式使用-floop-unroll-and-jam 标志。

#### 5.4.4 循环向量化

单指令多数据 (SIMD) 是 20 世纪 60 年代早期为实现并行而发展起来的机制之一。就像其名称一样，可以同时对多条信息执行相同的操作。实践中意味着什么？考虑下面的例子：

```
1 int a[128];
2 int b[128];
3 // initialize b
4 for (i = 0; i<128; i++)
5 a[i] = b[i] + 5;
```

前面的代码将循环 128 次，但若有一个功能强大的 CPU，可以通过同时计算数组中的两个或多个元素来更快地执行代码。这是可行的，因为连续的元素之间没有依赖关系，数组之间的数据也没有重叠。智能编译器可以将前面的循环转换为类似的东西（发生在汇编级别）：

```
1 for (i = 0; i<32; i+=4) {
2 a[i] = b[i] + 5;
3 a[i+1] = b[i+1] + 5;
4 a[i+2] = b[i+2] + 5;
5 a[i+3] = b[i+3] + 5;
6 }
```

GCC 将在-O3 处启用这种循环的自动向量化，Clang 默认启用它。这两个编译器提供了不同的标志来启用/禁用向量化：

- -ftree-vectorize -ftree-slp-vectorize 在 GCC 中启用向量化
- -fno-vectorize -fno-slp-vectorize 在 Clang 中禁用向量化

向量化的性能来自于利用 CPU 供应商提供的特殊指令，而不是简单地用展开版本替换原来的循环形式。因此，通过手动操作不可能达到相同的性能水平（而且，这不是非常干净的代码）。

优化器的作用对于在运行时增强程序的性能非常重要。通过有效地运用它的策略，将得到更大的回报。效率不仅在编码完成后很重要，而且在软件开发过程中也很重要。若编译时间很长，可以通过更好地管理过程来进行改进。

## 5.5. 编译过程

作为开发者和构建工程师，还需要考虑编译的其他方面——完成编译所需的时间，以及在构建解决方案的过程中发现和修复错误的难易程度。

### 5.5.1 减少编译时间

每天（或每小时）需要多次重新编译的繁忙项目中，尽可能快地编译是至关重要的。这不仅会影响代码编译-测试循环的紧凑程度，还会影响注意力和工作流程。

由于有独立的翻译单元，C++ 已经非常擅长管理编译时间。CMake 将只负责重新编译受最近更改影响的源代码。若需要进一步改进，可以使用一些技术——头文件预编译和统一构建。

### 预编译头文件

头文件 (.h) 在实际编译开始之前由预处理器包含在翻译单元中，所以每次.cpp 实现文件更改时都必须重新编译。若多个翻译文件使用相同的共享头文件，那么每次包含它时都必须编译它。这是一种浪费，但在很长一段时间里情况就是这样。

从 3.16 版本开始，CMake 提供了一个命令来启用头预编译。

这允许编译器将头文件与实现文件分开处理，从而加快编译速度。以下是相应的指令：

```
target_precompile_headers (<target>
 <INTERFACE|PUBLIC|PRIVATE> [header1...]
 [<INTERFACE|PUBLIC|PRIVATE> [header2...] ...])
```

添加的头文件列表存储在 PRECOMPILE\_HEADERS 目标属性中。第 4 章已经介绍，可以使用 PUBLIC 或 INTERFACE 关键字，通过传播的属性与依赖的目标共享，但对于通过 install() 指令导出的目标不行。其他项目不应该强制使用预编译的头文件。

#### 重要的 Note

若需要内部预编译头文件，并且仍然想安装-导出目标文件，\$<BUILD\_INTERFACE:...> 的生成器表达式将防止头文件出现在使用中，但仍然会添加到使用 export() 指令从构建树导出的目标中。

CMake 将把所有头文件的名称放在一个 cmake\_pch.h|xx 文件中，然后该文件将预编译为一个特定于编译器的二进制文件，扩展名为.pch、.gch 或.pchi。

可以这样使用：

```
chapter05/06-precompile/CMakeLists.txt

add_executable(precompiled hello.cpp)
target_precompile_headers(precompiled PRIVATE <iostream>)
```

```
// chapter05/06-precompile/hello.cpp

int main() {
 std::cout << "hello world" << std::endl;
}
```

main.cpp 文件中，不需要包含 cmake\_pch.h 或任何其他头文件——将由 cmake 使用特定于编译器的命令行选项强制包含。

前面的例子中，使用了一个内置头文件。这里，可以很容易地用类或函数定义添加自己的头文件：

- header.h 作为相对于当前源目录的文件，将包含在一个绝对路径中。
- [”header.h”] 根据编译器的实现进行解释，通常在 INCLUDE\_DIRECTORIES 变量中找到。使用 target\_include\_directories() 来配置。

一些在线引用会阻止预编译不属于标准库的头文件，比如 <iostream>，或者完全使用预编译的头文件。这是因为更改列表或编辑自定义头文件，将需要重新编译目标中的所有翻译单元。使用

CMake 不需要担心太多，特别是在正确地组织了项目的情况下（具有相对较小的目标，专注于单个领域）。每个目标都有一个单独的预编译头文件，其限制了头文件更改的影响。

另一方面，若头文件相当稳定，那么在另一个目标中重用预编译的头文件是个好主意。CMake 为此提供了一个方便的指令：

```
target_precompile_headers(<target> REUSE_FROM
<other_target>)
```

这将设置重用头文件的目标的 PRECOMPILE\_HEADERS\_REUSE\_FROM 属性，并在这些目标之间创建一个依赖关系。通过使用此方法，目标不再能够指定自己的预编译头文件。此外，所有编译选项、编译标志和编译定义必须在目标之间匹配。请注意需求，特别是使用双括号格式 ([[“header.h”]]) 使用头文件。两个目标都需要设置头文件包含路径，以确保编译器能够找到这些头文件。

## 统一构建

CMake 3.16 还引入了另一个编译时间优化特性——统一构建或巨型构建。统一构建用 #include 指令组合多个实现源文件（毕竟，编译器不知道包含的是头文件还是实现）。其中，有些是真的有用的，而另一些是潜在的有害的。

从最明显的一个开始——当 CMake 创建统一构建文件时，需要避免在不同的翻译单元重新编译头文件：

```
1 #include "source_a.cpp"
2 #include "source_b.cpp"
```

当这两个源文件都包含 #include "header.h" 时，由于有头文件包含守卫，相应的头文件只会解析一次（假设没有忘记添加这些守卫）。这没有预编译头文件那么优雅，但这是一个选项。

这种类型构建的第二个好处是，优化器现在可以在更大的规模上执行，并在所有绑定的源之间优化过程间调用，这类似于链接时优化。

然而，这些好处是有代价的。当减少了目标文件的数量和处理步骤时，也增加了处理更大文件所需的内存量。此外，减少了可并行工作的数量。编译器实际上并不擅长多线程编译——构建系统通常会启动许多编译任务，在不同的线程上同时执行所有文件。当把所有的文件聚集在一起时，这会使编译过程变得更加困难，因为 CMake 现在会在我们创建多少个大型构建之间调度并行构建。

统一构建中，还需要考虑一些 C++ 语义含义，这些可能不太容易捕捉到——跨文件隐藏符号的匿名命名空间现在的作用域是组。静态全局变量、函数和宏定义也会发生同样的情况。这可能导致名称冲突，或执行错误的函数重载。

重新编译时，大型构建是不可取的，因为会编译比所需的多得多的文件。当代码要以尽可能快的速度编译所有文件时，其工作效率最高。在 Qt Creator 上进行的测试表明，性能提升幅度在 20%–50% 之间（取决于使用的编译器）。

要启用统一构建，有两个选项：

- 将 CMAKE\_UNITY\_BUILD 变量设置为 true ——在此之后，定义的每个目标上都会初始化 UNITY\_BUILD 属性。
- 在每个应该使用统一构建的目标上手动设置 UNITY\_BUILD 为 true。

第二个选项是通过以下函数实现的：

```
set_target_properties(<target1> <target2> ...
 PROPERTIES UNITY_BUILD true)
```

通常，CMake 将创建包含 8 个源文件的构建，由目标的 `UNITY_BUILD_BATCH_SIZE` 属性指定（从 `CMAKE_UNITY_BUILD_BATCH_SIZE` 变量创建目标时复制）。这里，可以更改目标属性或默认变量。

从 3.18 版本开始，可以决定显式地定义如何将文件与命名组捆绑在一起。为此，改变目标的 `UNITY_BUILD_MODE` 属性为 `GROUP`（默认总是 `BATCH`）。然后，需要将源文件分配给组，通过设置他们的 `UNITY_GROUP` 属性为设置的名称：

```
set_property(SOURCE <src1> <src2>...
 PROPERTY UNITY_GROUP "GroupA")
```

然后，CMake 将忽略 `UNITY_BUILD_BATCH_SIZE`，并将组中的所有文件添加到单个统一构建中。

CMake 的文档建议默认情况下不要为公共项目启用统一构建，建议应用程序的最终用户能够通过提供 `DCMAKE_UNITY_BUILD` 命令行参数来决定他们是否需要统一构建。而且，若因为代码的编写方式而引起问题，应该显式地将目标的统一构建属性设置为 `false`。但是，没有什么可以阻止为内部使用的代码启用该特性，例如：公司内部或私人项目中。

## 不支持 C++20 的模块

若密切关注 C++ 标准发布，就应该了解 C++20 中引入的新特性——模块。这是一个重大的游戏规则改变者，其允许使用头文件时避免许多麻烦，减少构建时间，并允许更干净、更紧凑的代码，更容易浏览和推理。

本质上，不需要创建单独的头文件和实现文件，而是创建一个带有模块声明的文件：

```
1 export module hello_world;
2 import <iostream>;
3 export void hello() {
4 std::cout << "Hello world!\n";
5 }
```

然后，可以通过导入使用：

```
1 import hello_world;
2 int main() {
3 hello();
4 }
```

注意，这里不再依赖于预处理器；模块有自己的关键字——`import`、`export` 和 `module`。作为编写和构建 C++ 解决方案的新方法，主流编译器的最新版本已经可以执行支持模块的所有必要任务。我希望在本章开始的时候，CMake 中已经提供了对模块的一些早期支持。但这还没有发生，有可能在你买这本书的时候（或不久之后）就有了。

Kitware 开发人员已经创建（并在 3.20 中发布）了一个新的实验性特性，以支持对 Ninja 生成器的 C++20 模块依赖项扫描。目前，其仅针对编译器开发者，以便他们可以在开发依赖项扫描工具时

进行测试。

当这个备受期待的特性完成并在稳定版本中可用时，我建议深入研究它。希望它能够简化和加快编译速度，这是目前可用的方法都无法媲美的。

### 5.5.2 查找错误

作为开发者，花了很多时间来寻找 bug。这是一个可悲的事实。发现错误并解决它们通常会让我们感到恼火，特别是当这需要花费很长时间的时候。若是盲目查找，没有工具帮助我们，那就更困难了。这就是为什么我们应该小心地设置环境，使这个过程尽可能简单。为此，需要使用 `target_compile_options()` 配置编译器。那么，哪些编译选项可以帮助我们呢？

#### 错误和警告的配置

软件开发中有很多压力很大的事情——在半夜修复关键的 bug，在大型系统中处理高可见度、代价高昂的故障，处理烦人的编译错误，特别是那些难以理解或难以修复的错误。当为了简化工作和减少失败的机会而研究一个主题时，会发现很多关于如何配置编译器警告的建议。

一个很好的建议就是为所有构建启用 `-Werror` 标志作为默认值。这个标志的作用非常简单——所有警告都视为错误，除非解决了所有警告，否则代码不会编译。虽然这看起来像是一个好主意，但这从来都不是一个好主意。

警告不是错误是有原因的，这要由你来决定怎么做。开发者有忽视警告的自由，特别是当对解决方案进行试验和创建原型时，通常是一件好事。

另一方面，若有一段完美的、没有任何警告的、完美的代码，未来的修改破坏这种状态就太可惜了。启用警告视为错误，并持续使用有什么坏处呢？似乎没有。至少在编译器升级之前是这样。新版本的编译器倾向于严格限制已弃用的功能，或者只是更好地建议需要改进的地方。当不把所有的警告都视为错误时，没问题，但当启用了，会发现有一天在没有更改代码时，构建开始崩溃，或者更令人沮丧的是，需要快速修复一个与新警告完全无关的问题。那么，什么时候应该启用呢？

最简单的答案是，当在编写一个公共库时。然后，真的希望避免因为代码是在更严格的环境中编译而发出抱怨，抱怨代码不规范。若决定启用它，请确保跟上了编译器的新版本，及其引入的警告。

否则，让警告成为警告，并专注于错误。若觉得内部需要学究气，请使用 `-Wpedantic` 标志。这是一个有趣的选项——启用了严格的 ISO C 和 ISO C++ 要求的所有警告。请注意，不能使用此标志检查代码是否符合标准——其只会查找需要诊断消息的非 ISO 实践。

更宽容和接地气的开发者将满足 `-Wall` 和可选的 `-Wextra`，这些是认为是非常有用和有意义的警告。当有空闲时间时，应该在代码中修复它们。

还有许多其他的警告标志，根据项目的类型，这些警告标志可能是有用的。我建议您阅读所选编译器的手册，看看相应的警告具体代表了什么。

#### 调试构建

有时候，编译会崩溃。这通常发生在我们试图重构一堆代码或清理构建系统时。有时，问题很容易解决，但随后会出现更复杂的问题，需要深入了解配置步骤。我们已经知道如何打印更详细的 CMake 输出，但是如何分析每个阶段实际发生的事情呢？

## 各个阶段的调试

有一个`-save-temp`s 标志可以传递给编译器 (GCC 和 Clang 都有), 将强制每个阶段的输出存储在一个文件而不是内存中:

```
chapter05/07-debug/CMakeLists.txt

add_executable(debug hello.cpp)
target_compile_options(debug PRIVATE -fprofile-arcs -fcoverage-mapping)
```

上面的代码段通常会产生两个文件:

- <build-tree>/CMakeFiles/<target>.dir/<source>.ii: 存储预处理阶段的输出, 用注释解释源代码的每个部分来自哪里:

```
1 # 1 "/root/examples/chapter05/06-debug/hello.cpp"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # / / / ... removed for brevity ... / / /
6 # 252 "/usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h" 3
7 namespace std
8 {
9 typedef long unsigned int size_t;
10 typedef long int ptrdiff_t;
11 typedef decltype(nullptr) nullptr_t;
12 }
13 ...
```

- <build-tree>/CMakeFiles/<target>.dir/<source>.s: 语言分析阶段的输出, 为汇编阶段做好准备:

```
1 .file "hello.cpp"
2 .text
3 .section .rodata
4 .type _ZStL19piecewise_construct, @object
5 .size _ZStL19piecewise_construct, 1
6 _ZStL19piecewise_construct:
7 .zero 1
8 .local _ZStL8__ioinit
9 .comm _ZStL8__ioinit,1,1
10 .LC0:
11 .string "hello world"
12 .text
13 .globl main
14 .type main, @function
15 main:
16 (...)
```

根据问题的种类, 可以发现实际问题是什么。预处理器的输出对于发现错误很有用, 比如不正确的 include 路径 (提供了错误的库版本) 和导致不正确的 #ifdef 计算的错误。

语言分析的输出，对于特定处理器和解决关键优化问题很有用。

## 头文件的调试问题

错误包含的文件是一个非常难调试的问题。这是我在公司的一份工作，将整个代码库从一个构建系统移植到另一个。若发现自己处于一个需要精确理解哪些路径用来包含请求的头文件的位置，可以使用-H：

```
chapter05/07-debug/CMakeLists.txt

add_executable(debug hello.cpp)
target_compile_options(debug PRIVATE -H)
```

打印出来的结果看起来类似这样：

```
[25%] Building CXX object
CMakeFiles/inclusion.dir/hello.cpp.o
. /usr/include/c++/9/iostream
.. /usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h
... /usr/include/x86_64-linux-gnu/c++/9/bits/os_defines.h
.... /usr/include/features.h
-- removed for brevity --
.. /usr/include/c++/9/ostream
```

目标文件名之后，输出中的每一行都包含到头文件的路径。行首的一个点表示顶层包含 (#include 指令在 hello.cpp 中)。两个圆点表示该文件包含在 <iostream> 中。每一个进一步的点表示另一个嵌套级别。

输出的最后，可能还会发现对代码的改进建议：

```
Multiple include guards may be useful for:
/usr/include/c++/9/clocale
/usr/include/c++/9/cstdio
/usr/include/c++/9/cstdlib
```

不需要修复标准库，但可能会看到自己的一些头文件。有可能需要改正它们。

## 为调试器提供信息

机器码是用二进制格式编码的指令和数据的神秘代码，因为 CPU 并不关心程序的目标是什么或者所有指令的意义是什么。唯一的要求是代码的正确性。编译器将把上述所有信息转换为 CPU 指令的数字标识符、一些初始化内存的数据和数千个内存地址。换句话说，最终的二进制文件不需要包含实际的源代码、变量名、函数签名或程序员关心的任何其他细节。这是编译器的默认输出-

raw 和 dry。

这样做主要是为了节省空间和执行时没有太多的开销。巧合的是，我们也(在某种程度上)保护程序不受逆向工程的影响。是的，可以在没有源代码的情况下理解每个 CPU 指令的作用(例如，将这个整数复制到那个寄存器)，但即使是最基本的程序也包含了太多这样的东西，以至于很难全局性的考虑。

若是一个特别有动力的人，可以使用一个叫做反汇编的工具，有了大量的知识(和一点运气)，就能理解可能发生的事情。这种方法不是很实用，因为反汇编的代码没有原始符号，所以理清代码的位置非常困难和耗时。

相反，可以要求编译器将源代码存储在生成的二进制文件中，以及包含已编译代码和原始代码之间引用的映射。然后，可以将调试器与正在运行的程序挂钩，并查看在任何给定时刻正在执行哪一行源程序。这在我们编写代码时是必不可少的，比如：编写新功能或纠正错误。

这两个用例是两个配置的原因：Debug 和 Release。CMake 会在默认情况下向编译器提供一些标志来管理这个过程，首先将它们存储在全局变量中：

- CMAKE\_CXX\_FLAGS\_DEBUG 包含 -g
- CMAKE\_CXX\_FLAGS\_RELEASE 包含 -DNDEBUG

-g 标志仅仅表示添加调试信息，以操作系统的本机格式提供——stabs、COFF、XCOFF 或 DWARF。这些格式可以用诸如 gdb(GNU 调试器)这样的调试器访问，这对于像 CLion 这样的 IDE 已经足够好了(底层使用 gdb)。其他情况下，请参考所提供的调试器的手册，并检查所选编译器的相应标志是什么。

对于 RELEASE 配置，CMake 将添加-DNDEBUG 标志。这是一个预处理器定义，表明不是调试构建。启用此选项时，一些面向调试的宏可能无法工作。其中之一是 assert，在<assert.h> 头文件中可用。若决定在生产代码中使用断言，它们将完全不起作用：

```
1 int main(void)
2 {
3 bool my_boolean = false;
4 assert(my_boolean);
5 std::cout << "This shouldn't run. \n";
6 return 0;
7 }
```

assert(my\_boolean) 调用在 Release 配置中不会有任何作用，但在 Debug 配置会正常工作。若正在练习断言式编程，但仍然需要在发布版本中使用 assert()，该怎么办？要么改变 CMake 提供的默认值(从 CMAKE\_CXX\_FLAGS\_RELEASE 中移除 NDEBUG)，要么通过在包含头文件之前取消宏的定义：

```
1 #undef NDEBUG
2 #include <assert.h>
```

更多信息请参考 assert 说明文档：<https://en.cppreference.com/w/c/error/assert>

## 5.6. 总结

毫无疑问，编译是一个复杂的过程。对于所有的边缘案例和特定的需求，若没有一个好的工具，管理起来会很困难，CMake 在这方面支持得很好。

到目前为止我们学到了什么？首先讨论了什么是编译，以及在构建和运行操作系统中的应用程序方面的位置。然后，研究了编译的各个阶段，以及管理它们的内部工具。这对于解决在更高级的情况下可能遇到的所有问题非常有用。

然后，研究了如何要求 CMake 验证主机上可用的编译器，是否满足要构建的代码的所有必要要求。对于用户来说，看到要求他们升级的友好消息要比看到老编译器抱怨神秘错误（很容易被语言的新特性弄懵）好的多。

我们简要地讨论了如何向已经定义的目标添加源，然后继续讨论预处理器的配置。这是一个相当大的主题，因为这个阶段将所有的代码集中在一起，并决定将忽略哪些代码。讨论了提供文件的路径，并将自定义定义作为单个参数和批量添加（以及一些用例）。

然后，讨论了优化器，探讨了优化的所有一般级别以及它们所暗示的标志类型，也详细讨论了其中的一些级别——`finline`、`flop-unroll` 和 `ftree-vectorize`。

最后，研究如何管理编译的可行性了。在这里讨论了两个主题——减少编译时间（通过扩展，加强开发者的注意力）和查找错误。后者对于发现什么是坏的，以及如何坏的极其重要。

正确设置工具，并理解事情发生的原因对于确保代码的质量（以及我们的心理健康）非常重要。

下一章中，将学习链接，以及在构建库并在项目中使用它们时需要考虑的事情

### 5.6.1 扩展阅读

- 有关本章所涵盖主题的更多信息，可以参考以下内容：

<https://cmake.org/cmake/help/latest/manual/cmake-compilefeatures.7.html#supported-compilers>

- CMake 目标属性：

– <https://stackoverflow.com/questions/32411963/why-is-cmake-file-glob-evil>  
– [https://cmake.org/cmake/help/latest/command/target\\_sources.html](https://cmake.org/cmake/help/latest/command/target_sources.html)

- 提供包含文件的路径：

– <https://en.cppreference.com/w/cpp/preprocessor/include>  
– [https://cmake.org/cmake/help/latest/command/target\\_include\\_directories.html](https://cmake.org/cmake/help/latest/command/target_include_directories.html)

- 配置头文件：[https://cmake.org/cmake/help/latest/command/configure\\_file.html](https://cmake.org/cmake/help/latest/command/configure_file.html)
- 预编译头文件：[https://cmake.org/cmake/help/latest/command/target\\_preamble\\_headers.html](https://cmake.org/cmake/help/latest/command/target_preamble_headers.html)
- 统一构建：

– [https://cmake.org/cmake/help/latest/prop\\_tgt/UNITY\\_BUILD.html](https://cmake.org/cmake/help/latest/prop_tgt/UNITY_BUILD.html)

- <https://www.qt.io/blog/2019/08/01/precompiled-headers-and-unity-jumbo-builds-in-upcoming-cmake>
- 查找错误-编译器标志: <https://interrupt.memfault.com/blog/best-and-worst-gcc-clang-compiler-flags>
- 为什么使用库, 而不是目标文件: <https://stackoverflow.com/questions/23615282/object-files-vs-library-files-and-why>
- 关注点分离: <https://nalexn.github.io/separation-of-concerns/>

# 第6章 进行链接

有读者可能认为，在我们成功地将源代码编译为二进制文件之后，构建的工作就完成了。差不多吧——二进制文件包含 CPU 要执行的所有代码，但代码以非常复杂的方式分散在多个文件中。链接是一个简化事情的过程，使机器代码简洁、快速地使用。

快速浏览一下指令列表，就会发现 CMake 并没有提供那么多与链接相关的指令。不可否认，`target_link_libraries()` 是真正配置此步骤的指令。那么，为什么要用一整章的篇幅来阐述一条命令呢？在计算机科学中没有容易的东西，链接操作也不例外。

为了获得正确的结果，需要遵循规则——理解链接器的工作原理并掌握正确的知识。我们将讨论目标文件的内部结构，重定位和引用解析如何工作，以及其用途。我们将讨论最终可执行文件与它的组件之间的区别，以及系统如何复刻构建流程。

然后，我们将向您介绍各种库——静态、动态和模块，它们都称为库，但几乎没有什么相似之处。构建正确链接的可执行文件很大程度上依赖于有效的配置（并注意位置无关代码（PIC）等小细节）。

我们将了解链接的另一个麻烦——单一定义规则（ODR），需要让定义的数量完全正确。处理重复的符号有时是非常棘手的，特别是使用动态库时。然后，将了解为什么链接器有时无法找到外部符号，即使在可执行文件与适当的库链接时也是如此。

最后，将了解如何节省时间，并使用链接器来准备使用专用框架进行测试的解决方案。

本章中，我们将讨论以下主题：

- 掌握正确的链接方式
- 构建不同类型的库
- 用定义规则解决问题
- 连接顺序和未定义符号
- 分离 `main()` 进行测试

## 6.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter06>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 6.2. 掌握正确的链接方式

在第 5 章中讨论了 C++ 程序的生命周期，由五个主要阶段组成——编写、编译、链接、加载和执行。正确编译所有源代码之后，需要将它们放到一个可执行文件中。编译过程中产生的目标文件不能由处理器直接执行，这是为什么呢？

为了回答这个问题，就来看看编译器如何用 ELF 格式（类 Unix 系统和许多其他系统使用）构造对象文件的：

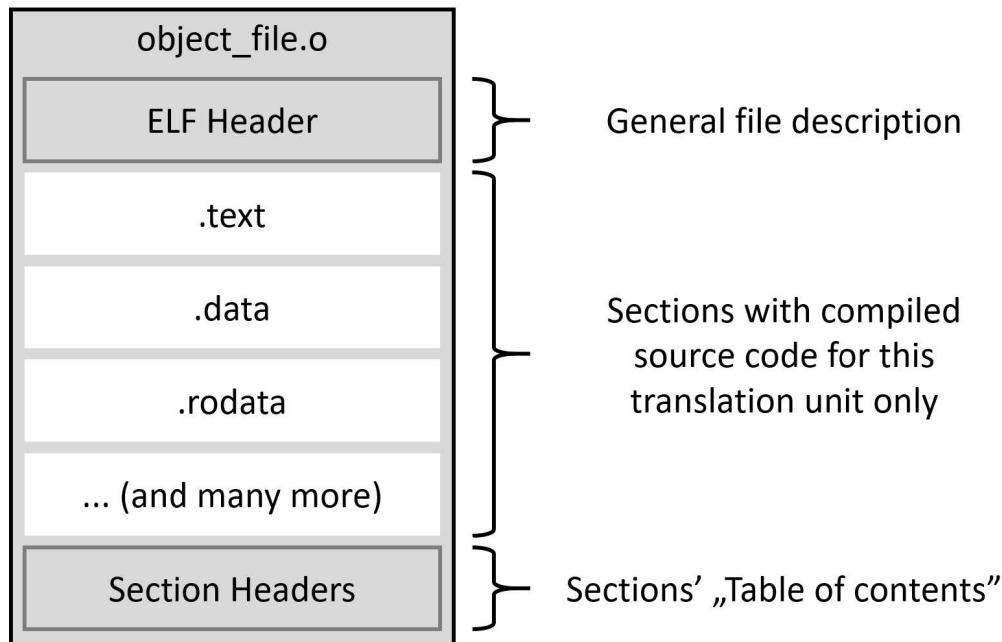


图 6.1 目标文件的结构

编译器将为每个翻译单元（为每个.cpp 文件）准备一个目标文件。这些文件将用于构建程序的内存映像。Object 文件包含以下元素：

- ELF 头标识目标操作系统、ELF 文件类型、目标指令集体系结构，以及 ELF 文件中两个头表的位置和大小信息——程序头表（不存在于目标文件中）和节头表。
- 包含按类型分组的信息的部分（下面将介绍）。
- 节头表，包含关于名称、类型、标志、内存中的目标地址、文件中的偏移量和其他杂项信息，用于了解文件中的哪些部分，以及它们在哪里，就像目录一样。

当编译器处理源代码时，将收集到的信息分组到几个单独的格中，这些格将放在属于它们自己的区段中。其中一些如下：

- .text 区段：机器代码，包含处理器要执行的所有指令
- .data 区段：初始化的全局对象和静态对象（变量）的所有值
- .bss 区段：未初始化的全局对象和静态对象（变量）的所有值，这些值将在程序启动时初始化为零
- .rodata 区段：常量的所有值（只读数据）
- .strtab 区段：一个字符串表，包含所有常量字符串，例如 hello.cpp 示例中的 Hello World
- .shstrtab 区段：包含所有部分名称的字符串表

这些区段非常类似于可执行文件的最终版本，其将放入 RAM 中以运行我们的应用程序，但不能直接将这个文件加载到内存中。因为每个目标文件都有自己的一组区段。若只是把它们连在一起，就会遇到各种各样的问题。将浪费大量的空间和时间，因为需要更多的 RAM 页，指令和数据将很难复制到 CPU 缓存中。整个系统必须要复杂得多，并且会浪费宝贵的 CPU 周期，在运行时跳过许多（可能是数万个）.text、.data 和其他部分。

我们要做的是把对象文件的每个部分，和所有其他对象文件中相同类型的部分放在一起。这个过程称为重定位（这就是 ELF 文件类型为目标文件重定位的原因）。除了将适当的部分组合在一起之外，还必须更新文件中的内部关联——即变量、函数、符号表索引或字符串表索引的地址。所有这些值都是对象文件的本地值，其编号从 0 开始。将文件捆绑在一起时，需要偏移这些值，使它们指向组合文件中的正确地址。

图 6.2 显示了重新定位的过程——重新定位.text，从所有链接的文件构建.data，然后是.rodata 和.strtab（为了简单起见，图中不包含头文件）：

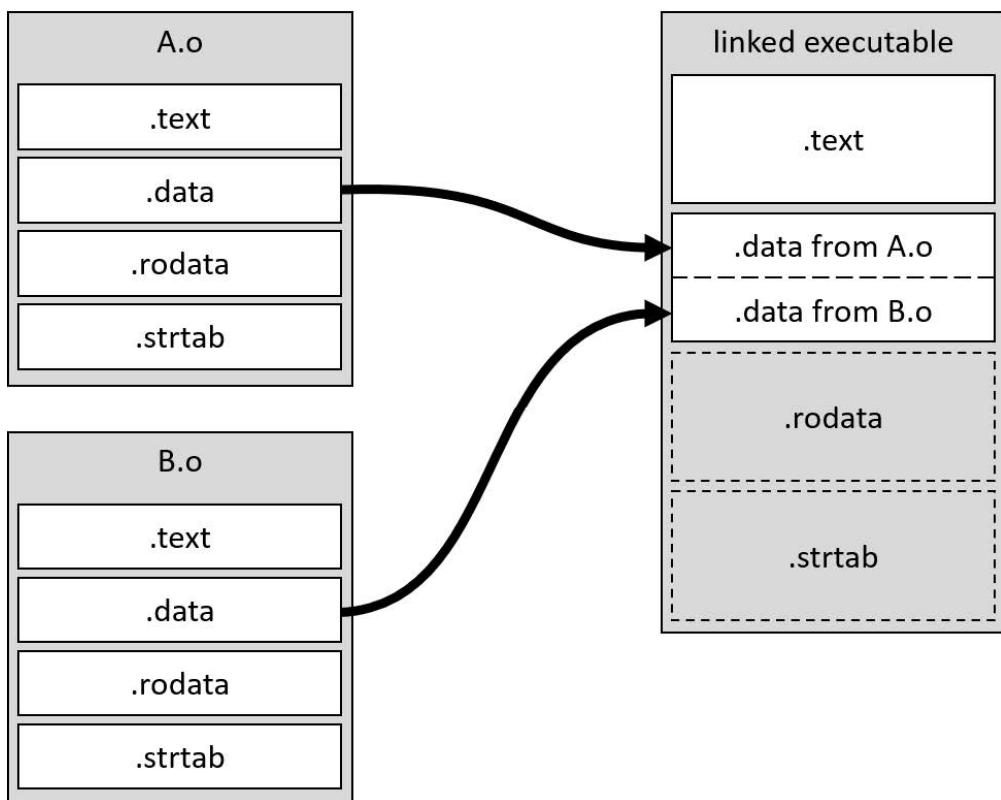


图 6.2 .data 区段的重定位

其次，链接器需要解析引用。每当来自翻译单元的代码引用另一个翻译单元中定义的符号时（例如：通过包含其头或使用 `extern` 关键字），编译器读取该声明并相信定义就在某个地方，稍后将提供该定义。链接器负责收集这些对外部符号的未解析引用，查找并填充它们在合并到可执行文件后驻留的地址。图 6.3 显示了一个简单的引用示例：

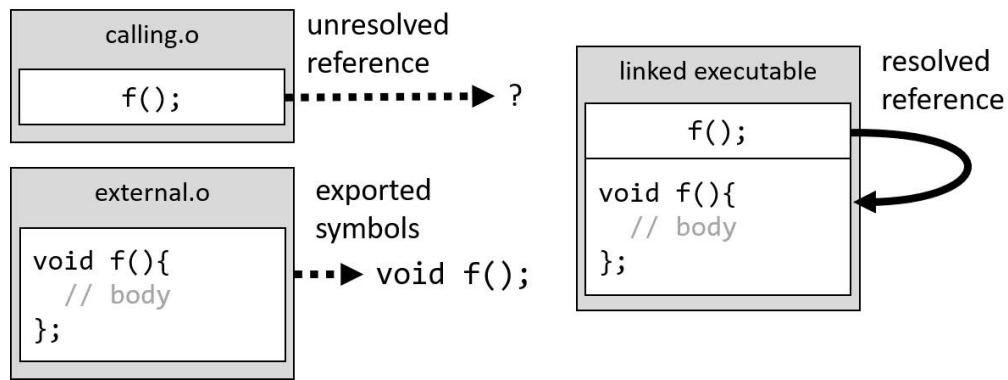


图 6.3 一个引用解析

若开发者不知道其是如何工作的，这部分链接可能是问题的来源。最终可能会得到无法解析的引用，这些引用找不到外部符号，或者恰恰相反——提供了太多的定义，链接器不知道选择哪一个。

最终的可执行文件看起来非常类似于目标文件，包含带有解析引用的重定位区段、区段头表，当然还有描述整个文件的 ELF 头。主要的区别是程序头的存在（如图 6.4 所示）。

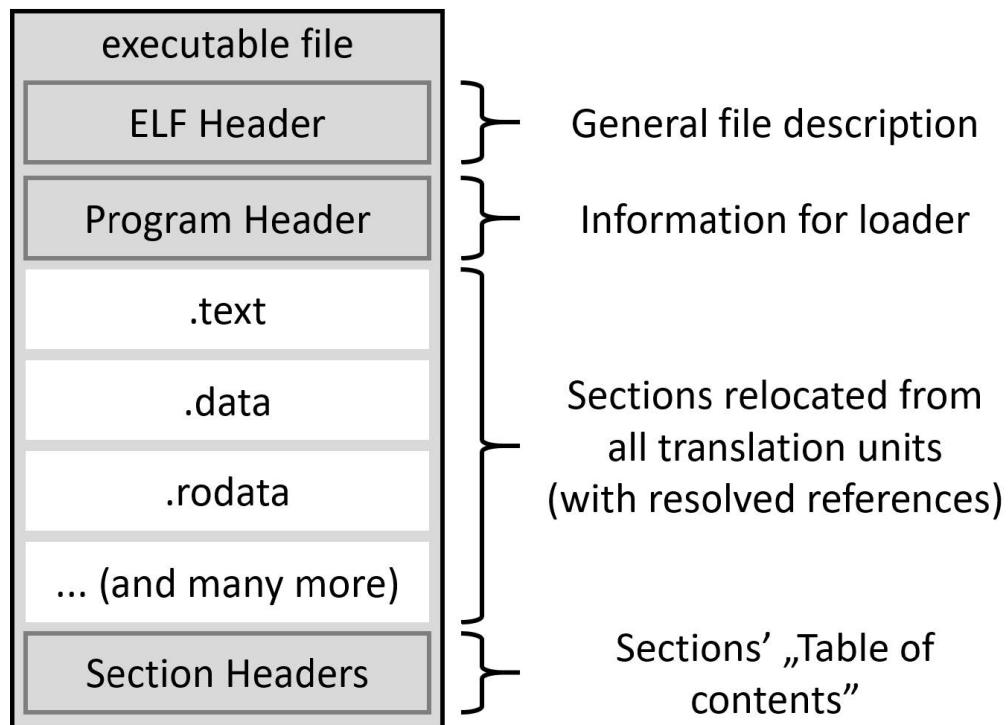


图 6.4 ELF 中可执行文件的结构

程序头放置在 ELF 头的右边，系统加载器将读取此头文件以创建进程映像。其头部会包含一些通用信息和内存布局的描述。布局中的每个条目代表一个称为段的内存片段。条目指定将读取哪些节、以什么顺序、虚拟内存中的哪个地址、标志是什么（读、写或执行），以及其他一些有用的细节。

目标文件也可以捆绑在库中，库是一个中间产品，可以在最终的可执行文件或另一个库中使用。下一节中，我们将讨论三种类型的库。

## 6.3. 构建不同类型的库

编译源代码之后，可能希望避免在同一个平台上再次编译，甚至尽可能地与外部项目共享。当然，可以简单地按照最初创建的方式提供所有目标文件，但这有一些缺点。分发多个文件并将它们单独添加到构建系统比较困难。这可能是一件麻烦的事，尤其是当他们数量众多的时候。相反，可以简单地将所有对象文件放到一个对象中并共享。可以用一个简单的 `add_library()` 指令创建这些库（与 `target_link_libraries()` 一起使用）。按照惯例，所有库都有共同的前缀 `lib`，并使用特定于系统的扩展来表示它们是哪种类型的库：

- 静态库在类 Unix 系统上的后缀为 `.a`，在 Windows 上的后缀为 `.lib`。
- 动态库在类 Unix 系统上的后缀为 `.so`，在 Windows 上的后缀为 `.dll`。

在构建库（静态、动态或模块）时，经常会遇到此过程的名称链接。甚至 CMake 在 chapter06/01-libraries 项目的构建输出中也这样称呼它：

```
[33%] Linking CXX static library libmy_static.a
[66%] Linking CXX shared library libmy_shared.so
[100%] Linking CXX shared module libmy_module.so
[100%] Built target module_gui
```

与看起来不同的是，链接器并不是用来创建前面所有的库的。执行重定位和引用解析也有例外。来看看每种库类型，了解它们的工作原理。

### 6.3.1 静态库

要构建一个静态库，可以简单地使用我们在前几章中看到的指令：

```
add_library(<name> [<source>...])
```

若 `BUILD_SHARED_LIBS` 变量没有设置为 `ON`，前面的代码将生成一个静态库。若想要构建一个静态库，可以提供一个关键字：

```
add_library(<name> STATIC [<source>...])
```

什么是静态库？本质上是存储在存档中的原始对象文件的集合。类 Unix 系统上，可以通过 `ar` 工具创建此类库。静态库是提供代码编译版本的最古老和最基本的机制。若想避免将依赖关系与可执行文件分离，就可以使用，因为可执行文件的大小和使用的内存会增加。

静态库可能包含一些额外的索引，以加快最终的链接过程。每个平台都使用自己的方法来生成这些内容。类 Unix 系统使用 `ranlib` 工具来实现。

### 6.3.2 动态库

可以用 `SHARED` 关键字构建动态库：

```
add_library(<name> SHARED [<source>...])
```

也可以通过设置 `BUILD_SHARED_LIBS` 变量为 `ON` 并使用短版本来实现:

```
add_library(<name> [<source>...])
```

与静态库的区别很明显。动态库是使用链接器构建的，将执行链接的两个阶段。所以将收到一个具有适当的区段头、区段和区段头表的文件(图 6.1)。

动态库(也称为共享对象)可以在多个不同的应用程序之间共享。一个操作系统会将这样一个库的单个实例与第一个使用它的程序一起加载到内存中，所有随后启动的程序都将提供相同的地址(这要感谢虚拟内存的复杂机制)。只有`.data` 和`.bss` 段将分别为使用库的每个进程创建(以便每个进程可以修改自己的变量而不影响其他使用者)。

由于采用了这种方法，系统中的整体内存使用更好了。若正在使用一个非常流行的库，可能不需要将它与程序一起发布。它很可能已经在目标机器上可用。但若不是这样，则希望用户在运行应用程序之前显式地安装它。当库的安装版本与预期版本不同时，这就有可能出现一些问题(这种类型的问题称为依赖地狱，更多信息可以在扩展阅读部分找到)。

### 6.3.3 模块库

要构建模块库，需要使用 `MODULE` 关键字:

```
add_library(<name> MODULE [<source>...])
```

这是动态库的一种，目的是作为运行时加载的插件使用，而不是在编译过程中链接到可执行文件。共享模块不会在程序启动时自动加载(像普通的共享库一样)。只有程序通过 `LoadLibrary`(Windows) 或 `dlopen()`/`dlsym()`(Linux/macOS) 等系统调用显式地请求它时，才会使用的到。

不应该尝试将可执行文件与模块链接起来，因为这并不能保证在所有平台上都能工作。若需要这样做，请使用动态库。

### 6.3.4 位置无关的代码

所有共享库和模块的源代码都应该在编译时启用位置无关代码标志。`CMake` 检查目标的 `POSITION_INDEPENDENT_CODE` 属性，并适当添加特定于编译器的编译标志，如 `gcc` 或 `clang` 的`-fPIC`。

`PIC` 是一个有点令人困惑的术语。现在，程序在某种意义上已经是位置无关的，因为使用虚拟内存抽象出实际的物理地址。当使用函数时，CPU 使用内存管理单元(MMU) 将虚拟地址(每个进程从 0 开始) 转换为分配时可用的物理地址。这些映射不必指向连续的物理地址或遵循任何其他特定的顺序。

`PIC` 是关于将符号(对函数和全局变量的引用)映射到其运行时地址。编译库期间，不知道哪些进程可能会使用。不可能预先确定库将在虚拟内存中的什么位置或以什么顺序加载。所以符号的地址是未知的，与库机器码的相对位置也是未知的。

为了解决这个问题，需要另一种间接方式。`PIC` 将向我们的输出添加一个新部分——全局偏移表(GOT)。最终，这个部分将成为包含共享库所需的所有符号的运行时地址的段。`GOT` 相对于`.text` 段的位置在链接过程中是已知的；因此，所有符号引用都可以(通过偏移量)指向当时的占位符`get`。

只有在第一次执行访问引用符号的指令时，内存中指向符号的实际值才会填充。这时，加载器将在 GOT 中设置该特定条目（这就是术语延迟加载的来源）。

动态库和模块的 POSITION\_INDEPENDENT\_CODE 属性会自动设置为 ON。然而，重要的是要记住，若动态库链接到另一个目标，例如静态库或对象库，那么也需要在该目标上设置此属性。方法如下：

```
set_target_properties(dependency_target
 PROPERTIES POSITION_INDEPENDENT_CODE
 ON)
```

若不这样做，就会在使用 CMake 时遇到麻烦，因为默认情况下，这个属性会按照第 4 章中描述的方式检查冲突。

说到符号，还有一个问题要讨论。下一节将讨论导致定义歧义和名称冲突。

## 6.4. 用定义规则解决问题

Phil Karlton 说得对：

在计算机科学中有两件难事：缓存失效和命名。

起名字很难有几个原因——必须精确、简单、简短，同时还要有表现力。这使它们变得有意义，并允许程序员理解原始实现背后的概念。C++ 和许多其他语言强加了另外一个要求——许多名称必须唯一。这体现在几个不同的方面，例如：开发者需要遵循 ODR。

在单个翻译单元（单个.cpp 文件）的范围内，即使多次声明相同的名称（变量、函数、类类型、枚举、概念或模板），也需要只定义一次。

对于在代码和非内联函数中有效使用的所有变量，该规则扩展到整个程序的作用域。考虑下面的例子：

```
1 // chapter06/02-odr-fail/shared.h
2
3 int i;
```

```
1 // chapter06/02-odr-fail/one.cpp
2
3 #include <iostream>
4 #include "shared.h"
5 int main() {
6 std::cout << i << std::endl;
7 }
```

```
1 // chapter06/02-odr-fail/two.cpp
2
3 #include "shared.h"
```

```
chapter06/02-odr-fail/two.cpp
```

```
cmake_minimum_required(VERSION 3.20.0)
```

```
project(ODR CXX)
set(CMAKE_CXX_STANDARD 20)
add_executable(odr one.cpp two.cpp)
```

我们创建了一个 shared.h 头文件，用于两个独立的翻译单元：

- one.cpp，只是把 i 打印到屏幕上
- two.cpp，除了包含头文件外什么都不做

然后将两者链接到一个可执行文件中，会看到以下错误：

```
[100%] Linking CXX executable odr
/usr/bin/ld: CMakeFiles/odr.dir/two.cpp.o:(.bss+0x0): multiple
definition of 'i'
; CMakeFiles/odr.dir/one.cpp.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

这些东西不能定义两次。但有一个明显的例外——若类型、模板和 `extern` 内联函数完全相同（具有相同的标记序列），则可以在多个翻译单元中重复其定义。可以通过用类的定义替换简单的定义 `int i;` 来证明这一点：

```
1 // chapter06/03-odr-success/shared.h
2
3 struct shared {
4 static inline int i = 1;
5 }
```

然后，可以这样使用：

```
1 // chapter06/03-odr-success/one.cpp
2
3 #include <iostream>
4 #include "shared.h"
5 int main() {
6 std::cout << shared::i << std::endl;
7 }
```

其他两个文件 two.cpp 和 CMakeLists.txt 与 02odrfail 示例中一样。这样的更改将链接成功：

```
-- Build files have been written to: /root/examples/
chapter06/03-odr-success/b
[33%] Building CXX object CMakeFiles/odr.dir/one.cpp.o
[66%] Building CXX object CMakeFiles/odr.dir/two.cpp.o
[100%] Linking CXX executable odr
[100%] Built target odr
```

或者，可以将变量标记为翻译单元的局部变量(不会被导出到目标文件之外)。为此，可以使用 `static` 关键字，如下所示：

```
1 // chapter06/04-odr-success/shared.h
2
3 static int i;
```

所有其他文件将保持相同，就像在原始示例中一样，并且链接仍然会成功。当然，前面代码中的变量存储在每个翻译单元的单独内存中，对其中一个的更改不会影响另一个。

#### 6.4.1 动态链接的重复符号

ODR 规则对静态库的作用和对目标文件的作用完全相同，但是当用动态库构建代码时，事情就不同了。链接器允许在这里复制符号。下面的示例中，将创建两个动态库 A 和 B，其中有一个 `duplicated()` 函数和两个 `A()` 和 `B()` 函数：

```
1 //chapter06/05-dynamic/a.cpp
2
3 #include <iostream>
4 void a() {
5 std::cout << "A" << std::endl;
6 }
7 void duplicated() {
8 std::cout << "duplicated A" << std::endl;
9 }
```

第二个实现文件几乎完全复制了第一个：

```
1 // chapter06/05-dynamic/b.cpp
2
3 #include <iostream>
4 void b() {
5 std::cout << "B" << std::endl;
6 }
7 void duplicated() {
8 std::cout << "duplicated B" << std::endl;
9 }
```

现在，使用每个函数来看看发生了什么(为了简单起见，将使用 `extern` 在本地声明它们)：

```
1 // chapter06/05-dynamic/main.cpp
2
3 extern void a();
4 extern void b();
5 extern void duplicated();
6
7 int main() {
8 a();
9 b();
10 duplicated();
11 }
```

前面的代码将运行来自每个库的函数，然后使用在两个动态库中使用相同签名定义的函数。你觉得会发生什么？这种情况下，连接顺序重要吗？来针对两种情况进行测试：

- main\_1 与 a 库链接
- main\_2 与 b 库链接

下面是代码：

```
// chapter06/05-dynamic/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Dynamic CXX)

add_library(a SHARED a.cpp)
add_library(b SHARED b.cpp)

add_executable(main_1 main.cpp)
target_link_libraries(main_1 a b)

add_executable(main_2 main.cpp)
target_link_libraries(main_2 b a)
```

构建并运行这两个可执行程序之后，将看到以下输出：

```
root@ce492a7cd64b:/root/examples/chapter06/05-dynamic# b/main_1
A B
duplicated A
root@ce492a7cd64b:/root/examples/chapter06/05-dynamic# b/main_2
A
B
duplicated B
```

啊哈！因此，链接器确实关心链接库的顺序。若不小心的话，这可能会造成一些混乱。实际上，命名冲突并不像看上去那么罕见。

这种行为也有一些例外。若定义本地可见的符号，将优先于从动态链接库中可用的符号。向 main.cpp 中添加以下函数会将两个二进制文件的最后一行输出更改为“duplicated MAIN”，如下所示：

```
1 #include <iostream>
2 void duplicated() {
3 std::cout << "duplicated MAIN" << std::endl;
4 }
```

从库导出名称时一定要非常小心，可能会遇到名称冲突。

#### 6.4.2 使用命名空间——不要依赖链接器

命名空间的概念是为了避免这些奇怪的问题，并以可管理的方式处理 ODR。建议将库代码包装在以库命名的命名空间中，这一点并不奇怪。这样，就可以避免所有符号重复的问题。

项目中可能会遇到这样的情况：一个动态库连接另一个动态库，然后连接另一个动态库，形成一个长长的依赖链。这并不罕见，特别是在更复杂的设置中。重要的是要记住，简单地将一个库链接到另一个库并不意味着类型的命名空间继承。这个链的每个链接中的符号仍然不受保护，保存在最初编译它们的命名空间中。

某些情况下，链接器的怪癖是有趣和有用的。接下来，让我们讨论一个不常见的问题——当正确定义的符号丢失而没有解释时，该怎么办？

### 6.5. 连接顺序和未定义符号

链接器经常会异想天开，毫无理由地抱怨事情。对于刚开始使用这个工具的开发者来说，这是一项考验。这并不奇怪，因为他们通常尽可能地避免接触构建配置。最终，会在可执行文件中更改某些内容（可能添加他们正在处理的库），然后一切就失控了。

考虑一个相当简单的依赖链——主可执行程序依赖于外部库，而外部库依赖于嵌套库（包含必要的 int b 变量）。突然，开发者的屏幕上出现了一条不显眼的消息：

```
outer.cpp:(.text+0x1f): undefined reference to 'b'
```

这并不是一个罕见的问题，这是忘记向链接器添加必要的库。但在本例中，库实际上已经正确地添加到 target\_link\_libraries() 中：

```
chapter06/06-order/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Order CXX)

add_library(outer outer.cpp)
add_library(nested nested.cpp)

add_executable(main main.cpp)
target_link_libraries(main nested outer)
```

那么！？很少有错误会像调试和理解那样令人恼火，在这里看到的是一个错误的连接顺序。让我们深入源代码来找出原因：

```
1 // chapter06/06-order/main.cpp
2
3 #include <iostream>
4 extern int a;
5 int main() {
```

```
6 std::cout << a << std::endl;
7 }
```

前面的代码看起来很简单——打印一个外部变量，可以在外部库中找到，提前用 `extern` 关键字声明了它。下面是该库的源代码：

```
1 // chapter06/06-order/outer.cpp
2
3 extern int b;
4 int a = b;
```

这也非常简单——`outer` 依赖于嵌套库提供外部变量 `b`，该变量赋值给导出变量 `a`。让我们看看嵌套的源代码，以确认没有遗漏定义：

```
1 // chapter06/06-order/nested.cpp
2
3 int b = 123;
```

实际上，已经为 `b` 提供了定义，由于它没有使用 `static` 关键字标记为本地，因此它从嵌套目标正确导出。正如之前看到的，这个目标与 `CMakeLists.txt` 中的主要可执行文件相链接：

```
target_link_libraries(main nested outer)
```

那么，对'`b`' 错误的未定义引用是从哪里来的呢？

解析未定义符号的工作方式是这样的——链接器从左向右处理二进制文件。当链接器遍历二进制文件时，将执行以下操作：

1. 收集从此二进制文件导出的所有未定义符号，并存储起来以备以后使用
2. 尝试用此二进制文件中定义的符号解析未定义的符号(从处理的二进制文件中收集)
3. 对下一个二进制文件重复此过程

若在整个操作完成后，若有符号仍未定义，则链接失败。

我们的例子中就是这样(`CMake` 把可执行目标的库放在库之前)：

1. 主要处理 `main.o`，获得对 `a` 的未定义引用，并收集它以用于将来的解析。
2. 处理 `libnested.a`，没有发现未定义的引用，因此没有什么需要解析的。
3. 处理 `libouter.a`，得到了对 `b` 的未定义引用，并解析了对 `A` 的引用。

我们确实正确地解析了对 `a` 变量的引用，但对 `b` 没有。需要做的是颠倒链接的顺序，使 `nested` 排在 `outer` 之后：

```
target_link_libraries(main outer nested)
```

另一个不那么优雅的选项是重复库(这对循环引用很有用)：

```
target_link_libraries(main nested outer nested)
```

最后，可以尝试使用特定于链接器的标志，如`--start-group` 或`--end-group`。详细信息请参阅链接器的文档，这些细节超出了本书的范围。

现在我们知道了如何解决常见的问题后，再来看看如何使用链接器。

## 6.6. 分离 main() 进行测试

链接器会执行 ODR，并确保所有外部符号在链接过程中提供的定义。我们可能会遇到的问题是，构建的正确测试。

理想情况下，应该测试在生产环境中运行的完全相同的源代码。详尽的测试流水应该构建源代码，生成的二进制文件上运行测试，然后打包并分发可执行文件（不包含测试本身）。

但要如何做到这一点呢？可执行程序有一个非常特定的执行流，这通常需要读取命令行参数。C++ 的编译特性并不真正支持可插入单元，这些可插入单元可以临时注入到二进制文件中，仅用于测试目的。看来我们需要一个非常复杂的方法来解决这个问题。

幸运的是，链接器可以以一种优雅的方式处理这个问题。考虑从程序的 main() 提取所有逻辑到一个外部函数，start\_program()：

```
1 // chapter06/07-testing/main.cpp
2
3 extern int start_program(int, const char**);
4 int main(int argc, const char** argv) {
5 return start_program(argc, argv);
6 }
```

现在跳过测试这个新的 main() 函数是合理的，只是将参数转发到别处定义的函数（在另一个文件中）。然后，可以创建一个库，其中包含包装在新函数 start\_program() 中的 main() 原始源代码。这个例子中，将使用一个简单的程序来检查命令行参数 count 是否大于 1：

```
1 // chapter06/07-testing/program.cpp
2
3 #include <iostream>
4 int start_program(int argc, const char** argv) {
5 if (argc <= 1) {
6 std::cout << "Not enough arguments" << std::endl;
7 return 1;
8 }
9 return 0;
10 }
```

现在可以准备一个项目来构建这个应用程序，并将这两个翻译单元链接在一起：

```
chapter06/07-testing/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Testing CXX)

add_library(program program.cpp)

add_executable(main main.cpp)
target_link_libraries(main program)
```

主要目标只是提供所需的 main() 函数，包含所有逻辑的程序目标。现在，可以通过创建另一个可执行文件，以及包含测试逻辑的 main() 来测试它。

现实场景中，GoogleTest 或 Catch2 等框架会提供 main()，可用于替换程序的入口点并运行所有已定义的测试。我们将在第 8 章中深入研究实际测试。现在，让我们专注于原理，并在另一个 main() 函数中编写自己的测试：

```
1 // chapter06/07-testing/test.cpp
2
3 #include <iostream>
4 extern int start_program(int, const char**);
5 using namespace std;
6 int main() {
7 auto exit_code = start_program(0, nullptr);
8 if (exit_code == 0)
9 cout << "Non-zero exit code expected" << endl;
10
11 const char* arguments[2] = {"hello", "world"};
12 exit_code = start_program(2, arguments);
13 if (exit_code != 0)
14 cout << "Zero exit code expected" << endl;
15 }
```

前面的代码将调用 start\_program 两次，带参数和不带参数，并检查返回的退出代码是否正确。干净的代码和优雅的测试实践方面，这个单元测试还有很多需要改进的地方，但至少这是一个开始。重要的是我们已经两次定义了 main()：

1. 在 main.cpp 中用于生产使用
2. 在 test.cpp 中进行测试

现在将把第二个可执行文件添加到 CMakeLists.txt 的底部：

```
add_executable(test test.cpp)
target_link_libraries(test program)
```

这将创建另一个目标，该目标与生产对象完全相同的二进制代码相链接，可以自由地调用所有导出函数。得益于此，我们可以自动运行所有代码路径，并检查它们是否按预期工作。太棒了！

## 6.7. 总结

CMake 中的链接看起来简单而微不足道，但它比看上去的要复杂得多。毕竟，链接可执行文件并不像把拼图拼在一起那么简单。当了解目标文件和库的结构后，会发现在程序可运行之前，需要移动一些东西。这些东西称为区段，在程序的生命周期中具有不同的作用——存储不同类型的数据、指令、符号名称等。链接器需要将它们组合到最终的二进制文件中。这个过程称为迁移。

还需要注意符号——跨所有翻译单元解析引用，并确保没有遗漏。然后，链接器可以创建程序头，并将其添加到最终的可执行文件中。它将包含针对系统加载器的指令，描述如何将合并的段转换为组成进程的运行时内存映像的区段。

我们还讨论了三种不同类型的库(静态、动态和模块),并解释了它们的区别以及哪些场景更适合某些场景。还讨论了 PIC——这是一个支持符号的惰性绑定的强大概念。

ODR 是一个 C++ 概念,在很大程度上由链接器执行。介绍了这个主题之后,简要地探讨了如何在静态库和动态库中处理最基本的符号复制。接下来是一些简短的建议,尽可能使用命名空间,在防止符号冲突方面不要过度依赖链接器。

对于这样一个看似简单的步骤(CMake 只提供了几个专门用于链接器的指令),确实有很多怪癖!要正确处理的一个棘手的事情是链接的顺序,特别是当库有嵌套依赖时。现在知道了如何处理一些基本的情况,以及可以研究其他方法来处理更复杂的情况。

最后,研究了如何利用链接器为我们的程序准备测试——通过将 main() 函数分离到另一个翻译单元。这能够引入另一个可执行文件,针对将在生产中运行的完全相同的机器代码运行测试。

现在了解了如何链接,可以检索外部库并在 CMake 项目中使用它们。下一章中,将学习如何在 CMake 中管理依赖项。

## 6.8. 扩展阅读

有关本章所涵盖主题的更多信息,请参阅以下连接:

- ELF 文件的结构:

[https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Formats](https://en.wikipedia.org/wiki/Executable_and_Linkable_Formats)

- CMake 手册的 add\_library() 页:

[https://cmake.org/cmake/help/latest/command/add\\_library.html](https://cmake.org/cmake/help/latest/command/add_library.html)

- 依赖地狱:

[https://en.wikipedia.org/wiki/Dependency\\_hell](https://en.wikipedia.org/wiki/Dependency_hell)

- 模块和动态库之间的区别:

<https://stackoverflow.com/questions/4845984/differencebetween-modules-and-shared-libraries>

# 第 7 章 管理依赖关系

解决方案是大是小并不重要。随着它的成熟，将最终决定引入外部依赖项。避免使用主流业务逻辑创建和维护代码的成本是很重要的。这样，就可以把时间花在对你和客户都重要的事情上。

外部依赖不仅用于提供框架和特性，还用于解决古怪的问题。无论是以特殊编译器（如 Protobuf）的形式，还是以测试框架（如 GTest）的形式，还可以在构建和控制代码质量的过程中发挥重要作用。

无论使用的是开源项目，还是使用公司中其他开发人员编写的项目，仍然需要一个良好的、干净的过程来管理外部依赖关系。自己解决这个问题将需要无数小时的设置和大量额外的支持工作。幸运的是，CMake 在适应依赖管理的不同风格和历史方法方面做得很好，同时跟上行业标准的不断发展。

要提供一个外部依赖项，应该首先检查主机系统是否已经有这个依赖项可用，因为最好避免不必要的下载和冗长的编译。我们将探索如何找到并将这些依赖项转换为 CMake 目标，以便在项目中使用。这可以通过多种方式实现，特别是当包支持开箱即用 CMake 或支持稍老的 PkgConfig 工具。不过，仍然可以编写自己的文件来检测并包含这样的依赖项。

我们将讨论当系统上没有依赖项时该怎么办？可以采取其他步骤来自动提供必要的文件。将考虑使用不同的 Git 方法来解决这个问题，并将整个 CMake 项目作为构建的一部分。

本章中，我们将讨论以下主题：

- 如何查找已安装的软件包
- 使用 FindPkgConfig0 发现遗留包
- 编写自己的查找模块
- 使用 Git 库
- 使用 ExternalProject 和 FetchContent 模块

## 7.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter07>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 <build tree> 和 <source tree>。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 7.2. 如何查找已安装的软件包

假设已经决定通过网络通信或静态存储数据来提高你的游戏水平，纯文本文件、JSON，甚至老式的 XML 都不行。所以希望将数据直接序列化为二进制格式，最好使用业界非常熟悉的库——例

如，Google 的协议缓冲区 (Protobuf)。找到文档，在系统中安装依赖项，然后呢？如何让 CMake 找到并引入的这个外部依赖呢？幸运的是，有一个 `find_package()` 指令。

让我们倒回去，从设置场景开始——必须安装要使用的依赖项，因为是 `find_package()` 嘛，顾名思义，只是关于在系统中发现包。假设已经安装了依赖项，或者解决方案的用户知道如何在提示时安装特定的、必要的依赖项。为了涵盖其他场景，需要提供一个备份计划（在后面的章节中可以找到更多关于备份计划的信息）。

对于 Protobuf，情况相当简单：可以从官方存储库 (<https://github.com/protocolbuffers/protobuf>) 中下载、编译和安装库，或者在操作系统中使用包管理器。若正在使用第 1 章中提到的 Docker 映像来执行这些示例，那么可以使用 Debian Linux。Protobuf 库和编译器的安装命令如下：

```
$ apt update
$ apt install protobuf-compiler libprotobuf-dev
```

每个系统都有自己的安装和管理包的方法。寻找包所在的路径可能是棘手和耗时的，特别是希望支持当前使用的大多数操作系统时。`find_package()` 通常可以帮你做到这一点，若有关的包提供了一个适当的配置文件，允许 CMake 确定支持包所需的变量。

现在，许多项目都与此需求兼容，并在安装过程中为 CMake 提供此文件。若计划使用一些不提供此功能的流行库，请先不要担心。很有可能 CMake 作者已经将该文件与 CMake 本身捆绑在一起（这些称为查找模块，以区别配置文件）。若情况并非如此，仍然其他选择：

- 为特定的包提供查找模块，并将其捆绑到我们的项目中。
- 编写一个配置文件，并要求包维护人员将包一起发布。

可能还没有完全准备好自己创建这样的合并请求，没关系，因为不需要这样做。CMake 附带超过 150 个查找模块，可以找到诸如 Boost、bzip2、curl、curses、GIF、GTK、iconv、ImageMagick、JPEG、Lua、OpenGL、OpenSSL、PNG、PostgreSQL、Qt、SDL、Threads、XML-RPC、X11 和 zlib 等库，还可以找到我们将在本例中使用的 Protobuf 文件。完整的列表可以在 CMake 文档中找到：[https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html#find\\_modules](https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html#find_modules)。

查找模块和配置文件都可以用 `find_package()` 指令在 CMake 项目中使用。CMake 查找匹配的查找模块，若找不到，就会查找配置文件。搜索将从存储在 `CMAKE_MODULE_PATH` 变量（默认为空）中的路径开始。当项目想要添加和使用外部查找模块时，可以配置此变量。接下来，CMake 将扫描已安装版本中可用的内置查找模块列表。

若没有找到适用的模块，就应该搜索相应的包配置文件。CMake 有一长串适合主机操作系统的路径，可以查找到符合以下模式的文件名：

- <CamelCasePackageName>Config.cmake
- <kebab-case-package-name>-config.cmake

接下来，来聊聊项目文件。本例中，并不打算设计一个具有远程过程调用和基于网络的解决方案。相反，只是想证明我可以构建和运行一个依赖于 Protobuf 的项目。为此，将创建一个.proto 文件，其中包含尽可能小的契约。若不是很熟悉 Protobuf，只需要知道这个库提供了一种以二进制形

式序列化结构化数据的机制。为此，需要提供这种结构的模式，用于从二进制形式向 C++ 对象进行读写。

这是我想到的：

```
1 // chapter07/01-find-package-variables/message.proto
2
3 syntax = "proto3";
4 message Message {
5 int32 id = 1;
6 }
```

若不熟悉 Protobuf 语法也不用担心（这不是这个例子真正的内容）。这只是一个简单的 Message，只包含一个 32 位整数。Protobuf 有一个特殊的编译器，可以读取这些文件并生成 C++ 源文件和头文件，然后应用就可以使用这些文件了。所以需要以某种方式将这个编译步骤添加到流程中。现在，让我们看看 main.cpp：

```
1 //chapter07/01-find-package-variables/main.cpp
2
3 #include "message.pb.h"
4 #include <fstream>
5 using namespace std;
6 int main()
7 {
8 Message m;
9 m.set_id(123);
10 m.PrintDebugString();
11 fstream fo("./hello.data", ios::binary | ios::out);
12 m.SerializeToOstream(&fo);
13 fo.close();
14 return 0;
15 }
```

Message 包含一个 id 字段。在 main.cpp 文件中，我创建了一个表示此消息的对象，将字段设置为 123，并将其调试信息打印到标准输出。接下来，我将创建一个文件流，将该对象的二进制版本写入其中，然后关闭流——这是序列化库最简单的用法。

注意，这里包含了一个 message.pb.h 头。这个文件还不存在，需要在 message.proto 编译期间由 protoc，即 Protobuf 编译器创建。听起来有些复杂，因为这样一个项目的列表文件一定非常长。其实一点也不！这就是 CMake 魔法发生的地方：

```
chapter07/01-find-package-variables/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(FindPackageProtobufVariables CXX)

find_package(Protobuf REQUIRED)
protobuf_generate_cpp(GENERATED_SRC GENERATED_HEADER
 message.proto)
```

```

add_executable(main main.cpp
 ${GENERATED_SRC} ${GENERATED_HEADER})
target_link_libraries(main PRIVATE ${Protobuf_LIBRARIES})
target_include_directories(main PRIVATE
 ${Protobuf_INCLUDE_DIRS}
 ${CMAKE_CURRENT_BINARY_DIR})

```

让我们来分析一波:

- 前两行我们已经知道了，创建项目并声明其语言
- `find_package(Protobuf REQUIRED)` 要求 CMake 运行绑定的 `FindProtobuf.cmake` 建立了 `Protobuf` 库。该查找模块将扫描常用路径，若没有找到库，则终止（因为提供了 `REQUIRED` 关键字）。其还将指定有用的变量和函数（例如下一行中的函数）
- `protobuf_generate_cpp` 是在 `protobuf` 查找模块中定义的自定义函数。在底层调用 `add_custom_command()`，后者使用适当的参数调用协议编译器。可以通过提供两个变量来使用这个函数，将生成的源文件 (`GENERATED_SRC`) 和头文件 (`GENERATED_HEADER`) 的路径，以及要编译的文件列表 (`message.proto`)。
- `add_executable`，将使用 `main.cpp` 和前面指令中配置的 `Protobuf` 文件创建可执行文件。
- `target_link_libraries` 会将 `find_package()` 找到的库（静态或动态）添加到主目标的链接指令中。
- `target_include_directories()` 添加包提供的必要的 `INCLUDE_DIRS` 和 `CMAKE_CURRENT_BINARY_DIR`。这里需要后者，这样编译器才能找到生成的 `message.pb.h` 头文件。

换句话说，其实现了以下目标：

- 查找库和编译器的位置
- 提供帮助函数来教 CMake 如何调用 `.proto` 文件的自定义编译器
- 添加包含包含和链接所需路径的变量

当使用 `find_package()` 时，可以预期会设置一些变量，无论使用的是内置的查找模块，还是与包绑定的包配置文件（假设找到了包）：

- `<PKG_NAME>_FOUND`
- `<PKG_NAME>_INCLUDE_DIRS` 或 `<PKG_NAME>_INCLUDES`
- `<PKG_NAME>_LIBRARIES` 或 `<PKG_NAME>_LIBRARIES` 或 `<PKG_NAME>_LIBS`
- `<PKG_NAME>_DEFINITIONS`
- `IMPORTED` 由查找模块或配置文件指定的目标

最后一点非常有趣——若包支持所谓的“现代 CMake”（围绕目标构建），其将提供那些导入的目标来代替（或伴随）这些变量，这允许更干净、更简单的代码。建议优先考虑目标而不是变量。

`Protobuf` 是一个很好的例子，因为提供了变量和导入的目标（自 CMake 3.10 以来）：`:protobuf::libprotobuf`, `:protobuf::libprotobuf-lite`, `:protobuf::libprotoc` 和 `:protobuf::protoc`。这样，就可以编写更简洁的代码：

```

chapter07/02-find-package-targets/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(FindPackageProtobufTargets CXX)

find_package(Protobuf REQUIRED)
protobuf_generate_cpp(GENERATED_SRC GENERATED_HEADER
 message.proto)

add_executable(main main.cpp
 ${GENERATED_SRC} ${GENERATED_HEADER})

target_link_libraries(main PRIVATE protobuf::libprotobuf)
target_include_directories(main PRIVATE
 ${CMAKE_CURRENT_BINARY_DIR})

```

导入的目标 `protobuf::libprotobuf` 隐式指定包含目录，并且由于传递依赖项（我称之为传播属性），它们与我们的主目标共享。同样的过程也发生在链接器和编译器标志上。

若知道特定的查找模块具体提供了什么，最好访问其在线文档。Protobuf 的文档可以在这里找到：<https://cmake.org/cmake/help/latest/module/FindProtobuf.html>。

#### 重要的 Note

简单起见，若在用户系统中找不到 `protobuf` 库（或其编译器），本节中的示例将会失败。但是真正健壮的解决方案应该通过检查 `Protobuf_FOUND` 变量并进行相应的操作来验证，或者为用户打印明确的诊断消息（这样用户就可以安装它），或者自动执行安装。

`find_package()` 指令最后要提到的是其选项。完整的列表有点冗长，所以这里只关注基本签名。看起来像这样：

```
find_package(<Name> [version] [EXACT] [QUIET] [REQUIRED])
```

最重要的选项如下：

- `[version]`，可以选择地请求特定的版本。使用 `major.minor.patch.tweak` 格式（例如 `1.22`）或提供一个范围- `1.22…1.40.1`（使用三个圆点作为分隔符）。
- `EXACT` 关键字需要精确的版本（这里不支持范围）。
- `QUIET` 关键字使所有关于已找到/未找到包的消息静默。
- 若没有找到包，`REQUIRED` 关键字将停止执行并打印诊断消息（即使启用了 `QUIET`）。

关于该指令的更多信息可以在文档页面中找到：[https://cmake.org/cmake/help/latest/command/find\\_package.html](https://cmake.org/cmake/help/latest/command/find_package.html)。

为构建系统自动使用的包提供配置文件的概念并不新鲜。当然这也不是 CMake 的发明，还有其他工具和格式可用于此目的。PkgConfig 就是其中之一，CMake 也提供了一个包装器模块来支持

它。

## 7.3. 使用 FindPkgConfig

管理依赖关系和发现所需的所有编译标志的问题和 C++ 库本身一样古老，有许多工具可以处理它。其中一个（曾经非常流行）工具叫做 PkgConfig([freedesktop.org/wiki/Software/pkg-config/](https://freedesktop.org/wiki/Software/pkg-config/))。通常在类 Unix 系统上可用（也可以在 macOS 和 Windows 上工作）。

pkg-config 正慢慢被其他更现代的解决方案淘汰。这里出现了一个问题——是否应该投入时间来支持它？答案——视情况而定：

- 若一个库非常流行，可能已经在 CMake 中有它的查找模块。这种情况下，就不需要它。
- 若没有查找模块（或者它对库不起作用），并且该库只提供 PkgConfig.pc，那么使用现成的即可。

许多（若不是大多数的话）库都采用了 CMake，并在当前版本中提供了一个包配置文件。若不需要发布解决方案，并且可以控制环境，请使用 `find_package()`，不要担心遗留版本的问题。

遗憾的是，并不是所有的环境都可以快速更新到库的最新版本。许多公司仍然在生产中使用遗留系统，这些系统不再获得最新的软件包，用户可能只能使用旧版本（但希望兼容）。通常，会提供一个.pc 文件。

此外，若项目能够为大多数用户提供开箱即用的服务，支持较老的 PkgConfig 格式可能值得。

开始使用 `find_package()`，若 `_FOUND` 为 `false`，退回到 PkgConfig。通过这种方式，可以使用一种环境升级的方式，可以只使用主方法，而不更改代码。

这个助手工具的概念非常简单——库的作者提供了一个包含编译和链接所需细节的小的.pc 文件：

```
prefix=/usr/local
exec_prefix=${prefix}
includedir=${prefix}/include
libdir=${exec_prefix}/lib

Name: foobar
Description: A foobar library
Version: 1.0.0
Cflags: -I${includedir}/foobar
Libs: -L${libdir} -lfoobar
```

该格式非常简单、轻量级，甚至支持基本的变量展开。这就是为什么许多开发人员更喜欢它，而不是像 CMake 这样复杂、健壮的解决方案。虽然 PkgConfig 非常容易使用，但功能非常有限：

- 检查系统中是否存在库，以及是否提供了.pc 文件
- 检查是否有足够的库版本可用
- 通过运行 `pkg-config --libs libfoo` 获取库的链接器标志
- 获取库的包含目录（该字段在技术上可以包含其他编译器标志）——`pkg-config --cflags libfoo`

要在构建场景中正确使用 PkgConfig，构建系统必须在操作系统中找到 `pkg-config` 可执行文件，运行几次并提供适当的参数，并将响应存储在变量中，以便稍后将它们传递给编译器。已经知道如何在 CMake 中做到这一点——扫描已知的用于存储帮助工具的路径来检查 PkgConfig 是否已安装，然后使用一些 `exec_program()` 指令来发现如何链接依赖项。尽管这些步骤是有限的，但当想要使用 PkgConfig 时，每次都这样做似乎有点不妥。

CMake 提供了一个方便的内置查找模块——`FindPkgConfig`。它遵循了常规查找模块的大多数规则，但没有提供 `PKG_CONFIG_INCLUDE_DIRS` 或 `PKG_CONFIG_LIBS` 变量，而是设置了一个直接指向二进制文件的变量——`PKG_CONFIG_EXECUTABLE`。意料之中的是，`PKG_CONFIG_FOUND` 变量也设置了——我们将使用它来确认该工具在系统中是可用的，然后用模块中定义的 `pkg_check_modules()` 帮助命令扫描包。

让我们在实践中看看。一个提供.pc 文件的流行库的例子是 PostgreSQL 数据库的客户端——`libpqxx`。

要在 Debian 上安装它，可以使用 `libpqxx-dev` 包（不同的操作系统可能需要不同的包）：

```
apt-get install libpqxx-dev
```

创建 `main.cpp` 文件，使用虚拟连接类：

```
1 // chapter07/02-find-pkg-config/main.cpp
2
3 #include <pqxx/pqxx>
4 int main()
5 {
6 // We're not actually connecting, but
7 // just proving that pqxx is available.
8 pqxx::nullconnection connection;
9 }
```

现在可以使用 PkgConfig 查找模块为前面的代码提供必要的依赖项：

```
chapter07/03-find-pkg-config/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(FindPkgConfig CXX)

find_package(PkgConfig REQUIRED)
pkg_check_modules(PQXX REQUIRED IMPORTED_TARGET libpqxx)
message("PQXX_FOUND: ${PQXX_FOUND}")

add_executable(main main.cpp)
target_link_libraries(main PRIVATE PkgConfig::PQXX)
```

让我们分析一波：

- 要求 CMake 使用 `find_package()` 指令查找 PkgConfig 可执行文件。若由于 REQUIRED 关键字而不存在 `pkg-config`, 则会失败。
- 调用 `FindPkgConfig` 查找模块中定义的 `pkg_check_modules()` 自定义宏, 以 `PQXX` 作为所选名称创建一个新的导入目标。查找模块将搜索名为 `libpqxx` 的依赖项, 若库因为 REQUIRED 关键字而不可用, 将再次失败。注意 `IMPORTED_TARGET` 关键字——若没有它, 就不会自动创建目标, 必须用宏手动定义它。
- 通过打印 `PQXX_FOUND` 确认诊断消息中的一切都是正确的。若没有在前面的命令中指定 REQUIRED, 可以在这里检查这个变量是否设置(可能是为了允许其他回退机制启动)。
- 创建 `main` 可执行文件。
- 链接由 `pkg_check_modules()` 创建的 `PkgConfig::PQXX` 导入目标。注意, `PkgConfig::` 是一个常量前缀, `PQXX` 来自传递给该命令的第一个参数。

这是引入尚不支持 CMake 的依赖项的一种相当方便的方法。这个查找模块有一些其他的方法和选项, 若有兴趣了解更多, 建议参考官方文档:<https://cmake.org/cmake/help/latest/module/FindPkgConfig.html>。

查找模块是为 CMake 提供关于已安装依赖项的一种方式。CMake 在所有主要平台上都广泛支持大多数流行的库, 但当我们想要使用一个还没有专门的查找模块的第三方库时, 该怎么办呢?

## 7.4. 编写自己的查找模块

极少数情况下, 在项目中会使用的库不提供 `config` 文件或 `PkgConfig` 文件, 而且 CMake 中也没有现成的查找模块。可以为该库编写一个自定义查找模块, 并将其与项目一起发布。这种情况并不理想, 但为了照顾项目的用户, 有时必须这样做。

上一节中已经熟悉了 `libpqxx`, 所以让我们编写一个漂亮的查找模块吧。首先编写一个新的 `FindPQXX.cmake` 文件, 将它存储在项目源代码树的 `cmake/module` 目录中。需要确保当使用 `find_package()` 时, CMake 可以找到这个查找模块, 所以会在 `CMakeLists.txt` 中将该目录追加到 `CMAKE_MODULE_PATH` 变量中。整个文件应该是这样的:

```
chapter07/04-find-package-custom/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0) project(FindPackageCustom CXX)

list(APPEND CMAKE_MODULE_PATH
"${CMAKE_SOURCE_DIR}/cmake/module/")
find_package(PQXX REQUIRED)
add_executable(main main.cpp)
target_link_libraries(main PRIVATE PQXX::PQXX)
```

现在, 需要编写查找模块。若 `FindPQXX.cmake` 文件为空: 若某些特定的变量没有设置(包括 `PQXX_FOUND`), CMake 不会报错, 即使用户使用 REQUIRED 调用 `find_package()`。这取决于查找模块的作者是否遵守 CMake 文档中列出的约定:

- 使用 `find_package(<PKG_NAME> REQUIRED)` 时, CMake 将提供一个 `<PKG_NAME>_FIND_REQUIRED` 变量, 并设置为 1。当没有找到库时, 查找模块应该使用 `message(FATAL_ERROR)`。
- 使用 `find_package(<PKG_NAME> QUIET)` 时, CMake 将提供一个 `<PKG_NAME>_FIND_QUIETLY` 变量, 并设置为 1。查找模块应该跳过打印诊断消息(前面提到的除外)。
- CMake 将提供 `<PKG_NAME>_FIND_VERSION` 变量, 设置为调用列表文件所需的版本。查找模块应该找到适当的版本或发出 `FATAL_ERROR` 信息。

当然, 为了与其他查找模块保持一致, 最好遵循前面的规则。为 PQXX 创建一个优雅的查找模块所需的步骤:

1. 若库和头文件的路径已知(由用户提供, 或来自前一次运行的缓存), 则使用这些路径并创建导入的目标。结束。
2. 否则, 找到嵌套依赖的库和头文件——PostgreSQL。
3. 已知的路径中搜索二进制版本的 PostgreSQL 客户端库。
4. 搜索 PostgreSQL 客户端包含头文件的已知路径。
5. 检查是否找到库和 `include` 头文件。是的话, 创建一个 `IMPORTED` 目标。

`IMPORTED` 目标的创建会发生两次——若用户从命令行提供库的路径, 或者自动找到。首先编写一个函数来处理搜索过程的结果并保持代码的 DRY。

要创建 `IMPORTED` 目标, 只需要一个 `IMPORTED` 关键字的库(在 `CMakeLists.txt` 中的 `target_link_libraries()` 指令中使用它)。库必须提供一个类型——将其标记为 `UNKNOWN` 表示我们不想检测所找到的库是静态的, 还是动态的, 只是想为链接器提供一个参数。

接下来, `IMPORTED_LOCATION` 和 `INTERFACE_INCLUDE_DIRECTORIES` 导入目标的必需属性设置为调用函数时使用的参数。还可以指定其他属性(如 `COMPILE_DEFINITIONS`), 这些对 PQXX 来说不是必要的。

之后, 将把路径存储在缓存变量中, 这样就不需要再次执行搜索了。`PQXX_FOUND` 是在缓存中显式设置的, 因此在全局变量作用域中是可见的(因此可以由用户的 `CMakeLists.txt` 访问)。

最后, 将缓存变量标记为高级, 除非启用了“高级”选项, 否则它们在 CMake GUI 中是不可见的。这是这些变量的常见做法, 我们也应该遵循这个惯例:

```
chapter07/04-find-package-custom/cmake/module/FindPQXX.cmake

function(add_imported_library library headers)
 add_library(PQXX::PQXX UNKNOWN IMPORTED)
 set_target_properties(PQXX::PQXX PROPERTIES
 IMPORTED_LOCATION ${library}
 INTERFACE_INCLUDE_DIRECTORIES ${headers}
)
 set(PQXX_FOUND 1 CACHE INTERNAL "PQXX found" FORCE)
 set(PQXX_LIBRARIES ${library})
```

```

CACHE STRING "Path to pqxx library" FORCE)
set(PQXX_INCLUDES ${headers}
 CACHE STRING "Path to pqxx headers" FORCE)
mark_as_advanced(FORCE PQXX_LIBRARIES)
mark_as_advanced(FORCE PQXX_INCLUDES)
endfunction()

```

接下来，介绍第一种情况——将 PQXX 安装在非标准位置的用户可以使用-D 参数通过命令行提供必要的路径。若是这种情况，只需调用刚刚定义的函数，并通过使用 return() 进行转义来放弃搜索。相信用户最清楚，并提供库及其依赖项 (PostgreSQL) 的正确路径。

若配置阶段在过去执行过，这个条件也为 true，因为 PQXX\_LIBRARIES 和 PQXX\_INCLUDES 变量缓存了。

```

if (PQXX_LIBRARIES AND PQXX_INCLUDES)
 add_imported_library(${PQXX_LIBRARIES} ${PQXX_INCLUDES})
 return()
endif()

```

现在是时候找到一些嵌套依赖项了。要使用 PQXX，主机还需要 PostgreSQL。查找模块中使用另一个查找模块没问题，但应该将 REQUIRED 和 QUIET 标志进行转发给 (这样嵌套的搜索行为与外部的搜索行为一致)。这不是复杂的逻辑，但应该尽量避免不必要的代码。

CMake 有一个内置的帮助宏，可以做这件事——find\_dependency()。有趣的是，文档指出它并不适合查找模块，因为若没有找到依赖项，会调用 return()。因为这是一个宏 (而不是函数)，所以 return() 将退出调用者的作用域 FindPQXX.cmake 文件，停止外部查找模块的执行。可能在某些情况下，这是不可取的，但在这个例子中，这正是我们想要做的——防止 CMake 掉进坑里，在已知 PostgreSQL 不可用时寻找 PQXX 的组件：

```

deliberately used in mind-module against the
documentation
include(CMakeFindDependencyMacro)
find_dependency(PostgreSQL)

```

要查找 PQXX 库，将设置一个\_PQXX\_DIR 帮助变量 (转换为 cmake 样式的路径)，并使用 find\_library() 扫描路径列表，在 PATHS 关键字后面提供路径。该指令将检查是否存在与另一个关键字 NAMES 后面提供的名称相匹配的库二进制文件。若找到匹配的文件，路径将存储在 PQXX\_LIBRARY\_PATH 变量中。否则，该变量将设置为 <var>-NOTFOUND，或是本例中的 PQXX\_HEADER\_PATH-NOTFOUND。

NO\_DEFAULT\_PATH 关键字禁用默认行为，该行为将扫描 CMake 为该主机环境提供的一长串默认路径：

```

file(TO_CMAKE_PATH "$ENV{PQXX_DIR}" _PQXX_DIR)
find_library(PQXX_LIBRARY_PATH NAMES libpqxx pqxx
 PATHS

```

```
${_PQXX_DIR}/lib/${CMAKE_LIBRARY_ARCHITECTURE}
(...) many other paths - removed for brevity
/usr/lib
NO_DEFAULT_PATH
)
```

接下来，将使用 `find_path()` 指令搜索所有已知的头文件，该命令的工作原理与 `find_library()` 非常相似。主要的区别是 `find_library()` 知道库的特定于系统的扩展名，并将根据需要隐式附加这些扩展名，而对于 `find_path()`，我们需要提供确切的名称。

另外，不要在这里与 `pqxx/pqxx` 混淆。它是一个实际的头文件，但是库作者故意省略了扩展名，以遵循 C++ 风格的 `#include` 指令（而不是遵循 C 风格的.h 扩展名）：

```
#include <pqxx/pqxx>

find_path(PQXX_HEADER_PATH NAMES pqxx/pqxx
PATHS
${_PQXX_DIR}/include
(...) many other paths - removed for brevity
/usr/include
NO_DEFAULT_PATH
)
```

现在是时候检查 `PQXX_LIBRARY_PATH` 和 `PQXX_HEADER_PATH` 变量是否包含`-NOTFOUND` 值了。同样，可以手动完成此操作，然后根据约定打印诊断消息或终止构建执行，或者可以使用 CMake 提供的 `FindPackageHandleStandardArgs` 列表文件中的 `find_package_handle_standard_args()` 助手函数。它是一个辅助指令，若填充路径变量，会将 `<PKG_NAME>_FOUND` 变量设置为 1，并提供关于成功和失败的正确诊断消息（考虑 `QUIET` 关键字）。当 `REQUIRED` 关键字传递给查找模块时，若提供的路径变量中的一个为空，其还将使用 `FATAL_ERROR` 终止执行。

若找到了库，将调用函数来定义导入的目标，并将路径存储在缓存中：

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(
 PQXX DEFAULT_MSG PQXX_LIBRARY_PATH PQXX_HEADER_PATH
)
if (PQXX_FOUND)
 add_imported_library(
 ${PQXX_LIBRARY_PATH};${POSTGRES_LIBRARIES}
 ${PQXX_HEADER_PATH};${POSTGRES_INCLUDE_DIRECTORIES}
)
endif()
```

这个查找模块将查找 PQXX 并创建适当的 PQXX::PQXX 目标。可以在本书示例库中找到这个文件: chapter07/04find-package-custom/cmake/module/FindPQXX.cmake.

若一个库很流行并且很可能已经安装在系统中, 那么这个方法就非常有效。然而, 并不是所有的库都是随时可用的。那我们能用 CMake 让用户更容易获取和构建这些依赖关系吗?

## 7.5. 使用 Git 库

许多项目以 Git 作为版本控制系统。假设我们的项目和外部库都在使用它, 是否有某种 Git 魔法可以将这些存储库链接在一起呢? 能否构建库的特定(或最新)版本作为构建项目的第一步? 如果是这样, 如何操作?

### 7.5.1 通过 Git 子模块提供外部库

可能的解决方案是使用 Git 内置的机制, 称为 Git 子模块。子模块允许项目存储库使用其他 Git 存储库, 而无需将引用的文件添加到项目存储库中。工作原理类似于软链接——指向特定的分支或在外部存储库中提交(需要显式地更新)。要向存储库中添加一个子模块(并克隆它的存储库), 执行以下命令:

```
git submodule add <repository-url>
```

若提取的存储库已经有子模块, 需要初始化它们:

```
git submodule update --init -- <local-path-to-submodule>
```

这是在解决方案中利用第三方代码的一种通用机制。这种方式的一个小缺点是, 当用户用根项目克隆存储库时, 不会自动提取子模块。需要显式使用 init/pull 命令, 也可以用 CMake 解决。首先, 看看如何在代码中使用新创建的子模块。

对于本例, 我决定编写一个小程序, 从 YAML 文件读取一个名称并在欢迎消息中打印出来。YAML 是一种很棒的、简单的格式, 用于存储人类可读的配置, 但机器解析相当复杂。我找到了一个由 Jesse Beder(和当时 92 个其他贡献者)编写的整洁的小项目, 解决了这个问题, 名为 yaml-cpp<https://github.com/jbeder/yaml-cpp>)。

这个例子相当简单。这是一个打印 Welcome <name> 消息的欢迎程序。name 的默认值是 Guest, 可以在 YAML 配置文件中指定不同的名称:

```
chapter07/05-git-submodule-manual/main.cpp

#include <string>
#include <iostream>
#include "yaml-cpp/yaml.h"

using namespace std;
```

```

int main() {
 string name = "Guest";
 YAML::Node config = YAML::LoadFile("config.yaml");
 if (config["name"])
 name = config["name"].as<string>();

 cout << "Welcome " << name << endl;
 return 0;
}

```

本例的配置文件只有一行:

```

chapter07/05-git-submodule-manual/config.yaml

name: Rafal

```

main.cpp 包含 “yaml-cpp/yaml.h” 头文件, 这需要克隆 yaml-cpp 项目并构建。创建一个 `extern` 目录来存储所有第三方依赖项 (参见 3.2.2 节的建议), 并添加一个子模块 Git, 引用这个库的仓库:

```

$ mkdir extern
$ cd extern
$ git submodule add https://github.com/jbeder/yaml-cpp.git
Cloning into 'chapter07/01-git-submodule-manual/extern/yamlcpp'...
remote: Enumerating objects: 8134, done.
remote: Total 8134 (delta 0), reused 0 (delta 0), pack-reused
8134
Receiving objects: 100% (8134/8134), 3.86 MiB | 3.24 MiB/s, done.
Resolving deltas: 100% (5307/5307), done.

```

Git 已经克隆了存储库, 现在可以将它作为一个依赖项添加到我们的项目中, 并让 CMake 负责构建:

```

chapter07/05-git-submodule-manual/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(GitSubmoduleManual CXX)

add_executable(welcome main.cpp)
configure_file(config.yaml config.yaml COPYONLY)

add_subdirectory(extern/yaml-cpp)
target_link_libraries(welcome PRIVATE yaml-cpp)

```

这里分解一下 CMake 的指令：

- 设置项目并添加 welcome 可执行文件。
- 接下来，使用 `configure_file`，但实际上不进行配置。通过 `COPYONLY` 关键字，只需复制 `config.yaml` 到构建树中，以便可执行程序可以在运行时中找到它。
- 添加 yaml-cpp 存储库的子目录。CMake 将它作为项目的一部分，递归地执行任何嵌套的 `CMakeLists.txt` 文件。
- 将库提供的 yaml-cpp 目标与 welcome 目标链接起来。

yaml-cpp 的作者遵循了第 3 章中的实践，并将公共头文件存储在一个单独的目录中——`<project-name>/include/<project-name>`。这允许库的使用者（如 `main.cpp`）包含 “`yaml-cpp/yaml.h`”，这样的命名实践对于查找头文件非常有用——可以立即知道是哪个库提供了这个头文件。

这并不复杂，但并不理想——用户必须在克隆存储库之后手动初始化添加的子模块。更糟糕的是，没有考虑到用户可能已经在其系统中安装了这个库，所以这个依赖的下载和构建可能是多余的。

## 自动初始化 Git 子模块

对开发人员来说，为用户提供简洁的体验并不总是件痛苦的事。若一个库提供了一个包配置文件，可以要求 `find_package()` 在已安装的库中搜索。如前所述，CMake 将首先检查是否有合适的查找模块，若没有，将查找配置文件。

我们已经知道，若 `<lib_name>_FOUND` 变量设置为 1，代表库找到了，可以直接使用它。还可以在没有找到库时采取行动，并提供方便的解决方案，改善用户体验：回到获取子模块并从源代码构建库。新克隆的存储库不会自动下载和初始化嵌套子模块看起来并不是那么糟糕，不是吗？

让对前面的例子进行扩展：

```
chapter07/06-git-submodule-auto/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(GitSubmoduleAuto CXX)

add_executable(welcome main.cpp)
configure_file(config.yaml config.yaml COPYONLY)
#####
find_package(yaml-cpp QUIET)
if(NOT yaml-cpp_FOUND)
 message("yaml-cpp not found, initializing git submodule")
 execute_process(
 COMMAND git submodule update --init -- extern/yaml-cpp
 WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
)
 add_subdirectory(extern/yaml-cpp)
endif()
```

```
#####
target_link_libraries(welcome PRIVATE yaml-cpp)
```

我们添加了 # 中显示的行:

- 尝试找到 yaml-cpp 并使用它。
- 如果不存在, 将打印一个简短的诊断消息并使用 `execute_process()` 指令初始化子模块。这可以有效地从引用的存储库中克隆文件。
- 最后, 使用 `add_subdirectory()` 使用源代码构建依赖项。

简单明了! 这也适用于不是用 CMake 构建的库——可以遵循 git 子模块的例子, 再次调用 `execute_process()`, 以同样的方式启动外部构建工具。

不幸的是, 若使用并发版本系统 (CVS)、Subversion (SVN)、Mercurial 或其他将代码交付给用户的工具, 这种方法就会失效。若不能依赖 Git 子模块, 还有什么替代方法?

## 7.5.2 不使用 Git 的项目克隆依赖项

若正在使用另一种 VCS 或在存档中提供源码, 那么可能很难依赖 Git 子模块为存储库引入外部依赖项。构建代码的环境可能已经安装了 Git, 并且可以执行 `git clone` 命令。

让我们看看该怎么做:

```
chapter07/07-git-clone/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(GitClone CXX)

add_executable(welcome main.cpp)
configure_file(config.yaml config.yaml COPYONLY)

find_package(yaml-cpp QUIET)
if (NOT yaml-cpp_FOUND)
 message("yaml-cpp not found, cloning git repository")
#####
find_package(Git)
if (NOT Git_FOUND)
 message(FATAL_ERROR "Git not found, can't initialize!")
endif ()
execute_process(
 COMMAND ${GIT_EXECUTABLE} clone
 https://github.com/jbeder/yaml-cpp.git
 WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/extern
)
#####
```

```
add_subdirectory(extern/yaml-cpp)
endif()
target_link_libraries(welcome PRIVATE yaml-cpp)
```

同样，# 中的行是 YAML 项目中的新部分。发生了什么：

- 首先通过 FindGit 查找模块检查 Git 是否可用。
- 若不可用，就会陷入困境。将发出 FATAL\_ERROR，并希望用户知道下一步要做什么。
- 否则，将使用 find\_package() 设置的 GIT\_EXECUTABLE 变量调用 execute\_process()，并克隆我们感兴趣的存储库。

Git 对于有一定经验的开发人员特别有吸引力，非常适合不包含对相同存储库的嵌套引用的较小项目。但若是这样，将发现可能需要多次克隆和构建同一个项目。若依赖项项目根本不使用 Git，则需要另一种解决方案。

## 7.6. 使用 ExternalProject 和 FetchContent 模块

关于 CMake 的在线参考将建议使用 ExternalProject 和 FetchContent 模块来处理更复杂项目中的依赖性管理。这是一个很好的建议，但通常是在没有给出相应的使用背景，这就出现了很多问题。这些模块是干什么用的？什么时候选择这一个，而不是另一个？它们究竟是如何工作的，之间又是如何相互作用的？有些答案比其他的更难找到，并且 CMake 的文档并没有对这个主题提供一个清晰的介绍，但这里会给出。

### 7.6.1 ExternalProject

CMake 3.0.0 引入了名为 ExternalProject 的模块，目的是添加对在线存储库中可用的外部项目的支持。随着时间的推移，该模块逐渐扩展以满足不同的需求，产生了相当复杂的命令—ExternalProject\_Add()。因为它提供了一系列令人印象深刻的的功能，并且可以接受超过 85 种不同的选择：

- 管理外部项目的目录结构
- 从 URL 下载源代码（若需要，还可以从压缩包中提取）
- 支持 Git、Subversion、Mercurial 和 CVS 库
- 若需要，可以获取更新
- 使用 CMake、Make 或用户指定的工具配置和构建项目
- 执行安装和运行测试
- 日志文件
- 终端请求用户输入
- 依赖于其他目标
- 向构建中添加自定义命令/步骤

ExternalProject 模块在构建阶段填充依赖项。对于 ExternalProject\_Add() 添加的每个外部项目，CMake 将执行以下步骤：

1. `mkdir` - 为外部项目创建子目录
2. `download` - 从存储库或 URL 获取项目文件
3. `update` - 重新运行时支持增量更新的下载方式
4. `patch` - 可选地执行一个补丁命令，根据项目的需要更改下载的文件
5. `configure` - 执行 CMake 项目的配置阶段，或者手动指定非 CMake 依赖项的命令
6. `build` - 执行 CMake 项目的构建阶段，而对于其他依赖项，执行 `make` 命令
7. `install` - 安装 CMake 项目，对于其他依赖，执行 `make install` 命令
8. `test` - 若选择性的定义了 `TEST_...`，则执行依赖项的测试

除测试步骤外，其他步骤完全遵循上述顺序，测试步骤可以通过 `TEST_BEFORE_INSTALL <bool>` 或 `TEST_AFTER_INSTALL <bool>` 选项在安装步骤之前或之后启用。

## 下载步骤选项

最有趣的是控制下载步骤的选项，或者 CMake 如何获取依赖项。首先，可以选择不使用 CMake 内置方法，而使用一个自定义指令 (这里支持生成器表达式):

```
DOWNLOAD_COMMAND <cmd>...
```

这就是告诉 CMake 忽略此步骤的所有其他选项，只执行一个特定于系统的命令。也接受空字符串，用于禁用此步骤。

## 使用 URL 下载依赖项

可以提供一个按顺序扫描的 URL 列表，直到下载成功。CMake 会识别下载的文件是否是压缩包，并在默认情况下将其解包:

```
URL <url1> [<url2>...]
```

一些选项允许进一步定制化这个方法的行为:

- `URL_HASH <algo>=<hashValue>` - 检查 `<algo>` 生成的下载文件的校验和是否与提供的匹配，这里建议检查下载文件的完整性。MD5、SHA1、SHA224、SHA256、SHA384、SHA512、SHA3\_224、SHA3\_256、SHA3\_384 和 SHA3\_512 支持的算法由 `string()` 指令定义。对于 MD5，可以使用简写选项 `URL_MD5 <MD5>`。
- `DOWNLOAD_NO_EXTRACT <bool>` - 下载后显式禁用提取。可以在后续步骤中通过访问 `<DOWNLOADED_FILE>` 来使用下载文件的文件名。
- `DOWNLOAD_NO_PROGRESS <bool>` - 不显示下载进度。
- `TIMEOUT <seconds>` 和 `INACTIVITY_TIMEOUT <seconds>` - 超时后终止在固定时间或不活动期间后的下载。
- `HTTP_USERNAME <username>` 和 `HTTP_PASSWORD <password>` - 选项为 HTTP 身份验证提供值，要避免在项目中硬编码凭证。
- `HTTP_HEADER <header1> [<header2>...]` - 在 HTTP 请求中发送附加的报头，来访问 AWS 中的内容或传递一些自定义令牌。

- **TLS\_VERIFY <bool>** - 验证 SSL 证书。若没有设置, CMake 将从 `CMAKE_TLS_VERIFY` 变量中读取该设置, 该变量默认设置为 `false`。跳过 TLS 验证是一种不安全的做法, 应该避免使用, 特别是在生产环境中。
- **TLS\_CAINFO <file>** - 若公司正在颁发自签名 SSL 证书, 这将非常有用。该选项提供到授权文件的路径; 若没有指定, CMake 将从 `CMAKE_TLS_CAINFO` 中读取该设置。

## 使用 Git 下载依赖项

要从 Git 下载依赖项, 需要确定主机安装了 Git 1.6.5 或更高版本。从 Git 中克隆需要以下选项:

```
GIT_REPOSITORY <url>
GIT_TAG <tag>
```

`<url>` 和 `<tag>` 都应该采用 git 命令能够理解的格式。此外, 建议使用特定的 git 提交哈希, 以确保生成的二进制文件可以跟踪到特定的提交, 并且没有不必要的 git 获取执行。若坚持使用分支, 请使用诸如 `origin/main` 这样的远程名称。这保证了本地克隆的正确同步。

其他选项如下:

- **GIT\_REMOTE\_NAME <name>** - 远程名称, 默认为 `origin`。
- **GIT\_SUBMODULES <module>...** - 指定应该更新哪些子模块。从 3.16 开始, 这个值默认为空 (以前, 所有子模块都更新了)。
- **GIT\_SUBMODULES\_RECURSE 1** - 启用子模块的递归更新。
- **GIT\_SHALLOW 1** - 执行浅克隆 (不下载历史提交)。出于性能考虑, 建议使用此选项。
- **TLS\_VERIFY <bool>** - 从 URL 下载依赖项一节解释了这个选项。Git 也可以使用, 为了安全性应该启用。

## 使用 Subversion 下载依赖项

使用 Subversion 下载, 应该指定以下选项:

```
SVN_REPOSITORY <url>
SVN_REVISION -r<rev>
```

此外, 可能需要提供以下信息:

- **SVN\_USERNAME <user>** 和 **SVN\_PASSWORD <password>** - 用于签出和更新的凭据, 不过需要避免在项目中硬编码它们。
- **SVN\_TRUST\_CERT <bool>** - 跳过 Subversion 服务器站点证书的验证。只有在到服务器的网络路径及其完整性是可信任的情况下才使用此选项。默认是禁用的。

## 使用 Mercurial 下载依赖项

这种模式非常简单, 需要提供两个选项:

```
HG_REPOSITORY <url>
HG_TAG <tag>
```

## 使用 CVS 下载依赖项

要检出 CVS 中的模块，需要提供以下三个选项：

```
CVS_REPOSITORY <cvsroot>
CVS_MODULE <module>
CVS_TAG <tag>
```

## 更新步骤选项

若下载方法支持更新，更新步骤将重新下载外部项目的文件。可以用两种方式覆盖这种行为：

- 用 UPDATE\_COMMAND <cmd> 提供在更新期间执行的自定义命令。
- 禁用更新步骤 (以允许使用断开连接的网络构建)- UPDATE\_DISCONNECTED <bool>。请注意，下载步骤 (在第一次构建期间) 仍然会发生。

## 补丁步骤选项

这是一个可选步骤，将在获取源之后执行，需要指定想要执行的命令：

```
PATCH_COMMAND <cmd>...
```

CMake 文档警告说，有些补丁可能比其他补丁更“黏”。例如，在 Git 中，更改的文件在更新期间不会恢复到原始状态，需要小心避免两次错误地对文件打补丁。理想情况下，patch 命令应该非常健壮。

### 重要的 Note

前面提到的选项列表只包含最有用的条目。请务必参考官方文档了解更多细节和其他步骤的选项描述: <https://cmake.org/cmake/help/latest/module/ExternalProject.html>.

## 使用 ExternalProject

依赖项在构建阶段创建，这样有两个效果——项目的命名空间完全独立，外部项目定义的目标在主项目中不可见。因为不能像使用 `find_package()` 指令一样使用 `target_link_libraries()`，所以后者尤其令人痛苦。没办法，因为两个构型阶段是分离的。在下载和配置依赖项之前，主项目必须完成配置阶段并启动构建阶段。这是一个问题，但我们会学习如何处理这个问题。现在，让我们看看 `ExternalProject_Add()` 如何与前面的 yaml-cpp 库一起工作：

```
chapter07/08-external-project-git/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(ExternalProjectGit CXX)

add_executable(welcome main.cpp)
```

```
configure_file(config.yaml config.yaml COPYONLY)

include(ExternalProject)
ExternalProject_Add(external-yaml-cpp
 GIT_REPOSITORY https://github.com/jbeder/yaml-cpp.git
 GIT_TAG yaml-cpp-0.6.3
)
target_link_libraries(welcome PRIVATE yaml-cpp)
```

以下是构建此项目所采取的步骤:

- 包含了 ExternalProject 模块来访问它的功能。
- 使用 FindExternalProject\_Add(), 该指令负责构建阶段下载必要的文件，并在系统中配置、构建和安装依赖项。

这里需要谨慎，并理解这个示例只能工作，因为 yaml-cpp 库在其 CMakeLists.txt 中定义了一个安装阶段。此阶段将库文件复制到系统中的标准位置。target\_link\_libraries() 的 yaml-cpp 参数解释为链接器——-lyaml-cpp。此行为与前面的示例不同，前面的示例中，显式地定义了 yaml-cpp 目标。若库不提供安装阶段（或者二进制版本的名称不匹配），链接器就会出错。

现在，应该更深入地研究每个阶段的配置，并解释如何使用不同的下载方法。我们会在 FetchContent 部分讨论这个问题，回到 ExternalProject 获取依赖项的延迟问题，不能在编译阶段使用外部项目的目标，因为在获取这些项目时，编译阶段已经结束了。CMake 将显式地保护用 FindExternalProject\_Add() 创建的目标，用一个特殊的 UTILITY 类型标记它。当错误地尝试在主项目中使用这样的目标时（可能链接），CMake 将抛出一个错误：

```
Target "external-yaml-cpp-build" of type UTILITY may not be
linked into another target.
```

为了避开这个限制，可以在技术上创建另一个目标，一个 IMPORTED 库，并使用它（就像使用 FindPQXX.cmake 那样），但这是一项非常艰巨的工作。更糟糕的是 CMake 实际上理解由外部 CMake 项目创建的目标（因为构建了它们）。主项目中重复这些声明并不是一个非常 DRY 的实践。

另一种可能的解决方案是将整个依赖项获取和构建提取到一个单独的子项目中，并在配置阶段构建该子项目，需要用 execute\_process() 启动另一个 CMake 实例。通过一些技巧和 add\_subdirectory()，将这个子项目的列表文件和二进制文件消耗到主项目中。这种方法（有时称为超级构建）是过时的和不必要的。这里就不详细讲了，因为这对初学者来说没什么用。若有兴趣，可以读读 Craig Scott 这篇写得很棒的文章：<https://crascit.com/2015/07/25/cmake-gtest/>。

总而言之，当项目之间存在名称空间冲突时，ExternalProject 可以让我们摆脱束缚，但 FetchContent 都要优雅得多。

## 7.6.2 FetchContent

现在，建议使用 FetchContent 模块来导入外部项目。这个模块从 3.11 版本就已经在 CMake 中可用了，但建议至少使用 3.14 才能有效地使用。

本质上，它是 ExternalProject 的高级包装器，提供类似的功能和更多功能。关键的区别在于执行阶段——与 ExternalProject 不同，FetchContent 在配置阶段填充依赖项，将外部项目声明的所有目标带入主项目的范围。

FetchContent 模块的使用需要三个步骤：

1. 用 `include(FetchContent)` 将该模块包含到项目中。
2. 使用 `FetchContent_Declare()` 指令配置依赖项。
3. 使用 `FetchContent_MakeAvailable()` 指令填充依赖项——下载、构建、安装，并将其列表文件添加到主项目并进行解析。

为什么 `Declare` 和 `MakeAvailable` 命令是分开的。这样做是为了在分层项目中启用配置重写。这里有一个场景——父项目依赖于 A 和 B 外部库。A 库也依赖于 B，但是 A 库的作者仍然使用旧版本，与父项目不同(图 7.1)：

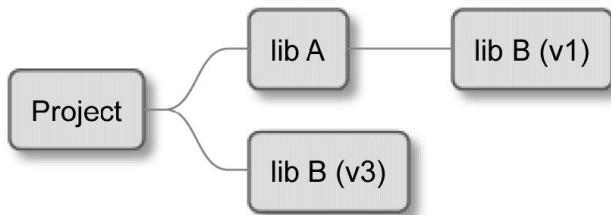


图 7.1 分层项目

而且对 B 库的依赖是可选的，取决于配置(假设特定于操作系统)。`MakeAvailable` 不能同时配置和填充依赖项，因为要覆盖 A 库中的版本，父项目将填充依赖项，而不管它在 A 库中的最终需要。

由于有单独的配置步骤，能够在父项目中指定一个版本，并在所有子项目和依赖项中使用：

```
FetchContent_Declare(
 gtest
 GIT_REPOSITORY https://github.com/google/googletest.git
 # release-1.11.0
 GIT_TAG e2239ee6043f73722e7aa812a459f54a28552929
)
```

使用 `googletest` 作为第一个参数对 `FetchContent_Declare()` 的后续调用都将忽略，以允许层次结构中最高的项目决定如何处理该依赖项。

`FetchContent_Declare()` 的签名与 `ExternalProject_Add()` 完全相同：

```
FetchContent_Declare(<depName> <contentOptions>...)
```

这不是巧合——这些参数将存储起来，直到使用 `FetchContent_MakeAvailable()`。然后，变量将转发至 `ExternalProject_Add()`。然而，并不是所有的选项都允许转发。可以指定下载、更新或补丁步骤的任何选项，但不能指定配置、构建、安装或测试步骤。

当配置就绪时，可以这样填充依赖项：

```
FetchContent_MakeAvailable(<depName>)
```

这将下载文件并将目标读取到项目中，但在调用期间发生了什么呢？在 CMake 3.14 中添加了 `FetchContent_MakeAvailable()` 来将最常用的场景封装在指令中。图 7.2 中，可以看到整个过程的细节：

1. 调用 `FetchContent_GetProperties()` 将 `FetchContent_Declare()` 设置的配置从全局变量读取到本地变量。
2. 检查（不区分大小写）带有此名称的依赖项是否已经填充，以避免下载两次。
3. 使用 `FetchContent_Populate()`，将通过转发设置的选项（但跳过禁用的选项）和下载依赖项来配置包装的 `ExternalProject` 模块。还会设置一些变量，以防止在后续调用中重新下载，并将必要的路径转发到下一个指令。
4. 最后，使用源码树和构建树作为参数调用 `add_subdirectory()`，以告诉父项目列表文件在哪里，以及将构建构件放在哪里。

通过调用 `add_subdirectory()`，有效地执行所获取项目的配置阶段，并检索在当前作用域中定义的目标。多方便啊！

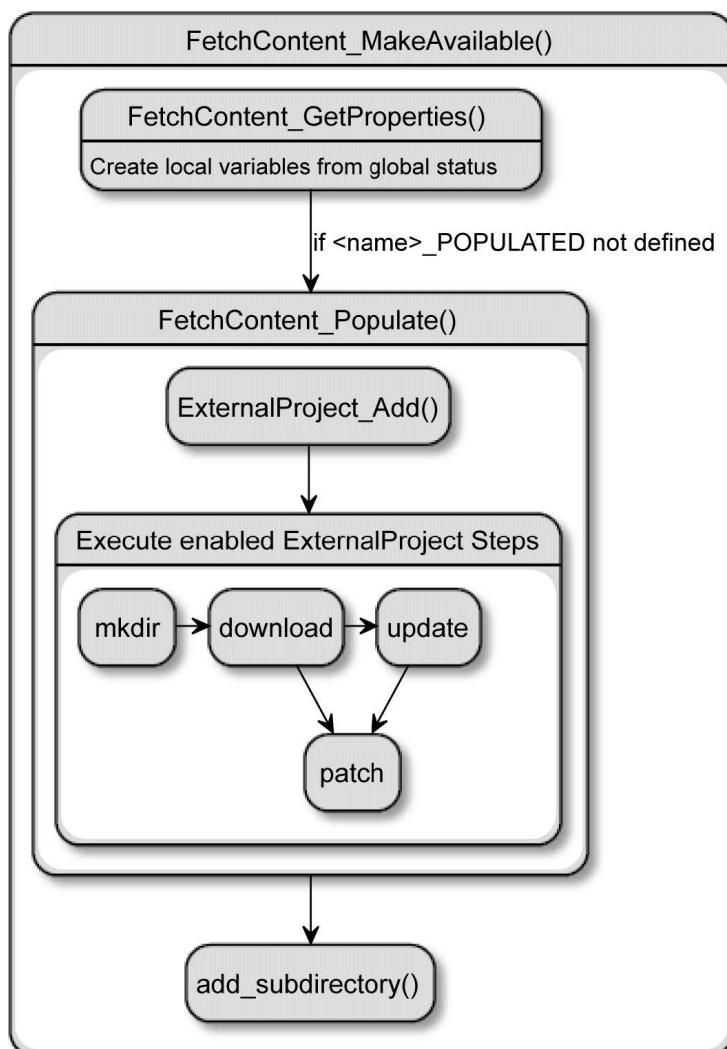


图 7.2 FetchContent\_MakeAvailable() 如何包装对 ExternalProject 的调用

显然，可能会遇到这样的情况：两个不相关的项目声明一个同名目标。这是一个只能通过退回到 ExternalProject 或其他方法来解决的问题，但这种情况并不经常发生。

为了使这个解释完整，必须辅以一个实际的例子。来看看当切换到 FetchContent 时，前一节中的列表文件是如何变化的：

```
chapter07/09-fetch-content/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(ExternalProjectGit CXX)

add_executable(welcome main.cpp)
configure_file(config.yaml config.yaml COPYONLY)

include(FetchContent)
FetchContent_Declare(external-yaml-cpp
 GIT_REPOSITORY https://github.com/jbeder/yaml-cpp.git
 GIT_TAG yaml-cpp-0.6.3
)
FetchContent_MakeAvailable(external-yaml-cpp)
target_link_libraries(welcome PRIVATE yaml-cpp)
```

ExternalProject\_Add 直接替换为 FetchContent\_Declare，并且添加了另一个指令——FetchContent\_MakeAvailable。代码中的变化很小，但实际差异却是巨大的！可以显式访问由 yaml-cpp 库创建的目标。为了证明这一点，可以使用 CMakePrintHelpers 辅助模块，并在前面的文件中添加以下行：

```
include(CMakePrintHelpers)
cmake_print_properties(TARGETS yaml-cpp
 PROPERTIES TYPE SOURCE_DIR)
```

现在，配置阶段将打印以下输出：

```
Properties for TARGET yaml-cpp:
yaml-cpp.TYPE = "STATIC_LIBRARY"
yaml-cpp.SOURCE_DIR = "/tmp/b/_deps/external-yaml-cpp-src"
```

目标的存在，是一个静态库，源目录驻留在构建树中。使用相同的辅助函数来调试 ExternalProject 示例中的目标只会返回：

```
No such TARGET "yaml-cpp" !
```

配置阶段无法识别目标。这就是为什么 FetchContent 要好得多，应该尽可能地使用它。

## 7.7. 总结

当使用现代的、支持良好的项目时，管理依赖关系并不复杂。大多数情况下，可以简单地依赖于系统中可用的库，若不可用则返回到 FetchContent。若依赖关系相对较小且易于构建，那么没什么问题。

对于一些非常大的库（比如 Qt），从源代码构建需要花费大量的时间。要在这些情况下提供自动依赖项解析，必须求助于提供与用户环境相匹配的编译版本库的包管理器。Apt 或 Conan 等外部工具不在本书的范围内，因为它们要么太依赖于系统，要么太复杂。

好消息是只要提供了这样做的明确指示，大多数用户知道如何安装项目可能需要的依赖项。本章中，了解了如何使用 CMake 的查找模块和绑定在库中的配置文件，并检测安装在系统中的包。

还了解了一个不支持 CMake 的库，而是与.pc 文件一起发布，我们将依赖于 PkgConfig 工具和与 CMake 绑定的 FindPkgConfig 查找模块。当使用上述方法找到库时，CMake 将自动创建构建目标，这非常优雅。还讨论了依赖 Git，及其子模块和克隆整个存储库。当其他方法不可行或难以实现时，这种方法就有用。

最后，探讨了 ExternalProject 模块及其功能和限制。研究了 FetchContent 如何扩展 ExternalProject 模块，两个模块间的异同，以及为什么 FetchContent 更好用。

现在，可以在项目中使用常规库了。另一种类型的依赖我们应该了解——测试框架。每个严肃的项目都需要进行正确性测试，CMake 是一个很好的工具，可以使这个过程自动化。我们将在下一部分学习如何进行自动化。

## 7.8. 扩展阅读

有关本章所涵盖主题的更多信息，请参阅以下连接：

- CMake 文档——使用依赖关系指南  
<https://cmake.org/cmake/help/latest/guide/using-dependencies/index.html>
- 教程：C++ 与 CMake 和 Git 的简单依赖管理：  
<https://www.foonathan.net/2016/07/cmake-dependency-handling/>
- 在依赖项目中使用 git——子模块：  
<https://stackoverflow.com/questions/43761594/>
- 使用 PkgConfig：  
<https://gitlab.kitware.com/cmake/community/-/wikis/doc/tutorials/How-To-FindLibraries#piggybacking-on-pkg-config>
- 关于查找模块中导入库的 UNKNOWN 类型的讨论：  
<https://gitlab.kitware.com/cmake/cmake/-/issues/19564>
- Git 子模块是什么：  
<https://www.jwlawson.co.uk/interest/2020/02/23/cmake-external-project.html>
- 如何使用 ExternalProject：

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/tutorials/How-To-FindLibraries#piggybacking-on-pkg-config>

- CMake FetchContent 和 ExternalProject:

<https://www.scivision.dev/cmake-fetchcontent-vs-external-project/>

- 使用 CMake 与外部项目联动:

<http://www.saoe.net/blog/usingcmake-with-external-projects/>

# 第三部分：自动化

完成前面的小节后，您已经可以算是一个合格的构建工程师了，能够使用 **CMake** 构建各种项目。成为 **CMake** 专业人员的最后一步是学习如何引入和自动化各种质量检查，并为协作工作和发布准备项目。大公司开发的高质量项目通常都有同样的理念：自动化重复的任务，这些任务会消耗重要决策时的脑力。

为了实现这一点，利用了 **CMake** 生态系统的强大功能，添加了在构建过程中完成的各种测试：代码样式的检查、单元测试，以及解决方案的静态和动态分析。还将通过使用工具生成漂亮的网页来简化文档过程，将打包和安装我们的项目，使其消费变得轻松，对于其他开发人员和最终用户都是如此。

作为一个总结，我们将把所学的一切整合成一个连贯的单元：一个经得起时间考验的专业项目。

- 第 8 章，测试框架
- 第 9 章，分析工具
- 第 10 章，生成文档
- 第 11 章，安装和打包
- 第 12 章，创建完整的项目

# 第 8 章 测试框架

专业人士都明白测试必须自动化。可能几年前就有人跟他们说过了，或者直到吃了苦头才明白。这种实践对于没有经验的开发者来说并不明显：似乎没有必要，额外的工作不会带来太多价值。难怪当刚开始编写代码时，会避免编写复杂的解决方案和贡献大量的代码库。最有可能的情况是，他们喜欢自己是项目的唯一开发者。这些早期的项目几乎不需要超过几个月的时间来完成，所以几乎没有机会看到代码在历史长河中是如何腐烂的。

这些因素让人感觉编写测试就是在浪费时间和精力。编程学徒可能会对自己说，每次执行“构建-运行”例程时，实际上都测试了代码。毕竟，他们已经手动确认了代码是否正常工作，并完成了预期的工作。终于到了进行下一个任务的时候了，对吧？

自动化测试保证了新的更改不会意外地破坏程序。本章中，将了解测试得重要性，以及如何使用 CTest(CMake 附带的一个工具) 来协调测试的执行。CTest 能够查询可用的测试、过滤、打乱、重复和时间限制。我们将探讨如何使用这些特性、控制 CTest 的输出，以及如何处理失败的测试。

接下来，将调整我们的项目结构，以支持测试并创建自己的测试运行器。在讨论了基本原则之后，将继续添加流行的测试框架：Catch2 和带有 mock 库的 GoogleTest。最后，将介绍使用 LCOV 的详细测试覆盖率报告。

本章中，我们将讨论以下主题：

- 为什么自动化测试值得这么麻烦？
- 使用 CTest 来标准化 CMake 中的测试
- 为 CTest 创建最基本的单元测试
- 单元测试框架
- 生成测试覆盖报告

## 8.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter08>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 <build tree> 和 <source tree>。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 8.2. 为什么自动化测试值得这么麻烦？

试想有一台机器在钢板上打孔的生产线。这些孔必须有特定的大小和形状，以便容纳螺栓，将成品连接在一起。这种工厂线的设计者将设置机器，测试孔是否正确，然后继续工作。在未来，工厂将使用不同的、更厚的钢材；工人会不小心改变孔的尺寸；或者，简单地说，需要打更多的孔，机

器必须升级。聪明的设计师会在生产线上的某些点进行质量控制检查，以确保产品符合规格并保留其关键质量。孔必须符合特定的要求，不管是如何进行的打孔：钻孔、打孔或激光切割。

同样的方法在软件开发中也有应用：很难预测哪些代码段会保持多年不变，哪些代码段会多次修改。随着软件功能的扩展，需要确保不会弄坏，即使是最好的程序员也会犯错误，他们不能预见所做的更改。似乎这还不够，开发人员经常在别人写的代码上工作，不知道早期做出的复杂假设。他们会阅读代码，建立大致的心理模型，添加必要的更改，并希望自己得到的正确结果。大多数情况下，这是对的——可能事实并非如此，引入的 bug 可能需要花费数小时，甚至数天来修复，更不用说它可能对产品和客户造成的损害了。

有时，会偶然发现一些非常难以理解和遵循的代码。不仅会质疑代码如何产生，以及做了什么，而且可能还会开展一场政治迫害，以找出是谁造成了这样的混乱。这种事在我身上发生过，也可能发生在你身上。有时候，代码是匆忙创建的，并没有完全理解要解决的问题。

作为开发者，不仅面临着截止日期或预算的压力。当要求半夜醒来去修复一个严重的错误时，肯定会惊讶于某些错误是如何通过代码检查的。

这些大部分都可以通过自动化测试来避免，这些代码块检查另一段代码（在生产中使用）的行为是否正确。每次有人进行更改，自动化测试应该在没有提示的情况下执行。

这通常作为构建过程的一部分进行，并且作为在将代码合并到存储库之前控制代码质量的步骤。

为了节省时间，可能会倾向于避免自动化测试，但这将会是一个代价高昂的教训。Steven Wright 说得很好：“经验是在需要它的时候才会得到的东西。”相信我：除非是为个人目的编写一次性脚本或为非生产实验创建原型，否则不要跳过编写测试。最初，可能会对精心编写的代码不断在测试中失败的事实感到恼火。但认真考虑一下，失败的测试只是阻止向生产中添加破坏性修改。现在投入的努力将会得到回报，因为修复 bug 的时间节省了（以及一整晚的睡眠）。添加和维护测试也并不像看起来那样困难。

### 8.3. 使用 CTest 来标准化 CMake 中的测试

最终，自动化测试只运行了一个可执行文件，该可执行文件将系统设置为给定状态，执行测试操作，并检查结果是否符合预期。可以将其看作是一种已知的方法，用于对“GIVEN \_ WHEN \_ THEN \_”的语句进行完形填空，并检查 SUT 是否为真。其实，有不止一种方法可以这样做。一切都取决于要使用的框架类型、如何将其连接到 SUT，以及具体配置是什么。即使像测试二进制文件文件名这样的事情，也会影响体验。由于没有公认的标准，开发人员会使用 `test_my_app` 的名称，另一个开发人员会使用 `unit_tests`，还有一个开发人员会使用一些模糊的名称，或者根本不提供测试。发现需要运行哪个文件、使用哪个框架、应该将哪些参数传递给运行程序，以及如何收集结果，这些都是用户希望避免的事情。

CMake 通过引入单独的 `ctest` 命令行工具来解决这个问题，并由项目作者通过列表文件进行配置，并提供了执行测试的统一方式：为使用 CMake 构建的每个项目提供相同的标准化接口。若遵循这个惯例，将享受到其他好处：将项目添加到（CI/CD）流水将更容易，将其显示在（IDE）中，如 Visual Studio 或 Clion——所有这些事情都将简化，将以少量的开销获得更强大的测试运行程序。

如何在已配置的项目上使用 CTest 执行测试？需要从以下三种操作模式中选择一种：

- 测试

- 构建和测试
- 仪表板客户端

最后一种模式可以将测试结果发送到一个名为 CDash 的单独工具(也来自 Kitware)。CDash 在一个易于导航的仪表板中收集和聚合软件质量测试结果，如下面的截图所示：

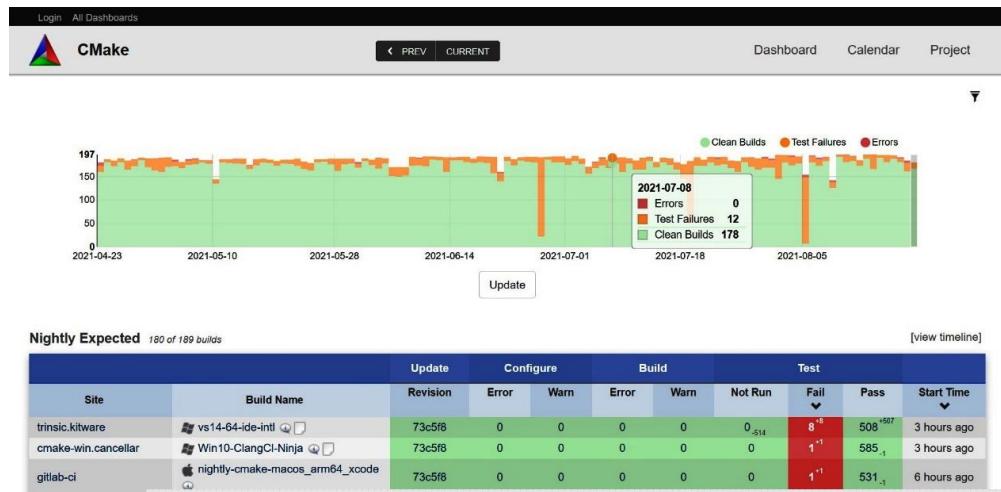


图 8.1 CDash 仪表板时间轴视图的截图

CDash 不在本书的讨论范围内，因为它是作为共享服务器使用的高级解决方案，公司中的所有开发人员应该都可以访问。

#### Note

若有兴趣在线学习更多信息，请参考 CMake 的官方文档并访问 CDash 的网站：

- <https://cmake.org/cmake/help/latest/manual/ctest.1.html#dashboard-client>
- <https://www.cdash.org/>

回到前两种模式。测试模式的命令行：

```
ctest [<options>]
```

这种模式下，使用 `cmake` 构建项目之后，应该在构建树中执行 `CTest`。这在开发周期中有点麻烦，因为需要执行多个命令并来回更改工作目录。为了简化这个过程，`CTest` 添加了第二个模式：构建-测试模式。

### 8.3.1 构建和测试模式

要使用这种模式，需要 `ctest` 有以`--build-and-test` 开头的参数：

```
ctest --build-and-test <path-to-source> <path-to-build>
 --build-generator <generator> [<options>...]
 [--build-options <opts>...]
 [--test-command <command> [<args>...]]
```

这是对常规测试模式的简单包装，需要一些构建配置选项，允许为第一种模式附加命令——可以传递给 `ctest <options>` 的选项在传递给 `ctest --build-and-test` 时都可以工作。要求是在`--test-command` 参数后面传递完整的命令。不过，构建-测试模式实际上不会运行任何测试，除非在`--test-command` 后面提供 `ctest` 关键字：

```
ctest --build-and-test project/source-tree /tmp/build-tree
 --build-generator "Unix Makefiles" --test-command ctest
```

命令中指定了源和构建路径，并选择一个构建生成器。所有这三个都是必须的，并且遵循 `cmake` 命令的规则。

可以向此模式传递附加参数。分为三组，分别控制配置、构建过程或测试。

以下是控制配置阶段的参数：

- `--build-options`—`cmake` 配置（不是构建工具）的选项都应该在`--test-command` 之前提供，`--test-command` 位于最后。
- `--build-two-config`—为 CMake 运行两次配置阶段。
- `--build-nocmake`—跳过配置阶段。
- `--build-generator-platform`, `--build-generator-toolset`—提供生成器特定的平台和工具集。
- `--build-makeprogram`—当使用基于 `make` 或 `ninja` 的生成器时，需要指定 `make` 可执行文件。

以下是控制构建阶段的点：

- `--build-target`—构建指定的目标（而不是所有目标）。
- `--build-noclean`—不构建干净目标的情况下进行构建。
- `--build-project`—提供所构建项目的名称。

用来控制测试阶段的参数：

- `--test-timeout`—限制测试的执行（以秒为单位）。

剩下的就是在`--test-command` 参数之后配置常规测试模式。

### 8.3.2 测试模式

假设已经构建了项目，并且在构建树中执行 `ctest`（或者使用构建-测试包装器），可以执行测试。

大多数情况下，不带参数的简单 `ctest` 命令通常就足以获得令人满意的结果。若所有测试都通过，`ctest` 将返回一个 0 退出代码。在 CI/CD 流水中使用此功能，以防止错误提交合并到存储库的生产分支。

编写好的测试可能与编写生产代码本身一样具有挑战性，将 SUT 设置为特定的状态，运行一个测试，然后销毁 SUT 实例。这个过程相当复杂，可能会产生各种各样的问题：交叉测试污染、时间和并发中断、资源争用、死锁导致的停滞，以及许多其他问题。

可以采用有助于发现和解决其中一些问题的策略。CTest 允许选择测试、测试顺序、产生的输出、时间限制、重复次数等。以下各节将提供必要的背景和对最有用的选项的简要概述，请参阅 CMake 文档以获得详尽的列表。

## 查询测试

需要做的第一件事是了解哪些测试实际上是为项目编写的。CTest 提供了一个-N 选项，其禁用执行，只打印一个列表：

```
ctest -N
Test project /tmp/b
Test #1: SumAddsTwoInts
Test #2: MultiplyMultipliesTwoInts
Total Tests: 2
```

希望在下一节中描述的筛选器中使用-N，以检查在应用筛选器时将执行哪些测试。

若需要可以自动化工具使用的 JSON 格式，请使用--show-only=json-v1 执行 ctest。

CTest 还提供了用 LABELS 关键字对测试进行分组的机制，列出所有可用的标签（而不实际执行任何测试），请使用--print-labels。当在列表文件中使用 add\_test(<name> <testcommand>) 指令手动定义测试时，这个选项很有用，可以通过测试属性指定单独的标签：

```
set_tests_properties(<name> PROPERTIES LABELS "<label>")
```

另一方面，稍后将讨论的框架提供自动测试发现，但还不支持这种细粒度级别的标记。

## 过滤测试

有很多理由只运行所有测试的一个子集——最常见的原因可能是需要调试单个失败的测试或正在处理的模块，所以没有必要等待所有其他测试。其他高级的测试场景甚至可以划分测试用例，并将负载分布到测试运行人员的团队中。

这些标志将根据提供的 <r> 正则表达式（正则表达式）筛选测试：

- -R <r>, --tests-regex <r>—只运行名称匹配 <r> 的测试
- -E <r>, --exclude-regex <r>—跳过名称匹配 <r> 的测试
- -L <r>, --label-regex <r>—只运行标签匹配 <r> 的测试
- -LE <r>, --label-exclude <regex>—跳过标签匹配 <r> 的测试

高级场景可以通过--tests-information 选项（或缩写形式-i）实现。使用此过滤器以逗号分隔的格式提供一个范围:<start>, <end>, <step>。任何字段都可以为空，在逗号之后，可以附加单个 <test-id> 值以运行额外的测试。下面是一些例子：

- -I 3,, 将跳过测试 1 和 2(从第三个测试开始执行)

- `-I ,2`, 只运行第一个和第二个测试
- `-I 2,,3` 从一行中的第二个测试开始, 每隔第三个测试运行一次
- `-I ,0,3,9,7` 只运行第三、第九和第七个测试

CTest 可以选择接受包含相同格式的规范的文件名, 用户更喜欢按名称筛选测试。此选项可用于为非常大的套件在多台机器上分发测试。

默认情况下, 与-R一起使用的-I选项将缩小执行范围(同时满足两个需求的测试才会运行)。若需要执行两者的并集(需求都可以满足), 则添加-U选项。

如前所述, 可以使用-N选项检查过滤的结果。

## 乱序测试

编写单元测试可能很棘手。遇到的一个更令人惊讶的问题是测试耦合, 即一个测试通过不完全设置, 或清除 SUT 状态而影响另一个测试的情况。换句话说, 要执行的第一个测试可能“泄漏”状态并污染第二个测试。这样的耦合是坏消息, 因为这引入了测试之间未知的隐式关系。

更糟糕的是, 这种错误可以很好地隐藏在复杂的测试场景中。将导致一个测试在不应该的情况下失败时, 可能会检测到, 但相反的情况也同样可能发生: 不正确的状态导致测试在不应该的情况下通过。这种错误地通过测试给了开发人员一种安全的错觉, 这甚至比根本不进行测试还要糟糕。对代码进行了正确测试的假设可能会鼓励更大胆的行动, 导致意想不到的结果。

发现此类问题的一种方法是单独运行每个测试, 在没有 CTest 的情况下直接从测试框架执行测试运行程序时, 情况就不是这样了。要运行单个测试, 需要向测试可执行文件传递一个特定于框架的参数。这允许检测在套件中通过, 但在单独执行时失败的测试。

另一方面, CTest 通过隐式执行子 CTest 实例中的每个测试用例, 有效地消除了所有基于内存的测试交叉污染。甚至可以添加--force-new-ctest-process 选项来强制使用单独的进程。

但当测试正在使用外部的、有竞争的资源(如 GPU、数据库或文件), 那么仅靠这一点是不起作用的。可以采取另一种预防措施, 简单地随机化测试执行的顺序。这种干扰往往足以最终检测出这种假阴性测试。CTest 通过--schedulerandom 选项支持这种策略。

## 处理失败

John C. Maxwell 有句名言: “尽早失败, 经常失败, 但总是向前失败。”这正是在运行单元测试时想要做的事情(也许在生活的其他领域也是如此)。除非在运行带有调试器的测试时, 否则不容易了解在哪里犯了错误, 因为 CTest 将保持内容简短, 只列出失败的测试, 而不打印任何输出。

由测试用例或 SUT 打印到标准输出的消息对于确定错误的确切位置可能非常有用。要查看它们, 可以使用--output-on-failure 参数。或者, 设置 CTEST\_OUTPUT\_ON\_FAILURE 变量也会有相同的效果。

根据解决方案的大小, 测试失败后停止执行可能有意义。可以通过--stop-on-failure 参数来实现。

CTest 存储失败测试的名称。为了在冗长的测试套件中节省时间, 可以将重点放在这些失败的测试上, 并在问题解决之前跳过运行通过的测试。

该特性是通过--rerun-failed 选项启用的(其他过滤器将忽略)。在解决所有问题后运行所有测试, 以确保在此期间没有引入回归。

当 CTest 没有检测到测试时, 要么没有测试, 要么项目有问题。默认情况下, ctest 将打印一条

警告消息并返回一个 0 退出代码，以避免混淆视听。大多数用户将有足够的上下文来理解他们遇到的情况和下一步要做什么。在某些环境中，`ctest` 将始终作为自动化流水的一部分执行。

然后，可能需要显式地表示，缺乏测试应该解释为错误（并返回非零退出码）。可以通过提供`--no-tests=error` 参数来配置这种行为。对于相反的行为（没有警告），使用`--no-tests=ignore` 选项。

## 重复测试

在职业生涯中，迟早会遇到大多数时候都能正常工作的测试。这些测试极有可能因为环境原因而失败：错误的模拟时间、事件循环问题、异步执行的糟糕处理、并行性、哈希冲突，以及其他并非每次运行都发生的非常复杂的场景。这些不可靠的测试被称为“片状测试”。

这种不一致似乎不是一个重要问题，测试不是真正的生产环境，这是它们有时失败的最终原因。但测试并不意味着复制每一个小细节，因为是不可行的。测试是一种模拟，对可能发生的情况的近似，这通常就足够了。若测试在下次执行时会通过，重新运行测试是否会造成伤害？

事实上，这里概述了三个主要问题：

- 若代码库中收集了足够多的零散测试，将成为代码更改顺利交付的严重障碍。当赶时间的时候，尤其令人沮丧：要么是在周五下午准备回家，要么是在为影响客户的严重问题进行关键修复。
- 不能确定是否由于测试环境的不适当，不可靠的测试正在失败。事实可能恰恰相反，失败是因为复制了生产中已经出现的罕见场景。只是还没有明显到可以拉响警报的程度。
- 不是测试有问题，而是代码有问题！环境有时不稳定，作为开发者，需要以确定的方式处理它。若 SUT 以这种方式运行，这是一个严重的错误——例如，代码可能正在从未初始化的内存中读取。

没有一种完美的方法可以解决上述所有情况，但可以通过`-repeat <mode>:<#>` 选项反复运行测试，从而增加识别不可靠测试的机会。有三种模式可供选择，如下所述：

- `until-fail`—运行测试 `<#>` 次；所有的测试都必须通过。
- `until-pass`—运行测试到 `<#>` 次；至少要通过一次。这在处理已知的不稳定、但太难调试或不能禁用的测试时非常有用。
- `after-timeout`—运行测试直到 `<#>` 次，但只有在测试超时时才重试。可以在负载过高的测试环境中使用。

建议尽可能快地调试不可靠的测试，若不能相信其能产生的结果，最好去掉它们。

## 控制输出

每次将信息打印到屏幕上都会立刻变得繁忙，CTest 减少噪音并收集它执行到日志文件的测试的输出，只提供关于常规运行的最有用的信息。当情况变糟，测试失败时，若启用了`--output-onfailure`，就可以得到一个总结信息和一些执行日志。

从经验中知道，“足够的信息”是足够的，但有时可能也想查看通过的测试的输出，也许是為了检查它们是否真的在工作（而不是无误地默默停止）。要访问更详细的输出，可以使用`-v` 选项（或者`--verbose`，若想在自动化流水中显式显示）。

若这还不够，可能需要`-VV` 或`--extra-verbose`。对于深入的调试，请使用`--debug`（但要准备好面

对包含所有细节的文本墙)。若想要的是相反的, CTest 还提供“禅模式”, 启用`-q` 或`--quiet`。没有输出将打印出来(可以停止担心, 学会享受平静)。这个选项除了迷惑人们之外没有其他用途, 但是要注意输出仍然会存储在测试文件中(默认情况下在`./Testing/Temporary` 中)。自动化流水可以检查退出代码是否为非零值, 并收集日志文件以进行进一步处理, 这些信息不会使不熟悉产品的开发者感到困惑(从而破坏主要输出)。

使用`-O <file>`, `--output-log <file>` 选项将日志存储在指定的路径中。若遇到了冗长的输出, 可以使用两个限制选项来限制每个测试的给定字节数:`--test-output-size-passed <size>` 和`--test-output-size-failed <size>`。

## 其他选项

对于测试需求, 还有其他有用的选项:

- `-C <cfg>, --build-config <cfg>`(仅支持多配置生成器)——指定要测试的配置。`Debug` 配置通常有调试符号, 但是 `Release` 也应该进行测试, 因为优化选项可能会潜在地影响 SUT 的行为。
- `-j <jobs>, --parallel <jobs>`—这将设置并行执行的测试数。这对于在开发过程中加速长时间测试的执行非常有用, 在繁忙的环境中(在共享测试运行器上), 可能会由于调度而产生不利影响。下一个选项可以缓解这种情况
- `--test-load <level>`—使用此选项调度并行测试, 使 CPU 负载不超过`<level>` 值(在尽最大努力的基础上)。
- `--timeout <seconds>`—使用此命令指定单个测试的默认时间限制。

现在已经了解了如何在许多不同的场景中执行 `ctest`。接下来, 让我们学习如何添加一个测试。

## 8.4. 为 CTest 创建最基本的单元测试

编写单元测试在技术上是不需要任何框架, 所要做的就是创建想要测试的类的实例, 执行它的一个方法, 并检查返回的新状态或值是否符合期望。然后, 报告结果并删除测试对象。

将使用以下目录结构:

```
- CMakeLists.txt
- src
 |- CMakeLists.txt
 |- calc.cpp
 |- calc.h
 |- main.cpp
```

```
- test
 |- CMakeLists.txt
 |- calc_test.cpp
```

从 main.cpp 开始，将使用一个 Calc 类：

```
1 // chapter08/01-no-framework/src/main.cpp
2
3 #include <iostream>
4 #include "calc.h"
5 using namespace std;
6
7 int main() {
8 Calc c;
9 cout << "2 + 2 = " << c.Sum(2, 2) << endl;
10 cout << "3 * 3 = " << c.Multiply(3, 3) << endl;
11 }
```

main.cpp 只是包含了 Calc.h 头文件，并调用 Calc 对象的两个方法。快速浏览一下 Calc(我们的 SUT) 的接口：

```
1 // chapter08/01-no-framework/src/calc.h
2
3 #pragma once
4 class Calc {
5 public:
6 int Sum(int a, int b);
7 int Multiply(int a, int b);
8 };
```

接口很简单。这里使用了 #pragma——其工作原理和常见的预处理器包括保护完全一样，尽管不是官方标准，但几乎所有现代编译器都可以识别。让我们看看类的实现：

```
1 // chapter08/01-no-framework/src/calc.cpp
2
3 #include "calc.h"
4 int Calc::Sum(int a, int b) {
5 return a + b;
6 }
7
8 int Calc::Multiply(int a, int b) {
9 return a * a; // a mistake!
10 }
```

啊哦！我们引入了一个错误！乘法忽略 b 参数，返回 a 的平方。应该通过正确编写的单元测试来检测。所以，让我们开始吧！

```
1 // chapter08/01-no-framework/test/calc_test.cpp
2
3 #include "calc.h"
4 #include <cstdlib>
5
6 void SumAddsTwoIntegers() {
7 Calc sut;
8 if (4 != sut.Sum(2, 2))
```

```

9 std::exit(1);
10 }
11
12 void MultiplyMultipliesTwoIntegers() {
13 Calc sut;
14 if(3 != sut.Multiply(1, 3))
15 std::exit(1);
16 }
```

通过编写两个测试方法启动 `calc_test.cpp` 文件，每个 SUT 的测试方法对应一个测试方法。若方法返回的值不符合预期，每个函数将调用 `std::exit(1)`。可以在这里使用 `assert()`、`abort()` 或 `terminate()`，但这将导致在 `ctest` 的输出中出现不太明显的 Subprocess 中止消息，而不是可读性更强的 Failed 消息。

是时候创建一个测试运行程序了。因为正确地运行将需要大量的工作，所以需要尽可能地简单。我们必须编写 `main()` 函数，以运行两个测试：

```

1 chapter08/01-no-framework/test/unit_tests.cpp
2 #include <string>
3 void SumAddsTwoIntegers();
4
5 void MultiplyMultipliesTwoIntegers();
6
7 int main(int argc, char *argv[]) {
8 if (argc < 2 || argv[1] == std::string("1"))
9 SumAddsTwoIntegers();
10
11 if (argc < 2 || argv[1] == std::string("2"))
12 MultiplyMultipliesTwoIntegers();
13 }
```

以下是具体情况：

- 声明两个外部函数，从另一个翻译单元链接。  
若没有提供参数，则执行两个测试 (`argv[]` 中的第 0 个元素始终是程序名)。
- 若第一个参数是测试的标识符，则执行。
- 若测试失败，将在内部调用 `exit()` 并返回退出代码 1。
- 若没有执行测试或所有测试都通过，则隐式返回并带有退出代码 0。

运行第一个测试，执行 `./unit_tests 1`；运行第二个，执行 `./unit_tests 2`。这里简化了代码，但仍然很难阅读。添加了测试之后，可能需要维护这一部分的开发者都不会有很多的精力，更不用说这个功能非常原始——调试这样的测试套件将是一项繁重的工作。来看看如何在 CTest 中使用它：

```

chapter08/01-no-framework/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(NoFrameworkTests CXX)
enable_testing()
```

```
add_subdirectory(src bin)
add_subdirectory(test)
```

通常从 `enable_testing()` 开始，这是告诉 CTest 需要在这个目录及其子目录中启用测试，将在每个子目录中包含两个嵌套的列表文件:src 和 test。突出显示的 bin 表明，需要将 src 子目录的二进制输出放在 `<build_tree>/bin` 中。否则，二进制文件将在 `<build_tree>/src` 中结束，这可能令人困惑。毕竟，构建的工件不是源文件。

src 目录的列表文件非常简单，包含一个简单的主目标定义：

```
chapter08/01-no-framework/src/CMakeLists.txt

add_executable(main main.cpp calc.cpp)
```

还需要一个 test 目录的列表文件：

```
chapter08/01-no-framework/test/CMakeLists.txt

add_executable(unit_tests
 unit_tests.cpp
 calc_test.cpp
 ../src/calc.cpp)
target_include_directories(unit_tests PRIVATE/src)

add_test(NAME SumAddsTwoInts COMMAND unit_tests 1)
add_test(NAME MultiplyMultipliesTwoInts COMMAND unit_tests 2)
```

现在已经定义了第二个 `unit_tests` 目标，也使用 `src/calc.cpp` 实现文件和各自的头文件。最后，显式添加两个测试:`SumAddsTwoInts` 和 `MultiplyMultipliesTwoInts`。每个都将其 ID 作为参数提供给 `add_test()` 指令。CTest 将接受 `COMMAND` 关键字之后提供的内容，并在子 shell 中执行，并收集输出和退出代码。不要太执着于在单元测试框架的 `add_test()`，会有处理测试用例的更好的方法，因此这里将跳过这里的详细描述。

这是 `ctest` 在构建树中执行时的实际方式：

```
ctest
Test project /tmp/b
 Start 1: SumAddsTwoInts
1/2 Test #1: SumAddsTwoInts Passed
0.00 sec
 Start 2: MultiplyMultipliesTwoInts
2/2 Test #2: MultiplyMultipliesTwoInts ***Failed
0.00 sec

50% tests passed, 1 tests failed out of 2
Total Test time (real) = 0.00 sec
The following tests FAILED:
 2 - MultiplyMultipliesTwoInts (Failed)
Errors while running CTest
Output from these tests are in: /tmp/b/Testing/Temporary/
LastTest.log
Use "--rerun-failed --output-on-failure" to re-run the failed
cases verbosely.
```

CTest 执行了两个测试，并报告其中一个测试失败——从 Calc::Multiply 返回的值没有达到预期。现在已知代码有一个 bug，需要修复它。

#### Note

可能已经注意到，大多数例子中都没有采用第 3 章中描述的项目结构，这样做是为了简短。本章讨论更高级的概念，因此使用完整的结构。在项目中（无论多么小），最好从一开始就遵循这个结构。一位智者曾经说过：“当你踏上道路，若不保持脚步的方向，谁也不知道你会到达哪里。”

应该避免将构建测试框架作为项目的一部分。即使是最基本的例子也很难看懂，还有很多开销，但没有任何价值。然而，使用单元测试框架之前，需要重新考虑项目的结构。

#### 8.4.1 为测试构建项目

C++ 有一些有限的内省功能，但不能提供像 Java 那样强大的追溯功能。这可能就是为什么为 C++ 代码编写测试和单元测试框架，比在其他更丰富的环境中要困难得多。这种方法的含义是开发者必须更多地参与编写可测试代码，不仅要更仔细地设计接口，而且还要回答关于实用性的问题，例如：如何避免重复编译，以及在测试和生产之间重用工件？

对于小项目来说，编译时间可能不是问题，但随着时间的流逝，项目会增长。前面的示例中，除了 main.cpp 文件之外，将所有 SUT 源附加到单元测试的可执行文件中。有些读者可能会注意到该文件中有代码没有经过测试（main() 本身的内容）。通过编译两次代码，产生的工件有可能不完全

相同。随着时间的推移，这些东西可能会产生分歧（由于添加了编译标志和预处理器指令）。当为代码库做出贡献的工程师很忙、缺乏经验或对项目不熟悉时，这特别危险。

处理这个问题有多种方法，但最优雅的方法是将整个解决方案构建为一个库，并将其与单元测试链接起来。有读者可能会问：“那要怎么运行它呢？”需要将链接到库，并运行其代码的可执行文件。

首先，将当前的 `main()` 函数重命名为 `run()`、`start_program()` 或类似的东西。然后，使用新的 `main()` 函数创建另一个实现文件 (`bootstrap.cpp`)，并且只使用这个函数。这将是适配器（或者包装器）：唯一作用是提供入口点，并调用 `run()` 转发命令行参数（若有的话）。剩下的就是把所有东西连接起来，就得到了一个可测试的项目。

通过重命名 `main()`，可以将 SUT 与测试联系起来，并测试它的主要功能。否则，将违反第 6 章中讨论的单一定义规则（ODR），因为测试运行程序需要它自己的入口点，单独的 `main()` 函数。

测试框架可以提供自己的 `main()` 函数的实现，因此不需要编写。通常，其将检测我们链接的所有测试，并根据需要配置执行。

这种方法产生的工件可以分为以下目标：

- 生产代码的 sut 库
- 使用 `main()` 包装器从 sut 调用 `run()` 进行引导
- 使用 `main()` 包装器进行单元测试，该包装器在 sut 上运行所有测试

下图显示了目标之间的符号关系：

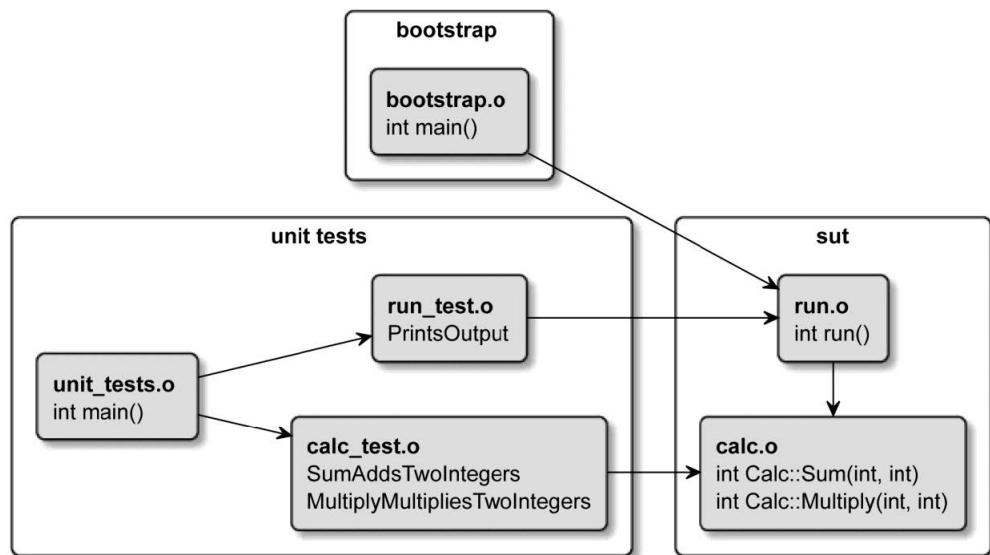


图 8.2 测试和生产可执行程序之间的共享工件

最终得到了 6 个实现文件，它们将产生各自的 (.o) 目标文件：

- `calc.cpp`—进行单元测试的 `Calc` 类。这称为测试单元（UUT），因为 UUT 是 SUT 的特化。
- `run.cpp`—原来的入口点重命名为 `run()`，现在可以对其进行测试。
- `bootstrap.cpp`—新的 `main()` 入口点调用 `run()`。
- `calc_test.cpp`—测试 `Calc` 类。
- `run_test.cpp`—`run()` 的新测试可以放在这里。

- `unit_tests.o`—单元测试的入口点，扩展为 `run()` 的进行测试。

将要构建的库，实际上不是一个实际的库：静态或动态。通过创建对象库，可以避免不必要的归档或链接。从技术上讲，通过依赖 SUT 的动态链接可以节省一些时间，但我们在测试和 SUT 中都做了修改，从而抵消了收益。

来看看文件是如何改变的，从之前的 `main.cpp` 开始：

```

1 // chapter08/02-structured/src/run.cpp
2
3 #include <iostream>
4 #include "calc.h"
5 using namespace std;
6
7 int run() {
8 Calc c;
9 cout << "2 + 2 = " << c.Sum(2, 2) << endl;
10 cout << "3 * 3 = " << c.Multiply(3, 3) << endl;
11 return 0;
12 }
```

没有太多区别：重命名文件和函数。还添加了 `return` 语句，因为编译器不会对非 `main()` 的函数隐式执行此操作。

新的 `main()` 函数如下所示：

```

1 // chapter08/02-structured/src/bootstrap.cpp
2
3 int run(); // declaration
4 int main() {
5 run();
6 }
```

尽可能简单——声明链接器将从另一个翻译单元提供 `run()` 函数，并调用它。旁边的变化是 `src` 列表文件：

```

chapter08/02-structured/src/CMakeLists.txt

add_library(sut STATIC calc.cpp run.cpp)
target_include_directories(sut PUBLIC .)

add_executable(bootstrap bootstrap.cpp)
target_link_libraries(bootstrap PRIVATE sut)
```

首先，创建了 `sut` 库并进行标记。作为一个 `PUBLIC` 包含目录，这样将传播到所有将链接 `sut`（即引导和 `unit_tests`）的目标。注意，`include` 目录是相对于列表文件的，因此可以使用点（`.`）来引用当前的 `<source_tree>/src` 目录。

是时候更新 `unit_tests` 目标了，删除对 `../src/calc.cpp` 文件的直接引用，并为 `unit_tests` 目标添加到 `sut` 的链接引用，还可以将在 `run_test.cpp` 文件中为主函数添加一个新的测试。简单起见，这里可以跳过，但若感兴趣，可以查看在线示例。下面是整个测试列表文件：

```
chapter08/02-structured/test/CMakeLists.txt

add_executable(unit_tests
 unit_tests.cpp
 calc_test.cpp
 run_test.cpp)
target_link_libraries(unit_tests PRIVATE sut)
```

需要注册新的测试:

```
add_test(NAME SumAddsTwoInts COMMAND unit_tests 1)
add_test(NAME MultiplyMultipliesTwoInts COMMAND unit_tests 2)
add_test(NAME RunOutputsCorrectEquations COMMAND unit_tests 3)
```

完成了! 通过遵循此实践可以确保在生产中使用的机器代码上执行测试。

#### Note

这里使用的目标名称 `sut` 和 `bootstrap`, 是为了从测试的角度清楚地说明其功能, 应该选择与生产代码(而不是测试)上下文相匹配的名称。例如, 对于一个 `FooApp`, 目标名是 `foo`, 而不是 `bootstrap`, `lib_foo` 而不是 `sut`。

现在知道了如何在适当的目标中构建可测试项目, 这里将焦点转移到测试框架本身。我们不希望手动地将每个测试用例添加到列表文件中, 对吧?

## 8.5. 单元测试框架

上一节证明了编写一个小型单元测试驱动程序并不复杂, 但专业开发人员实际上喜欢重新发明轮子(会比传统的更漂亮、更圆、更快)。不要落入这个陷阱: 这样会创建太多的样板文件, 可能会成为一个独立的项目。在解决方案中引入流行的单元测试框架, 使其与超越项目和公司的标准保持一致, 可以以低廉的成本提供免费的更新和扩展。你, 值得拥有!

如何将单元测试框架添加到项目中? 根据所选框架的规则在实现文件中编写测试, 并测试与框架提供的测试运行器链接起来。测试运行程序是将开始执行选定测试的入口点。与前面看到的 `unit_tests.cpp` 文件不同, 它们中的许多将自动检测所有测试。很漂亮!

我会在本章中介绍两个单元测试框架, 选择他们的原因如下所示:

- Catch2 是一个相对容易学习、支持良好且有文档记录的项目。提供了简单的测试用例, 但也为行为驱动开发(BDD)提供了优雅的宏。但缺少一些功能, 可以在需要时与外部工具结合使用。可以浏览其主页:<https://github.com/catchorg/Catch2>。
- GTest 也非常方便, 但更高级。主要特性是具有丰富的断言、用户定义的断言、死亡测试、致命和非致命失败、值和类型参数化测试、XML 测试报告生成和模拟。最后一个是在 GMock 模块中交付的, 可从相同的存储库获得:<https://github.com/google/googletest>。

应该选择哪种框架取决于自己的偏好和项目的规模。若更喜欢一个缓慢、渐进的过程，不需要那些花哨的东西，那就用 Catch2 吧。喜欢从深层入手，并有大量开发人员可用的话，可以选择 GTest。

### 8.5.1 Catch2

这由 Martin 维护的框架 Hořenovský 非常适合初学者和较小的项目。这并不是说它不能处理更大的应用程序，在某些情况可能需要额外的工具。若详细地讲下去，就会偏离本书的主题，但还是简单进行一下概述。首先，可以为 Calc 类编写单元测试的实现：

```
1 //chapter08/03-catch2/test/calc_test.cpp
2
3 #include <catch2/catch_test_macros.hpp>
4
5 #include "calc.h"
6 TEST_CASE("SumAddsTwoInts", "[calc]") {
7 Calc sut;
8 CHECK(4 == sut.Sum(2, 2));
9 }
10
11 TEST_CASE("MultiplyMultipliesTwoInts", "[calc]") {
12 Calc sut;
13 CHECK(12 == sut.Multiply(3, 4));
14 }
```

这几行代码比前面的例子中的测试强大得多。CHECK() 宏不仅会验证是否满足预期——实际上会收集所有失败的断言，并显示在单个输出中，以便可以进行修复，并避免重新编译。

现在，不需要添加这些测试，甚至不需要通知 CMake；可以忘记 add\_test()，Catch2 将自动向 CTest 注册测试。在配置了项目之后，添加框架非常容易，只需要使用 FetchContent() 将其引入项目即可。

有两个主要版本可供选择：v2 和 v3。版本 2 作为 C++11 的单头库（仅 #include <catch2/catch.hpp>）提供，最终将在版本 3 中弃用。它有多个头文件，可以编译为静态库，需要支持 C++14。当然，若还没使用现代 C++（C++11 不再认为是“现代的”）编译器，建议使用更新的版本。使用 Catch2 时，应该选择一个 Git 提交，并将其固定在列表文件中。换句话说，这里不能保证升级不会破坏测试的代码（很可能不会，但若不需要，就不要冒险使用 devel 分支）。为了获取 Catch2，需要向存储库提供一个 URL：

```
chapter08/03-catch2/test/CMakeLists.txt
include(FetchContent)
FetchContent_Declare(
 Catch2
 GIT_REPOSITORY https://github.com/catchorg/Catch2.git
 GIT_TAG v3.0.0
)
FetchContent_MakeAvailable(Catch2)
```

然后，定义 `unit_tests` 目标，并将其与 `sut` 和框架提供的入口点，以及 `Catch2::Catch2WithMain` 库链接起来。因为 `Catch2` 提供了 `main()`，所以这里不再使用 `unit_tests.cpp` 文件（这个文件可以删除）。代码如下所示：

```
chapter08/03-catch2/test/CMakeLists.txt (continued)

add_executable(unit_tests
 calc_test.cpp
 run_test.cpp)
target_link_libraries(unit_tests PRIVATE
 sut Catch2::Catch2WithMain)
```

最后，使用 `catch_discover_tests()` 指令，在 `Catch2` 提供的模块中，将检测 `unit_tests` 中所有测试用例，并将它们注册到 `CTest`：

```
#chapter08/03-catch2/test/CMakeLists.txt (continued)

list(APPEND CMAKE_MODULE_PATH ${catch2_SOURCE_DIR}/extras)
include(Catch)
catch_discover_tests(unit_tests)
```

完成了！我们已经在解决方案中添加了一个单元测试框架。现在来看看在实践中的情况。测试运行器的输出：

```
./test/unit_tests
unit_tests is a Catch v3.0.0 host application.
Run with -? for options

MultiplyMultipliesTwoInts

examples/chapter08/03-catch2/test/calc_test.cpp:9
.
.
.
examples/chapter08/03-catch2/test/calc_test.cpp:11: FAILED:
 CHECK(12 == sut.Multiply(3, 4))
with expansion:
 12 == 9
=====
test cases: 3 | 2 passed | 1 failed
assertions: 3 | 2 passed | 1 failed
```

直接执行运行程序（已编译的 `unit_test` 可执行文件）稍微快一些，通常情况下，应该使用 `ctest --output-on-failure`，而不是直接执行测试运行程序，以获得前面提到的所有 `ctest` 好处。注意，`Catch2`

能够方便地扩展 sut。

这就是 Catch2 的设置。若需要添加更多的测试，只需创建实现文件，并将其路径插入到 unit\_tests 目标的源列表中即可。

这个框架有很多有趣的技巧：事件监听器、数据生成器和微基准测试，但没有提供模拟功能。若不知道 mock 是什么的读者，请继续阅读。然而，若发现自己需要模拟，可以在 Catch2 旁边添加一个模拟框架：

- FakeIt (<https://github.com/eranpeer/FakeIt>)
- Hippomocks (<https://github.com/dascandy/hippomocks>)
- Trompeloeil (<https://github.com/rollbear/trompeloeil>)

对于更精简、更高级的体验，还有另一个框架值得了解。

### 8.5.2 GTest

使用 GTest 有几个优点：在 C++ 社区中得到了高度认可（多个 IDE 本身就支持）。这个星球上最大的搜索引擎公司在维护和使用它，所以其不太可能在短时间内变得过时或抛弃。可以测试 C++11 及以上版本，所以若使用旧一些的环境，也不必担心。

GTest 存储库包含两个项目：GTest（主要测试框架）和 GMock（一个添加了模拟功能的库），可以通过 FetchContent() 来下载。

#### 使用 GTest

项目需要遵循结构化测试的指导，下面是如何在这个框架中编写单元测试：

```
1 // chapter08/04-gtest/test/calc_test.cpp
2
3 #include <gtest/gtest.h>
4 #include "calc.h"
5
6 class CalcTestSuite : public ::testing::Test {
7 protected:
8 Calc sut_;
9 };
10
11 TEST_F(CalcTestSuite, SumAddsTwoInts) {
12 EXPECT_EQ(4, sut_.Sum(2, 2));
13 }
14
15 TEST_F(CalcTestSuite, MultiplyMultipliesTwoInts) {
16 EXPECT_EQ(12, sut_.Multiply(3, 4));
17 }
```

这个例子也将在 GMock 中使用，可以将测试放在单独的 CalcTestSuite 类中。测试套件是与组相关的测试，可以重用相同的字段、方法、设置（初始化）和拆卸（清理）步骤。要创建测试套件，需要声明继承自 ::testing::Test 的新类，并将重用的元素（字段、方法）放在其 protected 部分中。

测试套件中的每个测试用例都用 TEST\_F() 预处理器宏声明，该宏将测试套件和测试用例提供

的名称字符串化(还有简单的 TEST() 宏, 定义了不相关的测试)。因为在类中定义了 Calc sut\_, 所以每个测试用例都可以访问, 就好像测试是 CalcTestSuite 的一个方法一样。实际上, 每个测试用例都在自己的类中运行, 隐式地继承自 CalcTestSuite(这就是需要 protected 的原因)。注意, 重用字段并需要在连续的测试之间共享数据——其功能是为了保持代码 DRY。

GTest 不像 Catch2 那样为断言提供自然的语法。而需要使用显式比较, 例如 EXPECT\_EQ()。通常, 将期望值作为第一个参数, 将实际值作为第二个参数。还有许多其他的断言、帮助器和宏值得学习。

Note

有关 GTest 的详细信息, 请参阅官方参考资料 (<https://google.github.io/googletest/>)

要将这个依赖项添加到我们的项目中, 需要决定使用哪个版本。与 Catch2 不同, GTest 倾向于“live at head”的理念(源自 GTest 所依赖的 Abseil 项目), 其指出:“若从源代码构建依赖并遵循 API, 应该不会有问题。”(更多细节请参考扩展阅读部分。)

若遵循此规则(并且从源代码构建也不是问题), 可以将 Git 标记设置为主分支。否则, 从 GTest 存储库中选择一个版本。还可以选择首先在主机上搜索已安装的副本, 因为 CMake 提供了一个绑定的 FindGTest 模块来查找本地安装。从 3.20 版本开始, CMake 将使用上游的 GTestConfig.cmake 配置文件, 将替换依赖于查找模块的方式。

GTest 上添加依赖:

```
chapter08/04-gtest/test/CMakeLists.txt

include(FetchContent)
FetchContent_Declare(
 googletest
 GIT_REPOSITORY https://github.com/google/googletest.git
 GIT_TAG master
)
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
```

遵循与 Catch2 执行 FetchContent() 的方法, 并从源代码构建框架。唯一的区别是添加了 set(gtest ...) , 这是 gtest 作者推荐的, 以防止在 Windows 上覆盖父项目的编译器和链接器设置。

最后, 可以声明可执行的测试运行器, 并链接到 gtest\_main, 并通过内置的 CMake GoogleTest 模块自动找到我们的测试用例:

```
chapter08/04-gtest/test/CMakeLists.txt (continued)

add_executable(unit_tests
 calc_test.cpp
 run_test.cpp)
target_link_libraries(unit_tests PRIVATE sut gtest_main)
```

```
include(GoogleTest)
gtest_discover_tests(unit_tests)
```

这就完成了 GTest 的设置。测试运行器的输出比 Catch2 的输出要详细得多，可以通过`--gtest_brief=1` 将其限制为仅适用于失败用例：

```
./test/unit_tests --gtest_brief=1
~/examples/chapter08/04-gtest/test/calc_test.cpp:15: Failure
Expected equality of these values:
12
sut_.Multiply(3, 4)
Which is: 9
[FAILED] CalcTestSuite.MultiplyMultipliesTwoInts (0 ms)
[=====] 3 tests from 2 test suites ran. (0 ms total)
[PASSED] 2 tests.
```

从 CTest 运行时，会抑制噪声输出 (除非使用 `ctest --output-on-failure` 显式地启用)。

现在已经有了框架，再来讨论一下 mock。毕竟，当测试与其他元素结合在一起时，不可能是真正的“单元”。

### 8.5.3 GMock

编写真正的单元测试是要与其他代码隔离使执行的代码，这样的单元可以理解为是一个自包含的元素，要么是类，要么是组件。当然，若没有用 C++ 编写的程序，那么所有单元都是与其他单元隔离。最有可能的是，代码将严重依赖于类之间的某种形式的关系。这样做只有一个问题：类对象将需要另一个类对象，而另一个类又需要另一个类对象；从而，整个解决方案都参与了“单元测试”。更糟糕的是，代码可能与外部系统耦合，并依赖于它的状态——例如，数据库中的特定记录、传入的网络数据包，或存储在磁盘上的特定文件。

为了解耦单元以进行测试，开发人员使用测试副本或测试类使用的特殊版本的类。例子包括 `fake`, `stub` 和 `mock`。以下是一些粗略的定义：

- `fake` 是一些更复杂类的有限实现。一个例子可能是内存中的映射，而不是实际的数据库客户端。
- `stub` 为方法调用提供特定的、固定的回答，仅限于测试使用的响应。还可以记录调用了哪些方法，以及调用了多少次。
- `mock` 是 `stub` 的扩展版本，其还将验证在测试期间是否按预期调用了方法。

这样的测试双精度对象在测试开始时创建，并作为参数提供给测试类的构造函数，以代替实际对象。这种机制称为“依赖注入”。简单测试双精度的问题是它们太简单了。为了模拟不同测试场景的行为，必须提供许多不同的 `double`，一个用于耦合对象可能出现的状态。这不是很现实，而且会将测试代码分散到太多的文件中。这就是 GMock 的用武之地了：允许开发人员为特定的类创建通

用测试副本，并为每行测试定义其行为。GMock 称这些双精度对象为“mock”，但它们是上述所有类型的混合，具体取决于场合。

考虑下面的例子：向 Calc 类添加一个函数，向提供的参数添加一个随机数。其由 AddRandomNumber() 方法表示，该方法将这个和作为 int 返回。如何确认返回值确实是某个随机值和提供给类的值的精确和？依赖随机性是许多重要过程的关键，若不正确地使用，可能会有很严重的后果。检查所有的随机数，直到穷尽所有是不可能的。

为了测试，需要在一个可以模拟的类中包装一个随机数生成器（或者，用模拟替换）。mock 允许强制一个特定的响应，即“假”生成一个随机数。Calc 将在 AddRandomNumber() 中使用该值，并允许检查从该方法返回的值是否符合预期。随机数生成与另一个单元的清晰分离是一种方式（因为可以将一种类型的生成器交换为另一种类型的生成器）。

从抽象生成器的公共接口开始，可以在实际的生成器和 mock 中实现，使我们能够交替使用。我们将执行以下代码：

```
1 // chapter08/05-gmock/src/rng.h
2
3 #pragma once
4
5 class RandomNumberGenerator {
6 public:
7 virtual int Get() = 0;
8 virtual ~RandomNumberGenerator() = default;
9 }
```

实现此接口的类将为我们提供来自 Get() 方法的随机数。注意虚拟关键字——必须在所有需要模拟的方法上，除非想涉及更复杂的基于模板的模拟。还需要记住添加一个虚析构函数。接下来，必须扩展 Calc 类来接受和存储生成器：

```
1 // chapter08/05-gmock/src/calc.h
2
3 #pragma once
4 #include "rng.h"
5
6 class Calc {
7 RandomNumberGenerator* rng_;
8 public:
9 Calc(RandomNumberGenerator* rng);
10 int Sum(int a, int b);
11 int Multiply(int a, int b);
12 int AddRandomNumber(int a);
13 }
```

包含了头文件并添加了一个方法来提供随机添加，还创建了一个用于存储指向生成器的指针的字段，以及一个参数化的构造函数。这就是依赖注入在实践中的工作方式。实现了这些方法：

```
1 // chapter08/05-gmock/src/calc.cpp
2
3 #include "calc.h"
4
5 Calc::Calc(RandomNumberGenerator* rng) {
```

```

6 rng_ = rng;
7 }
8
9 int Calc::Sum(int a, int b) {
10 return a + b;
11}
12
13 int Calc::Multiply(int a, int b) {
14 return a * b; // now corrected
15}
16
17 int Calc::AddRandomNumber(int a) {
18 return a + rng_->Get();
19}

```

构造函数中，指针赋值给类字段。使用 AddRandomNumber() 中的这个字段来获取生成的值，生产代码将使用实数生成器，这些测试将使用模拟，需要解引用指针来启用多态性，也可以为不同的实现创建不同的生成器类。这里只需要一个：具有均匀分布的 Mersenne Twister 伪随机数生成器：

```

1 // chapter08/05-gmock/src/rng_mt19937.cpp
2
3 #include <random>
4 #include "rng_mt19937.h"
5 int RandomNumberGeneratorMt19937::Get() {
6 std::random_device rd;
7 std::mt19937 gen(rd());
8 std::uniform_int_distribution<> distrib(1, 6);
9 return distrib(gen);
10}

```

这段代码的效率不是很高，但对于这个简单的示例来说已经足够了。其目的是生成从 1 到 6 的数字并将它们返回给调用者。这个类的头文件非常简单：

```

1 // chapter08/05-gmock/src/rng_mt19937.h
2
3 #include "rng.h"
4 class RandomNumberGeneratorMt19937
5 : public RandomNumberGenerator {
6 public:
7 int Get() override;
8 }

```

生产代码中使用的方式：

```

1 // chapter08/05-gmock/src/run.cpp
2
3 #include <iostream>
4 #include "calc.h"
5 #include "rng_mt19937.h"
6
7 using namespace std;

```

```

8 int run() {
9 auto rng = new RandomNumberGeneratorMt19937();
10 Calc c(rng);
11 cout << "Random dice throw + 1 = "
12 << c.AddRandomNumber(1) << endl;
13 delete rng;
14 return 0;
15 }

```

这里已经创建了一个生成器，并将指向它的指针传递给 Calc 的构造函数。一切准备就绪，可以开始编写 mock 了。为了使事情有条理，开发人员通常将模拟放在一个单独的 test/mock 目录中。为了防止歧义，头文件名有一个 \_mock 后缀。下面是我们要执行的代码：

```

1 // chapter08/05-gmock/test/mocks/rng_mock.h
2
3 #pragma once
4 #include "gmock/gmock.h"
5
6 class RandomNumberGeneratorMock : public
7 RandomNumberGenerator {
8 public:
9 MOCK_METHOD(int, Get, (), (override));
10 }

```

添加了 gmock.h 头文件之后，可以声明我们的 mock。按计划，它是一个实现 RandomNumberGenerator 接口的类。不需要自己编写方法，而是需要使用 GMock 提供的 MOCK\_METHOD 宏，告诉框架应该模拟接口中的哪些方法。使用以下格式(注意括号)：

```

1 MOCK_METHOD(<return type>, <method name>,
2 (<argument list>), (<keywords>))

```

我们准备在测试套件中使用模拟(为了简洁起见，省略了之前的测试用例)：

```

1 // chapter08/05-gmock/test/calc_test.cpp
2
3 #include <gtest/gtest.h>
4 #include "calc.h"
5 #include "mocks/rng_mock.h"
6
7 using namespace ::testing;
8 class CalcTestSuite : public Test {
9 protected:
10 RandomNumberGeneratorMock rng_mock_;
11 Calc sut_{&rng_mock_};
12
13 };
14 TEST_F(CalcTestSuite, AddRandomNumberAddsThree) {
15 EXPECT_CALL(rng_mock_,
16 Get()).Times(1).WillOnce(Return(3));
17 EXPECT_EQ(4, sut_.AddRandomNumber(1));
18 }

```

分析一下这些更改: 加了新的头文件, 并在测试套件中为 `rng_mock_` 创建了一个新字段。接下来, 模拟的地址传递给 `sut_` 的构造函数, 因为字段是按声明顺序初始化的 (`rng_mock_` 在 `sut_` 之前)。

测试用例中, 在 `rng_mock_` 的 `Get()` 方法上调用 GMock 的 `EXPECT_CALL` 宏, 若在执行期间没有调用 `Get()` 方法, 则测试失败。Times 链式调用显式地声明了, 通过测试必须发生多少个调用。`WillOnce` 确定在调用方法后模拟框架做什么 (它返回 3)。

通过使用 GMock, 就能够在预期结果的同时表达模拟的行为。这极大地提高了可读性并简化了测试的维护。最重要的是, 它在每个测试用例中都提供了足够的弹性, 可以区分单个表达语句所发生的情况。

最后, 需要确保 `gmock` 库与测试运行器链接。为了实现这一点, 将它添加到 `target_link_libraries()` 中:

```
chapter08/05-gmock/test/CMakeLists.txt

include(FetchContent)
FetchContent_Declare(
 gtest
 GIT_REPOSITORY https://github.com/google/googletest.git
 GIT_TAG release-1.11.0
)
For Windows: Prevent overriding the parent project's
compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(gtest)
add_executable(unit_tests
 calc_test.cpp
 run_test.cpp)
target_link_libraries(unit_tests PRIVATE sut gtest_main
gmock)
include(GoogleTest)
gtest_discover_tests(unit_tests)
```

现在, 可以享受 GTest 框架了。GTest 和 GMock 都是非常高级的工具, 具有用于不同场合的概念、程序和辅助程序。这个例子(尽管有点长)只触及了表面, 但我鼓励您将它们合并到项目中, 因为这将极大地提高项目的代码质量。从官方文档中的 `mock for Dummies` 页面开始学习 GMock 的一个好地方(可以在扩展阅读部分中找到该页面的链接)。

有了合适的测试, 应该以某种方式衡量哪些测试了, 哪些没有, 并努力改善情况。最好使用自动工具来收集和报告这些信息。

## 8.6. 生成测试覆盖报告

向如小解决方案添加测试并不具有挑战性，真正的困难来自于高级和较长的程序。多年来，我发现当代码接近 1000 行时，慢慢地就很难跟踪哪些行和分支在测试期间执行了，跨越 3000 行之后，这几乎是不可能的，大多数专业应用程序的代码都比这多得多。为了处理这个问题，可以使用一个实用工具来理解测试用例“覆盖”了哪些代码行。这样的代码覆盖工具与 SUT 挂钩，并在测试期间收集关于每一行执行的信息，以便将其显示在一个报告中：

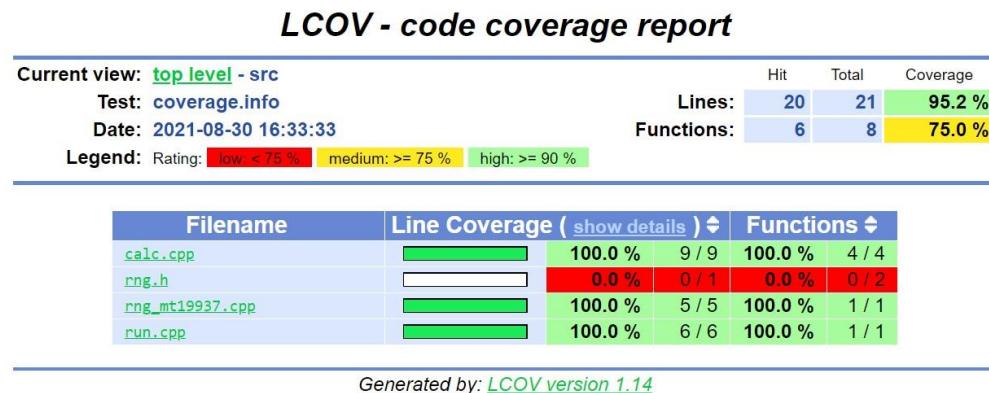


图 8.3 由 LCOV 生成的代码覆盖率报告

这些报告将显示测试覆盖了哪些文件，哪些没有。还可以查看每个文件的详细信息，并查看具体执行了哪些代码行以及执行了多少次。下面的截图中，Line 数据列表示 Calc 构造函数运行了 4 次，每个测试一次：

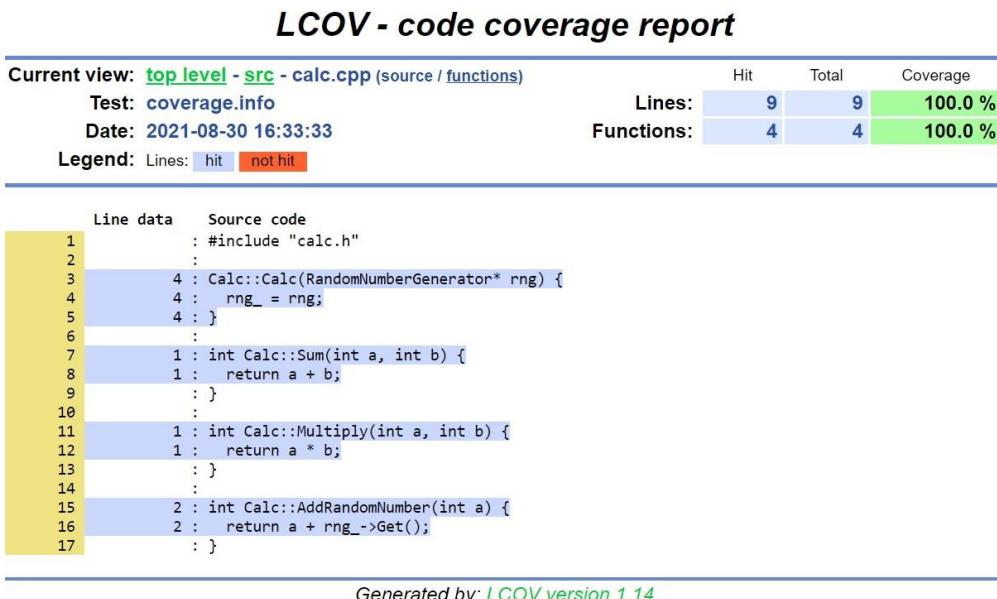


图 8.4 代码覆盖率报告的详细视图

有多种生成类似报告的方法，不同的平台和编译器中有所不同，但通常遵循相同的过程：准备要度量的 SUT，并获得基线、指标和报告。

最简单的工具叫做 LCOV，是 gcov 的图形前端，gcov 是来自 GNU 编译器集合 (GCC) 的覆盖实用程序。LCOV 将生成 HTML 覆盖率报告，并在内部使用 gcov 度量覆盖率。若正在使用 Clang，不必担心——Clang 支持以这种格式生成指标。可以从 Linux 测试项目 (<https://github.com/linux-test-project/lcov>) 维护的官方存储库中获得 LCOV，或者简单地使用包管理器。其为一个针对 Linux 的实用程序，也可以在 macOS 上运行它，但不支持 Windows 平台。最终用户通常不关心测试覆盖率，可以在构建环境中手动安装 LCOV，而不是将其绑定到项目中。

为了衡量覆盖率，需要做以下工作：

1. Debug 配置中编译，使用编译器标志启用代码覆盖。这将生成覆盖率注释 (.gcno) 文件。
2. 将测试可执行文件与 gcov 库链接起来。
3. 不运行测试的情况下，收集基线的覆盖率指标。
4. 运行测试。这将创建覆盖率数据 (.gcda) 文件。
5. 将指标收集到聚合的信息文件中。
6. 生成一个 (.html) 报告。

应该从解释为什么必须在 Debug 配置中编译代码开始。最重要的原因是，调试配置通常禁用任何带有-O0 标志的优化。CMake 默认在 CMAKE\_CXX\_FLAGS\_DEBUG 变量中这样做 (在文档中没有说明)。除非决定重写此变量，否则调试版本应该未优化。这是为了防止任何内联和其他类型的隐式代码简化。否则，将很难跟踪哪条机器指令来自哪行源代码。

首先，需要指示编译器将必要的工具添加到 SUT 中。要添加的确切标志是编译器特定的；然而，两个主要的编译器——gcc 和 clang——提供了相同的覆盖率标志，可以生成与 gcc 兼容的 gcov 数据。

将代码覆盖工具添加到上一节的示例 SUT 中：

```
chapter08/06-coverage/src/CMakeLists.txt

add_library(sut STATIC calc.cpp run.cpp rng_mt19937.cpp)
target_include_directories(sut PUBLIC .)
if (CMAKE_BUILD_TYPE STREQUAL Debug)
 target_compile_options(sut PRIVATE --coverage)
 target_link_options(sut PUBLIC --coverage)
 add_custom_command(TARGET sut PRE_BUILD COMMAND
 find ${CMAKE_BINARY_DIR} -type f
 -name '*.gcda' -exec rm {} +
)
endif()

add_executable(bootstrap bootstrap.cpp)
target_link_libraries(bootstrap PRIVATE sut)
```

一步步来分解：

1. 确保正在使用 if(STREQUAL) 在 Debug 配置中运行。记住，除非运行带有-DCMAKE\_BUILD\_TYPE=Debug 选项的 cmake，否则无法获得覆盖率。

2. 为 sut 库中的所有目标文件的 PRIVATE 编译选项添加--coverage。
3. 在 PUBLIC 链接器选项中添加--coverage:GCC 和 Clang 都将此解释为将 gcov(或兼容的) 库与所有依赖 sut(由于传播属性) 的目标链接起来的请求。
4. 引入 add\_custom\_command() 指令清除过时的.gcda 文件。添加此命令的原因，将在避免 SEGFAULT 问题时详细讨论。

这足以产生代码覆盖率。若正在使用诸如 Clion 之类的 IDE，将能够在覆盖范围内运行单元测试，并在内置报告视图中获得结果。然而，这可能在 CI/CD 中运行的自动化流水中都起作用。为了得到报告，需要用 LCOV 生成报告。

为了达到这个目的，可以定义一个叫做覆盖率的新目标。为了保持整洁，可以在测试列表文件中使用的另一个文件中定义一个函数 AddCoverage，如下所示：

```
chapter08/06-coverage/cmake/Coverage.cmake

function(AddCoverage target)
 find_program(LCOV_PATH lcov REQUIRED)
 find_program(GENHTML_PATH genhtml REQUIRED)

 add_custom_target(coverage
 COMMENT "Running coverage for ${target}..."
 COMMAND ${LCOV_PATH} -d . --zerocounters
 COMMAND ${LCOV_PATH} -d . --capture -o coverage.info
 COMMAND ${LCOV_PATH} -r coverage.info '/usr/include/*'
 COMMAND ${GENHTML_PATH} -o coverage filtered.info
 --legend
 COMMAND rm -rf coverage.info filtered.info
 WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
)
endfunction()
```

首先检测 lcov 和 genhtml(来自 LCOV 包的两个命令行工具) 的路径。REQUIRED 关键字指示 CMake 在未找到它们时抛出错误。接下来，按照以下步骤添加自定义覆盖率目标：

1. 清除以前运行的所有计数器。
2. 运行目标可执行文件(使用生成器表达式获取其路径)。\$<target\_file: target> 是一个异常生成器表达式，将隐式地在 TARGET 上添加一个依赖项，导致在执行命令之前构建。这里提供 target 作为该函数的参数。
3. 在系统头文件('/usr/include/\*') 上删除(-r) 不需要的覆盖数据，并输出到另一个文件(-o filtered.info)。
4. 覆盖率目录中生成一个 HTML 报告，并为一个--legend 添加颜色。
5. 删除临时的.info 文件。

## 6. 指定 WORKING\_DIRECTORY 关键字，将构建树设置为所有命令的工作目录。

这些是 GCC 和 Clang 的一般步骤，gcov 工具的版本必须与编译器的版本匹配，不能对 clang 编译的代码使用 GCC 的 gcov 工具。要将 lcov 指向 Clang 的 gcov 工具，可以使用--gcov-tool 参数。这里的问题是必须是一个可执行文件。为了解决这个问题，可以提供一个简单的包装器脚本(记得用 chmod +x 将其标记为可执行文件):

```
cmake/gcov-llvm-wrapper.sh

#!/bin/bash
exec llvm-cov gcov "$@"
```

在上一个函数中对 \${LCOV\_PATH} 的所有调用都应该获得以下标志:

```
--gcov-tool ${CMAKE_SOURCE_DIR}/cmake/gcov-llvm-wrapper.sh
```

确保这个函数可以包含在测试列表文件中，可以在主列表文件中扩展 include 的搜索路径:

```
chapter08/06-coverage/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Coverage CXX)
enable_testing()
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
add_subdirectory(src bin)
add_subdirectory(test)
```

这一小行允许在项目中包含 cmake 目录中的所有.cmake 文件。现在可以在测试列表文件中使用 Coverage.cmake:

```
chapter08/06-coverage/test/CMakeLists.txt (fragment)

... skipped unit_tests target declaration for brevity

include(Coverage)
AddCoverage(unit_tests)

include(GoogleTest)
gtest_discover_tests(unit_tests)
```

要构建这个目标，使用以下命令(注意第一个命令以 DCMAKE\_BUILD\_TYPE=Debug 构建类型选择结束):

```
cmake -B <binary_tree> -S <source_tree>
-DCMAKE_BUILD_TYPE=Debug
cmake --build <binary_tree> -t coverage
```

执行上述所有步骤后，将看到如下简短的输出：

```
Writing directory view page.
Overall coverage rate:
 lines.....: 95.2% (20 of 21 lines)
 functions...: 75.0% (6 of 8 functions)
[100%] Built target coverage
```

接下来，在浏览器中打开 `coverage/index.html` 文件并欣赏报告！不过有一个小问题……

### 8.6.1 避免 SEGFAULT 问题

开始在这样的解决方案中编辑源代码时可能会陷入麻烦。这是因为覆盖率信息分为了两部分：

- `gcno` 文件，或 GNU 覆盖注释，在编译 SUT 期间生成
- `gcda` 文件，或 GNU 覆盖数据，在测试运行期间生成和更新

“更新”功能是段错误的潜在来源。最初运行测试之后，会留下一堆 `gcda` 文件，这些文件不会删除。若对源代码做一些更改并重新编译目标文件，就会创建新的 `gcno` 文件。然而，没有删除步骤——旧的 `gcda` 文件仍然遵循陈旧的源文件。当执行 `unit_tests` 二进制文件时（发生在 `gtest_discover_tests` 宏中），覆盖率信息文件将不匹配，将收到一个 SEGFAULT(段错误) 错误。

为了避免这个问题，应该删除过时的 `gcda` 文件。由于 `sut` 实例是一个 STATIC 库，可以将 `add_custom_command(TARGET)` 与构建事件相关。清理将在重新构建开始之前执行。

在扩展阅读部分可以找到更多信息的链接。

## 8.7. 总结

看起来与适当的测试相关的复杂性似乎很大，不值得付出努力。有多少代码在没有测试的情况下运行，主要的争论是，测试软件是一项令人生畏的努力。若手动完成，情况会更糟。不幸的是，若没有严格的自动化测试，代码中问题的可见性都是不完整的或不存在的。未测试的代码通常编写得更快（并非总是如此），但读取、重构和修复肯定要慢得多。

本章中，概述了从一开始就进行测试的关键原因。其中最重要的是心理健康和良好的睡眠，没有一个开发人员躺在床上想：我等不及几个小时后被叫醒，去解决问题和 bug。但严肃地说：在将错误部署到生产环境之前，捕获它们可以挽救您（和公司）的生命。

说到测试实用程序，CMake 真正展示了它的实力。CTest 在检测错误测试（隔离、乱序、重复、超时）方面可以发挥神奇的作用。所有这些技术都非常方便，可以通过命令行中的一个简单标志直

接获得。还学习了如何使用 CTest 来列出测试、筛选它们并控制测试用例的输出，现在知道了全面采用标准解决方案的真正力量。用 CMake 构建的项目都可以使用相同的方式测试，而不需要细究其内部细节。

接下来，我们构建了项目，以简化测试过程，并在生产代码和测试运行程序之间重用相同的目标文件。编写自己的测试运行程序是很有趣的，但这里应该把重点放在程序解决的实际问题上，并投入时间采用流行第三方测试框架。

这里，我们学习了 Catch2 和 GTest 的基本知识。进一步深入研究了 GMock 库的细节，了解了测试双精度的工作原理，从而使真正的单元测试成为可能。最后，用 LCOV 设置了一些报告。毕竟，没有什么比硬数据更能证明解决方案是经过充分测试的了。

下一章中，将讨论更多有用的工具，以提高源代码的质量，并发现未知的问题。

## 8.8. 扩展阅读

更多资料，请访问以下链接：

- CMake 文档的 CTest  
<https://cmake.org/cmake/help/latest/manual/ctest.1.html>
- Catch2 文档：  
<https://github.com/catchorg/Catch2/blob-devel/docs/cmakeintegration.md>  
<https://github.com/catchorg/Catch2/blob-devel/docs/tutorial.md>
- GMock 教程：  
[https://google.github.io/googletest/gmock\\_for\\_dummies.html](https://google.github.io/googletest/gmock_for_dummies.html)
- Abseil：  
<https://abseil.io/>
- 与 Abseil 一起：  
<https://abseil.io/about/philosophy#we recommend that you choose to live-at-head>
- 为什么 Abseil 会成为 GTest 的依赖项：  
<https://github.com/google/googletest/issues/2883>
- GCC 的覆盖率：  
<https://gcc.gnu.org/onlinedocs/gcc/InstrumentationOptions.html>  
<https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html>  
<https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html>
- Clang 的覆盖率：  
<https://clang.llvm.org/docs/SourceBasedCodeCoverage.html>
- 命令行工具的 LCOV 文档：  
<http://ltp.sourceforge.net/coverage/lcov/lcov.1.php>  
<http://ltp.sourceforge.net/coverage/lcov/genhtml.1.php>
- GCOV 更新的功能：  
<https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html#Invoking-Gcov>

# 第9章 分析工具

即使对于非常有经验的开发人员来说，生成高质量的代码也不是一件容易的事。通过向解决方案添加测试，降低了在代码中犯错误的风险，但这还不足以避免更复杂的问题。每一个软件都包含如此多的细节，以至于跟踪它们会成为了一项复杂的工作，甚至会和维护产品的团队达成几十种约定和多种特殊的设计实践。

有些问题与编码风格有关：应该在代码中使用 80 列，还是 120 列？是否能使用 std::bind 或 Lambda 函数？是否可以使用 C 风格的数组？小函数应该在单行中定义吗？应该始终坚持使用 auto，还是只在增加可读性时才使用？

理想情况下，还可以避免一般情况下不正确的语句：无限循环、使用标准库保留的标识符、无意的精度损失、冗余的 if 语句，以及其他“非最佳实践”（参见扩展阅读部分的参考资料）。

另一件事情是代码的现代化：随着 C++ 的发展，提供了新的特性。跟踪并重构代码升级到最新标准也很困难。此外，手动工作会耗费太多时间，并可能引入错误，这对于大型代码库来说风险相当大。

最后，应该检查启动时是如何工作的：执行程序并检查其内存。使用后是否正确释放内存？是否正确地访问了初始化的数据？或者代码是否会解引用悬空指针？

手动管理所有这些问题低效且容易出错，可以使用程序来检查和执行规则，修复错误，并更新代码。是时候使用工具分析程序，代码将在每次构建时进行检查，以确保它符合标准。

本章中，我们将讨论以下主题：

- 格式化
- 静态检查
- Valgrind 的动态分析

## 9.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter09>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 <build tree> 和 <source tree>。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 9.2. 格式化

专业开发人员通常遵循规则，高级开发人员知道什么时候打破规则（他们可以证明需要这样做）。另一方面，据说非常资深的开发人员不会违反规则，因为不断向他人解释自己就是浪费时间。我的看法是，选择你认可的方式即可，专注于真正重要的事情，对产品有实际影响的事情。

当谈到编码风格和格式时，开发者面临着无数的选择：我们应该使用制表符。还是空格来缩进？若是空格，有多少？一列的字符限制是多少？在文件中呢？大多数情况下，这样的选择不会影响程序的行为，但确实会产生很多杂声，并开始冗长的讨论，而这些讨论不会给产品带来太多价值。

有些做法是公认的，但大多数时候是在争论个人偏好。毕竟，在超过 120 的列中强制执行 80 个字符是一种随意的选择。选择什么并不重要，因为它会影响软件代码的可读性，所以风格一致就好。

避免这种情况的最好方法是使用格式化工具，如 clang-format。若代码的格式不正确我们会得到提醒，甚至可以进行修复。

下面是一个格式化代码的命令示例：

```
clang-format -i --style=LLVM filename1.cpp filename2.cpp
```

-i 选项告诉 clang-format 就地编辑文件。--style 选择应该使用的支持的格式化样式:LLVM, Google, Chromium, Mozilla, WebKit, 或自定义(以文件方式提供)(扩展阅读部分有详细的链接)。

当然，我们不希望每次进行更改时都手动执行此命令，CMake 应该将此作为构建过程的一部分来处理。我们已经知道如何在系统中找到 clang-format(需要手动安装)。

还没有讨论如何将外部工具应用到所有源文件的过程。为此，我们会创建一个函数，可以放在 cmake 目录中：

```
chapter09/01-formatting/cmake/Format.cmake

function(Format target directory)
 find_program(CLANG_FORMAT_PATH clang-format REQUIRED)
 set(EXPRESSION h_hpp hh_c cc_cxx cpp)
 list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")
 file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
 LIST_DIRECTORIES false ${EXPRESSION})
)
add_custom_command(TARGET ${target} PRE_BUILD COMMAND
 ${CLANG_FORMAT_PATH} -i --style=file ${SOURCE_FILES}
)
endfunction()
```

Format 函数接受两个参数:target 和 directory，在目标构建之前，其将格式化目录中的所有源文件。

从技术上讲，并非目录中的所有文件都必须属于目标(而且目标源可能位于多个目录中)。然而，查找属于目标(以及可能依赖的目标)的所有源文件和头文件是一个复杂的过程，特别是需要过滤掉外部库，且是不应该格式化的头文件。这种情况下，处理目录更容易管理，所以可以为格式化目标的每个目录调用函数。

这个函数有以下步骤：

1. 找到系统中安装的 clang-format 二进制文件。若没有找到二进制文件，REQUIRED 关键字将

停止配置并报错。

2. 创建要格式化的文件扩展名列表(用作 globbing 表达式)。
3. 每个表达式前面加上到目录的路径。
4. 递归地搜索源和头文件(使用前面创建的列表),跳过目录,并将其路径放入 SOURCE\_FILES 变量中。
5. 将格式化命令作为目标的 PRE\_BUILD 步骤。

该命令将很好地用于中小型代码库。对于大量的文件,需要将绝对文件路径转换为相对路径,并使用目录作为工作目录执行格式化(list(transform)在这里很有用)。这可能是必要的,因为传递给 shell 的命令有长度限制(通常约为 13000 个字符),太长的路径根本不适合。

来看看如何在实践中使用这个函数,将使用以下项目结构:

```
- CMakeLists.txt
- .clang-format
- cmake
 |- Format.cmake
- src
 |- CMakeLists.txt
 |- header.h
 |- main.cpp
```

首先,需要设置项目,并将 cmake 目录添加到模块路径中:

```
chapter09/01-formatting/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Formatting CXX)
enable_testing()
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
add_subdirectory(src bin)
```

设置好之后,为 src 目录完成列表文件:

```
chapter09/01-formatting/src/CMakeLists.txt

add_executable(main main.cpp)
include(Format)
Format(main .)
```

我们已经创建了一个可执行的目标主文件,包括 Format.cmake 模块,并为当前目录(src)中的主目标使用 Format() 函数。

现在,需要一些未格式化的源文件。头文件中只是一个简单的函数:

```
1 // chapter09/01-formatting/src/header.h
2
3 int unused() { return 2 + 2; }
```

我们还将添加一个有很多空格的源文件:

```
1 // chapter09/01-formatting/src/main.cpp
2
3 #include <iostream>
4 using namespace std;
5 int main() {
6
7 cout << "Hello, world!" << endl;
8 }
```

差不多准备好了。剩下的就是格式化器的配置文件(通过命令行的-style=file参数启用):

```
chapter09/01-formatting/.clang-format

BasedOnStyle: Google
ColumnLimit: 140
UseTab: Never
AllowShortLoopsOnASingleLine: false
AllowShortFunctionsOnASingleLine: false
AllowShortIfStatementsOnASingleLine: false
```

Clang Format 将扫描父目录以找到.clang-format 文件, 该文件指定了确切的格式化规则。这允许指定每个小细节, 或者定制前面提到的标准。这个例子中, 选择从 Google 的编码风格开始, 并进行了一些调整: 将列限制为 140 个字符, 删除制表符, 并允许短循环、函数和 if 语句。

来看看在构建这个项目之后文件是什么样的(编译前会自动格式化):

```
1 // chapter09/01-formatting/src/header.h (formatted)
2
3 int unused() {
4 return 2 + 2;
5 }
```

头文件格式化了, 即使没有目标使用, 短函数也不允许在一行中出现。正如预期的那样, 格式化程序添加了新行。main.cpp 文件现在看起来也很舒服:

```
1 // chapter09/01-formatting/src/main.cpp (formatted)
2
3 #include <iostream>
4 using namespace std;
5 int main() {
6 cout << "Hello, world!" << endl;
7 }
```

删除不必要的空白, 标准化缩进。

添加自动格式化器并不需要花费太多精力, 而且会为代码检查节省大量时间。若曾经手动修改过一些空白, 就会了解这种感觉, 一致的格式会使代码更简洁。

### Note

将格式化应用于现有的代码库，很可能会对存储库中的大多数文件引入很大的一次性更改。若(或您的团队成员)有一些正在进行的工作，这可能会导致许多合并冲突。最好是在完成所有挂起的修改之后协调这些工作。若不能这样做，则考虑逐步进行，也许是在每个目录的基础上。您的开发同事会感谢您的。

格式化器是一个伟大而简单的工具，可以将代码的可视化方面整合在一起，但不是一个成熟的程序分析工具(主要关注空白)。要处理更高级的场景，需要对编译程序的源码进行代码静态分析。

## 9.3. 静态检查

静态程序分析是在不运行编译版本的情况下检查源代码的过程。静态检查器的严格应用极大地提高了代码的质量，更加一致，更不容易出现错误。引入已知安全漏洞的机会也减少了。C++ 社区已经创建了几十个静态检查器:Astrée、Clang-Tidy、clay、CMetrics、Cppcheck、Cplint、CQMetrics、ESBMC、FlawFinder、Flint、IKOS、Joern、PC-Lint、Scan-Build、Vera++ 等。

其中的许多都承认 CMake 是行业标准，并将提供开箱即用的支持(或集成教程)。一些构建工程师不想去写 CMake 代码，可以通过在线可用的外部模块来添加静态检查器，比如 Lars Bilke 在他的 GitHub 仓库:<https://github.com/bilke/cmake-modules> 中收集的那些模块。

这并不奇怪，普遍的误解是需要通过许多步骤来检查代码。这种复杂性的原因在于静态检查器：经常模仿真正编译器的行为，来理解代码中发生的事情。

Cppcheck 在其手册中推荐以下步骤：

1. 找到静态检查器的可执行文件。
2. 生成一个编译数据库，包含以下内容：

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
```

3. 生成的 JSON 文件上运行检查器：

```
<path-to-ccpcheck> --project=compile_commands.json
```

这些都应该作为构建的一部分进行，这样就不会遗忘。

既然 CMake 完全理解如何构建目标，就不能支持其中一些工具吗？至少是最受欢迎的？当然可以！尽管使用起来很简单，但这个功能在网上很难找到。CMake 支持在每个目标的基础上为以下工具启用检查器：

- include-what-you-use (<https://include-what-you-use.org>)
- Clang-Tidy (<https://clang.llvm.org/extrac clang-tidy>)
- link what you use (一个 CMake 内置检查器)
- cpplint (<https://github.com/cpplint/cpplint>)

- Cppchecker (<https://cppcheck.sourceforge.io>)

需要做的是设置一个适当的目标属性为一个分号分隔的列表，其中包含到检查器可执行文件的路径，后面跟着应该转发给检查器的命令行选项：

- <LANG>\_CLANG\_TIDY
- <LANG>\_CPPCHECK
- <LANG>\_CPPLINT
- <LANG>\_INCLUDE\_WHAT\_YOU\_USE
- LINK\_WHAT\_YOU\_USE

<LANG> 需要替换为所使用的语言，因此对于 C 源代码使用 C，对于 C++ 使用 CXX。若不需要在每个目标的基础上控制检查器，可以为项目中的所有目标指定一个默认值，通过设置适当的全局变量前缀 CMAKE\_：

```
set(CMAKE_CXX_CLANG_TIDY /usr/bin/clang-tidy-3.9;-checks=*)
```

之后定义的目标，都将以同样的方式设置其 CXX\_CLANG\_TIDY 属性，这将把分析过程添加到常规构建中。

另外，对检查器应该如何测试目标进行更细粒度的控制也是有价值的。可以写一个简单的函数来解决这个问题：

```
chapter09/02-clang-tidy/cmake/ClangTidy.cmake

function(AddClangTidy target)
 find_program(CLANG-TIDY_PATH clang-tidy REQUIRED)
 set_target_properties(${target}
 PROPERTIES CXX_CLANG_TIDY
 "${CLANG-TIDY_PATH};-checks=*;--warnings-as-errors=*")
endfunction()
```

AddClangTidy 函数有两个简单的步骤：

1. 找到 Clang-Tidy 二进制文件，并将其路径存储在 CLANG-TIDY\_PATH 中。若没有找到二进制文件，REQUIRED 关键字将停止配置并报错。
2. 对目标启用 Clang-Tidy，提供到二进制文件和自定义选项的路径，以启用所有检查，并将警告视为错误。

要使用这个函数，只需要包含模块并为所选目标调用即可：

```
chapter09/02-clang-tidy/src/CMakeLists.txt

add_library(sut STATIC calc.cpp run.cpp)
target_include_directories(sut PUBLIC .)
```

```
add_executable(bootstrap bootstrap.cpp)
target_link_libraries(bootstrap PRIVATE sut)

include(ClangTidy)
AddClangTidy(sut)
```

当构建解决方案时，可以看到 Clang-Tidy 的输出：

```
[6%] Building CXX object bin/CMakeFiles/sut.dir/calc.cpp.o
/root/examples/chapter09/04-clang-tidy/src/calc.cpp:3:11:
warning: method 'Sum' can be made static [readability-convertmember-f
int Calc::Sum(int a, int b) {
^

[12%] Building CXX object bin/CMakeFiles/sut.dir/run.cpp.o
/root/examples/chapter09/04-clang-tidy/src/run.cpp:1:1:
warning: #includes are not sorted properly [llvm/include-order]
#include <iostream>
^ ~~~~~
/root/examples/chapter09/04-clang-tidy/src/run.cpp:3:1:
warning: do not use namespace using-directives; use using declarations
using namespace std;
^

/root/examples/chapter09/04-clang-tidy/src/run.cpp:6:3:
warning: initializing non-owner 'Calc *' with a newly created
'gsl::owner<>' [cppcoreguidelines-owning-memory]
auto c = new Calc();
^
```

注意，除非将`--warnings-as-errors=*`选项添加到命令行参数中，否则构建将会成功。建议将执行的规则和破坏这些规则的构建失败一视同仁；通过这种方式，就可以防止不兼容的代码污染存储库。

Clang-Tidy 还提供了一个有趣的`--fix` 选项，可以自动纠正代码。这绝对是一个节省时间的方法，可以在需要多次检查时使用。与格式化一样，在将静态分析工具生成的更改引入到历史代码库时，请确保避免合并冲突。

根据用例、存储库的大小和团队偏好，可能应该选择一些匹配的检查器，添加太多也会成为麻烦。下面是 CMake 支持检查器的简介。

### 9.3.1 Clang-Tidy

以下是官方网站对 Clang-Tidy 的描述：

Clang-Tidy 是一个基于 clang 的 C++ 的“linter”工具。目的是提供一个可扩展的框架，用于诊

断和修复常见的编程错误，如样式违反、接口滥用或通过静态分析可以推断出的错误。Clang-Tidy 是模块化的，为编写新检查提供了方便的接口。

这个工具的多功能性真的令人印象深刻，提供了超过 400 个检查。与 ClangFormat 配合使用效果很好，因为自动应用的补丁（超过 150 个可用补丁）可以遵循相同的格式文件。提供的检查包括性能、可读性、现代化、`cpp-core-guidelines` 和改进容易出现 bug 的命名空间。

### 9.3.2 Cpplint

以下是官方网站对 Cpplint 的描述：

Cpplint 是一个命令行工具，根据谷歌的 C++ 样式指南检查 C/C++ 文件的样式问题。Cpplint 由 Google 公司在 [Google/styleguide](#) 上开发和维护。

这个 linter 是为了使代码与前面提到的 Google 编码风格保持一致。它用 Python 编写的，这可能是某些项目不需要的依赖项。这些修复以 Emacs、Eclipse、VS7、Junit 可使用的格式提供，也可以作为 sed 命令提供。

### 9.3.3 Cppcheck

以下是官方网站对 Cppcheck 的描述：

Cppcheck 是 C/C++ 代码的静态分析工具，提供了独特的代码分析来检测错误，并专注于检测未定义的行为和危险的编码结构，目标是尽量少出现假阳性。Cppcheck 旨在分析 C/C++ 代码，可以检测非标准语法（在嵌入式项目中常见）。

当涉及到避免误报所产生的不必要的杂音时，这个工具值得推荐。其已经建立得很好（超过 14 年的制作过程），而且仍然非常积极地维护着。另外，若代码不使用 Clang 编译，会发现它很有用。

### 9.3.4 include-what-you-use

以下是官方网站对 include-what-you-use 的描述：

include-what-you-use 的主要目标是删除多余的 `#include`，通过找出这个文件（对于.cc 和.h 文件）实际上不需要的 `#include` 来实现，并在可能的情况下用前置声明替换 `#include`。

若代码库很小，那么包含太多的头文件似乎不是一个大问题。在较大的项目中，避免不必要的头文件编译可以节省很多的时间。

### 9.3.5 link-what-you-use

以下是 CMake 博客上对 link-what-you-use 的描述：

这是 CMake 中内置的特性，使用 `ld` 和 `ldd` 选项来打印可执行程序链接的库比实际需要的多。

这也加快了构建时间。只是在这种情况下，关注的是不需要的二进制文件。

当软件错误可能影响人们的安全时，静态分析就至关重要，特别是在医疗、核能、航空、汽车和机械行业。聪明的开发人员知道，在要求较低的环境中遵循类似的实践没有坏处，尤其是在成本很低的情况下。构建过程中使用静态分析器，不仅比手动查找和修复 bug 方便得多，也可以使用 CMake 启用。对于质量敏感的软件（除了开发者之外还有其他人参与的所有软件）中，几乎没有理由跳过这些检查。

不幸的是，并不是所有的错误都能在程序执行之前捕获。做些什么可以让检查器更好地了解我们的项目呢？

## 9.4. Valgrind 的动态分析

Valgrind (<https://www.valgrind.org>) 是一个工具框架，允许构建动态分析实用程序——在运行时进行分析。提供了很多工具套件，允许进行各种调查和检查。其中一些工具如下：

- Memcheck - 检测内存管理问题
- Cachegrind - 配置 CPU 缓存，并指出缓存缺失和其他缓存问题
- Callgrind - Cachegrind 的扩展，提供了调用图的信息
- Massif - 堆分析器，显示程序的哪些部分在特定时段内使用堆
- Helgrind - 线程调试器，帮助解决数据竞争问题
- DRD - 更轻的限量版 Helgrind

列表中的每一个工具都非常方便。大多数包管理器都知道 Valgrind，并且可以轻松地将它安装到操作系统上（若使用的是 Linux，可能已经安装了它）。并且，官方网站提供了源代码，也可以自己构建。

我们将把重点限制在套件中最有用的应用程序上。当提到 Valgrind 时，通常指的是 Valgrind 的 Memcheck。让我们了解一下如何在 CMake 中使用它——这将为使用其他工具铺平道路。

### 9.4.1 Memcheck

调试内存问题时，Memcheck 不可缺少。开发者对如何管理内存有很大的控制权，所以这个主题在 C++ 中特别棘手。各种各样的错误都有可能发生：读取未分配的内存、读取已经释放的内存、多次尝试释放内存，以及写入不正确的地址。开发人员显然试图避免这些错误，但由于这些错误不显眼，甚至可以潜入最简单的程序。有时，只要一个遗忘的变量初始化，就会我们让陷入困境。

使用 Memcheck 的方式是这样的：

```
valgrind [valgrind-options] tested-binary [binary-options]
```

Memcheck 是 Valgrind 的默认工具，也可以显式选择它：

```
valgrind --tool=memcheck tested-binary
```

运行 Memcheck 的开销很大。手册（参见扩展阅读中的链接）里说，它可使程序慢 10-15 倍。为了避免每次运行测试时都等待 Valgrind，可以创建一个单独的目标，在需要测试代码时从命令行调用该目标。理想情况下，开发人员将在将更改合并到存储库的默认分支之前运行。这可以通过 Git 钩子完成，也可以作为 CI 流水中的一个步骤。要构建一个自定义目标，需要在生成阶段完成后使用以下命令：

```
cmake --build <build-tree> -t valgrind
```

添加这样一个目标并不难:

```
chapter09/03-valgrind/cmake/Valgrind.cmake

function(AddValgrind target)
 find_program(VALGRIND_PATH valgrind REQUIRED)
 add_custom_target(valgrind
 COMMAND ${VALGRIND_PATH} --leak-check=yes
 ${<TARGET_FILE:${target}>}
 WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
)
endfunction()
```

本例中，创建了一个 CMake 模块 (因此我们可以跨项目重用相同的文件) 包装函数，将接受要测试的目标。这里会发生两件事：

- CMake 搜索 valgrind 可执行文件的默认系统路径，并将其存储在 VALGRIND\_PATH 变量中。若没有找到二进制文件，REQUIRED 关键字将停止配置并报错。
- 创建了一个自定义目标 valgrind，将在目标二进制文件上执行 Memcheck 工具，这里还添加了一个检查内存泄漏的选项。

当涉及到 Valgrind 选项时，可以作为命令行参数提供，也可以使用以下方式提供：

1. ./valgrindrc 文件 (在主目录中)
2. \$VALGRIND\_OPTS 环境变量
3. ./Valgrindrc 文件 (在工作目录中)

这些是按顺序检查的。另外，最后一个文件只有在属于当前用户、是常规文件且没有标记为全球可写的情况下才会考虑。因为给 Valgrind 的选项可能是潜在有害，所以这是一种安全的机制。

要使用 AddValgrind 函数，应该为它提供一个 unit\_tests 目标：

```
chapter09/03-valgrind/test/CMakeLists.txt (fragment)

...
add_executable(unit_tests calc_test.cpp run_test.cpp)
...
include(Valgrind)
AddValgrind(unit_tests)
```

使用 Debug 配置生成构建树允许 Valgrind 进入调试信息，这使它的输出更加清晰。来看看在实践中是如何工作的：

```
cmake --build <build-tree> -t valgrind
```

这将构建 sut 和 unit\_tests 目标:

```
[100%] Built target unit_tests
```

开始执行 Memcheck, 其为提供了基本信息:

```
==954== Memcheck, a memory error detector
==954== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==954== Using Valgrind-3.15.0 and LibVEX; rerun with -h for
copyright info
==954== Command: ./unit_tests
```

前缀 ==954== 包含进程 ID。添加这一点是为了将 Valgrind 注释与测试过程的输出区分开来。

接下来, 使用 gtest 运行测试:

```
[=====] Running 3 tests from 2 test suites.
[-----] Global test environment set-up.
...
[=====] 3 tests from 2 test suites ran. (42 ms total)
[PASSED] 3 tests.
```

最后, 进行总结:

```
==954==
==954== HEAP SUMMARY:
==954== in use at exit: 1 bytes in 1 blocks
==954== total heap usage: 209 allocs, 208 frees, 115,555
bytes allocated
```

啊哦! 我们仍然使用至少 1 个字节。使用 malloc() 和 new 进行的分配与适当的 free() 和 delete 操作不匹配, 程序似乎有内存泄漏。Valgrind 提供了更多的细节来找到它:

```
==954== 1 bytes in 1 blocks are definitely lost in loss record
1 of 1
==954== at 0x483BE63: operator new(unsigned long) (in /usr/
lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.
so)
==954== by 0x114FC5: run() (run.cpp:6)
==954== by 0x1142B9: RunTest_RunOutputsCorrectEquations_
Test::TestBody() (run_test.cpp:14)
```

以 by 0x<address> 开头的行表示调用堆栈中的各个函数。这里截断了输出 (有一些输出来自 GTest)，以关注有趣的部分——最顶层的函数和源引用，run()(run.cpp:6): 最后，在底部找到总结信息：

```
==954== LEAK SUMMARY:
==954== definitely lost: 1 bytes in 1 blocks
==954== indirectly lost: 0 bytes in 0 blocks
==954== possibly lost: 0 bytes in 0 blocks
==954== still reachable: 0 bytes in 0 blocks
==954== suppressed: 0 bytes in 0 blocks
==954==
==954== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
from 0)
```

Valgrind 在发现非常复杂的问题方面做得很好，还能够更深入地挖掘，找到无法自动分类的可疑情况。这些发现将归入可能丢失的那一类。

来看看 Memcheck 在这个案例中发现了什么问题：

```
1 // chapter09/03-valgrind/src/run.cpp
2
3 #include <iostream>
4 #include "calc.h"
5 using namespace std;
6
7 int run() {
8 auto c = new Calc(); // ERROR
9 cout << "2 + 2 = " << c->Sum(2, 2) << endl;
10 cout << "3 * 3 = " << c->Multiply(3, 3) << endl;
11 return 0;
12 }
```

没错：突出显示的代码是错误的。事实上，我们确实创建了一个在测试结束之前没有删除的对象。这就是为什么拥有广泛的测试覆盖率是如此重要的确切原因。

Valgrind 是一个非常有用的工具，但是在处理更复杂的程序时，其输出会变得有点冗长。必须有一种方法，以更易于管理的形式收集这些信息。

#### 9.4.2 Memcheck-Cover

商业 IDE(如 CLion) 本身支持将 Valgrind 的输出解析为可以在 GUI 中轻松导航的内容，而无需通过控制台窗口滚动来查找正确的消息。若编辑器没有这个选项，可以通过使用第三方报告生成器更清楚地查看错误。David Garcin 编写的 Memcheck-cover 以生成 HTML 文件的形式提供了更好的体验，如图 9.1 所示：

**Valgrind's memcheck report**

---

**Result summary:**  
Pass: 0 / 1      Errors: 2

**Title:** • Command • Analysis name • Theme: • Light • Dark-grey • Dark-blue

≡ [ **ERROR** ] Command: unit\_tests

```
==1508== Memcheck, a memory error detector
==1508== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1508== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1508== Command: /tmp/b/test/unit_tests
==1508== Parent PID: 1492
==1508==
==1508==
==1508== HEAP SUMMARY:
==1508== in use at exit: 1 bytes in 1 blocks
==1508== total heap usage: 209 allocs, 208 frees, 115,555 bytes allocated
==1508==
==1508== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1508== at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1508== by 0x114FC5: run() (run.cpp:6)
==1508== by 0x1142B9: RunTest_RunOutputsCorrectEquations_Test::TestBody() (run_test.cpp:14)
==1508== by 0x151883: void testing::HandleSehExceptionsInMethodIfSupported(testing::Test::*, char const*) (gtest.cc:2607)
==1508== by 0x1496C2: void testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test::*, void (testing::Test::*, char const*) (gtest.cc:2607))
==1508== by 0x114BDF: testing::Test::Run() (gtest.cc:2602)
==1508== by 0x11EB21: testing::TestInfo::Run() (gtest.cc:961)
==1508== by 0x11F41B: testing::TestSuite::Run() (gtest.cc:3015)
==1508== by 0x12EDB0: testing::internal::UnitTestImpl::RunAllTests() (gtest.cc:5855)
==1508== by 0x152DB1: bool testing::internal::HandleSehExceptionsInMethodIfSupported(testing::internal::UnitTestImpl::*, bool (testing::internal::UnitTestImpl::*)(), char const*) (gtest.cc:2607)
==1508== by 0x14A900: bool testing::internal::HandleExceptionsInMethodIfSupported<testing::internal::UnitTestImpl, bool>(testing::internal::UnitTestImpl::*, bool (testing::internal::UnitTestImpl::*)(), char const*) (gtest.cc:2643)
==1508== by 0x12D515: testing::UnitTest::Run() (gtest.cc:5438)
==1508== by 0x115192: RUN_ALL_TESTS() (gtest.h:2490)
==1508== by 0x115114: main (gtest_main.cc:52)
==1508==
==1508== LEAK SUMMARY:
==1508== definitely lost: 1 bytes in 1 blocks
==1508== indirectly lost: 0 bytes in 0 blocks
==1508== possibly lost: 0 bytes in 0 blocks
==1508== still reachable: 0 bytes in 0 blocks
==1508== suppressed: 0 bytes in 0 blocks
==1508==
==1508== For lists of detected and suppressed errors, rerun with: -s
==1508== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

---

This report was generated using [memcheck-cover](#) v1.2

图 9.1 由 memcheck-cover 生成的报告

这个整洁的小项目可以在 GitHub(<https://github.com/Farigh/memcheck-cover>) 上找到，需要 Valgrind 和 gawk (GNU AWK 工具)。要使用它，需要在单独的 CMake 模块中准备一个 setup 函数。其由两部分组成：

- 获取和配置工具
- 添加执行 Valgrind 并生成报告的自定义目标

配置如下所示：

```
chapter09/04-memcheck/cmake/Memcheck.cmake

function(AddMemcheck target)
 include(FetchContent)
 FetchContent_Declare(
```

```

memcheck-cover
 GIT_REPOSITORY https://github.com/Farigh/memcheckcover.git
 GIT_TAG release-1.2
)
FetchContent_MakeAvailable(memcheck-cover)
set(MEMCHECK_PATH ${memcheck-cover_SOURCE_DIR}/bin)

```

第一部分中，我们遵循与常规依赖项相同的实践：包括 FetchContent 模块，并使用 FetchContent\_Declare 指定项目的存储库和所需的 Git 标记。接下来，启动获取过程，并使用 FetchContent\_Populate(由 FetchContent\_MakeAvailable 隐式调用) 设置的 memcheck-cover\_SOURCE\_DIR 变量配置到二进制文件的路径。

该函数的第二部分是创建生成报告的目标，将其命名为 memcheck(这样就不会与之前的 valgrind 目标重叠)：

```

chapter09/04-memcheck/cmake/Memcheck.cmake (continued)

add_custom_target(memcheck
 COMMAND ${MEMCHECK_PATH}/memcheck_runner.sh -o
 "${CMAKE_BINARY_DIR}/valgrind/report"
 -- ${TARGET_FILE}:${target}
 COMMAND ${MEMCHECK_PATH}/generate_html_report.sh
 -i "${CMAKE_BINARY_DIR}/valgrind"
 -o "${CMAKE_BINARY_DIR}/valgrind"
 WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
)
endfunction()

```

这发生在两个命令中：

- 首先，运行 memcheck\_runner.sh 包装器脚本，该脚本将执行 Valgrind 的 memcheck 并将输出收集到带有-o 参数的文件中。
- 然后，将解析输出并使用 generate\_html\_report.sh 创建报告。这个脚本需要使用-i 和-o 参数提供的输入和输出目录。

两个步骤都应该在 CMAKE\_BINARY\_DIR 工作目录中执行，以便单元测试二进制在需要时通过相对路径访问文件。

需要添加到列表文件的最后一件事当然是对这个函数的调用，与 AddValgrind 有相同的模式：

```

chapter09/04-memcheck/test/CMakeLists.txt (fragment)

include(Memcheck)
AddMemcheck(unit_tests)

```

用 Debug 配置生成一个构建系统之后，可以用以下方法构建目标：

```
cmake --build <build-tree> -t memcheck
```

然后，就可以享受格式化的报告了。为了真正享受它，需要在 `run.cpp` 中添加缺失的 `delete c;`，使它停止抱怨（或者，使用智能指针）。

## 9.5. 总结

“你将花更多的时间研究代码而不是创建代码——因此，应该优化阅读，而不是仅仅去编写代码。”

这句话出现在不止一本讨论干净代码实践的书中，像咒语一样反复出现。这也不奇怪，因为这是非常正确的，正如许多软件开发人员在实践中所测试的那样——以至于对于诸如空格的数量、换行符和 `#import` 语句的顺序等非常微小的事情的规则都已经成了规范。这样做不是因为小气，而是为了节省成本。通过遵循本章概述的实践，不需要担心手工正确格式化代码。作为构建的副作用，将自动格式化——我们无论如何都必须执行的步骤，以检查代码是否正常工作。通过引入 `ClangFormat`，还可以确保其看起来正确。

当然，我们想要的不仅仅是简单的空格校正，代码必须符合几十个其他的小规则。这是通过添加 `Clang-Tidy` 并配置来实现的，以适配所选择的编码风格。详细讨论了这个静态检查器，也提到了其他选项：`Cpplint`、`Cppcheck`、`Include-what-you-use` 和 `Link-what-you-use`。由于静态链接器相对较快，可以用很少精力将它们添加到构建中，而且这通常是物超所值的。

最后，研究了 `Valgrind` 实用程序，特别是 `Memcheck`，其允许调试与内存管理相关的问题：错误的读取、写入、释放等。这是一个非常方便的工具，可以节省数小时的手工检查，并防止错误潜入生产环境。不过，其执行速度可能有点慢，这就是为什么需要创建了单独的目标，以便在提交代码之前显式地运行它。还学习了如何使用 `Memcheck-Cover`（HTML 报告生成器），以更友好的形式表示 `Valgrind` 的输出。这在不支持运行 IDE 的环境中非常有用（例如 CI 流水）。

当然，我们并不局限于这些工具，还有很多免费和开源项目，以及相应的商业产品。这里，只是对这门学科的介绍而已，一定要探索什么是适合你自己的。下一章中，将会更深入地研究如何生成文档。

## 9.6. 扩展阅读

更多资料，请访问以下链接：

- C++ 核心指南由 Bjarne Stroustrup（C++ 的作者）编撰：  
<https://github.com/isocpp/CppCoreGuidelines>
- ClangFormat 的参考：  
<https://clang.llvm.org/docs/ClangFormat.html>
- C++ 的静态分析器——精选的列表：  
<https://github.com/analysistools-dev/static-analysis#cpp>
- CMake 中内置的静态检查器支持：

<https://blog.kitware.com/static-checks-with-cmake-cdash-iwyu-clang-tidy-lwyucplint-and-cppcheck/>

- 启用 ClangTidy 的目标属性:

[https://cmake.org/cmake/help/latest/prop\\_tgt/LANG\\_CLANG\\_TIDY.html](https://cmake.org/cmake/help/latest/prop_tgt/LANG_CLANG_TIDY.html)

- Valgrind 的手册:

<https://www.valgrind.org/docs/manual/manual-core.html>

# 第 10 章 生成文档

高质量的代码不仅要写得很好、工作良好、测试良好，而且还要有完整的文档记录。文档允许我们共享可能丢失的信息，描绘更大的图景，提供上下文，揭示意图，并最终让外部用户和维护人员更容易理解代码或项目。

还记得上次加入的新项目么？在迷宫般的目录和文件中迷失了好几个小时吗？这本可以避免的。真正优秀的文档可以让新手在几秒钟内找到他们想要的代码行。遗憾的是，缺少文档的问题常常被掩盖起来。这并不奇怪，这需要很多技巧，许多人并不擅长制作文档。最重要的是，文档和代码工作是分离的，除非制定了严格的更新和审查流程，否则很容易忘记更新文档。

有些团队（为了节省时间或受到管理人员的鼓励）遵循编写“自文档化代码”的实践。通过为文件名、函数、变量等选择有意义的、可读的标识符，从而避免记录的繁琐工作。虽然好的命名习惯是绝对正确的，但其不能取代文档。即使是最好的函数签名也不能保证传递所有必要的信息——例如，`int removeduplicates ()` 是非常描述性的，但并没有揭示返回的是什么！可能是找到的副本数量，剩下的物品数量，或者其他东西——这并不确定。记住：天下没有免费的午餐。

为了简化工作，专业人员使用自动文档生成器，它可以分析源文件中的代码和注释，以多种不同格式生成全面的文档。在 CMake 项目中添加这样的生成器非常简单——来看看文档是怎么生成！

本章中，我们将讨论以下主题：

- 添加 Doxygen
- 生成好看的文档

## 10.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter10>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 10.2. 添加 Doxygen

可以从 C++ 源码生成文档的成熟和流行的工具是 Doxygen，其第一个版本是由 Dimitri van Heesch 在 1997 年 10 月发布的。从那时起，其存储库 (<https://github.com/doxygen/doxygen>) 就得到了 180 多个贡献者的积极支持。

Doxygen 可以生成以下格式的文档：

- 超文本标记语言 (HTML)

- 富文本格式 (RTF)
- 便携式文件格式 (PDF)
- Lamport's TeX (LaTeX)
- PostScript (PS)
- Unix 手册 (man pages)
- Microsoft 课编译 HTML 帮助 (CHM)

若用 Doxygen 指定的格式提供额外信息的注释装饰代码，其将解析为丰富的输出文件，还可以分析代码结构以生成有用的图表。后者是可选的，并且需要 Graphviz 工具 (<https://graphviz.org/>) 支持。

开发人员首先应该回答以下问题：项目的用户是直接获得文档，还是自己生成文档（也许是在从源码构建时）？第一个选项意味着文档是随二进制文件一起提供的，可以在线获得，或者（不那么优雅地）与源代码一起存入存储库。

答案很重要，因为若希望用户在构建过程中生成文档，他们将需要系统中存在的依赖项。这不是一个太大的问题，因为 Doxygen 可以通过大多数包管理器（以及 Graphviz）获得，所有需要的只是一个简单的命令，例如 Debian 的这个命令：

```
apt-get install doxygen graphviz
```

Windows 上也有可用的二进制文件（查看该项目的网站）。

为用户生成文档，或在需要时处理添加依赖项。这已经在第 7 章介绍过了，这里就不再赘述了。Doxygen 是用 CMake 构建的，也可以使用源码生成。

在系统中安装 Doxygen 和 Graphviz 后，可以将生成添加到项目中。与网上消息来源不同，这并不像我们想象的那么困难或复杂。不需要创建外部配置文件，提供到 doxygen 可执行文件的路径，或添加自定义目标。从 CMake 3.9 开始，可以使用 FindDoxygen 查找模块中的 doxygen\_add\_docs() 函数，设置文档目标。

其签名是这样的：

```
doxygen_add_docs(targetName [sourceFilesOrDirs...]
[ALL] [USE_STAMP_FILE] [WORKING_DIRECTORY dir]
[COMMENT comment])
```

第一个参数指定目标名称，需要用-t 参数显式地构建 cmake（在生成构建树之后）：

```
cmake --build <build-tree> -t targetName
```

或者，可以通过添加 ALL 参数来构建（通常没有必要）。其他选项都是不言自明的，除了 USE\_STAMP\_FILE。这允许 CMake 在没有任何源文件更改的情况下跳过文档的重新生成（但要求 sourceFilesOrDirs 只包含文件）。

我们将遵循前几章的实践，创建一个带有辅助函数的程序模块（可以在其他项目中重用）：

```
chapter-10/01-doxygen/cmake/Doxygen.cmake
```

```

function(Doxygen input output)
 find_package(Doxygen)
 if(NOT DOXYGEN_FOUND)
 add_custom_target(doxygen COMMAND false
 COMMENT "Doxygen not found")
 return()
 endif()
 set(DOXYGEN_GENERATE_HTML YES)
 set(DOXYGEN_HTML_OUTPUT
 ${PROJECT_BINARY_DIR}/${output})

 doxygen_add_docs(doxygen
 ${PROJECT_SOURCE_DIR}/${input}
 COMMENT "Generate HTML documentation"
)
endfunction()

```

该函数接受两个参数——输入和输出目录——并将创建一个自定义 doxygen 目标:

- 首先，使用 CMake 的内置 Doxygen 查找模块来确定系统中是否有 Doxygen 可用。
- 若不可用，将创建一个伪 doxygen 目标，通知用户并运行一个错误命令，该命令(在类 Unix 系统上)返回 1，导致构建失败。这时用 return() 结束函数。
- 若 Doxygen 可用，将在提供的输出目录中生成 HTML。Doxygen 是可配置的(在官方文档中找到更多信息)，要设置任何选项，只需按照示例调用 set() 并在其名称前加上 DOXYGEN\_。
- 设置实际的 doxygen 目标: 所有 DOXYGEN\_ 变量将转发到 doxygen 的配置文件中，文档将从源树中提供的输入目录生成。

若文档是由用户生成的，步骤 2 可能需要安装必要的依赖项。

要使用这个函数，可以将其添加到项目的主列表文件中:

```

chapter-10/01-doxygen/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Doxygen CXX)
enable_testing()
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
add_subdirectory(src bin)

include(Doxygen)
Doxygen(src docs)

```

构建 doxygen 目标生成如下 HTML 文档:

# Doxygen

Main Page   Classes ▾   Files ▾    Search   Public Member Functions | List of all members

## Calc Class Reference

```
#include <calc.h>
```

### Public Member Functions

|                                                  |
|--------------------------------------------------|
| <b>Calc</b> ( <i>RandomNumberGenerator</i> *rng) |
| int <b>Sum</b> (int a, int b)                    |
| int <b>Multiply</b> (int a, int b)               |
| int <b>AddRandomNumber</b> (int a)               |

### Detailed Description

This class does some simple calculations

### Member Function Documentation

◆ **AddRandomNumber()**

```
int Calc::AddRandomNumber (int a)
```

Adds randomly generated number to the parameter

◆ **Multiply()**

```
int Calc::Multiply (int a,
 int b
)
```

Multiply... Who would have thought?

**Parameters**

**a** the first factor  
**b** the second factor

**Returns**

The product

图 10.1 Doxygen 生成的类引用

成员函数文档中可以看到，通过在头文件中用适当的注释在方法前面添加：

```

1 // chapter-10/01-doxygen/src/calc.h (fragment)
2
3 /**
4 Multiply... Who would have thought?
5 @param a the first factor
6 @param b the second factor
7 @result The product
8 */
9 int Multiply(int a, int b);

```

这种格式称为 Javadoc。打开带有双星号的注释块很重要:/\*\*。更多信息可以在 Doxygen 的文档块描述中找到(参见扩展阅读部分的链接)。

若安装了 Graphviz, Doxygen 将检测它并生成依赖关系图:

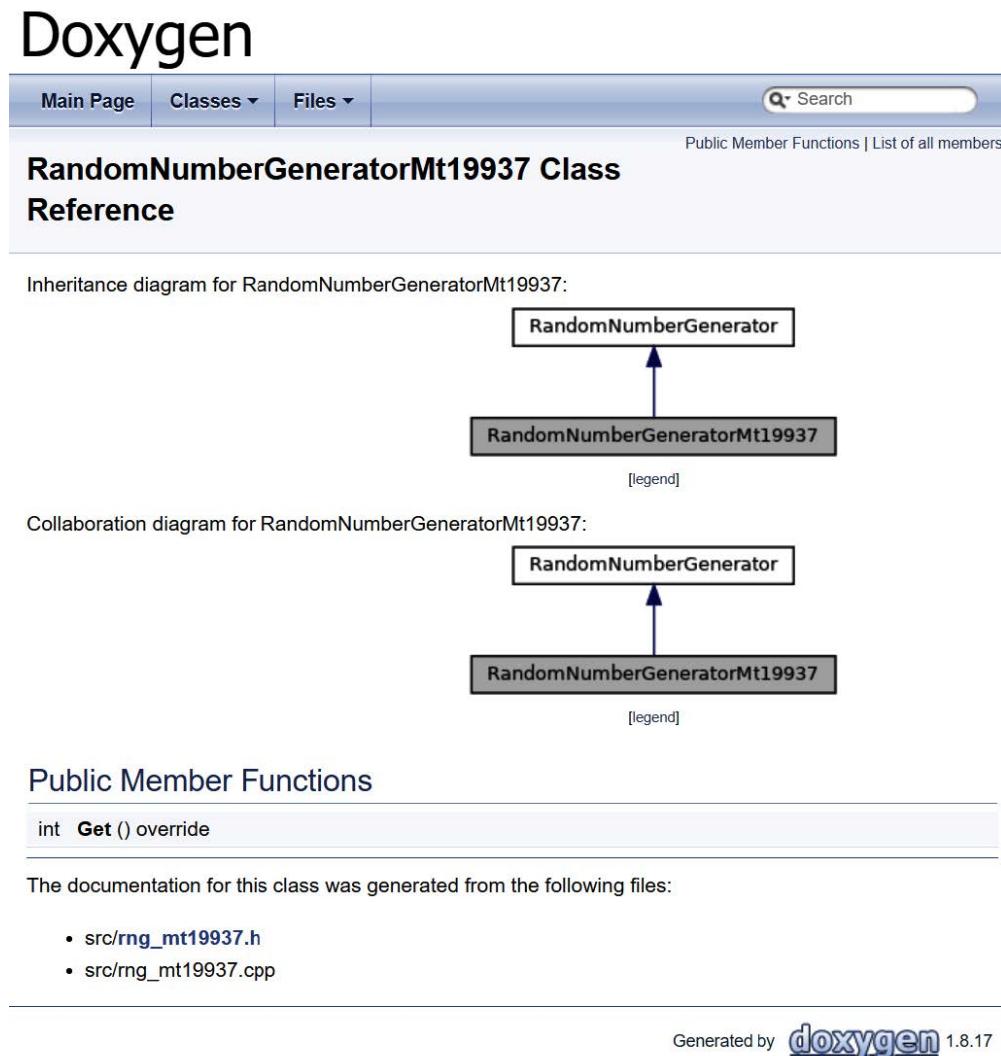


图 10.2 Doxygen 生成的继承和协作图

通过直接从源代码生成文档, 创建了一种机制, 可以使用整个开发周期中发生的更改快速更新文档。此外, 注释中的遗漏都有可能在代码审查期间发现。

许多开发者会抱怨 Doxygen 提供的设计过时了，这让他们在向客户展示生成的文档时犹豫不决。别担心，这个问题很容易解决。

## 10.3. 生成好看的文档

用干净、新颖的设计编写项目文档也很重要，若把所有的工作都投入到为前沿项目编写高质量的文档中，那么必须让用户这样看到它。Doxygen 拥有所有花哨的功能，但并不以紧跟最新的视觉时尚趋势而闻名。然而，这并不意味着需要付出很多努力才能改变这一点。

一个名为 jothepro 的开发人员创建了一个名为 doxygen-awesome-css 的主题，提供了一个现代的、可定制的设计，甚至有一个黑暗模式！可以在下面的截图中看到：

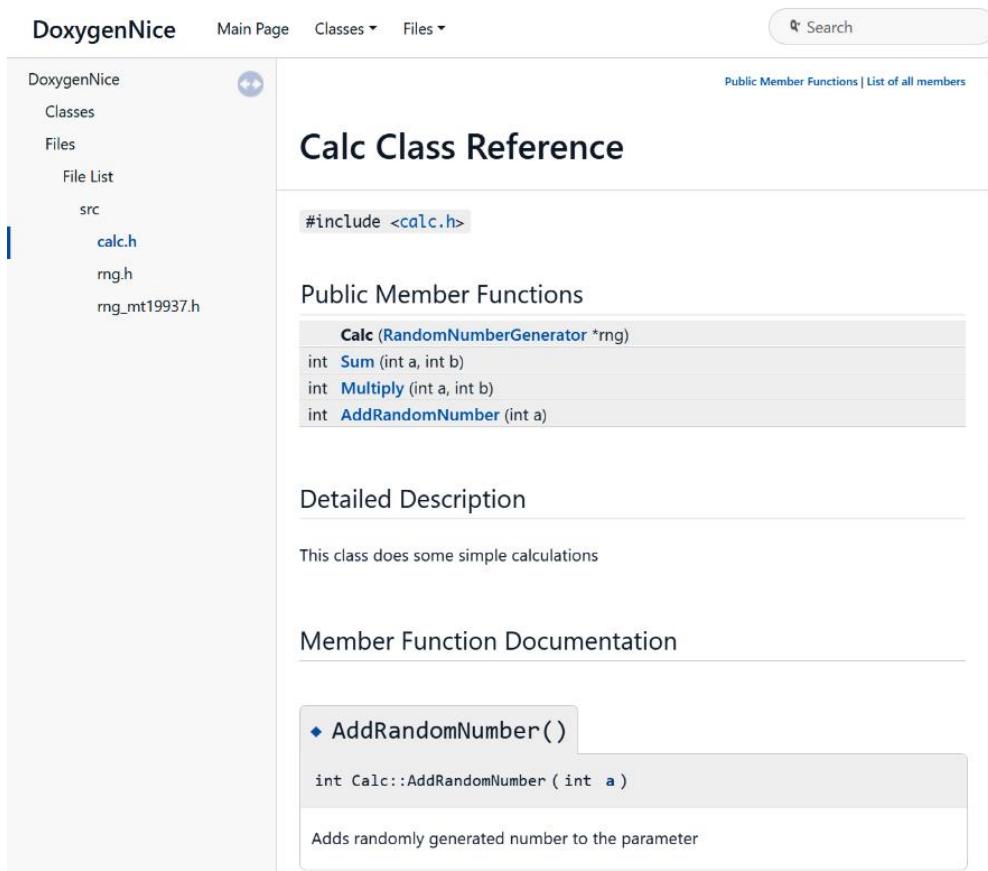


图 10.3 doxygen-awesome-css 主题中的 HTML 文档

该主题不需要任何额外的依赖项，可以很容易地从其 GitHub 页面 (<https://github.com/jothepro/doxygen-awesome-css>) 获取。

### Note

在线教程建议使用多个应用程序串行执行来升级体验，流行的方法是使用 Breathe 和 Exhale 扩展将 Doxygen 的输出转换为 Sphinx。这个过程看起来有点麻烦，会引入许多其他依赖项（比如 Python）。我建议尽可能保持工具的简单性，很有可能不是项目中的每个开发人员都能很好地理解 CMake，如此复杂的过程会给他们带来麻烦。

将这个主题的自动采用，来看看如何扩展我们的 Doxygen.cmake，通过添加一个新的宏来使用它：

```
chapter-10/02-doxygen-nice/cmake/Doxygen.cmake (fragment)

macro (UseDoxygenAwesomeCss)
 include(FetchContent)
 FetchContent_Declare(doxygen-awesome-css
 GIT_REPOSITORY
 https://github.com/jothepro/doxygen-awesome-css.git
 GIT_TAG
 v1.6.0
)
 FetchContent_MakeAvailable(doxygen-awesome-css)
 set(DOXYGEN_GENERATE_TREEVIEW YES)
 set(DOXYGEN_HAVE_DOT YES)
 set(DOXYGEN_DOT_IMAGE_FORMAT svg)
 set(DOXYGEN_DOT_TRANSPARENT YES)
 set(DOXYGEN_HTML_EXTRA_STYLESHEET
 ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome.css)
endmacro()
```

本书的前几章中，已经知道了所有这些命令，但为了更加清晰，来重申一下发生了什么：

- 将 doxy-awesome-css 使用 Git 下载下来，并通过 FetchContent 模块提供给项目。
- 按照主题的 README 文件的建议，配置了 Doxygen 的选项。
- DOXYGEN\_HTML\_EXTRA\_STYLESHEET 配置主题的.css 文件的路径，其将复制到输出目录。

最好在 Doxygen 函数中调用这个宏，就在 doxygen\_add\_docs() 之前：

```
chapter-10/02-doxygen-nice/cmake/Doxygen.cmake

function(Doxygen input output)
 ...
 UseDoxygenAwesomeCss()
 doxygen_add_docs (...)

endfunction()

macro (UseDoxygenAwesomeCss)
 ...
endmacro()
```

宏中的所有变量都在调用函数的作用域中设置。

现在可以在生成的 HTML 文档中享受现代风格，并自信地与世界进行分享。

## 10.4. 总结

这一简短的章节中，我们介绍了如何将文档生成工具 Doxygen 添加到 CMake 项目中，并使其变得优雅。这个过程不太复杂，将极大地改善解决方案中的信息流。花在添加文档上的时间将是一项值得的开销，特别是当发现您或您的团队成员在理解应用中的复杂关系方面有困难时。

可能会担心将 Doxygen 添加到一个从一开始就不使用文档生成的大型项目中会很困难，向每个函数添加注释所需的大量工作可能会让开发人员不堪重负。不要追求立即完成：从小处着手，只填写最近提交中接触到的元素的描述。即使是大部分不完整的文档也比完全没有文档要好。

通过生成文档，将强制它接近实际代码：若都在同一个文件中，那么维护书面解释与逻辑同步就更容易了。另外，与大多数程序员一样，您可能是一个非常忙碌的人，最终会忘记项目的一些小细节。记住：好记性不如烂笔头。帮自己一个忙，先把事情写下来。

下一章中，我们将学习如何使用 CMake 自动化项目的打包和安装。

## 10.5. 扩展阅读

- Doxygen 官网: <https://www.doxygen.nl/>
- FindDoxygen 查找模块文档: <https://cmake.org/cmake/help/latest/module/FindDoxygen.html>
- Doxygen 的文档: <https://www.doxygen.nl/manual/docblocks.html#specialblock>

### 10.5.1 其他文档生成工具

因为我们关注的是 CMake 支持的项目，所以还有许多其他的工具在本书中没有涉及。然而，其中一些可能更适合您。若喜欢探索，可以访问两个我觉得有趣的项目，将地址列在这里：

- Adobe 的 Hyde (<https://github.com/adobe/hyde>)  
针对 Clang 编译器，Hyde 生成了可以使用 Jekyll (<https://jekyllrb.com/>) 等工具的的 Markdown 文件，Jekyll 是 GitHub 支持的静态页面生成器。
- Standardese (<https://github.com/standardese/standardese>)  
使用 libclang 编译代码，并以 HTML、Markdown、LaTeX 和手册页的形式提供输出。其目标（相当大胆地）是成为下一个 Doxygen。

# 第 11 章 安装和打包

我们的项目已经构建、测试和文档化，是时候向用户发布了。本章主要讨论为此需要采取的最后两个步骤：安装和打包。这些都是高级技术，建立在我们目前所学的基础之上：管理目标及其依赖关系、使用需求、生成器表达式等。

安装允许项目在系统范围内发现和访问。本章中，将介绍如何导出目标，以便另一个项目可以在不安装的情况下使用，以及如何安装项目，以便系统上的程序都可以轻松使用，并学习如何配置项目，以便够自动地将不同的工件类型放在正确的目录中。为了处理更高级的场景，将介绍用于安装文件和目录的低层命令，以及用于执行定制脚本和 CMake 的低层指令。

接下来，将学习如何设置可重用的 CMake 包，以便通过从其他项目调用 `find_package()` 来发现它们，将解释如何确保目标及其定义不固定在文件系统上的特定位置。还将讨论如何编写基本和高级配置文件，以及与包相关的版本文件。

然后，为了模块化，将简要介绍组件的概念，包括 CMake 包和 `install()` 指令。所有这些准备工作都是为本章的最后一个方面做准备：使用 CPack 来生成归档文件、安装程序、打包文件和不同操作系统中所有包管理器都能识别的包。这些可用于携带预构建的工件、可执行文件和库。这是终端用户使用软件最简单的方式。

本章中，我们将讨论以下主题：

- 只导出，不安装
- 在系统上安装
- 创建可重用的包
- 定义组件
- 使用 CPack 打包

## 11.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter11>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 11.2. 只导出，不安装

如何使项目 A 的目标对项目 B 可用？通常使用 `find_package()` 指令，但这需要创建一个包并将其安装到系统上。这种方法没问题，但需要一些工作量。有时，只需要一种非常快速的方法来构建一个项目，并使其目标可用于其他项目。

可以通过包含 A 的主列表文件来节省一些时间: 已经包含了所有的目标定义, 还可能包含许多其他东西: 全局配置、需求、带有副作用的 CMake 指令、附加依赖项, 以及在 B 中不想要的目标(例如单元测试)。所以, 不要这样做。最好是通过提供项目 B 可以使用 `include()` 包含的目标导出文件来进行:

```
cmake_minimum_required(VERSION 3.20.0)
project(B)
include(/path/to/project-A/ProjectATargets.cmake)
```

这样做将为 A 的所有目标提供定义(如 `add_library()` 和 `add_executable()` 等指令), 并设置正确的属性。

当然, 不需要手动编写这样的文件——不是非常 DRY 的方法。CMake 可以用 `export()` 指令生成这些文件, 该指令具有以下签名:

```
export(TARGETS [target1 [target2 [...]]]
 [NAMESPACE <namespace>] [APPEND] FILE <path>
 [EXPORT_LINK_INTERFACE_LIBRARIES])
```

必须在 TARGET 关键字之后提供想要导出的所有目标, 并在 FILE 之后提供目标文件名。其他参数可选:

- 建议使用 NAMESPACE 作为提示, 说明目标已从其他项目导入。
- APPEND 告诉 CMake, 不应该在写入之前擦除文件的内容。
- EXPORT\_LINK\_INTERFACE\_LIBRARIES 将导出目标链接依赖项(包括导入的和配置特定的变量)。

用示例 Calc 库来看看它的作用, 其提供了两个简单的方法:

```
1 // chapter-11/01-export/src/include/calc/calc.h
2
3 #pragma once
4 int Sum(int a, int b);
5 int Multiply(int a, int b);
```

我们这样声明其目标:

```
chapter-11/01-export/src/CMakeLists.txt

add_library(calc STATIC calc.cpp)
target_include_directories(calc INTERFACE include)
```

然后, 用 `export(TARGETS)` 指令生成导出文件:

```
chapter-11/01-export/CMakeLists.txt (fragment)

cmake_minimum_required(VERSION 3.20.0)
project(ExportCalcCXX)
```

```

add_subdirectory(src bin)
set(EXPORT_DIR "${CMAKE_CURRENT_BINARY_DIR}/cmake")
export(TARGETS calc
 FILE "${EXPORT_DIR}/CalcTargets.cmake"
 NAMESPACE Calc:::
)
...

```

可以看到 EXPORT\_DIR 变量设置为构建树的 cmake 子目录 (根据.cmake 文件的约定)。导出目标声明文件 CalcTargets.cmake，对于包含此文件的项目，有一个可见的 calc::calc 目标 calc。

注意，这个导出文件还是一个包，这个文件中的所有路径都是绝对的，并且硬编码到构建树中，所以它们不可重定位。

`export()` 指令也有一个简短的版本：

```

export(EXPORT <export> [NAMESPACE <namespace>] [FILE
<path>])

```

但需要 `<export>` 名称，而不是导出目标的列表。这样的 `<export>` 实例是由 `install(targets)` 定义的目标的命名列表。这里有一个例子，展示了这种简版在实践中如何使用：

```

chapter-11/01-export/CMakeLists.txt (continued)

...
install(TARGETS calc EXPORT CalcTargets)
export(EXPORT CalcTargets
 FILE "${EXPORT_DIR}/CalcTargets2.cmake"
 NAMESPACE Calc:::
)

```

现在，`export()` 和 `install()` 指令之间共享一个目标列表。

两种生成导出文件的方法将产生相同的结果，将包含一些样板代码和几行定义目标的代码。将/tmp/b 设置为构建树路径：

```

/tmp/b/cmake/CalcTargets.cmake (fragment)

Create imported target Calc::calc
add_library(Calc::calc STATIC IMPORTED)
set_target_properties(Calc::calc PROPERTIES
 INTERFACE_INCLUDE_DIRECTORIES
 "/root/examples/chapter11/01-export/src/include"
)
Import target "Calc::calc" for configuration ""
set_property(TARGET Calc::calc APPEND PROPERTY

```

```

IMPORTED_CONFIGURATIONS NOCONFIG
)
set_target_properties(Calc::calc PROPERTIES
IMPORTED_LINK_INTERFACE_LANGUAGES_NOCONFIG "CXX"
IMPORTED_LOCATION_NOCONFIG "/tmp/b/libcalc.a"
)

```

通常，我们不会编辑这个文件，甚至不会打开它，这里想突出显示这个生成文件中的硬编码路径。在当前的形式中，包不可重定位。若想改变这一点，需要先克服一些困难。我们将在下一节中探讨这为什么很重要。

### 11.3. 在系统上安装

第1章中，了解了 CMake 提供了一种命令行模式，可以在系统中安装已构建的项目：

```
cmake --install <dir> [<options>]
```

<dir> 是生成构建树的路径 (必需的)，<options> 如下：

- `--config <cfg>`: 这将为多配置生成器选择生成配置。
- `--component <comp>`: 这将安装限制为给定的组件。
- `--default-directory-permissions <permissions>`: 这将设置已安装目录的默认权限 (以 `<u=rwx,g=rx,o=rx>` 格式)。
- `--prefix <prefix>`: 这指定了非默认安装路径 (存储在 `CMAKE_INSTALL_PREFIX` 变量中)。对于类 Unix 系统，默认为 `/usr/local`；对于 Windows 系统，默认为 `C:/Program Files/${PROJECT_NAME}`。
- `-v, --verbose`: 这会使输出冗长 (也可以通过设置 `VERBOSE` 环境变量来实现)。

安装可以包含许多步骤，但其核心是将生成的构件和必要的依赖项复制到系统上的某个目录中。使用 CMake 进行安装不仅为所有 CMake 项目引入了一个标准，还实现了以下功能：

- 根据工件的类型 (通过遵循 GNU 编码标准)，为工件提供特定于平台的安装路径。
- 通过生成目标导出文件增强安装过程，这允许其他项目直接重用项目目标
- 通过配置文件创建可发现的包，配置文件包装目标导出文件和作者定义的特定于包的 CMake 宏和函数

这些特性非常强大，节省了大量时间，简化了以这种方式准备的项目的使用。执行基本安装的第一步，是将构建的构件复制到目标目录。

这就引出了 `install()` 指令及其各种模式：

- `install(TARGETS)`: 这将安装诸如库和可执行程序等输出构件。
- `install(FILES|PROGRAMS)`: 这将安装各个文件并设置权限。
- `install(DIRECTORY)`: 这将安装整个目录。
- `install(SCRIPT|CODE)`: 这将在安装期间运行 CMake 脚本或代码段。

- `install(EXPORT)`: 这将生成并安装一个目标导出文件。

将这些命令添加到列表文件中将在编译树中生成 `cmake_install.cmake`。虽然可以使用 `cmake -P` 手动调用此脚本，但不建议这样做。当 `CMake --install` 执行时，`CMake` 将在内部使用这个文件。

#### Note

即将发布的 `CMake` 版本还将支持安装运行时工件和依赖集，因此请务必查看最新的文档以了解更多信息。

每个 `install()` 模式都有大量的选项集。其中一些是共享的，以相同的方式工作：

- `DESTINATION`: 指定了安装路径。相对路径将以 `CMAKE_INSTALL_PREFIX` 作为前缀，而绝对路径将逐字使用（并且不受 `cpack` 支持）。
- `PERMISSIONS`: 这将在支持它们的平台上设置文件权限。可用的值有：`OWNER_READ`、`OWNER_WRITE`、`OWNER_EXECUTE`、`GROUP_READ`、`GROUP_WRITE`、`GROUP_EXECUTE`、`WORLD_READ`、`WORLD_WRITE`、`WORLD_EXECUTE`、`SETUID` 和 `SETGID`。安装期间创建的目录的默认权限可以通过 `CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS` 来设置。
- `CONFIGURATIONS`: 指定了一个配置列表（`Debug`、`Release`）。只有当前构建配置在此列表中，此命令中跟随此关键字的选项才会应用。
- `OPTIONAL`: 这将禁止在安装的文件不存在时引发错误。

特定组件的安装中也使用两个共享选项：`COMPONENT` 和 `EXCLUDE_FROM_ALL`。将在定义组件一节中详细讨论这些内容。

先来看看第一种安装方式：`install(TARGETS)`。

#### 11.3.1 安装逻辑目标

由 `add_library()` 和 `add_executable()` 定义的目标可以通过 `install(TARGETS)` 指令轻松安装。构建系统生成的构件复制到适当的目标目录，并设置适当的文件权限。该模式的一般签名如下：

```
install(TARGETS <target>... [EXPORT <export-name>]
 [<output-artifact-configuration> ...]
 [INCLUDES DESTINATION [<dir> ...]]
)
```

初始模式说明符（即 `TARGETS`）之后，必须提供想要安装的目标的列表。这里，可以选择地将其分配给带有 `EXPORT` 选项的命名导出，该导出可以在 `export(EXPORT)` 和 `install(EXPORT)` 中使用，以生成目标导出文件。然后，必须配置输出构件的安装（按类型分组）。还可以在每个目标的 `INTERFACE_INCLUDE_DIRECTORIES` 属性中提供一个目录列表，这些目录将添加到目标导出文件中。

[`<output-artifact-configuration>`...] 会提供配置块的列表。单个块的完整语法如下所示：

```
<TYPE> [DESTINATION <dir>] [PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
```

```
[COMPONENT <component>]
[NAMELINK_COMPONENT <component>]
[OPTIONAL] [EXCLUDE_FROM_ALL]
[NAMELINK_ONLY|NAMELINK_SKIP]
```

每个输出工件块都必须以 <TYPE> 开头 (这是必需的)。CMake 可以识别其中几个:

- ARCHIVE: 静态库 (.a) 和 DLL 为基于 windows 的系统导入库 (.lib)。
- LIBRARY: 动态库 (.so), 而不是 dll。
- RUNTIME: 可执行文件和 dll。
- OBJECTS: Object 库中的对象文件。
- FRAMEWORK: 设置了 FRAMEWORK 属性的静态库和动态库 (将排除在 ARCHIVE 和 LIBRARY 之外), 这特定于 macOS。
- BUNDLE: 用 MACOSX\_BUNDLE 标记的可执行文件 (也不是 RUNTIME 的一部分)。
- PUBLIC\_HEADER, PRIVATE\_HEADER, RESOURCE: 具有相同名称的目标属性中指定的文件 (Apple 平台上, 应该在 FRAMEWORK 或 BUNDLE 目标上设置)。

CMake 文档声称, 若只配置一种工件类型 (例如 LIBRARY), 则只会安装这种类型。对于 CMake 版本 3.20.0, 这是不正确的: 所有构件都将安装, 就像它们配置了默认选项一样。这可以通过为所有不需要的工件类型指定 <TYPE> EXCLUDE\_FROM\_ALL 来解决。

#### Note

一个 install(TARGETS) 可以有多个工件配置块。请注意, 每次调用只能指定每种类型中的一种, 若想为调试和发布配置配置 ARCHIVE 工件的不同目的地, 必须使用两个单独的 install(TARGETS…ARCHIVE)。

也可以省略类型名, 并为所有构件指定选项:

```
install(TARGETS executable static_lib1
 DESTINATION /tmp
)
```

然后, 对这些目标生成的每个文件执行安装。

另外, 并不需要使用 DESTINATION 提供安装目录。来看看这是为什么。

## 为不同的平台找到目标

目标路径公式如下:

```
 ${CMAKE_INSTALL_PREFIX} + ${DESTINATION}
```

若没有提供 DESTINATION, CMake 将为每种类型使用内置默认值:

| 工件类型                             | 猜测内置路径  | 安装目录变量                   |
|----------------------------------|---------|--------------------------|
| RUNTIME                          | bin     | CMAKE_INSTALL_BINDIR     |
| LIBRARY,ARCHIVE                  | lib     | CMAKE_INSTALL_LIBDIR     |
| PRIVATE_HEADER,<br>PUBLIC_HEADER | include | CMAKE_INSTALL_INCLUDEDIR |

虽然默认路径有时很有用，但并不适用于所有情况。例如，默认情况下，CMake 会“猜测”库的 DESTINATION 应该是 lib。对于所有类 Unix 系统，库的安装路径将为 /usr/local/lib，而在 Windows 上则像 C:\Program Files (x86)\<project-name>\lib。对于支持多重架构的 Debian 来说，这不是一个很好的选择，因为当 INSTALL\_PREFIX 为 /usr 时，需要一个到特定架构（例如 i386-linux-gnu）的路径。为每个平台找到正确的路径是类 Unix 系统的常见问题。为了正确理解它，需要遵循 GNU 编码标准（在扩展阅读部分可以找到该标准的链接）。

进行“猜测”之前，CMake 将检查是否为该工件类型设置了 CMAKE\_INSTALL\_<DIR>DIR 变量，并以那里为起始路径。需要的是一种算法，检测平台并用适当的路径填充安装目录变量。CMake 通过提供 GNUInstallDirs 实用程序模块简化了这一点，该模块通过相应地装目录来处理大多数平台。在调用 install() 指令之前 include()，就可以完成设置。

需要自定义配置的用户可以通过命令行 -DCMAKE\_INSTALL\_BINDIR=/path/in/the/system 提供安装目录。

不过，安装库的公共头文件可能有点棘手。为什么呢？

## 处理公共头文件

install(TARGETS) 文档建议在库目标的 PUBLIC\_HEADER 属性中指定公共头文件（分号分隔的列表）：

```
chapter-11/02-install-targets/src/CMakeLists.txt

add_library(calc STATIC calc.cpp)
target_include_directories(calc INTERFACE include)
set_target_properties(calc PROPERTIES
 PUBLIC_HEADER src/include/calc/calc.h
)
```

若在 Unix 中使用默认的“猜测”路径，文件将最终位于 /usr/local/include 中，这并不一定是最佳实践。理想情况下，希望将这些公共头文件放在一个目录中，该目录将指示它们的起源并引入名称空间；例如：/usr/local/include/calc。这样，这个系统上的所有项目就都能使用安装后的文件了：

```
#include <calc/calc.h>
```

大多数预处理器将带尖括号的指令识别为扫描标准系统目录的请求。这就是 GNUInstallDirs 模块的作用所在，其为 install() 指令定义了安装变量，不过也可以显式地使用它们。本例中，我们想在公共头文件的目标 calc 前加上 CMAKE\_INSTALL\_INCLUDEDIR：

```
chapter-11/02-install-targets/CMakeLists.txt
```

```

cmake_minimum_required(VERSION 3.20.0)
project(InstallTargets CXX)
add_subdirectory(src bin)

include(GNUInstallDirs)
install(TARGETS calc
 ARCHIVE
 PUBLIC_HEADER
 DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
)

```

包含 src 中的文件列表 (定义了 calc 目标) 后，必须配置静态库及其公共头文件的安装。我们已经包含了 GNUInstallDirs 模块，并显式地为 PUBLIC\_HEADERS 指定了 DESTINATION。在安装模式下运行 cmake 会完全按照预期工作：

```

cmake -S <source-tree> -B <build-tree>
cmake --build <build-tree>
cmake --install <build-tree>
-- Install configuration: ""
-- Installing: /usr/local/lib/libcalc.a
-- Installing: /usr/local/include/calc/calc.h

```

这对于这种基本情况很有效，但有一个缺点：这种方式指定的文件不会保留目录结构，都将安装在相同的目标中，即使嵌套在不同的基目录中。

新版本 (CMake 3.23.0) 的计划，以更好地使用 FILE\_SET 关键字管理头文件：

```

target_sources(<target>
 [<PUBLIC|PRIVATE|INTERFACE>
 [FILE_SET <name> TYPE <type> [BASE_DIR <dir>] FILES]
 <files>...
]...
)

```

有关官方论坛讨论的链接，请参见扩展阅读部分。在结束该选项之前，可以将此机制与 PRIVATE\_HEADER 和 RESOURCE 工件类型一起使用。但是如何指定更复杂的安装目录结构呢？

### 11.3.2 底层安装

现代 CMake 正在远离对文件的直接操作。理想情况下，会将其添加到逻辑目标中，并将其作为更高级别的抽象来表示所有底层资源：源文件、头文件、资源、配置等。主要优点是代码的简洁：通常，只需要修改一行就可以向目标添加文件。

但将每个已安装的文件添加到目标，并不总是可行或方便的。对于这种情况，有三种选择:install(FILES)、install(PROGRAMS) 和 install(DIRECTORY)。

## 使用 `install(FILES|PROGRAMS)` 安装文件集

FILES 和 PROGRAMS 模式非常相似，可用于安装公共头文件、文档、shell 脚本、配置和各种资产（包括图像、音频文件和要在运行时使用的数据集）。

下面是指令的签名：

```
install(<FILES|PROGRAMS> files...
 TYPE <type> | DESTINATION <dir>
 [PERMISSIONS permissions...]
 [CONFIGURATIONS [Debug|Release|...]]
 [COMPONENT <component>]
 [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL])
```

FILES 和 PROGRAMS 的主要区别在于对新复制的文件的默认文件权限设置。install(PROGRAMS) 也会为所有用户设置 EXECUTE，而 install(FILES) 不会（两者都将设置 OWNER\_WRITE, OWNER\_READ, GROUP\_READ 和 WORLD\_READ）。可以通过提供可选的 PERMISSIONS 关键字改变这种行为，然后，选择前一个关键字作为已安装内容的指示器:FILES 和 PROGRAMS。我们已经讨论了权限、配置和可选的工作原理，COMPONENT 和 EXCLUDE\_FROM\_ALL 将在后面的定义组件一节中讨论。

了解了关键字之后，需要列出想要安装的所有文件。CMake 支持相对路径和绝对路径，以及生成器表达式。若文件路径以生成器表达式开头，那么其必须是绝对路径。

下一个必需的关键字是 TYPE 或 DESTINATION，可以显式地提供 DESTINATION 路径，或者要求 CMake 查找特定的 TYPE 文件。与 install(TARGETS) 不同，TYPE 没有声明要选择性地安装所提供的要安装文件的子集。然而，计算安装路径遵循相同的模式 (+ 符号表示平台特定的路径分隔符)：

```
 ${CMAKE_INSTALL_PREFIX} + ${DESTINATION}
```

类似地，每个 TYPE 都有内置的猜测路径：

| 文件类型        | 内置猜测路径           | 安装目录变量                      |
|-------------|------------------|-----------------------------|
| BIN         | bin              | CMAKE_INSTALL_BINDIR        |
| SBIN        | sbin             | CMAKE_INSTALL_SBINDIR       |
| LIB         | lib              | CMAKE_INSTALL_LIBDIR        |
| INCLUDE     | include          | CMAKE_INSTALL_INCLUDEDIR    |
| SYSCONF     | etc              | CMAKE_INSTALL_SYSCONFDIR    |
| SHAREDSTATE | com              | CMAKE_INSTALL_SHARESTATEDIR |
| LOCALSTATE  | var              | CMAKE_INSTALL_LOCALSTATEDIR |
| RUNSTATE    | \$LOCALSTATE/run | CMAKE_INSTALL_RUNSTATEDIR   |
| INFO        | \$DATAROOT       | CMAKE_INSTALL_DATADIR       |
| LOCALE      | \$DATAROOT/info  | CMAKE_INSTALL_INFODIR       |
| MAN         | \$DATAROOT/man   | CMAKE_INSTALL_MANDIR        |
| DOC         | \$DATAROOT/doc   | CMAKE_INSTALL_DOCDIR        |

这里的 behavior 遵循为不同平台计算正确目标一节中描述的相同原则: 若没有为这个 TYPE 文件设置安装目录变量, CMake 将退回到默认的“猜测”路径。同样, 可以使用 `GNUInstallDirs` 模块来实现可移植性。

表中的一些内置猜测路径带有安装目录变量的前缀:

- `$LOCALSTATE` 是 `CMAKE_INSTALL_LOCALSTATEDIR` 或者默认为 `var`
- `$DATAROOT` 是 `CMAKE_INSTALL_DATAROOTDIR` 或者默认为 `share`

与 `install(TARGETS)` 一样, 若包含 `GNUInstallDirs` 模块, 将提供特定于平台的安装目录变量。来看一个例子:

```
chapter-11/03-install-files/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(InstallFiles CXX)

include(GNUInstallDirs)
install(FILES
 src/include/calc/calc.h
 src/include/calc/nested/calc_extended.h
 DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
)
```

本例中, CMake 将在系统范围的 `include` 目录中特定于项目的子目录中安装两个只包含头文件的库——即 `calc.h` 和 `nested/calc_extended.h`。

### Note

从 GNUInstallDirs 源代码中知道，CMAKE\_INSTALL\_INCLUDEDIR 包含所有支持平台的相同路径，但为了可读性和与更多动态变量的一致性，仍然建议使用。例如，CMAKE\_INSTALL\_LIBDIR 会因架构和分布而不同——lib、lib64 或 lib/<multiarchtuple>。

CMake 3.20 还在 `install(FILES|PROGRAMS)` 指令中添加了一个比较有用的关键字，后面必须跟一个新的文件名。

本节中的示例展示了在适当的目录中安装文件很容易。但有一个问题，来看看安装输出：

```
cmake -S <source-tree> -B <build-tree>
cmake --build <build-tree>
cmake --install <build-tree>
-- Install configuration: ""
-- Installing: /usr/local/include/calc/calc.h
-- Installing: /usr/local/include/calc/calc_extended.h
```

两个文件都安装在同一个目录中，而不管是否嵌套。有时候，这并不是我们想要的。下一节中，我们将学习如何处理这个问题。

### 使用整个目录

若不想向安装命令中添加单个文件，则可以选择更通用的方法，使用整个目录。为此目的创建了 `install(DIRECTORY)` 模式，将把列出的目录复制到所选的目的。让我们看看：

```
install(DIRECTORY dirs...
 TYPE <type> | DESTINATION <dir>
 [FILE_PERMISSIONS permissions...]
 [DIRECTORY_PERMISSIONS permissions...]
 [USE_SOURCE_PERMISSIONS] [OPTIONAL] [MESSAGE_NEVER]
 [CONFIGURATIONS [Debug|Release|...]]
 [COMPONENT <component>] [EXCLUDE_FROM_ALL]
 [FILES_MATCHING]
 [[PATTERN <pattern> | REGEX <regex>] [EXCLUDE]
 [PERMISSIONS permissions...]] [...])
```

许多选项从 `install(FILES|PROGRAMS)` 开始重复，工作原理是一样的。有一个细节值得注意：若 `DIRECTORY` 关键字后提供的路径不以/结尾，路径的最后一个目录将追加到目标目录：

```
install(DIRECTORY a DESTINATION /x)
```

这将创建一个名为/x/a 的目录，并将 a 的内容复制到其中。现在，看看下面的代码：

```
install(DIRECTORY a/ DESTINATION /x)
```

这将直接将 a 的内容复制到/x。

install(DIRECTORY) 还引入了文件不可用的其他机制:

- 静默输出
- 扩展权限控制
- 文件/目录过滤

从输出静默选项 MESSAGE\_NEVER 开始, 安装期间禁用输出诊断。当安装的目录中有很多文件, 打印太乱的时候, 非常有用。

接下来是权限。此 install() 模式支持三种设置权限的选项:

- USE\_SOURCE\_PERMISSIONS 的工作原理和预期的一样——在原文件之后的已安装文件上设置权限。这只在 FILE\_PERMISSIONS 未设置时有效。
- FILE\_PERMISSIONS 也是不言自明, 其可以指定要对已安装文件和目录设置的权限。默认的权限是 OWNER\_WRITE, OWNER\_READ, GROUP\_READ 和 WORLD\_READ。
- DIRECTORY\_PERMISSIONS 工作原理与前一个选项类似, 但将为所有用户设置额外的 EXECUTE 权限(因为目录上的 EXECUTE, 在类 Unix 系统理解, 就是列出目录内容的权限)。

注意, CMake 将忽略不支持权限选项的平台上的权限选项。通过在每个过滤表达式之后添加 PERMISSIONS 关键字, 可以实现更多的权限控制: 与之匹配的文件或目录将接收在该关键字之后指定的权限。

来看看过滤器或“globbing”表达式。可以设置多个过滤器, 以控制从源目录安装哪些文件/目录。其有以下语法:

```
PATTERN <p> | REGEX <r> [EXCLUDE] [PERMISSIONS
<permissions>]
```

有两种匹配方法可供选择:

- 对于 PATTERN, 这是一个更简单的选项, 可以用? 占位符(匹配任何字符)和通配符, \*(匹配任何字符串)。只匹配以结尾的路径。
- 另一方面, REGEX 选项更高级——支持正则表达式。还允许匹配路径的任何部分(可以使用 ^ 和 \$ 来表示路径的开始和结束)。

还可以在第一个筛选器之前设置 FILES\_MATCHING 关键字, 这将使指定的筛选器应用于文件, 而不是目录。

记住两点:

- FILES\_MATCHING 需要一个包含过滤器, 可以排除一些文件, 但除非还需要添加一个表达式来包含其中的一些文件, 否则不会复制任何文件。但是, 将创建所有目录, 而不考虑过滤。
- 默认情况下会过滤所有子目录, 可能只能过滤掉它们。

对于每个过滤方法, 可以选择排除匹配的路径(这只在不使用 FILES\_MATCHING 时有效)。

通过在筛选器后添加 PERMISSIONS 关键字和所需权限列表, 可以为所有匹配的路径设置特定的权限。本例中, 将以三种不同的方式安装三个目录。将在运行时使用一些静态数据文件:

```
data
```

```
- data.csv
```

还需要一些公共头文件，位于 `src` 目录中的其他不相关文件中：

```
src
- include
 - calc
 - calc.h
 - ignored
 - empty. file
 - nested
 - calc_extended.h
```

最后，需要两个嵌套级别的配置文件。为了让事情更有趣，可使`/etc/calc/`的内容只有文件所有者可访问：

```
etc
- calc
 - nested.conf
- sample.conf
```

要用静态数据文件安装目录，将用 `install(directory)` 指令的最基本形式：

```
chapter-11/04-install-directories/CMakeLists.txt (fragment)

cmake_minimum_required(VERSION 3.20.0)
project(InstallDirectories CXX)
install(DIRECTORY data/ DESTINATION share/calc)
...
```

这个指令将简单地获取数据目录的所有内容，并将其放在  `${CMAKE_INSTALL_PREFIX}` 和 `share/calc` 中。注意，源路径以/符号结束，表示不想复制数据目录本身，而只是复制其内容。

第二种情况则相反：不添加尾随/，因为应该包含目录。因为依赖于包含文件类型的特定于系统的路径，是由 `GNUInstallDirs` 提供的（注意 `INCLUDE` 和 `EXCLUDE` 关键字如何表示不相关）：

```
chapter-11/04-install-directories/CMakeLists.txt (fragment)

...
include(GNUInstallDirs)
install(DIRECTORY src/include/calc TYPE INCLUDE
 PATTERN "ignored" EXCLUDE
 PATTERN "calc_extended.h" EXCLUDE
)
...
```

此外，从这个操作中排除了两个路径：整个忽略的目录和所有以 `calc_extended.h` 结尾的文件（记住 `PATTERN` 如何工作）。

第三种情况安装一些默认配置文件并设置其权限:

```
chapter-11/04-install-directories/CMakeLists.txt (fragment)

...
install(DIRECTORY etc/ TYPE SYSCONF
DIRECTORY_PERMISSIONS
 OWNER_READ OWNER_WRITE OWNER_EXECUTE
PATTERN "nested.conf"
 PERMISSIONS OWNER_READ OWNER_WRITE
)
)
```

我们对将 etc 从源路径追加到 SYSCONF 类型的路径 (包含 GNUInstallDirs 已经提供) 不感兴趣。我们最终将把文件放在/etc/etc 中。所以，这里必须指定两个权限规则:

- 子目录应该只能由所有者编辑和查看。
- 以 nested.conf 结尾的文件只能由所有者编辑。

安装目录处理许多不同的用例，但是对于真正高级的安装场景 (例如安装后配置)，可能涉及外部工具。那要怎么做呢？

### 11.3.3 安装期间使用脚本

若曾经在类 Unix 系统上安装过动态库，可能会记得，在使用它之前，可能需要告诉动态链接器扫描受信任的目录，并通过调用 ldconfig 来构建缓存 (有关参考资料，请参阅扩展阅读部分)。若想让安装完全自动化，CMake 提供 install(SCRIPT|CODE) 指令来支持这种情况。下面是完整的指令签名:

```
install([[SCRIPT <file>] [CODE <code>]]
[ALL_COMPONENTS | COMPONENT <component>]
[EXCLUDE_FROM_ALL] [...])
```

应该选择 SCRIPT 或 CODE 模式并提供适当的参数——要么运行的 CMake 脚本的路径，要么是要在安装期间执行的 CMake 代码段。要了解这是如何工作的，这里修改 02-install-targets 示例来构建一个动态库:

```
chapter-11/05-install-code/src/CMakeLists.txt

add_library(calc SHARED calc.cpp)
target_include_directories(calc INTERFACE include)
set_target_properties(calc PROPERTIES
 PUBLIC_HEADER src/include/calc/calc.h
)
```

需要在安装脚本中将工件类型从 ARCHIVE 更改为 LIBRARY，以复制文件。然后，可以在下面添加逻辑来运行 ldconfig：

```
chapter-11/05-install-code/CMakeLists.txt (fragment)

...
install(TARGETS calc LIBRARY
PUBLIC_HEADER
DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
)
if (UNIX)
install(CODE "execute_process(COMMAND ldconfig)")
endif()
```

if() 条件检查命令是否与操作系统匹配 (在 Windows 或 macOS 上执行 ldconfig 是不正确的)。当然，代码必须具有有效的 CMake 语法才能工作 (但在初始构建期间不会进行检查，任何故障都会在安装过程中出现)。

运行安装命令后，可以通过打印缓存库来确认：

```
cmake -S <source-tree> -B <build-tree>
cmake --build <build-tree>
cmake --install <build-tree>
-- Install configuration: ""
-- Installing: /usr/local/lib/libcalc.so
-- Installing: /usr/local/include/calc/calc.h
ldconfig -p | grep libcalc
libcalc.so (libc6,x86-64) => /usr/local/lib/libcalc.so
```

两种模式都支持生成器表达式，这个命令和 CMake 本身一样多才多艺，可以用于各种各样的事情：为用户打印消息、验证安装是否成功、大量配置、文件签名。

现在，知道了在系统上安装一组文件的所有不同方法。接下来，我们学习如何将其转换为其他 CMake 项目的可用包。

## 11.4. 创建可重用的包

前面的章节中已经使用了 `find_package()`，用来简化了整个过程。为了使项目可以通过这个指令访问，需要完成几个步骤，以便 CMake 可以将项目作为一个包来处理：

- 目标可以重新定位。
- 将目标导出文件安装到标准位置。
- 为包创建配置文件和版本文件。

为什么目标需要重新定位，如何做到呢？

#### 11.4.1 理解可重定位目标

安装解决了许多问题，但也引入了一些复杂性：CMAKE\_INSTALL\_PREFIX 不仅是特定于平台的，而且还可以由用户在安装阶段使用--prefix 设置。然而，目标导出文件是在安装之前生成的，在构建阶段并不知道安装的工件将去哪里。看看下面的代码：

```
chapter-11/01-export/src/CMakeLists.txt

add_library(calc STATIC calc.cpp)
target_include_directories(calc INTERFACE include)
```

本例中，将 include 目录添加到 calc 的 include 目录中。由于这是一个相对路径，CMake 导出的目标生成将在此路径前加上 CMAKE\_CURRENT\_SOURCE\_DIR 的内容，该变量指向该列表文件所在的目录。

然而，这并不能解决问题。安装项目不再需要来自源文件或构建树的文件，所有内容（包括库头文件）都可复制到共享位置，例如 Linux 上的 /usr/lib/calc/。我们不能在另一个项目中使用此代码段中定义的目标，因为目标的 include 目录路径仍然指向其源树。

CMake 用两个生成器表达式解决了这个问题，将根据上下文过滤掉表达式：

- \${<BUILD\_INTERFACE>}：这包括常规构建的内容，但不包括安装内容。
- \${<INSTALL\_INTERFACE>}：包括用于安装的内容，但不包括常规构建的内容。

下面的代码展示了如何在实践中使用：

```
chapter-11/06-install-export/src/CMakeLists.txt

add_library(calc STATIC calc.cpp)
target_include_directories(calc INTERFACE
 "${<BUILD_INTERFACE>:${CMAKE_CURRENT_SOURCE_DIR}/include}"
 "${<INSTALL_INTERFACE>:${CMAKE_INSTALL_INCLUDEDIR}>""
)
set_target_properties(calc PROPERTIES
 PUBLIC_HEADER src/include/calc/calc.h
)
```

对于常规的构建，calc 目标属性 INTERFACE\_INCLUDE\_DIRECTORIES 的值将展开：

```
"/root/examples/chapter-11/05-package/src/include" ""
```

空双引号表明排除 INSTALL\_INTERFACE 中的值，计算为空字符串。安装时，值会像这样展开：

```
"" "/usr/lib/calc/include"
```

这一次, BUILD\_INTERFACE 生成器表达式中提供的值计算为空字符串, 所以只剩下来自另一个生成器表达式的值。

关于 CMAKE\_INSTALL\_PREFIX 还有一句话: 这个变量不应该用作目标中指定路径的组件。其将在构建阶段进行计算, 使路径成为绝对路径, 不一定与安装阶段提供的路径相同 (用户可以使用--prefix 选项)。相反, 使用 \$<install\_prefix> 生成器表达式:

```
target_include_directories(my_target PUBLIC
 $<INSTALL_INTERFACE:$<INSTALL_PREFIX>/include>/MyTarget>
)
```

可以使用相对路径 (其会使用正确的安装前缀):

```
target_include_directories(my_target PUBLIC
 $<INSTALL_INTERFACE:include>/MyTarget>
)
```

请查看官方文档以获得更多的示例和信息 (链接可以在扩展阅读部分找到)。

既然目标是“兼容安装”, 就可以安全地生成, 并安装目标导出文件了。

#### 11.4.2 安装目标导出文件

在前面的章节中讨论了目标导出文件, 与用于安装的目标导出文件非常相似:

```
install(EXPORT <export-name> DESTINATION <dir>
 [NAMESPACE <namespace>] [[FILE <name>.cmake] |
 [PERMISSIONS permissions...]
 [CONFIGURATIONS [Debug|Release|...]]
 [EXPORT_LINK_INTERFACE_LIBRARIES]
 [COMPONENT <component>]
 [EXCLUDE_FROM_ALL])
```

这是“普通”的 export(export) 和其他 install() 指令的组合 (其选项的工作方式相同), 其为命名导出创建并安装目标导出文件, 该文件必须使用 install(TARGETS) 指令定义。这里的主要区别是, 生成的导出文件将包含在 INSTALL\_INTERFACE 生成器表达式中计算的目标路径, 而不是像 export(EXPORT) 那样在 BUILD\_INTERFACE 中计算的目标路径。

本例中, 将使用 chapter-11/06-install-export/src/CMakeLists.txt 为目标生成并安装目标导出文件。为此, 必须在顶层列表文件中使用 install(EXPORT):

```
chapter-11/06-install-export/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
```

```

project(InstallExport CXX)
include(GNUInstallDirs) # so it's available in ./src/
add_subdirectory(src bin)

install(TARGETS calc EXPORT CalcTargets ARCHIVE
PUBLIC_HEADER DESTINATION
${CMAKE_INSTALL_INCLUDEDIR}/calc
)
install(EXPORT CalcTargets
DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
NAMESPACE Calc::
)

```

请注意如何在 `install(EXPORT)` 中如何引用 `CalcTargets` 导出名称。

在构建树中运行 `cmake --install` 会在指定的目录生成导出文件:

```

...
-- Installing: /usr/local/lib/calc/cmake/CalcTargets.cmake
-- Installing: /usr/local/lib/calc/cmake/CalcTargets-noconfig.cmake

```

若由于某种原因，覆盖目标导出文件的默认名称(<导出名称>.cmake)不适合，可以添加 `FILE` 新名称。参数更改它(文件名必须以.cmake结尾)。

不要对此感到困惑——目标导出文件不是配置文件，还不能使用 `find_package()` 来使用安装的目标，可以使用 `include()` 直接导出文件。那么，如何定义可以让其他项目使用的包呢？让我们一探究竟！

#### 11.4.3 编写配置文件

完整的包定义由目标导出文件、包的配置文件和包的版本文件组成，从技术上讲，`find_package()` 工作所需要的只是一个配置文件。其认为是一个包定义，负责提供包函数和宏、检查需求、查找依赖关系，以及包括目标导出文件。

正如前面提到的，用户可以使用以下命令在系统的任何地方安装：

```
cmake --install <build tree> --prefix=<installation path>
```

这个前缀决定将安装的文件复制到哪里。为了支持这一点，需要确保以下几点：

- 目标属性上的路径可以重新定位。
- 配置文件中使用的路径是相对的。

要使用安装在非默认位置的包，项目需要在配置阶段通过 `CMAKE_PREFIX_PATH` 变量提供<安装路径>。可以使用下面的命令：

```
cmake -B <build tree> -DCMAKE_PREFIX_PATH=<installation path>
```

find\_package() 指令将以特定于平台的方式扫描文档 (链接在扩展阅读部分) 中列出的路径列表。在 Windows 和类 Unix 系统上检查的模式如下：

```
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)
```

在 lib/calc/cmake 这样的路径中安装 config 文件应该可以正常工作。另外，必须强调配置文件必须命名为 <PackageName>-config.cmake 或 <PackageName>Config.cmake，才能找到。

将 config 文件的安装添加到 06-install-export 示例中：

```
chapter-11/07-config-file/CMakeLists.txt (fragment)

...
install(EXPORT CalcTargets
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
 NAMESPACE Calc::
)
install(FILES "CalcConfig.cmake"
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
)
```

这将安装 CalcConfig。从相同的源目录 (CMAKE\_INSTALL\_LIBDIR 将计算为正确的 lib 路径)。可以提供的最基本的配置文件由包含目标导出文件组成：

```
chapter-11/07-config-file/CalcConfig.cmake

include("${CMAKE_CURRENT_LIST_DIR}/CalcTargets.cmake")
```

CMAKE\_CURRENT\_LIST\_DIR 变量指配置文件所在的目录。因为 CalcConfig.cmake 和 CalcTargets.cmake 安装在示例中的相同目录中 (通过 install(EXPORT) 设置)，目标导出文件将正确的包含。

为了确保包的可用性，需要创建一个简单的项目，只包含一个列表文件：

```
chapter-11/08-find-package/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(FindCalcPackage CXX)

find_package(Calc REQUIRED)
include(CMakePrintHelpers)
```

```

message("CMAKE_PREFIX_PATH: ${CMAKE_PREFIX_PATH}")
message("CALC_FOUND: ${Calc_FOUND}")
cmake_print_properties(TARGETS "Calc::calc" PROPERTIES
 IMPORTED_CONFIGURATIONS
 INTERFACE_INCLUDE_DIRECTORIES
)

```

为了在实践中测试这一点，可以构建并将 07-config-file 示例安装到一个目录中，然后构建 08-find-package，同时使用 CMAKE\_PREFIX\_PATH 参数引用：

```

cmake -S <source-tree-of-07> -B <build-tree-of-07>
cmake --build <build-tree-of-07>
cmake --install <build-tree-of-07>
cmake -S <source-tree-of-08> -B <build-tree-of-08>
-DCMAKE_PREFIX_PATH=<build-tree-of-07>

```

这将产生以下输出 (所有 <\_tree-of\_> 占位符将使用实际路径替换)：

```

CMAKE_PREFIX_PATH: <build-tree-of-07>
CALC_FOUND: 1
--
Properties for TARGET Calc::calc:
 Calc::calc.IMPORTED_CONFIGURATIONS = "NOCONFIG"
 Calc::calcINTERFACE_INCLUDE_DIRECTORIES =
 "<buildtree-of-07>/include"
-- Configuring done
-- Generating done
-- Build files have been written to: <build-tree-of-08>

```

找到了 CalcTargets.cmake，并正确地包含了，并将包含目录的路径设置为遵循所选的前缀。这解决了大多数情况下的打包问题。现在，让我们学习如何处理更高级的情况。

#### 11.4.4 创建高级配置文件

若要管理的东西多于一个目标导出文件，在配置文件中包含一些宏可能会很有用。CMakePackageConfigHelpers 模块可以使用 configure\_package\_config\_file()。这里，需要提供一个模板文件，将在该文件中插入 CMake 变量，以生成一个包含两个嵌入宏定义的配置文件：

- set\_and\_check(<variable><path>): 这类似于 set()，但它检查 <路径> 是否实际存在，否则将以 FATAL\_ERROR 失败。建议在配置文件中使用它来尽早检测不正确的路径。

- `check_required_components(<PackageName>)`: 这将添加到配置文件的末尾，并将验证包中是否已经找到了用户在 `find_package(<package> REQUIRED <component>)` 中所需要的所有组件。这是通过检查 `<package>_<component>_FOUND` 变量是否为真来实现。

生成配置文件时，可以为安装阶段准备更复杂的目录树路径。看看下面的签名：

```
configure_package_config_file(<template> <output>
 INSTALL_DESTINATION <path>
 [PATH_VARS <var1> <var2> ... <varN>]
 [NO_SET_AND_CHECK_MACRO]
 [NO_CHECK_REQUIRED_COMPONENTS_MACRO]
 [INSTALL_PREFIX <path>]
)
```

提供的文件是 `<template>`，将用变量存储并存储在 `<output>` 路径中。这里，`INSTALL_DESTINATION` 之后需要的路径将用来存储在 `PATH_VARS` 中列出变量的路径，使它们相对于安装目标路径。还可以通过提供 `INSTALL_DESTINATION` 作为基路径，来表明 `INSTALL_DESTINATION` 是相对于 `INSTALL_PREFIX` 的。

`NO_SET_AND_CHECK_MACRO` 和 `NO_CHECK_REQUIRED_COMPONENTS_MACRO` 告诉 CMake 不要将这些宏定义添加到生成的配置文件中。让来看看其在实践中的表现。同样，我们将扩展 06-install-export 示例：

```
chapter-11/09-advanced-config/CMakeLists.txt (fragment)

...
install(EXPORT CalcTargets
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
 NAMESPACE Calc::
)

include(CMakePackageConfigHelpers)
set(LIB_INSTALL_DIR ${CMAKE_INSTALL_LIBDIR}/calc)
configure_package_config_file(
 ${CMAKE_CURRENT_SOURCE_DIR}/CalcConfig.cmake.in
 "${CMAKE_CURRENT_BINARY_DIR}/CalcConfig.cmake"
 INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
 PATH_VARS LIB_INSTALL_DIR
)
install(FILES "${CMAKE_CURRENT_BINARY_DIR}/CalcConfig.cmake"
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
)
```

来看看在前面的代码中必须做什么：

1. `Include()` 带有辅助函数的实用程序模块。
2. `set()` 用于创建可重定位路径的变量。
3. 生成 `CalcConfig.cmake` 配置文件，使用位于源树中的 `CalcConfig.cmake.in` 模板构建树的配置文件。最后，提供 `LIB_INSTALL_DIR` 作为相对于 `INSTALL_DESTINATION` 或  `${CMAKE_INSTALL_LIBDIR}/calc/cmake` 的变量名。
4. 传递为安装构建树生成的配置文件 (`FILE`)。

注意，`install(FILE)` 中的 `DESTINATION` 和 `install(FILES)` 中的 `INSTALL_DESTINATION` 相同，以便能够正确地计算相对路径。

最后，需要一个配置文件模板 (文件名通常以.in 作为后缀):

```
chapter-11/09-advanced-config/CalcConfig.cmake.in

@PACKAGE_INIT@

set_and_check(CALC_LIB_DIR "@PACKAGE_LIB_INSTALL_DIR@")
include("${CALC_LIB_DIR}/cmake/CalcTargets.cmake")

check_required_components(Calc)
```

应该以 `@PACKAGE_INIT@` 占位符开始。生成器将用 `set_and_check` 和 `check_required_components` 的定义填充，以便其可用于项目。你可能从 `configure_file()` 中认出了这些 @ 占位符 @——工作原理与在 C++ 文件中相同。

接下来，将 `(CALC_LIB_DIR)` 设置为在 `@PACKAGE_LIB_INSTALL_DIR@` 占位符中传递的路径，将包含列表文件中提供的 `$LIB_INSTALL_DIR` 的路径，但其将相对于安装路径，并用来包含目标导出文件。

最后，`check_required_components()` 验证是否已经找到包使用者所需的所有组件。建议添加此指令，即使包没有任何组件，也要验证用户没有意外添加不受支持的需求。

`CalcConfig.cmake` 配置文件，当以这种方式生成时：

```
Expanded from @PACKAGE_INIT@ by
configure_package_config_file() #####
#####
Any changes to this file will be overwritten by the
next CMake run #####
#####
The input file was CalcConfig.cmake.in
get_filename_component(PACKAGE_PREFIX_DIR
"${CMAKE_CURRENT_LIST_DIR}/../../../../" ABSOLUTE)
macro(set_and_check _var _file) # ... removed for brevity
macro(check_required_components _NAME) # ... removed for
brevity
set_and_check(CALC_LIB_DIR
"${PACKAGE_PREFIX_DIR}/lib/calc")
```

```

include("${CALC_LIB_DIR}/cmake/CalcTargets.cmake")
check_required_components(Calc)
#####
#####

```

下图表显示了不同的包文件是如何相互关联的，从一个角度分析了这个问题：

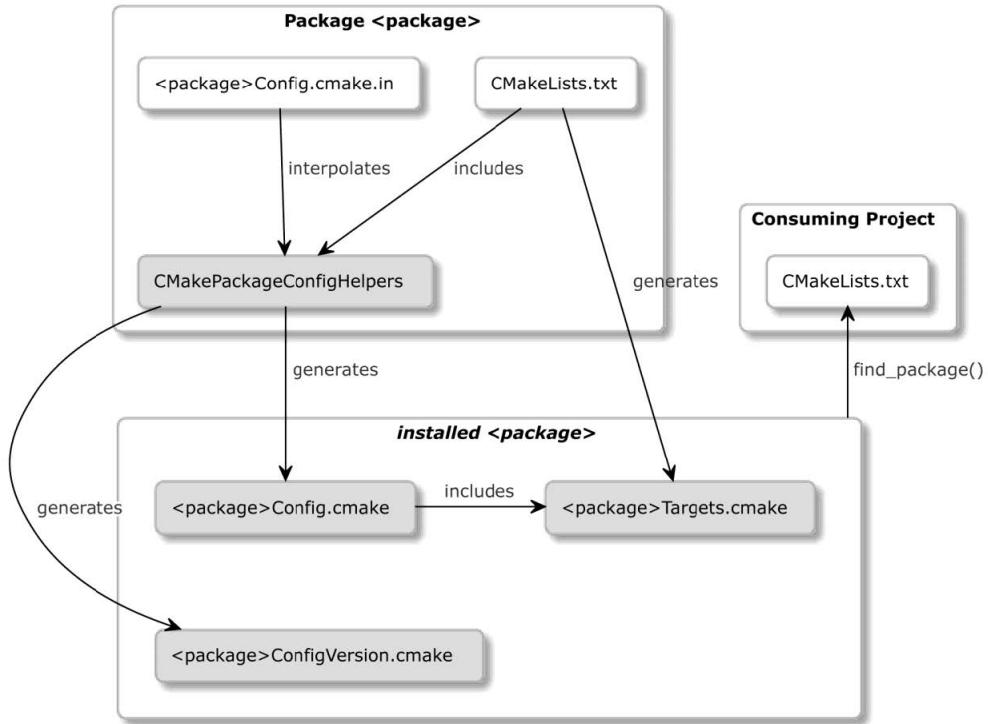


图 11.1 高级包的文件结构

包的所有必需的子依赖项也必须在包配置文件中找到。这可以通过调用 CMakeFindDependencyMacro 助手模块中的 `find_dependency()` 宏来实现。我们在第 7 章中学过了如何使用。

若决定向其他项目公开宏或函数，建议将定义放在一个单独的文件中，这样就可以使用 `include()` 包含配置文件了。

有趣的是，`CMakePackageConfigHelpers` 还提供了辅助指令来生成包的版本文件。

#### 11.4.5 生成包版本文件

随着包的增长，将慢慢获得新的功能，旧的功能将会标记为弃用，并最终删除。在使用包的开发人员可用的变更日志中跟踪这些修改是很重要的。当需要一个特定的特性时，开发人员可以找到支持它的最低版本，并使用它作为 `find_package()` 的参数：

```
find_package(Calc 1.2.3 REQUIRED)
```

`CMake` 将在配置文件中搜索 `Calc`，并检查是否有一个名为 `<config-file>-version.cmake` 的版本文件，或存在于同一目录中的 `<config-file>Version.cmake`，即 `CalcConfigVersion.cmake`。接下来，将读取该文件的版本信息以及它提供的与其他版本的兼容性。若没有按要求安装 1.2.3 版本，但可能安

装了 1.3.5，将标记为与任何旧版本“兼容”。CMake 很乐意接受这样的包，因为它知道包供应商提供向后兼容。

可以使用 CMakePackageConfigHelpers 实用模块通过 write\_basic\_package\_version\_file() 来生成包的版本文件：

```
write_basic_package_version_file(<filename> [VERSION <ver>]
 COMPATIBILITY <AnyNewerVersion | SameMajorVersion |
 SameMinorVersion | ExactVersion>
 [ARCH_INDEPENDENT]
)
```

首先，需要提供想要创建的工件的 <filename> 属性，必须遵循前面概述的规则。除此之外，应该在构建树中存储所有生成的文件。

可选地，可以传递一个显式的 VERSION(通常的格式，major.minor.patch)。若不这样做，则将使用 project() 中提供的版本。

COMPATIBILITY 关键字的含义不言自明：

- ExactVersion 必须匹配版本的所有三个组件，不支持范围版本：find\_package(<package> 1.2.8...1.3.4).
- 若前两个组件相同(忽略 patch)，则 SameMinorVersion 匹配。
- 若第一个组件相同(忽略 minor 和 patch)，则匹配 SameMajorVersion。
- AnyNewerVersion 似乎有一个相反的名称：将匹配任何旧版本。换句话说，版本 1.4.2 上的将很适合 find\_package(<package> 1.2.8).

通常，包都必须为与使用项目相匹配的相同架构构建(执行精确检查)。但对于不编译的包(只编译头库、宏包等)，可以指定 ARCH\_INDEPENDENT 关键字来跳过此检查。

现在，举一个实际的例子。下面的代码展示了如何为 06-install-export 示例中启动的项目提供版本文件：

```
chapter-11/10-version-file/CMakeLists.txt (fragment)

cmake_minimum_required(VERSION 3.20.0)
project(VersionFile VERSION 1.2.3 LANGUAGES CXX)
...
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
 "${CMAKE_CURRENT_BINARY_DIR}/CalcConfigVersion.cmake"
 COMPATIBILITY AnyNewerVersion
)
install(FILES "CalcConfig.cmake"
 "${CMAKE_CURRENT_BINARY_DIR}/CalcConfigVersion.cmake"
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
)
```

方便起见，需要在文件顶部的 `project()` 中配置包的版本。需要通过添加 `LANGUAGE` 关键字，从简短的 `project(<name> <languages>)` 语法切换到显式的完整语法。

包含辅助程序模块之后，调用生成命令将文件写入构建树，其名称符合 `find_package()` 要求的模式。这里，故意跳过 `VERSION` 关键字，以便从 `PROJECT_VERSION` 变量中读取版本。我们还将我们的包标记为完全向后兼容 `COMPATIBILITY AnyNewerVersion`。之后，将包版本文件安装到与 `CalcConfig.cmake` 相同的目录。就是这样，包已经完全配置好了。

下一节中，将学习什么是组件，以及如何将它们与包一起使用。

## 11.5. 定义组件

我们将通过澄清组件这个术语可能引起的混淆来开始讨论包组件。查看 `find_package()` 的完整签名：

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
[REQUIRED] [[COMPONENTS] [components...]]
[OPTIONAL_COMPONENTS components...]
[NO_POLICY_SCOPE])
```

这里的组件不应该与 `install()` 指令中使用的 `COMPONENT` 关键字混淆。尽管名称相同，但它们是不同的概念，必须分开理解。我们将在下面的小节中更详细地讨论。

### 11.5.1 在 `find_package()` 中使用组件

当使用带有组件列表的 `find_package()` 或 `OPTIONAL_COMPONENTS` 时，我们只对提供这些组件的包感兴趣。然而，要了解包需要验证的需求，若包供应商没有向创建高级配置文件一节中提到的配置文件添加必要的检查，那么什么也不会发生。

请求的组件传递到 `<package>_FIND_COMPONENTS` 变量中的配置文件（可选和不可选）。此外，对于每个非可选组件，将设置一个 `<package>_FIND_REQUIRED_<component>`。作为包的作者，可以编写一个宏来扫描这个列表，并检查是否已经提供了所有必需的组件。但这里不需要这样做——这是 `check_required_components()` 的工作。配置文件应该在找到必要的组件时设置 `<Package>_<Component>_FOUND` 变量，文件末尾的宏将检查是否设置了所有必需的变量。

### 11.5.2 在 `install()` 指令中使用组件

有些生成的工件可能不需要为所有场景安装。例如，一个项目可能为了开发目的而安装静态库和公共头文件，但默认情况下，只能为运行时安装一个动态库。为了使这种双重行为成为可能，可以通过使用 `COMPONENT` 关键字将构件分组到一个公共名称下，该关键字在所有 `install()` 指令中都可用。将安装限制到特定组件的用户可以通过运行以下命令（组件名称区分大小写）显式请求：

```
cmake --install <build tree> --component=<component name>
```

用 COMPONENT 关键字标记工件并不意味着它在默认情况下不会安装。为了防止这种情况发生，需要添加 EXCLUDE\_FROM\_ALL 关键字。

可以通过码示例来探索这些组件：

```
chapter-11/11-components/CMakeLists.txt (fragment)

...
install(TARGETS calc EXPORT CalcTargets
 ARCHIVE
 COMPONENT lib
 PUBLIC_HEADER
 DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
 COMPONENT headers
)
install(EXPORT CalcTargets
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
 NAMESPACE Calc::
 COMPONENT lib
)
install(CODE "MESSAGE(\"Installing 'extra' component\")"
 COMPONENT extra
 EXCLUDE_FROM_ALL
)
...
...
```

这些安装命令定义了以下组件：

- lib: 这包含静态库和目标导出文件。默认安装。
- headers: 包含公共头文件。默认安装。
- extra: 通过打印一条消息来执行一段代码。默认不安装。

这里重申一下：

- cmake --install 若没有--component 参数，将同时安装 lib 和 header 组件。
- cmake --install --component headers 将只安装公共头文件。
- cmake --install --component extra 将打印一个无法访问的消息 (因为有 EXCLUDE\_FROM\_ALL 关键字)。

若没有为安装的工件指定 COMPONENT 关键字，将从 CMAKE\_INSTALL\_DEFAULT\_COMPONENT\_NAME 变量中获得一个未指定的默认值。

Note

因为没有简单的方法列出 cmake 命令行中可用的所有组件，所以包的用户将从列出包组件的文档中受益，也许 INSTALL 文件是说明这些的好地方。

若对不存在的组件使用--component 参数调用 cmake，则该命令将成功执行，而不会出现任何警告或错误，就是不会安装任何东西。

将安装划分为组件使用户能够挑选想要安装的组件。我们主要讨论了将已安装的文件分组为组件，但也有一些过程步骤，如 install(SCRIPT|CODE) 或为动态库创建符号链接。

### 管理版本动态库的符号链接

安装的目标平台可能使用符号链接来帮助链接程序发现动态库的当前安装版本。创建一个 lib<name>.so 后，将其连接到 lib<name>.so.1，然后就可以通过-l<name> 来链接这个库。这种符号链接的创建在需要时由 CMake 的 install(TARGETS <target> LIBRARY) 处理。

然而，可以通过添加 NAMELINK\_SKIP 来决定将这个步骤移动到另一个 install() 指令中：

```
install(TARGETS <target> LIBRARY COMPONENT cmp
NAMELINK_SKIP)
```

要将符号链接分配给另一个组件 (而不是完全禁用)，可以为相同的目标重复 install() 指令，并指定一个不同的组件，后面跟着 NAMELINK\_ONLY 关键字：

```
install(TARGETS <target> LIBRARY COMPONENT lnk
NAMELINK_ONLY)
```

同样可以通过 NAMELINK\_COMPONENT 关键字实现：

```
install(TARGETS <target> LIBRARY
COMPONENT cmp NAMELINK_COMPONENT lnk)
```

现在已经配置了自动安装，可以使用 CMake 中包含的 CPack 工具为用户提供预构建的构件。

## 11.6. 使用 CPack 打包

从源码构建项目有它的好处，但会花费很长时间。对于只想使用包的最终用户来说，这不是最好的体验，特别是不是开发人员的情况下。软件分发的一种更方便的形式是使用二进制包，其中包含已编译的工件和运行时需要的其他静态文件。CMake 支持通过名为 cpack 的命令行工具生成多种类型的包。

下表列出了可用的包生成器：

| 名称           | 文件类型                                                                                                             | 平台                    |
|--------------|------------------------------------------------------------------------------------------------------------------|-----------------------|
| Archive      | 7Z - 7zip - (.7z)<br>TBZ2(.tar.bz2)<br>TGZ(.tar.gz)<br>TXZ(.tar.xz)<br>TZ(.tar.Z)<br>TZST(.tar.zst)<br>ZIP(.zip) | Cross-platform        |
| Bundle       | Bundle(.bundle)                                                                                                  | macOS                 |
| DEB          | DEB(.deb)                                                                                                        | Linux                 |
| DragNDrop    | DMG(.dmg)                                                                                                        | macOS                 |
| External     | JSON(.json)                                                                                                      | 与外部打包工具集成             |
| FreeBSD      | PKG(pkg)                                                                                                         | *BSD, Linux, OSX      |
| IFW          | Binary                                                                                                           | Linux, Windows, macOS |
| NSIS         | Binary(.exe)                                                                                                     | Windows               |
| NuGet        | NuGet(.nupkg)                                                                                                    | Windows               |
| productbuild | PKG(.pkg)                                                                                                        | macOS                 |
| RPM          | RPM(.rpm)                                                                                                        | Linux                 |
| WIX          | MSI(.msi)                                                                                                        | Windows               |

大多数生成器都具有丰富的配置。深入研究其细节会超出了本书的范围，所以一定要查看完整的文档，可以在扩展阅读部分找到。相反，我们会关注一般用例。

#### Note

包生成器不应该与构建系统生成器 (Unix Makefiles, Visual Studio 等) 混淆。

要使用 CPack，需要使用必要的 `install()` 指令正确配置项目的安装，并构建项目。`cpack` 将使用在构建树中生成的生成的 `cmake_install.cmake`，再基于配置文件 (`CPackConfig.cmake`) 准备二进制包。可以手动创建该文件，但使用 `include(CPack)` 将实用程序模块包含到项目的列表文件中更容易。其将在项目的构建树中生成配置，并在需要的地方提供所有默认值。

来看看如何扩展示例中的 11 个组件，使其能够与 CPack 一起工作：

```
chapter-11/12-cpack/CMakeLists.txt (fragment)

cmake_minimum_required(VERSION 3.20.0)
project(CPackPackage VERSION 1.2.3 LANGUAGES CXX)
include(GNUInstallDirs)
add_subdirectory(src bin)

install(...)
install(...)
```

```
install(...)

set(CPACK_PACKAGE_VENDOR "Rafal Swidzinski")
set(CPACK_PACKAGE_CONTACT "email@example.com")
set(CPACK_PACKAGE_DESCRIPTION "Simple Calculator")
include(CPack)
```

代码不难，所以不会过多地讨论（请参考模块文档，可以在扩展阅读部分找到）。CPack 模块将从 project() 指令中推断出一些值：

- CPACK\_PACKAGE\_NAME
- CPACK\_PACKAGE\_VERSION
- CPACK\_PACKAGE\_FILE\_NAME

最后一个值将用于生成输出包。其结构如下：

```
$CPACK_PACKAGE_NAME-$CPACK_PACKAGE_VERSION-$CPACK_SYSTEM_NAME
```

CPACK\_SYSTEM\_NAME 是目标操作系统的名称；例如，Linux 或 win32。例如，通过在 Debian 上执行 ZIP 生成器，CPack 将生成一个名为 CPackPackage-1.2.3-Linux.zip 的文件。

构建项目之后，可以通过在构建树中运行 cpack 二进制代码来生成实际的包：

```
cpack [<options>]
```

CPack 能够从当前工作目录中的配置文件中读取所有选项，但可以使用命令行覆盖这些设置：

- -G <generators>：这是一个用分号分隔的包生成器列表。默认值可以在 CPackConfig.cmake 的 CPACK\_GENERATOR 变量中指定。
- -C <configs>：这是一个分号分隔的构建配置 (Debug、Release) 列表，用于为其生成包 (多配置构建系统生成器所需)。
- -D <var>=<value>：这将使用 <value> 覆盖 CPackConfig.cmake 中设置的 <var>。
- --config <config-file>：这是应该使用的配置文件，而不是默认的 CPackConfig.cmake。
- --verbose, -V：提供详细输出。
- -P <packageName>：覆盖包名称。
- -R <packageVersion>：覆盖包版本。
- --vendor <vendorName>：覆盖包供应商。
- -B <packageDirectory>：指定 cpack 的输出目录 (默认情况下，这将是当前的工作目录)。

可以试着为 12 个包的输出生成包。我们将使用 ZIP、7Z 和 Debian 包生成器：

```
cpack -G "ZIP;7Z;DEB" -B packages
```

应该生成以下包：

- CPackPackage-1.2.3-Linux.7z
- CPackPackage-1.2.3-Linux.deb
- CPackPackage-1.2.3-Linux.zip

这种格式的二进制包可以发布到项目网站上，在 GitHub 中，或者直接发送包到最终用户的邮箱中。

## 11.7. 总结

若没有像 CMake 这样的工具，以跨平台的方式编写安装脚本是一项非常复杂的任务。虽然仍然需要一些工作来设置，但这是一个更加精简的过程，与本书中迄今为止使用的所有其他概念和技术紧密相连。

首先，我们学习了如何从项目中导出 CMake 目标，以便可以在其他项目中使用，而无需安装。然后，学习了如何安装已经为此目的配置的项目。

然后，开始从最重要的主题开始探索安装的基础：安装 CMake 目标。我们现在知道了 CMake 如何处理各种工件类型的不同目录，以及如何处理有些特殊的公共头文件。为了在较低级别上管理这些安装步骤，讨论了 `install()` 指令的各种模式，包括安装文件、程序和目录，以及在安装期间调用脚本。

解释了如何编写安装步骤之后，了解了 CMake 的可重用包。学习了如何使项目中的目标可重定位，以便包可以安装在用户想要的地方。然后，着重于使用 `find_package()` 形成一个可以让其他项目使用的包，这需要准备目标导出文件、配置文件和版本文件。

认识到不同的用户可能需要包的不同部分，了解了如何在安装组件中对构件和操作进行分组，以及与 CMake 包的组件之间的区别。最后，介绍了 CPack 并学习了如何准备基本的二进制包，这些包可用于以预编译的形式分发软件。

要完全掌握安装和打包的所有细节和复杂性还有很长的路要走，但本章为我们提供了处理最常见场景的坚实基础，并有信心进一步探索它们。

下一章中，将通过创建一个连贯的、专业的项目，将所学到的一切付诸实践。

## 11.8. 扩展阅读

更多资料，请访问以下链接：

- GNU 编码标准的目标：  
[https://www.gnu.org/prep/standards/html\\_node/Directory-Variables.html](https://www.gnu.org/prep/standards/html_node/Directory-Variables.html)
- 用 FILE\_SET 关键字管理新公共头文件的讨论：  
[https://gitlab.kitware.com/cmake/cmake/-/issues/22468#note\\_991860](https://gitlab.kitware.com/cmake/cmake/-/issues/22468#note_991860)
- 如何安装动态库：  
<https://tldp.org/HOWTO/ProgramLibrary-HOWTO/shared-libraries.html>
- 创建可重定位的包：  
<https://cmake.org/cmake/help/latest/guide/importing-exporting/index.html#creating-relocatable-packages/>

- `find_package()` 扫描配置文件的路径列表:  
[https://cmake.org/cmake/help/latest/command/find\\_package.html#config-mode-search-procedure](https://cmake.org/cmake/help/latest/command/find_package.html#config-mode-search-procedure)
- `CMakePackageConfigHelpers` 完整的文档地址:  
<https://cmake.org/cmake/help/latest/module/CMakePackageConfigHelpers.html>
- CPack 包生成器:  
<https://cmake.org/cmake/help/latest/manual/cpack-generators.7.html>
- 针对不同平台的首选包生成器:  
<https://stackoverflow.com/a/46013099>
- CPack 模块的文档:  
<https://cmake.org/cmake/help/latest/module/CPack.html>

# 第 12 章 创建完整的项目

我们收集了建造专业项目所需的所有知识，了解了结构化、构建、依赖管理、测试、分析、安装和打包。是时候通过创建一个连贯、专业的项目来将这些获得的技能付诸实践了。

重要的是要理解，即使是微不足道的程序也可受益于自动化的质量检查和简化的端到端流程，将原始代码转化为完整的解决方案。这通常是一项相当大的投资，需要许多步骤来准备这一切——若试图将这些机制添加到现有的代码库中（通常，已经非常庞大和复杂），则需要采取更多步骤。

这就是从一开始就使用 CMake 并在前面对所有流水进行设置的原因，不仅更容易配置，也能更有效地尽早进行测试，因为所有的质量控制和自动化构建都必须添加到长期项目中。

这正是在本章要做的——编写一个尽可能小而简单的新解决方案。只执行一个（几乎）功能——两个数字相加，限制业务代码的功能可以让我们更专注于项目的其他方面。

为了解决一个更复杂的问题，这个项目将构建一个库和一个可执行文件。该库将提供内部业务逻辑，也可以供其他项目作为 CMake 包使用。可执行文件将只针对最终用户，并将实现一个显示底层库功能的用户界面。

本章中，我们将讨论以下主题：

- 规划工作
- 项目布局
- 构建和管理依赖项
- 测试和程序分析
- 安装和打包
- 提供文档

## 12.1. 相关准备

可以在 GitHub 上找到本章中出现的代码 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp/tree/main/examples/chapter12>。

构建本书中提供的示例，推荐的命令：

```
cmake -B <build tree> -S <source tree>
cmake --build <build tree>
```

使用适当的路径替换 `<build tree>` 和 `<source tree>`。注意：构建树是目标/输出目录的路径，源代码树是源码所在的路径。

## 12.2. 规划工作

本章中构建的软件并不复杂——我们将创建一个简单的计算器，将两个数字相加（图 12.1）。作为一个控制台应用程序发布，带有文本用户界面和一个用于执行数学运算的库，这可能用于其他项目。虽然在现实中没有太多用处，但它将用于探索本书中讨论的所有技术，以及如何在实践中进行协同工作：

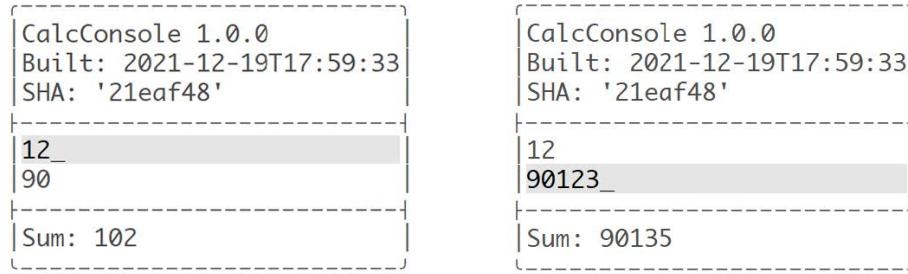


图 12.1 控制台计算器用户界面的两种状态

通常，项目要么生成面向用户的可执行文件，要么为开发人员生成库。同时具备这两种功能的项目很少见——一些应用程序提供独立的 SDK 或库来支持插件，另一种情况是提供其用法示例的库。我们在本章中构建的项目在某种程度上属于最后一类。

这里先回顾一下之前章节的内容，并选择相应的技术和工具来构建应用：

### 第 1 章:

提供了关于 CMake 的基本信息——如何安装它，并使用命令行来构建项目。这里提供的有关项目文件的信息是关键：不同文件的职责、常规使用的名称，以及一些奇淫技巧。本章中，还讨论了生成器的预设文件，但在本项目中先跳过这些文件。

### 第 2 章:

介绍了编写正确的列表文件和脚本所需的工具，分享了关于代码的基本信息：注释、命令调用和参数。我们还详细解释了变量、列表和控制结构，并给出了一些非常有用的知识。这些知识将应用于整个项目。

### 第 3 章:

这里的内容将对项目产生关键性影响：

- 指定最低的 CMake 版本将决定应用哪些 CMake 策略，命名、版本控制和配置项目语言会影响构建的基本行为。
- 了解形成目录和文件布局的项目分区和结构。
- 系统可以帮助我们处理不同的环境，特别是对于这个项目——例如，是否需要运行 ldconfig？
- 工具链配置允许 C++ 和编译器支持特定的标准需求。

本章还说明了禁用源内构建的原因。

### 第 4 章:

这里，强调了每个现代 CMake 项目如何使用目标，原因如下：

- 定义一些库和可执行程序（用于测试和生产）将使项目保持有组织和 DRY。
- 目标属性和传递性使用需求（传播属性）保持配置接近目标定义。
- 生成器表达式将在整个解决方案中出现，使它们尽可能简单。

这个项目中，将使用自定义命令为 Valgrind 和覆盖率报告生成文件，并且使用目标钩子（PRE\_BUILD），来清理覆盖率工具生成的.gcda 文件。

### 第 5 章:

没有编译就没有 C++ 项目，CMake 可以以多种方式调整这个过程：扩展目标的源、配置优化器和提供调试信息。对于这个项目，默认的编译标志就可以了，但我们将继续使用预处理器：

- 在编译后的可执行文件中存储构建元数据（项目版本、构建时间和 Git 提交 SHA），并将其显示给用户。
- 将启用头文件的预编译。在如此小的项目中，这并不是必须的，但会帮助我们实践这个知识点。

统一构建不是必要的——这个项目不够大，但也可以添加。

## 第 6 章：

提供了关于链接的信息（在任何项目中都很有用），其中大部分在默认情况下很有用。但是由于这个项目也提供了一个库，将显式地引用以下的一些构建说明：

- 用于测试和开发的静态库
- 要发布的动态库

本章概述了如何分离 main() 进行测试，我们的项目中也有测试。

## 第 7 章：

为了使项目更有趣，将引入一个外部依赖项：文本 UI 库。本章中描述了一些依赖管理方法。选择正确的一个并不太难：FetchContent 实用程序模块通常是推荐的，也是最方便的（除非解决本章中描述的特定情况）。

## 第 8 章：

必须有适当的自动化测试，以确保解决方案的质量不会随着时间而下降。我们将添加对 CTest 的支持，并为测试正确地构造项目（将应用前面提到的 main() 分离）。

此外，还讨论了两个测试框架：Catch2 和 GTest；对于这个项目，我们将使用后者。为了获得关于覆盖率的明确信息，将生成带有 LCOV 的 HTML 报告。

## 第 9 章：

要执行静态分析，可以从各种各样的工具中进行选择：Clang-Tidy、Cpplint、Cppcheck、include-what-you-use 和连接其他工具。本例中，将使用 Cppcheck，因为 Clang-Tidy 不能很好地处理用 GCC 编译的预编译头文件。动态分析将使用 Valgrind 的 Memcheck 工具完成，将使用 Memcheck 封面包装器生成 HTML 报告，源代码也将在编译过程中使用 ClangFormat 自动格式化。

## 第 10 章：

因为提供一个库作为这个项目的一部分，所以要提供一些与之配套的文档。CMake 可以用 Doxygen 自动生成，所以可以通过添加 doxygenawesome-css 来进行更新文档外观。

## 第 11 章：

最后，配置解决方案的安装和打包，将按照描述准备文件以形成包，以及目标定义。我们将通过包含 GNUInstallDirs 模块，将其和构件从构建目标安装到适当的目录中。我们将另外配置一些组件模块化，并与 CPack 一起使用。

专业项目还附带一些文本文件：README、LICENSE、INSTALL 等。我们会在最后简要地讨论这个问题。

### Note

为了使事情更简单，不会实现检查所有必需的实用程序和依赖项是否可用的逻辑。我们将在这里依赖 CMake 来显示诊断，并告诉用户缺少什么。若在阅读本书后发布的项目获得了其他人的关注，可能就需要考虑添加这些机制来改善用户体验了。

形成了一个清晰的计划之后，再从逻辑目标和目录结构两方面讨论实际的构建项目。

## 12.3. 项目布局

要构建项目，应该从清楚地了解将在其中创建什么逻辑目标开始。本例中，将遵循图 12.2 所示的结构：

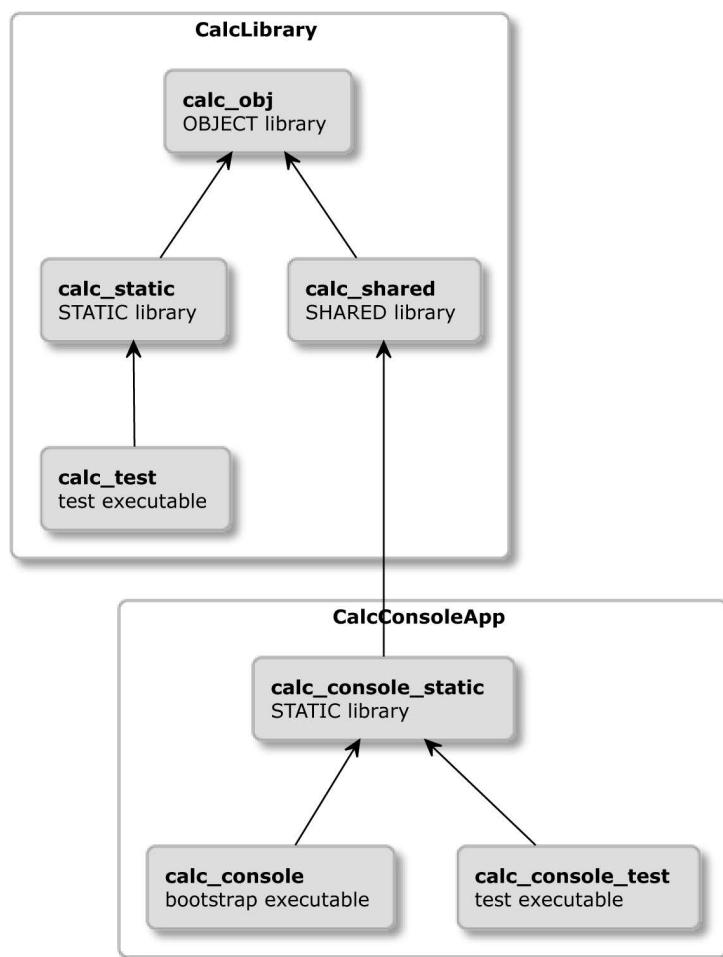


图 12.2 逻辑目标的结构

我们按照构建顺序来探索这个结构。首先，编译 calc\_obj，这是一个对象库。我们确实在书中几次提到了对象库，但并没有作为一个知识点来介绍。

### 12.3.1 对象库

对象库用于将多个源文件分组到单个逻辑目标下，并在构建期间编译到 (.o) 目标文件中。要创建一个对象库，使用 OBJECT 关键字与其他关键字相同：

```
add_library(<target> OBJECT <sources>)
```

构建过程中产生的对象文件可以通过 \$<TARGET\_OBJECTS:objlib> 生成器表达式作为编译后的元素添加到其他目标:

```
add_library(... $<TARGET_OBJECTS:objlib> ...)
add_executable(... $<TARGET_OBJECTS:objlib> ...)
```

或者，可以使用 `target_link_libraries()` 将其作为依赖项添加。

Calc 库中，对象库将有助于避免为库的静态和动态版本重复编译库源。只需要记住显式地使用 `POSITION_INDEPENDENT_CODE` 编译目标文件，因为这是动态库的需求。

解决了这些问题，回到这个项目的目标:`calc_obj` 将提供编译的目标文件，然后将用于 `calc_static` 和 `calc_shared` 库。它们之间的实际区别是什么？为什么要提供两个库？

### 12.3.2 比较动态库与静态库

第 6 章中简要介绍了这两种类型的库，对于使用相同动态库的多个程序，整体内存使用可能会更好，而且用户很可能已经拥有最流行的库，或者知道如何快速安装。更重要的是，动态库是在单独的文件中交付的，这些文件必须安装在特定的路径中，以便动态链接器找到，而静态库是作为可执行文件的一部分，因为不需要在内存中查找代码的位置，所以静态库使用起来稍微快一些。

作为库作者，可以决定是提供库的静态还是动态版本，或者可以简单地提供两个版本，并把这个决定留给使用库的开发者。我们在这里选择后者（只是为了了解如何完成）。

静态库将由 `calc_test` 目标使用，该目标将包含单元测试，以确保所提供的库业务功能按预期工作。我们从同一组已编译的目标文件构建两个版本，测试任何一个都可以，因为其行为没有区别。

由 `calc_console_static` 目标提供的控制台应用程序将使用动态库。该目标还将链接到一个外部依赖:Arthur Sonzogni 的功能终端用户界面 (FTXUI) 库（在扩展阅读部分有一个到 GitHub 项目的链接）。它为文本用户界面提供了一个无依赖性、跨平台的框架。

最后两个目标是 `calc_console` 和 `calc_console_test`。`calc_console` 目标只是一个围绕 `calc_console_static` 的 `bootstrap main()` 包装器，其目的是从业务代码中提取入口点。可以为其编写单元测试（需要提供接口），并使用 `calc_console_test` 运行。

现在知道需要建立什么目标，以及它们之间的关系。来看看如何用文件和目录来组织项目。

### 12.3.3 目录结构

该项目包含两个主要目标，`calc` 库和 `calc_console` 可执行文件，每个目标都位于 `src` 和 `test` 下的目录树中，以将生产代码与测试代码分离（如图 12.3 所示）。此外，文件放在另外两个目录中：

- 包含顶层配置和关键项目文档文件的根目录
- `cmake` 目录中包含用来构建和安装项目的所有工具模块和帮助文件：

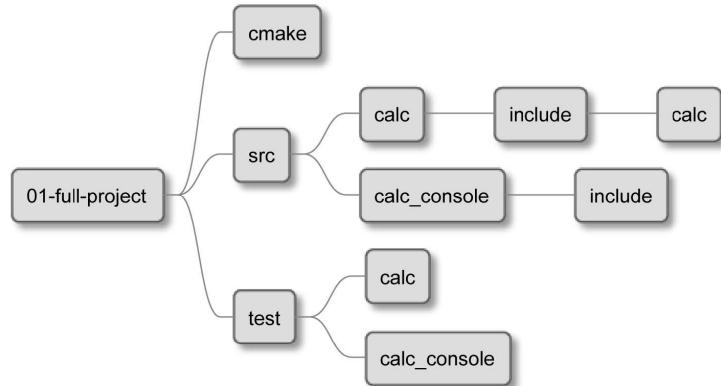


图 12.3 项目的目录结构

以下是四个主目录下文件的完整列表:

| <code>./</code>                                                                                                                                                                                       | <code>./test</code>                                                                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CHANGELOG<br>CMakeLists.txt<br>CalcConfig.cmake<br>INSTALL<br>LICENSE<br>README.md                                                                                                                    | CMakeLists.txt<br>calc/CMakeLists.txt<br>calc/calc_test.cpp<br>calc_console/CMakeLists.txt<br>calc_console/tui_test.cpp                                                                                |
| <code>./src</code>                                                                                                                                                                                    | <code>/cmake</code>                                                                                                                                                                                    |
| CMakeLists.txt<br>calc/CMakeLists.txt<br>calc/calc.cpp<br>calc/include/calc/calc.h<br>calc_console/CMakeLists.txt<br>calc_console/bootstrap.cpp<br>calc_console/include/tui.h<br>calc_console/tui.cpp | buildinfo.h.in<br>BuildInfo.cmake<br>Coverage.cmake<br>CppCheck.cmkae<br>Doxygen.cmake<br>Format.cmake<br>GetFTXUI.cmake<br>Install.cmake<br>Memcheck.cmake<br>NoInSourceBuilds.cmake<br>Testing.cmake |

最初，`cmake` 目录比业务代码更多，但随着项目功能的增长，这种情况将很快发生变化。启动一个清洁项目的努力是很重要的，但不要担心——很快就会得到回报。

我们将浏览所有文件，详细了解它们的作用以及它们在项目中扮演的角色。这将分为四个步骤：构建、测试、安装和生成文档。

## 12.4. 构建和管理依赖项

所有构建流程都以相同的方式工作。从顶层列表文件开始，向下导航到项目源树。图 12.4 显示了参与构建的项目文件。括号中的数字表示 CMake 脚本执行的顺序：

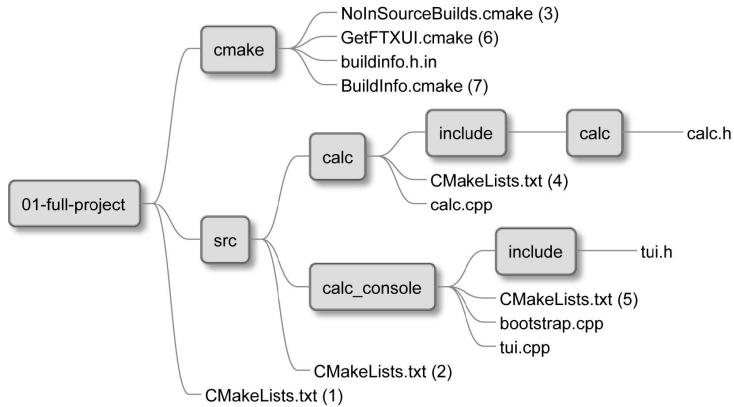


图 12.4 构建阶段使用的文件

顶层列表文件将配置项目，并加载嵌套元素：

```
chapter-12/01-full-project/CMakeLists.txt

cmake_minimum_required(VERSION 3.20.0)
project(Calc VERSION 1.0.0 LANGUAGES CXX)
list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

include(NoInSourceBuilds)

add_subdirectory(src bin)
add_subdirectory(test)

include(Install)
```

首先提供关键的项目细节，并添加到 CMake 实用程序模块的路径（项目中的 CMake 目录）。然后禁用源内构建（通过自定义模块），并包含两个关键目录：

- src，包含项目源（构建树中命名为 bin）
- test，包含所有测试实用程序

最后，将包含另一个模块，该模块将设置项目的安装。来看一看 NoInSourceBuilds 模块，了解它是如何工作的：

```
chapter-12/01-full-project/cmake/NoInSourceBuilds.cmake

if(PROJECT_SOURCE_DIR STREQUAL PROJECT_BINARY_DIR)
 message(FATAL_ERROR
```

```

"\n"
"In-source builds are not allowed.\n"
"Instead, provide a path to build tree like so:\n"
"cmake -B <destination>\n"
"\n"
"To remove files you accidentally created execute:\n"
"rm -rf CMakeFiles CMakeCache.txt\n"
)
endif()

```

这里没有什么奇怪的——只是检查用户是否提供了一个目标目录作为 `cmake` 命令的参数来存储生成的文件，必须是与项目源树不同的路径。若不是这样，会通知用户如何提供，以及在错误发生后如何清理存储库。

顶层列表文件然后包含 `src` 子目录，指示 CMake 读取其中的列表文件：

```

chapter-12/01-full-project/src/CMakeLists.txt

add_subdirectory(calc)
add_subdirectory(calc_console)

```

这个文件非常微妙——只是进入嵌套目录，执行其中的列表文件。让我们跟随 `calc` 库的列表文件——有点复杂，所以只讨论其中一部分。

#### 12.4.1 构建 Calc 库

`calc` 的列表文件包含一些测试配置，但现在将关注构建；其余的将在测试和程序分析部分讨论：

```

chapter-12/01-full-project/src/calc/CMakeLists.txt (fragment)

add_library(calc_obj OBJECT calc.cpp)
target_include_directories(calc_obj INTERFACE
 ${<BUILD_INTERFACE>: ${CMAKE_CURRENT_SOURCE_DIR}/include}>
 ${<INSTALL_INTERFACE>: ${CMAKE_INSTALL_INCLUDEDIR}}>
)
set_target_properties(calc_obj PROPERTIES
 PUBLIC_HEADER src/calc/include/calc/calc.h
 POSITION_INDEPENDENT_CODE 1
)
add_library(calc_shared SHARED)
target_link_libraries(calc_shared calc_obj)
add_library(calc_static STATIC)
target_link_libraries(calc_static calc_obj)

```

```
... testing and program analysis modules
... documentation generation
```

声明三个目标:

- 编译 calc.cpp 实现文件的对象库，并通过 PUBLIC\_HEADER 属性引用 calc.h 头文件，该属性可以在配置的 include 目录中找到（由于生成器表达式为特定模式提供了适当的路径——构建或安装）。通过使用这个库，避免了对其他目标的重复编译，还要启用 POSITION\_INDEPENDENT\_CODE，以便动态库可以使用生成的目标文件。
- calc\_shared，依赖于 calc\_obj 的动态库。
- calc\_static，依赖于 calc\_obj 的静态库。

这里将添加 calc 库的 C++ 代码:

```
1 // chapter-12/01-full-project/src/calc/include/calc/calc.h
2
3 #pragma once
4
5 namespace Calc {
6 int Sum(int a, int b);
7 int Multiply(int a, int b);
8 } // namespace Calc
```

这段代码非常基础：声明了两个包含在 Calc 命名空间中的全局函数（C++ 命名空间在库中非常有用，有助于避免名称冲突）。

实现文件也非常简单：

```
1 // chapter-12/01-full-project/src/calc/calc.cpp
2
3 namespace Calc {
4 int Sum(int a, int b) {
5 return a + b;
6 }
7
8 int Multiply(int a, int b) {
9 return a * b;
10 }
11 } // namespace Calc
```

这结束了 src/calc 目录中文件的解释。接下来是 src/calc\_console，并使用这个库构建控制台计算器的可执行文件。

#### 12.4.2 构建 Calc 终端可执行文件

calc\_console 的源目录包含几个文件：一个列表文件、两个实现文件（业务代码和一个引导程序）和一个头文件。列表文件如下所示：

```
chapter-12/01-full-project/src/calc_console/CMakeLists.txt (fragment)
include (GetFTXUI)
```

```

add_library(calc_console_static STATIC tui.cpp)
target_include_directories(calc_console_static PUBLIC
 include)
target_precompile_headers(calc_console_static PUBLIC
 <string>)
target_link_libraries(calc_console_static PUBLIC
 calc_shared
 ftxui::screen ftxui::dom ftxui::component)
include(BuildInfo)
BuildInfo(calc_console_static)
... testing and program analysis modules
... documentation generation

add_executable(calc_console bootstrap.cpp)
target_link_libraries(calc_console calc_console_static)

```

列表文件看起来很多，但作为有经验的 CMake 用户，可以很容易地理清里面发生的事情：

- 包含 CMake 模块来获取 FTXUI 依赖
- 声明 calc\_console\_static 目标，该目标包含业务代码，但不包含 main() 函数，以允许 GTest 定义自己的入口点。
- 添加头文件的预编译——只是添加一个标准的字符串头文件来证明这一点，但是对于较大的项目，可以添加更多的头文件（包括属于项目的头文件）。
- 将业务代码与 calc\_shared 库和 FTXUI 库链接起来。
- 添加要在这个目标上执行的所有操作：生成构建信息、测试、程序分析和文档。
- 添加并链接 calc\_console 引导可执行文件，并提供了入口点。

同样，这里把测试和文档的讨论推迟到本章的后续部分。来看看依赖管理和构建信息生成。

注意，我们更喜欢使用实用程序模块，而不是查找模块来引入 FTXUI。这是因为该依赖项不太可能已经存在于系统中。而不是希望找到它，我们将获取并安装它：

```

chapter-12/01-full-project/cmake/GetFTXUI.cmake

include(FetchContent)
FetchContent_Declare(
 FTXTUI
 GIT_REPOSITORY https://github.com/ArthurSonzogni/FTXUI.git
 GIT_TAG v0.11
)
option(FTXUI_ENABLE_INSTALL "" OFF)
option(FTXUI_BUILD_EXAMPLES "" OFF)
option(FTXUI_BUILD_DOCS "" OFF)

```

```
FetchContent_MakeAvailable(FTXTUI)
```

使用推荐的 FetchContent 方法，第 7 章有详细介绍，唯一不同的添加是对 option() 指令的调用。其可以跳过 FTXUI 构建的冗长步骤，并将其安装配置从该项目的安装中分离出来。对于 GTest 依赖关系也需要同样的操作。option() 指令在扩展阅读部分中有引用。

calc\_command 的列表文件还包含一个与构建相关的自定义实用程序模块:BuildInfo。我们将使用它来记录三个可以在可执行文件中显示的值:

- 当前 Git 提交的 SHA
- 构建的时间戳
- 顶层列表文件中指定的项目版本

在第 5 章中，可以使用 CMake 来捕获一些构建时的值，并通过模板文件将其提供给 C++ 代码——例如，使用 C++ 结构体:

```
1 // chapter-12/01-full-project/cmake/buildinfo.h.in
2
3 struct BuildInfo {
4 static inline const std::string CommitSHA =
5 "@COMMIT_SHA@";
6 static inline const std::string Timestamp =
7 "@TIMESTAMP@";
8 static inline const
9 std::string Version = "@PROJECT_VERSION@";
10};
```

为了在配置阶段填充该结构，可以使用以下代码:

```
chapter-12/01-full-project/cmake/BuildInfo.cmake

set(BUILDINFO_TEMPLATE_DIR ${CMAKE_CURRENT_LIST_DIR})
set(DESTINATION "${CMAKE_CURRENT_BINARY_DIR}/buildinfo")

string(TIMESTAMP TIMESTAMP)
find_program(GIT_PATH git REQUIRED)
execute_process(COMMAND
 ${GIT_PATH} log --pretty=format:'%h' -n 1
 OUTPUT_VARIABLE COMMIT_SHA)

configure_file(
 "${BUILDINFO_TEMPLATE_DIR}/buildinfo.h.in"
 "${DESTINATION}/buildinfo.h" @ONLY
)

function(BuildInfo target)
```

```

target_include_directories(${target} PRIVATE
${DESTINATION})
endfunction()

```

包含该模块将设置包含所需要的信息的变量，然后使用 `configure_file()` 来生成 `buildinfo.h`。剩下的就是调用 `BuildInfo` 函数，并添加生成文件的目录，以包含所需目标的目录。然后，可以与多个不同的使用者共享该文件，并且可能需要在列表文件的顶部添加 `include_guard(GLOBAL)`，以避免对每个目标都运行 `git` 命令。

深入研究控制台计算器的实现之前，不应该过分担心 `tui.cpp` 文件的复杂性。要完全理解它，只需要对 `FXTUI` 库有一定的了解——我们不想太深入。相反，让我们关注突出显示的行：

```

1 // chapter-12/01-full-project/src/calc_console/tui.cpp
2
3 #include "tui.h"
4 #include <ftxui/dom/elements.hpp>
5 #include "buildinfo.h" // 突出显示的行
6 #include "calc/calc.h" // 突出显示的行
7
8 using namespace ftxui;
9 using namespace std;
10
11 string a{"12"}, b{"90"};
12 auto input_a = Input(&a, "");
13 auto input_b = Input(&b, "");
14 auto component = Container::Vertical({input_a, input_b});
15
16 Component getTui() {
17 return Renderer(component, [&] {
18 auto sum = Calc::Sum(stoi(a), stoi(b)); // 突出显示的行
19 return vbox({
20 text("CalcConsole " + BuildInfo::Version), // 突出显示的行
21 text("Built: " + BuildInfo::Timestamp), // 突出显示的行
22 text("SHA: " + BuildInfo::CommitSHA), // 突出显示的行
23 separator(),
24 input_a->Render(),
25 input_b->Render(),
26 separator(),
27 text("Sum: " + to_string(sum)), // 突出显示的行
28 }) |
29 border;
30 });
31 }

```

这段代码提供了一个 `getTui()` 函数，返回一个 `ftxui::Component` 对象，该对象封装了一个带有标签、文本字段、分隔符和边框的交互式 UI 元素。若对它的详细工作原理感兴趣，可以在扩展阅读部分找到合适的参考资料。

更重要的是，了解一下 `include` 指令：其引用了用 `calc_obj` 目标和 `BuildInfo` 模块提供的头文件。

提供给 Renderer 类构造函数的 lambda 函数的第一行将调用库的 Calc::Sum 方法，并使用结果值打印带有 Sum 的标签（通过调用下面的 text() 函数）。

类似地，标签用于向用户显示在构建时，通过连续三次调用 text() 来收集 BuildInfo::values。

这个方法在相关的头文件中有声明：

```
1 // chapter-12/01-full-project/src/calc_console/include/tui.h
2
3 #include <ftxui/component/component.hpp>
4 ftxui::Component getTui();
```

然后，被 calc\_console 目标使用：

```
1 // chapter-12/01-full-project/src/calc_console/bootstrap.cpp
2
3 #include <ftxui/component/screen_interactive.hpp>
4 #include "tui.h"
5
6 int main(int argc, char** argv) {
7 ftxui::ScreenInteractive::FitComponent().Loop(getTui());
8 }
```

这段简短的代码利用 ftxui 创建一个交互式控制台屏幕，该屏幕接受 getTui() 返回的 Component 对象，使其对用户可见，并在循环中收集键盘事件，创建一个界面，如图 12.1 所示。同样，理解这一点并不真正重要，因为 ftxui 的主要目的是为我们提供一个外部依赖项，重要的是其可以使用它来练习 CMake 技术。

已经介绍了 src 目录中的所有文件。接下来，来看看测试和分析程序。

## 12.5. 测试和程序分析

程序分析和测试一起进行，以确保解决方案的质量。当使用测试代码时，运行 Valgrind 会变得更加一致。出于这个原因，可以将这两个东西配置在一起。图 12.5 演示了执行流程和设置它们所需的文件（一些代码片段将添加到 src 目录）：

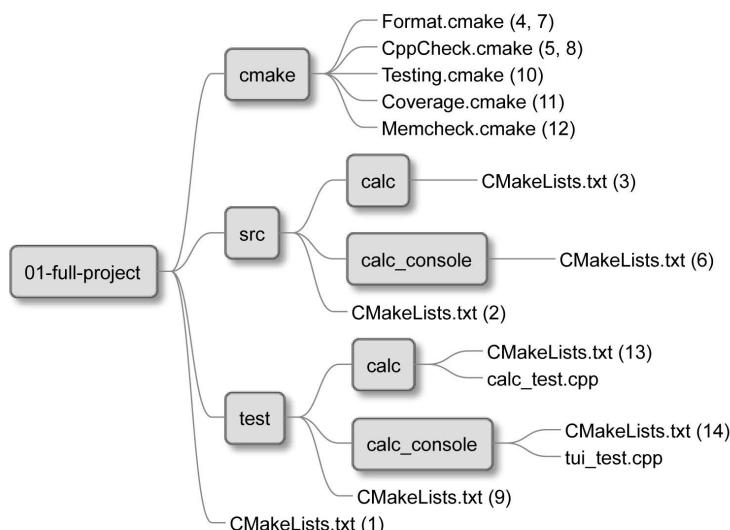


图 12.5 启动测试和程序分析

测试位于 test 目录中，其列表文件通过 add\_subdirectory() 指令从顶层列表文件中执行。来看看里面有什么：

```
chapter-12/01-full-project/test/CMakeLists.txt

include(Testing)
add_subdirectory(calc)
add_subdirectory(calc_console)
```

Testing 模块中定义的测试实用程序包含在这个级别，以允许两个目标组（来自 calc 和 calc\_console 目录）使用：

```
chapter-12/01-full-project/cmake/Testing.cmake (fragment)

enable_testing()
include(FetchContent)
FetchContent_Declare(
 googletest
 GIT_REPOSITORY https://github.com/google/googletest.git
 GIT_TAG release-1.11.0
)
For Windows: Prevent overriding the parent project's
compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
option(INSTALL_GMOCK "Install GMock" OFF)
option(INSTALL_GTEST "Install GTest" OFF)
FetchContent_MakeAvailable(googletest)
...
```

启用了测试并包含了 FetchContent 模块来获取 GTest 和 GMock，这个项目中并没有真正使用 GMock，但是这两个框架会捆绑在一个库中，因此还需要配置 GMock。该配置中突出显示的部分将这两个框架的安装从我们的项目安装中分离出来（将适当的 option() 设置为 OFF）。

接下来，需要创建一个功能，支持对业务目标进行彻底的测试。可以将其保存在同一个文件中：

```
chapter-12/01-full-project/cmake/Testing.cmake (continued)

...
include(GoogleTest)
include(Coverage)
include(Memcheck)

macro(AddTests target)
```

```

target_link_libraries(${target} PRIVATE gtest_main gmock)
gtest_discover_tests(${target})
AddCoverage(${target})
AddMemcheck(${target})
endmacro()

```

首先包含了必要的模块:GoogleTest 与 CMake 捆绑在一起, 但 Coverage 和 Memcheck 由我们编写。然后, 提供一个 AddTests 宏, 其为测试、代码覆盖和内存检查准备一个目标。让我们详细了解其是如何工作的。

### 12.5.1 准备覆盖率模块

添加覆盖多个目标有点棘手, 因为其包含几个步骤。首先引入两个函数, 用于在构建之间启用覆盖率跟踪和清理过时的跟踪文件:

```

chapter-12/01-full-project/cmake/Coverage.cmake (fragment)

function(EnableCoverage target)
 if (CMAKE_BUILD_TYPE STREQUAL Debug)
 target_compile_options(${target} PRIVATE --coverage
 -fno-inline)
 target_link_options(${target} PUBLIC --coverage)
 endif()
endfunction()

function(CleanCoverage target)
 add_custom_command(TARGET ${target} PRE_BUILD COMMAND
 find ${CMAKE_BINARY_DIR} -type f
 -name '*.gcda' -exec rm {} +
 endfunction()

```

当得到单独的目标配置 (calc\_... 和 calc\_console\_...), 前面的函数将在后面使用。Coverage 模块会提供一个生成自定义覆盖目标的函数:

```

chapter-12/01-full-project/cmake/Coverage.cmake (continued)

function(AddCoverage target)
 find_program(LCOV_PATH lcov REQUIRED)
 find_program(GENHTML_PATH genhtml REQUIRED)
 add_custom_target(coverage-${target}
 COMMAND ${LCOV_PATH} -d . --zerocounters
 COMMAND $<TARGET_FILE:$target>
 COMMAND ${LCOV_PATH} -d . --capture -o coverage.info

```

```

COMMAND ${LCOV_PATH} -r coverage.info '/usr/include/*'
 -o filtered.info
COMMAND ${GENHTML_PATH} -o coverage-${target}
 filtered.info --legend
COMMAND rm -rf coverage.info filtered.info
WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
)
endfunction()

```

AddCoverage() 在测试模块中的 AddTests() 函数中调用。它与第 8 章中介绍的略有不同，因为其考虑了目标的名称，并将其添加到输出路径，以避免冲突。

要为两个测试目标生成报告，只需要运行两个 cmake 命令 (在 Debug 构建类型配置项目之后):

```

cmake --build <build-tree> -t coverage-calc_test
cmake --build <build-tree> -t coverage-calc_console_test

```

现在是时候修改前面创建的 Memcheck 模块了 (第 9 章)，以处理多个目标。

### 12.5.2 准备 Memcheck 模块

AddTests() 使用 Valgrind 内存管理报告的生成，这里从这个模块设置开始:

```

chapter-12/01-full-project/cmake/Memcheck.cmake (fragment)

include(FetchContent)
FetchContent_Declare(
 memcheck-cover
 GIT_REPOSITORY https://github.com/Farigh/memcheckcover.git
 GIT_TAG release-1.2
)
FetchContent_MakeAvailable(memcheck-cover)

```

我们已经熟悉了这段代码，来看看如何为报表生成创建适当的目标函数:

```

chapter-12/01-full-project/cmake/Memcheck.cmake (continued)

function(AddMemcheck target)
 set(MEMCHECK_PATH ${memcheck-cover_SOURCE_DIR}/bin)
 set(REPORT_PATH "${CMAKE_BINARY_DIR}/valgrind-${target}")

 add_custom_target(memcheck-${target}
 COMMAND ${MEMCHECK_PATH}/memcheck_runner.sh -o

```

```

"${REPORT_PATH}/report"
-- ${TARGET_FILE}:${target}
COMMAND ${MEMCHECK_PATH}/generate_html_report.sh
-i ${REPORT_PATH}
-o ${REPORT_PATH}
WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
)
endfunction()

```

为了处理多个目标，`REPORT_PATH` 变量可置为存储特定于目标的报表的路径，并在后续的命令中使用该变量。

生成 Memcheck 报告可以通过以下命令实现 (Debug 构建类型中可以工作得更好):

```

cmake --build <build-tree> -t memcheck-calc_test
cmake --build <build-tree> -t memcheck-calc_console_test

```

这些都是 Testing 模块使用的所有模块，来看看这如何使用。

### 12.5.3 应用测试场景

要让测试起作用，需要做几件事:

1. 需要创建嵌套的列表文件，并为两个目录定义测试目标。
2. 单元测试需要编写并准备为可执行目标。
3. 需要对这些目标调用 `AddTests()`。
4. 测试软件 (SUT) 需要启用覆盖收集。
5. 收集的覆盖率应该在构建之间进行清理，以避免段错误。

正如 `test/CMakeLists.txt` 中所示，将创建两个嵌套的列表文件来配置我们的测试。再一次，将为库提供一个覆盖率测试:

```

chapter-12/01-full-project/test/calc/CMakeLists.txt (fragment)

add_executable(calc_test calc_test.cpp)
target_link_libraries(calc_test PRIVATE calc_static)
AddTests(calc_test)
EnableCoverage(calc_obj)

```

还将为可执行文件提供一个覆盖率测试:

```

chapter-12/01-full-project/test/calc_console/CMakeLists.txt (fragment)
add_executable(calc_console_test tui_test.cpp)
target_link_libraries(calc_console_test
PRIVATE calc_console_static)

```

```
AddTests(calc_console_test)
EnableCoverage(calc_console_static)
```

简单起见，可以提供尽可能简单的单元测试。一个文件将覆盖这个库：

```
// chapter-12/01-full-project/test/calc/calc_test.cpp

#include "calc/calc.h"
#include <gtest/gtest.h>

TEST(CalcTest, SumAddsTwoInts) {
 EXPECT_EQ(4, Calc::Sum(2, 2));
}

TEST(CalcTest, MultiplyMultipliesTwoInts) {
 EXPECT_EQ(12, Calc::Multiply(3, 4));
}
```

我们将有第二个文件来测试业务代码。为此，将使用 FXTUI 库。同样，并不期望您能够理解这段源码的所有细节。本章提供测试清单只是为了项目的完整性：

```
// chapter-12/01-full-project/test/calc_console/tui_test.cpp

#include "tui.h"
#include <gmock/gmock.h>
#include <gtest/gtest.h>
#include <ftxui/screen/screen.hpp>
using namespace ::ftxui;

TEST(ConsoleCalcTest, RunWorksWithDefaultValues) {
 auto component = getTui();
 auto document = component->Render();
 auto screen = Screen::Create(Dimension::Fit(document));
 Render(screen, document);
 auto output = screen.ToString();
 ASSERT_THAT(output, testing::HasSubstr("Sum: 102"));
}
```

这段测试代码只是将默认状态下的文本 UI 呈现给静态屏幕对象，然后将其存储在字符串中。为了让测试通过，输出需要包含一个带有默认和的子字符串。

现在，需要完成剩下的步骤：创建测试目标并准备好源码之后，使用 Testing 模块中的 AddTests() 函数将其注册到 CTest 中。

我们对库这样做：

```
chapter-12/01-full-project/test/calc/CMakeLists.txt (continued)

... calc_test target definition
AddTests(calc_test)
EnableCoverage(calc_obj)
```

然后为可执行文件这样做:

```
chapter-12/01-full-project/test/calc_console/CMakeLists.txt (continued)

... calc_console_test target definition
AddTests(calc_console_test)
EnableCoverage(calc_console_static)
```

随后, SUT 使用 EnableCoverage() 启用覆盖检测。注意, 在库的情况下, 必须向对象库添加工具, 而不是静态库。这是因为--coverage 标志必须添加到编译步骤中, 这是在构建 calc\_obj 时发生的。不过, 不能在这里添加对覆盖文件的清理, 因为 CMake 要求在与目标定义相同的目录中使用 add\_custom\_command 钩子。

这将我们带回到之前没有完成的 src/calc 和 src/calc\_console 列表文件。需要添加 CleanCoverage(calc\_static) 和 CleanCoverage(calc\_console\_static)(必须首先包含 Coverage 模块)。还需要向这些文件添加什么? 启用静态分析!

#### 12.5.4 添加静态分析工具

我们将业务代码列表文件的延续推迟到现在, 以便能够在适当的上下文中讨论添加的模块。可以在库列表文件中添加 CleanCoverage 函数和其他东西:

```
chapter-12/01-full-project/src/calc/CMakeLists.txt (continued)

... calc_static target definition
include(Coverage)
CleanCoverage(calc_static)
include(Format)
Format(calc_static .)
include(CppCheck)
AddCppCheck(calc_obj)
... documentation generation
```

还可以将它们添加到可执行文件中:

```
chapter-12/01-full-project/src/calc_console/CMakeLists.cmake (continued)

... calc_console_static target definition
include(BuildInfo)
BuildInfo(calc_console_static)
include(Coverage)
CleanCoverage(calc_console_static)
include(Format)
Format(calc_console_static .)
```

```

include(CppCheck)
AddCppCheck(calc_console_static)
... documentation generation
... calc_console bootstrap target definition

```

这些文件现在几乎已经完成了(正如第二条注释的建议,仍然需要添加文档代码,这将在自动文档生成部分进行)。

清单中出现了两个新模块:Format 和 CppCheck。来看看第一个问题:

```

chapter-12/01-full-project/cmake/Format.cmake

function(Format target directory)
 find_program(CLANG-FORMAT_PATH clang-format REQUIRED)
 set(EXPRESSION h hpp hh c cc cxx cpp)
 list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")
 file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
 LIST_DIRECTORIES false ${EXPRESSION})
)
add_custom_command(TARGET ${target} PRE_BUILD COMMAND
 ${CLANG-FORMAT_PATH} -i --style=file ${SOURCE_FILES}
)
endfunction()

```

Format() 函数与第 9 章中描述的格式化函数完全相同,只是在这里重用它。

接下来是全新的 CppCheck 模块:

```

chapter-12/01-full-project/cmake/CppCheck.cmake

function(AddCppCheck target)
 find_program(CPPCHECK_PATH cppcheck REQUIRED)
 set_target_properties(${target}
 PROPERTIES CXX_CPPCHECK
 "${CPPCHECK_PATH};--enable=warning;--error-exitcode=10"
)
endfunction()

```

这既简单又方便,会看到一些类似于 Clang-Tidy 模块(第 9 章)的地方。这就是 CMake 的优势——许多概念以相同的方式工作。注意传递给 cppcheck 的参数:

- **--enable=warning** – 指定想要获得警告消息,可以启用其他检查——更多详细信息请参阅 Cppcheck 手册(链接可在扩展阅读部分找到)。
- **--error-exitcode=10** – 这指定当 cppcheck 检测到问题时,希望获得错误代码。这可以是从 1 到 255 的任何数字(0 表示成功),尽管有些数字可以由系统保留。

使用非常方便——使用 AddCppCheck 将通知 CMake 需要在指定的目标上自动运行检查。

实际上已经在 src 和 test 子目录中创建了所有文件。现在，可以构建我们的解决方案并，对其进行全面测试。现在到了进行安装和打包的时候了。

## 12.6. 安装和打包

回到上一章讨论的主题，从设置安装和打包所需文件的快速概述开始：

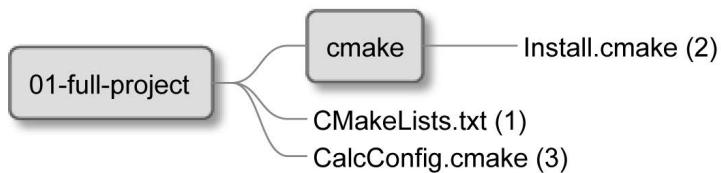


图 12.6 配置安装和打包的文件

这里只需要文件——大部分工作已经在前面的小节中完成了。顶层的列表文件包括一个 CMake 模块，将处理这个过程：

```
chapter-12/01-full-project/CMakeLists.txt (fragment)

...
include(Install)
```

我们有兴趣安装两个项目：

- Calc 库构件：静态库、动态库、头文件，以及目标导出的文件
- Calc 控制台可执行文件

包定义配置文件将只引入库目标，因为潜在的使用项目不会依赖于可执行文件。

配置安装步骤之后，将继续进行 CPack 配置。Install 模块的概述如下：

```
chapter-12/01-full-project/cmake/Install.cmake (overview)

Includes
Installation of Calc library
Installation of Calc Console executable
Configuration of CPack
```

一切都计划好了，所以是时候为库编写安装模块了。

### 12.6.1 库的安装

安装库最好从配置逻辑目标，并为其工件指定目的地开始。为了避免手动提供路径，将使用 GNUInstallDirs 模块提供的默认值。为了模块化，将把构件分组到组件中。默认安装将安装所有文件，可以选择只安装运行时组件，而跳过开发构件：

```
chapter-12/01-full-project/cmake/Install.cmake (fragment)

include(GNUInstallDirs)
Calc library
install(TARGETS calc_obj calc_shared calc_static
 EXPORT CalcLibrary
 ARCHIVE COMPONENT development
 LIBRARY COMPONENT runtime
 PUBLIC_HEADER DESTINATION
 ${CMAKE_INSTALL_INCLUDEDIR}/calc
 COMPONENT runtime
)
)
```

安装过程中，我们想用 ldconfig 注册复制的动态库：

```
chapter-12/01-full-project/cmake/Install.cmake (continued)

if (UNIX)
 install(CODE "execute_process(COMMAND ldconfig)"
 COMPONENT runtime
)
endif()
```

准备好这些步骤后，就可以通过将库包装在可重用的 CMake 包中，使它对其他 CMake 项目可见。需要生成并安装目标导出文件和包含它的配置文件：

```
chapter-12/01-full-project/cmake/Install.cmake (continued)

install(EXPORT CalcLibrary
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
 NAMESPACE Calc::
 COMPONENT runtime
)
install(FILES "CalcConfig.cmake"
 DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
)
```

对于非常简单的包，配置文件可以非常小：

```
chapter-12/01-full-project/CalcConfig.cmake\\

include("${CMAKE_CURRENT_LIST_DIR}/CalcLibrary.cmake")
```

现在，当构建解决方案后可以以`--install`模式运行`cmake`，将安装该库。剩下要安装的就是可执行文件了。

### 12.6.2 可执行文件的安装

安装二进制可执行程序是所有步骤中最简单的步骤，只需要使用一个命令：

```
chapter-12/01-full-project/cmake/Install.cmake (continued)

CalcConsole runtime
install(TARGETS calc_console
 RUNTIME COMPONENT runtime
)
```

完成了！现在来到了配置的最后一步——打包。

### 12.6.3 使用 CPack 打包

这时，就可以随心所欲地配置受支持的包类型。然而，对于这个项目，基本配置就足够了：

```
chapter-12/01-full-project/cmake/Install.cmake (continued)

CPack configuration
set(CPACK_PACKAGE_VENDOR "Rafal Swidzinski")
set(CPACK_PACKAGE_CONTACT "email@example.com")
set(CPACK_PACKAGE_DESCRIPTION "Simple Calculator")
include(CPack)
```

这种最小的设置适用于标准档案，例如 ZIP 文件。可以用一个命令来测试整个安装和打包（项目必须事先构建）：

```
cpack -G TGZ -B packages
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: Calc
CPack: - Install project: Calc []
CPack: Create package
CPack: - package: /tmp/b/packages/Calc-1.0.0-Linux.tar.gz
generated.
```

这就结束了安装和包装，下一个任务是生成文档。

## 12.7. 提供文档

当然，专业项目的最后一个要素是文档。其分为两类：

- 技术文档(接口、设计、类和文件)
- 一般文档(所有其他非技术文档)

正如在第10章中看到的，使用Doxygen可以使用CMake自动生成很多技术文档。

### 12.7.1 自动生成文档

需要提到的一点是：有些项目在构建阶段生成文档，并将其与项目的其余部分打包，这是个人喜好的问题。对于这个项目，不会这样做。您可能有很好的理由选择其他方式(例如在线托管文档)。

图12.7展示了这个过程中使用的执行流的概述：

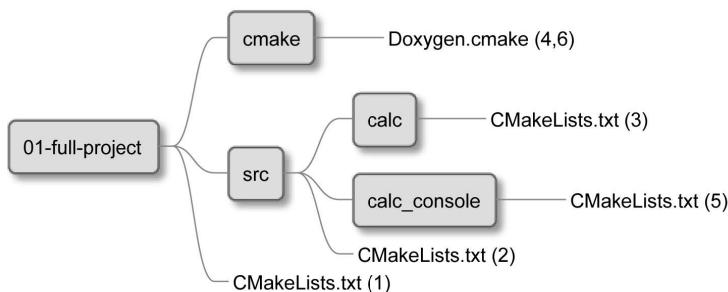


图12.7 用于生成文档的文件

我们的生成文档目标，将创建另一个CMake实用程序模块Doxygen。这里可以使用Doxygen查找模块，并下载Doxygen的-awesome-css主题：

```
chapter-12/01-full-project/cmake/Doxygen.cmake (fragment)

find_package(Doxygen REQUIRED)
include(FetchContent)
FetchContent_Declare(doxxygen-awesome-css
 GIT_REPOSITORY
 https://github.com/jothepro/doxygen-awesome-css.git
 GIT_TAG
 v1.6.0
)
FetchContent_MakeAvailable(doxxygen-awesome-css)
```

然后，需要一个函数来创建生成文档的目标。这将借鉴第10章介绍的代码，并对其进行修改以支持多个目标。

```
chapter-12/01-full-project/cmake/Doxygen.cmake (continued)

function(Doxygen target input)
```

```

set(NAME "doxygen-${target}")
set(DOXYGEN_HTML_OUTPUT
 ${PROJECT_BINARY_DIR}/${NAME})
set(DOXYGEN_GENERATE_HTML YES)
set(DOXYGEN_GENERATE_TREEVIEW YES)
set(DOXYGEN_HAVE_DOT YES)
set(DOXYGEN_DOT_IMAGE_FORMAT svg)
set(DOXYGEN_DOT_TRANSPARENT YES)
set(DOXYGEN_HTML_EXTRA_STYLESHEET
 ${doxygen-awesome-css_SOURCE_DIR}/doxygenawesome.css)
doxygen_add_docs(${NAME}
 ${PROJECT_SOURCE_DIR}/${input}
 COMMENT "Generate HTML documentation"
)
endfunction()

```

现在，需要通过为库目标调用这个函数：

```

chapter-12/01-full-project/src/calc/CMakeLists.txt (continued)

... calc_static target definition
... testing and program analysis modules
include(Doxygen)
Doxygen(calc src/calc)

```

我们为控制台可执行文件调用它：

```

chapter-12/01-full-project/src/calc_console/CMakeLists.txt (continued)

... calc_static target definition
... testing and program analysis modules
include(Doxygen)
Doxygen(calc_console src/calc_console)

add_executable(calc_console bootstrap.cpp)
target_link_libraries(calc_console calc_static)

```

项目中添加了两个新目标：doxygen-calc 和 doxygen-calc\_console，并且可以按需生成技术文档。还需要提供其他文件么？

## 12.7.2 非技术性文件

专业的项目应该始终包含至少两个存储在顶层目录中的文档：

- README -项目概述
- LICENSE -指定项目的软件许可证

也可以考虑添加以下内容:

- INSTALL -描述安装所需的步骤
- CHANGELOG -列出在不同版本中发生的重要更改
- AUTHORS -若项目有多个贡献者，则包含名单和联系方式
- BUGS -告知已知的错误，并指导如何报告新错误

当涉及到这些文件时，CMake 本身并没有发挥声明作用——没有自动的行为或脚本可以使用这些文件是 C++ 项目的重要组成部分，为了完整性应该将其涵盖。作为参考，这里将提供一个最小的示例文件集，从一个简短的 README 文件开始:

```
chapter-12/01-full-project/README.md

Calc Console
Calc Console is a calculator that adds two numbers in a
terminal. It does all the math by using a **Calc** library.
This library is also available in this package.

This application is written in C++ and built with CMake.

More information
- Installation instructions are in the INSTALL file
- License is in the LICENSE file
```

这句话很短，可能有点傻。注意扩展名.md——代表 Markdown，这是一种基于文本的格式化语言，易于阅读。像 GitHub 这样的网站和许多文本编辑器将以丰富的格式呈现这些文件。

INSTALL 文件看起来像这样:

```
chapter-12/01-full-project/INSTALL

To install this software you'll need to provide the following:

- C++ compiler supporting C++17
- CMake >= 3.20
- GIT
- Doxygen + Graphviz
- CPPCheck
- Valgrind

This project also depends on GTest, GMock and FXTUI. This
software is automatically pulled from external repositories
during the installation.

To configure the project type:
cmake -B <temporary-directory>

Then you can build the project:
```

```
cmake --build <temporary-directory>

And finally install it:
cmake --install <temporary-directory>

To generate the documentation run:
cmake --build <temporary-directory> -t doxygen-calc
cmake --build <temporary-directory> -t doxygen-calc_console
```

这个文件有点长，但涵盖了最重要的需求、步骤和命令，可以很好地满足需求。

LICENSE 文件有点棘手，因为需要版权法方面的一些专业知识(或其他)。我们不需要自己编写所有的条款，而是可以像许多其他项目一样使用现成的软件许可。对于这个项目，将使用 MIT 许可。根据具体项目的需要，可能想要选择其他的东西——查看扩展阅读部分，以获得一些有用的参考：

```
chapter-12/01-full-project/LICENSE

Copyright 2022 Rafal Swidzinski

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/
or sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

最后是 CHANGELOG，最好跟踪文件中的更改，以便使用项目的开发人员可以轻松地找到支持所需特性的版本。例如，在 0.8.2 版本中向库添加了乘法特性，这样说可能会很有用。像下面这样简单的方法已经很有帮助了：

```
chapter-12/01-full-project/CHANGELOG

1.0.0 Public version with installer
0.8.2 Multiplication added to the Calc Library
0.5.1 Introducing the Calc Console application
```

我们的项目现在已经完成了——可以构建、测试、生成包，将所有源代码上传到存储库，并发布工件。当然，若这可以自动发生，也可以使用 CI/CD 流水，这就更容易了。但那是另一本书所要做的事情了。

## 12.8. 总结

本章结束了我们通过 CMake 的漫长旅程。现在您完全理解了 CMake 的目标是解决什么问题，以及自动化这些解决方案需要哪些步骤。

前三章中，探索了所有的基础知识：什么是 CMake，用户如何利用它来激活原始源代码，CMake 的关键组件是什么，以及不同的项目文件有什么目的。我们解释了 CMake 的语法：注释、命令调用、参数、变量和控制结构。了解模块和子项目是如何工作的，正确的项目结构是什么，以及如何使用各种平台和工具链。

本书的第二部分教我们如何使用 CMake 进行构建：如何使用目标、定制命令、构建类型和生成器表达式。我们深入研究了编译的技术细节，以及预处理器和优化器的配置，讨论了链接并介绍了不同的库类型。然后，研究了 CMake 如何使用 FetchContent 和 ExternalProject 模块帮助管理项目的依赖性，还研究了 Git 子模块作为可能的替代方案，研究了如何使用 find\_package() 和 FindPkgConfig 查找已安装的包。若这些还不够，可以编写自己的查找模块。

最后一部分告诉我们如何进行自动化测试、分析、文档编制、安装和打包。我们研究了 CTest 和测试框架：Catch2、GoogleTest 和 GoogleMock。报告也包括在内。第 9 章，程序分析工具，了解了不同的分析工具：格式化器和静态检查器（Clang-Tidy、Cppcheck 等），并解释了如何从 Valgrind 套件中添加 Memcheck 内存分析器，这里简要描述了如何使用 Doxygen 生成文档，以及如何使文档看起来更美观。最后，演示了如何在系统上安装项目，创建可重用的 CMake 包，以及配置和使用 CPack 生成二进制包。

最后一章使用了所有这些知识来展示一个完全专业的项目。

恭喜你阅读完了这本书。我们介绍了开发、测试和打包高质量 C++ 软件所需的所有内容。从这里取得进步的最好方法是将所学到的应用到实践中，从而创建更多优秀的软件。祝诸君好运！

R.

## 12.9. 扩展阅读

更多资料，请访问以下链接：

- 构建静态库和动态库：

<https://stackoverflow.com/q/2152077>

- FXTUI 项目：

<https://github.com/ArthurSonzogni/FXTUI>

- option() 指令的文档：

<https://cmake.org/cmake/help/latest/command/option.html>

- (开源软件) 发布前的准备，由 Google 提供：

<https://opensource.google/docs/releasing/preparing/>

- 为什么不能对 gcc 预编译的头文件使用 Clang-Tidy:

[https://gitlab.kitware.com/cmake/cmake/-/issues/22081#note\\_943104](https://gitlab.kitware.com/cmake/cmake/-/issues/22081#note_943104)

- Cppcheck 手册:

<https://cppcheck.sourceforge.io/manual.pdf>

- 如何编写自述文件 README:

<https://www.freecodecamp.org/news/how-to-write-a-good-readme-file/>

- GitHub 项目知识共享许可证:

<https://github.com/santisoler/cc-licenses>

- GitHub 认可的常用项目许可:

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository>

# 附录：其它指令

每种语言都有实用命令，在各种场合都会派上用场。CMake 在这方面也没有什么不同：它为简单的算术、位操作、字符串操作，以及对列表和文件的操作提供了工具。有趣的是，需要它们的情况相对较少（多亏了多年来编写的所有增强和模块），但在更自动化的项目中仍然需要。

因此，本附录简要总结了各种命令及其多种模式。将此作为一个方便的离线参考或官方文档的简化版本。若需要更多信息，请访问提供的链接。

- `string()` 指令
- `list()` 指令
- `file()` 指令
- `math()` 指令

## string() 指令

`string()` 命令用于操作字符串。它提供了多种模式，可以对字符串执行不同的操作：搜索和替换、操作、比较、哈希码、生成和 JSON 操作（自 CMake 3.19 以来的最后一个可用模式）。

完整的细节可以在在线文档中找到：<https://cmake.org/cmake/help/latest/command/string.html>。

接受 `<input>` 参数的 `String()` 模式将接受多个 `<input>` 值，并在执行命令前将其连接起来：

```
string(PREPEND myVariable "a" "b" "c")
```

这等价于：

```
string(PREPEND myVariable "abc")
```

来研究一下所有可用的 `string()` 模式。

### 搜索和替换

有以下几种模式：

- `string(FIND <haystack> <pattern> <out> [REVERSE])` 在 `<haystack>` 字符串中搜索 `<pattern>`，并将找到的位置作为整数写入 `<out>` 变量。若使用 `REVERSE` 标志，将从字符串的末尾搜索到开头。这仅适用于 ASCII 字符串（不提供多字节支持）。
- `string(REPLACE <pattern> <replace> <out> <input>)` 用 `<replace>` 替换 `<input>` 中 `<pattern>` 的所有出现，并存储在 `<out>` 变量中。
- `string(REGEX MATCH <pattern> <out> <input>)` 将 `<input>` 中 `<pattern>` 的第一次出现与 `<replace>` 匹配，并将其存储在 `<out>` 变量中。
- `string(REGEX MATCHALL <pattern> <out> <input>)` 用 `<replace>` 匹配 `<input>` 中 `<pattern>` 的所有出现，并将它们作为逗号分隔的列表存储在 `<out>` 变量中。
- `string(REGEX REPLACE <pattern> <replace> <out> <input>)` 用 `<replace>` 表达式替换 `<input>` 中 `<pattern>` 的所有出现，并存储在 `<out>` 变量中。

正则表达式操作遵循标准库 `<regex>` 标头中定义的 C++ 语法。可以使用捕获组添加匹配 `<replace>` 表达式与数字占位符: `\1`, `\2`... (需要双反斜杠, 以便正确解析参数)。

## 操作

有以下几种模式:

- `string(APPEND <out> <input>)` 通过附加 `<input>` 字符串, 改变存储在 `<out>` 中的字符串。
- `string(PREPEND <out> <input>)` 通过在 `<input>` 字符串前面加上前缀, 改变存储在 `<out>` 中的字符串。
- `string(CONCAT <out> <input>)` 连接所有提供的 `<input>` 字符串并将它们存储在 `<out>` 变量中。
- `string(JOIN <glue> <out> <input>)` 将所有提供的 `<input>` 字符串与 `<glue>` 值交错, 并将其作为串联的字符串存储在 `<out>` 变量中 (不要对列表变量使用此模式)。
- `string(TOLOWER <string> <out>)` 将 `<string>` 转换为小写并将其存储在 `<out>` 变量中。
- `string(LENGTH <string> <out>)` 计算 `<string>` 的字节数, 并将结果存储在 `<out>` 变量中。
- `string(SUBSTRING <string> <begin> <length> <out>)` 从 `<begin>` 字节开始提取长度 `<length>` 字节的 `<string>` 的子字符串, 并将其存储在 `<out>` 变量中。 `length` 为 -1 可以理解为“直到字符串结束”。
- `string(STRIPI <string> <out>)` 从 `<string>` 中删除尾随和前导空格, 并将结果存储在 `<out>` 变量中。
- `string(GENEX_STRIP <string> <out>)` 删除 `<string>` 中使用的所有生成器表达式, 并将结果存储在 `<out>` 变量中。
- `string(REPEAT <string> <count> <out>)` 生成一个包含 `<string>` 的 `<count>` 次重复的字符串, 并将其存储在 `<out>` 变量中。

## 比较

字符串的比较形式如下所示:

```
string (COMPARE <operation> <stringA> <stringB> <out>)
```

`<operation>` 参数是以下参数之一: LESS, GREATER, EQUAL, NOTEQUAL, LESS\_EQUAL 或 GREATER\_EQUAL。将用于比较 `<stringA>` 与 `<stringB>`, 结果 (true 或 false) 将存储在 `<out>` 变量中。

## 哈希

哈希模式有以下签名:

```
string (<algorithm> <out> <string>)
```

哈希有两个输入 `<string>` 和 `<algorithm>`, 并将结果存储在 `<out>` 变量中。支持以下算法:

- MD5: Message-Digest Algorithm 5, RFC 1321
- SHA1: US Secure Hash Algorithm 1, RFC 3174
- SHA224: US Secure Hash Algorithms, RFC 4634
- SHA256: US Secure Hash Algorithms, RFC 4634

- SHA384: US Secure Hash Algorithms, RFC 4634
- SHA512: US Secure Hash Algorithms, RFC 4634
- SHA3\_224: Keccak SHA-3
- SHA3\_256: Keccak SHA-3
- SHA3\_384: Keccak SHA-3
- SHA3\_512: Keccak SHA-3

## 生成

有以下几种模式:

- string(ASCII <number>... <out>) 在 <out> 变量中存储给定 <number> 的 ASCII 字符。
- string(HEX <string> <out>) 将 <string> 转换为它的十六进制表示形式并将其存储在 <out> 变量中 (自 CMake 3.18 以来)。
- string(CONFIGURE <string> <out> [<@ONLY>] [<ESCAPE\_QUOTES>]) 与 configure\_file() 的工作原理完全相同，但适用于字符串。结果存储在 <out> 变量中。
- string(MAKE\_C\_IDENTIFIER <string> <out>) 将 <string> 中的非字母数字字符转换为下划线，并将结果存储在 <out> 变量中。
- string(RANDOM [<LENGTH> <len>] [<ALPHABET> <alphabet>] [<RANDOM\_SEED> <seed>] <out>) 使用来自随机种子 <seed> 的可选 <alphabet> 生成 <len> 字符的随机字符串 (默认为 5)，并将结果存储在 <out> 变量中。
- string(TIMESTAMP <out> [<format>] [<UTC>]) 生成一个表示当前日期和时间的字符串，并将其存储在 <out> 变量中。
- string(UUID <out> ...) 生成一个通用唯一标识符。这种模式使用起来有点复杂。

## JSON

对 JSON 格式字符串的操作使用以下签名:

```
string(JSON <out> [<ERROR_VARIABLE> <error>] <operation + args>)
```

有几种操作，都将结果存储在 <out> 变量中，错误存储在 <error> 变量中。操作及其参数如下:

- GET <json> <member|index>... 使用 <member> 路径或 <index> 从 <json> 字符串中返回一个或多个元素的值。
- TYPE <json> <member|index>... 使用 <member> 路径或 <index> 从 <json> 字符串中返回一个或多个元素的类型。
- MEMBER <json> <member|index>... <array-index> 使用 <member> 路径或 <index> 从 <json> 字符串返回 <array-index> 位置上的一个或多个数组类型元素的成员名。
- LENGTH <json> <member|index>... 使用 <member> 路径或 <index> 从 <json> 字符串中返回一个或多个数组类型元素的元素计数。
- REMOVE <json> <member|index>... 返回使用 <member> 路径或 <index> 从 <json> 字符串中删除一个或多个元素的结果。

- SET <json> <member|index>... <value> 使用 <member> 路径或 <index> 从 <json> 字符串中返回 <value> 上升到一个或多个元素的结果。
- EQUAL <jsonA> <jsonB> 计算 <jsonA> 和 <jsonB> 是否相等。

## list() 指令

这个指令提供了关于列表的基本操作: 读取、搜索、修改和排序。某些模式会改变列表 (改变原始值)。若以后需要它, 请确保复制原始值。

完整的细节可以在在线文档中找到:

<https://cmake.org/cmake/help/latest/command/list.html>

### 读取

有以下几种模式:

- list(LENGTH <list> <out>) 计算 <list> 变量中的元素, 并将结果存储在 <out> 变量中。
- list(GET <list> <index>... <out>) 将 <list> 元素与 <index> 索引列表一起复制到 <out> 变量。
- list(JOIN <list> <glue> <out>) 用 <glue> 分隔符 <list> 元素, 并将结果字符串存储在 <out> 变量中。
- list(SUBLIST <list> <begin> <length> <out>) 与 GET 模式类似, 但操作范围而不是显式索引。若 <length> 为-1, 则返回 <list> 变量中提供的从 <begin> 索引到列表末尾的元素。

### 搜索

该模式只是在 <list> 变量中查找 <needle> 元素的索引, 并将结果存储在 <out> 变量中 (若没有找到该元素, 则为-1):

```
list(FIND <list> <needle> <out>)
```

### 修改

有以下几种模式:

- list(APPEND <list> <element>...) 将一个或多个 <element> 值添加到 <list> 变量的末尾。
- list(PREPEND <list> [<element>...]) 与 APPEND 类似, 但在 <list> 变量的开头添加元素。
- list(FILTER <list> INCLUDE | EXCLUDE REGEX <pattern>) 过滤 <list> 变量以包含或排除值匹配 <pattern> 的元素。
- list(INSERT <list> <index> [<element>...]) 在给定的 <index> 处向 <list> 变量添加一个或多个 <element> 值。
- list(POP\_BACK <list> [<out>...]) 从 <list> 变量末尾删除一个元素, 并将其存储在可选的 <out> 变量中。若提供了多个 <out> 变量, 则将删除更多元素来填充它们。
- list(POP\_FRONT <list> [<out>...]) 类似于 POP\_BACK, 但从 <list> 变量的开头删除一个元素。
- list(REMOVE\_ITEM <list> <value>...) FILTER EXCLUDE 的简写, 但不支持正则表达式。
- list(REMOVE\_AT <list> <index>...) 从 <list> 中移除特定 <index> 处的元素。

- `list(REMOVE_DUPLICATES <list>)` 从 `<list>` 中删除重复项。
- `list(TRANSFORM <list> <action> [<selector>] [OUTPUT_VARIABLE <out>])` 对 `<list>` 元素应用特定的转换。

默认情况下，该动作应用于所有元素，但可以通过添加 `<selector>` 来限制效果。若提供了 `OUTPUT_VARIABLE` 关键字，提供的列表将发生变化（就地更改），结果将存储在 `<out>` 变量中。

以下选择器可用:`AT <index>`, `FOR <start> <stop>[<step>]`, 以及 `REGEX <pattern>`。操作包括 `APPEND <string>`, `PREPEND <string>`, `TOLOWER`, `TOUPPER`, `STRIP`, `GENEX_STRIP` 和 `REPLACE <expression>`。工作原理与具有相同名称的 `string()` 模式相同。

## 排序

有以下几种模式:

- `list(VERSE <list>)` 简单地颠倒 `<list>` 的顺序。
- `list(SORT <list>)` 按字典序对列表进行排序。更多高级选项请参考在线手册。

## file() 指令

该指令提供与文件相关的各种操作: 读取、传输、锁定和归档，还提供了检查文件系统和对表示路径的字符串的操作的模式。

完整的细节可以在在线文档中找到:

<https://cmake.org/cmake/help/latest/command/file.html>

## 读取

有以下几种模式:

- `file(READ <filename> <out> [OFFSET <o>] [LIMIT <max>] [HEX])` 从 `<filename>` 读取文件到 `<out>` 变量。读取可选地从偏移量 `<o>` 开始，并遵循 `<max>` 字节的可选限制。HEX 标志指定输出应该转换为十六进制表示。
- `file(STRINGS <filename> <out>)` 从 `<filename>` 处的文件读取字符串到 `<out>` 变量。
- `file(<algorithm> <filename> <out>)` 从 `<filename>` 的文件计算 `<algorithm>` 哈希，并将结果存储在 `<out>` 变量中。可用的算法与 `string()` 哈希算法相同。
- `file(TIMESTAMP <filename> <out> [<format>])` 生成文件名 `<filename>` 的文件时间戳的字符串表示形式，并将其存储在 `<out>` 变量中，可选地接受 `<format>` 字符串。
- `file(GET_RUNTIME_DEPENDENCIES [...])` 获取指定文件的运行时依赖项。这是一个高级命令，只能在 `install(CODE)` 或 `install(SCRIPT)` 中使用。

## 写入

有以下几种模式:

- `file({WRITE | APPEND} <filename> <content>...)` 将所有 `<content>` 参数写入或追加到 `<filename>` 的文件。若提供的系统路径不存在，将递归地创建它。
- `file({TOUCH | TOUCH_NOCREATE} [<filename>...])` 更新 `<filename>` 的时间戳。若文件不存在，将只在 `TOUCH` 模式下创建。
- `file(GENERATE OUTPUT <output-file> [...])` 是一种高级模式，它为当前 CMake Generator 的每个构建配置生成一个输出文件。
- `file(CONFIGURE OUTPUT <output-file> CONTENT <content> [...])` 像 `GENERATE_OUTPUT` 一样工作，也可以通过用值替换变量占位符来配置生成的文件。

## 文件系统

有以下几种模式:

- `file({GLOB | GLOB_RECURSE} <out> [...] [<globbingexpression>...])` 生成匹配 `<globbingexpression>` 的文件列表，并将其存储在 `<out>` 变量中。`GLOB_RECURSE` 模式也将扫描嵌套目录。
- `file(RENAMEN <oldname> <newname>)` 将文件从 `<oldname>` 移动到 `<newname>`。
- `file(REMOVE | REMOVE_RECURSE [<files>...])` 删除 `<files>`。`REMOVE_RECURSE` 会删除目录。
- `file(MAKE_DIRECTORY [<dir>...])` 创建目录
- `file(COPY <file>... DESTINATION <dir> [...])` 将文件复制到 `<dir>` 目标。提供筛选、设置权限、软链接等选项。
- `file(SIZE <filename> <out>)` 读取 `<filename>` 的字节大小，并将其存储在 `<out>` 变量中。
- `file(READ_SYMLINK <linkname> <out>)` 读取 `<linkname>` 符号链接的目标路径，并将其存储在 `<out>` 变量中。
- `file(CREATE_LINK <original> <linkname> [...])` 在 `<linkname>` 处创建一个指向 `<original>` 的软链接。
- `file({CHMOD|CHMOD_RECURSE} <files>... <directories>... PERMISSIONS <permissions>... [...])` 设置文件和目录的权限。

## 路径转换

有以下几种模式:

- `file(REAL_PATH <path> <out> [BASE_DIRECTORY <dir>])` 从相对路径计算绝对路径并将其存储在 `<out>` 变量中。可选地接受 `<dir>` 基目录。从 CMake 3.19 开始可用。
- `file(RELATIVE_PATH <out> <directory> <file>)` 计算相对于 `<directory>` 的 `<file>` 路径，并将其存储在 `<out>` 变量中。
- `file({TO_CMAKE_PATH | TO_NATIVE_PATH} <path> <out>)` 将转换为 CMake 路径 (用正斜杠分隔的目录) 到平台的本机路径并返回，结果存储在 `<out>` 变量中。

## 传输

有以下几种模式:

- file(DOWNLOAD <url> [<path>] [...]) 从 <url> 下载文件并将其存储在 path 中。
- file(UPLOAD <file> <url> [...]) 上传 <file> 到 URL。

## 锁定

锁定模式在 <path> 资源上放置一个建议锁:

```
file(LOCK <path> [DIRECTORY] [RELEASE]
 [GUARD <FUNCTION|FILE|PROCESS>]
 [RESULT_VARIABLE <out>]
 [TIMEOUT <seconds>])
```

该锁可以选择作用域为 FUNCTION、FILE 或 PROCESS，并以 <seconds> 的超时进行限制。要释放锁，需要 RELEASE 关键字，结果将存储在 <out> 变量中。

## 归档

归档文件的创建具有以下签名:

```
file(ARCHIVE_CREATE OUTPUT <destination> PATHS <source>...
 [FORMAT <format>]
 [COMPRESSION <type> [COMPRESSION_LEVEL <level>]]
 [MTIME <mtime>] [VERBOSE])
```

它在 <destination> 路径上创建一个存档文件，其中包含支持的格式之一的 <source> 文件:7zip、gnutar、pax、paxr、raw 或 zip (paxr 是默认格式)。若所选格式支持压缩级别，则可以将其提供为一个单位数整数 0-9，其中 0 为默认值。

提取模式具有以下签名:

```
file(ARCHIVE_EXTRACT INPUT <archive> [DESTINATION <dir>]
 [PATTERNS <patterns>...] [LIST_ONLY] [VERBOSE])
```

它将匹配可选 <patterns> 值的文件从 <archive> 提取到目标 <dir>。若有 LIST\_ONLY 关键字，则不会提取文件，而是只列出文件。

## math() 指令

CMake 还支持一些简单的算术操作。详细信息请参见在线文档:

<https://cmake.org/cmake/help/latest/command/math.html>

要计算一个数学表达式并将其作为可选 <format>(十六进制或十进制) 的字符串存储在 <out> 变量中，请使用以下签名:

```
math(EXPR <out> "<expression>" [OUTPUT_FORMAT <format>])
```

<expression> 值是一个字符串，支持 C 代码中出现的操作符 (它们在这里具有相同含义):

- 算术: +, -, \*, /, % 模除运算
- 位操作: | or, & and, ^ xor, not, << 左移位, >> 右移位
- 括号 (...)

常量值可以以十进制或十六进制格式提供。