

EXPERT INSIGHT

Modern CMake for C++

**Effortlessly build cutting-edge C++ code
and deliver high-quality solutions**

Foreword by:

Alexander Kushnir
Principal Software Engineer, Biosense Webster

Second Edition

Rafał Świdziński

packt

Modern CMake for C++ - Second Edition

轻松构建前沿 C++ 代码，提供高质量的解决方案

作者：Rafał Świdziński

译者：[陈晓伟](#)

目录

序	14
关于作者	15
关于评审	16
前言	17
适读人群	17
关于本书	17
如何阅读	18
下载示例	18
下载彩图	18
内容约定	18
第 1 章 使用 CMake	20
1.1. 示例下载	20
1.2. 基本知识	21
1.2.1 什么是 CMake?	21
1.2.2 工作原理	22
配置阶段	22
生成阶段	23
构建阶段	23
1.3. 安装 CMake	25
1.3.1 Docker	26
1.3.2 Windows	26
1.3.3 Linux	27
1.3.4 macOS	28
1.3.5 从源代码构建	28
1.4. 命令行	29
1.4.1 CMake 命令行	29
生成项目构建系统	29
构建项目	34

安装项目	36
执行脚本	37
命令行工具	38
工作流预设	38
获取帮助	38
1.4.2 CTest	39
1.4.3 CPack	39
1.4.4 CMake GUI	39
1.4.5 ccmake	40
1.5. 项目结构	41
1.5.1 源代码目录	41
1.5.2 构建目录	42
1.5.3 列表文件	42
项目文件	42
缓存文件	43
包定义文件	44
生成的文件	44
1.5.4 JSON 和 YAML 文件	44
预设文件	44
基于文件的 API	45
配置日志	45
1.5.5 Git 中忽略文件	45
1.6. 脚本和模块	46
1.6.1 脚本	46
1.6.2 工具模块	46
1.6.3 Find-模块	47
1.7. 总结	47
1.8. 扩展阅读	48
第 2 章 CMake 语言	49
2.1. 示例下载	49
2.2. 语法基础	50
2.2.1 注释	50
括号参数	52
引号参数	53
非引号参数	54
2.2.3 使用变量	55
2.2.3.1 变量引用	56

2.3.2 使用环境变量	57
2.3.3 使用缓存变量	58
2.3.4 如何在 CMake 中正确使用变量作用域	58
2.4. 使用列表	60
2.5. 控制结构	61
2.5.1 条件块	62
语法	62
2.5.2 循环	65
while()	65
foreach()	65
2.5.3 自定义命令	67
宏	67
函数	68
CMake 中的过程范式	69
关于命名约定	70
2.6. 常用命令	71
2.6.1 message()	71
2.6.2 include()	73
2.6.3 include_guard()	74
2.6.4 file()	74
2.6.5 execute_process()	74
2.7. 总结	75
2.8. 扩展阅读	75
第 3 章 主流的 IDE 中使用 CMake	76
3.1. 了解 IDE	76
3.1.1 选择 IDE	77
选择全面的 IDE	77
选择组织中广泛支持的 IDE	77
不要根据目标操作系统和平台选择 IDE	77
选择支持远程开发的 IDE	78
3.1.2 安装工具链	78
3.1.3 使用本书的示例与 IDE	79
3.2. CLion IDE	80
3.2.1 可能喜欢它的原因	80
3.2.2 第一步	81
3.2.3 高级功能：强大的调试器	82
3.3. Visual Studio Code	82
3.3.1 可能喜欢它的原因	83
3.3.2 第一步	83

3.3.3 高级功能: Dev Containers	84
3.4. Visual Studio IDE	84
3.4.1 可能喜欢它的原因	85
3.4.2 第一步	85
3.4.3 高级功能: 热重载调试	86
3.5. 总结	86
3.6. 扩展阅读	87
第 4 章 设置你的第一个 CMake 项目	88
4.1. 示例下载	88
4.2. 了解基本的指令和命令	89
4.2.1. 指定最低 CMake 版本	89
4.2.2. 定义语言和元数据	89
4.3. 划分项目	91
4.3.1. 使用子目录管理作用域	92
4.3.2. 何时使用嵌套项目	94
4.3.3. 保持外部项目外部化	94
4.4. 项目结构	94
4.5. 设置环境范围	98
4.5.1. 检测操作系统	98
4.5.2. 交叉编译——什么是主机和目标系统?	99
4.5.3. 简写变量	99
4.5.4. 主机系统信息	100
4.5.5. 平台是 32 位, 还是 64 位架构?	101
4.5.6. 系统的字节序是什么?	101
4.6. 配置工具链	101
4.6.1. 设置 C++ 标准	102
4.6.2. 坚持标准支持	102
4.6.3. 特定供应商的扩展	103
4.6.4. 过程间优化	103
4.6.5. 检查支持的编译器特性	104
4.6.6. 编译测试文件	104
4.7. 禁用源内构建	105
4.8. 总结	106
4.9. 扩展阅读	106
第 5 章 与目标一起工作	107
5.1. 示例下载	107
5.2. 理解目标的概念	107
5.2.1. 定义可执行目标	108

5.2.2. 定义库目标	109
5.2.3. 自定义目标	109
5.2.4. 依赖关系图	110
5.2.5. 可视化依赖关系	111
5.2.6. 设置目标的属性	113
传递目标的使用要求	113
处理冲突的传播属性	115
5.2.7. 识别伪目标	117
导入的目标	117
别名目标	117
接口库	117
5.2.8. 对象库	118
5.2.9. 构建目标	119
5.3. 编写自定义命令	119
5.3.1. 将自定义命令用作生成器	120
5.3.2. 将自定义命令用作目标钩子	121
5.4. 总结	122
5.5. 扩展阅读	122
第 6 章 使用生成器表达式	124
6.1. 示例下载	124
6.2. 什么是生成器表达式?	125
6.3. 学习通用表达式语法的基本规则	125
6.3.1. 嵌套	126
6.4. 条件扩展	126
6.4.1. 计算布尔值	127
逻辑运算符	127
比较	128
查询	128
6.5. 查询和转换	128
6.5.1. 处理字符串、列表和路径	128
6.5.2. 配置和平台的参数化	130
6.5.3. 调整工具链	131
6.5.4. 查询与目标相关的信息	132
6.5.5. 转义	134
6.6. 尝试示例	134
6.6.1. 构建配置	134
6.6.2. 系统特定的单行命令	135
6.6.3. 带有编译器特定标志的接口库	135
6.6.4. 嵌套生成器表达式	135

6.6.5. 布尔表达式与 BOOL 运算符的计算差异	136
6.7. 总结	137
6.8. 扩展阅读	137
第 7 章 用 CMake 编译 C++ 源代码	139
7.1. 示例下载	139
7.2. 编译的基础	139
7.2.1. 编译如何工作	140
7.2.2. 初始配置	141
要求编译器的特定功能	142
7.2.3. 管理目标源文件	143
7.3. 配置预处理器	144
7.3.1. 提供包含文件的路径	144
7.3.2. 预处理器定义	144
避免在单元测试中访问私有类字段	146
使用 git 提交跟踪编译版本	146
7.3.3. 配置头文件	147
7.4. 配置优化器	148
7.4.1. 通用级别	149
7.4.2. 函数内联	150
7.4.3. 循环展开	151
7.4.4. 循环向量化	152
7.5. 管理编译过程	153
7.5.1. 减少编译时间	153
预编译头文件	153
统一构建	154
7.5.2. 发现错误	156
配置错误和警告	156
调试构建	156
为调试器提供信息	158
7.6. 总结	160
7.7. 扩展阅读	160
第 8 章 链接可执行文件和库	162
8.1. 示例下载	162
8.2. 掌握正确链接的基本知识	162
8.3. 构建不同类型的库	166
8.3.1. 静态库	166
8.3.2. 共享库	167
8.3.3. 共享模块	168

8.3.4. 位置无关代码 (PIC)	168
8.4. 解决 ODR 的问题	169
8.4.1. 解决动态链接中的重复符号问题	171
8.4.2. 使用命名空间——不要依赖链接器	172
8.5. 连接和未解析符号的顺序	173
8.5.1. 处理未引用的符号	175
8.6. 分离 main() 进行测试	176
8.7. 总结	178
8.8. 扩展阅读	178
第 9 章 管理依赖关系	180
9.1. 示例下载	180
9.2. 使用已经安装的依赖项	180
9.2.1. 使用 CMake 的 find_package() 查找包 编写自己的 find 模块	181 185
9.2.2. 使用 FindPkgConfig 发现遗留包	190
9.3. 使用系统中不存在的依赖项	192
9.3.1. FetchContent	192
基本的 YAML 读取器示例	193
下载依赖项	195
更新和打补丁	195
更新和打补丁	197
尽可能使用已安装的依赖项	198
9.3.2. ExternalProject	199
9.4. 总结	200
9.5. 扩展阅读	201
第 10 章 使用 C++20 模块	202
10.1. 环境准备	202
10.2. C++20 模块	203
10.3. 使用 C++20 模块支持编写项目	205
10.3.1. CMake 3.26 和 3.27 中启用实验性支持	205
10.3.2. 启用对 CMake 3.28 及以上的支持	206
10.3.3. 设置编译器要求	207
10.3.4. 声明一个 C++ 模块	207
10.4. 配置工具链	208
10.5. 总结	209
10.6. 扩展阅读	209

第 11 章 测试框架	210
11.1. 示例下载	210
11.2. 为什么要自动化测试?	210
11.3. 使用 CTest 在 CMake 中标准化测试	211
11.3.1. 构建并测试模式	212
11.3.2. 测试模式	213
查询测试	213
过滤测试	214
打乱测试	214
处理失败	214
重复测试	215
控制输出	216
其他选项	216
11.4. 为 CTest 创建最基本的单元测试	217
11.5. 为测试构建项目	221
11.6. 单元测试框架	224
11.6.1. Catch2	224
11.6.2. GoogleTest	227
Using GTest	227
GMock	229
11.7. 生成测试覆盖率报告	233
11.7.1. 使用 LCOV 生成覆盖率报告	235
11.7.2. 避免 SEGFAULT 陷阱	238
11.8. 总结	238
11.9. 扩展阅读	239
第 12 章 程序分析工具	240
12.1. 示例下载	240
12.2. 执行格式化	240
12.3. 使用静态检查器	244
12.3.1. clang-tidy	246
12.3.2. Cpplint	246
12.3.3. Cppcheck	247
12.3.4. include-what-you-use	247
12.3.5. Link What You Use	247
12.4. 动态分析与 Valgrind	248
12.4.1. Memcheck	248
12.4.2. Memcheck-Cover	251
12.5. 总结	254
12.6. 扩展阅读	254

第 13 章 生成文档	255
13.1. 示例下载	255
13.2. 添加 Doxygen 到项目中	255
13.3. 生成具有现代感的文档	260
13.4. 使用自定义 HTML 增强输出	261
13.5. 总结	264
13.6. 扩展阅读	264
第 14 章 安装与打包	265
14.1. 示例下载	265
14.2. 导出而不安装	266
14.3. 安装项目	268
14.3.1. 安装逻辑目标	269
利用不同平台上的默认目标目录	271
处理公共头文件	271
14.3.2. 低层安装	273
使用 <code>install(FILES)</code> 和 <code>install(PROGRAMS)</code> 安装 .	273
处理整个目录	276
14.3.3. 安装过程中调用脚本	279
14.3.4. 安装运行时依赖项	280
14.4. 创建可重用的包	280
14.4.1. 了解可重定位目标的问题	281
14.4.2. 安装目标导出文件	282
14.4.3. 编写基本的配置文件	283
14.4.4. 创建高级配置文件	285
14.4.5. 生成包版本文件	288
14.5. 定义组件	290
14.5.1. 如何在 <code>find_package()</code> 中使用组件	290
14.5.2. 如何在 <code>install()</code> 命令中使用组件	290
14.5.3. 管理版本化共享库的符号链接	292
14.6. 使用 CPack	292
14.7. 总结	295
14.8. 扩展阅读	295
第 15 章 创建你的专业项目	297
15.1. 示例下载	297
15.2. 计划工作	298
15.3. 项目布局	300
15.3.1. 共享库与静态库	301
15.3.2. 项目文件结构	302

15.4. 构建和管理依赖关系	303
15.4.1. 构建 Calc 库	305
15.4.2. 构建 Calc 可执行文件	307
15.5. 测试和程序分析	310
15.5.1. 准备 Coverage 模块	312
15.5.2. 准备 Memcheck 模块	314
15.5.3. 应用测试场景	315
15.5.4. 添加静态分析工具	317
15.6. 安装与打包	318
15.6.1. 安装库	319
15.6.2. 可执行文件的安装	321
15.6.3. 使用 CPack 打包	321
15.7. 提供文档	322
15.7.1. 生成技术文档	322
15.7.2. 为专业项目编写非技术性文档	324
15.8. 总结	326
15.9. 扩展阅读	326
第 16 章 编写 CMake 预设	328
16.1. 示例下载	328
16.2. 使用项目中定义的预设	328
16.3. 编写预设文件	329
16.4. 特定阶段的预设	330
16.4.1. 各预设阶段的共同特性	330
独特的名称字段	330
可选字段	331
与配置阶段预设的关联	331
16.4.2. 定义配置阶段预设	331
16.4.3. 定义构建阶段预设	333
16.4.4. 定义测试阶段预设	334
16.4.5. 定义打包阶段预设	336
16.4.6. 添加安装预设	337
16.5. 定义工作流预设	338
16.6. 添加条件和宏	339
16.7. 总结	341
16.8. 扩展阅读	342
附录	343
17.1. 其他命令	343
17.2. string()	343

A.2.1. 搜索和替换	343
A.2.2. 操作	344
A.2.3. 比较	345
A.2.4. 哈希	345
A.2.5. 生成	346
A.2.6. JSON	346
17.3. list()	347
A.3.1. 读取	347
A.3.2. 查找	347
A.3.3. 修改	347
A.3.4. 排序	348
17.4. file()	348
A.4.1. 读取	349
A.4.2. 写入	349
A.4.3. 文件系统	349
A.4.4. 路径转换	350
A.4.5. 传输	350
A.4.6. 锁定	350
A.4.7. 归档	351
17.5. math()	351

序

在 C++ 的不断发展的中，掌握 CMake 对于致力于编写高效、可维护和可扩展程序的开发者来说是必不可少的。《Modern CMake for C++》（由 Rafał Świdziński 撰写）就像一座灯塔，引导开发者们穿梭于 CMake 之中。

这本书不仅仅是一本手册，也是一次旅行。从基础开始，随着章节的深入，读者将掌握高阶技术。本书的独特之处在于其实用的方法，现实例子和最佳实践贯穿全文，确保读者不仅理解概念，而且知道如何在项目中应用。

读完这本书后，读者不仅会对 CMake 有深入的理解，会对驾驭 C++ 有更多的信心。这是有助于编写更干净、更高效代码所需的知识和技能，进而成为熟练的 CMake 开发者。

《Modern CMake for C++》不仅仅是一本书，也是一个工具，将提升读者的 C++ 开发技能。无论是初学者还是专家，这本书都将有助于你掌握 CMake，使代码更加健壮、可维护和可扩展。

*Alexander Kushnir
Biosense Webster 首席软件工程师*

关于作者

Rafał Świdziński 是谷歌的一名资深工程师，拥有超过 12 年的全栈开发经验。他领导过思科 Meraki、亚马逊和爱立信等行业巨头的项目，体现了对创新的承诺。作为一名自愿选择居住在伦敦的人，他始终站在技术进步的前沿，参与了许多个人创业项目。最近他转向医疗保健领域的 AI。Rafał 重视顶尖的代码质量和工艺，也会通过 YouTube 频道和出版的书籍分享见解。

致 Zoe – 无汝，无书（如果没有你，我无法写出这本书）

关于评审

Eric Noulard 拥有法国 ENSEEIHT 的工程学位和法国 UVSQ 的计算机科学博士学位。Eric 有着丰富的 25 年编写和编译各种语言源代码的经验。自 2006 年开始使用 CMake，他也积极参与了 CMake 的发展。Eric 曾为私营公司和政府机构服务。目前，他在 Antidot 工作，这是一家专注于语义搜索、人工智能和内容可访问性的软件供应商。Eric 在研究团队中，负责将如生成性 AI 和先进 NLP 处理等新技术引入 Antidot 的旗舰产品 Fluid Topics。

Giovanni Romano 拥有 28 年的 IT 行业经验，从软件开发到应用程序/组件设计。目前他在 Leica Geosystem AG 担任高级软件工程师，专注于设计 SDK、微服务和低延迟后端。作为 Nokia/Blackberry Qt 大使，他相信开源软件并致力于为该框架做出贡献。他的兴趣包括云原生应用程序、Kubernetes、Docker 和 GitOps。他喜欢使用 C 语言编程和打网球。

前言

创建顶级软件并非易事。在线研究这个主题的开发者难以确定哪些建议是当前的，哪些方法已经更新，或已经有更好的实践方式。此外，大多数资源以混乱的方式解释过程，缺乏适当的背景、上下文和结构。

《Modern CMake for C++》提供了一个端到端的指南，通过全面处理 C++ 解决方案的构建，提供了更简单的体验。它不仅教你如何在项目中使用 CMake，还强调了如何使项目保持可维护性、优雅和简洁。该指南会引导你自动化完成许多项目中的常见任务，包括构建、测试和打包。

本书还会指导如何组织源目录、构建目标和创建包。随着了解的深入，将学习编译和链接可执行文件和库，详细理解这些过程，并优化每个步骤以获得最佳结果。此外，将发现如何将外部依赖项（如第三方库、测试框架、程序分析工具和文档生成器），进而整合到自己的项目中。最后，将学习如何导出、安装和打包解决方案，以供内部和外部使用。

完成这本书后，你将能以专业水平使用 CMake。

适读人群

学会了 C++ 之后，很快就会发现，仅仅掌握语言本身并不足以以最高标准交付项目。这本书填补了这一空白：它面向任何渴望成为更好的软件开发者，甚至专业构建工程师的人！如果想要从零开始学习现代 CMake 或提升和更新自己的 CMake 技能，阅读这本书吧。它将帮助你了解如何制作顶级的 C++ 项目，并从其他构建环境中过渡。

关于本书

第 1 章，*CMake 入门*，包括 CMake 的安装、命令行界面的使用，并介绍了 CMake 项目所需的基本构建块。

第 2 章，*CMake 语言*，包括 CCMake 语言的基本概念，包括命令调用、参数、变量、控制结构和注释。

第 3 章，主流 *IDE* 中使用 *CMake*，强调了集成开发环境（IDE）的重要性，指导读者选择 IDE，并为 Clion、Visual Studio Code 和 Visual Studio IDE 为例，提供设置和说明。

第 4 章，*开启第一个 CMake 项目*，了解如何在顶层文件中配置基本的 CMake 项目，结构化文件树，并准备必要的工具链进行开发。

第 5 章，*了解目标*，探讨了逻辑构建目标的概念，理解它们的属性和不同类型，并学习如何为 CMake 项目自定义命令。

第 6 章，*生成器表达式*，解释了生成器表达式的目的和语法，包括如何使用它们进行条件扩展、查询和转换。

第 7 章，*编译 C++ 源代码*，深入探讨了编译过程，配置预处理器和优化器，并了解减少构建时间和提高调试的技术。

第 8 章，*链接可执行文件和库*，理解链接机制，不同类型的库，单一定义规则，链接顺序，以及如何准备测试。

第 9 章，*CMake* 中管理依赖项，了解如何管理第三方库，为那些缺乏 *CMake* 支持的库添加支持，并从互联网上获取外部依赖。

第 10 章，使用 *C++20* 模块，介绍了 *C++20* 模块，展示了如何在 *CMake* 中启用，并相应地配置工具链。

第 11 章，测试框架，理解自动化测试的重要性，利用 *CMake* 内置的测试支持，并使用主流框架开始单元测试。

第 12 章，程序分析工具，了解如何自动格式化源代码，并在构建时间和运行时检测软件错误。

第 13 章，生成文档，如何使用 *Doxygen* 从源代码自动创建文档，并添加样式以增强文档外观。

第 14 章，安装和打包，如何对项目进行发布，无论是否安装，创建可重用包，并为打包指定单个组件。

第 15 章，创建专业项目，应用本书所学到的所有知识来开发一个全面、专业级别的项目。

第 16 章，编写 *CMake* 预设，将高级项目配置封装到使用 *CMake* 预设文件的工作流程中，使项目设置和管理更加高效。

附录 - 其他命令，作为与字符串、列表、文件和数学运算相关的各种 *CMake* 命令的参考。

如何阅读

本书假设你对 *C++* 和类 *Unix* 系统有基本的熟悉。尽管 *Unix* 知识不是严格的要求，但它将有助于理解本书中给出的示例。

本书的目标是 *CMake* 3.26，但这里描述的大多数技术应该适用于 *CMake* 3.15（之后添加的功能通常会突出显示）。有些章节已经更新到 *CMake* 3.28，以涵盖最新功能。

运行示例的环境准备在第 1-3 章中介绍，如果熟悉这个工具，推荐使用本书提供的 Docker 镜像。

下载示例

本书的代码托管在 GitHub 上，地址为<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E>。我们还有其他丰富的书籍和视频代码包，也可在 GitHub 中找到，地址为：<https://github.com/PacktPublishing/>。去看看吧！

下载彩图

我们还提供了一个 PDF 文件，其中包含了本书中使用的屏幕截图/图表的彩色图片。可以在[这里下载](https://packt.link/gbp/9781805121800)：<https://packt.link/gbp/9781805121800>

内容约定

本书中使用了一些文本约定。

代码块如下设置：

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Hello)
3 add_executable(Hello hello.cpp)
```

命令行输入或输出如下：

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

Note

警告或重要注释

Tip

提示和技巧

第 1 章 使用 CMake

软件开发的神奇之处在于，不仅是在创造能够运行的机制，还有创造解决方案的思想。

为了将想法变为现实，我们按照以下循环工作：设计、编码和测试。我们创造变化，并用编译器能理解的语言表达，继续检查其是否如期工作。要从源码中创建正确、高质量的软件，需要小心地重复执行容易出错的任务：调用正确的命令、检查语法、链接二进制文件、运行测试、报告问题等。

记住每个步骤需要付出巨大的努力。相反，我们希望更专注于编码，将其他事情委托给工具。理想情况下，这个过程会在更改代码后，通过一个按钮启动。这个过程应该是智能的、快速的、可扩展的，并且在不同操作系统和环境中的工作方式相同，应该得到多个集成开发环境（IDE）的支持。并且，可以将其简化为持续集成（CI）流水线，每次向仓库提交更改时，都会构建和测试软件。

CMake 是满足许多此类需求的答案，但要正确配置和使用也要花一些心思。CMake 并不是复杂性的来源，复杂性来自于要处理的东西。别担心，我们将系统地学习整个过程，你会了解到软件构建是多么“简单”。

我知道你急于开始编写自己的 CMake 项目，这正是本书大部分内容中做的事情。但是，由于你将主要为了用户（包括你自己）创建项目，因此了解一下他们的视角对你来说很重要。

我们从这一点开始，逐步成为一个 CMake 高级用户。我们将介绍一些基础知识：这个工具是什么，工作原理，以及如何安装。然后，将深入探讨命令行和操作模式。最后，将总结项目文件的不同用途，并解释如何在完全不创建项目的情况下使用 CMake。

本章中，将包括以下主题：

- 基础知识
- 安装 CMake
- 掌握命令行
- 项目文件
- 脚本和 Find-模块

1.1. 示例下载

可以在这个章节的 GitHub 上找到出现的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch01>。

为了构建本书提供的示例，请执行推荐命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。正如将在本章中学到的，`<build tree>` 是输出目录的路径，而 `<source tree>` 是源代码所在的位置。

要构建 C++ 程序，还需要适合的编译器。如果熟悉 Docker，可以使用在“安装 CMake”部分介绍的完全工具化的镜像。如果愿意手动设置 CMake，我们将在同一部分介绍如何安装。

1.2. 基本知识

C++ 源代码的编译看起很直接，我们从经典的 Hello World 示例开始。

以下代码位于 ch01/01-hello/hello.cpp：

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

要生成一个可执行文件，需要一个 C++ 编译器。CMake 不包含编译器，所以需要选择并安装一个。主流的选择包括：

- Microsoft Visual C++ 编译器
- GNU 编译器集合
- Clang/LLVM

大多数读者都熟悉编译器，它是学习 C++ 时不可或缺的，所以我们不会深入讨论选择和安装。本书中的示例将使用 GNU GCC，它是一个成熟的开源软件编译器，可以在许多平台上免费获得。

假设已经安装了编译器，运行它对于大多数系统来说都类似：

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

我们的代码正确，编译器会默默地产生一个可执行的二进制文件。现在，来运行它：

```
$ ./hello
Hello World!
```

运行命令来构建你的程序很简单，但随着我们的项目增长，将所有内容放在一个文件中是不可能的。干净的代码实践建议，源代码文件应该保持小而有序。手动编译每个文件可能会变得脆弱。必须有更好的方法！

1.2.1 什么是 CMake？

我们通过编写一个脚本来自动化构建，该脚本遍历我们的项目树并编译所有内容。为了避免不必要的编译，脚本将检测源代码自上次运行脚本以来是否已修改。现在，希望有一种方便的方式来管理，传递给编译器的每个文件的参数——最好是基于可配置的标准。此外，脚本应该知道如何将所有编译文件，链接成一个单一的二进制文件。甚至更好地构建整个解决方案，这些解决方案可以作为更大的项目中的模块重复使用和集成。

软件构建是一个多样化的过程：

- 编译可执行文件和库
- 管理依赖关系
- 测试

- 安装
- 打包
- 生成文档
- 再测试

创建一个模块化和 C++ 构建工具，以适应各种目的，需要很长时间。Kitware 的 Bill Hoffman 在 20 多年前实现了 CMake 的第一个版本，它非常成功。今天，它具有许多功能和广泛的社区支持。CMake 正在积极开发中，并已成为 C 和 C++ 程序员行业标准。

自动化构建代码的问题比 CMake 还要古老，因此也有很多选择：GNU Make、Autotools、SCons、Ninja、Premake 等。但 CMake 为什么能脱颖而出？

有几方面我觉得很重要：

- 专注于支持现代编译器和工具链。
- 跨平台——支持为 Windows、Linux、macOS 和 Cygwin 构建。
- 为流行的 IDE 生成项目文件：Microsoft Visual Studio、Xcode 和 Eclipse CDT。此外，它是其他项目的项目模型，如 CLion。
- 在抽象层次上操作——允许将文件分组为可重用的目标和项目。
- 有大量使用 CMake 构建的项目，提供了一种简单的方式将它们集成到自己的项目中。
- 将测试、打包和安装视为构建过程的固有部分。
- 旧的、不常用的功能会废弃，以保持精简。

CMake 提供了一致、流畅的体验。无论是使用 IDE 还是直接从命令行构建软件，最重要的是也照顾到构建后的阶段。

CI/CD 流水线可以使用相同的 CMake 配置和构建项目，即使所有先前环境都不同。

1.2.2 工作原理

你可能认为 CMake 是一个工具，在一端读取源代码，在另一端生成二进制文件——尽管这在原则上是对的，但这并不是全部。

CMake 不能独立构建——它依赖于系统中的其他工具来执行实际的编译、链接等任务。你可以把它看作是构建过程的指挥家：它知道需要完成哪些步骤，最终目标是什么，以及如何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

让我们详细探讨它们。

配置阶段

这个阶段是关于读取存储在目录中的项目信息，称为源码树，并为生成阶段准备一个输出目录或构建树。

CMake 首先检查项目是否以前配置过，并从 `CMakeCache.txt` 文件中读取缓存的配置变量。首次运行时，它会创建一个空的构建树，并收集它正在工作的环境的所有详细信息：例如，架构是什么，可用的编译器是什么，以及是否安装了链接器和打包器。此外，还会检查一个简单的测试程序是否可以正确编译。

接下来，`CMakeLists.txt` 项目配置文件被解析并执行（CMake 项目使用 CMake 的编码语言进行配置）。这个文件是 CMake 项目的最小配置（源文件可以在稍后添加），告诉 CMake 关于项目结构、其目标和依赖项（库和其他 CMake 包）。

这个过程中，CMake 将收集的信息存储在构建树中，例如系统详情、项目配置、日志和临时文件，这些信息会用于下一阶段。具体来说，`CMakeCache.txt` 文件用于存储更稳定的信息（例如编译器和工具的路径），这在整个构建序列再次执行时可以节省时间。

生成阶段

阅读项目配置后，CMake 将为确切的环境生成一套构建系统。构建系统只是为其他构建工具（例如，GNU Make 的 `Makefiles` 或 Ninja，以及 Visual Studio 的 IDE 项目文件）定制的配置文件。在这个阶段，CMake 仍然可以通过计算生成器表达式对构建配置进行一些调整。

Tip

生成阶段在配置阶段之后自动执行。因此，本书和其他资料有时会交替使用这两个阶段，当提到“配置”或“生成”构建系统时。要明确只运行配置阶段，可以使用 `cmake-gui` 工具。

构建阶段

为了生成项目中指定的最终产品（如可执行文件和库），CMake 必须运行适当的构建工具。这可以通过直接调用、通过 IDE 或使用适当的 CMake 命令来执行。反过来，这些构建工具将执行一系列步骤，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产品。

这种解决方案的美丽之处在于，能够根据需要为每个平台生成构建系统（使用相同的项目文件）：

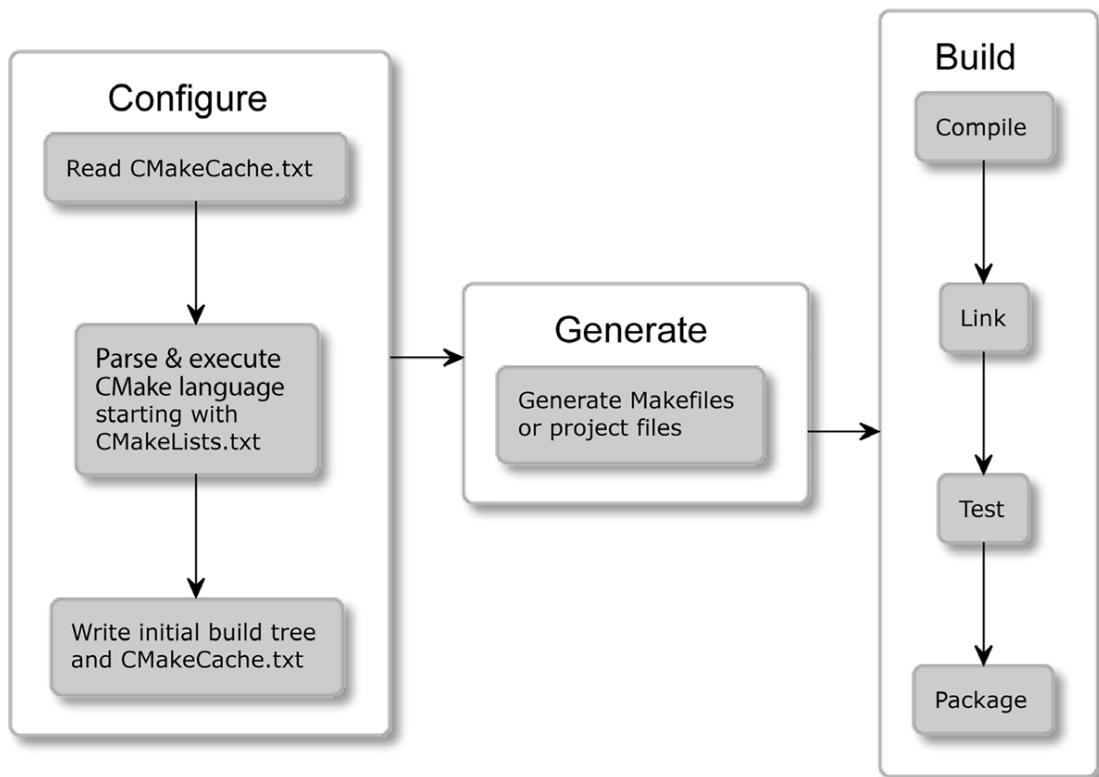


图 1.1: CMake 的各个阶段

还记得之前在“理解基础知识”部分中的 `hello.cpp` 应用程序吗？使用 CMake 构建它真的非常简单。只需要在源文件所在的同一目录下创建以下 `CMakeLists.txt` 文件。

`ch01/01-hello/CMakeLists.txt`

```

1 cmake_minimum_required(VERSION 3.26)
2 project(Hello)
3 add_executable(Hello hello.cpp)

```

创建此文件后，在该目录下执行以下命令：

```

cmake -B <build tree>
cmake --build <build tree>

```

请注意，`<build tree>` 是一个占位符，应该用一个临时目录的路径替换，该目录将包含生成的文件。

以下是在 Ubuntu 系统上运行的 Docker（Docker 是一个可以在其他系统上运行的虚拟机）的输出。

第一个命令生成构建系统：

```

~/examples/ch01/01-hello# cmake -B ~/build_tree
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info

```

```
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (1.0s)
-- Generating done (0.1s)
-- Build files have been written to: /root/build_tree
```

第二个命令构建项目：

```
~/examples/ch01/01-hello# cmake --build ~/build_tree
Scanning dependencies of target Hello
[ 50%] Building CXX object CMakeFiles/Hello.dir/hello.cpp.o
[100%] Linking CXX executable Hello
[100%] Built target Hello
```

最后，运行编译后的程序：

```
~/examples/ch01/01-hello# ./build_tree>Hello
Hello World!
```

这里，在构建树目录中生成了一个构建系统。接着，执行了构建阶段，并产生了一个可以运行的二进制文件。

现在知道结果是什么样子了，我敢肯定你会有很多问题：这个过程的先决条件是什么？这些命令是什么意思？为什么需要两个？如何编写自己的项目文件？别担心——这些问题将在接下来的部分中得到解答。

Note

本书将提供与当前版本的 CMake（撰写时为 3.26）相关的最重要信息。为了提供最佳建议，我明确避免了废弃和不推荐使用的功能，并强烈建议使用 CMake 3.15 或更新的版本（现代 CMake）。如果需要更多信息，可以在网上找到最新的完整文档，网址为 <https://cmake.org/cmake/help/>。

1.3. 安装 CMake

CMake 是一个跨平台的、开源的软件，用 C++ 编写。我们可以自己编译它；然而，最好不要这样做。可以从官方网站 <https://cmake.org/download/> 下载预编译的二进制文件。

基于 Unix 的系统可以直接从命令行提供准备好的安装包。

Note

CMake 并不包含编译器。如果系统还没有安装编译器，在使用 CMake 之前需要进行安装。确保将它们的执行文件路径添加到 PATH 环境变量中，这样 CMake 才能找到。为了避免在学习本书时遇到工具和依赖问题，我建议通过第一种安装方法进行实践：Docker。在现实世界的场景中，你当然会想要使用本地的版本，除非你最初就在虚拟化的环境中。

来看看 CMake 可以使用的环境。

1.3.1 Docker

Docker (<https://www.docker.com/>) 是一个跨平台的工具，提供操作系统级别的虚拟化，允许应用程序以定义良好的包形式（称为容器）进行传输。这些是自给自足的捆绑包，包含运行所需的所有库、依赖项和工具。Docker 在轻量级环境中执行其容器，彼此隔离。

这个概念使得分享完成特定过程所需的所有工具链变得极为方便，无需担心微小的环境差异。

Docker 平台有一个公共的容器镜像仓库，<https://registry.hub.docker.com/>，提供数百万个现成的镜像。

方便起见，我发布了两个 Docker 镜像：

- `swidzinski/cmake2:base`: 基于 Ubuntu 的镜像，包含构建时所需的精心挑选的工具和依赖项
- `swidzinski/cmake2:examples`: 基于上述工具链的镜像，包含本书中的所有项目和示例

第一个选项，是为那些想有一个干净的镜像来构建项目的读者准备，第二个选项是为我们在章节中进行示例实践而准备。

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

```
$ docker pull swidzinski/cmake2:examples
$ docker run -it swidzinski/cmake2:examples
root@b55e271a85b2:root@b55e271a85b2:#
```

请注意，示例位于以下格式的目录中

```
devuser/examples/examples/ch<N>/<M>-<title>
```

<N> 和 <M> 分别是零填充的章节和示例编号（例如 01, 08, 和 12）

1.3.2 Windows

Windows 上安装 CMake 非常简单——只需从官方网站下载适用于 32 位或 64 位的版本即可。还可以选择适用于 Windows Installer 的便携式 ZIP 或 MSI 包装包，它会将 CMake

的 bin 目录添加到 PATH 环境变量中（图 1.2），这样一来就可以在任何目录中使用它，而不会出现此类错误：

```
'cmake' 不是内部或外部命令，也不是可运行的程序或批处理文件。
```

如果选择 ZIP 包，需要手动安装。MSI 安装程序带有方便的 GUI：

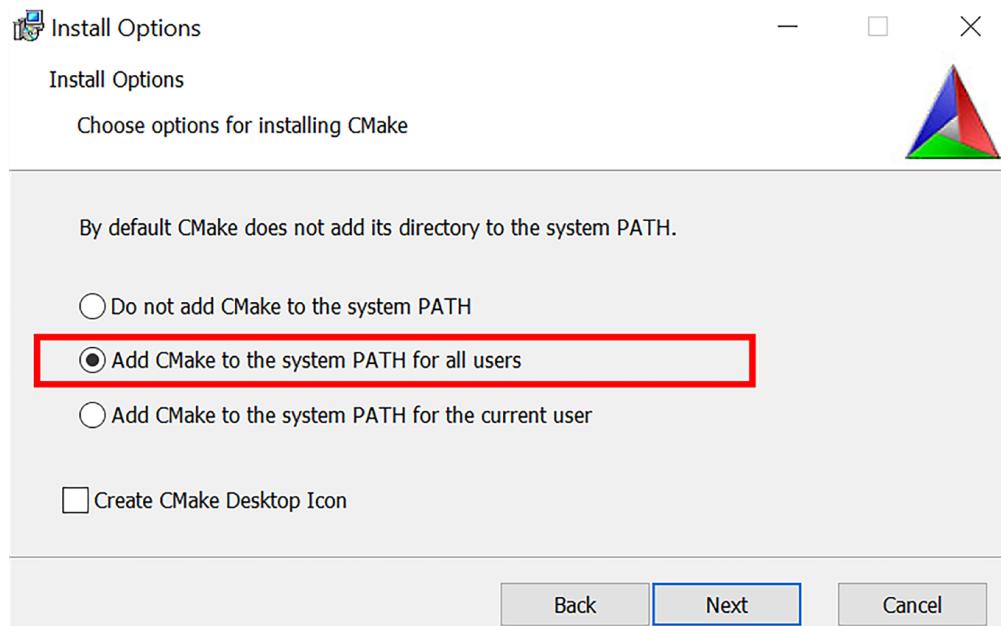


图 1.2：安装向导可以设置环境变量 PATH

这是一个开源软件，所以可以自己构建 CMake。然而，在 Windows 上，首先需要在系统上获取 CMake 的二进制文件，从而生成新版本的 CMake。

Windows 平台与其他平台没有什么不同，也需要一个构建工具来完成由 CMake 开始构建过程。这里的一个流行选择是 Visual Studio IDE，它附带了 C++ 编译器。社区版可以从 Microsoft 的网站免费获得：<https://visualstudio.microsoft.com/downloads/>。

1.3.3 Linux

Linux 上安装 CMake 的过程与安装其他软件包相同：命令行调用包管理器。包仓库通常会更新到 CMake 的最新版本，但通常不是最新版本。如果满意于此，并且使用像 Debian 或 Ubuntu 这样的发行版，最简单的方法就是直接安装适当的包：

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

对于 Red Hat 发行版，使用以下命令：

```
$ yum install cmake
```

Tip

当安装包时，包管理器将获取为操作系统配置的仓库中可用的最新版本。在许多情况下，包仓库不提供最新版本，而是提供经过时间考验的稳定版本，以确保可靠的工作。根据需求选择，旧版本可能不包含本书中描述的某些功能。

要获取最新版本，请参考官方 CMake 网站的下载部分。如果知道当前版本号，可以使用以下命令。

Linux x86_64 的命令：

```
$ VER=3.26.0 && wget https://github.com/Kitware/CMake/releases/download/v$VER/cmake-$VER-linux-x86_64.sh && chmod +x cmake-$VER-linux-x86_64.sh && ./cmake-$VER-linux-x86_64.sh
```

Linux AArch64 的命令：

```
$ VER=3.26.0 && wget https://github.com/Kitware/CMake/releases/download/v$VER/cmake-$VER-Linux-aarch64.sh && chmod +x cmake-$VER-Linux-aarch64.sh && ./cmake-$VER-Linux-aarch64.sh
```

或者，查看从源代码构建部分，了解如何相应的平台上自行编译 CMake。

1.3.4 macOS

这个平台也得到了 CMake 开发者的强烈支持。最流行的安装方法是通过 MacPorts：

```
$ sudo port install cmake
```

撰写本文时，MacPorts 中可用的最新版本是 3.24.4。要获取最新版本，请安装 cmake-devel 包：

```
$ sudo port install cmake-devel
```

或者，使用 Homebrew 包管理器：

```
$ brew install cmake
```

macOS 包管理器将涵盖所有必要步骤，除非从源代码构建，否则无法获得最新版本。

1.3.5 从源代码构建

如果使用其他平台，或者只是想体验尚未发布（或被你最喜欢的包仓库采用）的最新构建，请从官方网站下载源代码并自行编译：

```
$ wget https://github.com/Kitware/CMake/releases/  
download/v3.26.0/cmake-3.26.0.tar.gz  
$ tar xzf cmake-3.26.0.tar.gz  
$ cd cmake-3.26.0  
$ ./bootstrap  
$ make  
$ make install
```

从源代码构建相对较慢且需要更多步骤，没有其他方法可以自由选择 CMake 的版本。这对于系统包仓库中的包陈旧时特别有用：系统版本越旧，获得的更新就越少。

现在我们已经安装了 CMake，来看看怎么使用它吧！

1.4. 命令行

本书的大部分内容将教你，如何为用户提供准备 CMake 项目。为了满足其需求，需要彻底了解用户在不同场景中如何与 CMake 交互。这可以测试项目文件，并确保它们能正确工作。

CMake 是一组工具，包括五个可执行文件：

- `cmake`: 配置、生成和构建项目的核心可执行文件
- `ctest`: 运行和报告测试结果的测试驱动程序
- `cpack`: 生成安装程序和源包的打包程序
- `cmake-gui`: 窗口化的图形界面
- `ccmake`: 基于控制台的图形界面

此外，Kitware，CMake 的开发公司，还提供了一个名为 CDash 的单独工具，用于提供对我们项目构建健康状况的高级监控。

1.4.1 CMake 命令行

`cmake` 是 CMake 套件的主要二进制文件，并提供几种操作模式（有时也称为操作）：

- 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
- 运行工作流程预设
- 获取帮助

看看它们是如何工作的。

生成项目构建系统

构建项目的第一步是生成构建系统。以下是执行 CMake 生成项目构建系统操作的三种命令形式：

```
cmake [<options>] -S <source tree> -B <build tree>
cmake [<options>] <source tree>
cmake [<options>] <build tree>
```

目前，先关注选择正确的命令形式，讨论以下可用的 `<options>`。CMake 的一个重要特性是支持源外构建或支持，将构建工件存储在与源树不同的目录中。这是一种保持源目录干净，避免意外文件或忽略指令污染版本控制系统（VCS）的首选方法。

这就是为什么第一种命令形式最实用，允许指定源码树和生成的构建系统的路径，分别用 `-S` 和 `-B` 指定：

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

CMake 将从`./project` 目录读取项目文件，并在`./build` 目录中生成构建系统。

我们可以省略一个参数，`cmake` 会“猜测”我们打算使用当前目录作为省略的参数。省略`-S` 和 `-B` 参数将进行源内构建，并将构建工件与源文件一起存储，这是我们不想的。

运行 CMake 时要明确

不要使用第二种或第三种形式的 `cmake <directory>`，它们会产生混乱的源内构建。在第 4 章，设置第一个 CMake 项目，我们将学习如何防止用户这样做。

相同的命令如果之前在 `<directory>` 中已经存在构建，其行为会有所不同：会使用缓存的源路径并从此处重新构建。由于我们经常从终端命令历史记录中调用相同的命令，我们可能会在这里遇到麻烦；在使用这种形式之前，请始终检查你的 shell 是否在正确的目录中工作。

示例

使用来自上一目录的源在当前目录生成构建树：

```
cmake -S ..
```

使用当前目录的源在`./build` 目录生成构建树：

```
cmake -B build
```

选择生成器

可以在生成阶段指定几个选项。选择和配置生成器决定了，将使用系统中哪个构建工具来构建后续的构建项目部分，构建文件将是什么样子，以及构建树的结构将是什么样子。

所以，你应该关心生成器吗？答案通常是“不”。CMake 在许多平台上支持多种本地构建系统（除非同时安装了几个生成器），CMake 会选择一个。这可以通过设置 `CMAKE_GENERATOR` 环境变量或直接在命令行上指定生成器来覆盖：

```
cmake -G <generator name> -S <source tree> -B <build tree>
```

一些生成器（如 Visual Studio）支持对工具集（编译器）和平台（编译器或 SDK）进行更深入的指定。此外，CMake 将扫描环境变量以覆盖默认值：CMAKE_GENERATOR_TOOLSET 和 CMAKE_GENERATOR_PLATFORM。或者，值可以直接在命令行上指定：

```
cmake -G <generator name>
      -T <toolset spec>
      -A <platform name>
      -S <source tree> -B <build tree>
```

Windows 用户通常希望为其首选 IDE 生成一个构建系统。在 Linux 和 macOS 上，使用 Unix Makefiles 或 Ninja 生成器非常常见。

检查系统上可用的生成器，请使用以下命令：

```
cmake --help
```

在帮助信息的末尾，将获得一个生成器的完整列表，如下例所示，这是在 Windows 10 上产生的（为了便于阅读，部分输出截断了）：

以下是在此平台上可用的生成器：

```
Visual Studio 17 2022
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 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
```

CMake 支持许多不同风格的生成器和 IDE。

管理项目缓存

在配置阶段，CMake 会查询系统上的各种信息。这些操作可能需要一些时间，所以收集到的信息会缓存在构建树目录中的 `CMakeCache.txt` 文件里。有几个命令行选项可以有效的管理缓存的行为。

首先可以使用的是预填充缓存信息的能力：

```
cmake -C <initial cache script> -S <source tree> -B <build tree>
```

可以提供一个路径到 CMake 列表文件，这个文件（仅）包含一系列 `set()` 命令来指定用于初始化空构建树的变量。我们将在下一章讨论如何编写列表文件。

现有缓存变量的初始化和修改可以用另一种方式完成（当创建一个文件仅为设置几个变量有些过于繁琐时），可以直接在命令行中设置：

```
cmake -D <var>[:<type>]=<value> -S <source tree> -B <build tree>
```

`:<type>` 部分可选（由 GUI 使用）并且接受以下类型：BOOL, FILEPATH, PATH, STRING 或 INTERNAL。如果省略类型，CMake 会检查变量是否存在于 `CMakeCache.txt` 文件中并使用其类型；否则，将设置为 UNINITIALIZED。

我们经常通过命令行，设置的一个特别重要的变量是指定构建类型（`CMAKE_BUILD_TYPE`）。大多数 CMake 项目会在许多情况下，使用它来决定诸如诊断消息的详细程度、调试信息的存在，以及创建工作件的优化级别等情形。

对于单配置生成器（如 GNU Make 和 Ninja），应该在配置阶段指定构建类型，并为每种配置类型生成一个单独的构建树。

可以使用的值有 Debug, Release, MinSizeRel 或 RelWithDebInfo。缺少这些信息可能会对依赖它进行配置的项目产生未定义的影响。

这里是一个例子：

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

注意，多配置生成器在构建阶段进行配置。

为了诊断目的，还可以使用 `-L` 选项列出缓存变量：

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

有时，项目作者可能会在变量中提供有见地的帮助信息——要打印它们，请添加 `H` 修饰符：

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

令人惊讶的是，使用 `-D` 选项手动添加的自定义变量不会在这个打印输出中显示，除非指定了支持的类型。

可以使用以下选项删除一个或多个变量：

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

使用通配表达式支持 `*`（通配符）和`?`（任意字符）符号时要小心，因为很容易意外删除比预期更多的变量。

`-U` 和 `-D` 选项都可以重复多次使用。

调试和跟踪

`cmake` 命令可以使用多种选项来让了解其内部工作。要获取关于变量、命令、宏和其他设置的通用信息，请运行以下命令：

```
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，将打印有关缓存变量和日志文件中的构建消息的相关信息。

在项目中，我们使用 `message()` 命令来报告构建过程的详细信息。CMake 会根据当前的日志级别（默认为 `STATUS`）过滤这些日志输出。以下行指定了感兴趣的日志级别：

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

级别可以是以下任一选项：`ERROR`, `WARNING`, `NOTICE`, `STATUS`, `VERBOSE`, `DEBUG` 或 `TRACE`。也可以在 `CMAKE_MESSAGE_LOG_LEVEL` 缓存变量中指定此设置。

另一个有趣的选项允许在每个 `message()` 调用中显示日志上下文。为了调试非常复杂的项目，`CMAKE_MESSAGE_CONTEXT` 变量可以像栈一样使用。每当代码进入一个有趣的上下文时，就可以给它一个描述性的名称。这样做之后，消息将使用当前的 `CMAKE_MESSAGE_CONTEXT` 变量装饰，如下所示：

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

启用这种日志输出的选项如下：

```
cmake --log-context <source tree>
```

我们将在第 2 章中详细讨论命名上下文和日志命令。

如果其他所有方法都失败了，就需要使用重型武器——跟踪模式，它会打印每个执行的命令及其文件名、调用它的行号，以及传递的参数列表。可以按以下方式启用：

```
cmake --trace
```

不建议日常使用，因为输出非常长。

预设配置

有许多选项可供用户指定，以从您的项目生成构建树。当涉及到构建树路径、生成器、缓存和环境变量时，很容易感到困惑或遗漏某些细节。开发人员可以通过提供一个 `CMakePresets.json` 文件来简化用户与项目的交互方式，并在此文件中指定一些默认设置。

要列出所有可用的预设，请执行以下命令：

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

可以使用其中一个可用的预设：

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

要了解更多信息，请参阅本章的“项目文件”部分，以及第 16 章的内容。

清理构建树

有时可能需要删除生成的文件。这可能是由于构建之间环境发生了一些变化，或者只是为了确保从头开始工作。我们可以手动删除构建树目录，或在命令行中添加 `--fresh` 参数：

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

CMake 将以系统无关的方式删除 `CMakeCache.txt` 和 `CMakeFiles/` 文件夹，并从头开始重新生成构建系统。

构建项目

生成构建树之后，就可以进行项目的构建操作了。CMake 不仅知道如何为许多不同的构建工具生成输入文件，还可以根据项目需求为运行这些工具，并提供适当的参数。

避免直接调用 `make`

许多在线资料建议在生成阶段后，直接通过 `make` 命令运行 GNU Make。因为 GNU Make 是 Linux 和 macOS 的默认生成器，所以这个建议可以奏效。然而，建议使用本节描述的方法，其不依赖于特定的生成器，并且在所有平台上都得到官方支持。因此，无需担心应用程序每个用户的精确环境。

构建模式的语法如下：

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

大多数情况下，只需要提供最基本的参数即可构建：

```
cmake --build <build tree>
```

唯一必需的参数是生成的构建树的路径，这与生成阶段中使用 `-B` 参数传递的路径相同。

如果需要为所选的本地构建工具提供特殊参数，可以在命令末尾的 `--` 符号后传递：

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

看看还有哪些其他选项可用。

运行并行构建

默认情况下，许多构建工具都会使用多个并发进程来利用现代处理器，并行编译源代码。构建工具了解项目依赖结构，可以同时处理满足依赖关系的任务，以节省用户的时间。

如果希望在多核机器上更快地构建（或为了调试而强制单线程构建），可以覆盖该设置。

只需使用以下任一选项指定作业数量：

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

或者，可以使用 `CMAKE_BUILD_PARALLEL_LEVEL` 环境变量进行设置。命令行选项将会覆盖此变量。

选择要构建和清理的目标

每个项目由一个或多个称为目标的部分组成（将在本书第二部分讨论这些）。通常会想要构建所有可用的目标；但是偶尔，可能对跳过某些目标或明确排除在正常构建之外的目标感兴趣：

```
cmake --build <build tree> --target <target1> --target <target2> ...
```

可以通过重复 `--target` 参数来指定要构建的多个目标。此外，也可以使用简写形式 `-t` `<target>`。

清理构建树

有一个特殊的目标叫做 `clean`。构建它会产生特殊的效果，即从构建目录中移除所有产物，以便稍后可以从头开始创建。

可以这样启动清理过程：

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

另外，CMake 提供了一个方便的别名，如果想先清理再执行正常的构建：

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

这个操作与“清理构建树”部分中提到的清理不同，它只影响目标产物而不影响其他（例如缓存）。

为多配置生成器配置构建类型

我们已经对生成器有了一定的了解：它们有不同的类型和功能。其中一些生成器能够在一个构建树中同时构建 Debug 和 Release 构建类型，支持此功能的生成器包括 Ninja Multi-Config、Xcode 和 Visual Studio。其他每个生成器都是单一配置生成器，需要为每种配置类型单独构建树。

选择 Debug、Release、MinSizeRel 或 RelWithDebInfo 并指定如下：

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

否则，CMake 将使用 Debug 作为默认值。

调试构建过程

出现问题时，应该首先检查输出信息。然而，经验丰富的开发者知道，打印所有细节会令人困惑，所以通常默认隐藏这些细节。当需要查看底层情况时，可以要求 CMake 提供更详细的日志：

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

同样效果也可以通过设置 CMAKE_VERBOSE_MAKEFILE 缓存变量实现。

安装项目

当构建产物完成后，用户可以将它们安装到系统中。这意味着将文件复制到正确的目录，安装库或运行 CMake 脚本中的自定义安装逻辑。

安装模式的语法如下：

```
cmake --install <build tree> [<options>]
```

像其他操作模式一样，CMake 需要生成的构建树的路径：

```
cmake --install <build tree>
```

安装动作也有许多选项。

选择安装目录

可以使用自选的前缀来预置安装路径（当对某些目录的写入权限有限时）。例如，如果 /usr/local 路径前缀为 /home/user，则变为 /home/user/usr/local。

命令如下所示：

```
cmake --install <build tree> --install-prefix <prefix>
```

如果只用 CMake 3.21 或之前的版本，命令如下所示：

```
cmake --install <build tree> --prefix <prefix>
```

多配置生成器

就像在构建阶段一样，可以指定用于安装的构建类型（更多详情请参阅“构建项目”部分）。可用类型包括 Debug、Release、MinSizeRel 和 RelWithDebInfo：

```
cmake --install <build tree> --config <cfg>
```

选择组件

作为开发者，可能会选择将项目拆分为可以独立安装的组件。现在，假设有一组不需要在所有情况下使用的产物。这可能类似于 application、docs 和 extra-tools。

要安装单个组件，使用以下选项：

```
cmake --install <build tree> --component <component>
```

设置权限

如果安装在类 Unix 平台上，可以使用以下选项指定安装目录的默认权限，格式为 u=rwx, g=rwx, o=rwx：

```
cmake --install <build tree>
--default-directory-permissions <permissions>
```

调试

与构建阶段类似，也可以选择查看安装阶段的详细输出：

```
cmake --install <build tree> --verbose
cmake --install <build tree> -v
```

如果设置了 VERBOSE 环境变量，也能达到同样的效果。

执行脚本

项目使用 CMake 自定义语言配置，跨平台并且相当强大。既然已经有了这样的语言，为什么不将其用于其他任务呢？当然，CMake 可以运行独立的脚本，如下所示：

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

运行此类脚本不会进行配置或生成，也不会影响缓存。

有两种方法可以将值传递给此脚本：

- 通过使用 -D 选项定义的变量
- 通过可以在 -- 符号后传递的参数

CMake 将为传递给脚本的所有参数（包括--）创建 CMAKE_ARGV<n> 变量。

命令行工具

少数情况下，我们需要以平台无关的方式运行单个命令——例如复制文件或计算校验和。并非所有的平台都完全相同，因此并非所有命令在每个系统中都可用（或者它们有不同的名称）。

CMake 提供了一种模式，大多数常见命令都可以跨平台地以相同方式执行。其语法如下：

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

由于这种特定模式的应用相对有限，我们不会深入介绍。但是，如果对此感兴趣，我建议您通过运行 `cmake -E` 来列出所有可用的命令。为了快速了解提供的内容，CMake 3.26 支持以下命令：`capabilities`, `cat`, `chdir`, `compare_files`, `copy`, `copy_directory`, `copy_directory_if_different`, `copy_if_different`, `echo`, `echo_append`, `env`, `environment`, `make_directory`, `md5sum`, `sha1sum`, `sha224sum`, `sha256sum`, `sha384sum`, `sha512sum`, `remove`, `remove_directory`, `rename`, `rm`, `sleep`, `tar`, `time`, `touch`, `touch_nocreate`, `create_symlink`, `create_hardlink`, `true` 和 `false`。

如果缺少想要使用的命令，或需要更复杂的行为，可以考虑将它封装在脚本中，使用 `-P` 运行。

工作流预设

使用 CMake 构建项目分为三个阶段：配置、生成和构建。此外，还可以使用 CMake 运行自动化测试，甚至创建可重分配的包。通常，用户需要通过命令行单独手动执行每一个步骤。但是，高级项目可以指定工作流预设，将多个步骤捆绑成单一动作，只需一个命令即可执行。目前，只需知道用户可以通过运行以下命令，获取可用预设的列表：

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

可以执行一个工作流预设：

```
cmake --workflow --preset <name>
```

这将在第 16 章中详细介绍。

获取帮助

CMake 提供了广泛的帮助信息，可以通过其命令行访问。

帮助模式的语法如下：

```
cmake --help
```

这将输出一系列主题列表，以深入了解，并解释需要向命令添加哪些参数，以获取更多帮助。

1.4.2 CTest

为了生产高质量的代码并维护其质量，自动化测试是非常重要的。CMake 套件包含了一个用于此目的的命令行工具 CTest，旨在标准化测试的执行和报告方式。作为 CMake 用户，不需要了解特定项目测试的详细信息：使用了哪个框架或如何运行。CTest 提供了一个方便的接口来列出、过滤、随机化、重试和限制测试运行的时间。

要为已构建的项目运行测试，只需在生成的构建树中调用 `ctest`：

```
$ ctest
Test project /tmp/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
```

我们为这一主题专门编写了一整章：第 11 章。

1.4.3 CPack

构建并测试完我们优秀的软件后，我们就准备将其分享给全世界。少数高级用户可以完全接受源代码，但绝大多数用户出于方便和节省时间的原因使用预编译的二进制文件。

CMake 自带了一系列工具。CPack 是一个工具，可以为各种平台创建可重分配的包：压缩归档文件、可执行安装程序、向导、NuGet 包、macOS 应用程序包、DMG 包、RPM 包等。

CPack 的工作方式与 CMake 非常相似：使用 CMake 语言进行配置，并提供了许多包生成器供选择（不要与 CMake 构建系统生成器混淆）。

这个工具为成熟的 CMake 项目设计，我们将在第 14 章中详细介绍这个工具。

1.4.4 CMake GUI

Windows 版本的 CMake 配备了一个图形用户界面（GUI），用于配置先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 通过 `cmake-qt-gui` 包获取。

要访问 CMake GUI，运行 `cmake-gui` 可执行文件：

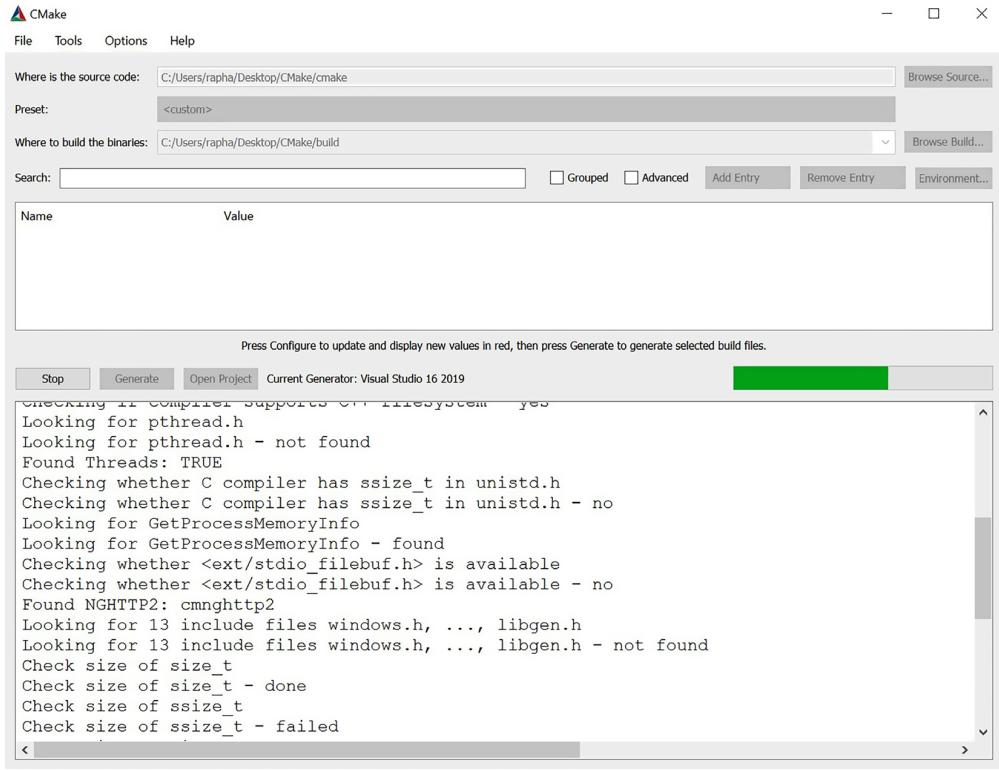


图 1.3: CMake GUI — 使用 Visual Studio 2019 生成器配置构建系统的阶段

GUI 应用程序为您的应用程序用户提供了便利：对于那些不熟悉命令行并更喜欢图形界面的人来说，它很有用。

推荐使用命令行工具

我肯定会推荐 GUI 给最终用户，但对于各位开发者，我建议避免手动阻塞步骤。这对于成熟的项目尤其有利，因为整个构建过程可以完全无需交互就能执行。

1.4.5 ccmake

ccmake 可执行文件是在类 Unix 平台上的 CMake 的交互式文本用户界面（在 Windows 上不可用，除非显式构建）（图 1.4），像 GUI 一样。

```
root@deb94335c57f: ~
Page 1 of 1
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX
/usr/local

CMAKE_BUILD_TYPE: Choose the type of build, options are: None Debug Release RelWithDebInfo MinSizeRel ...
Press [enter] to edit option Press [d] to delete an entry
Press [c] to configure
Press [h] for help      Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

图 1.4: ccmake 中的配置阶段

了解了这些之后，我们已经完成了对 CMake 套件基本命令行介绍。现在是时候探索 CMake 项目的结构了。

1.5. 项目结构

CMake 项目由大量的文件和目录构成。来大致了解一下每个组成部分的作用，以便进行调整。这些文件和目录可以分为几个类别：

- 项目文件，作为开发者会随着项目的成长修改这些文件。
- CMake 生成的文件，包含 CMake 语言命令，但并不供开发者编辑。任何手动更改都将被覆盖。
- 高级用户（即：非项目开发者）使用的文件，以便根据个人需求自定义 CMake 构建项目的方式。
- 一些临时文件，在特定上下文中提供有价值的信息。

本节还会建议哪些文件加入到版本控制系统（VCS）的忽略文件中。

1.5.1 源代码目录

这是项目的存放目录（也称为项目根目录），包含了所有的 C++ 源文件和 CMake 项目文件。以下是关于此目录的一些信息：

- 需要一个 CMakeLists.txt 配置文件。
- 此目录的路径通过用户在生成构建系统时，传递给 cmake 命令的 -S 参数。
- 避免在 CMake 代码中，硬编码源码目录的绝对路径——软件用户将会在另一个路径中存储项目。

在这个目录中初始化一个仓库是个好主意，可以使用 Git 这样的版本控制系统。

1.5.2 构建目录

CMake 根据用户的指定，在此目录中创建构建系统，以及构建过程中产生的一切：项目制品、临时配置、缓存、构建日志和原生构建工具（如 GNU Make）的输出。这个目录的其他名称，包括构建根目录和二进制目录。

需要记住的是：

- 构建配置（构建系统）和构建制品（如二进制文件、可执行文件和库，以及用于最终链接的对象文件和归档文件）将在此处创建。
- CMake 推荐将此目录置于源代码目录之外（即外源构建），可以避免污染项目（即内源构建）。
- 通过 -B 参数传递给 cmake 命令来指定。
- 这个目录不是生成文件的最终目的地。相反，建议项目包含一个安装阶段，将最终制品复制到系统中的适当位置，并移除所有用于构建的临时文件。

不要将此目录加入到版本控制系统（VCS）中——每个用户都会为自己选择一个。如果充分的理由进行内源构建，请确保将此目录加入到版本控制系统的忽略文件中（如`.gitignore`）。

1.5.3 列表文件

包含 CMake 语言的文件称为列表文件（listfiles），可以通过调用 `include()` 和 `find_package()` 或间接地通过 `add_subdirectory()` 来相互包含。CMake 对这些文件的命名没有强制规定，但按照惯例通常使用`.cmake` 扩展名。

项目文件

CMake 项目通过一个 `CMakeLists.txt` 列表文件进行配置（注意，由于历史原因，该文件具有非常规的扩展名）。此文件位于每个项目的源代码树的顶部，并且是在配置阶段首先执行。

顶层的 `CMakeLists.txt` 文件应至少包含以下两个命令：

- `cmake_minimum_required(VERSION <x.xx>)`: 设置期望的 CMake 版本，并告诉 CMake 如何处理遗留行为。
- `project(<name> <OPTIONS>)`: 命名项目（提供的名称将存储在 `PROJECT_NAME` 变量中），并指定配置项目的选项。

随着软件的增长，可能希望将其划分为较小的单元，这些单元可以单独配置和管理。CMake 支持通过子目录及其各自的 `CMakeLists.txt` 文件来实现这一点。

项目结构可能如下所示：

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

一个简单的顶层 CMakeLists.txt 文件可以用来整合整个项目：

```
1 cmake_minimum_required(VERSION 3.26)
2 project(app)
3 message("Top level CMakeLists.txt")
4 add_subdirectory(api)
```

项目的主要方面都在顶级文件中覆盖：管理依赖项、声明要求和检测环境。我们还使用 add_subdirectory(api) 命令来包含来自 api 子目录的另一个 CMakeLists.txt 文件，以执行与应用的 API 部分相关的特定步骤。

缓存文件

从列表文件中生成的缓存变量，将在首次运行配置阶段时被存储在 CMakeCache.txt 文件中。此文件位于构建树的根目录，并具有相当简单的格式（为简洁起见，省略了部分内容）：

```
# This is the CMakeCache file.
# For build in directory: /root/build tree
# It was generated by CMake: /usr/local/bin/cmake
# 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 ...
```

从注释中可以看出，EXTERNAL 部分的缓存条目是供用户修改的，而 INTERNAL 部分则由 CMake 管理。

以下是几个需要牢记的点：

- 可以手动管理此文件，也可以通过调用 cmake，或通过 ccmake 或 cmake-gui 来管理。
- 通过删除此文件可以将项目重置为其默认配置；文件将从列表文件中重新生成。

可以从列表文件读取和写入缓存变量，变量引用的计算会有些复杂。

包定义文件

CMake 生态系统的一个大部分是项目可以依赖的外部包，以无缝、跨平台的方式提供库和工具。希望提供 CMake 支持的包作者会将 CMake 包配置文件一起发布。

以下是几个注意的细节：

- 配置文件（原始拼写）包含有关如何使用库的二进制文件、头文件和辅助工具的信息。有时，会暴露在项目中使用的 CMake 宏和函数。
- 配置文件命名为 `<PackageName>-config.cmake` 或 `<PackageName>Config.cmake`。
- 使用 `find_package()` 命令来包含包。

需要特定版本的包，CMake 会将此与相关的 `<PackageName>-config-version.cmake` 或 `<PackageName>ConfigVersion.cmake`。

如果供应商没有为包提供配置文件，配置会与 CMake 本身捆绑，或者可以在项目中通过 Find-模块提供。

生成的文件

在生成阶段，许多文件由 `cmake` 可执行文件在构建树中生成。CMake 将它们用作 `cmake` 安装动作、CTest 和 CPack 的配置。

可能会看到：

- `cmake_install.cmake`
- `CTestTestfile.cmake`
- `CPackConfig.cmake`

如果正在实现源内构建，将它们添加到版本控制系统的忽略文件中吧。

1.5.4 JSON 和 YAML 文件

CMake 使用的其他格式包括 JavaScript 对象表示法 (JSON) 和另一种标记语言 (YAML)。这些文件作为与外部工具（如 IDE）通信的接口，或者提供易于生成和解析的配置。

预设文件

当需要具体指定缓存变量、选择的生成器、构建树路径等时，项目的高级配置可能会成为一个相对繁重的任务——尤其是有不止一种方式来构建项目时。这时预设就派上用场了——不需要通过命令行手动配置这些值，只需提供一个存储所有细节的文件，并与项目一起发布。自从 CMake 3.25 版本以来，预设还允许配置工作流程，将阶段（配置、构建、测试和打包）绑定到一个命名步骤列表中执行。

用户可以通过 GUI 选择预设，或者使用命令 `--list-presets` 并使用 `--preset=<preset>` 选项为构建系统选择预设。

预设存储在两个文件中：

- `CMakePresets.json`: 这是供项目作者提供官方预设的文件。

- `CMakeUserPresets.json`: 这是专供想要根据个人喜好自定义项目配置的用户使用的文件（可以将其添加到版本控制系统忽略文件中）。

预设不是项目必需的，只在高级场景中变得有用。

基于文件的 API

CMake 3.14 引入了一个 API，允许外部工具查询构建系统信息：生成文件的路径、缓存条目、工具链等。这里，只提到这个非常高级的话题，以避免在文档中遇到“基于文件的 API”这个术语时产生混淆。这个名字表明了它的工作方式：一个包含查询的 JSON 文件，必须放置在构建树内的一个特殊路径中。CMake 在生成构建系统时，会读取这个文件，并将响应写入另一个文件，以便外部应用程序解析。

基于文件的 API 是为了替换一种在 CMake 3.26 版本中废弃的机制，称为服务器模式（或 `cmake-server`）。

配置日志

自从 3.26 版本以来，CMake 将为配置阶段的深度调试提供一个结构化的日志文件，位置在：

```
<build tree>/CMakeFiles/CMakeConfigureLog.yaml
```

这是一个不需要特别注意的功能。

1.5.5 Git 中忽略文件

有许多版本控制系统；其中最受欢迎的是 Git。每当我们开始一个新项目时，确保只将必要的文件添加到仓库是很好的做法。如果指定了不需要的文件在 `.gitignore` 文件中，项目卫生更容易维护。例如，可能会排除生成的、特定于用户的或临时的文件。

Git 在形成新提交时会自动跳过。以下是我项目中使用的文件：

ch01/01-hello/.gitignore

```
CMakeUserPresets.json
# If in-source builds are used, exclude their output 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
```

现在掌握了一张项目文件的导航图。有些文件非常重要，将经常使用它们——其他的则不那么重要。虽然学习它们可能看起来像是浪费时间，但知道在哪里找不到答案可能非常有价值。无论如何，本章的最后一个问题是：还可以用 CMake 创建哪些自包含的单元？

1.6. 脚本和模块

CMake 主要是专注于构建项目以产生其他系统（如 CI/CD 和测试平台，或部署到机器上或存储在工件仓库中）所消耗的工件。然而，有两种其他概念也使用 CMake 语言：脚本和模块。让我们介绍一下它们是什么，以及它们之间的区别。

1.6.1 脚本

CMake 提供了一种与平台无关的编程语言，并附带了许多有用的命令。用这种语言编写的脚本可以与更大的项目捆绑在一起，也可以完全独立。

将其视为一种一致的跨平台工作方式。通常，为了执行任务，需要为 Linux 创建一个单独的 Bash 脚本，为 Windows 创建单独的批处理文件或 PowerShell 脚本等。CMake 抽象了这些，可以使用一个文件在所有平台上正常工作。当然，可以使用 Python、Perl 或 Ruby 等外部工具的脚本，但这会增加依赖性，并增加 C/C++ 项目的复杂性。既然大多数时候可以使用更简单的东西来完成工作，为什么要引入另一种语言呢？使用 CMake！

可以使用 -P 选项执行脚本：`cmake -P 脚本.cmake`。

但使用脚本文件的实际要求是什么呢？并不多：脚本可以很复杂，或者只是一个空文件。不过，仍然建议在每一个脚本的开头调用 `cmake_minimum_required()` 命令，以告诉 CMake 应该对这个项目后续的命令应用哪些策略。

以下是一个简单脚本的例子：

`ch01/02-script/script.cmake`

```
1 # An example of a script
2 cmake_minimum_required(VERSION 3.26.0)
3 message("Hello world")
4 file(WRITE Hello.txt "I am writing to a file")
```

运行脚本时，CMake 不会执行常规阶段（如配置或生成），也不会使用缓存，脚本中没有源代码树或构建树的概念。所以，在脚本模式下，项目特定的 CMake 命令不可用/不可使用。

1.6.2 工具模块

CMake 项目可以使用外部模块来增强其功能。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。它们从相当复杂的脚本（如 CPack 和 CTest 提供的脚本）到相对简单的脚本，如 `AddFileDependencies` 或 `TestBigEndian`。

CMake 发行版打包了超过 80 个不同的实用模块。如果这还不够，可以通过浏览精选列表（如 <https://github.com/onqtam/awesome-cmake>）从互联网上下载更多，或者自己编写模块。

要使用工具模块，需要 `include()` 命令。以下是展示这一操作的项目示例：

ch01/03-module/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(ModuleExample)
3 include (TestBigEndian)
4 test_big_endian(IS_BIG_ENDIAN)
5 if(IS_BIG_ENDIAN)
6     message("BIG_ENDIAN")
7 else()
8     message("LITTLE_ENDIAN")
9 endif()
```

我们将在它们与主题相关时，可以了解哪些模块可用。如果非常好奇，可以在 <https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html> 找到打包模块的完整列表。

1.6.3 Find-模块

在“包定义文件”部分，我提到了 CMake 有一种机制来查找属于不支持 CMake，且不提供 CMake 包配置文件的 external 依赖项的文件。这就是 Find-模块的作用。CMake 提供了超过 150 个 Find-模块，能够定位在系统上安装的这些包。与实用模块一样，网上还有更多的 Find-模块可用。

可以通过调用 `find_package()` 命令，并提供相关包的名称进行使用。这样的 Find-模块将玩一个捉迷藏游戏，并检查它所寻找的软件的所有已知位置。如果找到了文件，将定义包含其路径的变量（如该模块手册中所述）。现在，CMake 可以针对该依赖项进行构建。

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

1.7. 总结

现在已经了解了 CMake 是什么，以及它是如何工作的；了解了 CMake 工具家族的关键组成部分，以及它在各种系统上的安装方式。了解了所有通过命令行运行 CMake 的方式：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。知道了 CTest、CPack 和 GUI 应用程序。这将帮助你，以正确的视角为用户和其他开发者创建项目。此外，了解了项目由什么组成：目录、列表文件、配置、预设和帮助文件，以及在版本控制系统中应该忽略的内容。最后，知晓了其他非项目文件：独立的脚本和两种类型的模块——工具模块和 Find-模块。

下一章中，我们将了解如何使用 CMake 编程语言。将编写自己的列表文件，并为编写第一个脚本、项目和模块打开大门。

1.8. 扩展阅读

需要获取更多信息，可以参考以下资料：

- 官方 CMake 网页和文档：

<https://cmake.org/>

- 单配置生成器：

<https://cgold.readthedocs.io/en/latest/glossary/single-config.html>

- CMake GUI 中的阶段分离：

<https://stackoverflow.com/questions/39401003/>

第 2 章 CMake 语言

在 CMake 语言中编写代码比预期的要棘手。当你第一次阅读 CMake 列表文件时，可能会觉得里面的语言如此简单，以至于可以不需要理论就能实践。然后可能会试图进行更改并尝试代码，而没有彻底了解它实际上是如何工作的。开发者通常都很忙，而且与构建相关的问题通常不是那种让人愿意投入大量时间的事情。为了快速完成，我们倾向于基于直觉进行更改，希望它们可能会奏效。这种解决技术问题的方法称为**巫毒编程**。

CMake 语言看起来微不足道：在我们引入了扩展、修复、`hack` 或单行代码之后，突然发现有些东西不工作了。通常，调试的时间会超过理解本身所需的时间。幸运的是，这不会是我们的命运，因为这一章包含了实践中使用 CMake 语言所需的大部分知识。

本章中，不仅会学习 CMake 语言的基本构建块——注释、命令、变量和控制结构——还将了解必要的背景知识，并遵循最新实践尝试示例。

CMake 会让你处于一个独特的位置。一方面，扮演着构建工程师的角色，必须全面掌握编译器、平台及其所有相关方面。另一方面，是一个编写生成构建系统的代码的开发者。编写高质量的代码是一项具有挑战性的任务，需要多方面的方法。代码不仅要有功能性和可读性，还应该易于分析、扩展和维护。

最后，将介绍 CMake 中最实用和最常用的命令。也经常使用，但程度不如前面的命令将在附录“其他命令”（字符串、列表、文件和数学命令的参考指南）中介绍。

本章，将包括以下内容：

- 语法基础
- 变量操作
- 使用列表
- 控制结构
- 常用命令

2.1. 示例下载

可以在 GitHub 上找到本章中出现的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch02>。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。作为提醒：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的位置。

2.2. 语法基础

编写 CMake 代码类似于其他命令式语言：从上到下、从左到右逐行执行，偶尔会进入包含的文件或调用的函数。执行的起点取决于模式，可以是从源码树的根文件 (`CMakeLists.txt`) 开始，也可以是从作为 `cmake` 命令参数提供的 `.cmake` 脚本文件开始。

由于 CMake 脚本提供了对 CMake 语言的广泛支持（除了与项目相关的特性），我们将利用它们来练习本章中的 CMake 语法。当我们掌握了编写简单的列表文件，就可以进一步创建实际的项目文件。

脚本可以通过以下命令运行：`cmake -P script.cmake`

Note

CMake 支持 7 位 ASCII 文本文件，以便在所有平台之间移植，可以使用 `\n` 或 `\r\n` 行结束符。版本高于 3.2 的 CMake，支持带有可选字节序标记（BOM）的 UTF-8 和 UTF-16。

CMake 列表文件中的所有内容要么是注释，要么是命令。

2.1.1 注释

与 C++ 一样，有两种类型的注释：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。单行注释以井号符号开始，#：

```
1 # they can be placed on an empty line
2 message("Hi"); # or after a command like here.
```

多行括号注释的名字来源于它们的符号——以 # 开始，后跟开方括号 [，任意数量的等号 =（也可以是 0 个），以及另一个方括号 [. 要关闭括号注释，请使用相同数量的等号并反转方括号]：

```
1 #[=
2 bracket comment
3 #[[
4     nested bracket comment
5 ]]
6 #]=]
```

可以通过在括号注释的初始行，添加另一个 # 来快速禁用多行注释：

```
1 ###[ this is a single-line comment now
2 no longer commented
3 #[[
4     still, a nested comment
5 ]]
6 #]=] this is a single-line comment now
```

了解如何使用注释肯定有用，但这又引出了另一个问题：应该在什么时候使用？由于编写列表文件本质上就是编程，将我们最好的编码实践带到其中。

遵循这些实践的代码通常称为**整洁代码**——这个术语多年来被软件开发大师如 Robert C. Martin、Martin Fowler，以及许多其他作者使用。

围绕哪些实践是有益或有害，常常存在很多争议，注释也没有免于这些辩论。一切应该根据具体情况来判断，但普遍认同的指导原则认为，好的注释至少提供以下一项：

- 信息：可以解开复杂性的问题，如正则表达式模式或格式化字符串。
- 意图：当代码的意图从实现或接口中不明显时，可以解释代码意图。
- 澄清：可以解释难以重构或更改的概念。
- 警告后果：可以提供警告，尤其是在可能破坏其他事物时。
- 强调：可以强调难以在代码中表达的重要想法。
- 法律条款：有必要的时候需要添加，但这通常不是开发者擅长的领域。

最好通过应用更好的命名、重构或纠正代码来避免注释。省略以下注释：

- 强制性：这些是为了完整性而添加的，但并不是很重要。
- 冗余：这些重复了代码中已经清晰写明的内容。
- 误导性：如果不跟随代码更改，这些可能是过时或不正确的。
- 日志：这些记录了更改了什么，以及何时更改（应使用版本控制系统（VCS）代替）。
- 分隔符：只用于标记。

避免添加注释，采用更好的命名实践，并重构或纠正代码。编写优雅的代码是一项具有挑战性的任务，但它增强了读者的体验。由于阅读代码的时间比编写代码的时间多，应始终努力编写易于阅读的代码，而不仅仅是试图快速完成它。我建议查看本章末尾的“扩展阅读”，那里有一些关于整洁代码的好参考资料。如果对注释感兴趣，可在链接到我的 YouTube 视频“Which comments in your code ARE GOOD?”，其中深入探讨这一内容。

2.1.2 使用注释

使用命令是 CMake 列表文件的核心。为了运行一个命令，必须指定它的名字，然后是括号，在括号内可以包含由空格分隔的命令参数列表。

```
message("hello" world)
      ^          ^
      name       arguments
```

图 2.1：命令示例

命令名称不区分大小写，但在 CMake 社区中有一个约定，使用 `snake_case`（即用下划线连接的小写单词）。还可以定义自己的命令。

CMake 和 C++ 之间的一个重要区别是，CMake 中的命令调用不是表达式。所以不能将另一个命令作为参数传递给调用的命令，因为括号内的所有内容都视为特定命令的参数。

CMake 命令后也不需要加分号。这是因为源代码的每一行只能包含一个命令。

命令后面可以（可选地）跟一个注释：

```
1 command(argument1 "argument2" argument3) # comment
2 command2() #[[ multiline comment
```

但不能反着来：

```
1 #[[ bracket
2 ]] command()
```

CMake 列表文件中的所有内容要么是注释，要么是命令调用。CMake 语法真的很简单，在大多数情况下，这是一件好事。虽然有一些限制（不能使用表达式来递增计数器变量），但大部分情况下，因为 CMake 并不打算成为一种通用编程语言，这些限制可以接受。

CMake 提供了操作变量、控制执行流程、修改文件等的命令。为了简化操作，我们将在不同的示例中逐步介绍相关的命令。这些命令可以分为两组：

- 脚本命令：这些命令始终可用，改变命令处理器的状态，访问变量，并影响其他命令和环境。
- 项目命令：这些命令在项目中可用，操纵项目状态和构建目标。

每个命令都依赖于语言的其它元素才能发挥作用：变量、条件语句，最重要的是，命令行参数。现在，来看看如何使用它们。

2.1.3 注释参数

CMake 中的许多命令，需要以空格分隔的参数来配置行为。如图 2.1 所示，参数周围的引号使用方式相当奇特。虽然某些参数需要引号，但其他参数则不需要。背后的原因是什么？

在底层，CMake 唯一识别的数据类型是字符串，这就是为什么每个命令都期望其参数为零个或多个字符串。CMake 将评估每个参数为一个静态字符串，然后将它们传递给命令。评估意味着字符串插值，或者用另一个值替换字符串的一部分。这可能包括替换转义序列、展开变量引用（也称为变量插值），以及解包列表。

根据上下文，需要按需启用这种评估。为此，CMake 提供了三种类型的参数：

- 括号参数
- 引号参数
- 非引号参数

CMake 中的每种参数类型都有其独特性，并提供不同级别的评估。

括号参数

括号参数不会计算，用于将多行字符串原封不动地作为单个参数传递给命令。所以这样的参数，将包括以制表符和换行符形式的空白。

括号参数的格式与注释相同，以 [= [开始，以]]=] 结束，并且开头和结尾标记中的等号数量必须匹配（只要匹配，省略等号也是可以的）。与注释的唯一区别是括号参数不能嵌套。

以下是与 message() 命令一起使用此类参数的示例，该命令将所有传递的参数输出到屏幕上：

ch02/01-arguments/bracket.cmake

```
1 message([[multiline
2     bracket
3     argument
4 ]])
5 message([==[
6     because we used two equal-signs "==""
7     this command receives only a single argument
8     even if it includes two square brackets in a row
9     { "petsArray" = [[["mouse","cat"],["dog"]]] }
10 ]]==])
```

前面的示例中，可以看到不同形式的括号参数。第一个调用中在单独一行放置结束标签，如何在输出中引入一个空行：

```
$ cmake -P ch02/01-arguments/bracket.cmake
multiline
bracket
argument
because we used two equal-signs "==""
following is still a single argument:
{ "petsArray" = [[["mouse","cat"],["dog"]]] }
```

当传递包含双括号 (]) 的文本时，第二种形式非常有用，不会解释为标记参数的结束。

这类括号参数的使用有限——包含较长的文本块，这些文本块包含显示给用户的消息。大多数情况下，需要更动态的内容，例如引号参数。

引号参数

引号参数类似于常规的 C++ 字符串——这些参数将多个字符（包括空白）组合在一起，并将展开转义序列。与 C++ 字符串一样，以双引号字符”开头和结尾，因此要在输出字符串中包含引号字符，必须使用反斜杠进行转义，\"。其他熟悉的转义序列也得到支持：\\ 表示字面上的反斜杠，\t 是制表符，\n 是换行符，而 \r 是回车符。

这就是与 C++ 字符串相似之处结束的地方。引号参数可以跨越多行，并且它们会展开变量引用。可以将它们视为内置的 C 语言中的 `sprintf` 函数或 C++20 中的 `std::format` 函数。要将变量引用插入到您的参数中，请将变量名包裹在如下标记中：\${name}。我们将在本章的“使用变量”部分中更详细地讨论变量引用。

能猜到以下脚本的输出将有多少行吗？

ch02/01-arguments/quoted.cmake

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

来看看实际效果：

```
$ cmake -P ch02/01-arguments/quoted.cmake
1. escape sequence: "
in a quoted argument
2. multi...
line
3. and a variable reference: 3.26.0
```

没错——我们有一个转义的引号字符，一个换行转义序列，以及一个字面上的换行符。我们还访问了一个内置的 CMAKE_VERSION 变量，可以看到它在最后一行。

现在，来看看 CMake 如何处理没有引号的参数。

非引号参数

编程世界中，我们已经习惯了字符串以某种形式进行定界，例如使用单引号、双引号或反斜杠。CMake 偏离了这一惯例，引入了非引号参数。我们可能会争论，省略定界符可以使代码更容易阅读。这是真的吗？

非引号参数会评估转义序列和变量引用，但请注意分号 (;)，因为在 CMake 中，分号视为列表定界符。如果参数包含分号，CMake 会将它分割成多个参数。如果需要使用它们，请用反斜杠转义每个分号，\;。

可能会发现这些参数难以处理，所以这里使用插图来帮助理解这些参数的分割情况：

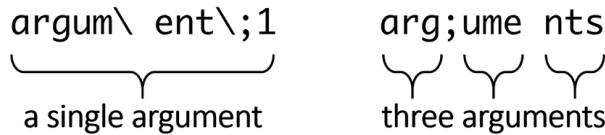


图 2.2：转义序列导致单独的标记解释为单个参数

注意非引号参数。一些 CMake 命令需要特定数量的参数，并忽略超出部分。如果参数意外地分离了，将得到难以调试的错误。

非引号参数不能包含未转义的引号 (‘)、井号 (#) 和反斜杠 (\)。如果觉得这还记不住，那么括号 (()) 只有在形成正确匹配成对时才允许。也就是说，将从一个开括号开始，并在关闭命令参数列表之前关闭它。

以下是一些示例，演示我们讨论过的规则：

ch02/01-arguments/unquoted.cmake

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

输出将是什么？一起来看看：

```
$ cmake -P ch02/01-arguments/unquoted.cmake
a single argument
twoarguments
threeseparatedarguments
3.16.3
()()()
```

即使是像 `message()` 这样的简单命令，对于分离的非引号参数也非常挑剔。在明确转义的情况下，单个参数中的空格会正确打印。然而，`twoarguments` 和 `threeseparatedarguments` 粘在一起，因为 `message()` 不会添加空格。

考虑到所有这些复杂性，什么时候使用非引号参数？一些 CMake 命令允许可选参数，这些参数前面有一个关键字，表示将提供可选参数。在这种情况下，使用非引号参数作为关键字可以使代码更加易读。例如：

```
1 project(myProject VERSION 1.2.3)
```

此命令中，`VERSION` 关键字及其后面的参数 `1.2.3` 是可选的。为了提高可读性，两者都没有使用引号。注意，关键字区分大小写。

现在了解了如何处理 CMake 参数的复杂性和特性，我们准备好探讨下一个有趣的话题——在 CMake 中使用各种变量。

2.3. 使用变量

CMake 中的变量是一个复杂的主题。不仅有三种变量类别——普通变量、缓存变量和环境变量，而且它们还存在于不同的变量作用域中，每个作用域如何影响其他作用域都有特定的规则。通常情况下，对这些规则的理解不足，会成为错误和头疼的根源。我建议你仔细研究这一部分，并确保在继续之前理解所有概念。

让我们从 CMake 中变量的几个关键事实开始：

- 变量名是区分大小写的，并且几乎可以包含任何字符。
- 所有变量在内部都存储为字符串，某些命令可以将它们解释为其他数据类型的值（甚至是列表！）。

基本的变量操作命令是 `set()` 和 `unset()`，但还有其他可以改变变量值的命令，如 `string()` 和 `list()`。

要声明一个普通变量，只需调用 `set()`，提供变量名和值：

ch02/02-variables/set.cmake

```
1 set(MyString1 "Text1")
2 set([[My String2]] "Text2")
3 set("My String 3" "Text3")
4 message(${MyString1})
5 message(${My\ String2})
6 message(${My\ String\ 3})
```

使用方括号和引号参数允许在变量名中包含空格。在稍后引用时，必须用反斜杠 (\) 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 (_)。

还要避免使用以下开头的保留名称（大写、小写或混合大小写）：CMAKE_、_CMAKE_，或下划线 (_)，后跟 CMake 命令的名称。

要取消设置变量，可以使用以下方式：unset(MyString1)。

Note

set() 命令接受普通文本变量名作为其第一个参数，但 message() 命令使用包裹在 \${} 语法中的变量引用。

如果用 \${} 语法包裹的变量提供给 set() 命令，会发生什么？

要回答这个问题，需要更好地理解变量引用。

2.3.1 变量引用

我之前在“命令参数”部分简要提到了引用，因为它们用于引用引号和非引号参数。要创建对已定义变量的引用，需要使用 \${} 语法，如下所示：message(\${MyString1})

在计算时，CMake 将按照从内到外的顺序遍历变量作用域，并用一个值替换 \${MyString1}，如果没有找到变量，则用空字符串替换（CMake 不会产生任何错误消息）。这个过程也称为变量求值、扩展或插值。

插值是从内到外进行的，从最内层的花括号对开始，向外移动。例如，遇到 \${MyOuter}\${MyInner}} 引用：

1. CMake 会首先尝试评估 MyInner，而不是搜索名为 MyOuter\${MyInner} 的变量。
2. 如果 MyInner 变量成功扩展，CMake 将使用新形成的引用重复扩展过程，直到无法进一步扩展为止。

为避免意外结果，建议不要在变量值中存储变量扩展。

CMake 将执行变量扩展到最大限度，并且在完成后，把得到的值作为参数传递给命令。这就是为什么当我们调用 set(\${MyInner} "Hi"); 时，实际上不会更改 MyInner 变量，而是会更改以 MyInner 存储的值为名称的变量，这不是想要的结果。

变量引用在变量类别方面的工作方式有些特殊：

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

没错，使用 \${}！你可能会从一个类别或另一个类别获取值：如果在当前作用域中设置了普通变量，则使用普通变量；如果没有设置，或者取消了设置，CMake 将使用同名的缓存变量。如果没有这样的变量，引用将计算为空字符串。

CMake 预定义了许多内置普通变量，用于不同的目的。例如，可以在--标记后传递命令行参数给脚本，它们将存储在 CMAKE_ARGV 变量中（CMAKE_ARGC 变量将包含计数）。

2.3.2 使用环境变量

这是最简单类型的变量。CMake 会复制启动 `cmake` 进程时环境中存在的变量，并在单个全局范围内使它们可用。要引用这些变量，请使用 `$ENV{}` 语法。

CMake 会更改这些变量，但更改仅会应用于正在运行的 `cmake` 进程中的本地副本，而不会实际更改系统环境；此外，这些更改对后续构建或测试的运行不可见，因此不推荐这样做。

注意，有几个环境变量会影响 CMake 行为的不同方面。例如，`CXX` 变量指定将用于编译 C++ 文件的执行文件。我们将介绍环境变量，它们与本书相关。完整列表可在文档中找到：<https://cmake.org/cmake/help/latest/manual/cmake-envvariables.7.html>。

Note

重要的是要意识到，如果将 `ENV` 变量用作命令的参数，则在生成构建系统期间将进行变量插值。所以将永久性地嵌入到构建树中，因此在构建阶段更改环境将不会有任何效果。

例如，以下是一个项目文件：

ch02/03-environment/CMakeLists.txt

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

前面的示例有两个步骤：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，该阶段将作为构建过程的一部分回显相同的变量。我们可以使用一个 `bash` 脚本，来测试在配置阶段，以及使用另一个值构建阶段会发生什么：

ch02/03-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 .
```

运行前面的代码清楚地显示，在配置期间设置的值会保留到生成的构建系统中：

```
$ ./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
```

现在，让我们继续讨论最后一类变量：缓存变量。

2.3.3 使用缓存变量

我们在第一章讨论 `cmake` 的命令行选项时首次提到了缓存变量，它们是存储在构建树的 `CMakeCache.txt` 文件中的持久变量。它们包含项目配置阶段收集的信息。这些信息来源于系统（编译器、链接器、工具等的路径）和用户，通过 GUI 或 `-D` 选项从命令行获取。再次强调，缓存变量在脚本中不可用；它们只存在于项目中。

如果 `${<name>}` 引用找不到当前作用域中定义的正常变量，但存在同名的缓存变量，则使用缓存变量，也可以通过 `$CACHE{<name>}` 语法显式引用，并使用 `set()` 命令的特殊形式定义：

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

与普通变量的 `set()` 命令不同，缓存变量需要额外的参数：`<type>` 和 `<docstring>`。这是因为这些变量可以由用户配置，并且 GUI 需要这些信息来适当地显示它们。

以下类型可以接受：

- `BOOL`: 一个布尔开关值。GUI 将显示一个复选框。
- `FILEPATH`: 磁盘上一个文件的路径。GUI 将打开一个文件对话框。
- `PATH`: 磁盘上一个目录的路径。GUI 将打开一个目录对话框。
- `STRING`: 一行文本。GUI 提供了一个要填充的文本字段，可以通过调用 `set_property(CACHE <variable> STRING <values>)` 来替换为下拉控件。
- `INTERNAL`: 一行文本。GUI 将跳过内部条目。内部条目可以用来在运行之间持久存储变量，使用此类型隐式添加 `FORCE` 关键字。

`<docstring>` 值只是一个标签，GUI 将在字段旁边显示它，以便向用户提供此设置的更多细节。即使对于内部类型也是必需的。

在代码中设置缓存变量在某种程度上，遵循与环境变量相同的规则——值只在当前 CMake 执行中覆盖。然而，如果缓存文件中不存在该变量或指定了可选的 `FORCE` 参数，该值将持久化：

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

与 C++ 类似，CMake 支持变量作用域。

2.3.4 如何在 CMake 中正确使用变量作用域

变量作用域可能是 CMake 语言中最奇特的概念。这可能是因为习惯了在通用编程语言中的实现方式，之所以提前解释这一点，是因为对作用域的错误理解，常常是难以发现和修复的 bug 的来源。

需要明确的是，作为一般概念的变量作用域，旨在用代码分隔不同的抽象层。作用域以树状方式相互嵌套。最外层的作用域（根作用域）称为全局作用域。作用域都可以称为局部作用域，以指示当前执行或讨论的作用域。作用域在变量之间创建边界，使得嵌套作用域可以访问在外层作用域中定义的变量，反之则不行。

CMake 有两种变量作用域：

- 文件：在文件内执行块和自定义函数时使用
- 目录：在调用 `add_subdirectory()` 命令以在嵌套目录中执行另一个 `CMakeLists.txt` 列表文件时使用

Note

条件块、循环块和宏不会创建独立的作用域。

那么，CMake 中变量作用域的实现方式有何不同？当创建嵌套作用域时，CMake 简单地用外层作用域中所有变量的副本填充它。随后的命令将影响这些副本。但是嵌套作用域的执行完成，所有副本都会删除，外层作用域的原始变量将恢复。

作用域在 CMake 中的工作方式，有一些在其他语言中不常见的有趣含义。在嵌套作用域中执行时，如果取消设置 (`unset()`) 在外层作用域中创建的变量，它将消失，但在当前的嵌套作用域中，因为变量是局部副本。如果现在引用这个变量，CMake 会确定没有定义这样的变量，会忽略外层作用域，并继续搜索缓存变量（认为是独立的）。

文件变量作用域通过 `block()` 和 `function()` 命令（但不包括 `macro()`）打开，并分别通过 `endblock()` 和 `endfunction()` 命令关闭。我们将在本章的命令定义部分介绍函数。现在，通过更简单的 `block()` 命令（在 CMake 3.25 中引入）来看看变量作用域在实践中是如何工作的。

考虑以下示例：

ch02/04-scope/scope.cmake

```
1 cmake_minimum_required(VERSION 3.26)
2
3 set(V 1)
4 message("> Global: ${V}")
5 block() # outer block
6   message(" > Outer: ${V}")
7   set(V 2)
8   block() # inner block
9     message(" > Inner: ${V}")
10    set(V 3)
11    message(" < Inner: ${V}")
12  endblock()
13  message(" < Outer: ${V}")
14 endblock()
15 message("< Global: ${V}")
```

最初在全局作用域中将变量 `V` 设置为 1。进入外层和内层块后，立即将它们分别更改为 2 和 3。还在进入和退出每个作用域时打印变量：

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

如前所述，当进入每个嵌套作用域时，变量值从外层作用域暂时复制，但在退出时恢复其原始值。这在输出的最后两行中反映出来。

`block()` 命令还可以将值传播到外层作用域（就像 C++ 默认会做的那样），但这需要通过 `PROPAGATE` 关键字显式启用。如果我们为内层块启用传播 (`block(PROPAGATE V)`)：

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

再次强调，我们影响了外部代码块的范围内，但没有影响到全局范围。

另一种修改外部作用域中的变量的方法是，为 `set()` 和 `unset()` 命令设置 `PARENT_SCOPE` 标志：

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

这种变通方法有一定的局限性，它不允许访问超过一级以上的变量。另一点值得注意的是，使用 `PARENT_SCOPE` 并不会改变当前作用域中的变量。

现在知道了如何处理基本变量，让我们来看一个特殊情况：由于所有变量都作为字符串存储，CMake 必须采取更创造性地方法来处理更复杂的数据结构，比如列表。

2.4. 使用列表

为了存储一个列表，CMake 会将所有元素连接成一个字符串，使用分号 (;) 作为分隔符：`a;list;of;5;elements`。可以在元素中使用反斜杠来转义分号：`a\;single\;element`。

要创建一个列表，我们可以使用 `set()` 命令：

```
1 set(myList a list of five elements)
```

由于列表的存储方式，以下命令将产生完全相同的效果：

```
1 set(myList "a;list;of;five;elements")
2 set(myList a list "of;five;elements")
```

CMake 会自动在未加引号的参数中解包列表。通过传递未加引号的 `myList` 引用，我们实际上向命令发送了更多的参数：

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

`message()` 命令将接收六个参数：“`the list is:`”，“`a`”，“`list`”，“`of`”，“`five`”和“`elements`”。这可能会产生意想不到的后果，因为输出将不带额外的空格打印参数：

```
the list is:a list of five elements
```

这是一个非常简单的机制，应该小心使用。

CMake 提供了一个 `list()` 命令，该命令提供了多种子命令来读取、搜索、修改和排序列表。以下是一个简短的摘要：

```
1 list(LENGTH <list> <out-var>)
2 list(GET <list> <element index> [<index> ...] <out-var>)
3 list(JOIN <list> <glue> <out-var>)
4 list(SUBLIST <list> <begin> <length> <out-var>)
5 list(FIND <list> <value> <out-var>)
6 list(APPEND <list> [<element>...])
7 list(FILTER <list> {INCLUDE | EXCLUDE} REGEX <regex>)
8 list(INSERT <list> <index> [<element>...])
9 list(POP_BACK <list> [<out-var>...])
10 list(POP_FRONT <list> [<out-var>...])
11 list(PREPEND <list> [<element>...])
12 list(REMOVE_ITEM <list> <value>...)
13 list(REMOVE_AT <list> <index>...)
14 list(REMOVE_DUPLICATES <list>)
15 list(TRANSFORM <list> <ACTION> [...])
16 list(REVERSE <list>)
17 list(SORT <list> [...])
```

大多数时候，我们的项目中并不真正需要使用列表。然而，如果发现自己处于那种罕见的情况，这个概念会很方便，可以在附录中找到 `list()` 命令的更深入的参考资料。

现在，知道了如何处理各种类型的列表和变量，让我们将焦点转移到控制执行流程上，并了解 CMake 中可用的控制结构。

2.5. 控制结构

CMake 语言要是没有控制结构就不完整了！其以命令的形式提供，并且分为三类：条件块、循环和命令定义。控制结构在脚本执行和项目构建系统生成期间执行。

2.5.1 条件块

CMake 支持的条件块是 `if()` 命令。所有条件块都必须用 `endif()` 命令关闭，并且可以按照以下顺序有任意数量的 `elseif()` 命令和一个可选的 `else()` 命令：

```
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()` 块控制执行的命令集：

- 如果 `if()` 中指定的 `<condition>` 表达式成立，将执行第一个部分。
- 否则，CMake 将执行属于此块中，第一个满足其条件的 `elseif()` 中的命令。
- 如果没有这样的命令，CMake 将检查是否提供了 `else()` 命令，并执行代码该部分中的命令。
- 如果前面的条件都不满足，执行将在 `endif()` 命令之后继续。

注意，条件块中不存在局部作用域。

提供的 `<condition>` 表达式，会根据非常简单的语法进行计算——让我们了解更多关于它的信息。

语法

`if()`、`elseif()` 和 `while()` 命令的语法基本相同。

逻辑运算符

`if()` 条件支持 NOT、AND 和 OR 逻辑运算符：

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

此外，嵌套条件可以，需要匹配的圆括号对 (())。与所有体面的语言一样，CMake 语言遵循求值顺序，并从最内层的括号开始：

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

字符串和变量的求值

由于历史原因（变量引用 `(${})` 语法并不总是存在），CMake 尝试计算未引用的参数，就好像是变量引用一样。换句话说，在条件中使用一个普通的变量名（例如，`QUX`）等于 `${QUX}`。这里有一个例子，也是一个陷阱：

```
1 set(BAZ FALSE)
2 set(QUX "BAZ")
```

```
3 if(${QUX})
```

这里的 `if()` 条件工作方式有些复杂——首先，将 `${QUX}` 求值为 `BAZ`，这是一个已识别的变量，进而解析为一个包含五个字符的字符串，拼写为 `FALSE`。字符串只有在等于 `ON`、`Y`、`YES`、`TRUE` 或非零数字，才可视为真（这些比较不区分大小写）。

所以，前面示例中的条件为假。

这里还有一个陷阱——如果一个条件的未加引号参数名，一个包含 `BAR` 这样的值的变量，那结果会是什么？看看以下代码示例：

```
1 set(FOO BAR)
2 if(FOO)
```

根据目前所知，条件将解析为假，因为 `BAR` 字符串不符合求值为真的条件。但情况并非如此，因为 CMake 在处理未加引号的变量引用时有一个例外。与加引号的参数不同，`FOO` 不会求值为 `BAR`，以生成 `if("BAR")` 语句（这将是不正确的）。相反，CMake 只有在以下情况下，才会将 `if(FOO)` 计算为假：

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

简单地判断一个未定义的变量，将为假：

```
1 if (CORGE)
```

当变量事先定义后，情况就改变了，条件计算为真：

```
1 set(CORGE "A VALUE")
2 if (CORGE)
```

Note

如果认为未加引号的 `if()` 参数的递归求值令人困惑，可以将变量引用用引号括起来：`if("${CORGE}")`。这将导致在提供的参数传递到 `if()` 命令之前对其进行求值，其行为与字符串一致。

换句话说，CMake 假定传递变量名给 `if()` 命令的用户，是在询问该变量是否定义了不等于“假”的值。为了明确检查变量是否已定义（忽略其值），可以使用以下方法：

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

比较值

支持以下操作符进行比较操作：

EQUAL, LESS, LESS_EQUAL, GREATER 和 GREATER_EQUAL

其他语言中常见的比较操作符，在 CMake 中不起作用。

它们可以用来比较数值：

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

可以通过在操作符前加上 VERSION_ 前缀来按照 [major[.minor[.patch[.tweak]]]] 的格式比较软件版本：

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

省略的组件视为零，非整数的版本组件将在此点截断比较字符串。

对于字典序字符串比较，需要在操作符前加上 STR 前缀（没有下划线）：

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

我们经常需要比简单相等更高级的比较机制，CMake 支持 POSIX 正则表达式匹配 (CMake 文档暗示支持扩展正则表达式 (ERE) 风格，但未提到对特定正则字符类的支持)。可以按以下方式使用 MATCHES 操作符：

```
<VARIABLE|STRING> MATCHES <regex>
```

匹配组会被捕获到 CMAKE_MATCH_<n> 变量中。

简单检查

之前提到了一个简单检查，DEFINED。也还有其他的检查，如果满足条件则返回真。

还可以检查以下内容：

- 值是否在列表中：<VARIABLE|STRING> IN_LIST <VARIABLE>
- 此版本的 CMake 中是否可以调用某个命令：COMMAND <command-name>
- 是否存在一个 CMake 策略：POLICY <policy-id>
- 是否使用 add_test() 添加了 CTest 测试：TEST <test-name>
- 是否定义了一个构建目标：TARGET <target-name>

我们将在第 5 章，使用目标中探讨构建目标。现在，让我们简单的认为，目标是通过 add_executable(), add_library() 或 add_custom_target() 命令创建的基本项目单元 (构建过程的)。

检查文件系统

CMake 提供了许多处理文件的方法。我们很少需要直接操作，通常会使用高级方法。作为参考，本书将在附录中提供与文件相关的命令的简短列表。以下操作符会经常使用（仅对绝对路径有明确的行为定义）：

- EXISTS <path-to-file-or-directory>: 检查文件或目录是否存在。也会符号链接进行解析（如果符号链接的目标存在，则返回真）。
- <file1> IS_NEWER_THAN <file2>: 检查哪个文件修改时间更晚。如果文件 1 比文件 2 修改时间更晚（或等于）或者两个文件中有一个不存在，则返回真。
- 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。

3.24 版本开始，CMake 支持简单的路径比较检查，这将折叠多个路径分隔符，但不会进行其他标准化：

```
1 if ("/a///b/c" PATH_EQUAL "/a/b/c") # returns true
```

对于更高级的路径操作，请参阅 `cmake_path()` 命令的文档。

完成了条件命令的语法介绍，接下来我们将讨论的控制结构是——循环。

2.5.2 循环

CMake 中，循环很简单——可以使用 `while()` 循环或 `foreach()` 循环，来重复执行同一组命令。这两个命令都支持循环控制机制：

- `break()` 循环将停止执行剩余的块，并跳出包围的循环。
- `continue()` 循环将停止当前迭代的执行，并从下一个迭代的开头开始。

注意，循环块中不会创建局部作用域。

`while()`

循环块使用 `while()` 命令开启，并使用 `endwhile()` 命令关闭。只要 `while()` 中提供的 `<condition>` 表达式为真，封闭的命令都将会执行。条件部分的语法与 `if()` 命令相同：

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

通过一些变量——`while` 循环可以替代 `for` 循环。实际上，使用 `foreach()` 循环会更容易。

`foreach()`

`foreach()` 块有几种变体，为给定列表中的每个值执行封闭的命令。像其他块一样，有打开和关闭命令：`foreach()` 和 `endforeach()`。

`foreach()` 的最简单形式类似 C++ 的 `for` 循环：

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

CMake 将从 0 迭代到 `<max>` (包括)。如果需要更多控制，可以使用第二个变体，提供 `<min>`, `<max>`, 以及可选的 `<step>`。所有参数都必须是非负整数，且 `<min>` 必须小于 `<max>`:

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

当 `foreach()` 处理列表时，才是彰显特色的时候：

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

CMake 将从一个或多个指定的 `<lists>` 列表变量中检索元素，以及一行定义的 `<items>` 值列表，将其放入 `<loop variable>` 中。然后，针对列表中的每个项执行所有命令，可以选择只提供列表或值（或者两者都提供）：

ch02/06-loops/foreach.cmake

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

将输出以下内容：

```
1
2
3
e
f
```

或者，可以使用简短版本（跳过 `IN` 关键字）得到相同的结果：

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

自从 3.17 版本以来，`foreach()` 支持压缩列表 (`ZIP_LISTS`)：

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

压缩列表的过程，涉及迭代多个列表，并对具有相同索引的对应项进行操作：

ch02/06-loops/foreach.cmake

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

CMake 将为提供的每个列表创建一个 `num_<N>` 变量，将用每个列表中的项填充这些变量。可以传递多个变量名（每个列表一个），每个列表将使用单独的变量来存储其项：

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

`ZIP_LISTS` 的两个例子都将产生相同的输出：

```
word=one, num=1
word=two, num=2
word=three, num=3
word=four, num=4
```

如果列表之间的项数不同，较短的列表的变量将为空。

自从 3.21 版本以来，`foreach()` 中的循环变量会限制在循环的局部作用域内使用。

好了，到此为止，关于循环的讨论也就差不多了结束了。

2.5.3 自定义命令

有两种方法可以自定义命令：可以使用 `macro()` 命令或 `function()` 命令。这两个命令相当于 C 语言中的预处理器宏和函数：

`macro()` 命令更像是查找替换指令，而不是 `function()` 那样的函数调用，所以宏不会在调用堆栈上保留信息。所以，宏中调用 `return()` 将返回到比使用点更高一层的调用语句（如果已经在顶层作用域中，可能会终止执行）。

`function()` 命令为变量创建局部作用域，这与 `macro()` 命令不同。我们会在下一节中，讨论这些细节。

这两种定义方法都允许，在定义命令的局部作用域中，引用的命名参数。此外，CMake 提供了以下变量来访问与调用相关的参数值：

- `${ARGC}`: 参数计数
- `${ARGV}`: 参数列表（所有参数）
- `${ARGV<index>}`: 特定索引（从 0 开始）处的参数值
- `${ARGN}`: 调用者在最后一个参数后传递的匿名参数列表

使用超出 `ARGC` 范围的索引访问数字参数，会导致未定义行为。要处理高级场景（通常参数数量未知），可以去阅读官方文档中，关于 `cmake_parse_arguments()` 的内容。如果决定使用带有命名参数的命令，每次调用都必须传递这些参数，否则调用将无效。

宏

宏与其他块定义类似：

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

此声明之后，可以通过调用其名称来执行宏（函数调用不区分大小写）。

宏不会在修改调用堆栈信息，或为变量添加新的作用域。以下示例展示了宏的这一行为：

ch02/08-definitions/macro.cmake

```
1 macro(MyMacro myVar)
2     set(myVar "new value")
3     message("argument: ${myVar}")
4 endmacro()
5 set(myVar "first value")
6 message("myVar is now: ${myVar}")
7 MyMacro("called value")
8 message("myVar is now: ${myVar}")
```

以下是脚本的输出

```
$ cmake -P ch02/08-definitions/macro.cmake
myVar is now: first value
argument: called value
myVar is now: new value
```

发生了什么？尽管将 `myVar` 设置为“新值”，但它并没有影响到 `message("argument: ${myVar}")` 的输出！因为传递给宏的参数不是作为真正的变量处理，而是作为常量的查找替换指令。

另一方面，全局作用域中的 `myVar` 变量从“初始值”修改为“新值”。这种行为有一定的副作用，被认为是一种不良实践。因为如果不阅读宏，就无法知道宏会修改哪些全局变量。建议尽可能使用函数，可能避免这种问题。

函数

要将命令声明为函数，请遵循以下语法：

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

函数需要一个名称，可以选择接受一系列参数名称。如前所述，函数会创建变量作用域。可以使用 `set()`，提供函数的一个命名参数，并且任何更改都将局限于函数内部（除非指定了 `PARENT_SCOPE`）。

函数遵循调用堆栈的规则，允许使用 `return()` 命令返回到调用作用域。从 CMake 3.25 开始，`return()` 命令允许一个可选的 `PROPAGATE` 关键字，后跟一系列变量名称。其目的与 `block()` 命令类似——指定变量的值从局部作用域传递到调用作用域。

CMake 为每个函数设置了以下变量（这些变量自 3.17 版本起可用）：

- `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际中的使用：

ch02/08-definitions/function.cmake

```

1 function(MyFunction FirstArg)
2     message("Function: ${CMAKE_CURRENT_FUNCTION}")
3     message("File: ${CMAKE_CURRENT_FUNCTION_LIST_FILE}")
4     message("FirstArg: ${FirstArg}")
5     set(FirstArg "new value")
6     message("FirstArg again: ${FirstArg}")
7     message("ARGV0: ${ARGV0} ARGV1: ${ARGV1} ARGC: ${ARGC}")
8 endfunction()
9 set(FirstArg "first value")
10 MyFunction("Value1" "Value2")
11 message("FirstArg in global scope: ${FirstArg}")

```

使用 `cmake -P function.cmake` 运行这个脚本将输出以下输出：

```

Function: MyFunction
File: /root/examples/ch02/08-definitions/function.cmake
FirstArg: Value1
FirstArg again: new value
ARGV0: Value1 ARGV1: Value2 ARGC: 2
FirstArg in global scope: first value

```

函数的总体语法和概念与宏非常相似，但不易受到隐式错误的影响。

CMake 中的过程范式

编写 CMake 代码，需要创建一个 `CMakeLists.txt` 列表文件，该文件将调用三个定义的命令，这些命令可能还会调用其自定义命令。图 2.3 对这种可能性，进行了展示：

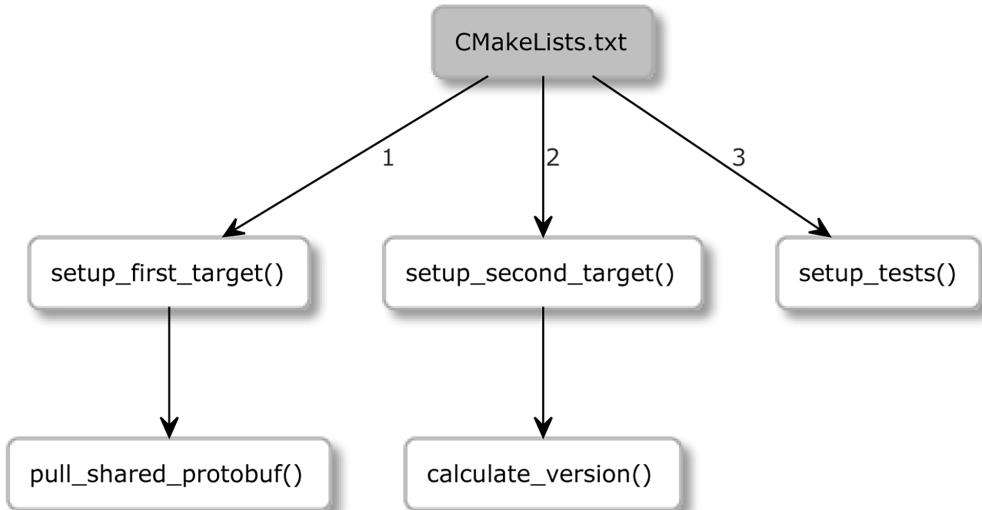


图 2.3：过程调用图

在 CMake 中，以过程风格编写代码可能会出现问题，必须在调用之前提供定义。CMake 解析器不接受其他方式，所以代码可能会写成这样：

```
1 cmake_minimum_required(....)
2 project(Procedural)
3
4 # Definitions
5 function(pull_shared_protobuf)
6 function(setup_first_target)
7 function(calculate_version)
8 function(setup_second_target)
9 function(setup_tests)
10
11 # Calls
12 setup_first_target()
13 setup_second_target()
14 setup_tests()
```

真是噩梦！一切都颠倒了！由于最低抽象级别的代码位于文件开头，这将使整段代码难以理解。结构正确的代码，应该在第一个子程序中列出最一般的步骤，然后提供详细一点的子程序，并将最详细的步骤放在文件的最后。

这个问题有解决方案，比如将命令定义移动到其他文件中，并在目录之间划分作用域。但还有一个简单而优雅的方法——在文件顶部声明一个入口点宏，并在文件末尾调用：

```
1 macro(main)
2     first_step()
3     second_step()
4     third_step()
5 endmacro()
6
7 function(first_step)
8 function(second_step)
9 function(third_step)
10
11 main()
```

这种方法会逐渐缩小作用域，并且在文件末尾才实际调用 `main()` 宏，所以 CMake 不会因为执行未定义的命令而抱怨。

那为什么使用宏而不是函数？能够不受限制地访问全局变量是很好的，而且由于无需向 `main()` 传递任何参数，所以不需要有任何的担心。

可以在本书的 GitHub 库的 `ch02/09-procedural/CMakeLists.txt` 列表文件中，找到这个简单的示例。

关于命名约定

软件开发中，命名是一项难题，并且保持一个易于阅读和理解的解决方案仍然非常重要。对于 CMake 脚本和项目，我们应该遵循干净代码的规则：

- 遵循一致的命名风格（在 CMake 社区中，snake_case 是可接受的标准）。
- 使用简短但有意义的名称（避免使用 func()，f() 等）。
- 避免在命名中使用双关语和只有编写者知道的东西。
- 使用可发音、可搜索的名称，不需要进行心智映射。

这样，我们结束了“如何使用正确的语法调用命令”。

接下来，来探讨一下哪些命令比较适合于入门。

2.6. 常用命令

CMake 提供了许多脚本命令，允许处理变量和环境。其中一些在附录中已经进行了详细介绍，例如：list()、string() 和 file()。其他，如：find_file()、find_package() 和 find_path()，更适合放在属于它们主题的章节中介绍。

本节中，我们将简述一些常用命令：

- message()
- include()
- include_guard()
- file()
- execute_process()

2.6.1 message()

message() 命令，将文本输出到标准输出，但其功能远不止于此。其有一个 MODE 参数，可以定义命令的行为，如下所示：message(<MODE> "text to print")。

可用的 MODE 如下所示：

- FATAL_ERROR：停止处理和生成。
- SEND_ERROR：继续处理，但跳过生成。
- WARNING：继续处理。
- AUTHOR_WARNING：输出警告，但继续处理。
- DEPRECATION：如果启用了 CMAKE_ERROR_DEPRECATED 或 CMAKE_WARN_DEPRECATED，则输出相应地信息。
- NOTICE 或省略模式（默认）：输出消息到 stderr，以吸引使用者的注意。
- STATUS：继续处理，推荐用于向用户显示的主要消息。
- VERBOSE：继续处理，应用于更详细的信息，通常不是非常必要。
- DEBUG：继续处理，应包含项目出现问题时，对处理问题有帮助的详细信息。
- TRACE：继续处理，建议在项目开发期间输出消息。通常，这类消息会在发布项目之前移除。

选择正确的模式需要花点心思，可以通过基于严重性（自 3.21 版本起）给输出文本着色来节省调试时间，甚至在声明不可恢复的错误后停止执行：

ch02/10-useful/message_error.cmake

```
1 message(FATAL_ERROR "Stop processing")
2 message("This won't be printed.")
```

消息将根据当前的日志级别（默认为 STATUS）打印，在上一章的调试和跟踪选项部分讨论了如何更改此设置。

在第 1 章中，我提到了使用 CMAKE_MESSAGE_CONTEXT 进行调试，现在是深入研究它的时候了。在此期间，我们来了解了这个主题三个关键部分：列表、作用域和函数。

在复杂的调试场景中，指出消息发生的上下文会非常有用。考虑以下输出，其中在 foo 函数中打印的消息有适当的前缀：

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

具体代码：

ch02/10-useful/message_context.cmake

```
1 function(foo)
2     list(APPEND CMAKE_MESSAGE_CONTEXT "foo")
3     message("foo message")
4 endfunction()
5
6 list(APPEND CMAKE_MESSAGE_CONTEXT "top")
7 message("Before `foo`")
8 foo()
9 message("After `foo`")
```

来分解一下：

- 首先，将 top 追加到上下文跟踪变量 CMAKE_MESSAGE_CONTEXT，然后打印最初的“Before ‘foo’”，匹配的前缀 [top] 将添加到输出中。
- 接下来，进入 foo() 函数时，我们在属于该函数的列表后，追加一个名为 foo 的新上下文，并输出另一个消息，该消息在输出中显示扩展的 [top.foo] 前缀。
- 最后，在函数执行完成后，我们打印“After ‘foo’”。消息以原始的 [foo] 作用域打印。为什么？因为变量作用域规则：更改的 CMAKE_MESSAGE_CONTEXT 变量只在函数作用域结束前有效，然后恢复为原始未更改的版本

message() 的另一个技巧是向 CMAKE_MESSAGE_INDENT 列表添加缩进（与 CMAKE_MESSAGE_CONTEXT 完全相同的方式）：

```
1 list(APPEND CMAKE_MESSAGE_INDENT " ")
2 message("Before `foo`")
```

```
3 foo()  
4 message("After `foo`")
```

然后，脚本输出看起来更简单：

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

由于 CMake 没有提供断点或其他调试工具，当事情没有按计划进行时，清晰的日志信息就显得尤为重要。

2.6.2 include()

将代码分割成不同的文件，以保持有序和分离。然后，可以通过 `include()` 从父列表文件引用它们：

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

如果提供一个文件名（带有`.cmake` 扩展名的路径），CMake 将尝试打开并执行。

注意，这也不会创建独立的变量作用域，因此该文件中对变量的修改都会对调用作用域产生影响。

如果文件不存在，CMake 将报错，除非使用 `OPTIONAL` 关键字指定相应文件为“可选的”。当需要知道 `include()` 是否成功时，可以提供 `RESULT_VARIABLE` 关键字以及变量的名称。在成功时，它的内容为文件的完整路径；在失败时，则为 `NOTFOUND`。

使用脚本模式下运行时，相对路径都将以当前工作目录解析相对路径。要强制相对于脚本本身进行搜索，请提供绝对路径：

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

如果不提供路径，但提供了模块的名称（不带`.cmake` 或其他），CMake 将尝试找到一个模块并包含它。CMake 将在 `CMAKE_MODULE_PATH` 中搜索名为 `<模块>.cmake` 的文件，然后是在 CMake 模块目录中搜索。

当 CMake 遍历源树，并包含了不同的列表文件时，将设置以下变量：

- `CMAKE_CURRENT_LIST_DIR`
- `CMAKE_CURRENT_LIST_FILE`
- `CMAKE_PARENT_LIST_FILE`
- `CMAKE_CURRENT_LIST_LINE`

2.6.3 include_guard()

对于有些文件，我们只希望包含一次，这时 `include_guard([DIRECTORY|GLOBAL])` 就可以使用了。

将 `include_guard()` 放在包含文件的顶部。当 CMake 第一次遇到它时，将在当前作用域中进行记录。如果文件再次包含，CMake 就不会再对该文件进行处理了。

避免在隔离相关的作用域，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子目录应用保护，而 `GLOBAL` 关键字将把保护应用于整个构建。

2.6.4 file()

为了了解可以在 CMake 脚本中执行的操作，来了解一下文件操作命令：

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

简而言之，`file()` 命令可以以系统无关的方式读取、写入和传输文件，以及与文件系统、文件锁、路径和存档等交互。有关更多详细信息，请参阅附录。

2.6.5 execute_process()

有时候，需要使用系统中的工具（毕竟，CMake 主要是一个构建系统生成器）。CMake 提供了一个命令以实现此目的：可以使用 `execute_process()` 来运行其他进程，并收集输出。这个命令非常适合脚本使用，也可以在项目中使用，但它仅在配置阶段有效。

以下是该命令的一般形式：

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

CMake 将使用操作系统的 API 来创建一个子进程（因此，像 `&&`，`||` 和 `>` 这样的 shell 操作符将不起作用），可以通过多次提供 `COMMAND` 参数来链接命令，并将一个命令的输出传递给另一个。

可选地 `TIMEOUT` 参数，用来在进程未在所需限制内完成任务时终止该进程，并且可以根据需要设置 `WORKING_DIRECTORY`。

所有任务的退出代码可以通过提供 `RESULTS_VARIABLE` 参数来收集到一个列表中。如果只对最后执行的命令的结果感兴趣，请使用单数形式：`RESULT_VARIABLE`。

为了收集输出，CMake 提供了两个参数：`OUTPUT_VARIABLE` 和 `ERROR_VARIABLE`（用法类似）。如果想合并 `stdout` 和 `stderr`，请为这两个参数使用相同的变量。

当为其他用户编写项目时，应该确保计划使用的命令，在声明支持的平台上可用。

2.7. 总结

本章打开了使用 CMake 编程的大门——现在能够编写出色、富有信息量的注释，并使用内置命令，了解了如何正确地提供各种类型的参数。仅凭这些知识，就能理解 CMake 列表文件的不寻常语法。我们讨论了 CMake 中的变量——特别是如何引用、设置和取消设置普通变量、缓存变量和环境变量。还深入探讨了文件和目录变量作用域的工作原理，如何创建它们，以及可能遇到的问题和问题的解决办法。我们还介绍了列表和控制结构，检查了条件的语法、逻辑操作、未引用参数的计算，以及字符串和变量。还学习了如何比较值、进行简单的检查，以及检查系统中文件的状态。这样，就能够编写条件块和 `while` 循环；在讨论循环时，我们还介绍了 `foreach` 循环的语法。

理解如何使用宏和函数语句，自定义命令无疑将促进编写更清晰、更有条理的代码。我们还讨论了改进代码结构和创建更具可读性名称的策略。

最后，介绍了 `message()` 命令及其多个日志级别。还研究了如何划分和包含列表文件，并且发现了一些有趣的命令。

全部的这些，为迎接下一章做好了充分的准备。

2.8. 扩展阅读

- 《干净的代码：敏捷软件工艺之道》（作者：Robert C. Martin）：

<https://amzn.to/3cm69DD>

- 《重构：改善既有代码的设计》（作者：Martin Fowler）：

<https://amzn.to/3cmWk8o>

- 哪些是好的代码注释？(Rafał Świdziński)：

<https://youtu.be/4t9bpo0THb8>

- CMake 中设置和使用变量的语法是什么？(StackOverflow)：

<https://stackoverflow.com/questions/31037882/whats-the-cmake-syntax-to-set-and-use-variables>

第 3 章 主流的 IDE 中使用 CMake

编程既是一门艺术，也是一种深奥的技术过程，二者都非常困难。因此，应尽可能优化这个过程。我们很少能通过简单地切换开关来获得更好的结果，但使用集成开发环境（IDE）无疑是一种好方法。

如果以前没有使用过合适的 IDE（或者认为像 Emacs 或 Vim 这样的文本处理器已经是最优选择），这一章节就是为您准备的。如果您是一位经验丰富的专业人士，并且已经对这一主题很熟悉，可以把这一章节作为当前最佳选择的快速概览，或许可以考虑转换，或者确认当前的工具就是最好的。

本章着重于为初入该领域的人提供易入门的介绍，对 IDE 的关键选择提供了一个温和的入门。我们将讨论为什么需要 IDE，以及如何选择最适合需求的 IDE。当然，市场上有很多选择，如果选择正确，有许多因素可能会影响您的生产力。我们将讨论一些在特定规模的组织中工作可能重要的考虑因素，确保能够把握这些细微之处，而不会陷入复杂性的漩涡。然后，我们将简要介绍工具链，讨论可用的选择。

然后，将介绍几种流行 IDE 的独特品质，如先进的 CLion，适应性强的 Visual Studio Code，以及功能强大的 Visual Studio IDE。每个部分都经过精心设计，以展示这些 IDE 的优势和高端功能。

本章中，将包含以下内容：

- 了解 IDE
- CLion IDE
- Visual Studio Code
- Visual Studio IDE

3.1. 了解 IDE

本节中，我们将讨论集成开发环境（IDE），以及它们如何显著提高开发速度和代码质量。让我们了解 IDE 是什么开始吧。

为什么使用 IDE，以及如何选择 IDE？IDE，或集成开发环境，是一种全面的工具，将各种专用工具结合在一起，简化了软件开发过程。创建专业项目的过程涉及许多步骤：设计、编码、构建、测试、打包、发布和维护。每个步骤都包含许多较小的任务，复杂性可能会让人不堪重负。IDE 通过提供一个平台来解决这一问题，该平台配备了一套由 IDE 创建者策划和配置的工具。这种集成能够无缝地使用这些工具，而无需为每个项目进行单独设置。

IDE 主要围绕代码编辑器、编译器和调试器展开。旨在提供足够的集成，能够编辑代码，立即编译，并附加调试器运行。IDE 可以包含构建工具链，或者允许开发者选择自己喜欢的编译器和调试器。编辑器通常是软件的核心部分，可以通过插件进行大量扩展，如代码高亮、格式化等。

更高级的 IDE 提供了非常复杂的功能，如热重载调试（在 Visual Studio 2022 中可用；继续阅读以了解更多信息）。此功能允许在调试器中运行代码，编辑它，并在不重新启动程序的情况下继续执行。还会发现重构工具，用于重命名符号或将代码提取到单独的函数中，以及静态

分析，用于在编译前识别错误。此外，IDE 提供了使用 Git 和其他版本控制系统的工具，这对于解决冲突来说是无价的。

可以看到，早期学习如何使用 IDE，并在组织中标准化这种使用是多么有益。来看看为什么选择合适的 IDE 很重要。

3.1.1 选择 IDE

社区公认为功能齐全的 IDE 有几个。致力于特定选择之前，建议先研究一下这个领域，特别是当前软件发布周期的速度和该领域的变化都非常快。

在我几年的企业经验中，很少有一个 IDE 提供的功能足够吸引人，使得有人从一种 IDE 切换到另一种 IDE。习惯的力量对开发者来说是第二天性，不应忽视。记住，当在 IDE 中感到舒适，它很可能会成为未来相当长一段时间的首选工具。这就是为什么仍然看到开发者使用 Vim（1991 年发布的基于控制台的文本编辑器），通过一堆插件使其像更现代的、基于 GUI 的 IDE 一样强大。

开发者选择一个 IDE，而不是另一个的原因有很多；其中一些非常重要（速度、可靠性、全面性、完整性），而其他的……则不那么重要。我想分享我对这个选择的个人观点，希望这会对阅读本书的人有用。

选择全面的 IDE

刚开始，可能会考虑使用简单的文本编辑器，并运行几个命令来构建代码。这种方法可行，尤其是试图理解基础知识时。这也会帮助理解初学者在没有 IDE 的情况下，可能会有的体验。

另一方面，IDE 是有目创造的，简化了开发人员在项目生命周期中处理的许多过程。尽管最初可能会让人感到不知所措，但请选择一个包含必要功能的全面 IDE。确保它尽可能完整，但也要注意成本，因为对于小型企业或个人开发者来说，IDE 可能会很昂贵。这是在手动管理上花费的时间与 IDE 提供的功能成本之间的平衡。

无论成本如何，始终选择具有强大社区支持的 IDE，以便在遇到问题时提供帮助。探索社区论坛和像 StackOverflow.com 这样的热门问答网站，检查用户是否得到了他们的答案。此外，选择一个由知名公司积极开发的 IDE。没人想浪费时间在长时间没有更新，可能会在不久的将来弃用或放弃的东西上。例如，GitHub 创建的编辑器 Atom 在发布了 7 年后淘汰。

选择组织中广泛支持的 IDE

反直觉的是，这可能与每位开发者的偏好不一致。您可能已经对大学、上一份工作或个人项目中的不同工具有所熟悉。如前所述，这种习惯可能会让您忽略公司的建议，坚持使用您所知道的工具。需要抵制这种诱惑，随着时间的推移，这样的选择会变得越来越具挑战性。

我分别在爱立信、亚马逊和思科工作过，只有一次尝试配置和维护非标准 IDE 的努力是值得的。因为我设法获得了足够的组织支持来集体解决问题，但阅读本书各位的主要目标应该是编写代码，而不是与不受支持的 IDE 作斗争。学习推荐的软件可能需要付出努力，但这比违背常规所需的努力要少（是的，Vim 在这场战斗中失败了；是时候继续前进了）。

不要根据目标操作系统和平台选择 IDE

如果您正在为 Linux 开发软件，需要使用 Linux 机器和基于 Linux 的 IDE。然而，C++

是一种可移植的语言，所以只要正确编写了代码，就应该在任何平台上以相同的方式编译和运行。当然，可能会在库上遇到问题，因为并非所有库都会默认安装，有些可能特定于平台。

严格遵循目标平台并不是必要的，有时甚至可能适得其反。例如，针对的是较旧或长期支持 (LTS) 版本的操作系统，可能无法使用最新的工具链版本。如果希望在不同于目标平台的平台上开发，可以这样做。

考虑交叉编译或远程开发。交叉编译涉及使用专用工具链，允许在一种平台（如 Windows）上运行的编译器为另一种平台（如 Linux）生成工件。这种方法在业界得到了广泛使用，并且得到了 CMake 的支持。另外，建议使用远程开发，将代码发送到目标机器，并在那里使用本地工具链进行构建。这种方法得到了许多 IDE 的支持，我们将在下一节中进行探讨。

选择支持远程开发的 IDE

虽然这不应该是首要标准，但在满足其他要求后，考虑 IDE 中的远程开发支持是有益的。随着时间的推移，即使是经验丰富的开发者也会遇到由于团队、项目甚至公司的变化而需要不同于他们常用操作系统的目标平台的项目。

如果首选 IDE 支持远程开发，可以继续使用它，利用在不同操作系统上编译和调试代码的能力，并在 IDE 的 GUI 中查看结果。远程开发相对于交叉编译的主要优势在于其集成的调试器支持，无需 CMake 项目级别的配置，过程更加简洁。此外，公司通常会提供强大的远程机器，允许开发者使用更便宜、更轻便的本地设备。

当然，有人可能会提出交叉编译提供了对开发环境的最大控制权，允许为测试进行临时更改。它不需要代码传输的带宽，支持低端的互联网连接或离线工作。然而，考虑到大多数软件开发都涉及互联网访问以获取信息，这可能是一个不那么关键的优势。使用像 Docker 这样的虚拟化环境可以运行本地生产副本并设置远程开发连接，提供安全性、可定制性和构建及部署容器的能力。

这里提到的考虑因素稍微倾向于在大公司工作的情况，在那里事情进展较慢，很难进行具有重大影响的改变。这些建议并不否定您决定根据需要优先考虑 IDE 的其他方面时，与 CMake 一起拥有完美完整体验的可能性。

3.1.2 安装工具链

IDE 集成了所有必要的工具来简化软件开发。这一过程的关键部分是构建二进制文件，有时在后台或即时进行，以为开发者提供额外信息。工具链是一系列工具的集合，如编译器、链接器、归档器、优化器、调试器，以及标准 C++ 库的实现。还包括其他有用的实用程序，如 bash、make、gawk、grep 等，用于构建程序。

一些 IDE 自带工具链或工具链下载器，而其他则没有。最好的做法是运行已安装的 IDE，并检查是否能够编译基本的测试程序。CMake 通常在配置阶段默认执行此操作，大多数 IDE 将此作为新项目初始化的一部分。如果此过程失败，IDE 或操作系统的包管理器可能会提示安装必要的工具。只需按照流程操作，因为这种流程通常准备得很充分。

如果没有提示，或者想使用特定的工具链，以下是一些基于平台的选项：

- GCC (<https://gcc.gnu.org/>) 用于 Linux、Windows (通过 MinGW 或 Cygwin)、macOS 以及许多其他平台。GCC 是最受欢迎和广泛使用的 C++ 编译器，支持大多数的平台和架构。

- Clang/LLVM (<https://clang.llvm.org/>) 用于 Linux、Windows、macOS 以及许多其他平台。Clang 是 C、C++ 和 Objective-C 编程语言的前端编译器，使用 LLVM 作为其后端。
- Microsoft Visual Studio/MSVC (<https://visualstudio.microsoft.com/>) 主要用于 Windows，通过 Visual Studio Code 和 CMake 提供跨平台支持。MSVC 是 Microsoft 提供的 C++ 编译器，通常在 Visual Studio IDE 内使用。
- MinGW-w64 (<http://mingw-w64.org/>) 用于 Windows。MinGW-w64 是原始 MinGW 项目进一步发展，旨在为 64 位 Windows 和新 API 提供更好的支持。
- Apple Clang (<https://developer.apple.com/xcode/cpp/>) 用于 macOS、iOS、iPadOS、watchOS 和 tvOS。Apple 的 Clang 版本，针对 Apple 的硬件和软件生态系统进行了优化，与 Xcode 集成。
- Cygwin (<https://www.cygwin.com/>) 用于 Windows。Cygwin 在 Windows 上提供了一个 POSIX 兼容的环境，允许使用 GCC 和其他 GNU 工具。

如果想快速开始，而不深入研究每个工具链，可以参考我的个人偏好：如果 IDE 没有提供工具链，Windows 上使用 MinGW，Linux 上使用 Clang/LLVM，macOS 上使用 Apple Clang。这些工具链都很好地适用于它们的主要平台，并且通常提供最佳体验。

3.1.3 使用本书的示例与 IDE

本书附带了一系列 CMake 项目的示例，可在官方 GitHub 仓库中找到：<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E>。

当然，探讨 IDE 的主题时，如何使用这个仓库，以及这里展示的所有 IDE？好吧，我们需要认识到，教你如何创建专业项目这本书本身并不是一个专业项目。它是一系列这样的项目，完成程度各不相同，可能在合理的情况下进行了简化。不幸的是（或者也许幸运的是？），IDE 并不是为了加载数十个项目，并方便地管理它们而构建的。它们通常将功能集中在加载一个正在编辑的项目上。

这让我们处于一个有些尴尬的位置：使用 IDE 导航示例集真的很难。当使用 IDE 加载示例集时，通过选择示例目录来打开它，大多数 IDE 会检测到多个 CMakeLists.txt 文件并要求选择一个。这样做之后，通常的初始化过程会发生，会写入临时文件，运行 CMake 配置和生成阶段，以使项目进入可以构建的状态。大多数 IDE 确实提供了在工作空间中切换不同目录（或项目）的方法，但可能并不像我们希望的那样直接。

如果您在这方面遇到困难，有两个选择：要么不使用 IDE 构建示例（而是使用控制台命令），要么每次将一个示例加载到新项目中。如果热衷于练习命令，我会推荐第一个选项，因为这些命令将来可能会派上用场，并且会让您更好地理解幕后发生的事情。这对于构建工程师来说通常是一个不错的选择，因为这种知识会经常使用。另一方面，如果主要作为开发者专注于代码的业务方面，那么早期使用 IDE 可能是最好的选择。

有了这些，让我们专注于回顾当今顶尖的 IDE，看看哪个可能最适合您。

3.2. CLion IDE

CLion 是一款付费的跨平台 IDE，适用于 Windows、macOS 和 Linux，由 JetBrains 开发。是的，这款软件是基于订阅的；截至 2024 年初，可以以 99.00 美元的价格获得一年的个人使用许可。更大的组织支付更多，初创企业支付更少。如果是学生或发布开源项目，可以获得免费许可。此外，还有 30 天的试用期来测试软件。这是列表中唯一一款不提供免费“社区”或精简版的 IDE。无论如何，这是一款由知名公司开发的坚实软件。

图 3.1 显示了 CLion IDE 在浅色模式（深色模式是默认选项）的外观：

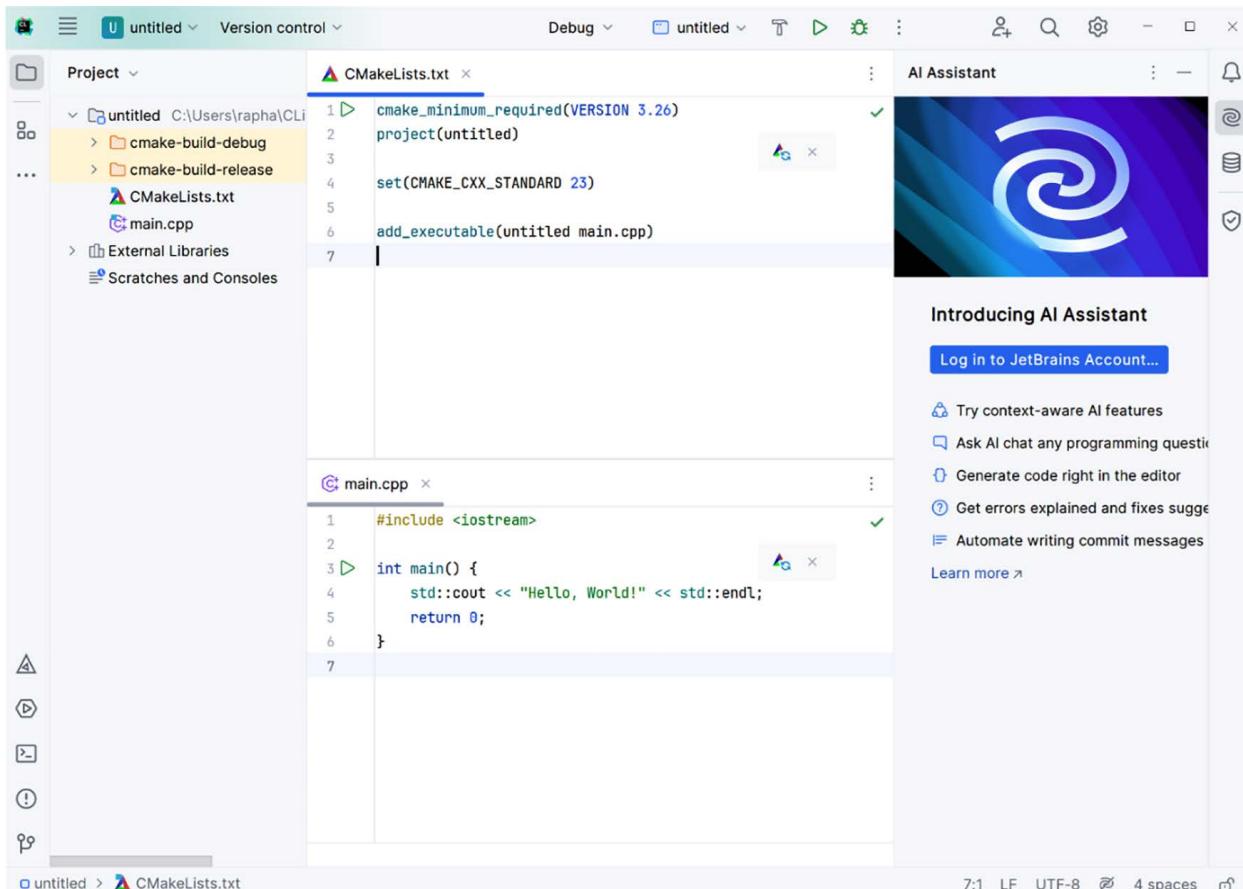


图 3.1: CLion IDE 的主窗口

这是一个功能齐全的 IDE，准备好应对可能抛给它的任何事情。来看看它如何脱颖而出的吧！

3.2.1 可能喜欢它的原因

与替代方案不同，CLion 支持 C 和 C++，这是它支持的第一种和唯一一种语言。这个 IDE 的许多功能都是专门为支持这个环境而设计的，并与 C/C++ 思维方式保持一致。当比较其他 IDE 的功能时，这一点非常明显：代码分析、代码导航、集成的调试器和重构工具，可以在像 Visual Studio IDE 这样的竞品中找到。然而，它们并不是那么深入和强大地面向 C/C++。

无论如何，CMake 可以完全集成到 CLion 中，是这个 IDE 中的首选项目格式。然而，Autotools 和 Makefile 项目处于早期支持状态，可以用来最终迁移到 CMake。CLion 原生

支持 CMake 的 CTest，并与许多单元测试框架集成，还有专门的流程来生成代码、运行测试和收集展示结果。可以使用 Google Test、Catch、Boost.Test 和 doctest。

我特别喜欢的一个功能是，能够使用 Docker 在容器中开发 C++ 程序——稍后会详细介绍。同时，让我们看看如何开始使用 CLion。

3.2.2 第一步

从官方网站下载 CLion (<https://www.jetbrains.com/clion>) 后，可以按照使用的平台进行通常的安装过程。CLion 在 Windows (图 3.2) 和 macOS (图 3.3) 上提供了一个合理的视觉安装程序。

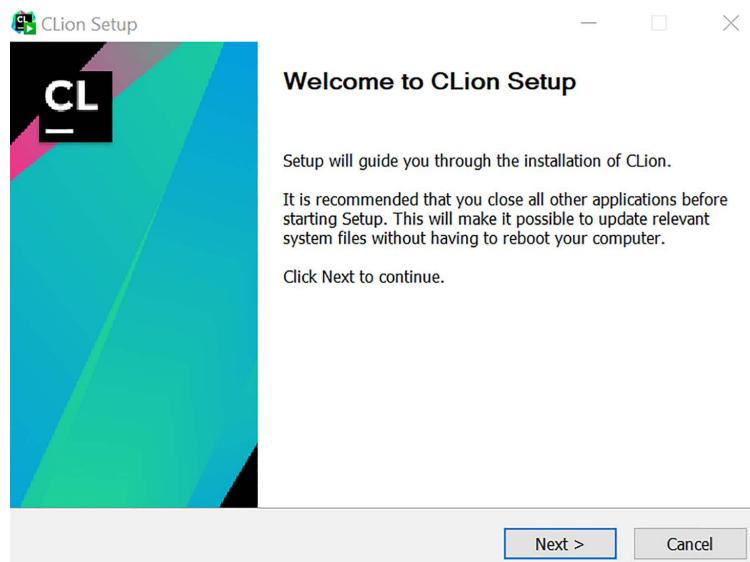


图 3.2: CLion 的 Windows 安装程序



图 3.3: CLion 的 macOS 安装程序

在 Linux 上，需要解压下载的存档并运行安装脚本：

```
tar -xzf CLion-<version>.tar.gz  
./CLion-<version>/bin/CLion.sh
```

这些说明可能已经过时，所以请务必去 CLion 官网进行确认。

首次运行时，将要求提供许可证代码或开始 30 天的免费试用期。选择第二个选项将允许试用 IDE 并确定它是否适合。接下来，将能够创建新项目并选择目标 C++ 版本。之后，CLion 将检测可用的编译器和 CMake 版本，并尝试构建一个测试项目以确认一切设置正确。在某些平台（如 macOS）上，可能会自动提示安装所需的开发者工具。在其他平台，可能需要自己设置，并确保它们在 PATH 环境变量中可用。

接下来，确保工具链已根据需求进行配置。工具链是按项目配置的，因此创建一个默认的 CMake 项目。然后，导航到设置/首选项 (Ctrl/Command + Alt + S) 并选择构建、执行、部署 > CMake。在这个选项卡上，可以配置构建配置文件（图 3.3）。可能很有用的是添加一个 Release 配置文件，来构建没有调试符号的优化工作件。要添加一个，只需在上方配置文件列表上按加号按钮。CLion 将创建一个默认的 Release 配置文件。还可以在主窗口顶部下拉菜单中，切换配置文件。

现在，可以简单地按 F9 来编译并运行带有调试器的程序。接下来，阅读 CLion 的官方文档，因为其中有很多有用的功能可以探索。

接下来，我将绍我最喜欢的部分：调试器。

3.2.3 高级功能：强大的调试器

CLion 的调试功能非常先进，特别是为 C++ 量身定制。我非常高兴地发现了一个最新的添加功能——CMake 调试，包括许多标准的调试功能：通过代码、断点、观察、内联值探索等。当事情并不像预期那样工作时，能够探索不同作用域（缓存、ENV 和当前作用域）中的变量是非常方便的。

对于 C++ 调试，您将获得 GNU 项目调试器 (GDB) 提供的许多标准功能，例如汇编视图、断点、单步执行、观察点等，但也有重大改进。在 CLion 中，会发现一个并行堆栈视图，允许以图形化的方式查看所有线程，及其当前堆栈帧。此外，还有一个高级内存视图功能，用于查看运行程序在 RAM 中的布局，并实时修改内存。CLion 还提供了多种其他工具来帮助您了解发生了什么：寄存器视图、代码反汇编、调试器控制台、核心转储调试、可执行文件的调试等。

最后，CLion 有一个非常出色的评估表达式功能，它非常神奇，甚至允许程序执行过程中修改对象。只需右键点击代码行，然后从菜单中选择此功能。

这就是关于 CLion 的全部内容；现在是时候看看另一个 IDE 了。

3.3. Visual Studio Code

Visual Studio Code (VS Code) 是一款免费的、跨平台的 IDE，适用于 Windows、macOS 和 Linux，由微软开发。不要将它与另一款微软产品混淆，即 Visual Studio IDE。

VS Code 因其广泛的扩展生态系统和对数百种编程语言的支持而受到青睐（据估计，超过 220 种不同的语言！）。微软收购 GitHub 时，VS Code 就传言为 Atom 的替代。

IDE 的整体设计非常出色，如图 3.4 所示。

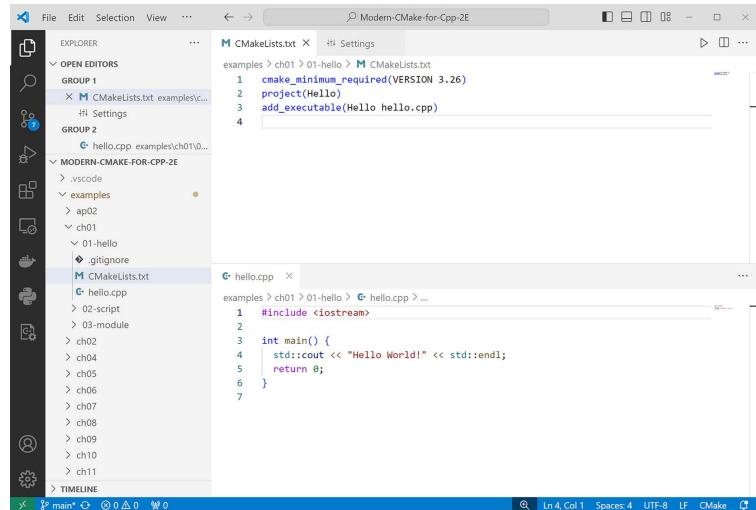


图 3.4: VS Code 的主窗口

现在，找出 VS Code 之所以特别的原因。

3.3.1 可能喜欢它的原因

在 VS Code 支持的众多语言中，C++ 并不是优先考虑的，有了许多高级语言扩展。这种权衡带来的好处是，能够在同一环境中根据需要切换多种语言。

这个工具的学习曲线稍微有些陡峭，大多数扩展都符合基本 UI 功能，而不是自行实现的高级界面。许多功能将通过命令面板（通过按 F1 访问）提供，这要求输入命令，而非点击图标或按钮。为了保持 VS Code 干净、快速且免费，这是一个合理的牺牲。实际上，这个 IDE 加载速度非常快，以至于我更喜欢将它作为通用文本编辑器使用。

VS Code 的真正强大之处在于它拥有大量出色的扩展，其中大多数都免费。对于 C++ 和 CMake，有专门的扩展可用，我们将在下一节看看如何配置它们。

3.3.2 第一步

VS Code 可以从官方网站获得：<https://code.visualstudio.com/>。该网站提供了 Windows 和 macOS，以及许多 Linux 发行版（Debian、Ubuntu、Red Hat、Fedora、SUSE）的下载列表。按照平台的常规流程安装软件后，需要通过扩展市场（通过按 Ctrl/Command + Shift + X）安装一堆扩展。以下是一些推荐的入门扩展：

- C/C++ by Microsoft
- C/C++ Extension Pack by Microsoft
- CMake by twxs
- CMake Tools by Microsoft

它们将提供标准的代码高亮和直接从 IDE 编译、运行和调试代码的能力，但需要自己安装工具链。通常，VS Code 会在开始打开相关文件时在弹出窗口中建议安装扩展，所以不必特意去寻找。

如果参与远程项目，我建议安装 Microsoft 的 Remote-SSH 扩展，因为这会使体验更加连贯和舒适；这个扩展不仅处理文件同步，还允许通过远程机器的调试器远程调试。

然而，还有一个更有趣的扩展。

3.3.3 高级功能：Dev Containers

如果要将应用程序部署到生产环境，无论是发送编译的工件还是运行构建过程，确保所有依赖项都存在至关重要；否则，将遇到各种问题。即使考虑了所有依赖项，不同的版本或配置也可能导致解决方案与开发或测试环境的行为不同，我多次遇到过这种情况。在虚拟化变得普遍之前，处理环境问题只是生活的一部分。

随着轻量级容器如 Docker 的引入，事情变得更加简单。可以运行一个包含您服务的最小化操作系统，并将其隔离到自己的空间中，并将所有依赖项打包到容器中。

直到最近，开发容器涉及手动构建、运行和通过 IDE 的远程会话连接到容器。这个过程并不太难，但它需要手动步骤，不同开发人员可能会以不同的方式执行。

近年来，微软发布了一个名为 Dev Containers (<https://containers.dev/>) 的开放标准，该规范主要由一个 devcontainer.json 文件组成，可以在项目仓库中放置该文件，指导 IDE 如何在一个容器中设置其开发环境。

要使用此功能，只需安装 Microsoft 的 Dev Containers 扩展，并将其指向一个适当准备的项目仓库。如果不介意切换主 CMakeLists.txt，可以尝试使用本书的仓库：

```
git@github.com:PacktPublishing/Modern-CMake-for-Cpp-2E.git
```

我可以确认，其他 IDE，如 CLion，也在采用这个标准。如果正面临这种情况，这是一个很好的实践。现在是时候转向微软家族的下一个产品了。

3.4. Visual Studio IDE

Visual Studio (VS) IDE 是微软开发的一款适用于 Windows 的 IDE。VS 曾为 macOS 提供支持，但将在 2024 年 8 月弃用，要将其与微软的另一个 IDE——VS Code 区分开来。

VS 有几个版本：社区版、专业版和企业版。社区版免费，允许一家公司最多有五名用户。规模更大的公司需要支付许可费用，每月每位用户起价 45 美元。图 3.5 展示了 VS 2022 的外观：

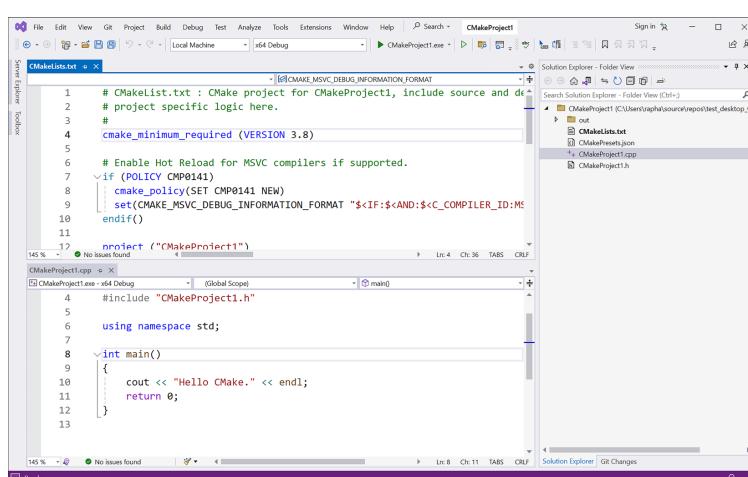


图 3.5: VS 2022 的主窗口

和其他 IDE 一样，也可以启用暗黑模式。

3.4.1 可能喜欢它的原因

这个 IDE 与 VS Code 共享许多功能，提供类似但更加精致的体验。套件中充满了功能，其中许多功能利用了图形用户界面、向导和可视元素。大多数这些功能都直接开箱即用，而不是通过扩展（尽管仍然有一个大型且广泛的包市场以获取更多功能）。

根据版本的不同，测试工具将涵盖广泛的测试：单元测试、性能测试、负载测试、手动测试、Test Explorer、测试覆盖率、IntelliTest 和代码分析。特别是分析器，这是一款非常宝贵的工具，在社区版中可用。

如果正在设计 Windows 桌面应用程序，VS 提供了可视编辑器和大量组件。对通用 Windows 平台 (UWP) 的支持非常广泛，这是 Windows 10 引入的 Windows 应用程序的 UI 标准。这种支持允许实现光滑、现代的设计，并且针对不同屏幕上的自适应控件进行了高度优化。

值得一提的是，尽管 VS 是一个仅限 Windows 的 IDE，但仍然可以开发针对 Linux 和移动平台 (Android 和 iOS) 的项目。此外，还支持使用 Windows 本地库和 Unreal Engine 的游戏开发者。

准备好亲自体验它的工作方式了吗？以下是开始的方法。

3.4.2 第一步

这个 IDE 仅适用于 Windows，并遵循标准的安装过程。首先从<https://visualstudio.microsoft.com/free-developer-offers/>下载安装程序。运行安装程序后，需要选择版本（社区版、专业版或企业版），并选择想要的安装的组件：

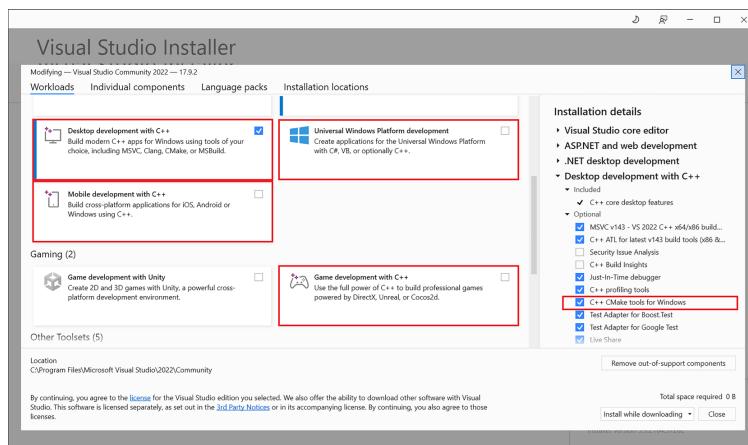


图 3.6: VS IDE 安装程序窗口

组件是允许 VS 支持特定语言、环境或程序格式的功能集，一些组件包括 Python、Node.js 或 .NET。当然，我们感兴趣的是与 C++ 相关的那些（图 3.6）；其针对不同的用例提供了广泛的支持：

- 使用 C++ 的桌面开发
- 使用 C++ 的通用 Windows 平台开发
- 使用 C++ 的游戏开发
- 使用 C++ 的移动开发
- 使用 C++ 的 Linux 开发

选择所需应用程序的选项并点击安装。不用担心安装所有选项——可以再次运行安装程序来修改选择。如果决定更精确地配置工作负载组件，请确保启用 C++ CMake 工具选项以获得对 CMake 的支持。

安装后，可以启动 IDE 并选择启动窗口上的“创建新项目”，将看到基于之前安装的工作负载的多个模板。要与 CMake 一起工作，请选择 CMake 项目模板。其他选项不一定使用 CMake。创建项目后，可以通过点击窗口顶部绿色的播放按钮开启；代码将编译，将看到基本的程序执行，输出如下：

```
Hello CMake.
```

现在，准备好使用 Visual Studio 与 CMake 一起工作了吗？

3.4.3 高级功能：热重载调试

运行 Visual Studio 可能会更耗资源并且启动时间更长，但它提供了许多无与伦比的功能。其中一个重要的改变者是热重载。以下是其使用方式：打开一个 C++ 项目，使用调试器启动它，在代码文件中进行更改，点击热重载按钮（或 Alt + F10），你的更改将立即在运行的应用程序中反映出来，同时保持状态。

为了确保热重载支持已启用，请在项目 > 属性 > C/C++ > 常规菜单中设置这两个选项：

- 调试信息格式必须设置为 **用于编辑并继续的程序数据库 /ZI**
- 启用增量链接必须设置为 **Yes /INCREMENTAL**

热重载的背后机制可能看起来像魔法，但它是一个非常实用的功能。存在一些限制，例如：对全局/静态数据、对象布局或“时间旅行”更改（如更改已构建对象的构造函数）的更改。

更多关于热重载的信息可以在官方文档中找到：<https://learn.microsoft.com/en-us/visualstudio/debugger/hot-reload>。

这就是我们对这三个主要 IDE 的探索。起初的学习曲线可能看起来很陡峭，但如果投入学习这些 IDE，当转向更高级的任务时，会很快看到回报的。

3.5. 总结

本章深入探讨了使用集成开发环境（IDE）来优化编程过程，特别是那些与 CMake 深度集成的 IDE。它为初学者和有经验的专业人士提供了全面指南，详细介绍了使用 IDE 的好处，以及如何选择最适合个人或组织需求的 IDE。

首先讨论了 IDE 在提升开发速度和代码质量方面的重要性，解释了什么是 IDE，以及它如何通过整合代码编辑器、编译器和调试器等工具简化软件开发的各个步骤。接着，简要回顾了工具链，解释了如果系统中没有安装工具链的必要性，并列出了最常见的几个选择。

讨论了如何开始使用 CLion 及其独特的功能，并深入了解了其调试功能。VS Code 是微软推出的一款免费、跨平台的 IDE，以其庞大的扩展生态系统和对众多编程语言的支持而闻名。指导各位完成了初始设置和关键扩展的安装，并介绍了一个名为 Dev Containers 的高级功能。仅适用于 Windows 的 VS IDE 提供了一个精致、功能丰富的环境，适合各种用户需求。设置过程、关键特性和热加载调试功能也进行了介绍。

每个 IDE 部分都提供了为何选择特定 IDE 的见解，开始的步骤，以及使该 IDE 脱颖而出的高级功能。我们还强调了远程开发支持的概念，突出了其在行业中的日益重要性。

总结来说，本章为开发者提供了一个基础指南，帮助他们理解和选择 IDE，清晰地概述了顶级选项、它们的独特优势，以及如何与 CMake 结合使用以提升编码效率和项目管理。

下一章中，我们将了解使用 CMake 进行项目设置的基础知识。

3.6. 扩展阅读

- Qt Creator IDE，另一个支持 CMake 的选项：

<https://www.qt.io/product/development-tools>

- Eclipse IDE C/C++ 开发者，也支持 CMake：

<https://www.eclipse.org/downloads/packages/release/2023-12/r/eclipse-idecc-developers>

- macOS 的 Xcode 也可以与 CMake 一起使用：

<https://medium.com/practical-coding/migrating-to-cmake-in-c-and-getting-it-working-with-xcode-50b7bb80ae3d>

- CodeLite 也是一个选择，有 CMake 插件：

<https://docs.codelite.org/plugins/cmake/>

第 4 章 设置你的第一个 CMake 项目

我们现在已经收集了足够的信息，可以开始讨论 CMake 的核心功能：构建项目。在 CMake 中，一个项目包含了所有源文件和必要的配置，以管理将解决方案变为现实的过程。配置过程从执行所有检查开始：验证目标平台是否受支持，确保所有必需的依赖项和工具的存在，并确认提供的编译器与所需特性兼容。

完成初步检查后，CMake 继续生成一个定制的构建系统，该构建系统针对所选的构建工具。然后，执行构建编译源文件，并将它们与各自的依赖项链接在一起，以创建输出工件。

生成的工件可以通过不同的方式分发给消费者，可以直接与用户共享作为二进制包，允许用户使用包管理器将它们安装到自己的系统中。另一种方式是将其作为单一的可执行安装程序分发。此外，最终用户还可以通过访问开源仓库中的项目自行创建工作。这种情况下，用户可以使用 CMake 在自己的机器上编译项目，并随后进行安装。

充分利用 CMake 项目，可以显著提高开发体验和生成的代码的整体质量。通过利用 CMake 的力量，许多日常任务可以自动化，在构建后执行测试和运行代码覆盖检查器、格式化器、验证器、校验器和其他工具。这种自动化不仅节省了时间，还确保了一致性，并在整个开发过程中推广了代码质量。

要充分发挥 CMake 项目的潜力，首先需要做出一些关键决策：如何正确配置整个项目，以及如何分割项目和设置源树，以便所有文件都能整齐地组织在正确的目录中。从一开始就建立一个连贯的结构和组织，CMake 项目就可以有效地管理和扩展。

接下来，我们将查看项目的构建环境。将了解我们正在使用的架构、可用的工具、支持的功能，以及正在使用的语言标准。为了确保一切同步，我们将编译一个测试 C++ 文件，并查看选择的编译器是否符合我们为项目设定的标准要求。这一切都是为了确保项目、工具，以及选择标准之间的无缝配合。

本章中，将包含以下内容：

- 理解基本指令和命令
- 分割项目
- 项目结构
- 设置环境范围
- 配置工具链
- 禁用源内构建

4.1. 示例下载

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch04>找到本章中出现的代码文件。

为了构建本书提供的示例，请使用以下推荐命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的目录。

4.2. 了解基本的指令和命令

第 1 章中，已经了解了一个简单的项目定义。这是一个带有 `CMakeLists.txt` 文件的目录，其中包含几个配置语言处理器的命令：

chapter01/01-hello/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Hello)
3 add_executable(Hello hello.cpp)
```

在同一章节的“项目文件”部分，我们了解了一些基本命令。

现在，让我们在这里深入介绍它们。

4.2.1. 指定最低 CMake 版本

项目文件和脚本的顶部使用 `cmake_minimum_required()` 命令，这个命令不仅验证系统是否具有正确的 CMake 版本，还会隐式触发另一个命令 `cmake_policy(VERSION)`，后者指定项目要使用的策略。这些策略定义了命令在 CMake 中的行为方式，它们随着 CMake 的发展，以及对支持的语言和 CMake 本身的改进而引入。

为了保持语言的清晰和简单，CMake 团队在出现向后不兼容的更改时引入了策略。每个策略都启用了与该更改相关的新行为。这些策略确保项目可以适应 CMake 不断发展的特性和功能，同时保持与旧代码库的兼容性。

通过调用 `cmake_minimum_required()`，我们告诉 CMake 需要应用在参数中提供的版本配置的默认策略。当 CMake 升级时，不必担心它会破坏项目，因为新版本的新策略不会启用。

策略可能会影响 CMake 的每一个方面，包括 `project()` 等其他重要命令。因此，在 `CMakeLists.txt` 文件开始时设置正在使用的版本非常重要；否则，将得到警告和错误。每个 CMake 版本都引入了许多策略。然而，除非在将旧项目升级到最新 CMake 版本时遇到挑战，否则没有必要深入细节。这种情况下，建议参考官方文档中关于策略的全面信息和指导：<https://cmake.org/cmake/help/latest/manual/cmake-policies.7.html>。

4.2.2. 定义语言和元数据

建议在 `cmake_minimum_required()` 之后立即放置 `project()` 命令，这样做将确保在配置项目时使用正确的策略。可以使用以下两种形式之一：

```
project(<PROJECT-NAME> [<language-name>...])
```

或者：

```
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>`，但其他参数可选。调用此命令将隐式设置以下变量：

```
PROJECT_NAME
CMAKE_PROJECT_NAME (only in the top-level CMakeLists.txt)
PROJECT_IS_TOP_LEVEL, <PROJECT-NAME>_IS_TOP_LEVEL
PROJECT_SOURCE_DIR, <PROJECT-NAME>_SOURCE_DIR
PROJECT_BINARY_DIR, <PROJECT-NAME>_BINARY_DIR
```

支持哪些语言？相当多。而且可以在同一时间使用多种语言！以下是可以用来配置项目的语言关键字列表：

- ASM, ASM_NASM, ASM_MASM, ASMMARMASM, ASM-ATT: 汇编语言
- C: C
- CXX: C++
- CUDA: Nvidia 的统一计算设备架构
- OJJC: Objective-C
- OJJCXX: Objective-C++
- Fortran: Fortran
- HIP: 用于 Nvidia 和 AMD 平台的异构（计算）接口便携性
- ISPC: 隐式 SPMD 程序编译器的语言
- CSharp: C#
- Java: Java（需要额外步骤，请参阅官方文档）

CMake 默认启用 C 和 C++，因此可能只想为 C++ 项目明确指定 CXX。为什么？因为 `project()` 命令将检测和测试您选择的语言的可用编译器，所以声明所需的编译器，将在配置阶段跳过对未使用语言的检查，从而节省时间。

指定 VERSION 关键字将自动设置可以用来配置包的变量，或者在编译期间在头文件中暴露的变量：

```
PROJECT_VERSION, <PROJECT-NAME>_VERSION
CMAKE_PROJECT_VERSION (only in the top-level CMakeLists.txt)
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
```

还可以设置 DESCRIPTION 和 HOMEPAGE_URL，这将为了类似目的设置以下变量：

```
PROJECT_DESCRIPTION, <PROJECT-NAME>_DESCRIPTION  
PROJECT_HOMEPAGE_URL, <PROJECT-NAME>_HOMEPAGE_URL
```

`cmake_minimum_required()` 和 `project()` 命令会创建一个基本的项目列表文件，并初始化一个空项目。虽然对于小型、单文件项目来说，结构可能不是一个大问题，但随着代码库的扩展，它变得至关重要。如何为此做准备呢？

4.3. 划分项目

随着解决方案在代码行数和包含的文件数量上的增长，就必须面对一个挑战：要么开始划分项目，要么面临复杂性的风险。解决这个问题有两种方法：分割 CMake 代码，并将源文件重新定位到子目录中。在这两种情况下，目标都是遵循“关注点分离”的设计原则。简单来说，我们将代码分解成更小的部分，将紧密相关的功能组合在一起，同时保持其他代码段分离，以建立清晰的界限。

第一章中讨论列表文件时。简要提到了项目划分。我们谈到了 `include()` 命令，其允许 CMake 执行外部文件中的代码。

这种方法有助于分离关注点，但作用有限——专业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，但如果作者不小心，其仍然会污染全局变量范围。

调用 `include()` 并不会在文件中，已经定义的范围之外引入任何范围或隔离。让我们通过一个例子来看看，为什么这是一个潜在的问题，这个例子是支持小型汽车租赁公司的软件。它将有多个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。如果所有这些文件都放在一个目录中，查找起来会是一场噩梦。因此，在项目的主目录中创建了一些目录，并将相关的文件移动到其中。我们的 `CMakeLists.txt` 文件看起来可能像这样：

ch04/01-partition/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)  
2 project(Rental CXX)  
3 add_executable(Rental  
4     main.cpp  
5     cars/car.cpp  
6     # more files in other directories  
7 )
```

这很棒，如你所见，我们仍然在顶层文件中包含了来自嵌套目录的源文件列表！为了增加关注点的分离，我们可以将源文件列表提取到另一个列表文件中，并将其存储在 `sources` 变量中：

ch04/02-include/cars/cars.cmake

```
1 set(sources  
2     cars/car.cpp  
3     # more files in other directories  
4 )
```

现在我们可以使用 `include()` 命令引用这个文件，以访问 `sources` 变量：

ch04/02-include/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(Rental CXX)
3 include(cars/cars.cmake)
4 add_executable(Rental
5     main.cpp
6     ${sources} # for cars/
7 )
```

CMake 会在与 `add_executable` 相同的作用域中设置 `sources`，用所有文件填充该变量。这个解决方案有效，但有几个缺点：

- **嵌套目录中的变量将污染顶层作用域（反之亦然）**：一个简单的例子中这不是问题，但在更复杂的多级树结构中，使用了多个变量，这可以迅速成为一个难以调试的问题。如果有多个包含文件都定义了自己的 `sources` 变量呢？
- **所有目录将共享相同的配置**：这个问题在项目经过多年的发展后会凸显。没有粒度，必须统一对待每个源文件，不能为代码的某些部分指定不同的编译标志，选择更新的语言版本，以及在代码的选定区域消除警告。一切都是全局的，所以需要同时对所有翻译单元进行更改。
- **存在共享的编译触发器**：任何配置的更改都意味着所有文件都需要重新编译，即使对其中一些文件得更改无意义。

所有路径都是相对于顶级目录的：注意在 `cars.cmake` 中，我们必须提供到 `cars/car.cpp` 文件的完整路径。这导致了很多重复的文本，损害了可读性，并且违反了干净编码的“不要重复自己”（DRY）原则（不必要的重复会导致错误）。

另一种方法是使用 `add_subdirectory()` 命令，它引入了变量作用域，让我们来了解一下。

4.3.1. 使用子目录管理作用域

按照文件系统的自然结构来构建项目，是一种常见的做法，其中嵌套的目录代表应用程序的离散元素，业务逻辑、GUI、API 和报告，最后是包含测试、外部依赖、脚本和文档的独立目录。为了支持这个概念，CMake 提供了以下命令：

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

正如已经确立的，这将添加一个源目录到我们的构建中。可选地，我们可以提供一个路径，构建的文件将写入该路径 (`binary_dir` 或构建树)。`EXCLUDE_FROM_ALL` 关键字将禁用子目录中定义的目标的自动构建（将在下一章讨论目标）。这可能有用于分隔项目中不需要核心功能的部分（例如：示例或扩展）。

`add_subdirectory()` 将计算 `source_dir` 路径（相对于当前目录）并解析其中的 `CMakeLists.txt` 文件。这个文件在目录作用域内解析，消除了前一种方法中提到的问题：

- 变量隔离到嵌套作用域。

- 嵌套工件可以独立配置。
- 修改嵌套的 CMakeLists.txt 文件不需要重新构建不相关的目标。
- 路径定位到目录，并且可以添加到父级包含路径。

这是我们的 add_subdirectory() 示例的目录结构：

```
|--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：

ch04/03-add_subdirectory/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(Rental CXX)
3 add_executable(Rental main.cpp)
4 add_subdirectory(cars)
5 target_link_libraries(Rental PRIVATE cars)
```

最后一行用于将 cars 目录的工件链接到 Rental 可执行文件。这是一个特定于目标的命令，我们将在第 5 章中详细讨论。

来看看嵌套的列表文件长什么样：

ch04/03-add_subdirectory/cars/CMakeLists.txt

```
1 add_library(cars OBJECT
2     car.cpp
3     # more files in other directories
4 )
5 target_include_directories(cars PUBLIC .)
```

这个例子中，使用 add_library() 来生成一个全局可见的目标 cars，并使用 target_include_directories() 将 cars 目录添加到其公共包含目录中。告诉 CMake cars.h 所在的位置，所以当使用 target_link_libraries() 时，main.cpp 文件可以在不提供相对路径的情况下使用该头文件：

```
1 #include "car.h"
```

我们可以在嵌套的列表文件中看到 add_library() 命令，那在这个例子中开始使用库了吗？没有。我们使用了 OBJECT 关键字，表示只对生成对象文件（与上一个例子完全一样），只是将它们分组在一个逻辑目标 (cars) 下。你可能已经对目标有了一些概念，保持这个想法——我们将在下一章详细解释。

4.3.2. 何时使用嵌套项目

前一节中，简要提到了在 `add_subdirectory()` 命令中使用的 `EXCLUDE_FROM_ALL` 参数，用于指示代码库中的外部元素。CMake 文档建议，如果有这样的部分存在于源树中，应该在其 `CMakeLists.txt` 文件中有自己的 `project()` 命令，以便生成自己的构建系统，且可以独立构建。

还有其他场景这种情况会有用吗？当然。例如，正在使用一个 CI/CD 构建的多个 C++ 项目时（可能是在构建框架或一组库时）。或者，正在将构建系统从传统的解决方案（如使用纯 `makefile` 的 GNU Make）移植过来。在这种情况下，可能希望有一个选项，慢慢地将事物分解成更独立的片段——将它们放入单独的构建过程中，或者只是在更小的范围内工作，就可以通过像 CLion 这样的 IDE 加载。可通过在嵌套目录的列表文件中，添加 `project()` 命令来实现这一点。只是不要忘记在它前面加上 `cmake_minimum_required()`。

既然支持项目嵌套，我们能否以某种方式连接并排构建的相关项目？

4.3.3. 保持外部项目外部化

虽然在 CMake 中从另一个项目引用内部，技术上可行，但这不是常规或推荐的做法。CMake 确实为这种情况提供了一些支持，包括使用 `load_cache()` 命令从另一个项目的缓存中加载数据。然而，使用这种方法可能会导致循环依赖和项目耦合的问题。最好避免使用这个命令并做出决定：相关项目应该是嵌套的，通过库连接，还是合并成一个项目？

这些是我们可用的划分工具：包括引入列表文件、添加子目录和嵌套项目。但我们应该如何使用它们，以保持项目易于维护、导航和扩展呢？为此，我们需要一个明确定义的项目结构。

4.4. 项目结构

随着项目的增长，要在项目列表文件和源码中找到所需内容变得越来越困难。因此，项目保持整洁就非常重要。

当需要交付一些重要且时间紧迫的更改，但这些更改不适合项目中的任一目录。现在，需要推送一个清理提交来重新组织文件结构，以便更改可以融入。或者更糟，决定随意放置，并添加一个 TODO，以后再处理这个问题。

这些问题累积起来，随着技术债务的增加，维护代码的成本也随之上升。当实时系统出现严重故障需要快速修复，或者不熟悉代码库的人需要引入更改时，这种情况会变得非常棘手。

因此，需要一个好的项目结构。但这意味着什么呢？可以从软件开发的其他领域，如系统设计，借鉴一些规则。项目应该具有以下特点：

- 易于查找和扩展
- 边界清晰（项目特定文件应包含在项目目录中）
- 单个目标遵循层次树结构

没有一个确定的解决方案，但在网上可用的各种项目结构模板中，我建议使用以下这个简单且可扩展的模板：

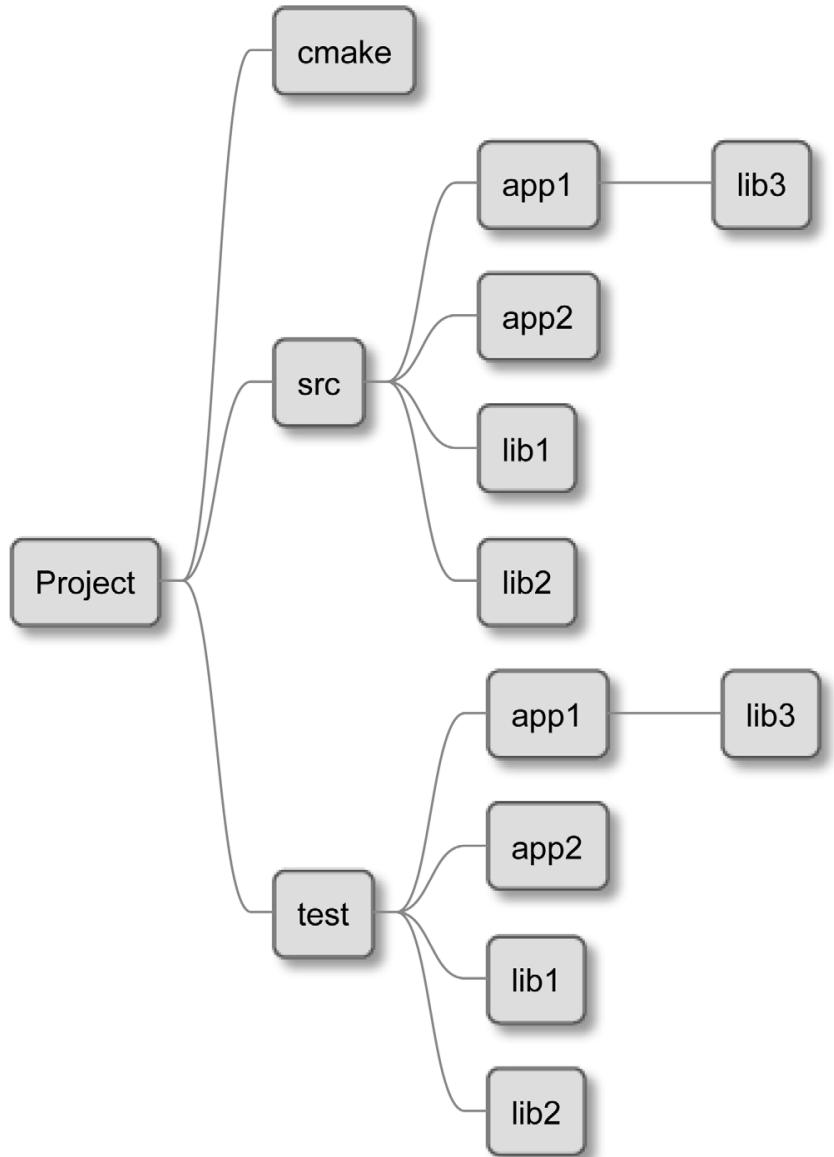


图 4.1：项目结构的一个示例

此项目为以下组件定义了目录：

- `cmake`: 共享宏和函数, `find_modules` 和一次性脚本
- `src`: 二进制库文件, 以及源文件和头文件
- `test`: 测试源码

此结构中, `CMakeLists.txt` 文件应存在于以下目录中: 顶层项目目录、`test` 和 `src` 及其所有子目录。主列表文件不应自行声明构建步骤, 而应配置项目的一般方面, 并通过 `add_subdirectory()` 命令将构建责任委托给嵌套的列表文件。如有需要, 这些列表文件可以进一步将工作委托给更深的层次。

Note

一些开发者建议将可执行文件与库分开, 并创建两个顶级目录而不是一个: `src` 和 `lib`。CMake 对这两种工件的处理方式相同, 在这个层次上的分离并不重要。

对于较大的项目，src 目录中有多个子目录会非常方便。但若只构建一个可执行文件或库，就可以省略，直接在 src 中存储源文件。无论如何，记得在那里添加一个 CMakeLists.txt 文件，并执行嵌套的列表文件。一个简单目标的文件树如下所示：

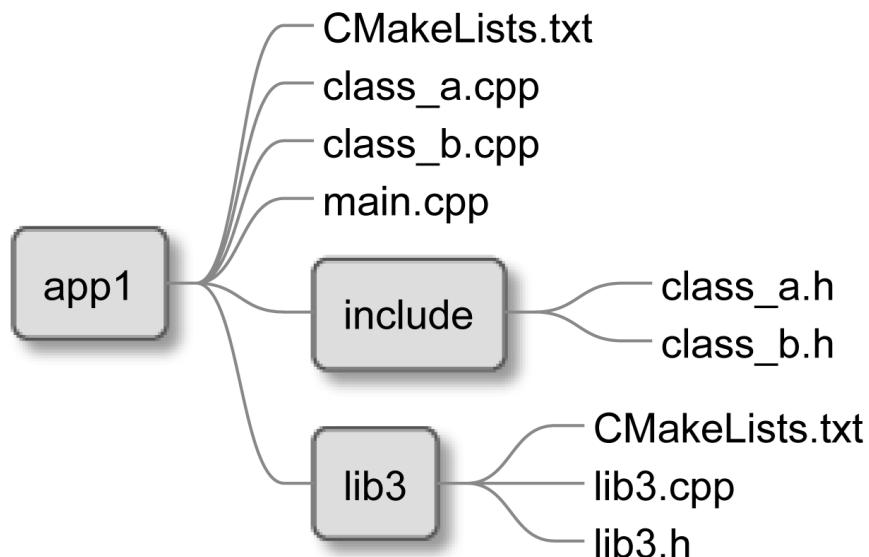


图 4.2：可执行文件的目录结构

图 4.1 中，可以看到 src 目录根目录下有一个 CMakeLists.txt 文件——将配置关键的项目设置，并包含来自嵌套目录的所有列表文件。app1 目录（如图 4.2 所示）包含另一个 CMakeLists.txt 文件，以及.cpp 实现文件：class_a.cpp 和 class_b.cpp。还有一个 main.cpp 文件，其中包含可执行文件的入口。

CMakeLists.txt 文件应定义一个目标，使用这些源文件构建一个可执行文件——我们将在下一章学习如何做到这一点。

头文件放置在 include 目录中，可以用来为其他 C++ 翻译单元声明符号。接下来，有一个仅对此可执行文件特定的库的 lib3 目录（项目中其他地方使用的库或对外部提供的库应位于 src 目录中）。这种结构提供了极大的灵活性，并可以轻松地扩展项目。随着添加更多类，可以方便地将它们分组到库中以提高编译速度。

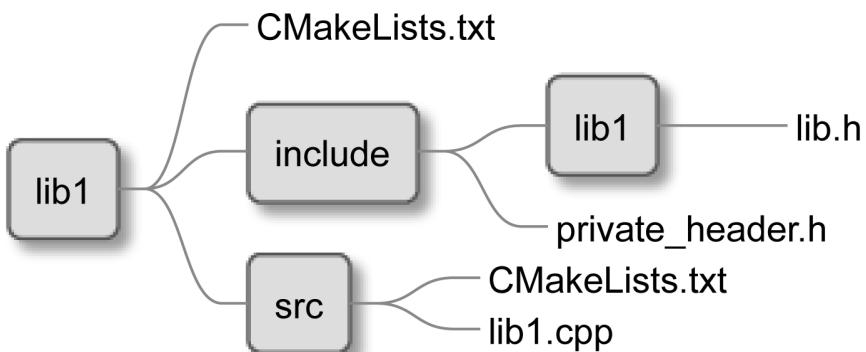


图 4.3：库的目录结构

库应遵循与可执行文件相同的结构：在 include 目录中添加了一个可选的 lib1 目录。当库打算超出项目范围时，会包含此目录，其包含其他项目在编译期间将使用的公共头文件。

我们已经讨论了文件如何在目录结构中布局。现在，是时候看看单个 CMakeLists.txt 文件如何组合成一个项目。

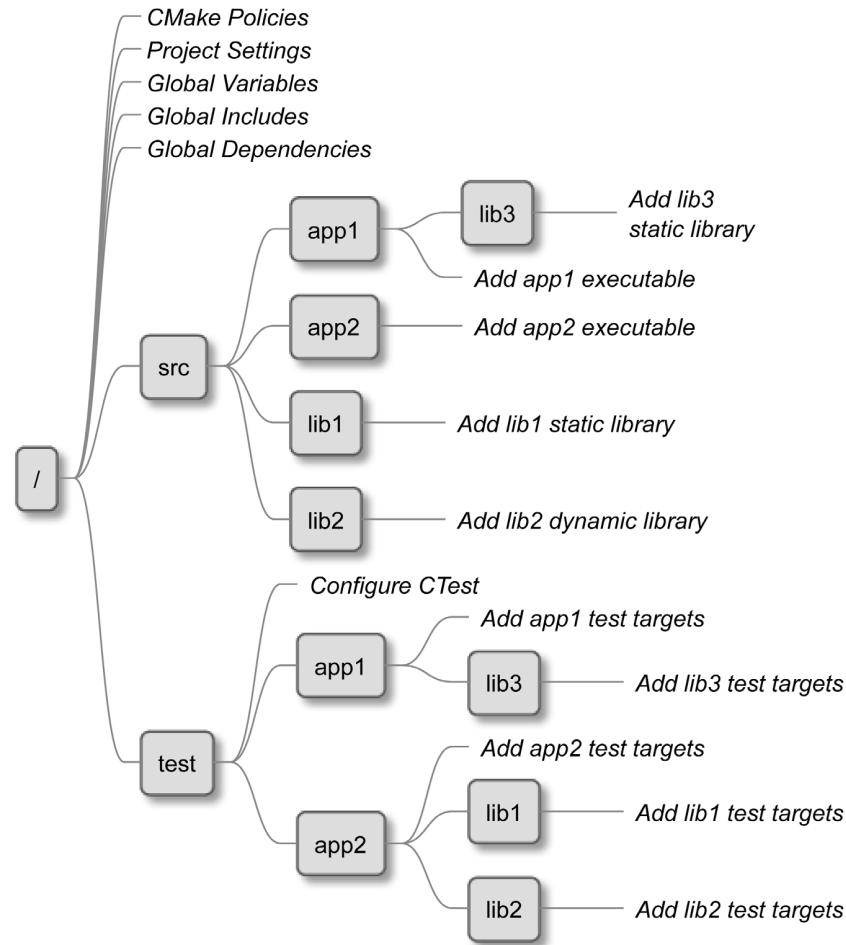


图 4.4：CMake 如何将列表文件合并到单个项目中

图中，每个框代表一个位于每个目录中的 CMakeLists.txt 列表文件，而其中的斜体标签代表每个文件执行的操作（从上到下）。再次从 CMake 的角度分析这个项目（有关详细信息，请查看 ch04/05-structure 目录中的示例）：

1. 执行从项目的根开始——即源树顶层的 CMakeLists.txt 列表文件。该文件将设置所需的最低 CMake 版本和相应的策略，设置项目名称、支持的语言和全局变量，并包含 cmake 目录中的文件，以便内容全局可用。
2. 下一步是调用 add_subdirectory(src bin) 命令进入 src 目录的范围（希望将编译后的工件放在 <binary_tree>/bin，而不是/bin 中）。
3. CMake 读取 src/CMakeLists.txt 文件，并发现其唯一目的是添加四个嵌套子目录：app1、app2、lib1 和 lib2。
4. CMake 进入 app1 的变量范围，并了解到另一个嵌套库 lib3，它有自己的 CMakeLists.txt 文件；然后进入 lib3 的范围，这是对目录结构的深度优先遍历。
5. lib3 库添加了一个同名静态库目标，CMake 返回到 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 返回到 src 的父范围。

7. CMake 进入剩余的嵌套范围，并执行列表文件，直到所有 `add_subdirectory()` 调用完成。
8. CMake 返回到顶层范围，并执行剩余的命令 `add_subdirectory(test)`。CMake 每次都会进入新的范围，并执行相应列表文件中的命令。
9. 收集并检查所有目标的正确性。CMake 现在有了生成构建系统所需的所有信息。

前面的步骤按照在列表文件中编写命令的确切顺序发生。某些情况下，这个顺序很重要，有时可能没有那么关键。

那么，何时是创建包含项目所有元素的目录的正确时机呢？应该一开始就做——为未来创建所需的一切，并保持目录为空——还是等到实际上有了需要放入自己类别的文件再说？这是一个选择——可以遵循极限编程（XP）的规则 YAGNI（你不会需要它），或者可以尝试使项目未来无忧，并为新来的开发者打下良好的基础。

尝试在这两种方法之间找到良好的平衡——如果怀疑项目可能有一天需要 `extern` 目录，那就添加（版本控制系统可能需要一个空的 `.keep` 文件来将目录签入仓库）。

另一种有效的方法是，通过创建 `README` 文件来指导他人放置他们的外部依赖，该文件概述了推荐的结构。这对于将来将要处理项目的经验不足的开发者有益。可能已经观察到了：开发者不愿意创建目录，尤其是在项目的根目录下。如果提供了一个好的项目结构，其他人就会倾向于遵循它。

有些项目几乎可以在任何环境中构建，而有些项目的要求非常特殊。顶层列表文件是确定适当行动方案的最佳位置。

4.5. 设置环境范围

CMake 提供了多种方式通过 `CMAKE_` 变量、`ENV` 变量和特殊命令来查询环境信息。例如，收集的信息可以用来支持跨平台的脚本。这些机制可以避免使用可能不易移植，或在不同环境中命名不同的特定平台的 `shell` 命令。

对于性能敏感型应用，了解构建平台的所有特性（例如，指令集、CPU 核心数等）将非常有用。然后，将这些信息传递给编译后的二进制文件，以便进行调优。

现在，就来探索 CMake 可提供的本地信息。

4.5.1. 检测操作系统

了解目标操作系统是什么很有用。即使是像文件系统这样的事物，在 Windows 和 Unix 之间也有很大差异，比如：大小写敏感性、文件路径结构、扩展名、权限等。一个系统上存在的命令在另一个系统上可能不可用，可能命名不同（例如，Unix 中的 `ifconfig` 和 Windows 中的 `ipconfig`）或者输出不同的内容。

如果需要用单个 CMake 脚本来支持多个目标操作系统，只需检查 `CMAKE_SYSTEM_NAME` 变量，以便采取相应地行动。

以下是一个简单的例子：

```

1 if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
2     message(STATUS "Doing things the usual way")
3 elseif(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
4     message(STATUS "Thinking differently")
5 elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
6     message(STATUS "I'm supported here too.")
7 elseif(CMAKE_SYSTEM_NAME STREQUAL "AIX")
8     message(STATUS "I buy mainframes.")
9 else()
10    message(STATUS "This is ${CMAKE_SYSTEM_NAME} speaking.")
11 endif()

```

如果需要，有一个包含操作系统版本的变量：CMAKE_SYSTEM_VERSION。然而，我的建议是尽量使解决方案与系统无关，并使用 CMake 内置的跨平台功能。特别是对于文件系统的操作，应该使用附录中描述的 `file()` 命令。

4.5.2. 交叉编译——什么是主机和目标系统？

交叉编译是指在一个机器上编译代码以在另一个目标平台上执行的过程。例如，使用适当的工具集，可以在 Windows 机器上运行 CMake 来编译 Android 应用程序。尽管交叉编译超出了本书的范围，但了解它如何影响 CMake 也很重要。

允许交叉编译的必要步骤是将 CMAKE_SYSTEM_NAME 和 CMAKE_SYSTEM_VERSION 变量设置为目标操作系统的适当值（CMake 文档中将其称为目标系统）。执行构建的操作系统称为主机系统。

无论配置如何，主机系统的信息总是可以在带有 `HOST` 关键字的变量名称中访问：`CMAKE_HOST_SYSTEM`, `CMAKE_HOST_SYSTEM_NAME`, `CMAKE_HOST_SYSTEM_PROCESSOR` 和 `CMAKE_HOST_SYSTEM_VERSION`。

还有一些带有 `HOST` 关键字的变量，其明确地引用主机系统。否则，所有变量都引用目标系统（通常是主机系统，除非在进行交叉编译）。

如果对交叉编译感兴趣，我建议参考 CMake 文档：<https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html>。

4.5.3. 简写变量

CMake 将预定义一些变量，提供关于主机和目标系统的信息。如果使用了特定的系统，相应的变量将设置为非假值（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`

`WIN32` 和 `CMAKE_HOST_WIN32` 变量对于 32 位和 64 位版本的 Windows，以及 MSYS 都将为真（这个值因历史原因而保留）。此外，UNIX 对于 Linux、macOS 和 Cygwin 也将为真。

4.5.4. 主机系统信息

CMake 可以提供更多的变量，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，或者总物理内存是多少。但这些信息不可用——只需要使用以下命令明确地请求：

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

需要提供一个目标变量和我们感兴趣的关键字列表。如果只提供一个关键字，变量将包含一个值；否则，它将是一个值列表。我们可以询问有关环境和操作系统的许多信息：

关键字	描述
HOSTNAME	主机名
FQDN	完全限定域名
TOTAL_VIRTUAL_MEMORY	以 MiB 为单位的虚拟内存总量
AVAILABLE_VIRTUAL_MEMORY	以 MiB 为单位的可用虚拟内存
TOTAL_PHYSICAL_MEMORY	以 MiB 为单位的总物理内存
AVAILABLE_PHYSICAL_MEMORY	以 MiB 为单位的可用物理内存
OS_NAME	如果存在，则输出 <code>uname -s</code> ；无论是 Windows、Linux，还是 Darwin
OS_RELEASE	操作系统子类型，如 Windows Professional
OS_VERSION	操作系统构建 ID
OS_PLATFORM	在 Windows 上和 <code>\$ENV{PROCESSOR_ARCHITECTURE}</code> 的值一样。在 Unix/macOS 上和 <code>uname -m</code> 一样

如果需要，甚至可以查询特定于处理器的信息：

关键字	描述
NUMBER_OF_LOGICAL_CORES	逻辑核数
NUMBER_OF_PHYSICAL_CORES	物理核数
HAS_SERIAL_NUMBER	如果处理器有序列号，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	如果处理器是 64 位的为 1
HAS_FPU	如果处理器有浮点单元为 1
HAS_MMX	如果处理器支持 MMX 指令为 1
HAS_MMX_PLUS	如果处理器支持 Ext. MMX 指令为 1

HAS_SSE	如果处理器支持 SSE 指令为 1
HAS_SSE2	如果处理器支持 SSE2 指令为 1
HAS_SSE_FP	如果处理器支持 SSE FP 指令为 1
HAS_SSE_MMX	如果处理器支持 SSE MMX 指令为 1
HAS_AMD_3DNOW	如果处理器支持 3DNow! 指令为 1
HAS_AMD_3DNOW_PLUS	如果处理器支持 3DNow+ 指令为 1
HAS_IA64	如果 IA64 处理器模拟 x86，则为 1

4.5.5. 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，但它给出了 64 位平台与 32 位平台不同的一个大致概念。

在 C++ 中，不同的架构意味着一些基本数据类型（int 和 long）和指针有不同的位宽。CMake 利用指针大小来收集有关目标机器的信息。这些信息可以通过 CMAKE_SIZEOF_VOID_P 变量获得，对于 64 位，包含值为 8（因为指针是 8 字节宽）；对于 32 位，包含值为 4（4 字节）：

```

1 if(CMAKE_SIZEOF_VOID_P EQUAL 8)
2   message(STATUS "Target is 64 bits")
3 endif()

```

4.5.6. 系统的字节序是什么？

架构可以根据字内字节顺序，或处理器的自然数据单位，分为大端序或小端序。在大端序系统中，最高有效字节存储在最低的内存地址，而最低有效字节存储在最高的内存地址。相反，在小端序系统中，字节顺序反转，最低有效字节存储在最低的内存地址，而最高有效字节存储在最高的内存地址。

大多数情况下，字节序并不重要，但是当编写需要可移植性的位操作代码时，CMake 将提供 BIG_ENDIAN 或 LITTLE_ENDIAN 值，这些值存储在 CMAKE_<LANG>_BYTE_ORDER 变量中，其中 <LANG> 是 C、CXX、OBJC 或 CUDA。

已经了解了如何查询环境，现在让我们将重点转移到项目的主要配置上。

4.6. 配置工具链

对于 CMake 项目，工具链包括用于构建和运行应用程序的所有工具——工作环境、生成器、CMake 可执行文件，以及编译器。

当构建因为一些神秘的编译和语法错误而停止时，一个经验较少的用户会感到多么困惑。他们必须深入源代码，试图理解发生了什么。

经过一个小时的调试，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是否支持所有必需的功能？当然可以！我们有方法指定这些要求。如果工具链不支持所有必需的功能，CMake 将提前停止并显示清晰的错误消息，要求用户介入。

4.6.1. 设置 C++ 标准

我们可能考虑的第一个步骤是，指定编译器应支持的所需 C++ 标准，以构建我们的项目。对于新项目，建议设置 C++14 作为最低标准，但最好是 C++17 或 C++20。从 CMake 3.20 开始，如果编译器支持，可以设置所需的标准为 C++23。此外，从 CMake 3.25 开始，还有一个选项可以将标准设置为 C++26，尽管这目前只是一个占位符。

Note

自 C++11 正式发布以来已经超过 10 年，它不再是现代 C++ 标准。除非目标环境非常老旧，否则不建议以这个版本开始项目

另一个坚持旧标准的原因可能是，如果正在构建难以升级的遗留目标。然而，C++ 委员会非常努力地保持 C++ 向后兼容，通常将标准升级到更高版本都不会有问题。

CMake 支持按目标逐一设置标准（这对于代码库的某些部分确实非常古老的部分很有用），但最好在整个项目中统一标准。这可以通过将 CMAKE_CXX_STANDARD 变量设置为以下值来实现：98、11、14、17、20、23 或 26，例如：

```
1 set(CMAKE_CXX_STANDARD 23)
```

这将作为随后定义的所有目标的默认值（最好将其设置在根列表文件的上方）。如果需要，可以对每个目标进行重写：

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

或者：

```
set_target_properties(<targets> PROPERTIES CXX_STANDARD <version>)
```

第二种方式允许在需要时指定多个目标。

4.6.2. 坚持标准支持

前一部分提到的 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——视为一个偏好。CMake 不知道代码是否用了，以前编译器中不可用的那些新特性。

如果我们确信这将不会成功，可以设置另一个变量（以与前一个相同的方式在每个目标上可重写），明确要求我们针对的标准：

```
1 set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

如果系统中的编译器不支持所需的标准，用户将看到以下消息，并且构建将停止：

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

要求 C++23 可能有点过，即使对于现代环境也是如此。但是 C++20 对于最新的系统来说应该可以，因为自从 2021/2022 年以来，GCC/Clang/MSVC 普遍都支持。

4.6.3. 特定供应商的扩展

根据组织中实施的政策，可能会对允许或禁用特定供应商的扩展感兴趣。这些是什么呢？可以说 C++ 标准对于某些编译器生产商的需求来说进展缓慢，所以他们决定为语言添加他们自己的功能——可以称之为扩展。例如，C++ 技术报告 1 (TR1) 是一个库扩展，引入了正则表达式、智能指针、哈希表和随机数生成器，这些在成为常见功能之前就已经存在。为了支持 GNU 项目发布的此类插件，CMake 将标准编译器标志 (-std=c++14) 替换为 -std=gnu++14。

一方面，其允许一些方便的功能。另一方面，代码将失去可移植性，因为在切换到不同的编译器（或者用户这样做时）可能无法构建！这也是一个按目标设置的属性，其中有一个默认变量，CMAKE_CXX_EXTENSIONS。CMake 在这方面更加宽松，默认允许扩展，除非明确告诉它不要这样做：

```
1 set(CMAKE_CXX_EXTENSIONS OFF)
```

如果可能的话，我建议这样做，因为这将坚持使用与供应商无关的代码。这样的代码不会给用户带来不必要的限制。类似于之前的选项，可以使用 set_property() 在每个目标上更改这个值。

4.6.4. 过程间优化

编译器在单个翻译单元级别优化代码，所以 .cpp 文件将进行预处理、编译，然后优化。在这些操作中生成的中间文件，然后传递给链接器，以创建单个二进制文件。然而，现代编译器在链接时，具有执行跨过程优化的能力，也称为链接时优化。这允许所有编译单元作为一个统一的模块进行优化，原则上会得到更好的结果（有时以构建速度更慢和内存消耗更多为代价）。

如果编译器支持过程间优化，那就使用它。我们将遵循相同的方法，负责此设置的变量称为 CMAKE_INTERPROCEDURAL_OPTIMIZATION。但在设置它之前，需要确保编译器是否支持：

```
1 include(CheckIPOSupported)  
2 check_ipo_supported(RESULT ipo_supported)  
3 set(CMAKE_INTERPROCEDURAL_OPTIMIZATION ${ipo_supported})
```

需要包含一个内置模块才能访问 check_ipo_supported() 命令。如果优化不支持，这段代码将优雅地失败，并回退到默认行为。

4.6.5. 检查支持的编译器特性

正如之前讨论的，如果构建失败，最好是在早期失败，这样就可以向用户提供清晰的反馈消息，并缩短等待时间。有时特别关心哪些 C++ 特性支持（哪些不支持），CMake 将在配置阶段询问编译器，并将可用特性列表存储在 `CMAKE_CXX_COMPILE_FEATURES` 变量中。我们可以编写一个非常具体的检查，并询问是否支持某个特性：

ch04/07-features/CMakeLists.txt

```
1 list(FIND CMAKE_CXX_COMPILE_FEATURES cxx_variable_templates result)
2 if(result EQUAL -1)
3     message(FATAL_ERROR "Variable templates are required for compilation.")
4 endif()
```

为每个我们使用的特性编写一个检查是一项艰巨的任务。甚至 CMake 的作者也建议只检查某些高级元特性是否存在：`cxx_std_98`、`cxx_std_11`、`cxx_std_14`、`cxx_std_17`、`cxx_std_20`、`cxx_std_23` 和 `cxx_std_26`。每个元特性都表示编译器支持特定的 C++ 标准，可以像之前示例中那样使用。

CMake 知道的所有特性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。

4.6.6. 编译测试文件

当我在使用 GCC 4.7.x 编译应用程序时，发生了一个特别有趣的情况。我已经在编译器的参考中手动确认了所有 C++11 特性都得到了支持。然而，解决方案仍然无法正确工作。代码默默地忽略了标准头文件的调用。结果发现，这个特定的编译器存在一个 bug，正则表达式库没有实现。

没有检查可以避免这类罕见错误（而且你不应该需要检查它们！），但可能会想要使用最新标准的一些尖端实验特性，而又不知道哪些编译器支持。可以通过创建一个使用特殊特性的测试文件，来测试项目是否能够工作，这个文件可以快速编译和执行。

CMake 提供了两个配置时间命令，`try_compile()` 和 `try_run()`，以验证目标平台上所需的一切是否得到支持。

`try_run()` 命令会给予更多的自由，因为它可以确保代码不仅编译成功，而且执行也正确（可能想要测试正则表达式是否工作）。当然，这不会在交叉编译场景中工作（因为主机无法运行为不同目标构建的可执行文件），这个检查的目的是向用户提供快速反馈（能正常编译），所以它不是用来运行单元测试或任何复杂的东西——文件尽可能简单：

ch04/08-test_run/main.cpp

```
1 #include <iostream>
2 int main()
3 {
4     std::cout << "Quick check if things work." << std::endl;
```

5 }

调用 `try_run()` 并不复杂。首先设置所需的标准，然后调用 `try_run()`，并将收集的信息输出给用户：

ch04/08-test_run/CMakeLists.txt

```
1 set(CMAKE_CXX_STANDARD 20)
2 set(CMAKE_CXX_STANDARD_REQUIRED ON)
3 set(CMAKE_CXX_EXTENSIONS OFF)
4 try_run(run_result compile_result
5         ${CMAKE_BINARY_DIR}/test_output
6         ${CMAKE_SOURCE_DIR}/main.cpp
7         RUN_OUTPUT_VARIABLE output)
8 message("run_result: ${run_result}")
9 message("compile_result: ${compile_result}")
10 message("output:\n" ${output})
```

这个命令看起来挺吓人，但实际上只需要几个参数就可以编译，并运行一个非常基本的测试文件。我还使用了可选的 `RUN_OUTPUT_VARIABLE` 关键字来收集 `stdout` 的输出。

下一步是扩展基本测试文件，使用实际项目中使用的 C++ 特性——比如通过添加一个变长模板，来查看目标机器上的编译器是否能够处理它。

最后，可以在条件块中检查收集的输出是否符合预期，并在出现问题时打印 `message(SEND_ERROR)`。`SEND_ERROR` 关键字将允许 CMake 继续配置阶段，但将阻止生成构建系统。这在显示所有遇到的错误后才终止构建之前非常有用。我们现在知道如何确保编译可以完整完成。

接下来，让我们转向下一个主题，禁用源内构建。

4.7. 禁用源内构建

第 1 章中，我们讨论了源内构建，以及如何建议总是指定构建路径在源码之外。这不仅允许更清晰的构建树和更简单的 `.gitignore` 文件，而且还能降低意外覆盖或删除源文件的风险。

若要提前停止构建，可以使用以下检查：

ch04/09-in-source/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(NoInSource CXX)
3 if(PROJECT_SOURCE_DIR STREQUAL PROJECT_BINARY_DIR)
4     message(FATAL_ERROR "In-source builds are not allowed")
5 endif()
6 message("Build successful!")
```

如果想要了解更多关于 `STR` 前缀和变量引用的信息，请复习第 2 章。

需要注意的是，无论前面的代码中做了什么，CMake 似乎仍然会创建一个 CMakeFiles/ 目录和一个 CMakeCache.txt 文件。

Note

可能会在线上找到建议使用未记录的变量，来确保用户无论如何都不能在源目录中写入。不建议依赖未记录的变量来限制在源目录中写入，它们可能不会在所有版本中都有效，并且可能会在没有警告的情况下被移除或修改。

如果担心用户将这些文件留在源目录中，将它们添加到 .gitignore (或等效的文件中)，并更改消息，请求手动清理。

4.8. 总结

本章中，包含了为构建健壮且面向未来的项目打下坚实基础的重要概念。讨论了设置最低 CMake 版本和配置项目，如名称、语言和元数据字段。这些可使项目能够有效地扩展。

探讨了项目分区，比较了基本的 `include()` 与 `add_subdirectory` 的使用，后者提供了诸如范围变量管理、简化路径和提高模块化等好处。创建嵌套项目并分别构建它们的能力，在逐渐将代码分解为更独立的单元时。证明了其价值。在理解了分区机制后，又深入研究了如何创建透明、健壮且可扩展的项目结构。检查了 CMake 遍历列表文件和配置步骤的正确顺序，并研究了如何限定目标机器和宿主机器的环境，它们之间的区别是什么，以及可以通过不同的查询获得关于平台和系统的哪些信息。我们还了解了配置工具链，包括指定所需的 C++ 版本，处理特定供应商的编译器扩展，以及启用重要的优化。最后，了解了如何测试编译器所需的功能，并执行示例文件以测试编译支持。

目前为止，了解的技术对于项目至关重要，但它们不足以使项目真正有用。为了增加项目的实用性，需要理解目标的概念。我们之前简要地触及了这个话题，而现在，已经对相关基础知识有了扎实的理解，就可以对其进行全面地探讨。

下一章将介绍的目标，将在进一步增强我们项目的功能性和有效性方面发挥关键作用。

4.9. 扩展阅读

- 关注点分离：

<https://nalexn.github.io/separation-of-concerns/>

- 完整的 CMake 变量参考文档：

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

- `try_compile` 和 `try_run` 文档：

https://cmake.org/cmake/help/latest/command/try_compile.html, https://cmake.org/cmake/help/latest/command/try_run.html

- `CheckIPOSupported` 参考文档：

<https://cmake.org/cmake/help/latest/module/CheckIPOSupported.html>

第 5 章 与目标一起工作

CMake 中，整个应用程序可以从单个源代码文件（如经典的 `helloworld.cpp`）构建。但同样也可以创建一个项目，其中可执行文件是由许多源文件构建的：几十个甚至几千个。许多初学者都是这样做的：用几个文件构建二进制文件，让项目在没有严格计划的情况下自然增长。根据需要不断添加文件，所有东西都已经直接链接到一个没有结构的单一二进制文件中。

作为软件开发者，我们故意划界限，并将组件指定为将一个或多个翻译单元（.cpp 文件）分组的部分，是为了提高代码的可读性，管理耦合和关联性，加快构建过程，并最终发现和提取可重用组件成为自治单元。

每个大型项目都会引入某种形式的划分，这里就是 CMake 目标发挥作用的地方。CMake 目标代表了一个专注于特定目标的逻辑单元。目标可以依赖于其他目标，构建遵循声明方法。CMake 负责确定构建目标的正确顺序，尽可能优化并行构建，并相应地执行必要步骤。作为一个通用原则，当一个目标构建时，会生成一个工件，该工件可以作为其他目标使用，或作为构建过程的最终输出。

注意“工件”这个词，我故意避免使用特定术语，因为 CMake 在生成可执行文件或库之外提供了灵活性。实际上，可以利用生成的构建系统来产生各种类型的输出：额外的源文件、头文件、目标文件、存档、配置文件等。唯一的要求是一个命令行工具（如编译器）、可选的输入文件，以及指定的输出路径。

目标是极其强大的概念，极大地简化了构建项目的过程。理解其如何工作，并掌握以优雅和有组织的方式配置技巧，也至关重要。这些知识确保了顺畅和高效的开发体验。

本章中，包含以下内容：

- 理解目标
- 设置目标的属性
- 编写自定义命令

5.1. 示例下载

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch05>找到本章中出现的代码文件。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保将 `<build tree>` 和 `<source tree>` 占位符替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的位置。

5.2. 理解目标的概念

如果曾经使用过 GNU Make，已经接触过目标（target）这个概念了。本质上，它是一个构建系统遵循的方式，用于将一组文件编译成另一个文件。可以是一个 .cpp 实现文件编译成一个 .o

对象文件，或者是一组 .o 文件打包成一个 .a 静态库。在构建系统中，目标有着众多的组合和可能性。

然而，CMake 在更高的抽象层次上工作，理解大多数语言如何将源文件构建为可执行文件，可以节省时间并跳过定义这些目标的中间步骤。所以，不需要像使用 GNU Make 那样编写显式的命令来编译 C++ 对象文件。一切只是一个 add_executable() 命令，后面跟着可执行目标的名字和源文件列表即可：

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

前面的章节中使用过这个命令，并且已经了解了在实践中，如何使用可执行目标——在生成步骤中，CMake 将创建一个构建系统，并用适当的配方填充它，以编译每个源文件，并将它们链接成一个单一的二进制可执行文件。

CMake 中，可以使用以下三个命令创建目标：

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

构建可执行文件或库之前，CMake 会进行一次检查，以确定生成的输出是否比源文件旧，这种机制帮助 CMake 避免重新创建已经是最新的工件。通过比较时间戳，CMake 有效地识别出哪些目标需要重新构建，从而减少了不必要的编译。

所有定义目标的命令都需要将目标名称作为第一个参数提供，这样就可以在其他命令中引用它，这些命令与目标进行交互，如 target_link_libraries()，target_sources() 或 target_include_directories()，稍后会介绍这些命令。

现在，先来看看可以定义哪些目标。

5.2.1. 定义可执行目标

定义可执行目标的命令 add_executable()（在前面的章节中已经使用过）：

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               [source1] [source2 ...])
```

如果为 Windows 编译，通过添加可选参数 WIN32 关键字，将生成一个不会显示默认控制台窗口（通常在这里看到输出流到 std::cout）的可执行文件。相反，应用程序将生成自己的 GUI。

下一个可选参数 MACOSX_BUNDLE 在某种程度上类似；它使得为 macOS/iOS 生成的应用程序可以在 Finder 中打开，作为 GUI 应用程序启动。

当使用 EXCLUDE_FROM_ALL 关键字时，将防止在常规默认构建中构建可执行目标。这样的目标必须在构建命令中明确提及：

```
cmake --build -t <target>
```

最后，需要提供编译目标的源文件列表。支持以下的扩展名：

- 对于 C 语言: c, m
- 对于 C++ 语言: C, M, c++, cc, cpp, cxx, m, mm, mpp, CPP, ixx, cppm, ccm, cxxm, c++m

我们没有将头文件添加到源文件列表中。这可以通过提供包含这些文件的目录路径和 `target_include_directories()` 命令隐式完成，或者使用 `target_sources()` 命令的 `FILE_SET` 功能（在 CMake 3.23 中添加）。这对于可执行文件来说是一个重要的话题，但由于它与目标正交且复杂，我们将在第 7 章再详细讨论。

5.2.2. 定义库目标

定义库与定义可执行文件非常相似，但不需要定义 GUI 方面的关键字。以下是该命令的签名：

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [<source>...])
```

关于名称、排除所有和源的规则与可执行目标完全一致。唯一的区别在于 `STATIC`、`SHARED` 和 `MODULE` 关键字。如果有使用库的经验，就知道这些定义了 CMake 将生成哪种工件：静态链接库、共享（动态库）或模块。这又是一个相当广泛的主题，我们将在第 8 章中深入讨论。

5.2.3. 自定义目标

自定义目标与可执行文件或库略有不同，通过执行明确给出的命令行来扩展构建功能，可用于：

- 计算其他二进制文件的校验和。
- 运行代码消毒器并收集结果。
- 将编译报告发送到指标通道。

从这张列表中可以猜到，自定义目标只在相当高级的项目中有用，所以我们只了解基础知识，然后继续讨论更重要的主题。

要定义一个自定义目标，请使用以下语法（为了简洁，省略了一些选项）：

```
add_custom_target(Name [ALL] [COMMAND command2 [args2...] ...])
```

自定义目标有一些需要考虑的缺点。由于涉及 `shell` 命令，可能是系统特定的，会限制可移植性。此外，自定义目标可能没有为 CMake 提供直接的方式，来确定生成的具体工件或副产品（如果有的话）。

自定义目标也不像可执行文件和库会进行陈旧性检查（不验证源文件是否比二进制文件更新），因为默认情况下它们不会添加到依赖关系图中（所以 `ALL` 关键字与 `EXCLUDE_FROM_ALL` 相反）。

来看看这个依赖关系图什么。

5.2.4. 依赖关系图

成熟的应用程序通常由许多组件构建而成，特别是内部库。从结构的角度来看，划分项目是有用的。当相关的事物包装在一个逻辑实体中时，就可以与其他目标链接：另一个库或一个可执行文件。这对于多个目标使用同一个库的情况尤其方便。图 5.1 描述了一个示例依赖关系图：

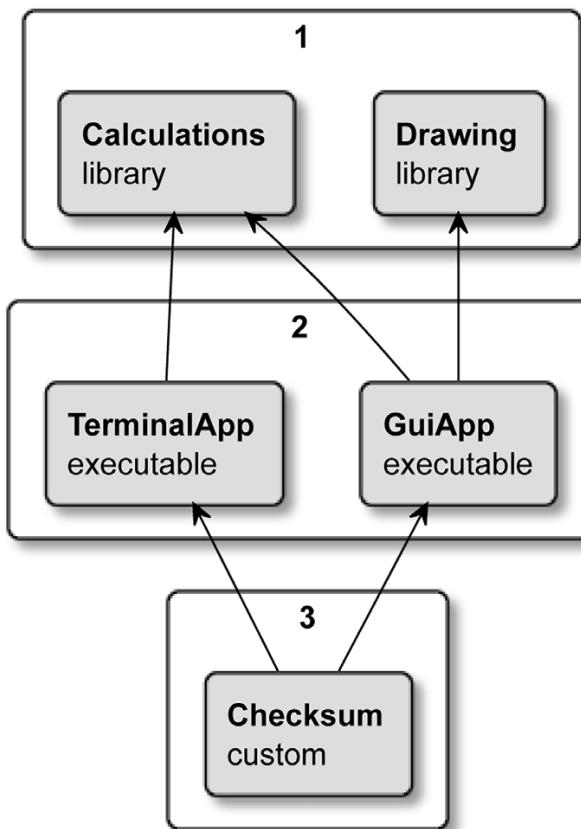


图 5.1: BankApp 项目中构建依赖的顺序

这个项目有两个库，两个可执行文件和一个自定义目标。用例是提供一个带有 GUI 的银行应用程序 (GuiApp)，以及一个作为自动化脚本一部分使用的命令行版本 (TerminalApp)。两个可执行文件都依赖于相同的 Calculations 库，但只有一个需要 Drawing 库。为了确保应用程序二进制文件的完整性，我们还将计算一个校验和，并通过单独的安全渠道分发它。CMake 在编写此类解决方案的列表文件时非常灵活：

ch05/01-targets/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(BankApp CXX)
3
4 add_executable(terminal_app terminal_app.cpp)
5 add_executable(gui_app gui_app.cpp)
6 target_link_libraries(terminal_app calculations)
7 target_link_libraries(gui_app calculations drawing)
8
9 add_library(calculations calculations.cpp)
```

```
10 add_library(drawing drawing.cpp)
11
12 add_custom_target(checksum ALL
13   COMMAND sh -c "cksum terminal_app>terminal.ck"
14   COMMAND sh -c "cksum gui_app>gui.ck"
15   BYPRODUCTS terminal.ck gui.ck
16   COMMENT "Checking the sums..."
17 )
```

通过使用 `target_link_libraries()` 命令将库与可执行文件链接。若不连接，会因未定义的符号，构建可执行文件失败。这里，发现在声明库之前就调用了这个命令吗？当 CMake 配置项目时，会收集关于目标和属性信息——名称、依赖关系、源文件和其他信息。

解析了所有文件之后，CMake 将尝试构建一个依赖关系图。像所有有效的依赖关系图一样，它们是有向无环图（DAG）。所以有一个清晰的哪个目标依赖于哪个目标，并且这样的依赖关系不会形成循环。

当在构建模式下执行 `cmake` 时，生成的构建系统将检查定义的顶层目标，并递归地构建依赖关系。来看一下啊图 5.1 中的例子：

1. 从顶层开始，构建组 1 中的两个库。
2. 当 Calculations 和 Drawing 库完成后，构建组 2 -GuiApp 和 TerminalApp。
3. 构建校验和目标；运行指定的命令行以生成校验和（`cksum` 是一个 Unix 校验和工具，这个例子不会在其他平台上构建）。

然而，这里有一个小问题——前面的解决方案不能保证校验和目标会在可执行文件之后构建。CMake 不知道校验和依赖于可执行二进制文件的存在，所以它可以自由地首先开始构建它。为了解决这个问题，我们可以在文件末尾放置 `add_dependencies()` 命令：

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

这将确保 CMake 理解校验和目标与可执行文件之间的关系。

那很好，但 `target_link_libraries()` 和 `add_dependencies()` 之间有什么区别呢？`target_link_libraries()` 旨在与实际库一起使用，并允许你控制属性传播。第二个命令仅用于顶级目标，以设置它们的构建顺序。

随着项目复杂性的增加，依赖树变得难以理解。要如何简化这个过程呢？

5.2.5. 可视化依赖关系

即使是小项目，也可能难以推理并与其它开发者分享。一个整洁的图表将大有帮助。毕竟，一图胜千言。我们可以自己动手绘制图表，就像图 5.1 那样。但这很繁琐，并且每次项目更改时都需要更新。幸运的是，CMake 有一个很棒的模块，可以生成 `dot/graphviz` 格式的依赖关系图，支持内部和外部依赖！

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，可以将其导入到 Graphviz 可视化软件中，该软件可以渲染图像或生成 PDF 或 SVG 文件，可以作为软件文档的一部分存储。每个人都喜欢优秀的文档，但几乎没有人喜欢创建它——现在，不需要了！

自定义目标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，就可以自定义图表了。使用 set(GRAPHVIZ_CUSTOM_TARGETS TRUE) 命令在图表中启用自定义目标：

ch05/01-targets/CMakeGraphVizOptions.cmake

```
1 set(GRAPHVIZ_CUSTOM_TARGETS TRUE)
```

其他选项允许添加图表名称、标题、节点前缀，并配置应包含或排除在输出中的目标（按名称或类型）。访问官方 CMake 文档以获取 CMakeGraphVizOptions 模块的完整描述。

甚至可以直接在浏览器中运行 Graphviz，地址是：<https://dreampuf.github.io/GraphvizOnline/>。

所需要做的就是将 test.dot 文件的内容复制粘贴到左边的窗口中，项目就会可视化（如图 5.2）。非常方便，不是吗？

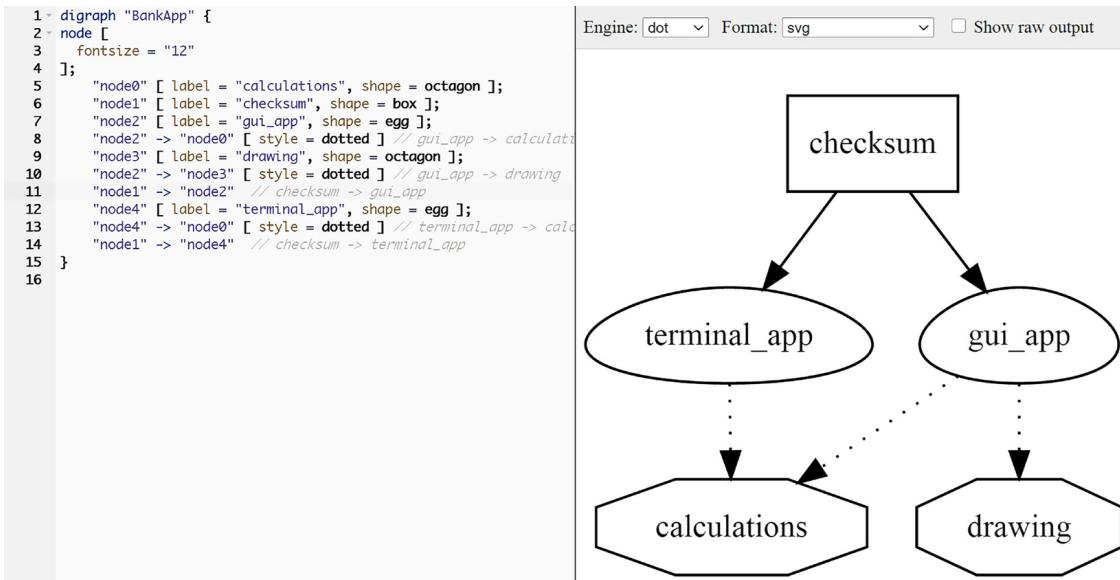


图 5.2：在 Graphviz 中可视化 BankApp 示例

使用这种方法，可以快速看到所有明确定义的目标。

现在理解了目标的概念，我们知道如何定义不同类型的目标，包括可执行文件、库和自定义目标，以及如何创建依赖关系图并打印它。让我们使用这些信息进行更深入的研究，看看如何配置它们。

5.2.6. 设置目标的属性

目标具有类似于 C++ 对象字段的属性。其中一些属性为了修改而设计的，而有些是只读的。CMake 定义了大量“已知属性”（请参阅扩展阅读部分），这些属性取决于目标的类型（可执行文件、库或自定义），也可以添加自己的属性。使用以下命令来操作目标的属性：

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

要在屏幕上输出目标属性，首先需要将其存储在 `<var>` 变量中，然后将其消息传递给用户。读取属性必须逐个进行；设置目标属性允许我们同时为多个目标指定多个属性。

Note

属性的概念不仅适用于目标；CMake 支持为其他范围设置属性：GLOBAL、DIRECTORY、SOURCE、INSTALL、TEST 和 CACHE。为了操作所有类型的属性，有通用的 `get_property()` 和 `set_property()` 命令。在某些项目中，会看到这些低层命令用来精确地完成 `set_target_properties()` 命令所做的事情：

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

通常，尽可能多地使用高级命令是更好的选择。有时，CMake 提供了简写命令。例如，`add_dependencies()` 是向 `MANUALLY_ADDED_DEPENDENCIES` 目标属性追加依赖关系的简写。这种情况下，可以使用 `get_target_property()` 精确地查询，就像其他属性一样。然而，不能使用 `set_target_properties()` 来更改（是只读的），因为 CMake 坚持使用 `add_dependencies()` 命令来限制操作仅为追加。

在接下来的章节中讨论编译和链接时，我们将介绍更多设置属性的命令。

现在，让我们专注于一个目标的属性如何传递给另一个目标。

传递目标的使用要求

命名是困难的，有时会得到一个难以理解的标签。在线 CMake 文档中，会遇到“传递使用要求”不幸的是这样一个令人费解的标题。让我们解开这个奇怪的名字，并提供一个更容易理解的术语。

从中间的术语开始：使用。正如我们之前讨论的，一个目标可能依赖于另一个。CMake 文档有时将这种依赖称为“使用”，即一个目标使用另一个目标。

在某些情况下，使用的目标会为自己设置特定的属性或依赖项，这反过来构成了使用它的其他目标的依赖：链接某些库、包含目录或要求特定的编译器特性。

解谜的最后一部分，“传递”这个词描述的行为正确（也许可以更简单一些）。CMake 将一些使用目标的属性/要求添加到使用目标的属性中。

所以，一些属性可以隐式地在目标之间传递（或简单地传播），因此更容易表达依赖关系。

简化这个整个概念，就像源目标（被使用的目标）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，来理解其为什么存在，以及是如何工作的：

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

这个目标命令将填充一个 `<source>` 目标的 `COMPILE_DEFINITIONS` 属性。编译定义就是传递给编译器的 `-Dname=definition` 标志，用于配置 C++ 预处理器定义（将在第 7 章中讨论）。这里有趣的部分是第二个参数，需要指定三个值中的一个，`INTERFACE`、`PUBLIC` 或 `PRIVATE`，以控制属性应该传递给哪个目标。现在，不要将这些与 C++ 访问修饰符混淆——这是一个全新的概念。

传播关键字的工作方式如下：

- `PRIVATE` 设置源目标的属性。
- `INTERFACE` 设置使用目标的目标属性。
- `PUBLIC` 设置源目标和使用目标的目标属性。

当属性不应传递给其他目标时，将其设置为 `PRIVATE`。当需要这样的传递时，选择 `PUBLIC`。如果处于一个源目标在其实现（.cpp 文件）中就不使用该属性，而在头文件中使用，并且这些属性传递给使用目标的目标，则应使用 `INTERFACE` 关键字。

这是如何工作的？为了管理这些属性，CMake 提供了一些命令，例如之前提到的 `target_compile_definitions()`。当指定 `PRIVATE` 或 `PUBLIC` 关键字时，CMake 将在目标的属性中存储提供的值，`COMPILE_DEFINITIONS`。此外，关键字是 `INTERFACE` 或 `PUBLIC`，将在具有 `INTERFACE_` 前缀的属性中存储值——`INTERFACE_COMPILE_DEFINITIONS`。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。就这样传播属性，或 CMake 所说的传递目标的使用要求。

使用 `set_target_properties()` 命令管理的属性可以在<https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html>找到，在目标属性部分（并非所有目标属性都可传递）。这里是最重要的几个：

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

我们将在接下来的页面中讨论大多数这些选项，但所有这些选项当然都在 CMake 手册中有详细描述。在以下链接中找到它们的详细描述：https://cmake.org/cmake/help/latest/prop_tgt.html

接下来的问题是，如何远传播这些属性。属性只设置在第一个目标上，还是发送到依赖图的最顶层？这可以决定。

为了在目标之间创建依赖关系，使用 `target_link_libraries()` 命令。这个命令需要一个传播关键字：

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

这个签名也指定了传播关键字，控制源目标的属性如何在目标目标中存储。图 5.3 展示了生成阶段（配置阶段完成后）传播属性会发生什么情况：

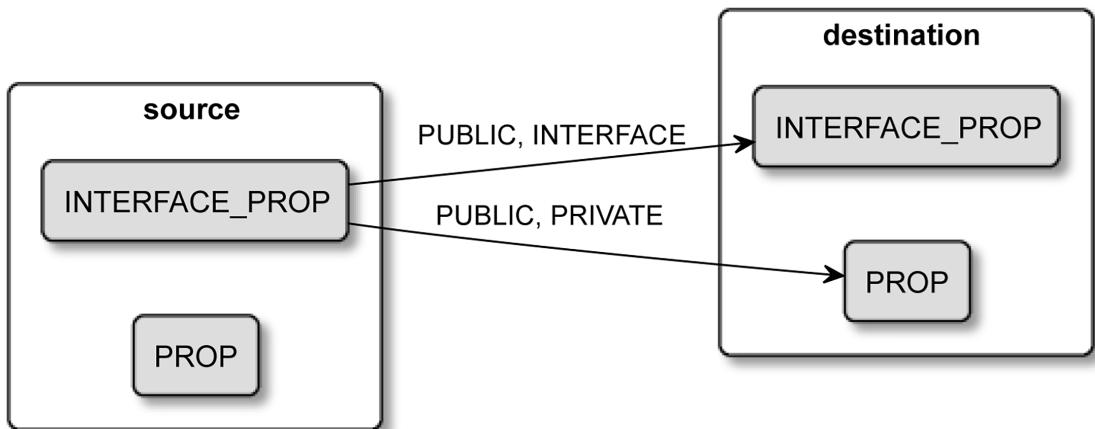


图 5.3：属性是如何在目标间进行传递的

传播关键字的工作方式：

- PRIVATE 将源值添加到源目标的私有属性。
- INTERFACE 将源值添加到源目标的接口属性。
- PUBLIC 将值添加到源目标的两个属性。

INTERFACE 属性仅用于将属性进一步传播到链中的下一个目标，而源目标在其构建过程中不会使用。

我们之前使用的基本 `target_link_libraries(...)` 命令隐式指定了 PUBLIC 关键字。

如果正确地为源目标设置了传播关键字，属性将自动放置在目标目标上——除非有冲突…

处理冲突的传播属性

当一个目标依赖于多个其他目标时，可能存在传播属性之间直接冲突的情况。例如，一个使用的目標将 `POSITION_INDEPENDENT_CODE` 属性设置为 `true`，而另一个设置为 `false`。CMake 将这种冲突理解为错误，并打印出类似于这样的错误信息：

```
CMake Error: The INTERFACE_POSITION_INDEPENDENT_CODE property of "source_target" does not
→ agree with the value of POSITION_INDEPENDENT_CODE already determined for
→ "destination_target".
```

开发者需要明确知道这种冲突，并且需要去解决它。CMake 有一些自己的属性，这些属性必须在源目标和目标目标之间“一致”。

有时，这可能很重要——例如，在多个目标中使用相同的库，然后将它们链接到一个可执行文件。如果这些源目标使用的是同一库的不同版本，可能会遇到问题。

为了确保只使用特定版本的库，可以创建一个自定义接口属性，INTERFACE_LIB_VERSION，并在其中存储版本。这还不够解决问题，因为 CMake 默认不会传播自定义属性（这个机制只适用于内置目标属性），必须明确地将自定义属性添加到“兼容”属性列表中。

每个目标都有四个这样的列表：

- COMPATIBLE_INTERFACE_BOOL
- COMPATIBLE_INTERFACE_STRING
- COMPATIBLE_INTERFACE_NUMBER_MAX
- COMPATIBLE_INTERFACE_NUMBER_MIN

将属性添加到它们中的任何一个，都会触发传播和兼容性检查。BOOL 列表将检查所有传递到目标目标的属性是否评估为相同的布尔值。类似地，STRING 将评估为字符串。NUMBER_MAX 和 NUMBER_MIN 略有不同——传递的值不必匹配，但目标目标将只接收最高或最低值。

这个例子将帮助我们了解，如何在实践中对其进行应用：

ch05/02-propagated/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(PropagatedProperties CXX)
3
4 add_library(source1 empty.cpp)
5 set_property(TARGET source1 PROPERTY INTERFACE_LIB_VERSION 4)
6 set_property(TARGET source1 APPEND PROPERTY
7             COMPATIBLE_INTERFACE_STRING LIB_VERSION)
8
9 add_library(source2 empty.cpp)
10 set_property(TARGET source2 PROPERTY INTERFACE_LIB_VERSION 4)
11
12 add_library(destination empty.cpp)
13
14 target_link_libraries(destination source1 source2)
```

这里，创建了三个目标；为了简化，所有目标都使用相同的空源文件。在两个源目标上，指定了带有 INTERFACE_前缀的自定义属性，并将其设置为相同的匹配库版本。两个源目标都链接到相应的目标。最后，我们在 source1 上指定了字符串兼容性，要求其作为属性（这里没有添加 INTERFACE_前缀）。

CMake 将这个自定义属性传播到相应目标，并检查所有源目标的版本是否完全匹配（兼容性属性只需在目标目标上设置一次）。

已经了解了常规目标是什么，那就来看看其他像目标、闻起来像目标，有时甚至表现得像目标的东西，但最终发现它们并不是真正的目标。

5.2.7. 识别伪目标

目标的概念非常有用，以至于能够将其某些行为借用到其他事物上；这些事物不是构建系统的输出，而是输入——外部依赖项、别名等。这些就是伪目标，或者是不出现在生成的构建系统中的目标：

- 导入的目标
- 别名目标
- 接口库

导入的目标

如果浏览了本书的目录，会了解将讨论 CMake 如何管理外部依赖项——其他项目、库等。IMPORTED 目标是这个过程的产物。CMake 可以定义它们，作为 `find_package()` 命令的结果。

可以调整这类目标的属性：编译定义、编译选项、包含目录等——甚至支持目标传递的使用要求。然而，应该将它们视为不可变的目标；不要更改它们的源文件或依赖关系。

IMPORTED 目标的定义范围可以是全局的，也可以是定义它们的目录的本地范围（在子目录中可见，但在父目录中不可见）。

别名目标

别名目标的确切作用就是你所期望的——为目标创建另一个不同的名称引用。可以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，不能安装或导出别名（在生成的构建系统中不可见）。

为什么还要有别名呢？有时，它们非常有用，比如项目的一部分（如子目录）需要以特定名称引用一个目标，而实际的实现可能因情况而异。例如，希望根据用户的选择构建解决方案中的库或导入它。

接口库

这是一个有趣的构造——一个不编译东西的库，而是作为一个实用目标。整个概念都围绕传播属性（传递使用要求）构建。

接口库主要有两个用途——代表仅包含头文件的库，以及将一堆传播属性打包成一个逻辑单元。

仅包含头文件的库可以用 `add_library(INTERFACE)` 轻松创建：

```
1 add_library(Eigen INTERFACE
2     src/eigen.h src/vector.h src/matrix.h
```

```
3 )
4 target_include_directories(Eigen INTERFACE
5   ${BUILD_INTERFACE}:${CMAKE_CURRENT_SOURCE_DIR}/src>
6   ${INSTALL_INTERFACE:include/Eigen}
7 )
```

在前面的代码片段中，创建了一个包含三个头的 Eigen 接口库。接下来，使用生成器表达式（这些以美元符号和尖括号表示，\$<…>，将在下一章解释），将包含目录设置为当目标导出时为 \${CMAKE_CURRENT_SOURCE_DIR}/src，安装时为 include/Eigen（这将在本章的末尾解释）。

要使用这样的库，只需链接它即可：

```
1 target_link_libraries(executable Eigen)
```

这里不会发生实际的链接，但 CMake 会将这个命令理解为，向可执行目标传播所有 INTERFACE 属性的请求。

第二种使用场景利用了相同的机制，但出于不同的目的——创建了一个逻辑目标，可以作为一个传播属性的占位符。然后，可以将这个目标用作其他目标的依赖，并以干净、方便的方式设置属性。这是一个例子：

```
1 add_library(warning_properties INTERFACE)
2 target_compile_options(warning_properties INTERFACE
3   -Wall -Wextra -Wpedantic
4 )
5 target_link_libraries(executable warning_properties)
```

add_library(INTERFACE) 命令创建了一个逻辑的 warning_properties 目标，用于在第二个命令中为可执行目标设置编译选项。我建议使用这些 INTERFACE 目标，它们可以提高代码的可读性和可重用性，这是将一堆魔法值重构为命名良好的变量的过程。我还建议明确地为接口库添加一个后缀，如 _properties，以便轻松区分接口库和常规库。

5.2.8. 对象库

对象库用于将多个源文件，组合成一个单一的逻辑目标，并在构建过程中将它们编译成 (.o) 对象文件。要创建一个对象库，遵循与创建其他库相同的方法，但使用 OBJECT 关键字：

```
add_library(<target> OBJECT <sources>)
```

构建过程中产生的对象文件可以作为其他目标的编译元素，使用 \${TARGET_OBJECTS:objlib} 生成器表达式：

```
add_library(... ${TARGET_OBJECTS:objlib} ...)
add_executable(... ${TARGET_OBJECTS:objlib} ...)
```

或者，可以使用 `target_link_libraries()` 命令将它们作为依赖项进行添加。

在 `Calc` 库的上下文中，对象库将非常有用，以避免为库的静态和共享版本编译库源的冗余。对于共享库，明确地编译具有 `POSITION_INDEPENDENT_CODE` 启用的对象文件是必要的。

回到项目的目标：`calc_obj` 将提供编译后的对象文件，然后将用于 `calc_static` 和 `calc_shared` 库。让我们探索这两种类型库之间的实际区别，并理解为什么可能需要创建两者。

伪目标是否穷尽了目标的概念？当然不是！我们仍然需要了解，这些目标是如何用于生成构建系统的。

5.2.9. 构建目标

项目的上下文和生成的构建系统中，“目标”一词可以有不同的含义。在生成构建系统的上下文中，CMake 将 CMake 语言编写的列表文件“编译”成所选构建工具的语言，例如为 GNU Make 创建 `Makefile`。这些生成的 `Makefile` 有自己的目标集。其中一些目标是列表文件中定义的目标的直接转换，而其他目标则在构建系统生成过程中隐式创建。

这样一个构建系统目标就是 `ALL`，CMake 默认生成为包含所有顶级列表文件目标的目标，如可执行文件和库（不一定是自定义目标）。当运行 `cmake --build <build tree>` 时，`ALL` 将构建所有顶级列表文件目标。第一章中，可以通过添加`--target <name>` 参数到 `cmake` 构建命令中来选择一个目标。

一些可执行文件或库可能不需要在每次构建中都存在，但我们希望将它们作为项目的一部分，以便在不常需要的场合中使用。为了优化默认构建，可以将它们从 `ALL` 目标中排除：

```
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，它们排除在 `ALL` 目标之外，除非明确地使用 `ALL` 关键字添加它们，就像我们在 `BankApp` 示例中所做的那样。

另一个隐式定义的构建目标是 `clean`，简单地从构建树中删除产生的工件。我们使用它来清除所有旧文件并从头开始构建。重要的是要理解，`clean` 并不会简单地删除构建目录中的所有内容。为了正确工作，需要手动指定自定义目标可能创建的文件作为 `BYPRODUCTS`（请参阅 `BankApp` 示例）。

我们对目标和它们不同方面的探索之旅结束了！了解了如何创建它们，配置它们的属性，使用伪目标，并决定它们是否应该默认构建。还有一个有趣的非目标机制，可以创建在所有实际目标中使用的自定义工件——自定义命令（不要与自定义目标混淆）。

5.3. 编写自定义命令

使用自定义目标有一个缺点——将它们添加到 `ALL` 目标或者依赖它们来构建其他目标，每次都会构建。有时候，这正是想要的，但需要自定义行为来生成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++

- 在另一个目标构建之前或之后，立即执行自定义操作

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，必须添加到依赖关系图中。有两种方法可以实现——将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。

5.3.1. 将自定义命令用作生成器

诚然，并非每个项目都需要从其他文件生成 C++ 代码。这样的情况可能是编译 Google 的 Protocol Buffer (Protobuf) 的`.proto` 文件。如果不熟悉这个库，Protobuf 是一个用于结构化数据的平台无关的二进制序列化器。

可以用来将对象编码和解码为二进制流：文件或网络连接。为了保持 Protobuf 跨平台并且在同时保持快速，Google 的工程师们发明了自己的 Protobuf 语言，在`.proto` 文件中定义模型，如下所示：

```
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
}
```

这样的文件可以用来在多种语言中编码数据——C++、Ruby、Go、Python、Java 等等。Google 提供了编译器——`protoc`，读取`.proto` 文件并为所选语言输出有效的结构和序列化源代码（稍后需要编译或解释）。聪明的工程师不会将这些生成的源文件检入仓库，而是使用原始的 Protobuf 格式并在构建链中添加一个步骤来生成源文件。

我们尚不知道如何检测目标主机上是否（以及在哪里）可用 Protobuf 编译器（我们将在第 9 章中介绍）。现在，假设编译器的 `protoc` 命令位于系统已知的某个位置，已经准备好了

person.proto 文件，并且知道 Protobuf 编译器将输出 person.pb.h 和 person.pb.cc 文件。以下是如何定义一个自定义命令来编译它们的例子：

```
1 add_custom_command(OUTPUT person.pb.h person.pb.cc  
2     COMMAND protoc ARGS person.proto  
3     DEPENDS person.proto  
4 )
```

然后，为了允许可执行文件进行序列化，可以简单地将输出文件添加到源文件中：

```
1 add_executable(serializer serializer.cpp person.pb.cc)
```

假设正确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，一切都会自动编译和更新。

一个简化的（实用性要小得多）示例是通过从另一个位置复制来创建必要的头文件：

ch05/03-command/CMakeLists.txt

```
1 add_executable(main main.cpp constants.h)  
2 target_include_directories(main PRIVATE ${CMAKE_BINARY_DIR})  
3 add_custom_command(OUTPUT constants.h COMMAND cp  
4     ARGS "${CMAKE_SOURCE_DIR}/template.xyz" constants.h)
```

这时，“编译器”是 cp 命令。它通过从源树中复制到构建树根目录，创建一个 constants.h 文件，从而满足 main 目标的依赖。

5.3.2. 将自定义命令用作目标钩子

add_custom_command() 命令的第二个版本引入了一种机制，用于在构建目标之前或之后执行命令：

```
1 add_custom_command(TARGET <target>  
2     PRE_BUILD | PRE_LINK | POST_BUILD  
3     COMMAND command1 [ARGS] [args1...]  
4     [COMMAND command2 [ARGS] [args2...] ...]  
5     [BYPRODUCTS [files...]]  
6     [WORKING_DIRECTORY dir]  
7     [COMMENT comment]  
8     [VERBATIM] [USES_TERMINAL]  
9     [COMMAND_EXPAND_LISTS])
```

通过第一个参数指定想要用新行为“增强”的目标，并在满足以下条件的情况下执行：

- PRE_BUILD 将在此目标的所有其他规则之前运行（仅限 Visual Studio 生成器；对于其他生成器，行为类似于 PRE_LINK）。

- PRE_LINK 将命令绑定在所有源代码编译完成后，但在链接（或归档）目标之前运行。不适用于自定义目标。
- POST_BUILD 将在此目标的所有其他规则执行完毕后运行。

使用这个版本的 add_custom_command()，可以复现之前 BankApp 示例中的校验和生成：

ch05/04-command/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.26)
2 project(Command CXX)
3 add_executable(main main.cpp)
4 add_custom_command(TARGET main POST_BUILD
5                     COMMAND cksum
6                     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。但是第一个参数中发生了什么？它不是一个变量，如果是变量，它会被大括号 (\${}) 包裹，而不是尖括号 (\$<>)。它是一个生成器表达式，计算结果为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，我们将在下一章中详细解释。

5.4. 总结

理解目标，是编写清晰、现代的 CMake 项目的关键。本章中，不仅讨论了什么是目标，以及如何定义三种不同类型的目标：可执行文件、库和自定义目标。还解释了目标如何通过依赖关系图相互依赖，并了解了如何使用 Graphviz 模块可视化。有了这种基本的理解，就能够了解目标的关键特性——属性。我们不仅介绍了一些在目标上设置常规属性的命令，还解决了传播属性，也称为“目标传递的使用要求”谜团。

这是一个难以攻克的问题，因为需要理解的不仅是如何控制传播哪些属性，还有这种传播如何影响后续的目标。此外，还发现了如何确保从多源属性的兼容性。

然后，简要讨论了伪目标：导入的目标、别名目标和接口库。所有这些在以后的项目中都会派上用场，特别是当了解如何将它们与传播属性连接起来。接着，讨论了生成的构建目标，以及对配置阶段的影响。之后，花了一些时间研究一种与目标相似，但又不完全是目标的机制：自定义命令。提到了如何生成其他目标（编译、翻译等）使用的文件，以及其钩子功能：在构建目标时执行的步骤。

有了如此坚实的基础，就可以进行下一个主题了——将 C++ 源代码编译成可执行文件和库。

5.5. 扩展阅读

- Graphviz 模块的文档：

<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/cmake-properties.7.html#properties-on-targets>

- 传递目标的使用要求:

<https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#target-usage-requirements>

第 6 章 使用生成器表达式

许多 CMake 用户在探索中并未遇到生成器表达式，这个概念相当高级。然而，对于准备进入普遍可用阶段，或首次向更广泛受众发布的项目来说，在导出、安装和打包方面起着重要的作用。如果只是想快速学习 CMake 的基础知识并专注于 C++ 方面，可以暂时跳过本章，以后再回来阅读。另一方面，我们现在讨论生成器表达式，因为接下来的章节在解释 CMake 更深入的内容时，会引用这些知识。

我们将从介绍生成器表达式的主题开始：它们是什么，有什么用途，以及如何形成和扩展的。接下来将简要介绍嵌套机制，并对条件扩展进行更详细的描述，可以使用布尔逻辑、比较操作和查询。当然，会将深入探讨表达式的内容。

首先，研究字符串、列表和路径的转换，专注于主要内容之前，了解基础知识很有必要。最终，生成器表达式在实际应用中用来获取在构建后期阶段可用的信息，并在适当的上下文中呈现。确定这个上下文有很大的价值，将了解如何根据用户选择的构建配置、当前平台和当前工具链，来参数化构建过程。也就是说，要确定正在使用的编译器、其版本，以及具有的功能。不仅如此，还将找出如何查询构建目标及其相关信息的属性。

为了确保能够充分理解生成器表达式，我在本章的最后部分包含了一些有趣的使用示例。还有一个关于如何查看生成器表达式输出的快速解释，这有点棘手。不过别担心，生成器表达式并不像看起来那么复杂。

本章中，将包含以下内容：

- 生成器表达式是什么？
- 通用表达式语法的基本规则
- 条件扩展
- 查询和转换
- 尝试示例

6.1. 示例下载

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch06>找到本章中出现的代码文件。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保将 `<build tree>` 和 `<source tree>` 占位符替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的位置路径。

6.2. 什么是生成器表达式？

CMake 在三个阶段构建解决方案：配置、生成和运行构建工具。通常，配置阶段所需数据都可以找到。然而，有时会遇到类似于“先有鸡还是先有蛋”的悖论。以第 5 章中的“将自定义命令作为目标钩子”的例子来说，一个目标需要知道另一个目标的二进制工件的路径。不幸的是，这个信息只有在所有列表文件解析且配置阶段完成后才能获得。

那如何解决这类问题呢？一个解决方案是为这个信息创建一个占位符，并将其计算推迟到下一个阶段——生成阶段。

这正是生成器表达式(也称为“genexes”)所做的。其围绕目标属性构建，如 LINK_LIBRARIES、INCLUDE_DIRECTORIES、COMPILE_DEFINITIONS，以及传播属性（尽管不是全部），其遵循类似于条件语句和变量计算的规则。

Note

生成器表达式将在生成阶段进行计算（配置完成且构建系统创建后），所以将它们的输出捕获到变量，并输出到控制台并不是直接的操作。

生成器表达式的数量众多，在某种程度上构成了自己的、特定领域的语言——这种语言支持条件表达式、逻辑操作、比较、转换、查询和排序。利用生成器表达式可以操作和查询字符串、列表、版本号、shell 路径、配置和构建目标。本章中，将简要概述这些概念，由于在大多数情况下不是必需的，所以将专注于基础知识。主要关注点将是生成器表达式的应用，即从目标的生成配置和构建环境状态中收集信息。为了完整参考，最好在线阅读官方的 CMake 手册（请参见“扩展阅读”部分获取 URL）。

通过例子解释都会更清楚，所以让我们直接进入正题，描述生成器表达式的语法。

6.3. 学习通用表达式语法的基本规则

要使用生成器表达式，需要将其添加到支持生成器表达式计算的 CMake 命令中。大多数特定于目标的命令都支持，还有许多其他命令（查看特定命令的官方文档以了解更多信息）。

一个经常与生成器表达式一起使用的命令是 target_compile_definitions()。要使用生成器表达式，需要将其作为命令参数提供，如下所示：

```
1 target_compile_definitions(foo PUBLIC BAR=$<TARGET_FILE:baz>)
```

这个命令向编译器的参数中添加了一个-D 定义标志（先忽略 PUBLIC），将 BAR 预处理器定义设置为 foo 目标产生的二进制工件的路径（生成器表达式以当前形式存储在变量中）。这种扩展推迟到了生成阶段，那时许多事情都已经配置并已知。

生成器表达式是什么样的呢？

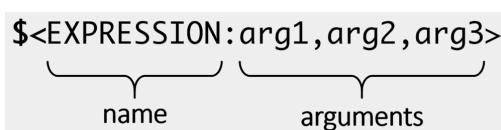


图 6.1：生成器表达式的语法

如图 6.1 所示，结构看起来相当简单：

- 以美元符号和左括号 (\$<) 开始。
- 添加 EXPRESSION 名称。
- 如果表达式需要参数，添加冒号 (:) 并提供 arg1, arg2 …argN 值，用逗号 (,) 分隔。
- 以大于号 (>) 结束表达式。

有些表达式不需要参数，例如 \$<PLATFORM_ID>。

除非明确指出，否则表达式通常是在使用表达式的目标上下文中进行计算的。这种关联是从使用表达式的命令中推断出的。前面的例子中，了解了 target_compile_definitions() 是如何提供 foo 作为其操作的目标。因此，在该命令中使用的特定于目标的生成器表达式将隐式地使用 foo。注意，示例中使用的生成器表达式 \$<TARGET_FILE> 需要目标属性作为其操作的上下文。还有一些生成器表达式不接受目标作为参数（如 \$<COMPILE_LANGUAGE>），将隐式地使用封闭命令的目标。这些将在后面详细讨论。

当使用生成器表达式的更高级功能时，可能会很快变得非常混乱和复杂，事先了解其细节信息非常重要。

6.3.1. 嵌套

将生成器表达式作为参数，传递给另一个生成器表达式的能力开始介绍，这也就是生成器表达式的嵌套：

```
$<UPPER_CASE:$<PLATFORM_ID>>
```

这个例子并不复杂，但很容易想象当增加嵌套级别，并在使用多个参数的命令中工作时会发生什么。

为了进一步复杂化，可以将常规变量在其中展开：

```
$<UPPER_CASE:${my_variable}>
```

变量 my_variable 将首先在配置阶段展开，生成器表达式将在生成阶段展开。这个特性有一些罕见的使用场景，但我强烈建议避免使用这种方式：生成器表达式提供了几乎所有必要的功能。将这些常规变量混合到表达式中增加了一层难以调试的间接性。此外，在配置阶段收集的信息通常会过时，因为用户会在构建或安装阶段通过命令行参数覆盖生成器表达式中使用的值。

了解了语法之后，让我们继续讨论生成器表达式中的基本机制。

6.4. 条件扩展

确定是否应该展开表达式，生成器表达式中通过布尔逻辑来支持。这是一个很棒的功能，但因为历史原因，其语法可能不一致且难以阅读。它有两种形式，第一种形式支持正面和负面路径：

```
$<IF:condition,true_string,false_string>
```

IF 表达式依赖于嵌套才能发挥作用：可以将任何一个参数替换为另一个表达式，并产生相当复杂的计算（甚至可以在一个 IF 条件中嵌套另一个）。这种形式需要正好三个参数，所以不能省略任何东西。在条件不满足时跳过值的最佳选项是以下这样：

```
$<IF:condition,true_string,>
```

有一个简写版本可以省略 IF 关键字和逗号：

```
$<condition:true_string>
```

其打破了将表达式名称作为第一个标记提供的惯例。我猜这里的意图是为了缩短表达式，并避免输入那些宝贵的几个字符，但结果可能真的很难理性化。以下是从 CMake 文档中取出的一例：

```
$<$<AND:$<COMPILE_LANGUAGE:CXX>,$<CXX_COMPILER_ID:AppleClang,Clang>>:COMPILING_CXX_WITH_CLANG>
```

这个表达式只有在用 Clang 编译器编译的 C++ 代码中才返回 COMPILING_CXX_WITH_CLANG（其他情况下返回空字符串）。我希望这个语法能够与常规 IF 命令的条件保持一致，但遗憾的是情况并非如此。现在，如果在某个地方看到了第二种形式，知道它是怎么工作的就好；但为了可读性，应该避免在自己的项目中使用。

6.4.1. 计算布尔值

生成器表达式计算为两种类型之一——布尔值或字符串。布尔值用 1（真）和 0（假）表示。没有专用的数值类型；除了布尔值之外的类型都只是字符串。

需要记住的是，作为条件表达式中的条件传递的嵌套表达式，明确要求计算为布尔值。

布尔类型可以隐式转换为字符串，但需要使用明确的 BOOL 运算符（稍后解释）来做相反的操作。

有三类表达式可计算为布尔值：逻辑运算符、比较表达式和查询。

逻辑运算符

有四个逻辑运算符：

- \$<NOT:arg>：否定布尔参数。
- \$<AND:arg1,arg2,arg3...>：如果所有参数都为真，则返回真。
- \$<OR:arg1,arg2,arg3...>：如果任一参数为真，则返回真。
- \$<BOOL:string_arg>：这将字符串参数从字符串转换为布尔类型。

使用 \$<BOOL> 的字符串转换在以下条件都不满足时，将计算为布尔真（1）：

- 字符串为空。
- 字符串是 0、FALSE、OFF、N、NO、IGNORE 或 NOTFOUND 的不区分大小写的等价物。
- 字符串以 -NOTFOUND 后缀结尾（区分大小写）。

比较

如果满足条件，比较将计算为 1，否则为 0。以下是一些可能有用的常见操作：

- `$<STREQUAL:arg1,arg2>`: 这以区分大小写的方式比较字符串。
- `$<EQUAL:arg1,arg2>`: 这将字符串转换为数字并比较相等性。
- `$<IN_LIST:arg,list>`: 这检查 `arg` 元素是否在 `list` 列表中（区分大小写）。
- `$<VERSION_EQUAL:v1,v2>`, `$<VERSION_LESS:v1,v2>`, `$<VERSION_GREATER:v1,v2>`, `$<VERSION_LESS_EQUAL:v1,v2>` 和 `$<VERSION_GREATER_EQUAL:v1,v2>` 以逐组件的方式比较版本。
- `$<PATH_EQUAL:path1,path2>`: 这比较两个路径的词法表示，不进行标准化（自 CMake 3.24 起）。

查询

查询直接从变量返回布尔值，或者作为操作的结果。

最简单的查询之一是：

```
$<TARGET_EXISTS:arg>
```

如果目标在配置阶段定义，将返回真。

现在，知道如何应用条件展开，使用逻辑运算符、比较和基本查询来计算为布尔值。但生成器表达式还有更多功能，特别是在查询的上下文中：可以在 IF 条件展开中使用，或者作为参数独立地传递给命令。

是时候在适当的上下文中介绍它们了。

6.5. 查询和转换

有许多生成器表达式可用，但为了避免迷失在细节中，我们将重点关注最常见的表达式。

先从可用的数据的基本转换开始。

6.5.1. 处理字符串、列表和路径

生成器表达式只提供了最基本的操作来转换和查询数据结构。在生成器阶段处理字符串，使用以下表达式：

- `$<LOWER_CASE:string>`, `$<UPPER_CASE:string>`: 这会将字符串转换为所需的小写。

从 CMake 3.15 开始，以下操作可用：

- `$<IN_LIST:string,list>`: 如果列表包含字符串值，则返回 `true`。
- `$<JOIN:list,d>`: 使用 `d` 分隔符将分号分隔的列表连接起来。
- `$<REMOVE_DUPLICATES:list>`: 去重列表（不排序）。
- `$<FILTER:list,INCLUDE|EXCLUDE,regex>`: 使用正则表达式从列表中包含/排除项目

从 3.27 开始，增加了 `$<LIST:OPERATION>` 生成器表达式，其中 `OPERATION` 是以下之一：

- LENGTH
- GET
- SUBLIST
- FIND
- JOIN
- APPEND
- PREPEND
- INSERT
- POP_BACK
- POP_FRONT
- REMOVE_ITEM
- REMOVE_AT
- REMOVE_DUPLICATES
- FILTER
- TRANSFORM
- REVERSE
- SORT

生成器表达式中处理列表相当罕见，所以只是了解其可能性。如果需要使用这些方式，请在线手册中查找如何使用这些操作的说明。

最后，可以查询和转换系统路径，这对于可移植性敏感的项目非常有用。自 CMake 3.24 以来，以下简单查询可用：

- `$<PATH:HAS_ROOT_NAME, path>`
- `$<PATH:HAS_ROOT_DIRECTORY, path>`
- `$<PATH:HAS_ROOT_PATH, path>`
- `$<PATH:HAS_FILENAME, path>`
- `$<PATH:HAS_EXTENSION, path>`
- `$<PATH:HAS_STEM, path>`
- `$<PATH:HAS_RELATIVE_PART, path>`
- `$<PATH:HAS_PARENT_PATH, path>`
- `$<PATH:IS_ABSOLUTE, path>`
- `$<PATH:IS_RELATIVE, path>`
- `$<PATH:IS_PREFIX[, NORMALIZE], prefix, path>`: 如果前缀是路径的前缀，则返回真。

相应地，可以检索能够检查的所有路径组件（自 CMake 3.27 起，不仅可以提供单个路径，还可以提供路径列表）：

- `$<PATH:CMAKE_PATH[, NORMALIZE], path...>`

- `$<PATH:APPEND,path...,input,...>`
- `$<PATH:REMOVE_FILENAME,path...>`
- `$<PATH:REPLACE_FILENAME,path...,input>`
- `$<PATH:REMOVE_EXTENSION[,LAST_ONLY],path...>`
- `$<PATH:REPLACE_EXTENSION[,LAST_ONLY],path...,input>`
- `$<PATH:NORMAL_PATH,path...>`
- `$<PATH:RELATIVE_PATH,path...,base_directory>`
- `$<PATH:ABSOLUTE_PATH[,NORMALIZE],path...,base_directory>`

此外，在 3.24 版本中引入了一些转换操作（只是为了完整性而列出它们）：

- `$<PATH:CMAKE_PATH[,NORMALIZE],path...>`
- `$<PATH:APPEND,path...,input,...>`
- `$<PATH:REMOVE_FILENAME,path...>`
- `$<PATH:REPLACE_FILENAME,path...,input>`
- `$<PATH:REMOVE_EXTENSION[,LAST_ONLY],path...>`
- `$<PATH:REPLACE_EXTENSION[,LAST_ONLY],path...,input>`
- `$<PATH:NORMAL_PATH,path...>`
- `$<PATH:RELATIVE_PATH,path...,base_directory>`
- `$<PATH:ABSOLUTE_PATH[,NORMALIZE],path...,base_directory>`

还有一个路径操作，提供的路径格式化为主机 shell 支持的样式：`$<SHELL_PATH:path ...>`。

再次，之前的表达式是为后续参考而引入的，而不是需要现在就记住。

6.5.2. 配置和平台的参数化

CMake 用户在构建项目时通常会提供所需构建配置的关键信息。大多数情况下，是 Debug 或 Release。可以使用生成器表达式，通过以下语句访问这些值：

- `$<CONFIG>`: 这将以字符串形式返回当前构建配置：Debug、Release 或另一个。
- `$<CONFIG:configs>`: 如果 configs 包含当前构建配置（不区分大小写的比较），则返回真。

第 4 章的“理解构建环境”部分讨论了平台，可以以与配置相同的方式读取相关信息：

- `$<PLATFORM_ID>`: 将以字符串形式返回当前平台 ID: Linux、Windows 或 Darwin (macOS)。
- `$<PLATFORM_ID:platform>`: 如果 platform 包含当前平台 ID，则为真。

这样的配置或平台特定的参数化，可以将其与之前讨论的条件展开一起使用：

```
$<IF:condition,true_string,false_string>
```

例如，可以为测试二进制文件和生产二进制文件应用不同的编译标志：

```
1 target_compile_definitions(my_target PRIVATE
2     $<IF:$<CONFIG:Debug>,Test,Production>
3 )
```

这只是开始，生成器表达式可以应对许多其他情况。

6.5.3. 调整工具链

工具链、工具包或简单地说，编译器和链接器在供应商之间并不一致。这带来了一系列后果。其中一些是积极的（特殊情况下性能更好），其他则不那么积极（配置多种多样，标志命名不一致等）。

生成器表达式通过提供可以用来缓解问题，并尽可能改进用户体验的查询集，来解决这个问题。

与构建配置和平台类似，有一些表达式返回有关工具链的信息，既有字符串也有布尔值。但必须指定感兴趣的语种（将 #LNG 替换为 C、CXX、CUDA、OBJC、OBJCXX、Fortran、HIP 或 ISPC）。在 3.21 版本中添加了对 HIP 的支持。

- \$<#LNG_COMPILER_ID>: 返回 #LNG 编译器的 CMake 编译器 ID。
- \$<#LNG_COMPILER_VERSION>: 返回 #LNG 编译器的 CMake 编译器版本。

为了检查 C++ 编译时将使用哪个编译器，应该使用 \$<CXX_COMPILER_ID> 生成器表达式。返回的值，即 CMake 的编译器 ID，是为每个支持的编译器定义的常量。可能会遇到如 AppleClang、ARMCC、Clang、GNU、Intel 和 MSVC 等值。完整的列表，请查看官方文档。

与前一个部分类似，也可以在条件表达式中利用工具链信息。如果提供的任何参数与特定值匹配，有一些查询会返回真：

- \$<#LNG_COMPILER_ID:ids>: 如果 ids 包含 CMake 的 #LNG 编译器 ID，则返回真。
- \$<#LNG_COMPILER_VERSION:vers>: 如果 vers 包含 CMake 的 #LNG 编译器版本，则返回真。
- \$<COMPILE_FEATURES:features>: 如果提供的所有 features 特征都由该目标的编译器支持，则返回真。

需要目标参数的命令中，例如 target_compile_definitions()，可以使用一个目标特定的表达式来获取字符串值：

- \$<COMPILE_LANGUAGE>: 这返回编译步骤中源文件的编程语言。
- \$<LINK_LANGUAGE>: 这返回链接步骤中源文件的编程语言。

为了计算一个简单的布尔查询：

- \$<COMPILE_LANGUAGE:langs>: 如果 langs 包含用于该目标编译的语言，则返回真。这可以用来为编译器提供特定于语言的标志。例如，为了使用 -fno-exceptions 标志编译目标 C++ 源：

```
1 target_compile_options(myapp
2   PRIVATE ${${COMPILER_LANGUAGE:CXX}:-fno-exceptions}
3 )
```

- `$<LINK_LANGUAGE:langs>` - 遵循与 `COMPILE_LANGUAGE` 相同的规则，如果 `langs` 包含用于该目标链接的语言，则返回真。

或者，为了查询更复杂的情况：

- `$<COMPILE_LANG_AND_ID:lang,compiler_ids...>`: 如果 `lang` 语言用于此目标，且编译器 ID 列表中的某个编译器将用于此编译，则返回真。这个表达式可以用来为特定编译器指定编译定义：

```
1 target_compile_definitions(myapp PRIVATE
2   ${${COMPILER_LANG_AND_ID:CXX,AppleClang,Clang}:CXX_CLANG}
3   ${${COMPILER_LANG_AND_ID:CXX,Intel}:CXX_INTEL}
4   ${${COMPILER_LANG_AND_ID:C,Clang}:C_CLANG}
5 )
```

- 这个例子中，对于使用 AppleClang 或 Clang 编译的 C++ 源 (CXX)，将设置 `-DCXX_CLANG` 定义。对于使用 Intel 编译器编译的 C++ 源，将设置 `-DCXX_INTEL` 定义标志。对于使用 Clang 编译的 C 源 (C)，将设置一个 `-DC_CLANG` 定义。
- `$<LINK_LANG_AND_ID:lang,compiler_ids...>`: 这像 `COMPILE_LANG_AND_ID` 一样工作，但检查链接步骤中使用的语言。使用表达式来指定特定语言和链接器组合的目标的链接库、链接选项、链接目录和链接依赖项。

这里需要强调的是，单个目标可以由多种语言的源文件组合而成。例如，可以链接 C 工件与 C++（应该在 `project()` 命令中声明这两种语言）。因此，引用特定语言的生成器表达式将用于一些源文件，但不会用于其他源文件。

6.5.4. 查询与目标相关的信息

有许多生成器表达式用于查询，目标属性和检查与目标相关的信息。需要注意的是，直到 CMake 3.19 版本，许多引用另一个目标的生成器表达式可用来自动创建它们之间的依赖关系。在最新版本的 CMake 中，这种行为已经不再发生。

一些生成器表达式会从调用的命令中推断目标；最常用的是基本查询，其返回目标属性的值：

```
$<TARGET_PROPERTY:prop>
```

- `target_link_libraries()` 命令中不太为人所知，但很有用的是 `$<LINK_ONLY:deps>` 生成器表达式。允许存储 `PRIVATE` 链接依赖，这些依赖不会通过传递的使用要求传播；这些在接口库中使用。

此外，还有一组与安装和导出相关的表达式，从它们使用的上下文中推断目标。后续我们将深入讨论它们，所以现在只给出一个快速的介绍：

- `$<INSTALL_PREFIX>`: 当目标使用 `install(EXPORT)` 导出或在内 `INSTALL_NAME_DIR` 评估时，返回安装前缀；否则，返回空。
- `$<INSTALL_INTERFACE:string>`: 使用 `install(EXPORT)` 导出时，返回 `string`。
- `$<BUILD_INTERFACE:string>`: 使用 `export()` 命令或同一构建系统中的另一个目标导出时，返回 `string`。
- `$<BUILD_LOCAL_INTERFACE:string>`: 同一构建系统中的另一个目标导出时，返回 `string`。

然而，大多数查询都需要明确提供目标名称作为第一个参数：

- `$<TARGET_EXISTS:target>`: 如果目标存在，则返回真。
- `$<TARGET_NAME_IF_EXISTS:target>`: 如果目标存在，则返回目标名称，否则返回空字符串。
- `$<TARGET_PROPERTY:target,prop>`: 返回目标 `prop` 属性的值。
- `$<TARGET_OBJECTS:target>`: 返回对象库目标的对象文件列表。

可以查询目标工件的路径：

- `$<TARGET_FILE:target>`: 返回完整的路径。
- `$<TARGET_FILE_NAME:target>`: 只返回文件名。
- `$<TARGET_FILE_BASE_NAME:target>`: 只返回基本名称。
- `$<TARGET_FILE_NAME:target>`: 返回不带前缀或后缀的基本名称（对于 `libmylib.so`，基本名称将是 `mylib`）。
- `$<TARGET_FILE_PREFIX:target>`: 只返回前缀（例如，`lib`）。
- `$<TARGET_FILE_SUFFIX:target>`: 只返回后缀（例如，`.so` 或 `.exe`）。
- `$<TARGET_FILE_DIR:target>`: 只返回目录。

有一些表达式提供与常规 `TARGET_FILE` 表达式相似的功能（每个表达式也接受 `_NAME`、`_BASE_NAME` 或 `_DIR` 后缀）：

- `TARGET_LINKER_FILE`: 查询在链接到目标时使用的文件的路径。通常，这是目标产生的库（`.a`、`.lib`、`.so`）。然而，在具有动态链接库（DLL）的平台，将是与目标 DLL 相关联的 `.lib` 导入库。
- `TARGET_PDB_FILE`: 查询链接器生成的程序数据库文件（`.pdb`）的路径。

管理库是一个复杂的话题，CMake 提供了很多生成器表达式来帮助，我们将在第 8 章中进行详细介绍。

最后，还有一些针对 Apple 包的特定表达式：

- `$<TARGET_BUNDLE_DIR:target>`: 这是目标（`my.app`、`my.framework` 或 `my.bundle`）的捆绑目录的完整路径。
- `$<TARGET_BUNDLE_CONTENT_DIR:target>`: 这是目标捆绑内容目录的完整路径。在 macOS 上，是 `my.app/Contents`、`my.framework` 或 `my.bundle/Contents`。其他软件开发工具包（SDKs）（如 iOS）具有扁平的捆绑结构——`my.app`、`my.framework` 或 `my.bundle`。

这些是与目标打交道的生成器表达式的主要内容。还有更多的内容，建议参考官方文档以获取完整的列表。

6.5.5. 转义

极少数情况下，可能需要向生成器表达式传递一个具有特殊含义的字符。为了转义这种行为，请使用以下表达式：

- `$<ANGLE-R>`: 一个 `>` 符号。
- `$<COMMA>`: 一个逗号符号。
- `$<SEMICOLON>`: 一个分号符号。

最后一个表达式在包含分号的参数时使用，可以用来避免列表展开。

现在已经介绍了所有的查询和转换，可以看到其在实践中的工作方式。让我们继续通过一些应用示例来进行学习吧！

6.6. 尝试示例

有了好的实际例子，理论就更容易掌握。显然，我们想要编写一些 CMake 代码并尝试一下。然而，由于生成器表达式直到配置完成之后才进行计算，不能使用配置时的命令，如 `message()` 来验证。为了调试生成器表达式需要使用一些特殊的技巧，可以使用以下方法：

- 将其写入文件 (`file()` 命令的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，并显式构建：`add_custom_target(gendbg COMMAND ${CMAKE_COMMAND} -E echo "$<...>")`

我推荐第一个选项进行简单的练习。记住，我们无法在这些命令中使用所有的表达式，因为有些是特定于目标的。

了解了这一点，再来看一下生成器表达式的某些用途。

6.6.1. 构建配置

在第 1 章中，讨论了构建类型，指定正在构建的配置——`Debug`、`Release` 等。可能有些情况下，希望根据正在进行的构建类型进行不同的操作。一个简单且直接的方法是使用 `$<CONFIG>` 生成器表达式：

```
target_compile_options(tgt $<$<CONFIG:DEBUG>:-finline-points>)
```

前面的例子检查配置是否等于 `DEBUG`；如果是这样，嵌套表达式计算为 1。然后外部的简写 `if` 表达式变为真，`-finline-points` 调试标志就添加到选项中。

了解这种形式很重要，这样就能够理解其他项目中的此类表达式，但我建议为了更好的可读性，可以使用更详细的 `$<IF:>.`

6.6.2. 系统特定的单行命令

生成器表达式还可以用来将冗长的 `if` 命令压缩成整洁的单行命令。

假设我们有以下代码：

```
1 if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
2     target_compile_definitions(myProject PRIVATE LINUX=1)
3 endif()
```

它告诉编译器，如果这是目标系统，则向参数中添加 `-DLINUX=1`。虽然这并不算太长，但可以用一个相当简单的表达式替换：

```
1 target_compile_definitions(myProject PRIVATE
2     ${${CMAKE_SYSTEM_NAME}:LINUX=1})
```

这样的代码运行良好，但可以在生成器表达式中打包的内容有限，直到变得难以阅读。此外，许多 CMake 用户会推迟学习生成器表达式，并难以跟踪发生的事情。幸运的是，在完成本章后，我们就不会遇到这样的问题。

6.6.3. 带有编译器特定标志的接口库

如第 5 章所讨论，接口库可以用来提供与编译器匹配的标志：

```
1 add_library(enable_rtti INTERFACE)
2 target_compile_options(enable_rtti INTERFACE
3     ${${COMPILER_ID}:GNU,-rtti}
4 )
```

即使如此简单的例子中，当嵌套太多生成器表达式时，理解表达式也会变得难以理解。但有时这是实现期望效果的唯一方式。以下是例子的解释：

- 检查 `COMPILER_ID` 是否为 `GNU`；如果是，将 OR 计算为 1。
- 如果不是，检查 `COMPILER_ID` 是否为 `Clang`，并将 OR 计算为 1。否则，将 OR 评估为 0。
- 如果 OR 计算为 1，将 `-rtti` 添加到 `enable_rtti` 编译选项中。否则，不做任何事情。

接下来，可以将库和可执行文件与 `enable_rtti` 接口库链接起来。如果编译器支持，CMake 将添加 `-rtti` 标志。RTTI 代表运行时类型信息，在 C++ 中与 `typeid` 等关键字一起使用，以便在运行时确定对象的类；除非代码使用这个功能，否则不需要启用该标志。

6.6.4. 嵌套生成器表达式

有时，会尝试在生成器表达式中嵌套元素，但会发生什么并不明显。我们可以通过生成一个测试，将其输出到调试文件来调试表达式。

看看会发生什么：

ch06/01-nesting/CMakeLists.txt

```
1 set(myvar "small text")
2 set(myvar2 "small text >")
3
4 file(GENERATE OUTPUT nesting CONTENT "
5   1 ${PLATFORM_ID}
6   2 ${UPPER_CASE:${PLATFORM_ID}}
7   3 ${UPPER_CASE:hello world}
8   4 ${UPPER_CASE:${myvar}}
9   5 ${UPPER_CASE:${myvar2}}
10  ")
```

在构建此项目后，可以使用 Unix 的 cat 命令读取生成的 nesting 文件：

```
# cat nesting

1 Linux
2 LINUX
3 HELLO WORLD
4 SMALL TEXT
5 SMALL text>
```

这就是每行的工作内容：

1. PLATFORM_ID 的输出值是 LINUX。
2. 嵌套值的输出将被正确地转换为大写的 LINUX。
3. 可以转换普通字符串。
4. 可以转换配置阶段的变量内容。
5. 变量将首先插值，然后闭合的尖括号 (>) 将解释为生成器表达式的一部分，只有字符串的部分大写。

注意，变量的内容可能会影响生成器表达式的扩展行为。如果需要在变量中使用尖括号，请使用 \${<ANGLE-R>}。

6.6.5. 布尔表达式与 BOOL 运算符的计算差异

当涉及到将布尔类型计算为字符串时，生成器表达式可能会有些令人困惑。了解它们与常规条件表达式的区别至关重要，尤其是从 IF 关键字开始：

ch06/02-boolean/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Boolean CXX)
3
```

```
4 file(GENERATE OUTPUT boolean CONTENT "
5   1 $<0:TRUE>
6   2 $<0:TRUE,FALSE> (won't work)
7   3 $<1:TRUE,FALSE>
8   4 $<IF:0,TRUE,FALSE>
9   5 $<IF:0,TRUE,>
10  "")
```

使用 Linux 的 cat 命令读取生成的文件：

1. 这是一个布尔展开，其中 BOOL 是 0；因此，TRUE 字符串不会写入。
2. 这是一个典型的错误——作者原本打算根据 BOOL 值打印 TRUE 或 FALSE，但也是一个布尔假的展开，两个参数当作一个参数处理，因此不会输出。
3. 这是相同错误的一个反转值——它是一个布尔真的展开，两者都在同一行写入。
4. 这是一个以 IF 开头的正确条件表达式——输出 FALSE，因为第一个参数是 0。
5. 这是条件表达式的正确用法，但当不需要为布尔假提供值时，应该使用第一行中的方式。

生成器表达式因其复杂的语法而声名狼藉，本例中提到的差异甚至可能会让经验丰富的构建者感到困惑。如果有所疑虑，将这样的表达式复制到另一个文件中，并通过添加缩进和空格来分析，以更好地理解。

了解了生成器表达式的工作示例，从而为我们使用它们做好了准备。接下来的章节将讨论许多与生成器表达式相关的主题。

6.7. 总结

本章全部内容都是关于剖析生成器表达式（或称“genexes”）的细节。我们从生成器表达式的形式和扩展的基础知识开始，并查了解了它的嵌套机制。深入探讨了条件扩展的强大功能，其利用了布尔逻辑、比较操作和查询。当根据用户选择的构建配置、平台和当前工具链等因素调整构建过程时，生成器表达式的这一方面尤为突出。

我们还了解了字符串、列表和路径的基本但重要的转换。一个重点亮点是使用查询在后期构建阶段收集的信息，并在上下文符合要求时呈现。现在也知道如何检查编译器的 ID、版本和功能。探索了使用生成器表达式查询构建目标属性，并提取了相关信息。并以实用的示例和可能的输出查看指南作为结尾。有了这些，就可以在项目中使用生成器表达式了。

下一章中，将学习如何使用 CMake 编译程序。具体来说，将讨论如何配置和优化这个过程。

6.8. 扩展阅读

- 官方文档中的生成器表达式：

<https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html>
1

- 支持的编译器 ID：

https://cmake.org/cmake/help/latest/variable/CMAKE_LANG_COMPILER_ID.html

- 在 CMake 中混合使用多种语言:

<https://stackoverflow.com/questions/8096887/mixing-c-and-c-with-cmake>

第 7 章 用 CMake 编译 C++ 源代码

简单的编译场景通常由工具链的默认配置处理，或者由集成开发环境（IDE）直接提供。专业环境中，商业需求往往需要更高级的功能。可能是对更高性能、更小的二进制文件、更好的可移植性、自动化测试或广泛的调试功能的需求——不胜枚举。要以连贯且面向未来的方式管理所有这些需求，很快就会变得复杂而混乱（特别是在需要支持多个平台时）。

编译过程在 C++ 书籍中通常没有解释得足够清楚（深入主题，如虚基类，似乎更有趣）。本章中，将通过探讨编译的不同方面来纠正这一点：将了解编译如何工作，其内部阶段是什么，以及如何影响二进制输出。

之后，将关注先决条件——讨论可以使用哪些命令来微调编译过程，如何要求编译器具有特定功能，以及如何正确指导编译器处理哪些输入文件。

然后，关注编译的第一阶段——预处理器。提供包含头文件的路径，并且将学习如何通过预处理器定义将 CMake 和构建环境中的变量插入。我们将涵盖最有趣的使用案例，并学习如何暴露 CMake 变量，以便它们可以在 C++ 中访问。

紧接着，将讨论优化器，以及不同的标志如何影响性能。还将讨论优化的成本，特别是它如何影响产生的二进制的可调试性，以及如果不希望这样该怎么办。

最后，将解释如何通过使用预编译头文件和统一构建来管理编译过程，以减少编译时间。并了解如何调试构建过程，并找出可能犯的错误。

本章中，将包含以下内容：

- 编译基础
- 配置预处理器
- 配置优化器
- 管理编译过程

7.1. 示例下载

可以在 GitHub 上的以下链接找到本章中出现的代码文件：<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch07>。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保将 `<build tree>` 和 `<source tree>` 占位符替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的位置。

7.2. 编译的基础

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。这使得我们能够使用类和对象等抽象概念来创建应用程序，而无需处理特定处理器的汇编语言中的繁琐细节。

我们不需要直接与 CPU 寄存器打交道，考虑短跳或长跳，或管理栈帧。编译语言更具表现力、可读性和安全性，鼓励创建可维护的代码，同时尽可能提供性能。

在 C++ 中，使用静态编译——整个程序必须转换为本地代码后才能执行。这与 Java 或 Python 等语言不同，后者在用户每次运行程序时都会实时解释和编译程序。每种方法都有其独特的优势。C++ 旨在提供大量高级工具，同时提供本地性能。C++ 编译器可以为几乎所有的架构生成独立的应用程序。

创建和运行 C++ 程序涉及以下步骤：

1. 设计应用程序：这包括规划应用程序的功能、结构和行为。设计完成后，按照最佳实践仔细编写源代码，以保持代码的可读性和可维护性。
2. 将单独的 .cpp 实现文件（也称为翻译单元）编译成目标文件：这一步涉及将编写的高级语言代码转换为低级机器代码。
3. 将目标文件链接成单一的可执行文件：此步骤中，还会链接所有其他依赖项，包括动态和静态库。这个过程创建了一个可以在预期平台上运行的可执行文件。

要运行程序，操作系统（OS）将使用一个名为加载器的工具，将程序的机器代码和所有必需的动态库映射到虚拟内存中。加载器然后读取程序头部，以确定执行应从何处开始，并开始运行指令。

在此阶段，程序的启动代码发挥作用。系统 C 库提供的名为_start 的特殊函数被调用。_start 函数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。在此之后，才会调用 main()，开发者会在其中填入自己的代码。

所以，幕后进行了大量工作。本章关注前面列表中的第二步。通过考虑大局，可以更好地理解决潜在问题可能起源于何处。尽管软件开发的复杂性看似难以理解，但其中并没有魔法。一切都有解释和原因。我们需要理解，即使编译步骤本身看起来是成功，程序在运行时可能会因编译方式而出错。编译器在其操作过程中无法检查所有边缘情况。因此，让我们找出编译器执行工作时实际发生的情况。

7.2.1. 编译如何工作

编译是将高级语言转换为低级语言的过程，这涉及生成机器代码，即特定处理器可以直接执行的指令，以特定于给定平台的二进制目标文件格式。在 Linux 上，最常用的格式是可执行和可链接格式（ELF），Windows 使用 PE/COFF 格式规范，macOS 为 Mach 对象（Mach-O 格式）。

目标文件是单个源文件的直接翻译。每个文件都必须单独编译，然后由链接器组合成单一的可执行文件或库。这种模块化过程在修改代码时，因为只有程序员更新的文件需要重新编译，可以显著节省时间。

编译器必须执行以下阶段以创建目标文件：

- 预处理
- 语法语义分析
- 汇编
- 优化
- 代码生成

让我们详细地解释它们。

大多数编译器会自动调用，但预处理可视为实际编译之前的准备步骤。其作用是执行源代码的基本操作；它执行 `#include` 指令，通过 `#define` 指令和 `-D` 标志用定义的值替换标识符，调用简单宏，并根据 `#ifdef`、`#elif` 和 `#endif` 指令有条件地包含或排除代码部分。预处理器对实际的 C++ 代码一无所知，它充当一个高级的查找和替换工具。

然而，预处理器在构建高级程序中的作用至关重要。将代码划分为部分，并在多个翻译单元之间共享声明，是代码可重用的基础。

接下来是语言学分析，编译器进行更复杂的操作。逐字符扫描预处理后的文件（现在包括预处理器插入的所有头文件）。通过称为词法分析的过程，将字符组合成有意义的符号——关键字、运算符、变量名等。

然后，将这些符号组装成链，并检查它们的顺序和存在是否遵守 C++ 的语法规则——这个过程称为语法分析或解析。这通常是生成大多数错误消息的阶段，因为其会识别语法问题。

最后，编译器进行语义分析。此阶段，编译器检查文件中的语句是否逻辑上合理。例如，确保满足所有类型正确性检查（不能将整数分配给字符串变量）。这种分析确保程序在编程语言的规则内有意义。

汇编阶段本质上是将这些符号，根据平台的可用指令集转换为 CPU 特定的指令。一些编译器实际生成汇编输出文件，然后将其传递给专用的汇编程序。这个程序生成 CPU 可以执行的机器代码。其他编译器直接在内存中生成此机器代码，这类编译器还提供生成人类可读汇编代码的文本输出的选项。

优化并不局限于编译过程中的单一步骤，而是每个阶段逐渐进行。在初始汇编产生之后，有一个明确的阶段专注于最小化寄存器使用和消除冗余代码。

一种有趣且值得注意的优化技术是内联展开或内联。这个过程中，编译器有效地“剪切”函数体，并将其“粘贴”到其调用的位置。

C++ 标准并未明确定义这种情况发生的条件——取决于实现。内联展开可以提高执行速度和减少内存使用，但同时也为调试带来了重大缺陷，执行的代码不再对应于源代码中的原始行。

代码生成阶段涉及将优化后的机器代码写入目标文件，格式与目标平台的规范相符。然而，这个目标文件尚未准备好执行——需要传递到链中的下一个工具，即链接器。链接器的工作是适当地重定位目标文件的部分，并解析对外部符号的引用，有效地准备文件执行。这一步骤标志着从美国信息交换标准代码（ASCII）源码，到可以直接由 CPU 处理的二进制可执行文件的转变。

这些阶段每一个都很重要，并且可以根据我们的具体需求进行配置。让我们看看如何使用 CMake 管理这个过程。

7.2.2. 初始配置

CMake 提供了几个可以影响编译每个阶段的命令：

- `target_compile_features()`: 要求编译器具有特定的功能来编译这个目标。
- `target_sources()`: 向已定义的目标添加源文件。
- `target_include_directories()`: 设置预处理器包含路径。
- `target_compile_definitions()`: 设置预处理器定义。

- `target_compile_options()`: 设置编译器特定的命令行选项。
- `target_precompile_headers()`: 设置要优化的外部头文件。

这些命令接受以下格式的类似参数：

```
target_...(<target name> <INTERFACE|PUBLIC|PRIVATE> <arguments>)
```

所以使用此命令设置的属性会通过传递使用要求，如第 5 章“处理目标”中所述，并且可以用于可执行文件和库，并且这些命令都支持生成器表达式。

要求编译器的特定功能

如第 4 章中的“检查支持的编译器功能”部分所讨论，预测问题并提供清晰的错误消息给软件用户至关重要——例如，当可用的编译器 X 不提供所需的功能 Y 时。这种方法比让用户解析，可能使用的兼容工具链产生的错误要友好得多。我们不想让用户将不兼容性问题归咎于我们的代码，而是他们过时的环境。

可以使用以下命令指定目标构建所需的所有功能：

```
target_compile_features(<target> <PRIVATE|PUBLIC|INTERFACE>
                      <feature> [...])
```

CMake 理解以下 `compiler_id` 的 C++ 标准和支持的编译器功能：

- AppleClang: 适用于 Xcode 版本 4.4+ 的 Apple Clang
- Clang: Clang 编译器版本 2.9+
- GNU: GNU 编译器版本 4.4+
- MSVC: Microsoft Visual Studio 版本 2010+
- SunPro: Oracle Solaris Studio 版本 12.4+
- Intel: Intel 编译器版本 12.1+

CMake 支持超过 60 种功能，以在官方文档中找到完整列表，该文档解释了 `CMAKE_CXX_KNOWN_FEATURES` 变量。然而，除非需要非常具体的东西，否则建议选择指示 C++ 标准的高级元功能：

- `cxx_std_14`
- `cxx_std_17`
- `cxx_std_20`
- `cxx_std_23`
- `cxx_std_26`

看看以下示例：

```
1 target_compile_features(my_target PUBLIC cxx_std_26)
```

这等同于 `set(CMAKE_CXX_STANDARD 26)` 和 `set(CMAKE_CXX_STANDARD_REQUIRED ON)`。不同之处在于 `target_compile_features()` 按目标工作，而不是全局地针对项目，如果需要为项目中的所有目标添加，可能会很繁琐。

关于 CMake 支持的编译器的详细信息，请参阅官方手册。

7.2.3. 管理目标源文件

我们已经知道如何告诉 CMake 哪些源文件构成一个单独的目标，无论是可执行文件还是库。可以通过在 `add_executable()` 或 `add_library()` 命令中提供文件列表来实现。

随着解决方案的扩展，每个目标的文件列表也会增长。这可能导致一些相当长的 `add_…()` 命令。要如何处理这种情况？一个方法可能是使用 GLOB 模式的 `file()` 命令，它可以收集子目录中的所有文件并将它们存储在变量中。可以将其作为目标声明的参数传递，不再担心文件列表：

```
1 file(GLOB helloworld_SRC "*.h" "*.cpp")
2 add_executable(helloworld ${helloworld_SRC})
```

然而，这种方法并不推荐。CMake 根据列表文件的变化生成构建系统，所以如果没有检测到变化，构建可能会在没有警告的情况下失败（这是开发者的噩梦）。此外，省略目标声明中的所有源文件，可能会干扰像 CLion 这样的 IDE 中的代码检查，因为它知道如何解析某些 CMake 命令，以理解项目。

在目标声明中使用变量，并不建议的另一个原因是：创建了一个间接层，导致开发者在阅读项目时必须解包目标定义。遵循这一建议，需要面临另一个问题：如何条件性地添加源文件？这是一个常见的场景，当处理特定于平台的实现文件时，如 `gui_linux.cpp` 和 `gui_windows.cpp`。

`target_sources()` 命令允许源文件附加到已创建的目标：

ch07/01-sources/CMakeLists.txt

```
1 add_executable(main main.cpp)
2 if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
3     target_sources(main PRIVATE gui_linux.cpp)
4 elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
5     target_sources(main PRIVATE gui_windows.cpp)
6 elseif(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
7     target_sources(main PRIVATE gui_macos.cpp)
8 else()
9     message(FATAL_ERROR "CMAKE_SYSTEM_NAME=${CMAKE_SYSTEM_NAME} not
10 supported.")
11 endif()
```

这样，每个平台都获得了自己的兼容文件集。但长列表的源文件怎么办？好吧，只能接受有些事情还不完美，并继续手动添加。如果正在处理一个非常长的列表，可能会发现项目结构有些问题，也许应该将一些源文件划分为库。

既然已经了解了编译的基本知识，让我们深入探讨第一步——预处理。

7.3. 配置预处理器

预处理器在构建过程中扮演着重要角色。考虑到其功能看起来相当直接和有限，在以下各节中，我们将介绍提供包含文件的路径以及使用预处理器定义。还将解释如何使用 CMake 来配置包含的头文件。

7.3.1. 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 和.hpp 头文件，它有两种形式：

- 尖括号形式：`#include <path-spec>`
- 引号形式：`#include "path-spec"`

预处理器将用路径规范中指定的文件内容替换这些指令。找到这些文件可能是一个挑战。应该搜索哪些目录，以及按照什么顺序？遗憾的是，C++ 标准并没有确切规定这一点，必须查阅正在使用的编译器的手册。

通常，尖括号形式将检查标准包含目录，这些目录包括存储在系统中的标准 C++ 库和标准 C 库头文件的目录。

引号形式首先在当前文件的目录中搜索包含的文件，然后检查尖括号形式的目录。CMake 提供了一个命令来操作搜索包含文件的路径

```
target_include_directories(<target> [SYSTEM] [AFTER|BEFORE]
                           <INTERFACE|PUBLIC|PRIVATE> [item1...]
                           [<INTERFACE|PUBLIC|PRIVATE> [item2...]
...])
```

这可以添加自定义路径，以便编译器扫描。CMake 将其添加到生成的构建系统中的编译器中，将使用特定的标志以提供（通常是`-I`）给编译器。

`target_include_directories()` 命令通过追加或预置目录，来修改目标的 `INCLUDE_DIRECTORIES` 属性，具体取决于是否使用了 `AFTER` 或 `BEFORE` 关键字。然而，由编译器决定，是否在默认目录之前或之后检查这里提供的目录（通常是之前）。

`SYSTEM` 关键字告诉编译器给定的目录应该视为标准系统目录（与尖括号形式一起使用）。对于许多编译器来说，这些目录是通过`-isystem` 标志传递的。

7.3.2. 预处理器定义

回想一下前面在讨论编译阶段时，提到预处理器的 `#define` 和 `#if`、`#elif`，以及 `#endif` 指令。来检查一下以下示例：

ch07/02-definitions/definitions.cpp

```
1 #include <iostream>
2 int main() {
```

```
3 #if defined(ABC)
4     std::cout << "ABC is defined!" << std::endl;
5 #endif
6 #if (DEF > 2*4-3)
7     std::cout << "DEF is greater than 5!" << std::endl;
8 #endif
9 }
```

目前，这个示例什么也没完成，因为 ABC 和 DEF 都没有定义（DEF 将默认为 0）。我们可以通过在此代码顶部添加两行来轻松更改：

```
1 #define ABC
2 #define DEF 8
```

编译并执行此代码后，可以在控制台看到两条消息：

```
ABC is defined!
DEF is greater than 5!
```

这可能看起来足够简单，但如果想基于外部因素（如操作系统、架构或其他因素）来条件这些部分呢？可以将值从 CMake 传递给 C++ 编译器。

`target_compile_definitions()` 命令就足够了：

ch07/02-definitions/CMakeLists.txt

```
1 set(VAR 8)
2 add_executable(define_definitions.cpp)
3 target_compile_definitions(define PRIVATE ABC "DEF=${VAR}")
```

前面的代码将表现得与两个 `#define` 语句完全一样，但有使用 CMake 的变量和生成器表达式的灵活性，并且可以将命令放在条件块中。

这些定义是通过-D 标志（例如，-DFOO=1）传递给编译器的，一些开发者会继续在这个命令中使用这个标志：

```
1 target_compile_definitions(hello PRIVATE -DFOO)
```

CMake 识别这一点，并将自动移除前导的-D 标志。它还会忽略空字符串，所以以下命令也是有效的：

```
1 target_compile_definitions(hello PRIVATE -D FOO)
```

这种情况下，-D 是一个单独的参数，移除后变成一个空字符串，然后忽略，从而确保正确的行为。

避免在单元测试中访问私有类字段

一些在线资源建议使用特定的-D 定义与 #ifdef/ifndef 指令结合用于单元测试的目的。这种方法最直接的应用是将公共访问说明符包含在条件包含中，当定义了 UNIT_TEST 时，有效地使所有字段变为公共的（默认情况下，类字段私有）：

```
1 class X {  
2     #ifdef UNIT_TEST  
3         public:  
4     #endif  
5         int x_;  
6 }
```

虽然这种方法提供了便利（允许测试直接访问私有成员），但并没有产生干净的代码。单元测试应该专注于验证公共接口内的方法的功能性，将底层实现视为一个黑盒，我建议只在万不得已的情况下使用这种方法。

使用 git 提交跟踪编译版本

思考一下哪些用例需要了解关于环境或文件系统的详细信息。专业环境中，一个典型的例子可能涉及，传递用于构建二进制文件的修订版本或提交 SHA。可以这样实现：

ch07/03-git/CMakeLists.txt

```
1 add_executable(print_commit print_commit.cpp)  
2 execute_process(COMMAND git log -1 --pretty=format:%h  
3                     OUTPUT_VARIABLE SHA)  
4 target_compile_definitions(print_commit  
5                             PRIVATE "SHA=${SHA}")
```

然后可以在应用程序中使用 SHA：

ch07/03-git/print_commit.cpp

```
1 #include <iostream>  
2 // special macros to convert definitions into c-strings:  
3 #define str(s) #s  
4 #define xstr(s) str(s)  
5 int main()  
6 {  
7     #if defined(SHA)  
8         std::cout << "GIT commit: " << xstr(SHA) << std::endl;  
9     #endif  
10 }
```

前面的代码要求用户安装了 Git，并将其添加到 PATH 中。当运行在生产服务器上的程序是持续集成/部署管道的结果时，这个功能尤其有用。如果软件出现问题，可以快速检查使用了哪个确切的 Git 提交来构建了有故障的产品。

跟踪确切的提交对于调试目的非常有用。将单个变量传递给 C++ 代码很简单，但是如果需要将数十个变量传递给头文件，该如何处理呢？

7.3.3. 配置头文件

通过 `target_compile_definitions()` 传递定义在变量众多时会变得繁琐。提供带有引用这些变量的占位符的头文件，并允许 CMake 填充它们，这样不是更容易吗？当然可以！

CMake 的 `configure_file(<input> <output>)` 命令使你能够从模板生成新文件，如下例所示：

ch07/04-configure/configure.h.in

```
1 #cmakedefine FOO_ENABLE
2 #cmakedefine FOO_STRING1 "@FOO_STRING1@"
3 #cmakedefine FOO_STRING2 "${FOO_STRING2}"
4 #cmakedefine FOO_UNDEFINED "@FOO_UNDEFINED@"
```

可以按以下方式使用这个命令：

ch07/04-configure/CMakeLists.txt

```
1 add_executable(configure configure.cpp)
2 set(FOO_ENABLE ON)
3 set(FOO_STRING1 "abc")
4 set(FOO_STRING2 "def")
5 configure_file(configure.h.in configured/configure.h)
6 target_include_directories(configure PRIVATE
7 ${CMAKE_CURRENT_BINARY_DIR})
```

然后 CMake 生成如下输出文件：

ch07/04-configure/<build_tree>/configured/configure.h

```
1 #define FOO_ENABLE
2 #define FOO_STRING1 "abc"
3 #define FOO_STRING2 "def"
4 /* #undef FOO_UNDEFINED */
```

，@VAR@ 和 \${VAR} 变量占位符使用 CMake 列表文件中的值替换了。此外，对于已定义的变量，#cmakedefine 替换为 #define，对于未定义的变量，则替换为/* #undef VAR */。如果要为 #if 块显式地 #define 1 或 #define 0，请使用 #cmakedefine01。

可以通过在实现文件中包含这个配置好的头文件，将其合并到应用程序中：

ch07/04-configure/configure.cpp

```

1 #include <iostream>
2 #include "configured/configure.h"
3
4 // special macros to convert definitions into c-strings:
5 #define str(s) #s
6 #define xstr(s) str(s)
7
8 using namespace std;
9 int main()
10 {
11 #ifdef FOO_ENABLE
12     cout << "FOO_ENABLE: ON" << endl;
13 #endif
14     cout << "FOO_STRING1: " << xstr(FOO_STRING1) << endl;
15     cout << "FOO_STRING2: " << xstr(FOO_STRING2) << endl;
16     cout << "FOO_UNDEFINED: " << xstr(FOO_UNDEFINED) << endl;
17 }

```

使用 `target_include_directories()` 命令将二进制树添加到包含搜索路径中，可以编译示例并从 CMake 接收填充的输出：

```

FOO_ENABLE: ON
FOO_STRING1: "abc"
FOO_STRING2: "def"
FOO_UNDEFINED: FOO_UNDEFINED

```

`configure_file()` 命令还包括一系列的格式化和文件权限选项，但由于篇幅限制，我们在这里不深入讨论。如果感兴趣，可以参考在线文档以获取更多详细信息（请参阅本章的“扩展阅读”部分）。

准备好头文件和源文件的完整编译之后，让我们讨论在后续步骤中输出代码如何形成。虽然无法直接控制语言分析或组装（这些步骤遵循严格的标准），但可以操纵优化器的配置。

7.4. 配置优化器

优化器将分析前几个阶段的结果，并使用多种策略，这些策略开发者通常不会直接使用，它们不符合干净代码的原则。但这没关系——优化器的基本作用是提高代码性能，力求降低 CPU 使用率、最小化寄存器使用和减少内存占用。当优化器遍历源代码时，针对目标 CPU，会将代码大量变形为无法识别的形式。

优化器不仅会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！如果明确确定某些代码多余，甚至会从重要函数的中间删除这些代码。通过回收内存，使得多个变量可以在不同时间占据同一个槽位。如果这样做可以减少几个周期的耗时，甚至可以将控制结构重塑为完全不同的形式。

如果开发者手动将这些技术应用于源码，代码将变成一个糟糕的、难以阅读的混乱，难以编写和理解。然而，因为编译器严格遵循提供的指令，所以当编译器应用这些技术时是有利的。优化器

是一个不知疲倦的怪兽，只有一个目的：加速执行速度，不管输出变得多么扭曲。

每个编译器都有自己的独特技巧，这与它支持的平台和遵循的思想一致。我们将了解 GNU GCC 和 LLVM Clang 中最常见的一些，以清楚什么是实用的和可实现的。

问题是——许多编译器默认情况下不会启用优化（包括 GCC）。当可以快时，为什么要慢下来？为了纠正这一点，可以使用 `target_compile_options()` 命令，明确声明我们对编译器的期望。

这个命令的语法与本章中的其他命令类似：

```
target_compile_options(<target> [BEFORE]
                      <INTERFACE|PUBLIC|PRIVATE> [items1...]
                      [<INTERFACE|PUBLIC|PRIVATE> [items2...]
...])
```

我们提供在构建目标时使用的命令行选项，还指定传播关键字。执行时，CMake 会将给定的选项附加到目标的适当 `COMPILE_OPTIONS` 变量中。如果要前置，可以使用可选的 `BEFORE` 关键字。某些情况下，顺序很重要。

注意，`target_compile_options()` 是一个通用命令，还可以用来提供其他类似编译器的参数，例如-D 定义，CMake 为此提供了 `target_compile_definition()` 命令。始终建议尽可能使用最专业的 CMake 命令，以保证在所有支持的编译器上以相同的方式工作。

7.4.1. 通用级别

优化器的所有不同行为，都可以通过特定的标志配置，可以作为编译选项传递。了解所有这些标志需要花费大量时间，并且需要对编译器、处理器和内存的内部有深入的了解。如果只想得到在大多数情况下都很好的最佳方案，能做什么呢？可以寻求一个通用解决方案——优化级别指定器。

大多数编译器提供从 0 到 3 的四个基本优化级别，使用 `-O< 级别 >` 选项指定。`-O0` 表示没有优化，通常它是编译器的默认级别。另一方面，`-O2` 认为是完全优化，会生成高度优化的代码，但代价是编译时间最慢。

还有一个介于两者之间的 `-O1` 级别，可以是一个很好的折衷方案——启用了一定量的优化机制，而不会过度减慢编译速度。

最后，可以使用 `-O3`，是与 `-O2` 相同的完全优化，但对子程序内联和循环向量化采取了更激进的方法。

还有一些优化变体，针对生成文件的大小（不一定是速度）进行优化——`-Os`。有一种超级积极的优化，`-Ofast`，是一个不严格遵循 C++ 标准的 `-O3` 优化。最明显的区别是使用 `-ffast-math` 和 `-ffinite-math` 标志，如果程序是关于精确计算（大多数是），最好别使用。

CMake 知道并非所有编译器都相同，为了给开发者提供标准化的体验，其为编译器提供了一些默认标志。它们存储在系统范围（非目标特定）变量中，用于使用的语言（C++ 为 `CXX`）和构建配置（`DEBUG` 或 `RELEASE`）：

- `CMAKE_CXX_FLAGS_DEBUG` 等于 `-g`
- `CMAKE_CXX_FLAGS_RELEASE` 等于 `-O3 -DNDEBUG`

调试配置不启用优化，而发布配置直接使用 -O3。可以使用 `set()` 命令直接更改它们，或者只添加一个目标编译选项，这将覆盖此默认行为。其他两个标志 (`-g`, `-DNDEBUG`) 与调试有关。

诸如 `CMAKE_<LANG>_FLAGS_<CONFIG>` 这样的变量是全局的——适用于所有目标。建议通过属性和命令（如 `target_compile_options()`）配置目标，而不是依赖全局变量。这样，可以更细致地控制目标。

通过选择带有 `-O<` 级别的 `>` 优化级别，间接的设置了一系列标志，每个标志控制特定的优化行为。然后，可以通过更多标志来微调优化：

- 使用 `-f` 选项启用 `-finline-functions`。
- 使用 `-fno` 选项禁用 `-fno-inline-functions`。

这些标志中的一些值得更好地理解，因为会影响到程序的工作方式，以及如何调试。

7.4.2. 函数内联

可以通过在类声明块内定义函数或显式使用 `inline` 关键字，鼓励编译器内联某些函数：

```
1 struct X {  
2     void im_inlined(){ cout << "hi\n"; };  
3     void me_too();  
4 };  
5 inline void X::me_too() { cout << "bye\n"; };
```

是否内联一个函数最终由编译器决定。如果启用了内联，并且函数在单个地方使用（或者相对较小的函数使用），则很可能会发生内联。

函数内联是一种有趣的优化技术，通过从目标函数中提取代码，并将其嵌入到函数调用的所有位置来操作。这个过程替换了原始调用，并节省了宝贵的 CPU 周期。

使用我刚刚定义的类：

```
1 int main() {  
2     X x;  
3     x.im_inlined();  
4     x.me_too();  
5     return 0;  
6 }
```

如果没有内联，代码将在 `main()` 中执行，直到方法调用。然后，将为 `im_inlined()` 创建一个新的帧，在单独的作用域中执行，然后返回到 `main()`。对于 `me_too()` 方法也会发生同样的情况。

然而，当发生内联时，编译器将替换调用，如下所示：

```
1 int main() {  
2     X x;  
3     cout << "hi\n";
```

```
4     cout << "bye\n";
5
6 }
```

这并不是一个精确的表现，因为内联发生在汇编或机器代码级别（而不是源代码级别），但它确实提供了一个大致的表现。

编译器使用内联来节省时间，绕过了创建和拆除新调用的需要，以及查找下一个要执行的指令的地址（并返回）的需求，并由于在接近的位置，从而增强了指令缓存。

然而，内联确实带来了一些显著的副作用。如果一个函数多次使用，必须复制到所有位置，导致文件大小增大和内存使用增加，特别是在为内存有限的低端设备开发软件时，需要注意。

此外，内联对调试产生了严重影响。内联代码不再位于原始行号，使得跟踪变得更加困难，有时甚至不可能。这就是为什么在内联的函数中放置的调试器断点，永远不会命中（尽管代码仍然以某种方式执行）。为了绕过这个问题，需要为调试构建禁用内联（以不测试确切发布构建版本为代价）。

可以通过为目标指定-00（零级别），或直接处理负责内联的标志来实现这一点：

- -finline-functions-called-once：适用于 GCC。
- -finline-functions：适用于 Clang 和 GCC。
- -finline-hint-functions：适用于 Clang。

可以使用-fno-inline…明确禁用内联，为了详细信息，建议参考特定编译器的文档版本。

7.4.3. 循环展开

循环展开是一种优化技术，这种策略旨在将循环转换为一系列完成相同结果的语句。这种方法以程序的小尺寸换取执行速度，消除了循环控制指令、指针算术和循环结束检查。

看一下示例：

```
1 void func() {
2     for(int i = 0; i < 3; i++)
3         cout << "hello\n";
4 }
```

之前的代码将变成如下形式：

```
1 void func() {
2     cout << "hello\n";
3     cout << "hello\n";
4     cout << "hello\n";
5 }
```

结果保持不变，但不再需要分配 i 变量，也不需要对其进行递增或与值 3 进行比较三次。如果在程序的生命周期中足够多次调用 func()，即使是这样一个短小和简单的函数，其展开也会产生显著的差异。

重要的是要理解两个限制因素。首先，循环展开只有在编译器知道或能够准确估计迭代次数时才有效。其次，循环展开可能会在现代 CPU 上产生不希望的结果，因为代码大小的增加可能会阻碍缓存命中。

每个编译器提供了这个标志的略有不同的版本：

- `-floop-unroll`: GCC 版本。
- `-funroll-loops`: Clang 版本。

如果不确定，请广泛测试这个标志是否影响特定程序，并明确地启用或禁用。GCC 上，与隐式启用的`-O3`一起，作为`-floop-unroll-and-jam`标志的一部分启用。

7.4.4. 循环向量化

称为单指令多数据 (SIMD) 的机制在 20 世纪 60 年代初为实现并行而开发。正如其名称，旨在同时对多个数据执行相同的操作。通过以下示例来了解一下：

```
1 int a[128];
2 int b[128];
3 // initialize b
4 for (i = 0; i<128; i++)
5     a[i] = b[i] + 5;
```

这样的代码会循环 128 次，但如果 CPU 能力强大，代码的执行可以通过同时计算两个或更多的数组元素而显著加速。这是由于连续元素之间不存在依赖关系，以及数组之间的数据重叠。聪明的编译器可以将前一个循环转换为类似以下形式（这发生在汇编级别）：

```
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`级别启用这样的循环向量化，Clang 默认启用。两个编译器都提供不同的标志来启用/禁用特定情况下的向量化：

- `-ftree-vectorize -ftree-slp-vectorize`: GCC 中启用向量化
- `-fno-vectorize -fno-slp-vectorize`: Clang 中禁用向量化

向量化的效率来自于 CPU 制造商提供的特殊指令的利用，而不是仅仅将循环的原始形式替换为展开版本。

优化器在提高程序的运行时性能方面起着关键作用。通过有效地运用其策略，可以得到更多的价值。效率不仅在于编码完成之后，而且在软件开发过程中也至关重要。如果编译时间过长，可以通过更好地管理过程来改善。

7.5. 管理编译过程

作为程序员和构建工程师，还必须考虑编译的其他方面，比如完成编译所需的时间，以及解决方案构建过程中识别和纠正错误的速度。

7.5.1. 减少编译时间

需要频繁重新编译的项目（可能每小时多次）中，确保编译过程尽可能快是至关重要的。这不仅影响代码-编译-测试循环的效率，还会影响专注度和工作流程。

由于单独的翻译单元，C++ 在管理编译时间方面已经做得很好了。CMake 将仅重新编译受最近更改影响的源文件。如果需要进一步改进，可以使用一些技术：预编译头文件和统一构建。

预编译头文件

头文件 (.h) 在编译开始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。此外，如果多个翻译文件使用相同的共享头文件，每次包含时都必须编译。这种方式效率低下，但长期以来一直是标准做法。

自从 3.16 版本以来，CMake 提供了一个命令来启用头文件预编译。这允许编译器单独处理头文件，从而加快编译过程。以下是该命令的语法：

```
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。我们可以通过选择 PUBLIC 或 INTERFACE 关键字使用传播属性将头文件与任何依赖目标共享，但不应该对使用 install() 命令导出的目标执行此操作。

Note

使用第 6 章中的 \$<BUILD_INTERFACE:> 生成器表达式，避免预编译头文件出现在安装时。然而，仍然可能通过 export() 命令从构建树导出的目标添加。如果现在这似乎令人困惑 - 不要担心，这将在第 14 章中进行介绍。

CMake 将所有头文件的名称放入 cmake_pch.h 或 cmake_pch.hxx 文件中，然后将其预编译为具有 .pch、.gch 或 .pchi 扩展名的编译器特定二进制文件。

可以在列表文件中这样使用：

ch07/06-precompile/CMakeLists.txt

```
1 add_executable(precopiled hello.cpp)
2 target_precompile_headers(precopiled PRIVATE <iostream>)
```

也可以在相应的源文件中使用：

ch07/06-precompile/hello.cpp

```
1 int main() {  
2     std::cout << "hello world" << std::endl;  
3 }
```

main.cpp 文件中，不需要包含 cmake_pch.h 或其他头文件 - CMake 会使用编译器特定的命令行选项包含。

前面的示例中，使用了一个内置头文件；然而，可以轻松地添加自己的带有类或函数定义的头文件。使用以下两种形式之一引用头文件：

- header.h（直接路径）将解析为相对于当前源目录的路径，并将使用绝对路径包含。

["header.h"]（双括号和引号）路径将根据目标的 INCLUDE_DIRECTORIES 属性进行扫描，该属性可以使用 target_include_directories() 进行配置。

一些在线参考资料可能不鼓励预编译不是标准库（如）一部分的头文件，或者根本不使用预编译头文件。这是因为更改列表或编辑自定义头文件，将导致目标中的所有翻译单元重新编译。使用 CMake，这个问题并不严重，特别是已经正确地结构化项目（具有相对较小的目标，专注于狭窄的领域）。每个目标都有一个单独的预编译头文件，这限制了头文件更改的影响。

如果头文件相对稳定，可能会决定在目标中重用预编译头文件。为此，CMake 提供了一个方便的命令：

```
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依赖关系。使用此方法，使用目标不能再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。

注意，特别是头文件使用双括号格式 ([["header.h"]])。两个目标都需要适当地设置它们的包含路径，以确保编译器能找到这些头文件。

统一构建

CMake 3.16 引入了另一个编译时间优化功能——统一构建，也称为统一构建或超级构建。统一构建通过利用 #include 指令将多个实现源文件组合在一起。这带来了一些有趣的后果，其中一些有益，而其他一些可能有害。

最明显的优势是在 CMake 创建统一构建文件时，会避免在不同翻译单元中重新编译头文件：

```
1 #include "source_a.cpp"  
2 #include "source_b.cpp"
```

当两个源文件都包含一个 #include "header.h" 行时，引用的文件将只解析一次。虽然这不如预编译头文件那样精细，但它是一种替代方案。

这种构建类型的第二个好处是，优化器现在可以在更大的范围内进行操作，并对捆绑的所有源文件中的跨过程调用进行优化。这与第 4 章中讨论的链接时间优化类似。

然而，这些好处是有代价的。随着减少了对象文件和处理步骤的数量，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量。编译器并不是特别擅长多线程编译，通常不需要——构建系统通常会启动许多编译任务，以便在不同的线程上同时执行所有文件。将所有文件组合在一起会复杂化这个问题，因为 CMake 现在可以并行编译的文件更少了。

统一构建中，还需要考虑一些 C++ 语义上的影响，这些影响可能并不那么明显——匿名命名空间隐藏跨文件的符号现在会限定在统一文件中，而不是单个翻译单元。同样，静态全局变量、函数和宏定义也会发生这种情况。这可能导致名称冲突，或者执行错误的函数重载。

重新编译时，统一构建的性能并不理想，因为会编译许多不必要的文件。代码需要尽可能快地编译所有文件时表现最佳。在 Qt Creator (一个流行的 GUI 库) 上进行的测试显示，可以期待性能提升在 20% 到 50% 之间 (取决于使用的编译器)。

要启用统一构建，有两种选择：

- 将 CMAKE_ UNITY _BUILD 变量设置为 true——将在定义的所有目标上初始化 UNITY _BUILD 属性。
- 手动将 UNITY _BUILD 目标属性设置为 true，适用于应该使用统一构建的所有目标。

第二个选项通过以下方式实现：

```
set_target_properties(<target1> <target2> ...
    PROPERTIES UNITY_BUILD true)
```

当然，手动设置许多目标的这些属性会增加工作量和维护成本，但可能需要这样做以在更精细的层面上控制这个设置。

默认情况下，CMake 将创建包含八个源文件的构建，这是通过目标的目标属性 UNITY _BUILD _BATCH _SIZE 指定的 (创建目标时从 CMAKE_ UNITY _BUILD _BATCH _SIZE 变量复制)，可以更改目标属性或默认变量。

从版本 3.18 开始，可以显式定义文件应该如何分组，并为它们命名。为此，请将目标的 UNITY _BUILD _MODE 属性更改为 GROUP (默认是 BATCH)。然后，通过设置它们的 UNITY _GROUP 属性为选择的名称分组源文件：

```
set_property(SOURCE <src1> <src2> PROPERTY UNITY_GROUP "GroupA")
```

CMake 将忽略 UNITY _BUILD _BATCH _SIZE，并将所有来自该组的文件添加到一个统一的构建中。

Make 的文档建议不要为公共项目默认启用统一构建。建议应用程序的最终用户通过提供 -DCMAKE_ UNITY _BUILD 命令行参数来决定是否想要统一构建。如果代码编写方式导致统一构建出现问题，应该明确地将目标的属性设置为 false。然而，对于内部使用的代码 (如在公司内部或私人项目中)，可以启用这个特性。

这些是为了 CMake 减少编译时间。编程的其他方面往往也会花费我们很多时间——其中耗时最多的就是调试。

7.5.2. 发现错误

作为开发者，我们花大量时间寻找 bug。识别错误并纠正它们的过程往往令人沮丧，尤其是当它需要长时间时。当我们没有工具来应对这些挑战性的情况时，难度会进一步增加。因此，必须要非常关注环境设置，以便简化这个过程，使其尽可能容易和可承受。一种实现这一目标的方法是通过配置编译器使用 `target_compile_options()`。那么，哪些编译选项可以帮助我们呢？

配置错误和警告

软件开发中有许多令人压力大的事情——半夜修复关键 bug，在大系统中处理高可见性、成本高昂的失败，以及处理恼人的编译错误。有些错误很难理解，而另一些则需要艰苦努力来修复。简化工作并减少失败的机会的探索中，会发现许多关于如何配置编译器警告的建议。

一个很好的建议是默认为所有构建启用 `-Werror` 标志。这个标志的功能表面上很简单——将所有的警告视为错误，阻止代码编译，直到解决。虽然这可能看起来是一个有益的方法，但很少是这样。

警告没有分类为错误是有原因的：是为了警告你。决定如何处理这些警告取决于你。当在实验或原型解决方案时，忽略一个警告的权限往往无价。

另一方面，如果代码是完美的、没有警告，那么允许未来的修改破坏这个完美的状态似乎是有遗憾的。开启它并让它保持在那里有什么害处？至少在编译器升级之前，似乎没有什么害处。新版本的编译器往往对过时的特性更严格，或者更擅长提供改进建议。虽然当警告保持为警告时是有益的，但当需要快速修复与新警告无关的问题时，可能导致未更改的代码出现意外的构建失败。

那么，什么时候允许启用所有可能的警告呢？答案是当你创建一个公共库时。在这种情况下，预先阻止问题，指出代码在比开发环境更严格的环境中可能会行为不当。如果选择启用这个设置，确保更新对新编译器版本引入的警告。此外，明确管理这个更新过程也很重要，这可以与任何代码更改分开处理。

否则，让警告保持原样，专注于错误。如果觉得有必要追求完美，可以使用 `-Wpedantic` 标志。这个特定的标志启用了由严格的 ISO C 和 ISO C++ 标准要求的警告。然而，这个标志并不确认符合标准，只是标识需要诊断消息的非 ISO 实践。

更宽松、更接地气的程序员会满足于 `-Wall`，可以选择与 `-Wextra` 结合，增加一点复杂性，这应该就足够了。这些警员认为是真正有用的，当时间允许时，应该在代码中处理它们。

根据项目类型，可能还有其他一些警告标志有帮助。我建议阅读编译器的手册，看看有哪些选项可用。

调试构建

偶尔，编译会失败。这通常发生在我们尝试重构大量代码或清理构建系统时。有时，问题可以轻松解决；然而，有些更复杂的问题需要深入调查配置步骤。我们已经在第 1 章中讨论了如何打印更详细的 CMake 输出，但如何分析在每个阶段实际发生了什么？

调试各个阶段

`-save-temps`，可以传递给 GCC 和 Clang 编译器，允许调试编译的各个阶段。这个标志将指示编译器将某些编译阶段的输出存储在文件中，而不是内存中。

```
1 add_executable(debug hello.cpp)
2 target_compile_options(debug PRIVATE -save-temps=obj)
```

启用这个选项将产生两个文件（.ii 和.s）每个翻译单元。

第一个，<build-tree>/CMakeFiles/<target>.dir/<source>.ii，存储了预处理阶段的输出，其中包含注释，解释了源代码的每个部分来自哪里：

```
# 1 "/root/examples/ch07/06-debug/hello.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# / / / ... removed for brevity ... / /
# 252 "/usr/include/x86_64-linux-gnu/c++/9/bits/c++config.h" 3
namespace std
{
    typedef long unsigned int size_t;
    typedef long int ptrdiff_t;
    typedef decltype(nullptr) nullptr_t;
}
```

第二个，<build-tree>/CMakeFiles/<target>.dir/<source>.s，包含了语言分析阶段的输出，准备进入汇编阶段：

```
.file "hello.cpp"
.text
.section .rodata
.type _ZStL19piecewise_construct, @object
.size _ZStL19piecewise_construct, 1
_ZStL19piecewise_construct:
.zero 1
.local _ZStL8__ioinit
.comm _ZStL8__ioinit,1,1
.LCO:
.string "hello world"
.text
.globl main
.type main, @function
main:
(...)
```

根据问题的类型，通常可以揭示实际的问题。例如，预处理器的输出可以识别 bug，如错误的 include 路径（可能提供错误的库版本），或者定义错误导致的错误的 #ifdef 评估。

同时，语言分析的输出对于针对特定处理器和解决关键优化问题特别有益。

调试头文件包含问题

调试错误包含的文件可能是一项具有挑战性的任务。我应该知道——在第一家公司工作时，我需要将整个代码库从一个构建系统移植到另一个构建系统。如果发现自己处于需要精确理解请求头文件包含路径的情况，可以考虑使用-H 编译选项：

ch07/07-debug/CMakeLists.txt

```
1 add_executable(debug hello.cpp)
2 target_compile_options(debug PRIVATE -H)
```

产生的输出将类似于以下内容：

```
[ 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
```

对象文件名称之后，输出中的每一行都包含一个头文件的路径。这个例子中，行首的一个点表示顶级包含（包含指令在 `hello.cpp` 中）。两个点表示该文件被后续文件（`iostream`）包含。每个额外的点表示嵌套的另一个级别。

输出的末尾，可能会找到一些代码改进的建议：

```
Multiple include guards may be useful for:
/usr/include/c++/9/clocale
/usr/include/c++/9/cstdio
/usr/include/c++/9/cstdlib
```

虽然不需要处理标准库中的问题，但可能会看到一些自己的头文件列出。在这种情况下，再考虑进行修改。

为调试器提供信息

机器代码是一系列指令和数据的神秘列表，这些指令和数据以二进制格式编码，不传达任何更大的意义或目标。这是因为 CPU 不在乎程序的目标是什么，或者所有指令的含义是什么。唯一的要求是代码的正确性。编译器会将所有这些内容转换为 CPU 指令的数字标识符，并在需要时存储数据以初始化内存，并提供成千上万的内存地址。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、函数签名或其他程序员关心的细节。这是编译器的默认输出——原始而简单。

这样做是为了节省空间，并执行时避免过多的开销。巧合的是，我们也以某种方式保护了应用程序免受逆向工程的影响。是的，可以理解每个 CPU 指令做什么，而无需源代码（将这个值复制到那个寄存器）。

如果你是一个特别有动力的人，可以使用一个叫做反汇编器的工具，并且凭借大量的知识（以及一些运气），将能够解码可能发生的事情。但这种方法并不实用，因为反汇编代码没有原始符号，这使得很难和慢地弄清楚它们去了哪里。

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，并与编译后的代码和原始代码之间的引用映射一起。然后，可以将调试器附加到一个正在运行的程序上，并查看在任何给定的时刻执行的是哪一行源代码。当编写新功能或更正错误时，这必不可少。

这两个用例是存在两种构建配置的原因：Debug 和 Release。CMake 会默认向编译器提供一些标志来管理这个过程，首先存储在全局变量中：

- CMAKE_CXX_FLAGS_DEBUG 包含 -g
- CMAKE_CXX_FLAGS_RELEASE 包含 -DNDEBUG

-g 标志的意思是“添加调试信息”，以操作系统的原生格式提供：stabs、COFF、XCOFF 或 DWARF。这些格式随后可以被像 gdb（GNU 调试器）这样的调试器访问。通常，这对像 CLion 这样的 IDE 来说已经足够了（后台使用 gdb）。其他情况下，请参考提供的调试器手册，并了解选择的编译器的适当标志。

对于 Release 配置，CMake 将添加-DNDEBUG 标志。这是一个预处理器定义——不是调试构建”这个选项会禁用一些面向调试的宏。其中之一是 assert，在 <assert.h> 头文件中可用。如果决定在生产代码中使用断言，其将不起作用：

```
1 int main(void)
2 {
3     assert(false); // bled
4     std::cout << "This shouldn't run. \n";
5     return 0;
6 }
```

在 Release 配置中，assert(false) 调用不会有任何效果，但在 Debug 中它会停止执行。如果在实践断言性编程，仍需要在 Release 构建中使用 assert()，可以改变 CMake 提供的默认设置（从 CMAKE_CXX_FLAGS_RELEASE 中移除 NDEBUG），或者通过在包含头文件之前取消定义宏来实现硬编码覆盖：

```
1 #undef NDEBUG
2 #include <assert.h>
```

更多信息请参阅 assert 参考：<https://en.cppreference.com/w/c/error/assert>。

如果断言可以在编译时完成，可以考虑用 C++11 引入的 static_assert() 替换 assert()，因为这个函数不像 assert() 那样受到 #ifndef(NDEBUG) 预处理器指令的保护。

通过这些，我们已经了解了如何管理编译过程。

7.6. 总结

我们已经完成了又一章的学习！毫无疑问，编译是一个复杂的过程。考虑到所有的边缘情况和特定要求，如果没有一个强大的工具，管理起来可能会很困难。幸运的是，CMake 在这里为我们提供了支持。

那么，到目前为止我们学到了什么？从讨论编译是什么，以及它如何适应在操作系统中构建和运行应用程序开始。然后，检查了编译的各个阶段，以及管理它们的内部工具。这对于解决未来可能遇到的问题价值连城。

接下来，探讨了如何使用 CMake 来验证宿主上可用的编译器，是否满足代码构建的必要要求。正如已经建立的，对于解决方案的用户来说，最好看到一个友好的消息提示他们升级，而不是一个过时的编译器输出的神秘错误。后者无法处理语言的新特性，这无疑是显著更好的体验。

我们简要讨论了如何向已定义的目标添加源文件，然后继续讨论预处理的配置。这是一个相当重要的主题，因为这一阶段将所有代码片段汇集在一起，并确定哪些部分将忽略。我们讨论了提供文件路径和添加自定义定义的方法，既可以单独添加，也可以批量添加（以及一些使用案例）。然后，讨论了优化器，探索了所有一般优化级别，以及它们隐式添加的标志。还详细讨论了其中的一些—`finline`、`floop-unroll` 和 `ftree-vectorize`。

最后，研究如何管理编译的可行性。在这里解决了两个主要方面—减少编译时间（这进而有助于保持程序员的专注）和查找错误。后者对于识别什么是错误的，以及为什么错误极为重要。正确设置工具并理解为什么会发生这些事情，对于确保代码质量（以及保持心理健康）做出了巨大贡献。

下一章中，我们将了解链接，以及构建库，并在我们的项目中使用构建的库。

7.7. 扩展阅读

- CMake 支持的编译特性和编译器：

<https://cmake.org/cmake/help/latest/manual/cmake-compile-features.7.html#supported-compilers>

- 管理目标的源文件：

<https://stackoverflow.com/questions/32411963/why-is-cmake-file-glob-evil>, https://cmake.org/cmake/help/latest/command/target_sources.html

- `#include` 关键字：

<https://en.cppreference.com/w/cpp/preprocessor/include>

- 提供包含文件的路径：

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

- 头文件的预编译：

https://cmake.org/cmake/help/latest/command/target_precompile_headers.html

- 统一构建：

https://cmake.org/cmake/help/latest/prop_tgt/UNITY_BUILD.html

- 预编译头文件的统一构建:

[https://www.qt.io/blog/2019/08/01/precompiled-headers-and-unity-jumbo-builds
-in-upcoming-cmake](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>

- 为什么使用库文件而不是对象文件:

<https://stackoverflow.com/questions/23615282/object-files-vs-libraryfiles-and-why>

- 关注点分离:

<https://nalexn.github.io/separation-of-concerns/>

第 8 章 链接可执行文件和库

当我们成功地将源代码编译成二进制文件，貌似作为构建工程师的角色就完成了。然而，事实并非如此。虽然二进制文件确实包含了 CPU 执行所需的所有代码，但这些代码以复杂的方式分布在多个文件中。我们不希望 CPU 在不同的文件中搜索单个代码片段。相反，是要将这些单独的单元合并成一个文件。为了实现这一点，需要使用一个称为“链接”的过程。

CMake 的链接命令并不多，其中 `target_link_libraries()` 其中之一。那为什么还要为单个命令写一整个章节呢？在计算机科学中，几乎没有什么东西是简单的，链接也不例外：为了得到正确的结果，需要对整体进行了解——需要知道链接器如何工作，并且要掌握基础知识。将讨论目标文件的内部结构，如何工作的重定位和引用解析机制，以及其用途。我们将讨论最终可执行文件与组件的不同之处，以及系统在将程序加载到内存时，如何构建进程映像。

然后，介绍各种类型的库：静态库、共享库和共享模块。它们都称为“库”，但又非常不同。创建链接良好的可执行文件，依赖于正确的配置和处理特定细节，例如：位置无关代码（PIC）。

了解链接的另一个难题——唯一定义规则（ODR）。准确地定义符号的数量至关重要。管理重复的符号很具有挑战性，尤其是对于共享库。此外，将探讨为什么链接器有时无法定位外部符号，即使可执行文件已正确链接到相关库。

最后，将了解如何高效地使用链接器，为解决方案在特定框架内进行测试做好准备。

本章中，包含以下内容：

- 链接的基本知识
- 构建不同类型的库
- 解决 ODR 问题
- 链接和未解析符号的顺序
- 为测试分离 `main()`

8.1. 示例下载

可以在 GitHub 上找到本章中存在的代码文件，网址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch08>。

要构建本书中提供的示例，请使用推荐的命令：

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

请确保用适当的路径替换 `<build tree>` 和 `<source tree>` 占位符。作为提醒：`<build tree>` 是目标/输出目录的路径，`<source tree>` 是源码所在的路径。

8.2. 掌握正确链接的基本知识

在第 7 章中，我们讨论了 C++ 程序的生命周期，它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后，需要将它们组合成一个可执行文件。编译产生的对象文

件不能直接由处理器执行，为什么会这样呢？

为了回答这个问题，来了解对象文件是广泛使用的可执行和链接格式（ELF）的一种变体，这在类 Unix 系统和其他许多系统中都很常见。像 Windows 或 macOS 这样的系统有自己的格式，但我们将专注于 ELF 来解释这个原理。图 8.1 显示了编译器如何构建这些文件的结构：

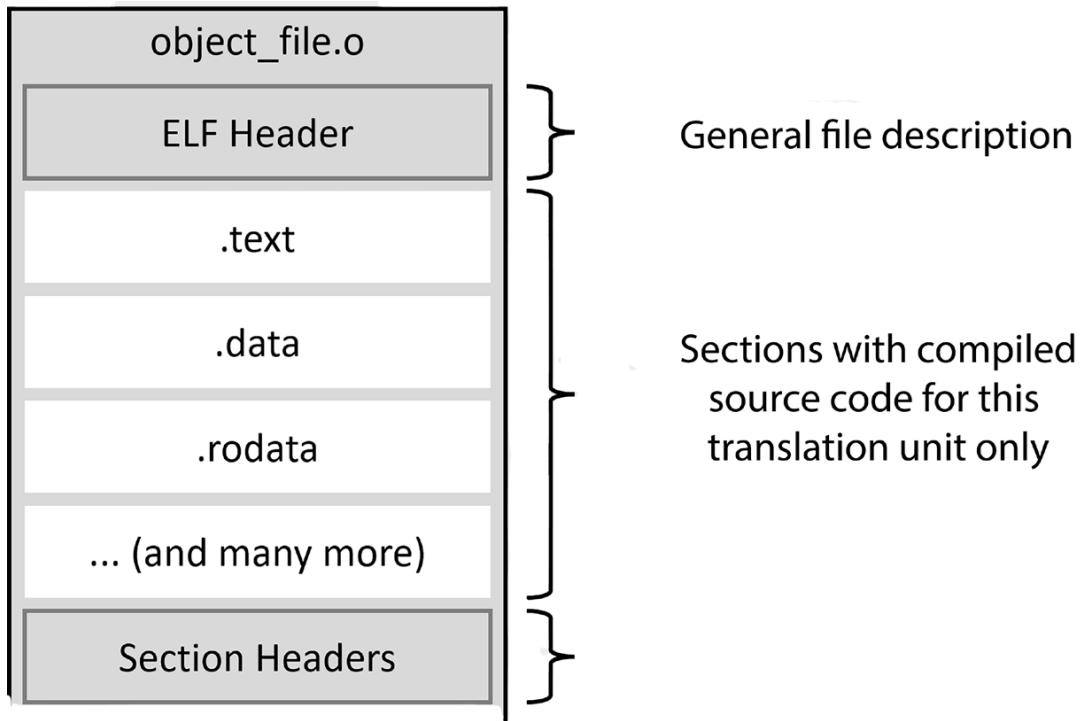


图 8.1：对象文件的结构

编译器将为每个翻译单元（每个 .cpp 文件）准备一个对象文件，这些文件将用于构建程序的内存映像。对象文件包含以下内容：

- ELF 头部，标识目标操作系统（OS）、文件类型、目标指令集架构，以及 ELF 文件中两个头部表的位置和大小的详细信息：程序头部表（在对象文件中不存在）和节头部表。
- 按类型分组信息的二进制节。
- 节头部表，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，以及其他信息。用于了解这个文件中有哪些节，以及它们的位置，就像目录一样。

当编译器处理源代码时，将收集的信息分类到不同的节中。这些节构成了 ELF 文件的核心，位于 ELF 头部和节头部之间。以下是一些例子：

- .text 节包含所有指定给处理器执行的机器代码指令。
- .data 节保存初始化的全局和静态变量的值。
- .bss 节为未初始化的全局和静态变量保留空间，这些变量在程序开始时初始化为零。
- .rodata 节保存常量的值，使其成为一个只读数据段。
- .strtab 节是一个字符串表，包含常量字符串，例如：来自基本 hello.cpp 示例的“Hello World”。
- .shstrtab 节是一个字符串表，保存所有其他节的名字。

这些反映了最终放入 RAM 运行的可执行文件。然而，不能简单地将对象文件连接在一起，然后将结果文件加载到内存中。不加考虑的合并会导致一系列复杂问题，例如：会浪费空间和时间，消耗过多的 RAM 页面。将指令和数据传输到 CPU 缓存也会变得笨拙，整个系统将不得不处理增加的复杂性，浪费宝贵的周期。执行过程会在无数的 .text、.data 和其他节之间跳转。

我们将采取更有组织的方法：每个对象文件的节将与其他对象文件中相同类型的节组合在一起，这个过程称为重定位，这就是为什么对象文件的 ELF 文件类型会标记为“可重定位”。但重定位不仅仅是组装匹配的节，还涉及更新文件内的内部引用，例如：变量、函数、符号表索引和字符串表索引的地址。每个这些值都是其自身对象文件的本地值，从零开始编号。在合并文件时，必须调整这些值，以确保它们引用合并文件中的正确地址。

图 8.2 显示了重定位的操作过程——.text 节已经重定位，.data 节正在从所有链接的文件中组装，.rodata 和.strtab 节将遵循相同的流程（简单起见，图中不包含头部）：

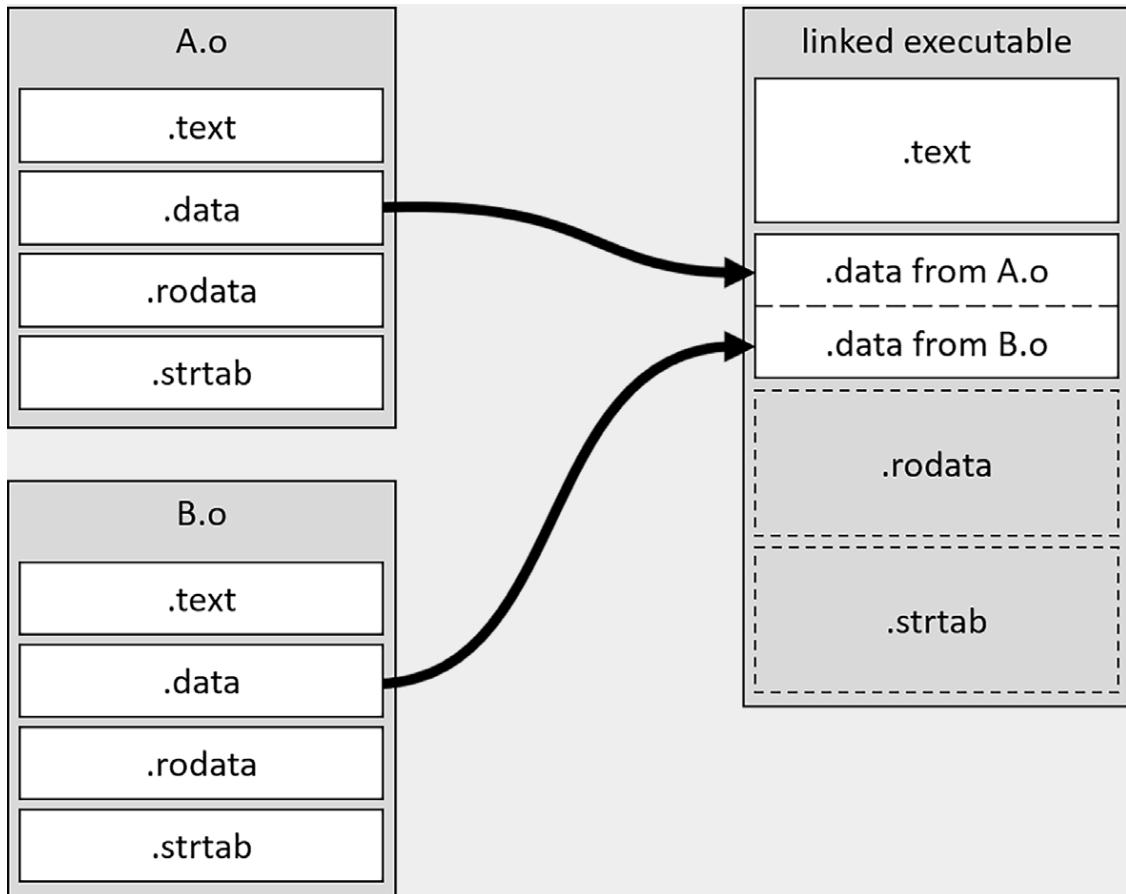


图 8.2: .data 节的重定位

接下来，链接器需要解析引用。当一个翻译单元的代码引用另一个单元中定义的符号时，无论是通过包含其头文件，还是使用 `extern` 关键字，编译器都会承认这个声明。假设稍后会提供定义。链接器的主要角色是收集这些未解决的外部符号引用，然后识别并在合并的可执行文件中填充其所属的地址。图 8.3 显示了这个引用解析过程的简单示例：

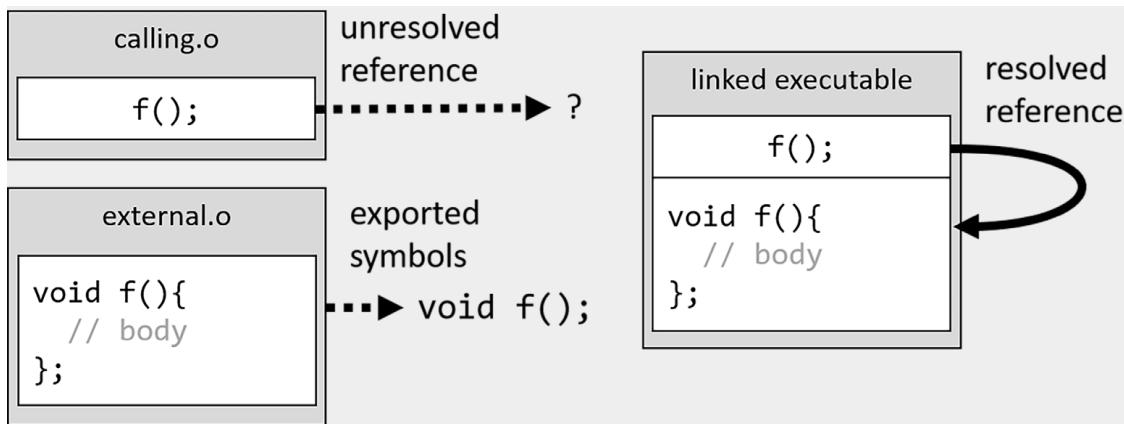


图 8.3: 引用解析

如果开发者不了解这是如何工作的，这部分链接可能会成为问题的来源。我们可能会得到无法找到其对应外部符号的未解决引用，或者相反：提供了太多的定义，而链接器不知道选择哪一个。

最终的可执行文件与对象文件非常相似，包含了解决引用的重定位节、节头部表，当然还有描述整个文件的 ELF 头部。主要的区别在于存在程序头部，如下图所示：

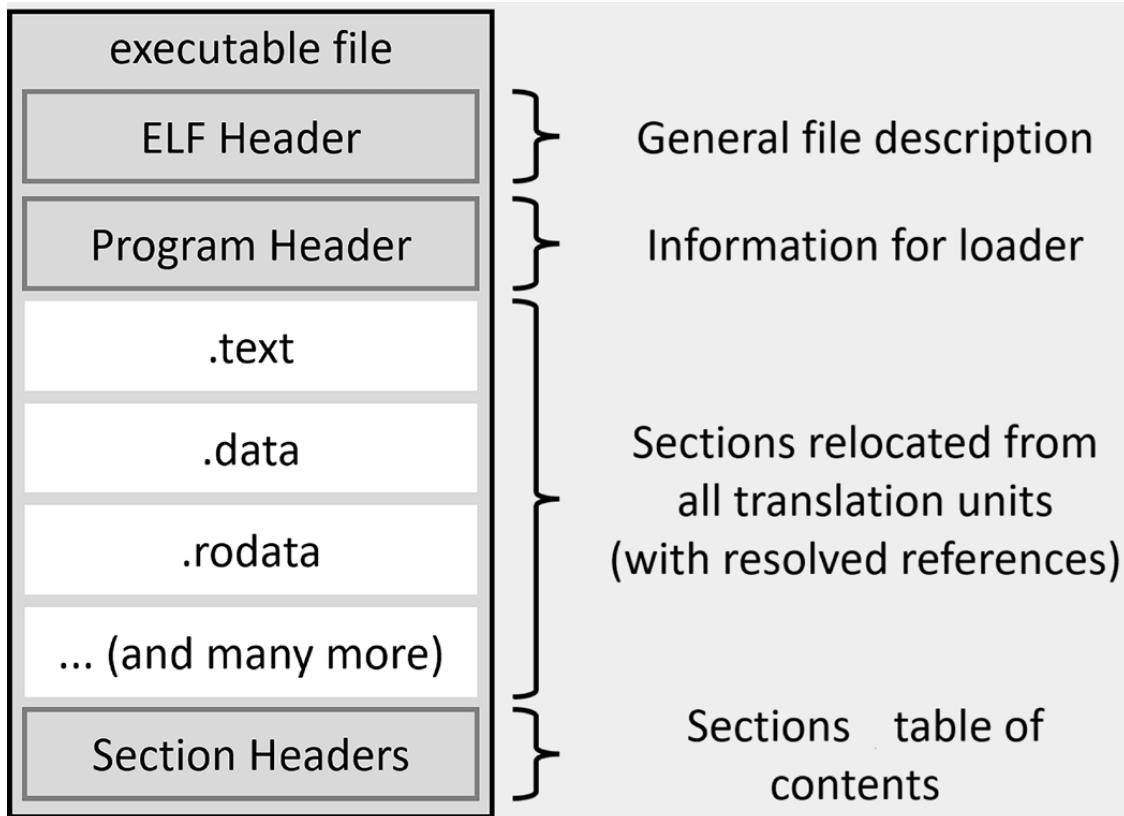


图 8.4: ELF 中可执行文件的结构

程序头部位于 ELF 头部之后。操作系统的加载器将读取这个程序头部来设置程序，配置内存布局，并创建进程映像。将复制程序头部中指定哪些节的条目，按照什么顺序，以及复制到虚拟内存中的哪些地址中。还包含关于访问控制标志（读、写或执行）的信息，以及一些其他信息。创建的进程中的每个命名节，将由一个内存片段表示。

对象文件也可以打包在库中，库是一个中间产品，可以用于最终的可执行文件或另一个库。

现在了解了链接的工作原理，让我们继续下一部分，在那里将讨论三种不同类型的库。

8.3. 构建不同类型的库

编译源码后，最好避免同一平台的代码重新编译，甚至将编译输出与外部项目共享。可以将最初生成的单个对象文件分发出去，但这会带来挑战。分发多个文件，并将它们逐个集成到构建系统中可能很麻烦，特别是在处理大量文件时。更有效的方法是将所有对象文件合并为一个单元以供共享。CMake 大大简化了这个任务。可以使用简单的 `add_library()` 命令（与 `target_link_libraries()` 命令配对）生成这些库。

按照约定，所有库都有一个公共前缀（即 `lib`），并使用特定于系统的扩展名，来表示它们是什么类型的库：

- 类 Unix 系统中，静态库具有`.a` 扩展名，在 Windows 上为`.lib`。
- 某些类 Unix 系统（如 Linux）上，共享库（和模块）具有`.so` 扩展名，而在其他系统（如 macOS）上为`.dylib`。在 Windows 上，扩展名为`.dll`。
- 共享模块通常使用与共享库相同的扩展名，但不总是如此。在 macOS 上，可以使用`.so`，特别是当模块是从另一个 Unix 平台移植过来时。

构建库（静态、共享或共享模块）的过程通常称为“链接”，在 `ch08/01-libraries` 项目的构建输出中看到：

```
[ 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
```

然而，前面提到的库并非需要使用链接器来创建。对于某些库，该过程可能会跳过某些步骤，如重定位和引用解析。

8.3.1. 静态库

静态库本质上是存储在存档中的原始对象文件的集合，会通过索引来加速链接过程。在类 Unix 系统中，这样的存档可以通过 `ar` 工具创建，并使用 `ranlib` 进行索引。

构建过程中，只有静态库中必要的符号，才会导入最终的可执行文件中，优化其大小和内存使用。这种选择性集成确保了可执行文件是自包含的，消除了在运行时对外部文件的需求。

要创建静态库，可以简单地使用前面章节中已经看到的命令：

```
add_library(<name> [<source>...])
```

这个简写代码默认会生成一个静态库，可以通过将 `BUILD_SHARED_LIBS` 变量设置为 `ON` 来覆盖。如果想要构建一个静态库，可以提供一个显式关键字：

```
add_library(<name> STATIC [<source>...])
```

特别是当在同一台机器上运行的多个应用程序共享编译后的代码时，使用静态库可能并不总是理想的选择。

8.3.2. 共享库

共享库与静态库有显著不同。是使用链接器构建的，链接器完成了链接的两个阶段。这产生了一个包含节头、节和节头表的完整文件，如图 8.1 所示。

共享库，通常称为共享对象，可以使用多个不同的应用程序同时使用。当第一个程序使用共享库时，操作系统会将该库的一个实例加载到内存中。随后，操作系统为其他程序提供相同的地址，这要归功于复杂的虚拟内存机制。然而，对于每个使用该库的进程，库的.data 和.bss 段是分别实例化的。这确保了每个进程，都可以调整其变量，而不影响其他进程。

感谢这种方法，系统整体内存使用得到了优化。如果使用主流认可的库，可能不需要将其包含在程序中，它可能已经存在于目标机器上。但若没有预安装，用户需要在运行应用程序之前手动安装。如果安装的库版本与预期版本不同，这可能导致潜在问题，这类问题称为“依赖地狱”。

我们可以通过显式使用 `SHARED` 关键字来构建共享库：

```
add_library(<name> SHARED [<source>...])
```

由于共享库在程序初始化期间加载到操作系统的内存中，因此执行程序与磁盘上的实际库文件之间没有关联。相反，链接是间接完成的。类 Unix 系统中，这是通过共享对象名 (SONAME) 实现的，可以将其理解为库的“逻辑名称”。

这允许在库版本控制上具有灵活性，并确保对库的向后兼容性更改，不会破坏依赖的应用程序。

可以使用生成器表达式，查询产生的 SONAME 文件的一些路径属性（确保将 `target` 替换为目标名称）：

- `$<TARGET_SONAME_FILE:target>` 返回完整路径 (`.so.3`)。
- `$<TARGET_SONAME_FILE_NAME:target>` 只返回文件名。
- `$<TARGET_SONAME_FILE_DIR:target>` 只返回相应文件夹路径。

本书后面将介绍的高级场景中非常有用，包括：

- 打包和安装过程中正确使用生成的库。
- 编写自定义的 CMake 规则进行依赖管理。
- 测试过程中利用 SONAME。
- 构建后命令中复制或重命名生成的库。

可能对其他特定于操作系统的工件有类似的需求，CMake 提供了两个生成器表达式，提供了与 SONAME 相同的后缀。对于 Windows：

- `$<TARGET_LINKER_FILE:target>` 返回与生成的动态链接库 (DLL) 关联的.lib 导入库的完整路径。请注意，.lib 扩展名与静态 Windows 库相同，但应用并不相同。
- `$<TARGET_RUNTIME_DLLS:target>` 返回目标在运行时依赖的 DLL 列表。
- `$<TARGET_PDB_FILE:target>` 返回.pdb 程序数据库文件的完整路径 (用于调试)。

由于共享库在程序初始化期间加载到操作系统的内存中，因此适用于提前知道程序将使用哪些库的情况。在运行时确定这些的情况又如何呢？

8.3.3. 共享模块

共享模块或模块库是共享库的一种变体，设计用于在运行时作为插件加载。与在程序启动时自动加载的标准共享库不同，共享模块仅在程序明确请求时加载。可以通过以下系统调用完成：

- 在 Windows 上使用 LoadLibrary
- 在 Linux 和 macOS 上使用 dlopen()，然后是 dlsym()

这种方法的主要原因是节约内存。许多软件应用程序，在整个进程生命周期中都未使用高级功能。每次都将这些功能加载到内存中将非常低效。

或者，提供一种途径，以便通过可以单独销售、交付和加载的专业功能来扩展主程序。

要构建共享模块，需要使用 MODULE 关键字：

```
add_library(<name> MODULE [<source>...])
```

模块设计为与将使用它的可执行文件分开部署，所以不应该尝试将模块与可执行文件链接。

8.3.4. 位置无关代码 (PIC)

由于使用了虚拟内存，程序本质上是某种程度上位置无关的。这项技术抽象了物理地址。当调用一个函数时，CPU 使用内存管理单元 (MMU) 将虚拟地址（每个进程从 0 开始）转换为相应的物理地址（在分配时确定）。有趣的是，这些映射并不总是遵循特定的顺序。

编译库引入了不确定性：不清楚哪些进程可能会使用库，或者在虚拟内存中的位置。无法预测符号的地址，或其相对于库的机器代码的位置。为了处理这个问题，需要另一层间接寻址。

PIC 将符号（如对函数和全局变量的引用）映射到运行时地址。PIC 在二进制文件中引入了一个新的节：全局偏移表 (GOT)。链接期间，计算了 GOT 节相对于 .text 节（程序代码）的相对位置。所有符号引用将通过一个偏移量指向 GOT 中的占位符。

当程序加载时，GOT 节转换为一个内存段。随着时间的推移，这个段累积了符号的运行时地址。这种方法称为“延迟加载”，确保加载器仅在需要时填充特定的 GOT 条目。

共享库和模块的所有源代码必须在使用 PIC 标志激活的情况下编译。通过将 POSITION_INDEPENDENT_CODE 目标属性设置为 ON，将告诉 CMake 适当地添加编译器特定的标志，例如为 GCC 或 Clang 添加 -fPIC。

这个属性对于共享库是自动启用的。如果共享库依赖于另一个目标，例如静态或对象库，也必须将这个属性应用于依赖目标：

```
set_target_properties(dependency  
    PROPERTIES POSITION_INDEPENDENT_CODE ON)
```

因为会检查这个属性的一致性，所以忽略这一步将导致 CMake 中的冲突。

我们的下一个讨论点转向符号。特别是，探讨名称冲突，这可能导致模糊和定义不一致。

8.4. 解决 ODR 的问题

Phil Karlton, Netscape 的主要怪人和技术愿景家，说下面的话时是正确的：

Tip

“计算机科学中有两件难事：缓存失效和给予命名。”

名称之所以难以处理，有几个原因。它们必须精确而简单，简洁而有表现力。这不仅赋予了它们意义，还使程序员能够掌握底层原始实现背后的概念。C++ 和许多其他语言增加了另一个规定：大多数名称必须唯一。

这一要求以 ODR 的形式体现：在单个翻译单元（单个.cpp 文件）的范围内，即使某个名称（无论是变量、函数、类类型、枚举、概念还是模板）多次声明，也必须精确地定义它一次。“声明”是引入符号，而“定义”则提供其所有细节，如变量的值或函数的主体。

链接期间，此规则扩展到整个程序，包含在代码中实际使用的所有非内联函数和变量。考虑以下由三个源文件组成的示例：

ch08/02-odr-fail/shared.h

```
1 int i;
```

ch08/02-odr-fail/one.cpp

```
1 #include <iostream>
2 #include "shared.h"
3
4 int main() {
5     std::cout << i << std::endl;
6 }
```

ch08/02-odr-fail/two.cpp

```
1 #include "shared.h"
```

还包括一个列表文件：

ch08/02-odr-fail/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(ODR_CXX)
3 set(CMAKE_CXX_STANDARD 20)
4 add_executable(odr one.cpp two.cpp)
```

这个例子非常简单——创建了一个 shared.h 头文件，定义了变量 i，其在两个单独的翻译单元中使用：

- one.cpp 只是将 i 打印到屏幕上
- two.cpp 只包含头文件

但当尝试构建示例时，链接器产生以下错误：

```
/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
```

符号不能定义多次，但有一个重要的例外。类型、模板和 `extern` 内联函数可以在多个翻译单元中有重复的定义，但前提是这些定义必须完全相同（它们有完全相同的标记序列）。

为了证明这一点，用一个类型的定义替换变量的定义：

ch08/03-odr-success/shared.h

```
1 struct shared {
2     static inline int i = 1;
3 }
```

然后，这样使用：

ch08/03-odr-success/one.cpp

```
1 #include <iostream>
2 #include "shared.h"
3 int main() {
4     std::cout << shared::i << std::endl;
5 }
```

其他两个文件，two.cpp 和 CMakeLists.txt，保持与 02-odr-fail 示例相同。这样的更改会链接成功：

```
[ 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
```

另外，可以将变量标记为翻译单元局部（不会导出到对象文件之外）。为此，将使用 `static` 关键字（这个关键字具有上下文相关性，所以不要将它与类中的 `static` 关键字混淆）：

ch08/04-odr-success/shared.h

```
1 static int i;
```

如果尝试链接这个示例，会看到它是有效的，所以静态变量对于每个翻译单元是分开存储的。因此，对一个的修改不会影响另一个。

ODR 规则对于静态库和对象文件完全相同，但在使用共享库构建代码时，情况就没那么明了——让我们来看一下。

8.4.1. 解决动态链接中的重复符号问题

链接器将在此处允许重复的符号。以下示例中，我们将创建两个共享库 A 和 B，每个库都有一个 `duplicated()` 函数和两个唯一的 `a()` 和 `b()` 函数：

ch08/05-dynamic/a.cpp

```
1 #include <iostream>
2 void a() {
3     std::cout << "A" << std::endl;
4 }
5 void duplicated() {
6     std::cout << "duplicated A" << std::endl;
7 }
```

第二个实现文件几乎与第一个完全相同：

ch08/05-dynamic/b.cpp

```
1 #include <iostream>
2 void b() {
3     std::cout << "B" << std::endl;
4 }
5 void duplicated() {
6     std::cout << "duplicated B" << std::endl;
7 }
```

现在，使用每个函数来看看会发生什么（简单起见，将本地声明为 `extern`）：

ch08/05-dynamic/main.cpp

```
1 extern void a();
2 extern void b();
3 extern void duplicated();
4 int main() {
5     a();
6     b();
7     duplicated();
8 }
```

前面的代码将运行每个库中的唯一函数，然后调用在两个动态库中定义的具有相同签名的函数。你认为会发生什么？链接顺序会有关系吗？让我们针对以下两种情况进行测试：

- `main_1` 目标将首先与 `a` 库链接

- main_2 目标将首先与 b 库链接

列表文件如下所示：

ch08/05-dynamic/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Dynamic CXX)
3 add_library(a SHARED a.cpp)
4 add_library(b SHARED b.cpp)
5 add_executable(main_1 main.cpp)
6 target_link_libraries(main_1 a b)
7 add_executable(main_2 main.cpp)
8 target_link_libraries(main_2 b a)
```

构建并运行两个可执行文件后，将看到以下输出：

```
root@ce492a7cd64b:/root/examples/ch08/05-dynamic# b/main_1
A
B
duplicated A

root@ce492a7cd64b:/root/examples/ch08/05-dynamic# b/main_2
A
B
duplicated B
```

啊哈！显然，链接库的顺序对链接器有关系。如果不警惕，这可能会导致混淆。与人们可能认为的不同，命名冲突在实践中并不少见。

如果定义了本地可见的符号，其将优先于从 DLL 中可用的符号。在 main.cpp 中定义 duplicated() 函数将覆盖两个目标的行为。

在从库中导出名称时，总是要非常小心，因为迟早会遇到命名冲突。

8.4.2. 使用命名空间——不要依赖链接器

C++ 命名空间是为了避免此类奇怪的问题，并更有效地处理 ODR 而发明的。最佳实践是将你的库代码包装在以库命名的命名空间中。

这种策略有助于防止由于重复符号引起的复杂问题。项目中，我们可能会遇到一个共享库链接到另一个库的情况，形成一个长链。在复杂的配置中，这种情况很常见。然而，理解仅仅将一个库链接到另一个库，并不会引入任何类型的命名空间继承至关重要。这个链中的每个链接的符号都保留在编译时的原始命名空间中。

虽然链接器的复杂性很吸引人，偶尔也是必要的，但另一个紧迫的问题经常出现：正确定义的符号会神秘消失。让我们在下一节中深入探讨这个问题。

8.5. 连接和未解析符号的顺序

链接器的行为有时可能显得反复无常，无缘无故地抛出错误。这对于不熟悉这个工具复杂性的初级开发者来说，常常是一个特别棘手的挑战。他们通常会尽可能长时间地避免处理构建配置，但当需要做出更改时——也许是要整合他们开发的库——问题就会全面爆发。

考虑以下情况：一个相对直接的依赖链，其中主可执行文件依赖于一个“外部”库。而这个外部库又依赖于包含必要变量 `int b` 的“嵌套”库。突然之间，一个神秘的错误信息出现在程序员面前：

```
outer.cpp:(.text+0x1f): undefined reference to 'b'
```

此类错误很常见。通常，表示链接器中忘记了一个库。然而，在这个场景中，库似乎已经正确地添加到了 `target_link_libraries()` 命令中：

ch08/06-unresolved/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Order CXX)
3 add_library(outer outer.cpp)
4 add_library(nested nested.cpp)
5 add_executable(main main.cpp)
6 target_link_libraries(main nested outer)
```

问题是什么呢！？很少有错误能像这样调试和理解时让人如此愤怒。我们看到的是链接顺序不正确：

ch08/06-unresolved/main.cpp

```
1 #include <iostream>
2 extern int a;
3 int main() {
4     std::cout << a << std::endl;
5 }
```

代码看起来很简单——将打印外部变量 `a`，该变量可以在 `outer` 库中找到。提前使用 `extern` 关键字声明，以下是该库的源码：

ch08/06-unresolved/outer.cpp

```
1 extern int b;
2 int a = b;
```

这相当简单——`outer` 依赖于嵌套库提供外部变量 `b`，该变量赋值给变量 `a`。来看看 `nested` 的源代码，以确认没有遗漏定义：

ch08/06-unresolved/nested.cpp

```
1 int b = 123;
```

确实，我们为 `b` 提供了定义，并且由于它没有使用 `static` 关键字标记为局部变量，因此会从 `nested` 目标正确导出。如我们之前所见，此目标在 `CMakeLists.txt` 中与主可执行文件链接：

```
1 target_link_libraries(main nested outer)
```

那么，对’`b`’的未定义引用错误从何而来？解析未定义的符号是这样工作的——链接器从左到右处理二进制文件。

当链接器遍历二进制文件时，将执行以下操作：

1. 收集此二进制文件导出的所有未定义符号，并存储起来以备后用。
2. 尝试用此二进制文件中定义的符号解析未定义的符号（来自迄今为止处理的所有二进制文件）。
3. 对下一个二进制文件重复此过程。

如果在整个操作完成后仍有符号未定义，链接将失败。这就是我们示例中的情况（CMake 将可执行文件目标的对象文件放在库的前面）：

1. 链接器处理了 `main.o`，发现对 `a` 变量的未定义引用，并将其收集起来以备将来解析。
2. 链接器处理了 `libnested.a`，没有发现未定义的引用，也没有需要解析的。
3. 链接器处理了 `libouter.a`，发现对 `b` 变量的未定义引用，并解析了对 `a` 变量的引用。

我们正确地解析了对 `a` 变量的引用，但未解析对 `b` 变量的引用。要纠正这一点，我们需要反转链接顺序，使 `nested` 在 `outer` 之后：

```
1 target_link_libraries(main outer nested)
```

有时，会遇到循环引用，其中翻译单元相互定义符号，没有一种有效的顺序可以满足所有引用。解决这个问题的唯一方法是对某些目标进行两次处理：

```
1 target_link_libraries(main nested outer nested)
```

这是一种常见做法，但使用起来略显不雅。如果有幸使用 CMake 3.24 或更高版本，可以使用 `$<LINK_GROUP>` 生成器表达式与 `RESCAN` 功能，该功能添加了链接器特定的标志，如`--start-group` 或`--end-group`，以确保能找到所有的符号：

```
1 target_link_libraries(main "$<LINK_GROUP:RESCAN,nested,outer>")
```

这种机制引入了额外的处理步骤，应当只在必要时使用。需要循环引用的情况非常罕见（并且是合理的）。遇到这个问题通常表明设计不佳，其在 Linux、BSD、SunOS，以及使用 GNU 工具链的 Windows 上得到支持。

现在准备处理 ODR 问题。我们还可能遇到哪些其他问题？链接过程中神秘地缺失符号。来看看这是关于什么的。

8.5.1. 处理未引用的符号

当创建库时，尤其是静态库，基本上是由多个对象文件捆绑在一起的归档文件。我们提到过，一些归档工具可能会创建符号索引以加速链接过程。这些索引提供了每个符号与定义它们的对象文件之间的映射。当解析一个符号时，包含该符号的对象文件会合并到最终的二进制文件中（一些链接器通过只包含文件的具体部分来进一步优化）。如果静态库中的对象文件没有引用符号，则可能会完全忽略该对象文件。因此，只有实际使用的静态库部分，才会出现在最终的二进制文件中。

然而，可能需要一些未引用的符号：

- 静态初始化：如果库有需要在 `main()` 之前初始化的全局对象（即构造函数需要执行），并且这些对象在其它地方没有直接引用；链接器可能会将其从最终二进制文件中排除。
- 插件架构：如果正在开发一个插件系统（带有模块库），其中代码需要在运行时进行识别和加载，而不需要直接引用。
- 静态库中的未使用代码：如果正在开发一个包含实用函数或代码的静态库，这些代码并非总是直接引用，但仍然希望它们出现在最终二进制文件中。
- 模板实例化：对于重度依赖模板的库；如果未明确提及，一些模板实例化可能会在链接过程中忽略。
- 链接问题：特别是对于复杂的构建系统或详尽的代码库，链接可能会产生不可预测的结果，其中某些符号或代码部分似乎缺失。

这时，强制在链接过程中包含所有对象文件很有用，可通过一种称为“全归档链接”的模式来实现。

特定的编译器链接标志如下：

- GCC 使用 `--whole-archive`
- Clang 使用 `--force-load`
- MSVC 使用 `/WHOLEARCHIVE`

为了实现这一点，可以使用 `target_link_options()` 命令：

```
1 target_link_options(tgt INTERFACE
2     -Wl,--whole-archive $<TARGET_FILE:lib1> -Wl,--no-whole-archive
3 )
```

然而，这个命令是特定于链接器的，因此结合生成器表达式，来检测不同的编译器，并提供必要的标志。幸运的是，CMake 3.24 引入了一个新的生成器表达式用于此目的：

```
1 target_link_libraries(tgt INTERFACE
2     "$<LINK_LIBRARY:WHOLE_ARCHIVE,lib1>"
3 )
```

使用这种方法可以确保 `tgt` 目标包含 `lib1` 库的所有对象文件。

尽管如此，还需要考虑一些潜在的问题：

- 增加二进制文件大小：这个标志可能会大幅增加最终二进制文件大小，其包含了指定库的所有对象文件，无论是否使用。

- 符号冲突的可能性：引入所有符号可能会导致与其他符号冲突，从而引发链接错误。
- 维护负担：过度依赖此类标志会掩盖代码设计或结构中的潜在问题。

了解了如何解决常见的链接挑战后，我们现在可以继续准备项目的测试了。

8.6. 分离 main() 进行测试

链接器已经强制执行 ODR，并确保在链接过程中所有外部符号，都提供定义。另一个需要面临的与链接器相关的挑战是，对项目进行优雅且高效的测试。

理想情况下，应该测试与生产环境中运行完全相同的源代码。全面的测试管道会构建源代码，对生成的二进制文件运行测试，然后打包和分发可执行文件（可选地排除测试本身）。

但是，如何实现呢？可执行文件通常具有精确的执行流程，通常涉及读取命令行参数。C++ 的编译特性不容易支持可临时插入二进制文件中，进行测试的可插拔单元。这表明我们需要一个微妙的策略来应对这个挑战。

幸运的是，可以使用链接器，以优雅的方式来处理这个问题。考虑将程序的所有逻辑从 main() 函数中提取出来，放到一个外部函数 start_program() 中：

ch08/07-testing/main.cpp

```

1  extern int start_program(int, const char**);
2  int main(int argc, const char** argv) {
3      return start_program(argc, argv);
4 }
```

当 main() 函数以这种形式编写时，跳过对其的测试是合理的；只是将参数转发到一个在别处定义的函数（在另一个文件中）。然后，可以创建一个包含原始 main() 源代码的新函数 start_program() 的库。这个例子中，代码检查命令行参数计数是否大于 1：

ch08/07-testing/program.cpp

```

1 #include <iostream>
2 int start_program(int argc, const char** argv) {
3     if (argc <= 1) {
4         std::cout << "Not enough arguments" << std::endl;
5         return 1;
6     }
7     return 0;
8 }
```

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：

ch08/07-testing/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.26)
2 project(Testing CXX)
```

```
3 add_library(program program.cpp)
4 add_executable(main main.cpp)
5 target_link_libraries(main program)
```

main 目标只提供了所需的 main() 函数，命令行参数验证逻辑包含在 program 目标中。现在可以通过创建另一个带有 main() 函数的可执行文件来进行测试，该文件将作为测试用例。

在现实世界的场景中，像 GoogleTest 或 Catch2 这样的框架将提供自己的 main() 方法，可以用来替换程序的入口点并运行所有定义的测试。我们将在第 11 章中深入讨论实际的测试主题。现在，先来关注一般性问题，并在 main() 函数中直接编写测试用例：

ch08/07-testing/test.cpp

```
1 #include <iostream>
2 extern int start_program(int, const char**);
3 using namespace std;
4 int main()
5 {
6     cout << "Test 1: Passing zero arguments to start_program:\n";
7     auto exit_code = start_program(0, nullptr);
8     if (exit_code == 0)
9         cout << "Test FAILED: Unexpected zero exit code.\n";
10    else
11        cout << "Test PASSED: Non-zero exit code returned.\n";
12    cout << endl;
13
14    cout << "Test 2: Passing 2 arguments to start_program:\n";
15    const char *arguments[2] = {"hello", "world"};
16    exit_code = start_program(2, arguments);
17    if (exit_code != 0)
18        cout << "Test FAILED: Unexpected non-zero exit code\n";
19    else
20        cout << "Test PASSED\n";
21 }
```

前面的代码将两次调用 start_program，一次不带参数，一次带参数，并检查返回的退出代码是否正确。如果测试正确执行，将看到以下输出：

```
./test
Test 1: Passing zero arguments to start_program:
Not enough arguments
Test PASSED: Non-zero exit code returned

Test 2: Passing 2 arguments to start_program:
Test PASSED
```

“参数不足”这一行来自 start_program()，并且是预期的错误消息（正在检查程序是否正确地失败）。

这个单元测试在干净代码和优雅测试实践方面还有很多不足，但这是一个开始。

现在，已经两次定义了 `main()`：

- `main.cpp` 用于生产环境
- `test.cpp` 用于测试目的

现在，在 `CMakeLists.txt` 的底部定义测试可执行文件：

```
1 add_executable(test test.cpp)
2 target_link_libraries(test program)
```

这个添加创建了一个新的目标，其与我们的生产代码链接相同的二进制代码。然而，这给予了我们灵活性，可以按需调用所有导出的函数。多亏了这个，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！

8.7. 总结

在 CMake 中进行链接可能一开始看起来很简单，但当我们深入挖掘时，会发现其背后大有乾坤。毕竟，链接可执行文件并不像拼图那样简单。当深入研究对象文件和库的结构时，存储各种类型的数据、指令、符号名称等节（section）都需要重新排序。在程序运行之前，这些节需要经过重定位。

解析符号也非常关键。链接器必须对所有翻译单元中的引用进行排序，确保没有遗漏。当这些都处理好了，链接器就会创建程序头部，并将其放入最终的可执行文件中。这个头部为系统加载器提供指令，详细说明如何将合并的节转换为段，这些段将构成进程的运行时内存镜像。我们还讨论了三种类型的库：静态库、共享库和共享模块。探讨了它们的区别，以及在哪些场景下某种库可能比其他库更适合使用。此外，还提到了 PIC ——一个强大的概念，促进了符号的延迟绑定。

ODR 是 C++ 的一个概念，由链接器强制执行。我们查看了如何在静态库和动态库中处理最基本的符号重复问题。还强调了尽可能使用命名空间的价值，并建议不要过分依赖链接器来避免符号冲突。

对于一个看似简单的步骤（考虑到 CMake 专门用于链接的有限命令），其确实有其复杂性。特别是在处理具有嵌套和循环依赖关系的库时，其中一个比较棘手的问题是链接的顺序。现在我们了解了链接器如何选择最终进入二进制文件的符号，以及如何在需要时覆盖这种行为。

最后，研究了如何利用链接器为测试准备程序——通过将 `main()` 函数分离到另一个翻译单元。这使我们能够引入另一个可执行文件，该文件针对与生产中将要执行的确切机器代码运行测试。

凭借对链接的新知识，已准备好将外部库引入 CMake 项目。下一章中，将探讨如何在 CMake 中管理依赖项。

8.8. 扩展阅读

- 可执行与链接格式（ELF）文件结构：

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- CMake 手册关于 `add_library()` 的用法:

https://cmake.org/cmake/help/latest/command/add_library.html

- 依赖地狱 (dependency hell):

https://en.wikipedia.org/wiki/Dependency_hell

- 模块与共享库之间的区别:

<https://stackoverflow.com/questions/4845984/difference-between-modules-and-shared-libraries>

第 9 章 管理依赖关系

解决方案的大小起始并不重要；但随着它的增长，可能会选择依赖其他项目。避免创建和维护样板代码的工作至关重要，可以为真正重要的事情腾出时间：业务逻辑。外部依赖项有多种用途，并提供框架和功能，解决复杂问题，在构建和确保代码质量中发挥关键作用。这些依赖项各不相同，从专业的编译器如 Protocol Buffers (ProtocolBuf) 到测试框架如 Google Test。

使用开源项目或内部代码时，高效地管理外部依赖项是必不可少的。如果手动完成这项工作，将需要大量的设置时间和持续的支持。幸运的是，CMake 擅长处理各种依赖管理方法，同时紧跟行业标准。

首先将了解如何识别和利用宿主系统上已经存在的依赖项，从而避免不必要的下载和延长的编译时间。这项任务相对简单，许多软件包要么与 CMake 兼容，要么直接由 CMake 支持。我们还将探讨如何指导 CMake 定位和包含那些缺乏这种本地支持的依赖项。对于历史遗留的软件包，采用另一种方法可能更有益：可以使用曾经流行的 pkg-config 工具来处理更繁琐的任务。

此外，还将深入探讨如何管理在线可用，但尚未安装在系统上的依赖项。我们将研究如何从 HTTP 服务器、Git 和其他类型的仓库中获取这些依赖项。我们还将讨论如何选择最佳方法：首先在系统中搜索，如果未找到该软件包，再转而获取。最后，将回顾一种在特殊情况下可能适用的下载外部项目的旧技术。

本章，将包含以下内容：

- 使用已安装的依赖项
- 使用系统中不存在的依赖项

9.1. 示例下载

可以在 GitHub 上找到本章中出现的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch09>。

为了构建本书提供的示例，请使用以下推荐的命令：

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

请确保将 `<build tree>` 和 `<source tree>` 占位符替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的位置。

9.2. 使用已经安装的依赖项

当项目依赖于一个很流行的库时，操作系统很可能已经安装了这个库的包。我们只需要将其连接到项目的构建过程中。那么，该如何操作呢？需要找出系统中包的位置，以便 CMake 可以使用其文件。可以手动完成这个操作，但每个环境都有所不同。在一个系统上有效路径可能在另一个系统上无效。因此，在构建时，应该自动查找这些路径。有不同的方法可以实现这一点，但最好的方法是使用 CMake 内置的 `find_package()` 命令，它了解如何找到许多常用的包。

如果我们的包不受支持，那么有两个选择：

- 可以编写一个名为 `find-module` 的插件来帮助 `find_package()`
- 可以使用一种较老的方法，称为 `pkg-config`

先从推荐选项开始。

9.2.1. 使用 CMake 的 `find_package()` 查找包

先从以下场景开始：想要改进网络通信或数据存储的方式。简单的纯文本文件或像 JSON 和 XML 这样的开放文本格式在大小方面太冗长。使用二进制格式会有所帮助，而像 Google 的 Protobuf 这样的知名库就是答案。

已经阅读了说明，并在系统上安装了所需的软件。现在怎么办？如何让 CMake 的 `find_package()` 找到并使用这个库呢？

要执行此示例，必须安装想要使用的依赖项，`find_package()` 命令只查找已经存在于系统上的包。假定已经安装了所有内容，或者用户知道如何安装所需的软件。如果想处理其他情况，需要备用计划。

使用 Protobuf 时，情况相当简单：可以自己从官方仓库（<https://github.com/protobuf/protobuf>）下载、编译并安装库，或者使用操作系统的包管理器。如果按照第 1 章中提到的 Docker 镜像进行这些示例，所有依赖项就是已经安装好了的，不需要做任何事情。但如果想自己尝试安装，以下是在 Debian Linux 上安装 Protobuf 库和编译器的命令：

```
$ apt update  
$ apt install protobuf-compiler libprotobuf-dev
```

如今，许多项目选择支持 CMake，通过创建配置文件，并在安装过程中将其放入适当的系统目录来进行。配置文件是选择支持 CMake 的项目的基本部分。

如果想使用一个没有配置文件的库，请不要担心。CMake 支持一种外部机制来查找此类库，称为 `find` 模块。与配置文件不同，`find` 模块不是帮助定位的项目的一部分。实际上，CMake 本身通常带有这些 `find` 模块，用于许多流行的库。

如果陷入困境，既没有配置文件也没有 `find` 模块，也还有其他选择：

- 为特定包编写自己的 `find` 模块，并将它们包含在项目中
- 使用 `FindPkgConfig` 模块来利用传统的 Unix 包定义文件
- 编写配置文件，并要求包维护者将其包含在内

可能认为没有准备好自己创建这样的合并请求，很可能不需要这样做。CMake 自带了 150 多个 `find` 模块，可以找到如 Boost、bzip2、curl、curses、GIF、GTK、iconv、ImageMagick、JPEG、Lua、OpenGL、OpenSSL、PNG、PostgreSQL、Qt、SDL、Threads、XML-RPC、X11 和 zlib 等库，以及在此示例中将使用的 Protobuf 文件。完整的列表可以在 CMake 文档中找到（请参阅“扩展阅读”）。

`find` 模块和配置文件都可以与 CMake 的 `find_package()` 命令一起使用，CMake 首先检查其内置的 `find` 模块。如果找不到所需的模块，会继续检查不同包提供的配置文件。CMake

扫描通常安装包的路径（取决于操作系统），查找匹配以下模式的文件：

- <CamelCasePackageName>Config.cmake
- <kebab-case-package-name>-config.cmake

如果想将外部 `find` 模块添加到项目中，请设置 `CMAKE_MODULE_PATH` 变量。CMake 将首先扫描此目录。

回到示例，目标很简单：我想展示可以有效地构建一个使用 Protobuf 的项目。别担心，不需要了解 Protobuf 来理解发生的事情。简单来说，Protobuf 是一个将数据以特定二进制格式保存的库。这使得写入和从文件或网络读取 C++ 对象变得容易。为了设置这个，我们使用一个`.proto` 文件来给 Protobuf 数据结构：

ch09/01-find-package-variables/message.proto

```
syntax = "proto3";
message Message {
    int32 id = 1;
}
```

这段代码是一个简单的架构定义，包括一个 32 位整数。Protobuf 包附带了一个二进制文件，可以将这些`.proto` 文件编译成 C++ 源文件和头文件，以便应用程序可以使用。需要将这个编译步骤添加到构建过程中，但稍后会回到这个话题。现在，来看看 `main.cpp` 文件是如何使用 Protobuf 生成的输出：

ch09/01-find-package-variables/main.cpp

```
1 #include "message.pb.h"
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     Message m;
7     m.set_id(123);
8     m.PrintDebugString();
9     fstream fo("./hello.data", ios::binary | ios::out);
10    m.SerializeToOstream(&fo);
11    fo.close();
12    return 0;
13 }
```

已经包含了预期的 Protobuf 生成的 `message.pb.h` 头文件。这个头文件将包含在 `message.proto` 中配置的 `Message` 对象的定义。在 `main()` 函数中，创建了一个简单的 `Message` 对象。将它的 `id` 字段设置为 123 作为一个随机示例，然后将其调试信息打印到标准输出。接下来，这个对象的二进制版本可写入文件流。这是像 Protobuf 这样的序列化库最基本的使用场景。

编译 `main.cpp` 之前，必须生成 `message.pb.h` 头文件。这是由 `protoc`，即 Protobuf

编译器完成，将 message.proto 作为输入。管理这个过程听起来很复杂，但实际上并不复杂！

这就是 CMake 魔法发生的地方：

ch09/01-find-package-variables/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(FindPackageProtobufVariables CXX)
3 find_package(Protobuf REQUIRED)
4 protobuf_generate_cpp(GENERATED_SRC GENERATED_HEADER
5                         message.proto)
6 add_executable(main main.cpp ${GENERATED_SRC} ${GENERATED_HEADER})
7 target_link_libraries(main PRIVATE ${Protobuf_LIBRARIES})
8 target_include_directories(main PRIVATE
9                             ${Protobuf_INCLUDE_DIRS} ${CMAKE_CURRENT_BINARY_DIR})
10 )
```

我们分解一下：

- 前两行很直接：设置项目并指定将使用 C++ 语言。
- `find_package(Protobuf REQUIRED)` 告诉 CMake 找到 Protobuf 库（通过执行捆绑的 `FindProtobuf.cmake` `find` 模块）并为项目准备使用。因为我们使用了 `REQUIRED` 关键字，如果找不到库，构建将停止。
- `protobuf_generate_cpp` 是 `Protobuf find` 模块中定义的自定义函数，自动化调用 `protoc` 编译器的过程。成功编译后，将生成的源文件路径存储在提供的前两个参数 `GENERATED_SRC` 和 `GENERATED_HEADER` 中。后续参数将视为要编译的文件列表 (`message.proto`)。
- `add_executable` 使用 `main.cpp` 和 Protobuf 生成的文件创建可执行文件。
- `target_link_libraries` 告诉 CMake 将 Protobuf 库链接到可执行文件。
- `target_include_directories`(将必要的 `INCLUDE_DIRS` 提供的路径和 `CMAKE_CURRENT_BINARY_DIR` 添加到包含路径中。后者告诉编译器在哪里找到 `message.pb.h` 头文件。

Protobuf `find` 模块提供了以下功能：

- 找到 Protobuf 库及其编译器。
- 提供了帮助函数来编译.proto 文件。
- 设置了包含和链接的路径变量。

虽然不是每个模块都像 Protobuf 那样提供方便的帮助函数，但大多数模块确实设置了一些关键的变量，这些变量对于管理项目中的依赖项很有用。无论使用内置的 `find` 模块还是配置文件，在成功找到包之后，可以期望设置以下全部或部分变量：

- `<PKG_NAME>_FOUND`: 这表明是否成功找到了包。
- `<PKG_NAME>_INCLUDE_DIRS` 或 `<PKG_NAME>_INCLUDES`: 这指向包的头文件所在的目录。
- `<PKG_NAME>_LIBRARIES` 或 `<PKG_NAME>_LIBS`: 这些是要链接的库的列表。

- <PKG_NAME>_DEFINITIONS：包含包所需的编译器定义。

运行 `find_package()` 后，可以立即检查 `<PKG_NAME>_FOUND` 变量，以查看 CMake 是否成功找到包。

如果包模块是为 CMake 3.10 或更高版本编写的，还可能会提供目标定义。这些目标将指定为 IMPORTED 目标，以区分其源自外部的依赖。

Protobuf 是学习 CMake 中依赖项时的一个很好的探索示例，它定义了特定于模块的变量和 IMPORTED 目标。这样的目标允许我们编写更简洁的代码：

ch09/02-find-package-targets/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.26.0)
2 project(FindPackageProtobufTargets CXX)
3 find_package(Protobuf REQUIRED)
4 protobuf_generate_cpp(GENERATED_SRC GENERATED_HEADER
5   message.proto)
6 add_executable(main main.cpp ${GENERATED_SRC} ${GENERATED_HEADER})
7 target_link_libraries(main PRIVATE protobuf::libprotobuf)
8 target_include_directories(main PRIVATE
9   ${CMAKE_CURRENT_BINARY_DIR})

```

看看这段高亮代码与之前示例版本的对比：不是使用列出文件和目录的变量，而是使用 IMPORTED 目标是一个好主意。这种方法简化了列表文件。自动处理瞬态使用要求或传播属性，如下面使用 `protobuf::libprotobuf` 目标所示。

Note

如果想确切知道一个特定的 `find` 模块提供了什么，最好的资源是在线文档。例如，可以在 CMake 官方网站的这个链接上找到 Protobuf 的详细信息：<https://cmake.org/cmake/help/latest/module/FindProtobuf.html>。

为了保持简单，本节中的示例如果用户系统中没有找到 Protobuf 库将会直接失败。但真正健壮的解决方案，应该验证 `Protobuf_FOUND` 变量，并为用户呈现一个清晰的诊断信息（以便可以安装），或者自动执行安装。我们将在本章后面学习如何做到这一点。

`find_package()` 命令有几个可选参数可以使用。虽然有一个更长的参数列表，但这里将重点关注关键的几个。该命令的基本格式如下：

```
find_package(<Name> [version] [EXACT] [QUIET] [REQUIRED])
```

分解一下这些可选参数的含义：

`version` 这指定了需要的包的最小版本，格式为 `major.minor.patch.tweak`（例如 1.22）。还可以指定一个范围，如 1.22…1.40.1，使用三个点作为分隔符。

- `EXACT`：与非范围 `[version]` 一起使用，告诉 CMake 需要确切的版本，而不是更新的版本。

- QUIET：这将抑制关于包是否被找到的所有消息。
- REQUIRED：如果找不到包，这将停止构建，并且即使使用了 QUIET，也会显示诊断信息。

如果确信一个包存在于系统中，但 `find_package()` 没有找到它，有一种方法可以进一步挖掘。从 CMake 3.24 开始，可以在调试模式下运行配置阶段以获取更多信息。使用以下命令：

```
cmake -B <build tree> -S <source tree> --debug-find-pkg=<pkg>
```

使用这个命令时要小心。确保正确地输入包名，它对大小写敏感。

关于 `find_package()` 命令的更多信息可以在文档页面这里找到：https://cmake.org/cmake/help/latest/command/find_package.html。

Find 模块旨在作为一种非常方便的方式，为 CMake 提供有关已安装依赖项的信息。大多数流行的库在所有主要平台上都得到了 CMake 的广泛支持。但当我们想要使用一个尚未有专用 `find` 模块的第三方库时，应该怎么办呢？

编写自己的 `find` 模块

极少数情况下，项目中真正想使用的库没有提供配置文件，CMake 中也没有现成的 `find` 模块。这时，可以为该库编写一个自定义的 `find` 模块，并将其与项目一起分发。这种情况并不理想，但在照顾到项目用户的情况下，必须这样做。

我们可以尝试为 `libpqxx` 库编写一个自定义的 `find` 模块，这是一个 PostgreSQL 数据库的客户端。`libpqxx` 在本书的 Docker 镜像中预安装，所以如果使用那个镜像，就不用担心了。Debian 用户可以使用 `libpqxx-dev` 包来安装（其他操作系统可能需要不同的命令）：

```
apt-get install libpqxx-dev
```

我们将从在项目的源树中的 `cmake/module` 目录中创建一个名为 `FindPQXX.cmake` 的新文件开始。为了确保调用 `find_package()` 时，CMake 可以发现这个 `find` 模块，我们将其路径使用 `list(APPEND)`，添加到 `CMakeLists.txt` 中的 `CMAKE_MODULE_PATH` 变量中。注意，在搜索其他位置之前，CMake 会首先检查 `CMAKE_MODULE_PATH` 中列出的目录以查找 `find` 模块。完整列表文件应该看起来像这样：

ch09/03-find-package-custom/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(FindPackageCustom CXX)
3 list(APPEND CMAKE_MODULE_PATH
4       "${CMAKE_SOURCE_DIR}/cmake/module/")
5 find_package(PQXX REQUIRED)
6 add_executable(main main.cpp)
7 target_link_libraries(main PRIVATE PQXX::PQXX)
```

有了这些设置，可以继续编写实际的 `find` 模块。如果 `FindPQXX.cmake` 文件为空，即使使用 `REQUIRED` 调用 `find_package()`，CMake 也不会引发错误。`find` 模块的作者有责任设置正确的变量，并遵循最佳实践（例如：引发错误）。根据 CMake 的指南，这里有一些关键点需要注意：

- 当调用 `find_package(<PKG_NAME> REQUIRED)` 时, CMake 会将 `<PKG_NAME>_FIND_REQUIRED` 变量设置为 1。`find` 模块应该在找不到库时使用 `message(FATAL_ERROR)`。
- 当使用 `find_package(<PKG_NAME> QUIET)` 时, CMake 会将 `<PKG_NAME>_FIND_QUIETLY` 设置为 1。`find` 模块应避免显示任何其他消息。
- CMake 会将 `<PKG_NAME>_FIND_VERSION` 变量设置为在列表文件中指定的版本。如果 `find` 模块无法定位正确的版本, 应该引发一个 `FATAL_ERROR`。

当然, 最好遵循上述规则, 以便与其他 `find` 模块保持一致。

为了为 PQXX 创建一个优雅的 `find` 模块, 可以遵循以下步骤:

1. 如果库和头文件的路径已经知道 (由用户提供或从之前的运行的缓存中检索), 使用这些路径来创建一个 `IMPORTED` 目标。如果这样做, 可以停止。
2. 如果路径未知, 首先找到底层依赖项 (这种情况下是 PostgreSQL) 的库和头文件。
3. 接下来, 搜索已知路径以定位 PostgreSQL 客户端库的二进制版本。
4. 同样, 扫描已知路径以找到 PostgreSQL 客户端的头文件。
5. 最后, 确认是否同时找到了库和头文件。如果是, 创建一个 `IMPORTED` 目标。

为了为 PQXX 创建一个健壮的 `find` 模块, 我们将关注几个重要任务。首先, 创建 `IMPORTED` 目标可以在两种情况下发生——用户指定库的路径或路径自动检测。为了保持代码整洁并避免重复, 将编写一个函数来管理搜索过程的结果。

定义 `IMPORTED` 目标

要设置一个 `IMPORTED` 目标, 实际上只需要用 `IMPORTED` 关键字定义一个库。这允许在调用 `CMakeLists.txt` 列表文件时使用 `target_link_libraries()` 命令。我们需要指定库的类型, 为了简化, 将其标记为 `UNKNOWN`。所以, 我们不关心库是静态还是动态的, 只是想向链接器传递一个参数。

接下来, 需要为目标设置一些基本属性 — 即 `IMPORTED_LOCATION` 和 `INTERFACE_INCLUDE_DIRECTORIES`。使用函数提供的参数来设置这些属性, 可以指定其他属性, 如 `COMPILE_DEFINITIONS`, 但对于 PQXX 来说没有必要。

然后, 为了使 `find` 模块更高效, 将找到的路径存储在缓存变量中。

这样, 就不必在未来的运行中重复搜索了。值得注意的是, 需要明确地将 `PQXX_FOUND` 设置为缓存, 使其全局可用, 并允许用户的 `CMakeLists.txt` 引用它。

最后, 将这些缓存变量标记为高级, 除非激活高级选项, 否则在 CMake GUI 中会隐藏它们。这也是我们也将采用的常见最佳实践。

以下是这些操作的代码:

`ch09/03-find-package-custom/cmake/module/FindPQXX.cmake`

```

1 # Defining IMPORTED targets
2 function(define_imported_target library headers)
3     add_library(PQXX::PQXX UNKNOWN IMPORTED)
4     set_target_properties(PQXX::PQXX PROPERTIES
5         IMPORTED_LOCATION ${library})

```

```

6     INTERFACE_INCLUDE_DIRECTORIES ${headers}
7   )
8   set(PQXX_FOUND 1 CACHE INTERNAL "PQXX found" FORCE)
9   set(PQXX_LIBRARIES ${library}
10    CACHE STRING "Path to pqxx library" FORCE)
11  set(PQXX_INCLUDES ${headers}
12    CACHE STRING "Path to pqxx headers" FORCE)
13  mark_as_advanced(FORCE PQXX_LIBRARIES)
14  mark_as_advanced(FORCE PQXX_INCLUDES)
15 endfunction()

```

现在，我们将讨论如何使用自定义或之前存储的路径来进行快速设置。

接受用户提供的路径并重用缓存值

考虑一种情况，即用户在非标准位置安装了 PQXX，并通过命令行参数使用-D 提供了所需的路径。如果是这样，我们立即调用我们之前定义的函数并使用 return() 停止搜索。假设用户已经提供了库，及其依赖项（如 PostgreSQL）的准确路径：

ch09/03-find-package-custom/cmake/module/FindPQXX.cmake (continued)

```

1 ...
2
3 # Accepting user-provided paths and reusing cached values
4 if (PQXX_LIBRARIES AND PQXX_INCLUDES)
5   define_imported_target(${PQXX_LIBRARIES} ${PQXX_INCLUDES})
6   return()
7 endif()

```

如果配置之前已经进行过，因为变量 PQXX_LIBRARIES 和 PQXX_INCLUDES 存储在缓存中，这个条件将成立。

现在，是处理 PQXX 依赖的其他库的时候了。

搜索嵌套依赖项

为了使用 PQXX，主机系统也必须安装了 PostgreSQL。虽然在当前 find 模块中使用另一个 find 模块是可行的，但应该传递 REQUIRED 和 QUIET 标志，以确保嵌套搜索和主搜索之间的行为一致。为此，将设置两个辅助变量来存储需要传递的关键词，并根据 CMake 接收到的参数来对其进行填充：PQXX_FIND_QUIETLY 和 PQXX_FIND_REQUIRED。

```

1 # Searching for nested dependencies
2 set(QUIET_ARG)
3 if(PQXX_FIND_QUIETLY)
4   set(QUIET_ARG QUIET)
5 endif()
6
7 set(REQUIRED_ARG)
8 if(PQXX_FIND_REQUIRED)

```

```
9     set(REQUIRED_ARG REQUIRED)
10    endif()
11    find_package(PostgreSQL ${QUIET_ARG} ${REQUIRED_ARG})
```

完成这一步后，我们将深入探讨如何准确地定位 PQXX 库在操作系统中的位置。

搜索库文件

CMake 提供了 `find_library()` 命令来帮助查找库文件。这个命令将接受要查找的文件名和可能的路径列表，格式化为 CMake 的路径样式：

```
find_library(<VAR_NAME> NAMES <NAMES> PATHS <PATHS> <...>)
```

`<VAR_NAME>` 将作为存储命令输出的变量的名称。如果找到匹配的文件，其路径将存储在 `<VAR_NAME>` 变量中。否则，`<VAR_NAME>-NOTFOUND` 变量将设置为 1。我们将使用 `PQXX_LIBRARY_PATH` 作为我们的 `VAR_NAME`，所以将得到 `PQXX_LIBRARY_PATH` 中的路径，或者 `PQXX_LIBRARY_PATH-NOTFOUND` 中的 1。

PQXX 库通常将其位置导出到 `$ENV{PQXX_DIR}` 环境变量，所以系统可能已经知道其位置。可以包含 `file(TO_CMAKE_PATH)` 这个格式化的路径：

ch09/03-find-package-custom/cmake/module/FindPQXX.cmake (continued)

```
1 ...
2
3 # Searching for library files
4 file(TO_CMAKE_PATH "$ENV{PQXX_DIR}" _PQXX_DIR)
5 find_library(PQXX_LIBRARY_PATH NAMES libpqxx pqxx
6             PATHS
7             ${_PQXX_DIR}/lib/${CMAKE_LIBRARY_ARCHITECTURE}
8             # (...) many other paths - removed for brevity
9             /usr/lib
10            NO_DEFAULT_PATH
11 )
```

`NO_DEFAULT_PATH` 关键字指示 CMake 跳过其标准搜索路径列表。虽然不想这样做（默认路径通常正确），但使用 `NO_DEFAULT_PATH` 允许在必要时明确指定相应的搜索位置。

接下来，我们将查找库所需的头文件，库的用户可以包含这些头文件。

搜索头文件

为了搜索所有已知的头文件，将使用 `find_path()` 命令，其与 `find_library()` 命令非常相似。主要区别在于 `find_library()` 会自动添加系统特定的库扩展名，而使用 `find_path()` 时，需要指定确切的名称。

此外，不要在这里混淆 `pqxx/pqxx`。这是一个实际的头文件，但库创建者会故意省略其扩展名，以与 C++ `#include` 指令保持一致。这使得它可以用尖括号使用，如下所示：`#include <pqxx/pqxx>`。

以下是片段：

ch09/03-find-package-custom/cmake/module/FindPQXX.cmake (续)

```
1 ...
2 # Searching for header files
3 find_path(PQXX_HEADER_PATH NAMES pqxx/pqxx
4           PATHS
5             ${_PQXX_DIR}/include
6             # (...) many other paths - removed for brevity
7             /usr/include
8             NO_DEFAULT_PATH
9 )
```

接下来，我们将探讨如何完成搜索过程，处理缺失的路径，并调用定义导入目标的函数。

返回最终结果

现在，检查是否设置了 `PQXX_LIBRARY_PATH-NOTFOUND` 或 `PQXX_HEADER_PATHNOTFOUND` 变量。可以手动打印诊断消息并停止构建，或者使用 CMake 的 `find_package_handle_standard_args()` 辅助函数。如果这个函数的路径变量正确填充，则将 `<PKG_NAME>_FOUND` 变量设置为 1。它还提供适当的诊断消息（会尊重 `QUIET` 关键字），并在 `find_package()` 调用中提供 `REQUIRED` 关键字时，如果路径变量未找到，则会以 `FATAL_ERROR` 停止执行。

如果找到了库，将调用之前编写的函数来定义 `IMPORTED` 目标，并将路径存储在缓存中：

ch09/03-find-package-custom/cmake/module/FindPQXX.cmake (continued)

```
1 ...
2
3 # Returning the final results
4 include(FindPackageHandleStandardArgs)
5 find_package_handle_standard_args(
6   PQXX DEFAULT_MSG PQXX_LIBRARY_PATH PQXX_HEADER_PATH
7 )
8 if (PQXX_FOUND)
9   define_imported_target(
10     "${PQXX_LIBRARY_PATH};${POSTGRES_LIBRARIES}"
11     "${PQXX_HEADER_PATH};${POSTGRES_INCLUDE_DIRECTORIES}"
12   )
13 elseif(PQXX_FIND_REQUIRED)
14   message(FATAL_ERROR "Required PQXX library not found")
15 endif()
```

就是这样！这个 `find` 模块将找到 `PQXX` 并创建适当的 `PQXX::PQXX` 目标。完整的文件可以在本书的示例存储库中找到。

对于那些得到良好支持，且很可能已经安装的库，这种方法非常有效。但如果处理的是较旧的、不太受支持的包怎么办？类 Unix 系统有一个名为 `pkg-config` 的工具，CMake 也有一个有用的包装模块来支持它。

9.2.2. 使用 FindPkgConfig 发现遗留包

管理依赖项并找出必要的编译标志，这和 C++ 库本身一样古老。为了应对这个问题，已经开发了各种工具，从简单的机制到集成了构建系统和 IDE 的综合解决方案。PkgConfig (freedesktop.org/wiki/Software/pkg-config) 就是这样一个工具，它曾经非常流行，在类 Unix 系统中非常常见，也适用于 macOS 和 Windows。

然而，更现代的解决方案正逐渐取代 PkgConfig。那么，仍然需要考虑支持它吗？可能性不大，原因如下：

- 如果库没有提供 .pc PkgConfig 文件，为一种过时的工具编写定义文件就没有多大价值
- 可以选择一个支持 CMake 的库的新版本（将在本章稍后讨论如何从互联网上下载依赖项）
- 该包广泛使用，CMake 的最新版本可能已经包含了它的 find 模块
- 在线有社区创建的 find 模块可用，并且其许可证允许你使用它，也是另一个不错的选择
- 能编写和维护自己的 find 模块

只有正在使用一个已经提供 PkgConfig .pc 文件的库版本，并且没有可用的配置模块或 find 模块时，才考虑使用 PkgConfig。另外，创建自己的 find 模块不是一个可行的选项时，应该有充分的理由。如果确信不需要 PkgConfig，可以跳过这个部分。

但并不是所有环境都可以快速更新到库的最新版本。许多公司仍在生产中使用遗留系统，这些系统不再接收最新包。如果在系统中有一个特定库的 .pc 文件，看起来就像这里显示的 foobar 文件一样：

```
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
```

PkgConfig 的格式很简单，许多熟悉这个工具的开发者出于习惯而使用，而不是学习更高级的系统如 CMake。尽管很简单，PkgConfig 可以检查特定库，及其版本是否可用，并且还可以获取库的链接标志和目录信息。

要与 CMake 一起使用它，需要找到你系统上的 pkg-config 工具，运行特定的命令，然后将结果存储起来，供编译器以后使用。每次使用 PkgConfig 时都进行这些步骤可能会觉得很多工作。幸运的是，CMake 提供了一个 FindPkgConfig find 模块。如果找到了 pkg-config，将设置 PKG_CONFIG_FOUND。然后可以使用 pkg_check_modules() 来查找需要的包。

之前的章节中，已经熟悉了 libpqxx，它提供了 .pc 文件，我们尝试使用 PkgConfig 来找到它。为了实际操作，我们写一个简单的 main.cpp 文件：

```
ch09/04-find-pkg-config/main.cpp
```

```
1 #include <pqxx/pqxx>
2 int main()
3 {
4     // We're not actually connecting, but
5     // just proving that pqxx is available.
6     pqxx::nullconnection connection;
7 }
```

在列表文件中，通常从 `find_package()` 函数开始，如果找不到库，就切换到 `PkgConfig`。这种方法在环境更新时很有用，可以不修改代码就继续使用。为了保持这个示例简洁，我们省略了这一部分。

ch09/04-find-pkg-config/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(FindPkgConfig CXX)
3 find_package(PkgConfig REQUIRED)
4 pkg_check_modules(PQXX REQUIRED IMPORTED_TARGET libpqxx)
5 message("PQXX_FOUND: ${PQXX_FOUND}")
6 add_executable(main main.cpp)
7 target_link_libraries(main PRIVATE PkgConfig::PQXX)
```

分解一下发生了什么：

1. 使用 `find_package()` 命令来定位 `PkgConfig`。如果 `pkg-config` 缺失，则由于 `REQUIRED` 关键字，过程将停止。
2. `FindPkgConfig` `find` 模块中的 `pkg_check_modules()`，自定义宏设置了一个名为 `PQXX` 的新 `IMPORTED` 目标。`find` 模块寻找 `libpqxx` 依赖项，如果找不到，将再次因为 `REQUIRED` 关键字而失败。`IMPORTED_TARGET` 关键字至关重要；否则，需要手动定义目标。
3. 使用 `message()` 函数验证设置，显示 `PQXX_FOUND`。如果之前没有使用 `REQUIRED`，这里需要检查变量是否设置，以激活其他备选方案。
4. 使用 `add_executable()` 声明主可执行文件。
5. 最后，使用 `target_link_libraries()` 将 `PkgConfig::PQXX` 目标链接起来，这个目标是由 `pkg_check_modules()` 导入的。注意，`PkgConfig::` 是一个固定的前缀，`PQXX` 从传递给宏的第一个参数派生出来。

使用这个选项比为没有 `CMake` 支持的依赖项创建 `find` 模块更快，但也有一些缺点。一它依赖于较旧的 `pkg-config` 工具，这可能在构建项目的操作系统中不可用。此外，这种方法创建了一个特殊案例，需要以与其它方法不同的方式进行维护。

我们讨论了如何与计算机上已安装的依赖项一起工作。然而，这只是故事的一部分。很多时候，项目将发送给可能没有所有所需依赖项的用户。让我们看看如何处理这种情况。

9.3. 使用系统中不存在的依赖项

CMake 在管理依赖项方面表现出色。若在系统尚未安装某些依赖项时，可以采取几种方法。如果使用的是 CMake 3.14 或更高版本，FetchContent 模块是管理依赖项的最佳选择。本质上，FetchContent 是 ExternalProject 的包装。不仅简化了流程，还增加了一些功能。我们将在本章后面深入探讨 ExternalProject。现在，只需知道两者之间的主要区别在于执行顺序就好：

- FetchContent 在配置阶段引入依赖项。
- ExternalProject 在构建阶段引入依赖项。

顺序很重要，因为在配置阶段由 FetchContent 定义的目标将位于同一个命名空间中，因此可以轻松在项目中使用。我们可以将它们与其他目标链接，就像是我们自定义的目标一样。但在极少数情况下，这不是我们所希望的，这时就需要使用 ExternalProject。

首先，了解如何处理大多数情况。

9.3.1. FetchContent

FetchContent 模块非常有用，提供以下功能：

- 管理外部项目的目录结构
- 从 URL 下载源代码（如果需要，还可以从存档中提取）
- 支持 Git、Subversion、Mercurial 和 CVS（并发版本系统）仓库
- 在需要时获取更新
- 使用 CMake、Make 或用户指定的工具配置和构建项目
- 提供对其他目标的嵌套依赖

使用 FetchContent 模块涉及以下三个主要步骤：

1. 使用 `include(FetchContent)` 将模块添加到项目中。
2. 使用 `FetchContent_Declare()` 命令配置依赖项。告知 FetchContent 依赖项的位置，以及应该使用哪个版本。
3. 使用 `FetchContent_MakeAvailable()` 命令完成依赖项设置。这将下载、构建、安装，并将列表文件添加到主项目以供解析。

为什么步骤 2 和 3 是分开的？是为了允许在多层次项目中进行配置覆盖。例如，一个依赖于外部库 A 和 B 的项目。库 A 也依赖于 B，但其作者使用的是一个与父项目版本不同的旧版本（图 9.1）：

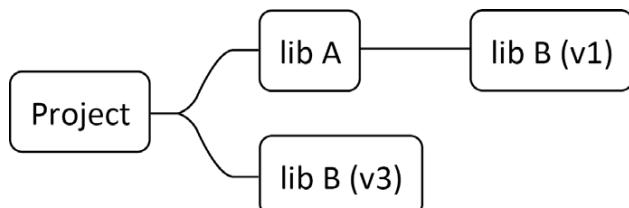


图 9.1：层次化项目结构

如果配置和下载发生在同一个命令中，即使新版本向后兼容，父项目也无法使用新版本，因为依赖项已经为旧版本配置了导入的目标，这会引入库的目标名称和文件冲突。

为了指定所需的版本，最顶层的项目必须在库 A 完全设置之前调用 FetchContent_Declare() 命令，并为 B 提供覆盖配置。因为 B 依赖项已经配置，所以将忽略随后在 A 中调用的 FetchContent_Declare()。

来看看 FetchContent_Declare() 命令的签名：

```
FetchContent_Declare(<depName> <contentOptions>...)
```

depName 是依赖项的唯一标识符，稍后将由 FetchContent_MakeAvailable() 命令使用。

contentOptions 提供了依赖项的详细配置，这可能相当复杂。重要的是，在底层，FetchContent_Declare() 使用较旧的 ExternalProject_Add() 命令。事实上，提供给 FetchContent_Declare 的大多数参数都会直接转发给内部调用。在详细解释所有参数之前，先看一个从 GitHub 下载依赖项的工作示例。

基本的 YAML 读取器示例

我编写了一个小程序，它从 YAML 文件中读取用户名并在欢迎消息中打印出来。YAML 是一种很好的、简单的格式，用于存储人类可读的配置，但对于机器来说解析起来相当复杂。我找到了一个整洁的小项目，名为 yaml-cpp，由 Jesse Beder 创建(<https://github.com/jbeder/yaml-cpp>)，解决了这个问题。

这个例子相当简单。它是一个输出欢迎消息的问候程序。name 的默认值将是 Guest，但可以通过 YAML 配置文件指定不同的名称。以下是 C++ 代码：

ch09/05-fetch-content/main.cpp

```
1 #include <iostream>
2 #include <yaml-cpp/yaml.h>
3
4
5 using namespace std;
6 int main() {
7     string name = "Guest";
8
9     YAML::Node config = YAML::LoadFile("config.yaml");
10    if (config["name"])
11        name = config["name"].as<string>();
12
13    cout << "Welcome " << name << endl;
14    return 0;
15 }
```

这个例子的配置文件只有一行：

ch09/05-fetch-content/config.yaml

```
name: Rafal
```

我们将在其他部分重用这个示例，所以花点时间理解其是如何工作的。现在已经准备好了代码，来看看如何构建并获取依赖：

ch09/05-fetch-content/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(ExternalProjectGit CXX)
3 add_executable(welcome main.cpp)
4 configure_file(config.yaml config.yaml COPYONLY)
5 include(FetchContent)
6 FetchContent_Declare(external-yaml-cpp
7     GIT_REPOSITORY https://github.com/jbeder/yaml-cpp.git
8     GIT_TAG 0.8.0
9 )
10 FetchContent_MakeAvailable(external-yaml-cpp)
11 target_link_libraries(welcome PRIVATE yaml-cpp::yaml-cpp)
```

可以显式访问由 yaml-cpp 库创建的目标，我们将使用 CMakePrintHelpers 帮助模块：

```
1 include(CMakePrintHelpers)
2 cmake_print_properties(TARGETS yaml-cpp::yaml-cpp
3                         PROPERTIES TYPE SOURCE_DIR)
```

当构建项目时，配置阶段将打印以下输出：

```
Properties for TARGET yaml-cpp::yaml-cpp:
yaml-cpp.TYPE = "STATIC_LIBRARY"
yaml-cpp.SOURCE_DIR = "/tmp/b/_deps/external-yaml-cpp-src"
```

这告诉我们由 external-yaml-cpp 依赖定义的目标存在；它是一个静态库，其源目录位于构建树内。这个打印输出对于实际项目来说不是必需的，但如果不确定如何正确包含导入的目标，它可以帮助调试。

由于我们已经使用 `configure_file()` 命令将.yaml 文件复制到输出，我们可以运行程序：

```
~/examples/ch09/05-fetch-content$ ./tmp/b/welcome
Welcome Rafal
```

一切运行顺利！不费吹灰之力，就引入了一个外部依赖，并在我们的项目中使用了它。

如果需要多个依赖，应该编写多个对 `FetchContent_Declare()` 命令的调用，每次选择一个唯一的标识符。但不需要多次调用 `FetchContent_MakeAvailable()`，它支持多个标识符（这些标识符不区分大小写）：

```
1 FetchContent_MakeAvailable(lib-A lib-B lib-C)
```

现在，再来了解一下如何编写依赖声明。

下载依赖项

`FetchContent_Declare()` 命令提供了广泛的选择，这些选择来自 `ExternalProject` 模块。可以执行三个操作：

- 下载依赖项
- 更新依赖项
- 补丁依赖项

首先，看看最常见的场景：从互联网上获取文件。

CMake 支持多种下载源：

- HTTP 服务器 (URL)
- Git
- Subversion
- Mercurial
- CVS

从列表顶部开始，首先探索如何从 URL 下载依赖项，并自定义流程以满足我们的需求。

更新和打补丁

可以提供一个 URL 列表，按顺序扫描，直到下载成功。CMake 将识别下载的文件是否是存档，并默认解压它。

基本声明：

```
FetchContent_Declare(dependency-id
                      URL <url1> [<url2>...]
)
```

以下是一些其他选项，用于进一步自定义此方法：

- `URL_HASH <algo>=<hashValue>`： 检查会通过，生成的下载文件的校验和是否与提供的 `<hashValue>` 匹配。推荐使用此选项以保证下载的完整性。以下算法受支持： MD5, SHA1, SHA224, SHA256, SHA384, SHA512, SHA3_224, SHA3_256, SHA3_384 和 SHA3_512
- `DOWNLOAD_NO_EXTRACT <bool>`： 这会显式禁用下载后的解压，可以在后续步骤中通过访问 `<DOWNLOADED_FILE>` 变量来使用下载文件的文件名。
- `DOWNLOAD_NO_PROGRESS <bool>`： 这会显式禁用下载进度的日志记录。
- `TIMEOUT <seconds>` 和 `INACTIVITY_TIMEOUT <seconds>`： 这些选项设置超时，以在固定的总时间或非活动期后终止下载。
- `HTTP_USERNAME <username>` 和 `HTTP_PASSWORD <password>`： 这些选项配置 HTTP 认证。请注意不要硬编码凭据。

- `HTTP_HEADER <header1> [<header2>...]`: 这会向 HTTP 请求添加头部信息，这对于 AWS 或自定义令牌很有用。
- `TLS_VERIFY <bool>`: 这会验证 SSL 证书。如果未设置，CMake 将从 `CMAKE_TLS_VERIFY` 变量中读取此设置，该变量默认设置为 `false`。跳过 TLS 验证是一种不安全的、不良的做法，尤其是在生产环境中应避免。
- `TLS_CAINFO <file>`: 这提供了到权威文件的路径；如果未指定，CMake 将从 `CMAKE_TLS_CAINFO` 变量中读取此设置。如果你的公司有颁发自签名 SSL 证书，这就很有用了。

大多数开发者会参考像 GitHub 这样的在线仓库，来获取最新版本的库。以下是操作方法。

从 Git 下载

要从 Git 下载依赖项，请确保系统安装了 Git 1.6.5 或更高版本。以下选项对于从 Git 克隆项目至关重要：

```
FetchContent_Declare(dependency-id
    GIT_REPOSITORY <url>
    GIT_TAG <tag>
)
```

`<url>` 和 `<tag>` 应与 git 命令兼容。生产环境中，建议使用特定的 git 哈希（而不是标签）以确保生成二进制文件的可追溯性，并避免不必要的 git fetch 操作。如果更喜欢使用分支，请坚持使用 `origin/main` 这样的远程名称。这确保了本地克隆的正确同步。

其他选项包括：

- `GIT_REMOTE_NAME <name>`: 这设置了远程名称（默认为 `origin`）。
- `GIT_SUBMODULES <module>...`: 这指定要更新的子模块；从 3.16 版本开始，此值默认为 `none`（之前，所有子模块都会更新）。
- `GIT_SUBMODULES_RECURSE 1`: 这启用子模块的递归更新。
- `GIT_SHALLOW 1`: 这执行浅克隆，由于跳过下载历史提交，因此速度更快。
- `TLS_VERIFY <bool>`: 这会验证 SSL 证书。如果未设置，CMake 将从 `CMAKE_TLS_VERIFY` 变量中读取此设置，该变量默认设置为 `false`；跳过 TLS 验证是一种不安全的、不良的做法，尤其是在生产环境中应避免。

如果依赖项存储在 Subversion 中，也可以使用 CMake 获取。

从 Subversion 下载

```
FetchContent_Declare(dependency-id
    SVN_REPOSITORY <url>
    SVN_REVISION -r<rev>
)
```

此外，可以提供以下内容：

- SVN_USERNAME <user> 和 SVN_PASSWORD <password>: 这些提供检出和更新的凭据，避免在项目中硬编码这些。
- SVN_TRUST_CERT <bool>: 这会跳过对 Subversion 服务器站点证书的验证。只有当服务器的网络路径，及其完整性是可信的时，才使用此选项。

Subversion 与 CMake 配合使用非常简单。Mercurial 也是如此。

从 Mercurial 下载

这种模式非常直接，需要提供两个参数就可以了：

```
FetchContent_Declare(dependency-id
    HG_REPOSITORY <url>
    HG_TAG <tag>
)
```

最后，可以使用 CVS 来提供依赖项。

从 CVS 下载

要从 CVS 检出模块，需要提供以下三个参数：

```
FetchContent_Declare(dependency-id
    CVS_REPOSITORY <cvsroot>
    CVS_MODULE <module>
    CVS_TAG <tag>
)
```

这样，我们就了解了 FetchContent_Declare() 的所有下载选项。CMake 支持在成功下载后执行的其他步骤。

更新和打补丁

默认情况下，更新步骤将重新下载外部项目的文件，如果下载方法支持更新，例如：配置了指向 main 或 master 分支的 Git 依赖项。可以通过以下两种方式覆盖此行为：

- 提供在更新期间执行的定制命令，使用 UPDATE_COMMAND <cmd>。
- 完全禁用更新步骤（以允许在没有网络连接的情况下构建）-UPDATE_DISCONNECTED <bool>。请注意，依赖项仍然会在第一次构建时下载。

另一方面，补丁是一个可选步骤，将在更新获取后执行。要启用它，需要指定要执行的确切命令，使用 PATCH_COMMAND <cmd>。

CMake 文档警告，某些补丁可能比其他补丁更“粘”。例如，Git 更改的文件在更新期间不会恢复到原始状态，需要避免对文件进行两次补丁。理想情况下，补丁命令应该是健壮且幂等（任意多次执行所产生的影响均与一次执行的影响相同）的。

可以链接更新和补丁命令：

```
FetchContent_Declare(dependency-id
    GIT_REPOSITORY <url>
```

```
    GIT_TAG <tag>
    UPDATE_COMMAND <cmd>
    PATCH_COMMAND <cmd>
)
```

下载依赖项在尚未在系统上时很有帮助，但它们已经在了呢？如何使用本地版本？

尽可能使用已安装的依赖项

从版本 3.24 开始，CMake 引入了一个特性，允许 FetchContent 在依赖项已经本地可用时跳过下载。要启用此功能，只需在声明中添加 FIND_PACKAGE_ARGS 关键字：

```
FetchContent_Declare(dependency-id
    GIT_REPOSITORY <url>
    GIT_TAG <tag>
    FIND_PACKAGE_ARGS <args>
)
```

这个关键字指示 FetchContent 模块在启动下载之前使用 `find_package()` 函数。如果本地找到包，就将使用该包，不会发生下载或构建。注意，这个关键字将消耗所有后续参数，所以应该是命令中的最后一个。

这里是更新先前示例的方法：

ch09/06-fetch-content-find-package/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(ExternalProjectGit CXX)
3
4 add_executable(welcome main.cpp)
5 configure_file(config.yaml config.yaml COPYONLY)
6
7 include(FetchContent)
8 FetchContent_Declare(external-yaml-cpp
9     GIT_REPOSITORY      https://github.com/jbeder/yaml-cpp.git
10    GIT_TAG            0.8.0
11    FIND_PACKAGE_ARGS NAMES yaml-cpp
12 )
13 FetchContent_MakeAvailable(external-yaml-cpp)
14 target_link_libraries(welcome PRIVATE yaml-cpp::yaml-cpp)
15 include(CMakePrintHelpers)
16 cmake_print_properties(TARGETS yaml-cpp::yaml-cpp
17                         PROPERTIES TYPE SOURCE_DIR
18                         INTERFACE_INCLUDE_DIRECTORIES
19 )
```

我们做了两个关键更改：

1. 添加了 `FIND_PACKAGE_ARGS` 和 `NAMES` 关键字，以指定正在寻找 `yaml-cpp` 包。如果没有 `NAMES`，CMake 会默认使用 `dependency-id`，在本例中为 `external-yaml-cpp`。

2. 打印属性中添加了 `INTERFACE_INCLUDE_DIRECTORIES`。这是一次性检查，可以手动验证是否在使用已安装的包，或者是否下载了一个新的包。

测试之前，请确保包实际上已经安装在系统上。如果没有安装，可以使用以下命令安装：

```
git clone https://github.com/jbeder/yaml-cpp.git
cmake -S yaml-cpp -B build-dir
cmake --build build-dir
cmake --install build-dir
```

有了这个设置，就可以构建我们的项目。如果一切顺利，应该会看到来自 `cmake_print_properties()` 命令的调试输出。这将表明我们正在使用本地版本，如 `INTERFACE_INCLUDE_DIRECTORIES` 属性所示。记住，这个输出是特定于环境，其结果可能会有所不同。

```
--  
Properties for TARGET yaml-cpp::yaml-cpp:  
yaml-cpp::yaml-cpp.TYPE = "STATIC_LIBRARY"  
yaml-cpp::yaml-cpp.INTERFACE_INCLUDE_DIRECTORIES =  
"/usr/local/include"
```

如果不使用 CMake 3.24，或者想支持使用旧版本的用户，请考虑手动运行 `find_package()` 命令。这样，就只会下载尚未安装的包：

```
1 find_package(yaml-cpp QUIET)
2 if (NOT TARGET yaml-cpp::yaml-cpp)
3     # download missing dependency
4 endif()
```

无论选择哪种方法，首先尝试使用本地版本，只有在依赖项找不到时才下载，可以提供最佳的用户体验。

在 `FetchContent` 引入之前，CMake 有一个更简单的模块称为 `ExternalProject`。尽管 `FetchContent` 在大多数情况下是推荐的选择，但 `ExternalProject` 仍然有其自身的用处。

9.3.2. ExternalProject

如前所述，在 `FetchContent` 引入 CMake 之前，有一个模块执行类似的功能：`ExternalProject`（在 3.0.0 版本中添加），用于从在线仓库获取外部项目。多年来，该模块逐渐扩展以满足不同的需求，导致 `ExternalProject_Add()` 命令变得相当复杂。

`ExternalProject` 模块在构建阶段填充依赖项。这与 `FetchContent` 不同，后者在配置阶段执行。由于这个区别，`ExternalProject` 不能像 `FetchContent` 那样将目标导入项目。另一方面，`ExternalProject` 可以直接将依赖项安装到系统中，执行测试，以及其他事情，例如：覆盖配置和构建所使用的命令。

某些情况下，由于使用这个遗留模块需要大量开销，可将其视为一种好奇心。我们主要在这里介绍它，以展示当前方法是如何从它演变而来的。

ExternalProject 提供了一个 ExternalProject_Add 命令来配置依赖项。

这里是一个例子：

```
1 include(ExternalProject)
2 ExternalProject_Add(external-yaml-cpp
3     GIT_REPOSITORY    https://github.com/jbeder/yaml-cpp.git
4     GIT_TAG          0.8.0
5     INSTALL_COMMAND  ""
6     TEST_COMMAND     ""
7 )
```

它与 FetchContent_Declare 非常相似。示例中有两个关键字：INSTALL_COMMAND 和 TEST_COMMAND。这个例子中，用于抑制依赖项的安装和测试，它们通常在构建期间执行。ExternalProject 执行许多步骤，这些步骤都可以进行深入的配置，并且按以下顺序执行：

1. mkdir: 为外部项目创建一个子目录。
2. download: 从仓库或 URL 下载项目文件。
3. update: 如果 fetch 方法支持，则下载更新。
4. patch : 执行一个修改下载文件的补丁命令。
5. configure: 执行配置阶段。
6. build: 为 CMake 项目执行构建阶段。
7. install: 安装 CMake 项目。
8. test: 执行测试。

对于每个步骤（除了 mkdir），可以通过添加 <STEP>_COMMAND 关键字来覆盖默认行为。还有许多其他选项——请参阅在线文档以获取完整的参考。如果出于某种原因想使用这种方法，而非推荐的 FetchContent，可以通过在 CMake 内部执行 CMake 来应用一种技巧，以导入目标。有关更多详细信息，请查看本书存储库中的 ch09/05-external-project 代码示例。

通常，我们会依赖库在系统中的可用性。如果不可用，会求助于 FetchContent，这种方法特别适合于那些小且快速编译的依赖项。

对于更重要的库（如 Qt），这种方法可能会很耗时。这时，提供预编译库的包管理器更适合用户的环境。虽然像 Apt 或 Conan 这样的工具提供了解决方案，但这要么太系统特定，要么复杂，不适合本书介绍。好消息是，只要提供了清晰的安装说明，大多数用户可以安装项目可能需要的依赖项。

9.4. 总结

本章为提供了使用 CMake 的查找模块识别系统安装的软件包的知识，以及如何利用随库提供的配置文件。对于不支持 CMake 但包含 .pc 文件的旧库，可以使用 PkgConfig 工具和 CMake 内置的 FindPkgConfig 查找模块。

还探讨了 FetchContent 模块的功能。这个模块允许在配置 CMake 时从各种来源下载依赖项，同时首先扫描系统，从而避免不必要的下载。我们提到了这些模块的历史背景，并讨论了在特殊情况下使用 ExternalProject 模块的选项。CMake 设计为通过讨论的方法定位库时自动生成构建目标，这为过程增加了一层便利和优雅。

有了这个基础，就可以将标准库整合到项目中了。

下一章中，我们将学习如何使用 C++20 模块，在较小规模上提供可重用的代码。

9.5. 扩展阅读

- CMake 文档 - 提供的查找模块：

<https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html#findmodules>

- CMake 文档 - 使用依赖项指南：

<https://cmake.org/cmake/help/latest/guide/using-dependencies/index.html>

- CMake 和使用 git-submodule 管理依赖项目：

<https://stackoverflow.com/questions/43761594/>

- 利用 PkgConfig：

<https://gitlab.kitware.com/cmake/community/-/wikis/doc/tutorials/How-To-Find-Libraries#piggybacking-on-pkg-config>

- 如何使用 ExternalProject：

<https://www.jwlawson.co.uk/interest/2020/02/23/cmake-external-project.html>

- CMake FetchContent 与 ExternalProject 的比较：

<https://www.scivision.dev/cmake-fetchcontent-vs-external-project/>

- 使用 CMake 与外部项目：

<http://www.saoe.net/blog/using-cmake-with-external-projects/>

第 10 章 使用 C++20 模块

C++20 引入了一个新特性：模块，可以用模块文件替代了头文件中的纯文本符号声明，该模块文件将预编译为二进制格式，减少了构建时间。

我们将讨论 CMake 中 C++20 模块的内容，从 C++20 模块作为一个概念开始：相对于标准头文件的优点，以及如何简化源码单元的管理。尽管简化构建过程令人兴奋，但本章强调了其采纳的道路既困难又漫长。

理论部分结束后，我们将继续讨论在项目中实现模块的实际方面：将讨论在早期版本的 CMake 中启用它们的实验性支持，以及在 CMake 3.28 中的完整发布。

通过 C++20 模块的旅程不仅仅是为了理解一个新特性——是关于重新思考在大型 C++ 项目中组件如何交互。本章结束时，不仅会了解模块的理论知识，还能通过示例获得实践经验，增强利用该特性实现更好的项目成果。

本章中，包含以下内容：

- C++20 模块是什么？
- 使用 C++20 模块支持的编写项目
- 配置工具链

10.1. 环境准备

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch10>找到本章中出现的代码文件。

尝试本章中的示例，需要以下工具链：

- CMake 3.26 或更新版本（推荐 3.28）
- 生成器：
 - Ninja 1.11 及更新版本（Ninja 和 Ninja Multi-Config）
 - Visual Studio 17 2022 及更新版本
- 编译器：
 - MSVC 工具集 14.34 及更新版本
 - Clang 16 及更新版本
 - GCC 14（针对 2023 年 9 月 20 日之后的开发分支）及更新版本

如果熟悉 Docker，可以使用第 1 章中引入的全套工具镜像。

要构建本章提供的示例，请使用以下命令：

```
cmake -B <build tree> -S <source tree> -G "Ninja" -D CMAKE_CXX_COMPILER=clang++-18 && cmake  
→ --build <build tree>
```

请确保将占位符 <build tree> 和 <source tree> 替换为适当的路径。

10.2. C++20 模块

三年前有写过关于如何使用 C++ 模块文章，尽管模块已经为 C++20 规范的一部分，但 C++ 生态系统的支持仍然没有准备好适配这个特性。幸运的是，自从本书的第一版以来，很多事情都发生了变化，随着 CMake 3.28 的发布，C++20 模块得到了正式支持（尽管从 3.26 版本开始就已经有了实验性支持）。

三年似乎实现一个特性是很长的时间，但必须要记住，这不只是取决于 CMake。许多碎片必须汇集在一起并良好工作。首先，需要编译器理解如何处理模块，然后构建系统如 GNU Make 或 Ninja 必须能够与模块一起工作，只有这样 CMake 才能使用这些新机制来提供对模块的支持。

这说明，并不是每个人都会使用最新的兼容工具，即使是现在，当前的支持仍然处于早期阶段。这些限制使得模块不适合大多数人们。所以也许不要急于依赖它们，来构建生产级别的项目。

尽管如此，如果你是尖端解决方案的爱好者，那么将会得到一次享受！如果可以严格控制项目的构建环境，例如：使用专用机器或构建容器化（Docker 等），可以在内部使用模块。只需小心行事，并理解该方式可能会有所不同。可能会有一个时刻，会因为一个工具的缺失或对特性的错误实现，而完全放弃模块。

C++ 构建的上下文中，“模块”是一个非常重的词。我们之前在本书中讨论过 CMake 的模块：find 模块、工具模块等。而 C++ 模块与 CMake 模块无关，它是 C++20 版本中添加的语言的原生特性。

在其核心，一个 C++ 模块是个单一源文件，将头文件和实现文件的功能封装成一个连贯的代码单元。其包括两个主要部分：

- 二进制模块接口（BMI）类似于头文件的目的，但它是二进制格式，当其他翻译单元使用时，会减少了重新编译的需求。
- 模块实现单元提供模块的实现、定义和内部细节。其内容不能从模块外部直接访问，有效地封装了实现细节。

引入模块是为了减少编译时间，解决预处理器和传统头文件的一些问题。来看看在典型的传统项目中，多个翻译单元如何粘合在一起。

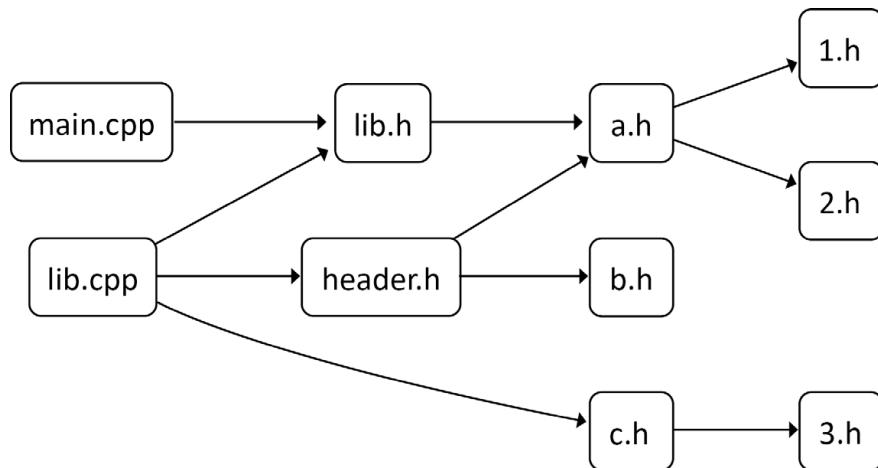


图 10.1：使用传统头文件的项目结构

前面的图显示了预处理器如何遍历项目树来构建程序。正如第 7 章中，为了构建每个翻译单元，预处理器机械地将文件拼接在一起，所以会生成一个包含所有由预处理器展开（包含的头文

件) 的长文件。这样, `main.cpp` 将先包含自己的源码, 然后是 `lib.h`、`a.h`、`1.h` 和 `2.h` 的内容。只有然后编译器才会启动并开始解析每一个字符以生成二进制目标文件。这样做本身并没有错, 直到我们意识到为了编译 `lib.cpp`, `main.cpp` 中包含的头文件必须重新编译。并且这种冗余会随着每个翻译单元的添加而增加。

传统头文件还有其他问题:

- 需要包含保护, 忘记时会导致问题。
- 循环引用的符号需要前置声明。
- 对头文件进行小的更改, 需要重新编译所有翻译单元。
- 预处理器宏可能难以调试和维护。

模块解决了其中的许多问题, 但一些问题仍然相关: 模块与头文件一样, 可以相互依赖。当一个模块导入另一个模块时, 仍然需要按照正确的顺序编译它们, 从最内层的模块开始。因为模块的尺寸往往要大得多, 所以这通常不是一个大问题。许多情况下, 整个库可以存储在单个模块中。

来看看模块在实际中如何编写和使用。这个简单的例子中, 我们只返回两个参数的和:

`ch10/01-cxx-modules/math.cpp`

```
1 export module math;
2 export int add(int a, int b) {
3     return a + b;
4 }
```

一条语句开始, 就告诉程序的其余部分这确实是一个名为 `math` 的模块。然后跟着一个用 `export` 关键字指定为, 可以从模块外部访问的常规函数定义。

Note

模块文件的扩展名与常规 C++ 源代码的不同。这是一个约定俗成的问题, 不应影响代码的处理方式。最好是基于将使用的工具链来选择扩展名:

- `.ixx` 是 MSVC 的扩展名。
- `.cppm` 是 Clang 的扩展名。
- `.cxx` 是 GCC 的扩展名。

要使用这个模块, 需要在程序中导入:

`ch10/01-cxx-modules/main.cpp`

```
1 import math;
2
3 #include <iostream>
4
5 int main() {
6     std::cout << "Addition 2 + 2 = " << add(2, 2) << std::endl;
7     return 0;
```

导入 `math` 语句足以将模块中导出的符号，直接引入主程序。现在可以在 `main()` 函数的主体中使用 `add()` 函数。从表面上看，模块与头文件非常相似。但是，若尝试像往常一样编写 CMake 列表文件，将无法成功地构建项目。那么，是时候介绍使用 C++ 模块的必要步骤了。

10.3. 使用 C++20 模块支持编写项目

本书主要讨论 CMake 3.26，但 CMake 经常更新，版本 3.28 就在本章完成前就发布了。如果使用的是这个版本或更新的版本，可以通过将 `cmake_minimum_required()` 命令设置为 VERSION 3.28.0 来访问使用最新功能。

另一方面，如果坚持使用旧版本，或者想要迎合可能尚未升级的更广泛的受众，需要启用实验性支持才能在 CMake 中使用 C++20 模块。

让我们探讨如何做到这一点。

10.3.1. CMake 3.26 和 3.27 中启用实验性支持

实验性支持代表一种协议：作为开发者，承认这个功能尚未准备好投入生产，应仅用于测试目的。要签署这样的协议，需要在项目的列表文件中，将 `CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API` 变量设置为 CMake 版本的一个特定值。

Note

CMake 的官方 Kitware 仓库托管了一个问题跟踪器，可以在其中搜索标签 area:cxxmodules。直到 3.28 版本发布，只报告了一个问题（在 3.25.0 中），这是潜在稳定功能的一个很好的指标。如果决定启用实验，请构建项目，以确认它适用于用户。

以下是在 CMake 的仓库和文档中可以找到的标志：

- 3c375311-a3c9-4396-a187-3227ef642046 对应 3.25（无正式文件）
- 2182bf5c-ef0d-489a-91da-49dbc3090d2a 对应 3.26
- aa1f7df0-828a-4fcd-9afc-2dc80491aca7 对应 3.27

如果不能访问 CMake 3.25，那就头痛了，模块在该版本 之前不可用。此外，如果 CMake 版本早于 3.27，还需要设置一个变量来为模块启用动态依赖：

```
1 set(CMAKE_EXPERIMENTAL_CXX_MODULE_DYNDEP 1)
```

以下是如何为当前版本自动选择正确的 API 密钥，以及明确不支持版本构建（这个例子中，将只支持 CMake 3.26 及以上的版本）。

ch10/01-cxx-modules/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.26.0)
2 project(CXXModules CXX)
3
4 # turn on the experimental API
5 if(CMAKE_VERSION VERSION_GREATER_EQUAL 3.28.0)
6     # Assume that C++ sources do import modules
7     cmake_policy(SET CMP0155 NEW)
8 elseif(CMAKE_VERSION VERSION_GREATER_EQUAL 3.27.0)
9     set(CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API
10         "aa1f7df0-828a-4fcf-9afc-2dc80491aca7")
11 elseif(CMAKE_VERSION VERSION_GREATER_EQUAL 3.26.0)
12     set(CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API
13         "2182bf5c-ef0d-489a-91da-49dbc3090d2a")
14     set(CMAKE_EXPERIMENTAL_CXX_MODULE_DYNDEP 1)
15 else()
16     message(FATAL_ERROR "Version lower than 3.26 not supported")
17 endif()

```

逐条解释：

- 首先，检查版本是否为 3.28 或更新。允许使用 `cmake_policy()` 启用 CMP0155 策略。如果想支持 3.28 以下的版本，这是必需的。
- 如果不是这种情况，将检查版本是否高于 3.27。如果是，将设置适当的 API 密钥。
- 如果不低于 3.27，将检查它是否高于 3.26。如果是这样，设置适当的 API 密钥，并启用实验性 C++20 模块动态依赖标志。
- 如果版本低于 3.26，项目不支持，并将输出致命错误消息通知用户。

这使我们能够支持从 3.26 开始的 CMake 版本范围。如果有幸在每个将构建项目的环境中运行 CMake 3.28，则上述 `if()` 块是不必要的。那么什么是必要的呢？

10.3.2. 启用对 CMake 3.28 及以上的支持

要从 3.28 开始使用 C++20 模块，必须明确声明这个版本为最低版本：

```

1 cmake_minimum_required(VERSION 3.28.0)
2 project(CXXModules CXX)

```

如果将最低要求版本设置为 3.28 或以上，将默认启用 CMP0155 策略。继续阅读以了解在定义模块之前需要配置的东西。如果需要 3.27 或更低的版本，即使项目是使用 CMake 3.28 或更新的版本构建的，构建也可能会失败。

接下来要考虑的是编译器要求。

10.3.3. 设置编译器要求

无论是使用 CMake 3.26、3.27、3.28 还是更新的版本来构建，为了使用 C++ 模块创建解决方案，都需要设置两个全局变量。第一个是禁用不受支持的 C++ 扩展，第二个是确保编译器支持所需的标准。

ch10/01-cxx-modules/CMakeLists.txt (续)

```
1 # Libc++ has no support compiler extensions for modules.  
2 set(CMAKE_CXX_EXTENSIONS OFF)  
3 set(CMAKE_CXX_STANDARD 20)
```

由于支持模块的编译器数量非常有限，设置标准可能看起来有些多余。然而，这对于保护项目未来不受影响是一种良好的习惯。

总体配置相当简单，到此结束。我们现在可以在 CMake 中定义一个模块。

10.3.4. 声明一个 C++ 模块

CMake 模块定义利用了 `target_sources()` 命令和 `FILE_SET` 关键字：

```
1 target_sources(math  
2     PUBLIC FILE_SET CXX_MODULES TYPE CXX_MODULES FILES math.cppm  
3 )
```

这里，引入了一种新的文件集类型：`CXX_MODULES`。这种类型默认情况下只在 CMake 3.28 及以后的版本中支持。对于 3.26，需要启用实验性 API。如果没有适当的支持，将会出现如下错误消息：

```
CMake Error at CMakeLists.txt:25 (target_sources):  
target_sources File set TYPE may only be "HEADERS"
```

如果在构建输出中看到这个，请检查代码是否正确。如果使用的版本的 API 密钥值不正确，也会出现这个消息。

如前所述，在同一个二进制文件中使用模块定义模块是有好处的。

但在创建库时，这些优势更为明显。这样的库可以在其他项目中使用，或者在同一项目中让其他库使用，从而进一步增强模块化。

要声明模块并将其与主程序链接，使用以下 CMake 配置：

ch10/01-cxx-modules/CMakeLists.txt (continued)

```
1 add_library(math  
2 target_sources(math  
3     PUBLIC FILE_SET CXX_MODULES FILES math.cppm  
4 )
```

```
5 target_compile_features(math PUBLIC cxx_std_20)
6 set_target_properties(math PROPERTIES CXX_EXTENSIONS OFF)
7
8 add_executable(main main.cpp)
9 target_link_libraries(main PRIVATE math)
```

为了确保这个库可以在其他项目中使用，必须使用 `target_compile_features()` 命令并明确要求 `cxx_std_20`。此外，还需要在目标级别重复设置 `CXX_EXTENSIONS OFF`。如果没有这个，CMake 将生成错误并停止构建。这似乎有些多余，可能会在 CMake 的未来版本中得到解决。

项目设置完成后，终于到了构建它的时候了。

10.4. 配置工具链

根据 Kitware 网站上的博客文章（见“扩展阅读”部分），CMake 最早在 3.25 版本就支持模块功能。尽管 3.28 版本正式支持此功能，但这并不是我们要享受模块便利性的唯一拼图。

下一个要求集中在构建系统上：需要支持动态依赖。截至目前，只有两个选择：

- Ninja 1.11 及更新版本（Ninja 和 Ninja Multi-Config）
- Visual Studio 17 2022 及更新版本

同样，编译器需要以特定格式生成映射源依赖的文件，以供 CMake 使用。这种格式在 Kitware 开发者撰写的一篇论文中有描述，这篇论文称为 p1589r5。该论文已提交给所有主流编译器以供实施。目前，只有以下三种编译器实现了所需的格式：

- Clang 16
- Visual Studio 2022 17.4 (19.34) 中的 MSVC
- GCC 14（针对开发分支，2023 年 9 月 20 日之后）及更新版本

假设环境中所有必要的工具（可以使用为本书提供的 Docker 镜像），并且 Make 项目已准备好构建，剩下的就是配置 CMake 以使用所需的工具链。

```
cmake -B <build tree> -S <source tree> -G "Ninja"
```

此命令将配置项目以使用 Ninja 构建系统。下一步是设置编译器。如果默认编译器不支持模块，并且已安装了另一个编译器来尝试，可以通过定义全局变量 `CMAKE_CXX_COMPILER` 来实现，如下所示：

```
cmake -B <build tree> -S <source tree> -G "Ninja" -D CMAKE_CXX_COMPILER=clang++-18
```

我们选择 Clang 18，因为它是撰写本文时（包含在 Docker 镜像中）可用的最新版本。成功配置后（会看到一些关于实验性功能的警告），需要构建项目：

```
cmake --build <build tree>
```

和往常一样，确保用适当的路径替换占位符 `<build tree>` 和 `<source tree>`。如果一切顺利，可以运行程序，观察模块功能按预期工作：

```
$ ./main  
Addition 2 + 2 = 4
```

就这样，C++20 模块在实际中得以应用。

Note

“扩展阅读”部分包括来自 Kitware 的博客文章和关于 C++ 编译器的源依赖格式的提案，提供了更多关于 C++20 模块部署和使用的深入见解。

10.5. 总结

本章中，我们深入探讨了 C++20 模块，了解了它们与 CMake 模块的区别，并代表了 C++ 在简化编译和解决与冗余头文件编译，及处理预处理器宏相关挑战方面的进步。

我们通过一个简单的示例演示了如何编写和导入 C++20 模块。然后，探讨了如何为 C++20 模块设置 CMake。由于这个特性是实验性的，需要设置特定的变量，提供了一系列条件语句来确保项目能够正确配置正在使用的 CMake 版本。关于必要的工具，我们强调构建系统必须支持动态依赖，目前的选择是 Ninja 1.11 或更新版本。对于编译器支持，Clang 16 和 Visual Studio 2022 17.4 (19.34) 中的 MSVC 适合完全支持 C++20 模块，而 GCC 的支持仍在等待中。此外，还指导各位通过配置 CMake 来使用选定的工具链，包括选择构建系统生成器和设置编译器版本。配置和构建项目后，可以运行程序来查看 C++20 模块的实际应用。

下一章中，我们将了解自动化测试的重要性及其应用，以及 CMake 对测试框架的支持。

10.6. 扩展阅读

- 描述新特性的博客文章

<https://www.kitware.com/import-cmake-c20-modules/>

- 针对 C++ 编译器的建议依赖格式：

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1689r5.html>

第 11 章 测试框架

经验丰富的专业人士知道，测试必须自动化。这是几年前有人向他们解释的，或者通过艰难的方式学到的。对于没有经验的程序员来说，这一做法并不那么明显，看起来像是额外的、不必要的劳动，并不会带来太多价值。这是可以理解的：当某人刚开始编写代码时，还没有创建真正复杂的解决方案，也没有在大型的代码库上工作。很可能，是自己项目的唯一开发者。这些早期项目很少需要超过几个月就能完成，因此很难看到代码在较长时间内是如何恶化的。

所有这些因素都导致人们认为编写测试是浪费时间和精力。编程新手可能会告诉自己，每次进行构建和运行流程时，实际上确实在测试代码。毕竟，已经手动确认了代码能够正常工作，并做到了预期效果。所以，是时候继续下一个任务了，对吧？自动化测试确保新的更改不会无意中破坏程序。本章中，将学习为什么测试很重要，以及如何使用 CTest 来协调测试执行。CTest 可以查询可用的测试，过滤执行，随机排序，重复执行，并设置时间限制。我们将探讨如何使用这些功能，控制 CTest 的输出，并处理测试失败。

接下来，将修改项目的结构以适应测试，并创建自己的测试运行器。

介绍了基本原理之后，将继续添加流行的测试框架：Catch2 和 GoogleTest（也称为 GTest），以及其模拟库。最后，将介绍使用 LCOV 进行详细的测试覆盖率报告。

本章中，将包含以下内容：

- 为什么自动化测试值得麻烦？
- 使用 CTest 在 CMake 中标准化测试
- 为 CTest 创建最基本的单元测试
- 单元测试框架
- 生成测试覆盖率报告

11.1. 示例下载

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch11>找到本章中出现的代码文件。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保用适当的路径替换占位符 <build tree> 和 <source tree>。提醒一下，<build tree> 是目标/输出目录的路径，而 <source tree> 是源码所在的位置。

11.2. 为什么要自动化测试？

一条生产线上，一台机器在钢板上打孔。这些孔需要特定的大小和形状，以便为成品安装螺栓。生产线的设计者会设置好机器，测试这些孔，然后继续下一步。最终，某些事情会发生变化：钢材可能更厚，工人可能调整了孔的大小，或者因为设计变更需要打更多的孔。一个聪明的设计者会在

关键点安装质量控制检查，以确保产品符合规格。孔是如何形成的并不重要：钻孔、冲孔，还是激光切割。

同样的原则也适用于软件开发。很难预测哪些代码能够多年保持稳定，哪些将经历多次修订。随着软件功能的扩展，必须确保不会无意中破坏已有的东西。我们也会犯错误。即使是最优秀的开发者也无法预见每一次更改的所有影响。开发者经常要处理他们最初并未编写的代码，可能并不了解所有背后的假设。他们会阅读代码，形成心理模型，进行更改，并希望一切顺利。当这种方式不奏效时，修复错误可能需要数小时或数天的时间，并且会对产品和用户产生负面影响。

有时候，会遇到难以理解的代码。甚至可能开始责怪别人造成了混乱，结果发现是自己造成的。这种情况通常发生在编写代码时过于匆忙，没有完全理解问题的情况下。

作为开发者，我们不仅受到项目截止日期或有限预算的压力；有时候还需要在夜间醒来修复关键问题。令人惊讶的是，一些不那么明显的错误是如何在代码审查中溜掉的呢？

自动化测试可以预防大多数这些问题。这些测试是代码片段，用于验证另一段代码的行为是否正确。顾名思义，每当有人进行更改时，这些测试会自动运行，通常作为构建过程的一部分。它们通常为一个步骤，以确保在将代码合并到仓库之前，保证代码的质量。

有人可能会为了节省时间而跳过创建自动化测试，但这将是一个代价高昂的错误。正如史蒂文·赖特所说：“经验是在你真正需要之后才获得的东西。”除非正在编写一次性脚本或进行实验，否则不要跳过测试。可能会因为精心编写的代码不断测试失败而感到沮丧，但一个失败的测试意味着刚刚避免在生产环境中引入一个重大问题。现在花在测试上的时间，将节省以后在修复错误上的时间——晚上能睡得更香。并且，测试也不是难以添加和维护东西。

11.3. 使用 CTest 在 CMake 中标准化测试

最终，自动化测试不过是运行一个可执行文件，将系统置于测试状态（SUT），执行想要测试的操作，并检查结果是否符合预期。可以视为一种结构化的方式来完成“GIVEN_<CONDITION>_WHEN_<SCENARIO>_THEN_<EXPECTED-OUTCOME>”，并验证这对于 SUT 是否成立。一些资源建议按照这种非常方式命名你的测试函数：例如，`GIVEN_4_and_2_WHEN_Sum_THEN_returns_6`。

实现和执行这些测试的方法有很多，取决于选择的框架、如何将其与 SUT 连接，以及相应的设置。对于一个首次接触项目的用户来说，即使是测试二进制文件这样的小细节，也会影响他们的体验。由于没有标准的命名约定，开发者可能会将他们的测试可执行文件命名为 `test_my_app`，另一个可能会选择 `unit_tests`，第三个可能会选择更不直观的名称，或者完全跳过测试。弄清楚要运行哪个文件、使用哪个框架、传递哪些参数，以及如何收集结果都是用户想要避免的麻烦。

CMake 通过一个单独的 `ctest` 命令行工具解决了这个问题。通过项目作者通过列表文件配置，提供了一种标准化的运行测试方式。这个统一的界面适用于使用 CMake 构建的每个项目。遵循这个标准，将享受到其他好处：将项目集成到持续集成/持续部署（CI/CD）中变得更加容易，测试在 IDE（如 Visual Studio 或 CLion）中显示得更方便。最重要的是，只需最小努力就能获得一个健壮的测试运行工具。

那么，如何在已经配置的项目中使用 CTest 运行测试呢？需要选择以下三种操作模式之一：

- 仪表盘模式

- 测试模式
- 构建并测试模式

仪表盘模式会将测试结果发送到一个名为 CDash 的单独工具，也来自 Kitware。CDash 收集并展示软件质量测试结果，在一个易于导航的仪表盘中。这个主题对于非常大的项目很有用，但超出了本书的范围。

测试模式的命令行如下：

```
ctest [<options>]
```

此模式下，应在使用 CMake 构建项目后，在构建树中运行 CTest。有许多选项可用，但在深入讨论之前，需要解决一个小的不便：必须在构建树中运行 ctest 二进制文件，并且只有在项目构建之后才能运行。

为了简化操作，CTest 提供了构建并测试模式。首先探讨这个模式，这样稍后可以全神贯注于测试模式。

11.3.1. 构建并测试模式

要使用此模式，需要执行 ctest 后跟--build-and-test：

```
ctest --build-and-test <source-tree> <build-tree>
  --build-generator <generator> [<options>...]
  [--build-options <opts>...]
  [--test-command <command> [<args>...]]
```

这是测试模式的一个简单包装，接受构建配置选项和--test-command 参数后的测试命令。重要的是，除非在--test-command 后包含 ctest 关键字，否则不会运行任何测试：

```
ctest --build-and-test project/source-tree /tmp/build-tree --buildgenerator "Unix Makefiles"
→ --test-command ctest
```

此命令中，指定源和构建路径，并选择一个构建生成器。所有这三个都是必需的，并遵循第 1 章中详细描述的 cmake 命令的规则。

可以添加更多参数，这些参数通常分为以下三个类别：配置控制、构建过程或测试设置。

配置阶段的参数如下：

- --build-options ——为 cmake 配置包含选项。在--test-command 之前放置，必须放在最后。
- --build-two-config ——对 CMake 运行两次配置阶段。
- --build-nocmake ——跳过配置阶段。
- --build-generator-platform ——提供生成器特定的平台。
- --build-generator-toolset ——提供生成器特定的工具集。
- --build-makeprogram ——为基于 Make 或 Ninja 的生成器指定 make 可执行文件。

构建阶段的参数如下：

- `--build-target` ——指定要构建的目标。
- `--build-noclean` ——构建 `clean` 目标之前进行构建。
- `--build-project` ——命名正在构建的项目。

测试阶段的参数如下：

- `--test-timeout` ——设置测试的时间限制，以秒为单位。

现在可以配置测试模式，方法是在`--test-command cmake` 后添加参数，或者直接运行测试模式。

11.3.2. 测试模式

构建项目后，可以在构建目录中使用 `ctest` 命令运行测试。如果使用构建并测试模式，就会一起完成。在没有其他标志的情况下运行 `ctest`，通常足以满足大多数情况。如果所有测试都成功，`ctest` 将返回 0 的退出代码（在类 Unix 系统中），可以在 CI/CD 中验证，以防止将故障更改合并到生产分支。

编写好的测试可能与编写生产代码本身一样具有挑战性，将系统置于特定的状态（SUT），运行单个测试，然后销毁 SUT 实例。这个过程相当复杂，可能会产生各种问题：跨测试污染、时间并发干扰、资源争用、死锁导致的冻结执行，以及其他许多问题。

CTest 提供了多种选项来缓解这些问题，可以控制哪些测试运行、执行顺序、生成的输出、时间限制和重复率，以及其他方面。接下来的部分将提供必要的上下文和最有用选项的简要概述。

查询测试

我们可能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`-N` 选项，禁用执行并只打印列表：

```
# ctest -N
Test project /tmp/b
Test #1: SumAddsTwoInts
Test #2: MultiplyMultipliesTwoInts
Total Tests: 2
```

使用`-N` 与下一节中描述的过滤器一起使用，以检查当应用过滤器时哪些测试会执行。

如果需要一个可以自动化工具消费的 JSON 格式，可以执行 `ctest` 并使用`--show-only=json-v1`。

CTest 还提供了一个使用 `LABELS` 关键字来分组测试的机制，列出所有可用的标签（无需实际执行测试），请使用`--print-labels`。这个选项在手动定义测试时非常有用，例如在列表文件中使用 `add_test()` 命令，然后就可以通过测试属性指定个别标签：

```
1 set_tests_properties(<name> PROPERTIES LABELS "<label>")
```

然而，来自各种框架的自动化测试方法可能不支持这种标签。

过滤测试

有时可能只想运行特定测试而不是整个套件。例如，正在调试一个失败的单个测试，没有必要运行其他所有测试。还可以使用这种机制来为大型项目跨多台机器分发测试。

这些标志将根据提供的正则表达式 (regex) 过滤测试：

- `-R <r>`, `--tests-regex <r>` - 只运行 `<r>` 与匹配测试名称的测试
- `-E <r>`, `--exclude-regex <r>` - 跳过与 `<r>` 匹配测试名称的测试
- `-L <r>`, `--label-regex <r>` - 只运行与 `<r>` 匹配标签的测试
- `-LE <r>`, `--label-exclude <regex>` - 跳过与 `<r>` 匹配标签的测试

高级场景可以通过使用`--tests-information` 选项（或更短的`-I` 形式）来实现。此选项用逗号分隔的格式 `<start>,<end>,<step>,<test-IDs>` 的范围，可以省略其他字段但保留逗号。`<test-IDs>` 选项是一个逗号分隔的测试序号的列表。例如：

- `-I 3,,` 跳过测试 1 和 2（执行从第三个测试开始）
- `-I ,2,` 只运行第一个和第二个测试
- `-I 2,,3` 每行运行第三个测试，从第二行开始
- `-I ,0,,3,9,7` 只运行第三个、第九个和第七个测试

还可以将这些范围指定在一个文件中，以在分布式方式上在多台机器上执行非常庞大的测试套件。当与`-R` 一起使用`-I` 时，只有满足两个条件的测试才会运行。如果想运行满足任一条件的测试，请使用`-U` 选项。如前所述，可以使用`-N` 选项来检查过滤的结果。

打乱测试

编写单元测试可能会遇到一些意想不到的问题，其中一个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 的状态来影响另一个测试。换句话说，第一个执行的测试可能会“泄漏”其状态，并污染第二个测试。这种耦合是坏消息，因为它引入了测试之间的未知、隐式关系。

更糟糕的是，这种错误往往隐藏在测试场景的复杂性中。可能会在随机失败的情况下检测到，但相反的情况也同样可能：一个不正确的状态，可能会导致测试在没有错误的情况下通过。这些错误的测试会给人一种安全感的错觉，这比完全没有测试还要糟糕。认为代码正确测试的假设可能，会鼓励更大胆的行动，导致意外的结果。

发现这类问题的一个方法是，独立运行每个测试。通常，当直接从测试框架执行测试运行器而没有 CTest 时，这并不是情况。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。这允许检测那些在套件中通过，但在单独执行时失败的测试。

另一方面，CTest 通过隐式地在子 CTest 实例中执行每个测试用例，有效地消除了所有基于内存的测试交叉污染。甚至可以更进一步，添加`--force-new-ctest-process` 选项来强制使用单独的进程。

不幸的是，如果测试使用了外部、争用的资源，如 GPU、数据库或文件，那么仅凭这一点可能无法奏效。可以采取的另一项预防措施是随机化测试执行的顺序，引入这种变化通常足以最终检测出假性通过的测试。CTest 支持这一策略，可通过`--schedule-random` 选项进行打乱。

处理失败

这里有一个著名的约翰·C·麦克斯韦的名言：“尽早失败，经常失败，但总是向前失败。”向

前失败意味着从我们的错误中学习，这就是我们运行单元测试（以及在生活的其他领域）时想要做的事情。除非在运行测试时附带调试器，否则很难发现自己犯了什么错误，因为 CTest 会保持简洁，只列出失败的测试，而不会实际打印它们的输出。

测试用例或 SUT 打印到标准输出的消息可能非常有价值，以确定确切出了什么问题。要看到它们，可以运行 `ctest` 并使用`--output-on-failure`，或设置 `CTEST_OUTPUT_ON_FAILURE` 环境变量也会有同样的效果。

根据解决方案的大小，测试失败后停止执行可能有意义。这可以通过向 `ctest` 提供`--stop-on-failure` 参数来实现。

CTest 会存储失败的测试名称。为了节省长时间测试套件的时间，可以专注于这些失败的测试，跳过运行通过测试，直到问题解决。这一特性是通过`--rerun-failed` 选项启用的（其他过滤器都将忽略）。解决所有问题后，记得运行所有测试，以确保在此期间没有引入回归。

当 CTest 没有检测到任何测试时，可能有两种情况：要么测试不存在，要么项目存在问题。默认情况下，`ctest` 会打印一个警告消息并返回 0 的退出代码，以避免混淆。大多数用户都有足够的上下文来理解遇到了哪种情况，以及下一步该做什么。然而，在某些环境中，`ctest` 总是作为自动化流水线的一部分执行。这时，可能需要明确指出，缺乏测试应视为错误（并返回非零退出代码），可以通过`--no-tests=error` 参数来进行配置。对于相反的行为（无警告），请使用`--no-tests=ignore` 选项。

重复测试

职业生涯中，迟早会遇到那些大多数时间都运行正确的测试，我想强调的是“大多数”。偶尔，这些测试会因为环境原因而失败：例如，由于错误地模拟时间、事件循环问题、异步执行的处理不当、并行性、哈希冲突，以及其他在每次运行中都不会发生的非常复杂的场景。这些不可靠的测试称为易碎测试。

这种不一致似乎是一个不太重要的问题。我们可能会说，测试不是一个真正的生产环境，这就是为什么它们有时会失败的原因。这种说法确实有一定的道理：测试不是为了复现每一个细节。测试是一种模拟，是对可能发生的事情的近似，这通常就足够了。如果在下次运行时会通过，那么重新运行测试会有什么伤害呢？实际上，这确实有伤害。

有三个主要关注点，如下所述：

- 如果代码库中有足够多的易碎测试，将成为平滑交付代码变更的严重障碍。当急于回家在周五下午，或者急于向客户交付一个严重问题的关键修复时，这尤其令人沮丧。
- 不能真正确定易碎测试是否因为测试环境的不完善而失败。可能是相反的情况：它们失败是因为复现了一个已经在生产环境中发生的罕见场景。这还没有明显到足以发出警报……但已经足够了。
- 不是测试易碎，而是你的代码！环境有时会出问题——作为开发者，我们以确定性的方式处理这些问题。如果 SUT 以这种方式行为，那就是一个严重的错误的迹象——代码可能会从未初始化的内存中读取。

没有完美的方法来解决所有上述情况——可能的原因太多了。然而，我们可以通过多次运行它们并使用`-repeat <mode>:<#>option` 选项来增加识别易碎测试的机会。有三种模式可供选择：

- `until-fail` ——运行测试 `<#>` 次；所有运行都必须通过。

- `until-pass` ——最多运行测试 `<#>` 次；它必须至少通过一次。这适用于处理已知易碎但过于复杂和重要而无法调试或禁用的测试。
- `after-timeout` ——最多运行测试 `<#>` 次，但仅在测试超时时才重试。在繁忙的测试环境中使用。

一个普遍的建议是尽快调试易碎测试，或者如果不能信任产生一致的结果，就尽快消除它们。

控制输出

将每一条信息都打印到屏幕上会变得非常繁忙。`CTest` 减少了噪音，并将执行的测试输出收集到日志文件中，只在常规运行中提供最有用的信息。当事情出错测试失败时，可以期望一个总结，以及如果启用了`--output-on-failure`，可能还会有一些日志，如之前所述。

从经验中了解，“足够的信息”的确足够，直到它不再足够。有时，可能还想看到通过的测试的输出，以检查是否真的在正常工作（而不是悄悄地停止而没有错误）。要获取更详细的输出，可以添加`-V` 选项（或`--verbose`，如果想在自动化流水线中明确表示）。如果这还不够，可能会需要使用`-VV` 或`--extra-verbose`。对于极其深入的调试，可以使用`-debug`（但要做好准备，可能会出现带有所有细节的文本墙）。

如果寻找相反的情况，`CTest` 也提供了“禅模式”，通过`-Q` 或`--quiet` 启用。那时不会有任何输出（可以停止担心，学会爱上这个 bug）。这个选项似乎除了让人困惑之外没有其他用途，但要注意输出仍然会存储在测试文件中（默认情况下存储在`./Testing/Temporary`）。自动化流水线可以检查退出代码是否为非零值，并收集日志文件进行进一步处理，而不会在主要输出中添加可能使不熟悉产品的开发者困惑的细节。

要将日志存储在特定路径，请使用`-O <file>`，`--output-log <file>` 选项。如果输出过长，有两个限制选项可以限制每个测试的字节数：`--test-output-size-passed` 和`--test-output-size-failed <size>`。

其他选项

还有一些其他选项可以满足日常测试需求：

- `-C <cfg>`, `--build-config <cfg>` ——指定要测试的配置。调试配置通常具有调试符号，使事情更容易理解，因为重优化选项可能会影响 SUT 的行为，所以发布版本也应该测试。这个选项仅适用于多配置生成器。
- `-j <jobs>`, `--parallel <jobs>` ——设置并行执行的测试数量。这对于加快长时间测试的执行非常有用。在繁忙的环境中（共享的测试运行器上），这可能会由于调度而产生不利影响。这可以通过下一个选项稍作缓解。
- `--test-load <level>` ——以一种方式安排并行测试，使得 CPU 负载不超过值（尽最大努力）。
- `--timeout <seconds>` ——指定单个测试的默认时间限制。

现在，已经了解了在许多不同场景下如何执行 `ctest`，接下来让我们学习如何添加一个简单的测试。

11.4. 为 CTest 创建最基本的单元测试

编写单元测试从技术上讲，可以不使用任何类型的框架。我们所要做的就是创建想要测试的类的实例，执行其一个方法，并检查新的状态或返回的值是否符合我们的预期。然后，报告结果并删除测试的对象。

将使用以下结构：

```
- CMakeLists.txt
- src
|- CMakeLists.txt
|- calc.cpp
|- calc.h
|- main.cpp
- test
|- CMakeLists.txt
|- calc_test.cpp
```

从 `main.cpp` 开始，看到其使用了 `Calc` 类：

ch11/01-no-framework/src/main.cpp

```
1 #include <iostream>
2 #include "calc.h"
3 using namespace std;
4 int main() {
5     Calc c;
6     cout << "2 + 2 = " << c.Sum(2, 2) << endl;
7     cout << "3 * 3 = " << c.Multiply(3, 3) << endl;
8 }
```

没什么太花哨的——`main.cpp` 只是包含了 `calc.h` 头文件，并调用了 `Calc` 对象的两个方法。让我们快速地看一下 SUT（测试系统）`Calc` 的接口：

ch11/01-no-framework/src/calc.h

```
1 #pragma once
2 class Calc {
3     public:
4     int Sum(int a, int b);
5     int Multiply(int a, int b);
6 };
```

这个接口尽可能简单。这里使用 `#pragma once`——工作原理与常见的预处理器包含保护完全一样，尽管它不是官方标准的一部分。

Note

包含保护是头文件中的一行简短的代码，用于防止在同一个父文件中多次包含。

看看类的实现：

ch11/01-no-framework/src/calc.cpp

```
1 #include "calc.h"
2 int Calc::Sum(int a, int b) {
3     return a + b;
4 }
5 int Calc::Multiply(int a, int b) {
6     return a * a; // a mistake!
7 }
```

哎呀！我们引入了一个错误！`Multiply` 忽略了 `b` 参数，并返回了 `a` 的平方。这应该正确编写的单元测试检测到。所以，让我们来写一些：

ch11/01-no-framework/test/calc_test.cpp

```
1 #include "calc.h"
2 #include <cassert>
3 void SumAddsTwoIntegers() {
4     Calc sut;
5     if (4 != sut.Sum(2, 2))
6         std::exit(1);
7 }
8 void MultiplyMultipliesTwoIntegers() {
9     Calc sut;
10    if(3 != sut.Multiply(1, 3))
11        std::exit(1);
12 }
```

从 `calc_test.cpp` 文件开始编写两个测试方法，每个测试的 SUT 方法一个。如果从调用方法返回的值不符合预期，每个函数将调用 `std::exit(1)`。这里可以使用 `assert()`、`abort()` 或 `terminate()`，但这样会在 `ctest` 的输出中得到一个不太明确的子进程终止消息，而不是更易读的失败消息。

是时候创建一个测试运行器了。我们的将尽可能简单，以避免引入大量工作。看看为了运行两个测试而编写的 `main()` 函数：

ch11/01-no-framework/test/unit_tests.cpp

```
1 #include <iostream>
2 void SumAddsTwoIntegers();
3 void MultiplyMultipliesTwoIntegers();
```

```
4 int main(int argc, char *argv[]) {
5     if (argc < 2 || argv[1] == std::string("1"))
6         SumAddsTwoIntegers();
7     if (argc < 2 || argv[1] == std::string("2"))
8         MultiplyMultipliesTwoIntegers();
9 }
```

以下是发生的情况的分解：

1. 声明两个外部函数，这些函数将从一个翻译单元中链接。
2. 如果没有提供参数，执行两个测试（`argv[]` 中的第一个元素始终是程序名称）。
3. 如果第一个参数是测试的标识符，执行它。
4. 如果任何测试失败，将内部调用 `exit()` 并返回 1 退出代码。
5. 如果没有执行测试或所有测试都通过，将隐式返回 0 退出代码。

运行第一个测试，执行：

```
./unit_tests 1
```

运行第二个测试，执行：

```
./unit_tests 2
```

我们尽可能简化了代码，但它仍然难以阅读。可能需要维护这一部分的人，在添加了更多测试后，都不会感到轻松。这个功能相当原始——调试这样的测试将很困难。尽管如此，来看看如何使用 CTest 来使用它：

ch11/01-no-framework/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(NoFrameworkTests CXX)
3 include(CTest)
4 add_subdirectory(src bin)
5 add_subdirectory(test)
```

从常规的头文件开始，并 `include(CTest)`——启用了 CTest，并且应该在顶级 `CMakeLists.txt` 中始终这样做。接下来，在 `src` 和 `test` 每个子目录中包含两个嵌套的列表文件。指定的 `bin` 值表示我们希望将 `src` 子目录的二进制输出放置在 `<build_tree>/bin` 中。否则，二进制文件最终会出现在 `<build_tree>/src` 中，这可能会使用户感到困惑，因为构建工件不是源文件。

对于 `src` 目录，列表文件是直接的，并包含一个简单的 `main` 目标：

ch11/01-no-framework/src/CMakeLists.txt

```
1 add_executable(main main.cpp calc.cpp)
```

还需要为 test 目录创建一个列表文件：

ch11/01-no-framework/test/CMakeLists.txt

```
1 add_executable(unit_tests
2     unit_tests.cpp
3     calc_test.cpp
4     ../../src/calc.cpp)
5 target_include_directories(unit_tests PRIVATE ../../src)
6 add_test(NAME SumAddsTwoInts COMMAND unit_tests 1)
7 add_test(NAME MultiplyMultipliesTwoInts COMMAND unit_tests 2)
```

现在定义了第二个 unit_tests 目标，使用了 src/calc.cpp 实现文件及其相应的头文件。最后，明确添加了两个测试：

- SumAddsTwoInts
- MultiplyMultipliesTwoInts

每个测试都将其 ID 作为参数传递给 add_test() 命令。CTest 将简单地接受 COMMAND 关键字之后提供的内容，并在子 shell 中执行，收集输出和退出代码。不要过于依赖 add_test() 方法；在后面的单元测试框架部分，将了解处理测试用例的更好方法。

要运行测试，请在构建树中执行 ctest：

```
# 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，并报告其中一个失败了—Calc::Multiply 返回的值不符合预期，非常好。现在知道我们的代码有一个错误，需要有人修复它。

Note

迄今为止的大多数示例中，我们并不一定采用了第 4 章中描述的项目结构，这样做是为了保持简洁。这一章讨论了更高级的概念，使用完整结构是合理的。在项目（无论多小）中，最好从一开始就遵循这种结构。正如一位智者所说：“踏上这条路时，如果不保持警惕，就不知道会去到哪里。”

现在已经很清楚了，为项目从头开始构建测试框架并不是一个好主意。即使是最基本的示例也难以阅读，有很多开销，并且没有增加任何价值。然而，在我们采用单元测试框架之前，需要重新考虑项目的结构。

11.5. 为测试构建项目

C++ 具备有限的内省能力，但无法提供像 Java 那样强大的反射特性。这可能是为什么在 C++ 代码中编写测试和单元测试框架，比在其他功能更丰富的环境中更具挑战性的原因。这种有限方法的一个结果是，开发者需要更深入地参与编写可测试的代码。需要仔细设计接口，并考虑实际应用。例如，如何避免编译代码两次，并在测试和生产之间重用工件？

对于较小的项目来说，编译时间可能不是大问题，但随着项目的发展，缩短编译循环的需求仍然存在。前面的例子中，将所有 SUT 源文件包含在单元测试可执行文件中，除了 `main.cpp` 文件。如果仔细观察，会注意到该文件中的一些代码没有测试（即 `main()` 函数本身的内容）。编译代码两次会引入一个轻微的风险，即产生的工件可能不会完全相同。随着时间的推移，这些差异可能会逐渐增加，特别是在添加编译标志和预处理器指令时，如果贡献者急于完成、经验不足或不熟悉项目，这可能会带来风险。

这个问题有多种解决方案，但最直接的方法是将整个解决方案构建为一个库，并与单元测试链接。可能会有人想知道然后如何运行。那么就创建一个引导可执行文件，它与库链接并执行其代码。

首先，将当前的 `main()` 函数重命名为 `run()` 或 `start_program()`。然后，创建一个只包含新的 `main()` 函数的实现文件 (`bootstrap.cpp`)。这个函数充当适配器：唯一作用是提供一个入口点并调用 `run()`，传递命令行参数。将所有东西链接在一起后，最终会得到一个可测试的项目。

通过重命名 `main()`，现在可以将 SUT 与测试链接，并测试其 `main` 功能。否则，会违反第 8 章讨论的“单一定义规则”(ODR)，测试运行器也需要自己的 `main()` 函数。

注意，测试框架默认提供它自己的 `main()` 函数，它会自动检测所有链接的测试，并根据配置运行它们。

这种方法产生的工件可以分为以下目标：

- 包含生产代码的 sut 库
- 引导程序，其中包含调用 sut 中 `run()` 的 `main()` 包装器
- 单元测试，其中包含运行所有 sut 测试的 `main()` 包装器

下面的图表显示了目标之间的符号关系：

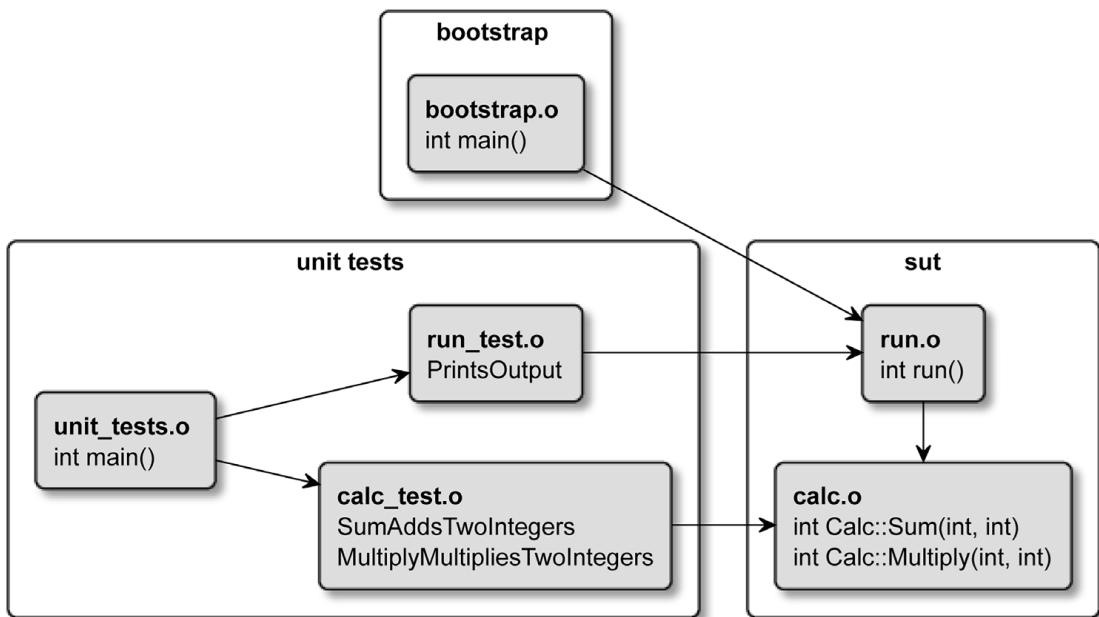


图 11.1：测试和生产可执行文件之间共享工件

最终得到六个实现文件，分别产生各自的（.o）对象文件：

- **calc.cpp**: 将要进行单元测试的 **Calc** 类。这称为单元测试对象（UUT），因为 UUT 是 SUT 的一个特化。
- **run.cpp**: 原始入口点重命名为 **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**: 新的 **run()** 测试可以放在这里。
- **unit_tests.o**: 单元测试的入口点，扩展为调用 **run()** 的测试。

我们即将构建的库不一定是静态或共享库。通过选择对象库，可以避免不必要的归档或链接。从技术上讲，使用动态链接 SUT 可以节省一些时间，但经常发现自己同时修改两个目标：测试和 SUT，这抵消了节省的时间。

让我们看看之前名为 **main.cpp** 的文件是如何变化的：

ch11/02-structured/src/run.cpp

```

1 #include <iostream>
2 #include "calc.h"
3 using namespace std;
4 int run() {
5     Calc c;
6     cout << "2 + 2 = " << c.Sum(2, 2) << endl;
7     cout << "3 * 3 = " << c.Multiply(3, 3) << endl;
8     return 0;
9 }

```

变化很小：文件和函数重命名，添加了一个返回语句，因为编译器不会隐式地为 **main()** 之

外的函数添加返回语句。

新的 `main()` 函数如下所示：

ch11/02-structured/src/bootstrap.cpp

```
1 int run(); // declaration
2 int main() {
3     run();
4 }
```

保持简单，我们声明链接器将提供来自另一个翻译单元的 `run()` 函数，并调用它。

接下来是 `src` 下的列表文件：

ch11/02-structured/src/CMakeLists.txt

```
1 add_library(sut STATIC calc.cpp run.cpp)
2 target_include_directories(sut PUBLIC .)
3 add_executable(bootstrap bootstrap.cpp)
4 target_link_libraries(bootstrap PRIVATE sut)
```

首先，创建一个 SUT 库，并将`.` 标记为 PUBLIC 包含目录，这样它就会传播到所有与 SUT 链接的目标（即 `bootstrap` 和 `unit_tests`）。请注意，包含目录相对于列表文件，允许使用点`(.)` 来引用当前的 `<source_tree>/src` 目录。

现在是更新 `unit_tests` 目标的时候了。我们将替换对`…/src/calc.cpp` 文件的直接引用，改为对 `sut` 的链接引用，用于 `unit_tests` 目标，还将为 `run_test.cpp` 文件中的主函数添加一个新的测试。为了简洁，我们将省略对此的讨论，但如果感兴趣，可以查看本书仓库中的示例。

同时，以下是整个测试列表文件：

ch11/02-structured/test/CMakeLists.txt

```
1 add_executable(unit_tests
2                 unit_tests.cpp
3                 calc_test.cpp
4                 run_test.cpp)
5 target_link_libraries(unit_tests PRIVATE sut)
6 add_test(NAME SumAddsTwoInts COMMAND unit_tests 1)
7 add_test(NAME MultiplyMultipliesTwoInts COMMAND unit_tests 2)
8 add_test(NAME RunOutputsCorrectEquations COMMAND unit_tests 3)
```

完成了！我们按需注册了新的测试。通过遵循这种做法，可以确保测试是在用于生产中的机器代码上执行的。

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表明它们是关于什么的。在实际项目中，应该选择与生产代码上下文（而不是测试）匹配的名称。例如，对于一个 `FooApp`，将目标命名为 `foo` 而不是 `bootstrap`，将 `lib_foo` 而不是 `sut`。

现在，我们知道了如何在适当的目标中构建一个可测试的项目，再将焦点转移到测试框架本身。我们不想手动将每个测试案例添加到列表文件中，对吧？

11.6. 单元测试框架

上一节展示了编写一个小的单元测试程序并不复杂。虽然可能不太美观，一些专业开发者喜欢重新发明轮子，认为他们的版本在各个方面都会更好。避免这个陷阱：最终会创建大量样板代码，以至于它本身就可以成为一个项目。使用流行的单元测试框架，可以使解决方案与多个项目和公司认可的标准保持一致，并且通常会有免费的更新和扩展，不会有任何损失。

如何将单元测试框架整合到项目中？当然，通过根据所选框架的规则实现测试，然后将这些测试与框架提供的测试运行器链接起来，测试运行器启动所选测试的执行并收集结果。与我们之前的 `unit_tests.cpp` 文件不同，许多框架会自动检测所有测试，并使其对 `CTest` 可见。

这一章中，我选择了两个单元测试框架来介绍：

- Catch2 相对易于学习，具有良好的支持和文档。提供了基本的测试用例，还包括用于行为驱动开发 (BDD) 的优雅宏。虽然可能缺少一些功能，但可以在需要时通过外部工具进行补充。访问它的主页：<https://github.com/catchorg/Catch2>。
- GoogleTest (GTest) 方便但更高级。提供了丰富的功能集，如各种断言、死亡测试，以及值和类型参数化的测试，甚至支持通过其 GMock 模块生成 XML 测试报告和模拟。可以在[这里找到它](https://github.com/google/googletest)：<https://github.com/google/googletest>。

框架的选择取决于学习偏好和项目大小。如果喜欢循序渐进，不需要完整的功能集，Catch2 是一个不错的选择。那些喜欢一头扎进去，并且需要全面工具集的人会发现 GoogleTest 更合适。

11.6.1. Catch2

这个框架由 Martin Hořeňovský 维护，非常适合初学者和小型项目。这并不是说它不能适应更大的应用程序，但能在某些领域需要其他工具。开始之前，先检查一下我们的 `Calc` 类的单元测试实现：

`ch11/03-catch2/test/calc_test.cpp`

```
1 #include <catch2/catch_test_macros.hpp>
2 #include "calc.h"
3
4 TEST_CASE("SumAddsTwoInts", "[calc]") {
```

```
5     Calc sut;
6     CHECK(4 == sut.Sum(2, 2));
7 }
8
9 TEST_CASE("MultiplyMultipliesTwoInts", "[calc]") {
10    Calc sut;
11    CHECK(12 == sut.Multiply(3, 4));
12 }
```

这几行比我们之前的例子更有力。CHECK() 宏不仅验证期望，收集所有失败的断言并一起呈现，以避免不断的重新编译。

最好的部分是什么？不需要手动将这些测试添加到列表文件中，以通知 CMake 它们的存在。忘记 add_test(); 不再需要它了，Catch2 将自动将测试注册到 CTest。按照上一节讨论的方式配置了项目，添加框架就很简单了。使用 FetchContent() 将其带入项目。

可以选择两个主要版本：Catch2 v2 和 Catch2 v3。版本 2 是单头文件库的遗留选项，适用于 C++11。版本 3 编译为静态库，需要 C++14。建议选择最新版本。

使用 Catch2 时，确保选择一个 Git 标签并在列表文件中固定它。通过主分支升级可能不会平稳进行。

Note

企业环境中，很可能会在 CI 流水线中运行测试。这时，请记住设置环境，使其已经安装了系统中的依赖项，这样每次构建时都不需要重新获取。正如第 9 章中所提到的，可以使用 FetchContent_Declare() 命令，并使用 FIND_PACKAGE_ARGS 关键字来使用系统中的包。

我们将在列表文件中使用 3.4.0 版本：

ch11/03-catch2/test/CMakeLists.txt

```
1 include(FetchContent)
2 FetchContent_Declare(
3     Catch2
4     GIT_REPOSITORY https://github.com/catchorg/Catch2.git
5     GIT_TAG v3.4.0
6 )
7 FetchContent_MakeAvailable(Catch2)
```

然后，需要定义我们的 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。由于 Catch2 提供了自己的 main() 函数，不再使用 unit_tests.cpp 文件（文件可以删除）。代码如下所示：

ch11/03-catch2/test/CMakeLists.txt (continued)

```
1 add_executable(unit_tests calc_test.cpp run_test.cpp)
2 target_link_libraries(unit_tests PRIVATE
```

```
sut Catch2::Catch2WithMain)
```

最后，使用 Catch2 模块定义的 `catch_discover_tests()` 命令自动从 `unit_tests` 检测所有测试用例，并使用 CTest 注册它们，如下所示：

ch11/03-catch2/test/CMakeLists.txt (continued)

```
1 list(APPEND CMAKE_MODULE_PATH ${catch2_SOURCE_DIR}/extras)
2 include(Catch)
3 catch_discover_tests(unit_tests)
```

完成！我们刚刚向我们的解决方案添加了一个单元测试框架。现在让看看实践中的效果。测试运行器的输出如下：

```
# ./test/unit_tests
unit_tests is a Catch2 v3.4.0 host application.
Run with -? for options
-----
MultiplyMultipliesTwoInts
-----
/root/examples/ch11/03-catch2/test/calc_test.cpp:9
.....
/root/examples/ch11/03-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
```

Catch2 能够将 `sut.Multiply(3, 4)` 表达式扩展为 9，给我们更多的上下文，这在调试中非常有帮助。

注意，直接执行运行器二进制文件（编译后的单元测试可执行文件）可能比使用 `ctest` 稍微快一些，但 CTest 提供的优势值得在速度上有些损失。

这就是 Catch2 的设置。如果将来需要添加更多测试，只需创建新的实现文件，并将它们的路径添加到 `unit_tests` 目标的源列表中。

Catch2 提供了各种功能，如事件监听器、数据生成器和微基准测试，但它缺乏内置的模拟功能。如果不熟悉模拟，将在下一节中讨论。可以使用以下模拟框架，将模拟添加到 Catch2 中：

- `FakeIt` (<https://github.com/eranpeer/FakeIt>)
- `Hippomocks` (<https://github.com/dascandy/hippomocks>)
- `Trompeloeil` (<https://github.com/rollbear/trompeloeil>)

然而，对于更流畅、更高级的体验，还有一个值得关注的框架，那就是 GoogleTest。

11.6.2. GoogleTest

使用 GoogleTest 有几个重要的优点：存在已久，在 C++ 社区中广受认可，因此多个 IDE 都支持。世界最大的搜索引擎公司维护并广泛使用，因此它不太可能过时或废弃。它可以测试 C++11 及以上版本，这对于在旧环境中工作的人来说是个好消息。

GoogleTest 仓库包含两个项目：GTest（主测试框架）和 GMock（一个添加模拟功能的库）。所以，可以通过一次 FetchContent() 调用下载两者。

Using GTest

为了使用 GTest，项目需要遵循“为测试结构化项目”一节中的指示。这是在这个框架中编写单元测试的方法：

ch11/04-gtest/test/calc_test.cpp

```
1 #include <gtest/gtest.h>
2 #include "calc.h"
3
4 class CalcTestSuite : public ::testing::Test {
5     protected:
6     Calc sut_;
7 };
8
9 TEST_F(CalcTestSuite, SumAddsTwoInts) {
10     EXPECT_EQ(4, sut_.Sum(2, 2));
11 }
12
13 TEST_F(CalcTestSuite, MultiplyMultipliesTwoInts) {
14     EXPECT_EQ(12, sut_.Multiply(3, 4));
15 }
```

由于这个例子也将用于 GMock，我选择将测试放在一个单一的 CalcTestSuite 类中。测试套件将相关测试组合在一起，以便可以重用相同的字段、方法、设置和清理步骤。要创建一个测试套件，声明一个新类，它从::testing::Test 继承，并在其受保护的部分放置可重用的元素。

测试套件内的每个测试用例都使用 TEST_F() 宏声明。对于独立的测试，存在一个更简单的 TEST() 宏。因为我们已经在类中定义了 Calc sut_，每个测试用例都可以将其作为 CalcTestSuite 的方法来访问。实际上，每个测试用例在继承自 CalcTestSuite 的自己的实例中运行，这就是为什么需要保护关键字的原因。注意，可重用的字段不是为了在连续测试之间共享数据；它们的存在是为了保持代码的简洁性。

GTest 不像 Catch2 那样提供自然断言的语法，可以使用显式的比较，例如 EXPECT_EQ()。按照惯例，预期值放在前面，实际值放在后面。GTest 有许多其他类型的断言、辅助函数和宏值得探索。关于 GTest 的详细信息，请参阅官方参考资料 (<https://google.github.io/googletest/>)。

要向项目中添加这个依赖，需要决定使用哪个版本。与 Catch2 不同，GoogleTest 倾向于“活在新分支”的哲学（起源于 GTest 依赖的 Abseil 项目）。它声明：“如果从源代码构建依赖项并遵循 API，就不应该遇到任何问题。”（请参阅扩展阅读部分，以获取更多详细信息。）如

果愿意遵循这个规则（并且从源代码构建不是问题），可以将 Git 标签设置为 master 分支。否则，就从 GoogleTest 仓库中选择一个版本。

Note

商业环境中，可能会在 CI 管道中运行测试。这种情况下，请记住设置环境，使其已经安装了系统中的依赖项，这样每次构建时都不需要重新获取它们。正如第 9 章所提到的，可以使用 FetchContent_Declare() 命令，并使用 FIND_PACKAGE_ARGS 关键字来使用系统中的包。

添加对 GTest 的依赖看起来是这样的：

ch11/04-gtest/test/CMakeLists.txt

```
1 include(FetchContent)
2 FetchContent_Declare(
3     gtest
4     GIT_REPOSITORY https://github.com/google/googletest.git
5     GIT_TAG v1.14.0
6 )
7 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
8 FetchContent_MakeAvailable(gtest)
```

遵循与 Catch2 相同的方法——执行 FetchContent() 并从源代码构建框架。唯一的区别是添加了 set(gtest...) 命令，这是 GoogleTest 作者建议的，以避免在 Windows 上覆盖父项目的编译器和链接器设置。

最后，可以声明测试运行器可执行文件，将其与 gtest_main 链接，并使用内置的 CMake GoogleTest 模块自动发现测试用例：

ch11/04-gtest/test/CMakeLists.txt (continued)

```
1 add_executable(unit_tests
2                 calc_test.cpp
3                 run_test.cpp)
4 target_link_libraries(unit_tests PRIVATE sut gtest_main)
5 include(GoogleTest)
6 gtest_discover_tests(unit_tests)
```

完成了 GTest 的设置。直接执行的测试运行器的输出比 Catch2 更详细，但可以通过传递--gtest_brief=1 来限制它只显示失败信息：

```
# ./test/unit_tests --gtest_brief=1
~/examples/ch11/04-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
```

幸运的是，即使输出很嘈杂，当从 CTest 运行时，也会抑制（除非明确启用它，通过 `ctest --output-on-failure` 命令行）。

既然已经有了框架，让我们讨论一下模拟。毕竟，当没有与其他元素紧密耦合时，没有测试可以真正称为“单元测试”。

GMock

编写纯粹的单元测试是关于执行一个与代码的其他部分隔离的代码片段。这样的测试单元必须是自包含的元素，要么是一个类，要么是一个组件。当然，几乎没有用 C++ 编写的程序的所有单元都能与其他单元清楚地隔离。

很可能，代码将严重依赖某些类之间的关联关系。唯一的麻烦是：这类类的对象将需要另一个类的对象，而那些对象将需要另一个。在了解之前，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，并依赖于其状态。例如，可能依赖于数据库中的特定记录、进入的网络数据包或磁盘上存储的特定文件。

为了测试目的解耦单元，开发人员使用相似测试或用于单元测试的特殊类版本。一些例子包括假对象、存根和模拟。以下是这些术语的粗略定义：

- 一个假对象是更复杂机制的有限实现，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、预制的答案，限于测试使用的响应，还可以记录调用了哪些方法，以及这些方法调用的次数。
- mock 是存根的一个稍微扩展版本，还将验证在测试期间是否按预期调用了方法。

相似测试在测试开始时创建，并作为参数传递给被测试类的构造函数，以代替真实对象。这种机制称为依赖注入。

简单相似测试的问题在于它们过于简单。为了模拟不同测试场景的行为，可能需要提供许多不同的用例，一个用于耦合对象可能处于的每个状态。这并不实用，并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：允许开发人员为特定类创建通用的相似测试，并为每个测试在线定义其行为。GMock 将这些相似测试称为“模拟”，但它们是上述所有相似测试的混合，并取决于场合。

让我们为 Calc 类添加一个功能，该功能将为提供的参数添加一个随机数，将由 `AddRandomNumber()` 方法表示，该方法返回这个和作为 int。如何确认返回值确实是一个随机数和提供的类的值的准确和？随机生成的数字对于许多重要过程至关重要，如果使用不正确，可能会有各种后果。检查所有可能的随机数，直到用尽所有可能性并不实用。

为了测试它，需要将随机数生成器封装在一个可以模拟（或换句话说，可以替换为模拟）的类中。模拟将允许强制一个特定的响应，这用于“伪造”随机数的生成。Calc 将使用该值在 `AddRandomNumber()` 中，并允许检查从该方法返回的值是否符合预期。随机数生成与其他单元的清晰分离是一种附加价值（因为能够交换生成器）。

让我们从抽象生成器的公共接口开始。这个头文件将允许在实际生成器和模拟中实现，并使它

们可以相互替换：

ch11/05-gmock/src/rng.h

```
1 #pragma once
2 class RandomNumberGenerator {
3     public:
4         virtual int Get() = 0;
5         virtual ~RandomNumberGenerator() = default;
6     };
```

实现这个接口的类将为我们提供 `Get()` 方法中的随机数。注意，虚拟关键字必须放在所有方法上，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拟析构函数。

接下来，需要扩展 `Calc` 类以接受和存储生成器，这样就可以在发布构建中提供真实的生成器，或者在测试中使用模拟：

ch11/05-gmock/src/calc.h

```
1 #pragma once
2 #include "rng.h"
3 class Calc {
4     RandomNumberGenerator* rng_;
5 public:
6     Calc(RandomNumberGenerator* rng);
7     int Sum(int a, int b);
8     int Multiply(int a, int b);
9     int AddRandomNumber(int a);
10 };
```

这里包含了头文件，并添加了一个提供随机添加的方法。此外，创建了一个字段来存储生成器的指针，以及一个参数化构造函数，这就是依赖注入在实践中的应用。现在，我们实现这些方法：

ch11/05-gmock/src/calc.cpp

```
1 #include "calc.h"
2 Calc::Calc(RandomNumberGenerator* rng) {
3     rng_ = rng;
4 }
5 int Calc::Sum(int a, int b) {
6     return a + b;
7 }
8 int Calc::Multiply(int a, int b) {
9     return a * b; // now corrected
10 }
11 int Calc::AddRandomNumber(int a) {
12     return a + rng_->Get();
13 }
```

构造函数中，正在将提供的指针分配给一个类字段。然后，使用这个字段在 AddRandomNumber() 中获取生成值。生产代码将使用一个真实的生成器；测试将使用模拟。记住，需要解引用指针以启用多态。作为一个额外的奖励，可以为不同的实现创建不同的生成器类。我只需要一个：Mersenne Twister 伪随机生成器，具有均匀分布：

ch11/05-gmock/src/rng_mt19937.cpp

```
1 #include <random>
2 #include "rng_mt19937.h"
3 int RandomNumberGeneratorMt19937::Get() {
4     std::random_device rd;
5     std::mt19937 gen(rd());
6     std::uniform_int_distribution<> distrib(1, 6);
7     return distrib(gen);
8 }
```

创建一个新实例对每个调用并不是非常有效，但可以满足这个简单的示例。目的是生成 1 到 6 之间的数字并返回给调用者。

这个类的头文件只提供了一个方法的签名：

ch11/05-gmock/src/rng_mt19937.h

```
1 #include "rng.h"
2 class RandomNumberGeneratorMt19937
3     : public RandomNumberGenerator {
4 public:
5     int Get() override;
6 };
```

并且在生产代码中：

ch11/05-gmock/src/run.cpp

```
1 #include <iostream>
2 #include "calc.h"
3 #include "rng_mt19937.h"
4 using namespace std;
5 int run() {
6     auto rng = new RandomNumberGeneratorMt19937();
7     Calc c(rng);
8     cout << "Random dice throw + 1 = "
9         << c.AddRandomNumber(1) << endl;
10    delete rng;
11    return 0;
12 }
```

我们已经创建了一个生成器，并将其指针传递给了 Calc 的构造函数。一切准备就绪，可以

开始编写我们的模拟。为了保持组织性，开发人员通常会将模拟放在一个单独的 test/mocks 目录中。为了防止歧义，头文件名称有一个_mock 后缀。

这是模拟的代码：

ch11/05-gmock/test/mocks/rng_mock.h

```
1 #pragma once
2 #include "gmock/gmock.h"
3 class RandomNumberGeneratorMock : public
4 RandomNumberGenerator {
5 public:
6     MOCK_METHOD(int, Get, (), (override));
7 }
```

添加了 gmock.h 头文件之后，可以声明我们的模拟。正如计划，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。这些宏告诉框架哪些接口的方法应该模拟。使用以下格式（需要使用大量括号）：

```
MOCK_METHOD(<return type>, <method name>,
           (<argument list>), (<keywords>))
```

我们已经准备好在测试套件中使用模拟（为了简洁，省略了之前的测试案例）：

ch11/05-gmock/test/calc_test.cpp

```
1 #include <gtest/gtest.h>
2 #include "calc.h"
3 #include "mocks/rng_mock.h"
4 using namespace ::testing;
5 class CalcTestSuite : public Test {
6 protected:
7     RandomNumberGeneratorMock rng_mock_;
8     Calc sut_{&rng_mock_};
9 };
10 TEST_F(CalcTestSuite, AddRandomNumberAddsThree) {
11     EXPECT_CALL(rng_mock_, Get()).Times(1).WillOnce(Return(3));
12     EXPECT_EQ(4, sut_.AddRandomNumber(1));
13 }
```

分解一下变化：在测试套件中添加了新的头文件，并创建了一个新的字段 rng_mock_。因为字段是在声明的顺序中初始化的 (rng_mock_ 先于 sut_)，可以将模拟的地址传递给 sut_ 的构造函数。

我们的测试案例中，对 rng_mock_ 的 Get() 方法调用了 GMock 的 EXPECT_CALL 宏。这告诉框架如果在执行过程中没有调用 Get() 方法，则测试失败。链式调用的 Times 明确指出为了通过测试，必须发生多少次调用。WillOnce 确定在方法调用后，模拟框架将执行的操作（返

回 3)。

通过使用 GMock，能够表达模拟行为与预期结果并行，这大大提高了可读性和测试的维护性。最重要的是，为每个测试案例提供了灵活性，能够通过一个表达式的语句来区分发生的事情。

最后，为了构建项目，需要确保 gmock 库与测试运行器链接。为了实现这一点，可以将它添加到 target_link_libraries() 列表中：

ch11/05-gmock/test/CMakeLists.txt

```
1 include(FetchContent)
2 FetchContent_Declare(
3     googletest
4     GIT_REPOSITORY https://github.com/google/googletest.git
5     GIT_TAG release-1.14.0
6 )
7 # For Windows: Prevent overriding the parent project's compiler/linker settings
8 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
9 FetchContent_MakeAvailable(googletest)
10 add_executable(unit_tests
11     calc_test.cpp
12     run_test.cpp)
13 target_link_libraries(unit_tests PRIVATE sut gtest_main gmock)
14 include(GoogleTest)
15 gtest_discover_tests(unit_tests)
```

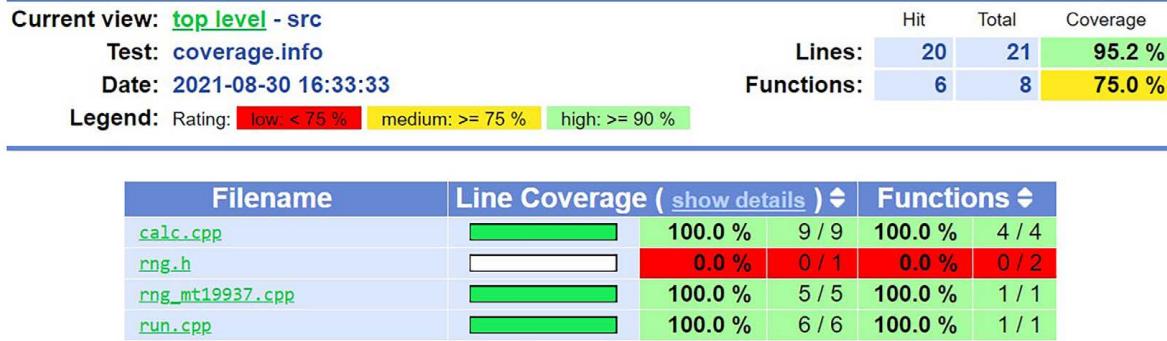
现在，可以享受 GoogleTest 框架的所有好处。GTest 和 GMock 都是高级工具，拥有多种概念、实用工具和帮助程序，适用于不同的情况。这个例子（尽管有点长）只是触及了可能性的表面。我鼓励各位读者将它们纳入自己的项目中，因为它们将极大地提高你的工作质量。GMock 的“Mocking for Dummies”页面是一个很好的起点（可以在扩展阅读部分找到这个链接）。

有了测试，应该以某种方式衡量哪些测试了，哪些没有测试到，并努力改善这种情况。最好使用自动化工具来收集和报告这些信息。

11.7. 生成测试覆盖率报告

向小解决方案中添加测试很简单，真正的困难出现在更高级和更长的程序中。多年来，我发现当我的代码行数接近 1,000 行时，逐渐难以追踪哪些行和分支在测试中执行。超过 3,000 行后，几乎变得不可能。大多数专业应用程序的代码量都会超过这个数量，许多经理用来协商解决技术债务的关键指标就是代码覆盖率，因此知道如何生成有用的报告，对于获取这些讨论的实际数据是有帮助的。为了解决这个问题，可以使用一个工具来理解，测试用例“覆盖”哪些代码行。这样的代码覆盖率工具会连接到测试系统 (SUT)，在测试期间收集每行代码的执行信息，并以方便的报告形式展示：

LCOV - code coverage report

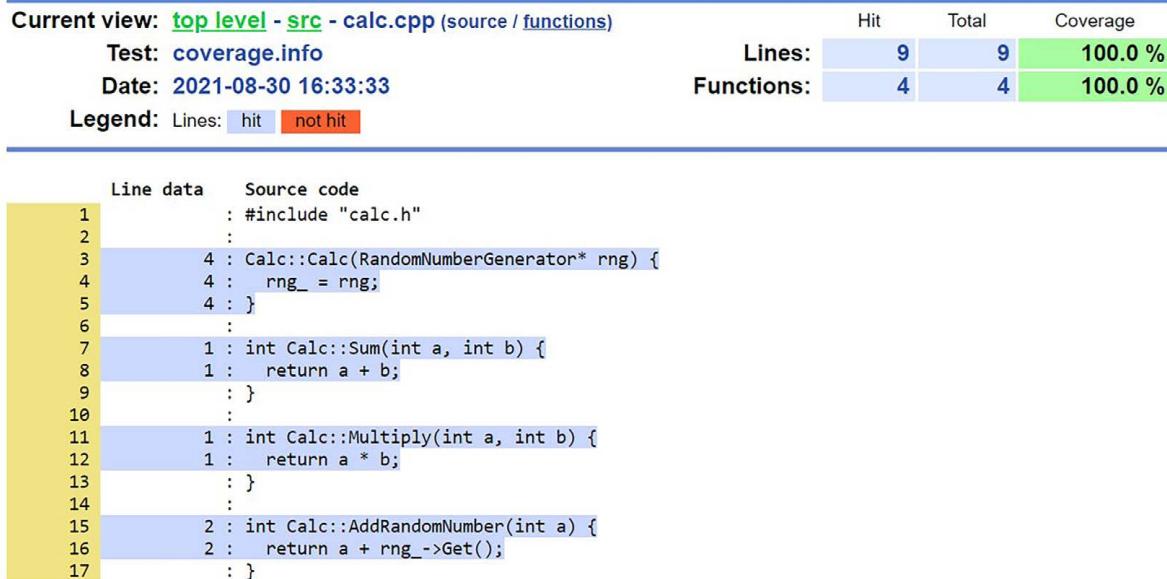


Generated by: [LCOV version 1.14](#)

图 11.2：由 LCOV 生成的代码覆盖率报告

这些报告将展示哪些文件被测试覆盖到，哪些没有。不仅如此，还可以深入查看每个文件的详细信息，准确了解哪些代码行被执行以及执行次数。以下屏幕截图中，行数据列显示 Calc 构造函数运行了 4 次，每次测试运行一次：

LCOV - code coverage report



Generated by: [LCOV version 1.14](#)

图 11.3：代码覆盖率报告的详细视图

生成类似报告的方法有很多，因平台和编译器而异，但通常遵循相同的流程：准备要测量的 SUT 并获取基线，测量，报告。

最简单的工具是 LCOV，是 gcov 的图形前端，gcov 是 GNU 编译器集合 (GCC) 中的一个覆盖率工具。来看看如何使用它。

11.7.1. 使用 LCOV 生成覆盖率报告

LCOV 将生成 HTML 覆盖率报告，并在内部使用 gcov 来测量覆盖率。如果使用的是 Clang，也不用担心——Clang 支持以这种格式生成度量。可以从 Linux 测试项目（Linux Test Project）维护的官方仓库（<https://github.com/linuxtest-project/lcov>）获取 LCOV，或者简单地使用包管理器，这是一个面向 Linux 的工具。

可以在 macOS 上运行，但不支持 Windows 平台。最终用户通常不关心测试覆盖率，所以可以在自己的构建环境中手动安装 LCOV，而不是将其纳入项目。

要测量覆盖率，需要执行以下操作：

- 启用代码覆盖率的编译器标志下，以 Debug 配置编译。这将生成覆盖率注释（.gcno）文件。
- 测试可执行文件与 gcov 库链接。
- 收集基线的覆盖率度量，不运行测试。
- 运行测试，这将创建覆盖率数据（.gcda）文件。
- 将信息收集到聚合信息文件中。
- 生成（.html）报告。

应该首先解释为什么代码必须以 Debug 配置编译。最重要的原因是，Debug 配置会使用 -O0 标志禁用所有优化。CMake 默认在 CMAKE_CXX_FLAGS_DEBUG 变量中这样做（在文档中都没有说明这一点）。除非决定覆盖这个变量，否则 Debug 构建应该是未优化的。这是为了防止内联和其他类型的隐式代码简化。否则，将难以追踪哪个机器指令来自源代码的哪一行。

第一步中，需要指示编译器向 SUT 添加必要的检测。要添加的确切标志是编译器特定的；然而，两个主要编译器（GCC 和 Clang）提供了相同的--coverage 标志来启用覆盖率检测，以生成与 GCC 兼容的 gcov 格式的数据。

以下是如何将覆盖率检测添加到上一节的示例 SUT：

ch11/06-coverage/src/CMakeLists.txt

```
1 add_library(sut STATIC calc.cpp run.cpp rng_mt19937.cpp)
2 target_include_directories(sut PUBLIC .)
3 if (CMAKE_BUILD_TYPE STREQUAL Debug)
4     target_compile_options(sut PRIVATE --coverage)
5     target_link_options(sut PUBLIC --coverage)
6     add_custom_command(TARGET sut PRE_BUILD COMMAND
7         find ${CMAKE_BINARY_DIR} -type f
8         -name '*.gcda' -exec rm {} +
9     )
10    add_executable(bootstrap bootstrap.cpp)
11    target_link_libraries(bootstrap PRIVATE sut)
```

逐步分解上面的代码：

1. 使用 if(STREQUAL) 命令确保在 Debug 配置中运行。除非使用-DCMAKE_BUILD_TYPE=Debug 选项运行 cmake，否则将无法获得覆盖率。

2. 将--coverage 添加到 sut 库所有目标文件的 PRIVATE 编译选项中。
3. 将--coverage 添加到 PUBLIC 链接选项中：GCC 和 Clang 都将此解释为将 gcov（或兼容）库与所有依赖 sut 的目标链接的请求（由于传播属性）。
4. 引入 add_custom_command() 命令，以清理过时的.gcda 文件。在“避免 SEGFAULT 陷阱”部分会详细讨论了添加此命令的原因。

这样就足以生成代码覆盖率。如果使用的是 CLion 等 IDE，也能够运行带有覆盖率的单元测试，并在内置报告视图中获取结果。然而，这可能运行的 CI/CD 中都不会起作用。要获取报告，需要使用 LCOV 生成。

为此，最好定义一个名为 coverage 的新目标。为了保持整洁，将在另一个文件中定义一个单独的函数 AddCoverage，以在测试列表文件中使用：

ch11/06-coverage/cmake/Coverage.cmake

```

1 function(AddCoverage target)
2     find_program(LCOV_PATH lcov REQUIRED)
3     find_program(GENHTML_PATH genhtml REQUIRED)
4     add_custom_target(coverage
5         COMMENT "Running coverage for ${target}..."
6         COMMAND ${LCOV_PATH} -d . --zerocounters
7         COMMAND $<TARGET_FILE:${target}>
8         COMMAND ${LCOV_PATH} -d . --capture -o coverage.info
9         COMMAND ${LCOV_PATH} -r coverage.info '/usr/include/*'
10            -o filtered.info
11         COMMAND ${GENHTML_PATH} -o coverage filtered.info
12            --legend
13         COMMAND rm -rf coverage.info filtered.info
14         WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
15     )
16 endfunction()

```

前面的代码中，首先检测 lcov 和 genhtml 的路径（LCOV 包中的两个命令行工具）。REQUIRED 关键字指示 CMake 在找不到它们时抛出错误。接下来，添加一个自定义的覆盖率目标，步骤如下：

1. 清除先前运行的计数器。
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> 是一个异常的生成器表达式，将在此情况下隐式添加对目标的依赖，可在执行所有命令之前构建。我们将提供 target 作为此函数的参数。
3. 从当前目录（-d .）收集解决方案的度量，并输出到文件（-o coverage.info）。
4. 移除（-r）系统头文件（'/usr/include/*'）上的不需要的覆盖率数据，并输出到另一个文件（-o filtered.info）。
5. 在 coverage 目录中生成 HTML 报告，并添加--legend 颜色。
6. 删除临时.info 文件。
7. 指定 WORKING_DIRECTORY 关键字将二进制树设置为所有命令的工作目录。

这些是 GCC 和 Clang 的一般步骤。重要的是 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。这里唯一的问题是，其必须是一个单个可执行文件。为了解决这个问题，可以提供一个简单的包装脚本（需要使用 chmod +x 将其标记为可执行）：

```
# cmake/gcov-llvm-wrapper.sh
#!/bin/bash
exec llvm-cov gcov "$@"
```

这样，之前函数中所有对 \${LCOV_PATH} 的调用都将接收以下标志：

```
--gcov-tool ${CMAKE_SOURCE_DIR}/cmake/gcov-llvm-wrapper.sh
```

确保这个函数可以在测试列表文件中包含，可以通过在主列表文件中扩展包含搜索路径来实现：

ch11/06-coverage/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(Coverage CXX)
3 include(CTest)
4 list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
5 add_subdirectory(src bin)
6 add_subdirectory(test)
```

突出显示的行允许包含 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：

ch11/06-coverage/test/CMakeLists.txt (fragment)

```
1 # ... skipped unit_tests target declaration for brevity
2 include(Coverage)
3 AddCoverage(unit_tests)
4 include(GoogleTest)
5 gtest_discover_tests(unit_tests)
```

要构建覆盖率目标，请使用以下命令（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
```

完成所有上述步骤后，将看到一个简短的摘要，如下所示：

```
Writing directory view page.
Overall coverage rate:
```

```
lines.....: 95.7% (22 of 23 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，享受报告！不过，这里有一个小问题…

11.7.2. 避免 SEGFAULT 陷阱

开始编辑此类构建解决方案的源代码时，可能会陷入麻烦。这是因为覆盖信息分割成了两部分：

- `gcno` 文件，即 GNU 覆盖笔记，在 SUT 的编译过程中生成
- `gcda` 文件，即 GNU 覆盖数据，在测试运行期间生成和更新

“更新”功能可能是段错误的一个潜在来源，在最初运行测试后，留下了一堆没有移除的 `gcda` 文件。如果我们对源代码进行一些更改并重新编译对象文件，将创建新的 `gcno` 文件。但是，没有擦除步骤——之前的测试运行生成的 `gcda` 文件会跟随过时的源码。当执行 `unit_tests` 二进制文件（这在 `gtest_discover_tests` 宏中发生）时，覆盖信息文件将不匹配，将会收到一个 SEGFAULT（段错误）错误。

为了避免这个问题，我们应该清除任何过时的 `gcda` 文件。由于 `sut` 实例是一个 STATIC 库，可以将 `add_custom_command(TARGET)` 命令与构建事件挂钩，清理将在重新构建开始前执行。

在“扩展阅读”部分找到更多信息的相关链接。

11.8. 总结

表面上，与恰当测试相关的复杂性似乎如此之大，以至于它们不值得付出努力。有多少代码在外运行却没有任何测试，这真是令人惊讶，主要是测试软件是一项令人生畏的任务。如果是手动进行，则更是如此。不幸的是，如果没有严格的自动化测试，代码中的问题都是不完整或根本不可见的。未经测试的代码可能更快编写（但并非总是如此）；然而，要阅读、重构和修复这些代码，绝对要慢得多。

本章中，概述了从一开始就进行测试工作的几个关键原因，其中最引人注目的是心理健康和良好的夜间睡眠。没有一个开发者躺在床上想：等不及几个小时后被打扰，去处理一些生产环境中的火灾和修复错误。但认真地说，在将错误部署到生产环境之前捕捉它们，对你（和公司）可能是一根救命稻草。

当涉及到测试工具时，CMake 在这里真正展现了它的强大。CTest 在检测故障测试方面能发挥奇迹：隔离、混洗、重复和超时。所有这些技术都非常有用，可以通过一个方便的命令行标志获得。我们了解到如何使用 CTest 来列出测试，过滤，并控制测试用例的输出，但最重要的是，现在知道如何使用标准解决方案的真正力量。任何用 CMake 构建的项目都可以进行完全相同的测试，而无需探究其内部细节。

接下来，我们结构化了项目，简化了测试过程，并在生产代码和测试运行器之间重用相同的对象文件。编写自己的测试运行器很有趣，但也许我们应专注于我们程序实际应解决的问题，并花时间接受一个主流的第三方测试框架。

说到这，我们学习了 Catch2 和 GoogleTest 的基础知识。进一步深入研究了 GMock 库的细节，并理解了测试替身，如何工作以实现真正的单元测试。最后，使用 LCOV 设置了一些报告。毕竟，没有什么比硬数据更能证明，我们的解决方案确实是经过全面测试的。

下一章中，将讨论更多有用的工具，以提高代码的质量，并找出潜在的问题。

11.9. 扩展阅读

- 关于 CTest 的 CMake 文档：

<https://cmake.org/cmake/help/latest/manual/ctest.1.html>

- Catch2 文档：

<https://github.com/catchorg/Catch2/blob/devel/docs/>

- GMock 教程：

https://google.github.io/googletest/gmock_for_dummies.html

- Abseil：

<https://abseil.io/>

- 使用 Abseil 的实时更新：

<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 中的覆盖率：

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

<https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html>

<https://gcc.gnu.org/onlinedocs/gcc/Gcov-Data-Files.html>

- Clang 中的覆盖率：

<https://clang.llvm.org/docs/SourceBasedCodeCoverage.html>

- LCOV 命令行工具的文档：

<https://helpmanual.io/man1/lcov/>

- LCOV 项目仓库：

<https://github.com/linux-test-project/lcov>

- GCOV 更新功能：

<https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html#Invoking-Gcov>

第 12 章 程序分析工具

编写高质量的代码并非易事，即使对于经验丰富的开发者来说也是如此。通过在解决方案中包含测试，降低了在主代码中犯基本错误的可能性，但这还不足以避免更复杂的问题。每一款软件都包含如此多的细节，跟踪它们可以成为一份全职工工作。各种约定和特定的设计实践由负责维护产品的团队建立。

有些问题与一致的编码风格相关：代码应该使用 80 列还是 120 列？应该允许使用 `std::bind`，还是坚持使用 `lambda` 函数？使用 C 风格数组是否可以接受？应该将小函数写在一行中吗？应该总是使用 `auto`，还是只在它提高可读性时使用？理想情况下，应该避免使用公认的通常不正确的语句：无限循环、使用标准库保留的标识符、无意中的数据丢失、不必要的 `if` 语句，以及不是“最佳实践”的东西（更多信息请参见“扩展阅读”部分）。

另一个需要考虑的方面是代码现代化。随着 C++ 的发展，引入了新特性，以了解更新到最新标准的挑战性。此外，手动执行此操作既耗时又增加了引入错误的风险，特别是在大型代码库中。最后，应该检查事物在运行时的操作情况：运行程序并检查其内存。内存使用后是否正确释放？是否正在访问正确初始化的数据？还是代码试图访问不存在的指针？

手动管理所有这些挑战和问题既耗时又容易出错。幸运的是，可以使用自动化工具来检查和执行规则，纠正错误，并使代码保持最新。是时候探索程序分析工具了。代码将在每次构建时受到严格审查，以确保其符合行业标准。

本章中，将包含以下内容：

- 执行格式化
- 使用静态检查器
- 使用 Valgrind 进行动态分析

12.1. 示例下载

可以在 GitHub 上找到本章中出现的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch12>。

为了构建本书提供的示例，请使用以下推荐的命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。提醒一下，`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是存放源码的路径。

12.2. 执行格式化

专业开发者通常会遵循规则。据说高级开发者知道何时打破规则，他们能够证明其必要性。另一方面，非常资深开发者通常避免打破规则，以节省时间解释他们的选择。关键是要关注真正影响产品的问题，而不是纠结于细节。

编码风格和格式方面，开发者面临许多选择：应该使用制表符还是空格进行缩进？如果是空格，多少个？列或文件的字符限制应该是多少？这些选择通常不会改变程序的行为，但可能会引发无价值的长时间讨论。

尽管存在通用做法，但辩论往往围绕个人偏好和轶事证据展开。例如，选择每列 80 个字符，而不是 120 个是任意的。重要的是保持一致的样式，不一致可能会妨碍代码的可读性。为确保一致性，建议使用像 clang-format 这样的格式化工具。这个工具可以通知代码是否格式不正确，甚至可以自动更正。以下是一个格式化代码的示例命令：

```
clang-format -i --style=LLVM filename1.cpp filename2.cpp
```

-i 选项指示 clang-format 直接编辑文件，而 --style 指定要使用的格式化风格，如 LLVM、Google、Chromium、Mozilla、WebKit，或提供在文件中的自定义风格（更多细节请参见“扩展阅读”部分）。

当然，我们不想每次更改后都手动执行此命令；CMake 应该作为构建过程的一部分来处理这个问题。我们已经知道如何在系统上定位 clang-format（事先需要手动安装）。尚未了解的是，如何将此外部工具应用于所有源文件。为此，创建一个方便的函数，可以从再 cmake 目录包含它：

ch12/01-formatting/cmake/Format.cmake

```
1 function(Format target directory)
2     find_program(CLANG_FORMAT_PATH clang-format REQUIRED)
3     set(EXPRESSION h hpp hh c cc cxx cpp)
4     list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")
5     file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
6         LIST_DIRECTORIES false ${EXPRESSION})
7     )
8     add_custom_command(TARGET ${target} PRE_BUILD COMMAND
9         ${CLANG_FORMAT_PATH} -i --style=file ${SOURCE_FILES}
10    )
11 endfunction()
```

Format 函数接受两个参数：target 和 directory，将在构建目标之前格式化目录中的所有源文件。

从技术上讲，目录中的所有文件不必属于目标，且目标源代码可能会分布在多个目录中。然而，尤其是需要排除外部库的头文件时，跟踪与目标相关的所有源文件和头文件会很复杂。这时，关注目录比关注逻辑目标更容易。我们可以为每个需要格式化的目录调用该函数。

此函数具有以下步骤：

1. 查找已安装的 clang-format 二进制文件。如果找不到二进制文件，REQUIRED 关键字将使配置停止并报错。
2. 创建要格式化的文件扩展名列表（用作 globbing 表达式）。
3. 在每个表达式前加上目录路径。
4. 使用之前创建的列表递归搜索源文件和头文件，将找到的文件路径放入 SOURCE_FILES 变量（但跳过找到的目录路径）。

5. 将格式化命令附加到目标的 PRE_BUILD 步骤。

这种方法适用于小型到中型代码库。对于大型代码库，可能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运行格式化命令。这可能是由于 shell 命令中的字符限制，通常在约 13000 个字符处达到上限。

现在，来探讨如何实际使用这个函数。以下是我们项目的结构：

```
- CMakeLists.txt
- .clang-format
- cmake
  |- Format.cmake
- src
  |- CMakeLists.txt
  |- header.h
  |- main.cpp
```

首先，设置项目并将 `cmake` 目录添加到模块路径中，以便稍后包含：

ch12/01-formatting/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Formatting CXX)
3 enable_testing()
4 list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
5 add_subdirectory(src bin)
```

接下来，填充 `src` 目录的列表文件：

ch12/01-formatting/src/CMakeLists.txt

```
1 add_executable(main main.cpp)
2 include(Format)
3 Format(main .)
```

这很简单。我们创建一个名为 `main` 的可执行目标，包含 `Format.cmake` 模块，并调用当前目录 (`src`) 中 `main` 目标的 `Format()` 函数。

现在，需要一些未格式化的源文件。头文件包含一个简单的未使用函数：

ch12/01-formatting/src/header.h

```
1 int unused() { return 2 + 2; }
```

还将包含一个带有过多错误空白的源文件：

ch12/01-formatting/src/main.cpp

```
1 #include <iostream>
2
3     using namespace std;
4
5         int main() {
6             cout << "Hello, world!" << endl;
7         }
```

快完成了。只需要格式化器的配置文件，通过 `--style=file` 命令行参数启用：

ch12/01-formatting/.clang-format

```
BasedOnStyle: Google
ColumnLimit: 140
UseTab: Never
AllowShortLoopsOnASingleLine: false
AllowShortFunctionsOnASingleLine: false
AllowShortIfStatementsOnASingleLine: false
```

ClangFormat 将扫描父目录以查找 `.clang-format` 文件，该文件指定了确切的格式化规则。这让我们可以自定义每一个细节。我这里的情况是，从 Google 的编码风格开始，并做了一些调整：设置 140 个字符的列限制，不使用制表符，并且不允许短循环、函数或单行 `if` 语句。构建项目后（格式化在编译前自动进行），文件看起来会像这样：

ch12/01-formatting/src/header.h (格式化后)

```
1 int unused() {
2     return 2 + 2;
3 }
```

即使目标没有使用头文件，其也格式化了。短函数不能放在单行上，正如预期的那样，添加了新行。现在 `main.cpp` 文件看起来也很整洁。不需要的空白已经消失，缩进也标准化了：

ch12/01-formatting/src/main.cpp (formatted)

```
1 #include <iostream>
2
3     using namespace std;
4
5         int main() {
6             cout << "Hello, world!" << endl;
7         }
```

自动化格式化在代码审查期间节省了时间。如果曾经因为空白问题而不得不修改提交，会知道这带来了多大的安慰。一致的格式化让你的代码毫不费力地保持清洁。

Note

将格式化应用于整个代码库，很可能对仓库中的大多数文件造成一次性的大更改。如果你（或你的团队成员）正在进行一些工作，这可能会引起很多合并冲突。最好的做法是在所有挂

起更改完成后协调这样的努力。如果这不可能，考虑逐步采用，或许可以按目录进行，团队成员会感激你的。

尽管格式化器在使代码视觉上一致方面表现出色，但它不是一个全面的程序分析工具。对于更高级的需求，需要设计用于静态分析的其他工具。

12.3. 使用静态检查器

静态程序分析涉及在不运行的情况下检查源代码。一致地使用静态检查器可以显著提高代码质量，使其更加一致，不易受错误和已知安全漏洞的影响。C++ 社区提供了广泛的静态检查器，如 Astrée、clang-tidy、CLazy、CMetrics、Cppcheck、Cplint、CQMetrics、ESBMC、FlawFinder、Flint、IKOS、Joern、PC-Lint、Scan-Build、Vera++ 等。

许多这些工具将 CMake 视为行业标准，并提供现成的支持或集成教程。一些构建工程师更喜欢不编写 CMake 代码，而是通过在线可用的外部模块包含静态检查器。一个例子是 Lars Bilke 在他的 GitHub 仓库中的集合：<https://github.com/bilke/cmake-modules>。

有一种普遍的观点认为设置静态检查器很复杂。这种观念的存在，是因为静态检查器经常模拟真实编译器的行为来理解代码，但这并不一定困难。

Cppcheck 在其手册中概述了以下简单步骤：

1. 定位静态检查器的可执行文件。
2. 使用以下内容生成编译数据库：
 - `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON.`
3. 使用生成的 JSON 文件运行检查器：
 - `<path-to-clang> --project=compile_commands.json`

这些步骤应集成到构建过程中，以确保不被忽视。

既然 CMake 知道如何构建目标，它是否也能支持任何静态检查器？绝对可以，而且比想象的要简单。CMake 允许按目标启用以下检查器：

- `include-what-you-use` (<https://include-what-you-use.org>)
- `clang-tidy` (<https://clang.llvm.org/extrac clang-tidy>)
- `Link What You Use` (CMake 内置检查器)
- `Cplint` (<https://github.com/cplint/cplint>)
- `Cppcheck` (<https://cppcheck.sourceforge.io>)

要启用这些检查器，请将目标属性设置为以分号分隔的列表，其中包含检查器可执行文件的路径和要转发的命令行选项：

- `<LANG>_CLANG_TIDY`
- `<LANG>_CPPCHECK`
- `<LANG>_CPPLINT`
- `<LANG>_INCLUDE_WHAT_YOU_USE`

- `LINK_WHAT_YOU_USE`

将 `<LANG>` 替换为 `C` 以用于 `C` 源文件，替换为 `CXX` 以用于 `C++` 源文件。如果想要为所有项目目标启用检查器，请设置以 `CMAKE_` 为前缀的全局变量——例如：

```
set(CMAKE_CXX_CLANG_TIDY /usr/bin/clang-tidy-3.9;-checks=*)
```

声明之后定义的目标，都将其 `CXX_CLANG_TIDY` 属性设置为该值，启用此分析可能会略微延长构建时间。另一方面，对目标如何被检查器测试的控制更加详细可能很有用。我们可以创建一个简单的函数来处理这个问题：

ch12/02-clang-tidy/cmake/ClangTidy.cmake

```
1 function(AddClangTidy target)
2     find_program(CLANG-TIDY_PATH clang-tidy REQUIRED)
3     set_target_properties(${target}
4         PROPERTIES CXX_CLANG_TIDY
5             "${CLANG-TIDY_PATH};-checks=*-;--warnings-as-errors=*"')
6     )
7 endfunction()
```

`AddClangTidy` 函数遵循两个基本步骤：

1. 定位 `clang-tidy` 二进制文件并将其路径存储在 `CLANG-TIDY_PATH` 中。`REQUIRED` 关键字在找不到二进制文件时，配置将停止并报错。
2. 通过提供二进制路径和特定选项来为目标启用 `clang-tidy`，以激活所有检查并将警告视为错误。

要使用此函数，只需包含模块并针对所选目标调用它：

ch12/02-clang-tidy/src/CMakeLists.txt

```
1 add_library(sut STATIC calc.cpp run.cpp)
2 target_include_directories(sut PUBLIC .)
3 add_executable(bootstrap bootstrap.cpp)
4 target_link_libraries(bootstrap PRIVATE sut)
5 include(ClangTidy)
6 AddClangTidy(sut)
```

这种方法简洁而有效。构建解决方案时，`clang-tidy` 的输出如下所示：

```
[ 6%] Building CXX object bin/CMakeFiles/sut.dir/calc.cpp.o
/root/examples/ch12/04-clang-tidy/src/calc.cpp:3:11: warning: method 'Sum'
can be made static [readability-convert-member-functions-to-static]
int Calc::Sum(int a, int b) {
^

[ 12%] Building CXX object bin/CMakeFiles/sut.dir/run.cpp.o
/root/examples/ch12/04-clang-tidy/src/run.cpp:1:1: warning: #includes are
```

```
not sorted properly [llvm/include-order]
#include <iostream>
^ ~~~~~
/root/examples/ch12/04-clang-tidy/src/run.cpp:3:1: warning: do not use
namespace using-directives; use using-declarations instead [google-buildusing-namespace]
using namespace std;
^

/root/examples/ch12/04-clang-tidy/src/run.cpp:6:3: warning: initializing
non-owner 'Calc *' with a newly created 'gsl::owner<>' [cppcoreguidelinesowning-memory]
auto c = new Calc();
```

注意，除非在命令行参数中添加 `--warnings-as-errors=*` 选项，否则构建将成功。组织应该决定一套必须严格遵循的规则，以防止不符合规范的代码进入仓库。

`clang-tidy` 还提供了一个有用的 `--fix` 选项，当可能时自动更正你的代码。这个特性是一个宝贵的时间节省器，特别是在扩大检查列表时特别有用。与格式化类似，在将静态分析工具做出的更改添加到现有代码库时，要小心合并冲突。

根据情况、仓库大小和团队偏好，应该选择最适合需求的几个检查器，包含太多可能会变得干扰性。这里是对 CMake 直接支持的检查器的简要概述。

12.3.1. clang-tidy

以下是 `clang-tidy` 官方网站上的介绍：

Tip

`clang-tidy` 是基于 `clang` 的 C++ “lint” 工具，目的是提供一个可扩展的框架，用于诊断和修复典型的编程错误，如风格违规、接口误用，可以通过静态分析推断出的错误。`clang-tidy` 是模块化的，并为编写新检查提供了方便的接口。

这个工具相当强大，提供了超过 400 个检查。它与 `ClangFormat` 配合得很好，可以自动应用修复（超过 150 个可用），以符合同一格式文件。提供的检查覆盖了性能、可读性、现代化、C++ 核心指南和易出错区域。

12.3.2. CppLint

以下是 `Cpplint` 官方网站上的描述：

Tip

`Cpplint` 是一个命令行工具，用于检查遵循 Google 的 C++ 风格指南的 C/C++ 文件中的风格问题。`Cpplint` 由 Google Inc. 开发和维护。

这个 linter 旨在使代码与 Google 的风格指南保持一致。它使用 Python 编写，可能对某些项目引入了不受欢迎的依赖。修复格式可供 Emacs、Eclipse、VS7、Junit 和 sed 命令使用。

12.3.3. Cppcheck

以下是 Cppcheck 官方网站上的介绍：

Tip

Cppcheck 是一个用于 C/C++ 代码的静态分析工具，提供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构造。目标是尽量减少误报。Cppcheck 设计为即使代码具有非标准语法（在嵌入式项目中很常见），也能分析 C/C++ 代码。

这个工具特别擅长于最小化误报，使其成为一个可靠的代码分析选项。它已经存在了 14 年以上，并且仍在积极维护。如果代码与 Clang 不兼容，可以考虑使用。

12.3.4. include-what-you-use

以下是 include-what-you-use 官方网站上的描述：

Tip

include-what-you-use 的主要目标是移除多余的 #include。通过确定哪些 #include 实际上不需要此文件（对于.cc 和.h 文件），并在可能的情况下用前向声明替换 #include 来实现。

虽然，小项目中拥有太多包含的头文件，可能看起来不是一个大问题，但避免不必要的头文件编译所节省的时间在大型项目中会迅速累积。

12.3.5. Link What You Use

以下是 CMake 博客上“Link what you use”的描述：

Tip

这是一个内置的 CMake 特性，使用 ld 和 ldd 的选项来打印出如果可执行文件链接了比实际所需更多的库。

静态分析在医学、核能、航空、汽车和机械等行业中扮演着关键角色。在这些行业中，软件错误可能是致命的。明智的开发者会在不关键的环境中测试这些实践，尤其是当成本很低时。在构建过程中使用静态分析，不仅比手动查找和修复错误更经济，而且使用 CMake 也很容易启用。我甚

至可以说，在质量敏感的软件中，几乎没有理由跳过这些检查，这包括除了开发者之外，还涉及其他人的软件。

这个特性还有助于，通过专注于消除不必要的二进制文件来加速构建时间，但不是所有的错误都能在运行程序之前检测到，也不是所有的错误都能在运行程序之前都能检测到的。幸运的是，可以采取额外的步骤来更深入地了解我们的项目，比如使用 Valgrind。

12.4. 动态分析与 Valgrind

Valgrind (<https://www.valgrind.org>) 是一个用于构建动态分析工具的 *nix 仪器化框架，可以在程序运行时执行分析。它附带了一系列工具，用于各种类型的调查和检查。其中一些工具包括：

- Memcheck：检测内存管理问题
- Cachegrind：分析 CPU 缓存，识别缓存未命中和其他问题
- Callgrind：Cachegrind 的扩展，提供关于调用图的额外信息
- Massif：堆分析器，显示程序的不同部分随时间如何使用堆
- Helgrind：用于数据竞争问题的线程调试器
- DRD：Helgrind 的一个更轻量级、功能更有限的版本

这个列表中的每个工具在需要时都非常有用。大多数系统包管理器都了解 Valgrind，可以轻松地在操作系统上安装。如果使用的是 Linux，可能已经安装好了。此外，官方网站为那些喜欢自己构建的用户提供了源代码。

我们的讨论将主要关注 Memcheck，这是 Valgrind 套件中最常用的工具（当开发者提到 Valgrind 时，通常指的是 Valgrind 的 Memcheck）。我们将探讨如何使用 CMake 与之配合，这将在以后发现有必要时更容易采用套件中的其他工具。

12.4.1. Memcheck

Memcheck 对于调试内存问题非常有价值，这个问题在 C++ 中可能特别复杂。开发者对内存管理有控制权，可能会犯各种错误。这些错误可能包括读取未分配的或已释放的内存，多次释放内存，甚至写入错误的地址。这些错误很容易被忽视，甚至潜入到最简单的程序中。有时，只是一个遗忘的变量初始化就足以遇到麻烦。

调用 Memcheck 的命令如下：

```
valgrind [valgrind-options] tested-binary [binary-options]
```

Memcheck 是 Valgrind 的默认工具，也可以像这样明确指定：

```
valgrind --tool=memcheck tested-binary
```

运行 Memcheck 可能会大大减慢程序的速度；手册（请参阅“扩展阅读”中的链接）指出，使用它检测的程序可能会慢 10-15 倍。为了避免每次运行测试都等待 Valgrind，可以将创建

一个单独的目标，在需要测试代码时从命令行调用。理想情况下，这应该在新代码合并到主代码库之前完成。可以将此步骤包含在早期的 Git 钩子中，或者作为持续集成（CI）流程的一部分。

要在 CMake 中创建 Valgrind 的自定义目标，可以在 CMake 生成阶段之后使用以下命令：

```
cmake --build <build-tree> -t valgrind
```

以下是如何在 CMake 中添加此类目标：

ch12/03-valgrind/cmake/Valgrind.cmake

```
1 function(AddValgrind target)
2     find_program(VALGRIND_PATH valgrind REQUIRED)
3     add_custom_target(valgrind
4         COMMAND ${VALGRIND_PATH} --leak-check=yes
5         ${<TARGET_FILE:$target>}
6         WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
7     )
8 endfunction()
```

这个例子中，定义了一个名为 AddValgrind 的 CMake 函数，其接受要测试的目标（可以在项目中重用）。这里发生了两件事情：

1. CMake 检查默认系统路径以查找 valgrind 可执行文件，并将其路径存储在 VALGRIND_PATH 变量中。如果未找到二进制文件，则 REQUIRED 关键字将使配置出错停止。
2. 创建了一个名为 valgrind 的自定义目标，对指定的二进制文件运行 Memcheck，并始终检查内存泄漏的选项。

可以在以下方式设置 Valgrind 选项：

- 在 /.valgrindrc 文件中（家目录中）
- 通过 \$VALGRIND_OPTS 环境变量
- 在./.valgrindrc 文件中（工作目录中）

这些选项按该顺序检查。另外，只有当最后一个文件属于当前用户、是常规文件且未标记为全局可写时，才会考虑该文件。这是一种安全机制，因为给 Valgrind 的选项可能具有潜在的危害。

要使用 AddValgrind 函数，将其与 unit_tests 目标一起提供，我们希望在一个精细控制的环境中（如单元测试）运行：

ch12/03-valgrind/test/CMakeLists.txt (片段)

```
1 # ...
2 add_executable(unit_tests calc_test.cpp run_test.cpp)
3 # ...
4 include(Valgrind)
5 AddValgrind(unit_tests)
```

记住，使用 Debug 配置生成构建树可以让 Valgrind 利用调试信息，使其输出更加清晰。来看看这在实践中是如何工作的：

```
# cmake -B <build tree> -S <source tree> -DCMAKE_BUILD_TYPE=Debug  
# cmake --build <build-tree> -t valgrind
```

这将配置项目，构建 sut 和 unit_tests 目标，并启动 Memcheck 的执行，它将提供一些信息：

```
[100%] Built target unit_tests  
==954== Memcheck, a memory error detector  
==954== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==954== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info  
==954== Command: ./unit_tests
```

前缀 ==954== 包含进程 ID，有助于区分 Valgrind 的评论和测试进程的输出。

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，输出了一个摘要：

```
==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
```

哎呀！我们至少还在使用 1 字节。使用 malloc() 和 new 进行的分配，没有相应的 free() 和 delete 操作匹配。看起来我们的程序中有内存泄漏。Valgrind 提供了更多细节来找到它：

```
==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 0x<address> 开头的行指示调用堆栈中的单个函数。我截断了输出（它包含了一些来自 GTest 的噪音），以专注于有趣的部分——最顶层的函数和源引用，run() (run.cpp:6)：

最后，在底部找到了：

```
==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 在查找复杂问题方面非常出色。有时，甚至可以更深入地挖掘，找到不容易归类的问题。这些将在“可能丢失”行中显示。

让我们看看 Memcheck 在这个案例中找到的问题是什么：

ch12/03-valgrind/src/run.cpp

```
1 #include <iostream>
2 #include "calc.h"
3 using namespace std;
4 int run() {
5     auto c = new Calc();
6     cout << "2 + 2 = " << c->Sum(2, 2) << endl;
7     cout << "3 * 3 = " << c->Multiply(3, 3) << endl;
8     return 0;
9 }
```

实际上，我们创建了一个在测试结束前没有删除的对象。这正是为什么测试覆盖非常重要的原因。

Valgrind 是一个有用的工具，但在复杂的程序中，其输出可能会变得令人不知所措。有一种更有效管理这些信息的方法——就是 Memcheck-Cover 项目。

12.4.2. Memcheck-Cover

像 CLion 这样的商业 IDE 可以直接解析 Valgrind 的输出，这使得通过图形界面导航变得更加容易，而无需在控制台中滚动。如果编辑器缺少这个功能，第三方报告生成器可以提供更清晰的视图。David Garcin 开发的 Memcheck-Cover 通过创建一个 HTML 文件提供了更好的体验：

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== 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 0x483BB63: 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::internal::HandleSehExceptionsInMethodIfSupported(testing::Test::*, void (testing::Test::*)(char const*)) (gtest.cc:2607)
==1508==    by 0x1496C2: void testing::internal::HandleExceptionsInMethodIfSupported(testing::Test::*, void (testing::Test::*)(char const*)) (gtest.cc:2643)
==1508==    by 0x11E0DF: testing::Test::Run() (gtest.cc:2682)
==1508==    by 0x11EB21: testing::TestInfo::Run() (gtest.cc:2861)
==1508==    by 0x11F41E: testing::TestSuite::Run() (gtest.cc:3015)
==1508==    by 0x12EDB0: testing::internal::UnitTestFixture::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::*)(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 12.1: 由 Memcheck-Cover 生成的报告

这个整洁的小项目在 GitHub 上可用 (<https://github.com/Farigh/memcheck-cover>)；需要 Valgrind 和 gawk (GNU AWK 工具)。要使用它，需要准备一个单独的 CMake 模块中的设置函数。其将包括两部分：

1. 获取和配置工具
2. 添加一个自定义目标来运行 Valgrind 并生成报告

配置如下所示：

ch12/04-memcheck/cmake/Memcheck.cmake

```
1 function(AddMemcheck target)
2     include(FetchContent)
3     FetchContent_Declare(
4         memcheck-cover
5         GIT_REPOSITORY https://github.com/Farigh/memcheck-cover.git
6         GIT_TAG release-1.2
7     )
8     FetchContent_MakeAvailable(memcheck-cover)
9     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做法：包括 `FetchContent` 模块，并使用 `FetchContent_Declare` 指定项目的仓库和所需的 Git 标签。接下来，启动 `fetch` 过程并配置二进制路径，使用由 `FetchContent_Populate` 设置的 `memcheckcover_SOURCE_DIR` 变量（由 `FetchContent_MakeAvailable` 隐式调用）。

函数的第二部分是创建生成报告的目标。我们将称之为 `memcheck`（这样它就不会与之前的 `valgrind` 目标重叠）：

ch12/04-memcheck/cmake/Memcheck.cmake (continued)

```
1 add_custom_target(memcheck
2     COMMAND ${MEMCHECK_PATH}/memcheck_runner.sh -o
3         "${CMAKE_BINARY_DIR}/valgrind/report"
4     -- ${TARGET_FILE}:${target}
5     COMMAND ${MEMCHECK_PATH}/generate_html_report.sh
6         -i "${CMAKE_BINARY_DIR}/valgrind"
7         -o "${CMAKE_BINARY_DIR}/valgrind"
8     WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
9 )
10 endfunction()
```

这发生在两个命令中：

1. 首先，运行 `memcheck_runner.sh` 包装脚本，将执行 Valgrind 的 Memcheck 并收集输出到由`-o` 参数提供的文件中。
2. 然后，使用 `generate_html_report.sh` 解析输出并创建报告。此脚本需要由`-i` 和`-o` 参数提供的输入和输出目录。

这两个步骤都应该在 `CMAKE_BINARY_DIR` 工作目录中执行，以便单元测试二进制文件可以通过相对路径访问文件。

当然，需要添加到列表文件的是，对这个函数的调用：

ch12/04-memcheck/test/CMakeLists.txt (片段)

```
1 include(Memcheck)
2 AddMemcheck(unit_tests)
```

生成带有 `Debug` 配置的构建系统后，可以使用以下命令构建目标：

```
# cmake -B <build tree> -S <source tree> -DCMAKE_BUILD_TYPE=Debug
# cmake --build <build-tree> -t memcheck
```

然后，就可以享受格式化报告和生成的 HTML 页面了！

12.5. 总结

“你将在阅读代码上花费的时间比编写代码的时间要多，应该优先优化代码的可读性。”这一原则在许多关于整洁代码的书籍中都有所体现。它得到了许多软件开发者的经验支持，这也是为什么像空格数量、换行，以及 `#import` 语句的顺序这样的小细节都要标准化。这种标准化不仅仅是为了做到细致，而是为了节省时间。遵循本章中的实践，可以忘记手动格式化代码。构建时，代码会自动格式化，这是无论如何都要进行的测试代码的步骤。使用 `ClangFormat`，可以确保格式化符合选择的标准。

除了简单的空格调整，代码还应该满足许多其他指南。这时 `clang-tidy` 就派上用场了，其有助于执行团队或组织约定的编码规范。我们深入讨论了这款静态检查器，还提到了其他选项，如 `Cpplint`、`Cppcheck`、`include-what-you-use` 和 `Link What You Use`。由于静态链接器相对较快，可以几乎不投入成本地将其添加到构建中，这通常非常值得。

我们还检查了 `Valgrind` 工具，特别是 `Memcheck`，用于识别内存管理问题，如错误的读写操作。这个工具在避免手动调试数小时，和防止生产环境中出现漏洞方面，具有无法估量的价值。我们介绍了一种方法，通过 `Memcheck-Cover` 这个 HTML 报告生成器，使 `Valgrind` 的输出更加用户友好。这在无法运行 IDE 的环境中使用特别有用，比如 CI 流程。

本章只是一个起点。还有许多其他免费和商业工具有助于提高代码质量。探索它们，找到最适合你的工具。下一章中，我们将深入探讨生成文档的内容。

12.6. 扩展阅读

- 由 C++ 的作者 Bjarne Stroustrup 策划的 C++ 核心指南：
<https://github.com/isocpp/CppCoreGuidelines>
- ClangFormat 参考手册：
<https://clang.llvm.org/docs/ClangFormat.html>
- C++ 的静态分析工具——精选列表：
<https://github.com/analysis-tools-dev/static-analysis#cpp>
- CMake 内置的静态检查器支持：
<https://www.kitware.com//static-checks-with-cmake-cdash-iwyu-clang-tidy-lwyu-cpplint-and-cppcheck/>
- 启用 `clang-tidy` 的目标属性：
https://cmake.org/cmake/help/latest/prop_tgt/LANG_CLANG_TIDY.html
- Valgrind 手册：
<https://www.valgrind.org/docs/manual/manual-core.html>

第 13 章 生成文档

高质量的代码不仅仅是编写良好、可运行且经过测试的——还应该有详尽的文档。文档使我们能够分享可能丢失的信息，描绘更大的蓝图，提供上下文，揭示意图，并且最终——对外部用户和维护者进行介绍。

你还记得上次加入一个新项目，在目录和文件的迷宫中迷失数小时的经历吗？这种情况是可以避免的。真正优秀的文档，可以让一个完全的新手在几秒钟内找到他们所需的代码行。遗憾的是，缺少文档的问题常常会被忽视。这并不奇怪——它需要相当多的技巧，而我们中的许多人并不擅长于此。此外，文档和代码很快就会过时。除非实施严格的更新和审查过程，否则很容易忘记文档也需要关注。

一些团队（为了节省时间或因为经理鼓励他们这样做）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、可读的标识符，希望避免编写文档的麻烦。即使是最优秀的函数签名也不能确保传达所有必要的信息——例如，`int removeDuplicates()`；是描述性的，但它没有揭示返回了什么。可能是找到的重复项的数量，剩余项目的数量，或其他东西——都不清楚。虽然良好的命名习惯绝对正确，但它不能取代认真编写文档的行为。记住：世上没有免费的午餐。

为了使事情变得更容易，专业人士使用自动文档生成器来分析源文件中的代码和注释，以生成各种格式的全面文档。将这样的生成器添加到 CMake 项目中非常简单——来看看如何操作！

本章中，将包含以下内容：

- 将 Doxygen 添加到项目
- 生成具有现代外观的文档
- 使用自定义 HTML 增强输出

13.1. 示例下载

可以在 GitHub 上找到本章中出现的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch13>。

为了构建本书提供的示例，请使用以下推荐命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的路径。

13.2. 添加 Doxygen 到项目中

Doxygen 是最成熟和流行的从 C++ 源代码生成文档的工具。当我说“成熟”时，我是认真的：第一个版本是由 Dimitri van Heesch 在 1997 年 10 月发布的。从那时起，它已经广为流传，并且几乎有 250 名贡献者活跃地支持 (<https://github.com/doxygen/doxygen>)。

可能会担心将 Doxygen 整合到，没有使用文档生成的较大项目中。确实，为每个函数添加注释可能看起来令人生畏。然而，我鼓励从小处着手。专注于记录最近在最新提交中处理过的元素。记住，即使是部分完整的文档，也比完全没有文档要好，并且它逐渐帮助建立对项目的更全面的理解。

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)
- 可移植文档格式 t (PDF)
- Lamport TeX (LaTeX)
- PostScript (PS)
- Unix 手册 (man 页面)
- 微软 HTML 帮助手册 (.CHM)

如果使用 Doxygen 指定的格式，在代码中添加提供信息注释，其将解析这些注释以丰富输出文件。此外，将分析代码结构以生成有用的图表和图形。后者是可选的，其需要外部 Graphviz 工具 (<https://graphviz.org/>)。

开发者首先应该考虑以下问题：项目的用户只会接收文档，还是自己生成文档（可能是从源代码构建时）？第一个选项意味着文档随二进制文件分发，在线提供，或者（不太优雅地）与源代码一起检入到仓库中。

这很重要，如果希望用户在构建过程中生成文档，将需要在系统中存在依赖项。这并不是一个严重的问题，因为 Doxygen 和 Graphviz 可以通过大多数包管理器获得，并且只需要一个简单的命令，例如在 Debian 上：

```
apt-get install doxygen graphviz
```

Windows 的二进制文件也可以在项目网站上找到（请参阅“扩展阅读”部分）。

然而，一些用户可能不太愿意安装这个工具。我们必须决定是为用户生成文档，还是让他们在需要时添加依赖项。项目也可以像第 9 章描述的那样，为用户自动添加。注意，Doxygen 是用 CMake 构建的。

当 Doxygen 和 Graphviz 安装到系统中后，可以将生成过程添加到我们的项目中。与一些在线资源建议的相反，这并不像看起来那么困难或复杂。不需要创建外部配置文件，提供 Doxygen 可执行文件的路径，或添加自定义目标。自从 CMake 3.9 以来，可以使用 FindDoxygen 查找模块中的 doxygen_add_docs() 函数，其会设置文档目标。

```
doxygen_add_docs(targetName [sourceFilesOrDirs...]
[ALL] [WORKING_DIRECTORY dir] [COMMENT comment])
```

第一个参数指定目标名称，需要使用-t 参数显式构建它，通过 cmake 生成构建树：

```
# cmake --build <build-tree> -t targetName
```

或者，可以通过添加 ALL 参数来确保始终构建文档。WORKING_DIRECTORY 选项很直接，

指定命令的运行目录。COMMENT 选项设置的值将在文档生成开始之前显示，提供有用的信息或指令。

我们将遵循前几章的做法，并创建一个带有辅助函数的实用模块（以便可以在其他项目中重用）：

ch13/01-doxygen/cmake/Doxygen.cmake

```
1 function(Doxygen input output)
2     find_package(Doxygen)
3     if (NOT DOXYGEN_FOUND)
4         add_custom_target(doxxygen COMMAND false
5             COMMENT "Doxygen not found")
6         return()
7     endif()
8     set(DOXYGEN_GENERATE_HTML YES)
9     set(DOXYGEN_HTML_OUTPUT
10        ${PROJECT_BINARY_DIR}/${output})
11    doxygen_add_docs(doxxygen
12        ${PROJECT_SOURCE_DIR}/${input}
13        COMMENT "Generate HTML documentation"
14    )
15 endfunction()
```

该函数接受两个参数——输入和输出目录——并创建一个自定义的 doxygen 目标：

1. 首先，使用 CMake 内置的 Doxygen 查找模块来确定，系统中是否可用 Doxygen。
2. 如果不可用，创建一个虚拟的 doxygen 目标，通知用户并运行 false 命令（在类 Unix 系统中返回 1，导致构建失败）。我们在此处用 `return()` 终止函数。
3. 如果 Doxygen 可用，将其配置为在提供的输出目录中生成 HTML 输出。Doxygen 非常可配置（更多信息请参阅官方文档）。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_。
4. 设置实际的 doxygen 目标。所有 DOXYGEN_ 变量都将被转发到 Doxygen 的配置文件中，并将从源树中提供的输入目录生成文档。

如果文档由用户生成，那么第二步可能应该涉及安装 Doxygen。

要使用这个函数，可以将其纳入我们项目的主列表文件中：

ch13/01-doxygen/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(Doxygen CXX)
3 enable_testing()
4 list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
5 add_subdirectory(src bin)
6 include(Doxygen)
7 Doxygen(src docs)
```

一点也不难！构建 doxygen 目标将生成如下所示的 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

Calc (RandomNumberGenerator *rng)
int Sum (int a, int b)
int Multiply (int a, int b)
int AddRandomNumber (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

图 13.1：使用 Doxygen 生成的类参考

为了在成员函数文档中添加重要细节，可以在头文件中的 C++ 方法声明前加上适当的注释：

ch13/01-doxygent/src/calc.h (片段)

```

1  /**
2   * Multiply... Who would have thought?
3   * @param a the first factor
4   * @param b the second factor
5   * @result The product
6  */
7  int Multiply(int a, int b);

```

这种格式被称为 Javadoc。重要的是，注释块要以双星号开始：`/**`。更多关于 Doxygen 的 docblocks 的描述可以在“扩展阅读”部分的链接中找到。带有此类注释的 `Multiply` 函数将呈现如下所示的图形：

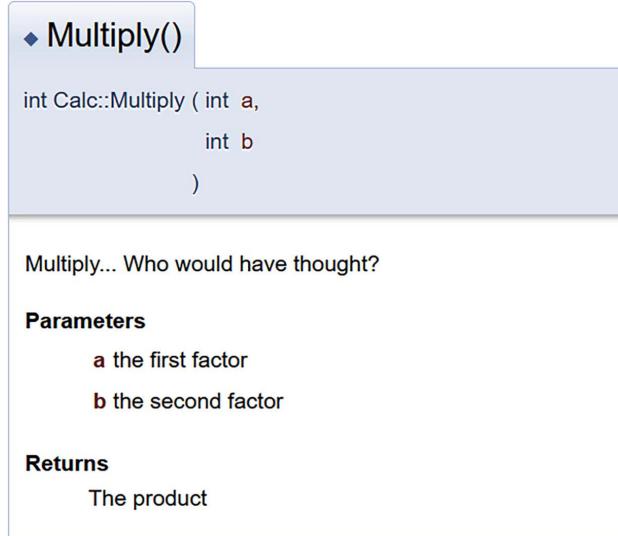


图 13.2: 参数和结果的注释

如果安装了 Graphviz, Doxygen 将检测到它并生成依赖关系图:

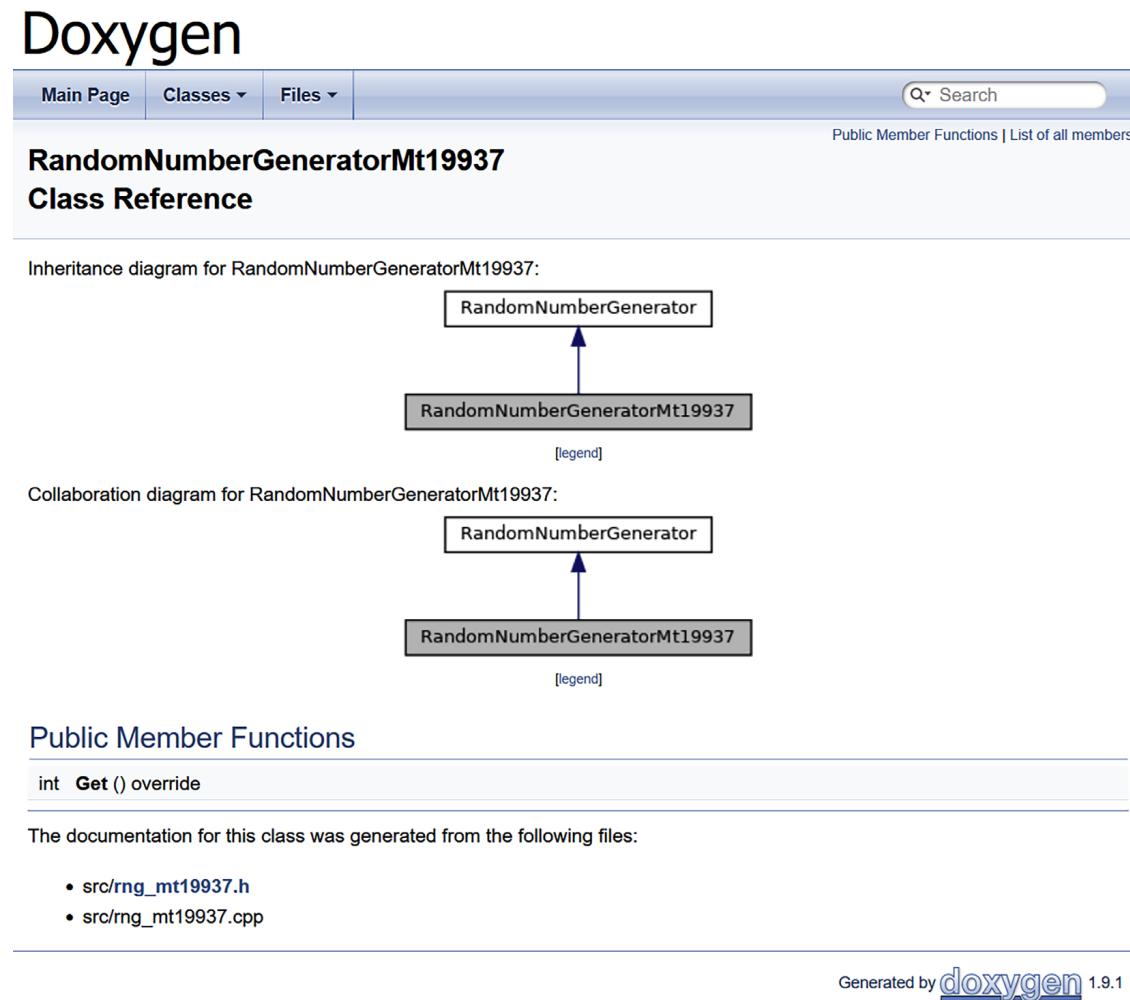


图 13.3: Doxygen 生成的继承和协作图

通过直接从源代码生成文档，建立了一个过程，使得在开发周期中与代码更改同步快速更新成为可能。此外，代码审查期间很可能会注意到忽略的注释更新。

许多开发者表示担忧，Doxygen 提供的设计看起来过时，这让他们犹豫是否向客户展示生成的文档。然而，这个问题有一个简单的解决方案。

13.3. 生成具有现代感的文档

使用清新的设计，来记录项目非常重要。毕竟，如果投入大量工作来为尖端项目编写高质量的文档，那么用户必须将其视为如此。尽管 Doxygen 功能丰富，但它并不以遵循最新的视觉趋势而闻名。然而，改造其外观并不需要大量的努力。

幸运的是，一位名为 `jotheapro` 的开发者创建了一个名为 `doxygen-awesome-css` 的主题，提供了一个现代化、可定制的设计。以下截图展示了这个主题：

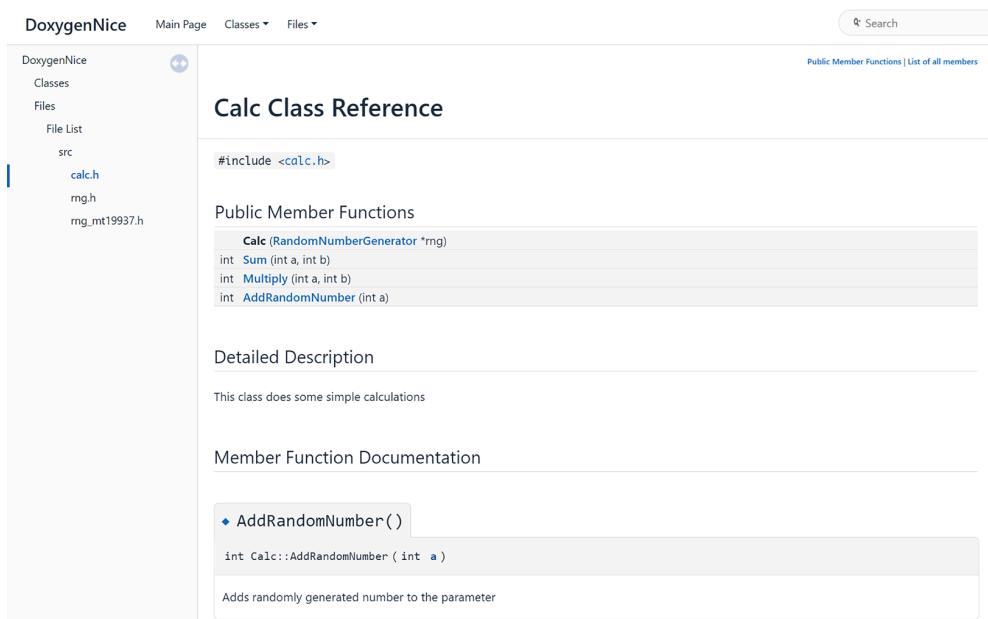


图 13.4：使用 `doxygen-awesome-css` 主题的 HTML 文档

这个主题不需要任何依赖，可以轻松地从其 GitHub 页面获取，网址为：<https://github.com/jotheapro/doxygen-awesome-css>。

Note

有些在线资源推荐使用应用程序组合，比如通过 Sphinx 的 Breathe 和 Exhale 扩展来转换 Doxygen 的输出，这种方法可能比较复杂且依赖较多（例如需要 Python）。对于一个并非所有成员都深入了解 CMake 的团队来说，更简单的方法通常更实用。

我们可以通过一个自动化过程高效地实现这个主题。看看如何通过添加一个新的宏，来扩展 `Doxygen.cmake` 文件并使用：

`ch13/02-doxygen-nice/cmake/Doxygen.cmake` (片段)

```
1 macro(UseDoxygenAwesomeCss)
2   include(FetchContent)
3   FetchContent_Declare(doxygen-awesome-css)
```

```

4     GIT_REPOSITORY
5         https://github.com/jothepro/doxygen-awesome-css.git
6     GIT_TAG
7         V2.3.1
8 )
9 FetchContent_MakeAvailable(doxygen-awesome-css)
10 set(DOXYGEN_GENERATE_TREEVIEW YES)
11 set(DOXYGEN_HAVE_DOT YES)
12 set(DOXYGEN_DOT_IMAGE_FORMAT svg)
13 set(DOXYGEN_DOT_TRANSPARENT YES)
14 set(DOXYGEN_HTML_EXTRA_STYLESHEET
15     ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome.css)
16 endmacro()

```

我们已经从本书的前几章中了解了所有这些命令，但为了完全清晰，再来看看发生了什么：

1. 使用 FetchContent 模块从 Git 获取 doxygen-awesome-css
2. 为 Doxygen 配置额外的选项（这些选项特别由主题的 README 文件推荐）
3. 将主题的 css 文件复制到 Doxygen 的输出目录

最好在 Doxygen 函数中调用这个宏，正好在 doxygen_add_docs() 之前：

ch13/02-doxygen-nice/cmake/Doxygen.cmake (片段)

```

1 function(Doxygen input output)
2 # ...
3 UseDoxygenAwesomeCss()
4 doxygen_add_docs (...)
5 endfunction()

6
7 macro(UseDoxygenAwesomeCss)
8 # ...
9 endmacro()

```

记住，宏中的所有变量都在调用函数的作用域内设置。现在，可以享受生成的 HTML 文档中的现代风格，并且自豪地与世界分享。然而，我们的演示主题提供了一些 JavaScript 模块来增强体验。

最后，我们要如何包含它们？

13.4. 使用自定义 HTML 增强输出

Doxygen Awesome 提供了一些额外的功能，这些功能可以通过在文档头部的 HTML <head> 标签内包含几个 JavaScript 代码片段来启用。这些功能非常有用，允许用户在浅色和深色模式之间切换、为代码片段添加复制按钮、提供段落标题的永久链接以及创建交互式的目录。

然而，实现这些功能需要将额外的代码复制到输出目录，并将其包含在生成的 HTML 文件中。以下是需要在 </head> 标签前包含的 JavaScript 代码：

ch13/cmake/extrah_headers

```
<script type="text/javascript" src="$relpath^doxygen-awesome-darkmodetoggle.js"></script>
<script type="text/javascript" src="$relpath^doxygen-awesome-fragmentcopy-button.js"></script>
<script type="text/javascript" src="$relpath^doxygen-awesome-paragraphlink.js"></script>
<script type="text/javascript" src="$relpath^doxygen-awesome-interactivetoc.js"></script>

<script type="text/javascript">
    DoxygenAwesomeDarkModeToggle.init()
    DoxygenAwesomeFragmentCopyButton.init()
    DoxygenAwesomeParagraphLink.init()
    DoxygenAwesomeInteractiveToc.init()
</script>
```

这段代码首先会包含几个 JavaScript 文件，然后初始化不同的扩展。不幸的是，这段代码不能简单地添加到某个变量中。相反，需要用一个自定义文件覆盖默认的头部文件。这种覆盖可以通过向 Doxygen 的 HTML_HEADER 配置变量提供该文件的路径来完成。

为了创建一个自定义头部而无需硬编码全部内容，可以使用 Doxygen 的命令行工具来生成一个默认的头部文件，并在生成文档之前编辑它：

```
doxygen -w html header.html footer.html style.css
```

尽管我们不会使用或更改 footer.html 或 style.css，但它们是必需的参数，所以无论如何都需要创建它们。

最后，需要自动在 </head> 标签前插入 ch13/cmake/extrah_headers 文件的内容以包含所需的 JavaScript。这可以通过 Unix 命令行工具 sed 来完成，其会就地编辑 header.html 文件：

```
sed -i '/</head>/r ch13/cmake/extrah_headers' header.html
```

现在需要用 CMake 语言来编写这些步骤。以下是实现这一目标的宏：

ch13/02-doxygen-nice/cmake/Doxygen.cmake (片段)

```
1 macro(UseDoxygenAwesomeExtensions)
2     set(DOXYGEN_HTML_EXTRA_FILES
3         ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome-darkmode-toggle.js
4         ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome-fragment-copybutton.js
5         ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome-paragraph-link.js
6         ${doxygen-awesome-css_SOURCE_DIR}/doxygen-awesome-interactive-toc.js
7     )
8
9     execute_process(
10        COMMAND doxygen -w html header.html footer.html style.css
11        WORKING_DIRECTORY ${PROJECT_BINARY_DIR}
12    )
```

```

13 execute_process(
14     COMMAND sed -i
15     "/<\\>/r ${PROJECT_SOURCE_DIR}/cmake/extrah_headers"
16     header.html
17     WORKING_DIRECTORY ${PROJECT_BINARY_DIR}
18 )
19 set(DOXYGEN_HTML_HEADER ${PROJECT_BINARY_DIR}/header.html)
20 endmacro()

```

这段代码看起来很复杂，但仔细看过后，会发现它其实相当简单。

下面列出它的功能：

1. 将四个 JavaScript 文件复制到输出目录
2. 执行 doxygen 命令来生成默认的 HTML 文件
3. 执行 sed 命令将所需的 JavaScript 注入头部
4. 用自定义版本覆盖默认的头部

为了完成集成，可以在启用基本样式表之后调用这个宏：

ch13/02-doxygen-nice/cmake/Doxygen.cmake (fragment)

```

1 function(Doxygen input output)
2 # ...
3     UseDoxygenAwesomeCss()
4     UseDoxygenAwesomeExtensions()
5 # ...
6 endfunction()

```

本例的完整代码以及实际示例可以在本书的在线仓库中找到，我建议你在实际环境中阅读和探索这些示例。

其他文档生成工具

有许多其他工具没有在这本书中涵盖，我们主要关注由 CMake 支持的项目。不过，其中一些可能更适合各位的具体情况。如果愿意尝试新事物，可以访问两个我觉得有趣的项目的网站：

- Adobe 的 Hyde (<https://github.com/adobe/hyde>)：针对 Clang 编译器设计，Hyde 生成 Markdown 文件，这些文件可以像 Jekyll(<https://jekyllrb.com/>)这样的静态页面生成器消费，Jekyll 是 GitHub 支持的工具
- Standardese (<https://github.com/standardese/standardese>)：使用 libclang 编译代码，并提供 HTML、Markdown、LaTeX 和手册页格式的输出，有望成为下一个 Doxygen。

13.5. 总结

本章中，深入探讨了将 Doxygen 这一强大的文档生成工具添加到您的 CMake 项目中，并提升其吸引力的实际操作。尽管这项任务看起来有些令人生畏，但实际上它是相当易于管理的，并且能显著提升解决方案中信息的流动性和清晰度。各位到后面会发现，特别是当个人或团队成员努力理解应用程序中的复杂关系时，投入时间和精力添加和维护文档是一笔值得的投资。探讨了如何使用 CMake 内置的 Doxygen 支持来实际生成文档之后，我们稍微转变了一下方向，确保文档不仅可读，而且易读。

由于过时的设计可能对眼睛造成负担，我们探讨了生成 HTML 的替代外观。这是通过使用 Doxygen Awesome 扩展来完成的。为了启用它带来的增强功能，我们通过添加必要的 javascript 来自定义标准页头。

通过生成文档，确保了它与实际代码的接近性，这使得将书面解释与逻辑保持同步变得更加容易，尤其是它们都在同一个文件中。此外，作为开发者，可能同时在处理许多任务和细节。文档作为一种记忆辅助工具，帮助保留和回忆项目的复杂性。记住，“即使是最短的铅笔也比最长的记忆要长。”帮助一下自己——把写下来，从而取得成功。

总结来说，本章强调了 Doxygen 在项目管理工具中的价值，有助于团队的理解和沟通。

下一章中，我将介绍如何使用 CMake 自动化项目的打包和安装，进一步提升项目的管理技巧。

13.6. 扩展阅读

- Doxygen 官方网站：
<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>

第 14 章 安装与打包

我们的项目已经构建、测试并记录在案。现在，到了将其发布给用户的时候。本章主要关注需要采取的最后两个步骤：安装和打包。这些是我们迄今为止所学的一切基础上的高级技术：管理目标和它们的依赖关系，短暂的使用需求，生成器表达式等。

安装使得项目能够在整个系统中发现和访问。我们将介绍如何在不进行安装的情况下导出目标，以供其他项目使用；以及如何安装项目，以便于整个系统轻松访问。将了解如何配置项目，以自动将各种工件类型放置到适当的目录中。为了处理更高级的场景，将介绍用于安装文件和目录，以及执行自定义脚本和 CMake 命令的低层命令。

接下来，将探索设置可重用的 CMake 包，其他项目可以使用 `find_package()` 命令来发现它们。我们将解释如何确保目标，和定义特定文件的系统位置。我们还将讨论如何编写基本和高级的配置文件，以及与包相关联的版本文件。然后，为了模块化，我们将简要介绍组件的概念，这既适用于 CMake 包也适用于 `install()` 命令。所有这些准备工作将为本章最后要介绍的方面铺平道路：使用 CPack 生成各种操作系统中的包管理器都能识别的存档、安装程序、捆绑包和包。这些包可以分发预构建的工件、可执行文件和库。这是最终用户开始使用软件的最简单方式。

本章中，将包含以下内容：

- 无需安装即可导出
- 在系统上安装项目
- 创建可重用包
- 定义组件
- 使用 CPack 打包

14.1. 示例下载

可以在 GitHub 上的<https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch14>找到本章中出现的代码文件。

要构建本书提供的示例，请使用以下推荐命令：

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

要安装示例，请使用以下命令：

```
cmake --install <build tree>
```

请确保将 `<build tree>` 和 `<source tree>` 占位符替换为适当的路径。提醒一下：`<build tree>` 是目标/输出目录的路径，而 `<source tree>` 是源码所在的路径。

14.2. 导出而不安装

我们如何使项目 A 的目标对项目 B 可用？通常，我们会使用 `find_package()` 命令，但这需要创建一个包并在系统上安装。虽然这种方法很有用，但它涉及到一些工作。有时候，我们只是需要一个快速构建项目，并使其目标对其他项目可用。

一种节省时间的方法是在项目 B 中包含项目 A 的主列表文件，该文件已经包含了所有目标定义。然而，这个文件可能还包括全局配置、具有副作用的 CMake 命令、额外的依赖项，以及 B 项目可能不需要的目标（比如单元测试）。所以，这不是最好的方法。相反，可以为项目 B 提供一个目标导出文件，通过 `include()` 命令包含：

```
1 cmake_minimum_required(VERSION 3.26.0)
2 project(B)
3 include(/path/to/A/TargetsOfA.cmake)
```

使用 `add_library()` 和 `add_executable()` 等命令定义 A 的所有目标，并设置正确的属性。

必须在 `TARGETS` 关键字后指定要导出的所有目标，并在 `FILE` 后提供目标文件名。其他参数是可选的：

```
export(TARGETS [target1 [target2 [...]]]
      [NAMESPACE <namespace>] [APPEND] FILE <path>
      [EXPORT_LINK_INTERFACE_LIBRARIES]
)
```

以下是各个参数的解释：

- `NAMESPACE` 推荐用来指示目标是从其他项目导入的。
- `APPEND` 防止 CMake 在写入前清除文件内容。
- `EXPORT_LINK_INTERFACE_LIBRARIES` 导出目标的链接依赖项（包括导入的和特定配置的变体）。

让我们将这种导出方法应用到 Calc 库示例中，该库提供了两个简单的方法：

ch14/01-export/src/include/calc/basic.h

```
1 #pragma once
2 int Sum(int a, int b);
3 int Multiply(int a, int b);
```

我们需要声明 Calc 目标以便我们有东西可以导出：

ch14/01-export/src/CMakeLists.txt

```
1 add_library(calc STATIC basic.cpp)
2 target_include_directories(calc INTERFACE include)
```

然后，使用 `export(TARGETS)` 命令生成导出文件：

ch14/01-export/CMakeLists.txt (fragment)

```
1 cmake_minimum_required(VERSION 3.26)
2 project(ExportCalcCXX)
3 add_subdirectory(src bin)
4 set(EXPORT_DIR "${CMAKE_CURRENT_BINARY_DIR}/cmake")
5 export(TARGETS calc
6     FILE "${EXPORT_DIR}/CalcTargets.cmake"
7     NAMESPACE Calc::
8 )
```

导出目标声明文件将位于构建树的 `cmake` 子目录中（遵循 `.cmake` 文件的约定），为了避免稍后重复路径，我们将其设置在 `EXPORT_DIR` 变量中。然后，调用 `export()` 来生成目标声明文件 `CalcTargets.cmake`，其中包含 `calc` 目标。对于包含这个文件的项目，可以直接使用 `Calc::calc`。

注意，这个导出文件还是一个包。更重要的是，这个文件中的所有路径都是绝对路径，并且硬编码到构建树中，这使得它们不可重定位。

`export()` 命令还有一个使用 `EXPORT` 关键字的简写版本：

```
export(EXPORT <export> [NAMESPACE <namespace>] [FILE <path>])
```

然而，它需要一个预定义的导出名称，而不是要导出的目标列表，`<export>` 这样的实例是由 `install(TARGETS)` 创建的目标命名列表。

以下是如何在实践中使用这种简写的示例：

ch14/01-export/CMakeLists.txt (continued)

```
1 install(TARGETS calc EXPORT CalcTargets)
2 export(EXPORT CalcTargets
3     FILE "${EXPORT_DIR}/CalcTargets2.cmake"
4     NAMESPACE Calc::
5 )
```

这段代码的工作方式与前面的示例类似，但现在它在 `export()` 和 `install()` 命令之间共享了同一个目标列表。

两种生成导出文件的方法产生类似的结果，包括一些样板代码和定义目标的几行。将 `<build-tree>` 设置为构建树路径后，将创建一个类似以下的目标导出文件：

<build-tree>/cmake/CalcTargets.cmake (片段)

```
1 # Create imported target Calc::calc
2 add_library(Calc::calc STATIC IMPORTED)
3 set_target_properties(Calc::calc PROPERTIES
```

```

4 INTERFACE_INCLUDE_DIRECTORIES
5   "/<source-tree>/include"
6 )
7 # Import target "Calc::calc" for configuration ""
8 set_property(TARGET Calc::calc APPEND PROPERTY
9   IMPORTED_CONFIGURATIONS NOCONFIG
10 )
11 set_target_properties(Calc::calc PROPERTIES
12   IMPORTED_LINK_INTERFACE_LANGUAGES_NOCONFIG "CXX"
13   IMPORTED_LOCATION_NOCONFIG "/<build-tree>/libcalc.a"
14 )

```

通常，我们不会编辑甚至打开这个文件，但重要的是文件中的路径为硬编码的（参见突出显示的行）。在当前的形式下，构建的项目不可重定位。要改变这一点，需要采取一些其他步骤。下一节中，我们将解释什么是重定位，及其重要性。

14.3. 安装项目

在第 1 章中，我们指出 CMake 提供了一个命令行模式，用于在系统上安装构建的项目：

```
cmake --install <dir> [<options>]
```

这里，`<dir>` 是指向生成的构建树路径（必需的）。包括：

- `--config <cfg>`: 这用于为多配置生成器选择构建配置。
- `--component <comp>`: 这将安装限制在给定的组件内。
- `--default-directory-permissions <permissions>`: 这为安装的目录设置默认权限（格式为 `<u=rwx,g=rx,o=rx>`）。
- `--install-prefix <prefix>`: 这指定非默认的安装路径（在 `CMAKE_INSTALL_PREFIX` 变量中）。在类 Unix 系统中默认为 `/usr/local`，在 Windows 中默认为 `C:/Program Files/${PROJECT_NAME}`。在 CMake 3.21 之前，需要使用一个不太明确的选项：`--prefix <prefix>`。
- `-v, --verbose`: 这增加了输出信息的详细程度（也可以通过设置 `VERBOSE` 环境变量实现）。

安装通常涉及将生成的工件和必要的依赖项复制到系统目录。使用 CMake 为所有 CMake 项目引入了一个方便的安装标准：

- 为不同类型的工件提供特定于平台的安装路径（遵循 GNU 编码标准）。
- 通过生成目标导出文件来增强安装过程，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，包装目标导出文件和作者定义的特定于包的 CMake 宏和函数。

这些功能非常强大，节省了大量时间并简化了以这种方式准备的项目使用。执行基本安装的第一步是将构建的工件复制到目标目录。这就引出了 `install()` 命令及其各种模式：

- `install(TARGETS)`: 这安装输出工件，如库和可执行文件。
- `install(FILES|PROGRAMS)`: 这安装单个文件并设置它们的权限。这些文件不需要是逻辑目标的一部分。
- `install(DIRECTORY)`: 这安装整个目录。
- `install(SCRIPT|CODE)`: 这在安装过程中运行 CMake 脚本或代码片段。
- `install(EXPORT)`: 这生成并安装目标导出文件。
- `install(RUNTIME_DEPENDENCY_SET <set-name> [...])`: 这安装项目中定义的运行时依赖项集。
- `install(IMPORTED_RUNTIME_ARTIFACTS <target>... [...])`: 这查询导入的目标以获取运行时工件并安装它们。

将这些命令添加到列表文件中，会在构建树中生成一个 `cmake_install.cmake` 文件。虽然可以手动用 `cmake -P` 调用这个脚本，但这并不推荐。该文件旨在由 CMake 在执行 `cmake --install` 时内部使用。

每个 `install()` 模式都有完整的一套选项，其中一些在模式之间共享：

- `DESTINATION`: 这指定安装路径。相对路径会以 `CMAKE_INSTALL_PREFIX` 为前缀，而绝对路径会按原样使用（并且不受 `cpack` 支持）。
- `PERMISSIONS`: 这在支持的平台设置文件权限。可用的值包括 `OWNER_READ`, `OWNER_WRITE`, `OWNER_EXECUTE`, `GROUP_READ`, `GROUP_WRITE`, `GROUP_EXECUTE`, `WORLD_READ`, `WORLD_WRITE`, `WORLD_EXECUTE`, `SETUID` 和 `SETGID`。安装时创建的默认目录权限可以通过 `CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS` 变量来设置。
- `CONFIGURATIONS`: 这指定配置（`Debug`, `Release`）。跟随此关键字的选项仅当当前构建配置在列表中时才适用。
- `OPTIONAL`: 避免安装文件不存在时出现错误。

两个共享选项，`COMPONENT` 和 `EXCLUDE_FROM_ALL`，用于特定于组件的安装。这些将在本章后面的“定义组件”部分讨论。现在，来看看第一个安装模式：`install(TARGETS)`。

14.3.1. 安装逻辑目标

由 `add_library()` 和 `add_executable()` 定义的目标可以通过 `install(TARGETS)` 命令轻松安装，将构建系统生成的工件复制到适当的目标目录，并为设置合适的文件权限。此模式的通用签名为：

```
install(TARGETS <target>... [EXPORT <export-name>]
        [<output-artifact-configuration> ...]
        [INCLUDES DESTINATION [<dir> ...]])
)
```

指定了初始模式标识符，即 `TARGETS` 之后，必须提供想要安装的目标列表。在这里，可以选择性地使用 `EXPORT` 选项将它们分配给一个命名的导出，这可以在 `export(EXPORT)`

和 `install(EXPORT)` 中使用，以生成目标导出文件。然后，必须配置输出工件的安装（按类型分组）。可选地，可以提供一个目录列表，这些目录将添加到每个目标在其 `INTERFACE_INCLUDE_DIRECTORIES` 属性中的目标导出文件中。

`[<output-artifact-configuration>...]` 提供了一个配置块列表。单个块的完整语法如下：

```
<TYPE> [DESTINATION <dir>]
[PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[NAMELINK_COMPONENT <component>]
[OPTIONAL] [EXCLUDE_FROM_ALL]
[NAMELINK_ONLY|NAMELINK_SKIP]
```

命令要求每个输出工件块以 `<TYPE>` 开头（这是唯一必需的元素）。CMake 识别以下几种类型：

- ARCHIVE：静态库（.a）和 Windows 系统上的 DLL 导入库（.lib）。
- LIBRARY：共享库（.so），但不包括 DLL。
- RUNTIME：可执行文件和 DLL。
- OBJECTS：来自 OBJECT 库的对象文件。
- FRAMEWORK：设置了 FRAMEWORK 属性的静态和共享库（从 ARCHIVE 和 LIBRARY 中排除），是 macOS 特定的。
- BUNDLE：标记有 MACOSX_BUNDLE 的可执行文件（也不属于 RUNTIME）。
- FILE_SET <set>：指定给目标的文件集 <set> 中的文件。可以是 C++ 头文件或 C++ 模块头文件（自 CMake 3.23 起）。
- PUBLIC_HEADER，PRIVATE_HEADER，RESOURCE：在目标属性中指定相同名称的文件（在 Apple 平台上，应该设置在 FRAMEWORK 或 BUNDLE 目标上）。

CMake 文档声称，如果只配置一个工件类型（例如，LIBRARY），只有这个类型会安装。对于 CMake 版本 3.26.0，所有工件都会安装。这可以通过为所有不需要的工件类型指定 `EXCLUDE_FROM_ALL` 来解决。

Note

单个 `install(TARGETS)` 命令可以有多个工件配置块。但是请注意，每个调用中只能指定每种类型的一个。也就是说，如果想要为 Debug 和 Release 配置配置不同的 ARCHIVE 工件的目标，必须进行两次单独的 `install(TARGETS ...ARCHIVE)` 调用。

也可以省略类型名称，并为所有工件指定选项。安装将针对这些目标产生的每个文件执行，而不管其类型：

```
1 install(TARGETS executable, static_lib1
2   DESTINATION /tmp
3 )
```

许多情况下，由于内置的默认值，需要显式提供 DESTINATION，但在处理不同平台时需要记住一些注意事项。

利用不同平台上的默认目标目录

当 CMake 安装你的项目的文件时，会将它们复制到系统中的特定目录。不同类型的文件属于不同的目录。这个目录由以下公式确定：

```
1 ${CMAKE_INSTALL_PREFIX} + ${DESTINATION}
```

如前一部分所述，可以为安装显式提供 DESTINATION 组件，或者让 CMake 根据工件的类型使用内置的默认值：

工件类型	内置默认值	安装目录变量
RUNTIME	bin	CMAKE_INSTALL_BINDIR
LIBRARY ARCHIVE	lib	CMAKE_INSTALL_LIBDIR
PUBLIC_HEADER PRIVATE_HEADER FILE_SET (类型 HEADERS)	include	CMAKE_INSTALL_INCLUDEDIR

表 14.1：每种工件类型的默认目标位置

虽然默认路径很有用，但并不总是合适。例如，CMake 将库的默认 DESTINATION 设置为 lib。对于类 Unix 系统，库的完整路径为 /usr/local/lib，而在 Windows 上则类似于 C:\Program Files (x86)\<project-name>\lib。然而，这对于支持多架构的 Debian 来说并不理想，当 INSTALL_PREFIX 是 /usr 时，需要一个特定于架构的路径（例如，i386-linux-gnu）。为每个平台确定正确的路径对于类 Unix 系统来说是一个常见的挑战。为了解决这个问题，请遵循 GNU 编码标准，该链接添加在本章末尾的进一步阅读部分。

我们可以通过设置 CMAKE_INSTALL_<DIRTYPE>_DIR 变量来覆盖每个值的默认目标。不是开发一个算法来检测平台，并将适当的路径分配给安装目录变量，而是使用 CMake 的 GNUInstallDirs 实用模块。该模块通过相应地设置安装目录变量来处理大多数平台，只需在 install() 命令之前使用 include() 包含它即可。

需要自定义配置的用户可以通过命令行参数覆盖安装目录变量：

```
-DCMAKE_INSTALL_BINDIR=/path/in/the/system
```

然而，安装库的公共头文件仍然存在挑战。

处理公共头文件

CMake 中管理公共头文件时，最佳实践是将它们存储在一个指示其来源，并引入命名空间的目录中，例如 /usr/local/include/calc。这允许它们在 C++ 项目中使用包含指令来使用：

```
1 #include <calc/basic.h>
```

大多数预处理器将尖括号指令解释为请求扫描标准系统目录，可以使用 `GNUInstallDirs` 模块来自动填充安装路径的 `DESTINATION` 部分，确保头文件最终位于 `include` 目录中。

自从 CMake 3.23.0 起，可以使用 `target_sources()` 命令和 `FILE_SET` 关键字明确地将头文件添加到适当的目标以进行安装。这种方法更为可取，其负责了头文件的重新定位。以下是语法：

```
target_sources(<target>
    [<PUBLIC|PRIVATE|INTERFACE>
        [FILE_SET <name> TYPE <type> [BASE_DIR <dir>] FILES]
        <files>...
    ]...
)
```

假设头文件位于 `src/include/calc` 目录中：

ch14/02-install-targets/src/CMakeLists.txt (片段)

```
1 add_library(calc STATIC basic.cpp)
2 target_include_directories(calc INTERFACE include)
3 target_sources(calc PUBLIC FILE_SET HEADERS
4                 BASE_DIRS include
5                 FILES include/calc/basic.h
6 )
```

前面的代码片段定义了一个名为 `HEADERS` 的新目标文件集。我们在这里使用了一个特殊情况：如果文件集的名称与可用类型之一匹配，CMake 会假定文件集是此类类型，从而消除了明确定义类型的需要。如果使用不同的名称，请记住使用适当的 `TYPE <TYPE>` 关键字来定义 `FILE_SET` 的类型。

定义了文件集后，我们可以在安装命令中使用它：

ch14/02-install-targets/src/CMakeLists.txt (续)

```
1 ...
2
3 include(GNUInstallDirs)
4 install(TARGETS calc ARCHIVE FILE_SET HEADERS)
```

包含了 `GNUInstallDirs` 模块并配置了 `calc` 静态库，及其头文件的安装。以安装模式运行 `cmake` 按预期工作：

```
# 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/basic.h
```

对 FILE_SET HEADERS 关键字的支持是相对较新的更新，但并非所有环境都会提供较新版本的 CMake。

如果还停留在 3.23 之前的版本，需要以分号分隔的列表形式指定公共头文件（在库目标的 PUBLIC_HEADER 属性中），并手动处理重新定位：

ch14/03-install-targets-legacy/src/CMakeLists.txt (片段)

```
1 add_library(calc STATIC basic.cpp)
2 target_include_directories(calc INTERFACE include)
3 set_target_properties(calc PROPERTIES
4   PUBLIC_HEADER src/include/calc/basic.h
5 )
```

还需要更改目标目录，以便在包含路径中包含库名称：

ch14/02-install-targets-legacy/src/CMakeLists.txt (续)

```
1 ...
2 include(GNUInstallDirs)
3 install(TARGETS calc
4   ARCHIVE
5   PUBLIC_HEADER
6   DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
7 )
```

因为指定在 PUBLIC_HEADER 属性中的文件不会保留目录结构，所以需要在路径中插入 /calc。即使嵌套在不同的基本目录中，也都会安装到同一个目标位置。因为这个缺点，才进行了 FILE_SET 的开发。

现在，了解了如何处理大多数安装情况，但是应该如何处理更高级的场景呢？

14.3.2. 低层安装

现代 CMake 正在远离直接操作文件。理想情况下，应该将文件添加到逻辑目标中，使用它作为更高层次的抽象来代表所有底层资源：源文件、头文件、资源、配置等。主要优点是代码的简洁性；通常，将文件添加到目标只需要更改不超过一行。

不幸的是，将每个安装的文件添加到目标并不可能或方便的。这种情况下，有三个选项可用：`install(FILES)`、`install(PROGRAMS)` 和 `install(DIRECTORY)`。

使用 `install(FILES)` 和 `install(PROGRAMS)` 安装

`FILES` 和 `PROGRAMS` 模式非常相似，可以用来安装各种资产，包括公共头文件、文档、shell 脚本、配置，以及像图像、音频文件和数据集这样的运行时资源。

以下是命令签名：

```

install(<FILES|PROGRAMS> files...
    TYPE <type> | DESTINATION <dir>
    [PERMISSIONS permissions...]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL]
)

```

FILES 和 PROGRAMS 之间的主要区别在于复制文件时设置的默认文件权限。install(PROGRAMS) 为所有用户设置 EXECUTE，而 install(FILE) 则不设置（尽管两者都会设置 OWNER_WRITE, OWNER_READ, GROUP_READ 和 WORLD_READ）。

可以使用可选的 PERMISSIONS 关键字来修改这种行为，然后选择领先的关键字 (FILE 或 PROGRAMS) 作为安装内容的指示器。我们已经介绍了 PERMISSIONS、CONFIGURATIONS 和 OPTIONAL 的用法。COMPONENT 和 EXCLUDE_FROM_ALL 将在后面的定义组件部分讨论。

初始关键字之后，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。请记住，如果文件路径以生成器表达式开头，则必须是绝对路径。

下一个必需的关键字是 TYPE 或 DESTINATION。可以选择明确提供 DESTINATION 路径，或者要求 CMake 为特定 TYPE 文件查找它。与 install(TARGETS) 不同，在此上下文中，TYPE 并不选择要安装的提供的文件子集。尽管如此，安装路径的计算遵循相同的模式（其中 + 符号表示平台特定的路径分隔符）：

1 \${CMAKE_INSTALL_PREFIX} + \${DESTINATION}

同样，每种 TYPE 都将具有内置的默认值：

文件类型	内置默认值	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
DATA	\$DATAROOT	CMAKE_INSTALL_DATADIR
INFO	\$DATAROOT/info	CMAKE_INSTALL_INFODIR
LOCALE	\$DATAROOT/locale	CMAKE_INSTALL_LOCALEDIR
MAN	\$DATAROOT/man	CMAKE_INSTALL_MANDIR
DOC	\$DATAROOT/doc	CMAKE_INSTALL_DOCDIR

表 14.2：每种文件类型的内置默认值

这里的行为遵循了之前描述的“为不同平台利用默认目标目录”子节中的相同原则：如果未为这种文件 TYPE 设置安装目录变量，CMake 将提供一个内置的默认路径。为了可移植性，可以使用 `GNUInstallDirs` 模块。

表中的某些内置猜测带有安装目录变量的前缀：

- `$OCALSTATE` 是 `CMAKE_INSTALL_LOCALSTATEDIR`, 默认为 `var`。
- `$DATAROOT` 是 `CMAKE_INSTALL_DATAROOTDIR`, 默认为 `share`。

与 `install(TARGETS)` 一样，`GNUInstallDirs` 模块将为不同平台提供特定的安装目录变量。来看一个例子：

ch14/04-install-files/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(InstallFiles CXX)
3 include(GNUInstallDirs)
4 install(FILES
5     src/include/calc/basic.h
6     src/include/calc/nested/calc_extended.h
7     DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/calc
8 )
```

，CMake 将两个仅包含头文件的库 (`basic.h` 和 `nested/calc_extended.h`)，安装到系统 `include` 目录中的项目特定子目录中。

Note

从 `GNUInstallDirs` 源码中了解，`CMAKE_INSTALL_INCLUDEDIR` 对所有支持平台的值相同。然而，仍然推荐使用，以提高可读性和与更动态变量的一致性。例如，`CMAKE_INSTALL_LIBDIR` 根据架构和发行版而变化——`lib`、`lib64` 或 `lib/`。

从 CMake 3.20 开始，可以使用 `RENAME` 关键字与 `install(FILES)` 和 `install(PROGRAMS)` 命令一起使用。这个关键字后面必须跟着一个新的文件名，并且只有在命令安装单个文件时才有效。

本节中的示例演示了将文件安装到适当目录的便利性。然而，存在一个问题——观察安装输出：

```
# cmake -S <source-tree> -B <build-tree>
# cmake --build <build-tree>
# cmake --install <build-tree>
-- Install configuration: ""
-- Installing: /usr/local/include/calc/basic.h
-- Installing: /usr/local/include/calc/calc_extended.h
```

两个文件都安装到了同一个目录中，而不管它们的原始嵌套结构如何。有时，这并不理想。在下一节中，我们将探讨如何处理这种情况。

处理整个目录

如果向安装命令添加单个文件不合适，可以处理整个目录。`install(DIRECTORY)` 模式就是为了这个目的设计的，指定的目录原样复制到所选的目的地。

```
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...]] [...]
)
```

许多选项与 `install(FILES)` 和 `install(PROGRAMS)` 中的选项类似，并按相同方式工作。一个关键的细节是，如果 `DIRECTORY` 关键字后面的路径不以/结尾，路径的最后一个目录将附加到目的地。例如：

```
1 install(DIRECTORY aaa DESTINATION /xxx)
```

这个命令会创建一个目录，`/xxx/aaa`，并将 `aaa` 的内容复制到其中。相比之下，以下命令会将 `aaa` 的内容直接复制到`/xxx`：

```
1 install(DIRECTORY aaa/ DESTINATION /xxx)
```

`install(DIRECTORY)` 还引入了其他在文件上不可用的机制：

- 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，禁用了安装过程中的输出诊断。当有大量文件需要安装，并且打印所有文件会过于嘈杂时，这个选项非常有用。

关于权限，`install(DIRECTORY)` 支持三个选项：

- `USE_SOURCE_PERMISSIONS` 设置安装文件的原有文件权限，只有在 `FILE_PERMISSIONS` 未设置时才有效。
- `FILE_PERMISSIONS` 允许指定为安装的文件和目录设置的权限，默认权限是 `OWNER_WRITE`、`OWNER_READ`、`GROUP_READ` 和 `WORLD_READ`。
- `DIRECTORY_PERMISSIONS` 与 `FILE_PERMISSIONS` 类似，但它将为所有用户设置 `EXECUTE` 权限（这是因为 Unix-like 系统中目录的 `EXECUTE` 权限表示列出其内容的权限）。

注意，CMake 在那些不支持权限选项的平台上将忽略这些选项。通过在每个过滤表达式后添加 `PERMISSIONS` 关键字并指定一个权限列表，可以实现更细致的权限控制。匹配这些的文件或目录将获得指定的权限。

现在，让我们看看过滤器或“通配”表达式。控制源目录中哪些文件/目录将安装，并遵循以下语法：

```
PATTERN <pat> | REGEX <reg> [EXCLUDE] [PERMISSIONS <perm>]
```

有两种匹配方法可供选择：

- 使用 `PATTERN`，这是更简单的选项，可以提供一个带有? 占位符（匹配任何字符）和 * 通配符（匹配任何字符串）的模式。只有以结尾的路径才会匹配。
- `REGEX` 选项更高级，支持正则表达式。允许匹配路径的部分，尽管 ^ 和 \$ 锚点仍然可以表示路径的开始和结束。

可选地，第一个过滤器之前可以设置 `FILES_MATCHING` 关键字，指定过滤器将应用于文件而不是目录。

请记住两个事项：

- `FILES_MATCHING` 需要一个包容性过滤器。可以排除一些文件，但如果不同时包含一些文件，则不会有文件被复制。然而，所有目录都将创建，无论如何过滤。
- 所有子目录默认都会包含，只能过滤掉。

对于每个过滤方法，可以选择使用 `EXCLUDE` 命令来排除匹配的路径（这仅在未使用 `FILES_MATCHING` 时有效）。

为所有匹配的路径设置特定权限，可以通过在过滤器后添加 `PERMISSIONS` 关键字并指定一个权限列表来实现。通过一个例子来探索这一点，以不同的方式安装三个目录。有一些用于运行时使用的静态数据文件：

```
data
- data.csv
```

还需要一些位于 `src` 目录中的公共头文件，其中还有其他不相关的文件：

```
src
- include
  - calc
    - basic.h
    - ignored
      - empty.file
  - nested
    - calc_extended.h
```

最后，需要两个不同嵌套级别的配置文件。为了增加趣味性，`/etc/calc/` 的内容只能被文件所有者访问：

```
etc
- calc
  - nested.conf
- sample.conf
```

要安装包含静态数据文件的目录，从项目中最基本的 `install(DIRECTORY)` 命令开始：

ch14/05-install-directories/CMakeLists.txt (片段)

```
1 cmake_minimum_required(VERSION 3.26)
2 project(InstallDirectories CXX)
3 install(DIRECTORY data/ DESTINATION share/calc)
```

这个命令将简单地取我们数据目录的所有内容，并将其放入 `${CMAKE_INSTALL_PREFIX}` 和 `share/calc`。我们的源路径以/符号结束，以表明不想复制数据目录本身，只想复制其内容。

第二个情况正好相反：不添加源路径的尾随/，因为目录应该包含。这是因为依赖于 `INCLUDE` 文件类型的系统特定路径，这是由 `GNUInstallDirs` 提供的（注意 `INCLUDE` 和 `EXCLUDE` 关键字代表不相关的概念）：

ch14/05-install-directories/CMakeLists.txt (片段)

```
1 ...
2 include(GNUInstallDirs)
3 install(DIRECTORY src/include/calc TYPE INCLUDE
4         PATTERN "ignored" EXCLUDE
5         PATTERN "calc_extended.h" EXCLUDE
6     )
```

此外，还从这次操作中排除了两个路径：整个 `ignored` 目录和所有以 `calc_extended.h` 结尾的文件（记住 `PATTERN` 是如何工作的）。

第三个案例安装了一些默认配置文件并设置了其权限：

ch14/05-install-directories/CMakeLists.txt (片段)

```
1 install(DIRECTORY etc/ TYPE SYSCONF
2         DIRECTORY_PERMISSIONS
3             OWNER_READ OWNER_WRITE OWNER_EXECUTE
4             PATTERN "nested.conf"
5             PERMISSIONS OWNER_READ OWNER_WRITE
6     )
```

避免在 `SYSCONF` 路径中添加 `etc` (`GNUInstallDirs` 已经提供了这个路径) 以避免重复。我们设置了两个权限规则：子目录只能由所有者编辑和列出，以 `nested.conf` 结尾的文件只能由所有者编辑。

安装目录覆盖了各种用例，但对于其他高级场景（如安装后配置），可能需要外部工具。我们如何集成它们？

14.3.3. 安装过程中调用脚本

如果曾在类 Unix 系统上安装过共享库，可能会记得需要指导动态链接器扫描可信目录，并使用 `ldconfig` 构建其缓存（请参阅扩展阅读部分以获取参考文献）。为了实现完全自动化的安装，CMake 提供了 `install(SCRIPT)` 和 `install(CODE)` 模式。这是完整的语法：

```
install([[SCRIPT <file>] [CODE <code>]]
        [ALL_COMPONENTS | COMPONENT <component>]
        [EXCLUDE_FROM_ALL] [...]
    )
```

在 `SCRIPT` 和 `CODE` 模式之间进行选择，并提供必要的参数——在安装过程中要运行的 CMake 脚本路径或要执行的 CMake 代码片段。为了说明，将修改 `02-install-targets` 示例以构建共享库：

ch14/06-install-code/src/CMakeLists.txt

```
1 add_library(calc SHARED basic.cpp)
2 target_include_directories(calc INTERFACE include)
3 target_sources(calc PUBLIC FILE_SET HEADERS
4                 BASE_DIRS include
5                 FILES include/calc/basic.h
6 )
```

安装脚本中将工件类型从 `ARCHIVE` 改为 `LIBRARY`，并在之后添加运行 `ldconfig` 的逻辑：

ch14/06-install-code/CMakeLists.txt (片段)

```
1 install(TARGETS calc LIBRARY FILE_SET HEADERS)
2 if (UNIX)
3     install(CODE "execute_process(COMMAND ldconfig)")
4 endif()
```

`if()` 条件确保命令适用于操作系统 (`ldconfig` 不应在 Windows 或 macOS 上执行)，提供的代码必须在 CMake 中语法正确（错误仅在安装过程中出现）。

运行安装命令后，通过打印缓存库来确认其成功：

```
# 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/basic.h
# ldconfig -p | grep libcalc
libcalc.so (libc6,x86-64) => /usr/local/lib/libcalc.so
```

SCRIPT 和 CODE 模式都支持生成器表达式，为这个命令增加了灵活性。可以用于各种目的：打印用户消息、验证成功安装、广泛配置、文件签名等。

接下来，让我们深入探讨 CMake 安装中的运行时依赖管理，这是 CMake 的最新特性之一。

14.3.4. 安装运行时依赖项

已经涵盖了所有可安装工件及其相应的命令，最后一个要讨论的主题是运行时依赖。执行程序和共享库通常依赖于其他必须在系统中存在的库，并在程序初始化时动态加载。从 CMake 3.21 版本开始，CMake 可以为每个目标构建这些所需库的列表，并通过引用二进制文件的适当部分在构建时捕获它们的位置。这个列表随后可以用来在系统中安装这些运行时工件，以供将来使用。

对于项目中的目标，这可以通过以下两个步骤实现：

```
install(TARGETS ... RUNTIME_DEPENDENCY_SET <set-name>)
install(RUNTIME_DEPENDENCY_SET <set-name> <arg>...)
```

或者，通过一个具有相同效果的单一命令来完成：

```
install(TARGETS ... RUNTIME_DEPENDENCIES <arg>...)
```

如果目标是通过导入而不是在项目中定义的，运行时依赖项可以按照以下方式安装：

```
install(IMPORTED_RUNTIME_ARTIFACTS <target>...)
```

上述代码段可以通过添加 RUNTIME_DEPENDENCY_SET <set-name> 参数来扩展，以创建一个可以稍后用于 `install(RUNTIME_DEPENDENCY_SET)` 命令的命名引用。

如果项目可以从这个功能中受益，我建议查看 `install()` 命令的官方文档以了解更多。

现在，了解了可以在系统上，以各种方式安装文件的所有不同方法，接下来来探讨如何将它们转换为其他 CMake 项目本地的可用包。

14.4. 创建可重用的包

我们已经在前面的章节中广泛使用了 `find_package()`，并观察到了其便利性和简洁性。为了让项目可以通过这个命令访问，需要完成几个步骤，以便 CMake 可以将项目视为一个包：

1. 使目标可重定位。
2. 将目标导出文件安装到标准位置。
3. 为该包创建一个配置文件。
4. 为该包生成一个版本文件。

从头开始：为什么目标需要是可重定位的，如何做呢？

14.4.1. 了解可重定位目标的问题

安装解决了许多问题，但也引入了一些复杂性。CMAKE_INSTALL_PREFIX 是平台特定的，可以在安装阶段由用户通过`--install-prefix` 命令行参数设置。挑战在于，目标导出文件在安装之前生成，即在构建阶段，此时安装的目的地是未知的。

ch14/03-install-targets-legacy/src/CMakeLists.txt

```
1 add_library(calc STATIC basic.cpp)
2 target_include_directories(calc INTERFACE include)
3 set_target_properties(calc PROPERTIES
4   PUBLIC_HEADER src/include/calc/basic.h
5 )
```

这个例子中，特别将 `include` 目录添加到 `calc` 的包含目录中。由于这是一个相对路径，CMake 的导出目标生成隐式地将这个路径与 `CMAKE_CURRENT_SOURCE_DIR` 变量的内容前置。指向这个列表文件所在的目录。

问题来了：安装后，项目不能依赖源或构建树中的文件。包括库头文件在内的所有内容都复制到一个共享位置，例如 Linux 上的 `/usr/lib/calc/`。在这个代码段中定义的目标不适合在另一个项目中使用，因为它的包含目录路径仍然指向其源树。

CMake 通过生成器表达式解决这个问题，这些表达式根据上下文替换为参数或一个空字符串：

- `$<BUILD_INTERFACE:...>`: 常规构建中评估为'...' 参数，但在安装时排除它。
- `$<INSTALL_INTERFACE:...>`: 安装时评估为'...' 参数，但在常规构建时排除它。
- `$<BUILD_LOCAL_INTERFACE:...>`: 当在同一构建系统中的另一个目标使用时评估为'...' 参数（在 CMake 3.26 中添加）。

这些表达式允许将使用哪个路径的决定，推迟到构建和安装的过程的后期阶段。以下是如何在实践中使用：

ch14/07-install-export-legacy/src/CMakeLists.txt (片段)

```
1 add_library(calc STATIC basic.cpp)
2 target_include_directories(calc INTERFACE
3   "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>"
4   "$<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>"
5 )
6 set_target_properties(calc PROPERTIES
7   PUBLIC_HEADER "include/calc/basic.h"
8 )
```

`target_include_directories()` 中，我们关注最后两个参数。使用的生成器表达式是互斥的，所以在最后一步中只使用其中一个参数，另一个将擦除。

对于常规构建，`calc` 目标的 `INTERFACE_INCLUDE_DIRECTORIES` 属性将使用第一个参数展开：

```
1 "/root/examples/ch14/07-install-export/src/include" ""
```

安装时，第二个参数值将展开：

```
1 "" "/usr/lib/calc/include"
```

Note

最终值中不存在引号；在这里添加是为了清晰地表示空文本值。

关于 CMAKE_INSTALL_PREFIX：不应将其用作目标中指定的路径的组成部分。将在构建阶段进行计算，使路径变为绝对路径，可能与安装阶段提供的路径不同（如果使用了--install-prefix 选项）。相反，使用 \$<INSTALL_PREFIX> 生成器表达式：

```
1 target_include_directories(my_target PUBLIC
2   $<INSTALL_INTERFACE:$<INSTALL_PREFIX>/include/MyTarget>
3 )
```

或者，可以使用相对路径，将与正确的安装前缀前置：

```
1 target_include_directories(my_target PUBLIC
2   $<INSTALL_INTERFACE:include/MyTarget>
3 )
```

有关更多示例和信息，请查阅官方文档（可以在“扩展阅读”部分找到链接）。

现在目标是兼容安装的，我们可以安全地生成，并安装它们的目标导出文件。

14.4.2. 安装目标导出文件

之前在“无需安装的导出”一节中提到了目标导出文件。安装目标导出文件的过程非常相似，创建它们的命令语法也是如此：

```
1 install(EXPORT <export-name> DESTINATION <dir>
2   [NAMESPACE <namespace>] [[FILE <name>.cmake] |
3   [PERMISSIONS permissions...]]
4   [CONFIGURATIONS [Debug|Release|...]]
5   [EXPORT_LINK_INTERFACE_LIBRARIES]
6   [COMPONENT <component>]
7   [EXCLUDE_FROM_ALL])
```

该命令将创建并安装一个命名导出，该导出必须使用 `install(TARGETS)` 命令定义。这里的关键区别是，生成的导出文件将包含使用 `INSTALL_INTERFACE` 生成器表达式计算的目标路径，这与 `export(EXPORT)` 不同，后者使用 `BUILD_INTERFACE`。所以，需要小心我们的包含文件和其他相对引用的文件。

再次，如果 CMake 3.23 或更高版本正确使用 FILE_SET_HEADERS，这不会成为问题。来看看如何为 ch14/02-installexport 示例中的目标生成和安装导出文件。为此，必须在调用 `install(TARGETS)` 命令后，调用 `install(EXPORT)` 命令：

ch14/07-install-export/src/CMakeLists.txt

```
1 add_library(calc STATIC basic.cpp)
2 target_sources(calc
3     PUBLIC FILE_SET_HEADERS BASE_DIRS include
4     FILES "include/calc/basic.h"
5 )
6
7 include(GNUInstallDirs)
8 install(TARGETS calc EXPORT CalcTargets ARCHIVE FILE_SET_HEADERS)
9 install(EXPORT CalcTargets
10    DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
11    NAMESPACE Calc::
12 )
```

注意 `install(EXPORT)` 命令中对 `CalcTargets` 导出名称的引用。在构建树中运行 `cmake --install` 将导致导出文件在指定的目的地生成：

```
...
-- Installing: /usr/local/lib/calc/cmake/CalcTargets.cmake
-- Installing: /usr/local/lib/calc/cmake/CalcTargets-noconfig.cmake
```

如果需要覆盖默认的目标导出文件名 (`<export name>.cmake`)，可以通过添加 `FILE new-name.cmake` 参数来更改它（文件名必须以.cmake 结尾）。

不要混淆这一点——目标导出文件不是配置文件，所以暂时还不能使用 `find_package()` 来使用已安装的目标。如果必要，可以直接 `include()` 导出文件。所以，如何定义一个可以让其他项目使用的包呢？

14.4.3. 编写基本的配置文件

完整的包定义包括目标导出文件、包的配置文件和包的版本文件。从技术上讲，为了使 `find_package()` 工作，唯一需要的是一个配置文件。作为包定义，负责提供任何包函数和宏、检查要求、查找依赖关系，并包括目标导出文件。

用户可以使用以下命令，在系统上的任何位置安装包：

```
# cmake --install <build tree> --install-prefix=<path>
```

这个前缀决定了安装的文件将复制到的位置。为了支持这一点，必须确保以下几点：

- 目标属性上的路径是可重定位的。
- 配置文件中使用的路径相对于它。

要使用那些安装在非默认位置的包，使用项目的需要通过在配置阶段提供 `<installation path>` 来设置 `CMAKE_PREFIX_PATH` 变量：

```
# cmake -B <build tree> -DCMAKE_PREFIX_PATH=<installation path>
```

`find_package()` 命令将按照平台特定的方式检查文档中概述的路径列表（请参阅“扩展阅读”部分）。在 Windows 和类 Unix 系统上检查的一个模式是：

```
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)
```

这表明将配置文件安装在类似于 `lib/calc/cmake` 的路径上应该可以工作，CMake 要求配置文件的名称以`-config.cmake` 或 `Config.cmake` 结尾才能找到。

我们将配置文件的安装添加到 `06-install-export` 示例中：

ch14/09-config-file/CMakeLists.txt (fragment)

```
1 ...
2 install(EXPORT CalcTargets
3   DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
4   NAMESPACE Calc::
5 )
6 install(FILES "CalcConfig.cmake"
7   DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
8 )
```

该命令从相同的源目录安装 `CalcConfig.cmake` (`CMAKE_INSTALL_LIBDIR` 将计算为平台上的正确 `lib` 路径)。

最简单的配置文件包含一行，包括目标导出文件：

ch14/09-config-file/CalcConfig.cmake

```
1 include("${CMAKE_CURRENT_LIST_DIR}/CalcTargets.cmake")
```

`CMAKE_CURRENT_LIST_DIR` 指的是配置文件所在的目录。在我们的示例中，`CalcConfig.cmake` 和 `CalcTargets.cmake` 安装在同一目录中（如 `install(EXPORT)` 设置），因此目标导出文件将正确包含。

为了验证我们包的可用性，将创建一个简单的项目，其中包含一个列表文件：

ch14/10-find-package/CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.26)
2 project(FindCalcPackage CXX)
3 find_package(Calc REQUIRED)
4 include(CMakePrintHelpers)
5 message("CMAKE_PREFIX_PATH: ${CMAKE_PREFIX_PATH}")
6 message("CALC_FOUND: ${Calc_FOUND}")
```

```
7 cmake_print_properties(TARGETS "Calc::calc" PROPERTIES
8   IMPORTED_CONFIGURATIONS
9   INTERFACE_INCLUDE_DIRECTORIES
10 )
```

为了测试这一点，首先构建并安装 `09-config-file` 示例到某个目录，然后构建 `10-find-package`，同时使用 `DCMAKE_PREFIX_PATH` 参数引用：

```
# cmake -S <source-tree-of-08> -B <build-tree-of-08>
# cmake --build <build-tree-of-08>
# cmake --install <build-tree-of-08>
# cmake -S <source-tree-of-09> -B <build-tree-of-09>
-DCMAKE_PREFIX_PATH=<build-tree-of-08>
```

这将产生以下输出（所有 `<*_tree-of_>` 占位符将替换为实际路径）：

```
CMAKE_PREFIX_PATH: <build-tree-of-08>
CALC_FOUND: 1
--
Properties for TARGET Calc::calc:
  Calc::calc.IMPORTED_CONFIGURATIONS = "NOCONFIG"
  Calc::calcINTERFACE_INCLUDE_DIRECTORIES = "<build-tree-of-08>/include"
-- Configuring done
-- Generating done
-- Build files have been written to: <build-tree-of-09>
```

此输出表明 `CalcTargets.cmake` 文件找到并正确包含，包括目录的路径遵循所选择的预设。这个解决方案适用于基本的打包情况。

14.4.4. 创建高级配置文件

如果需要管理多个目标导出文件，包括在配置文件中添加几个宏可能会很有用。`CMakePackageConfigHelpers` 实用模块提供了对 `configure_package_config_file()` 命令的访问。要使用它，请提供一个模板文件，该文件将用 CMake 变量进行插值，以生成包含两个嵌入式宏定义的配置文件：

- `set_and_check(<variable> <path>)`: 这类似于 `set()`，但它会检查 `<path>` 确实存在，否则会以 `FATAL_ERROR` 失败。建议用配置文件，以尽早检测错误的路径。
- `check_required_components(<PackageName>)`: 此命令添加到配置文件的末尾，验证是否找到了 `find_package(<package> REQUIRED <component>)` 中用户所需的所有组件。

配置文件生成期间，可以准备复杂目录树路径以进行安装。这是命令的签名：

```

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>]
)

```

该 `<template>` 文件将被插值变量后存储在 `<output>` 路径中。`INSTALL_DESTINATION` 路径用于将 `PATH_VARS` 中存储的路径转换为相对于安装目的地的相对路径，`INSTALL_PREFIX` 可以作为基本路径提供，以指示相对于它的 `INSTALL_DESTINATION`。

`NO_SET_AND_CHECK_MACRO` 和 `NO_CHECK_REQUIRED_COMPONENTS_MACRO` 选项告诉 CMake 不要将这些宏定义添加到生成的配置文件中。让我们在实践中看看这种生成方式，扩展 07-install-export 示例：

ch14/11-advanced-config/CMakeLists.txt (片段)

```

1 ...
2 install(EXPORT CalcTargets
3     DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
4     NAMESPACE Calc::
5 )
6 include(CMakePackageConfigHelpers)
7 set(LIB_INSTALL_DIR ${CMAKE_INSTALL_LIBDIR}/calc)
8 configure_package_config_file(
9     ${CMAKE_CURRENT_SOURCE_DIR}/CalcConfig.cmake.in
10    "${${CMAKE_CURRENT_BINARY_DIR}/CalcConfig.cmake}"
11    INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
12    PATH_VARS LIB_INSTALL_DIR
13 )
14 install(FILES "${${CMAKE_CURRENT_BINARY_DIR}/CalcConfig.cmake"
15    DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
16 )

```

前面的代码中：

1. 使用 `include()` 包含带有帮助器的实用模块。
2. 使用 `set()` 设置一个变量，该变量将用于创建可重定位路径。
3. 使用 `CalcConfig.cmake.in` 模板为构建树生成 `CalcConfig.cmake` 配置文件，并提供 `LIB_INSTALL_DIR` 作为变量名，计算为相对于 `INSTALL_DESTINATION` 或 `${CMAKE_INSTALL_LIBDIR}/calc/cmake` 的相对路径。
4. 将为构建树生成的配置文件传递给 `install(FILE)`。

`install(FILE)` 命令中的 `DESTINATION` 路径和 `configure_package_config_file()` 命令中的 `INSTALL_DESTINATION` 路径相同，确保了配置文件内部的正确相对路径计算。

最后，需要一个配置文件模板（通常以.in 结尾）：

ch14/11-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)
```

这个模板以 @PACKAGE_INIT@ 占位符开始。生成器将用 set_and_check 和 check_required_components 宏的定义填充。

下一行将 CALC_LIB_DIR 设置为通过 @PACKAGE_LIB_INSTALL_DIR@ 占位符传递的路径。CMake 将填充它，提供在列表文件中提供的 \$LIB_INSTALL_DIR，但计算为相对于安装路径的相对路径。随后，该路径用于 include() 命令以包含目标导出文件。最后，check_required_components() 验证是否找到了使用此包的项目所需的所有组件。推荐使用这个命令，即使包没有组件，也要确保用户只使用受支持的依赖项。否则，用户可能会错误地认为他们已经成功添加了组件（可能仅存在于包的新版本中）。

当以这种方式生成时，CalcConfig.cmake 配置文件看起来像这样：

```
1 ##### Expanded from @PACKAGE_INIT@ by  
2 configure_package_config_file() #####  
3 ##### Any changes to this file will be overwritten by the  
4 next CMake run #####  
5 ##### The input file was CalcConfig.cmake.in #####  
6  
7 get_filename_component(PACKAGE_PREFIX_DIR  
8     "${CMAKE_CURRENT_LIST_DIR}/../../.." ABSOLUTE)  
9  
10 macro(set_and_check _var _file)  
11     # ... removed for brevity  
12 endmacro()  
13 macro(check_required_components _NAME)  
14     # ... removed for brevity  
15 endmacro()  
16  
17 #####  
18 set_and_check(CALC_LIB_DIR "${PACKAGE_PREFIX_DIR}/lib/calc")  
19 include("${CALC_LIB_DIR}/cmake/CalcTargets.cmake")  
20 check_required_components(Calc)
```

以下图表展示了各种包文件之间的关系，有助于更好地理解这一主题：

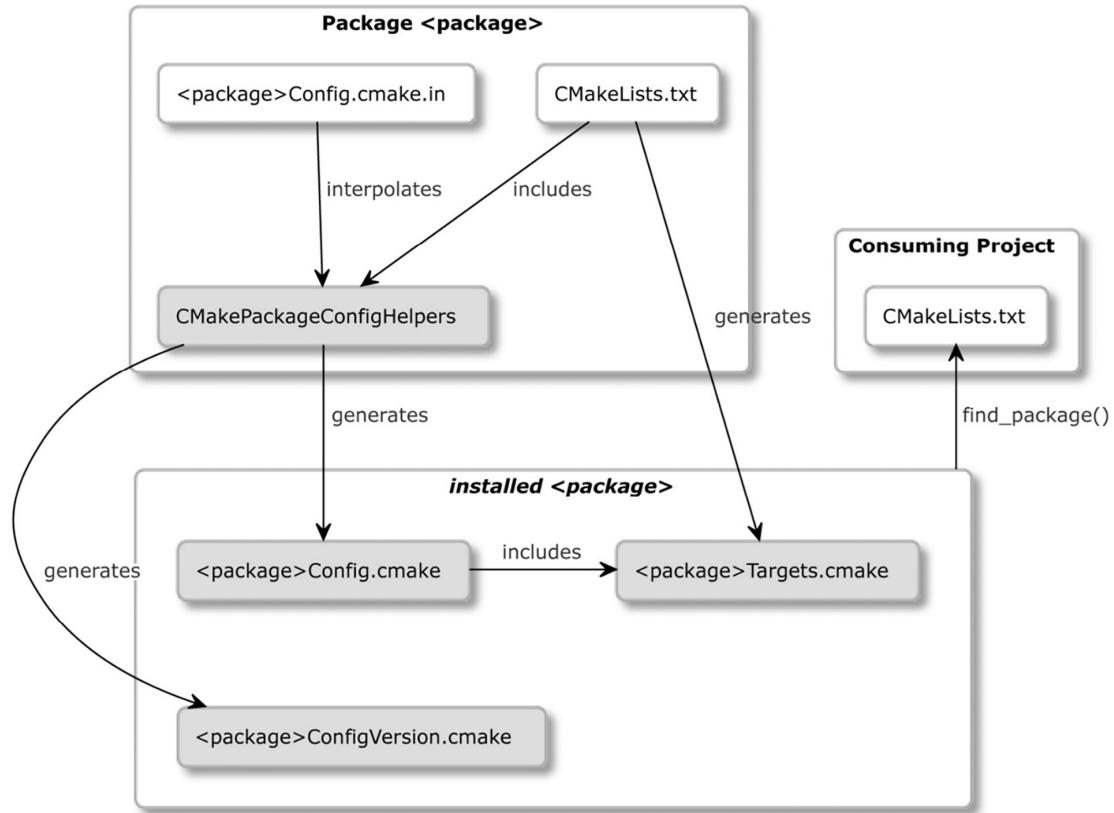


图 14.1：高级包的文件结构

包的所有必需的子依赖项也必须在包的配置文件中找到，这可以通过调用 CMakeFindDependencyMacro 助手中的 `find_dependency()` 宏来实现。

向使用项目的宏或函数的定义，都应该放在一个单独的文件中，并从包的配置文件中包含。有趣的是，CMakePackageConfigHelpers 也帮助生成包版本文件。接下来，我们将探讨这一点。

14.4.5. 生成包版本文件

随着包不断发展，获得新功能并淘汰旧功能，对于跟踪这些变化并在开发者使用的包中可访问的变更日志中记录这些变化至关重要。当需要特定功能时，使用包的开发者可以在 `find_package()` 中指定支持该功能的最低版本：

```
1 find_package(Calc 1.2.3 REQUIRED)
```

CMake 将搜索 Calc 的配置文件，并检查是否在同一目录中存在名为 `<configfile>-version.cmake` 或 `<config-file>Version.cmake` 的版本文件（例如，`CalcConfigVersion.cmake`）。该文件包含版本信息，并指定了与其他版本的兼容性。例如，即使没有安装确切的版本 1.2.3，可能会安装版本 1.3.5，它标记为与旧版本兼容。CMake 将接受这个包，知道是向后兼容的。

可以使用 CMakePackageConfigHelpers 实用模块通过调用 `write_basic_package_version_file()` 来生成包版本文件：

```

write_basic_package_version_file(
    <filename> [VERSION <ver>]
    COMPATIBILITY <AnyNewerVersion | SameMajorVersion |
    SameMinorVersion | ExactVersion>
    [ARCH_INDEPENDENT]
)

```

首先，提供作为文件名；确保它遵循之前讨论的命名规则。可选地，可以传递一个明确的版本号（主版本. 次版本. 补丁版本格式）。如果不提供，将使用项目（）命令中指定的版本（如果项目没有指定版本，将发生错误）。

`COMPATIBILITY` 关键字表示：

- `ExactVersion` 必须匹配所有三个版本组件，不支持范围版本：例如，`find_package(1.2.8…1.3.4)`。
- `SameMinorVersion` 如果前两个组件相同（忽略补丁版本）。
- `SameMajorVersion` 如果第一个组件相同（忽略次版本和补丁版本）。
- `AnyNewerVersion`，与它的名称相反，匹配较旧的版本：例如，版本 1.4.2 与 `find_package(<package> 1.2.8)` 兼容。

对于依赖于架构的包，需要精确的架构匹配。然而，对于不依赖于架构的包（如仅包含头文件的库或宏包），可以指定 `ARCH_INDEPENDENT` 关键字以跳过此检查。

以下代码示例展示了如何在 07-install-export 示例中提供的项目中提供版本文件：

ch14/12-version-file/CMakeLists.txt (片段)

```

1 cmake_minimum_required(VERSION 3.26)
2 project(VersionFile VERSION 1.2.3 LANGUAGES CXX)
3 ...
4 include(CMakePackageConfigHelpers)
5 write_basic_package_version_file(
6     "${CMAKE_CURRENT_BINARY_DIR}/CalcConfigVersion.cmake"
7     COMPATIBILITY AnyNewerVersion
8 )
9 install(FILES "CalcConfig.cmake"
10        "${CMAKE_CURRENT_BINARY_DIR}/CalcConfigVersion.cmake"
11        DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
12 )

```

方便起见，在文件顶部配置包的版本，在 `project()` 命令中，从简短的 `project(<name><languages>)` 语法切换到明确的、完整的语法，通过添加 `LANGUAGE` 关键字。

包含助手模块后，生成版本文件并将其与 `CalcConfig.cmake` 一起安装。通过省略 `VERSION` 关键字，使用 `PROJECT_VERSION` 变量。该包标记为完全向后兼容，与 `COMPATIBILITY AnyNewerVersion` 兼容，这将在与 `CalcConfig.cmake` 相同的安装位置安装包版本文件。就这样——包已经完全配置好了。

有了这些，就结束了关于包创建的主题。现在知道如何处理重定位，以及为什么它很重要；如何安装目标导出文件，以及如何编写配置和版本文件。

下一节中，我们将探讨组件及其在包中的使用。

14.5. 定义组件

我们将从解释术语“组件”可能引起的混淆开始。考虑 `find_package()` 的完整签名：

```
find_package(<PackageName>
    [version] [EXACT] [QUIET] [MODULE] [REQUIRED]
    [[COMPONENTS] [components...]]
    [OPTIONAL_COMPONENTS components...]
    [NO_POLICY_SCOPE]
)
```

重要的是不要将这里提到的组件，与 `install()` 命令中使用的 `COMPONENT` 关键字混淆。尽管它们名称相同，但是不同的概念，必须分开理解。我们将在以下子节中进一步探讨这一点。

14.5.1. 如何在 `find_package()` 中使用组件

当调用带有 `COMPONENTS` 或 `OPTIONAL_COMPONENTS` 列表的 `find_package()` 时，告诉 CMake 我们只对提供这些组件的包感兴趣。然而，理解验证这一要求是包的责任至关重要。如果包提供商没有在配置文件中实现必要的检查，则不会按预期进行。

请求的组件通过 `<package>_FIND_COMPONENTS` 变量传递给配置文件（包括可选和不可选的）。对于每个不可选组件，都会设置一个 `<package>_FIND_REQUIRED_<component>` 变量。包作者可以编写宏来扫描这个列表，并验证所有必需组件的提供情况。`check_required_components()` 函数就为此目的服务。当找到必要的组件时，配置文件应设置 `<package>_<component>_FOUND` 变量。文件末尾的一个宏将验证是否设置了所有必需的变量。

14.5.2. 如何在 `install()` 命令中使用组件

并非在所有情况下都需要安装所有生成的工件。例如，一个项目可能为了开发而安装静态库和公共头文件，但默认情况下，可能只需要为运行时安装一个共享库。为了启用这种双重行为，可以使用 `COMPONENT` 关键字将工件分组在 `install()` 命令下的一个通用名称下。有兴趣限制安装到特定组件的用户可以通过执行以下区分大小写的命令来实现：

```
cmake --install <build tree>
--component=<component1 name> --component=<component2 name>
```

为工件分配 `COMPONENT` 关键字并不会自动将其从默认安装中排除。要实现这种排除，必须添加 `EXCLUDE_FROM_ALL` 关键字。

让我们在代码示例中探讨这个概念：

ch14/13-components/CMakeLists.txt (片段)

```
1 install(TARGETS calc EXPORT CalcTargets
2     ARCHIVE
3         COMPONENT lib
4         FILE_SET HEADERS
5             COMPONENT headers
6     )
7 install(EXPORT CalcTargets
8     DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake
9     NAMESPACE Calc::
10    COMPONENT lib
11 )
12 install(CODE "MESSAGE(\"Installing 'extra' component\")"
13     COMPONENT extra
14     EXCLUDE_FROM_ALL
15 )
```

前面的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`: 这包含 C++ 头文件。也默认安装。
- `extra`: 这执行一段代码以打印消息。不默认安装。

让我们重申：

- `cmake --install` 没有 `--component` 参数将安装 `lib` 和 `headers` 组件。
- `cmake --install --component headers` 将只安装公共头文件。
- `cmake --install --component extra` 将打印一个否则无法访问的消息 (`EXCLUDE_FROM_ALL` 关键字阻止了这一点)。

如果没有为安装的工件指定 `COMPONENT` 关键字，它默认为未指定，由 `CMAKE_INSTALL_DEFAULT_COMPONENT_NAME` 变量定义。

Note

由于无法从 `cmake` 命令行列出所有可用组件，因此彻底记录包组件对于用户来说可能非常有帮助。安装“`README`”文件是放置这些信息的绝佳位置。

如果 `cmake` 使用`--component` 参数调用一个不存在的组件，命令将成功完成，不会有警告或错误，但不会安装任何东西。

将我们的安装分区为组件，允许用户选择性地安装包的部分。现在来管理版本化共享库的符号链接，这是一个优化你的安装过程的有用功能。

14.5.3. 管理版本化共享库的符号链接

安装的目标平台可以使用符号链接，来帮助链接器发现共享库的当前安装版本。创建 `lib<name>` 后。所以符号链接到 `lib<name>`。

.so.1 文件是指共享库的一种符号链接形式，通常用于版本管理。可以通过向链接器传递 `-l<name>` 参数来链接这样的库。CMake 的 `install(TARGETS <target> LIBRARY)` 块处理了在必要时创建这类符号链接的工作。可以选择将这一步骤移到另一个 `install()` 命令中，通过在这个块中添加 `NAMELINK_SKIP` 来实现：

```
install(TARGETS <target> LIBRARY  
        COMPONENT cmp NAMELINK_SKIP)
```

如果要将符号链接分配给另一个组件而不是完全禁用，可以再次使用 `install()` 命令对同一个目标进行安装，并指定不同的组件，后面跟上 `NAMELINK_ONLY` 关键词：

```
install(TARGETS <target> LIBRARY  
        COMPONENT lnk NAMELINK_ONLY)
```

同样的效果也可以通过使用 `NAMELINK_COMPONENT` 关键词来达到：

```
install(TARGETS <target> LIBRARY  
        COMPONENT cmp NAMELINK_COMPONENT lnk)
```

现在我们已经配置了自动安装过程，可以使用随 CMake 一起提供的 CPack 工具来为用户提供预构建的工件。

14.6. 使用 CPack

虽然从源代码构建项目有其好处，但对于最终用户特别是非开发人员来说，这可能会既耗时又复杂。一种更方便的分发方法是使用二进制包，其中包含了编译后的制品和其他必要的静态文件。CMake 支持使用名为 `cpack` 的命令行工具来生成此类包。

要生成一个包，需要为目标平台和包类型选择一个合适的包生成器。不要将包生成器与像 Unix Makefiles 或 Visual Studio 这样的构建系统生成器混淆。

下表列出了可用的包生成器：

生成器名称	生成的文件类型	平台
Archive	7Z, 7zip - (.7z) TBZ2 (.tar.bz2) TGZ (.tar.gz) TXZ (.tar.xz) TZ (.tar.Z) TZST (.tar.zst) ZIP (.zip)	跨平台
Bundle	macOS Bundle (.bundle)	macOS
Cygwin	Cygwin packages	Cygwin
DEB	Debian packages (.deb)	Linux
External	第三方打包程序使用的 JSON (.json) 文件	跨平台
FreeBSD	PKG (.pkg)	*BSD, Linux, macOS
IFW	QT 安装程序二进制文件	Linux, Windows, macOS
NSIS	Binary (.exe)	Windows
NuGet	NuGet 包 (.nupkg)	Windows
productbuild	PKG (.pkg)	macOS
RPM	RPM (.rpm)	Linux
WIX	Microsoft Installer (.msi)	Windows

表 14.3: 可用的包生成器

大多数这些生成器都有广泛的配置选项。本书不打算深入探讨所有细节，相应信息可以在“扩展阅读”部分找到更多信息。

为了使用 CPack，需要使用必要的 `install()` 命令来配置项目的安装，并构建项目。CPack 会根据构建树中的 `CPackConfig.cmake` 文件准备二进制包。虽然可以手动创建这个文件，但在项目的列表文件中使用 `include(CPack)` 更加简便，它会在构建树中生成配置文件并提供所需的默认值。

让我们扩展 13-components 示例以供 CPack 使用：

ch14/14-cpack/CMakeLists.txt (fragment)

```

1 cmake_minimum_required(VERSION 3.26)
2 project(CPackPackage VERSION 1.2.3 LANGUAGES CXX)
3 include(GNUInstallDirs)
4 add_subdirectory(src bin)
5 install(...)
6 install(...)
7 install(...)
8 set(CPACK_PACKAGE_VENDOR "Rafal Swidzinski")
9 set(CPACK_PACKAGE_CONTACT "email@example.com")
10 set(CPACK_PACKAGE_DESCRIPTION "Simple Calculator")
11 include(CPack)

```

CPack 模块从 `project()` 命令中提取以下变量：

- `CPACK_PACKAGE_NAME`
- `CPACK_PACKAGE_VERSION`
- `CPACK_PACKAGE_FILE_NAME`

The `CPACK_PACKAGE_FILE_NAME` 存储了包名的结构：

```
$CPACK_PACKAGE_NAME-$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，在 `Debian` 上执行 ZIP 生成器时，CPack 将生成一个名为 `CPackPackage-1.2.3-Linux.zip` 的文件。

要在构建项目后生成包，请转到项目的构建树并运行：

```
cpack [<options>]
```

CPack 从 `CPackConfig.cmake` 文件读取选项，可以覆盖这些设置：

- `-G <generators>`: 以分号分隔的包生成器列表。默认值可以在 `CPackConfig.cmake` 中的 `CPACK_GENERATOR` 变量中指定。
- `-C <configs>`: 以分号分隔的构建配置列表 (`debug`, `release`)，用于生成包（对于多配置构建系统生成器是必需的）。
- `-D <var>=<value>`: 此选项覆盖 `CPackConfig.cmake` 文件中设置的变量。
- `--config <config-file>`: 此选项使用指定的配置文件代替默认的 `CPackConfig.cmake` 文件。`cmake`。
- `--verbose, -V`: 此选项提供详细的输出。
- `-P <packageName>`: 此选项覆盖包名。
- `-R <packageVersion>`: 此选项覆盖包版本。
- `--vendor <vendorName>`: 此选项覆盖包供应商。
- `-B <packageDirectory>`: 此选项指定 `cpack` 的输出目录（默认情况下，这将是当前工作目录）。

让我们尝试为我们的 `14-cpack` 示例项目生成包。我们将使用 ZIP、7Z 和 Debian 包生成器：

```
cpack -G "ZIP;7Z;DEB" -B packages
```

应该会得到以下这些包：

- `CPackPackage-1.2.3-Linux.7z`
- `CPackPackage-1.2.3-Linux.deb`
- `CPackPackage-1.2.3-Linux.zip`

这些二进制包已准备好发布在项目网站上、GitHub Release 页面中，或作为最终用户的包存储库。

14.7. 总结

编写跨平台安装脚本的复杂性可能会令人望而却步，但 CMake 显著简化了这一任务。尽管它需要一些初始设置，CMake 流程化了这一过程，与这本书中探讨的概念和技术无缝集成。

我们从理解如何从项目中导出 CMake 目标开始，使得它们可以在不需要安装的情况下在其他项目中使用。接下来，深入了解已经为导出配置的项目的安装。探讨安装基础时，专注于一个关键方面：安装 CMake 目标。现在掌握了 CMake 如何为不同工件类型分配不同目的地，以及公共头文件的特殊考虑。我们还检查了 `install()` 命令的其他模式，包括安装文件、程序和目录，以及在安装过程中执行脚本。

然后，CMake 可创建重用包。我们探索了如何使项目目标可重定位，从而方便用户定义安装位置。这包括创建可以通过 `find_package()` 使用的完全定义的包，涉及准备目标导出文件、配置文件和版本文件。考虑到不同用户的需求，了解了如何将工件和操作分组到安装组件中，将它们与 CMake 包的组件区分开来。我们的探索最终以 CPack 的介绍达到尾声，我们了解了如何准备基本的二进制包，提供了一种有效的方法来分发预编译的软件。虽然掌握 CMake 中安装和打包的细微之处是一个持续的过程，但这一章为我们奠定了坚实的基础。它使我们能够处理常见场景，并且自信地进一步深入。

下一章中，我们将运用我们积累的知识，通过创建一个连贯的、专业级别的项目，展示这些 CMake 技术的实际应用。

14.8. 扩展阅读

- GNU 目标编码标准

https://www.gnu.org/prep/standards/html_node/Directory-Variables.html

- 讨论使用 `FILE_SET` 的新关键字：

https://gitlab.kitware.com/cmake/cmake/-/issues/22468#note_991860

- 如何安装共享库：

<https://tldp.org/HOWTO/Program-Library-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()` 搜索配置文件时扫描的路径列表：

https://cmake.org/cmake/help/latest/command/find_package.html#configmode-search-procedure

- `CMakePackageConfigHelpers` 的完整文档：

<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>

- 关于不同平台首选包生成器的讨论：

<https://stackoverflow.com/a/46013099>

- CPack 实用模块文档：

<https://cmake.org/cmake/help/latest/module/CPack.html>

第 15 章 创建你的专业项目

我们已经汇集了构建专业项目所需的所有必要知识，包括结构化、构建、依赖管理、测试、分析、安装和打包等方面。现在，是时候运用这些技能来创建一个连贯且专业的项目了。重要的是要理解，即使是简单的程序也能从自动化的质量检查，将原始代码转变为完整解决方案的无缝流程中获益。确实，实施这些检查和流程是一项重大的投资，这需要许多步骤来正确地设置一切。当将这些机制添加到现有的代码库时尤其如此，这些代码库往往庞大且复杂。因此，从一开始就使用 CMake 并尽早建立所有必要的过程是非常有益的。这样配置起来更容易，也更高效，因为这些质量控制和构建自动化最终无论如何都需要集成到长期项目中。

本章中，将开发一个新的解决方案，尽可能地保持简单，同时充分利用本书迄今为止讨论过的 CMake 实践。为了简化问题，仅实现一个实用的功能——即两个数相加。这样的基础业务代码可使我们能够专注于前面章节中学到的与构建相关的项目方面。为了处理一个与构建更加相关的挑战性问题，此项目将包含一个库和一个可执行文件。

该库将处理内部业务逻辑，并作为 CMake 包供其他项目使用。而可执行文件，旨在供最终用户使用，将提供一个用户界面来展示该库的功能。

本章中，将包含以下内容：

- 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

15.1. 示例下载

可以在 GitHub 上找到本章中存在的代码文件，地址为 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch15>。

为了构建本书提供的示例，请使用推荐的命令：

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

请确保将占位符 `<build tree>` 和 `<source tree>` 替换为适当的路径。作为提醒：`<build tree>` 是指向目标/输出目录的路径，而 `<source tree>` 是源代码所在的位置的路径。

本章使用 GCC 编译，以提供与使用 lcov 工具收集结果的代码覆盖率仪器兼容性。如果想使用 llvm 或其他工具链进行编译，请确保根据需要调整覆盖率处理过程。

要运行测试，请执行以下命令：

```
ctest --test-dir <build tree>
```

或者，只需从 build tree 目录执行：

```
ctest
```

本章中，测试结果将输出到 test 子目录中。

15.2. 计划工作

本章将构建的软件并不复杂——将创建一个简单的计算器，可以实现两个数字的相加（图 15.1）。这是一个控制台应用程序，具有文本用户界面，利用第三方库和独立的计算库，这些库可以用于其他项目。尽管这个项目可能没有重要的实际应用，但其简单性非常适合演示本书讨论的各种技术应用。

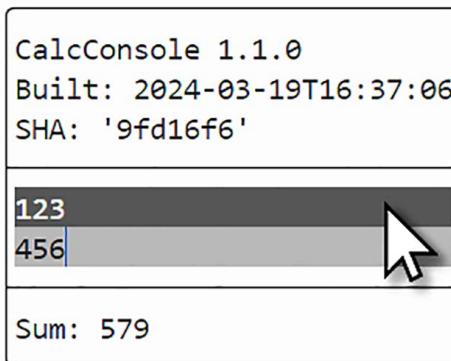


图 15.1：项目在支持鼠标的终端中执行的文本用户界面

通常，项目要么生成面向用户的可执行文件，要么为开发者生成库。项目同时产生这两者的情况较少，尽管这种情况确实存在。例如，一些应用程序附带了独立的 SDK 或库以帮助开发插件。另一个例子是附带了使用示例的库。我们的项目属于后者，展示了库的功能。

我们将通过回顾章节列表，回忆每个章节的内容，并选择将用来构建应用程序的技术和工具来开始规划：

- 第 1 章，CMake 入门：

本章提供了关于 CMake 的基本细节，包括安装和用于构建项目的命令行使用。还包含了关于项目文件的基本信息，如作用、典型的命名约定和特殊性。

- 第 2 章，CMake 语言：

我们介绍了编写正确的 CMake 列表文件和脚本所需的工具，包含了代码基础，如注释、命令调用和参数。我们解释了变量、列表和控制结构，引入了几个有用的命令。这个基础在我们的项目中至关重要。

- 第 3 章，在 IDE 中使用 CMake：

我们讨论了三个 IDE——CLion、VS Code 和 Visual Studio IDE，介绍了它们的优点。在最终项目中，选择 IDE（或不选择）由你决定。做出决定后，可以在该项目中使用 Dev 容器开始，只需几个步骤就可以构建 Docker 镜像（或者直接从 Docker Hub 获取）。在容器中运行镜像可以确保开发环境与生产环境相匹配。

- 第 4 章，设置 CMake 项目：

配置项目至关重要，其决定了将生效的 CMake 策略、命名、版本控制和编程语言。我们将使用本章来影响构建过程的基本行为。

我们还将遵循建立的项目分区和结构来确定目录和文件的布局，并利用系统发现变量以适应不同的构建环境。工具链配置是另一个关键方面，可以强制使用特定的 C++ 版本和编译器支持的标准。按照章节的建议，我们将禁用源内构建以保持工作区清洁。

- 第 5 章，使用目标：

了解到每个现代 CMake 项目都广泛使用目标，当然也会应用目标来定义一些库和可执行文件（用于测试和生产），这将使项目保持组织并确保我们遵循 DRY（不要重复自己）的原则。对目标属性和传递使用要求（传播属性）的了解，将能够使配置接近目标定义。

- 第 6 章，使用生成器表达式：

生成器表达式在项目中大量使用，力求使这些表达式尽可能简单。项目将包含自定义命令以生成 Valgrind 和覆盖率报告的文件。此外，还将使用目标钩子，特别是 PRE_BUILD，来清理覆盖率检测过程产生的.gcda 文件。

- 第 7 章，使用 CMake 编译 C++ 源代码：

没有 C++ 项目的编译是不可能的。基础知识相当简单，但 CMake 允许我们以许多方式调整这个过程：扩展目标源代码、配置优化器并提供调试信息。对于这个项目，默认的编译标志就可以了，也研究了一下预处理器：

- 我们将构建元数据（项目版本、构建时间和 Git 提交 SHA）存储在编译后的可执行文件中并向用户展示。
- 我们将启用预编译头文件。在如此小的项目中，这并不是真正必要的，但它将帮助我们练习这个概念。

不需要 Unity 构建。

- 第 8 章，链接可执行文件和库：

我们将获得默认情况下对项目都有用的链接的一般信息。此外，由于这个项目包含一个库，将明确引用以下特定构建指令：

- 用于测试和开发的静态库
- 用于发布的共享库

本章还概述了如何隔离 main() 函数以用于测试目的，我们将采用这种做法。

- 第 9 章，管理依赖关系：

为了增强项目的吸引力，将引入一个外部依赖：一个基于文本的用户界面库。第 9 章探讨了管理依赖关系的各种方法。选择将很简单：FetchContent 实用模块通常推荐且最方便。

- 第 10 章，使用 C++20 模块：

尽管我们已经探讨了使用 C++20 模块，以及支持此功能的环境要求（CMake 3.28，最新编译器），但其广泛支持仍然不足。为了确保项目的可访问性，我们暂时不会引入模块。

- 第 11 章：测试框架

实施适当的自动化测试，对于确保解决方案质量随时间保持一致至关重要。我们将集成 CTest 并组织项目以方便测试，并应用之前提到的 main() 函数分离方法。

本章将讨论两种测试框架：Catch2 和 GTest 与 GMock；我们将使用后者。为了获取覆

覆盖率的详细信息，我们将使用 LCOV 生成 HTML 报告。

- 第 12 章：程序分析工具

对于静态分析，可以从一系列工具中选择：Clang-Tidy、Cpplint、Cppcheck、include-what-you-use (IWWU) 和 link-what-you-use (LWWU)。我们将选择 Cppcheck，因为 Clang-Tidy 与使用 GCC 构建的预编译头文件兼容性较差。

动态分析将使用 Valgrind 的 Memcheck 工具，并配合 Memcheck-cover 包装器来生成 HTML 报告。此外，在构建过程中，源码将自动通过 ClangFormat 进行格式化。

- 第 13 章：文档生成

提供文档对于我们项目中的库来说是必不可少的。CMake 支持使用 Doxygen 自动化生成文档。我们将采用这种方法，并在设计中加入 doxygen-awesome-css 主题以更新样式。

- 第 14 章：安装与打包

最后，将配置解决方案的安装和打包，并准备文件形成包，包括目标定义。安装这些内容及构建目标产生的工件到合适的目录中，通过包含 GNUInstallDirs 模块实现。还将配置一些组件以模块化解决方案，并为 CPack 做好准备。

专业的项目通常会附带一些文本文件：README、LICENSE、INSTALL 等。我们将在章节末尾简要介绍这些文件。

为了简化流程，不会实现自定义逻辑来检查所有必需的工具和依赖项是否可用。我们将依赖于 CMake 来显示其诊断信息并告诉用户缺少什么。如果项目获得了重要的关注，可能需要考虑添加这些机制以改善用户体验。

有了清晰的计划后，让我们讨论如何实际地构建项目结构，包括逻辑目标和目录结构。

15.3. 项目布局

为了构建任何项目，应该首先明确项目内部将创建哪些逻辑目标。这个案例中，将遵循下图所示的结构：

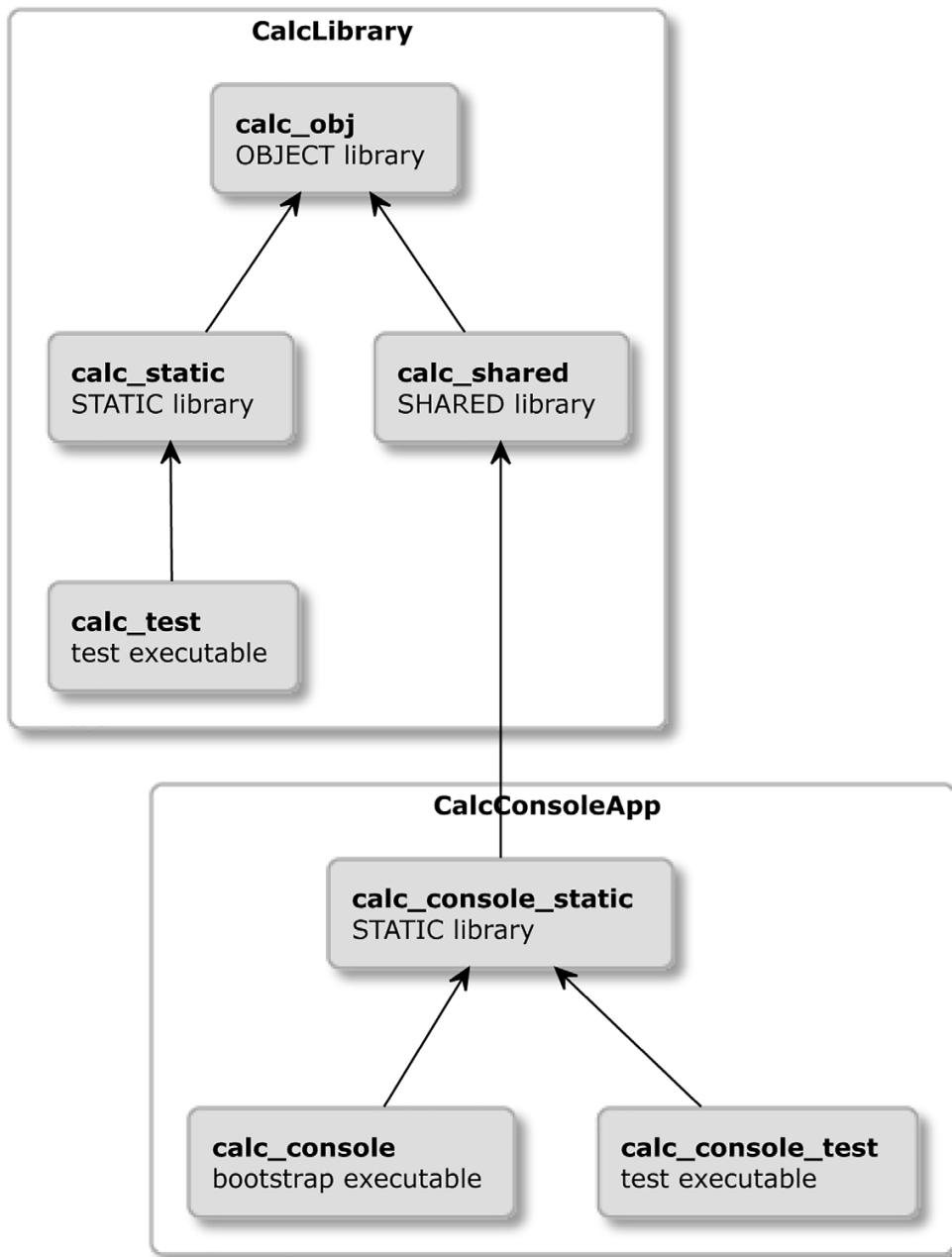


图 15.2：逻辑目标的结构

按照构建顺序来探索这个结构。首先，编译 `calc_obj`，一个对象库。然后，将注意力转向静态库和共享库。

15.3.1. 共享库与静态库

第 8 章中，介绍了共享库和静态库。当多个程序使用相同的库时，共享库可以减少整体内存使用量，用户通常已经安装了流行的库或者知道如何快速安装它们。

更重要的是，共享库是独立的文件，必须放置在特定路径以便动态链接器能够找到它们。相比之下，静态库直接嵌入到可执行文件中，这使得使用更快，不需要额外步骤在内存中定位代码。

作为库的作者，可以决定提供静态库，还是共享库，也可以同时发布两个版本，将这个决定留给使用库的开发者。既然在积累知识，就来提供两个版本的库。

`calc_test` 目标包含了单元测试以验证库的核心功能，它将使用静态库。虽然是从相同的对象文件构建两种类型的库，但无论使用哪种类型的库进行测试都可以接受，它们的功能相同。与 `calc_console_static` 目标相关的控制台应用程序将使用共享库。此目标还链接了一个外部依赖，即 Arthur Sonzogni 的 Functional Terminal (X) User Interface (FTXUI) 库（在扩展阅读部分中有 GitHub 项目的链接）。

最后两个目标，`calc_console` 和 `calc_console_test`，旨在解决测试可执行文件中的常见问题：测试框架和可执行文件提供的多个入口点之间的冲突。为此，可将 `main()` 函数隔离到一个引导目标 `calc_console` 中，该目标仅仅调用 `calc_console_static` 中的主要函数。

理解了必要的目标及其相互关系之后，下一步是通过适当的文件和目录来组织项目的结构。

15.3.2. 项目文件结构

项目由两个关键元素组成：`calc` 库和 `calc_console` 可执行文件。为了有效地组织项目，将采用以下目录结构：

- `src` 包含所有发布的目标的源代码和库头文件。
- `test` 包含上述库和可执行文件的测试。
- `cmake` 包含用于构建和安装项目的 CMake 辅助模块和文件。
- 根目录包含顶级配置和文档文件。

这一结构（如图 15.3 所示）确保了职责的清晰划分，便于更轻松地导航和维护项目：

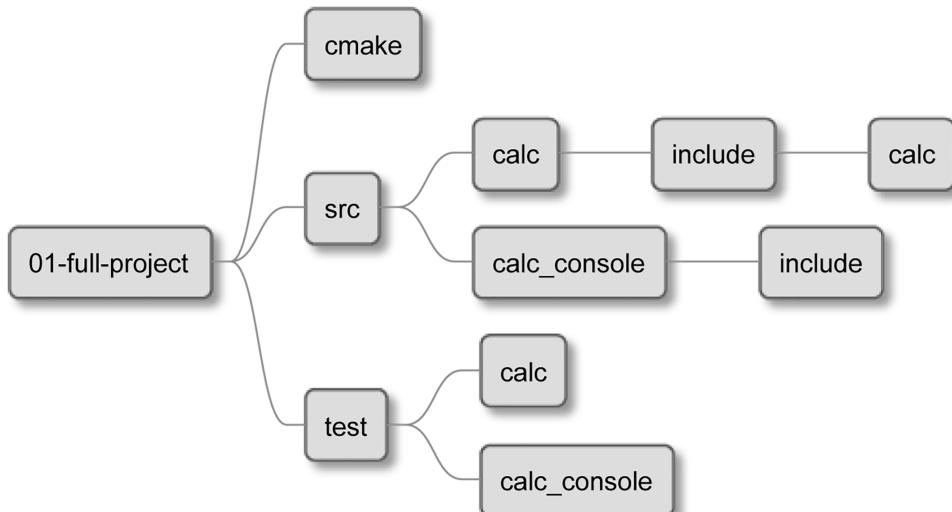


图 15.3：项目的目录结构

下面是每个主要目录中的文件完整列表：

根目录	<code>./test</code>
CHANGELOG	CMakeLists.txt

CMakeLists.txt	calc/CMakeLists.txt
INSTALL	calc/calc_test.cpp
LICENSER	calc_console/CMakeLists.txt
EADME.md	calc_console/tui_test.cpp
./src	./cmake
CMakeLists.txt	BuildInfo.cmake
calc/CMakeLists.txt	Coverage.cmake
calc/CalcConfig.cmake	CppCheck.cmake
calc/basic.cpp	Doxygen.cmake
calc/include/calc/basic.h	Format.cmake
calc_console/CMakeLists.txt	GetFTXUI.cmake
calc_console/bootstrap.cpp	Packaging.cmake
calc_console/include/tui.h	Memcheck.cmake
calc_console/tui.cpp	NoInSourceBuilds.cmake
	Testing.cmake
	buildinfo.h.in
	doxygen_extra_headers

表 15.1: 项目的文件结构

看起来 CMake 引入了相当大的开销，初始阶段 `cmake` 目录包含的内容比实际业务代码还要多，但随着项目的扩展，这种状况将会改变。建立干净且有组织的项目结构需要较大的初始努力，但请放心，这种投资在未来将带来显著的好处。

我们将在整个章节中详细介绍表 15.1 中列出的所有文件，以及其作用和在项目中的角色。这将分为四个步骤：构建、测试、安装和提供文档。

15.4. 构建和管理依赖关系

所有的构建过程都遵循同样的程序。我们从顶层列表文件开始，并向下遍历项目的源代码树。图 15.4 展示了构建过程中涉及的项目文件，括号内的数字表示 CMake 脚本执行的顺序。

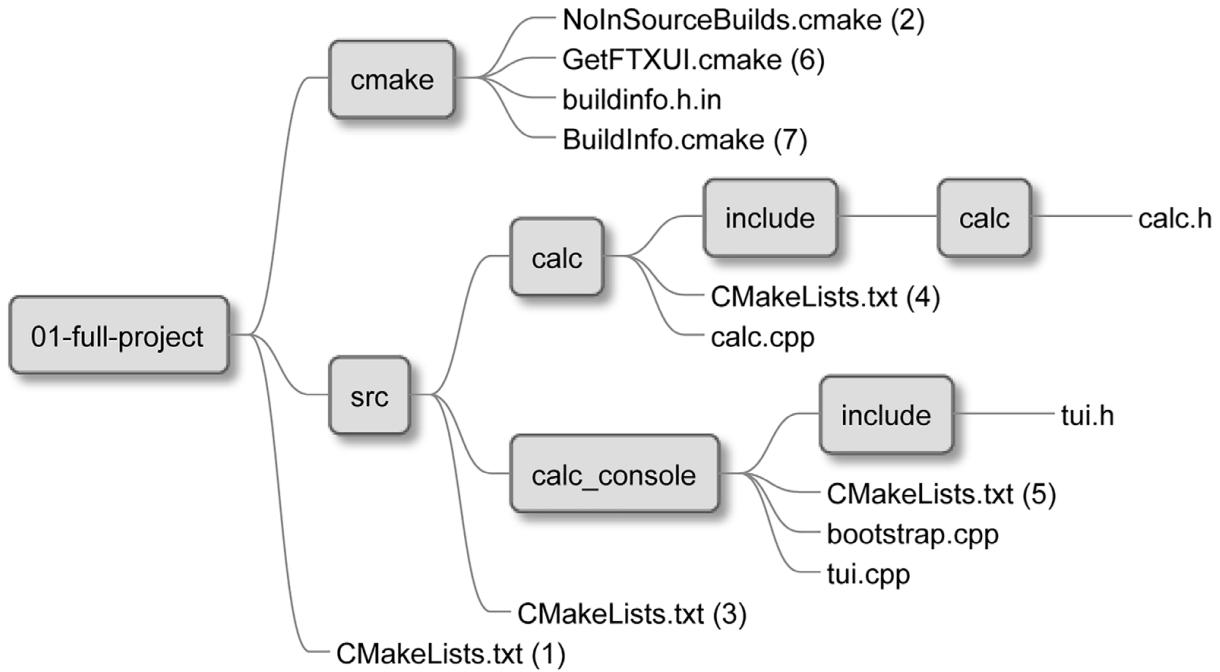


图 15.4：构建阶段使用的文件

顶层的 CMakeLists.txt (1) 配置项目：

ch15/01-full-project/CMakeLists.txt

```

1  cmake_minimum_required(VERSION 3.26)
2  project(Calc VERSION 1.1.0 LANGUAGES CXX)
3  list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

4
5  include(NoInSourceBuilds)
6
7  include(CTest)
8
9  add_subdirectory(src bin)
10 add_subdirectory(test)

11
12 include(Packaging)

```

首先指定基本的项目详情并设置 CMake 实用模块的路径（项目中的 `cmake` 目录），接着使用一个自定义模块来防止原地构建。

随后，使用内置的 `CTest` 模块启用测试。这应该在项目的根级别完成，这条命令会在二进制树中创建 `CTestTestfile.cmake` 文件，相对于其在源码树的位置。如果放在其他地方，则 `ctest` 将无法找到它。

接下来，我们包含两个关键目录：

- `src`，包含项目源代码（在构建树中命名为 `bin`）
- `test`，包含所有测试工具

最后，包含 `Packaging` 模块，相关细节将在安装和打包部分讨论。
让我们来查看 `NoInSourceBuilds` 实用模块，以理解其功能：

ch15/01-full-project/cmake/NoInSourceBuilds.cmake

```
1 if(PROPERTY_SOURCE_DIR STREQUAL PROPERTY_BINARY_DIR)
2     message(FATAL_ERROR
3         "\n"
4         "In-source builds are not allowed.\n"
5         "Instead, provide a path to build tree like so:\n"
6         "cmake -B <destination>\n"
7         "\n"
8         "To remove files you accidentally created execute:\n"
9         "rm -rf CMakeFiles CMakeCache.txt\n"
10    )
11 endif()
```

这里没有什么意外，检查用户是否使用 `cmake` 命令提供了单独的目的地目录来存放生成的文件。它必须与项目的源码树路径不同。如果没有，会指导用户如何指定目的地，并如何清理仓库中的错误文件。

顶层列表文件接着包含了 `src` 子目录，指示 CMake 处理其中的列表文件：

ch15/01-full-project/src/CMakeLists.txt

```
1 include(Coverage)
2 include(Format)
3 include(CppCheck)
4 include(Doxygen)
5
6 add_subdirectory(calc)
7 add_subdirectory(calc_console)
```

这个文件很简单——包含了来自`./cmake` 目录的所有模块，并指示 CMake 执行这些目录中的列表文件。

接下来，来看一下 `calc` 库的列表文件。这个文件有些复杂，我们将逐段解析并讨论。

15.4.1. 构建 Calc 库

`calc` 目录中的列表文件配置了这个库的各个方面，但现在只关注构建部分：

ch15/01-full-project/src/calc/CMakeLists.txt (片段)

```
1 add_library(calc_obj OBJECT basic.cpp)
2 target_sources(calc_obj
3                 PUBLIC FILE_SET HEADERS
4                 BASE_DIRS include)
```

```

5      FILES include/calc/basic.h
6  )
7  set_target_properties(calc_obj PROPERTIES
8      POSITION_INDEPENDENT_CODE 1
9  )
10
11 # ... instrumentation of calc_obj for coverage
12
13 add_library(calc_shared SHARED)
14 target_link_libraries(calc_shared calc_obj)
15
16 add_library(calc_static STATIC)
17 target_link_libraries(calc_static calc_obj)
18
19 # ... testing and program analysis modules
20 # ... documentation generation
21 # ... installation

```

我们定义了 3 个目标：

- calc_obj，一个对象库，编译 basic.cpp 实现文件，basic.h 头文件通过 target_sources() 命令中的 FILE_SET 关键字包含进来。这隐式地配置了适当的包含目录，以便在构建和安装模式下正确导出。通过创建一个对象库，避免了为两个库版本重复编译，但重要的是要启用 POSITION_INDEPENDENT_CODE，以便共享库可以依赖这个目标。
- calc_shared，依赖于 calc_obj 的共享库。
- calc_static，依赖于 calc_obj 的静态库。

作为上下文，这里是 basic 库的 C++ 头文件。这个头文件简单地声明了 Calc 命名空间中的两个函数，避免名称冲突：

ch15/01-full-project/src/calc/include/calc/basic.h

```

1 #pragma once
2 namespace Calc {
3     int Add(int a, int b);
4     int Subtract(int a, int b);
5 } // namespace Calc

```

实现文件也很简单：

ch15/01-full-project/src/calc/basic.cpp

```

1 namespace Calc {
2     int Add(int a, int b) {
3         return a + b;
4     }

```

```
5
6     int Subtract(int a, int b) {
7         return a - b;
8     }
9 } // namespace Calc
```

这就完成了对 `src/calc` 目录中文件的解释，接下来是 `src/calc_console` 目录，以及使用这个库构建控制台计算器的可执行文件。

15.4.2. 构建 Calc 可执行文件

`calc_console` 目录包含几个文件：一个列表文件、两个实现文件（业务逻辑和引导文件），以及一个头文件。列表文件如下：

`ch15/01-full-project/src/calc_console/CMakeLists.txt (fragment)`

```
1 add_library(calc_console_static STATIC tui.cpp)
2 target_include_directories(calc_console_static PUBLIC include)
3 target_precompile_headers(calc_console_static PUBLIC <string>)
4
5 include(GetFTXUI)
6 target_link_libraries(calc_console_static PUBLIC calc_shared
7                         ftxui::screen ftxui::dom ftxui::component)
8
9 include(BuildInfo)
10 BuildInfo(calc_console_static)
11
12 # ... instrumentation of calc_console_static for coverage
13 # ... testing and program analysis modules
14 # ... documentation generation
15
16 add_executable(calc_console bootstrap.cpp)
17 target_link_libraries(calc_console calc_console_static)
18
19 # ... installation
```

尽管列表文件看起来很复杂，但作为经验丰富的 CMake 用户，现在很容易解读其内容：

1. 定义 `calc_console_static` 目标，包含不含 `main()` 函数的业务代码，以允许与具有自己入口点的 GTest 链接。
2. 配置包含目录。可以通过 `FILE_SET` 逐个添加头文件，但由于其是内部的，这里简化了这一步骤。
3. 实现头文件预编译，这里以 `<string>` 头文件为例演示，虽然大型项目可能会包含更多的头文件。
4. 包含一个自定义的 CMake 模块来获取 FTXUI 依赖项。

5. 将业务代码与 `calc_shared` 共享库和 FTXUI 组件链接起来。
6. 添加一个自定义模块来生成构建信息，并将其嵌入到工件中。
7. 概述了针对此目标的其他步骤：`coverage` 配置、测试、程序分析和文档生成。
8. 创建并链接 `calc_console` 引导可执行文件，建立入口点。
9. 概述安装过程。

我们将在本章后续的部分中探讨测试、文档和安装过程。

大多数用户不太可能已经安装了它，所以包含了 `GetFTXUI` 实用模块而不是寻找系统中的配置模块。我们只是获取并构建：

ch15/01-full-project/cmake/GetFTXUI.cmake

```

1 include(FetchContent)
2 FetchContent_Declare(
3     FTXTUI
4     GIT_REPOSITORY https://github.com/ArthurSonzogni/FTXUI.git
5     GIT_TAG v0.11
6 )
7 option(FTXUI_ENABLE_INSTALL "" OFF)
8 option(FTXUI_BUILD_EXAMPLES "" OFF)
9 option(FTXUI_BUILD_DOCS "" OFF)
10 FetchContent_MakeAvailable(FTXTUI)

```

我们正在使用推荐的 `FetchContent` 方法，唯一不寻常的添加是 `option()` 命令的调用，这让我们可以绕过 FTXUI 的冗长构建步骤，并阻止其安装步骤影响本项目的安装过程。更多详情，请参考扩展阅读部分。

`calc_console` 目录的列表文件还包括另一个与构建相关的自定义实用模块：`BuildInfo`。这个模块将捕获三块信息以在可执行文件中显示：

- 当前 Git 提交的 SHA
- 构建时间戳
- 在顶层列表文件中指定的项目版本

正如我们在第 7 章中介绍的，CMake 可以捕获构建时的值，并通过模板文件传递给 C++ 代码，例如通过结构体：

ch15/01-full-project/cmake/buildinfo.h.in

```

1 struct BuildInfo {
2     static inline const std::string CommitSHA = "@COMMIT_SHA@";
3     static inline const std::string Timestamp = "@TIMESTAMP@";
4     static inline const std::string Version = "@PROJECT_VERSION@";
5 };

```

为了在配置阶段填充这个结构体，将使用以下代码：

ch15/01-full-project/cmake/BuildInfo.cmake

```

1 set(BUILDINFO_TEMPLATE_DIR ${CMAKE_CURRENT_LIST_DIR})
2 set(DESTINATION "${CMAKE_CURRENT_BINARY_DIR}/buildinfo")
3 string(TIMESTAMP TIMESTAMP)
4
5 find_program(GIT_PATH git REQUIRED)
6 execute_process(COMMAND ${GIT_PATH} log --pretty=format:'%h' -n 1
7                         OUTPUT_VARIABLE COMMIT_SHA)
8
9 configure_file(
10     "${BUILDINFO_TEMPLATE_DIR}/buildinfo.h.in"
11     "${DESTINATION}/buildinfo.h" @ONLY
12 )
13
14 function(BuildInfo target)
15     target_include_directories(${target} PRIVATE ${DESTINATION})
16 endfunction()

```

包含该模块后，设置了变量来捕获所需的信息，并使用 `configure_file()` 来生成 `buildinfo.h`。最后一步是调用 `BuildInfo` 函数，将生成文件的目录加入到目标的包含目录中。

产生的头文件，可以根据需要与多个不同的消费者共享。这种情况下，可能想要在列表文件的顶部添加 `include_guard(GLOBAL)` 来避免为每个目标运行 `git` 命令。

深入了解控制台计算器的实现之前，我想强调不需要深入理解 `tui.cpp` 文件或 `FTXUI` 库的复杂性，这对于我们的目的来说不是必需的。

ch15/01-full-project/src/calc_console/tui.cpp

```

1 #include "tui.h"
2 #include <ftxui/dom/elements.hpp>
3 #include "buildinfo.h"
4 #include "calc/basic.h"
5
6 using namespace ftxui;
7 using namespace std;
8
9 string a{"12"}, b{"90"};
10 auto input_a = Input(&a, "");
11 auto input_b = Input(&b, "");
12 auto component = Container::Vertical({input_a, input_b});
13
14 Component getTui() {
15     return Renderer(component, [&] {
16         auto sum = Calc::Add(stoi(a), stoi(b));
17         return vbox({
18             text("CalcConsole " + BuildInfo::Version),

```

```

19     text("Built: " + BuildInfo::Timestamp),
20     text("SHA: " + BuildInfo::CommitSHA),
21     separator(),
22     input_a->Render(),
23     input_b->Render(),
24     separator(),
25     text("Sum: " + to_string(sum)),
26   }) |
27   border;
28 });
29 }

```

这段代码提供了 `getTui()` 函数，该函数返回一个 `ftxui::Component` 对象，这是一个封装了交互式 UI 元素的对象，如标签、文本字段、分割线和边框。

更重要的是，包含指令链接到了 `calc_obj` 目标的头文件和 `BuildInfo` 模块。交互从 `lambda` 函数开始，调用 `Calc::Add`，并使用 `text()` 函数显示结果。

在构建时收集的 `buildinfo.h` 中的值以类似的方式使用，并将在运行时显示给用户。

除了 `tui.cpp` 外，还有一个头文件：

ch15/01-full-project/src/calc_console/include/tui.h

```

1 #include <ftxui/component/component.hpp>
2 ftxui::Component getTui();

```

这个头文件在 `calc_console` 目标中的引导文件中使用：

ch15/01-full-project/src/calc_console/bootstrap.cpp

```

1 #include <ftxui/component/screen_interactive.hpp>
2
3 #include "tui.h"
4
5 int main(int argc, char** argv) {
6     ftxui::ScreenInteractive::FitComponent().Loop(getTui());
7 }

```

这段简短的代码使用 FTXUI 初始化了一个交互式的控制台屏幕，显示 `getTui()` 返回的 `Component` 对象，并在一个循环中处理键盘输入。现在所有 `src` 目录中的文件都已经介绍过了，我们可以继续进行测试和程序分析。

15.5. 测试和程序分析

程序分析和测试是确保解决方案质量的重要组成部分。例如，在运行测试代码时使用 `Valgrind` 会更加有效（一致性和覆盖范围）。因此，可同一个地方配置测试和程序分析。图 15.5 展示了设置它们所需的执行流程和文件：

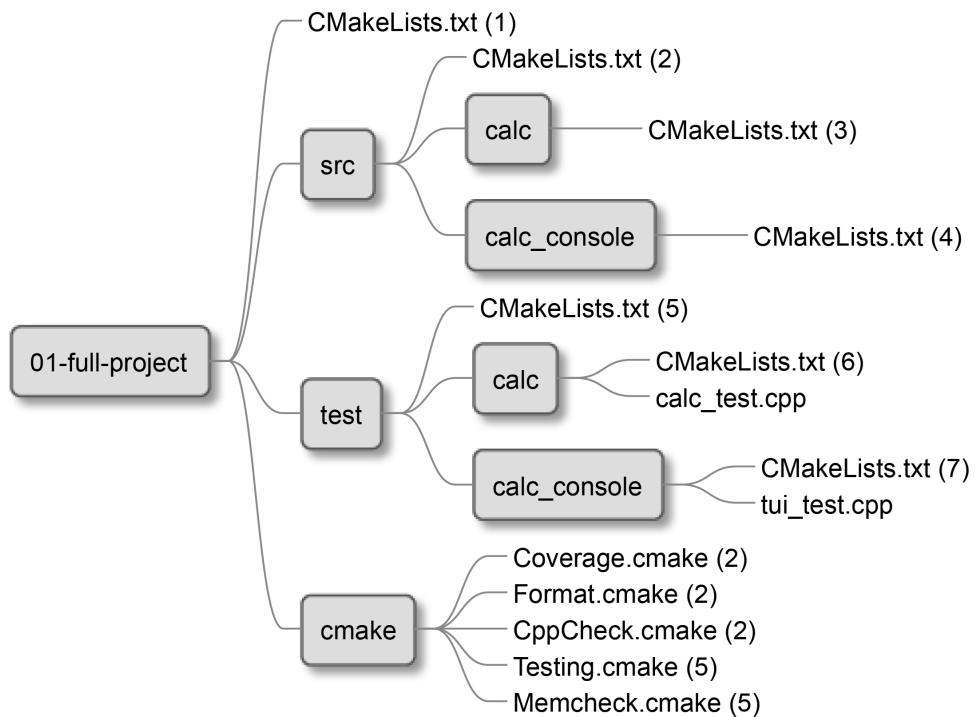


图 15.5：用于启用测试和程序分析的文件

括号中的数字代表列表文件的处理顺序。从顶层的列表文件开始，添加 `src` 和 `test` 目录：

- `src` 目录中，包含 `Coverage`、`Format` 和 `CppCheck` 模块，并添加 `src/calc` 和 `src/calc_console` 目录。
- `src/calc` 目录中，定义目标并使用包含的模块进行配置。
- `src/calc_console` 目录中，定义目标并使用包含的模块进行配置。
- `test` 目录中，包含 `Testing`（其中包含了 `Memcheck`）并添加 `test/calc` 和 `test/calc_console` 目录。
- `test/calc` 目录中，定义测试目标并使用包含的模块进行配置。
- `test/calc_console` 目录中，定义测试目标并使用包含的模块进行配置。

检查 `test` 目录下的列表文件：

`ch15/01-full-project/test/CMakeLists.txt`

```

1 include(Testing)
2 add_subdirectory(calc)
3 add_subdirectory(calc_console)
  
```

这个级别上，包含 `Testing` 实用程序模块以提供来自 `calc` 和 `calc_console` 目录的目标组的功能：

`ch15/01-full-project/cmake/Testing.cmake (fragment)`

```

1 include(FetchContent)
2 FetchContent_Declare(
  
```

```

3     googletest
4     GIT_REPOSITORY https://github.com/google/googletest.git
5     GIT_TAG v1.14.0
6 )
7 # For Windows: Prevent overriding the parent project's
8 # compiler/linker settings
9 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
10 option(INSTALL_GMOCK "Install GMock" OFF)
11 option(INSTALL_GTEST "Install GTest" OFF)
12 FetchContent_MakeAvailable(googletest)
13
14 # ...

```

我们启用了测试，并包含了 FetchContent 模块以获取 GTest 和 GMock。虽然 GMock 在这个项目中没有使用，但它与 GTest 在同一仓库中，所以使用同样的配置。关键的配置步骤是通过使用 option() 命令防止这些框架的安装，影响到我们的项目的安装。

同一文件中，我们定义了一个 AddTests() 函数来方便地全面测试业务目标：

ch15/01-full-project/cmake/Testing.cmake (continued)

```

1 # ...
2 include(GoogleTest)
3 include(Coverage)
4 include(Memcheck)
5 macro(AddTests target)
6   message("Adding tests to ${target}")
7   target_link_libraries(${target} PRIVATE gtest_main gmock)
8   gtest_discover_tests(${target})
9   AddCoverage(${target})
10  AddMemcheck(${target})
11 endmacro()

```

首先，包含了必要的模块：GoogleTest 与 CMake 捆绑在一起，而 Coverage 和 Memcheck 是项目中包含的自定义实用程序模块。然后提供了 AddTests 宏来准备目标进行测试、应用覆盖率度量，以及内存检查，AddCoverage() 和 AddMemcheck() 函数分别定义在各自的程序模块中。现在，可以继续实现。

15.5.1. 准备 Coverage 模块

各种目标上添加覆盖率涉及几个步骤，Coverage 模块提供了一个函数来为指定的目标定义覆盖率目标：

ch15/01-full-project/cmake/Coverage.cmake (片段)

```

1 function(AddCoverage target)
2   find_program(LCOV_PATH lcov REQUIRED)

```

```

3   find_program(GENHTML_PATH genhtml REQUIRED)
4   add_custom_target(coverage-${target}
5     COMMAND ${LCOV_PATH} -d . --zerocounters
6     COMMAND ${TARGET_FILE:${target}}
7     COMMAND ${LCOV_PATH} -d . --capture -o coverage.info
8     COMMAND ${LCOV_PATH} -r coverage.info '/usr/include/*'
9     -o filtered.info
10    COMMAND ${GENHTML_PATH} -o coverage-${target}
11      filtered.info --legend
12    COMMAND rm -rf coverage.info filtered.info
13    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}
14  )
15 endfunction()
16
17 # ...

```

此实现与第 11 章中介绍的略有不同，它现在包括目标名称作为输出路径的一部分，以避免名称冲突。接下来，需要一个函数来清除之前的覆盖率结果：

ch15/01-full-project/cmake/Coverage.cmake (续)

```

1 # ...
2
3 function(CleanCoverage target)
4   add_custom_command(TARGET ${target} PRE_BUILD COMMAND
5     find ${CMAKE_BINARY_DIR} -type f
6     -name '*.gcda' -exec rm {} +
7 endfunction()
8
9 # ...

```

此外，我们还有一个函数来准备目标以进行覆盖率分析：

ch15/01-full-project/cmake/Coverage.cmake (片段)

```

1 # ...
2 function(InstrumentForCoverage target)
3   if (CMAKE_BUILD_TYPE STREQUAL Debug)
4     target_compile_options(${target}
5       PRIVATE --coverage -fno-inline)
6     target_link_options(${target} PUBLIC --coverage)
7   endif()
8 endfunction()

```

`InstrumentForCoverage()` 函数应用于 `src/calc` 和 `src/calc_console`，使目标 `calc_obj` 和 `calc_console_static` 在执行时能够生成覆盖率数据文件。

为了生成两个测试目标的报告，在使用 Debug 构建类型配置项目后，执行以下 cmake 命令：

```
cmake --build <build-tree> -t coverage-calc_test  
cmake --build <build-tree> -t coverage-calc_console_test
```

接下来，我们希望对我们定义的多个目标进行动态程序分析，所以为了应用在第 12 章中介绍的 Memcheck 模块，需要稍微调整它以扫描多个目标。

15.5.2. 准备 Memcheck 模块

由 AddTests() 发起的 Valgrind 内存管理报告的生成从 Memcheck 模块开始，从初始设置开始：

ch15/01-full-project/cmake/Memcheck.cmake (片段)

```
1 include(FetchContent)  
2 FetchContent_Declare(  
3     memcheck-cover  
4     GIT_REPOSITORY https://github.com/Farigh/memcheck-cover.git  
5     GIT_TAG release-1.2  
6 )  
7 FetchContent_MakeAvailable(memcheck-cover)
```

这部分代码我们已经很熟悉了，来看看创建生成报告所需目标的函数：

ch15/01-full-project/cmake/Memcheck.cmake (续)

```
1 function(AddMemcheck target)  
2     set(MEMCHECK_PATH ${memcheck-cover_SOURCE_DIR}/bin)  
3     set(REPORT_PATH "${CMAKE_BINARY_DIR}/valgrind-${target}")  
4     add_custom_target(memcheck-${target}  
5         COMMAND ${MEMCHECK_PATH}/memcheck_runner.sh -o  
6             "${REPORT_PATH}/report"  
7             -- ${TARGET_FILE:${target}}  
8         COMMAND ${MEMCHECK_PATH}/generate_html_report.sh  
9             -i ${REPORT_PATH}  
10            -o ${REPORT_PATH}  
11            WORKING_DIRECTORY ${CMAKE_BINARY_DIR}  
12        )  
13    endfunction()
```

我们稍微改进了第 12 章中的 AddMemcheck() 函数以处理多个目标，让 REPORT_PATH 变量成为目标特定的。

为了生成 Memcheck 报告，请使用以下命令（使用 Debug 构建类型进行配置时，生成报告更为有效）：

```
cmake --build <build-tree> -t memcheck-calc_test  
cmake --build <build-tree> -t memcheck-calc_console_test
```

好的，我们定义了 Coverage 和 Memcheck 模块（在 Testing 模块中使用），现在让我们看看如何配置实际的测试目标。

15.5.3. 应用测试场景

为了实施测试，我们将遵循以下场景：

1. 编写单元测试。
2. 使用 AddTests() 定义和配置测试的可执行目标。
3. 为被测软件（Software Under Test, SUT）配置度量以启用覆盖率收集。
4. 确保在构建之间清除覆盖率数据以防止分段错误。

让我们从要编写的单元测试开始。为了保持简洁，我们将提供最简单的（也许有点不完整）单元测试。首先，测试库：

ch15/01-full-project/test/calc/basic_test.cpp

```
1 #include "calc/basic.h"  
2 #include <gtest/gtest.h>  
3  
4 TEST(CalcTest, SumAddsTwoInts) {  
    EXPECT_EQ(4, Calc::Add(2, 2));  
}  
5  
6  
7  
8 TEST(CalcTest, SubtractsTwoInts) {  
    EXPECT_EQ(6, Calc::Subtract(8, 2));  
}
```

接着是针对控制台的测试——为此，将使用 FXTUI 库。再次，完全理解源代码并非必要；这些测试仅用于说明目的：

ch15/01-full-project/test/calc_console/tui_test.cpp

```
1 #include "tui.h"  
2  
3 #include <gmock/gmock.h>  
4 #include <gtest/gtest.h>  
5  
6 #include <ftxui/screen/screen.hpp>  
7  
8 using namespace ::ftxui;  
9  
10 TEST(ConsoleCalcTest, RunWorksWithDefaultValues) {  
    auto component = getTui();
```

```

12     auto document = component->Render();
13     auto screen = Screen::Create(Dimension::Fit(document));
14     Render(screen, document);
15     auto output = screen.ToString();
16     ASSERT_EQ(output, testing::HasSubstr("Sum: 102"));
17 }
```

此测试将 UI 渲染到静态 Screen 对象，并检查字符串输出是否包含预期的总和。这不是一个很好的测试，但至少它很短。

现在，使用两个嵌套的列表文件来配置我们的测试，为生成库：

ch15/01-full-project/test/calc/CMakeLists.txt

```

1 add_executable(calc_test basic_test.cpp)
2 target_link_libraries(calc_test PRIVATE calc_static)
3 AddTests(calc_test)
```

为生成可执行文件：

ch15/01-full-project/test/calc_console/CMakeLists.txt

```

1 add_executable(calc_console_test tui_test.cpp)
2 target_link_libraries(calc_console_test
3                         PRIVATE calc_console_static)
4 AddTests(calc_console_test)
```

这些配置使得 CTest 可以执行测试。还需要为业务逻辑目标准备覆盖率分析，并确保覆盖率数据在构建之间得到刷新。

向 calc 库目标添加必要的指令：

ch15/01-full-project/src/calc/CMakeLists.txt (续)

```

1 # ... calc_obj target definition
2
3 InstrumentForCoverage(calc_obj)
4
5 # ... calc_shared target definition
6 # ... calc_static target definition
7
8 CleanCoverage(calc_static)
```

为 calc_obj 添加了 InstrumentForCoverage，带有额外的--coverage 标志，但是 CleanCoverage() 是为 calc_static 目标调用的。通常，会出于一致性考虑将其应用于 calc_obj，但在这里使用的是 PRE_BUILD 关键字，而 CMake 不允许对对象库使用 PRE_BUILD、PRE_LINK 或 POST_BUILD 钩子。

最后，还将为控制台目标添加仪器化和清理指令：

ch15/01-full-project/src/calc_console/CMakeLists.txt (续)

```
1 # ... calc_console_test target definition
2 # ... BuildInfo
3
4 InstrumentForCoverage(calc_console_static)
5 CleanCoverage(calc_console_static)
```

通过这些步骤，CTest 现在已经设置好来运行测试和收集覆盖率。下一步，将添加指令以启用静态分析，因为我们希望项目在首次构建和后续所有构建中都能保持高质量。

15.5.4. 添加静态分析工具

我们正在接近为目标配置质量保证的完成阶段。最后一步，涉及到启用自动格式化和集成 CppCheck：

ch15/01-full-project/src/calc/CMakeLists.txt (续)

```
1 # ... calc_static target definition
2 # ... Coverage instrumentation and cleaning
3
4 Format(calc_static .)
5
6 AddCppCheck(calc_obj)
```

这里我们遇到了一个小问题：calc_obj 不能有 PRE_BUILD 钩子，所以改为对 calc_static 应用格式化。还需要确保 calc_console_static 目标进行了格式化和检查：

ch15/01-full-project/src/calc_console/CMakeLists.cmake (续)

```
1 # ... calc_console_test target definition
2 # ... BuildInfo
3 # ... Coverage instrumentation and cleaning
4
5 Format(calc_console_static .)
6
7 AddCppCheck(calc_console_static)
```

还需要定义 Format 和 CppCheck 函数。从 Format() 开始，我们借用第 12 章中的代码：

ch15/01-full-project/cmake/Format.cmake

```

1 function(Format target directory)
2     find_program(CLANG-FORMAT_PATH clang-format REQUIRED)
3     set(EXPRESSION h hpp hh c cc cxx cpp)
4     list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")
5     file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
6         LIST_DIRECTORIES false ${EXPRESSION})
7     )
8     add_custom_command(TARGET ${target} PRE_BUILD COMMAND
9         ${CLANG-FORMAT_PATH} -i --style=file ${SOURCE_FILES}
10    )
11 endfunction()

```

为了将 CppCheck 与我们的源码集成，我们使用：

ch15/01-full-project/cmake/CppCheck.cmake

```

1 function(AddCppCheck target)
2     find_program(CPPCHECK_PATH cppcheck REQUIRED)
3     set_target_properties(${target}
4         PROPERTIES CXX_CPPCHECK
5             "${CPPCHECK_PATH};--enable=warning;--error-exitcode=10"
6     )
7 endfunction()

```

这是简单且方便的做法。可能会发现这与 Clang-Tidy 模块（来自第 12 章）展示出 CMake 功能的一致性。

cppcheck 的参数如下：

- `--enable=warning`: 启用警告消息。要启用更多检查，请参阅 Cppcheck 手册（见扩展阅读部分）。
- `--error-exitcode=1`: 设置 `cppcheck` 检测到问题时返回的错误码。这可以是 1 到 255 之间的任何数字（0 表示成功），尽管有些数字可能被系统保留。

随着 `src` 和 `test` 目录中的所有文件创建完毕，解决方案可以构建并进行全面测试。可以继续进行安装和打包步骤。

15.6. 安装与打包

图 15.6 展示了如何配置项目以便进行安装和打包：

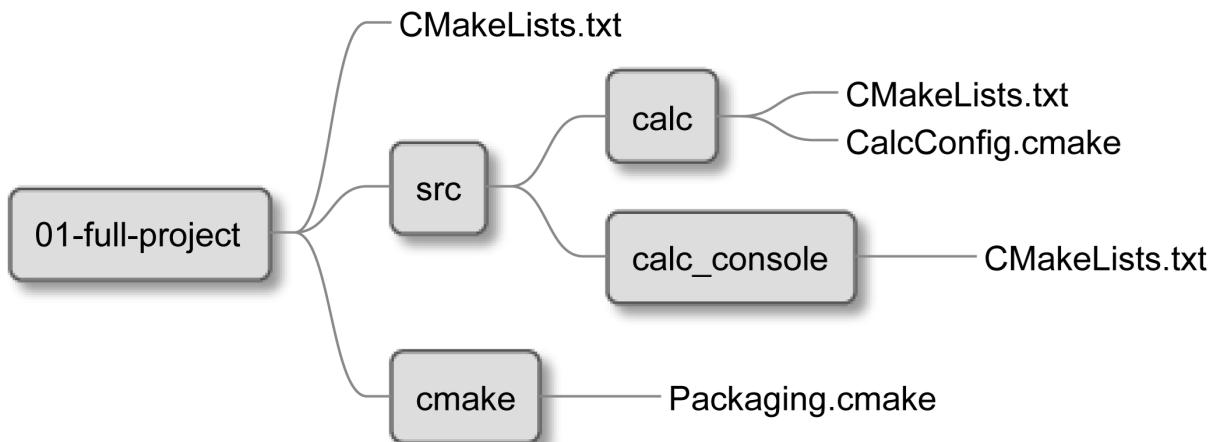


图 15.6：配置安装和打包的文件

顶层列表文件包含了 Packaging 模块：

ch15/01-full-project/CMakeLists.txt (片段)

```

1 # ... configure project
2 # ... enable testing
3 # ... include src and test subdirectories
4
5 include(Packaging)

```

Packaging 模块详细说明了项目的包配置，这部分将在使用 CPack 进行打包的部分进行探讨。现在，关注的是安装三个主要组件：

- Calc 库工件：静态和共享库、头文件以及目标导出文件
- Calc 库的包定义配置文件
- Calc 控制台可执行文件

一切已经计划好了，现在是时候安装库。

15.6.1. 安装库

为了安装库，首先定义逻辑目标及其工件的目的地，利用 GNUInstallDirs 模块的默认值以避免手动指定路径。工件将分组到组件中，默认安装会安装所有文件，可以选择只安装运行时组件，并跳过开发工件：

ch15/01-full-project/src/calc/CMakeLists.txt (续)

```

1 # ... calc library targets definition
2 # ... configuration, testing, program analysis
3
4 # Installation
5 include(GNUInstallDirs)

```

```
6 install(TARGETS calc_obj calc_shared calc_static  
7   EXPORT CalcLibrary  
8   ARCHIVE COMPONENT development  
9   LIBRARY COMPONENT runtime  
10  FILE_SET HEADERS COMPONENT runtime  
11 )
```

对于 UNIX 系统，我们还配置了安装后注册共享库到 ldconfig：

ch15/01-full-project/src/calc/CMakeLists.txt (续)

```
1 if (UNIX)  
2   install(CODE "execute_process(COMMAND ldconfig)"  
3     COMPONENT runtime  
4   )  
5 endif()
```

为了使其他 CMake 项目能够复用，将通过生成并安装一个目标导出文件和一个引用它的配置文件来打包库：

ch15/01-full-project/src/calc/CMakeLists.txt (续)

```
1 install(EXPORT CalcLibrary  
2   DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake  
3   NAMESPACE Calc::  
4   COMPONENT runtime  
5 )  
6  
7 install(FILES "CalcConfig.cmake"  
8   DESTINATION ${CMAKE_INSTALL_LIBDIR}/calc/cmake  
9 )
```

为了简化，CalcConfig.cmake 文件保持最小化：

ch15/01-full-project/src/calc/CalcConfig.cmake

```
1 include("${CMAKE_CURRENT_LIST_DIR}/CalcLibrary.cmake")
```

这个文件位于 src/calc 目录下，它只包含了库目标。如果还有其他目录的目标定义，比如 calc_console，通常会把 CalcConfig.cmake 放在顶层目录或 src 目录下。

现在，库已经准备好通过 cmake --install 命令在构建项目后进行安装。不过，还需要配置可执行文件的安装。

15.6.2. 可执行文件的安装

当然，我们希望用户能够在他们的系统中使用可执行文件，因此需要使用 CMake 来安装。准备二进制可执行文件的安装很简单；为了实现这一点，只需要包含 `GNUInstallDirs` 并使用 `install()` 命令：

ch15/01-full-project/src/calc_console/CMakeLists.txt (续)

```
1 # ... calc_console_static definition
2 # ... configuration, testing, program analysis
3 # ... calc_console bootstrap executable definition
4
5 # Installation
6 include(GNUInstallDirs)
7 install(TARGETS calc_console
8         RUNTIME COMPONENT runtime
9     )
```

这样，可执行文件就设置好进行安装了。现在，继续进行打包。

15.6.3. 使用 CPack 打包

可以尽情配置大量的支持的包类型；但对于这个项目，基本配置就够了：

ch15/01-full-project/cmake/Packaging.cmake

```
1 # CPack configuration
2 set(CPACK_PACKAGE_VENDOR "Rafal Swidzinski")
3 set(CPACK_PACKAGE_CONTACT "email@example.com")
4 set(CPACK_PACKAGE_DESCRIPTION "Simple Calculator")
5 include(CPack)
```

这样的最小化配置对于标准存档，如 ZIP 文件，工作得很好。为了在构建项目后测试安装和打包过程，请在构建树内使用以下命令：

```
# 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: .../packages/Calc-1.0.0-Linux.tar.gz generated.
```

这就完成了安装和打包的过程；接下来的任务是提供文档。

15.7. 提供文档

专业项目的最后一环是文档。未经文档化的项目在团队合作或者与外部受众共享时很难理解和导航。我甚至可以说，开发者在离开某个特定文件一段时间后，经常会阅读自己的文档来理解其中的内容。

文档对于法律和合规性方面也很重要，并且可以告知用户如何使用软件。如果时间允许的话，应该投入精力为项目设置文档。

文档通常分为两类：

- 技术性文档（涵盖接口、设计、类和文件）
- 一般性文档（包含所有其他非技术性文档）

正如在第 13 章中所看到的那样，许多技术文档都可以使用 CMake 和 Doxygen 自动生成。

15.7.1. 生成技术文档

虽然有些项目会在构建阶段生成文档并将其包含在包中，但在这个项目中并没有这样做。但如果文档需要在线托管的话，可能会有正当的理由选择这样做。

图 15.7 概述了文档生成的过程：

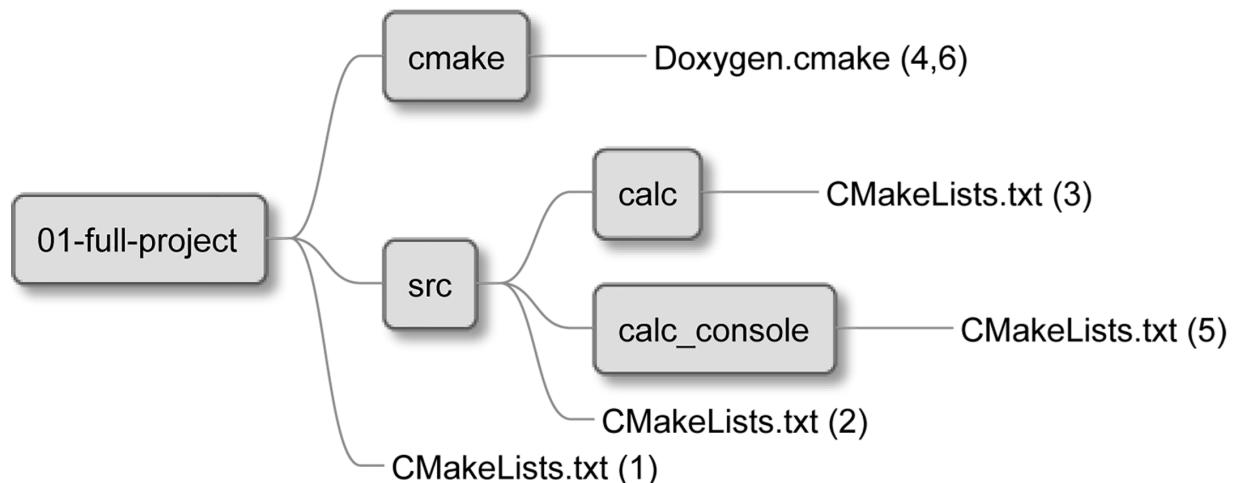


图 15.7：用于生成文档的文件

为了生成文档，将创建另一个 CMake 实用模块——Doxygen。首先使用 Doxygen 找模块，并下载 doxygen-awesome-css 项目用于主题：

ch15/01-full-project/cmake/Doxygen.cmake (片段)

```
1 find_package(Doxygen REQUIRED)
2
3 include(FetchContent)
4 FetchContent_Declare(doxygen-awesome-css
5   GIT_REPOSITORY
6     https://github.com/jothepro/doxygen-awesome-css.git
```

```

7     GIT_TAG
8         v2.3.1
9     )
10    FetchContent_MakeAvailable(doxygen-awesome-css)

```

然后，需要一个函数来创建生成文档的目标。我们将修改第 13 章中的代码，以支持多个目标：

ch15/01-full-project/cmake/Doxygen.cmake (续)

```

1 function(Doxygen target input)
2     set(NAME "doxygen-${target}")
3     set(DOXYGEN_GENERATE_HTML YES)
4     set(DOXYGEN_HTML_OUTPUT ${PROJECT_BINARY_DIR}/${output})
5
6     UseDoxygenAwesomeCss()
7     UseDoxygenAwesomeExtensions()
8
9     doxygen_add_docs("doxygen-${target}"
10        ${PROJECT_SOURCE_DIR}/${input}
11        COMMENT "Generate HTML documentation"
12    )
13 endfunction()
14
15 # ... copied from Ch13:
16 # UseDoxygenAwesomeCss
17 # UseDoxygenAwesomeExtensions

```

使用此函数为库目标调用：

ch15/01-full-project/src/calc/CMakeLists.txt (片段)

```

1 # ... calc_static target definition
2 # ... testing and program analysis modules
3
4 Doxygen(calc src/calc)
5
6 # ... file continues

```

以及为控制台可执行文件：

ch15/01-full-project/src/calc_console/CMakeLists.txt (片段)

```

1 # ... calc_static target definition
2 # ... testing and program analysis modules
3 Doxygen(calc_console src/calc_console)
4
5 # ... file continues

```

这个设置为项目添加了两个目标: doxygen-calc 和 doxygen-calc_console, 按需生成技术文档。现在, 考虑应该包含哪些其他文档。

15.7.2. 为专业项目编写非技术性文档

专业项目应该包含一组存储在顶层目录中的非技术性文档, 这对于全面理解和法律清晰度至关重要:

- README: 提供项目的总体描述
- LICENSE: 详细说明项目使用的法律条款和发行细节

可以考虑的其他文档包括:

- INSTALL: 提供逐步的安装指南
- CHANGELOG: 展示各版本的重要更改
- AUTHORS: 列出贡献者及其联系方式 (如果有多个贡献者)
- BUGS: 提供已知问题的建议和报告新问题的详细信息

CMake 不会直接与这些文件交互, 它们不涉及自动化处理或脚本。然而, 这些文件的存在对于一个文档完备的 C++ 项目来说是至关重要的。下面是每个文档的一个简单示例:

ch15/01-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
```

这是一个简短的例子, 可能看起来有点傻。注意`.md` 扩展名——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, 都会以丰富的格式渲染这些文件。

INSTALL 文件将如下所示:

ch15/01-full-project/INSTALL

```
To install this software you'll need to provide the following:

- C++ compiler supporting C++17
- CMake >= 3.26
```

- 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
```

LICENSE 文件稍微有点棘手，因为需要一些版权法的专业知识。而不是自己编写所有条款，我们可以像许多其他项目一样，使用现成可用的软件许可证。对于这个项目，将使用 MIT 许可证，这是一个极其宽容的许可。请查阅扩展阅读部分获取一些有用的参考资料：

ch15/01-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，我们有 CHANGELOG。正如前面提到的，最好在文件中记录变更，以便使用项目的开发者可以轻松找到他们所需的特性所在的版本。例如，可以说在 0.8.2 版本中向库中添加了乘法功能。像下面这样简单的记录就已经很有帮助了：

ch15/01-full-project/CHANGELOG

```
1.1.0 Updated for CMake 3.26 in 2nd edition of the book
1.0.0 Public version with installer
0.8.2 Multiplication added to the Calc Library
0.5.1 Introducing the Calc Console application
0.2.0 Basic Calc library with Sum function
```

有了这些文档，项目不仅获得了操作结构，还能有效地传达其使用方法、变更和法律考量，确保用户和贡献者可以获得所有必要的信息。

15.8. 总结

本章中，基于目前为止所学的内容构建了一个专业项目。让我们快速回顾一下。

首先规划了项目的布局，并讨论了哪些文件将位于哪个目录中。基于以往的经验，以及想要实践更高级场景的愿望，界定了一个面向用户的主应用程序和一个其他开发者可能使用的库。这塑造了目录的结构，以及希望构建的 CMake 目标之间的关系。随后，配置了各个构建目标：提供了库的源代码，定义了它的目标，并设置了使用位置独立代码参数供使用。面向用户的应用程序也可以定义其可执行目标，提供了源代码，并配置了其依赖项：FTXUI 库。

有了待构建的制品之后，继续增强了项目的测试和质量保证功能。添加了覆盖率模块来生成覆盖率报告，使用 Memcheck 在运行时通过 Valgrind 验证解决方案，并执行了静态分析 CppCheck。

这样一个项目现在已经准备好安装，所以我们使用学到的技术为库和可执行文件创建了适当的安装条目，并为 CPack 准备了包配置。最后的任务是确保项目文档正确无误，因此设置了使用 Doxygen 自动生成文档，并编写了几份基本文档来处理软件分发中不太技术性的方面。

这就完成了项目配置，并且现在可以轻松地使用几个精确的 CMake 命令来构建和安装项目。但如果我们可以只用一个简单的命令来完成整个过程呢？让我们在最后一章：第 16 章中探索这个主题。

15.9. 扩展阅读

- 构建静态库和共享库的相关信息：

<https://stackoverflow.com/q/2152077>

- FXTUI 库项目：
<https://github.com/ArthurSonzogni/FTXUI>
- option() 命令的文档：
<https://cmake.org/cmake/help/latest/command/option.html>
- Google 关于开源软件发布的准备指南：
<https://opensource.google/docs/releasing/preparing/>
- 为什么不能在 GCC 预编译头文件中使用 Clang-Tidy：
https://gitlab.kitware.com/cmake/cmake/-/issues/22081#note_943104
- Cppcheck 手册：
<https://cppcheck.sourceforge.io/manual.pdf>
- 如何撰写 README 文件：
<https://www.freecodecamp.org/news/how-to-write-a-good-readme-file/>
- GitHub 项目的 Creative Commons 许可证：
<https://github.com/santisoler/cc-licenses>
- GitHub 认可的常用项目许可证：
<https://docs.github.com/enrepositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository>

第 16 章 编写 CMake 预设

CMake 3.19 版本中引入了预设 (Presets)，目的是简化项目设置的管理。预设出现之前，用户需要记住冗长的命令行配置，或是直接在项目文件中设置覆盖选项，这往往会变得复杂且容易出错。预设让用户能够以更简单的方式处理，诸如用于配置项目的生成器、并发构建任务的数量，以及要构建或测试的项目组件等设置。通过使用预设，CMake 变得更加易于使用。用户可以一次性设置预设，并在需要时使用它们，从而使每次 CMake 执行更加一致且易于理解。还有助于跨不同用户和计算机标准化设置，从而简化协作项目的工作流程。

预设兼容 CMake 的四种主要模式：配置构建系统、构建、运行测试和打包。允许用户将这些部分链接在一起形成工作流，使整个过程更加自动化和有序。此外，预设提供了如条件和宏表达式（或简称宏）等功能，给予用户更大的控制力。

本章中，将包含以下内容：

- 使用项目中定义的预设
- 编写预设文件
- 定义特定阶段的预设
- 定义工作流预设
- 添加条件和宏

16.1. 示例下载

可以在 GitHub 上找到本章中存在的代码文件，地址是 <https://github.com/PacktPublishing/Modern-CMake-for-Cpp-2E/tree/main/examples/ch16>。

执行本章示例所需的命令将在每个章节中提供。

16.2. 使用项目中定义的预设

项目的配置可能成为一个复杂的任务，特别是需要具体指定缓存变量、选定的生成器等元素时——尤其是当项目有多种构建方式的时候，这时预设就变得非常有用。不必记住命令行参数或是编写 shell 脚本，来以不同的参数运行 cmake，而是可以创建一个预设文件，并将所需的配置存储在项目中。

CMake 使用两个可选文件来存储项目预设：

- CMakePresets.json：由项目作者提供的官方预设。
- CMakeUserPresets.json：专为希望向项目添加自定义预设的用户设计。项目应当将此文件添加到版本控制系统忽略列表中，以确保自定义设置不会共享到仓库中。

预设文件必须放置在项目的顶层目录中以便 CMake 能够识别。每个预设文件可以为每个阶段定义多个预设：配置 (configure)、构建 (build)、测试 (test)、打包 (package)，以及涵盖多个阶段的工作流 (workflow) 预设。然后用户可以通过集成开发环境 (IDE)、图形用户界面 (GUI) 或者命令行选择并执行一个预设。

预设可以通过在命令行中添加 `--list-presets` 参数来列出，具体取决于我们要列出的阶段。例如，列出构建预设可以用下面的命令：

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

列出测试预设可以用下面的命令：

```
ctest --list-preset
```

要使用一个预设，需要遵循相同的模式，并在 `--preset` 参数之后提供预设名称。

另外，不能使用 `cmake` 命令来列出打包预设，需要使用 `cpack`。这里是一个用于打包预设的命令行示例：

```
cpack --preset <preset-name>
```

选择好预设后，还可以添加特定于阶段的命令行参数，例如指定构建树或者安装路径。添加的参数会覆盖预设中设置的内容。

工作流预设有特殊情况，运行 `cmake` 命令时如果存在 `--workflow` 参数，则可以列出并应用工作流预设：

```
$ cmake --workflow --list-presets
Available workflow presets:
"myWorkflow"
$ cmake --workflow --preset myWorkflow
Executing workflow step 1 of 4: configure preset "myConfigure"
...
```

这就是如何在项目中应用和查看可用预设的方法。现在，让我们探索一下如何构建预设文件。

16.3. 编写预设文件

CMake 在项目的顶层目录中搜索 `CMakePresets.json` 和 `CMakeUserPresets.json` 文件。这两个文件使用相同的 JSON 结构来定义预设，它们之间的区别不大。其格式是一个 JSON 对象，包含以下键：

- `version`: 这是一个必需的整数，指定了预设 JSON 架构的版本。
- `cmakeMinimumRequired`: 这是一个对象，指定了所需的 CMake 版本。
- `include`: 这是一个字符串数组，从数组中提供的文件路径包含外部预设（自第 4 版架构开始）。
- `configurePresets`: 这是一个对象数组，定义了配置阶段的预设。
- `buildPresets`: 这是一个对象数组，定义了构建阶段的预设。
- `testPresets`: 这是一个对象数组，专门针对测试阶段的预设。
- `packagePresets`: 这是一个对象数组，专门针对打包阶段的预设。
- `workflowPresets`: 这是一个对象数组，专门针对工作流模式的预设。

- `vendor`: 这是一个对象，包含由 IDE 和其他供应商定义的自定义设置；CMake 不处理这个字段。

编写预设时，CMake 要求存在 `version` 入口；其他值则可选。下面是一个预设文件的例子（实际的预设将在后续部分添加）：

ch16/01-presets/CMakePresets.json

```

1  {
2      "version": 6,
3      "cmakeMinimumRequired": {
4          "major": 3,
5          "minor": 26,
6          "patch": 0
7      },
8      "include": [],
9      "configurePresets": [],
10     "buildPresets": [],
11     "testPresets": [],
12     "packagePresets": [],
13     "workflowPresets": [],
14     "vendor": {
15         "data": "IDE-specific information"
16     }
17 }
```

上述例子中，并不需要添加空数组；除了 `version` 以外的条目都是可选的。顺便说一下，对于 CMake 3.26 的适当架构版本是 6。

已经了解了预设文件的结构，接下来我们就学习如何实际定义这些预设。

16.4. 特定阶段的预设

特定阶段的预设仅仅是配置各个 CMake 阶段的预设：配置、构建、测试、打包和安装，允许以精细和结构化的方式来定义构建配置。以下是所有预设阶段共有的特性概览，随后将介绍如何为各个阶段定义预设。

16.4.1. 各预设阶段的共同特性

无论 CMake 阶段如何，有三个特性用于配置预设，这些特性包括独特的名称字段、可选字段，以及与配置阶段预设的关联。

独特的名称字段

每个预设在其所属阶段内必须有一个独特的名称字段。如果 `CMakeUserPresets.json` 存在，它会隐式地包含 `CMakePresets.json`（如果该文件也存在），所以两个文件共享命名空间，避免了名称重复。例如，不能在两个文件中都有名为 `myPreset` 的打包阶段预设。

一个最小化的预设文件可能如下所示：

```
1  {
2      "version": 6,
3      "configurePresets": [
4          {
5              "name": "myPreset"
6          },
7          {
8              "name": "myPreset2"
9          }
10     ]
11 }
```

可选字段

每个特定阶段的预设都使用相同的可选字段：

- `displayName`: 一个字符串，为预设提供一个用户友好的名称。
- `description`: 一个字符串，说明预设的功能。
- `inherits`: 一个字符串或字符串数组，有效地复制了在此字段中命名的预设的配置作为基础，进一步扩展或修改。
- `hidden`: 一个布尔值，隐藏预设使其不显示在列表中；这样的隐藏预设只能通过继承使用。
- `environment`: 一个对象，为这个阶段覆盖环境变量；每个键标识一个单独的变量，值可以是字符串或 `null`；它支持宏。
- `condition`: 一个对象，启用或禁用此预设（稍后会有更多介绍）。
- `vendor`: 一个自定义对象，包含供应商特定的值，并遵循与根级 `vendor` 字段相同的约定。

预设可以形成类似图的继承结构，前提是不存在循环依赖。`CMakeUserPresets.json` 可以从项目级别的预设继承，但反之则不行。

与配置阶段预设的关联

所有特定阶段的预设都必须与配置预设相关联，它们必须知道构建树的位置。虽然配置预设本质上与自身关联，但构建、测试和打包预设，需要明确地通过 `configurePreset` 字段定义这种关联。

这种关联并不意味着当决定运行后续预设时，CMake 将自动执行配置预设。仍然需要手动执行每个预设，或者使用工作流预设（稍后会介绍这一点）。

有了这些基础概念，可以继续深入探讨各个阶段的预设细节，从配置阶段开始。随着进展，将探索这些预设如何相互作用，以及如何用来简化 CMake 中的项目配置和构建过程。

16.4.2. 定义配置阶段预设

配置预设位于 `configurePresets` 数组中，可以通过向命令行添加 `--list-presets` 参数来列出，具体针对配置阶段：

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

要用选定的预设配置项目，请在 `--preset` 参数后指定其名称：

```
cmake --preset myConfigurationPreset
```

配置预设有像名称和描述这样的通用字段，但它也有自己的一套可选字段。以下是最重要的几个字段的简化描述：

- `generator`: 一个字符串，指定了预设使用的生成器；对于小于第 3 版的架构是必需的。
- `architecture` 和 `toolset`: 一个字符串，配置支持这些选项的生成器。
- `binaryDir`: 一个字符串，提供了构建树的相对或绝对路径；对于小于第 3 版的架构是必需的；支持宏。
- `installDir`: 一个字符串，提供了安装目录的相对或绝对路径；对于小于第 3 版的架构是必需的；支持宏。
- `cacheVariables`: 一个映射/map，定义了缓存变量；值支持宏。

定义 `cacheVariables` 映射时，要记住项目中变量解析的顺序。如图 16.1 所示，通过命令行定义的任何缓存变量都会覆盖预设变量。缓存或环境预设变量都会覆盖来自缓存文件或主机环境的变量。

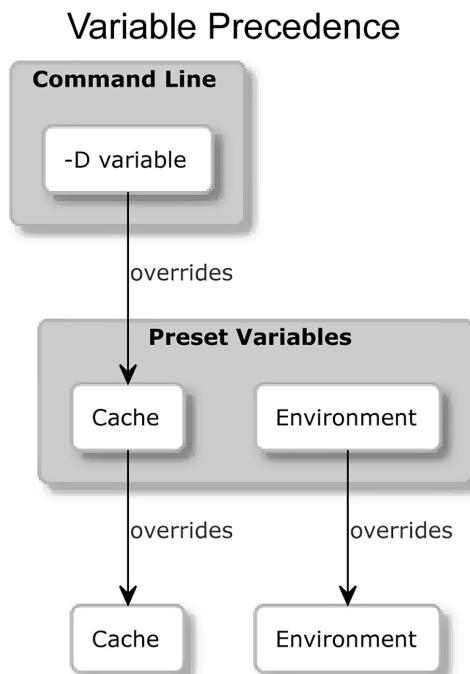


图 16.1：预设如何覆盖 CMakeCache.txt 和系统环境变量

声明一个简单的 `myConfigure` 配置预设，其指定了生成器、构建树和安装路径：

ch16/01-presets/CMakePresets.json (continued)

```
1 ...
2 "configurePresets": [
```

```
3     {
4         "name": "myConfigure",
5         "displayName": "Configure Preset",
6         "description": "Ninja generator",
7         "generator": "Ninja",
8         "binaryDir": "${sourceDir}/build",
9         "installDir": "${sourceDir}/build/install"
10    }
11 ],
12 ...
```

对配置预设的介绍已经完成，接下来是构建阶段预设。

16.4.3. 定义构建阶段预设

不会惊讶于发现构建预设位于 `buildPresets` 数组中。可以通过向命令行添加 `--list-presets` 参数来列出它们，具体针对构建阶段：

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

要用选定的预设构建项目，请在 `--preset` 参数后指定其名称：

```
cmake --build --preset myBuildingPreset
```

构建预设有像名称和描述这样的通用字段，其也有一套自己的可选字段。以下是最重要的几个字段的简化描述：

- `jobs`: 一个整数，设置了用于构建项目的并行作业的数量。
- `targets`: 一个字符串或字符串数组，设置了要构建的目标，并支持宏。
- `configuration`: 一个字符串，确定了多配置生成器的构建类型 (Debug, Release 等)。
- `cleanFirst`: 一个布尔值，确保在构建前总是清理项目。

现在，可以这样写一个构建预设：

ch16/01-presets/CMakePresets.json (续)

```
1 ...
2     "buildPresets": [
3         {
4             "name": "myBuild",
5             "displayName": "Build Preset",
6             "description": "Four jobs",
7             "configurePreset": "myConfigure",
8             "jobs": 4
9         }
```

```
10   ],
11 ...
```

必需的 `configurePreset` 字段设置为指向我们在上一节中定义的 `myConfigure` 预设。现在，可以继续介绍测试预设。

16.4.4. 定义测试阶段预设

测试预设位于 `testPresets` 数组中。可以通过向命令行添加 `--list-presets` 参数来列出它们，具体针对测试阶段：

```
ctest --list-presets
```

要用预设测试项目，请在 `--preset` 参数后指定其名称：

```
ctest --preset myTestPreset
```

测试预设有它自己的一套可选字段。以下是最重要的几个字段的简化描述：

- `configuration`: 一个字符串，确定了多配置生成器的构建类型（`Debug`, `Release` 等）
- `output`: 一个对象，配置输出。
- `filter`: 一个对象，指定要运行哪些测试。
- `execution`: 一个对象，配置测试的执行。

每个对象将相应的命令行选项映射到配置值，但这并不是一个详尽的列表。请参阅“扩展阅读”部分获取完整的参考。

`output` 对象的可选条目包括：

- `shortProgress`: 布尔值；进度将在一行内报告。
- `verbosity`: 一个字符串，将输出的详细程度设置为以下级别之一：默认、详细或其他信息。
- `outputOnFailure`: 布尔值；在测试失败时打印程序输出。
- `quiet`: 布尔值；抑制所有输出。

对于 `exclude`，接受的一些条目是：

- `name`: 一个字符串，排除匹配正则表达式模式的测试名称，并支持宏。
- `label`: 一个字符串，排除匹配正则表达式模式的测试标签，并支持宏。
- `fixtures`: 一个对象，确定要排除哪些测试夹具（参见官方文档了解更多详情）。

最后，`execution` 对象接受以下可选条目：

- `outputLogFile`: 一个字符串，指定了输出日志文件的路径，并支持宏。

`filter` 对象接受 `include` 和 `exclude` 键来配置测试案例的筛选；

```
1 "testPresets": [
2   {
```

```

3     "name": "myTest",
4     "configurePreset": "myConfigure",
5     "filter": {
6       "include": {
7         ... name, label, index, useUnion ...
8       },
9       "exclude": {
10      ... name, label, fixtures ...
11    }
12  }
13 ],
14 ...
15 ...

```

每个键定义了自己的选项对象：

对于 `include`:

- `name`: 一个字符串，包含匹配正则表达式模式的测试名称，并支持宏。
- `label`: 一个字符串，包含匹配正则表达式模式的测试标签，并支持宏。
- `index`: 一个对象，通过接受起始、结束和步长整数以及，一个特定测试的整数数组来选择要运行的测试，支持宏。
- `useUnion`: 一个布尔值，启用使用由 `index` 和 `name` 确定的测试的并集，而不是交集。

对于 `exclude`，条目包括：

- `name`: 一个字符串，排除匹配正则表达式模式的测试名称，并支持宏。
- `label`: 一个字符串，排除匹配正则表达式模式的测试标签，并支持宏。
- `fixtures`: 一个对象，确定要从测试中排除哪些夹具（参见官方文档了解更多信息）。

最后，`execution` 对象可以在这里添加：

```

1   "testPresets": [
2     {
3       "name": "myTest",
4       "configurePreset": "myConfigure",
5       "execution": {
6         ... stopOnFailure, enableFailover, ...
7         ... jobs, repeat, scheduleRandom, ...
8         ... timeout, noTestsAction ...
9       }
10      }
11    ],
12 ...

```

接受以下可选条目：

- `stopOnFailure`: 一个布尔值，任何测试失败则停止测试。
- `enableFailover`: 一个布尔值，恢复之前中断的测试。

- `jobs`: 一个整数，以并行方式运行多个测试。
- `repeat`: 一个对象，确定如何重复测试；该对象必须具有以下字段：
 - `mode` -一个字符串，其值可以是: `until-fail`, `until-pass`, `aftertimeout`。
 - `count` -一个整数，确定重复次数。
- `scheduleRandom`: 一个布尔值，启用随机顺序执行测试。
- `timeout`: 一个整数，设置所有测试总执行时间的限制（秒）。
- `noTestsAction`: 一个字符串，定义如果没有找到测试时的动作，选项包括: `default`, `error`, 和 `ignore`。

尽管有许多配置选项，简单的预设也可行：

ch16/01-presets/CMakePresets.json (续)

```

1 ...
2 "testPresets": [
3   {
4     "name": "myTest",
5     "displayName": "Test Preset",
6     "description": "Output short progress",
7     "configurePreset": "myConfigure",
8     "output": {
9       "shortProgress": true
10    }
11  }
12 ],
13 ...

```

与构建预设一样，我们也为新的测试预设设置了必需的 `configurePreset` 字段，以便整洁地将它们联系起来。来看一下最后一个特定阶段的预设类型：打包预设。

16.4.5. 定义打包阶段预设

打包预设是在第 6 版架构中引入的，所以需要至少使用 CMake 3.25 才能使用它们。这些预设应该包含在 `packagePresets` 数组中，也可以通过向命令行追加 `--list-presets` 参数来显示，具体针对打包阶段：

```
cpack --list-presets
```

要用预设创建项目包，请在 `--preset` 参数后指定其名称：

```
cpack --preset myTestPreset
```

打包预设利用与其他预设相同的共享字段，并引入了一些特定于自身的可选字段：

- `generators`: 一个字符串数组，设置了要使用的包生成器 (ZIP, 7Z, DEB 等)。

- configuration: 一个字符串数组，确定了 CPack 要打包的构建类型列表。(Debug, Release 等)。
- filter: 一个对象，指定了要运行的测试。
- packageName, packageVersion, packageDirectory 和 vendorName: 字符串，指定了创建的包的元数据。

在预设文件中添加一个简洁的打包预设：

ch16/01-presets/CMakePresets.json (续)

```

1 ...
2     "packagePresets": [
3         {
4             "name": "myPackage",
5             "displayName": "Package Preset",
6             "description": "ZIP generator",
7             "configurePreset": "myConfigure",
8             "generators": [
9                 "ZIP"
10            ]
11        }
12    ],
13 ...

```

这样的配置将简化项目包的创建，但还缺少一个关键成分：项目安装。

16.4.6. 添加安装预设

可能已经注意到 CMakePresets.json 对象不支持定义“installPresets”。没有明确的方式通过预设安装项目，这似乎有些奇怪，因为配置预设提供了 installDir 字段！那么是否需要诉诸于手动安装命令？

幸运的是，不需要。有一个变通方法可以让我们使用构建预设实现目标。

来看一下：

ch16/01-presets/CMakePresets.json (续)

```

1 ...
2     "buildPresets": [
3         {
4             "name": "myBuild",
5             ...
6         },
7         {
8             "name": "myInstall",
9             "displayName": "Installation",
10            "targets" : "install",

```

```
11     "configurePreset": "myConfigure"
12   }
13 ],
14 ...
```

可以创建一个构建预设，其中 `targets` 字段设置为 `install`。当正确配置安装时，`install` 目标由项目隐式定义。使用这个预设构建将执行必要的步骤，来安装项目到在关联的配置预设中指定的 `installDir`（如果 `installDir` 字段为空，则使用默认位置）：

```
$ cmake --build --preset myInstall
[0/1] Install the project...
-- Install configuration: ""
-- Installing: .../install/include/calc/basic.h
-- Installing: .../install/lib/libcalc_shared.so
-- Installing: .../install/lib/libcalc_static.a
-- Installing: .../install/lib/calc/cmake/CalcLibrary.cmake
-- Installing: .../install/lib/calc/cmake/CalcLibrary-noconfig.cmake
-- Installing: .../install/lib/calc/cmake/CalcConfig.cmake
-- Installing: .../install/bin/calc_console
-- Set non-toolchain portion of runtime path of ".../install/bin/calc_
console" to ""
```

这个巧妙的方法可以帮助我们节省一些步骤。如果可以为最终用户提供一个单一命令，从配置到安装全部处理好，那就更好了。确实可以做到，通过工作流预设。

16.5. 定义工作流预设

工作流预设是我们项目的终极自动化解决方案，允许我们按照预定顺序自动执行多个特定阶段的预设。这样，实际上可以用一步完成端到端的构建。

要发现项目的可用工作流，我们可以执行以下命令：

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

要选择并应用一个预设，使用以下命令：

```
cmake --workflow --preset <preset-name>
```

此外，使用 `--fresh` 标志，可以清除构建树和缓存。

定义工作流预设非常简单；需要定义一个名称，并且可以像为特定阶段的预设那样可选地提供 `displayName` 和 `description`。之后，必须枚举出工作流应该执行的所有特定阶段的预设。这是通过提供一个 `steps` 数组来完成的，数组中的对象包含 `type` 和 `name` 属性：

ch16/01-presets/CMakePresets.json (续)

```
1 ...
2 "workflowPresets": [
3 {
4     "name": "myWorkflow",
5     "steps": [
6         {
7             "type": "configure",
8             "name": "myConfigure"
9         },
10        {
11            "type": "build",
12            "name": "myBuild"
13        },
14        {
15            "type": "test",
16            "name": "myTest"
17        },
18        {
19            "type": "package",
20            "name": "myPackage"
21        },
22        {
23            "type": "build",
24            "name": "myInstall"
25        }
26    ]
27 ...
```

`steps` 数组中的每个对象引用了本章中早些时候定义的预设，指示其类型（`configure`, `build`, `test`, 或 `package`）和名称。这些预设一起执行所有必要的步骤，只需一条命令即可从头开始完全构建和安装项目：

```
cmake --workflow --preset myWorkflow
```

工作流预设是用于自动化 C++ 构建、测试、打包和安装的解决方案。接下来，让我们探讨如何通过条件和宏来管理一些特殊情况。

16.6. 添加条件和宏

当讨论每个特定阶段预设的一般字段时，提到了 `condition` 字段。现在是回到这个话题的时候了。`condition` 字段可以启用或禁用一个预设，在与工作流集成时揭示其真正的潜力，它允许在某些条件下绕过不适合的预设，并创建替代的预设。

条件要求预设架构版本 3 或以上（在 CMake 3.22 中引入），并且是 JSON 对象，编码了一些简单的逻辑操作，可以根据所使用的操作系统、环境变量甚至选定的生成器等条件来判断预设

是否适用。CMake 通过宏提供了这些数据，这些宏基本上是一组只读变量，可以在预设文件中使用。

condition 对象的结构根据检查类型的不同而变化。每个条件都必须包含一个 type 字段，以及由该类型定义的其他字段。其基本类型包括：

- const: 这个检查 value 字段中提供的值是否为布尔真。
- equals 和 notEquals: 比较 lhs 字段的值与 rhs 字段的值。
- inList 和 notInList: 这些检查 string 字段中提供的值是否存在子 list 字段的数组中。
- matches 和 notMatches: 这些判断 string 字段的值是否符合 regex 字段中定义的模式。

一个示例条件如下所示：

```
1 "condition": {  
2     "type": "equals",  
3     "lhs": "${hostSystemName}",  
4     "rhs": "Windows"  
5 }
```

const 条件的实际用途，主要是为了在不从 JSON 文件中删除预设的情况下禁用。除了 const 之外，所有基本条件都允许在其引入的字段 (lhs, rhs, string, list, regex) 中使用宏。

高级条件类型，类似于“not”，“and”和“or”操作，使用其他条件作为参数：

- not: 这是对 condition 字段中提供的条件进行布尔反转。
- anyOf 和 allOf: 这些检查 conditions 数组中的任意条件或所有条件是否为真。

例如：

```
1 "condition": {  
2     "type": "anyOf",  
3     "conditions": [  
4         {  
5             "type": "equals",  
6             "lhs": "${hostSystemName}",  
7             "rhs": "Windows"  
8         },  
9         {  
10            "type": "equals",  
11            "lhs": "${hostSystemName}",  
12            "rhs": "Linux"  
13        }  
14    ]  
15 }
```

这个条件在系统为 Linux 或 Windows 时计算为真。

通过这些示例，引入了第一个宏：\${hostSystemName}。宏遵循简单的语法，并限于特定实例：

- \${sourceDir}: 源代码树的路径。
- \${sourceParentDir}: 源代码树父目录的路径。
- \${sourceDirName}: 项目的目录名。
- \${presetName}: 预设的名称。
- \${generator}: 用于创建构建系统的生成器。
- \${hostSystemName}: 系统名称：Linux, Windows, 或 macOS 上的 Darwin。
- \${fileDir}: 包含当前预设的文件名（当使用 include 数组导入外部预设时适用）。
- \${dollar}: 转义的美元符号 \$。
- \${pathListSep}: 、环境特定的路径分隔符。
- \${env{<variable-name>}}: 如果预设指定了环境变量，则返回该环境变量（区分大小写），否则返回父环境中的值。
- \${env{<variable-name>}}: 返回父环境中的环境变量。
- \${vendor{<macro-name>}}: 允许 IDE 厂商引入自己的宏。

这些宏为在预设及其条件中的使用提供了足够的灵活性，使我们能够根据需要有效地切换工作流步骤。

16.7. 总结

我们刚刚完成了一个关于 CMake 预设的全面概述，这些预设是从 CMake 3.19 版本开始引入的，旨在简化项目管理。预设让产品作者可以通过配置项目构建和交付的所有阶段来为用户提供整洁的体验。预设不仅简化了 CMake 的使用，还提高了一致性，并允许根据环境进行设置。

我们解释了 CMakePresets.json 和 CMakeUserPresets.json 文件的结构和用法，并提供了关于定义各种类型的预设的见解，包括配置预设、构建预设、测试预设、打包预设和工作流预设。每种类型都进行了详细的描述：了解了常见的字段，如何在内部构建预设、建立它们之间的继承关系，以及为最终用户提供特定的配置选项。

对于配置预设，讨论了重要的主题，如选择生成器、构建目录和安装目录，并通过 configurePreset 字段将预设链接在一起。现在知道了如何处理构建预设和设置构建作业数量、目标以及清理选项。随后，学习了测试预设如何通过广泛的过滤和排序选项、输出格式化，以及执行参数（如超时和容错）来辅助测试选择。了解了如何通过指定打包生成器、过滤选项，以及打包元数据来管理打包预设。甚至还介绍了一种通过专门的构建预设应用，来执行安装阶段的变通方法。

接着，我们发现了工作流预设如何将多个特定阶段的预设组合在一起。最后，讨论了条件和宏表达式，为项目作者提供了对单个预设的行为，及其集成到工作流中的更大控制权。

我们的 CMake 之旅至此结束！恭喜你——现在拥有了开发、测试和打包高质量 C++ 软件所需的所有工具。最好的前进方式就是应用你所学的知识去创造优秀的软件给你的用户。祝各位好运！

16.8. 扩展阅读

- 预设的官方文档：

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

附录

17.1. 其他命令

每种编程语言都包含了对各种任务有用的实用命令，CMake 也不例外。它提供了用于算术运算、位操作、字符串处理以及列表和文件操作的工具。尽管由于增强功能和大量模块的发展，这些命令的需求已经减少，但在高度自动化的项目中它们仍然重要。如今，可能会更多地在使用 `cmake -P <filename>` 调用的 CMake 脚本中发现它们的作用。

因此，本附录总结了各种 CMake 命令及其不同的模式，可以作为方便的离线参考，或是官方文档的简化版。对于更详细的信息，建议查阅提供的链接。

此参考适用于 CMake 3.26.6 版本。

本附录中，包括以下内容：

- `string()`
- `list()`
- `file()`
- `math()`

17.2. `string()`

`string()` 命令用于处理字符串。具有多种模式，可以对字符串执行不同的操作：搜索替换、操作、比较、哈希、生成，以及 JSON 操作（最后一个自 CMake 3.19 开始可用）。

完整的详情可以在在线文档中找到：<https://cmake.org/cmake/help/latest/command/string.html>

Note

接受 `<input>` 参数的 `string()` 模式将接受多个 `<input>` 值并在命令执行前将它们连接起来，因此：

```
string(PREPEND myVariable "a" "b" "c")
```

等同于以下操作：

```
string(PREPEND myVariable "abc")
```

可用的 `string()` 模式包括搜索替换、操作、比较、哈希、生成以及 JSON。

A.2.1. 搜索和替换

以下是可用的模式：

- `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。如果使用了 REVERSE 标志，则从字符串末尾向开头搜索。这仅适用于 ASCII 字符串（不支持多字节字符）。
- `string(REPLACE <pattern> <replace> <out> <input>)`: 在 `<input>` 中替换所有 `<pattern>` 的出现，并将结果存储在 `<out>` 变量中。
- `string(REGEX MATCH <pattern> <out> <input>)`: 使用正则表达式匹配 `<input>` 中 `<pattern>` 的首次出现，并将结果存储在 `<out>` 变量中。
- `string(REGEX MATCHALL <pattern> <out> <input>)`: 使用正则表达式匹配 `<input>` 中所有 `<pattern>` 的出现，并将结果作为一个逗号分隔的列表存储在 `<out>` 变量中。
- `string(REGEX REPLACE <pattern> <replace> <out> <input>)`: 使用正则表达式替换 `<input>` 中所有 `<pattern>` 的出现，并将替换表达式的结果存储在 `<out>` 变量中。

正则表达式操作遵循 C++ 语法，如标准库中的 `<regex>` 头文件所定义。可以使用捕获组将匹配添加到 `<replace>` 表达式中，使用数字占位符：`\\\1, \\\2 ...`（需要双反斜杠以正确解析参数）。

A.2.2. 操作

以下是可用的模式：

- `string(APPEND <out> <input>)`: 通过追加 `<input>` 字符串来修改存储在 `<out>` 中的字符串。
- `string(PREPEND <out> <input>)`: 通过前置 `<input>` 字符串来修改存储在 `<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>)`: 在所有提供的 `<input>` 字符串之间插入 `<glue>` 值，并将它们作为连接后的字符串存储在 `<out>` 变量中（不要将此模式用于列表变量）。
- `string(TOLOWER <string> <out>)`: 将 `<string>` 转换为小写并将其存储在 `<out>` 变量中。
- `string(TOUPPER <string> <out>)`: 将 `<string>` 转换为大写并将其存储在 `<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>)`: 提取从 `<begin>` 字节开始长度为 `<length>` 的子字符串，并将其存储在 `<out>` 变量中。将 -1 作为长度被理解为“直到字符串末尾”。

- `string(STRIPE <string> <out>)`: 从 `<string>` 中移除前后空白并将结果存储在 `<out>` 变量中。
- `string(GENEX_STRIP <string> <out>)`: 从 `<string>` 中移除所有生成器表达式并将结果存储在 `<out>` 变量中。
- `string(REPEAT <string> <count> <out>)`: 生成包含 `<count>` 次重复的 `<string>` 的字符串，并将其存储在 `<out>` 变量中。

A.2.3. 比较

字符串比较采用以下形式：

```
1 string(COMPARE <operation> <stringA> <stringB> <out>)
```

`<operation>` 参数可为：

- LESS
- GREATER
- EQUAL
- NOTEQUAL
- LESS_EQUAL
- GREATER_EQUAL

用于比较 `<stringA>` 和 `<stringB>`，并将结果（真或假）存储在 `<out>` 变量中。

A.2.4. 哈希

哈希模式具有以下签名：

```
1 string(<hashing-algorithm> <out> <string>)
```

使用 `<hashing-algorithm>` 对 `<string>` 进行哈希处理，并将结果存储在 `<out>` 变量中。支持以下算法：

- MD5: 消息摘要算法 5, RFC 1321
- SHA1: 美国安全哈希算法 1, RFC 3174
- SHA224: 美国安全哈希算法, RFC 4634
- SHA256: 美国安全哈希算法, RFC 4634
- SHA384: 美国安全哈希算法, RFC 4634
- SHA512: 美国安全哈希算法, RFC 4634
- SHA3_224: Keccak SHA-3
- SHA3_256: Keccak SHA-3
- SHA3_384: Keccak SHA-3
- SHA3_512: Keccak SHA-3

A.2.5. 生成

以下是可用的模式：

- `string(ASCII <number>... <out>)`: 将给定 `<number>` 的 ASCII 字符存储在 `<out>` 变量中。
- `string(HEX <string> <out>)`: 将 `<string>` 转换为其十六进制表示并将其存储在 `<out>` 变量中（自 CMake 3.18 开始）。
- `string(CONFIGURE <string> <out> [<ONLY>] [<ESCAPE_QUOTES>])`: 完全像 `configure_file()` 一样工作，但针对字符串。结果存储在 `<out>` 变量中。作为提醒，使用 `<ONLY>` 关键词将限制替换为形如 `@VARIABLE@` 的变量。
- `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>)`: 使用随机种子 `<seed>` 和可选的 `<alphabet>` 生成长度为 `<len>`（默认为 5）的随机字符串，并将结果存储在 `<out>` 变量中。
- `string(TIMESTAMP <out> [<format>] [<UTC>])`: 生成代表当前日期和时间的字符串，并将其存储在 `<out>` 变量中。
- `string(UUID <out> <namespace> <name> <type>)`: 生成全局唯一标识符。此模式的应用稍微复杂，需要提供一个命名空间（必须是一个 UUID）、一个名称（例如，域名）以及一个类型（MD5 或 SHA1）。

A.2.6. JSON

对 JSON 格式的字符串的操作采用以下签名：

```
1 string(JSON <out> [<ERROR_VARIABLE> <error>] <operation + args>)
```

有几个操作可用。都将结果存储在 `<out>` 变量中，错误则存储在 `<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>...` : 使用 `<member>` 路径或 `<index>` 从 `<json>` 字符串中返回位于 `<array-index>` 位置的一个或多个数组类型的元素的成员名称。
- `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>...` : 使用 `<member>` 路径或 `<index>` 从 `<json>` 字符串中移除一个或多个元素。

- SET <json> <member|index>... : 使用 <member> 路径或 <index> 将 <value> 更新或插入到 <json> 字符串中的一个或多个元素中。
- EQUAL <jsonA> <jsonB>: 评估 <jsonA> 和 <jsonB> 是否相等。

17.3. list()

此命令提供了基本的列表操作：读取、查找、修改及排序。某些模式会改变列表（即修改原始值）。如果稍后还需要原始值，请确保先复制一份。

完整详情可以在在线文档中找到：

<https://cmake.org/cmake/help/latest/command/list.html>

可用的 list() 模式的类别包括读取、查找、修改及排序。

A.3.1. 读取

以下是可用的模式：

- list(LENGTH <list> <out>): 计算 <list> 变量中的元素数量，并将结果存储在 <out> 变量中。
- list(GET <list> <index>... <out>): 将由 <index> 索引列表指定的 <list> 元素复制到 <out> 变量中。
- list(JOIN <list> <glue> <out>): 使用 <glue> 分隔符交错 <list> 元素，并将结果字符串存储在 <out> 变量中。
- list(SUBLIST <list> <begin> <length> <out>): 其作用类似于 GET 模式，但不是使用显式索引而是使用范围。如果 <length> 是 -1，则从 <begin> 索引到 <list> 变量中的列表末尾的所有元素都会被返回。

A.3.2. 查找

此模式仅仅是在 <list> 变量中查找 <needle> 元素的索引，并将结果存储在 <out> 变量中（如果没有找到该元素，则存储 -1）：

```
1 list(FIND <list> <needle> <out>)
```

A.3.3. 修改

以下是可用的模式：

- 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>...]): 其作用类似于 APPEND，但将元素添加到 <list> 变量的开头。

- `list(FILTER <list> {INCLUDE | EXCLUDE} REGEX <pattern>)`: 根据 `<pattern>` 值过滤 `<list>` 变量，以 INCLUDE 或 EXCLUDE 匹配的元素。
- `list(INSERT <list> <index> [<element>...])`: 在给定的 `<index>` 位置将一个或多个 `<element>` 值添加到 `<list>` 变量中。
- `list(POP_BACK <list> [<out>...])`: 从 `<list>` 变量的末尾移除一个元素，并将它存储在可选的 `<out>` 变量中。如果提供了多个 `<out>` 变量，则会移除更多元素来填充它们。
- `list(POP_FRONT <list> [<out>...])`: 其作用类似于 `POP_BACK`，但从 `<list>` 变量的开头移除元素。
- `list(REMOVE_ITEM <list> <value>...)`: 是 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>` 的元素。
- `list(REMOVE_DUPLICATES <list>)`: 从 `<list>` 中移除重复项。
- `list(TRANSFORM <list> <action> [<selector>] [OUTPUT_VARIABLE <out>])`: 对 `<list>` 元素应用特定的转换。默认情况下，操作应用于所有元素，但可以通过添加 `<selector>` 来限制其效果。除非提供 OUTPUT_VARIABLE 关键词，结果将存储在 `<out>` 变量中，否则将修改提供的列表（就地更改）。

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，以及 REPLACE `<pattern> <expression>`。它们的工作方式与具有相同名称的 `string()` 模式完全相同。

A.3.4. 排序

以下是可用的模式：

- `list(VERSE <list>)`: 简单地反转 `<list>` 的顺序。
- `list(SORT <list>)`: 按字母顺序对列表进行排序。

有关更高级选项的参考，请参阅在线手册。

17.4. file()

此命令提供了与文件相关的各种操作：读取、传输、锁定及归档。还提供了用于检查文件系统和对路径字符串进行操作的模式。

完整详情可以在在线文档中找到：

<https://cmake.org/cmake/help/latest/command/file.html>

可用的 `file()` 模式的类别包括读取、写入、文件系统、路径转换、传输、锁定及归档。

A.4.1. 读取

以下是可用的模式：

- `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，并遵循最大字节数 `<max>` 的限制。HEX 标志指明输出应转换为十六进制表示。
- `file(STRINGS <filename> <out>)`: 从 `<filename>` 中读取字符串到 `<out>` 变量。
- `file(<hashing-algorithm> <filename> <out>)`: 计算 `<filename>` 文件的 `<hashing-algorithm>` 哈希值，并将结果存储在 `<out>` 变量中。可用的算法与 `string()` 哈希函数相同。
- `file(TIMESTAMP <filename> <out> [<format>])`: 生成 `<filename>` 文件的时间戳字符串表示，并将其存储在 `<out>` 变量中。可选地接受一个 `<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，只应在 `install(CODE)` 或 `install(SCRIPT)` 场景中使用。自 CMake 3.21 开始可用。

A.4.2. 写入

以下是可用的模式：

- `file({WRITE | APPEND} <filename> <content>...)`: 将所有 `<content>` 参数写入或追加到 `<filename>` 文件中。如果提供的系统路径不存在，将会递归创建。
- `file({TOUCH | TOUCH_NOCREATE} [<filename>...])`: 更新 `<filename>` 的时间戳。如果文件不存在，则仅在 `TOUCH` 模式下创建。
- `file(GENERATE OUTPUT <output-file> [...])`: 这是一种高级模式，为当前 CMake 生成器的每个构建配置生成输出文件。
- `file(CONFIGURE OUTPUT <output-file> CONTENT <content> [...])` : 其作用类似于 `GENERATE_OUTPUT`，但同时通过替换变量占位符来配置生成的文件。

A.4.3. 文件系统

以下是可用的模式：

- `file({GLOB | GLOB_RECURSE} <out> [...] [<globbing-expression>...])`: 生成匹配 `<globbing-expression>` 的文件列表，并将其存储在 `<out>` 变量中。`GLOB_RECURSE` 模式也会扫描嵌套目录。
- `file(RENAME <oldname> <newname>)`: 将文件从 `<oldname>` 移动到 `<newname>`。
- `file({REMOVE | REMOVE_RECURSE} [<files>...])`: 删除 `<files>`。`REMOVE_RECURSE` 也会删除目录。
- `file(MAKE_DIRECTORY [<dir>...])`: 创建目录。

- `file(COPY <file>... DESTINATION <dir> [...])`: 将文件复制到 `<dir>` 目的地。它提供了过滤、设置权限、跟随符号链接链等选项。
- `file(COPY_FILE <file> <destination> [...])`: 将单个文件复制到 `<destination>` 路径。自 CMake 3.21 开始可用。
- `file(SIZE <filename> <out>)`: 读取 `<filename>` 的大小 (以字节为单位)，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>` 创建指向 `<original>` 的符号链接。
- `file({CHMOD|CHMOD_RECURSE} <files>... <directories>... PERMISSIONS <permissions>... [...])`: 设置文件和目录的权限。
- `file(GET_RUNTIME_DEPENDENCIES [...])`: 收集各种类型文件的运行时依赖项：可执行文件、库和模块。与 `install(RUNTIME_DEPENDENCY_SET)` 结合使用。

A.4.4. 路径转换

以下是可用的模式：

- `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>` 基础目录。自 CMake 3.19 开始可用。
- `file(RELATIVE_PATH <out> <directory> <file>)`: 计算 `<file>` 相对于 `<directory>` 的路径，并将其存储在 `<out>` 变量中。
- `file({T0_CMAKE_PATH | T0_NATIVE_PATH} <path> <out>)`: 将 `<path>` 转换为 CMake 路径 (目录以正斜杠分隔) 到平台原生路径及反之。结果存储在 `<out>` 变量中。

A.4.5. 传输

以下是可用的模式：

- `file(DOWNLOAD <url> [<path>] [...])`: 从 `<url>` 下载文件，并将其存储在 `<path>` 中。
- `file(UPLOAD <file> <url> [...])`: 将 `<file>` 上传到 URL。

A.4.6. 锁定

锁定模式在 `<path>` 资源上放置了一个锁

```
file(LOCK <path> [DIRECTORY] [RELEASE]
    [GUARD <FUNCTION|FILE|PROCESS>]
```

```
[RESULT_VARIABLE <out>] [TIMEOUT <seconds>]
```

```
)
```

此锁可选地限定在 FUNCTION、FILE 或 PROCESS，并且可以用 <seconds> 的超时时间限制。为了释放锁，提供 RELEASE 关键词。结果将存储在 <out> 变量中。

A.4.7. 归档

归档创建提供以下签名：

```
file(ARCHIVE_CREATE OUTPUT <destination> PATHS <source>...
  [FORMAT <format>]
  [COMPRESSION <type> [COMPRESSION_LEVEL <level>]]
  [MTIME <mtime>] [VERBOSE]
)
```

它在 <destination> 路径创建一个归档文件，其中包含 <source> 文件之一种支持的格式：7zip、gnutar、pax、paxr、raw 或 zip（默认为 paxr）。如果选定的格式支持压缩级别，则可以提供一个单数字整数 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]
)
```

从 <archive> 中提取匹配可选 <patterns> 值的文件到目的地 <dir>。如果提供了 LIST_ONLY 关键词，则文件不会提取，而只会列出。

17.5. math()

CMake 还支持一些简单的算术运算。请参阅在线文档获取完整详情：

<https://cmake.org/cmake/help/latest/command/math.html>

要计算一个数学表达式并将结果以字符串形式存储在 <out> 变量中，可选地使用 <format> (HEXADECIMAL 或 DECIMAL) 格式化输出，可以使用以下签名：

```
1 math(EXPR <out> "<expression>" [OUTPUT_FORMAT <format>])
```

<expression> 的值是一个字符串，支持 C 代码中存在的运算符（它们在这里具有相同的意义）：

- 算术运算符：+，-，*，/ 和% 模运算
- 位运算符：| 或运算，& 与运算，^ 异或运算，~ 非运算，<< 左移和 >> 右移
- 括号 (...)

常数值可以以十进制或十六进制格式提供。