



# CMake Best Practices

Discover proven techniques for creating and maintaining programming projects with CMake

(使用 CMake 创建和维护项目)

作者: Dominik Berner, Mustafa Kemal Gilor

译者: 陈晓伟

## 本书概述

CMake 是一个强大的工具，用于执行各种各样的任务，需要一个起点对 CMake 进行学习。本书更专注于常见的任务，边实践边学习 CMake。CMake 文档很全面，但缺少具有代表性的例子，说明如何将源码组合在一起，特别是互联网上还有有很奇淫技巧和过时的解决方案。本书的重点是帮助读者把需要做的事情串在一起，编写 CMake，从而创建简洁和可维护的项目。

阅读本书后，不仅可以掌握 CMake 的基础知识，还可以通过构建大型复杂项目和在任何编程环境中运行的构建示例。还可以使用集成和自动化工具，以提高整体软件的质量，例如：测试框架、模糊测试和自动生成文档。编写代码只是工作的一半，本书还会引导读者对程序进行安装、打包和分发，这些都是为使用现代化的 CI/CD 工作流程需要的基础功能。

阅读完本书，将能以最好的方式使用 CMake 建立和维护复杂的软件项目。

## 关键特性

- 理解 CMake 是什么，如何工作和交互
- 了解如何正确地创建和维护结构 CMake 项目
- 探索工具和技术，以最大程度的使用 CMake 管理项目

## 将会学到

- 构建结构良好的 CMake 项目
- 跨项目模块化和重用 CMake 代码
- 将各种静态分析、检测、格式化和文档生成工具集成到 CMake 项目中
- 尝试进行跨平台构建
- 了解如何使用不同的工具链
- 为项目构建一个定义良好，且可移植的构建环境

## 作者简介

**Dominik Berner** 是一位拥有 20 年专业软件开发经验的软件工程师、博客作者和演讲者。主要使用 C++，还参与了许多软件项目，从为初创公司的外科手术模拟器编写软件，到为 MedTech 行业的大型企业维护大型平台，再到为介于两者之间的公司创建物联网解决方案。他相信，良好的设计和可维护的构建环境是使团队高效编写软件，并创建高质量软件的关键因素之一。当他不写代码时，会去博客写一些文章，或者在会议上发表一些关于软件开发的演讲。

**Mustafa Kemal Gilor** 是一位经验丰富的专业人员，从事电信、国防工业和开源软件的性能关键软件开发。他的专长是高性能和可扩展的软件设计、网络技术、DevOps 和软件架构。他对计算机的兴趣在童年时期就显现出来了。他在 12 岁左右学习编程破解 MMORPG 游戏，从那时起他就一直在编写软件。他最喜欢的编程语言是 C++，并且喜欢做框架设计和系统编程，也是 CMake 的坚定倡导者。他的职业生涯中，维护了许多代码库，并将许多遗留（非 CMake）项目移植到 CMake。

## 审评者介绍

**Richard Von Lehe** 住在明尼苏达州的双城地区。过去的几年中，他在软件项目中经常使用 CMake，包括正畸建模、建筑控制、无人机避免碰撞和专用打印机。闲暇时，他喜欢和家人，以及宠物兔 Gus 一起放松，还喜欢骑自行车和弹吉他。

**Toni Solarin-Solada** 是一名软件工程师，专门设计抽象底层操作系统服务的跨平台编程库。

## 本书相关

- Github 地址：<https://github.com/xiaoweichen/CMake-Best-Practices>

# 前言

软件世界每天都在发展，CMake 也不例外。经过 20 多年的不断发展，现在 CMake 是构建 C++ 应用程序的行业标准。虽然 CMake 具有非常丰富的功能，但其文档中的例子和指南实在是太少了。而这，恰恰就是 CMake 实践的最佳切入点。

本书不会去解释 CMake 的细节和特性，而会通过例子来说明在构建软件时，如何使用 CMake 完成各种任务（而不是覆盖每种边缘情况）。本书的目的是让事情尽可能简单，同时推荐的最佳实践，从而使读者只需要知道自己需要的 CMake 功能即可。

我们将先解释概念，然后用例子来说明，通过实践进行学习，并且可将这些知识应用到日常工作中。估计本书的读者大多是工程师，所以我们对书的内容进行了相应的调整。写这本书的时候，我们首先扮演的角色是软件工程师，然后才是作者。因此，本书的内容会更实用，而不仅停留在理论层面，从而证明了相应的技术可以在日常工作中使用。

从工程师中来，再回到工程师中去。希望各位能享受这本书。

## 适宜读者

这本书针对软件工程师和构建系统维护人员，他们经常使用 C 或 C++，从而可以使用 CMake 来减轻日常任务的重担。基本的 C++ 和编程知识有助于读者更好地理解书中的例子。

## 本书内容

第 1 章，启用 *CMake*，介绍 CMake，安装 CMake 和使用 CMake 尝试进行构建，从而可以了解如何手动安装 CMake 最新的稳定版本（即使包管理器没有提供）。还介绍了 CMake 的基本概念，以及为什么 CMake 是构建系统生成器，而不是构建系统，以及了解 CMake 如何与现代 C++（和 C）一起工作。

第 2 章，*CMake* 的最佳使用方式，了解如何在 GUI 和命令行中使用 CMake，以及 CMake 如何与一些常见的 IDE 和编辑器集成。

第 3 章，创建 *CMake* 项目，通过项目来构建可执行文件和库，并将两者连接在一起。

第 4 章，打包、部署和安装 *CMake* 项目，了解如何创建软件项目的可分发版本。可以了解如何添加安装说明和使用 CPack(CMake 的打包程序) 打包项目。

第 5 章，集成第三方库和依赖管理，了解如何将现有的第三方库集成到项目中。还可以了解如何添加已经安装在系统上的库、外部 CMake 项目和非 CMake 项目。

第 6 章，自动生成文档，探索如何从代码中生成文档，doxygen、dot(graphviz) 和 plantuml 可以作为构建过程的一部分。

第 7 章，集成代码质量工具，了解如何将单元测试、代码消毒器、静态代码分析和代码覆盖率工具集成到项目中，并展示 CMake 如何发现和执行测试。

第 8 章，执行自定义任务，了解如何将工具集成到构建过程中，将外部程序包放置到自定义目标中，或将它们挂载到构建过程中进行执行。这里将介绍如何使用自定义任务生成文件，以及如何使用其他目标生成的文件。还将了解如何在 CMake 构建配置期间执行系统命令，以及如何使用 CMake 脚本模式创建与平台无关的命令。

第 9 章，创建可复制的构建环境，了解在各种机器（包括 CI/CD 管道）之间如何构建可移植的环境，以及如何使用 Docker、sysroot 和 CMake 预置来让构建在任何地方都能“开箱即用”。

第 10 章，处理大项目和分布式存储库，使用 CMake 简化了跨多个 git 库的项目管理。将了解如何创建一个超级构建，该超级构建允许构建特定版本，以及最新的夜间构建。我们将探索超级构建需要什么先决条件，以及如何使用。

第 11 章，自动化模糊测试，了解如何在 CMake 中集成和使用模糊测试工具。

第 12 章，跨平台编译和自定义工具链，演示了如何使用跨平台工具链。还将了解如何编写自己的工具链定义，并在 CMake 中使用不同的工具链。

第 13 章，重用 *CMake* 代码，了解如何使用 CMake 模块，以及如何泛化 CMake 文件。还将了解如何编写可通用的模块，这些模块可以单独提供。

第 14 章，优化和维护 *CMake* 项目，建议如何获得更快的构建时间，并提供了一些技巧和技巧，以保持 CMake 项目长时间内的整洁。

第 15 章，迁移到 *CMake*，了解如何在不停止开发的情况下，将现有的大型代码库迁移到 CMake 的高级策略。

第 16 章，对 *CMake* 进行贡献，若想进行贡献，去哪里、需要什么，以及基本的贡献指南。还将指引您在哪里找到更多相信的信息或更具体的文献。

## 编译环境

将 CMake 3.21 或更新版本，以及安装的至少支持 C++14 的现代 C++ 编译器。一些例子可能需要额外的软件来运行，在相关章节中会提到。示例中使用的所有软件都是开源的，可以免费获得。

书中涉及的软件/硬件	操作系统
CMake 3.13.4 或更高版本	Linux, Windows 或 macOS
GCC、Clang 或 MSVC	Linux, Windows 或 macOS
Git	Linux, Windows 或 macOS

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

## 下载示例

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

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

## 下载彩图

我们还提供了一个 PDF 文件，其中有本书中使用的屏幕截图和图表的彩图。下载地址：[https://static.packt-cdn.com/downloads/9781803239729\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781803239729_ColorImages.pdf)



# 目录

# 第一部分：基础知识

第 1 章中，将了解如何使用 CMake，了解其基本概念，以及对 CMake 的简要介绍。

第 2 章中，将介绍如何在命令行、GUI 或各种 IDE 和编辑器中使用 CMake，展示如何更改各种配置选项和选择不同的编译器。

第 3 章中，将介绍如何创建一个简单的 CMake 项目，构建可执行文件和库。

本节包括以下几章：

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

# 第 1 章 启用 CMake

使用 C++ 和 C 的开发者可能听说过 CMake。过去的 20 年，CMake 已经发展成为构建 C++ 应用的行业标准。但 CMake 不仅是构建系统——还是一个构建系统生成器，可以为其他构建系统（如 Makefile、Ninja、Visual Studio、Qt Creator、Android Studio 和 Xcode）生成相应的工程。CMake 不仅限于构建软件——还支持安装、打包和测试。

作为行业标准，CMake 是 C++ 开发者必须了解的。

本章中，将对 CMake 进行大概的介绍，并学习构建第一个程序所需的基础知识。了解 CMake 的构建过程，以及如何使用 CMake 来配置构建。

本章中，将讨论以下主题：

- 简介 CMake
- 安装 CMake
- CMake 的构建过程
- 书写 CMake 文件
- 不同工具链和构建配置

## 1.1. 相关准备

运行本章中的例子，需要支持 C++17 的编译器。这些示例不复杂，不需要使用新标准的功能。

建议使用下列编译器来运行示例：

- **Linux:** GCC 9 或更高版本，Clang 10 或更高版本
- **Windows:** MSVC 19 或更高版本，MinGW 9.0.0 或更高版本
- **macOS:** Apple Clang 10 或更高版本

### Note

我们提供了现成的 Docker 容器，里面配置好了环境，可以用来运行本书的示例。

可以在这里找到这个容器镜像<https://github.com/PacktPublishing/CMake-Best-Practices>。

## 1.2. 简介 CMake

CMake 开源，并且可以在许多平台上使用，还与编译器无关，从而成为构建和分发跨平台软件的强大工具。这些特性使它成为对软件极具价值的工具——可以自动化构建，以及内置的代码质量检验。

CMake 由三个命令行工具组成：

- `cmake`: CMake 本身，用于生成构建指令
- `ctest`: CMake 的测试程序，用于检测和运行测试
- `cpack`: CMake 的打包工具，将软件打包成方便的安装程序，如 `deb`、`RPM` 和自解压缩的安装程序

还有两种交互工具:

- `cmake-gui`: 配置项目的图形界面
- `ccmake`: 配置 CMake 的交互式终端界面

`cmake-gui` 可以方便地配置 CMake 构建，并选择要使用的编译器:

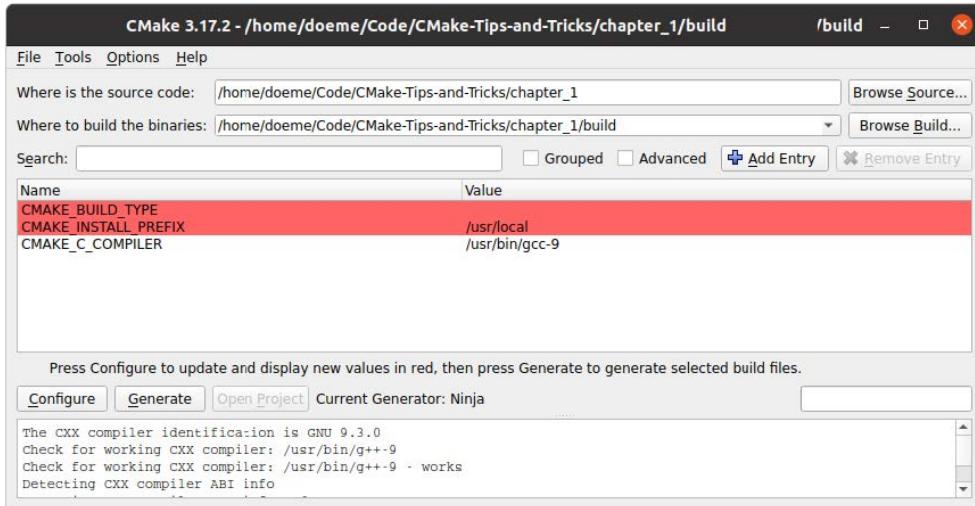


图 1.1 配置项目后的 `cmake-gui`

若使用控制终端工作，但希望可以交互式的进行 CMake 配置，那么 `ccmake` 是不错的选择。虽然不像 `cmake-gui` 那样方便，但提供了相同的功能，可以通过 ssh shell 或类似的远程对 CMake 进行配置:

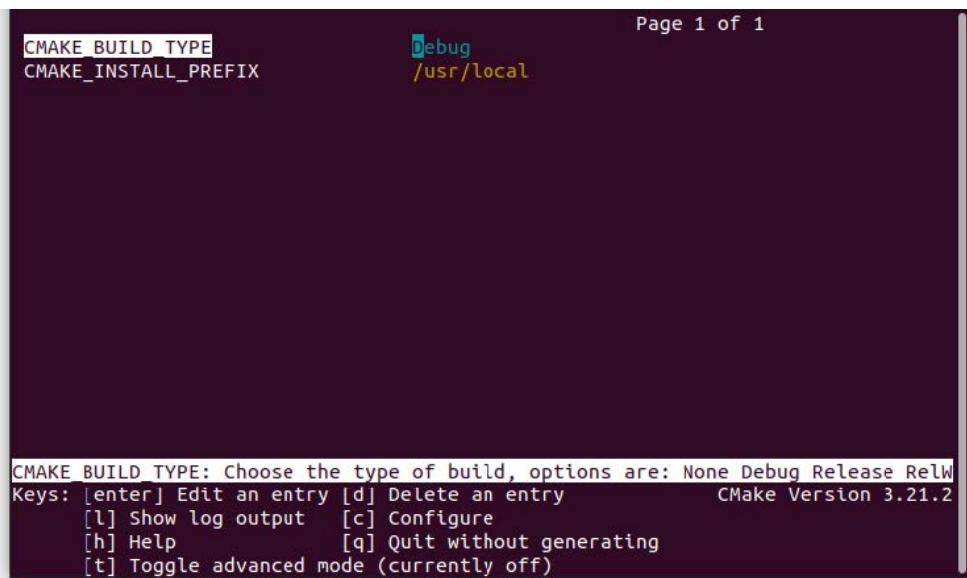


图 1.2 使用 `ccmake` 配置项目

与常规模构建系统相比，CMake 有很多优势。首先是跨平台方面，可以更容易地为各种编译器和平台创建构建指令，而不需要深入了解各自构建系统的细节。

然后，CMake 能够发现系统库和依赖关系，这大大减轻了查找正确库的烦恼。还有就是 CMake 与包管理器（如 Conan 和 vcpkg）集成得很好。

其不仅能够为多个平台构建软件，而且对测试、安装和打包的原生支持，使 CMake 成为构建软件的有力候选者。其能够完成从构建、测试到打包的一系列工作，对于长期维护项目有极大的帮助。

其实，CMake 本身对系统的依赖很少，并且可以在命令行上运行，这使得它非常适合成为 CI/CD 流水中的自动化构建系统。

已经介绍了 CMake 可以做什么，接下来来安装 CMake。

## 1.3. 安装 CMake

CMake 可以从<https://cmake.org/download/>免费下载。既可以下载预编译的二进制文件，也可以下载源代码。通常，预编译的二进制文件就足够使用了，但由于 CMake 本身的依赖很少，构建一个版本也是可能的。

Linux 的主要发行版都在其包存储库上提供 CMake。虽然 CMake 的预打包版本通常不是最新版本，若系统定期更新，通常也够用。

### Note

本书中示例使用的 CMake 的最低版本是 3.21，建议您手动下载相应的 CMake 版本。

### 1.3.1 使用源代码构建 CMake

CMake 使用 C++ 编写的，并使用 Make 来构建。可以从源码构建 CMake，但通常下载的二进制预编译文件就可以了。

从<https://cmake.org/download/>下载源包后，解压到文件夹，并运行以下命令：

```
./configure make
```

若想构建 cmake-gui，请配置--qt-gui 选项，这需要安装 Qt。配置需要一段时间，成功后可以使用以下命令安装 CMake：

```
make install
```

使用命令测试安装是否成功。

```
cmake --version
```

打印出 CMake 的版本：

```
cmake version 3.21.2
CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

## 1.4. 构建第一个工程

现在，试试安装的 CMake 是否有效。我们提供了一个 `hello world` 示例，可以立即下载，并进行构建。打开一个控制台，输入以下内容：

```
git clone https://github.com/PacktPublishing/CMake-Best-Practices.git
cd CMake-Best-Practices/chapter01
mkdir build
cd build
cmake ..
cmake --build .
```

这将产生名为 `chapter1` 的可执行文件，在控制台上输出 `Welcome to CMake Best Practices.`

详细看看这里发生了什么：

1. 首先，使用 Git 签出示例存储库，然后创建构建文件夹。示例 CMake 项目的文件结构在构建前是这样的：

```
.
├── CMakeLists.txt
└── build
    └── src
        └── main.cpp
```

除了源代码的文件夹外，还有一个名为 `CMakeLists.txt` 的文件。这个文件包含 CMake 关于如何创建项目的构建说明，以及如何构建的说明。每个 CMake 项目在项目的根目录下都有一个 `CMakeLists.txt` 文件，但是在不同的子文件夹中可能有许多同名的文件。

2. 下载存储库之后，使用 `cmake` 启动构建过程。CMake 的构建过程分为两个阶段。第一步称为配置，读取 `CMakeLists.txt` 文件，并为系统的本地构建工具链生成指令。第二步，执行这些构建指令，并构建可执行文件或库。

配置步骤中，将检查构建需求，解析依赖关系，并生成构建指令。

3. 项目还会创建一个名为 `CMakeCache.txt` 的文件，其中包含创建构建指令所需的信息。`cmake --build` 的下一个阶段，通过内部调用 CMake 来执行构建；若在 Windows 上，则通过调用

Visual Studio 编译器来实现，这是编译二进制文件的实际方式。若一切顺利，在构建文件夹中会有一个名为 chapter1 的可执行文件。

简单起见，前面的示例中进入构建目录，并使用相对路径来查找源文件夹。这通常很方便，但想从其他地方使用 CMake，可以使用--S 选项来选择源文件，以及使用--B 选项来指定构建文件夹：

```
cmake -S /path/to/source -B /path/to/build  
cmake --build /path/to/build
```

在持续集成环境中使用 CMake 时，显式传递构建目录和源目录通常很方便，也助于提高可维护性。若想为不同的配置创建不同的构建目录，例如在构建跨平台软件时，这也会很有帮助。

#### 1.4.1 简单的 CMakeLists.txt 示例

对于非常简单的 hello world 例子，CMakeLists.txt 文件只包含几行指令：

```
cmake_minimum_required(VERSION 3.21)  
project(  
    ch1_simple_executable  
    VERSION 1.0  
    DESCRIPTION "A simple C++ project to demonstrate basic CMake usage"  
    LANGUAGES CXX)  
  
add_executable(chapter1)  
target_sources(chapter1 PRIVATE src/main.cpp)
```

更详细地说明一下：

- 第一行定义了构建这个项目所需的 CMake 的最低版本。每个 CMakeLists.txt 文件都以这个指令开始。这是用来提示用户，若项目使用了 CMake 的特性，可能只有在某个版本以上才可用。通常，建议将版本设置为支持项目中使用特性的最低版本。
- 下一行是要构建的项目的名称、版本和描述，后面是项目中使用的编程语言。这里，使用 CXX 将其标记为一个 C++ 项目。
- add\_executable 告诉 CMake 要构建一个可执行文件（与库或自定义目标不同）。
- target\_sources 告诉 CMake 在哪里查找名为 chapter1 的可执行文件的源文件，并且源文件的可见性仅限于可执行文件。

祝贺你——现在可以用 CMake 创建程序了。为了理解这些指令背后发生了什么，需要继续了解一下 CMake 的构建过程。

## 1.5. 了解 CMake 的构建过程

CMake 的构建过程分为两步，如下图所示。首先，若使用时没有特殊标志，CMake 会在配置过程中扫描系统中可用的工具链，再决定输出。第二步是调用 `cmake --build` 时的实际编译和构建：

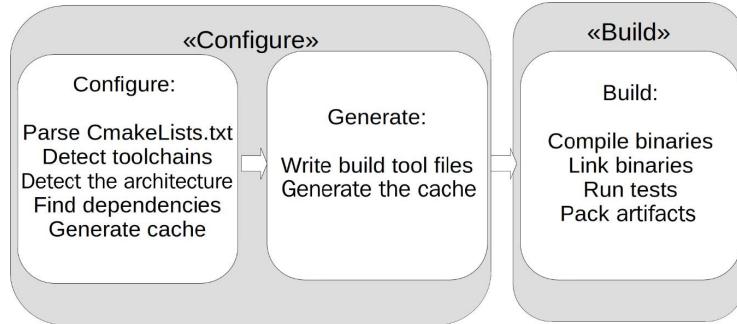


图 1.3 CMake 的两阶段构建过程

标准输出是 Unix Makefiles，除非检测到唯一的编译器是 Microsoft Visual Studio，这种情况下将创建 Visual Studio 解决方案 (.sln)。

修改生成器，可以使用-G 选项：

```
cmake .. -G Ninja
```

这将生成 Ninja(<https://ninja-build.org/>) 的构建文件。CMake 支持很多生成器，Ninja 是其中之一。在 CMake 网站上可以找到一个生成器支持列表：<https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html>。

生成器主要有两种类型——一种是 Makefile 风格的生成器和 Ninja 生成器，多以命令行方式使用，另一种是为 IDE(如 Visual Studio 或 Xcode) 创建构建工程文件。

这里，CMake 会区分单配置生成器和多配置生成器。对于单配置生成器，必须在每次更改配置时重写构建文件；多配置构建系统可以自行管理不同的配置，不需要重新生成。尽管本书中的示例使用的是单配置生成器，但它们也适用于多配置生成器。对于大多数示例，所选生成器是无关紧要的(否则，就会特别说明)：

生成器	多配置
Makefile(所有版本)	No
Ninja	No
Ninja Multi-Config	Yes
Xcode	Yes
Visual Studio	Yes

此外，还有一些生成器，可以为编辑器或 IDE 生成项目信息，如 Sublime Text 2、Kate editor、Code::Blocks 和 Eclipse。对于每个应用，可以选择编辑器或 IDE 使用 Make，还是 Ninja 进行内部构建。

构建完成后，CMake 将在构建文件夹中多了许多文件，其中最重要的就是 CMakeCache.txt，其存储了所有检测到的配置。注意，当使用 cmake-gui 时，第一步会分为配置项目和生成构建文件。但当以命令行运行时，这些步骤合并为一个。配置完成后，将在构建文件夹执行构建。

### 1.5.1 源文件夹和构建文件夹

CMake 中存在两个逻辑文件夹。一个是源文件夹，包含项目的层次结构集；另一个是构建文件夹，包含构建指令、缓存，以及所有生成的二进制文件和工件。

源文件夹是 CMakeLists.txt 文件所在的位置。构建文件夹可以放在源文件夹中，也可以将其放在另一个位置。两种方式都可以；注意，对于本书中的示例，我们决定将构建文件夹放在源文件夹中。构建文件夹通常命名为 build，但也可以使用其他名称，包括不同平台的前缀和后缀。当在源代码树中使用构建文件夹时，最好将其添加到 .gitignore 中。

当配置 CMake 项目时，将在构建文件夹中重新创建源文件夹的项目和文件夹结构，以便所有构建工件都位于相同的位置。每个文件夹中，都有一个名为 CMakeFiles 的子文件夹，其中包含 CMake 配置步骤生成的信息。

下面显示了一个 CMake 项目的目录结构：

```
.  
├── simple_executable  
│   ├── CMakeLists.txt  
│   └── src  
│       └── main.cpp  
└── CMakeLists.txt
```

当执行 CMake 配置时，CMake 项目的文件结构会映射到 build 文件夹中。每个包含 CMakeLists.txt 文件的文件夹将进行映射，将创建一个名为 CMakeFiles 的子文件夹，其中包含用于构建的信息：

```
└── build  
    ├── simple_executable  
    │   └── CMakeFiles  
    └── CMakeFiles
```

我们已经使用已有项目来学习了 CMake 构建过程。了解了配置和构建步骤以及生成器，我们需要 CMakeLists.txt 文件来传递必要的信息给 CMake。接下来，了解一下 CMakeLists.txt 文件怎么写，以及 CMake 语言如何工作。

## 1.6. 书写 CMake 文件

写 CMake 时，有几个主要概念和语言特性需要知道。我们不会在这里详述语言的种种细节，若需要详细了解可以参考 CMake 的官方文档。下面的小节中，将简介主要概念和语言特性。后续章节将深入探讨其中细节。

完整文档地址<https://cmake.org/cmake/help/latest/manual/cmake-language.7.html>。

### 1.6.1 CMake 语言——鸟瞰全景

CMake 使用 CMakeLists.txt 文件的配置文件来确定构建规范。这些文件用脚本语言编写，通常也称为 CMake。语言本身很简单，支持变量、字符串函数、宏、函数定义和导入其他 CMake 文件。

除了列表，不支持数据结构，如结构或类。但若操作得当，这种简单性使得 CMake 项目具有更好的可维护性。

语法基于关键字和以空格分隔的参数。例如，下面的指令会将相应的文件添加到库中：

```
target_sources(MyLibrary
    PUBLIC include/api.h
    PRIVATE src/internals.cpp src/foo.cpp)
```

PUBLIC 和 PRIVATE 关键字表示文件链接到这个库时的可见性，并充当文件列表之间的分隔符。

此外，CMake 语言支持“生成器表达式”，可以在生成构建系统时进行，通常用于为每个构建配置指定特殊信息。

### 工程

CMake 会将各种构建工件(如库、可执行文件、测试和文档)组织到项目中。虽然不同项目可以相互封装，但这里需要一个根项目。每个 CMakeLists.txt 文件对应一个项目，所以每个项目必须在源目录中有一个单独的文件夹。

项目描述如下：

```
project(
    ch1_simple_executable
    VERSION 1.0
    DESCRIPTION "A simple C++ project to demonstrate basic CMake usage"
    LANGUAGES CXX
)
```

正在解析的当前项目存储在 PROJECT\_NAME 变量中。根项目存储在 CMAKE\_PROJECT\_NAME 中，这对于确定一个项目是独立的还是封装在另一个项目中很有用。从 3.21 版本开始，还有一个 PROJECT\_IS\_TOP\_LEVEL 变量来直接确定当前项目是否是顶层项目。此外，使用 <PROJECT-NAME>\_IS\_TOP\_LEVEL，可以检测特定的项目是否为顶层项目。

下面是关于项目的其他内置变量，可以在根项目的值前加上 CMAKE\_ 前缀。若没有在 `project()` 指令中定义，则字符串为空：

- `PROJECT_DESCRIPTION`: 项目的描述字符串
- `PROJECT_HOMEPAGE_URL`: 项目的 URL 字符串
- `PROJECT_VERSION`: 项目的完整版本信息
- `PROJECT_VERSION_MAJOR`: 版本字符串的第一个数字
- `PROJECT_VERSION_MINOR`: 版本字符串的第二个数字
- `PROJECT_VERSION_PATCH`: 版本字符串的第三个数字
- `PROJECT_VERSION_TWEAK`: 版本字符串的第四个数字

每个项目都有一个源目录和二进制目录，假设下面的例子中的每个 `CMakeLists.txt` 文件都定义了一个项目：

```
.  
└── CMakeLists.txt #defines project("CMakeBestPractices")...  
└── simple_executable  
    └── CMakeLists.txt # defines project("Chapter 1")...
```

当解析根目录下的 `CMakeLists.txt` 文件时，`PROJECT_NAME` 和 `CMAKE_PROJECT_NAME` 都将是 `CMakeBestPractices`。当解析 `chapter01/CMakeLists.txt` 时，`PROJECT_NAME` 变量将变为“`chapter1`”，但 `CMAKE_PROJECT_NAME` 还是 `CMakeBestPractices`，并设置在根文件夹中。

尽管项目可以嵌套，但最好以独立的方式编写。虽然它们可能依赖于文件层次结构中较低的其他项目，但应该没有必要将一个项目作为另一个项目的子项目的必要。可以在同一个 `CMakeLists.txt` 文件中使用多个 `project()`，但并不推荐这样做，其会使项目混乱，难以维护。通常，最好为每个项目创建一个 `CMakeLists.txt` 文件，并用子文件夹组织结构。

本书的 GitHub 库，包含了本书中的例子，以分层的方式组织，其中每一章都是一个单独的项目，可能包含更多的项目，用于不同的部分和示例。

虽然每个示例都可以单独构建，但也可以从库的根目录构建整本书的所有示例。

## 变量

变量是 CMake 语言的核心部分。可以使用 `set` 指令设置变量，使用 `unset` 指令删除变量。变量名区分大小写。下面的例子展示了如何设置一个名为 `MYVAR` 的变量，并将其赋值为 `1234`：

```
set(MYVAR "1234")
```

要删除 `MYVAR` 变量，可以使用 `unset`：

```
unset(MYVAR)
```

一般的代码约定使用全大写命名变量。在内部，变量总是表示为字符串。

这里，可以用 `$` 符号和花括号来访问变量的值：

```
message(STATUS "The content of MYVAR are ${MYVAR}")
```

变量的引用可以嵌套，并由内而外求值：

```
${outer_${inner_variable}_variable}
```

变量的作用域可以通过以下方式确定：

- 函数作用域：在函数内部设置的变量只在函数内部可见。
- 目录作用域：源树中的每个子目录绑定变量，并包括来自父目录的变量。
- 持久缓存：缓存的变量可以是系统的，也可以是用户定义的。在多次运行中保持它们的值不变。

将 PARENT\_SCOPE 选项传递给 `set()` 会使变量在父作用域中可见。

CMake 提供了各种预定义变量，通常以 CMAKE\_ 为前缀。完整的列表地址<https://cmake.org/cmake/help/latest/manual/cmake-variables.7.html>。

## 列表

尽管 CMake 在内部将变量存储为字符串，但可以在 CMake 中使用分号分隔值来处理列表。列表可以通过传递多个未加引号的变量给 `set()`，或直接作为一个分号分隔的字符串来创建：

```
set(MYLIST abc def ghi)
set(MYLIST "abc;def;ghi")
```

使用 `list` 指令可以通过修改列表内容、重新排序或查找内容来操作列表。下面的代码将在 MYLIST 中查询 abc 值的索引，然后检索该值并将其存储在名为 ABC 的变量中：

```
list(FIND MYLIST def ABC_INDEX)
list(GET MYLIST ${ABC_INDEX} ABC)
```

要向列表追加一个值，可以使用 APPEND 关键字。这里，xyz 值会添加到 MYLIST 中：

```
list(APPEND MYLIST "xyz")
```

## 缓存变量和选项

CMake 缓存了一些变量，以便后续的构建运行得更快，变量存储在 CMakeCache.txt 文件中。通常，不必手动编辑改文件，但对其修改可以用于调试构建。

所有用于配置构建的变量都会进行缓存。要缓存一个自定义变量 ch1\_MYVAR，其值为 foo 值，可以使用 `set` 指令：

```
set(ch1_MYVAR foo CACHE STRING "Variable foo that configures bar")
```

注意，缓存变量必须有一个类型和一个文档字符串，以提供它们的快照。

大多数自动生成的缓存变量都标记为“高级”，这意味着它们在默认情况下对 `cmake-gui` 和 `cmake` 中用户是隐藏的。要使它们可见，必须显式地切换。若有额外的缓存变量是由 `CMakeLists.txt` 文件生成的，也可以通过调用 `mark_as_advanced(MYVAR)` 指令来隐藏：

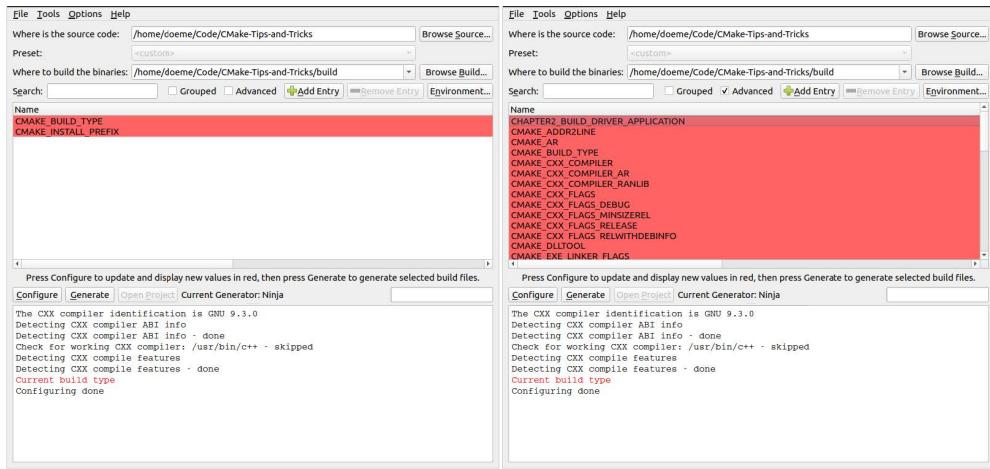


图 1.4 左边——cmake-gui 没有显示标记为“高级”的变量。右边——标记“高级”复选框将显示所有标记为高级变量

根据经验，用户经常需要变更的选项或变量很少会标记为“高级”。

对于简单的布尔型缓存变量，CMake 还提供了 option 指令，默认值为 OFF，除非另有说明。也可以通过 CMakeDependentOption 模块相互依赖：

```
option(CHAPTER1_PRINT_LANGUAGE_EXAMPLES
      "Print examples for each language" OFF)

include(CMakeDependentOption)
cmake_dependent_option(CHAPTER1_PRINT_HELLO_WORLD
                      "Print a nice greeting from chapter1" ON
                      CHAPTER1_PRINT_LANGUAGE_EXAMPLES ON)
```

选项通常是指定简单项目配置的方法，是 bool 类型的缓存变量。若已经存在与该选项同名的变量，则 option 不会执行任何操作。

## 属性

CMake 中的属性是附加到特定对象或 CMake 范围的值，如文件、目标、目录或测试用例。属性可以通过 set\_property 指令来设置。要读取属性的值，可以使用 get\_property。默认情况下，set\_property 会覆盖已经存储在属性中的值，可以通过传递 APPEND 或 APPEND\_STRING 将值添加到当前值中。

完整签名如下：

```
set_property(<Scope> <EntityName>
            [APPEND] [APPEND_STRING]
            PROPERTY <propertyName> [<values>])
```

作用域说明符可以有以下值：

- GLOBAL：影响整个构建过程的全局属性。

- DIRECTORY <dir>: 属性绑定到当前目录或 <dir> 中指定的目录，也可以直接使用 `set_directory_properties` 进行设置。
- TARGET <targets>: 特定目标的属性，也可以使用 `set_target_properties` 进行设置。
- SOURCE <files>: 将属性应用于源文件列表，也可以直接使用 `set_source_files_properties` 进行设置。此外，还有 SOURCE DIRECTORY 和 SOURCE TARGET\_DIRECTORY 扩展选项：
  - DIRECTORY <dirs>: 为目录范围内的源文件设置属性，该目录必须解析为当前目录或 `add_subdirectory` 添加的目录。
  - TARGET\_DIRECTORY <targets>: 这将属性设置为创建指定目标的目录，目标必须在设置属性前已经存在。
- INSTALL <files>: 这将设置已安装文件的属性，可以用来控制 `cpack` 的行为。
- TEST <tests>: 这将设置测试的属性，也可以直接使用 `set_test_properties` 进行设置。
- CACHE <entry>: 这将设置缓存变量的属性。最常见的方法包括将变量设置为高级或向其添加文档字符串。

支持的属性的完整列表，请查阅<https://cmake.org/cmake/help/latest/manual/cmakeproperties.7.html>。

当修改属性时，可以使用 `set_target_properties` 和 `set test_properties`，而非 `set_property`。使用显式指令可以避免属性名之间的错误和混淆，可读性会更强。还有一个 `define_property` 指令，其会创建一个不设置值的属性。我们不建议使用，因为属性总需要有一个默认值。

## 循环和条件

和其他编程语言一样，CMake 支持条件块和循环块。条件块位于 `if()`、`elseif()`、`else()` 和 `endif()` 语句之间。条件使用各种关键字表示。

一元关键字可以在值前加前缀：

```
if(DEFINED MY_VAR)
```

条件中使用的一元关键字如下：

- COMMAND: 若提供的值是命令，则为 True
- EXISTS: 检查文件或路径是否存在
- DEFINED: 若值是一个已定义的变量，则为 True

此外，还有一元文件系统的条件：

- EXISTS: 若给定的文件或目录存在，则为 True
- IS\_DIRECTORY: 检查提供的路径是否为目录
- IS\_SYMLINK: 若提供的路径是符号链接，则为 True
- IS\_ABSOLUTE: 检查提供的路径是否为绝对路径

二元测试比较两个值，并放在要比较的值之间：

```
if(MYVAR STREQUAL "FOO")
```

二元运算符如下：

- LESS, GREATER, EQUAL, LESS\_EQUAL 和 GREATER\_EQUAL: 比较数值。
- STRLESS, STREQUAL, STRGREATER, STRLESS\_EQUAL 和 STRGREATER\_EQUAL: 按字典序比较字符串。
- VERSION\_LESS, VERSION\_EQUAL, VERSION\_GREATER, VERSION\_LESS\_EQUAL 和 VERSION\_GREATER\_EQUAL: 比较版本字符串。
- MATCHES: 使用正则表达式进行匹配。
- IS\_NEWER\_THAN: 比较两个文件中哪一个最近修改过。但这不是很精确，若两个文件具有相同的时间戳，也会返回 True。还有更令人困惑的结果，若其中任何一个文件丢失，结果也为 True。

最后，就是布尔运算符 OR、AND 和 NOT。

循环可以通过 `while()` 和 `endwhile()`，或 `foreach()` 和 `endforeach()` 实现。循环可以使用 `break()` 终止，`continue()` 会中止当前的迭代并立即开始下一个迭代。

`while` 循环接受与 `if` 语句相同的条件。下面的示例只要 `MYVAR` 小于 5 就会循环。为了增加变量，我们使用 `math()` 指令：

```
set(MYVAR 0)
while(MYVAR LESS "5")
  message(STATUS "Chapter1: MYVAR is '${MYVAR}'")
  math(EXPR MYVAR "${MYVAR}+1")
endwhile()
```

除了 `while` 循环之外，CMake 还知道用于遍历列表或范围的循环：

```
foreach(ITEM IN LISTS MYLIST)
# do something with ${ITEM}
endforeach()
```

`for` 循环可以使用 `RANGE` 关键字：

```
foreach(ITEM RANGE 0 5)
# do something with ${ITEM}
endforeach()
```

`foreach()` 的 `RANGE` 版本可以只用一个结束值，不过指定开始值和结束值是一个很好的习惯。

## 函数

函数由 `function()/endfunction()` 定义。函数为变量创建了一个新的作用域，因此所有在内部定义的变量都不能从外部访问，除非将 `PARENT_SCOPE` 选项传递给 `set()`。

函数不区分大小写，通过 `function` 后的名称加上圆括号来使用函数：

```
function(foo ARG1)
# do something
endfunction()
# invoke foo with parameter bar
foo("bar")
```

函数是使 CMake 部分可重用方法，当在做大型项目时，函数会经常使用。

## 宏

CMake 宏使用 `macro()`/`endmacro()` 定义，有点像函数。不同的是，函数参数是真变量，而在宏中是字符串替换。这意味着必须使用大括号访问宏的所有参数。

另一个区别是，通过调用函数，作用区域转移到函数内部。执行宏时，就好像宏的主体粘贴到调用位置一样，宏不会创建变量和控制流的作用域。因此，避免在宏中调用 `return()`。

## 目标

CMake 的构建系统会组织一组逻辑目标，这些逻辑目标对应于可执行文件、库或自定义命令或工件，如文档或类似的文件。

CMake 中有三种主要的方法来创建目标——`add_executable`, `add_library` 和 `add_custom_target`。前两个用于创建可执行文件和静态或动态库，而第三个可以包含要执行的定制命令。

目标间可以相互依赖，因此一个目标必须在另一个目标之前建立。

在为构建配置或编译器选项设置属性时，使用目标变量是个好习惯。一些目标属性具有可见性修饰符，如 `PRIVATE`、`PUBLIC` 或 `INTERFACE`，以表示哪些需求是可传递的——哪些属性必须由依赖的目标“继承”。

## 生成器表达式

生成器表达式是在构建的配置阶段进行的语句。大多数函数允许使用生成器表达式，只有少数例外。以 `$<OPERATOR:VALUE>` 的形式使用，其中 `OPERATOR` 或直接使用，或与 `VALUE` 进行比较。这里，可以将生成器表达式看作小型的内联 if 语句。

下面的例子中，若编译器是 GCC、Clang 或 Apple Clang，则使用生成器表达式为 `my_target` 启用`-Wall` 编译器标志。注意 GCC 的 `COMPILER_ID` 标识为“GNU”：

```
target_compile_options(my_target PRIVATE
"$<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall>")
```

若 `CXX_COMPILER_ID` 变量匹配 GNU, Clang, AppleClang 列表中任意一个，则附加`-Wall` 选项到目标——也就是 `my_target`。生成器表达式在编写独立于平台和编译器的 CMake 文件时非常方便。

除了查询值，生成器表达式还可以用于转换字符串和列表：

```
$<LOWER_CASE:CMake>
```

这将输出“cmake”。

想要了解生成器表达式更多详情，可参阅<https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html>。

CMake 支持各种构建系统、编译器和链接器，所以经常用于为不同的平台构建软件。下一节中，将学习如何在 CMake 中使用不同的工具链，以及如何配置不同的构建类型，如 Debug 或 Release。

## CMake 策略

对于顶层的 CMakeLists.txt 文件的第一行，必须要有 `cmake_minimum_required`，因为其会设置用于构建项目的内部 CMake 策略。

策略用于保持跨多个 CMake 版本的向后兼容性，可以配置为使用 OLD 行为，所以 CMake 的行为需要向后兼容，或者配置为 NEW，使新策略生效。由于每个新版本都将引入新的规则和功能，因此将使用策略来警告向后兼容性问题。可以使用 `cmake_policy` 禁用或启用策略。

下面的例子中，`CMP0121` 策略设置为向后兼容的值。`CMP0121` 在 CMake 3.21 引入，用于检查 `list()` 的索引变量是否为有效格式——是否为整数：

```
cmake_minimum_required(VERSION 3.21)
cmake_policy(SET CMP0121 OLD)

list(APPEND MYLIST "abc;def;ghi")
list(GET MYLIST "any" OUT_VAR)
```

通过设置 `cmake_policy(SET CMP0121 OLD)`，启用向后兼容，并且前面的代码不会产生警告，尽管这里使用“any”（不是一个整数）索引访问 `MYLIST`。

在配置步骤中将策略设置为 NEW 将会抛出一个错误——[build] `list index: any is not a valid index.`

除非包含遗留项目，否则避免设置策略

通常，策略应该通过 `cmake_minimum_required` 来控制，而不是通过更改单个策略来控制。更改策略常用于，将历史遗留项目作为子文件夹包含在其中时。

我们已经了解了用于配置构建系统的 CMake 语言背后的基本概念，CMake 用于为不同类型的构建和语言生成构建指令。下一节中，我们将了解如何指定要使用的编译器，以及如何配置构建。

## 1.7. 不同工具链和构建配置

CMake 的强大之处在于，可以为各种编译器工具链使用相同的构建规范。工具链通常由一系列程序组成，这些程序可以编译和链接二进制文件，以及创建存档和类似的文件。

CMake 支持各种可以配置工具链的语言。本书中，我们将重点关注 C++。配置不同编程语言的工具链是通过用相应的语言标签，可以使用以下变量替换 `CXX` 部分来完成：

- C

- CXX -C++
- CUDA
- OBJC -Objective C
- OBJCXX -Objective C++
- Fortran
- HIP -针对 NVIDIA 和 AMD GPU 的 HIP C++ 运行时 API
- ISPC -基于 C 语言的 SPMD 编程语言
- ASM -汇编编译器

若一个项目没有指定语言，就假定使用了 C 和 CXX，CMake 将通过检查系统自动检测要使用的工具链。可以通过环境变量进行配置，或者在交叉编译的情况下，通过提供工具链文件进行配置。此工具链存储在缓存中，若工具链发生更改，则必须删除并重新构建缓存。若安装了多个编译器，可以通过在调用 CMake 修改环境变量，CC 对应于 C 编译器，CXX 对应于 C++ 编译器，从而指定一个非默认编译器。这里，我们使用 CXX 环境变量来覆盖 CMake 中使用的默认编译器：

```
CXX=g++-7 cmake /path/to/the/source
```

也可以通过使用-D 传递相应的 cmake 变量来覆盖 C++ 编译器，如下所示：

```
cmake -D CMAKE_CXX_COMPILER=g++-7 /path/to/source
```

两种方法都可以确保 CMake 使用 GCC 版本 7 来构建，而不是使用系统中任何默认的编译器。避免在 CMakeLists.txt 文件中设置编译器工具链，因为这与 CMake 文件应该是平台和编译器无关的范式相冲突。

默认情况下，链接器由编译器自动选择，但也可以通过将链接器可执行文件路径传递给 CMAKE\_CXX\_LINKER 变量，来指定不同的连接器。

### 1.7.1 构建类型

当构建 C++ 应用时，有各种构建类型很常见，例如包含所有调试符号的 Debug 构建和优化的 Release 构建。

CMake 本身提供了四种构建类型：

- **Debug:** 未优化，包含所有调试符号，所有的断言都是启用的。这与 GCC 和 Clang 设置 -O0 -g 是一样的。
- **Release:** 对运行速度进行优化，没有调试符号和断言禁用。通常，这是用于交付的构建类型。这与 -O3 -DNDEBUG 相同。
- **RelWithDebInfo:** 提供优化，并包括调试符号，禁用断言，这与 -O2 -g -DNDEBUG 相同。
- **MinSizeRel:** 这和 Release 是一样的，但优化了二进制大小，而不是速度，这与 -Os -DNDEBUG 相同。注意，并不是所有平台上的生成器都支持此配置。

注意，构建类型必须在配置状态期间传递，并且只与单目标生成器相关，如 CMake 或 Ninja。对于像 MSVC 这样的多目标生成器，因为构建系统本身可以构建所有构建类型，所以不能直接使用。可以创建自定义构建类型，但由于它们并不适用于每个生成器，因此不鼓励这样做。

由于 CMake 支持很多的工具链、生成器和语言，常见的问题是找到并维护这些选项的工作组合，预设会有帮助解决这个问题。

## 1.8. 使用预设置维护良好的构建配置

使用 CMake 构建软件时，常见的问题是共享环境配置来构建一个项目。通常，人们习惯于具体指定构建工件应该放在哪里，在哪个平台上使用哪个生成器，或者希望 CI 环境与本地进行相同的配置后进行构建。CMake 3.19 在 2020 年 12 月发布，所有这些信息可以存储在 CMakePresets.json 文件中，放在项目的根目录中。此外，每个用户都可以将他们配置添加到 CMakeUserPresets.json 文件中。基本预设通常置于版本控制之下，但用户预设不会签入版本系统。这两个文件遵循相同的 JSON 格式，顶层写法如下所示：

```
{  
  "version": 3,  
  "cmakeMinimumRequired": {  
    "major": 3,  
    "minor": 21,  
    "patch": 0  
  },  
  "configurePresets": [...],  
  "buildPresets": [...],  
  "testPresets": [...]  
}
```

1. 第一行 “version” :3 表示 JSON 文件的模式版本。CMake 3.21 最多支持版本 3，但预计新版本将带来该模式的新版本。
2. 接下来， cmakeMinimumRequired... 指定使用哪个版本的 CMake。尽管这是可选的，但将其放在这里并与 CMakeLists.txt 文件中指定的版本匹配是一个很好的做法。
3. 之后，可以使用 configurePresets、buildPresets 和 testPresets 添加不同构建阶段的各种预设。顾名思义， configurePresets 应用于 CMake 的构建过程的配置阶段，而其他两个用于构建和测试阶段。构建和测试预设可以继承一个或多个配置预设。若没有指定继承，将应用于前面的所有步骤。

要查看项目中配置了哪些预设值，请运行 `cmake --list-presets` 查看可用预设值的列表。要使用预设进行生成，请执行 `cmake --build --preset name`。

要查看 JSON 模式的完整规范，请参阅<https://cmake.org/cmake/help/v3.21/manual/cmake-presets.7.html>。

预设是分享构建的环境一种方式。编写本文的时候，越来越多的 IDE 和编辑器都在本地添加了

对 CMake 预设的支持，特别是在使用工具链处理交叉编译时。这里，我们只对 CMake 预设进行概述，这个主题将在第 12 章中更详细地讨论。

## 1.9. 总结

本章中，对 CMake 进行了简要概述。首先，了解了如何安装和运行一个简单的构建。然后，介绍用于编写 CMake 文件的最重要的语言特性，同时了解了 CMake 的两阶段构建过程。

现在，应该能够构建本书的 GitHub 库:<https://github.com/PacktPublishing/CMake-Best-Practices> 中提供的示例。了解了 CMake 语言的核心特性，如变量、目标和策略。我们简要介绍了函数和宏，以及用于流控制的条件语句和循环。随着您继续阅读本书，将使用到所学到的知识，来进一步的了解其他良好实践和技术，从而从简单的单目标项目转移到通过良好的 CMake 设置保持复杂软件项目的可维护性。

下一章中，将了解如何在 CMake 中执行一些最常见的任务，以及 CMake 如何与各种 IDE 一起工作。

## 1.10. 扩展阅读

要了解本章中主题的更多信息，请查看以下连接：

- CMake 官方文档: <https://cmake.org/cmake/help/latest/>
- CMake 官方教程: <https://cmake.org/cmake/help/latest/guide/tutorial/index.html>

## 1.11. 练习题

回答以下问题来测试对本章的理解：

1. 如何开始 CMake 的配置步骤？
2. 如何开始 CMake 的构建步骤？
3. CMake 中的哪个可执行文件用于运行测试？
4. CMake 的哪个可执行文件用于打包？
5. CMake 中“目标”的作用？
6. 属性和变量之间的区别是什么？
7. CMake 预设的作用是什么？

# 第 2 章 CMake 的最佳使用方式

上一章中，了解了 CMake 的基本概念。现在，我们将学习如何与 CMake 交互。首先，了解一下如何配置、构建和安装现有项目。

本章将探讨 CMake 提供的接口，以及一些流行的 IDE 和编辑器集成。本章将涉及以下内容：

- 通过命令行使用 CMake
- 使用 cmake-gui 和 ccmake
- IDE 和编辑器集成 (Visual Studio, Visual Studio Code 和 Qt Creator)

## 2.1. 相关准备

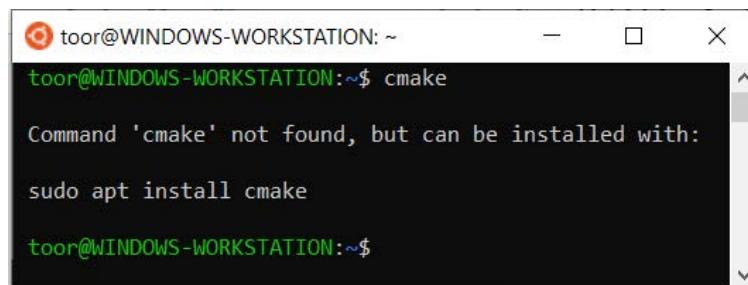
深入细节之前，下面的例子需要满足一些要求：

- **CMake-Best-Practices** 的 Git 库：这是包含该书所有示范内容的主要存储库，可以在网上找到<https://github.com/PacktPublishing/CMake-Best-Practices>。

## 2.2. 通过命令行使用 CMake

使用 CMake 最常见的方式是通过命令行 (CLI)。本节中，将了解如何使用 CLI 执行最基本的 CMake 操作。

与 CMake CLI 的交互可以通过在终端中输入 CMake 命令来完成，假设安装了 CMake，并且 CMake 可执行文件包含在系统的 PATH 变量 (或等效变量) 中。就可以通过在终端中直接使用 cmake 进行验证：



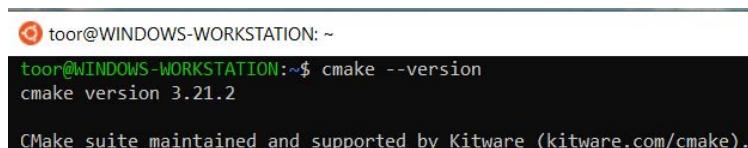
```
toor@WINDOWS-WORKSTATION: ~
toor@WINDOWS-WORKSTATION:~$ cmake
Command 'cmake' not found, but can be installed with:
sudo apt install cmake
toor@WINDOWS-WORKSTATION:~$
```

图 2.1 使用 cmake 命令

若终端抱怨缺少命令，需要安装 CMake 或者将其可执行文件的目录添加到系统的 PATH 环境变量中。关于如何向系统的 PATH 变量添加路径，请参考操作系统指南。

在安装 CMake 并将其添加到 PATH 变量后 (如果需要)，需要测试 CMake 是否可用。可以在命令行中执行 cmake --version，检查 CMake 版本。

---



```
toor@WINDOWS-WORKSTATION: ~
toor@WINDOWS-WORKSTATION:~$ cmake --version
cmake version 3.21.2
CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

图 2.2 终端中查看 CMake 版本

CMake 版本将以 <maj.min.rev> 的形式输出。

#### Note

若版本与安装的不匹配，可能系统中安装了多个 CMake。由于本书包含了为 CMake 3.21 及以上版本编写的示例，因此建议在进一步深入前修复该问题。

安装 CMake 之后，应该安装构建系统和编译器。对于类似 Debian 的操作系统（例如，Debian 和 Ubuntu），这可以通过 `sudo apt install build-essential` 命令轻松完成，包含对 `gcc`、`g++` 和 `make` 的安装。

CLI 的用法将在 Ubuntu 20.04 环境中说明，在其他环境中的用法也相同。

### 2.2.1 了解 CMake 命令行的基本操作

下面列出了使用 CMake 命令行应该了解的三件事：

- 配置
- 构建
- 安装

学习了基础知识之后，将能够构建和安装 CMake 项目。

#### 通过命令行配置项目

要通过命令行配置 CMake 项目，可以使用 `CMake -G "Unix Makefiles" -S -B <output_directory>` 的方式。`-S` 参数用于指定要配置的 CMake 项目，而 `-B` 参数用于指定配置输出目录。最后，`-G` 参数指定将用于生成构建系统的生成器，配置过程的结果将写入 `<output_directory>`。

在项目根构建目录中配置我们书中的示例项目：

```
toor@WINDOWS-WORKSTATION:~/workspace$ git clone https://github.com/PacktPublishing/CMake-Tips-and-Tricks.git
Cloning into 'CMake-Tips-and-Tricks'...
remote: Enumerating objects: 95, done.
remote: Counting objects: 100% (95/95), done.
remote: Compressing objects: 100% (59/59), done.
remote: Total 95 (delta 32), reused 85 (delta 26), pack-reused 0
Unpacking objects: 100% (95/95), 10.49 KiB | 38.00 KiB/s, done.
toor@WINDOWS-WORKSTATION:~/workspace$ ls
CMake-Tips-and-Tricks
```

图 2.3 克隆示例代码存储库

#### 重要 Note

项目必须存在于环境中。若没有，需要在 Git 在终端中执行 `git clone https://github.com/PacktPublishing/CMake-Best-Practices.git`。

现在进入 `CMake-Best-Practices` 目录，使用 `cmake -G "Unix Makefiles" -S . -B ./build:`

```
toor@WINDOWS-WORKSTATION: ~/workspace/CMake-Tips-and-Tricks
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -S . -B build
-- The CXX compiler identification is GNU 9.3.0
-- 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
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$
```

图 2.4 使用 CMake 配置示例代码

使用”Unix Makefiles” (-G "Unix Makefiles") 生成器在当前目录 (-S .) 生成 CMake 项目的构建系统 (-B ./build) 目录。

CMake 将设置源文件夹为当前文件夹中的项目。当省略构建类型时，CMake 使用 Debug 构建类型 (项目默认的 CMAKE\_BUILD\_TYPE)。

后续章节中，我们将学习配置步骤中使用的基本设置。

### 修改构建类型

CMake 不假定构建类型。为了设置构建类型，必须在配置阶段提供一个名为 CMAKE\_BUILD\_TYPE 的变量，变量必须以-D 作为前缀。

要获得 Release 版本，而不是 Debug 版本，需要在配置命令中添加 CMAKE\_BUILD\_TYPE 变量，该命令为:cmake -G "Unix Makefiles" -DCMAKE\_BUILD\_TYPE:STRING=Release -S . -B ./build。

#### Note

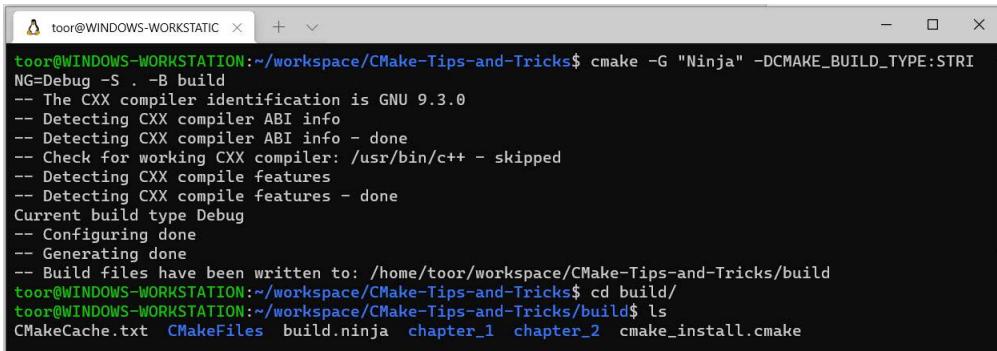
CMAKE\_BUILD\_TYPE 变量只对单配置生成器有意义，例如 Unix Makefiles 和 Ninja。在多配置生成器中，例如 Visual Studio，构建类型是一个构建时参数，不是一个配置时参数，因此不能通过使用 CMAKE\_BUILD\_TYPE 来配置。

### 更换生成器类型

根据不同的环境，CMake 会在默认情况下尝试选择合适的生成器。要显式指定生成器，-G 参数必须提供有效的生成器名称。例如，若想使用 Ninja 作为构建系统，而不是 make，可以这样：

```
cmake -G "Ninja" -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B ./build
```

输出的信息应如下图所示：

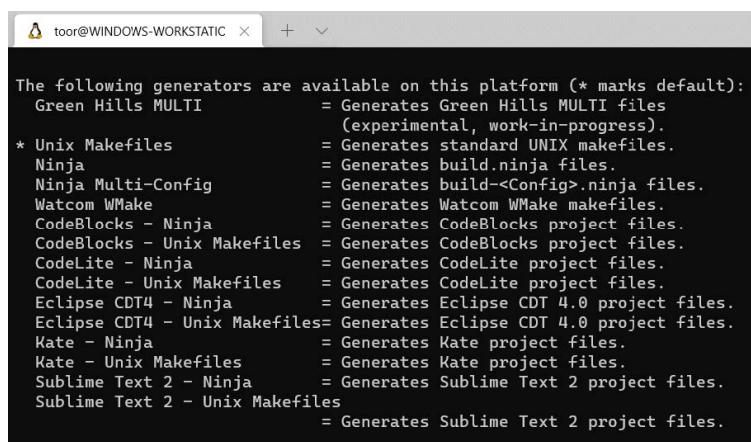


```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Ninja" -DCMAKE_BUILD_TYPE:STR
NG=Debug -S . -B build
-- The CXX compiler identification is GNU 9.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Current build type Debug
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cd build/
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks/build$ ls
CMakeCache.txt  CMakeFiles  build.ninja  chapter_1  chapter_2  cmake_install.cmake
```

图 2.5 检查 CMake 的 Ninja 生成器输出

这将导致 CMake 生成 Ninja 的构建文件，而不是生成 makefile 文件。

为了查看环境中可用的生成器类型，可使用 `cmake --help`。可用的生成器将在帮助文本生成器部分的末尾列出：



```
The following generators are available on this platform (* marks default):
  Green Hills MULTI           = Generates Green Hills MULTI files
                                (experimental, work-in-progress).
* Unix Makefiles             = Generates standard UNIX makefiles.
  Ninja                       = Generates build.ninja files.
  Ninja Multi-Config          = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
  CodeLite - Ninja             = Generates CodeLite project files.
  CodeLite - Unix Makefiles   = Generates CodeLite project files.
  Eclipse CDT4 - Ninja        = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
  Kate - Ninja                 = Generates Kate project files.
  Kate - Unix Makefiles       = Generates Kate project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
  Sublime Text 2 - Unix Makefiles= Generates Sublime Text 2 project files.
```

图 2.6 可用生成器列表

带有星号的生成器是当前环境的默认值。

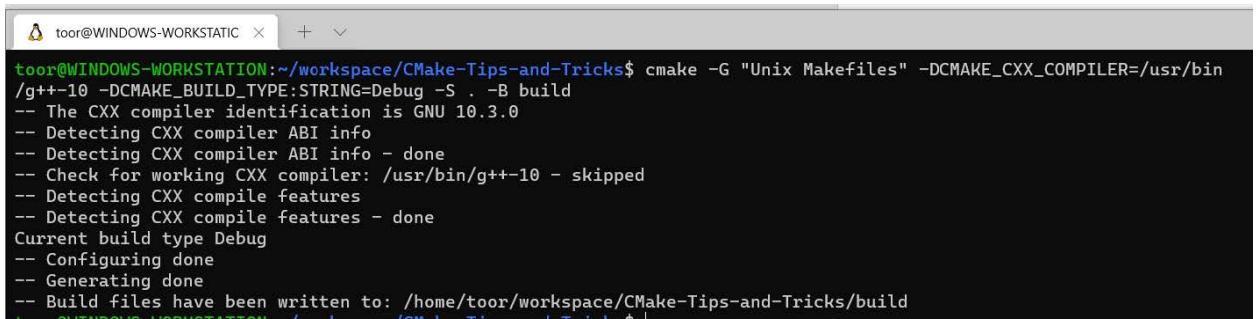
## 修改编译器

CMake 中，编译器可以通过 `CMAKE_<LANG>_COMPILER` 来指定。为了改变编译器，必须在配置命令中提供 `CMAKE_<LANG>_COMPILER`。对于 C/C++ 项目，通常修改的是 `CMAKE_C_COMPILER`(C 编译器) 和 `CMAKE_CXX_COMPILER`(C++ 编译器)。编译器标志由 `CMAKE_<LANG>_FLAGS` 控制。此变量可用于保存与配置无关的编译器标志。

例如，尝试使用 `g++-10` 作为 C++ 编译器，但它不是默认编译器：

```
cmake -G "Unix Makefiles" -DCMAKE_CXX_COMPILER=/usr/bin/g++-10 -S .
-B ./build
```

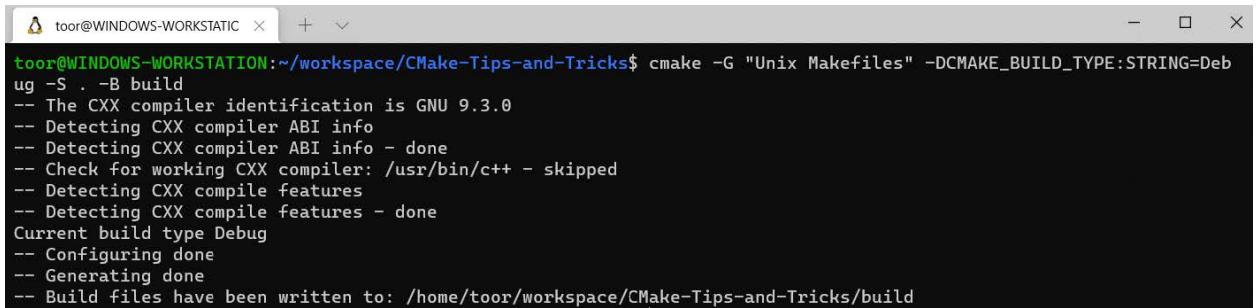
可以看到使用了 `g++-10`，而不是系统默认的编译器 `g++-9`：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Unix Makefiles" -DCMAKE_CXX_COMPILER=/usr/bin/g++-10 -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B build
-- The CXX compiler identification is GNU 10.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/g++-10 - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
Current build type Debug
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
```

图 2.7 使用不同的编译器配置项目 (g++-10)

没有编译器规范的情况下，CMake 更倾向于在这种环境下使用 g++-9:



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B build
-- The CXX compiler identification is GNU 9.3.0
-- 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
Current build type Debug
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
```

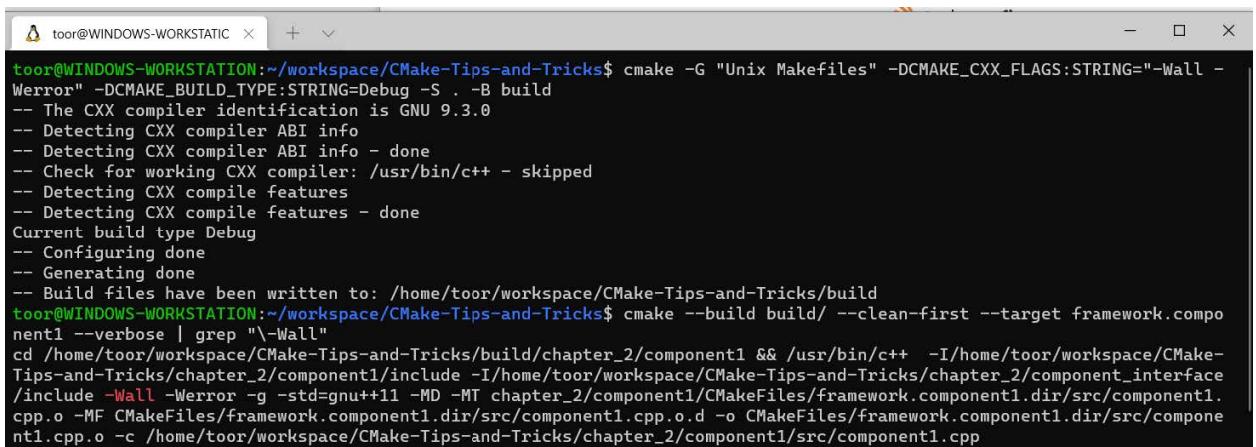
图 2.8 不使用编译器配置的情况

## 向编译器传递编译标志

为了演示如何指定编译器标志，假设需要启用警视为错误。这在 gcc 工具链中分别用-Wall 和-Werror 编译器标志控制。因此，需要将这些标志传递给 C++ 编译器：

```
cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall -Werror" -S . B ./build S . -B ./build
```

下面的例子中，将标志 (-Wall 和-Werror) 传递给了编译器：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall -Werror" -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B build
-- The CXX compiler identification is GNU 9.3.0
-- 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
Current build type Debug
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake --build build/ --clean-first --target framework.component1 --verbose | grep "\-Wall"
cd /home/toor/workspace/CMake-Tips-and-Tricks/build/chapter_2/component1 && /usr/bin/c++ -I/home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1/include -I/home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1/include -Wall -Werror -g -std=gnu++11 -MD -MT chapter_2/component1/CMakeFiles/framework.component1.dir/src/component1.cpp.o -MF CMakeFiles/framework.component1.dir/src/component1.cpp.o.d -o CMakeFiles/framework.component1.dir/src/component1.cpp.o -c /home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1/src/component1.cpp
```

图 2.9 向 C++ 编译器传递标志

构建标志可以为每个构建类型定制，方法是在它们后面加上大写的构建类型字符串。有四个变量用于四个不同的构建类型，它们对于根据编译器标志指定构建类型非常有用。这些变量中指定的标志只在配置构建类型匹配时有效：

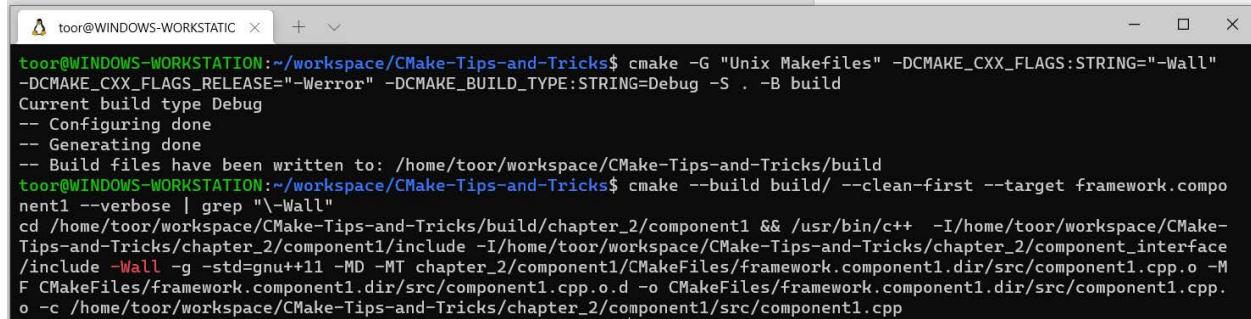
1. CMAKE\_<LANG>\_FLAGS\_DEBUG
2. CMAKE\_<LANG>\_FLAGS\_RELEASE
3. CMAKE\_<LANG>\_FLAGS\_RELWITHDEBINFO
4. CMAKE\_<LANG>\_FLAGS\_MINSIZEREL

若希望只在 Release 构建中将警告视为错误，可以使用构建类型特定的编译器标志。

下面的示例，演示了如何使用特定于构建类型的编译器标志：

```
cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall  
-Werror" -DCMAKE_CXX_FLAGS_RELEASE:STRING="-O3" -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B ./build
```

前面的命令中有一个 CMAKE\_CXX\_FLAGS\_RELEASE 参数，只有当构建类型为 Release 时，这个变量中的内容才会传递给编译器。由于构建类型指定为 Debug，可以看到传递给编译器的标志中没有-O3 标志：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall"  
-DCMAKE_CXX_FLAGS_RELEASE:STRING="-Wall" -DCMAKE_BUILD_TYPE:STRING=Debug -S . -B build  
Current build type: Debug  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build  
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake --build build/ --clean-first --target framework.compo  
nent1 --verbose | grep "\-Wall"  
cd /home/toor/workspace/CMake-Tips-and-Tricks/build/chapter_2/component1 && /usr/bin/c++ -I/home/toor/workspace/CMake-  
Tips-and-Tricks/chapter_2/component1/include -I/home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1_interface/  
include -Wall -g -std=gnu++11 -MD -MT chapter_2/component1/CMakeFiles/framework.component1.dir/src/component1.cpp.o -M  
F CMakeFiles/framework.component1.dir/src/component1.cpp.o.d -o CMakeFiles/framework.component1.dir/src/component1.cpp.  
o -c /home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1/src/component1.cpp
```

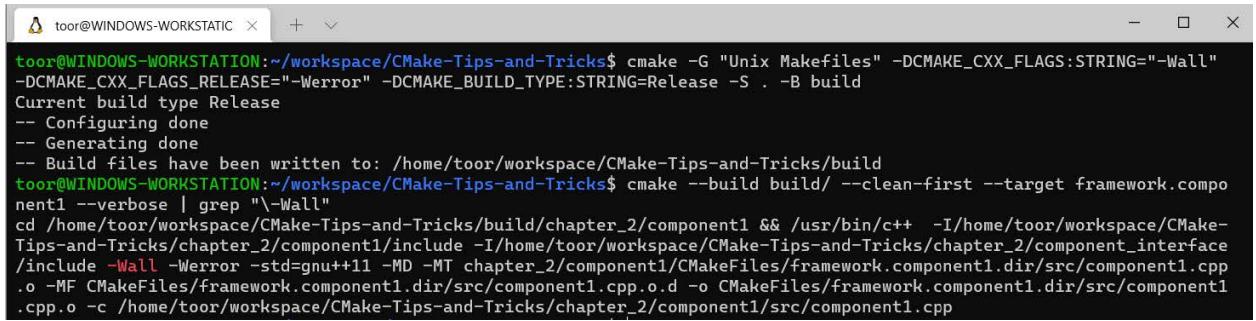
图 2.10 根据构建类型指定标志——Debug 版本中没有-O3 标志

图 2.10 中，CMake 发出一个关于指定，但未使用变量的警告，CMAKE\_CXX\_FLAGS\_RELEASE。这确认了 CMAKE\_CXX\_FLAGS\_RELEASE 没有在 Debug 构建类型中使用。当构建类型指定为 Release 时，可以看到-O3 标志：

```
cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall  
-Werror" -DCMAKE_CXX_FLAGS_RELEASE:STRING="-O3"  
-DCMAKE_BUILD_TYPE:STRING= "Release" -S . -B ./build
```

相当于对 CMake 说，配置位于当前目录的 CMake 项目，使用“Unix Makefiles”生成器来构建/文件夹。对于所有构建类型，无条件地将-Wall 标志传递给编译器。若构建类型是 Release，也传递-O3 标志。

以下是当构建类型设置为 Release 时命令的输出：



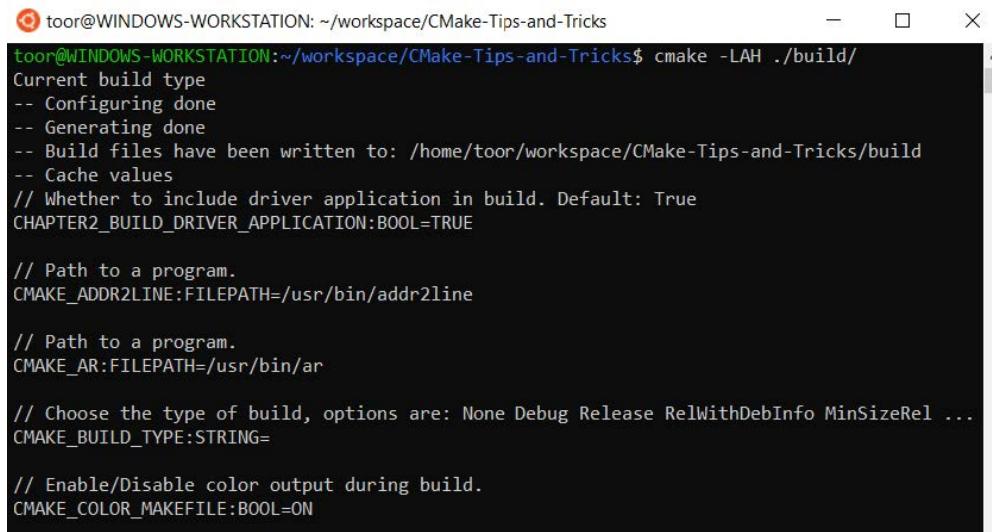
```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -G "Unix Makefiles" -DCMAKE_CXX_FLAGS:STRING="-Wall"
-DCMAKE_CXX_FLAGS_RELEASE="-Werror" -DCMAKE_BUILD_TYPE:STRING=Release -S . -B build
Current build type Release
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake --build build/ --clean-first --target framework.component1 --verbose | grep "\-Wall"
cd /home/toor/workspace/CMake-Tips-and-Tricks/build/chapter_2/component1 && /usr/bin/c++ -I/home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component_interface/include -Wall -Werror -std=gnu++11 -MD -MT chapter_2/component1/CMakeFiles/framework.component1.dir/src/component1.cpp.o -MF CMakeFiles/framework.component1.dir/src/component1.cpp.o.d -o CMakeFiles/framework.component1.dir/src/component1.cpp.o -c /home/toor/workspace/CMake-Tips-and-Tricks/chapter_2/component1/src/component1.cpp
```

图 2.11 根据构建类型指定标志——在 Release 版本中有-O3 标志

图 2.11 中，可以确认-O3 标志也传递给了编译器。注意，即使 RelWithDebInfo 和 MinSizeRel 也是 Release 版本，它们与 Release 版本类型是分开的，因此在 CMAKE\_<LANG>\_FLAGS\_RELEASE 中指定的标志不适用于它们。

## 缓存变量列表

可以通过 `cmake -L ./build/` 列出所有缓存的变量（参见图 2.12）。默认情况下，这不会显示与每个变量关联的高级变量和帮助字符串。也可以使用 `cmake -LAH ./build/` 来显示它们。



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake -LAH ./build/
Current build type
-- Configuring done
-- Generating done
-- Build files have been written to: /home/toor/workspace/CMake-Tips-and-Tricks/build
-- Cache values
// Whether to include driver application in build. Default: True
CHAPTER2_BUILD_DRIVER_APPLICATION:BOOL=TRUE

// Path to a program.
CMAKE_ADDR2LINE:FILEPATH=/usr/bin/addr2line

// Path to a program.
CMAKE_AR:FILEPATH=/usr/bin/ar

// Choose the type of build, options are: None Debug Release RelWithDebInfo MinSizeRel ...
CMAKE_BUILD_TYPE:STRING=

// Enable/Disable color output during build.
CMAKE_COLOR_MAKEFILE:BOOL=ON
```

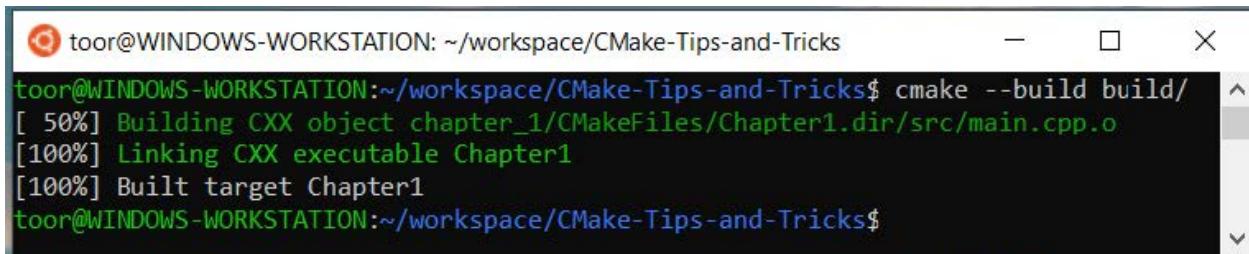
图 2.12 CMake 转储的缓存变量列表

## 通过 CLI 构建配置好的项目

要生成已配置的工程，需要使用 `cmake --build ./build` 在构建文件夹中配置的 CMake 项目。

也可以使用 `cd build && make` 进行构建。使用 `cmake --build` 的好处是，省去了调用特定于构建系统的命令。在构建 CI 管道或构建脚本时，可以在不更改构建命令的情况下，更改构建系统生成器。

可以看到 `cmake --build ./build` 的输出示例如下所示：



```
toor@WINDOWS-WORKSTATION: ~/workspace/CMake-Tips-and-Tricks
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake --build build/
[ 50%] Building CXX object chapter_1/CMakeFiles/Chapter1.dir/src/main.cpp.o
[100%] Linking CXX executable Chapter1
[100%] Built target Chapter1
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$
```

图 2.13 构建配置好的项目

## 并行构建

还可以在发出生成命令时，自定义生成时间详细信息。要指定并行构建的任务数，可以在 `cmake --build` 中追加 `--parallel <job_count>`。

要并行构建，请使用 `cmake --build ./build --parallel 2`，其中数字 2 指定了任务数。对于构建系统，建议的任务数最多为每个硬件线程一个作业。多核系统中，还建议至少使用比可用硬件线程数少一个的线程，以避免在构建过程中影响系统的响应能力。

### Note

通常可以在每个硬件线程中使用多个任务，从而获得更快的构建时间，因为构建过程主要受 I/O 限制，但实际情况可能有所不同，需要进行实验和观察。

此外，某些构建系统（如 Ninja）会尝试利用系统中可用的尽可能多的硬件线程，若目标是使用系统中的所有硬件线程，那么为此类构建系统指定任务数就多余了。可以在 Linux 环境中，使用 `nproc` 命令来获取硬件线程计数。

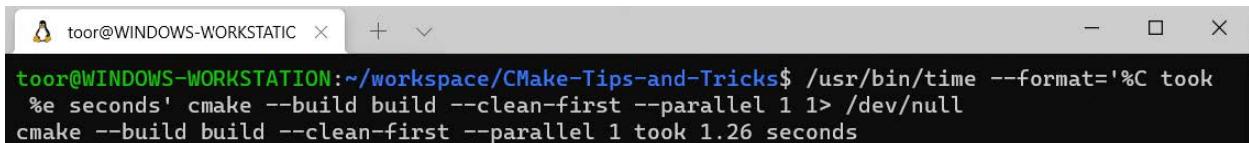
在期望在不同环境中调用的命令（如 CI/CD 脚本和构建脚本）中，不使用环境相关变量的固定值，这是一个很好的实践。下面是一个使用 `nproc` 动态确定并行任务数量的构建命令示例：

```
cmake --build ./build/ --parallel $(( $(nproc) - 1 ))
```

观察不同的任务数如何影响构建时间，将使用时间工具来测试每次命令调用的时间。环境如下：

- OS: Ubuntu 20.04.3 LTS (Focal Fossa)
- CPU: AMD Ryzen Threadripper 1950X 16-Core Processor (32 线程)
- RAM: 32 GB

对于一个任务（`--parallel 1`），构建时间如下所示：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ /usr/bin/time --format='%C took
%e seconds' cmake --build build --clean-first --parallel 1 1> /dev/null
cmake --build build --clean-first --parallel 1 took 1.26 seconds
```

图 2.14 一个任务的并行构建时间结果

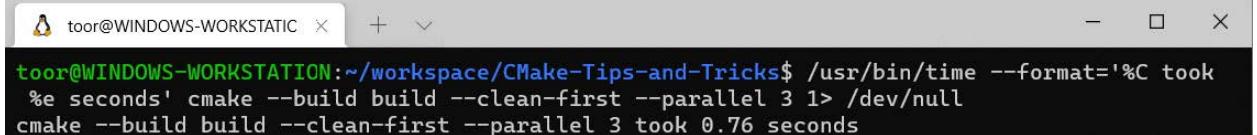
两个任务（`--parallel 2`）的构建时间结果如下：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ /usr/bin/time --format='%.C took %e seconds' cmake --build build --clean-first --parallel 2 1> /dev/null
cmake --build build --clean-first --parallel 2 took 0.96 seconds
```

图 2.15 两个任务的并行构建时间结果

三个任务 (--parallel 3) 的构建时间结果如下:



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ /usr/bin/time --format='%.C took %e seconds' cmake --build build --clean-first --parallel 3 1> /dev/null
cmake --build build --clean-first --parallel 3 took 0.76 seconds
```

图 2.16 三个任务的并行构建时间结果

四个任务 (--parallel 4) 的构建时间结果如下:



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ /usr/bin/time --format='%.C took %e seconds' cmake --build build --clean-first --parallel 4 1> /dev/null
cmake --build build --clean-first --parallel 4 took 0.75 seconds
```

图 2.17 四个任务的并行构建时间结果

即使是一个非常简单的项目，也可以清楚地看到多个任务并行可以减少构建时间。从一个作业到两个任务减少了 0.3 秒的构建时间，而从两个任务到三个任务则减少了 0.2 秒。但是，从 3 个作业到 4 个任务只会产生 0.01 秒的差异，这意味着已经达到了这个项目的构建并行性的极限，增加更多的工作将不会在构建时间上实现显著的差异。

## 只构建特定的目标

通常，CMake 将构建已配置的所有可用目标。由于构建所有目标并不总是理想的，CMake 允许通过--target 子选项构建目标的子集，子选项可以指定多次:

```
cmake --build ./build/ --target "ch2_framework_component1"
--target "ch2_framework_component2"
```

该命令将构建范围限制为 ch2\_framework\_component1 和 ch2\_framework\_component2 目标。若这些目标也依赖于其他目标，它们也会构建。

## 构建之前删除以前的构建

若想要运行一个干净的构建，首先就要删除之前的构建。为此，可以使用--clean-first 子选项。此子选项将调用一个特殊的目标，该目标清除由构建过程生成的所有构件(例如，调用 make clean)。

下面是一个示例，说明如何对名为 build 的构建文件夹执行此操作:

```
cmake --build ./build/ --clean-first
```

## 调试构建过程

正如在前面的向编译器传递标志一节中所做的那样，可能希望检查构建过程中使用哪些参数调用了哪些命令。--verbose 指示 CMake 以 verbose 模式调用所有构建命令，前提是该命令支持 verbose 模式。这使我们能够轻松地调试编译和链接错误。

要在 verbose 模式下构建名为 build 的文件夹，可以使用--build:

```
cmake --build ./build/ --verbose
```

## 将命令行参数传递给构建工具

若需要将参数传递给底层构建工具，可以在命令的末尾加上--，并输入给定的参数:

```
cmake --build ./build/ -- --trace
```

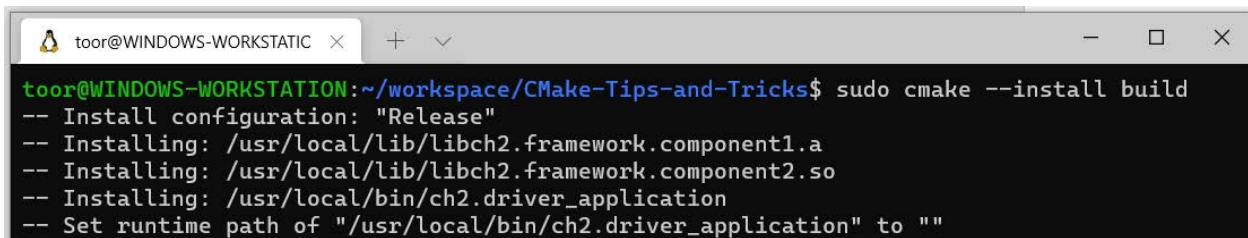
--trace 将直接传递给构建工具，例子中就是 make。这将使 make 为构建每个文件的编译和连接输出跟踪信息。

## 通过 CLI 安装项目

CMake 允许在环境中进行安装，CMake 代码必须使用 `install()` 指令来指定当调用 `cmake --install(或构建系统等效)` 时安装什么。`chapter2` 的内容已经以这样的方式配置用于说明指令。我们将在第 4 章中学习如何使 CMake 目标可安装。

`cmake --install` 需要一个已经配置和生成的项目，执行 `cmake --install <project_binary_dir>` 来安装工程。我们的例子中，`build` 作为项目二进制目录，所以 `<project_binary_dir>` 将为 `build`。

示例输出如下图所示:



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ sudo cmake --install build
-- Install configuration: "Release"
-- Installing: /usr/local/lib/libch2.framework.component1.a
-- Installing: /usr/local/lib/libch2.framework.component2.so
-- Installing: /usr/local/bin/ch2.driver_application
-- Set runtime path of "/usr/local/bin/ch2.driver_application" to ""
```

图 2.18 安装工程

不同环境的默认安装目录不同。对于类 Unix 环境，默认为 `/usr/local`，而在 Windows 环境中，默认为 `C:/Program Files`。

## Tip

在尝试安装项目之前，必须已经构建了项目。

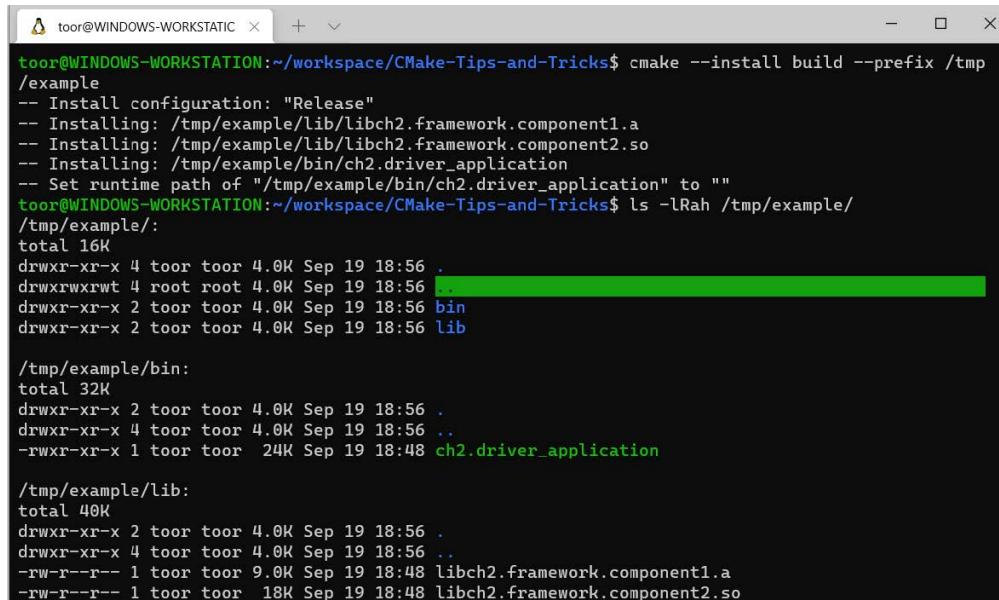
为了能够成功安装项目，必须拥有适当的权限来写入安装目标目录。

### 修改默认安装路径

要更改默认安装目录，可以指定的`--prefix`参数，如下所示：

```
cmake --install build --prefix /tmp/example
```

将安装目录指定为/tmp/example，使用 `cmake --install` 后，/tmp/example 目录下的内容如下图所示：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ cmake --install build --prefix /tmp/example
-- Install configuration: "Release"
-- Installing: /tmp/example/lib/libch2.framework.component1.a
-- Installing: /tmp/example/lib/libch2.framework.component2.so
-- Installing: /tmp/example/bin/ch2.driver_application
-- Set runtime path of "/tmp/example/bin/ch2.driver_application" to ""
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ ls -lRah /tmp/example/
/tmp/example/:
total 16K
drwxr-xr-x 4 toor toor 4.0K Sep 19 18:56 .
drwxrwxrwt 4 root root 4.0K Sep 19 18:56 ..
drwxr-xr-x 2 toor toor 4.0K Sep 19 18:56 bin
drwxr-xr-x 2 toor toor 4.0K Sep 19 18:56 lib

/tmp/example/bin:
total 32K
drwxr-xr-x 2 toor toor 4.0K Sep 19 18:56 .
drwxr-xr-x 4 toor toor 4.0K Sep 19 18:56 ..
-rw-r--r-- 1 toor toor 24K Sep 19 18:48 ch2.driver_application

/tmp/example/lib:
total 40K
drwxr-xr-x 2 toor toor 4.0K Sep 19 18:56 .
drwxr-xr-x 4 toor toor 4.0K Sep 19 18:56 ..
-rw-r--r-- 1 toor toor 9.0K Sep 19 18:48 libch2.framework.component1.a
-rw-r--r-- 1 toor toor 18K Sep 19 18:48 libch2.framework.component2.so
```

图 2.19 将项目安装到不同的路径

可以看到，安装根目录成功更改为/tmp/example。

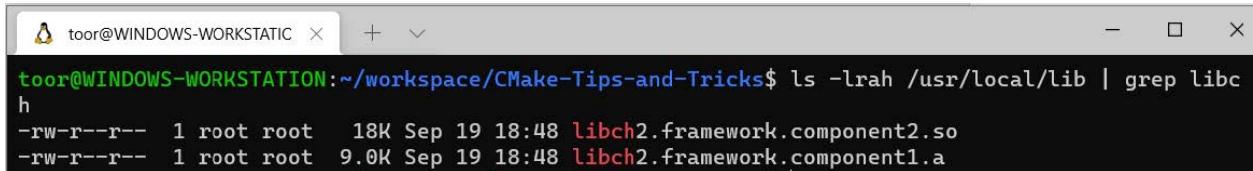
### 安装时瘦身二进制文件

软件世界中，构建工件通常与一些额外的信息捆绑在一起，例如：调试所需的符号表。这些信息在最终产品时可能不是必需的，并且可能会极大地增加二进制大小。若希望减少最终产品的存储空间，瘦身二进制文件可能是一个不错的选择。瘦身的另一个好处是，这会使逆向工程二进制文件变得更加困难，因为从二进制文件中剥离了基本的符号信息。

CMake 的`--install` 允许在安装操作时瘦身二进制文件。可以通过在`--install` 中使用`--strip` 选项来启用：

```
cmake --install build --strip
```

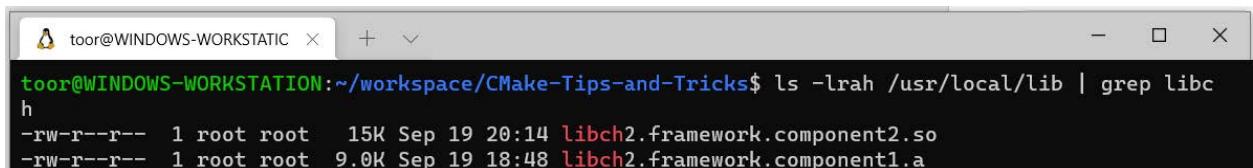
下面的示例中，可以观察未瘦身和瘦身的二进制文件之间的大小差异。注意，瘦身静态库有一些限制，CMake 默认情况下不会执行：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ ls -lrah /usr/local/lib | grep libch2
-rw-r--r-- 1 root root 18K Sep 19 18:48 libch2.framework.component2.so
-rw-r--r-- 1 root root 9.0K Sep 19 18:48 libch2.framework.component1.a
```

图 2.20 工件大小 (未瘦身)

使用瘦身的 (cmake - install build --strip) 二进制文件，大小差异如下图所示：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ ls -lrah /usr/local/lib | grep libch2
-rw-r--r-- 1 root root 15K Sep 19 20:14 libch2.framework.component2.so
-rw-r--r-- 1 root root 9.0K Sep 19 18:48 libch2.framework.component1.a
```

图 2.21 工件大小 (已瘦身)

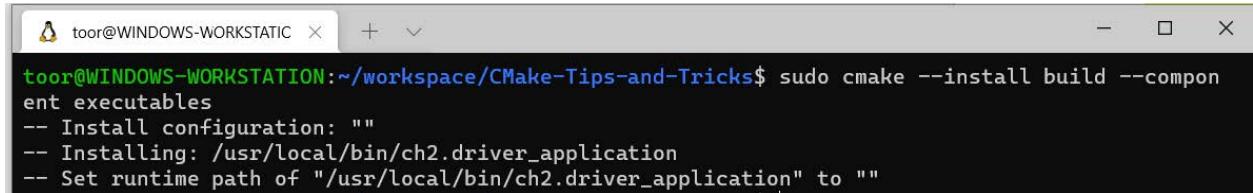
### 安装特定组件 (基于组件的安装)

若项目在 `install()` 中使用了 CMake 的 COMPONENT 特性，可以通过指定组件名称来安装特定的组件。组件特性允许将安装分离为子部分，为了说明这个功能，chapter2 示例构造成两个组件，分别是库和可执行文件。

为了安装特定的组件，`cmake --install` 需要附加`--component` 参数：

```
cmake --install build --component executables
```

下面是一个示例：



```
toor@WINDOWS-WORKSTATION:~/workspace/CMake-Tips-and-Tricks$ sudo cmake --install build --component executables
-- Install configuration: ""
-- Installing: /usr/local/bin/ch2.driver_application
-- Set runtime path of "/usr/local/bin/ch2.driver_application" to ""
```

图 2.22 只安装特定的组件

### 安装特定的配置 (仅用于多配置生成器)

有些生成器支持同一个构建配置的多个配置 (例如，Visual Studio)。对于这种类型的生成器，`--install` 选项提供了一个`--config` 参数来指定要安装的二进制文件配置。

这里有一个例子：

```
cmake --install build --config Debug
```

#### Note

示例中使用的命令参数非常长且明确，显式地指定参数允许我们在每次运行中获得一致的结果，无论在哪个环境中运行命令。若没有-G 参数，CMake 将默认使用环境首选的构建系统生成器。我们的座右铭是：显式总是比隐式好，前者使我们的意图更加清晰，也使 CMake 代码更容易维护。

我们已经介绍了 CMake 命令行用法的基本原理，继续学习其他可用的交互形式——CMake 的图形界面。

## 2.3. CMake-GUI 和 ccmake 的高级配置

大多数界面做的事情相同，只是外观不同而已；因此，前一节中介绍的大多数内容在这里也同样适用。我们要改变的是互动的形式，而不是实际使用的工具。

#### Note

请检查 ccmake 命令在终端中是否可用。若没有，请验证 PATH 变量是否设置正确，并检查安装是否正确。

### 2.3.1 如何使用 ccmake

ccmake 是 CMake 的一个基于终端的图形用户界面 (GUI)，允许用户编辑缓存的 CMake 变量。不叫它 GUI，术语终端用户界面 (TUI) 可能更适合，因为没有传统的 Shell UI 元素，窗口和按钮使用文本接口在终端中呈现。

ccmake 是 CMake 的一部分，所以除了 CMake 之外，不需要额外的安装。使用 ccmake 与在 CLI 中使用 CMake 相同，只是缺少了构建和安装步骤。主要的区别是 ccmake 是基于终端的图形界面，用于交互式编辑缓存的 CMake 变量。ccmake 的状态栏将显示每个设置及其可能值的描述。

使用 ccmake 配置项目时，与之前在通过 CLI 配置项目一样：

```
ccmake -G "Unix Makefiles" -S . -B ./build
```

使用示例如下所示：

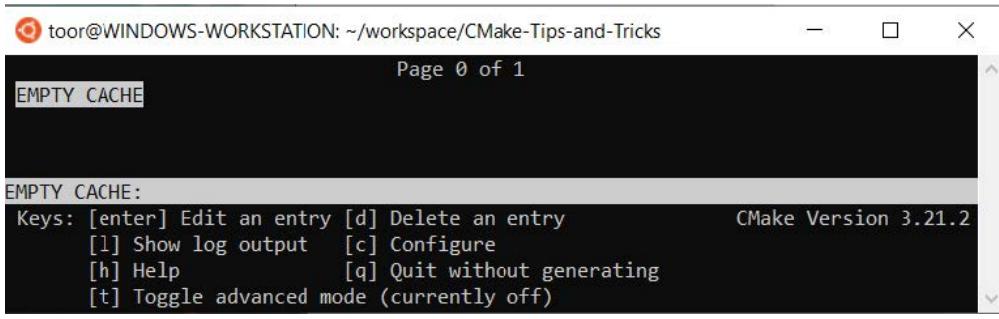


图 2.23 ccmake 主界面

执行该命令后，将出现基于终端的用户界面。初始页面是可以编辑 CMake 变量的页面，空缓存意味着没有预先配置和 CMake 缓存文件 (CMakeCache.txt) 目前是空的。为了编辑变量，必须首先配置项目。要进行配置，请按下键盘上的 C 键，如 Keys: 部分所示。

按下 C 键后，将执行 CMake 配置步骤，并进入日志输出界面，输出配置信息：

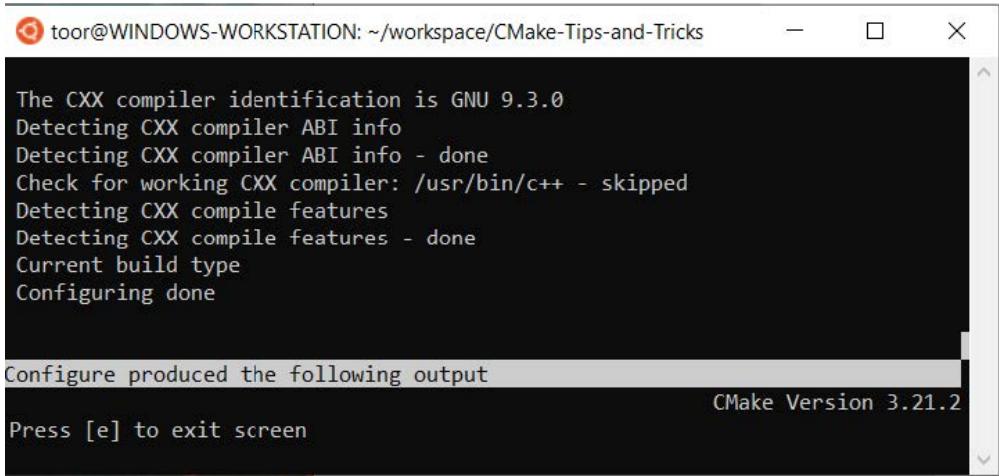


图 2.24 配置后 ccmake 的日志界面

关闭日志输出屏幕并返回到主屏幕，请按 E。返回时，注意 EMPTY CACHE 会替换成 CMakeCache.txt 文件中的变量名称。要选择一个变量，请使用键盘上的向上和向下箭头键。选中的变量将以白色高亮显示，如下图所示：

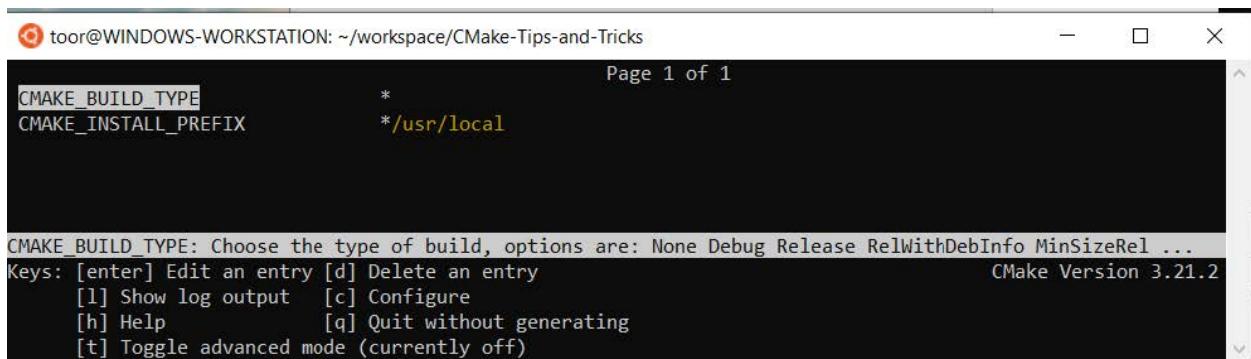
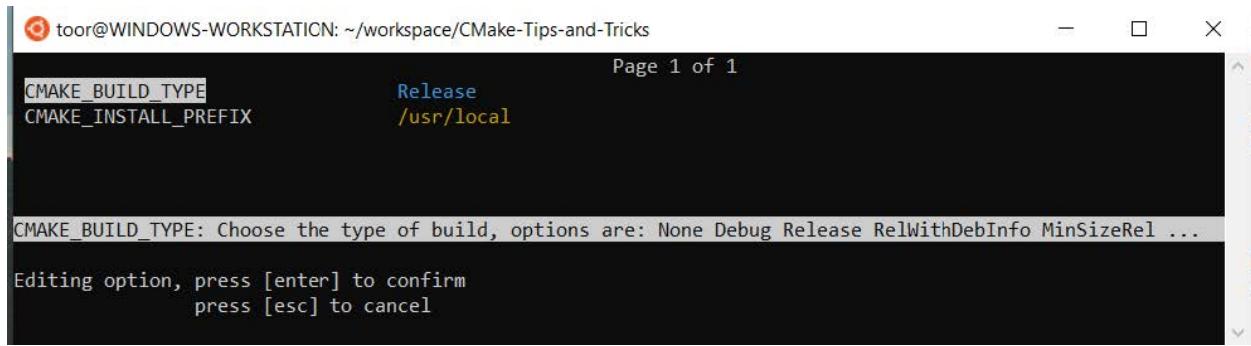


图 2.25 配置完成后 ccmake 的主界面

前面的屏幕截图中，选择了 CMAKE\_BUILD\_TYPE。在右边，显示了 CMake 变量的当前值。对于 CMAKE\_BUILD\_TYPE，现在是空的。变量值旁边的星号表示该变量的值刚刚更改了之前的配置，可以按回车键进行编辑，也可以按 D 键删除。下图显示了修改变量后 ccmake 主界面的样子：



toor@WINDOWS-WORKSTATION: ~/workspace/CMake-Tips-and-Tricks

Page 1 of 1

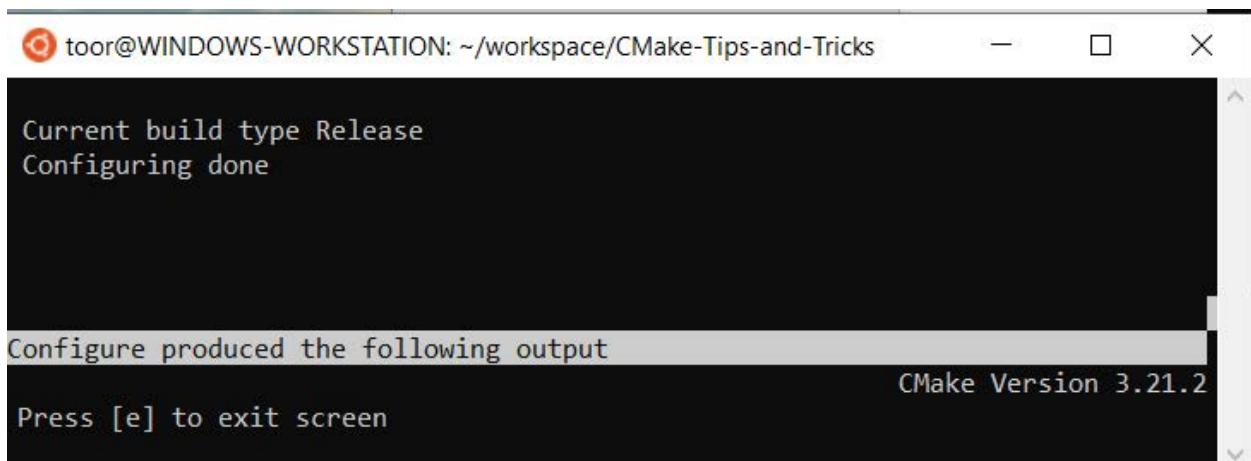
CMAKE_BUILD_TYPE	Release
CMAKE_INSTALL_PREFIX	/usr/local

CMAKE\_BUILD\_TYPE: Choose the type of build, options are: None Debug Release RelWithDebInfo MinSizeRel ...

Editing option, press [enter] to confirm  
press [esc] to cancel

图 2.26 ccmake 的主界面随变量变化

设置 CMAKE\_BUILD\_TYPE 为 Release 并再次配置:



toor@WINDOWS-WORKSTATION: ~/workspace/CMake-Tips-and-Tricks

Current build type Release  
Configuring done

Configure produced the following output

CMake Version 3.21.2

Press [e] to exit screen

图 2.27 ccmake 配置输出 (Release 版)

可以观察到构建类型现在设置为 Release。返回到上一个屏幕，按下 g(生成) 按钮保存更改。可以按 q(退出而不生成) 按钮丢弃更改。

要编辑其他变量，例如：CMAKE\_CXX\_COMPILER 和 CMAKE\_CXX\_FLAGS，高级模式应该打开。默认情况下，可以通过 `mark_as_advanced()` 将这些变量标记为高级变量；默认情况下，在图形界面中是隐藏的，主界面按“t”键可切换到高级模式。

```
CHAPTER2_BUILD_DRIVER_APPLICATION ON
CMAKE_ADDR2LINE /usr/bin/addr2line
CMAKE_AR /usr/bin/ar
CMAKE_BUILD_TYPE ON
CMAKE_COLOR_MAKEFILE /usr/bin/c++
CMAKE_CXX_COMPILER /usr/bin/gcc-ar-9
CMAKE_CXX_COMPILER_AR /usr/bin/gcc-ranlib-9
CMAKE_CXX_COMPILER_RANLIB /usr/bin/gcc-ranlib-9
CMAKE_CXX_FLAGS -g
CMAKE_CXX_FLAGS_DEBUG -Og -DNDEBUG
CMAKE_CXX_FLAGS_MINSIZEREL -Os -DNDEBUG
CMAKE_CXX_FLAGS_RELEASE -O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO -O2 -g -DNDEBUG
CMAKE_DLLTOOL CMAKE_DLLTOOL-NOTFOUND

CHAPTER2_BUILD_DRIVER_APPLICATION: Whether to include driver application in build. Default: True
Keys: [enter] Edit an entry [d] Delete an entry
      [l] Show log output [c] Configure
      [h] Help [q] Quit without generating
      [t] Toggle advanced mode (currently on)
```

图 2.28 - ccmake 的高级模式

激活高级模式后，选项将变得可见。可以观察和修改它们的值，就像普通变量一样。以前隐藏的名为 CHAPTER2\_BUILD\_DRIVER\_APPLICATION 现在出现了，这是一个用户定义的 CMake 变量。该变量的定义如下：

```
# Option to exclude driver application from build.
set(CHAPTER2_BUILD_DRIVER_APPLICATION TRUE CACHE BOOL
    "Whether to include driver application in build. Default: True")
# Hide this option from GUI's by default.
mark_as_advanced(CHAPTER2_BUILD_DRIVER_APPLICATION)
```

CHAPTER2\_BUILD\_DRIVER\_APPLICATION 是一个布尔类型的缓存变量，默认值为 true。它标记为“高级”，这就是为什么它在非高级模式中不显示的原因。

### 2.3.2 使用 cmake-gui

若认为 CLI 反直觉，或更喜欢 GUI 而不是 CLI，那么 CMake 也有跨平台的 GUI。与 ccmake 相比，cmake-gui 提供了更多的功能，比如：环境编辑器和正则表达式资源管理器。

CMake GUI 是默认 CMake 安装的一部分；除了 CMake 之外，不需要额外安装，主要目的是允许用户配置 CMake 项目。要启动 cmake-gui，请在终端中输入 cmake-gui。Windows 下，也可以从开始菜单中找到它。若这些方法都不起作用，进入 CMake 的安装路径，它应该在 bin 目录下。

### Note

若在 Windows 环境中启动 cmake-gui，并且打算使用 Visual Studio 提供的工具链，从 IDE 的适当的本地工具命令提示符启动 cmake-gui。若有多个版本的 IDE，请确保正在使用正确的 Native Tools 命令提示符。否则，CMake 可能无法发现所需的工具，如编译器，或可能发现不正确的工具。更多信息请参考<https://docs.microsoft.com/en-us/visualstudio/ide/reference/command-prompt-powershell?view=vs-2019>。

下面是 CMake GUI 的主窗口：

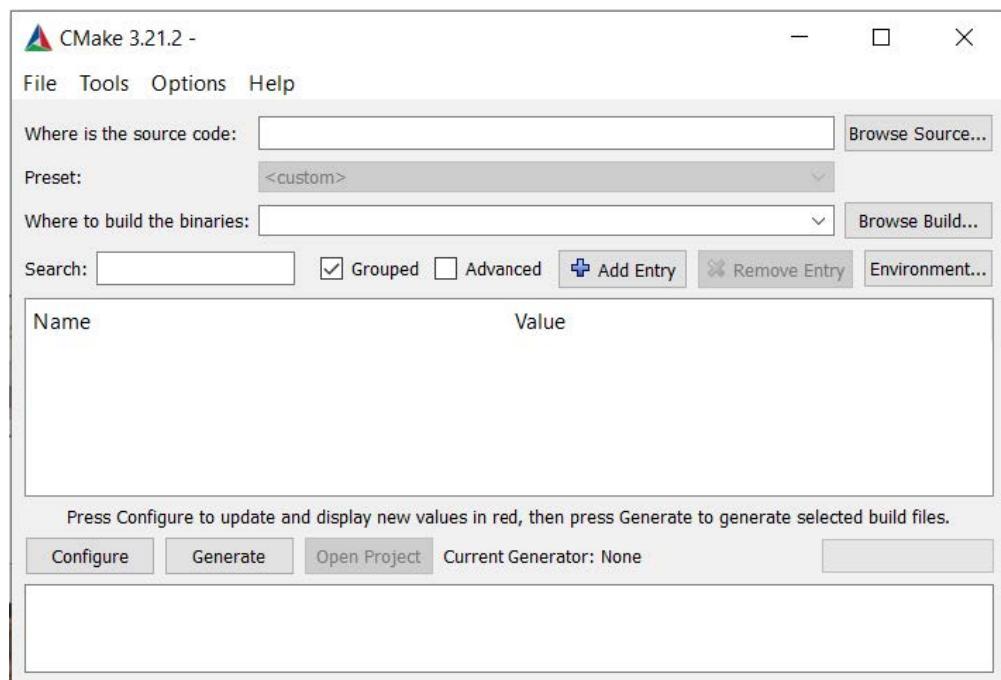


图 2.29 CMake GUI 主窗口

CMake GUI 的主屏幕包含以下内容：

- 源码路径字段
- 输出路径字段
- 配置和生成按钮
- 缓存变量列表

要开始配置项目，请单击 Browse Source…按钮，选择项目的根目录，通过单击 Browse Build…按钮为项目选择一个输出目录。此路径将是所选生成器生成的输出文件的路径。

设置源和输出路径后，单击 Configure 开始配置所选项目。CMake GUI 会让你选择使用的生成器、平台选择(若生成器支持的话)、工具集和编译器等细节，如下图所示：

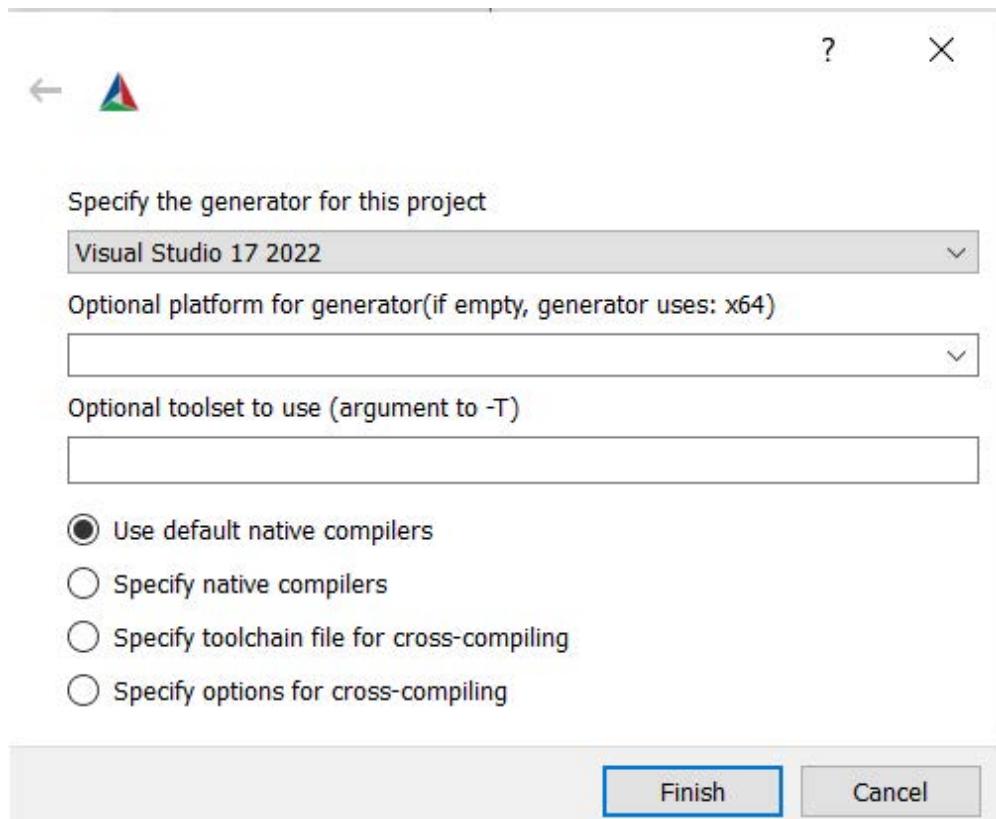


图 2.30 CMake GUI 生成器选择窗口

根据环境填写这些信息之后，单击 Finish 继续。CMake GUI 将开始使用给定的详细信息配置项目，并在日志部分报告输出。配置成功后，也会在缓存变量列表部分看到缓存变量：

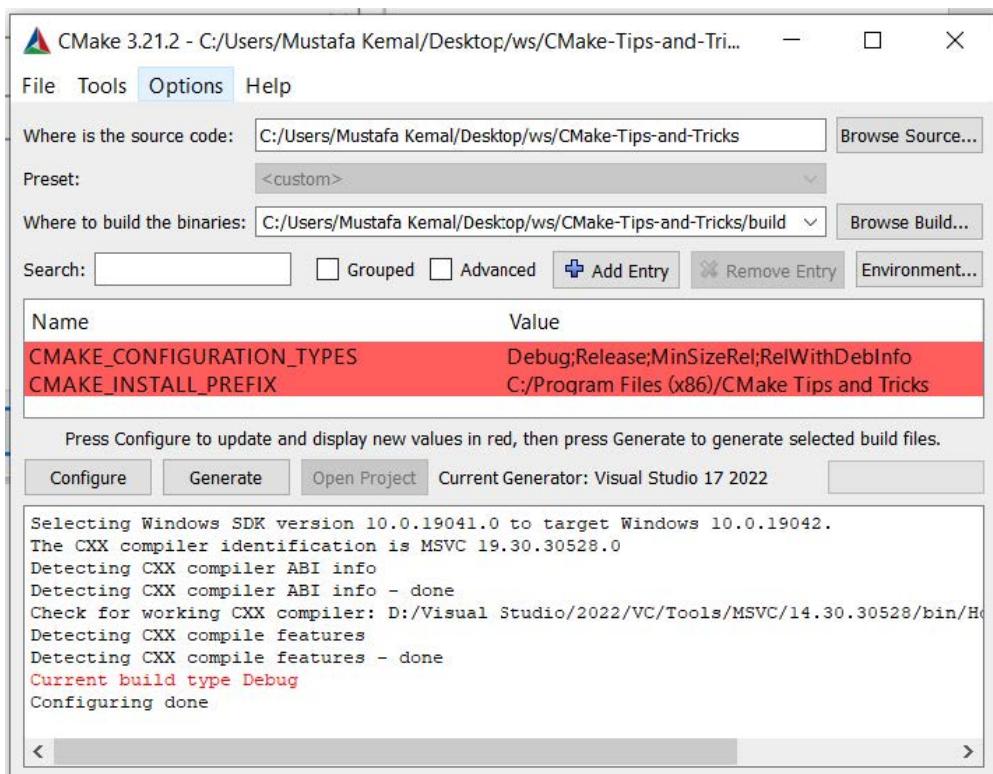


图 2.31 配置后的 CMake GUI

若一切正常，请按 **Generate** 按钮生成所选构建系统所需的构建文件。对于 Visual Studio 生成器，生成的文件是.sln 和.cxxproj 以及其他文件。生成项目之后，**Open project** 按钮将启用，可用其使用适当的编辑器或 IDE 打开生成的项目。若构建系统没有与任何 IDE 关联（例如，makefiles），那么生成的文件将显示出来，并可以使用 IDE 构建项目。

#### 重要 Note

生成的项目只是生成器的工件，对生成的项目文件的更改 (.sln, .cxxproj) 将不会保存，并将在下次生成中丢弃。当修改 CMakeLists.txt 文件或编辑 CMakeCache.txt 文件时，不要忘记重新生成项目文件（直接或间接）。版本控制方面，应该将生成的项目文件视为构建工件，不该将它们添加到版本控制中。可以通过 CMake 和合适的生成器生成项目来重新生成。

有时，项目可能需要调整一些缓存变量，或者需要使用不同的构建类型。要更改缓存变量，请单击所需缓存变量的值。根据变量类型的不同，可能会显示复选框，而不是字符串。若需要的变量在列表中不可见，则可能是高级变量，只有选中窗口上的 **advanced** 复选框时才可见。还可以使用搜索框更容易地定位变量，还可以在 cmake-gui 的高级模式下看到：

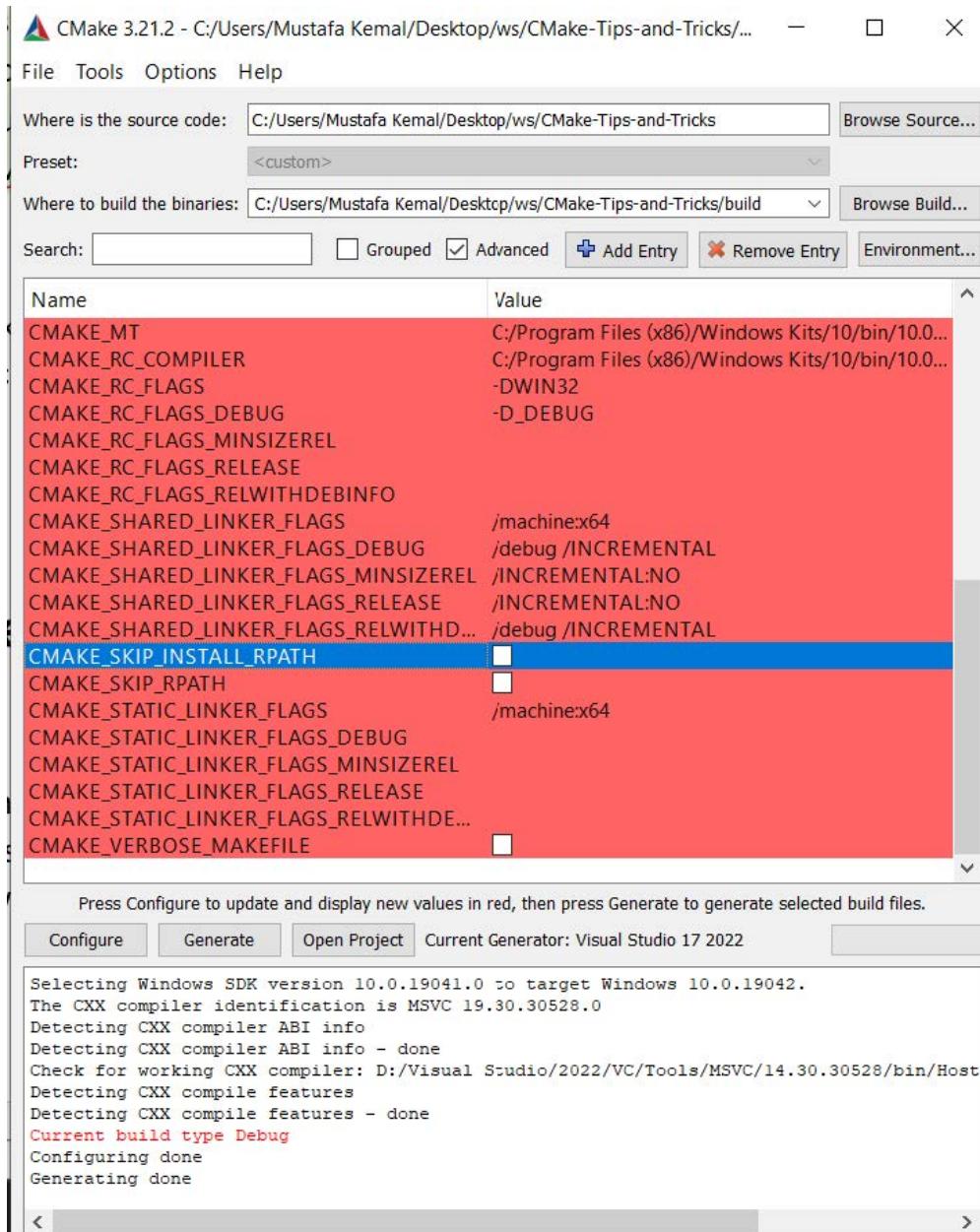


图 2.32 高级模式下的 cmake-gui

调整缓存值之后，单击 Configure，然后单击 Generate 应用更改。

### Tip

另一个特性是分组特性，可以将缓存变量分组到公共前缀中（有的话）。组名由变量名的第一部分确定，直到第一个下划线为止。

已经介绍了 cmake-gui 的最基本特性。处理其他杂项之前，若需要重新加载缓存值或删除缓存并从头开始，可以在文件菜单中找到“Reload Cache”和“Delete Cache”菜单项。

### 2.3.3 修改环境变量

CMake GUI 提供了一个方便的环境变量编辑器，允许对环境变量进行 CRUD 操作。要访问它，只需单击主屏幕上的 Environment…按钮。点击后，会弹出“Environment Editor”窗口：

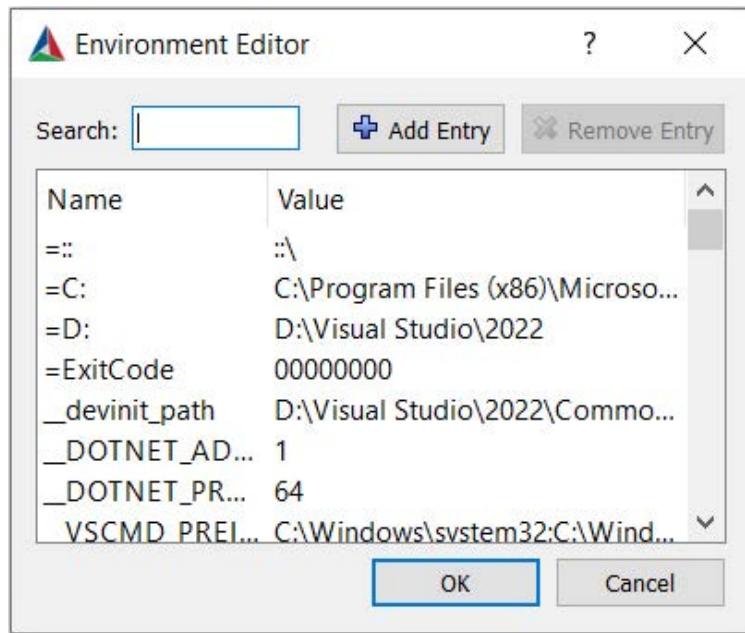


图 2.33 CMake GUI 环境变量编辑器

Environment Editor 窗口包含当前环境中存在的环境变量列表。要编辑环境变量，双击表中所需环境变量的值字段。该窗口还允许通过添加条目和删除条目按钮添加和删除信息。

### 2.3.4 对正则表达式求值

有没有想过适用正则表达式，在 CMake 中会得到什么样的结果？若有这样的想法，可能已经通过 `message()` 打印了 `regex` 匹配结果的变量。这里来看下 CMake GUI 的正则表达式资源管理器工具。

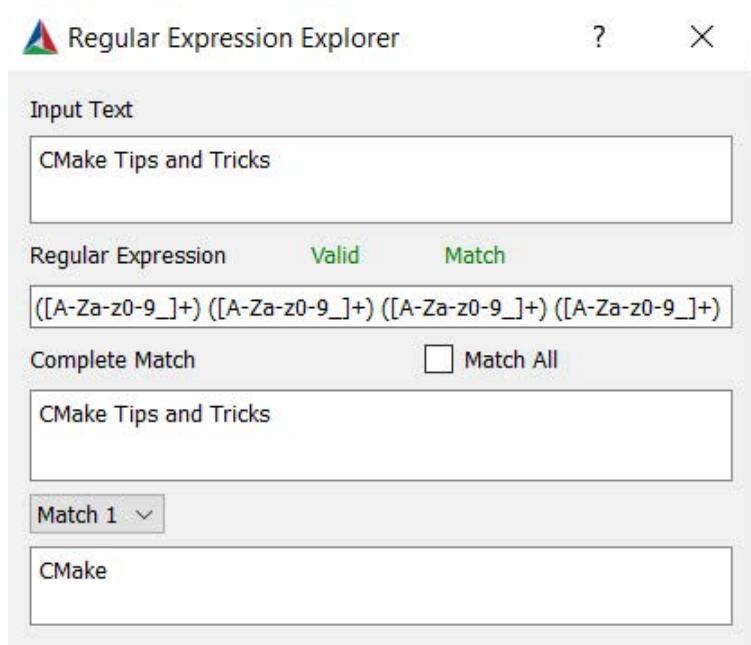


图 2.34 CMake GUI 正则表达式资源管理器

这个隐藏的工具可以使用 CMake 的 `regex` 引擎调试正则表达式，其位于“工具”菜单中，名称为 Regular Expressions Explorer…，使用起来非常简单：

1. 在正则表达式字段中输入表达式。  
该工具将检查表达式是否有效。若有效，屏幕上的有效文本将是绿色的。若 CMake 的 `regex` 引擎不喜欢你给出的表达式，它会变成红色。
2. 在 Input Text 字段中输入测试字符串。正则表达式将与此文本匹配。
3. 若有匹配，窗口上的 match 将从红色变为绿色。匹配的字符串将在完全匹配字段中输出。
4. 匹配时，捕获组将被分配给匹配 1、匹配 2 和匹配 N(有的话)。

本节中，我们学习了如何使用 CMake 的本机图形界面。接下来，通过了解 CMake 的 IDE 和编辑器集成来继续学习 CMake 的使用。

## 2.4. Visual Studio、Visual Studio Code 和 Qt Creator 中使用 CMake

作为软件开发中的常用工具，CMake 可以与各种 IDE 和源代码编辑器集成。使用 IDE 或编辑器的同时，这样的集成对用户来说可能会更方便。本节中，将介绍 CMake 如何与一些主流 IDE 和编辑器的集成。

本节的主要重点是研究和学习 CMake 与这些工具的集成。本节假设读者已有使用将要交互的 IDE/编辑器的经验。

先从 Visual Studio 开始吧！

### 2.4.1 Visual Studio

与其他流行的 IDE 不同，Visual Studio 直到 2017 年才原生支持 CMake。那一年，微软决定引入了对处理 CMake 项目的内置支持，并发布了 Visual Studio 2017。从那时起，内置支持 CMake 就成为了 Visual Studio IDE 的一个特性。

开始介绍前，安装 Visual Studio 2017 或更高版本，对于老版本的 Visual Studio 没有这个特性。我们的例子中，将使用 Visual Studio 2022 社区版。

## 创建一个 CMake 项目

Visual Studio 项目可以基于模板创建。在 VS2017 及以上版本中，有一个 CMake 项目模板。我们将学习如何使用这个模板来创建新的 CMake 项目。

要用 Visual Studio 创建一个新的 CMake 项目，请单击欢迎页面上的 **Create a new project** 按钮。或者，可以通过在主 IDE 窗口中单击 **File | New | Project** 来访问，或者使用 **Ctrl + Shift + N (New Project)** 快捷键。VS2022 的欢迎界面是这样的：

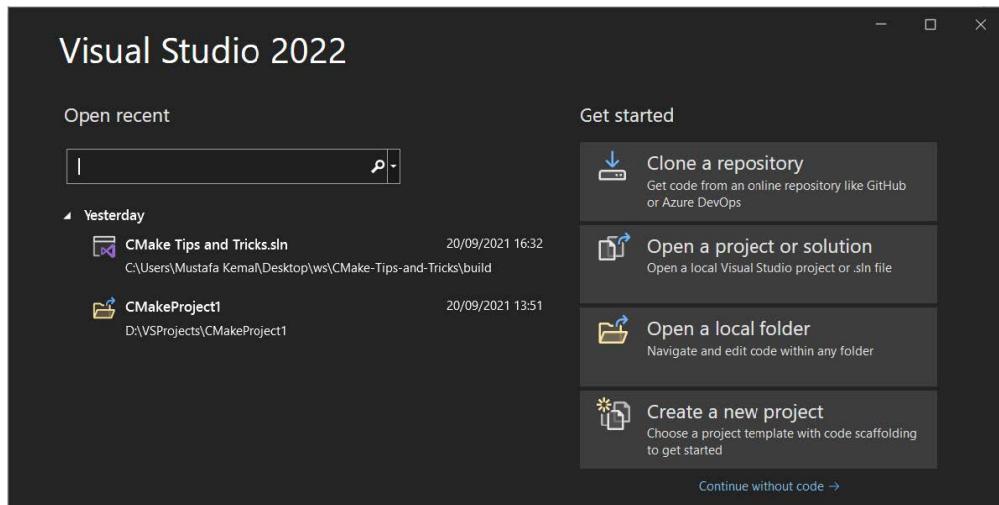


图 2.35 Visual Studio 2022 欢迎界面

在“创建新项目”窗口上，双击项目模板列表中的 CMake project。可以使用位于列表顶部的搜索栏来筛选项目模板：

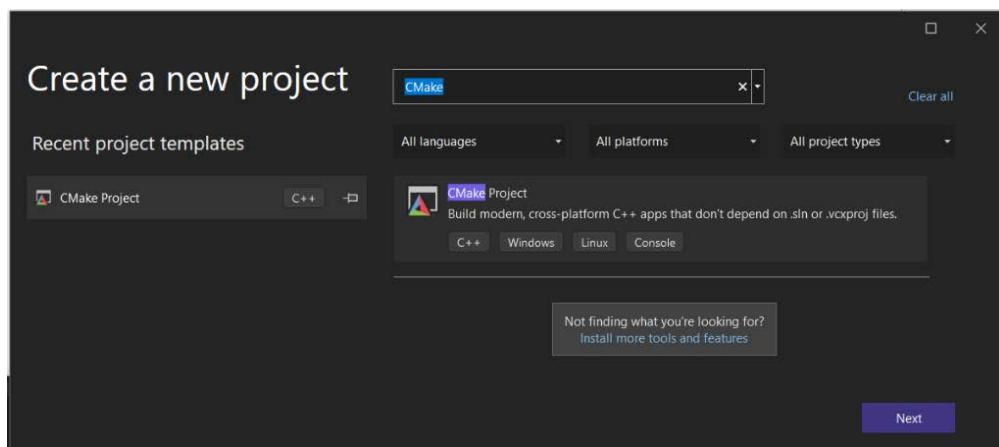


图 2.36 - Visual Studio 2022 创建新项目界面

单击 **Next** 之后，将出现项目配置窗口。可以给 CMake 项目起一个名字，并选择将新项目放在哪里。示例中，将使用默认项目名称 **CMakeProject1**。

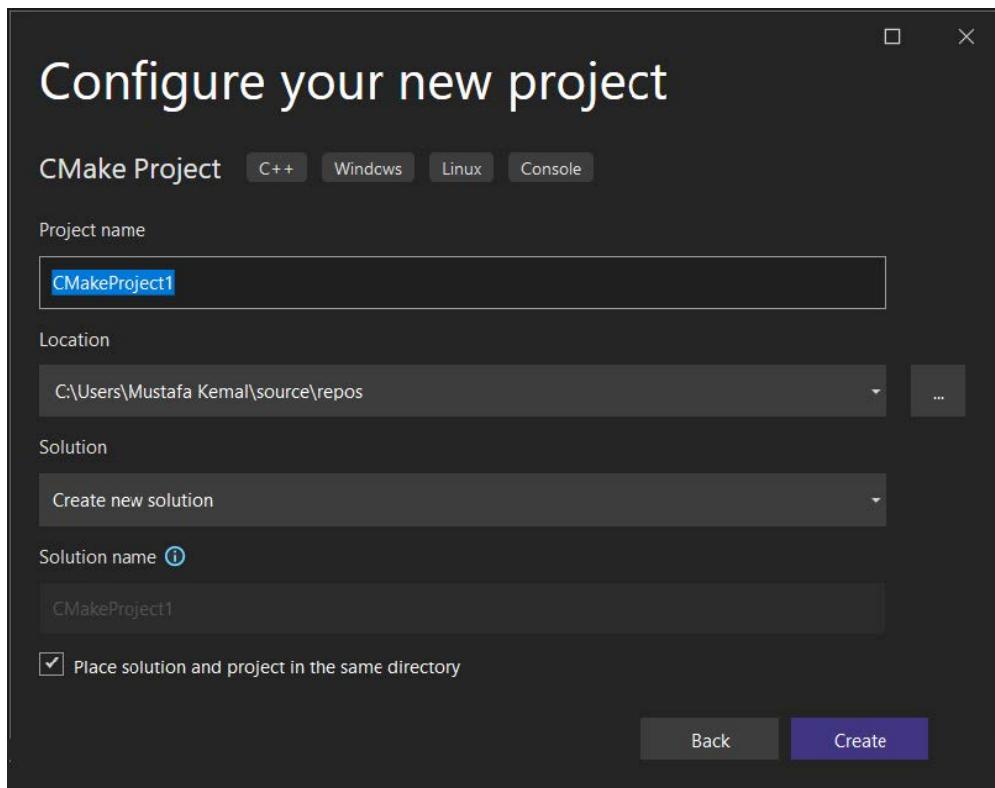


图 2.37 Visual Studio 2022 新项目配置界面

填写详细信息后，单击 Create 创建新的 CMake 项目。生成的项目将包含一个 CMakeLists.txt 文件，一个 C++ 源文件和一个 C++ 头文件，以所选的项目名称命名。新建的项目布局如下图所示：

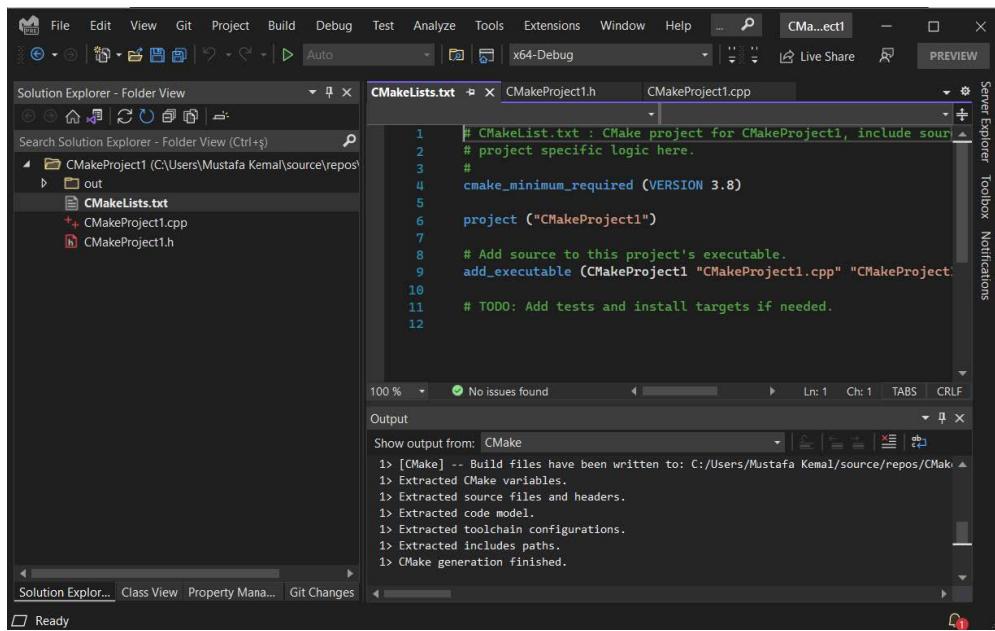


图 2.38 用 Visual Studio 创建的新的 CMake 项目

## 打开已有的 CMake 项目

打开现有的 CMake 项目，File | Open | CMake... 并选择待打开项目的顶层 CMakeLists.txt 文件。下图显示了“打开”菜单的样子：

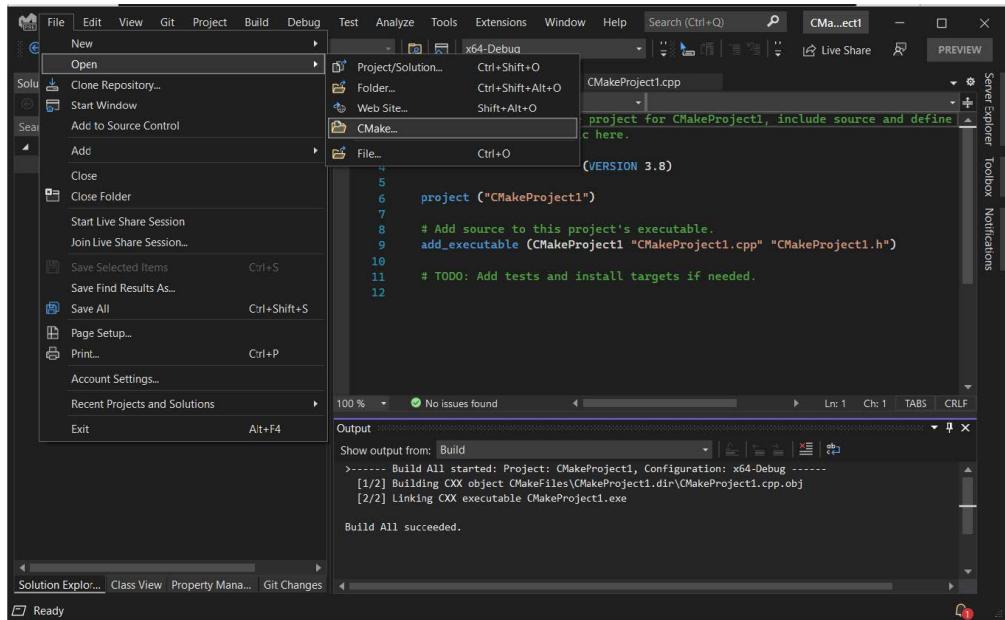


图 2.39 CMake 项目打开菜单

接下来，来看看如何配置和构建 CMake 项目。

### 配置和构建 CMake 项目

要在 Visual Studio 中构建 CMake 项目，请先转到 Project | Configure。这将调用 CMake 配置步骤并生成所需的构建系统文件。配置完成后，单击 Build | Build All 以生成项目。也可以通过 F7 快捷键来触发 Build All。

注意，当保存 CMakeLists.txt 文件时，Visual Studio 将自动重新配置，该文件是项目的一部分。

### 使用 CMake 目标执行通用操作

Visual Studio 使用启动目标的概念来执行目标所需的操作，比如构建、调试和启动。要将 CMake 目标设置为启动目标，请使用工具栏上的“选择启动目标”下拉框。Visual Studio 将自动在配置中使用 CMake 目标填充这个下拉框。

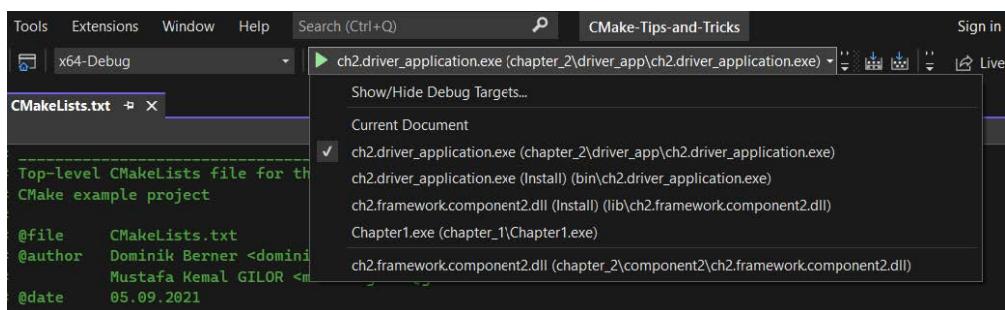


图 2.40 启动目标的下拉菜单

设置启动目标后，可以使用诸如调试、构建或启动等操作：

- 要调试，首先单击“Debug | Startup Target”，然后单击“Debug | Start Debugging”或使用 F5 快捷键。
- 不调试的情况下启动，单击“Start without debug”或使用 Ctrl + F5 快捷键。

- 要构建，单击 Build，或单击 Build | Build <target>，或使用 Ctrl + B 键盘快捷键。
- 按钮位置如下图所示：



图 2.41 工具栏按钮的位置

本节中，已经介绍了 Visual Studio CMake 集成的基础知识。下一节中，将继续了解另一个微软的产品，Visual Studio Code。

#### 2.4.2 Visual Studio Code

Visual Studio Code (VSCode) 是微软开发的一款开源代码编辑器。它不是一个 IDE，但可以很强大，并通过扩展具有类似 IDE 的特性。扩展市场有各种各样的附加内容，从主题到语言服务器。可以找到几乎所有东西的扩展，这使 VSCode 既强大又讨喜，VSCode 也有一个官方的 CMake 扩展。这个扩展最初由 Colby Pike 开发 (也称为 vector-of-bool)，但现在由微软官方维护。

本节中，将学习如何安装扩展，并使用 VS Code 执行 CMake。

先确认 VSCode 必须已经安装在您的环境中。若还没装，请访问<https://code.visualstudio.com/learn/get-started/basics>了解下载和安装的详细信息。

此外，我们将经常访问命令面板。强烈建议先来熟悉一下命令面板，下面是截图：

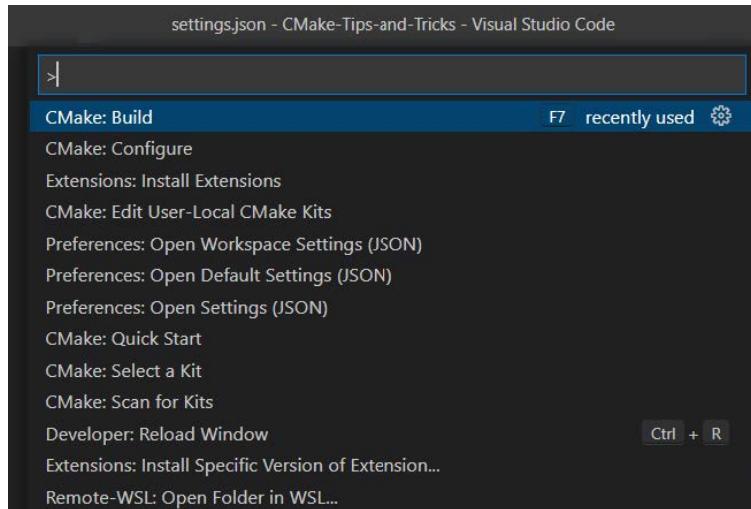


图 2.42 VSCode 的命令面板

访问命令面板的快捷键是 F1 和 Ctrl + Shift + P。

#### 安装扩展

安装扩展是非常直接和简单的。要使用 CLI 安装它，适用以下命令 (若使用的是 Insiders 版本，则用 code-insiders 替换 code)：

```
code --install-extension ms-vscode.cmake-tools
```

也可以在 VSCode GUI 中做同样的事情。打开 VSCode，单击左侧导航窗格上的 Extensions 导航到 Extensions 页面。或者，可以使用 Ctrl + Shift + X 快捷键。在扩展搜索框中键入 CMake 工具，选择 CMake 工具。注意不要将它与 CMake 扩展混淆，然后安装。

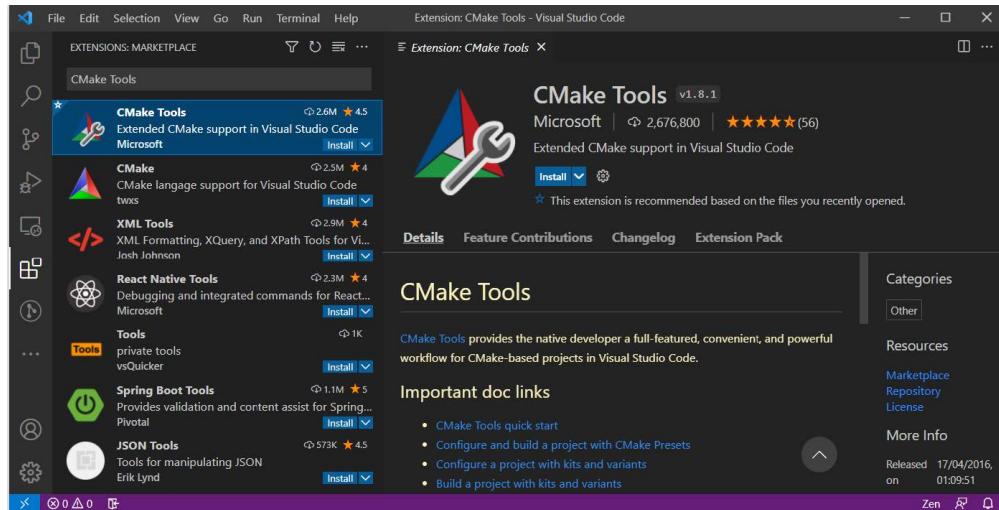


图 2.43 VSCode 扩展市场

安装完成后，就可以使用了。

## 快速启动

VSCode CMake 工具扩展提供了一个快速启动选项，可以引导一个 CMake 项目，使用示例 C++ 代码。要使用它，首先使用 File | Open Folder… 打开目标文件夹，然后按 F1，并键入 cmake quick start。选择“CMake: Quick Start”，按“Enter”键。

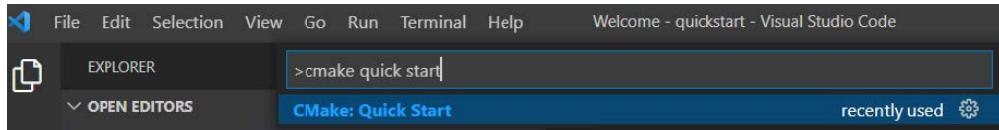


图 2.44 命令面板——定位 CMake: 快速入门

首先，扩展会询问使用哪个套件，并选择适合新项目的选项。

选择工具包之后，系统会要求输入项目名称，这将是顶层 CMake 项目的名称。

最后，将显示选择一个示例应用程序，将会要求创建一个可执行的应用程序项目或一个库项目。选择其中之一，然后创建 CMake 项目，CMakeLists.txt 和 main.cpp 文件将自动生成。

## 打开现有的项目

VSCode 中打开 CMake 项目没什么特别。打开包含顶层 CMakeLists.txt 文件的文件夹，CMake 工具扩展将自动识别该文件夹为 CMake 项目，所有 CMake 相关的命令都可以在 VSCode 命令面板上可使用。

## 配置、构建和清理项目

要配置 CMake 项目，从命令面板中选择 CMake: Configure 菜单项。构建项目，通过从命令面板中选择 CMake: Set build target 菜单项来选择一个构建目标，可以选择在调用构建时构建的内容。最

后，选择 CMake: Build 来构建选定的构建目标。要生成一个特定的目标而不将其设置为生成目标，请使用 CMake: build target 菜单项。

要清理构建工件，使用 CMake: Clean 命令面板项。这将运行 CMake 的 clean 目标，并删除所有构建工件。

## 调试目标

要调试一个目标，从命令面板中选择 CMake: Set debug target 菜单项来选择一个调试目标。会看到列出的可调试目标。

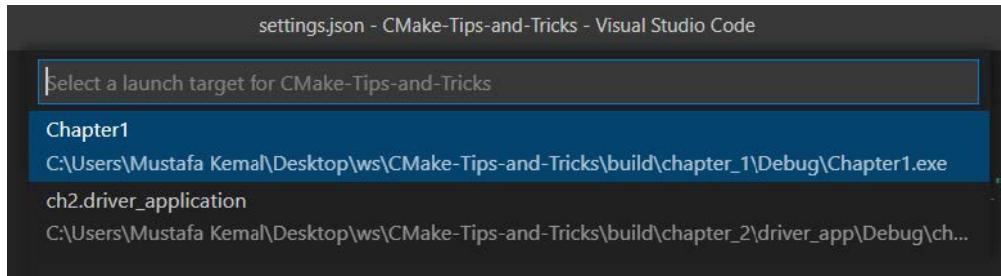


图 2.45 选择调试目标

选择目标并从命令面板中选择 CMake: Debug (Ctrl + F5)，所选目标将在调试器下启动。

若想在没有调试器的情况下运行选定的目标，选择 CMake: run without Debugging (Shift + F5)。

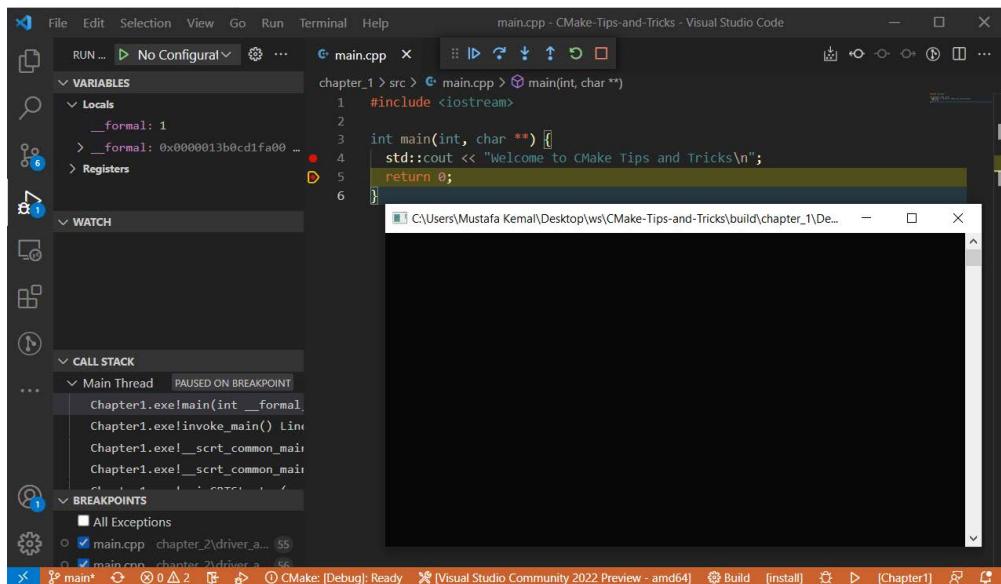


图 2.46 正在调试可执行的 chapter1

下一节中，将研究如何为已调试的目标提供参数。

## 向调试的目标传递参数

试图调试的目标可能需要命令行参数。要将命令行参数传递给调试目标，请打开 VSCode 的 settings.json 文件，并添加以下代码：

```
"cmake.debugConfig": {  
    "args": [  
        "-DNAME=Mustafa Kemal"  
    ]  
}
```

```

    "<argument1>",
    "<argument2>"
]
}

```

JSON 中的 args 数组中，可以放置目标所需的参数。这些参数将无条件地传递给所有未来的调试目标。若希望对参数进行更细粒度的控制，最好定义一个 launch.json 文件。

## 处理套件

CMake 工具扩展中的套件表示可用于构建项目的工具组合，套件这个术语就是工具链的意思。工具包使在多编译器环境中工作变得更容易，允许用户选择使用哪一个编译器。工具包可以通过扩展自动发现、从工具链文件读取或由用户手动定义。

要查看项目可用的套件，选择 CMake：从命令面板中选择套件菜单项 (F1 或 Ctrl+Shift+P)。

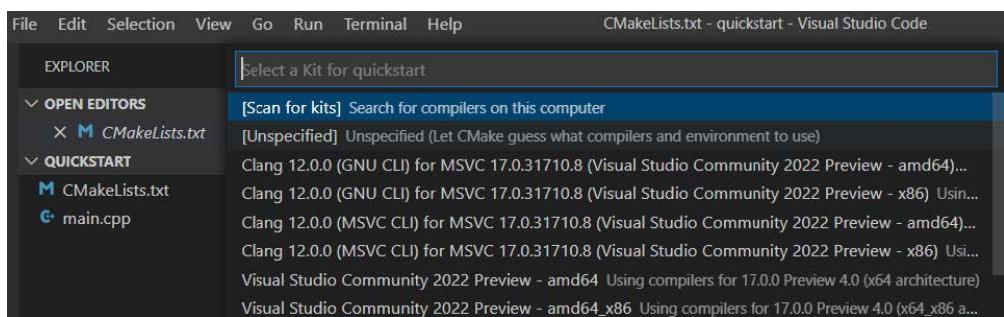


图 2.47 套件选择列表

选择的工具包将用于配置 CMake 项目，从而工具包中定义的工具将用于编译项目。套件选择将自动触发 CMake 配置。

通常，扩展自动扫描套件，工具链会作为选项列在工具包选择菜单中。若工具链没有显示在这里，意味着 CMake 工具没发现它，首先尝试重新扫描套件。若仍然没有，可以通过将它们添加到用户本地的 cmake-tools-kits.json(1) 文件中，手动定义工具包。

扩展在发现工具链方面做得很好，通常不必添加一个新工具包。这里有一个工具包模板，可以自定义并将其添加到用户本地的 cmake-tools-kits.json 文件中来定义一个新工具包。要打开用户本地套件文件，从命令面板中选择 CMake: Edit user-local CMake kits 菜单项：

```
{
  "name": "<name of the kit>",
  "compilers": {
    "CXX": "<absolute-path-to-c++-compiler>",
    "C": "<absolute-path-to-c-compiler>"
  }
}
```

#### Note

旧版本的 CMake 工具扩展，`cmake-tools-kits.json` 文件可以命名为 `cmake-kits.json`。

若工具包名称与 CMake Tools 自动生成的名称冲突，CMake Tools 将在扫描时覆盖新定义的工具包。因此，最好为工具包提供唯一的名称。

有关套件的更多信息，请参阅<https://github.com/microsoft/vscode-cmake-tools/blob/develop/docs/kits.md>。

### 2.4.3 Qt Creator

Qt Creator 是另一个支持 CMake 项目的 IDE，不需要任何额外的插件。本节中，将快速了解一下 Qt Creator 对 CMake 的支持情况。

首先要确保环境中正确安装和配置了 IDE。示例中使用的是 Qt Creator 5.0.1。

#### 加装 CMake

为了在 Qt Creator 中使用 CMake，CMake 的路径必须在 Qt Creator 中定义。要查看和定义 CMake 路径，请浏览 Tools | Options | Kits | CMake。

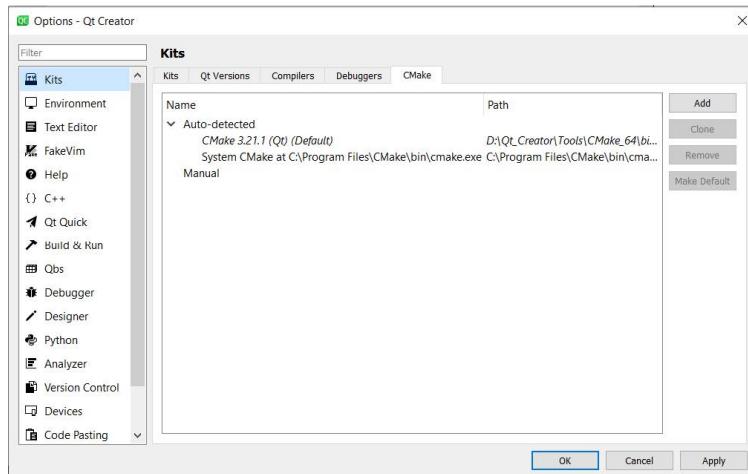


图 2.48 Qt Creator CMake 路径设置

Qt Creator 能够发现系统中的 CMake 安装。自动检测部分下的第一个是与 Qt Creator 一起提供的 CMake 可执行文件，第二个是系统的 CMake 安装。要选择在 Qt Creator 中运行的 CMake 可执行文件，请选择所需的选项并单击 Make Default 按钮。

要添加新的 CMake 可执行文件。先单击 add，这将在 Manual 部分添加一个新条目，并弹出一个窗口来填写新条目的详细信息。

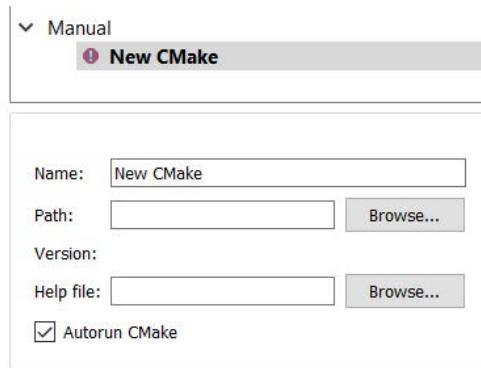


图 2.49 添加新的 CMake 可执行文件

窗口字段的详细描述如下：

- Name: 用于区分新的 CMake 可执行条目的唯一名称。
- Path: CMake 可执行路径 (CMake/CMake.exe)。
- Version: CMake 的版本 (由 Qt Creator 推导)。
- Help file: 用于可执行文件的可选 Qt Creator 帮助文件，在按 F1 时出现 CMake 帮助。
- Autorun CMake: 检查 CMakeLists.txt 文件的更改，以自动运行 CMake。

填写详细信息后，单击 Apply 将新的 CMake 可执行文件添加到 Qt Creator 中。若打算使用 Qt Creator，请将其设置为默认值。

## 创建 CMake 项目

在 Qt Creator 中创建 CMake 项目的步骤，与创建常规项目的步骤完全相同。Qt Creator 不将 CMake 视为外部构建系统生成器，其允许用户在三个构建系统生成器之间进行选择，分别是 qmake、cmake 和 qbs。

要在 Qt Creator 中创建一个 CMake 项目，点击 File | New File or project…(Ctrl + N)，从新建文件或项目窗口选择项目类型。这里使用 Qt Widgets Application 作为我们的例子。

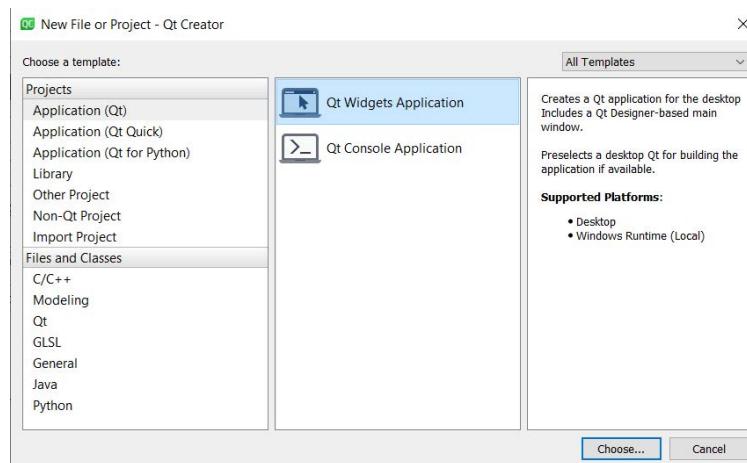


图 2.50 Qt Creator 新建文件或项目窗口

选择之后，将出现项目创建向导。根据需要填写详细信息。在 Define Build System 步骤中选择 CMake，如下图所示：

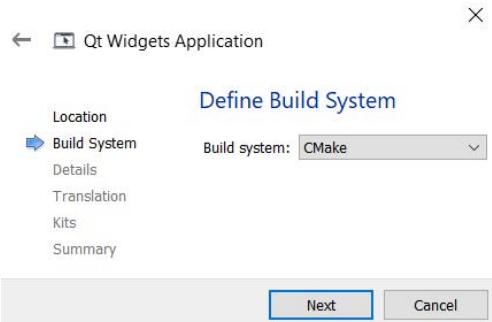


图 2.51 Qt Creator 新建项目向导构建系统选择

这样就得到了一个带有 CMake 构建系统的 Qt 应用程序。

新 CMake 项目如下图所示:

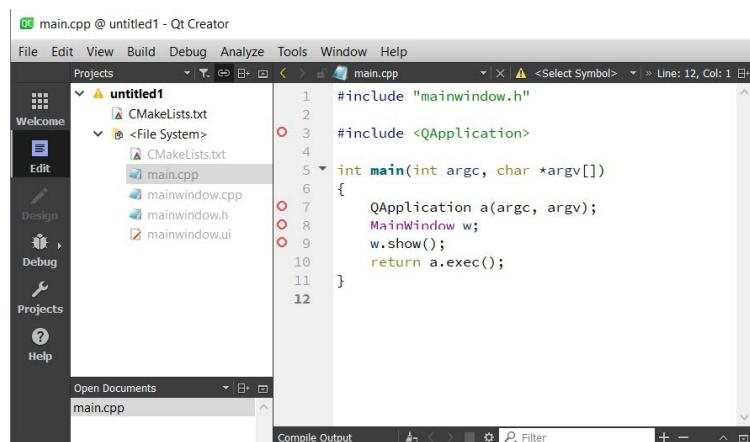


图 2.52 生成基于 CMake 的 Qt widgets 项目

## 打开现有的 CMake 项目

Qt Creator 中打开现有的 CMake 项目，通过 File | Open File or Project... (Ctrl + O)(Ctrl + O) 菜单项。选择项目的顶层 CMakeLists.txt 文件，然后单击 Open。Qt Creator 将提示，需要为项目选择一个工具包。选择相应套件，然后单击 Configure Project 按钮。将打开项目，CMake 配置步骤将与所选套件一起运行。

例如，用 Qt Creator 打开的 CMake 最佳实践项目如下图所示:

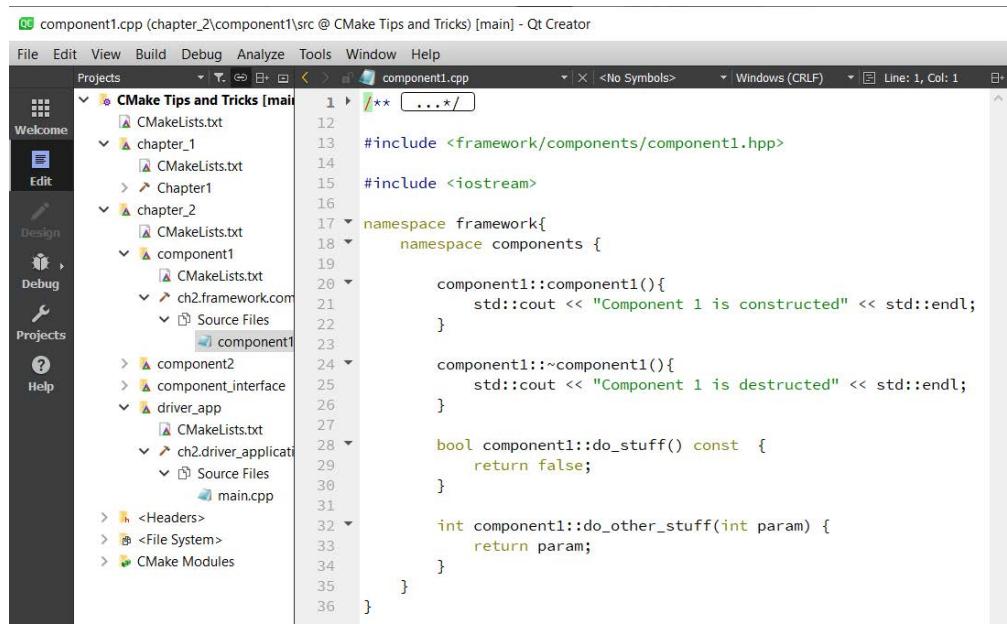


图 2.53 Qt Creator 中查看 CMake Tips and Tricks 示例项目

第一次打开 CMake 项目后，Qt Creator 会在项目的根目录下创建名为 CMakeLists.txt.user 的文件。该文件包含不能存储在 CMakeLists.txt 文件中的特定于 Qt 的详细信息，如工具包信息和编辑器设置。

## 配置和构建

大多数情况下(项目打开并保存更改到 CMakeLists.txt)，Qt Creator 将自动运行 CMake 配置，无需手动运行。要手动运行 CMake 配置，请单击“Build | Run CMake”菜单项。

配置完成后，按左上角的锤子图标来构建项目。也可以使用快捷键“Ctrl + B”。这将构建整个 CMake 项目。要只构建特定的 CMake 目标，请使用“Build”按钮旁边的“Locator”。输入 cm，然后按键盘上的空格键。



图 2.54 Qt Creator 定位器建议

定位器将显示可用于构建的 CMake 目标。通过高亮显示并按下 Enter 键来选择所需的目标，或者直接使用鼠标单击目标。

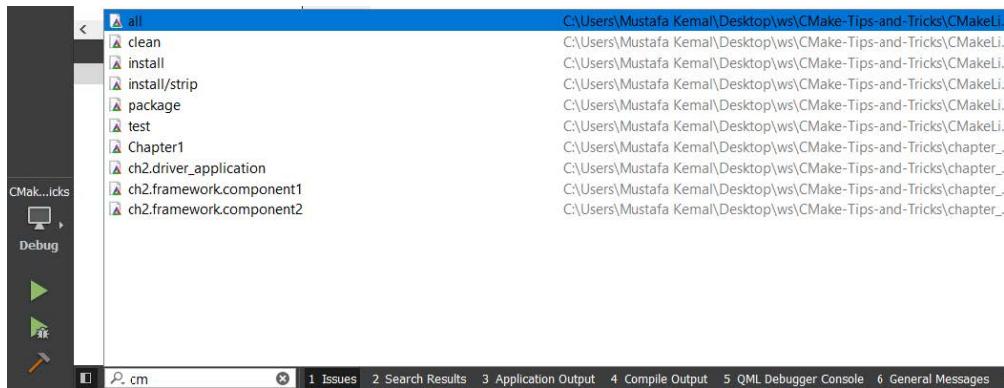


图 2.55 可构建的 CMake 目标显示在定位器上

将构建选中的 CMake 目标 (当然还有其依赖项)。

## 运行和调试

要运行或调试 CMake 目标, 请按下套件选择按钮 (左侧导航栏上的计算机图标) 并选择 CMake 目标。然后, 单击运行按钮 (工具箱选择器下的播放图标) 进行运行, 或者单击调试按钮 (带有 bug 的播放图标) 进行调试。

套件选择器菜单内容如下图所示:

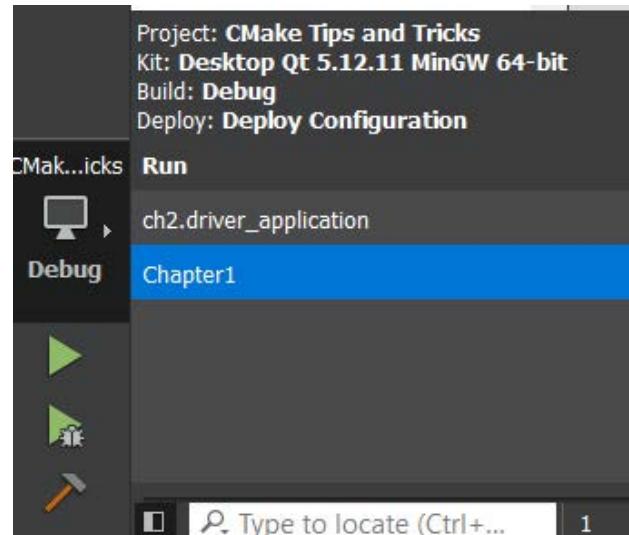


图 2.56 显示 CMake 目标的套件选择器

这里, 我们总结了在 Qt Creator 中使用 CMake 的基础知识。要了解更高级的主题, 可以参考扩展阅读中提供的资源。

## 2.5. 总结

本章中, 介绍了与 CMake 交互的基本方法, 即 CLI 和 GUI, 还介绍了与各种 IDE 和编辑器的集成。使用任何一种工具都需要了解如何与之交互。学习交互方式可以让我们更好地利用工具, 更容易地达到我们的目标。

下一章中, 将讨论 CMake 项目的构建模块, 将从头开始创建一个结构良好、可用于生产的 CMake 项目。

## 2.6. 练习题

为了巩固你在本章所学到的知识，试着回答以下问题。若很难回答这些问题，那就回到相关的一部分，重新回顾一下相关内容：

1. 描述 CMake 项目如何使用 CLI，将项目构建目录 build 配置于项目根目录下的每一种方式：
  - A. 使用另一个 C++ 编译器，位于 /usr/bin/clang++
  - B. 使用 Ninja 生成器
  - C. Debug 编译类型需要 -Wall 编译器标志
2. 描述一下 Q1 中之前配置的项目是如何使用 CMake 使用以下命令行构建的：
  - A. 8 个并行任务
  - B. Unix Makefiles 生成器中的 --trace 选项
3. 描述如何使用 CMake 使用命令行在 /opt/project 下安装 Q1 中构建的项目。
4. 假设已经配置并构建了 CMake-Best-Practices 项目，使用哪个命令能只安装 ch2.libraries 组件？
5. CMake 中的“高级”变量是什么？

## 2.7. 扩展阅读

关于本章中讨论的主题，有很多指南和文档。可以列出一个推荐清单：

- CMake CLI 文档: <https://cmake.org/cmake/help/latest/manual/cmake.1.html>。
- 在 Visual Studio 中配置 CMake 项目: <https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=msvc-160>。
- Visual Studio 支持的 CMake: <https://devblogs.microsoft.com/cppblog/cmake-support-in-visual-studio/>。
- VSCode 的 CMake 工具扩展: <https://devblogs.microsoft.com/cppblog/cmake-tools-extension-for-visual-studio-code/>。
- VSCode CMake 工具文档: <https://github.com/microsoft/vscode-cmake-tools/tree/develop/docs#cmake-tools-for-visual-studio-code-documentation>。
- VSCode 调试: <https://code.visualstudio.com/docs/editor/debugging>。
- Qt Creator 定位器使用指南: <https://doc.qt.io/qtcreator/creatoreditor-locator.html>。
- Qt Creator 用户接口: <https://doc.qt.io/qtcreator/creatorquick-tour.html>。

# 第 3 章 创建 CMake 项目

了解了如何使用 CMake 及其基本概念，我们只看到了 CMake 如何与已经存在的代码一起使用，但更有趣的部分是使用 CMake 构建应用程序。本章中，将了解如何构建可执行文件和库，以及如何将一起使用。我们将深入了解如何创建不同种类的库，介绍一些关于如何构造 CMake 项目的良好实践。由于库通常附带各种编译器设置，需要了解如何设置，并在必要时传递给依赖库。由于项目中的依赖关系会变得相当复杂，还要了解如何可视化不同目标之间的依赖关系。

本章中，将讨论以下主题：

- 建立项目
- 创建“hello world”可执行文件
- 创建库
- 结合在一起使用

## 3.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本
- 本章的示例和源代码都可以在本书的 GitHub 库中找到（在 chapter03 子文件夹中），<https://github.com/PacktPublishing/CMake-Best-Practices>。

## 3.2. 创建项目

虽然 CMake 可以用于任何文件结构，但有一些关于如何组织文件的良好实践。本书中的例子使用了以下模式：

```
|── CMakeLists.txt  
|── build  
|── include/project_name  
└── src
```

最小的项目结构中有三个文件夹和一个文件。

- build：放置构建文件和二进制文件的文件夹。
- include/project\_name：此文件夹包含从项目外部公开访问的所有头文件，包含 `<project_name/somefile.h>` 使它更容易看出头文件来自哪个库。
- src：此文件夹包含所有私有的源文件和头文件。
- CMakeLists.txt：这是主 CMake 文件。

构建文件夹可以放置在任何地方，放在项目根目录最方便，但强烈建议不要选择任何非空文件夹作为构建文件夹。特别是将构建好的文件放入 `include` 或 `src` 中，这是一种糟糕的实践。其他文件夹，如 `test` 或 `doc`，在组织测试项目和文档页面时就很方便。

### 3.2.1 嵌套项目

将项目相互嵌套时，每个项目都应该映射上面的文件结构，并且应该编写每个 `CMakeLists.txt` 以便子项目可以独立构建。每个子项目的 `CMakeLists.txt` 文件应该指定 `cmake_minimum_required`，以及可选的项目定义。我们将在第 9 章中深入介绍大型项目和超级构建。

嵌套项目看起来像这样：

```
├── CMakeLists.txt
├── build
└── include/project_name
├── src
└── subproject
    ├── CMakeLists.txt
    ├── include
    │   └── subproject
    └── src
```

文件夹结构在子项目文件夹中重复。坚持这样的文件夹结构，并使子项目能够独立构建，这样就更容易移植项目。其还允许只构建项目的一部分，这对于构建时间可能相当长的大型项目非常有用。

现在已经了解了文件结构，让我们创建一个没有特殊依赖的可执行文件。本章的后面，我们将创建各种各样的库，并将它们集合在一起。

## 3.3. 创建“hello world”可执行文件

首先，从一个简单的 `hello world` C++ 程序创建一个简单的可执行文件。下面的 C++ 程序将打印出 `Welcome to CMake Best Practices`：

```
1 #include <iostream>
2 int main(int, char **)
3 {
4     std::cout << "Welcome to CMake Best Practices\n";
5     return 0;
}
```

要构建此文件，需要编译它并为可执行文件指定一个名称。来看看 `CMakeLists.txt` 文件是怎样构建这个可执行文件的：

```
cmake_minimum_required(VERSION 3.21)
```

```

project(
    ch3_hello_world_standalone
    VERSION 1.0
    DESCRIPTION
        "A simple C++ project to demonstrate creating a standalone
        executables"
    HOMEPAGE_URL https://github.com/PacktPublishing/CMake-Best-Practices
    LANGUAGES CXX
)

add_executable(ch3_hello_world_standalone)
target_sources(ch3_hello_world_standalone PRIVATE src/main.cpp)

```

第一行，`cmake_minimum_required(VERSION 3.21)`，期望看到的 CMake 的版本，以及 CMake 将启用哪些特性。本书的例子中都使用 CMake 3.21，但是出于兼容性的原因，读者们可以选择一个较低的版本。

对于本例，3.1 版本将是绝对的最小值，因为在此之前，`target_sources` 不可用。将 `cmake_minimum_required` 指令放在每个 CMakeLists.txt 文件的顶部是一个很好的做法。

接下来，使用 `project()` 指令设置项目。第一个参数是项目的名称——我们的例子中为“ch3\_hello\_world\_standalone”。

接下来，版本设置为 1.0。下面是一个简短的描述和主页的 URL。最后，`LANGUAGES CXX` 属性指定正在构建一个 C++ 项目。除了项目名称之外，所有参数都可选。

调用 `add_executable(ch3_hello_world_standalone)` 指令，会创建一个名为 `ch3_hello_world_standalone` 的目标。这也将是可执行的文件名。

现在已经创建了目标，使用 `target_sources` 完成了向目标添加 C++ 源文件。`ch3_hello_world_standalone` 是目标名，在 `add_executable` 中指定。`PRIVATE` 定义源仅用于构建此目标，而不用于依赖的目标。在范围说明符之后，有一个相对于当前 CMakeLists.txt 文件路径的源文件列表。如果需要，当前处理的 CMakeLists.txt 文件的位置可以通过 `CMAKE_CURRENT_SOURCE_DIR` 得到。

源码可以直接添加到 `add_executable`，也可以单独使用 `target_sources`，将它们与 `target_sources` 一起添加。通过使用 `PRIVATE`、`PUBLIC` 或 `INTERFACE`，可以显式地定义在何处使用源码。但是，指定 `PRIVATE` 以外的内容只对库目标有意义。

经常看到的一种常见模式是用项目名称来命名项目的可执行文件：

```

project(hello_world
...
)

add_executable(${PROJECT_NAME})

```

乍一看这似乎很方便，但不推荐这样做。项目名称和目标具有不同的语义含义，因此应该将其独立对待，因此应该避免使用 PROJECT\_NAME 作为目标的名称。

可执行文件非常重要，而且非常容易创建，除非构建的是一个庞然大物，否则应该使用库来模块化和分发代码。下一节中，我们将学习如何构建库，以及如何处理不同的链接方法。

## 3.4. 创建库

创建库的工作方式与创建可执行文件的方式类似，但因为库目标通常是由其他目标使用的，要么在同一个项目中，要么由其他项目使用。因为库通常有一个内部和公开可见的 API，所以在向项目添加文件时必须考虑到这一点。

简单的库项目可以这样：

```
cmake_minimum_required(VERSION 3.21)

project(
    ch3_hello
    VERSION 1.0.0
    DESCRIPTION
        "A simple C++ project to demonstrate creating executables
        and libraries in CMake"
    LANGUAGES CXX)

add_library(hello)

target_sources(
    ch3_hello
    PRIVATE src/hello.cpp src/internal.cpp)

target_compile_features(ch3_hello PUBLIC cxx_std_17)

target_include_directories(
    ch3_hello
    PRIVATE src/hello
    PUBLIC include)
```

同样，该文件以设置 cmake\_minimum\_required 和项目信息开始。

接下来，使用 add\_library 创建库的目标——本例中，库的类型没有确定。可以传递 STATIC 或 SHARED 来显式确定库的类型，这里可以省略设置该类型，我们允许库的使用者选择如何构建和链接。通常，静态库很容易处理。

若省略了库的类型，则 BUILD\_SHARED\_LIBS 将决定库是默认构建为动态库还是静态库。这个变量不应该在项目的 CMake 文件中设置，应该由构建者传递。

使用 `target_sources` 为库添加源文件。第一个参数是目标名称，后面跟 PRIVATE、PUBLIC 或 INTERFACE 关键字分隔相应源文件。实践中，源文件使用 PRIVATE 添加，PRIVATE 和 PUBLIC 关键字指定在何处使用源代码进行编译。PRIVATE 指定的源文件将只在目标 `ch3_hello` 中使用。若使用 PUBLIC，那么源文件也会将附加到 `ch3_hello` 和依赖 `ch3_hello` 的目标上，这通常不是我们想要的结果。INTERFACE 关键字说明源文件不会添加到 `ch3_hello` 目标中，而是会添加到依赖到 `ch3_hello` 的目标上。通常，为目标指定为 PRIVATE 的内容都可以视为构建需求。最后，包含目录使用 `target_include_directories` 设置。该指令指定的文件夹内的所有文件都可以使用 `#include <file.hpp>`(带尖括号) 来访问。

PRIVATE 包含不会包含目标属性中的路径，也就是 `INTERFACE_INCLUDE_DIRECTORIES`。当目标依赖于该库时，CMake 将读取该属性以确定哪些包含目录可见。

由于标准库的 C++ 代码使用了与现代版本 C++ 绑定的特性，如 C++11/14/17/20 或 C++23(即将发布)，必须设置 `cxx_std_17` 属性。由于用于编译库本身并与需要其接口，因此将其设置为 PUBLIC。只有在头文件包含需要特定标准的代码时，才需要将其设置为 PUBLIC 或 INTERFACE。若只有内部代码依赖于某个标准，则首选将其设置为 PRIVATE。通常，尽量将公共 C++ 标准设置为最低的可用标准，也可以只启用现代 C++ 标准的某些特性。

完整的可用编译特性列表可参阅[https://cmake.org/cmake/help/latest/prop\\_gbl/CMAKE\\_CXX\\_KNOWN\\_FEATURES.html](https://cmake.org/cmake/help/latest/prop_gbl/CMAKE_CXX_KNOWN_FEATURES.html)。

### 3.4.1 命名库

当使用 `add_library(<name>)` 创建库时，库名称在项目中必须全局唯一。默认情况下，库的实际文件名是根据平台上的约定构造的，例如 `lib<name>`。在 Linux 上为 `<name>.lib`，在 Windows 上为 `<名称>.dll`。通过设置目标的 `OUTPUT_NAME` 属性，可以更改文件的名称。可以在下面的例子中看到，输出文件的名称已经从 `ch3_hello` 改为 `hello`:

```
add_library(ch3_hello)

set_target_properties(
    ch3_hello
    PROPERTIES OUTPUT_NAME hello
)
```

避免使用 `lib` 前缀或后缀的库名称，因为 CMake 可能在文件名后面或前面添加适当的字符串。当然，这取决于平台。

动态库的常用命名约定是在文件名中添加版本以指定构建版本和 API 版本，通过指定 `VERSION` 和 `SOVERSION` 属性，CMake 将在构建和安装库时创建必要的文件名和符号链接:

```
set_target_properties(
    hello
    PROPERTIES VERSION ${PROJECT_VERSION} # Contains 1.2.3
    SOVERSION ${PROJECT_VERSION_MAJOR} # Contains only 1
)
```

Linux 上，这个示例将会使 libhello.so.1.0.0 的文件名带有来自 libhello 的符号链接。所以，libhello.so.1 指向实际的库文件。下面的截图展示了生成的文件和指向它的符号链接：

```
lrwxrwxrwx 1 conan      14 Sep 30 19:51 libhellod.so -> libhellod.so.1
lrwxrwxrwx 1 conan      18 Sep 30 19:51 libhellod.so.1 -> libhellod.so.1.0.0
-rwxr-xr-x 1 conan 91912 Sep 30 19:51 libhellod.so.1.0.0
```

图 3.1 使用 SOVERSION 属性构建库文件和生成的符号链接

项目中经常看到的另一种约定，是为各种构建配置的文件名添加不同的后缀。CMake 通过设置 CMAKE\_<CONFIG>\_POSTFIX 全局变量或添加 <CONFIG>\_POSTFIX 属性来处理这个问题。若设置了此变量，后缀将自动添加到非可执行目标。与大多数全局变量一样，应该通过命令行传递给 CMake，或者作为预置，而不是硬编码在 CMakeLists.txt 文件中。

调试库的后缀也可以显式地设置为单个目标，如下面的例子所示：

```
set_target_properties(
    hello
    PROPERTIES DEBUG_POSTFIX d)
```

这将使库文件和符号链接命名为 libhellod。由于在 CMake 中链接库是针对目标而不是文件名进行的，因此会自动选择正确的文件名。然而，链接动态库时要注意的是符号可见性。

### 3.4.2 动态库中的符号可见性

要链接到动态库，链接器必须知道哪些符号可以从库外部使用。这些符号可以是类、函数、类型等，使它们可见的过程称为导出。

指定符号可见性时，编译器有不同的方式和默认行为，这使得以独立于平台的方式指定符号可见性有点麻烦。从默认的编译器可见性开始；gcc 和 clang 假设所有的符号都是可见的，而 Visual Studio 编译器默认情况下会隐藏所有的符号，除非显式导出。设置 CMAKE\_WINDOWS\_EXPORT\_ALL\_SYMBOLS，可以改变 MSVC 的默认行为，这是一种暴力的解决方法，只有当库的所有符号都应该导出时才能使用。

虽然将所有的符号设置为公共可见是确保链接容易的一种简单方法，但也有一些缺点：

- 通过导出所有内容，就无法阻止依赖目标使用内部代码。
- 外部代码使用每个符号，链接器不能丢弃死代码，因此库体积往往会上膨胀。若库中包含模板，则尤其如此，因为模板往往会大幅增加符号的数量。
- 由于导出了每个符号，关于哪些应该隐藏或内部符号的唯一线索必须来自文档。
- 暴露库的内部符号会暴露本应隐藏的东西。

#### 所有符号可见

将动态库中的所有符号设置为可见时要小心，特别是在关心安全问题或二进制文件的大小很重要时。

#### 更改默认可见性

要更改符号的默认可见性，请将 `<LANG>_VISIBILITY_PRESET` 属性设置为 `HIDDEN`。此属性可以全局设置，也可以针对单个库目标设置。`<LANG>` 会替换为编写库的语言，例如：CXX 替换为 C++，C 替换为 C。若所有要导出的符号都是隐藏符号，必须在代码中特别标记。最常见的方法是指定一个预处理器定义来确定一个符号是否可见：

```
1 class HELLO_EXPORT Hello {
2 ...
3 };
```

`HELLO_EXPORT` 将包含这样的信息：当编译库时，该符号是否导出，或者当对库进行链接时，它是否应该导入。GCC 和 Clang 使用 `_attribute_(…)` 关键字来确定此行为，而在 Windows 上使用 `_declspec(…)`。编写以跨平台方式处理此问题的头文件并不是一项轻松的任务，特别是若还要考虑库可能构建为静态库和对象库。幸运的是，CMake 提供了 `generate_export_header` 宏，由 `GenerateExportHeader` 模块导入。

下面的例子中，`hello` 库的符号默认设置为隐藏。然后，通过使用 `generate_export_header` 宏再次单独启用。另外，本例将 `VISIBILITY_INLINES_HIDDEN` 属性设置为 `TRUE`，通过隐藏内联类成员函数进一步减少导出的符号表。并不严格要求设置内联的可见性，但通常在设置了默认可见性后才会这样做：

```
add_library(ch3_hello_shared SHARED)
set_property(
    TARGET ch3_hello_shared
    PROPERTY CXX_VISIBILITY_PRESET "hidden")
set_property(
    TARGET ch3_hello_shared
    PROPERTY VISIBILITY_INLINES_HIDDEN TRUE)
include(GenerateExportHeader)
generate_export_header(
    ch3_hello_shared
    EXPORT_FILE_NAME
    export/hello/export_hello.hpp)

target_include_directories(
    ch3_hello_shared PUBLIC
    "${CMAKE_CURRENT_BINARY_DIR}/export")
```

`generate_export_header` 的调用在 `CMAKE_CURRENT_BINARY_DIR/export/hello` 目录中创建了一个名为 `export_hello.hpp` 的文件，该文件可以包含在库的文件中。将这些生成的文件放在构建目录的子文件夹中是一个好做法，这样只将目录的一部分添加到包含路径中。生成文件的 `include` 结构应该与库的其他部分的 `include` 结构匹配。在这个例子中，通过 `#include<hello_public_header.h>` 来包含所有公共头文件，那么导出头文件也应该放在名为 `hello` 的文件夹中。生成的文件也必须添加到安装说明中。此外，要创建导出文件，必须将用于导出符号的必要编译器标志设置为目标。

因为生成的头文件必须包含在声明要导出的类、函数和类型的文件中，所以 CMAKE\_CURRENT\_BINARY\_DIR/export 添加到 target\_include\_directories 中。注意，必须是 PUBLIC，这样依赖库才可以找到该文件。

关于 generate\_export\_header 宏还有很多选项，但本节中看到的内容涵盖了大多数用例。关于设置符号可见性的其他信息，可以查阅官方 CMake 文档<https://cmake.org/cmake/help/latest/module/GenerateExportHeader.html>。

### 3.4.3 接口或纯头文件库

纯头文件的库有点特殊，因为不需要编译；相反，可以导出它们的头文件，以便直接包含在其他库中。大多数情况下，头文件库的工作方式与普通库类似，但是头文件使用 INTERFACE，而非 PUBLIC。

由于仅包含头文件的库不需要编译，因此不会向目标添加源文件。下面的代码创建了一个小的纯头文件库：

```
project(
    ch3_hello_header_only
    VERSION 1.0
    DESCRIPTION "Chapter 3 header only example"
    LANGUAGES CXX)

add_library(hello_header_only INTERFACE)
target_include_directories(hello_header_only INTERFACE include/)
target_compile_features(hello_header_only INTERFACE cxx_std_17)
```

CMake 3.19 版本之前，INTERFACE 库不能使用 target\_sources。现在，纯头文件库可以不列出源文件。

## 对象库——仅供内部使用

有时，可能想要分离代码，以便部分代码可以重用，而不需要创建完整的库。当想在可执行测试和单元测试中使用某些代码时，通常的做法是不需要重新编译所有代码两次。为此，CMake 提供了对象库，其中的源代码是编译的，但不进行归档或链接。通过 add\_library(MyLibrary object) 创建对象库。

自 CMake 3.12 起，这些对象可以像普通库一样使用，只需将它们添加到 target\_link\_libraries 函数中。3.12 版本之前，对象库需要添加生成器表达式，也就是 \$<TARGET\_OBJECTS:MyLibrary>。这将在生成构建系统期间扩展为一个对象列表。这种方式现在还可以用，但不推荐这样做，因为这很快就变得不可维护，特别是在一个项目中有多个对象库的情况下。

### 何时使用对象库

对象库可以在不公开模块的情况下加快代码的构建和模块化。

使用对象库，可以覆盖所有不同类型的库。编写和维护库本身很有趣，但除非将其集成到更大的项目中，否则它们什么也做不了。那么，让我们看看如何在可执行文件中，使用目前定义的库。

## 3.5. 将它们结合在一起——使用库

我们已经创建了三个不同的库——一个静态或动态库，一个接口或头文件库，以及一个预编译但没有链接的对象库。

了解如何在共享项目的可执行文件中使用它们。将它们安装为系统库或使用它们作为外部依赖，将在第 5 章中介绍。

所以，可以把 `add_library` 放在同一个 `CMakeLists.txt` 文件中，或者使用 `add_subdirectory` 将其整合起来。两者都是有效的选项，并取决于项目的设置方式。

下面的例子中，假设在 `hello_lib`、`hello_header_only` 和 `hello_object` 目录中已经用 `CMakeLists.txt` 文件定义了三个库。可以使用 `add_subdirectory` 包含这些库。这里，创建了一个名为 `chapter3` 的新目标，它是可执行文件。然后，将这些库用 `target_link_libraries` 添加到可执行文件中：

```
add_subdirectory(hello_lib)
add_subdirectory(hello_header_only)
add_subdirectory(hello_object)

add_executable(chapter3)
target_sources(chapter3 PRIVATE src/main.cpp)
target_link_libraries(chapter3 PRIVATE hello_header_only hello
hello_object)
```

`target_link_libraries` 的目标也可以是另一个库。同样，库的链接说明符，可以是以下任意一个：

- **PRIVATE:** 用于链接库，但不是公共接口的一部分。只有在构建目标时才需要链接库。
- **INTERFACE:** 没有链接到库，但是公共接口的一部分。当在其他地方使用目标时，链接库是必需的。这通常仅限头文件库时使用。
- **PUBLIC:** 链接到库，是公共接口的一部分。因此，该库既是构建依赖项，也是使用依赖项。

## 不推荐的做法

本书的作者极力反对下列实践，因为它们会创建难以维护的项目，使得在不同的构建环境之间难于移植。但为了完整起见，我们将它们包含进来。

除了传递 PUBLIC、PRIVATE 或 INTERFACE 之后的另一个目标，还可以传递库的完整路径或库的文件名，例如 /usr/share/lib/mylib.so，或者只是 mylib.so。这是实践可行的，但不鼓励这样做，因为这会降低 CMake 项目的可移植性。另外，也可以通过传递诸如 -nolibc 之类的链接器标志，还是不建议这么做。若所有目标都需要特殊的链接器标志，使用命令行传递是首选的方式。若单个库需要特殊标志，使用 target\_link\_options 则是首选法，最好与命令行上设置的选项结合使用。

下一节中，我们将研究如何设置编译器和链接器选项。

### 3.5.1 设置编译器和链接器选项

C++ 编译器有很多选项来设置一些常见的标志，从外部设置预处理器定义也是一种常见的做法。CMake 中，这些是使用 target\_compile\_options 传递，使用 target\_link\_options 更改链接器行为，但编译器和链接器可能有不同的设置标志的方法。例如，在 GCC 和 Clang 中，选项用减号 (-) 传递，而 Microsoft 编译器将斜杠 (/) 作为选项的前缀。但是通过第 1 章中介绍的生成器表达式，可以很容易地在 CMake 中处理这个问题：

```
target_compile_options(
    hello
    PRIVATE $<$<CXX_COMPILER_ID:MSVC>:/SomeOption>
           $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-someOption>
)
```

让我们详细了解一下生成器表达式。

\$<\$<CXX\_COMPILER\_ID:MSVC>:/SomeOption> 是一个嵌套的生成器表达式，由内而外求值。生成器表达式在生成阶段进行计算。首先，当 C++ 编译器等于 MSVC 时，\$<CXX\_COMPILER\_ID:MSVC> 为 true。若是这种情况，那么外部表达式将返回 /SomeOption，然后传递给编译器。若内部表达式的计算结果为 false，则不传递。

\$<\$<CXX\_COMPILER\_ID:GNU,Clang,AppleClang>:-fopenmp> 的工作原理类似，但不是只检查单个值，而是传递一个包含 GNU, Clang, AppleClang 的列表。若 CXX\_COMPILER\_ID 匹配其中任何一个，内部表达式计算为 true，someOption 会传递给编译器。

将编译器或链接器选项传递为 PRIVATE，将其标记为与库接口不需要的此目标的构建需求。若使用 PUBLIC，那么编译选项也成为构建需求，所有依赖于原始目标的目标将使用相同的编译选项。将编译器选项暴露给依赖的目标是需要谨慎做的事情。若编译器选项只用于使用目标而不用于构建目标，则可以使用关键字 INTERFACE。在构建纯头文件库时，这是最常见的情况。

编译器选项的特殊情况是编译定义，其会传递给底层程序。这通过 target\_compile\_definitions 进行传递。

## 调试编译器选项

要查看所有编译选项，可以查看生成的构建文件，例如 Makefiles 或 Visual Studio 项目。更方便的方法是让 CMake 将所有编译命令导出为 JSON。

通过使用 `CMAKE_EXPORT_COMPILE_COMMANDS`，将生成一个名为 `compile_commands.json` 的文件，其包含用于编译的完整命令。

启用此选项并运行 CMake 将产生如下结果：

```
{  
    "directory": "/workspaces/CMake-Best-Practices/build",  
    "command": "/usr/bin/g++ -I/workspaces/CMake-Best-Practices/  
chapter03/hello_header_only/include  
    -I/workspaces/CMake-Best-Practices/chapter03/hello_lib/include  
    -I/workspaces/CMake-Best-Practices/chapter03/hello_object_lib/include  
    -g -fopenmp -o  
chapter03/CMakeFiles/chapter03.dir/src/main.cpp.o -c /  
workspaces/CMake-Best-Practices/chapter03/src/main.cpp",  
    "file": "/workspaces/CMake-Best-Practices/chapter03/src/main.cpp"  
},
```

注意，上一个示例中添加了手动指定的-fopenMP 标志。`compile_commands.json` 可以用作构建系统无关的方式来加载命令。一些 IDE，如 VS Code 和 CLion，可以解释 JSON 文件并生成项目信息。也常用于调试编译器选项，以防某些事情没有按预期工作。编译命令数据库的完整规范可以在<https://clang.llvm.org/docs/JSONCompilationDatabase.html>找到。

### 3.5.2 库别名

库别名是在不创建新的构建目标的情况下引用库的一种方法，有时称为命名空间。常见的模式是从项目中安装的每个库以 `MyProject::library` 的形式创建一个库别名，可以用于对多个目标进行语义分组。有助于避免命名方面的冲突，特别是当项目包含公共目标时，比如名为 `utils` 的库、`helper` 和类似的库。在相同的命名空间下收集相同项目的所有目标是一个很好的实践。在链接来自其他项目的库时，包含名称空间可以避免意外包含错误的库。本章中所有的库目标都将使用一个命名空间作为别名来分组它们，以便它们可以使用命名空间引用：

```
add_library(Chapter3::hello ALIAS hello)  
...  
target_link_libraries(SomeLibrary PRIVATE Chapter3::hello)
```

除了帮助确定目标的来源，CMake 使用命名空间来识别导入的目标，并提供更好的诊断消息。

## 使用命名空间

作为一种良好的实践，始终使用名称空间对目标进行别名，并使用”命名空间::前缀”进行引用。

命名空间是组织构建构件的一种很好的方式。但有时这还不够，我们希望看到更大的视野，了解“什么正在使用什么”，以及哪个工件依赖于哪个库。CMake 可以帮助创建提供这样深刻见解的依赖图，我们将在下一章中看到。

## 3.6. 总结

现在，已经可以使用 CMake 创建应用程序和库，并开始构建更复杂的项目了。已经学习了如何将不同的目标链接在一起，以及如何将编译器和链接器选项传递给目标。我们还讨论了仅供内部使用的对象库，并讨论了动态库的符号可见性。最后，学习了如何自动记录这些依赖关系，以便对大型项目有一个了解。

下一章中，将学习如何在不同的平台上打包和安装应用程序和库。

## 3.7. 练习题

回答以下问题来测试对本章的理解：

1. 创建可执行目标的 CMake 指令是什么？
2. 创建库目标的 CMake 指令是什么？
3. 如何指定库是静态，还是动态？
4. 对象库有什么特别之处？在哪里使用？
5. 如何为动态库指定默认的符号可见性？
6. 如何为目标指定编译器选项，以及如何查看编译命令？

# 第二部分：实战 CMake

这部分内容的目标，就是让读者使用 CMake 建立一个软件项目。第 4 章和第 5 章将介绍如何安装和打包项目，依赖管理，以及包管理器的使用。第 6 章、第 7 章和第 8 章将介绍如何将外部工具集成到 CMake 项目中，以创建文档、确保代码质量或完成构建软件所需的任务。第 9 章和第 10 章将为构建项目创建可重用的环境，并处理分布式存储库和大型项目。最后，第 11 章将展示如何使用 CMake 进行模糊测试。

本节包括以下几章：

- 第 4 章，打包、部署和安装
- 第 5 章，集成第三方库和依赖管理
- 第 6 章，自动生成文档
- 第 7 章，集成代码质量工具
- 第 8 章，执行自定义任务
- 第 9 章，创建可复制的构建环境
- 第 10 章，处理大项目和分布式存储库
- 第 11 章，自动化模糊测试

# 第 4 章 打包、部署和安装

构建只是软件项目故事的一半，另一半是关于交付软件给使用者。使用者是项目最大利益相关方，使用者可能有各种各样的体验和目的，他们可能是开发人员、包维护人员、高级用户或普通人，理解它们的用例、场景和需求很重要。由于软件大多是抽象的，假设项目是烘培的豆子——它可能很美味，在工厂闻起来很香，但不恰当的包装会缩短保质期，使其难以运输或消费，这将使产品不太可能被消费者所接受。即使产品一开始很棒，消费者也不会注意到，因为糟糕的包装使他们对产品的体验也很糟糕。因此，从一开始就做好这些事情很重要，使消费者高兴会给产品带来更多价值。

CMake 有良好的内部支撑和工具，使安装和包装更容易，CMake 可以利用现有的项目代码来做这些事情。因此，使项目可安装或打包不会产生任何维护成本。本章中，我们将学习如何利用 CMake 现有的能力来安装和打包部署。

本章中，将讨论以下主题：

- 使目标可安装
- 提供项目配置信息
- 创建安装包

## 4.1. 相关准备

开始这一章前，应该对 CMake 的目标有所了解（需要了解第 1 章和第 3 章的内容）。本章将以这些知识为基础。

请从本书的 GitHub 库中获取本章的示例<https://github.com/PacktPublishing/CMake-Best-Practices>。本章的示例在 chapter04 子文件夹中。

## 4.2. 使目标可安装

项目中支持部署的方法是可安装，这里的“可安装”并不是指安装预先制作好的软件包，而是用户必须获取项目的源码，并从头开始构建。可安装的项目需要额外的构建系统，用于在系统上安装运行时或开发构件。构建系统将在这里执行安装操作，因为这里使用 CMake 来生成构建系统文件，所以 CMake 必须生成相关的安装代码。本节中，将了解如何来写这部分 CMake 代码。

### 4.2.1 install() 指令

`install(…)` 指令是一个内置的 CMake 命令，允许生成用于安装目标、文件、目录等的构建系统说明。CMake 不会生成安装指令，除非明确地说明。因此，安装总是在开发者的控制中。来看下它的基本用法。

#### 安装 CMake 目标

要使 CMake 目标可安装，必须用至少一个参数指定 TARGETS 参数。指令的签名如下所示：

```
install(TARGETS <target>... [...])
```

TARGETS 参数表示 `install` 可以接受一组 CMake 目标，以便为其生成安装代码，只安装目标的输出构件。最常见的目标输出工件定义如下：

- ARCHIVE (静态库、DLL 导入库和链接器导入文件):
  - 除了在 macOS 中标记为 FRAMEWORK 的目标
- LIBRARY (动态库):
  - 除了在 macOS 中标记为 FRAMEWORK 的目标
  - 除了 dll (Windows)
- RUNTIME (可执行文件和 dll):
  - 除了在 macOS 中标记为 MACOSX\_BUNDLE 的目标

使目标可安装之后，CMake 将生成必要的安装代码，以安装输出的构件，这里先将可执行目标安装在一起。要查看 `install(...)` 指令的运行情况，可以看看第 4 章示例 1 的 CMakeLists.txt 文件，可以在 chapter04/ex01\_executable 文件夹中找到：

```
add_executable(ch4_ex01_executable)
target_sources(ch4_ex01_executable src/main.cpp)
target_compile_features(ch4_ex01_executable PRIVATE cxx_std_11)
install(TARGETS ch4_ex01_executable)
```

前面的代码中，定义了一个名为 `ch4_ex01_executable` 的可执行目标，并在随后的两行中填充了它的属性。最后一行 `install(...)` 是我们感兴趣的，是 `ch4_ex01_executable` 所需的安装代码。

要检查 `ch4_ex01_executable` 是否可以安装，可以构建项目并通过 CLI 进行安装：

```
cmake -S . -B ./build
cmake --build ./build
cmake --install ./build --prefix /tmp/install-test
```

#### Note

不必为 `cmake --install` 指定 `--prefix` 参数，也可以使用 `CMAKE_INSTALL_PREFIX` 来修改非默认的安装路径。

当使用 CMake 和多配置生成器 (如 Ninja multi-config 和 Visual Studio) 时，请为 `cmake --build` 和 `cmake --install` 指定 `--config` 参数：

```
# For multi-config generators:
cmake --build ./build --config Debug
cmake --install ./build --prefix /tmp/install-test
--config Debug
```

看一下 `cmake --install` 做了什么:

```
-- Install configuration: ""
-- Installing: /tmp/install-test/lib/libch2.framework.component1.a
-- Installing: /tmp/install-test/lib/libch2.framework.component2.so
-- Installing: /tmp/install-test/bin/ch2.driver_application
-- Set runtime path of "/tmp/install-test/bin/
    ch2.driver_application" to ""
-- Installing: /tmp/install-test/bin/ch4_ex01_executable
```

前面输出的最后一行中，可以看到 `ch4_ex01_executable` 目标的输出构件——安装了 `ch4_ex01_executable` 二进制文件。由于 `ch4_ex01_executable` 目标拥有的唯一输出工件，所以该目标确实已经可安装。

注意 `ch4_ex01_executable` 没有直接安装在 `/tmp/installtest` 目录中，`install` 将它放在 `bin` 子目录中，这是因为 CMake 知道什么样的工件应该放在哪里。传统的 UNIX 系统中，二进制文件存放在 `/usr/bin` 中，而库文件存放在 `/usr/lib` 中，CMake 知道 `add_executable()` 会生成一个可执行的二进制文件，所以将其放入 `bin` 子目录中。这些目录的默认值由 CMake 提供，具体取决于目标类型。提供默认安装路径信息的 CMake 模块称为 `GNUInstallDirs` 模块。`GNUInstallDirs` 模块定义了各种 `CMAKE_INSTALL_` 的路径。默认安装目录如下表所示:

目标类型	GNUInstallDirs 变量	默认位置
RUNTIME	<code> \${CMAKE_INSTALL_BINDIR}</code>	<code>bin</code>
LIBRARY	<code> \${CMAKE_INSTALL_LIBDIR}</code>	<code>lib</code>
ARCHIVE	<code> \${CMAKE_INSTALL_LIBDIR}</code>	<code>lib</code>
PRIVATE_HEADER	<code> \${CMAKE_INSTALL_INCLUDEDIR}</code>	<code>include</code>
PUBLIC_HEADER	<code> \${CMAKE_INSTALL_INCLUDEDIR}</code>	<code>include</code>

为了覆盖内置默认值，在 `install(...)` 指令中需要使用 `<TARGET_TYPE> DESTINATION` 参数。这里，我们尝试将默认的运行时安装目录改为 `qbin`，而不是 `bin`。只需要对 `install(...)` 指令做一个小修改:

```
# ...
install(TARGETS ch4_ex01_executable
        RUNTIME DESTINATION qbin
    )
```

进行修改后，可以重新运行配置、构建和安装命令。可以通过 `cmake --install` 检查输出来确认运行时目标已经更改。与第一次不同，可以观察到 `ch4_ex01_executable` 二进制文件放入了 `qbin`，而不是默认的 (`bin`) 目录中:

```
# ...
-- Installing: /tmp/install-test/qbin/ch4_ex01_executable
```

现在，来看另一个例子，这次将安装 STATIC 库。回顾一下第 4 章示例 2 的 CMakeLists.txt 文件，可以在 chapter04/ex02\_static 文件夹中找到。由于空格的原因，注释和 project(… )省略：

```
add_library(ch4_ex02_static STATIC)
target_sources(ch4_ex02_static PRIVATE src/lib.cpp)
target_include_directories(ch4_ex02_static PUBLIC include)
target_compile_features(ch4_ex02_static PRIVATE cxx_std_11)
include(GNUInstallDirs)
install(TARGETS ch4_ex02_static)
install (
  DIRECTORY include/
  DESTINATION "${CMAKE_INSTALL_INCLUDEDIR}"
)
```

这与我们前面的示例有一点不同。首先，会有多一个 DIRECTORY 参数，这是使静态库的头文件可安装。这样做的原因是 CMake 不会安装任何非输出工件，而 STATIC 库目标只生成一个二进制文件作为输出工件。头文件不是输出工件，应该单独安装。

#### Note

DIRECTORY 参数中末尾的斜杠会导致 CMake 复制文件夹的内容，而不是按名称复制文件夹。CMake 处理尾随斜杠的方式与 Linux rsync 命令相同。

### 4.2.2 安装文件和目录

正如前一节中看到的，安装的东西并不总是目标输出构件的一部分。它们可能是目标的运行时依赖项，例如图片、源文件、脚本和配置文件。CMake 提供了 `install(FILES…)` 和 `install(DIRECTORY…)` 指令来安装任何特定的文件或目录。先从安装文件开始吧！

#### 安装文件

`install(FILES…)` 指令接受一个或多个文件作为参数，还需要 TYPE 或 DESTINATION 参数，这两个参数用于确定指定文件的目标目录。TYPE 用于指示哪些文件将使用该文件类型的默认路径作为安装目录，可以通过设置相关的 GNUInstallDirs 变量来重写默认值。下表显示了有效的 TYPE 值，以及它们的目录映射：

类型	<b>GUNInstallDirs</b> 变量	默认值
BIN	<code> \${CMAKE_INSTALL_BINDIR}</code>	bin
LIB	<code> \${CMAKE_INSTALL_SBINDIR}</code>	sbin
LIB	<code> \${CMAKE_INSTALL_LIBDIR}</code>	lib
INCLUDE	<code> \${CMAKE_INSTALL_INCLUDEDIR}</code>	include
SYSCONF	<code> \${CMAKE_INSTALL_SYSCONFDIR}</code>	etc
SHAREDSTATE	<code> \${CMAKE_INSTALL_SHARESTATEDIR}</code>	com
LOCALSTATE	<code> \${CMAKE_INSTALL_LOCALSTATEDIR}</code>	var
RUNSTATE	<code> \${CMAKE_INSTALL_RUNSTATEDIR}</code>	<LOCALSTATE dir>/run
DATA	<code> \${CMAKE_INSTALL_DATADIR}</code>	<DATAROOT dir>
INFO	<code> \${CMAKE_INSTALL_INFODIR}</code>	<DATAROOT dir>/info
LOCALE	<code> \${CMAKE_INSTALL_LOCALEDIR}</code>	<DATAROOT dir>/locale
MAN	<code> \${CMAKE_INSTALL_MANDIR}</code>	<DATAROOT dir>/man
DOC	<code> \${CMAKE_INSTALL_DOCDIR}</code>	<DATAROOT dir>/doc

若不想使用 TYPE，可以使用 DESTINATION。可以通过 `install(...)` 为指定文件提供自定义目标。

另一种形式的 `install(FILES...)` 是 `install(PROGRAMS...)`，这与 `install(FILES...)` 相同，但需要为安装文件设置 OWNER\_EXECUTE, GROUP\_EXECUTE 和 WORLD\_EXECUTE 权限。这对最终用户执行的二进制文件或脚本文件是有意义的。

为了理解 `install(FILES | PROGRAMS...)`，来看一个示例，第 4 章的例 3(chapter04/ex03\_file)。其包含三个文件:chapter4\_greeter\_content、chapter4\_greeter.py 和 CMakeLists.txt。首先，看看 CMakeLists.txt:

```
install(FILES "${CMAKE_CURRENT_LIST_DIR}/chapter4_greeter_content"
        DESTINATION "${CMAKE_INSTALL_BINDIR}")
install(PROGRAMS "${CMAKE_CURRENT_LIST_DIR}/chapter4_greeter.py"
        DESTINATION "${CMAKE_INSTALL_BINDIR}" RENAME chapter4_greeter)
```

第一个 `install(...)` 中，CMake 在当前的 CMakeLists.txt(chapter04/ex03\_file) 中安装了 chapter4\_greeter\_content 文件，在系统默认 BIN 目录下。第二个 `install(...)` 中，CMake 在默认 BIN 目录中安装 chapter4\_greeter.py，名称为 chapter4\_greeter。

#### Note

RENAME 参数只对单文件的 `install(...)` 有效。

有了这些 `install(...)` 指令，CMake 应该在  `${CMAKE_INSTALL_PREFIX}/bin` 目录下安装 chapter4\_greeter.py 和 chapter4\_greeter\_content。通过 CLI 来构建和安装这个项目：

```
cmake -S . -B ./build  
cmake --build ./build  
cmake --install ./build --prefix /tmp/install-test
```

看看 `cmake --install` 做了什么:

```
/* ... */  
-- Installing: /tmp/install-test/bin/chapter4_greeter_content  
-- Installing: /tmp/install-test/bin/chapter4_greeter
```

上面的输出确认了 CMake 为 `chapter4_greeter_content` 和 `chapter4_greeting.py` 文件生成了所需的安装代码。最后，因为使用了 `PROGRAMS` 参数来安装 `chapter4_greeter`，检查一下是否可以执行:

```
15:01 $ /tmp/install-test/bin/chapter4_greeter  
['Hello from installed file!']
```

至此，已经了解了 `install(FILES | PROGRAMS ...)`。接下来，就是安装目录。

## 安装目录

`install(DIRECTORY ...)` 指令用于安装目录，目录的结构将按原样复制到目标。目录既可以作为一个整体安装，也可以有选择地安装。先从最基本的目录安装示例开始:

```
install(DIRECTORY dir1 dir2 dir3 TYPE LOCALSTATE)
```

前面的示例将把 `dir1`、`dir2` 和 `dir3` 目录安装到  `${CMAKE_INSTALL_PREFIX}/var` 目录中，以及所有子文件夹和文件。有时，不能直接安装文件夹的全部内容。幸运的是，CMake 允许安装命令根据通配符模式和正则表达式包含或排除目录内容。这次我们有选择地安装 `dir1`、`dir2` 和 `dir3`:

```
include(GNUInstallDirs)  
install(DIRECTORY dir1 DESTINATION  
       ${CMAKE_INSTALL_LOCALSTATEDIR} FILES_MATCHING PATTERN "*.*")  
install(DIRECTORY dir2 DESTINATION  
       ${CMAKE_INSTALL_LOCALSTATEDIR} FILES_MATCHING PATTERN "*.hpp"  
       EXCLUDE PATTERN "")  
install(DIRECTORY dir3 DESTINATION  
       ${CMAKE_INSTALL_LOCALSTATEDIR} PATTERN "bin" EXCLUDE)
```

前面的例子中，使用 `FILES_MATCHING` 参数来定义文件选择的条件。`FILES_MATCHING` 后面可以跟 `PATTERN` 或 `REGEX` 参数。`PATTERN` 可以定义通配符模式，而 `REGEX` 可以定义正则表

达式。通常，这些表达式用于包含文件。若希望排除匹配条件的文件，可以将 EXCLUDE 参数附加到模式中。由于 FILES\_MATCHING 参数，这些过滤器没有应用于子目录名。我们还在最后一个 `install(...)` 指令中使用了 PATTERN，并且没有添加 FILES\_MATCHING 前缀，这可以过滤子目录而不是文件。这将只安装 dir1 中扩展名为.x 的文件，dir2 中没有扩展名为.hpp 的文件，以及 dir3 中除 bin 文件夹外的所有内容。这个例子可以在第 4 章找到，在 chapter04/ex04\_directory 文件夹中。让我们编译并安装，看看是否符合我们的预期：

```
cmake -S . -B ./build  
cmake -build ./build  
cmake -install ./build -prefix /tmp/install-test
```

`cmake --install` 的输出应该是这样的：

```
-- Installing: /tmp/install-test/var/dir1  
-- Installing: /tmp/install-test/var/dir1/subdir  
-- Installing: /tmp/install-test/var/dir1/subdir/asset5.x  
-- Installing: /tmp/install-test/var/dir1/asset1.x  
-- Installing: /tmp/install-test/var/dir2  
-- Installing: /tmp/install-test/var/dir2/chapter4_hello.dat  
-- Installing: /tmp/install-test/var/dir3  
-- Installing: /tmp/install-test/var/dir3/asset4
```

#### Note

FILES\_MATCHING 不能在 PATTERN 或 REGEX 之后使用，反之亦然。

输出中，可以看到只有扩展名为.x 的文件是从 dir1 中选取的，因为在第一个 `install(...)` 中使用 FILES\_MATCHING PATTERN “\*.x”，导致 asset2 文件没有被安装。另外，dir2/chapter4\_hello.dat 文件已经安装，并且跳过 dir2/chapter4\_hello.hpp 文件，这是因为在第二个 `install(...)` 中指定了 FILES\_MATCHING PATTERN “\*.hpp” EXCLUDE PATTERN “\*”。最后，安装了 dir3/asset4 文件，并且完全跳过了 dir3/bin 目录，因为在最后一个 `install(...)` 中指定了 PATTERN “bin” EXCLUDE 参数。

通过 `install(DIRECTORY...)`，了解了 `install(...)` 的基本知识。接下来，继续看看 `install(...)` 的其他常见参数。

### 4.2.3 `install()` 的其他常见参数

`install()` 的第一个参数指示要安装什么，还有一些参数允许进行定制安装。让我们一起了解一下这些常见参数。

## DESTINATION

这个参数允许 `install(...)` 指定安装目录，目录可以是相对路径，也可以是绝对路径。相对路径是相对于 `CMAKE_INSTALL_PREFIX` 的，建议使用相对路径使安装可重定位。另外，使用相对路径进行打包也很重要，因为 `cpack` 要求安装路径是相对路径。最好使用以相关 `GNUInstallDirs` 变量开始的路径，以便包维护人员在需要时覆盖安装目标。`DESTINATION` 参数可以与 `TARGETS`, `FILES`, `IMPORTED_RUNTIME_ARTIFACTS`, `EXPORT` 和 `DIRECTORY` 安装类型一起使用。

## PERMISSIONS

此参数允许在支持的平台上更改已安装文件的权限。可用权限为:OWNER\_READ、`OWNER_WRITE`、`OWNER_EXECUTE`、`GROUP_READ`、`GROUP_WRITE`、`GROUP_EXECUTE`、`WORLD_READ`、`WORLD_WRITE`、`WORLD_EXECUTE`、`SETUID` 和 `SETGID`。`PERMISSIONS` 参数可以与 `TARGETS`, `FILES`, `IMPORTED_RUNTIME_ARTIFACTS`, `EXPORT` 和 `DIRECTORY` 安装类型一起使用。

## CONFIGURATIONS

允许指定构建配置时限制参数的应用。

## OPTIONAL

此参数使要安装的文件为可选文件，因此当文件不存在时，安装不会失败。可选参数可以与 `TARGETS`, `FILES`, `IMPORTED_RUNTIME_ARTIFACTS` 和 `DIRECTORY` 安装类型一起使用。

本节中，学习了如何使目标、文件和目录可安装。下一节中，将学习如何生成配置信息，以便可以将 CMake 项目直接导入到另一个 CMake 项目中。

## 4.3. 提供项目配置信息

上一节中，学习了如何使项目可安装，以便其他人可以通过在他们的系统上安装使用。但是有时候，交付工件是不够的。例如，若交付一个库，也必须很容易导入到一个项——特别是 CMake 项目。本节中，我们将学习如何使这个导入过程更容易（为其他 CMake 项目）。

若要导入的项目具有正确的配置文件，有很多导入库的方便方法。其中一个是使用 `find_package()`（将在第 5 章中讨论）。若有使用者在工作中使用 CMake，只需要 `find_package(your_project_name)` 就可以使用你的代码，他们肯定会很高兴。本节中，将学习如何生成所需的配置文件，使 `find_package()` 在项目中工作。

CMake 使用依赖的首选方式是通过包。包为基于 CMake 的构建系统传递依赖信息，包可以是 `Config-file` 包、`Find-module` 包或 `pkg-config` 包。所有的包类型都可以通过 `find_package()` 找到并使用。出于对篇幅和简单性的考虑，本节只介绍 `Config-file` 包。其余的方法大多是缺少配置文件的变通方法，所以最好将它们留在过去。让我们了解一下 `Config-file` 包。

### 4.3.1 进入 CMake 包世界——Config-file 包

Config-file 包基于包含包内容信息的配置文件。该信息指示包的内容位置，因此 CMake 读取该文件并使用该包。因此，只使用包配置文件就可以使用该包。

有两种类型的配置文件——包配置文件和可选的包版本文件，两个文件都必须有一个特定的命名。包配置文件可以命名为 `<ProjectName>Config.cmake` 或 `<projectname>-config.cmake`，这取决于个人喜好。这两个符号将由 `CMake` 在 `find_package(ProjectName)/find_package(projectname)` 调用中选择。包配置文件的内容如下所示：

```
set(Foo_INCLUDE_DIRS ${PREFIX}/include/foo-1.2)
set(Foo_LIBRARIES ${PREFIX}/lib/foo-1.2/libfoo.a)
```

这里， `${PREFIX}` 是项目的安装前缀，是一个变量，因为安装前缀可以根据系统的类型更改，也可以由用户修改。

与包配置文件类似，包版本文件可以命名为 `<ProjectName>ConfigVersion.cmake` 或 `<projectname>-config-version.cmake`。CMake 期望包配置和包版本文件出现在 `find_package(...)` 的搜索路径中，可以在 CMake 的帮助下创建这些文件。搜索包时，`find_package(...)` 会查找 `<CMAKE_PREFIX_PATH>/cmake` 目录。我们的示例中，将把 Config-file 包配置文件放到这个文件夹中。

要创建 config-file 包，需要学习一些东西，比如导出目标和 `CmakePackageConfigHelpers` 模块。为了了解它，这里需要一个实际的例子。我们将使用第 4 章示例 5，使其成为具有 config-file 包，这个例子位于 `chapter04/ex05_config_file_package` 文件夹。来看看 `chapter04/ex05_config_file_package` 目录下的 `CMakeLists.txt` 文件（注释和项目命令在空格处被省略；另外，与主题无关的行这里不会再提及）：

```
include(GNUInstallDirs)
set(ch4_ex05_lib_INSTALL_CMAKEDIR cmake CACHE PATH
    "Installation directory for config-file package cmake files")
/* ... */
```

`CMakeLists.txt` 文件非常类似于 `chapter4/ex02_static`，并支持 config-file 打包。第一行 `include(GNUInstallDirs)` 用于包含 `GNUInstallDirs` 模块。这提供了 `CMAKE_INSTALL_INCLUDEDIR` 变量，将在稍后使用。`set(ch4_ex05_lib_INSTALL_CMAKEDIR...)` 是一个用户定义的变量，用于设置 config-file 打包配置文件的安装目录。这是一个相对路径，应该在 `install(...)` 指令中使用：

```
/* ... */
target_include_directories(ch4_ex05_lib PUBLIC
    ${BUILD_INTERFACE}:${CMAKE_CURRENT_SOURCE_DIR}/include)
target_compile_features(ch4_ex05_lib PUBLIC cxx_std_11)
/* ... */
```

`target_include_directories(…)` 与通常使用的方式不同。因为在将目标导入到另一个项目时，不存在构建时包含路径，其使用生成器表达式来区分构建时包含目录和安装时包含目录。下面的一组命令使目标可安装：

```
/* ... */
install(TARGETS ch4_ex05_lib
        EXPORT ch4_ex05_lib_export
        INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
)
install(
    DIRECTORY ${PROJECT_SOURCE_DIR}/include/
    DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
)
/* ... */
```

`install(TARGETS…)` 也和平常有一点不同，包含 `EXPORT` 参数，用于从给定的 `install(…)` 目标创建一个导出名称。然后，可以使用此导出名称导出这些目标。使用 `INCLUDES DESTINATION` 参数指定的路径，将用于填充导出目标的 `INTERFACE_INCLUDE_DIRECTORIES` 属性，并自动使用安装前缀路径作为前缀。这里，`install(DIRECTORY…)` 用于安装目标的头文件，位于 `${PROJECT_SOURCE_DIR}/include/`，在  `${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_INCLUDEDIR}` 目录中。 `${CMAKE_INSTALL_INCLUDEDIR}` 可以让使用者覆盖此安装的 `include` 目录。现在，可以根据前面例子中创建的导出名称创建一个导出文件：

```
/* ... */
install(EXPORT ch4_ex05_lib_export
        FILE ch4_ex05_lib-config.cmake
        NAMESPACE ch4_ex05_lib::
        DESTINATION ${ch4_ex05_lib_INSTALL_CMAKEDIR})
)
/* ... */
```

`install(EXPORT…)` 是这个文件中最重要的一段代码，是执行实际目标导出的代码。其生成一个 CMake 文件，其中包含给定导出名称中的所有导出目标。`EXPORT` 参数接受现有的导出名称来进行导出，它引用的是 `ch4_ex05_lib_export` 导出名称，我们在之前的 `install(TARGETS…)` 中创建的。`FILE` 用于确定导出的文件名，并设置为 `ch4_ex05_lib-config.cmake`。`NAMESPACE` 用于为所有导出的目标添加命名空间前缀。这允许将所有导出的目标连接到通用的命名空间下，并避免与具有相似目标名称的包发生冲突。最后，`DESTINATION` 确定生成导出文件的安装路径。设置为  `${ch4_ex05_lib_INSTALL_CMAKEDIR}` 以便 `find_package()` 发现它。

### Note

除了导出的目标之外，没有提供额外的内容，所以导出文件的名称是 ch4\_ex05\_lib-config.cmake。它是此包所需的包配置文件名称。这样做是因为示例项目不需要满足任何依赖关系，可以按原样直接导入。若需要额外的操作，建议使用中间包配置文件来满足这些依赖关系，然后包含导出的文件。

使用 `install(EXPORT...)`，获得了 `ch4_ex05_lib-config.cmake` 文件。从而目标可以通过 `find_package(...)` 来使用，要实现对 `find_package(...)` 的完全支持，还需要一个步骤，即获取 `ch4_ex05_lib-config-version.cmake` 文件：

```
/* ... */
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
    "ch4_ex05_lib-config-version.cmake"
    # Package compatibility strategy. SameMajorVersion is
    # essentially 'semantic versioning'.
    COMPATIBILITY SameMajorVersion
)
install(FILES
    "${CMAKE_CURRENT_BINARY_DIR}/ch4_ex05_lib-config-version.cmake"
    DESTINATION "${ch4_ex05_lib_INSTALL_CMAKEDIR}"
)
/* end of the file */
```

最后几行中，可以找到生成和安装 `ch4_ex05_lib-config-version.cmake` 文件所需的代码。使用 `include(CMakePackageConfigHelpers)`，导入 CMakePackageConfigHelpers 模块。这个模块提供了 `write_basic_package_version_file(...)` 函数，用于根据给定的参数自动生成包版本文件。第一个位置参数是输出的文件名。VERSION 用于以 major.minor.patch 形式指定生成包的版本，允许 `write_basic_package_version_file` 从项目版本自动获取。COMPATIBILITY 可以根据版本的值指定兼容性策略。SameMajorVersion 表示此包与具有此包相同主版本值的任何版本兼容，其他可能的值为 AnyNewerVersion、SameMinorVersion 和 ExactVersion。

现在，来测试一下这个方法是否有效。为了测试包配置，我们必须定期安装项目：

```
cmake -S . -B ./build
cmake --build ./build
cmake --install ./build --prefix /tmp/install-test
```

`cmake --install` 的输出如下所示：

```

/* ... */

-- Installing: /tmp/install-test/cmake/ch4_ex05_lib-config.cmake
-- Installing: /tmp/install-test/cmake/ch4_ex05_lib-ConfigNoConfig.
cmake
-- Installing: /tmp/install-test/cmake/ch4_ex05_lib-ConfigVersion.
cmake
/*...*/

```

可以看到包配置文件已经成功安装在/tmp/install-test/cmake 目录下。我将检查这些文件内容的工作留给读者作为练习。为了使用这个包，再来看下 chapter04/ex05\_consumer 示例中的 CMakeLists.txt 文件：

```

if(NOT PROJECT_IS_TOP_LEVEL)
  message(FATAL_ERROR "The chapter 4, ex05_consumer project is
intended to be a standalone, top-level project. Do not
include this directory.")
endif()
find_package(ch4_ex05_lib 1 CONFIG REQUIRED)
add_executable(ch4_ex05_consumer src/main.cpp)
target_compile_features(ch4_ex05_consumer PRIVATE cxx_std_11)
target_link_libraries(
  ch4_ex05_consumer ch4_ex05_lib::ch4_ex05_lib)

```

前几行中，可以看到关于项目是否是顶层项目的验证。这个示例是一个外部应用程序，所以不应该是根示例项目的一部分。因此，可以保证用包导出的目标，而不是根项目的目录，根项目也不包括 ex05\_consumer 文件夹。接下来，有一个 find\_package(...)，其中 ch4\_ex05\_lib 作为包名给出。还明确要求包的主版本为 1；find\_package(...) 只考虑 CONFIG 包和 find\_package(...) 中指定的包是必需的。在随后的行中，一个常规的可执行文件定义为 ch4\_ex05\_consumer，它在 ch4\_ex05\_lib 命名空间(ch4\_ex05\_lib::ch4\_ex05\_lib) 中链接到 ch4\_ex05\_lib。ch4\_ex05\_lib::ch4\_ex05\_lib 是我们在包中定义的实际目标。来看看源文件 src/main.cpp：

```

1 #include <chapter04/ex05/lib.hpp>
2 int main(void) {
3   chapter04::ex05::greeter g;
4   g.greet();
5 }

```

这是一个包含 chapter04/ex05/lib.hpp 的简单应用程序，创建了一个 greeter 类的实例，并调用了 greet() 函数。试着编译并运行这个应用程序：

```
cd chapter04/ex05_consumer  
cmake -S . -B build/ -DCMAKE_PREFIX_PATH:STRING=/tmp/install-test  
cmake --build build/  
./build/ch4_ex05_consumer
```

已经使用自定义前缀 (/tmp/install-test) 安装了包，所以可以通过设置 CMAKE\_PREFIX\_PATH 变量来表明这一点。这会导致 `find_package(...)` 也会在 /tmp/install-test 中搜索包。对于默认的前缀安装，不需要此参数设置。若一切顺利的话，我们应该看到输出为 Hello, world!：

```
./build/ch4_ex05_consumer  
Hello, world!
```

这样，使用者就可以使用我们的配置文件。接下来，通过学习如何使用 CPack 进行打包来结束本节。

## 4.4. 创建安装包

已经看到了 CMake 是如何构建软件项目的，是时候向大家介绍 CMake 的打包工具 CPack 了。它默认随 CMake 安装一起提供，可以利用现有的 CMake 代码来生成特定于平台的安装和包。CPack 在概念上类似于 CMake，基于生成包而不是构建系统文件的生成器。下表显示了从 3.21.3 版本起可用的 CPack 生成器类型：

生成器	描述
7Z	7-zip 压缩包
DEB	Debian 包
External	CPack 外部包
IFW	Qt 安装程序框架
NSIS	Null 软安装程序
NSIS64	Null 软安装程序 (64 位)
NuGet	NuGet 包
RPM	RPM 包
STGZ	自解压 TAR gzip 压缩包
TBZ2	Tar BZip2 压缩包
TGZ	Tar GZIP 压缩包
TXZ	Tar XZ 压缩包
TZ	Tar 压缩包
TZST	Tar Zstandard 压缩包
ZIP	Zip 压缩包

CPack 使用 CMake 的安装机制来填充包的内容，CPack 使用 CPackConfig.cmake 中的配置细节，CPackSourceConfig.cmake 文件生成包。这些文件可以手动填写，也可以由 CMake 在 CPack 模块的帮助下自动生成。在已有的 CMake 项目上使用 CPack，就像包含 CPack 模块一样简单，前提是该项目已有 `install(...)`。包含 CPack 模块会导致 CMake 生成 CPackConfig.cmake 和 CPackSourceConfig.cmake 文件，这是打包项目所需的 CPack 配置。另外，附加包目标将用于构建步骤。这一步将构建项目并运行 CPack，以便开始打包。当 CMake 或用户正确填充了 CPack 配置文件，就可以使用 CPack。CPack 模块允许定制包装过程，从而可以设置大量的 CPack 变量。这些变量分为两组——普通变量和生成器特定变量。公共变量影响所有包生成器，而生成器特定的变量只影响特定类型的生成器。下表显示了我们在示例中最常使用的 CPack 变量：

变量名	描述	默认值
CPACK_PACKAGE_NAME	包名	项目名
CPACK_PACKAGE_VENDOR	包供应商名	”Humanity”
CPACK_PACKAGE_VERSION_MAJOR	包主要版本	项目主要版本
CPACK_PACKAGE_VERSION_MINOR	包次要版本	项目次要版本
CPACK_PACKAGE_VERSION_PATCH	包补丁版本	项目补丁版本
CPACK_GENERATOR	使用的 CPack 生成器列表	N/A
CPACK_THREADS	支持并行的线程数	1

包含 CPack 模块之前，必须对变量进行更改，否则使用默认值。这里通过一个示例来了解 CPack 的实际应用，将使用第 4 章，例 6(`chapter04/ex06_pack`)的示例。此示例构造为一个独立的项目，并不是根示例项目的一部分。它是一个常规项目，有两个子目录，分别名为 `executable` 和 `library`。可执行目录的 CMakeLists.txt 文件如下所示：

```
add_executable(ch4_ex06_executable src/main.cpp)
target_compile_features(ch4_ex06_executable PRIVATE cxx_std_11)
target_link_libraries(
    ch4_ex06_executable PRIVATE ch4_ex06_library)
install(TARGETS ch4_ex06_executable)
```

库目录的 CMakeLists.txt 文件如下所示：

```
add_library(ch4_ex06_library STATIC src/lib.cpp)
target_compile_features(ch4_ex06_library PRIVATE cxx_std_11)
target_include_directories(ch4_ex06_library PUBLIC include)
set_target_properties(ch4_ex06_library PROPERTIES PUBLIC_HEADER
    include/chapter04/ex06/lib.hpp)
include(GNUInstallDirs)
# Defines the ${CMAKE_INSTALL_INCLUDEDIR} variable.
install(TARGETS ch4_ex06_library)
install (
    DIRECTORY ${PROJECT_SOURCE_DIR}/include/
```

```
DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

这些文件夹的 CMakeLists.txt 文件包含常规的、可安装的 CMake 目标，并没有声明关于 CPack 的内容。让我们看看顶层的 CMakeLists.txt 文件：

```
cmake_minimum_required(VERSION 3.21)
project(
    ch4_ex06_pack
    VERSION 1.0
    DESCRIPTION "Chapter 4 Example 06, Packaging with CPack"
    LANGUAGES CXX)
if(NOT PROJECT_IS_TOP_LEVEL)
    message(FATAL_ERROR "The chapter 4, ex06_pack project is
        intended to be a standalone, top-level project.
        Do not include this directory.")
endif()

add_subdirectory(executable)
add_subdirectory(library)

set(CPACK_PACKAGE_VENDOR "CBP Authors")
set(CPACK_GENERATOR "DEB;RPM;TBZ2")
set(CPACK_THREADS 0)
set(CPACK_DEBIAN_PACKAGE_MAINTAINER "CBP Authors")
include(CPack)
```

顶层 CMakeLists.txt 文件除了最后四行之外，几乎和一个普通的顶层 CMakeLists.txt 文件无异。它设置了三个与 CPack 相关的变量，然后包含了 CPack 模块，这四行足以提供基本的 CPack 支持。CPACK\_PACKAGE\_NAME 和 CPACK\_PACKAGE\_VERSION\_\* 变量没有让 CPACK 从顶层项目的名称和版本参数中进行推断。让我们配置这个项目，看看是否工作正常：

```
cd chapter04/ex06_pack
cmake -S . -B build/
```

项目配置完成后，CpackConfig.cmake 和 CpackConfigSource.cmake 文件将生成到 build/CPack\* 目录下。看看他们是否存在：

```
$ ls build/CPack*
build/CPackConfig.cmake build/CPackSourceConfig.cmake
```

这里，可以看到 CPack 配置文件是自动生成的。构建它，并尝试用 CPack 打包项目：

```
cmake --build build/
cpack --config build/CPackConfig.cmake -B build/
```

参数--config 是 CPack 命令的主要输入。参数-B 修改了默认的包目录，CPack 将把它的工件写入该目录。看看 CPack 的输出：

```
CPack: Create package using DEB
/*...*/
CPack: - package: /home/user/workspace/personal/CMake-Best-Practices/
chapter04/ex06_pack/build/ch4_ex06_pack-1.0-Linux.deb
generated.

CPack: Create package using RPM
/*...*/
CPack: - package: /home/user/workspace/personal/CMake-Best-Practices/
chapter04/ex06_pack/build/ch4_ex06_pack-1.0-Linux.rpm
generated.

CPack: Create package using TBZ2
/*...*/
CPack: - package: /home/user/workspace/personal/CMake-Best-Practices/
chapter04/ex06_pack/build/ch4_ex06_pack-1.0-Linux.tar.bz2
generated.
```

这里，可以看到 CPack 使用 DEB、RPM 和 TBZ2 生成器生成了 ch4\_ex06\_pack-1.0-Linux.deb, ch4\_ex06\_pack-1.0-Linux.rpm 和 ch4\_ex06\_pack-1.0-Linux.tar.bz2。尝试在 Debian 环境中安装生成的 Debian 包：

```
sudo dpkg -i build/ch4_ex06_pack-1.0-Linux.deb
```

若安装无误，应该能够在命令行中直接使用 ch4\_ex06\_executable：

```
13:38 $ ch4_ex06_executable  
Hello, world!
```

没问题! 作为练习, 请尝试安装 RPM 和 tar.bz2 包。

本节中, 学习了如何使用 CPack 来打包我们的项目, 这绝不是一个详尽的指南, CPack 本身值得用几章来详细介绍。

## 4.5. 总结

本章中, 学习了使目标可安装的基础知识, 以及如何为开发环境和使用者环境打包项目。部署是专业软件项目中非常重要的方面, 有了本章中工具的帮助, 就能够轻松地处理这样的部署需求。

下一章, 将学习如何将第三方库集成到 CMake 项目中。

## 4.6. 练习题

回答以下问题来测试对本章的理解:

1. 如何使 CMake 目标可安装?
2. 当使用 `install(TARGETS)` 安装目标时, 安装了哪些文件?
3. 对于库目标, 头文件是由 `install(TARGETS)` 安装的吗? 为什么? 若不是, 如何安装它们?
4. GNUInstallDirs 模块提供了什么?
5. 如何在目标目录中有选择地安装目录内容?
6. 为什么在指定安装目标目录时要使用相对路径?
7. `config-file` 包需要哪些文件?
8. “输出目标”是什么意思?
9. 如何使用 CPack 打包?

# 第 5 章 集成第三方库和依赖管理

介绍了如何使用 CMake 构建和安装。这一章，看看如何使用那些不是 CMake 项目的文件、库和程序。本章的第一部分是关于如何找到这些文件，而后将专注于如何管理依赖来构建 CMake 项目。

使用 CMake 的最大优势是内置了用于发现第三方库的依赖项管理。本章中，将了解如何集成安装在系统上的库和本地下载的依赖项。此外，将了解如何下载第三方库，并将其作为二进制文件使用，以及如何直接从 CMake 项目的源代码进行构建。

我们将了解如何为 CMake 编写指令，以便找到系统上的库。最后，将了解如何在 CMake 中使用包管理器，如 Conan 和 vcpkg。本章所述的依赖关系管理将有助于使用 CMake 创建稳定和可移植的构建，无论是使用预编译二进制文件，还是从头编译它们，设置 CMake 以一种结构化和一致性的方式处理依赖关系，将减少未来修复损坏的构建所花费的时间。以下是本章中涉及的主题：

- 查找文件、程序和路径
- 使用第三方库
- 包管理器
- 获取依赖项源代码

## 5.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本

另外，一些示例需要安装 OpenSSL 1.1 才能编译。一些示例从不同的在线位置提取依赖项，因此需要连接网络。所有的示例和源代码都可以从本书的 GitHub 库中获得，<https://github.com/PacktPublishing/CMake-Best-Practices>。

外部包管理器的示例需要在系统上安装 Conan(版本 1.40 或更新) 和 vcpkg 才能运行。可以在这里获得：

- Conan: <https://conan.io/>
- Vcpkg: <https://github.com/microsoft/vcpkg>

## 5.2. 查找文件、程序和路径

大多数项目的规模和复杂性都会迅速增长，依赖于文件、库，甚至是项目外部的程序。CMake 提供的内置指令可用来进行查找。搜索和寻找好像相当简单，但仔细分析一下，有很多事情需要考虑。首先，必须处理在哪里查找文件的搜索顺序。然后，可能想要添加文件可能所在的其他位置。最后，必须考虑不同操作系统之间的差异。

CMake 能够查找定义目标、包括路径和包特定变量的整个包。更多细节请参考 CMake 项目部分中的库。

有五个 `find_`…指令，它们选项和行为非常相似：

- `find_file`: 定位单个文件。
- `find_path`: 查找包含特定文件的目录。
- `find_library`: 查找库文件。
- `find_program`: 查找可执行程序。
- `find_package`: 查找完整的包。

所有这些命令都以类似的方式工作，但在查找位置方面有一些区别。特别是，`find_package`不仅定位文件、查找包，还使得文件在 CMake 项目中更易于使用。本章中，在介绍如何查找包之前，先了解一下 `find` 指令。

### 5.2.1 查找文件和路径

要查找的最底层最基本的是文件和路径。`find_file` 和 `find_path` 具有相同的签名，其唯一区别是 `find_path` 存储的是文件目录，而 `find_file` 将存储完整的路径，包括文件名。`find_file` 的签名如下：

```
find_file (
<VAR>
name | NAMES name1 [name2 ...]
[HINTS [path | ENV var]... ]
[PATHS [path | ENV var]... ]
[PATH_SUFFIXES suffix1 [suffix2 ...]]
[DOC "cache documentation string"]
[NO_CACHE]
[REQUIRED]
[NO_DEFAULT_PATH]
[NO_PACKAGE_ROOT_PATH]
[NO_CMAKE_PATH]
[NO_CMAKE_ENVIRONMENT_PATH]
[NO_SYSTEM_ENVIRONMENT_PATH]
[NO_CMAKE_SYSTEM_PATH]
[CMAKE_FIND_ROOT_PATH_BOTH |
ONLY_CMAKE_FIND_ROOT_PATH |
NO_CMAKE_FIND_ROOT_PATH]
)
```

上面的命令搜索单个文件(文件名)，或者搜索可能的文件名列表(使用 NAMES)。生成的路径存储在作为 `<VAR>` 变量中。若找不到文件，变量将为 `<VARIABLENAME>-NOTFOUND`。

若要搜索的文件名有变，比如不同的大写字母或命名约定，可能包含版本号，那么可以使用名称列表。传递名称列表时，当找到第一个文件，搜索就会停止，所以应按首选方式对名称进行排序。

## 搜索包含版本号的文件

建议在搜索包含某种版本编号形式的文件名前，先搜索没有版本编号的文件名。这是为了使本地构建的文件优于操作系统安装的文件。

HINTS 和 PATHS 选项包含搜索文件的默认位置之外的其他位置。PATH\_SUFFIXES 可以包含许多子目录，在这些位置的下面搜索相应的子目录。

`find_…`指令在已定义的位置和顺序中进行搜索，`NO_…_PATH`参数可以用来跳过相应的位置。下表显示了搜索位置的顺序和跳过位置的选项：

位置	跳过的选项
包的根变量	<code>NO_PACKAGE_ROOT_PATH</code>
特定于 CMake 的缓存变量	<code>NO_CMAKE_PATH</code>
特定于 CMake 的环境变量	<code>NO_CMAKE_ENVIRONMENT_PATH</code>
HINTS 选项中的路径	
系统环境变量	<code>NO_SYSTEM_ENVIRONMENT_PATH</code>
系统缓存变量	<code>NO_CMAKE_SYSTEM_PATH</code>
PATHS 选项中的路径	

仔细地了解一下搜索顺序，以及不同位置的含义：

- 包的根变量：只在 `find_file` 作为 `find_package` 的一部分时使用才有用。
- 特定于 CMake 的缓存变量：从 macOS 的 `CMAKE_PREFIX_PATH`, `CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH` 缓存变量派生的位置。通常设置 `CMAKE_PREFIX_PATH` 缓存变量优于其他两种类型，因为它可用于所有 `find_` 指令。前缀路径是位于常用文件结构（如 bin、lib、include 等）的搜索基点。`CMAKE_PREFIX_PATH` 是一个路径列表，对于其中每一个，`find_file` 都会搜索/include 或/include/\${`CMAKE_LIBRARY_ARCHITECTURE`} (若变量已经设置)。一般 CMake 会自动设置变量，开发人员不应该更改。特定于架构的路径优先于通用路径。
- `CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH` 缓存变量，仅当标准目录结构不适用时才应使用。它们不向路径添加额外的 include 后缀。
- 通过将 `NO_CMAKE_PATH` 选项传递给命令，或者全局地将 `CMAKE_FIND_USE_PATH` 变量设置为 `false`，可以跳过搜索这些路径。
- 系统环境变量：从 `CMAKE_PREFIX_PATH`、`CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH` 系统环境变量派生。这些变量的工作方式与缓存变量相同，但通常在 CMake 的外部设置。
- 在 Unix 平台上，列表用冒号 (:) 而不是分号 (;) 分隔，以符合特定于平台的环境变量。
- HINTS 选项中的路径：这些是手动指定的附加搜索位置。可以从属性值等其他值构造，也可以依赖于以前找到的文件或路径。
- 系统环境变量：INCLUDE 和 PATH 环境变量都可以包含要搜索的目录列表。同样，在 Unix 平台上，列表是用冒号 (:)，而不是分号 (;) 分隔的。

- Windows 上，以更复杂的方式处理 PATHS。对于其中每个，通过删除尾随的 bin 或 sbin 目录来提取基本路径。若设置了 CMAKE\_LIBRARY\_ARCHITECTURE，则添加 include/\${CMAKE\_LIBRARY\_ARCHITECTURE} 子目录作为每个路径的第一优先级。之后，搜索 include(不带后缀)。这样，才会搜索原始路径 (可能以 bin 或 sbin 结束，也可能不以 bin 或 sbin 结束)。传递 NO\_SYSTEM\_ENVIRONMENT\_PATH 变量或将 CMAKE\_FIND\_USE\_CMAKE\_SYSTEM\_PATH 变量设置为 false 将跳过环境变量中的位置。
- 假设 PATH 选项包含 C:\myfolder\bin;C:\yourfolder，和 CMAKE\_LIBRARY\_ARCHITECTURE 设置为 x86\_64，搜索顺序如下：
  1. C:\myfolder\include\x86\_64
  2. C:\myfolder\include\
  3. C:\myfolder\bin
  4. C:\yourfolder\include\x86\_64
  5. C:\yourfolder\include\
  6. C:\yourfolder\
- 系统缓存变量: CMAKE\_SYSTEM\_PREFIX\_PATH 和 CMAKE\_SYSTEM\_FRAMEWORK\_PATH 变量类似于 CMAKE 特定的缓存变量。这些变量不应该由开发人员更改，而是在 CMake 设置平台工具链时进行配置。例外是，若提供了自定义工具链文件，例如使用 sysroot 或交叉编译。
- 除了 NO\_CMAKE\_SYSTEM\_PATH 选项，CMAKE\_FIND\_USE\_CMAKE\_SYSTEM\_PATH 变量可以设置为 false，以跳过由系统特定的缓存变量提供的位置的搜索。
- PATHS 选项中的路径: 与提示 HINTS 相同，这些是手动提供的附加搜索位置。尽管在技术上没有禁止，但按照惯例，PATHS 变量应该是固定路径，并且不依赖于其他值。

若只搜索由提示或路径提供的位置，添加 NO\_DEFAULT\_PATH 选项将跳过其他所有位置。

有时，可能希望忽略用于搜索的特定路径，路径列表可以在 CMAKE\_IGNORE\_PATH 或 CMAKE\_SYSTEM\_IGNORE\_PATH 中指定。这两个变量在设计时都考虑到了交叉编译场景，很少在其他情况下使用。

## 交叉编译时搜索文件

交叉编译时，因为工具链会自包含其目录结构，所以搜索文件的过程比较特殊。首先，需要在工具链的目录中查找文件，通过设置 CMAKE\_FIND\_ROOT 变量，可以将所有搜索的原点更改为新位置。

此外，CMAKE\_SYSROOT、CMAKE\_SYSROOT\_COMPILE 和 CMAKE\_SYSROOT\_LINK 变量会影响搜索位置，但它们应该只在工具链文件中设置，而不是由项目本身设置。若常规搜索位置已经在 sysroot 或 CMAKE\_FIND\_ROOT 指定的位置下，则不会更改。以波浪号 (~) 开头并传递给 find\_ 指令的路径都不会更改，以避免跳过用户主目录下的目录。

通常，CMake 首先在上一段中提供的位置中搜索，然后继续搜索主机系统。这种行为可以通过设置 CMAKE\_FIND\_ROOT\_PATH\_MODE\_INCLUDE 变量为 BOTH、NEVER 或 ONLY 进行全局性修改。或者，可以为 find\_file 设置 CMAKE\_FIND\_ROOT\_PATH\_BOTH，ONLY\_CMAKE\_FIND\_ROOT\_PATH，或者 NO\_CMAKE\_FIND\_ROOT\_PATH。

下表显示了在不同的搜索模式下，设置选项或变量时的搜索顺序：

模式	选项	搜索顺序
BOTH	CMAKE_FIND_ROOT_PATH_BOTH	<ul style="list-style-type: none"> <li>• CMAKE_FIND_ROOT_PATH</li> <li>• CMAKE_SYSROOT_COMPILE</li> <li>• CMAKE_SYSROOT_LINK</li> <li>• CMAKE_SYSROOT</li> <li>• 常规搜索位置</li> </ul>
NEVER	NO_CMAKE_FIND_ROOT_PATH	<ul style="list-style-type: none"> <li>• 常规搜索位置</li> </ul>
ONLY	ONLY_CMAKE_FIND_ROOT_PATH	<ul style="list-style-type: none"> <li>• CMAKE_FIND_ROOT_PATH</li> <li>• CMAKE_SYSROOT_COMPILE</li> <li>• CMAKE_SYSROOT_LINK</li> <li>• CMAKE_SYSROOT</li> <li>• 常规路径或 CMAKE_STAGING_PREFIX 路径下</li> </ul>

CMAKE\_STAGING\_PREFIX 用于为交叉编译提供安装路径, 不应该通过在 CMAKE\_SYSROOT 进行安装从而进行改变。

### 5.2.2 查找应用

查找可执行文件非常类似于查找文件和路径, `find_program` 指令与 `find_file` 几乎有相同的签名。此外, `find_program` 有 `NAMES_PER_DIR` 选项, 表明每次搜索一个目录, 并在每个目录中搜索所有提供的文件名, 而不是在每个目录中搜索每个文件。

Windows 上, .exe 和.com 文件扩展名会自动添加到提供的文件名中, 而非.bat 或.cmd。

`find_program` 使用的缓存变量与 `find_file` 使用的缓存变量略有不同:

- `find_program` 自动将 bin 和 sbin 添加到由 `CMAKE_PREFIX_PATH` 提供的搜索位置。
- `CMAKE_LIBRARY_ARCHITECTURE` 中的值会忽略, 没有任何作用。
- `CMAKE_PROGRAM_PATH` 替换 `CMAKE_INCLUDE_PATH`。
- `CMAKE_APPBUNDLE_PATH` 替换 `CMAKE_FRAMEWORK_PATH`。
- `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM` 用于更改搜索程序的模式。

与其他 `find` 指令一样, 若 CMake 无法找到程序, `find_program` 将变量的值设置为 `<varname>-NOTFOUND`。这通常有助于确定是否应该启用, 依赖于某个外部程序的自定义构建步骤。

### 5.2.3 查找库

查找库是查找文件的特殊情况, 因此 `find_library` 支持与 `find_file` 相同的选项集。与 `find_program` 类似, 也有 `NAMES_PER_DIR` 选项, 该选项首先检查所有文件名, 然后再移动到下一个目录。查找常规文件和查找库之间的区别在于, `find_library` 自动将特定于平台的命名约定应用于文件名。在 Unix 平台上, 名称将以 lib 作为前缀, 而在 Windows 上, 将添加.dll 或.lib 扩展名。

同样, 缓存变量与 `find_file` 和 `find_program` 中使用的变量略有不同:

- `find_library` 通过 `CMAKE_PREFIX_PATH` 将 `lib` 添加到搜索位置，其使用 `CMAKE_LIBRARY_PATH` 代替 `CMAKE_INCLUDE_PATH` 来查找库。`CMAKE_FRAMEWORK_PATH` 的作用类似于 `find_file`。`CMAKE_LIBRARY_ARCHITECTURE` 的工作原理与 `find_file` 相同。
- 这是通过将各自的文件夹附加到搜索路径来完成的。`find_library` 与 `find_file` 搜索方式相同，也是遍历 `PATH` 环境变量中的位置，在每个前缀后面追加了 `lib`。另外，若设置了 `LIB` 环境变量，则使用 `LIB` 环境变量，而不是 `INCLUDE`。
- `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY` 用于更改库的搜索模式。

`CMake` 通常知道关于 32 位和 64 位搜索位置的约定，例如：为相同名称的不同库使用 `lib32` 和 `lib64` 文件夹的平台。该行为由 `FIND_LIBRARY_USE_LIB[32|64|X32]_PATHS` 变量控制，该变量控制应该首先搜索什么。此外，项目可以使用 `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX` 变量定义自己的后缀，该变量将覆盖其他变量的行为。这样做的需求非常少，修改 `CMakeLists.txt` 文件中的搜索顺序很快就会使项目难以维护，并严重影响在系统间的可移植性。

### 查找静态或动态库

大多数情况下，简单地将库名称传递给 `CMake` 就足够了。而有时，必须重写该行为。这样做的原因是，在某些平台上，是否应该优先使用静态版本的库，而非动态版本的库，反之亦然。做到这一点的最佳方法是将 `find_library` 拆分为两个调用，而不是试图在一个调用中实现这一点。若静态库与动态库位于不同的目录中，则鲁棒性更强：

```
find_library(MYSTUFF_LIBRARY libmystuff.a)
find_library(MYSTUFF_LIBRARY mystuff)
```

Windows 上不能使用这种方法，因为 `dll` 的静态库和导入库具有相同的名称 `.lib` 后缀，因此不能通过名称进行区分。`find_file`、`find_path`、`find_program` 和 `find_library` 在查找特定的文件时很方便。另一方面，依赖性发生在更高的级别上，这正是 `find_package` 的用武之地。使用 `find_package`，不需要首先搜索所有的 `include` 文件，然后是库文件，手动将它们添加到目标中，并最终解释所有特定于平台的行为。接下来让我们深入研究如何查找依赖项。

## 5.3. 使用第三方库

若正在认真地编写软件，那么项目迟早会依赖于项目外部的库。而不是寻找单独的库文件或头文件，`CMake` 社区和本书作者推荐的方法是使用 `find_package`，在 `CMake` 中查找依赖项的首选方法是使用包。

包提供了一组关于 `CMake` 和生成构建系统依赖关系的信息，可以以两种形式集成到项目中。可以通过上游项目提供的配置细节（也称为配置文件包）来实现，也可以通过所谓的查找模块包来实现，查找模块包通常定义在与包无关的地方，由 `CMake` 本身或使用包的项目来实现。这两种类型都可以通过使用 `find_package` 找到，结果是一组导入的目标和/或一组变量，其中包含与构建系统相关的信息。

`findPkgConfig` 模块使用 `find-pkg` 查找依赖项的相关元信息，还间接支持包的方式。

通常，`find` 模块用于定位依赖项，例如：上游没有为包配置提供必要的信息时。不要将它们与 CMake 实用程序模块混淆，后者与 `include()` 一起使用。只要可能，就应该使用上游提供的包而不是 `find` 模块，修复上游项目以提供必要的信息要优于编写 `find` 模块。

注意，`find_package` 有两个签名：一个基本签名或短签名，一个完整签名或长签名。通常，使用短签名就足以找到正在寻找的包，因为它更容易维护，应该是首选。短格式同时支持模块和配置包，而长格式只支持配置模式。

短模式的签名如下：

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
[REQUIRED] [[COMPONENTS] [components...]])
[OPTIONAL_COMPONENTS components...]
[NO_POLICY_SCOPE])
```

假设想要编写一个程序，通过 OpenSSL 库的适当功能将字符串转换为 sha256 哈希码。为了编译和链接这个示例，必须通知 CMake 该项目需要 OpenSSL 库，然后将其附加到目标中。现在，假设必要的库已经安装在系统上的默认位置；例如：Linux 使用 apt、RPM 或类似的包管理器，Windows 使用 chocolatey，macOS 使用 brew。

CMakeLists.txt 文件可能是这样的：

```
find_package(OpenSSL REQUIRED COMPONENTS SSL)
add_executable(find_package_example)
target_link_libraries(find_package_example PRIVATE OpenSSL::SSL)
```

上面的例子做了以下事情：

1. 第一行 `find_package(OpenSSL REQUIRED COMPONENTS SSL)`，正在为 OpenSSL 寻找一组库和头文件。具体来说，正在寻找 SSL 组件，而忽略加密组件。REQUIRED 关键字说明必须使用其来构建这个项目，若没有找到库，CMake 将停止配置过程，并出现错误。
2. 找到包后，使用 `target_link_library` 将库链接到目标，并链接 OpenSSL 包提供的 `OpenSSL::SSL` 目标。

若依赖指定了版本，则可以将其指定为 `major[.minor[.patch[.tweak]]]`，或使用 `versionMin..[<]versionMax` 的方式作为版本划定范围。对于版本范围，`versionMin` 和 `versionMax` 应该具有相同的格式，通过指定小于号，将排除 `versionMax`。

但从 3.21 版本起，CMake 无法查询可用组件的模块。因此，必须依赖模块或库提供者的文档，找出哪些组件是可用的。可用的模块可以通过以下命令查询：

```
cmake --help-module-list #< lists all available modules
cmake --help-module <mod> #< prints the documentation for
module <mod>
cmake --help-modules #< lists all modules and their
documentation
```

可以在以下网站找到 CMake 的模块列表<https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html>。

### 查找单个库和文件

可以查找单独的库和文件，但首选的方法是包。查找单独的文件，将它们提供给 CMake 将在编写查找模块的章节中介绍。

在模块模式下运行时，`find_package` 会搜索名为 `Find<PackageName>.cmake` 的文件；首先会在 `CMAKE_MODULE_PATH` 指定的路径中搜索，然后是在外部提供的路径中查找模块。

在配置模式下运行时，`find_package` 会搜索以下的文件：

- `<lowercasePackageName>-config.cmake`
- `<PackageName>Config.cmake`
- `<lowercasePackageName>-config-version.cmake` (若指定了版本详细信息)
- `<PackageName>ConfigVersion.cmake` (若指定了版本详细信息)

所有的搜索将按照明确的顺序进行，可以通过将相应的选项传递给 CMake 来跳过一些位置。`find_package` 比其他 `find_` 指令包含更多的选项。下表展示了搜索顺序：

位置	跳过命令中的选项
包的根变量	<code>NO_PACKAGE_ROOT_PATH</code>
特定于 CMake 的缓存变量	<code>NO_CMAKE_PATH</code>
特定于 CMake 的环境变量	<code>NO_CMAKE_ENVIRONMENT_PATH</code>
HINTS 选项中指定的路径	
系统环境变量	<code>NO_SYSTEM_ENVIRONMENT_PATH</code>
用户包的注册表	<code>NO_CMAKE_PACKAGE_REGISTRY</code>
特定于系统的缓存变量	<code>NO_CMAKE_SYSTEM_PATH</code>
系统包的注册表	<code>NO_CMAKE_SYSTEM_PACKAGE_REGISTRY</code>
PATHS 选项中指定的路径	

- 包的根变量：每个 `find_package` 包的根变量存储在名为 `<PackageName>_ROOT` 中，是搜索包的首选。包的根变量与 `CMAKE_PREFIX_PATH` 相同，不仅用于 `find_package`，也可用于其他 `find_` 指令。
- 特定于 CMake 的缓存变量：从 `CMAKE_PREFIX_PATH` 派生的位置。对于 macOS，`CMAKE_FRAMEWORK_PATH` 变量也是一个搜索位置。
- 通过设置 `CMAKE_FIND_USE_CMAKE_PATH` 变量为 `false`，将跳过 CMake 特定缓存变量的位置。
- 特定于 CMake 的环境变量：除了指定 `CMAKE_PREFIX_PATH` 和 `CMAKE_FRAMEWORK_PATH` 作为缓存变量外，若设置为环境变量，CMake 也会搜索它们。
- 将 `CMAKE_FIND_USE_ENVIRONMENT_PATH` 设置为 `false` 将禁用此行为。
- `find_package` 的 HINTS：这些是传递给 `find_package` 的可选路径。

- 特定于系统的环境变量: PATH 环境变量用于查找包和文件，并删除尾随的 bin 和 sbin 目录。每个系统的默认位置，例如 /usr、/lib 和类似的位置。
- 用户包的注册表: 包要么在标准位置中找到，要么在通过 CMAKE\_PREFIX\_PATH 选项传递给 CMake 的位置中找到。包注册表是告诉 CMake 在哪里查找依赖项的另一种方法。用户注册表对当前用户帐户有效，而系统包注册表在系统范围内有效。Windows 上，用户包的注册表位置存储在下面位置:
  - HKEY\_CURRENT\_USER\Software\Kitware\CMake\Packages\<packageName>
  - 在 Unix 平台上，存储在用户的主目录中: ./cmake/packages/<PackageName>
- 特定于平台的缓存变量: 对于 find\_package，特定于平台的缓存变量 CMAKE\_SYSTEM\_PREFIX\_PATH, CMAKE\_SYSTEM\_FRAMEWORK\_PATH 和 CMAKE\_SYSTEM\_APPBUNDLE\_PATH 的工作方式与其他 find 类似。它们由 CMake 设置，不应该由项目更改。
- 系统包的注册表: 与用户包注册表类似，这是 CMake 查找包的位置。在 Windows 上，存储在 HKEY\_LOCAL\_MACHINE\Software\itware\CMake\Packages\<packageName>\。
- Unix 系统不提供系统包的注册表。
- 来自 find\_package 的路径: 是传递给 find\_package 的可选路径。通常，HINTS 选项是从其他值计算出来的，或者依赖于变量，而 PATHS 选项是固定路径。

具体来说，当在配置模式下查找包时，CMake 将在各种前缀下查找以下文件结构:

```
<prefix>/
<prefix>/ (cmake|CMake) /
<prefix>/<packageName>*/ 
<prefix>/<packageName>*/(cmake|CMake) /
<prefix>/ (lib/<arch>|lib*|share) /cmake/<packageName>*/ 
<prefix>/ (lib/<arch>|lib*|share) /<packageName>*/ 
<prefix>/ (lib/<arch>|lib*|share) /<packageName>*/(cmake|CMake) / 
<prefix>/<packageName>*/(lib/<arch>|lib*|share) /cmake/ 
    <packageName>*/ 
<prefix>/<packageName>*/(lib/<arch>|lib*|share) /<packageName>*/ 
<prefix>/<packageName>*/(lib/<arch>|lib*|share) /<packageName>*/ 
    (cmake|CMake) /
```

macOS 平台上，还会搜索以下文件夹:

```
<prefix>/<packageName>.framework/Resources/
<prefix>/<packageName>.framework/Resources/CMake/
<prefix>/<packageName>.framework/Versions/*/Resources/
<prefix>/<packageName>.framework/Versions/*/Resources/CMake/
<prefix>/<packageName>.app/Contents/Resources/
<prefix>/<packageName>.app/Contents/Resources/CMake/
```

可以在官方的 CMake 文档中找到更多关于包的信息<https://cmake.org/cmake/help/latest/manual/cmake-packages.7.html>。

就模块而言，我们只讨论了如何查找现有的模块。但若想要寻找既没有集成到 CMake 中，也没有在标准位置中，或者没有为 CMake 提供配置说明的依赖项，会发生什么呢？让我们在下一节中寻找答案吧。

### 5.3.1 编写查找模块

目前仍然有很多库没有使用 CMake 管理，或者不导出 CMake 包。若可以将它们安装在系统的默认位置，找到这些库通常也不是问题。当使用仅为某个项目所需的专有第三方库，或者使用从系统包管理器安装的库构建的不同版本的库时，使用对应版本的库就成了件麻烦事。

若正在并行地开发多个项目，可能希望在本地处理每个项目的依赖关系。无论哪种方式，都应在本地管理依赖项，而不是过多依赖于系统安装的东西，这是一种很好的实践方式。

若依赖项不存在模块和配置文件，解决方案就是编写 find 模块。我们的目标是提供足够的信息，以便可以使用 `find_package` 找到。

find 模块如何找到必要的头文件和二进制文件，以及为 CMake 创建导入目标的指令。如本章前面所述，当使用 `find_package` 时，CMake 在 `CMAKE_MODULE_PATH` 中搜索名为 `Find<PackageName>.cmake` 的文件。

假设正在构建一个项目，已经下载或构建了依赖项，并在使用之前将其放置在一个名为 `dep` 的文件夹中。所以，项目目录结构看起来可能是这样：

```
|── dep <-- The folder where we locally keep dependencies  
|── cmake  
|   └── FindLibImagePipeline.cmake <-- This is what we need to write  
|── CMakeLists.txt <-- Main CmakeLists.txt  
|── src  
|   └── *.cpp files
```

要做的第一件事是将 `cmake` 文件夹添加到 `CMAKE_MODULE_PATH` 中，这是一个列表。因此，可在 `CMakeLists.txt` 文件中添加以下行：

```
list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")
```

CMake 会在 `cmake` 文件夹中查找 find 模块。通常，find 模块按以下顺序执行：

1. 查找属于包的文件。
2. 为包设置包含 `include` 路径和库目录的变量。
3. 为导入的包设置目标。
4. 为目标设置属性。

一个简单的 `FindModules.cmake` 看起来像这样：

```
cmake_minimum_required(VERSION 3.21)
```

```

find_library(
    OBSCURE_LIBRARY
    NAMES obscure
    HINTS ${PROJECT_SOURCE_DIR}/dep/
    PATH_SUFFIXES lib bin build/Release build/Debug
)

find_path(
    OBSCURE_INCLUDE_DIR
    NAMES obscure/obscure.hpp
    HINTS ${PROJECT_SOURCE_DIR}/dep/include/
)

include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(
    Obscure
    DEFAULT_MSG
    OBSCURE_LIBRARY
    OBSCURE_INCLUDE_DIR
)
mark_as_advanced(OBSCURE_LIBRARY OBSCURE_INCLUDE_DIR)

if(NOT TARGET Obscure::Obscure)
    add_library(Obscure::Obscure UNKNOWN IMPORTED)
    set_target_properties(Obscure::Obscure PROPERTIES
        IMPORTED_LOCATION "${OBSCURE_LIBRARY}"
        INTERFACE_INCLUDE_DIRECTORIES "${OBSCURE_INCLUDE_DIR}"
        IMPORTED_LINK_INTERFACE_LANGUAGES "CXX"
    )
endif()

```

例子中，可以观察到：

- 首先，使用 `find_library` 指令搜索属于依赖项的实际库文件。若找到了，其路径会包括实际的文件名，并存储在 `OBSCURE_LIBRARY` 变量中，通常称为 `<PACKAGENAME>_LIBRARY`。`NAMES` 参数是库的可能名称列表，名称会自动扩展为通用前缀和扩展名。前面的示例中，我们寻找名为“`obscure`”的文件 `libobscure`，则会找到 `libobscure.so` 或 `obscure.dll`。关于搜索顺序、提示和路径的更多细节将在本节后面介绍。
- 接下来，`Find` 模块尝试定位包含路径。通过查找库的已知路径模式来实现，通常是公共头文件。结果会存储在 `OBSCURE_INCLUDE_DIR` 变量中，通常这个变量名为

<PACKAGENAME>\_INCLUDE\_DIR。

3. 由于处理 find 模块的所有需求可能非常繁琐，而且会有重复，所以 CMake 提供了 FindPackageHandleStandardArgs 模块，该模块提供了一个函数 find\_package\_handle\_standard\_args 来处理常见的情况，该函数处理 REQUIRED、QUIET 和 find\_package 的版本相关参数。find\_package\_handle\_standard\_args 有一个短签名版本和一个长签名版本。本例中使用短签名版本：

```
find_package_handle_standard_args(<PackageName>
    (DEFAULT_MSG|<custom-failure-message>)
    <required-var>...
)
```

4. 对于大多数情况，find\_package\_handle\_standard\_args 的简短形式就足够了。其中，find\_package\_handle\_standard\_args 函数将包名作为第一个参数和包所需的变量列表。DEFAULT\_MSG 参数表明在成功或失败的情况下打印默认消息，这取决于 find\_package 是用 REQUIRED，还是 QUIET。消息可以自定义，但建议尽可能使用默认消息。这样，所有 find\_package 的消息就一致了。前面的例子中，find\_package\_handle\_standard\_args 检查已传递的 OBSCURE\_LIBRARY 和 OBSCURE\_INCLUDE\_DIR 变量是否有效，从而会设置 <PACKAGENAME>\_FOUND 变量。
5. 若一切顺利，find 模块将定义目标，这对于检查目标是否存在是很有帮助（以避免在有多个调用为相同的依赖时，覆盖目标的情况）。使用 add\_library 创建目标，因为还不能确定它是静态库还是动态库，所以类型为 UNKNOWN，并设置了 IMPORTED 标志。
6. 最后，设置库的属性。推荐的最小设置是 IMPORTED\_LOCATION 属性，和包含文件所在的位置 INTERFACE\_INCLUDE\_DIR。

若一切正常，可以像这样使用库：

```
find_package(Obscure PRIVATE REQUIRED)
...
target_link_libraries(find_module_example Obscure::Obscure)
```

现在我们了解了如何将其他库添加到您的项目中（已经可以使用的话）。但如何在一开始就将这些库引入系统呢？让我们在下一节中找出答案。

## 5.4. 包管理器

将依赖项添加到项目中最简单的方法是使用 apt-get、brew 或 Chocolatey 进行安装。安装的缺点是，可能会使用许多不同版本的库污染系统，并且所寻找的版本可能根本不可用。若正在处理多个项目，同时对依赖关系有不同的需求，那么这一点尤其重要。通常，开发人员会在本地为每个项目下载依赖项，以便每个项目能够独立工作。处理依赖关系的一种方法是使用包管理器，例如 Conan 或 vcpkg。

在依赖性管理方面，使用专用的包管理器有许多优点。处理 C++ 依赖的两个比较流行的是 Conan 和 vcpkg，都可以处理复杂的构建系统。想要完全了解它们需要阅读相关的书籍，所以这里

只介绍使用它们所需的基本知识。本书中，我们将着重于使用 CMake 项目中已经提供的包，而不是自己创建的包。

### 5.4.1 获取依赖——Conan

过去的几年里，Conan 包管理器获得了很大的普及，主要是因为它与 CMake 集成得非常好。Conan 是一个分布式包管理器，构建在客户机/服务器架构上。所以，本地客户端可以获取或上传包到一个或多个远程服务器。

Conan 最强大的特性是可以为多个平台、配置和版本创建和管理二进制包。创建包时，使用 `conanfile.py` 文件描述它们，该文件列出了所有依赖项、源和构建指令。

使用 Conan 客户机构建包并将其上传到远程服务器。这还有一个好处，若找不到适合本地配置的二进制包，可以使用源码在本地构建。

用 CMake 使用 Conan 的方法是使用 CMake 本身的 Conan。若不想这样做，可以在外部使用 Conan，但建议在使用 Conan 之前使用 `find_program` 检查 Conan 程序是否存在。

为了直接从 CMake 中使用 Conan，Conan 提供了一个 CMake 包装程序供下载。下面的例子下载了 `conan-cmake` 包装器，然后从 ConanCenter 中提取 `fmt` 格式化库作为项目中的常规库：

```
if(NOT EXISTS "${CMAKE_CURRENT_BINARY_DIR}/conan.cmake")
    message(STATUS "Downloading conan.cmake from
https://github.com/conan-io/cmake-conan")
    file( DOWNLOAD
        "https://raw.githubusercontent.com/conan-io/
        cmakeconan/0.17.0/conan.cmake"
        "${CMAKE_CURRENT_BINARY_DIR}/conan.cmake"
        EXPECTED_HASH
        SHA256=3bef79da16c2e031dc429e1dac87a08b9226418b300ce00
        4cc125a82687baeef
        STATUS download_status
    )

    if(NOT download_status MATCHES "^0;")
        message(FATAL_ERROR
            "Downloading conan.cmake failed with ${download_status}")
    endif()
endif()

include(${CMAKE_CURRENT_BINARY_DIR}/conan.cmake)

conan_cmake_autodetect(CONAN_SETTINGS)

conan_cmake_configure(REQUIRES fmt/6.1.2
```

```

GENERATORS cmake_find_package_multi)
conan_cmake_install(PATH_OR_REFERENCE .
BUILD missing
SETTINGS ${CONAN_SETTINGS}
)

list(APPEND CMAKE_PREFIX_PATH ${CMAKE_CURRENT_BINARY_DIR})
find_package(fmt 6.1 REQUIRED)
add_executable(conan_example)
target_link_libraries(conan_example PRIVATE fmt::fmt)

```

CMake 代码做了以下事情:

1. 若还没有下载 cmake-conan, conan.cmake 文件将会下载到当前二进制目录下。
2. 接下来, 使 Conan 函数可用。
3. 当包含 Conan, 就可以从当前的 CMake 配置中检测设置, 如编译器、平台等, 并将它们存储在 CONAN\_SETTING 变量中。
4. conan\_cmake\_configure 函数定义了 fmt, 并为 conan 设置了生成器, 这样就可以使用 find\_package 来包含依赖项。这将生成一个 conanfile.txt 文件, 其中包含当前构建目录中 Conan 的必要指令。
5. 最后, conan\_cmake\_install 会安装相应的依赖项。
6. PATH\_OR\_REFERENCE 会指出依赖关系的定义所在的位置。该命令的运行位置与 conan\_cmake\_configure 所处位置相同, 因此将搜索相同的目录。若 BUILD 缺失, 则这些包不能作为二进制文件从远程服务器获得, 那就需要在本地进行构建。
7. SETTINGS 将检索到设置传递给 Conan。
8. 由于生成的查找模块位于当前二进制目录中, 所以必须添加到 CMAKE\_MODULE\_PATH 中。
9. 完成下载后, 可以使用 find\_package 来包含依赖项, 然后像往常一样添加到现有目标中。

还可以直接提供 conanfile.txt 文件:

```

[requires]
fmt/6.1.2
[generators]
cmake_find_package

```

不运行 conan\_cmake\_configure 和 conan\_cmake\_install, 而是通过 conan\_cmake\_run 调用 conan。调整前面的示例以使用 conanfile.txt 文件将类似于以下内容(下载 conan.cmake 文件保持不变):

```

include(${CMAKE_CURRENT_BINARY_DIR}/conan.cmake)
conan_cmake_autodetect(CONAN_SETTINGS)
conan_cmake_run(CONANFILE ${CMAKE_CURRENT_LIST_DIR}/conanfile.txt
BASIC_SETUP
BUILD missing

```

```
SETTINGS ${CONAN_SETTINGS})  
list(APPEND CMAKE_PREFIX_PATH ${CMAKE_CURRENT_BINARY_DIR})  
find_package(fmt 6.1 REQUIRED)  
add_executable(conan_conanfile_example)  
target_link_libraries(conan_conanfile_example PRIVATE fmt::fmt)
```

此设置与前面的示例类似，不同之处是 Conan 指向 conanfile.txt 文件。BASIC\_SETUP 会告诉 Conan 自动创建必要的 CMake 变量，conan\_cmake\_run 也可以用来运行所有 conan 命令。

当然，Conan 也在 CMake 外部使用。有些人认为这是一种更简洁的方法，因为 Conan 和 CMake 之间的关注点是分离的，但维护起来会很繁琐。关于依赖项的信息必须在两个地方进行跟踪，而且构建配置，例如：编译器、libc 版本、平台等，不仅必须为 CMake 配置，也必须为 Conan 配置。

完整的 Conan 文档可参阅<https://docs.conan.io/en/latest/>。

#### 5.4.2 使用 vcpkg 进行依赖管理

另一个流行的开源包管理器是微软的 vcpkg，工作方式类似于 Conan，使用客户机/服务器架构。最初构建它是为了与 Visual Studio 编译器环境一起工作，后来添加了 CMake。包既可以在所谓的经典模式下调用 vcpkg 手动安装，也可以在清单模式下直接从 CMake 安装。使用经典方式安装 vcpkg 包的命令如下：

```
vcpkg install [packages]
```

当以清单模式运行时，项目的依赖项在 vcpkg.json 中定义，文件在项目的根目录下。清单模式有一个很大的优势，可以更好地与 CMake 集成，因此请尽可能使用清单模式。vcpkg 清单可能是这样的：

```
{  
  "name" : "vcpkg-example",  
  "version-semver" : "0.0.1",  
  "dependencies" :  
  [  
    "someLibrary",  
    "anotherLibrary",  
  ]  
}
```

为了让 CMake 找到这些包，vcpkg 工具链文件必须传递给 CMake，对 CMake 的使用如下所示：

```
cmake -S <source_dir> -D <binary_dir> -DCMAKE_TOOLCHAIN_  
FILE=[vcpkg root]/scripts/buildsystems/vcpkg.cmake
```

若以清单模式运行，则 `vcppkg.json` 文件中指定的包将自动下载并安装在本地。若以经典模式运行，则必须在运行 CMake 之前手动安装这些包。当传递 `vcppkg` 工具链文件时，可以使用 `find_package` 和 `target_link_libraries` 使用已安装的包。

微软建议将 `vcppkg` 作为子模块安装在存储库中，与 CMake 根项目处于同一级别，可以安装在任何地方。设置工具链文件可能会在交叉编译时导致问题，因为 `CMAKE_TOOLCHAIN_FILE` 可能已经指向一个不同的文件，所以第二个工具链文件可以通过 `VCPKG_CHAINLOAD_TOOLCHAIN_FILE` 传递：

```
cmake -S <source_dir> -D <binary_dir> -DCMAKE_TOOLCHAIN_
FILE=[vcppkg root]/scripts/buildsystems/vcppkg.cmake -DVCPKG
_CHAINLOAD_TOOLCHAIN_FILE=/path/to/other/toolchain.cmake
```

Conan 和 `vcppkg` 只是 C++ 和 CMake 中流行的两个包管理器。当然，还有更多知识点这里没有提及，但需要单独的书来描述它们。特别是当项目变得更加复杂时，强烈建议使用包管理器。

选择哪个包管理器取决于开发项目的环境和个人偏好。Conan 比 `vcppkg` 有一点优势，因为它在更多的平台上得到支持，可以在 Python 运行的地方运行。就交叉编译的特性和能力而言，两者不相上下。总之，Conan 提供了更高级的配置选项和对包的控制，这是以更复杂的处理为代价的。处理本地依赖关系的另一种方法是通过使用容器、`sysroot` 等创建完全隔离的构建环境，这将在第 12 章中讨论。现在，假设我们正在使用标准系统安装运行 CMake。

处理特定于项目的依赖项时，推荐使用包管理器进行依赖项管理。然而，有时无法选择使用包管理器。这可能是因为公司政策或其他原因。这种情况下，CMake 还支持将下载依赖项作为源，并将它们作为外部目标集成到项目中。

## 5.5. 获取依赖项源代码

有几种方法可以将依赖项作为源获取到项目中。一种相对直接（但危险）的方法是，将代码手动下载或克隆到项目内的子文件夹，然后使用 `add_subdirectory` 添加这个文件夹。虽然这样做很有效，而且速度很快，但很快就会变得乏味且难以维护。所以，这应该尽可能实现自动化。

### Note

下载并将第三方软件的副本集成到产品中的做法，称为供应商方式。优点是常常使构建软件变得容易，但在打包库方面产生了问题。通过使用包管理器或在系统上的某个位置安装第三方软件，可以避免使用供应商方式。

### 5.5.1 下载依赖项作为源代码

获取外部内容的基础模块是 `ExternalProject` 模块和更复杂的 `FetchContent` 模块，后者使用了 `ExternalProject`。虽然 `ExternalProject` 提供了更多的灵活性，但 `FetchContent` 使用起来更方便，特别是下载的项目也可以使用 CMake 构建。它们都将项目作为源文件下载，并可用于构建当前项目。

## 使用 FetchContent

对于构建的外部项目，使用 `FetchContent` 模块是添加源依赖项的一种方法。对于二进制依赖，使用 `find_package` 和 `Find<PackageName>.cmake` 式的方式仍是首选。`ExternalProject` 和 `FetchContent` 之间的主要区别是，`FetchContent` 可以在配置时下载和配置外部项目，而 `ExternalProject` 需要在构建时中完成所有工作。这样做的缺点是，源及其配置在配置时不可用。

没使用 `FetchContent` 之前，需要使用 `Git` 子模块手动下载依赖项，然后使用 `add_subdirectory` 添加源。在某些情况下这是可行的，但维护起来则会很麻烦。

`FetchContent` 提供了一系列函数来拉取源依赖项，主要是 `FetchContent_Declare`，它定义了下载和构建 `FetchContent_MakeAvailable` 的参数，`FetchContent_MakeAvailable` 填充依赖项的目标，并使它们可用于构建。下面的例子中，`bertrand` 库是通过 `Git` 将代码从 `GitHub` 上拉下来的：

```
include(FetchContent)
FetchContent_Declare(
    bertrand
    GIT_REPOSITORY https://github.com/bernedom/bertrand.git
    GIT_TAG 0.0.17)

FetchContent_MakeAvailable(bertrand)

add_executable(fetch_content_example)
target_link_libraries(
    fetch_content_example
    PRIVATE bertrand::bertrand
)
```

3.14 版本之后，可以使用 `FetchContent_MakeAvailable`，建议通过使用 `FetchContent_Populate` 手动填充项目，因为它的简单性使代码库非常可维护。和 `ExternalProject` 一样，`FetchContent` 也可以从 `HTTP(S)`、`Git`、`SVN`、`Mercurial` 和 `CVS` 下载，相应的实践（例如为下载的内容指定 `MD5` 哈希值或使用 `Git` 哈希值）也适用。

`FetchContent_MakeAvailable` 是使基于 `CMake` 的外部项目可用的推荐方法，但若想对外部项目有更多的控制，也可以手动填充项目。下面的示例与前面的示例相同，但更详细：

```
FetchContent_Declare(
    bertrand
    GIT_REPOSITORY https://github.com/bernedom/bertrand.git
    GIT_TAG 0.0.17

if(NOT bertrand_POPULATED)
    FetchContent_Populate(bertrand)
```

```
add_subdirectory(${bertrand_SOURCE_DIR} ${bertrand_BINARY_DIR})
endif()
```

FetchContent\_Populate 需要指定的附加选项，可以更细粒度地控制构建。签名如下：

```
FetchContent_Populate( <name>
    [QUIET]
    [SUBBUILD_DIR <subBuildDir>]
    [SOURCE_DIR <srcDir>]
    [BINARY_DIR <binDir>]
    ...
)
```

来看看 FetchContent\_Populate 的选项：

- QUIET：若成功，可以指定此值来抑制屏幕输出。若失败，即使指定了允许调试的选项，也会显示输出。
- SUBBUILD\_DIR：这指定了外部项目的位置。默认值是 \${CMAKE\_CURRENT\_BINARY\_DIR}/<name>-subbuild。通常，这个选项应该保持不变。
- SOURCE\_DIR 和 BINARY\_DIR：这更改了外部项目的源目录和构建目录的位置。对于 SOURCE\_DIR，默认设置为 \${CMAKE\_CURRENT\_BINARY\_DIR}/<lcname>-src；对于 BINARY\_DIR，默认设置为 \${CMAKE\_CURRENT\_BINARY\_DIR}/<lcname>-build。
- 添加的附加参数将传递到底层的 ExternalProject\_Add。然而，FetchContent 禁止编辑不同步骤的命令，因此试图修改 CONFIGURE\_COMMAND、BUILD\_COMMAND、INSTALL\_COMMAND 和 TEST\_COMMAND 的操作，将导致 FetchContent\_Populate 的失败，并显示相应的错误信息。

#### Note

若发现自己处于需要将选项传递给底层 ExternalProject\_Add 的情况下，可以考虑直接使用 ExternalProject。

关于源和构建目录的信息，以及项目是否填充，可以通过读取 <name>\_SOURCE\_DIR、<name>\_BINARY\_DIR 和 <name>\_POPULATED 变量或使用 FetchContent\_GetProperties 来检索。注意 <name> 或是全大写，或是全小写。尽管大小写不同，这是为了让 CMake 识别不同的包。

FetchContent 的另一个优势是，可以处理外部项目的公共依赖，并避免多次下载和构建。第一次在 FetchContent 上定义依赖时会缓存，之后的定义都会忽略。这样做可让父项目可以改写子项目的依赖关系。

假设有一个名为 MyProject 的顶层项目，它获取两个外部项目，project\_A 和 project\_B，每个项目都依赖于名为 AwesomeLib 的外部项目，但依赖于不同的版本。通常，我们不想下载和使用两个版本的 AwesomeLib，只想使用一个版本以避免冲突。下图显示了目前依赖关系的样子：

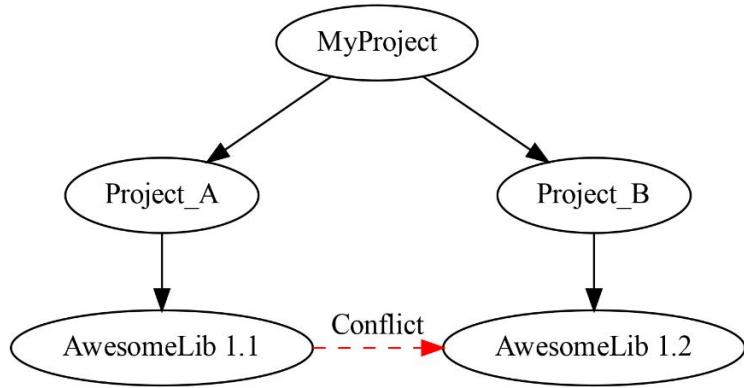


图 5.1 Project\_A 和 Project\_B 依赖于不同版本的 AwesomeLib

为了解决这个问题，可以通过在顶层的 CMakeLists.txt 文件中使用 `FetchContent_Declare` 对 AwesomeLib 进行调用，来指定要提取哪个版本的 AwesomeLib。这里，CMakeLists.txt 文件中声明的顺序并不重要，重要的是声明级别。因为 Project\_A 和 Project\_B 都使用了 AwesomeLib 的源码，所以顶层项目不需要使用 `FetchContent_MakeAvailable` 或 `FetchContent_Populate`：

```

include(FetchContent)
FetchContent_Declare(Project_A GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_Declare(Project_B GIT_REPOSITORY ... GIT_TAG ...)

# Force AwesomeLib dependency to a certain version
FetchContent_Declare(AwesomeLib
    GIT_REPOSITORY ...
    GIT_TAG 1.2)
FetchContent_MakeAvailable(Project_A)
FetchContent_MakeAvailable(Project_B)

```

这将使所有项目都使用 AwesomeLib 的 1.2 版本。当然，只有当 Project\_A 和 Project\_B 需要的版本之间的接口是兼容的情况下，才有以下依赖：

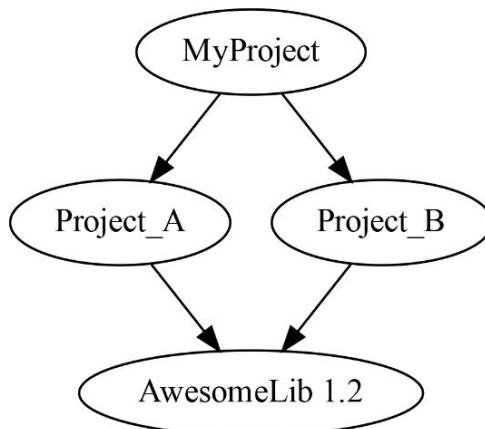


图 5.2 MyProject 声明了 AwesomeLib 版本后的依赖关系图

添加依赖项作为源有优点，也有缺点，这会增加了配置和构建时间。第 9 章中，我们将使用分布式库处理超级构建，并提供关于处理源依赖关系的更多信息。

本章的开头，讨论了 `find_package`，其可以用来包含二进制依赖项，但是没有讨论如何使用 CMake 方便地下载本地二进制依赖项。虽然 `FetchContent` 和 `ExternalProject` 可以用，但这不是其目的。相反，使用专门的包管理器会更合适，如 Conan 和 vcpkg。

## 使用 `ExternalProject`

`ExternalProject` 模块用于下载和构建未集成到主项目中的外部项目。构建外部项目时，构建完全隔离，不会自动接管与架构或平台相关的设置。这种隔离可以避免在命名目标或组件时发生冲突。外部项目创建主目标和几个包含以下独立构建步骤的子目标：

1. 下载: `ExternalProject` 可以通过几种方式下载内容，例如 HTTPS 下载，或者通过访问版本控制系统，如 Git、Subversion、Mercurial 和 CVS。若内容打包，下载后将进行解包操作。
2. 更新和打补丁: 若从 Server Configuration Monitor (SCM) 提取内容，则可以对下载的源代码进行打补丁或更新到最新版本。
3. 配置: 若下载的源代码使用 CMake，则在其上执行配置步骤。对于非 CMake 项目，可以提供执行配置的自定义命令。
4. 构建: 默认情况下，用于构建依赖项的构建工具与主项目中使用的构建工具相同，但若不这样做，可以提供自定义命令。若提供了自定义构建命令，则由用户来确保传递必要的编译器标志。
5. 安装: 隔离构建可以在本地安装，通常安装在主项目的构建树中的某个位置。
6. 测试: 若外部内容附带测试，则主项目可以选择性运行。通常，不运行测试。

包括下载在内的所有步骤都在构建时运行，根据外部项目的不同，这增加了构建的耗时。CMake 缓存下载和构建，开销主要是第一次运行时的耗时，除非外部项目进行了更改。可以向外部构建添加更多步骤，但对于大多数项目，默认的步骤已经足够了。这些步骤可以自定义，也可以省略。

以下示例中，`bertrand` 库通过 HTTPS 下载，并安装在当前构建目录中：

```
include(ExternalProject)
ExternalProject_Add(
    bertrand
    URL https://github.com/bernedom/bertrand/archive
    /refs/tags/0.0.17.tar.gz
    URL_HASH MD5=354141c50b8707f2574b69f30cef0238
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/bertrand_install
    CMAKE_CACHE_ARGS -DBERTRAND_BUILD_TESTING:BOOL=OFF
    -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
```

`ExternalProject` 模块在默认情况下不可用，必须通过 `include(ExternalProject)` 将其包含在第一行中。由于外部库安装在本地构建目录中，因此指定了 `INSTALL_DIR`。由于 `bertrand` 本身是一个 CMake 项目，可以通过 `CMAKE_INSTALL_PREFIX` 来构建项目，安装目录为 `<INSTALL_DIR>`(一个占位符)，指向 `INSTALL_DIR` 选项。`ExternalProject` 了解各种目录的占位符，例如 `<SOURCE_DIR>`、`<BINARy_DIR>` 和 `<DOWNLOAD_DIR>`。完整的列表，请参考模

块文档<https://cmake.org/cmake/help/latest/module/ExternalProject.html>。

### 下载验证

强烈建议将为下载 URL 添加哈希码，因为若工件的内容发生更改，这会将让您知晓。

为此，依赖 bertrand 的目标都必须在外部依赖之后构建。由于 bertrand 是一个纯头文件的库，所以想要将 include 路径添加到目标。在 CMake 中目标使用外部项目：

```
ExternalProject_Get_Property(bertrand INSTALL_DIR)
set(BERTRAND_DOWNLOADED_INSTALL_DIR "${INSTALL_DIR}")

# Create a target to build an executable
add_executable(external_project_example)

# make the executable to be built depend on the external project
# to force downloading first
add_dependencies(external_project_example bertrand)

# make the header file for bertrand available
target_include_directories(external_project_example PRIVATE
${BERTRAND_DOWNLOADED_INSTALL_DIR}/include)
```

第一行中，使用 ExternalProject\_Get\_Property 检索安装目录，并将其存储在 INSTALL\_DIR 中。但变量名总是与属性相同，因此建议在检索后立即将其存储在具有惟一名称的变量中。

接下来，创建目标并依赖于 ExternalProject\_Add 的目标。这对于执行构建的正确顺序很有必要。

最后，使用 target\_include\_directories 将安装路径添加到目标。此外，若外部项目不是由 CMake 构建的，可以导入由外部库提供的 CMake 目标。

通过相应的选项从源代码管理系统下载。对于 Git，通常像这样：

```
ExternalProject_Add(MyProject GIT_REPOSITORY
https://github.com/PacktPublishing/SomeRandomProject.git
GIT_TAG 56cc1aaaf50918f208e2ff2ef5e8ec0111097fb8d )
```

GIT\_TAG 可以是任何有效提交版本号，包括标记名和长、短哈希码。若省略 GIT\_TAG，则会下载默认分支（通常称为 main 或 master）的最新版本，建议下载指定版本。因为标签和分支可以修改（尽管在实践中很少这样做），所以最健壮的方法是定义一个提交哈希码。SVN 与从 Git 类似，更多细节请参考官方文档。

## 处理非 CMake 项目和交叉编译

`ExternalProject` 的一个常见用例是构建依赖关系，有时需要使用 CMake 之外的工具进行处理，比如：autotools 或 automake。这时，需要指定如下的配置和构建命令：

```
find_program(MAKE_EXECUTABLE NAMES nmake gmake make)
ExternalProject_Add(MyAutotoolsProject
    URL someUrl
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/myProject_install
    CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix=<INSTALL_DIR>
    BUILD_COMMAND ${MAKE_EXECUTABLE}
)
```

第一个 `find_program` 用于查询 `make` 的版本，并将其存储在 `MAKE_EXECUTABLE` 变量中。外部项目的一个常见问题是，必须控制依赖项的安装位置。大多数项目都希望安装到默认的系统位置，这通常需要 Root 特权，可能会污染系统，可以将必要的选项传递给配置或构建步骤。另一种处理方法是将 `INSTALL_COMMAND` 替换为一个空字符串来完全避免安装过程：

```
ExternalProject_Add(MyAutotoolsProject
    URL someUrl
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    BUILD_COMMAND ${MAKE_EXECUTABLE}
    INSTALL_COMMAND ""
)
```

使用此类的非 CMake 项目的问题是，它们没有为直接使用依赖定义必要的目标，为了在另一个目标中使用外部构建的库，通常必须将完整的库名称添加到 `target_link_libraries` 中，这样做的缺点是必须手动维护不同平台的文件的不同名称和位置。`find_library` 或 `find_file` 发生在配置时，所以对于 `ExternalProject` 没有用，`ExternalProject` 只在构建时才创建必要的文件。

另一个常见的用例是 `ExternalProject` 为不同的目标平台构建现有源目录的内容。本例中，直接省略了处理下载的参数。若外部项目使用 CMake 进行构建，则可以将工具链文件作为 CMake 选项传递给外部项目。这里有常见问题是，`ExternalProject` 不会识别外部项目源的更改，因此 CMake 可能不会重新构建它们。因此，应该传递 `BUILD_ALWAYS` 选项，其缺点是使构建时间变长：

```
ExternalProject_Add(ProjectForADifferentPlatform
    SOURCE_DIR
        ${CMAKE_CURRENT_LIST_DIR}/ProjectForADifferentPlatform
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/
        ProjectForADifferentPlatform-install
    CMAKE_ARGS
        -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
        -D CMAKE_BUILD_TYPE=Release
        -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
    BUILD_ALWAYS YES
```

```
)
```

## 管理 ExternalProject 中的步骤

如上一节所述，可以进一步配置 ExternalProject 的步骤，并以更细粒度的方式使用。通过传递 STEP\_TARGETS 选项或使用 ExternalProject\_Add\_StepTargets，告诉 ExternalProject 为每个步骤创建常规目标。以下调用将外部项目的配置步骤和构建步骤形成目标：

```
ExternalProject_Add(MyProject
    # various options
    STEP_TARGETS configure build
)
ExternalProject_Add_StepTargets(MyProject configure build)
```

目标以 <mainName>-step 命名。前面的示例中，将创建另外两个目标，MyProject-configure 和 MyProject-build。创建步骤目标有两个主要用途：可以创建按照下载、配置、构建、安装或测试顺序排序的自定义步骤，或者可以使步骤依赖于其他目标。这些目标可以是普通的目标，通过 add\_executable、add\_library 或 add\_custom\_target 创建，也可以添加入其他可执行文件的目标中。常见的情况是外部项目相互依赖，因此一个项目的配置步骤必须依赖另一个项目。下一个例子中，项目 B 的配置步骤将取决于项目 A 的完成情况：

```
ExternalProject_Add(ProjectA
    ... # various options
    STEP_TARGETS install
)
ExternalProject_Add(ProjectB
    ... # various options
)
ExternalProject_Add_StepDependencies(ProjectB configure ProjectA)
```

最后，还可以创建插入到外部项目中的自定义步骤。添加步骤可以通过 ExternalProject\_Add\_Step 指令完成。自定义步骤的名称不能与预定义步骤相同（例如创建文件夹、下载、更新、补丁、配置、构建、安装或测试）。下面的示例将创建一个步骤，构建后将外部项目的许可信息添加到特定的 tar 文件中：

```
ExternalProject_Add_Step(bertrand_downloaded copy_license
    COMMAND ${CMAKE_COMMAND} -E tar "cvzf"
        ${CMAKE_CURRENT_BINARY_DIR}/licenses.tar.gz <SOURCE_DIR>/LICENSE
    DEPENDS build
)
```

总之，ExternalProject 是一个非常强大的工具，但管理起来可能会很复杂，正是这种灵活性也使得 ExternalProject 难以使用。虽然可以隔离构建，但它需要项目维护者手动将来自外

部项目内部工作的信息公开给 CMake。更具有讽刺意味的是，这正是 CMake 最初所不愿看到的情况。

## 5.6. 总结

本章中，介绍了查找文件、库和程序的一般方法，以及查找 CMake 包更复杂的方法。学习了若无法通过提供自己的 `find` 模块自动找到导入的包定义，如何创建包。我们研究了基于源代码的依赖关系与 `ExternalProject` 和 `FetchContent`，以及如何使用 CMake 构建非 CMake 项目。

另外，觉得依赖管理方面变得更加太复杂，还简要的介绍了 Conan 和 vcpkg 这两个包管理器，它们与 CMake 集成得非常好。

依赖管理是一个很复杂的话题，有时可能很乏味，但花时间用本章描述的技术正确地设置它是值得的。CMake 的多功能性，及其查找依赖项的方法是其优点，也是其缺点。通过使用各种 `find_` 指令、`FetchContent`、`ExternalProject`，或者将可用的包管理器与 CMake 集成，可以将依赖项集成到项目中。然而，有这么多的方法可以选择，找到最好的会很困难，但我们还是建议尽可能使用 `find_package`。CMake 越受欢迎，其他项目就越有可能无缝集成。

下一章中，将学习如何为代码自动生成和打包文档。

## 5.7. 练习题

回答以下问题来测试对本章的理解：

1. `find_programs` 会搜索哪些目录？
2. 应该为 `find` 模块导入的目标设置哪些属性？
3. 在查找时，哪个选项会更优先，`HINTS` 还是 `PATHS`？
4. `ExternalProject` 在什么阶段下载外部内容？
5. `FetchContent` 在什么阶段下载外部内容？

# 第6章 自动生成文档

文档是项目的重要组成部分。文档传达的是用户无法直接获得的信息，是一种关于项目的意图、功能、能力和限制的方法，使技术人员和非技术人员都理解项目，并能在项目中工作。但是编写文档确实是一个耗时的过程，因此利用工具生成文档。

本章将探讨如何将 Doxygen、DOT 和 PlantUML 集成到 CMake 中，以加快文档编制过程。这些工具可以减少代码和文档之间的切换，并减轻文档的维护负担。

为了理解本章，我们将讨论以下主题：

- 用代码生成文档
- 用 CPack 打包和分发文档
- 创建 CMake 目标的依赖关系图

## 6.1. 相关准备

深入学习本章前，应该对第 4 章和第 5 章的内容有所了解，本章用到的知识在这两章中都有涉及。此外，建议从<https://github.com/PacktPublishing/CMake-Best-Practices/tree/main/chapter06>获取本章的示例内容。所有的示例都假设使用项目提供的开发环境容器，该容器可以在<https://github.com/PacktPublishing/CMake-Best-Practices>找到，是一个类似 debian 的环境。若使用不同的环境，则命令和输出可能略有不同。若不使用提供的 Docker 镜像，请确保环境中已经安装了 Doxygen、PlantUML 和 Graphviz。有关安装详细信息，请参阅包管理器的说明。

通过学习从现有代码生成文档的方法，来深入研究文档领域。

## 6.2. 用代码生成文档

大多数人都会以某种方式组织他们的软件项目，组织是方法和过程(如面向对象(OO)设计、编程语言规则、个人偏好、习惯或由项目规则规定的)。虽然规则和约定往往很无聊，但遵守它们会使项目结构更容易理解，有条理的材料对人类和计算机都好。当程序、规则、秩序和组织存在时，计算机可以对其进行理解，文档生成软件正是利用了这一点。

现在介绍 C 和 C++ 编程语言中最著名的文档生成软件之一:Doxygen。我们将学习如何将 Doxygen 与 CMake 集成，以自动生成 CMake 项目的文档。

接下来，先来了解一下 Doxygen。

### 6.2.1 了解 Doxygen

Doxygen 是一个非常流行的 C++ 项目文档软件，允许从代码生成文档。Doxygen 理解 C 和 C++ 语法，能够以编译器能够看到的方式看到代码结构。这允许 Doxygen 深入到软件项目的结构中，查看所有的类定义、命名空间、匿名函数、封装、变量、继承关系等。Doxygen 将这些信息与开发者编写的内联代码文档结合起来。最终的结果是生成可读的各种文档，其兼容在线和离线阅读。

但天下没有免费的午餐。为了能够理解代码注释，Doxygen 要求注释采用预定义的一组格式。为了不分散本章的重点，请查看<https://www.doxygen.nl/manual/docblocks.html>，找出

与 Doxygen 兼容的注释格式。我们将在示例中使用 Javadoc 风格的注释。这里提供了一个 C++ 函数的 Javadoc 注释示例：

```
1 /**
2 * Does foo with @p bar and @p baz
3 *
4 * @param [in] bar Level of awesomeness
5 * @param [in] baz Reason of awesomeness
6 */
7 void foo(int bar, const char* baz) {}
```

Doxygen 还需要一个 Doxyfile，其包含文档生成的所有参数，比如输出格式、排除的文件模式、项目名称等。因为配置参数太多，开始配置 Doxygen 可能会让人望而生畏，但 CMake 可以自动生成 Doxyfile。

随着我们深入了解，将了解为项目使用文档生成软件的好处。通过这种方式，可使文档与代码保持一致变得更容易，也会让绘制代码结构变得更容易。

理论到此为止，开始使用 Doxygen 和 CMake 吧。

## 6.2.2 Doxygen 和 CMake

CMake 是面向 C++ 的构建系统生成器，对集成 C++ 项目的外部工具有很好的支持，将 Doxygen 与 CMake 整合起来非常简单。使用 FindDoxygen.cmake 将 Doxygen 集成到我们的项目中。通常，该模块由 CMake 提供，不需要额外设置。

FindDoxygen.cmake 是由 `find_package()` 指定使用的模块包文件。主要用途是在环境中定位 Doxygen，并提供一些实用功能，以支持在 CMake 项目中生成文档。为了说明 Doxygen 的能力，本节将使用第 6 章的示例 01。目标是一个简单的计算器库，并将其 README 文件生成文档。库的接口定义：

```
1 class calculator : private calculator_interface {
2 public:
3 /**
4 * Calculate the sum of two numbers, @p augend lhs and @p addend
5 *
6 * @param [in] augend The number to which @p addend is added
7 * @param [in] addend The number which is added to @p augend
8 *
9 * @return double Sum of two numbers, @p lhs and @p rhs
10 */
11 virtual double sum(double augend, double addend)
12 override;
13 /**
14 * Calculate the difference of @p rhs from @p lhs
15 *
16 * @param [in] minuend The number to which @p subtrahend is subtracted
17 * @param [in] subtrahend The number which is to be subtracted from @p minuend
18 *
19 * @return double Difference of two numbers, @p minuend and @p subtrahend
20 */
```

```

21     virtual double sub(double minuend, double subtrahend)
22     override;
23     /*...*/
24 } // class calculator

```

calculator 类实现了在 calculator\_interface 类中定义的类接口，以 Javadoc 格式进行了正确的文档记录。我们期望 Doxygen 为 calculator 和 calculator\_interface 类生成应用程序编程接口 (API) 文档。类定义在 calculator.hpp 文件中，可以在 chapter06/ex01\_doxdocgen 目录的 include/chapter6/ex01 子目录下找到。另外，这里有一个 Markdown 文件 README.md。在 chapter06/ex01\_doxdocgen 目录中这包含了关于示例项目布局的基本信息，我们希望这个文件成为文档的主页。输入材料准备好后，通过检查示例的 CMakeLists.txt 文件也就是 chapter06/ex01\_doxdocgen/CMakeLists.txt 来进一步了解这个示例。CMakeLists.txt 文件从寻找 Doxygen 包开始：

```

find_package(Doxygen)
set(DOXYGEN_OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/docs")
set(DOXYGEN_GENERATE_HTML YES)
set(DOXYGEN_GENERATE_MAN YES)
set(DOXYGEN_MARKDOWN_SUPPORT YES)
set(DOXYGEN_AUTOLINK_SUPPORT YES)
set(DOXYGEN_HAVE_DOT YES)
set(DOXYGEN_COLLABORATION_GRAPH YES)
set(DOXYGEN_CLASS_GRAPH YES)
set(DOXYGEN_UML_LOOK YES)
set(DOXYGEN_DOT_UML_DETAILS YES)
set(DOXYGEN_DOT_WRAP_THRESHOLD 100)
set(DOXYGEN_CALL_GRAPH YES)
set(DOXYGEN QUIET YES)

```

`find_package(...)` 将使用 `FindDoxygen.cmake` 安装提供的 `cmake` 模块，以便在环境中查找 Doxygen(若存在的话)。省略 REQUIRED 参数是为了包维护人员打包项目，这可以确保在他们的环境中找到 Doxygen，然后再进行下一步。接下来的几行设置了几个 Doxygen 配置，这些配置将存在由 CMake 生成的 Doxyfile 中。下面列出每个选项的说明：

- DOXYGEN\_OUTPUT\_DIRECTORY: 设置 Doxygen 的输出目录。
- DOXYGEN\_GENERATE\_HTML: 生成超文本标记语言 (HTML)。
- DOXYGEN\_GENERATE\_MAN: 生成 MAN 页面。
- DOXYGEN\_AUTOLINK\_SUPPORT: Doxygen 自动链接语言符号和文件名到相关文档页面 (若可用)。
- DOXYGEN\_HAVE\_DOT: Doxygen 环境有可用的 dot，该命令可用于生成图像。这将使 Doxygen 能够使用依赖关系图、继承图和协作图来丰富生成的文档。
- DOXYGEN\_COLLABORATION\_GRAPH: 为类生成协作图。
- DOXYGEN\_CLASS\_GRAPH: 生成类图。
- DOXYGEN\_UML\_LOOK: Instructs 生成类似统一建模语言 (UML) 的图。

- DOXYGEN\_DOT\_UML\_DETAILS: 将类型和参数信息添加到 UML 图中。
- DOXYGEN\_DOT\_WRAP\_THRESHOLD: 为 UML 图设置行换行阈值。
- DOXYGEN\_CALL\_GRAPH: 为函数文档中的函数生成调用图。
- DOXYGEN QUIET: 静默生成到标准输出 (stdout) 的 Doxygen 输出。

Doxygen 的选项相当多，若想进一步定制文档生成，请查看可以在 Doxyfile 中使用的完整参数列表，<https://www.doxygen.nl/manual/config.html>。要在 CMake 中设置 Doxygen 选项，在变量名前加上 Doxygen\_ 并使用 `set()` 设置就好。写好旁注后，回到示例代码，前面显示的 CMake 代码后面跟着目标声明。以下代码行定义了一个静态库，其中包含了文档的示例代码：

```
add_library(ch6_ex01_doxdocgen_lib STATIC)
target_sources(ch6_ex01_doxdocgen_lib PRIVATE src/calculator.cpp)
target_include_directories(ch6_ex01_doxdocgen_lib PUBLIC include)
target_compile_features(ch6_ex01_doxdocgen_lib PRIVATE cxx_std_11)
```

随后，以下代码行定义了一个使用前面静态库目标的可执行文件：

```
add_executable(ch6_ex01_doxdocgen_exe src/main.cpp)
target_compile_features(ch6_ex01_doxdocgen_exe PRIVATE cxx_std_11)
target_link_libraries(
    ch6_ex01_doxdocgen_exe PRIVATE ch6_ex01_doxdocgen_lib)
```

最后，使用 `doxygen_add_docs(...)` 来生成文档：

```
doxygen_add_docs(
    ch6_ex01_doxdocgen_generate_docs
    "${CMAKE_CURRENT_LIST_DIR}"
    ALL
    COMMENT
    "Generating documentation for Chapter 6 - Example 01 with Doxygen"
)
```

`doxygen_add_docs(...)` 由 `FindDoxygen.cmake` 模块提供。其唯一目的是为文档生成创建 CMake 目标，`doxygen_add_docs(...)` 函数的签名（省略不相关的参数）：

```
doxygen_add_docs(targetName
    [filesOrDirs...]
    [ALL]
    [COMMENT comment])
```

第一个参数 `targetName` 是文档目标的名称，该函数将生成一个名为 `targetName` 的自定义目标。这个目标将触发 Doxygen，并在构建时使用代码创建文档。下一个参数列表 `filesOrDirs`，其包含想要从文档生成的代码的文件或目录的列表。`ALL` 参数用于使 CMake 的 `ALL` 元目标依赖于 `doxygen_add_docs(...)` 创建的文档目标，因此在构建 `ALL` 元目标时自动生成文档。最后，

COMMENT 参数用于让 CMake 在构建目标时输出一条消息。COMMENT 主要用于调试，以便快速知道是否生成了文档。

对 doxygen\_add\_docs(… ) 进行了简单的介绍后，回到示例代码，并解释 doxygen\_add\_docs(… ) 在场景中做了什么。其创建了一个名为 ch6\_ex01\_doxdocgen\_generate\_docs 的目标，将 \${CMAKE\_CURRENT\_LIST\_DIR} 添加到文档生成目录，请求 ALL 元目标依赖于它，并指定在构建目标时打印 COMMENT 参数。

好了，是时候测试一下这个是否有效了。进入 chapter06/ 目录，在 build/ 目录中使用以下命令配置项目：

```
cd chapter06/  
cmake -S . -B build/
```

检查 CMake 输出，查看配置是否成功。若配置成功，CMake 成功地在环境中找到了 Doxygen。应该可以在 CMake 的输出中看到：

```
Found Doxygen: /usr/bin/doxygen (found version "1.9.1")  
  found components: doxygen dot
```

成功配置之后，尝试使用以下命令进行构建：

```
cmake --build build/
```

构建输出中，应该能够看到提供给 COMMENT 参数的文在 CMake 输出中，所以文档目标正在构建，Doxygen 正在运行。这里没有为 CMake 构建命令指定--target 参数，这将直接构建 ALL 元目标。由于我们已经将 ALL 参数赋给了 doxygen\_add\_docs(… )，因此也构建了一个 ch6\_ex01\_doxdocgen\_generate\_docs 目标。build 命令的输出应该类似于这样：

```
[ 20%] Generating documentation for Chapter 6 - Example 01
with Doxygen
Doxygen version used: 1.9.1
Searching for include files...
Searching for example files...
/*...*/
Running dot...
Running dot for graph 1/1
lookup cache used 9/65536 hits=13 misses=9
finished...
[ 20%] Built target ch6_ex01_doxdocgen_generate_docs
[ 40%] Building CXX object ex01_doxdocgen/CmakeFiles
```

```
/ch6_ex01_doxdocgen_lib.dir/src/calculator.cpp.o
[ 60%] Linking CXX static library
libch6_ex01_doxdocgen_lib.a
[ 60%] Built target ch6_ex01_doxdocgen_lib
[ 80%] Building CXX object ex01_doxdocgen/CmakeFiles
/ch6_ex01_doxdocgen_exe.dir/src/main.cpp.o
[100%] Linking CXX executable ch6_ex01_doxdocgen_exe
[100%] Built target ch6_ex01_doxdocgen_exe
```

我们已经成功地构建了项目和文档，检查一下在 \${CMAKE\_CURRENT\_BINARY\_DIR}/docs 输出文件夹中生成的文档：

```
14:27 $ ls build/ex01_doxdocgen/docs/
html man
```

可以看到 Doxygen 将 HTML 和 MAN 页面输出发送到 html/ 和 man/ 目录中，检查每种类型的最终结果。要检查生成的 MAN 页面，只需输入以下命令：

```
man build/ex01_doxdocgen/docs/man/man3
/chapter6_ex01_calculator.3

NAME
    chapter6::ex01::calculator - The 'calculator' class
    interface.

SYNOPSIS
    #include <calculator.hpp>
    Static Public Member Functions
        static double sum (double augend, double addend)
        static double sub (double minuend, double
            subtrahend)
        static double mul (double multiplicand, double
            multiplier)
        static double div (double dividend, double divisor)

Detailed Description
    The 'calculator' class interface.

Member Function Documentation
```

```
double chapter6::ex01::calculator::div (double
    dividend, double divisor) [static]
    Divide dividend with divisor
Parameters
    dividend The number to be divided by divisor
    divisor The number by which divisor is to be
        divided
Returns
    double Quotient of two numbers, dividend and
        divisor
Manual page chapter6_ex01_calculator.3 line 1 (press h for
help or q to quit)
```

太棒了！查一下 HTML 输出，注释变成了 MAN 页面。使用浏览器打开 build/ex01\_doxdocgen/docs/html/index.html 文件：

```
google-chrome build/ex01_doxdocgen/docs/html/index.html
```

这将显示文档的主页面，如下图所示：

Main Page Classes ▾ Files ▾ Search

## Main Page

### Chapter 6 - Example 01

This example is intended to illustrate integration between CMake and the Doxygen.

#### Project structure

Project contains a static library and an executable target. Static library consist of two header files and one source file (`calculator.hpp`, `calculator_interface.hpp`, `calculator.cpp`), whereas executable only contains a single source file (`main.cpp`).

#### Static library (ch6\_ex01\_doxdocgen\_lib)

An example library that provides a class named `calculator`. This class contains four static functions named `sum(...)`, `sub(...)`, `div(...)` and `mul(...)`. In order to be able to illustrate documentation generation, these functions are properly documented in Doxygen JavaDoc format.

#### Example application(ch6\_ex01\_doxdocgen\_exe)

The application that consumes the `calculator` class and prints basic four arithmetic operation outputs to the stdout. Example application is not important for this example's purpose. It is included for completeness.

图 6.1 文档的主页

可以看到 Doxygen 渲染了 README.md 文件内容到主页面上，主页只是作为示例提供的。Doxygen 可以在生成的文档中嵌入任意数量的 Markdown 文件，甚至可以用到相关文档的链接替换了文件名、类名和函数名，可以通过 Doxygen 的 AUTOLINK 特性和 @ref 指令实现。单击主页面的静态库部分下面的 `calculator` 链接，以访问 `calculator` 类的 API 文档。`calculator` 类文档页面应该如下所示：

Public Member Functions | Public Attributes |  
List of all members

## chapter6::ex01::calculator Class Reference

The basic 'calculator' class. More...

```
#include <calculator.hpp>
```

Inheritance diagram for chapter6::ex01::calculator:

```
graph TD; chapter6::ex01::calculator_interface --> chapter6::ex01::calculator
```

chapter6::ex01::calculator\_interface

+ virtual double sum(double augend, double addend)=0  
+ virtual double sub(double minuend, double subtrahend)=0  
+ virtual double mul(double multiplicand, double multiplier)=0  
+ virtual double div(double dividend, double divisor)=0  
+ virtual ~calculator\_interface()=default

chapter6::ex01::calculator

+ double last\_result  
+ virtual double sum(double augend, double addend) override  
+ virtual double sub(double minuend, double subtrahend) override  
+ virtual double mul(double multiplicand, double multiplier) override  
+ virtual double div(double dividend, double divisor) override

### Public Member Functions

virtual double `sum` (double augend, double addend) override  
virtual double `sub` (double minuend, double subtrahend) override  
virtual double `mul` (double multiplicand, double multiplier) override  
virtual double `div` (double dividend, double divisor) override

### Public Attributes

double `last_result` {}

图 6.2 为 calculator 类生成的 HTML 文档 (基本布局)

可以看到 Doxygen 知道 calculator 类继承自 calculator\_interface，并为 calculator 类绘制了继承图。

#### Note

Doxygen 需要 dot 工具来渲染图表，dot 在 Graphviz 软件包中。

此外，生成的图包含 UML 样式的函数名和符号。下图显示了成员函数文档：

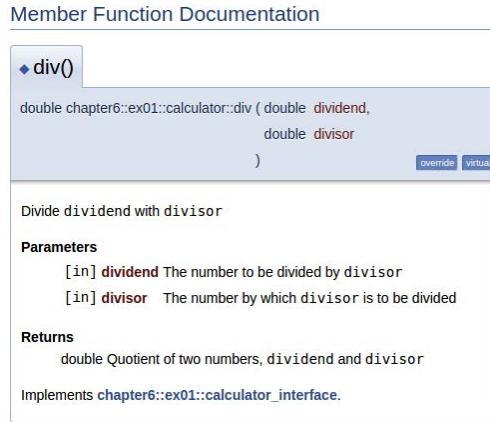


图 6.3 为 calculator 类的 div() 函数生成的文档

如图 6.3 所示，Doxygen 在将内容放入清晰易读的布局方面做得很好。最后，导航到 Files | File List | main.cpp，查看 main.cpp 的文档，以说明依赖关系图是什么样子的。可以在下面的截图中看到文档页面：

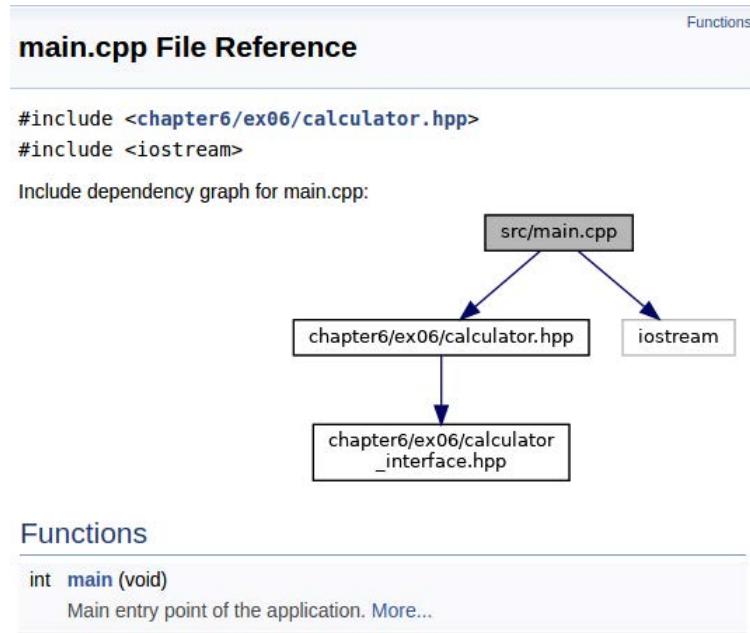


图 6.4 main.cpp 文档页面

从上图的依赖关系图可以看出, main.cpp 文件直接依赖于 iostream 和 chapter6/ex01/calculator.hpp 文件, 间接依赖于 chapter6/ex01/calculator\_interface.hpp 文件。文档中提供的依赖信息非常有用, 使用者可以确切地知道文件的依赖关系。若再向下滚动一点, 会看到 main() 函数的调用图:

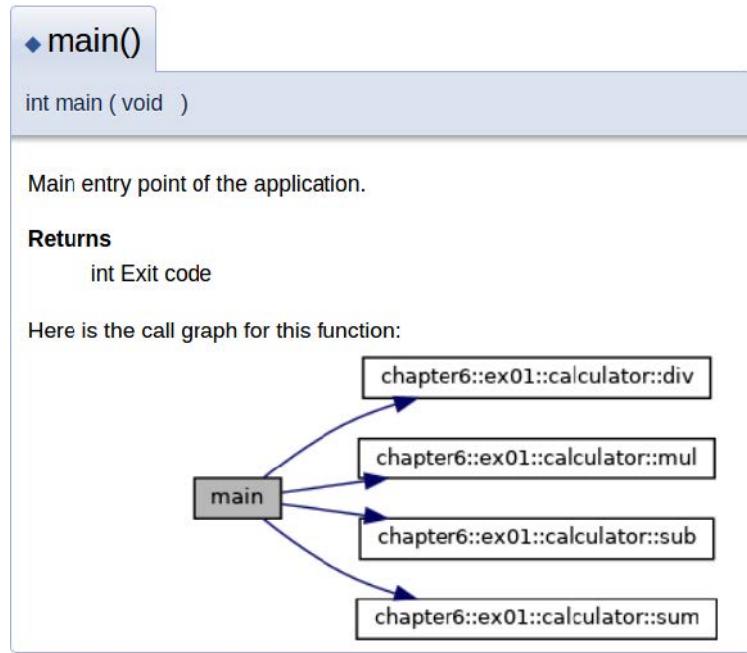


图 6.5 main() 函数的调用图

我们用不到 20 行的 CMake 代码, 为代码生成了两种不同格式的图表文档。多酷啊! 有了这种能力, 就很难找到借口来逃避文档化。接下来的部分会将定制的 UML 图嵌入到文档中来丰富我们的知识, 一起来看看吧!

### 6.2.3 将 UML 图嵌入到文档中

前一节中, 学习了如何利用 Doxygen 为 CMake 项目生成图表和文档, 但是并不是每个图表都可以从代码中推断出来。我们可能需要绘制自定义图像, 来说明实体与代码上下文中不可用的外部系统之间的关系。要解决这个问题, 需要在代码或注释中提供该上下文, 以便再次使用文档生成。这可以通过 Doxygen 实现, Doxygen 允许将 PlantUML 图表嵌入到注释中, 这使我们能够绘制 PlantUML 支持的图表。但在开始将 PlantUML 代码放入 Doxygen 注释前, 有一件事必须注意: 在 Doxygen 中需要启用 PlantUML。

在 Doxygen 中启用 PlantUML 支持非常简单。Doxygen 需要将 PLANTUML\_JAR\_PATH 变量设置为 PLANTUML.jar 文件的位置, 所以得找出文件的位置。可以使用 `find_path(...)`, `find_path(...)` 和 `find_program(...)` 类似, 指定用于定位文件的路径。使用本节第 6 章的例 02, 看一下示例代码的 CMakeLists 文件, 它位于 chapter06/ex02\_doxplantuml/CMakeLists.txt。从 `find_path(...)` 开始:

```
find_path(PLANTUML_JAR_PATH NAMES plantuml.jar HINTS
          "/usr/share/plantuml" REQUIRED)
find_package(Doxygen REQUIRED)
set(DOXYGEN_OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/docs")
```

```

set(DOXYGEN_GENERATE_HTML YES)
set(DOXYGEN_AUTOLINK_SUPPORT YES)
set(DOXYGEN_PLANTUML_JAR_PATH "${PLANTUML_JAR_PATH}")
set(DOXYGEN QUIET YES)

```

`find_path(...)` 的输出为 `PLANTUML_JAR_PATH`, `NAME` 是搜索位置中的文件名, `HINTS` 是默认搜索位置之外的路径, 这对于在非标准位置很有用。最后, `REQUIRED` 参数用于使 `plantuml.jar` 必须找到, 因此当无法找到 `plantuml.jar` 时, CMake 将失败并退出。下面的 Doxygen 配置部分与我们之前的示例 (第 6 章的示例 01) 完全相同, 不同的是将 `DOXYGEN_PLANTUML_JAR_PATH` 设置为找到的 PlantUML 目录路径。此外, 本例中不需要的变量也会省略了。Doxygen 现在应该可以使用 PlantUML 了, 我们用一个 PlantUML 示例图进行测试, 将其嵌入到 `src/main.cpp` 源文件中:

```

1 /**
2 * @brief Main entry point of the application
3 @startuml{system_interaction.png} "System Interaction
4     Diagram"
5 user -> executable : main()
6 user -> stdin
7 executable -> executable: read_stdin()
8 executable -> stdout
9 @enduml
10 * @return int Exit code
11 */
12 int main(void) {
13     std::cout << "Greetings from the echo application!" <<
14     std::endl;
15     std::string input;
16     while (std::getline(std::cin, input)) {
17         std::cout << input;
18     }
19 }

```

`@startuml` 和 `@enduml` Doxygen 注释关键字分别用于指示 PlantUML 图的开始和结束。普通的 PlantUML 代码可以放在 `@startuml - @enduml` 块中。示例中, 简单的应用程序系统交互图。若一切按计划进行, 应该在 `main()` 函数的文档中看到嵌入的 PlantUML 图。可以用下面所示的代码构建示例来生成文档:

```

cd chapter06/
cmake -S ./ -B build/
cmake --build build/

```

第二个示例的文档构建好了。使用浏览器打开生成的 `build/ex02_doxplantuml/docs/html/index.html` 文档:

```
google-chrome build/ex02_doxplantuml/docs/html/index.html
```

会有以下输出：

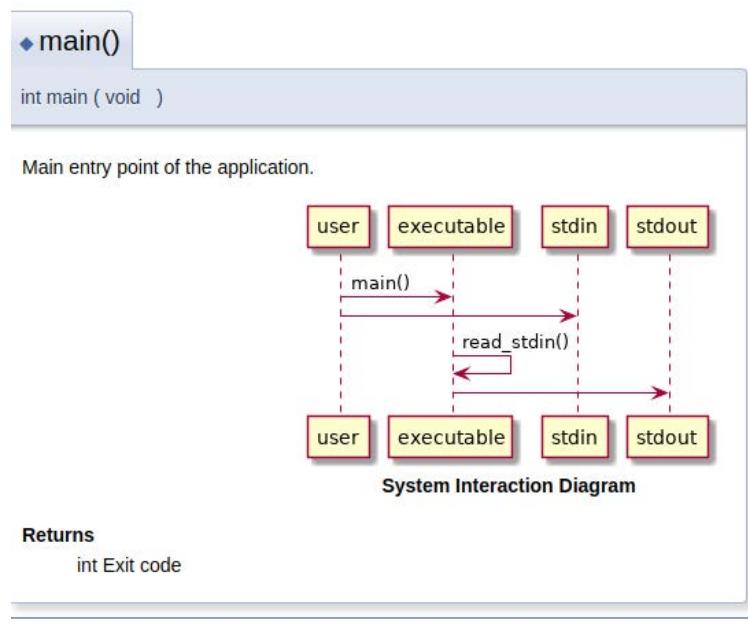


图 6.6 main() 函数文档中嵌入 PlantUML 图

图 6.6 中，可以看到 Doxygen 生成了 PlantUML 图，并将其嵌入到文档中。现在，就可以将自定义图嵌入到生成的文档中了，这可以让我们分析复杂的系统和依赖关系，而不必与外部绘图工具进行交互。

有了生成文档的正确工具，是时候学习如何打包和交付文档了。下一节中，我们将学习交付文档的方法，以及所涉及的软件。

### 6.3. 用 CPack 打包和分发文档

打包文档与打包软件及其工件没有什么不同——文档是项目的工件。因此，我们将使用在第 4 章中学到的知识来打包文档。若还没有阅读第 4 章，强烈建议在阅读本节之前阅读。

这一节，我们使用第 6 章的例 01，将使在此示例中生成的文档可安装和可打包。让我们回顾 CMakeLists.txt 文件，它位于 chapter06/ex01\_doxdocgen/文件夹中。使用以下代码，将使 HTML 和 MAN 文档可安装：

```
include(GNUInstallDirs)
install(DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/docs/html/"
       DESTINATION "${CMAKE_INSTALL_DOCDIR}"
       COMPONENT ch6_ex01_html)
install(DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}/docs/man/"
       DESTINATION "${CMAKE_INSTALL_MANDIR}"
       COMPONENT ch6_ex01_man)
```

第 4 章中使用 `install(DIRECTORY…)` 来安装文件夹，同时保留其结构。我们通过将 `docs/html` 和 `docs/man` 安装到 `GNUInstallDirs` 模块提供的默认文档和手册页目录中，使生成的文档可安装。另外，若是可安装的，就意味着也是可打包的，因为 CMake 能够从 `install(…)` 生成所需的打包代码。因此，包含 CPack 模块来为这个示例启用打包。代码如下所示：

```
set(CPACK_PACKAGE_NAME cbp_chapter6_example01)
set(CPACK_PACKAGE_VENDOR "CBP Authors")
set(CPACK_GENERATOR "DEB;RPM;TBZ2")
set(CPACK_DEBIAN_PACKAGE_MAINTAINER "CBP Authors")
include(CPack)
```

就这么简单。尝试使用以下命令来构建和打包示例项目：

```
cd chapter06/
cmake -S . -B build/
cmake --build build/
cpack --config build/CPackConfig.cmake -B build/pak
```

这里，配置并构建了 `chapter06/` 的代码，并使用 CPack 来使用生成的 `CPackConfig.cmake` 将项目打包到 `build/pak` 文件夹中。为了检查是否一切正常，通过使用以下命令将生成的包的内容提取到 `/tmp/ch6-ex01` 目录中：

```
dpkg -x build/pak/cbp_chapter6_example01-1.0-Linux.deb
/tmp/ch6-ex01
export MANPATH=/tmp/ch6-ex01/usr/share/man/
```

解压完成后，文档在 `/tmp/ch6-ex01/usr/share` 路径下可用。因为我们使用的是非默认路径，所以可以使用 `MANPATH` 环境变量让 `man` 命令知道文档的路径。首先检查是否可以通过调用 `man` 命令访问手册页：

```
man chapter6_ex01_calculator
```

`chapter6_ex01_calculator` 名称由 Doxygen 从 `chapter6::ex01::calculator` 类名称自动推导出来的，应该能够看到我们在前一节中介绍的手册页输出。

我们已经了解了关于生成和打包文档的知识。接下来，将学习如何生成 CMake 目标的依赖关系图。

## 6.4. 创建 CMake 目标的依赖关系图

我们已经讨论了软件代码的文档化和图形化，但在大型项目中，可能还需要文档化和可视化 CMake 代码。CMake 目标之间的关系可能很复杂，这可能使跟踪所有依赖关系变得困难。CMake 可以通过提供一个显示目标之间所有依赖关系的图表来帮助解决这个问题。通过 `cmake --graphviz=my-project.dot /path/to/build/dir`，CMake 将用 dot 创建包含目标如何相互依赖的文件。

DOT 语言是一种用于图形的描述语言，可以由许多程序进行解释，其中最著名的是免费的 Graphviz。DOT 文件可以使用 Graphviz 中的 DOT 命令行实用程序转换为图像，甚至可移植文档格式 (PDF) 文件:dot -Tpng filename.dot -o out.png。

为本章的示例项目运行这些命令将产生类似如下的输出:

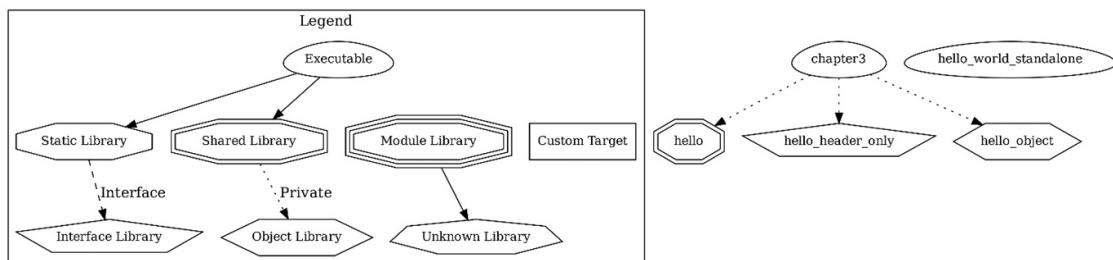


图 6.7 用 DOT 语言可视化的第 3 章的项目结构

行为和选项可以通过 `CMakeGraphVizOptions` 中提供的变量来控制。创建 DOT 图时，CMake 将查找一个名为 `CMakeGraphVizOptions` 的文件。若在 `PROJECT_SOURCE_DIR` 和 `PROJECT_BINARY_DIR` 目录中找到，将使用其中提供的值。配置文件的例子:

```
set(GRAPHVIZ_GRAPH_NAME "CMake Best Practices")
set(GRAPHVIZ_GENERATE_PER_TARGET FALSE)
set(GRAPHVIZ_GENERATE_DEPENDERS FALSE)
```

默认情况下，CMake 为所有目标创建依赖关系图。将 `GRAPHVIZ_GENERATE_PER_TARGET` 和 `GRAPHVIZ_GENERATE_DEPENDERS` 设置为 FALSE 将减少生成的文件数量。可以在 CMake 文档<https://cmake.org/cmake/help/latest/module/CMakeGraphVizOptions.html>中找到完整的选项集。

## 6.5. 总结

本章中，简要介绍了 Doxygen，并学习了如何从代码生成文档，以及如何将生成的文档打包用于部署。当涉及到软件项目时，掌握这些技能至关重要。从代码生成文档大大减少了技术文档的工作量，并且几乎无维护成本。从软件专业人员的角度来看，自动化确定性的东西并以不同的表示方式生成可推断的信息是最理想的，这种方法为其他需要更多人类解决问题技能的工程任务创造了空间和时间。自动化任务降低了维护成本，使产品更加稳定，并减少了对人力资源的总体需求。通过让机器做同样的工作，将人力转化为电力的方法，机器在执行确定性工作方面比人类表现得好得惊人。它们从来不会生病，很少损坏，而且很容易扩展，从来不会累，自动化是利用这种力量的一种方式。

本书的主要目的不是教你如何做事，而是如何让机器为特定的任务工作。这种方法确实需要首先学习，但若正在做一个成本很高的操作，而机器完全可以替代手动运行，那么继续手动完成这项工作就是在浪费宝贵的时间。投资于自动化——这是一项利润丰厚的投资，很快就能收回成本。

下一章中，将学习如何通过将单元测试、代码消毒、静态代码分析、微基准测试和代码覆盖工具集成到 CMake 项目中来提高代码质量，当然，我们也会将它们自动化。

## 6.6. 练习题

回答以下问题来测试对本章的理解：

1. Doxygen 是什么？
2. 将 Doxygen 集成到 CMake 项目中最简单的方法是什么？
3. Doxygen 会画图表吗？如果是，如何启用？
4. 应该使用哪些 Doxygen 标签将 PlantUML 图嵌入到 Doxygen 文档中？
5. 应该采取哪些配置步骤才能使 Doxygen 使用 PlantUML？
6. 假设手册/页面输出在 build/文件夹下，如何使文档可安装？

# 第 7 章 集成代码质量工具

之前关注的是构建和安装项目，以及生成文档和处理外部依赖关系。编写高质量软件时的另一项主要任务是测试，通过各种其他方法确保代码质量处于所需的水平。要实现高代码质量，仅仅编写单元测试，并偶尔执行它们是不够的。若想要生产高质量的软件，拥有与构建系统轻松集成的适当的测试工具是必要的。只有构建和测试能够毫不费力地一起工作，开发者才能专注于编写良好的测试，而不是专注于运行这些测试。测试驱动开发等方法为提高软件质量带来了巨大的价值。

不仅仅是编写测试来提高质量。编写测试是一种方式，使用覆盖报告检查测试的有效性，并使用静态代码分析确保通用代码质量是另一种方法。

虽然测试、覆盖和静态代码分析有助于确定代码是否按预期功能运行，但有些工具通常只使用特定的编译器或需要特殊的编译器设置。为了从这些工具中获益，可能需要用不同的编译器以不同的方式编译相同的源代码。但这正是 CMake 所擅长的，这就是为什么 CMake 可以通过使这些高质量的工具来帮助提高代码的质量。

许多用于确保代码高质量技术的好处是自动化。随着 CI/CD 系统的易用，为高质量的软件创建高度自动化检查相当容易，特别是使用 CMake，这些事情通常可以在定义软件如何构建的地方进行配置和执行。本章中，将学习如何使用 CMake 定义和编排测试，以及如何创建代码覆盖率报告以查看软件的哪些部分进行了测试。我们将介绍如何集成各种代码消毒器和静态代码分析器，以在编译时检查代码质量。同时，将展示包含所有工具的方法，以及如何为运行静态代码质量工具创建专用的构建类型。

我们将讨论以下主题：

- 定义、发现和运行测试
- 生成代码覆盖率报告
- 代码消杀
- 静态代码分析
- 创建自定义构建类型

## 7.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本

代码覆盖、消毒程序和静态代码分析的示例需要运行 GCC 或 Clang，不能与 MSVC 一起工作。要在 Windows 上运行 Clang，请参阅第 8 章，其中介绍了工具链文件。有些示例需要安装 Catch2 单元测试套件进行编译。有些例子从不同的在线位置提取依赖项，因此还需要网络连接。

除了可用的编译器，示例需要以下软件：

- Gcov 和 Gcovr 的代码覆盖率示例
- Cppcheck、Cpplint 和 include-what-you-use 用于静态代码分析器的示例

所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是<https://github.com/PacktPublishing/CMake-Best-Practices>。

若缺少相应的软件，应将相应的示例将从构建中排除。

## 7.2. 定义、发现和运行测试

测试是以软件质量为傲的软件工程师的主要方式。目前已有很多单元测试框架，特别是对于 C++，CMake 包含了可与多数流行框架一起使用的模块。

总的来说，所有单元测试框架都会做以下工作：

- 制定和分组测试用例。
- 包含某种形式的断言，以检查各种测试条件。
- 发现并运行测试用例，可以是全部的，也可以是其中的一个选择。
- 生成各种格式的测试结果，例如纯文本、JSON、XML，可能还有更多。

通过 CTest，CMake 可以以内置的方法来执行测试。设置 `enable_testing()`，并使用 `add_test()` 添加了测试的 CMake 项目都支持运行测试。`enable_testing()` 将在当前目录和其子目录中启用，并添加测试。因此在使用 `add_subdirectory` 前，通常将其设置在顶层的 `CMakeLists.txt` 中。若使用 `include(CTest)`，CMake 的 CTest 模块会自动设置 `enable_testing`，除非 `BUILD_TESTING` 为 OFF。

根据 `BUILD_TESTING` 选项禁用构建和运行测试也是一种很好的实践。这里的常用模式是将项目中与测试相关的所有部分放在其子文件夹中，并且只在 `BUILD_TESTING` 设置为 ON 时包含子文件夹。

CTest 模块通常应该只包含在项目的顶层 `CMakeLists.txt` 中。自从 CMake 版本 3.21 以来，`PROJECT_IS_TOP_LEVEL` 可以用来测试当前的 `CMakeLists.txt` 是否为顶层文件。对于项目的顶层目录和使用 `ExternalProject` 添加的项目顶层目录，此变量为 `True`。对于使用 `add_subdirectory` 或 `FetchContent` 添加的目录，该值为 `False`。因此，CTest 应该包括如下内容：

```
project(CMakeBestPractice)
...
if(PROJECT_IS_TOP_LEVEL)
    include(CTest)
endif()
```

单元测试本质上是在内部运行一系列断言的小程序，若断言失败，将返回一个非零返回值。有许多框架和库可以帮助组织测试和编写断言，但从外部来看，检查断言和返回相应的值是核心功能。

可以使用 `add_test` 将测试添加到 `CMakeLists.txt` 中：

```
add_test(NAME <name> COMMAND <command> [<arg>...]
[CONFIGURATIONS <config>...]
[WORKING_DIRECTORY <dir>]
[COMMAND_EXPAND_LISTS])
```

COMMAND 可以是项目中定义的可执行目标，也可以是可执行文件的完整路径，测试所需的参数也包括在内。因为 CMake 会自动替换到可执行文件的路径，所以使用目标名称为首选。CONFIGURATION 选项用于判断对哪个构建配置有效。对于大多数测试用例来说，这并不重要，但是对于微基准测试来说，就非常有用，WORKING\_DIRECTORY 应该是一个绝对路径。默认情况下，测试在 CMAKE\_CURRENT\_BINARY\_DIR 中执行。COMMAND\_EXPAND\_LISTS 确保传递给作为 COMMAND 选项的列表都会展开。

包含测试的小项目：

```
cmake_minimum_required(VERSION 3.21)
project("simple_test" VERSION 1.0)

enable_testing()

add_executable(simple_test)
target_sources(simple_test PRIVATE src/main.cpp)

add_test(NAME example_test COMMAND simple_test)
```

本例中，一个名为 simple\_test 的可执行目标用作 example\_test 的测试。

CTest 将使用有关测试的信息并执行它们。通过独立运行 ctest 或作为 CMake 构建步骤的一部分，来执行测试。以下两个命令都可执行测试：

```
ctest --test-dir <build_dir>
cmake --build <build_dir> --target test
```

调用 CTest 作为构建目标的好处是，CMake 会首先检查所有需要的目标是否构建了，并且在最新的版本上 CTest 也可以这样做：

```
ctest --build-and-test <source_dir> <build_dir>
```

ctest 的输出看起来可能像这样：

```
Test project /workspaces/CMake-Best-Practices/build
  Start 1: example_test
1/3 Test #1: example_test .....***Failed
0.00 sec
  Start 2: pass_fail_test
2/3 Test #2: pass_fail_test ..... Passed
```

```

0.00 sec
    Start 3: timeout_test
3/3 Test #3: timeout_test ..... Passed
0.50 sec

67% tests passed, 1 tests failed out of 3

Total Test time (real) = 0.51 sec

The following tests FAILED:
    1 - example_test (Failed)
Errors while running CTest
Output from these tests are in:
/wkspaces/CMake-Best-Practices/build/Testing/Temporary/LastTest
Use "--rerun-failed --output-on-failure" to re-run the failed
cases verbosely.

```

通常，测试静默所有输出到标准输出。通过-V 或--verbose 命令行参数，可将测试输出到屏幕。但通常只对失败测试的输出感兴趣，使用--output-on-failure 通常是更好的选择，只有失败的测试才会产生输出。对于非常详细的测试，可以使用--test-output-size-passed <size> 和--test-output-size-failed <size> 选项来限制输出的大小，其中大小是字节 (byte) 数。

构建树中有一个或多个 add\_test 将会在 CMAKE\_CURRENT\_BINARY\_DIR 中为 CTest 生成一个输入文件。CTest 的输入文件不一定位于项目的顶层，而是在定义的地方。要列出所有测试但不执行，可以使用 CTest 的-N 选项。

CTest 的特性会在运行之间缓存测试状态，可以只运行上次运行失败的测试。为此，运行 ctest --rerun-failed 将只运行上次运行中失败的测试。

有时，若要修复单个失败的测试，则不希望执行完整的测试集。-E 和-R 命令行选项接受与测试名称匹配的正则表达式 (正则表达式)。-E 选项排除匹配模式的测试，-R 选项选择要包含的测试。这些选项可以组合使用。以下命令将运行所有以 FeatureX 开头的测试，但不包括名为 FeatureX\_Test\_1 的测试：

```
ctest -R ^FeatureX -E FeatureX_Test_1
```

有选择地执行测试的另一种方法是使用 LABELS 属性进行标记，然后使用 CTest 的-L 选项选择要运行的标签。一个测试可以有多个用分号分隔的标签：

```
add_test(NAME labeled_test_1 COMMAND someTest)
set_tests_properties(labeled_test PROPERTIES LABELS "example")
```

```
add_test(NAME labeled_test_2 COMMAND anotherTest)
set_tests_properties(labeled_test_2 PROPERTIES LABELS "will_fail" )

add_test(NAME labeled_test_3 COMMAND YetAnotherText)
set_tests_properties(labeled_test_3
    PROPERTIES LABELS "example;will_fail")
```

-L 命令行选项接受一个正则表达式来过滤标签:

```
ctest -L example
```

这将只执行 labeled\_test\_1 和 labeled\_test\_3，因为它们都分配了 example 标签。

通过制定相应的正则表达式，可以组合多个标签:

```
ctest -L "example|will_fail"
```

这将执行示例中的所有测试，但不会执行没有分配标签的测试。

使用标签将有助于标记设计为失败或类似的测试，或标记仅与某些执行上下文中相关的测试。

对于基于正则表达式或标签的测试选择，最后一种选择是使用-I 选项，它接受分配的测试编号。

-I 选项的参数有点复杂:

```
ctest -I [Start,End,Stride,test#,test#,...|Test file]
```

通过 Start、End 和 Stride，可以指定要执行的测试的范围。这三个数字是与显式测试数字 test# 相结合的范围，或传递包含参数的文件。

下面的调用将执行从 1 到 10 的所有奇数测试:

```
ctest -I 1,10,2
```

因此，将执行测试 1、3、5、7 和 9。以下命令将只执行测试和 8:

```
ctest -I ,0,,6,7
```

这个调用中，End 设置为 0，因此没有执行。要合并范围和显式测试数，以下命令将执行从 1 到 10 的所有奇数测试，并额外执行测试 6 和 8:

```
ctest -I 1,10,2,6,8
```

-I 选项处理繁琐，加上添加新测试可能会重新分配数字，这是在实践中很少使用它的两个原因。通常，根据标签或测试名称进行过滤是首选方式。

编写测试时的另一个常见问题是它们不够独立。因此，test 2 可能会意外地依赖于 test 1 的前一次执行。为了避免这种依赖，CTest 可使用--schedule-random 参数随机化测试执行顺序。这将确保测试以任意顺序执行。

### 7.2.1 自动发现测试

`add_test` 定义测试是向 CTest 添加测试的一种方法，缺点是这会将整个可执行文件注册为单个测试。通常，单个可执行文件将包含多个单元测试，不仅仅是一个。当可执行文件中的一个测试失败时，可能很难确定到底哪个测试失败了。

以下测试代码的 C++ 文件，假设 Fibonacci 函数包含一个错误，那么 `Fibonacci(0)` 将不会像它应该返回的那样返回 1：

```
1 TEST_CASE("Fibonacci(0) returns 1"){ REQUIRE(Fibonacci(0) == 1); }
2 TEST_CASE("Fibonacci(1) returns 1"){ REQUIRE(Fibonacci(1) == 1); }
3 TEST_CASE("Fibonacci(2) returns 2"){ REQUIRE(Fibonacci(2) == 2); }
4 TEST_CASE("Fibonacci(5) returns 8"){ REQUIRE(Fibonacci(5) == 8); }
```

若所有这些测试都编译到同一个名为 `Fibonacci` 的可执行文件中，那么使用 `add_test` 添加只会指示可执行文件失败，而不会指示前面代码块中看到的哪个场景失败。

测试结果看起来像这样：

```
Test project /workspaces/CMake-Best-Practices/build
Start 5: Fibonacci
1/1 Test #5: Fibonacci .....***Failed
0.00 sec
0% tests passed, 1 tests failed out of 1
Total Test time (real) = 0.01 sec
The following tests FAILED:
  5 - Fibonacci (Failed)
```

这对找出哪个测试用例失败没什么帮助，但在 Catch2 和 GoogleTest 中，有一种方法可以将内部测试公开给 CTest，从而将它们作为常规测试执行。对于 GoogleTest，这样做的模块是由 CMake 本身提供的；Catch2 在自己的 CMake 集成中提供了这个功能。使用 Catch2 发现测试由 `catch_discover_tests` 完成，而对于 GoogleTest，则使用 `gtest_discover_tests`。以下示例在 Catch2 框架中编写的测试，并添加到 CTest：

```
find_package(Catch2)
```

```
include(Catch)

add_executable(Fibonacci)
catch_discover_tests(Fibonacci)
```

为了使函数可用，必须包含 Catch 模块。对于 GoogleTest，工作原理相似：

```
include(GoogleTest)

add_executable(Fibonacci)
gtest_discover_tests(Fibonacci)
```

使用发现功能时，测试可执行文件中定义的每个测试用例都将视为 CTest 自己的测试，CTest 的结果可能如下所示：

```
Start 5: Fibonacci(0) returns 1
1/4 Test #5: Fibonacci(0) returns 1 .....***Failed 0.00 sec
    Start 6: Fibonacci(1) returns 1
2/4 Test #6: Fibonacci(1) returns 1 ..... Passed 0.00 sec
    Start 7: Fibonacci(2) returns 2
3/4 Test #7: Fibonacci(2) returns 2 ..... Passed 0.00 sec
    Start 8: Fibonacci(5) returns 8
4/4 Test #8: Fibonacci(5) returns 8 ..... Passed 0.00 sec

75% tests passed, 1 tests failed out of 4
Total Test time (real) = 0.02 sec
The following tests FAILED:
  5 - Fibonacci(0) returns 1 (Failed)
```

现在，可以看到哪些定义的测试用例失败了，Fibonacci(0) 返回 1 个测试用例没有像预期的那样运行。当使用带有集成测试功能的编辑器或 IDE 时，这很方便。这两个发现函数都通过运行指定的可执行文件来工作，该可执行文件有一个选项，即只打印测试名称并在 CTest 内部注册，因此每个构建步骤都会增加少量开销。更细粒度地发现测试还有一个好处，即 CMake 可以更好地并行执行测试。

gtest\_discover\_tests 和 catch\_discover\_tests 也有丰富的选项，比如为测试名称添加前缀或后缀，或者为生成的测试添加一组属性。函数的完整文档可以在这里找到：

- Catch2: <https://github.com/catchorg/Catch2/blob/devel/docs/cmake-integration.md>
- GoogleTest: <https://cmake.org/cmake/help/v3.21/module/GoogleTest.html>

Catch2 和 GoogleTest 只是众多测试框架中的两个;可能有更多的测试套件自带这种功能,而这些功能可能是作者不知道的。现在,让我们从寻找测试开始,进一步了解如何控制测试的行为。

### 7.2.2 确定测试成功或失败的高级方法

通常,CTest 根据命令的返回值确定测试是失败还是通过。0 表示所有测试都成功,否则将解释为失败。

有时,返回值不足以确定测试通过或失败。若需要检查某个字符串的程序输出,可以使用 FAIL\_REGULAR\_EXPRESSION 和 PASS\_REGULAR\_EXPRESSION 测试属性:

```
set_tests_properties(some_test PROPERTIES  
    FAIL_REGULAR_EXPRESSION "[W|w]arning|[E|e]rror"  
    PASS_REGULAR_EXPRESSION "[S|s]uccess")
```

若输出包含“Warning”或“Error”,这些属性将导致 some\_test 测试失败。若有“Success”字符串,则认为测试通过。如果设置了 PASS\_REGULAR\_EXPRESSION,则只有该字符串存在,测试才会认为通过。这两种情况下,将忽略返回值。若需要忽略测试的某个返回值,可以使用 SKIP\_RETURN\_CODE 选项。

有时,会需要测试失败,设置 WILL\_FAIL 为 true 将导致测试结果颠倒:

```
add_test(NAME SomeFailingTest COMMAND SomeFailingTest)  
set_tests_properties(SomeFailingTest PROPERTIES WILL_FAIL True)
```

这通常比禁用测试要好,因为测试在每次测试运行时仍然会执行,若测试意外地再次开始通过,开发人员会知道。测试失败的一种特殊情况,是当测试无法返回或需要花费太多时间才能完成。对于这种情况,CTest 提供了添加测试超时的方法,甚至在失败的情况下重新测试。

### 7.2.3 处理超时和重复的测试

有时,不仅对测试的成功或失败感兴趣,还对完成测试需要多长时间感兴趣。TIMEOUT 测试属性需要数秒来确定测试的最大运行时。若测试超过时间,则终止测试并认为测试失败。以下命令将限制测试的测试执行时间为 10 秒:

```
set_tests_properties(timeout_test PROPERTIES TIMEOUT 10)
```

对于有陷入无限循环或由于某种原因永远挂起的风险的测试,TIMEOUT 属性就很好用。

或者,CTest 接受--timeout 参数来设置全局超时时限,该时限应用于没有指定 TIMEOUT 属性的所有测试。对于那些定义了 TIMEOUT 的测试,CMakeLists.txt 中定义的超时,优先于通过命令行传递的时限。

为了避免长时间的测试执行,CTest 命令行接受--stop-time 参数,将当天的实时作为完整测试集的时间限制。下面的命令将为每个测试设置一个默认的 30 秒超时,并且测试必须在 23:59 之前完成:

```
ctest --timeout 30 --stop-time 23:59
```

有时，由于我们无法控制的因素，测试可能会偶尔出现超时。通常是需要某种形式的网络通信或具有某种带宽限制的资源的测试，让测试运行的唯一方法是再次尝试。为此，`--repeat aftertimeout:n` 命令行参数可以传递给 CTest，其中 `n` 是一个数字。

`--repeat` 参数有三个选项：

- `after-timeout`: 若发生超时，这将重试测试多次。当超时后重复测试，`--timeout` 选项应该传递给 CTest。
- `until-pass`: 这将重新运行测试，直到测试通过或达到重试次数为止，将此设置为 CI 环境中的规则并不是一个好主意。
- `until-fail`: 测试将重新运行多次，直到它们失败。若测试偶尔失败，则经常使用此方法，以了解这种情况发生的频率。`--repeatuntil-fail` 参数的工作原理完全类似于`--repeat:until-fail:n`。

如前所述，测试失败的原因可能是测试所依赖的资源不可用。外部资源不可用的情况是它们被来自测试的请求淹没。访问外部资源时超时的另一个常见原因是，当测试运行时该资源不可用。下一节中，将了解如何编写用于确保在测试运行之前启动资源的测试固件。

#### 7.2.4 编写测试固件

通常，测试应该彼此独立。有时，测试可能依赖于不受测试本身控制的前提条件。例如，为了测试客户端，测试可能需要运行一个服务器。这些依赖关系可以通过使用 `FIXTURE_SETUP`、`FIXTURE_CLEANUP` 和 `FIXTURE_REQUIRED` 测试属性将它们定义为测试固件。所有三个属性都接受一个字符串列表来标识固件，测试可以通过定义 `FIXTURE_REQUIRED` 属性来指定相应的固件。这将确保固件测试在执行前成功完成。类似地，测试可以在 `FIXTURE_CLEANUP` 中进行声明，以表明必须在该固件测试完成后方可运行。无论测试是成功还是失败，清理部分中定义的固件总是会运行：

```
add_test(NAME start_server COMMAND echo_server --start)
set_tests_properties(start_server PROPERTIES FIXTURES_SETUP server)
add_test(NAME stop_server COMMAND echo_server --stop)
set_tests_properties(stop_server PROPERTIES FIXTURES_CLEANUP server)

add_test(NAME client_test COMMAND echo_client)
set_tests_properties(client_test PROPERTIES FIXTURES_REQUIRED server)
```

本例中，名为 `echo_server` 的程序当作固件，以便另一个 `echo_client` 的程序使用它。带有`--start` 和`--stop` 参数的 `echo_server` 的执行名称带有 `start_server` 和 `stop_server` 的测试。`start_server` 测试标记为带有名称 `server` 的固件设置。`stop_server` 测试也进行了类似设置，但标记为固件清理例程。最后，将设置名为 `client_test` 的实际测试，并将它作为服务器固件必需通过的先决条件。

现在，若使用 CTest 运行 `client_test` 测试，则会自动调用固件。固件测试在 CTest 的输出中显示为常规测试：

```
ctest -R client
Test project CMake-Best-Practices:
  Start 9: start_server
1/3 Test #9: start_server ..... Passed 0.00 sec
  Start 11: client_test
```

```
2/3 Test #11: client_test..... Passed 0.00 sec
  Start 10: stop_server
3/3 Test #10: stop_server ..... Passed 0.00 sec
```

CTest 用只匹配客户端测试的正则表达式筛选器，但 CTest 还是会启动固件。为了在并行执行测试时，不使测试固件压力过重，可以将它们定义为资源。

### 7.2.5 管理测试资源——并行运行测试

若项目有许多测试，那并行执行它们将加快测试的速度。默认情况下，CTest 按顺序运行测试，通过将-j 选项传递给 CTest 调用，这些测试可以并行运行，或并行线程数量可以在 CTEST\_PARALLEL\_LEVEL 环境变量中定义。通常，CTest 假定每个测试将在单个 CPU 上运行。若测试需要多个处理器才能成功运行，可以设置测试的 PROCESSORS 属性来定义所需的处理器数量：

```
add_test(NAME concurrency_test COMMAND concurrency_tests)
set_tests_properties(concurrency_test PROPERTIES PROCESSORS 2)
```

concurrency\_test 测试需要两个 CPU 才能运行。当使用-j 8 并行运行测试时，concurrency\_test 将占用八个可用的并行执行“槽”中的两个。本例中，若 PROCESSORS 属性设置为 8，则没有其他测试可以与 concurrency\_test 并行运行。当为 PROCESSORS 设置的值高于系统上可用的并行槽数或 CPU 数时，就会在有 CPU 空闲的时候，立即运行测试。有时，有些测试不仅需要特定数量的处理器，而且需要在不运行其他测试的情况下单独运行。为了实现这一点，RUN\_SERIAL 属性可以在测试中设置为 true。这会对整体测试性能产生严重影响，因此要谨慎使用。更细粒度的控制方法是使用 RESOURCE\_LOCK 属性，该属性包含一个字符串列表。字符串没有特殊含义，除非 CTest 阻止两个测试并列运行(若列出的字符串相同)。通过这种方式，可以在不停止整个测试执行的情况下实现部分序列化，这也是指定测试是否需要特定的唯一资源的好方法，例如对某个文件、数据库或类似的东西进行测试时。考虑下面的例子：

```
set_tests_properties(database_test_1 database_test_2
  database_test_3 PROPERTIES RESOURCE_LOCK database)
set_tests_properties(some_other_test PROPERTIES RESOURCE_LOCK
  fileX)
set_tests_properties(yet_another_test PROPERTIES RESOURCE_LOCK
```

```
"database;fileX ")
```

本例中, database\_test\_1、database\_test\_2 和 database\_test\_3 测试将禁止并行运行。some\_other\_test 测试将不受数据库测试的影响, 但 yet\_another\_test 将不会与数据库测试和 some\_other\_test 一起运行。

### 固件为源

虽然技术上不需要, 但若 RESOURCE\_LOCK 与 FIXTURE\_SETUP、FIXTURE\_CLEANUP 和 FIXTURE\_REQUIRED 一起使用, 那么为相同的资源使用相同的标识符也是一种很好的方式。

当测试需要独占访问某些资源时, 使用 RESOURCE\_LOCK 管理测试的并行性非常方便。大多数情况下, 管理并行性就完全足够了。从 CMake 3.16 开始, 可以通过 RESOURCE\_GROUPS 属性在更细粒度的级别上进行控制。资源组不仅允许指定使用什么资源, 还允许指定使用多少资源。常见的场景是定义特定贪婪操作可能需要的内存量, 或者避免超过特定服务的连接限制。开发使用 GPU 进行通用计算的项目时, 资源组通常会发挥作用, 以定义每个测试需要多少个 GPU 插槽。

与简单的资源锁相比, 资源组的复杂性高了很多。要使用它们, CTest 必须做以下事情:

- 了解测试需要运行哪些资源, 是通过设置项目中的测试属性来定义的。
- 了解系统有哪些可用资源, 是在运行测试时从项目外部完成的。
- 传递关于使用哪些资源进行测试的信息, 是通过使用环境变量来实现的。

与资源锁一样, 资源组是用于标识资源的字符串。绑定到标签的实际资源的定义留给用户。资源组定义为名称: 值对, 若有多个组, 则以逗号分隔。测试可以用 RESOURCE\_GROUPS 属性定义使用什么资源:

```
set_property(TEST SomeTest PROPERTY RESOURCE_GROUPS
  cpus:2,mem_mb:500
  servers:1,clients:1
  servers:1,clients:2
  4,servers:1,clients:1
)
```

前面的示例中, SomeTest 表示它使用两个 CPU 和 500 MB 内存。总共使用了一个客户机-服务器对的 6 个实例, 每对都分配了一些服务器和客户机。第一对包含一个服务器实例和一个客户端实例, 第二对需要一个服务器和两个客户端实例。

最后一行, 4,servers:1,clients:1, 使用同一对的四个实例, 由一个服务器资源和一个客户机资源组成。除了所需的 CPU 和内存外, 除非总共有 6 个服务器和 7 个客户机可用, 否则此测试将不会运行。

可用的系统资源在传递给 CTest 的 JSON 文件中指定, 可以通过 CTest --resource-spec-file 命令行参数指定, 也可以通过调用 CMake 时设置 CTEST\_RESOURCE\_SPEC\_FILE 变量指定。设置变量应该通过使用 cmake -D 来完成, 而不是在 CMakeLists.txt 中直接设置, 指定系统资源应该从项目外部完成。

上面示例的资源指定文件如下所示:

```
{  
    "version": {  
        "major": 1,  
        "minor": 0  
    },  
    "local": [  
        {  
            "mem_mb": [  
                {  
                    "id": "memory_pool_0",  
                    "slots": 4096  
                }  
            ],  
            "cpus": [  
                {  
                    "id": "cpu_0",  
                    "slots": 8  
                }  
            ],  
            "servers": [  
                {  
                    "id": "0",  
                    "slots": 4  
                },  
                {  
                    "id": "1",  
                    "slots": 4  
                }  
            ],  
            "clients": [  
                {  
                    "id": "0",  
                    "slots": 8  
                },  
                {  
                    "id": "1",  
                    "slots": 8  
                }  
            ]  
        }  
    ]  
}
```

```
    ]  
}  
]  
}
```

该文件指定一个系统，共有 4,096 MB 内存、8 个 CPU、2x4 服务器实例和 2x8 客户端实例，共包含 8 个服务器和 16 个客户端。若测试的资源请求不能满足可用的系统资源，将无法运行测试，会出现如下错误：

```
ctest -j $(nproc) --resource-spec-file ./resources.json  
  
Test project /workspaces/CMake-Best-Practices/chapter07  
/resource_group_example/build  
Start 2: resource_test_2  
          Start 3: resource_test_3  
Insufficient resources for test resource_test_3:  
  
Test requested resources of type 'mem_mb' in the following  
amounts:  
8096 slots  
but only the following units were available:  
'memory_pool_0': 4096 slots  
  
Resource spec file:  
  
./resources.json
```

当前示例将能够在此环境下运行，因为它总共需要 6 个服务器和 7 个客户机。CTest 没有办法确保指定的资源是否实际可用，这是用户或 CI 系统需要确定的事情。例如，一个资源文件可能指定有八个可用的 CPU，而硬件实际上只有四个核。

关于已分配资源组的信息通过环境变量传递给测试，CTEST\_RESOURCE\_GROUP\_COUNT 环境变量指定分配给测试的资源组的总数。若没有设置，则在没有环境文件的情况下使用 CTest。测试应该检查这一点并相应地采取行动，若一个测试在没有资源的情况下不能运行，要么失败，要么通过返回各自的返回代码或在 SKIP\_RETURN\_CODE 或 SKIP\_REGULAR\_EXPRESSION 属性中定义的字符串来指示它没有运行。分配给测试的资源组通过成对的环境变量指定：

- CTEST\_RESOURCE\_GROUP\_<ID> 将包含资源组的类型。前面的示例中，为”mem\_mb”、“cpus”、“clients”或”servers”。
- CTEST\_RESOURCE\_GROUP\_<ID>\_<TYPE>，将包含用于类型的一对 id:slots。

这取决于如何使用和内部分布资源组的测试实现。

编写和运行测试显然是提高代码质量的主要方式。然而，另一个有趣的指标通常是测试实际覆盖了多少代码。调查和报告代码覆盖率可以提供有趣的提示，这不仅是关于软件测试的范围，还关乎于差异。

## 7.3. 生成代码覆盖率报告

了解代码有多少进行了测试有很大好处，这会对给定软件的测试情况有一个好的印象。还可以向开发人员提供关于测试未涉及的执行路径和边缘用例的提示。

有一些工具可以在 C++ 中实现代码覆盖率，最流行的是 GNU 的 Gcov。它已经存在多年了，可以很好地与 GCC 和 Clang 一起工作。它不能与微软的 Visual Studio 一起工作，但是使用 Clang 来构建和运行该软件为 Windows 提供了一个可行的替代方案。或者，可以使用工具 OpenCppCoverage 用 MSVC 构建，在 Windows 上获取覆盖数据。

Gcov 生成的覆盖率信息可以通过 Gcovr 或 LCOV 工具在总结报告中收集。

### 7.3.1 生成覆盖率报告

本节中，将了解如何使用 Gcovr 创建这样的报告。生成代码覆盖率报告的大致方式如下：

1. 要测试的程序和库用特殊的标志编译，因此需要公开覆盖率信息。
2. 程序运行，覆盖率信息存储在一个文件中。
3. 覆盖分析程序，如 Gcovr 或 LCOV，分析覆盖文件并生成报告。
4. 可以存储或进一步分析报告，以显示覆盖率的趋势。

代码覆盖率的一个常见设置是，获得关于单元测试覆盖了多少项目代码的信息。为了做到这一点，必须使用必要的标志来编译代码，以便公开信息。

`<LANG>_COMPILER_FLAGS` 缓存变量应该通过命令行传递给 CMake。当使用 GCC 或 Clang 时，命令可能会是这样：

```
cmake -S <sourceDir> -B <BuildDir> -DCMAKE_CXX_FLAGS=-fprofile-generate
```

另一种方法是定义各自的预设，详见第 9 章。构建覆盖测试时，在编译时启用调试信息并使用`-Og` 标志禁用优化。此外，指定`-fkeep-inline-functions` 和 `-fkeep-staticconsts` 编译器标志将阻止优化静态和内联函数（若从未使用过的话）。这将确保所有可能执行的分支都编译到代码中，否则，覆盖率报告可能会产生误导，特别是对于内联函数。

覆盖率报告不仅适用于单个可执行程序，还适用于库。然而，编译库时也必须打开覆盖率标志。

由于覆盖的编译器标志是全局的，选项会传递给使用 `FetchContent` 或 `ExternalProject` 添加的项目，这可能会增加编译时间。

启用覆盖标志的情况下，使用 GCC 或 Clang 编译源代码将为每个目标文件和可执行文件在构建目录中创建`.gcno` 文件。这些文件包含 Gcov 的元信息，关于哪些调用和执行路径在各自的编译单元中可用。为了找出使用了哪些路径，必须运行程序。

对于想要找出测试的代码覆盖率的设置，运行 `ctest` 将生成覆盖率结果。或者，直接运行可执行程序也会产生相同的结果。运行启用覆盖的可执行程序将在构建目录中生成`.gcda` 文件，其中包含有关各自目标文件中调用的信息。

生成这些文件后，运行 `Gcovr` 将创建关于覆盖率的信息。默认情况下，`Gcovr` 将信息输出到标准输出，也可以生成 HTML 页面、JSON 文件或 SonarQube 报告。

```
gcovr -r <SOURCE_DIR> <BINARY_DIR> -html
```

这样会产生一个 HTML 文件：

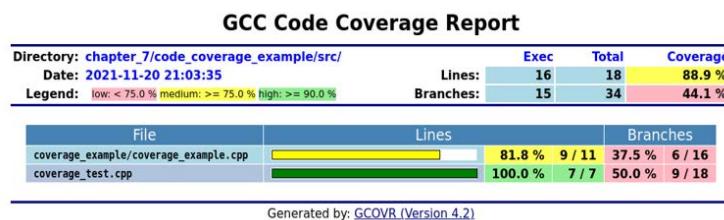


图 7.1 覆盖测试运行的示例输出

`Gcovr` 的另一个替代方案是 `LCOV`，其工作原理与 `Gcovr` 非常相似。与 `Gcovr` 相比，`LCOV` 不能直接产生 HTML 或 XML 输出，而是将覆盖率信息以中间格式组合，然后可由各种转换器使用。为了生成 HTML 输出，经常使用 `genhtml`。要使用 `LCOV` 生成报告，命令如下所示：

```
lcov -c -d <BINARY_DIR> -o <OUTPUT_FILE>
genhtml -o <HTML_OUTPUT_PATH> <LCOV_OUTPUT>
```

使用 `LCOV` 生成的覆盖率报告可能如下所示：



图 7.2 使用 LCOV 生成的覆盖率报告示例

这只会为最后一次运行创建覆盖率报告。若想将它们组合成一个时间序列，以查看代码覆盖率是上升还是下降，可以使用各种 CI 工具，如 Codecov 和 Cobertura 来完成此工作。这些工具通常可以解析来自 Gcovr 或 LCOV 的输出，并将其组装成显示覆盖率趋势。Gcovr 的详细文档可参见<https://gcovr.com/en/stable/>。

### 7.3.2 为 MSVC 创建覆盖率报告

当使用 Microsoft Visual Studio 构建软件时，OpenCppCoverage 工具是 Gcov 的替代方案，其工作原理是分析 MSVC 编译器生成的程序数据库 (.pdb)，而不是用标志编译源代码。为单个可执行文件生成 HTML 覆盖率报告的命令可能如下所示：

```
OpenCppCoverage.exe --export_type html:coverage.html --
MyProgram.exe arg1 arg2
```

由于这只会为单个可执行文件生成覆盖率报告，OpenCppCoverage 可以读取前几轮的输入，并将它们组合成如下报告：

```
OpenCppCoverage.exe --export_type binary:program1.cov --
program1.exe
OpenCppCoverage.exe --export_type binary:program2.cov --
program2.exe
OpenCppCoverage.exe --input_coverage=program1.cov --input_
coverage= program2.cov --export_type html:coverage.html
```

这将把前两次运行的输出合并到一个公共报告中。要使用覆盖率信息，`export_type` 选项必须是 `binary`。

覆盖率报告的常见用途是找出项目中定义的测试覆盖了多少代码，由于 CTest 将运行测试作为子进程，`--cover_children` 选项必须传递给 OpenCppCoverage。为了避免为系统库生成覆盖率报告，可能需要添加一个模块和一个源过滤器。命令看起来像这样：

```
OpenCppCoverage.exe --cover_children --modules <build_dir> --
sources <source_dir> -- ctest.exe --build-config Debug
```

这种方法的小缺点是覆盖率报告将包括 CTest 本身的覆盖率。生成的 HTML 报告可能像这样：

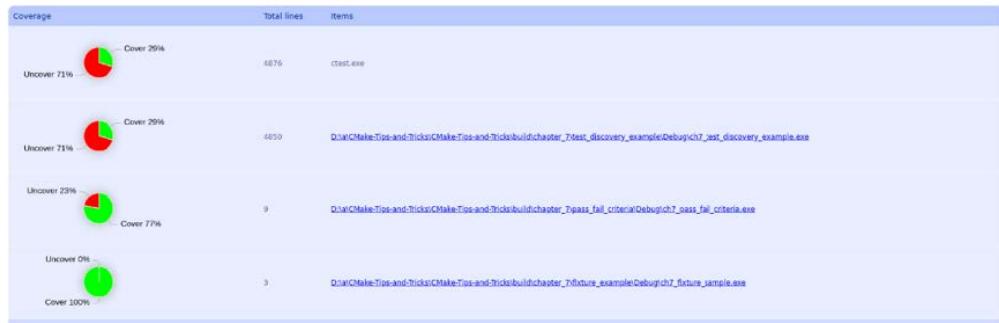


图 7.3 使用 OpenCppCoverage 生成的覆盖率报告

命令行之外的另一种选择是使用 Visual Studio 的插件，插件可以在 Visual Studio 市场上找到：<https://marketplace.visualstudio.com/items?itemName=OpenCppCoverage.OpenCppCoveragePlugin>。

完整的文档，请参考 OpenCppCoverage 的 GitHub 页面：<https://github.com/OpenCppCoverage/OpenCppCoverage>

了解所提供的测试覆盖了多少代码，是代码质量非常重要的信息。在许多受监管的行业（如医疗技术、航空或汽车行业）中，监管机构可能会要求提供代码覆盖率报告。然而，仅仅知道执行了多少代码显然是不够的，底层代码的质量更加重要。一些编译器在消毒器的帮助下，提供有用的工具来检测代码中的常见错误。下一节中，将学习如何使用 CMake 使用和应用消毒器。

## 7.4. 代码消杀

如今的编译器不仅可以将文本转换为二进制，还是一组复杂的软件套件，具有确保代码质量的内置功能。特别是随着 LLVM 和 Clang 的出现，对于编译器对代码质量问题的了解程度显得非常重要。这些质量工具通常称为消毒器，通过将某些标志传递给编译器和链接器来启用。

代码消毒器是将质量检查引入代码的方法，编译器用注释和钩子装饰二进制代码，以检测各种运行时问题。当执行代码时，检查并确认注释是否有任何违规行为。消毒程序相对较快，会对程序的运行时行为产生影响。若消毒程序进行了捕获，则会使用 `abort()` 终止程序并返回非零错误码。这在测试中特别有用，任何违反消毒器的测试都将标记为失败。

以下是最常见的消毒器：

- 地址消毒程序 (ASan) 检测内存访问错误，如越界和释放后使用错误。
- 泄漏消毒器 (LSan) 是 ASan 的一部分，可用于检测内存泄漏。
- GCC 和 Clang 中，有一些通用 ASan 的定制版本，例如用于检测 Linux 内核中的内存错误的内核地址消毒程序 (KASAN)。
- 某些平台上，ASans 甚至可以在硬件辅助下运行。
- 内存消毒器 (MSan) 检测未初始化的内存读取。
- 线程消毒器 (TSan) 将报告数据竞争。由于 TSan 的工作方式，不能与 ASan 和 LSan 一起运行。
- 未定义行为消毒器 (UBSan) 检测并报告代码导致未定义行为的情况，初始化前使用变量或操作符优先级不明确是常见的例子。

Clang 套件在消毒器可用性方面非常领先，GCC 紧随其后。微软在适应这些特性方面有点慢，但也已经开始在编译器中包括消毒程序。撰写本书时，Visual Studio 19 附带的 MSVC 版本 16.9 是第一个包含对 ASan 的支持的版本。关于各自的消毒器的作用以及如何详细配置它们的详细信息，各种编译器的文档非常有帮助。

通过传递各种编译器标志来启用消毒器，这些标志导致编译器向二进制文件添加额外的调试信息。当执行二进制文件时，消毒程序代码将执行检查，并将错误信息输出到 `stderr`。由于需要为消毒器执行代码，以查找潜在的错误，因此拥有较高的代码覆盖率会提高消毒器的可靠性。

要在 GCC 或 Clang 中启用 ASan，必须传递`-fsanitize=<sanitizer>` 编译器标志。对于 MSVC，对应的选项是`/fsanitize=<sanitizer>`。

编译器标志通过 `CMAKE_CXX_FLAGS` 缓存变量传入 CMake。因此，启用消毒器的情况下，从命令行调用 CMake 就像这样：

```
cmake -S <sourceDir> -B <BuildDir> -DCMAKE_CXX_FLAGS=-fsanitize=<sanitizer>
```

当使用 CMake 预设时，可以在那里定义包含编译器标志的缓存变量。全局设置消毒器选项也会影响在设置标志后使用 `FetchContent` 或 `ExternalProject` 的项目，所以要谨慎。对于 ASan，在 GCC 和 Clang 上使用`-fsanitize=address`，在 MSVC 上使用`/fsanitize=address`。MSan 使用`-fsanitize=memory`，LSan 使用`-fsanitize=leak`，TSan 使用`-fsanitize=thread`，UBSan 仅在编写本书时使用`-fsanitize=undefined` 来启用 GCC 和 Clang。要为 ASan、LSan 和 MSan 获得更简洁的输出，请

告诉编译器显式地保留帧指针。这是通过在 GCC 和 Clang 中设置-fno-omit-framepointer 来实现的。MSVC 只支持 x86 版本的/Oy-选项。

#### Note

不建议在 CMakeLists.txt 中设置 CMAKE\_CXX\_FLAGS 变量来启用消毒程序，因为消毒程序既不是构建的，也没有使用项目定义目标。此外，在 CMakeLists.txt 中设置 CMAKE\_CXX\_FLAGS 变量可能与用户可能从命令行传递的内容发生冲突。

消毒器是提高代码质量的一个非常强大的工具。与单元测试和覆盖率报告一起，提供了确保代码质量的四个主要方式中的三个。确保代码质量的第四个选项是使用静态代码分析器。

## 7.5. 静态代码分析

单元测试、消毒程序和覆盖率报告都依赖于实际运行的代码，以检测可能的错误。静态代码分析在不运行代码的情况下分析代码，所有编译的代码都可以进行分析，而不仅仅是测试覆盖的部分。当然，这也意味着可以发现各种各样的问题。静态代码分析的一个缺点是运行测试需要很长时间。

CMake 支持一些用于静态代码分析的工具，这些工具可以通过设置属性或全局变量来启用，并且需要安装和在系统路径中可以让 CMake 找到的外部程序。链接使用系统的链接器，因此不需要进一步安装。CMake 支持的工具如下所示：

- Clang-Tidy 是一个 C++ linter 工具，包含了大量错误、违反代码风格和接口滥用的检查。通过 <LANG>\_CLANG\_TIDY 属性或 CMAKE\_<LANG>\_CLANG\_TIDY 变量启用。
- Cppcheck 是另一个用于 C++ 的静态代码分析工具。通过 <LANG>\_CPPCHECK 属性或 CMAKE\_<LANG>\_CPPCHECK 变量启用。
- cpplint 是 C++ 代码风格检查器。通过 <LANG>\_CPPLINT 属性或 CMAKE\_<LANG>\_CPPLINT 变量启用。cpplint 最初是在谷歌开发的，所以内部硬编码了谷歌的 C++ 样式风格。
- include what you use (lwyu) 是一个 Python 程序，解析 C++ 源文件并确定所包含的文件中哪些是真正需要的。通过 <LANG>\_INCLUDE\_WHAT\_YOU\_USE 属性或 CMAKE\_<LANG>\_INCLUDE\_WHAT\_YOU\_USE 变量启用。
- link what you use (lwyu) 是 CMake 的内置特性，若可执行文件链接了未使用的库，会输出警告。通过 LINK\_WHAT\_YOU\_USE 属性或 CMAKE\_LINK\_WHAT\_YOU\_USE 变量启用。注意，这与所选的语言无关。

对于所有工具，<LANG> 不是 C 就是 CXX。属性是一个列表，其中包含各自的可执行参数和命令行参数。CMake 3.21 中，静态代码分析器只支持 Ninja 和 Makefile 生成器的自动执行。Visual Studio 在 IDE 的设置上处理静态代码分析器，这是 CMake 无法控制的。Lwyu 是一个特例，因为其为 LDD 或 ld 链接器使用了特殊的标志，所以 LINK\_WHAT\_YOU\_USE 属性只是一个布尔值，从而说明只有使用 ELF 平台才能支持 lwyu。

如覆盖报告和消毒器一样，静态代码分析工具是通过命令行，将各自变量中的命令传递给 CMake 或使用预设来启用的。若设置了该变量，则在编译源文件时将自动执行静态代码分析器。为构建启用 clang-tidy 如下所示：

```
cmake -S <sourceDir> -B <buildDir> -DCMAKE_CXX_CLANG_TIDY="clang-tidy;-checks=*;-header-filter=<sourceDir>/*"
```

命令和参数格式化为一个列表。上面的例子中，所有对 clangtidy 的检查都是使用`-checks=*`启用的，并且添加了一个过滤器，只对当前项目的包含文件应用 clangtidy，使用`-header-filter=<sourceDir/*>`。当使用 Cppcheck、Cpplint 和 iwyu 时，同样的模式也可以工作：

```
cmake -S <sourceDir> -B <buildDir> -DCMAKE_CXX_CPPCYHECK="cppcheck;--enable=warning;--inconclusive;--force;--inline-support"
cmake -S <sourceDir> -B <buildDir> -DCMAKE_CXX_CPPLINT="cpplint"
cmake -S <sourceDir> -B <buildDir> -CMAKE_CXX_INCLUDE_WHAT_YOU_USE="iwyu;-Xiwyu;any;-Xiwyu;iwyu;-Xiwyu;args;--verbose=5"
```

静态代码分析器将在编译项目中的文件时运行，发现的输出都将与正常的编译器警告或错误一起打印出来。默认情况下，分析器的所有非关键发现都不会导致构建失败。对于零容忍警告的高质量软件，可以将适当的标志传递给 Cppcheck 和 Clang-Tidy：

- 对于 Clang-Tidy，传递`--warnings-as-errors=*` 将会在编译时发现问题或警告时失败
- 对于 Cppcheck，若发现问题，传递`--error-exitcode=1` 参数将使 Cppcheck 以 1 退出，而不是 0，从而使构建失败。
- Iwyu 和 cpplint 缺少类似的标志。

Clang-Tidy 的一个非常好的特性是，可以自动将修复程序应用到源文件。这可以通过向 Clang-Tidy 传递`--fix` 和`--fix-error` 来实现。

### 增量构建时的注意事项

所有的静态代码分析器只有在文件实际编译的情况下才能工作。为了确保静态代码分析器捕获所有错误，必须在一个干净的构建中运行。

除了 iwyu 之外，所有静态代码分析器都会检查源文件以查找问题；另一方面，iwyu 将检查二进制文件以查找未使用的依赖项。

iwyu 分析器的目的是帮助加快构建并降低依赖关系树的复杂性，iwyu 的命令定义在 CMAKE\_LINK\_WHAT\_YOU\_USE\_CHECK 中。这个变量只是一个布尔选项，若设置了将各自的标志传递给链接器，以输出任何未使用的直接依赖项。CMake 3.21 版本中，这定义为 ldd -u -r，ldd 的使用说明该分析器只能用于 ELF 平台。iwyu 可以通过一个简单的选项来启用：

```
cmake -S <sourceDir> -B <buildDir> -DCMAKE_LINK_WHAT_YOU_USE=TRUE
```

lwyu 的输出可能是这样：

```
[100%] Linking CXX executable ch7_lwyu_example
Warning: Unused direct dependencies:
/lib/x86_64-linux-gnu/libssl.so.1.1
/lib/x86_64-linux-gnu/libcrypto.so.1.1
```

本例中，显示 libssl.so 连接了但没有使用。

将各种静态代码分析器与 iwyu 和 lwyu 结合在一起，有助于保持代码库体积最小。本章中，我们已经了解了如何定义测试、消毒和静态代码分析，它们处理检查代码是否正确运行。现在看到的问题是，若必须为所有单个目标启用各种组合，CMakeLists.txt 可能会变得混乱，特别是对于大型项目。一个干净的替代方案是提供自定义构建类型，以支持全局编译时的代码分析。

## 7.6. 创建自定义构建类型

我们讨论了 Debug、Release、RelWithDebInfo 和 MinSizeRel 等构建类型，这些都是由 CMake 默认提供的。这些构建类型可以通过将全局标志传递给所有目标的自定义构建类型进行扩展。对于依赖于某些编译器标志的代码质量工具，提供自定义构建类型可以简化 CMakeLists.txt，特别是对于大型项目。创建自定义构建类型也可以直接影响全局 CMAKE\_<LANG>\_FLAGS。

### 别覆盖 CMAKE\_<LANG>\_FLAGS

设置全局编译器选项优先于 CMakeLists.txt 中的 CMAKE\_<LANG>\_FLAGS。这些标志可在项目外部设置，可以通过命令行传递，也可以通过工具链文件提供。在项目内部修改很有可能会影响从外部设置的情况。

对于多配置生成器，如 MSVC 或 Ninja Multi-Config，可用的构建类型存储在 CMAKE\_CONFIGURATION\_TYPES 缓存变量中。对于单个配置生成器，如 Make 或 Ninja，当前构建类型存储在 CMAKE\_BUILD\_TYPE 变量中。应该在顶层项目上定义自定义构建类型。

Coverage 的自定义构建类型可以像这样添加到 CMakeLists.txt：

```
get_property(IS_MULTI_CONFIG GLOBAL PROPERTY GENERATOR_IS_MULTI)
if(IS_MULTI_CONFIG_GENERATOR)
  if(NOT "Coverage" IN_LIST CMAKE_CONFIGURATION_TYPES)
    list(APPEND CMAKE_CONFIGURATION_TYPES Coverage)
  endif()
else()
```

```

set(KNOWN_BUILD_TYPES Debug Release RelWithDebInfo Coverage)
set_property(CACHE CMAKE_BUILD_TYPE
    PROPERTY STRINGS ${KNOWN_BUILD_TYPES}
)
if(NOT CMAKE_BUILD_TYPE IN_LIST KNOWN_BUILD_TYPES)
    message(FATAL_ERROR "Unknown build type: ${CMAKE_BUILD_TYPE}")
endif()

```

看看例子中发生了什么:

- 首先, 确定当前生成器是多配置, 还是单配置, 存储在 GENERATOR\_IS\_MULTI\_CONFIG 全局属性中。由于该属性不能直接在 if 语句中使用, 因此该属性检索并存储在 IS\_MULTI\_CONFIG 变量中。
- 若当前生成器确实是多配置生成器, 则将名为 Coverage 的自定义构建配置添加到 CMAKE\_CONFIGURATION\_TYPES 中, 并使生成器可用, 但只有在不存在的情况下可用。
- 若生成器是单配置生成器, 则通过设置 CMAKE\_BUILD\_TYPE 缓存变量的 STRINGS 属性来添加 Coverage 构建存在的提示。这将在 CMake GUI 中创建一个带有有效选项的下拉菜单。为方便起见, 支持的构建类型存储在 KNOWN\_BUILD\_TYPES 变量中。
- 由于当前生成类型通常从外部提供给单配置生成器, 因此谨慎的做法是检查未知的生成类型, 若指定了未知的生成类型, 则中止配置。打印 FATAL\_ERROR 消息将使 CMake 停止构建。

这样, Coverage 构建类型就添加到 CMake 中了, 但构建类型还没有配置为向构建中添加自定义编译器和链接器标志。要定义标志, 需要使用两组缓存变量:

- CMAKE\_<LANG>\_FLAGS\_<CONFIGURATION>
- CMAKE\_<TARGET\_TYPE>\_LINKER\_FLAGS\_<CONFIGURATION>

<CONFIGURATION> 是自定义构建类型的名称, <LANG> 是编程语言, 而链接器标志的 <TARGET\_TYPE> 是可执行的或各种类型的库。将自定义构建的配置建立在现有构建类型的基础上, 以便重用配置选项。以下示例基于 Debug 构建类型的标志为 Clang 或 GCC 兼容编译器设置 Coverage 构建类型:

```

set(CMAKE_C_FLAGS_COVERAGE
    "${CMAKE_C_FLAGS_DEBUG} --coverage" CACHE STRING ""
)
set(CMAKE_CXX_FLAGS_COVERAGE
    "${CMAKE_CXX_FLAGS_DEBUG} --coverage" CACHE STRING ""
)
set(CMAKE_EXE_LINKER_FLAGS_COVERAGE
    "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --coverage" CACHE STRING ""
)
set(CMAKE_SHARED_LINKER_FLAGS_COVERAGE
    "${CMAKE_SHARED_LINKER_FLAGS_DEBUG} --coverage"
    CACHE STRING ""
)

```

```
)
```

这些标志还可以包含生成器表达式，以便在设置标志时考虑到不同的编译器。将标志标记为高级有助于防止用户对变量的意外更改：

```
mark_as_advanced(CMAKE_C_FLAGS_COVERAGE
                  CMAKE_CXX_FLAGS_COVERAGE
                  CMAKE_EXE_LINKER_FLAGS_COVERAGE
                  CMAKE_SHARED_LINKER_FLAGS_COVERAGE
                  CMAKE_STATIC_LINKER_FLAGS_COVERAGE
                  CMAKE_MODULE_LINKER_FLAGS_COVERAGE
)
```

有时库文件名应该反映它们是用特殊的构建类型创建的。为自定义构建类型设置 CMAKE\_<CONFIGURATION>\_POSTFIX 将实现此目的。这是 Debug 版本的常见方式，因此当打包在一起时，可以将文件与发布版本版本区分开来。与此相关的是 DEBUG\_CONFIGURATIONS 全局属性，包含认为是未优化的配置，并用于调试。若自定义构建视为非 Release 构建，则应考虑向属性中添加如下内容：

```
set_property(GLOBAL APPEND PROPERTY DEBUG_CONFIGURATIONS Coverage)
```

DEBUG\_CONFIGURATION 属性应该在调用 target\_link\_libraries 之前在顶层项目中设置。DEBUG\_CONFIGURATIONS 属性目前只有 target\_link\_libraries 使用，由于历史原因，这些库可以以 DEBUG 作为前缀，或者进行优化，以表明应该链接到各自的构建配置中。现在，这很少使用了，因为生成器表达式的控制粒度更细。

本章就到此结束。我们已经了解了测试和质量工具最常用的方式，并希望可以为您的卓越软件质量之旅做出贡献。

## 7.7. 总结

保持软件高质量是一项巨大而复杂的任务，如今有如此多的工具和技术来测试软件。通过本章描述的技术和工具，我们希望对现代 C++ 开发中使用的常见任务和工具有一个简要的概述。CTest 和 CMake 可以帮助编排各种测试，以最大限度地利用这些工具。本章中，了解了如何定义和运行测试，如何并行执行它们，以及如何管理测试资源。我们已经讨论了如何定义测试固件，以及如何定义高级方法来根据测试的输出确定测试是否成功。

演示了如何使用 Gcov 设置代码覆盖率报告，以及如何定义自定义构建类型来传递所需的编译器标志。了解了如何在 CMake 项目中包含各种用于静态代码分析的工具，以及如何使用各种编译器的消毒器。最后，简要介绍了如何使用 Catch2 框架定义和运行微基准测试。

下一章中，将看到如何使用外部程序来格式化代码并加速构建。

## 7.8. 练习题

回答以下问题来测试对本章的理解：

1. 如何在 CMake 中定义测试?
2. 如何让 CTest 执行特定的测试?
3. 如何重复执行不稳定的测试，直到它成功或失败?
4. 测试如何以并行和随机顺序运行?
5. 如何防止多个测试同时使用一个测试资源?
6. 如何为目标启用静态代码分析器?
7. 如何使用自定义构建类型?

# 第 8 章 执行自定义任务

构建和发布软件可能是一项复杂的任务，任何工具都无法完成所有不同的任务。有时，可能希望执行编译器或 CMake 功能没有涵盖的任务。常见的任务包括打包构建工件、创建哈希以验证下载，或者为构建生成或定制输入文件。还有许多其他特定的任务，这些任务依赖于内置某个软件的环境。

本章中，将学习如何在 CMake 项目中包含这样的自定义任务，以及如何创建自定义构建目标和自定义命令。将讨论如何创建和管理目标之间的依赖关系，以及如何从标准构建中包含或排除它们。

项目的构建步骤中包含这样的外部程序可以帮助确保代码保持一致，即使有许多人参与其中。由于 CMake 构建易自动化，因此使用 CMake 调用必要的命令，可以很容易地将这些工具应用到各种机器或 CI 环境中。

将学习如何定义自定义任务以及如何控制它们的执行时间，将重点关注管理自定义任务和常规目标之间的依赖关系。由于 CMake 经常用于跨多个平台提供构建信息，还将学习如何定义通用任务，使它们在能运行 CMake 的地方运行。

我们将讨论以下主题：

- 使用外部程序
- 构建时执行自定义任务
- 配置时执行自定义任务
- 复制和修改文件
- 执行平台无关的命令

## 8.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本

本章的所有示例和源代码都可以在本书的 GitHub 库中找到，<https://github.com/Packt Publishing/CMake-Best-Practices>。容易缺少相应软件，可将相应的示例将从构建中排除。

## 8.2. 使用外部程序

CMake 具有相当多的功能，因此在构建软件时可以执行许多任务。某些情况下，开发人员需要做一些 CMake 没有涉及到的事情。常见的例子包括：运行为目标文件做一些预处理或后处理的特殊工具，使用为编译器生成输入的源代码生成器，以及压缩和打包 CPack 无法处理的构件。构建步骤中必须完成的特殊任务的列表可能非常多。CMake 支持三种执行自定义任务的方式：

- 通过 `add_custom_target` 定义执行命令的目标

- 通过 `add_custom_command` 将自定义命令附加到现有的目标，或者通过使目标依赖于由自定义命令生成的文件
- 通过 `execute_process`，在配置步骤中执行命令

只要有可能，应该在构建步骤中调用外部程序，因为配置步骤对调用难以控制，应该尽可能快地运行。

让我们学习如何定义在构建时执行的任务。

### 8.3. 构建时执行自定义任务

添加自定义任务的通用方法是创建自定义目标，该目标以命令序列的形式执行外部任务。自定义目标的处理方式与其他库或可执行目标相同，不同的是不调用编译器和链接器，它们执行由用户定义的操作。可以使用 `add_custom_target` 定义自定义目标：

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

`add_custom_target` 的核心是通过 `COMMAND` 选项传递的命令列表。虽然第一个命令可以不带这个选项，但最好在 `add_custom_target` 中添加 `COMMAND` 选项。默认情况下，定制目标只在显式请求时执行，除非指定了 `ALL` 选项。自定义目标总认为是过时的，因此总是运行指定的命令，而不管是否会反复产生相同的结果。使用 `DEPENDS` 关键字，可以使定制目标依赖于使用 `add_custom_command` 或其他目标定义的定制命令的文件和输出。要使自定义目标依赖于另一个目标，可以使用 `add_dependencies`。若使用自定义目标创建文件，可以在 `BYPRODUCTS` 选项下列出这些文件。列出的文件都将使用 `GENERATED` 属性标记，CMake 使用该属性来确定构建是否过期，并找出需要清理的文件，但使用 `add_custom_command` 创建文件的任务可能更适合。

通常，命令在当前二进制目录中执行，该目录在 `CMAKE_CURRENT_BINARY_DIRECTORY` 缓存变量中。若需要修改，这可以通过 `WORKING_DIRECTORY` 选项来更改。该选项可以是绝对路径，也可以是相对路径（当前二进制目录的相对路径）。

`COMMENT` 选项用于指定在命令运行之前打印的消息，若命令以静默方式运行，那么可以使用这个选项。但并不是所有的生成器都显示这些消息，因此使用它来显示关键信息不太可靠。

`VERBATIM` 标志使所有命令直接传递到平台，而无需进一步转义或由底层 Shell 替换变量。CMake 本身仍然会替换传递给命令或参数的变量。当转义可能出现问题时，建议使用 `VERBATIM` 标志。编写自定义任务，使其独立于底层平台也是一种很好的方式。

可能的话，USES\_TERMINAL 选项指示 CMake 提供对终端的命令访问。若使用了 Ninja 生成器，则在终端作业池中运行。该池中的所有命令都是串行执行的。

使用 Ninja 生成文件时，可以使用 JOB\_POOL 选项来控制作业的并发性。它很少使用，也不能与 USES\_TERMINAL 标志一起使用。但开发者很少需要干预 Ninja 的工作池，并且处理它也不是一件小事。若想了解更多信息，可以在 CMake 的 JOB\_POOLS 属性的官方文档中找到更多信息。

SOURCES 属性接受与自定义目标相关联的源文件列表。该属性不影响源文件，但可以使文件在某些 IDE 中可见。若命令依赖于与项目一起交付的文件，如脚本，则应该在这里添加这些文件。

COMMAND\_EXPAND\_LISTS 选项将列表传递给命令前展开列表。有时这是必要的，因为在 CMake 中，列表只是由分号分隔的字符串，这可能会导致语法错误。当使用 COMMAND\_EXPAND\_LISTS 选项时，分号将替换为适当的空白字符，具体取决于平台。扩展包括使用 \$<JOIN:> 的生成器表达式生成的列表。

下面是一个自定义目标的例子，使用一个叫做 CreateHash 的外部程序为另一个目标的输出哈希值：

```
add_executable(SomeExe)
add_custom_target(CreateHash ALL COMMAND Somehasher
$<TARGET_FILE:SomeExe>)
```

这个例子创建了一个名为 CreateHash 的自定义目标，使用 SomeExe 目标的二进制文件作为参数调用外部 SomeHasher 程序。注意，二进制文件是使用 \$<TARGET\_FILE:SomeExe> 生成器表达式。这有两个目的：消除了用户跟踪目标二进制文件文件名的需要，并在两个目标之间添加了隐式依赖。CMake 将识别这些隐式依赖关系，并按正确的顺序执行目标。若生成所需文件的目标还没有构建，CMake 将自动构建。也可以使用 \$<TARGET\_FILE:> 生成器直接执行由另一个目标创建的可执行文件。以下生成器表达式会导致目标之间的隐式依赖：

- \$<TARGET\_FILE:target>：包含目标的主二进制文件的完整路径，例如.exe、.so 或.dll。
- \$<TARGET\_LINKER\_FILE: target>：包含用于链接到目标的文件的完整路径。这通常是库文件本身，在 Windows 上，它将是与 DLL 关联的.lib 文件。
- \$<TARGET SONAME FILE: target>：这包含库文件及其全名，包括 SOVERSION 属性设置的任何数字，例如.so.3。
- \$<TARGET\_PDB\_FILE: target>：这包含用于调试的生成的程序数据库文件的完整路径。创建自定义目标是在构建时执行外部任务的一种方法。另一种方法是定义自定义命令，可用于向现有目标（包括自定义目标）添加自定义任务。

### 8.3.1 向现有目标添加自定义任务

有时，在构建目标时可能需要执行外部任务。CMake 中可以使用 add\_custom\_command 来实现这一点，它有两个签名。一个用于将命令与现有目标挂钩，而另一个用于生成文件。向现有目标添加命令的签名如下所示：

```
add_custom_command(TARGET <target>
    PRE_BUILD | PRE_LINK | POST_BUILD
    COMMAND command1 [ARGS] [args1...]
```

```
[COMMAND command2 [ARGS] [args2...] ...]
[BYPRODUCTS [files...]]
[WORKING_DIRECTORY dir]
[COMMENT comment]
[VERBATIM] [USES_TERMINAL]
[COMMAND_EXPAND_LISTS])
```

大多数选项的工作原理类似于 `add_custom_target` 中的选项。TARGET 属性可以是当前目录中定义的目标，这是该命令的一个限制。可以在以下时段将命令连接到构建中：

- `PRE_BUILD`: 在 Visual Studio 中，此命令在执行其他构建步骤之前执行。当使用其他生成器时，会在 `PRE_LINK` 命令之前运行。
- `PRE_LINK`: 此命令将在编译源代码之后运行，在可执行文件或存档工具链接到静态库之前运行。
- `POS_BUILD`: 这将在执行所有其他构建规则后运行该命令。

执行自定义步骤最常见的方法是使用 `POST_BUILD`; 其他两个选项很少使用，要么是因为支持有限，要么是因为它们既不能影响链接，也不能影响构建。

向现有目标添加自定义命令相对简单。下面的代码添加了一个命令，在每次编译后生成并存储构建文件的哈希值：

```
add_executable(MyExecutable)

add_custom_command(TARGET MyExecutable
    POST_BUILD
    COMMAND hasher ${TARGET_FILE:ch8_custom_command_example}
        ${CMAKE_CURRENT_BINARY_DIR}/MyExecutable.sha256
    COMMENT "Creating hash for MyExecutable"
)
```

这个例子中，名为 `hasher` 的自定义可执行文件，用来生成 `MyExecutable` 目标输出文件的哈希值。

需要在构建之前，执行某些操作的原因是更改文件或生成信息。为此，第二个签名通常是更好的选择。

### 8.3.2 使用自定义任务生成文件

通常，希望自定义任务产生特定的输出文件。这可以通过定义自定义目标，并在目标之间设置必要的依赖项来实现，或者通过与构建步骤挂钩，但 `PRE_BUILD` 不可靠，因为只有 Visual Studio 生成器正确地支持它。因此，更好的方法是创建一个自定义命令，通过使用 `add_custom_command` 函数的第二个签名来创建文件：

```
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])
```

`add_custom_command` 的这个签名定义了一个生成 `OUTPUT` 中指定的文件的命令。该命令的大多数选项都非常类似于 `add_custom_target` 和将自定义任务挂钩到构建步骤的签名。`DEPENDS` 选项可用于手动指定文件或目标的依赖项。相比之下，自定义目标的 `DEPENDS` 选项只能指向文件。若构建或 CMake 更新了依赖项，则会再次运行自定义命令。`MAIN_DEPENDENCY` 选项与此密切相关，指定命令的主要输入文件，工作原理与 `DEPENDS` 选项类似，只不过它只接受一个文件。`MAIN_DEPENDENCY` 主要用来告诉 Visual Studio 在哪里添加自定义命令。

#### Note

若源文件出现在 `MAIN_DEPENDENCY` 中，那么自定义命令将取代所列文件的正常编译，这可能导致链接器报错。

其他两个与依赖相关的选项 `IMPLICIT_DEPENDS` 和 `DEPFILE` 很少使用，因为它们的支持仅限于 `Makefile` 生成器。`IMPLICIT_DEPENDS` 告诉 CMake 使用 C 或 C++ 扫描器来检测所列文件的任何编译时依赖项，并从中创建依赖项。另一个选项 `DEPFILE` 可以用来指向 `.d` 依赖文件，该文件是由 `Makefile` 项目生成的 `.d` 文件，最初起源于 `GNU make` 项目，用起来很强大也很复杂，不应该对大多数项目进行手动管理。下面的例子说明了如何使用自定义命令，在常规目标文件运行前生成一个源文件：

```
add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/main.cpp
    COMMAND sourceFileGenerator ${CMAKE_CURRENT_SOURCE_DIR}/message.txt
    ${CMAKE_CURRENT_BINARY_DIR}/main.cpp
    COMMENT "Creating main.cpp from message.txt"
    DEPENDS message.txt
    VERBATIM
)
add_executable(
    ch8_create_source_file_example
    ${CMAKE_CURRENT_BINARY_DIR}/main.cpp
```

```
)
```

首先，自定义命令将当前二进制指令中的 main.cpp 文件定义为 OUTPUT 文件。然后，定义生成该文件的命令——这里使用名为 sourceFileGenerator 的程序——将消息文件转换为.cpp 文件。DEPENDS 部分说明，每次 message.txt 文件更改时都应该重新运行此命令。

之后，为可执行文件创建目标。由于可执行文件引用自定义命令的 OUTPUT 部分中指定的 main.cpp 文件，CMake 将隐式地在命令和目标之间添加必要的依赖关系。因为适用于所有生成器，所以以这种方式使用自定义命令比使用 PRE\_BUILD 指令更加可靠和可移植。有时为了创建所需的输出，需要多个命令。若存在产生相同输出的前一个命令，则可以使用 APPEND 选项将命令链接起来。使用 APPEND 的自定义命令只需要 COMMAND 和 DEPENDS 选项，忽略其他选项。若两个命令产生相同的输出文件，CMake 将输出错误，除非指定了 APPEND。若命令是可选执行，这很有用。看看下面的例子：

```
add_custom_command(OUTPUT archive.tar.gz
COMMAND cmake -E tar czf ${CMAKE_CURRENT_BINARY_DIR}/archive.tar.gz
$<TARGET_FILE:MyTarget>
COMMENT "Creating Archive for MyTarget"
VERBATIM
)

add_custom_command(OUTPUT archive.tar.gz
COMMAND cmake -E tar czf ${CMAKE_CURRENT_BINARY_DIR}/archive.tar.gz
${CMAKE_CURRENT_SOURCE_DIR}/SomeFile.txt
APPEND
)
```

目标的输出文件 MyTarget 添加到 tar.gz 文件中，另一个文件添加到相同的文件中。注意，第一个命令自动依赖于 MyTarget，因为其使用命令创建相应的二进制文件，但不会由构建自动执行。第二个自定义命令列出与第一个命令相同的输出文件，但将压缩文件添加为第二个输出。通过指定 APPEND，在执行第一个命令时自动执行第二个命令。若缺少 APPEND 关键字，CMake 将输出类似于下面的错误：

```
CMake Error at CMakeLists.txt:30 (add_custom_command):
Attempt to add a custom rule to output
  /create_hash_example/build/hash_example.md5.rule
which already has a custom rule.
```

本例中的定制命令隐式地依赖于 MyTarget，但不会自动执行。要执行它们，建议创建一个依赖于输出文件的自定义目标：

```
add_custom_target(create_archive ALL DEPENDS
```

```
 ${CMAKE_CURRENT_BINARY_DIR}/archive.tar.gz  
 )
```

这里，创建了一个名为 `create_archive` 的自定义目标它作为 All 构建的一部分执行。因为依赖于自定义命令的输出，所以构建目标将调用自定义命令，依赖于 `MyTarget`。若压缩包不是新的，则构建 `create_archive` 也会触发 `MyTarget` 的构建。

`add_custom_command` 和 `add_custom_target` 自定义任务都在 CMake 的构建步骤中执行，可以在配置时添加任务。我们将在下一节中讨论这个问题。

## 8.4. 配置时执行自定义任务

要在配置时执行自定义任务，可以使用 `execute_process`: 生成在构建前需要的信息，或者需要更新文件以重新运行 CMake。另一个情况是在配置步骤中生成 `CMakeLists.txt` 文件或其他输入文件，也可以通过 `configure_file` 实现。

`execute_process` 的工作原理与 `add_custom_target` 和 `add_custom_command` 非常相似。然而，区别是 `execute_process` 可以在变量或文件中捕获到 `stdout` 和 `stderr` 的输出。`execute_process` 的签名如下：

```
execute_process (COMMAND <cmd1> [<arguments>]  
                  [COMMAND <cmd2> [<arguments>]]...  
                  [WORKING_DIRECTORY <directory>]  
                  [TIMEOUT <seconds>]  
                  [RESULT_VARIABLE <variable>]  
                  [RESULTS_VARIABLE <variable>]  
                  [OUTPUT_VARIABLE <variable>]  
                  [ERROR_VARIABLE <variable>]  
                  [INPUT_FILE <file>]  
                  [OUTPUT_FILE <file>]  
                  [ERROR_FILE <file>]  
                  [OUTPUT_QUIET]  
                  [ERROR_QUIET]  
                  [COMMAND_ECHO <where>]  
                  [OUTPUT_STRIP_TRAILING_WHITESPACE]  
                  [ERROR_STRIP_TRAILING_WHITESPACE]  
                  [ENCODING <name>]  
                  [ECHO_OUTPUT_VARIABLE]  
                  [ECHO_ERROR_VARIABLE]  
                  [COMMAND_ERROR_IS_FATAL <ANY|LAST>])
```

`execute_process` 接受在 `WORKING_DIRECTORY` 中执行的 `COMMAND` 列表。最后一个要执行的命令的返回代码可以存储在 `RESULT_VARIABLE` 定义的变量中，或可以将使用分号分隔的变

量列表传递给 RESULTS\_VARIABLE。若使用的是列表，则命令将按与已定义的变量相同的顺序存储命令的返回代码。若定义的变量比命令少，那么多余的返回代码都将忽略。若定义了 TIMEOUT，有子进程都无法结束时，结果变量将包含超时。自 CMake 版本 3.19 以来，COMMAND\_ERROR\_IS\_FATAL 选项可用，进程失败（或是最后一个）时中止执行。下面的例子中，若命令返回非零值，CMake 的配置步骤将失败并出现错误：

```
execute_process(  
    COMMAND SomeExecutable  
    COMMAND AnotherExecutable  
    COMMAND_ERROR_IS_FATAL_ANY  
)
```

stdout 或 stderr 的输出可以使用 OUTPUT\_VARIABLE 或 ERROR\_VARIABLE 捕获。一种替代方案是，通过使用 OUTPUT\_FILE 或 ERROR\_FILE 重定向到文件，也可以通过传递 OUTPUT QUIET 或 ERROR QUIET 忽略。在变量和文件中不可能捕获输出，这将导致两者中的一个为空。哪些保留，哪些丢弃取决于平台。若不重定向，OUTPUT\_\* 选项可以将指定输出发送给 CMake 进程。

若使用变量捕获输出，但仍然需要显示，可以添加 ECHO\_<STREAM>\_VARIABLE。CMake 还可以通过向 COMMAND\_ECHO 选项传递 STDOUT、STDERR 或 NONE 来输出命令。但若输出在文件中捕获，这将没有效果。若为 stdout 和 stderr 指定了相同的变量或文件，则将合并结果，并且可以通过 INPUT\_FILE 选项传递一个文件来控制第一个命令的输入流。

变量的输出可以使用 <STREAM>\_STRIP\_TRAILING\_WHITESPACE 选项的方式控制，该选项将消除输出末尾的空白。当将输出重定向到文件时，这个选项无效。Windows 上，ENCODING 选项可用于控制输出，并接受以下值：

- NONE: 不执行重新编码。这将保持 CMake 的内部编码，即 UTF-8。
- AUTO: 使用当前控制台的编码。若不可用，将使用 ANSI。
- ANSI: 使用 ANSI 进行编码。
- OEM: 使用平台定义的编码。
- UTF8 或 UTF-8: 强制使用 UTF-8 编码。

使用 execute\_process 的原因是收集构建所需的信息，然后将其传递给项目。假设希望通过传递宏定义来将 Git 修订编译为可执行文件。这种方法的缺点是，对于要执行的定制任务，必须调用 CMake，而不仅仅是构建系统。因此，使用带 OUTPUT 参数的使用 add\_custom\_command 可能是更现实的解决方案，但出于演示目的，这个示例应该足够了。以下是在配置时读取 Git 哈希值，并将其作为编译定义传递给目标的示例：

```
find_package(Git REQUIRED)  
execute_process(COMMAND ${GIT_EXECUTABLE} "rev-parse" "--short"  
    "HEAD"  
    OUTPUT_VARIABLE GIT_REVISION  
    OUTPUT_STRIP_TRAILING_WHITESPACE  
    COMMAND_ERROR_IS_FATAL ANY  
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})
```

```
add_executable(SomeExe src/main.cpp)
target_compile_definitions(SomeExe PRIVATE VERSION="\${GIT_REVISION}")
```

传递给 `execute_process` 的 `git` 命令将在当前执行 `CMakeLists.txt` 文件的目录中执行。生成的哈希存储在 `GIT_REVISION` 变量中，若命令因任何原因失败，配置过程将停止并出现错误。

通过宏定义将信息从 `execute_process` 传递到编译器，并不是最优的方式。更好的解决方案是，生成一个包含此信息的头文件。CMake 还有 `configure_file` 指令可以用于此目的，我们将在下一节中看到。

## 8.5. 复制和修改文件

构建软件时，常见的任务是在构建前将一些文件复制到特定位置或进行修改。在配置时执行自定义任务一节中，示例会检索 Git 提交，并将其作为宏定义传递给编译器。一种更好的方法是生成一个包含必要信息的头文件。虽然只是代码片段，并将其写入文件可能导致代码特定于平台，从而很危险。CMake 对此的解决方案是使用 `configure_file` 指令，可以将文件从一个位置复制到另一个位置，并在此过程中修改其内容，其签名如下：

```
configure_file(<input> <output>
    NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
    FILE_PERMISSIONS <permissions>...)
    [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
    [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF]])
```

`configure_file` 将 `<input>` 文件复制到 `<output>` 文件，将创建输出文件的路径，路径可以是相对路径或绝对路径。若使用相对路径，则将从当前源目录搜索输入文件，但输出文件的路径将相对于当前构建目录。若不能写入输出文件，命令失败，配置停止。通常，输出文件与目标文件具有相同的权限，若当前用户与输入文件所属的用户不同，所有权可能会发生变化。若添加了 `NO_SOURCE_PERMISSION`，则权限不会转移，输出文件将获得默认权限：`rw-r--r--`。也可以使用 `FILE_PERMISSIONS` 选项手动指定权限，该选项接受一个三位数的数字作为参数。`USE_SOURCE_PERMISSION` 已经是默认选项，这个选项只是为了更明确地进行说明。

除非传递 `COPYONLY`，否则 `configure_file` 也会在复制到输出路径时替换输入文件的部分内容，`configure_file` 将用同名变量的值替换所有引用为 `\${SOME_VARIABLE}` 或 `@SOME_VARIABLE@` 的变量。若在 `CMakeLists.txt` 中定义了一个变量，当使用 `configure_file` 时，相应的值会写入输出文件。若未指定变量，则输出文件将在相应的位置包含一个空字符串。若有一个 `hello.txt.in` 文件，包含以下信息：

```
Hello \${GUEST} from @GREETER@
```

`CMakeLists.txt` 文件中，`configure_file` 可用来配置 `hello.txt.in` 文件：

```
set(GUEST "World")
set(GREETER "The Universe")
```

```
configure_file(hello.txt.in hello.txt)
```

生成的 hello.txt 文件的内容为 Hello World from The Universe。若使用 @ONLY 选项，则只替换 @GREETER@，结果内容将是 Hello \${GUEST} from The Universe。转换 CMake 文件时，@ONLY 很有用，因为 CMake 文件可能包含不应该被括号括起来的变量。ESCAPE\_QUOTES 将用反斜杠转义目标文件中的引号。configure\_file 将转换换行符，以便目标文件与当前平台匹配。默认的行为可以通过 NEWLINE\_STYLE 来改变。UNIX 或 LF 将使用\n 作为换行符，而 DOS、WIN32 和 CRLF 将使用\r\n。同时设置 NEWLINE\_STYLE 和 COPYONLY 选项会导致错误，而设置 COPYONLY 不会影响换行符的样式。

回到示例，我们将写哈希值写入一个头文件作为输入：

```
#define CMAKE_BEST_PRACTICES_VERSION "@GIT_REVISION@"
The CMakeLists.txt could look something like this:
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-parse --short HEAD
    OUTPUT_VARIABLE GIT_REVISION
    OUTPUT_STRIP_TRAILING_WHITESPACE
    COMMAND_ERROR_IS_FATAL ANY
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})

configure_file(version.h.in
    ${CMAKE_CURRENT_SOURCE_DIR}/src/version.h @ONLY)
```

版本信息作为宏定义传递，Git 哈希信息首先使用 execute\_process 进行检索。之后，configure\_file 复制文件，@GIT\_REVISION@ 会替换为当前提交的短哈希值。

使用预处理器定义时，configure\_file 将替换所有以 #cmakedefine VAR… 形式出现的行。使用 #define VAR 或 /\* undef VAR \*/，这取决于 VAR 是否包含解释为 true 或 false 的值。

有一个名为 version.in.h 的文件，包含以下两行：

```
#cmakedefine GIT_VERSION_ENABLE
#cmakedefine GIT_VERSION "@GIT_REVISION@"
```

附带的 CMakeLists.txt 文件看起来像这样：

```
option(GIT_VERSION_ENABLE "Define revision in a header file"
    ON)
if(GIT_VERSION_ENABLE)
    execute_process(
        COMMAND ${GIT_EXECUTABLE} rev-parse --short HEAD
        OUTPUT_VARIABLE GIT_REVISION
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    )
endif()
```

```
configure_file(version.h.in  
${CMAKE_CURRENT_SOURCE_DIR}/src/version.h @ONLY)
```

配置结束后，若 `GIT_REVISION_ENABLE` 处于启用状态，结果文件将包含以下内容：

```
1 #define GIT_VERSION_ENABLE  
2 #define CMAKE_BEST_PRACTICES_VERSION "c030d83"
```

若关闭 `GIT_REVISION_ENABLE`，生成文件将包含以下内容：

```
1 /* #undef GIT_VERSION_ENABLE */  
2 /* #undef GIT_REVISION */
```

总之，`configure_file` 对于为构建准备输入非常有用。除了生成源文件外，还经常用于生成 CMake 文件，然后将这些文件包含在 `CMakeLists.txt` 文件中。这样做的优点是允许独立于平台的文件复制和修改，这在跨平台工作时很重要。因为 `configure_file` 和 `execute_process` 通常同时进行，所以要确保执行的命令也是平台相关的。下一节中，将了解如何使用 CMake 定义与平台无关的命令和脚本。

## 8.6. 执行平台无关的命令

CMake 成功的关键是它允许在多种平台上构建相同的软件，同样其也必须以一种不假定使用平台或编译器的方式编写。这很有挑战性，特别是在处理自定义任务时。`cmake` 命令行程序支持-E 标志，可以用于执行常见任务，如文件操作和创建哈希值。大多数 `cmake -E` 命令用于与文件相关操作，如创建、复制、重命名和删除文件，以及创建目录。支持文件系统链接的系统，CMake 还可以在文件之间创建符号软硬链接。此外，CMake 可以使用 `tar` 命令创建文件包，并使用 `cat` 命令连接文本文件，还可以用于为文件创建各种哈希值。

还有一些操作可以提供关于当前系统的信息。功能操作将打印出 CMake 的功能，例如：支持哪些生成器，以及 CMake 当前运行的版本。`environment` 命令将打印已设置的环境变量的列表。

通过不带参数的运行 `cmake -E`，可以获得对命令行选项的完整信息。CMake 的在线文档可参见<https://cmake.org/cmake/help/v3.21/manual/cmake.1.html#run-a-commandline-tool>。

### 平台无关的文件操作

当文件操作必须由自定义任务执行时，可以使用 `cmake -E`。

大多数情况下，可以用 `cmake -E`，有时需要进行更复杂的操作。为此，CMake 可以以脚本模式运行。

#### 8.6.1 CMake 文件作为脚本执行

创建跨平台脚本时，CMake 的脚本模式是非常强大的特性，因为允许创建完全与平台无关的脚本。通过调用 `cmake -P <script>.cmake`，则执行指定的 CMake 文件。脚本文件可能不包含定义构

建目标的命令。参数可以作为带有-D 标志的变量传递，但这必须在-P 选项之前完成。或者，参数可能只放在脚本名称之后，这样就可以使用 CMAKE\_ARGV[n] 检索它们，并且参数的数量存储在 CMAKE\_ARGC 变量中。下面的脚本生成一个文件的哈希码，并将其存储在另一个文件中：

```
cmake_minimum_required(VERSION 3.21)
if(CMAKE_ARGC LESS 5)
    message(FATAL_ERROR "Usage: cmake -P CreateSha256.cmake
        file_to_hash target_file")
endif()

set(FILE_TO_HASH ${CMAKE_ARGV3})
set(TARGET_FILE ${CMAKE_ARGV4})

# Read the source file and generate the hash for it
file(SHA256 "${FILE_TO_HASH}" GENERATED_HASH)

# write the hash to a new file
file(WRITE "${TARGET_FILE}" "${GENERATED_HASH}")
```

这个脚本的使用方式为 `cmake -P CreateSha256.cmake <input_file> <output_file>`。注意，前三个参数是 `cmake`、`-P` 和脚本的名称 (`CreateSha256.cmake`)。尽管不是严格要求的，脚本文件应该总是在开头包含 `cmake_minimum_required`。另一种不带位置参数定义脚本的方法如下所示：

```
cmake_minimum_required(VERSION 3.21)
if(NOT FILE_TO_HASH OR NOT TARGET_FILE)
    message(FATAL_ERROR "Usage: cmake -DFILE_TO_HASH=<input_file>
        -DTARGET_FILE=<target file> -P CreateSha256.cmake")
endif()

# Read the source file and generate the hash for it
file(SHA256 "${FILE_TO_HASH}" GENERATED_HASH)

# write the hash to a new file
file(WRITE "${TARGET_FILE}" "${GENERATED_HASH}")
```

必须使用显式传递的变量调用脚本，如下所示：

```
cmake -DFILE_TO_HASH=<input>
      -DTARGET_FILE=<target> -P CreateSha256.cmake
```

这两种方法也可以结合使用。一种常见的模式是期望所有简单的强制参数作为位置参数，可选

或更复杂的参数作为已定义变量。将脚本模式与 `add_custom_command`、`add_custom_target` 或 `execute_process` 相结合，可以创建独立于平台的构建指令。从前面生成哈希值的示例如下所示：

```
add_custom_target(Create_hash_target ALL
COMMAND cmake -P ${CMAKE_CURRENT_SOURCE_DIR}/cmake/CreateSha256.cmake
$<TARGET_FILE:SomeTarget>
${CMAKE_CURRENT_BINARY_DIR}/hash_example.sha256
)

add_custom_command(TARGET SomeTarget
POST_BUILD
COMMAND cmake -P ${CMAKE_CURRENT_SOURCE_DIR}/cmake/CreateSha256.cmake
$<TARGET_FILE:SomeTarget>
${CMAKE_CURRENT_BINARY_DIR}/hash_example.sha256
)
```

将 CMake 的脚本模式，与在项目的配置或构建阶段执行定制命令的方法相结合，可以在定义构建过程时提供很大的自由度，甚至可以针对不同的平台。然而，在构建过程中添加太多的逻辑可能会使其难于维护。当需要编写脚本或向 `CMakeLists.txt` 文件添加自定义命令时，最好是考虑一下，这一步是否属于构建过程，还是在用户设置开发环境时让他们设置。

## 8.7. 总结

本章中，学习了如何通过执行外部任务和程序自定义构建。介绍了如何将自定义构建操作添加为目标，如何将它们添加到现有目标，以及如何在配置步骤中执行。还讨论了命令如何生成文件，以及 CMake 如何使用 `configure_file` 复制和修改文件。最后，学习了如何使用 CMake 命令行实用程序以独立于平台的方式执行任务。

自定义 CMake 构建是一个非常强大的工具，但也使构建更加脆弱，因为在执行自定义任务时，构建的复杂性也会增加。尽管有时不可避免，但依赖于安装的编译器和链接器以外的程序，可能会让构建在一个没有安装这些程序或不可用的平台上失败。必须特别小心，以确保自定义任务不会假设系统的情况。最后，执行自定义任务可能会给构建系统带来性能损失。

但若小心使用自定义构建步骤，则是增加构建内聚性的好方法，因为许多与构建相关的任务都可以在构建定义所在的位置定义。这可以使任务自动化变得更加容易，如创建构建工件的哈希或在打包文件中包括所有文档。

下一章中，将学习如何使构建环境在不同系统之间可移植。了解如何使用预设来定义配置 CMake 项目的方法，以及如何将构建环境包装到容器中，还有如何使用 `sysroot` 来定义工具链和库，以便在系统之间移植。

## 8.8. 练习题

回答以下问题来测试对本章的理解:

1. `add_custom_command` 和 `execute_process` 的主要区别是什么?
2. `add_custom_command` 为什么有两个签名?
3. `add_custom_command` 的 `PRE_BUILD`, `PRE_LINK` 和 `POST_BUILD` 选项有什么不同?
4. 定义变量用 `configure_file` 替换的两种方法是什么?
5. 如何控制 `configure_file` 的替换行为?
6. CMake 命令行工具执行任务的两个标志是什么?

# 第9章 创建可复制的构建环境

构建软件可能很复杂，特别是涉及到依赖项或特殊工具的情况。在一台机器上编译的程序可能不能在另一台机器上运行，因为缺少一些关键的软件。依靠软件项目文档的正确性来找出所有构建需求，通常是不够的，因此开发者需要花大量的时间梳理各种错误消息，以找出构建失败的原因。

构建或持续集成(CI)环境中，人们避免升级任何东西，因为他们担心每一次更改都可能破坏构建软件的能力。因为担心无法再发布产品，这甚至导致公司拒绝升级正在使用的编译器工具链。创建关于构建环境的健壮和可移植的信息绝对是游戏规则的改变者。通过预置，CMake 提供了定义配置项目的通用方法的可能性。当与工具链文件、Docker 容器和 sysroot 结合使用时，创建可以在不同机器上重新创建的构建环境将变得容易得多。

本章中，将学习如何定义 CMake 预设来配置、构建和测试 CMake 项目，以及如何定义和使用工具链文件。我们将简要介绍使用容器构建软件，并学习如何使用 sysroot 工具链文件创建独立的构建环境。本章的主要主题如下：

- 使用 CMake 预设
- 使用容器进行构建
- 使用 sysroot 隔离构建环境

## 9.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本

对于使用构建容器的示例，需要使用 Docker。

本书的所有示例和源代码都可以在 GitHub 库中找到。本章中，CMake 预设和构建容器的示例都在库的根文件夹中。若缺少相应的软件，可以将相应的示例将从构建中排除。库可以在这里找到：<https://github.com/PacktPublishing/CMake-Best-Practices>。

## 9.2. 使用 CMake 预设

虽然多配置、编译器和平台上构建软件是 CMake 的优势，但这也是其最大的弱点，因为这通常使开发者很难确定哪些构建设置进行过测试，并且适用于给定的软件。3.19 版本起，CMake 有了预设的特性，可以以可靠和方便的方式处理这些场景。以前，开发人员必须依靠文档和约定来确定 CMake 项目的首选配置。预设可以指定构建目录、要使用的生成器、目标架构、工具链、缓存变量和要与项目一起使用的环境变量。从 CMake 3.20 开始，就有了影响构建和测试阶段的预设。

为了使用预设，项目的顶层目录必须包含名为 CMakePresets.json 或 CMakeUserPresets.json 的文件。若两个文件都存在，将先解析 CMakePresets.json，再解析 CMakeUserPresets.json。这两个文件有相同的格式，但使用方式略有不同：

- CMakePresets.json 由项目本身提供，并处理项目特定的事情，比如运行 CI 构建，或若项目本身需要哪些工具链，应该使用哪些工具链进行交叉编译。CMakePresets.json 用于特定的项目，不应该引用项目结构外的文件或路径。由于这些预设与项目紧密相连，通常也处于版本控制中。
- CMakeUserPresets.json 通常由开发人员定义，以便在自己的机器或构建环境中使用。CMakeUserPresets.json 可以尽可能的具体，包含项目之外的路径或特定系统设置特有的路径。因此，项目不应提供此文件，也不应将其置于版本控制中。

预设是将缓存变量、编译器标志等移出 CMakeLists.txt 文件的好方法，同时以一种可以在 CMake 中使用的方式保持可用的信息，从而提高项目的可移植性。若预置可用，通过命令 `cmake --listpresets` 可以从源目录中列出预设：

```
Available configure presets:

"ninja-debug" - Ninja (Debug)
"ninja-release" - Ninja (Release)
```

若设置了 `displayName` 属性，预设的名称以带引号的形式列出。在命令行选择预设时，名称需要带引号。

CMake GUI 将在源目录中显示所有可用的预设：

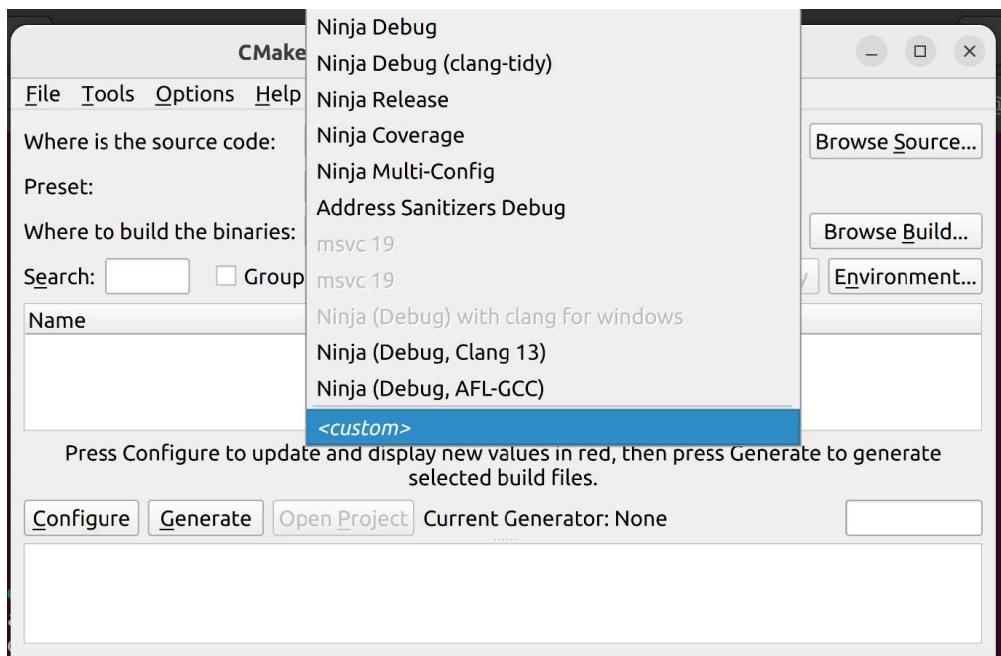


图 9.1 CMake GUI 中列出可用的预设

CMake 的 3.21 版本，`ccmake` 命令行配置工具不支持预设。需要通过以下命令使用预设，可以从顶层目录中选择配置预设：

```
cmake --preset=name
```

CMakePresets.json 和 CMakeUserPresets.json 的整体结构是这样的:

```
{
  "version": 3,
  "cmakeMinimumRequired": {"major": 3, "minor": 21, "patch": 0},
  "configurePresets": [...],
  "buildPresets": [...],
  "testPresets": [...],
  "vendor": {
    "microsoft.com/VisualStudioSettings/CMake/1.9": {
      "intelliSenseMode": "windows-msvc-x64"
    }
  }
}
```

version 字段指定要使用的 JSON 模式。版本 1 是 CMake 3.19 的第一个版本，只支持 configurePresets。版本 2 增加了 buildPresets 和 testPresets，CMake 3.20 开始支持；版本 3 增加了更多选项，CMake 3.21 开始支持。

可选的 cmakeMinimumRequired 字段可以用来定义构建此项目所需的 CMake 的最小版本。由于最低要求通常也在 CMakeLists.txt 文件中说明，这通常会省略。

这三个列表：configurePresets、buildPresets 和 testPresets，每个列表都包含了用于配置、构建和测试项目的配置。构建和测试的预置要求至少有一个配置预置，将在本节后面看到。

vendor 字段包含特定于供应商或 IDE 信息的可选映射。CMake 不解释该字段的内容，除非验证 JSON 格式。映射的键应该是由斜线分隔的特定于供应商。前面的示例中，供应商预置的键是 microsoft.com/VisualStudioSettings/CMake/1.9。供应商字段中的值可以是任何有效的 JSON 格式。

要使用预置，至少有一个配置预置，定义了 CMake 配置生成系统的环境。至少应该指定在配置时要使用的构建路径和生成器。配置预设还可为单配置生成器设置常用的缓存变量，如 CMAKE\_BUILD\_TYPE。包含配置预设的预设，在 Debug 模式下用 Ninja 生成器构建：

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "ninja",
      "displayName": "Ninja Debug",
      "description": "build in debug mode using Ninja generator",
      "generator": "Ninja",
      "binaryDir": "build",
    }
  ]
}
```

```
"cacheVariables": { "CMAKE_BUILD_TYPE": "Debug" }  
}  
]  
}
```

所有预设都必须在预设块内有唯一的名称。由于一些 GUI 应用程序只显示分配了 displayName 字段的预设值，因此强烈建议设置这个字段。

### 预设的命名约定

CMakePresets.json 中命名由项目定义的预设是一个很好的方式，以一种不与开发人员在 CMakeUserPresets.json 中定义名称冲突的方式。常见的约定是在项目定义的预设前加上 ci-，以标记其为 CI 环境所设置。

预置的版本 1 和 2 中，binaryDir 和 generator 字段是强制的；版本 3 中，变成了可选的。若没有设置任何字段，其行为与不设置预设的行为相同。CMake 的命令行选项将覆盖在预置中的相关值。因此，若设置了 binaryDir，则会在调用 cmake --preset= 时自动创建 (-B 选项传递的值将覆盖预设中的值)。

缓存变量既可以定义为“键: 值”对，如前面的例子所示，也可以定义为 JSON 对象，这允许指定变量类型。文件路径可以这样指定：

```
"cacheVariables": {  
  "CMAKE_TOOLCHAIN_FILE": {  
    "type": "FILEPATH",  
    "value": "${sourceDir}/cmake/toolchain.cmake"  
  }  
}
```

若使用 key:value 形式，该类型将视为 STRING，除非是 true 或 false(没有引号)，将解释为 BOOL。注意示例中的 \${sourceDir}，这个宏在使用预设时展开。

以下是一些常用宏：

- \${sourceDir}: 指向项目源目录，\${sourceParentDir} 指向源目录的父目录。不带源目录路径的目录名可以通过 \${sourceDirName} 获得。若 \${sourceDir} 是/home/sandy/MyProject，\${sourceDirName} 将是 MyProject，\${sourceParentDir} 将是/home/sandy/。
- \${generator}: 这包含由所使用的当前预设指定的生成器。对于构建和测试预设，包含配置预设的生成器。
- \${hostSystemName}: 主机操作系统的系统名称，与 CMAKE\_HOST\_SYSTEM 相同。该值可以是 uname -s 或 Linux、Windows 或 Darwin(用于 macOS) 的结果。
- \${env{<variable-name>}}: 包含名为<variable-name>的环境变量。若在预置中使用环境字段定义变量，则使用此值。使用 \${env{<variable-name>}} 的工作原理类似，即使它定义了，该值也取自父环境，这允许在现有的环境变量中增加值。因为其不允许循环引用，所以不能使用 \${env{...}}。注意，在 Windows 环境中，变量是不区分大小写的；在预设中使用的变量区分大小写。因

此，建议与环境变量的 Shell 环境保持一致。

- `$vendor{<macro-name>}`: 这是 IDE 供应商插入自己宏的扩展点。由于 CMake 不能解释这些宏，使用 `$vendor{…}` 宏的预设将会忽略。
- `${dollar}`: 这是字面美元符号 \$ 的占位符。

修改预置环境的工作原理类似于设置缓存变量：通过设置环境字段，该字段包含“键：值”对的映射。即使值为空，也始终设置环境变量。环境变量可以相互引用，只要不包含循环引用：

```
{  
  "version": 3,  
  "configurePresets": [  
    {  
      "name": "ci-ninja",  
      "generator": "Ninja",  
      "binaryDir": "build",  
      "environment": {  
        "PATH": "${sourceDir}/scripts:$env{PATH}",  
        "LOCAL_PATH": "$env{PATH}",  
        "EMPTY" : null  
      }  
    }  
  ]  
}
```

本例中，PATH 环境变量通过从项目结构中来添加路径，使用 `$env{PATH}` 宏可以确保该值来自预设值外，然后 LOCAL\_PATH 变量通过使用 `$env{PATH}` 宏引用修改过的 PATH 环境变量。只要 PATH 环境变量不包含会创建循环引用的 `$env{LOCAL_PATH}`，这个引用就没问题。通过传递 null 来取消设置 EMPTY 环境变量。注意，null 没有加引号。除非使用构建预设或测试预设，否则环境不会转发至相应的步骤。若使用了构建或测试预设，但不应用来自配置预设的环境，则可以在将 `inheritConfigureEnvironment` 字段设置为 false，显式地进行声明。

### 9.2.1 继承预设

预置可以从与 `inherited` 字段的相同类型的其他预置继承，该字段可以包含单个预置或预置列表。从父字段继承字段时，可以覆盖预置或添加其他字段。这有助于避免通用构建块的代码重复。结合隐藏字段，这可以减少 `CMakePreset.json` 文件的大小。考虑下面的例子：

```
{  
  "version": 3,  
  "configurePresets": [  
    {  
      "name": "ci-ninja",  
      "generator": "Ninja",  
      "environment": {  
        "FOO": "bar"  
      },  
      "inherited": "parent-preset"  
    }  
  ]  
}
```

```

    "hidden": true,
    "binaryDir": "build"
},
{
  "name": "ci-ninja-debug",
  "inherits": "ci-ninja",
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Debug"
  }
},
{
  "name": "ci-ninja-release",
  "inherits": "ci-ninja",
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Release"
  }
}
]
}

```

本例中，`ci-ninja-debug` 和 `ci-ninja-release` 预置都继承自隐藏的 `ci-ninja` 构建预置，并将 `CMAKE_BUILD_TYPE` 缓存变量设置为各自的配置。仍然可以使用隐藏的预设，但在调用 `cmake --list-presets` 时不会显示。CMakeUserPreset.json 中定义的预设可能继承自 CMakePreset.json。

前面的例子中，预设从一个父级继承，但也可以从多个父级继承。下面的例子展示了 CMakeUserPreset.json 如何与前面例子中的 CMakePreset.json 一起工作：

```

{
  "version": 3,
  "configurePresets": [
    {
      "name": "gcc-11",
      "hidden": true,
      "binaryDir": "build",
      "cacheVariables": {
        "CMAKE_C_COMPILER": "gcc-11",
        "CMAKE_CXX_COMPILER": "g++-11"
      }
    },
    {
      "name": "ninja-debug-gcc",
      "inherits": ["ci-ninja-debug", "gcc-11"]
    }
  ]
}

```

```
},  
]  
}
```

用户提供了一个预设，它显式选择 GCC 11 作为名为 `gcc-11` 的编译器。之后，`ninja-debug-gcc` 预设从 `CMakePreset.json` 中定义的 `ninja-debug` 预设继承值，并将其与用户提供的 `gcc-11` 预置相结合。若两个父预设为同一字段定义不同的值，则继承列表中第一个出现的值优先。

## 9.2.2 预设的条件

有时预设只在特定的条件下才有意义，例如：对于特定的构建平台。使用 Visual Studio 生成器的配置预设只在 Windows 环境中有用，若条件选项不满足条件，则可以禁用预设。继承父预设中定义的条件。条件可以是常量、字符串比较或检查列表是否包含值，可以从预设的版本 3 中获得。以下配置预设只会在 Windows 上启用：

```
{  
  "name": "ci-msvc-19",  
  "generator": "Visual Studio 16 2019",  
  "binaryDir": "build",  
  "condition": {  
    "type": "equals",  
    "lhs": "${hostSystemName}",  
    "rhs": "Windows"  
  }  
}
```

前面的示例中，若使用 `${hostSystemName}` 宏检索主机系统的名称，然后将其与 Windows 字符串进行比较，则构建预设是启用的。若 `${hostSystemName}` 匹配，则启用预设，否则禁用预设，尝试使用它将导致错误。比较字符串时，大小写很重要：对于不区分大小写的测试，可以使用接受正则表达式的 `matches` 或 `notMatches` 类型。

对于更复杂的条件，支持使用 `allOf`、`anyOf` 和 `not` 操作符的布尔逻辑嵌套。若配置预置只在 Windows 和 Linux 中启用，而在 macOS 中不启用，则预置和条件可以如下所示：

```
{  
  "name": "WindowsAndLinuxOnly",  
  "condition": {  
    "type": "anyOf",  
    "conditions": [  
      {  
        "type": "equals",  
        "lhs": "${hostSystemName}",  
        "rhs": "Windows"  
      },  
      {  
        "type": "equals",  
        "lhs": "${hostSystemName}",  
        "rhs": "Linux"  
      }  
    ]  
  }  
}
```

```
        {
          "type": "equals",
          "lhs": "${hostSystemName}",
          "rhs": "Linux"
        }
      ]
}
```

每个条件还可以包含更多的嵌套条件，尽管这样做会增加预设的复杂性。

目前，我们只在示例中看到了配置预置，还有构建预置和测试预置。构建和测试预置的语法非常类似于配置预置和许多字段，如 name、displayName 和 inherit，条件的工作原理与配置预置相同。

生成预设必须在 configure 预设字段中指定一个配置预设，或者从指定配置预设的另一个生成预设继承。构建目录由配置预设确定，并且继承来自配置预设的环境，除非将 inheritConfigureEnvironment 字段设置为 false。构建预设可以指定要构建的目标列表：

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "ci-msvc-19",
      "displayName": "msvc 19",
      "description": "Configuring for msvc 19",
      "generator": "Visual Studio 16 2019",
      "binaryDir": "build"
    }
  ],
  "buildPresets": [
    {
      "name": "ci-msvc-debug",
      "configurePreset": "ci-msvc-19",
      "configuration": "Debug"
    },
    {
      "name": "ci-msvc-release",
      "configurePreset": "ci-msvc-19",
      "configuration": "Release"
    },
    {
      "name": "ci-documentation",
      "configurePreset": "ci-msvc-19",
      "targets": [
        "docs"
      ]
    }
  ]
}
```

```
        "api-doc",
        "doc"
    ]
}
]
}
```

示例中，定义了三个构建预置。其中两个用于指定 Visual Studio 的构建配置，第三个列出了作为文档构建的一部分的 api-doc 和文档目标。调用 ci-msvc 构建预设将构建“all”目标，而 ci-documentation 将只构建列出的目标。可用的构建预设列表可以用 cmake --build --list-presets 来检索。

测试预设的工作原理与构建预设相似，不同的是它们与 CTest 一起使用。类似地，在项目根目录上调用 ctest --list-presets 将列出可用的测试预置。测试预设可用于选择或排除某些测试、指定固件选项或控制测试的输出，测试预设的例子：

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "ci-ninja",
      ...
    }
  ],
  "testPresets": [
    {
      "name": "ci-feature-X",
      "configurePreset": "ci-ninja",
      "filter": {
        "include": {
          "name": "feature-X"
        },
        "exclude": {
          "label": "integration"
        }
      }
    }
  ]
}
```

前面的示例中，添加了一个测试预置，过滤包含 feature-X 测试，排除标记为 integration 的测试。这相当于在构建目录下使用以下命令：

```
ctest --tests-regex feature-X --label-exclude integration
```

自引入目标以来，CMake 预设可以说是少数几个改变 CMake 使用方式的特性。它们是一个折衷方案，可以在交付公共配置和与项目一起构建选项的同时，仍然保持 CMakeLists.txt 文件与平台无关。然而，有时提供必要的设置是不够的，还会希望共享一个构建环境，在该环境中可以确定软件可以编译。一种选择是定义一个包含 CMake 和必要库的容器。

## 9.3. 使用容器进行构建

容器化带来的好处是开发人员可以在一定程度上控制构建环境。容器化构建环境对于设置 CI 环境也有很大帮助。容器的运行时软件有很多，其中 Docker 最受欢迎。容器化的深入讨论超出了本书的范围，所以我们使用 Docker 运行本书的示例。

构建容器包含完整的构建系统，包括 CMake 和构建特定软件所需的工具和库。通过提供容器定义 (Dockerfile) 和项目一起提供，或者通过公开可访问的容器注册表提供，任何人都可以使用容器来构建软件。好处是，除了运行容器所需的软件外，开发人员不需要安装其他库或工具。缺点是构建可能需要更长的时间，并且不是所有 IDE 和工具都支持容器。Visual Studio Code 对在容器支持的很好，可参考<https://code.visualstudio.com/docs/remote/containers> 了解更多详情。

使用构建容器的工作流程如下：

1. 定义容器并构建。
2. 将源代码的副本装入构建容器中。
3. 运行用于在容器内构建的命令。

构建简单 C++ 应用程序的 Dockerfile 如下所示：

```
FROM alpine:3.15
RUN apk add --no-cache cmake ninja g++ bash make git
RUN <any command to install additional libraries etc.>
```

这将定义一个基于 Alpine Linux 3.15 的容器，并安装 cmake、ninja、bash、g++、make 和 git。实际容器可能有工具和库安装在里面以方便工作；然而，为了说明如何使用容器构建软件，拥有这样一个小容器就足够了。下面的 Docker 命令构建容器映像，并用 builder\_minimal 标记：

```
docker build . -t builder_minimal
```

当容器是本地的副本，那么源就挂载在容器中，所有的 CMake 命令都可以在容器中执行。假设用户从源目录执行 Docker 命令，配置 CMake 构建项目的命令可能如下所示：

```
docker run --user 1000:1000 --rm -v $(pwd):/workspace  
builder_minimal cmake -S /workspace -B /workspace/build  
docker run --user 1000:1000 --rm -v $(pwd):/workspace  
builder_minimal cmake --build /workspace/build
```

这将启动创建容器并在其中执行 CMake 命令。使用-v 将本地目录作为/workspace 挂载到容器中。因为 Docker 容器通常使用 root 作为默认用户，所以使用的用户 ID 和组是通过--user 选项传递的。在类 Unix 操作系统上，这应该与主机的用户 ID 相匹配，因此创建的文件也可以在容器外部编辑。--rm 标志告诉 Docker 在容器退出后删除该容器。

使用容器的另一种方法是通过向 docker run 命令传递-ti 标志以交互模式运行：

```
docker run --user 1000:1000 --rm -ti -v $(pwd):/workspace  
builder_minimal
```

这将在容器内启动 shell，可以直接使用构建命令，而不需要每次都重新启动容器。

关于编辑器或 IDE 和构建容器如何协同工作，有几种策略。若 IDE 本身支持，或者像 Visual Studio Code 那样通过扩展支持最好了。否则，在容器中装入合适的编辑器，并在容器中执行也是方便开发软件的一种策略。另一种方法是在主机系统上进行编辑并重新配置 Docker，不直接调用 CMake，而是启动容器时直接执行 CMake。

这里展示的是使用容器作为构建环境的最低要求，随着越来越多的 IDE 支持使用容器构建环境，使用容器将变得容易得多。容器使构建环境在不同机器之间可移植，并且可以帮助确保项目的所有开发人员使用相同的构建环境。将容器定义文件置于版本控制之下也是一个好主意，以便对构建环境的必要更改与代码一起跟踪。

容器是创建隔离构建环境的一种可移植的方法，但若由于某种原因这没办法实现，那么另一种创建可移植构建环境的方法是使用 sysroot。

## 9.4. 使用 sysroot 隔离构建环境

简单地说，系统根目录（或简称 sysroot）是构建系统认为的根目录，可以从中查找头文件和库。这个目录包含用于编译软件的平台的根文件系统的精简版本，经常用于为其他平台交叉编译软件时。若构建容器不是无法完成，sysroot 可以作为提供已定义构建环境的替代方案。

要在 CMake 中使用 sysroot，需要一个工具链文件，这些文件定义了用来编译和链接软件的工具，以及在哪里可以找到库。正常的构建中，CMake 通过自动检测系统检测工具链。工具链文件通过 CMAKE\_TOOLCHAIN\_FILE 传递给 CMake：

```
cmake -S <source_dir> -B <binary_dir> -DCMAKE_TOOLCHAIN_FILE=<path/to/toolchain.cmake>
```

从 3.21 版本起，CMake 支持`--toolchain` 选项来传递工具链文件，这相当于使用 `CMAKE_TOOLCHAIN_FILE` 缓存变量。

或者，可以使用 CMake 预设作为缓存变量来传递工具链文件。至少，与 `sysroot` 一起使用的工具链文件将定义 `CMAKE_SYSROOT` 变量，以指向 `sysroot`，并定义 `CMAKE_<LANG>_COMPILER` 变量，以指向与 `sysroot` 中库兼容的编译器。为了避免将来自 `sysroot` 外部的依赖项与安装在主机系统上的文件混淆在一起，通常还设置了用于控制 `find` 指令查找位置的变量。最小的工具链文件可能是这样的：

```
set(CMAKE_SYSTEM_NAME Linux)

set(CMAKE_SYSROOT /path/to/sysroot/)
set(CMAKE_STAGING_PREFIX path/to/staging/directory)

set(CMAKE_C_COMPILER /path/to/sysroot/usr/bin/gcc-10)
set(CMAKE_CXX_COMPILER /path/to/sysroot/usr/bin/g++-10)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE BOTH)
```

1. 使用 `CMAKE_SYSTEM_NAME` 变量来设置目标系统的系统名称，在 `sysroot` 中为其编译文件的系统。
2. 通过设置 `CMAKE_SYSROOT` 变量来设置 `sysroot` 的路径。`CMAKE_STAGING_PREFIX` 可选，用于指定在主机上安装项目构件的位置。指定暂存前缀有助于保持 `sysroot` 和主机文件系统的整洁；否则，构件的安装都将发生在主机文件系统上。
3. 设置 `CMAKE_C_COMPILER` 和 `CMAKE_CXX_COMPILER` 变量，将编译器设置为 `sysroot` 附带的编译器。
4. 设置 CMake 中 `find` 指令的搜索行为。`CMAKE_FIND_ROOT_PATH_MODE_*` 变量可以为 `ONLY`、`NEVER` 和 `BOTH`。若设置为 `ONLY`，CMake 将只搜索 `sysroot` 中的文件类型；若设置为 `NEVER`，则搜索将只考虑主机文件结构。若设置为 `BOTH`，则将搜索主机系统路径和 `sysroot` 路径。注意，`CMAKE_STAGING_PREFIX` 认为是系统路径，因此为了搜索 `sysroot` 和暂存目录，必须同时选中两者。本例中，配置的方式是将所有头文件和库限制在 `sysroot` 中，而 `find_program` 将只查找主机系统，而 `find_package` 将在两个文件系统内查找。

设置 `CMAKE_SYSROOT` 不会自动设置安装构建构件的位置。对于生成的二进制文件与主机系统兼容的情况，这可能是预期的行为。但比如交叉编译，这就不是我们想要的

结果，所以建议设置 `CMAKE_STAGING_PREFIX`。设置暂存目录有两个效果：第一，暂存目录中进行安装；第二，暂存目录将添加到 `find_` 指令的搜索前缀中。需要注意的是，暂存目录将添加到 `CMAKE_SYSTEM_PREFIX_PATH` 中，这有一个问题，即来自前面示例的 `CMAKE_FIND_ROOT_PATH_MODE_XXX` 变量必须设置为 `BOTH`，这样才能找到安装在暂存区域中的包、库和程序。

#### CMAKE\_STAGING\_PREFIX 和 CMAKE\_INSTALL\_PREFIX

若同时设置了 `CMAKE_STAGING_PREFIX` 和 `CMAKE_INSTALL_PREFIX`，则暂存前缀优先。

根据经验，只要工具链与主机系统兼容，就可以省略暂存，否则需要进行定义。

与容器相比，`sysroot` 的一个缺点是不能启动，也不能在其中执行命令。因此，若工具链和 `sysroot` 与主机平台不兼容，产生的文件在不移动到目标平台或使用模拟器的情况下，无法执行。

## 9.5. 总结

`CMake` 的主要优势是多功能性，可以使用各种工具链为大量平台构建软件。这样的缺点是，开发人员有时很难找到软件的工作配置。但是通过提供 `CMake` 预设、容器和 `sysroot`，可使 `CMake` 项目会变得更容易使用。

本章中，我们研究了如何定义 `CMake` 预设来定义工作配置设置，以及如何创建构建和测试定义。然后，简要介绍了如何创建 `Docker` 容器，以及如何在其中调用 `CMake` 命令。然后以简要介绍 `sysroot` 和工具链文件结束本章。关于工具链和 `sysroot` 的更多内容将在第 12 章中介绍。

下一章中，将学习如何使用大型分布式项目作为超级构建，将了解如何处理不同的版本，以及如何以可管理的方式使用多个库组装项目。

## 9.6. 练习题

回答以下问题来测试对本章的理解：

1. `CMakePresets.json` 和 `CMakeUserPresets.json` 有什么不同？
2. 如何在命令行上使用预设？
3. 预设存在哪三种类型？它们如何相互依赖？
4. 预设配置应该定义的最小值是多少？
5. 当从多个预置继承时，若一个值多次指定，哪个优先？
6. 使用容器构建的常规策略是什么？
7. 与 `sysroot` 使用的工具链文件通常需要定义什么？

# 第 10 章 处理大项目和分布式存储库

每个大项目都有自己的一组依赖关系，处理这些依赖关系的最简单方法是使用包管理器，例如 Conan 或 vcpkg。但是由于公司策略、项目需求或缺乏资源，使用包管理器并不总是可行。因此，项目作者可能会参考传统的、老式的方法来处理依赖性。处理这些依赖关系的通常方法可能包括将所有的依赖关系嵌入到存储库的构建代码中。或者，项目作者可能决定让最终用户从头处理依赖项。这两种方法都不优雅，有自己的缺点。

所以呢？

可以使用超级构建。

超级构建是一种方法，可用于从项目代码中分离满足依赖关系所需的逻辑，类似于包管理器的工作方式，我们可以称这个方法为穷人版的包管理器。将依赖逻辑从项目代码中分离出来，可以使我们拥有更灵活和可维护的项目结构。本章中，将详细了解如何实现。

我们将讨论以下主题：

- 超级构建的要求和需求
- 跨多个代码库的构建
- 超级构建中的版本一致性

## 10.1. 相关准备

深入学习本章之前，应该对第 5 章的内容有所了解。本章将遵循实例教学的方法，建议从<https://github.com/PacktPublishing/CMake-Best-Practices/tree/main/chapter10>获取本章的示例内容。所有的示例都假设您将使用<https://github.com/PacktPublishing/CMake-Best-Practices>提供的容器运行。

通过检查先决条件和需求开始学习超级构建。

## 10.2. 超级构建的要求和需求

超级构建可以构建为构建多个项目的大型构建，也可以构建为处理依赖关系的项目内子模块，CMake 已经有稳定的方法获取存储库。比如，ExternalProject 和 FetchContent 是处理外部依赖的 CMake 模块。我们的例子中，将使用 FetchContent 模块。使用 CMake 提供的方法不是严格的要求，超级构建也可以通过使用版本控制系统工具来构建，比如 git 子模块或 git 子树。因为 CMake 是本书的重点，而且 git 对 FetchContent 支持的很好，所以我们更喜欢使用它。

让我们继续了解如何构建跨越多个代码库的项目。

## 10.3. 跨多个代码库的构建

软件项目都会直接或间接地使用多个代码存储库。处理本地项目代码最简单，但软件项目很少是独立的。若没有适当的依赖管理策略，事情可能会变得复杂。本章的第一个建议是若可以的话使用包管理器，包管理器大大减少了花费在依赖项管理上的工作。若不能使用包管理器，可能需要使用自己的迷你项目专用包管理器，这称为超级构建。

超级构建通常用于实现项目自给自足的依赖关系，所以项目能够在不需要用户干预的情况下满足依赖关系，拥有这样的能力对所有使用来说都非常方便。

### 10.3.1 推荐的方法-FetchContent

我们将使用第 10 章 01 例进行演示，先从 CMakeLists.txt 文件开始。为了简单起见，省略了前七行：

```
if(CH10_EX01_USE_SUPERBUILD)
    include(superbuild.cmake)
else()
    find_package(GTest 1.10.0 REQUIRED)
    find_package(benchmark 1.6.1 REQUIRED)
endif()

add_executable(ch10_ex01_tests)
target_sources(ch10_ex01_tests PRIVATE src/tests.cpp)
target_link_libraries(ch10_ex01_tests PRIVATE GTest::Main)

add_executable(ch10_ex01_benchmarks)
target_sources(ch10_ex01_benchmarks PRIVATE src/benchmarks.cpp)
target_link_libraries(ch10_ex01_benchmarks PRIVATE benchmark::benchmark)
```

这是一个简单的 CMakeLists.txt 文件，定义了两个目标 ch10\_ex01\_tests 和 ch10\_ex01\_benchmark。这些目标分别依赖于 Google Test 和 Google Benchmark 库。这些库可以通过超级构建或 find\_package(...) 调用找到和定义，这取决于 CH10\_EX01\_USE\_SUPERBUILD 变量。我们一直使用 find\_package(...) 确定路径，来看一下超级构建文件 superbuild.cmake：

```
include(FetchContent)
FetchContent_Declare(benchmark
    GIT_REPOSITORY https://github.com/google/benchmark.git
    GIT_TAG v1.6.1
)
FetchContent_Declare(GTest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG release-1.10.0
)

FetchContent_MakeAvailable(GTest benchmark)
add_library(GTest::Main ALIAS gtest_main)
```

第一行中，包含了 FetchContent 模块，使用它来获取依赖。接下来的六行中，FetchContent\_Declare 用于声明两个外部目标，benchmark 和 GTest，并通过 Git 获取。

因此，调用 `FetchContent_MakeAvailable(...)` 使目标可用。最后，`add_library(...)` 用于为 `gtest_main` 目标定义一个名为 `GTest::Main` 的别名目标。这样做是为了保持 `find_package(...)` 和超级构建目标名称间的兼容性。因为 `find_package(...)` 和超级构建目标名称已经兼容，所以没有为 `benchmark` 定义别名目标。

通过调用以下命令来配置和构建这个示例：

```
cd chapter10/ex01_external_deps
cmake -S ./ -B build -DCH10_EX01_USE_SUPERBUILD:BOOL=ON
cmake --build build/ --parallel $(nproc)
```

在前两行中，进入 `example_10/` 文件夹并配置项目。注意，这里 `CH10_EX01_USE_SUPERBUILD` 为 `ON`，以便启用超级构建代码。最后一行中，用 `N` 个并行作业构建项目，其中 `N` 是 `nproc` 的结果。

由于有了 `find_package(...)` 的路径，若环境中的 `google test ≥ 1.10.0` 和 `google benchmark ≥ 1.6.1`，那么构建也可以在不启用超级构建的情况下正常工作。这允许包维护人员在不打补丁的情况下更改依赖项版本。像这样的小型定制点对于可移植性和可重复性非常重要。

接下来，再来看一个使用 `ExternalProject` 模块的超级构建示例。

### 10.3.2 传统的方式-`ExternalProject_Add`

在 `FetchContent` 出现之前，大多数人会使用 `ExternalProject_Add` 来实现超级构建方法。该功能由 `ExternalProject` 模块提供。本节中，将看到一个使用 `ExternalProject_Add` 的超级构建示例，以了解它与 `FetchContent` 模块的区别。

来一起看看第 10 章中的 `CMakeLists.txt` 文件，示例 02(省略了注释和项目指令)：

```
# ...
include(superbuild.cmake)
add_executable(ch10_ex02_tests)
target_sources(ch10_ex02_tests PRIVATE src/tests.cpp)
target_link_libraries(ch10_ex02_tests PRIVATE catch2)
```

同样，该项目是一个包含单 C++ 源文件的单元测试项目，但这次使用的是 `Catch2`。`CMakeLists.txt` 文件包括 `superbuild.cmake` 文件，定义一个可执行目标，并将 `Catch2` 库链接到目标。这个示例没有使用 `FindPackage(...)` 来查找 `Catch2` 库。与 `FetchContent` 不同，`ExternalProject` 在构建时获取并构建外部依赖项。由于 `Catch2` 库的内容在配置时不可用，因此我们无法在这里使用 `FindPackage(...)`。`FindPackage(...)` 在配置时运行，并要求提供包文件。来看下 `superbuild.cmake`：

```
include(ExternalProject)
ExternalProject_Add(catch2_download
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v2.13.9
```

```

INSTALL_COMMAND ""

# For disabling the warning that treated as an error
CMAKE_ARGS -DCMAKE_CXX_FLAGS="-Wno-error=pragmas"
)

SET(CATCH2_INCLUDE_DIR ${CMAKE_CURRENT_BINARY_DIR}
/catch2_download-prefix/src/catch2_download/single_include)
file(MAKE_DIRECTORY ${CATCH2_INCLUDE_DIR})
add_library(catch2 IMPORTED INTERFACE GLOBAL)
add_dependencies(catch2 catch2_download)
set_target_properties(catch2 PROPERTIES
"INTERFACE_INCLUDE_DIRECTORIES" "${CATCH2_INCLUDE_DIR}")

```

superbuild.cmake 模块包括 ExternalProject 模块，使用 ExternalProject\_Add，用 GIT\_REPOSITORY、GIT\_TAG、INSTALL\_COMMAND 和 CMAKE\_ARGS 参数声明一个名为 catch2\_download 的目标。ExternalProject\_Add 可以从不同的源获取依赖项，示例试图通过使用 Git 获取依赖项。GIT\_REPOSITORY 和 GIT\_TAG 参数分别用于指定目标 git 库 URL 和在 git 克隆后要检出的标记。因为 Catch2 是一个 CMake 项目，需要提供给 ExternalProject\_Add 的参数量最小。ExternalProject\_Add 默认知道如何配置、构建和安装 CMake 项目，因此不需要 CONFIGURE\_COMMAND 或 BUILD\_COMMAND 参数。空的 INSTALL\_COMMAND 参数用于在构建之后禁用并安装依赖项。最后一个参数，CMAKE\_ARGS，用于将 CMAKE 参数传递给外部项目的配置步骤。我们使用它来静默关于 Catch2 编译中的语法警告（作为错误处理）。

ExternalProject\_Add 将所需的库获取到前缀路径中并构建，要使用获取的内容，必须首先将其导入到项目中。因为不能使用 FindPackage(...) 进行导入，所以需要手动做一些工作，其中之一是定义 Catch2 目标的 include 目录。因为 Catch2 是一个纯头文件库，所以用头文件定义一个接口目标就够了。这里声明了 CATCH2\_INCLUDE\_DIR 变量来设置将包含 CATCH2 头文件的目录，使用这个变量来设置本例中创建的导入目标的 INTERFACE\_INCLUDE\_DIRECTORIES 属性。接下来，使用 file(MAKE\_DIRECTORY \${CATCH2\_INCLUDE\_DIR}) 指令来创建 include 目录。由于 ExternalProject\_Add 的工作方式，在执行构建步骤之前不会出现 Catch2 内容。设置一个目标的 INTERFACE\_INCLUDE\_DIRECTORIES 需要给定的目录存在，所以使用了一个小 hack 作为解决方案。最后三行中，为 Catch2 声明一个导入的 INTERFACE 库，使这个库依赖于 catch2\_download 目标，并设置导入库的 INTERFACE\_INCLUDE\_DIRECTORIES。

试着配置和构建来检查以上设置是否有效：

```

cd chapter10/ex02_external_deps_with_extproject
cmake -S ./ -B build
cmake --build build/ --parallel $(nproc)

```

若一切正常，会看到类似这样的输出：

```
[ 10%] Creating directories for 'catch2_download'
[ 20%] Performing download step (git clone) for
'catch2_download'
Cloning into 'catch2_download'...
HEAD is now at 62fd6605 v2.13.9
[ 30%] Performing update step for 'catch2_download'
[ 40%] No patch step for 'catch2_download'
[ 50%] Performing configure step for 'catch2_download'
/* ... */
[ 60%] Performing build step for 'catch2_download'
/* ... */
[ 70%] No install step for 'catch2_download'
[ 80%] Completed 'catch2_download'
[ 80%] Built target catch2_download
/* ... */
[100%] Built target ch10_ex02_tests
```

好了，似乎已经成功构建了测试可执行文件。通过运行`./build/ch10_ex02_tests`可执行文件来检查它是否有效：

```
=====
All tests passed (4 assertions in 1 test case)
```

接下来，了解如何在超级构建中使用 QT 框架，并创建简单的 QT 应用程序。

### 10.3.3 福利 - 超级构建中使用 Qt 6 框架

我们处理的都是体积相当小的库。来试试一些更复杂的东西，比如在超级构建中使用一个大框架，比如 Qt 框架。对于这一部分，我们将使用第 10 章，示例 03 的例子。

#### 重要的 Note

若打算在提供的 Docker 容器之外尝试这个示例，可能必须安装 Qt 运行时所需的一些附加依赖项。类 Debian 系统所需的包如下：`libgl1-mesa-dev libglu1-mesa-dev '^libxcb.*-dev' libx11-xcb-dev libglu1-mesa-dev libxrender-dev libxi-dev libxkbcommon-dev libxkbcommon-x11-dev`。

该示例包含一个源文件 `main.cpp`，输出一个简单的 Qt 窗口应用程序和一条消息：

```
1 #include <qapplication.h>
2 #include <QPushButton.h>
```

```

3 int main( int argc, char **argv )
4 {
5     QApplication a( argc, argv );
6     QPushButton hello( "Hello from CMake Best Practices!",
7         0 );
8     hello.resize( 250, 30 );
9     hello.show();
10    return a.exec();
11 }

```

我们的目标是能够编译这个 Qt 应用程序，而不需要用户自己安装 Qt 框架。超级构建应该会自动安装 Qt 6，应用程序应该能够使用该框架。先看一下这个例子的 CMakeLists.txt:

```

if(CH10_EX03_USE_SUPERBUILD)
    include(superbuild.cmake)
else()
    set(CMAKE_AUTOMOC ON)
    set(CMAKE_AUTORCC ON)
    set(CMAKE_AUTOUIC ON)

    find_package(Qt6 COMPONENTS Core Widgets REQUIRED)
endif()

add_executable(ch10_ex03_simple_qt_app main.cpp)
target_compile_features(ch10_ex03_simple_qt_app PRIVATE cxx_std_11)
target_link_libraries(ch10_ex03_simple_qt_app Qt6::Core Qt6::Widgets)

```

与第一个示例类似，CMakeLists.txt 文件包括 superbuild.cmake 文件，取决于选项标志。若用户选择为该示例使用超级构建，那么将包括超级构建模块。否则，依赖将尝试使用 `find_package(...)` 在系统中定位。最后三行中，定义了一个可执行目标，设置了目标的 C++ 标准，并将定义的目标链接到 Qt6::Core 和 Qt6::Widgets 目标。这些目标要么由超级构建定义，要么由 `find_package(...)` 定义，这取决于用户是否选择使用超级构建。继续来看 superbuild.cmake:

```

include(FetchContent)
message(STATUS "Chapter 10, example 03 superbuild enabled.
    Will try to satisfy dependencies for the example.")
# Enable message output for FetchContent commands
set(FETCHCONTENT_QUIET FALSE)
set(QT_BUILD_SUBMODULES "qtbase" CACHE STRING "Submodules to build")
set(QT_WILL_BUILD_TOOLS on)
set(QT_FEATURE_sql off)
set(QT_FEATURE_network off)
set(QT_FEATURE_dbus off)
set(QT_FEATURE_opengl off)

```

```

set(QT_FEATURE_testlib off)
set(QT_BUILD_STANDALONE_TESTS off)
set(QT_BUILD_EXAMPLES off)
set(QT_BUILD_TESTS off)

FetchContent_Declare(qt6
    GIT_REPOSITORY https://github.com/qt/qt5.git
    GIT_TAG v6.3.0
    GIT_SHALLOW TRUE
    GIT_PROGRESS TRUE # Since the clone process is lengthy,
                      show progress of download
    GIT_SUBMODULES qtbase # The only QT submodule we need
)
FetchContent_MakeAvailable(qt6)

```

`superbuild.cmake` 文件使用 `FetchContent` 模块来获取 Qt 依赖。由于 Qt 的获取和准备过程可能很长，一些未使用的 Qt 框架特性会禁用。为了更好地跟踪进度，启用了 `FetchContent` 消息输出。让我们试着通过运行以下命令来配置和编译这个示例：

```

cd chapter10/ex03_simple_qt_app/
cmake -S ./ -B build -DCH10_EX03_USE_SUPERBUILD:BOOL=ON
cmake --build build/ --parallel $(nproc)

```

若一切顺利，应该会看到类似的输出：

```
/*...*/
[ 11%] Creating directories for 'qt6-populate'
[ 22%] Performing download step (git clone) for
'qt6-populate'
Cloning into 'qt6-src'...
/*...*/
[100%] Completed 'qt6-populate'
[100%] Built target qt6-populate
/*...*/
-- Configuring done
-- Generating done
/*...*/
[ 0%] Generating ../../mkspecs/modules
```

```
/qt_lib_widgets_private.pri
[ 0%] Generating ../../mkspecs/modules
/qt_lib_gui_private.pri
[ 0%] Generating ../../mkspecs/modules/qt_lib_core_private.pri
/* ... */
[ 98%] Linking CXX executable ch10_ex03_simple_qt_app
[ 98%] Built target ch10_ex03_simple_qt_app
/*...*/
```

一切顺利！让我们用以下命令运行生成的可执行文件来检查它是否工作：

```
./build/ch10_ex03_simple_qt_app
```

一切正常的话，应该会弹出一个小 GUI 窗口：

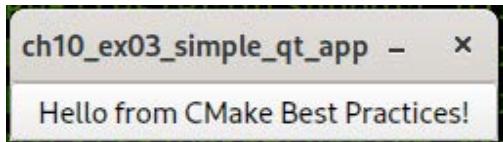


图 10.1 简单的 Qt 应用程序窗口

我们已经完成了学习如何在 CMake 项目中使用超级构建。接下来，来看看如何在超级构建中保持版本一致性。

## 10.4. 超级构建中的版本一致性

版本一致性对于软件项目很重要，在软件世界中没有什么是一成不变的，软件随着时间的推移而发展和变化。这样的更改通常需要提前确认，要么通过对新版本运行一系列测试，要么通过对调用代码本身进行更改。理想情况下，上游代码中的更改不应该对复制现有构建产生影响，直到我们希望它们这样做。如果针对这个组合进行了软件验证和测试，则应该始终使用 `z.q` 依赖版本构建项目的 `x.y` 版本。这样做的原因是，即使没有 API 或 ABI 更改，上游依赖项中的最小更改也可能影响软件的行为。若不保证版本一致性，软件将没有定义良好的行为。因此，找到提供版本一致性的方法很重要。

确保超级构建中的版本一致性取决于超级构建的组织方式。对于从版本控制系统获取的库，这相对容易。克隆项目，但不要原样使用它，而是签出到特定的分支或标签。若没有这样的锚点，则检出到特定的提交，这将为超级构建提供未来的保障。但这可能还不够，标签可能会修改，分支可能会强行推入，历史可能会改写。为了减少这种风险，可能更倾向于 Fork 项目，并将作为上游库。这样，就可以完全控制上游库的内容，但这种方法带来增加了维护的负担。

这个故事告诉我们，不要盲目地逆流而上，随时关注最近的变化。对于作为存档文件使用的第三方依赖项，始终检查它们的哈希摘要。通过这种方式，将确保在为项目使用期望的版本；若这有变化，则需要手动确认。

## 10.5. 总结

本章中，简要介绍了超级构建的概念，以及如何使用超级构建进行依赖管理。在缺乏包管理器的情况下，超级构建是一种非侵入性，且功能强大的依赖性管理方法。

下一章中，我们将学习在软件中进行模糊测试的知识，以及如何使用 CMake 自动化模糊测试。

## 10.6. 练习题

回答以下问题来测试对本章的理解：

1. 什么是超级构建？
2. 哪些场景中可以使用超级构建？
3. 如何在超级构建中保持版本一致？

# 第 11 章 自动化模糊测试

软件测试对于确保软件系统的行为符合预期至关重要，对基于场景的测试策略的需求仍然很高，但只有基于场景的方法还不够。实际中，所使用的东西并不像测试环境中那样单纯。许多不同的变量影响软件的行为：环境、用户、硬件、操作系统或第三方库可能会意外地影响软件。随着软件使用的越来越广泛，找到一种可以涵盖软件所有可能方面的测试套件变得非常困难。谢天谢地，模糊测试来了！

本章中，我们将学习模糊测试的基础知识，并学习如何使用该技术来测试软件，使其更可靠。此外，将学习如何将模糊测试工具集成到 CMake 中，以轻松定义模糊目标。

我们将讨论以下主题：

- 简介模糊测试
- 集成 AFL/libfuzzer

## 11.1. 相关准备

深入学习本章之前，建议阅读第 7 章。本章将遵循实例教学的方法，建议从<https://github.com/PacktPublishing/CMake-Best-Practices/tree/main/chapter11>获取本章的示例内容。所有的示例都假设您将使用<https://github.com/PacktPublishing/CMake-Best-Practices>提供的容器运行。

注意，libFuzzer 是 LLVM 工具链的一部分，需要安装 LLVM 并将其作为工具链使用。打包的开发环境预装了 llvm-13。AFL++ 是一个第三方工具，可以从<https://github.com/AFLplusplus/AFLplusplus>获得，了解有关 AFL++ 的例子。安装指南可在<https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md>获得。一些 Linux 发行版在他们的存储库中有预构建的 AFL++ 版本，可以通过包管理器进行安装。

先从学习一些关于模糊测试的基础知识开始。

## 11.2. 简介模糊测试

进一步讨论之前，先来了解一下模糊测试。模糊测试 (Fuzzing) 是一种向软件系统提供随机、意外数据的测试方法，以观察系统在特定输入下的行为，模糊测试会报告遇到的意外行为。这使我们能够发现其他测试策略和代码评审遗漏的 Bug。发现输入是否会导致安全问题或故障会比较困难，但模糊测试非常有效。当正确使用模糊测试时，可以轻松发现绝大多数关键的安全 Bug，如远程代码执行或特权升级。因此，了解模糊测试很重要。

模糊测试既可以手工完成，也可以在软件的帮助下自动完成。第二种方法更有效，可以利用机器算力进行模糊测试。基于语料库的、覆盖引导的模糊工具，是许多软件项目必备的工具。幸运的是，我们有相当不错的工具可以模糊测试 C 和 C++ 项目。模糊处理可以分为两大类：引导模糊测试和黑盒模糊测试。黑盒模糊测试是一种暴力的测试方法，依赖待测系统 (SUT) 对测试输入的反应，而向引导模糊测试由覆盖率分析或用户自身自动引导。引导模糊测试可以认为是白盒测试或灰盒测试，因为引导依赖于来自实现的反馈。引导模糊测试是发现软件未知边界的好方法。

为了使模糊测试过程有效，生成的输入不能是随机的。输入必须有效，能够通过模糊测试目标的初始基本检查，这样模糊测试器才能深入研究系统。因此，用户可能需要通过提供一组为 SUT 提供最大覆盖率的输入来启动模糊测试，这一组输入称为语料库。模糊测试器改变初始语料库数据，生成与原始输入相似但触发不同行为的输入变化。触发意外行为或覆盖之前未发现路径的输入数据，可以通过模糊测试工具保存到语料库中，以扩展语料库数据以供以后使用。模糊测试可能不需要初始语料库数据，这样模糊测试就需要从头开始生成输入数据。本章将采用基于语料库的引导性方式。

深入讨论这个主题之前，让我们先了解一个问题——模糊化不能替代其他测试策略或类型，比如单元测试或系统测试。模糊测试的目的是发现 Bug 和与预期不符的行为，所以应该使用模糊测试增强现有的测试策略。注意，模糊测试要求系统或单元是可测试的。因此，若还没有合适的测试，建议首先了解传统的测试策略。

好了，我们已经学了很多关于模糊的知识。接下来，将学习如何在 C 和 C++ 项目中使用两个著名的模糊测试软件 AFL++ 和 libFuzzer。

## 11.3. 集成 AFL/libfuzzer

这一节中，将了解可能使用过的两个著名的模糊测试工具，即 libFuzzer 和 AFL++，从 libFuzzer 开始。

### 11.3.1 libFuzzer

libFuzzer 是 LLVM 项目中的模糊测试库。若项目可以使用 LLVM 工具链进行编译，那么 libFuzzer 是默认的模糊测试器，使用 libFuzzer 需要额外的编译器/链接器标志和定义一个函数。通过一个示例来深入了解更多的细节。

为了在实践中进行模糊测试，先从 chapter11/ex01\_libfuzzer\_static\_lib 开始。例子中，假设有一个易受攻击的静态库目标，名为 message\_printer。这是一个简单的库，只有一个接口函数 print()，接受 const char\* 和 length 参数。假设这个库存在的漏洞，是因为没有意识到的边界情况而触发的。函数实现如下所示：

```
1 void message_printer::print(const char *msg, std::uint32_t len) {
2     constexpr char a [] = "Hello from CMake Best
3     Practices!";
4     if(len < sizeof(a)) {
5         return;
6     }
7     if(std::memcmp(a, msg, len) == 0) {
8         volatile char* ptr=nullptr;
9         // attempt to write an invalid memory location
10        // it is undefined behavior
11        *ptr = 'a';
12    }
13    /* ... */
14 }
```

print() 解引空指针会导致未定义行为，当 msg 输入恰好包含在 a 变量中，表示函数中未发现漏洞的字符串时。例子中的漏洞很容易发现，但现实中的漏洞和未定义行为要比这隐藏的深得多。

目标在手了，现在需要一个小型驱动程序，将 libFuzzer 生成的模糊测试数据提供给 message\_printer::print()，从而执行模糊测试。为此，必须将驱动器程序与 message\_printer 库，以及 libFuzzer 链接起来。先看看 CMakeLists.txt 的驱动程序示例：

```
add_executable(ch11_ex01_libfuzzer_fuzz)
target_sources(ch11_ex01_libfuzzer_fuzz PRIVATE fuzz_library.cpp)
target_compile_features(ch11_ex01_libfuzzer_fuzz PRIVATE cxx_std_11)
target_link_libraries(
    ch11_ex01_libfuzzer_fuzz PRIVATE ch11_ex01_libfuzzer_static)
target_compile_options(
    ch11_ex01_libfuzzer_fuzz PRIVATE -fsanitize=fuzzer)
target_link_libraries(
    ch11_ex01_libfuzzer_fuzz PRIVATE -fsanitize=fuzzer)
```

我们定义了一个名为 ch11\_ex01\_libfuzzer\_fuzz 的可执行目标，它链接到模糊测试目标 ch11\_ex01\_libfuzzer\_static。这里，需要将-fsanitize=fuzzer 参数同时传递给编译器和链接器，为 LLVM 工具链启用 libFuzzer。但 libFuzzer 是特定于 llvm 的，目前还不能用于 GCC。以上就是模糊测试驱动程序 CMake 端的全部了。

模糊测试驱动程序的实现非常简单。为了从 libFuzzer 得到生成的模糊测试数据，需要实现 libFuzzer 能够识别的函数，这个函数称为模糊测试入口。函数有一个特定的声明：

```
1 extern "C" int LLVMFuzzerTestOneInput(const std::uint8_t
2 *data, std::size_t size) {
3     /* fuzzing entry-point impl. */
4 }
```

libFuzzer 以字节跨度的形式生成模糊测试数据，并用生成的字节跨度调用 LLVMFuzzerTestOneInput。字节范围从 &data[0] 到 &data[size-1]，跨度应在函数体中输入模糊目标。这个函数在驱动程序的应用如 chapter11/ex01\_libfuzzer\_static\_lib/fuzz/fuzz\_library.cpp 所示：

```
1 #include <cstdint>
2 #include <library/library.hpp>
3 extern "C" int LLVMFuzzerTestOneInput(
4     const std::uint8_t *data, std::size_t size) {
5     chapter11::ex01::message_printer printer;
6     printer.print(reinterpret_cast<const char*>(data), size);
7     return 0;
8 }
```

模糊测试函数的实现很简单。我们构建了一个 message\_printer 的实例，并使用 libFuzzer 生成的数据和大小调用了 print()，其余的由 libFuzzer 自动处理。细心的读者可能已经注意到，示例应用程序没有 main() 函数。libFuzzer 库提供了一个 main() 函数，引导模糊测试框架最终调用模糊测试的入口点。

现在来看看驱动程序应用的实际情况，用以下命令配置和构建示例：

```
cd chapter11/
cmake --preset="ninja-debug-clang-13" -S ./ -B build/
cmake --build build/
```

因为 libFuzzer 示例需要 clang 编译器，所以显式地指定了 ninja-debug-clang-13 预设。构建好这个例子之后，可以执行下面的命令，来运行模糊测试的驱动程序：

```
cd build/ex01_libfuzzer_static_lib/fuzz/
./ch11_ex01_libfuzzer_fuzz
After running for a second, the driver application should
output something like this:
./build/ex01_libfuzzer_static_lib/fuzz/ch11_ex01_libfuzzer_fuzz
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1815993229
INFO: Loaded 1 modules (1 inline 8-bit counters): 1
[0x485fa0, 0x485fa1),
INFO: Loaded 1 PC tables (1 PCs): 1 [0x4723d8,0x4723e8),
INFO: -max_len is not provided; libFuzzer will not generate
inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty
corpus
#2 INITED cov: 1 ft: 1 corp: 1/1b exec/s: 0 rss: 26Mb
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==228856==ERROR: UndefinedBehaviorSanitizer: SEGV on
unknown address 0x000000000000 (pc 0x000000464194 bp
0x7ffdad9733f0 sp 0x7ffdad9733a0 T228856)
==228856==The signal is caused by a WRITE memory access.
==228856==Hint: address points to the zero page.
#0 0x464194 in chapter11::ex01::message_printer::
print(char const*, unsigned int) /home/mustafa
/workspace/personal/CMake-Best-Practices/chapter11
/ex01_libfuzzer_static_lib/src/library.cpp:26:14
#1 0x4640ea in LLVMFuzzerTestOneInput /home/mustafa
/workspace/personal/CMake-Best-Practices/chapter11
/ex01_libfuzzer_static_lib/fuzz/fuzz_library.cpp:7:11
/* ... */
```

```
==228856==ABORTING
MS: 4 CopyPart-CMP-CrossOver-CMP- DE: "Hello from CMake
Best Practices!\x004\x97\xad\xfd\x7f\x00\x00\x00dH\x00"-"
"Hello from CMake Best Practices!\x004\x97\xad\xfd\
\x7f\x00\x00\x00dH\x00.\x00"-; base unit: adc83b19e793491b
1c6ea0fd8b46cd9f32e592fc
0x48,0x65,0x6c,0x6c,0x6f,0x20,0x66,0x72,0x6f,0x6d,0x20,0x43
,0x4d,0x61,0x6b,0x65,0x20,0x42,0x65,0x73,0x74,0x20,0x50,0x7
2,0x61,0x63,0x74,0x69,0x63,0x65,0x73,0x21,0x0,0x34,0x97,0xa
d,0xfd,0x7f,0x0,0x0,0x64,0x48,0x0,0x2e,0x0,
Hello from CMake Best Practices!\x004\x97\xad\xfd\x7f\x00
\x00\x00dH\x00.\x00
artifact_prefix='.'; Test unit written to ./crash-e95d3
e40c04fcfa86897df986d66f7345bb0d4b1
Base64: SGVsbG8gZnJvbSBDTWFrZSBCZXN0IFByYWN0a
WNlcycEANJet/X8AAABkSAAuAA==
```

即使之前没有提供语料库，libFuzzer 也能够立即触发未定义行为。这样做的原因是因为 LLVM 的 libFuzzer 在目标函数上可以得到更高的覆盖率。想要从“CMake Best Practices”中找到“Hello !”，使用随机生成的字符串是不可能做到的。这类似于破解 32 位密码，用目前的硬件几乎不可能做到。使用这种方法，模糊测试器不可能找到隐藏的特别深的问题。因此，-fsanitize=fuzzer 标志的 LLVM 在这里所做的是将比较数据分割为更易于管理的多个比较数据。对于 32 个字符长的字符串，LLVM 会进行 {N} 次小比较。结果比较类似于如下方式：

```
1 if (*msg[0] == a[0]) {
2     if (*msg[1] == a[1]) {
3         if (*msg[2] == a[2]) {
4             if (*msg[3] == a[3]) {
5                 /*...*/
6             }
7         }
8     }
9 }
```

以这种方式进行比较可以让 libFuzzer 通过代码覆盖率数据快速找到每个字符。但这仍然类似于破解 32 位密码，只是这次密码验证系统会反馈是否正确猜测了每个数字，这种反馈将发现时间从无限缩短到瞬间。由于这种行为，我将 libFuzzer 称为灰盒模糊测试库。事实上，该技术是通过对汇编代码进行剪裁，使模糊测试更加高效，但我不认为这会带给模糊测试偏见。出于这个原因，我也不认为这种方法是完全的白盒策略，但这显然不是一个黑箱策略。因此，灰盒这个术语很适合用来描述这种方法。

使用编译器引导的模糊测试工具有很多好处。除了 libFuzzer 之外，LLVM 工具链还有许多其他

有用的工具，例如未定义行为清理器 (UBSan)、地址清理器 (ASan) 和内存清理器 (MSan)。考虑在项目中支持 LLVM 编译，以受益于所有这些有用的工具。启用这些工具对目标进行模糊测试，以发现那些原本不会被注意到的 bug 是一个好主意。

举一个真实的例子，我建议大家看一下<https://google.github.io/clusterfuzz/setting-up-fuzzing/heartbleed-example/>，了解使用 libFuzzer 如何发现 OpenSSL 著名的 Heartbleed 漏洞。另外，可以考虑在<https://llvm.org/docs/LibFuzzer.html>上阅读 LLVM 的官方 libFuzzer 文档，包含了关于 libFuzzer 的模糊概念和使用示例等信息。

接下来，我们将研究另一个模糊测试工具——American Fuzzy Lop(AFL)。

### 11.3.2 AFL++

若因为某种原因无法使用 GCC，并且无法使用 LLVM 编译代码，那么可以使用 AFL 模糊测试器替换 libFuzzer。但原始的项目不再维护了，但项目的一个更好的增强版本 AFL++ 仍然在维护中。这一节中，将学习如何在 GCC 中使用 AFL++。

遵循类似于前面 libFuzzer 示例的示例，但使用的是 AFL++ 和 GCC，而不是 libFuzzer 和 LLVM。在继续之前，确保预置在 CMakePresets.json 中 afl-gcc 和 afl-g++ 是可用的。示例项目已经包含了这样的预设：

```
"configurePresets": [
/*...*/
, {
  "name": "afl-gcc",
  "description": "AFL GCC compiler",
  "hidden": true,
  "cacheVariables": {
    "CMAKE_C_COMPILER": {
      "type": "STRING",
      "value": "/usr/bin/afl-gcc"
    },
    "CMAKE_CXX_COMPILER": {
      "type": "STRING",
      "value": "/usr/bin/afl-g++"
    }
  }
, /*...*/
]
```

本节中，将使用 chapter11/ex02\_afl\_static\_lib 的示例。其中，有一个简单的 URI 解码器实现，我们想用 AFL++ 来模糊测试它。URI 解码器是随意实现，因此包含了严重的 Bug。URI 解码器包含一个名为 decode() 的方法：

```

1 char *uri_helper::decode(const char *str) {
2     thread_local char result[100];
3     for (auto *p = result; *str; ++str) {
4         if (*str == '%') {
5             const auto a = *++str;
6             const auto b = *++str;
7             *p++ = (a <= '9' ? a - '0' : a - 'a') * 16 + (b <=
8                 '9' ? b - '0' : b - 'a');
9         } else if (*str == '+') {
10             // replace + with space
11             *p++ = ' ';
12         } else {
13             // copy as-is
14             *p++ = *str;
15         }
16     }
17     return result;
18 }
```

该函数的预期效果是将每个两位数的 URI 编码字符，替换为对应的 ASCII 码，将加号替换为空格，并保持其他字符不变。然而，这个实现中还有一些意想不到的效果。第一个是函数没有检查 `result` 变量是否有足够的空间存储解码后的字符，当结果输出大于结果缓冲区的大小时，可能会导致缓冲区溢出。第二个函数使用空终止符作为循环条件，但不做检查地递增和解引用循环变量 `str`，这会导致读写越界。

函数中的 Bug 数量与审阅所花费的时间成正比，让我们开始模糊测试吧。

为了用 AFL++ 模糊测试 Bug，需要实现一个简单的驱动程序。驱动程序使用 `uri_helper` 库，并将标准输入提供给 `decode`。实现如下所示：

```

1 #include <library/uri_helper.hpp>
2 #include <iostream>
3
4 int main() {
5     const auto input = std::string(
6         std::istreambuf_iterator<char>(std::cin),
7         std::istreambuf_iterator<char>());
8     );
9     chapter11::ex02::uri_helper uut{};
10    std::cout << uut.decode(input.c_str());
11 }
```

前面的驱动程序应用程序实现可以与 AFL++ 一起使用。解决了这些问题之后，还需要注意最后一件事，就是 AFL++ 的初始语料库数据。

与 libFuzzer 不同，AFL++ 需要一些示例输入来启动。示例输入应该看起来像测试单元所期望的输入。由于我们的目标是 URI 解码器，一些 URI 编码的字符串应该可以做到这一点。为此，在 `chapter11/ex02_afl_static_lib/corpus` 文件夹下提供了一些 URI 编码的字符串。设置好所有内容后，通过运行以下命令来编译示例：

```
cd chapter11/  
cmake --preset="ninja-debug-afl-gcc" -S ./ -B build/  
cmake --build build/
```

这里使用 `ninja-debug-afl-gcc` 作为 CMake 配置预设，这个预设使用 `afl-gcc` 和 `afl-g++` 作为编译器。这允许 `afl++` 在编译后的代码上执行所需的插装，类似于 `libFuzzer` 的 `-fsanitize=fuzzer` 标志。构建完成后，对 fuzzing 驱动 `ch11_ex02_afl_fuzz` 运行 `afl-fuzz` 命令，并使用以下命令：

```
afl-fuzz -i ex02_afl_static_lib/corpus/ -o build/FINDINGS -m none  
-- build/ex02_afl_static_lib/fuzz/ch11_ex02_afl_fuzz
```

#### 重要的 Note

若看到关于核心转储通知的错误，那是因为系统使用了外部实用程序来处理核心转储通知。为了解决这个问题，要么应用 `afl-fuzz` 所推荐的操作，要么在执行 `afl-fuzz` 命令之前执行 `AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1`。注意，第二个选项仅推荐用于演示，不推荐用于日常使用。

这个命令乍一看可能很吓人，但其实很简单，让我们将命令进行一下解析。`-i ex02_afl_static_lib/corpus/`参数用于指定语料库数据，即前面讨论过的编码 `uri` 的示例。`-o build/discoveries` 参数是保存 `afl++` 将生成的路径。所有导致奇怪行为（如崩溃和暂停）的输入都将保存在这里。最后，`-m none` 参数用于指定运行的内存限制，其中 `none` 代表无限制。运行该命令后，应该会看到 `afl++` 的 fuzzing 进度报告的屏幕输出：

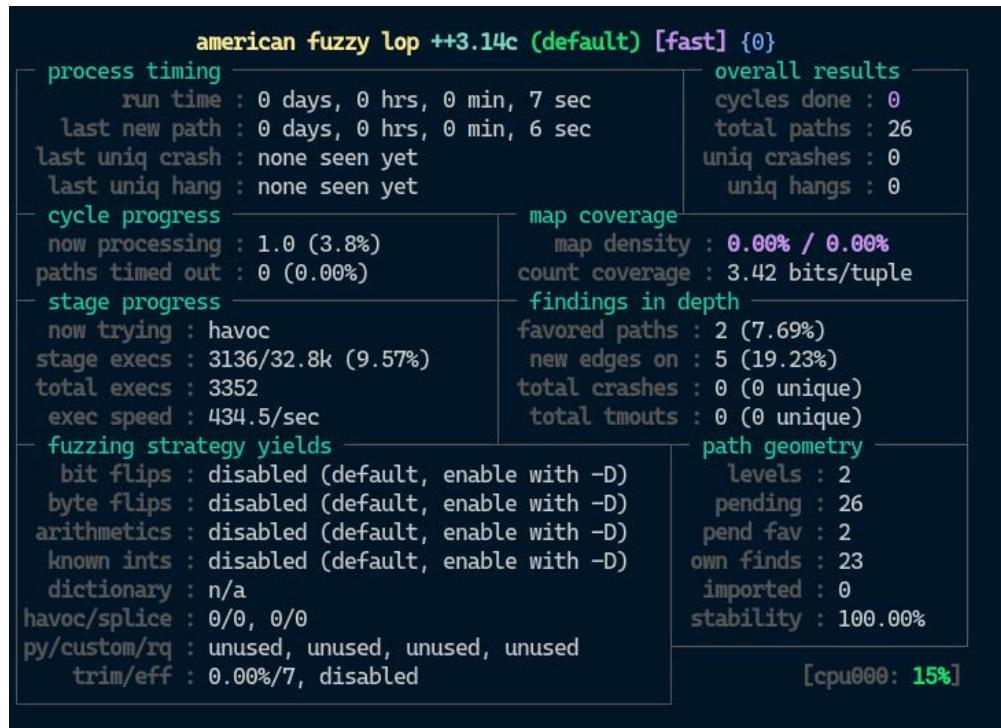


图 11.1 AFL++ 模糊测试的进度报告

屏幕上显示着一些有趣的统计数据，其中最显著的是 uniq 崩溃和 uniq 挂起。若这些数字非零，这意味着模糊测试器可能在待测单元上发现了一些 Bug。模糊测试器会不断地生成新的输入，统计数据也会相应地更新。可能需要等待几分钟或几周才能发现一些问题。让这个工具运行几分钟后，就发现了一些崩溃的情况，如下图所示：

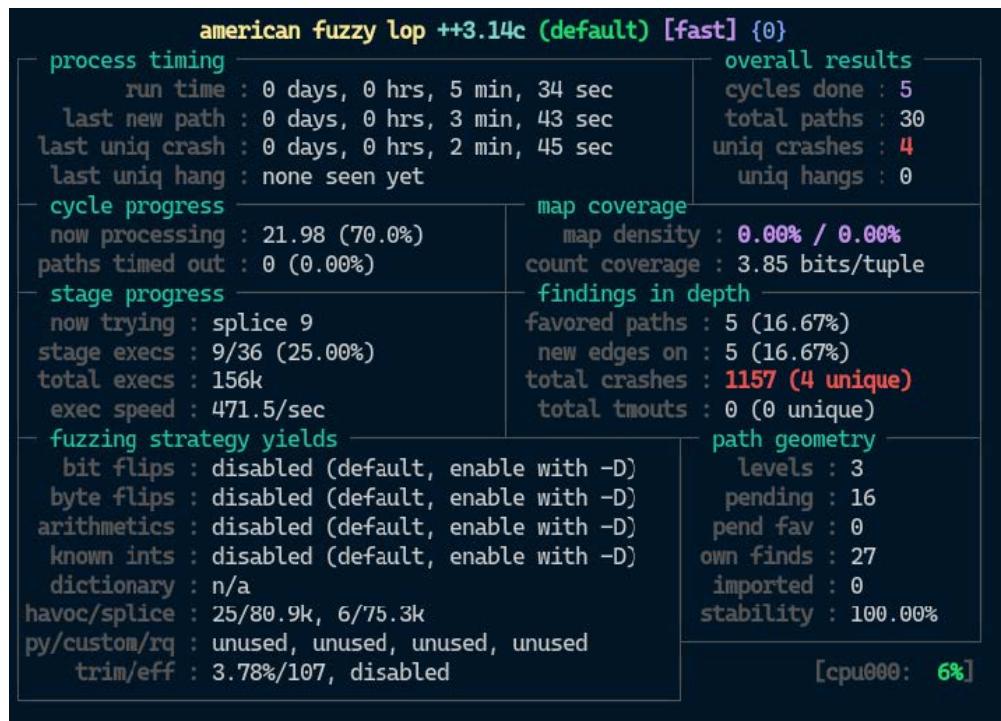


图 11.2 发现一些崩溃后的 AFL++ 模糊测试的进度报告

通过按键盘上的 Ctrl + C 取消运行，并查看导致报告崩溃的输入。导致崩溃的输入应该保存在



1. 模糊测试是什么?
2. 模糊测试中的语料库数据是干嘛的?
3. 如何在 CMake 中为目标启用 libFuzzer ?
4. 说出一个使用模糊测试可以发现的软件错误。

# 第三部分：掌控细节

将学习如何在跨平台项目上使用 CMake，重用 CMake 代码，优化和维护现有的项目，以及迁移非 CMake 项目到 CMake。还将了解如何为 CMake 项目做出贡献，以及推荐的扩展阅读材料。

本节包括以下几章：

- 第 12 章，跨平台编译和自定义工具链
- 第 13 章，重用 CMake 代码
- 第 14 章，优化和维护 CMake 项目
- 第 15 章，迁移到 CMake
- 第 16 章，对 CMake 进行贡献

# 第 12 章 跨平台编译和自定义工具链

CMake 支持软件的跨平台构建，只要运行 CMake 的系统上有必要的工具，就可以为其他平台构建项目。构建软件时，通常要了解编译器和链接器，这些是构建软件的基本工具，但在构建软件时还有一些其他工具、库和文件，这些统称为工具链。

本书所有的示例都是运行在同一系统上构建，CMake 通常能很好地找到正确的工具链。然而，若软件是为另一个平台构建，则工具链通常由开发人员指定。工具链的定义可能相对简单，只指定目标平台，也可能复杂到指定构建软件所需的每个工具的路径，或者为特定芯片组创建二进制文件所需的特定编译器标志。

交叉编译中，工具链通常和系统根目录 (sysroot) 一起使用。sysroot 是目标平台的精简文件系统目录。构建软件时，将其视为根文件夹，用于查找必要的库和文件，以便为预期的目标平台进行编译和链接。

虽然交叉编译开始可能令人生畏，但当正确使用 CMake 后，就没那么困难了。本章将介绍如何使用工具链文件，以及如何自己编写工具链文件。我们将详细研究在构建软件的特定阶段使用了哪些工具。最后，再来了解如何设置 CMake，从而可以使用模拟器运行测试。

我们将讨论以下主题：

- 使用跨平台工具链
- 创建工具链
- 测试交叉编译的二进制文件
- 测试工具链支持的功能

本章结束时，将能够处理现有的工具链，以及如何使用 CMake 为不同平台构建和测试软件。我们将深入研究如何测试编译器的某个特性，以确定是否符合我们的预期。

## 12.1. 相关准备

与前几章一样，这些示例是用 CMake 3.21 测试的，并在以下编译器上运行：

- GNU 编译器集合 9 (GCC 9) 或更高版本，包括用于 arm 硬浮点 (armhf) 架构的交叉编译器。
- clang12 或更高版本。
- Microsoft Visual Studio c++ 19 (MSVC 19) 或更高版本。
- 对于 Android 的例子，需要 Android 原生开发工具包 (Android NDK) 23b 或更高版本。
- 对于 Apple 嵌入式的例子，推荐使用 Xcode 12 或更高版本和 iOS 软件开发工具包 12.4 (iOS SDK 12.4)。

本书的所有示例和源代码都可以在 GitHub 库中找到。本章中，CMake 预设和构建容器的示例都在库的根文件夹中。若缺少相应的软件，可以将相应的示例将从构建中排除。库可以在这里找到：<https://github.com/PacktPublishing/CMake-Best-Practices>。

## 12.2. 使用跨平台工具链

为多个平台构建软件时，最直接的方法是在目标系统上编译。这样做的缺点是每个开发人员都必须有一个目标系统，才能构建软件。若是桌面系统可能还好，若是功能较弱的设备（如嵌入式系统），可能会因为缺乏适当的开发工具或编译软件需要花费很长时间。

因此，从开发人员的角度来看，更方便的方法是使用交叉编译。从而软件工程师在自己的机器上编写代码并构建软件，但生成的二进制文件适用于不同的平台。构建软件的机器通常称为主机平台，而运行软件的平台则称为目标平台。例如，开发人员在运行 Linux 的 x64 桌面计算机上编写代码，但生成的二进制文件是针对 arm64 处理器上的嵌入式 Linux 的。所以主机平台是 x64 Linux，目标平台是 arm64 Linux。要交叉编译软件，需要了解以下两件事：

- 工具链可以生成正确格式的二进制文件
- 提供编译目标系统目的依赖项

工具链是一组工具，如编译器、链接器和打包器，用于生成运行在主机系统上为目标系统生成输出的二进制文件。依赖项通常收集在 sysroot 目录中，sysroot 是包含精简版本的根文件系统的目录，所需的库存储在其中。对于交叉编译，这些目录为搜索依赖项的根目录。

一些工具，例如用于构建嵌入式 Linux 发行版的 Yocto Project (YP)，可以构建 sysroot 和工具链，用于开箱即用的交叉编译。这种自动化通常很方便，但当不可用时，手动创建 sysroot 可以像从目标平台复制或挂载文件系统到主机上的目录一样简单。有时，工具链和 sysroot 的组合也称为 sdk。这些 sdk 还可能包含用于调试的进一步工具或用于运行跨平台构建的模拟器的定义，CMake 使用工具链文件为交叉编译配置构建工具和 sysroot。工具链文件是普通的 CMake 脚本，主要设置一些缓存变量来描述目标平台和工具链的各个组件的位置。工具链文件可以通过设置 CMAKE\_TOOLCHAIN\_FILE 变量传递给 CMake，或使用自 CMake 3.21 起支持的--toolchain 选项：

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake -S <SourceDir> -B <BuildDir>
cmake --toolchain toolchain.cmake -S <SourceDir> -B <BuildDir>
```

两种方式是等价的。若 CMAKE\_TOOLCHAIN\_FILE 设置为一个环境变量，CMake 也会解释这个变量。若使用 CMake 预设，配置预设可以配置一个带有 toolchainFile 选项的工具链文件：

```
{
  "name": "arm64-build-debug",
  "generator": "ninja",
  "displayName": "Arm 64 Debug",
  "toolchainFile": "/path/to/toolchain.cmake",
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Debug"
  }
},
```

`toolchainFile` 选项支持宏扩展。若工具链文件的路径是一个相对路径，CMake 将首先相对于构建目录查找，若在那里没有，将从源目录搜索。由于 `CMAKE_TOOLCHAIN_FILE` 是一个缓存变量，只需要在 CMake 的第一次运行时指定，后续的运行将使用缓存的值。

第一次运行时，CMake 将执行一些内部检查，以确定工具链支持哪些特性。不管工具链是用工具链文件指定的，还是使用默认的系统工具链。CMake 将在第一次运行时输出各种特性和属性的测试结果：

```
-- The CXX compiler identification is GNU 9.3.0
-- The C compiler identification is GNU 9.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/arm-linuxgnueabihf-g++- 9 - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/arm-linux-gnueabigcc-9 - skipped
-- Detecting C compile features
-- Detecting C compile features - done
```

特性的检测通常发生在 `CMakeLists.txt` 文件中对 `project()` 的第一次使用时，但当之后的 `project()` 来启用之前禁用的语言，则将触发进一步的检测。若使用 `enable_language()` 来启用 `CMakeLists.txt` 文件中的另一种编程语言，也会发生同样的情况。

由于工具链的特性和测试结果可以缓存，因此无法更改已配置构建目录的工具链。CMake 可能会检测到工具链已经更改，但缓存变量的替换不完全。因此，在更改工具链之前，最好完全删除构建目录。

#### 配置完成后切换工具链

切换工具链之前，始终要完全清空构建目录。只删除 `CMakeCache.txt` 文件是不够的，因为与工具链相关的内容可能会缓存在不同的位置。

CMake 的工作模式是一个项目应该对所有东西使用相同的工具链。因此，不直接支持使用多个工具链。若这是真的需要的，项目中需要不同工具链的部分必须配置为子构建。

工具链体积应该保持尽可能小，并且与项目完全解耦。理想情况下，可用于不同的项目。工具链文件通常与用于交叉编译的 SDK 或 sysroot 捆绑在一起。然而，有时需要手动书写。

### 12.3. 创建工具链

对于交叉编译工具链很难的误解可能源于这样一个事实，互联网上有许多过于复杂的工具链文件。其中许多是为 CMake 的早期版本编写的，因此实现需要许多额外的测试和检查，现在这些测试和检查现在已经内置为 CMake 的一部分。CMake 工具链文件基本上做以下几件事：

- 定义目标系统和架构。

- 为平台构建软件提供所需工具的路径，通常是编译器。
- 为编译器和链接器设置默认标志。
- 若是交叉编译，则指向 sysroot 和暂存目录。
- 为 CMake 的 `find_` 指令设置搜索顺序提示。更改搜索顺序是项目可以定义的，这属于工具链文件，还是应该由项目处理信息，目前具有争议。

执行这些操作的工具链示例如下所示：

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_C_COMPILER /usr/bin/arm-linux-gnueabi-gcc-9)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabihf-g++-9)

set(CMAKE_C_FLAGS_INIT -pedantic)
set(CMAKE_CXX_FLAGS_INIT -pedantic)

set(CMAKE_SYSROOT /home/builder/raspi-sysroot/)
set(CMAKE_STAGING_PREFIX /home/builder/raspi-sysroot-staging/)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE BOTH)
```

本例将定义一个工具链，目标是在 Advanced RISC Machine (ARM) 处理器上运行的 Linux 操作系统的构建。要使用的 C 和 C++ 编译器使用一个版本的 GCC，并且安装在主机系统的/usr/bin/文件夹中。编译器用`-pedantic` 标记严格的输出，输出国际标准组织 (ISO) C 和 ISO C++ 标准化要求的所有警告。接下来，用于查找所需库的 sysroot 设置为/home/builder/raspi-sysroot/，交叉编译时安装程序的暂存目录设置为/home/builder/raspi-sysroot-staging/。最后，更改 CMake 的搜索行为，以便在主机系统上搜索程序，而在 sysroot 中搜索库(包括文件和包)。工具链文件是否应该影响搜索行为，目前存在争议。因为只有项目知道要寻找什么，所以在工具链文件中做假设可能会破坏项目结构。但只有工具链知道要使用哪个系统根目录，以及其中包含什么类型的文件，所以让工具链来定义可能比较方便。一种中间方法是使用 CMake 预设来定义工具链和搜索行为，而不是将其放入项目或工具链文件中。

### 12.3.1 定义目标系统

交叉编译的目标系统由以下三个变量定义: `CMAKE_SYSTEM_NAME`、`CMAKE_SYSTEM_PROCESSOR` 和 `CMAKE_SYSTEM_VERSION`。与它们对应的是 `CMAKE_HOST_SYSTEM_NAME`、`CMAKE_HOST_SYSTEM_PROCESSOR` 和 `CMAKE_HOST_SYSTEM_VERSION` 变量，这些变量描述了执行构建的平台。

`CMAKE_SYSTEM_NAME` 变量描述要为其构建软件的目标操作系统，会让 CMake 将 `CMAKE_CROSSCOMPILING` 设置为 true。典型的值是 Linux、Windows、Darwin、Android 或 QNX，也可以使用更具体的平台名称，如 WindowsPhone、WindowsCE、WindowsStore 等。对于嵌入式设备，`CMAKE_SYSTEM_NAME` 变量可设置为 Generic。在撰写本书时，CMake 文档中还没有支持系统的官方列表，可以检查本地 CMake 安装中的/Modules/Platform 文件夹中的文件。

`CMAKE_SYSTEM_PROCESSOR` 变量用于描述平台的硬件架构，没有指定则设置为 `CMAKE_HOST_SYSTEM_PROCESSOR` 变量的值。当 64 位平台交叉编译 32 位平台时，也应该设置目标处理器架构，即使处理器的类型相同。对于 Android 和 Apple 平台，处理器通常不指定。当对 Apple 目标进行交叉编译时，实际的设备由所使用的 SDK 定义，该 SDK 由 `CMAKE OSX_SYSROOT` 变量指定。当交叉编译 Android，有专门的变量，如 `CMAKE_ANDROID_ARCH_ABI`，`CMAKE_ANDROID_ARM_MODE` 和可选的 `CMAKE_ANDROID_ARM_NEON` 来控制目标架构。

定义目标系统的最后一个变量是 `CMAKE_SYSTEM_VERSION`，其取决于所构建的系统。对于 WindowsCE、WindowsStore 和 WindowsPhone，它将用于定义要使用的 Windows SDK 的哪个版本。在 Linux 上经常省略，或者可能包含目标系统的内核修订版(若与此相关)。

使用 `CMAKE_SYSTEM_NAME`、`CMAKE_SYSTEM_PROCESSOR` 和 `CMAKE_SYSTEM_VERSION` 变量，目标平台通常是完全指定的。然而，一些生成器(如 Visual Studio)直接支持原生平台，可以使用 CMake 的-A 命令行选项设置体系结构，如下所示：

```
cmake -G "Visual Studio 2019" -A Win32 -T host=x64
```

当使用预设时，架构设定可以在配置预设中使用，以达到同样的效果。当定义了目标系统，就等于定义了实际构建软件的工具。

一些编译器，如 Clang 和 QNX(非 Unix GCC (QNX GCC))，本质上是交叉编译器以平台为参数。为了将参数传递给这些编译器，使用了 `CMAKE_<LANG>_COMPILER_TARGET` 变量。对于 Clang 来说，该值是一个目标三元组，如 arm-linux-gnueabihf，而对于 QNX GCC，编译器名称和目标的值为 gcc\_ntoarmv7le。Clang 支持的三元组在其官方文档<https://clang.llvm.org/docs/CrossCompilation.html>中有描述。

有关 QNX 的可用选项，可以参考<https://www.qnx.com/developers/docs/>的 QNX 文档。

因此，使用 Clang 的工具链文件可能像这样：

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_C_COMPILER /usr/bin/clang)
set(CMAKE_C_COMPILER_TARGET arm-linux-gnueabihf)

set(CMAKE_CXX_COMPILER /usr/bin/clang++)
set(CMAKE_CXX_COMPILER_TARGET arm-linux-gnueabihf)
```

本例中，Clang 用于为运行在 ARM 处理器上的 Linux 系统编译 C 和 C++ 代码，该系统支持硬件浮点。定义目标系统通常对要使用的构建工具有直接影响。下一节中，我们将研究如何选择编译器和相关工具进行交叉编译。

### 12.3.2 选择构建工具

构建软件时，编译器是主要工具。大多数情况下，设置编译器就足够了。编译器的路径由 `CMAKE_<LANG>_COMPILER` 缓存变量设置，该变量可以在工具链文件中设置，也可以手动传递给 CMake。若路径是绝对的，将直接使用；否则，将使用与 `find_program()` 相同的搜索顺序进行搜索，这就是为什么必须谨慎对待更改工具链文件中搜索行为的原因。若工具链文件和用户都没有指定编译器，CMake 将根据指定的目标平台和生成器自动选择一个编译器。此外，编译器可以设置在以 `<LANG>` 命名的环境变量上。因此，C 将设置 C 编译器，CXX 设置 C++ 编译器，ASM 设置汇编器等。

一些生成器（如 Visual Studio）支持的不同的工具集定义。可以使用 `-T` 命令行选项进行设置。以下命令将告诉 CMake 为 Visual Studio 生成代码，为 32 位系统生成二进制文件，但要使用 64 位编译器来编译：

```
cmake -G "Visual Studio 2019" -A Win32 -T host=x64
```

这些值也可以通过工具链文件中的 `CMAKE_GENERATOR_TOOLSET` 变量来设置。这不应该在项目内部设置，因为它破坏了 CMake 项目文件不受生成器和平台影响的想法。

对于 Visual Studio 用户来说，通过安装相同版本的预览版和正式发布版，可以安装同一 Visual Studio 版本的多个实例。若是这种情况，`CMAKE_GENERATOR_INSTANCE` 变量可以设置为工具链文件中 Visual Studio 的绝对安装路径。

通过指定要使用的编译器，CMake 将为编译器和链接器选择默认标志，并通过设置 `CMAKE_<LANG>_FLAGS` 和 `CMAKE_<LANG>_FLAGS_<CONFIG>` 使其在项目中可用，其中 `<LANG>` 代表编程语言，`<CONFIG>` 表示构建配置，如 `Debug` 或 `Release` 等。默认的链接器标志是由 `CMAKE_<TARGETTYPE>_LINKER_FLAGS` 和 `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>` 变量设置的，其中 `<TARGETTYPE>` 是 `EXE`、`STATIC`、`SHARED` 或 `MODULE`。

要将自定义标志添加到默认标志中，每个变量都有一个 `_INIT` 的变量——例如，`CMAKE_<LANG>_FLAGS_INIT`。使用工具链文件时，使用 `_INIT` 变量设置必要的标志。用 GCC 从 64 位主机为 32 位目标编译的工具链文件如下所示：

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR i686)

set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)
```

```
set(CMAKE_C_FLAGS_INIT -m32)
set(CMAKE_CXX_FLAGS_INIT -m32)

set(CMAKE_EXE_LINKER_FLAGS_INIT -m32)
set(CMAKE_SHARED_LINKER_FLAGS_INIT -m32)
set(CMAKE_STATIC_LINKER_FLAGS_INIT -m32)
set(CMAKE_MODULE_LINKER_FLAGS_INIT -m32)
```

对于简单的项目，设置目标系统和工具链可能已经足够创建二进制文件了，但是对于更复杂的项目，可能需要访问目标系统的库和头文件，需要在工具链文件中指定 `sysroot`。

### 12.3.3 设置 `sysroot`

交叉编译时，所有链接的依赖项必须与目标平台匹配，处理此问题的常用方法是创建 `sysroot`，它是文件夹中目标系统的根文件系统。虽然 `sysroot` 可能包含完整的系统，但它们只提供所需的内容。

将 `CMAKE_SYSROOT` 设置为 `sysroot` 的路径。若设置了该值，CMake 将默认情况下首先在 `sysroot` 中查找库和头文件，除非另有指定。大多数情况下，CMake 还会自动设置必要的编译器和链接器标志，使工具与 `sysroot` 一起工作。

不应该直接在 `sysroot` 中安装构建构件的情况下，可以设置 `CMAKE_STAGING_PREFIX` 变量来提供另一种安装路径。`sysroot` 应该保持干净或者将其挂载为只读时，通常会出现这种情况。`CMAKE_STAGING_PREFIX` 设置不会将这个目录添加到 `CMAKE_SYSTEM_PREFIX_PATH` 中，所以只有当工具链中的 `CMAKE_FIND_ROOT_PATH_MODE_PACKAGE` 变量设置为 `BOTH` 或 `NEVER` 时，才会使用 `find_package()` 来查找安装在暂存目录中的东西。

定义目标系统和设置工具链配置、`sysroot` 和暂存目录通常是交叉编译所需的全部内容。两个例外是对 Android 和 Apple 的 iOS、tvOS 或 watchOS 的交叉编译。

### 12.3.4 交叉编译——Android

以前，Android 的 NDK 和各种 CMake 版本之间的兼容性有时很不愉快，因为新版本的 NDK 突然之间不再像以前的版本那样与 CMake 工作。然而，现在已经有了很大的改善，因为 r23 版本的 Android NDK 现在将使用 CMake(3.21 或更高的版本) 的内部支持来处理 Android NDK。Android NDK 与 CMake 集成的官方文档可以在这里看到：<https://developer.android.com/ndk/guides/cmake>。

从 r23 起，NDK 提供 CMake 工具链文件，位于 `<NDK_ROOT>/build/cmake/android.toolchain.cmake`，可以像常规的工具链文件一样使用。NDK 还包括基于 clang 的工具链所需的所有工具，因此通常不需要定义进一步的工具。为了控制目标平台，以下 CMake 变量应该通过命令行或 CMake 预设设置：

- `ANDROID_ABI`: 指定要使用的目地应用程序二进制接口 (ABI)。有效值为 `armeabi-v7a`、`arm64-v8a`、`x86` 和 `x86_64`。为 Android 交叉编译时，应该始终设置这个值。

- ANDROID\_ARM\_NEON: 启用 armeabi-v7a 的 NEON 支持。此变量对其他 ABI 版本没有影响。当使用 r21 版本以上的 NDK 时, NEON 支持默认启用, 很少需要禁用。
- ANDROID\_ARM\_MODE: 指定是否为 armeabi-v7a 生成 ARM 或 Thumb 指令。有效值是 thumb 或 arm。此变量对其他 ABI 版本没有影响。
- ANDROID\_LD: 选择是使用默认链接器, 还是使用 llvm 中的实验性的 lld。有效值是默认值或 lld, 但由于 lld 处于实验状态, 这个变量通常在生产构建中忽略。
- ANDROID\_PLATFORM: 指定应用程序支持的最小 API 级别, 格式为 \$API\_LEVEL, android-\$API\_LEVEL, 或 android-\$API\_LETTER, 其中 \$API\_LEVEL 是一个数字, \$API\_LETTER 是平台的版本代码, ANDROID\_NATIVE\_API\_LEVEL 是变量的别名。虽然没有必要严格地设置 API 级别, 但通常都会这样做。
- ANDROID\_STL: 指定用于此应用程序的标准模板库 (STL)。这可以是 c++\_static(这是默认值), c++\_shared, none 或 system。无论是 c++\_shared 还是 c++\_static 都需要支持现代 C++。系统库只为 C 库头文件提供 new、delete 和 C++ 包装器, 而没有库提供 STL 支持。

使用 CMake 来配置 Android 版本可能像这样:

```
cmake -S . -B build --toolchain <NDK_DIR>/build/cmake/android
      .toolchain.cmake -DANDROID_ABI=armeabi-v7a -DANDROID_PLATFORM=23
```

这个调用将指定一个需要 API 级别 23 或更高的构建, 这对应于 32 位 ARM 中央处理单元 (CPU) 的 Android 6.0 或更高。

使用 NDK 提供工具链的另一种选择是将 CMake 指向 Android NDK 的位置, 对于 r23 版本或更高版本的 NDK, 这是推荐的方法。然后使用各自的 CMake 变量进行目标平台的配置, 将 CMAKE\_SYSTEM\_NAME 变量设置为 android, 将 CMAKE\_ANDROID\_NDK 变量设置为 android NDK 的位置, 就可以在 CMake 中使用 NDK 了。这既可以通过命令行进行, 也可以在工具链文件中进行。若 ANDROID\_NDK\_ROOT 或 ANDROID\_NDK 环境变量已经设置, 将作为 CMAKE\_ANDROID\_NDK 的值。

以这种方式使用 NDK 时, 配置在 CMAKE\_ 变量上定义, 相当于直接调用 NDK 的工具链文件时使用的变量:

- CMAKE\_ANDROID\_API 或 CMAKE\_SYSTEM\_VERSION 用于指定要构建的最低 API 级别。
- CMAKE\_ANDROID\_ARCH\_ABI 用于指定要使用的 ABI 模式。
- CMAKE\_ANDROID\_STL\_TYPE 指定要使用的 STL。

用 Android NDK 配置 CMake 的工具链文件示例如下:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 21)
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK /path/to/the/android-ndk-r23b)
set(CMAKE_ANDROID_STL_TYPE c++_static)
```

使用 Visual Studio 生成器为 Android 交叉编译时，CMake 需要安装 NVIDIA Nsight Tegra 的 Visual Studio 插件，或使用 Android NDK 的 Visual Studio 工具。当使用 Visual Studio 构建 Android 二进制文件时，可以通过将 CMAKE\_ANDROID\_NDK 变量设置为 NDK 的位置来使用 CMake 的 Android NDK 的内置支持。

CMake 3.20 起，NDK 的最新版本和 CMake 版本使得交叉编译 Android 本地代码变得更加容易。交叉编译的另一个特殊情况是针对 Apple 的 iOS、tvOS 或 watchOS。

### 12.3.5 交叉编译——iOS、tvOS 或 watchOS

推荐交叉编译 Apple 的 iPhone、Apple TV 或 Apple Watch 的方法是使用 Xcode 生成器。Apple 在为这些设备构建应用程序时使用的工具非常有限，因此使用 macOS 或运行 macOS 的虚拟机 (VM) 是必要的。虽然也可以使用 Makefiles 或 Ninja 文件，但需要对 Apple 系统有更深入的了解才能正确配置。

为了对这些设备进行交叉编译，需要使用 Apple 设备 SDK，并将 CMAKE\_SYSTEM\_NAME 变量设置为 iOS、tvOS 或 watchOS，如下例所示：

```
cmake -S <SourceDir> -B <BuildDir> -G Xcode -DCMAKE_SYSTEM_NAME=iOS
```

对于相对现代的 sdk 和 3.14 或更高版本的 CMake，这通常就够了。默认情况下，使用系统上可用的最新设备 SDK，但若确实需要，可以通过将 CMAKE OSX\_SYSROOT 变量设置为 SDK 的路径来选择不同的 SDK。最小目标平台版本可以通过 CMAKE\_OSX\_DEPLOYMENT\_TARGET 变量指定。

在为 iPhone、Apple TV 或 Apple watch 交叉编译时，目标可以是真实的设备，也可以是不同 sdk 附带的设备模拟器。然而，Xcode 内置支持在构建部分切换，所以 CMake 不需要运行两次。如果选择了 Xcode 生成器，CMake 内部使用 xcodebuild 命令行工具，它支持-sdk 选项来选择所需的 SDK。当通过 CMake 构建时，选项可以像这样使用：

```
cmake -build <BuildDir> -- -sdk <sdk>
```

把带有指定值的-sdk 选项传递给 xcodebuild。iOS 允许的值为 iphoneos 或 iphonesimulator，Apple TV 设备允许的值为 appletvos 或 appletsimulator，Apple watch 允许的值为 watchos 或 watchsimulator。Apple 嵌入式平台要求对某些构建构件进行强制签名。

对于 Xcode 生成器，开发团队标识符 (ID)，通常是大约 10 个字符的短字符串，可以通过 CMAKE\_XCODE\_ATTRIBUTE DEVELOPMENT\_TEAM 缓存变量来指定。

为 Apple 嵌入式设备构建时，模拟器可以方便地测试代码，而不需要每次都部署到设备上。这种情况下，最好通过 Xcode 或 xcodebuild 进行测试，但对于其他平台，交叉编译的代码可以直接通过 CMake 和 CTest 进行测试。

## 12.4. 测试交叉编译的二进制文件

能够毫不费力地为不同的架构交叉编译二进制文件，这为相关人员的开发人员工作带来了很多便利。但这些工作并不只构建二进制文件，还包括运行测试。若软件也在主机工具链上编译，并且测试足够通用，那么在主机上运行测试可能是测试软件的最简单方法。尽管在切换工具链和频繁地重新构建时，可能会花费一些时间。若这不可能或太耗时，一种替代方法就是在真正的目标硬件上运行测试，但这取决于硬件的可用性和在硬件上设置测试的工作量。因此，中间方案是在目标平台的模拟器中运行测试（若这可用）。

定义一个运行测试的模拟器，需要使用 CROSSCOMPILING\_EMULATOR 目标属性，既可以为单个目标设置，也可以通过设置 CMAKE\_CROSSCOMPILING\_EMULATOR 缓存变量全局设置，该缓存变量包含一个分号分隔的命令和参数列表，用于运行模拟器。若全局设置，该命令将添加到 add\_test()、add\_custom\_command() 和 add\_custom\_target() 中指定命令的前缀处，将用于运行 try\_run() 生成的可执行文件，所以用于构建的所有定制命令也必须在模拟器中访问和运行。CROSSCOMPILING\_EMULATOR 属性并不一定是实际的模拟器——可以是任何程序，比如将二进制文件复制到目标计算机上，并在那里执行的脚本。

设置 CMAKE\_CROSSCOMPILING\_EMULATOR 应该通过工具链文件、命令行或配置的前缀进行。用于为 ARM 交叉编译 C++ 代码的工具链文件示例，其中使用了开源模拟器 QEMU 来运行测试，如下所示：

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_SYSROOT /path/to/arm/sysroot/)
set(CMAKE_CXX_COMPILER /usr/bin/clang++)
set(CMAKE_CXX_COMPILER_TARGET arm-linux-gnueabihf)

set(CMAKE_CROSSCOMPILING_EMULATOR "qemu-arm;-L;${CMAKE_SYSROOT}")
```

除了设置目标系统的信息和交叉编译示例中最后一行的工具链之外，还将模拟器命令设置为 qemu-arm -L/path/to/arm/sysroot。假设 CMakeLists.txt 文件包含如下测试：

```
add_test(NAME exampleTest COMMAND exampleExe)
```

当运行 CTest 时，测试命令转换为以下内容：

```
qemu-arm "-L" "/path/to/arm/sysroot/" "/path/to/build-dir/exampleExe"
```

模拟器中运行测试可以加快开发人员的工作流程，因为它消除了主机工具链和目标工具链之间切换的必要，并且不需要为每个表面测试将构建构件移动到目标硬件。在持续集成 (CI) 构建中，使用这样的仿真器也很方便。

关于 CMAKE\_CROSSCOMPILING\_EMULATOR，还可以用来将测试封装在调试工具中，如 Valgrind 或类似的调试工具。由于运行指定的模拟器可执行文件不依赖于

`CMAKE_CROSSCOMPILING` 变量, `CMAKE_CROSSCOMPILING_EMULATOR` 变量的缺点是, 设置 `CMAKE_CROSSCOMPILING_EMULATOR` 变量将影响 `try_run()`, 该指令通常用于测试工具链支持的特性或依赖, 并且作为调试实用程序, 可能导致编译器测试失败。可能有必要在已经缓存的构建上运行, 其中 `try_run()` 的结果已经缓存。正因为如此, 不应该永久地使用 `CMAKE_CROSSCOMPILING_EMULATOR` 变量来运行调试程序执行, 而应该在查找问题时, 在特定的开发场景下执行。

本节中, 提到了 CMake 的 `try_run()` 指令, 该指令与密切相关的 `try_compile()` 指令一起使用, 可以检查编译器或工具链中某些特性的可用性。下一节中, 我们将更深入地研究这两个指令和测试工具链的特性。

#### 12.4.1 测试工具链支持的功能

当 CMake 第一次在项目树上运行时, 就会为编译器和语言特性执行各种测试。对 `project()` 或 `enable_language()` 的使用都会触发测试, 但结果可能在上一次运行中缓存了, 这也是不建议在现有构建中切换工具链的原因。

大多数检查将在内部使用 `try_compile()` 指令来执行这些测试。这个命令实际上是由用户进行检测, 或工具链构建一个小的二进制文件。所有相关的全局变量, 如 `CMAKE_<LANG>_FLAGS` 将转发至 `try_compile()`。

与其密切相关的是 `try_run()`, 该指令在内部调用 `try_compile()`, 若成功, 将尝试运行程序。对于常规编译器检查, 不使用 `try_run()`, 对它的使用通常都会在项目中定义。

要编写自定义检查, 而不是直接调用 `try_compile()` 和 `try_run()`, 建议使用 `CheckSourceCompiles` 或 `CheckSourceRuns` 模块的 `check_source_compiles()` 或 `check_source_runs()`, 这些指令自 CMake 3.19 起就已经可用。大多数情况下, 足以生成所需的信息, 而不需要处理更复杂的 `try_compile()` 或 `try_run()`。这两个指令的签名非常相似:

```
check_source_compiles(<lang> <code> <resultVar>
    [FAIL_REGEX <regex1> [<regex2>...]] [SRC_EXT <extension>])
check_source_runs(<lang> <code> <resultVar>
    [SRC_EXT <extension>])
```

`<lang>` 参数指定 CMake 支持的一种语言, 例如对于 C++ 来说是 C 或 CXX。`<code>` 是将作为可执行文件链接的字符串的代码, 因此必须包含 `main()` 函数。编译的结果将作为布尔值存储在 `<resultvar>` 缓存变量中。若为 `check_source_compiles` 提供 `FAIL_REGEX`, 编译的输出将根据提供的表达式进行检查。代码将保存在一个临时文件中, 扩展名与所选语言匹配; 若文件的扩展名与默认扩展名不同, 可以通过 `SRC_EXT` 选项指定。

也有一些特定于语言的模块, 称为 `Check<LANG>SourceCompiles` 和 `Check<LANG>SourceRuns`, 提供了各自的指令:

```
include(CheckCSourceCompiles)
check_c_source_compiles(code resultVar
    [FAIL_REGEX regexes...])
```

```

)
include(CheckCXXSourceCompiles)
check_cxx_source_compiles(code resultVar
    [FAIL_REGEX regexes...])
)

```

假设 C++ 项目可以使用标准库的原子功能，若不支持，则退回到不同的实现。编译器对此的检查如下所示：

```

include(CheckSourceCompiles)

check_source_compiles(CXX "
#include <atomic>
int main(){
    std::atomic<unsigned int> x;
    x.fetch_add(1);
    x.fetch_sub(1);
}" HAS_STD_ATOMIC)

```

包含模块后，小程序使用 `check_source_compiles()`，在该程序中使用要检查的功能。若代码编译成功，`HAS_STD_ATOMIC` 将设置为 `true`；否则，设置为 `false`。该测试在项目配置期间执行，会将打印如下状态消息：

```

[cmake] -- Performing Test HAS_STD_ATOMIC
[cmake] -- Performing Test HAS_STD_ATOMIC - Success

```

结果将缓存，这样 CMake 后续运行不会再次执行测试。很多情况下，检查程序是否编译已经提供了关于工具链的某个特性的足够信息，但有时必须运行底层程序才能获得所需的信息。为此，`check_source_runs()` 类似于 `check_source_compiles()`。`check_source_runs()` 的使用注意事项是：若设置了 `CMAKE_CROSSCOMPILING` 但没有设置模拟器命令，测试将只编译测试而不运行，除非设置了 `CMAKE_CROSSCOMPILING_EMULATOR`。

有许多形式的 `CMAKE_REQUIRED_*` 变量来控制检查如何编译代码。注意，这些变量缺少特定于语言的部分，若在运行针对不同语言的测试的项目上工作，则需要特别注意这一点。以下是对其中一些变量的解释：

- `CMAKE_REQUIRED_FLAGS` 用于在 `CMAKE_<LANG>_FLAGS` 或 `CMAKE_<LANG>_FLAGS_<CONFIG>` 变量中指定的任何标志后，传递额外的标志给编译器。
- `CMAKE_REQUIRED_DEFINITIONS` 指定格式为 `-DFOO=bar` 的编译器宏定义。
- `CMAKE_REQUIRED_INCLUDES` 指定要搜索的头文件目录列表。

- CMAKE\_REQUIRED\_LIBRARIES 指定链接程序时要添加的库列表。这些可以是库的文件名，也可以是导入的 CMake 目标。
- CMAKE\_REQUIRED\_LINK\_OPTIONS 指定链接器标志的列表。
- CMAKE\_REQUIRED QUIET 可设置为 true，来静默自检查的状态消息。

检查需要相互隔离的情况下，CMakePushCheckState 模块提供了 cmake\_push\_check\_state(), cmake\_pop\_check\_state() 和 cmake\_reset\_check\_state() 来存储配置，恢复之前的配置，并重置配置：

```

include(CMakePushCheckState)
cmake_push_check_state()

# Push the state and clean it to start with a clean check state
cmake_reset_check_state()

include(CheckCompilerFlag)
check_compiler_flag(CXX -Wall WALL_FLAG_SUPPORTED)

if(WALL_FLAG_SUPPORTED)
  set(CMAKE_REQUIRED_FLAGS -Wall)

  # Preserve -Wall and add more things for extra checks
  cmake_push_check_state()
  set(CMAKE_REQUIRED_INCLUDES ${CMAKE_CURRENT_SOURCE_DIR}/include)

  include(CheckSymbolExists)
  check_symbol_exists(hello "hello.hpp" HAVE_HELLO_SYMBOL)

  cmake_pop_check_state()
endif()

# restore all CMAKE_REQUIRED_VARIABLEs to original state
cmake_pop_check_state()

```

检查编译或运行测试程序的更复杂的 try\_compile() 和 try\_run()。虽然可以外部使用，但它们主要用于内部使用，因此建议参考命令的官方文档，就不在这里解释它们了。

通过编译和运行程序来检查编译器特性，是检查工具链特性的一种通用方法。有些检查非常常见，以至于 CMake 为它们提供了专门的模块和功能。

## 12.4.2 工具链和语言特性的常规检查

对于一些常规的特性检查，例如检查是否支持编译器标志或是否存在头文件，CMake 提供了自己的模块。从 CMake 3.19 起，以该语言作为参数的通用模块就存在了，但相应的 Check<LANG>... 仍然可以使用于特定的语言模块。

CheckLanguage 模块是检查某一语言的编译器是否可用的测试，若没有设置

`CMAKE_<LANG>_COMPILER` 变量，可用来检查某一语言的编译器是否可用。检查 Fortran 是否可用的示例如下：

```
include(CheckLanguage)
check_language(Fortran)
if(CMAKE_Fortran_COMPILER)
    enable_language(Fortran)
else()
    message(STATUS "No Fortran support")
endif()
```

若检查成功，则设置相应的 `CMAKE_<LANG>_COMPILER` 变量。若变量是在检查之前设置的，则没有效果。

`CheckCompilerFlag` 提供了 `check_compiler_flag()` 来检查当前编译器是否支持某个标志。在内部，将编译一个非常简单的程序，并解析输出以获得诊断消息。检查编译器支持 `CMAKE_<LANG>_FLAGS` 的标志将成功运行；否则，`check_compiler_flag()` 将失败。下面的例子就在检查 C++ 编译器是否支持-Wall 标志：

```
include(CheckCompilerFlag)
check_compiler_flag(CXX -Wall WALL_FLAG_SUPPORTED)
```

若支持-Wall 标志，`WALL_FLAG_SUPPORTED` 缓存变量为 `true`；否则，为 `false`。

检查链接器标志的相应模块称为 `CheckLinkerFlag`，其工作原理类似于检查编译器标志，但链接器标志不会直接传递给链接器。由于通常会通过编译器调用链接器，因此向链接器传递附加标志可以使用-Wl 或-Xlinker 这样的前缀来告诉编译器将标志传递过去。由于该标志是特定于编译器的，CMake 提供了 `LINKER:` 前缀来自动替代该命令。例如，要向链接器传递标志，来生成关于执行时间和内存消耗的统计信息：

```
include(CheckLinkerFlag)
check_linker_flag(CXX LINKER:-stats LINKER_STATS_FLAG_SUPPORTED)
```

若链接器支持-stats 标志，则 `LINKER_STATS_FLAG_SUPPORTED` 变量则为 `true`。

比较常用的检查模块有 `CheckLibraryExists`、`CheckIncludeFile` 和 `CheckIncludeFileCXX`，用于检查某些库或包含文件是否存在某些位置。

CMake 提供了更详细的检查，可能非常特定于项目；例如，`CheckSymbolExists` 和 `CheckSymbolExistsCXX` 模块检查某个符号是否作为预处理器定义、变量或函数存在。`CheckStructHasMember` 将检查结构是否有某个成员，而 `CheckTypeSize` 可以使用 `CheckPrototypeDefinition` 检查非用户类型的大小，以及 C 和 C++ 函数原型的定义。

CMake 提供了相当多的检查，而且可用的检查列表可能会随着 CMake 的发展而增长。虽然检查在某些情况下有用，但应该注意不要进行过多的测试。检查的数量和复杂性将对配置步骤的速度产生相当大的影响，但有时并没有提供太多的好处。项目中进行大量的检查，也可能暗示项目有着不必要的复杂性。

## 12.5. 总结

支持交叉编译是 CMake 的特点。本章中，我们研究了如何定义用于交叉编译的工具链文件，以及如何使用 sysroot 在不同的目标平台上使用库。交叉编译的特殊情况是 Android 和 Apple 移动设备，它们依赖于特定的 sdk。通过在其他平台上使用模拟器，或在模拟器中进行测试的简单了解，将了解为各种目标平台构建高质量软件所需的所有基本信息。

本章的最后一部分涉及到为某些特性测试工具链的高级主题。虽然大多数项目不需要关注这些细节，但了解一下也没什么错。

下一章将讨论如何使 CMake 代码在多个项目中可重用，而不需要每次重写所有东西。

## 12.6. 练习题

回答以下问题来测试对本章的理解：

1. CMake 如何使用工具链文件？
2. 用于交叉编译的工具链文件中定义了什么？
3. 在 sysroot 的上下文中，暂存目录中有什么？
4. CMake 如何使用模拟器进行测试？
5. 什么触发了编译器特性的检测？
6. 如何存储和恢复编译器检查的配置？
7. CMAKE\_CROSSCOMPILING 对编译器检查有什么影响？
8. 为什么在切换工具链时应该完全清除构建目录，而不仅仅是删除缓存文件？

# 第 13 章 重用 CMake 代码

为项目编写构建系统代码不是一件容易的事。项目维护者和开发人员花费大量精力编写 CMake 代码，以配置编译器标志、项目构建变量、第三方库和工具集成。处理多个 CMake 项目时，从头开始为项目无关的细节编写 CMake 代码可能会带来很大的负担。为项目编写的配置上述细节的大部分 CMake 代码可以在项目之间重用。考虑到这一点，开发一种策略使 CMake 代码可以友好的重用会很有帮助。解决这个问题的直接方法是将 CMake 代码视为常规代码，并应用一些最基本的编码原则：不要重复自己 (DRY) 原则和单责任原则 (SRP)。

若考虑到可重用性，CMake 代码可以很容易地重用。将 CMake 代码分离为模块和函数，使 CMake 代码可重用的方法与使软件代码可重用没有什么不同。CMake 本身毕竟是一种脚本语言，可以很自然地将 CMake 代码视为常规代码，并在处理它时应用软件设计原则。与函数式脚本语言一样，CMake 具有以下重用性的基本能力：

- 包含其他 CMake 文件
- 函数/宏
- 可移植性

本章中，我们将学习为一个项目编写 CMake 代码的方法，并考虑到可重用性，以及在 CMake 项目中重用 CMake 代码。还将讨论版本控制和在项目之间共享通用 CMake 代码的方法。将讨论以下主题：

- 了解 CMake 模块
- 模块的基本构建块——函数和宏
- 编写一个 CMake 模块

## 13.1. 相关准备

深入学习本章之前，可以重新阅读第 1 章，温习之前所学的知识。建议从这里获取本章的示例内容：<https://github.com/PacktPublishing/CMake-Best-Practices/tree/main/chapter13>。对于所有示例，假设您将使用此项目提供的容器运行：<https://github.com/PacktPublishing/CMake-Best-Practices>。

先从学习 CMake 可重用性的基础知识开始吧。

## 13.2. 了解 CMake 模块

CMake 模块包含 CMake 代码、函数和宏，放在一起以服务于特定的目的。模块可以为其他 CMake 代码提供函数和宏，并在包含时执行 CMake 命令。通常，CMake 内置了许多预先制作好的模块。这些模块提供了实用程序来使用 CMake 代码，并允许发现第三方工具和依赖项 (Find\*.cmake 模块)。CMake 默认提供的模块列表可在<https://cmake.org/cmake/help/latest/manual/cmakemodules.7.html> 获得。官方文档将模块分为以下两类：

- 实用工具模块
- 查找模块

实用工具模块提供相应的工具，而查找模块用于搜索系统中的第三方软件。我们已经在第 4 章和第 5 章中详细介绍了查找模块。因此，在本章中专门关注实用工具模块。前面的章节中使用了一些 CMake 提供的实用模块，其中一些模块是 `GNUInstallDirs`、`CPack`、`FetchContent` 和 `ExternalProject`，这些模块位于 CMake 安装文件夹下。

为了更好地理解实用工具模块的概念，先从研究 CMake 提供的实用程序模块开始，先来看看 `ProcessorCount` 实用程序模块。可以在<https://github.com/Kitware/CMake/blob/master/Modules/ProcessorCount.cmake>找到这个模块的源文件。`ProcessorCount` 模块可在 CMake 代码中检索系统的 CPU 核心计数的模块。`ProcessorCount.cmake` 定义了名为 `ProcessorCount` 的函数，其接受名为 `var` 的参数。该函数的实现大致如下：

```
function(ProcessorCount var)
    # Unknown:
    set(count 0)
    if(WIN32)
        set(count "$ENV{NUMBER_OF_PROCESSORS}")
    endif()
    if(NOT count)
        # Mac, FreeBSD, OpenBSD (systems with sysctl):
        # ... mac-specific approach ...
    endif()
    if(NOT count)
        # Linux (systems with nproc):
        # ... linux-specific approach ...
    endif()
    # ... Other platforms, alternative fallback methods ...
    # Lastly:
    set(${var} ${count} PARENT_SCOPE)
endfunction()
```

`ProcessorCount` 函数尝试几种不同的方法来检索主机的 CPU 核心计数。`ProcessorCount` 模块的用法很简单：

```
include(ProcessorCount)
ProcessorCount(CORE_COUNT)
message(STATUS "Core count: ${CORE_COUNT}")
```

使用 CMake 模块非常简单，只需将该模块包含到所需的 CMake 文件中即可。`include()` 具有传递性，此行之后的代码可以使用模块中包含的所有 CMake 定义。

现在我们对实用工具模块有了大致的了解。继续了解更多关于实用程序模块的基本构建模块：函数和宏。

### 13.3. 模块的基本构建块——函数和宏

我们需要一些基本的构建块来创建实用模块。实用工具模块最基本的构建模块是函数和宏，因此很有必要了解它们的工作原理。先从函数开始。

#### 13.3.1 函数

函数是 CMake 语言的特性，可以定义一个逻辑代码块，来执行 CMake 指令。函数以 `function(...)` 开始，函数体包含 CMake 命令，以 `endfunction()` 结束。`function()` 需要一个名称作为第一个参数，以及可选的函数参数名称：

```
function(<name> [<arg1> ...])
  <commands>
endfunction()
```

函数定义了一个新的变量作用域，因此对 CMake 变量所做的更改只在函数体中可见，独立作用域是函数最重要的属性。有了新的作用域就能避免意外地将变量暴露给使用者，除非我们想这样做。大多数时候，我们希望在函数的作用域中包含更改，并且只将函数的结果反映给使用者。由于 CMake 没有返回值的概念，我们将采用在调用者的作用域方法中定义一个变量来将函数结果返回给使用者。

先定义一个简单的函数来检索当前 Git 分支名称：

```
function(git_get_branch_name result_var_name)
  execute_process(
    COMMAND git symbolic-ref -q --short HEAD
    WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    OUTPUT_VARIABLE git_current_branch_name
    OUTPUT_STRIP_TRAILING_WHITESPACE
    ERROR_QUIET
  )
  set(${result_var_name} ${git_current_branch_name}
    PARENT_SCOPE)
endfunction()
```

`git_get_branch_name` 函数接受名为 `result_var_name` 的参数。这个参数是将在使用者的作用域中定义的变量的名称，用于将 Git 分支名称返回给使用者。或者，可以使用一个常量变量名，例如 `GIT_CURRENT_BRANCH_NAME`，并去掉 `result_var_name` 参数，但若项目已经使用了 `GIT_CURRENT_BRANCH_NAME` 名称，这可能会出问题。

经验法则是将命名留给使用者，因为它具有最大的灵活性和可移植性。为了检索当前 Git 分支名称，使用 `execute_process()` 运行了 `git symbolic-ref -q --short HEAD` 命令，其结果存储在函数作用域中的 `git_current_branch_name` 变量中。变量在函数的作用域中，所以使用者不能看到 `git_current_branch_name` 变量。因此，需要使用 `set(${result_var_name})`

`git_get_branch_name()` 来定义一个变量，使用外部作用域中 `result_var_name` 的值和本地 `git_current_branch_name` 变量的值相同。

`PARENT_SCOPE` 参数改变了 `set(...)` 指令的作用域，因此它在使用者的作用域内。`git_get_branch_name` 函数的用法如下：

```
git_get_branch_name(branch_n)
message(STATUS "Current git branch name is: ${branch_n}")
```

接下来看下宏。

### 13.3.2 宏

如果函数的作用域理解起来有困难，可以考虑使用 `macro(...)` 来代替。宏以 `macro(...)` 开始，以 `endmacro()` 结束。函数和宏在各个方面的行为都相似，但有一点不同：宏不定义新的变量作用域。回到 `git` 分支的例子，考虑到 `execute_process(...)` 已经有 `OUTPUT_VARIABLE` 参数，将 `git_get_branch_name` 定义为宏，而不是函数会更方便些：

```
macro(git_get_branch_name_m result_var_name)
  execute_process(
    COMMAND git symbolic-ref -q --short HEAD
    WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    OUTPUT_VARIABLE ${result_var_name}
    OUTPUT_STRIP_TRAILING_WHITESPACE
    ERROR_QUIET
  )
endmacro()
```

`git_get_branch_name_m` 宏的用法与 `git_get_branch_name()` 函数完全相同：

```
git_get_branch_name_m(branch_nn)
message(STATUS "Current git branch name is: ${branch_nn}")
```

我们已经学习了如何在需要时定义函数或宏。接下来，定义一个 CMake 模块。

## 13.4. 编写一个 CMake 模块

上一节中，我们学习了如何使用函数和宏在 CMake 项目中提供有用的功能。现在，来学习如何将这些函数和宏移动到单独的 CMake 模块中。

创建和使用简单的 CMake 模块文件非常简单：

1. 创建一个 `<module_name>.cmake` 文件。
2. `<module_name>.cmake` 文件中定义宏/函数。
3. 将 `<module_name>.cmake` 文件包含在必要的文件当中

按照这些步骤一起创建一个模块，作为上一个 `git` 分支名称示例的后续，我们扩展范围编写一个 CMake 模块，该模块通过使用 `git` 命令提供检索分支名称、HEAD 提交哈希、当前作者名称和当前作

者电子邮件信息的能力。这一部分中，将使用 chapter13/ex01\_git\_utility 的示例。示例文件夹包含一个 CMakeLists.txt 文件和一个 git.cmake 文件，文件存放在 cmake 文件夹下。先来看看 cmake/git.cmake:

```
# ...
include_guard(DIRECTORY)
macro(git_get_branch_name result_var_name)
    execute_process(
        COMMAND git symbolic-ref -q --short HEAD
        WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
        OUTPUT_VARIABLE ${result_var_name}
        OUTPUT_STRIP_TRAILING_WHITESPACE
        ERROR_QUIET
    )
endmacro()
# ... git_get_head_commit_hash(), git_get_config_value()
```

git.cmake 文件是 cmake 实用工具模块文件，包含三个宏，分别名为 git\_get\_branch\_name、git\_get\_head\_commit\_hash 和 git\_get\_config\_value。此外，文件的顶部有 include\_guard(DIRECTORY)。这类似于 C/C++ 中的 #pragma once 预处理器指令，防止文件多次包含。DIRECTORY 参数表示 include\_guard 在目录范围内定义，该文件最多只能在当前目录内或以下包含一次。另外，可以指定 GLOBAL 参数(而不是 DIRECTORY)，以限制只包含一次文件，而不考虑范围。

看看如何使用 git.cmake 模块文件，研究下 chapter13/ex01\_git\_utility 中的 CMakeLists.txt 文件：

```
cmake_minimum_required(VERSION 3.21)
project(
    ch13_ex01_git_module
    VERSION 1.0
    DESCRIPTION "Chapter 13 Example 01, git utility module example"
    LANGUAGES CXX)
# Include the git.cmake module.
# Full relative path is given, since cmake/ is not in the
# CMAKE_MODULE_PATH
include(cmake/git.cmake)
git_get_branch_name(current_branch_name)
git_get_head_commit_hash(current_head)
git_get_config_value("user.name" current_user_name)
git_get_config_value("user.email" current_user_email)

message STATUS "-----")
message STATUS "VCS (git) info:")
message STATUS "\tBranch: ${current_branch_name}")
message STATUS "\tCommit hash: ${current_head}")
```

```
message(STATUS "\tAuthor name: ${current_user_name}")
message(STATUS "\tAuthor e-mail: ${current_user_email}")
message(STATUS "-----")
```

通过指定模块文件的完整相对路径，CMakeLists.txt 文件包含 git.cmake 文件。模块提供的 git\_get\_branch\_name、git\_get\_head\_commit\_hash 和 git\_get\_config\_value 宏分别用于检索分支名称、提交哈希、作者名和电子邮件到 current\_branch\_name、current\_head、current\_user\_name 和 current\_user\_email。最后，通过 message(… ) 指令将这些变量打印在屏幕上。配置示例项目，看看刚编写的 git 模块是否能如预期的那样工作：

```
cd chapter13/ex01_git_utility/
cmake -S ./ -B ./build
```

命令的输出应该如下所示：

```
-- The CXX compiler identification is GNU 9.4.0
-- 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
-----
-- VCS (git) info:
-- Branch: chapter-development/chapter13
-- Commit hash: 1d5a32649e74e4132e7b66292ab23aaeed327fdc
-- Author name: Mustafa Kemal GIJOR
-- Author e-mail: mustafagilor@gmail.com
-----
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/toor/workspace/CMake-Best-Practices/chapter13
/ex01_git_utility/build
```

正如我们所看到的，已经成功地从 git 命令中检索了信息。我们的第一个 CMake 模块工作正常。

### 13.4.1 案例研究——处理项目元数据文件

假设有一个每行包含键值对的环境文件，项目中包含一些关于项目的元数据（例如，项目版本和依赖项）的外部文件并不罕见。该文件可以是不同的格式，例如：JSON 或换行分隔的“键-值”对。当前的任务是创建一个实用工具模块，该模块读取环境变量文件，并在文件中为每个“键-值”对定义一个 CMake 变量。文件的内容如下所示：

```
KEY1="Value1"  
KEY2="Value2"
```

本节中，将遵循 chapter13/ex02\_envfile\_utility 的示例。从 cmake/envfile-utils.cmake 开始：

```
include_guard(DIRECTORY)  
function(read_environment_file ENVIRONMENT_FILE_NAME)  
    file(STRINGS ${ENVIRONMENT_FILE_NAME} KVP_LIST  
        ENCODING  
        UTF-8)  
  
    foreach(ENV_VAR_DECL IN LISTS KVP_LIST)  
        string(STRIPE ENV_VAR_DECL ${ENV_VAR_DECL})  
        string(LENGTH ENV_VAR_DECL ENV_VAR_DECL_LEN)  
        if(ENV_VAR_DECL_LEN EQUAL 0)  
            continue()  
        endif()  
        string(SUBSTRING ${ENV_VAR_DECL} 0 1  
            ENV_VAR_DECL_FC)  
        if(ENV_VAR_DECL_FC STREQUAL "#")  
            continue()  
        endif()  
        string(REPLACE "=" ";" ENV_VAR_SPLIT  
            ${ENV_VAR_DECL})  
        list(GET ENV_VAR_SPLIT 0 ENV_VAR_NAME)  
        list(GET ENV_VAR_SPLIT 1 ENV_VAR_VALUE)  
        string(REPLACE "\\" "" ENV_VAR_VALUE  
            ${ENV_VAR_VALUE})  
        set(${ENV_VAR_NAME} ${ENV_VAR_VALUE} PARENT_SCOPE)  
    endforeach()  
endfunction()
```

envfile-utils.cmake 实用工具模块包含函数 read\_environment\_file，以键值对列表的格式读取一个环境文件。这个函数将文件中的所有行读入 KVP\_LIST 变量，然后遍历所有行。每一行都用 (=) 等号标记分隔，然后等号的左侧作为变量名，而右侧作为变量值，将每个“键-值”对定义为 CMake 变量，空行和注释行将跳过。关于模块的使用，来看看 chapter13/ex02\_envfile\_utility/CMakeLists.txt：

```
# Add cmake folder to the module path, so subsequent
```

```

# include() calls
# can directly include modules under cmake/ folder by
# specifying the name only.
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
    ${PROJECT_SOURCE_DIR}/cmake/)
add_subdirectory(test-executable)

```

可能已经注意到，cmake 文件夹现在添加到 CMAKE\_MODULE\_PATH 变量中，该变量是 include(...) 指令将搜索的路径集合，默认情况下是空。所以现在可以直接在当前和子 CMakeLists.txt 中使用名称包含 envfile-utils 模块。最后，看看 chapter13/ex02\_envfile\_utility/test-executable/CMakeLists.txt:

```

# ....
# Include the module by name
include(envfile-utils)
read_environment_file("${PROJECT_SOURCE_DIR}/variables.env")
add_executable(ch13_ex02_envfile_utility_test)
target_sources(ch13_ex02_envfile_utility_test
PRIVATE test.cpp)
target_compile_features(ch13_ex02_envfile_utility_test
PRIVATE cxx_std_11)
target_compile_definitions(ch13_ex02_envfile_utility_test
PRIVATE PROJECT_VERSION="${TEST_PROJECT_VERSION}"
PROJECT_AUTHOR="${TEST_PROJECT_AUTHOR}")

```

可以按名称包含 envfile-utils 环境文件读取器模块，因为包含 envfile-utils.cmake 的文件夹已经添加到 CMAKE\_MODULE\_PATH 变量中。read\_environment\_file() 函数用来读取同文件夹下的 variables.env 文件，包含以下键值对：

```

# This file contains some metadata about the project
TEST_PROJECT_VERSION="1.0.2"
TEST_PROJECT_AUTHOR="CBP Authors"

```

因此，使用 read\_environment\_file() 之后，期望 TEST\_PROJECT\_VERSION 和 TEST\_PROJECT\_AUTHOR 变量在当前的 CMake 作用域中定义，并在文件中指定各自的值。为验证这一点，定义了一个名为 ch13\_ex02\_envfile\_utility\_test 的可执行目标，并将 TEST\_PROJECT\_VERSION 和 TEST\_PROJECT\_AUTHOR 变量作为宏定义传递给目标。最后，目标的源文件 test.cpp 将 TEST\_PROJECT\_VERSION 和 TEST\_PROJECT\_AUTHOR 宏定义打印到控制台：

```

1 #include <cstdio>
2 int main(void) {
3     std::printf("Version '%s', author '%s'\n",
4     TEST_PROJECT_VERSION, TEST_PROJECT_AUTHOR);

```

好了，让我们编译并运行应用程序：

```
cd chapter13/ex02_envfile_utility
cmake -S ./ -B ./build
cmake --build build
./build/test-executable/ch13_ex02_envfile_utility_test
# Will output: Version '1.0.2', author 'CBP Authors'
```

我们已经成功地从源树中读取了键值对格式的文件，将每个键值对定义为 CMake 变量，然后将这些变量作为宏定义公开给应用程序。

尽管编写 CMake 模块非常简单，但这里有一些建议：

- 为函数/宏使用唯一的名称。
- 为所有模块函数/宏使用共同的前缀。
- 避免对非函数作用域变量使用常量名。
- 为模块使用 `include_guard()`。
- 若模块需要打印消息，请为模块提供静默模式。
- 不要暴露模块的内部。
- 为简单的命令包装器使用宏，其他的都使用函数。

接下来，我们将探讨在项目之间共享 CMake 模块的方法。

#### 13.4.2 项目间共享 CMake 模块的建议

共享 CMake 模块的推荐方法是为 CMake 模块维护一个单独的项目，然后将该项目作为外部资源合并，可以直接通过 Git 子模块/子树或 CMake 的 FetchContent/ExternalProject。这样，所有可重用的 CMake 实用程序都可以维护在单个项目下，并可以传播到所有下游项目。将 CMake 模块放入在线 Git 托管平台（如 GitHub 或 GitLab）的存储库中，将使大多数人使用该模块更加方便。因为 CMake 支持直接从 Git 获取内容，所以使用共享库将非常简单。

为了演示如何使用外部 CMake 模块项目，我们将使用名为 Hadouken 的开源 CMake 实用模块项目。该项目可从<https://github.com/mustafakemalgi/or/hadouken>访问。该项目包含用于工具集成、目标创建和特性检查的 CMake 实用工具模块。

这里，使用 chapter13/ex03\_hadouken 的例子。用这个例子获取 Hadouken，然后使用 Hadouken 项目的目标创建助手实用程序。先来看看 `CMakeLists.txt`：

```
cmake_minimum_required(VERSION 3.21)
project(
    ch13_ex03_hadouken
    VERSION 1.0
    DESCRIPTION
```

```

"Chapter 13 Example 03, external CMake modules (hadouken) example"
LANGUAGES CXX)
include(FetchContent)
# Declare hadouken dependency details.
FetchContent_Declare(hadouken
    GIT_REPOSITORY https://github.com/mustafakemalgilor/hadouken.git
    GIT_TAG 7d0447fcadf8e93d25f242b9bb251ecbcf67f8cb
    SOURCE_DIR "${CMAKE_CURRENT_LIST_DIR}/.hadouken"
)
FetchContent_MakeAvailable(hadouken)
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
    ${PROJECT_SOURCE_DIR}/.hadouken/cmake/modules/)
include(misc/Log)
include(misc/Utility)
include(core/MakeCompilationUnit)
include(core/MakeTarget)
# Create an executable target by using Hadouken's
# make_target() utility function
make_target(TYPE EXECUTABLE)

```

前面的例子中，使用了 `FetchContent_Declare` 和 `FetchContent_MakeAvailable` 将 Hadouken 检索到项目中。然后，Hadouken 项目的模块目录添加到 `CMAKE_MODULE_PATH` 中，通过 `include(...)` 指令使用 Hadouken 项目的 CMake 实用模块，从而包含了 Log、Utility、`MakeCompilationUnit` 和 `MakeTarget` 模块。最后，使用 `make_target()` 函数确保我们可以使用外部 CMake 模块项目的函数。

`make_target(...)` 函数是由 Hadouken 项目的核心/`MakeTarget` 模块提供的，是 `add_executable` 和 `add_library` 指令的包装函数。`make_target(TYPE EXECUTABLE)` 将包括 `src/` 文件夹下的所有源文件，并通过 `add_executable(...)` 创建一个可执行目标。让我们来配置和构建项目，看看情况是否如此：

```

cd chapter13/ex03_hadouken
cmake -S ./ -B build/
cmake --build build

```

输出应该如下所示：

```
[ 50%] Building CXX object  
CMakeFiles/ch13_ex03_hadouken.dir/src/main.cpp.o  
[100%] Linking CXX executable ch13_ex03_hadouken  
[100%] Built target ch13_ex03_hadouken
```

定义了目标 ch13\_ex03\_hadouken，并将源文件 main.cpp 作为源文件包含在目标中。证实了我们可以在 CMake 代码中使用外部 CMake 模块项目。

接下来，我们将总结本章学到的知识，以及下一章将学到的知识。

## 13.5. 总结

本章中，学习了如何构建 CMake 项目以支持可重用性。学习了如何实现 CMake 实用程序模块，如何共享，以及如何使用他人编写的实用程序模块。有了利用 CMake 模块的能力，可以更好地组织我们的项目，更好地与团队成员协作。有了这些知识，CMake 项目将更容易维护。CMake 项目之间的通用的、可重用的代码将成长为一个有用的模块集合，使 CMake 编写项目更容易。

这里我想说明的是，CMake 是一种脚本语言，应该这样使用，使用软件设计原则和模式管理 CMake 代码会使其更易于维护。将 CMake 代码组织成函数和模块，尽可能多地重用和共享 CMake 代码。请不要忽略构建系统的代码，否则可能需要从头开始编写。

下一章中，我们将学习优化和维护 CMake 项目的方法。

## 13.6. 练习题

回答以下问题来测试对本章的理解：

1. CMake 中可重用的最基本构建块是什么？
2. CMake 模块是什么？
3. 如何使用 CMake 模块？
4. CMAKE\_MODULE\_PATH 是做什么的？
5. 说出一种在项目之间共享 CMake 模块的方法。
6. CMake 中函数和宏的主要区别是什么？

# 第 14 章 优化和维护 CMake 项目

软件项目会有一定的生命周期，对于一些项目来说，在 10 年或更长时间内处于活跃的开发也有可能。但即使项目没有那么长的生命，也会随着时间出现混乱和难以维护的部分。通常，维护一个项目不仅是重构代码或添加特性，这需要保持构建信息和依赖关系的更新。

随着项目变得越来越复杂，构建时间通常也会增加，以至于开发可能会因为漫长的等待而变得乏味。长时间的构建不仅不方便，还可能迫使开发人员走捷径，会让尝试使用项目变得困难。若每次构建都需要数小时才能完成，若每次推送到 CI/CD 流水中都需要数小时才能返回，那么就很难尝试新的东西。

除了选择一个好的、模块化的项目结构来提高增量构建的效率外，CMake 还有一些特性来帮助分析和优化构建时间。若使用 CMake 还不够，那么使用诸如编译器缓存 (ccache) 等技术来缓存构建结果，或预编译头文件可以进一步帮助加快增量构建。

优化构建时间可以产生良好的结果，极大地改善开发人员的日常工作，甚至是可以节省很多成本，因为 CI/CD 流水可能需要更少的资源来构建项目。然而，过度优化的系统可能变得脆弱和易崩溃，并且优化构建时间可能需要与易维护程度进行的权衡。

本章中，将介绍一些维护项目的技巧，以及如何组织项目以控制维护工作。然后，我们将深入分析构建性能，并了解如何加快构建速度。

本章将讨论以下内容：

- 保持 CMake 项目的可维护性
- 对 CMake 构建进行性能分析
- 优化构建性能

## 14.1. 相关准备

和前面的章节一样，所有的例子都已经用 CMake 3.21 测试过了，并可以由以下编译器编译：

- GCC 9 或更高版本
- Clang 12 或更高版本
- MSVC 19 或更高版本

为了查看分析数据，需要一个 Google 跟踪格式的查看器，可以使用 Google Chrome 浏览器。

使用 ccache 的例子用 Clang 和 GCC 测试，而不是用 MSVC。要获得 ccache，使用操作系统的包管理器或从<https://ccache.dev/>获取。

例子可以在<https://github.com/PacktPublishing/CMake-Best-Practices>上找到。

## 14.2. 保持 CMake 项目的可维护性

长期维护 CMake 项目时，经常会有一些任务定期出现，比如：向项目中添加新文件或增加依赖项的版本，这些相对容易处理；然后，是添加新的工具链或用于交叉编译的平台；最后，当要使用新功能（如预设）时，需要更新 CMake。

定期更新 CMake，并利用新特性可以保持项目的可维护性。虽然更新每一个新版本通常不现实，但检查 CMake 的新特性，并在它们发布时使用它们可能会使项目更容易维护。例如，CMake 3.19 版本中引入的 CMake 预设就是这样一个特性，可能使许多复杂的 CMakeLists.txt 文件变得更简单。

保持依赖关系处于最新状态，并处于控制之下，通常是维护者需要完成的任务。这里，使用一致的概念来处理依赖将使项目维护更容易。除了小项目，我们推荐使用第 5 章中介绍的包管理器。由于包管理器的设计目的是将管理依赖关系的复杂性转移给包管理器，而不是将其暴露给维护者，因此会使维护者的工作更加轻松。

使项目具有可维护性的根本是有效的项目结构，这样就可以很容易找到相应的东西，并彼此独立地进行改进。选择的结构在很大程度上取决于项目的环境和规模，因此适用于一个项目的方法可能并不适用于另一个项目。

保持大型项目可维护性的最大好处是使用合适的项目结构。虽然项目组织的细节取决于开发项目的实际情况，但良好的实践有助于保持对项目的描述。保持项目的可维护性，可以从项目的主 CMakeLists.txt 文件开始。对于大型项目，主 CMakeLists.txt 应该处理以下事情：

- 整个项目的基本设置，例如使用 `project()`、获取工具链、支持程序和辅助库。这还包括设置语言标准、搜索行为，以及在项目范围内设置编译器标志和搜索路径。
- 处理横向依赖关系，特别是像 Boost 和 Qt 这样的大型框架。根据依赖关系的复杂性，创建并包含一个子目录，通过其自带的 CMakeLists.txt 来处理获取依赖关系有助于保持项目的可维护性。我们推荐使用 `add_subdirectory`，这样搜索依赖项的临时变量的作用域都限于子目录，除非显式地标记为缓存变量。
- 若构建目标不止几个，可将它们移到各自的子目录中，并用 `add_subdirectory()` 将它们包含进来，这将有助于保持各个文件短小且可以自包含。以松耦合和高内聚为设计原则，将使库和可执行程序更容易独立维护，可能每个库和可执行项目都要有自己的 CMakeLists.txt。
- 单元测试保持在所测试的单元附近，还是作为根级别测试文件夹的子文件夹，取决于个人喜好。将测试放在独立的子目录中，带有自己的 CMakeLists.txt，这样可以更容易地处理特定于测试的依赖项和编译器设置。
- 项目的打包和安装说明应该集中在项目的顶层。若安装说明和包装说明太大，可以将它们放在自己的 CMakeLists.txt 中，并在主 CMakeLists.txt 中进行包含。

以这种方式组织项目将简化项目中的导航，并有助于避免 CMake 文件中不必要的代码重复，特别是当项目随着时间的推移而变大时。

好的项目设置可能会决定是每天都和构建系统打交道，还可以平稳地运行。使用本书中的技术和实践将有助于 CMake 项目的可维护性。如第 9 章和第 12 章中所述，通过使用 CMake 预置和构建容器或 sysroot，有一个明确定义的构建环境，将有助于使构建在开发人员和 CI 系统之间更具可移植性。最后，将自定义 CMake 代码组织成宏和函数，如第 13 章所述，这将有助于避免冗余和重复。

除了 CMake 文件的复杂性，当项目变大时，更长的配置和构建时间通常是另一个需要考虑的问题。为了管理不断增长的构建和配置时间，CMake 提供了一些特性来进行优化。

### 14.3. 对 CMake 构建进行性能分析

CMake 项目变大时，配置可能会花费相当长的时间，特别是当加载了外部内容或需要对工具链特性进行大量检查时。优化的第一步是检查配置过程的哪一部分占用了多少时间。从 3.18 版本开始，CMake 包含了命令行选项来生成漂亮的分析图，以调查配置过程中花费的时间。通过使用`--profiling-output` 和`--profiling-format` 分析标志，CMake 将创建分析输出。编写本书时，仅支持输出格式的 Google 跟踪格式，但还是需要指定格式和文件来创建分析信息。调用 CMake 来创建一个概要图可以这样操作：

```
cmake -S <sourceDir> -B <buildDir> --profiling-output  
./profiling.json --profiling-format=google-trace
```

当前目录下，将分析数据输出到 `profiling.json` 文件中。通过在地址栏中输入 `about://tracing`，可以使用 Google Chrome 查看输出文件。本书缓存的 GitHub 项目的输出如下所示：

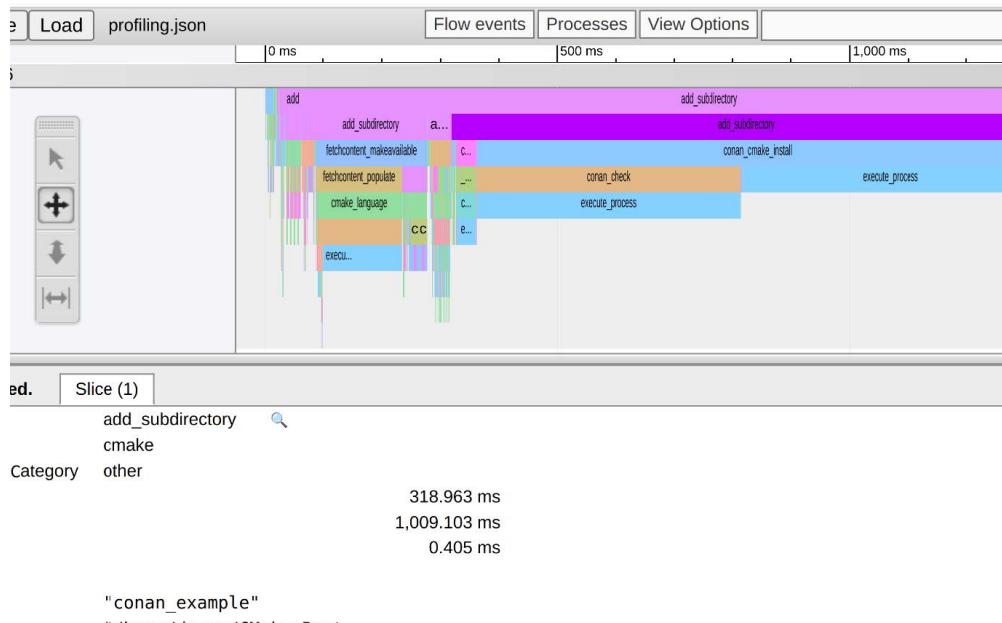


图 14.1 CMake 项目在 Google Chrome 中的概要图示例

配置项目时，有个 `add_subdirectory` 占用了大部分时间。本例中，这是 `chapter05` 子目录，需要 3 秒多一点的时间来完成。深入研究一下，就会发现这些例子都使用了 Conan 包管理器，即两次调用 `conan_cmake_install` 使得配置的开销相对较大，将对 Conan 的调用集中到一个更高的目录中，可以将 CMake 的配置运行时间减少一半。

为了正确地解释分析输出，有助于相互比较不同的 CMake 运行情况，特别是将 CMake 运行在干净的缓存上与使用缓存信息的缓存上进行比较。若 CMake 只在干净的缓存上运行，但增量运行足够快，这对开发人员来说仍然可以接受。然而，若增量 CMake 运行也需要很久，这可能会造成更大的问题。分析它们可以确定每次运行配置时是否执行了不必要的步骤。

修复缓慢的构建步骤取决于具体的情况，但配置时间过长的常见原因是每次都要下载文件，因为一开始没有检查文件是否存在。分析可能经常会发现，像 `execute_process` 或 `try_compile`

这样的指令占用了大量的执行时间。最简单的“解决方法”是尝试摆脱这些调用，但这些调用的存在往往是有原因的。通常情况下，跟踪指令的堆栈会找到降低调用这些函数频率的方法。结果进行缓存，或者用 `execute_process` 创建的文件不需要每次都生成。

特别是在交叉编译时，`find` 指令可能也会占用大量时间。如第 5 章所述，通过改变 `CMAKE_FIND_ROOT_PATH_MODE_*` 变量来改变搜索顺序，可能会有所帮助。要更彻底地分析为什么 `find` 会占用太多时间，可以通过将 `CMAKE_FIND_DEBUG_MODE` 变量设置为 `true`，来告诉 CMake 启用相应的调试输出。由于这将打印出很多信息，因此最好只对某些调用启用此选项，如下所示：

```
set(CMAKE_FIND_DEBUG_MODE TRUE)
find_package(...)
set(CMAKE_FIND_DEBUG_MODE FALSE)
```

CMake 的配置选项允许对构建过程的配置阶段进行配置，必须使用相应的生成器来分析实际的编译和时间。大多数生成器要么支持一些分析选项，要么记录所需的信息。对于 Visual Studio 生成器，`vcperf` 工具 (<https://github.com/microsoft/vcperf>) 将提供很多信息。使用 Ninja 时，可以使用 `ninjatracing` 工具 (<https://github.com/nico/ninjatracing>) 将 `.ninja_log` 文件转换为 Google 跟踪格式。虽然 CMake 不支持分析软件的实际编译和链接，但它提供了改进构建时间的方法，我们将在下一节中看到。

## 14.4. 优化构建性能

除了编译时间，C++ 项目中导致长构建时间的主要原因，通常是目标或文件之间不必要的依赖关系。若目标之间有不必要的链接要求，则构建系统将限制并行执行构建任务，并且目标将频繁地重新链接。如第 6 章所述，创建目标的依赖关系图有助于识别依赖关系。若生成的图看起来更像一条绳子，而不是一棵树，那么优化和重构项目结构可能会带来很多性能提升。第 7 章将代码质量工具与 CMake 无缝集成，介绍了包含所使用的工具并链接所使用的工具，这类工具可以帮助识别不必要的依赖。另一个常见的问题是 C 或 C++ 项目在公共头文件中暴露了太多的私有信息，经常导致频繁的重构并降低增量构建的有效性。

一个相对安全提高性能的选项是将 `CMAKE_OPTIMIZE_DEPENDENCIES` 缓存变量设置为 `true`，这将使 CMake 在生成时删除一些不需要的静态或对象库的依赖项。若使用大量静态或对象库以及深度依赖图，这可能会在编译时间方面产生了一些收益。

一般来说，优化项目结构和代码模块化，通常比优化代码对构建性能的影响更大。编译和链接一个由许多小文件组成的项目，要比由几个大文件组成的项目花费更多的时间。CMake 可以通过统一构建帮助提高构建性能，将几个文件合并为一个更大的文件。

### 14.4.1 统一构建

CMake 的统一构建，可以通过将多个文件连接成更大的文件来提高构建性能，从而减少需要编译的文件数量。因为包含文件只处理一次，而不是对每个小文件都进行处理，所以这可能会减少构建时间。因此，若许多文件包含相同的头文件，并且编译器难以消化头文件，这将产生很大的效果。通常，这些头文件包含大量宏或模板元编程。创建统一构建可以显著提高构建时间，特别是当使用

大型头文件库时，如数学特征库。另一方面，统一构建也有缺点，即增量构建可能需要更长的时间。当只有单个文件发生更改时，项目才需要重新编译和链接。

通过将 CMAKE\_UNITY\_BUILD 缓存变量设置为 true，CMake 将把源合并为一个或多个统一源并构建它们。生成的文件使用 `unity_<lang>_<Nr>.<lang>` 模式和位于项目构建目录下名为 Unity 的文件夹中。C++ 的 Unity 文件将命名为 `unity_0_cxx.cxx`、`unity_1_cxx.cxx` 等，C 文件命名为 `unity_0_c.c` 等。这个变量不需要在 `CMakeLists.txt` 中设置，而是通过命令行或预设传递，因为它可能取决于是否需要统一构建。若需要并且可能合并文件，CMake 将决定项目的语言。例如，由于头文件没有编译，其不会添加到统一源中。对于 C 和 C++，这没问题；对于其他语言，这可能没什么用。

统一构建最适合由许多小文件组成的项目，若源文件本身已经很大，那么在编译时统一构建可能会出现内存耗尽的风险。若只有少数文件在这方面有问题，可以通过设置 `SKIP_UNITY_BUILD_INCLUSION` 属性的源文件从统一构建中排除：

```
target_sources(ch14_unity_build PRIVATE
    src/main.cpp
    src/fibonacci.cpp
    src/eratosthenes.cpp
)

set_source_files_properties(src/eratosthenes.cpp PROPERTIES
    SKIP_UNITY_BUILD_INCLUSION YES)
```

这个例子中，`eratosthenes.cpp` 文件将从统一构建中排除，而 `main.cpp` 和 `fibonacci.cpp` 将包含在单个编译单元中。若前面的项目已经配置，`unit_0_cxx.cxx` 文件将包含如下内容：

```
1 /* generated by CMake */
2
3 #include "/chapter14/unity_build/src/main.cpp"
4 #include "/chapter14/unity_build/src/fibonacci.cpp"
```

注意，原始源文件只包含在统一文件中，而不是复制到文件中。

自 CMake 3.18 起，统一构建支持两种模式，由 `CMAKE_UNITY_BUILD_MODE` 或 `UNITY_BUILD_MODE` 目标属性控制。该模式可以是 `BATCH` 或 `GROUP`，默认为 `BATCH`。`BATCH` 模式中，CMake 根据添加到目标中的顺序，确定默认情况下将哪些文件分组在一起。目标中的所有文件都将分配到统一批文件中，除非明确地进行排除。`GROUP` 模式中，每个目标必须显式指定如何将文件分组在一起，未分配给组的文件将单独编译。虽然组模式提供了更精确的控制，但使用 `BATCH` 模式通常是首选的模式，因为它降低了维护成本。

默认情况下，当 `UNITY_BUILD_MODE` 属性设置为 `BATCH` 时，CMake 将批量收集 8 个文件。通过设置目标的 `UNITY_BUILD_BATCH_SIZE` 属性进行修改。要进行全局的批处理大小设置，可以使用 `CMAKE_UNITY_BUILD_BATCH_SIZE` 缓存变量。批处理大小应该谨慎选择，若设置得过低不会带来什么性能提升，而设置得太高可能会导致编译器使用过多内存或编译单元触发某些限制。若批处理大小设置为 0，那么目标的所有文件将合并到一个批处理中，但是由于前面提到的原因，不建议这样做。

GROUP 模式下，不应用批处理大小，但文件必须通过设置源文件的 UNITY\_GROUP 属性分配给组：

```
add_executable(ch14_unity_build_group)

target_sources(ch14_unity_build_group PRIVATE
    src/main.cpp
    src/fibonacci.cpp
    src/eratosthenes.cpp
    src/pythagoras.cpp
)
set_target_properties(ch14_unity_build_group PROPERTIES
    UNITY_BUILD_MODE GROUP)
set_source_files_properties(src/main.cpp src/fibonacci.cpp
    PROPERTIES UNITY_GROUP group1)
set_source_files_properties(src/eratosthenes.cpp
    src/pythagoras.cpp PROPERTIES UNITY_GROUP group2)
```

本例中，main.cpp 和 fibonacci.cpp 文件将分在一起，eratosthenes.cpp 和 pythagoras.cpp 将在不同的组中进行编译。GROUP 模式下，生成的文件命名为 unity\_<groupName>\_<lang>.<lang>。本例中，文件将命名为 unity\_group1\_cxx.cxx 和 unity\_group2\_cxx.cxx。

根据项目的结构，使用统一构建可以对构建性能产生显著影响。另一种经常用于提高构建时间的技术是预编译头文件。

#### 14.4.2 预编译头文件

预编译头文件可以显著提高编译时间，特别是在处理头文件是编译时间的重要组成部分，或头文件包含在许多不同编译单元中的情况下。预编译头文件是通过将一些头文件编译成二进制格式来工作的，这种二进制格式对编译器来说更容易处理。从 CMake 3.16 开始，直接支持预编译头文件，并且大多数主要编译器都支持某种形式的预编译头文件。

可以使用 target\_precompile\_headers 指令将预编译头文件添加到目标文件中：

```
target_precompile_headers(<target>
    <INTERFACE|PUBLIC|PRIVATE> [header1...]
    [<INTERFACE|PUBLIC|PRIVATE> [header2...] ...])
```

PRIVATE、PUBLIC 和 INTERFACE 关键字具有通常的含义。大多数情况下，应该使用 PRIVATE。指令中指定的头文件将收集在编译文件夹中 cmake\_pch.h 或 cmake\_pch.hxx 文件中，通过相应的编译器标志强制包含在所有源文件中，因此源文件不需要 #include "cmake\_pch.h"。

头文件可以指定为普通文件名，用尖括号，或者用双引号，这里必须用双方括号进行转义：

```
target_precompile_headers(SomeTarget PRIVATE myHeader.h
    [[ "external_header.h" ]])
```

```
<unordered_map>
)
```

本例中，将从当前源目录中搜索 myHeader.h，而在 include 目录中搜索 external\_header.h 和 unordered\_map 头文件。

大型项目中，多个目标中使用的预编译头文件很常见。可以使用 target\_precompile\_headers 的 REUSE\_FROM 选项，而不是每次都重新定义：

```
target_precompile_headers(<target> REUSE_FROM
<other_target>)
```

重用预编译的头文件会自动从 target 引入对 other\_target 的依赖。两个目标都将启用相同的编译器选项、标志和定义。若不是这样，有些编译器就会发出警告。

只有在当前目标没有定义自己的预编译头文件集时，才能使用来自另一个目标的预编译头文件。若目标已经定义了预编译的头文件，CMake 将报错。

预编译头文件在包含的头文件很少更改时，对缩短构建时间非常有效。编译器、系统或外部依赖项提供的头文件通常都可以包含在预编译头文件中。哪些头文件带来的好处最大，这需要尝试和衡量。

与统一构建一起，预编译头文件可以显著提高编译时间，特别是对于头文件频繁重用的项目。优化增量构建时间的第三种方法是使用编译器缓存，即 ccache。

#### 14.4.3 编译器缓存 (ccache) 加速重建

ccache 的工作原理是缓存编译，并检测何时再次完成相同的编译。撰写本书时，缓存编译结果的最流行的程序是 ccache，它是在 LGPL 3 下发布的开源程序，还不支持 Visual Studio。只要在两次运行之间不删除缓存，ccache 不仅会进行增量构建，还会影响新的构建。创建的缓存在相同编译器系统间可移植，并且可以存储在远程数据库中，这样多个开发人员可以访问相同的缓存。ccache 官方支持 GCC、Clang 和 NVCC，但也有人声称已经在 MSVC 和 Intel 编译器上运行过。将 ccache 与 CMake 一起使用时，它与 Makefile 和 Ninja 生成器一起使用的效果最好。

要在 CMake 中使用 ccache，需要使用 CMAKE\_<LANG>\_COMPILER\_LAUNCHER 缓存变量，其中 <LANG> 可替换为相应的编程语言。推荐的方法是使用预设值，但是要在 CMakeLists.txt 中为 C 和 C++ 启用 ccache，可以使用以下代码：

```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
  set(CMAKE_C_COMPILER_LAUNCHER ${CCACHE_PROGRAM})
  set(CMAKE_CXX_COMPILER_LAUNCHER ${CCACHE_PROGRAM})
endif()
```

从预设值或命令行传递变量也是很好的替代方法，使用环境变量完成 ccache 的配置。

使用默认配置的 ccache 可能已经在构建时间方面带来了相当大的改进，但若构建稍微复杂一点，可能需要进一步配置。要配置 ccache，可以使用以 CCACHE\_开头的某些环境变量；有关所有配置选项的完整文档，请参阅 ccache 文档。需要特别注意的常见场景包括：将 ccache 与预编译头相结

合，使用 `FetchContent` 管理包含的依赖项，以及将 `ccache` 与其他编译器包装器相结合，例如用于分布式构建的 `distcc` 或 `icecc`。对于这些场景，需要使用以下环境变量：

- 为了使用预编译头文件，可将 `CCACHE_SLOPPINESS` 设置为 `pchDefines,time_macros`。这样，`ccache` 不能检测到预编译头文件中 `#define` 的更改，也不能判断在创建预编译头文件时是否使用了 `_TIME_`、`_DATE_` 或 `_TIMESTAMP_`。或将 `include_file_mtime` 设置到 `CCACHE_SLOPPINESS` 中，可能会进一步提高缓存命中，但也会带来可以忽略的条件竞争。
- 当从源代码构建的大型依赖时，例如：通过 `FetchContent`，将 `CCACHE_BASEDIR` 设置为 `CMAKE_BINARY_DIR` 可能会提高缓存命中率，这可能会带来性能的提升，特别是当有许多(子)项目具有相同的依赖项时。另一方面，若项目本身的源代码需要花费更多的时间来编译，则将其设置为 `CMAKE_SOURCE_DIR` 可能会带来更好的结果。但这也需要试验，以了解哪种方法能带来更好的结果。
- 为了与其他编译器包装器一起工作，使用 `CCACHE_PREFIX` 环境变量可以为其他编译器包装器添加命令在链接多个包装器时，建议首先使用 `ccache`，这样其他包装器的结果也可以缓存。

推荐的做法是使用第 9 章中描述的配置预设值，将环境变量传递给 CMake。这可以与检测 `CMakeLists.txt` 中的 `ccache` 相结合，也可以使用以下预设来传递 `ccache` 命令：

```
{
  "name" : "ccache-env",
  ...
  "environment": {
    "CCACHE_BASEDIR" : "${sourceDir}",
    "CCACHE_SLOPPINESS" : "pchDefines,time_macros"
  }
},
```

使用这些配置，`ccache` 可以在编译时间方面带来非常大的好处，但缓存编译器结果是一个挺复杂的问题，因此要获得全部好处，应该参考 `ccache` 的文档。使用 `ccache` 可能通过相对简单的设置，就能带来最大的性能优势。其他工具，如用于分布式构建的 `distcc`，从 CMake 的角度来看工作非常类似，但需要更多的设置工作。

#### 14.4.4 分布式构建

分布式构建，通过将部分编译转移到网络上的不同机器来工作。这需要设置可以接受连接的服务器，然后配置客户机，使其能够连接到这些服务器。使用以下命令为 `distcc` 设置服务器：

```
distccd --daemon --allow client1 client2
```

`client1` 和 `client2` 是各自构建服务器的主机名或 IP 地址。客户端通过将 `CMAKE_<LANG>_COMPILER_LAUNCHER` 设置为 `distcc`，配置 CMake 使用 `distcc` 将类似于 `ccache`。服务器列表可以通过配置文件配置，也可以通过 `DISTCC_HOSTS` 环境变量配置。与 `ccache`

配置不同，这是特定于主机的，因此配置应该放在用户预设中，而不是特定于项目的预设中。各自的预设可能看起来像这样：

```
{  
    "name" : "distcc-env",  
    ...  
    "environment": {  
        "DISTCC_HOSTS" : "localhost buildsvr1,cpp,lzo  
host123,cpp,lzo"  
    }  
},
```

注意 buildsvr1 主机后面的 cpp 后缀。这将使 distcc 进入所谓的“泵模式”，这种模式通过将预处理分配到服务器来进一步提高编译速度。lzo 后缀说明 distcc 使用的是压缩通信。

分布式构建的缺点是，为了获得速度优势，网络必须足够快，否则传输编译信息的成本可能高于减少的构建时间。在大多数本地网络中，这种情况很容易发生。若机器在处理器架构、编译器和操作系统方面相似，那么分布式构建可以很好地工作。虽然可以使用 distcc 进行交叉编译，但要进行设置可能需要相当多的工作。通过结合良好的编码实践，在大型项目中工作的预编译头文件和编译器缓存仍然可以工作，而不必为每次构建等待几分钟。

## 14.5. 总结

本章中，讨论了一些构造和维护 CMake 项目的一般技巧，特别是大型项目。随着项目规模的增加，配置和构建时间通常会增加，这可能会成为开发人员工作中的障碍。我们研究了 CMake 分析特性如何成为发现配置过程中性能消耗的有用工具，尽管不能用于分析编译本身。

为了帮助解决长时间的编译问题，展示了如何使用统一构建和 CMake 的预编译头文件来改善编译时间。若这些没有带来满意的效果，可以使用编译器缓存（如 ccache）或分布式编译器（如 distcc），为编译器命令添加前缀。

优化构建性能是一件非常复杂的事情，即使找到正确的工具和方法组合来最大限度地利用 CMake 可能有点乏味。然而，过度优化构建的缺点是，构建可能更容易失败，过程中增加的复杂性可能需要更深入的理解，以及更多的专业知识才能进行长期维护。

下一章中，我们将概述从其他构建系统迁移到 CMake 项目的高级策略。

## 14.6. 练习题

回答以下问题来测试对本章的理解：

1. 使用哪个命令行标志可获取 CMake 生成分析信息？
2. 统一构建如何优化编译时间？
3. 统一构建中的 BATCH 和 GROUP 模式有什么区别？
4. 如何将预编译头文件添加到目标中？
5. CMake 如何处理编译器缓存？

# 第 15 章 迁移到 CMake

目前，仍然有一些项目（有时是大型项目）使用不同的构建系统。当然，只要它符合需求，就没有什么错。然而，出于某些原因，可能希望切换到 CMake。例如，也许软件在不同的 IDE 或不同的平台上构建，或者依赖管理变得很麻烦。另一种情况是，仓库结构从大型单体仓库变更为每个库项目的分布式仓库。无论什么原因，迁移到 CMake 可能是一个挑战，特别是对于大型项目。

虽然一次性转换项目是首选的方式，但通常情况下，非技术需求可能无法做到这一点。例如，在迁移过程中可能仍然需要在某些部分进行开发，或者由于各种超出团队控制范围的需求，项目的某些部分无法从一开始就迁移。

因此，我们通常需要循序渐进的方法。更改构建系统很可能会影响 CI/CD 流程，因此也应该考虑这一点。本章中，我们将介绍一些高级策略，了解如何将项目逐步迁移到 CMake。但请注意，具体的迁移路径依赖于具体情况。例如，从一个基于 makefile 的项目迁移到一个单独的仓库，和从一个基于 gradle 的跨多个仓库的项目迁移的效果是不同的。

更改构建系统，甚至可能更改项目结构，对所有参与的人来说可能具有破坏性，因为他们将习惯于使用现有的结构和构建系统。因此，不应该轻率地决定切换到构建系统，而应该只在由显著好处的情况下才这么做。

虽然本章主要关注迁移项目的 CMake 方面，但迁移并不以切换构建系统为目标，而是有其他主要目标，例如简化项目结构或减少项目各部分之间的耦合，以便它们更容易独立维护。谈到这些好处时，不一定是纯技术上的好处，例如可以更好地并行化构建，从而获得更快的构建速度。好处还可能更多地来自“社交”方面，例如：拥有标准化的、众所周知的软件开发方法将减少新开发人员的时间。

本章中，我们将讨论以下内容：

- 高级迁移策略
- 迁移小项目
- 将大型项目迁移到 CMake

本章中，将介绍一些从构建系统迁移到 CMake 的高级概念。迁移小型项目可能非常简单，而大型、复杂的项目则需要预先进行更多的规划。本章结束时，将对将不同规模的项目迁移到 CMake 的不同策略有一个很好的了解。此外，我们将提供一些关于迁移时要检查什么的提示，迁移的大致分步指南，以及如何与遗留构建系统交互。

## 15.1. 相关准备

本章没有具体的技术要求，因为它展示的是概念而不是具体的例子。但当迁移到 CMake 时，建议使用最新版本的 CMake。本章中的例子假设使用的是 CMake 3.21 或更高版本。

## 15.2. 高级迁移策略

将软件项目迁移到 CMake 前，首先要回答一些关于现有项目的问题，并定义目标应该是什么样子。通常，软件项目定义了如何处理以下事情：

- 软件的各个部分(即库和可执行程序)是如何编译的,如何链接在一起的
- 使用哪些外部依赖项,如何找到,以及如何在项目中使用
- 要构建哪些测试,以及如何运行
- 如何安装或打包软件
- 提供额外的信息,如许可证信息、文档、变更日志等

有些项目可能只定义上述要点的一部分。但这些是我们作为开发人员,希望在项目设置中处理的任务。这些任务以结构化的方式定义,例如使用 `makefile` 或特定于 IDE 的项目定义。关于项目如何组织和结构化有无数种方法,适用于一种环境的方法可能不适用于另一种环境。因此,需要对情况进行评估。

有一些工具可以自动将某些构建系统转换为 CMake,例如 `qmake`、`Autotools` 或 `Visual Studio`,但是生成的 CMake 文件的质量不保证,并且它们倾向于假设某些约定,不建议使用。

此外,项目可能会定义如何在 CI/CD 流水中构建、测试和部署,虽然这是密切相关的,但 CI/CD 流水的定义通常不视为项目描述的一部分,而是定义为使用项目的使用者。从一个构建系统变更到另一个构建系统将不可避免地影响 CI/CD 流水,而且对 CI/CD 基础设施进行现代化或更改的愿望通常是更改构建系统的导火索。

只有当旧的构建方式不再使用时,迁移才算完成。建议当项目迁移到 CMake,就删除旧的构建指令,以消除与旧构建方式保持向后兼容性的需要。

理想情况下,项目的所有部分都将迁移到 CMake。某些情况下,这是不可能的,或者是否应该迁移项目的一部分,在经济上可能都有问题。例如,一个项目可能依赖于一个不再积极维护的库,并且注定很快就会淘汰。最好是,迁移工作可以作为一个触发器来实际移除依赖;然而,这通常不可行。在遗留的依赖项无法完全删除的情况下,从项目中删除它可能是一个好主意。这样,就不再将其认为是内部依赖项,而是具有自己的发布周期的外部依赖。此外,若这也可能完成或工作量太大,为这个特定的库做一个例外,并在有限的时间内使用遗留的构建系统与 `ExternalProject` 共存,可能是一种解决方案。对于本章讨论的迁移策略,要区分内部依赖和外部依赖。内部依赖项是指由要迁移的项目的同一组织或人员积极开发的依赖项,因此开发人员可以更改构建过程。外部依赖是指开发人员对构建过程或代码的控制有限或无法控制的依赖。

迁移项目时要考虑一件事,在迁移期间有多少人将无法从事项目工作,以及旧的构建软件的方式和 CMake 必须同时维护多长时间,改变构建系统会对开发人员的工作流程造成很大干扰。有时候,项目的某些部分在完全迁移之前无法继续工作。解决这个问题的最简单方法是暂时停止功能开发,让所有人都来帮助迁移。若这是不可能的,那么良好的沟通和工作划分则是必须的。话虽如此,还是要避免中途停止迁移的情况:迁移大型项目的某些部分时,有些部分仍在使用旧的方式构建软件,这很可能会给两种构建带来更麻烦的问题,从而使得两种方法无法很好的工作。

那么,在将项目从一个构建系统迁移到另一个构建系统时,该如何进行下一步呢?对于主要具有外部依赖关系的小型项目,这可能非常简单。

## 15.3. 迁移小项目

我们将小项目定义为仅包含少数目标且通常一起部署的项目。小型项目在单个库中是自包含的,可以相对快速地了解它们,这些项目可能是构建单个库或带有一些外部依赖的可执行文件的项目。这时,迁移到 CMake 相对简单。对于小型项目,在第一次迭代中,将所有内容放在一个文件

中，可能是获得相对快速和早期结果的最简单方法。若项目已经正确构建，那么重新排列文件，并将 CMakeLists.txt 拆分为多个部分，以便使用 `add_subdirectory()` 会容易得多。

迁移到 CMake 的一般方法如下：

1. 在项目的根目录中创建一个空的 CMakeLists.txt 文件。
2. 识别项目中的目标和相关文件，并在 CMakeLists.txt 文件中创建适当的目标。
3. 找到所有外部依赖项并包含路径，并在必要时将它们添加到 CMake 目标中。
4. 确定必要的编译器特性、标志和编译器定义(若有的话)，并使它们对 CMake 可用。
5. 通过创建必要的目标并使用 `add_test()`，将测试迁移到 CTest。
6. 确定 CMake 的安装或打包说明，包括需要安装的资源文件等。
7. 清理并把项目做好。创建预设，若需要的话重新排列文件和文件夹，需要的话对 CMakeLists.txt 文件进行拆分。

当然，每个步骤具体要做什么，很大程度上取决于原始项目是如何组织的，以及使用了哪种技术。通常，迁移需要多次迭代 CMakeLists.txt 文件，直到一切正常工作，若 CMake 项目的第一实现看起来还不是特别好，这很正常。

对于小型项目来说，处理依赖关系是比较困难的任务之一，因为有一些关于在哪里找到依赖关系，以及它们内部结构如何或隐藏在项目内部的隐含假设。第 5 章介绍过，使用包管理器可以大大降低处理依赖的复杂度。

迁移小型的、基本独立的项目的过程是相对简单的，不过根据最初设置的混乱程度，可能需要相当多的工作才能将一切组织起来并重新运行。更大的组织中，几个这样的小项目可能在一个软件组合中一起使用，这个组合也可以描述为一个项目，迁移需要更多的计划才能继续。

## 15.4. 将大型项目迁移到 CMake

迁移包含大量库和多个可执行文件的大型项目可能是一个相当大的挑战，这些项目实际上可能是多个层次嵌套的项目，具有一个或多个将多个子项目拉在一起的根项目，而这些根项目本身又包含或需要多个子项目。根据软件组合的大小和复杂性，许多共享公共子项目的根项目可能并排存在，这可能使迁移复杂化。创建项目和子项目的依赖关系图(如下图所示)，通常可以帮助我们弄清楚迁移的顺序。每个项目本身可能包含多个项目或目标，这些项目或目标有自己的依赖项：

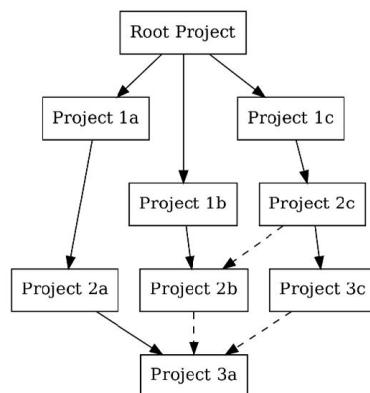


图 15.1 显示各种依赖关系的示例项目层次结构

迁移之前，要做的第一件事是彻底分析项目之间如何相互依赖，以及需要以什么顺序构建。根据项目的状态，最终的图可能会非常大，所以确定从哪里开始可能会有点困难。实际上，依赖图往往不像本书中展示的那样简洁。先理清项目脉络，再迁移到 CMake，还是先迁移到 CMake 再理清项目脉络更容易，这取决于实际情况。若项目非常大且复杂，那么首先在图中找到尽可能独立的孤岛，并从那里开始着手。

对于复杂的、分层的项目，有两种迁移策略。一种是自顶向下的方法，首先迁移并替换根项目，然后根据哪个项目具有最少的依赖关系对子项目进行排序。第二种是自底向上的方法，从依赖项最多的项目开始，逐个迁移各个项目。

自顶向下的方法有一个好处，确保完整的项目可以在早期用 CMake 进行构建、测试和打包，但要求现有的构建系统使用 `ExternalProject` 的方式集成到 CMake 中。自上而下的方法的缺点可能是，早期会生成的 CMake 项目包含大量自定义代码，用于使用旧系统构建的包。实践中，使用一些临时方法在构建中，包含现有项目是相对快速地获得良好结果的实用方法，并且在一定程度上减轻了为相同子项目维护两个构建系统的工作量。

自底向上方法的好处是：每个库迁移到 CMake 已经迁移可以使用依赖项。缺点是只有在所有子项目都能用 CMake 构建后，根项目才能替换。即使项目从下往上迁移，好的做法是尽早创建根 CMake 项目，与原始构建系统中的根项目同时存在。这就可以尽早在新的 CMake 项目中放入外部依赖项、安装配置和打包说明。

下面的图表说明了自顶向下和自底向上策略是如何同时出现的。方框旁边的数字表示迁移的顺序：

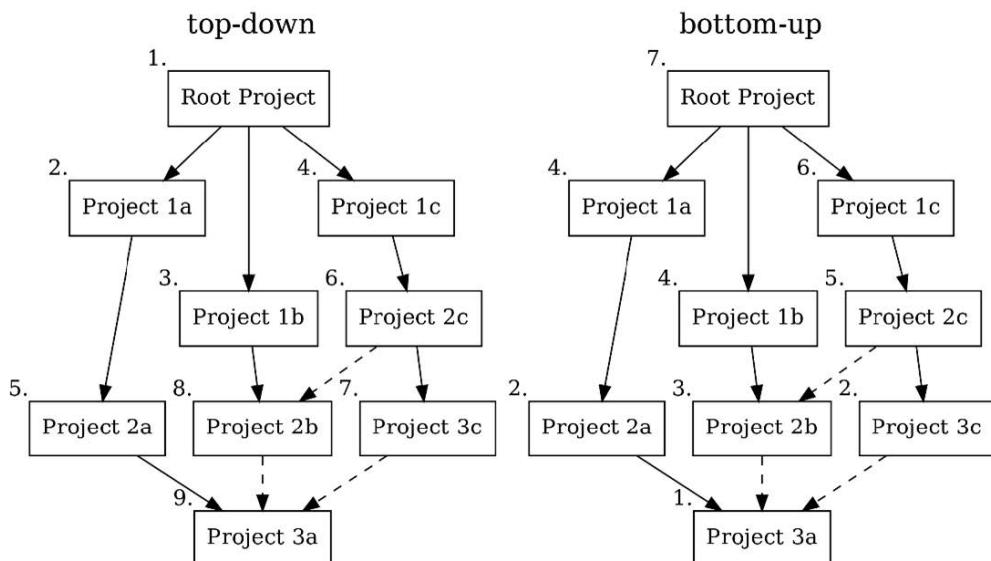


图 15.2 迁移顺序的示例

除了整体的迁移策略之外，另一件需要考虑的事情是项目是要设置为超级构建，还是常规项目。当使用自顶向下的方法时，超级构建结构可能更容易迁移，因为好处是更容易通过设计集成非 CMake 项目。

无论选择自顶向下的方法，还是自底向上的方法来迁移单个项目，迁移大型项目的策略看起来都有以下几点：

1. 分析依赖关系、项目层次结构和部署单元是什么。

2. 决定迁移策略，以及是使用常规项目结构，还是超级构建。
3. 创建或迁移根项目，并使用 `ExternalProject`、`FetchContent` 或中间查找模块(若使用二进制包)拉入所有尚未转换的项目。
4. 使用 CMake 处理项目级别的依赖关系。
5. 如本章最后一节所述，将子项目逐个地转换为 CMake。若使用中间查找模块，则逐个替换：
  - i 可以将依赖项处理更改为包管理器。
  - ii 找到常用选项，并将它们传递至根目录并创建预设。
6. 迁移子节点时，若还没有完成，可以在 CMake 中使用该子节点的已打包文件。
7. 清理、重新组织文件和项目、提高性能等。

通过分析现有的项目层次结构和依赖关系来开始迁移，这将帮助您建立一个迁移计划，以便与所有相关人员进行沟通。创建可视化关系图通常是一个很好的方式，尽管对于非常大的项目，这本身就可能成为一个相当大的挑战。创建迁移计划时要牢记的另一个要点是，确定什么是通常部署在一起，以及哪个子项目有哪个发布频率。很少更改和发布的项目对于迁移来说，可能不像那些经常修改和发布的项目那么重要，确定部署单元与如何打包项目密切相关。根据组织方式的不同，在所有项目迁移之前，可能无法将打包迁移到 CMake。

我们主要讨论了子项目，但在分析现有结构时，重要的是要认识到哪些子项目实际上是可以作为独立的项目，在整个项目上下文中构建的项目；哪些子项目可以作为常规的 CMake 目标来处理，而很少在项目上下文中构建。

创建 `CMakeLists.txt` 主文件将涵盖基本的项目设置，并包括必要的模块，如 `FetchContent`、`CTest`、`CPack` 等。虽然不是直接在 `CMakeLists.txt` 文件中，但也可以在这里设置工具链文件、构建容器或用于交叉编译的 `sysroot`。对于大型项目，主 `CMakeLists.txt` 通常不直接包含目标。相反，可将它们包含在 `add_subdirectory` 或 `FetchContent` 中，或者 `ExternalProject` 的超级构建中。主 `CMakeLists.txt` 应该有以下的结构：

1. 项目定义和 CMake 的最低版本。
2. 全局属性和默认变量，如最低语言标准、自定义构建类型以及搜索和模块路径。
3. 项目范围内使用的模块和辅助函数。
4. 通过 `find_package()` 包含的项目范围的外部依赖项。
5. 构建目标和子项目，使用 `add_subdirectory`、`FetchContent` 添加，或者在超构建下使用 `ExternalProject`。
6. 整个项目的测试，每个子项目都有自己的单元测试，但是集成或系统测试可能位于顶层。
7. `CPack` 的包装说明。

根据定义的复杂性，将外部依赖项的处理、测试和打包放到自己的子目录中，可能有助于保持 CMake 文件的简短和简洁。整个项目使用的外部依赖可能是大型软件框架，如 Qt 或 Boost，也可能是小型但经常使用的实用工具库。

对于自顶向下的方法，可以一开始导入子项目，然后逐个迁移。当使用自底向上策略时，构建目标和子项目很可能为空，然后随着项目的迁移而填充。迁移子项目时，要留意可以传递到根项目，或到预设的公共依赖项或构建选项。

迁移了所有的子项目，仍然有一些维护任务，例如组织打包、交付和分组测试。此外，在迁移所有内容后，CMake 文件中仍有一些混乱也实属正常，因此需要再进行一轮清理，确保迁移项目的功能已经处于可用状态。

迁移大型项目是一项挑战，特别是当构建过程很复杂且缺乏文档时。软件的构建方式有很多种，本节描述的策略试图给出一种通用的方法。然而，每个迁移都有自己独特的方式。某些情况下，构建系统非常复杂，前面描述的迁移策略非但没有帮助，反而起到了反作用。例如，将未迁移的项目包含到 CMake 中非常困难，因此逐步迁移可能比从头开始构建更费力。让我们来了解一下，在使用自顶向下的方法开始时，如何包含使用原始构建系统的子项目。

#### 15.4.1 自顶向下迁移时集成历史遗留项目

对于自顶向下的迁移策略，现有的项目在一开始就提供给 CMake。最简单的方法是使用 ExternalProject，不管是否打算进行超级构建。导入的目标可以直接定义，也可以使用 find 模块定义。对于常规项目，这只是一个中间步骤，以便能够相对快速地构建完整的项目，并将配置和构建顺序的控制权移交给 CMake。生成的 CMake 代码可能不是特别友好，但第一个目标是使用 CMake 构建根项目。在迁移子项目时，一定要一步步地清理。对于由多个库组成的普通项目，或者通过 Git 子模块或类似的方式引入依赖项的项目，ExternalProject\_Add 会省略下载，而是指定 SOURCE\_DIR 属性。包含 Autotools 项目的 CMake 代码可能如下所示：

```
include(ExternalProject)
set(ExtInstallDir ${CMAKE_CURRENT_BINARY_DIR}/install)
ExternalProject_Add(SubProjectXYZ_ext
    SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/SubProjectXYZ/
    INSTALL_DIR ${ExtInstallDir}
    CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix <INSTALL_DIR>
    INSTALL_COMMAND make install
    BUILD_COMMAND make
)
add_library(SubProjectXYZ::SubProjectXYZ IMPORTED SHARED)
set_target_properties(SubProjectXYZ::SubProjectXYZ
    PROPERTIES IMPORTED_LOCATION
        "${CMAKE_CURRENT_LIST_DIR}/SubProjectXYZ/lib/libSubProjectXYZ.so"
    INTERFACE_INCLUDE_DIRECTORIES
        "${CMAKE_CURRENT_LIST_DIR}/SubProjectXYZ/include"
    IMPORTED_LINK_INTERFACE_LANGUAGES "CXX"
)
...
add_dependencies(SomeTarget SubProjectXYZ_ext)
target_link_libraries(SomeTarget SubProjectXYZ::SubProjectXYZ)
```

因为 `ExternalProject` 只在构建时提供内容，所以这种方法只适用于本地文件夹中可用的子项目。因为包含导入目标的目录，当在 `target_link_libraries` 中使用时，必须在配置时存在，导出位置应该指向源目录，而不是外部项目的安装位置。

### 临时解决方案

这里描述的使用 `ExternalProject` 和 `FetchContent` 是一种临时解决办法，以便在迁移时能够将遗留项目包含在 CMake 构建中，这些都不是适合在生产环境中使用的良好实践。该模式允许使用原始的构建系统，并将提供一个导入的目标来链接已经迁移的项目。无论创建这种中间项目结构的努力，是否通过能够在早期使用 CMake 构建完整的项目来证明合理，都必须对每种情况单独考虑。

若从 Microsoft Visual Studio 迁移而不是使用 `ExternalProject`，可以使用 `include_external_msproject()` 直接包含项目文件。

## 15.5. 总结

本章中，学习了将不同规模的项目迁移到 CMake 的概念和策略。当将项目迁移到 CMake 时，所要做的工作和实际工作在很大程度上取决于项目的个人偏好。通过这里描述的方法，选择正确的策略将会更容易。更改构建过程和开发人员的工作流程通常具有破坏性，因此必须仔细考虑是否值得这样做。如本书所述，将项目切换到 CMake，将为构建高质量软件提供所有特性和实践的可能性。此外，拥有一个干净且维护良好的构建系统，可以让开发人员专注于他们的主要任务，即编写代码和交付软件。

接下来就是本书的最后一章了，这一章是关于如何进入 CMake 社区的，寻找扩展阅读的材料，并如何为 CMake 本身做出贡献。

## 15.6. 练习题

回答以下问题来测试对本章的理解：

1. 移植大型项目的两种策略是什么？
2. 当为迁移项目选择自底向上的方法时，首先迁移哪个子项目或目标？
3. 当选择自顶向下的方法时，应该首先迁移哪些项目？
4. 自顶向下方法的优缺点是什么？
5. 使用自底向上方法进行迁移的优缺点是什么？

# 第 16 章 对 CMake 进行贡献

我们已经学到了很多关于 CMake 的知识。但正如你现在可能已经意识到的，CMake 是一个巨大的生态系统，一本书不足以涵盖可以讨论的大量主题。本章中，我们将看一看这些资源，它们可能会帮助你更好地了解 CMake，以及参与 CMake 项目本身的方法。

CMake 是一个灵活的工具，在软件行业的许多项目中使用。因此，CMake 有不断增长的社区支持它。网上有很多资源可以学习和排除您可能遇到的 CMake 问题。

为了理解本章分享的技能，我们将讨论以下内容：

- 找到 CMake 社区
- 为 CMake 贡献力量
- 推荐的书籍和博客

## 16.1. 预备知识

这是一个通读式的章节，没有实践或示例。所以，唯一的要求就是一个安静的地方，还有你的时间。

## 16.2. 找到 CMake 社区

使用 CMake 之后，有些使用者可能会觉得有必要与 CMake 的开发者或其他使用者交换想法，并找到一个平台向可能知道答案的人提问。为此，有一些在线平台在这里进行推荐。

### 16.2.1 Stack Overflow

Stack Overflow 是一个流行的 Q&A 平台，也是大多数开发人员的首选。若对 CMake 有问题或有任何疑问，可以首先在 Stack Overflow 上搜索问题的答案。很可能有人经历过这种问题，或者以前问过相同或类似的问题。还可以查看流行问题列表，以发现使用 CMake 做事的新方法。

当提问时，确保 `cmake` 标签标记了问题。这将使那些对回答 `cmake` 相关问题感兴趣的人更容易找到您的问题。可以通过<https://stackoverflow.com/> 访问 Stack Overflow 主页。

### 16.2.2 Reddit (`r/cmake`)

Reddit 是一个很受欢迎的地方，有专门的、独立的类似公告的区域，叫做看板 (subreddits)，用于围绕主题交换想法。Reddit 也有一个 `r/cmake` 版块，里面有针对 `cmake` 的问题、声明和分享。可以找到许多有用的 GitHub 库，获得有关最新 CMake 发布的通知，并找到可以帮助您的博客文章和材料。可以通过<https://www.reddit.com/r/cmake/> 访问 `r/cmake` 版块。

### 16.2.3 CMake 论坛

CMake 论坛的主要地方 CMake 开发者和用户见面，完全专注于 CMake 的特定事项。该论坛包含公告、如何使用 CMake 的指南、社区空间、CMake 开发空间以及可能感兴趣的更多内容。可以通过<https://discourse.cmake.org/> 访问讨论论坛。

#### 16.2.4 Kitware CMake 的 GitLab 库

Kitware 的 CMake 库也是一个很好的资源，可以找到一些问题的解决方案。尝试在<https://gitlab.kitware.com/cmake/cmake/-/issues>的问题列表中搜索你的问题。这是一个很好的方式，其他人可能已经提交了一个关于这个主题的问题。若没找到，可以通过遵循 CMake 的贡献规则来创建一个新问题。

上述列表并不详尽，还有更多的在线论坛。这四个平台足以让您开始学习。接下来，我们将了解如何为 CMake 项目本身做出贡献。

### 16.3. 为 CMake 贡献力量

CMake 是由 Kitware 开发的开源软件。Kitware 有一个专用的 GitLab 库 (<https://gitlab.kitware.com/cmake>) 中维护 CMake 的开发活动。所有的东西都是开源和透明的，所以参与 CMake 非常容易。可以查看问题，合并请求，并参与到 CMake 的开发中。若认为您在 CMake 中发现了一个 bug 或想要提出功能请求，可以在<https://gitlab.kitware.com/cmake/cmake/-/issues>上创建一个新 issue。若有改进 CMake 的想法，首先通过创建一个 issue 来讨论这个想法。还可以在[https://gitlab.kitware.com/cmake/cmake/-/merge\\_requests](https://gitlab.kitware.com/cmake/cmake/-/merge_requests)上查看合并请求，以帮助检查正在开发的代码。

对开源软件的贡献对于开源世界的可持续性至关重要。请不要犹豫，以任何方便的方式帮助开源社区。您提供的帮助可能很小，但小贡献很快就会积累成更大的成就。接下来，我们将看一些可能会觉得有用的阅读资料。

### 16.4. 推荐的书籍和博客

有很多关于 CMake 的书籍、博客和资源。下面是一份精心挑选的资源列表。下面的列表将让你了解更多关于 CMake 的信息：

- CMake 官方文档: <https://cmake.org/documentation/>  
CMake 的官方文档。内容权威且最新。
- awesome-cmake: <https://github.com/onqtam/awesome-cmake>  
大量的 cmake 相关资源，并定期更新。
- CMake 入门: 有用的资源: <https://embeddedartistry.com/blog/2017/10/04/getting-started-with-cmake-helpfulresources/>  
由嵌入式 Artistry 收集的 CMake 资源
- 现代 CMake 入门: <https://cliutils.gitlab.io/modern-cmake/>  
一本关于详细学习现代 CMake 的在线书籍和很好的资源，由 Henry Schreiner 和许多其他贡献者撰写。
- 更现代的 CMake: <https://hsf-training.github.io/hsf-training-cmake-webpage/01-intro/index.html>  
这是一本现代 CMake 介绍的书，由 HEP 软件基金会编写。
- 更现代的 CMake: <https://www.youtube.com/watch?v=y7ndUhdQuU8>

这段 YouTube 视频是 Deniz Bahadir 在 2018 年 C++ 会议上的演讲，主要目的是提供正确使用 CMake 的技巧。

- 深入理解 CMake——为库作者们: <https://www.youtube.com/watch?v=m0DwB4OvDXk>

这个 YouTube 视频是 CMake 项目的共同维护者 Craig Scott 的 CppCon 演讲，涵盖了面向库开发的 CMake 主题。

- Daniel Pfeifer 的《Effective CMake》: <https://www.youtube.com/watch?v=bsXLMQ6WgIk>

这段 YouTube 视频是 Daniel Pfeifer 关于有效使用 CMake 的演讲，涵盖了 CMake 的使用方式。

- 专业 CMake: 实用指南: <https://crascit.com/professional-cmake/>

这是一本由 CMake 的共同维护者 Craig Scott 撰写的综合性书籍，包含许多在其他地方找不到的细节。

- learning-cmake: <https://github.com/Akagi201/learning-cmake>

这个库是一个示例集合，用于学习 CMake 中的不同用例。

- cmake-examples: <https://github.com/ttroy50/cmake-examples>

这是学习 CMake 中不同用例的又一个很好的示例集合。

说了这么多，我们又到了一个章节的结尾。接下来，我们将总结本章所了解的知识。

## 16.5. 总结

这一章中，我们简要地讨论了可以在网上找到的 CMake 社区，这些社区对 CMake 有贡献，以及阅读和观看的好建议。关于 CMake 的材料和谈话不计其数，内容也在一天天增长。时刻关注关于 CMake 的更新，定期访问论坛，使自己处于最新的状态。

若正在阅读这段话，那么恭喜你！本书要讨论的主题已经全部讲完了。这是最后一章。不要忘记将本书中学到的知识应用到你的日常工作中。我们很享受这段旅程，希望你从本书中学到的知识能对你的实际学习和工作有所帮助。

# 参考答案

这一部分是对所有章节练习题的回答。

## 第 1 章

1. cmake -S /path/to/**source** -B /path/to/build

2. cmake -build /path/to/build

3. ctest

4. cpack

5. 目标是 CMake 组织构建的逻辑单元，可以是可执行文件、库或包含自定义命令。
6. 与变量不同，属性附加到特定的对象或作用域。
7. CMake 预设用于共享构建的工作配置。

## 第 2 章

1. 以下是答案：

A  
cmake -S . -B ./build -DCMAKE\_CXX\_COMPILER:STRING="/usr/bin/clang++"

B  
cmake -S . -B ./build -G "Ninja"

C  
cmake -S . -B ./build  
-DCMAKE\_BUILD\_FLAGS\_DEBUG:STRING="-Wall"

2. 前面在问题 1 中配置的项目可以通过命令行使用 CMake 构建：

```
cmake -S . -B ./build  
-DCMAKE_CXX_COMPILER:STRING= "/usr/bin/clang++ "
```

A

```
cmake --build ./build --parallel 8
```

B

```
cmake --build ./build -- VERBOSE=1
```

C

```
cmake --install ./build --prefix=/opt/project
```

3.

```
cmake --install ./build --component ch2.libraries
```

4.

5. 它是一个 CMake 缓存变量，可以通过 `mark_as_advanced()` 指令将其标记为高级，并在 GUI 中隐藏。

## 第 3 章

1. `add_executable`

2. `add_library`

3. 通过添加 SHARED 或 STATIC 关键字，或设置 `BUILD_SHARED_LIBS` 全局变量。
4. 通过设置 `<LANG>_VISIBILITY_PRESET` 全局属性。

## 第 4 章

1. 可以通过 `install(TARGETS <target_name>)` 指令来实现。
2. 指定目标的输出工件。
3. 不能，因为头文件没有分类为目标的输出工件，必须通过 `install(DIRECTORY)` 进行安装。
4. `GNUInstallDirs` 模块为安装提供特定于系统的默认路径，例如 bin、lib 和 include。
5. 使用 `install(DIRECTORY)` 指令的 `PATTERN` 和 `FILES_MATCHING` 参数。

## 第 5 章

1. `find_file`, `find_path`, `find_library`, `find_program` 和 `find_package`。

2. IMPORTED\_LOCATION 和 INTERFACE\_INCLUDE\_DIRECTORIES 属性。
3. HINTS 优先于 PATHS。
4. ExternalProject 在构建时下载外部内容。
5. FetchContent 在配置时下载外部内容。

## 第 6 章

1. Doxygen 是 C 和 C++ 项目文档生成工具的实际标准。
2. 因为 CMake 已经为 Doxygen 提供了一个 find 模块，也可以通过 `find_package(...)` 查找。
3. 是的。Doxygen 可以绘制图形，若环境中有诸如 DOT、Graphviz 和 PlantUML 等绘图软件可用的话。要启用 DOT 绘图，将 HAVE\_DOT 设置为 TRUE 就足够了。对于 PlantUML，`PlantUML_JAR_PATH` 需要设置为包含 `plantuml.jar` 文件的路径。
4. `@startuml` 和 `@enduml`。
5. `PLANTUML_JAR_PATH` 需要设置为包含 `plantuml.jar` 文件的路径。
6. 通过 `install(DIRECTORY)`。

## 第 7 章

1. 通过使用 `add_test` 函数定义测试。
2. 可以在测试名称上使用正则表达式 (`ctest -R`)，也可以使用测试编号 (`ctest -I`)。
3. 通过调用 `ctest --repeat:until-pass:n` 或 `ctest --repeat:untilfail:n`。
4. `ctest -j <num_of_jobs> --schedule-random`。
5. 通过为各自的测试设置 `RESOURCE_LOCK` 或 `RESOURCE_GROUP` 属性。
6. 静态代码分析器是通过，将包含参数的命令行传递给各自的目标属性来启用的。
7. 通过将它们添加到多配置类型生成器的 `CMAKE_CONFIGURATION_TYPES` 属性中，或者将它们添加到 `CMAKE_BUILD_TYPE` 属性中。

## 第 8 章

1. 使用 `add_custom_command` 添加的命令在构建时执行，而使用 `execute_process` 添加的命令在配置时执行。
2. 一个签名用于创建自定义构建步骤，而另一个用于生成文件。
3. 所有生成器只可靠地支持 `POST_BUILD`。
4. 变量可以定义为  `${VAR}`  或  `@VAR@` 。
5. 变量替换可以通过传递 `@ONLY` 来控制，只替换定义为  `@VAR@`  的变量，或者通过指定 `COPYONLY` 选项来控制，不执行替换。
6. 使用 `cmake -E`，可以直接执行常见任务。使用 `cmake -P`，将 `.cmake` 文件作为脚本执行。

## 第 9 章

1. CMakePresets.json 通常与项目一起维护和交付，而 CMakeUserPresets.json 通常与项目一起维护和交付由用户维护。关于语法和内容，没有区别。
2. 通过调用 `cmake --preset=presetName`, `cmake --build --preset=presetName`, 或 `ctest --preset=presetName`。
3. 有配置、构建和测试预置，构建和测试预设值依赖于配置预设值来确定构建目录。
4. 预设配置应该定义名称、生成器和要使用的构建目录。
5. 设置值的第一个预设值优先。
6. 使用编辑器对构建容器的本机支持，从容器内运行编辑器，或者每次启动容器以调用容器内的单个命令。
7. 系统名称、sysroot 的位置、要使用的编译器以及 `find_` 指令的行为。

## 第 10 章

1. 超级构建是构建使用多库软件项目的一种方法。
2. 我们没有包管理器，并且希望项目能够满足自己的依赖。
3. 使用锚点，如分支、标记或提交哈希。

## 第 11 章

1. 模糊测试是一种测试技术，将计算机生成的数据输入到系统或函数中，以检查目标对象的行为是否符合预期。
2. 语料库数据是在模糊测试运行期间持续存在的所有有趣输入的集合，语料库可以随着时间的推移而增长。
3. 通过传递`-fsanitize=fuzzer` 给目标的编译器和链接器标志 (`target_compile_options` 和 `target_link_options`)
4. OpenSSL - Heartbleed 和 Bash - Shellshock

## 第 12 章

1. 工具链文件可以通过`--toolchain` 命令行标志、`CMAKE_TOOLCHAIN_FILE` 变量传递，也可以通过 `CMAKE` 预置中的 `toolchainFile` 选项传递。
2. 通常，交叉编译需要在工具链文件中执行以下操作：
  - A 定义目标系统和架构
  - B 提供构建软件所需工具的路径
  - C 设置编译器和链接器的默认标志
  - D 指向 sysroot，若使用交叉编译，可以指向暂存目录
  - E 设置 CMake 的 `find_` 指令的搜索顺序

3. 暂存目录用 `CMAKE_STAGING_PREFIX` 变量设置，若 `sysroot` 不需要修改，就把构建的工件放在这里。
4. 模拟器命令以分号分隔的列表形式在 `CMAKE_CROSSCOMPILING_EMULATOR` 变量中传递。
5. 项目中调用 `project()` 或 `enable_language()` 都会触发对特性的检测。
6. 编译器检查的配置上下文可以用 `cmake_push_check_state()` 保存，也可以用 `cmake_pop_check_state()` 恢复到之前的状态。
7. 若设置了 `CMAKE_CROSSCOMPILING`，`try_run()` 调用将只编译测试，而不会运行它，除非设置了模拟器命令。
8. 构建目录应该完全清除，因为在删除缓存时，编译器检查的临时构件可能无法正确地重新构建。

## 第 13 章

1. 函数和宏。
2. CMake 模块包含 CMake 代码、函数和用于特定目的的宏。
3. 通过将其纳入所需的范围。
4. 向 `include(...)` 指令的搜索路径添加路径。
5. 通过使用 `git submodules/subtrees` 或 CMake 的 `FetchContent/ExternalProject`。
6. 函数定义一个新的变量作用域，而宏不是。

## 第 14 章

1. 使用 `--profilingformat <filename>` `--profilingformat=google-trace`。
2. 通过将各种编译单元组合成一个编译单元，重新链接的需求就会减少。
3. 批处理模式下，CMake 自动将源分组在一起;GROUP 模式下，分组必须由用户指定。默认情况下，批处理模式对统一构建的所有源进行分组，而组只将明确标记的文件添加到统一构建中。
4. 通过使用 `target_precompile_headers` 函数。预编译的头文件会自动包含，而不需要在文件中使用 `#include`。
5. 在编译器命令前加上 `CMAKE_<LANG>_COMPILER_LAUNCHER` 指定的命令。

## 第 15 章

1. 大型项目可以自顶向下或自底向上迁移。
2. 使用自底向上方法时，应该首先迁移那些具有最多传入依赖项的项目或目标。
3. 当选择自顶向下的方法时，应该首先迁移依赖最少的项目。
4. 自顶向下的方法允许使用 CMake 作为入口点来快速构建整个项目。此外，对于每个迁移的项目，在项目完成时可以丢弃旧的构建系统。缺点是自顶向下的方法需要一些中间代码。

5. 自底向上方法比自顶向下方法需要更少的中间代码，并允许从一开始就编写干净的 CMake 代码。这样做的缺点是，只有在迁移了所有子项目之后，才能构建完整的项目。