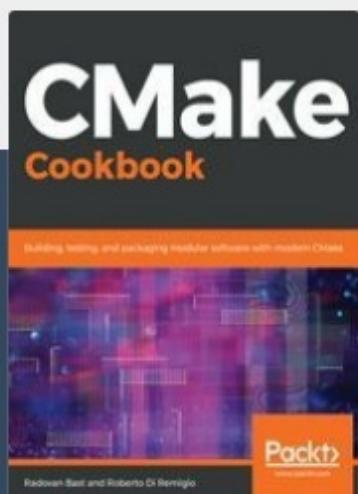


CMake菜谱 (CMake Cookbook 中文版)



使用CMake软件对项目模块，进行构建、测试和打包。



下载手机APP
畅享精彩阅读

目 录

致谢

Introduction

前言

第0章 配置环境

0.1 获取代码

0.2 Docker镜像

0.3 安装必要的软件

0.4 测试环境

0.5 上报问题并提出改进建议

第1章 从可执行文件到库

1.1 将单个源文件编译为可执行文件

1.2 切换生成器

1.3 构建和链接静态库和动态库

1.4 用条件句控制编译

1.5 向用户显示选项

1.6 指定编译器

1.7 切换构建类型

1.8 设置编译器选项

1.9 为语言设定标准

1.10 使用控制流

第2章 检测环境

2.1 检测操作系统

2.2 处理与平台相关的源代码

2.3 处理与编译器相关的源代码

2.4 检测处理器体系结构

2.5 检测处理器指令集

2.6 为Eigen库使能向量化

第3章 检测外部库和程序

3.1 检测Python解释器

3.2 检测Python库

3.3 检测Python模块和包

3.4 检测BLAS和LAPACK数学库

3.5 检测OpenMP的并行环境

3.6 检测MPI的并行环境

3.7 检测Eigen库

3.8 检测Boost库

- 3.9 检测外部库:I. 使用pkg-config
- 3.10 检测外部库:II. 自定义find模块

第4章 创建和运行测试

- 4.1 创建一个简单的单元测试
- 4.2 使用Catch2库进行单元测试
- 4.3 使用Google Test库进行单元测试
- 4.4 使用Boost Test进行单元测试
- 4.5 使用动态分析来检测内存缺陷
- 4.6 预期测试失败
- 4.7 使用超时测试运行时间过长的测试
- 4.8 并行测试
- 4.9 运行测试子集
- 4.10 使用测试固件

第5章 配置时和构建时的操作

- 5.1 使用平台无关的文件操作
- 5.2 配置时运行自定义命令
- 5.3 构建时运行自定义命令:I. 使用add_custom_command
- 5.4 构建时运行自定义命令:II. 使用add_custom_target
- 5.5 构建时为特定目标运行自定义命令
- 5.6 探究编译和链接命令
- 5.7 探究编译器标志命令
- 5.8 探究可执行命令
- 5.9 使用生成器表达式微调配置和编译

第6章 生成源码

- 6.1 配置时生成源码
- 6.2 使用Python在配置时生成源码
- 6.3 构建时使用Python生成源码
- 6.4 记录项目版本信息以便报告
- 6.5 从文件中记录项目版本
- 6.6 配置时记录Git Hash值
- 6.7 构建时记录Git Hash值

第7章 构建项目

- 7.1 使用函数和宏重用代码
- 7.2 将CMake源代码分成模块
- 7.3 编写函数来测试和设置编译器标志
- 7.4 用指定参数定义函数或宏
- 7.5 重新定义函数和宏
- 7.6 使用废弃函数、宏和变量

7.7 add_subdirectory的限定范围

7.8 使用target_sources避免全局变量

7.9 组织Fortran项目

第8章 超级构建模式

8.1 使用超级构建模式

8.2 使用超级构建管理依赖项:I.Boost库

8.3 使用超级构建管理依赖项:II.FFTW库

8.4 使用超级构建管理依赖项:III.Google Test框架

8.5 使用超级构建支持项目

第9章 语言混合项目

9.1 使用C/C++库构建Fortran项目

9.2 使用Fortran库构建C/C++项目

9.3 使用Cython构建C++和Python项目

9.4 使用Boost.Python构建C++和Python项目

9.5 使用pybind11构建C++和Python项目

9.6 使用Python CFFI混合C，C++，Fortran和Python

第10章 编写安装程序

10.1 安装项目

10.2 生成输出头文件

10.3 输出目标

10.4 安装超级构建

第11章 打包项目

11.1 生成源代码和二进制包

11.2 通过PyPI发布使用CMake/pybind11构建的C++/Python项目

11.3 通过PyPI发布使用CMake/CFFI构建C/Fortran/Python项目

11.4 以Conda包的形式发布一个简单的项目

11.5 将Conda包作为依赖项发布给项目

第12章 构建文档

12.1 使用Doxygen构建文档

12.2 使用Sphinx构建文档

12.3 结合Doxygen和Sphinx

第13章 选择生成器和交叉编译

13.1 使用CMake构建Visual Studio 2017项目

13.2 交叉编译hello world示例

13.3 使用OpenMP并行化交叉编译Windows二进制文件

第14章 测试面板

14.1 将测试部署到CDash

14.2 CDash显示测试覆盖率

14.3 使用AddressSanifier向CDash报告内存缺陷

14.4 使用ThreadSanitizer向CDash报告数据争用

第15章 使用CMake构建已有项目

15.1 如何开始迁移项目

15.2 生成文件并编写平台检查

15.3 检测所需的链接和依赖关系

15.4 复制编译标志

15.5 移植测试

15.6 移植安装目标

15.7 进一步迁移的措施

15.8 项目转换为CMake的常见问题

第16章 可能感兴趣的书

16.1 留下评论——让其他读者知道你的想法

致谢

当前文档 《CMake菜谱 (CMake Cookbook中文版)》 由 进击的皇虫 使用 书栈网 (BookStack.CN) 进行构建，生成于 2020-02-13。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：陈晓伟 译 <https://github.com/xiaoweiChen/CMake-Cookbook>

文档地址：<http://www.bookstack.cn/books/CMake-Cookbook>

书栈官网：<https://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

CMake Cookbook

- 作者: Radovan Bast & Roberto Di Remigio
- 译者: 陈晓伟
- 首次出版于2018年9月
- ISBN 978-1-78847-071-1

本书概述

使用CMake软件对项目模块，进行构建、测试和打包。

本书作者

Radovan Bast就职于特罗姆森的挪威北极大学(UiT, University of Norway in Troms)的高性能计算小组工作，对项目的代码精简进行指导。他拥有化学博士学位，并作为开发人员参与了许多量子化学项目。喜欢新的编程语言和技术，并向学生和研究人员传授编程经验。在2008年与CMake结缘，并移植了大量源码，并且将一些源码迁移到CMake社区。

Roberto Di Remigio是挪威大学(特罗姆森)和美国弗吉尼亚理工学院的化学博士后研究员，目前在研究随机方法和求解模型。是[PCMSolver](#)和[Psi4](#)开源量子化学项目的开发人员。为量子化学的发展做出了贡献，其参与的项目有：Dirac、MRCPP、Dalton、LSDalton、Xcun和ReSpect。经常使用C++和Fortran编码。

我们要感谢本书评审Eric Noulard和Eric Noulard的宝贵意见和建议。特别是Eric的反馈和建议，明显提高了本书的质量。我们还感谢Loria.Burns对第8章到第11章的评论和建议。特别感谢特罗姆森公共图书馆为写作和思考提供了一个良好环境。我们非常感谢Travis CI、GmbH、Appveyor Systems公司和Circle Internet Services公司提供的测试设备和支持—正是由于他们的支持，我们才有信心在主流操作系统中完成本书的示例。

本书评审

Eric Noulard博士，具有法国恩塞伊特大学的工程学学位，法国乌夫斯克大学的计算机科学博士学位。20多年来，使用多种语言编写源码。从2006年开始使用CMake，这些年来也一直是CMake的积极贡献者。其职业生涯中，曾为私人公司和政府机构工作。现在就职于Antiot，开发和营销高端信息检索技术和解决方案。

Eric Noulard来自以色列，是一名软件开发人员和作家。从2000年起就为置身于各种开源和开放文化项目。除此之外，还发起过游戏解决方案，比如Pysol FC系列纸牌游戏，采用了财富模式，解决了290多个[欧拉问题](#)。在平时，会写了一些故事、随笔和格言之类的文章。

本书相关

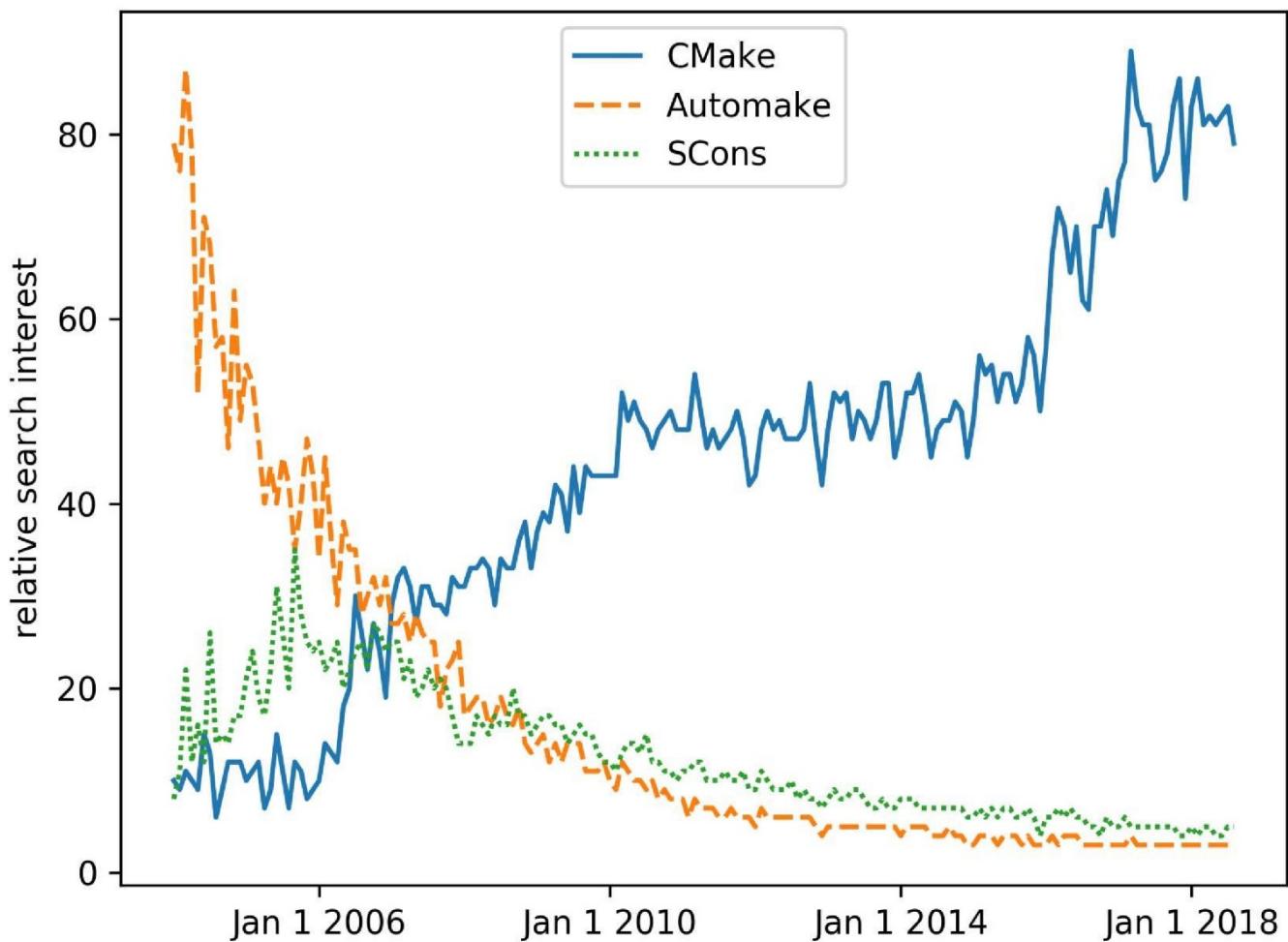
- github翻译地址: <https://github.com/xiaoweiChen/CMake-Cookbook>
- gitbook 在线阅读: <https://app.gitbook.com/@chenxiaowei/s/cmake-cookbook/>
- 本书源码下载地址: <https://github.com/dev-cafe/cmake-cookbook>
- 本书源码下载地址: <https://github.com/PacktPublishing/CMake-Cookbook>

前言

我们日常生活的每一个方面都有计算机软件的存在：它可以为我们触发的闹钟，并每时每刻的为我们提供通信、银行、天气、公交、日历、会议、旅行、相册、电视、音乐流、社交媒体、晚餐和电影预定等服务信息。

软件内部具有一定的层次结构：应用程序是基于框架构建，框架基于开发库，开发库使用更小的库或可执行文件，当然还有更小的组件。开发库和可执行文件通常需要使用源代码进行构建。我们通常只看到最外层，但软件内部需要仔细组织和构建。本书就是使用CMake，对使用源代码构建开发库和可执行文件的项目进行组织。

CMake及其姊妹CTest、CPack和CDash已经成为源码构建的主要工具集，在使用和流行性方面超过了许多类似工具，例如备受推崇的GNU自动工具和最新的基于Python的SCons构建系统。



随着时间的推移，搜索三个流行的构建系统：CMake、Automake和SCons。对比标准是通过搜索相关术语的次数来衡量的，其中数据由谷歌提供。

CMake项目的始于1999年，当时开发公司Kitware被委托设计一套新的工具来简化研究人员的日常工作软件。目标很明确：提供一组工具，可以在不同平台上配置、构建、测试和部署项目。有关CMake项

书中设计的精彩叙述，请访问 <https://www.aosabook.org/en/cmake.html>。

CMake是一个构建生成器，提供了强大的领域特定语言(DSL)来描述构建系统应该实现的功能。这是CMake的主要优势之一，它允许使用相同的CMake脚本集生成平台原生构建系统。CMake软件工具集，使开发人员可以完全控制给定项目的生命周期：

- **CMake**是描述如何在所有主要硬件和操作系统上配置、构建和安装项目，无论是构建可执行文件、库，还是两者都要构建。
- **CTest**定义测试、测试套件，并设置应该如何执行。
- **CPack**为打包需求提供了DSL。
- **CDash**将项目的测试结果在面板中展示。

俗话说得好：挖得越深，发现的石头(阻碍，困难)越多。为了编写这本书，我们仔细地对软件层进行了深入挖掘，这也是CMake的目标。不同的平台上构建不同的软件组件和库时，我们遇到的阻碍和承受的工作量有时令人畏惧，且每个组件和库都有自己的特点。不过，我们已经清除了许多阻碍，也很高兴与读者分享我们的成果和技巧。En... 总会留下一些石头，但每一块石头都会带来新的理解，社区欢迎你分享这些理解。

适读人群

编写能够在许多不同平台上本地、可靠并高效地运行的软件，对于工业和社会的所有部门都至关重要。软件构建系统就是这项任务的中心。它们是软件开发生命周期管理的关键部分：从孵化和原型开发到测试，一直到打包、部署和发布。

CMake旨在帮助您管理这些操作：如果希望使用CMake管理构建系统的软件开发人员，或者希望能够理解，并能修改其他人编写的CMake代码，那么这本书非常合适您。

覆盖内容

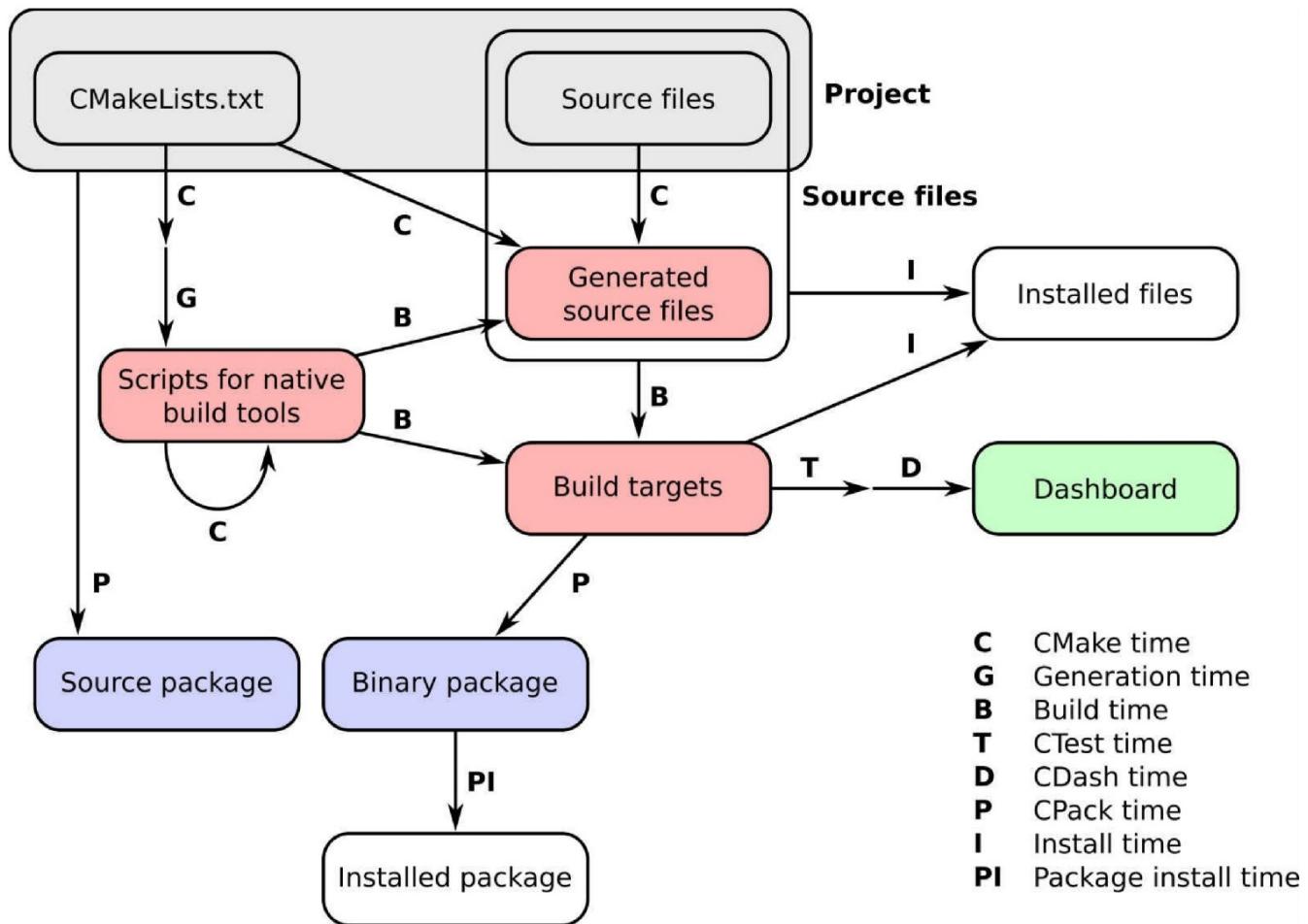
本书中有一系列循序渐进的任务。每一任务中，我们都介绍了相关的CMake信息，从而展示如何实现目标，而不是用细节来折磨读者。本书的最后，您将能够自信地处理日益复杂的操作，并在自己的实际项目中利用书中的技巧。

我们将讨论以下内容：

- 使用CMake配置、构建、测试和安装代码项目
- 检测用于条件编译的操作系统、处理器、库、文件和程序
- 提高代码的可移植性
- CMake帮助下，将大型代码库重构为模块
- 构建多语言项目
- 知道如何调整他人编写的CMake配置文件

- 打包项目进行发布
- 将项目迁移为CMake构建

CMake管理的工作流发生在许多阶段(time)，我们称之为时序。可以简洁地总结如下图：



- **CMake time**或**configure time**，是CMake运行时的情况。这个阶段中，CMake将处理项目中的CMakeLists.txt文件并配置它。
- **Generation time**配置成功后，CMake将生成本地构建工具所需的脚本，以执行项目中的后续步骤。
- **Build time**这是在平台和工具原生构建脚本上调用原生构建工具的时候，这些脚本以前是由CMake生成的。此时，将调用编译器，并在特定的构建目录中构建目标(可执行文件和库)。注意递归的CMake time箭头：这看起来令人困惑，但是我们将在本书中多次使用它，用来实现平台无关的构建。
- **CTest time**或**test time**，运行项目的测试套件，以检查目标是否按预期执行。
- **CDash time**或**report time**，将测试结果上传到面板，与其他开发人员共享。
- **Install time**，将项目的目标、源文件、可执行文件和库从构建目录安装到安装位置。
- **CPack time**或**packaging time**，将项目打包以便发布，可以是源代码，也可以是二进制代码。
- **Package install time**，在系统范围内安装新生成的包。

本书的路线图如下：

第1章，从简单的可执行文件到库，如何配置和使用CMake构建简单的可执行文件和库。

第2章，检测环境，如何使用简单的CMake命令与操作系统和处理器体系结构交互。

第3章，检测外部库和程序，如何简化对项目依赖项的检测。

第4章，创建和运行测试，解释如何利用CMake和CTest的功能来定义和运行测试。

第5章，配置时操作和构建时操作，如何使用CMake在构建过程的不同阶段执行定制化操作。

第6章，生成源码，CMake命令可自动生成源码。

第7章，结构化项目，用于组织您的项目，使它们更易于维护。

第8章，超级构建，解释了CMake超级构建模式，用于管理关键项目的依赖关系。

第9章，混合语言项目，构建不同编程语言混合的项目。

第10章，编写一个安装程序，使用CMake安装项目。

第11章，打包项目，如何使用CPack生成源文件，并将源文件打包，以及构建用于发布的Python和Conda包。

第12章，生成文档，如何使用CMake也生成代码的文档。

第13章，选择生成器和交叉编译，如何使用CMake交叉编译项目。

第14章，测试面板，如何将测试结果报告到在面板上。

第15章，将项目移植到CMake中，将展示实践示例、注意事项和一些技巧，这些将帮助您将项目移植到基于CMake的构建系统中。

预备知识

这是一本为程序员写的书，我们假设您以具备一定的基本知识，并熟悉以下内容：

- 熟悉命令行方式
- 熟悉本地开发软件的环境
- 熟悉编译语言C++、C或Fortran，以及您使用的编译器
- 熟悉Python

示例源码

您可以从 <https://github.com/dev-cafe/cmake-cookbook> 下载本书的示例代码示例。有关详细信息，请参见设置系统部分。

彩图下载

我们还提供了一个PDF文件，其中包含本书中使用的屏幕截图/图表的彩色图像。您可以在这里下载：
http://www.packtpub.com/sites/default/files/downloads/CMakeCookbook_ColorImages.pdf

使用惯例

本书中使用了许多文本约定。

CodeInText：表示文本、文件夹名称、文件名、模块名称和目标名称中的代码命令。

代码块设置如下：

```
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
project(recipe-01 LANGUAGES CXX)
add_executable(hello-world hello-world.cpp)
```

任何命令行输入都是粗体的，并在命令前面包含一个\$提示符来输入：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
```

为了区分命令行输入和输出，我们将输出设置为非粗体：(当前译本不支持)

```
1. $ ./hello-world
2. Hello World!
```

NOTE 重要的提示会出现该标识。

TIPS 提示和技巧会出现该标识。

(PS:原始标识由于在线上观看会使内容错位，观看体验下降，从而换成文字)

额外读物

CMake的在线文档非常全面，我们将在书中引用它：<https://cmake.org/documentation/>

编写这本书的过程中，我们还受到了其他文章的启发：

- Daniel Pfeifer在GitHub上的介绍：https://github.com/boostcon/cppnow_presentations_2017/blob/master/05-19-2017_friday/effective_cmake_daniel_pfeifer_cppnow_05-19-2017.pdf
- Eric Noulard编写的CMake教程，可以在GitHub上找到：
<https://github.com/TheErk/CMake-tutorial>
- Craig Scott的“cmake相关”博文：<https://crascit.com/tag/cmake/>

我们还推荐浏览Viktor Kirilov收集的CMake资源、脚本、模块和示例的列表：

<https://github.com/onqtam/awesome-cmake>

我们的书并不是唯一一本涉及CMake的书：

- 2015年，由Ken Martin和Bill Hoffman合著的《精通CMake》，Kitware Inc.出版。
- Craig Scott的《Professional CMake》 <https://crascit.com/professional-cmake/>

联系方式

我们欢迎读者的反馈。

源代码改进和问题：请直接向 <https://github.com/dev-cafe/cmake-cookbook> 提出，并通过创建 <https://github.com/dev-cafe/cmake-cookbook/issues> 进行讨论

内容反馈：电子邮件 feedback@packtpub.com，并在邮件主题中提到书名。如果你对本书的任何方面有任何疑问，请发邮件至 questions@packtpub.com

勘误表：虽然我们已经尽了一切努力确保我们的内容的准确性，但错误还是会有。如果您在这本书中发现了错误，务必告诉我们，不胜感激。可以访问 www.packtpub.com/submit-errata，选择您的图书，单击Errata提交表单链接，并输入详细信息。

盗版：如果您在互联网上发现我们作品的任何形式的非法拷贝，希望提供相应地址或网站名称，我们将不胜感激。请通过 copyright@packtpub.com 联系我们，并提供该材料的链接。

如果你对名作者感兴趣：如果有一个你擅长的主题，并且你对写作或为一本书做贡献感兴趣，请访问 authors.packtpub.com

第0章 配置环境

学习CMake之前，需要对系统进行设置，这样才能运行所有示例。

本章的主要内容有：

- 如何获取代码
- 如何在GNU/Linux、macOS和Windows上安装运行示例所需的所有工具
- 自动化测试如何工作
- 如何报告问题，并提出改进建议

我们会尽可能让初学者看懂本书的内容。不过，这本书并非完全适合零基础人士。我们假设，您对构建目标平台上可用的软件，及本地工具有基本的了解。有Git版本控制的经验，可与源码库进行“互动”（不是必需）。

0.1 获取代码

本书的源代码可以在GitHub上找到，网址是 <https://github.com/dev-cafe/cmake-cookbook>。开源代码遵循MIT许可：只要原始版权和许可声明包含在软件/源代码的任何副本中，可以以任何方式重用和重新混合代码。许可的全文可以在 <https://opensource.org/licenses/MIT> 中看到。

为了测试源码，需要使用Git获取代码：

- 主要的GNU/Linux发行版都可以通过包管理器安装Git。也可以从Git项目网站 <https://git-scm.com> 下载二进制发行版，进行安装。
- MacOS上，可以使用自制或MacPorts安装Git。
- Windows上，可以从git项目网站(<https://git-scm.com>)下载git可执行安装文件。

可以通过github桌面客户端访问这些示例，网址为 <https://desktop.github.com>。

另一种选择是从 <https://github.com/dev-cafe/cmake-cookbook> 下载zip文件。

安装Git后，可以将远程库克隆到本地计算机，如下所示：

```
1. $ git clone https://github.com/dev-cafe/cmake-cookbook.git
```

这将创建一个名为 `cmake-cookbook` 的文件夹。本书内容与源码的章节对应，书中章节的编号和源码的顺序相同。

在GNU/Linux、MacOS和Windows上，使用最新的持续集成进行测试。我们会在之后讨论测试的设置。

我们用标签v1.0标记了与本书中打印的示例相对应的版本。为了与书中内容对应，可以如下获取此特定版本：

```
$ git clone --single-branch -b v1.0 https://github.com/dev-cafe/cmake-cookbook.git
```

我们希望收到Bug修复，并且Github库将继续发展。要获取更新，可以选择库的master分支。

0.2 Docker镜像

在Docker中进行环境搭建，无疑是非常方便的(依赖项都已经安装好了)。我们的Docker镜像是基于Ubuntu 18.04的镜像制作，您可以按照官方文档<https://docs.docker.com> 在您的操作系统上安装Docker。

Docker安装好后，您可以下载并运行我们的镜像，然后可以对本书示例进行测试：

```
1. $ docker run -it devcafe/cmake-cookbook_ubuntu-18.04
2. $ git clone https://github.com/dev-cafe/cmake-
3. cookbook.git
4. $ cd cmake-cookbook
5. $ pipenv install --three
6. $ pipenv run python testing/collect_tests.py 'chapter-*/*/recipe-*'
```

0.3 安装必要的软件

与在Docker中使用不同，另一种选择是直接在主机操作系统上安装依赖项。为此，我们概括了一个工具栈，可以作为示例的基础。您必须安装以下组件：

1. CMake
2. 编译器
3. 自动化构建工具
4. Python

我们还会详细介绍，如何安装所需的某些依赖项。

0.3.1 获取CMake

本书要使用的CMake最低需要为3.5。只有少数示例，演示了3.5版之后引入的新功能。每个示例都有提示，指出示例代码在哪里可用，以及所需的CMake的最低版本。提示信息如下：

NOTE:这个示例的代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe10> 中找到，其中包括一个C示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行了测试。

有些(如果不是大多数)示例仍然适用于较低版本的CMake。但是，我们没有测试过这个。我们认为CMake 3.5是大多数系统和发行版的默认软件，而且升级CMake也没什么难度。

CMake可以以多种方式安装。下载并提取由Kitware维护的二进制发行版，可以在所有平台上运行，下载页面位于 <https://cmake.org/download/>。

大多数GNU/Linux发行版都在包管理器中提供了CMake。然而，在一些发行版中，版本可能比较旧，因此下载由Kitware提供的二进制文件当然是首选。下面的命令将从CMake打包的版本中下载并安装在 `$HOME/Deps/CMake` (根据您的偏好调整此路径)下的CMake 3.5.2：

```

1. $ cmake_version="3.5.2"
2. $ target_path=$HOME/Deps/cmake/${cmake_version}
   $ cmake_url="https://cmake.org/files/v${cmake_version%.*}/cmake-
3. ${cmake_version}-Linux-x86_64.tar.gz"
4. $ mkdir -p "${target_path}"
5. $ curl -Ls "${cmake_url}" | tar -xz -C "${target_path}" --strip-components=1
6. $ export PATH=$HOME/Deps/cmake/${cmake_version}/bin$PATH:+:$PATH
7. $ cmake --version

```

macOS获取最新版本的CMake：

```
1. $ brew upgrade cmake
```

Windows上，可以使用Visual Studio 2017，它提供了CMake支持。Visual Studio 2017的安装记录在第13章，可选生成器和交叉编译，示例技巧1，使用Visual Studio 2017构建CMake项目。

或者，可以从 <https://www.msys2.org> 下载MSYS2安装程序，按照其中给出的说明更新包列表，然后使用包管理器 pacman 安装CMake。下面的代码正在构建64位版本：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-cmake
```

对于32位版本，请使用以下代码(为了简单起见，我们以后只会提到64位版本)：

```
1. $ pacman -S mingw64/mingw-w64-i686-cmake
```

MSYS2的另一个特性是在Windows上提供了一个终端，比较像Unix操作系统上的终端，提供可用的开发环境。

0.3.2 编译器

我们将需要C++、C和Fortran的编译器。编译器的版本需要比较新，因为我们需要在大多数示例中支持最新的语言标准。CMake为来自商业和非商业供应商的许多编译器，提供了非常好的支持。为了让示例始终能够跨平台，并尽可能独立于操作系统，我们使用了开源编译器：

- GNU/Linux上，GNU编译器集合(GCC)是直接的选择。它是免费的，适用于所有发行版。例如，在Ubuntu上，可以安装以下编译器：

```
1. $ sudo apt-get install g++ gcc gfortran
```

- 在LLVM家族中，Clang也是C++和C编译器的一个很好的选择：

```
1. $ sudo apt-get install clang clang++ gfortran
```

- macOS上，XCode附带的LLVM编译器适用于C++和C。我们在macOS测试中使用了GCC的Fortran编译器。GCC编译器必须使用包管理器单独安装：

```
1. $ brew install gcc
```

- Windows上，可以使用Visual Studio测试C++和C示例。或者，可以使用MSYS2安装程序，MSYS2环境中(对于64位版本)使用以下单个命令安装整个工具链，包括C++、C和Fortran编译

器：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-toolchain
```

0.3.3 自动化构建工具

自动化构建工具为示例中的项目提供构建和链接的基础设施，最终会安装和使用什么，很大程度上取决于操作系统：

- GNU/Linux上，GNU Make(很可能)在安装编译器时自动安装。
- macOS上，XCode将提供GNU Make。
- Windows上，Visual Studio提供了完整的基础设施。MSYS2环境中，GNU Make作为mingw64/mingw-w64-x86_64工具链包的一部分，进行安装。

为了获得最大的可移植性，我们尽可能使示例不受这些系统相关细节的影响。这种方法的优点是配置、构建和链接，是每个编译器的固有特性。

Ninja是一个不错的自动化构建工具，适用于GNU/Linux、macOS和Windows。Ninja注重速度，特别是增量重构。为GNU/Linux、macOS和Windows预先打包的二进制文件可以在GitHub库中找到，网址是 <https://github.com/ninja-build/ninja/releases>。

Fortran项目中使用CMake和Ninja需要注意。使用CMake 3.7.2或更高版本是必要的，Kitware还有维护Ninja，相关包可以在 <https://github.com/Kitware/ninja/releases> 上找到。

在GNU/Linux上，可以使用以下一系列命令安装Ninja：

```
1. $ mkdir -p ninja
$ 
ninja_url="https://github.com/Kitware/ninja/releases/download/v1.8.2.g3bbbe.kitware.ninja/ninja-1.8.2.g3bbbe.kitware.dyndep-1.jobserver-1_x86_64-linux-gnu.tar.gz"
2. 1.jobserver-1/ninja-1.8.2.g3bbbe.kitware.dyndep-1.jobserver-1_x86_64-linux-gnu.tar.gz
3. $ curl -Ls ${ninja_url} | tar -xz -C ninja --strip-components=1
4. $ export PATH=$HOME/Deps/ninja${PATH:+:$PATH}
```

Windows上，使用MSYS2环境(假设是64位版本)执行以下命令：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-ninja
```

NOTE: 我们建议阅读这篇文章 <http://www.aosabook.org/en/posa/ninja.html>，里面是对NINJA编译器的历史和设计的选择，进行启发性的讨论。

0.3.4 Python

本书主要关于CMake，但是其中的一些方法，需要使用Python。因此，也需要对Python进行安装：解释器、头文件和库。Python 2.7的生命周期结束于2020年，因此我们将使用Python 3.5。

在Ubuntu 14.04 LTS上(这是Travis CI使用的环境，我们后面会讨论)，Python 3.5可以安装如下：

```
1. sudo apt-get install python3.5-dev
```

Windows可使用MSYS2环境，Python安装方法如下(假设是64位版本)：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-python3
2. $ pacman -S mingw64/mingw-w64-x86_64-python3-pip
3. $ python3 -m pip install pipenv
```

为了运行已经写好的测试机制，还需要一些特定的Python模块。可以使用包管理器在系统范围内安装这些包，也可以在隔离的环境中安装。建议采用后一种方法：

- 可以在不影响系统环境的情况下，将安装包进行清理/安装。
- 可以在没有管理员权限的情况下安装包。
- 可以降低软件版本和依赖项冲突的风险。
- 为了复现性，可以更好地控制包的依赖性。

为此，我们准备了一个 `Pipfile` 。结合 `pipfile.lock`，可以使用 `Pipenv` (<http://pipenv.readthedocs>)。创建一个独立的环境，并安装所有包。要为示例库创建此环境，可在库的顶层目录中运行以下命令：

```
1. $ pip install --user pip pipenv --upgrade
2. $ pipenv install --python python3.5
```

执行 `pipenv shell` 命令会进入一个命令行环境，其中包含特定版本的Python和可用的包。执行 `exit` 将退出当前环境。当然，还可以使用 `pipenv run` 在隔离的环境中直接执行命令。

或者，可以将库中的 `requirements.txt` 文件与 `Virtualenv` (<http://docs.pythonguide.org/en/latest/dev/virtualenvs/>)和 `pip` 结合使用，以达到相同的效果：

```
1. $ virtualenv --python=python3.5 venv
2. $ source venv/bin/activate
3. $ pip install -r requirements.txt
```

可以使用 `deactivate` 命令退出虚拟环境。

另一种选择是使用 `Conda` 环境，我们建议安装 `Miniconda`。将把最新的 `Miniconda` 安装到 GNU/Linux 的 `$HOME/Deps/conda` 目录(从 https://repo.continuum.io/miniconda/miniconda3-latestlinux-x86_64.sh 下载)或 macOS(从 https://repo.continuum.io/miniconda/miniconda3-latestmacosx-x86_64.sh 下载)：

```

1. $ curl -Ls https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-
2. x86_64.sh > miniconda.sh
3. $ bash miniconda.sh -b -p "$HOME"/Deps/conda &> /dev/null
4. $ touch "$HOME"/Deps/conda/conda-meta/pinned
5. $ export PATH=$HOME/Deps/conda/bin${PATH:+:$PATH}
6. $ conda config --set show_channel_urls True
7. $ conda config --set changeps1 no
8. $ conda update --all
9. $ conda clean -tipy

```

Windows 上，可以从 https://repo.continuum.io/miniconda/Miniconda3-latest-Windows-x86_64.exe 下载最新的 `Miniconda`。该软件包可以使用 `PowerShell` 安装，如下：

```

1. $basedir = $pwd.Path + "\"
2. $filepath = $basedir + "Miniconda3-latest-Windows-x86_64.exe"
3. $Anaconda_loc = "C:\Deps\conda"
4. $args = "/InstallationType=JustMe /AddToPath=0 /RegisterPython=0 /S
   /D=$Anaconda_loc"
5. Start-Process -FilePath $filepath -ArgumentList $args -Wait -Passthru
6. $conda_path = $Anaconda_loc + "\Scripts\conda.exe"
7. $args = "config --set show_channel_urls True"
8. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
9. $args = "config --set changeps1 no"
10. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
11. $args = "update --all"
12. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
13. $args = "clean -tipy"
14. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru

```

安装了 `Conda` 后，Python 模块可以按如下方式安装：

```

1. $ conda create -n cmake-cookbook python=3.5
2. $ conda activate cmake-cookbook
3. $ conda install --file requirements.txt

```

执行 `conda deactivate` 将退出 `conda` 的环境。

0.3.5 依赖软件

有些示例需要额外的依赖，这些软件将在这里介绍。

0.3.5.1 BLAS和LAPACK

大多数Linux发行版都为BLAS和LAPACK提供包。例如，在Ubuntu 14.04 LTS上，您可以运行以下命令：

```
1. $ sudo apt-get install libatlas-dev liblapack-dev liblapacke-dev
```

macOS上，XCode附带的加速库可以满足我们的需要。

Windows使用MSYS2环境，可以按如下方式安装这些库(假设是64位版本)：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-openblas
```

或者，可以从GitHub (<https://github.com/Referlapack/lapack>) 下载BLAS和LAPACK的参考实现，并从源代码编译库。商业供应商为平台提供安装程序，安装包中有BLAS和LAPACK相关的API。

0.3.5.2 消息传递接口(MPI)

MPI有许多商业和非商业实现。这里，安装免费的非商业实现就足够了。在Ubuntu 14.04 LTS上，我们推荐 `OpenMPI`。可使用以下命令安装：

```
1. $ sudo apt-get install openmpi-bin libopenmpi-dev
```

在macOS上，`Homebrew` 发布了 `MPICH`：

```
1. $ brew install mpich
```

还可以从 <https://www.open-mpi.org/software/> 上获取源代码，编译 `OpenMPI`。对于Windows，Microsoft MPI可以通过 [https://msdn.microsoft.com/en-us/library/bb524831\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx) 下载安装。

0.3.5.3 线性代数模板库

一些示例需要线性代数模板库，版本为3.3或更高。如果包管理器不提供 `Eigen`，可以使用在线打包

源(<http://eigen.tuxfamily.org>)安装它。例如，在GNU/Linux和macOS上，可以将 `Eigen` 安装到 `$HOME/Deps/Eigen` 目录：

```

1. $ eigen_version="3.3.4"
2. $ mkdir -p eigen
   $ curl -Ls http://bitbucket.org/eigen/eigen/get/${eigen_version}.tar.gz | tar -
3. xz -C eigen --strip-components=1
4. $ cd eigen
5. $ cmake -H. -Bbuild_eigen -
6. DCMAKE_INSTALL_PREFIX="$HOME/Deps/eigen" &> /dev/null
7. $ cmake --build build_eigen -- install &> /dev/null

```

0.3.5.4 Boost库

`Boost` 库适用于各种操作系统，大多数Linux发行版都通过它们的包管理器提供该库的安装。例如，在Ubuntu 14.04 LTS上，`Boost` 文件系统库、`Boost Python` 库和 `Boost` 测试库可以通过以下命令安装：

```

$ sudo apt-get install libboost-filesystem-dev libboost-python-dev libboost-
1. test-dev

```

对于macOS，`MacPorts` 和自制程序都为最新版本的 `Boost` 提供了安装包。我们在macOS上的测试设置安装 `Boost` 如下：

```

1. $ brew cask uninstall --force oclint
2. $ brew uninstall --force --ignore-dependencies boost
3. $ brew install boost
4. $ brew install boost-python3

```

Windows的二进制发行版也可以从 `Boost` 网站 <http://www.boost.org> 下载。或者，可以从 <https://www.boost.org> 下载源代码，并自己编译 `Boost` 库。

0.3.5.5 交叉编译器

在类Debian/Ubuntu系统上，可以使用以下命令安装交叉编译器：

```

1. $ sudo apt-get install gcc-mingw-w64 g++-mingw-w64 gfortran-mingw-w64

```

在macOS上，使用 `Brew`，可以安装以下交叉编译器：

```

1. $ brew install mingw-w64

```

其他包管理器提供相应的包。使用打包的跨编译器的另一种方法，是使用M交叉环境(<https://mxe.cc>)，并从源代码对其进行构建。

0.3.5.6 ZeroMQ, pkg-config, UUID和Doxygen

Ubuntu 14.04 LTS上，这些包可以安装如下：

```
1. $ sudo apt-get install pkg-config libzmq3-dev doxygen graphviz-dev uuid-dev
```

macOS上，我们建议使用 `Brew` 安装：

```
1. $ brew install ossp-uuid pkg-config zeromq doxygen
```

`pkg-config` 程序和 `UUID` 库只在类Unix系统上可用。Windows上使用MSYS2环境，可以按如下方式安装这些依赖项(假设是64位版本)：

```
1. $ pacman -S mingw64/mingw-w64-x86_64-zeromq
2. $ pacman -S mingw64/mingw-w64-x86_64-pkg-config
3. $ pacman -S mingw64/mingw-w64-x86_64-doxygen
4. $ pacman -S mingw64/mingw-w64-x86_64-graphviz
```

0.3.5.7 Conda的构建和部署

想要使用 `Conda` 打包的示例的话，需要 `Miniconda` 和 `Conda` 构建和部署工具。`Miniconda` 的安装说明之前已经给出。要在GNU/Linux和macOS上安装 `Conda` 构建和部署工具，请运行以下命令：

```
1. $ conda install --yes --quiet conda-build anaconda-client jinja2 setuptools
2. $ conda clean -tipsy
3. $ conda info -a
```

这些工具也可以安装在Windows上：

```
1. $conda_path = "C:\Deps\conda\Scripts\conda.exe"
2. $args = "install --yes --quiet conda-build anaconda-client jinja2 setuptools"
3. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
4. $args = "clean -tipsy"
5. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
6. $args = "info -a"
7. Start-Process -FilePath "$conda_path" -ArgumentList $args -Wait -Passthru
```

0.4 测试环境

示例在下列持续集成(CI)上进行过测试：

- Travis(<https://travis-ci.org>)用于GNU/Linux和macOS
- Appveyor(<https://www.appveyor.com>)用于Windows
- CircleCI (<https://circleci.com>)用于附加的GNU/Linux测试和商业编译器

CI服务的配置文件可以在示例库中找到(<https://github.com/dev-cafe/cmake-cookbook/>)：

- Travis的配置文件为 `travis.yml`
- Appveyor的配置文件为 `.appveyor.yml`
- CircleCI的配置文件为 `.circleci/config.yml`
- Travis和Appveyor的其他安装脚本，可以在 `testing/dependencies` 文件夹中找到。

NOTE: GNU/Linux系统上，Travis使用CMake 3.5.2和CMake 3.12.1对实例进行测试。macOS系统上用CMake 3.12.1进行测试。Appveyor使用CMake 3.11.3进行测试。Circle使用CMake 3.12.1进行测试。

测试机制是一组Python脚本，包含在 `testing` 文件夹中。脚本 `collect_tests.py` 将运行测试并报告它们的状态。示例也可以单独测试，也可以批量测试；`collect_tests.py` 接受正则表达式作为命令行输入，例如：

```
1. $ pipenv run python testing/collect_tests.py 'chapter-0[1,7]/recipe-0[1,2,5]'
```

该命令将对第1章和第7章的示例1、2和5进行测试。输出的示例如下：

```
$ pipenv run python testing/collect_tests.py 'chapter-01/recipe-05'
recipe: Presenting options to the user

/home/bast/tmp/cmake-cookbook/chapter-01/recipe-05/cxx-example
CMake Warning:
Manually-specified variables were not used by the project:

  CMAKE_C_COMPILER
  CMAKE_Fortran_COMPILER

configuring ... OK
building configuration Debug ... OK
```

要获得更详细的输出，可以设置环境变量 `VERBOSE_OUTPUT=ON` :

```
$ env VERBOSE_OUTPUT=ON pipenv run python testing/collect_tests.py 'chapter-  
1. */recipe-*'
```

0.5 上报问题并提出改进建议

请将遇到的问题反馈到 <https://github.com/dev-cafe/cmake-cookbook/issues>。

要对源码库进行贡献，我们建议对原始库 <https://github.com/dev-cafe/cmake-cookbook> 进行Fork，并使用Pull Request提交更改，可以参考这个页面
<https://help.github.com/articles/creating-a-pull-request-from-a-fork/>。

对于非重要更改，我们建议在发送Pull Request之前，首先在

<https://github.com/devcafe/cmake-cookbook/issues> 上创建一个问题进行描述，并讨论所要更改的问题。

第1章 从可执行文件到库

本章的主要内容有：

- 将单个源码文件编译为可执行文件
- 切换生成器
- 构建和连接静态库与动态库
- 用条件语句控制编译
- 向用户显示选项
- 指定编译器
- 切换构建类型
- 设置编译器选项
- 为语言设定标准
- 使用控制流进行构造

本章的示例将指导您完成构建代码所需的基本任务：编译可执行文件、编译库、根据用户输入执行构建操作等等。CMake是一个构建系统生成器，特别适合于独立平台和编译器。除非另有说明，否则所有配置都独立于操作系统，它们可以在GNU/Linux、macOS和Windows的系统下运行。

本书的示例主要为C++项目设计，并使用C++示例进行了演示，但CMake也可以用于其他语言的项目，包括C和Fortran。我们会尝试一些有意思的配置，其中包含了一些C++、C和Fortran语言示例。您可以根据自己喜好，选择性了解。有些示例是定制的，以突出在选择特定语言时需要面临的挑战。

1.1 将单个源文件编译为可执行文件

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-01> 中找到，包含C++、C和Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本节示例中，我们将演示如何运行CMake配置和构建一个简单的项目。该项目由单个源文件组成，用于生成可执行文件。我们将用C++讨论这个项目，您在GitHub示例库中可以找到C和Fortran的例子。

准备工作

我们希望将以下源代码编译为单个可执行文件：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4.
5. std::string say_hello() { return std::string("Hello, CMake world!"); }
6.
7. int main() {
8.     std::cout << say_hello() << std::endl;
9.     return EXIT_SUCCESS;
10. }
```

具体实施

除了源文件之外，我们还需要向CMake提供项目配置描述。该描述使用CMake完成，完整的文档可以在 <https://cmake.org/cmake/help/latest/> 找到。我们把CMake指令放入一个名为 `CMakeLists.txt` 的文件中。

NOTE: 文件的名称区分大小写，必须命名为 `CMakeLists.txt`，CMake才能够解析。

具体步骤如下：

1. 用编辑器打开一个文本文件，将这个文件命名为 `CMakeLists.txt`。
2. 第一行，设置CMake所需的最低版本。如果使用的CMake版本低于该版本，则会发出致命错误：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
```

3. 第二行，声明了项目的名称(`recipe-01`)和支持的编程语言(CXX代表C++)：

```
1. project(recipe-01 LANGUAGES CXX)
```

4. 指示CMake创建一个新目标：可执行文件 `hello-world`。这个可执行文件是通过编译和链接源文件 `hello-world.cpp` 生成的。CMake将为编译器使用默认设置，并自动选择生成工具：

```
1. add_executable(hello-world hello-world.cpp)
```

5. 将该文件与源文件 `hello-world.cpp` 放在相同的目录中。记住，它只能被命名为 `CMakeLists.txt`。

6. 现在，可以通过创建 `build` 目录，在 `build` 目录下来配置项目：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- The CXX compiler identification is GNU 8.1.0
6. -- Check for working CXX compiler: /usr/bin/c++
7. -- Check for working CXX compiler: /usr/bin/c++ -- works
8. -- Detecting CXX compiler ABI info
9. -- Detecting CXX compiler ABI info - done
10. -- Detecting CXX compile features
11. -- Detecting CXX compile features - done
12. -- Configuring done
13. -- Generating done
   -- Build files have been written to: /home/user/cmake-cookbook/chapter-
14. 01/recipe-01/cxx-example/build
```

7. 如果一切顺利，项目的配置已经在 `build` 目录中生成。我们现在可以编译可执行文件：

```
1. $ cmake --build .
2.
3. Scanning dependencies of target hello-world
4. [ 50%] Building CXX object CMakeFiles/hello-world.dir/hello-world.cpp.o
5. [100%] Linking CXX executable hello-world
6. [100%] Built target hello-world
```

工作原理

示例中，我们使用了一个简单的 `CMakeLists.txt` 来构建“Hello world”可执行文件：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES CXX)
3. add_executable(hello-world hello-world.cpp)
```

NOTE: CMake语言不区分大小写，但是参数区分大小写。

TIPS: CMake中，C++是默认的编程语言。不过，我们还是建议使用 `LANGUAGES` 选项在 `project` 命令中显式地声明项目的语言。

要配置项目并生成构建器，我们必须通过命令行界面(CLI)运行CMake。CMake CLI提供了许多选项，`cmake -help` 将输出以显示列出所有可用选项的完整帮助信息，我们将在书中对这些选项进行更多地了解。正如您将从 `cmake -help` 的输出中显示的内容，它们中的大多数选项会让您访问CMake手册，查看详细信息。通过下列命令生成构建器：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
```

这里，我们创建了一个目录 `build` (生成构建器的位置)，进入 `build` 目录，并通过指定 `CMakeLists.txt` 的位置(本例中位于父目录中)来调用CMake。可以使用以下命令行来实现相同的效果：

```
1. $ cmake -H. -Bbuild
```

该命令是跨平台的，使用了 `-H` 和 `-B` 为CLI选项。`-H` 表示当前目录中搜索根 `CMakeLists.txt` 文件。`-Bbuild` 告诉CMake在一个名为 `build` 的目录中生成所有的文件。

NOTE: `cmake -H. -Bbuild` 也属于CMake标准使用方式：

<https://cmake.org/pipermail/cmake-developers/2018-January/030520.html>。不过，我们将在本书中使用传统方法(创建一个构建目录，进入其中，并通过将CMake指向 `CMakeLists.txt` 的位置来配置项目)。

运行 `cmake` 命令会输出一系列状态消息，显示配置信息：

```
1. $ cmake ..
2.
3. -- The CXX compiler identification is GNU 8.1.0
4. -- Check for working CXX compiler: /usr/bin/c++
5. -- Check for working CXX compiler: /usr/bin/c++ -- works
6. -- Detecting CXX compiler ABI info
```

```

7. -- Detecting CXX compiler ABI info - done
8. -- Detecting CXX compile features
9. -- Detecting CXX compile features - done
10. -- Configuring done
11. -- Generating done
    -- Build files have been written to: /home/user/cmake-cookbook/chapter-
12. 01/recipe-01/cxx-example/build

```

NOTE: 在与 `CMakeLists.txt` 相同的目录中执行 `cmake .`，原则上足以配置一个项目。然而，CMake会将所有生成的文件写到项目的根目录中。这将是一个源代码内构建，通常是不推荐的，因为这会混合源代码和项目的目录树。我们首选的是源外构建。

CMake是一个构建系统生成器。将描述构建系统(如: Unix Makefile、Ninja、Visual Studio等)应当如何操作才能编译代码。然后，CMake为所选的构建系统生成相应的指令。默认情况下，在GNU/Linux和macOS系统上，CMake使用Unix Makefile生成器。Windows上，Visual Studio是默认的生成器。在下一个示例中，我们将进一步研究生成器，并在第13章中重新讨论生成器。

GNU/Linux上，CMake默认生成Unix Makefile来构建项目：

- `Makefile` : `make` 将运行指令来构建项目。
- `CMakefile` : 包含临时文件的目录，CMake用于检测操作系统、编译器等。此外，根据所选的生成器，它还包含特定的文件。
- `cmake_install.cmake` : 处理安装规则的CMake脚本，在项目安装时使用。
- `CMakeCache.txt` : 如文件名所示，CMake缓存。CMake在重新运行配置时使用这个文件。

要构建示例项目，我们运行以下命令：

```
1. $ cmake --build .
```

最后，CMake不强制指定构建目录执行名称或位置，我们完全可以把它放在项目路径之外。这样做同样有效：

```

1. $ mkdir -p /tmp/someplace
2. $ cd /tmp/someplace
3. $ cmake /path/to/source
4. $ cmake --build .

```

更多信息

官方文档 <https://cmake.org/runningcmake/> 给出了运行CMake的简要概述。由CMake生成的构建系统，即上面给出的示例中的Makefile，将包含为给定项目构建目标文件、可执行文件和库的目

标及规则。`hello-world` 可执行文件是在当前示例中的唯一目标，运行以下命令：

```
1. $ cmake --build . --target help
2.
3. The following are some of the valid targets for this Makefile:
4. ... all (the default if no target is provided)
5. ... clean
6. ... depend
7. ... rebuild_cache
8. ... hello-world
9. ... edit_cache
10. ... hello-world.o
11. ... hello-world.i
12. ... hello-world.s
```

CMake生成的目标比构建可执行文件的目标要多。可以使用 `cmake --build . --target <target-name>` 语法，实现如下功能：

- **all**(或Visual Studio generator中的ALL_BUILD)是默认目标，将在项目中构建所有目标。
- **clean**，删除所有生成的文件。
- **rebuild_cache**，将调用CMake为源文件生成依赖(如果有的话)。
- **edit_cache**，这个目标允许直接编辑缓存。

对于更复杂的项目，通过测试阶段和安装规则，CMake将生成额外的目标：

- **test**(或Visual Studio generator中的RUN_TESTS)将在CTest的帮助下运行测试套件。我们将在第4章中详细讨论测试和CTest。
- **install**，将执行项目安装规则。我们将在第10章中讨论安装规则。
- **package**，此目标将调用CPack为项目生成可分发的包。打包和CPack将在第11章中讨论。

1.2 切换生成器

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-02> 中找到，其中有一个C++、C和Fortran示例。该配置在CMake 3.5版(或更高版本)下测试没问题，并且已经在GNU/Linux、macOS和Windows上进行了测试。

CMake是一个构建系统生成器，可以使用单个CMakeLists.txt为不同平台上的不同工具集配置项目。您可以在CMakeLists.txt中描述构建系统必须运行的操作，以配置并编译代码。基于这些指令，CMake将为所选的构建系统(Unix Makefile、Ninja、Visual Studio等等)生成相应的指令。我们将在第13章中重新讨论生成器。

准备工作

CMake针对不同平台支持本地构建工具列表。同时支持命令行工具(如Unix Makefile和Ninja)和集成开发环境(IDE)工具。用以下命令，可在平台上找到生成器名单，以及已安装的CMake版本：

```
1. $ cmake --help
```

这个命令的输出，将列出CMake命令行界面上所有的选项，您会找到可用生成器的列表。例如，安装了CMake 3.11.2的GNU/Linux机器上的输出：

```
1. Generators
2. The following generators are available on this platform:
3. Unix Makefiles = Generates standard UNIX makefiles.
4. Ninja = Generates build.ninja files.
5. Watcom WMake = Generates Watcom WMake makefiles.
6. CodeBlocks - Ninja = Generates CodeBlocks project files.
7. CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
8. CodeLite - Ninja = Generates CodeLite project files.
9. CodeLite - Unix Makefiles = Generates CodeLite project files.
10. Sublime Text 2 - Ninja = Generates Sublime Text 2 project files.
11. Sublime Text 2 - Unix Makefiles = Generates Sublime Text 2 project files.
12. Kate - Ninja = Generates Kate project files.
13. Kate - Unix Makefiles = Generates Kate project files.
14. Eclipse CDT4 - Ninja = Generates Eclipse CDT 4.0 project files.
15. Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
```

使用此示例，我们将展示为项目切换生成器是多么**EASY**。

具体实施

我们将重用前一节示例中的 `hello-world.cpp` 和 `CMakeLists.txt`。惟一的区别在使用CMake时，因为现在必须显式地使用命令行方式，用 `-G` 切换生成器。

1. 首先，使用以下步骤配置项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake -G Ninja ..
4.
5. -- The CXX compiler identification is GNU 8.1.0
6. -- Check for working CXX compiler: /usr/bin/c++
7. -- Check for working CXX compiler: /usr/bin/c++ -- works
8. -- Detecting CXX compiler ABI info
9. -- Detecting CXX compiler ABI info - done
10. -- Detecting CXX compile features
11. -- Detecting CXX compile features - done
12. -- Configuring done
13. -- Generating done
    -- Build files have been written to: /home/user/cmake-cookbook/chapter-
14. 01/recipe-02/cxx-exampl

```

2. 第二步，构建项目：

```

1. $ cmake --build .
2.
3. [2/2] Linking CXX executable hello-world

```

如何工作

与前一个配置相比，每一步的输出没什么变化。每个生成器都有自己的文件集，所以编译步骤的输出和构建目录的内容是不同的：

- `build.ninja` 和 `rules.ninja`：包含Ninja的所有构建语句和构建规则。
- `CMakeCache.txt`：CMake会在这个文件中进行缓存，与生成器无关。
- `CMakeFiles`：包含由CMake在配置期间生成的临时文件。
- `cmake_install.cmake`：CMake脚本处理安装规则，并在安装时使用。

`cmake --build .` 将 `ninja` 命令封装在一个跨平台的接口中。

更多信息

我们将在第13章中讨论可选生成器和交叉编译。

要了解关于生成器的更多信息，CMake官方文档是一个很好的选择：

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

1.3 构建和链接静态库和动态库

NOTE: 这个示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-03> 找到，其中有C++和Fortran示例。该配置在CMake 3.5版(或更高版本)测试有效的，并且已经在GNU/Linux、macOS和Windows上进行了测试。

项目中会有单个源文件构建的多个可执行文件的可能。项目中有多个源文件，通常分布在不同子目录中。这种实践有助于项目的源代码结构，而且支持模块化、代码重用和关注点分离。同时，这种分离可以简化并加速项目的重新编译。本示例中，我们将展示如何将源代码编译到库中，以及如何链接这些库。

准备工作

回看第一个例子，这里并不再为可执行文件提供单个源文件，我们现在将引入一个类，用来包装要打印到屏幕上的消息。更新一下的 `hello-world.cpp`：

```

1. #include "Message.hpp"
2.
3. #include <cstdlib>
4. #include <iostream>
5.
6. int main() {
7.     Message say_hello("Hello, CMake World!");
8.     std::cout << say_hello << std::endl;
9.
10.    Message say_goodbye("Goodbye, CMake World");
11.    std::cout << say_goodbye << std::endl;
12.
13.    return EXIT_SUCCESS;
14. }
```

`Message` 类包装了一个字符串，并提供重载过的 `<<` 操作，并且包括两个源码文件：`Message.hpp` 头文件与 `Message.cpp` 源文件。`Message.hpp` 中的接口包含以下内容：

```

1. #pragma once
2.
3. #include <iostream>
4. #include <string>
5.
6. class Message {
```

```

7. public:
8.     Message(const std::string &m) : message_(m) {}
9.     friend std::ostream &operator<<(std::ostream &os, Message &obj) {
10.        return obj.printObject(os);
11.    }
12. private:
13.     std::string message_;
14.     std::ostream &printObject(std::ostream &os);
15. };

```

`Message.cpp` 实现如下：

```

1. #include "Message.hpp"
2.
3. #include <iostream>
4. #include <string>
5.
6. std::ostream &Message::printObject(std::ostream &os) {
7.     os << "This is my very nice message: " << std::endl;
8.     os << message_;
9.     return os;
10. }

```

具体实施

这里有两个文件需要编译，所以 `CMakeLists.txt` 必须进行修改。本例中，先把它们编译成一个库，而不是直接编译成可执行文件：

1. 创建目标—静态库。库的名称和源码文件名相同，具体代码如下：

```

1. add_library(message
2.   STATIC
3.   Message.hpp
4.   Message.cpp
5. )

```

2. 创建 `hello-world` 可执行文件的目标部分不需要修改：

```
1. add_executable(hello-world hello-world.cpp)
```

3. 最后，将目标库链接到可执行目标：

```
1. target_link_libraries(hello-world message)
```

4. 对项目进行配置和构建。库编译完成后，将连接到 `hello-world` 可执行文件中：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5.
6. Scanning dependencies of target message
7. [ 25%] Building CXX object CMakeFiles/message.dir/Message.cpp.o
8. [ 50%] Linking CXX static library libmessage.a
9. [ 50%] Built target message
10. Scanning dependencies of target hello-world
11. [ 75%] Building CXX object CMakeFiles/hello-world.dir/hello-world.cpp.o
12. [100%] Linking CXX executable hello-world
13. [100%] Built target hello-world
```

```
1. $ ./hello-world
2.
3. This is my very nice message:
4. Hello, CMake World!
5. This is my very nice message:
6. Goodbye, CMake World
```

工作原理

本节引入了两个新命令：

- `add_library(message STATIC Message.hpp Message.cpp)`：生成必要的构建指令，将指定的源码编译到库中。`add_library` 的第一个参数是目标名。整个 `CMakeLists.txt` 中，可使用相同的名称来引用库。生成的库的实际名称将由CMake通过在前面添加前缀 `lib` 和适当的扩展名作为后缀来形成。生成库是根据第二个参数(`STATIC` 或 `SHARED`)和操作系统确定的。
- `target_link_libraries(hello-world message)`：将库链接到可执行文件。此命令还确保 `hello-world` 可执行文件可以正确地依赖于消息库。因此，在消息库链接到 `hello-world` 可执行文件之前，需要完成消息库的构建。

编译成功后，构建目录包含 `libmessage.a` 一个静态库(在GNU/Linux上)和 `hello-world` 可执行文件。

CMake接受其他值作为 `add_library` 的第二个参数的有效值，我们来看下本书会用到的值：

- **STATIC**: 用于创建静态库，即编译文件的打包存档，以便在链接其他目标时使用，例如：可执行文件。
- **SHARED**: 用于创建动态库，即可以动态链接，并在运行时加载的库。可以在 `CMakeLists.txt` 中使用 `add_library(message SHARED Message.hpp Message.cpp)` 从静态库切换到动态共享对象(DSO)。
- **OBJECT**: 可将给定 `add_library` 的列表中的源码编译到目标文件，不将它们归档到静态库中，也不能将它们链接到共享对象中。如果需要一次性创建静态库和动态库，那么使用对象库尤其有用。我们将在本示例中演示。
- **MODULE**: 又为DSO组。与 `SHARED` 库不同，它们不链接到项目中的任何目标，不过可以进行动态加载。该参数可以用于构建运行时插件。

CMake还能够生成特殊类型的库，这不会在构建系统中产生输出，但是对于组织目标之间的依赖关系，和构建需求非常有用：

- **IMPORTED**: 此类库目标表示位于项目外部的库。此类库的主要用途是，对现有依赖项进行构建。因此，`IMPORTED` 库将被视为不可变的。我们将在本书的其他章节演示使用 `IMPORTED` 库的示例。参见：
<https://cmake.org/cmake/help/latest/manual/cmakebuildsystem.7.html#imported-targets>
- **INTERFACE**: 与 `IMPORTED` 库类似。不过，该类型库可变，没有位置信息。它主要用于项目之外的目标构建使用。我们将在本章第5节中演示 `INTERFACE` 库的示例。参见：
<https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#interface-libraries>
- **ALIAS**: 顾名思义，这种库为项目中已存在的库目标定义别名。不过，不能为 `IMPORTED` 库选择别名。参见：<https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html#alias-libraries>

本例中，我们使用 `add_library` 直接集合了源代码。后面的章节中，我们将使用 `target_sources` 汇集源码，特别是在第7章。请参见Craig Scott的这篇精彩博文：
https://crascit.com/2016/01/31/enhanced-source-file-handling-with-target_sources/，其中有对 `target_sources` 命令的具体使用。

更多信息

现在展示 `OBJECT` 库的使用，修改 `CMakeLists.txt`，如下：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES CXX)
3.
```

```

4. add_library(message-objs
5.     OBJECT
6.         Message.hpp
7.         Message.cpp
8.     )
9.
10. # this is only needed for older compilers
11. # but doesn't hurt either to have it
12. set_target_properties(message-objs
13.     PROPERTIES
14.         POSITION_INDEPENDENT_CODE 1
15.     )
16.
17. add_library(message-shared
18.     SHARED
19.     ${TARGET_OBJECTS: message-objs}
20.     )
21.
22. add_library(message-static
23.     STATIC
24.     ${TARGET_OBJECTS: message-objs}
25.     )
26.
27. add_executable(hello-world hello-world.cpp)
28.
29. target_link_libraries(hello-world message-static)

```

首先，`add_library` 改为 `add_library(Message-objs OBJECT Message.hpp Message.cpp)`。此外，需要保证编译的目标文件与生成位置无关。可以通过使用 `set_target_properties` 命令，设置 `message-objs` 目标的相应属性来实现。

NOTE: 可能在某些平台和/或使用较老的编译器上，需要显式地为目标设置 `POSITION_INDEPENDENT_CODE` 属性。

现在，可以使用这个对象库来获取静态库(`message-static`)和动态库(`message-shared`)。要注意引用对象库的生成器表达式语法： `${TARGET_OBJECTS: message-objs}`。生成器表达式是CMake在生成时(即配置之后)构造，用于生成特定于配置的构建输出。参见：
<https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html>。我们将在第5章中深入研究生成器表达式。最后，将 `hello-world` 可执行文件链接到消息库的静态版本。

是否可以让CMake生成同名的两个库？换句话说，它们都可以被称为 `message`，而不是 `message-`

`static` 和 `message-share` d吗？我们需要修改这两个目标的属性：

```
1. add_library(message-shared
2.     SHARED
3.     ${TARGET_OBJECTS:message-objs}
4. )
5.
6. set_target_properties(message-shared
7.     PROPERTIES
8.     OUTPUT_NAME "message"
9. )
10.
11. add_library(message-static
12.     STATIC
13.     ${TARGET_OBJECTS:message-objs}
14. )
15.
16. set_target_properties(message-static
17.     PROPERTIES
18.     OUTPUT_NAME "message"
19. )
```

我们可以链接到DSO吗？这取决于操作系统和编译器：

1. GNU/Linux和macOS上，不管选择什么编译器，它都可以工作。
2. Windows上，不能与Visual Studio兼容，但可以与MinGW和MSYS2兼容。

这是为什么呢？生成好的DSO组需要程序员限制符号的可见性。需要在编译器的帮助下实现，但不同的操作系统和编译器上，约定不同。CMake有一个机制来处理这个问题，我们将在第10章中解释它如何工作。

1.4 用条件句控制编译

NOTE: 这个示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-04> 找到，其中有一个C++示例。该配置在CMake 3.5版(或更高版本)测试有效的，并且已经在GNU/Linux、macOS和Windows上进行了测试。

目前为止，看到的示例比较简单，CMake执行流是线性的：从一组源文件到单个可执行文件，也可以生成静态库或动态库。为了确保完全控制构建项目、配置、编译和链接所涉及的所有步骤的执行流，CMake提供了自己的语言。本节中，我们将探索条件结构 `if-else- else-endif` 的使用。

NOTE: CMake语言相当庞杂，由基本的控制结构、特定于CMake的命令和使用新函数模块化扩展语言的基础设施组成。完整的概览可以在这里找到：

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

具体实施

从与上一个示例的的源代码开始，我们希望能够在不同的两种行为之间进行切换：

1. 将 `Message.hpp` 和 `Message.cpp` 构建成一个库(静态或动态)，然后将生成库链接到 `hello-world` 可执行文件中。
2. 将 `Message.hpp`，`Message.cpp` 和 `hello-world.cpp` 构建成一个可执行文件，但不生成任何一个库。

让我们来看看如何使用 `CMakeLists.txt` 来实现：

1. 首先，定义最低CMake版本、项目名称和支持的语言：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-04 LANGUAGES CXX)
```

2. 我们引入了一个新变量 `USE_LIBRARY`，这是一个逻辑变量，值为 `OFF`。我们还打印了它的值：

```
1. set(USE_LIBRARY OFF)
2.
3. message(STATUS "Compile sources into a library? ${USE_LIBRARY}")
```

3. CMake中定义 `BUILD_SHARED_LIBS` 全局变量，并设置为 `OFF`。调用 `add_library` 并省略第二个参数，将构建一个静态库：

```
1. set(BUILD_SHARED_LIBS OFF)
```

4. 然后，引入一个变量 `_sources`，包括 `Message.hpp` 和 `Message.cpp`：

```
1. list(APPEND _sources Message.hpp Message.cpp)
```

5. 然后，引入一个基于 `USE_LIBRARY` 值的 `if-else` 语句。如果逻辑为真，则 `Message.hpp` 和 `Message.cpp` 将被打包成一个库：

```
1. if(USE_LIBRARY)
2.   # add_library will create a static library
3.   # since BUILD_SHARED_LIBS is OFF
4.   add_library(message ${_sources})
5.   add_executable(hello-world hello-world.cpp)
6.   target_link_libraries(hello-world message)
7. else()
8.   add_executable(hello-world hello-world.cpp ${_sources})
9. endif()
```

6. 我们可以再次使用相同的命令集进行构建。由于 `USE_LIBRARY` 为 `OFF`，`hello-world` 可执行文件将使用所有源文件来编译。可以通过在GNU/Linux上，运行 `objdump -x` 命令进行验证。

工作原理

我们介绍了两个变量：`USE_LIBRARY` 和 `BUILD_SHARED_LIBS`。这两个变量都设置为 `OFF`。如CMake语言文档中描述，逻辑真或假可以用多种方式表示：

- 如果将逻辑变量设置为以下任意一种：`1`、`ON`、`YES`、`true`、`Y` 或非零数，则逻辑变量为 `true`。
- 如果将逻辑变量设置为以下任意一种：`0`、`OFF`、`NO`、`false`、`N`、`IGNORE`、`NOTFOUND`、空字符串，或者以 `-NOTFOUND` 为后缀，则逻辑变量为 `false`。

`USE_LIBRARY` 变量将在第一个和第二个行为之间切换。`BUILD_SHARED_LIBS` 是CMake的一个全局标志。因为CMake内部要查询 `BUILD_SHARED_LIBS` 全局变量，所以 `add_library` 命令可以在不传递 `STATIC/SHARED/OBJECT` 参数的情况下调用；如果为 `false` 或未定义，将生成一个静态库。

这个例子说明，可以引入条件来控制CMake中的执行流。但是，当前的设置不允许从外部切换，不需要手动修改 `CMakeLists.txt`。原则上，我们希望能够向用户开放所有设置，这样就可以在不修改构建代码的情况下调整配置，稍后将展示如何做到这一点。

NOTE: `else()` 和 `endif()` 中的 `()`，可能会让刚开始学习CMake代码的同学感到惊讶。其历史原因是，因为其能够指出指令的作用范围。例如，可以使用 `if(USE_LIBRARY)...else(USE_LIBRARY)...endif(USE_LIBIRAY)`。这个格式并不唯一，可以根据个人喜好来决定使用哪种格式。

TIPS: `_sources` 变量是一个局部变量，不应该在当前范围之外使用，可以在名称前加下划线。

1.5 向用户显示选项

NOTE: 这个示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-05> 找到，其中有一个C++示例。该配置在CMake 3.5版(或更高版本)测试有效的，并且已经在GNU/Linux、macOS和Windows上进行了测试。

前面的配置中，我们引入了条件句：通过硬编码的方式给定逻辑变量值。不过，这会影响用户修改这些变量。CMake代码没有向读者传达，该值可以从外部进行修改。推荐在 `CMakeLists.txt` 中使用 `option()` 命令，以选项的形式显示逻辑开关，用于外部设置，从而切换构建系统的生成行为。本节的示例将向您展示，如何使用这个命令。

具体实施

看一下前面示例中的静态/动态库示例。与其硬编码 `USE_LIBRARY` 为 `ON` 或 `OFF`，现在为其设置一个默认值，同时也可以从外部进行更改：

- 用一个选项替换上一个示例的 `set(USE_LIBRARY OFF)` 命令。该选项将修改 `USE_LIBRARY` 的值，并设置其默认值为 `OFF`：

```
1. option(USE_LIBRARY "Compile sources into a library" OFF)
```

- 现在，可以通过CMake的 `-D` CLI选项，将信息传递给CMake来切换库的行为：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake -D USE_LIBRARY=ON ..
4.
5. -- ...
6. -- Compile sources into a library? ON
7. -- ...
8.
9. $ cmake --build .
10.
11. Scanning dependencies of target message
12. [ 25%] Building CXX object CMakeFiles/message.dir/Message.cpp.o
13. [ 50%] Linking CXX static library libmessage.a
14. [ 50%] Built target message
15. Scanning dependencies of target hello-world
16. [ 75%] Building CXX object CMakeFiles/hello-world.dir/hello-world.cpp.o
17. [100%] Linking CXX executable hello-world
```

```
18. [100%] Built target hello-world
```

-D 开关用于为CMake设置任何类型的变量：逻辑变量、路径等等。

工作原理

`option` 可接受三个参数：

```
option(<option_variable> "help string" [initial value])
```

- `<option_variable>` 表示该选项的变量的名称。
- `"help string"` 记录选项的字符串，在CMake的终端或图形用户界面中可见。
- `[initial value]` 选项的默认值，可以是 `ON` 或 `OFF`。

更多信息

有时选项之间会有依赖的情况。示例中，我们提供生成静态库或动态库的选项。但是，如果没有将 `USE_LIBRARY` 逻辑设置为 `ON`，则此选项没有任何意义。CMake提供 `cmake_dependent_option()` 命令用来定义依赖于其他选项的选项：

```
1. include(CMakeDependentOption)
2.
3. # second option depends on the value of the first
4. cmake_dependent_option(
5.     MAKE_STATIC_LIBRARY "Compile sources into a static library" OFF
6.     "USE_LIBRARY" ON
7. )
8.
9. # third option depends on the value of the first
10. cmake_dependent_option(
11.     MAKE_SHARED_LIBRARY "Compile sources into a shared library" ON
12.     "USE_LIBRARY" ON
13. )
```

如果 `USE_LIBRARY` 为 `ON`，`MAKE_STATIC_LIBRARY` 默认值为 `OFF`，否则 `MAKE_SHARED_LIBRARY` 默认值为 `ON`。可以这样运行：

```
1. $ cmake -D USE_LIBRARY=OFF -D MAKE_SHARED_LIBRARY=ON ..
```

这仍然不会构建库，因为 `USE_LIBRARY` 仍然为 `OFF`。

CMake有适当的机制，通过包含模块来扩展其语法和功能，这些模块要么是CMake自带的，要么是定制的。本例中，包含了一个名为 `CMakeDependentOption` 的模块。如果没有 `include` 这个模块，`cmake_dependent_option()` 命令将不可用。参见
<https://cmake.org/cmake/help/latest/module/CMakeDependentOption.html>

TIPS: 手册中的任何模块都可以以命令行的方式使用 `cmake --help-module <name-of-module>`。例如，`cmake --help-module CMakeDependentOption` 将打印刚才讨论的模块的手册页(帮助页面)。

1.6 指定编译器

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-06> 中找到，其中有一个C++/C示例。该配置在CMake 3.5版(或更高版本)下测试没问题，并且已经在GNU/Linux、macOS和Windows上进行了测试。

目前为止，我们还没有过多考虑如何选择编译器。CMake可以根据平台和生成器选择编译器，还能将编译器标志设置为默认值。然而，我们通常控制编译器的选择。在后面的示例中，我们还将考虑构建类型的选择，并展示如何控制编译器标志。

具体实施

如何选择一个特定的编译器？例如，如果想使用Intel或Portland Group编译器怎么办？CMake将语言的编译器存储在 `CMAKE_<LANG>_COMPILER` 变量中，其中 `<LANG>` 是受支持的任何一种语言，对于我们的目的是 `CXX`、`C` 或 `Fortran`。用户可以通过以下两种方式之一设置此变量：

1. 使用CLI中的 `-D` 选项，例如：

```
1. $ cmake -D CMAKE_CXX_COMPILER=clang++ ..
```

2. 通过导出环境变量 `CXX` (C++编译器)、`CC` (C编译器)和 `FC` (Fortran编译器)。例如，使用这个命令使用 `clang++` 作为 `C++` 编译器：

```
1. $ env CXX=clang++ cmake ..
```

到目前为止讨论的示例，都可以通过传递适当的选项，配置合适的编译器。

NOTE: CMake了解运行环境，可以通过其CLI的 `-D` 开关或环境变量设置许多选项。前一种机制覆盖后一种机制，但是我们建议使用 `-D` 显式设置选项。显式优于隐式，因为环境变量可能被设置为不适合(当前项目)的值。

我们在这里假设，其他编译器在标准路径中可用，CMake在标准路径中执行查找编译器。如果不是这样，用户将需要将完整的编译器可执行文件或包装器路径传递给CMake。

TIPS: 我们建议使用 `-D CMAKE_<LANG>_COMPILER` CLI选项设置编译器，而不是导出 `CXX`、`CC` 和 `FC`。这是确保跨平台并与非POSIX兼容的唯一方法。为了避免变量污染环境，这些变量可能会影响与项目一起构建的外部库环境。

工作原理

配置时，CMake会进行一系列平台测试，以确定哪些编译器可用，以及它们是否适合当前的项目。一个合适的编译器不仅取决于我们所使用的平台，还取决于我们想要使用的生成器。CMake执行的第一个测试基于项目语言的编译器的名称。例如，`cc` 是一个工作的 `C` 编译器，那么它将用作 `C` 项目的默认编译器。GNU/Linux上，使用Unix Makefile或Ninja时，GCC家族中的编译器很可能 是 `C++`、`C` 和 `Fortran` 的默认选择。Microsoft Windows上，将选择Visual Studio中的 `C++` 和 `C` 编译器(前提是Visual Studio是生成器)。如果选择MinGW或MSYS Makefile作为生成器，则默认使用MinGW编译器。

更多信息

我们的平台上的CMake，在哪里可以找到可用的编译器和编译器标志？CMake提供 `--system-information` 标志，它将把关于系统的所有信息转储到屏幕或文件中。要查看这个信息，请尝试以下操作：

```
1. $ cmake --system-information information.txt
```

文件中(本例中是 `information.txt`)可以看到 `CMAKE_CXX_COMPILER`、`CMAKE_C_COMPILER` 和 `CMAKE_Fortran_COMPILER` 的默认值，以及默认标志。我们将在下一个示例中看到相关的标志。

CMake提供了额外的变量来与编译器交互：

- `CMAKE_<LANG>_COMPILER_LOADED`：如果为项目启用了语言 `<LANG>`，则将设置为 `TRUE`。
- `CMAKE_<LANG>_COMPILER_ID`：编译器标识字符串，编译器供应商所特有。例如，`GCC` 用于 GNU编译器集合，`AppleClang` 用于macOS上的Clang，`MSVC` 用于Microsoft Visual Studio编译器。注意，不能保证为所有编译器或语言定义此变量。
- `CMAKE_COMPILER_IS_GNU<LANG>`：如果语言 `<LANG>` 是GNU编译器集合的一部分，则将此逻辑变量设置为 `TRUE`。注意变量名的 `<LANG>` 部分遵循GNU约定：C语言为 `CC`，C++语言为 `CXX`，Fortran语言为 `G77`。
- `CMAKE_<LANG>_COMPILER_VERSION`：此变量包含一个字符串，该字符串给定语言的编译器版本。版本信息在 `major[.minor[.patch[.tweak]]]` 中给出。但是，对于 `CMAKE_<LANG>_COMPILER_ID`，不能保证所有编译器或语言都定义了此变量。

我们可以尝试使用不同的编译器，配置下面的示例 `CMakeLists.txt`。这个例子中，我们将使用CMake变量来探索已使用的编译器(及版本)：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-06 LANGUAGES C CXX)
3.
4. message(STATUS "Is the C++ compiler loaded? ${CMAKE_CXX_COMPILER_LOADED}")
```

```

5. if(CMAKE_CXX_COMPILER_LOADED)
6.   message(STATUS "The C++ compiler ID is: ${CMAKE_CXX_COMPILER_ID}")
7.   message(STATUS "Is the C++ from GNU? ${CMAKE_COMPILER_IS_GNUCXX}")
8.   message(STATUS "The C++ compiler version is:
9.   ${CMAKE_CXX_COMPILER_VERSION}")
10. endif()
11. message(STATUS "Is the C compiler loaded? ${CMAKE_C_COMPILER_LOADED}")
12. if(CMAKE_C_COMPILER_LOADED)
13.   message(STATUS "The C compiler ID is: ${CMAKE_C_COMPILER_ID}")
14.   message(STATUS "Is the C from GNU? ${CMAKE_COMPILER_IS_GNUCC}")
15.   message(STATUS "The C compiler version is: ${CMAKE_C_COMPILER_VERSION}")
16. endif()

```

注意，这个例子不包含任何目标，没有要构建的东西，我们只关注配置步骤：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Is the C++ compiler loaded? 1
7. -- The C++ compiler ID is: GNU
8. -- Is the C++ from GNU? 1
9. -- The C++ compiler version is: 8.1.0
10. -- Is the C compiler loaded? 1
11. -- The C compiler ID is: GNU
12. -- Is the C from GNU? 1
13. -- The C compiler version is: 8.1.0
14. ...

```

当然，输出将取决于可用和已选择的编译器(及版本)。

1.7 切换构建类型

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-07> 中找到，包含一个C++/C示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

CMake可以配置构建类型，例如：Debug、Release等。配置时，可以为Debug或Release构建设置相关的选项或属性，例如：编译器和链接器标志。控制生成构建系统使用的配置变量是 `CMAKE_BUILD_TYPE`。该变量默认为空，CMake识别的值为：

1. **Debug**: 用于在没有优化的情况下，使用带有调试符号构建库或可执行文件。
2. **Release**: 用于构建的优化的库或可执行文件，不包含调试符号。
3. **RelWithDebInfo**: 用于构建较少的优化库或可执行文件，包含调试符号。
4. **MinSizeRel**: 用于不增加目标代码大小的优化方式，来构建库或可执行文件。

具体实施

示例中，我们将展示如何为项目设置构建类型：

1. 首先，定义最低CMake版本、项目名称和支持的语言：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-07 LANGUAGES C CXX)
```

2. 然后，设置一个默认的构建类型(本例中是Release)，并打印一条消息。要注意的是，该变量被设置为缓存变量，可以通过缓存进行编辑：

```
1. if(NOT CMAKE_BUILD_TYPE)
2.   set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
3. endif()
4. message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")
```

3. 最后，打印出CMake设置的相应编译标志：

```
1. message(STATUS "C flags, Debug configuration: ${CMAKE_C_FLAGS_DEBUG}")
2. message(STATUS "C flags, Release configuration: ${CMAKE_C_FLAGS_RELEASE}")
  message(STATUS "C flags, Release configuration with Debug info:
3. ${CMAKE_C_FLAGS_RELWITHDEBINFO}")
  message(STATUS "C flags, minimal Release configuration:
4. ${CMAKE_C_FLAGS_MINSIZEREL}")
```

```

5. message(STATUS "C++ flags, Debug configuration: ${CMAKE_CXX_FLAGS_DEBUG}")
   message(STATUS "C++ flags, Release configuration:
6. ${CMAKE_CXX_FLAGS_RELEASE}")
   message(STATUS "C++ flags, Release configuration with Debug info:
7. ${CMAKE_CXX_FLAGS_RELWITHDEBINFO}")
   message(STATUS "C++ flags, minimal Release configuration:
8. ${CMAKE_CXX_FLAGS_MINSIZEREL}")

```

4. 验证配置的输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Build type: Release
7. -- C flags, Debug configuration: -g
8. -- C flags, Release configuration: -O3 -DNDEBUG
9. -- C flags, Release configuration with Debug info: -O2 -g -DNDEBUG
10. -- C flags, minimal Release configuration: -Os -DNDEBUG
11. -- C++ flags, Debug configuration: -g
12. -- C++ flags, Release configuration: -O3 -DNDEBUG
13. -- C++ flags, Release configuration with Debug info: -O2 -g -DNDEBUG
14. -- C++ flags, minimal Release configuration: -Os -DNDEBUG

```

5. 切换构建类型：

```

1. $ cmake -D CMAKE_BUILD_TYPE=Debug ..
2.
3. -- Build type: Debug
4. -- C flags, Debug configuration: -g
5. -- C flags, Release configuration: -O3 -DNDEBUG
6. -- C flags, Release configuration with Debug info: -O2 -g -DNDEBUG
7. -- C flags, minimal Release configuration: -Os -DNDEBUG
8. -- C++ flags, Debug configuration: -g
9. -- C++ flags, Release configuration: -O3 -DNDEBUG
10. -- C++ flags, Release configuration with Debug info: -O2 -g -DNDEBUG
11. -- C++ flags, minimal Release configuration: -Os -DNDEBUG

```

工作原理

我们演示了如何设置默认构建类型，以及如何(从命令行)覆盖它。这样，就可以控制项目，是使用优化，还是关闭优化启用调试。我们还看到了不同配置使用了哪些标志，这主要取决于选择的编译器。需要在运行CMake时显式地打印标志，也可以仔细阅读运行 `CMake --system-information` 的输出，以了解当前平台、默认编译器和语言的默认组合是什么。下一个示例中，我们将讨论如何为不同的编译器和不同的构建类型，扩展或调整编译器标志。

更多信息

我们展示了变量 `CMAKE_BUILD_TYPE`，如何切换生成构建系统的配置(这个链接中有说明：https://cmake.org/cmake/help/v3.5/variable/CMAKE_BUILD_TYPE.html)。Release 和Debug配置中构建项目通常很有用，例如：评估编译器优化级别的效果。对于单配置生成器，如Unix Makefile、MSYS Makefile或Ninja，因为要对项目重新配置，这里需要运行CMake两次。不过，CMake也支持复合配置生成器。这些通常是集成开发环境提供的项目文件，最显著的是Visual Studio和Xcode，它们可以同时处理多个配置。可以使用 `CMAKE_CONFIGURATION_TYPES` 变量可以对这些生成器的可用配置类型进行调整，该变量将接受一个值列表(可从这个链接获得文档：https://cmake.org/cmake/help/v3.5/variable/CMAKE_CONFIGURATION_TYPES.html)。

下面是对Visual Studio的CMake调用：

```
1. $ mkdir -p build
2. $ cd build
   $ cmake .. -G"Visual Studio 12 2017 Win64" -D
3. CMAKE_CONFIGURATION_TYPES="Release;Debug"
```

将为Release和Debug配置生成一个构建树。然后，您可以使 `--config` 标志来决定构建这两个中的哪一个：

```
1. $ cmake --build . --config Release
```

NOTE:当使用单配置生成器开发代码时，为*Release*版和*Debug*创建单独的构建目录，两者使用相同的源代码。这样，就可以在两者之间切换，而不用重新配置和编译。

1.8 设置编译器选项

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-08> 中找到，有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例展示了如何探测CMake，从而获得关于编译器的信息，以及如何切换项目中的编译器。后一个任务是控制项目的编译器标志。CMake为调整或扩展编译器标志提供了很大的灵活性，您可以选择下面两种方法：

- CMake将编译选项视为目标属性。因此，可以根据每个目标设置编译选项，而不需要覆盖CMake默认值。
- 可以使用 `-D` CLI标志直接修改 `CMAKE_<LANG>_FLAGS_<CONFIG>` 变量。这将影响项目中的所有目标，并覆盖或扩展CMake默认值。

本示例中，我们将展示这两种方法。

准备工作

编写一个示例程序，计算不同几何形状的面积，`computer_area.cpp`：

```

1. #include "geometry_circle.hpp"
2. #include "geometry_polygon.hpp"
3. #include "geometry_rhombus.hpp"
4. #include "geometry_square.hpp"
5.
6. #include <cstdlib>
7. #include <iostream>
8.
9. int main() {
10.     using namespace geometry;
11.
12.     double radius = 2.5293;
13.     double A_circle = area::circle(radius);
14.     std::cout << "A circle of radius " << radius << " has an area of " <<
15.     A_circle
16.             << std::endl;
17.     int nSides = 19;
18.     double side = 1.29312;
19.     double A_polygon = area::polygon(nSides, side);

```

```

20.     std::cout << "A regular polygon of " << nSides << " sides of length " << side
21.             << " has an area of " << A_polygon << std::endl;
22.
23.     double d1 = 5.0;
24.     double d2 = 7.8912;
25.     double A_rhombus = area::rhombus(d1, d2);
26.     std::cout << "A rhombus of major diagonal " << d1 << " and minor diagonal "
27.             << d2
28.                 << " has an area of " << A_rhombus << std::endl;
29.
30.     double l = 10.0;
31.     double A_square = area::square(l);
32.     std::cout << "A square of side " << l << " has an area of " << A_square
33.             << std::endl;
34.
35.     return EXIT_SUCCESS;
36. }
```

函数的各种实现分布在不同的文件中，每个几何形状都有一个头文件和源文件。总共有4个头文件和5个源文件要编译：

```

1. .
2. └─ CMakeLists.txt
3. └─ compute-areas.cpp
4. └─ geometry_circle.cpp
5. └─ geometry_circle.hpp
6. └─ geometry_polygon.cpp
7. └─ geometry_polygon.hpp
8. └─ geometry_rhombus.cpp
9. └─ geometry_rhombus.hpp
10. └─ geometry_square.cpp
11. └─ geometry_square.hpp
```

我们不会为所有文件提供清单，读者可以参考 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-08>。

具体实施

现在已经有了源代码，我们的目标是配置项目，并使用编译器标志进行实验：

1. 设置CMake的最低版本：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
```

2. 声明项目名称和语言：

```
1. project(recipe-08 LANGUAGES CXX)
```

3. 然后，打印当前编译器标志。CMake将对所有C++目标使用这些：

```
1. message("C++ compiler flags: ${CMAKE_CXX_FLAGS}")
```

4. 为目標准备了标志列表，其中一些将无法在Windows上使用：

```
1. list(APPEND flags "-fPIC" "-Wall")
2. if(NOT WIN32)
3.   list(APPEND flags "-Wextra" "-Wpedantic")
4. endif()
```

5. 添加了一个新的目標——`geometry` 库，并列出它的源依赖关系：

```
1. add_library(geometry
2.   STATIC
3.   geometry_circle.cpp
4.   geometry_circle.hpp
5.   geometry_polygon.cpp
6.   geometry_polygon.hpp
7.   geometry_rhombus.cpp
8.   geometry_rhombus.hpp
9.   geometry_square.cpp
10.  geometry_square.hpp
11. )
```

6. 为这个库目標设置了编译选项：

```
1. target_compile_options(geometry
2.   PRIVATE
3.   ${flags}
4. )
```

7. 然后，将生成 `compute-areas` 可执行文件作为一个目標：

```
1. add_executable(compute-areas compute-areas.cpp)
```

8. 还为可执行目标设置了编译选项：

```
1. target_compile_options(compute-areas
2.   PRIVATE
3.   "-fPIC"
4. )
```

9. 最后，将可执行文件链接到geometry库：

```
1. target_link_libraries(compute-areas geometry)
```

如何工作

本例中，警告标志有 `-Wall`、`-Wextra` 和 `-Wpedantic`，将这些标志添加到 `geometry` 目标的编译选项中；`compute-areas` 和 `geometry` 目标都将使用 `-fPIC` 标志。编译选项可以添加三个级别的可见性：`INTERFACE`、`PUBLIC` 和 `PRIVATE`。

可见性的含义如下：

- **PRIVATE**，编译选项会应用于给定的目标，不会传递给与目标相关的目标。我们的示例中，即使 `compute-areas` 将链接到 `geometry` 库，`compute-areas` 也不会继承 `geometry` 目标上设置的编译器选项。
- **INTERFACE**，给定的编译选项将只应用于指定目标，并传递给与目标相关的目标。
- **PUBLIC**，编译选项将应用于指定目标和使用它的目标。

目标属性的可见性CMake的核心，我们将在本书中经常讨论这个话题。以这种方式添加编译选项，不会影响全局CMake变量 `CMAKE_<LANG>_FLAGS_<CONFIG>`，并能更细粒度控制在哪些目标上使用哪些选项。

我们如何验证，这些标志是否按照我们的意图正确使用呢？或者换句话说，如何确定项目在CMake构建时，实际使用了哪些编译标志？一种方法是，使用CMake将额外的参数传递给本地构建工具。本例中会设置环境变量 `VERBOSE=1`：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build . -- VERBOSE=1
5.
6. ... lots of output ...
```

```

7.
8. [ 14%] Building CXX object CMakeFiles/geometry.dir/geometry_circle.cpp.o
   /usr/bin/c++ -fPIC -Wall -Wextra -Wpedantic -o
   CMakeFiles/geometry.dir/geometry_circle.cpp.o -c /home/bast/tmp/cmake-
9. cookbook/chapter-01/recipe-08/cxx-example/geometry_circle.cpp
10. [ 28%] Building CXX object CMakeFiles/geometry.dir/geometry_polygon.cpp.o
    /usr/bin/c++ -fPIC -Wall -Wextra -Wpedantic -o
    CMakeFiles/geometry.dir/geometry_polygon.cpp.o -c /home/bast/tmp/cmake-
11. cookbook/chapter-01/recipe-08/cxx-example/geometry_polygon.cpp
12. [ 42%] Building CXX object CMakeFiles/geometry.dir/geometry_rhombus.cpp.o
    /usr/bin/c++ -fPIC -Wall -Wextra -Wpedantic -o
    CMakeFiles/geometry.dir/geometry_rhombus.cpp.o -c /home/bast/tmp/cmake-
13. cookbook/chapter-01/recipe-08/cxx-example/geometry_rhombus.cpp
14. [ 57%] Building CXX object CMakeFiles/geometry.dir/geometry_square.cpp.o
    /usr/bin/c++ -fPIC -Wall -Wextra -Wpedantic -o
    CMakeFiles/geometry.dir/geometry_square.cpp.o -c /home/bast/tmp/cmake-
15. cookbook/chapter-01/recipe-08/cxx-example/geometry_square.cpp
16.
17. ... more output ...
18.
19. [ 85%] Building CXX object CMakeFiles/compute-areas.dir/compute-areas.cpp.o
   /usr/bin/c++ -fPIC -o CMakeFiles/compute-areas.dir/compute-areas.cpp.o -c
20. /home/bast/tmp/cmake-cookbook/chapter-01/recipe-08/cxx-example/compute-areas.cpp
21.
22. ... more output ...

```

输出确认编译标志，确认指令设置正确。

控制编译器标志的第二种方法，不用对 `CMakeLists.txt` 进行修改。如果想在这个项目中修改 `geometry` 和 `compute-areas` 目标的编译器选项，可以使用CMake参数进行配置：

```
1. $ cmake -D CMAKE_CXX_FLAGS="-fno-exceptions -fno-rtti" ..
```

这个命令将编译项目，禁用异常和运行时类型标识(RTTI)。

也可以使用全局标志，可以使用 `CMakeLists.txt` 运行以下命令：

```
1. $ cmake -D CMAKE_CXX_FLAGS="-fno-exceptions -fno-rtti" ..
```

这将使用 `-fno-rtti - fpic - wall - Wextra - wpedantic` 配置 `geometry` 目标，同时使用 `-fno exception -fno-rtti - fpic` 配置 `compute-areas`。

NOTE:本书中，我们推荐为每个目标设置编译器标志。使用 `target_compile_options()` 不仅允许对

编译选项进行细粒度控制，而且还可以更好地与CMake的更高级特性进行集成。

更多信息

大多数时候，编译器有特性标示。当前的例子只适用于 `GCC` 和 `Clang`；其他供应商的编译器不确定是否会理解(如果不是全部)这些标志。如果项目是真正跨平台，那么这个问题就必须得到解决，有三种方法可以解决这个问题。

最典型的方法是将所需编译器标志列表附加到每个配置类型CMake变

量 `CMAKE_<LANG>_FLAGS_<CONFIG>`。标志确定设置为给定编译器有效的标志，因此将包含在 `if-endif` 子句中，用于检查 `CMAKE_<LANG>_COMPILER_ID` 变量，例如：

```

1. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
2.   list(APPEND CMAKE_CXX_FLAGS "-fno-rtti" "-fno-exceptions")
    list(APPEND CMAKE_CXX_FLAGS_DEBUG "-Wsuggest-final-types" "-Wsuggest-final-
3. methods" "-Wsuggest-override")
4.   list(APPEND CMAKE_CXX_FLAGS_RELEASE "-O3" "-Wno-unused")
5. endif()
6. if(CMAKE_CXX_COMPILER_ID MATCHES Clang)
    list(APPEND CMAKE_CXX_FLAGS "-fno-rtti" "-fno-exceptions" "-Qunused-
7. arguments" "-fcolor-diagnostics")
8.   list(APPEND CMAKE_CXX_FLAGS_DEBUG "-Wdocumentation")
9.   list(APPEND CMAKE_CXX_FLAGS_RELEASE "-O3" "-Wno-unused")
10. endif()
```

更细粒度的方法是，不修改 `CMAKE_<LANG>_FLAGS_<CONFIG>` 变量，而是定义特定的标志列表：

```

1. set(COMPILER_FLAGS)
2. set(COMPILER_FLAGS_DEBUG)
3. set(COMPILER_FLAGS_RELEASE)
4.
5. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
6.   list(APPEND CXX_FLAGS "-fno-rtti" "-fno-exceptions")
    list(APPEND CXX_FLAGS_DEBUG "-Wsuggest-final-types" "-Wsuggest-final-methods"
7. "-Wsuggest-override")
8.   list(APPEND CXX_FLAGS_RELEASE "-O3" "-Wno-unused")
9. endif()
10.
11. if(CMAKE_CXX_COMPILER_ID MATCHES Clang)
    list(APPEND CXX_FLAGS "-fno-rtti" "-fno-exceptions" "-Qunused-arguments" "-
12. fcolor-diagnostics")
13.   list(APPEND CXX_FLAGS_DEBUG "-Wdocumentation")
```

```

14.    list(APPEND CXX_FLAGS_RELEASE "-O3" "-Wno-unused")
15. endif()

```

稍后，使用生成器表达式来设置编译器标志的基础上，为每个配置和每个目标生成构建系统：

```

1. target_compile_option(compute-areas
2. PRIVATE
3. ${CXX_FLAGS}
4. "$<${CONFIG:Debug}>:${CXX_FLAGS_DEBUG}>""
5. "$<${CONFIG:Release}>:${CXX_FLAGS_RELEASE}>""
6. )

```

当前示例中展示了这两种方法，我们推荐后者（特定于项目的变量和 `target_compile_options`）。

两种方法都有效，并在许多项目中得到广泛应用。不过，每种方式都有缺点。`CMAKE_<LANG>_COMPILER_ID` 不能保证为所有编译器都定义。此外，一些标志可能会被弃用，或者在编译器的较晚版本中引入。与 `CMAKE_<LANG>_COMPILER_ID` 类似，`CMAKE_<LANG>_COMPILER_VERSION` 变量不能保证为所有语言和供应商都提供定义。尽管检查这些变量的方式非常流行，但我们认为更健壮的替代方法是检查所需的标志集是否与给定的编译器一起工作，这样项目中实际上只使用有效的标志。结合特定于项目的变量、`target_compile_options` 和生成器表达式，会让解决方案变得非常强大。我们将在第7章的第3节中展示，如何使用 `check-and-set` 模式。

1.9 为语言设定标准

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-09> 中找到，包含一个C++和Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

编程语言有不同的标准，即提供改进的语言版本。启用新标准是通过设置适当的编译器标志来实现的。前面的示例中，我们已经展示了如何为每个目标或全局进行配置。3.1版本中，CMake引入了一个独立于平台和编译器的机制，用于为 C++ 和 C 设置语言标准：为目标设置 `<LANG>_STANDARD` 属性。

准备工作

对于下面的示例，需要一个符合 C++14 标准或更高版本的 C++ 编译器。此示例代码定义了动物的多态，我们使用 `std::unique_ptr` 作为结构中的基类：

```
1. std::unique_ptr<Animal> cat = Cat("Simon");
2. std::unique_ptr<Animal> dog = Dog("Marlowe");
```

没有为各种子类型显式地使用构造函数，而是使用工厂方法的实现。工厂方法使用 C++11 的可变参数模板实现。它包含继承层次结构中每个对象的创建函数映射：

```
1. typedef std::function<std::unique_ptr<Animal>(<const
2. std::string &>)> CreateAnimal;
```

基于预先分配的标签来分派它们，创建对象：

```
1. std::unique_ptr<Animal> simon = farm.create("CAT", "Simon");
2. std::unique_ptr<Animal> marlowe = farm.create("DOG", "Marlowe");
```

标签和创建功能在工厂使用前已注册：

```
1. Factory<CreateAnimal> farm;
   farm.subscribe("CAT", [](const std::string & n) { return std::make_unique<Cat>
2. (n); });
   farm.subscribe("DOG", [](const std::string & n) { return std::make_unique<Dog>
3. (n); });
```

使用 C++11 Lambda 函数定义创建函数，使用 `std::make_unique` 来避免引入裸指针的操作。这个

工厂函数是在 C++14 中引入。

NOTE: CMake的这一功能是在3.1版中添加的，并且还在更新。CMake的后续版本为 C++ 标准的后续版本和不同的编译器，提供了越来越好的支持。我们建议您在文档页面上检查您选择的编译器是否受支持：<https://cmake.org/cmake/help/latest/manual/cmake-compile-features.7.html#supported-compiler>

具体实施

将逐步构建 CMakeLists.txt，并展示如何设置语言标准(本例中是 C++14)：

1. 声明最低要求的CMake版本，项目名称和语言：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-09 LANGUAGES CXX)
```

2. 要求在Windows上导出所有库符号：

```
1. set(CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS ON)
```

3. 需要为库添加一个目标，这将编译源代码为一个动态库：

```
1. add_library(animals
2.   SHARED
3.     Animal.cpp
4.     Animal.hpp
5.     Cat.cpp
6.     Cat.hpp
7.     Dog.cpp
8.     Dog.hpp
9.     Factory.hpp
10. )
```

4. 现在，为目标设置了 CXX_STANDARD 、 CXX_EXTENSIONS 和 CXX_STANDARD_REQUIRED 属性。还设置了 position_independent_ent_code 属性，以避免在使用一些编译器构建DSO时出现问题：

```
1. set_target_properties(animals
2.   PROPERTIES
3.     CXX_STANDARD 14
4.     CXX_EXTENSIONS OFF
```

```

5.     CXX_STANDARD_REQUIRED ON
6.     POSITION_INDEPENDENT_CODE 1
7. )

```

5. 然后，为“动物农场”的可执行文件添加一个新目标，并设置它的属性：

```

1. add_executable(animal-farm animal-farm.cpp)
2. set_target_properties(animal-farm
3.   PROPERTIES
4.     CXX_STANDARD 14
5.     CXX_EXTENSIONS OFF
6.     CXX_STANDARD_REQUIRED ON
7. )

```

6. 最后，将可执行文件链接到库：

```
1. target_link_libraries(animal-farm animals)
```

7. 现在，来看看猫和狗都说了什么：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./animal-farm
6.
7. I'm Simon the cat!
8. I'm Marlowe the dog!

```

工作原理

步骤4和步骤5中，我们为动物和动物农场目标设置了一些属性：

- **CXX_STANDARD**会设置我们想要的标准。
- **CXX_EXTENSIONS**告诉CMake，只启用 ISO C++ 标准的编译器标志，而不使用特定编译器的扩展。
- **CXX_STANDARD_REQUIRED**指定所选标准的版本。如果这个版本不可用，CMake将停止配置并出现错误。当这个属性被设置为 OFF 时，CMake将寻找下一个标准的最新版本，直到一个合适的标志。这意味着，首先查找 C++14，然后是 C++11，然后是 C++98。（译者注：目前会从 C++20 或 C++17 开始查找）

NOTE: 本书发布时，还没有 `Fortran_STANDARD` 可用，但是可以使
用 `target_compile_options` 设置标准，可以参见：<https://github.com/devcafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-09>

TIPS: 如果语言标准是所有目标共享的全局属性，那么可以

将 `CMAKE_<LANG>_STANDARD`、`CMAKE_<LANG>_EXTENSIONS` 和 `CMAKE_<LANG>_STANDARD_REQUIREMENT` 变量设置为相应的值。所有目标上的对应属性都将使用这些设置。

更多信息

通过引入编译特性，CMake对语言标准提供了更精细的控制。这些是语言标准引入的特性，比如 `C++11` 中的可变参数模板和 `Lambda` 表达式，以及 `C++14` 中的自动返回类型推断。可以使
用 `target_compile_features()` 命令要求为特定的目标提供特定的特性，CMake将自动为标准设置
正确的编译器标志。也可以让CMake为可选编译器特性，生成兼容头文件。

TIPS: 我们建议阅读CMake在线文档，全面了解 `cmake-compile-features` 和如何处理编译特性和语
言标准：<https://cmake.org/cmake/help/latest/manual/cmake-compile-features.7.html>。

1.10 使用控制流

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-01/recipe-10> 中找到，有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本章前面的示例中，已经使用过 `if-else-endif`。CMake还提供了创建循环的语言工具：`foreach` `endforeach` 和 `while-endwhile`。两者都可以与 `break` 结合使用，以便尽早从循环中跳出。本示例将展示如何使用 `foreach`，来循环源文件列表。我们将应用这样的循环，在引入新目标的前提下，来为一组源文件进行优化降级。

准备工作

将重用第8节中的几何示例，目标是通过将一些源代码汇集到一个列表中，从而微调编译器的优化。

具体实施

下面是 `CMakeLists.txt` 中要的详细步骤：

1. 与示例8中一样，指定了CMake的最低版本、项目名称和语言，并声明了几何库目标：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-10 LANGUAGES CXX)
3. add_library(geometry
4.   STATIC
5.   geometry_circle.cpp
6.   geometry_circle.hpp
7.   geometry_polygon.cpp
8.   geometry_polygon.hpp
9.   geometry_rhombus.cpp
10.  geometry_rhombus.hpp
11.  geometry_square.cpp
12.  geometry_square.hpp
13. )
```

2. 使用 `-O3` 编译器优化级别编译库，对目标设置一个私有编译器选项：

```

1. target_compile_options(geometry
2.   PRIVATE
3.   -O3)
```

```
4.    )
```

3. 然后，生成一个源文件列表，以较低的优化选项进行编译：

```
1. list(
2.   APPEND sources_with_lower_optimization
3.   geometry_circle.cpp
4.   geometry_rhombus.cpp
5. )
```

4. 循环这些源文件，将它们的优化级别调到 `-O2`。使用它们的源文件属性完成：

```
1. message(STATUS "Setting source properties using IN LISTS syntax:")
2. foreach(_source IN LISTS sources_with_lower_optimization)
3.   set_source_files_properties(${_source} PROPERTIES COMPILE_FLAGS -O2)
4.   message(STATUS "Appending -O2 flag for ${_source}")
5. endforeach()
```

5. 为了确保设置属性，再次循环并在打印每个源文件的 `COMPILE_FLAGS` 属性：

```
1. message(STATUS "Querying sources properties using plain syntax:")
2. foreach(_source ${sources_with_lower_optimization})
3.   get_source_file_property(_flags ${_source} COMPILE_FLAGS)
4.   message(STATUS "Source ${_source} has the following extra COMPILE_FLAGS:
5.   ${_flags}")
5. endforeach()
```

6. 最后，添加 `compute-areas` 可执行目标，并将 `geometry` 库连接上去：

```
1. add_executable(compute-areas compute-areas.cpp)
2. target_link_libraries(compute-areas geometry)
```

7. 验证在配置步骤中正确设置了标志：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Setting source properties using IN LISTS syntax:
7. -- Appending -O2 flag for geometry_circle.cpp
```

```

8. -- Appending -O2 flag for geometry_rhombus.cpp
9. -- Querying sources properties using plain syntax:
10. -- Source geometry_circle.cpp has the following extra COMPILE_FLAGS: -O2
11. -- Source geometry_rhombus.cpp has the following extra COMPILE_FLAGS: -O2

```

8. 最后，还使用 `VERBOSE=1` 检查构建步骤。将看到 `-O2` 标志添加在 `-O3` 标志之后，但是最后一个优化级别标志(在本例中是 `-O2`)不同：

```
1. $ cmake --build . -- VERBOSE=1
```

工作原理

`foreach-endforeach` 语法可用于在变量列表上，表示重复特定任务。本示例中，使用它来操作、设置和获取项目中特定文件的编译器标志。CMake代码片段中引入了另外两个新命令：

- `set_source_files_properties(file PROPERTIES property value)`，它将属性设置为给定文件的传递值。与目标非常相似，文件在CMake中也有属性，允许对构建系统进行非常细粒度的控制。源文件的可用属性列表可以在这里找到：
<https://cmake.org/cmake/help/v3.5/manual/cmake-properties.7.html#source-file-properties>。
- `get_source_file_property(VAR file property)`，检索给定文件所需属性的值，并将其存储在CMake `VAR` 变量中。

NOTE: CMake中，列表是用分号分隔的字符串组。列表可以由 `list` 或 `set` 命令创建。例如，`set(var a b c d e)` 和 `list(APPEND a b c d e)` 都创建了列表 `a;b;c;d;e`。

TIPS: 为了对一组文件降低优化，将它们收集到一个单独的目标(库)中，并为这个目标显式地设置优化级别，而不是附加一个标志，这样可能会更简洁，不过在本示例中，我们的重点是 `foreach-endforeach`。

更多信息

`foreach()` 的四种使用方式：

- `foreach(loop_var arg1 arg2 ...)`：其中提供循环变量和显式项列表。当为 `sources_with_lower_optimization` 中的项打印编译器标志集时，使用此表单。注意，如果项目列表位于变量中，则必须显式展开它；也就是说， `${sources_with_lower_optimization}` 必须作为参数传递。
- 通过指定一个范围，可以对整数进行循环，例如：`foreach(loop_var range total)` 或 `foreach(loop_var range start stop [step])`。

- 对列表值变量的循环，例如：`foreach(loop_var IN LISTS [list1[...]])`。参数解释为列表，其内容就会自动展开。
- 对变量的循环，例如：`foreach(loop_var IN ITEMS [item1 [...]])`。参数的内容没有展开。

第2章 检测环境

本章的主要内容有：

- 检测操作系统
- 处理与平台相关的源码
- 处理与编译器相关的源码
- 检测处理器体系结构
- 检测处理器指令集
- 为Eigen库使能向量化

尽管CMake跨平台，但有时源代码并不是完全可移植(例如：当使用依赖于供应商的扩展时)，我们努力使源代码能够跨平台、操作系统和编译器。这个过程中会发现，有必要根据平台不同的方式配置和/或构建代码。这对于历史代码或交叉编译尤其重要，我们将在第13章中讨论这个主题。了解处理器指令集也有助于优化特定目标平台的性能。本章会介绍，检测环境的方法，并给出建议。

2.1 检测操作系统

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-01> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

CMake是一组跨平台工具。不过，了解操作系统(OS)上执行配置或构建步骤也很重要。从而与操作系统相关的CMake代码，会根据操作系统启用条件编译，或者在可用或必要时使用特定于编译器的扩展。本示例中，我们将通过一个不需要编译任何源代码的示例，演示如何使用CMake检测操作系统。为了简单起见，我们只考虑配置过程。

具体实施

我们将用一个非常简单的 `CMakeLists.txt` 进行演示：

- 首先，定义CMake最低版本和项目名称。请注意，语言是 `NONE`：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES NONE)
```

- 然后，根据检测到的操作系统信息打印消息：

```
1. if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
2.   message(STATUS "Configuring on/for Linux")
3. elseif(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
4.   message(STATUS "Configuring on/for macOS")
5. elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
6.   message(STATUS "Configuring on/for Windows")
7. elseif(CMAKE_SYSTEM_NAME STREQUAL "AIX")
8.   message(STATUS "Configuring on/for IBM AIX")
9. else()
10.   message(STATUS "Configuring on/for ${CMAKE_SYSTEM_NAME}")
11. endif()
```

测试之前，检查前面的代码块，并考虑相应系统上的具体行为。

- 现在，测试配置项目：

```
1. $ mkdir -p build
2. $ cd build
```

```
3. $ cmake ..
```

4. 关于CMake输出，这里有一行很有趣—在Linux系统上(在其他系统上，输出会不同)：

```
1. -- Configuring on/for Linux
```

工作原理

CMake为目标操作系统定义了 `CMAKE_SYSTEM_NAME`，因此不需要使用定制命令、工具或脚本来查询此信息。然后，可以使用此变量的值实现特定于操作系统的条件和解决方案。在具有 `uname` 命令的系统上，将此变量设置为 `uname -s` 的输出。该变量在macOS上设置为“Darwin”。在Linux和Windows上，它分别计算为“Linux”和“Windows”。我们了解了如何在特定的操作系统上执行特定的CMake代码。当然，应该尽量减少这种定制化行为，以便简化迁移到新平台的过程。

NOTE: 为了最小化从一个平台转移到另一个平台时的成本，应该避免直接使用Shell命令，还应该避免显式的路径分隔符(*Linux*和*macOS*上的前斜杠和*Windows*上的后斜杠)。*CMake*代码中只使用前斜杠作为路径分隔符，*CMake*将自动将它们转换为所涉及的操作系统环境。

2.2 处理与平台相关的源代码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-02> 中找到，包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

理想情况下，应该避免依赖于平台的源代码，但是有时我们没有选择，特别是当要求配置和编译不是自己编写的代码时。本示例中，将演示如何使用CMake根据操作系统编译源代码。

准备工作

修改 `hello-world.cpp` 示例代码，将第1章第1节的例子进行修改：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4.
5. std::string say_hello() {
6. #ifdef IS_WINDOWS
7.     return std::string("Hello from Windows!");
8. #elif IS_LINUX
9.     return std::string("Hello from Linux!");
10. #elif IS_MACOS
11.    return std::string("Hello from macOS!");
12. #else
13.    return std::string("Hello from an unknown system!");
14. #endif
15. }
16.
17. int main() {
18.     std::cout << say_hello() << std::endl;
19.     return EXIT_SUCCESS;
20. }
```

具体实施

完成一个 `CMakeLists.txt` 实例，使我们能够基于目标操作系统有条件地编译源代码：

- 首先，设置了CMake最低版本、项目名称和支持的语言：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-02 LANGUAGES CXX)

```

2. 然后，定义可执行文件及其对应的源文件：

```

1. add_executable(hello-world hello-world.cpp)

```

3. 通过定义以下目标编译定义，让预处理器知道系统名称：

```

1. if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
2.   target_compile_definitions(hello-world PUBLIC "IS_LINUX")
3. endif()
4. if(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
5.   target_compile_definitions(hello-world PUBLIC "IS_MACOS")
6. endif()
7. if(CMAKE_SYSTEM_NAME STREQUAL "Windows")
8.   target_compile_definitions(hello-world PUBLIC "IS_WINDOWS")
9. endif()

```

继续之前，先检查前面的表达式，并考虑在不同系统上有哪些行为。

4. 现在，准备测试它，并配置项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./hello-world
6.
7. Hello from Linux!

```

Windows系统上，将看到来自Windows的Hello。其他操作系统将产生不同的输出。

工作原理

`hello-world.cpp` 示例中，有趣的部分是基于预处理器定义 `IS_WINDOWS`、`IS_LINUX` 或 `IS_MACOS` 的条件编译：

```

1. std::string say_hello() {
2. #ifdef IS_WINDOWS
3.   return std::string("Hello from Windows!");

```

```

4. #elif IS_LINUX
5.     return std::string("Hello from Linux!");
6. #elif IS_MACOS
7.     return std::string("Hello from macOS!");
8. #else
9.     return std::string("Hello from an unknown system!");
10. #endif
11. }
```

这些定义在CMakeLists.txt中配置时定义，通过使用 `target_compile_definition` 在预处理阶段使用。可以不重复 `if-endif` 语句，以更紧凑的表达式实现，我们将在下一个示例中演示这种重构方式。也可以把 `if-endif` 语句加入到一个 `if-else-else-endif` 语句中。这个阶段，可以使 `add_definitions(-DIS_LINUX)` 来设置定义(当然，可以根据平台调整定义)，而不是使用 `target_compile_definition`。使用 `add_definitions` 的缺点是，会修改编译整个项目的定义，而 `target_compile_definitions` 给我们机会，将定义限制于一个特定的目标，以及通过 `PRIVATE|PUBLIC|INTERFACE` 限定符，限制这些定义可见性。第1章的第8节，对这些限定符有详细的说明：

- **PRIVATE**，编译定义将只应用于给定的目标，而不应用于相关的其他目标。
- **INTERFACE**，对给定目标的编译定义将只应用于使用它的目标。
- **PUBLIC**，编译定义将应用于给定的目标和使用它的所有其他目标。

NOTE: 将项目中的源代码与平台相关性最小化，可使移植更加容易。

2.3 处理与编译器相关的源代码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-03> 中找到，包含一个C++和Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

这个方法与前面的方法类似，我们将使用CMake来编译依赖于环境的条件源代码：本例将依赖于编译器。为了可移植性，我们尽量避免去编写新代码，但遇到有依赖的情况我们也要去解决，特别是当使用历史代码或处理编译器依赖工具，如sanitizers。从这一章和前一章的示例中，我们已经掌握了实现这一目标的所有方法。尽管如此，讨论与编译器相关的源代码的处理问题还是很有用的，这样我们将有机会从另一方面了解CMake。

准备工作

本示例中，我们将从 C++ 中的一个示例开始，稍后我们将演示一个 Fortran 示例，并尝试重构和简化CMake代码。

看一下 hello-world.cpp 源代码：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4.
5. std::string say_hello() {
6. #ifdef IS_INTEL_CXX_COMPILER
7.     // only compiled when Intel compiler is selected
8.     // such compiler will not compile the other branches
9.     return std::string("Hello Intel compiler!");
10. #elif IS_GNU_CXX_COMPILER
11.     // only compiled when GNU compiler is selected
12.     // such compiler will not compile the other branches
13.     return std::string("Hello GNU compiler!");
14. #elif IS_PGI_CXX_COMPILER
15.     // etc.
16.     return std::string("Hello PGI compiler!");
17. #elif IS_XL_CXX_COMPILER
18.     return std::string("Hello XL compiler!");
19. #else
20.     return std::string("Hello unknown compiler - have we met before?");

```

```

21. #endif
22. }
23.
24. int main() {
25.     std::cout << say_hello() << std::endl;
26.     std::cout << "compiler name is " COMPILER_NAME << std::endl;
27.     return EXIT_SUCCESS;
28. }
```

Fortran 示例(`hello-world.F90`):

```

1. program hello
2.
3.     implicit none
4. #ifdef IS_Intel_FORTRAN_COMPILER
5.     print *, 'Hello Intel compiler!'
6. #elif IS_GNU_FORTRAN_COMPILER
7.     print *, 'Hello GNU compiler!'
8. #elif IS_PGI_FORTRAN_COMPILER
9.     print *, 'Hello PGI compiler!'
10. #elif IS_XL_FORTRAN_COMPILER
11.    print *, 'Hello XL compiler!'
12. #else
13.    print *, 'Hello unknown compiler - have we met before?'
14. #endif
15.
16. end program
```

具体实施

我们将从 **C++** 的例子开始，然后再看 **Fortran** 的例子：

1. `CMakeLists.txt` 文件中，定义了CMake最低版本、项目名称和支持的语言：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES CXX)
```

2. 然后，定义可执行目标及其对应的源文件：

```
1. add_executable(hello-world hello-world.cpp)
```

3. 通过定义以下目标编译定义，让预处理器了解编译器的名称和供应商：

```

target_compile_definitions(hello-world PUBLIC
1. "COMPILER_NAME=\${CMAKE_CXX_COMPILER_ID}\\")

2.

3. if(CMAKE_CXX_COMPILER_ID MATCHES Intel)
4.   target_compile_definitions(hello-world PUBLIC "IS_INTEL_CXX_COMPILER")
5. endif()
6. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
7.   target_compile_definitions(hello-world PUBLIC "IS_GNU_CXX_COMPILER")
8. endif()
9. if(CMAKE_CXX_COMPILER_ID MATCHES PGI)
10.  target_compile_definitions(hello-world PUBLIC "IS_PGI_CXX_COMPILER")
11. endif()
12. if(CMAKE_CXX_COMPILER_ID MATCHES XL)
13.  target_compile_definitions(hello-world PUBLIC "IS_XL_CXX_COMPILER")
14. endif()

```

现在我们已经可以预测结果了：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./hello-world
6.
7. Hello GNU compiler!

```

使用不同的编译器，此示例代码将打印不同的问候语。

前一个示例的 `CMakeLists.txt` 文件中的 `if` 语句似乎是重复的，我们不喜欢重复的语句。能更简洁地表达吗？当然可以！为此，让我们再来看看 `Fortran` 示例。

`Fortran` 例子的 `CMakeLists.txt` 文件中，我们需要做以下工作：

1. 需要使 `Fortran` 语言：

```
1. project(recipe-03 LANGUAGES Fortran)
```

2. 然后，定义可执行文件及其对应的源文件。在本例中，使用大写 `.F90` 后缀：

```
1. add_executable(hello-world hello-world.F90)
```

3. 我们通过定义下面的目标编译定义，让预处理器非常清楚地了解编译器：

```
1. target_compile_definitions(hello-world
2.   PUBLIC "IS_${CMAKE_Fortran_COMPILER_ID}_FORTRAN_COMPILER"
3. )
```

其余行为与 `C++` 示例相同。

工作原理

`CMakeLists.txt` 会在配置时，进行预处理定义，并传递给预处理器。`Fortran` 示例包含非常紧凑的表达式，我们使用 `CMAKE_Fortran_COMPILER_ID` 变量，通过 `target_compile_definition` 使用构造预处理器进行预处理定义。为了适应这种情况，我们必须将“Intel”从 `IS_INTEL_CXX_COMPILER` 更改为 `IS_Intel_FORTRAN_COMPILER`。通过使用相应的 `CMAKE_C_COMPILER_ID` 和 `CMAKE_CXX_COMPILER_ID` 变量，我们可以在 `C` 或 `C++` 中实现相同的效果。但是，请注意，`CMAKE_<LANG>_COMPILER_ID` 不能保证为所有编译器或语言都定义。

NOTE: 对于应该预处理的 `Fortran` 代码使用 `.F90` 后缀，对于不需要预处理的代码使用 `.f90` 后缀。

2.4 检测处理器体系结构

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-04> 中找到，包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

19世纪70年代，出现的64位整数运算和本世纪初出现的用于个人计算机的64位寻址，扩大了内存寻址范围，开发商投入了大量资源来移植为32位体系结构硬编码，以支持64位寻址。许多博客文章，如<https://www.viva64.com/en/a/0004/>，致力于讨论将 C++ 代码移植到64位平台中的典型问题和解决方案。虽然，避免显式硬编码的方式非常明智，但需要在使用CMake配置的代码中适应硬编码限制。本示例中，我们会来讨论检测主机处理器体系结构的选项。

准备工作

我们以下面的 `arch-dependent.cpp` 代码为例：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4.
5. #define STRINGIFY(x) #x
6. #define TOSTRING(x) STRINGIFY(x)
7.
8. std::string say_hello()
9. {
10.     std::string arch_info(TOSTRING(ARCHITECTURE));
11.     arch_info += std::string(" architecture. ");
12. #ifdef IS_32_BIT_ARCH
13.     return arch_info + std::string("Compiled on a 32 bit host processor.");
14. #elif IS_64_BIT_ARCH
15.     return arch_info + std::string("Compiled on a 64 bit host processor.");
16. #else
17.     return arch_info + std::string("Neither 32 nor 64 bit, puzzling ...");
18. #endif
19. }
20.
21. int main()
22. {
23.     std::cout << say_hello() << std::endl;
24.     return EXIT_SUCCESS;

```

25. }

具体实施

`CMakeLists.txt` 文件中，我们需要以下内容：

- 首先，定义可执行文件及其源文件依赖关系：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-04 LANGUAGES CXX)
3. add_executable(arch-dependent arch-dependent.cpp)
```

- 检查空指针类型的大小。CMake的 `CMAKE_SIZEOF_VOID_P` 变量会告诉我们CPU是32位还是64位。我们通过状态消息让用户知道检测到的大小，并设置预处理器定义：

```

1. if(CMAKE_SIZEOF_VOID_P EQUAL 8)
2.   target_compile_definitions(arch-dependent PUBLIC "IS_64_BIT_ARCH")
3.   message(STATUS "Target is 64 bits")
4. else()
5.   target_compile_definitions(arch-dependent PUBLIC "IS_32_BIT_ARCH")
6.   message(STATUS "Target is 32 bits")
7. endif()
```

- 通过定义以下目标编译定义，让预处理器了解主机处理器架构，同时在配置过程中打印状态消息：

```

1. if(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "i386")
2.   message(STATUS "i386 architecture detected")
3. elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "i686")
4.   message(STATUS "i686 architecture detected")
5. elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "x86_64")
6.   message(STATUS "x86_64 architecture detected")
7. else()
8.   message(STATUS "host processor architecture is unknown")
9. endif()
10. target_compile_definitions(arch-dependent
11.   PUBLIC "ARCHITECTURE=${CMAKE_HOST_SYSTEM_PROCESSOR}")
12. )
```

- 配置项目，并注意状态消息（打印出的信息可能会发生变化）：

```
1. $ mkdir -p build
```

```

2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Target is 64 bits
7. -- x86_64 architecture detected
8. ...

```

5. 最后，构建并执行代码(实际输出将取决于处理器架构)：

```

1. $ cmake --build .
2. $ ./arch-dependent
3.
4. x86_64 architecture. Compiled on a 64 bit host processor.

```

工作原理

CMake定义了 `CMAKE_HOST_SYSTEM_PROCESSOR` 变量，以包含当前运行的处理器的名称。可以设置为“i386”、“i686”、“x86_64”、“AMD64”等等，当然，这取决于当前的CPU。`CMAKE_SIZEOF_VOID_P` 为void指针的大小。我们可以在CMake配置时进行查询，以便修改目标或目标编译定义。可以基于检测到的主机处理器体系结构，使用预处理器定义，确定需要编译的分支源代码。正如在前面的示例中所讨论的，编写新代码时应该避免这种依赖，但在处理遗留代码或交叉编译时，这种依赖是有用的，交叉编译会在第13章进行讨论。

NOTE: 使用 `CMAKE_SIZEOF_VOID_P` 是检查当前CPU是否具有32位或64位架构的唯一“真正”可移植的方法。

更多信息

除了 `CMAKE_HOST_SYSTEM_PROCESSOR`，CMake还定义了 `CMAKE_SYSTEM_PROCESSOR` 变量。前者包含当前运行的CPU在CMake的名称，而后者将包含当前正在为其构建的CPU的名称。这是一个细微的差别，在交叉编译时起着非常重要的作用。我们将在第13章，看到更多关于交叉编译的内容。另一种让CMake检测主机处理器体系结构，是使用 `C` 或 `C++` 中 定义的符号，结合CMake的 `try_run` 函数，尝试构建执行的源代码(见第5.8节)分支的预处理符号。这将返回已定义错误码，这些错误可以在CMake端捕获(此策略的灵感来自

<https://github.com/axr/cmake/blob/master/targetarch.cmake>)：

```

1. #if defined(__i386) || defined(__i386__)
2.     #error cmake_arch i386

```

```

1. #elif defined(__x86_64) || defined(__x86_64__)
2.     defined(_M_X64)
3.     #error cmake_arch x86_64
4. #endif

```

这种策略也是检测目标处理器体系结构的推荐策略，因为CMake似乎没有提供可移植的内在解决方案。另一种选择，将只使用CMake，完全不使用预处理器，代价是为每种情况设置不同的源文件，然后使用 `target_source` 命令将其设置为可执行目标 `arch-dependent` 依赖的源文件：

```

1. add_executable(arch-dependent "")
2.
3. if(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "i386")
4.     message(STATUS "i386 architecture detected")
5.     target_sources(arch-dependent
6.                     PRIVATE
7.                     arch-dependent-i386.cpp
8.                 )
9. elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "i686")
10.    message(STATUS "i686 architecture detected")
11.    target_sources(arch-dependent
12.                   PRIVATE
13.                   arch-dependent-i686.cpp
14.                 )
15. elseif(CMAKE_HOST_SYSTEM_PROCESSOR MATCHES "x86_64")
16.     message(STATUS "x86_64 architecture detected")
17.     target_sources(arch-dependent
18.                     PRIVATE
19.                     arch-dependent-x86_64.cpp
20.                 )
21. else()
22.     message(STATUS "host processor architecture is unknown")
23. endif()

```

这种方法，显然需要对现有项目进行更多的工作，因为源文件需要分离。此外，不同源文件之间的代码复制肯定也会成为问题。

2.5 检测处理器指令集

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-05> 中找到，包含一个C++示例。该示例在CMake 3.10版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例中，我们将讨论如何在CMake的帮助下检测主机处理器支持的指令集。这个功能是较新版本添加到CMake中的，需要CMake 3.10或更高版本。检测到的主机系统信息，可用于设置相应的编译器标志，或实现可选的源代码编译，或根据主机系统生成源代码。本示例中，我们的目标是检测主机系统信息，使用预处理器定义将其传递给 C++ 源代码，并将信息打印到输出中。

准备工作

我们是 C++ 源码(`processor-info.cpp`)如下所示：

```

1. #include "config.h"
2.
3. #include <cstdlib>
4. #include <iostream>
5.
6. int main()
7. {
8.     std::cout << "Number of logical cores: "
9.             << NUMBER_OF_LOGICAL_CORES << std::endl;
10.    std::cout << "Number of physical cores: "
11.            << NUMBER_OF_PHYSICAL_CORES << std::endl;
12.    std::cout << "Total virtual memory in megabytes: "
13.            << TOTAL_VIRTUAL_MEMORY << std::endl;
14.    std::cout << "Available virtual memory in megabytes: "
15.            << AVAILABLE_VIRTUAL_MEMORY << std::endl;
16.    std::cout << "Total physical memory in megabytes: "
17.            << TOTAL_PHYSICAL_MEMORY << std::endl;
18.    std::cout << "Available physical memory in megabytes: "
19.            << AVAILABLE_PHYSICAL_MEMORY << std::endl;
20.    std::cout << "Processor is 64Bit: "
21.            << IS_64BIT << std::endl;
22.    std::cout << "Processor has floating point unit: "
23.            << HAS_FPU << std::endl;
24.    std::cout << "Processor supports MMX instructions: "
25.            << HAS_MMX << std::endl;
```

```

26.     std::cout << "Processor supports Ext. MMX instructions: "
27.             << HAS_MMX_PLUS << std::endl;
28.     std::cout << "Processor supports SSE instructions: "
29.             << HAS_SSE << std::endl;
30.     std::cout << "Processor supports SSE2 instructions: "
31.             << HAS_SSE2 << std::endl;
32.     std::cout << "Processor supports SSE FP instructions: "
33.             << HAS_SSE_FP << std::endl;
34.     std::cout << "Processor supports SSE MMX instructions: "
35.             << HAS_SSE_MMX << std::endl;
36.     std::cout << "Processor supports 3DNow instructions: "
37.             << HAS_AMD_3DNOW << std::endl;
38.     std::cout << "Processor supports 3DNow+ instructions: "
39.             << HAS_AMD_3DNOW_PLUS << std::endl;
40.     std::cout << "IA64 processor emulating x86 : "
41.             << HAS_IA64 << std::endl;
42.     std::cout << "OS name: "
43.             << OS_NAME << std::endl;
44.     std::cout << "OS sub-type: "
45.             << OS_RELEASE << std::endl;
46.     std::cout << "OS build ID: "
47.             << OS_VERSION << std::endl;
48.     std::cout << "OS platform: "
49.             << OS_PLATFORM << std::endl;
50.     return EXIT_SUCCESS;
51. }

```

其包含 `config.h` 头文件，我们将使用 `config.h.in` 生成这个文件。`config.h.in` 如下：

```

1. #pragma once
2.
3. #define NUMBER_OF_LOGICAL_CORES @_NUMBER_OF_LOGICAL_CORES@
4. #define NUMBER_OF_PHYSICAL_CORES @_NUMBER_OF_PHYSICAL_CORES@
5. #define TOTAL_VIRTUAL_MEMORY @_TOTAL_VIRTUAL_MEMORY@
6. #define AVAILABLE_VIRTUAL_MEMORY @_AVAILABLE_VIRTUAL_MEMORY@
7. #define TOTAL_PHYSICAL_MEMORY @_TOTAL_PHYSICAL_MEMORY@
8. #define AVAILABLE_PHYSICAL_MEMORY @_AVAILABLE_PHYSICAL_MEMORY@
9. #define IS_64BIT @_IS_64BIT@
10. #define HAS_FPU @_HAS_FPU@
11. #define HAS_MMX @_HAS_MMX@
12. #define HAS_MMX_PLUS @_HAS_MMX_PLUS@
13. #define HAS_SSE @_HAS_SSE@

```

```

14. #define HAS_SSE2 @_HAS_SSE2@
15. #define HAS_SSE_FP @_HAS_SSE_FP@
16. #define HAS_SSE_MMX @_HAS_SSE_MMX@
17. #define HAS_AMD_3DNOW @_HAS_AMD_3DNOW@
18. #define HAS_AMD_3DNOW_PLUS @_HAS_AMD_3DNOW_PLUS@
19. #define HAS_IA64 @_HAS_IA64@
20. #define OS_NAME "@_OS_NAME@"
21. #define OS_RELEASE "@_OS_RELEASE@"
22. #define OS_VERSION "@_OS_VERSION@"
23. #define OS_PLATFORM "@_OS_PLATFORM@"

```

如何实施

我们将使用CMake为平台填充 `config.h` 中的定义，并将示例源文件编译为可执行文件：

- 首先，我们定义了CMake最低版本、项目名称和项目语言：

```

1. cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
2. project(recipe-05 CXX)

```

- 然后，定义目标可执行文件及其源文件，并包括目录：

```

1. add_executable(processor-info "")
2.
3. target_sources(processor-info
4.   PRIVATE
5.     processor-info.cpp
6. )
7.
8. target_include_directories(processor-info
9.   PRIVATE
10.    ${PROJECT_BINARY_DIR}
11. )

```

- 继续查询主机系统的信息，获取一些关键字：

```

1. foreach(key
2.   IN ITEMS
3.     NUMBER_OF_LOGICAL_CORES
4.     NUMBER_OF_PHYSICAL_CORES
5.     TOTAL_VIRTUAL_MEMORY

```

```

6.      AVAILABLE_VIRTUAL_MEMORY
7.      TOTAL_PHYSICAL_MEMORY
8.      AVAILABLE_PHYSICAL_MEMORY
9.      IS_64BIT
10.     HAS_FPU
11.     HAS_MMX
12.     HAS_MMX_PLUS
13.     HAS_SSE
14.     HAS_SSE2
15.     HAS_SSE_FP
16.     HAS_SSE_MMX
17.     HAS_AMD_3DNOW
18.     HAS_AMD_3DNOW_PLUS
19.     HAS_IA64
20.     OS_NAME
21.     OS_RELEASE
22.     OS_VERSION
23.     OS_PLATFORM
24.   )
25.   cmake_host_system_information(RESULT _${key} QUERY ${key})
26. endforeach()

```

4. 定义了相应的变量后，配置 `config.h`：

```
1. configure_file(config.h.in config.h @ONLY)
```

5. 现在准备好配置、构建和测试项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./processor-info
6.
7. Number of logical cores: 4
8. Number of physical cores: 2
9. Total virtual memory in megabytes: 15258
10. Available virtual memory in megabytes: 14678
11. Total physical memory in megabytes: 7858
12. Available physical memory in megabytes: 4072
13. Processor is 64Bit: 1
14. Processor has floating point unit: 1

```

```

15. Processor supports MMX instructions: 1
16. Processor supports Ext. MMX instructions: 0
17. Processor supports SSE instructions: 1
18. Processor supports SSE2 instructions: 1
19. Processor supports SSE FP instructions: 0
20. Processor supports SSE MMX instructions: 0
21. Processor supports 3DNow instructions: 0
22. Processor supports 3DNow+ instructions: 0
23. IA64 processor emulating x86 : 0
24. OS name: Linux
25. OS sub-type: 4.16.7-1-ARCH
26. OS build ID: #1 SMP PREEMPT Wed May 2 21:12:36 UTC 2018
27. OS platform: x86_64

```

6. 输出会随着处理器的不同而变化。

工作原理

`CMakeLists.txt` 中的 `foreach` 循环会查询多个键值，并定义相应的变量。此示例的核心函数是 `cmake_host_system_information`，它查询运行CMake的主机系统的系统信息。本例中，我们对每个键使用了一个函数调用。然后，使用这些变量来配置 `config.h.in` 中的占位符，输入并生成 `config.h`。此配置使用 `configure_file` 命令完成。最后，`config.h` 包含在 `processor-info.cpp` 中。编译后，它将把值打印到屏幕上。我们将在第5章(配置时和构建时操作)和第6章(生成源代码)中重新讨论这种方法。

更多信息

对于更细粒度的处理器指令集检测，请考虑以下模块：

<https://github.com/VcDevel/Vc/blob/master/cmake/OptimizeForArchitecture.cmake>。有时候，构建代码的主机可能与运行代码的主机不一样。在计算集群中，登录节点的体系结构可能与计算节点上的体系结构不同。解决此问题的一种方法是，将配置和编译作为计算步骤，提交并部署到相应计算节点上。

2.6 为Eigen库使能向量化

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-02/recipe-06> 中找到，包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

处理器的向量功能，可以提高代码的性能。对于某些类型的运算来说尤为甚之，例如：线性代数。本示例将展示如何使能矢量化，以便使用线性代数的Eigen C++库加速可执行文件。

准备工作

我们用Eigen C++模板库，来进行线性代数计算，并展示如何设置编译器标志来启用向量化。这个示例的源代码 `linear-algebra.cpp` 文件：

```
1. #include <chrono>
2. #include <iostream>
3.
4. #include <Eigen/Dense>
5.
6. EIGEN_DONT_INLINE
7. double simple_function(Eigen::VectorXd &va, Eigen::VectorXd &vb)
8. {
9.     // this simple function computes the dot product of two vectors
10.    // of course it could be expressed more compactly
11.    double d = va.dot(vb);
12.    return d;
13. }
14.
15. int main()
16. {
17.     int len = 1000000;
18.     int num_repetitions = 100;
19.
20.     // generate two random vectors
21.     Eigen::VectorXd va = Eigen::VectorXd::Random(len);
22.     Eigen::VectorXd vb = Eigen::VectorXd::Random(len);
23.
24.     double result;
25.     auto start = std::chrono::system_clock::now();
26.     for (auto i = 0; i < num_repetitions; i++)
```

```

27. {
28.     result = simple_function(va, vb);
29. }
30. auto end = std::chrono::system_clock::now();
31. auto elapsed_seconds = end - start;
32.
33. std::cout << "result: " << result << std::endl;
34. std::cout << "elapsed seconds: " << elapsed_seconds.count() << std::endl;
35. }
```

我们期望向量化可以加快 `simple_function` 中的点积操作。

如何实施

根据Eigen库的文档，设置适当的编译器标志就足以生成向量化的代码。让我们看看 `CMakeLists.txt`：

1. 声明一个 `C++11` 项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-06 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 使用Eigen库，我们需要在系统上找到它的头文件：

```
1. find_package(Eigen3 3.3 REQUIRED CONFIG)
```

3. `CheckCXXCompilerFlag.cmake` 标准模块文件：

```
1. include(CheckCXXCompilerFlag)
```

4. 检查 `-march=native` 编译器标志是否工作：

```
1. check_cxx_compiler_flag("-march=native" _march_native_works)
```

5. 另一个选项 `-xHost` 编译器标志也开启：

```
1. check_cxx_compiler_flag("-xHost" _xhost_works)
```

6. 设置了一个空变量 `_CXX_FLAGS`，来保存刚才检查的两个编译器中找到的编译器标志。如果看到 `_march_native_works`，我们将 `_CXX_FLAGS` 设置为 `-march=native`。如果看到 `_xhost_works`，我们将 `_CXX_FLAGS` 设置为 `-xHost`。如果它们都不起作用，`_CXX_FLAGS` 将为空，并禁用矢量化：

```
1. set(_CXX_FLAGS)
2. if(_march_native_works)
   message(STATUS "Using processor's vector instructions (-march=native
3. compiler flag set)")
4.   set(_CXX_FLAGS "-march=native")
5. elseif(_xhost_works)
   message(STATUS "Using processor's vector instructions (-xHost compiler
6. flag set)")
7.   set(_CXX_FLAGS "-xHost")
8. else()
9.   message(STATUS "No suitable compiler flag found for vectorization")
10. endif()
```

7. 为了便于比较，我们还为未优化的版本定义了一个可执行目标，不使用优化标志：

```
1. add_executable(linear-algebra-unoptimized linear-algebra.cpp)
2.
3. target_link_libraries(linear-algebra-unoptimized
4. PRIVATE
5.   Eigen3::Eigen
6. )
```

8. 此外，我们定义了一个优化版本：

```
1. add_executable(linear-algebra linear-algebra.cpp)
2.
3. target_compile_options(linear-algebra
4. PRIVATE
5.   ${_CXX_FLAGS}
6. )
7.
8. target_link_libraries(linear-algebra
9. PRIVATE
10.  Eigen3::Eigen
```

11.)

9. 让我们比较一下这两个可执行文件—首先我们配置(在本例中，`-march=native_works`)：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Performing Test _march_native_works
7. -- Performing Test _march_native_works - Success
8. -- Performing Test _xhost_works
9. -- Performing Test _xhost_works - Failed
10. -- Using processor's vector instructions (-march=native compiler flag set)
11. ...

```

10. 最后，让我们编译可执行文件，并比较运行时间：

```

1. $ cmake --build .
2. $ ./linear-algebra-unoptimized
3.
4. result: -261.505
5. elapsed seconds: 1.97964
6.
7. $ ./linear-algebra
8.
9. result: -261.505
10. elapsed seconds: 1.05048

```

工作原理

大多数处理器提供向量指令集，代码可以利用这些特性，获得更高的性能。由于线性代数运算可以从Eigen库中获得很好的加速，所以在使用Eigen库时，就要考虑向量化。我们所要做的就是，指示编译器为我们检查处理器，并为当前体系结构生成本机指令。不同的编译器供应商会使用不同的标志来实现这一点：GNU编译器使用`-march=native` 标志来实现这一点，而Intel编译器使用`-xHost` 标志。使用`CheckCXXCompilerFlag.cmake` 模块提供的`check_cxx_compiler_flag` 函数进行编译器标志的检查：

```
check_cxx_compiler_flag("-march=native" _march_native_works)
```

这个函数接受两个参数：

- 第一个是要检查的编译器标志。
- 第二个是用来存储检查结果(true或false)的变量。如果检查为真，我们将工作标志添加到 `_CXX_FLAGS` 变量中，该变量将用于为可执行目标设置编译器标志。

更多信息

本示例可与前一示例相结合，可以使用 `cmake_host_system_information` 查询处理器功能。

第3章 检测外部库和程序

本章中主要内容有：

- 检测Python解释器
- 检测Python库
- 检测Python模块和包
- 检测BLAS和LAPACK数学库
- 检测OpenMP并行环境
- 检测MPI并行环境
- 检测Eigen库
- 检测Boost库
- 检测外部库：I . 使用pkg-config
- 检测外部库：II . 书写find模块

我们的项目常常会依赖于其他项目和库。本章将演示，如何检测外部库、框架和项目，以及如何链接到这些库。CMake有一组预打包模块，用于检测常用库和程序，例如：Python和Boost。可以使用 `cmake --help-module-list` 获得现有模块的列表。但是，不是所有的库和程序都包含在其中，有时必须自己编写检测脚本。本章将讨论相应的工具，了解CMake的 `find` 族命令：

- `find_file`: 在相应路径下查找命名文件
- `find_library`: 查找一个库文件
- `find_package`: 从外部项目查找和加载设置
- `find_path`: 查找包含指定文件的目录
- `find_program`: 找到一个可执行程序

NOTE: 可以使用 `--help-command` 命令行显示CMake内置命令的打印文档。

3.1 检测Python解释器

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-01> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Python是一种非常流行的语言。许多项目用Python编写的工具，从而将主程序和库打包在一起，或者在配置或构建过程中使用Python脚本。这种情况下，确保运行时对Python解释器的依赖也需要得到满足。本示例将展示如何检测和使用Python解释器。

我们将介绍 `find_package` 命令，这个命令将贯穿本章。

具体实施

我们将逐步建立 `CMakeLists.txt` 文件：

- 首先，定义CMake最低版本和项目名称。注意，这里不需要任何语言支持：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES NONE)

```

- 然后，使用 `find_package` 命令找到Python解释器：

```
1. find_package(PythonInterp REQUIRED)
```

- 然后，执行Python命令并捕获它的输出和返回值：

```

1. execute_process(
2.   COMMAND
3.     ${PYTHON_EXECUTABLE} "-c" "print('Hello, world!')"
4.   RESULT_VARIABLE _status
5.   OUTPUT_VARIABLE _hello_world
6.   ERROR_QUIET
7.   OUTPUT_STRIP_TRAILING_WHITESPACE
8. )

```

- 最后，打印Python命令的返回值和输出：

```

1. message(STATUS "RESULT_VARIABLE is: ${_status}")
2. message(STATUS "OUTPUT_VARIABLE is: ${_hello_world}")

```

5. 配置项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- Found PythonInterp: /usr/bin/python (found version "3.6.5")
6. -- RESULT_VARIABLE is: 0
7. -- OUTPUT_VARIABLE is: Hello, world!
8. -- Configuring done
9. -- Generating done
   -- Build files have been written to: /home/user/cmake-cookbook/chapter-
10. 03/recipe-01/example/build

```

工作原理

`find_package` 是用于发现和设置包的CMake模块的命令。这些模块包含CMake命令，用于标识系统标准位置中的包。CMake模块文件称为 `Find<name>.cmake`，当调用 `find_package(<name>)` 时，模块中的命令将会运行。

除了在系统上实际查找包模块之外，查找模块还会设置了一些有用的变量，反映实际找到了什么，也可以在自己的 `CMakeLists.txt` 中使用这些变量。对于Python解释器，相关模块为 `FindPythonInterp.cmake` 附带的设置了一些CMake变量：

- `PYTHONINTERP_FOUND`: 是否找到解释器
- `PYTHON_EXECUTABLE`: Python解释器到可执行文件的路径
- `PYTHON_VERSION_STRING`: Python解释器的完整版本信息
- `PYTHON_VERSION_MAJOR`: Python解释器的主要版本号
- `PYTHON_VERSION_MINOR`: Python解释器的次要版本号
- `PYTHON_VERSION_PATCH`: Python解释器的补丁版本号

可以强制CMake，查找特定版本的包。例如，要求Python解释器的版本大于或等于

2.7: `find_package(PythonInterp 2.7)`

可以强制满足依赖关系：

```
1. find_package(PythonInterp REQUIRED)
```

如果在查找位置中没有找到适合Python解释器的可执行文件，CMake将中止配置。

TIPS: CMake有很多查找软件包的模块。我们建议在CMake在线文档中查询 `Find<package>.cmake` 模块，并在使用它们之前详细阅读它们的文档。`find_package` 命令的

文档可以参考 https://cmake.org/cmake/help/v3.5/command/find_package.html。在线文档的一个很好的替代方法是浏览 <https://github.com/Kitware/CMake/tree/master/Modules> 中的CMake模块源代码—它们记录了模块使用的变量，以及模块可以在 `CMakeLists.txt` 中使用的变量。

更多信息

软件包没有安装在标准位置时，CMake无法正确定位它们。用户可以使用CLI的 `-D` 参数传递相应的选项，告诉CMake查看特定的位置。Python解释器可以使用以下配置：

```
1. $ cmake -D PYTHON_EXECUTABLE=/custom/location/python ..
```

这将指定非标准 `/custom/location/python` 安装目录中的Python可执行文件。

NOTE: 每个包都是不同的，`Find<package>.cmake` 模块试图提供统一的检测接口。当CMake无法找到模块包时，我们建议您阅读相应检测模块的文档，以了解如何正确地使用CMake模块。可以在终端中直接浏览文档，本例中可使用 `cmake --help-module FindPythonInterp` 查看。

除了检测包之外，我们还想提到一个便于打印变量的helper模块。本示例中，我们使用了以下方法：

```
1. message(STATUS "RESULT_VARIABLE is: ${_status}")
2. message(STATUS "OUTPUT_VARIABLE is: ${_hello_world}")
```

使用以下工具进行调试：

```
1. include(CMakePrintHelpers)
2. cmake_print_variables(_status _hello_world)
```

将产生以下输出：

```
1. -- _status="0" ; _hello_world="Hello, world!"
```

有关打印属性和变量的更多信息，请参考

<https://cmake.org/cmake/help/v3.5/module/CMakePrintHelpers.html>。

3.2 检测Python库

NOTE:此示例代码可以在 <https://github.com/devcafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-02> 中找到，有一个C示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

可以使用Python工具来分析和操作程序的输出。然而，还有更强大的方法可以将解释语言(如Python)与编译语言(如C或C++)组合在一起使用。一种是扩展Python，通过编译成共享库的C或C++模块在这些类型上提供新类型和新功能，这是第9章的主题。另一种是将Python解释器嵌入到C或C++程序中。两种方法都需要下列条件：

- Python解释器的工作版本
- Python头文件Python.h的可用性
- Python运行时库libpython

三个组件所使用的Python版本必须相同。我们已经演示了如何找到Python解释器；本示例中，我们将展示另外两种方式。

准备工作

我们将一个简单的Python代码，嵌入到C程序中，可以在Python文档页面上找到。源文件称为 `hello-embedded-python.c`：

```

1. #include <Python.h>
2.
3. int main(int argc, char *argv[]) {
4.     Py_SetProgramName(argv[0]); /* optional but recommended */
5.     Py_Initialize();
6.     PyRun_SimpleString("from time import time,ctime\n"
7.                        "print 'Today is',ctime(time())\n");
8.     Py_Finalize();
9.     return 0;
10. }
```

此代码将在程序中初始化Python解释器的实例，并使用Python的 `time` 模块，打印日期。

NOTE:嵌入代码可以在Python文档页面的
<https://docs.python.org/2/extending/embedding.html> 和
<https://docs.python.org/3/extending/embedding.html> 中找到。

具体实施

以下是 `CMakeLists.txt` 中的步骤：

1. 包含CMake最低版本、项目名称和所需语言：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-02 LANGUAGES C)
```

2. 制使用C99标准，这不严格要求与Python链接，但有时你可能需要对Python进行连接：

```
1. set(CMAKE_C_STANDARD 99)
2. set(CMAKE_C_EXTENSIONS OFF)
3. set(CMAKE_C_STANDARD_REQUIRED ON)
```

3. 找到Python解释器。这是一个 `REQUIRED` 依赖：

```
1. find_package(PythonInterp REQUIRED)
```

4. 找到Python头文件和库的模块，称为 `FindPythonLibs.cmake`：

```
find_package(PythonLibs ${PYTHON_VERSION_MAJOR}.${PYTHON_VERSION_MINOR})
1. EXACT REQUIRED
```

5. 使用 `hello-embedded-python.c` 源文件，添加一个可执行目标：

```
1. add_executable(hello-embedded-python hello-embedded-python.c)
```

6. 可执行文件包含 `Python.h` 头文件。因此，这个目标的 `include` 目录必须包含Python的 `include` 目录，可以通过 `PYTHON_INCLUDE_DIRS` 变量进行指定：

```
1. target_include_directories(hello-embedded-python
2.   PRIVATE
3.     ${PYTHON_INCLUDE_DIRS}
4. )
```

7. 最后，将可执行文件链接到Python库，通过 `PYTHON_LIBRARIES` 变量访问：

```
1. target_link_libraries(hello-embedded-python
2.   PRIVATE
3.     ${PYTHON_LIBRARIES})
```

```
4.      )
```

8. 现在，进行构建：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Found PythonInterp: /usr/bin/python (found version "3.6.5")
   -- Found PythonLibs: /usr/lib/libpython3.6m.so (found suitable exact
7. version "3.6.5")
```

9. 最后，执行构建，并运行可执行文件：

```
1. $ cmake --build .
2. $ ./hello-embedded-python
3.
4. Today is Thu Jun 7 22:26:02 2018
```

工作原理

`FindPythonLibs.cmake` 模块将查找Python头文件和库的标准位置。由于，我们的项目需要这些依赖项，如果没有找到这些依赖项，将停止配置，并报出错误。

注意，我们显式地要求CMake检测安装的Python可执行文件。这是为了确保可执行文件、头文件和库都有一个匹配的版本。这对于不同版本，可能在运行时导致崩溃。我们通过 `FindPythonInterp.cmake` 中定义的 `PYTHON_VERSION_MAJOR` 和 `PYTHON_VERSION_MINOR` 来实现：

```
1. find_package(PythonInterp REQUIRED)
   find_package(PythonLibs ${PYTHON_VERSION_MAJOR}.${PYTHON_VERSION_MINOR} EXACT
2. REQUIRED)
```

使用 `EXACT` 关键字，限制CMake检测特定的版本，在本例中是匹配的相应Python版本的包括文件和库。我们可以使用 `PYTHON_VERSION_STRING` 变量，进行更接近的匹配：

```
1. find_package(PythonInterp REQUIRED)
2. find_package(PythonLibs ${PYTHON_VERSION_STRING} EXACT REQUIRED)
```

更多信息

当Python不在标准安装目录中，我们如何确定Python头文件和库的位置是正确的？对于Python解释器，可以通过CLI的 `-D PYTHON_LIBRARY` 和 `PYTHON_INCLUDE_DIR` 选项来强制CMake查找特定的目录。这些选项指定了以下内容：

- **PYTHON_LIBRARY**: 指向Python库的路径
- **PYTHON_INCLUDE_DIR**: Python.h所在的路径

这样，就能获得所需的Python版本。

TIPS:有时需要将 `-D PYTHON_EXECUTABLE` 、 `-D PYTHON_LIBRARY` 和 `-D PYTHON_INCLUDE_DIR` 传递给CMake CLI，以便找到及定位相应的版本的组件。

要将Python解释器及其开发组件匹配为完全相同的版本可能非常困难，对于那些将它们安装在非标准位置或系统上安装了多个版本的情况尤其如此。CMake 3.12版本中增加了新的Python检测模块，旨在解决这个棘手的问题。我们 `CMakeLists.txt` 的检测部分也将简化为：

```
find_package(Python COMPONENTS Interpreter Development REQUIRED)
```

我们建议您阅读新模块的文档，地址是：

<https://cmake.org/cmake/help/v3.12/module/FindPython.html>

3.3 检测Python模块和包

NOTE: 此示例代码可以在 <https://github.com/devcafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-03> 中找到，包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例中，我们演示了如何检测Python解释器，以及如何编译一个简单的C程序(嵌入Python解释器)。通常，代码将依赖于特定的Python模块，无论是Python工具、嵌入Python的程序，还是扩展Python的库。例如，科学界非常流行使用NumPy处理矩阵问题。依赖于Python模块或包的项目中，确定满足对这些Python模块的依赖非常重要。本示例将展示如何探测用户的环境，以找到特定的Python模块和包。

准备工作

我们将尝试在C++程序中嵌入一个稍微复杂一点的例子。这个示例再次引用[Python在线文档](#)，并展示了如何通过调用编译后的C++可执行文件，来执行用户定义的Python模块中的函数。

Python 3示例代码([Py3-pure-embedding.cpp](#))包含以下源代码(请参见<https://docs.python.org/2/extending/embedding.html#pure-embedded>与Python 2代码等效)：

```

1. #include <Python.h>
2. int main(int argc, char* argv[]) {
3.     PyObject* pName, * pModule, * pDict, * pFunc;
4.     PyObject* pArgs, * pValue;
5.     int i;
6.     if (argc < 3) {
7.         fprintf(stderr, "Usage: pure-embedding pythonfile funcname [args]\n");
8.         return 1;
9.     }
10.    Py_Initialize();
11.    PyRun_SimpleString("import sys");
12.    PyRun_SimpleString("sys.path.append(\".\")");
13.    pName = PyUnicode_DecodeFSDefault(argv[1]);
14.    /* Error checking of pName left out */
15.    pModule = PyImport_Import(pName);
16.    Py_DECREF(pName);
17.    if (pModule != NULL) {
18.        pFunc = PyObject_GetAttrString(pModule, argv[2]);
19.        /* pFunc is a new reference */

```

```
20.     if (pFunc && PyCallable_Check(pFunc)) {
21.         pArgs = PyTuple_New(argc - 3);
22.         for (i = 0; i < argc - 3; ++i) {
23.             pValue = PyLong_FromLong(atoi(argv[i + 3]));
24.             if (!pValue) {
25.                 Py_DECREF(pArgs);
26.                 Py_DECREF(pModule);
27.                 fprintf(stderr, "Cannot convert argument\n");
28.                 return 1;
29.             }
30.             /* pValue reference stolen here: */
31.             PyTuple_SetItem(pArgs, i, pValue);
32.         }
33.         pValue = PyObject_CallObject(pFunc, pArgs);
34.         Py_DECREF(pArgs);
35.         if (pValue != NULL) {
36.             printf("Result of call: %ld\n", PyLong_AsLong(pValue));
37.             Py_DECREF(pValue);
38.         }
39.         else {
40.             Py_DECREF(pFunc);
41.             Py_DECREF(pModule);
42.             PyErr_Print();
43.             fprintf(stderr, "Call failed\n");
44.             return 1;
45.         }
46.     }
47.     else {
48.         if (PyErr_Occurred())
49.             PyErr_Print();
50.         fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
51.     }
52.     Py_XDECREF(pFunc);
53.     Py_DECREF(pModule);
54. }
55. else {
56.     PyErr_Print();
57.     fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
58.     return 1;
59. }
60. Py_Finalize();
61. return 0;
```

62. }

我们希望嵌入的Python代码([use_numpy.py](#))使用NumPy设置一个矩阵，所有矩阵元素都为1.0：

```

1. import numpy as np
2. def print_ones(rows, cols):
3.     A = np.ones(shape=(rows, cols), dtype=float)
4.     print(A)
5.
6.     # we return the number of elements to verify
7.     # that the C++ code is able to receive return values
8.     num_elements = rows*cols
9.     return(num_elements)
```

具体实施

下面的代码中，我们能够使用CMake检查NumPy是否可用。我们需要确保Python解释器、头文件和库在系统上是可用的。然后，将再来确认NumPy的可用性：

- 首先，我们定义了最低CMake版本、项目名称、语言和C++标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

- 查找解释器、头文件和库的方法与前面的方法完全相同：

```

1. find_package(PythonInterp REQUIRED)
   find_package(PythonLibs ${PYTHON_VERSION_MAJOR}.${PYTHON_VERSION_MINOR})
2. EXACT REQUIRED)
```

- 正确打包的Python模块，指定安装位置和版本。可以在[CMakeLists.txt](#)中执行Python脚本进行探测：

```

1. execute_process(
2.   COMMAND
3.     ${PYTHON_EXECUTABLE} "-c" "import re, numpy;
3.     print(re.compile('/__init__.py.*').sub('',numpy.__file__))"
4.   RESULT_VARIABLE _numpy_status
```

```

5.    OUTPUT_VARIABLE _numpy_location
6.    ERROR QUIET
7.    OUTPUT_STRIP_TRAILING_WHITESPACE
8.  )

```

4. 如果找到NumPy，则 `_numpy_status` 变量为整数，否则为错误的字符串，而 `_numpy_location` 将包含NumPy模块的路径。如果找到NumPy，则将它的位置保存到一个名为 `NumPy` 的新变量中。注意，新变量被缓存，这意味着CMake创建了一个持久性变量，用户稍后可以修改该变量：

```

1. if(NOT _numpy_status)
2.   set(NumPy ${_numpy_location} CACHE STRING "Location of NumPy")
3. endif()

```

5. 下一步是检查模块的版本。同样，我们在 `CMakeLists.txt` 中施加了一些Python魔法，将版本保存到 `_numpy_version` 变量中：

```

1. execute_process(
2.   COMMAND
3.     ${PYTHON_EXECUTABLE} "-c" "import numpy; print(numpy.__version__)"
4.   OUTPUT_VARIABLE _numpy_version
5.   ERROR QUIET
6.   OUTPUT_STRIP_TRAILING_WHITESPACE
7. )

```

6. 最后，`FindPackageHandleStandardArgs` 的CMake包以正确的格式设置 `NumPy_FOUND` 变量和输出信息：

```

1. include(FindPackageHandleStandardArgs)
2. find_package_handle_standard_args(NumPy
3.   FOUND_VAR NumPy_FOUND
4.   REQUIRED_VARS NumPy
5.   VERSION_VAR _numpy_version
6. )

```

7. 一旦正确的找到所有依赖项，我们就可以编译可执行文件，并将其链接到Python库：

```

1. add_executable(pure-embedding "")
2.
3. target_sources(pure-embedding
4.   PRIVATE

```

```

5.      Py${PYTHON_VERSION_MAJOR}-pure-embedding.cpp
6.    )
7.
8. target_include_directories(pure-embedding
9.   PRIVATE
10.    ${PYTHON_INCLUDE_DIRS}
11.  )
12.
13. target_link_libraries(pure-embedding
14.   PRIVATE
15.    ${PYTHON_LIBRARIES}
16.  )

```

8. 我们还必须保证 `use_numpy.py` 在 `build` 目录中可用：

```

1. add_custom_command(
2.   OUTPUT
3.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py
4.   COMMAND
5.     ${CMAKE_COMMAND} -E copy_if_different
6.     ${CMAKE_CURRENT_SOURCE_DIR}/use_numpy.py
7.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py
8.   DEPENDS
9.     ${CMAKE_CURRENT_SOURCE_DIR}/use_numpy.py
10.
11. # make sure building pure-embedding triggers the above custom command
12. target_sources(pure-embedding
13.   PRIVATE
14.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py
15.   )

```

9. 现在，我们可以测试嵌入的代码：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- ...
6. -- Found PythonInterp: /usr/bin/python (found version "3.6.5")
   -- Found PythonLibs: /usr/lib/libpython3.6m.so (found suitable exact
7.   version "3.6.5")

```

```

    -- Found NumPy: /usr/lib/python3.6/site-packages/numpy (found version
8. "1.14.3")
9.
10. $ cmake --build .
11. $ ./pure-embedding use_numpy print_ones 2 3
12.
13. [[1. 1. 1.]
14. [1. 1. 1.]]
15. Result of call: 6

```

工作原理

例子中有三个新的CMake命令，需要 `include(FindPackageHandleStandardArgs)` :

- `execute_process`
- `add_custom_command`
- `find_package_handle_standard_args`

`execute_process` 将作为通过子进程执行一个或多个命令。最后，子进程返回值将保存到变量作为参数，传递给 `RESULT_VARIABLE`，而管道标准输出和标准错误的内容将被保存到变量作为参数传递给 `OUTPUT_VARIABLE` 和 `ERROR_VARIABLE`。`execute_process` 可以执行任何操作，并使用它们的结果来推断系统配置。本例中，用它来确保NumPy可用，然后获得模块版本。

`find_package_handle_standard_args` 提供了，用于处理与查找相关程序和库的标准工具。引用此命令时，可以正确的处理与版本相关的选项(`REQUIRED` 和 `EXACT`)，而无需更多的CMake代码。稍后将介绍 `QUIET` 和 `COMPONENTS` 选项。本示例中，使用了以下方法：

```

1. include(FindPackageHandleStandardArgs)
2. find_package_handle_standard_args(NumPy
3.   FOUND_VAR NumPy_FOUND
4.   REQUIRED_VARS NumPy
5.   VERSION_VAR _numpy_version
6. )

```

所有必需的变量都设置为有效的文件路径(NumPy)后，发送到模块(`NumPy_FOUND`)。它还将版本保存在可传递的版本变量(`_numpy_version`)中并打印：

```
1. -- Found NumPy: /usr/lib/python3.6/site-packages/numpy (found version "1.14.3")
```

目前的示例中，没有进一步使用这些变量。如果返回 `NumPy_FOUND` 为 `FALSE`，则停止配置。

最后，将 `use_numpy.py` 复制到 `build` 目录，对代码进行注释：

```
1. add_custom_command(  
2.   OUTPUT  
3.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py  
4.   COMMAND  
5.     ${CMAKE_COMMAND} -E copy_if_different  
6.     ${CMAKE_CURRENT_SOURCE_DIR}/use_numpy.py  
7.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py  
8.   DEPENDS  
9.     ${CMAKE_CURRENT_SOURCE_DIR}/use_numpy.py  
10.  
11. target_sources(pure-embedding  
12.   PRIVATE  
13.     ${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py  
14. )
```

我们也可以使用 `file(COPY...)` 命令来实现复制。这里，我们选择使用 `add_custom_command`，来确保文件在每次更改时都会被复制，而不仅仅是第一次运行配置时。我们将在第5章更详细地讨论 `add_custom_command`。还要注意 `target_sources` 命令，它将依赖项添加到 `${CMAKE_CURRENT_BINARY_DIR}/use_numpy.py`；这样做是为了确保构建目标，能够触发之前的命令。

3.4 检测BLAS和LAPACK数学库

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-04> 中找到，有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

许多数据算法严重依赖于矩阵和向量运算。例如：矩阵-向量和矩阵-矩阵乘法，求线性方程组的解，特征值和特征向量的计算或奇异值分解。这些操作在代码库中非常普遍，因为操作的数据量比较大，因此高效的实现有绝对的必要。幸运的是，有专家库可用：基本线性代数子程序(BLAS)和线性代数包(LAPACK)，为许多线性代数操作提供了标准API。供应商有不同的实现，但都共享API。虽然，用于数学库底层实现，实际所用的编程语言会随着时间而变化(Fortran、C、Assembly)，但是也都是Fortran调用接口。考虑到调用街扩，本示例中的任务要链接到这些库，并展示如何用不同语言编写的库。

准备工作

为了展示数学库的检测和连接，我们编译一个C++程序，将矩阵的维数作为命令行输入，生成一个随机的方阵A，一个随机向量b，并计算线性系统方程： $Ax = b$ 。另外，将对向量b的进行随机缩放。这里，需要使用的子程序是BLAS中的DSCAL和LAPACK中的DGESV来求线性方程组的解。示例C++代码的清单(`linear-algebra.cpp`)：

```

1. #include "CxxBLAS.hpp"
2. #include "CxxLAPACK.hpp"
3.
4. #include <iostream>
5. #include <random>
6. #include <vector>
7.
8. int main(int argc, char** argv) {
9.     if (argc != 2) {
10.         std::cout << "Usage: ./linear-algebra dim" << std::endl;
11.         return EXIT_FAILURE;
12.     }
13.
14.     // Generate a uniform distribution of real number between -1.0 and 1.0
15.     std::random_device rd;
16.     std::mt19937 mt(rd());
17.     std::uniform_real_distribution<double> dist(-1.0, 1.0);
18.
19.     // Allocate matrices and right-hand side vector

```

```

20. int dim = std::atoi(argv[1]);
21. std::vector<double> A(dim * dim);
22. std::vector<double> b(dim);
23. std::vector<int> ipiv(dim);
24. // Fill matrix and RHS with random numbers between -1.0 and 1.0
25. for (int r = 0; r < dim; r++) {
26.     for (int c = 0; c < dim; c++) {
27.         A[r + c * dim] = dist(mt);
28.     }
29.     b[r] = dist(mt);
30. }
31.
32. // Scale RHS vector by a random number between -1.0 and 1.0
33. C_DSCAL(dim, dist(mt), b.data(), 1);
34. std::cout << "C_DSCAL done" << std::endl;
35.
36. // Save matrix and RHS
37. std::vector<double> A1(A);
38. std::vector<double> b1(b);
39. int info;
40. info = C_DGESV(dim, 1, A.data(), dim, ipiv.data(), b.data(), dim);
41. std::cout << "C_DGESV done" << std::endl;
42. std::cout << "info is " << info << std::endl;
43.
44. double eps = 0.0;
45. for (int i = 0; i < dim; ++i) {
46.     double sum = 0.0;
47.     for (int j = 0; j < dim; ++j)
48.         sum += A1[i + j * dim] * b1[j];
49.     eps += std::abs(b1[i] - sum);
50. }
51. std::cout << "check is " << eps << std::endl;
52.
53. return 0;
54. }
```

使用C++11的随机库来生成-1.0到1.0之间的随机分布。`C_DSCAL` 和 `C_DGESV` 分别是到BLAS和LAPACK库的接口。为了避免名称混淆，将在下面来进一步讨论CMake模块：

文件 `CxxBLAS.hpp` 用 `extern "C"` 封装链接BLAS：

```
1. #pragma once
```

```

2. #include "fc_mangle.h"
3. #include <cstddef>
4. #ifdef __cplusplus
5. extern "C" {
6. #endif
7. extern void DSCAL(int *n, double *alpha, double *vec, int *inc);
8. #ifdef __cplusplus
9. }
10. #endif
11. void C_DSCAL(size_t length, double alpha, double *vec, int inc);

```

对应的实现文件 `CxxBLAS.cpp` :

```

1. #include "CxxBLAS.hpp"
2.
3. #include <climits>
4.
5. // see http://www.netlib.no/netlib/blas/dscal.f
6. void C_DSCAL(size_t length, double alpha, double *vec, int inc) {
7.     int big_blocks = (int)(length / INT_MAX);
8.     int small_size = (int)(length % INT_MAX);
9.     for (int block = 0; block <= big_blocks; block++) {
10.         double *vec_s = &vec[block * inc * (size_t)INT_MAX];
11.         signed int length_s = (block == big_blocks) ? small_size : INT_MAX;
12.         ::DSCAL(&length_s, &alpha, vec_s, &inc);
13.     }
14. }

```

`CxxLAPACK.hpp` 和 `CxxLAPACK.cpp` 为LAPACK调用执行相应的转换。

具体实施

对应的 `CMakeLists.txt` 包含以下构建块:

1. 我们定义了CMake最低版本，项目名称和支持的语言：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-04 LANGUAGES CXX C Fortran)

```

2. 使用C++11标准：

```

1. set(CMAKE_CXX_STANDARD 11)
2. set(CMAKE_CXX_EXTENSIONS OFF)
3. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

3. 此外，我们验证Fortran和C/C++编译器是否能协同工作，并生成头文件，这个文件可以处理名称混乱。两个功能都由 `FortranCInterface` 模块提供：

```

1. include(FortranCInterface)
2.
3. FortranCInterface_VERIFY(CXX)
4.
5. FortranCInterface_HEADER(
6.   fc_mangle.h
7.   MACRO_NAMESPACE "FC_"
8.   SYMBOLS DSCAL DGESV
9. )

```

4. 然后，找到BLAS和LAPACK：

```

1. find_package(BLAS REQUIRED)
2. find_package(LAPACK REQUIRED)

```

5. 接下来，添加一个库，其中包含BLAS和LAPACK包装器的源代码，并链接到 `LAPACK_LIBRARIES`，其中也包含 `BLAS_LIBRARIES`：

```

1. add_library(math "")
2.
3. target_sources(math
4.   PRIVATE
5.   CxxBLAS.cpp
6.   CxxLAPACK.cpp
7. )
8.
9. target_include_directories(math
10. PUBLIC
11.   ${CMAKE_CURRENT_SOURCE_DIR}
12.   ${CMAKE_CURRENT_BINARY_DIR}
13. )
14.
15. target_link_libraries(math
16. PUBLIC

```

```

17.      ${LAPACK_LIBRARIES}
18.  )

```

6. 注意，目标的包含目录和链接库声明为 `PUBLIC`，因此任何依赖于数学库的附加目标也将在其包含目录中。

7. 最后，我们添加一个可执行目标并链接 `math`：

```

1. add_executable(linear-algebra "")
2.
3. target_sources(linear-algebra
4.   PRIVATE
5.     linear-algebra.cpp
6.   )
7.
8. target_link_libraries(linear-algebra
9.   PRIVATE
10.    math
11.   )

```

8. 配置时，我们可以关注相关的打印输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. ...
6. -- Detecting Fortran/C Interface
7. -- Detecting Fortran/C Interface - Found GLOBAL and MODULE mangling
8. -- Verifying Fortran/C Compiler Compatibility
9. -- Verifying Fortran/C Compiler Compatibility - Success
10. ...
11. -- Found BLAS: /usr/lib/libblas.so
12. ...
13. -- A library with LAPACK API found.
14. ...

```

9. 最后，构建并测试可执行文件：

```

1. $ cmake --build .
2. $ ./linear-algebra 1000
3.

```

```

4. C_DSCAL done
5. C_DGESV done
6. info is 0
7. check is 1.54284e-10

```

工作原理

`FindBLAS.cmake` 和 `FindLAPACK.cmake` 将在标准位置查找BLAS和LAPACK库。对于前者，该模块有 `SGEMM` 函数的Fortran实现，一般用于单精度矩阵乘积。对于后者，该模块有 `CHEEV` 函数的Fortran实现，用于计算复杂厄米矩阵的特征值和特征向量。查找在CMake内部，通过编译一个小程序来完成，该程序调用这些函数，并尝试链接到候选库。如果失败，则表示相应库不存在于系统上。

生成机器码时，每个编译器都会处理符号混淆，不幸的是，这种操作并不通用，而与编译器相关。为了解决这个问题，我们使用 `FortranCInterface` 模块(

<https://cmake.org/cmake/help/v3.5/module/FortranCInterface.html>)验证Fortran和C/C++能否混合编译，然后生成一个Fortran-C接口头文件 `fc_mangle.h`，这个文件用来解决编译器性的问题。然后，必须将生成的 `fc_mann.h` 包含在接口头文件 `CxxBLAS.hpp` 和 `CxxLAPACK.hpp` 中。为了使用 `FortranCInterface`，我们需要在 `LANGUAGES` 列表中添加C和Fortran支持。当然，也可以定义自己的预处理器定义，但是可移植性会差很多。

我们将在第9章中更详细地讨论Fortran和C的互操作性。

NOTE: 目前，BLAS和LAPACK的许多实现已经在Fortran外附带了一层C包装。这些包装器多年来已经标准化，称为CBLAS和LAPACK。

更多信息

许多算法代码比较依赖于矩阵代数运算，使用BLAS和LAPACK API的高性能实现就非常重要了。供应商为不同的体系结构和并行环境提供不同的库，`FindBLAS.cmake` 和 `FindLAPACK.cmake` 可能的无法定位到当前库。如果发生这种情况，可以通过 `-D` 选项显式地从CLI对库进行设置。

3.5 检测OpenMP的并行环境

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-05> 中找到，有一个C++和一个Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-05> 中也有一个适用于CMake 3.5的示例。

目前，市面上的计算机几乎都是多核机器，对于性能敏感的程序，我们必须关注这些多核处理器，并在编程模型中使用并发。OpenMP是多核处理器上并行性的标准之一。为了从OpenMP并行化中获得性能收益，通常不需要修改或重写现有程序。一旦确定了代码中的性能关键部分，例如：使用分析工具，程序员就可以通过预处理器指令，指示编译器为这些区域生成可并行的代码。

本示例中，我们将展示如何编译一个包含OpenMP指令的程序(前提是使用一个支持OpenMP的编译器)。有许多支持OpenMP的Fortran、C和C++编译器。对于相对较新的CMake版本，为OpenMP提供了非常好的支持。本示例将展示如何在使用CMake 3.9或更高版本时，使用简单C++和Fortran程序来链接到OpenMP。

NOTE: 根据Linux发行版的不同，Clang编译器的默认版本可能不支持OpenMP。使用或非苹果版本的Clang(例如，Conda提供的)或GNU编译器，除非单独安装libomp库(<https://iscinumpy.gitlab.io/post/omp-on-high-sierra/>)，否则本节示例将无法在macOS上工作。

准备工作

C和C++程序可以通过包含 `omp.h` 头文件和链接到正确的库，来使用OpenMP功能。编译器将在性能关键部分之前添加预处理指令，并生成并行代码。在本示例中，我们将构建以下示例源代码(`example.cpp`)。这段代码从1到N求和，其中N作为命令行参数：

```

1. #include <iostream>
2. #include <omp.h>
3. #include <string>
4.
5. int main(int argc, char *argv[])
6. {
7.     std::cout << "number of available processors: " << omp_get_num_procs()
8.             << std::endl;
9.     std::cout << "number of threads: " << omp_get_max_threads() << std::endl;
10.    auto n = std::stol(argv[1]);
11.    std::cout << "we will form sum of numbers from 1 to " << n << std::endl;

```

```

12. // start timer
13. auto t0 = omp_get_wtime();
14. auto s = 0LL;
15. #pragma omp parallel for reduction(+: s)
16. for (auto i = 1; i <= n; i++)
17. {
18.     s += i;
19. }
20. // stop timer
21. auto t1 = omp_get_wtime();
22.
23. std::cout << "sum: " << s << std::endl;
    std::cout << "elapsed wall clock time: " << t1 - t0 << " seconds" <<
24. std::endl;
25.
26. return 0;
27. }
```

在Fortran语言中，需要使用 `omp_lib` 模块并链接到库。在性能关键部分之前的代码注释中，可以再次使用并行指令。例如：`F90` 需要包含以下内容：

```

1. program example
2.
3. use omp_lib
4.
5. implicit none
6.
7. integer(8) :: i, n, s
8. character(len=32) :: arg
9. real(8) :: t0, t1
10.
11. print *, "number of available processors:", omp_get_num_procs()
12. print *, "number of threads:", omp_get_max_threads()
13.
14. call get_command_argument(1, arg)
15. read(arg, *) n
16.
17. print *, "we will form sum of numbers from 1 to", n
18.
19. ! start timer
20. t0 = omp_get_wtime()
21.
```

```

22.   s = 0
23. !$omp parallel do reduction(+:s)
24.   do i = 1, n
25.     s = s + i
26.   end do
27.
28.   ! stop timer
29.   t1 = omp_get_wtime()
30.
31.   print *, "sum:", s
32.   print *, "elapsed wall clock time (seconds):", t1 - t0
33.
34. end program

```

具体实施

对于C++和Fortran的例子，[CMakeLists.txt](#) 将遵循一个模板，该模板在这两种语言上很相似：

- 两者都定义了CMake最低版本、项目名称和语言(CXX或Fortran；我们将展示C++版本)：

```

1. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
2. project(recipe-05 LANGUAGES CXX)

```

- 使用C++11标准：

```

1. set(CMAKE_CXX_STANDARD 11)
2. set(CMAKE_CXX_EXTENSIONS OFF)
3. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

- 调用find_package来搜索OpenMP：

```
1. find_package(OpenMP REQUIRED)
```

- 最后，我们定义可执行目标，并链接到FindOpenMP模块提供的导入目标(在Fortran的情况下，我们链接到 [OpenMP::OpenMP_Fortran](#))：

```

1. add_executable(example example.cpp)
2. target_link_libraries(example
3.   PUBLIC
4.     OpenMP::OpenMP_CXX
5.   )

```

5. 现在，可以配置和构建代码了：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
```

6. 并行测试(在本例中使用了4个内核)：

```
1. $ ./example 1000000000
2.
3. number of available processors: 4
4. number of threads: 4
5. we will form sum of numbers from 1 to 1000000000
6. sum: 500000000500000000
7. elapsed wall clock time: 1.08343 seconds
```

7. 为了比较，我们可以重新运行这个例子，并将OpenMP线程的数量设置为1：

```
1. $ env OMP_NUM_THREADS=1 ./example 1000000000
2.
3. number of available processors: 4
4. number of threads: 1
5. we will form sum of numbers from 1 to 1000000000
6. sum: 500000000500000000
7. elapsed wall clock time: 2.96427 seconds
```

工作原理

我们的示例很简单：编译代码，并运行在多个内核上时，我们会看到加速效果。加速效果并不是 `OMP_NUM_THREADS` 的倍数，不过本示例中并不关心，因为我们更关注的是如何使用CMake配置需要使用OpenMP的项目。我们发现链接到OpenMP非常简单，这要感谢 `FindOpenMP` 模块：

```
1. target_link_libraries(example
2.   PUBLIC
3.     OpenMP::OpenMP_CXX
4. )
```

我们不关心编译标志或包含目录—这些设置和依赖项是在 `OpenMP::OpenMP_CXX` 中定义的

(`IMPORTED` 类型)。如第1章第3节中提到的，`IMPORTED` 库是伪目标，它完全是我们自己项目的外部依赖项。要使用OpenMP，需要设置一些编译器标志，包括目录和链接库。所有这些都包含在 `OpenMP::OpenMP_CXX` 的属性上，并通过使用 `target_link_libraries` 命令传递给 `example`。这使得在CMake中，使用库变得非常容易。我们可以使用 `cmake_print_properties` 命令打印接口的属性，该命令由 `CMakePrintHelpers.CMake` 模块提供：

```

1. include(CMakePrintHelpers)
2. cmake_print_properties(
3.   TARGETS
4.     OpenMP::OpenMP_CXX
5.   PROPERTIES
6.     INTERFACE_COMPILE_OPTIONS
7.     INTERFACE_INCLUDE_DIRECTORIES
8.     INTERFACE_LINK_LIBRARIES
9. )

```

所有属性都有 `INTERFACE_` 前缀，因为这些属性对所需目标，需要以接口形式提供，并且目标以接口的方式使用OpenMP。

对于低于3.9的CMake版本：

```

1. add_executable(example example.cpp)
2.
3. target_compile_options(example
4.   PUBLIC
5.     ${OpenMP_CXX_FLAGS}
6. )
7.
8. set_target_properties(example
9.   PROPERTIES
10.    LINK_FLAGS ${OpenMP_CXX_FLAGS}
11. )

```

对于低于3.5的CMake版本，我们需要为Fortran项目显式定义编译标志。

在这个示例中，我们讨论了C++和Fortran。相同的参数和方法对于C项目也有效。

3.6 检测MPI的并行环境

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-06> 中找到，包含一个C++和一个C的示例。该示例在CMake 3.9版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-06> 中也有一个适用于CMake 3.5的C示例。

消息传递接口(Message Passing Interface, MPI)，可以作为OpenMP(共享内存并行方式)的补充，它也是分布式系统上并行程序的实际标准。尽管，最新的MPI实现也允许共享内存并行，但高性能计算中的一种典型方法就是在计算节点上OpenMP与MPI结合使用。MPI标准的实施包括：

1. 运行时库
2. 头文件和Fortran 90模块
3. 编译器的包装器，用来调用编译器，使用额外的参数来构建MPI库，以处理目录和库。通常，包装器 `mpic++/mpicc/mpicxx` 用于C++，`mpicc` 用于C，`mpifort` 用于Fortran。
4. 启动MPI：应该启动程序，以编译代码的并行执行。它的名称依赖于实现，可以使用这几个命令启动：`mpirun`、`mpiexec` 或 `orterun`。

本示例，将展示如何在系统上找到合适的MPI实现，从而编译一个简单的“Hello, World”MPI例程。

准备工作

示例代码(`hello-mpi.cpp`，可从<http://www.mpitutorial.com> 下载)将在本示例中进行编译，它将初始化MPI库，让每个进程打印其名称：

```

1. #include <iostream>
2.
3. #include <mpi.h>
4.
5. int main(int argc, char **argv)
6. {
7.     // Initialize the MPI environment. The two arguments to MPI_Init are not
8.     // currently used by MPI implementations, but are there in case future
9.     // implementations might need the arguments.
10.    MPI_Init(NULL, NULL);
11.
12.    // Get the number of processes
13.    int world_size;
14.    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

```

```

15.
16. // Get the rank of the process
17. int world_rank;
18. MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
19.
20. // Get the name of the processor
21. char processor_name[MPI_MAX_PROCESSOR_NAME];
22. int name_len;
23. MPI_Get_processor_name(processor_name, &name_len);
24.
25. // Print off a hello world message
26. std::cout << "Hello world from processor " << processor_name << ", rank "
   << world_rank << " out of " << world_size << " processors" <<
27. std::endl;
28.
29. // Finalize the MPI environment. No more MPI calls can be made after this
30. MPI_Finalize();
31. }

```

具体实施

这个示例中，我们先查找MPI实现：库、头文件、编译器包装器和启动器。为此，我们将用到 `FindMPI.cmake` 标准CMake模块：

- 首先，定义了CMake最低版本、项目名称、支持的语言和语言标准：

```

1. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
2.
3. project(recipe-06 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

- 然后，调用 `find_package` 来定位MPI：

```
1. find_package(MPI REQUIRED)
```

- 与前面的配置类似，定义了可执行文件的名称和相关源码，并链接到目标：

```
1. add_executable(hello-mpi hello-mpi.cpp)
```

```

2.
3. target_link_libraries(hello-mpi
4.   PUBLIC
5.     MPI::MPI_CXX
6. )

```

4. 配置和构建可执行文件：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake .. # -D CMAKE_CXX_COMPILER=mpicxx C++例子中可加，加与不加对于构建结果没
3. 有影响(╯^╰)
4.
5. -- ...
6. -- Found MPI_CXX: /usr/lib/openmpi/libmpi_cxx.so (found version "3.1")
7. -- Found MPI: TRUE (found version "3.1")
8. -- ...
9.
10. $ cmake --build .

```

5. 为了并行执行这个程序，我们使用 `mpirun` 启动器(本例中，启动了两个任务)：

```

1. $ mpirun -np 2 ./hello-mpi
2.
3. Hello world from processor larry, rank 1 out of 2 processors
4. Hello world from processor larry, rank 0 out of 2 processors

```

工作原理

请记住，编译包装器是对MPI库编译器的封装。底层实现中，将会调用相同的编译器，并使用额外的参数(如成功构建并行程序所需的头文件包含路径和库)来扩充它。

编译和链接源文件时，包装器用了哪些标志？我们可以使用 `--showme` 选项来查看。要找出编译器的标志，我们可以这样使用：

```

1. $ mpicxx --showme:compile
2.
3. -pthread

```

为了找出链接器标志，我们可以这样：

```
1. $ mpicxx --showme:link
2.
3. -pthread -Wl,-rpath -Wl,/usr/lib/openmpi -Wl,--enable-new-dtags -
3. L/usr/lib/openmpi -lmpi_cxx -lmpi
```

与之前的OpenMP配置类似，我们发现到MPI的链接非常简单，这要归功于 [FindMPI](#) 模块提供的目标：

正如在前面的配方中所讨论的，对于CMake版本低于3.9，需要更多的工作量：

```
1. add_executable(hello-mpi hello-mpi.c)
2.
3. target_compile_options(hello-mpi
4.   PUBLIC
5.     ${MPI_CXX_COMPILE_FLAGS}
6. )
7.
8. target_include_directories(hello-mpi
9.   PUBLIC
10.    ${MPI_CXX_INCLUDE_PATH}
11. )
12.
13. target_link_libraries(hello-mpi
14.   PUBLIC
15.    ${MPI_CXX_LIBRARIES}
16. )
```

本示例中，我们讨论了C++项目。其中的参数和方法对于C或Fortran项目同样有效。

3.7 检测Eigen库

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-07> 中找到，包含一个C++的示例。该示例在CMake 3.9版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-06> 中也有一个适用于CMake 3.5的C++示例。

BLAS库为矩阵和向量操作提供了标准化接口。不过，这个接口用Fortran语言书写。虽然已经展示了如何使用C++直接使用这些库，但在现代C++程序中，希望有更高级的接口。

纯头文件实现的Eigen库，使用模板编程来提供接口。矩阵和向量的计算，会在编译时进行数据类型检查，以确保兼容所有维度的矩阵。密集和稀疏矩阵的运算，也可使用表达式模板高效的进行实现，如：矩阵-矩阵乘积，线性系统求解器和特征值问题。从3.3版开始，Eigen可以链接到BLAS和LAPACK库中，这可以将某些操作实现进行卸载，使库的实现更加灵活，从而获得更多的性能收益。

本示例将展示如何查找Eigen库，使用OpenMP并行化，并将部分工作转移到BLAS库。

本示例中会实现，矩阵-向量乘法和LU分解)，可以选择卸载BLAS和LAPACK库中的一些实现。这个示例中，只考虑将在BLAS库中卸载。

准备工作

本例中，我们编译一个程序，该程序会从命令行获取的随机方阵和维向量。然后我们将用LU分解来解线性方程组 $\mathbf{Ax}=\mathbf{b}$ 。以下是源代码(`linear-algebra.cpp`)：

```

1. #include <chrono>
2. #include <cmath>
3. #include <cstdlib>
4. #include <iomanip>
5. #include <iostream>
6. #include <vector>
7.
8. #include <Eigen/Dense>
9.
10. int main(int argc, char **argv)
11. {
12.     if (argc != 2)
13.     {
14.         std::cout << "Usage: ./linear-algebra dim" << std::endl;
15.         return EXIT_FAILURE;

```

```
16. }
17. std::chrono::time_point<std::chrono::system_clock> start, end;
18. std::chrono::duration<double> elapsed_seconds;
19. std::time_t end_time;
20. std::cout << "Number of threads used by Eigen: " << Eigen::nbThreads()
21.             << std::endl;
22.
23. // Allocate matrices and right-hand side vector
24. start = std::chrono::system_clock::now();
25. int dim = std::atoi(argv[1]);
26. Eigen::MatrixXd A = Eigen::MatrixXd::Random(dim, dim);
27. Eigen::VectorXd b = Eigen::VectorXd::Random(dim);
28. end = std::chrono::system_clock::now();
29.
30. // Report times
31. elapsed_seconds = end - start;
32. end_time = std::chrono::system_clock::to_time_t(end);
33. std::cout << "matrices allocated and initialized "
34.             << std::put_time(std::localtime(&end_time), "%a %b %d %Y
35. %r\n")
36.             << "elapsed time: " << elapsed_seconds.count() << "s\n";
37.
38. start = std::chrono::system_clock::now();
39. // Save matrix and RHS
40. Eigen::MatrixXd A1 = A;
41. Eigen::VectorXd b1 = b;
42. end = std::chrono::system_clock::now();
43. end_time = std::chrono::system_clock::to_time_t(end);
44. std::cout << "Scaling done, A and b saved "
45.             << std::put_time(std::localtime(&end_time), "%a %b %d %Y %r\n")
46.             << "elapsed time: " << elapsed_seconds.count() << "s\n";
47. start = std::chrono::system_clock::now();
48. Eigen::VectorXd x = A.lu().solve(b);
49. end = std::chrono::system_clock::now();
50.
51. // Report times
52. elapsed_seconds = end - start;
53. end_time = std::chrono::system_clock::to_time_t(end);
54. double relative_error = (A * x - b).norm() / b.norm();
55. std::cout << "Linear system solver done "
56.             << std::put_time(std::localtime(&end_time), "%a %b %d %Y %r\n")
57.             << "elapsed time: " << elapsed_seconds.count() << "s\n";
```

```

58.     std::cout << "relative error is " << relative_error << std::endl;
59.
60.     return 0;
61. }
```

矩阵-向量乘法和LU分解是在Eigen库中实现的，但是可以选择BLAS和LAPACK库中的实现。在这个示例中，我们只考虑BLAS库中的实现。

具体实施

这个示例中，我们将用到Eigen和BLAS库，以及OpenMP。使用OpenMP将Eigen并行化，并从BLAS库中卸载部分线性代数实现：

- 首先声明CMake最低版本、项目名称和使用C++11语言标准：

```

1. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
2.
3. project(recipe-07 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

- 因为Eigen可以使用共享内存的方式，所以可以使用OpenMP并行处理计算密集型操作：

```
1. find_package(OpenMP REQUIRED)
```

- 调用 `find_package` 来搜索Eigen(将在下一小节中讨论)：

```
1. find_package(Eigen3 3.3 REQUIRED CONFIG)
```

- 如果找到Eigen，我们将打印状态信息。注意，使用的是 `Eigen3::Eigen`，这是一个 `IMPORT` 目标，可通过提供的CMake脚本找到这个目标：

```

1. if(TARGET Eigen3::Eigen)
    message(STATUS "Eigen3 v${EIGEN3_VERSION_STRING} found in
2. ${EIGEN3_INCLUDE_DIR}")
3. endif()
```

- 接下来，将源文件声明为可执行目标：

```
1. add_executable(linear-algebra linear-algebra.cpp)
```

6. 然后，找到BLAS。注意，现在不需要依赖项：

```
1. find_package(BLAS)
```

7. 如果找到BLAS，我们可为可执行目标，设置相应的宏定义和链接库：

```
1. if(BLAS_FOUND)
2.   message(STATUS "Eigen will use some subroutines from BLAS.")
   message(STATUS "See: http://eigen.tuxfamily.org/dox-
3.   devel/TopicUsingBlasLapack.html")
4.   target_compile_definitions(linear-algebra
5.     PRIVATE
6.       EIGEN_USE_BLAS
7.     )
8.   target_link_libraries(linear-algebra
9.     PUBLIC
10.    ${BLAS_LIBRARIES}
11.   )
12. else()
13.   message(STATUS "BLAS not found. Using Eigen own functions")
14. endif()
```

8. 最后，我们链接到 `Eigen3::Eigen` 和 `OpenMP::OpenMP_CXX` 目标。这就可以设置所有必要的编译标示和链接标志：

```
1. target_link_libraries(linear-algebra
2.   PUBLIC
3.     Eigen3::Eigen
4.     OpenMP::OpenMP_CXX
5.   )
```

9. 开始配置：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- ...
6. -- Found OpenMP_CXX: -fopenmp (found version "4.5")
```

```

7. -- Found OpenMP: TRUE (found version "4.5")
8. -- Eigen3 v3.3.4 found in /usr/include/eigen3
9. --
10. -- Found BLAS: /usr/lib/libblas.so
11. -- Eigen will use some subroutines from BLAS.
12. -- See: http://eigen.tuxfamily.org/dox-devel/TopicUsingBlasLapack.html

```

10. 最后，编译并测试代码。注意，可执行文件使用四个线程运行：

```

1. $ cmake --build .
2. $ ./linear-algebra 1000
3.
4. Number of threads used by Eigen: 4
5. matrices allocated and initialized Sun Jun 17 2018 11:04:20 AM
6. elapsed time: 0.0492328s
7. Scaling done, A and b saved Sun Jun 17 2018 11:04:20 AM
8. elapsed time: 0.0492328s
9. Linear system solver done Sun Jun 17 2018 11:04:20 AM
10. elapsed time: 0.483142s
11. relative error is 4.21946e-13

```

工作原理

Eigen支持CMake查找，这样配置项目就会变得很容易。从3.3版开始，Eigen提供了CMake模块，这些模块将导出相应的目标 `Eigen3::Eigen`。

`find_package` 可以通过选项传递，届时CMake将不会使用 `FindEigen3.cmake` 模块，而是通过特定的 `Eigen3Config.cmake`，`Eigen3ConfigVersion.cmake` 和 `Eigen3Targets.cmake` 提供 Eigen3安装的标准位置(`<installation-prefix>/share/eigen3/cmake`)。这种包定位模式称为“Config”模式，比 `Find<package>.cmake` 方式更加通用。有关“模块”模式和“配置”模式的更多信息，可参考官方文档

https://cmake.org/cmake/help/v3.5/command/find_package.html。

虽然Eigen3、BLAS和OpenMP声明为 `PUBLIC` 依赖项，但 `EIGEN_USE_BLAS` 编译定义声明为 `PRIVATE`。可以在单独的库目标中汇集库依赖项，而不是直接链接可执行文件。使用 `PUBLIC/PRIVATE` 关键字，可以根据库目标的依赖关系调整相应标志和定义。

更多信息

CMake将在预定义的位置层次结构中查找配置模块。首先

是 `CMAKE_PREFIX_PATH` , `<package>_DIR` 是接下来的搜索路径。因此，如果Eigen3安装在非标准位置，可以使用这两个选项来告诉CMake在哪里查找它：

1. 通过将Eigen3的安装前缀传递给 `CMAKE_PREFIX_PATH` :

```
1. $ cmake -D CMAKE_PREFIX_PATH=<installation-prefix> ..
```

2. 通过传递配置文件的位置作为 `Eigen3_DIR` :

```
1. $ cmake -D Eigen3_DIR=<installation-prefix>/share/eigen3/cmake ..
```

3.8 检测Boost库

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-08> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Boost是一组C++通用库。这些库提供了许多功能，这些功能在现代C++项目中不可或缺，但是还不能通过C++标准使用这些功能。例如，Boost为元编程、处理可选参数和文件系统操作等提供了相应的组件。这些库中有许多特性后来被C++11、C++14和C++17标准所采用，但是对于保持与旧编译器兼容性的代码库来说，许多Boost组件仍然是首选。

本示例将向您展示如何检测和链接Boost库的一些组件。

准备工作

我们将编译的源码是Boost提供的文件系统库与文件系统交互的示例。这个库可以跨平台使用，并将操作系统和文件系统之间的差异抽象为一致的API。下面的代码(`path-info.cpp`)将接受一个路径作为参数，并将其组件的报告打印到屏幕上：

```

1. #include <iostream>
2.
3. #include <boost/filesystem.hpp>
4.
5. using namespace std;
6. using namespace boost::filesystem;
7. const char *say_what(bool b) { return b ? "true" : "false"; }
8. int main(int argc, char *argv[])
9. {
10.     if (argc < 2)
11.     {
12.         cout
13.             << "Usage: path_info path-element [path-element...]\n"
14.             "Composes a path via operator/= from one or more path-element
15. arguments\n"
16.             "Example: path_info foo/bar baz\n"
17.             "#ifdef BOOST_POSIX_API
18.                 " would report info about the composed path foo/bar/baz\n";
19.             "#else // BOOST_WINDOWS_API
20.                 " would report info about the composed path foo/bar\\baz\n";
21.             "#endif

```

```

21.     return 1;
22. }
23. path p;
24. for ( ; argc > 1; --argc, ++argv)
25.     p /= argv[1]; // compose path p from the command line arguments
26. cout << "\ncomposed path:\n";
27. cout << " operator<<()-----: " << p << "\n";
28. cout << " make_preferred()-----: " << p.make_preferred() << "\n";
29. cout << "\nelements:\n";
30. for (auto element : p)
31.     cout << " " << element << '\n';
32. cout << "\nobservers, native format:" << endl;
33. #ifdef BOOST_POSIX_API
34.     cout << " native()-----: " << p.native() << endl;
35.     cout << " c_str()-----: " << p.c_str() << endl;
36. #else // BOOST_WINDOWS_API
37.     wcout << L" native()-----: " << p.native() << endl;
38.     wcout << L" c_str()-----: " << p.c_str() << endl;
39. #endif
40.     cout << " string()-----: " << p.string() << endl;
41.     wcout << L" wstring()-----: " << p.wstring() << endl;
42.     cout << "\nobservers, generic format:\n";
43.     cout << " generic_string()-----: " << p.generic_string() << endl;
44.     wcout << L" generic_wstring()-----: " << p.generic_wstring() << endl;
45.     cout << "\ndecomposition:\n";
46.     cout << " root_name()-----: " << p.root_name() << '\n';
47.     cout << " root_directory()-----: " << p.root_directory() << '\n';
48.     cout << " root_path()-----: " << p.root_path() << '\n';
49.     cout << " relative_path()-----: " << p.relative_path() << '\n';
50.     cout << " parent_path()-----: " << p.parent_path() << '\n';
51.     cout << " filename()-----: " << p.filename() << '\n';
52.     cout << " stem()-----: " << p.stem() << '\n';
53.     cout << " extension()-----: " << p.extension() << '\n';
54.     cout << "\nquery:\n";
55.     cout << " empty()-----: " << say_what(p.empty()) << '\n';
56.     cout << " is_absolute()-----: " << say_what(p.is_absolute()) << '\n';
57.     cout << " has_root_name()-----: " << say_what(p.has_root_name()) << '\n';
58.     cout << " has_root_directory()-----: " << say_what(p.has_root_directory()) <<
59.         '\n';
60.     cout << " has_root_path()-----: " << say_what(p.has_root_path()) << '\n';
61.     cout << " has_relative_path()-----: " << say_what(p.has_relative_path()) <<
62.         '\n';
63.     cout << " has_parent_path()-----: " << say_what(p.has_parent_path()) << '\n';

```

```

62.     cout << " has_filename()-----: " << say_what(p.has_filename()) << '\n';
63.     cout << " has_stem()-----: " << say_what(p.has_stem()) << '\n';
64.     cout << " has_extension()-----: " << say_what(p.has_extension()) << '\n';
65.     return 0;
66. }
```

具体实施

Boost由许多不同的库组成，这些库可以独立使用。CMake可将这个库集合，表示为组件的集合。`FindBoost.cmake` 模块不仅可以搜索库集合的完整安装，还可以搜索集合中的特定组件及其依赖项(如果有的话)。我们将逐步建立相应的 `CMakeLists.txt`：

- 首先，声明CMake最低版本、项目名称、语言，并使用C++11标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-08 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

- 然后，使用 `find_package` 搜索Boost。若需要对Boost强制性依赖，需要一个参数。这个例子中，只需要文件系统组件，所以将它作为参数传递给 `find_package`：

```
1. find_package(Boost 1.54 REQUIRED COMPONENTS filesystem)
```

- 添加可执行目标，编译源文件：

```
1. add_executable(path-info path-info.cpp)
```

- 最后，将目标链接到Boost库组件。由于依赖项声明为 `PUBLIC`，依赖于Boost的目标将自动获取依赖项：

```

1. target_link_libraries(path-info
2.   PUBLIC
3.     Boost::filesystem
4.   )
```

工作原理

`FindBoost.cmake` 是本示例中所使用的CMake模块，其会在标准系统安装目录中找到Boost库。由于我们链接的是 `Boost::filesystem`，CMake将自动设置包含目录并调整编译和链接标志。如果 Boost库安装在非标准位置，可以在配置时使用 `BOOST_ROOT` 变量传递Boost安装的根目录，以便让 CMake搜索非标准路径：

```
1. $ cmake -D BOOST_ROOT=/custom/boost
```

或者，可以同时传递包含头文件的 `BOOST_INCLUDEDIR` 变量和库目录的 `BOOST_LIBRARYDIR` 变量：

```
$ cmake -D BOOST_INCLUDEDIR=/custom/boost/include -  
1. DBOOST_LIBRARYDIR=/custom/boost/lib
```

3.9 检测外部库：I. 使用pkg-config

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-09> 中找到，包含一个C的示例。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-09> 中也有一个适用于CMake 3.5的示例。

目前为止，我们已经讨论了两种检测外部依赖关系的方法：

- 使用CMake自带的 `find-module`，但并不是所有的包在CMake的 `find` 模块都找得到。
- 使用 `<package>Config.cmake`，`<package>ConfigVersion.cmake` 和 `<package>Targets.cmake`，这些文件由软件包供应商提供，并与软件包一起安装在标准位置的cmake文件夹下。

如果某个依赖项既不提供查找模块，也不提供供应商打包的CMake文件，该怎么办？在这种情况下，我们只有两个选择：

- 依赖 `pkg-config` 程序，来找到系统上的包。这依赖于包供应商在 `.pc` 配置文件中，其中有关于发行包的元数据。
- 为依赖项编写自己的 `find-package` 模块。

本示例中，将展示如何利用CMake中的 `pkg-config` 来定位ZeroMQ消息库。下一个示例中，将编写一个find模块，展示如何为ZeroMQ编写属于自己 `find` 模块。

准备工作

我们构建的代码来自ZeroMQ手册 <http://zguide.zeromq.org/page:all> 的示例。由两个源文件 `hwserver.c` 和 `hwclient.c` 组成，这两个源文件将构建为两个独立的可执行文件。执行时，它们将打印“Hello, World”。

具体实施

这是一个C项目，我们将使用C99标准，逐步构建 `CMakeLists.txt` 文件：

1. 声明一个C项目，并要求符合C99标准：

```
1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2.
3. project(recipe-09 LANGUAGES C)
```

```

4.
5. set(CMAKE_C_STANDARD 99)
6. set(CMAKE_C_EXTENSIONS OFF)
7. set(CMAKE_C_STANDARD_REQUIRED ON)

```

2. 使用CMake附带的find-module，查找 `pkg-config`。这里在 `find_package` 中传递了 `QUIET` 参数。只有在没有找到 `pkg-config` 时，CMake才会报错：

```
1. find_package(PkgConfig REQUIRED QUIET)
```

3. 找到 `pkg-config` 时，我们将使用 `pkg_search_module` 函数，以搜索任何附带包配置 `.pc` 文件的库或程序。该示例中，我们查找ZeroMQ库：

```

1. pkg_search_module(
2.   ZeroMQ
3.   REQUIRED
4.   libzeromq libzmq lib0mq
5.   IMPORTED_TARGET
6. )

```

4. 如果找到ZeroMQ库，则打印状态消息：

```

1. if(TARGET PkgConfig::ZeroMQ)
2.   message(STATUS "Found ZeroMQ")
3. endif()

```

5. 然后，添加两个可执行目标，并链接到ZeroMQ。这将自动设置包括目录和链接库：

```

1. add_executable(hwserver hwserver.c)
2. target_link_libraries(hwserver PkgConfig::ZeroMQ)
3. add_executable(hwclient hwclient.c)
4. target_link_libraries(hwclient PkgConfig::ZeroMQ)

```

6. 现在，我们可以配置和构建示例：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .

```

7. 在终端中，启动服务器，启动时会输出类似于本例的消息：

```
1. Current 0MQ version is 4.2.2
```

8. 然后，在另一个终端启动客户端，它将打印如下内容：

```
1. Connecting to hello world server...
2. Sending Hello 0...
3. Received World 0
4. Sending Hello 1...
5. Received World 1
6. Sending Hello 2...
7. ...
```

工作

当找到 `pkg-config` 时，CMake需要提供两个函数，来封装这个程序提供的功能：

- `pkg_check_modules`，查找传递列表中的所有模块(库和/或程序)
- `pkg_search_module`，要在传递的列表中找到第一个工作模块

与 `find_package` 一样，这些函数接受 `REQUIRED` 和 `QUIET` 参数。更详细地说，我们对 `pkg_search_module` 的调用如下：

```
1. pkg_search_module(
2.   ZeroMQ
3.   REQUIRED
4.   libzeromq libzmq lib0mq
5.   IMPORTED_TARGET
6. )
```

这里，第一个参数是前缀，它将用于命名存储搜索ZeroMQ库结果的目标：`PkgConfig::ZeroMQ`。注意，我们需要为系统上的库名传递不同的选项：`libzeromq`、`libzmq` 和 `lib0mq`。这是因为不同的操作系统和包管理器，可为同一个包选择不同的名称。

NOTE: `pkg_check_modules` 和 `pkg_search_module` 函数添加了 `IMPORTED_TARGET` 选项，并在 CMake 3.6中定义导入目标的功能。3.6之前的版本，只定义了变量 `ZeroMQ_INCLUDE_DIRS` (用于 `include`目录)和 `ZeroMQ_LIBRARIES` (用于链接库)，供后续使用。

3.10 检测外部库:II. 自定义find模块

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-03/recipe-10> 中找到，包含一个C的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

此示例补充了上一节的示例，我们将展示如何编写一个 `find` 模块来定位系统上的ZeroMQ消息库，以便能够在非Unix操作系统上检测该库。我们重用服务器-客户端示例代码。

如何实施

这是一个C项目，使用C99标准，并逐步构建CMakeLists.txt文件：

1. 声明一个C项目，并要求符合C99标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-10 LANGUAGES C)
4.
5. set(CMAKE_C_STANDARD 99)
6. set(CMAKE_C_EXTENSIONS OFF)
7. set(CMAKE_C_STANDARD_REQUIRED ON)

```

2. 将当前源目录 `CMAKE_CURRENT_SOURCE_DIR`，添加到CMake将查找模块的路径列表 `CMAKE_MODULE_PATH` 中。这样CMake就可以找到，我们自定义的 `FindZeroMQ.cmake` 模块：

```
1. list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
```

3. 现在 `FindZeroMQ.cmake` 模块是可用的，可以通过这个模块来搜索项目所需的依赖项。由于我们没有使用 `QUIET` 选项来查找 `find_package`，所以当找到库时，状态消息将自动打印：

```
1. find_package(ZeroMQ REQUIRED)
```

4. 我们继续添加 `hwserver` 可执行目标。头文件包含目录和链接库是使用 `find_package` 命令成功后，使用 `ZeroMQ_INCLUDE_DIRS` 和 `ZeroMQ_LIBRARIES` 变量进行指定的：

```

1. add_executable(hwserver hwserver.c)
2. target_include_directories(hwserver

```

```

3.    PRIVATE
4.      ${ZeroMQ_INCLUDE_DIRS}
5.    )
6. target_link_libraries(hwserver
7.    PRIVATE
8.      ${ZeroMQ_LIBRARIES}
9.    )

```

5. 最后，我们对 `hwclient` 可执行目标执行相同的操作：

```

1. add_executable(hwclient hwclient.c)
2. target_include_directories(hwclient
3.    PRIVATE
4.      ${ZeroMQ_INCLUDE_DIRS}
5.    )
6. target_link_libraries(hwclient
7.    PRIVATE
8.      ${ZeroMQ_LIBRARIES}
9.    )

```

此示例的主 `CMakeLists.txt` 在使用 `FindZeroMQ.cmake` 时，与前一个示例中使用的 `CMakeLists.txt` 不同。这个模块使用 `find_path` 和 `find_library` CMake内置命令，搜索 ZeroMQ头文件和库，并使用 `find_package_handle_standard_args` 设置相关变量，就像我们在第3节中做的那样。

1. `FindZeroMQ.cmake` 中，检查了 `ZeroMQ_ROOT` 变量是否设置。此变量可用于ZeroMQ库的检测，并引导到自定义安装目录。用户可能设置了 `ZeroMQ_ROOT` 作为环境变量，我们也会进行检查了：

```

1. if(NOT ZeroMQ_ROOT)
2.   set(ZeroMQ_ROOT "$ENV{ZeroMQ_ROOT}")
3. endif()

```

2. 然后，搜索系统上 `zmq.h` 头文件的位置。这是基于 `_ZeroMQ_ROOT` 变量和 `find_path` 命令进行的：

```

1. if(NOT ZeroMQ_ROOT)
2.   find_path(_ZeroMQ_ROOT NAMES include/zmq.h)
3. else()
4.   set(_ZeroMQ_ROOT "${ZeroMQ_ROOT}")
5. endif()

```

```

6.
7. find_path(ZeroMQ_INCLUDE_DIRS NAMES zmq.h HINTS ${_ZeroMQ_ROOT}/include)

```

3. 如果成功找到头文件，则将 `ZeroMQ_INCLUDE_DIRS` 设置为其位置。我们继续通过使用字符串操作和正则表达式，寻找相应版本的ZeroMQ库：

```

1. set(_ZeroMQ_H ${ZeroMQ_INCLUDE_DIRS}/zmq.h)
2.
3. function(_zmqver_EXTRACT _ZeroMQ_VER_COMPONENT _ZeroMQ_VER_OUTPUT)
4. set(CMAKE_MATCH_1 "0")
   set(_ZeroMQ_expr "^[ \t]*#define[ \t]+${_ZeroMQ_VER_COMPONENT}[ \t]+
5. ([0-9]+)$")
6. file(STRINGS "${_ZeroMQ_H}" _ZeroMQ_ver REGEX "${_ZeroMQ_expr}")
7. string(REGEX MATCH "${_ZeroMQ_expr}" ZeroMQ_ver "${_ZeroMQ_ver}")
8. set(${_ZeroMQ_VER_OUTPUT} "${CMAKE_MATCH_1}" PARENT_SCOPE)
9. endfunction()
10.
11. _zmqver_EXTRACT("ZMQ_VERSION_MAJOR" ZeroMQ_VERSION_MAJOR)
12. _zmqver_EXTRACT("ZMQ_VERSION_MINOR" ZeroMQ_VERSION_MINOR)
13. _zmqver_EXTRACT("ZMQ_VERSION_PATCH" ZeroMQ_VERSION_PATCH)

```

4. 然后，为 `find_package_handle_standard_args` 准备 `ZeroMQ_VERSION` 变量：

```

1. if(ZeroMQ_FIND_VERSION_COUNT GREATER 2)
   set(ZeroMQ_VERSION
2. "${ZeroMQ_VERSION_MAJOR}.${ZeroMQ_VERSION_MINOR}.${ZeroMQ_VERSION_PATCH}")
3. else()
4.   set(ZeroMQ_VERSION "${ZeroMQ_VERSION_MAJOR}.${ZeroMQ_VERSION_MINOR}")
5. endif()

```

5. 使用 `find_library` 命令搜索ZeroMQ库。因为库的命名有所不同，这里我们需要区分Unix的平台和Windows平台：

```

1. if(NOT ${CMAKE_C_PLATFORM_ID} STREQUAL "Windows")
2.   find_library(ZeroMQ_LIBRARIES
3.     NAMES
4.       zmq
5.     HINTS
6.       ${_ZeroMQ_ROOT}/lib
7.       ${_ZeroMQ_ROOT}/lib/x86_64-linux-gnu
8.   )
9. else()

```

```

10.    find_library(ZeroMQ_LIBRARIES
11.        NAMES
12.            libzmq
13.            "libzmq-mt-
14.            ${ZeroMQ_VERSION_MAJOR}_${ZeroMQ_VERSION_MINOR}_${ZeroMQ_VERSION_PATCH}"
15.            "libzmq-${CMAKE_VS_PLATFORM_TOOLSET}-mt-
16.            ${ZeroMQ_VERSION_MAJOR}_${ZeroMQ_VERSION_MINOR}_${ZeroMQ_VERSION_PATCH}"
17.            "libzmq-${CMAKE_VS_PLATFORM_TOOLSET}-mt-gd-
18.            ${ZeroMQ_VERSION_MAJOR}_${ZeroMQ_VERSION_MINOR}_${ZeroMQ_VERSION_PATCH}"
19.        HINTS
20.            ${_ZeroMQ_ROOT}/lib
21.    )
endif()

```

6. 最后，包含了标准 `FindPackageHandleStandardArgs.cmake`，并调用相应的CMake命令。如果找到所有需要的变量，并且版本匹配，则将 `ZeroMQ_FOUND` 变量设置为 `TRUE`：

```

1. include(FindPackageHandleStandardArgs)
2.
3. find_package_handle_standard_args(ZeroMQ
4.     FOUND_VAR
5.         ZeroMQ_FOUND
6.     REQUIRED_VARS
7.         ZeroMQ_INCLUDE_DIRS
8.     ZeroMQ_LIBRARIES
9.     VERSION_VAR
10.    ZeroMQ_VERSION
11. )

```

NOTE: 刚才描述的 `FindZeroMQ.cmake` 模块已经在

<https://github.com/zeromq/azmq/blob/master/config/FindZeroMQ.cmake> 上进行了修改。

工作原理

`find-module` 通常遵循特定的模式：

1. 检查用户是否为所需的包提供了自定义位置。

2. 使用 `find_` 家族中的命令搜索所需包的必需组件，即头文件、库、可执行程序等等。我们使用 `find_path` 查找头文件的完整路径，并使用 `find_library` 查找库。CMake还提供 `find_file` 、 `find_program` 和 `find_package` 。这些命令的签名如下：

1. `find_path(<VAR> NAMES name PATHS paths)`

3. 如果搜索成功， `<VAR>` 将保存搜索结果；如果搜索失败，则会设置为 `<VAR>-NOTFOUND` 。 `NAMES` 和 `PATHS` 分别是CMake应该查找的文件的名称和搜索应该指向的路径。
4. 初步搜索的结果中，可以提取版本号。示例中，ZeroMQ头文件包含库版本，可以使用字符串操作和正则表达式提取库版本信息。
5. 最后，调用 `find_package_handle_standard_args` 命令。处理 `find_package` 命令的 `REQUIRED` 、 `QUIET` 和版本参数，并设置 `ZeroMQ_FOUND` 变量。

NOTE:任何CMake命令的完整文档都可以从命令行获得。例如，`cmake --help-command find_file` 将输出 `find_file` 命令的手册页。对于CMake标准模块的手册，可以在CLI使用 `--help-module` 看到。例如，`cmake --help-module FindPackageHandleStandardArgs` 将输出 `FindPackageHandleStandardArgs.cmake` 的手册页面。

更多信息

总而言之，有四种方式可用于找到依赖包：

1. 使用由包供应商提供CMake文件 `<package>Config.cmake` ， `<package>ConfigVersion.cmake` 和 `<package>Targets.cmake` ，通常会在包的标准安装位置查找。
2. 无论是由CMake还是第三方提供的模块，为所需包使用 `find-module` 。
3. 使用 `pkg-config` ，如本节的示例所示。
4. 如果这些都不可行，那么编写自己的 `find` 模块。

这四种可选方案按相关性进行了排序，每种方法也都有其挑战。

目前，并不是所有的包供应商都提供CMake的Find文件，不过正变得越来越普遍。因为导出CMake目标，使得第三方代码很容易使用它所依赖的库和/或程序附加的依赖。

从一开始，`Find-module` 就一直是CMake中定位依赖的主流手段。但是，它们中的大多数仍然依赖于设置依赖项使用的变量，比如 `Boost_INCLUDE_DIRS` 、 `PYTHON_INTERPRETER` 等等。这种方式很难在第三方发布自己的包时，确保依赖关系被满足。

使用 `pkg-config` 的方法可以很好地进行适配，因为它已经成为Unix系统的标准。然而，也由于这个原因，它不是一个完全跨平台的方法。此外，如CMake文档所述，在某些情况下，用户可能会意外地覆

盖检测包，并导致 `pkg-config` 提供不正确的信息。

最后的方法是编写自己的查找模块脚本，就像本示例中那样。这是可行的，并且依赖于 `FindPackageHandleStandardArgs.cmake`。然而，编写一个全面的查找模块脚本绝非易事；有需要考虑很多可能性，我们在Unix和Windows平台上，为查找ZeroMQ库文件演示了一个例子。

所有软件开发人员都非常清楚这些问题和困难，正如CMake邮件列表上讨论所示：

<https://cmake.org/pipermail/cmake/2018-May/067556.html>。`pkg-config` 在Unix包开发人员中是可以接受的，但是它不能很容易地移植到非Unix平台。CMake配置文件功能强大，但并非所有软件开发人员都熟悉CMake语法。公共包规范项目是统一用于包查找的 `pkg-config` 和CMake配置文件方法的最新尝试。您可以在项目的网站上找到更多信息：

<https://mwoehlke.github.io/cps/>

在第10章中将讨论，如何使用前面讨论中概述的第一种方法，使第三方应用程序，找到自己的包：为项目提供自己的CMake查找文件。

第4章 创建和运行测试

本章的主要内容有：

- 创建一个简单的单元测试
- 使用Catch2库进行单元测试
- 使用Google Test库进行单元测试
- 使用Boost Test进行单元测试
- 使用动态分析来检测内存缺陷
- 预期测试失败
- 使用超时测试运行时间过长的测试
- 并行测试
- 运行测试子集
- 使用测试固件

测试代码是开发工具的核心组件。通过单元测试和集成测试自动化测试，不仅可以帮助开发人员尽早回归功能检测，还可以帮助开发人员参与，并了解项目。它可以帮助新开发人员向项目代码提交修改，并确保预期的功能性。对于验证安装是否保留了代码的功能时，自动化测试必不可少。从一开始对单元、模块或库进行测试，可以使用一种纯函数式的风格，将全局变量和全局状态最小化，可让开发者的具有更模块化、更简单的编程风格。

本章中，我们将演示如何使用流行的测试库和框架，将测试集成到CMake构建结构中，并谨记以下目标：

- 让用户、开发人员和持续集成服务很容易地运行测试集。应该像使用 `Unix Makefile` 时，键入 `make test` 一样简单。
- 通过最小化测试时间，高效地运行测试，最大限度地提高运行测试的概率——理想情况下，每次代码修改都该如此。

4.1 创建一个简单的单元测试

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-01> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

CTest是CMake的测试工具，本示例中，我们将使用CTest进行单元测试。为了保持对CMake/CTest的关注，我们的测试代码会尽可能的简单。计划是编写和测试能够对整数求和的代码，示例代码只会对整数进行累加，不处理浮点数。就像年轻的卡尔·弗里德里希·高斯(Carl Friedrich Gauss)，被他的老师测试从1到100求和所有自然数一样，我们将要求代码做同样的事情。为了说明CMake没有对实际测试的语言进行任何限制，我们不仅使用C++可执行文件测试代码，还使用Python脚本和shell脚本作为测试代码。为了简单起见，我们将不使用任何测试库来实现，但是我们将在后面的示例中介绍C++测试框架。

准备工作

代码示例由三个文件组成。实现源文件 `sum_integs.cpp` 对整数向量进行求和，并返回累加结果：

```

1. #include "sum_integers.hpp"
2.
3. #include <vector>
4.
5. int sum_integers(const std::vector<int> integers) {
6.     auto sum = 0;
7.     for (auto i : integers) {
8.         sum += i;
9.     }
10.    return sum;
11. }
```

这个示例是否是优雅的实现并不重要，接口以 `sum_integers` 的形式导出。接口在 `sum_integers.hpp` 文件中声明，详情如下：

```

1. #pragma once
2.
3. #include <vector>
4.
5. int sum_integers(const std::vector<int> integers);
```

最后，`main`函数在 `main.cpp` 中定义，从 `argv[]` 中收集命令行参数，将它们转换成整数向量，调用 `sum_integers` 函数，并将结果打印到输出中：

```

1. #include "sum_integers.hpp"
2.
3. #include <iostream>
4. #include <string>
5. #include <vector>
6.
7. // we assume all arguments are integers and we sum them up
8. // for simplicity we do not verify the type of arguments
9. int main(int argc, char *argv[]) {
10.     std::vector<int> integers;
11.     for (auto i = 1; i < argc; i++) {
12.         integers.push_back(std::stoi(argv[i]));
13.     }
14.     auto sum = sum_integers(integers);
15.
16.     std::cout << sum << std::endl;
17. }
```

测试这段代码使用C++实现(`test.cpp`)，Bash shell脚本实现(`test.sh`)和Python脚本实现(`test.py`)，只要实现可以返回一个零或非零值，从而CMake可以解释为成功或失败。

C++例子(`test.cpp`)中，我们通过调用 `sum_integers` 来验证 $1 + 2 + 3 + 4 + 5 = 15$:

```

1. #include "sum_integers.hpp"
2.
3. #include <vector>
4.
5. int main() {
6.     auto integers = {1, 2, 3, 4, 5};
7.
8.     if (sum_integers(integers) == 15) {
9.         return 0;
10.    } else {
11.        return 1;
12.    }
13. }
```

Bash shell脚本调用可执行文件：

```

1. #!/usr/bin/env bash
2.
3. EXECUTABLE=$1
4.
5. OUTPUT=$( $EXECUTABLE 1 2 3 4 )
6.
7. if [ "$OUTPUT" = "10" ]
8. then
9.     exit 0
10. else
11.     exit 1
12. fi

```

此外，Python脚本调用可执行文件(使用 `--executable` 命令行参数传递)，并使用 `--short` 命令行参数执行：

```

1. import subprocess
2. import argparse
3.
4. # test script expects the executable as argument
5. parser = argparse.ArgumentParser()
6. parser.add_argument('--executable',
7.                     help='full path to executable')
8. parser.add_argument('--short',
9.                     default=False,
10.                    action='store_true',
11.                    help='run a shorter test')
12. args = parser.parse_args()
13.
14. def execute_cpp_code(integers):
15.     result = subprocess.check_output([args.executable] + integers)
16.     return int(result)
17.
18. if args.short:
19.     # we collect [1, 2, ..., 100] as a list of strings
20.     result = execute_cpp_code([str(i) for i in range(1, 101)])
21.     assert result == 5050, 'summing up to 100 failed'
22. else:
23.     # we collect [1, 2, ..., 1000] as a list of strings
24.     result = execute_cpp_code([str(i) for i in range(1, 1001)])
25.     assert result == 500500, 'summing up to 1000 failed'

```

具体实施

现在，我们将逐步描述如何为项目设置测试：

- 对于这个例子，我们需要C++11支持，可用的Python解释器，以及Bash shell：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-01 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
8.
9. find_package(PythonInterp REQUIRED)
10. find_program(BASH_EXECUTABLE NAMES bash REQUIRED)

```

- 然后，定义库及主要可执行文件的依赖关系，以及测试可执行文件：

```

1. # example library
2. add_library(sum_integers sum_integers.cpp)
3.
4. # main code
5. add_executable(sum_up main.cpp)
6. target_link_libraries(sum_up sum_integers)
7.
8. # testing binary
9. add_executable(cpp_test test.cpp)
10. target_link_libraries(cpp_test sum_integers)

```

- 最后，打开测试功能并定义四个测试。最后两个测试，调用相同的Python脚本，先没有任何命令行参数，再使用 `--short`：

```

1. enable_testing()
2.
3. add_test(
4.   NAME bash_test
      COMMAND ${BASH_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.sh
5.   $<TARGET_FILE:sum_up>
6. )
7.
8. add_test(

```

```

9.    NAME cpp_test
10.   COMMAND $<TARGET_FILE:cpp_test>
11. )
12.
13. add_test(
14.   NAME python_test_long
15.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py --
16.   executable ${TARGET_FILE:sum_up}
17. )
18. add_test(
19.   NAME python_test_short
20.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py --short
21.   --executable ${TARGET_FILE:sum_up}
22. )

```

4. 现在，我们已经准备好配置和构建代码。先手动进行测试：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./sum_up 1 2 3 4 5
6.
7. 15

```

5. 然后，我们可以用 `ctest` 运行测试集：

```

1. $ ctest
2.
3. Test project /home/user/cmake-recipes/chapter-04/recipe-01/cxx-
4. example/build
5. Start 1: bash_test
6. 1/4 Test #1: bash_test ..... Passed 0.01 sec
7. Start 2: cpp_test
8. 2/4 Test #2: cpp_test ..... Passed 0.00 sec
9. Start 3: python_test_long
10. 3/4 Test #3: python_test_long ..... Passed 0.06 sec
11. Start 4: python_test_short
12. 4/4 Test #4: python_test_short ..... Passed 0.05 sec
13. 100% tests passed, 0 tests failed out of 4
14. Total Test time (real) = 0.12 sec

```

6. 还应该尝试中断实现，以验证测试集是否能捕捉到更改。

工作原理

这里的两个关键命令：

- `enable_testing()`，测试这个目录和所有子文件夹(因为我们把它放在主 `CMakeLists.txt`)。
- `add_test()`，定义了一个新的测试，并设置测试名称和运行命令。

```
1. add_test(
2.   NAME cpp_test
3.   COMMAND $<TARGET_FILE:cpp_test>
4. )
```

上面的例子中，使用了生成器表达式：`$<TARGET_FILE:cpp_test>`。生成器表达式，是在生成构建系统生成时的表达式。我们将在第5章第9节中详细地描述生成器表达式。此时，我们可以声明 `$<TARGET_FILE:cpp_test>` 变量，将使用 `cpp_test` 可执行目标的完整路径进行替换。

生成器表达式在测试时非常方便，因为不必显式地将可执行程序的位置和名称，可以硬编码到测试中。以一种可移植的方式实现这一点非常麻烦，因为可执行文件和可执行后缀(例如，Windows上是 `.exe` 后缀)的位置在不同的操作系统、构建类型和生成器之间可能有所不同。使用生成器表达式，我们不必显式地了解位置和名称。

也可以将参数传递给要运行的 `test` 命令，例如：

```
1. add_test(
2.   NAME python_test_short
3.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py --short --
4. )
```

这个例子中，我们按顺序运行测试，并展示如何缩短总测试时间并行执行测试(第8节)，执行测试用例的子集(第9节)。这里，可以自定义测试命令，可以以任何编程语言运行测试集。CTest关心的是，通过命令的返回码测试用例是否通过。CTest遵循的标准约定是，返回零意味着成功，非零返回意味着失败。可以返回零或非零的脚本，都可以做测试用例。

既然知道了如何定义和执行测试，那么了解如何诊断测试失败也很重要。为此，我们可以在代码中引入一个bug，让所有测试都失败：

```
1. Start 1: bash_test
2. 1/4 Test #1: bash_test .....***Failed 0.01 sec
```

```

3.      Start 2: cpp_test
4.  2/4 Test #2: cpp_test .....***Failed 0.00 sec
5.      Start 3: python_test_long
6.  3/4 Test #3: python_test_long .....***Failed 0.06 sec
7.      Start 4: python_test_short
8.  4/4 Test #4: python_test_short .....***Failed 0.06 sec
9.
10. 0% tests passed, 4 tests failed out of 4
11.
12. Total Test time (real) = 0.13 sec
13.
14. The following tests FAILED:
15. 1 - bash_test (Failed)
16. 2 - cpp_test (Failed)
17. 3 - python_test_long (Failed)
18. 4 - python_test_short (Failed)
19. Errors while running CTest

```

如果我们想了解更多，可以查看文件 `test/Temporary/lasttestsfailure.log`。这个文件包含测试命令的完整输出，并且在分析阶段，要查看的第一个地方。使用以下CLI开关，可以从CTest获得更详细的测试输出：

- `--output-on-failure` : 将测试程序生成的任何内容打印到屏幕上，以免测试失败。
- `-v` : 将启用测试的详细输出。
- `-vv` : 启用更详细的输出。

CTest提供了一个非常方便快捷的方式，可以重新运行以前失败的测试；要使用的CLI开关是 `--rerun-failed`，在调试期间非常有用。

更多信息

考虑以下定义：

```

1. add_test(
2.   NAME python_test_long
3.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py --executable
4.   $<TARGET_FILE:sum_up>
5. )

```

前面的定义可以通过显式指定脚本运行的 `WORKING_DIRECTORY` 重新表达，如下：

```
1. add_test(
```

4.1 创建一个简单的单元测试

```
2. NAME python_test_long
3. COMMAND ${PYTHON_EXECUTABLE} test.py --executable ${TARGET_FILE:sum_up}
4. WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
5. )
```

测试名称可以包含 / 字符，按名称组织相关测试也很有用，例如：

```
1. add_test(
2.   NAME python/long
3.   COMMAND ${PYTHON_EXECUTABLE} test.py --executable ${TARGET_FILE:sum_up}
4.   WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
5. )
```

有时候，我们需要为测试脚本设置环境变量。这可以通过 `set_tests_properties` 实现：

```
1. set_tests_properties(python_test
2.   PROPERTIES
3.     ENVIRONMENT
4.     ACCOUNT_MODULE_PATH=${CMAKE_CURRENT_SOURCE_DIR}
5.     ACCOUNT_HEADER_FILE=${CMAKE_CURRENT_SOURCE_DIR}/account/account.h
6.     ACCOUNT_LIBRARY_FILE=${TARGET_FILE:account}
7. )
```

这种方法在不同的平台上并不总可行，CMake提供了解决这个问题的方法。下面的代码片段与上面给出的代码片段相同，在执行实际的Python测试脚本之前，通过 `CMAKE_COMMAND` 调用CMake来预先设置环境变量：

```
1. add_test(
2.   NAME
3.     python_test
4.   COMMAND
5.     ${CMAKE_COMMAND} -E env
6.     ACCOUNT_MODULE_PATH=${CMAKE_CURRENT_SOURCE_DIR}
7.     ACCOUNT_HEADER_FILE=${CMAKE_CURRENT_SOURCE_DIR}/account/account.h
8.     ACCOUNT_LIBRARY_FILE=${TARGET_FILE:account}
9.     ${PYTHON_EXECUTABLE}
10.    ${CMAKE_CURRENT_SOURCE_DIR}/account/test.py
11. )
```

同样，要注意使用生成器表达式 `${TARGET_FILE:account}` 来传递库文件的位置。

我们已经使用 `ctest` 命令执行测试，CMake还将为生成器创建目标(Unix Makefile生成器

为 `make test` , Ninja工具为 `ninja test` , 或者Visual Studio为 `RUN_TESTS`)。这意味着, 还有另一种(几乎)可移植的方法来运行测试:

```
1. $ cmake --build . --target test
```

不幸的是, 当使用Visual Studio生成器时, 我们需要使用 `RUN_TESTS` 来代替:

```
1. $ cmake --build . --target RUN_TESTS
```

NOTE: `ctest` 提供了丰富的命令行参数。其中一些内容将在以后的示例中探讨。要获得完整的列表, 需要使用 `ctest --help` 来查看。命令 `cmake --help-manual ctest` 会将向屏幕输出完整的 `ctest` 手册。

4.2 使用Catch2库进行单元测试

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-02> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的配置中，使用返回码来表示 `test.cpp` 测试的成功或失败。对于简单功能没问题，但是通常情况下，我们想要使用一个测试框架，它提供了相关基础设施来运行更复杂的测试，包括固定方式进行测试，与数值公差的比较，以及在测试失败时输出更好的错误报告。这里，我们用目前比较流行的测试库 Catch2(<https://github.com/catchorg/Catch2>)来进行演示。这个测试框架有个很好的特性，它可以通过单个头库包含在项目中进行测试，这使得编译和更新框架特别容易。这个配置中，我们将CMake和Catch2结合使用，来测试上一个求和代码。

我们需要 `catch.hpp` 头文件，可以从 <https://github.com/catchorg/Catch2> (我们使用的是版本2.0.1)下载，并将它与 `test.cpp` 一起放在项目的根目录下。

准备工作

`main.cpp`、`sum_integers.cpp` 和 `sum_integers.hpp` 与之前的示例相同，但将更新 `test.cpp`：

```

1. #include "sum_integers.hpp"
2.
3. // this tells catch to provide a main()
4. // only do this in one cpp file
5. #define CATCH_CONFIG_MAIN
6. #include "catch.hpp"
7. #include <vector>
8.
9. TEST_CASE("Sum of integers for a short vector", "[short]")
10. {
11.     auto integers = {1, 2, 3, 4, 5};
12.     REQUIRE(sum_integers(integers) == 15);
13. }
14.
15. TEST_CASE("Sum of integers for a longer vector", "[long]")
16. {
17.     std::vector<int> integers;
18.     for (int i = 1; i < 1001; ++i)
19.     {

```

```

20.     integers.push_back(i);
21. }
22. REQUIRE(sum_integers(integers) == 500500);
23. }
```

`catch.hpp` 头文件可以从<https://github.com/catchorg/Catch2> (版本为2.0.1)下载，并将它与 `test.cpp` 放在项目的根目录中。

具体实施

使用Catch2库，需要修改之前的所使用 `CMakeList.txt`：

1. 保持 `CMakeLists.txt` 大多数部分内容不变：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-02 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # example library
13. add_library(sum_integers sum_integers.cpp)
14.
15. # main code
16. add_executable(sum_up main.cpp)
17. target_link_libraries(sum_up sum_integers)
18.
19. # testing binary
20. add_executable(cpp_test test.cpp)
21. target_link_libraries(cpp_test sum_integers)
```

2. 对于上一个示例的配置，需要保留一个测试，并重命名它。注意，`--success` 选项可传递给单兀测试的可执行文件。这是一个Catch2选项，测试成功时，也会有输出：

```

1. enable_testing()
2.
```

```

3. add_test(
4.   NAME catch_test
5.   COMMAND $<TARGET_FILE:cpp_test> --success
6. )

```

3. 就是这样！让我们来配置、构建和测试。CTest中，使用 `-V` 选项运行测试，以获得单元测试可执行文件的输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest -V
6.
7. UpdateCTestConfiguration from :/home/user/cmake-cookbook/chapter-04/recipe-02/example/build/DartConfiguration.tcl
8. UpdateCTestConfiguration from :/home/user/cmake-cookbook/chapter-04/recipe-02/example/build/DartConfiguration.tcl
9. Test project /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/build
10. Constructing a list of tests
11. Done constructing a list of tests
12. Updating test list for fixtures
13. Added 0 tests to meet fixture requirements
14. Checking test dependency graph...
15. Checking test dependency graph end
16. test 1
17. Start 1: catch_test
18. 1: Test command: /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/build/cpp_test "--success"
19. 1: Test timeout computed to be: 100000000
20. 1:
21. 1: ~~~~~
22. 1: cpp_test is a Catch v2.0.1 host application.
23. 1: Run with -? for options
24. 1:
25. 1: -----
26. 1: Sum of integers for a short vector
27. 1: -----
28. 1: /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:10
29. 1: .....
30. 1:
31. 1: /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:12:

```

```

32. 1: PASSED:
33. 1: REQUIRE( sum_integers(integers) == 15 )
34. 1: with expansion:
35. 1: 15 == 15
36. 1:
37. 1: -----
38. 1: Sum of integers for a longer vector
39. 1: -----
40. 1: /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:15
41. 1: .....
42. 1:
43. 1: /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:20:
44. 1: PASSED:
45. 1: REQUIRE( sum_integers(integers) == 500500 )
46. 1: with expansion:
47. 1: 500500 (0x7a314) == 500500 (0x7a314)
48. 1:
49. 1: =====
50. 1: All tests passed (2 assertions in 2 test cases)
51. 1:
52. 1/1 Test #1: catch_test ..... Passed 0.00 s
53.
54. 100% tests passed, 0 tests failed out of 1
55.
56. Total Test time (real) = 0.00 se

```

4. 我们也可以测试 `cpp_test` 的二进制文件，可以直接从Catch2中看到输出：

```

1. $ ./cpp_test --success
2.
3. -----
4. cpp_test is a Catch v2.0.1 host application.
5. Run with -? for options
6. -----
7. Sum of integers for a short vector
8. -----
9. /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:10
10. .....
11. /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:12:
12. PASSED:
13. REQUIRE( sum_integers(integers) == 15 )

```

```

14. with expansion:
15. 15 == 15
16. -----
17. Sum of integers for a longer vector
18. -----
19. /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:15
20. .....
21. /home/user/cmake-cookbook/chapter-04/recipe-02/cxx-example/test.cpp:20:
22. PASSED:
23. REQUIRE( sum_integers(integers) == 500500 )
24. with expansion:
25. 500500 (0x7a314) == 500500 (0x7a314)
26. =====
27. All tests passed (2 assertions in 2 test cases)

```

5. Catch2将生成一个可执行文件，还可以尝试执行以下命令，以探索单元测试框架提供的选项：

```
1. $ ./cpp_test --help
```

工作原理

Catch2是一个单头文件测试框架，所以不需要定义和构建额外的目标。只需要确保CMake能找到 `catch.hpp`，从而构建 `test.cpp` 即可。为了方便起见，将它放在与 `test.cpp` 相同的目录中，我们可以选择一个不同的位置，并使用 `target_include_directory` 指示该位置。另一种方法是将头部封装到接口库中，这可以在Catch2文档中说明(
<https://github.com/catchorg/catch2/blob/master/docs/build.systems.md#cmake>)：

```

1. # Prepare "Catch" library for other executables
2. set(CATCH_INCLUDE_DIR
3. ${CMAKE_CURRENT_SOURCE_DIR}/catch)
4.
5. add_library(Catch
6. INTERFACE)
7.
8. target_include_directories(Catch INTERFACE
9. ${CATCH_INCLUDE_DIR})

```

然后，我们对库进行如下链接：

1. target_link_libraries(`cpp_test Catch`)

回想一下第3中的讨论，在第1章从简单的可执行库到接口库，是CMake提供的伪目标库，这些伪目标库对于指定项目外部目标的需求非常有用。

更多信息

这是一个简单的例子，主要关注CMake。当然，Catch2提供了更多功能。有关Catch2框架的完整文档，可访问 <https://github.com/catchorg/Catch2>。

Catch2代码库包含有CMake函数，用于解析Catch测试并自动创建CMake测试，不需要显式地输入 `add_test()` 函数，可见

<https://github.com/catchorg/Catch2/blob/master/contrib/ParseAndAddCatchTests.cmake>。

4.3 使用Google Test库进行单元测试

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-03> 中找到，包含一个C++的示例。该示例在CMake 3.11版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。在代码库中，有一个支持CMake 3.5的例子。

本示例中，我们将演示如何在CMake的帮助下使用Google Test框架实现单元测试。与前一个配置相比，Google Test框架不仅仅是一个头文件，也是一个库，包含两个需要构建和链接的文件。可以将它们与我们的代码项目放在一起，但是为了使代码项目更加轻量级，我们将选择在配置时，下载一个定义良好的Google Test，然后构建框架并链接它。我们将使用较新的 `FetchContent` 模块(从CMake版本3.11开始可用)。第8章中会继续讨论 `FetchContent`，在这里将讨论模块在底层是如何工作的，并且还将演示如何使用 `ExternalProject_Add` 进行模拟。此示例的灵感来自(改编自) <https://cmake.org/cmake/help/v3.11/module/FetchContent.html> 示例。

准备工作

`main.cpp`、`sum_integers.cpp` 和 `sum_integers.hpp` 与之前相同，修改 `test.cpp`：

```

1. #include "sum_integers.hpp"
2. #include "gtest/gtest.h"
3.
4. #include <vector>
5.
6. int main(int argc, char **argv) {
7.     ::testing::InitGoogleTest(&argc, argv);
8.     return RUN_ALL_TESTS();
9. }
10.
11. TEST(example, sum_zero) {
12.     auto integers = {1, -1, 2, -2, 3, -3};
13.     auto result = sum_integers(integers);
14.     ASSERT_EQ(result, 0);
15. }
16.
17. TEST(example, sum_five) {
18.     auto integers = {1, 2, 3, 4, 5};
19.     auto result = sum_integers(integers);
20.     ASSERT_EQ(result, 15);
21. }
```

如上面的代码所示，我们显式地将 `gtest.h`，而不将其他Google Test源放在代码项目存储库中，会在配置时使用 `FetchContent` 模块下载它们。

具体实施

下面的步骤描述了如何设置 `CMakeLists.txt`，使用GTest编译可执行文件及其相应的测试：

- 与前两个示例相比，`CMakeLists.txt` 的开头基本没有变化，CMake 3.11才能使用 `FetchContent` 模块：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.11 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-03 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11. set(CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS ON)
12.
13. # example library
14. add_library(sum_integers sum_integers.cpp)
15.
16. # main code
17. add_executable(sum_up main.cpp)
18. target_link_libraries(sum_up sum_integers)
```

- 然后引入一个 `if`，检查 `ENABLE_UNIT_TESTS`。默认情况下，它为 `ON`，但有时需要设置为 `OFF`，以免在没有网络连接时，也能使用Google Test：

```

1. option(ENABLE_UNIT_TESTS "Enable unit tests" ON)
2. message(STATUS "Enable testing: ${ENABLE_UNIT_TESTS}")
3.
4. if(ENABLE_UNIT_TESTS)
5.     # all the remaining CMake code will be placed here
6. endif()
```

- `if` 内部包含 `FetchContent` 模块，声明要获取的新内容，并查询其属性：

```

1. include(FetchContent)
2.
3. FetchContent_Declare(
4.   googletest
5.   GIT_REPOSITORY https://github.com/google/googletest.git
6.   GIT_TAG release-1.8.0
7. )
8.
9. FetchContent_GetProperties(googletest)

```

4. 如果内容还没有获取到，将尝试获取并配置它。这需要添加几个可以链接的目标。本例中，我们对 `gtest_main` 感兴趣。该示例还包含一些变通方法，用于使用在Visual Studio下的编译：

```

1. if(NOT googletest_POPULATED)
2.   FetchContent_Populate(googletest)
3.
4.   # Prevent GoogleTest from overriding our compiler/linker options
5.   # when building with Visual Studio
6.   set(gtest_force_shared_crt ON CACHE BOOL "") FORCE)
7.   # Prevent GoogleTest from using PThreads
8.   set(gtest_disable_pthreads ON CACHE BOOL "") FORCE)
9.
10.  # adds the targets: gtest, gtest_main, gmock, gmock_main
11.  add_subdirectory(
12.    ${googletest_SOURCE_DIR}
13.    ${googletest_BINARY_DIR}
14.  )
15.
16.  # Silence std::tr1 warning on MSVC
17.  if(MSVC)
18.    foreach(_tgt gtest gtest_main gmock gmock_main)
19.      target_compile_definitions(${_tgt}
20.        PRIVATE
21.          "_SILENCE_TR1_NAMESPACE_DEPRECATED_WARNING"
22.      )
23.    endforeach()
24.  endif()
25. endif()

```

5. 然后，使用 `target_sources` 和 `target_link_libraries` 命令，定义 `cpp_test` 可执行目标并指定它的源文件：

```

1. add_executable(cpp_test "")
2.
3. target_sources(cpp_test
4.   PRIVATE
5.     test.cpp
6.   )
7.
8. target_link_libraries(cpp_test
9.   PRIVATE
10.    sum_integers
11.    gtest_main
12. )

```

6. 最后，使用 `enable_testing` 和 `add_test` 命令来定义单元测试：

```

1. enable_testing()
2.
3. add_test(
4.   NAME google_test
5.   COMMAND ${TARGET_FILE:cpp_test}
6. )

```

7. 现在，准备配置、构建和测试项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
7.      Test project /home/user/cmake-cookbook/chapter-04/recipe-03/cxx-
8. example/build
9.      Start 1: google_test
10.     1/1 Test #1: google_test ..... Passed 0.00 sec
11. 100% tests passed, 0 tests failed out of 1
12.
13. Total Test time (real) = 0.00 sec

```

8. 可以直接运行 `cpp_test`：

```

1. $ ./cpp_test
2.
3. [=====] Running 2 tests from 1 test case.
4. [-----] Global test environment set-up.
5. [-----] 2 tests from example
6. [ RUN ] example.sum_zero
7. [ OK ] example.sum_zero (0 ms)
8. [ RUN ] example.sum_five
9. [ OK ] example.sum_five (0 ms)
10. [-----] 2 tests from example (0 ms total)
11.
12. [-----] Global test environment tear-down
13. [=====] 2 tests from 1 test case ran. (0 ms total)
14. [ PASSED ] 2 tests.

```

工作原理

`FetchContent` 模块支持通过 `ExternalProject` 模块，在配置时填充内容，并在其3.11版本中成为CMake的标准部分。而 `ExternalProject_Add()` 在构建时(见第8章)进行下载操作，这样 `FetchContent` 模块使得构建可以立即进行，这样获取的主要项目和外部项目(在本例中为Google Test)仅在第一次执行CMake时调用，使用 `add_subdirectory` 可以嵌套。

为了获取Google Test，首先声明外部内容：

```

1. include(FetchContent)
2.
3. FetchContent_Declare(
4.     googletest
5.     GIT_REPOSITORY https://github.com/google/googletest.git
6.     GIT_TAG release-1.8.0
7. )

```

本例中，我们获取了一个带有特定标记的Git库(release-1.8.0)，但是我们也可以从Subversion、Mercurial或HTTP(S)源获取一个外部项目。有关可用选项，可参考相应的 `ExternalProject_Add` 命令的选项，网址是<https://cmake.org/cmake/help/v3.11/module/ExternalProject.html>。

调用 `FetchContent_Populate()` 之前，检查是否已经使用 `FetchContent_GetProperties()` 命令处理了内容填充；否则，调用 `FetchContent_Populate()` 超过一次后，就会抛出错误。

`FetchContent_Populate(googletest)` 用于填充源并定

义 `googletest_SOURCE_DIR` 和 `googletest_BINARY_DIR`，可以使用它们来处理Google Test项目(使用 `add_subdirectory()`，因为它恰好也是一个CMake项目)：

```
1. add_subdirectory(
2.   ${googletest_SOURCE_DIR}
3.   ${googletest_BINARY_DIR}
4. )
```

前面定义了以下目标：`gtest`、`gtest_main`、`gmock` 和 `gmock_main`。这个配置中，作为单元测试示例的库依赖项，我们只对 `gtest_main` 目标感兴趣：

```
1. target_link_libraries(cpp_test
2.   PRIVATE
3.   sum_integers
4.   gtest_main
5. )
```

构建代码时，可以看到如何正确地对Google Test进行配置和构建。有时，我们希望升级到更新的Google Test版本，这时需要更改的唯一一行就是详细说明 `GIT_TAG` 的那一行。

更多信息

了解了 `FetchContent` 及其构建时的近亲 `ExternalProject_Add`，我们将在第8章中重新讨论这些命令。有关可用选项的详细讨论，可参考<https://cmake.org/cmake/help/v3.11/module/FetchContent.html>。

本示例中，我们在配置时获取源代码，也可以将它们安装在系统环境中，并使用 `FindGTest` 模块来检测库和头文件(<https://cmake.org/cmake/help/v3.5/module/FindTest.html>)。从3.9版开始，CMake还提供了一个Google Test模块(<https://cmake.org/cmake/help/v3.9/module/GoogleTest.html>)，它提供了一个 `gtest_add_tests` 函数。通过搜索Google Test宏的源代码，可以使用此函数自动添加测试。

当然，Google Test有许多有趣的特性，可在<https://github.com/google/googletest> 查看。

4.4 使用Boost Test进行单元测试

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-04> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Boost Test是在C++社区中，一个非常流行的单元测试框架。本例中，我们将演示如何使用Boost Test，对求和示例代码进行单元测试。

准备工作

`main.cpp`、`sum_integers.cpp` 和 `sum_integers.hpp` 与之前的示例相同，将更新 `test.cpp` 作为使用Boost Test库进行的单元测试：

```

1. #include "sum_integers.hpp"
2.
3. #include <vector>
4.
5. #define BOOST_TEST_MODULE example_test_suite
6. #include <boost/test/unit_test.hpp>
7. BOOST_AUTO_TEST_CASE(add_example)
8. {
9.     auto integers = {1, 2, 3, 4, 5};
10.    auto result = sum_integers(integers);
11.    BOOST_REQUIRE(result == 15);
12. }
```

具体实施

以下是使用Boost Test构建项目的步骤：

1. 先从 `CMakeLists.txt` 开始：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-04 LANGUAGES CXX)
6.
```

```

7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # example library
13. add_library(sum_integers sum_integers.cpp)
14.
15. # main code
16. add_executable(sum_up main.cpp)
17. target_link_libraries(sum_up sum_integers)

```

2. 检测Boost库并将 `cpp_test` 链接到它：

```

1. find_package(Boost 1.54 REQUIRED COMPONENTS unit_test_framework)
2.
3. add_executable(cpp_test test.cpp)
4.
5. target_link_libraries(cpp_test
6.   PRIVATE
7.   sum_integers
8.   Boost::unit_test_framework
9. )
10.
11. # avoid undefined reference to "main" in test.cpp
12. target_compile_definitions(cpp_test
13.   PRIVATE
14.   BOOST_TEST_DYN_LINK
15. )

```

3. 最后，定义单元测试：

```

1. enable_testing()
2.
3. add_test(
4.   NAME boost_test
5.   COMMAND $<TARGET_FILE:cpp_test>
6. )

```

4. 下面是需要配置、构建和测试代码的所有内容：

```

1. $ mkdir -p build

```

```

2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
    Test project /home/user/cmake-recipes/chapter-04/recipe-04/cxx-
7. example/build
8. Start 1: boost_test
9. 1/1 Test #1: boost_test ..... Passed 0.01 sec
10. 100% tests passed, 0 tests failed out of 1
11. Total Test time (real) = 0.01 sec
12.
13. $ ./cpp_test
14.
15. Running 1 test case...
16. *** No errors detected

```

工作原理

使用 `find_package` 来检测Boost的 `unit_test_framework` 组件(参见第3章，第8节)。我们认为这个组件是 `REQUIRED` 的，如果在系统环境中找不到它，配置将停止。`cpp_test` 目标需要知道在哪里可以找到Boost头文件，并且需要链接到相应的库；它们都由 `IMPORTED` 库目标 `Boost::unit_test_framework` 提供，该目标由 `find_package` 设置。

更多信息

本示例中，我们假设系统上安装了Boost。或者，我们可以在编译时获取并构建Boost依赖项。然而，Boost不是轻量级依赖项。我们的示例代码中，我们只使用了最基本的设施，但是Boost提供了丰富的特性和选项，有兴趣的读者可以去这里看

看：http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/index.html。

4.5 使用动态分析来检测内存缺陷

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-05> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

内存缺陷：写入或读取越界，或者内存泄漏(已分配但从未释放的内存)，会产生难以跟踪的bug，最好尽早将它们检查出来。Valgrind(<http://valgrind.org>)是一个通用的工具，用来检测内存缺陷和内存泄漏。本节中，我们将在使用CMake/CTest测试时使用Valgrind对内存问题进行警告。

准备工作

对于这个配置，需要三个文件。第一个是测试的实现(我们可以调用文件 `leaky_implementation.cpp`)：

```

1. #include "leaky_implementation.hpp"
2.
3. int do_some_work() {
4.
5.     // we allocate an array
6.     double *my_array = new double[1000];
7.
8.     // do some work
9.     // ...
10.
11.    // we forget to deallocate it
12.    // delete[] my_array;
13.
14.    return 0;
15. }
```

还需要相应的头文件(`leaky_implementation.hpp`)：

```

1. #pragma once
2.
3. int do_some_work();
```

并且，需要测试文件(`test.cpp`)：

```
1. #include "leaky_implementation.hpp"
```

```

2.
3. int main() {
4.     int return_code = do_some_work();
5.
6.     return return_code;
7. }
```

我们希望测试通过，因为 `return_code` 硬编码为 `0`。这里我们也期望检测到内存泄漏，因为 `my_array` 没有释放。

具体实施

下面展示了如何设置CMakeLists.txt来执行代码动态分析：

1. 我们首先定义CMake最低版本、项目名称、语言、目标和依赖关系：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-05 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
8.
9. add_library(example_library leaky_implementation.cpp)
10.
11. add_executable(cpp_test test.cpp)
12. target_link_libraries(cpp_test example_library)
```

2. 然后，定义测试目标，还定义了 `MEMORYCHECK_COMMAND`：

```

1. find_program(MEMORYCHECK_COMMAND NAMES valgrind)
2. set(MEMORYCHECK_COMMAND_OPTIONS "--trace-children=yes --leak-check=full")
3.
4. # add memcheck test action
5. include(CTest)
6.
7. enable_testing()
8.
9. add_test(
10.   NAME cpp_test
11.   COMMAND $<TARGET_FILE:cpp_test>
```

12.)

3. 运行测试集，报告测试通过情况，如下所示：

```

1. $ ctest
2.
3.     Test project /home/user/cmake-recipes/chapter-04/recipe-05/cxx-
4. example/build
5. Start 1: cpp_test
6. 1/1 Test #1: cpp_test ..... Passed 0.00 sec
7. 100% tests passed, 0 tests failed out of 1
8. Total Test time (real) = 0.00 sec

```

4. 现在，我们希望检查内存缺陷，可以观察到被检测到的内存泄漏：

```

1. $ ctest -T memcheck
2.
3. Site: myhost
4. Build name: Linux-c++
5. Create new tag: 20171127-1717 - Experimental
    Memory check project /home/user/cmake-recipes/chapter-04/recipe-05/cxx-
6. example/build
7. Start 1: cpp_test
8. 1/1 MemCheck #1: cpp_test ..... Passed 0.40 sec
9. 100% tests passed, 0 tests failed out of 1
10. Total Test time (real) = 0.40 sec
11. -- Processing memory checking output:
12. 1/1 MemCheck: #1: cpp_test ..... Defects: 1
13. MemCheck log files can be found here: (* corresponds to test number)
14. /home/user/cmake-recipes/chapter-04/recipe-05/cxx-example/build/Testing/Temp
15. Memory checking results:
16. Memory Leak - 1

```

5. 最后一步，应该尝试修复内存泄漏，并验证 `ctest -T memcheck` 没有报告错误。

工作原理

使用 `find_program(MEMORYCHECK_COMMAND NAMES valgrind)` 查找valgrind，并将 `MEMORYCHECK_COMMAND` 设置为其绝对路径。我们显式地包含CTest模块来启用 `memcheck` 测试操作，可以使用 `CTest -T memcheck` 来启用这个操作。此外，使用 `set(MEMORYCHECK_COMMAND_OPTIONS "--trace-children=yes --leak-check=full")`，将相关

参数传递给Valgrind。内存检查会创建一个日志文件，该文件可用于详细记录内存缺陷信息。

NOTE:一些工具，如代码覆盖率和静态分析工具，可以进行类似地设置。然而，其中一些工具的使用更加复杂，因为需要专门的构建和工具链。*Sanitizers*就是这样一个例子。有关更多信息，请参见<https://github.com/arsenm/sanitizers-cmake>。另外，请参阅第14章，其中讨论了*AddressSanitizer*和*ThreadSanitizer*。

更多信息

该方法可向测试面板报告内存缺陷，这里演示的功能也可以独立于测试面板使用。我们将在第14章中重新讨论，与CDash一起使用的情况。

有关Valgrind及其特性和选项的文档，请参见<http://valgrind.org>。

4.6 预期测试失败

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-06> 中找到，包含一个C++的示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

理想情况下，我们希望所有的测试能在每个平台上通过。然而，也可能想要测试预期的失败或异常是否会在受控的设置中进行。这种情况下，我们将把预期的失败定义为成功。我们认为，这通常应该交给测试框架(例如：Catch2或Google Test)的任务，它应该检查预期的失败并向CMake报告成功。但是，在某些情况下，您可能希望将测试的非零返回代码定义为成功；换句话说，您可能想要颠倒成功和失败的定义。在本示例中，我们将演示这种情况。

准备工作

这个配置的测试用例是一个很小的Python脚本(`test.py`)，它总是返回1，CMake将其解释为失败：

```
1. import sys
2.
3. # simulate a failing test
4. sys.exit(1)
```

实施步骤

如何编写CMakeLists.txt来完成我们的任务：

1. 这个示例中，不需要任何语言支持从CMake，但需要Python：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-06 LANGUAGES NONE)
3. find_package(PythonInterp REQUIRED)
```

2. 然后，定义测试并告诉CMake，测试预期会失败：

```
1. enable_testing()
2. add_test(example ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py)
3. set_tests_properties(example PROPERTIES WILL_FAIL true)
```

3. 最后，报告是一个成功的测试，如下所示：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
7. Test project /home/user/cmake-recipes/chapter-04/recipe-06/example/build
8. Start 1: example
9. 1/1 Test #1: example ..... Passed 0.00 sec
10. 100% tests passed, 0 tests failed out of 1
11. Total Test time (real) = 0.01 sec

```

工作原理

使用 `set_tests_properties(example PROPERTIES WILL_FAIL true)`，将属性 `WILL_FAIL` 设置为 `true`，这将转换成功与失败。但是，这个特性不应该用来临时修复损坏的测试。

更多信息

如果需要更大的灵活性，可以将测试属性 `PASS_REGULAR_EXPRESSION` 和 `FAIL_REGULAR_EXPRESSION` 与 `set_tests_properties` 组合使用。如果设置了这些参数，测试输出将根据参数给出的正则表达式列表进行检查，如果匹配了正则表达式，测试将通过或失败。可以在测试中设置其他属性，完整的属性列表可以参考：<https://cmake.org/cmake/help/v3.5/manual/cmake-properties.7.html#properties-on-tests>。

4.7 使用超时测试运行时间过长的测试

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-07> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

理想情况下，测试集应该花很短的时间进行，以便开发人员经常运行测试，并使每个提交(变更集)进行测试成为可能(或更容易)。然而，有些测试可能会花费更长的时间或者被卡住(例如，由于高文件I/O负载)，我们可能需要设置超时来终止耗时过长的测试，它们延迟了整个测试，并阻塞了部署管道。本示例中，我们将演示一种设置超时的方法，可以针对每个测试设置不同的超时。

准备工作

这个示例是一个Python脚本(`test.py`)，它总是返回0。为了保持这种简单性，并保持对CMake方面的关注，测试脚本除了等待两秒钟外什么也不做。实际中，这个测试脚本将执行更有意义的工作：

```

1. import sys
2. import time
3.
4. # wait for 2 seconds
5. time.sleep(2)
6.
7. # report success
8. sys.exit(0)
```

具体实施

我们需要通知CTest终止测试，如下：

1. 我们定义项目名称，启用测试，并定义测试：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name
5. project(recipe-07 LANGUAGES NONE)
6.
7. # detect python
8. find_package(PythonInterp REQUIRED)
```

```

9.
10. # define tests
11. enable_testing()
12.
13. # we expect this test to run for 2 seconds
14. add_test(example ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py)

```

2. 另外，我们为测试指定时限，设置为10秒：

```
1. set_tests_properties(example PROPERTIES TIMEOUT 10)
```

3. 知道了如何进行配置和构建，并希望测试能够通过：

```

1. $ ctest
2.
3. Test project /home/user/cmake-recipes/chapter-04/recipe-07/example/build
4. Start 1: example
5. 1/1 Test #1: example ..... Passed 2.01 sec
6. 100% tests passed, 0 tests failed out of 1
7. Total Test time (real) = 2.01 sec

```

4. 现在，为了验证超时是否有效，我们将 `test.py` 中的 `sleep` 命令增加到11秒，并重新运行测试：

```

1. $ ctest
2.
3. Test project /home/user/cmake-recipes/chapter-04/recipe-07/example/build
4. Start 1: example
5. 1/1 Test #1: example .....***Timeout 10.01 sec
6. 0% tests passed, 1 tests failed out of 1
7. Total Test time (real) = 10.01 sec
8. The following tests FAILED:
9. 1 - example (Timeout)
10. Errors while running CTest

```

工作原理

`TIMEOUT` 是一个方便的属性，可以使用 `set_tests_properties` 为单个测试指定超时时间。如果测试运行超过了这个设置时间，不管出于什么原因（测试已经停止或者机器太慢），测试将被终止并标记为失败。

4.8 并行测试

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-08> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

大多数现代计算机都有4个或更多个CPU核芯。CTest有个非常棒的特性，能够并行运行测试，如果您有多个可用的核。这可以减少测试的总时间，而减少总测试时间才是真正重要的，从而开发人员频繁地进行测试。本示例中，我们将演示这个特性，并讨论如何优化测试以获得最大的性能。

其他测试可以进行相应地表示，我们把这些测试脚本放在 `CMakeLists.txt` 同目录下面的test目录中。

准备工作

我们假设测试集包含标记为a, b, ..., j的测试用例，每一个都有特定的持续时间：

测试用例	该单元的耗时
a, b, c, d	0.5
e, f, g	1.5
h	2.5
i	3.5
j	4.5

时间单位可以是分钟，但是为了保持简单和简短，我们将使用秒。为简单起见，我们可以用Python脚本表示 `test a`，它消耗0.5个时间单位：

```

1. import sys
2. import time
3.
4. # wait for 0.5 seconds
5. time.sleep(0.5)
6.
7. # finally report success
8. sys.exit(0)

```

其他测试同理。我们将把这些脚本放在 `CMakeLists.txt` 下面，一个名为 `test` 的目录中。

具体实施

对于这个示例，我们需要声明一个测试列表，如下：

1. `CMakeLists.txt` 非常简单：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name
5. project(recipe-08 LANGUAGES NONE)
6.
7. # detect python
8. find_package(PythonInterp REQUIRED)
9.
10. # define tests
11. enable_testing()
12. add_test(a ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/a.py)
13. add_test(b ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/b.py)
14. add_test(c ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/c.py)
15. add_test(d ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/d.py)
16. add_test(e ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/e.py)
17. add_test(f ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/f.py)
18. add_test(g ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/g.py)
19. add_test(h ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/h.py)
20. add_test(i ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/i.py)
21. add_test(j ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/j.py)

```

2. 我们可以配置项目，使用 `ctest` 运行测试，总共需要17秒：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ ctest
5.
6. Start 1: a
7. 1/10 Test #1: a ..... Passed 0.51 sec
8. Start 2: b
9. 2/10 Test #2: b ..... Passed 0.51 sec
10. Start 3: c
11. 3/10 Test #3: c ..... Passed 0.51 sec
12. Start 4: d
13. 4/10 Test #4: d ..... Passed 0.51 sec
14. Start 5: e

```

```

15. 5/10 Test #5: e ..... Passed 1.51 sec
16. Start 6: f
17. 6/10 Test #6: f ..... Passed 1.51 sec
18. Start 7: g
19. 7/10 Test #7: g ..... Passed 1.51 sec
20. Start 8: h
21. 8/10 Test #8: h ..... Passed 2.51 sec
22. Start 9: i
23. 9/10 Test #9: i ..... Passed 3.51 sec
24. Start 10: j
25. 10/10 Test #10: j ..... Passed 4.51 sec
26. 100% tests passed, 0 tests failed out of 10
27. Total Test time (real) = 17.11 sec

```

3. 现在，如果机器有4个内核可用，我们可以在不到5秒的时间内在4个内核上运行测试集：

```

1. $ ctest --parallel 4
2.
3. Start 10: j
4. Start 9: i
5. Start 8: h
6. Start 5: e
7. 1/10 Test #5: e ..... Passed 1.51 sec
8. Start 7: g
9. 2/10 Test #8: h ..... Passed 2.51 sec
10. Start 6: f
11. 3/10 Test #7: g ..... Passed 1.51 sec
12. Start 3: c
13. 4/10 Test #9: i ..... Passed 3.63 sec
14. 5/10 Test #3: c ..... Passed 0.60 sec
15. Start 2: b
16. Start 4: d
17. 6/10 Test #6: f ..... Passed 1.51 sec
18. 7/10 Test #4: d ..... Passed 0.59 sec
19. 8/10 Test #2: b ..... Passed 0.59 sec
20. Start 1: a
21. 9/10 Test #10: j ..... Passed 4.51 sec
22. 10/10 Test #1: a ..... Passed 0.51 sec
23. 100% tests passed, 0 tests failed out of 10
24. Total Test time (real) = 4.74 sec

```

工作原理

可以观察到，在并行情况下，测试j、i、h和e同时开始。当并行运行时，总测试时间会有显著的减少。观察 `ctest --parallel 4` 的输出，我们可以看到并行测试运行从最长的测试开始，最后运行最短的测试。从最长的测试开始是一个非常好的策略。这就像打包移动的盒子：从较大的项目开始，然后用较小的项目填补空白。a-j测试在4个核上的叠加比较，从最长的开始，如下图所示：

```
1. --> time
2. core 1: jjjjjjjjj
3. core 2: iiisiibd
4. core 3: hhhhggg
5. core 4: eeeeefac
```

按照定义测试的顺序运行，运行结果如下：

```
1. --> time
2. core 1: aeeeeiiiiii
3. core 2: bffffjjjjjjjjj
4. core 3: cggg
5. core 4: dhhhhh
```

按照定义测试的顺序运行测试，总的来说需要更多的时间，因为这会让2个核大部分时间处于空闲状态（这里的核3和核4）。CMake知道每个测试的时间成本，是因为我们先顺序运行了测试，将每个测试的成本数据记录在 `test/Temporary/CTestCostData.txt` 文件中：

```
1. a 1 0.506776
2. b 1 0.507882
3. c 1 0.508175
4. d 1 0.504618
5. e 1 1.51006
6. f 1 1.50975
7. g 1 1.50648
8. h 1 2.51032
9. i 1 3.50475
10. j 1 4.51111
```

如果在配置项目之后立即开始并行测试，它将按照定义测试的顺序运行测试，在4个核上的总测试时间明显会更长。这意味着什么呢？这意味着，我们应该减少的时间成本来安排测试？这是一种决策，但事实证明还有另一种方法，我们可以自己表示每次测试的时间成本：

```
1. add_test(a ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/a.py)
```

```
2. add_test(b ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/b.py)
3. add_test(c ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/c.py)
4. add_test(d ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/d.py)
5. set_tests_properties(a b c d PROPERTIES COST 0.5)
6.
7. add_test(e ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/e.py)
8. add_test(f ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/f.py)
9. add_test(g ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/g.py)
10. set_tests_properties(e f g PROPERTIES COST 1.5)
11.
12. add_test(h ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/h.py)
13. set_tests_properties(h PROPERTIES COST 2.5)
14.
15. add_test(i ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/i.py)
16. set_tests_properties(i PROPERTIES COST 3.5)
17.
18. add_test(j ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/j.py)
19. set_tests_properties(j PROPERTIES COST 4.5)
```

成本参数可以是一个估计值，也可以从 `test/Temporary/CTestCostData.txt` 中提取。

更多信息

除了使用 `ctest --parallel N`，还可以使用环境变量 `CTEST_PARALLEL_LEVEL` 将其设置为所需的级别。

4.9 运行测试子集

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-09> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例中，我们学习了如何在CMake的帮助下并行运行测试，并讨论了从最长的测试开始是最高效的。虽然，这种策略将总测试时间最小化，但是在特定特性的代码开发期间，或者在调试期间，我们可能不希望运行整个测试集。对于调试和代码开发，我们只需要能够运行选定的测试子集。在本示例中，我们将实现这一策略。

准备工作

在这个例子中，我们假设总共有六个测试：前三个测试比较短，名称分别为 `feature-a`、`feature-b` 和 `feature-c`，还有三个长测试，名称分别是 `feature-d`、`benchmark-a` 和 `benchmark-b`。这个示例中，我们可以用Python脚本表示这些测试，可以在其中调整休眠时间：

```

1. import sys
2. import time
3.
4. # wait for 0.1 seconds
5. time.sleep(0.1)
6.
7. # finally report success
8. sys.exit(0)
```

具体实施

以下是我们CMakeLists.txt文件内容的详细内容：

1. `CMakeLists.txt` 中，定义了六个测试：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. # project name
4. project(recipe-09 LANGUAGES NONE)
5.
6. # detect python
```

```

7. find_package(PythonInterp REQUIRED)
8.
9. # define tests
10. enable_testing()
11.
12. add_test(
13.   NAME feature-a
14.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
15.   a.py
16. )
17. add_test(
18.   NAME feature-b
19.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
20.   b.py
21. )
22. add_test(
23.   NAME feature-c
24.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
25.   c.py
26. )
27. add_test(
28.   NAME feature-d
29.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
30.   d.py
31. )
32. add_test(
33.   NAME benchmark-a
34.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/benchmark-
35.   a.py
36. )
37. add_test(
38.   NAME benchmark-b
39.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/benchmark-
40.   b.py
41. )

```

2. 此外，我们给较短的测试贴上 `quick` 的标签，给较长的测试贴上 `long` 的标签：

```

1. set_tests_properties(
2.   feature-a
3.   feature-b
4.   feature-c
5.   PROPERTIES

```

```

6.     LABELS "quick"
7.   )
8. set_tests_properties(
9.   feature-d
10.  benchmark-a
11.  benchmark-b
12.  PROPERTIES
13.    LABELS "long"
14.  )

```

3. 我们现在可以运行测试集了，如下：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ ctest
5.
6. Start 1: feature-a
7. 1/6 Test #1: feature-a ..... Passed 0.11 sec
8. Start 2: feature-b
9. 2/6 Test #2: feature-b ..... Passed 0.11 sec
10. Start 3: feature-c
11. 3/6 Test #3: feature-c ..... Passed 0.11 sec
12. Start 4: feature-d
13. 4/6 Test #4: feature-d ..... Passed 0.51 sec
14. Start 5: benchmark-a
15. 5/6 Test #5: benchmark-a ..... Passed 0.51 sec
16. Start 6: benchmark-b
17. 6/6 Test #6: benchmark-b ..... Passed 0.51 sec
18. 100% tests passed, 0 tests failed out of 6
19. Label Time Summary:
20. long = 1.54 sec*proc (3 tests)
21. quick = 0.33 sec*proc (3 tests)
22. Total Test time (real) = 1.87 sec

```

工作原理

现在每个测试都有一个名称和一个标签。CMake中所有的测试都是有编号的，所以它们也带有唯一编号。定义了测试标签之后，我们现在可以运行整个集合，或者根据它们的名称(使用正则表达式)、标签或编号运行测试。

按名称运行测试(运行所有具有名称匹配功能的测试)：

```
1. $ ctest -R feature
2.
3. Start 1: feature-a
4. 1/4 Test #1: feature-a ..... Passed 0.11 sec
5. Start 2: feature-b
6. 2/4 Test #2: feature-b ..... Passed 0.11 sec
7. Start 3: feature-c
8. 3/4 Test #3: feature-c ..... Passed 0.11 sec
9. Start 4: feature-d
10. 4/4 Test #4: feature-d ..... Passed 0.51 sec
11. 100% tests passed, 0 tests failed out of 4
```

按照标签运行测试(运行所有的长测试)：

```
1. $ ctest -L long
2.
3. Start 4: feature-d
4. 1/3 Test #4: feature-d ..... Passed 0.51 sec
5. Start 5: benchmark-a
6. 2/3 Test #5: benchmark-a ..... Passed 0.51 sec
7. Start 6: benchmark-b
8. 3/3 Test #6: benchmark-b ..... Passed 0.51 sec
9. 100% tests passed, 0 tests failed out of 3
```

根据数量运行测试(运行测试2到4)产生的结果是：

```
1. $ ctest -I 2,4
2.
3. Start 2: feature-b
4. 1/3 Test #2: feature-b ..... Passed 0.11 sec
5. Start 3: feature-c
6. 2/3 Test #3: feature-c ..... Passed 0.11 sec
7. Start 4: feature-d
8. 3/3 Test #4: feature-d ..... Passed 0.51 sec
9. 100% tests passed, 0 tests failed out of 3
```

更多信息

尝试使用 `$ ctest --help`，将看到有大量的选项可供用来定制测试。

4.10 使用测试固件

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-04/recipe-10> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

这个示例的灵感来自于Craig Scott，我们建议读者也参考相应的博客文章来了解更多的背景知识，<https://crascit.com/2016/10/18/test-fixtures-withcmake-ctest/>，此示例的动机是演示如何使用测试固件。这对于更复杂的测试非常有用，这些测试需要在测试运行前进行设置，以及在测试完成后执行清理操作(例如：创建示例数据库、设置连接、断开连接、清理测试数据库等等)。我们需要运行一个设置或清理操作的测试，并能够以一种可预测和健壮的方式自动触发这些步骤，而不需要引入代码重复。这些设置和清理步骤可以委托给测试框架(例如Google Test或Catch2)，我们在[这里](#)将演示如何在CMake级别实现测试固件。

准备工作

我们将准备4个Python脚本，并将它们放在 `test` 目录下：`setup.py`、`features-a.py`、`features-b.py` 和 `clean-up.py`。

具体实施

我们从 `CMakeLists.txt` 结构开始，附加一些步骤如下：

1. 基础CMake语句：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name
5. project(recipe-10 LANGUAGES NONE)
6.
7. # detect python
8. find_package(PythonInterp REQUIRED)
9.
10. # define tests
11. enable_testing()

```

2. 然后，定义了4个测试步骤，并将它们绑定到一个固件上：

```

1. add_test(
2.   NAME setup
3.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/setup.py
4. )
5. set_tests_properties(
6.   setup
7.   PROPERTIES
8.     FIXTURES_SETUP my(fixture
9.   )
10. add_test(
11.   NAME feature-a
12.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
13. a.py
14.   )
15. add_test(
16.   NAME feature-b
17.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/feature-
18. b.py
19.   )
20. set_tests_properties(
21.   feature-a
22.   feature-b
23.   PROPERTIES
24.     FIXTURES_REQUIRED my(fixture
25.   )
26. add_test(
27.   NAME cleanup
28.   COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test/cleanup.py
29.   )
30. set_tests_properties(
31.   cleanup
32.   PROPERTIES
33.     FIXTURES_CLEANUP my(fixture
34.   )

```

3. 运行整个集合，如下面的输出所示：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ ctest
5.

```

```

6. Start 1: setup
7. 1/4 Test #1: setup ..... Passed 0.01 sec
8. Start 2: feature-a
9. 2/4 Test #2: feature-a ..... Passed 0.01 sec
10. Start 3: feature-b
11. 3/4 Test #3: feature-b ..... Passed 0.00 sec
12. Start 4: cleanup
13. 4/4 Test #4: cleanup ..... Passed 0.01 sec
14.
15. 100% tests passed, 0 tests failed out of 4

```

4. 然而，当我们试图单独运行测试特性时。它正确地调用设置步骤和清理步骤：

```

1. $ ctest -R feature-a
2.
3. Start 1: setup
4. 1/3 Test #1: setup ..... Passed 0.01 sec
5. Start 2: feature-a
6. 2/3 Test #2: feature-a ..... Passed 0.00 sec
7. Start 4: cleanup
8. 3/3 Test #4: cleanup ..... Passed 0.01 sec
9.
10. 100% tests passed, 0 tests failed out of 3

```

工作原理

在本例中，我们定义了一个文本固件，并将其称为 `my-fixture`。我们为安装测试提供了 `FIXTURES_SETUP` 属性，并为清理测试了 `FIXTURES_CLEANUP` 属性，并且使用 `FIXTURES_REQUIRED`，我们确保测试 `feature-a` 和 `feature-b` 都需要安装和清理步骤才能运行。将它们绑定在一起，可以确保在定义良好的状态下，进入和离开相应的步骤。

第5章 配置时和构建时的操作

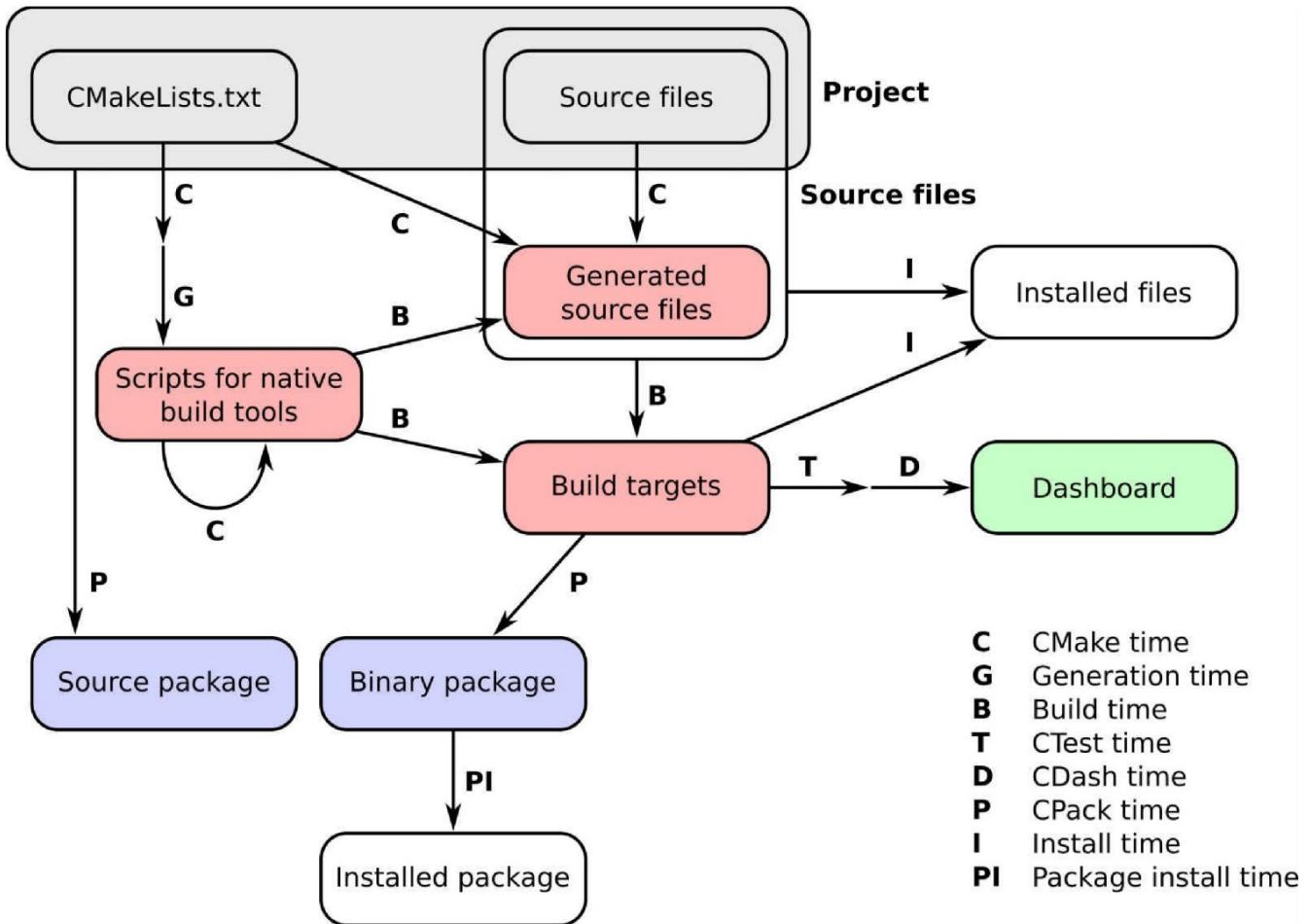
本章的主要内容有：

- 使用平台无关的文件操作
- 配置时运行自定义命令
- 构建时运行自定义命令：I. 使用add_custom_command
- 构建时运行自定义命令：II. 使用add_custom_target
- 构建时为特定目标运行自定义命令
- 探究编译和链接命令
- 探究编译器标志命令
- 探究可执行命令
- 使用生成器表达式微调配置和编译

我们将学习如何在配置和构建时，执行自定义操作。先简单回顾一下，与CMake工作流程相关的时序：

1. **CMake**时或构建时：CMake正在运行，并处理项目中的 `CMakeLists.txt` 文件。
2. 生成时：生成构建工具(如Makefile或Visual Studio项目文件)。
3. 构建时：由CMake生成相应平台的原生构建脚本，在脚本中调用原生工具构建。此时，将调用编译器在特定的构建目录中构建目标(可执行文件和库)。
4. **CTest**时或测试时：运行测试套件以检查目标是否按预期执行。
5. **CDash**时或报告时：当测试结果上传到仪表板上，与其他开发人员共享测试报告。
6. 安装时：当目标、源文件、可执行程序和库，从构建目录安装到相应位置。
7. **CPack**时或打包时：将项目打包用以分发时，可以是源码，也可以是二进制。
8. 包安装时：新生成的包在系统范围内安装。

完整的工作流程和对应的时序，如下图所示：



本章会介绍在配置和构建时的自定义行为，我们将学习如何使用这些命令：

- `execute_process`, 从CMake中执行任意进程，并检索它们的输出。
- `add_custom_target`, 创建执行自定义命令的目标。
- `add_custom_command`, 指定必须执行的命令，以生成文件或在其他目标的特定生成事件中生成。

5.1 使用平台无关的文件操作

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-01> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

有些项目构建时，可能需要与平台的文件系统进行交互。也就是检查文件是否存在、创建新文件来存储临时信息、创建或提取打包文件等等。使用CMake不仅能够在不同的平台上生成构建系统，还能够在不复杂的逻辑情况下，进行文件操作，从而独立于操作系统。本示例将展示，如何以可移植的方式下载库文件。

准备工作

我们将展示如何提取Eigen库文件，并使用提取的源文件编译我们的项目。这个示例中，将重用第3章第7节的线性代数例子 `linear-algebra.cpp`，用来检测外部库和程序、检测特征库。这里，假设已经包含Eigen库文件，已在项目构建前下载。

具体实施

项目需要解压缩Eigen打包文件，并相应地为目标设置包含目录：

1. 首先，使能C++11项目：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-01 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 我们将自定义目标添加到构建系统中，自定义目标将提取构建目录中的库文件：

```
1. add_custom_target(unpack-eigen
2.   ALL
3.   COMMAND
4.     ${CMAKE_COMMAND} -E tar xzf ${CMAKE_CURRENT_SOURCE_DIR}/eigen-eigen-
5.   5a0156e40feb.tar.gz
6.   COMMAND
7.     ${CMAKE_COMMAND} -E rename eigen-eigen-5a0156e40feb eigen-3.3.4
```

```

7.      WORKING_DIRECTORY
8.      ${CMAKE_CURRENT_BINARY_DIR}
9.      COMMENT
10.     "Unpacking Eigen3 in ${CMAKE_CURRENT_BINARY_DIR}/eigen-3.3.4"
11.   )

```

3. 为源文件添加了一个可执行目标：

```
1. add_executable(linear-algebra linear-algebra.cpp)
```

4. 由于源文件的编译依赖于Eigen头文件，需要显式地指定可执行目标对自定义目标的依赖关系：

```
1. add_dependencies(linear-algebra unpack-eigen)
```

5. 最后，指定包含哪些目录：

```

1. target_include_directories(linear-algebra
2.   PRIVATE
3.   ${CMAKE_CURRENT_BINARY_DIR}/eigen-3.3.4
4. )

```

工作原理

细看 `add_custom_target` 这个命令：

```

1. add_custom_target(unpack-eigen
2.   ALL
3.   COMMAND
4.     ${CMAKE_COMMAND} -E tar xzf ${CMAKE_CURRENT_SOURCE_DIR}/eigen-eigen-
5. 5a0156e40feb.tar.gz
6.   COMMAND
7.     ${CMAKE_COMMAND} -E rename eigen-eigen-5a0156e40feb eigen-3.3.4
8.   WORKING_DIRECTORY
9.     ${CMAKE_CURRENT_BINARY_DIR}
10.    COMMENT
11.   "Unpacking Eigen3 in ${CMAKE_CURRENT_BINARY_DIR}/eigen-3.3.4"
12. )

```

构建系统中引入了一个名为 `unpack-eigen` 的目标。因为我们传递了 `ALL` 参数，目标将始终被执行。`COMMAND` 参数指定要执行哪些命令。本例中，我们希望提取存档并将提取的目录重命名

为 `eigen -3.3.4`，通过以下两个命令实现：

```
1. ${CMAKE_COMMAND} -E tar xzf ${CMAKE_CURRENT_SOURCE_DIR}/eigen-eigen-
2. 5a0156e40feb.tar.gz
3. ${CMAKE_COMMAND} -E rename eigen-eigen-5a0156e40feb eigen-3.3.4
```

注意，使用 `-E` 标志调用CMake命令本身来执行实际的工作。对于许多常见操作，CMake实现了一个对所有操作系统都通用的接口，这使得构建系统独立于特定的平台。`add_custom_target` 命令中的下一个参数是工作目录。我们的示例中，它对应于构建目录：`CMAKE_CURRENT_BINARY_DIR`。最后一个参数 `COMMENT`，用于指定CMake在执行自定义目标时输出什么样的消息。

更多信息

构建过程中必须执行一系列没有输出的命令时，可以使用 `add_custom_target` 命令。正如我们在本示例中所示，可以将自定义目标指定为项目中其他目标的依赖项。此外，自定义目标还可以依赖于其他目标。

使用 `-E` 标志可以以与操作系统无关的方式，运行许多公共操作。运行 `cmake -E` 或 `cmake -E help` 可以获得特定操作系统的完整列表。例如，这是Linux系统上命令的摘要：

```
1. Usage: cmake -E <command> [arguments...]
2. Available commands:
   capabilities           - Report capabilities built into cmake in JSON
   format
   chdir dir cmd [args...] - run command in a given directory
   compare_files file1 file2 - check if file1 is same as file2
   copy <file>... destination - copy files to destination (either file or
   directory)
   copy_directory <dir>... destination - copy content of <dir>... directories
   to 'destination' directory
   copy_if_different <file>... destination - copy files if it has changed
   echo [<string>...]       - displays arguments as text
   echo_append [<string>...] - displays arguments as text but no new line
   env [--unset=NAME]... [NAME=VALUE]... COMMAND [ARG]...
   - run command in a modified environment
   environment            - display the current environment
   make_directory <dir>... - create parent and <dir> directories
   md5sum <file>...        - create MD5 checksum of files
   sha1sum <file>...        - create SHA1 checksum of files
   sha224sum <file>...       - create SHA224 checksum of files
   sha256sum <file>...       - create SHA256 checksum of files
   sha384sum <file>...       - create SHA384 checksum of files
```

20.	sha512sum <file>...	- create SHA512 checksum of files
21.	remove [-f] <file>...	- remove the file(s), use -f to force it
22.	remove_directory dir	- remove a directory and its contents
23.	rename oldname newname	- rename a file or directory (on one volume)
24.	server	- start cmake in server mode
25.	sleep <number>...	- sleep for given number of seconds
26.	tar [cxt][vf][zjJ] file.tar [file/dir1 file/dir2 ...]	- create or extract a tar or zip archive
27.	time command [args...]	- run command and display elapsed time
29.	touch file	- touch a file.
30.	touch_nocreate file	- touch a file but do not create it.
31.	Available on UNIX only:	
32.	create_symlink old new	- create a symbolic link new -> old

5.2 配置时运行自定义命令

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-05/recipe-02> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

运行CMake生成构建系统，从而指定原生构建工具必须执行哪些命令，以及按照什么顺序执行。我们已经了解了CMake如何在配置时运行许多子任务，以便找到工作的编译器和必要的依赖项。本示例中，我们将讨论如何使用 `execute_process` 命令在配置时运行定制化命令。

具体实施

第3章第3节中，我们已经展示了 `execute_process` 查找Python模块NumPy时的用法。本例中，我们将使用 `execute_process` 命令来确定，是否存在特定的Python模块(本例中为Python CFFI)，如果存在，我们在进行版本确定：

- 对于这个简单的例子，不需要语言支持：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-02 LANGUAGES NONE)
```

- 我们要求Python解释器执行一个简短的代码片段，因此，需要使用 `find_package` 来查找解释器：

```
1. find_package(PythonInterp REQUIRED)
```

- 然后，调用 `execute_process` 来运行一个简短的Python代码段；下一节中，我们将更详细地讨论这个命令：

```
1. # this is set as variable to prepare
2. # for abstraction using loops or functions
3. set(_module_name "cffi")
4.
5. execute_process(
6.   COMMAND
7.     ${PYTHON_EXECUTABLE} "-c" "import ${_module_name};"
8.     print(${_module_name}.__version__)
9.   OUTPUT_VARIABLE _stdout
10.    ERROR_VARIABLE _stderr
11.    OUTPUT_STRIP_TRAILING_WHITESPACE
```

```

11.    ERROR_STRIP_TRAILING_WHITESPACE
12.  )

```

4. 然后，打印结果：

```

1. if(_stderr MATCHES "ModuleNotFoundError")
2.   message(STATUS "Module ${_module_name} not found")
3. else()
4.   message(STATUS "Found module ${_module_name} v${_stdout}")
5. endif()

```

5. 下面是一个配置示例(假设Python CFFI包安装在相应的Python环境中)：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
-- Found PythonInterp: /home/user/cmake-cookbook/chapter-05/recipe-
5. 02/example/venv/bin/python (found version "3.6.5")
6. -- Found module cffi v1.11.5

```

工作原理

`execute_process` 命令将从当前正在执行的CMake进程中派生一个或多个子进程，从而提供了在配置项目时运行任意命令的方法。可以在一次调用 `execute_process` 时执行多个命令。但请注意，每个命令的输出将通过管道传输到下一个命令中。该命令接受多个参数：

- `WORKING_DIRECTORY`，指定应该在哪个目录中执行命令。
- `RESULT_VARIABLE`将包含进程运行的结果。这要么是一个整数，表示执行成功，要么是一个带有错误条件的字符串。
- `OUTPUT_VARIABLE`和`ERROR_VARIABLE`将包含执行命令的标准输出和标准错误。由于命令的输出是通过管道传输的，因此只有最后一个命令的标准输出才会保存到`OUTPUT_VARIABLE`中。
- `INPUT_FILE`指定标准输入重定向的文件名
- `OUTPUT_FILE`指定标准输出重定向的文件名
- `ERROR_FILE`指定标准错误输出重定向的文件名
- 设置`OUTPUT_QUIET`和`ERROR_QUIET`后，CMake将静默地忽略标准输出和标准错误。
- 设置`OUTPUT_STRIP_TRAILING_WHITESPACE`，可以删除运行命令的标准输出中的任何尾随空格
- 设置`ERROR_STRIP_TRAILING_WHITESPACE`，可以删除运行命令的错误输出中的任何尾随空格。

有了这些了解这些参数，回到我们的例子当中：

```

1. set(_module_name "cffi")
2.
3. execute_process(
4.   COMMAND
5.     ${PYTHON_EXECUTABLE} "-c" "import ${_module_name};"
6.     print(${_module_name}.__version__)
7.     OUTPUT_VARIABLE _stdout
8.     ERROR_VARIABLE _stderr
9.     OUTPUT_STRIP_TRAILING_WHITESPACE
10.    ERROR_STRIP_TRAILING_WHITESPACE
11. )
12. if(_stderr MATCHES "ModuleNotFoundError")
13.   message(STATUS "Module ${_module_name} not found")
14. else()
15.   message(STATUS "Found module ${_module_name} v${_stdout}")
endif()
```

该命令检查 `python -c "import cffi; print(cffi.__version__)"` 的输出。如果没有找到模块，`_stderr` 将包含 `ModuleNotFoundError`，我们将在if语句中对其进行检查。本例中，我们将打印 `Module cffi not found`。如果导入成功，Python代码将打印模块的版本，该模块通过管道输入 `_stdout`，这样就可以打印如下内容：

```
1. message(STATUS "Found module ${_module_name} v${_stdout}")
```

更多信息

本例中，只打印了结果，但实际项目中，可以警告、中止配置，或者设置可以查询的变量，来切换某些配置选项。

代码示例会扩展到多个Python模块(如Cython)，以避免代码重复。一种选择是使用 `foreach` 循环模块名，另一种方法是将代码封装为函数或宏。我们将在第7章中讨论这些封装。

第9章中，我们将使用Python CFFI和Cython。现在的示例，可以作为有用的、可重用的代码片段，来检测这些包是否存在。

5.3 构建时运行自定义命令：I . 使用add_custom_command

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-03> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

项目的构建目标取决于命令的结果，这些命令只能在构建系统生成完成后的构建执行。CMake提供了三个选项来在构建时执行自定义命令：

1. 使用 `add_custom_command` 编译目标，生成输出文件。
2. `add_custom_target` 的执行没有输出。
3. 构建目标前后，`add_custom_command` 的执行可以没有输出。

这三个选项强制执行特定的语义，并且不可互换。接下来的三个示例将演示具体的用法。

准备工作

我们将重用第3章第4节中的C++示例，以说明如何使用 `add_custom_command` 的第一个选项。代码示例中，我们了解了现有的BLAS和LAPACK库，并编译了一个很小的C++包装器库，以调用线性代数的Fortran实现。

我们将把代码分成两部分。`linear-algebra.cpp` 的源文件与第3章、第4章没有区别，并且将包含线性代数包装器库的头文件和针对编译库的链接。源代码将打包到一个压缩的tar存档文件中，该存档文件随示例项目一起提供。存档文件将在构建时提取，并在可执行文件生成之前，编译线性代数的包装器库。

具体实施

`CMakeLists.txt` 必须包含一个自定义命令，来提取线性代数包装器库的源代码：

1. 从CMake最低版本、项目名称和支持语言的定义开始：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES CXX Fortran)
```

2. 选择C++11标准：

```
1. set(CMAKE_CXX_STANDARD 11)
2. set(CMAKE_CXX_EXTENSIONS OFF)
```

```
3. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

3. 然后，在系统上查找BLAS和LAPACK库：

```
1. find_package(BLAS REQUIRED)
2. find_package(LAPACK REQUIRED)
```

4. 声明一个变量 `wrap_BLAS_LAPACK_sources` 来保存 `wrap_BLAS_LAPACK.tar.gz` 压缩包文件的名称：

```
1. set(wrap_BLAS_LAPACK_sources
2. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.hpp
3. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.cpp
4. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.hpp
5. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.cpp
6. )
```

5. 声明自定义命令来提取 `wrap_BLAS_LAPACK.tar.gz` 压缩包，并更新提取文件的时间戳。注意这个 `wrap_BLAS_LAPACK_sources` 变量的预期输出：

```
1. add_custom_command(
2.   OUTPUT
3.     ${wrap_BLAS_LAPACK_sources}
4.   COMMAND
5.     ${CMAKE_COMMAND} -E tar xzf
6.   ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
7.   COMMAND
8.     ${CMAKE_COMMAND} -E touch ${wrap_BLAS_LAPACK_sources}
9.   WORKING_DIRECTORY
10.    ${CMAKE_CURRENT_BINARY_DIR}
11.    DEPENDS
12.      ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
13.    COMMENT
14.      "Unpacking C++ wrappers for BLAS/LAPACK"
15.    VERBATIM
16.  )
```

6. 接下来，添加一个库目标，源文件是新解压出来的：

```
1. add_library(math "")
2.
```

```

3. target_sources(math
4.   PRIVATE
5.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.cpp
6.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.cpp
7.   PUBLIC
8.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.hpp
9.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.hpp
10. )
11.
12. target_include_directories(math
13.   INTERFACE
14.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK
15. )
16.
17. target_link_libraries(math
18.   PUBLIC
19.     ${LAPACK_LIBRARIES}
20. )

```

7. 最后，添加 `linear-algebra` 可执行目标。可执行目标链接到库：

```

1. add_executable(linear-algebra linear-algebra.cpp)
2.
3. target_link_libraries(linear-algebra
4.   PRIVATE
5.     math
6. )

```

8. 我们配置、构建和执行示例：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./linear-algebra 1000
6.
7. C_DSCAL done
8. C_DGESV done
9. info is 0
10. check is 4.35597e-10

```

工作原理

让我们来了解一下 `add_custom_command` 的使用：

```

1. add_custom_command(
2.   OUTPUT
3.     ${wrap_BLAS_LAPACK_sources}
4.   COMMAND
5.     ${CMAKE_COMMAND} -E tar xzf
6.   ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
7.   COMMAND
8.     ${CMAKE_COMMAND} -E touch ${wrap_BLAS_LAPACK_sources}
9.   WORKING_DIRECTORY
10.    ${CMAKE_CURRENT_BINARY_DIR}
11.  DEPENDS
12.    ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
13.  COMMENT
14.    "Unpacking C++ wrappers for BLAS/LAPACK"
15.  VERBATIM
16. )

```

`add_custom_command` 向目标添加规则，并通过执行命令生成输出。`add_custom_command` 中声明的任何目标，即在相同的 `CMakeLists.txt` 中声明的任何目标，使用输出的任何文件作为源文件的目标，在构建时会有规则生成这些文件。因此，源文件生成在构建时，目标和自定义命令在构建系统生成时，将自动处理依赖关系。

我们的例子中，输出是压缩 `tar` 包，其中包含有源文件。要检测和使用这些文件，必须在构建时提取打包文件。通过使用带有 `-E` 标志的CMake命令，以实现平台独立性。下一个命令会更新提取文件的时间戳。这样做是为了确保没有处理陈旧文件。`WORKING_DIRECTORY` 可以指定在何处执行命令。示例中，`CMAKE_CURRENT_BINARY_DIR` 是当前正在处理的构建目录。`DEPENDS` 参数列出了自定义命令的依赖项。例子中，压缩的 `tar` 是一个依赖项。CMake使用 `COMMENT` 字段在构建时打印状态消息。最后，`VERBATIM` 告诉CMake为生成器和平台生成正确的命令，从而确保完全独立。

我们来仔细看看这用使用方式和打包库的创建：

```

1. add_library(math "")
2.
3. target_sources(math
4.   PRIVATE
5.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.cpp
6.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.cpp
7.   PUBLIC

```

```

8.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.hpp
9.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.hpp
10.    )
11.
12. target_include_directories(math
13.   INTERFACE
14.     ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK
15.   )
16.
17. target_link_libraries(math
18.   PUBLIC
19.   ${LAPACK_LIBRARIES}
20.   )

```

我们声明一个没有源的库目标，是因为后续使用 `target_sources` 填充目标的源。这里实现了一个非常重要的目标，即让依赖于此目标的目标，了解需要哪些目录和头文件，以便成功地使用库。C++源文件的目标是 `PRIVATE`，因此只用于构建库。因为目标及其依赖项都需要使用它们来成功编译，所以头文件是 `PUBLIC`。包含目录使用 `target_include_categories` 指定，其中 `wrap_BLAS_LAPACK` 声明为 `INTERFACE`，因为只有依赖于 `math` 目标的目标需要它。

`add_custom_command` 有两个限制：

- 只有在相同的 `CMakeLists.txt` 中，指定了所有依赖于其输出的目标时才有效。
- 对于不同的独立目标，使用 `add_custom_command` 的输出可以重新执行定制命令。这可能会导致冲突，应该避免这种情况的发生。

第二个限制，可以使用 `add_dependencies` 来避免。不过，规避这两个限制的正确方法是使用 `add_custom_target` 命令，我们将在下一节的示例中详细介绍。

5.4 构建时运行自定义命令:II. 使用add_custom_target

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-04> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

我们在前面的示例，讨论了 `add_custom_command` 有一些限制，可以通过 `add_custom_target` 绕过这些限制。这个CMake命令将引入新的目标，与 `add_custom_command` 相反，这些目标依次执行不返回输出。可以将 `add_custom_target` 和 `add_custom_command` 结合使用。使用这种方法，可以与其依赖项所在目录不同的目录指定自定义目标，CMake基础设施对项目设计模块化非常有用。

准备工作

我们将重用前一节示例，对源码进行简单的修改。特别是，将把压缩后的 `tar` 打包文件放在名为 `deps` 的子目录中，而不是存储在主目录中。这个子目录包含它自己的 `CMakeLists.txt`，将由主 `CMakeLists.txt` 调用。

具体实施

我们将从主 `CMakeLists.txt` 开始，然后讨论 `deps/CMakeLists.txt`：

1. 声明启用C++11：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-04 LANGUAGES CXX Fortran)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 现在，继续讨论 `deps/CMakeLists.txt`。这通过 `add_subdirectory` 命令实现：

1. `add_subdirectory(deps)`

3. `deps/CMakeLists.txt` 中，我们首先定位必要的库(BLAS和LAPACK)：

1. `find_package(BLAS REQUIRED)`

```
2. find_package(LAPACK REQUIRED)
```

4. 然后, 我们将 `tar` 包的内容汇集到一个变量 `MATH_SRCS` 中:

```
1. set(MATH_SRCS
2. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.cpp
3. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.cpp
4. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.hpp
5. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.hpp
6. )
```

5. 列出要打包的源之后, 定义一个目标和一个命令。这个组合用于提取 `${CMAKE_CURRENT_BINARY_DIR}` 中的包。但是, 这里我们在一个不同的范围内, 引用 `deps/CMakeLists.txt`, 因此 `tar` 包将存放在到主项目构建目录下的 `deps` 子目录中:

```
1. add_custom_target(BLAS_LAPACK_wrappers
2. WORKING_DIRECTORY
3. ${CMAKE_CURRENT_BINARY_DIR}
4. DEPENDS
5. ${MATH_SRCS}
6. COMMENT
7. "Intermediate BLAS_LAPACK_wrappers target"
8. VERBATIM
9. )
10.
11. add_custom_command(
12. OUTPUT
13. ${MATH_SRCS}
14. COMMAND
15. ${CMAKE_COMMAND} -E tar xzf
16. ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
17. WORKING_DIRECTORY
18. ${CMAKE_CURRENT_BINARY_DIR}
19. DEPENDS
20. ${CMAKE_CURRENT_SOURCE_DIR}/wrap_BLAS_LAPACK.tar.gz
21. COMMENT
22. "Unpacking C++ wrappers for BLAS/LAPACK"
23. )
```

6. 添加数学库作为目标, 并指定相应的源, 包括目录和链接库:

```
1. add_library(math "")
```

```

2.
3. target_sources(math
4.   PRIVATE
5.     ${MATH_SRCS}
6.   )
7.
8. target_include_directories(math
9.   INTERFACE
10.    ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK
11.  )
12.
13. # BLAS_LIBRARIES are included in LAPACK_LIBRARIES
14. target_link_libraries(math
15.   PUBLIC
16.     ${LAPACK_LIBRARIES}
17.  )

```

7. 执行完 `deps/CMakeLists.txt` 中的命令，返回到父范围，定义可执行目标，并将其链接到另一个目录的数学库：

```

1. add_executable(linear-algebra linear-algebra.cpp)
2.
3. target_link_libraries(linear-algebra
4.   PRIVATE
5.     math
6.   )

```

工作原理

用户可以使用 `add_custom_target`，在目标中执行定制命令。这与我们前面讨论的 `add_custom_command` 略有不同。`add_custom_target` 添加的目标没有输出，因此总会执行。因此，可以在子目录中引入自定义目标，并且仍然能够在主 `CMakeLists.txt` 中引用它。

本例中，使用 `add_custom_target` 和 `add_custom_command` 提取了源文件的包。这些源文件稍后用于编译另一个库，我们设法在另一个(父)目录范围内链接这个库。构建 `CMakeLists.txt` 文件的过程中，`tar` 包是在 `deps` 下，`deps` 是项目构建目录下的一个子目录。这是因为在CMake中，构建树的结构与源树的层次结构相同。

这个示例中有一个值得注意的细节，就是我们把数学库的源标记为 `PRIVATE`：

```
1. set(MATH_SRCS
```

```

2. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.cpp
3. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.cpp
4. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxBLAS.hpp
5. ${CMAKE_CURRENT_BINARY_DIR}/wrap_BLAS_LAPACK/CxxLAPACK.hpp
6. )
7.
8. # ...
9.
10. add_library(math "")
11.
12. target_sources(math
13.   PRIVATE
14.   ${MATH_SRCS}
15. )
16.
17. # ...

```

虽然这些源代码是 `PRIVATE`，但我们在父范围内编译了 `linear-algebra.cpp`，并且这个源代码包括 `CxxBLAS.hpp` 和 `CxxLAPACK.hpp`。为什么这里使用 `PRIVATE`，以及如何编译 `linear-algebra.cpp`，并构建可执行文件呢？如果将头文件标记为 `PUBLIC`，CMake就会在创建时停止，并出现一个错误，“无法找到源文件”，因为要生成(提取)还不存在于文件树中的源文件。

这是一个已知的限制(参见<https://gitlab.kitware.com/cmake/cmake/issues/1633>，以及相关的博客文章:<https://samthursfield.wordpress.com/2015/11/21/cmake-dependencies-targets-and-files-and-custom-commands>)。我们通过声明源代码为 `PRIVATE` 来解决这个限制。这样CMake时，没有获得对不存在源文件的依赖。但是，CMake内置的C/C++文件依赖关系扫描器在构建时获取它们，并编译和链接源代码。

5.5 构建时为特定目标运行自定义命令

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-05> 中找到，其中包含一个Fortran例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本节示例将展示，如何使用 `add_custom_command` 的第二个参数，来执行没有输出的自定义操作，这对于构建或链接特定目标之前或之后执行某些操作非常有用。由于自定义命令仅在必须构建目标本身时才执行，因此我们实现了对其执行的目标级控制。我们将通过一个示例来演示，在构建目标之前打印目标的链接，然后在编译后，立即测量编译后，可执行文件的静态分配大小。

准备工作

本示例中，我们将使用Fortran代码(`example.f90`)：

```

1. program example
2.
3.   implicit none
4.
5.   real(8) :: array(20000000)
6.   real(8) :: r
7.   integer :: i
8.
9.   do i = 1, size(array)
10.    call random_number(r)
11.    array(i) = r
12. end do
13.
14. print *, sum(array)
15.
16. end program

```

虽然我们选择了Fortran，但Fortran代码的对于后面的讨论并不重要，因为有很多遗留的Fortran代码，存在静态分配大小的问题。

这段代码中，我们定义了一个包含20,000,000双精度浮点数的数组，这个数组占用160MB的内存。在这里，我们并不是推荐这样的编程实践。一般来说，这些内存的分配和代码中是否使用这段内存无关。一个更好的方法是只在需要时动态分配数组，随后立即释放。

示例代码用随机数填充数组，并计算它们的和—这样是为了确保数组确实被使用，并且编译器不会优化

分配。我们将使用Python脚本(`static-size.py`)来统计二进制文件静态分配的大小，该脚本用size命令来封装：

```

1. import subprocess
2. import sys
3.
4. # for simplicity we do not check number of
5. # arguments and whether the file really exists
6. file_path = sys.argv[-1]
7. try:
8.     output = subprocess.check_output(['size', file_path]).decode('utf-8')
9. except FileNotFoundError:
10.     print('command "size" is not available on this platform')
11.     sys.exit(0)
12.
13. size = 0.0
14. for line in output.split('\n'):
15.     if file_path in line:
16.         # we are interested in the 4th number on this line
17.         size = int(line.split()[3])
18.
19. print('{0:.3f} MB'.format(size/1.0e6))

```

要打印链接行，我们将使用第二个Python helper脚本(`echo-file.py`)打印文件的内容：

```

1. import sys
2.
3. # for simplicity we do not verify the number and
4. # type of arguments
5. file_path = sys.argv[-1]
6. try:
7.     with open(file_path, 'r') as f:
8.         print(f.read())
9. except FileNotFoundError:
10.     print('ERROR: file {} not found'.format(file_path))

```

具体实施

来看看 `CMakeLists.txt`：

- 首先声明一个Fortran项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-05 LANGUAGES Fortran)

```

2. 例子依赖于Python解释器，所以以一种可移植的方式执行helper脚本：

```
1. find_package(PythonInterp REQUIRED)
```

3. 本例中，默認為“Release”构建类型，以便CMake添加优化标志：

```

1. if(NOT CMAKE_BUILD_TYPE)
2.   set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
3. endif()

```

4. 现在，定义可执行目标：

```

1. add_executable(example "")
2.
3. target_sources(example
4.   PRIVATE
5.   example.f90
6. )

```

5. 然后，定义一个自定义命令，在 `example` 目标在已链接之前，打印链接行：

```

1. add_custom_command(
2.   TARGET
3.   example
4.   PRE_LINK
5.   COMMAND
6.   ${PYTHON_EXECUTABLE}
7.   ${CMAKE_CURRENT_SOURCE_DIR}/echo-file.py
8.   ${CMAKE_CURRENT_BINARY_DIR}/CMakeFiles/example.dir/link.txt
9.   COMMENT
10.  "link line:"
11.  VERBATIM
12. )

```

6. 测试一下。观察打印的链接行和可执行文件的静态大小：

```

1. $ mkdir -p build
2. $ cd build

```

```

3. $ cmake ..
4. $ cmake --build .
5.
6. Scanning dependencies of target example
7. [ 50%] Building Fortran object CMakeFiles/example.dir/example.f90.o
8. [100%] Linking Fortran executable example
9. link line:
/usr/bin/f95 -O3 -DNDEBUG -O3 CMakeFiles/example.dir/example.f90.o -o
10. example
11. static size of executable:
12. 160.003 MB
13. [100%] Built target example

```

工作原理

当声明了库或可执行目标，就可以使用 `add_custom_command` 将其他命令锁定到目标上。这些命令将在特定的时间执行，与它们所附加的目标的执行相关联。CMake通过以下选项，定制命令执行顺序：

- **PRE_BUILD**: 在执行与目标相关的任何其他规则之前执行的命令。
- **PRE_LINK**: 使用此选项，命令在编译目标之后，调用链接器或归档器之前执行。Visual Studio 7或更高版本之外的生成器中使用 `PRE_BUILD` 将被解释为 `PRE_LINK`。
- **POST_BUILD**: 如前所述，这些命令将在执行给定目标的所有规则之后运行。

本例中，将两个自定义命令绑定到可执行目标。`PRE_LINK` 命令

将 `${CMAKE_CURRENT_BINARY_DIR}/CMakeFiles/example.dir/link.txt` 的内容打印到屏幕上。在我们的例子中，链接行是这样的：

```

1. link line:
2. /usr/bin/f95 -O3 -DNDEBUG -O3 CMakeFiles/example.dir/example.f90.o -o example

```

使用Python包装器来实现这一点，它依赖于shell命令。

第二步中，`POST_BUILD` 自定义命令调用Python helper脚本 `static-size.py`，生成器表达式 `${<target_file:example>}` 作为参数。CMake将在生成时(即生成生成系统时)将生成器表达式扩展到目标文件路径。然后，Python脚本 `static-size.py` 使用size命令获取可执行文件的静态分配大小，将其转换为MB，并打印结果。我们的例子中，获得了预期的160 MB：

```

1. static size of executable:
2. 160.003 MB

```

5.6 探究编译和链接命令

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-06> 中找到，其中包含一个C++例子。该示例在CMake 3.9版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。代码库还有一个与CMake 3.5兼容的示例。

生成构建系统期间最常见的操作，是试图评估在哪种系统上构建项目。这意味着要找出哪些功能工作，哪些不工作，并相应地调整项目的编译。使用的方法是查询依赖项是否被满足的信号，或者在代码库中是否启用工作区。接下来的几个示例，将展示如何使用CMake执行这些操作。我们将特别讨论以下事宜：

1. 如何确保代码能成功编译为可执行文件。
2. 如何确保编译器理解相应的标志。
3. 如何确保特定代码能成功编译为运行可执行程序。

准备工作

示例将展示如何使用来自对应的 `Check<LANG>SourceCompiles.cmake` 标准模块的 `check_<lang>_source_compiles` 函数，以评估给定编译器是否可以将预定义的代码编译成可执行文件。该命令可帮助你确定：

- 编译器支持所需的特性。
- 链接器工作正常，并理解特定的标志。
- 可以使用 `find_package` 找到的包含目录和库。

本示例中，我们将展示如何检测OpenMP 4.5标准的循环特性，以便在C++可执行文件中使用。使用一个C++源文件，来探测编译器是否支持这样的特性。CMake提供了一个附加命令 `try_compile` 来探究编译。本示例将展示，如何使用这两种方法。

TIPS: 可以使用CMake命令行界面来获取关于特定模块(`cmake --help-module <module-name>`)和命令(`cmake --help-command <command-name>`)的文档。示例中，`cmake --help-module checkCXXSourceCompiles` 将把 `check_cxx_source_compiles` 函数的文档输出到屏幕上，而 `cmake --help-command try_compile` 将对 `try_compile` 命令执行相同的操作。

具体实施

我们将同时使用 `try_compile` 和 `check_cxx_source_compiles`，并比较这两个命令的工作方式：

1. 创建一个C++11工程：

```

1. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
2. project(recipe-06 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 查找编译器支持的OpenMP：

```

1. find_package(OpenMP)
2.
3. if(OpenMP_FOUND)
4.     # ... <- the steps below will be placed here
5. else()
6.     message(STATUS "OpenMP not found: no test for taskloop is run")
7. endif()

```

3. 如果找到OpenMP，再检查所需的特性是否可用。为此，设置了一个临时目录，`try_compile` 将在这个目录下来生成中间文件。我们把它放在前面步骤中引入的 `if` 语句中：

```
1. set(_scratch_dir ${CMAKE_CURRENT_BINARY_DIR}/omp_try_compile)
```

4. 调用 `try_compile` 生成一个小项目，以尝试编译源文件 `taskloop.cpp`。编译成功或失败的状态，将保存到 `omp_taskloop_test_1` 变量中。需要为这个示例编译设置适当的编译器标志、包括目录和链接库。因为使用导入的目标 `OpenMP::OpenMP_CXX`，所以只需将 `LINK_LIBRARIES` 选项设置为 `try_compile` 即可。如果编译成功，则任务循环特性可用，我们为用户打印一条消息：

```

1. try_compile(
2.   omp_taskloop_test_1
3.   ${_scratch_dir}
4.   SOURCES
5.   ${CMAKE_CURRENT_SOURCE_DIR}/taskloop.cpp
6.   LINK_LIBRARIES
7.   OpenMP::OpenMP_CXX
8. )
9. message(STATUS "Result of try_compile: ${omp_taskloop_test_1}")

```

5. 要使用 `check_cxx_source_compiles` 函数，需要包含 `CheckCXXSourceCompiles.cmake` 模块文件。其他语言也有类似的模块文件，C(`CheckCSourceCompiles.cmake`) 和

```
Fortran( CheckFortranSourceCompiles.cmake ):
```

```
1. include(CheckCXXSourceCompiles)
```

6. 我们复制源文件的内容，通过 `file(READ ...)` 命令读取内容到一个变量中，试图编译和连接这个变量：

```
1. file(READ ${CMAKE_CURRENT_SOURCE_DIR}/taskloop.cpp _snippet)
```

7. 我们设置了 `CMAKE_REQUIRED_LIBRARIES`。这对于下一步正确调用编译器是必需的。注意使用导入的 `OpenMP::OpenMP_CXX` 目标，它还将设置正确的编译器标志和包含目录：

```
1. set(CMAKE_REQUIRED_LIBRARIES OpenMP::OpenMP_CXX)
```

8. 使用代码片段作为参数，调用 `check_cxx_source_compiles` 函数。检查结果将保存到 `omp_taskloop_test_2` 变量中：

```
1. check_cxx_source_compiles("${_snippet}" omp_taskloop_test_2)
```

9. 调用 `check_cxx_source_compiles` 并向用户打印消息之前，我们取消了变量的设置：

```
1. unset(CMAKE_REQUIRED_LIBRARIES)
message(STATUS "Result of check_cxx_source_compiles:
2. ${omp_taskloop_test_2}"
```

10. 最后，进行测试：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. --
6. -- Found OpenMP_CXX: -fopenmp (found version "4.5")
7. -- Found OpenMP: TRUE (found version "4.5")
8. -- Result of try_compile: TRUE
9. -- Performing Test omp_taskloop_test_2
10. -- Performing Test omp_taskloop_test_2 - Success
11. -- Result of check_cxx_source_compiles: 1
```

工作原理

`try_compile` 和 `check_cxx_source_compiles` 都将编译源文件，并将其链接到可执行文件中。如果这些操作成功，那么输出变量 `omp_task_loop_test_1` (前者) 和 `omp_task_loop_test_2` (后者) 将被设置为 `TRUE`。然而，这两个命令实现的方式略有不同。`check_<lang>_source_compiles` 命令是 `try_compile` 命令的简化包装。因此，它提供了一个接口：

1. 要编译的代码片段必须作为CMake变量传入。大多数情况下，这意味着必须使用 `file(READ ...)` 来读取文件。然后，代码片段被保存到构建目录的 `CMakeFiles/CMakeTmp` 子目录中。
2. 微调编译和链接，必须通过设置以下CMake变量进行：
 - `CMAKE_REQUIRED_FLAGS`: 设置编译器标志。
 - `CMAKE_REQUIRED_DEFINITIONS`: 设置预编译宏。
 - `CMAKE_REQUIRED_INCLUDES`: 设置包含目录列表。
 - `CMAKE_REQUIRED_LIBRARIES`: 设置可执行目标能够连接的库列表。
3. 调用 `check_<lang>_compiles_function` 之后，必须手动取消对这些变量的设置，以确保后续使用中，不会保留当前内容。

NOTE: 使用CMake 3.9中可以对于OpenMP目标进行导入，但是目前的配置也可以使用CMake的早期版本，通过手动为 `check_cxx_source_compiles` 设置所需的标志和库：`set(CMAKE_REQUIRED_FLAGS ${OpenMP_CXX_FLAGS})` 和 `set(CMAKE_REQUIRED_LIBRARIES ${OpenMP_CXX_LIBRARIES})`。

TIPS: Fortran下，CMake代码的格式通常是固定的，但也有意外情况。为了处理这些意外，需要为 `check_fortran_source_compiles` 设置 `-ffree-form` 编译标志。可以通过 `set(CMAKE_REQUIRED_FLAGS "-ffree-form")` 实现。

这个接口反映了：测试编译是通过，在CMake调用中直接生成和执行构建和连接命令来执行的。

命令 `try_compile` 提供了更完整的接口和两种不同的操作模式：

1. 以一个完整的CMake项目作为输入，并基于它的 `CMakeLists.txt` 配置、构建和链接。这种操作模式提供了更好的灵活性，因为要编译项目的复杂度是可以选择的。
2. 提供了源文件，和用于包含目录、链接库和编译器标志的配置选项。

因此，`try_compile` 基于在项目上调用CMake，其中 `CMakeLists.txt` 已经存在(在第一种操作模式中)，或者基于传递给 `try_compile` 的参数动态生成文件。

更多信息

本示例中概述的类型检查并不总是万无一失的，并且可能产生假阳性和假阴性。作为一个例子，可以尝试注释掉包含 `CMAKE_REQUIRED_LIBRARIES` 的行。运行这个例子仍然会报告“成功”，这是因为编译器将忽略OpenMP的 `pragma` 字段。

当返回了错误的结果时，应该怎么做？构建目录的 `CMakeFiles` 子目录中

的 `CMakeOutput.log` 和 `CMakeError.log` 文件会提供一些线索。它们记录了CMake运行的操作的标准输出和标准错误。如果怀疑结果有误，应该通过搜索保存编译检查结果的变量集来检查前者。如果你怀疑有误报，你应该检查后者。

调试 `try_compile` 需要一些注意事项。即使检查不成功，CMake也会删除由该命令生成的所有文件。幸运的是，`debug-trycompile` 将阻止CMake进行删除。如果你的代码中多个 `try_compile` 调用，一次只能调试一个：

1. 运行CMake，不使用 `--debug-trycompile`，将运行所有 `try_compile` 命令，并清理它们的执行目录和文件。
2. 从CMake缓存中删除保存检查结果的变量。缓存保存到 `CMakeCache.txt` 文件中。要清除变量的内容，可以使用 `-U` 的CLI开关，后面跟着变量的名称，它将被解释为一个全局表达式，因此可以使用 `*` 和 `?`：

```
1. $ cmake -U <variable-name>
```

3. 再次运行CMake，使用 `--debug-trycompile`。只有清除缓存的检查才会重新运行。这次不会清理执行目录和文件。

TIPS: `try_compile` 提供了灵活和干净的接口，特别是当编译的代码不是一个简短的代码时。我们建议在测试编译时，小代码片段时使用 `check_<lang>_source_compile`。其他情况下，选择 `try_compile`。

5.7 探究编译器标志命令

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-07> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

设置编译器标志，对是否能正确编译至关重要。不同的编译器供应商，为类似的特性实现有不同的标志。即使是来自同一供应商的不同编译器版本，在可用标志上也可能存在细微的差异。有时，会引入一些便于调试或优化目的的新标志。本示例中，我们将展示如何检查所选编译器是否可用某些标志。

准备工作

Sanitizers(请参考<https://github.com/google/Sanitizers>)已经成为静态和动态代码分析的非常有用的工具。通过使用适当的标志重新编译代码并链接到必要的库，可以检查内存错误(地址清理器)、未初始化的读取(内存清理器)、线程安全(线程清理器)和未定义的行为(未定义的行为清理器)相关的问题。与同类型分析工具相比，Sanitizers带来的性能损失通常要小得多，而且往往提供关于检测到的问题的更详细的信息。缺点是，代码(可能还有工具链的一部分)需要使用附加的标志重新编译。

本示例中，我们将设置一个项目，使用不同的Sanitizers来编译代码，并展示如何检查，编译器标志是否正确使用。

具体实施

Clang编译器已经提供了Sanitizers，GCC也将其引入工具集中。它们是为C和C++程序而设计的。最新版本的Fortran也能使用这些编译标志，并生成正确的仪表化库和可执行程序。不过，本文将重点介绍C++示例。

1. 声明一个C++11项目：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-07 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 声明列表 `CXX_BASIC_FLAGS`，其中包含构建项目时始终使用的编译器标志 `-g3` 和 `-O1`：

```
1. list(APPEND CXX_BASIC_FLAGS "-g3" "-O1")
```

3. 这里需要包括CMake模块 `CheckCXXCompilerFlag.cmake` 。C的模块为 `CheckCCompilerFlag.cmake` , Fortran的模块为 `CheckFortranCompilerFlag.cmake` (Fortran的模块是在CMake 3.3添加)：

```
1. include(CheckCXXCompilerFlag)
```

4. 我们声明一个 `ASAN_FLAGS` 变量，它包含Sanitizer所需的标志，并设置 `CMAKE_REQUIRED_FLAGS` 变量，`check_cxx_compiler_flag` 函数在内部使用该变量：

```
1. set(ASAN_FLAGS "-fsanitize=address -fno-omit-frame-pointer")
2. set(CMAKE_REQUIRED_FLAGS ${ASAN_FLAGS})
```

5. 我们调用 `check_cxx_compiler_flag` 来确保编译器理解 `ASAN_FLAGS` 变量中的标志。调用函数后，我们取消设置 `CMAKE_REQUIRED_FLAGS` :

```
1. check_cxx_compiler_flag(${ASAN_FLAGS} asan_works)
2. unset(CMAKE_REQUIRED_FLAGS)
```

6. 如果编译器理解这些选项，我们将变量转换为一个列表，用分号替换空格：

```
1. if(asan_works)
2.   string(REPLACE " \" ";" _asan_flags ${ASAN_FLAGS})
```

7. 我们添加了一个可执行的目标，为代码定位Sanitizer：

```
1. add_executable(asan-example asan-example.cpp)
```

8. 我们为可执行文件设置编译器标志，以包含基本的和Sanitizer标志：

```
1. target_compile_options(asan-example
2.   PUBLIC
3.   ${CXX_BASIC_FLAGS}
4.   ${_asan_flags}
5. )
```

9. 最后，我们还将Sanitizer标志添加到链接器使用的标志集中。这将关闭 `if(asan_works)` 块：

```
1. target_link_libraries(asan-example PUBLIC ${_asan_flags})
2. endif()
```

完整的示例源代码还展示了如何编译和链接线程、内存和未定义的行为清理器的示例可执行程序。这里不详细讨论这些，因为我们使用相同的模式来检查编译器标志。

NOTE: 在GitHub上可以找到一个定制的CMake模块，用于在您的系统上寻找对Sanitizer的支持：<https://github.com/arsenm/sanitizers-cmake>

工作原理

`check_<lang>_compiler_flag` 函数只是 `check_<lang>_source_compiles` 函数的包装器。这些包装器为特定代码提供了一种快捷方式。在用例中，检查特定代码片段是否编译并不重要，重要的是编译器是否理解一组标志。

Sanitizer的编译器标志也需要传递给链接器。可以使用 `check_<lang>_compiler_flag` 函数来实现，我们需要在调用之前设置 `CMAKE_REQUIRED_FLAGS` 变量。否则，作为第一个参数传递的标志将只对编译器使用。

当前配置中需要注意的是，使用字符串变量和列表来设置编译器标志。使用 `target_compile_options` 和 `target_link_libraries` 函数的字符串变量，将导致编译器和/或链接器报错。CMake将传递引用的这些选项，从而导致解析错误。这说明有必要用列表和随后的字符串操作来表示这些选项，并用分号替换字符串变量中的空格。实际上，CMake中的列表是分号分隔的字符串。

更多信息

我们将在第7章，编写一个函数来测试和设置编译器标志，到时候再回来回顾，并概括测试和设置编译器标志的模式。

5.8 探究可执行命令

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-08> 中找到，其中包含一个C/C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

目前为止，我们已经展示了如何检查给定的源代码，是否可以由所选的编译器编译，以及如何确保所需的编译器和链接器标志可用。此示例中，将显示如何检查是否可以在当前系统上编译、链接和运行代码。

准备工作

本示例的代码示例是复用第3章第9节的配置，并进行微小的改动。之前，我们展示了如何在您的系统上找到ZeroMQ库并将其链接到一个C程序中。本示例中，在生成实际的C++程序之前，我们将检查一个使用GNU/Linux上的系统UUID库的小型C程序是否能够实际运行。

具体实施

开始构建C++项目之前，我们希望检查GNU/Linux上的UUID系统库是否可以被链接。这可以通过以下一系列步骤来实现：

1. 声明一个混合的C和C++11程序。这是必要的，因为我们要编译和运行的测试代码片段是使用C语言完成：

```

1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2. project(recipe-08 LANGUAGES CXX C)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 我们需要在系统上找到UUID库。这通过使用 `pkg-config` 实现的。要求搜索返回一个CMake导入目标使用 `IMPORTED_TARGET` 参数：

```

1. find_package(PkgConfig REQUIRED QUIET)
2. pkg_search_module(UUID REQUIRED uuid IMPORTED_TARGET)
3. if(TARGET PkgConfig::UUID)
4.   message(STATUS "Found libuuid")
5. endif()

```

3. 接下来，需要使用 `CheckCSourceRuns.cmake` 模块。C++的是 `CheckCXXSourceRuns.cmake` 模块。但到CMake 3.11为止，Fortran语言还没有这样的模块：

```
1. include(CheckCSourceRuns)
```

4. 我们声明一个 `_test_uuid` 变量，其中包含要编译和运行的C代码段：

```
1. set(_test_uuid
2. "
3. #include <uuid/uuid.h>
4. int main(int argc, char * argv[]) {
5.     uuid_t uuid;
6.     uuid_generate(uuid);
7.     return 0;
8. }
9. ")
```

5. 我们声明 `CMAKE_REQUIRED_LIBRARIES` 变量后，对 `check_c_source_runs` 函数的调用。接下来，调用 `check_c_source_runs`，其中测试代码作为第一个参数，`_runs` 变量作为第二个参数，以保存执行的检查结果。之后，取消 `CMAKE_REQUIRED_LIBRARIES` 变量的设置：

```
1. set(CMAKE_REQUIRED_LIBRARIES PkgConfig::UUID)
2. check_c_source_runs("${_test_uuid}" _runs)
3. unset(CMAKE_REQUIRED_LIBRARIES)
```

6. 如果检查没有成功，要么是代码段没有编译，要么是没有运行，我们会用致命的错误停止配置：

```
1. if(NOT _runs)
2.     message(FATAL_ERROR "Cannot run a simple C executable using libuuid!")
3. endif()
```

7. 若成功，我们继续添加C++可执行文件作为目标，并链接到UUID：

```
1. add_executable(use-uuid use-uuid.cpp)
2. target_link_libraries(use-uuid
3.   PUBLIC
4.   PkgConfig::UUID
5. )
```

工作原理

`check_<lang>_source_runs` 用于C和C++的函数，与 `check_<lang>_source_compile` 相同，但在实际运行生成的可执行文件的地方需要添加一个步骤。对于 `check_<lang>_source_compiles`，`check_<lang>_source_runs` 的执行可以通过以下变量来进行：

- `CMAKE_REQUIRED_FLAGS`: 设置编译器标志。
- `CMAKE_REQUIRED_DEFINITIONS`: 设置预编译宏。
- `CMAKE_REQUIRED_INCLUDES`: 设置包含目录列表。
- `CMAKE_REQUIRED_LIBRARIES`: 设置可执行目标需要连接的库列表。

由于使用 `pkg_search_module` 生成的为导入目标，所以只需要将 `CMAKE_REQUIRE_LIBRARIES` 设置为 `PkgConfig::UUID`，就可以正确设置包含目录。

正如 `check_<lang>_source_compiles` 是 `try_compile` 的包装器，`check_<lang>_source_runs` 是CMake中另一个功能更强大的命令的包装器：`try_run`。因此，可以编写一个 `CheckFortranSourceRuns.cmake` 模块，通过适当包装 `try_run`，提供与C和C++模块相同的功能。

NOTE: `pkg_search_module` 只能定义导入目标(CMake 3.6)，但目前的示例可以使工作，3.6之前版本的CMake可以通过手动设置所需的包括目录和库 `check_c_source_runs` 如下：
`set(CMAKE_REQUIRED_INCLUDES $ {UUID_INCLUDE_DIRS})` 和 `set(CMAKE_REQUIRED_LIBRARIES $ {UUID_LIBRARIES})`。

5.9 使用生成器表达式微调配置和编译

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-5/recipe-09> 中找到，其中包含一个C++例子。该示例在CMake 3.9版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

CMake提供了一种特定于领域的语言，来描述如何配置和构建项目。自然会引入描述特定条件的变量，并在 `CMakeLists.txt` 中包含基于此的条件语句。

本示例中，我们将重新讨论生成器表达式。第4章中，以简洁地引用显式的测试可执行路径，使用了这些表达式。生成器表达式为逻辑和信息表达式，提供了一个强大而紧凑的模式，这些表达式在生成构建系统时进行评估，并生成特定于每个构建配置的信息。换句话说，生成器表达式用于引用仅在生成时已知，但在配置时未知或难于知晓的信息；对于文件名、文件位置和库文件后缀尤其如此。

本例中，我们将使用生成器表达式，有条件地设置预处理器定义，并有条件地链接到消息传递接口库(Message Passing Interface, MPI)，并允许我们串行或使用MPI构建相同的源代码。

NOTE: 本例中，我们将使用一个导入的目标来链接到MPI，该目标仅从CMake 3.9开始可用。但是，生成器表达式可以移植到CMake 3.0或更高版本。

准备工作

我们将编译以下示例源代码(`example.cpp`)：

```

1. #include <iostream>
2.
3. #ifdef HAVE_MPI
4. #include <mpi.h>
5. #endif
6. int main()
7. {
8. #ifdef HAVE_MPI
9. // initialize MPI
10. MPI_Init(NULL, NULL);
11.
12. // query and print the rank
13. int rank;
14. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15. std::cout << "hello from rank " << rank << std::endl;
16.
17. // initialize MPI

```

```

18.     MPI_Finalize();
19. #else
20.     std::cout << "hello from a sequential binary" << std::endl;
21. #endif /* HAVE_MPI */
22. }
```

代码包含预处理语句(`#ifdef HAVE_MPI ... #else ... #endif`), 这样我们就可以用相同的源代码编译一个顺序的或并行的可执行文件了。

具体实施

编写 `CMakeLists.txt` 文件时, 我们将重用第3章第6节的一些构建块:

1. 声明一个C++11项目:

```

1. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
2. project(recipe-09 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 然后, 我们引入一个选项 `USE_MPI` 来选择MPI并行化, 并将其设置为默认值 `ON`。如果为 `ON`, 我们使用 `find_package` 来定位MPI环境:

```

1. option(USE_MPI "Use MPI parallelization" ON)
2. if(USE_MPI)
3.     find_package(MPI REQUIRED)
4. endif()
```

3. 然后定义可执行目标, 并有条件地设置相应的库依赖项(`MPI::MPI_CXX`)和预处理器定义(`HAVE_MPI`), 稍后将对此进行解释:

```

1. add_executable(example example.cpp)
2. target_link_libraries(example
3.   PUBLIC
4.     ${$<$<BOOL:$\{MPI_FOUND\}>:MPI::MPI_CXX}
5.   )
6. target_compile_definitions(example
7.   PRIVATE
8.     ${$<$<BOOL:$\{MPI_FOUND\}>:HAVE_MPI}
9.   )
```

4. 如果找到MPI，还将打印由 `FindMPI.cmake` 导出的 `INTERFACE_LINK_LIBRARIES`，为了方便演示，使用了 `cmake_print_properties()` 函数：

```

1. if(MPI_FOUND)
2.   include(CMakePrintHelpers)
3.   cmake_print_properties(
4.     TARGETS MPI::MPI_CXX
5.     PROPERTIES INTERFACE_LINK_LIBRARIES
6.   )
7. endif()

```

5. 首先使用默认MPI配置。观察 `cmake_print_properties()` 的输出：

```

1. $ mkdir -p build_mpi
2. $ cd build_mpi
3. $ cmake ..
4.
5. --
6. --
7. Properties for TARGET MPI::MPI_CXX:
  MPI::MPI_CXX.INTERFACE_LINK_LIBRARIES = "-Wl,-rpath -Wl,/usr/lib/openmpi -
    Wl,--enable-new-dtags -
8. pthread;/usr/lib/openmpi/libmpi_cxx.so;/usr/lib/openmpi/libmpi.so"

```

6. 编译并运行并行例子：

```

1. $ cmake --build .
2. $ mpirun -np 2 ./example
3.
4. hello from rank 0
5. hello from rank 1

```

7. 现在，创建一个新的构建目录，这次构建串行版本：

```

1. $ mkdir -p build_seq
2. $ cd build_seq
3. $ cmake -D USE_MPI=OFF ..
4. $ cmake --build .
5. $ ./example
6.
7. hello from a sequential binary

```

工作原理

CMake分两个阶段生成项目的构建系统：配置阶段(解析 `CMakeLists.txt`)和生成阶段(实际生成构建环境)。生成器表达式在第二阶段进行计算，可以使用仅在生成时才能知道的信息来调整构建系统。生成器表达式在交叉编译时特别有用，一些可用的信息只有解析 `CMakeLists.txt` 之后，或在多配置项目后获取，构建系统生成的所有项目可以有不同的配置，比如Debug和Release。

本例中，将使用生成器表达式有条件地设置链接依赖项并编译定义。为此，可以关注这两个表达式：

```

1. target_link_libraries(example
2.   PUBLIC
3.     ${$<BOOL:${MPI_FOUND}>:MPI::MPI_CXX}
4. )
5. target_compile_definitions(example
6.   PRIVATE
7.     ${$<BOOL:${MPI_FOUND}>:HAVE_MPI}
8. )

```

如果 `MPI_FOUND` 为真，那么 `$$<BOOL:${MPI_FOUND}>` 的值将为1。本例中，`$$<BOOL:${MPI_FOUND}>:MPI::MPI_CXX` 将计算 `MPI::MPI_CXX`，第二个生成器表达式将计算结果存在 `HAVE_MPI`。如果将 `USE_MPI` 设置为 `OFF`，则 `MPI_FOUND` 为假，两个生成器表达式的值都为空字符串，因此不会引入链接依赖关系，也不会设置预处理定义。

我们可以通过 `if` 来达到同样的效果：

```

1. if(MPI_FOUND)
2.   target_link_libraries(example
3.     PUBLIC
4.       MPI::MPI_CXX
5.   )
6.
7.   target_compile_definitions(example
8.     PRIVATE
9.       HAVE_MPI
10.    )
11. endif()

```

这个解决方案不太优雅，但可读性更好。我们可以使用生成器表达式来重新表达 `if` 语句，而这个选择取决于个人喜好。但当我们需要访问或操作文件路径时，生成器表达式尤其出色，因为使用变量和 `if` 构造这些路径可能比较困难。本例中，我们更注重生成器表达式的可读性。第4章中，我们使用生成器表达式来解析特定目标的文件路径。第11章中，我们会再次来讨论生成器。

更多信息

CMake提供了三种类型的生成器表达式：

- 逻辑表达式，基本模式为 `$<condition:outcome>`。基本条件为0表示false，1表示true，但是只要使用了正确的关键字，任何布尔值都可以作为条件变量。
- 信息表达式，基本模式为 `$<information>` 或 `$<information:input>`。这些表达式对一些构建系统信息求值，例如：包含目录、目标属性等等。这些表达式的输入参数可能是目标的名称，比如表达式 `$<TARGET_PROPERTY:tgt,prop>`，将获得的信息是tgt目标上的prop属性。
- 输出表达式，基本模式为 `$<operation>` 或 `$<operation:input>`。这些表达式可能基于一些输入参数，生成一个输出。它们的输出可以直接在CMake命令中使用，也可以与其他生成器表达式组合使用。例如，`-I$<JOIN:$<TARGET_PROPERTY:INCLUDE_DIRECTORIES>, -I>` 将生成一个字符串，其中包含正在处理的目标的包含目录，每个目录的前缀由 `-I` 表示。

有关生成器表达式的完整列表，请参

考<https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html>

第6章 生成源码

本章的主要内容如下：

- 配置时生成源码
- 使用Python在配置时生成源码
- 构建时使用Python生成源码
- 记录项目版本信息以便报告
- 从文件中记录项目版本
- 配置时记录Git Hash值
- 构建时记录Git Hash值

大多数项目，使用版本控制跟踪源码。源代码通常作为构建系统的输入，将其转换为o文件、库或可执行程序。某些情况下，我们使用构建系统在配置或构建步骤时生成源代码。根据配置步骤中收集的信息，对源代码进行微调。另一个常用的方式，是记录有关配置或编译的信息，以保证代码行为可重现性。本章中，我们将演示使用CMake提供的源代码生成工具，以及各种相关的策略。

6.1 配置时生成源码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-01> 中找到，其中包含一个Fortran/C例子。该示例在CMake 3.10版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows(使用MSYS Makefiles)上进行过测试。

代码生成在配置时发生，例如：CMake可以检测操作系统和可用库；基于这些信息，我们可以定制构建的源代码。本节和下面的章节中，我们将演示如何生成一个简单源文件，该文件定义了一个函数，用于报告构建系统配置。

准备工作

此示例的代码使用Fortran和C语言编写，第9章将讨论混合语言编程。主程序是一个简单的Fortran可执行程序，它调用一个C函数 `print_info()`，该函数将打印配置信息。值得注意的是，在使用Fortran 2003时，编译器将处理命名问题(对于C函数的接口声明)，如示例所示。我们将使用的 `example.f90` 作为源文件：

```

1. program hello_world
2.
3.   implicit none
4.
5.   interface
6.     subroutine print_info() bind(c, name="print_info")
7.   end subroutine
8. end interface
9.
10. call print_info()
11.
12. end program

```

C函数 `print_info()` 在模板文件 `print_info.c.in` 中定义。在配置时，以 `@` 开头和结尾的变量将被替换为实际值：

```

1. #include <stdio.h>
2. #include <unistd.h>
3.
4. void print_info(void)
5. {
6.   printf("\n");

```

```

7.     printf("Configuration and build information\n");
8.     printf("-----\n");
9.     printf("\n");
10.    printf("Who compiled | %s\n", "@_user_name@");
11.    printf("Compilation hostname | %s\n", "@_host_name@");
12.    printf("Fully qualified domain name | %s\n", "@_fqdn@");
13.    printf("Operating system | %s\n",
14.           "@_os_name@, @_os_release@, @_os_version@");
15.    printf("Platform | %s\n", "@_os_platform@");
16.    printf("Processor info | %s\n",
17.           "@_processor_name@, @_processor_description@");
18.    printf("CMake version | %s\n", "@CMAKE_VERSION@");
19.    printf("CMake generator | %s\n", "@CMAKE_GENERATOR@");
20.    printf("Configuration time | %s\n", "@_configuration_time@");
21.    printf("Fortran compiler | %s\n", "@CMAKE_Fortran_COMPILER@");
22.    printf("C compiler | %s\n", "@CMAKE_C_COMPILER@");
23.    printf("\n");
24.
25.    fflush(stdout);
26. }
```

具体实施

在CMakeLists.txt中，我们首先必须对选项进行配置，并用它们的值替换 `print_info.c.in` 中相应的占位符。然后，将Fortran和C源代码编译成一个可执行文件：

1. 声明了一个Fortran-C混合项目：

```

1. cmake_minimum_required(VERSION 3.10 FATAL_ERROR)
2. project(recipe-01 LANGUAGES Fortran C)
```

2. 使用 `execute_process` 为项目获取当且使用者的信息：

```

1. execute_process(
2.   COMMAND
3.     whoami
4.   TIMEOUT
5.     1
6.   OUTPUT_VARIABLE
7.     _user_name
8.   OUTPUT_STRIP_TRAILING_WHITESPACE
```

```
9.    )
```

3. 使用 `cmake_host_system_information()` 函数(已经在第2章第5节遇到过)，可以查询很多系统信息：

```
1. # host name information
2. cmake_host_system_information(RESULT _host_name QUERY HOSTNAME)
3. cmake_host_system_information(RESULT _fqdn QUERY FQDN)
4.
5. # processor information
6. cmake_host_system_information(RESULT _processor_name QUERY PROCESSOR_NAME)
  cmake_host_system_information(RESULT _processor_description QUERY
7. PROCESSOR_DESCRIPTION)
8.
9. # os information
10. cmake_host_system_information(RESULT _os_name QUERY OS_NAME)
11. cmake_host_system_information(RESULT _os_release QUERY OS_RELEASE)
12. cmake_host_system_information(RESULT _os_version QUERY OS_VERSION)
13. cmake_host_system_information(RESULT _os_platform QUERY OS_PLATFORM)
```

4. 捕获配置时的时间戳，并通过使用字符串操作函数：

```
1. string(TIMESTAMP _configuration_time "%Y-%m-%d %H:%M:%S [UTC]" UTC)
```

5. 现在，准备好配置模板文件 `print_info.c.in`。通过CMake的 `configure_file` 函数生成代码。注意，这里只要求以 `@` 开头和结尾的字符串被替换：

```
1. configure_file(print_info.c.in print_info.c @ONLY)
```

6. 最后，我们添加一个可执行目标，并定义目标源：

```
1. add_executable(example "")
2. target_sources(example
3.   PRIVATE
4.   example.f90
5.   ${CMAKE_CURRENT_BINARY_DIR}/print_info.c
6. )
```

7. 下面是一个输出示例：

```
1. $ mkdir -p build
```

```

2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./example
6.
7. Configuration and build information
8. -----
9. Who compiled | somebody
10. Compilation hostname | laptop
11. Fully qualified domain name | laptop
    Operating system | Linux, 4.16.13-1-ARCH, #1 SMP PREEMPT Thu May 31
12. 23:29:29 UTC 2018
13. Platform | x86_64
    Processor info | Unknown P6 family, 2 core Intel(R) Core(TM) i5-5200U CPU
14. @ 2.20GHz
15. CMake version | 3.11.3
16. CMake generator | Unix Makefiles
17. Configuration time | 2018-06-25 15:38:03 [UTC]
18. Fortran compiler | /usr/bin/f95
19. C compiler | /usr/bin/cc

```

工作原理

`configure_file` 命令可以复制文件，并用变量值替换它们的内容。示例中，使用 `configure_file` 修改模板文件的内容，并将其复制到一个位置，然后将其编译到可执行文件中。如何调用 `configure_file`：

```
1. configure_file(print_info.c.in print_info.c @ONLY)
```

第一个参数是模板的名称为 `print_info.c.in`。CMake假设输入文件的目录，与项目的根目录相对；也就是说，在 `${CMAKE_CURRENT_SOURCE_DIR}/print_info.c.in`。我们选择 `print_info.c`，作为第二个参数是配置文件的名称。假设输出文件位于相对于项目构建目录的位置： `${CMAKE_CURRENT_BINARY_DIR}/print_info.c`。

输入和输出文件作为参数时，CMake不仅将配置 `@VAR@` 变量，还将配置 `${VAR}` 变量。如果 `${VAR}` 是语法的一部分，并且不应该修改(例如在shell脚本中)，那么就很不方便。为了在引导CMake，应该将选项 `@ONLY` 传递给 `configure_file` 的调用，如前所述。

更多信息

注意，用值替换占位符时，CMake中的变量名应该与将要配置的文件中使用的变量名完全相同，并放

在 `@` 之间。可以在调用 `configure_file` 时定义的任何CMake变量。我们的示例中，这包括所有内置的CMake变量，如 `CMAKE_VERSION` 或 `CMAKE_GENERATOR`。此外，每当修改模板文件时，重新生成代码将触发生成系统的重新生成。这样，配置的文件将始终保持最新。

TIPS: 通过使用 `CMake --help-variable-list`，可以从CMake手册中获得完整的内部CMake变量列表。

NOTE: `file(GENERATE...)` 为提供了一个有趣的替代 `configure_file`，这是因为 `file` 允许将生成器表达式作为配置文件的一部分进行计算。但是，每次运行CMake时，`file(GENERATE...)` 都会更新输出文件，这将强制重新构建依赖于该输出的所有目标。详细可参见 <https://crascit.com/2017/04/18/generated-sources-in-cmake-build>。

6.2 使用Python在配置时生成源码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-02> 中找到，其中包含一个Fortran/C例子。该示例在CMake 3.10版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows(使用MSYS Makefile)上进行过测试。

本示例中，我们将再次从模板 `print_info.c.in` 生成 `print_info.c`。但这一次，将假设CMake函数 `configure_file()` 没有创建源文件，然后使用Python脚本模拟这个过程。当然，对于实际的项目，我们可能更倾向于使用 `configure_file()`，但有时使用Python生成源代码的需要时，我们也应该知道如何应对。

这个示例有严重的限制，不能完全模拟 `configure_file()`。我们在这里介绍的方法，不能生成一个自动依赖项，该依赖项将在构建时重新生成 `print_info.c`。换句话说，如果在配置之后删除生成的 `print_info.c`，则不会重新生成该文件，构建也会失败。要正确地模拟 `configure_file()`，需要使用 `add_custom_command()` 和 `add_custom_target()`。我们将在第3节中使用它们，来克服这个限制。

这个示例中，我们将使用一个简单的Python脚本。这个脚本将读取 `print_info.c.in`。用从CMake传递给Python脚本的参数替换文件中的占位符。对于更复杂的模板，我们建议使用外部工具，比如Jinja(参见<http://jinja.pocoo.org>)。

```

1. def configure_file(input_file, output_file, vars_dict):
2.
3.     with input_file.open('r') as f:
4.         template = f.read()
5.
6.     for var in vars_dict:
7.         template = template.replace('@' + var + '@', vars_dict[var])
8.
9.     with output_file.open('w') as f:
10.        f.write(template)

```

这个函数读取一个输入文件，遍历 `vars_dict` 变量中的目录，并用对应的值替换 `@key@`，再将结果写入输出文件。这里的键值对，将由CMake提供。

准备工作

`print_info.c.in` 和 `example.f90` 与之前的示例相同。此外，我们将使用Python脚本 `configuration.py`，它提供了一个函数：

```

1. def configure_file(input_file, output_file, vars_dict):
2.     with input_file.open('r') as f:
3.         template = f.read()
4.
5.     for var in vars_dict:
6.         template = template.replace('@' + var + '@', vars_dict[var])
7.
8.     with output_file.open('w') as f:
9.         f.write(template)

```

该函数读取输入文件，遍历 `vars_dict` 字典的所有键，用对应的值替换模式 `@key@`，并将结果写入输出文件(键值由CMake提供)。

具体实施

与前面的示例类似，我们需要配置一个模板文件，但这一次，使用Python脚本模拟 `configure_file()` 函数。我们保持CMakeLists.txt基本不变，并提供一组命令进行替换操作 `configure_file(print_info.c.in print_info.c @ONLY)`，接下来将逐步介绍这些命令：

- 首先，构造一个变量 `_config_script`，它将包含一个Python脚本，稍后我们将执行这个脚本：

```

1. set(_config_script
2. "
3. from pathlib import Path
4. source_dir = Path('${CMAKE_CURRENT_SOURCE_DIR}')
5. binary_dir = Path('${CMAKE_CURRENT_BINARY_DIR}')
6. input_file = source_dir / 'print_info.c.in'
7. output_file = binary_dir / 'print_info.c'
8.
9. import sys
10. sys.path.insert(0, str(source_dir))
11.
12. from configurator import configure_file
13. vars_dict = {
14.     '_user_name': '${_user_name}',
15.     '_host_name': '${_host_name}',
16.     '_fqdn': '${_fqdn}',
17.     '_processor_name': '${_processor_name}',
18.     '_processor_description': '${_processor_description}',
19.     '_os_name': '${_os_name}',
```

```

20.      '_os_release': '${_os_release}',
21.      '_os_version': '${_os_version}',
22.      '_os_platform': '${_os_platform}',
23.      '_configuration_time': '${_configuration_time}',
24.      'CMAKE_VERSION': '${CMAKE_VERSION}',
25.      'CMAKE_GENERATOR': '${CMAKE_GENERATOR}',
26.      'CMAKE_Fortran_COMPILER': '${CMAKE_Fortran_COMPILER}',
27.      'CMAKE_C_COMPILER': '${CMAKE_C_COMPILER}',
28.  }
29. configure_file(input_file, output_file, vars_dict)
30. ")

```

2. 使用 `find_package` 让CMake使用Python解释器：

- `find_package(PythonInterp QUIET REQUIRED)`

3. 如果找到Python解释器，则可以在CMake中执行 `_config_script`，并生成 `print_info.c` 文件：

```

1. execute_process(
2.   COMMAND
3.     ${PYTHON_EXECUTABLE} "-c" ${_config_script}
4. )

```

4. 之后，定义可执行目标和依赖项，这与前一个示例相同。所以，得到的输出没有变化。

工作原理

回顾一下对CMakeLists.txt的更改。

我们执行了一个Python脚本生成 `print_info.c`。运行Python脚本前，首先检测Python解释器，并构造Python脚本。Python脚本导入 `configure_file` 函数，我们在 `configuration.py` 中定义了这个函数。为它提供用于读写的文件位置，并将其值作为键值对。

此示例展示了生成配置的另一种方法，将生成任务委托给外部脚本，可以将配置报告编译成可执行文件，甚至库目标。我们在前面的配置中认为的第一种方法更简洁，但是使用本示例中提供的方法，我们可以灵活地使用Python(或其他语言)，实现任何在配置时间所需的步骤。使用当前方法，我们可以通过脚本的方式执行类似 `cmake_host_system_information()` 的操作。

但要记住，这种方法也有其局限性，它不能在构建时重新生成 `print_info.c` 的自动依赖项。下一个示例中，我们应对这个挑战。

更多信息

我们可以使用 `get_cmake_property(_vars VARIABLES)` 来获得所有变量的列表，而不是显式地构造 `vars_dict`（这感觉有点重复），并且可以遍历 `_vars` 的所有元素来访问它们的值：

```
1. get_cmake_property(_vars VARIABLES)
2. foreach(_var IN ITEMS ${_vars})
3.   message("variable ${_var} has the value ${${_var}}")
4. endforeach()
```

使用这种方法，可以隐式地构建 `vars_dict`。但是，必须注意转义包含字符的值，例如：`;`，Python会将其解析为一条指令的末尾。

6.3 构建时使用Python生成源码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-03> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

构建时根据某些规则生成冗长和重复的代码，同时避免在源代码存储库中显式地跟踪生成的代码生成源代码，是开发人员工具箱中的一个重要工具，例如：根据检测到的平台或体系结构生成不同的源代码。或者，可以使用Python，根据配置时收集的输入，在构建时生成高效的C++代码。其他生成器解析器，比如：Flex (<https://github.com/westes/flex>) 和 Bison(<https://www.gnu.org/software/bison/>)；元对象编译器，如Qt的 moc(<http://doc.qt.io/qt5/moc.html>)；序列化框架，如谷歌的protobuf (<https://developers.google.com/protocol-buffers/>)。

准备工作

为了提供一个具体的例子，我们需要编写代码来验证一个数字是否是质数。现在有很多算法，例如：可以用埃拉托色尼的筛子(sieve of Eratosthenes)来分离质数和非质数。如果有很多验证数字，我们不希望对每一个数字都进行Eratosthenes筛选。我们想要做的是将所有质数一次制表，直到数字的上限，然后使用一个表查的方式，找来验证大量的数字。

本例中，将在编译时使用Python为查找表(质数向量)生成C++代码。当然，为了解决这个特殊的编程问题，我们还可以使用C++生成查询表，并且可以在运行时执行查询。

让我们从 `generate.py` 脚本开始。这个脚本接受两个命令行参数——一个整数范围和一个输出文件名：

```

1. """
2. Generates C++ vector of prime numbers up to max_number
3. using sieve of Eratosthenes.
4. """
5. import pathlib
6. import sys
7.
8. # for simplicity we do not verify argument list
9. max_number = int(sys.argv[-2])
10. output_file_name = pathlib.Path(sys.argv[-1])
11.
12. numbers = range(2, max_number + 1)
13. is_prime = {number: True for number in numbers}
14.
```

```

15. for number in numbers:
16.     current_position = number
17.     if is_prime[current_position]:
18.         while current_position <= max_number:
19.             current_position += number
20.             is_prime[current_position] = False
21.
22. primes = (number for number in numbers if is_prime[number])
23.
24. code = """#pragma once
25.
26. #include <vector>
27.
28. const std::size_t max_number = {max_number};
29. std::vector<int> & primes() {{
30.     static std::vector<int> primes;
31.     {push_back}
32.     return primes;
33. }
34. """
35. push_back = '\n'.join([' primes.push_back({:d});'.format(x) for x in primes])
36. output_file_name.write_text(
37.     code.format(max_number=max_number, push_back=push_back))

```

我们的目标是生成一个 `primes.hpp`，并将其包含在下面的示例代码中：

```

1. #include "primes.hpp"
2.
3. #include <iostream>
4. #include <vector>
5.
6. int main() {
7.     std::cout << "all prime numbers up to " << max_number << ":";
8.
9.     for (auto prime : primes())
10.         std::cout << " " << prime;
11.
12.     std::cout << std::endl;
13.
14.     return 0;
15. }

```

具体实施

下面是CMakeLists.txt命令的详解：

- 首先，定义项目并检测Python解释器：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
6. find_package(PythonInterp QUIET REQUIRED)

```

- 将生成的代码放在 `${CMAKE_CURRENT_BINARY_DIR}/generate` 下，需要告诉CMake创建这个目录：

```
1. file(MAKE_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/generated)
```

- Python脚本要求质数的上限，使用下面的命令，我们可以设置一个默认值：

```
1. set(MAX_NUMBER "100" CACHE STRING "Upper bound for primes")
```

- 接下来，定义一个自定义命令来生成头文件：

```

1. add_custom_command(
2.   OUTPUT
3.     ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
4.   COMMAND
5.     ${PYTHON_EXECUTABLE} generate.py ${MAX_NUMBER}
5.   ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
6.   WORKING_DIRECTORY
7.     ${CMAKE_CURRENT_SOURCE_DIR}
8.   DEPENDS
9.     generate.py
10. )

```

- 最后，定义可执行文件及其目标，包括目录和依赖关系：

```

1. add_executable(example "")
2. target_sources(example
3.   PRIVATE

```

```

4.     example.cpp
5.     ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
6.   )
7. target_include_directories(example
8. PRIVATE
9.     ${CMAKE_CURRENT_BINARY_DIR}/generated
10. )

```

6. 准备测试：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./example
    all prime numbers up to 100: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53
6. 59 61 67 71 73 79

```

具体实施

为了生成头文件，我们定义了一个自定义命令，它执行 `generate.py` 脚本，并接受 `MAX_NUMBER` 和文件路径(`${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp`)作为参数：

```

1. add_custom_command(
2.   OUTPUT
3.     ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
4.   COMMAND
5.     ${PYTHON_EXECUTABLE} generate.py ${MAX_NUMBER}
5. ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
6.   WORKING_DIRECTORY
7.     ${CMAKE_CURRENT_SOURCE_DIR}
8.   DEPENDS
9.     generate.py
10. )

```

为了生成源代码，我们需要在可执行文件的定义中，使用 `target_sources` 很容易实现添加源代码作为依赖项：

```

1. target_sources(example
2. PRIVATE

```

```
3.      example.cpp
4.      ${CMAKE_CURRENT_BINARY_DIR}/generated/primes.hpp
5.  )
```

前面的代码中，我们不需要定义新的目标。头文件将作为示例的依赖项生成，并在每次 `generate.py` 脚本更改时重新生成。如果代码生成脚本生成多个源文件，那么要将所有生成的文件列出，做为某些目标的依赖项。

更多信息

我们提到所有的生成文件，都应该作为某个目标的依赖项。但是，我们可能不知道这个文件列表，因为它是生成文件的脚本决定的，这取决于我们提供给配置的输入。这种情况下，我们可能会尝试使用 `file(GLOB...)` 将生成的文件收集到一个列表中(参见 <https://cmake.org/cmake/help/v3.5/command/file.html>)。

`file(GLOB...)` 在配置时执行，而代码生成是在构建时发生的。因此可能需要一个间接操作，将 `file(GLOB...)` 命令放在一个单独的CMake脚本中，使用 `${CMAKE_COMMAND} -P` 执行该脚本，以便在构建时获得生成的文件列表。

6.4 记录项目版本信息以便报告

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-04> 中找到，其中包含一个C和Fortran例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

代码版本很重要，不仅是为了可重复性，还为了记录API功能或简化支持请求和bug报告。源代码通常处于某种版本控制之下，例如：可以使用Git标记附加额外版本号(参见<https://semver.org>)。然而，不仅需要对源代码进行版本控制，而且可执行文件还需要记录项目版本，以便将其打印到代码输出或用户界面上。

本例中，将在CMake源文件中定义版本号。我们的目标是在配置项目时将程序版本记录到头文件中。然后，生成的头文件可以包含在代码的正确位置和时间，以便将代码版本打印到输出文件或屏幕上。

准备工作

将使用以下C文件(`example.c`)打印版本信息：

```

1. #include "version.h"
2.
3. #include <stdio.h>
4.
5. int main() {
6.     printf("This is output from code %s\n", PROJECT_VERSION);
7.     printf("Major version number: %i\n", PROJECT_VERSION_MAJOR);
8.     printf("Minor version number: %i\n", PROJECT_VERSION_MINOR);
9.
10.    printf("Hello CMake world!\n");
11. }
```

这里，假设 `PROJECT_VERSION_MAJOR`、`PROJECT_VERSION_MINOR` 和 `PROJECT_VERSION` 是在 `version.h` 中定义的。目标是从以下模板中生成 `version.h.in`：

```

1. #pragma once
2.
3. #define PROJECT_VERSION_MAJOR @PROJECT_VERSION_MAJOR@
4. #define PROJECT_VERSION_MINOR @PROJECT_VERSION_MINOR@
5. #define PROJECT_VERSION_PATCH @PROJECT_VERSION_PATCH@
6.
```

```
7. #define PROJECT_VERSION "v@PROJECT_VERSION@"
```

这里使用预处理器定义，也可以使用字符串或整数常量来提高类型安全性（稍后我们将对此进行演示）。从CMake的角度来看，这两种方法是相同的。

如何实施

我们将按照以下步骤，在模板头文件中对版本进行注册：

1. 要跟踪代码版本，我们可以在CMakeLists.txt中调用CMake的 `project` 时定义项目版本：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-04 VERSION 2.0.1 LANGUAGES C)
```

2. 然后，基于 `version.h.in` 生成 `version.h`：

```
1. configure_file(
2.   version.h.in
3.   generated/version.h
4.   @ONLY
5. )
```

3. 最后，我们定义了可执行文件，并提供了目标包含路径：

```
1. add_executable(example example.c)
2. target_include_directories(example
3.   PRIVATE
4.     ${CMAKE_CURRENT_BINARY_DIR}/generated
5. )
```

工作原理

当使用版本参数调用CMake的 `project` 时，CMake将为项目设置 `PROJECT_VERSION_MAJOR`、`PROJECT_VERSION_MINOR` 和 `PROJECT_VERSION_PATCH`。此示例中的关键命令是 `configure_file`，它接受一个输入文件（本例中是 `version.h.in`），并通过将 `@` 之间的占位符替换成对应的CMake变量，生成一个输出文件（本例中是 `generated/version.h`）。它将 `@PROJECT_VERSION_MAJOR@` 替换为2，以此类推。使用关键字 `@ONLY`，我们将 `configure_file` 限制为只替换 `@variables@`，而不修改 `${variables}` 。后一种形式在 `version.h.in` 中没有使用。但是，当使用CMake配置shell脚本时，会经常出现。

生成的头文件可以包含在示例代码中，可以打印版本信息：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./example
6.
7. This is output from code v2.0.1
8. Major version number: 2
9. Minor version number: 0
10. Hello CMake world!

```

NOTE: CMake以 `x.y.z` 格式给出的版本号，并将变量 `PROJECT_VERSION` 和 `<project-name>_VERSION` 设置为给定的值。此外，`PROJECT_VERSION_MAJOR` (`<project-name>_VERSION_MAJOR`)，`PROJECT_VERSION_MINOR` (`<project-name>_VERSION_MINOR`)，`PROJECT_VERSION_PATCH` (`<project-name>_VERSION_PATCH`) 和 `PROJECT_VERSION_TWEAK` (`<project-name>_VERSION_TWEAK`)，将分别设置为 `X`，`Y`，`Z` 和 `t`。

更多信息

为了确保只有当CMake变量被认为是一个真正的常量时，才定义预处理器变量，可以使用 `configure_file`，在配置的头文件中使用 `#cmakedefin` 而不是 `#define`。

根据是否定义了CMake变量并将其计算为一个真正的常量，`#cmakedefine YOUR_VARIABLE` 将被替换为 `#define YOUR_VARIABLE ...` 或者 `/* #undef YOUR_VARIABLE */`。还有 `#cmakedefine01`，将根据变量是否定义，将变量设置为 `0` 或 `1`。

6.5 从文件中记录项目版本

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-05> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

这个示例的目的和前一个相似，但是出发点不同。我们计划是从文件中读取版本信息，而不是将其设置在CMakeLists.txt中。将版本保存在单独文件中的动机，是允许其他构建框架或开发工具使用独立于CMake的信息，而无需将信息复制到多个文件中。与CMake并行使用的构建框架的一个例子是Sphinx文档框架，它生成文档并将其部署到阅读文档服务中，以便在线提供代码文档。

准备工作

我们将从一个名为 `VERSION` 的文件开始，其中包含以下内容：

```
1. 2.0.1-rc-2
```

这一次，选择更安全的数据类型，并将 `PROGRAM_VERSION` 定义为 `version.hpp.in` 中的字符串常量：

```
1. #pragma once
2. #include <string>
3. const std::string PROGRAM_VERSION = "@PROGRAM_VERSION@";
```

下面的源码(`example.cpp`)，将包含生成的 `version.hpp`：

```
1. // provides PROGRAM_VERSION
2. #include "version.hpp"
3. #include <iostream>
4.
5. int main() {
6.     std::cout << "This is output from code v" << PROGRAM_VERSION
7.     << std::endl;
8.     std::cout << "Hello CMake world!" << std::endl;
9. }
```

具体实施

逐步来完成我们的任务：

1. CMakeLists.txt 定义了最低版本、项目名称、语言和标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-05 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 从文件中读取版本信息如下：

```

1. if(EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/VERSION")
2.   file(READ "${CMAKE_CURRENT_SOURCE_DIR}/VERSION" PROGRAM_VERSION)
3.   string(STRIPE "${PROGRAM_VERSION}" PROGRAM_VERSION)
4. else()
5.   message(FATAL_ERROR "File ${CMAKE_CURRENT_SOURCE_DIR}/VERSION not
6. found")
endif()

```

3. 配置头文件：

```

1. configure_file(
2.   version.hpp.in
3.   generated/version.hpp
4.   @ONLY
5. )

```

4. 最后，定义了可执行文件及其依赖关系：

```

1. add_executable(example example.cpp)
2. target_include_directories(example
3.   PRIVATE
4.     ${CMAKE_CURRENT_BINARY_DIR}/generated
5. )

```

5. 进行测试：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./example

```

```
6.  
7. This is output from code v2.0.1-rc-2  
8. Hello CMake world!
```

工作原理

我们使用以下构造，从一个名为VERSION的文件中读取版本字符串：

```
1. if(EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/VERSION")  
2.   file(READ "${CMAKE_CURRENT_SOURCE_DIR}/VERSION" PROGRAM_VERSION)  
3.   string(STRIPE "${PROGRAM_VERSION}" PROGRAM_VERSION)  
4. else()  
5.   message(FATAL_ERROR "File ${CMAKE_CURRENT_SOURCE_DIR}/VERSION not found")  
6. endif()
```

这里，首先检查该文件是否存在，如果不存在，则发出错误消息。如果存在，将内容读入 `PROGRAM_VERSION` 变量中，该变量会去掉尾部的空格。当设置了变量 `PROGRAM_VERSION`，就可以使用它来配置 `version.hpp.in`，生成 `generated/version.hpp`：

```
1. configure_file(  
2.   version.hpp.in  
3.   generated/version.hpp  
4.   @ONLY  
5. )
```

6.6 配置时记录Git Hash值

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-06> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

大多数现代源代码存储库都使用Git作为版本控制系统进行跟踪，这可以归功于存储库托管平台GitHub的流行。因此，我们将在本示例中使用Git；然而，实际中会根据具体的动机和实现，可以转化为其他版本控制系统。我们以Git为例，提交的Git Hash决定了源代码的状态。因此，为了标记可执行文件，我们将尝试将Git Hash记录到可执行文件中，方法是将哈希字符串记录在一个头文件中，该头文件可以包含在代码中。

准备工作

我们需要两个源文件，类似于前面的示例。其中一个将配置记录的Hash(`version.hpp.in`)，详情如下：

```
1. #pragma once
2. #include <string>
3. const std::string GIT_HASH = "@GIT_HASH@";
```

还需要一个示例源文件(`example.cpp`)，将Hash打印到屏幕上：

```
1. #include "version.hpp"
2.
3. #include <iostream>
4.
5. int main() {
    std::cout << "This code has been configured from version " << GIT_HASH <<
6. std::endl;
7. }
```

此示例还假定在Git存储库中至少有一个提交。因此，使用`git init` 初始化这个示例，并使用`git add <filename>`，然后使用`git commit` 创建提交，以便获得一个有意义的示例。

具体实施

下面演示了从Git记录版本信息的步骤：

1. 定义项目和支持语言：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-06 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 定义 `GIT_HASH` 变量：

```

1. # in case Git is not available, we default to "unknown"
2. set(GIT_HASH "unknown")
3.
4. # find Git and if available set GIT_HASH variable
5. find_package(Git QUIET)
6. if(GIT_FOUND)
7.   execute_process(
8.     COMMAND ${GIT_EXECUTABLE} log -1 --pretty=format:%h
9.     OUTPUT_VARIABLE GIT_HASH
10.    OUTPUT_STRIP_TRAILING_WHITESPACE
11.    ERROR_QUIET
12.    WORKING_DIRECTORY
13.    ${CMAKE_CURRENT_SOURCE_DIR}
14.  )
15. endif()
16.
17. message(STATUS "Git hash is ${GIT_HASH}")

```

3. `CMakeLists.txt` 剩余的部分，类似于之前的示例：

```

1. # generate file version.hpp based on version.hpp.in
2. configure_file(
3.   version.hpp.in
4.   generated/version.hpp
5.   @ONLY
6. )
7.
8. # example code
9. add_executable(example example.cpp)
10.
11. # needs to find the generated header file
12. target_include_directories(example

```

```

13.    PRIVATE
14.        ${CMAKE_CURRENT_BINARY_DIR}/generated
15.    )

```

4. 验证输出(Hash不同):

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ./example
6.
7. This code has been configured from version d58c64f

```

工作原理

使用 `find_package(Git QUIET)` 来检测系统上是否有可用的Git。如果有 (`GIT_FOUND` 为 `True`)，运行一个Git命令： `${GIT_EXECUTABLE} log -1 --pretty=format:%h`。这个命令给出了当前提交Hash的简短版本。当然，这里我们可以灵活地运行Git命令。我们要求 `execute_process` 命令将结果放入名为 `GIT_HASH` 的变量中，然后删除任何尾随的空格。使用 `ERROR_QUIET`，如果Git命令由于某种原因失败，我们不会停止配置。

由于Git命令可能会失败(源代码已经分发到Git存储库之外)，或者Git在系统上不可用，我们希望为这个变量设置一个默认值，如下所示：

```
1. set(GIT_HASH "unknown")
```

此示例有一个问题，`Git Hash`是在配置时记录的，而不是在构建时记录。下一个示例中，我们将演示如何实现后一种方法。

6.7 构建时记录Git Hash值

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-6/recipe-07> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例中，在配置时记录了代码存储库(Git Hash)的状态。然而，前一种方法有一个令人不满意的地方，如果在配置代码之后更改分支或提交更改，则源代码中包含的版本记录可能指向错误的Git Hash值。在这个示例中，我们将演示如何在构建时记录Git Hash(或者，执行其他操作)，以确保每次构建代码时都运行这些操作，因为我们可能只配置一次，但是会构建多次。

准备工作

我们将使用与之前示例相同的 `version.hpp.in`，只会对 `example.cpp` 文件进行修改，以确保它打印构建时Git提交Hash值：

```

1. #include "version.hpp"
2.
3. #include <iostream>
4.
5. int main() {
    std::cout << "This code has been built from version " << GIT_HASH <<
6. std::endl;
7. }
```

具体实施

将Git信息保存到 `version.hpp` 头文件在构建时需要进行以下操作：

- 把前一个示例的 `CMakeLists.txt` 中的大部分代码移到一个单独的文件中，并将该文件命名为 `git-hash.cmake`：

```

1. # in case Git is not available, we default to "unknown"
2. set(GIT_HASH "unknown")
3.
4. # find Git and if available set GIT_HASH variable
5. find_package(Git QUIET)
6. if(GIT_FOUND)
7.     execute_process(
```

```

8.     COMMAND ${GIT_EXECUTABLE} log -1 --pretty=format:%h
9.     OUTPUT_VARIABLE GIT_HASH
10.    OUTPUT_STRIP_TRAILING_WHITESPACE
11.    ERROR_QUIET
12.  )
13. endif()
14.
15. message(STATUS "Git hash is ${GIT_HASH}")
16.
17. # generate file version.hpp based on version.hpp.in
18. configure_file(
19.   ${CMAKE_CURRENT_LIST_DIR}/version.hpp.in
20.   ${TARGET_DIR}/generated/version.hpp
21.   @ONLY
22. )

```

2. CMakeLists.txt 熟悉的部分：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3. # project name and language
4. project(recipe-07 LANGUAGES CXX)
5. # require C++11
6. set(CMAKE_CXX_STANDARD 11)
7. set(CMAKE_CXX_EXTENSIONS OFF)
8. set(CMAKE_CXX_STANDARD_REQUIRED ON)
9. # example code
10. add_executable(example example.cpp)
11. # needs to find the generated header file
12. target_include_directories(example
13.   PRIVATE
14.   ${CMAKE_CURRENT_BINARY_DIR}/generated
15. )

```

3. CMakeLists.txt 的剩余部分，记录了每次编译代码时的 Git Hash：

```

1. add_custom_command(
2.   OUTPUT
3.   ${CMAKE_CURRENT_BINARY_DIR}/generated/version.hpp
4.   ALL
5.   COMMAND

```

```

1. ${CMAKE_COMMAND} -D TARGET_DIR=${CMAKE_CURRENT_BINARY_DIR} -P
2. ${CMAKE_CURRENT_SOURCE_DIR}/git-hash.cmake
3. WORKING_DIRECTORY
4. ${CMAKE_CURRENT_SOURCE_DIR}
5. )
6.
7.
8.
9.
10.
11. # rebuild version.hpp every time
12. add_custom_target(
13.   get_git_hash
14.   ALL
15.   DEPENDS
16.   ${CMAKE_CURRENT_BINARY_DIR}/generated/version.hpp
17. )
18.
19. # version.hpp has to be generated
20. # before we start building example
21. add_dependencies(example get_git_hash)

```

工作原理

示例中，在构建时执行CMake代码。为此，定义了一个自定义命令：

```

1. add_custom_command(
2.   OUTPUT
3.   ${CMAKE_CURRENT_BINARY_DIR}/generated/version.hpp
4.   ALL
5.   COMMAND
6.   ${CMAKE_COMMAND} -D TARGET_DIR=${CMAKE_CURRENT_BINARY_DIR} -P
7.   ${CMAKE_CURRENT_SOURCE_DIR}/git-hash.cmake
8.   WORKING_DIRECTORY
9.   ${CMAKE_CURRENT_SOURCE_DIR}
10. )

```

我们还定义了一个目标：

```

1. add_custom_target(
2.   get_git_hash
3.   ALL
4.   DEPENDS
5.   ${CMAKE_CURRENT_BINARY_DIR}/generated/version.hpp
6. )

```

自定义命令调用CMake来执行 `git-hash.cmake` 脚本。这里使用CLI的 `-P` 开关，通过传入脚本的位置实现的。请注意，可以像往常一样使用CLI开关 `-D` 传递选项。`git-hash.cmake` 脚本生成 `${TARGET_DIR}/generated/version.hpp` 。自定义目标被添加到 `ALL` 目标中，并且依赖于自定义命令的输出。换句话说，当构建默认目标时，我们确保自定义命令已经运行。此外，自定义命令将 `ALL` 目标作为输出。这样，我们就能确保每次都会生成 `version.hpp` 了。

更多信息

我们可以改进配置，以便在记录的 `Git Hash` 外，包含其他的信息。检测构建环境是否“污染”(即是否包含未提交的更改和未跟踪的文件)，或者“干净”。可以使用 `git describe --abbrev=7 --long --always --dirty --tags` 检测这些信息。根据可重现性，甚至可以将Git的状态，完整输出记录到头文件中，我们将这些功能作为课后习题留给读者自己完成。

第7章 构建项目

本章的主要内容如下：

- 使用函数和宏重用代码
- 将CMake源代码分成模块
- 编写函数来测试和设置编译器标志
- 用指定参数定义函数或宏
- 重新定义函数和宏
- 使用废弃函数、宏和变量
- `add_subdirectory`的限定范围
- 使用`target_sources`避免全局变量
- 组织Fortran项目

前几章中，我们已经使用了一些CMake构建块来配置和构建的项目。本章中，我们将讨论如何组合这些构建块，并引入抽象，并最小化代码重复、全局变量、全局状态和显式排序，以免`CMakeLists.txt`文件过于庞大。目标是为模块化CMake代码结构和限制变量范围提供模式。我们将讨论一些策略，也将帮助我们控制中大型代码项目的CMake代码复杂性。

7.1 使用函数和宏重用代码

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-01> 中找到，其中包含一个C++例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

任何编程语言中，函数允许我们抽象(隐藏)细节并避免代码重复，CMake也不例外。本示例中，我们将以宏和函数为例进行讨论，并介绍一个宏，以便方便地定义测试和设置测试的顺序。我们的目标是定义一个宏，能够替换 `add_test` 和 `set_tests_properties`，用于定义每组和设置每个测试的预期开销(第4章，第8节)。

准备工作

我们将基于第4章第2节中的例子。`main.cpp`、`sum_integers.cpp` 和 `sum_integers.hpp` 文件不变，用来计算命令行参数提供的整数队列的和。单元测试(`test.cpp`)的源代码也没有改变。我们还需要Catch 2头文件，`catch.hpp`。与第4章相反，我们将把源文件放到子目录中，并形成以下文件树(稍后我们将讨论CMake代码)：

```

1. .
2. └── CMakeLists.txt
3. └── src
4.   ├── CMakeLists.txt
5.   ├── main.cpp
6.   ├── sum_integers.cpp
7.   └── sum_integers.hpp
8. └── tests
9.   ├── catch.hpp
10.  ├── CMakeLists.txt
11.  └── test.cpp

```

具体实施

1. 定义了CMake最低版本、项目名称和支持的语言，并要求支持C++11标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 根据GNU标准定义 `binary` 和 `library` 路径：

```

1. include(GNUInstallDirs)
2.
3. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
4.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
5. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
6.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
7. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
8.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

```

3. 最后，使用 `add_subdirectory` 调用 `src/CMakeLists.txt` 和 `tests/CMakeLists.txt`：

```

1. add_subdirectory(src)
2. enable_testing()
3. add_subdirectory(tests)

```

4. `src/CMakeLists.txt` 定义了源码目标：

```

1. set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)
2. add_library(sum_integers sum_integers.cpp)
3. add_executable(sum_up main.cpp)
4. target_link_libraries(sum_up sum_integers)

```

5. `tests/CMakeLists.txt` 中，构建并链接 `cpp_test` 可执行文件：

```

1. add_executable(cpp_test test.cpp)
2. target_link_libraries(cpp_test sum_integers)

```

6. 定义一个新宏 `add_catch_test`：

```

1. macro(add_catch_test _name _cost)
2.   math(EXPR num_macro_calls "${num_macro_calls} + 1")
3.   message(STATUS "add_catch_test called with ${ARGC} arguments: ${ARGV}")
4.
5.   set(_argn "${ARGN}")
6.   if(_argn)
7.     message(STATUS "oops - macro received argument(s) we did not expect:
8. ${ARGN}")
9.   endif()

```

```

10.    add_test(
11.        NAME
12.        ${_name}
13.        COMMAND
14.        ${<TARGET_FILE:cpp_test>}
15.        [${_name}] --success --out
16.        ${PROJECT_BINARY_DIR}/tests/${_name}.log --durations yes
17.        WORKING_DIRECTORY
18.        ${CMAKE_CURRENT_BINARY_DIR}
19.    )
20.
21.    set_tests_properties(
22.        ${_name}
23.        PROPERTIES
24.        COST ${_cost}
25.    )
26. endmacro()

```

7. 最后，使用 `add_catch_test` 定义了两个测试。此外，还设置和打印了变量的值：

```

1. set(num_macro_calls 0)
2. add_catch_test(short 1.5)
3. add_catch_test(long 2.5 extra_argument)
   message(STATUS "in total there were ${num_macro_calls} calls to
4. add_catch_test")

```

8. 现在，进行测试。配置项目(输出行如下所示)：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- ...
6. -- add_catch_test called with 2 arguments: short;1.5
7. -- add_catch_test called with 3 arguments: long;2.5;extra_argument
8. -- oops - macro received argument(s) we did not expect: extra_argument
9. -- in total there were 2 calls to add_catch_test
10. --

```

9. 最后，构建并运行测试：

```
1. $ cmake --build .
```

```
2. $ ctest
```

10. 长时间的测试会先开始：

```
1. Start 2: long
2. 1/2 Test #2: long ..... Passed 0.00 sec
3. Start 1: short
4. 2/2 Test #1: short ..... Passed 0.00 sec
5.
6. 100% tests passed, 0 tests failed out of 2
```

工作原理

这个配置中的新添加了 `add_catch_test` 宏。这个宏需要两个参数 `_name` 和 `_cost`，可以在宏中使用这些参数来调用 `add_test` 和 `set_tests_properties`。参数前面的下划线，是为了向读者表明这些参数只能在宏中访问。另外，宏自动填充了 `${ARGC}` (参数数量) 和 `${ARGV}` (参数列表)，我们可以在输出中验证了这一点：

```
1. -- add_catch_test called with 2 arguments: short;1.5
2. -- add_catch_test called with 3 arguments: long;2.5;extra_argument
```

宏还定义了 `${ARGN}`，用于保存最后一个参数之后的参数列表。此外，我们还可以使用 `${ARGV0}`、 `${ARGV1}` 等来处理参数。我们演示一下，如何捕捉到调用中的额外参数 (`extra_argument`)：

```
1. add_catch_test(long 2.5 extra_argument)
```

我们使用了以下方法：

```
1. set(_argn "${ARGN}")
2. if(_argn)
   message(STATUS "oops - macro received argument(s) we did not expect:
3. ${ARGN}")
4. endif()
```

这个 `if` 语句中，我们引入一个新变量，但不能直接查询 `ARGN` ，因为它不是通常意义上的CMake变量。使用这个宏，我们可以通过它们的名称和命令来定义测试，还可以指示预期的开销，这会让耗时长的测试在耗时短测试之前启动，这要归功于 `COST` 属性。

我们可以用一个函数来实现它，而不是使用相同语法的宏：

```

1. function(add_catch_test _name _cost)
2. ...
3. endfunction()

```

宏和函数之间的区别在于它们的变量范围。宏在调用者的范围内执行，而函数有自己的变量范围。换句话说，如果我们使用宏，需要设置或修改对调用者可用的变量。如果不设置或修改输出变量，最好使用函数。我们注意到，可以在函数中修改父作用域变量，但这必须使用 `PARENT_SCOPE` 显式表示：

```

1. set(variable_visible_outside "some value" PARENT_SCOPE)

```

为了演示作用域，我们在定义宏之后编写了以下调用：

```

1. set(num_macro_calls 0)
2. add_catch_test(short 1.5)
3. add_catch_test(long 2.5 extra_argument)
   message(STATUS "in total there were ${num_macro_calls} calls to"
4. add_catch_test")

```

在宏内部，将 `num_macro_calls` 加1：

```

1. math(EXPR num_macro_calls "${num_macro_calls} + 1")

```

这时产生的输出：

```

1. -- in total there were 2 calls to add_catch_test

```

如果我们将宏更改为函数，测试仍然可以工作，但是 `num_macro_calls` 在父范围内的所有调用中始终为0。将CMake宏想象成类似函数是很有用的，这些函数被直接替换到它们被调用的地方（在C语言中内联）。将CMake函数想象成黑盒函数很有必要。黑盒中，除非显式地将其定义为 `PARENT_SCOPE`，否则不会返回任何内容。CMake中的函数没有返回值。

更多信息

可以在宏中嵌套函数调用，也可以在函数中嵌套宏调用，但是这就需要仔细考虑变量的作用范围。如果功能可以使用函数实现，那么这可能比宏更好，因为它对父范围状态提供了更多的默认控制。

我们还应该提到在 `src/cmakelist.txt` 中使用 `CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE`：

```

1. set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)

```

这个命令会将当前目录，添加到 `CMakeLists.txt` 中定义的所有目标的 `interface_include_directory` 属性中。换句话说，我们不需要使用 `target_include_directory` 来添加 `cpp_test` 所需头文件的位置。

7.2 将CMake源代码分成模块

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-02> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

项目通常从单个 `CMakeLists.txt` 文件开始，随着时间的推移，这个文件会逐渐增长。本示例中，我们将演示一种将 `CMakeLists.txt` 分割成更小单元的机制。将 `CMakeLists.txt` 拆分为模块有几个动机，这些模块可以包含在主 `CMakeLists.txt` 或其他模块中：

- 主 `CMakeLists.txt` 更易于阅读。
- CMake模块可以在其他项目中重用。
- 与函数相结合，模块可以帮助我们限制变量的作用范围。

本示例中，我们将演示如何定义和包含一个宏，该宏允许我们获得CMake的彩色输出(用于重要的状态消息或警告)。

准备工作

本例中，我们将使用两个文件，主 `CMakeLists.txt` 和 `cmake/colors.cmake`：

```

1. .
2. └── cmake
3.   └── colors.cmake
4. └── CMakeLists.txt

```

`cmake/colors.cmake` 文件包含彩色输出的定义：

```

1. # colorize CMake output
2. # code adapted from stackoverflow: http://stackoverflow.com/a/19578320
3. # from post authored by https://stackoverflow.com/users/2556117/fraser
4. macro(define_colors)
5.   if(WIN32)
6.     # has no effect on WIN32
7.     set(ColourReset "")
8.     set(ColourBold "")
9.     set(Red "")
10.    set(Green "")
11.    set(Yellow "")
12.    set(Blue "")

```

```

13.     set(Magenta "")
14.     set(Cyan "")
15.     set(White "")
16.     set(BoldRed "")
17.     set(BoldGreen "")
18.     set(BoldYellow "")
19.     set(BoldBlue "")
20.     set(BoldMagenta "")
21.     set(BoldCyan "")
22.     set(BoldWhite "")
23. else()
24.     string(ASCII 27 Esc)
25.     set(ColourReset "${Esc}[m")
26.     set(ColourBold "${Esc}[1m")
27.     set(Red "${Esc}[31m")
28.     set(Green "${Esc}[32m")
29.     set(Yellow "${Esc}[33m")
30.     set(Blue "${Esc}[34m")
31.     set(Magenta "${Esc}[35m")
32.     set(Cyan "${Esc}[36m")
33.     set(White "${Esc}[37m")
34.     set(BoldRed "${Esc}[1;31m")
35.     set(BoldGreen "${Esc}[1;32m")
36.     set(BoldYellow "${Esc}[1;33m")
37.     set(BoldBlue "${Esc}[1;34m")
38.     set(BoldMagenta "${Esc}[1;35m")
39.     set(BoldCyan "${Esc}[1;36m")
40.     set(BoldWhite "${Esc}[1;37m")
41. endif()
42. endmacro()

```

具体实施

来看下我们如何使用颜色定义，来生成彩色状态消息：

- 从一个熟悉的头部开始：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-02 LANGUAGES NONE)

```

- 然后，将 `cmake` 子目录添加到CMake模块搜索的路径列表中：

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

3. 包括 `colors.cmake` 模块，调用其中定义的宏：

```
1. include(colors)
2. define_colors()
```

4. 最后，打印了不同颜色的信息：

```
1. message(STATUS "This is a normal message")
2. message(STATUS "${Red}This is a red${ColourReset}")
3. message(STATUS "${BoldRed}This is a bold red${ColourReset}")
4. message(STATUS "${Green}This is a green${ColourReset}")
5. message(STATUS "${BoldMagenta}This is bold${ColourReset}")
```

5. 测试一下(如果使用macOS或Linux，以下的输出应该出现屏幕上)：

```
-- This is a normal message
-- This is a red
-- This is a bold red
-- This is a green
-- This is bold
-- Configuring done
-- Generating done
-- Build files have been written to: /home/roberto/Workspace/robertodr/cmake-cookbook/chapter-07/recipe-02/example/build
```

工作原理

这个例子中，不需要编译代码，也不需要语言支持，我们已经用 `LANGUAGES NONE` 明确了这一点：

```
1. project(recipe-02 LANGUAGES NONE)
```

我们定义了 `define_colors` 宏，并将其放在 `cmake/colors.cmake`。因为还是希望使用调用宏中定义的变量，来更改消息中的颜色，所以我们选择使用宏而不是函数。我们使用以下行包括宏和调用 `define_colors`：

```
1. include(colors)
2. define_colors()
```

我们还需要告诉CMake去哪里查找宏：

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

`include(colors)` 命令指示CMake搜索 `${CMAKE_MODULE_PATH}` , 查找名称为 `colors.cmake` 的模块。

例子中，我们没有按以下的方式进行：

```
1. list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")
2. include(colors)
```

而是使用一个显式包含的方式：

```
1. include(cmake/colors.cmake)
```

更多信息

推荐的做法是在模块中定义宏或函数，然后调用宏或函数。将包含模块用作函数调用不是很好的方式。除了定义函数和宏以及查找程序、库和路径之外，包含模块不应该做更多的事情。实际的 `include` 命令不应该定义或修改变量，其原因是重复的 `include` (可能是偶然的)不应该引入任何不想要的副作用。在第5节中，我们将创建一个防止多次包含的保护机制。

7.3 编写函数来测试和设置编译器标志

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-03> 中找到，其中包含一个C/C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前两个示例中，我们使用了宏。本示例中，将使用一个函数来抽象细节并避免代码重复。我们将实现一个接受编译器标志列表的函数。该函数将尝试用这些标志逐个编译测试代码，并返回编译器理解的第一个标志。这样，我们将了解几个新特性：函数、列表操作、字符串操作，以及检查编译器是否支持相应的标志。

准备工作

按照上一个示例的推荐，我们将在(`set_compiler_flag.cmake`)模块中定义函数，然后调用函数。该模块包含以下代码，我们将在后面详细讨论：

```

1. include(CheckCCCompilerFlag)
2. include(CheckCXXCompilerFlag)
3. include(CheckFortranCompilerFlag)
4. function(set_compiler_flag _result _lang)
5.   # build a list of flags from the arguments
6.   set(_list_of_flags)
7.   # also figure out whether the function
8.   # is required to find a flag
9.   set(_flag_is_required FALSE)
10.  foreach(_arg IN ITEMS ${ARGN})
11.    string(TOUPPER "${_arg}" _arg_uppercase)
12.    if(_arg_uppercase STREQUAL "REQUIRED")
13.      set(_flag_is_required TRUE)
14.    else()
15.      list(APPEND _list_of_flags "${_arg}")
16.    endif()
17.  endforeach()
18.
19.  set(_flag_found FALSE)
20.  # loop over all flags, try to find the first which works
21.  foreach(flag IN ITEMS ${_list_of_flags})
22.    unset(_flag_works CACHE)
23.    if(_lang STREQUAL "C")
24.      check_c_compiler_flag("${flag}" _flag_works)

```

```

25.     elseif(_lang STREQUAL "CXX")
26.         check_cxx_compiler_flag("${flag}" _flag_works)
27.     elseif(_lang STREQUAL "Fortran")
28.         check_Fortran_compiler_flag("${flag}" _flag_works)
29.     else()
30.         message(FATAL_ERROR "Unknown language in set_compiler_flag:
31. ${_lang}")
32.     endif()
33.
34.     # if the flag works, use it, and exit
35.     # otherwise try next flag
36.     if(_flag_works)
37.         set(${_result} "${flag}" PARENT_SCOPE)
38.         set(_flag_found TRUE)
39.         break()
40.     endif()
41.     endforeach()
42.
43.     # raise an error if no flag was found
44.     if(_flag_is_required AND NOT _flag_found)
45.         message(FATAL_ERROR "None of the required flags were supported")
46.     endif()
47. endfunction()

```

具体实施

展示如何在CMakeLists.txt中使用 `set_compiler_flag` 函数：

1. 定义最低CMake版本、项目名称和支持的语言(本例中是C和C++)：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES C CXX)

```

2. 显示包含 `set_compiler_flag.cmake` :

```

1. include(set_compiler_flag.cmake)

```

3. 测试C标志列表：

```

1. set_compiler_flag(
2.   working_compile_flag C REQUIRED

```

```

3.   "-foo" # this should fail
4.   "-wrong" # this should fail
5.   "-wrong" # this should fail
6.   "-Wall" # this should work with GNU
7.   "-warn all" # this should work with Intel
8.   "-Minform=inform" # this should work with PGI
9.   "-nope" # this should fail
10.  )
11.
12. message(STATUS "working C compile flag: ${working_compile_flag}")

```

4. 测试C++标志列表：

```

1. set_compiler_flag(
2.   working_compile_flag CXX REQUIRED
3.   "-foo" # this should fail
4.   "-g" # this should work with GNU, Intel, PGI
5.   "/RTCcsu" # this should work with MSVC
6.   )
7.
8. message(STATUS "working CXX compile flag: ${working_compile_flag}")

```

5. 现在，我们可以配置项目并验证输出。只显示相关的输出，相应的输出可能会因编译器的不同而有所不同：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- ...
6. -- Performing Test _flag_works
7. -- Performing Test _flag_works - Failed
8. -- Performing Test _flag_works
9. -- Performing Test _flag_works - Failed
10. -- Performing Test _flag_works
11. -- Performing Test _flag_works - Failed
12. -- Performing Test _flag_works
13. -- Performing Test _flag_works - Success
14. -- working C compile flag: -Wall
15. -- Performing Test _flag_works
16. -- Performing Test _flag_works - Failed
17. -- Performing Test _flag_works

```

```
18. -- Performing Test _flag_works - Success
19. -- working CXX compile flag: -g
20. -- ...
```

工作原理

这里使用的模式是：

1. 定义一个函数或宏，并将其放入模块中
2. 包含模块
3. 调用函数或宏

从输出中，可以看到代码检查列表中的每个标志。一旦检查成功，它就打印成功的编译标志。看看 `set_compiler_flag.cmake` 模块的内部，这个模块又包含三个模块：

```
1. include(CheckCCompilerFlag)
2. include(CheckCXXCompilerFlag)
3. include(CheckFortranCompilerFlag)
```

这都是标准的CMake模块，CMake将在 `${CMAKE_MODULE_PATH}` 中找到它们。这些模块分别提供 `check_c_compiler_flag` 、 `check_cxx_compiler_flag` 和 `check_fortran_compiler_flag` 宏。然后定义函数：

```
1. function(set_compiler_flag _result _lang)
2. ...
3. endfunction()
```

`set_compiler_flag` 函数需要两个参数，`_result`（保存成功编译标志或为空字符串）和 `_lang`（指定语言：C、C++或Fortran）。

我们也能这样调用函数：

```
1. set_compiler_flag(working_compile_flag C REQUIRED "-Wall" "-warn all")
```

这里有五个调用参数，但是函数头只需要两个参数。这意味着 `REQUIRED` 、 `-Wall` 和 `-warn all` 将放在 `${ARGN}` 中。从 `${ARGN}` 开始，我们首先使用 `foreach` 构建一个标志列表。同时，从标志列表中过滤出 `REQUIRED`，并使用它来设置 `_flag_is_required`：

```
1. # build a list of flags from the arguments
2. set(_list_of_flags)
3. # also figure out whether the function
```

```

4. # is required to find a flag
5. set(_flag_is_required FALSE)
6. foreach(_arg IN ITEMS ${ARGN})
7.   string(TOUPPER "${_arg}" _arg_uppercase)
8.   if(_arg_uppercase STREQUAL "REQUIRED")
9.     set(_flag_is_required TRUE)
10.    else()
11.      list(APPEND _list_of_flags "${_arg}")
12.    endif()
13. endforeach()

```

现在，我们将循环 `_${list_of_flags}`，尝试每个标志，如果 `_flag_works` 被设置为 `TRUE`，我们将 `_flag_found` 设置为 `TRUE`，并中止进一步的搜索：

```

1. set(_flag_found FALSE)
2. # loop over all flags, try to find the first which works
3. foreach(flag IN ITEMS ${_list_of_flags})
4.
5.   unset(_flag_works CACHE)
6.   if(_lang STREQUAL "C")
7.     check_c_compiler_flag("${flag}" _flag_works)
8.   elseif(_lang STREQUAL "CXX")
9.     check_cxx_compiler_flag("${flag}" _flag_works)
10.  elseif(_lang STREQUAL "Fortran")
11.    check_Fortran_compiler_flag("${flag}" _flag_works)
12.  else()
13.    message(FATAL_ERROR "Unknown language in set_compiler_flag: ${_lang}")
14.  endif()
15.
16.  # if the flag works, use it, and exit
17.  # otherwise try next flag
18.  if(_flag_works)
19.    set(${_result} "${flag}" PARENT_SCOPE)
20.    set(_flag_found TRUE)
21.    break()
22.  endif()
23. endforeach()

```

`unset(_flag_works CACHE)` 确保 `check_*_compiler_flag` 的结果，不会在使用 `_flag_works result` 变量时，使用的是缓存结果。

如果找到了标志，并且 `_flag_works` 设置为 `TRUE`，我们就将 `_result` 映射到的变量：

```
1. set(${_result} "${flag}" PARENT_SCOPE)
```

这需要使用 `PARENT_SCOPE` 来完成，因为我们正在修改一个变量，希望打印并在函数体外部使用该变量。请注意，如何使用 `_${result}` 语法解引用，从父范围传递的变量 `_result` 的值。不管函数的名称是什么，这对于确保工作标志被设置非常有必要。如果没有找到任何标志，并且该标志设置了 `REQUIRED`，那我们将使用一条错误消息停止配置：

```
1. # raise an error if no flag was found
2. if(_flag_is_required AND NOT _flag_found)
3.   message(FATAL_ERROR "None of the required flags were supported")
4. endif()
```

更多信息

我们也可以使用宏来完成这个任务，而使用函数可以对范围有更多的控制。我们知道函数只能修改结果变量。

另外，需要在编译和链接时设置一些标志，方法是为 `check_<lang>_compiler_flag` 函数设置 `CMAKE_REQUIRED_FLAGS`。如第5章，第7节中讨论的那样，Sanitizer就是这种情况。



7.4 用指定参数定义函数或宏

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-04> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例中，我们研究了函数和宏，并使用了位置参数。这个示例中，我们将定义一个带有命名参数的函数。我们将复用第1节中的示例，使用函数和宏重用代码，而不是使用以下代码定义测试：`add_catch_test(short 1.5)`。

我们将这样调用函数：

```

1. add_catch_test(
2.     NAME
3.     short
4.     LABELS
5.     short
6.     cpp_test
7.     COST
8.     1.5
9. )
```

准备工作

我们使用第1节中的示例，使用函数和宏重用代码，并保持C++源代码不变，文件树保持不变：

```

1. .
2. └── cmake
3.   └── testing.cmake
4. └── CMakeLists.txt
5. └── src
6.   ├── CMakeLists.txt
7.   ├── main.cpp
8.   ├── sum_integers.cpp
9.   └── sum_integers.hpp
10. └── tests
11.   ├── catch.hpp
12.   ├── CMakeLists.txt
13.   └── test.cpp
```

具体实施

我们对CMake代码进行一些修改，如下所示：

1. `CMakeLists.txt` 顶部中只增加了一行，因为我们将包括位于 `cmake` 下面的模块：

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

2. 保持 `src/CMakeLists.txt`。
3. `tests/CMakeLists.txt` 中，将 `add_catch_test` 函数定义移动到 `cmake/testing.cmake`，并且定义两个测试：

```
1. add_executable(cpp_test test.cpp)
2. target_link_libraries(cpp_test sum_integers)
3.
4. include(testing)
5.
6. add_catch_test(
7.   NAME
8.   short
9.   LABELS
10.  short
11.  cpp_test
12.  COST
13.  1.5
14. )
15.
16. add_catch_test(
17.   NAME
18.   long
19.   LABELS
20.   long
21.   cpp_test
22.   COST
23.   2.5
24. )
```

4. `add_catch_test` 在 `cmake/testing.cmake` 中定义：

```
1. function(add_catch_test)
2.   set(options)
```

```

3.    set(oneValueArgs NAME COST)
4.    set(multiValueArgs LABELS DEPENDS REFERENCE_FILES)
5.    cmake_parse_arguments(add_catch_test
6.        "${options}"
7.        "${oneValueArgs}"
8.        "${multiValueArgs}"
9.        ${ARGN}
10.    )
11.   message(STATUS "defining a test ...")
12.   message(STATUS " NAME: ${add_catch_test_NAME}")
13.   message(STATUS " LABELS: ${add_catch_test_LABELS}")
14.   message(STATUS " COST: ${add_catch_test_COST}")
15.   message(STATUS " REFERENCE_FILES: ${add_catch_test_REFERENCE_FILES}")
16.
17.   add_test(
18.       NAME
19.           ${add_catch_test_NAME}
20.       COMMAND
21.           ${TARGET_FILE:cpp_test}
22.           [${add_catch_test_NAME}] --success --out
23.               ${PROJECT_BINARY_DIR}/tests/${add_catch_test_NAME}.log --durations
24.       yes
25.       WORKING_DIRECTORY
26.           ${CMAKE_CURRENT_BINARY_DIR}
27.   )
28.   set_tests_properties(${add_catch_test_NAME}
29.       PROPERTIES
30.           LABELS "${add_catch_test_LABELS}"
31.   )
32.
33.   if(add_catch_test_COST)
34.       set_tests_properties(${add_catch_test_NAME}
35.           PROPERTIES
36.               COST ${add_catch_test_COST}
37.       )
38.   endif()
39.
40.   if(add_catch_test_DEPENDS)
41.       set_tests_properties(${add_catch_test_NAME}
42.           PROPERTIES
43.               DEPENDS ${add_catch_test_DEPENDS})

```

```

44.      )
45.  endif()
46.
47.  if(add_catch_test_REFERENCE_FILES)
48.    file(
49.      COPY
50.        ${add_catch_test_REFERENCE_FILES}
51.      DESTINATION
52.        ${CMAKE_CURRENT_BINARY_DIR}
53.      )
54.  endif()
55. endfunction()

```

5. 测试输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- ...
6. -- defining a test ...
7. -- NAME: short
8. -- LABELS: short;cpp_test
9. -- COST: 1.5
10. -- REFERENCE_FILES:
11. -- defining a test ...
12. -- NAME: long
13. -- LABELS: long;cpp_test
14. -- COST: 2.5
15. -- REFERENCE_FILES:
16. --

```

6. 最后，编译并测试：

```

1. $ cmake --build .
2. $ ctest

```

工作原理

示例的特点是其命名参数，因此我们可以将重点放在 `cmake/testing.cmake` 模块上。CMake 提供 `cmake_parse_arguments` 命令，我们使用函数名 (`add_catch_test`) 选项 (我们的例子中

是 `none`)、单值参数(`NAME` 和 `COST`)和多值参数(`LABELS` 、 `DEPENDS` 和 `REFERENCE_FILES`)调用该命令：

```

1. function(add_catch_test)
2.   set(options)
3.   set(oneValueArgs NAME COST)
4.   set(multiValueArgs LABELS DEPENDS REFERENCE_FILES)
5.   cmake_parse_arguments(add_catch_test
6.     "${options}"
7.     "${oneValueArgs}"
8.     "${multiValueArgs}"
9.     ${ARGN}
10.    )
11. ...
12. endfunction()

```

`cmake_parse_arguments` 命令解析选项和参数，并在例子中定义如下：

- `add_catch_test_NAME`
- `add_catch_test_COST`
- `add_catch_test_LABELS`
- `add_catch_test_DEPENDS`
- `add_catch_test_REFERENCE_FILES`

可以查询，并在函数中使用这些变量。这种方法使我们有机会用更健壮的接口和更具有可读的函数/宏调用，来实现函数和宏。

更多信息

选项关键字(本例中我们没有使用)由 `cmake_parse_arguments` 定义

为 `TRUE` 或 `FALSE` 。 `add_catch_test` 函数，还提供 `test` 命令作为一个命名参数，为了更简洁的演示，我们省略了这个参数。

TIPS: `cmake_parse_arguments` 命令在 `cmake 3.5` 的版本前中
的 `CMakeParseArguments.cmake` 定义。因此，可以在 `CMake/test.cmake` 顶部的使
用 `include(CMakeParseArguments)` 命令使此示例能与 `CMake` 早期版本一起工作。

7.5 重新定义函数和宏

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-05> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

我们已经提到模块包含不应该用作函数调用，因为模块可能被包含多次。本示例中，我们将编写我们自己的“包含保护”机制，如果多次包含一个模块，将触发警告。内置的 `include_guard` 命令从3.10版开始可以使用，对于C/C++头文件，它的行为就像 `#pragma` 一样。对于当前版本的CMake，我们将演示如何重新定义函数和宏，并且展示如何检查CMake版本，对于低于3.10的版本，我们将使用定制的“包含保护”机制。

准备工作

这个例子中，我们将使用三个文件：

```
1. .
2. └── cmake
3.   ├── custom.cmake
4.   └── include_guard.cmake
5. └── CMakeLists.txt
```

`custom.cmake` 模块包含以下代码：

```
1. include_guard(GLOBAL)
2. message(STATUS "custom.cmake is included and processed")
```

我们稍后会对 `cmake/include_guard.cmake` 进行讨论。

具体实施

我们对三个CMake文件的逐步分解：

- 示例中，我们不会编译任何代码，因此我们的语言要求是 `NONE`：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-05 LANGUAGES NONE)
```

- 定义一个 `include_guard` 宏，将其放在一个单独的模块中：

```

1. # (re)defines include_guard
2. include(cmake/include_guard.cmake)

```

3. `cmake/include_guard.cmake` 文件包含以下内容(稍后将详细讨论):

```

1. macro(include_guard)
2.   if (CMAKE_VERSION VERSION_LESS "3.10")
3.     # for CMake below 3.10 we define our
4.     # own include_guard(GLOBAL)
5.     message(STATUS "calling our custom include_guard")
6.
7.     # if this macro is called the first time
8.     # we start with an empty list
9.     if(NOT DEFINED included_modules)
10.      set(included_modules)
11.    endif()
12.
13.    if ("${CMAKE_CURRENT_LIST_FILE}" IN_LIST included_modules)
14.      message(WARNING "module ${CMAKE_CURRENT_LIST_FILE} processed more
15. than once")
16.    endif()
17.    list(APPEND included_modules ${CMAKE_CURRENT_LIST_FILE})
18.  else()
19.    # for CMake 3.10 or higher we augment
20.    # the built-in include_guard
21.    message(STATUS "calling the built-in include_guard")
22.
23.    _include_guard(${ARGV})
24.  endif()
25. endmacro()

```

4. 主CMakeLists.txt中，我们模拟了两次包含自定义模块的情况：

```

1. include(cmake/custom.cmake)
2. include(cmake/custom.cmake)

```

5. 最后，使用以下命令进行配置：

```

1. $ mkdir -p build
2. $ cd build

```

```
3. $ cmake ..
```

6. 使用CMake 3.10及更高版本的结果如下：

```
1. -- calling the built-in include_guard
2. -- custom.cmake is included and processed
3. -- calling the built-in include_guard
```

7. 使用CMake得到3.10以下的结果如下：

```
1. - calling our custom include_guard
2. -- custom.cmake is included and processed
3. -- calling our custom include_guard
4. CMake Warning at cmake/include_guard.cmake:7 (message):
5. module
6. /home/user/example/cmake/custom.cmake
7. processed more than once
8. Call Stack (most recent call first):
9. cmake/custom.cmake:1 (include_guard)
10. CMakeLists.txt:12 (include)
```

工作原理

`include_guard` 宏包含两个分支，一个用于CMake低于3.10，另一个用于CMake高于3.10：

```
1. macro(include_guard)
2.   if (CMAKE_VERSION VERSION_LESS "3.10")
3.     #
4.   else()
5.     #
6.   endif()
7. endmacro()
```

如果CMake版本低于3.10，进入第一个分支，并且内置的 `include_guard` 不可用，所以我们自定义了一个：

```
1. message(STATUS "calling our custom include_guard")
2.
3. # if this macro is called the first time
4. # we start with an empty list
```

```

5. if(NOT DEFINED included_modules)
6.     set(included_modules)
7. endif()
8.
9. if ("${CMAKE_CURRENT_LIST_FILE}" IN_LIST included_modules)
    message(WARNING "module ${CMAKE_CURRENT_LIST_FILE} processed more than
10. once")
11. endif()
12.
13. list(APPEND included_modules ${CMAKE_CURRENT_LIST_FILE})

```

如果第一次调用宏，则 `included_modules` 变量没有定义，因此我们将其设置为空列表。然后检查 `${CMAKE_CURRENT_LIST_FILE}` 是否是 `included_modules` 列表中的元素。如果是，则会发出警告；如果没有，我们将 `${CMAKE_CURRENT_LIST_FILE}` 追加到这个列表。CMake输出中，我们可以验证自定义模块的第二个包含确实会导致警告。

CMake 3.10及更高版本的情况有所不同；在这种情况下，存在一个内置的 `include_guard`，我们用自己的宏接收到参数并调用它：

```

1. macro(include_guard)
2.   if (CMAKE_VERSION VERSION_LESS "3.10")
3.     #
4.   else()
5.     message(STATUS "calling the built-in include_guard")
6.
7.     _include_guard(${ARGV})
8.   endif()
9. endmacro()

```

这里，`_include_guard(${ARGV})` 指向内置的 `include_guard`。本例中，使用自定义消息（“调用内置的 `include_guard`”）进行了扩展。这种模式为我们提供了一种机制，来重新定义自己的或内置的函数和宏，这对于调试或记录日志来说非常有用。

NOTE:这种模式可能很有用，但是应该谨慎使用，因为CMake不会对重新定义的宏或函数进行警告。

7.6 使用废弃函数、宏和变量

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-06> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

“废弃”是在不断发展的项目开发过程中一种重要机制，它向开发人员发出信号，表明将来某个函数、宏或变量将被删除或替换。在一段时间内，函数、宏或变量将继续可访问，但会发出警告，最终可能会上升为错误。

准备工作

我们将从以下CMake项目开始：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-06 LANGUAGES NONE)
4.
5. macro(custom_include_guard)
6.   if(NOT DEFINED included_modules)
7.     set(included_modules)
8.   endif()
9.
10.  if ("${CMAKE_CURRENT_LIST_FILE}" IN_LIST included_modules)
11.    message(WARNING "module ${CMAKE_CURRENT_LIST_FILE} processed more than
12. once")
13.  endif()
14.  list(APPEND included_modules ${CMAKE_CURRENT_LIST_FILE})
15. endmacro()
16.
17. include(cmake/custom.cmake)
18.
19. message(STATUS "list of all included modules: ${included_modules}")

```

这段代码定义了一个自定义的“包含保护”机制，包括一个自定义模块(与前一个示例中的模块相同)，并打印所有包含模块的列表。对于CMake 3.10或更高版本有内置的 `include_guard`。但是，不能简单地删除 `custom_include_guard` 和 `${included_modules}`，而是使用一个“废弃”警告来弃用宏和变量。某个时候，可以将该警告转换为 `FATAL_ERROR`，使代码停止配置，并迫使开发人员对代码进行修改，切换到内置命令。

具体实施

“废弃”函数、宏和变量的方法如下：

- 首先，定义一个函数，我们将使用它来弃用一个变量：

```

1. function(deprecate_variable _variable _access)
2.   if(_access STREQUAL "READ_ACCESS")
3.     message(DEPRECATION "variable ${_variable} is deprecated")
4.   endif()
5. endfunction()

```

- 然后，如果CMake的版本大于3.9，我们重新定义 `custom_include_guard` 并将 `variable_watch` 附加到 `included_modules` 中：

```

1. if (CMAKE_VERSION VERSION_GREATER "3.9")
2.   # deprecate custom_include_guard
3.   macro(custom_include_guard)
4.     message(DEPRECATION "custom_include_guard is deprecated - use built-in
5. include_guard instead")
6.     _custom_include_guard(${ARGV})
7.   endmacro()
8.
9.   # deprecate variable included_modules
10.  variable_watch(included_modules deprecate_variable)
endif()

```

- CMake3.10以下版本的项目会产生以下结果：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- custom.cmake is included and processed
6. -- list of all included modules: /home/user/example/cmake/custom.cmake

```

- CMake 3.10及以上将产生预期的“废弃”警告：

```

1. CMake Deprecation Warning at CMakeLists.txt:26 (message):
2. custom_include_guard is deprecated - use built-in include_guard instead
3. Call Stack (most recent call first):

```

```

4. cmake/custom.cmake:1 (custom_include_guard)
5. CMakeLists.txt:34 (include)
6. -- custom.cmake is included and processed
7. CMake Deprecation Warning at CMakeLists.txt:19 (message):
8. variable included_modules is deprecated
9. Call Stack (most recent call first):
10. CMakeLists.txt:9999 (deprecate_variable)
11. CMakeLists.txt:36 (message)
12. -- list of all included modules: /home/user/example/cmake/custom.cmake

```

工作原理

弃用函数或宏相当于重新定义它，如前面的示例所示，并使用 `DEPRECATION` 打印消息：

```

1. macro(somemacro)
2.   message(DEPRECATION "somemacro is deprecated")
3.   _somemacro(${ARGV})
4. endmacro()

```

可以通过定义以下变量来实现对变量的弃用：

```

1. function(deprecate_variable _variable _access)
2.   if(_access STREQUAL "READ_ACCESS")
3.     message(DEPRECATION "variable ${_variable} is deprecated")
4.   endif()
5. endfunction()

```

然后，这个函数被添加到将要“废弃”的变量上：

```
1. variable_watch(somevariable deprecate_variable)
```

如果在本例中 `${included_modules}` 是读取 (`READ_ACCESS`)，那么 `deprecate_variable` 函数将发出带有 `DEPRECATION` 的消息。

7.7 add_subdirectory的限定范围

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-07> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本章剩下的示例中，我们将讨论构建项目的策略，并限制变量的范围和副作用，目的是降低代码的复杂性和简化项目的维护。这个示例中，我们将把一个项目分割成几个范围有限的CMakeLists.txt文件，这些文件将使用 `add_subdirectory` 命令进行处理。

准备工作

由于我们希望展示和讨论如何构造一个复杂的项目，所以需要一个比“hello world”项目更复杂的例子：

- https://en.wikipedia.org/wiki/Cellular_automaton#Elementary_cellular_automata
- <http://mathworld.wolfram.com/ElementaryCellularAutomaton.html>

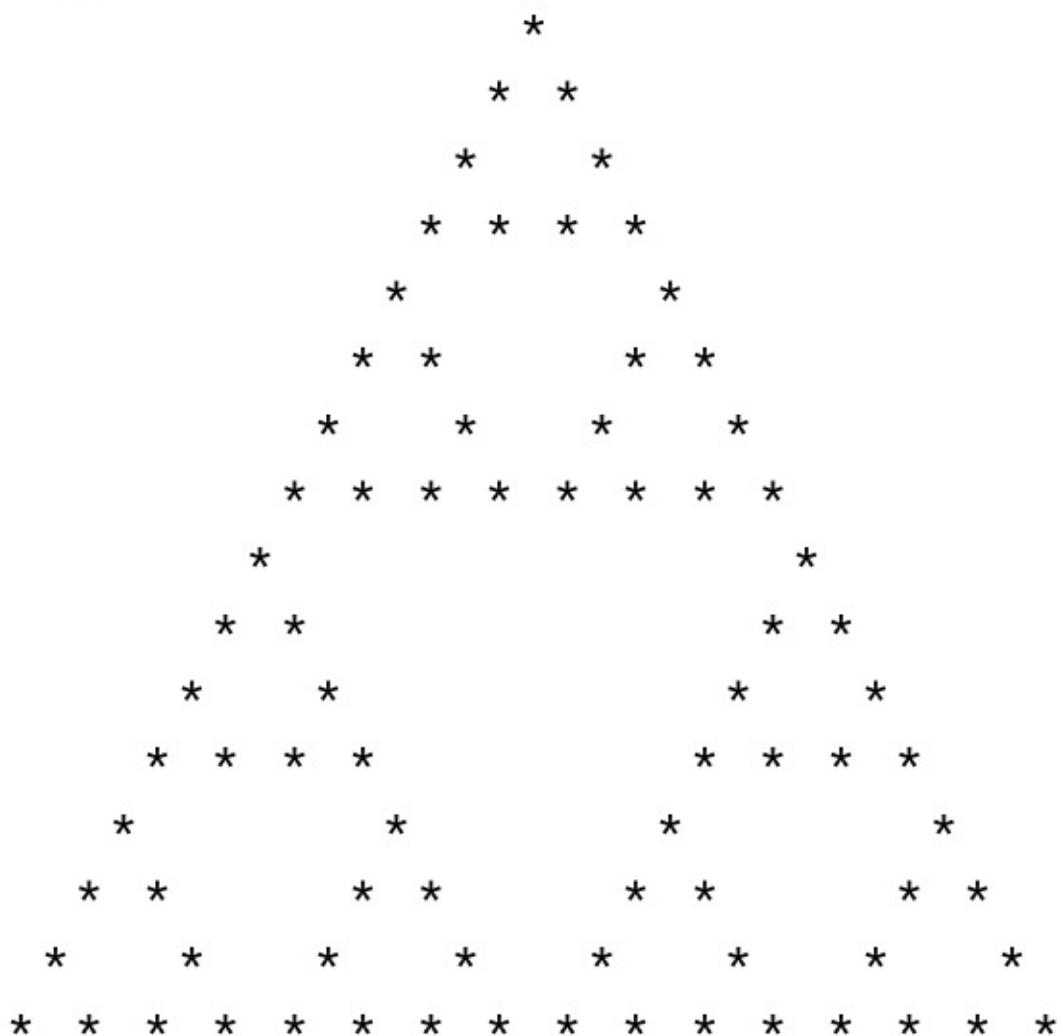
我们的代码将能够计算任何256个基本细胞自动机，例如：规则90 (Wolfram代码)：

```
$ ./bin/automata 40 15 90
```

length: 40

number of steps: 15

rule: 90



我们示例代码项目的结构如下：

```

1. .
2. |--- CMakeLists.txt
3. |--- external
4. |   |--- CMakeLists.txt
5. |   |--- conversion.cpp
6. |   |--- conversion.hpp
7. |   |--- README.md
8. |--- src
9. |   |--- CMakeLists.txt

```

```

10. |   └── evolution
11. |     |   └── CMakeLists.txt
12. |     |   └── evolution.cpp
13. |     |   └── evolution.hpp
14. |   └── initial
15. |     |   └── CMakeLists.txt
16. |     |   └── initial.cpp
17. |     |   └── initial.hpp
18. |   └── io
19. |     |   └── CMakeLists.txt
20. |     |   └── io.cpp
21. |     |   └── io.hpp
22. |   └── main.cpp
23. |   └── parser
24. |     |   └── CMakeLists.txt
25. |     |   └── parser.cpp
26. |     |   └── parser.hpp
27. └── tests
28.   └── catch.hpp
29.   └── CMakeLists.txt
30.   └── test.cpp

```

我们将代码分成许多库来模拟真实的大中型项目，可以将源代码组织到库中，然后将库链接到可执行文件中。

主要功能在 `src/main.cpp` 中：

```

1. #include "conversion.hpp"
2. #include "evolution.hpp"
3. #include "initial.hpp"
4. #include "io.hpp"
5. #include "parser.hpp"
6.
7. #include <iostream>
8.
9. int main(int argc, char *argv[]) {
10.   // parse arguments
11.   int length, num_steps, rule_decimal;
12.   std::tie(length, num_steps, rule_decimal) = parse_arguments(argc, argv);
13.
14.   // print information about parameters
15.   std::cout << "length: " << length << std::endl;

```

```

16.     std::cout << "number of steps: " << num_steps << std::endl;
17.     std::cout << "rule: " << rule_decimal << std::endl;
18.
19.     // obtain binary representation for the rule
20.     std::string rule_binary = binary_representation(rule_decimal);
21.
22.     // create initial distribution
23.     std::vector<int> row = initial_distribution(length);
24.
25.     // print initial configuration
26.     print_row(row);
27.
28.     // the system evolves, print each step
29.     for (int step = 0; step < num_steps; step++) {
30.         row = evolve(row, rule_binary);
31.         print_row(row);
32.     }
33. }
```

`external/conversion.cpp` 文件包含要从十进制转换为二进制的代码。

我们在这里模拟这段代码是由 `src` 外部的“外部”库提供的：

```

1. #include "conversion.hpp"
2. #include <bitset>
3. #include <string>
4. std::string binary_representation(const int decimal) {
5.     return std::bitset<8>(decimal).to_string();
6. }
```

`src/evolution/evolution.cpp` 文件为一个时限传播系统：

```

1. #include "evolution.hpp"
2.
3. #include <string>
4. #include <vector>
5.
6. std::vector<int> evolve(const std::vector<int> row, const std::string
7. rule_binary) {
8.     std::vector<int> result;
9.     for (auto i = 0; i < row.size(); ++i) {
```

```

10.     auto left = (i == 0 ? row.size() : i) - 1;
11.     auto center = i;
12.     auto right = (i + 1) % row.size();
13.     auto ancestors = 4 * row[left] + 2 * row[center] + 1 * row[right];
14.     ancestors = 7 - ancestors;
15.     auto new_state = std::stoi(rule_binary.substr(ancestors, 1));
16.     result.push_back(new_state);
17. }
18. return result;
19. }
```

`src/initial/initial.cpp` 文件，对出进行初始化：

```

1. #include "initial.hpp"
2.
3. #include <vector>
4.
5. std::vector<int> initial_distribution(const int length) {
6.
7.     // we start with a vector which is zeroed out
8.     std::vector<int> result(length, 0);
9.
10.    // more or less in the middle we place a living cell
11.    result[length / 2] = 1;
12.
13.    return result;
14. }
```

`src/io/io.cpp` 文件包含一个函数输出打印行：

```

1. #include "io.hpp"
2. #include <algorithm>
3. #include <iostream>
4. #include <vector>
5. void print_row(const std::vector<int> row) {
6.     std::for_each(row.begin(), row.end(), [](int const &value) {
7.         std::cout << (value == 1 ? '*' : ' ');
8.     });
9.     std::cout << std::endl;
10. }
```

`src/parser/parser.cpp` 文件解析命令行输入：

```

1. #include "parser.hpp"
2.
3. #include <cassert>
4. #include <string>
5. #include <tuple>
6.
7. std::tuple<int, int, int> parse_arguments(int argc, char *argv[]) {
8.     assert(argc == 4 && "program called with wrong number of arguments");
9.
10.    auto length = std::stoi(argv[1]);
11.    auto num_steps = std::stoi(argv[2]);
12.    auto rule_decimal = std::stoi(argv[3]);
13.
14.    return std::make_tuple(length, num_steps, rule_decimal);
15. }
```

最后，`tests/test.cpp` 包含两个使用Catch2库的单元测试：

```

1. #include "evolution.hpp"
2.
3. // this tells catch to provide a main()
4. // only do this in one cpp file
5. #define CATCH_CONFIG_MAIN
6. #include "catch.hpp"
7.
8. #include <string>
9. #include <vector>
10.
11. TEST_CASE("Apply rule 90", "[rule-90]") {
12.     std::vector<int> row = {0, 1, 0, 1, 0, 1, 0, 1, 0};
13.     std::string rule = "01011010";
14.     std::vector<int> expected_result = {1, 0, 0, 0, 0, 0, 0, 0, 1};
15.     REQUIRE(evolve(row, rule) == expected_result);
16. }
17.
18. TEST_CASE("Apply rule 222", "[rule-222]") {
19.     std::vector<int> row = {0, 0, 0, 0, 1, 0, 0, 0, 0};
20.     std::string rule = "11011110";
21.     std::vector<int> expected_result = {0, 0, 0, 1, 1, 1, 0, 0, 0};
22.     REQUIRE(evolve(row, rule) == expected_result);
23. }
```

相应的头文件包含函数声明。有人可能会说，对于这个小代码示例，项目包含了太多子目录。请注意，这只是一个项目的简化示例，通常包含每个库的许多源文件，理想情况下，这些文件被放在到单独的目录中。

具体实施

让我们来详细解释一下CMake所需的功能：

1. `CMakeLists.txt` 顶部非常类似于第1节，代码重用与函数和宏：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-07 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
8.
9. include(GNUInstallDirs)
10. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
11. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
12. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
13. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
14. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
16.
17. # defines targets and sources
18. add_subdirectory(src)
19.
20. # contains an "external" library we will link to
21. add_subdirectory(external)
22.
23. # enable testing and define tests
24. enable_testing()
25. add_subdirectory(tests)

```

2. 目标和源在 `src/CMakeLists.txt` 中定义(转换目标除外)：

```

1. add_executable(automata main.cpp)
2.
3. add_subdirectory(evolution)
4. add_subdirectory(initial)

```

```

5. add_subdirectory(io)
6. add_subdirectory(parser)
7.
8. target_link_libraries(automata
9.   PRIVATE
10.  conversion
11.  evolution
12.  initial
13.  io
14.  parser
15. )

```

3. 转换库在 `external/CMakeLists.txt` 中定义：

```

1. add_library(conversion "")
2.
3. target_sources(conversion
4.   PRIVATE
5.   ${CMAKE_CURRENT_LIST_DIR}/conversion.cpp
6.   PUBLIC
7.   ${CMAKE_CURRENT_LIST_DIR}/conversion.hpp
8. )
9.
10. target_include_directories(conversion
11.   PUBLIC
12.   ${CMAKE_CURRENT_LIST_DIR}
13. )

```

4. `src/CMakeLists.txt` 文件添加了更多的子目录，这些子目录又包含 `CMakeLists.txt` 文件。`src/evolution/CMakeLists.txt` 包含以下内容：

```

1. add_library(evolution "")
2.
3. target_sources(evolution
4.   PRIVATE
5.   evolution.cpp
6.   PUBLIC
7.   ${CMAKE_CURRENT_LIST_DIR}/evolution.hpp
8. )
9.
10. target_include_directories(evolution
11.   PUBLIC

```

```

12.      ${CMAKE_CURRENT_LIST_DIR}
13.  )

```

5. 单元测试在 `tests/CMakeLists.txt` 中注册：

```

1. add_executable(cpp_test test.cpp)
2.
3. target_link_libraries(cpp_test evolution)
4.
5. add_test(
6.   NAME
7.   test_evolution
8.   COMMAND
9.   ${TARGET_FILE:cpp_test}
10. )

```

6. 配置和构建项目产生以下输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5.
6. Scanning dependencies of target conversion
  [ 7%] Building CXX object
7. external/CMakeFiles/conversion.dir/conversion.cpp.o
8. [ 14%] Linking CXX static library ../lib64/libconversion.a
9. [ 14%] Built target conversion
10. Scanning dependencies of target evolution
    [ 21%] Building CXX object
11. src/evolution/CMakeFiles/evolution.dir/evolution.cpp.o
12. [ 28%] Linking CXX static library ../../lib64/libevolution.a
13. [ 28%] Built target evolution
14. Scanning dependencies of target initial
    [ 35%] Building CXX object
15. src/initial/CMakeFiles/initial.dir/initial.cpp.o
16. [ 42%] Linking CXX static library ../../lib64/libinitial.a
17. [ 42%] Built target initial
18. Scanning dependencies of target io
19. [ 50%] Building CXX object src/io/CMakeFiles/io.dir/io.cpp.o
20. [ 57%] Linking CXX static library ../../lib64/libio.a
21. [ 57%] Built target io

```

```

22. Scanning dependencies of target parser
23. [ 64%] Building CXX object src/parser/CMakeFiles/parser.dir/parser.cpp.o
24. [ 71%] Linking CXX static library ../../lib64/libparser.a
25. [ 71%] Built target parser
26. Scanning dependencies of target automata
27. [ 78%] Building CXX object src/CMakeFiles/automata.dir/main.cpp.o
28. [ 85%] Linking CXX executable ../../bin/automata
29. [ 85%] Built target automata
30. Scanning dependencies of target cpp_test
31. [ 92%] Building CXX object tests/CMakeFiles/cpp_test.dir/test.cpp.o
32. [100%] Linking CXX executable ../../bin/cpp_test
33. [100%] Built target cpp_test

```

7. 最后，运行单元测试：

```

1. $ ctest
2.
3. Running tests...
4. Start 1: test_evolution
5. 1/1 Test #1: test_evolution ..... Passed 0.00 sec
6. 100% tests passed, 0 tests failed out of 1

```

工作原理

我们可以将所有代码放到一个源文件中。不过，每次编辑都需要重新编译。将源文件分割成更小、更易于管理的单元是有意义的。可以将所有源代码都编译成一个库或可执行文件。实际上，项目更喜欢将源代码编译分成更小的、定义良好的库。这样做既是为了本地化和简化依赖项，也是为了简化代码维护。这意味着如在这里所做的那样，由许多库构建一个项目是一种常见的情况。

为了讨论CMake结构，我们可以从定义每个库的单个CMakeLists.txt文件开始，自底向上进行，例如 `src/evolution/CMakeLists.txt`：

```

1. add_library(evolution "")
2.
3. target_sources(evolution
4.   PRIVATE
5.     evolution.cpp
6.   PUBLIC
7.     ${CMAKE_CURRENT_LIST_DIR}/evolution.hpp
8. )
9.

```

```

10. target_include_directories(evolution
11.   PUBLIC
12.   ${CMAKE_CURRENT_LIST_DIR}
13. )

```

这些单独的 `CMakeLists.txt` 文件定义了库。本例中，我们首先使用 `add_library` 定义库名，然后定义它的源和包含目录，以及它们的目标可见性：实现文件(`evolution.cpp` : `PRIVATE`)，而接口头文件 `evolution.hpp` 定义为 `PUBLIC`，因为我们将在 `main.cpp` 和 `test.cpp` 中访问它。定义尽可能接近代码目标的好处是，对于该库的修改，只需要变更该目录中的文件即可；换句话说，也就是库依赖项被封装。

向上移动一层，库在 `src/CMakeLists.txt` 中封装：

```

1. add_executable(automata main.cpp)
2.
3. add_subdirectory(evolution)
4. add_subdirectory(initial)
5. add_subdirectory(io)
6. add_subdirectory(parser)
7.
8. target_link_libraries(automata
9.   PRIVATE
10.  conversion
11.  evolution
12.  initial
13.  io
14.  parser
15. )

```

文件在主 `CMakeLists.txt` 中被引用。这意味着使用 `CMakeLists.txt` 文件，构建我们的项目。这种方法对于许多项目来说是可用的，并且它可以扩展到更大型的项目，而不需要在目录间的全局变量中包含源文件列表。`add_subdirectory` 方法的另一个好处是它隔离了作用范围，因为子目录中定义的变量在父范围中不能访问。

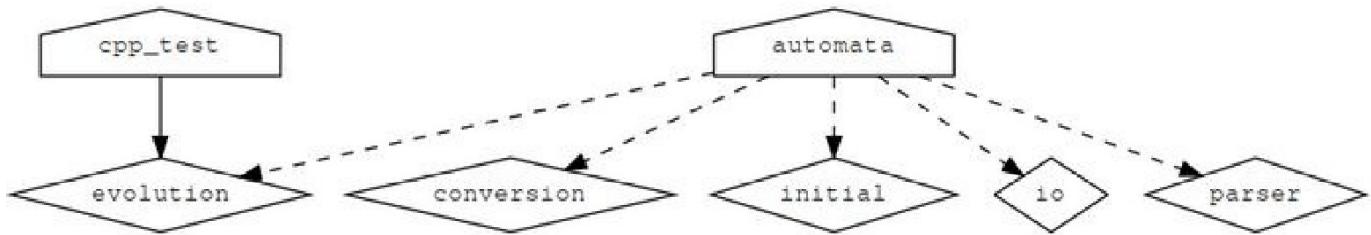
更多信息

使用 `add_subdirectory` 调用树构建项目的一个限制是，CMake不允许将 `target_link_libraries` 与定义在当前目录范围之外的目标一起使用。对于本示例来说，这不是问题。在下一个示例中，我们将演示另一种方法，我们不使用 `add_subdirectory`，而是使用 `module include` 来组装不同的 `CMakeLists.txt` 文件，它允许我们链接到当前目录之外定义的目标。

CMake可以使用Graphviz图形可视化软件(<http://www.graphviz.org>)生成项目的依赖关系图：

```
1. $ cd build
2. $ cmake --graphviz=example.dot ...
3. $ dot -T png example.dot -o example.png
```

生成的图表将显示不同目录下的目标之间的依赖关系：



本书中，我们一直在构建源代码之外的代码，以保持源代码树和构建树是分开的。这是推荐的方式，允许我们使用相同的源代码配置不同的构建(顺序的或并行的，Debug或Release)，而不需要复制源代码，也不需要在源代码树中生成目标文件。使用以下代码片段，可以保护您的项目免受内部构建的影响：

```
1. if(${PROJECT_SOURCE_DIR} STREQUAL ${PROJECT_BINARY_DIR})
   message(FATAL_ERROR "In-source builds not allowed. Please make a new
2. directory (called a build directory) and run CMake from there.")
3. endif()
```

认识到构建结构与源结构类似很有用。示例中，将 `message` 打印输出插入到 `src/CMakeLists.txt` 中：

```
1. message("current binary dir is ${CMAKE_CURRENT_BINARY_DIR}")
```

在 `build` 下构建项目时，我们将看到 `build/src` 的打印输出。

在CMake的3.12版本中，`OBJECT` 库是组织大型项目的另一种可行方法。对我们的示例的惟一修改是在库的 `CMakeLists.txt` 中。源文件将被编译成目标文件：既不存档到静态库中，也不链接到动态库中。例如：

```
1. add_library(io OBJECT "")
2.
3. target_sources(io
4.   PRIVATE
5.   io.cpp)
```

```
6. PUBLIC
7.     ${CMAKE_CURRENT_LIST_DIR}/io.hpp
8. )
9.
10. target_include_directories(io
11. PUBLIC
12.     ${CMAKE_CURRENT_LIST_DIR}
13. )
```

主 `CMakeLists.txt` 保持不变： `automata` 可执行目标将这些目标文件链接到最终的可执行文件。使用也有要求需求，例如：在对象库上设置的目录、编译标志和链接库，将被正确地继承。有关CMake 3.12中引入的对象库新特性的更多细节，请参考官方文档：
<https://cmake.org/cmake/help/v3.12/manual/cmake-buildsystem.7.html#object-libraries>

7.8 使用target_sources避免全局变量

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-08> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例中，我们将讨论前一个示例的另一种方法，并不使用 `add_subdirectory` 的情况下，使用 `module include` 组装不同的CMakeLists.txt文件。这种方法的灵感来自 https://crascit.com/2016/01/31/enhance-sours-file-handling-with-target_sources/，其允许我们使用 `target_link_libraries` 链接到当前目录之外定义的目标。

准备工作

将使用与前一个示例相同的源代码。惟一的更改将出现在 `CMakeLists.txt` 文件中，我们将在下面的部分中讨论这些更改。

具体实施

1. 主 `CMakeLists.txt` 包含以下内容：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-08 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
8.
9. include(GNUInstallDirs)
10. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
11. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
12. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
13. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
14. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
16.
17. # defines targets and sources
18. include(src/CMakeLists.txt)
19. include(external/CMakeLists.txt)
```

```

20.
21. enable_testing()
22. add_subdirectory(tests)

```

2. 与前一个示例相比，`external/CMakeLists.txt` 文件没有变化。

3. `src/CMakeLists.txt` 文件定义了两个库(automaton和evolution)：

```

1. add_library(automaton "")
2. add_library(evolution "")
3.
4. include(${CMAKE_CURRENT_LIST_DIR}/evolution/CMakeLists.txt)
5. include(${CMAKE_CURRENT_LIST_DIR}/initial/CMakeLists.txt)
6. include(${CMAKE_CURRENT_LIST_DIR}/io/CMakeLists.txt)
7. include(${CMAKE_CURRENT_LIST_DIR}/parser/CMakeLists.txt)
8.
9. add_executable(automata "")
10.
11. target_sources(automata
12.   PRIVATE
13.     ${CMAKE_CURRENT_LIST_DIR}/main.cpp
14.   )
15.
16. target_link_libraries(automata
17.   PRIVATE
18.     automaton
19.     conversion
20.   )

```

4. `src/evolution/CMakeLists.txt` 文件包含以下内容：

```

1. target_sources(automaton
2.   PRIVATE
3.     ${CMAKE_CURRENT_LIST_DIR}/evolution.cpp
4.   PUBLIC
5.     ${CMAKE_CURRENT_LIST_DIR}/evolution.hpp
6.   )
7.
8. target_include_directories(automaton
9.   PUBLIC
10.    ${CMAKE_CURRENT_LIST_DIR}
11.   )

```

```

12.
13. target_sources(evolution
14.   PRIVATE
15.     ${CMAKE_CURRENT_LIST_DIR}/evolution.cpp
16.   PUBLIC
17.     ${CMAKE_CURRENT_LIST_DIR}/evolution.hpp
18. )
19.
20. target_include_directories(evolution
21.   PUBLIC
22.     ${CMAKE_CURRENT_LIST_DIR}
23. )

```

5. 其余 `CMakeLists.txt` 文件和 `src/initial/CMakeLists.txt` 相同：

```

1. target_sources(automaton
2.   PRIVATE
3.     ${CMAKE_CURRENT_LIST_DIR}/initial.cpp
4.   PUBLIC
5.     ${CMAKE_CURRENT_LIST_DIR}/initial.hpp
6. )
7.
8. target_include_directories(automaton
9.   PUBLIC
10.    ${CMAKE_CURRENT_LIST_DIR}
11. )

```

6. 配置、构建和测试的结果与前面的方法相同：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build build
5. $ ctest
6.
7. Running tests...
8. Start 1: test_evolution
9. 1/1 Test #1: test_evolution ..... Passed 0.00 sec
10. 100% tests passed, 0 tests failed out of 1

```

工作原理

与之前的示例不同，我们定义了三个库：

- conversion(在external定义)
- automaton(包含除转换之外的所有源)
- evolution(在 `src/evolution` 中定义，并通过 `cpp_test` 链接)

本例中，通过使用 `include()` 引用 `CMakeLists.txt` 文件，我们在父范围内，仍然能保持所有目标可用：

```
1. include(src/CMakeLists.txt)
2. include(external/CMakeLists.txt)
```

我们可以构建一个包含树，记住当进入子目录(`src/CMakeLists.txt`)时，我们需要使用相对于父范围的路径：

```
1. include(${CMAKE_CURRENT_LIST_DIR}/evolution/CMakeLists.txt)
2. include(${CMAKE_CURRENT_LIST_DIR}/initial/CMakeLists.txt)
3. include(${CMAKE_CURRENT_LIST_DIR}/io/CMakeLists.txt)
4. include(${CMAKE_CURRENT_LIST_DIR}/parser/CMakeLists.txt)
```

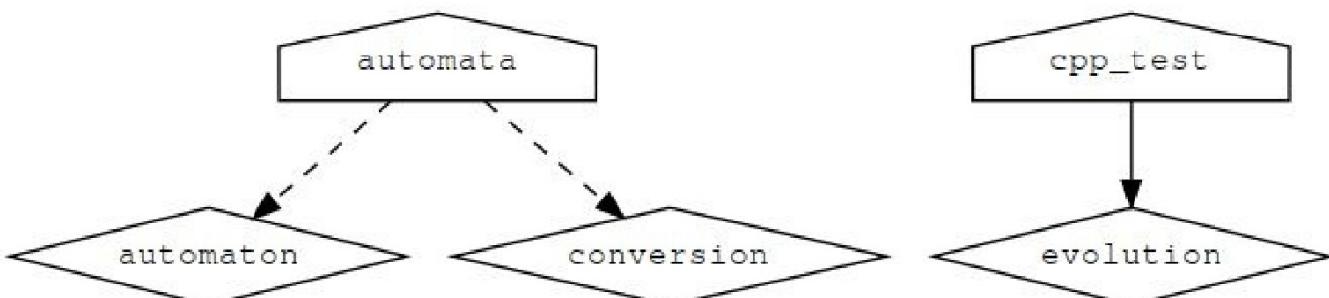
这样，我们就可以定义并链接到通过 `include()` 语句访问文件树中任何位置的目标。但是，我们应该选择在对维护人员和代码贡献者容易看到的地方，去定义它们。

更多信息

我们可以再次使用CMake和Graphviz (<http://www.graphviz.org/>)生成这个项目的依赖关系图：

```
1. $ cd build
2. $ cmake -Dgraphviz=example.dot ..
3. $ dot -T png example.dot -o example.png
```

对于当前设置，我们得到如下依赖关系图：



7.9 组织Fortran项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-7/recipe-09> 中找到，其中有一个Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

我们来讨论如何构造和组织Fortran项目，原因有二：

1. 现在，仍然有很多Fortran项目，特别是在数字软件中(有关通用Fortran软件项目的更全面列表，请参见<http://fortranwiki.org/fortran/show/Libraries>)。
2. 对于不使用CMake的项目，Fortran 90(以及更高版本)可能更难构建，因为Fortran模块强制执行编译顺序。换句话说，对于手工编写的Makefile，通常需要为Fortran模块文件编写依赖扫描程序。

正如我们在本示例中所示，现代CMake允许我们以非常紧凑和模块化的方式配置和构建项目。作为一个例子，我们将使用前两个示例中的基本元胞自动机，现在将其移植到Fortran。

准备工作

文件树结构与前两个示例非常相似。我们用Fortran源代码替换了C++，现在就没有头文件了：

```

1. .
2. |   CMakeLists.txt
3. |   external
4. |     |   CMakeLists.txt
5. |     |   conversion.f90
6. |     |   README.md
7. |   src
8. |     |   CMakeLists.txt
9. |     |   evolution
10. |       |   ancestors.f90
11. |       |   CMakeLists.txt
12. |       |   empty.f90
13. |       |   evolution.f90
14. |       |   initial
15. |       |   CMakeLists.txt
16. |       |   initial.f90
17. |       |   io
18. |       |   CMakeLists.txt
19. |       |   io.f90

```

```

20. |   └── main.f90
21. |   └── parser
22. |     ├── CMakeLists.txt
23. |     └── parser.f90
24. └── tests
25.   ├── CMakeLists.txt
26.   └── test.f90

```

主程序在 `src/main.f90` 中：

```

1. program example
2.
3. use parser, only: get_arg_as_int
4. use conversion, only: binary_representation
5. use initial, only: initial_distribution
6. use io, only: print_row
7. use evolution, only: evolve
8.
9. implicit none
10.
11. integer :: num_steps
12. integer :: length
13. integer :: rule_decimal
14. integer :: rule_binary(8)
15. integer, allocatable :: row(:)
16. integer :: step
17.
18. ! parse arguments
19. num_steps = get_arg_as_int(1)
20. length = get_arg_as_int(2)
21. rule_decimal = get_arg_as_int(3)
22.
23. ! print information about parameters
24. print *, "number of steps: ", num_steps
25. print *, "length: ", length
26. print *, "rule: ", rule_decimal
27.
28. ! obtain binary representation for the rule
29. rule_binary = binary_representation(rule_decimal)
30.
31. ! create initial distribution
32. allocate(row(length))

```

```

33.   call initial_distribution(row)
34.
35.   ! print initial configuration
36.   call print_row(row)
37.
38.   ! the system evolves, print each step
39.   do step = 1, num_steps
40.     call evolve(row, rule_binary)
41.     call print_row(row)
42.   end do
43.
44.   deallocate(row)
45. end program

```

与前面的示例一样，我们已经将conversion模块放入 `external/conversion.f90` 中：

```

1. module conversion
2.
3. implicit none
4. public binary_representation
5. private
6.
7. contains
8.
9. pure function binary_representation(n_decimal)
10.    integer, intent(in) :: n_decimal
11.    integer :: binary_representation(8)
12.    integer :: pos
13.    integer :: n
14.
15.    binary_representation = 0
16.    pos = 8
17.    n = n_decimal
18.    do while (n > 0)
19.      binary_representation(pos) = mod(n, 2)
20.      n = (n - binary_representation(pos))/2
21.      pos = pos - 1
22.    end do
23.  end function
24.
25. end module

```

evolution库分成三个文件，大部分在 `src/evolution/evolution.f90` 中：

```
1. module evolution
2.
3.   implicit none
4.   public evolve
5.   private
6.
7. contains
8.
9.   subroutine not_visible()
10.    ! no-op call to demonstrate private/public visibility
11.    call empty_subroutine_no_interface()
12. end subroutine
13.
14. pure subroutine evolve(row, rule_binary)
15.   use ancestors, only: compute_ancestors
16.
17.   integer, intent(inout) :: row(:)
18.   integer, intent(in) :: rule_binary(8)
19.   integer :: i
20.   integer :: left, center, right
21.   integer :: ancestry
22.   integer, allocatable :: new_row(:)
23.
24.   allocate(new_row(size(row)))
25.
26.   do i = 1, size(row)
27.     left = i - 1
28.     center = i
29.     right = i + 1
30.
31.     if (left < 1) left = left + size(row)
32.     if (right > size(row)) right = right - size(row)
33.
34.     ancestry = compute_ancestors(row, left, center, right)
35.     new_row(i) = rule_binary(ancestry)
36.   end do
37.
38.   row = new_row
39.   deallocate(new_row)
40.
```

```

41. end subroutine
42.
43. end module

```

祖先计算是在 `src/evolution/ancestors.f90` :

```

1. module ancestors
2.
3. implicit none
4. public compute_ancestors
5. private
6.
7. contains
8. pure integer function compute_ancestors(row, left, center, right) result(i)
9.     integer, intent(in) :: row(:)
10.    integer, intent(in) :: left, center, right
11.
12.    i = 4*row(left) + 2*row(center) + 1*row(right)
13.    i = 8 - i
14. end function
15. end module

```

还有一个“空”模块在 `src/evolution/empty.f90` 中:

```

1. module empty
2.
3. implicit none
4. public empty_subroutine
5. private
6.
7. contains
8.
9. subroutine empty_subroutine()
10. end subroutine
11.
12. end module
13.
14. subroutine
15. empty_subroutine_no_interface()
16. use empty, only: empty_subroutine
17. call empty_subroutine()
18. end subroutine

```

启动条件的代码位于 `src/initial/initial.f90` :

```
1. module initial
2.
3.   implicit none
4.   public initial_distribution
5.   private
6.
7. contains
8.
9.   pure subroutine initial_distribution(row)
10.    integer, intent(out) :: row(:)
11.
12.    row = 0
13.    row(size(row)/2) = 1
14.  end subroutine
15.
16. end module
```

`src/io/io.f90` 包含一个打印输出:

```
1. module io
2.
3.   implicit none
4.   public print_row
5.   private
6.
7. contains
8.
9.   subroutine print_row(row)
10.    integer, intent(in) :: row(:)
11.    character(size(row)) :: line
12.    integer :: i
13.
14.    do i = 1, size(row)
15.      if (row(i) == 1) then
16.        line(i:i) = '*'
17.      else
18.        line(i:i) = ' '
19.      end if
20.    end do
```

```

21.
22.     print *, line
23. end subroutine
24.
25. end module

```

`src/parser/parser.f90` 用于解析命令行参数：

```

1. module parser
2.
3. implicit none
4. public get_arg_as_int
5. private
6.
7. contains
8.
9. integer function get_arg_as_int(n) result(i)
10.    integer, intent(in) :: n
11.    character(len=32) :: arg
12.
13.    call get_command_argument(n, arg)
14.    read(arg, *) i
15. end function
16. end module

```

最后，使用 `tests/test.f90` 对上面的实现进行测试：

```

1. program test
2.
3. use evolution, only: evolve
4.
5. implicit none
6.
7. integer :: row(9)
8. integer :: expected_result(9)
9. integer :: rule_binary(8)
10. integer :: i
11.
12. ! test rule 90
13. row = (/0, 1, 0, 1, 0, 1, 0, 1, 0/)
14. rule_binary = (/0, 1, 0, 1, 1, 0, 1, 0/)
15. call evolve(row, rule_binary)

```

```

16.     expected_result = (/1, 0, 0, 0, 0, 0, 0, 0, 1/)
17.     do i = 1, 9
18.         if (row(i) /= expected_result(i)) then
19.             print *, 'ERROR: test for rule 90 failed'
20.             call exit(1)
21.         end if
22.     end do
23.
24. ! test rule 222
25. row = (/0, 0, 0, 0, 1, 0, 0, 0, 0/)
26. rule_binary = (/1, 1, 0, 1, 1, 1, 1, 0/)
27. call evolve(row, rule_binary)
28. expected_result = (/0, 0, 0, 1, 1, 1, 0, 0, 0/)
29. do i = 1, 9
30.     if (row(i) /= expected_result(i)) then
31.         print *, 'ERROR: test for rule 222 failed'
32.         call exit(1)
33.     end if
34. end do
35. end program

```

具体实施

1. 主 `CMakeLists.txt` 类似于第7节，我们只是将CXX换成Fortran，去掉C++11的要求：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-09 LANGUAGES Fortran)
4.
5. include(GNUInstallDirs)
6. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
7. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
8. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
9. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
10. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
11. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
12.
13. # defines targets and sources
14. add_subdirectory(src)
15.
16. # contains an "external" library we will link to

```

```

17. add_subdirectory(external)
18.
19. # enable testing and define tests
20. enable_testing()
21. add_subdirectory(tests)

```

2. 目标和源在 `src/CMakeLists.txt` 中定义(conversion目标除外)：

```

1. add_executable(automata main.f90)
2.
3. add_subdirectory(evolution)
4. add_subdirectory(initial)
5. add_subdirectory(io)
6. add_subdirectory(parser)
7.
8. target_link_libraries(automata
9.   PRIVATE
10.    conversion
11.    evolution
12.    initial
13.    io
14.    parser
15. )

```

3. conversion库在 `external/CMakeLists.txt` 中定义：

```

1. add_library(conversion "")
2.
3. target_sources(conversion
4.   PUBLIC
5.   ${CMAKE_CURRENT_LIST_DIR}/conversion.f90
6. )

```

4. `src/CMakeLists.txt` 文件添加了更多的子目录，这些子目录又包含 `CMakeLists.txt` 文件。它们在结构上都是相似的，例如：`src/initial/CMakeLists.txt` 包含以下内容：

```

1. add_library(initial "")
2.
3. target_sources(initial
4.   PUBLIC
5.   ${CMAKE_CURRENT_LIST_DIR}/initial.f90
6. )

```

5. 有个例外的是 `src/evolution/CMakeLists.txt` 中的evolution库，我们将其分为三个源文件：

```

1. add_library(evolution "")
2.
3. target_sources(evolution
4.   PRIVATE
5.     empty.f90
6.   PUBLIC
7.     ${CMAKE_CURRENT_LIST_DIR}/ancestors.f90
8.     ${CMAKE_CURRENT_LIST_DIR}/evolution.f90
9. )

```

6. 单元测试在 `tests/CMakeLists.txt` 中注册：

```

1. add_executable(fortran_test test.f90)
2.
3. target_link_libraries(fortran_test evolution)
4.
5. add_test(
6.   NAME
7.     test_evolution
8.   COMMAND
9.     ${TARGET_FILE:fortran_test}
10. )

```

7. 配置和构建项目，将产生以下输出：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. Scanning dependencies of target conversion
 [ 4%] Building Fortran object
6. external/CMakeFiles/conversion.dir/conversion.f90.o
7. [ 8%] Linking Fortran static library ../lib64/libconversion.a
8. [ 8%] Built target conversion
9. Scanning dependencies of target evolution
 [ 12%] Building Fortran object
10. src/evolution/CMakeFiles/evolution.dir/ancestors.f90.o

```

```
[ 16%] Building Fortran object
11. src/evolution/CMakeFiles/evolution.dir/empty.f90.o
    [ 20%] Building Fortran object
12. src/evolution/CMakeFiles/evolution.dir/evolution.f90.o
13. [ 25%] Linking Fortran static library ../../lib64/libevolution.a
14. [ 25%] Built target evolution
15. Scanning dependencies of target initial
    [ 29%] Building Fortran object
16. src/initial/CMakeFiles/initial.dir/initial.f90.o
17. [ 33%] Linking Fortran static library ../../lib64/libinitial.a
18. [ 33%] Built target initial
19. Scanning dependencies of target io
20. [ 37%] Building Fortran object src/io/CMakeFiles/io.dir/io.f90.o
21. [ 41%] Linking Fortran static library ../../lib64/libio.a
22. [ 41%] Built target io
23. Scanning dependencies of target parser
    [ 45%] Building Fortran object
24. src/parser/CMakeFiles/parser.dir/parser.f90.o
25. [ 50%] Linking Fortran static library ../../lib64/libparser.a
26. [ 50%] Built target parser
27. Scanning dependencies of target example
    [ 54%] Building Fortran object
28. src/CMakeFiles/example.dir/_/external/conversion.f90.o
    [ 58%] Building Fortran object
29. src/CMakeFiles/example.dir/evolution/ancestors.f90.o
    [ 62%] Building Fortran object
30. src/CMakeFiles/example.dir/evolution/evolution.f90.o
    [ 66%] Building Fortran object
31. src/CMakeFiles/example.dir/initial/initial.f90.o
32. [ 70%] Building Fortran object src/CMakeFiles/example.dir/io/io.f90.o
    [ 75%] Building Fortran object
33. src/CMakeFiles/example.dir/parser/parser.f90.o
34. [ 79%] Building Fortran object src/CMakeFiles/example.dir/main.f90.o
35. [ 83%] Linking Fortran executable ../bin/example
36. [ 83%] Built target example
37. Scanning dependencies of target fortran_test
    [ 87%] Building Fortran object
38. tests/CMakeFiles/fortran_test.dir/_/src/evolution/ancestors.f90.o
    [ 91%] Building Fortran object
39. tests/CMakeFiles/fortran_test.dir/_/src/evolution/evolution.f90.o
    [ 95%] Building Fortran object
40. tests/CMakeFiles/fortran_test.dir/test.f90.o
41. [100%] Linking Fortran executable
```

8. 最后，运行单元测试：

```

1. $ ctest
2.
3. Running tests...
4. Start 1: test_evolution
5. 1/1 Test #1: test_evolution ..... Passed 0.00 sec
6.
7. 100% tests passed, 0 tests failed out of 1

```

工作原理

第7节中使用 `add_subdirectory` 限制范围，将从下往上讨论CMake结构，从定义每个库的单个 `CMakeLists.txt` 文件开始，比如 `src/evolution/CMakeLists.txt`：

```

1. add_library(evolution "")
2. target_sources(evolution
3.   PRIVATE
4.     empty.f90
5.   PUBLIC
6.     ${CMAKE_CURRENT_LIST_DIR}/ancestors.f90
7.     ${CMAKE_CURRENT_LIST_DIR}/evolution.f90
8. )

```

这些独立的 `CMakeLists.txt` 文件定义了源文件的库，遵循与前两个示例相同的方式：开发或维护人员可以对其中文件分而治之。

首先用 `add_library` 定义库名，然后定义它的源和包含目录，以及它们的目标可见性。这种情况下，因为它们的模块接口是在库之外访问，所以 `ancestors.f90` 和 `evolution.f90` 都是 `PUBLIC`，而模块接口 `empty.f90` 不能在文件之外访问，因此将其标记为 `PRIVATE`。

向上移动一层，库在 `src/CMakeLists.txt` 中封装：

```

1. add_executable(automata main.f90)
2.
3. add_subdirectory(evolution)
4. add_subdirectory(initial)
5. add_subdirectory(io)
6. add_subdirectory(parser)
7.
8. target_link_libraries(automata

```

```
9.    PRIVATE
10.   conversion
11.   evolution
12.   initial
13.   io
14.   parser
15. )
```

这个文件在主 `CMakeLists.txt` 中被引用。这意味着我们使用 `CMakeLists.txt` 文件(使用 `add_subdirectory` 添加)构建项目。正如第7节中讨论的，使用 `add_subdirectory` 限制范围，这种方法可以扩展到更大型的项目，而不需要在多个目录之间的全局变量中携带源文件列表，还可以隔离范围和名称空间。

将这个Fortran示例与C++版本(第7节)进行比较，我们可以注意到，在Fortran的情况下，相对的CMake工作量比较小；我们不需要使用 `target_include_directory`，因为没有头文件，接口是通过生成的Fortran模块文件进行通信。另外，我们既不需要担心 `target_sources` 中列出的源文件的顺序，也不需要在库之间强制执行任何显式依赖关系。CMake能够从源文件依赖项推断Fortran模块依赖项。使用 `target_sources` 与 `PRIVATE` 和 `PUBLIC` 资源结合使用，以紧凑和健壮的方式表示接口。

更多信息

这个示例中，我们没有指定应该放置Fortran模块文件的目录，并且保持了这个透明。模块文件的位置可以通过设置 `CMAKE_Fortran_MODULE_DIRECTORY` 变量来指定。注意，也可以将其设置为 `Fortran_MODULE_DIRECTORY`，从而实现更好的控制。详细可见：https://cmake.org/cmake/help/v3.5/prop_tgt/Fortran_MODULE_DIRECTORY.html

第8章 超级构建模式

本章的主要内容如下：

- 使用超级级构建模式
- 使用超级构建管理依赖项: I . Boost库
- 使用超级构建管理依赖项: II . FFTW库
- 使用超级构建管理依赖项: III . Google Test框架
- 使用超级构建支持项目

每个项目都需要处理依赖关系，使用CMake很容易查询这些依赖关系，是否存在于配置项目中。第3章，展示了如何找到安装在系统上的依赖项，到目前为止我们一直使用这种模式。但是，当不能满足依赖关系，我们只能使配置失败，并向用户警告失败的原因。然而，使用CMake可以组织我们的项目，如果在系统上找不到依赖项，就可以自动获取和构建依赖项。本章将介绍和分析 `ExternalProject.cmake` 和 `FetchContent.cmake` 标准模块，及在超级构建模式中的使用。前者允许在构建时检索项目的依赖项，后者允许我们在配置时检索依赖项(CMake的3.11版本后添加)。使用超级构建模式，我们可以利用CMake作为包管理器：相同的项目中，将以相同的方式处理依赖项，无论依赖项在系统上是已经可用，还是需要重新构建。接下来的5个示例，将带您了解该模式，并展示如何使用它来获取和构建依赖关系。

NOTE:这两个模块都有大量的在线文档。`ExternalProject.cmake`，可以参考<https://cmake.org/cmake/help/v3.5/module/ExternalProject.html>。
。 `FetchContent.cmake`，可以参考<https://cmake.org/cmake/help/v3.11/module/FetchContent.html>。

8.1 使用超级构建模式

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-8/recipe-01> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例通过一个简单示例，介绍超级构建模式。我们将展示如何使用 `ExternalProject_Add` 命令来构建一个的“Hello, World”程序。

准备工作

本示例将从以下源代码(`Hello-World.cpp`)构建“Hello, World”可执行文件：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4.
5.     std::string say_hello() { return std::string("Hello, CMake superbuild world!"); }
6.
7. int main()
8. {
9.     std::cout << say_hello() << std::endl;
10.    return EXIT_SUCCESS;
11. }
```

项目结构如下：

```

1. .
2. └── CMakeLists.txt
3. └── src
4.     ├── CMakeLists.txt
5.     └── hello-world.cpp
```

具体实施

让我们看一下根目录下的CMakeLists.txt：

1. 声明一个C++11项目，以及CMake最低版本：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-01 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 为当前目录和底层目录设置 `EP_BASE` 目录属性：

```
1. set_property(DIRECTORY PROPERTY EP_BASE ${CMAKE_BINARY_DIR}/subprojects)
```

3. 包括 `ExternalProject.cmake` 标准模块。该模块提供了 `ExternalProject_Add` 函数：

```
1. include(ExternalProject)
```

4. “Hello, World”源代码通过调用 `ExternalProject_Add` 函数作为外部项目添加的。外部项目的名称为 `recipe-01_core`：

```
1. ExternalProject_Add(${PROJECT_NAME}_core
```

5. 使用 `SOURCE_DIR` 选项为外部项目设置源目录：

```

1. SOURCE_DIR
2. ${CMAKE_CURRENT_LIST_DIR}/src

```

6. `src` 子目录包含一个完整的CMake项目。为了配置和构建它，通过 `CMAKE_ARGS` 选项将适当的CMake选项传递给外部项目。例子中，只需要通过C++编译器和C++标准的要求即可：

```

1. CMAKE_ARGS
2. -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
3. -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
4. -DCMAKE_CXX_EXTENSIONS=${CMAKE_CXX_EXTENSIONS}
5. -DCMAKE_CXX_STANDARD_REQUIRED=${CMAKE_CXX_STANDARD_REQUIRED}

```

7. 我们还设置了C++编译器标志。这些通过使用 `CMAKE_CACHE_ARGS` 选项传递到 `ExternalProject_Add` 中：

```

1. CMAKE_CACHE_ARGS
2. -DCMAKE_CXX_FLAGS:STRING=${CMAKE_CXX_FLAGS}

```

8. 我们配置外部项目，使它进行构建：

```
1. BUILD_ALWAYS
2. 1
```

9. 安装步骤不会执行任何操作(我们将在第4节中重新讨论安装，在第10章中安装超级构建，并编写安装程序)：

```
1. INSTALL_COMMAND
2. ""
3. )
```

现在，我们来看看 `src/CMakeLists.txt`。由于我们将“Hello, World”源文件作为一个外部项目添加，这是一个独立项目的 `CMakeLists.txt` 文件：

1. 这里声明CMake版本最低要求：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
```

2. 声明一个C++项目：

```
1. project(recipe-01_core LANGUAGES CXX)
```

3. 最终，使用 `hello-world.cpp` 源码文件生成可执行目标 `hello-world`：

```
1. add_executable(hello-world hello-world.cpp)
```

配置构建项目：

```
1. $ mkdir -p build
2. $ cmake ..
3. $ cmake --build .
```

构建目录的结构稍微复杂一些，`subprojects` 文件夹的内容如下：

```
1. build/subprojects/
2.   └── Build
3.     └── recipe-01_core
4.       ├── CMakeCache.txt
5.       ├── CMakeFiles
6.       └── cmake_install.cmake
```

```

7. |     └─ hello-world
8. |         └─ Makefile
9. └─ Download
10.    └─ recipe-01_core
11. └─ Install
12.    └─ recipe-01_core
13. └─ Stamp
14.    └─ recipe-01_core
15.        └─ recipe-01_core-configure
16.        └─ recipe-01_core-done
17.        └─ recipe-01_core-download
18.        └─ recipe-01_core-install
19.        └─ recipe-01_core-mkdir
20.        └─ recipe-01_core-patch
21.        └─ recipe-01_core-update
22. └─ tmp
23.    └─ recipe-01_core
24.        └─ recipe-01_core-cache-.cmake
25.        └─ recipe-01_core-cfgcmd.txt
26.        └─ recipe-01_core-cfgcmd.txt.in

```

`recipe-01_core` 已经构建到 `build/subprojects` 子目录中，称为 `Build/recipe-01_core` (这是我们设置的 `EP_BASE`)。

`hello-world` 可执行文件在 `Build/recipe-01_core` 下创建，其他子文件夹 `tmp/recipe-01_core` 和 `Stamp/recipe-01_core` 包含临时文件，比如：CMake缓存脚本 `recipe-01_core-cache-.cmake` 和已执行的外部构建项目的各步骤的时间戳文件。

工作原理

`ExternalProject_Add` 命令可用于添加第三方源。然而，第一个例子展示了，如何将自己的项目，分为不同CMake项目的集合管理。本例中，主 `CMakeLists.txt` 和子 `CMakeLists.txt` 都声明了一个CMake项目，它们都使用了 `project` 命令。

`ExternalProject_Add` 有许多选项，可用于外部项目的配置和编译等所有方面。这些选择可以分为以下几类：

- **Directory**: 它们用于调优源码的结构，并为外部项目构建目录。本例中，我们使用 `SOURCE_DIR` 选项让CMake知道源文件在 `${CMAKE_CURRENT_LIST_DIR}/src` 文件夹中。用于构建项目和存储临时文件的目录，也可以在此类选项或目录属性中指定。通过设置 `EP_BASE` 目录属性，CMake将按照以下布局为各个子项目设置所有目录：

```

1. TMP_DIR = <EP_BASE>/tmp/<name>
2. STAMP_DIR = <EP_BASE>/Stamp/<name>
3. DOWNLOAD_DIR = <EP_BASE>/Download/<name>
4. SOURCE_DIR = <EP_BASE>/Source/<name>
5. BINARY_DIR = <EP_BASE>/Build/<name>
6. INSTALL_DIR = <EP_BASE>/Install/<name>

```

- **Download**: 外部项目的代码可能需要从在线存储库或资源处下载。
- **Update**和**Patch**: 可用于定义如何更新外部项目的源代码或如何应用补丁。
- **Configure**: 默认情况下, CMake会假定外部项目是使用CMake配置的。如下面的示例所示, 我们并不局限于这种情况。如果外部项目是CMake项目, `ExternalProject_Add` 将调用CMake可执行文件, 并传递选项。对于当前的示例, 我们通过 `CMAKE_ARGS` 和 `CMAKE_CACHE_ARGS` 选项传递配置参数。前者作为命令行参数直接传递, 而后者通过CMake脚本文件传递。示例中, 脚本文件位于 `build/subprojects/tmp/recipe-01_core/recipe-01_core- cache-.cmake`。然后, 配置如以下所示:

```

1. $ cmake -DCMAKE_CXX_COMPILER=g++ -DCMAKE_CXX_STANDARD=11
2. -DCMAKE_CXX_EXTENSIONS=OFF -DCMAKE_CXX_STANDARD_REQUIRED=ON
   -C/home/roberto/Workspace/robetodr/cmake-cookbook/chapter-08/recipe-01/cxx-
3. "-GUnix Makefiles" /home/roberto/Workspace/robetodr/cmake-cookbook/chapter-

```

- **Build**: 可用于调整外部项目的实际编译。我们的示例使用 `BUILD_ALWAYS` 选项确保外部项目总会重新构建。
- **Install**: 这些选项用于配置应该如何安装外部项目。我们的示例将 `INSTALL_COMMAND` 保留为空, 我们将在第10章(编写安装程序)中更详细地讨论与CMake的安装。
- **Test**: 为基于源代码构建的软件运行测试总是不错的想法。`ExternalProject_Add` 的这类选项可以用于此目的。我们的示例没有使用这些选项, 因为“Hello, World”示例没有任何测试, 但是在第5节中, 您将管理超级构建的项目, 届时将触发一个测试步骤。

`ExternalProject.cmake` 定义了 `ExternalProject_Get_Property` 命令, 该命令对于检索外部项目的属性非常有用。外部项目的属性是在首次调用 `ExternalProject_Add` 命令时设置的。例如, 在配置 `recipe-01_core` 时, 检索要传递给CMake的参数可以通过以下方法实现:

```

1. ExternalProject_Get_Property(${PROJECT_NAME}_core CMAKE_ARGS)
2. message(STATUS "CMAKE_ARGS of ${PROJECT_NAME}_core ${CMAKE_ARGS}")

```

NOTE: `ExternalProject_Add` 的完整选项列表可以在CMake文档中找

到:https://cmake.org/cmake/help/v3.5/module/ExternalProject.html#command:ExternalProject_Add

更多信息

下面的示例中，我们将详细讨论 `ExternalProject_Add` 命令的灵活性。然而，有时我们希望使用的外部项目可能需要执行额外的步骤。由于这个原因，`ExternalProject.cmake` 模块定义了以下附加命令：

1. `ExternalProject_Add_Step` : 当添加了外部项目，此命令允许将附加的命令作为自定义步骤锁定在其上。参见:https://cmake.org/cmake/help/v3.5/module/externalproject.html#command:externalproject_add_step
2. `ExternalProject_Add_StepTargets` : 允许将外部项目中的步骤(例如：构建和测试步骤)定义为单独的目标。这意味着可以从完整的外部项目中单独触发这些步骤，并允许对项目中的复杂依赖项，进行细粒度控制。参见:https://cmake.org/cmake/help/v3.5/module/ExternalProject.html#command:externalproject_add_steptargets
3. `ExternalProject_Add_StepDependencies` : 外部项目的步骤有时可能依赖于外部目标，而这个命令的设计目的就是处理这些情况。参见:https://cmake.org/cmake/help/v3.5/module/ExternalProject.html#command:externalproject_add_stepdependencies

8.2 使用超级构建管理依赖项：I. Boost库

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-8/recipe-02> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Boost库提供了丰富的C++基础工具，在C++开发人员中很受欢迎。第3章中，已经展示了如何在系统上找到Boost库。然而，有时系统上可能没有项目所需的Boost版本。这个示例将展示如何利用超级构建模式来交付代码，并确保在缺少依赖项时，不会让CMake停止配置。我们将重用在第3章第8节的示例代码，以超构建的形式重新组织。这是项目的文件结构：

```

1. .
2. └── CMakeLists.txt
3. └── external
4.   └── upstream
5.     └── boost
6.       ├── CMakeLists.txt
7.       └── CMakeLists.txt
8. └── src
9.   ├── CMakeLists.txt
10.  └── path-info.cpp

```

注意到项目源代码树中有四个 `CMakeLists.txt` 文件。下面的部分将对这些文件进行详解。

具体实施

从根目录的 `CMakeLists.txt` 开始：

1. 声明一个C++11项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-02 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 对 `EP_BASE` 进行属性设置：

```
1. set_property(DIRECTORY PROPERTY EP_BASE ${CMAKE_BINARY_DIR}/subprojects)
```

3. 我们设置了 `STAGED_INSTALL_PREFIX` 变量。此目录将用于安装构建树中的依赖项：

```
1. set(STAGED_INSTALL_PREFIX ${CMAKE_BINARY_DIR}/stage)
2. message(STATUS "${PROJECT_NAME} staged install: ${STAGED_INSTALL_PREFIX}")
```

4. 项目需要Boost库的文件系统和系统组件。我们声明了一个列表变量来保存这个信息，并设置了Boost所需的最低版本：

```
1. list(APPEND BOOST_COMPONENTS_REQUIRED filesystem system)
2. set(Boost_MINIMUM_REQUIRED 1.61)
```

5. 添加 `external/upstream` 子目录，它将依次添加 `external/upstream/boost` 子目录：

```
1. add_subdirectory(external/upstream)
```

6. 然后，包括 `ExternalProject.cmake` 标准模块，其中定义了 `ExternalProject_Add` 命令，它是超级构建的关键：

```
1. include(ExternalProject)
```

7. 项目位于 `src` 子目录下，我们将它添加为一个外部项目。使用 `CMAKE_ARGS` 和 `CMAKE_CACHE_ARGS` 传递CMake选项：

```
1. ExternalProject_Add(${PROJECT_NAME}_core
2.   DEPENDS
3.     boost_external
4.   SOURCE_DIR
5.     ${CMAKE_CURRENT_LIST_DIR}/src
6.   CMAKE_ARGS
7.     -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
8.     -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
9.     -DCMAKE_CXX_EXTENSIONS=${CMAKE_CXX_EXTENSIONS}
10.    -DCMAKE_CXX_STANDARD_REQUIRED=${CMAKE_CXX_STANDARD_REQUIRED}
11.   CMAKE_CACHE_ARGS
12.     -DCMAKE_CXX_FLAGS:STRING=${CMAKE_CXX_FLAGS}
13.     -DCMAKE_INCLUDE_PATH:PATH=${BOOST_INCLUDEDIR}
14.     -DCMAKE_LIBRARY_PATH:PATH=${BOOST_LIBRARYDIR}
15.   BUILD_ALWAYS
```

```

16.      1
17.      INSTALL_COMMAND
18.      ""
19.  )

```

现在让我们看看 `external/upstream` 中的 `CMakeLists.txt`。这个文件只是添加了boost文件夹作为一个额外的目录：

1. `add_subdirectory(boost)`

`external/upstream/boost` 中的 `CMakeLists.txt` 描述了满足对Boost的依赖所需的操作。我们的目标很简单，如果没有安装所需的版本，下载源打包文件并构建它：

1. 首先，我们试图找到所需Boost组件的最低版本：

```

find_package(Boost ${Boost_MINIMUM_REQUIRED} QUIET COMPONENTS
1. "${${BOOST_COMPONENTS_REQUIRED}}")

```

2. 如果找到这些，则添加一个接口库目标 `boost_external`。这是一个虚拟目标，需要在我们的超级构建中正确处理构建顺序：

```

1. if(Boost_FOUND)
   message(STATUS "Found Boost version
2. ${Boost_MAJOR_VERSION}.${Boost_MINOR_VERSION}.${Boost_SUBMINOR_VERSION}")
3. add_library(boost_external INTERFACE)
4. else()
5.   # ... discussed below
6. endif()

```

3. 如果 `find_package` 没有成功，或者正在强制进行超级构建，我们需要建立一个本地构建的Boost。为此，我们进入 `else` 部分：

```

1. else()
   message(STATUS "Boost ${Boost_MINIMUM_REQUIRED} could not be located,
2. Building Boost 1.61.0 instead.")

```

4. 由于这些库不使用CMake，我们需要为它们的原生构建工具链准备参数。首先为Boost设置编译器：

```

1. if(CMAKE_CXX_COMPILER_ID MATCHES "GNU")
2.   if(APPLE)

```

```

3.      set(_toolset "darwin")
4.    else()
5.      set(_toolset "gcc")
6.    endif()
7. elseif(CMAKE_CXX_COMPILER_ID MATCHES ".*Clang")
8.   set(_toolset "clang")
9. elseif(CMAKE_CXX_COMPILER_ID MATCHES "Intel")
10.  if(APPLE)
11.    set(_toolset "intel-darwin")
12.  else()
13.    set(_toolset "intel-linux")
14.  endif()
15. endif()

```

5. 我们准备了基于所需组件构建的库列表，定义了一些列表变量：`_build_byproducts`，包含要构建的库的绝对路径；`_b2_select_libraries`，包含要构建的库的列；和`_bootstrap_select_libraries`，这是一个字符串，与`_b2_needed_components`具有相同的内容，但格式不同：

```

1. if(NOT "${BOOST_COMPONENTS_REQUIRED}" STREQUAL "")
2.   # Replace unit_test_framework (used by CMake's find_package) with test
3.   # understood by Boost build toolchain)
4.   string(REPLACE "unit_test_framework" "test" _b2_needed_components
5.   "${BOOST_COMPONENTS_REQUIRED}")
6.   # Generate argument for BUILD_BYPRODUCTS
7.   set(_build_byproducts)
8.   set(_b2_select_libraries)
9.   foreach(_lib IN LISTS _b2_needed_components)
10.    list(APPEND _build_byproducts
11.      ${STAGED_INSTALL_PREFIX}/boost/lib/libboost_${_lib}${CMAKE_SHARED_LIBRARY_SUFFIX})
12.    list(APPEND _b2_select_libraries --with-${_lib})
13. endforeach()
14.   # Transform the ;-separated list to a , -separated list (digested by the
15.   # Boost build toolchain!)
16.   string(REPLACE ";" "," _b2_needed_components "${_b2_needed_components}")
17.   set(_bootstrap_select_libraries "--with-
18.   libraries=${_b2_needed_components}")
19.   string(REPLACE ";" " " printout "${BOOST_COMPONENTS_REQUIRED}")
20.   message(STATUS " Libraries to be built: ${printout}")
21. endif()

```

6. 现在，可以将Boost添加为外部项目。首先，在下载选项类中指定下载URL和

checksum。 `DOWNLOAD_NO_PROGRESS` 设置为1，以禁止打印下载进度信息：

```

1. include(ExternalProject)
2. ExternalProject_Add(boost_external
3.   URL
4.   https://sourceforge.net/projects/boost/files/boost/1.61.0/boost_1_61_0.zip
5.   URL_HASH
6.   SHA256=02d420e6908016d4ac74dfc712eec7d9616a7fc0da78b0a1b5b937536b2e01e8
7.   DOWNLOAD_NO_PROGRESS
8.   1

```

7. 接下来，设置更新/补丁和配置选项：

```

1. UPDATE_COMMAND
2. """
3. CONFIGURE_COMMAND
4.   <SOURCE_DIR>/bootstrap.sh
5.   --with-toolset=${_toolset}
6.   --prefix=${STAGED_INSTALL_PREFIX}/boost
7. ${_bootstrap_select_libraries}

```

8. 构建选项使用 `BUILD_COMMAND` 设置。`BUILD_IN_SOURCE` 设置为1时，表示构建将在源目录中发生。这里，将 `LOG_BUILD` 设置为1，以便将生成脚本中的输出记录到文件中：

```

1. BUILD_COMMAND
2. <SOURCE_DIR>/b2 -q
3. link=shared
4. threading=multi
5. variant=release
6. toolset=${_toolset}
7. ${_b2_select_libraries}
8. LOG_BUILD
9. 1
10. BUILD_IN_SOURCE
11. 1

```

9. 安装选项是使用 `INSTALL_COMMAND` 指令设置的。注意使用 `LOG_INSTALL` 选项，还可以将安装步骤记录到文件中：

```
1. INSTALL_COMMAND
```

```

2.      <SOURCE_DIR>/b2 -q install
3.          link=shared
4.          threading=multi
5.          variant=release
6.          toolset=${_toolset}
7.          ${_b2_select_libraries}
8. LOG_INSTALL
9.      1

```

10. 最后，库列表为 `BUILD_BYPRODUCTS` 并关闭 `ExternalProject_Add` 命令：

```

1. BUILD_BYPRODUCTS
2. "${_build_byproducts}"
3. )

```

11. 我们设置了一些变量来指导检测新安装的Boost：

```

1. set(
2.   BOOST_ROOT ${STAGED_INSTALL_PREFIX}/boost
3.   CACHE PATH "Path to internally built Boost installation root"
4.   FORCE
5. )
6. set(
7.   BOOST_INCLUDEDIR ${BOOST_ROOT}/include
8.   CACHE PATH "Path to internally built Boost include directories"
9.   FORCE
10. )
11. set(
12.   BOOST_LIBRARYDIR ${BOOST_ROOT}/lib
13.   CACHE PATH "Path to internally built Boost library directories"
14.   FORCE
15. )

```

12. `else` 分支中，执行的最后一个操作是取消所有内部变量的设置：

```

1. unset(_toolset)
2. unset(_b2_needed_components)
3. unset(_build_byproducts)
4. unset(_b2_select_libraries)
5. unset(_bootstrap_select_libraries)

```

最后，让我们看看 `src/CMakeLists.txt`。这个文件描述了一个独立的项目：

1. 声明一个C++项目：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-02_core LANGUAGES CXX)
```

2. 调用 `find_package` 寻找项目依赖的Boost。从主 `CMakeLists.txt` 中配置的项目，可以保证始终满足依赖关系，方法是使用预先安装在系统上的Boost，或者使用我们作为子项目构建的Boost：

```
1. find_package(Boost 1.61 REQUIRED COMPONENTS filesystem)
```

3. 添加可执行目标，并链接库：

```
1. add_executable(path-info path-info.cpp)
2. target_link_libraries(path-info
3.   PUBLIC
4.     Boost::filesystem
5. )
```

NOTE: 导入目标虽然很简单，但不能保证对任意Boost和CMake版本组合都有效。这是因为CMake的 `FindBoost.cmake` 模块会创建手工导入的目标。因此，当CMake有未知版本发布时，可能会有 `Boost_LIBRARIES` 和 `Boost_INCLUDE_DIRS`，没有导入情况

(<https://stackoverflow.com/questions/42123509/cmake-finds-boost-but-the-imported-targets-not-available-for-boost-version>)。

工作原理

此示例展示了如何利用超级构建模式，来整合项目的依赖项。让我们再看一下项目的文件结构：

```
1.
2.   |
3.   +-- CMakeLists.txt
4.   |
5.   +-- external
6.       |
7.       +-- upstream
8.           |
9.           +-- boost
10.          |
11.          +-- CMakeLists.txt
12.          |
13.          +-- CMakeLists.txt
14.      |
15.      +-- src
16.          |
17.          +-- CMakeLists.txt
18.          |
19.          +-- path-info.cpp
```

我们在项目源代码树中，引入了4个 `CMakeLists.txt` 文件：

1. 主 `CMakeLists.txt` 将配合超级构建。
2. `external/upstream` 中的文件将引导我们到 `boost` 子目录。
3. `external/upstream/boost/CMakeLists.txt` 将处理Boost的依赖。
4. 最后，`src` 下的 `CMakeLists.txt` 将构建我们的示例代码(其依赖于Boost)。

从 `external/upstream/boost/CMakeLists.txt` 文件开始讨论。Boost使用它自己的构建系统，因此需要在 `ExternalProject_Add` 中详细配置，以便正确设置所有内容：

1. 保留目录选项的默认值。
2. 下载步骤将从在线服务器下载所需版本的Boost。因此，我们设置了 `URL` 和 `URL_HASH`。`URL_HASH` 用于检查下载文件的完整性。由于我们不希望看到下载的进度报告，所以将 `DOWNLOAD_NO_PROGRESS` 选项设置为true。
3. 更新步骤留空。如果需要重新构建，我们不想再次下载Boost。
4. 配置步骤将使用由Boost在 `CONFIGURE_COMMAND` 中提供的配置工具完成。由于我们希望超级构建是跨平台的，所以我们使用 `<SOURCE_DIR>` 变量来引用未打包源的位置：

```

1. CONFIGURE_COMMAND
2. <SOURCE_DIR>/bootstrap.sh
3. --with-toolset=${_toolset}
4. --prefix=${STAGED_INSTALL_PREFIX}/boost
5. ${_bootstrap_select_libraries}

```

5. 将 `BUILD_IN_SOURCE` 选项设置为true，说明这是一个内置的构建。`BUILD_COMMAND` 使用Boost本机构建工具 `b2`。由于我们将在源代码中构建，所以我们再次使用 `<SOURCE_DIR>` 变量来引用未打包源代码的位置。
6. 然后，来看安装选项。Boost使用本地构建工具管理安装。事实上，构建和安装命令可以整合为一个命令。
7. 输出日志选项 `LOG_BUILD` 和 `LOG_INSTALL` 直接用于为 `ExternalProject_Add` 构建和安装操作编写日志文件，而不是输出到屏幕上。
8. 最后，`BUILD_BYPRODUCTS` 选项允许 `ExternalProject_Add` 在后续构建中，跟踪新构建的Boost库。

构建Boost之后，构建目录中的 `${STAGED_INSTALL_PREFIX}/Boost` 文件夹将包含所需的库。我们需要将此信息传递给我们的项目，该构建系统是在 `src/CMakeLists.txt` 中生成的。为了实现这个目标，我们在主 `CMakeLists.txt` 的 `ExternalProject_Add` 中传递两个额外

的 `CMAKE_CACHE_ARGS` :

1. `CMAKE_INCLUDE_PATH`: CMake查找C/C++头文件的路径
2. `CMAKE_LIBRARY_PATH`: CMake将查找库的路径

将这些变量设置成新构建的Boost安装路径，可以确保正确地获取依赖项。

TIPS: 在配置项目时将 `CMAKE_DISABLE_FIND_PACKAGE_Boost` 设置为 `ON`，将跳过对Boost库的检测，并始终执行超级构建。参考文

档:https://cmake.org/cmake/help/v3.5/variable/CMAKE_DISABLE_FIND_PACKAGE_PackageName.html。

8.3 使用超级构建管理依赖项：II .FFTW库

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-8/recipe-03> 中找到，其中有一个C示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

对于CMake支持的所有项目，超级构建模式可用于管理相当复杂的依赖关系。正如在前面的示例所演示的，CMake并不需要管理各种子项目。与前一个示例相反，这个示例中的外部子项目将是一个CMake项目，并将展示如何使用超级构建，下载、构建和安装FFTW库。FFTW是一个快速傅里叶变换库，可在<http://www.fftw.org> 免费获得。

我们项目的代码 `fftw_example.c` 位于src子目录中，它将计算源代码中定义的函数的傅里叶变换。

准备工作

这个示例的目录布局，是超级构建中非常常见的结构：

```

1. .
2. └── CMakeLists.txt
3. └── external
4.   └── upstream
5.     ├── CMakeLists.txt
6.     └── fftw3
7.       └── CMakeLists.txt
8. └── src
9.   ├── CMakeLists.txt
10.  └── fftw_example.c

```

代码 `fftw_example.c` 位于 `src` 子目录中，它将调用傅里叶变换函数。

具体实施

从主 `CMakeLists.txt` 开始，这里将整个超级构建过程放在一起：

1. 声明一个支持C99的项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03 LANGUAGES C)
3. set(CMAKE_C_STANDARD 99)
4. set(CMAKE_C_EXTENSIONS OFF)

```

```
5. set(CMAKE_C_STANDARD_REQUIRED ON)
```

2. 和上一个示例一样，我们设置了 `EP_BASE` 目录属性和阶段安装目录：

```
1. set_property(DIRECTORY PROPERTY EP_BASE ${CMAKE_BINARY_DIR}/subprojects)
2. set(STAGED_INSTALL_PREFIX ${CMAKE_BINARY_DIR}/stage)
3. message(STATUS "${PROJECT_NAME} staged install: ${STAGED_INSTALL_PREFIX}")
```

3. 对FFTW的依赖关系在 `external/upstream` 子目录中检查，我们会将这个子目录添加到构建系统中：

```
1. add_subdirectory(external/upstream)
```

4. 包含 `ExternalProject.cmake` 模块：

```
1. include(ExternalProject)
```

5. 我们为 `recipe-03_core` 声明了外部项目。这个项目的源代码在 `${CMAKE_CURRENT_LIST_DIR}/src` 文件夹中。该项目设置为 `FFT3_DIR` 选项，选择正确的FFTW库：

```
1. ExternalProject_Add(${PROJECT_NAME}_core
2.   DEPENDS
3.     fftw3_external
4.   SOURCE_DIR
5.     ${CMAKE_CURRENT_LIST_DIR}/src
6.   CMAKE_ARGS
7.     -DFFTW3_DIR=${FFT3_DIR}
8.     -DCMAKE_C_STANDARD=${CMAKE_C_STANDARD}
9.     -DCMAKE_C_EXTENSIONS=${CMAKE_C_EXTENSIONS}
10.    -DCMAKE_C_STANDARD_REQUIRED=${CMAKE_C_STANDARD_REQUIRED}
11.   CMAKE_CACHE_ARGS
12.     -DCMAKE_C_FLAGS:STRING=${CMAKE_C_FLAGS}
13.     -DCMAKE_PREFIX_PATH:PATH=${CMAKE_PREFIX_PATH}
14.   BUILD_ALWAYS
15.     1
16.   INSTALL_COMMAND
17.     ""
18. )
```

`external/upstream` 子目录还包含一个 `CMakeLists.txt`：

这个文件中，添加 `fftw3` 文件夹作为构建系统中的另一个子目录：

1. `add_subdirectory(fftw3)`

`external/upstream/fftw3` 中的 `CMakeLists.txt` 负责处理依赖关系：

1. 首先，尝试在系统上找到FFTW3库。注意，我们配置 `find_package` 使用的参数：

1. `find_package(FFTW3 CONFIG QUIET)`

2. 如果找到了库，就可以导入目标 `FFTW3::FFTW3` 来链接它。我们向用户打印一条消息，显示库的位置。我们添加一个虚拟 `INTERFACE` 库 `fftw3_external`。超级建设中，这需要正确地固定子项目之间的依赖树：

```

1. find_package(FFTW3 CONFIG QUIET)
2.
3. if(FFTW3_FOUND)
4.   get_property(_loc TARGET FFTW3::fftw3 PROPERTY LOCATION)
5.   message(STATUS "Found FFTW3: ${_loc} (found version ${FFTW3_VERSION})")
6.   add_library(fftw3_external INTERFACE) # dummy
7. else()
8.   # this branch will be discussed below
9. endif()

```

3. 如果CMake无法找到预安装版本的FFTW，我们将进入 `else` 分支。这个分支中，使用 `ExternalProject_Add` 下载、构建和安装它。外部项目的名称为 `fftw3_external`。`fftw3_external` 项目将从官方地址下载，下载完成后将使用MD5校验和进行文件完整性检查：

```

message(STATUS "Suitable FFTW3 could not be located. Downloading and
1. building!")
2.
3. include(ExternalProject)
4. ExternalProject_Add(fftw3_external
5.   URL
6.     http://www.fftw.org/fftw-3.3.8.tar.gz
7.   URL_HASH
8.     MD5=8aac833c943d8e90d51b697b27d4384d

```

4. 禁用打印下载进程，并将更新命令定义为空：

```

1.      OWNLOAD_NO_PROGRESS
2.          1
3.      UPDATE_COMMAND
4.          ""

```

5. 配置、构建和安装输出将被记录到一个文件中：

```

1.      LOG_CONFIGURE
2.          1
3.      LOG_BUILD
4.          1
5.      LOG_INSTALL
6.          1

```

6. 将 `fftw3_external` 项目的安装前缀设置为之前定义的 `STAGED_INSTALL_PREFIX` 目录，并关闭FFTW3的测试套件构建：

```

1.      CMAKE_ARGS
2.          -DCMAKE_INSTALL_PREFIX=${STAGED_INSTALL_PREFIX}
3.          -DBUILD_TESTS=OFF

```

7. 如果在Windows上构建，通过生成器表达式设置 `WITH OUR_MALLOC` 预处理器选项，并关闭 `ExternalProject_Add` 命令：

```

1.      CMAKE_CACHE_ARGS
2.          -DCMAKE_C_FLAGS:STRING=$<$<BOOL:WIN32>:-DWITH OUR_MALLOC>
3.      )

```

8. 最后，定义 `FFTW3_DIR` 变量并缓存它。CMake将使用该变量作为 `FFTW3::FFTW3` 目标的搜索目录：

```

1.      include(GNUInstallDirs)
2.
3.      set(
4.          FFTW3_DIR ${STAGED_INSTALL_PREFIX}/${CMAKE_INSTALL_LIBDIR}/cmake/fftw3
5.          CACHE PATH "Path to internally built FFTW3Config.cmake"
6.          FORCE
7.      )

```

`src` 文件夹中的CMakeLists.txt相当简洁：

1. 同样在这个文件中，我们声明了一个C项目：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-03_core LANGUAGES C)
```

2. 使用 `find_package` 来检测FFTW库，再次使用配置检测模式：

```
1. find_package(FFTW3 CONFIG REQUIRED)
2. get_property(_loc TARGET FFTW3::fftw3 PROPERTY LOCATION)
3. message(STATUS "Found FFTW3: ${_loc} (found version ${FFTW3_VERSION})")
```

3. 将 `fftw_example.c` 源文件添加到可执行目标 `fftw_example`：

```
1. add_executable(fftw_example fftw_example.c)
```

4. 为可执行目标设置链接库：

```
1. target_link_libraries(fftw_example
2.   PRIVATE
3.   FFTW3::fftw3
4. )
```

工作原理

本示例演示了如何下载、构建和安装由CMake管理其构建系统的外部项目。与前一个示例(必须使用自定义构建系统)相反，这个超级构建设置相当简洁。需要注意的是，使用 `find_package` 命令了配置选项；这说明CMake首先查找 `FFTW3Config.cmake`，以定位FFTW3库，将库导出为第三方项目获取的目标。目标包含库的版本、配置和位置，即关于如何配置和构建目标的完整信息。如果系统上没有安装库，我们需要声明 `FFTW3Config.cmake` 文件的位置。这可以通过设置 `FFTW3_DIR` 变量来实现。这是 `external/upstream/fftw3/CMakeLists.txt` 文件中的最后一步。使用 `GNUInstallDirs.cmake` 模块，我们将 `FFTW3_DIR` 设置为缓存变量，以便稍后在超级构建中使用。

TIPS: 配置项目时将 `CMAKE_DISABLE_FIND_PACKAGE_FFTW3` 设置为 `ON`，将跳过对FFTW库的检测，并始终执行超级构建。参考：https://cmake.org/cmake/help/v3.5/variable/CMAKE_DISABLE_FIND_PACKAGE_PackageName.html

8.4 使用超级构建管理依赖项:III.Google Test框架

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-8/recipe-04> 中找到，其中有一个C++示例。该示例在CMake 3.11版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。在库中也有一个例子可以在CMake 3.5下使用。

第4章第3节中，我们使用Google Test框架实现单元测试，并在配置时使用 `FetchContent` 模块获取Google Test源(自CMake 3.11开始可用)。本章中，我们将重新讨论这个方法，较少关注测试方面，并更深入地研究 `FetchContent`。它提供了一个简单通用的模块，可以在配置时组装项目依赖项。对于3.11以下的CMake，我们还将讨论如何在配置时使用 `ExternalProject_Add` 模拟 `FetchContent`。

准备工作

这个示例中，我们将复用第4章第3节的源码，构建 `main.cpp`、`sum_integer.cpp` 和 `sum_integers.hpp` 和 `test.cpp`。我们将在配置时使用 `FetchContent` 或 `ExternalProject_Add` 下载所有必需的Google Test源，在此示例中，只关注在配置时获取依赖项，而不是实际的源代码及其单元测试。

具体实施

这个示例中，我们只关注如何获取Google Test源来构建 `gtest_main`，并链接到Google Test库。关于这个目标如何用于测试示例源的讨论，请读者参考第4章第3节：

- 首先包括 `FetchContent` 模块，它将提供需要的声明、查询和填充依赖项函数：

```
1. include(FetchContent)
```

- 然后，声明内容—名称、存储库位置和要获取的精确版本：

```
1. FetchContent_Declare(
2.   googletest
3.   GIT_REPOSITORY https://github.com/google/googletest.git
4.   GIT_TAG release-1.8.0
5. )
```

- 查询内容是否已经被获取/填充：

```
1. FetchContent_GetProperties(googletest)
```

4. 前面的函数定义了 `googletest_POPULATED`。如果内容还没有填充，我们获取内容并配置子项目：

```
1. if(NOT googletest_POPULATED)
2.   FetchContent_Populate(googletest)
3.
4. # ...
5.
6. # adds the targets: gtest, gtest_main, gmock, gmock_main
7. add_subdirectory(
8.   ${googletest_SOURCE_DIR}
9.   ${googletest_BINARY_DIR}
10. )
11.
12. # ...
13.
14. endif()
```

5. 注意配置时获取内容的方式：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
```

6. 这将生成以下构建目录树。Google Test源现已就绪，剩下的就交由CMake处理，并提供所需的目标：

```
1. build/
2. └─ ...
3.   └─ _deps
4.     └─ gtest-build
5.       └─ ...
6.       └─ ...
7.     └─ gtest-src
8.       └─ ...
9.       └─ ...
10.      └─ gtest-subbuild
11.        └─ ...
12.        └─ ...
```

13. └ ...

工作原理

`FetchContent` 模块支持在配置时填充内容。例子中，获取了一个Git库，其中有一个Git标签：

```

1. FetchContent_Declare(
2.   googletest
3.   GIT_REPOSITORY https://github.com/google/googletest.git
4.   GIT_TAG release-1.8.0
5. )
```

CMake的3.11版本中，`FetchContent` 已经成为CMake的标准部分。下面的代码中，将尝试在配置时使用 `ExternalProject_Add` 模拟 `FetchContent`。这不仅适用于较老的CMake版本，而且可以让我们更深入地了解 `FetchContent` 层下面发生了什么，并为使用 `ExternalProject_Add` 在构建时获取项目，提供一个有趣的替代方法。我们的目标是编写一个 `fetch_git_repo` 宏，并将它放在 `fetch_git_repo` 中。这样就可以获取相应的内容了：

```

1. include(fetch_git_repo.cmake)
2.
3. fetch_git_repo(
4.   googletest
5.   ${CMAKE_BINARY_DIR}/_deps
6.   https://github.com/google/googletest.git
7.   release-1.8.0
8. )
9.
10. # ...
11.
12. # adds the targets: gtest, gtest_main, gmock, gmock_main
13. add_subdirectory(
14.   ${googletest_SOURCE_DIR}
15.   ${googletest_BINARY_DIR}
16. )
```

17.

18. # ...

这类似于 `FetchContent` 的使用。在底层实现中，我们将使用 `ExternalProject_Add`。现在打开模块，检查 `fetch_git_repo.cmake` 中定义的 `fetch_git_repo`：

```

1. macro(fetch_git_repo _project_name _download_root _git_url _git_tag)
2.
3.   set(${_project_name}_SOURCE_DIR ${_download_root}/${_project_name}-src)
4.   set(${_project_name}_BINARY_DIR ${_download_root}/${_project_name}-build)
5.
6.   # variables used configuring fetch_git_repo_sub.cmake
7.   set(FETCH_PROJECT_NAME ${_project_name})
8.   set(FETCH_SOURCE_DIR ${${_project_name}_SOURCE_DIR})
9.   set(FETCH_BINARY_DIR ${${_project_name}_BINARY_DIR})
10.  set(FETCH_GIT_REPOSITORY ${_git_url})
11.  set(FETCH_GIT_TAG ${_git_tag})
12.
13.  configure_file(
14.    ${CMAKE_CURRENT_LIST_DIR}/fetch_at_configure_step.in
15.    ${_download_root}/CMakeLists.txt
16.    @ONLY
17.  )
18.
19.  # undefine them again
20.  unset(FETCH_PROJECT_NAME)
21.  unset(FETCH_SOURCE_DIR)
22.  unset(FETCH_BINARY_DIR)
23.  unset(FETCH_GIT_REPOSITORY)
24.  unset(FETCH_GIT_TAG)
25.
26.  # configure sub-project
27.  execute_process(
28.    COMMAND
29.    "${CMAKE_COMMAND}" -G "${CMAKE_GENERATOR}" .
30.    WORKING_DIRECTORY
31.    ${_download_root}
32.  )
33.
34.  # build sub-project which triggers ExternalProject_Add
35.  execute_process(
36.    COMMAND
37.    "${CMAKE_COMMAND}" --build .
38.    WORKING_DIRECTORY
39.    ${_download_root}
40.  )
41. endmacro()

```

宏接收项目名称、下载根目录、Git存储库URL和一个Git标记。宏定义了 `$_project_name_SOURCE_DIR` 和 `$_project_name_BINARY_DIR`，我们需要在 `fetch_git_repo` 生命周期范围内使用定义的 `$_project_name_SOURCE_DIR` 和 `$_project_name_BINARY_DIR`，因为要使用它们对子目录进行配置：

```
1. add_subdirectory(
2.   ${googletest_SOURCE_DIR}
3.   ${googletest_BINARY_DIR}
4. )
```

`fetch_git_repo` 宏中，我们希望使用 `ExternalProject_Add` 在配置时获取外部项目，通过三个步骤实现了这一点：

- 首先，配置 `fetch_at_configure_step.in`：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(fetch_git_repo_sub LANGUAGES NONE)
4.
5. include(ExternalProject)
6.
7. ExternalProject_Add(
8.   @FETCH_PROJECT_NAME@
9.   SOURCE_DIR "@FETCH_SOURCE_DIR@"
10.  BINARY_DIR "@FETCH_BINARY_DIR@"
11.  GIT_REPOSITORY
12.  @FETCH_GIT_REPOSITORY@
13.  GIT_TAG
14.  @FETCH_GIT_TAG@
15.  CONFIGURE_COMMAND ""
16.  BUILD_COMMAND ""
17.  INSTALL_COMMAND ""
18.  TEST_COMMAND ""
19. )
```

使用 `configure_file`，可以生成一个 `CMakeLists.txt` 文件，前面的占位符被 `fetch_git_repo.cmake` 中的值替换。注意，前面的 `ExternalProject_Add` 命令仅用于获取，而不仅是配置、构建、安装或测试。

- 其次，使用配置步骤在配置时触发 `ExternalProject_Add`（从主项目的角度）：

```

1. # configure sub-project
2. execute_process(
3.   COMMAND
4.   "${CMAKE_COMMAND}" -G "${CMAKE_GENERATOR}" .
5.   WORKING_DIRECTORY
6.   ${_download_root}
7. )

```

3. 最后在 `fetch_git_repo.cmake` 中触发配置时构建步骤:

```

1. # build sub-project which triggers ExternalProject_Add
2. execute_process(
3.   COMMAND
4.   "${CMAKE_COMMAND}" --build .
5.   WORKING_DIRECTORY
6.   ${_download_root}
7. )

```

这个解决方案的一个优点是，由于外部依赖项不是由 `ExternalProject_Add` 配置的，所以不需要通过 `ExternalProject_Add` 调用任何配置，将其引导至项目。我们可以使用 `add_subdirectory` 配置和构建模块，就像外部依赖项是项目源代码树的一部分一样。聪明的伪装！

更多信息

有关 `FetchContent` 选项的详细讨论，请参考<https://cmake.org/cmake/help/v3.11/module/FetchContent.html>配置时 `ExternalProject_Add` 的解决方案灵感来自Craig Scott，博客文章：<https://crascit.com/2015/07/25/cgtest/>

8.5 使用超级构建支持项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-8/recipe-05> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

`ExternalProject` 和 `FetchContent` 是CMake库中两个非常强大的工具。经过前面的示例，我们应该相信超级构建方法，在管理复杂依赖关系的项目时是多么有用。目前为止，我们已经展示了如何使用 `ExternalProject` 来处理以下问题：

- 存储在源树中的源
- 从在线服务器上，检索/获取可用的存档资源

前面的示例展示了，如何使用 `FetchContent` 处理开源Git存储库中可用的依赖项。本示例将展示，如何使用 `ExternalProject` 达到同样的效果。最后，将介绍一个示例，该示例将在第10章第4节中重用。

准备工作

这个超级构建的源代码树现在应该很熟悉了：

```

1. .
2. └── CMakeLists.txt
3. └── external
4.   └── upstream
5.     ├── CMakeLists.txt
6.     └── message
7.       └── CMakeLists.txt
8. └── src
9.   ├── CMakeLists.txt
10.  └── use_message.cpp

```

根目录有一个 `CMakeLists.txt`，我们知道它会配合超级构建。子目录 `src` 和 `external` 中是我们自己的源代码，CMake指令需要满足对消息库的依赖，我们将在本例中构建消息库。

具体实施

目前为止，建立超级构建的过程应该已经很熟悉了。让我们再次看看必要的步骤，从根目录的 `CMakeLists.txt` 开始：

1. 声明一个C++11项目，并对项目构建类型的默认值进行设置。

```

1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2.
3. project(recipe-05 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)
8.
9. if(NOT DEFINED CMAKE_BUILD_TYPE OR "${CMAKE_BUILD_TYPE}" STREQUAL "")
10.   set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
11. endif()
12.
13. message(STATUS "Build type set to ${CMAKE_BUILD_TYPE}")

```

2. 设置 `EP_BASE` 目录属性。这将固定 `ExternalProject` 管理所有子项目的布局：

```
1. set_property(DIRECTORY PROPERTY EP_BASE ${CMAKE_BINARY_DIR}/subprojects)
```

3. 我们设置了 `STAGED_INSTALL_PREFIX`。与之前一样，这个位置将作为依赖项的构建树中的安装目录：

```

1. set(STAGED_INSTALL_PREFIX ${CMAKE_BINARY_DIR}/stage)
2. message(STATUS "${PROJECT_NAME} staged install: ${STAGED_INSTALL_PREFIX}")

```

4. 将 `external/upstream` 作为子目录添加：

```
1. add_subdirectory(external/upstream)
```

5. 添加 `ExternalProject_Add`，这样我们的项目也将由超级构建管理：

```

1. include(ExternalProject)
2. ExternalProject_Add(${PROJECT_NAME}_core
3.   DEPENDS
4.     message_external
5.   SOURCE_DIR
6.     ${CMAKE_CURRENT_SOURCE_DIR}/src
7.   CMAKE_ARGS
8.     -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
9.     -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}

```

```

10.      -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
11.      -DCMAKE_CXX_EXTENSIONS=${CMAKE_CXX_EXTENSIONS}
12.      -DCMAKE_CXX_STANDARD_REQUIRED=${CMAKE_CXX_STANDARD_REQUIRED}
13.      -Dmessage_DIR=${message_DIR}
14.      CMAKE_CACHE_ARGS
15.      -DCMAKE_CXX_FLAGS:STRING=${CMAKE_CXX_FLAGS}
16.      -DCMAKE_PREFIX_PATH:PATH=${CMAKE_PREFIX_PATH}
17.      BUILD_ALWAYS
18.      1
19.      INSTALL_COMMAND
20.      ""
21.  )

```

`external/upstream` 的 `CMakeLists.txt` 中只包含一条命令：

1. `add_subdirectory(message)`

跳转到 `message` 文件夹，我们会看到对消息库的依赖的常用命令：

1. 首先，调用 `find_package` 找到一个合适版本的库：

1. `find_package(message 1 CONFIG QUIET)`

2. 如果找到，会通知用户，并添加一个虚拟 `INTERFACE` 库：

1. `get_property(_loc TARGET message::message-shared PROPERTY LOCATION)`
`message(STATUS "Found message: ${_loc} (found version`
`2. ${message_VERSION})")`
`3. add_library(message_external INTERFACE) # dummy`

3. 如果没有找到，再次通知用户并继续使用 `ExternalProject_Add`：

```

message(STATUS "Suitable message could not be located, Building message
1. instead.")

```

4. 该项目托管在一个公共Git库中，使用 `GIT_TAG` 选项指定下载哪个分支。和之前一样，将 `UPDATE_COMMAND` 选项置为空：

1. `include(ExternalProject)`
2. `ExternalProject_Add(message_external`
3. `GIT_REPOSITORY`
4. `https://github.com/dev-cafe/message.git`

```

5.    GIT_TAG
6.      master
7.    UPDATE_COMMAND
8.      ""

```

5. 外部项目使用CMake配置和构建，传递必要的构建选项：

```

1.    CMAKE_ARGS
2.      -DCMAKE_INSTALL_PREFIX=${STAGED_INSTALL_PREFIX}
3.      -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
4.      -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
5.      -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
6.      -DCMAKE_CXX_EXTENSIONS=${CMAKE_CXX_EXTENSIONS}
7.      -DCMAKE_CXX_STANDARD_REQUIRED=${CMAKE_CXX_STANDARD_REQUIRED}
8.    CMAKE_CACHE_ARGS
9.      -DCMAKE_CXX_FLAGS:STRING=${CMAKE_CXX_FLAGS}

```

6. 项目安装后进行测试：

```

1.    TEST_AFTER_INSTALL
2.      1

```

7. 我们不希望看到下载进度，也不希望在屏幕上报告配置、构建和安装信息，所以选择关闭 `ExternalProject_Add` :

```

1.    DOWNLOAD_NO_PROGRESS
2.      1
3.    LOG_CONFIGURE
4.      1
5.    LOG_BUILD
6.      1
7.    LOG_INSTALL
8.      1
9.  )

```

8. 为了确保子项目在超级构建的其余部分中是可见的，我们设置了 `message_DIR` 目录：

```

1. if(WIN32 AND NOT CYGWIN)
2.   set(DEF_message_DIR ${STAGED_INSTALL_PREFIX}/CMake)
3. else()
4.   set(DEF_message_DIR ${STAGED_INSTALL_PREFIX}/share/cmake/message)

```

```

5. endif()
6.
7. file(TO_NATIVE_PATH "${DEF_message_DIR}" DEF_message_DIR)
8. set(message_DIR ${DEF_message_DIR})
9. CACHE PATH "Path to internally built messageConfig.cmake" FORCE)

```

最后，来看一下 `src` 目录上的 `CMakeLists.txt`：

1. 同样，声明一个C++11项目：

```

1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2. project(recipe-05_core
3. LANGUAGES CXX
4. )
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 项目需要消息库：

```

1. find_package(message 1 CONFIG REQUIRED)
2. get_property(_loc TARGET message::message-shared PROPERTY LOCATION)
   message(STATUS "Found message: ${_loc} (found version
3. ${message_VERSION})")

```

3. 声明一个可执行目标，并将其链接到消息动态库：

```

1. add_executable(use_message use_message.cpp)
2.
3. target_link_libraries(use_message
4.   PUBLIC
5.     message::message-shared
6.   )

```

工作原理

示例展示了 `ExternalProject_Add` 的一些新选项：

1. **GIT_REPOSITORY**: 这可以用来指定包含依赖项源的存储库的URL。CMake还可以使用其他版本控制系统，比如CVS (`CVS_REPOSITORY`)、SVN (`SVN_REPOSITORY`)或Mercurial (`HG_REPOSITORY`)。

2. **GIT_TAG**: 默认情况下, CMake将检出给定存储库的默认分支。然而, 最好依赖于一个稳定的版本。这可以通过这个选项指定, 它可以接受Git将任何标识符识别为“版本”信息, 例如: Git提交SHA、Git标记或分支名称。CMake所理解的其他版本控制系统也可以使用类似的选项。
3. **TEST_AFTER_INSTALL**: 依赖项很可能有自己的测试套件, 您可能希望运行测试套件, 以确保在超级构建期间一切顺利。此选项将在安装步骤之后立即运行测试。

`ExternalProject_Add` 可以理解的其他测试选项如下:

- **TEST_BEFORE_INSTALL**: 将在安装步骤之前运行测试套件
- **TEST_EXCLUDE_FROM_MAIN**: 可以从测试套件中, 删除对外部项目的主要目标的依赖

这些选项都假定外部项目使用CTest管理测试。如果外部项目不使用CTest来管理测试, 我们可以通过 `TEST_COMMAND` 选项来执行测试。

即使是为属于自己项目的模块引入超级构建模式, 也需要引入额外的层, 重新声明小型CMake项目, 并通过 `ExternalProject_Add` 显式地传递配置设置。引入这个附加层的好处是, 清晰地分离了变量和目标范围, 这可以帮助管理由多个组件组成的项目中的复杂性、依赖性和名称空间, 这些组件可以是内部的, 也可以是外部的, 并由CMake组合在一起。

第9章 语言混合项目

本章的主要内容如下：

- 使用C/C++库构建Fortran项目
- 使用Fortran库构建C/C++项目
- 使用Cython构建C++和Python项目
- 使用Boost.Python构建C++和Python项目
- 使用pybind11构建C++和Python项目
- 使用Python CFFI混合C, C++, Fortran和Python

有很多的库比较适合特定领域的任务。我们的库直接使用这些专业库，是一种快捷的方式，这样就可以使用来自其他专家组的多年经验进行开发。随着计算机体系结构和编译器的发展，编程语言也在不断发展。几年前，大多数科学软件都是用Fortran语言编写的，而现在，C/C++和解释语言Python正占据着语言中心舞台。将编译语言代码与解释语言的代码集成在一起，变得确实越来越普遍，这样做有以下好处：

- 用户可以需要进行定制和扩展功能，以满足需求。
- 可以将Python等语言的表达能力与编译语言的性能结合起来，后者在内存寻址方面效率接近于极致，达到两全其美的目的。

正如之前的示例中展示的那样，可以使用 `project` 命令通过 `LANGUAGES` 关键字设置项目中使用的语言。CMake支持许多(但不是所有)编译的编程语言。从CMake 3.5开始，各种风格的汇编(如ASM-ATT, ASM, ASM-MASM和ASM- NASM)、C、C++、Fortran、Java、RC (Windows资源编译器)和Swift都可以选择。CMake 3.8增加了对另外两种语言的支持：C#和CUDA(请参阅发布说明：<https://cmake.org/cmake/help/v3.8/release/3.8.html#languages>)。

本章中，我们将展示如何以一种可移植且跨平台的方式集成用不同编译(C/C++和Fortran)和解释语言(Python)编写的代码。我们将展示如何利用CMake和一些工具集成不同编程语言。

9.1 使用C/C++库构建Fortran项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-01> 中找到，其中有两个示例：一个是Fortran与C的混例，另一个是Fortran和C++的混例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Fortran作为高性能计算语言有着悠久的历史。目前，许多线性代数库仍然使用Fortran语言编写，许多大型的数字处理包也保持与过去几十年的代码兼容。而Fortran提出了一个很自然的语法处理数值数组，它缺乏与操作系统交互，所以为了编程的通用性，需要一个互操作性层(使用C实现)，才发布了Fortran 2003标准。本示例将展示如何用C系统库和自定义C代码来对接Fortran代码。

准备工作

第7章中，我们把项目结构列作为一个树。每个子目录都有一个 `CMakeLists.txt` 文件，其中包含与该目录相关的指令。这使我们可以对子目录进行限制中，如这个例子：

```

1. .
2. └── CMakeLists.txt
3. └── src
4.     ├── bt-randomgen-example.f90
5.     ├── CMakeLists.txt
6.     └── interfaces
7.         ├── CMakeLists.txt
8.         ├── interface_backtrace.f90
9.         ├── interface_randomgen.f90
10.        └── randomgen.c
11.        └── utils
12.            ├── CMakeLists.txt
13.            └── util_strings.f90

```

我们的例子中，`src` 子目录中包括 `bt-randomgen-example.f90`，会将源码编译成可执行文件。另外两个子目录 `interface` 和 `utils` 包含更多的源代码，这些源代码将被编译成库。

`interfaces` 子目录中的源代码展示了如何包装向后追踪的C系统库。例如，`interface_backtrace.f90`：

```

1. module interface_backtrace
2.
3.   implicit none

```

```

4.
5.      interface
6.          function backtrace(buffer, size) result(bt) bind(C, name="backtrace")
7.              use, intrinsic :: iso_c_binding, only: c_int, c_ptr
8.              type(c_ptr) :: buffer
9.              integer(c_int), value :: size
10.             integer(c_int) :: bt
11.         end function
12.
13.         subroutine backtrace_symbols_fd(buffer, size, fd) bind(C,
14.             name="backtrace_symbols_fd")
15.             use, intrinsic :: iso_c_binding, only: c_int, c_ptr
16.             type(c_ptr) :: buffer
17.             integer(c_int), value :: size, fd
18.         end subroutine
19.     end interface
20. end module

```

上面的例子演示了：

- 内置 `iso_c_binding` 模块，确保Fortran和C类型和函数的互操作性。
- `interface` 声明，将函数在单独库中绑定到相应的符号上。
- `bind(C)` 属性，为声明的函数进行命名修饰。

这个子目录还包含两个源文件：

- `randomgen.c`: 这是一个C源文件，它对外公开了一个函数，使用C标准 `rand` 函数在一个区间内生成随机整数。
- `interface_randomgen.f90`: 它将C函数封装在Fortran可执行文件中使用。

具体实施

我们有4个 `CMakeLists.txt` 实例要查看—根目录下1个，子目录下3个。让我们从根目录的 `CMakeLists.txt` 开始：

1. 声明一个Fortran和C的混合语言项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES Fortran C)

```

2. CMake将静态库和动态库保存在 `build` 目录下的 `lib` 目录中。可执行文件保存在 `bin` 目录下，Fortran编译模块文件保存在 `modules` 目录下：

```

1. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/lib)
2. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/lib)
3. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/bin)
4. set(CMAKE_Fortran_MODULE_DIRECTORY
5. ${CMAKE_CURRENT_BINARY_DIR}/modules)

```

3. 接下来，我们进入第一个子 `CMakeLists.txt`，添加 `src` 子目录：

```
1. add_subdirectory(src)
```

4. `src/CMakeLists.txt` 文件添加了两个子目录：

```

1. add_subdirectory(interfaces)
2. add_subdirectory(utils)

```

在 `interfaces` 子目录中，我们将执行以下操作：

1. 包括 `FortranCInterface.cmak` 模块，并验证C和Fortran编译器可以正确地交互：

```

1. include(FortranCInterface)
2. FortranCInterface_VERIFY()

```

2. 接下来，我们找到Backtrace系统库，因为我们想在Fortran代码中使用它：

```
1. find_package(Backtrace REQUIRED)
```

3. 然后，创建一个共享库目标，其中包含Backtrace包装器、随机数生成器，以及Fortran包装器的源文件：

```

1. add_library(bt-randomgen-wrap SHARED "")
2.
3. target_sources(bt-randomgen-wrap
4.   PRIVATE
5.     interface_backtrace.f90
6.     interface_randomgen.f90
7.     randomgen.c
8. )

```

4. 我们还为新生成的库目标设置了链接库。使用 `PUBLIC` 属性，以便连接到其他目标时，能正确地看到依赖关系：

```

1. target_link_libraries(bt-randomgen-wrap
2. PUBLIC
3.     ${Backtrace_LIBRARIES}
4. )

```

`utils` 子目录中，还有一个 `CMakeLists.txt`，其只有一单行程序：我们创建一个新的库目标，子目录中的源文件将被编译到这个目标库中。并与这个目标没有依赖关系：

```
1. add_library(utils SHARED util_strings.f90)
```

回到 `src/CMakeLists.txt`：

1. 使用 `bt-randomgen-example.f90` 添加一个可执行目标：

```
1. add_executable(bt-randomgen-example bt-randomgen-example.f90)
```

2. 最后，将在子 `CMakeLists.txt` 中生成的库目标，并链接到可执行目标：

```

1. target_link_libraries(bt-randomgen-example
2. PRIVATE
3.     bt-randomgen-wrap
4.     utils
5. )

```

工作原理

确定链接了正确库之后，需要保证程序能够正确调用函数。每个编译器在生成机器码时都会执行命名检查。不过，这种操作的约定不是通用的，而是与编译器相关的。`FortranCInterface`，我们已经在第3章第4节时，检查所选C编译器与Fortran编译器的兼容性。对于当前的目的，命名检查并不是一个真正的问题。Fortran 2003标准提供了可选 `name` 参数的函数和子例程定义了 `bind` 属性。如果提供了这个参数，编译器将使用程序员指定的名称为这些子例程和函数生成符号。例如，`backtrace`函数可以从C语言中暴露给Fortran，并保留其命名：

```
1. function backtrace(buffer, size) result(bt) bind(C, name="backtrace")
```

更多信息

`interface/CMakeLists.txt` 中的CMake代码还表明，可以使用不同语言的源文件创建库。CMake能够做到以下几点：

- 列出的源文件中获取目标文件，并识别要使用哪个编译器。
- 选择适当的链接器，以便构建库(或可执行文件)。

CMake如何决定使用哪个编译器？在 `project` 命令时使用参数 `LANGUAGES` 指定，这样CMake会检查系统上给定语言编译器。当使用源文件列表添加目标时，CMake将根据文件扩展名选择适当地编译器。因此，以 `.c` 结尾的文件使用C编译器编译，而以 `.f90` 结尾的文件(如果需要预处理，可以使用 `.F90`)将使用Fortran编译器编译。类似地，对于C++，`.cpp` 或 `.cxx` 扩展将触发 `C++` 编译器。我们只列出了C/C++和Fortran语言的一些可能的、有效的文件扩展名，但是CMake可以识别更多的扩展名。如果您的项目中的文件扩展名，由于某种原因不在可识别的扩展名之列，该怎么办？源文件属性可以用来告诉CMake在特定的源文件上使用哪个编译器，就像这样：

```
1. set_source_files_properties(my_source_file..hxx
2.   PROPERTIES
3.     LANGUAGE CXX
4. )
```

那链接器呢？CMake如何确定目标的链接器语言？对于不混合编程语言的目标很简单：通过生成目标文件的编译器命令调用链接器即可。如果目标混合了多个语言，就像示例中一样，则根据在语言混合中，优先级最高的语言来选择链接器语言。比如，我们的示例中混合了Fortran和C，因此Fortran语言比C语言具有更高的优先级，因此使用Fortran用作链接器语言。当混合使用Fortran和C++时，后者具有更高的优先级，因此C++被用作链接器语言。就像编译器语言一样，我们可以通过目标相应的 `LINKER_LANGUAGE` 属性，强制CMake为我们的目标使用特定的链接器语言：

```
1. set_target_properties(my_target
2.   PROPERTIES
3.     LINKER_LANGUAGE Fortran
4. )
```

9.2 使用Fortran库构建C/C++项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-02> 中找到，其中有一个示例：一个是C++、C和Fortran的混例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

第3章第4节，展示了如何检测Fortran编写的BLAS和LAPACK线性代数库，以及如何在C++代码中使用它们。这里，将重新讨论这个方式，但这次的角度有所不同：较少地关注检测外部库，会更深入地讨论混合C++和Fortran的方面，以及名称混乱的问题。

准备工作

本示例中，我们将重用第3章第4节源代码。虽然，我们不会修改源码或头文件，但我们会按照第7章“结构化项目”中，讨论的建议修改项目树结构，并得到以下源代码结构：

```

1. .
2. └── CMakeLists.txt
3. └── README.md
4. └── src
5.     ├── CMakeLists.txt
6.     ├── linear-algebra.cpp
7.     └── math
8.         ├── CMakeLists.txt
9.         ├── CxxBLAS.cpp
10.        ├── CxxBLAS.hpp
11.        ├── CxxLAPACK.cpp
12.        └── CxxLAPACK.hpp

```

这里，收集了BLAS和LAPACK的所有包装器，它们提供了 `src/math` 下的数学库了，主要程序为 `linear-algebra.cpp`。因此，所有源都在 `src` 子目录下。我们还将CMake代码分割为三个 `CMakeLists.txt` 文件，现在来讨论这些文件。

具体实施

这个项目混合了C++(作为该示例的主程序语言)和C(封装Fortran子例程所需的语言)。在根目录下的 `CMakeLists.txt` 文件中，我们需要做以下操作：

1. 声明一个混合语言项目，并选择C++标准：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-02 LANGUAGES CXX C Fortran)
4.
5. set(CMAKE_CXX_STANDARD 11)
6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 使用 `GNUInstallDirs` 模块来设置CMake将静态和动态库，以及可执行文件保存的标准目录。我们还指示CMake将Fortran编译的模块文件放在 `modules` 目录下：

```

1. include(GNUInstallDirs)
2. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
3.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
4. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
5.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
6. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
7.     ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
8. set(CMAKE_Fortran_MODULE_DIRECTORY ${PROJECT_BINARY_DIR}/modules)

```

3. 然后，进入下一个子目录：

```
1. add_subdirectory(src)
```

子文件 `src/CMakeLists.txt` 添加了另一个目录 `math`，其中包含线性代数包装器。在 `src/math/CMakeLists.txt` 中，我们需要以下操作：

1. 调用 `find_package` 来获取BLAS和LAPACK库的位置：

```

1. find_package(BLAS REQUIRED)
2. find_package(LAPACK REQUIRED)

```

2. 包含 `FortranCInterface.cmake` 模块，并验证Fortran、C和C++编译器是否兼容：

```

1. include(FortranCInterface)
2. FortranCInterface_VERIFY(CXX)

```

3. 我们还需要生成预处理器宏来处理BLAS和LAPACK子例程的名称问题。同样，`FortranCInterface` 通过在当前构建目录中生成一个名为 `fc_mangl.h` 的头文件来提供协助：

```

1.  FortranCInterface_HEADER(
2.      fc_mangle.h
3.      MACRO_NAMESPACE "FC_"
4.      SYMBOLS DSCAL DGESV
5.  )

```

4. 接下来，添加了一个库，其中包含BLAS和LAPACK包装器的源代码。我们还指定要找到头文件和库的目录。注意 `PUBLIC` 属性，它允许其他依赖于 `math` 的目标正确地获得它们的依赖关系：

```

1. add_library(math "")
2.
3. target_sources(math
4.     PRIVATE
5.     CxxBLAS.cpp
6.     CxxLAPACK.cpp
7. )
8.
9. target_include_directories(math
10.    PUBLIC
11.    ${CMAKE_CURRENT_SOURCE_DIR}
12.    ${CMAKE_CURRENT_BINARY_DIR}
13. )
14. target_link_libraries(math
15.    PUBLIC
16.    ${LAPACK_LIBRARIES}
17. )

```

回到 `src/CMakeLists.txt`，我们最终添加了一个可执行目标，并将其链接到BLAS/LAPACK包装器的数学库：

```

1. add_executable(linear-algebra "")
2.
3. target_sources(linear-algebra
4.     PRIVATE
5.     linear-algebra.cpp
6. )
7.
8. target_link_libraries(linear- algebra
9.     PRIVATE
10.    math
11. )

```

工作原理

使用 `find_package` 确定了要链接到的库。方法和之前一样，需要确保程序能够正确地调用它们定义的函数。第3章第4节中，我们面临的问题是编译器的名称符号混乱。我们使用 `FortranCInterface` 模块来检查所选的C和C++编译器与Fortran编译器的兼容性。我们还使用 `FortranCInterface_HEADER` 函数生成带有宏的头文件，以处理Fortran子例程的名称混乱。并通过以下代码实现：

```

1. FortranCInterface_HEADER(
2.   fc_mangle.h
3.   MACRO_NAMESPACE "FC_"
4.   SYMBOLS DSCAL DGESV
5. )

```

这个命令将生成 `fc_mangle.h` 头文件，其中包含从Fortran编译器推断的名称混乱宏，并将其保存到当前二进制目录 `CMAKE_CURRENT_BINARY_DIR` 中。我们小心地将 `CMAKE_CURRENT_BINARY_DIR` 设置为数学目标的包含路径。生成的 `fc_mangle.h` 如下：

```

1. #ifndef FC_HEADER_INCLUDED
2. #define FC_HEADER_INCLUDED
3.
4. /* Mangling for Fortran global symbols without underscores. */
5. #define FC_GLOBAL(name,NAME) name##_
6.
7. /* Mangling for Fortran global symbols with underscores. */
8. #define FC_GLOBAL_(name,NAME) name##_
9.
10. /* Mangling for Fortran module symbols without underscores. */
11. #define FC_MODULE(mod_name,name, mod_NAME,NAME) __##mod_name##_MOD_##name
12.
13. /* Mangling for Fortran module symbols with underscores. */
14. #define FC_MODULE_(mod_name,name, mod_NAME,NAME) __##mod_name##_MOD_##name
15.
16. /* Mangle some symbols automatically. */
17. #define DSCAL FC_GLOBAL(dscal, DSCAL)
18. #define DGESV FC_GLOBAL(dgesv, DGESV)
19. #endif

```

本例中的编译器使用下划线进行错误处理。由于Fortran不区分大小写，子例程可能以小写或大写出现，这就说明将这两种情况传递给宏的必要性。注意，CMake还将为隐藏在Fortran模块后面的符号生成宏。

NOTE: 现在，*BLAS*和*LAPACK*的许多实现都在*Fortran*子例程附带了一个*C*的包装层。这些包装器已经标准化，分别称为*CBLAS*和*LAPACKE*。

由于已经将源组织成库目标和可执行目标，所以我们应该对目标的 `PUBLIC`、`INTERFACE` 和 `PRIVATE` 可见性属性的使用进行评论。与源文件一样，包括目录、编译定义和选项，当与 `target_link_libraries` 一起使用时，这些属性的含义是相同的：

- 使用 `PRIVATE` 属性，库将只链接到当前目标，而不链接到使用它的任何其他目标。
- 使用 `INTERFACE` 属性，库将只链接到使用当前目标作为依赖项的目标。
- 使用 `PUBLIC` 属性，库将被链接到当前目标，以及将其作为依赖项使用的任何其他目标。

9.3 使用Cython构建C++和Python项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-03> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Cython是一个静态编译器，它允许为Python编写C扩展。Cython是一个非常强大的工具，使用Cython编程语言(基于Pyrex)。Cython的一个典型用例是加快Python代码的速度，它也可以用于通过Cython层使Python与C(++)接口对接。本示例中，我们将重点介绍后一种用例，并演示如何在CMake的帮助下使用Cython与C(++)和Python进行对接。

准备工作

我们将使用以下C++代码(`account.cpp`)：

```
1. #include "account.hpp"
2. Account::Account() : balance(0.0) {}
3. Account::~Account() {}
4. void Account::deposit(const double amount) { balance += amount; }
5. void Account::withdraw(const double amount) { balance -= amount; }
6. double Account::get_balance() const { return balance; }
```

代码提供了以下接口(`account.hpp`)：

```
1. #pragma once
2.
3. class Account {
4. public:
5.     Account();
6.     ~Account();
7.
8.     void deposit(const double amount);
9.     void withdraw(const double amount);
10.    double get_balance() const;
11.
12. private:
13.     double balance;
14. };
```

使用这个示例代码，我们可以创建余额为零的银行帐户。可以在帐户上存款和取款，还可以使

用 `get_balance()` 查询帐户余额。余额本身是 `Account` 类的私有成员。

我们的目标是能够直接从Python与这个C++类进行交互。换句话说，在Python方面，我们希望能够做到这一点：

```

1. account = Account()
2.
3. account.deposit(100.0)
4. account.withdraw(50.0)
5.
6. balance = account.get_balance()

```

为此，需要一个Cython接口文件(调用 `account.pyx`)：

```

1. # describe the c++ interface
2. cdef extern from "account.hpp":
3.     cdef cppclass Account:
4.         Account() except +
5.         void deposit(double)
6.         void withdraw(double)
7.         double get_balance()
8.
9. # describe the python interface
10. cdef class pyAccount:
11.     cdef Account *thisptr
12.     def __cinit__(self):
13.         self.thisptr = new Account()
14.     def __dealloc__(self):
15.         del self.thisptr
16.     def deposit(self, amount):
17.         self.thisptr.deposit(amount)
18.     def withdraw(self, amount):
19.         self.thisptr.withdraw(amount)
20.     def get_balance(self):
21.         return self.thisptr.get_balance()

```

具体实施

如何生成Python接口：

1. `CMakeLists.txt` 定义CMake依赖项、项目名称和语言：

```

1. # define minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3. # project name and supported language
4. project(recipe-03 LANGUAGES CXX)
5. # require C++11
6. set(CMAKE_CXX_STANDARD 11)
7. set(CMAKE_CXX_EXTENSIONS OFF)
8. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. Windows上，最好不要保留未定义的构建类型，这样我们就可以将该项目的构建类型与Python环境的构建类型相匹配。这里我们认为Release类型：

```

1. if(NOT CMAKE_BUILD_TYPE)
2.     set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
3. endif()

```

3. 在示例中，还需要Python解释器：

```
1. find_package(PythonInterp REQUIRED)
```

4. 下面的CMake代码将构建Python模块：

```

1. # directory containing UseCython.cmake and FindCython.cmake
2. list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake-cython)
3.
4. # this defines cython_add_module
5. include(UseCython)
6.
7. # tells UseCython to compile this file as a c++ file
8. set_source_files_properties(account.pyx PROPERTIES CYTHON_IS_CXX TRUE)
9.
10. # create python module
11. cython_add_module(account account.pyx account.cpp)
12.
13. # location of account.hpp
14. target_include_directories(account
15.     PRIVATE
16.     ${CMAKE_CURRENT_SOURCE_DIR}
17. )

```

5. 定义一个测试：

```

1. # turn on testing
2. enable_testing()
3.
4. # define test
5. add_test(
6.     NAME
7.     python_test
8.     COMMAND
9.     ${CMAKE_COMMAND} -E env
10.    ACCOUNT_MODULE_PATH=${<TARGET_FILE_DIR:account>}
11.    ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py
12. )

```

6. `python_test` 执行 `test.py`，这里进行一些存款和取款操作，并验证余额：

```

1. import os
2. import sys
3. sys.path.append(os.getenv('ACCOUNT_MODULE_PATH'))
4.
5. from account import pyAccount as Account
6.
7. account1 = Account()
8.
9. account1.deposit(100.0)
10. account1.deposit(100.0)
11.
12. account2 = Account()
13.
14. account2.deposit(200.0)
15. account2.deposit(200.0)
16.
17. account1.withdraw(50.0)
18.
19. assert account1.get_balance() == 150.0
20. assert account2.get_balance() == 400.0

```

7. 有了这个，我们就可以配置、构建和测试代码了：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .

```

```

5. $ ctest
6.
7. Start 1: python_test
8. 1/1 Test #1: python_test ..... Passed 0.03 sec
9. 100% tests passed, 0 tests failed out of 1
10. Total Test time (real) = 0.03 sec

```

工作原理

本示例中，使用一个相对简单的 `CMakeLists.txt` 文件对接了Python和C++，但是是通过使用 `FindCython.cmake` 进行的实现。`UseCython.cmake` 模块，放置在 `cmake-cython` 下。这些模块包括使用以下代码：

```

1. # directory contains UseCython.cmake and FindCython.cmake
2. list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake-cython)
3.
4. # this defines cython_add_module
5. include(UseCython)

```

`FindCython.cmake` 包含在 `UseCython.cmake` 中，并定义了 `${CYTHON_EXECUTABLE}` 变量。后一个模块定义了 `cython_add_module` 和 `cython_add_standalone_executable` 函数，它们分别用于创建Python模块和独立的可执行程序。这两个模块都可从 <https://github.com/thewtex/cython-cmake-example/tree/master/cmake> 下载。

这个示例中，使用 `cython_add_module` 创建一个Python模块库。注意，将使用非标准的 `CYTHON_IS_CXX` 源文件属性设置为 `TRUE`，以便 `cython_add_module` 函数知道如何将 `pyx` 作为 `C++` 文件进行编译：

```

1. # tells UseCython to compile this file as a c++ file
2. set_source_files_properties(account.pyx PROPERTIES CYTHON_IS_CXX TRUE)
3.
4. # create python module
5. cython_add_module(account account.pyx account.cpp)

```

Python模块在 `${CMAKE_CURRENT_BINARY_DIR}` 中创建，为了让Python的 `test.py` 脚本找到它，我们使用一个自定义环境变量传递相关的路径，该环境变量用于在 `test.py` 中设置 `path` 变量。请注意，如何将命令设置为调用CMake可执行文件本身，以便在执行Python脚本之前设置本地环境。这为我们提供了平台独立性，并避免了环境污染：

```

1. add_test(

```

```

2. NAME
3.     python_test
4. COMMAND
5.     ${CMAKE_COMMAND} -E env ACCOUNT_MODULE_PATH=$<TARGET_FILE_DIR:account>
6.     ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py
7. )

```

我们来看看 `account.pyx` 文件，这是Python与C++之间的接口文件，并对C++接口进行描述：

```

1. # describe the c++ interface
2. cdef extern from "account.hpp":
3.     cdef cppclass Account:
4.         Account() except +
5.         void deposit(double)
6.         void withdraw(double)
7.         double get_balance()

```

可以看到 `cinit` 构造函数、`__dealloc__` 析构函数以及 `deposit` 和 `withdraw` 方法是如何与对应的C++实现相匹配的。

总之，发现了一种机制，通过引入对Cython模块的依赖来耦合Python和C++。该模块可以通过 `pip` 安装到虚拟环境或Pipenv中，或者使用Anaconda来安装。

更多信息

C语言可以进行类似地耦合。如果希望利用构造函数和析构函数，我们可以在C接口之上封装一个C++层。

类型化Memoryview提供了有趣的功能，可以映射和访问由C/C++直接在Python中分配的内存，而不需要任何创

建：<http://cython.readthedocs.io/en/latest/src/userguide/memoryviews.html> 。它们使得将NumPy数组直接映射为C++数组成为可能。

9.4 使用Boost.Python构建C++和Python项目

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-04> 中找到，其中有一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Boost库为C++代码提供了Python接口。本示例将展示如何在依赖于Boost的C++项目中使用CMake，之后将其作为Python模块发布。我们将重用前面的示例，并尝试用Cython示例中的C++实现(`account.cpp`)进行交互。

准备工作

保持`account.cpp`不变的同时，修改前一个示例中的接口文件(`account.hpp`)：

```

1. #pragma once
2.
3. #define BOOST_PYTHON_STATIC_LIB
4. #include <boost/python.hpp>
5.
6. class Account
7. {
8. public:
9.     Account();
10.    ~Account();
11.    void deposit(const double amount);
12.    void withdraw(const double amount);
13.    double get_balance() const;
14.
15. private:
16.    double balance;
17. };
18.
19. namespace py = boost::python;
20.
21. BOOST_PYTHON_MODULE(account)
22. {
23.     py::class_<Account>("Account")
24.         .def("deposit", &Account::deposit)
25.         .def("withdraw", &Account::withdraw)
26.         .def("get_balance", &Account::get_balance);

```

27. }

具体实施

如何在C++项目中使用Boost.Python的步骤：

1. 和之前一样，首先定义最低版本、项目名称、支持语言和默认构建类型：

```

1. # define minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and supported language
5. project(recipe-04 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # we default to Release build type
13. if(NOT CMAKE_BUILD_TYPE)
14.     set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
15. endif()

```

2. 本示例中，依赖Python和Boost库，以及使用Python进行测试。Boost.Python组件依赖于Boost版本和Python版本，因此需要对这两个组件的名称进行检测：

```

1. # for testing we will need the python interpreter
2. find_package(PythonInterp REQUIRED)
3.
4. # we require python development headers
    find_package(PythonLibs ${PYTHON_VERSION_MAJOR}.${PYTHON_VERSION_MINOR}
5. EXACT REQUIRED)
6.
7. # now search for the boost component
8. # depending on the boost version it is called either python,
9. # python2, python27, python3, python36, python37, ...
10.
11. list(
12.     APPEND _components
13.     python${PYTHON_VERSION_MAJOR}${PYTHON_VERSION_MINOR}

```

```

14.     python${PYTHON_VERSION_MAJOR}
15.     python
16.   )
17.
18. set(_boost_component_found "")
19.
20. foreach(_component IN ITEMS ${_components})
21.   find_package(Boost COMPONENTS ${_component})
22.   if(Boost_FOUND)
23.     set(_boost_component_found ${_component})
24.     break()
25.   endif()
26. endforeach()
27.
28. if(_boost_component_found STREQUAL "")
29.   message(FATAL_ERROR "No matching Boost.Python component found")
30. endif()

```

3. 使用以下命令，定义Python模块及其依赖项：

```

1. # create python module
2. add_library(account
3.   MODULE
4.   account.cpp
5.   )
6.
7. target_link_libraries(account
8.   PUBLIC
9.   Boost::${_boost_component_found}
10.  ${PYTHON_LIBRARIES}
11.  )
12.
13. target_include_directories(account
14.   PRIVATE
15.   ${PYTHON_INCLUDE_DIRS}
16.   )
17.
18. # prevent cmake from creating a "lib" prefix
19. set_target_properties(account
20.   PROPERTIES
21.   PREFIX ""
22.   )

```

```

23.
24. if(WIN32)
25.   # python will not import dll but expects pyd
26.   set_target_properties(account
27.     PROPERTIES
28.       SUFFIX ".pyd"
29.   )
30. endif()

```

4. 最后，定义了一个测试：

```

1. # turn on testing
2. enable_testing()
3.
4. # define test
5. add_test(
6.   NAME
7.   python_test
8.   COMMAND
9.   ${CMAKE_COMMAND} -E env
10.  ACCOUNT_MODULE_PATH=${TARGET_FILE_DIR:account}
11.  ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/test.py
12. )

```

5. 配置、编译和测试：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
7. Start 1: python_test
8. 1/1 Test #1: python_test ..... Passed 0.10 sec
9. 100% tests passed, 0 tests failed out of 1
10. Total Test time (real) = 0.11 sec

```

工作原理

现在，不依赖于Cython模块，而是依赖于在系统上的Boost库，以及Python的开发头文件和库。

Python的开发头文件和库的搜索方法如下：

```

1. find_package(PythonInterp REQUIRED)
   find_package(PythonLibs ${PYTHON_VERSION_MAJOR}.${PYTHON_VERSION_MINOR} EXACT
2. REQUIRED)
```

首先搜索解释器，然后搜索开发头和库。此外，对 `PythonLibs` 的搜索要求开发头文件和库的主版本和次版本，与解释器的完全相同。但是，命令组合不能保证找到完全匹配的版本。

定位Boost.Python时，我们试图定位的组件的名称既依赖于Boost版本，也依赖于我们的Python环境。根据Boost版本的不同，可以调用python、python2、python3、python27、python36、python37等等。我们从特定的名称搜索到更通用的名称，已经解决了这个问题，只有在没有找到匹配的名称时才会失败：

```

1. list(
2.     APPEND _components
3.     python${PYTHON_VERSION_MAJOR}${PYTHON_VERSION_MINOR}
4.     python${PYTHON_VERSION_MAJOR}
5.     python
6. )
7.
8. set(_boost_component_found "")
9.
10. foreach(_component IN ITEMS ${_components})
11.     find_package(Boost COMPONENTS ${_component})
12.     if(Boost_FOUND)
13.         set(_boost_component_found ${_component})
14.         break()
15.     endif()
16. endforeach()
17.
18. if(_boost_component_found STREQUAL "")
19.     message(FATAL_ERROR "No matching Boost.Python component found")
20. endif()
```

可以通过设置额外的CMake变量，来调整Boost库的使用方式。例如，CMake提供了以下选项：

- `Boost_USE_STATIC_LIBS`: 设置为ON之后，可以使用静态版本的Boost库。
- `Boost_USE_MULTITHREADED`: 设置为ON之后，可以切换成多线程版本。
- `Boost_USE_STATIC_RUNTIME`: 设置为ON之后，可以在C++运行时静态的连接不同版本的Boost库。

此示例的另一个特点是使用 `add_library` 的模块选项。我们已经从第1章第3节了解到，CMake接受以下选项作为 `add_library` 的第二个有效参数：

- **STATIC**: 创建静态库，也就是对象文件的存档，用于链接其他目标时使用，例如：可执行文件
- **SHARED**: 创建共享库，也就是可以动态链接并在运行时加载的库
- **OBJECT**: 创建对象库，也就是对象文件不需要将它们归档到静态库中，也不需要将它们链接到共享对象中

`MODULE` 选项将生成一个插件库，也就是动态共享对象(DSO)，没有动态链接到任何可执行文件，但是仍然可以在运行时加载。由于我们使用C++来扩展Python，所以Python解释器需要能够在运行时加载我们的库。使用 `MODULE` 选项进行 `add_library`，可以避免系统在库名前添加前缀(例如：Unix系统上的lib)。后一项操作是通过设置适当的目标属性来执行的，如下所示：

```
1. set_target_properties(account
2.   PROPERTIES
3.     PREFIX ""
4. )
```

完成Python和C++接口的示例，需要向Python代码描述如何连接到C++层，并列出对Python可见的符号，我们也有可能重新命名这些符号。在上一个示例中，我们在另一个单独的 `account.pyx` 文件这样用过。当使用Boost.Python时，我们直接用C++代码描述接口，理想情况下接近期望的接口类或函数定义：

```
1. BOOST_PYTHON_MODULE(account) {
2.     py::class_<Account>("Account")
3.     .def("deposit", &Account::deposit)
4.     .def("withdraw", &Account::withdraw)
5.     .def("get_balance", &Account::get_balance);
6. }
```

`BOOST_PYTHON_MODULE` 模板包含在 `<boost/python>` 中，负责创建Python接口。该模块将公开一个 `Account` Python类，该类映射到C++类。这种情况下，我们不需要显式地声明构造函数和析构函数——编译器会有默认实现，并在创建Python对象时自动调用：

```
1. myaccount = Account()
```

当对象超出范围并被回收时，将调用析构函数。另外，观察 `BOOST_PYTHON_MODULE` 如何声明 `deposit`、`withdraw` 和 `get_balance` 函数，并将它们映射为相应的C++类方法。

这样，Python可以在 `PYTHONPATH` 中找到编译后的模块。这个示例中，我们实现了Python和C++层之间相对干净的分离。Python代码的功能不受限制，不需要类型注释或重写名称，并保持Python风

格：

```
1. from account import Account
2.
3. account1 = Account()
4.
5. account1.deposit(100.0)
6. account1.deposit(100.0)
7.
8. account2 = Account()
9.
10. account2.deposit(200.0)
11. account2.deposit(200.0)
12.
13. account1.withdraw(50.0)
14.
15. assert account1.get_balance() == 150.0
16. assert account2.get_balance() == 400.0
```

更多信息

这个示例中，我们依赖于系统上安装的Boost，因此CMake代码会尝试检测相应的库。或者，可以将Boost源与项目一起提供，并将此依赖项，作为项目的一部分构建。Boost使用的是种可移植的方式将Python与C(++)进行连接。然而，与编译器支持和C++标准相关的可移植性是有代价的，因为Boost.Python不是轻量级依赖项。在接下来的示例中，我们将讨论Boost.Python的轻量级替代方案。

9.5 使用pybind11构建C++和Python项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-05> 中找到，其中有一个C++示例。该示例在CMake 3.11版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的示例中，我们使用Boost.Python与C(C++)接口。本示例中，我们将尝试使用pybind11将Python与C++接口。其实现利用了C++11的特性，因此需要支持C++11的编译器。我们将演示在配置时如何获取pybind11依赖和构建我们的项目，包括一个使用FetchContent方法的Python接口，我们在第4章第3节和第8章第4节中有过讨论。在第11章第2节时，会通过PyPI发布一个用CMake/pybind11构建的C++/Python项目。届时将重新讨论这个例子，并展示如何打包它，使它可以用pip安装。

准备工作

我们将保持 `account.cpp` 不变，只修改 `account.cpp`：

```

1. #pragma once
2. #include <pybind11/pybind11.h>
3. class Account
4. {
5. public:
6.     Account();
7.     ~Account();
8.     void deposit(const double amount);
9.     void withdraw(const double amount);
10.    double get_balance() const;
11.
12. private:
13.    double balance;
14. };
15. namespace py = pybind11;
16. PYBIND11_MODULE(account, m)
17. {
18.     py::class_<Account>(m, "Account")
19.         .def(py::init())
20.         .def("deposit", &Account::deposit)
21.         .def("withdraw", &Account::withdraw)
22.         .def("get_balance", &Account::get_balance);
23. }
```

按照pybind11文档的方式，通过CMake构建

(<https://pybind11.readthedocs.io/en/stable/compile>)。并使用 `add_subdirectory` 将pybind11导入项目。但是，不会将pybind11源代码显式地放到项目目录中，而是演示如何在配置时使用 `FetchContent` (<https://cmake.org/cmake/help/v3.11/module/FetchContent.html>)。

为了在下一个示例中更好地重用代码，我们还将把所有源代码放到子目录中，并使用下面的项目布局：

```

1. .
2. └── account
3.   ├── account.cpp
4.   ├── account.hpp
5.   ├── CMakeLists.txt
6.   └── test.py
7. └── CMakeLists.txt

```

具体实施

让我们详细分析一下这个项目中，各个 `CMakeLists.txt` 文件的内容：

1. 主 `CMakeLists.txt` 文件：

```

1. # define minimum cmake version
2. cmake_minimum_required(VERSION 3.11 FATAL_ERROR)
3.
4. # project name and supported language
5. project(recipe-05 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 这个文件中，查询了用于测试的Python解释器：

```
1. find_package(PythonInterp REQUIRED)
```

3. 然后，包含 `account` 子目录：

```
1. add_subdirectory(account)
```

4. 定义单元测试：

```

1. # turn on testing
2. enable_testing()
3.
4. # define test
5. add_test(
6.   NAME
7.   python_test
8.   COMMAND
9.   ${CMAKE_COMMAND} -E env ACCOUNT_MODULE_PATH=${TARGET_FILE_DIR:account}
10.  ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/account/test.py
11. )

```

5. account/CMakeLists.txt 中，在配置时获取pybind11的源码：

```

1. include(FetchContent)
2.
3. FetchContent_Declare(
4.   pybind11_sources
5.   GIT_REPOSITORY https://github.com/pybind/pybind11.git
6.   GIT_TAG v2.2
7. )
8.
9. FetchContent_GetProperties(pybind11_sources)
10.
11. if(NOT pybind11_sources_POPULATED)
12.   FetchContent_Populate(pybind11_sources)
13.
14. add_subdirectory(
15.   ${pybind11_sources_SOURCE_DIR}
16.   ${pybind11_sources_BINARY_DIR}
17. )
18. endif()

```

6. 最后，定义Python模块。再次使用模块选项 `add_library` 。并将库目标的前缀和后缀属性设置为 `PYTHON_MODULE_PREFIX` 和 `PYTHON_MODULE_EXTENSION` ，这两个值由pybind11适当地推断出来：

```

1. add_library(account
2.   MODULE
3.   account.cpp

```

```

4.      )
5.
6. target_link_libraries(account
7.   PUBLIC
8.   pybind11::module
9.   )
10.
11. set_target_properties(account
12.   PROPERTIES
13.   PREFIX "${PYTHON_MODULE_PREFIX}"
14.   SUFFIX "${PYTHON_MODULE_EXTENSION}"
15.   )

```

7. 进行测试：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
7. Start 1: python_test
8. 1/1 Test #1: python_test ..... Passed 0.04 sec
9. 100% tests passed, 0 tests failed out of 1
10. Total Test time (real) = 0.04 sec

```

工作原理

pybind11的功能和使用与Boost.Python非常类似。pybind11是一个更轻量级的依赖—不过需要编译器支持C++11。`account.hpp` 中的接口定义与之前的示例非常类似：

```

1. #include <pybind11/pybind11.h>
2. // ...
3. namespace py = pybind11;
4. PYBIND11_MODULE(account, m)
5. {
6.     py::class_<Account>(m, "Account")
7.         .def(py::init())
8.         .def("deposit", &Account::deposit)
9.         .def("withdraw", &Account::withdraw)
10.        .def("get_balance", &Account::get_balance);

```

```
11. }
```

同样，我们可以了解到Python方法是如何映射到C++函数的。解释 `PYBIND11_MODULE` 库是在导入的目标 `pybind11::module` 中定义，使用以下代码包括了这个模块：

```
1. add_subdirectory(
2.   ${pybind11_sources_SOURCE_DIR}
3.   ${pybind11_sources_BINARY_DIR}
4. )
```

与之前的示例有两个不同之处：

- 不需要在系统上安装pybind11
- `${pybind11_sources_SOURCE_DIR}` 子目录，包含pybind11的 `CMakeList.txt` 中，在我们开始构建项目时，这个目录并不存在

这个挑战的解决方案是用 `FetchContent`，在配置时获取pybind11源代码和CMake模块，以便可以使用 `add_subdirectory` 引用。使用 `FetchContent` 模式，可以假设pybind11在构建树中可用，并允许构建和链接Python模块：

```
1. add_library(account
2.   MODULE
3.   account.cpp
4. )
5.
6. target_link_libraries(account
7.   PUBLIC
8.   pybind11::module
9. )
```

使用下面的命令，确保Python模块库得到一个定义良好的前缀和后缀，并与Python环境兼容：

```
1. set_target_properties(account
2.   PROPERTIES
3.   PREFIX ${PYTHON_MODULE_PREFIX}
4.   SUFFIX ${PYTHON_MODULE_EXTENSION}
5. )
```

主 `CMakeLists.txt` 文件的其余部分，都在执行测试(与前一个示例使用相同的 `test.py`)。

更多信息

我们可以将pybind11源代码包含在项目源代码存储库中，这将简化CMake结构，并消除在编译时对pybind11源代码进行网络访问的要求。或者，我们可以将pybind11源路径定义为一个Git子模块(<https://git-scm.com/book/en/v2/Git-Tools-Submodules>)，以应对pybind11源依赖项的更新。

在示例中，我们使用 `FetchContent` 解决了这个问题，它提供了一种非常紧凑的方法来引用CMake子项目，而不是显式地跟踪它的源代码。同样，我们也可以使用超级构建的方法来解决这个问题(参见第8章)。

要查看如何简单函数、定义文档注释、映射内存缓冲区等进阶阅读，请参考pybind11文档：<https://pybind11.readthedocs.io>

9.6 使用Python CFFI混合C，C++，Fortran和Python

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-9/recipe-06> 中找到，其中有一个C++示例和一个Fortran示例。该示例在CMake 3.11版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

前面的三个示例中，我们使用Cython、Boost.Python和pybind11作为连接Python和C++的工具。之前的示例中，主要连接的是C++接口。然而，可能会遇到这样的情况：将Python与Fortran或其他语言进行接口。

本示例中，我们将使用Python C的外部函数接口(CFFI，参见<https://cffi.readthedocs.io>)。由于C是通用语言，大多数编程语言(包括Fortran)都能够与C接口进行通信，所以Python CFFI是将Python与大量语言结合在一起的工具。Python CFFI的特性是，生成简单且非侵入性的C接口，这意味着它既不限制语言特性中的Python层，也不会对C层以下的代码有任何限制。

本示例中，将使用前面示例的银行帐户示例，通过C接口将Python CFFI应用于Python和C++。我们的目标是实现一个上下文感知的接口。接口中，我们可以实例化几个银行帐户，每个帐户都带有其内部状态。我们将通过讨论如何使用Python CFFI来连接Python和Fortran来结束本教程。

第11章第3节中，通过PyPI分发一个用CMake/CFFI构建的C/Fortran/Python项目，届时我们将重新讨论这个例子，并展示如何打包它，使它可以用 `pip` 安装。

准备工作

我们从C++实现和接口开始，把它们放在名为 `account/implementation` 的子目录中。实现文件(`cpp_implementation.cpp`)类似于之前的示例，但是包含有断言，因为我们将对象的状态保持在一个不透明的句柄中，所以必须确保对象在访问时已经创建：

```

1. #include "cpp_implementation.hpp"
2.
3. #include <cassert>
4.
5. Account::Account()
6. {
7.     balance = 0.0;
8.     is_initialized = true;
9. }
```

```

10. Account::~Account()
11. {
12.     assert(is_initialized);
13.     is_initialized = false;
14. }
15. void Account::deposit(const double amount)
16. {
17.     assert(is_initialized);
18.     balance += amount;
19. }
20. void Account::withdraw(const double amount)
21. {
22.     assert(is_initialized);
23.     balance -= amount;
24. }
25. double Account::get_balance() const
26. {
27.     assert(is_initialized);
28.     return balance;
29. }

```

接口文件(`cpp_implementation.hpp`)包含如下内容：

```

1. #pragma once
2.
3. class Account
4. {
5. public:
6.     Account();
7.     ~Account();
8.     void deposit(const double amount);
9.     void withdraw(const double amount);
10.    double get_balance() const;
11.
12. private:
13.    double balance;
14.    bool is_initialized;
15. };

```

此外，我们隔离了C-C++接口(`c_cpp_interface.cpp`)。这将是我们与Python CFFI连接的接口：

```

1. #include "account.h"
2. #include "cpp_implementation.hpp"
3.
4. #define AS_TYPE(Type, Obj) reinterpret_cast<Type *>(Obj)
5. #define AS_CTYPE(Type, Obj) reinterpret_cast<const Type *>(Obj)
6.
7. account_context_t *account_new()
8. {
9.     return AS_TYPE(account_context_t, new Account());
10. }
11. void account_free(account_context_t *context) { delete AS_TYPE(Account,
12. context); }
13. void account_deposit(account_context_t *context, const double amount)
14. {
15.     return AS_TYPE(Account, context)->deposit(amount);
16. }
17. void account_withdraw(account_context_t *context, const double amount)
18. {
19.     return AS_TYPE(Account, context)->withdraw(amount);
20. }
21. double account_get_balance(const account_context_t *context)
22. {
23.     return AS_CTYPE(Account, context)->get_balance();
}

```

`account` 目录下，我们声明了C接口(`account.h`)：

```

1. #ifndef ACCOUNT_API
2. #include "account_export.h"
3. #define ACCOUNT_API ACCOUNT_EXPORT
4. #endif
5. #ifdef __cplusplus
6. extern "C"
7. {
8. #endif
9.     struct account_context;
10.    typedef struct account_context account_context_t;
11.    ACCOUNT_API
12.    account_context_t *account_new();
13.    ACCOUNT_API
14.    void account_free(account_context_t *context);
15.    ACCOUNT_API

```

```

16. void account_deposit(account_context_t *context, const double amount);
17. ACCOUNT_API
18. void account_withdraw(account_context_t *context, const double amount);
19. ACCOUNT_API
20. double account_get_balance(const account_context_t *context);
21. #ifdef __cplusplus
22. }
23. #endif
24. #endif /* ACCOUNT_H_INCLUDED */

```

我们还描述了Python接口，将在稍后对此进行讨论(`__init__.py`)：

```

1. from subprocess import check_output
2. from cffi import FFI
3. import os
4. import sys
5. from configparser import ConfigParser
6. from pathlib import Path
7.
8. def get_lib_handle(definitions, header_file, library_file):
9.     ffi = FFI()
10.    command = ['cc', '-E'] + definitions + [header_file]
11.    interface = check_output(command).decode('utf-8')
12.    # remove possible \r characters on windows which
13.    # would confuse cdef
14.    _interface = [l.strip('\r') for l in interface.split('\n')]
15.    ffi.cdef('\n'.join(_interface))
16.    lib = ffi.dlopen(library_file)
17.    return lib
18.
19. # this interface requires the header file and library file
20. # and these can be either provided by interface_file_names.cfg
21. # in the same path as this file
22. # or if this is not found then using environment variables
23. _this_path = Path(os.path.dirname(os.path.realpath(__file__)))
24. _cfg_file = _this_path / 'interface_file_names.cfg'
25. if _cfg_file.exists():
26.     config = ConfigParser()
27.     config.read(_cfg_file)
28.     header_file_name = config.get('configuration', 'header_file_name')
29.     _header_file = _this_path / 'include' / header_file_name
30.     _header_file = str(_header_file)

```

```

31.     library_file_name = config.get('configuration', 'library_file_name')
32.     _library_file = _this_path / 'lib' / library_file_name
33.     _library_file = str(_library_file)
34. else:
35.     _header_file = os.getenv('ACCOUNT_HEADER_FILE')
36.     assert _header_file is not None
37.     _library_file = os.getenv('ACCOUNT_LIBRARY_FILE')
38.     assert _library_file is not None
39.
40.     _lib = get_lib_handle(definitions=['-DACCOUNT_API=' , '-DACCOUNT_NOINCLUDE'],
41.                           header_file=_header_file,
42.                           library_file=_library_file)
43. # we change names to obtain a more pythonic API
44.     new = _lib.account_new
45.     free = _lib.account_free
46.     deposit = _lib.account_deposit
47.     withdraw = _lib.account_withdraw
48.     get_balance = _lib.account_get_balance
49.
50.     __all__ = [
51.         '__version__',
52.         'new',
53.         'free',
54.         'deposit',
55.         'withdraw',
56.         'get_balance',
57.     ]

```

我们看到，这个接口的大部分工作是通用的和可重用的，实际的接口相当薄。

项目的布局为：

```

1. .
2. └── account
3.   ├── account.h
4.   ├── CMakeLists.txt
5.   └── implementation
6.     ├── c_cpp_interface.cpp
7.     ├── cpp_implementation.cpp
8.     └── cpp_implementation.hpp
9.     └── __init__.py
10.    └── test.py

```

11. └── CMakeLists.txt

具体实施

现在使用CMake来组合这些文件，形成一个Python模块：

1. 主 `CMakeLists.txt` 文件包含一个头文件。此外，根据GNU标准，设置编译库的位置：

```

1. # define minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and supported language
5. project(recipe-06 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # specify where to place libraries
13. include(GNUInstallDirs)
14. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
```

2. 第二步，是在 `account` 子目录下包含接口和实现的定义：

```

1. # interface and sources
2. add_subdirectory(account)
```

3. 主 `CMakeLists.txt` 文件以测试定义(需要Python解释器)结束：

```

1. # turn on testing
2. enable_testing()
3.
4. # require python
5. find_package(PythonInterp REQUIRED)
6.
7. # define test
8. add_test(
9.   NAME
10.  python_test
```

```

11.    COMMAND
      ${CMAKE_COMMAND} -E env
12.    ACCOUNT_MODULE_PATH=${CMAKE_CURRENT_SOURCE_DIR}

13.    ACCOUNT_HEADER_FILE=${CMAKE_CURRENT_SOURCE_DIR}/account/account.h
14.          ACCOUNT_LIBRARY_FILE=$<TARGET_FILE:account>
15.    ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/account/test.py
16.  )

```

4. `account/CMakeLists.txt` 中定义了动态库目标:

```

1. add_library(account
2.   SHARED
3.     implementation/c_cpp_interface.cpp
4.     implementation/cpp_implementation.cpp
5.   )
6.
7. target_include_directories(account
8.   PRIVATE
9.   ${CMAKE_CURRENT_SOURCE_DIR}
10.  ${CMAKE_CURRENT_BINARY_DIR}
11. )

```

5. 导出一个可移植的头文件:

```

1. include(GenerateExportHeader)
2. generate_export_header(account
3.   BASE_NAME account
4. )

```

6. 使用Python-C接口进行对接:

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5. $ ctest
6.
7. Start 1: python_test
8. 1/1 Test #1: python_test ..... Passed 0.14 sec
9. 100% tests passed, 0 tests failed out of 1

```

工作原理

虽然，之前的示例要求我们显式地声明Python-C接口，并将Python名称映射到C(++)符号，但Python CFFI从C头文件(示例中是 `account.h`)推断出这种映射。我们只需要向Python CFFI层提供描述C接口的头文件和包含符号的动态库。在主 `CMakeLists.txt` 文件中使用了环境变量集来实现这一点，这些环境变量可以在 `__init__.py` 中找到：

```

1. # ...
2. def get_lib_handle(definitions, header_file, library_file):
3.     ffi = FFI()
4.     command = ['cc', '-E'] + definitions + [header_file]
5.     interface = check_output(command).decode('utf-8')
6.
7.     # remove possible \r characters on windows which
8.     # would confuse cdef
9.     _interface = [l.strip('\r') for l in interface.split('\n')]
10.
11.    ffi.cdef('\n'.join(_interface))
12.    lib = ffi.dlopen(library_file)
13.    return lib
14.
15. # ...
16.
17. _this_path = Path(os.path.dirname(os.path.realpath(__file__)))
18. _cfg_file = _this_path / 'interface_file_names.cfg'
19. if _cfg_file.exists():
20.     # we will discuss this section in chapter 11, recipe 3
21. else:
22.     _header_file = os.getenv('ACCOUNT_HEADER_FILE')
23.     assert _header_file is not None
24.     _library_file = os.getenv('ACCOUNT_LIBRARY_FILE')
25.     assert _library_file is not None
26.     _lib = get_lib_handle(definitions=['-DACCOUNT_API=', '-DACCOUNT_NOINCLUDE'],
27.                           header_file=_header_file,
28.                           library_file=_library_file)
29. # ...

```

`get_lib_handle` 函数打开头文件(使用 `ffi.cdef`)并解析加载库(使用 `ffi.dlopen`)。并返回库对象。前面的文件是通用的，可以在不进行修改的情况下重用，用于与Python和C或使用Python CFFI的其他语言进行接口的其他项目。

`_lib` 库对象可以直接导出，这里有一个额外的步骤，使Python接口在使用时，感觉更像Python：

```

1. # we change names to obtain a more pythonic API
2. new = _lib.account_new
3. free = _lib.account_free
4. deposit = _lib.account_deposit
5. withdraw = _lib.account_withdraw
6. get_balance = _lib.account_get_balance
7.
8. __all__ = [
9.     '__version__',
10.    'new',
11.    'free',
12.    'deposit',
13.    'withdraw',
14.    'get_balance',
15. ]

```

有了这个变化，可以将例子写成下面的方式：

```

1. import account
2. account1 = account.new()
3. account.deposit(account1, 100.0)

```

另一种选择则不那么直观：

```

1. from account import lib
2. account1 = lib.account_new()
3. lib.account_deposit(account1, 100.0)

```

需要注意的是，如何使用API来实例化和跟踪上下文：

```

1. account1 = account.new()
2. account.deposit(account1, 10.0)
3.
4. account2 = account.new()
5. account.withdraw(account1, 5.0)
6. account.deposit(account2, 5.0)

```

为了导入 `account` 的Python模块，需要提供 `ACCOUNT_HEADER_FILE` 和 `ACCOUNT_LIBRARY_FILE` 环境变量，就像测试中那样：

```
1. add_test(
```

```

2. NAME
3.     python_test
4. COMMAND
5.     ${CMAKE_COMMAND} -E env ACCOUNT_MODULE_PATH=${CMAKE_CURRENT_SOURCE_DIR}

6. ACCOUNT_HEADER_FILE=${CMAKE_CURRENT_SOURCE_DIR}/account/account.h
7.             ACCOUNT_LIBRARY_FILE=$<TARGET_FILE:account>
8.     ${PYTHON_EXECUTABLE} ${CMAKE_CURRENT_SOURCE_DIR}/account/test.py
9. )

```

第11章中，将讨论如何创建一个可以用 `pip` 安装的Python包，其中头文件和库文件将安装在定义良好的位置，这样就不必定义任何使用Python模块的环境变量。

讨论了Python方面的接口之后，现在看下C的接口。`account.h` 内容为：

```

1. struct account_context;
2. typedef struct account_context account_context_t;
3.
4. ACCOUNT_API
5. account_context_t *account_new();
6.
7. ACCOUNT_API
8. void account_free(account_context_t *context);
9.
10. ACCOUNT_API
11. void account_deposit(account_context_t *context, const double amount);
12.
13. ACCOUNT_API
14. void account_withdraw(account_context_t *context, const double amount);
15.
16. ACCOUNT_API
17. double account_get_balance(const account_context_t *context);

```

黑盒句柄 `account_context` 会保存对象的状态。`ACCOUNT_API` 定义在 `account_export.h` 中，由 `account/interface/CMakeLists.txt` 生成：

```

1. include(GenerateExportHeader)
2. generate_export_header(account
3.   BASE_NAME account
4. )

```

`account_export.h` 头文件定义了接口函数的可见性，并确保这是以一种可移植的方式完成的，实现

可以在 `cpp_implementation.cpp` 中找到。它包含 `is_initialized` 布尔变量，可以检查这个布尔值确保API函数按照预期的顺序调用：上下文在创建之前或释放之后都不应该被访问。

更多信息

设计Python-C接口时，必须仔细考虑在哪一端分配数组：数组可以在Python端分配并传递给C(++)实现，也可以在返回指针的C(++)实现上分配。后一种方法适用于缓冲区大小事先未知的情况。但返回到分配给C(++)端的数组指针可能会有问题，因为这可能导致Python垃圾收集导致内存泄漏，而Python垃圾收集不会“查看”分配给它的数组。我们建议设计C API，使数组可以在外部分配并传递给C实现。然后，可以在 `__init__.py` 中分配这些数组，如下例所示：

```

1. from cffi import FFI
2. import numpy as np
3.
4. _ffi = FFI()
5.
6. def return_array(context, array_len):
7.     # create numpy array
8.     array_np = np.zeros(array_len, dtype=np.float64)
9.
10.    # cast a pointer to its data
11.    array_p = _ffi.cast("double *", array_np.ctypes.data)
12.
13.    # pass the pointer
14.    _lib.mylib_myfunction(context, array_len, array_p)
15.
16.    # return the array as a list
17.    return array_np.tolist()
```

`return_array` 函数返回一个Python列表。因为在Python端完成了所有的分配工作，所以不必担心内存泄漏，可以将清理工作留给垃圾收集。

对于Fortran示例，读者可以参考以下Git库：<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter09/recipe06/Fortran-example>。与C++实现的主要区别在于，account库是由Fortran 90源文件编译而成的，我们在 `account/CMakeLists.txt` 中使用了Fortran 90源文件：

```

1. add_library(account
2.   SHARED
3.   implementation/fortran_implementation.f90
4. )
```

上下文保存在用户定义的类型中：

```

1. type :: account
2.   private
3.   real(c_double) :: balance
4.   logical :: is_initialized = .false.
5. end type

```

Fortran实现可以使用 `iso_c_binding` 模块解析 `account.h` 中定义的符号和方法：

```

1. module account_implementation
2.
3.   use, intrinsic :: iso_c_binding, only: c_double, c_ptr
4.
5.   implicit none
6.
7.   private
8.
9.   public account_new
10.  public account_free
11.  public account_deposit
12.  public account_withdraw
13.  public account_get_balance
14.
15. type :: account
16.   private
17.   real(c_double) :: balance
18.   logical :: is_initialized = .false.
19. end type
20.
21. contains
22.
23. type(c_ptr) function account_new() bind (c)
24.   use, intrinsic :: iso_c_binding, only: c_loc
25.   type(account), pointer :: f_context
26.   type(c_ptr) :: context
27.
28.   allocate(f_context)
29.   context = c_loc(f_context)
30.   account_new = context
31.   f_context%balance = 0.0d0
32.   f_context%is_initialized = .true.

```

```
33. end function
34.
35. subroutine account_free(context) bind (c)
36.   use, intrinsic :: iso_c_binding, only: c_f_pointer
37.   type(c_ptr), value :: context
38.   type(account), pointer :: f_context
39.
40.   call c_f_pointer(context, f_context)
41.   call check_valid_context(f_context)
42.   f_context%balance = 0.0d0
43.   f_context%is_initialized = .false.
44.   deallocate(f_context)
45. end subroutine
46.
47. subroutine check_valid_context(f_context)
48.   type(account), pointer, intent(in) :: f_context
49.   if (.not. associated(f_context)) then
50.     print *, 'ERROR: context is not associated'
51.     stop 1
52.   end if
53.   if (.not. f_context%is_initialized) then
54.     print *, 'ERROR: context is not initialized'
55.     stop 1
56.   end if
57. end subroutine
58.
59. subroutine account_withdraw(context, amount) bind (c)
60.   use, intrinsic :: iso_c_binding, only: c_f_pointer
61.   type(c_ptr), value :: context
62.   real(c_double), value :: amount
63.   type(account), pointer :: f_context
64.
65.   call c_f_pointer(context, f_context)
66.   call check_valid_context(f_context)
67.   f_context%balance = f_context%balance - amount
68. end subroutine
69.
70. subroutine account_deposit(context, amount) bind (c)
71.   use, intrinsic :: iso_c_binding, only: c_f_pointer
72.   type(c_ptr), value :: context
73.   real(c_double), value :: amount
74.   type(account), pointer :: f_context
```

```
75.  
76.     call c_f_pointer(context, f_context)  
77.     call check_valid_context(f_context)  
78.     f_context%balance = f_context%balance + amount  
79. end subroutine  
80.  
81. real(c_double) function account_get_balance(context) bind (c)  
82.   use, intrinsic :: iso_c_binding, only: c_f_pointer  
83.   type(c_ptr), value, intent(in) :: context  
84.   type(account), pointer :: f_context  
85.  
86.   call c_f_pointer(context, f_context)  
87.   call check_valid_context(f_context)  
88.   account_get_balance = f_context%balance  
89. end function  
90. end module
```

这个示例和解决方案的灵感来自Armin Ronacher的帖子“BeautifulNative Libraries”：

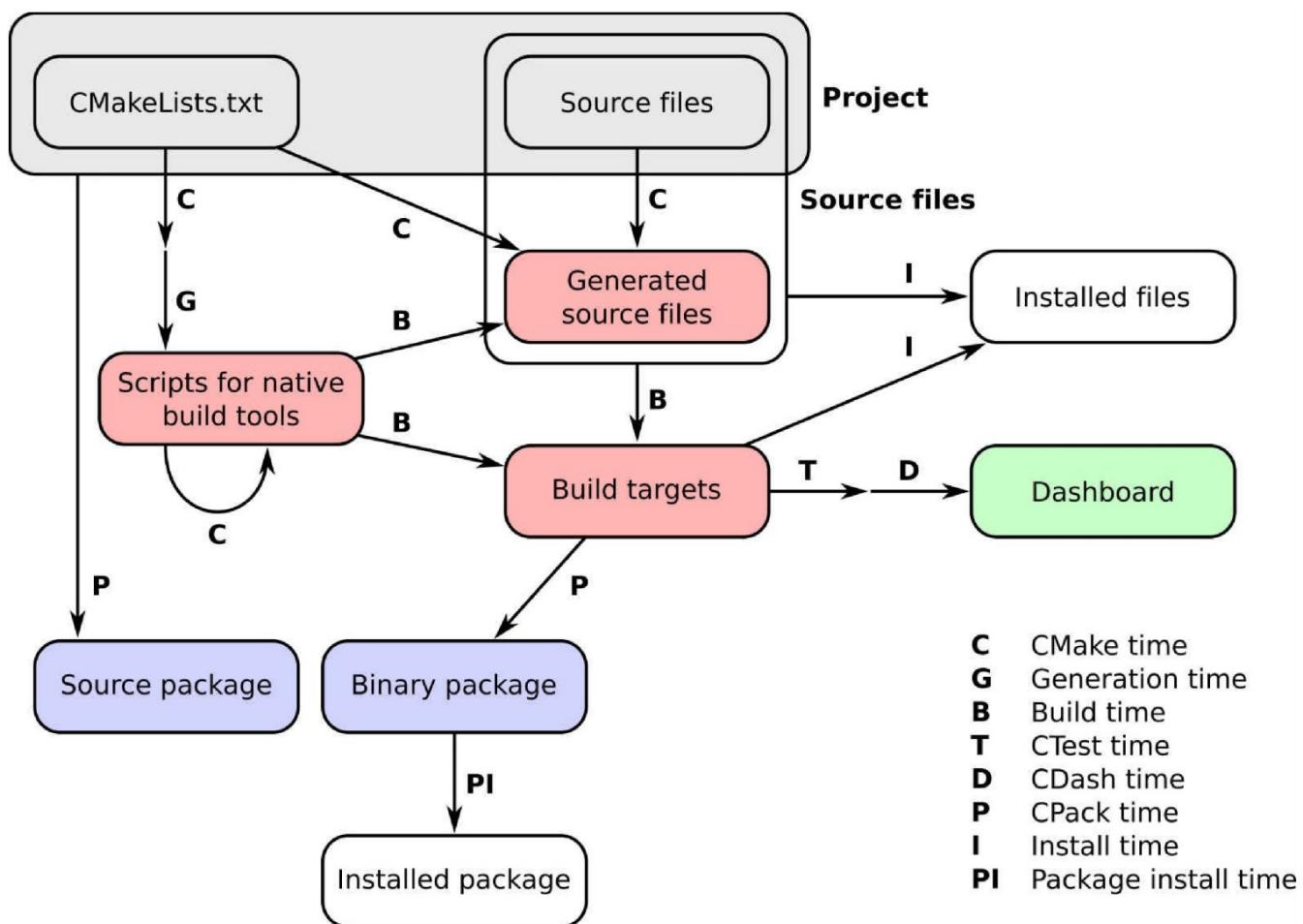
<http://lucumr.pocoo.org/2013/8/18/beautiful-native-libraries/>

第10章 编写安装程序

本章主要内容：

- 安装项目
- 生成输出头文件
- 输出目标
- 安装超级构建

前几章中，我们展示了如何使用CMake配置、构建和测试项目。安装项目是很重要的一部分，本章将演示如何实现这一点。本章涵盖了下图中，安装时的所有操作：



我们将指导完成各个步骤，直到完成安装一个简单的C++项目：从项目中构建的文件，并复制到正确的目录，确保其他项目使用CMake时可以找到该工程的输出目标。本章中的4个示例将建立在第1章第3节的示例基础上。之前，我们试图构建一个非常简单的库，并将其链接到一个可执行文件中。我们还展示了如何使用相同的源文件构建静态库和动态库。本章中，我们将会讨论安装时所发生的事情。

10.1 安装项目

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-10/recipe-01> 中找到，其中有一个C++示例和一个Fortran示例。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

第一个示例中，将介绍我们的小项目和一些基本概念，这些概念也将在后面的示例中使用。安装文件、库和可执行文件是一项非常基础的任务，但是也可能会带来一些缺陷。我们将带您了解这些问题，并展示如何使用CMake有效地避开这些缺陷。

准备工作

第1章第3节的示例，几乎复用：只添加对UUID库的依赖。这个依赖是有条件的，如果没有找到UUID库，我们将通过预处理程序排除使用UUID库的代码。项目布局如下：

```

1. .
2. └── CMakeLists.txt
3. └── src
4.   ├── CMakeLists.txt
5.   ├── hello-world.cpp
6.   ├── Message.cpp
7.   └── Message.hpp
8. └── tests
9.   └── CMakeLists.txt

```

我们已经看到，有三个 `CMakeLists.txt`，一个是主 `CMakeLists.txt`，另一个是位于 `src` 目录下的，还有一个是位于 `test` 目录下的。

`Message.hpp` 头文件包含以下内容：

```

1. #pragma once
2.
3. #include <iostream>
4. #include <string>
5.
6. class Message
7. {
8. public:
9.     Message(const std::string &m) : message_(m) {}

```

```

10.     friend std::ostream &operator<<(std::ostream &os, Message &obj)
11.     {
12.         return obj.printObject(os);
13.     }
14.
15. private:
16.     std::string message_;
17.     std::ostream &printObject(std::ostream &os);
18. };
19.
20. std::string getUUID();

```

Message.cpp 中有相应的实现：

```

1. #include "Message.hpp"
2. #include <iostream>
3. #include <string>
4. #ifdef HAVE_UUID
5. #include <uuid/uuid.h>
6. #endif
7. std::ostream &Message::printObject(std::ostream &os)
8. {
9.     os << "This is my very nice message: " << std::endl;
10.    os << message_ << std::endl;
11.    os << "...and here is its UUID: " << getUUID();
12.    return os;
13. }
14. #ifdef HAVE_UUID
15. std::string getUUID()
16. {
17.     uuid_t uuid;
18.     uuid_generate(uuid);
19.     char uuid_str[37];
20.     uuid_unparse_lower(uuid, uuid_str);
21.     uuid_clear(uuid);
22.     std::string uuid_cxx(uuid_str);
23.     return uuid_cxx;
24. }
25. #else
26. std::string getUUID()
27. {
28.     return "Ooooops, no UUID for you!";

```

```
29. }
30. #endif
```

最后，示例 `hello-world.cpp` 内容如下：

```
1. #include <cstdlib>
2. #include <iostream>
3. #include "Message.hpp"
4. int main()
5. {
6.     Message say_hello("Hello, CMake World!");
7.     std::cout << say_hello << std::endl;
8.     Message say_goodbye("Goodbye, CMake World");
9.     std::cout << say_goodbye << std::endl;
10.    return EXIT_SUCCESS;
11. }
```

具体实施

我们先来看一下主 `CMakeLists.txt` :

1. 声明CMake最低版本，并定义一个C++11项目。请注意，我们已经为我们的项目设置了一个版本，在 `project` 中使用 `VERSION` 进行指定：

```
1. # CMake 3.6 needed for IMPORTED_TARGET option
2. # to pkg_search_module
3. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
4. project(recipe-01
5. LANGUAGES CXX
6. VERSION 1.0.0
7. )
8. # <<< General set up >>>
9. set(CMAKE_CXX_STANDARD 11)
10. set(CMAKE_CXX_EXTENSIONS OFF)
11. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 用户可以通过 `CMAKE_INSTALL_PREFIX` 变量定义安装目录。CMake会给这个变量设置一个默认值：Windows上的 `C:\Program Files` 和Unix上的 `/usr/local`。我们将会打印安装目录的信息：

```
1. message(STATUS "Project will be installed to ${CMAKE_INSTALL_PREFIX}")
```

3. 默认情况下，我们更喜欢以Release的方式配置项目。用户可以通过 `CMAKE_BUILD_TYPE` 设置此变量，从而改变配置类型，我们将检查是否存在这种情况。如果没有，将设置为默认值：

```

1. if(NOT CMAKE_BUILD_TYPE)
2.   set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
3. endif()
4. message(STATUS "Build type set to ${CMAKE_BUILD_TYPE}")

```

4. 接下来，告诉CMake在何处构建可执行、静态和动态库目标。便于在用户不打算安装项目的情况下，访问这些构建目标。这里使用标准CMake的 `GNUInstallDirs.cmake` 模块。这将确保的项目布局的合理性和可移植性：

```

1. include(GNUInstallDirs)
2.
3. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
4.      ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
5. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
6.      ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
7. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
8.      ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

```

5. 虽然，前面的命令配置了构建目录中输出的位置，但是需要下面的命令来配置可执行程序、库以及安装前缀中包含的文件的位置。它们大致遵循相同的布局，但是我们定义了新的 `INSTALL_LIBDIR` 、 `INSTALL_BINDIR` 、 `INSTALL_INCLUDEDIR` 和 `INSTALL_CMAKEDIR` 变量。当然，也可以覆盖这些变量：

```

1. # Offer the user the choice of overriding the installation directories
2. set(INSTALL_LIBDIR ${CMAKE_INSTALL_LIBDIR} CACHE PATH "Installation
3. directory for libraries")
4. set(INSTALL_BINDIR ${CMAKE_INSTALL_BINDIR} CACHE PATH "Installation
5. directory for executables")
6. set(INSTALL_INCLUDEDIR ${CMAKE_INSTALL_INCLUDEDIR} CACHE PATH
7. "Installation directory for header files")
8. if(WIN32 AND NOT CYGWIN)
9.   set(DEF_INSTALL_CMAKEDIR CMake)
10. else()
11.   set(DEF_INSTALL_CMAKEDIR share/cmake/${PROJECT_NAME})
12. endif()
13. set(INSTALL_CMAKEDIR ${DEF_INSTALL_CMAKEDIR} CACHE PATH "Installation
14. directory for CMake files")

```

6. 报告组件安装的路径：

```

1. # Report to user
2. foreach(p LIB BIN INCLUDE CMAKE)
3.   file(TO_NATIVE_PATH ${CMAKE_INSTALL_PREFIX}/${INSTALL_${p}DIR} _path )
4.   message(STATUS "Installing ${p} components to ${_path}")
5.   unset(_path)
6. endforeach()

```

7. 主 `CMakeLists.txt` 文件中的最后一个指令添加 `src` 子目录，启用测试，并添加 `tests` 子目录：

```

1. add_subdirectory(src)
2. enable_testing()
3. add_subdirectory(tests)

```

现在我们继续分析 `src/CMakeLists.txt`，其定义了构建的实际目标：

1. 我们的项目依赖于UUID库：

```

1. # Search for pkg-config and UUID
2. find_package(PkgConfig QUIET)
3. if(PKG_CONFIG_FOUND)
4.   pkg_search_module(UUID uuid IMPORTED_TARGET)
5.   if(TARGET PkgConfig::UUID)
6.     message(STATUS "Found libuuid")
7.     set(UUID_FOUND TRUE)
8.   endif()
9. endif()

```

2. 我们希望建立一个动态库，将该目标声明为 `message-shared`：

```
1. add_library(message-shared SHARED "")
```

3. 这个目标由 `target_sources` 命令指定：

```

1. target_sources(message-shared
2.   PRIVATE
3.   ${CMAKE_CURRENT_LIST_DIR}/Message.cpp
4.   )

```

4. 我们为目标声明编译时定义和链接库。请注意，所有这些都是 **PUBLIC**，以确保所有依赖的目标将正确继承它们：

```

1. target_compile_definitions(message-shared
2. PUBLIC
3.     $<$<BOOL:$UUID_FOUND>:HAVE_UUID>
4. )
5. target_link_libraries(message-shared
6. PUBLIC
7.     $<$<BOOL:$UUID_FOUND>:PkgConfig::UUID>
8. )

```

5. 然后设置目标的附加属性：

```

1. set_target_properties(message-shared
2. PROPERTIES
3.     POSITION_INDEPENDENT_CODE 1
4.     SOVERSION ${PROJECT_VERSION_MAJOR}
5.     OUTPUT_NAME "message"
6.     DEBUG_POSTFIX "_d"
7.     PUBLIC_HEADER "Message.hpp"
8.     MACOSX_RPATH ON
9.     WINDOWS_EXPORT_ALL_SYMBOLS ON
10.    )

```

6. 最后，为“Hello, world”程序添加可执行目标：

```
1. add_executable(hello-world_wDSO hello-world.cpp)
```

7. **hello-world_wDSO** 可执行目标，会链接到动态库：

```

1. target_link_libraries(hello-world_wDSO
2. PUBLIC
3.     message-shared
4. )

```

src/CMakeLists.txt 文件中，还包含安装指令。考虑这些之前，我们需要设置可执行文件的 **RPATH**：

1. 使用CMake路径操作，我们可以设置 **message_RPATH** 变量。这将为GNU/Linux和macOS设置适当的 **RPATH**：

```

1. RPATH
   file(RELATIVE_PATH _rel ${CMAKE_INSTALL_PREFIX}/${INSTALL_BINDIR}
2. ${CMAKE_INSTALL_PREFIX})
3. if(APPLE)
4.     set(_rpath "@loader_path/${_rel}")
5. else()
6.     set(_rpath "\$ORIGIN/${_rel}")
7. endif()
8. file(TO_NATIVE_PATH "${_rpath}/${INSTALL_LIBDIR}" message_RPATH)

```

2. 现在，可以使用这个变量来设置可执行目标 `hello-world_wDSO` 的 `RPATH`（通过目标属性实现）。我们也可以设置额外的属性，稍后会对此进行更多的讨论：

```

1. set_target_properties(hello-world_wDSO
2.   PROPERTIES
3.     MACOSX_RPATH ON
4.     SKIP_BUILD_RPATH OFF
5.     BUILD_WITH_INSTALL_RPATH OFF
6.     INSTALL_RPATH "${message_RPATH}"
7.     INSTALL_RPATH_USE_LINK_PATH ON
8.   )

```

3. 终于可以安装库、头文件和可执行文件了！使用CMake提供的 `install` 命令来指定安装位置。注意，路径是相对的，我们将在后续进一步讨论这一点：

```

1. install(
2.   TARGETS
3.     message-shared
4.     hello-world_wDSO
5.   ARCHIVE
6.     DESTINATION ${INSTALL_LIBDIR}
7.     COMPONENT lib
8.   RUNTIME
9.     DESTINATION ${INSTALL_BINDIR}
10.    COMPONENT bin
11.  LIBRARY
12.    DESTINATION ${INSTALL_LIBDIR}
13.    COMPONENT lib
14.  PUBLIC_HEADER
15.    DESTINATION ${INSTALL_INCLUDEDIR}/message
16.    COMPONENT dev
17.  )

```

`tests` 目录中的 `CMakeLists.txt` 文件包含简单的指令，以确保“Hello, World”可执行文件能够正确运行：

```

1. add_test(
2.   NAME test_shared
3.   COMMAND $<TARGET_FILE:hello-world_wDSO>
4. )

```

现在让我们配置、构建和安装项目，并查看结果。添加安装指令时，CMake就会生成一个名为 `install` 的新目标，该目标将运行安装规则：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake -G"Unix Makefiles" -DCMAKE_INSTALL_PREFIX=$HOME/Software/recipe-01
4. $ cmake --build . --target install

```

GNU/Linux构建目录的内容如下：

```

1. build
2. └── bin
3.   └── hello-world_wDSO
4.   └── CMakeCache.txt
5.   └── CMakeFiles
6.   └── cmake_install.cmake
7.   └── CTestTestfile.cmake
8.   └── install_manifest.txt
9.   └── lib64
10.    └── libmessage.so -> libmessage.so.1
11.    └── libmessage.so.1
12.   └── Makefile
13.   └── src
14.   └── Testing
15.   └── tests

```

另一方面，在安装位置，可以找到如下的目录结构：

```

1. $HOME/Software/recipe-01/
2. └── bin
3.   └── hello-world_wDSO
4.   └── include

```

```

5. |   └ message
6. |     └ Message.hpp
7. └ lib64
8.   └ libmessage.so -> libmessage.so.1
9.   └ libmessage.so.1

```

这意味着安装指令中给出的位置，是相对于用户给定的 `CMAKE_INSTALL_PREFIX` 路径。

工作原理

这个示例有三个要点我们需要更详细地讨论：

- 使用 `GNUInstallDirs.cmake` 定义目标安装的标准位置
- 在动态库和可执行目标上设置的属性，特别是 `RPATH` 的处理
- 安装指令

安装到标准位置

对于项目的安装来说，什么是好的布局呢？如果只有自己使用该项目，那就无所谓好或坏的布局。然而，一旦向外部发布产品，和他人共用该项目，就应该在安装项目时提供一个合理的布局。幸运的是，我们可以遵循一些标准，CMake可以帮助我们做到这一点。实际上，`GNUInstallDirs.cmake` 模块所做的就是定义这样一组变量，这些变量是安装不同类型文件的子目录的名称。在例子中，使用了以下内容：

- `*CMAKE_INSTALL_BINDIR`: 这将用于定义用户可执行文件所在的子目录，即所选安装目录下的 `bin` 目录。
- `CMAKE_INSTALL_LIBDIR`: 这将扩展到目标代码库(即静态库和动态库)所在的子目录。在64位系统上，它是 `lib64`，而在32位系统上，它只是 `lib`。
- `CMAKE_INSTALL_INCLUDEDIR`: 最后，我们使用这个变量为C头文件获取正确的子目录，该变量为 `include`。

然而，用户可能希望覆盖这些选项。我们允许在主 `CMakeLists.txt` 文件中使用以下方式覆盖选项：

```

1. # Offer the user the choice
2. of overriding the installation directories
3. set(INSTALL_LIBDIR ${CMAKE_INSTALL_LIBDIR} CACHE PATH
4. "Installation directory for libraries")
5. set(INSTALL_BINDIR ${CMAKE_INSTALL_BINDIR} CACHE PATH
6. "Installation directory for executables")
7. set(INSTALL_INCLUDEDIR ${CMAKE_INSTALL_INCLUDEDIR} CACHE
8. PATH "Installation directory for header files")

```

这重新定义了在项目中使用的 `INSTALL_BINDIR`、`INSTALL_LIBDIR` 和 `INSTALL_INCLUDEDIR` 变量。我们还定义了 `INSTALL_CMAKEDIR` 变量，但它所扮演的角色将在接下来的几个示例中详细讨论。

TIPS: `GNUInstallDirs.cmake` 模块定义了额外的变量，这些变量将有助于，将已安装的文件放置到所选安装前缀的子目录中。请参考CMake在线文

档:<https://cmake.org/cmake/help/v3.6/module/GNUInstallDirs.html>

目标属性和RPATH处理

让我们更仔细地看看在动态库目标上设置的属性，需要设置以下内容：

- `POSITION_INDEPENDENT_CODE 1`：设置生成位置无关代码所需的编译器标志。有关更多信息，请参考https://en.wikipedia.org/wiki/position-independent_code
- `SOVERSION ${PROJECT_VERSION_MAJOR}`：这是动态库提供的应用程序编程接口(API)版本。在设置语义版本之后，将其设置为与项目的主版本一致。CMake目标也有一个版本属性，可以用来指定目标的构建版本。注意，`SOVERSION` 和 `VERSION` 有所不同：随着时间的推移，提供相同API的多个构建版本。本例中，我们不关心这种的粒度控制：仅使用 `SOVERSION` 属性设置API版本就足够了，CMake将为我们把 `VERSION` 设置为相同的值。相关详细信息，请参考官方文档:https://cmake.org/cmake/help/latest/prop_tgt/SOVERSION.html
- `OUTPUT_NAME "message"`：这告诉CMake库的名称 `message`，而不是目标 `message-shared` 的名称，`libmessage.so.1` 将在构建时生成。从前面给出的构建目录和安装目录的也可以看出，`libmessage.so` 的符号链接也将生成。
- `DEBUG_POSTFIX "_d"`：这告诉CMake，如果我们以Debug配置构建项目，则将 `_d` 后缀添加到生成的动态库。
- `PUBLIC_HEADER "Message.hpp"`：我们使用这个属性来设置头文件列表(本例中只有一个头文件)，声明提供的API函数。这主要用于macOS上的动态库目标，也可以用于其他操作系统和目标。有关详细信息，请参见官方文档:https://cmake.org/cmake/help/v3.6/prop_tgt/PUBLIC_HEADER.html
- `MACOSX_RPATH ON`：这将动态库的 `install_name` 部分(目录)设置为macOS上的 `@rpath`。
- `WINDOWS_EXPORT_ALL_SYMBOLS ON`：这将强制在Windows上编译以导出所有符号。注意，这通常不是一个好的方式，我们将在第2节中展示如何生成导出头文件，以及如何在不同的平台上保证符号的可见性。

现在讨论一下 `RPATH`。我们将 `hello-world_wDSO` 可执行文件链接到 `libmessage.so.1`，这意味着在执行时，将加载动态库。因此，有关库位置的信息需要在某个地方进行编码，以便加载程序能够成功地完成其工作。库的定位有两种方法：

- 通过设置环境变量通知链接器：
 - GNU/Linux上，这需要将路径附加到 `LD_LIBRARY_PATH` 环境变量中。注意，这很可能会污染系统中所有应用程序的链接器路径，并可能导致符号冲突(https://gms.tf/ld_library_path-considered-harmful.htm)。

- macOS上，可以设置 `DYLD_LIBRARY_PATH` 变量。这与GNU/Linux上的 `LD_LIBRARY_PATH` 有相同的问题，可以通过使用 `DYLD_FALLBACK_LIBRARY_PATH` 变量来(部分的)改善这种情况。请看下面的链接，获取相关例子：
<https://stackoverflow.com/a/3172515/2528668>
- 可被编码到可执行文件中，使用 `RPATH` 可以设置可执行文件的运行时搜索路径

后一种方法更健壮。但是，设置动态对象的 `RPATH` 时，应该选择哪个路径？我们需要确保可执行文件总是找到正确的动态库，不管它是在构建树中运行还是在安装树中运行。这需要通过设置 `hello-world_wDSO` 目标的 `RPATH` 相关属性来实现的，通过 `$ORIGIN` (在GNU/Linux上) 或 `@loader_path` (在macOS上) 变量来查找与可执行文件本身位置相关的路径：

```

1. # Prepare RPATH
  file(RELATIVE_PATH _rel ${CMAKE_INSTALL_PREFIX}/${INSTALL_BINDIR}
2. ${CMAKE_INSTALL_PREFIX})
3. if(APPLE)
4.   set(_rpath "@loader_path/${_rel}")
5. else()
6.   set(_rpath "\$ORIGIN/${_rel}")
7. endif()
8. file(TO_NATIVE_PATH "${_rpath}/${INSTALL_LIBDIR}" message_RPATH)

```

当设置了 `message_RPATH` 变量，目标属性将完成剩下的工作：

```

1. set_target_properties(hello-world_wDSO
2.   PROPERTIES
3.   MACOSX_RPATH ON
4.   SKIP_BUILD_RPATH OFF
5.   BUILD_WITH_INSTALL_RPATH OFF
6.   INSTALL_RPATH "${message_RPATH}"
7.   INSTALL_RPATH_USE_LINK_PATH ON
8. )

```

让我们详细研究一下这个命令：

- `SKIP_BUILD_RPATH OFF`：告诉CMake生成适当的 `RPATH`，以便能够在构建树中运行可执行文件。
- `BUILD_WITH_INSTALL_RPATH OFF`：关闭生成可执行目标，使其 `RPATH` 调整为与安装树的 `RPATH` 相同。在构建树中不运行可执行文件。
- `INSTALL_RPATH "${message_RPATH}"`：将已安装的可执行目标的 `RPATH` 设置为先前的路径。
- `INSTALL_RPATH_USE_LINK_PATH ON`：告诉CMake将链接器搜索路径附加到可执行文件的 `RPATH` 中。

NOTE: 加载器在 Unix 系统上如何工作的更多信息，可参见：http://longwei.github.io/rpath_origin/

安装指令

最后，看一下安装指令。我们需要安装一个可执行文件、一个库和一个头文件。可执行文件和库是构建目标，因此我们使用安装命令的 `TARGETS` 选项。可以同时设置多个目标的安装规则：CMake 知道它们是什么类型的目标，无论其是可执行程序库、动态库，还是静态库：

```
1. install(
2.   TARGETS
3.   message-shared
4.   hello-world_wDSO
```

可执行文件将安装在 `RUNTIME DESTINATION`，将其设置为 `${INSTALL_BINDIR}`。动态库安装到 `LIBRARY DESTINATION`，将其设置为 `${INSTALL_LIBDIR}`。静态库将安装到 `ARCHIVE DESTINATION`，将其设置为 `${INSTALL_LIBDIR}`：

```
1. ARCHIVE
2. DESTINATION ${INSTALL_LIBDIR}
3. COMPONENT lib
4. RUNTIME
5. DESTINATION ${INSTALL_BINDIR}
6. COMPONENT bin
7. LIBRARY
8. DESTINATION ${INSTALL_LIBDIR}
9. COMPONENT lib
```

注意，这里不仅指定了 `DESTINATION`，还指定了 `COMPONENT`。使用 `cmake --build . --target install` 安装命令，所有组件会按预期安装完毕。然而，有时只安装其中一些可用的。这就是 `COMPONENT` 关键字帮助我们做的事情。例如，当只要求安装库，我们可以执行以下步骤：

```
1. $ cmake -D COMPONENT=lib -P cmake_install.cmake
```

自从 `Message.hpp` 头文件设置为项目的公共头文件，我们可以使用 `PUBLIC_HEADER` 关键字将其与其他目标安装到选择的目的地： `${INSTALL_INCLUDEDIR}/message`。库用户现在可以包含头文件：`#include <message/Message.hpp>`，这需要在编译时，使用 `-I` 选项将正确的头文件查找路径位置传递给编译器。

安装指令中的各种目标地址会被解释为相对路径，除非使用绝对路径。但是相对于哪里呢？根据不同的安装工具而不同，而 CMake 可以去计算目标地址的绝对路径。当使用 `cmake --build . --target`

`install`，路径将相对于 `CMAKE_INSTALL_PREFIX` 计算。但当使用CPack时，绝对路径将相对于 `CPACK_PACKAGING_INSTALL_PREFIX` 计算。CPack的用法将在第11章中介绍。

NOTE: Unix `Makefile` 和 `Ninja` 生成器还提供了另一种机制：`DESTDIR`。可以在 `DESTDIR` 指定的目录下重新定位整个安装树。也就是说，`env DESTDIR=/tmp/stage cmake --build . --target install` 将安装相对于 `CMAKE_INSTALL_PREFIX` 和 `/tmp/stage` 目录。可以在这里阅读更多信息：https://www.gnu.org/prep/standards/html_node/DESTDIR.html

更多信息

正确设置 `RPATH` 可能相当麻烦，但这对于用户来说无法避免。默认情况下，CMake设置可执行程序的 `RPATH`，假设它们将从构建树运行。但是，安装之后 `RPATH` 被清除，当用户想要运行 `hello-world_wDSO` 时，就会出现问题。使用Linux上的 `ldd` 工具，我们可以检查构建树中的 `hello-world_wDSO` 可执行文件，运行 `ldd hello-world_wDSO` 将得到以下结果：

```
libmessage.so.1 => /home/user/cmake-cookbook/chapter-10/recipe-01/cxx-
1. example/build/lib64/libmessage.so.1(0x00007f7a92e44000)
```

在安装目录中运行 `ldd hello-world_wDSO` 将得到以下结果：

```
1. libmessage.so.1 => Not found
```

这显然是不行的。但是，总是硬编码 `RPATH` 来指向构建树或安装目录也是错误的：这两个位置中的任何一个都可能被删除，从而导致可执行文件的损坏。这里给出的解决方案为构建树和安装目录中的可执行文件设置了不同的 `RPATH`，因此它总是指向“有意义”的位置；也就是说，尽可能接近可执行文件。在构建树中运行 `ldd` 显示相同的输出：

```
1. libmessage.so.1 => /home/roberto/Workspace/robetodr/cmake-
2. cookbook/chapter-10/recipe-01/cxx-example/build/lib64/libmessage.so.1
3. (0x00007f7a92e44000)
```

另外，在安装目录下，我们得到：

```
libmessage.so.1 => /home/roberto/Software/ch10r01/bin/../lib64/libmessage.so.1
1. (0x00007fb2a725000)
```

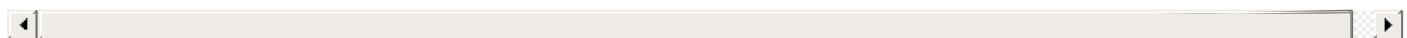
我们使用了带有目标参数的CMake安装命令，因为我们需要安装构建目标。而该命令还有另外4个参数：

- **FILES** 和 **PROGRAMS**，分别用于安装文件或程序。安装后，并设置安装文件适当的权限。对于文件，对所有者具有读和写权限，对组以及其他用户和组具有读权限。对于程序，将授予执行权

限。注意，**PROGRAMS** 要与非构建目标的可执行程序一起使用。参见：

<https://cmake.org/cmake/help/v3.6/command/install.html#installing-files>

- **DIRECTORY**，用于安装目录。当只给出一个目录名时，它通常被理解为相对于当前源目录。可以对目录的安装粒度进行控制。请参考在线文档：
<https://cmake.org/cmake/help/v3.6/command/install.html#installing-directories>
- **SCRIPT**，可以使用它在CMake脚本中定义自定义安装规则。参见：
<https://cmake.org/cmake/help/v3.6/command/install.html#custom-installation-logic>
- **EXPORT**，我们将此参数的讨论推迟到第3节，该参数用于导出目标。



10.2 生成输出头文件

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-10/recipe-02> 中找到，其中有一个C++示例。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

设想一下，当我们的小型库非常受欢迎时，许多人都在使用它。然而，一些客户希望在安装时使用静态库，而另一些客户也注意到所有符号在动态库中都是可见的。最佳方式是规定动态库只公开最小的符号，从而限制代码中定义的对象和函数对外的可见性。我们希望在默认情况下，动态库定义的所有符号都对外隐藏。这将使得项目的贡献者，能够清楚地划分库和外部代码之间的接口，因为他们必须显式地标记所有要在项目外部使用的符号。因此，我们需要完成以下工作：

- 使用同一组源文件构建动态库和静态库
- 确保正确分隔动态库中符号的可见性

第1章第3节中，已经展示了CMake提供了与平台无关的方式实现的功能。但是，没有处理符号可见性的问题。我们将用当前的配方重新讨论这两点。

准备工作

我们仍将使用与前一个示例中基本相同的代码，但是我们需要修改 `src/CMakeLists.txt` 和 `Message.hpp` 头文件。后者将包括新的、自动生成的头文件 `messageExport.h`：

```

1. #pragma once
2.
3. #include
4. #include
5.
6. #include "messageExport.h"
7.
8. class message_EXPORT Message
9. {
10. public:
11.     Message(const std::string &m) : message_(m) {}
12.
13.     friend std::ostream &operator<<(std::ostream &os, Message &obj)
14.     {
15.         return obj.printObject(os);
16.     }

```

```

17.
18. private:
19.     std::string message_;
20.     std::ostream &printObject(std::ostream &os);
21. };
22.
23. std::string getUUID();

```

`Message` 类的声明中引入了 `message_EXPORT` 预处理器指令，这个指令将让编译器生成对库的用户可见的符号。

具体实施

除了项目的名称外，主 `CMakeLists.txt` 文件没有改变。首先，看看 `src` 子目录中的 `CMakeLists.txt` 文件，所有工作实际上都在这里进行。我们将重点展示对之前示例的修改之处：

1. 为消息传递库声明 `SHARED` 库目标及其源。注意，编译定义和链接库没有改变：

```

1. add_library(message-shared SHARED "")
2.
3. target_sources(message-shared
4.   PRIVATE
5.     ${CMAKE_CURRENT_LIST_DIR}/Message.cpp
6. )
7.
8. target_compile_definitions(message-shared
9.   PUBLIC
10.    $<$<BOOL:$<UUID_FOUND>>:HAVE_UUID>
11. )
12.
13. target_link_libraries(message-shared
14.   PUBLIC
15.    $<$<BOOL:$<UUID_FOUND>>:PkgConfig::UUID>
16. )

```

2. 设置目标属性。将 `${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h` 头文件添加到公共头列表中，作为 `PUBLIC_HEADER` 目标属性的参数。`CXX_VISIBILITY_PRESET` 置和 `VISIBILITY_INLINES_HIDDEN` 属性将在下一节中讨论：

```

1. set_target_properties(message-shared
2.   PROPERTIES

```

```

3.      POSITION_INDEPENDENT_CODE 1
4.      CXX_VISIBILITY_PRESET hidden
5.      VISIBILITY_INLINES_HIDDEN 1
6.      SOVERSION ${PROJECT_VERSION_MAJOR}
7.      OUTPUT_NAME "message"
8.      DEBUG_POSTFIX "_d"
    PUBLIC_HEADER
9.      "Message.hpp;${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
10.     MACOSX_RPATH ON
11.   )

```

3. 包含 `GenerateExportHeader.cmake` 模块并调用 `generate_export_header` 函数，这将在构建目录的子目录中生成 `messageExport.h` 头文件。我们将稍后会详细讨论这个函数和生成的头文件：

```

1. include(GenerateExportHeader)
2. generate_export_header(message-shared
3.   BASE_NAME "message"
4.   EXPORT_MACRO_NAME "message_EXPORT"
5.   EXPORT_FILE_NAME
6.   "${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
7.   DEPRECATED_MACRO_NAME "message_DEPRECATED"
8.   NO_EXPORT_MACRO_NAME "message_NO_EXPORT"
9.   STATIC_DEFINE "message_STATIC_DEFINE"
10.  NO_DEPRECATED_MACRO_NAME "message_NO_DEPRECATED"
11. )

```

4. 当要更改符号的可见性(从其默认值-隐藏值)时，都应该包含导出头文件。我们已经在 `Message.hpp` 头文件例这样做了，因为想在库中公开一些符号。现在将 `${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}` 目录作为 `message-shared` 目标的 `PUBLIC` 包含目录列出：

```

1. target_include_directories(message-shared
2.   PUBLIC
3.   ${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}
4. )

```

现在，可以将注意力转向静态库的生成：

- 添加一个库目标来生成静态库。将编译与静态库相同的源文件，以获得此动态库目标：

```

1. add_library(message-static STATIC "")
2. target_sources(message-static
3.   PRIVATE
4.     ${CMAKE_CURRENT_LIST_DIR}/Message.cpp
5. )

```

2. 设置编译器定义，包含目录和链接库，就像我们为动态库目标所做的一样。但请注意，我们添加了 `message_STATIC_DEFINE` 编译时宏定义，为了确保我们的符号可以适当地暴露：

```

1. target_compile_definitions(message-static
2.   PUBLIC
3.     message_STATIC_DEFINE
4.     ${$<$<BOOL:$UUID_FOUND>:HAVE_UUID}
5. )
6.
7. target_include_directories(message-static
8.   PUBLIC
9.     ${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}
10. )
11.
12. target_link_libraries(message-static
13.   PUBLIC
14.     ${$<$<BOOL:$UUID_FOUND>:PkgConfig::UUID}
15. )

```

3. 还设置了 `message-static` 目标的属性：

```

1. set_target_properties(message-static
2.   PROPERTIES
3.     POSITION_INDEPENDENT_CODE 1
4.     ARCHIVE_OUTPUT_NAME "message"
5.     DEBUG_POSTFIX "_sd"
6.     RELEASE_POSTFIX "_s"
7.     PUBLIC_HEADER
8.     "Message.hpp;${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
9.   )

```

4. 除了链接到消息动态库目标的 `hello-world_wDSO` 可执行目标之外，还定义了另一个可执行目标 `hello-world_wAR`，这个链接指向静态库：

```

1. add_executable(hello-world_wAR hello-world.cpp)

```

```

2.
3. target_link_libraries(hello-world_wAR
4.     PUBLIC
5.     message-static
6. )

```

5. 安装指令现在多了 `message-static` 和 `hello-world_wAR` 目标，其他没有改变：

```

1. install(
2.   TARGETS
3.   message-shared
4.   message-static
5.   hello-world_wDSO
6.   hello-world_wAR
7.   ARCHIVE
8.   DESTINATION ${INSTALL_LIBDIR}
9.   COMPONENT lib
10.  RUNTIME
11.  DESTINATION ${INSTALL_BINDIR}
12.  COMPONENT bin
13.  LIBRARY
14.  DESTINATION ${INSTALL_LIBDIR}
15.  COMPONENT lib
16.  PUBLIC_HEADER
17.  DESTINATION ${INSTALL_INCLUDEDIR}/message
18.  COMPONENT dev
19. )

```

工作原理

此示例演示了，如何设置动态库的符号可见性。最好的方式是在默认情况下隐藏所有符号，显式地只公开那些需要使用的符号。这需要分为两步实现。首先，需要指示编译器隐藏符号。当然，不同的编译器将有不同的可用选项，并且直接在 `CMakeLists.txt` 中设置这些选项并不是跨平台的。CMake通过在动态库目标上设置两个属性，提供了一种健壮的跨平台方法来设置符号的可见性：

- `CXX_VISIBILITY_PRESET hidden`：这将隐藏所有符号，除非显式地标记了其他符号。当使用GNU编译器时，这将为目标添加 `-fvisibility=hidden` 标志。
- `VISIBILITY_INLINES_HIDDEN 1`：这将隐藏内联函数的符号。如果使用GNU编译器，这对应于 `-fvisibility-inlines-hidden`

Windows上，这都是默认行为。实际上，我们需要在前面的示例中通过设

置 `WINDOWS_EXPORT_ALL_SYMBOLS` 属性为 `ON` 来覆盖它。

如何标记可见的符号？这由预处理器决定，因此需要提供相应的预处理宏，这些宏可以扩展到所选平台上，以便编译器能够理解可见性属性。CMake中有现成的 `GenerateExportHeader.cmake` 模块。这个模块定义了 `generate_export_header` 函数，我们调用它的过程如下：

```

1. include(GenerateExportHeader)
2. generate_export_header(message-shared
3.   BASE_NAME "message"
4.   EXPORT_MACRO_NAME "message_EXPORT"
5.   EXPORT_FILE_NAME "${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
6.   DEPRECATED_MACRO_NAME "message_DEPRECATED"
7.   NO_EXPORT_MACRO_NAME "message_NO_EXPORT"
8.   STATIC_DEFINE "message_STATIC_DEFINE"
9.   NO_DEPRECATED_MACRO_NAME "message_NO_DEPRECATED"
10.  DEFINE_NO_DEPRECATED
11. )

```

该函数生成 `messageExport.h` 头文件，其中包含预处理器所需的宏。根据 `EXPORT_FILE_NAME` 选项的请求，在目录 `${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}` 中生成该文件。如果该选项为空，则头文件将在当前二进制目录中生成。这个函数的第一个参数是现有的目标（示例中是 `message-shared`），函数的基本调用只需要传递现有目标的名称即可。可选参数，用于细粒度的控制所有生成宏，也可以传递：

- `BASE_NAME`: 设置生成的头文件和宏的名称。
- `EXPORT_MACRO_NAME`: 设置导出宏的名称。
- `EXPORT_FILE_NAME`: 设置导出头文件的名称。
- `DEPRECATED_MACRO_NAME`: 设置弃用宏的名称。这是用来标记将要废弃的代码，如果客户使用该宏定义，编译器将发出一个将要废弃的警告。
- `NO_EXPORT_MACRO_NAME`: 设置不导出宏的名字。
- `STATIC_DEFINE`: 用于定义宏的名称，以便使用相同源编译静态库时使用。
- `NO_DEPRECATED_MACRO_NAME`: 设置宏的名称，在编译时将“将要废弃”的代码排除在外。
- `DEFINE_NO_DEPRECATED`: 指示CMake生成预处理器代码，以从编译中排除“将要废弃”的代码。

GNU/Linux上，使用GNU编译器，CMake将生成以下 `messageExport.h` 头文件：

```

1. #ifndef message_EXPORT_H
2. #define message_EXPORT_H
3. #ifdef message_STATIC_DEFINE
4. # define message_EXPORT
5. # define message_NO_EXPORT

```

```

6. #else
7. #ifndef message_EXPORT
8. #ifdef message_shared_EXPORTS
9. /* We are building this library */
10. #define message_EXPORT __attribute__((visibility("default")))
11. #else
12. /* We are using this library */
13. #define message_EXPORT __attribute__((visibility("default")))
14. #endif
15. #endif
16. #ifndef message_NO_EXPORT
17. #define message_NO_EXPORT __attribute__((visibility("hidden")))
18. #endif
19. #endif
20. #ifndef message_DEPRECATED
21. #define message_DEPRECATED __attribute__ ((deprecated))
22. #endif
23. #ifndef message_DEPRECATED_EXPORT
24. #define message_DEPRECATED_EXPORT message_EXPORT message_DEPRECATED
25. #endif
26. #ifndef message_DEPRECATED_NO_EXPORT
27. #define message_DEPRECATED_NO_EXPORT message_NO_EXPORT message_DEPRECATED
28. #endif
29. #if 1 /* DEFINE_NO_DEPRECATED */
30. #ifndef message_NO_DEPRECATED
31. #define message_NO_DEPRECATED
32. #endif
33. #endif
34. #endif

```

我们可以使用 `message_EXPORT` 宏，预先处理用户公开类和函数。弃用可以通过在前面加上 `message_DEPRECATED` 宏来实现。

从 `messageExport.h` 头文件的内容可以看出，所有符号都应该在静态库中可见，这就 是 `message_STATIC_DEFINE` 宏起了作用。当声明了目标，我们就将其设置为编译时定义。静态库的 其他目标属性如下：

- `ARCHIVE_OUTPUT_NAME "message"`：这将确保库文件的名称是 `message`，而不是 `message-static`。
- `DEBUG_POSTFIX "_sd"`：这将把给定的后缀附加到库名称中。当目标构建类型为Release时，为静态库添加“_sd”后缀。
- `RELEASE_POSTFIX "_s"`：这与前面的属性类似，当目标构建类型为Release时，为静态库添加

后缀“_s”。

更多信息

构建动态库时，隐藏内部符号是一个很好的方式。这意味着库会缩小，因为向用户公开的内容要小于库中的内容。这定义了应用程序二进制接口(ABI)，通常情况下应该与应用程序编程接口(API)一致。这分两个阶段进行：

1. 使用适当的编译器标志。
2. 使用预处理器变量(示例中是 `message_EXPORT`)标记要导出的符号。编译时，将解除这些符号(类和函数)的隐藏。

静态库只是目标文件的归档。因此，可以将源代码编译成目标文件，然后归档器将它们捆绑到归档文件中。这时没有ABI的概念：所有符号在默认情况下都是可见的，编译器的可见标志不影响静态归档。但是，如果要从相同的源文件构建动态和静态库，则需要一种方法来赋予 `message_EXPORT` 预处理变量意义，这两种情况都会出现在代码中。这里使用 `GenerateExportHeader.cmake` 模块，它定义一个包含所有逻辑的头文件，用于给出这个预处理变量的正确定义。对于动态库，它将给定的平台与编译器相组合。注意，根据构建或使用动态库，宏定义也会发生变化。幸运的是，CMake为我们解决了这个问题。对于静态库，它将扩展为一个空字符串，执行我们期望的操作—什么也不做。

细心的读者会注意到，构建此处所示的静态和共享库实际上需要编译源代码两次。对于我们的简单示例来说，这不是一个很大的开销，但会显得相当麻烦，即使对于只比示例稍大一点的项目来说，也是如此。为什么我们选择这种方法，而不是使用第1章第3节的方式呢？`OBJECT` 库负责编译库的第一步：从源文件到对象文件。该步骤中，预处理器将介入并计算 `message_EXPORT`。由于对象库的编译只发生一次，`message_EXPORT` 被计算为构建动态库库或静态库兼容的值。因此，为了避免歧义，我们选择了更健壮的方法，即编译两次，为的就是让预处理器正确地评估变量的可见性。

NOTE: 有关动态共享对象、静态存档和符号可见性的更多细节，建议阅读：<http://people.redhat.com/drepper/dsohowto.pdf>

10.3 输出目标

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-10/recipe-03> 中找到，其中有一个C++示例。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

可以假设，消息库在开源社区取得了巨大的成功。人们非常喜欢它，并在自己的项目中使用它将消息打印到屏幕上。用户特别喜欢每个打印的消息都有惟一的标识符。但用户也希望，当他们编译并安装了库，库就能更容易找到。这个示例将展示CMake如何让我们导出目标，以便其他使用CMake的项目可以轻松地获取它们。

准备工作

源代码与之前的示例一致，项目结构如下：

```

1. .
2. └── cmake
3.   └── messageConfig.cmake.in
4. └── CMakeLists.txt
5. └── src
6.   ├── CMakeLists.txt
7.   ├── hello-world.cpp
8.   ├── Message.cpp
9.   └── Message.hpp
10. └── tests
11.   ├── CMakeLists.txt
12.   └── use_target
13.     ├── CMakeLists.txt
14.     └── use_message.cpp

```

注意，cmake子目录中添加了一个 `messageConfig.cmake.in`。这个文件将包含导出的目标，还添加了一个测试来检查项目的安装和导出是否按预期工作。

具体实施

同样，主 `CMakeLists.txt` 文件相对于前一个示例来说没有变化。移动到包含我们的源代码的子目录 `src` 中：

1. 需要找到UUID库，可以重用之前示例中的代码：

```

1. # Search for pkg-config and UUID
2. find_package(PkgConfig QUIET)
3. if(PKG_CONFIG_FOUND)
4.     pkg_search_module(UUID uuid IMPORTED_TARGET)
5.     if(TARGET PkgConfig::UUID)
6.         message(STATUS "Found libuuid")
7.         set(UUID_FOUND TRUE)
8.     endif()
9. endif()

```

2. 接下来，设置动态库目标并生成导出头文件：

```

1. add_library(message-shared SHARED "")
2. include(GenerateExportHeader)
3.
4. generate_export_header(message-shared
5.   BASE_NAME "message"
6.   EXPORT_MACRO_NAME "message_EXPORT"
7.   EXPORT_FILE_NAME
8.   "${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
9.   DEPRECATED_MACRO_NAME "message_DEPRECATED"
10.  NO_EXPORT_MACRO_NAME "message_NO_EXPORT"
11.  STATIC_DEFINE "message_STATIC_DEFINE"
12.  NO_DEPRECATED_MACRO_NAME "message_NO_DEPRECATED"
13.  DEFINE_NO_DEPRECATED
14. )
15. target_sources(message-shared
16.   PRIVATE
17.   ${CMAKE_CURRENT_LIST_DIR}/Message.cpp
18. )

```

3. 为目标设置了 `PUBLIC` 和 `INTERFACE` 编译定义。注意 `$<INSTALL_INTERFACE:...>` 生成器表达式的使用：

```

1. target_compile_definitions(message-shared
2. PUBLIC
3.   $<$<BOOL:$UUID_FOUND>:HAVE_UUID>
4. INTERFACE
5.   $<INSTALL_INTERFACE:USING_message>
6. )

```

4. 链接库和目标属性与前一个示例一样：

```

1. target_link_libraries(message-static
2.   PUBLIC
3.     ${$<BOOL:${UUID_FOUND}>:PkgConfig::UUID}
4.   )
5.
6. set_target_properties(message-static
7.   PROPERTIES
8.     POSITION_INDEPENDENT_CODE 1
9.     ARCHIVE_OUTPUT_NAME "message"
10.    DEBUG_POSTFIX "_sd"
11.    RELEASE_POSTFIX "_s"
12.    PUBLIC_HEADER
13.    "Message.hpp;${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}/messageExport.h"
14.  )

```

5. 可执行文件的生成，与前一个示例中使用的命令完全相同：

```

1. add_executable(hello-world_wDSO hello-world.cpp)
2.
3. target_link_libraries(hello-world_wDSO
4.   PUBLIC
5.     message-shared
6.   )
7.
8. # Prepare RPATH
9. file(RELATIVE_PATH _rel ${CMAKE_INSTALL_PREFIX}/${INSTALL_BINDIR}
10. ${CMAKE_INSTALL_PREFIX})
11. if(APPLE)
12.   set(_rpath "@loader_path/${_rel}")
13. else()
14.   set(_rpath "\$ORIGIN/${_rel}")
15. endif()
16. file(TO_NATIVE_PATH "${_rpath}/${INSTALL_LIBDIR}" message_RPATH)
17. set_target_properties(hello-world_wDSO
18.   PROPERTIES
19.     MACOSX_RPATH ON
20.     SKIP_BUILD_RPATH OFF
21.     BUILD_WITH_INSTALL_RPATH OFF
22.     INSTALL_RPATH "${message_RPATH}"
23.     INSTALL_RPATH_USE_LINK_PATH ON
24.   )

```

```

25.
26. add_executable(hello-world_wAR hello-world.cpp)
27.
28. target_link_libraries(hello-world_wAR
29.   PUBLIC
30.   message-static
31. )

```

现在，来看看安装规则：

1. 因为CMake可以正确地将每个目标放在正确的地方，所以把目标的安装规则都列在一起。这次，添加了 `EXPORT` 关键字，这样CMake将为目标生成一个导出的目标文件：

```

1. install(
2.   TARGETS
3.   message-shared
4.   message-static
5.   hello-world_wDSO
6.   hello-world_wAR
7.   EXPORT
8.   messageTargets
9.   ARCHIVE
10.  DESTINATION ${INSTALL_LIBDIR}
11.  COMPONENT lib
12.  RUNTIME
13.  DESTINATION ${INSTALL_BINDIR}
14.  COMPONENT bin
15.  LIBRARY
16.  DESTINATION ${INSTALL_LIBDIR}
17.  COMPONENT lib
18.  PUBLIC_HEADER
19.  DESTINATION ${INSTALL_INCLUDEDIR}/message
20.  COMPONENT dev
21. )

```

2. 自动生成的导出目标文件称为 `messageTargets.cmake`，需要显式地指定它的安装规则。这个文件的目标是 `INSTALL_CMAKEDIR`，在主 `CMakeLists.txt` 文件中定义：

```

1. install(
2.   EXPORT
3.   messageTargets
4.   NAMESPACE

```

```

5.     "message::"
6. DESTINATION
7.     ${INSTALL_CMAKEDIR}
8. COMPONENT
9.     dev
10. )

```

3. 最后，需要生成正确的CMake配置文件。这些将确保下游项目能够找到消息库导出的目标。为此，首先包括 `CMakePackageConfigHelpers.cmake` 标准模块：

```
1. include(CMakePackageConfigHelpers)
```

4. 让CMake为我们的库，生成一个包含版本信息的文件：

```

1. write_basic_package_version_file(
2.     ${CMAKE_CURRENT_BINARY_DIR}/messageConfigVersion.cmake
3.     VERSION ${PROJECT_VERSION}
4.     COMPATIBILITY SameMajorVersion
5. )

```

5. 使用 `configure_package_config_file` 函数，我们生成了实际的CMake配置文件。这是基于模板 `cmake/messageConfig.cmake.in` 文件：

```

1. configure_package_config_file(
2.     ${PROJECT_SOURCE_DIR}/cmake/messageConfig.cmake.in
3.     ${CMAKE_CURRENT_BINARY_DIR}/messageConfig.cmake
4.     INSTALL_DESTINATION ${INSTALL_CMAKEDIR}
5. )

```

6. 最后，为这两个自动生成的配置文件设置了安装规则：

```

1. install(
2. FILES
3.     ${CMAKE_CURRENT_BINARY_DIR}/messageConfig.cmake
4.     ${CMAKE_CURRENT_BINARY_DIR}/messageConfigVersion.cmake
5. DESTINATION
6.     ${INSTALL_CMAKEDIR}
7. )

```

`cmake/messageConfig.cmake` 的内容是什么？该文件的顶部有相关的说明，可以作为用户文档供使用者查看。让我们看看实际的CMake命令：

1. 占位符将使用 `configure_package_config_file` 命令进行替换：

```
1. @PACKAGE_INIT@
```

2. 包括为目标自动生成的导出文件：

```
1. include("${CMAKE_CURRENT_LIST_DIR}/messageTargets.cmake")
```

3. 检查静态库和动态库，以及两个“Hello, World”可执行文件是否带有CMake提供的 `check_required_components` 函数：

```
1. check_required_components(
2.   "message-shared"
3.   "message-static"
4.   "message-hello-world_wDSO"
5.   "message-hello-world_wAR"
6. )
```

4. 检查目标 `PkgConfig::UUID` 是否存在。如果没有，我们再次搜索UUID库(只在非Windows操作系统下有效)：

```
1. if(NOT WIN32)
2.   if(NOT TARGET PkgConfig::UUID)
3.     find_package(PkgConfig REQUIRED QUIET)
4.     pkg_search_module(UUID REQUIRED uuid IMPORTED_TARGET)
5.   endif()
6. endif()
```

测试一下：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake -DCMAKE_INSTALL_PREFIX=$HOME/Software/recipe-03 ..
4. $ cmake --build . --target install
```

安装树应该如下所示：

```
1. $HOME/Software/recipe-03/
2. └── bin
3.   └── hello-world_wAR
4.   └── hello-world_wDSO
```

```

5. └── include
6.   └── message
7.     ├── messageExport.h
8.     └── Message.hpp
9. └── lib64
10.  └── libmessage_s.a
11.  └── libmessage.so -> libmessage.so.1
12.  └── libmessage.so.1
13. └── share
14.   └── cmake
15.     └── recipe-03
16.       ├── messageConfig.cmake
17.       ├── messageConfigVersion.cmake
18.       ├── messageTargets.cmake
19.       └── messageTargets-release.cmake

```

出现了一个 `share` 子目录，其中包含我们要求CMake自动生成的所有文件。现在开始，消息库的用户可以在他们自己的 `CMakeLists.txt` 文件中找到消息库，只要他们设置 `message_DIR` 的CMake变量，指向安装树中的 `share/cmake/message` 目录：

```
1. find_package(message 1 CONFIG REQUIRED)
```

工作原理

这个示例涵盖了很多领域。对于构建系统将要执行的操作，CMake目标是一个非常有用的抽象概念。使用 `PRIVATE`、`PUBLIC` 和 `INTERFACE` 关键字，我们可以设置项目中的目标进行交互。在实践中，这允许我们定义目标A的依赖关系，将如何影响目标B(依赖于A)。如果库维护人员提供了适当的CMake配置文件，那么只需很少的CMake命令就可以轻松地解决所有依赖关系。

这个问题可以通过遵循 `message-static`、`message-shared`、`hello-world_wDSO` 和 `hello-world_wAR` 目标概述的模式来解决。我们将单独分析 `message-shared` 目标的CMake命令，这里只是进行一般性讨论：

- 生成目标在项目构建中列出其依赖项。对UUID库的链接是 `message-shared` 的 `PUBLIC` 需求，因为它将用于在项目中构建目标和在下游项目中构建目标。编译时宏定义和包含目录需要在 `PUBLIC` 级或 `INTERFACE` 级目标上进行设置。它们实际上是在项目中构建目标时所需要的，其他的只与下游项目相关。此外，其中一些只有在项目安装之后才会相关联。这里使用了 `$<BUILD_INTERFACE:...>` 和 `$<INSTALL_INTERFACE:...>` 生成器表达式。只有消息库外部的下游目标才需要这些，也就是说，只有在安装了目标之后，它们才会变得可见。我们的例子中，应用如下：

- 只有在项目中使用了 `message-shared` 库，那么 `$<BUILD_INTERFACE:${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}>` 才会扩展成 `${CMAKE_BINARY_DIR}/${INSTALL_INCLUDEDIR}`
- 只有在 `message-shared` 库在另一个构建树中，作为一个已导出目标，那么 `$<INSTALL_INTERFACE:${INSTALL_INCLUDEDIR}>` 将会扩展成 `${INSTALL_INCLUDEDIR}`

2. 描述目标的安装规则，包括生成文件的名称。

3. 描述CMake生成的导出文件的安装规则 `messageTargets.cmake` 文件将安装到 `INSTALL_CMAKEDIR`。目标导出文件的安装规则的名称空间选项，将把给定字符串前置到目标的名称中，这有助于避免来自不同项目的目标之间的名称冲突。`INSTALL_CMAKEDIR` 变量是在主 `CMakeLists.txt` 文件中设置的：

```

1. if(WIN32 AND NOT CYGWIN)
2.   set(DEF_INSTALL_CMAKEDIR CMake)
3. else()
4.   set(DEF_INSTALL_CMAKEDIR share/cmake/${PROJECT_NAME})
5. endif()
6. set(INSTALL_CMAKEDIR ${DEF_INSTALL_CMAKEDIR} CACHE PATH "Installation
   directory for CMake files")

```

`CMakeLists.txt` 的最后一部分生成配置文件。包括 `CMakePackageConfigHelpers.cmake` 模块，分三步完成：

- 调用 `write_basic_package_version_file` 函数生成一个版本文件包。宏的第一个参数是版本控制文件的路径：`messageConfigVersion.cmake`。版本格式为 `Major.Minor.Patch`，并使用 `PROJECT_VERSION` 指定版本，还可以指定与库的新版本的兼容性。例子中，当库具有相同的主版本时，为了保证兼容性，使用了相同的 `SameMajorVersion` 参数。
- 接下来，配置模板文件 `messageConfig.cmake.in`，该文件位于 `cmake` 子目录中。
- 最后，为新生成的文件设置安装规则。两者都将安装在 `INSTALL_CMAKEDIR` 下。

更多信息

消息库的客户现在非常高兴，因为终于可以在自己的系统上安装这个库，对自己的 `CMakeLists.txt` 进行简单的修改，就能找到消息库：

```
1. find_package(message VERSION 1 REQUIRED)
```

客户可以用以下方式配置他们的项目：

```
1. $ cmake -Dmessage_DIR=/path/to/message/share/cmake/message ..
```

我们示例中包含的测试，显示了如何检查目标的安装是否按照计划进行。看看 `tests` 文件夹的结构，我们注意到 `use_target` 子目录：

```
1. tests/
2. └── CMakeLists.txt
3.   └── use_target
4.     ├── CMakeLists.txt
5.     └── use_message.cpp
```

这个目录包含一个使用导出目标的小项目。有趣的部分是在 `CMakeLists.txt` 文件中指定的测试：

1. 我们测试小项目，可以配置为使用已安装的库。这是 `use-target` 测试固件的设置步骤，可以参考第4章第10节：

```
1. add_test(
2.   NAME use-target_configure
3.   COMMAND
4.     ${CMAKE_COMMAND} -H${CMAKE_CURRENT_LIST_DIR}/use_target
5.                 -B${CMAKE_CURRENT_BINARY_DIR}/build_use-target
6.                 -G${CMAKE_GENERATOR}
7.                 -Dmessage_DIR=${CMAKE_INSTALL_PREFIX}/${INSTALL_CMAKEDIR}
8.                 -DCMAKE_BUILD_TYPE=$<CONFIGURATION>
9.
10. )
11.
12. set_tests_properties(use-target_configure
13.   PROPERTIES
14.     FIXTURES_SETUP use-target
15. )
```

2. 测试了小项目可以构建：

```
1. add_test(
2.   NAME use-target_build
3.   COMMAND
4.     ${CMAKE_COMMAND} --build ${CMAKE_CURRENT_BINARY_DIR}/build_use-target
5.                 --config $<CONFIGURATION>
6. )
7.
```

```

8. set_tests_properties(use-target-build
9.   PROPERTIES
10.    FIXTURES_REQUIRED use-target
11. )

```

3. 小项目的测试也会运行：

```

1. set(_test_target)
2. if(MSVC)
3.   set(_test_target "RUN_TESTS")
4. else()
5.   set(_test_target "test")
6. endif()
7.
8. add_test(
9.   NAME use-target-test
10.  COMMAND
11.    ${CMAKE_COMMAND} --build ${CMAKE_CURRENT_BINARY_DIR}/build_use-target
12.          --target ${_test_target}
13.          --config ${<CONFIGURATION>}
14. )
15. set_tests_properties(use-target-test
16.   PROPERTIES
17.    FIXTURES_REQUIRED use-target
18. )
19. unset(_test_target)

```

4. 最后，我们拆除固件：

```

1. add_test(
2.   NAME use-target-cleanup
3.   COMMAND
4.     ${CMAKE_COMMAND} -E remove_directory
5.     ${CMAKE_CURRENT_BINARY_DIR}/build_use-target
6.
7. set_tests_properties(use-target-cleanup
8.   PROPERTIES
9.    FIXTURES_CLEANUP use-target
10. )

```

注意，这些测试只能在项目安装之后运行。

10.4 安装超级构建

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-10/recipe-04> 中找到，其中有一个C++示例。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

我们的消息库取得了巨大的成功，许多其他程序员都使用它，并且非常满意。也希望在自己的项目中使用它，但是不确定如何正确地管理依赖关系。可以用自己的代码附带消息库的源代码，但是如果该库已经安装在系统上了应该怎么做呢？第8章，展示了超级构建的场景，但是不确定如何安装这样的项目。本示例将带您了解安装超级构建的安装细节。

准备工作

此示例将针对消息库，构建一个简单的可执行链接。项目布局如下：

```

1.   |- cmake
2.   |   |- install_hook.cmake.in
3.   |   |- print_rpath.py
4.   |- CMakeLists.txt
5.   |- external
6.   |   |- upstream
7.   |       |- CMakeLists.txt
8.   |       |- message
9.   |           |- CMakeLists.txt
10.  |- src
11.  |   |- CMakeLists.txt
12.  |   |- use_message.cpp

```

主 `CMakeLists.txt` 文件配合超级构建，`external` 子目录包含处理依赖项的CMake指令。`cmake` 子目录包含一个Python脚本和一个模板CMake脚本。这些将用于安装方面的微调，CMake脚本首先进行配置，然后调用Python脚本打印 `use_message` 可执行文件的 `RPATH`：

```

1. import shlex
2. import subprocess
3. import sys
4.
5. def main():
6.     patcher = sys.argv[1]
7.     elfobj = sys.argv[2]
8.     tools = {'patchelf': '--print-rpath', 'chrpath': '--list', 'otool': '-L'}

```

```

9. if patcher not in tools.keys():
10.     raise RuntimeError('Unknown tool {}'.format(patcher))
11.     cmd = shlex.split('{:s} {:s} {:s}'.format(patcher, tools[patcher], elfobj))
12.     rpath = subprocess.run(
13.         cmd,
14.         bufsize=1,
15.         stdout=subprocess.PIPE,
16.         stderr=subprocess.PIPE,
17.         universal_newlines=True)
18.     print(rpath.stdout)
19.
20. if __name__ == "__main__":
21.     main()

```

使用平台原生工具可以轻松地打印 `RPATH`，稍后我们将在本示例中讨论这些工具。

最后，`src` 子目录包含项目的 `CMakeLists.txt` 和源文件。`use_message.cpp` 源文件包含以下内容：

```

1. #include <cstdlib>
2. #include <iostream>
3.
4. #ifdef USING_message
5. #include <message/Message.hpp>
6. void messaging()
7. {
8.     Message say_hello("Hello, World! From a client of yours!");
9.     std::cout << say_hello << std::endl;
10.    Message say_goodbye("Goodbye, World! From a client of yours!");
11.    std::cout << say_goodbye << std::endl;
12. }
13. #else
14. void messaging()
15. {
16.     std::cout << "Hello, World! From a client of yours!" << std::endl;
17.     std::cout << "Goodbye, World! From a client of yours!" << std::endl;
18. }
19. #endif
20.
21. int main()
22. {
23.     messaging();

```

```

24.     return EXIT_SUCCESS;
25. }
```

具体实施

我们将从主 `CMakeLists.txt` 文件开始，它用来协调超级构建：

- 与之前的示例相同。首先声明一个C++11项目，设置了默认安装路径、构建类型、目标的输出目录，以及安装树中组件的布局：

```

1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2.
3. project(recipe-04
4.   LANGUAGES CXX
5.   VERSION 1.0.0
6.   )
7.
8. # <<< General set up >>>
9.
10. set(CMAKE_CXX_STANDARD 11)
11. set(CMAKE_CXX_EXTENSIONS OFF)
12. set(CMAKE_CXX_STANDARD_REQUIRED ON)
13.
14. if(NOT CMAKE_BUILD_TYPE)
15.   set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
16. endif()
17.
18. message(STATUS "Build type set to ${CMAKE_BUILD_TYPE}")
19.
20. message(STATUS "Project will be installed to ${CMAKE_INSTALL_PREFIX}")
21.
22. include(GNUInstallDirs)
23.
24. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
25.   ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
26. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
27.   ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
28. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
29.   ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
30.
31. # Offer the user the choice of overriding the installation directories
```

```

      set(INSTALL_LIBDIR ${CMAKE_INSTALL_LIBDIR} CACHE PATH "Installation
32. directory for libraries")
      set(INSTALL_BINDIR ${CMAKE_INSTALL_BINDIR} CACHE PATH "Installation
33. directory for executables")
      set(INSTALL_INCLUDEDIR ${CMAKE_INSTALL_INCLUDEDIR} CACHE PATH
34. "Installation directory for header files")
35. if(WIN32 AND NOT CYGWIN)
36.   set(DEF_INSTALL_CMAKEDIR CMake)
37. else()
38.   set(DEF_INSTALL_CMAKEDIR share/cmake/${PROJECT_NAME})
39. endif()
      set(INSTALL_CMAKEDIR ${DEF_INSTALL_CMAKEDIR} CACHE PATH "Installation
40. directory for CMake files")
41.
42. # Report to user
43. foreach(p LIB BIN INCLUDE CMAKE)
44.   file(TO_NATIVE_PATH ${CMAKE_INSTALL_PREFIX}/${INSTALL_${p}DIR} _path )
45.   message(STATUS "Installing ${p} components to ${_path}")
46.   unset(_path)
47. endforeach()

```

2. 设置了 `EP_BASE` 目录属性，这将为超构建中的子项目设置布局。所有子项目都将在 `CMAKE_BINARY_DIR` 的子项目文件夹下生成：

```
1. set_property(DIRECTORY PROPERTY EP_BASE ${CMAKE_BINARY_DIR}/subprojects)
```

3. 然后，声明 `STAGED_INSTALL_PREFIX` 变量。这个变量指向构建目录下的 `stage` 子目录，项目将在构建期间安装在这里。这是一种沙箱安装过程，让我们有机会检查整个超级构建的布局：

```
1. set(STAGED_INSTALL_PREFIX ${CMAKE_BINARY_DIR}/stage)
2. message(STATUS "${PROJECT_NAME} staged install: ${STAGED_INSTALL_PREFIX}")
```

4. 添加 `external/upstream` 子目录。其中包括使用CMake指令来管理我们的上游依赖关系，在我们的例子中，就是消息库：

```
1. add_subdirectory(external/upstream)
```

5. 然后，包含 `ExternalProject.cmake` 标准模块：

```
1. include(ExternalProject)
```

6. 将自己的项目作为外部项目添加，调用 `ExternalProject_Add` 命令。`SOURCE_DIR` 用于指定源位于 `src` 子目录中。我们会选择适当的CMake参数来配置我们的项目。这里，使用 `STAGED_INSTALL_PREFIX` 作为子项目的安装目录：

```

1. ExternalProject_Add(${PROJECT_NAME}_core
2.   DEPENDS
3.   message_external
4.   SOURCE_DIR
5.   ${CMAKE_CURRENT_SOURCE_DIR}/src
6.   CMAKE_ARGS
7.   -DCMAKE_INSTALL_PREFIX=${STAGED_INSTALL_PREFIX}
8.   -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
9.   -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
10.  -DCMAKE_CXX_FLAGS=${CMAKE_CXX_FLAGS}
11.  -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
12.  -DCMAKE_CXX_EXTENSIONS=${CMAKE_CXX_EXTENSIONS}
13.  -DCMAKE_CXX_STANDARD_REQUIRED=${CMAKE_CXX_STANDARD_REQUIRED}
14.  -Dmessage_DIR=${message_DIR}
15.  CMAKE_CACHE_ARGS
16.  -DCMAKE_PREFIX_PATH:PATH=${CMAKE_PREFIX_PATH}
17.  BUILD_ALWAYS
18.  1
19. )

```

7. 现在，为 `use_message` 添加一个测试，并由 `recipe-04_core` 构建。这将运行 `use_message` 可执行文件的安装，即位于构建树中的安装：

```

1. enable_testing()
2.
3. add_test(
4.   NAME
5.   check_use_message
6.   COMMAND
7.   ${STAGED_INSTALL_PREFIX}/${INSTALL_BINDIR}/use_message
8. )

```

8. 最后，可以声明安装规则。因为所需要的东西都已经安装在暂存区域中，我们只要将暂存区域的内容复制到安装目录即可：

```

1. install(
2.   DIRECTORY

```

```

3.      ${STAGED_INSTALL_PREFIX}/
4. DESTINATION
5.
6. USE_SOURCE_PERMISSIONS
7. )

```

9. 使用 `SCRIPT` 参数声明一个附加的安装规则。CMake脚本的 `install_hook.cmake` 将被执行，但只在GNU/Linux和macOS上执行。这个脚本将打印已安装的可执行文件的 `RPATH`，并运行它。我们将在下一节详细地讨论这个问题：

```

1. if(UNIX)
2.   set(PRINT_SCRIPT "${CMAKE_CURRENT_LIST_DIR}/cmake/print_rpath.py")
3.   configure_file(cmake/install_hook.cmake.in install_hook.cmake @ONLY)
4.   install(
5.     SCRIPT
6.       ${CMAKE_CURRENT_BINARY_DIR}/install_hook.cmake
7.   )
8. endif()

```

`-Dmessage_DIR=${message_DIR}` 已作为CMake参数传递给项目，这将正确设置消息库依赖项的位置。`message_DIR` 的值在 `external/upstream/message` 目录下的 `CMakeLists.txt` 文件中定义。这个文件处理依赖于消息库，让我们看看是如何处理的：

- 首先，搜索并找到包。用户可能已经在系统的某个地方安装了，并在配置时传递了 `message_DIR`：

```
1. find_package(message 1 CONFIG QUIET)
```

- 如果找到了消息库，我们将向用户报告目标的位置和版本，并添加一个虚拟的 `message_external` 目标。这里，需要虚拟目标来正确处理超构建的依赖关系：

```

1. if(message_FOUND)
2.   get_property(_loc TARGET message::message-shared PROPERTY LOCATION)
   message(STATUS "Found message: ${_loc} (found version
3. ${message_VERSION})")
4.   add_library(message_external INTERFACE) # dummy

```

- 如果没有找到这个库，我们将把它添加为一个外部项目，从在线Git存储库下载它，然后编译它。安装路径、构建类型和安装目录布局都是由主 `CMakeLists.txt` 文件设置，C++编译器和标志也是如此。项目将安装到 `STAGED_INSTALL_PREFIX` 下，然后进行测试：

```

1. else()
2.   include(ExternalProject)
3.   message(STATUS "Suitable message could not be located, Building message
4. instead.")
5.   ExternalProject_Add(message_external
6.     GIT_REPOSITORY
7.       https://github.com/dev-cafe/message.git
8.     GIT_TAG
9.       master
10.    UPDATE_COMMAND
11.      ""
12.    CMAKE_ARGS
13.      -DCMAKE_INSTALL_PREFIX=${STAGED_INSTALL_PREFIX}
14.      -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
15.      -DCMAKE_CXX_COMPILER=${CMAKE_CXX_COMPILER}
16.    CMAKE_CACHE_ARGS
17.      -DCMAKE_CXX_FLAGS:STRING=${CMAKE_CXX_FLAGS}
18.    TEST_AFTER_INSTALL
19.      1
20.    DOWNLOAD_NO_PROGRESS
21.      1
22.    LOG_CONFIGURE
23.      1
24.    LOG_BUILD
25.      1
26.    LOG_INSTALL
27.      1
28.  )

```

4. 最后，将 `message_DIR` 目录进行设置，为指向新构建的 `messageConfig.cmake` 文件指明安装路径。注意，这些路径被保存到 `CMakeCache` 中：

```

1. if(WIN32 AND NOT CYGWIN)
2.   set(DEF_message_DIR ${STAGED_INSTALL_PREFIX}/CMake)
3. else()
4.   set(DEF_message_DIR ${STAGED_INSTALL_PREFIX}/share/cmake/message)
5. endif()
6. file(TO_NATIVE_PATH "${DEF_message_DIR}" DEF_message_DIR)
7. set(message_DIR ${DEF_message_DIR})
8.   CACHE PATH "Path to internally built messageConfig.cmake" FORCE)
9. endif()

```

我们终于准备好编译我们自己的项目，并成功地将其链接到消息库（无论是系统上已有的消息库，还是新构建的消息库）。由于这是一个超级构建，`src` 子目录下的代码是一个完全独立的CMake项目：

1. 声明一个C++11项目：

```

1. cmake_minimum_required(VERSION 3.6 FATAL_ERROR)
2.
3. project(recipe-04_core
4.   LANGUAGES CXX
5. )
6.
7. set(CMAKE_CXX_STANDARD 11)
8. set(CMAKE_CXX_EXTENSIONS OFF)
9. set(CMAKE_CXX_STANDARD_REQUIRED ON)
10.
11. include(GNUInstallDirs)
12.
13. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
14.   ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
15. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
16.   ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
17. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
18.   ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

```

2. 尝试找到消息库。超级构建中，正确设置 `message_DIR`：

```

1. find_package(message 1 CONFIG REQUIRED)
2. get_property(_loc TARGET message::message-shared PROPERTY LOCATION)
   message(STATUS "Found message: ${_loc} (found version
3. ${message_VERSION})")

```

3. 添加可执行目标 `use_message`，该目标由 `use_message.cpp` 源文件创建，并连接到 `message::message-shared` 目标：

```

1. add_executable(use_message use_message.cpp)
2.
3. target_link_libraries(use_message
4.   PUBLIC
5.     message::message-shared
6. )

```

4. 为 `use_message` 设置目标属性。再次对 `RPATH` 进行设置：

```

1. # Prepare RPATH
  file(RELATIVE_PATH _rel ${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_BINDIR}
2. ${CMAKE_INSTALL_PREFIX})
3. if(APPLE)
4.   set(_rpath "@loader_path/${_rel}")
5. else()
6.   set(_rpath "\$ORIGIN/${_rel}")
7. endif()
8. file(TO_NATIVE_PATH "${_rpath}/${CMAKE_INSTALL_LIBDIR}" use_message_RPATH)
9.
10. set_target_properties(use_message
11.   PROPERTIES
12.     MACOSX_RPATH ON
13.     SKIP_BUILD_RPATH OFF
14.     BUILD_WITH_INSTALL_RPATH OFF
15.     INSTALL_RPATH "${use_message_RPATH}"
16.     INSTALL_RPATH_USE_LINK_PATH ON
17. )

```

5. 最后，为 `use_message` 目标设置了安装规则：

```

1. install(
2.   TARGETS
3.     use_message
4.   RUNTIME
5.   DESTINATION ${CMAKE_INSTALL_BINDIR}
6.   COMPONENT bin
7. )

```

现在瞧瞧CMake脚本模板 `install_hook.cmake.in` 的内容：

1. CMake脚本在我们的主项目范围之外执行，因此没有定义变量或目标的概念。因此，需要设置变量来保存已安装的 `use_message` 可执行文件的完整路径。注意使用 `@INSTALL_BINDIR@`，它将由 `configure_file` 解析：

```
1. set(_executable ${CMAKE_INSTALL_PREFIX}@/INSTALL_BINDIR@/use_message)
```

2. 需要找到平台本机可执行工具，使用该工具打印已安装的可执行文件的 `RPATH`。我们将搜索 `chrpath`、`patchelf` 和 `otool`。当找到已安装的程序时，向用户提供有用的状态信息，并且退出搜索：

```

1. set(_patcher)
2. list(APPEND _patchers chrpath patchelf otool)
3. foreach(p IN LISTS _patchers)
4.   find_program(${p}_FOUND
5.     NAMES
6.       ${p}
7.   )
8.   if(${p}_FOUND)
9.     set(_patcher ${p})
10.    message(STATUS "ELF patching tool ${_patcher} FOUND")
11.   break()
12. endif()
13. endforeach()

```

3. 检查 `_patcher` 变量是否为空，这意味着PatchELF工具是否可用。当为空时，我们要进行的操作将会失败，所以会发出一个致命错误，提醒用户需要安装PatchELF工具：

```

1. if(NOT _patcher)
   message(FATAL_ERROR "ELF patching tool NOT FOUND!\nPlease install one
2. of chrpath, patchelf or otool")

```

4. 当PatchELF工具找到了，则继续。我们调用Python脚本 `print_rpath.py`，将 `_executable` 变量作为参数传递给 `execute_process`：

```

1. find_package(PythonInterp REQUIRED QUIET)
2. execute_process(
3.   COMMAND
4.     ${PYTHON_EXECUTABLE} @PRINT_SCRIPT@ "${_patcher}"
5.     "${_executable}"
6.   RESULT_VARIABLE _res
7.   OUTPUT_VARIABLE _out
8.   ERROR_VARIABLE _err
9.   OUTPUT_STRIP_TRAILING_WHITESPACE
10. )

```

5. 检查 `_res` 变量的返回代码。如果执行成功，将打印 `_out` 变量中捕获的标准输出流。否则，打印退出前捕获的标准输出和错误流：

```

1. if(_res EQUAL 0)
2.   message(STATUS "RPATH for ${_executable} is ${_out}")
3. else()

```

```

4.     message(STATUS "Something went wrong!")
5.     message(STATUS "Standard output from print_rpath.py: ${_out}")
6.     message(STATUS "Standard error from print_rpath.py: ${_err}")
    message(FATAL_ERROR "${_patcher} could NOT obtain RPATH for
7. ${_executable}")
8. endif()
9. endif()

```

6. 再使用 `execute_process` 来运行已安装的 `use_message` 可执行目标：

```

1. execute_process(
2.   COMMAND ${_executable}
3.   RESULT_VARIABLE _res
4.   OUTPUT_VARIABLE _out
5.   ERROR_VARIABLE _err
6.   OUTPUT_STRIP_TRAILING_WHITESPACE
7. )

```

7. 最后，向用户报告 `execute_process` 的结果：

```

1. if(_res EQUAL 0)
2.   message(STATUS "Running ${_executable}:\n ${_out}")
3. else()
4.   message(STATUS "Something went wrong!")
5.   message(STATUS "Standard output from running ${_executable}:\n ${_out}")
6.   message(STATUS "Standard error from running ${_executable}:\n ${_err}")
7.   message(FATAL_ERROR "Something went wrong with ${_executable}")
8. endif()

```

工作原理

CMake工具箱中，超级构建是非常有用的模式。它通过将复杂的项目划分为更小、更容易管理的子项目来管理它们。此外，可以使用CMake作为构建项目的包管理器。CMake可以搜索依赖项，如果在系统上找不到依赖项，则重新构建它们。这里需要三个 `CMakeLists.txt` 文件：

- 主 `CMakeLists.txt` 文件包含项目和依赖项共享的设置，还包括我们自己的项目（作为外部项目）。本例中，我们选择的名称为 `${PROJECT_NAME}_core`；也就是 `recipe-04_core`，因为项目名称 `recipe-04` 用于超级构建。
- 外部 `CMakeLists.txt` 文件将尝试查找上游依赖项，并在导入目标和构建目标之间进行切换，这取决于是否找到了依赖项。对于每个依赖项，最好有单独的子目录，其中包含一个 `CMakeLists.txt` 文件。

- 最后，我们项目的 `CMakeLists.txt` 文件，可以构建一个独立的CMake项目。在原则上，我们可以自己配置和构建它，而不需要超级构建提供的依赖关系管理工具。

当对消息库的依赖关系未得到满足时，将首先考虑超级构建：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake -DCMAKE_INSTALL_PREFIX=$HOME/Software/recipe-04 ..
```

让CMake查找库，这是我们得到的输出：

```
1. -- The CXX compiler identification is GNU 7.3.0
   -- Check for working CXX compiler: /nix/store/gqq2vrcq7krqi9rrl6pphvsg81sb8pjw-
2. gcc-wrapper-7.3.0/bin/g++
   -- Check for working CXX compiler: /nix/store/gqq2vrcq7krqi9rrl6pphvsg81sb8pjw-
3. gcc-wrapper-7.3.0/bin/g++ -- works
4. -- Detecting CXX compiler ABI info
5. -- Detecting CXX compiler ABI info - done
6. -- Detecting CXX compile features
7. -- Detecting CXX compile features - done
8. -- Project will be installed to /home/roberto/Software/recipe-04
9. -- Build type set to Release
10. -- Installing LIB components to /home/roberto/Software/recipe-04/lib64
11. -- Installing BIN components to /home/roberto/Software/recipe-04/bin
12. -- Installing INCLUDE components to /home/roberto/Software/recipe-04/include
   -- Installing CMAKE components to /home/roberto/Software/recipe-
13. 04/share/cmake/recipe-04
   -- recipe-04 staged install: /home/roberto/Workspace/robertodr/cmake-cookbook/chapter-
14. 10/recipe-04/cxx-example/build/stage
15. -- Suitable message could not be located, Building message instead.
16. -- Configuring done
17. -- Generating done
   -- Build files have been written to: /home/roberto/Workspace/robertodr/cmake-
18. cookbook/chapter-10/recipe-04/cxx-example/build
```

根据指令，CMake报告如下：

- 安装将分阶段进入构建树。分阶段安装是对实际安装过程进行沙箱化的一种方法。作为开发人员，这对于在运行安装命令之前检查所有库、可执行程序和文件是否安装在正确的位置非常有用。对于用户来说，可在构建目录中给出了相同的结构。这样，即使没有运行正确的安装，我们的项目也可以立即使用。
- 系统上没有找到合适的消息库。然后，CMake将运行在构建项目之前构建库所提供的命令，以满

足这种依赖性。

如果库已经位于系统的已知位置，我们可以将 `-Dmessage_DIR` 选项传递给CMake：

```
$ cmake -DCMAKE_INSTALL_PREFIX=$HOME/Software/use_message -
1. Dmessage_DIR=$HOME/Software/message/share/cmake/message ..
```

事实上，这个库已经找到并导入。我们对自己的项目进行建造操作：

```
1. -- The CXX compiler identification is GNU 7.3.0
   -- Check for working CXX compiler: /nix/store/gqq2vrcq7krqi9rrl6pphvsg81sb8pjw-
2. gcc-wrapper-7.3.0/bin/g++
   -- Check for working CXX compiler: /nix/store/gqq2vrcq7krqi9rrl6pphvsg81sb8pjw-
3. gcc-wrapper-7.3.0/bin/g++ -- works
4. -- Detecting CXX compiler ABI info
5. -- Detecting CXX compiler ABI info - done
6. -- Detecting CXX compile features
7. -- Detecting CXX compile features - done
8. -- Project will be installed to /home/roberto/Software/recipe-04
9. -- Build type set to Release
10. -- Installing LIB components to /home/roberto/Software/recipe-04/lib64
11. -- Installing BIN components to /home/roberto/Software/recipe-04/bin
12. -- Installing INCLUDE components to /home/roberto/Software/recipe-04/include
   -- Installing CMAKE components to /home/roberto/Software/recipe-
13. 04/share/cmake/recipe-04
   -- recipe-04 staged install: /home/roberto/Workspace/robetodr/cmake-cookbook/chapter-10/recipe-04/cxx-example/build/stage
14. 10/recipe-04/cxx-example/build/stage
15. -- Checking for one of the modules 'uuid'
   -- Found message: /home/roberto/Software/message/lib64/libmessage.so.1 (found
16. version 1.0.0)
17. -- Configuring done
18. -- Generating done
   -- Build files have been written to: /home/roberto/Workspace/robetodr/cmake-cookbook/chapter-10/recipe-04/cxx-example/build
19.
```

项目的最终安装规则是，将安装文件复制到 `CMAKE_INSTALL_PREFIX`：

```
1. install(
2.   DIRECTORY
3.     ${STAGED_INSTALL_PREFIX}/
4.   DESTINATION
5.   .
```

```

6.    USE_SOURCE_PERMISSIONS
7.  )

```

注意使用 `.` 而不是绝对路径 `${CMAKE_INSTALL_PREFIX}`，这样CPack工具就可以正确理解该规则。CPack的用法将在第11章中介绍。

`recipe-04_core` 项目构建一个简单的可执行目标，该目标链接到消息动态库。正如本章前几节所讨论，为了让可执行文件正确运行，需要正确设置 `RPATH`。本章的第1节展示了，如何在CMake的帮助下实现这一点，同样的模式在 `CMakeLists.txt` 中被重用，用于创建 `use_message` 的可执行目标：

```

file(RELATIVE_PATH _rel ${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_BINDIR}
1. ${CMAKE_INSTALL_PREFIX})
2. if(APPLE)
3.   set(_rpath "@loader_path/${_rel}")
4. else()
5.   set(_rpath "\$ORIGIN/${_rel}")
6. endif()
7. file(TO_NATIVE_PATH "${_rpath}/${CMAKE_INSTALL_LIBDIR}" use_message_RPATH)
8.
9. set_target_properties(use_message
10. PROPERTIES
11.   MACOSX_RPATH ON
12.   SKIP_BUILD_RPATH OFF
13.   BUILD_WITH_INSTALL_RPATH OFF
14.   INSTALL_RPATH "${use_message_RPATH}"
15.   INSTALL_RPATH_USE_LINK_PATH ON
16. )

```

为了检查这是否合适，可以使用本机工具打印已安装的可执行文件的 `RPATH`。我们将对该工具的调用，封装到Python脚本中，并将其进一步封装到CMake脚本中。最后，使用 `SCRIPT` 关键字将CMake脚本作为安装规则调用：

```

1. if(UNIX)
2.   set(PRINT_SCRIPT "${CMAKE_CURRENT_LIST_DIR}/cmake/print_rpath.py")
3.   configure_file(cmake/install_hook.cmake.in install_hook.cmake @ONLY)
4.   install(
5.     SCRIPT
6.       ${CMAKE_CURRENT_BINARY_DIR}/install_hook.cmake
7.   )
8. endif()

```

脚本是在安装最后进行执行：

```
1. $ cmake --build build --target install
```

GNU/Linux系统上，我们将看到以下输出：

```
1. Install the project...
2. -- Install configuration: "Release"
3. -- Installing: /home/roberto/Software/recipe-04/
4. -- Installing: /home/roberto/Software/recipe-04/.lib64
5. -- Installing: /home/roberto/Software/recipe-04/.lib64/libmessage.so
6. -- Installing: /home/roberto/Software/recipe-04/.lib64/libmessage_s.a
7. -- Installing: /home/roberto/Software/recipe-04/.lib64/libmessage.so.1
8. -- Installing: /home/roberto/Software/recipe-04/.include
9. -- Installing: /home/roberto/Software/recipe-04/.include/message
10. -- Installing: /home/roberto/Software/recipe-04/.include/message/Message.hpp
    -- Installing: /home/roberto/Software/recipe-
11. 04/.include/message/messageExport.h
12. -- Installing: /home/roberto/Software/recipe-04./share
13. -- Installing: /home/roberto/Software/recipe-04./share/cmake
14. -- Installing: /home/roberto/Software/recipe-04./share/cmake/message
    -- Installing: /home/roberto/Software/recipe-
15. 04./share/cmake/message/messageTargets-release.cmake
    -- Installing: /home/roberto/Software/recipe-
16. 04./share/cmake/message/messageConfigVersion.cmake
    -- Installing: /home/roberto/Software/recipe-
17. 04./share/cmake/message/messageConfig.cmake
    -- Installing: /home/roberto/Software/recipe-
18. 04./share/cmake/message/messageTargets.cmake
19. -- Installing: /home/roberto/Software/recipe-04./bin
20. -- Installing: /home/roberto/Software/recipe-04./bin/hello-world_wAR
21. -- Installing: /home/roberto/Software/recipe-04./bin/use_message
22. -- Installing: /home/roberto/Software/recipe-04./bin/hello-world_wDSO
23. -- ELF patching tool chrpath FOUND
    -- RPATH for /home/roberto/Software/recipe-04/bin/use_message is /home/roberto/So
24. 2.31.1/lib:/nix/store/2kcrj1ksd2a14bm5sky182fv2xwfhfap-glibc-2.26-131/lib:/nix/st
25. -- Running /home/roberto/Software/recipe-04/bin/use_message:
26. This is my very nice message:
27. Hello, World! From a client of yours!
28. ...and here is its UUID: a8014bf7-5dfa-45e2-8408-12e9a5941825
29. This is my very nice message:
30. Goodbye, World! From a client of yours!
31. ...and here is its UUID: ac971ef4-7606-460f-9144-1ad96f713647
```

NOTE: 我们建议使用的工具是 *PatchELF* (<https://nixos.org/patchelf.html>)、*chrpath* (<https://linux.die.net/man/1/chrpath>) 和 *otool* (<http://www.manpagez.com/man/1/otool/>)。第一种方法适用于 *GNU/Linux* 和 *macOS*，而 *chrpath* 和 *otool* 分别适用于 *GNU/Linux* 和 *macOS*。

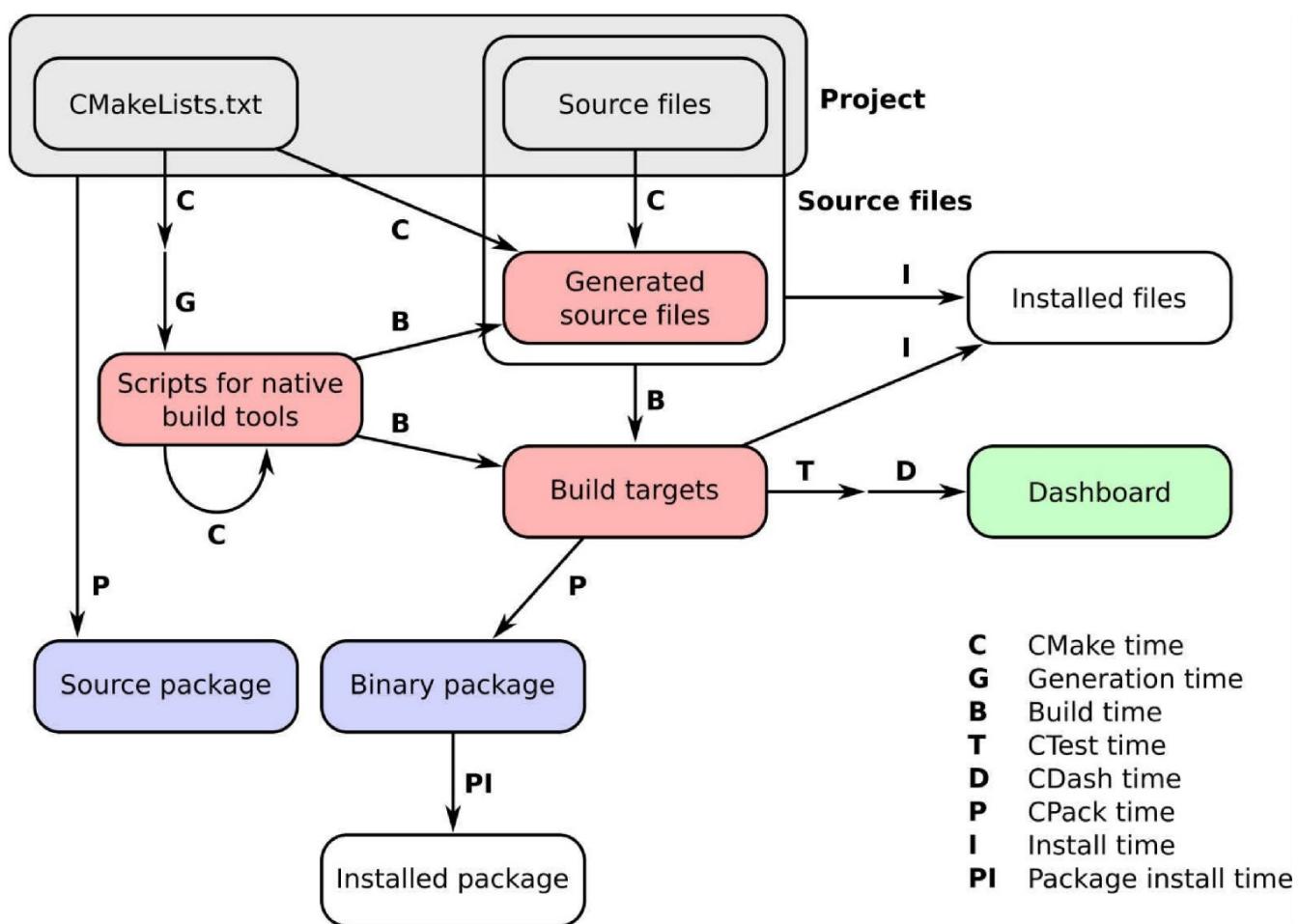
第11章 打包项目

本章的主要内容有：

- 生成源代码和二进制包
- 使用CMake/pybind11构建的C++/Python项目，通过PyPI发布
- 使用CMake/CFFI构建C/Fortran/Python项目，通过PyPI发布
- 以Conda包的形式发布一个简单的项目
- 将Conda包作为依赖项发布给项目

目前为止，已经从源代码编译并安装了软件包—这意味着可以通过Git获取项目，并手动执行配置、构建、测试和安装。然而，在实际中，软件包通常是使用管理器来安装的，比如Apt、DNF、Pacman、pip和Conda。我们需要以各种格式发布我们的代码项目—作为源文件或二进制安装程序。

下图显示了使用CMake的项目的各个阶段，我们现在方案就是其中的打包时(PI)：



本章中，我们将探讨不同的打包策略。首先，讨论使用CMake中的工具CPack进行打包，还提供打包和上传CMake项目到Python包索引(PyPI, <https://pypi.org>)和Anaconda云(<https://anaconda.org>)的方法，这些都是通过包管理器pip和Conda

(<https://conda.io/docs/>)分发包的平台。对于PyPI，我们将演示如何打包和分发混合C++/Python或C/Fortran/Python的项目。对于Conda，我们将展示如何对依赖于其他库的C++项目进行打包。

11.1 生成源代码和二进制包

NOTE: 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-11/recipe-01> 中找到。该示例在CMake 3.6版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

如果代码是开源的，用户将能够下载项目的源代码，并使用完全定制的CMake脚本自行构建。当然，打包操作也可以使用脚本完成，但是CPack提供了更简单和可移植的替代方案。本示例将指导您创建一些包：

- 源代码包：可以将源代码直接压缩成归档文件，进行发送。用户将不必为特定的版本控制系统操心。
- 二进制包：工具将新构建的目标以打包的方式到归档文件中。这个功能非常有用，但可能不够健壮，无法发布库和可执行程序。
- 平台原生的二进制安装：CPack能够以许多不同的格式生成二进制安装程序，因此可以将软件发布到不同的平台。我们将展示如何生成安装程序：
 - 基于Debian的GNU/Linux发行版的 `.deb` 格式：
<https://manpages.debian.org/unstable/dpkg-dev/deb.5.en.html>
 - 基于Red Hat的GNU/Linux发行版的 `.rpm` 格式：<http://rpm.org/>
 - macOS包的 `.dmg` 格式：
<https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html>
 - Windows的NSIS格式：http://nsis.sourceforge.net/Main_Page

准备工作

我们将使用第10章第3节的示例，项目树由以下目录和文件组成：

```

1. .
2. └── cmake
3.   ├── coffee.icns
4.   ├── Info.plist.in
5.   └── messageConfig.cmake.in
6. └── CMakeCPack.cmake
7. └── CMakeLists.txt
8. └── INSTALL.md
9. └── LICENSE
10. └── src
11.   └── CMakeLists.txt

```

```

12. |   └── hello-world.cpp
13. |   └── Message.cpp
14. |   └── Message.hpp
15. └── tests
16.   └── CMakeLists.txt
17.   └── use_target
18.     └── CMakeLists.txt
19.     └── use_message.cpp

```

由于本示例的重点是使用CPack，所以不会讨论源码。我们只会在 `CMakeCPack.cmake` 中添加打包指令。此外，还添加了 `INSTALL.md` 和 `LICENSE` 文件：打包要求需要包含安装说明和项目许可信息。

具体实施

让我们看看需要添加到这个项目中的打包指令。我们将在 `CMakeCPack.cmake` 中收集它们，并在 `CMakeLists.txt` 的末尾包含这个模块 `include(cmakecpack.cmake)`：

1. 我们声明包的名称，与项目的名称相同，因此我们使用 `PROJECT_NAME` 的CMake变量：

```
1. set(CPACK_PACKAGE_NAME "${PROJECT_NAME}")
```

2. 声明包的供应商：

```
1. set(CPACK_PACKAGE_VENDOR "CMake Cookbook")
```

3. 打包的源代码将包括一个描述文件。这是带有安装说明的纯文本文件：

```
1. set(CPACK_PACKAGE_DESCRIPTION_FILE "${PROJECT_SOURCE_DIR}/INSTALL.md")
```

4. 还添加了一个包的描述：

```
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "message: a small messaging
1. library")
```

5. 许可证文件也将包括在包中：

```
1. set(CPACK_RESOURCE_FILE_LICENSE "${PROJECT_SOURCE_DIR}/LICENSE")
```

6. 从发布包中安装时，文件将放在 `/opt/recipe-01` 目录下：

```
1. set(CPACK_PACKAGING_INSTALL_PREFIX "/opt/${PROJECT_NAME}")
```

7. CPack所需的主要、次要和补丁版本：

```
1. set(CPACK_PACKAGE_VERSION_MAJOR "${PROJECT_VERSION_MAJOR}")
2. set(CPACK_PACKAGE_VERSION_MINOR "${PROJECT_VERSION_MINOR}")
3. set(CPACK_PACKAGE_VERSION_PATCH "${PROJECT_VERSION_PATCH}")
```

8. 设置了在包装的时候需要忽略的文件列表和目录：

```
1. set(CPACK_SOURCE_IGNORE_FILES "${PROJECT_BINARY_DIR};/.git/;.gitignore")
```

9. 列出了源代码归档的打包生成器—在我们的例子中是 `ZIP`，用于生成 `.ZIP` 归档，`TGZ` 用于 `.tar.gz` 归档：

```
1. set(CPACK_SOURCE_GENERATOR "ZIP;TGZ")
```

10. 我们还列出了二进制存档生成器：

```
1. set(CPACK_GENERATOR "ZIP;TGZ")
```

11. 现在也可声明平台原生二进制安装程序，从DEB和RPM包生成器开始，不过只适用于GNU/Linux：

```
1. if(UNIX)
2.   if(CMAKE_SYSTEM_NAME MATCHES Linux)
3.     list(APPEND CPACK_GENERATOR "DEB")
4.     set(CPACK_DEBIAN_PACKAGE_MAINTAINER "robertodr")
5.     set(CPACK_DEBIAN_PACKAGE_SECTION "devel")
6.     set(CPACK_DEBIAN_PACKAGE_DEPENDS "uuid-dev")
7.
8.     list(APPEND CPACK_GENERATOR "RPM")
9.     set(CPACK_RPM_PACKAGE_RELEASE "1")
10.    set(CPACK_RPM_PACKAGE_LICENSE "MIT")
11.    set(CPACK_RPM_PACKAGE_REQUIRES "uuid-devel")
12.  endif()
13. endif()
```

12. 如果我们在Windows上，我们会想要生成一个NSIS安装程序：

```
1. if(WIN32 OR MINGW)
```

```

2. list(APPEND CPACK_GENERATOR "NSIS")
3. set(CPACK_NSIS_PACKAGE_NAME "message")
4. set(CPACK_NSIS_CONTACT "robertdr")
5. set(CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL ON)
6. endif()

```

13. 另一方面，在macOS上，bundle包是我们的安装程序的选择：

```

1. if(APPLE)
2. list(APPEND CPACK_GENERATOR "Bundle")
3. set(CPACK_BUNDLE_NAME "message")
configure_file(${PROJECT_SOURCE_DIR}/cmake/Info.plist.in Info.plist
4. @ONLY)
5. set(CPACK_BUNDLE_PLIST ${CMAKE_CURRENT_BINARY_DIR}/Info.plist)
6. set(CPACK_BUNDLE_ICON ${PROJECT_SOURCE_DIR}/cmake/coffee.icns)
7. endif()

```

14. 我们在现有系统的包装生成器上，向用户打印一条信息：

```
1. message(STATUS "CPack generators: ${CPACK_GENERATOR}")
```

15. 最后，我们包括了 `CPack.cmake` 标准模块。这将向构建系统添加一个包和一个 `package_source` 目标：

```
1. include(CPack)
```

现在来配置这个项目：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..

```

使用下面的命令，我们可以列出可用的目标(示例输出是在使用Unix Makefile作为生成器的GNU/Linux系统上获得的)：

```

1. $ cmake --build . --target help
2.
3. The following are some of the valid targets for this Makefile:
4. ... all (the default if no target is provided)
5. ... clean
6. ... depend

```

```

7. ... install/striр
8. ... install
9. ... package_source
10. ... package
11. ... install/local
12. ... test
13. ... list_install_components
14. ... edit_cache
15. ... rebuild_cache
16. ... hello-world
17. ... message

```

我们可以看到 `package` 和 `package_source` 目标是可用的。可以使用以下命令生成源包：

```

1. $ cmake --build . --target package_source
2.
3. Run CPack packaging tool for source...
4. CPack: Create package using ZIP
5. CPack: Install projects
   CPack: - Install directory: /home/user/cmake-cookbook/chapter-11/recipe-01/cxx-
6. example
7. CPack: Create package
   CPack: - package: /home/user/cmake-cookbook/chapter- 11/recipe-01/cxx-
8. example/build/recipe-01-1.0.0-Source.zip generated.
9. CPack: Create package using TGZ
10. CPack: Install projects
    CPack: - Install directory: /home/user/cmake-cookbook/chapter- 11/recipe-
11. 01/cxx-example
12. CPack: Create package
    CPack: - package: /home/user/cmake-cookbook/chapter-11/recipe-01/cxx-
13. example/build/recipe-01- 1.0.0-Source.tar.gz generated.

```

同样，也可以构建二进制包：

```
1. $ cmake --build . --target package message-1.0.0-Linux.deb
```

例子中，最后得到了以下二进制包：

```

1. message-1.0.0-Linux.rpm
2. message-1.0.0-Linux.tar.gz
3. message-1.0.0-Linux.zip

```

工作原理

CPack可用于生成用于分发的包。生成构建系统时，我们在 `CMakeCPack.cmake` 中列出了CPack指令，用于在构建目录下生成 `CPackConfig.cmake`。当运行以 `package` 或 `package_source` 目标的CMake命令时，CPack会自动调用，参数是自动生成的配置文件。实际上，这两个新目标是对CPack简单规则的使用。与CMake一样，CPack也有生成器的概念。CMake上下文中的生成器是用于生成本地构建脚本的工具，例如Unix Makefile或Visual Studio项目文件，而CPack上下文中的生成器是用于打包的工具。我们列出了这些变量，并对不同的平台进行了特别的关注，为源包和二进制包定义了 `CPACK_SOURCE_GENERATOR` 和 `CPACK_GENERATOR` 变量。因此，`DEB` 包生成器将调用 `Debian` 打包实用程序，而 `TGZ` 生成器将调用给定平台上的归档工具。我们可以直接在 `build` 目录中调用CPack，并选择要与 `-G` 命令行选项一起使用的生成器。`RPM` 包可以通过以下步骤生成：

```

1. $ cd build
2. $ cpack -G RPM
3.
4. CPack: Create package using RPM
5. CPack: Install projects
6. CPack: - Run preinstall target for: recipe-01
7. CPack: - Install project: recipe-01
8. CPack: Create package
   CPackRPM: Will use GENERATED spec file: /home/user/cmake-cookbook/chapter-11/reci
9. example/build/_CPack_Packages/Linux/RPM/SPECS/recipe-01.spec
   CPack: - package: /home/user/cmake-cookbook/chapter-11/recipe-01/cxx-example/buil
10. 01-1.0.0-Linux.rpm generated.

```

对于任何发行版，无论是源代码还是二进制文件，我们只需要打包用户需要的内容，因此整个构建目录和其他与版本控制相关的文件，都必须从要打包的文件列表中排除。我们的例子中，排除列表使用下面的命令声明：

```
1. set(CPACK_SOURCE_IGNORE_FILES "${PROJECT_BINARY_DIR};/.git/;.gitignore")
```

我们还需要指定包的基本信息，例如：名称、简短描述和版本。这个信息是通过CMake变量设置的，当包含相应的模块时，CMake变量被传递给CPack。

NOTE:由于CMake 3.9中的 `project()` 命令接受 `DESCRIPTION` 字段，该字段带有一个描述项目的短字符串。CMake将设置一个 `PROJECT_DESCRIPTION`，可以用它来重置 `CPACK_PACKAGE_DESCRIPTION_SUMMARY`。

让我们详细看看，可以为示例项目生成的不同类型的包的说明。

打包源码

我们的示例中，决定对源存档使用 `TGZ` 和 `ZIP` 生成器。这些文件将分别生成 `.tar.gz` 和 `.zip` 压缩文件。我们可以检查生成的 `.tar.gz` 文件的内容：

```
1. $ tar tf recipe-01-1.0.0-Source.tar.gz
2.
3. recipe-01-1.0.0-Source/opt/
4. recipe-01-1.0.0-Source/opt/recipe-01/
5. recipe-01-1.0.0-Source/opt/recipe-01/cmake/
6. recipe-01-1.0.0-Source/opt/recipe-01/cmake/coffee.icns
7. recipe-01-1.0.0-Source/opt/recipe-01/cmake/Info.plist.in
8. recipe-01-1.0.0-Source/opt/recipe-01/cmake/messageConfig.cmake.in
9. recipe-01-1.0.0-Source/opt/recipe-01/CMakeLists.txt
10. recipe-01-1.0.0-Source/opt/recipe-01/src/
11. recipe-01-1.0.0-Source/opt/recipe-01/src/Message.hpp
12. recipe-01-1.0.0-Source/opt/recipe-01/src/CMakeLists.txt
13. recipe-01-1.0.0-Source/opt/recipe-01/src/Message.cpp
14. recipe-01-1.0.0-Source/opt/recipe-01/src/hello-world.cpp
15. recipe-01-1.0.0-Source/opt/recipe-01/LICENSE
16. recipe-01-1.0.0-Source/opt/recipe-01/tests/
17. recipe-01-1.0.0-Source/opt/recipe-01/tests/CMakeLists.txt
18. recipe-01-1.0.0-Source/opt/recipe-01/tests/use_target/
19. recipe-01-1.0.0-Source/opt/recipe-01/tests/use_target/CMakeLists.txt
20. recipe-01-1.0.0-Source/opt/recipe-01/tests/use_target/use_message.cpp
21. recipe-01-1.0.0-Source/opt/recipe-01/INSTALL.md
```

与预期相同，只包含源码树的内容。注意 `INSTALL.md` 和 `LICENSE` 文件也包括在内，可以通过 `CPACK_PACKAGE_DESCRIPTION_FILE` 和 `CPACK_RESOURCE_FILE_LICENSE` 变量指定。

NOTE: Visual Studio 生成器无法解析 `package_source` 目标：<https://gitlab.kitware.com/cmake/cmake/issues/13058>。

二进制包

创建二进制存档时，CPack将打包 `CMakeCPack.cmake` 中描述的目标的内容。因此，在我们的示例中，`hello-world`可执行文件、消息动态库以及相应的头文件都将以 `.tar.gz` 和 `.zip` 的格式打包。此外，还将打包CMake配置文件。这对于需要链接到我们的库的其他项目非常有用。包中使用的安装目录可能与从构建树中安装项目时使用的前缀不同，可以使 `CPACK_PACKAGING_INSTALL_PREFIX` 变量来实现这一点。我们的示例中，我们将它设置为系统上的特定位置：`/opt/recipe-01`。

```

1. $ tar tf recipe-01-1.0.0-Linux.tar.gz
2.
3. recipe-01- 1.0.0-Linux/opt/
4. recipe-01-1.0.0-Linux/opt/recipe-01/
5. recipe-01-1.0.0- Linux/opt/recipe-01/bin/
6. recipe-01-1.0.0-Linux/opt/recipe-01/bin/hello-world
7. recipe-01-1.0.0-Linux/opt/recipe-01/share/
8. recipe-01-1.0.0- Linux/opt/recipe-01/share/cmake/
9. recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-01/
10. recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-01/messageConfig.cmake
    recipe-01-1.0.0- Linux/opt/recipe-01/share/cmake/recipe-01/messageTargets-
11. hello-world.cmake
    recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-
12. 01/messageConfigVersion.cmake
    recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-01/messageTargets-
13. hello-world-release.cmake
    recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-01/messageTargets-
14. release.cmake
15. recipe-01-1.0.0-Linux/opt/recipe-01/share/cmake/recipe-01/messageTargets.cmake
16. recipe-01-1.0.0- Linux/opt/recipe-01/include/
17. recipe-01-1.0.0-Linux/opt/recipe-01/include/message/
18. recipe-01-1.0.0-Linux/opt/recipe-01/include/message/Message.hpp
19. recipe-01-1.0.0-Linux/opt/recipe-01/include/message/messageExport.h
20. recipe-01-1.0.0-Linux/opt/recipe-01/lib64/
21. recipe-01-1.0.0-Linux/opt/recipe-01/lib64/libmessage.so
22. recipe-01-1.0.0-Linux/opt/recipe-01/lib64/libmessage.so.1`
```

平台原生的二进制安装

我们希望每个平台原生二进制安装程序的配置略有不同。可以在单个 `CMakeCPack.cmake` 中使用 CPack 管理这些差异，就像例子中做的那样。

对于 GNU/Linux 系统，配置了 `DEB` 和 `RPM` 生成器：

```

1. if(UNIX)
2.   if(CMAKE_SYSTEM_NAME MATCHES Linux)
3.     list(APPEND CPACK_GENERATOR "DEB")
4.     set(CPACK_DEBIAN_PACKAGE_MAINTAINER "robertodr")
5.     set(CPACK_DEBIAN_PACKAGE_SECTION "devel")
6.     set(CPACK_DEBIAN_PACKAGE_DEPENDS "uuid-dev")
7.
8.     list(APPEND CPACK_GENERATOR "RPM")
```

```

9.      set(CPACK_RPM_PACKAGE_RELEASE "1")
10.     set(CPACK_RPM_PACKAGE_LICENSE "MIT")
11.     set(CPACK_RPM_PACKAGE_REQUIRES "uuid-devel")
12.   endif()
13. endif()

```

我们的示例依赖于UUID

库，`CPACK_DEBIAN_PACKAGE_DEPENDS` 和 `cpack_rpm_package_require` 选项允许指定，包和数据库中对其他包的依赖关系。可以使用`dpkg`和`rpm`程序分别分析生成的 `.deb` 和 `.rpm` 包的内容。

注意，`CPACK_PACKAGING_INSTALL_PREFIX` 也会影响这些包生成器：我们的包将安装到 `/opt/recipe-01`。

CMake真正提供了跨平台和可移植构建系统的支持。下面将使用Nullsoft脚本安装系统(NSIS)创建一个安装程序：

```

1. if(WIN32 OR MINGW)
2. list(APPEND CPACK_GENERATOR "NSIS")
3. set(CPACK_NSIS_PACKAGE_NAME "message")
4. set(CPACK_NSIS_CONTACT "robertdr")
5. set(CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL ON)
6. endif()

```

如果在macOS上构建项目，将启用 `Bundle packager`：

```

1. if(APPLE)
2. list(APPEND CPACK_GENERATOR "Bundle")
3. set(CPACK_BUNDLE_NAME "message")
4. configure_file(${PROJECT_SOURCE_DIR}/cmake/Info.plist.in Info.plist @ONLY)
5. set(CPACK_BUNDLE_PLIST ${CMAKE_CURRENT_BINARY_DIR}/Info.plist)
6. set(CPACK_BUNDLE_ICON ${PROJECT_SOURCE_DIR}/cmake/coffee.icns)
7. endif()

```

macOS的示例中，需要为包配置属性列表文件，这是通过 `configure_file` 实现的。`Info.plist` 的位置和包的图标，这些都可以通过CPack的变量进行设置。

NOTE:可以在这里阅读，关于属性列表格式的更多信息：https://en.wikipedia.org/wiki/Property_list

更多信息

对 `CMakeCPack.cmake` 进行设置，要比列出CPack的配置选项简单的多，我们可以将 `CPACK_*` 变

量的每个生成器设置放在单独的文件中，比如 `CMakeCPackOptions.cmake`，并将这些设置包含到 `CMakeCPack.cmake` 使用 `set(CPACK_PROJECT_CONFIG_FILE "${PROJECT_SOURCE_DIR}/CMakeCPackOptions.cmake")` 将设置包含入 `CMakeCPack.cmake` 中。还可以在CMake时配置该文件，然后在CPack时包含该文件，这为配置多格式包生成器提供了一种简洁的方法(参见<https://cmake.org/cmake/help/v3.6/module/CPack.html>)。

与CMake中的所有工具一样，CPack功能强大、功能多样，并且提供了更多的灵活性和选项。感兴趣的读者应该看官方文档的命令行界面CPack

(<https://cmake.org/cmake/help/v3.6/manual/cpack.1.html>)手册页，如何使用CPack生成器打包相关项目的更多细节(<https://cmake.org/cmake/help/v3.6/module/CPack.html>)。

11.2 通过PyPI发布使用CMake/pybind11构建的C++/Python项目

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-11/recipe-02> 中找到。该示例在CMake 3.11版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例中，我们将以第9章第5节的代码的pybind11为例，为其添加相关的安装目标和pip打包信息，并将项目上传到PyPI。我们要实现一个可以使用pip安装，并运行CMake从而获取底层pybind11依赖项的项目。

准备工作

要通过PyPI分发包的话，需要一个<https://pypi.org> 帐户。当然，也可以先从本地路径进行安装练习。

TIPS:建议使用Pipenv (<https://docs.pipenv.org>)或虚拟环境 (<https://virtualenv.pypa>)安装这个包和其他的Python包。

我们基于第9章第5节的项目，它包含一个主 `CMakeLists.txt` 文件和一个 `account/CMakeLists.txt` 文件，配置帐户示例目标时，使用如下的项目树：

```

1. .
2. └── account
3.   ├── account.cpp
4.   ├── account.hpp
5.   └── CMakeLists.txt
6.   └── test.py
7. └── CMakeLists.txt

```

示例中，`account.cpp`，`account.hpp` 和 `test.py` 没有任何变化。修改 `account/CMakeLists.txt`，并为pip添加几个文件，以便能够构建安装包。为此，需要根目录中的另外三个文件：`README.rst`，`MANIFEST.in` 和 `setup.py`。

`README.rst` 中包含关于项目的s文档：

```

1. Example project
2. =====
3.
4. Project description in here ...

```

MANIFEST.in 列出了需要安装的Python模块:

```
1. include README.rst CMakeLists.txt
2. recursive-include account *.cpp *.hpp CMakeLists.txt
```

最后，**setup.py** 包含构建指令和安装项目的说明:

```
1. import distutils.command.build as _build
2. import os
3. import sys
4. from distutils import spawn
5. from distutils.sysconfig import get_python_lib
6. from setuptools import setup
7.
8.
9. def extend_build():
10.     class build(_build.build):
11.         def run(self):
12.             cwd = os.getcwd()
13.             if spawn.find_executable('cmake') is None:
14.                 sys.stderr.write("CMake is required to build this package.\n")
15.                 sys.exit(-1)
16.             _source_dir = os.path.split(__file__)[0]
17.             _build_dir = os.path.join(_source_dir, 'build_setup_py')
18.             _prefix = get_python_lib()
19.             try:
20.                 cmake_configure_command = [
21.                     'cmake',
22.                     '-H{0}'.format(_source_dir),
23.                     '-B{0}'.format(_build_dir),
24.                     '-DCMAKE_INSTALL_PREFIX={0}'.format(_prefix),
25.                 ]
26.                 _generator = os.getenv('CMAKE_GENERATOR')
27.                 if _generator is not None:
28.                     cmake_configure_command.append(
29.                         '-G{0}'.format(_generator))
30.                     spawn.spawn(cmake_configure_command)
31.                     spawn.spawn(
32.                         ['cmake', '--build', _build_dir, '--target', 'install'])
33.                     os.chdir(cwd)
34.             except spawn.DistutilsExecError:
35.                 sys.stderr.write("Error while building with CMake\n")
```

```

36.         sys.exit(-1)
37.         _build.build.run(self)
38.     return build
39.
40. _here = os.path.abspath(os.path.dirname(__file__))
41.
42. if sys.version_info[0] < 3:
43.     with open(os.path.join(_here, 'README.rst')) as f:
44.         long_description = f.read()
45. else:
46.     with open(os.path.join(_here, 'README.rst'), encoding='utf-8') as f:
47.         long_description = f.read()
48.
49. _this_package = 'account'
50.
51. version = {}
52. with open(os.path.join(_here, _this_package, 'version.py')) as f:
53.     exec(f.read(), version)
54.
55. setup(
56.     name=_this_package,
57.     version=version['__version__'],
58.     description='Description in here.',
59.     long_description=long_description,
60.     author='Bruce Wayne',
61.     author_email='bruce.wayne@example.com',
62.     url='http://example.com',
63.     license='MIT',
64.     packages=[_this_package],
65.     include_package_data=True,
66.     classifiers=[
67.         'Development Status :: 3 - Alpha',
68.         'Intended Audience :: Science/Research',
69.         'Programming Language :: Python :: 2.7',
70.         'Programming Language :: Python :: 3.6'
71.     ],
72.     cmdclass={'build': extend_build()})

```

account 子目录中放置一个 `__init__.py` 脚本：

```

1. from .version import __version__
2. from .account import Account

```

```

3. __all__ = [
4.     '__version__',
5.     'Account',
6. ]

```

再放一个 `version.py` 脚本：

```
1. __version__ = '0.0.0'
```

项目的文件结构如下：

```

1. .
2. └── account
3.     ├── account.cpp
4.     ├── account.hpp
5.     ├── CMakeLists.txt
6.     ├── __init__.py
7.     ├── test.py
8.     └── version.py
9. └── CMakeLists.txt
10. └── MANIFEST.in
11. └── README.rst
12. └── setup.py

```

具体实施

本示例基于第9章第5节项目的基础上。

首先，修改 `account/CMakeLists.txt`，添加安装目标：

```

1. install(
2.   TARGETS
3.   account
4.   LIBRARY
5.   DESTINATION account
6. )

```

安装目标时，`README.rst`，

`MANIFEST.in`，`setup.py`、`__init__.py` 和 `version.py` 将放置在对应的位置上，我们准备使用pybind11测试安装过程：

1. 为此，在某处创建一个新目录，我们将在那里测试安装。
2. 在创建的目录中，从本地路径运行 `pipenv install`。调整本地路径，指向 `setup.py` 的目录：

```
1. $ pipenv install /path/to/cxx-example
```

3. 在Pipenv环境中打开一个Python shell：

```
1. $ pipenv run python
```

4. Python shell中，可以测试我们的CMake包：

```
1. >>> from account import Account
2. >>> account1 = Account()
3. >>> account1.deposit(100.0)
4. >>> account1.deposit(100.0)
5. >>> account1.withdraw(50.0)
6. >>> print(account1.get_balance())
7. 150.0
```

工作原理

`${CMAKE_CURRENT_BINARY_DIR}` 目录包含编译后的 `account.cpython-36m-x86_64-linux-gnu.so`，这个动态库就是使用pybind11构建Python模块。但是请注意，它的名称取决于操作系统（本例中是64位Linux）和Python环境（本例中是Python 3.6）。`setup.py` 脚本将运行CMake，并根据所选的Python环境（系统Python，Pipenv或虚拟环境）将Python模块安装到正确的路径下。

不过，在安装模块时面临两个挑战：

- 名称可变
- CMake外部设置路径

可以使用下面的安装目标来解决这个问题，将在`setup.py`中定义安装目标位置：

```
1. install(
2.   TARGETS
3.   account
4.   LIBRARY
5.   DESTINATION account
6. )
```

指示CMake将编译好的Python模块文件安装到相对于安装目标位置的 `account` 子目录中(第10章中详细讨论了如何设置目标位置)。`setup.py` 将通过设置 `CMAKE_INSTALL_PREFIX` 来设置安装位置，并根据Python环境指向正确的路径。

让我们看看 `setup.py` 如何实现的。自下而上来看一下脚本：

```

1. setup(
2.     name=_this_package,
3.     version=version['__version__'],
4.     description='Description in here.',
5.     long_description=long_description,
6.     author='Bruce Wayne',
7.     author_email='bruce.wayne@example.com',
8.     url='http://example.com',
9.     license='MIT',
10.    packages=[_this_package],
11.    include_package_data=True,
12.    classifiers=[
13.        'Development Status :: 3 - Alpha',
14.        'Intended Audience :: Science/Research',
15.        'Programming Language :: Python :: 2.7',
16.        'Programming Language :: Python :: 3.6'
17.    ],
18.    cmdclass={'build': extend_build()})

```

该脚本包含许多占位符，还包含一些自解释的语句。这里我们将重点介绍最后一个指令 `cmdclass`。这个指令中，通过自定义 `extend_build` 函数扩展默认的构建步骤。这个默认的构建步骤如下：

```

1. def extend_build():
2.     class build(_build.build):
3.         def run(self):
4.             cwd = os.getcwd()
5.             if spawn.find_executable('cmake') is None:
6.                 sys.stderr.write("CMake is required to build this package.\n")
7.                 sys.exit(-1)
8.             _source_dir = os.path.split(__file__)[0]
9.             _build_dir = os.path.join(_source_dir, 'build_setup_py')
10.            _prefix = get_python_lib()
11.            try:
12.                cmake_configure_command = [
13.                    'cmake',
14.                    '-H{0}'.format(_source_dir),

```

```

15.         '-B{0}'.format(_build_dir),
16.         '-DCMAKE_INSTALL_PREFIX={0}'.format(_prefix),
17.     ]
18.     _generator = os.getenv('CMAKE_GENERATOR')
19.     if _generator is not None:
20.         cmake_configure_command.append(
21.             '-G{0}'.format(_generator))
22.     spawn.spawn(cmake_configure_command)
23.     spawn.spawn(
24.         ['cmake', '--build', _build_dir, '--target', 'install'])
25.     os.chdir(cwd)
26. except spawn.DistutilsExecError:
27.     sys.stderr.write("Error while building with CMake\n")
28.     sys.exit(-1)
29.     _build.build.run(self)
30. return build

```

首先，检查CMake是否可用。函数执行了两个CMake命令：

```

1.     cmake_configure_command = [
2.         'cmake',
3.         '-H{0}'.format(_source_dir),
4.         '-B{0}'.format(_build_dir),
5.         '-DCMAKE_INSTALL_PREFIX={0}'.format(_prefix),
6.     ]
7.     _generator = os.getenv('CMAKE_GENERATOR')
8.     if _generator is not None:
9.         cmake_configure_command.append(
10.             '-G{0}'.format(_generator))
11.     spawn.spawn(cmake_configure_command)
12.     spawn.spawn(
13.         ['cmake', '--build', _build_dir, '--target', 'install'])

```

我们可以设置 `CMAKE_GENERATOR` 环境变量来修改生成器。安装目录如下方式设置：

```
1. _prefix = get_python_lib()
```

从安装目录的根目录下，通过 `distutils.sysconfig` 导入 `get_python_lib` 函数。`cmake --build _build_dir --target install` 命令以一种可移植的方式，构建和安装我们的项目。使用 `_build_dir` 而不使用 `build` 的原因是，在测试本地安装时，项目可能已经包含了一个 `build` 目录，这将与新安装过程发生冲突。对于已经上传到PyPI的包，构建目录的名称并不会带

来什么影响。

更多信息

现在我们已经测试了本地安装，准备将包上传到PyPI。在此之前，请确保 `setup.py` 中的元数据(例如：项目名称、联系方式和许可协议信息)是合理的，并且项目名称没有与PyPI已存在项目重名。在上传到<https://pypi.org> 之前，先测试PyPI(<https://test.pypi.org>)上，进行上载和下载的尝试。

上传之前，我们需要在主目录中创建一个名为 `.pypirc` 的文件，其中包含(替换成自己的 `yourusername` 和 `yourpassword`)：

```

1. [distutils]account
2. index-servers=
3. pypi
4. pypitest
5.
6. [pypi]
7. username = yourusername
8. password = yourpassword
9.
10. [pypitest]
11. repository = https://test.pypi.org/legacy/
12. username = yourusername
13. password = yourpassword

```

我们将分两步进行。首先，我们在本地创建Release包：

```
1. $ python setup.py sdist
```

第二步中，使用Twine上传生成的分布数据(我们将Twine安装到本地的Pipenv中)：

```

1. $ pipenv run twine upload dist/* -r pypitest
2.
3. Uploading distributions to https://test.pypi.org/legacy/
4. Uploading yourpackage-0.0.0.tar.gz

```

下一步，从测试实例到，将包安装到一个隔离的环境中：

```

1. $ pipenv shell
2. $ pip install --index-url https://test.pypi.org/simple/ yourpackage

```

当一切正常，就将我们的包上传到了PyPI：

```
1. $ pipenv run twine upload dist/* -r pypi
```

11.3 通过PyPI发布使用CMake/CFFI构建C/Fortran/Python项目

NOTE:此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-11/recipe-03> 中找到，其中有一个C++和Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

基于第9章第6节的示例，我们将重用前一个示例中的构建块，不过这次使用Python CFFI来提供Python接口，而不是pybind11。这个示例中，我们通过PyPI共享一个Fortran项目，这个项目可以是C或C++项目，也可以是任何公开C接口的语言，非Fortran就可以。

准备工作

项目将使用如下的目录结构：

```

1. .
2. └── account
3.   ├── account.h
4.   ├── CMakeLists.txt
5.   ├── implementation
6.     └── fortran_implementation.f90
7.   ├── __init__.py
8.   ├── interface_file_names.cfg.in
9.   ├── test.py
10.  └── version.py
11. └── CMakeLists.txt
12. └── MANIFEST.in
13. └── README.rst
14. └── setup.py

```

主 `CMakeLists.txt` 文件和 `account` 下面的所有源文件(`account/CMakeLists.txt` 除外)与第9章中的使用方式相同。`README.rst` 文件与前面的示例相同。`setup.py` 脚本比上一个示例多了一行(包含 `install_require =['cffi']` 的那一行)：

```

1. # ... up to this line the script is unchanged
2. setup(
3.   name=_this_package,
4.   version=version['__version__'],

```

```

5.     description='Description in here.',
6.     long_description=long_description,
7.     author='Bruce Wayne',
8.     author_email='bruce.wayne@example.com',
9.     url='http://example.com',
10.    license='MIT',
11.    packages=[_this_package],
12.    install_requires=['cffi'],
13.    include_package_data=True,
14.    classifiers=[
15.        'Development Status :: 3 - Alpha',
16.        'Intended Audience :: Science/Research',
17.        'Programming Language :: Python :: 2.7',
18.        'Programming Language :: Python :: 3.6'
19.    ],
20.    cmdclass={'build': extend_build()})

```

`MANIFEST.in` 应该与Python模块和包一起安装，并包含以下内容：

```

1. include README.rst CMakeLists.txt
2. recursive-include account *.h *.f90 CMakeLists.txt

```

`account` 子目录下，我们看到两个新文件。一个 `version.py` 文件，其为 `setup.py` 保存项目的版本信息：

```
1. __version__ = '0.0.0'
```

子目录还包含 `interface_file_names.cfg.in` 文件：

```

1. [configuration]
2. header_file_name = account.h
3. library_file_name = ${TARGET_FILE_NAME:account}

```

具体实施

讨论一下实现打包的步骤：

- 示例基于第9章第6节，使用Python CFFI扩展了 `account/CMakeLists.txt`，增加以下指令：

```

1. file(
2.     GENERATE OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/interface_file_names.cfg

```

```

3.     INPUT ${CMAKE_CURRENT_SOURCE_DIR}/interface_file_names.cfg.in
4.   )
5.
6. set_target_properties(account
7.   PROPERTIES
8.     PUBLIC_HEADER "account.h;${CMAKE_CURRENT_BINARY_DIR}/account_export.h"
9.     RESOURCE "${CMAKE_CURRENT_BINARY_DIR}/interface_file_names.cfg"
10.  )
11.
12. install(
13.   TARGETS
14.   account
15.   LIBRARY
16.   DESTINATION account/lib
17.   RUNTIME
18.   DESTINATION account/lib
19.   PUBLIC_HEADER
20.   DESTINATION account/include
21.   RESOURCE
22.   DESTINATION account
23. )

```

安装目标和附加文件准备好之后，就可以测试安装了。为此，会在某处创建一个新目录，我们将在那里测试安装。

- 新创建的目录中，我们从本地路径运行pipenv install。调整本地路径，指向 `setup.py` 脚本保存的目录：

```
1. $ pipenv install /path/to/fortran-example
```

- 现在在Pipenv环境中生成一个Python shell：

```
1. $ pipenv run python
```

- Python shell中，可以测试CMake包：

```

1. >>> import account
2. >>> account1 = account.new()
3. >>> account.deposit(account1, 100.0)
4. >>> account.deposit(account1, 100.0)
5. >>> account.withdraw(account1, 50.0)
6. >>> print(account.get_balance(account1))

```

```

7.
8. 150.0

```

工作原理

使用Python CFFI和CMake安装混合语言项目的扩展与第9章第6节的例子相对比，和使用Python CFFI的Python包多了两个额外的步骤：

1. 需要 `setup.py`
2. 安装目标时，FFI所需的头文件和动态库文件，需要安装在正确的路径中，具体路径取决于所选择的Python环境

`setup.py` 的结构与前面的示例几乎一致，唯一的修改是包含 `install_requires = ['cffi']`，以确保安装示例包时，也获取并安装了所需的Python CFFI。`setup.py` 脚本会自动安装 `__init__.py` 和 `version.py`。`MANIFEST.in` 中的改变不仅有 `README.rst` 和CMake文件，还有头文件和Fortran源文件：

- ```

1. include README.rst CMakeLists.txt
2. recursive-include account *.h *.f90 CMakeLists.txt

```

这个示例中，使用Python CFFI和 `setup.py` 打包CMake项目时，我们会面临三个挑战：

- 需要将 `account.h` 和 `account_export.h` 头文件，以及动态库复制到系统环境中Python模块的位置。
- 需要告诉 `__init__.py`，在哪里可以找到这些头文件和库。第9章第6节中，我们使用环境变量解决了这些问题，不过使用Python模块时，不可能每次去都设置这些变量。
- Python方面，我们不知道动态库文件的确切名称(后缀)，因为这取决于操作系统。

让我们从最后一点开始说起：不知道确切的名称，但在CMake生成构建系统时是知道的，因此我们在 `interface_file_names.cfg.in` 中使用生成器表达式，对占位符进行展开：

- ```

1. [configuration]
2. header_file_name = account.h
3. library_file_name = ${TARGET_FILE_NAME:account}

```

输入文件用来生成 `${CMAKE_CURRENT_BINARY_DIR}/interface_file_names.cfg`：

- ```

1. file(
2. GENERATE OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/interface_file_names.cfg
3. INPUT ${CMAKE_CURRENT_SOURCE_DIR}/interface_file_names.cfg.in
4.)

```

然后，将两个头文件定义为 `PUBLIC_HEADER` (参见第10章)，配置文件定义为 `RESOURCE`：

```

1. set_target_properties(account
2. PROPERTIES
3. PUBLIC_HEADER "account.h;${CMAKE_CURRENT_BINARY_DIR}/account_export.h"
4. RESOURCE "${CMAKE_CURRENT_BINARY_DIR}/interface_file_names.cfg"
5.)

```

最后，将库、头文件和配置文件安装到 `setup.py` 定义的安装路径中：

```

1. install(
2. TARGETS
3. account
4. LIBRARY
5. DESTINATION account/lib
6. RUNTIME
7. DESTINATION account/lib
8. PUBLIC_HEADER
9. DESTINATION account/include
10. RESOURCE
11. DESTINATION account
12.)

```

注意，我们为库和运行时都设置了指向 `account/lib` 的目标。这对于Windows很重要，因为动态库具有可执行入口点，因此我们必须同时指定这两个入口点。

Python包将能够找到这些文件，要使用 `account/__init__.py` 来完成：

```

1. # this interface requires the header file and library file
2. # and these can be either provided by interface_file_names.cfg
3. # in the same path as this file
4. # or if this is not found then using environment variables
5. _this_path = Path(os.path.dirname(os.path.realpath(__file__)))
6. _cfg_file = _this_path / 'interface_file_names.cfg'
7. if _cfg_file.exists():
8. config = ConfigParser()
9. config.read(_cfg_file)
10. header_file_name = config.get('configuration', 'header_file_name')
11. _header_file = _this_path / 'include' / header_file_name
12. _header_file = str(_header_file)
13. library_file_name = config.get('configuration', 'library_file_name')
14. _library_file = _this_path / 'lib' / library_file_name

```

```
15. _library_file = str(_library_file)
16. else:
17. _header_file = os.getenv('ACCOUNT_HEADER_FILE')
18. assert _header_file is not None
19. _library_file = os.getenv('ACCOUNT_LIBRARY_FILE')
20. assert _library_file is not None
```

本例中，将找到 `_cfg_file` 并进行解析，`setup.py` 将找到 `include` 下的头文件和 `lib` 下的库，并将它们传递给CFFI，从而构造库对象。这也是为什么，使用 `lib` 作为安装目标 `DESTINATION`，而不使用 `CMAKE_INSTALL_LIBDIR` 的原因(否则可能会让 `account/__init__.py` 混淆)。

## 更多信息

将包放到PyPI测试和生产实例中的后续步骤，因为有些步骤是类似的，所以可以直接参考前面的示例。

## 11.4 以Conda包的形式发布一个简单的项目

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-11/recipe-04> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

虽然PyPI是发布Python包的标准平台，但Anaconda (<https://anaconda.org>) 更为可能更为流行，因为它不仅允许使用Python接口发布Python或混合项目，还允许对非Python项目进行打包和依赖关系管理。这个示例中，我们将为一个非常简单的C++示例项目准备一个Conda包，该项目使用CMake配置和构建，除了C++之外没有依赖关系。下一个示例中，我们将会来看看一个更复杂的Conda包。

### 准备工作

我们的目标是打包以下示例代码(`example.cpp`)：

```
1. #include <iostream>
2. int main() {
3. std::cout << "hello from your conda package!" << std::endl;
4. return 0;
5. }
```

### 具体实施

1. `CMakeLists.txt` 文件给出了最低版本要求、项目名称和支持的语言：

```
1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-04 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 使用 `example.cpp` 构建 `hello-conda` 可执行目标：

```
1. add_executable(hello-conda "")
2. target_sources(hello-conda
3. PRIVATE
4. example.cpp
5.)
```

### 3. 使用 `CMakeLists.txt` 定义安装目标:

```

1. install(
2. TARGETS
3. hello-conda
4. DESTINATION
5. bin
6.)

```

### 4. 将在一个名为 `meta.yaml` 的文件中, 对Conda包进行描述。我们将把它放在 `conda-recipe` 目录下, 文件结构如下:

```

1. .
2. └── CMakeLists.txt
3. └── conda-recipe
4. └── meta.yaml
5. └── example.cpp

```

### 5. `meta.yaml` 包含如下内容:

```

1. package:
2. name: conda-example-simple
3. version: "0.0.0"
4.
5. source:
6. path: ... / # this can be changed to git-url
7.
8. build:
9. number: 0
10. binary_relocation: true
11. script:
12. - cmake -H -Bbuild_conda -G "${CMAKE_GENERATOR}" -
13. DCMAKE_INSTALL_PREFIX=${PREFIX} # [not win]
14. - cmake -H -Bbuild_conda -G "%CMAKE_GENERATOR%" -
15. DCMAKE_INSTALL_PREFIX="%LIBRARY_PREFIX%" # [win]
16. - cmake - -build build_conda - -target install
17. requirements:
18. build:
19. - cmake >=3.5
20. - { { compiler('cxx') } }

```

```

20.
21. about:
22. home: http://www.example.com
23. license: MIT
24. summary: "Summary in here ..."
```

## 6. 现在来构建包:

- \$ conda build conda-recipe

## 7. 过程中屏幕上看到大量输出，但是一旦构建完成，就可以对包进行安装。首先，在本地进行测试：

- \$ conda install --use-local conda-example-simple

## 8. 现在准备测试安装包，打开一个新的终端(假设Anaconda处于激活状态)，并输入以下内容：

- \$ hello-conda
- 
- hello from your conda package!

## 9. 测试成功后，再移除包装：

- \$ conda remove conda-example-simple

# 工作原理

`CMakeLists.txt` 中，安装目标是这个示例的一个基本组件：

```

1. install(
2. TARGETS
3. hello-conda
4. DESTINATION
5. bin
6.)
```

目标的二进制文件会安装到  `${CMAKE_INSTALL_PREFIX}/bin` 中。变量由Conda定义，并且构建步骤中定义在 `meta.yaml` :

- build:

```
2. number: 0
3. binary_relocation: true
4. script:
 - cmake -H. -Bbuild_conda -G "${CMAKE_GENERATOR}" -
5. DCMAKE_INSTALL_PREFIX=${PREFIX} # [not win]
 - cmake -H. -Bbuild_conda -G "%CMAKE_GENERATOR%" -
6. DCMAKE_INSTALL_PREFIX="%LIBRARY_PREFIX%" # [win]
7. - cmake --build build_conda --target install
```

将安装目录设置为  `${prefix}` (Conda的内部变量)，然后构建并安装项目。调用构建目录命名为  `build_conda` 的动机与前面的示例类似：特定的构建目录名可能已经命名为  `build` 。

## 更多信息

配置文件  `meta.yaml` 可为任何项目指定构建、测试和安装步骤。详情请参考官方文档：<https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html>

要将Conda包上传到Anaconda云，请遵循官方的Anaconda文档：  
<https://docs.anaconda.com/anaconda-cloud/user-guide/>

此外，也可以考虑将Miniconda，作为Anaconda的轻量级替代品：<https://conda.io/miniconda.html>

## 11.5 将Conda包作为依赖项发布给项目

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-11/recipe-05> 中找到。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

这个示例中，我们将基于之前示例的结果，并且为CMake项目准备一个更真实和复杂的Conda包，这将取决于DGEMM的函数实现，对于矩阵与矩阵的乘法，可以使用Intel的MKL库进行。Intel的MKL库可以以Conda包的形式提供。此示例将为我们提供一个工具集，用于准备和共享具有依赖关系的Conda包。

### 准备工作

对于这个示例，我们将使用与前一个示例中的Conda配置，和相同的文件命名和目录结构：

```

1. .
2. └── CMakeLists.txt
3. └── conda-recipe
4. └── meta.yaml
5. └── example.cpp

```

示例文件(`example.cpp`)将执行矩阵-矩阵乘法，并将MKL库返回的结果与“nody”实现进行比较：

```

1. #include "mkl.h"
2.
3. #include <cassert>
4. #include <cmath>
5. #include <iostream>
6. #include <random>
7.
8. int main() {
9. // generate a uniform distribution of real number between -1.0 and 1.0
10. std::random_device rd;
11. std::mt19937 mt(rd());
12. std::uniform_real_distribution< double > dist(-1.0, 1.0);
13.
14. int m = 500;
15. int k = 1000;
16. int n = 2000;
17.

```

```

18. double *A = (double *)mkl_malloc(m * k * sizeof(double), 64);
19. double *B = (double *)mkl_malloc(k * n * sizeof(double), 64);
20. double *C = (double *)mkl_malloc(m * n * sizeof(double), 64);
21. double * D = new double[m * n];
22.
23. for (int i = 0; i < (m * k); i++) {
24. A[i] = dist(mt);
25. }
26.
27. for (int i = 0; i < (k * n); i++) {
28. B[i] = dist(mt);
29. }
30.
31. for (int i = 0; i < (m * n); i++) {
32. C[i] = 0.0;
33. }
34.
35. double alpha = 1.0;
36. double beta = 0.0;
37. cblas_dgemm(CblasRowMajor,
38. CblasNoTrans,
39. CblasNoTrans,
40. m,
41. n,
42. k,
43. alpha,
44. A,
45. k,
46. B,
47. n,
48. beta,
49. C,
50. n);
51.
52. // D_mn = A_mk B_kn
53. for (int r = 0; r < m; r++) {
54. for (int c = 0; c < n; c++) {
55. D[r * n + c] = 0.0;
56. for (int i = 0; i < k; i++) {
57. D[r * n + c] += A[r * k + i] * B[i * n + c];
58. }
59. }
}

```

```

60. }
61.
62. // compare the two matrices
63. double r = 0.0;
64. for (int i = 0; i < (m * n); i++) {
65. r += std::pow(C[i] - D[i], 2.0);
66. }
67. assert (r < 1.0e-12 && "ERROR: matrices C and D do not match");
68.
69. mkl_free(A);
70. mkl_free(B);
71. mkl_free(C);
72. delete[] D;
73.
74. std::cout << "MKL DGEMM example worked!" << std::endl;
75.
76. return 0;
77. }
```

我们还需要修改 `meta.yaml`。然而，与上一个示例相比，唯一的变化是在依赖项中加入了 `mkl-devel`：

```

1. package:
2. name: conda-example-dgemm
3. version: "0.0.0"
4.
5. source:
6. path: ./ # this can be changed to git-url
7.
8. build:
9. number: 0
10. script:
11. - cmake -H. -Bbuild_conda -G "${CMAKE_GENERATOR}"
12. -DCMAKE_INSTALL_PREFIX=${PREFIX} # [not win]
13. - cmake -H. -Bbuild_conda -G "%CMAKE_GENERATOR%"
14. -DCMAKE_INSTALL_PREFIX="%LIBRARY_PREFIX%" # [win]
15. - cmake --build build_conda --target install
16.
17. requirements:
18. build:
19. - cmake >=3.5
20. - {{ compiler('cxx') }}
```

```

21. host:
22. - mkl - devel 2018
23.
24. about:
25. home: http://www.example.com
26. license: MIT
27. summary: "Summary in here ..."
```

## 具体实施

1. `CMakeLists.txt` 文件声明了最低版本、项目名称和支持的语言：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-05 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

2. 使用 `example.cpp` 构建 `dgemm-example` 可执行目标：

```

1. add_executable(dgemm-example "")
2. target_sources(dgemm-example
3. PRIVATE
4. example.cpp
5.)
```

3. 然后，需要找到通过 `MKL-devel` 安装的MKL库。我们准备了一个名为 `IntelMKL` 的 `INTERFACE` 库，该库可以用于其他目标，并将为依赖的目标设置包括目录、编译器选项和链接库。根据Intel的建议(<https://software.intel.com/en-us/articles/intel-mml-link-line-advisor/>)进行设置。首先，设置编译器选项：

```

1. add_library(IntelMKL INTERFACE)
2.
3. target_compile_options(IntelMKL
4. INTERFACE
5. ${$<$OR:$<CXX_COMPILER_ID:GNU>,$<CXX_COMPILER_ID:AppleClang>>:-m64}
6.)
```

4. 接下来，查找 `mkl.h` 头文件，并为 `IntelMKL` 目标设置 `include` 目录：

```

1. find_path(_mkl_h
2. NAMES
3. mkl.h
4. HINTS
5. ${CMAKE_INSTALL_PREFIX}/include
6.)
7.
8. target_include_directories(IntelMKL
9. INTERFACE
10. ${_mkl_h}
11.)
12.
13. message(STATUS "MKL header file FOUND: ${_mkl_h}")

```

## 5. 最后，为 `IntelMKL` 目标设置链接库：

```

1. find_library(_mkl_libs
2. NAMES
3. mkl_rt
4. HINTS
5. ${CMAKE_INSTALL_PREFIX}/lib
6.)
7. message(STATUS "MKL single dynamic library FOUND: ${_mkl_libs}")
8.
9. find_package(Threads QUIET)
10. target_link_libraries(IntelMKL
11. INTERFACE
12. ${_mkl_libs})
13. ${$<$<OR:$<CXX_COMPILER_ID:GNU>,$<CXX_COMPILER_ID:AppleClang>>:Threads::Threads}
14. ${$<$<OR:$<CXX_COMPILER_ID:GNU>,$<CXX_COMPILER_ID:AppleClang>>:m>}
15.)

```

## 6. 使用 `cmake_print_properties` 函数，打印 `IntelMKL` 目标的信息：

```

1. include(CMakePrintHelpers)
2. cmake_print_properties(
3. TARGETS
4. IntelMKL
5. PROPERTIES
6. INTERFACE_COMPILE_OPTIONS

```

```

7. INTERFACE_INCLUDE_DIRECTORIES
8. INTERFACE_LINK_LIBRARIES
9.)

```

7. 将这些库连接到 `dgemm-example` :

```

1. target_link_libraries(dgemm-example
2. PRIVATE
3. IntelMKL
4.)

```

8. `CMakeLists.txt` 中定义了安装目标:

```

1. install(
2. TARGETS
3. dgemm-example
4. DESTINATION
5. bin
6.)

```

9. 尝试构建包:

```
1. $ conda build conda-recipe
```

10. 过程中屏幕上将看到大量输出, 但是一旦构建完成, 就可以对包进行安装包。首先, 在本地进行安装测试:

```
1. $ conda install --use-local conda-example-dgemm
```

11. 现在测试安装, 打开一个新的终端(假设Anaconda处于激活状态), 并输入:

```

1. $ dgemm-example
2.
3. MKL DGEMM example worked!

```

12. 安装成功之后, 再进行卸载:

```
1. $ conda remove conda-example-dgemm
```

## 工作原理

`meta.yaml` 中的变化就是 `mml-devel` 依赖项。从CMake的角度来看，这里的挑战是定位Anaconda安装的MKL库。幸运的是，我们知道它位于  `${CMAKE_INSTALL_PREFIX}` 中。可以使用在线的 `Intel MKL link line advisor` (<https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/>) 查看如何根据选择的平台和编译器，将MKL链接到我们的项目中，我们会将此信息封装到 `INTERFACE` 库中。这个解决方案非常适合类MKL的情况：库不是由我们的项目或任何子项目创建的目标，但是它仍然需要以一种方式进行处理；也就是：设置编译器标志，包括目录和链接库。`INTERFACE` 库是构建系统中的目标，但不创建任何构建输出(至少不会直接创建)。但由于它们是目标，我们可对它们的属性进行设置。这样与“实际”目标一样，可以安装、导出和导入。

首先，我们用 `INTERFACE` 属性声明一个名为 `IntelMKL` 的新库。然后，根据需要设置属性，并使用 `INTERFACE` 属性在目标上调用适当的CMake命令：

- `target_compile_options`: 用于设置 `INTERFACE_COMPILE_OPTIONS`。示例中，设置了 `-m64`，不过这个标志只有GNU和AppleClang编译器能够识别。并且，我们使用生成器表达式来实现。
- `target_include_directories`: 用于设置 `INTERFACE_INCLUDE_DIRECTORIES`。使用 `find_path`，可以在找到系统上的 `mk1.h` 头文件后设置这些参数。
- `target_link_libraries`: 用于设置 `INTERFACE_LINK_LIBRARIES`。我们决定链接动态库 `libmkl_rt.so`，并用 `find_library` 搜索它。GNU或AppleClang编译器还需要将可执行文件链接到线程和数学库。同样，这些情况可以使用生成器表达式优雅地进行处理。

在 `IntelMKL` 目标上设置的属性后，可以通过 `cmake_print_properties` 命令将属性进行打印。最后，链接到 `IntelMKL` 目标，这将设置编译器标志，包括目录和链接库：

```
1. target_link_libraries(dgemm-example
2. PRIVATE
3. IntelMKL
4.)
```

## 更多信息

Anaconda云上包含大量包。使用上述方法，可以为CMake项目构建依赖于其他Conda包的Conda包。这样，就可以探索软件功能的各种可能性，并与他人分享您的软件包！

# 第12章 构建文档

---

本章的主要内容有：

- 使用Doxygen构建文档
- 使用Sphinx构建文档
- 结合Doxygen和Sphinx

文档在所有的软件项目都是有必要的：对于用户来说，了解如何获得并构建代码，并且如何有效地使用源代码或库；对于开发人员来说，文档可用来描述你源码细节，并帮助其他程序员参与其中为该项目作出贡献。本章将展示如何使用CMake构建代码文档，这里使用了两个流行的文档框架：Doxygen和Sphinx。

## 12.1 使用Doxygen构建文档

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-12/recipe-01> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Doxygen(<http://www.doxygen.nl>)是非常流行的源代码文档工具。可以在代码中添加文档标记作为注释，而后运行Doxygen提取这些注释，并以Doxyfile配置文件中定义的格式创建文档。Doxygen可以输出HTML、XML，甚至LaTeX或PDF。本示例将展示，如何使用CMake来构建Doxygen文档。

## 准备工作

使用前几章中介绍的消息库的简化版本。目录结构如下：

```

1. .
2. └── cmake
3. └── UseDoxygenDoc.cmake
4. └── CMakeLists.txt
5. └── docs
6. └── Doxyfile.in
7. └── front_page.md
8. └── src
9. └── CMakeLists.txt
10. └── hello-world.cpp
11. └── Message.cpp
12. └── Message.hpp

```

我们仍然在 `src` 子目录下放置源代码，并且在CMake子目录中有自定义的CMake模块。由于重点是文档，所以消除了对UUID的依赖，并简化了源代码。最大的区别是头文件中的大量代码注释：

```

1. #pragma once
2.
3. #include <iostream>
4. #include <string>
5.
6. /* ! \file Message.hpp */
7.
8. /*! \class Message
9. * \brief Forwards string to screen

```

```

10. * \author Roberto Di Remigio
11. * \date 2018
12. *
13.
14. class Message {
15. public:
16. /*! \brief Constructor from a string
17. * \param[in] m a message
18. */
19. Message(const std::string &m) : message_(m) {}
20. /*! \brief Constructor from a character array
21. * \param[in] m a message
22. */
23. Message(const char * m): message_(std::string(m)){}
24.
25. friend std::ostream &operator<<(std::ostream &os, Message &obj) {
26. return obj.printObject(os);
27. }
28. private:
29. /*! The message to be forwarded to screen */
30. std::string message_;
31. /*! \brief Function to forward message to screen
32. * \param[in, out] os output stream
33. */
34. std::ostream &printObject(std::ostream &os);
35. };

```

这些注释的格式是 `/*! */`，并包含一些Doxygen可以理解的特殊标记(参见 <http://www.stack.nl/~dimitri/Doxygen/manual/docblocks.html> )。

## 具体实施

首先，来看看根目录下的 `CMakeLists.txt`：

1. 我们声明了一个C++11项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2. project(recipe-01 LANGUAGES CXX)
3. set(CMAKE_CXX_STANDARD 11)
4. set(CMAKE_CXX_EXTENSIONS OFF)
5. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

## 2. 为动态库和静态库，以及可执行文件定义了输出目录：

```

1. include(GNUInstallDirs)
2. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
3. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
4. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
5. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
6. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
7. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

```

## 3. 将 `cmake` 子目录追加到 `CMAKE_MODULE_PATH`。这是需要CMake找到我们的自定义模块：

```
1. list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
```

## 4. `UseDoxygenDoc.cmake` 自定义模块。将在后面讨论它的内容：

```
1. include(UseDoxygenDoc)
```

## 5. 然后添加 `src` 子目录：

```
1. add_subdirectory(src)
```

`src` 子目录中的 `CMakeLists.txt` 文件包含以下构建块：

### 1. 添加了消息库：

```

1. add_library(message STATIC
2. Message.hpp
3. Message.cpp
4.)

```

### 2. 然后，声明 `add_doxxygen_doc` 函数。这个函数可以理解这些参数：BUILD\_DIR、DOXY\_FILE、TARGET\_NAME和COMMENT。使用 `cmake_parse_arguments` 标准CMake命令解析这些参数：

```

1. function(add_doxxygen_doc)
2. set(options)
3. set(oneValueArgs BUILD_DIR DOXY_FILE TARGET_NAME COMMENT)
4. set(multiValueArgs)
5.
6. cmake_parse_arguments(DOXY_DOC

```

```

7. "${options}"
8. "${oneValueArgs}"
9. "${multiValueArgs}"
10. ${ARGN}
11.)
12.
13. # ...
14. endfunction()

```

3. Doxyfile包含用于构建文档的所有Doxygen设置。一个模板 `Doxyfile.in` 文件作为函数参数 `DOXY_FILE` 传递，并解析为 `DOXY_DOC_DOXY_FILE` 变量。使用如下方式，配置模板文件 `Doxyfile.in`：

```

1. configure_file(
2. ${DOXY_DOC_DOXY_FILE}
3. ${DOXY_DOC_BUILD_DIR}/Doxyfile
4. @ONLY
5.)

```

4. 然后，定义了一个名为 `DOXY_DOC_TARGET_NAME` 的自定义目标，它将使用Doxyfile中的设置执行Doxygen，并在 `DOXY_DOC_BUILD_DIR` 中输出结果：

```

1. add_custom_target(${DOXY_DOC_TARGET_NAME}
2. COMMAND
3. ${DOXYGEN_EXECUTABLE} Doxyfile
4. WORKING_DIRECTORY
5. ${DOXY_DOC_BUILD_DIR}
6. COMMENT
7. "Building ${DOXY_DOC_COMMENT} with Doxygen"
8. VERBATIM
9.)

```

5. 最后，为用户打印一条状态信息：

```

message(STATUS "Added ${DOXY_DOC_TARGET_NAME} [Doxygen] target to build
1. documentation")

```

可以像往常一样配置项目：

```

1. $ mkdir -p build
2. $ cd build

```

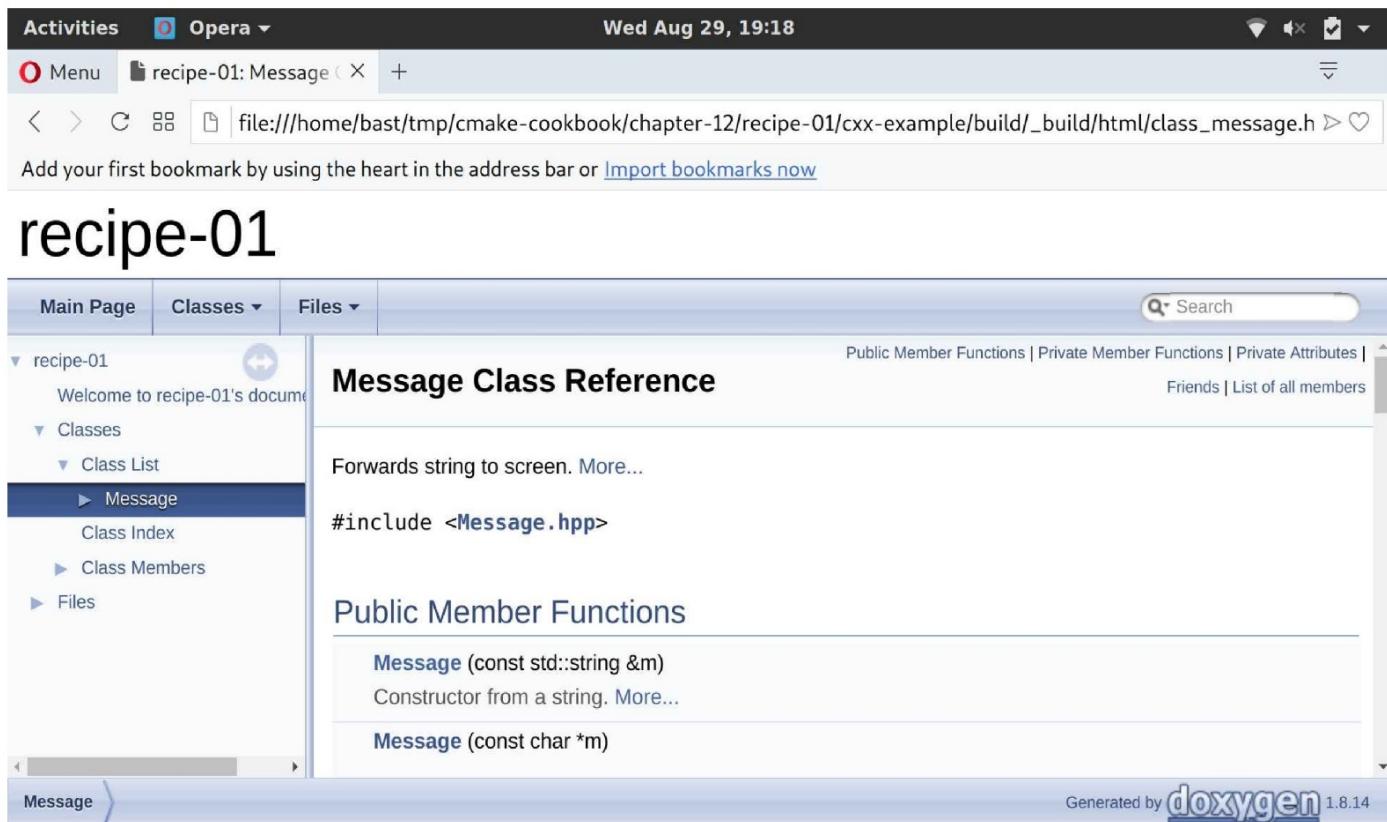
```
3. $ cmake ..
4. $ cmake --build .
```

可以通过调用自定义文档目标来构建文档：

```
1. $ cmake --build . --target docs
```

您将注意到构建树中出现了一个 `_build` 子目录。它包含Doxygen从源文件生成的HTML文档。用浏览器打开 `index.html` 将显示Doxygen欢迎页面。

如果导航到类列表，例如：可以浏览Message类的文档：



## 工作原理

默认情况下，CMake不支持文档构建。但是，我们可以使用 `add_custom_target` 执行任意操作。需要注意的是，需要确保构建文档所需的工具(本例中是Doxygen和Perl)在系统上可用。

此外，请注意 `UseDoxygenDoc.cmake` 自定义模块只做以下工作：

- 执行对Doxygen和Perl可执行程序的搜索
- 定义函数

使用 `add_doxygen_doc` 函数对文档目标进行创建。这个显式模式要优于隐式模式，我们也认为这是很好的实践方式：不要使用模块来执行类似宏(或函数)的操作。

为了限制变量定义的范围和可能出现的副作用，我们使用函数而不是宏实现了 `add_doxygen_doc`。在这种情况下，函数和宏都可以工作(并且会产生相同的结果)，但是建议优先使用函数而不是宏，除非需要修改父范围中的变量。

**NOTE:**在`cmake 3.9`中添加了 `FindDoxygen.cmake` 模块。实现了 `doxygen_add_docs` 函数，其行为与我们在本示例中给出的宏类似。要了解更多细节，请访问 <https://cmake.org/cmake/help/v3.9/module/FindDoxygen.html> 查看在线文档。

## 12.2 使用Sphinx构建文档

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-12/recipe-02> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

Sphinx是一个Python程序，也是一个非常流行的文档系统(<http://www.sphinx-doc.org>)。当与Python项目一起使用时，可以为 `docstring` 解析源文件，并自动为函数和类生成文档页面。然而，Sphinx不仅限于Python，还可以解析reStructuredText、Markdown，并生成HTML、ePUB或PDF文档。还有在线阅读服务(<https://readthedocs.org>)，它提供了一种快速编写和部署文档的方法。本示例将向您展示，如何使用CMake构建Sphinx文档。

## 准备工作

我们希望建立一个简单的网站，记录我们的消息库输出的信息。源码树现在看起来如下：

```

1. .
2. └── cmake
3. ├── FindSphinx.cmake
4. └── UseSphinxDoc.cmake
5. └── CMakeLists.txt
6. └── docs
7. ├── conf.py.in
8. └── index.rst
9. └── src
10. ├── CMakeLists.txt
11. ├── hello-world.cpp
12. ├── Message.cpp
13. └── Message.hpp

```

`cmake` 子目录中有一些自定义模块，`docs` 子目录以纯文本reStructuredText格式的网站主页，`index.rst` 和一个带有Sphinx的设置Python模板文件 `conf.py.in`，这个模板文件可以使用 `sphinx-quickstart` 程序自动生成。

## 具体实施

与之前的示例相比，我们将修改主 `CMakeLists.txt` 文件，并实现一个函数(`add_sphinx_doc`)：

1. 将 `cmake` 文件夹附加到 `CMAKE_MODULE_PATH` 之后，我们将包括 `UseSphinxDoc.cmake` 自

定义模块：

```
1. list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
2. include(UseSphinxDoc)
```

2. `UseSphinxDoc.cmake` 模块定义了 `add_sphinx_doc` 函数。我们使用关键字参数调用这个函数，以便对Sphinx文档的构建进行设置。自定义文档目标将称为 `docs`：

```
1. add_sphinx_doc(
2. SOURCE_DIR
3. ${CMAKE_CURRENT_SOURCE_DIR}/docs
4. BUILD_DIR
5. ${CMAKE_CURRENT_BINARY_DIR}/_build
6. CACHE_DIR
7. ${CMAKE_CURRENT_BINARY_DIR}/_doctrees
8. HTML_DIR
9. ${CMAKE_CURRENT_BINARY_DIR}/sphinx_html
10. CONF_FILE
11. ${CMAKE_CURRENT_SOURCE_DIR}/docs/conf.py.in
12. TARGET_NAME
13. docs
14. COMMENT
15. "HTML documentation"
16.)
```

`UseSphinxDoc.cmake` 模块遵循相同的显式方式，这样的使用方式要优于在前一个示例中的隐式方式：

1. 需要找到Python解释器和Sphinx可执行文件，如下：

```
1. find_package(PythonInterp REQUIRED)
2. find_package(Sphinx REQUIRED)
```

2. 然后，用一个值关键字参数定义 `add_sphinx_doc` 函数，并用 `cmake_parse_arguments` 解析：

```
1. function(add_sphinx_doc)
2. set(options)
3. set(oneValueArgs
4. SOURCE_DIR
5. BUILD_DIR
6. CACHE_DIR)
```

```

7. HTML_DIR
8. CONF_FILE
9. TARGET_NAME
10. COMMENT
11.)
12.
13. set(multiValueArgs)
14.
15. cmake_parse_arguments(SPHINX_DOC
16. "${options}"
17. "${oneValueArgs}"
18. "${multiValueArgs}"
19. ${ARGN}
20.)
21.
22. # ...
23.
24. endfunction()

```

3. 模板文件 `conf.py.in` 作为 `CONF_FILE` 关键字参数传递，在 `SPHINX_DOC_BUILD_DIR` 中配置为 `conf.py`：

```

1. configure_file(
2. ${SPHINX_DOC_CONF_FILE}
3. ${SPHINX_DOC_BUILD_DIR}/conf.py
4. @ONLY
5.)

```

4. 添加了一个名为 `SPHINX_DOC_TARGET_NAME` 的自定义目标，用Sphinx来编排文档构建：

```

1. add_custom_target(${SPHINX_DOC_TARGET_NAME}
2. COMMAND
3. ${SPHINX_EXECUTABLE}
4. -q
5. -b html
6. -c ${SPHINX_DOC_BUILD_DIR}
7. -d ${SPHINX_DOC_CACHE_DIR}
8. ${SPHINX_DOC_SOURCE_DIR}
9. ${SPHINX_DOC_HTML_DIR}
10. COMMENT
11. "Building ${SPHINX_DOC_COMMENT} with Sphinx"
12. VERBATIM

```

13. )

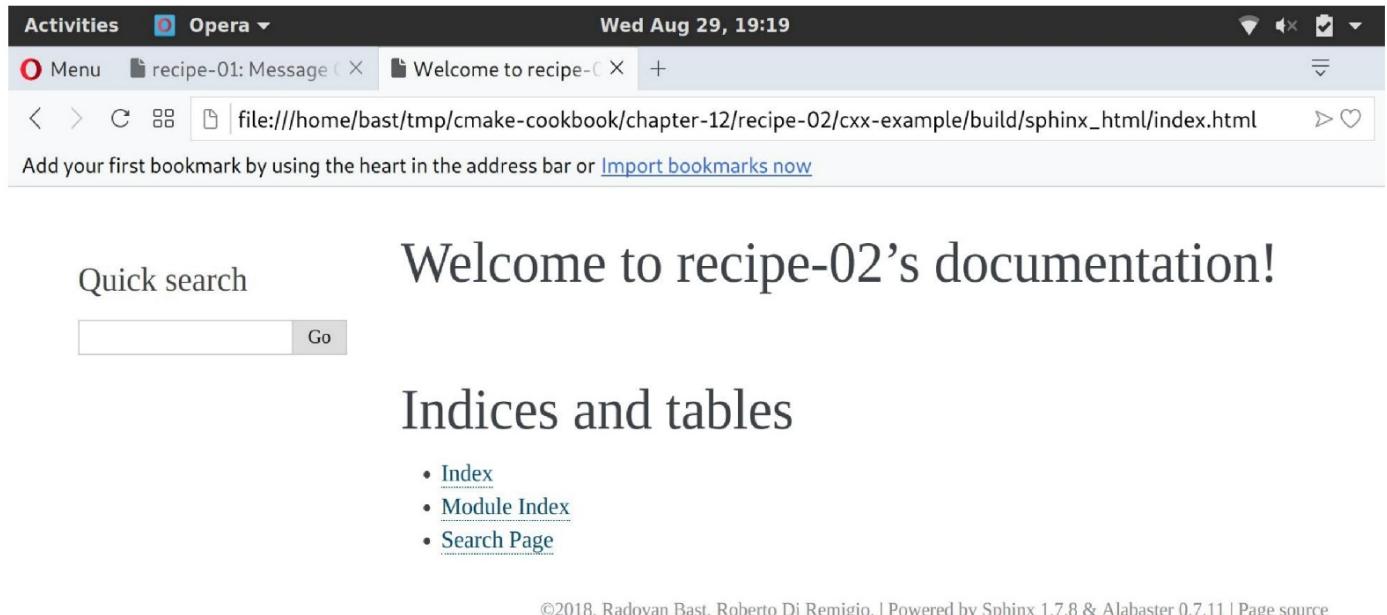
## 5. 最后，打印一条状态信息：

```
message(STATUS "Added ${SPHINX_DOC_TARGET_NAME} [Sphinx] target to build
1. documentation")
```

## 6. 配置项目并构建了文档目标：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build . --target docs
```

这将生成 `SPHINX_DOC_HTML_DIR` 中的HTML文档生成树的子目录。同样，可以使用浏览器打开 `index.html`，并查看文档：



## 工作原理

我们利用 `add_custom_target` 的功能，可以向构建系统添加任意的构建目标。本例中，文档将使用 Sphinx构建。由于Sphinx是一个可以与其他Python模块一起扩展的Python程序，所以 `docs` 目标将依赖于Python解释器。我们使用 `find_package` 确保依赖关系。需要注意的是，`FindSphinx.cmake` 模块不是一个标准的CMake模块；它的副本包含在项目源代码中，位于 `cmake` 子目录下。

## 12.3 结合Doxygen和Sphinx

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-12/recipe-03> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

我们有一个C++项目，因此Doxygen是生成源代码文档的理想选择。然而，我们也希望发布面向用户的文档，例如：介绍设计选择。所以我们想使用Sphinx，因为生成的HTML也可以在移动设备上查看，而且可以部署文档进行在线阅读(<https://readthedocs.org>)。本教程将演示如何使用Breathe插件(<https://breathe.readthedocs.io>)组合Doxygen和Sphinx。

### 准备工作

这个示例的目录结构，类似于之前的两个示例：

```

1. .
2. └── cmake
3. ├── FindPythonModule.cmake
4. ├── FindSphinx.cmake
5. └── UseBreathe.cmake
6. └── CMakeLists.txt
7. └── docs
8. ├── code-reference
9. │ ├── classes-and-functions.rst
10. │ └── message.rst
11. ├── conf.py.in
12. ├── Doxyfile.in
13. └── index.rst
14. └── src
15. ├── CMakeLists.txt
16. ├── hello-world.cpp
17. ├── Message.cpp
18. └── Message.hpp

```

`docs` 子目录现在同时包含一个 `Doxyfile.in` 和一个 `conf.py.in` 模板文件。模板文件中，分别设置了Doxygen和Sphinx。此外，还有一个 `code-referenc` 子目录。

`code-referenc` 子目录中的文件包含Breathe指令，用来在Sphinx中包含doxygen生成的文档：

```

1. Messaging classes

```

```

2. =====
3. Message
4. -----
5. .. doxygenclass:: Message
6. :project: recipe-03
7. :members:
8. :protected-members:
9. :private-members:

```

这将输出Message类的文档。

## 具体实施

`src` 目录中的 `CMakeLists.txt` 文件没有改变。主 `CMakeLists.txt` 文件中有修改：

1. 包含 `UseBreathe.cmake` 自定义模块：

```

1. list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
2. include(UseBreathe)

```

2. 调用 `add_breathe_doc` 函数，这个函数是在自定义模块中定义的，它接受关键字参数，来设置 Doxygen 和 Sphinx：

```

1. add_breathe_doc(
2. SOURCE_DIR
3. ${CMAKE_CURRENT_SOURCE_DIR}/docs
4. BUILD_DIR
5. ${CMAKE_CURRENT_BINARY_DIR}/_build
6. CACHE_DIR
7. ${CMAKE_CURRENT_BINARY_DIR}/_doctrees
8. HTML_DIR
9. ${CMAKE_CURRENT_BINARY_DIR}/html
10. DOXY_FILE
11. ${CMAKE_CURRENT_SOURCE_DIR}/docs/Doxyfile.in
12. CONF_FILE
13. ${CMAKE_CURRENT_SOURCE_DIR}/docs/conf.py.in
14. TARGET_NAME
15. docs
16. COMMENT
17. "HTML documentation"
18.)

```

让我们看一下 `UseBreatheDoc.cmake` 模块，其遵循了与我们在前两个示例中描述的显式模式。具体描述如下：

1. 文档生成依赖于Doxygen：

```
1. find_package(Doxygen REQUIRED)
2. find_package(Perl REQUIRED)
```

2. 还依赖于Python解释器和Sphinx：

```
1. find_package(PythonInterp REQUIRED)
2. find_package(Sphinx REQUIRED)
```

3. 此外，还必须找到breathe的Python模块。这里，我们使用 `FindPythonModule.cmake` 模块：

```
1. include(FindPythonModule)
2. find_python_module(breathe REQUIRED)
```

4. 定义了 `add_breathe_doc` 函数，这个函数有一个单值关键字参数，我们将使用 `cmake_parse_arguments` 命令解析它：

```
1. function(add_breathe_doc)
2. set(options)
3. set(oneValueArgs
4. SOURCE_DIR
5. BUILD_DIR
6. CACHE_DIR
7. HTML_DIR
8. DOXY_FILE
9. CONF_FILE
10. TARGET_NAME
11. COMMENT
12.)
13. set(multiValueArgs)
14.
15. cmake_parse_arguments(BREATHE_DOC
16. "${options}"
17. "${oneValueArgs}"
18. "${multiValueArgs}"
19. ${ARGN}
20.)
21.
```

```

22. # ...
23.
24. endfunction()

```

5. `BREATHE_DOC_CONF_FILE` 中的Sphinx模板文件，会通过 `conf.py` 配置到的 `BREATHE_DOC_BUILD_DIR` 目录下：

```

1. configure_file(
2. ${BREATHE_DOC_CONF_FILE}
3. ${BREATHE_DOC_BUILD_DIR}/conf.py
4. @ONLY
5.)

```

6. 相应地，Doxygen的 `BREATHE_DOC_DOXY_FILE` 模板文件配置为 `BREATHE_DOC_BUILD_DIR` 中的Doxyfile：

```

1. configure_file(
2. ${BREATHE_DOC_DOXY_FILE}
3. ${BREATHE_DOC_BUILD_DIR}/Doxyfile
4. @ONLY
5.)

```

7. 添加 `BREATHE_DOC_TARGET_NAME` 自定义目标。注意，只有Sphinx在运行时，对Doxygen的调用才发生在 `BREATHE_DOC_SPHINX_FILE` 中：

```

1. add_custom_target(${BREATHE_DOC_TARGET_NAME}
2. COMMAND
3. ${SPHINX_EXECUTABLE}
4. -q
5. -b html
6. -c ${BREATHE_DOC_BUILD_DIR}
7. -d ${BREATHE_DOC_CACHE_DIR}
8. ${BREATHE_DOC_SOURCE_DIR}
9. ${BREATHE_DOC_HTML_DIR}
10. COMMENT
11. "Building ${BREATHE_DOC_TARGET_NAME} documentation with Breathe,
12. Sphinx and Doxygen"
13. VERBATIM
14.)

```

8. 最后，打印一条状态信息：

```
message(STATUS "Added ${BREATHE_DOC_TARGET_NAME} [Breathe+Sphinx+Doxygen]
1. target to build documentation")
```

## 9. 配置完成后，构建文档：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build . --target docs
```

该文档将在 `BREATHE_DOC_HTML_DIR` 子目录中可用。启动浏览器打开 `index.html` 文件后，可以导航到Message类的文档：

The screenshot shows a browser window with the address bar containing `file:///home/bast/tmp/cmake-cookbook/chapter-12/recipe-03/cxx-example/build/html/code-reference/messages`. The main content area displays the generated documentation for the `Message` class under the `Messaging classes` heading. The `Message` class is described as forwarding strings to the screen. It includes information about the author (Roberto Di Remigio) and the date (2018). A section for public functions lists the `Message` constructor, which takes a string and constructs a new `Message` object.

## 工作原理

尽管在声明定制的 `BREATHE_DOC_TARGET_NAME` 目标时只调用了Sphinx，但这里Doxygen和Sphinx都在运行。这要感谢Sphinx的 `conf.py` 文件中的以下设置：

```
1. def run_doxygen(folder):
2. """Run the doxygen make command in the designated folder"""
3.
4. try:
```

```
5. retcode = subprocess.call("cd {}; doxygen".format(folder), shell=True)
6. if retcode < 0:
7. sys.stderr.write(
8. "doxygen terminated by signal {}".format(-retcode))
9. except OSError as e:
10. sys.stderr.write("doxygen execution failed: {}".format(e))
11.
12.
13. def setup(app):
14. run_doxygen('@BREATHE_DOC_BUILD_DIR@')
```

Doxygen将生成XML输出，Breathe插件将能够与所选择的Sphinx文档样式一致的形式，呈现XML输出。

# 第13章 选择生成器和交叉编译

本章主要内容有：

- 使用Visual Studio 2017构建CMake项目
- 交叉编译hello world示例
- 使用OpenMP并行化交叉编译Windows二进制文件

CMake本身不构建可执行程序和库。不过，CMake配置一个项目，并生成构建工具或框架用于构建项目的文件。在GNU/Linux和macOS上，CMake通常生成Unix Makefile(也存在替代方式)。在Windows上，通常生成Visual Studio项目文件或MinGW或MSYS Makefile。CMake包括本地构建工具或集成开发环境(IDE)的生成器。可以通过以下链接阅读更多关于它们的信息：<https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html>

可以使用 `cmake -G` 的方式来选择生成器：

```
1. $ cmake -G "Visual Studio 15 2017"
```

不是每个平台上所有的生成器都可用，而且CMake在运行时获取平台信息。要查看当前平台上所有可用生成器的列表，请键入以下命令：

```
1. $ cmake -G
```

本章中，我们不会使用所有生成器，但是本书中的大多数示例都使用了Unix Makefile、MSYS Makefile、Ninja和Visual Studio 15 2017进行了测试。

我们将重点讨论Windows平台上的开发，将演示不使用命令行，如何使用Visual Studio 15 2017直接构建CMake项目。还会讨论如何在Linux或macOS系统上，交叉编译Windows的可执行文件。

## 13.1 使用CMake构建Visual Studio 2017项目

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-13/recipe-01> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

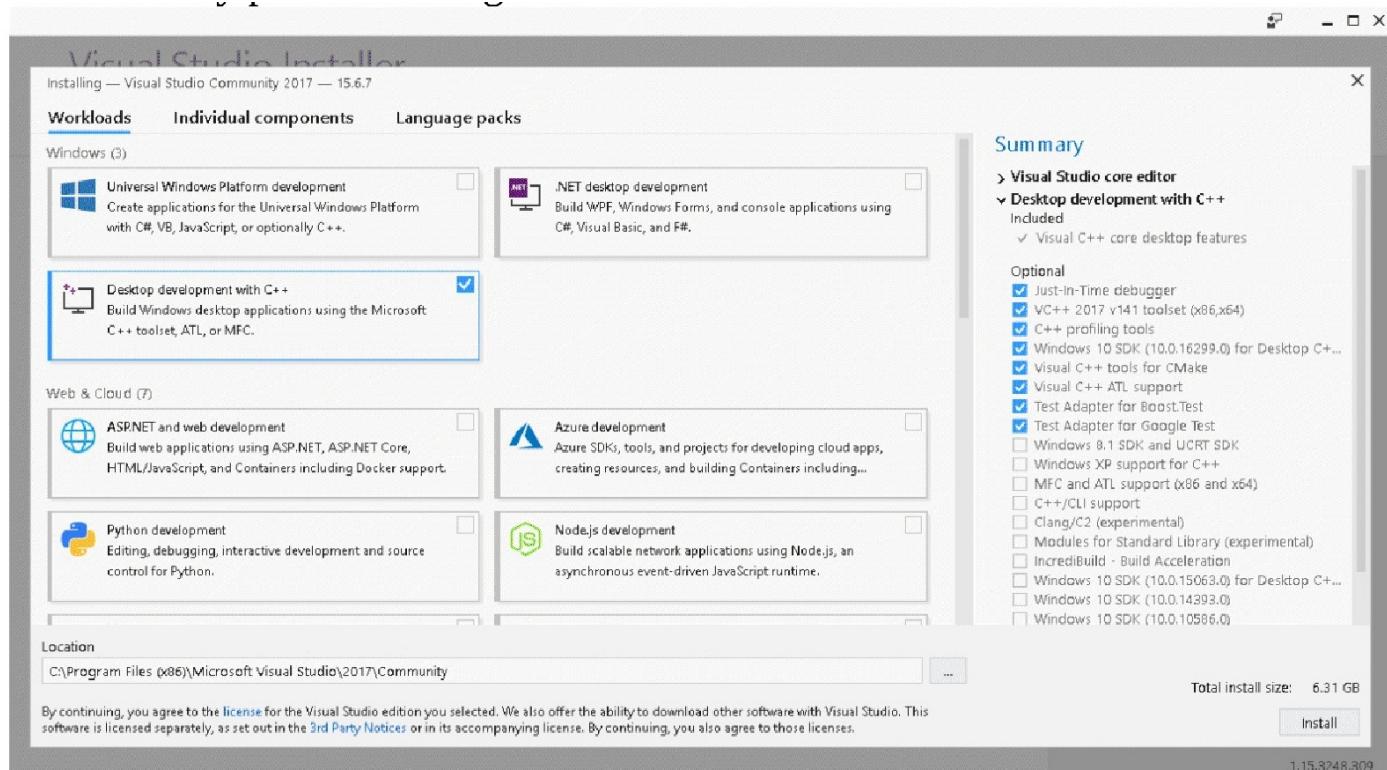
早期版本的Visual Studio要求开发人员在不同的Windows版本中编辑源代码并运行CMake命令，但Visual Studio 2017引入了对CMake项目的内置支持(<https://aka.ms/cmake>)，它允许在同一个IDE中发生整个编码、配置、构建和测试工作流。本示例中，不需要使用命令行，我们将直接使用Visual Studio 2017构建一个简单的“hello world”CMake示例项目。

### 准备工作

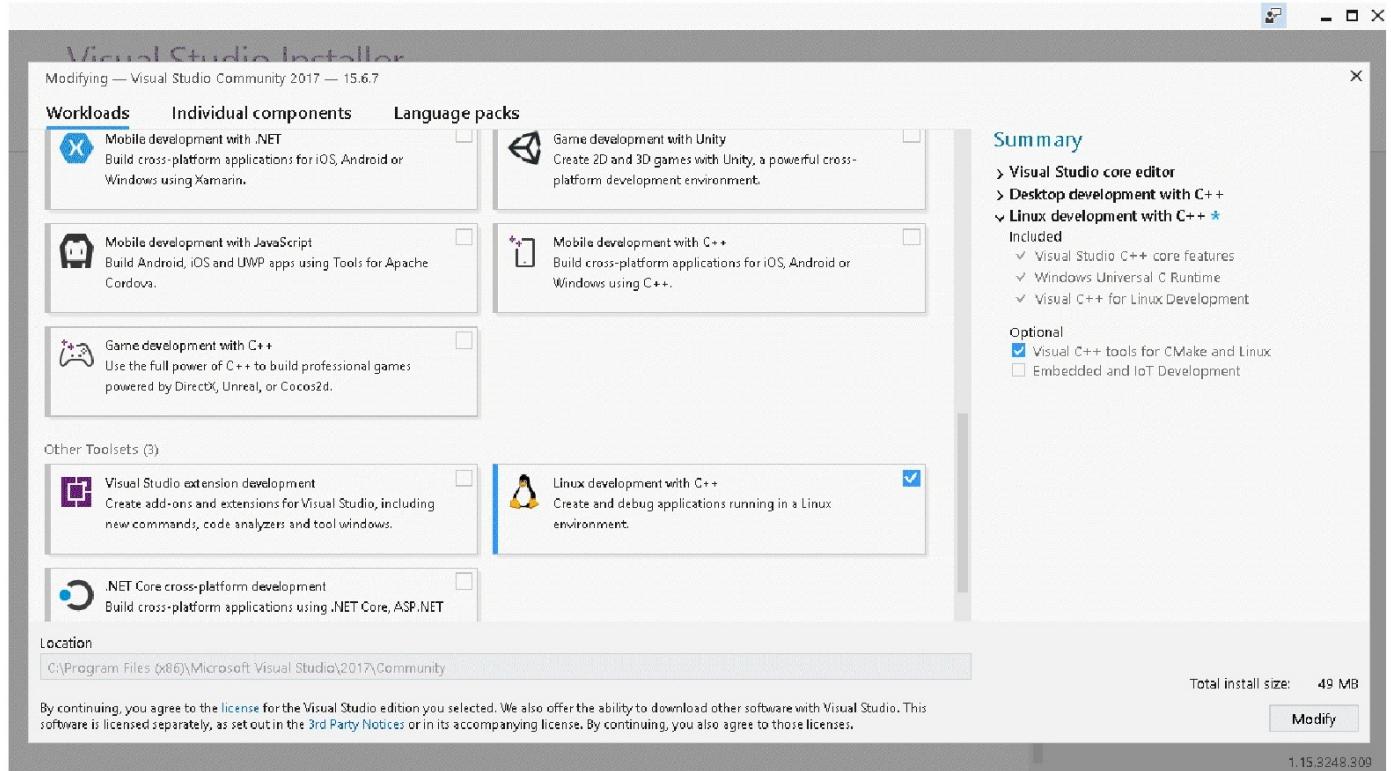
首先，下载并安装Visual Studio Community 2017

(<https://www.visualstudio.com/downloads/>)。在撰写本文时，这个版本是免费的，有30天的试用期。我们将遵循的视频中的步骤：[https://www.youtube.com/watch?v=\\_1KxJjV8r3Y](https://www.youtube.com/watch?v=_1KxJjV8r3Y)

运行安装程序时，在左侧面板上选择 **Desktop development with C++**，并在右侧的Summary面板上选择用于CMake的Visual C++工具：



Visual Studio 2017 15.4中，还可以为Linux平台进行构建。为此，在工具集中选择 **Linux development with C++**：



选择后，只要配置Linux服务器的访问权限，就可以从Visual Studio中同时对Windows和Linux机器进行构建。我们不在本章中演示这种方法。

这个示例中，我们将在Windows上构建一个二进制文件，我们的目标是配置和构建以下示例代码  
( `hello-world.cpp` )：

```

1. #include <cstdlib>
2. #include <iostream>
3. #include <string>
4. const std::string cmake_system_name = SYSTEM_NAME;
5. int main() {
6. std::cout << "Hello from " << cmake_system_name << std::endl;
7.
8. return EXIT_SUCCESS;
9. }
```

## 具体实施

创建相应的源码：

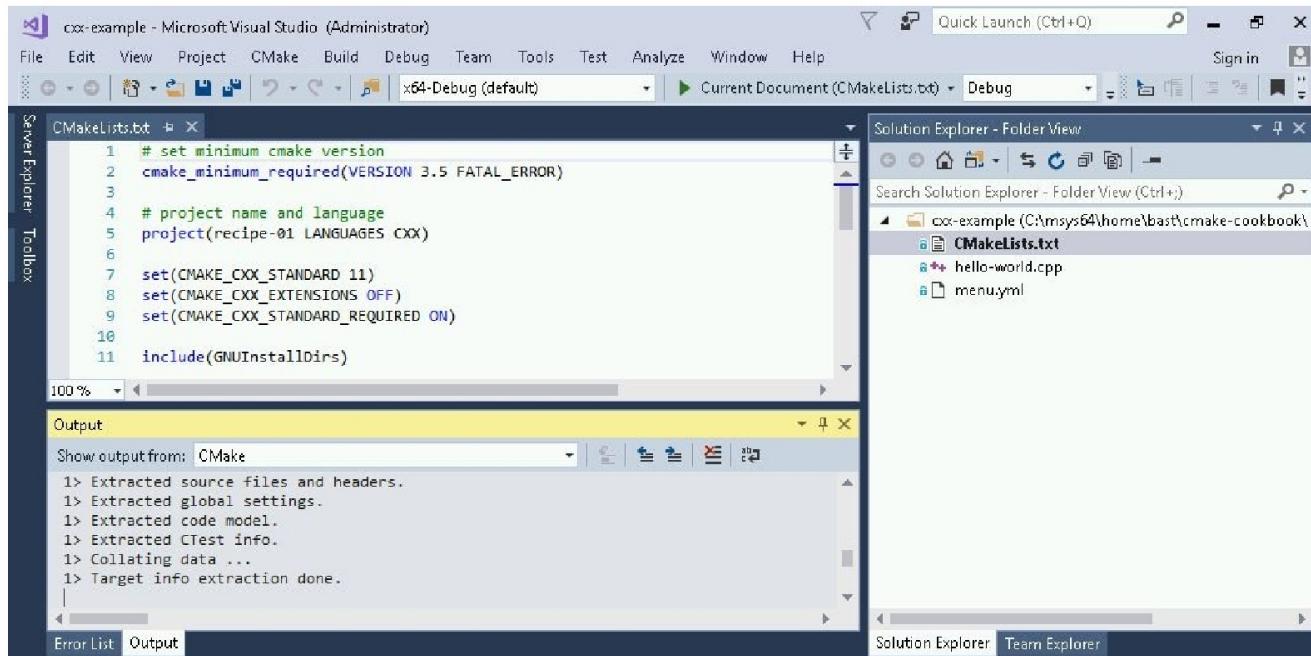
1. 创建一个目录，并将 `hello-world.cpp` 放在新目录中。
2. 目录中，创建一个 `CMakeLists.txt` 文件，其内容为：

```

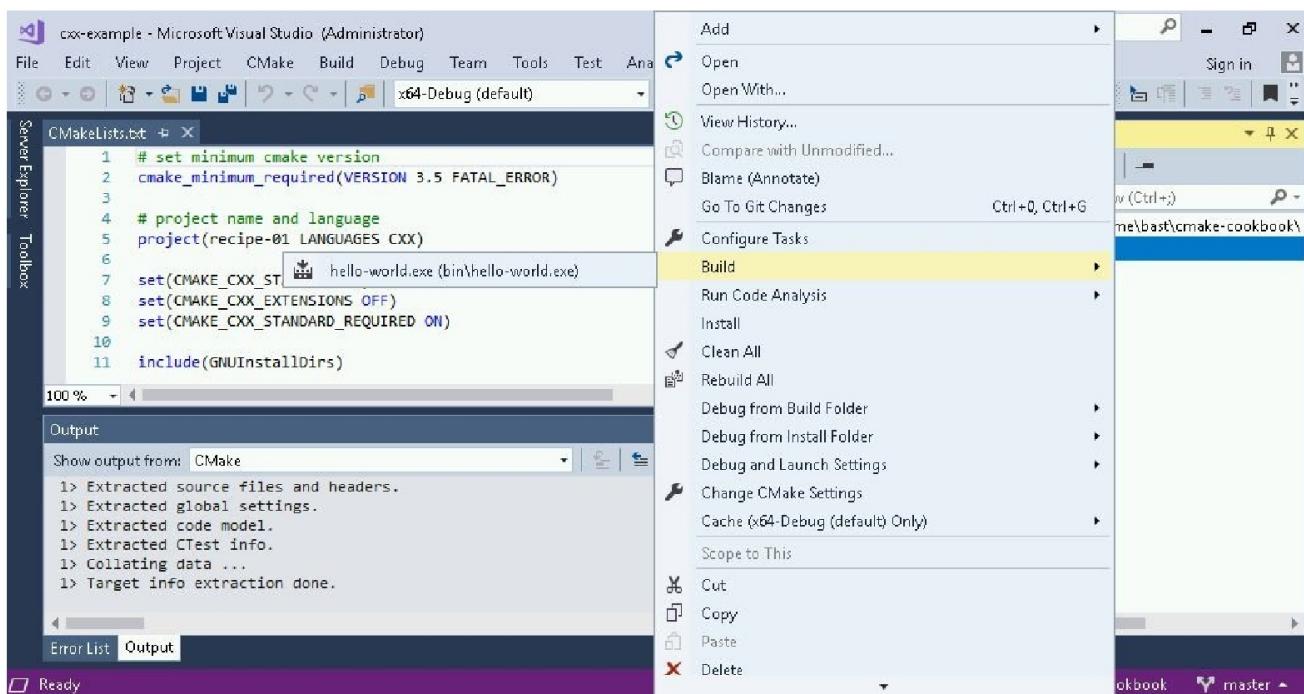
1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-01 LANGUAGES CXX)
6.
7. set(CMAKE_CXX_STANDARD 11)
8. set(CMAKE_CXX_EXTENSIONS OFF)
9. set(CMAKE_CXX_STANDARD_REQUIRED ON)
10.
11. include(GNUInstallDirs)
12. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
13. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
14. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
16. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
17. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
18.
19. # define executable and its source file
20. add_executable(hello-world hello-world.cpp)
21.
22. # we will print the system name in the code
23. target_compile_definitions(hello-world
24. PUBLIC
25. "SYSTEM_NAME=\"${CMAKE_SYSTEM_NAME}\""
26.)
27.
28. install(
29. TARGETS
30. hello-world
31. DESTINATION
32. ${CMAKE_INSTALL_BINDIR}
33.)

```

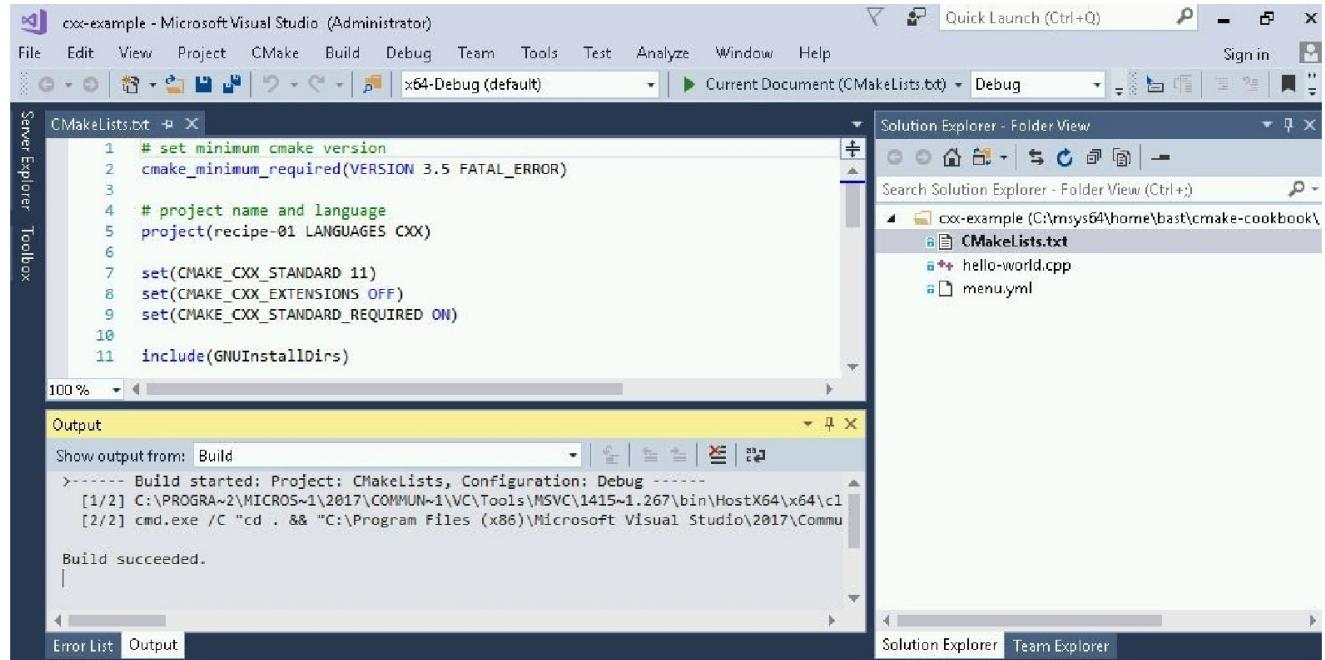
3. 打开Visual Studio 2017，然后通过下面的 `File ->Open -> Folder`，选择到新创建的包含源文件和 `CMakeLists.txt` 的文件夹下。
4. 打开文件夹后，请注意CMake配置步骤是如何运行的(面板底部)：



5. 现在，可以右键单击 **CMakeLists.txt** (右面板)，并选择 **Build** :



6. 构建项目(参见底部面板上的输出)：

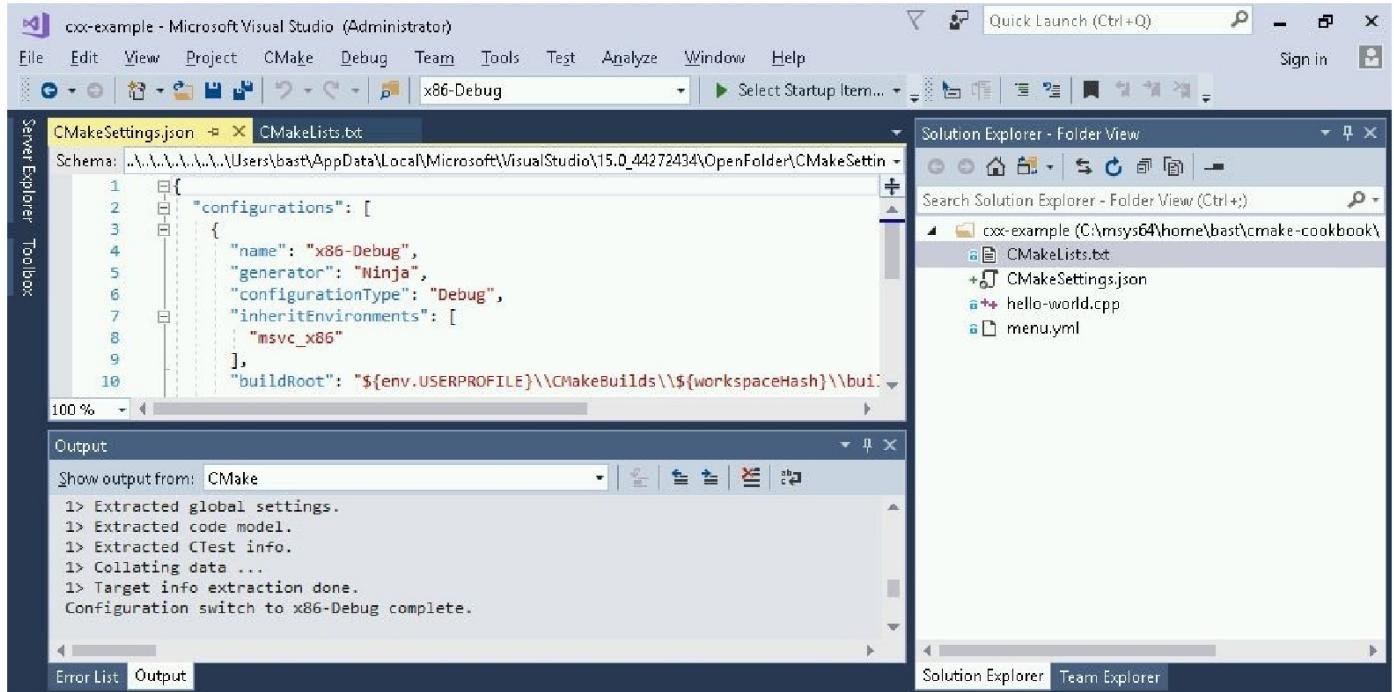


- 这就成功地编译了可执行文件。下一小节中，我们将学习如何定位可执行文件，并更改构建和安装路径。

## 工作原理

我们已经看到Visual Studio 2017能很好地对接CMake，并且已经能够在IDE中配置和构建代码。除了构建步骤之外，还可以运行安装或测试步骤。可以通过右键单击 `CMakeLists.txt`（右面板），访问这些文件。

然而，配置步骤是自动运行的，我们可能更想去修改配置选项。我们还想知道实际的构建和安装路径，以便测试我们的可执行文件。为此，我们可以选择 `CMake -> Change CMake Settings`，如下图所示：



面板左上角，可以检查和修改生成器(本例中是Ninja)、设置、参数以及路径。构建路径在前面的显示中可以看到。这些设置被分组为构建类型(`x86-Debug`、`x86-Release`等等)，我们可以在面板栏顶部的中间部分，通过选择切换构建类型。

现在知道了构建路径，可以测试编译后的可执行文件：

1. `$ ./hello-world.exe`
- 2.
3. `Hello from Windows`

当然，构建和安装路径可以进行修改。

## 更多信息

- Visual Studio支持CMake: <https://aka.ms/cmake>
- 使用CMake，基于Visual C++开发Linux应用：<https://blogs.msdn.microsoft.com/vcblog/2017/08/25/visual-c-for-linux-development-with-cmake/>
- Visual Studio官方文档：<https://www.visualstudio.com/vs/features/ide/>

## 13.2 交叉编译hello world示例

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-13/recipe-01> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

这个示例中，我们将重用“Hello World”示例，并将代码从Linux或macOS交叉编译到Windows。换句话说，我们将在Linux或macOS上配置和编译代码，并生成Windows平台的可执行文件

### 准备工作

我们从 `hello world` 示例(`hello-world.cpp`)开始：

```

1. #include <iostream>
2. #include <omp.h>
3. #include <string>
4.
5. int main(int argc, char *argv[])
6. {
7. std::cout << "number of available processors: " << omp_get_num_procs()
8. << std::endl;
9. std::cout << "number of threads: " << omp_get_max_threads() << std::endl;
10.
11. auto n = std::stol(argv[1]);
12. std::cout << "we will form sum of numbers from 1 to " << n << std::endl;
13.
14. // start timer
15. auto t0 = omp_get_wtime();
16.
17. auto s = 0LL;
18. #pragma omp parallel for reduction(+: s)
19. for (auto i = 1; i <= n; i++)
20. {
21. s += i;
22. }
23. // stop timer
24. auto t1 = omp_get_wtime();
25.
26. std::cout << "sum: " << s << std::endl;
```

```

 std::cout << "elapsed wall clock time: " << t1 - t0 << " seconds" <<
27. std::endl;
28.
29. return 0;
30. }
```

我们还将使用与前一个示例相同的 `CMakeLists.txt` :

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-01 LANGUAGES CXX)
6.
7. set(CMAKE_CXX_STANDARD 11)
8. set(CMAKE_CXX_EXTENSIONS OFF)
9. set(CMAKE_CXX_STANDARD_REQUIRED ON)
10.
11. include(GNUInstallDirs)
12. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
13. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
14. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
16. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
17. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
18.
19. # define executable and its source file
20. add_executable(hello-world hello-world.cpp)
21.
22. # we will print the system name in the code
23. target_compile_definitions(hello-world
24. PUBLIC
25. "SYSTEM_NAME=\"${CMAKE_SYSTEM_NAME}\""
26.)
27.
28. install(
29. TARGETS
30. hello-world
31. DESTINATION
32. ${CMAKE_INSTALL_BINDIR}
33.)
```

为了交叉编译源代码，我们需要安装一个C++交叉编译器，也可以为C和Fortran安装一个交叉编译器。可以使用打包的MinGW编译器，作为打包的交叉编译器的替代方案。还可以使用MXE (M cross environment)从源代码构建一套交叉编译器：<http://mxe.cc>

## 具体实施

我们将按照以下步骤，在这个交叉编译的“hello world”示例中创建三个文件：

1. 创建一个文件夹，其中包括 `hello-world.cpp` 和 `CMakeLists.txt`。
2. 再创建一个 `toolchain.cmake` 文件，其内容为：

```

1. # the name of the target operating system
2. set(CMAKE_SYSTEM_NAME Windows)
3.
4. # which compilers to use
5. set(CMAKE_CXX_COMPILER i686-w64-mingw32-g++)
6.
7. # adjust the default behaviour of the find commands:
8. # search headers and libraries in the target environment
9. set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
10. set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
11.
12. # search programs in the host environment
13. set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)

```

3. 将 `CMAKE_CXX_COMPILER` 设置为对应的编译器(路径)。
4. 然后，通过将 `CMAKE_TOOLCHAIN_FILE` 指向工具链文件，从而配置代码(本例中，使用了从源代码构建的MXE编译器)：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake -D CMAKE_TOOLCHAIN_FILE=toolchain.cmake ..
4.
5. -- The CXX compiler identification is GNU 5.4.0
 -- Check for working CXX compiler: /home/user/mxe/usr/bin/i686-w64-
6. mingw32.static-g++
 -- Check for working CXX compiler: /home/user/mxe/usr/bin/i686-w64-
7. mingw32.static-g++ -- works
8. -- Detecting CXX compiler ABI info
9. -- Detecting CXX compiler ABI info - done

```

```

10. -- Detecting CXX compile features
11. -- Detecting CXX compile features - done
12. -- Configuring done
13. -- Generating done
 -- Build files have been written to: /home/user/cmake-recipes/chapter-
14. 13/recipe-01/cxx-example/build

```

## 5. 现在，构建可执行文件：

```

1. $ cmake --build .
2.
3. Scanning dependencies of target hello-world
4. [50%] Building CXX object CMakeFiles/hello-world.dir/hello-world.cpp.obj
5. [100%] Linking CXX executable bin/hello-world.exe
6. [100%] Built target hello-world

```

## 6. 注意，我们已经在Linux上获得 `hello-world.exe`。将二进制文件复制到Windows上。

## 7. 在Windows上可以看到如下的输出：

```
1. Hello from Windows
```

## 8. 如你所见，这个二进制可以在Windows下工作。

# 工作原理

由于与目标环境(Windows)不同的主机环境(在本例中是GNU/Linux或macOS)上配置和构建代码，所以我们需要向CMake提供关于目标环境的信息，这些信息记录在 `toolchain.cmake` 文件中(<https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html#cross-compiling>)。

首先，提供目标操作系统的名称：

```
1. set(CMAKE_SYSTEM_NAME Windows)
```

然后，指定编译器：

```

1. set(CMAKE_C_COMPILER i686-w64-mingw32-gcc)
2. set(CMAKE_CXX_COMPILER i686-w64-mingw32-g++)
3. set(CMAKE_Fortran_COMPILER i686-w64-mingw32-gfortran)

```

这个例子中，我们不需要检测任何库或头文件。如果必要的话，我们将使用以下命令指定根路径：

```
1. set(CMAKE_FIND_ROOT_PATH /path/to/target/environment)
```

例如，提供MXE编译器的安装路径。

最后，调整 `find` 命令的默认行为。我们指示CMake在目标环境中查找头文件和库文件：

```
1. set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
2. set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
```

在主机环境中的搜索程序：

```
1. set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
```

## 更多信息

有关各种选项的更详细讨论，请参见：

<https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html#cross-compiling>

## 13.3 使用OpenMP并行化交叉编译Windows二进制文件

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-13/recipe-02> 中找到，其中包含一个C++示例和Fortran示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

在这个示例中，我们将交叉编译一个OpenMP并行化的Windows二进制文件。

### 准备工作

我们将使用第3章第5节中的未修改的源代码，示例代码将所有自然数加到N（`example.cpp`）：

```

1. #include <iostream>
2. #include <omp.h>
3. #include <string>
4.
5. int main(int argc, char *argv[]) {
6. std::cout << "number of available processors: " << omp_get_num_procs()
7. << std::endl;
8. std::cout << "number of threads: " << omp_get_max_threads() << std::endl;
9.
10. auto n = std::stol(argv[1]);
11. std::cout << "we will form sum of numbers from 1 to " << n << std::endl;
12.
13. // start timer
14. auto t0 = omp_get_wtime();
15.
16. auto s = 0LL;
17. #pragma omp parallel for reduction(+: s)
18. for (auto i = 1; i <= n; i++) {
19. s += i;
20. }
21.
22. // stop timer
23. auto t1 = omp_get_wtime();
24.
25. std::cout << "sum: " << s << std::endl;
26. std::cout << "elapsed wall clock time: " << t1 - t0 << " seconds" <<
27. std::endl;

```

```

27.
28. return 0;
29. }
```

**CMakeLists.txt** 检测OpenMP并行环境方面基本没有变化，除了有一个额外的安装目标：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.9 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-02 LANGUAGES CXX)
6.
7. set(CMAKE_CXX_STANDARD 11)
8. set(CMAKE_CXX_EXTENSIONS OFF)
9. set(CMAKE_CXX_STANDARD_REQUIRED ON)
10.
11. include(GNUInstallDirs)
12. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
13. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
14. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
15. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
16. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
17. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})
18.
19. find_package(OpenMP REQUIRED)
20.
21. add_executable(example example.cpp)
22.
23. target_link_libraries(example
24. PUBLIC
25. OpenMP::OpenMP_CXX
26.)
27.
28. install(
29. TARGETS
30. example
31. DESTINATION
32. ${CMAKE_INSTALL_BINDIR}
33.)
```

## 具体实施

通过以下步骤，我们将设法交叉编译一个OpenMP并行化的Windows可执行文件：

1. 创建一个包含 `example.cpp` 和 `CMakeLists.txt` 的目录。
2. 我们将使用与之前例子相同的 `toolchain.cmake`：

```

1. # the name of the target operating system
2. set(CMAKE_SYSTEM_NAME Windows)
3.
4. # which compilers to use
5. set(CMAKE_CXX_COMPILER i686-w64-mingw32-g++)
6.
7. # adjust the default behaviour of the find commands:
8. # search headers and libraries in the target environment
9. set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
10. set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
11. # search programs in the host environment
12. set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)

```

3. 将 `CMAKE_CXX_COMPILER` 设置为对应的编译器(路径)。
4. 然后，通过 `CMAKE_TOOLCHAIN_FILE` 指向工具链文件来配置代码(本例中，使用了从源代码构建的MXE编译器)：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake -D CMAKE_TOOLCHAIN_FILE=toolchain.cmake ..
4.
5. -- The CXX compiler identification is GNU 5.4.0
 -- Check for working CXX compiler: /home/user/mxe/usr/bin/i686-w64-
6. mingw32.static-g++
 -- Check for working CXX compiler: /home/user/mxe/usr/bin/i686-w64-
7. mingw32.static-g++ -- works
8. -- Detecting CXX compiler ABI info
9. -- Detecting CXX compiler ABI info - done
10. -- Detecting CXX compile features
11. -- Detecting CXX compile features - done
12. -- Found OpenMP_CXX: -fopenmp (found version "4.0")
13. -- Found OpenMP: TRUE (found version "4.0")
14. -- Configuring done
15. -- Generating done
 -- Build files have been written to: /home/user/cmake-recipes/chapter-
16. 13/recipe-02/cxx-example/build

```

## 5. 构建可执行文件:

```

1. $ cmake --build .
2.
3. Scanning dependencies of target example
4. [50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
5. [100%] Linking CXX executable bin/example.exe
6. [100%] Built target example

```

6. 将 `example.exe` 拷贝到Windows环境下。

7. Windows环境下，将看到如下的输出：

```

1. $ set OMP_NUM_THREADS=1
2. $ example.exe 1000000000
3.
4. number of available processors: 2
5. number of threads: 1
6. we will form sum of numbers from 1 to 1000000000
7. sum: 500000000500000000
8. elapsed wall clock time: 2.641 seconds
9.
10. $ set OMP_NUM_THREADS=2
11. $ example.exe 1000000000
12.
13. number of available processors: 2
14. number of threads: 2
15. we will form sum of numbers from 1 to 1000000000
16. sum: 500000000500000000
17. elapsed wall clock time: 1.328 seconds

```

8. 正如我们所看到的，二进制文件可以在Windows上工作，而且由于OpenMP并行化，我们可以观察到加速效果！

## 工作原理

我们已经成功地使用一个简单的工具链进行交叉编译了一个可执行文件，并可以在Windows平台上并行执行。我们可以通过设置 `OMP_NUM_THREADS` 来指定OpenMP线程的数量。从一个线程到两个线程，我们观察到运行时从2.6秒减少到1.3秒。有关工具链文件的讨论，请参阅前面的示例。

## 更多信息

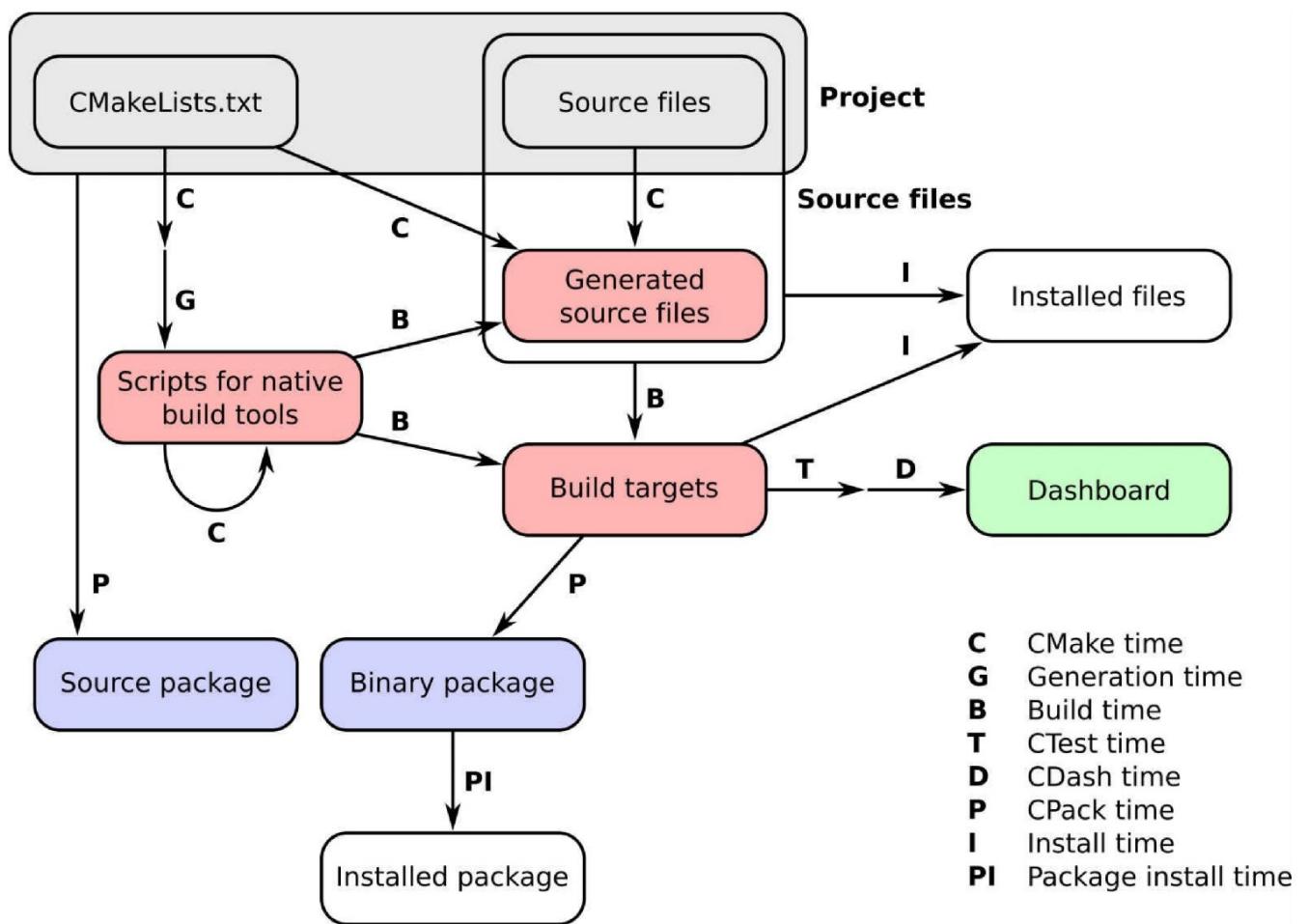
可以交叉编译一组目标平台(例如: Android), 可以参考: <https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html>

# 第14章 测试面板

本章的主要内容有：

- 将测试部署到CDash面板
- CDash面板显示测试覆盖率
- 使用AddressSanifier向CDash报告内存缺陷
- 使用ThreadSanitizer向CDash报告数据争用

CDash是一个web服务，用于汇集CTest在测试运行期间、夜间测试期间或在持续集成中的测试结果。面板报告就是我们所说的**CDash**时，如下图所示：



本章中，我们将向CDash报告测试结果。将讨论报告测试覆盖率的策略，以及分别使用AddressSanifier和ThreadSanitizer等工具，收集的内存缺陷和数据争用问题。

有两种方法向CDash报告结果：

1. 通过构建的测试目标
2. 使用CTest脚本

在前两个示例中使用建立测试目标的方式，在后两个示例中使用CTest脚本。

## CDash环境

---

CDash的安装需要使用PHP和SSL的web服务器(Apache、NGINX或IIS)，并访问MySQL或PostgreSQL数据库服务器。详细讨论CDash web服务的设置超出了本书的范围，读者们可以参考官方文档：<https://public.kitware.com/Wiki/CDash:Installation>

Kitware提供了两个面板(<https://my.cdash.org> 和 <https://open.cdash.org>)，因此本章中的示例并不需要安装CDash。我们将在示例中参考已经提供的面板。

对于想要自己安装CDash的读者，我们建议使用MySQL作为后端，因为这是<https://my.cdash.org> 和 <https://open.cdash.org> 的配置方式，而且社区也对这种搭配方式进行了测试。

**NOTE:**也可以使用Docker来安装CDash。官方镜像的请求在CDash的跟踪器上处于打开状态，网址是<https://github.com/Kitware/CDash/issues/562>

## 14.1 将测试部署到CDash

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-14/recipe-01> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例中，我们将扩展第4章第1节的测试示例，并将测试结果部署到<https://my.cdash.org/index.php?project=cmake-cookbook>，这是在Kitware为社区提供的公共面板(<https://my.cdash.org>)的基础上，为本书创建的专属面板。

### 准备工作

我们将从重用第1节中的示例源代码，该测试将整数作为命令行参数进行求和。该示例由三个源文件组成：`main.cpp`、`sum_integer.cpp` 和 `sum_integers.hpp`。我们还将重用第4章(创建和运行测试)中的 `test.cpp` 文件，但这里将它重命名为 `test_short.cpp`。我们将使用 `test_long.cpp` 扩展这个例子：

```

1. #include "sum_integers.hpp"
2.
3. #include <numeric>
4. #include <vector>
5.
6. int main() {
7.
8. // creates vector {1, 2, 3, ..., 999, 1000}
9. std::vector integers(1000);
10. std::iota(integers.begin(), integers.end(), 1);
11.
12. if (sum_integers(integers) == 500500) {
13. return 0;
14. } else {
15. return 1;
16. }
17. }
```

然后，将这些文件组织成以下文件树：

```

1. .
2. └── CMakeLists.txt
3. └── CTestConfig.cmake
```

```

4. └── src
5. ├── CMakeLists.txt
6. ├── main.cpp
7. ├── sum_integers.cpp
8. └── sum_integers.hpp
9. └── tests
10. ├── CMakeLists.txt
11. ├── test_long.cpp
12. └── test_short.cpp

```

## 具体实施

现在，我们将演示如何配置、构建、测试。最后，将示例项目的测试结果提交到面板的过程：

- 源目标在 `src/CMakeLists.txt` 中定义，如下：

```

1. # example library
2. add_library(sum_integers "")
3.
4. target_sources(sum_integers
5. PRIVATE
6. sum_integers.cpp
7. PUBLIC
8. ${CMAKE_CURRENT_LIST_DIR}/sum_integers.hpp
9.)
10.
11. target_include_directories(sum_integers
12. PUBLIC
13. ${CMAKE_CURRENT_LIST_DIR}
14.)
15.
16. # main code
17. add_executable(sum_up main.cpp)
18.
19. target_link_libraries(sum_up sum_integers)

```

- `tests/CMakeLists.txt` 中定义了测试：

```

1. add_executable(test_short test_short.cpp)
2. target_link_libraries(test_short sum_integers)
3.

```

```

4. add_executable(test_long test_long.cpp)
5. target_link_libraries(test_long sum_integers)
6.
7. add_test(
8. NAME
9. test_short
10. COMMAND
11. $<TARGET_FILE:test_short>
12.)
13.
14. add_test(
15. NAME
16. test_long
17. COMMAND
18. $<TARGET_FILE:test_long>
19.)

```

3. 主 `CMakeLists.txt` 文件引用前面的两个文件，这个配置中的新元素是 `include(CTest)`，这样就可以向CDash仪表板报告结果：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-01 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # process src/CMakeLists.txt
13. add_subdirectory(src)
14. enable_testing()
15.
16. # allow to report to a cdash dashboard
17. include(CTest)
18.
19. # process tests/CMakeLists.txt
20. add_subdirectory(tests)

```

4. 另外，我们创建文件 `CTestConfig.cmake` 与主 `CMakeLists.txt` 文件位于同一目录中。这个

新文件包含以下几行：

```
1. set(CTEST_DROP_METHOD "http")
2. set(CTEST_DROP_SITE "my.cdash.org")
3. set(CTEST_DROP_LOCATION "/submit.php?project=cmake-cookbook")
4. set(CTEST_DROP_SITE_CDASH TRUE)
```

5. 我们现在已经准备好配置和构建项目：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
```

6. 构建后，运行测试集，并向面板报告测试结果：

```
1. $ ctest --dashboard Experimental
2.
3. Site: larry
4. Build name: Linux-c++
5. Create new tag: 20180408-1449 - Experimental
6. Configure project
7. Each . represents 1024 bytes of output
8. . Size of output: OK
9. Build project
10. Each symbol represents 1024 bytes of output.
11. '!' represents an error and '*' a warning.
12. . Size of output: OK
13. 0 Compiler errors
14. 0 Compiler warnings
 Test project /home/user/cmake-recipes/chapter-15/recipe-01/cxx-
15. example/build
16. Start 1: test_short
17. 1/2 Test #1: test_short Passed 0.00 sec
18. Start 2: test_long
19. 2/2 Test #2: test_long Passed 0.00 sec
20. 100% tests passed, 0 tests failed out of 2
21. Total Test time (real) = 0.01 sec
22. Performing coverage
23. Cannot find any coverage files. Ignoring Coverage request.
24. Submit files (using http)
25. Using HTTP submit method
```

```

26. Drop site:http://my.cdash.org/submit.php?project=cmake-cookbook
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-01/cxx-example/build/Test
27. 1449/Build.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-01/cxx-example/build/Test
28. 1449/Configure.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-01/cxx-example/build/Test
29. 1449/Test.xml
30. Submission successful

```



7. 最后，可以在浏览器中看到测试结果(本例中，测试结果上报到

<https://my.cdash.org/index.php?project=cmake-cookbook> )：

The screenshot shows the Cmake-Cookbook project dashboard. At the top, there's a navigation bar with links for Login, All Dashboards, Plans & Pricing, and Support. The date is listed as Sunday, April 08 2018 11:03:25 EDT. Below the navigation is a search bar and a gear icon. The main area has a title 'Cmake-Cookbook' with a logo. A message says 'No file changed as of Sunday, April 08 2018 - 01:00 EDT'. A table titled 'Experimental' shows build statistics for 'larry' on 'Linux-c++'. The table includes columns for Site, Build Name, Update, Configure, Build, and Test. The 'Test' column shows 0 errors, 0 warnings, 0 not run, 0 fails, and 2 passes. The 'Build Time' is listed as '13 minutes ago'.

## 工作原理

可以从更高级的角度展示工作流，CTest运行测试并在XML文件中记录结果。然后，将这些XML文件发送到CDash服务器，在那里可以浏览和分析它们。通过单击数字 2，获得关于通过或失败测试的更多的细节信息(本例中，没有失败的测试)。如下图所示，详细记录了运行测试的机器的信息，以及时间信息。同样，单个测试的测试输出也可以在线浏览。

Testing started on 2018-04-08 14:49:51

**Site Name:**larry  
**Build Name:**Linux-c++  
**Total time:**0s  
**OS Name:**Linux  
**OS Platform:**x86\_64  
**OS Release:**4.15.13-1-ARCH  
**OS Version:**#1 SMP PREEMPT Sun Mar 25 11:27:57 UTC 2018  
**Compiler Name:**/usr/bin/c++  
**Compiler Version:**7.3.1

2 tests passed.

| Name       | Status | Time | Summary |
|------------|--------|------|---------|
| test_long  | Passed | 0s   | Stable  |
| test_short | Passed | 0s   | Stable  |

[Download Table as CSV File](#)

CTest支持三种不同的提交模式：

- 实验性构建
- 夜间构建
- 持续构建

我们使用了 `ctest --dashboard Experimental` (实验性构建提交)，因此，测试结果显示在实验模式之下。实验模式对于测试代码的当前状态、调试新的仪表板脚本、调试CDash服务器或项目非常有用。夜间构建模式，将把代码更新(或降级)到最接近最近夜间构建开始时的存储库，这些可以在 `CTestConfig.cmake` 中设置。其为接收更新频繁的项目的所有夜间测试提供一个定义良好的参考。例如，夜间开始时间可以设置为世界时的“午夜”：

```
1. set(CTEST_NIGHTLY_START_TIME "00:00:00 UTC")
```

持续模式对于集成工作流非常有用，它将把代码更新到最新版本。

**TIPS:** 构建、测试和提交到实验面板只需要一个命令— `cmake --build . --target Experimental`

## 更多信息

这个示例中，我们直接从测试目标部署到CDash。我们将在本章后面的第3和第4部分中，使用专用的CTest脚本。

CDash不仅可以监视测试是否通过或失败，还可以看到测试时间。可以为测试计时进行配置：如果测试花费的时间超过分配的时间，它将被标记为失败。这对于基准测试非常有用，可以在重构代码时自动检测性能测试用例的性能情况。

有关CDash定义和配置设置的详细讨论，请参见官方CDash文档，网址为  
<https://public.kitware.com/Wiki/CDash:Documentation>

## 14.2 CDash显示测试覆盖率

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-14/recipe-02> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

本示例中，我们将测试覆盖率报告给CDash，面板上将能够逐行浏览测试覆盖率分析，以便识别未测试或未使用的代码。

### 准备工作

我们将扩展前一节的源代码，在 `src/sum_integers.cpp` 中做一个小的修改，添加一个函数 `sum_integers_unused`：

```

1. #include "sum_integers.hpp"
2.
3. #include <vector>
4.
5. int sum_integers(const std::vector integers) {
6. auto sum = 0;
7.
8. for (auto i : integers) {
9. sum += i;
10. }
11.
12. return sum;
13. }
14.
15. int sum_integers_unused(const std::vector integers) {
16. auto sum = 0;
17.
18. for (auto i : integers) {
19. sum += i;
20. }
21.
22. return sum;
23. }
```

我们使用gcov(<https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>)通过覆盖率分析检测这个未使用的代码。

# 具体实施

通过以下步骤，我们将使用覆盖率分析，并将结果上传到面板：

1. 主 `CMakeLists.txt` 和 `tests/CMakeLists.txt` 文件与前一个示例相同。
2. 我们将扩展 `src/CMakeLists.txt`，并提供一个选项来添加用于代码覆盖率的编译标志。此选项默认启用：

```

1. option(ENABLE_COVERAGE "Enable coverage" ON)
2.
3. if(ENABLE_COVERAGE)
4. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
5. message(STATUS "Coverage analysis with gcov enabled")
6. target_compile_options(sum_integers
7. PUBLIC
8. -fprofile-arcs -ftest-coverage -g
9.)
10. target_link_libraries(sum_integers
11. PUBLIC
12. gcov
13.)
14. else()
15. message(WARNING "Coverage not supported for this compiler")
16. endif()
17. endif()
```

3. 然后，配置、构建，并将结果上传CDash：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build . --target Experimental
```

4. 最后一步，执行测试覆盖率分析：

```

1. Performing coverage
2. Processing coverage (each . represents one file):
3. ...
4. Accumulating results (each . represents one file):
5. ...
6. Covered LOC: 14
```

```

7. Not covered LOC: 7
8. Total LOC: 21
9. Percentage Coverage: 66.67%
10. Submit files (using http)
11. Using HTTP submit method
12. Drop site:http://my.cdash.org/submit.php?project=cmake-cookbook
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-02/cxx-example/build/
13. 1530/Build.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-02/cxx-example/build/
14. 1530/Configure.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-02/cxx-example/build/
15. 1530/Coverage.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-02/cxx-example/build/
16. 1530/CoverageLog-0.xml
 Uploaded: /home/user/cmake-recipes/chapter-14/recipe-02/cxx-example/build/
17. 1530/Test.xml
18. Submission successful

```



5. 最后，可以在浏览器中验证测试结果(本例的测试结果报告在

<https://my.cdash.org/index.php?project=cmake-cookbook> )：

| Experimental |            | Update | Configure |      | Build |      | Test    |      |      |               |
|--------------|------------|--------|-----------|------|-------|------|---------|------|------|---------------|
| Site         | Build Name | Files  | Error     | Warn | Error | Warn | Not Run | Fail | Pass | Build Time    |
| larry        | Linux-c++  |        | 0         | 0    | 0     | 0    | 0       | 0    | 2    | 3 minutes ago |

| Coverage |            | Percentage | LOC Tested | LOC Untested | Date          |
|----------|------------|------------|------------|--------------|---------------|
| Site     | Build Name |            |            |              |               |
| larry    | Linux-c++  | 66.67%     | 14         | 7            | 3 minutes ago |

## 工作原理

测试覆盖率为66.67%。为了得到更深入的了解，我们可以点击百分比，得到两个子目录的覆盖率分析：



通过浏览器子目录链接，我们可以检查单个文件的测试覆盖率，甚至可以逐行浏览摘要(例如，[src/sum\\_integs.cpp](#) )：

**Coverage File:** ./src/sum\_integers.cpp

```

1 | #include "sum_integers.hpp"
2 |
3 | #include <vector>
4
5 2 | int sum_integers(const std::vector<int> integers) {
6 2 | auto sum = 0;
7 1007 | for (auto i : integers) {
8 1005 | sum += i;
9 |
10 2 | return sum;
11 |
12
13 0 | int sum_integers_unused(const std::vector<int> integers) {
14 0 | auto sum = 0;
15 0 | for (auto i : integers) {
16 0 | sum += i;
17 |
18 0 | return sum;
19 |
20

```

运行测试时，绿线部分已经被覆盖，而红线部分则没有。通过这个方法，我们不仅可以标识未使用的/未测试的代码(使用 `sum_integers_used` 函数)，还可以查看每一行代码被遍历的频率。例如，代码行 `sum += i` 已经被访问了1005次(在 `test_short` 期间访问了5次，在 `test_long` 期间访问了1000次)。测试覆盖率分析是自动化测试不可或缺的功能，CDash为我们提供了一个界面，可以在浏览器中图形化地浏览分析结果。

## 更多信息

为了更多的了解该特性，我们推荐读者阅读下面的博客文章，它更深入的讨论了CDash的覆盖特性：<https://blog.kitware.com/additional-coverage-features-in-cdash/>

## 14.3 使用AddressSanifier向CDash报告内存缺陷

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-14/recipe-03> 中找到，其中包含一个C++示例和一个Fortran例子。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

AddressSanitizer(ASan)是可用于C++、C和Fortran的内存检测。它可以发现内存缺陷，比如：在空闲后使用、返回后使用、作用域后使用、缓冲区溢出、初始化顺序错误和内存泄漏(请参见<https://github.com/google/sanitizers/wiki/AddressSanitizer> )。从3.1版本开始，AddressSanitizer是LLVM的一部分；从4.8版本开始，作为GCC的一部分。在这个示例中，我们将在代码中加入两个bug，正常的测试中可能无法检测到。为了检测这些bug，我们将使用AddressSanitizer工具，并将CTest与动态分析结合起来，从而将缺陷报告给CDash。

## 准备工作

这个例子中，我们将使用两个源文件和两个测试集：

```

1. .
2. └── CMakeLists.txt
3. └── CTestConfig.cmake
4. └── dashboard.cmake
5. └── src
6. ├── buggy.cpp
7. ├── buggy.hpp
8. └── CMakeLists.txt
9. └── tests
10. ├── CMakeLists.txt
11. ├── leaky.cpp
12. └── use_after_free.cpp

```

`buggy.cpp` 包含有两个bug：

```

1. #include "buggy.hpp"
2.
3. #include <iostream>
4.
5. int function_leaky() {
6. double *my_array = new double[1000];
7. // do some work ...

```

```

8. // we forget to deallocate the array
9. // delete[] my_array;
10. return 0;
11. }
12.
13. int function_use_after_free() {
14. double *another_array = new double[1000];
15. // do some work ...
16. // deallocate it, good!
17. delete[] another_array;
18. // however, we accidentally use the array
19. // after it has been deallocated
20. std::cout << "not sure what we get: " << another_array[123] << std::endl;
21. return 0;
22. }
```

这些函数在相应的头文件中声明( `buggy.hpp` )：

```

1. #pragma once
2. int function_leaky();
3. int function_use_after_free();
```

测试文件 `leaky.cpp` 中将会验证 `function_leaky` 的返回值：

```

1. #include "buggy.hpp"
2. int main() {
3. int return_code = function_leaky();
4. return return_code;
5. }
```

相应地，`use_after_free.cpp` 会检查 `function_use_after_free` 的返回值：

```

1. #include "buggy.hpp"
2. int main() {
3. int return_code = function_use_after_free();
4. return return_code;
5. }
```

## 具体实施

为了使用ASan，我们需要使用特定的标志来编译代码。然后，我们将运行测试并将它们提交到面板。

## 1. 生成bug库的工作将在 `src/CMakeLists.txt` 中完成：

```

1. add_library(buggy "")
2.
3. target_sources(buggy
4. PRIVATE
5. buggy.cpp
6. PUBLIC
7. ${CMAKE_CURRENT_LIST_DIR}/buggy.hpp
8.)
9.
10. target_include_directories(buggy
11. PUBLIC
12. ${CMAKE_CURRENT_LIST_DIR}
13.)

```

## 2. 在文件 `src/CMakeLists.txt` 中，我们将添加一个选项用于使用ASan：

```

1. option(ENABLE_ASAN "Enable AddressSanitizer" OFF)
2.
3. if(ENABLE_ASAN)
4. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
5. message(STATUS "AddressSanitizer enabled")
6. target_compile_options(buggy
7. PUBLIC
8. -g -O1 -fsanitize=address -fno-omit-frame-pointer
9.)
10. target_link_libraries(buggy
11. PUBLIC
12. asan
13.)
14. else()
15. message(WARNING "AddressSanitizer not supported for this compiler")
16. endif()
17. endif()

```

## 3. 测试在 `tests/CMakeLists.txt` 中定义：

```

1. foreach(_test IN ITEMS leaky use_after_free)
2. add_executable(${_test} ${_test}.cpp)
3. target_link_libraries(${_test} buggy)
4.

```

```

5. add_test(
6. NAME
7. ${_test}
8. COMMAND
9. ${TARGET_FILE}:${_test}>
10.)
11. endforeach()

```

4. 主 `CMakeLists.txt` 与之前的示例基本相同：

```

1. # set minimum cmake version
2. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
3.
4. # project name and language
5. project(recipe-03 LANGUAGES CXX)
6.
7. # require C++11
8. set(CMAKE_CXX_STANDARD 11)
9. set(CMAKE_CXX_EXTENSIONS OFF)
10. set(CMAKE_CXX_STANDARD_REQUIRED ON)
11.
12. # process src/CMakeLists.txt
13. add_subdirectory(src)
14. enable_testing()
15.
16. # allow to report to a cdash dashboard
17. include(CTest)
18.
19. # process tests/CMakeLists.txt
20. add_subdirectory(tests)

```

5. `CTestConfig.cmake` 也没有修改：

```

1. set(CTEST_DROP_METHOD "http")
2. set(CTEST_DROP_SITE "my.cdash.org")
3. set(CTEST_DROP_LOCATION "/submit.php?project=cmake-cookbook")
4. set(CTEST_DROP_SITE_CDASH TRUE)

```

6. 这个示例中，我们使用CTest脚本向CDash提交结果；为此，我们将创建一个文件 `dashboard.cmake`（与主 `CMakeLists.txt` 和 `CTestConfig.cmake` 位于同一个目录下）：

```

1. set(CTEST_PROJECT_NAME "example")
2. cmake_host_system_information(RESULT _site QUERY HOSTNAME)
3. set(CTEST_SITE ${_site})
 set(CTEST_BUILD_NAME "${CMAKE_SYSTEM_NAME}-"
4. ${CMAKE_HOST_SYSTEM_PROCESSOR}")
5.
6. set(CTEST_SOURCE_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}")
7. set(CTEST_BINARY_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}/build")
8.
9. include(ProcessorCount)
10. ProcessorCount(N)
11. if(NOT N EQUAL 0)
12. set(CTEST_BUILD_FLAGS -j${N})
13. set(ctest_test_args ${ctest_test_args} PARALLEL_LEVEL ${N})
14. endif()
15.
16. ctest_start(Experimental)
17.
18. ctest_configure(
19. OPTIONS
20. -DENABLE_ASAN:BOOL=ON
21.)
22.
23. ctest_build()
24. ctest_test()
25.
26. set(CTEST_MEMORYCHECK_TYPE "AddressSanitizer")
27. ctest_memcheck()
28.
29. ctest_submit()

```

7. 我们将执行 `dashboard.cmake` 脚本。注意，我们使用 `CTEST_CMAKE_GENERATOR` 与生成器选项的方式：

```

1. $ ctest -S dashboard.cmake -D
2.
3. CTEST_CMAKE_GENERATOR="Unix Makefiles"
4. Each . represents 1024 bytes of output
5. . Size of output: OK
6. Each symbol represents 1024 bytes of output.
7. '!' represents an error and '*' a warning.
8. . Size of output: 1K

```

8. 结果将会出现在CDash网站上：

The screenshot shows the CDash dashboard for the 'Cmake-Cookbook' project. At the top, there are links for 'Login', 'All Dashboards', 'Plans & Pricing', and 'Support'. The date 'Sunday, April 08 2018 11:44:01 EDT' is displayed. Below the header, there's a 'Cmake-Cookbook' logo and navigation tabs for 'Dashboard', 'Calendar', 'Previous', 'Current', and 'Project'. A message 'No file changed as of Sunday, April 08 2018 - 01:00 EDT' is shown. The main area is divided into sections: 'Experimental' (Build Name: Linux-x86\_64, Build Time: 2 minutes ago) and 'Dynamic Analysis' (Checker: AddressSanitizer, Defect Count: 2). The 'Dynamic Analysis' section includes a table with columns for Site, Build Name, Checker, Defect Count, and Date.

| Site      | Build Name   | Checker          | Defect Count | Date          |
|-----------|--------------|------------------|--------------|---------------|
| localhost | Linux-x86_64 | AddressSanitizer | 2            | 2 minutes ago |

## 具体实施

这个示例中，成功地向仪表板的动态分析部分报告了内存错误。我们可以通过浏览缺陷详细信息，得到进一步的了解：

**Dynamic analysis started on 2018-04-08 15:41:20**

**Site Name:** localhost

**Build Name:** Linux-x86\_64

| Name           | Status | Direct leak | heap-use-after-free | Labels |
|----------------|--------|-------------|---------------------|--------|
| leaky          | Failed | 1           |                     |        |
| use_after_free | Failed |             | 1                   |        |

通过单击各个链接，可以浏览完整信息的输出。

注意，也可以在本地生成AddressSanitizer报告。这个例子中，我们需要设置 `ENABLE_ASAN` :

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake -DENABLE_ASAN=ON ..
4. $ cmake --build .
5. $ cmake --build . --target test
6.
7. Start 1: leaky
8. 1/2 Test #1: leaky ***Failed 0.07 sec
9. Start 2: use_after_free
10. 2/2 Test #2: use_after_free ***Failed 0.04 sec
11. 0% tests passed, 2 tests failed out of 2

```

运行 `leaky` 测试，直接产生以下结果：

```

1. $./build/tests/leaky
2.
3. =====
4. ==18536==ERROR: LeakSanitizer: detected memory leaks
5. Direct leak of 8000 byte(s) in 1 object(s) allocated from:
#0 0x7ff984da1669 in operator new[](unsigned long)
6. /build/gcc/src/gcc/libsanitizer/asan/asan_new_delete.cc:82
#1 0x564925c93fd2 in function_leaky() /home/user/cmake-recipes/chapter-
7. 14/recipe-03/cxx-example/src/buggy.cpp:7
#2 0x564925c93fb2 in main /home/user/cmake-recipes/chapter-14/recipe-03/cxx-
8. example/tests/leaky.cpp:4
#3 0x7ff98403df49 in __libc_start_main (/usr/lib/libc.so.6+0x20f49)
9. SUMMARY: AddressSanitizer: 8000 byte(s) leaked in 1 allocation(s).

```

相应地，我们可以直接运行 `use_after_free`，得到详细的输出：

```

1. $./build/tests/use_after_free
2.
3. =====
==18571==ERROR: AddressSanitizer: heap-use-after-free on address 0x6250000004d8
4. at pc 0x557ffa8b0102 bp 0x7ffe8c560200 sp 0x7ffe8c5601f0
5. READ of size 8 at 0x6250000004d8 thread T0
#0 0x557ffa8b0101 in function_use_after() /home/user/cmake-
6. recipes/chapter-14/recipe-03/cxx-example/src/buggy.cpp:28
#1 0x557ffa8affb2 in main /home/user/cmake-recipes/chapter-14/recipe-03/cxx-
7. example/tests/use_after_free.cpp:4
#2 0x7ff1d6088f49 in __libc_start_main (/usr/lib/libc.so.6+0x20f49)
#3 0x557ffa8afec9 in _start (/home/user/cmake-recipes/chapter-14/recipe-03/cxx-
9. example/build/tests/use_after_free+0xec9)
0x6250000004d8 is located 984 bytes inside of 8000-byte region
10. [0x625000000100,0x625000002040)
11. freed by thread T0 here:
#0 0x7ff1d6ded5a9 in operator delete[](void*)
12. /build/gcc/src/gcc/libsanitizer/asan/asan_new_delete.cc:128
#1 0x557ffa8afffa in function_use_after_free() /home/user/cmake-
13. recipes/chapter-14/recipe-03/cxx-example/src/buggy.cpp:24
#2 0x557ffa8affb2 in main /home/user/cmake-recipes/chapter-14/recipe-03/cxx-
14. example/tests/use_after_free.cpp:4
#3 0x7ff1d6088f49 in __libc_start_main (/usr/lib/libc.so.6+0x20f49)
15. previously allocated by thread T0 here:

```

```

#0 0x7ff1d6dec669 in operator new[](unsigned long)
17. /build/gcc/src/gcc/libasanizer/asan/asan_new_delete.cc:82
 #1 0x557ffa8affea in function_use_after_free() /home/user/cmake-
18. recipes/chapter-14/recipe-03/cxx-example/src/buggy.cpp:19
 #2 0x557ffa8affb2 in main /home/user/cmake-recipes/chapter-14/recipe-03/cxx-
19. example/tests/use_after_free.cpp:4
20. #3 0x7ff1d6088f49 in __libc_start_main (/usr/lib/libc.so.6+0x20f49)
 SUMMARY: AddressSanitizer: heap-use-after-free /home/user/cmake-
 recipes/chapter-14/recipe-03/cxx-example/src/buggy.cpp:28 in
21. function_use_after_free()
22. Shadow bytes around the buggy address:
23. 0x0c4a7fff8040: fd fd
24. 0x0c4a7fff8050: fd fd
25. 0x0c4a7fff8060: fd fd
26. 0x0c4a7fff8070: fd fd
27. 0x0c4a7fff8080: fd fd
28. =>0x0c4a7fff8090: fd fd
29. 0x0c4a7fff80a0: fd fd
30. 0x0c4a7fff80b0: fd fd
31. 0x0c4a7fff80c0: fd fd
32. 0x0c4a7fff80d0: fd fd
33. 0x0c4a7fff80e0: fd fd
34. Shadow byte legend (one shadow byte represents 8 application bytes):
35. Addressable: 00
36. Partially addressable: 01 02 03 04 05 06 07
37. Heap left redzone: fa
38. Freed heap region: fd
39. Stack left redzone: f1
40. Stack mid redzone: f2
41. Stack right redzone: f3
42. Stack after return: f5
43. Stack use after scope: f8
44. Global redzone: f9
45. Global init order: f6
46. Poisoned by user: f7
47. Container overflow: fc
48. Array cookie: ac
49. Intra object redzone: bb
50. ASan internal: fe
51. Left alloca redzone: ca
52. Right alloca redzone: cb
53. ==18571==ABORTING

```

如果我们在没有AddressSanitizer的情况下进行测试(默认情况下 `ENABLE_ASAN` 是关闭的), 就不会报告错误:

```

1. $ mkdir -p build_no_asan
2. $ cd build_no_asan
3. $ cmake ..
4. $ cmake --build .
5. $ cmake --build . --target test
6.
7. Start 1: leaky
8. 1/2 Test #1: leaky Passed 0.00 sec
9. Start 2: use_after_free
10. 2/2 Test #2: use_after_free Passed 0.00 sec
11. 100% tests passed, 0 tests failed out of 2

```

实际上, 泄漏只会浪费内存, 而 `use_after_free` 可能会导致未定义行为。调试这些问题的一种方法是使用valgrind (<http://valgrind.org> )。

与前两个示例相反, 我们使用了CTest脚本来配置、构建和测试代码, 并将报告提交到面板。要了解此示例的工作原理, 请仔细查看 `dashboard.cmake` 脚本。首先, 我们定义项目名称并设置主机报告和构建名称:

```

1. set(CTEST_PROJECT_NAME "example")
2. cmake_host_system_information(RESULT _site QUERY HOSTNAME)
3. set(CTEST_SITE ${_site})
4. set(CTEST_BUILD_NAME "${CMAKE_SYSTEM_NAME}-${CMAKE_HOST_SYSTEM_PROCESSOR}")

```

我们的例子中, `CTEST_BUILD_NAME` 的计算结果是 `Linux-x86_64` 。不同的操作系统下, 可能会观察到不同的结果。

接下来, 我们为源和构建目录指定路径:

```

1. set(CTEST_SOURCE_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}")
2. set(CTEST_BINARY_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}/build")

```

我们可以将生成器设置为 `Unix Makefile` :

```
1. set(CTEST_CMAKE_GENERATOR "Unix Makefiles")
```

但是, 对于更具移植性的测试脚本, 我们更愿意通过命令行提供生成器:

```
1. $ ctest -S dashboard.cmake -D CTEST_CMAKE_GENERATOR="Unix Makefiles"
```

`dashboard.cmake` 中的下一个代码片段，将计算出机器上可用的CPU芯数量，并将测试步骤的并行级设置为可用CPU芯数量，以使总测试时间最小化：

```
1. include(ProcessorCount)
2. ProcessorCount(N)
3. if(NOT N EQUAL 0)
4. set(CTEST_BUILD_FLAGS -j${N})
5. set(ctest_test_args ${ctest_test_args} PARALLEL_LEVEL ${N})
6. endif()
```

接下来，我们开始测试步骤并配置代码，将 `ENABLE_ASAN` 设置为 `ON`：

```
1. ctest_start(Experimental)
2.
3. ctest_configure(
4. OPTIONS
5. -DENABLE_ASAN:BOOL=ON
6.)
```

`dashboard.cmake` 其他命令为映射到构建、测试、内存检查和提交步骤：

```
1. ctest_build()
2. ctest_test()
3.
4. set(CTEST_MEMORYCHECK_TYPE "AddressSanitizer")
5.
6. ctest_memcheck()
7. ctest_submit()
```

## 更多信息

细心的读者会注意到，在链接目标之前，我们没有在系统上搜索AddressSanitizer。实际上，库查找工作已经提前做完，以避免在链接阶段出现意外。

有关AddressSanitizer文档和示例的更多信息，请参见

<https://github.com/google/sanitizers/wiki/AddressSanitizer>。

AddressSanitizer并不仅限于C和C++。对于Fortran示例，读者可以参考

<https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-14/recipe->

### 03/fortran-example 。

**NOTE:**可以在<https://github.com/arsenm/sanitizers-cmake> 上找到CMake程序，用来查找杀毒程序和调整编译器标志

下面的博客文章讨论了如何添加对动态分析工具的支持，对我们很有启发性：<https://blog.kitware.com/ctest-cdash-add-support-for-new-dynamic-analysis-tools/>

## 14.4 使用ThreadSanitizer向CDash报告数据争用

**NOTE:** 此示例代码可以在 <https://github.com/dev-cafe/cmake-cookbook/tree/v1.0/chapter-14/recipe-03> 中找到，其中包含一个C++示例。该示例在CMake 3.5版(或更高版本)中是有效的，并且已经在GNU/Linux、macOS和Windows上进行过测试。

在这个示例中，我们将重用前一个示例中的方法，但是使用ThreadSanitizer或TSan，结合CTest和CDash，来检查数据竞争，并将它们报告给CDash。ThreadSanitizer的文档可以在网上找到，<https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>

### 准备工作

这个示例中，我们将使用以下示例代码(`example.cpp`)：

```

1. #include <chrono>
2. #include <iostream>
3. #include <thread>
4.
5. static const int num_threads = 16;
6.
7. void increase(int i, int &s) {
8. std::this_thread::sleep_for(std::chrono::seconds(1));
9. std::cout << "thread " << i << " increases " << s++ << std::endl;
10. }
11.
12. int main() {
13. std::thread t[num_threads];
14.
15. int s = 0;
16.
17. // start threads
18. for (auto i = 0; i < num_threads; i++) {
19. t[i] = std::thread(increase, i, std::ref(s));
20. }
21.
22. // join threads with main thread
23. for (auto i = 0; i < num_threads; i++) {
24. t[i].join();
25. }
26.
```

```

27. std::cout << "final s: " << s << std::endl;
28.
29. return 0;
30. }
```

这个示例代码中，我们启动16个线程，每个线程都调用 `increase` 函数。`increase` 函数休眠1s，然后打印并递增一个整数 `s`。我们预计此示例代码将显示数据竞争，因为所有线程读取和修改相同的地址，而不需要任何显式同步或协调。换句话说，我们期望在代码末尾打印的最终 `s`，每次的结果都不同。代码有bug，我们将尝试在ThreadSanitizer的帮助下识别数据竞争。如果不运行 ThreadSanitizer，我们可能不会看到代码有任何问题：

```

1. $./example
2.
3. thread thread 0 increases 01 increases 1
4. thread 9 increases 2
5. thread 4 increases 3
6. thread 10 increases 4
7. thread 2 increases 5
8. thread 3 increases 6
9. thread 13 increases 7
10. thread thread 7 increases 8
11. thread 14 increases 9
12. thread 8 increases 10
13. thread 12 increases 11
14. thread 15 increases 12
15. thread 11 increases 13
16.
17. 5 increases 14
18. thread 6 increases 15
19. final s: 16
```

## 具体实施

- 文件 `CMakeLists.txt` 首先定义一个受支持的最低版本、项目名称、受支持的语言。在本例中，定义了C++11标准项目：

```

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
2.
3. project(recipe-04 LANGUAGES CXX)
4.
5. set(CMAKE_CXX_STANDARD 11)
```

```

6. set(CMAKE_CXX_EXTENSIONS OFF)
7. set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

2. 接下来，找到线程库，定义可执行文件，并将其链接到线程库：

```

1. find_package(Threads REQUIRED)
2.
3. add_executable(example example.cpp)
4.
5. target_link_libraries(example
6. PUBLIC
7. Threads::Threads
8.)

```

3. 然后，提供编译选项和代码，并链接到ThreadSanitizer：

```

1. option(ENABLE_TSAN "Enable ThreadSanitizer" OFF)
2.
3. if(ENABLE_TSAN)
4. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
5. message(STATUS "ThreadSanitizer enabled")
6. target_compile_options(example
7. PUBLIC
8. -g -O1 -fsanitize=thread -fno-omit-frame-pointer -fPIC
9.)
10. target_link_libraries(example
11. PUBLIC
12. tsan
13.)
14. else()
15. message(WARNING "ThreadSanitizer not supported for this compiler")
16. endif()
17. endif()

```

4. 最后，编译测试用例：

```

1. enable_testing()
2.
3. # allow to report to a cdash dashboard
4. include(CTest)
5.
6. add_test(

```

```

7. NAME
8. example
9. COMMAND
10. $<TARGET_FILE:example>
11.)

```

5. `CTestConfig.cmake` 没有变化：

```

1. set(CTEST_DROP_METHOD "http")
2. set(CTEST_DROP_SITE "my.cdash.org")
3. set(CTEST_DROP_LOCATION "/submit.php?project=cmake-cookbook")
4. set(CTEST_DROP_SITE_CDASH TRUE)

```

6. `dashboard.cmake` 需要为TSan进行简单修改：

```

1. set(CTEST_PROJECT_NAME "example")
2. cmake_host_system_information(RESULT _site QUERY HOSTNAME)
3. set(CTEST_SITE ${_site})
 set(CTEST_BUILD_NAME "${CMAKE_SYSTEM_NAME}-"
4. ${CMAKE_HOST_SYSTEM_PROCESSOR})
5.
6. set(CTEST_SOURCE_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}")
7. set(CTEST_BINARY_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}/build")
8.
9. include(ProcessorCount)
10. ProcessorCount(N)
11. if(NOT N EQUAL 0)
12. set(CTEST_BUILD_FLAGS -j${N})
13. set(ctest_test_args ${ctest_test_args} PARALLEL_LEVEL ${N})
14. endif()
15.
16. ctest_start(Experimental)
17.
18. ctest_configure(
19. OPTIONS
20. -DENABLE_TSAN:BOOL=ON
21.)
22.
23. ctest_build()
24. ctest_test()
25.
26. set(CTEST_MEMORYCHECK_TYPE "ThreadSanitizer")

```

```

27. ctest_memcheck()
28.
29. ctest_submit()

```

7. 让我们以这个例子为例。通过 `CTEST_CMAKE_GENERATOR` 选项来设置生成器：

```

1. $ ctest -S dashboard.cmake -D CTEST_CMAKE_GENERATOR="Unix Makefiles"
2.
3. Each . represents 1024 bytes of output
4. . Size of output: OK
5. Each symbol represents 1024 bytes of output.
6. !!! represents an error and ** a warning.
7. . Size of output: OK

```

8. 在面板上，我们将看到以下内容：

The screenshot shows the CDash project 'Cmake-Cookbook' dashboard. At the top, there's a navigation bar with links for Login, All Dashboards, Plans & Pricing, Support, and the current date, Sunday, April 08 2018 11:45:03 EDT. Below the navigation is the project logo and name 'Cmake-Cookbook'. A message indicates 'No file changed as of Sunday, April 08 2018 - 01:00 EDT'. The main content area is divided into sections: 'Experimental' and 'Dynamic Analysis'. The 'Experimental' section has a table with columns for Site, Build Name, Update, Configure, Build, and Test. It shows data for 'localhost' with a build named 'Linux-x86\_64'. The 'Test' row shows 0 errors, 0 warnings, 0 not run, 1 fail, and 0 passes. The 'Build Time' is listed as 'just now'. The 'Dynamic Analysis' section has a table with columns for Site, Build Name, Checker, Defect Count, and Date. It shows data for 'localhost' with a build named 'Linux-x86\_64' using the 'ThreadSanitizer' checker, with a defect count of 2 and a date of 'just now'.

9. 我们可以看到更详细的动态分析：

## Dynamic analysis started on 2018-04-08 15:44:35

| Name    | Status | data race | Labels |
|---------|--------|-----------|--------|
| example | Failed | 2         |        |

## 工作原理

该示例 `CMakeLists.txt` 的核心部分：

```

1. option(ENABLE_TSAN "Enable ThreadSanitizer" OFF)
2.
3. if(ENABLE_TSAN)
4. if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
5. message(STATUS "ThreadSanitizer enabled")
6. target_compile_options(example
7. PUBLIC
8. -g -O1 -fsanitize=thread -fno-omit-frame-pointer -fPIC
9.)
10. target_link_libraries(example
11. PUBLIC
12. tsan
13.)
14. else()
15. message(WARNING "ThreadSanitizer not supported for this compiler")
16. endif()
17. endif()

```

`dashboard.cmake` 也需要更新：

```

1. # ...
2.
3. ctest_start(Experimental)
4.
5. ctest_configure(
6. OPTIONS
7. -DENABLE_TSAN:BOOL=ON
8.)
9.
10. ctest_build()
11. ctest_test()
12.
13. set(CTEST_MEMORYCHECK_TYPE "ThreadSanitizer")
14. ctest_memcheck()
15.
16. ctest_submit()

```

和上一个示例一样，我们也可以在本地查看ThreadSanitizer的输出：

```

1. $ mkdir -p build
2. $ cd build

```

```

3. $ cmake -DENABLE_TSAN=ON ..
4. $ cmake --build .
5. $ cmake --build . --target test
6.
7. Start 1: example
8. 1/1 Test #1: example***Failed 1.07 sec
9. 0% tests passed, 1 tests failed out of 1
10. $./build/example
11. thread 0 increases 0
12. =====
13. WARNING: ThreadSanitizer: data race (pid=24563)
14. ... lots of output ...
 SUMMARY: ThreadSanitizer: data race /home/user/cmake-recipes/chapter-14/recipe-
15. 04/cxx-example/example

```

## 更多信息

对使用OpenMP的应用TSan是很常见的，但是请注意，在某些情况下，OpenMP会在TSan下生成误检的结果。对于Clang编译器，一个解决方案是用 `-DLIBOMP_TSAN_SUPPORT=TRUE` 重新编译编译器本身及其 `libomp`。通常，以合理的方式使用TSan可能需要重新编译整个工具堆栈，以避免误报。在使用pybind11的C++项目的情况下，我们可能需要重新编译Python，并启用TSan来获得有意义的东西。或者，Python绑定可以通过使用TSan抑制而被排除在外，如

<https://github.com/google/sanitizers/wiki/threadsanitizerssuppression>。例如：如果一个动态库同时被一个经过TSan的二进制文件和一个Python插件调用，那么这种情况可能是不可能使用TSan。

下面的博客文章讨论了如何添加对动态分析工具的支持：<https://blog.kitware.com/ctest-cdash-add-support-for-new-dynamic-analysis-tools/>

# 第15章 使用CMake构建已有项目

在本书的最后一章中，我们将结合前几章中讨论过的许多不同的构建块，并将它们应用到实际项目中。我们的目标是一步步地演示如何将一个重要的项目使用CMake进行构建。提供关于移植项目或将CMake添加到遗留代码的建议(无论是来自Autotools、手工编写的配置脚本和Makefile，还是来自Visual Studio)。

为了得到一个实际示例，我们将使用Vim编辑器(<https://www.vim.org>)的源代码，并尝试将配置和编译，从Autotools迁移到CMake。

为了使讨论和示例相对简单，我们不会将整个Vim项目迁移到CMake，而是挑出最重要的部分。只构建Vim的核心版本，不支持图形用户界面(GUI)。我们将获取能够编译的Vim源码版本，并使用CMake，用书中介绍的其他工具进行配置、构建和测试。

本章主要有以下内容：

- 如何开始迁移项目
- 生成文件并编写平台检查
- 检测所需的依赖关系和链接
- 复制编译标志
- 移植测试
- 移植安装目标
- 项目转换为CMake的常见问题

## 15.1 如何开始迁移项目

我们将首先说明，在哪里可以找到我们的示例，然后对移植，进行逐步的讨论。

### 复制要移植的示例

我们将从Vim源代码库的v8.1.0290发行标记开始(<https://github.com/vim/vim>)，我们的工作基于Git提交哈希值b476cb7进行。通过克隆Vim的源代码库并检出特定版本的代码，可以复制以下步骤：

```
1. $ git clone --single-branch -b v8.1.0290 https://github.com/vim/vim.git
```

或者，我们的解决方案可以在 `cmake-support` 分支上找到，网址是 <https://github.com/dev-cafe/vim>，并使用以下方法克隆下来：

```
1. $ git clone --single-branch -b cmake-support https://github.com/dev-cafe/vim
```

在本例中，我们将使用CMake模拟 `./configure --enable-gui=no` 的配置方式。

为了与后面的解决方案进行比较，建议读者也可以研究以下Neovim项目(<https://github.com/neovim/neovim>)，这是传统Vi编辑器的一个分支，提供了一个CMake构建系统。

### 创建一个主CMakeLists.txt

首先，我们在源代码存储库的根目录中创建主 `CMakeLists.txt`，在这里我们设置了最低CMake版本、项目名称和支持的语言，在本例中是C：

```
1. cmake_minimum_required(VERSION
2. 3.5 FATAL_ERROR)
3. project(vim LANGUAGES C)
```

添加任何目标或源之前，可以设置默认的构建类型。本例中，我们认为Release配置，这将打开某些编译器优化选项：

```
1. if(NOT CMAKE_BUILD_TYPE)
2. set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
3. endif()
```

我们也使用可移植的安装目录变量：

```

1. include(GNUInstallDirs)
2. set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
3. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
4. set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
5. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
6. set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
7. ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

```

作为一个完整性检查，我们可以尝试配置和构建项目，但到目前为止还没有目标，所以构建步骤的输出是空的：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .

```

我们一会儿就要开始添加目标了。

## 如何让常规和CMake配置共存

CMake的一个特性是在源代码之外构建，构建目录可以是任何目录，而不必是项目目录的子目录。这意味着，我们可以将一个项目移植到CMake，而不影响以前/现在的配置和构建机制。对于一个重要的项目的迁移，CMake文件可以与其他构建框架共存，从而允许一个渐进的迁移，包括选项、特性和可移植性，并允许开发社区人员适应新的框架。为了允许传统配置和CMake配置共存一段时间，一个典型的策略是收集 `CMakeLists.txt` 文件中的所有CMake代码，以及CMake子目录下的所有辅助CMake源文件的示例中，我们不会引入CMake子目录，而是保持辅助文件要求他们接近目标和来源，但会顾及使用的传统Autotools构建修改的所有文件，但有一个例外：我们将一些修改自动生成文件构建目录下，而不是在源代码树中。

```

1. $./configure --enable-gui=no
2.
3. ... lot of output ...
4.
5. $ make > build.log

```

我们的示例中(这里没有显示`build.log`的内容)，我们能够验证编译了哪些源文件以及使用了哪些编译标志(`-Iproto -DHAVE_CONFIG_H -g -O2 -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=1`)。日志文件中，我们可以做如下推断：

- 所有对象文件都链接到二进制文件中
- 不生成库
- 可执行目标与下列库进行连接: `-lSM -lICE -lXpm -lXt -lX11 -lXdmcp -lSM -lICE -lM -ltinfo -lelf -lns1 -lacl -lattr -lgpm -ldl`

通过在使用 `message` 对工程进行调试时, 选择添加选项、目标、源和依赖项, 我们将逐步实现一个可工作的构建。

## 获取传统构建的记录

向配置添加任何目标之前, 通常有必要看看传统构建的行为, 并将配置和构建步骤的输出保存到日志文件中。对于我们的Vim示例, 可以使用以下方法实现:

```
1. $./configure --enable-gui=no
2.
3. ... lot of output ...
4.
5. $ make > build.log
```

示例中(这里没有显示`build.log`的完整内容), 我们能够验证编译了哪些源文件以及使用了哪些编译标志(`-I.-Iproto -DHAVE_CONFIG_H -g -O2 -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=1`)。从日志文件中, 推断如下:

- 所有对象文件都链接到一个二进制文件中
- 没有生成库
- 可执行目标链接到以下库: `-lSM -lXpm -lXt -lX11 -lXdmcp -lSM -lSM - linfo -lelf -lns1 -lacl -lattr -lgpm -ldl`

## 调试迁移项目

当目标和命令逐渐移动到CMake端时, 使用 `message` 命令打印变量的值就非常有用了:

```
1. message(STATUS "for debugging printing the value of ${some_variable}")
```

在使用消息进行调试时, 添加选项、目标、源和依赖项, 我们将逐步实现一个可工作的构建。

## 实现选项

找出传统配置为用户提供的选项(例如, 通过 `./configure --help`)。Vim项目提供了一个非常长的选项和标志列表, 为了使本章的讨论保持简单, 我们只在CMake端实现四个选项:

```

1. --disable-netbeans Disable NetBeans integration support.
2. --disable-channel Disable process communication support.
3. --enable-terminal Enable terminal emulation support.
4. --with-features=TYPE tiny, small, normal, big or huge (default: huge)

```

我们还将忽略任何GUI支持和模拟 `--enable-gui=no`，因为它将使示例复杂化。

我们将在CMakeLists.txt中添加以下选项(有默认值)：

```

1. option(ENABLE_NETBEANS "Enable netbeans" ON)
2. option(ENABLE_CHANNEL "Enable channel" ON)
3. option(ENABLE_TERMINAL "Enable terminal" ON)

```

我们可以用 `cmake -D FEATURES=value` 定义的变量 `FEATURES` 来模拟 `--with-features` 标志。如果不进行设置，它默认值为“huge”：

```

1. if(NOT FEATURES)
2. set(FEATURES "huge" CACHE STRING
3. "FEATURES chosen by the user at CMake configure time")
4. endif()

```

我们为使用者提供了一个值 `FEATURES`：

```

1. list(APPEND _available_features "tiny" "small" "normal" "big" "huge")
2. if(NOT FEATURES IN_LIST _available_features)
 message(FATAL_ERROR "Unknown features: \"${FEATURES}\". Allowed values are:
3. ${_available_features}.")
4. endif()
5. set_property(CACHE FEATURES PROPERTY STRINGS ${_available_features})

```

最后一行 `set_property(CACHE FEATURES PROPERTY STRINGS ${_available_features})`，当使用 `cmake-gui` 配置项目，则有不错的效果，用户可根据选择字段清单，选择已经定义了的 `FEATURES` (参见<https://blog.kitware.com/constraining-values-with-comboboxes-in-cmake-gui/>)。

选项可以放在主 `CMakeLists.txt` 中，也可以在查

询 `ENABLE_NETBEANS`、`ENABLE_CHANNEL`、`ENABLE_TERMINAL` 和 `FEATURES` 的定义附近。前一种策略的优点是，选项列在一个地方，不需要遍历 `CMakeLists.txt` 文件来查找选项的定义。因为我们还没有定义任何目标，所以可以先将选项保存在一个文件中，但是稍后会将选项移到离目标更近的地方，通过本地化作用域，得到可重用的CMake构建块。

# 从可执行的目标开始，进行本地化

让我们添加一些源码。在Vim示例中，源文件位于 `src` 下，为了保持主 `CMakeLists.txt` 的可读性和可维持性，我们将创建一个新文件 `src/CMakeLists.txt`，并将其添加到主 `CMakeLists.txt` 中，从而可以在自己的目录范围内处理该文件：

```
1. add_subdirectory(src)
```

在 `src/CMakeLists.txt` 中，可以定义可执行目标，并列出从 `build.log` 中获取所有源码：

```
1. add_executable(vim
 arabic.c beval.c buffer.c blowfish.c crypt.c crypt_zip.c dict.c diff.c
 digraph.c edit.c eval.c evalfunc.c ex_cmds.c ex_cmds2.c ex_docmd.c ex_eval.c
 ex_getln.c farsi.c fileio.c fold.c getchar.c hardcopy.c hashtable.c if_cscope.c
 if_xcmdsrv.c list.c mark.c memline.c menu.c misc1.c misc2.c move.c mbyte.c
 normal.c ops.c option.c os_unix.c auto/pathdef.c popupmnu.c pty.c quickfix.c
 regexp.c screen.c search.c sha256.c spell.c spellfile.c syntax.c tag.c term.c
 terminal.c ui.c undo.c userfunc.c window.c libvterm/src/encoding.c
 libvterm/src/keyboard.c libvterm/src/mouse.c libvterm/src/parser.c
 libvterm/src/pen.c libvterm/src/screen.c libvterm/src/state.c
 libvterm/src/unicode.c libvterm/src/vterm.c netbeans.c channel.c charset.c
2. json.c main.c memfile.c message.c version.c
3.)
```

这是一个开始。这种情况下，代码甚至不会配置，因为源列表包含生成的文件。讨论生成文件和链接依赖项之前，我们把这一长列表拆分一下，以限制目标依赖项的范围，并使项目更易于管理。如果我们将它们分组到目标，这将使CMake更容易地找到源文件依赖项，并避免很长的链接行。

对于Vim示例，我们可以进一步了解来自 `src/Makefile` 和 `src/configure.ac` 的源码文件进行分组。这些文件中，大多数源文件都是必需的。有些源文件是可选的(`netbeans.c` 应该只在 `ENABLE_NETBEANS` 打开时构建，而 `channel.c` 应该只在 `ENABLE_CHANNEL` 打开时构建)。此外，我们可以将所有源代码分组到 `src/libvterm/` 下，并使用 `ENABLE_TERMINAL` 可选地编译它们。

这样，我们将CMake结构重组，构成如下的树结构：

```
1. .
2. └── CMakeLists.txt
3. └── src
4. ├── CMakeLists.txt
5. └── libvterm
6. └── CMakeLists.txt
```

顶层文件使用 `add_subdirectory(src)` 添加 `src/CMakeLists.txt`。 `src/CMakeLists.txt` 文件包含三个目标(一个可执行文件和两个库)，每个目标都带有编译定义和包含目录。首先定义可执行文件：

```

1. add_executable(vim
2. main.c
3.)
4.
5. target_compile_definitions(vim
6. PRIVATE
7. "HAVE_CONFIG_H"
8.)
```

然后，定义一些需要源码文件的目标：

```

1. add_library(basic_sources "")
2.
3. target_sources(basic_sources
4. PRIVATE
5. arabic.c beval.c blowfish.c buffer.c charset.c
6. crypt.c crypt_zip.c dict.c diff.c digraph.c
7. edit.c eval.c evalfunc.c ex_cmds.c ex_cmds2.c
8. ex_docmd.c ex_eval.c ex_getln.c farsi.c fileio.c
9. fold.c getchar.c hardcopy.c hashtable.c if_cscope.c
10. if_xcmdsrv.c json.c list.c main.c mark.c
11. memfile.c memline.c menu.c message.c misc1.c
12. misc2.c move.c mbyte.c normal.c ops.c
13. option.c os_unix.c auto/pathdef.c popupmnu.c pty.c
14. quickfix.c regexp.c screen.c search.c sha256.c
15. spell.c spellfile.c syntax.c tag.c term.c
16. terminal.c ui.c undo.c userfunc.c version.c
17. window.c
18.)
19.
20. target_include_directories(basic_sources
21. PRIVATE
22. ${CMAKE_CURRENT_LIST_DIR}/proto
23. ${CMAKE_CURRENT_LIST_DIR}
24. ${CMAKE_CURRENT_BINARY_DIR}
25.)
26.
27. target_compile_definitions(basic_sources
```

```

28. PRIVATE
29. "HAVE_CONFIG_H"
30.)
31.
32. target_link_libraries(vim
33. PUBLIC
34. basic_sources
35.)

```

然后，定义一些可选源码文件的目标：

```

1. add_library(extra_sources "")
2.
3. if(ENABLE_NETBEANS)
4. target_sources(extra_sources
5. PRIVATE
6. netbeans.c
7.)
8. endif()
9.
10. if(ENABLE_CHANNEL)
11. target_sources(extra_sources
12. PRIVATE
13. channel.c
14.)
15. endif()
16.
17. target_include_directories(extra_sources
18. PUBLIC
19. ${CMAKE_CURRENT_LIST_DIR}/proto
20. ${CMAKE_CURRENT_BINARY_DIR}
21.)
22.
23. target_compile_definitions(extra_sources
24. PRIVATE
25. "HAVE_CONFIG_H"
26.)
27.
28. target_link_libraries(vim
29. PUBLIC
30. extra_sources
31.)

```

使用以下代码，对连接 `src/libvterm/` 子目录进行选择：

```
1. if(ENABLE_TERMINAL)
2. add_subdirectory(libvterm)
3.
4. target_link_libraries(vim
5. PUBLIC
6. libvterm
7.)
8. endif()
```

对应的 `src/libvterm/CMakeLists.txt` 包含以下内容：

```
1. add_library(libvterm "")
2.
3. target_sources(libvterm
4. PRIVATE
5. src/encoding.c
6. src/keyboard.c
7. src/mouse.c
8. src/parser.c
9. src/pen.c
10. src/screen.c
11. src/state.c
12. src/unicode.c
13. src/vterm.c
14.)
15.
16. target_include_directories(libvterm
17. PUBLIC
18. ${CMAKE_CURRENT_LIST_DIR}/include
19.)
20.
21. target_compile_definitions(libvterm
22. PRIVATE
23. "HAVE_CONFIG_H"
24. "INLINE="
25. "VSNPRINTF=vim_vsnprintf"
26. "IS_COMBINING_FUNCTION=utf_iscomposing_uint"
27. "WCWIDTH_FUNCTION=utf_uint2cells"
28.)
```

我们已经从 `build.log` 中获取了编译信息。树结构的优点是，目标的定义靠近源的位置。如果我们决定重构代码并重命名或移动目录，描述目标的CMake文件就会随着源文件一起移动。

我们的示例代码还没有配置(除非在成功的Autotools构建之后尝试配置)，现在来试试：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4.
5. -- The C compiler identification is GNU 8.2.0
6. -- Check for working C compiler: /usr/bin/cc
7. -- Check for working C compiler: /usr/bin/cc -- works
8. -- Detecting C compiler ABI info
9. -- Detecting C compiler ABI info - done
10. -- Detecting C compile features
11. -- Detecting C compile features - done
12. -- Configuring done
13. CMake Error at src/CMakeLists.txt:12 (add_library):
14. Cannot find source file:
15. auto/pathdef.c
16. Tried extensions .c .C .c++ .cc .cpp .cxx .cu .m .M .mm .h .hh .h++ .hm
17. .hpp .hxx .in .txx
```

这里需要生成 `auto/pathdef.c` (和其他文件)，我们将在下一节中考虑这些文件。

## 15.2 生成文件并编写平台检查

对于Vim示例，我们需要在配置时生成三个文

件，`src/auto/pathdef.c`、`src/auto/config.h` 和 `src/auto/osdef.h`：

- `pathdef.c`: 记录安装路径、编译/链接标志、当前用户和主机名
- `config.h`: 编译系统的环境
- `osdef.h`: 由 `src/osdef.sh` 生成的文件

这种情况相当普遍。需要CMake配置文件，配置时执行一个脚本，执行许多平台检查命令，来生成 `config.h`。特别是，对于那些可移植的项目，平台检查非常普遍。

在原始目录树中，文件在 `src` 文件夹下生成。而我们将使用不同的方法：这些文件会生成在 `build` 目录中。这样做的原因是生成的文件通常依赖于所选择的选项、编译器或构建类型，我们希望保持同一个源，可以适配多个构建。要在 `build` 目录中启用生成，我们必须对生成文件的脚本进行改动。

## 构造文件

我们将把与生成文件相关的函数集中放在 `src/autogenerate.cmake` 中。在定义可执行目标之前，在 `src/CMakeLists.txt` 中调用这些函数：

```

1. # generate config.h, pathdef.c, and osdef.h
2. include(autogenerate.cmake)
3. generate_config_h()
4. generate_pathdef_c()
5. generate_osdef_h()
6.
7. add_executable(vim
8. main.c
9.)
10.
11. # ...

```

`src/autogenerate.cmake` 中包含了其他检测头文件、函数和库等几个函数：

```

1. include(CheckTypeSize)
2. include(CheckFunctionExists)
3. include(CheckIncludeFiles)
4. include(CheckLibraryExists)

```

```

5. include(CheckCSourceCompiles)
6.
7. function(generate_config_h)
8. # ... to be written
9. endfunction()
10.
11. function(generate_pathdef_c)
12. # ... to be written
13. endfunction()
14.
15. function(generate_osdef_h)
16. # ... to be written
17. endfunction()

```

我们选择了一些用于生成文件的函数，而不是用宏或“裸”CMake代码。在前几章中讨论过的，这是避免了一些问题：

- 避免多次生成文件，以防多次包含模块。我们可以使用一个包含保护来防止意外地多次运行代码。
- 保证了对函数中变量范围的完全控制。这避免了这些定义溢出，从而出现变量污染的情况。

## 根据系统配置预处理宏定义

`config.h` 文件以 `src/config.h.in` 为目标所生成的，其中包含根据系统功能配置的预处理标志：

```

1. /* Define if we have EBCDIC code */
2. #undef EBCDIC
3.
4. /* Define unless no X support found */
5. #undef HAVE_X11
6.
7. /* Define when terminfo support found */
8. #undef TERMINFO
9.
10. /* Define when termcap.h contains ospeed */
11.
12. #undef HAVE_OSPEED
13. /* ... */

```

生成的 `src/config.h` 示例类似如下情况(定义可以根据环境的不同而不同)：

```

1. /* Define if we have EBCDIC code */
2. /* #undef EBCDIC */
3.
4. /* Define unless no X support found */
5. #define HAVE_X11 1
6.
7. /* Define when terminfo support found */
8. #define TERMINFO 1
9.
10. /* Define when termcap.h contains ospeed */
11.
12. /* #undef HAVE_OSPEED */
13. /* ... */

```

这个页面是一个很好的平台检查示例：

<https://gitlab.kitware.com/cmake/community/wikis/doc/tutorials/How-To-Write-Platform-Checks>

在 `src/configure.ac` 中，我们可以检查需要执行哪些平台检查，从而来设置相应的预处理定义。

我们将使

用 `#cmakedefine` ([https://cmake.org/cmake/help/v3.5/command/configure\\_file.html?highlight=cmakedefine](https://cmake.org/cmake/help/v3.5/command/configure_file.html?highlight=cmakedefine)) 为了确保不破坏现有的Autotools构建，我们将复制 `config.h.in` 为 `config.h.cmake.in`，并将所有 `#undef SOME_DEFINITION` 更改为 `#cmakedefine SOME_DEFINITION @SOME_DEFINITION@`。

在 `generate_config_h` 函数中，先定义两个变量：

```

1. set(TERMINFO 1)
2. set(UNIX 1)
3.
4. # this is hardcoded to keep the discussion in the book chapter
5. # which describes the migration to CMake simpler
6. set(TIME_WITH_SYS_TIME 1)
7. set(RETSIGTYPE void)
8. set(SIGRETURN return)
9.
10. find_package(X11)
11. set(HAVE_X11 ${X11_FOUND})

```

然后，我们执行几个类型检查：

```

1. check_type_size("int" VIM_SIZEOF_INT)
2. check_type_size("long" VIM_SIZEOF_LONG)
3. check_type_size("time_t" SIZEOF_TIME_T)
4. check_type_size("off_t" SIZEOF_OFF_T)

```

然后，我们对函数进行循环，检查系统是否能够解析：

```

1. foreach(
2. _function IN ITEMS
3. fchdir fchmod fsync getcwd getpseudotty
4. getpwent getpwnam getpwuid getrlimit gettimeofday getwd lstat
5. memset mkdtemp nanosleep opendir putenv qsort readlink select setenv
6. getpgid setpgid setsid sigaltstack sigstack sigset sigsetjmp sigaction
7. sigprocmask sigvec strcasecmp strerror strftime stricmp strncasecmp
8. strnicmp strpbrk strtol towlower towupper iswupper
9. usleep utime utimes mblen ftruncate
10.)
11.
12. string(TOUPPER "${_function}" _function_uppercase)
13. check_function_exists(${_function} HAVE_${_function_uppercase})
14. endforeach()

```

验证库是否包含特定函数：

```

1. check_library_exists(tinfo tgetent "" HAVE_TGETENT)
2.
3. if(NOT HAVE_TGETENT)
 message(FATAL_ERROR "Could not find the tgetent() function. You need to
4. install a terminal library; for example ncurses.")
5. endif()

```

然后，我们循环头文件，检查它们是否可用：

```

1. foreach(
2. _header IN ITEMS
3. setjmp.h dirent.h
4. stdint.h stdlib.h string.h
5. sys/select.h sys/utsname.h termcap.h fcntl.h
6. sgtty.h sys/ioctl.h sys/time.h sys/types.h
7. termio.h iconv.h inttypes.h langinfo.h math.h
8. unistd.h stropts.h errno.h sys/resource.h

```

```

9. sys/systeminfo.h locale.h sys/stream.h termios.h
10. libc.h sys/statfs.h poll.h sys/poll.h pwd.h
11. utime.h sys/param.h libintl.h libgen.h
12. util/debug.h util/msg18n.h frame.h sys/acl.h
13. sys/access.h sys/sysinfo.h wchar.h wctype.h
14.)
15.
16. string(TOUPPER "${_header}" _header_uppercase)
17. string(REPLACE "/" "_" _header_normalized "${_header_uppercase}")
18. string(REPLACE "." "_" _header_normalized "${_header_normalized}")
19. check_include_files(${_header} HAVE_${_header_normalized})
20. endforeach()

```

然后，我们将CMake选项从转换为预处理定义：

```

1. string(TOUPPER "${FEATURES}" _features_upper)
2. set(FEAT_${_features_upper} 1)
3.
4. set(FEAT_NETBEANS_INTG ${ENABLE_NETBEANS})
5. set(FEAT_JOB_CHANNEL ${ENABLE_CHANNEL})
6. set(FEAT_TERMINAL ${ENABLE_TERMINAL})

```

最后，我们检查是否能够编译一个特定的代码片段：

```

1. check_c_source_compiles(
2. "
3. #include <sys/types.h>
4. #include <sys/stat.h>
5.
6. int
7. main ()
8. {
9. struct stat st;
10. int n;
11. stat("/", &st);
12. n = (int)st.st_blksize;
13. ;
14. return 0;
15. }
16. "
17. HAVE_ST_BLKSIZE
18.)

```

然后，使用定义的变量配置 `src/config.h.cmake.in` 生成 `config.h`，其中包含 `generate_config_h` 函数：

```

1. configure_file(
2. ${CMAKE_CURRENT_LIST_DIR}/config.h.cmake.in
3. ${CMAKE_CURRENT_BINARY_DIR}/auto/config.h
4. @ONLY
5.)

```

## 使用路径和编译器标志配置文件

从 `src/pathdef.c.in` 生成 `pathdef.c`：

```

1. #include "vim.h"
2.
3. char_u *default_vim_dir = (char_u *)"@_default_vim_dir@";
4. char_u *default_vimruntime_dir = (char_u *)"@_default_vimruntime_dir@";
5. char_u *all_cflags = (char_u *)"@_all_cflags@";
6. char_u *all_lflags = (char_u *)"@_all_lflags@";
7. char_u *compiled_user = (char_u *)"@_compiled_user@";
8. char_u *compiled_sys = (char_u *)"@_compiled_sys@";

```

`generate_pathdef_c` 函数在 `src/pathdef.c.in` 进行配置。为了简单起见，我们省略了链接标志：

```

1. function(generate_pathdef_c)
2. set(_default_vim_dir ${CMAKE_INSTALL_PREFIX})
3. set(_default_vimruntime_dir ${_default_vim_dir})
4.
5. set(_all_cflags "${CMAKE_C_COMPILER} ${CMAKE_C_FLAGS}")
6. if(CMAKE_BUILD_TYPE STREQUAL "Release")
7. set(_all_cflags "${_all_cflags} ${CMAKE_C_FLAGS_RELEASE}")
8. else()
9. set(_all_cflags "${_all_cflags} ${CMAKE_C_FLAGS_DEBUG}")
10. endif()
11.
12. # it would require a bit more work and execute commands at build time
13. # to get the link line into the binary
14. set(_all_lflags "undefined")
15.

```

```

16. if(WIN32)
17. set(_compiled_user $ENV{USERNAME})
18. else()
19. set(_compiled_user $ENV{USER})
20. endif()
21.
22. cmake_host_system_information(RESULT _compiled_sys QUERY HOSTNAME)
23.
24. configure_file(
25. ${CMAKE_CURRENT_LIST_DIR}/pathdef.c.in
26. ${CMAKE_CURRENT_BINARY_DIR}/auto/pathdef.c
27. @ONLY
28.)
29. endfunction()

```

## 配置时执行shell脚本

最后，我们使用以下函数生成 `osdef.h`：

```

1. function(generate_osdef_h)
2. find_program(BASH_EXECUTABLE bash)
3.
4. execute_process(
5. COMMAND
6. ${BASH_EXECUTABLE} osdef.sh ${CMAKE_CURRENT_BINARY_DIR}
7. WORKING_DIRECTORY
8. ${CMAKE_CURRENT_LIST_DIR}
9.)
10. endfunction()

```

为了在  `${CMAKE_CURRENT_BINARY_DIR}/src/auto` 而不是 `src/auto` 中生成 `osdef.h`，我们必须调整 `osdef.sh` 以接受  `${CMAKE_CURRENT_BINARY_DIR}` 作为命令行参数。

`osdef.sh` 中，我们会检查是否给定了这个参数：

```

1. if [$# -eq 0]
2. then
3. # there are no arguments
4. # assume the target directory is current directory
5. target_directory=$PWD
6. else

```

```
7. # target directory is provided as argument
8. target_directory=$1
9. fi
```

然后，生成  `${target_directory}/auto/osdef.h`。为此，我们还必须在 `osdef.sh` 中调整以下行：

```
1. $CC -I. -I$srcdir -
2. I${target_directory} -E osdef0.c >osdef0.cc
```

## 15.3 检测所需的链接和依赖关系

现在已经生成了所有文件，让我们重新构建。我们应该能够配置和编译源代码，不过不能链接：

```

1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
4. $ cmake --build .
5.
6. ...
7. Scanning dependencies of target vim
8. [98%] Building C object src/CMakeFiles/vim.dir/main.c.o
9. [100%] Linking C executable ../bin/vim
10. ../../lib64/libbasic_sources.a(term.c.o): In function `set_shellsize.part.12':
11. term.c:(.text+0x2bd): undefined reference to `tputs'
12. ../../lib64/libbasic_sources.a(term.c.o): In function `getlinecol':
13. term.c:(.text+0x902): undefined reference to `tgetent'
14. term.c:(.text+0x915): undefined reference to `tgetent'
15. term.c:(.text+0x935): undefined reference to `tgetnum'
16. term.c:(.text+0x948): undefined reference to `tgetnum'
17. ... many other undefined references ...

```

同样，可以从Autotools编译中获取日志文件，特别是链接行，通过在 `src/CMakeLists.txt` 中添加以下代码来解决缺少的依赖关系：

```

1. # find X11 and link to it
2. find_package(X11 REQUIRED)
3. if(X11_FOUND)
4. target_link_libraries(vim
5. PUBLIC
6. ${X11_LIBRARIES})
7.)
8. endif()
9.
10. # a couple of more system libraries that the code requires
11. foreach(_library IN ITEMS Xt SM m tinfo acl gpm dl)
12. find_library(${_library}_found ${_library} REQUIRED)
13. if(${_library}_found)
14. target_link_libraries(vim
15. PUBLIC

```

```
16. ${_library}
17.)
18. endif()
19. endforeach()
```

我们可以添加一个库的依赖目标，并且不需要构建，以及不需要将库目标放在一个列表变量中，否则将破坏CMake代码的自变量，特别是对于较大的项目而言。

修改之后，编译和链接：

```
1. $ cmake --build .
2.
3. ...
4. Scanning dependencies of target vim
5. [98%] Building C object src/CMakeFiles/vim.dir/main.c.o
6. [100%] Linking C executable ../../bin/vim
7. [100%] Built target vim
```

现在，我们可以执行编译后的二进制文件，我们新编译的Vim就可使用了！

## 15.4 复制编译标志

现在，让我们尝试调整编译器标志来进行引用构建。

### 定义编译器标志

目前为止，我们还没有定义任何自定义编译器标志，参考Autotools构建中，代码是使用的编译标志有 `-g -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=1 -O2`，这些标示都是GNU C编译器可以识别的。

我们的第一个定义如下：

```
1. if(CMAKE_C_COMPILER_ID MATCHES GNU)
 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -U_FORTIFY_SOURCE -
2. D_FORTIFY_SOURCE=1 -O2")
3. endif()
```

并且，在生成源文件之前，我们将把这段代码放在 `src/CMakeLists.txt` 的顶部(因为 `pathdef.c` 有使用到  `${CMAKE_C_FLAGS}` )：

```
1. # <- we will define flags right here
2. include(autogenerate.cmake)
3. generate_config_h()
4. generate_pathdef_c()
5. generate_osdef_h()
```

编译器标志定义的一个小修改是将 `-O2` 定义为Release配置标志，并关闭Debug的配置：

```
1. if(CMAKE_C_COMPILER_ID MATCHES GNU)
2. set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -U_FORTIFY_SOURCE
3. -D_FORTIFY_SOURCE=1")
4. set(CMAKE_C_FLAGS_RELEASE "-O2")
5. set(CMAKE_C_FLAGS_DEBUG "-O0")
6. endif()
```

请使用 `make VERBOSE=1` 验证，构建是否使用了预期的标志。

### 编译器标志的作用域

在这个特殊的示例项目中，所有源文件都使用相同的编译标志。对于其他项目，我们可能不希望像上面

那样全局定义编译标志，而是使用 `target_compile_options` 为每个目标分别定义编译标志。这样做好处是更灵活、范围更小。在我们的例子中，这能减少不必要的代码复制。

## 15.5 移植测试

现在，来讨论如何将测试从引用构建移植到CMake。

### 准备工作

如果移植的项目包含测试目标，或任何形式的自动化测试，以及测试脚本。第一步，运行传统的测试步骤，并记录所使用的命令。对于Vim项目，可以从 `src/testdir/Makefile` 开始。

在 `src/testdir/Makefile` 和测试脚本中的一些对于测试的定义，我们将

在 `src/testdir/CMakeLists.txt` 中进行相应的定义。所以，我们必须

在 `src/CMakeLists.txt` 中引用它：

1. `add_subdirectory(testdir)`

处理 `src/CMakeLists.txt` 之前，我们还应该在主 `CMakeLists.txt` 中启用测试：

1. `# enable the test target`
2. `enable_testing()`
- 3.
4. `# process src/CMakeLists.txt in its own scope`
5. `add_subdirectory(src)`

目前为止，使用 `add_test` 填充 `src/testdir/CMakeLists.txt` 之前，测试目标为空。

在 `add_test` 中指定要运行的测试名称和命令。该命令可以用任何语言编写。CMake的关键部分是，如果测试成功，脚本返回零；如果测试失败，脚本返回非零。对于Vim，我们需要多步骤测试，这将在下一节中讨论。

### 实现多步测试

在 `src/testdir/Makefile` 的目标表明，Vim代码运行测试多步测试：

1. Vim脚本可执行测试流程，产生一个输出文件
2. 输出文件是与参考文件进行比，，如果这些文件相同，测试成功
3. 删除临时文件

由于 `add_test` 只能执行一个命令，因此无法以可移植的方式将其放到单个 `add_test` 中。一种解决方案是在Python脚本中定义测试步骤，并使用一些参数执行Python脚本。这里提供的另一种选择，也是跨平台的，在单独的CMake脚本中定义测试步骤，并使用 `add_test` 执行这个脚本。我们将

在 `src/testdir/test.cmake` 中定义测试步骤：

```

1. function(execute_test _vim_executable _working_dir _test_script)
2. # generates test.out
3. execute_process(
4. COMMAND ${_vim_executable} -f -u unix.vim -U NONE --noplugin --not-a-term -
5. s dotest.in ${_test_script}.in
6. WORKING_DIRECTORY ${_working_dir}
7.)
8.
9. # compares test*.ok and test.out
10. execute_process(
11. COMMAND ${CMAKE_COMMAND} -E compare_files ${_test_script}.ok test.out
12. WORKING_DIRECTORY ${_working_dir}
13. RESULT_VARIABLE files_differ
14. OUTPUT_QUIET
15. ERROR_QUIET
16.)
17.
18. # removes leftovers
19. file(REMOVE ${_working_dir}/dotest)
20.
21. # we let the test fail if the files differ
22. if(files_differ)
23. message(SEND_ERROR "test ${_test_script} failed")
24. endif()
25.
26. execute_test(${VIM_EXECUTABLE} ${WORKING_DIR} ${TEST_SCRIPT})

```

同样，我们选择函数而不是宏，为的是使得变量不会超出函数作用域。它将调用 `execute_test` 函数，处理这个脚本。但是，我们必须确保  `${VIM_EXECUTABLE}` 、  `${WORKING_DIR}` 和  `${TEST_SCRIPT}` 是在外部定义。`src/testdir/CMakeLists.txt` 中定义：

```

1. add_test(
2. NAME
3. test1
4. COMMAND
5. ${CMAKE_COMMAND} -D VIM_EXECUTABLE=${<TARGET_FILE:vim>}
6. -D WORKING_DIR=${CMAKE_CURRENT_LIST_DIR}
7. -D TEST_SCRIPT=test1
8. -P ${CMAKE_CURRENT_LIST_DIR}/test.cmake
9. WORKING_DIRECTORY

```

```
10. ${PROJECT_BINARY_DIR}
11.)
```

Vim项目有很多测试，但是在这个例子中，我们只移植了一个(test1)。

## 测试建议

---

对于移植测试，我们可以给出至少两个建议。

1. 要确保测试并不总是报告成功，如果破坏了代码或修改了验证数据，请验证测试是否失败。
2. 添加测试的成本估算，以便在并行运行时，首先启动较长的测试，以最小化总测试时间。

## 15.6 移植安装目标

现在可以配置、编译、链接和测试代码，但是没有测试安装目标。我们将在本节中添加这个目标。

Autotools的构建和安装方式：

```
1. $./configure --prefix=/some/install/path
2. $ make
3. $ make install
```

以下是CMake的方式：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake -D CMAKE_INSTALL_PREFIX=/some/install/path ..
4. $ cmake --build .
5. $ cmake --build . --target install
```

要添加安装目标，需要在 `src/CMakeLists.txt` 中添加以下代码：

```
1. install(
2. TARGETS
3. vim
4. RUNTIME DESTINATION
5. ${CMAKE_INSTALL_BINDIR}
6.)
```

本例中，只安装了可执行文件。Vim项目需要安装大量文件(符号链接和文档文件)，为了使本节易于理解，我们就没有迁移示例中所有的安装目标。对于自己的项目而言，应该验证安装步骤的结果是否匹配之前构建框架的安装目标。

## 15.7 进一步迁移的措施

成功地移植到CMake之后，下一步应该本地化目标和变量的范围：考虑将选项、目标和变量移到更靠近使用和修改它们的地方。避免全局变量，因为它们将按CMake命令顺序进行创建，而这个顺序可能不明显，从而会导致CMake代码变得混乱。强制分离变量范围的一种方法是将较大的项目划分为CMake项目，这些项目使用超构建块组成。从而，可考虑将大型 `CMakeLists.txt` 文件分割成更小的模块。

接下来的步骤，可以是在其他平台和操作系统上进行配置和编译，以便增强CMake代码的鲁棒性，使其更具可移植性。

最后，将项目迁移到新的构建框架时，开发人员社区也需要去适应。为了帮助您的同事进行培训、文档编制和代码评审。将代码移植到CMake中最困难的部分，可能是改变相关人员的使用习惯。

## 15.8 项目转换为CMake的常见问题

我们总结一下，在这一章中所学到的知识。

### 代码修改总结

在本章中，讨论了如何将项目移植到CMake进行构建。我们以Vim项目为例，添加了以下文件：

```

1. .
2. └── CMakeLists.txt
3. └── src
4. ├── autogenerate.cmake
5. ├── CMakeLists.txt
6. ├── config.h.cmake.in
7. ├── libvterm
8. │ └── CMakeLists.txt
9. ├── pathdef.c.in
10. └── testdir
11. ├── CMakeLists.txt
12. └── test.cmake

```

可以在线查看修改：<https://github.com/dev-cafe/vim/compare/b476cb7...cmakesupport>

为了简单起见，我们省略了许多选项和调整，并将重点放在最重要的步骤上。

### 常见问题

在结束讨论之前，我们想指出一些迁移到CMake时常见的问题。

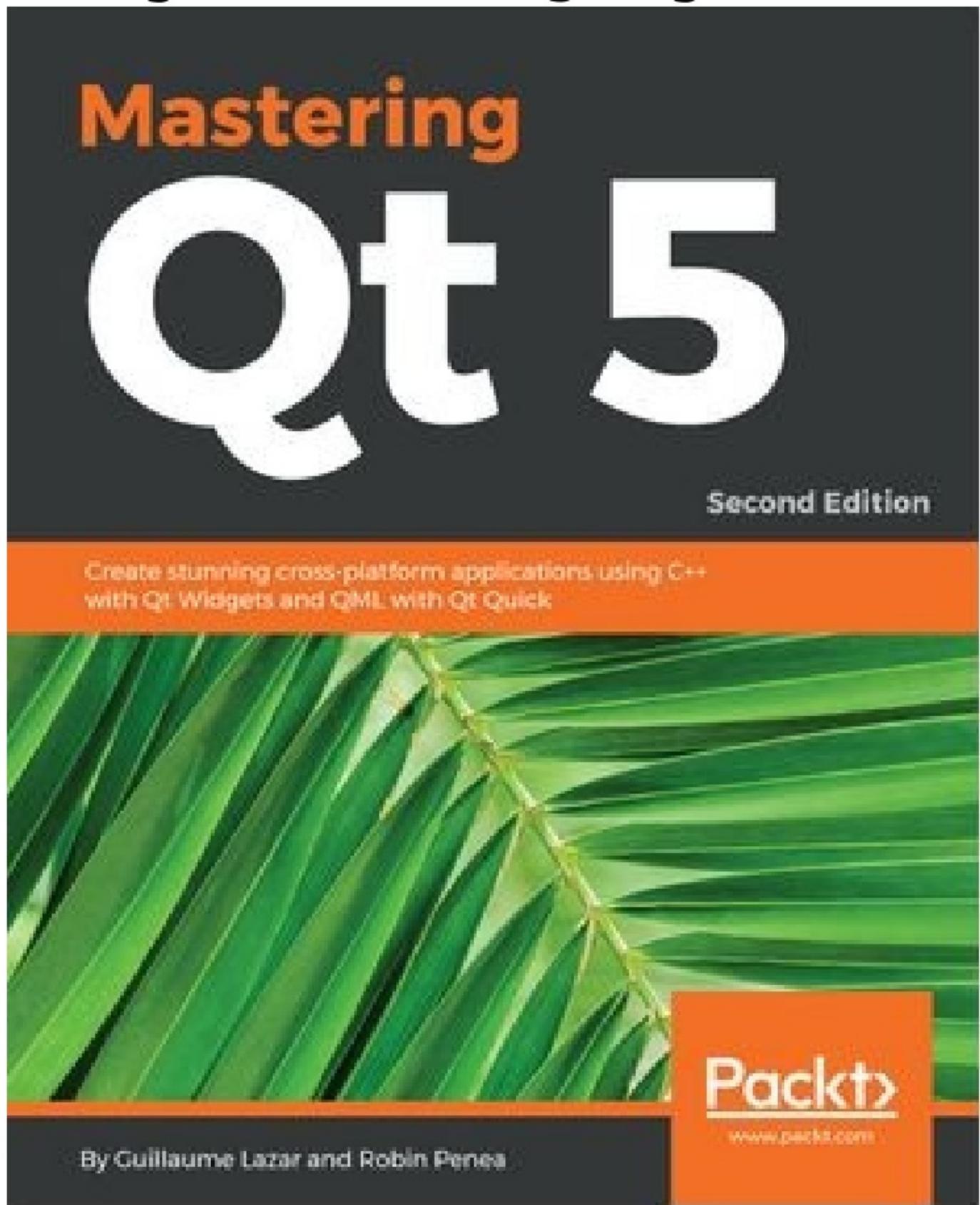
- 全局变量代码异味：这点适用于任何编程语言，CMake也不例外。跨CMake文件的变量，特别是从子到父 `CMakeLists.txt` 文件的“向上”传递的变量，这是明显的“异味代码”。通常，会有一种更好的方法来传输依赖关系。理想情况下，依赖项应该通过目标导入。与其将库列表组装成一个变量并在文件之间携带该变量，不如逐个链接到定义库的地方。不是将源文件组装成变量，而是使用 `target_sources` 添加源文件。当链接到库时，在可用时使用导入的目标，而不是变量。
- 最小化顺序的影响：CMake不是一种声明性语言，但是也不应该使用命令式范式进行处理。执行严格命令的代码往往是脆弱的，这也与变量有关(见上一段)。一些语句和模块的顺序是必要的，但是为了实现健壮的CMake框架，我们应该避免不必要的顺序强制。应该多使用 `target_sources`、`target_compile_definition`、`target_include_directory` 和 `tar`

`get_link_libraries`。避免使用全局范围语句，如 `add_definition`、`include_directory` 和 `link_libraries`，从而避免定义全局编译标志。如果可能，为每个目标定义编译标志。

- 不在**build**目录之外生成文件：强烈建议不要将生成的文件放在构建目录之外。原因是生成的文件通常依赖于所选择的选项、编译器或构建类型。如果写入原目录树，我们就放弃了用同一套源码维护多个构建的可能性，并且会使构建步骤的重现复杂化。
- 尽可能使用函数，而不是宏：它们的作用范围不同，功能范围也有限定。所有变量修改都需要显式标记，这也向读者展示了重新定义的变量。如果可以最好使用函数，必要时再使用宏。
- 避免**shell**命令：Shell可能不能移植到其他平台(如Windows)。可以使用CMake中的命令或函数。如果没有可用的CMake等效函数，请考虑调用Python脚本。
- **Fortran**中，注意后缀：需要预处理的Fortran源文件是大写的 `.F90` 后缀。无预处理的源文件应该以 `.f90` 为后缀。
- 避免显式路径：这条建议在定义目标和引用文件时都适用。当引用当前路径时，可使用 `CMAKE_CURRENT_LIST_DIR`。这样做好处是，当移动或重命名一个目录时，构建不会出问题。
- 不应该在函数调用中进行模块包含：将CMake代码模块化是一个很好的策略，但是包含模块不应该执行CMake代码。相反，将CMake代码封装到函数和宏中，并在包含模块之后显式地调用这些函数和宏。当意外地多次包含模块时，这条建议可以防止意外的副作用，并使执行CMake代码模块的操作更易读。

## 第16章 可能感兴趣的书

如果你喜欢本书，你可能会对Packt的其他书感兴趣：

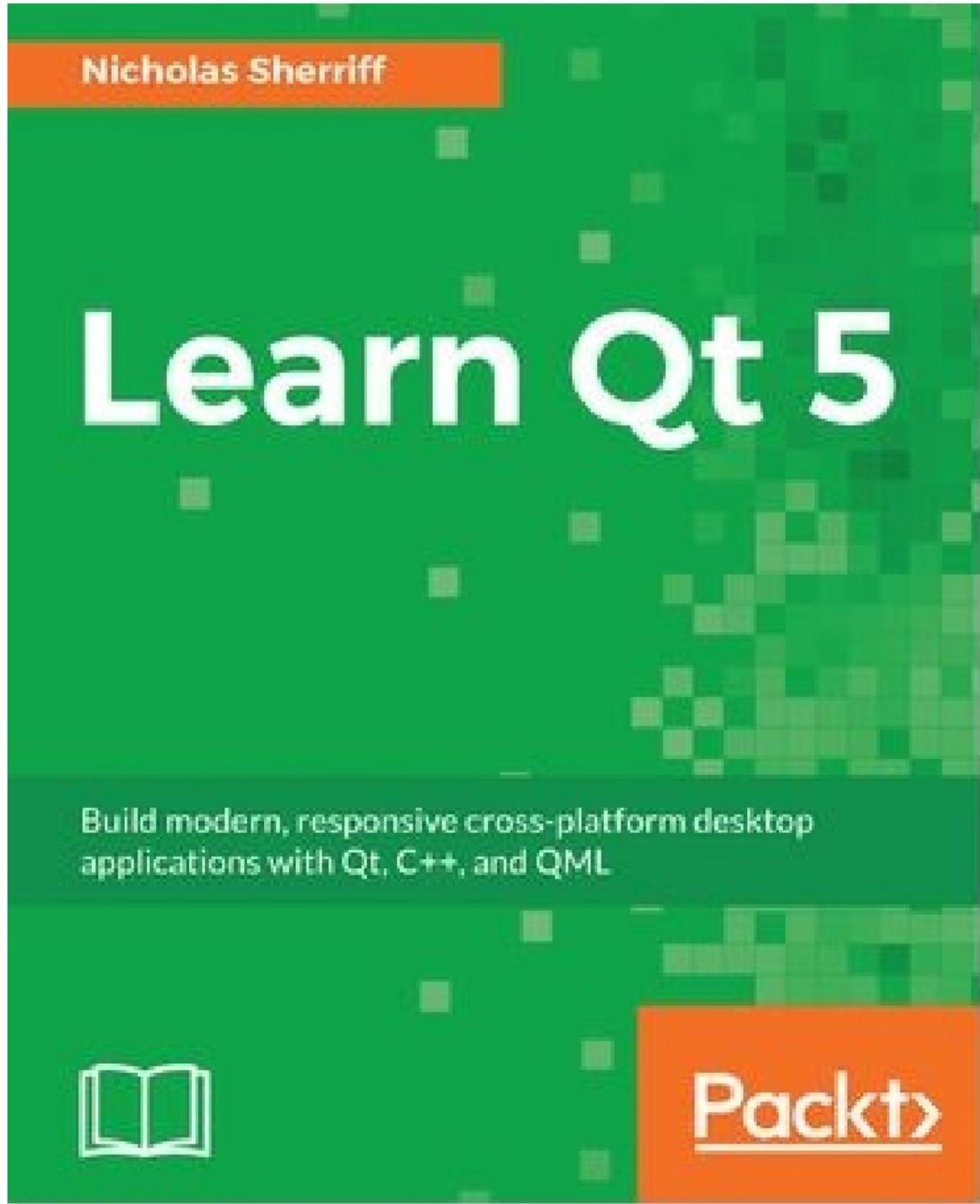


## **Mastering Qt 5 - Second Edition**

Guillaume Lazar, Robin Penea

ISBN: 978-1-78899-539-9

- 使用Qt Widgets和Qt Quick 2创建漂亮的UI
- 使用Qt框架开发功能强大的跨平台应用程序
- 使用Qt设计器设计GUI，并在其中构建用于UI预览的库
- 在C++中处理用户与Qt信号/插槽机制的交互
- 准备跨平台项目来托管第三方库
- 使用Qt动画框架来创造惊人的效果
- 使用Qt和嵌入式平台部署移动应用程序
- 使用Qt Gamepad与游戏平板交互



**Learn QT 5**

Nicholas Sherriff

ISBN: 978-1-78847-885-4

- 安装/配置Qt框架和Qt Creator IDE
- 使用QMake控制创建一个新的多项目解决方案
- 用QML实现一个丰富的用户界面
- 学习QtTest的基础知识，以及如何集成单元测试
- 构建自我感知的数据实体，这些数据实体和JSON进行互相转换
- 使用SQLite和CRUD管理数据
- 接触互联网并使用RSS
- 生成应用程序包以分发给其他用户

## 16.1 留下评论——让其他读者知道你的想法

---

请在你购买此书的网站上留下评论，与他人分享你对这本书的看法。如果你从亚马逊购买了这本书，请在亚马逊主页上给我们留下真实的评论。这一点至关重要，这样其他读者就可以看到，并根据您的意见来做出购买决定，我们也可以了解客户对我们的产品的看法，作者们也可以看到您对他们作品的反馈。这只会占用您几分钟的时间，但对其他读者、作者和Packt都很有价值。此致敬礼！