

# PROFESSIONAL **CMAKE**

A PRACTICAL GUIDE



---

CRAIG SCOTT

# 目录

Introduction	1.1
前言	1.2
致谢	1.3
第一部分：基础知识	1.4
第1章：介绍	1.4.1
第2章：创建项目	1.4.2
第3章：一个简单的项目	1.4.3
第4章：构建目标	1.4.4
第5章：变量	1.4.5
第6章：控制流	1.4.6
第7章：子目录	1.4.7
第8章：函数和宏	1.4.8
第9章：属性	1.4.9
第10章：表达式	1.4.10
第11章：模块	1.4.11
第12章：策略	1.4.12
第二部分：深度构建	1.5
第13章：构建类型	1.5.1
第14章：编译和链接	1.5.2
第15章：语言要求	1.5.3
第16章：目标类型	1.5.4
第17章：自定义任务	1.5.5
第18章：文件处理	1.5.6
第19章：指定版本	1.5.7
第20章：库	1.5.8
第21章：工具链与交叉编译	1.5.9
第22章：Apple专属特性	1.5.10
第三部分：常用功能	1.6
第23章：查找	1.6.1
第24章：测试	1.6.2
第25章：安装	1.6.3
第26章：打包	1.6.4
第27章：扩展	1.6.5
第28章：项目结构	1.6.6

# Professional CMake : A Practical Guide

- 作者: Craig Scott
- 译者: 陈晓伟
- Version: 1.0.3

翻译是译者用自己的思想，换一种语言，对原作者想法的重新阐释。鉴于我的学识所限，误解和错译在所难免。如果你能买到本书的原版，且有能力阅读英文，请直接去读原文。因为与之相较，我的译文可能根本不值得一读。

— 云风，程序员修炼之道第2版译者

## 本书概述

**CMake**是非常流行的工具，用于构建、测试和打包项目。因成熟性、特性多、平台支持性广和工具链灵活的特点，使得在开发社区中得以广泛使用。近年来，**3.x**系列带来了更多强大的功能，也产生了**现代CMake**这一术语。本书主要为开发人员提供最新、最实用的**CMake**指南，并且关注实践。作为**CMake**教程，遵循从基本概念到高级主题的逻辑顺序。对于有经验的读者，可以直接去看自己感兴趣的部分。

本书第一部分介绍了**CMake**的基本部分，为了让读者快速上手，以实践了解项目，然后在可管理和结构化的单元中，逐步建立对**CMake**概念的理解。这里不仅呈现了关键的概念，也让读者对如何有效地使用**CMake**，以及如何避免常见问题有所了解。

本书的中间部分更详细地介绍了各个关键部分，不过更侧重于任务。许多案例解决了开发人员经常面临的问题，而且这些章节覆盖了很多**3.x**中新加入或改进的特性。有些章节会更深入的讨论**CMake**，诸如如何构建库、管理跨平台版本，以及与**Apple**平台相关的技术，如包、框架和签名等。展示了开发人员如何最大限度地利用**CMake**的特点。并对一些常见但不推荐的方式进行介绍，解释为什么这些方式不推荐，应该如何避免。

本书的最后一部分，读者的关注点需要扩展到项目之外，从不同的角度来看待项目。管理项目间依赖关系是开发人员面临的困难之一，最后的章节将直接进行讨论。读者们可以了解依赖关系和将项目供他人使用的方法，并讨论如何准备各种格式的软件包，以及如何遵循平台约定。还有，对可测试性的深入理解，会让开发人员对其项目更有信心。这一章专注于项目组织，汇集了许多早期案例，演示了如何构建一个**CMake**项目来提高可维护性，使日常开发更加方便，缩短了新开发人员融入项目的时间。有一章会专门讨论构建性能，以及减少构建时间的各种工具和技术。最后一章会对**Qt**的支持进行介绍，会讨论**CMake**和**Qt**提供的特性，涵盖了 `moc`、`uic` 和 `rcc` 支持、翻译处理和**Qt**部署工具使用等主题。

## 本书相关

- 英文原版PDF: <https://b-ok.global/book/3591186/69f05f>
- 本书最新版本获取地址: <https://crascit.com/professional-cmake>

# 前言

几年前，**CMake**的公开资料少的可怜。官方参考文档对于愿意去探索的人来说非常有用，但无法作为一种渐进的、结构化地学习**CMake**的方式。一些维基和个人网站有一些有用的内容，也有许多(过时或有问题的)建议和例子。这意味着不了解**CMake**的新人很难了使用**CMake**。

那时，我一直在写博客，利用空闲时间做一些事情，并加深自己在软件开发方面的技术积累。我经常记录一些工作中与同事的交流，或在开发中出现的问题，我发现这对其他人也很帮助。然后，写书的想法就诞生了。花了近两年半的时间，就出现了你现在正在读的这本书。

这个过程中，有一个有趣的时刻，我现在回想起来有些好笑。一位同事抱怨说，他希望**CMake**拥有一个功能。对于这个功能我想了好几个月，直到有一天我决定自己去尝试添加这个功能，看看到底有多难。这个功能在测试结束后，现在成为了**CMake**正式版本的一部分。更重要的，是我在这个过程中所获得的经验。那些一起工作的研发人员、所使用的工具，以及过程的乐趣。从那时起，我更乐于深入地参与其中，并满足于志愿维护的角色。

# 致谢

当你在感谢所有那些为你的书的出版做出贡献的人时，才会意识到有多少人参与其中。没有他人的慷慨、耐心和关注，现在的所有就都不会存在，没有挑战、考验和返工，同样也不会成功。它依赖于那些善良的人(有时他们自己也没有察觉!)参与到这些活动中来，就像隐形的作者一样。

如果没有CMake公司及其员工的持续支持，CMake公司就不会像今天这样强大和充满活力。我想特别提一下Brad King，他是CMake项目的负责人，他对待CMake的新贡献者非常的包容和鼓励，会让人备受鼓舞。通过观察他与开发人员和用户互动的方式，他具有强大的领导力，并且非常尊重他人，我从他身上学到了很多东西。多年来CMake的许多贡献者的努力也值得赞扬，他们的努力往往是自愿的。

许多人审阅了这本书的案例，如果没有他们，技术上的准确性和可读性会受到影响。CMake的同事Gregor Jasny和Christian Pfeiffer在整个评审过程的贡献卓越，我非常感谢他们的建议和见解。也感谢Nils Gladitz的意见，特别是在如此短的时间内。我还要感谢我过去的同事马特·博尔格(Matt Bolger)和拉克兰·赫瑟顿(Lachlan Hetherton)，他们都提供了建设性的意见，并提供了一些令我耳目一新的意见。

特别值得一提的是我的同事Mike Wake。本书中的许多案例都在实际开发的项目上经过研究和测试。在如何从可用性和健壮性的角度改进方面，出现了错误的方向和技术讨论。他支持给这些事情以缓冲和鼓励，他愿意承受一些短期(有时不是很短期)的痛苦，这对提炼许多技术到实践的例子起到了重要作用。我也非常感谢在自己压力过大和非常疲惫的时期，他能在恰当的时间及时地提出建议和鼓励。

我也要感谢在Asciidoctor背后的人们，感谢他们用软件编译编辑了这本书。尽管这些方式的规模、复杂性和技术性并不是很高，但我一直惊讶于可使自助出版不仅成为一个可行的选择，而且是一种令人愉快的体验。现在从作者到读者的距离比几年前要短得多，也简单得多，使得更多的潜在作者能够接触到它，也给可以给社区带来了好处。感谢这个很棒的工具！

这本书的封面和网站上的一些辅助材料是我对图形设计的理解。对于我的朋友兼设计师V来说，你设法理解我随意、不连贯和相互冲突的想法。我不明白你是怎么做到的，但我喜欢现在的结果！

每本书的致谢部分，作者都会向家人和配偶致谢。他们需要付出巨大的理解和牺牲来忍受你的疲劳，你不能做很多普通人会做的事情，你决定花更多的时间在一个项目上而不是他们。我真的无法表达我对我妻子的深深的感激之情，因为在这本书的写作和出版的整个过程中，她是如此的支持和耐心。我是一个非常幸运的人。

# 第一部分：基础知识

了解任何工具的基本功能和使用方法之前，使用这个工具，很可能会达不到预期目的。一个人所有的时间学习关于某件事的理论，而不亲自动手会得到相当无聊的经验，并经常导致过于理想化的理解。本书的第一部分按照**CMake**的特性和概念的逻辑进行推进，使读者能够立即进行实验，并在每一章中越做越好。其目标是逐步构建有效使用**CMake**所需的基础知识体系，并能够立即将这些知识付诸实践。

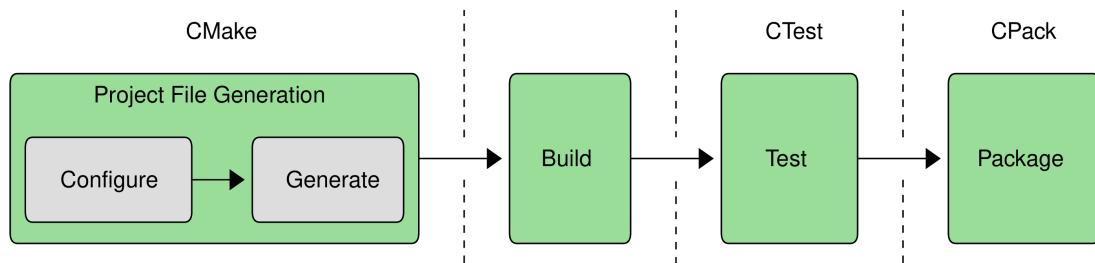
前几章的重点是建立可执行文件或库，给新手开发人员快速介绍**CMake**。接下来的章节将扩展这些知识，会演示如何最大限度地利用**CMake**所提供的内容。所介绍的技术是针对实际项目的使用，旨在建立良好的习惯，并教授适用于大项目和处理复杂场景的方法。

书的后半部分会依赖于第一部分的案例。已经使用**CMake**一段时间的人可能会发现这些主题会非常熟悉，不过这些案例中还包括从实际项目和与**CMake**社区的交互中得来的知识。即使是有经验的用户也应该发现，每一章末尾的总结还是非常有用的。

# 第1章：介绍

无论是经验丰富的开发人员，还是刚开始从事开发工作的菜鸟，对工具的使用都需要一个熟悉的过程，以便将项目的源代码转化为用户可以使用的东西。编译器、链接器、测试框架、打包系统等都保证了部署的质量、软件的健壮性。虽然有些平台有主流的IDE(例如：**Xcode**和**Visual Studio**)可以简化一些操作，但需要支持多个平台的项目不能总使用IDE。多个平台的支持增加了软件的复杂性，可能会影响到从工具集到功能的行为。所以开发人员在试图掌握全局的过程中，有时处于失控状态，这也是可以理解的。

有一些工具可以使这个过程更容易管理。**CMake**就是这样的工具，或者说**CMake**是一套工具，涵盖了从创建构建到生产，以及发布的一切操作。不仅涵盖了从头到尾的流程，还有广泛的平台、工具和语言支持。使用**CMake**时，了解它的世界观对使用非常有帮助。简单的说，**CMake**从开始到结束的过程看起来如下图所示：



第一阶段需要对项目进行描述，并生成特定于平台的项目文件，这些文件与开发人员选择的构建工具(例如**make**、**Xcode**、**Visual Studio**等)适配。这是**CMake**有名的设置阶段，**CMake**工具套件还包括**CTest**和**CPack**，分别用于管理测试和打包。从开始到结束的整个过程可以由**CMake**驱动，测试和打包步骤可以简单地作为构建中的附加目标，**CMake**可以调用构建工具。

开始使用**CMake**之前，开发人员需要确保**CMake**已经安装。有些平台会自带**CMake**(大多数Linux发行版都通过包管理器提供了**CMake**)，但是其版本通常相对过旧，建议开发人员使用最新的**CMake**版本。特别是在为**Apple**平台开发时，**Xcode**和**SDK**等工具更新的频率特别高，应用商店的需求会随着时间而变化。官方的**CMake**包可以下载并解压缩到目录中，从而不会影响任何系统范围的**CMake**安装。我们鼓励开发人员利用这一点，使用(相对稳定的)发行版**CMake**。

**CMake**提供了相应的[参考文档](#)，可以从**CMake**官方网站获取。参考文档对于查找各种命令、选项、关键字等非常有帮助，开发人员可以将其添加为书签，以便需要时快速参考。**CMake**用户邮件列表也是好建议来源，可用于在文档没有提供足够指导的情况下询问与**CMake**相关的问题。

# 第2章： 创建项目

如果没有构建系统，项目仅为文件的集合。**CMake**为构建项目制定了规则，首先需要有一个名为**CMakeLists.txt**的文件，该文件定义了构建什么、如何构建、运行哪些测试以及创建哪些包。该文件对整个项目进行了描述。**CMake**构建时，会将其解析为构建工具的项目文件。**CMakeLists.txt**是一个普通的文本文件，开发人员可以在文本编辑器或开发环境中对其进行编辑。这个文件的内容将在后面的章节中详细介绍。目前，知道**CMake**在设置和执行构建过程中要做的事情就足够了。

源目录和二进制目录概念是**CMake**的基础。源目录是**CMakeLists.txt**文件所在的文件夹，项目的源文件和构建所需的其他文件都组织在该位置下。源目录经常处于**Git**、**Subversion**或其他版本控制工具的控制之下。

二进制目录是生成构建内容的地方，通常称为构建目录。**CMake**通常使用“二进制目录”，但是在开发人员中，“构建目录”更常用。这本书倾向于后者，因为更直观。**CMake**所选择的构建工具(例如：**make**, **Visual Studio**等)，**CTest**和**CPack**都在构建目录中创建。可执行文件、库、测试输出和包都在构建目录中创建。**CMake**还在构建目录中创建一个**CMakeCache.txt**的特殊文件，存储各种信息，以便在后续运行时重用。开发人员通常不需要关心**CMakeCache.txt**文件，后面的章节将详细讨论此文件。构建工具的项目文件(如**Xcode**或**Visual Studio**项目文件、**makefile**文件等)也是在构建目录中创建的，这些并不需要版本控制。**CMakeLists.txt**文件是项目的描述，生成的项目文件应该视为构建输出。

开发人员处理项目时，需要确定构建目录相对于源目录的位置。有两种方式：源内构建和源外构建。

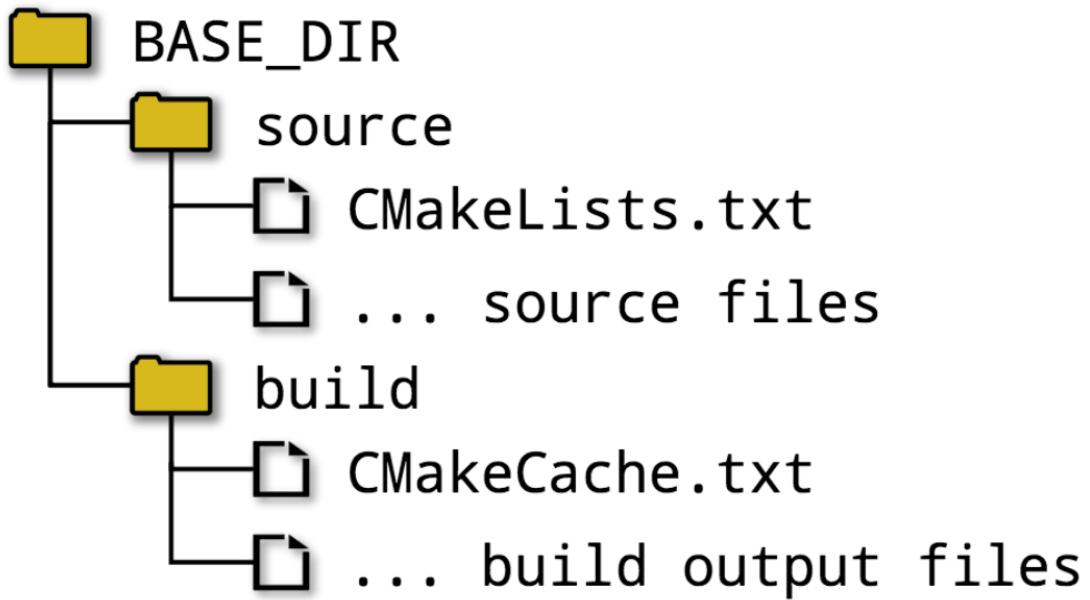
## 2.1. 源内构建

源目录和构建目录可以相同，但不推荐这样做，这种方式称为源内构建。菜鸟开发人员通常会使用这种方法，因为看起来简单。然而，内源构建的主要问题在于，构建输出与源文件混杂在一起。这种缺乏分离的方式导致目录与各种文件和目录混杂在一起，使得项目管理变得非常困难，并有着构建输出覆盖源文件的风险。这种方式还使版本控制系统的工作难度增加，因为有许多由构建创建的文件，源代码控制工具必须知道要忽略哪些文件，或者在提交时手动排除这些文件。源内构建的另一个缺点是，清除所有构建输出并使用干净的源重新开发非常困难。由于这些原因，不推荐开发人员在任何的情况下使用源内构建方式进行构建。

## 2.2. 源外构建

更可取的方式是让源目录和构建目录不同，这称为源外构建。这使得源文件和构建输出完全分离，从而避免了混合问题。源外构建还有一个优势，开发人员可以为同一个源目录创建多个构建目录，这就允许使用不同的选项设置构建项目，比如：**Debug**和**Release**版本等等。

本书将使用源外构建，并遵循源目录和构建目录位于同级目录下的模式。构建目录将称为**build**(或其他)。目录结构类似如下方式：



一些开发人员将构建目录作为源目录的子目录，虽然也是源外构建，但是仍然带有内源构建的一些缺点。除非有很好的理由这样构造，否则建议将构建目录完全置于源目录之外。

### 2.3. 生成项目文件

选择好目录结构后，开发人员运行CMake，读入CMakeLists.txt文件并在构建目录中创建项目文件。开发人员通过选择生成器来选择要创建的项目文件。CMake支持一系列生成器，下表列出了常用的生成器。

Category	Generator Examples	Multi-config
Visual Studio	Visual Studio 15 2017	Yes
	Visual Studio 14 2015	
	:	
	Xcode	
Xcode	Xcode	Yes
Ninja	Ninja	No
Makefiles	Unix Makefiles	No
	MSYS Makefiles	
	MinGW Makefiles	
	NMake Makefiles	

一些生成器支持多种配置(如Debug、Release等)，其允许开发者在不同的配置中进行选择，不必重新运行CMake。CMake很适合为Xcode和Visual Studio等IDE生成项目文件。对于不支持多配置的生成器，开发人员必须重新运行CMake在Debug、Release等之间切换构建方式。通常在与特定编译器(Qt Creator、KDevelop等)关联不那么紧密的IDE环境中，CMake都有很好的支持。

运行CMake最基本的方法是通过CMake命令行。调用它的最简单方法是将目录切换到build目录，并将生成器类型和源树位置的选项传递给CMake。例如：

```
mkdir build  
cd build  
cmake -G "Unix Makefiles" ../source
```

如果省略了-G选项，CMake将根据平台选择默认的生成器类型。对于所有的生成器类型，CMake将执行测试并询问系统，以确定如何建立项目文件。这包括验证编译器是否工作、确定支持的编译器特性集和其他各种任务。在CMake完成验证之前，将记录各种信息，如以下的信息：

```
-- Configuring done  
-- Generating done  
-- Build files have been written to: /some/path/build
```

上面展示了项目文件的创建时的两个步骤：配置和生成。在配置阶段，CMake读取CMakeLists.txt文件，并构建整个项目的内部表示文件。完成这些之后，生成阶段将创建项目文件。配置和生成之间的区别对于CMake使用来说并不重要，在后面的章节中，配置和生成的分离就会变得很重要。第10章生成器表达式中会有更详细的介绍。

CMake完成运行时，在构建目录中保存一个CMakeCache.txt文件。CMake用这个文件保存信息，当CMake再次运行时，可以重用第一次的信息，加速项目的生成。如后面章节所述，还允许在运行间保存开发人员选项。CMake还有一个GUI应用程序，cmake-gui，可以作为CMake命令行工具的替代。GUI应用程序的介绍会在第5章详细介绍。

## 2.4. 运行构建工具

此时，项目文件已经可用，开发人员可以按照习惯的方式使用相应的构建工具。构建目录将包含必要的项目文件，这些文件可以加载到IDE中，通过命令行工具等等。当然，CMake也可以“代表”构建工具进行构建：

```
cmake --build /some/path/build --config Debug --target MyApp
```

这种方式更适用于常用IDE的开发人员，--build选项指向CMake项目生成的构建目录。对于多个配置，--config选项指定要构建哪个配置，而单个配置生成器将忽略--config选项，而依赖于执行CMake项目生成时的信息。第13章会详细介绍构建配置。--target选项可以用来告诉构建工具要构建什么目标，如果省略该选项，将构建默认目标。

虽然开发人员通常会在开发中直接调用构建工具，但是通过cmake命令来调用构建工具，在自动化构建中会更有用。可以使用这种方式进行构建：

```
mkdir build  
cd build  
cmake -G "Unix Makefiles" ../source  
cmake --build . --config Release --target MyApp
```

如果开发人员希望测试不同的生成器，要做的就是更改**CMake**选项 `-G` 的参数，然后选择正确的构建工具。为了让 `cmake --build` 正常工作，构建工具甚至不需要在用户的PATH环境变量中(第一次调用`cmake`时可能需要配置上)。

## 2.5. 总结

即使是第一次使用**CMake**，也建议养成将构建目录与源目录完全分离的习惯。尽早体验这种方式的好处，为了使用相同的源目录设置两个或更多不同的构建，一个版本可以配置成**Debug**，另一个版本可以配置成**Release**。另一种选择是为不同的构建目录使用不同的项目生成器，比如：**Unix Makefiles**和**Xcode**。这有助于确定特定构建工具的依赖，或检查生成器类型之间的不同编译器设置。

项目的早期集中使用特定类型的生成器非常简单，特别是开发人员不习惯编写跨平台软件时。然而，项目通常会有超出其初始范围的趋势时，项目通常需要支持其他平台，因此需要支持不同的生成器。定期使用与常用生成器不同的生成器检查构建，可以避免在使用特定生成器时的问题。这样，项目可以很好地使用任何新的生成器。一种测试策略是，在每个平台上使用默认生成器类型，以及另一种类型构建项目。**Ninja**生成器是后者最佳的选择，因为它支持的平台最广泛，还能进行非常高效的构建。如果在为项目编写构建脚本，通过 `cmake --build` 调用构建工具，而非直接调用构建工具。无需对脚本进行修改，就可以轻松地在生成器之间进行切换。

# 第3章：一个简单的项目

所有的CMake项目都从源目录下的CMakeLists.txt文件开始，可以把这个文件看作CMake项目文件，其定义构建过程中所有的内容，从源码和构建目标，到测试、打包和自定义任务。可以简单到只有几行，也可以相当复杂，从其他目录中拉入更多的文件。CMakeLists.txt是文本文件，可以直接编辑。

与源码的相比，CMake有自己的语言，有许多程序员熟悉的东西，比如变量、函数、宏、条件、循环、注释等等。这些概念和特性将在后面介绍，现在我们的目标只是将简单的构建作为起点。下面是一个简单的CMakeLists.txt文件，会生成一个可执行文件。

```
cmake_minimum_required(VERSION 3.2)
project(MyApp)
add_executable(myExe main.cpp)
```

上面示例中的每一行都会执行一个CMake命令，CMake命令类似于函数调用，但不直接返回值(除了支持参数时，后面的章节将展示如何将值返回)。参数之间可以用空格分隔，也可以跨多行分隔：

```
add_executable(myExe
  main.cpp
  src1.cpp
  src2.cpp
)
```

命令名称也是不区分大小写的，所以一下形式的效果都相同：

```
add_executable(myExe main.cpp)
ADD_EXECUTABLE(myExe main.cpp)
Add_Executable(myExe main.cpp)
```

每个人书写的方式可能不相同，但更常见的是命令名都使用小写字母(对于内置命令，CMake文档也遵循这种惯例)。

## 3.1. 管理CMake版本

CMake也在不断更新和扩展，以增加对新工具、平台和特性的支持。CMake的开发人员会确保新版本的向后兼容性，所以当用户更新CMake时，项目应该可以如同以前一样构建。有时，需要更改CMake的行为，或在新版本中引入更为严格的检查和警告。CMake没有要求所有的项目立即处理这个问题，而是提供了策略机制，允许项目使用“和CMake版本X.Y.Z的行为一样”。这使得CMake可以修复bug并引入新特性，并且仍然保持以往版本的行为。

项目指定CMake版本行为的详细信息是使用 `cmake_minimum_required()` 命令。这应该是CMakeLists.txt文件的第一行，这样项目的需求就会放在在其他事情之前。这个命令做了两件事：

- 指定了项目所需的CMake的最低版本。如果CMakeLists.txt文件使用的CMake版本比指定的版本低，将立即停止构建，并出现错误报告。
- 强制设置将CMake行为匹配到对应版本。

如果CMakeLists.txt没有在任何其他命令之前调用 `cmake_minimum_required()`，CMake将发出警告。它需要知道如何为所有后续处理设置策略。对于大多数项目，就像它的名字那样，使用 `cmake_minimum_required()` 指定所需的最低CMake版本就足够了。它还暗示CMake应该与特定版本的行为相同。第12章中会更详细地讨论了策略

设置，并解释如何根据需要进行定制。

`cmake_minimum_required()` 命令很简单：

```
cmake_minimum_required(VERSION major.minor[.patch[.tweak]])
```

`VERSION` 关键字必须出现，提供的版本详细信息必须有 `major.minor`。大多数项目中，`patch` 和 `tweak` 没有必要，因为新特性通常只在 `minor` 版本更新中出现（这是3.0版本后的官方行为）。只有当修复某个错误时，项目才应该指定 `patch` 部分。`3.x` 的CMake 使用了 `tweak`，但项目不需要进行指定。

开发人员应该仔细考虑项目需要的最低CMake版本。版本3.2可能是所有新项目中最老的一个，因为它为现代CMake技术提供了相当完整的特性集。2.8.12的特性覆盖率降低了，缺少了许多有用的特性，但对较老的项目还是可用的。在此之前的版本缺乏实质性的特性，许多现代CMake技术还用不了。如果工作在移动平台，如iOS上，可能需要最近版本的CMake来支持最新的操作系统版本等等。

作为经验法则，选择最新CMake版本不会对构建项目产生重大的问题。最大的困难通常是需要支持旧平台的项目，其中系统提供的CMake版本可能相当旧。对于这种情况，开发人员应该考虑安装一个新版本。另一方面，如果项目本身是其他项目的依赖项，选择最新的CMake可能会成为障碍。这种情况下，可以使用一些新CMake版本的功能。使用老版本的主要缺点是，可能会弃用更多的警告，因为新版CMake会警告旧的行为，以鼓励项目更新。

## 3.2. project()

每个CMake项目都应该包含一个 `project()` 命令，它应该在 `cmake_minimum_required()` 之后出现。该命令常见的选项如下所示：

```
project(projectName  
[VERSION major[.minor[.patch[.tweak]]]]  
[LANGUAGES languageName ...]  
)
```

项目名称是必需的，只能是字母、数字、下划线(\_)和连字符(-)，字母和下划线比较常用。由于不允许使用空格，所以项目名称不必用引号括起来。这个名字用于项目的生成器（如Xcode和Visual Studio），也可以运用于各种项目的其他部分，比如默认打包名称和文档元数据，项目名称等等。`projectName` 是 `project()` 命令唯一强制性参数。

`VERSION` 只在CMake 3.0之后的版本支持。与 `projectName` 一样，CMake使用版本细节来填充变量并作为默认元数据，除此之外版本信息没有任何意义。尽管如此，定义项目版本是一个好习惯，以便项目的其他部分可以引用。第19章将深入讨论这个问题，并解释如何在CMakeLists.txt文件中引用版本信息。

`LANGUAGES` 参数定义了应该为项目启用的编程语言，支持的语言包括 `c`、`cxx`、`Fortran`、`ASM`、`Java` 等。如果指定了多种语言，请用空格分隔。某些情况下，项目可能不使用任何语言，这可以使用 `NONE`。如果没有语言选项，CMake将默认为 `c` 和 `cxx`。3.0之前的CMake版本不支持 `LANGUAGES` 关键字，但使用旧形式的命令仍然可以指定 `LANGUAGES`，操作如下：

```
project(myProj C CXX)
```

新项目应将CMake的最低版本至少指定为3.0，并使用关键字 `LANGUAGES`。

`project()` 命令不仅仅是填充变量。它的重要职责是检查每种语言的编译器，并确保它们能够正确地编译和链接，这样编译器和链接器设置的问题会在很早就发现。一旦这些检查通过，**CMake**就会设置变量和属性来控制所支持语言的构建。如果**CMakeLists.txt**文件没有调用 `project()` 或者调用得不够早，**CMake**会在内部隐式调用默认语言 `c` 和 `cxx`，以确保依赖于编译器和链接器的其他命令正确设置。后面的章节详细介绍了如何建立工具链，并演示了如何查询和修改编译器标记、编译器位置等。

当**CMake**执行的编译器和链接器检查成功时，它们的结果会进行缓存，这样就不必在后续的**CMake**运行中重复。这些缓存信息存储在**CMakeCache.txt**文件的构建目录中。关于检查的额外细节可以在构建区的子目录中找到，开发人员只需要在使用新的或不常见的编译器时，或在设置用于交叉编译的工具链文件时，才需要查看。

### 3.3. 构建可执行文件

`add_executable()` 命令告诉**CMake**为一组源文件创建一个可执行文件。此命令的形式为：

```
add_executable(targetName source1 [source2 ...])
```

这将创建一个可执行文件，可以在**CMake**项目中作为 `targetName` 引用。这个名称可以包含字母、数字、下划线和连字符。构建项目时，在构建目录中创建一个与平台相关名称的可执行文件。

```
add_executable(myApp main.cpp)
```

默认情况下，可执行文件的名称在Windows上是**myApp.exe**，在基于Unix的平台(如macOS、Linux等)上是**myApp**。可执行文件的名称可以用目标属性来指定。通过使用不同的名称多次调用 `add_executable()`，可以在**CMakeLists.txt**文件中定义多个可执行文件。如果在多个 `add_executable()` 命令中使用相同的名称，**CMake**将失败并显示错误信息。

### 3.4. 注释

结束本章之前，演示一下如何向**CMakeLists.txt**文件添加注释。在本书中会大量使用注释，并且鼓励开发人员养成注释项目的习惯，就像注释普通源代码一样。**CMake**遵循与Unix Shell脚本类似的注释约定，任何以`#`字符开头的行都视为注释，除了在带引号的字符串中，**CMakeLists.txt**文件中一行中`#`之后的任何内容都视为注释。下面展示了一些注释示例，并与本章介绍的内容结合在一起：

```
cmake_minimum_required(VERSION 3.2)

# We don't use the C++ compiler, so don't let project()
# test for it in case the platform doesn't have one
project(MyApp VERSION 4.7.2 LANGUAGES C)

# Primary tool for this project
add_executable(mainTool
    main.c
    debug.c # Optimized away for release builds
)

# Helpful diagnostic tool for development and testing
add_executable(testTool testTool.c)
```

### 3.5. 总结

确保CMake项目都有 `cmake_minimum_required()` 命令作为其CMakeLists.txt文件的第一行。当决定最低版本号时，版本越新，项目能使用的特性就越多。该项目可能会更好地适应新平台或操作系统。相反，如果项目打算作为操作系统本身的一部分构建和发布(常见的Linux)，那么最低CMake版本可能由该发行版提供的CMake版本决定。

如果需要CMake 3.0或更高版本，最好尽早考虑项目版本号，并尽快将版本号合并到 `project()` 命令中。要克服现有过程的惰性，在项目生命周期的后期改变版本号非常困难。在决定版本控制策略时，请考虑诸如语义版本控制之类的方式。

# 第4章：构建目标

如前一章所示，在CMake中构建可执行文件不会很难。给出的例子需要为可执行文件定义目标名称，并列出要编译的源文件：

```
add_executable(myApp main.cpp)
```

假设开发者想要构建一个控制台可执行文件(当然CMake也允许开发者定义其他类型的可执行文件)，比如苹果平台上的应用程序包和Windows GUI应用程序。本章只讨论 `add_executable()` 提供的其他选项。

除了可执行文件之外，开发人员还需要构建和链接库。CMake支持几种不同类型的库，包括静态、动态、模块和框架。CMake提供了强大的特性来管理目标间的依赖关系，以及如何链接库，以及如何在CMake中使用它们。还会介绍了一些基本变量和属性的使用，以说明这些CMake特性是如何与库和目标相关联的。

## 4.1. 可执行文件

`add_executable()` 命令的完整形式如下：

```
add_executable(targetName [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               source1 [source2 ...]
)
```

与之前的区别是有新的可选关键字：

### *WIN32*

在Windows平台上构建可执行文件时，此选项表示CMake将该可执行文件构建为Windows GUI应用程序。实际上，会使用 `WinMain()` 创建入口，而不是 `main()`，并且会添加链接选项 `/SUBSYSTEM:WINDOWS`。其他平台会忽略 `WIN32` 选项。

### *MACOSX\_BUNDLE*

它会引导CMake在苹果平台上构建应用程序包。与选项名称不同，不仅适用于macOS，也适用于其他苹果平台，如iOS。此选项的确切效果在不同平台间略有不同。例如，在macOS上，`app bundle`有一个非常特定的目录结构，而iOS的目录结构是扁平的。CMake还会生成一个 `Info.plist` 文件。这些细节将在22.2节中详细地介绍。非Apple平台上，会忽略 `MACOSX_BUNDLE` 关键字。

### *EXCLUDE\_FROM\_ALL*

项目定义了许多目标，但默认情况下只需要构建一部分。当在构建没有指定目标时，将构建默认的 `ALL` 目标(根据使用CMake生成器的不同，名称略有不同，比如：Xcode的 `ALL_BUILD`)。如果可执行文件用 `EXCLUDE_FROM_ALL` 定义，就不会包含在默认的 `ALL` 目标中。然后，只有当构建命令显式地请求该可执行文件，或者作为默认构建对其中有依赖时，可执行文件才会构建。

除此之外，还有其他形式的 `add_executable()`，它们对现有可执行文件或目标是一种引用，而非要构建的新文件。这些别名可执行文件将在第16章中详细介绍。

## 4.2. 定义库

创建可执行文件是任何构建系统的基本需求。对于许多大型项目，创建和使用库也是保持项目可管理性的关键。`CMake`支持构建不同类型的库，可以处理平台差异，同时支持库的本地特性。库目标使用 `add_library()` 命令构建，命令中有许多参数。其中基本参数如下所示：

```
add_library(targetName [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...]
)
```

使用 `add_executable()` 可以构建可执行文件。`CMakeLists.txt` 文件中使用 `targetName` 来引用库，默认构建库的名称源于 `targetName`。`EXCLUDE_FROM_ALL` 关键字具有与 `add_executable()` 相同的效果，避免库包含在默认 `ALL` 目标中。构建库的类型由其余的三个关键字指定：`STATIC`、`SHARED` 或 `MODULE`。

#### **STATIC**

指定为静态库。在 Windows 上构建的库为 `targetName.lib`，而在类 Unix 平台上，通常是 `libtargetName.a`。

#### **SHARED**

指定动态库。在 Windows 上的库名称是 `targetName.dll`，在 Apple 平台上是 `libtargetName.dylib`，在类 Unix 平台上，通常是 `libtargetName.so`。Apple 平台上，动态库也可以标记为框架，这个将在第 22.3 节中介绍。

#### **MODULE**

这种方式类似于动态库，不同的是在运行时动态加载。通常会作为插件或可选组件，用户可以选择加载或不加载。Windows 平台上，不会为 DLL 创建导入库。

其实可以省略要构建的库类型的关键字，除非项目特别需要特定类型的库，并让开发人员在构建项目时进行选择。这样，库可以静态的，也可以是动态的，由名为 `BUILD_SHARED_LIBS` 的 CMake 变量的值决定。如果 `BUILD_SHARED_LIBS` 设置为 `true`，库将构建为动态库，否则将构建为静态库。与变量的设置方法相同，可以通过在 `cmake` 命令行执行时，使用 `-D` 选项进行定义：

```
cmake -DBUILD_SHARED_LIBS=YES /path/to/source
```

也可以在 `CMakeLists.txt` 文件中直接设置，只要在 `add_library()` 命令之前设置以下内容就可以了。但如果开发者想要改变它，就需要修改 `CMakeLists.txt` (即灵活性较差)：

```
set(BUILD_SHARED_LIBS YES)
```

与可执行文件一样，库目标也可以引用已有的二进制文件或目标，而不必是由项目构建出来的。另外还支持另一种类型的伪库，用于收集 `obj` 文件，而非创建静态库。这些将在第 16 章目标类型中详细讨论。

## 4.3. 目标连接

考虑搭建项目的目标时，开发人员通常习惯于考虑 A 库依赖于 B 库，因此将 A 链接到 B。这是处理库的传统方式，而在实际中，库之间存在几种不同类型的依赖关系：

#### **PRIVATE**

私有依赖项指定 A 库自己使用 B 库。其他链接到 A 库的目标都不需要知道 B 库，因为它只影响 A 库的内部实现。

#### **PUBLIC**

公共依赖关系指定**A**库不仅在内部使用**B**库，还在其接口中使用**B**库。这样，其他任何使用**A**库的目标也会依赖**B**库。例如，在**A**库中使用的函数，至少有一个在**B**库中定义和实现，因此代码需要依赖**B**库。

### INTERFACE

接口依赖规定，为了使用**A**库，也必须使用**B**库的一部分。当依赖于纯头文件库时，在`add_library()`中使用`INTERFACE`就非常有用了。

**CMake**通过`target_link_libraries()`命令，可以获得更丰富的依赖关系，该命令的一般形式为：

```
target_link_libraries(targetName
<PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
[<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
...
)
```

允许项目定义一个库如何依赖于其他库。然后，**CMake**负责管理以这种方式的链接。

```
add_library(collector src1.cpp)
add_library(algo src2.cpp)
add_library(engine src3.cpp)
add_library(ui src4.cpp)
add_executable(myApp main.cpp)
target_link_libraries(collector
PUBLIC ui
PRIVATE algo engine
)
target_link_libraries(myApp PRIVATE collector)
```

上例中，`ui`库`PUBLIC`链接到`collector`，因此即使`myApp`只直接链接到`collector`，由于这种关系，`myApp`也将链接到`ui`。另一方面，`algo`和`engine`作为`PRIVATE`链接到`collector`，所以`myApp`不会直接链接。第16.2节会讨论静态库的其他行为，这些行为可能会进行更多的链接，以满足依赖关系，包括循环依赖关系。

后面的章节将介绍其他`target_…()`系列的命令，这些命令增强了目标之间的依赖。**通过**`target_link_libraries()`连接时，允许编译器/链接器选项和头文件搜索路径从一个目标执行到另一个目标。这些特性从**CMake 2.8.11**到**3.2**渐进式添加，从而让**CMakeLists.txt**更加简单。

后面的章节还会讨论了更复杂的源目录层次。`target_link_libraries()`所使用的`targetName`，必须在`add_executable()`或`add_library()`命令之后调用，所连接的库目标需要在同一目录中进行定义。

## 4.4. 非目标连接

上一节中，所有的链接项都是当前**CMake**的目标，但`target_link_libraries()`命令要灵活很多。除了**CMake**目标之外，`target_link_libraries()`也可以通过以下内容指定：

### 库文件的完整路径

**CMake**将把库文件添加到链接器命令中。如果库文件更改，**CMake**将检测更改并重新链接目标。**CMake 3.3**版本中，链接器命令总是使用完整路径，但在**3.3**版本之前，某些情况下，**CMake**可能会要求链接器搜索库（例如：使用`/usr/lib/libfoo.so`替换`-lfoo`）。**3.3**版本之前的行为大多是历史遗留的，但对于感兴趣的读者来说，可以在**CMP0060**策略下的**CMake**文档看到完整的信息。

### 库名

如果只给出库的名称而没有路径，链接器将搜索该库(例如：`foo`变成`-lfoo`或`foo.lib`，具体取决于平台)。这对于系统提供的库来说很常见。

### 连接标识

有种特殊情况，需要以`-l`或`-framework`以外的连字符开头的形式添加到链接器命令中的标志。**CMake**会警告，因为这些应该只用于 `PRIVATE` 构建，因为如果定义为 `PUBLIC` 或 `INTERFACE`，会携带其他目标，这可能并不安全。

除了上述内容之外，由于历史原因，任何配置项前面都可以有一个关键字 `debug`、`optimized` 或 `general`。这些关键字的作用是，根据构建确定是否配置为调试，从而进一步细化应该在什么时候包含后面的配置项(参见第13章)。如果某个项前面有 `debug` 关键字，只有在生成为调试时才会添加该项。如果项前面有 `optimized` 的关键字，则只有在构建时(不是Debug构建)才会添加。`general` 关键字为所有构建配置添加选项，如果没有使用任何关键字，`general` 就是默认行为。对于新项目，应该避免使用 `debug`、`optimized` 和 `general` 的关键字，因为现代CMake有更清晰、更灵活和更健壮的特性可以实现同样的效果。

## 4.5. 老式CMake

`target_link_libraries()` 还有其他一些形式，这些形式早在2.8.11版本之前就已是**CMake**的一部分。这些选项有助于理解老式CMake项目，但不推荐在新项目中使用。应该在命令中使用 `PRIVATE`、`PUBLIC` 和 `INTERFACE`，从而更准确地表达依赖关系。

```
target_link_libraries(targetName item [item...])
```

上述形式通常等价于使用了 `PUBLIC`，但在某些情况下，可能视为 `PRIVATE`。如果项目混合使用新旧命令形式定义库的依赖关系，那么老式的方式通常会视为 `PRIVATE`。

另一种不推荐的形式如下：

```
target_link_libraries(targetName
    LINK_INTERFACE_LIBRARIES item [item...]
)
```

这是上面新形式 `INTERFACE` 关键字的一个前导，但是**CMake**文档不推荐这样使用。它的行为可以影响不同的目标属性，并使用相应的策略设置控制其行为。对于开发人员来说，这会产生一些疑惑，使用新的参数形式可以避免这种疑惑。

```
target_link_libraries(targetName
    <LINK_PRIVATE|LINK_PUBLIC> lib [lib...]
    [<LINK_PRIVATE|LINK_PUBLIC> lib [lib...]]
)
```

与老式参数类似，参数表是新形式 `PRIVATE` 和 `PUBLIC` 关键字的前导。同样，老式参数对于目标属性同样会产生疑惑，`PRIVATE / PUBLIC` 关键字会更适合于新项目。

## 4.6. 总结

目标名称与项目名称无关。常见的教程和例子使用同一个变量命名，也用于项目名称和可执行的目标，像这样：

```
# Poor practice, but very common
set(projectName MyExample)
project(${projectName})
add_executable(${projectName} ...)
```

这只适用于比较简单的项目。最好将项目名称和可执行文件名称分开，即使一开始是相同的。直接设置项目名称(而不是通过变量)，根据目标所做的事情(而不是它所属的项目)选择目标名称，并假定项目将定义多个目标。这能有助于好习惯的养成，在处理复杂项目时会很重要。

命名库的目标时，不要用lib作为名称的开头或结尾。许多平台上(除了Windows之外的所有平台)，构建实际的库名称时，lib会自动成为前缀，以使其符合平台的使用习惯。如果目标名称已经以lib开头，则生成的库文件名将以 `liblibsomething...` 为名称(通常会认为这是一个错误)。

除非有硬性原因需要这样做，否则尽量避免直接将库指定 `STATIC` 或 `SHARED`。这在选择静态库或动态库时有更大的灵活性，并可以作为整个项目范围的策略。可以通过 `BUILD_SHARED_LIBS` 变量修改默认值，而不必修改对 `add_library()` 的每次使用。

目标在调用 `target_link_libraries()` 时需要指定 `PRIVATE`、`PUBLIC` 和/或 `INTERFACE`。随着项目越来越复杂，这三个关键字对如何处理目标间的依赖关系有更大的影响。项目开始时这样使用会让开发人员考虑目标之间的依赖关系，这有助于更早地暴露项目结构的问题。

# 第5章：变量

前面的章节展示了如何定义目标和构建，不过CMake还附带了一整套其他特性，这些特性具有极大的灵活性和便利性。这一章来聊一聊CMake中变量的使用。

## 5.1. 基本变量

像任何计算机语言一样，变量是CMake的基础。定义变量的基本方法是使用 `set()`。可以在CMakeLists.txt中定义一个普通变量：

```
set(varName value... [PARENT_SCOPE])
```

变量的名称 `varName` 可以包含字母、数字和下划线，字母是区分大小写的。名称也可能包含字符`./-+`，但这些很罕见。

CMake变量有的作用域，就像其他语言中变量的作用域会限制在特定的函数或文件中一样，变量不能在其作用域外进行读取或修改。相比其他语言，CMake中的变量作用域更灵活一些，本章的简单例子中，我们将变量的作用域看作是全局的。第7章和第8章会介绍局部作用域的情况，并展示 `PARENT_SCOPE` 关键字是如何提高变量在作用域中的可见性。

CMake将所有变量都作为字符串处理。不同的上下文中，变量可能解释为不同的类型，但最终只是字符串。当设置一个变量时，CMake不要求这些值用引号括起来，除非值中包含空格。如果给定多个值，这些值将用分号连接在一起，生成的字符串就是CMake列表：

```
set(myVar a b c) # myVar = "a;b;c"  
set(myVar a;b;c) # myVar = "a;b;c"  
set(myVar "a b c") # myVar = "a b c"  
set(myVar a ;b;c) # myVar = "a;b;c"  
set(myVar a "b c") # myVar = "a;b c"
```

变量的值通过  `${myVar}` 获得，可以在任何需要的地方使用。CMake特别灵活，可以递归地使用这种形式或指定另一个变量的名称来设置。另外，CMake在使用变量之前不需要定义变量。使用未定义的变量只会替换一个空字符串，而不会出现错误或警告，这与Unix shell脚本非常类似。

```
set(foo ab) # foo = "ab"  
set(bar ${foo}cd) # bar = "abcd"  
set(baz ${foo} cd) # baz = "ab;cd"  
set(myVar ba) # myVar = "ba"  
set(big "${${myVar}r}ef") # big = "${bar}ef" = "abcdef"  
set(${foo} xyz) # ab = "xyz"  
set(bar ${notSetVar}) # bar = ""
```

字符串不限于单行，可以包含换行字符，还可以包含引号，不过需要使用反斜杠进行转义。

```
set(myVar "goes here")  
set(multiLine "First line ${myVar}  
Second line with a \"quoted\" word")
```

如果使用CMake 3.0或更高版本，可以使用lua的括号语法，内容的开始用 [= [ 标记，结束用 ]= ] 标记。方括号之间可以出现任意数量的 = 字符，但开始和结束必须使用相同数量的 = 字符。如果开始的方括号后面紧接着一个换行符，第一个换行符将忽略，但是随后的换行符不会忽略。此外，括号内的内容不会进行进一步的转换(即没有变量替换或转义)。

```
# Simple multi-line content with bracket syntax,
# no = needed between the square bracket markers
set(multiLine [[
First line
Second line
]])
# Bracket syntax prevents unwanted substitution
set(shellScript [=[
#!/bin/bash
[[ -n "${USER}" ]] && echo "Have USER"
=])
# Equivalent code without bracket syntax
set(shellScript
"#!/bin/bash
[[ -n \"\$USER\" ]] && echo \"Have USER\"
")
```

如上所示，括号语法特别适合于定义Unix shell脚本之类的内容。这些内容使用 \${...} 语法来实现自己的目的，并经常包含引号，但是使用括号语法意味着这些内容不需要转义，这与CMake内容的传统引号风格不同。在 [ 和 ] 标记之间使用任意数量的 = 字符的灵活性也意味着嵌入的方括号不会误解为标记。第18章包括了更多的例子，这些例子强调了括号语法是一个更好的选择的情况。

可以调用 `unset()` 或 `set()` 来解除变量的设置。下面两行是等价的，如果 `myVar` 不存在，也不会有错误或警告：

```
set(myVar)
unset(myVar)
```

除了自己定义的变量外，许多CMake命令的行为会受到特定变量的影响。这是CMake用来定制命令或修改默认值的常见模式，这样就不必为每个命令、目标定义进行重复执行。CMake参考文档通常会列出可能影响该命令行为的变量。本书后面的章节也展示了一些变量，以及影响构建的方式或提供关于构建的信息。

## 5.2. 环境变量

CMake允许使用普通变量符号进行检索，以及设置环境变量的值。环境变量通过 `$ENV{varName}` 获得，可以在任何使用 `${varName}` 的地方使用。设置环境变量的方法与普通变量相同，要设置的变量不是 `varName`，而是 `ENV{varName}`：

```
set(ENV{PATH} "$ENV{PATH}:/opt/myDir")
```

设置这样的环境变量只会影响当前的CMake实例。一旦CMake运行结束，对环境变量的更改就会丢失，对环境变量的更改在构建时不可见。因此，在CMakeLists.txt文件中设置环境变量很少会用到。

## 5.3. 缓存变量

除了上面讨论的常规变量之外，CMake还支持缓存变量。与一般的变量不同，缓存变量的生命周期仅限于CMakeLists.txt文件的处理，缓存变量存储在名为CMakeCache.txt的特殊文件中，并且在CMake运行期间持久存在。一旦设置好了，缓存变量就会保持不变，直到显式地将它们从缓存中删除。缓存变量的值与普通变

量的检索方法完全相同(即  `${myVar}` 形式), 但 `set()` 命令在设置缓存变量时有所不同:

```
set(varName value... CACHE type "docstring" [FORCE])
```

当 `CACHE` 关键字出现时, `set()` 命令将其当做一个名为 `varName` 的缓存变量, 而不是普通的变量。缓存变量比普通变量附带更多的信息, 包括类型和文档字符串。设置缓存变量时, 必须同时提供这两个参数, 文档字符串可以为空。类型和文档字符串都不会影响CMake如何处理变量, 只会让GUI工具以更合适的形式将变量呈现给用户。CMake在处理过程中将变量作为字符串处理, 类型只是为了改善GUI工具中的用户体验。类型必须是下列之一:

#### **BOOL**

缓存变量是一个布尔值。GUI工具使用复选框或类似的工具来表示变量。变量所包含的底层字符串值, 遵循CMake将布尔值表示为字符串的一种方式(`ON/OFF`, `TRUE/FALSE`, `1/0`, 等等——参见6.1.1节获得详细信息)。

#### **FILEPATH**

缓存变量表示为某个文件的路径。GUI工具为用户提供一个对话框来修改变量的值。

#### **PATH**

与 `FILEPATH` 类似, 但GUI工具会提供了目录选择对话框。

#### **STRING**

变量会当作字符串处理。默认情况下, GUI使用单行文本编辑工具来操作变量的值。可以使用缓存变量属性, 为GUI提供一组预定义值, 以组合框或类似的形式呈现。

#### **INTERNAL**

该变量对用户不可用。内部缓存变量有时用于持久地记录项目的内部信息, 例如: 缓存密集查询或计算的结果。GUI不显示内部变量。

GUI通常使用文档字符串作为缓存变量的工具提示, 或者在选择变量时作为简要的描述。文档字符串应该简短, 并且由纯文本组成(例如, 没有HTML标记等)。后续将进一步讨论 `FORCE` 关键字。

设置布尔缓存变量非常常见, CMake为此提供了单独的命令。开发人员可以使用 `option()` 代替冗长的 `set()`:

```
option(optVar helpString [initialValue])
```

如果省略 `initialValue`, 将使用默认值 `OFF`。如果提供, `initialValue` 必须符合 `set()` 接受的布尔值。以上内容等价于:

```
set(optVar initialValue CACHE BOOL helpString)
```

与 `set()` 相比, `option()` 命令更清楚地表达了布尔缓存变量的行为, 因此是首选命令。

普通变量和缓存变量之间的区别是, `set()` 只会在 `FORCE` 关键字存在的情况下覆盖缓存变量。定义缓存变量时, `set()` 更类似于 `set-if-not-set`, 与 `option()` 命令的作用类似(没有 `FORCE` 功能)。这样做的主要原因是缓存变量为开发人员定制。可以将CMakeLists.txt文件中的值硬编码为普通变量, 也可以使用缓存变量, 以便开发人员不必编辑CMakeLists.txt文件就可以覆盖。变量可以通过交互式GUI工具或脚本进行修改, 而不需要更改项目本身中的内容。

有一点不好理解，就是普通变量和缓存变量是两个独立的东西。可能普通变量和缓存变量同名不同值的情况。这种情况下，使用 `set(myVar foo)` 时，CMake将检索普通变量的值而不是缓存变量，或者换一种方式，普通变量的优先级高于缓存变量。例外情况是，在设置缓存变量的值时，如果在调用 `set()` 之前缓存变量不存在，或者使用了 `FORCE`，那么当前范围内的任何普通变量都会删除。注意，这意味着在第一次运行和随后的 CMake运行中可能会得到不同的行为，因为在第一次运行中，缓存变量不存在，但在随后的运行中会存在。因此，第一次运行时，将会隐藏普通变量，但在随后的运行中不会这样做。举个例子说明这个问题：

```
set(myVar foo) # Local myVar
set(result ${myVar}) # result = foo
set(myVar bar CACHE STRING "") # Cache myVar
set(result ${myVar}) # First run: result = bar
# Subsequent runs: result = foo
set(myVar fred)
set(result ${myVar}) # result = fred
```

简单地说，是 `set(myVar)` 将检索分配给 `myVar` 的最后一个值，不管它是普通变量还是缓存变量。在第7章和第8章会解释这种行为，进一步说明变量的作用域如何影响 `set(myVar)` 返回的值。

## 5.4. 控制缓存变量

项目可以使用 `set()` 和 `option()` 为开发人员构建有用的定制选项。通过这些选项可以打开或关闭不同的构建部分，可以设置外部包的路径，可以修改编译器和链接器的标志等等。后面的章节将介绍缓存变量的用法，但先要了解操作这些变量的方法。开发人员可以通过两种方式来完成这项工作，一种是通过cmake命令行，另一种是使用GUI工具。

### 5.4.1. 使用命令行设置缓存变量

CMake允许通过传递给CMake的命令行选项操作缓存变量。主要是 `-D` 选项，用于定义缓存变量的值。

```
cmake -D myVar:type=someValue ...
```

`someValue` 将替换之前 `myVar` 缓存变量的值。这种行为就像使用 `set()` 命令的 `CACHE` 和 `FORCE` 选项设置一样。命令行选项只需要一次，可以存储在缓存中以便后续运行，因此不需要每次运行cmake时提供。可以在 cmake命令行中提供多个 `-D` 选项来设置多个变量。

以这种方式定义缓存变量时，就不必在CMakeLists.txt文件中进行设置(即不需要相应的 `set()` 命令)。命令行上定义的缓存变量的说明字符串为空。类型也可以省略，可以给变量类似于 `INTERNAL` 的特殊类型。下面展示了通过命令行设置缓存变量的示例：

```
cmake -D foo:BOOL=ON ...
cmake -D "bar:STRING=This contains spaces" ...
cmake -D hideMe=mysteryValue ...
cmake -D helpers:FILEPATH=subdir/helpers.txt ...
cmake -D helpDir:PATH=/opt/helpThings ...
```

注意，如果设置包含空格的缓存变量，`-D` 选项给出的整个值就应该用引号括起来。

有一种特殊情况是处理cmake命令行上没有声明类型的值。如果项目的CMakeLists.txt文件试图设置相同的缓存变量，并将类型指定为 `FILEPATH` 或路径，当缓存变量的值是一个相对路径，CMake会将它作为相对于调用 CMake的目录，并自动将其转换成一个绝对路径。这种方式并不是没有缺点，因为cmake可以从任何目录调

用，而不仅仅是构建目录。因此，建议开发人员在cmake命令行上为表示某种路径的变量指定变量时，始终包含一个类型。在命令行上总定变量的类型是个好习惯，这样能以最合适的形式显示在GUI中。

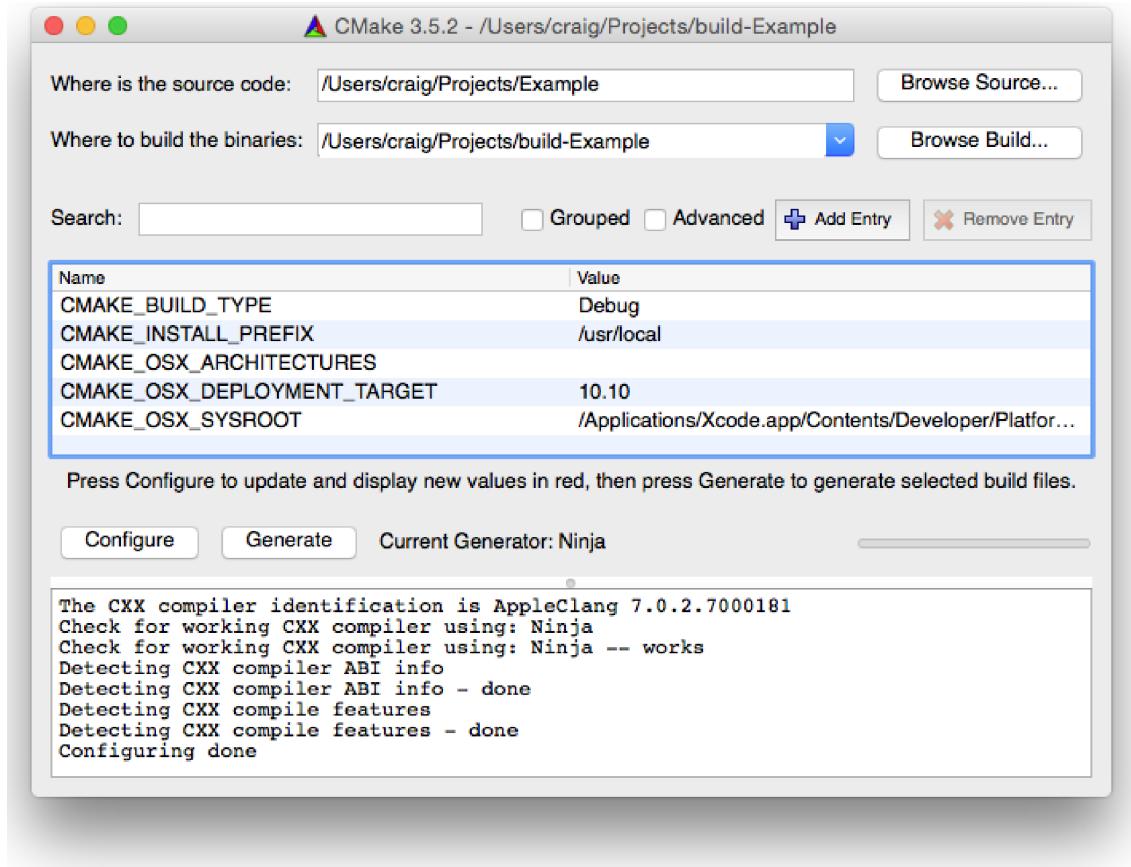
还可以使用 -U 选项从缓存中删除变量，必要时可以重复该选项以删除多个变量。-U 选项支持 \* 和 ? 通配符，但是需要避免删除超出预期的内容并使缓存处于不可构建状态。一般来说，除非确定使用通配符的安全性，否则建议只删除不带通配符的变量。

```
cmake -U 'help*' -U foo ...
```

### 5.4.2. CMake GUI

通过命令行设置缓存变量，是自动构建脚本和通CMake操作的基本部分。对于日常开发来说，CMake提供的GUI可提供更好的用户体验。CMake提供了两个GUI：cmake-gui和ccmake，其允许开发人员交互地操作缓存变量。cmake-gui需要桌面平台支持GUI显示，而ccmake使用一个文本的接口，可以在只有文本的环境中使用，比如：ssh连接。两者都包含在官方的平台CMake发布包中。

cmake-gui用户界面如下图所示。顶部允许定义项目的源目录和构建目录，中间部分是可以查看和编辑缓存变量的地方，而在底部是配置和生成按钮及其日志区域。



源目录必须设置为包含CMakeLists.txt文件的根目录。CMake将在构建目录中生成所有构建输出(推荐的目录布局在第2章中讨论过)。对于新项目，这两个目录都必须设置，对于现有项目，设置构建目录将更新源目录，因为源位置会存储在构建目录的缓存中。

第2.3节中介绍了CMake的设置过程。第一阶段，读取CMakeLists.txt文件，并在内存中构建项目，这称为配置阶段。如果配置阶段成功，则可以执行生成阶段，在build目录中创建构建工具的项目文件。从命令行运行cmake时，这两个阶段都是自动执行的，在GUI中分别由Configure和Generate按钮触发。每次启动配置步骤

时，UI中间显示的缓存变量都会更新。任何新添加的变量或在上次运行中更改的值都将以红色高亮显示(当项目首次加载时，所有变量都将高亮显示)。良好的习惯是重新运行 `Configure`，直到没有任何更改，这可以确保复杂项目的配置正确。有些项目中，启用某些选项可能会添加更多的选项。当所有缓存变量都没有显示红色，就可以进行 `Generate`。上一个示例显示了在 `Configure` 阶段，没有对任何缓存变量进行更改后的日志输出。

将鼠标悬停在任何缓存变量上都将显示包含该变量的描述字符串提示。还可以使用 `Add Entry` 按钮手动添加新的缓存变量，这相当于使用空文档字符串发出 `set()` 命令。缓存变量可以通过 `Remove Entry` 按钮删除，`CMake` 很可能会在下一次运行时重新创建该变量。

单击变量允许编辑其值，布尔值显示为选框，文件和路径有一个浏览文件系统按钮，字符串通常显示为文本行编辑。作为一种特殊情况，可以为 `STRING` 类型的缓存变量提供一组值，以便在 `CMake GUI` 的组合框中显示，而不是显示为简单的文本输入。这是通过设置缓存变量的 `STRINGS` 属性来实现的(详细内容见9.6节，这里只是为了显示方便):

```
set(trafficLight Green CACHE STRING "Status of something")
set_property(CACHE trafficLight PROPERTY STRINGS Red Orange Green)
```

`trafficLight` 缓存变量初始值为 `Green`。当用户尝试在 `cmake-gui` 中修改 `trafficLight` 时，将出现一个包含三个值红色、橙色和绿色的组合框。在变量上设置 `STRINGS` 属性并不会阻止为该变量分配其他值。该变量可以通过 `CMakeLists.txt` 文件中的 `set()` 命令或通过手动编辑 `CMakeCache.txt` 文件等其他方法获得。

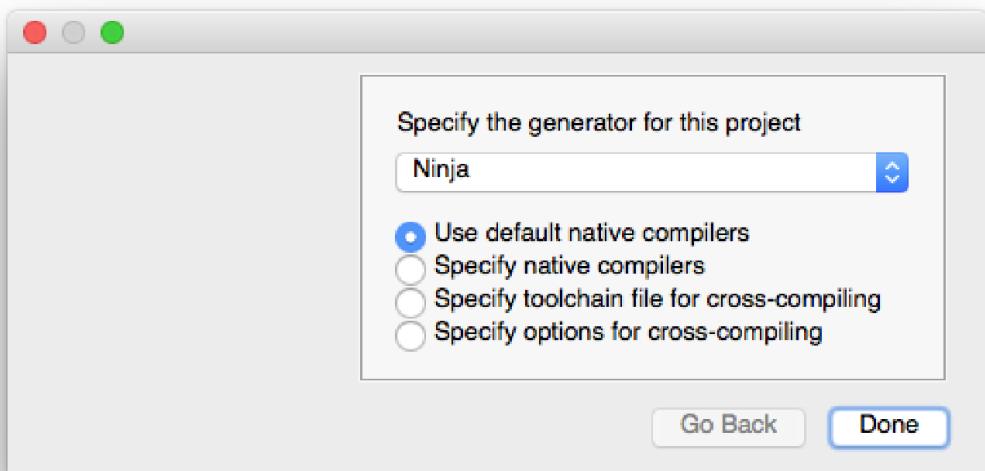
缓存变量可以将属性标记为高级或非高级。这也只会影响变量在 `cmake-gui` 中的显示方式，而不会影响 `CMake` 在处理过程中如何使用变量。默认情况下，`cmake-gui` 只显示非高级变量，通常只显示开发人员感兴趣的主要变量。启用 `Advanced` 选项可以显示除那些标记为内部的缓存变量外的所有缓存变量(查看内部变量的唯一方法是使用文本编辑器编辑 `CMakeCache.txt` 文件，因为不打算由开发人员直接操作)。`CMakeLists.txt` 文件中，可以使用 `mark_as_advanced()` 命令将变量标记为高级:

```
mark_as_advanced([CLEAR|FORCE] var1 [var2...])
```

`CLEAR` 关键字确保变量不标记为高级，而 `FORCE` 关键字确保变量标记为高级。如果没有这两个关键字，只有当变量还没有设置高级/非高级状态时，才会标记为高级。

通过选择分组选项，可以根据变量名的首字符到第一个下划线将变量分在一起，从而使查看高级变量变得更容易。过滤显示的变量列表的另一种方法是在搜索区域中输入文本，这样就只显示名称或值中包含指定文本的变量。

`Configure` 在新项目上第一次运行时，开发人员会看到类似于下图所示的对话框:



这个对话框是指定CMake生成器和工具链的地方。生成器的选择通常取决于开发人员的喜好，组合框中提供了可用的选项。如果项目依赖于特定生成器的功能，根据项目不同，生成器的选择可能更受限制。由于Apple平台的独特特性，比如：签名和iOS/tvOS/watchOS支持，需要Xcode生成器。一旦为项目选择了生成器，就需要删除缓存进行更改。如果需要，可以从File菜单中进行更改。

对于工具链选项，每个选项都需要开发人员提供。使用默认的本机编译器是普通桌面开发的常见选择，该选项不需要进一步的信息。如果需要更多的控制，开发人员可以重选编译器，并在后续对话框中给出编译器的路径。如果有单独的工具链文件可用，不仅可用于定制编译器，还可用于定制目标环境、编译器标志和其他东西。最后，开发人员可以指定交叉编译的全部选项，但不建议这样做。工具链文件可以提供相同的信息，优点是可以根据需要进行重用。

ccmake工具提供了与cmake-gui应用程序相同的功能。

A screenshot of a Terminal window titled "Terminal - ccmake - 93x15". The title bar says "ccmake". The main area shows configuration options for OSX:

```
CMAKE_BUILD_TYPE          Debug
CMAKE_INSTALL_PREFIX        /usr/local
CMAKE_OSX_ARCHITECTURES    [REDACTED]
CMAKE_OSX_DEPLOYMENT_TARGET 10.10
CMAKE_OSX_SYSROOT           /Applications/Xcode.app/Contents/Developer/Platforms/MacOS
```

Page 1 of 1

CCMAKE\_OSX\_ARCHITECTURES: Build architectures for OSX

Press [enter] to edit option CMake Version 3.5.2
Press [c] to configure
Press [h] for help Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)

与cmake-gui选择源目录和构建目录不同，必须在ccmake命令行中指定源目录或构建目录，就像cmake命令一样。

ccmake接口的主要缺点是无法像cmake-gui版本那样方便地捕获日志输出。这里也没有提供筛选显示变量的能力，而且编辑变量的方法也不像cmake-gui那样丰富。除此之外，当完整的cmake-gui不可用时，例如：终端连接上不能支持UI转发时，ccmake工具是一个替代选择。

## 5.5. 调试变量和诊断

当项目变得复杂或在调试意外行为时，**CMake**在运行时打印的诊断消息和变量值可能会很有用。通常使用 `message()` 命令来实现：

```
message([mode] msg1 [msg2]...)
```

如果指定了多个 `msg`，会连接在一起成为一个字符串，没有任何分隔符。这通常不是开发人员想要的，因此更常见的用法是使用单个 `message`。打印结果之前，可以使用变量值对其求值。

```
set(myVar HiThere)
message("The value of myVar = ${myVar}")
```

这将会输出：

```
The value of myVar = HiThere
```

`message()` 命令接受可选的 `mode` 关键字，该关键字会影响消息的输出方式，在某些情况下会在出现错误时停止构建。有效的 `mode` 值为：

### **STATUS**

附加信息。消息前面通常有两个连字符

### **WARNING**

**CMake**警告，通常在支持的地方(**CMake**命令行控制台或**cmake-gui**日志区域)以红色突出显示。处理将继续下去。

### **AUTHOR\_WARNING**

与警告类似，但只在启用开发模式时的警告显示(需要**cmake**命令行上的**-Wdev**选项)。通常不使用这种特殊类型的消息。

### **SEND\_ERROR**

指示将以红色突出显示的错误消息。处理将继续，直到配置阶段完成，但不会执行生成。这就像允许进一步尝试处理的错误，但最终仍然为失败。

### **FATAL\_ERROR**

表示一个错误。打印消息，处理将立即停止。日志通常还会记录 `message()` 命令的位置。

### **DEPRECATION**

用于记录弃用消息。如果 `CMAKE_ERROR_DEPRECATED` 变量定义为`true`，则该消息将视为错误。如果 `CMAKE_WARN_DEPRECATED` 定义为`true`，则该消息将视为警告。如果这两个变量都没有定义，则消息将不会显示。

如果没有提供 `mode` 关键字，则认为该消息是重要信息，不需要进行任何修改就可以记录。但是，需要注意的是，使用 `STATUS` 模式进行日志记录与不使用 `mode` 关键字进行日志记录完全不同。使用 `STATUS` 模式，消息将打印在命令与其他**CMake**消息之前，还有两个连字符前缀。当没有任何 `mode` 关键字时，没有连字符前缀。

**CMake**提供的另一种帮助调试变量使用的机制是 `variable_watch()` 命令。这适用于更复杂的项目，其中可能不清楚变量如何以特定值结束。监视一个变量时，所有读取或修改它的尝试都会记录下来。

```
variable_watch(myVar [command])
```

绝大多数情况下，列出要监视的变量就足够了，因为它会记录对指定变量的所有访问。然而，为了实现最终的控制，可以给出一个命令，该命令将在每次读取或修改变量时执行。**CMake**的函数或宏，可以传递以下参数：变量名、类型的访问、变量的值、当前列表文件的名称和文件列表堆栈(列表文件第7章中讨论)。但是，实际中很少使用`variable_watch()`。

## 5.6. 处理字符串

随着项目复杂性的增加，也需要实现有关如何管理变量的逻辑。**CMake**为此提供了`string()`命令，提供了有用的字符串处理功能。此命令能够查找和替换操作、正则表达式匹配、大小写转换、删除空格和其他常见字符串操作。下面给出了一些更常用的功能，**CMake**参考文档应该是所有可用操作及其行为的规范。

`string()`的第一个参数定义了要执行的操作，后续参数取决于请求的操作。这些参数通常需要(至少)一个输入字符串，因为**CMake**命令不能有返回值，所以也要输入一个输出变量。在下面的示例中，这个输出变量为`outVar`。

```
string(FIND inputString subString outVar [REVERSE])
```

`FIND`在`inputString`中搜索子字符串，并在`outVar`中存储找到的子字符串的索引(第一个字符为索引0)，除非指定反向，否则以第一个出现的为准。如果`subString`没有出现在`inputString`中，则`outVar`的值将为-1。

```
set(longStr abcdefabcdef)
set(shortBit def)
string(FIND ${longStr} ${shortBit} fwdIndex)
string(FIND ${longStr} ${shortBit} revIndex REVERSE)
message("fwdIndex = ${fwdIndex}, revIndex = ${revIndex}")
```

结果为：

```
fwdIndex = 3, revIndex = 9
```

替换子字符串遵循以下的模式：

```
string(REPLACE matchString replaceWith outVar input [input...])
```

`REPLACE`操作将使用`replaceWith`替换输入字符串中每个`matchString`，并将结果存储在`outVar`。当给定多个输入字符串时，在搜索替换之前，它们将连接在一起，每个字符串之间没有任何分隔符。有时这会导致意外的匹配，大多数情况下，开发人员提供一个输入字符串就好。

正则表达式`REGEX`也可以支持，有几个不同的变体可用，并由第二个参数决定：

```
string(REGEX MATCH regex outVar input [input...])
string(REGEX MATCHALL regex outVar input [input...])
string(REGEX REPLACE regex replaceWith outVar input [input...])
```

要匹配的正则表达式`REGEX`可以使用基本正则表达式语法(有关完整规范，请参阅**CMake**参考文档)，不支持一些常见特性，比如：逻辑非。替换之前，将输入字符串连接起来。匹配操作只查找第一个匹配项并将其存储在`outVar`中。`MATCHALL`查找所有匹配项并将它们作为列表存储在`outVar`中。`REPLACE`将返回整个输入字符串。

串，并将匹配项全替换为 `replaceWith`。可以使用符号 `\1`、`\2` 等来代替引用匹配，但请注意，反斜杠本身必须在CMake中进行转义。

```
set(longStr abcdefabcdef)
string(REGEX MATCHALL "[ace]" matchVar ${longStr})
string(REGEX REPLACE "([de])" "X\1Y" replVar ${longStr})
message("matchVar = ${matchVar}")
message("replVar = ${replVar}")
```

输出：

```
matchVar = a;c;e;a;c;e
replVar = abcXdYXeYfabcXdYXeYf
```

提取子字符串也可以：

```
string(SUBSTRING input index length outVar)
```

索引是一个整数，定义要从输入中提取的子字符串开始。提取最大长度的字符，或者如果长度为-1，则返回的子字符串将包含所有字符。注意，在CMake 3.1和更早的版本中，如果长度超过了字符串的末尾就会报错。

字符串的长度很容易地得到，字符串可以很容易地转换为大写或小写。从字符串的开头和结尾去掉空格也很简单。这些操作的语法都采用相同的形式：

```
string(LENGTH input outVar)
string(TOLOWER input outVar)
string(TOUPPER input outVar)
string(STRIPIEWS input outVar)
```

CMake还提供其他操作，如字符串比较、哈希、时间戳等，但在日常CMake项目中并不常见。感兴趣的读者可以查阅CMake参考文档中的 `string()` 命令以获得详细信息。

## 5.7. 列表

列表在CMake中广泛使用。列表只是一个用分号分隔的列表项的单个字符串，这使得操作单个列表项变得不太方便。CMake提供了 `list()` 命令来简化这些任务。与 `string()` 命令一样，`list()` 第一个参数作为执行参数，第二个参数是要操作的列表，必须是一个变量(例如，不允许传递一个原始列表，比如`a;b;c`)。

列表操作是计算目标的数量，并从列表中检索一个或多个目标：

```
list(LENGTH listVar outVar)
list(GET listVar index [index...] outVar)
```

```
# Example
set(myList a b c) # Creates the list "a;b;c"
list(LENGTH myList len)
message("length = ${len}")
list(GET myList 2 1 letters)
message("letters = ${letters}")
```

输出为：

```
length = 3
letters = c;b
```

追加和插入也很简单:

```
list(APPEND listVar item [item...])
list(INSERT listVar index item [item...])
```

与 LENGTH 和 GET 不同, APPEND 和 INSERT 直接作用于 listVar , 并在适当的地方修改它:

```
set(myList a b c)
list(APPEND myList d e f)
message("myList (first) = ${myList}")

list(INSERT myList 2 X Y Z)
message("myList (second) = ${myList}")
```

输出:

```
myList (first) = a;b;c;d;e;f
myList (second) = a;b;X;Y;Z;c;d;e;f
```

列表中查找特定目标:

```
list(FIND myList value outVar)
```

```
# Example
set(myList a b c d e)
list(FIND myList d index)
message("index = ${index}")
```

输出为:

```
index = 3
```

删除供了三种操作, 所有这些操作都直接对列表进行修改:

```
list(REMOVE_ITEM myList value [value...])
list(REMOVE_AT myList index [index...])
list(REMOVE_DUPLICATES myList)
```

REMOVE\_ITEM 可用于从列表中删除一个或多个目标项。如果目标项不在列表中, 也不会出错。另外, REMOVE\_AT 指定一个或多个要删除的索引, 如果指定的任何索引超过列表的末尾, CMake 将停止并报错。 REMOVE\_DUPLICATES 将确保列表不包含重复项。

列表项也可以重新排序与反向排序(按字母顺序):

```
list(VERSE myList)
list(SORT myList)
```

对于所有接受索引作为输入的列表操作，索引可以是负数，表示计数从列表的末尾开始。这样使用时，列表中的最后一项的索引为-1，倒数第二项为-2，依此类推。

上面描述了大多数可用的 `list()` 子命令。以上提到的都是在CMake 3.0之后就支持了。要获得子命令的完整支持列表，包括在以后CMake版本中添加的子命令，读者应该去参考CMake文档。

## 5.8. 数学表达式

另一种常见的变量操作是数学计算。CMake提供了 `math()` 命令来执行基本的数学计算：

```
math(EXPR outVar mathExpr)
```

第一个参数必须使用关键字 `EXPR`，而 `mathExpr` 定义要计算的表达式，结果将存储在 `outVar` 中。表达式可以使用以下操作符中的任何一个，这些操作符在C代码中的含义相同：`+ - * / % | & ^ ~ << >> * / %`。也支持括号，并且具有常规的数学意义。`mathExpr` 中变量可以使用  `${myVar}` 进行定义后引用。

```
set(x 3)
set(y 7)
math(EXPR z "${x}+${y} / 2")
message("result = ${z}")
```

期望输出为：

```
result = 5
```

## 5.9. 总结

CMake GUI是一种高效的方法，可以快速、轻松地理解项目的构建选项，并在开发过程中根据需要修改它们。花一点时间来熟悉它将简化处理更复杂项目的工作。还为开发人员提供了很好的工作基础，如果需要试验，诸如编译器设置之类的东西(因为这些很容易在GUI环境中找到和修改)。

使用缓存变量来控制是否启用构建的可选部分，而不是在CMake之外的构建脚本中编码逻辑。这使得在CMake GUI和解析CMakeLists.txt文件的工具中变得非常简单(越来越多的IDE环境正在获得这种能力)。

尽量避免依赖已定义的环境变量，除了 `PATH` 或类似的操作系统的变量。构建应该是可预测的、可靠的和容易修改的，但是如果依赖于环境变量，这可能会成为新手开发人员在构建环境时的痛点。此外，与调用构建本身时相比，CMake运行时的环境可能会发生变化。因此，最好通过缓存变量直接将信息传递给CMake。

作为构建的一部分的日志消息，请使用 `message()` 命令。如果消息是一般的信息，最好使用 `STATUS`，这样在构建日志中不会出现混乱。为了方便，临时调试信息通常不使用 `mode` 关键字，如果可能在任何时间内都作为项目的一部分，最好能指定模式(通常是 `STATUS`)。

CMake提供了大量的预定义变量，这些变量提供了系统的细节或影响了某些方面的行为。其中一些变量在项目大量使用，比如仅为特定平台(WIN32、APPLE、UNIX等)构建时定义的变量。因此，建议开发人员快速浏览一下CMake文档页面上列出的预定义变量，以熟悉这些变量。

# 第6章：控制流

大多数CMake项目在特定的情况下会使用一些特定的步骤。例如，项目可能希望仅在特定编译器或在为特定平台构建时使用特定的编译器标记。在其他情况下，项目可能需要迭代一组值，或者不断重复某些步骤，直到满足某个条件。CMake以大多数软件开发人员熟悉的方式，对这些控制提供了很好的支持。`if()` 命令提供了`if-then-else` 行为，循环通过`foreach()` 和`while()` 命令提供的。这三个命令与大多数编程语言的行为相同，同时也添加了CMake的特性。

## 6.1. if()

`if()` 命令的形式如下(可以提供多个`elseif()` 子句):

```
if(expression1)
# commands ...
elseif(expression2)
# commands ...
else()
# commands ...
endif()
```

早期版本的CMake要求将`expression1` 作为`else()` 和`endif()` 的参数，自从CMake 2.8.0之后就没有这个要求了。尽管遇到使用这种旧形式的项目和示例代码仍然很常见，因为它读起来有些混乱，所以不推荐新项目使用这种形式。新项目应该保留`else()` 和`endif()` 参数为空。

`if()` 和`elseif()` 命令中的表达式可以采取不同的形式。CMake提供了传统的布尔逻辑以及各种其他方式，如测试文件系统、版本比较和检查存在性。

### 6.1.1. 表达式

表达式中最基本的形式中只一个常量:

```
if(value)
```

CMake的真假逻辑比大多数编程语言复杂一些。对于未引用的常数，规则如下:

- 如果`value` 是一个没有引号的常数，值为1、ON、YES、TRUE、Y或一个非零数，将视为TRUE。不区分大小写。
- 如果`value` 是一个不带引号的常量，值为0、OFF、NO、FALSE、N、IGNORE、NOTFOUND、空字符串或以`-NOTFOUND` 结尾的字符串，则将其视为FALSE。同样，不区分大小写。
- 如果以上两种情况都不适用，则将视为变量名(或者可能是字符串)，并按照接下来的描述进行计算。

下面的例子中，为了说明只展示了`if(...)` 部分，省略了相应的分支操作和`endif()`:

```

# Examples of unquoted constants
if(YES)
if(0)
if(TRUE)

# These are also treated as unquoted constants because the
# variable evaluation occurs before if() sees the values
set(A YES)
set(B 0)
if(${A}) # Evaluates to true
if(${B}) # Evaluates to false

# Does not match any of the true or false constants, so proceed
# to testing as a variable name in the fall through case below
if(someLetters)

# Quoted value, so bypass the true/false constant matching
# and fall through to testing as a variable name or string
if("someLetters")

```

CMake参考文档中的if使用，形式如下所示：

```
if(<variable|string>)
```

实践中的如果表达式是：

- 变量(可能未定义)的无引号名称。
- 可引用字符串。

当使用无引号的变量名时，将该变量的值会与假常量进行比较。如果这些值都不匹配，则表达式的结果为真。未定义的变量将计算为一个空字符串，该字符串与一个**False**常量匹配，因此将产生一个**False**结果。

```

# Common pattern, often used with variables defined
# by commands such as option(enableSomething "...")
if(enableSomething)
# ...
endif()

```

然而，当if表达式是引用字符串时，行为就更复杂了：

- 不管字符串的值是多少，在CMake 3.1或更高版本中，一个带引号的字符串总是计算为**False**(但这可以设置，参见第12章)。
- 在CMake 3.1之前，如果字符串的值与现有变量的名称匹配，那么引用字符串将替换为该变量的名称(未引用)，进行重复测试。

以上两种情况都可能让开发人员感到奇怪（至少CMake 3.1的行为是可预测的）。3.1之前的行为会导致字符串替换，当字符串值碰巧与变量名匹配时，这个变量名可能是在很远的地方定义的。引用值有潜在混淆的可能时，建议避免在 `if(something)` 中使用引号。当然，有更好的比较表达式，可以更健壮地方式处理字符串，后面的6.1.3节中会有介绍。

## 6.1.2. 逻辑操作符

CMake支持 `AND`、`OR` 和 `NOT` 逻辑操作符，可以使用括号控制优先级。

```

# Logical operators
if(NOT expression)
if(expression1 AND expression2)
if(expression1 OR expression2)

# Example with parentheses
if(NOT (expression1 AND (expression2 OR expression3)))

```

按照通常的约定，首先计算括号内的表达式，从最里面的括号开始。

### 6.1.3. 比较测试

CMake将比较测试分为三个不同的类别：数字、字符串和版本号，不过语法形式都遵循相同的模式：

```
if(value1 OPERATOR value2)
```

两个操作数 `value1` 和 `value2` 既可以是变量名，也可以是(用引号括起来的)具体值。如果值与已定义变量的名称相同，那将视为变量。否则，将直接当作字符串或值处理。不过，引号中的值具有歧义性。在CMake 3.1之前，与变量名匹配的字符串替换成该变量的值。CMake 3.1及以后版本的行为使用了引用值而不会替换，这正是开发人员所期望的。

这三个比较类别都支持相同的操作集，但是每个类别的 `OPERATOR` 名称不同。下表总结了支持的操作符：

数字	字符串	版本号
LESS	STRLESS	VERSION_LESS
GREATER	STRGREATERT	VERSION_GREATER
EQUAL	STREQUAL	VERSION_EQUAL
LESS_EQUAL\$^1\$	STRLESS_EQUAL\$^1\$	VERSION_LESS_EQUAL\$^1\$
GREATER_EQUAL\$^1\$	STRGREATERT_EQUAL\$^1\$	VERSION_GREATER_EQUAL\$^1\$

\$^1\$仅适用于CMake 3.7及以后的版本。

数值比较与预期一样，比较左边和右边的值。但请注意，如果操作数不是数字，并且不止一个数字时，行为就不完全符合官方文档，CMake通常也不会报错。根据数字和非数字的混合，表达式的结果不可预期。

```

# Valid numeric expressions, all evaluating as true
if(2 GREATER 1)
if("23" EQUAL 23)
set(val 42)
if(${val} EQUAL 42)
if("${val}" EQUAL 42)

# Invalid expression that evaluates as true with at
# least some CMake versions. Do not rely on this behavior.
if("23a" EQUAL 23)

```

版本号比较有点像数字比较的增强版。版本号假设为 `major[.minor[.patch[.]]]` 的形式，每个组件都应该是正整数。比较两个版本号时，首先比较 `major`。只有当 `major` 相等时，才会比较 `minor` (如果存在的话)，以此类推。缺失的部分视为零。下面的例子中，表达式都为True：

```
if(1.2 VERSION_EQUAL 1.2.0)
if(1.2 VERSION_LESS 1.2.3)
if(1.2.3 VERSION_GREATER 1.2 )
if(2.0.1 VERSION_GREATER 1.9.7)
if(1.8.2 VERSION_LESS 2 )
```

版本号与数字比较具有同样的健壮性。版本号的每个部分都应该是整数，如果没有这个限制，比较结果是未定义的。

对于字符串，是按字典顺序进行比较。没有对字符串的内容做任何假设，但是要注意前面描述变量/字符串替代的可能性。字符串比较是发生替换的常见情况之一。

**CMake**还支持根据正则表达式比较字符串：

```
if(value MATCHES regex)
```

同样遵循上面定义的变量或字符串规则，并与 `regex` 正则表达式进行比较。如果匹配，则表达式的结果为 `True`。虽然**CMake**文档没有为 `if()` 命令定义支持的正则表达式语法，但在其他地方定义了正则表达式语法（例如，参见 `string()` 命令）。本质上，**CMake**支持基本的正则表达式语法。

括号可用于获取部分匹配值。命令将以 `CMAKE_MATCH_<n>` 的形式设置变量，其中 `<n>` 是要匹配的组。整个匹配的字符串存储在0组中。

```
if("Hi from ${who}" MATCHES "Hi from (Fred|Barney).*")
message("${CMAKE_MATCH_1} says hello")
endif()
```

## 6.1.4. 文件系统的比较

**CMake**还包括一组可用于查询文件系统的比较。支持下列表达式：

```
if(EXISTS pathToFileOrDir)
if(IS_DIRECTORY pathToDir)
if(IS_SYMLINK fileName)
if(IS_ABSOLUTE path)
if(file1 IS_NEWER_THAN file2)
```

上面的内容都能自解释，有几点需要注意。如果文件丢失或两个文件有相同的时间戳（包括如果同一个文件 `file1` 和 `file2`），`IS_NEWER_THAN` 操作符会返回 `True`。因此，实际执行 `IS_NEWER_THAN` 之前，需要判断 `file1` 和 `file2` 是否存在，如果其中任何一个文件丢失了，那么 `IS_NEWER_THAN` 的结果通常不是开发人员所期望的。使用 `IS_NEWER_THAN` 时也应该给出完整路径。

需要注意的另一点是，与大多数`if`表达式不同，没有任何文件系统操作符在没有 `{}$` 的情况下执行任何变量/字符串替换（不考虑引号）。

## 6.1.5. 存在性测试

最后一类`if`表达式支持测试是否存在各种**CMake**实例。它们在较大、更复杂的项目中特别有用，这些项目中，某些部分可能存在，也可能不存在，或者启用了。

```
if(DEFINED name)
if(COMMAND name)
if(POLICY name)
if(TARGET name)
if(TEST name) # Available from CMake 3.4 onward
```

如果在**if**命令执行时存在，上面的每一个都将返回**True**。

#### **DEFINED**

如果指定名称的变量存在，则返回**True**。变量的值无关紧要，只测试存在性。这也可以用来检查是否定义了特定的环境变量：

```
if(DEFINED SOMEVAR) # Checks for a CMake variable
if(DEFINED ENV{SOMEVAR}) # Checks for an environment variable
```

#### **COMMAND**

测试指定名称的**CMake**命令、函数或宏是否存在。这对于在尝试使用某个东西之前，检查它是否定义。对于**CMake**提供的命令，最好测试**CMake**版本，但是对于项目提供的函数和宏(参见第8章)，用命令来测试存在是非常有用的。

#### **POLICY**

测试某一特定策略是否已知。策略名通常采用 `CMPxxxx` 的形式，其中 `xxxx` 部分总是一个四位数。关于这个主题的详细信息，请参阅第12章。

#### **TARGET**

如果**CMake**目标已由 `add_executable()`、`add_library()` 或 `add_custom_target()` 定义，则返回**True**。目标可以在任何目录中定义，这个测试在复杂的项目结构中特别有用，这些复杂的项目结构引入其他外部项目，并且这些项目可能共享共同的依赖项(例如，这种**if**测试可以用于在尝试创建目标之前检查目标是否定义)。

#### **TEST**

如果**CMake**测试前已经由 `add_test()` 命令定义(在第24章中详细介绍)，则返回**True**。

这个测试在**CMake** 3.5及以后可用：

```
if(value IN_LIST listVar)
```

如果变量 `listVar` 包含指定的值，该表达式将返回**True**，`value` 遵循变量或字符串规则，但 `listVar` 必须是列表变量的名称。

### **6.1.6. 常见用例**

`if()` 的一些用法非常常见，其中许多依赖于预定义的**CMake**变量，特别是与编译器和目标平台相关的变量。不幸的是，这种基于错误变量的表达式很常见。例如，假设一个项目有两个C++源文件，一个用Visual Studio或与之兼容的构建器(如Intel)构建，另一个用所有其他编译器构建。这种逻辑通常是这样实现的：

```
if(WIN32)
set(platformImpl source_win.cpp)
else()
set(platformImpl source_generic.cpp)
endif()
```

虽然这可能适用于大多数项目，但并没有正确的表达约束。例如，考虑一个在Windows上构建但使用MinGW编译器的项目。对于这种情况，`source_generic.cpp`可能是更合适的源文件。以下可以更准确地实现上述目标：

```
if(MSVC)
    set(platformImpl source_msvc.cpp)
else()
    set(platformImpl source_generic.cpp)
endif()
```

另一个例子涉及到正在使用的CMake生成器的条件行为。特别是，CMake在使用Xcode生成器构建时，提供了其他生成器不支持的附加特性。项目有时会假设为macOS构建将使用Xcode生成器，但这并不是必须的(通常不是)。下面是一些不正确的用法：

```
if(APPLE)
    # Some Xcode-specific settings here...
else()
    # Things for other platforms here...
endif()
```

这看上去是正确的做法，但如果在macOS上使用不同的生成器(如Ninja或Unix makefile)，那就会失败。  
用 `APPLE` 没有表达正确的测试条件，这里应该测试CMake生成器：

```
if(CMAKE_GENERATOR STREQUAL "Xcode")
    # Some Xcode-specific settings here...
else()
    # Things for other CMake generators here...
endif()
```

上面的例子都是测试平台，而不是约束实际涉及的实体。因为平台是最容易理解和测试的，但使用不准确的约束会不必要地限制开发人员可用的生成器选择，或者导致完全错误的行为。

另一个常见的示例(这次使用得很恰当)是基于是否设置了特定的CMake选项，从而有条件地包含目标。

```
option(BUILD_MYLIB "Enable building the myLib target")
if(BUILD_MYLIB)
    add_library(myLib src1.cpp src2.cpp)
endif()
```

更复杂的项目通常使用上述模式，有条件地包含目录或执行，基于CMake选项或缓存变量的其他任务。开发人员可以在不直接编辑CMakeLists.txt文件的情况下打开/关闭该选项或将该变量设置为非默认值。对于由持续集成系统等驱动的脚本构建特别有用，这些脚本构建可能需要启用或禁用某些构建部分。

## 6.2. 循环

许多CMake项目中，另一个需求是对列表或范围内的值进行操作。另外，可能需要多次操作，直到满足条件为止。CMake可以很好地涵盖了这些需求，并添加了一些功能。

### 6.2.1. `foreach()`

CMake提供 `foreach()` 命令，使项目能够遍历一组项或值。`foreach()` 有几种不同的形式，其中最基本的是：

```
foreach(loopVar arg1 arg2 ...)
# ...
endforeach()
```

上面的形式中，对每个 `argN` 值，将 `loopVar` 设置为该参数，执行循环。不执行变量/字符串测试，参数按照值的方式使用。不需要显式地列出每一项，参数也可以通过一个或多个列表来指定变量，常用形式为：

```
foreach(loopVar IN [LISTS listVar1 ...] [ITEMS item1 ...])
# ...
endforeach()
```

更常规的形式中，可以使用 `ITEMS` 关键字指定单个参数，但是 `LISTS` 关键字允许指定一个或多个列表变量。使用常规形式时，必须提供一个或两个 `ITEMS` 和/或 `LISTS`。两者都提供时，`ITEMS` 必须出现在 `LISTS` 之后。`listVarN` 列表变量允许是一个空列表。下面的例子有助于阐明这个更常规的用法。

```
set(list1 A B)
set(list2)
set(foo WillNotBeShown)
foreach(loopVar IN LISTS list1 list2 ITEMS foo bar)
    message("Iteration for: ${loopVar}")
endforeach()
```

上面代码的输出为：

```
Iteration for: A
Iteration for: B
Iteration for: foo
Iteration for: bar
```

`foreach()` 命令还支持对数值的迭代：

```
foreach(loopVar RANGE start stop [step])
```

使用 `foreach()` 时，循环执行时将 `loopVar` 设置为范围内开始停止(包括)的每个值。如果提供了 `step` 选项，则在每次迭代后将此值替换为前一个值，当结果大于 `stop` 时循环停止。

`RANGE` 形式只接受一个参数，像这样：

```
foreach(loopVar RANGE value)
```

这相当于 `foreach(loopVar RANGE 0 value)`，循环体将执行 `(value + 1)` 次，更直观的数值是循环体执行的次数，显式地指定开始和停止值会让循环更为清楚。

与 `if()` 和 `endif()` 命令的类似，CMake的早期版本中(即2.8.0之前)，所有的 `foreach()` 命令都需要将 `loopVar` 也指定为 `endforeach()` 参数。这降低了可读性，所以不鼓励在新项目中对 `endforeach()` 指定 `loopVar` 参数。

### 6.2.2. `while()`

CMake提供的另一个循环是 `while()`：

```
while(condition)
# ...
endwhile()
```

如果条件为True(遵循与 `if()` 语句中的表达式相同的规则), 则执行循环体。重复此操作, 直到条件计算为 `False`或提前退出循环(参见下一节)。在2.8.0之前的版本中, 必须在 `endwhile()` 命令中重复该条件, 对于新项目不鼓励这样做。

### 6.2.3. 中断循环

`while()` 和 `foreach()` 循环都支持使用 `break()` 提前退出循环, 或者使用 `continue()` 跳至下一个迭代。这些命令的行为就像C语言一样, 都只在内部的封闭循环上操作。下面的示例演示了该行为:

```
foreach(outerVar IN ITEMS a b c)
unset(s)
foreach(innerVar IN ITEMS 1 2 3)
# Stop inner loop once string s gets long
list(APPEND s "${outerVar}${innerVar}")
string(LENGTH s length)
if(length GREATER 5)
break() ①
endif()

# Do no more processing if outer var is "b"
if(outerVar STREQUAL "b")
continue() ②
endif()
message("Processing ${outerVar}-${innerVar}")
endforeach()

message("Accumulated list: ${s}")
endforeach()
```

① 提前结束 `innerVar` 的循环。② 结束当前 `innerVar` 的迭代, 跳转至下一个迭代。

上面例子的输出:

```
Processing a-1
Processing a-2
Processing a-3
Accumulated list: a1;a2;a3
Accumulated list: b1;b2;b3
Processing c-1
Processing c-2
Processing c-3
Accumulated list: c1;c2;c3
```

## 6.3. 总结

`if()`、`foreach()` 和 `while()` 命令中的变量尽可能减少字符串形式。避免使用带引号的一元表达式, 最好使用字符串比较操作。强烈推荐使用3.1为最低版本来禁用旧的行为, 该行为允许将引用的字符串隐式转换为变量名。

如果正则表达式匹配 `if(xxx MATCHES regex)` 时, 捕获到所需的变量, 通常可以使用 `CMAKE_MATCH_<n>` 在比较后获取变量。当下一个正则表达式执行后, 将覆盖这些变量。

使用循环命令，可以避免模棱两可或误导的情况。如果使用 `foreach()` 的 `RANGE` 形式，则指定开始值和结束值。如果在进行迭代，请考虑使用 `IN LISTS` 或 `IN ITEMS` 的方式，会比 `foreach(loopVar item1 item2...)` 的方式更简单易懂。

# 第7章：子目录

对于简单的项目来说，将所有内容保存在一个目录中没什么问题，但大多数项目倾向于将文件放到多个目录中。通常会发现不同的文件类型或分组有自己的目录，或者属于逻辑功能文件在项目的目录层次结构中，有属于它们的部分。虽然目录结构可能由开发人员对项目的想法所形成，但项目结构也会影响构建系统。

任何多目录项目中有两个基本CMake命令是 `add_subdirectory()` 和 `include()`。这些命令将另一个文件或目录的内容引入到构建中，从而允许构建逻辑分布在整个目录结构中。这样做的好处有：

- 本地化构建，这意味着构建可以在相关的目录中定义。
- 构建可以由子组件组成，这些子组件的独立于顶层项目(项目使用git子库或嵌入第三方源)。
- 由于目录可以自包含，因此只需选择是否在该目录中添加，就可以轻松地打开或关闭部分构建。

`add_subdirectory()` 和 `include()` 具有不同的特性，所以了解其优缺点非常重要。

## 7.1. add\_subdirectory()

`add_subdirectory()` 允许项目将另一个目录引入到构建中。该目录必须有CMakeLists.txt文件，调用 `add_subdirectory()` 处理时，在项目的构建中会为其创建相应的目录。

```
add_subdirectory(sourceDir [ binaryDir ] [ EXCLUDE_FROM_ALL ])
```

`sourceDir` 不必是源树中的子目录(不过通常都是)，可以添加任何目录，将 `sourceDir` 指定为绝对路径或相对路径，相对路径相对于当前源目录。通常只在添加主源之外的目录时才需要绝对路径。

通常，不需要指定 `binaryDir`。当省略这个目录时，CMake会创建一个具有相同名称的目录 `sourceDir`。如果 `sourceDir` 包含任何路径组件，这些组件将映射到CMake创建的 `binaryDir` 中。另外，可以显式地将 `binaryDir` 指定为绝对路径或相对路径，相对路径是相对于当前二进制目录进行计算的(稍后将详细讨论)。如果 `sourceDir` 是源之外的路径，则需要指定 `binaryDir`，因为相应的路径不能再进行自动构建。

`EXCLUDE_FROM_ALL` 关键字，用于控制子目录中的目标是否包含在项目的 `ALL` 目标中。不过，对于某些CMake版本和项目生成器，并不总是像预期的那样工作，甚至可能构建失败。

### 7.1.1. 源目录和构建目录

有时候，开发人员需要知道构建目录与当前源目录对应的位置，比如：在运行时需要复制文件或执行自定义构建任务。使用 `add_subdirectory()`，源和构建目录结构都可以是任意的。甚至可以使用同一个源进行多个构建。因此开发人员需要CMake的帮助来确定感兴趣的目录。为此，CMake提供了许多变量，用于跟踪当前正在处理的CMakeLists.txt文件的源目录和构建目录。在CMake处理每个文件时，将自动更新以下只读变量。它们总是以绝对路径显示：

`CMAKE_SOURCE_DIR`

源的最顶层目录(即最顶层CMakeLists.txt文件所在的位置)。这个变量永远不会变。

`CMAKE_BINARY_DIR`

构建的最顶层目录。这个变量永远不会改变。

`CMAKE_CURRENT_SOURCE_DIR`

当前处理的CMakeLists.txt文件的目录。通过 `add_subdirectory()` 处理新文件时更新。

#### CMAKE\_CURRENT\_BINARY\_DIR

当前处理的CMakeLists.txt文件对应的构建目录。每次调用 `add_subdirectory()` 时进行更新。

下面的例子有助于对以上变量的理解：

##### 顶层CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyApp)

message("top: CMAKE_SOURCE_DIR = ${CMAKE_SOURCE_DIR}")
message("top: CMAKE_BINARY_DIR = ${CMAKE_BINARY_DIR}")
message("top: CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top: CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")

add_subdirectory(mysub)

message("top: CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("top: CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

##### mysub/CMakeLists.txt

```
message("mysub: CMAKE_SOURCE_DIR = ${CMAKE_SOURCE_DIR}")
message("mysub: CMAKE_BINARY_DIR = ${CMAKE_BINARY_DIR}")
message("mysub: CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("mysub: CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
```

上面的例子，如果顶层CMakeLists.txt文件在/somewhere/src目录下，而build目录为/somewhere/build，则会有以下输出：

```
top: CMAKE_SOURCE_DIR = /somewhere/src
top: CMAKE_BINARY_DIR = /somewhere/build
top: CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top: CMAKE_CURRENT_BINARY_DIR = /somewhere/build
mysub: CMAKE_SOURCE_DIR = /somewhere/src
mysub: CMAKE_BINARY_DIR = /somewhere/build
mysub: CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/mysub
mysub: CMAKE_CURRENT_BINARY_DIR = /somewhere/build/mysub
top: CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
top: CMAKE_CURRENT_BINARY_DIR = /somewhere/build
```

## 7.1.2. 作用域

第5章中，简要提到了作用域的概念。`add_subdirectory()` 的效果是CMake为处理该目录的CMakeLists.txt文件创建一个新的作用域。这个新的作用域相当于一个子作用域：

- 调用作用域中定义的所有变量对子作用域是可见的，并且子作用域可以像其他变量一样读取它们的值。
- 子作用域中创建的任何新变量对调用作用域都不可见。
- 对子作用域内变量的任何更改都是该子作用域的局部更改。即使该变量存在于调用作用域中，调用作用域的变量也不会因此改变。在子作用域中修改的变量类似于在处理离开子作用域时丢弃的新变量。

换句话说，在进入子作用域时，会接收在调用作用域中定义的所有变量的副本。对子节点中变量的任何更改都将在子节点的副本上执行，而调用者的变量将保持不变。下面的例子可以很好地说明这种行为：

##### CMakeLists.txt

```

set(myVar foo)
message("Parent (before): myVar = ${myVar}")
message("Parent (before): childVar = ${childVar}")

add_subdirectory(subdir)

message("Parent (after): myVar = ${myVar}")
message("Parent (after): childVar = ${childVar}")

```

#### *subdir/CMakeLists.txt*

```

message("Child (before): myVar = ${myVar}")
message("Child (before): childVar = ${childVar}")

set(myVar bar)
set(childVar fuzz)

message("Child (after): myVar = ${myVar}")
message("Child (after): childVar = ${childVar}")

```

输出为：

```

Parent (before): myVar = foo ①
Parent (before): childVar = ②
Child (before): myVar = foo ③
Child (before): childVar = ④
Child (after): myVar = bar ⑤
Child (after): childVar = fuzz ⑥
Parent (after): myVar = foo ⑦
Parent (after): childVar = ⑧

```

① `myVar` 在父级定义。② `childVar` 在父级没有定义，所以是一个空字符串。③ `myVar` 在子作用域可见。  
 ④ `childVar` 在进入子作用域之前仍未定义。⑤ `myVar` 在子作用域中进行了修改。⑥ `childVar` 在子作用域中进行了赋值。  
 ⑦ 当返回到父作用域时，`myVar` 仍然拥有 `add_subdirectory()` 之前的值。子范围内对 `myVar` 的修改在父作用域中不可见。⑧ `childVar` 是在子范围内定义的，所以在父作用域内不可见，结果为空字符串。

上述确定变量作用域的行为展示了 `add_subdirectory()` 的重要特征。它允许添加的目录更改它任何变量，而不影响调用范围外的变量。

然而，有时需要在添加的目录中对调用者可见的变量更改。例如，该目录可能负责收集一组源文件名，并将其作为文件列表传递回父文件。这就是 `set()` 命令中 `PARENT_SCOPE` 关键字的目的。使用 `PARENT_SCOPE` 时，变量设置在父作用域，并不是在当前作用域内。重要的是，它并不意味着在父作用域和当前作用域中都设置变量。稍微修改一下前面的例子，就能看到 `PARENT_SCOPE` 的效果了：

#### *CMakeLists.txt*

```

set(myVar foo)
message("Parent (before): myVar = ${myVar}")
add_subdirectory(subdir)
message("Parent (after): myVar = ${myVar}")

```

#### *subdir/CMakeLists.txt*

```

message("Child (before): myVar = ${myVar}")
set(myVar bar PARENT_SCOPE)
message("Child (after): myVar = ${myVar}")

```

输出为:

```
Parent (before): myVar = foo
Child (before): myVar = foo
Child (after): myVar = foo ①
Parent (after): myVar = bar ②
```

① 子作用域中的 `myVar` 不受 `set()` 的影响，因为 `PARENT_SCOPE` 关键字告诉CMake修改父作用域中的 `myVar`，而不是本地的。② 父对象的 `myVar` 被子作用域中的 `set()` 修改。

使用 `PARENT_SCOPE` 可以防止命令修改同名的任何局部变量，所以如果局部作用域没有重用与父变量相同的变量名，这样可以减少混淆。

`subdir/CMakeLists.txt`

```
set(localVar bar)
set(myVar ${localVar} PARENT_SCOPE)
```

上面的示例很简单，但对于实际项目来说，可能有许多命令可以在设置父类的 `myVar` 变量之前设置 `localVar` 的值。

不仅变量受到作用域的影响，策略和某些变量的属性也会受到影响。对于策略，`add_subdirectory()` 会创建一个新的作用域，在这个作用域中，可以在不影响父策略的情况下进行策略更改。类似地，可以在子目录的 `CMakeLists.txt` 文件中设置目录属性，这些属性不会影响父目录属性。后续的章节中会更详细地介绍。

## 7.2. `include()`

CMake提供的另一种从其他目录加入内容的方法是 `include()`，其有以下两种形式:

```
include(fileName [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
include(module [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
```

第一种形式有点类似于 `add_subdirectory()`，但有一些区别:

- `include()` 读入的是文件名，而 `add_subdirectory()` 读入一个目录，并在目录中查找 `CMakeLists.txt` 文件。传递给 `include()` 的文件名通常的后缀名为 `.cmake`，当然可以是其他后缀。
- `include()` 没有引入新的变量范围，而 `add_subdirectory()` 引入了新的变量范围。
- 默认情况下，这两个命令都引入了新的策略范围，可以通过 `NO_POLICY_SCOPE` 选项告诉 `include()` 不要这样做(`add_subdirectory()` 没有这样的选项)。
- 当 `include()` 处理文件时，`CMAKE_CURRENT_SOURCE_DIR` 和 `CMAKE_CURRENT_BINARY_DIR` 的值不会改变，而 `add_subdirectory()` 会改变这两个变量。稍后将对此进行更详细讨论。

`include()` 的第二种形式用于加载已命名的模块，这是第11章中的主题。除第一点外，上述所有要点也适用于第二种形式。

在使用 `include()` 时 `CMAKE_CURRENT_SOURCE_DIR` 没有改变，因此要计算出所在的目录似乎很困难。`CMAKE_CURRENT_SOURCE_DIR` 包含使用 `include()` 文件的位置。而且，在使用 `include()` 时，文件的名称可以随意，因此包含的文件很难确定名称。为了解决这些情况，CMake提供了一组额外的变量:

`CMAKE_CURRENT_LIST_DIR`

类似于 `CMAKE_CURRENT_SOURCE_DIR`，但会在处理包含的文件时更新。需要当前文件的目录时可以使用该变量，总表示为绝对路径。

### CMAKE\_CURRENT\_LIST\_FILE

提供当前文件的名称，保存文件的绝对路径。

### CMAKE\_CURRENT\_LIST\_LINE

保存目前正在处理的文件的行号。很少需要此变量，某些调试场景中可能会用到。

值得注意的是，上述三个变量适用于CMake处理任何文件。即使对于通过 `add_subdirectory()` 加入的 `CMakeLists.txt` 文件，也具有与上面描述相同的值，`CMAKE_CURRENT_LIST_DIR` 与 `CMAKE_CURRENT_SOURCE_DIR` 具有相同的值。下面的例子将进行演示：

#### *CMakeLists.txt*

```
add_subdirectory(subdir)
message("====")
include(subdir/CMakeLists.txt)
```

`subdir/CMakeLists.txt`.

```
message("CMAKE_CURRENT_SOURCE_DIR = ${CMAKE_CURRENT_SOURCE_DIR}")
message("CMAKE_CURRENT_BINARY_DIR = ${CMAKE_CURRENT_BINARY_DIR}")
message("CMAKE_CURRENT_LIST_DIR = ${CMAKE_CURRENT_LIST_DIR}")
message("CMAKE_CURRENT_LIST_FILE = ${CMAKE_CURRENT_LIST_FILE}")
message("CMAKE_CURRENT_LIST_LINE = ${CMAKE_CURRENT_LIST_LINE}")
```

输出如下：

```
CMAKE_CURRENT_SOURCE_DIR = /somewhere/src/subdir
CMAKE_CURRENT_BINARY_DIR = /somewhere/build/subdir
CMAKE_CURRENT_LIST_DIR = /somewhere/src/subdir
CMAKE_CURRENT_LIST_FILE = /somewhere/src/subdir/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE = 5
=====
CMAKE_CURRENT_SOURCE_DIR = /somewhere/src
CMAKE_CURRENT_BINARY_DIR = /somewhere/build
CMAKE_CURRENT_LIST_DIR = /somewhere/src/subdir
CMAKE_CURRENT_LIST_FILE = /somewhere/src/subdir/CMakeLists.txt
CMAKE_CURRENT_LIST_LINE = 5
```

上面的示例还展示了 `include()` 的另一个有趣的特性。可用于包含已包含在构建中的文件。如果一个复杂项目的不同子目录都希望在项目的公共区域的某个文件中使用CMake代码，那么可以单独 `include()` 该文件。

## 7.3. 提前终止处理

某些情况下，项目可能希望停止处理。`return()` 命令可以完成该功能，但不能返回值，它唯一的作用是结束当前范围的处理。如果不是在函数内部调用，那么 `return()` 将结束对当前文件的处理(无论 `include()` 还是 `add_subdirectory()` 引入)。8.4节会讨论在函数内部调用 `return()` 的影响，包括可能导致当前文件返回的常见错误。

如前一节所述，项目的不同部分可能包含多个位置的相同文件。有时需要检查，只包含一次文件，提前返回后续的包含，以防止多次重新处理文件。这与包含C/C++头文件的情况非常相似：

```
if(DEFINED cool_stuff_include_guard)
    return()
endif()

set(cool_stuff_include_guard 1)
# ...
```

CMake 3.10或更高版本中，用一个专用的命令来表达，其行为类似于C/C++的 `#pragma once`：

```
include_guard()
```

与手动 `if-endif` 代码相比，这更加健壮，因为只在内部处理变量名称。该命令还接受一个可选的关键字参数 `DIRECTORY` 或 `GLOBAL`，以指定范围，会在范围内检查以前处理过的文件，但在大多数情况下不太需要这些关键字。如果没有指定任何参数，则默认为变量作用域，其效果与 `if-endif` 完全相同。如果文件在项目的其他地方处理过(即忽略变量作用域)，`GLOBAL` 可以让文件的处理结束。`DIRECTORY` 仅在当前目录范围内及其子范围内进行检查。

## 7.4. 总结

使用 `add_subdirectory()` 或 `include()` 将另一个目录引入构建过程，哪种是最佳选择并不好说。`add_subdirectory()` 简单，并且在保持目录自包含方面做得更好，因为创建了自己的作用域。一些CMake命令有一些限制，这些限制只允许对定义在当前文件范围内的内容进行操作，因此 `include()` 更适合这些情况。第28.5.1节会讨论这方面的内容。

大多数项目更倾向于使用 `add_subdirectory()` 而不是 `include()`。这样有更清晰的定义，并允许给定目录的 CMakeLists.txt 专注于该目录的内容。不过随着项目发展，会在某些目录中使用 `include()`。遵循此策略将促进项目更好的进行本地化，并且倾向于只在需要的地方引入复杂性。并不是说 `include()` 本身比 `add_subdirectory()` 更复杂，但是使用 `include()` 往往会需要更明确地说明文件路径，因为CMake考虑的是当前源目录，而不是包含的文件的目录。还有一些其他原因，从不同目录调用某些命令会有一些限制，因此 `add_subdirectory()` 可能会更加灵活。

不管使用 `add_subdirectory()`、`include()`，还是两者的组合，推荐使用 `CMAKE_CURRENT_LIST_DIR` 变量，而非 `CMAKE_CURRENT_SOURCE_DIR`。使用 `CMAKE_CURRENT_LIST_DIR`，当项目变得越来越复杂时，可以更容易地在 `add_subdirectory()` 和 `include()` 之间进行切换，可以更容易地修改目录进行重构。

如果项目需要CMake 3.10或更高版本，防止文件的多个包含时，最好使用不带参数的 `include_guard()` 命令，而非 `if-endif` 块。

# 第8章： 函数和宏

目前为止，**CMake**的已经很像一种编程语言了，支持变量、**if-then-else**逻辑、循环和文件包含等。当然，**CMake**还支持函数和宏等编程概念。就像在其他编程语言一样，函数和宏是项目和开发人员用于扩展**CMake**功能和封装重复任务的主要机制。其允许开发者定义可重用的**CMake**代码块，就像调用内置**CMake**命令一样。它们也是**CMake**模块系统的基石(第11章中单独介绍)。

## 8.1. 基本概念

**CMake**中的函数和宏与C/C++中的函数和宏有非常相似的特征。函数引入了一个新的作用域，函数参数可以在函数体中访问的变量。另一方面，宏有效地将其主体粘贴到调用点，并且宏参数可替换为简单的字符串。这些行为与 `#define` 在C/C++的工作方式相同。**CMake**函数或宏定义如下：

```
function(name [arg1 [arg2 [...]]])
    # Function body (i.e. commands) ...
endfunction()

macro(name [arg1 [arg2 [...]]])
    # Macro body (i.e. commands) ...
endmacro()
```

当定义函数或宏后，调用方式与任何其他**CMake**命令相同。然后，函数或宏的主体在调用点执行。例如：

```
function(print_me)
    message("Hello from inside a function")
    message("All done")
endfunction()

# Called like so:
print_me()
```

如上所示，`name` 参数定义了用于调用函数或宏的名称，它应该只包含字母、数字和下划线。名称是大小写不敏感的，因此大写/小写更多的是样式问题(**CMake**文档遵循的约定是命令名称都是小写的，单词由下划线分隔)。**CMake**的早期版本要求名称作为 `endfunction()` 或 `endmacro()` 的参数重复，但新项目没必要这样做。

## 8.2. 参数处理要点

函数和宏的参数处理方式相同，不过有一个非常重要的区别。对于函数来说，每个参数都是**CMake**变量，例如：可以作为变量在 `if()` 语句中进行判断。相比之下，宏参数是字符串替换项，因此无论使用什么作为宏调用的参数，基本上都会粘贴到宏主体中的参数中。如果在 `if()` 语句中使用宏参数，那么参数将视为字符串而不是变量。下面的例子和输出说明了其中差异：

```

function(func arg)
if(DEFINED arg)
    message("Function arg is a defined variable")
else()
    message("Function arg is NOT a defined variable")
endif()
endfunction()

macro(macr arg)
if(DEFINED arg)
    message("Macro arg is a defined variable")
else()
    message("Macro arg is NOT a defined variable")
endif()
endmacro()

func(foobar)
macr(foobar)

```

```

Function arg is a defined variable
Macro arg is NOT a defined variable

```

除了这种差异外，函数和宏在参数处理方面都支持相同的特性。函数定义的每个参数都作为代表参数的(区分大小写的)标签。对于函数，该标签的作用类似于变量，而对于宏类似于字符串替换。该参数的值可以在函数或宏体中使用变量符号访问。

```

function(func myArg)
message("myArg = ${myArg}")
endfunction()

macro(macr myArg)
message("myArg = ${myArg}")
endmacro()

func(foobar)
macr(foobar)

```

对 `func()` 和 `macr()` 的调用，会有相同的输出：

```

myArg = foobar

```

除了命名参数之外，函数和宏还有一组自定义变量，允许处理命名参数之外的参数或代替命名参数：

### **ARGC**

传递给函数的参数总数。它计算已命名参数和给定的其他未命名参数的数量。

### **ARGV**

这是一个列表变量，包含传递给函数的每个参数，包括指定的参数和未命名参数。

### **ARGN**

和`ARGV`类似，只包含命名参数以外的参数(即可选的、未命名参数)。

除此之外，每个单独的参数都可以用一个 `ARG#` 形式的名称来引用，其中#是参数的编号(例如：`ARG1`、`ARG2`等)。其包括命名参数，所以第一个命名参数也可以通过引用`ARG1`来获取。

使用 `ARG...` 变量包括支持可选参数和实现可以处理任意数量项的命令。考虑一个定义可执行目标的函数，将该目标链接到某个库，并为其定义测试用例。编写测试用例时，经常会遇到这样的函数(将在第24章中介绍)。与其为每个测试用例重复这些步骤，函数允许只定义一次步骤，然后每个测试用例都变成简单的调用。

```
# Use a named argument for the target and treat all remaining
# (unnamed) arguments as the source files for the test case
function(add_mytest targetName)
    add_executable(${targetName} ${ARGN})
    target_link_libraries(${targetName} PRIVATE foobar)
    add_test(NAME ${targetName}
        COMMAND ${targetName})
)
endfunction()

# Define some test cases using the above function
add_mytest(smallTest small.cpp)
add_mytest(bigTest big.cpp algo.cpp net.cpp)
```

上面的例子说明了 `ARGN` 的特性，允许函数或宏接受可变数量的参数，但必须指定一组命名参数。但是，需要注意会导致意外行为的情况。因为宏的参数作为字符串替换而不是变量，如果使用 `ARGN` 引用变量名，变量将在宏观的范围有效。下面的例子说明了这种情况：

```
# WARNING: This macro is misleading
macro(dangerous)
    # Which ARGN?
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endmacro()

function(func)
    dangerous(1 2)
endfunction()

func(3)
```

输出如下：

```
Argument: 3
```

`LISTS` 关键字与 `foreach()` 一起使用时，必须给定一个变量名，但宏提供的 `ARGN` 不是一个变量名。当从另一个函数内部调用宏时，宏最终使用该函数的 `ARGN` 变量，而不是宏本身的 `ARGN`。将宏体的内容直接粘贴到调用它的函数中时(这是CMake会做的)，情况就变得明了：

```
function(func)
    # Now it is clear, ARGN here will use the arguments from func
    foreach(arg IN LISTS ARGN)
        message("Argument: ${arg}")
    endforeach()
endfunction()
```

这种情况下，可以考虑将宏改为函数，或者如果必须保持为宏，则避免将参数视为变量。对于上面的示例，可以将`dangerous()`的实现改为使用 `foreach( IN ITEMS ${ARGN})` 的函数实现。

## 8.3. 关键字参数

上一节演示了如何使用 `ARG...` 变量来处理一组可变的参数。对于只需要一组变量或可选参数的简单情况，该功能就足够了，但如果必须支持多个可选或可选参数集，处理就会变得相当繁琐。此外，与支持关键字参数和灵活参数排序的CMake内置命令相比，上面处理参数的方式相对比较死板。看一下 `target_link_libraries()` 命令：

```
target_link_libraries(targetName
<PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
[<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
...
)
```

`targetName` 作为第一个参数，此后调用者可以以任何顺序提供任意数量的 `PRIVATE`、`PUBLIC` 或 `INTERFACE` 部分，每个部分允许包含任意数量的参数。用户定义的函数和宏也可以通过使用 `cmake_parse_arguments()` 命令支持同样的灵活性：

```
include(CMakeParseArguments) # Needed only for CMake 3.4 and earlier
cmake_parse_arguments(prefix
noValueKeywords
singleValueKeywords
multiValueKeywords
argsToParse)
```

`cmake_parse_arguments()` 命令过去是由 `CMakeParseArguments` 模块提供的，但在 `CMake 3.5` 中成为了一个内置命令。`include(CMakeParseArguments)` 行在 `CMake 3.5` 及以后版本中什么也不做，而在 `CMake` 的早期版本中，它将定义 `cmake_parse_arguments()` 命令(参见第11章了解更多关于 `include()` 的用法)。上面的使用方式是为了确保无论使用什么版本的 `CMake`，命令都可用。

`cmake_parse_arguments()` 接受 `argsToParse` 提供的参数，并根据指定的关键字集处理它们。通常，`argsToParse` 指定为 `ARGVN`，传递给函数或宏的一组未命名参数。每个关键字参数都是该函数或宏支持的关键字名称的列表，因此都应该用引号括起来，以确保解析的正确性。

`noValueKeywords` 定义为独立的关键字参数，其作用类似于开关。`singleValueKeywords` 在使用时关键字之后需要一个额外的参数，而 `multiValueKeywords` 在关键字之后需要零个或多个额外的参数。关键字惯例都是大写的，单词之间用下划线分隔。要注意，关键字不应该太长，否则会很麻烦。

`cmake_parse_arguments()` 返回时，对于每个关键字，将有一个对应的变量可用，其名称由指定的前缀、下划线和关键字名称组成。例如，如果前缀为 `ARG`，则对应于名为 `FOO` 的关键字的变量将是 `ARG_FOO`。如果某个关键字没有出现在 `argsToParse` 中，则对应的变量将为空。下面展示了三种不同的关键字类型是如何定义和处理的：

```

function(func)
    # Define the supported set of keywords
    set(prefix ARG)
    set(noValues ENABLE_NET COOL_STUFF)
    set(singleValues TARGET)
    set(multiValues SOURCES IMAGES)

    # Process the arguments passed in
    include(CMakeParseArguments)
    cmake_parse_arguments(${prefix}
        "${noValues}"
        "${singleValues}"
        "${multiValues}"
        ${ARGN})

    # Log details for each supported keyword
    message("Option summary:")
    foreach(arg IN LISTS noValues)
        if(${prefix}_${arg})
            message(" ${arg} enabled")
        else()
            message(" ${arg} disabled")
        endif()
    endforeach()

    foreach(arg IN LISTS singleValues multiValues)
        # Single argument values will print as a simple string
        # Multiple argument values will print as a list
        message(" ${arg} = ${prefix}_${arg}")
    endforeach()
endfunction()

# Examples of calling with different combinations
# of keyword arguments
func(SOURCES foo.cpp bar.cpp TARGET myApp ENABLE_NET)
func(COOL_STUFF TARGET dummy IMAGES here.png there.png gone.png)

```

输出如下：

```

Option summary:
ENABLE_NET enabled
COOL_STUFF disabled
TARGET = myApp
SOURCES = foo.cpp;bar.cpp
IMAGES =
Option summary:
ENABLE_NET disabled
COOL_STUFF enabled
TARGET = dummy
SOURCES =
IMAGES = here.png;there.png;gone.png

```

与使用命名参数和/或 `ARG_` 变量的参数相比，`cmake_parse_arguments()` 有很多优点。

- 基于关键字，可读性提高了，因为参数可以自解释。其他开发人员通常不需要查看函数实现或文档来理解每个参数的含义。
- 调用者可以选择参数的顺序。
- 调用者可以省略不需要的参数。
- 每个关键字都必须传递给 `cmake_parse_arguments()`，并且在函数顶部调用，这样函数支持的参数就非常清楚了。

- 基于关键字的参数的解析由 `cmake_parse_arguments()` 处理，不是通过临时、手动编码的解析器，因此消除了参数解析的错误。

## 8.4. 作用域

函数和宏之间的根本区别是，函数引入了新变量范围，而宏没有。函数内定义或修改的变量对函数外的同名变量没有影响。就变量而言，函数本质上是在一个沙箱中，这与宏不同，宏与调用者共享相同的变量作用域。注意，函数不引入新的策略范围(请参阅12.3节)。

与C/C++不同，CMake函数和宏不支持直接返回值。此外，由于函数引入了变量作用域，因此没有简单的方法将信息传递回调用者，但事实并非如此。与7.1.2节中的 `add_subdirectory()` 相同，“Scope”也可以用于函数。`set()` 命令的 `PARENT_SCOPE` 关键字可以用于修改调用作用域中的变量，而不是函数中的局部变量。虽然这与从函数返回值不同，但它确实允许将值传递回调用者。

常见的方法是允许变量名作为函数参数，调用者传递的参数仍在控制变量，函数结果的名称。这是 `cmake_parse_arguments()` 的使用方法，其前缀参数确定的所有变量名称需要在调用者的控制范围内。下面的例子演示了如何实现：

```
function(func resultVar1 resultVar2)
  set(${resultVar1} "First result" PARENT_SCOPE)
  set(${resultVar2} "Second result" PARENT_SCOPE)
endfunction()

func(myVar otherVar)
message("myVar: ${myVar}")
message("otherVar: ${otherVar}")
```

输出如下：

```
myVar: First result
otherVar: Second result
```

另一种方法是让函数记录设置的变量，而不是让调用者指定。这不太好，这样降低了函数的灵活性，并可能导致变量名冲突。最好使用上述方法让调用者控制正在设置或修改的变量名。

宏的处理方式与函数相同，通过将变量作为参数传入来指定要设置的变量的名称。唯一的区别是 `PARENT_SCOPE` 关键字不应该在宏中使用，因为它会修改了调用者作用域中的变量。宏会影响每次 `set()` 调用的调用范围，而函数只会在 `PARENT_SCOPE` 显式给定给 `set()` 时影响使用范围。

第7.3节中，讨论了 `return()` 作为提前结束文件或函数中的处理的一种方法。`return()` 不返回值，只返回父作用域。如果在函数中调用 `return()`，处理程序会立即返回给调用者，即跳过函数的其余部分。另一方面，`return()` 在宏中的行为不同。因为宏没有引入新的作用域，所以 `return()` 语句的行为取决于调用宏的位置。宏的任何 `return()` 语句实际上都是从宏的作用域返回，而不是从宏本身返回。考虑下面的例子：

```

macro(inner)
    message("From inner")
    return() # Usually dangerous within a macro
    message("Never printed")
endmacro()

function(outer)
    message("From outer before calling inner")
    inner()
    message("Also never printed")
endfunction()

outer()

```

输出如下：

```

From outer before calling inner
From inner

```

为什么第二个函数没有打印，原因就是在宏中使用了 `return`：

```

function(outer)
    message("From outer before calling inner")

    # === Pasted macro body ===
    message("From inner")
    return()
    message("Never printed")
    # === End of macro body ===

    message("Also never printed")
endfunction()

outer()

```

对于 `return()` 导致离开函数的原因已经清楚了，即使最初是从宏内部调用的。由于宏不创建自己的作用域，`return()` 的结果通常不是预期的。

## 8.5. 命令重载

当 `function()` 或 `macro()` 用来定义新命令时，如果命令名称已经存在，那么(没有文档记录)CMake的行为就是使用旧命令(除了前面有一个下划线)。开发人员有时会利用这种行为，包装现有的命令，像这样：

```

function(someFunc)
    # Do something...
endfunction()

# Later in the project...
function(someFunc)
    if(...)
        # Override the behavior with something else...
    else()
        # WARNING: Intended to call the original command, but it is not safe
        _someFunc()
    endif()
endfunction()

```

如果该命令仅这样覆盖一次，那么没什么问题。但是如果再次覆盖，原始命令将不可再访问。前一个命令的前缀“保存”只适用于当前名称，不会递归应用于以前的所有重写。这有可能导致无限递归，如下面的例子所示：

```
function(printme)
    message("Hello from first")
endfunction()

function(printme)
    message("Hello from second")
    _printme()
endfunction()

function(printme)
    message("Hello from third")
    _printme()
endfunction()

printme()
```

人们会天真地期望输出如下：

```
Hello from third
Hello from second
Hello from first
```

但是，第一个实现从未被调用，因为第二个实现最终在一个无限循环中调用自己。**CMake**处理过程中，会发生以下情况：

1. `printme` 的第一个创建并作为同名的可用命令。以前不存在该名称的命令，因此不需要其他操作。
2. 遇到 `printme` 的第二个实现。**CMake**通过该名称找到一个现有命令，因此将名称定义为 `_printme` 以指向旧命令，并将 `printme` 设置为指向新定义。
3. 遇到了 `printme` 的第三个实现。**CMake**发现与现有的命令的名称重复，重新定义了名字 `_printme` 指向旧的命令(这是第二个实现)和 `printme` 设置为指向新定义。

当调用 `printme()` 时，执行进入第三个实现，该实现调用 `_printme()`。这进入第二个实现，也调用 `_printme()`，但是 `_printme()` 再次指向第二个实现和无限递归结果，永远不会到达第一个实现。

通常，只要函数或宏不像上面讨论的那样调用前面的实现，就可以重写它。项目应该简单地假设新实现会取代旧实现，旧实现认为不再可用。

## 8.6. 总结

函数和宏是在整个项目中重用**CMake**代码的方法。通常，最好使用函数而不是宏，因为在函数中使用新变量的作用域可以更好地隔离该函数对调用作用域的影响。宏一般只应该在宏主体的内容，确实需要在调用者的范围内执行的情况下使用。要避免意外行为，还应避免从宏内部调用 `return()`。

除了非常简单的函数或宏之外，强烈建议使用 `cmake_parse_arguments()` 提供的基于关键字参数的处理。这将带来更好的可用性和健壮性(例如，不会混淆参数)。还允许在未来更容易地扩展函数，因为不依赖参数的顺序，或者始终提供所有参数(即使不相关)。

与函数和宏不同，一种常见的做法是指定一个特定的目录(通常在项目的顶层之下)，其中可以收集各种 `XXX.cmake` 文件。该目录就像一个功能目录，可以方便地从项目中的任何地方访问。每个文件都可以提供适当的函数、宏、变量和其他特性。使用`.cmake`文件名后缀允许 `include()` 命令查找作为模块的文件，这将在第

11章中详细介绍。它还允许IDE工具识别文件类型并用CMake语法高亮显示。

不要定义或调用名称以单个下划线开头的函数或宏。特别是，当函数或宏重新定义现有命令时，不要依赖于未文档化的行为，即命令的旧实现通过这样的名称可用。一旦命令重写超过一次，那原始实现就不可再访问。这个未文档化的行为甚至可能在未来的CMake版本中删除，所以不应该使用。类似地，不要覆盖任何内置CMake命令，将它们视为禁止，这样项目将能够确定内置命令的行为符合官方文档，并且不会有使原始命令变得不可访问。

# 第9章：属性

属性几乎影响构建的方方面面，从源文件如何编译，到二进制文件在打包安装程序中的安装位置。它们会附加到一个特定的实体，无论是目录、目标、源文件、测试用例、缓存变量，甚至是整个构建过程本身。属性不像变量那样保存独立的值，而是提供特定的附加信息。

对于CMake的新手来说，属性有时会与变量混淆。虽然在功能和特性方面，两者看起来很相似，但属性的用途却截然不同。变量不依附于任何特定的实体，项目定义和使用变量是很常见的。将其与CMake通常定义良好并记录在案的属性进行比较，这些属性总是适用于特定实体。造成两者混淆的可能原因是属性的默认值有时是由变量提供的。CMake用于相关属性和变量的名称通常遵循相同的模式，即变量名是 `CMAKE_` 为前缀的属性名。

## 9.1. 通用命令属性

CMake的命令提供了许多用于操作属性。其中通用的是 `set_property()` 和 `get_property()`，可以在任何类型的实体上设置和获取任何属性。这些命令要求将实体的类型，指定为命令参数以及一些特定的信息。

```
set_property(entitySpecific  
[APPEND] [APPEND_STRING]  
PROPERTY propName [value1 [value2 [...]]])
```

`entitySpecific` 定义了设置属性的实体。它必须是以下属性之一：

```
GLOBAL  
DIRECTORY [dir]  
TARGET [target1 [target2 [...]]]  
SOURCE [source1 [source2 [...]]]  
INSTALL [file1 [file2 [...]]]  
TEST [test1 [test2 [...]]]  
CACHE [var1 [var2 [...]]]
```

上面每个词的第一个词定义了设置属性的实体类型。`GLOBAL` 表示构建本身，因此不需要特定的实体名称。对于 `DIRECTORY`，如果没有名为 `dir` 的目录，则使用当前源目录。对于所有其他类型的实体，可以列出该类型任意数量的项。

`PROPERTY` 关键字将剩下的参数标记为定义属性名和值。`propName` 通常会匹配CMake文档中定义的属性，值是特定于属性的。除了CMake定义的属性外，还允许项目创建新的属性。这些特定于项目的属性意味着什么，以及如何影响构建，这取决于项目。如果这样做，对于项目来说，明智的做法是在属性名称上使用一些特定于项目的前缀，以避免与CMake或其他第三方包定义的属性发生名称冲突。

使用 `APPEND` 和 `APPEND_STRING` 关键字来控制已有(命名)属性的更新方式。如果两个关键字都没有指定，则将值替换为之前的(任意)值。`APPEND` 关键字可以将行为的附加属性值累加成一个列表，而 `APPEND_STRING` 关键字需要现有值和附加的新值，这里连接两个字符串，而不是一个列表(参见下面还特别注意继承属性进一步)。下表展示了不同之处：

之前值	新值	无关键字	<code>APPEND</code>	<code>APPEND_STRING</code>
foo	bar	bar	foo;bar	foobar
a;b	c;d	c;d	a;b;c;d	a;bc;d

`get_property()` 的参数列表如下:

```
get_property(resultVar entitySpecific  
PROPERTY propName  
[DEFINED | SET | BRIEF_DOCS | FULL_DOCS])
```

`PROPERTY` 关键字是必需的，并且后面要跟检索属性的名称。检索的结果存储在变量中，该变量的名称由 `resultVar` 给出。`entitySpecific` 部分类似于 `set_property()`，必须是以下内容之一：

```
GLOBAL  
DIRECTORY [dir]  
TARGET target  
SOURCE source  
INSTALL file  
TEST test  
CACHE var  
VARIABLE
```

与前面一样，`GLOBAL` 将构建作为整体来引用。`DIRECTORY` 可以与指定目录一起使用，也可以不指定特定目录，如果没有提供目录，则默认为当前源目录。大多数作用域，必须命名该作用域内的特定实例，并将检索附加到该实例的属性请求中。

`VARIABLE` 略有不同，变量名指定为 `propName`，而不是附加到 `VARIABLE` 关键字上。这有点不直观，可以考虑一下如果变量使用 `VARIABLE` 关键字命名为实体。在这种情况下，属性名没有指定，将 `VARIABLE` 看作指定作用范围可能会有所帮助。

如果没有提供任何可选关键字，则检索命名属性的值，这是 `get_property()` 命令的使用方式。`VARIABLE` 作用域与 `get_property()` 使用比较少，变量值可以直接通过 `$(...)` 获得，它比使用 `get_property()` 更清晰、更简单。

可选关键字可用于检索有关属性的详细信息，而不仅仅是值：

#### **DEFINED**

检索的结果是布尔值，确定命名的属性是否定义。在 `VARIABLE` 的作用域内查询，只有在指定的变量使用 `define_property()` 显式定义时，结果才为真。

#### **SET**

检索的结果将是布尔值，确定命名的属性是否已经设置。这是比 `DEFINED` 更强的测试，测试命名的属性是否已经给定了值(值本身是不相关的)。

#### **BRIEF\_DOCS**

检索已命名属性的字符串描述。如果没有为该属性定义字符串描述，结果将是字符串 `NOTFOUND`。

#### **FULL\_DOCS**

检索已命名属性的完整文档描述。如果没有为该属性定义字符串描述，结果将是字符串 `NOTFOUND`。

可选关键字中，除了 `SET` 外的所有关键字都用的不多，除非项目显式地调用 `define_property()` 来填充实例。这个命令还有以下形式：

```
define_property(entityType  
PROPERTY propName [INHERITED]  
BRIEF_DOCS briefDoc [moreBriefDocs...]  
FULL_DOCS fullDoc [moreFullDocs...])
```

这个命令不设置属性的值，只有文档和是否从别处继承(如果它尚未确定)。`entityType` 必须为 `GLOBAL`、`DIRECTORY`、`TARGET`、`SOURCE`、`TEST`、`VARIABLE` 或 `CACHED_VARIABLE` 之一，并使用 `propName` 指定属性定义。尽管与 `get_property()` 一样，`VARIABLE` 变量名可以指定为 `propName`，但没有指定任何实例。简短描述通常应该为单行，而完整的文档可以更长或多行。

如果在定义属性时使用 `INHERITED` 选项，并且该属性没有在指定的范围中设置，`get_property()` 命令将连接到父作用域，例如：请求了一个目录属性，但没有为指定的目录设置该属性，父目录作用域的属性将在目录作用域结构中递归查询，直到找到该属性或到达源的顶层。如果在顶级目录中仍然找不到，将会搜索全局作用域。类似地，如果 `TARGET`、`SOURCE` 或 `TEST` 属性要求但没有设置为指定的实例，该目录将设定为搜索范围(包括递归目录结构和 `GLOBAL` 范围，如果必要的话)。因为 `VARIABLE` 或 `CACHE` 已经根据涉及到父变量范围，所以没有提供这样的连接父作用域的功能。

`INHERITED` 属性的继承行为仅适用于特定属性类型的 `get_property()` 及类似 `get_...` 的函数。当使用 `set_property()` 设置 `APPEND` 或 `APPEND_STRING` 时，只考虑属性的当前值(即在追加值时不发生继承)。

**CMake** 对每种类型都有大量预定义属性。开发人员应查阅**CMake**参考文档，了解可用属性及其预期用途。后面的章节中，我们将讨论这些属性，以及与其他**CMake**命令、变量和特性。

## 9.2. 全局属性

全局属性与整体构建相关。它们通常用于修改构建工具，如何启动或工具行为的其他方面、定义项目文件如何构造的方面，以及提供某种程度的构建级别的信息。

除了 `set_property()` 和 `get_property()` 可以操作属性外，**CMake** 还提供了 `get_cmake_property()` 用于查询全局实例。它不仅仅是 `get_property()` 的简写，还可以用于检索任何具有全局属性的值。

```
get_cmake_property(resultVar property)
```

像 `get_property()` 一样，`resultVar` 是一个变量的名称，当命令返回时，请求的属性的值将存储在该变量中。属性参数可以是任何全局属性的名称，也可以是以下属性之一：

### VARIABLES

返回所有常规(而非缓存)变量的列表。

### CACHE\_VARIABLES

返回所有缓存变量的列表。

### COMMANDS

返回所有已定义的命令、函数和宏列表。命令由**CMake**预定义，而函数和宏可以由**CMake**(通常通过模块)或项目本身定义。一些返回的名称会对应于未文档化的或不打算直接用于项目的内部实例。返回的名称可能与最初定义的名称的大小写不同。

### MACROS

返回已定义宏的列表。这将是属性返回 `COMMANDS` 的子集，但请注意名称的大小写可能与属性报告的 `COMMANDS` 不同。

### COMPONENTS

返回由 `install()` 命令定义的所有组件列表，会在第25章中介绍。

从技术上讲，这些只读的属性不是全局属性(例如，不能使用 `get_property()` 检索它们)，但是理论上非常相似。这些属性只能通过 `get_cmake_property()` 检索。

## 9.3. 目录属性

目录还有自己的属性集，目录属性位于作用域的全局属性，并且只影响单个目标的属性。因此，目录属性主要关注于设置目标属性的默认值和覆盖全局属性，或当前目录的默认值。一些只读目录属性也提供了一定的信息，保存有关构建如何到达目录、在此点定义了哪些内容等信息。

方便起见，**CMake**提供了用于设置和获取目录属性的专属命令。设置命令的定义如下：

```
set_directory_properties(PROPERTIES prop1 val1 [prop2 val2 ...])
```

这个特定于目录的设置命令缺少 `APPEND` 或 `APPEND_STRING` 选项。只能用于设置或替换属性，不能直接添加到现有属性。与通用的 `set_property()` 相比，此命令的限制是只适用于当前目录。项目可以选择在方便的地方使用这种方式，而在其他地方使用通用方式，或者为了一致性，任何地方都使用通用方式。没有正确不正确的問題，只是个人偏好的问题。

指定目录属性获取命令有两种形式：

```
get_directory_property(resultVar [DIRECTORY dir] property)
get_directory_property(resultVar [DIRECTORY dir] DEFINITION varName)
```

第一种形式用于从特定目录或从当前目录(如果没有使用 `DIRECTORY` 参数)获取属性的值。第二种形式检索变量的值，看起来不是那么有用，但是提供了一种从当前目录范围以外的其他目录范围(当使用目录参数时)获取变量值的方法。实践中，第二种形式很少需要，应避免调试以外的场景使用。

对于 `get_directory_property()` 的任何一种形式，如果使用了 `DIRECTORY` 参数，则指定的目录必须处理过。**CMake**不可能知道尚未遇到目录的作用域属性。

## 9.4. 目标属性

**CMake**中很少有东西能如此直接地影响目标如何构建为目标属性，通过编译选项控制编译源文件，从而构建指定位置上的二进制类型文件和中间文件。一些目标属性会影响IDE中的显示，而其他属性会影响编译/链接时使用的工具。简而言之，目标属性是收集将源文件实际转换为二进制文件细节的地方。

**CMake**中已经有许多用于操作目标属性的方法。除了通用的 `set_property()` 和 `get_property()` 命令，**CMake**还为特定目标提供了一些等价操作：

```
set_target_properties(target1 [target2...]
PROPERTIES
propertyName1 value1
[propertyName2 value2] ... )
get_target_property(resultVar target propertyName)
```

`set_target_properties()` 缺乏 `set_property()` 的灵活性，但为常见情况提供了更简单的语法。`set_target_properties()` 命令不支持直接将属性添加到现有属性中，如果需要为给定的属性提供列表值，需要在 `set_target_properties()` 命令中以字符串形式指定该值，例如：“`this;is;a;list`”。

`get_target_property()` 命令是 `get_property()` 的简化版本。它只关注于提供一种简单的方法来获取目标属性的值，是通用命令的简写版本。

除了通用和特定于目标的属性获取器和设置器之外，**CMake**还有许多修改目标属性的命令。特别是 `target_...` 命令是**CMake**的一个关键部分，除了琐碎的**CMake**项目之外，所有的**CMake**项目都会使用。这些命令不仅定义了特定的目标属性，还定义了如何将这些信息传播到与其相关的其他目标。第14章基本内容涵盖了这些命令，以及它们如何与其他目标属性进行关联。

## 9.5. 源码属性

**CMake**还支持对单个源文件设置属性。允许在文件的基础上对编译器标志进行细粒度操作。也允许告诉**CMake**或构建工具如何对待这个文件，如指示是否生成作为构建的一部分，用什么编译器，或选择与编译器无关的工具处理文件等等。

项目很少需要查询或修改源文件属性，但对于需要的情况，**CMake**提供专用的 `setter` 和 `getter` 命令，这使得任务更容易。这些命令与其他特定属性的 `setter` 和 `getter` 命令类似：

```
set_source_files_properties(file1 [file2...]
    PROPERTIES
    propertyName1 value1
    [propertyName2 value2] ... )
get_source_file_property(resultVar sourceFile propertyName)
```

依旧是是没有提供附加功能的 `setter`，而 `getter` 只是语法简写为通用的 `get_property()` 命令，并没有提供新的功能。

源码属性对相同目录范围内的目标可见。如果源文件的设置属性出现在不同的目录范围，目标不会看到改变的属性，因此编译，以及相应源文件将不受影响。源文件可以编译成多个目标，因此设置的任何源码属性都应该添加该源码的所有有意义目标。

开始使用源属性之前，开发人员应该了解实现细节。使用**CMake**生成器(特别是Unix **Makefiles**生成器)时，源和源属性之间的依赖性比预期的要强。特别是源属性用于修改特定源文件(而不是整个目标)的编译器标记，更改源的编译器标记会导致重新构建目标的所有源。这是**Makefile**设置中处理依赖项细节的限制，在**Makefile**中，测试每个单独源代码的编译器标志是否改变会带来极大的性能损失，因此**CMake**开发人员选择在目标级别实现依赖项。项目可能会通过细节，只是对一个或者两个源码文件用特定的编译器进行编译。在19.2节会讨论更好的替代方式，来应对源属性不同的副作用。

## 9.6. 缓存变量属性

缓存变量上的属性与其他属性的目的略有不同。大多数情况下，缓存变量属性更多的是在**CMake GUI**和基于控制台的**ccmake**工具中处理缓存变量，而不是以任何方式影响构建。也没有额外的命令可以操纵他们，所以通过 `set_property()` 和 `get_property()` 命令修改时，必须使用 `CACHE` 关键字。

第5.3节中讨论了缓存变量，反映在缓存变量的属性中。

- 每个缓存变量都有一个类型，类型必须是 `BOOL`、`FILEPATH`、`PATH`、`STRING` 或 `INTERNAL`。可以使用带有属性名称类型的 `get_property()` 获得此类型。类型会影响**CMake GUI**和**ccmake**在UI中显示缓存变量的方式，以及用于编辑值UI的类型。任何内部类型的变量都不会在**CMake GUI**或**ccmake**中显示。
- 可以使用 `mark_as_advanced()` 命令将缓存变量标记为高级(在**GUI**中默认不可见)，该命令实际上只是设置了布尔型 `ADVANCED` 缓存变量属性。**CMake GUI**和**ccmake**工具都提供了显示或隐藏高级缓存变量的选项，允许用户选择，是只关注主要的基本变量，还是查看全部的非内部变量。
- 缓存变量的描述字符串通常是在调用 `set()` 命令时设置，也可以使用 `HELPSTRING` 缓存变量属性修改或读取。这个描述字符串在**CMake GUI**中用作工具提示，在**ccmake**工具中用作单行的帮助提示。

- 如果缓存变量的类型是 `STRING`，**CMake GUI**将查找名为 `STRING` 的缓存变量属性。如果不是空的，应该是变量的有效值列表，然后**CMake GUI**会将该变量显示为这些值的组合框，而不是任意的文本输入的部件。**ccmake**中，在缓存变量上按输入顺序遍历。注意**CMake**并没有强制要求缓存变量必须是字符串属性中的一个值，这只是为了方便**CMake GUI**和**ccmake**工具。当**CMake**运行**configure**步骤时，仍然将缓存变量视为一个任意字符串，因此仍然可以在**CMake**命令行或通过项目中的 `set()` 命令为缓存变量提供任何值。

## 9.7. 其他属性类型

**CMake**还支持单个属性的测试，并提供了属性**setter**和**getter**命令的通常测试特定版本：

```
set_tests_properties(test1 [test2...]
PROPERTIES
propertyName1 value1
[propertyName2 value2] ... )

get_test_property(resultVar test propertyName)
```

这些只是通用命令的更简洁的版本，它们缺乏附加功能，但在某些情况下可能使用起来更方便。测试将在第24章中详细讨论。

**CMake**支持的另一种属性是针对安装文件。这些属性特定于打包类型，大多数项目通常不需要这些属性。

## 9.8. 总结

属性是**CMake**的重要组成部分。一系列命令能够设置、修改或查询各种类型的属性，其中一些属性影响了项目之间的依赖关系。

- 除了特殊的全局属性外，所有属性都可以使用 `set_property()` 进行操作，这使得开发人员可以预测它，并在需要时提供灵活的 `APPEND` 功能。具体属性可能在某些情况下更方便进行操作，比如允许多个属性设置，不过缺少 `APPEND` 功能可能引导一些项目只是用 `set_property()`。虽然常见的错误是使用特定于属性的命令替换而不是附加到属性值中，但这两种方法都不是正确的，只能算是个人偏好。
- 对于目标属性，强烈建议使用 `target_...()` 命令，而不是直接操作相关的目标属性。这些命令不仅可以操作特定目标上的属性，还可以在目标之间建立依赖关系，以便**CMake**可以自动传播一些属性。第14章会讨论这一系列的主题，强调了对 `target_...()` 命令的强烈偏好。
- 在编译器选项的控制级别上，源码属性提供了更细粒度的操作，这些对项目的构建行为有潜在影响。当编译选项只有少数源文件改变时，一些**CMake**生成器可能会重新构建一些不必要的东西。项目应该考虑使用其他可用的替代方案，如第19.2节提供的技术。

# 第10章：表达式

运行CMake时，开发人员认为这是一个简单的步骤，生成相关的一组特定于生成器的项目文件(例如：`Visual Studio`解决方案和项目文件，`Xcode`项目，`Unix makefile`或Ninja输入文件)。然而，这是两个不同的步骤。运行CMake时，输出日志的结尾通常是这样的：

```
-- Configuring done  
-- Generating done  
-- Build files have been written to: /some/path/build
```

调用CMake时，先读入并处理位于源顶层的`CMakeLists.txt`，再执行命令、函数等创建项目的内部表示，这为“配置步骤”。控制台日志的大多数输出都在此阶段生成，包括`message()`的内容。配置步骤的最后，将`-- configuration done`输出在日志中。

当CMake完成了对`CMakeLists.txt`文件的读取和处理，就会执行“生成步骤”，这是使用配置步骤中构建的内部表示，创建构建工具的项目文件的地方。大多数情况下，开发者通常会忽略“生成步骤”。控制台日志总是在配置步骤完成后，立即显示生成消息，但理解这两个阶段特别重要。

考虑为多配置CMake生成器(如Xcode或Visual Studio)处理的项目。当读取`CMakeLists.txt`文件时，CMake不知道将为哪个配置构建。多配置设置，有不止一个选择(如调试、发布等)。如果`CMakeLists.txt`文件想要做一些事情，比如：将一个文件复制到给定目标的最终可执行文件所在的目录中，这似乎就会出现问题，因为该目录的位置取决于正在构建的配置。需要某种占位符来告诉CMake“正在构建的配置，最终可执行文件的目录”。

这是生成器表达式提供的功能，提供了一种编码逻辑的方法，这些逻辑在配置时不会生成，会在生成阶段生成。这可以用来执行逻辑，输出字符串信息，比如构建目录，项目名称、平台和更多的细节。可以根据正在执行的构建或安装，提供不同的内容。

生成器表达式支持广泛，但也不是哪里都能使用。CMake参考文档中，如果特定的命令或属性支持生成器表达式，文档会提到。随着时间的推移，支持生成器表达式的属性集越来越多，一些CMake版本也扩展了表达式集。项目应该确认需要的最低CMake版本，以确保使用的CMake支持使用的生成器表达式。

## 10.1. 简单的布尔逻辑

生成器表达式使用`$<...>`指定，其中尖括号内的内容可以采用几种不同的形式。下面是最基本的生成器表达式：

```
$<1:...>  
$<0:...>  
$<BOOL:...>
```

对于`$<1:...>`，表达式的结果是`...`部分，而对于`$<0:...>`，将忽略`...`部分，表达式会产生空字符串。`$<BOOL:...>`表达式可将CMake的布尔值转换为0，其他都转换为1(关于CMake认为的错误值的细节，请参阅6.1.1节的讨论)。这些生成器表达式提供了简单而强大的方法。还支持逻辑操作：

```
$<AND:expr[,expr...]>  
$<OR:expr[,expr...]>  
$<NOT:expr>
```

每个表达式的值为1或0。`AND` 和 `OR` 表达式可以接受任意数量的用逗号分隔的参数，并提供相应的结果，但不接受单个表达式。由于 `AND`、`OR` 和 `NOT` 要求表达式的计算值仅为0或1，可以考虑将这些表达式包装在 `$<BOOL:...>` 中，强制执行对正确或错误表达式的逻辑。

CMake 3.8及以后版本中，使用 `$<IF:...>` 表达式也可以非常方便地表达 `if-then-else` 逻辑：

```
$<IF:expr,value1,value0>
```

通常，表达式的值必须为1或0。如果表达式计算结果为1，则结果为 `value1`。如果 `expr` 计算结果为0，则结果为 `value0`。CMake 3.8之前，等价逻辑必须用更详细的方式来表达，需要给出表达式两次：

```
$<expr:value1>$<$<NOT:expr>:value0>
```

生成器表达式可以嵌套，允许构造任意复杂度的表达式。上面的示例显示了嵌套条件，生成器表达式的任何部分都可以嵌套。下面的例子演示了到目前为止所讨论的特性：

表达式	结果	备注
<code>\$&lt;1:foo&gt;</code>	foo	
<code>\$&lt;0:foo&gt;</code>		
<code>\$&lt;true:foo&gt;</code>		
<code>\$&lt;\$&lt;BOOL:true&gt;:foo&gt;</code>	foo	
<code>\$&lt;\$&lt;NOT:0&gt;:foo&gt;</code>	foo	
<code>\$&lt;\$&lt;NOT:1&gt;:foo&gt;</code>		错误，不能是单个1或0
<code>\$&lt;\$&lt;NOT:true&gt;:foo&gt;</code>	foo	
<code>\$&lt;\$&lt;AND:1,0&gt;:foo&gt;</code>		
<code>\$&lt;\$&lt;OR:1,0&gt;:foo&gt;</code>	foo	错误，不能是单个1或0
<code>\$&lt;1:\$&lt;\$&lt;BOOL:false&gt;:foo&gt;&gt;</code>		
<code>\$&lt;IF:\$&lt;BOOL:\${foo}&gt;,yes,no&gt;</code>	<code>{\${foo}}</code> 中的内容将依赖于表达式 <code>yes</code> 和 <code>no</code> 中的内容	

与 `if()` 一样，CMake 提供对生成器表达式中字符串、数字和版本测试的支持，尽管语法略有不同。如果满足相应的条件，值为1，否则为0。

```
$<STREQUAL:string1,string2>
$<EQUAL:number1,number2>
$<VERSION_EQUAL:version1,version2>
$<VERSION_GREATER:version1,version2>
$<VERSION_LESS:version1,version2>
```

另一个条件表达式是测试构建类型：

```
$<CONFIG:arg>
```

如果 `arg` 对应于实际构建的构建类型，则该值将为1，对于其他构建类型，则为0。它的常见用途是仅为调试构建提供编译器标志，或者为不同的构建类型选择不同的实现。例如：

```

add_executable(myApp src1.cpp src2.cpp)

# Before CMake 3.8
target_link_libraries(myApp PRIVATE
$<$<CONFIG:Debug>;checkedAlgo>
$<$<NOT:$<CONFIG:Debug>>;fastAlgo>
)

# CMake 3.8 or later allows a more concise form
target_link_libraries(myApp PRIVATE
$<IF:$<CONFIG:Debug>,checkedAlgo,fastAlgo>
)

```

上面的代码会将可执行文件链接到 `checkedAlgo` 的调试库，以及用于所有其他类型的 `fastAlgo` 库。`$<CONFIG:>` 生成器表达式是唯一能够提供这种功能的方法，适用于所有CMake项目生成器，包括像Xcode或Visual Studio这样的多配置生成器。这会在13.2节中有更详细的介绍。

CMake提供了更多的基于平台和编译器细节、CMake策略设置等的条件测试。开发人员应该查阅CMake参考文档，以获得支持表达式的完整信息。

## 10.2. 目标的信息

生成器表达式的另一个常见的用法是提供目标的信息。目标的任何属性可以通过以下两种形式获得：

```

$<TARGET_PROPERTY:target,property>
$<TARGET_PROPERTY:property>

```

第一个方式提供指定目标的命名属性，而第二个方式通过生成器表达式检索目标的属性。

虽然 `TARGET_PROPERTY` 是一种非常灵活的表达式类型，但并不总是获取目标信息的最佳方式。CMake还提供了其他表达式，这些表达式提供了关于目标构建的二进制文件目录和文件名的详细信息。以下是最常见的 `TARGET_FILE` 生成器表达式集：

### `TARGET_FILE`

这将获取生成目标二进制文件的绝对路径和文件名，包括与平台相关的任何文件前缀和后缀(例如`.exe`、`.dylib`)。对于基于Unix的平台，其中动态库文件名中通常包含版本信息。

### `TARGET_FILE_NAME`

和 `TARGET_FILE` 一样，但不获取路径(即只提供文件名)。

### `TARGET_FILE_DIR`

与 `TARGET_FILE` 相同，但没有文件名。这是获取构建最终可执行文件或库的目录最健壮的方法。使用像Xcode或Visual Studio这样的多配置生成器时，值会有所不同。

以上三个 `TARGET_FILE` 表达式在后期构建步骤中，复制文件的自定义构建规则特别有用。除了 `TARGET_FILE` 表达式之外，CMake还提供了特定于库的表达式，只是在处理文件名前缀和/或后缀的细节略有不同。这些表达式的名称以 `TARGET_LINKER_FILE` 和 `TARGET SONAME FILE` 开头，但不像 `TARGET_FILE` 表达式那样常用。

支持Windows平台的项目，还可以获得目标PDB文件的详细信息。同样，这些主要用于自定义构建。

以 `TARGET_PDB_FILE` 开始的表达式遵循与 `TARGET_PROPERTY` 类似的模式，提供生成器表达式(目标的)PDB文件路径和文件名详细信息。

另一个与目标相关的生成器表达式也需要提一下。**CMake**允许将库目标定义为对象库，只是**CMake**与目标关联的对象文件的集合，并不会创建最终的库文件。因为是目标文件，所以不能作为单个单元进行链接(尽管**CMake 3.12**放松了这一限制)。相反，必须像添加源一样将它们添加到目标中。**CMake**会在链接阶段包括那些对象文件，就像编译目标源创建的对象文件一样。这要使用 `$<TARGET_OBJECTS:>` 生成器表达式完成的，该表达式适合于在使用 `add_executable()` 或 `add_library()` 时列出对象文件。

```
# Define an object library
add_library(objLib OBJECT src1.cpp src2.cpp)

# Define two executables which each have their own source
# file as well as the object files from objLib
add_executable(app1 app1.cpp $<TARGET_OBJECTS:objLib>)
add_executable(app2 app2.cpp $<TARGET_OBJECTS:objLib>)
```

上面的示例中，没有为 `objLib` 创建单独的库，但 `src1.cpp` 和 `src2.cpp` 源文件仍然只编译一次。可以避免创建静态库的时间成本，或者动态库的符号解析的时间成本，同时还可以避免多次编译相同的源。

## 10.3. 基本信息

生成器表达式可以提供的不仅是目标信息，还可以获取有关所使用的编译器、目标平台、构建配置名称等信息。这些表达式倾向于在更特殊情况下使用，例如：处理自定义编译器或解决特定于特定编译器或工具链的问题。这些表达式也会引起误用，比如：构造了一些东西的路径，而这些东西原本可以使用更健壮的方法(如使用 `TARGET_FILE` 表达式或其他**CMake**特性)获得。依赖更通用的信息生成器表达式作为解决问题的方法之前，开发人员应该考虑这些表达的意图。这里列出了一些比较常见的表达式和实际表达式，作为进阶阅读的起点：

```
$<CONFIG>
```

生成类型。使用 `CMAKE_BUILD_TYPE` 变量，并非适用于**Xcode**或**Visual Studio**这样的多配置生成器。早期的**CMake**使用了现在已经废弃的 `$<CONFIGURATION>` 表达式，现在项目使用 `$<CONFIG>`。

```
$<PLATFORM_ID>
```

标识目标构建平台。在交叉编译时非常有用，特别是构建多个平台时(例如：设备和模拟器构建)。这个生成器表达式与 `CMAKE_SYSTEM_NAME` 变量密切相关，项目应该考虑在特定情况下使用该变量。

```
$<C_COMPILER_VERSION>, $<CXX_COMPILER_VERSION>
```

只在某些情况下，如果编译器版本是比一些特定版本或更新。可以通过 `$<VERSION_???:>` 生成器表达式。例如，如果**C++**编译器版本小于4.2.0，要生成字符串 `OLD_COMPILER`，可以使用下面的表达式：

```
$<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
```

只有在已知编译器类型，且编译器的特定行为需要由项目以某种方式处理的情况下，才会使用这种表达式。这可能是一种有用的技术，但如果过于依赖此类表达式，则会降低项目的可移植性。

```
$<LOWER_CASE:>, $<UPPER_CASE:>
```

作为执行字符串比较之前的一个步骤，任何内容都可以通过这些表达式转换为大写或小写。例如：

```
$<STREQUAL:$<UPPER_CASE:${someVar}>,FOOBAR>
```

## 10.4. 总结

与其他功能相比，生成器表达式是CMake添加的新特性。正因为如此，网上关于CMake的很多资料都没有使用过，因为生成器表达式通常比旧的方法更健壮，通用性更强。一些常见的例子中出于好意，导致的逻辑只适用于受支持的项目生成器或平台，但是使用合适的生成器表达式就能消除这种限制。对于试图为不同构建类型做不同事情的项目来说，这一点尤为重要。因此，开发人员应该熟悉生成器表达式。上面提到的那些表达式只是CMake支持的子集，应该可以覆盖大多数开发人员可能面临的情况。

如果使用得当，生成器表达式可以生成更简洁的CMakeLists.txt文件。例如，根据构建类型有条件地包含源文件，可以相对简洁地完成，正如前面给出的\$<config:...>的示例所示。这样减少了if-then-else逻辑的数量，只要生成器表达式不复杂，会有更好的可读性。生成器表达式也非常适合处理根据目标或构建类型而变化的内容。

相反，当试图将所有东西都变成生成器表达式，可能导致表达式过于复杂，最终模糊了逻辑，难以调试。与以往一样，开发人员应该更喜欢清晰，而不是聪明的表达式，对于生成器表达式更是如此。首先CMake是否提供了专门的命令来实现同样的结果，各种CMake模块提供了针对特定第三方包或执行特定任务的针对性功能。还有各种各样的变量和属性可以简化或完全取代生成器表达式。花几分钟查阅CMake参考文档，可以节省大量必要的时间来构建复杂的生成器表达式。

# 第11章：模块

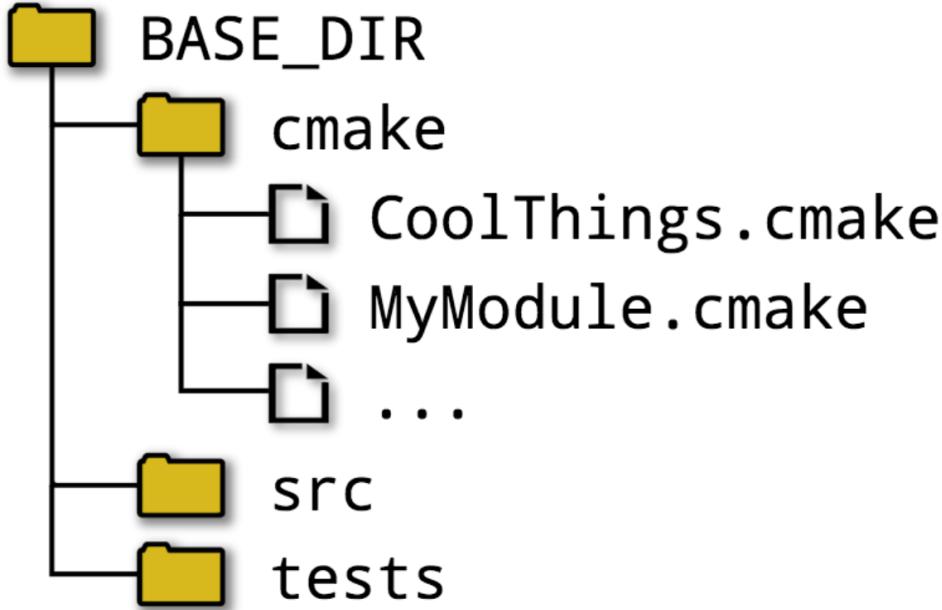
前面的章节主要关注CMake的核心方面。变量、属性、流控制、生成器表达式、函数等等，每个组件都可以认为是CMake语言的一部分。模块是在核心语言特性上构建的CMake代码预构块。提供丰富的功能，项目可以用来完成各种各样的目标。模块可以打包为普通的CMake代码，因此是可读的，模块也可以成为了解如何在CMake中完成工作的学习资源。

可以将模块收集在一起，作为CMake发布的一部分放在单独的目录中。项目可以以两种方式使用模块，直接使用，作为外部包使用。使用模块的直接方法是使用 `include()` 将模块代码注入当前作用域。这就像在7.2节中讨论的一样，只需提供模块的基本名称给 `include()` 即可，`include()` 的所有选项与以前完全相同。

```
include(module [OPTIONAL] [RESULT_VARIABLE myVar] [NO_POLICY_SCOPE])
```

当给定模块名称时，`include()` 命令将查找已定义的文件位置，该文件的名称是以`.cmake`为后缀(区分大小写)。例如：`include(FooBar)`，CMake会查找名为 `FooBar.cmake` 的文件。在区分大小写的系统中，比如 Linux，文件名为`foobar.cmake`则无法匹配。

查找模块文件时，CMake首先查询变量 `CMAKE_MODULE_PATH`。假设这是一个目录列表，CMake将按顺序搜索每个目录。将使用首先匹配的文件，如果没有找到匹配的文件，或者 `CMAKE_MODULE_PATH` 是空，CMake将在自己内部的模块目录中搜索。这个搜索顺序允许通过向 `CMAKE_MODULE_PATH` 添加目录来添加模块。一种模式是将项目模块文件收集到单个目录中，并将其添加到 `CMAKE_MODULE_PATH` 中顶层 `CMakeLists.txt` 文件开头附近的位置。下面的目录结构显示了这种方式：



相应的 `CMakeLists.txt` 文件只需要将 `cmake` 目录添加到 `CMAKE_MODULE_PATH` 中，然后可以在加载每个模块时使用文件名调用 `include()`。

`CMakeLists.txt`:

```

cmake_minimum_required(VERSION 3.0)
project(Example)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

# Inject code from project-provided modules
include(CoolThings)
include(MyModule)

```

**CMake**查找模块的搜索顺序有一个例外。如果调用 `include()` 的文件在**CMake**自己的内部模块目录中，查询 `CMAKE_MODULE_PATH` 之前将首先搜索内部模块目录。这可以防止项目代码(故意地)用自己的模块替换正式模块的行为。

使用模块的另一种方法是使用 `find_package()`。在23.5节中有讨论，目前该命令的简化形式没有任何关键字：

```
find_package(PackageName)
```

以这种方式使用时，除其行为与 `include()` 相似，**CMake**将搜索名为`FindPackageName`的文件，而不是`PackageName.cmake`。通过这种方法，外部包的详细信息可以引入到构建中，包括导入目标、定义文件、库或程序位置变量、可选组件的信息、版本信息等等。与 `find_package()` 相关联的选项和特性集要比 `include()` 和第23章提供的丰富得多。

本章的剩余部分将介绍一些有趣的模块，这些模块存在于**CMake**发布版中。这绝不是一个全面的集合，但确实提供了一些好用的功能。其他模块将在后面的章节中介绍，它们的功能与所讨论的主题密切相关。**CMake**文档提供了所有可用模块的完整列表，每个模块都有自己的帮助信息，解释了模块提供了什么，以及如何使用它。尽管文档的质量因模块的不同而有所不同，但也可以先了解一下。

## 11.1. 开发建议

**CMakePrintHelpers**模块提供了两个宏，这两个宏让开发过程中打印属性和变量更加方便。这两个宏并不打算永久使用，而旨在帮助开发人员快速、轻松地临时记录信息，以帮助调试项目。

```

cmake_print_properties(
[TARGETS target1 [target2...]]
[SOURCES source1 [source2...]]
[DIRECTORIES dir1 [dir2...]]
[TESTS test1 [test2...]]
[CACHE_ENTRIES var1 [var2...]]
PROPERTIES property1 [property2...]
)

```

这个宏实际上是将 `get_property()` 和 `message()` 组合到一起。准确地指定其中一种属性类型，并且列出实例，打印每个命名属性。在记录多个实例和/或属性信息时特别方便。例如：

```

add_executable(myApp main.c)
add_executable(myAlias ALIAS myApp)
add_library(myLib STATIC src.cpp)

include(CMakePrintHelpers)
cmake_print_properties(TARGETS myApp myLib myAlias
PROPERTIES TYPE ALIASED_TARGET)

```

输出如下所示：

```
Properties for TARGET myApp:  
myApp.TYPE = "EXECUTABLE"  
myApp.ALIASED_TARGET = <NOTFOUND>  
Properties for TARGET myLib:  
myLib.TYPE = "STATIC_LIBRARY"  
myLib.ALIASED_TARGET = <NOTFOUND>  
Properties for TARGET myAlias:  
myAlias.TYPE = "EXECUTABLE"  
myAlias.ALIASED_TARGET = "myApp"
```

该模块还提供了一个函数，来记录一个或多个变量的值：

```
cmake_print_variables(var1 [var2...])
```

这对所有变量都有效，不管是否在项目中进行设置。

```
set(foo "My variable")  
unset(bar)  
  
include(CMakePrintHelpers)  
cmake_print_variables(foo bar CMAKE_VERSION)
```

上面的输出内容如下：

```
foo="My variable" ; bar="" ; CMAKE_VERSION="3.8.2"
```

## 11.2. 字节顺序

使用嵌入式平台或项目时，最好了解一下系统的字节顺序。`TestBigEndian` 模块提供了 `test_big_endian()` 宏，该宏编译一个测试程序来确定系统的字节顺序。然后缓存此结果，以便后续CMake不用重做该测试。这个宏只有一个参数，它是一个变量的名字，用来存储布尔结果(`true` 意味着是大端系统)：

```
include(TestBigEndian)  
test_big_endian(isBigEndian)  
message("Is target system big endian: ${isBigEndian}")
```

## 11.3. 检查存在性和支持程度

CMake的模块涵盖更全面的领域，检查是否存在或支持。这类模块的工作方式基本相同，即编写少量测试代码，然后尝试编译、链接并运行结果，以确认代码中测试的内容是否支持。所有这些模块名称都以`Check`开头。

一些基本的`Check...`模块可以编译并链接到测试可执行文件中，然后返回成功/失败的结果。这些模块的名称都为`Check<LANG>SourceCompiles`，每个模块都有相关的宏执行测试：

```

include(CheckCSourceCompiles)
check_c_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckCXXSourceCompiles)
check_cxx_source_compiles(code resultVar [FAIL_REGEX regex])

include(CheckFortranSourceCompiles)
check_fortran_source_compiles(code resultVar [FAIL_REGEX regex] [SRC_EXT extension])

```

对于每个宏，代码参数应该是字符串，该字符串包含为所选语言生成可执行文件的源代码。尝试编译和链接代码的结果，缓存变量存储在 `resultVar` 中，`true` 表示成功，`false` 可以是空字符串、错误消息等。测试执行一次之后，后续的CMake运行将使用缓存的结果，而不再次执行测试。即使测试的代码被更改，也会出现这种情况，因此要强制重新求值，必须手动删除缓存变量。如果指定了 `FAIL_REGEX` 选项，则可以使用附加条件。如果测试编译和链接的输出与 `regex` 正则表达式匹配，即使代码成功编译和链接，检查也将视为失败。

```

include(CheckCSourceCompiles)
check_c_source_compiles(
    int main(int argc, char* argv[])
    {
        int myVar;
        return 0;
    }" nowarnUnused FAIL_REGEX "[Ww]arn"
if(nowarnUnused)
    message("Unused variables do not generate warnings by default")
endif()

```

对于Fortran，文件扩展名会影响编译器处理源文件的方式，因此可以使用 `SRC_EXT` 显式地指定文件扩展名，获得预期的行为。C或C++没有等效的选项。

调用任何编译测试宏来影响编译代码的方式之前，可以设置 `CMAKE_REQUIRED_...`

#### `CMAKE_REQUIRED_FLAGS`

相关的 `CMAKE<LANG>FLAGS` 和 `CMAKE<LANG>FLAGS_<CONFIG>` 变量需要传递给编译器，作为附加标志(参见14.3节)，这个字符串中对多个标志进行了分离。

#### `CMAKE_REQUIRED_DEFINITIONS`

每个定义以 `-DFOO` 或 `-DFOO=bar` 的形式指定。

#### `CMAKE_REQUIRED_INCLUDES`

指定搜索目录。多个路径必须指定为CMake列表，空格会视为路径的一部分。

#### `CMAKE_REQUIRED_LIBRARIES`

CMake的库列表添加到链接阶段。不要在库名前加上 `-l` 或类似的前缀，只提供库名或CMake导入的目标名称即可(第16章)。

#### `CMAKE_REQUIRED_QUIET`

如果存于此选项，宏将不会打印任何状态消息。

这些变量用于构造内部检查的 `try_compile()` 参数。`try_compile()` 的CMake文档讨论了检查可能会影响其他的变量，而 `try_compile()` 与工具链选择相关的相关行为，会在第21.5节中介绍。

除了检查代码是否可以构建，CMake提供了测试C或C++代码是否可以成功执行的模块。成功与否是通过可执行文件执行后的退出码来确定，0视为成功，其他值表示失败。

```
include(CheckCSourceRuns)
check_c_source_runs(code resultVar)

include(CheckCXXSourceRuns)
check_cxx_source_runs(code resultVar)
```

这些宏没有 `FAIL_REGEX` 选项，因为成功或失败由测试的退出码决定。如果不能构建代码，也会视为失败。所以使用 `check_c_source_compiles()` 和 `check_cxx_source_compiles()` 构建代码，也会产生同样的效果。

对于交叉编译到不同目标平台的构建，`check_c_source_runs()` 和 `check_cxx_source_runs()` 宏的行为完全不同。如果提供细节信息，可以在模拟器下运行代码，这会大大降低CMake的速度。如果没有提供模拟器信息，宏将通过变量提供预先确定的结果，而不会尝试运行任何东西。这个相当高级的主题在CMake的 `try_run()` 命令文档中介绍，这是宏内部执行检查时所使用的命令。

某些类型的检查非常常见，CMake为此提供了专门的模块。这些方法删除了定义测试代码的许多模板文件，并允许项目为检查指定最小的信息集。这些通常只是对 `Check<LANG>SourceCompiles` 模块所提供的宏进行包装，因此用于定制如何构建测试代码的那一组变量仍然可用。这些更专业的模块会检查编译器标记、预处理符号、函数、变量、头文件等信息。

可以使用 `Check<LANG>CompilerFlag` 来检查对特定编译器标志的支持，每个模块提供一个宏，名称遵循同一个模式：

```
include(CheckCCompilerFlag)
check_c_compiler_flag(flag resultVar)

include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(flag resultVar)

include(CheckFortranCompilerFlag)
check_fortran_compiler_flag(flag resultVar)
```

标志检查宏在内部更新 `CMAKE_REQUIRED_DEFINITIONS` 变量，以便在调用 `check<LANG>source_compiles()` 宏时使用简单的测试文件。失败正则表达式的内部集合也作为 `FAIL_REGEX` 选项传递，确定测试该标志是否会触发特定的断言消息。如果没有触发匹配的断言消息，调用的结果将是 `true`。这意味着，任何导致编译器警告(编译成功)，仍会视为失败。还有，这些宏假设在相关 `CMAKE<LANG>FLAGS` 变量中已经存在的任何标志(参见14.3节)本身不会让编译器产生任何编译警告。如果产生编译警告，那么每个标记测试宏的逻辑都将失败，所有检查的结果都将失败。

另外两个值得注意的模块是 `CheckSymbolExists` 和 `CheckCXXSymbolExists`。前者提供了宏来构建测试C代码的可执行文件，后者与测试C++代码的可执行文件做同样的事情。两者都检查特定符号是否作为预处理符号(即可以通过 `#ifdef` 语句进行测试的符号)、函数或变量。

```
include(CheckSymbolExists)
check_symbol_exists(symbol headers resultVar)

include(CheckCXXSymbolExists)
check_cxx_symbol_exists(symbol headers resultVar)
```

`header` 中指定的每一项(如果需要给出多个 `header`，则使用CMake列表)，相应的 `#include` 将添加到测试源代码中。大多数情况下，选中的符号将由这些头文件定义。测试的结果存储在 `resultVar` 缓存变量中。

使用函数和变量时，符号需要解析为可执行文件的一部分。如果函数或变量由库提供，则必须将该库作为测试的一部分进行链接，这可以使用 `CMAKE_REQUIRED_LIBRARIES` 变量完成。

```
include(CheckSymbolExists)
check_symbol_exists(sprintf stdio.h HAVE_SPRINTF)

include(CheckCXXSymbolExists)
set(CMAKE_REQUIRED_LIBRARIES SomeCxxSDK)
check_cxx_symbol_exists(SomeCxxInitFunc somecxxsdk.h HAVE_SOMECKXSDK)
```

这些宏可以检查的函数和变量种类有一些限制，只有那些满足预处理器符号命名的符号才能使用。`check_cxx_symbol_exists()` 的要求很严，所以只能检查全局名称空间中的非模板函数或变量，所以对任何作用域( `::` )或模板标记( `<>` )、预处理器符号都无效。就更不可能区分同一函数的不同重载，因此也不能检查重载。

其他模块会提供与 `CheckSymbolExists` 类似功能或功能子集。这些模块要么来自CMake的早期版本，要么用于C或C++以外的其他语言。`CheckFunctionExists` 模块已经弃用，并且 `CheckVariableExists` 模块提供了比 `CheckSymbolExists` 更多的内容。`CheckFortranFunctionExists` 模块可能对那些使用Fortran的项目有用，但请注意不存在 `CheckFortranVariableExists` 模块。Fortran项目可能希望使用 `CheckFortranSourceCompiles` 来保持一致性。

其他模块会提供更详细的检查。使用 `CheckStructHasMember` 可以测试结构成员，使用 `CheckPrototypeDefinition` 可以测试特定的C或C++函数原型，使用 `CheckTypeSize` 可以测试非用户类型的大小。其他更高级别的检查也是可能的，如 `CheckLanguage`、`CheckLibraryExists` 和各种 `CheckIncludeFile...` 模块都会提供支持。随着CMake的发展，进一步的检查模块会陆续的添加到CMake中，因此请参阅CMake模块文档以查看当前可用功能的完整集。

进行多次检查或执行检查时，需要在相互隔离或与当前范围于其他部分隔离的情况下进行，检查之前和之后手动保存和恢复状态可能会很麻烦。特别是各种 `CMAKE_REQUIRED_...` 变量需要保存和恢复。为了实现这一点，CMake提供了 `CMakePushCheckState` 模块，该模块定义了以下三个宏：

```
cmake_push_check_state([RESET])
cmake_pop_check_state()
cmake_reset_check_state()
```

这些宏允许将各种 `CMAKE_REQUIRED_...` 变量作为集合，并将状态推入或弹出堆栈。每次调用 `cmake_push_check_state()`，都会将 `CMAKE_REQUIRED_...` 变量(以及仅由 `CheckTypeSize` 模块使用的 `CMAKE_EXTRA_INCLUDE_FILES` 变量)设置为虚拟变量。`cmake_pop_check_state()` 则相反，丢弃 `CMAKE_REQUIRED_...` 变量的当前值，并恢复为前一个堆栈的值。`cmake_reset_check_state()` 宏可以清除所有 `CMAKE_REQUIRED_...` 变量，而 `cmake_push_check_state()` 的重置选项也可以清除一部分的变量。但CMake 3.10之前存在一个重置选项的bug，因此对于需要使用3.10之前版本的项目，最好使用 `cmake_reset_check_state()`。

```

include(CheckSymbolExists)
include(CMakePushCheckState)

# Start with a known state we can modify and undo later
cmake_push_check_state() # Could use RESET option, but needs CMake >= 3.10
cmake_reset_check_state() # Separate call, safe for all CMake versions
set(CMAKE_REQUIRED_FLAGS -Wall)
check_symbol_exists(FOO_VERSION foo/version.h HAVE_FOO)

if(HAVE_FOO)
    # Preserve -Wall and add more things for extra checks
    cmake_push_check_state()
    set(CMAKE_REQUIRED_INCLUDES foo/inc.h foo/more.h)
    set(CMAKE_REQUIRED_DEFINES -DFOOBXX=1)
    check_symbol_exists(FOOBAR "" HAVE_FOOBAR)
    check_symbol_exists(FOOBAZ "" HAVE_FOOBAZ)
    check_symbol_exists(FOOB00 "" HAVE_FOOB00)
    cmake_pop_check_state()
    # Now back to just -Wall
endif()

# Clear all the CMAKE_REQUIRED_... variables for this last check
cmake_reset_check_state()
check_symbol_exists(__TIME__ "" HAVE_PPTIME)

# Restore all CMAKE_REQUIRED_... variables to their original values
# from the top of this example
cmake_pop_check_state()

```

## 11.4. 其他模块

CMake对某些语言有出色的内置支持，特别是C和C++。它还包括许多模块，以可扩展和可配置的方式提供对语言的支持。通过定义相关的函数、宏、变量和属性，这些模块允许某些语言或与语言相关包在项目中可用。其中许多模块是作为对 `find_package()` 调用的支持提供的(参见第23.5节)，而其他模块旨在通过 `include()` 将内容引入当前作用域直接使用。下面的模块列表提供了可用的语言支持：

- CSharpUtilities
- FindCUDA (注意，在最近的CMake版本中，已经被CUDA的支持所取代，CUDA本身就是一种语言)
- FindJava, FindJNI, UseJava
- FindLua
- FindMatlab
- FindPerl, FindPerlLibs
- FindPython, FindPythonInterp
- FindPHP4
- FindRuby
- FindSWIG, UseSWIG
- FindTCL
- FortranCInterface

此外，还提供了与外部数据和项目交互的模块，这将在第27章中讨论。还提供了许多模块来进行测试和打包。这些工具与CMake套件的CTest和CPack工具有着密切关系，这些将在第24章和第26章中进行介绍。

## 11.5. 总结

**CMake**模块集合提供了丰富的功能。通过在特定目录下添加定制模块，并将该路径附加到 `CMAKE_MODULE_PATH` 变量中，项目可以轻松地扩展功能集。在 `include()` 调用中，应该优先使用 `CMAKE_MODULE_PATH`，而不是硬编码为绝对或相对路径，鼓励通用**CMake**逻辑进行解耦。这反过来使得重定位**CMake**模块到不同的目录作为一个项目拓展，或重用与不同的项目更加容易。实际上，组织构建自己的模块集合并不罕见，甚至将它们存储在单独目录中。通过在每个项目中适当地设置 `CMAKE_MODULE_PATH`，这些可重用的**CMake**构建块就可以根据需要进行使用了。

随着时间的推移，开发人员通常会接触到越来越多的场景，**CMake**模块可以为这些场景提供有用的快捷方式或现成的解决方案。有时，对可用模块的快速浏览可能会发现意想不到的奖励，或者新模块可能提供了一个更好维护的实现，而项目在此之前一直在以一种较差的方式实现。**CMake**的模块有潜在的开发人员池和跨不同平台的项目，这样的情况使得使用模块有很多好处，所以许多情况下，模块的方式可能提供一个更引人注目的替代方案。然而，不同模块的质量不同，有些模块在**CMake**早期就体现了价值，如果不跟上**CMake**或相关模块领域的变化，这些模块有时就会不那么有用。对于 `Find...` 模块来说尤其如此，它们不会像人们期望的那样紧密地跟踪软件包的最新版本。另一方面，模块只是普通的**CMake**代码，所以任何人都可以检查、学习、改进或更新，而不需要学习除了基本**CMake**以外的东西。事实上，对于希望参与**CMake**本身工作的开发人员来说，模块是一个很好的起点。

**CMake**提供的大量不同的 `Check...` 模块是一件喜忧参半的事情。开发人员可能会过于热衷于检查所有类型的东西，这可能会导致配置阶段的速度变慢，有时会导致一些问题。考虑在实现和维护检查的时间以及项目的复杂性方面，收益是否大于成本。有时，一些明智的检查就足以覆盖大部分情况，或者捕获可能导致以后难以跟踪问题的细微问题。此外，如果使用任何 `Check...` 模块，目标会将检查逻辑与调用的范围隔离开来。强烈建议使用 `CMakePushCheckState` 模块，但如果需要支持3.10之前的**CMake**版本，则尽可能避免使用 `cmake_push_check_state()` 的 `RESET` 选项。

# 第12章：策略

**CMake**已经发展了很长一段时间，引入了很多新的功能，修复了不少bug，并改变了某些特性以解决缺陷或引入改进。虽然新功能的引入不太可能给现有项目的构建带来问题，但如果依赖于老式行为，任何改变都有可能破坏项目。出于这个原因，**CMake**的开发人员小心地确保更改能够向后兼容，并为更新到新行为的项目提供一个直接的、可控的迁移方式。

对新旧行为的控制可以通过**CMake**策略机制完成。策略并不是开发者经常接触到的东西，当开发人员使用了另一个**CMake**版本，新的**CMake**版本有时会发布这样的警告，强调这个项目应该如何更新/使用新式的行为。

## 12.1. 政策控制

**CMake**的策略功能与 `cmake_minimum_required()` 命令紧密相连，该命令在第3章中介绍过。该命令不仅指定了项目所需的最低**CMake**版本，还将**CMake**的行为设置为与给定的版本相匹配。因此，当项目以 `cmake_minimum_required(VERSION 3.2)` 开始时，就表明至少需要**CMake 3.2**版本，并且该项目希望当前**CMake**的行为和3.2版本一样。开发者可以在任何时间点对**CMake**进行替换(向高版本)，并且项目仍然会像以前一样构建。

但是，有时项目需要比 `cmake_minimum_required()` 更细粒度的控制。考虑以下场景：

- 项目设置了较低的**CMake**版本，如果可能，也希望使用新式行为。
- 项目的一部分是无法修改(例如：可能来自一个外部代码库)，它依赖的旧式行为在**CMake**的新版本中已经修改。不过，项目的其余部分希望采用新式行为。
- 项目严重依赖于一些旧式行为，这些行为需要大量的成本进行更新。项目的一些部分想要利用最近的**CMake**特性，但是对于特定的更改，旧式行为需要保留，直到有时间来更新项目为止。

这些的示例中，仅 `cmake_minimum_required()` 命令提供的控制是不够的。可以通过 `cmake_policy()` 命令对策略进行更具体的控制，该命令有许多不同粒度的方式。在最粗粒度的方式与 `cmake_minimum_required()` 的效果非常接近：

```
cmake_policy(VERSION major[.minor[.patch[.tweak]]])
```

这种方式中，命令更改了**CMake**的行为以匹配指定版本的行为，`cmake_minimum_required()` 命令会隐式设置**CMake**的行为。除了项目顶部强制调用 `cmake_minimum_required()` 来强制最低**CMake**版本外，这两种方式在很大程度上可互换。除了顶级**CMakeLists.txt**文件的开始部分，使用 `cmake_policy()` 通常可以更清楚地传达意图，当项目需要对项目的某个部分执行特定版本的行为时，就如下面的例子所示：

```
cmake_minimum_required(VERSION 3.7)
project(WithLegacy)

# Uses recent CMake features
add_subdirectory(modernDir)

# Imported from another project, relies on old behavior
cmake_policy(VERSION 2.8.11)
add_subdirectory(legacyDir)
```

**CMake 3.12**(以向后兼容的方式)扩展了这个功能，允许项目指定一个版本范围，而不是一个版本，可以指定 `cmake_minimum_required()` 或 `cmake_policy(VERSION)`。范围是用三个点 ... 在最低和最高版本之间，没有空格。范围表示正在使用的**CMake**版本必须至少是最小值，要使用的行为应该是指定的最大值和正在运行的**CMake**

版本的最小值。这相当于项目表示，“我至少需要CMake X，但我可以安全地使用CMake Y的策略”。下面的例子展示了一个项目只需要CMake 3.7的两种方法，但是仍然支持CMake 3.12之前所有策略的新式行为(如果运行的CMake版本支持的话):

```
cmake_minimum_required(VERSION 3.7...3.12)
cmake_policy(VERSION 3.7...3.12)
```

3.12之前的CMake版本实际上只看到一个版本号，并忽略 ...3.12 部分，而 3.12 之后的版本可以理解为一个范围。

CMake还提供了能力集，可以单独的控制行为变化与 `SET` 的形式:

```
cmake_policy(SET CMPxxxx NEW)
cmake_policy(SET CMPxxxx OLD)
```

每个单独的行为更改都有自己的策略号 `CMPxxxx`，其中 `xxxx` 总是四位数。通过指定 `NEW` 或 `OLD`，项目告诉 CMake为特定策略使用 `NEW` 或 `OLD` 行为。CMake文档提供了策略的完整列表，并解释了每种策略的新旧行为。

3.0版本之前CMake允许项目将 `get_target_property()` 与不存在的目标一起使用。这种情况下，属性值返回为 `NOTFOUND`，而不是直接报错，但项目很可能包含不正确的逻辑。因此，从3.0版本开始，如果遇到这样的情况，CMake会在出现该错误时停止。如果一个项目依赖于旧式行为，可以继续使用策略**CMP0045**:

```
# Allow non-existent target with get_target_property()
cmake_policy(SET CMP0045 OLD)

# Would halt with an error without the above policy change
get_target_property(outVar doesNotExist COMPILE_DEFINITIONS)
```

为 `NEW` 操作制定策略就不那么常见了。一种情况是，项目想要设置一个较低的CMake版本，但如果使用的是较新的版本，仍然可以利用后续的特性。例如，在CMake 3.2中引入了策略**CMP0055**，提供对 `break()` 命令使用情况的检查。如果项目仍然想要支持在早期CMake版本中构建，在运行新的CMake时必须显式启用检查。

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

if(CMAKE_VERSION VERSION_GREATER 3.1)
    # Enable stronger checking of break() command usage
    cmake_policy(SET CMP0055 NEW)
endif()
```

测试 `CMAKE_VERSION` 变量是确定策略是否可用的一种方法，而 `if()` 命令提供了一种更直接的方法，`if(POLICY...)` 形式。上述方式也可以这样执行:

```
cmake_minimum_required(VERSION 3.0)
project(PolicyExample)

# Only set the policy if the version of CMake being used
# knows about that policy number
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

还可以获得特定策略的当前状态。策略设置可能在模块文件中，可能是由CMake本身或项目提供。然而，项目模块根据策略设置改变其行为很罕见。

```
cmake_policy(GET CMPxxxx outVar)
```

存储在 `outVar` 中的值将是旧的、新的或空的。`cmake_minimum_required(VERSION...)` 和 `cmake_policy(VERSION...)` 命令重置所有策略的状态。在指定CMake版本或更早版本引入的策略将重置为 `NEW`。在指定版本之后添加的策略将重置为空。

如果CMake检测到项目正在依赖一些于旧式行为、并与新式行为有冲突，或其行为含糊不清时，在相关策略未设置时会发出警告。这些警告是向开发人员展示CMake策略的最常见方式。它们的设计为噪音大且信息量大，其鼓励开发商更新项目以适应新的行为。某些情况下，即使策略已明确设置，也可能会发出弃用警告，但这通常只针对已弃用很长时间(许多版本)的策略。

有时不能立即处理策略警告，这些警告可能也不需要处理。处理此问题的首选方法是显式地将策略设置为所需的行为(旧式或新式)，这将关闭警告。但这中办法并不总是可行，比如项目的较深的部分调用了 `cmake_minimum_required(VERSION...)` 或 `cmake_policy(VERSION...)`，重新设置策略状态。作为解决这种情况的临时方法，CMake提供了 `CMAKE_POLICY_DEFAULT_CMPxxxx` 和 `CMAKE_POLICY_WARNING_CMPxxxx` 变量，其中 `xxxx` 是通常的四位数策略号。这些不由项目设置，而是由开发人员临时作为缓存变量来启用/禁用警告，或者检查项目是否在启用特定策略时发出警告。长期解决方案是解决上述警告所突显的根本问题。然而，对于项目来说，设置这些变量中的一个来屏蔽已知无害的警告是合适的。

## 12.2. 策略范围

有时策略设置只需要应用于文件的特定部分，所以CMake提供了一个策略堆栈，可以用来简化这个过程，而不需要手动保存任何想临时改变策略的现有值：

```
cmake_policy(PUSH)
cmake_policy(POP)
```

所有策略的现有状态可以通过 `PUSH` 操作保存，并通过相应的 `POP` 丢弃当前状态。每次 `PUSH` 都需要最终需要一个匹配的 `POP`。这两者之间，项目可以修改需要的任何策略设置，而不必显式地保存每个策略。同样，模块文件是策略堆栈操作的常见位置之一。为一个模块文件制定一些暂时策略：

```
# Save existing policy state
cmake_policy(PUSH)

# Set some policies to OLD to preserve a few old behaviors
cmake_policy(SET CMP0060 OLD) # Library path linking behavior
cmake_policy(SET CMP0021 OLD) # Tolerate relative INCLUDE_DIRECTORIES

# Do various processing here...

# Restore earlier policy settings
cmake_policy(POP)
```

有些命令隐式地将一个新的策略状态推送到栈上，在稍后某个定义的点再次弹出。一个例子是 `add_subdirectory()`，它在进入指定的子目录时将新的策略范围压入堆栈，并在命令返回时弹出它。`include()` 命令执行类似的操作，在开始处理指定的文件之前推入一个新的策略范围，并在处理完该文件后再次弹出该文件。`find_package()` 命令也对 `include()` 执行类似的操作，即在启动和完成与之相关的 `FindXXX.cmake` 模块文件的处理时，分别进行压入和弹出。

`include()` 和 `find_package()` 命令还支持 `NO_POLICY_SCOPE` 选项，该选项可以防止策略栈的自动push-pop(`add_subdirectory()` 没有这样的选项)。CMake的早期版本中，`include()` 和 `find_package()` 不会自动在策略栈上推入和弹出。添加 `NO_POLICY_SCOPE` 选项是为了让以后使用CMake版本的项目在特定部分恢复到旧的行为，但是通常不推荐这样用，而且对于新项目也没必要。

## 12.3. 总结

可能的情况下，项目应该更喜欢使用CMake版本级别的策略，而不是操作特定的策略。设置策略来匹配特定的CMake版本的行为，会使项目更容易理解和更新，而对单个策略的更改可能很难通过多个目录级别的跟踪，特别是与版本级策略更改的交叉使用，而版本级策略更改总是会重置。

选择如何指定要遵循的CMake版本时，`cmake_minimum_required(VERSION)` 和 `cmake_policy(VERSION)` 之间通常会选择后者。这方面的第一个例外是在项目的顶级CMakeLists.txt文件的开始，以及可以跨多个项目重用的模块文件的开始部分。对于第二个例外，最好使用 `cmake_minimum_required(VERSION)`，因为使用该模块的项目可能会强制执行最低CMake版本，但是该模块可能有自己特定的最小版本需求。除了这些情况，`cmake_policy(VERSION)` 通常能更清楚地表达了意图，但从策略的角度来看，这两个命令都能有效地实现相同的目的。

在项目确实需要操作特定策略的情况下，应该使用 `if(policy...)` 检查策略是否可用，而不是测试 `CMAKE_VERSION` 变量。比较以下两种设置策略行为的方法，并注意检查和执行如何使用策略一致的方法：

```
# Version-level policy enforcement
if(NOT CMAKE_VERSION VERSION_LESS 3.4)
    cmake_policy(VERSION 3.4)
endif()

# Individual policy-level enforcement
if(POLICY CMP0055)
    cmake_policy(SET CMP0055 NEW)
endif()
```

如果项目需要在本地操作多个单独的策略，应该使用 `cmake_policy(PUSH)` 和 `cmake_policy(POP)` 的包围该部分，以确保与作用域的其余部分隔离。特别注意可能退出该代码段的 `return()` 语句，并确保留下相应pop的push。还要注意的是，`add_subdirectory()`、`include()` 和 `find_package()` 都自动在策略堆栈上推入和弹出策略，所以不需要显式推入和弹出，除非需要为拉入的文件在本地更改策略设置。项目应该避免使用 `NO_POLICY_SCOPE` 关键字，因为它仅用于处理早期CMake版本的行为变化，现在已经很少使用于新项目了。

为了避免修改函数内的策略设置。函数没有引入新的策略范围，如果没有使用适当的push-pop正确地隔离更改，则策略更改可能会影响调用方。此外，函数实现的策略设置来自函数的范围，而不是调用函数的范围。因此，最好在定义函数的范围内调整策略设置，而不是在函数本身内。

作为最后的部分，允许开发人员或项目为解决一些具体政策的情况，使用 `CMAKE_POLICY_DEFAULT_CMPxxxx` 和 `CMAKE_POLICY_WARNING_CMPxxxx` 变量。开发人员可以使用它们临时更改特定策略设置的默认值，或者防止有关特定策略的警告。项目应该避免设置这些变量，以便开发人员能够在本地进行控制，但在某些情况下，即使调用 `cmake_minimum_required()` 或 `cmake_policy(VERSION)`，也可以使用它们来确保特定策略的行为或警告持续存在。不过，在可能的情况下，项目应该更新到新式行为，而非依赖于这些变量。

## 第二部分：深度构建

前几章中，我们介绍了CMake的基本，了解了核心语言特性、关键概念和重要的构建模块，为进一步探索CMake的功能提供了基础。

本书的第二部分中，构建生成将成为重点。章节涵盖了工具链和构建配置、不同类型的目标、执行自定义任务以及处理特定于平台的特性。对这些领域的理解就是简单且复杂与健壮且易于维护项目之间的分水岭。

# 第13章：构建类型

构建类型(某些IDE工具中也称为构建配置或构建方案)是一种高级控件，可以选择不同的编译器和链接器行为。构建类型是本章的主题，下一章将介绍控制编译器和链接器选项的具体细节。这两章节涵盖了每个CMake开发人员都会使用到的操作。

## 13.1. 构建类型的基本概念

构建类型会影响与构建有关的所有内容，主要对编译器和链接器行为有直接影响，也对项目使用的目录结构有影响，这反过来又会影响开发人员设置本地开发环境。

开发人员通常认为构建只有两种：调试或发布。对于调试，编译器标志用于记录信息，调试器可以使用这些信息将机器指令与源代码关联起来。这样的构建中，禁用优化以便在执行程序时，容易直接生成机器指令。另一方面，发布版本通常启用了优化，并且没有调试信息。

这些就是CMake构建类型的例子。当项目能定义构建类型时，CMake提供的默认构建类型对于大多数项目来说是足够的：

### **Debug**

由于禁用优化和调试信息，这种构建通常在开发和调试期间使用，通常提供最快的构建时间和最佳的交互调试。

### **Release**

这种构建类型通常提供了完全的优化，并且没有调试信息，不过一些平台在某些情况下可能生成调试符号。通常最终产品发布，使用该构建类型。

### **RelWithDebInfo**

这在某种程度上是前两者的混合。它的目标是提供接近发布版本的性能，但仍然允许某种程度的调试。当调试生成的性能甚至对于调试来说不可接受时，此构建类型非常有用。注意，RelWithDebInfo会默认禁用断言。

### **MinSizeRel**

这种构建类型通常只用于受限的资源环境，比如嵌入式设备。代码优化的大小而不是速度，而是调试信息。

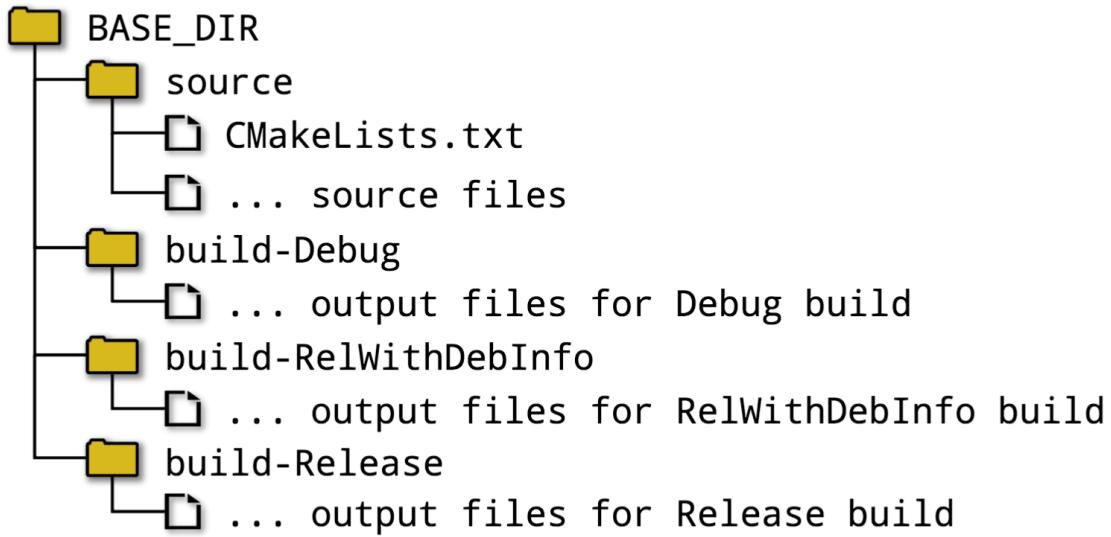
每种构建类型都会产生不同的编译器和链接器标志。还可能改变其他行为，比如：改变编译哪些源文件或链接到什么库。这些细节将在接下来的几节中讨论，在开始讨论之前，理解如何选择构建类型，以及如何避免一些常见问题。

### 13.1.1. 单配置生成器

第2.3节中，介绍了不同类型的项目生成器，如Makefile和Ninja，对每个构建目录只支持一种构建类型。对于这些生成器，必须通过设置`CMAKE_BUILD_TYPE`缓存变量来选择构建类型。例如，要使用Ninja配置并构建一个项目，可以使用如下命令：

```
cmake -G Ninja -DCMAKE_BUILD_TYPE:STRING=Debug ../source  
cmake --build .
```

`CMAKE_BUILD_TYPE` 缓存变量可以在CMake GUI中更改。与不同的构建类型之间切换不同，另一种策略是为每种构建类型设置单独的构建目录，所有这些目录仍然使用相同的源。目录结构可能像这样：



如果经常在构建类型之间切换，这种安排可以避免因为编译器标记的变化而重新编译相同的源代码。还可以使得单配置生成器充当多配置生成器的角色，像Qt Creator这样的IDE环境支持在构建目录之间切换，就像Xcode或Visual Studio允许在构建方案或配置之间切换一样简单。

### 13.1.2. 多配置生成器

一些生成器，特别是Xcode和Visual Studio，支持在单个构建目录中进行多种配置。这些生成器忽略`CMAKE_BUILD_TYPE` 缓存变量，而是要求开发人员在IDE中选择构建类型，或者在构建时使用命令行选项。配置和构建这样的项目如下所示：

```
cmake -G Xcode ../source  
cmake --build . --config Debug
```

Xcode IDE构建时，构建类型由构建方案控制，而在Visual Studio IDE中，当前解决方案可以控制构建类型。这两个环境都为不同的构建类型保留单独的目录，因此在构建之间的切换不会导致重新构建。对于单配置生成器，所做的工作与上面描述的多个构建目录安排是一样的，只是IDE代表开发人员处理了目录结构。

## 13.2. 常见错误

对于单个配置生成器，生成类型是在配置时指定，而对于多个配置生成器，生成类型在生成时指定。这个区别非常重要，它意味着当CMake处理项目的CMakeLists.txt文件时，构建类型并不已知。考虑下面这段CMake代码，演示了一个不正确的模式：

```
# WARNING: Do not do this!  
if(CMAKE_BUILD_TYPE STREQUAL "Debug")  
    # Do something only for debug builds  
endif()
```

上面的方法对于基于Makefile和Ninja的生成器来说很友好，但是对于Xcode和Visual Studio来说就不行了。实际上，项目中任何基于`CMAKE_BUILD_TYPE`的逻辑都有问题，除非确认开启了单个配置生成器的检查。对于多配置生成器，这个变量很可能是空的，即使不是空的，因为构建将忽略它，值也认为是不可靠的。项目应该使用其他更健壮的替代，比如基于`$<CONFIG:...>`的生成器表达式，而不是在CMakeLists.txt文件中引用`CMAKE_BUILD_TYPE`。

编写脚本构建时，一个常见的问题是假定使用了特定的生成器，或者没有正确考虑单个和多个配置生成器之间的差异。理想情况下，开发人员应该能够更改生成器，并且脚本的其余部分仍然能够正常工作。单配置生成器将忽略任何构建时规范，而多个配置生成器将忽略`CMAKE_BUILD_TYPE`变量，因此脚本可以考虑这两种情况。例如：

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../source
cmake --build . --config Release
```

对于上面的示例，开发人员可以简单地更改给-G参数的生成器名称，脚本的其余部分将不用更改。

不显式地为单个配置生成器设置`CMAKE_BUILD_TYPE`也很常见，但通常不是开发人员想要的。单个配置生成器的行为是，如果没有设置`CMAKE_BUILD_TYPE`，则构建类型为空。这会导致一些开发人员误解为空构建类型等同于Debug，但事实并非如此。这种情况下，不使用特定于配置的编译器或链接器标志，因此行为由编译器和链接器的默认行为决定。虽然这类似于Debug构建类型的行为，但实际情况却无法保证。

### 13.3. 自定义构建类型

有时项目可能希望将构建类型限制为默认值的子集，或者希望添加具有一组特殊编译器和链接器标志为自定义构建类型。后一种方法的例子是添加用于分析或代码覆盖的构建类型，这两者都需要特定的编译器和链接器设置。

开发人员可以在两个主要位置看到构建类型集。当使用Xcode和Visual Studio这样的多配置生成器时，IDE会提供一个下拉列表或者类似的东西，开发者可以从中选择想要的构建配置。对于单配置生成器(如Makefile或Ninja)，直接为`CMAKE_BUILD_TYPE`缓存变量输入构建类型，而CMake GUI可以提供一个组合框。这两种情况的机制是不同的，必须分别处理它们。

多配置生成器知道的构建类型集，并由`CMAKE_CONFIGURATION_TYPES`缓存变量控制。第一个遇到的`project()`命令用一个默认列表填充缓存变量(如果还没有定义的话)，但是项目可以在此之后修改同名的非缓存变量(修改缓存变量是不安全的，因为它可能会放弃开发人员所做的更改)。自定义构建类型可以通过将它们添加到`CMAKE_CONFIGURATION_TYPES`来定义构建，不需要的构建类型可以从列表中删除。

但是，如果`CMAKE_CONFIGURATION_TYPES`还没有定义，则要避免设置。CMake 3.9之前，判断是否使用了多配置生成器的常见方法是检查`CMAKE_CONFIGURATION_TYPES`是否为非空。甚至CMake本身的某些部分在3.11之前也是这样做的。虽然这种方法通常是准确的，但是即使使用单个配置生成器，也可以看到由项目单方面地设置`CMAKE_CONFIGURATION_TYPES`的情况。这可能导致对正在使用的生成器类型做出错误的判断。为了解决这个问题，CMake 3.9添加了一个新的`GENERATOR_IS_MULTI_CONFIG`全局属性，当使用多配置生成器时，该属性设置为`true`，从而提供了一种获得信息方法，而不是依赖于`CMAKE_CONFIGURATION_TYPES`。即便如此，检查`CMAKE_CONFIGURATION_TYPES`仍然是一种流行的模式，项目应该在它存在时修改它，而不要自己创建。还应该注意的是，在CMake 3.11之前，向`CMAKE_CONFIGURATION_TYPES`添加自定义构建类型在技术上是不安全的。CMake的某些部分只考虑了默认的构建类型，项目仍然可以用早期的CMake版本有效地定义自定义构建类型，这取决于如何使用CMake。为了更好的健壮性，如果要定义自定义构建类型，仍然建议使用CMake 3.11以上的版本。

另一个方面，开发人员可能会在 `CMAKE_CONFIGURATION_TYPES` 缓存变量中添加自定义的类型，或者删除一些不感兴趣的类型。因此，项目不应该对定义了或没有定义的配置类型做任何假设。

考虑到以上几点，下面的例子展示了项目为多配置生成器添加自定义构建类型的最佳方式：

```
cmake_minimum_required(3.11)
project(Foo)
if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
endif()
# Set relevant Profile-specific flag variables if not already set...
```

对于单配置生成器，只有一个构建类型，由 `CMAKE_BUILD_TYPE` 缓存变量指定。**CMake GUI**中，这通常是文本字段，因此开发人员可以编辑它的内容。如在9.6节所讨论，缓存变量可以由字符串定义一组有效值。然后**CMake GUI**将该变量作为包含有效值的组合框，而不是文本字段显示。

```
set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
    STRINGS Debug Release Profile)
```

属性只能从项目的**CMakeLists.txt**中更改，因此可以安全地设置 `STRINGS` 属性。要注意，设置缓存变量的 `STRINGS` 属性并不能保证缓存变量将持有值，它只能控制变量在**CMake GUI**应用程序中的显示方式。开发人员可以在**cmake**命令行中将 `CMAKE_BUILD_TYPE` 设置为任何值，或者手动修改**CMakeCache.txt**文件。为了严格要求变量具有所定义的值，项目本身必须显式地执行该测试。

```
set(allowableBuildTypes Debug Release Profile)
# WARNING: This logic is not sufficient
if(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
    message(FATAL_ERROR "${CMAKE_BUILD_TYPE} is not a defined build type")
endif()
```

`CMAKE_BUILD_TYPE` 的默认值是空字符串，除非开发人员显式地设置，否则上面的内容将对单/多配置生成器造成致命错误。特别是对于不使用 `CMAKE_BUILD_TYPE` 的多配置生成器。如果 `CMAKE_BUILD_TYPE` 没有设置，可以通过让提供一个默认值来处理。最终结果是这样的：

```
cmake_minimum_required(3.11)
project(Foo)

if(CMAKE_CONFIGURATION_TYPES)
    if(NOT "Profile" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES Profile)
    endif()
else()
    set(allowableBuildTypes Debug Release Profile)
    set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
        STRINGS "${allowableBuildTypes}")
    if(NOT CMAKE_BUILD_TYPE)
        set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
    elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
        message(FATAL_ERROR "Invalid build type: ${CMAKE_BUILD_TYPE}")
    endif()
endif()

# Set relevant Profile-specific flag variables if not already set...
```

上面讨论了仅允许选择定制的构建类型，但没有定义任何关于该构建类型的内容。当选中构建类型时，会指定CMake应该使用哪些特定于配置的变量，并且还会影响依赖于当前配置的生成器表达式(例如如`$<CONFIG>`和`$<CONFIG:...>`)逻辑。这些变量和生成器表达式将在下一章详细讨论，现在主要对下面两类变量感兴趣：

- `CMAKE_<LANG>_FLAGS_<CONFIG>`
- `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>`

这些可以用来在没有提供`_<CONFIG>`后缀的变量时，在默认设置上添加额外的编译器和链接器标志。例如，自定义配置文件构建类型的标志可以定义如下：

```
set(CMAKE_C_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")  
set(CMAKE_CXX_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")  
set(CMAKE_EXE_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")  
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")  
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")  
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE "-p -g -O2" CACHE STRING "")
```

上面假设有一个兼容GCC的编译器来保持示例的简单性，并启用分析及调试符号和最大程度优化。另一种方法是将编译器和链接器标志建立在其他构建类型的基础上，并添加额外标志。只要在`project()`命令之后执行，就可以执行此操作，因为该命令会填充默认的编译器和链接器标志变量。对于性能测试，`RelWithDebInfo`是一个很好的选择：

```
set(CMAKE_C_FLAGS_PROFILE  
    "${CMAKE_C_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")  
set(CMAKE_CXX_FLAGS_PROFILE  
    "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")  
set(CMAKE_EXE_LINKER_FLAGS_PROFILE  
    "${CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")  
set(CMAKE_SHARED_LINKER_FLAGS_PROFILE  
    "${CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")  
set(CMAKE_STATIC_LINKER_FLAGS_PROFILE  
    "${CMAKE_STATIC_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")  
set(CMAKE_MODULE_LINKER_FLAGS_PROFILE  
    "${CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO} -p" CACHE STRING "")
```

每个自定义配置都应该定义相关的编译器和链接器标志变量。对于一些多配置生成器类型，CMake将检查所需的变量存在，如果没有设置，构建将失败并报错。

另一个可能为自定义构建类型定义的变量是`CMAKE_<CONFIG>_POSTFIX`。它用于初始化每个库目标的`<CONFIG>_POSTFIX`属性，并在指定配置构建时，将其值附加到这些目标的文件名中。这使得来自多种构建类型的库可以放在同一个目录中，不会相互覆盖。`CMAKE_DEBUG_POSTFIX`经常设置为`d`或`debug`这样的值，特别是在Visual Studio构建中，调试和非调试构建中必须使用不同的`d11`，因此需要包含这两种构建类型的库。在上面定义的自定义配置构建类型的为Profile情况下，可以写为：

```
set(CMAKE_PROFILE_POSTFIX _profile)
```

如果创建包含多种构建类型的包，强烈建议为每种构建类型设置`CMAKE_<CONFIG>_POSTFIX`。按照惯例，发布版本的后缀通常是空的。注意，`<CONFIG>_POSTFIX`目标属性不适用于Apple平台。

由于历史原因，传递给`target_link_libraries()`命令的项可以加上调试或优化关键字作为前缀，以表明命名项应该分别在调试或非调试版本中链接。如果在`DEBUG_CONFIGURATIONS`全局属性中列出了构建类型，则认为是调试构建，否则是优化。对于自定义构建类型，如果此场景中将其作为调试构建，则应该将其名称添加到此全

局属性中。例如，一个项目定义了它自己的自定义构建类型 `StrictChecker`，并且该构建类型是一个非优化的调试构建类型，可以(也应该)像这样写：

```
set_property(GLOBAL PROPERTY APPEND DEBUG_CONFIGURATIONS StrictChecker)
```

新项目通常使用生成器表达式，而不使用 `target_link_libraries()` 命令的 `debug` 和 `optimized` 关键字。下一章更详细地讨论了这一问题。

## 13.4. 总结

开发人员不应该假设某个特定的CMake生成器会用来构建项目。同一项目的其他开发人员可能更喜欢使用不同的生成器，可能是与他们的IDE工具集成得更好，或者在之后的CMake版本中可能会添加对新生成器的支持。某些构建工具可能有一些bug，这些bug会对项目构建产生影响。因此在修复这些bug之前，使用可替代的生成器很有必要。如果假设使用了特定的CMake生成器，扩展项目所支持的平台集也会受阻。

使用单配置生成器(如Makefile或Ninja)时，考虑使用多个构建目录，每个构建目录对应一个构建类型。这允许在构建类型之间切换，而不必每次都强制重新编译。这提供了与多配置生成器原生行为相似的特性，并且可以帮助Qt Creator等IDE工具模拟多配置功能。

对于单个配置生成器，如果 `CMAKE_BUILD_TYPE` 为空，则考虑将其设置为合适的默认值。虽然空构建类型在技术上有效，但开发人员也经常误解为调试构建。此外，避免基于 `CMAKE_BUILD_TYPE` 创建逻辑，除非确认会使用单配置生成器。即使这样，这样的逻辑也很脆弱，并且可能会使用生成器表达式进行更通用和健壮地表达。

如果正在使用多配置生成器，或者该变量已经存在，只考虑修改 `CMAKE_CONFIGURATION_TYPES` 变量。如果添加自定义生成类型或删除默认生成类型之一，请不要修改缓存变量，而要更改同名的常规变量(将优先于缓存变量)。进行添加和删除单个项目，而不是完全替换列表。这两种方法都有助于避免干扰开发人员对缓存变量所做的修改。

如果需要CMake 3.9或更高版本，请使用 `GENERATOR_IS_MULTI_CONFIG` 全局属性来确定查询生成器类型，而不是依赖 `CMAKE_CONFIGURATION_TYPES` 来执行检查。

一种常见但不正确的做法是查询 `LOCATION` 目标属性，以确定目标的输出文件名，一个相关的错误是在自定义命令中假设特定的构建输出目录结构。这些方法并不适用于所有构建类型，在配置时对多个配置生成器的位置是未知的，而且构建输出目录结构在不同的CMake生成器类型中是不同的。所以应该使用 `$<TARGET_FILE:>` 这样的生成器表达式，因为它为所有生成器提供了所需的路径，无论是单配置还是多配置。

# 第14章：编译和链接

前一章讨论了构建类型，以及与选择的编译器和链接器行为之间的关系。本章讨论如何控制编译器和链接器的行为。本章提到的概念是每个CMake开发人员应该熟悉的技术。

深入研究之前，随着CMake的发展，控制编译器和链接器的行为有所改进。关注点可以从构建全局的视角，转移到可以控制每个单独目标的需求视角，以及这些需求应该或不应该传递到(依赖于的)其他目标上。这需要思维上进行转变，会影响项目如何有效地确定建立目标。CMake成熟的特性可以用在控制粗粒度行为上，但代价是失去定义目标之间的关系。以目标为中心的特性通常应该优先考虑使用，因为它们极大地提高了构建的健壮性，并提供了对编译器和链接器更精确的控制。更新的特性在其行为和使用方式上也趋于一致。

## 14.1. 目标属性

CMake的属性系统中，目标属性是控制编译器和链接器标志的主要机制。一些属性提供指定标志的能力，而其他属性则专注于功能，因此可以抽象出平台或编译器的差异。本章重点介绍更通用的属性，后面的章节将介绍一些具体的属性。

### 14.1.1. 编译器标示

控制编译器标示的基本目标属性如下，每个属性都包含一个列表：

#### `INCLUDE_DIRECTORIES`

用作头文件搜索路径的目录列表，都必须是绝对路径。CMake会为每条路径添加一个编译器标记，并在前面加上适当的前缀(通常是`-I`或`/I`)。创建目标时，此目标属性的初始值从同名的文件夹属性中获取。

#### `COMPILE_DEFINITIONS`

保存要在编译命令行上设置的定义列表。定义的形式为`VAR`或`VAR=VALUE`，CMake将其转换为编译器适用的形式(通常为`-DVAR...`或`/DVAR...`)。创建目标时，此目标属性的初始值为空。有一个同名的目录属性，但不为这个目标属性提供初始值。并且，目录和目标属性将会在最终的编译器命令行中合并。

#### `COMPILE_OPTIONS`

此属性中提供的既不是头搜索路径，也不是符号定义的编译器标志。创建目标时，此目标属性的初始值从同名的文件夹属性中获取。

较旧的、现在已不推荐使用的目标属性，其名称为`COMPILE_FLAGS`，与`COMPILE_OPTIONS`类似。`COMPILE_FLAGS`属性视为直接包含在编译器命令行中的单个字符串。因此，需要手工转义，而`COMPILE_OPTIONS`是一个列表，CMake会自动执行需要的转义或引号。

`INCLUDE_DIRECTORIES`和`COMPILE_DEFINITIONS`属性实际上只是方便为项目设置编译器标志。所有的编译器特定标志都由`COMPILE_OPTIONS`属性中提供。

上面的三个目标属性都有一个同名的目标属性，只是前面加上了`INTERFACE_`前缀。这些接口属性执行相同的操作，只是不应用于目标本身，而应用于链接到的其他目标。换句话说，用于指定的目标应该会继承的编译器标志。由于这个原因，通常称为使用需求，而非`INTERFACE`属性的构建需求。第16章目标类型中将讨论两种特殊库类型：`IMPORTED`和`INTERFACE`。这些特殊的库类型只支持`INTERFACE_...`目标属性，而不支持非`INTERFACE_...`属性。

与非接口对应项不同，上面的 `INTERFACE_...` 属性都不从目录属性初始化。他们开始都为空，当设置头文件搜索路径后，会定义和传递编译器标志到相应的目标上。

上述所有目标属性都支持生成器表达式。这对于 `COMPILE_OPTIONS` 属性特别有用，因为只允许在满足某些条件的情况下添加特定的标志，比如：针对特定的编译器进行操作。另一种常见用法是获取与其他目标相关的路径，并将作为包含目录的一部分。

如果需要对单个源文件操作编译器标志，那么目标属性的粒度就不够用了。这种情况下，**CMake** 提供了 `COMPILE_DEFINITIONS`、`COMPILE_FLAGS` 和 `COMPILE_OPTIONS` 源文件属性（`COMPILE_OPTIONS` 源文件属性仅在**CMake 3.11**中添加），这些类似于目标中的同名属性。注意，支持生成器表达式已经落后于目标属性，与 `COMPILE_DEFINITIONS` 源文件属性获得生成器表达式**CMake 3.8**，而其他两个则在**3.11**的支持。此外，**Xcode** 项目文件格式不支持配置特定的源文件属性，所以针对**Apple** 平台时，`$<CONFIG>` 或 `$<CONFIG:...>` 不应该用于源文件属性。还要记住**9.5**节中的警告，在使用源文件属性时，会产生性能问题。

## 14.1.2. 连接标志

与链接器标记相关的目标属性与编译器标志相似，但涉及的属性较少：

### `LINK_LIBRARIES`

此属性包含目标直接链接的所有库的列表。创建目标时为空，并且支持生成器表达式。列出的库可以是下列之一：

- 库的路径，通常为绝对路径。
- 只有库名而没有路径，通常也没有任何特定于平台的文件名前缀(如lib)或后缀(如：.a、.so、.dll)。
- **CMake**库目标的名称。当使用链接器命令时，**CMake**会将其转换为已构建库的路径，包括适合平台的文件名的前缀或后缀。因为**CMake**会代表项目处理不同的平台差异和路径，所以使用**CMake**目标名称通常是首选。

**CMake** 使用适当的链接器标志连接 `LINK_LIBRARIES` 属性中的每个链接。

### `LINK_FLAGS`

包含要传递给链接器的标志列表，用于可执行程序、动态库或模块库目标。对于静态库构建目标将忽略该属性。此属性用于一般链接器标记，不用于指定其他链接库的标记，不确定生成器表达式是否能支持。当创建目标时，此属性将为空。

### `STATIC_LIBRARY_FLAGS`

与 `LINK_FLAGS` 类似，不过用于静态库构建的目标。

与编译器属性不同，`LINK_LIBRARIES` 只有一个等效的接口属性 `INTERFACE_LINK_LIBRARIES`。没有与 `LINK_FLAGS` 或 `STATIC_LIBRARY_FLAGS` 等效的接口属性。

一些较老的项目中，可能会遇到名为 `LINK_INTERFACE_LIBRARIES` 的目标属性，它是 `INTERFACE_LINK_LIBRARIES` 的前身。这个旧属性自**CMake 2.8.12**已经弃用，可以启用策略**CMP0022**来使用。新项目更推荐使用 `INTERFACE_LINK_LIBRARIES`。

`LINK_FLAGS` 和 `STATIC_LIBRARY_FLAGS` 属性不支持生成器表达式，但确有相关的配置属性：

- `LINK_FLAGS_<CONFIG>`
- `STATIC_LIBRARY_FLAGS_<CONFIG>`

当 `<CONFIG>` 匹配正在构建的配置时，除了非配置标记外，还将使用上面两个标记。

## 14.1.3. 目标属性的命令

上述目标属性通常不直接进行操作。`CMake`可以以一种更方便、更健壮的方式修改它们，也鼓励明确依赖和目标之间传递行为的规范。4.3节在介绍 `target_link_libraries()` 时，解释了对的内部目标使用 `PRIVATE`、`PUBLIC` 和 `INTERFACE` 的依赖关系如何表示。前面的讨论集中在目标之间的依赖关系上，但这现些关键字的效果可以变得更为精确。

```
target_link_libraries(  
    targetName  
    <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]  
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]  
    ...  
)
```

### `PRIVATE`

`PRIVATE` 后面的项只影响 `targetName` 的行为。只有非 `INTERFACE_...` 的目标属性会改变影响范围(例如： `LINK_LIBRARIES`，`LINK_FLAGS` 和 `STATIC_LIBRARY_FLAGS` )。

### `INTERFACE`

这是对 `PRIVATE` 的补充，`INTERFACE` 关键字后面的项只对链接到 `targetName` 的目标有影响。只有 `INTERFACE_...` 的目标属性会改变影响范围(例如： `INTERFACE_LINK_LIBRARIES` )。

### `PUBLIC`

这相当于将 `PRIVATE` 和 `INTERFACE` 的并集。

大多数情况下，开发人员可能会感觉第4.3节的解释更加直观，但是上面更精确的描述可以用在更复杂的项目中，属性可能会以不同寻常的方式操纵行为。上面的描述也恰好与操纵编译器标志的 `target_...()` 命令行为非常接近。实际上，都遵循相同的模式，以相同的方式应用于 `PRIVATE`、`PUBLIC` 和 `INTERFACE` 。

```
target_include_directories(  
    targetName [BEFORE] [SYSTEM]  
    <PRIVATE|PUBLIC|INTERFACE> dir1 [dir2 ...]  
    [<PRIVATE|PUBLIC|INTERFACE> dir3 [dir4 ...]]  
    ...  
)
```

`target_include_directories()` 命令将头文件搜索路径添加到 `INCLUDE_DIRECTORIES` 和 `INTERFACE_INCLUDE_DIRECTORIES` 目标属性中。使用 `PRIVATE` 关键字时目录添加到 `INCLUDE_DIRECTORIES` 目标属性中，使用 `INTERFACE` 关键字时目录添加到 `INTERFACE_INCLUDE_DIRECTORIES` 目标属性中。使用 `PUBLIC` 关键字，则会将目录添加到 `INCLUDE_DIRECTORIES` 和 `INTERFACE_INCLUDE_DIRECTORIES` 目标属性中。

通常，调用 `target_include_directories()` 指定的目录都会附加到相关的目标属性中，这使得添加多个路径变得很容易。如果需要，可以使用 `BEFORE` 关键字将目录添加到当前目标属性的目录之前。

如果指定了 `SYSTEM` 关键字，编译器将在某些平台上将目录视为系统包含路径。这样做可能会跳过某些编译器警告，或改变文件依赖关系的处理方式，还会影响某些编译器头路径搜索的顺序。开发人员有时会尝试使用 `SYSTEM` 关键字“沉默”来自头文件的警告，而不是直接解决这些警告。如果这些头文件是项目的一部分，那么 `SYSTEM` 通常不是一个合适的选项。通常，`SYSTEM` 用于项目外部的路径(但即使这样也很少需要它)。

值得注意的是路径由导入指定目标 `INTERFACE_INCLUDE_DIRECTORIES` 属性将目标置于 `SYSTEM` 的默认路径。因为导入的目标假定来自项目外部，因此相关联的头文件应该以类似于其他系统提供的头文件的方式处理。项目可以通过将目标 `NO_SYSTEM_FROM_IMPORTED` 属性设置为 `true` 来覆盖，这将避免所有导入目标成为系统默认路径。导入目标的话题将在第16章中详细介绍。

`target_include_directories()` 命令提供了另一种直接操纵目标属性的方式。项目也可以指定相对路径，而不仅是绝对路径。相对路径将自动转换为绝对路径，并将路径视为相对于当前源目录的路径。

因为 `target_include_directories()` 命令基本上只是填充相关的目标属性，所以属性的所有常见特性都会应用。特别是可以使用生成器表达式，这一特性在安装目标和创建包时很重要。`$<BUILD_INTERFACE:...>` 和 `$<INSTALL_INTERFACE:...>` 生成器表达式允许为构建和安装指定不同的路径。安装目标通常用相对路径，将解释为相对于基地安装位置而不是源目录。第25.2.1节详细介绍了指定头文件搜索路径的内容。

```
target_compile_definitions(  
    targetName  
    <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]  
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]  
    ...  
)
```

`target_compile_definitions()` 命令非常简单，每个项都具有 `VAR` 或 `VAR=VALUE` 的形式。`PRIVATE` 项填充 `COMPILE_DEFINITIONS` 目标属性，而 `INTERFACE` 项填充 `INTERFACE_COMPILE_DEFINITIONS` 目标属性。`PUBLIC` 项会填充这两个目标属性。可以使用生成器表达式，但通常以相同的方式处理构建和安装情况。

```
target_compile_options(  
    targetName [BEFORE]  
    <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]  
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]  
    ...  
)
```

`target_compile_options()` 也非常简单。每一项都视为一个编译器选项，`PRIVATE` 项填充 `COMPILE_OPTIONS` 目标属性和 `INTERFACE` 项填充 `INTERFACE_COMPILE_OPTIONS` 目标属性。通常，`PUBLIC` 项会填充两个目标属性。对于所有情况，每个项都附加到现有的目标属性值上，可以使用 `BEFORE` 关键字作为前缀。可以使用生成器表达式，但通常以相同的方式处理构建和安装情况。

## 14.2. 目录属性和命令

**CMake** 3.0 及以后版本中，目标属性是用来指定编译器和链接器标志的首选项，因为它能够定义如何与相互链接的目标交互。**CMake** 的早期版本中，目标属性不那么突出，通常用在指定目录级别的属性。这些目录属性和通常用于操作它们的命令缺乏基于目标的等价物，从而显示的结果一样，这是项目应该尽可能避免使用它们的另一个原因。尽管如此，由于许多在线教程和示例仍然使用它们，开发人员应该了解一下目录级属性和命令。

```
include_directories([AFTER | BEFORE] [SYSTEM] dir1 [dir2...])
```

简单地说，`include_directories()` 将头文件搜索路径添加到在当前目录范围。默认情况下，路径会附加到现有的目录列表中，可以通过将 `CMAKE_INCLUDE_DIRECTORIES_BEFORE` 变量设置为 `ON` 来改变默认设置。还可以通过 `BEFORE` 和 `AFTER` 选项对每次调用进行控制，以显式地指示应该如何处理该调用的路径。项目应该小心设置 `CMAKE_INCLUDE_DIRECTORIES_BEFORE`，因为大多数开发人员可能会假定附加目录的默认行为是起作用的。`SYSTEM` 关键字与 `target_include_directories()` 具有相同的效果。

为 `include_directories()` 提供的路径可以是相对的，也可以是绝对的。相对路径转换为绝对路径自动视为相对于当前的源目录。路径也可能包含生成器表达式。

`include_directories()` 实际的细节比上面简单的解释要复杂得多。首先，调用 `include_directory()` 有两个效果：

- 将列出的路径添加到当前CMakeLists.txt文件的 `INCLUDE_DIRECTORIES` 目录属性中。这意味着在当前目录和子目录中创建的所有目标，都会将目录添加到 `INCLUDE_DIRECTORIES` 目标属性中。
- 当前CMakeLists.txt文件中创建的任何目标(或者更准确地说，当前目录范围)都将添加路径到其 `INCLUDE_DIRECTORIES` 目标属性中，即使这些目标是在调用 `INCLUDE_DIRECTORIES()` 之前创建的。这仅适用于当前CMakeLists.txt中创建的目标文件或通过 `include()` 包含的其他文件。

上面的第二点往往会让许多开发人员感到惊讶。为了避免可能导致这种混乱的情况，如果必须使用 `include_directory()`，最好在CMakeLists.txt文件中使用。创建任何目标或使用 `include()` 或 `add_subdirectory()` 拉入任何子目录之前，最好在CMakeLists.txt文件中调用 `include_directory()`。

```
add_definitions(-DSomeSymbol /DFoo=Value ...)
remove_definitions(-DSomeSymbol /DFoo=Value ...)
```

`add_definitions()` 和 `remove_definitions()` 命令添加和删除 `COMPILER_DEFINITIONS` 目录属性中的条目。每个条目都应该以 `-D` 或 `/D` 开头，这是绝大多数编译器使用的两种标记方式。在定义 `COMPILER_DEFINITIONS` 目录属性之前，CMake会去掉这个标志前缀，因此使用哪个前缀无关紧要(无论项目构建在哪个编译器或平台上)。

像 `include_directory()` 一样，这两个命令会影响当前CMakeLists.txt文件中创建的所有目标，甚至会影响那些在调用 `add_definitions()` 或 `remove_definitions()` 之前创建的目标。在子目录范围中创建的目标只有在调用之后才会受到影响。这是CMake使用 `COMPILER_DEFINITIONS` 目录属性的结果。

虽然不推荐，但也可以使用这些命令指定(除定义之外的)编译器标志。如果CMake不能识别出编译器定义的特定项，那么该项将会不加修改地添加到 `COMPILER_OPTIONS` 目录属性中。这种行为是由于历史原因出现的，但是新项目应该避免这种行为(参见下面的 `add_compile_options()` 命令以获得另一种选择)。

由于底层目录属性支持生成器表达式，所以可以执行这两个命令，但也有一些注意事项。生成器表达式只能用于定义有值的部分，而不能用于名称部分(即只能在 `-DVAR=值` 中的“=”之后使用，或者对于 `-DVAR` 是无法用的)。还要注意，这些命令只修改目录属性，它们不影响 `COMPILER_DEFINITIONS` 的目标属性。

`add_definitions()` 命令有许多缺点。要求在每个条目前面加上 `-D` 或 `/D` 以将其视为定义，这与其他CMake行为不一致。忽略前缀使得命令将项作为通用选项处理，这一事实对于命令的名称来说也是违反直觉的。此外，对生成器表达式的限制只支持 `KEY=VALUE` 定义的值部分，这也是对前缀有要求的原因。CMake 3.12引入了 `add_compile_definitions()` 作为 `add_definitions()` 的替代：

```
add_compile_definitions(SomeSymbol Foo=Value ...)
```

新命令只处理编译定义，不需要在每个项上使用任何前缀，并且可以使用生成器表达式。新命令的名称和定义项的处理与 `target_compile_definitions()` 一致。`add_compile_definitions()` 仍然会影响在同一个目录中创建的所有目标，无论这些目标创建在 `add_compile_definitions()` 之前或之后，因为这是 `COMPILER_DEFINITIONS` 目录属性的操作，而不是命令本身的操作。

```
add_compile_options(opt1 [opt2 ...])
```

`add_compile_options()` 用于提供任意的编译器选项。与 `include_directory()`、`add_definitions()`、`remove_definitions()` 和 `add_compile_definitions()` 不同，它的行为非常简单。`add_compile_options()` 的每个选项都会添加到 `COMPILER_OPTIONS` 目录属性中。随后在当前目录范围及后续创建的每个目标，都将在自己的 `COMPILER_OPTIONS` 目标属性中继承这些选项。调用之前创建的任何目标都不会受到影响。与其他目录属性行为相比，这种行为更接近开发人员的预期。此外，生成器表达式是由底层目录和目标属性控制，所以 `add_compile_options()` 也支持生成器表达式。

```
link_libraries(item1 [item2 ...] [ [debug | optimized | general] item] ...)
link_directories(dir1 [dir2 ...])
```

早期的CMake版本中，这两个命令是告诉CMake将库链接到其他目标的主要方式。它们会影响当前目录范围内，以及命令调用后创建的所有目标，但现有的目标不会受到影响(即类似于`add_compile_options()`的行为)。`link_libraries()`中指定的项可以是CMake目标、库名、库的完整路径，甚至是链接器标志。

简单地说，可以通过在调试构建类型前面加上关键字`Debug`，使一个项只适用于调试构建类型，或者通过在构建类型前面加上关键字`optimized`使其适用于除调试之外的所有构建类型。项前面可以加上关键字`general`，表明适用于所有构建类型，但是由于`general`是默认的，这样做没有什么意义。所有三个关键字只影响它后面的单个项，而不是下一个关键字之前的所有项。强烈建议不要使用这些关键字，因为生成器表达式可以更好地控制应该何时添加项。要考虑自定义生成类型，如果生成类型在`DEBUG_CONFIGURATIONS`全局属性中列出，则将其视为调试配置。

`link_directory()`所添加的目录只有在给CMake一个要链接的库名时才会起作用。CMake将提供的路径添加到链接器命令行中，并让链接器自己查找这些库。如果给出了相对路径，将视为相对于当前源目录(CMake的早期版本有不同的行为，请参阅策略CMP0015的文档了解详细信息)。完整的路径或CMake目标的名称通常是最首选，而且当`link_directory()`添加了链接器搜索目录，项目就没有方法来删除搜索路径(如果需要的话)。由于这些原因，应该尽可能避免添加链接器的搜索目录。

## 14.3. 编译器和链接器变量

属性是项目修改编译器和链接器标志的主要方式。用户不能直接操作属性，因此项目可以完全控制如何设置属性。但在某些情况下，用户需要添加他们自己的编译器或链接器标志。他们可能希望添加更多的警告选项，打开特殊的编译器特性，如Sanitizer或调试开关等等。在这些情况下，使用变量更合适。

CMake提供了一组变量，用于指定编译器和链接器标志，并将各种目录、目标和源文件属性提供的标志合并。它们通常是缓存变量，以便用户方便地查看和修改，但也可以在项目的CMakeLists.txt文件中设置为常规的CMake变量(这是项目应该避免的)。CMake在第一次在构建目录中运行时，会为缓存变量提供合适的默认值。

直接影响编译器标示的主要变量有以下形式：

- `CMAKE_<LANG>_FLAGS`
- `CMAKE_<LANG>_FLAGS_<CONFIG>`

这个变量族中，`<LANG>`对应于编译的语言，典型的值有`c`、`cxx`、`Fortran`、`Swift`等。`<CONFIG>`部分是一个大写字符串，对应于其中一种构建类型，如`DEBUG`、`RELEASE`、`RELWITHDEBINFO`或`MINSIZEREL`。第一个变量将应用于所有构建类型，包括`CMAKE_BUILD_TYPE`为空的单个配置生成器，而第二个变量应用于由`<CONFIG>`指定的构建类型。因此，使用调试配置构建的C++文件将具有来自`CMAKE_CXX_FLAGS`和`CMAKE_CXX_FLAGS_DEBUG`的编译器标记。

第一个`project()`命令会为这些不存在的缓存变量创建缓存变量(这有点简化了，更完整的解释在第21章中给出)。因此，第一次运行CMake之后，它们的值很容易在CMake GUI应用程序中找到。举个例子，对于一个特定的编译器，下面C++语言变量是默认的：

CMAKE_CXX_FLAGS	
CMAKE_CXX_FLAGS_DEBUG	-g -O0
CMAKE_CXX_FLAGS_RELEASE	-O3 -DNDEBUG
CMAKE_CXX_FLAGS_RELWITHDEBINFO	-O2 -g -DNDEBUG
CMAKE_CXX_FLAGS_MINSIZEREL	-Os -DNDEBUG

链接器标志的处理也是类似的，由以下变量族控制：

- CMAKE\_<TARGETTYPE>\_LINKER\_FLAGS
- CMAKE\_<TARGETTYPE>\_LINKER\_FLAGS\_<CONFIG>

这些变量用于特定类型的目标，在第4章中介绍过。变量名的 <TARGETTYPE> 部分必须是以下类型之一：

#### *EXE*

目标由 add\_executable() 创建。

#### *SHARED*

用 add\_library(name SHARED...) 或等效的方法创建的目标，比如省略 SHARED 关键字，但将 BUILD\_SHARED\_LIBS 变量设置为 true。

#### *STATIC*

用 add\_library(name STATIC...) 或等效的方法创建的目标，比如省略 STATIC 关键字，但 BUILD\_SHARED\_LIBS 变量设置为 false 或未定义。

#### *MODULE*

目标使用 add\_library(name MODULE ...) 创建。

如编译器标志一样，CMAKE\_<TARGETTYPE>\_LINKER\_FLAGS 在链接构建配置时使用，而 CMAKE\_<TARGETTYPE>\_LINKER\_FLAGS\_<CONFIG> 标志只在相应的配置中使用。某些平台上，部分或整个链接器标志为空很常见。

CMake 教程和示例代码经常使用上述变量来控制编译器和链接器标志。这在 CMake 3.0 之前是相当普遍的做法，但是随着 CMake 3.0 及以后的关注点转移到以目标为中心的模型上，这样的例子不再推荐。它们经常会导至一些非常常见的错误，下面列出了一些比较常见的错误。

#### 编译器/链接器变量是单个字符串，而不是列表

如果需要设置多个编译器标志，需要指定为单个字符串，而不是列表。如果标记变量的内容包含分号，CMake 将不能正确处理分号变量，如果项目指定了分号，列表将转换成分号。

```
# Wrong, list used instead of a string
set(CMAKE_CXX_FLAGS -Wall -Werror)

# Correct, but see later sections for why appending would be preferred
set(CMAKE_CXX_FLAGS "-Wall -Werror")

# Appending to existing flags the correct way (two methods)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
string(APPEND CMAKE_CXX_FLAGS " -Wall -Werror")
```

#### 区分缓存和非缓存变量

上面提到的所有变量都是缓存变量。可以定义同名的非缓存变量，变量将覆盖当前目录范围及其子目录（即 `add_subdirectory()` 创建的子目录）的缓存变量。但当项目试图强制更新缓存变量时，可能会出现问题。下面的代码往往会使项目难以工作，并可能导致开发人员感觉在与项目战斗，当他们想要改变标志时，可以通过CMake GUI或类似方法：

```
# Case 1: Only has an effect if the variable isn't already in the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags")

# Case 2: Using FORCE to always update the cache variable, but this overwrites
# any changes a developer might make to the cache
set(CMAKE_CXX_FLAGS "-Wall -Werror" CACHE STRING "C++ flags" FORCE)

# Case 3: FORCE + append = recipe for disaster (see discussion below)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror" CACHE STRING "C++ flags" FORCE)
```

上面的第一个例子是初学CMake的开发人员常见的疏忽。如果没有 `FORCE` 关键字，`set()` 命令只更新尚未定义的缓存变量。CMake的首次运行可能因此出现做开发人员计划（如果放置在任何 `project()` 命令），但如果是指定其他的标示，因为变量将已经在缓存中，所以这种变化不会应用到现有的构建。发现这一点后，通常的反应是使用 `FORCE` 来确保缓存变量的更新，如第二种情况所示，这又会产生另一个问题。缓存如果项目以这种方式使用 `FORCE` 单方面的设置缓存变量，开发人员对该缓存变量所做的修改都会丢失。第三种情况的问题更大，因为每次运行CMake时，都会再次添加标志，导致标志集不断增长和重复。为编译器和链接器标记使用 `FORCE` 来更新缓存不是一个好主意。

正确的行为不是简单地删除 `FORCE` 关键字，而是设置一个非缓存变量。然后，将标志附加到当前值，因为缓存变量保持不变，因此每次CMake运行都从缓存变量的相同标志集开始，而不管CMake被调用的频率有多高。开发人员对缓存变量所做的任何更改也将保留。

```
# Preserves the cache variable contents, appends new flags safely
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
```

### 选择附加而不是替换

如上所述，开发人员有时会试图单方面在CMakeLists.txt文件中设置编译器标志，如下所示：

```
# Not ideal, discards any developer settings from cache
set(CMAKE_CXX_FLAGS "-Wall -Werror")
```

因为这个设置会丢弃已设置的缓存变量，开发人员很容易地使用自己的标志。像这样替换现有的标志，迫使开发人员深入到项目文件中，以找到在哪里以及如何修改修改相关标志的行。对于有许多子目录的复杂项目，是相当头痛的。可能的情况下，项目应该倾向于向现有值添加标志。

合理的例外原则可能是，如果项目需要执行一组特定的编译器和链接器标示。在这种情况下，可行的方法是尽早将变量值设置在顶层CMakeLists.txt文件，最好在顶端 `cmake_minimum_required()` 命令后。请记住，随着时间的推移，项目可能成为另一个项目的子项，此时它将不再是构建的最高级别，这种折衷的适用性可能会减少。

### 了解什么时候使用变量值

更模糊的方面是编译器和链接器标志变量构建的过程，会使用它们的值。人们会期望以下代码的行为如注释描述一样：

```

# Save the original set of flags so we can restore them later
set(oldCxxFlags "${CMAKE_CXX_FLAGS}")

# This library has stringent build requirements, so enforce them just for it alone
# WARNING: This doesn't do what it may appear to do!
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror")
add_library(strictReq STATIC ...)

# Less strict requirements from here, so restore the original set of compiler flags
set(CMAKE_CXX_FLAGS "${oldCxxFlags}")
add_library(relaxedReq STATIC ...)

```

令人惊讶的是，按照上面的安排，`strictReq` 库将不会使用 `-Wall -Werror` 标志来构建。直观地说，我们可能期望CMake使用的是 `add_library()` 时的变量值，但实际上是在处理结束时使用的该目录作用域的变量值。换句话说，重要的是该变量在该目录的CMakeLists.txt文件末尾持有的值。对于不知情的人来说，这可能会在各种情况下导致意想不到的结果。

开发人员会让这种行为欺骗的主要原因是，把编译器和链接器变量当作立即应用到任何创建的目标上。另一个相关的陷阱，是在创建目标并且所包含的文件修改了编译器或链接器变量之后使用 `include()`，这会改变当前目录范围中已定义目标的编译器和链接器标志。由于编译器和链接器变量有这种延迟特性，使用它们可能会让项目变得很脆弱。理想情况下，项目只会在顶级CMakeLists.txt文件的修改它们(如果要修改的话)，以减少误用的机会。

## 14.4. 总结

本章涵盖CMake早期版本以来中经历的一些最重要的改进。读者可以在网上和其他地方找到大量的例子和教程，这些例子仍然推荐使用使用变量和目录属性命令的旧方法的模式和方法，但是应该理解 `target_...()` 命令应该是CMake 3.0+时代的首选方法。

项目应该使用 `target_link_libraries()` 定义目标之间的所有依赖关系。这清楚地表达了目标之间的关系，并明确地向项目的所有开发人员说明了目标是如何关联的。`target_link_libraries()` 应该优于 `link_libraries()` 或直接操作目标或目录属性。类似地，`target_...()` 提供了一种比变量、目录属性命令或直接操作属性更干净、更一致和更健壮的操作编译器和链接器标志的方法。以下的指南可能会有用：

- 如果可能，最好使用 `target_...()` 来描述目标之间的关系，并修改编译器和链接器的行为。
- 一般情况下，最好避免使用目录属性命令。虽然 `target_...()` 在一些特定的环境中很方便，但是使用 `target_...()` 将建立一个项目中所有开发人员都可以遵循的清晰模式。如果必须使用目录属性命令，请尽早在CMakeLists.txt文件中使用，以避免出现在前面几节中描述的一些不太直观的行为。
- 避免直接操作会影响编译器和链接器行为的目标和目录属性，理解属性的作用，以及不同的命令如何操作它们，最好使用专门的目标和目录特定命令。不过，在调试意外的编译器或链接器命令行标志时，查询目标属性可能会很有用。
- 最好避免修改各种 `CMAKE_...` 标志变量和特定配置的对应项，可能开发人员只希望在本地环境下改变他们。如果需要在整个项目的基础上应用更改，可以考虑在项目的顶层使用一些策略目录属性命令，但要考虑是否真的应该单方面应用这些设置。一个例外是在工具链文件中定义了初始的默认值(参见第21章会对这个领域进行详细的讨论)。

开发人员应该熟悉 `PRIVATE`、`PUBLIC` 和 `INTERFACE` 概念。他们是 `target_...()` 命令集的关键部分，并且他们在项目的安装和包装阶段也很重要。`PRIVATE` 是指目标本身，`INTERFACE` 是指与目标相链接的目标，而 `PUBLIC` 是前两者行为的结合。虽然标记为 `PUBLIC` 很诱人，但这可能会暴露出超出目标范围的依赖关系，构建时间可能会受到影响，`PRIVATE` 依赖关系可能会强加到其他不应该知道它们的目标上。这反过来会对其他领域产生强烈的影响，比如符号可见性(在20.5节中详细讨论)。因此，倾向于明确需要依赖其他目标时，将目标的 `PRIVATE` 改为 `PUBLIC`。

`INTERFACE` 关键字通常用于导入或接口库目标，或者偶尔用于向某些(项目中不允许开发人员更改的部分中定义的)目标添加缺少的依赖项。这样的例子包括为旧CMake版本编写的项目的子部分，因此不使用 `target_...`() 命令，或者导入外部目标库，这些库忽略了目标所需要的一些重要标志。对于 `target_link_libraries()` 之外的所有 `target_...`() 命令，指定的目标可以在项目的任何地方定义，唯一的要求是目标已经在 `target_...`() 命令调用之前创建。因此，可以从项目的任何部分将编译器接口依赖项附加到目标，但链接器接口依赖项只能在创建目标时在所在的目录范围内完成。CMake开发人员正在积极讨论这一限制，并可能在未来的版本中删除这个限制。

# 第15章：语言要求

随着C和C++语言的不断发展，开发人员越来越需要理解支持使用C和/或C++的编译器和链接器标志。不同的编译器使用不同的标志，即使使用相同的编译器和链接器，也可以使用标志来选择不同的标准库实现。

C++11支持相对较新的特性，CMake还没有直接支持选择使用哪种标准，所以项目只能自己添加所需的标志。CMake 3.1中，引入了一些特性，允许以一致和方便的方式选择C和C++标准，并抽象出各种编译器和链接器的差异。这种支持在后续版本中得到了扩展，从CMake 3.6开始，涵盖了最常见的编译器(CMake 3.2添加了大部分编译器支持，3.6添加了Intel编译器)。

CMake为指定语言提供了两种主要方法。第一个是直接设置语言标准，第二个是允许项目指定它们需要的语言特性，并让CMake选择适当的语言标准。虽然功能主要是由C和C++语言驱动的，但其他语言和伪语言(如CUDA)也可以支持。

## 15.1. 直接设定语言标准

项目控制构建所使用语言标准的最简单方法是直接设置，使用这种方法，开发人员不需要知道或指定代码使用的语言特性，只需要设置一个数字，表示代码假设支持的标准。这不仅容易理解和使用，还相对简单，可以确保在整个项目中使用相同的标准。这在链接阶段变得很重要，在这个阶段，应该在所有链接库和目标文件之间使用一致的标准库。

与CMake的常见模式一样，目标属性用于控制构建目标源以及连接可执行或动态库时使用哪一种标准。对于给定的语言，有三个与指定标准相关的目标属性(`<LANG>`必须是C或CXX中的一个，对于新的CMake版本，CUDA也是一个选项)：

### `<LANG>_STANDARD`

将项目指定为目标使用的语言标准。支持此特性的第一个CMake版本中，`c_STANDARD`的有效值为90、99或11，而`cxx_STANDARD`的有效值为98、11或14。CMake 3.8中，也支持17；CMake 3.12中，可以使用20。人们有理由认为，以后的CMake版本会随着时间的推移增加对其他语言标准的支持。CMake 3.8还支持值为98或11的`cuda_STANDARD`，`cxx_STANDARD`通常会控制的CUDA特定版本。当创建一个目标时，该属性的初始值可以从`CMAKE_<LANG>_STANDARD`变量中获取。

### `<LANG>_STANDARD_REQUIRED`

当`<LANG>_STANDARD`属性指定了项目语言标准时，`<LANG>_STANDARD_REQUIRED`决定了该语言标准是作为最低要求，还是仅仅作为“可用时使用”的标准。人们可能会直观地认为`<LANG>_STANDARD`默认是必需的，而`<LANG>_STANDARD_REQUIRED`属性默认是关闭的。当关闭时，如果编译器不支持请求的标准，CMake将把请求回退到更早的标准，而不是报错终止。这种回退的行为通常是新开发人员无法处理的，可能会造成混乱。因此，对于大多数项目来说，当指定`<LANG>_STANDARD`属性时，需要将其对应的`<LANG>_STANDARD_REQUIRED`属性设置为`true`，以确保特定要求的标准会视为一个严格的要求。当创建一个目标时，该属性的初始值会从`CMAKE_<LANG>_STANDARD_REQUIRED`变量中获取。

### `<LANG>_EXTENSIONS`

许多编译器都支持对语言标准的扩展，通常会提供一个编译器和/或链接器标志来启用或禁用这些扩展。`<LANG>_EXTENSIONS`目标属性就是控制是否启用这些特定的扩展。对于一些编译器/连接器，该设置可以改变目标与标准库(见下面的例子)。请注意，对于许多编译器/连接器，可以使用相同的标志来控制语言标准和是否启用扩展。如果一个项目设置了`<LANG>_EXTENSIONS`属性，也应该设置`<LANG>_STANDARD`属性，否则会忽略`<LANG>_EXTENSIONS`。创建目标时，`<LANG>_EXTENSIONS`属性的初始值从`CMAKE_<LANG>_EXTENSIONS`变量中获取。

项目通常会设置为上述目标属性提供的默认值，而不是直接设置目标属性。这确保了项目中的所有目标都以一致的方式生成。此外，强烈建议项目设置所有这三个属性/变量。对于开发人员来说，`<LANG>_STANDARD_REQUIRED` 和 `<LANG>_EXTENSIONS` 的默认值相对来说不是很直观，因此通过显式地设置它们，项目可以清楚地知道期望的标准行为是什么。几个示例可以帮助演示其用法：

```
# Require C++11 and disable extensions for all targets
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

当使用GCC或Clang时，上面的代码通常会添加`-std=c++11`标志。它还可以根据平台添加链接器标记，如`-stdlib=libc++`。针对Visual Studio编译器VS2015 Update 3之前，如果编译器不支持C++11则不会有标示添加。还要注意的是，在Visual Studio 15 Update 3中，编译器支持指定C++标准，但仅适用于C++14及以后版本，C++14是默认设置。

相比之下，下面的示例请求一个更新的C++版本，并启用了编译器扩展，结果生成了一个GCC/Clang编译器标记，如`-std=gnu++14`。Visual Studio编译器在默认情况下也支持所要求的标准，这取决于编译器的版本。如果正在使用的编译器不支持所要求的C++标准，CMake将配置编译器使用它所支持的最新C++标准。

```
# Use C++14 if available and allow compiler extensions for all targets
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED OFF)
set(CMAKE_CXX_EXTENSIONS ON)
```

C的情况非常类似。下面的例子展示了如何设置C标准的详细信息，这次只针对特定的目标：

```
# Build target foo with C99, no compiler extensions
set_target_properties(foo PROPERTIES
  C_STANDARD 99
  C_STANDARD_REQUIRED ON
  C_EXTENSIONS OFF
)
```

`<LANG>_STANDARD`指定了一个最低标准，不一定是一个确切的要求。某些情况下，CMake可能会选择一个近期标准来满足编译特性需求(下面将讨论)。

## 15.2. 根据特性需求设置语言标准

直接为一个目标或整个项目设置语言标准是管理标准需求的最简单方法，当项目开发人员知道哪个语言版本提供项目代码所使用的特性时，这是最合适的方法。当使用大量语言特性时，这种方式特别方便，因为每个特性都不必显式地指定。然而，在某些情况下，开发人员可能更喜欢声明代码使用的语言特性，而让CMake选择适当的语言标准。与直接指定标准不同，这样做还有一个额外的好处，即编译特性可以成为目标接口的一部分，因此可以在链接到它的其他目标上强制执行。

编译特性需求由目标属性`COMPILE_FEATURES`和`INTERFACE_COMPILE_FEATURES`控制，但是这些属性通常使用`target_compile_features()`填充，而不是直接操作。这个形式与CMake提供的`target_...()`非常相似：

```
target_compile_features(  
    targetName  
    <PRIVATE|PUBLIC|INTERFACE> feature1 [feature2 ...]  
    [<PRIVATE|PUBLIC|INTERFACE> feature3 [feature4 ...]]  
    ...  
)
```

`PRIVATE`、`PUBLIC` 和 `INTERFACE` 关键字有它们的含义。`PRIVATE` 特性填充 `COMPILE_FEATURES` 属性，就仅应用于目标本身。使用 `INTERFACE` 关键字指定的那些特性填充 `INTERFACE_COMPILE_FEATURES` 属性，该属性会应用于链接到 `targetName` 的目标。

指定为 `PUBLIC` 的特性将添加到这两个属性中，因此将应用到目标本身，以及链接到它的其他目标中。

每个特性必须是底层编译器支持的特性。**CMake**提供了两个已知特性列表：`CMAKE_<LANG>_KNOWN_FEATURES`，包含该语言所有已知特性；`CMAKE_<LANG>_COMPILE_FEATURES`，包含编译器支持的特性。如果编译器不支持所请求的特性，**CMake**将报告错误。开发人员可以在**CMake**文档中找到 `CMAKE_<LANG>_KNOWN_FEATURES` 变量，不仅列出了特定版本的**CMake**所理解的特性，还包含了对每个特性的标准文档的引用。请注意，并不是所有的功能都提供了特定的语言版本可以显式地指定使用编译功能。例如，新的C++ STL类型、函数等没有相关的特性。

**CMake 3.8**中，每种语言的元特性可用来表示特定的语言标准，而不是特定的编译特性。这些元特性采用 `<lang>_std<value>` 的形式，当列为必需的编译特性时，**CMake**将确保使用支持该语言标准的编译器标志。例如：添加一个编译特性，确保目标和任何链接到它的支持C++14，可以使用以下功能：

```
target_compile_features(targetName PUBLIC cxx_std_14)
```

如果项目需要支持比3.8更早的**CMake**版本，那么上面的元特性将不可用。这种，每个编译特性都必须单独列出，这是不实际的，也是不完整的。这通常会限制编译特性的有用性，因为项目通常选择通过前一节中描述的目标属性来设置语言标准。

当一个目标同时设置了它的 `<LANG>_STANDARD` 属性并指定了编译特性(直接或传递接口特性的结果)时，**CMake**将强制执行更强的标准要求。在下面的例子中，`foo`用C++14构建，`bar`用C++17构建，`guff`用C++14构建：

```
set_target_properties(foo PROPERTIES CXX_STANDARD 11)  
target_compile_features(foo PUBLIC cxx_std_14)  
  
set_target_properties(bar PROPERTIES CXX_STANDARD 17)  
target_compile_features(bar PRIVATE cxx_std_11)  
  
set_target_properties(guff PROPERTIES CXX_STANDARD 11)  
target_link_libraries(guff PRIVATE foo)
```

请注意，这意味着可以使用比项目预期的更最新的语言标准，在某些情况下可能会导致编译错误。例如，C++17删除了 `std::auto_ptr`，所以如果代码希望用较旧的语言标准进行编译，仍然使用 `std::auto_ptr` 时，如果工具链严格执行删除操作，则无法进行编译。

### 15.2.1. 检测和使用可选语言特性

有些项目能够处理支持或不支持的语言特性。例如：如果编译器支持它们，可能提供一个示例，或者只定义某些函数重载。项目可能支持一些可选的编译器特性，例如：用于指导开发人员的关键字，或为编译器提供捕获常见错误的增强性关键字。像 `final` 和 `override` 这样的C++关键字就是常见的例子。

**CMake**提供了许多处理上述场景的方法。一种方法是使用生成器表达式，根据特定编译器特性的可用性，有条件地设置编译器定义或包含目录。这些可能有点冗长，但提供了极大的灵活性，并基于特性的功能支持非常精确地处理。考虑下面的例子：

```
add_library(foo ...)

# Make override a feature requirement only if available
target_compile_features(foo PUBLIC
$<$<COMPILE_FEATURES:cxx_override>:cxx_override>
)

# Define the foo_OVERRIDE symbol so it provides the
# override keyword if available or empty otherwise
target_compile_definitions(foo PUBLIC
$<$<COMPILE_FEATURES:cxx_override>:-Dfoo_OVERRIDE=override>
$<$<NOT:$<COMPILE_FEATURES:cxx_override>>:-Dfoo_OVERRIDE>
)
```

上面的代码将允许如下代码为任何C++编译器编译，不管它是否支持 `override` 关键字：

```
class MyClass : public Base
{
public:
    void func() foo_OVERRIDE;
...
};
```

除了 `override` 关键字之外，许多其他特性也可以以相同的方式使用条件定义符。C++关键字，如 `final`，`constexpr`，`noexcept` 等能都可以使用，从而产生有效和正确的代码。其他关键字，如 `nullptr` 和 `static_assert`，如果编译器不支持，则可以使用其他实现。为每个特性指定生成器表达式来覆盖受支持和不受支持的情况将是冗长乏味的，但CMake通过模块系统提供了一种更方便的机制。`WriteCompilerDetectionHeader` 模块定义了一个名为 `write_compiler_detection_header()` 的函数，该函数可以自动进行此类处理。它生成一个头文件，项目的源代码可以 `#include` 该头文件，以获取适当指定的编译器定义。该函数的简化版本如下所示，只显示强制选项。

```
write_compiler_detection_header(
FILE fileName
PREFIX prefix
COMPILERS compiler1 [compiler2 ...]
FEATURES feature1 [feature2 ...]
)
```

该函数将向指定的文件名输出一个C/C++头文件，其中的内容将列出的每个特性定义适当的宏。每个特性都有一个宏，其形式为 `prefix_COMPILER_UPPERCASEFEATURE`，其值将为1或0，具体取决于所使用的编译器是否支持该特性。一些特性还可能具有 `prefix_UPPERCASEFEATURE` 形式的宏，为每个编译器提供该特性最合适的实现，包括编译器的不同版本。

考虑一个C++项目，它可以使用 `override`、`final` 和 `nullptr` 关键字(如果可用的话)，目标是支持GNU、Clang、Visual Studio和Intel编译器在这些编译器支持的任何平台上。如果编译器支持右值引用，项目还将定义 `move` 构造函数。下面将在构建目录中写出一个名为 `foo_compiler_detection.h` 的头文件，并以字符串 `foo_` 作为每个宏的前缀：

```
include(WriteCompilerDetectionHeader)
write_compiler_detection_header(
    FILE foo_compiler_detection.h
    PREFIX foo
    COMPILERS GNU Clang MSVC Intel
    FEATURES cxx_override
    cxx_final
    cxx_nullptr
    cxx_rvalue_references
)
```

C++代码使用上面的宏定义可能看起来像这样：

```
#include "foo_compiler_detection.h"

class MyClass foo_FINAL : public Base
{
public:
#if foo_COMPILER_CXX_RVALUE_REFERENCES
    MyClass(MyClass&& c);
#endif
    void func1() foo_OVERRIDE;
    void func2(int* p = foo_NULLPTR);
};
```

使用上述源文件的目标仍需要选择适当的语言标准，这种情况下，由于后退实现可用，所需的标准可以指定，但不是必需的：

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED OFF)
set(CMAKE_CXX_EXTENSIONS OFF)

add_library(foo MyClass.cpp)

# The header is written to the build directory
# so ensure we add that to the header search path
target_include_directories(foo
    PUBLIC "${CMAKE_CURRENT_BINARY_DIR}"
)
```

CMake为相当多的特性提供了后退实现，所有这些在 `WriteCompilerDetectionHeader` 模块文档中都有描述。`write_compiler_detection_header()` 命令还接受许多没有提到的可选参数，这些参数可以控制生成的头文件的结构和位置，并在生成的头文件的开始和结束处添加任意内容。感兴趣的读者可以参考CMake模块文档，以获得完整的细节。

深入使用 `WriteCompilerDetectionHeader` 模块之前，项目应该仔细考虑使用编译器检测头文件是否值得。它可以扩展项目，是支持的编译器范围的优秀工具。特别是，长期存在的项目可能会发现，在某些平台上仍然需要支持较老的编译器时，它是一个有用的垫脚石，当然也可以将代码库更新为最新的语言特性。使用该模块的主要缺点是可能会降低源代码的可读性。强制所有源文件使用替代(生成的)符号名而不是标准语言关键字也很困难，因为这对一些开发人员来说不大能想得到。

## 15.3. 总结

项目应该避免直接设置编译器和链接器标志，来控制所使用的语言标准。所需的标志因编译器的不同而不同，因此使用CMake提供的特性更健壮、更可维护、更方便，并允许适当地填充。CMakeLists.txt文件也将更清楚地表达意图，而不是通常晦涩难懂的编译器和链接器标志。

控制语言标准最简单的方法是使

用 `CMAKE_<LANG>_STANDARD`、`CMAKE_<LANG>_STANDARD_REQUIRED` 和 `CMAKE_<LANG>_EXTENSIONS` 变量。这些可用于为整个项目设置语言标准，确保所有目标的标准一致。这可以帮助避免链接不一致的标准库和其他链接问题。理想情况下，应该在顶级CMakeLists.txt文件的第一个 `project()` 命令后设置这些变量。项目应该始终将这三个变量放在一起，以明确应该如何执行语言标准要求，以及是否允许编译器进行扩展。忽略 `CMAKE_<LANG>_STANDARD_REQUIRED` 或 `CMAKE_<LANG>_EXTENSIONS` 通常会导致意想不到的行为，因为默认值可能不是一些开发人员所期望的。

如果语言标准只需要对某些目标执行，而不需要对其他目标执行，那

么 `<LANG>_STANDARD`、`<LANG>_STANDARD_REQUIRED` 和 `<LANG>_EXTENSIONS` 目标属性可以在单个目标上设置，而不是在整个项目中设置。这些属性的行为就好像是私有的一样，只指定目标的需求，而不指定任何链接到它的目标。因此，这给项目增加了更多的负担，以确保所有目标都正确指定了语言标准细节。实际上，使用变量在项目范围内设置语言需求通常比使用每个目标属性更容易、更健壮。最好使用这些变量，除非项目需要针对不同的目标使用不同的语言标准。

如果使用CMake 3.8或更高版本，可以使用编译特性为每个目标指定所需的语言标

准。`target_compile_features()` 命令简化了这一过程，并清楚地指定了这些需求是 `PRIVATE`、`PUBLIC` 的还是 `INTERFACE`。以这种方式指定语言需求的主要优点是，可以通过 `PUBLIC` 和 `INTERFACE` 关系将其传递到其他目标上。当目标在导出和安装时，这些要求也保留了下来。但请注意，只提供 `<LANG>_STANDARD` 和 `<LANG>_STANDARD_REQUIRED` 的目标属性行为，所以仍然应该使用 `<LANG>_EXTENSIONS` 目标属性或 `CMAKE_<LANG>_EXTENSIONS` 变量来控制是否允许编译器扩展。这些 `_EXTENSIONS` 属性/变量通常只与对应的 `<LANG>_STANDARD` 生效，由于设置编译器和链接器经常将这两个结合到一个标志，所以最终还要指定 `<LANG>_STANDARD`，即使用编译特性。因此，使用项目范围的变量会让项目更容易、更健壮。

指定单独的编译特性，可以在每个目标级别上对语言需求进行细粒度控制。开发人员很难保证目标使用的所有特性都明确指定，因此始终存在语言需求是否正确定义的问题。随着代码开发的持续进行，也很容易过时。大多数项目会发现以这种方式指定语言需求既繁琐又脆弱，所以只有在情况明确需要时才应该使用它们。对一种语言添加时，比如：为即将发布的语言使用建议的特性时，编译特性可能是一种有用的方法，CMake支持这些特性。一般来说，项目应该更喜欢使用变量或属性来在更高的级别上设置语言需求，以获得更好的可维护性和健壮性。另外，通过编译设置标准元特性(像 `cxx_std_11`)还可以避免许多设置单独特性的间题。对于语言标准，CMake不再定义单个的特性，只提供元特性。

项目可以检测可用的编译功能，并提供功能可用的实现。CMake甚至通过 `WriteCompilerDetectionHeader` 模块提供了一些方便的宏，使这个任务更容易完成。项目通常只考虑使用这些特性作为过渡路径更新旧代码库时，使用新的语言特性，因为它们往往会让开发者感觉不那么自然(因为降低了代码的可读性)。一个明显的例外是项目用于多种编译器语言标准的支持，可以有所不同。对于这个场景，在使用现代编译器时，对特定语言特性的可选支持可能有助于减少编译器警告和编码错误。对于大多数开发人员来说，这些好处应该与增加代码的复杂性和可读性在降低，以及不太自然的代码样式间进行权衡。

# 第16章：目标类型

**CMake**支持各种各样的目标类型，可以定义不同的目标类型，作为对其他实体目标的引用，而不是自己构建。可以用来收集传递属性和依赖关系，而无需生成二进制文件，或者是一种库，即对象文件的集合，而不是传统的静态或动态库。可以将许多东西抽象为目标，以隐藏平台差异、文件系统中的位置、文件名等的复杂性。本章涵盖了所有这些不同的目标类型，并讨论如何使用它们。

另一类目标是实用程序或自定义目标。这些可用于执行任意命令和定义自定义构建规则，从而允许项目实现所需的任何行为。它们有自己专用的命令和独特的行为，将在下一章深入介绍。

## 16.1. 可执行类型

`add_executable()` 不仅仅是第4章中介绍的可用于构建简单目标。还有另外两种形式，可用于定义引用其他内容的可执行目标。所支持的方式包括：

```
add_executable(  
    targetName [WIN32] [MACOSX_BUNDLE]  
    [EXCLUDE_FROM_ALL]  
    source1 [source2 ...])  
add_executable(targetName IMPORTED [GLOBAL])  
add_executable(aliasName ALIAS targetName)
```

`IMPORTED` 的方式可以为现有的可执行文件创建**CMake**目标，而不是由项目构建的目标。通过创建一个可执行文件的目标，项目的其他部分可以将它视为对待项目本身的构建(有一些限制)。可以用**CMake**自动将目标名称替换为其在磁盘上位置的上下文，比如测试或自定义任务执行时(这两个内容将在后面的章节中介绍)。与常规目标的区别之一是，导入的目标不能安装，这个主题将在第25章中讨论。

定义导入的可执行目标时，需要先设置某些目标属性，然后才能使用。大多数导入目标的相关属性的名称都以 `IMPORTED` 开头，但是对于可执行文件来说，`IMPORTED_LOCATION` 和 `IMPORTED_LOCATION_<CONFIG>` 是最重要的属性。需要导入可执行文件的位置时，**CMake**会首先查看配置属性，只有当没有设置时，才会查看 `IMPORTED_LOCATION` 属性。通常，位置不需要特定的配置，因此通常只设置 `IMPORTED_LOCATION`。

没有定义 `GLOBAL` 时，导入目标只会在当前目录范围内可见，当添加了 `GLOBAL` 属性，目标则随处可见。不过，由项目构建的可执行目标总是全局的。造成这种情况的原因，以及目标可见度降低的相关影响，会在16.3节中进一步讨论。

`ALIAS` 目标只是在**CMake**中引用目标的只读方式，并没有使用别名创建新的构建目标。别名只能指向真实的目标(不支持别名的别名)，并且不能安装或导出。**CMake 3.11**之前，导入的目标也不能混叠，**CMake 3.11**放宽了一些限制，允许混叠导入的目标，但只允许具有全局可见性的导入目标。

## 16.2. 库类型

`add_library()` 也有许多不同的形式。第4章介绍的基本形式，大多数开发人员都熟悉的是，建立简单的目标通常可以用于定义库，可以用来定义对象库、静态库或动态库。该命令的扩展基本形式为：

```
add_library(  
    targetName [STATIC | SHARED | MODULE | OBJECT]  
    [EXCLUDE_FROM_ALL]  
    source1 [source2 ...])
```

**CMake 3.12**之前，对象库不能像其他库那样链接(不能与 `target_link_libraries()` 一起使用)，需要使用 `$<TARGET_OBJECTS:objLib>` 生成器表达式作为另一个可执行文件或库目标的部分源。由于不能链接，因此不提供对作为对象/源添加的目标的传递依赖关系。这使得对象库用起来很不方便，因为头文件搜索路径、编译器定义等必须手动到添加依赖于它们的目标上。

**CMake 3.12**引入了一些特性，这些特性使对象库的行为更像其他类型的库，但也有一些注意事项。**CMake 3.12**中，对象库可以与 `target_link_libraries()` 一起使用，要么作为添加的目标(命令的第一个参数)，要么作为添加的库之一。但因为添加的是对象文件而不是实际的库，因为要防止对象文件多次添加到目标中，所以传递特性非常受限。一个简单的解释是，对象文件只添加到直接链接到对象库的目标中，而不是传递到其他对象库中。不过，对象库的使用确实可以像普通库那样传递。

开发人员会发现对象库更自然。然而，通常情况下，如果有选择的话，静态库在**CMake**项目中通常是更方便的选择。在依赖于**CMake 3.12**中对象库的扩展特性之前，请考虑静态库是否更合适。

和可执行文件一样，库也可以定义为导入的目标。打包过程中创建的配置文件或Find模块实现中大量使用了这些工具。他们不定义构建库的目标，而是作为外部提供的引用库(例如：已经存在的系统，是由当前**CMake**以外的项目的部分提供)。

```
add_library(  
    targetName (STATIC | SHARED | MODULE | OBJECT | UNKNOWN)  
    IMPORTED [GLOBAL]  
)
```

库类型需要在 `targetName` 后给出。如果新目标引用库的类型是已知的，则应该这样指定。这将允许**CMake**在各种情况下将导入的目标，可以视为命名类型的库目标。该类型只能设置为**CMake 3.9**或更高版本的 `OBJECT` (该版本之前不支持导入的对象库)。如果库类型是未知的，`UNKNOWN` 类型显式给出，这种情况下，**CMake**只能使用库的完整路径。这意味着更少的检查，并且在Windows环境下构建时，不处理DLL导入库。

除了对象库之外，导入的目标所代表的文件系统的位置需要由 `IMPORTED_LOCATION` 和/或 `IMPORTED_LOCATION_<CONFIG>` 属性来指定(即与导入的可执行文件相同)。Windows平台上，两个属性都应该设置：`IMPORTED_LOCATION` 应持有DLL的位置，`IMPORTED_IMPLIB` 应持有相关导入库的位置，通常是LIB文件的位置 (`..._<CONFIG>` 的方式也可以设置这些属性，应该优先考虑)。对于对象库，必须将 `IMPORTED_OBJECTS` 属性设置为导入的目标的对象文件列表，而不是上面的那些位置属性。

导入库还支持许多其他目标属性，其中大多数属性可以由**CMake**自动设置。需要手动编写配置包的开发人员应该参考**CMake**参考文档，以了解其他可能的情况，和相关的 `IMPORTED_...` 目标属性。大多数项目会依赖**CMake**生成这样的文件，所以这样的操作时不多见的。

默认情况下，导入的库定义为本地目标，这意味着只在当前目录范围内可见。可以使用 `GLOBAL` 关键字使它们具有全局可见性。创建库时可能没有 `GLOBAL` 关键字，但随后会将其提升为全局可见性，这是16.3节中讨论的主题。

```
# Windows-specific example of imported library  
add_library(myWindowsLib SHARED IMPORTED)  
set_target_properties(  
    myWindowsLib PROPERTIES  
    IMPORTED_LOCATION /some/path/bin/foo.dll  
    IMPORTED_IMPLIB /some/path/lib/foo.lib  
)
```

```
# Assume FOO_LIB holds the location of the library but its type is unknown
add_library(mysteryLib UNKNOWN IMPORTED)
set_target_properties(
    mysteryLib PROPERTIES
    IMPORTED_LOCATION ${FOO_LIB}
)
```

```
# Imported object library, Windows example shown
add_library(myObjLib OBJECT IMPORTED)
set_target_properties(
    myObjLib PROPERTIES
    IMPORTED_OBJECTS /some/path/obj1.obj # These .obj files would be .o
    /some/path/obj2.obj # on most other platforms
)

# Regular executable target using imported object library.
# Platform differences are already handled by myObjLib.
add_executable(myExe ${TARGET_SOURCES:myObjLib})
```

另一种形式的 `add_library()` 允许定义接口库，主要收集应用于链接到它们的目标使用需求和依赖关系。使用示例是仅用于纯头文件库，其中没有需要链接的物理库，但是头文件搜索路径、编译器定义等需要传递到使用头文件库的地方。

```
add_library(targetName INTERFACE [IMPORTED [GLOBAL]])
```

不同的 `target_…()` 都可以与 `INTERFACE` 关键字一起使用，以定义接口库的使用需求。也可以直接使用 `set_property()` 或 `set_target_properties()` 设置相关的 `INTERFACE_…` 属性，但是使用 `target_…()` 更安全、更容易。

```
add_library(myHeaderOnlyToolkit INTERFACE)
target_include_directories(
    myHeaderOnlyToolkit
    INTERFACE /some/path/include
)
target_compile_definitions(
    myHeaderOnlyToolkit
    INTERFACE COOL_FEATURE=1
    ${$<<COMPILE_FEATURES:cxx_std_11>:HAVE_CXX11}
)
add_executable(myApp ...)
target_link_libraries(
    myApp PRIVATE myHeaderOnlyToolkit)
```

上面的示例中，`myApp` 目标链接指向 `myHeaderOnlyToolkit` 接口库。当编译 `myApp` 源码时，将使用 `/some/path/include` 作为头文件搜索路径，并在编译器命令行中提供一个编译时定义 `COOL_FEATURE=1`。如果构建 `myApp` 目标时支持C++11，那就和定义了 `HAVE_CXX11` 一样。然后，`myHeaderOnlyToolkit` 中的头文件可以使用这个符号来确定声明和定义的内容，而不是依赖C++提供的 `_cplusplus` 符号（`_cplusplus` 的值对于编译器来说通常是不可靠的）。

接口库的另一个用途是为链接更大的库提供方便，其中封装了应该在集合中的库。例如：

```

# Regular library targets
add_library(algo_fast ...)
add_library(algo_accurate ...)
add_library(algo_beta ...)

# Convenience interface library
add_library(algo_all INTERFACE)
target_link_libraries(
    algo_all INTERFACE
    algo_fast
    algo_accurate
    $<$<BOOL:${ENABLE_ALGO_BETA}>:algo_beta>
)

# Other targets link to the interface library
# instead of each of the real libraries
add_executable(myApp ...)
target_link_libraries(myApp PRIVATE algo_all)

```

如果CMake选项变量 `ENABLE_ALGO_BETA` 为真，那么库列表中将只包含 `algo_beta`。其他目标简单地链接到 `algo_all`，`algo_beta` 的条件链接由接口库处理。这是使用接口库来抽象实际要链接、定义什么的细节的例子，这样针对它们链接的目标就不必为实现这些细节。这样在不同的平台上，就可以用来做一些完全不同的抽象库结构，开关库的实现基于某些条件(变量、生成器表达式等)，提供一个旧库目标名称(库结构进行了重构)(例如：分成单独的库)等等。

虽然 `INTERFACE` 库的用例很容易理解，但是添加 `IMPORTED` 关键字来生成 `INTERFACE IMPORTED` 库有时会引起混淆。当导出或安装 `INTERFACE` 库以便在项目外部使用时，通常会出现这种组合。当另一个项目使用时，仍然起到 `INTERFACE` 库的作用。但是添加为 `IMPORTED` 以表明库来自其他地方时，这样做的效果是将库的可见性限制为当前目录范围，而不是全局可见性。除了下面讨论的一个例外，添加 `GLOBAL` 关键字将在库中生成 `INTERFACE IMPORTED GLOBAL` 关键字组合，与单独使用 `INTERFACE` 相比，实际差别不大。所以，没必要使用(实际上也禁止) `INTERFACE IMPORTED` 库来设置 `IMPORTED_LOCATION`。

**CMake 3.11之前**，`target_...()` 不能用于在任何类型的 `IMPORTED` 库上设置 `INTERFACE_...` 属性。但是，可以使 `set_property()` 或 `set_target_properties()` 设置这些属性。**CMake 3.11**删除了使用 `target_...()` 来设置这些属性的限制，因此尽管过去的 `INTERFACE IMPORTED` 非常类似于普通 `IMPORTED` 库，但在**CMake 3.11**中，在限制方面更接近于普通的 `INTERFACE` 库。

下表总结了各种关键字组合所支持的功能：

关键字	可见性	导入位置信息	设置接口属性	可安装性
<code>INTERFACE</code>	全局	禁止	任何形式	<code>Yes</code>
<code>IMPORTED</code>	本地	需要	受限\$^*\$	<code>No</code>
<code>IMPORTED GLOBAL</code>	全局	需要	受限\$^*\$	<code>No</code>
<code>INTERFACE IMPORTED</code>	本地	禁止	受限\$^*\$	<code>No</code>
<code>INTERFACE IMPORTED GLOBAL</code>	全局	禁止	受限\$^*\$	<code>No</code>

\$^\*\$只有在使用**CMake 3.11**或更高版本时，才能使用 `target_...()` 命令来设置 `INTERFACE_...` 属性。任何版本的**CMake**中，`INTERFACE_...` 都可以用 `set_property()` 或 `set_target_properties()` 进行设置。

人们可能会认为不同的接口和导入库的组合过于复杂和混乱。然而，对于大多数开发人员来说，导入的目标通常是在幕后创建的，看起来和常规的目标无异。上表中的所有组合，只有普通 `INTERFACE` 目标通常由项目直接定义。第25章，将介绍其他组合的动机和机制。

`add_library()` 的最后一种形式用于定义别名库:

```
add_library(aliasName ALIAS otherTarget)
```

库别名类似于可执行别名。它作为引用另一个库的只读方式，但不创建新的构建目标。无法安装库别名，也无法将其定义为另一个别名。**CMake 3.11**之前，不能为导入的目标创建别名库，但是在**CMake 3.11**中对导入的目标进行了更改，这个限制放宽了，现在可以为全局可见的导入目标创建别名了。

库别名的一个常见的用法与**CMake 3.0**中引入的特性有关。对于将要安装或打包的每个库，创建匹配的别名库，其名称为 `projNamespace::originalTargetName`。一个项目中的所有这些别名通常都共享相同的 `projNamespace`。例如：

```
# Any sort of real library (SHARED, STATIC, MODULE
# or possibly OBJECT)
add_library(myRealThings SHARED src1.cpp ...)
add_library(otherThings STATIC srcA.cpp ...)

# Aliases to the above with special names
add_library(BagOfBeans::myRealThings ALIAS myRealThings)
add_library(BagOfBeans::otherThings ALIAS otherThings)
```

项目本身中，其他目标将链接到实际目标或命名空间目标(两者具有相同的效果)。使用别名的动机来自于，安装项目和其他连接到由安装/打包的配置文件创建的导入目标的内容。这些配置文件将定义导入库的名称空间名称，而不只是原始名称。然后，项目将链接到带有命名空间的名称上。例如：

```
# Pull in imported targets from an installed package.
# See details in Chapter 23: Finding Things
find_package(BagOfBeans REQUIRED)

# Define an executable that links to the imported
# library from the installed package
add_executable(eatLunch main.cpp ...)
target_link_libraries(eatLunch PRIVATE
    BagOfBeans::myRealThings
)
```

在某个时候，上面的项目想要将 `BagOfBeans` 项目直接合并到它自己的构建中，而不是寻找一个已安装的包，它可以这样做而不改变它的链接关系，因为 `BagOfBeans` 项目为命名空间名称提供了一个别名：

```
# Add BagOfBeans directly to this project, making
# all of its targets directly available
add_subdirectory(BagOfBeans)

# Same definition of linking relationship still works
add_executable(eatLunch main.cpp ...)
target_link_libraries(eatLunch PRIVATE
    BagOfBeans::myRealThings
)
```

**CMake**将始终将具有双冒号( `::` )的名称，视为别名或导入目标的名称。对不同的目标类型，使用这样的名称都会导致错误。也许，当目标名称用作 `target_link_library()` 的一部分时，如果**CMake**不知道该名称中的目标时将报错。将其与普通名称进行比较，如果**CMake**不知道由该名称提供的目标，则将其视为假定由系统提供的库，这会导致错误在构建时变得很明显。

```

add_executable(main main.cpp)
add_library(bar STATIC ...)
add_library(foo::bar ALIAS bar)

# Typo in name being linked to, CMake will assume a
# library called "bart" will be provided by the
# system at link time and won't issue an error.
target_link_libraries(main PRIVATE bart)

# Typo in name being linked to, CMake flags an error
# at generation time because a namespaced name must
# be a CMake target.
target_link_libraries(main PRIVATE foo::bart)

```

因此，链接到带有名称空间的名称会更加健壮。强烈建议项目至少为要安装/打包的所有目标，定义带有名称空间的别名。这种带有名称空间的别名甚至可以在项目本身中使用，而不仅仅是将其作为预构建包或子项目在其他项目中使用。

## 16.3. 快速导入目标

不使用 `GLOBAL` 关键字进行定义时，导入的目标只在创建它们的目录作用域中(或子作用域中)可见，这种行为源于作为查找模块或包配置文件的一部分。`Find`模块或包配置文件定义的任何内容通常都具有局部可见性，因此通常不应该添加全局可见性的目标。这允许项目层次结构的不同部分，可以以不同的设置拉入相同的包和模块，而不相互干扰。

在某些情况下，需要创建具有全局可见性的导入目标，例如确保在整个项目中始终如一地使用特定包的相同版本或实例。创建导入库时添加 `GLOBAL` 关键字可以实现这一点，但项目可能无法控制执行创建的命令。为了给项目提供解决这种情况的方法，`CMake 3.11`引入了通过将目标的 `IMPORTED_GLOBAL` 属性设置为 `true` 来提升导入目标的全局可见性的能力。请注意，这是单向转换，不能将全局目标降级为局部可见性。

```

# Imported library created with local visibility.
# This could be in an external file brought in
# by an include() call rather than in the same
# file as the lines further below.
add_library(builtElsewhere STATIC IMPORTED)
set_target_properties(builtElsewhere PROPERTIES
  IMPORTED_LOCATION /path/to/libSomething.a
)

# Promote the imported target to global visibility
set_target_properties(builtElsewhere PROPERTIES
  IMPORTED_GLOBAL TRUE
)

```

必须注意的是，导入的目标只有在定义与在相应的作用域中才能提升。父或子范围中定义的导入目标时，不能进行提升。`include()` 命令没有引入新的目录作用域，`find_package()` 调用也没有引入，因此可以通过提升导入到构建中定义的目标。事实上，促进导入目标的能力的主要用例。还应该注意的是，一旦导入的目标提升为全局可见，就能够支持创建该目标的别名。

## 16.4. 总结

**CMake** 3.0版本对项目管理目标之间的依赖关系和需求的方式进行了重大更改，而不是通过变量指定大多数事情，然后手动管理项目，或目录级命令将适用于所有目标目录，每个目标有在自己的属性上获得必要信息的能力。这种转变焦点的靶点驱动(**target-centric**)模式，可让伪目标类型更好灵活和准确的表达内部目标间的关系。特别是，开发人员应该熟悉接口库，因为它们为捕获和表达关系提供了很大帮助，而无需创建或引用文件。它们对表示头文件库、资源集合和许多其他场景的详细信息很有用，强烈推荐使用它们，而不是试图使用变量或目录级命令来实现相同的结果。

当项目开始使用外部构建的包，或者引用**Find**模块找到的文件系统中的工具，就会遇到导入的目标情况。开发人员应该习惯使用导入目标，但是通常没有必要了解它们如何定义的所有细节，除非需要编写**Find**模块或手动创建一个包的配置文件。第25章中讨论了一些具体的情况，在安装时开发人员可能会遇到导入目标的某些限制，但是这样的情况并不常见。

许多旧的**CMake**模块只提供变量来引用导入的实体。从**CMake** 3.0开始，这些模块正在逐步更新，以提供适当的导入目标。对于那些需要引用外部工具或库的项目，如果可以，最好通过导入目标来引用。这些通常可以更好地抽象出诸如平台差异、选择依赖选项的工具等内容。更重要的是，使用需求随后可由**CMake**处理。如果可以选择导入库或引用相同内容的变量，请尽可能选择导入库。

比起对象库，定义静态库更好一些。静态库更简单，从早期的**CMake**版本得到了更完整和健壮的支持，并且对于大多数开发人员很好地理解。对象库有其用途，但是灵活性不如静态库。特别是，对象库不能链接(**CMake** 3.12之前)，因此不支持传递依赖关系。这迫使项目手动应用此类依赖关系，从而增加了出现错误和遗漏的机会，还减少了库目标通常提供的封装。甚至名称也会在开发人员中引起一些误会，因为对象库不是真正的库，而只是一组未组合的对象文件，但开发人员有时希望它表现得像一个真正的库。**CMake** 3.12的变化模糊了这一区别，但剩下的差异仍然为出现意外留下有空间。

命名目标时，不要使用过于通用的目标名称。全局可见的目标名称必须是唯一的，并且当在较大的层次结构安排中使用时，名称可能与其他项目的目标冲突。另外，考虑为不是项目私有的目标(最终被安装或打包的每个目标)添加一个别名 `namespace::target`。这允许项目链接到带有命名空间的目标名称，而不是真正的目标名称，这种技术使项目能够相对容易地在构建子项目本身和使用预构建安装的项目之间进行切换。虽然，这最初看起来是没有多少收获的额外工作，但它正在成为**CMake**社区中预期的标准实践，特别是对于那些需要花费大量时间来构建的项目。这个模式将在第25.3节中进一步讨论。

不可避免的是，在某些情况下，重命名或重构一个库可能是可取的，但是可能存在现有库目标可以链接到的外部项目。这种情况下，可以使用别名目标为重命名的目标提供旧名称，以便那些外部项目可以在方便的时候继续构建和更新。分割库时，使用旧的目标名称定义接口库，同时定义库的链接依赖关系。例如：

```
# Old library previously defined like this:  
add_library(deepCompute SHARED ...)  
  
# Now the library has been split in two, so define  
# an interface library with the old name to effectively  
# forward on the link dependency to the new libraries  
add_library(computeAlgoA SHARED ...)  
add_library(computeAlgoB SHARED ...)  
  
add_library(deepCompute INTERFACE)  
target_link_libraries(deepCompute INTERFACE  
    computeAlgoA  
    computeAlgoB  
)
```

# 第17章：自定义任务

构建工具不可能实现项目需要的所有特性。某些情况下，开发人员将需要执行超出支持范围的功能。例如，可能需要运行特殊的工具来生成源文件，或者在生成目标之后对其进行处理。可能需要复制、验证文件或计算哈希值；可能需要归档构建或联系通知服务。这些任务并不总是可预料的，该模式可以通用构建系统提供这些功能。

**CMake**通过自定义命令和自定义目标来完成任务。这些命令允许在构建时执行命令(或一组命令)，以完成项目的要求。**CMake**还支持在配置时执行任务，这些需要依赖构建之前完成的任务，甚至在处理当前 `CMakeLists.txt` 前完成的任务。

## 17.1. 自定义目标

项目可以定义自己的自定义目标，这些目标在构建时执行。使用 `add_custom_target()`，可以定义任意自定义目标：

```
add_custom_target(  
    targetName [ALL]  
    [command1 [args1...]]  
    [COMMAND command2 [args2...]]  
    [DEPENDS depends1...]  
    [BYPRODUCTS [files...]]  
    [WORKING_DIRECTORY dir]  
    [COMMENT comment]  
    [VERBATIM]  
    [USES_TERMINAL]  
    [SOURCES source1 [source2...]]  
)
```

指定 `targetName` 为新目标，`ALL` 选项使所有目标都依赖于这个新目标(各种生成器对`ALL`目标的命名略有不同，通常类似于 `all`、`ALL`)。如果没有 `ALL` 选项，只有在明确目标或构建其他依赖于该目标时才会构建。自定义目标总认为是过时的，因此任何依赖于它的目标更新时，都会重新执行自定义目标的命令。

构建自定义目标时，将按照给定的顺序执行命令，每个命令都有任意数量的参数。为改善可读性，参数可以跨越多个行。第一个命令不需要 `COMMAND` 关键字，但是为了清晰，建议在第一个命令中也包含 `COMMAND` 关键字。指定多个命令时更要这样做，为每个命令使用一致的形式。

可以定义命令来执行在平台可执行的任何操作。典型的命令涉及运行脚本或可执行文件，但也可以作为构建的一部分创建的可执行目标。如果将另一个可执行目标的名字是列为执行命令，**CMake**将自动为其他目标的可执行文件更换的构建位置。不管使用的是平台还是**CMake**生成器，这都有效，因此不必担心不同输出目录结构、文件名等的各种平台和生成器的差异。如果需要使用另一个目标作为其中一个命令的参数(**CMake**不会自动执行替换)，使用 `TARGET_FILE` 生成器表达式进行替换会更加简单。项目可以利用这些特性，让**CMake**提供目标位置，而不是手工硬编码路径，这样可以让项目以最小的成本支持所有平台和生成器。下面的例子展示了如何定义自定义的目标，并使用另外两个目标作为命令和参数列表的一部分：

```
add_executable(hasher hasher.cpp)  
add_library(myLib api.cpp)  
add_custom_target(createHash  
    COMMAND hasher $<TARGET_FILE:myLib>  
)
```

当目标作为执行命令时，**CMake**会自动在可执行目标上创建依赖项，以确保在构建自定义目标前构建。如果生成器表达式中的命令或参数引用了该目标，也会自动在该目标上创建依赖项。如果需要任何其他目标上的依赖项，可以使用 `add_dependencies()`。如果依赖关系存在于文件而不是目标上，则可以使用 `DEPENDS` 关键字在 `add_custom_target()` 中指定。注意，依赖关系不应用于目标依赖关系，而应仅用于文件依赖关系。当文件是由其他自定义命令生成时，`DEPENDS` 关键字尤其有用(参见17.3节)，**CMake**将设置必要的依赖关系，以确保其他自定义命令在构建这个自定义目标前执行。由于遗留特性允许对多个路径进行匹配，所以对于 `DEPENDS` 始终使用绝对路径，设置相对路径可能会出现意外。

提供多个命令时，每一个命令都会列出顺序执行。但是，项目不应该假定任何**Shell**环境行为，因为每个命令可能在单独的**Shell**环境中运行，或者根本不需要任何**Shell**环境。自定义命令应该独立执行，没有任何外壳特性(如重定向、变量替换等)，只按顺序执行命令。其中一些特性可以在某些平台上工作，但并没有得到普遍的支持。另外，由于没有保证特定的**Shell**行为，不同平台上对可执行名称或参数进行转义可能有不同的处理方式。为了减少这些差异，可以使用 `VERBATIM` 选项来确保解析**CMakeLists.txt**文件时，只有**CMake**的部分进行了转义。平台不再执行转义操作，因此开发人员可以对造方式放心。如果希望避免相关性，建议使用 `VERBATIM` 关键字。

默认情况下，执行命令的目录是当前二进制目录。这可以通过 `WORKING_DIRECTORY` 选项进行更改，该选项可以是绝对路径，也可以是相对路径，是相对于当前二进制目录。这意味着不需要使用  `${CMAKE_CURRENT_BINARY_DIR}` 作为工作目录的一部分。

可以使用 `BYPRODUCTS` 选项列出在运行命令时创建的其他文件。如果使用**Ninja**生成器，运行这组自定义命令需要另一个目标创建的文件，需要使用该选项。作为输出的文件，当标记为已生成时(对于所有生成器类型，而不仅仅是**Ninja**)，这可以确保构建工具知道如何正确处理与输出文件相关的依赖。对于自定义目标将文件作为输出生成的情况，请考虑使用 `add_custom_command()` 是否是输出内容更合适的方法(参见17.3节)。

如果在控制台的命令不产生输出，使用 `COMMENT` 选项指定一条短消息会很有用。指定的消息在运行命令前会记录下来，因此如果命令由于某种原因静默地失败了，注释可以作为标记，指示构建失败的位置。但是请注意，对于一些生成器，注释将不会显示，因此这不是一种可靠的机制。**17.5**节中给出了一个普遍支持的替代方案。

`USES_TERMINAL` 是另一个与控制台相关的选项，它让命令直接访问终端(如果可能的话)。使用**Ninja**生成器时，是将命令放置在控制台中。某些情况下，这可能会有更好的输出缓冲行为，比如：帮助**IDE**捕获并以更及时的方式呈现构建输出。如果非**IDE**构建需要交互式输入，也很有用。**CMake 3.2**和更高版本才支持 `USES_TERMINAL` 选项。

`SOURCES` 选项允许列出任意文件，然后将这些文件与自定义目标关联。这些文件可能在命令中使用，也可能是一些与目标没太大关联的附加文件，比如文档等。带有源的文件对构建或依赖关系没有影响，纯粹是为了将这些文件与目标进行关联，以便**IDE**可以显示它们。这个特性有时可以通过定义虚拟自定义目标和不带命令的源在**IDE**中显示。虽然这样可行，缺点是构建的目标没有意义。许多项目认为这是一个可接受的折衷，有些开发人员认为这是一种反模式。

## 17.2. 添加构建步骤

自定义命令有时不需要定义新目标，可以指定在现有目标构建时要执行的其他步骤。`add_custom_command()` 应该与 `TARGET` 关键字一起使用，如下所示：

```
add_custom_command(  
    TARGET targetName buildStage  
    COMMAND command1 [args1...]  
    [COMMAND command2 [args2...]]  
    [WORKING_DIRECTORY dir]  
    [BYPRODUCTS files...]  
    [COMMENT comment]  
    [VERBATIM]  
    [USES_TERMINAL]  
)
```

大多数选项与 `add_custom_target()` 的非常相似，但是上面的方式没有定义新的目标，而是将命令附加到现有目标上。现有目标可以是可执行目标或库目标，甚至是自定义目标(会有一些限制)。命令将作为构建 `targetName` 的一部分执行， `buildStage` 需要是以下参数之一：

#### `PRE_BUILD`

命令应该在指定目标的其他规则前运行。请注意，只有Visual Studio生成器支持此选项，并且仅适用于Visual Studio 7或更高版本。所有CMake生成器都将其作为 `PRE_LINK` 处理。鉴于对这个选项的支持有限，项目应该不使用 `PRE_BUILD` 自定义命令结构。

#### `PRE_LINK`

命令将在源代码编译后、链接之前运行。对于静态库目标，命令将在库工具打包前运行。自定义目标，不支持 `PRE_LINK`。

#### `POST_BUILD`

命令将在所有其他规则之后运行。所有目标类型和生成器都支持此选项，因此应将其作为构建阶段的首选。

`POST_BUILD` 任务相对比常见，但是很少需要 `PRE_LINK` 和 `PRE_BUILD`，因为可以通过 `add_custom_command()` 的输出形式来避免(参见下一节)。

可以对 `add_custom_command()` 进行多次调用，将多组自定义命令追加到特定目标上，例如：让命令在一个工作目录运行，而其他命令在其他地方运行。

```
add_executable(myExe main.cpp)  
  
add_custom_command(TARGET myExe POST_BUILD  
    COMMAND script1 $<TARGET_FILE:myExe>  
)  
  
# Additional command which will run after the above from a different directory  
add_custom_command(TARGET myExe POST_BUILD  
    COMMAND writeHash $<TARGET_FILE:myExe>  
    BYPRODUCTS ${CMAKE_BINARY_DIR}/verify/myExe.md5  
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/verify  
)
```

## 17.3. 生成命令

将命令定义为目标的附加构建步骤有许多常见的用例。然而，有时项目需要通过运行一个或一系列命令来创建一个或多个文件，该文件的生成实际上并不属于任何现有目标。使用 `add_custom_command()` 的 `OUTPUT` 形式，实现了与 `TARGET` 方式相同的选项，还实现了与依赖项处理和附加到以前的 `OUTPUT` 命令集相关的附加选项。

```

add_custom_command(
    OUTPUT output1 [output2...]
    COMMAND command1 [args1...]
    [COMMAND command2 [args2...]]
    [WORKING_DIRECTORY dir]
    [BYPRODUCTS files...]
    [COMMENT comment]
    [VERBATIM]
    [USES_TERMINAL]
    [APPEND]
    [DEPENDS [depends1...]
    [MAIN_DEPENDENCY depend]
    [IMPLICIT_DEPENDS <lang1> depend1
    [<lang2> depend2...]]
    [DEPFILE depfile]
)

```

与指定构建前/构建后阶段不同，此方式要求在 `OUTPUT` 关键字之后提供一个或多个输出文件名，**CMake**将这些命令解释为生成指定输出文件的方法。如果输出文件没有指定路径或指定了相对路径，则是相对于当前二进制目录。

因为没有目标定义，所以这种形式不会构建输出文件。但如果同一目录范围内定义的其他目标依赖于任何输出文件，**CMake**将自动创建依赖关系，以确保输出文件在需要的目标之前生成。该目标可以是普通的可执行文件、库目标，甚至可以是自定义目标。实际上，定义自定义目标只是为开发人员提供一种触发自定义命令的方法。下面对上一节的哈希示例进行了修改，演示了这种方法：

```

add_executable(myExe main.cpp)

# Output file with relative path, generated in the build directory
add_custom_command(OUTPUT myExe.md5
    COMMAND writeHash ${TARGET_FILE:myExe}
)

# Absolute path needed for DEPENDS, otherwise relative to source directory
add_custom_target(computeHash
    DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/myExe.md5
)

```

当以这种方式定义时，构建的 `myExe` 目标不会运行生成哈希值的步骤，这与前面的示例不同，该示例将哈希命令添加为 `myExe` 目标的 `POST_BUILD` 步骤。相反，在开发人员明确请求将其作为构建目标时，才会生成哈希值。这允许在需要时定义和调用可选步骤，如果额外的步骤很耗时或不总相关时，就很有用。

`add_custom_command()` 也可以生成目标使用的文件，比如：生成源文件。接下来的例子中，生成一个源文件，然后作为可执行项目的一部分进行编译。

```

add_executable(generator generator.cpp)

add_custom_command(OUTPUT onTheFly.cpp
    COMMAND generator
)

add_executable(myExe ${CMAKE_CURRENT_BINARY_DIR}/onTheFly.cpp)

```

**CMake**自动识别出 `myExe` 需要自定义命令生成的源文件，而源文件需要生成器可执行文件。请求要构建的 `myExe` 目标，将在构建 `myExe` 之前构建生成器和生成的源文件。但请注意，这种依赖关系有局限性。考虑以下情况：

- `onTheFly.cpp`最初并不存在。
- 构建 `myExe` 目标的结果如下:
  - 生成器目标是新的。
  - 执行自定义命令来创建`onTheFly.cpp`。
  - 构建 `myExe` 目标。
- 现在修改`generator.cpp`文件。
- 再次构建 `myExe` 目标，这次的结果如下:
  - 生成器目标是最新的。因为源文件修改了，这将导致生成器重新生成可执行文件。
  - 因为`onTheFly.cpp`已经存在，所以不执行自定义命令。
  - 因为源文件保持不变，不会重新构建 `myExe` 目标。

人们可能会惯性地认为，如果重新构建生成器目标，那么也应该重新运行自定义命令。**CMake**自动创建的依赖项不会强制执行，其创建了一个较弱的依赖项，确保生成器是最新的，但自定义命令只在输出文件完全找不到时运行。为了在生成器目标重建时重新运行自定义命令，必须显式指定依赖项，而不是使用**CMake**自动创建的依赖项。

可以使用 `DEPENDS` 手动指定依赖项的配置项，可以是**CMake**目标或文件(与之相比，`add_custom_target()` 的 `DEPENDS` 选项只能是配置文件)。如果是目标，需要在自定义命令的输出文件更新时更新。如果修改了配置的文件，如果有任何需要自定义命令输出的文件，则将执行自定义命令。此外，如果任何配置文件本身是同一目录范围内的另一个自定义命令的输出文件，则首先执行另一个自定义命令。对于 `add_custom_target()`，如果 `DEPENDS` 的依赖项中有文件，则始终使用绝对路径，以避免未定义行为。

虽然**CMake**的自动依赖关系看起来很方便，但项仍需要使用 `DEPENDS` 配置所需要的目标和文件，以确保依赖关系的完整性。因为第一次构建将运行自定义命令创建输出文件，很容易错误地省略 `DEPENDS` 部分。除非删除输出文件，否则后续构建不会重新运行自定义命令。这一点在复杂的项目中，常常很难发现，直到开发人员遇到这种情况，并试图找出为什么没有在预期时重建才能发现问题。因此，开发人员应该配置依赖部分，除非自定义命令不需要构建创建的任何内容或项目的任何文件。

另一个常见错误是没有在自定义命令需要的文件上创建依赖项，但是该文件没有作为执行命令的一部分。这样的文件需要出现在 `DEPENDS` 中，这样构建才健壮。

`add_custom_command()` 还支持其他与依赖相关的选项。`MAIN_DEPENDENCY` 用于标识源文件，该文件视为自定义命令的主要依赖。对于列出的文件，效果与 `DEPENDS` 相同，一些生成器会应用额外的逻辑，比如：在**IDE**项目中将特定的命令在某处执行。如果源文件为 `MAIN_DEPENDENCY`，然后自定义命令成为源文件的替代品，依赖项会在编译前执行。这可能会导致一些意想不到的结果。考虑下面的例子：

```
add_custom_command(OUTPUT transformed.cpp
  COMMAND transform
  ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
  transformed.cpp
  MAIN_DEPENDENCY ${CMAKE_CURRENT_SOURCE_DIR}/original.cpp
)
add_executable(original original.cpp)
add_executable(transformed transformed.cpp)
```

因为`original.cpp`不会编译到对象文件中，上面的操作会导致原始目标的链接器报错，所以根本不会有对象文件(因此也不会有`main()`函数)产生。因为构建工具把`original.cpp`作为用于创建`transformed.cpp`的输入文件。这个问题可以通过 `DEPENDS` 来解决，而不是 `MAIN_DEPENDENCY`，因为这会保留依赖关系，不会替换`original.cpp`源文件的编译规则。

大多数项目的生成器不支持其他两个与依赖选项 `IMPLICIT_DEPENDS` 和 `DEPFILE`。除了**Makefile**生成器外，其他所有生成器都忽略 `IMPLICIT_DEPENDS`。如果使用的不是**Ninja**生成器，使用 `DEPFILE` 时会报错。`IMPLICIT_DEPENDS` 指示**CMake**调用C或C++扫描器来确定文件的依赖关系，而 `DEPFILE` 可生成特定于**Ninja**的 `.d` 依赖关系文件。因

为支持这两个选项的生成器数量有限，所以项目通常要避免使用它们。

将更多依赖项或命令附加到同一个输出文件或目标时，`OUTPUT` 和 `TARGET` 的行为也略有不同。对于 `OUTPUT`，必须指定 `APPEND` 关键字，并且对于随后对 `add_custom_command()` 的使用中，第一个 `OUTPUT` 文件必须相同。`COMMAND` 和 `DEPENDS` 可用于对同一输出文件的第二次调用和后续调用，当存在 `APPEND` 关键字时，将忽略其他选项，如 `MAIN_DEPENDENCY`、`WORKING_DIRECTORY` 和 `COMMENT`。对于 `TARGET`，同一目标的第二次调用和后续调用 `add_custom_command()`，不需要附加关键字。还可以为每次调用指定 `COMMENT` 和 `WORKING_DIRECTORY` 选项。

## 17.4. 配置时任务

`add_custom_target()` 和 `add_custom_command()` 定义了在构建阶段要执行的命令。这是典型的自定义命令，但一些情况下，自定义任务需要在配置阶段执行，例如：

- 执行外部命令以获取配置期间的信息。
- 更新文件的任何时候重新运行CMake。
- 生成CMakeLists.txt或其他需要作为当前配置步骤的文件。

CMake提供 `execute_process()`，用于在配置阶段执行命令：

```
execute_process(  
    COMMAND command1 [args1...]  
    [COMMAND command2 [args2...]]  
    [WORKING_DIRECTORY directory]  
    [RESULT_VARIABLE resultVar]  
    [RESULTS_VARIABLE resultsVar]  
    [OUTPUT_VARIABLE outputVar]  
    [ERROR_VARIABLE errorVar]  
    [OUTPUT_STRIP_TRAILING_WHITESPACE]  
    [ERROR_STRIP_TRAILING_WHITESPACE]  
    [INPUT_FILE inFile]  
    [OUTPUT_FILE outFile]  
    [ERROR_FILE errorFile]  
    [OUTPUT_QUIET]  
    [ERROR_QUIET]  
    [TIMEOUT seconds]  
)
```

与 `add_custom_command()` 和 `add_custom_target()` 类似，一个或多个命令指定了任务，`WORKING_DIRECTORY` 选项可用于控制运行命令的位置。命令传递到操作系统执行，没有中间Shell环境。因此，不支持输入/输出重定向和环境变量这样的特性，命令会立即运行。

多个命令会按顺序执行，但不是完全独立，而是将命令的标准输出通过管道传输到下一个命令作为输入。没有任何其他选项的情况下，最后一个命令的输出发送到CMake进程本身，但每个命令的标准错误会发送到CMake进程的标准错误流中。

可以捕获标准输出流和错误流并存储在变量中，最后一个命令的输出命令的集合可以指定变量来存储 `OUTPUT_VARIABLE`。类似地，所有命令的错误流都可以存储在 `ERROR_VARIABLE` 选项的变量中。将相同的变量名传递给这两个选项将导致标准输出和错误输出合并，并将合并的结果存储在指定的变量中。如果 `OUTPUT_STRIP_TRAILING_WHITESPACE` 选项存在，将从存储在输出变量中的内容中删除尾随的空白，而 `ERROR_STRIP_TRAILING_WHITESPACE` 选项将对存储在错误输出变量中的内容执行类似的操作。如果使用输出或错误变量的内容进行字符串比较，常见的问题是是否有考虑尾随的空格，因此需要删掉这些空格。

可以将标准输出流和错误输出流发送到文件中，`OUTPUT_FILE` 和 `ERROR_FILE` 选项可用于指定流发送到文件的名称，两个选项填写一样，则为合并流中的结果到指定的文件中。此外，可以使用 `INPUT_FILE` 选项为第一个命令的输入流指定文件。但请注意，`OUTPUT_STRIP_TRAILING_WHITESPACE` 和 `ERROR_STRIP_TRAILING_WHITESPACE` 选项对发

送文件的内容没有影响。

不能捕获同一流并同时发送到文件，但可以将不同的流发送到不同的位置，比如将输出流发送到变量，将错误流发送到文件，反之亦然。还可以使用 `OUTPUT_QUIET` 和 `ERROR_QUIET` 选项以静默的方式丢弃流。如果只关心命令的成功或失败，这些选项非常有用。

可以使用 `RESULT_VARIABLE` 选项捕获命令集的成功或失败状态。运行命令的结果将以最后一个命令的整数返回码或包含某种错误消息字符串的形式，存储在指定的变量中。`if()` 将非空的错误字符串和0以外的整数值都处理为布尔值`true`(除非某个项目有满足特殊情况的错误字符串，参见6.1.1节)。因此，检查 `execute_process()` 调用是否成功通常比较简单：

```
execute_process(COMMAND runSomeScript
    RESULT_VARIABLE result)
if(result)
    message(FATAL_ERROR "runSomeScript failed: ${result}")
endif()
```

**CMake 3.10** 中，如果需要每条单独命令的结果，可以使用 `RESULTS_VARIABLE` 选项。该选项将每条命令的结果以列表的方式存储在 `resultsVar` 变量中。

`TIMEOUT` 选项可用于处理运行时间超过预期或可能永远无法完成的命令。这确保了配置步骤不会无限期地阻塞，并允许将异常的时间配置视为错误。但请注意，`TIMEOUT` 选项本身不会让**CMake**停止并报错。仍然需要使用 `RESULT_VARIABLE` 捕获该命令的结果，然后必须检查该变量，如前面的示例所示，如果命令运行的时间超过了超时阈值，`result` 变量将保存一个错误字符串，提示由于超时而终止命令，这就是为什么建议打印 `result` 变量的原因。

**CMake** 执行命令时，子进程很大程度上继承了与主进程相同的环境。一个例外是**CMake**首次运行在项目中，子进程的 `cc` 和 `cxx` 环境变量显式地设置为C和C++编译器(如果主要项目使得C和C++语言)。对于后续的**CMake**运行，`cc` 和 `cxx` 环境变量不会以这种方式进行替换，如果执行的操作依赖于 `cc` 和/或调用 `execute_process()` 时具有相同值的 `cxx`，可能会导致意外的结果。这种未文档化的行为在早期的**CMake**中就已经存在了，甚至可以追溯到已被 `execute_process()` 替换的 `exec_program()` 命令。添加它是为了方便子进程配置和运行与主项目使用相同编译器。在某些情况下，子进程可能不希望保留编译器，比如当主构建是交叉编译的时候，子进程使用默认的主机编译器。这种情况下，可以将名为 `CMAKE_GENERATOR_NO_COMPILER_ENV` 的变量设置为布尔`true`，这样**CMake**就不会为任何 `execute_process()` 调用(甚至是初始调用)设置 `cc` 和 `cxx`。

## 17.5. 平台独立命令

`add_custom_command()`、`add_custom_target()` 和 `execute_process()` 为项目提供了很大的自由度。任何**CMake**不直接支持的任务，都可以使用主机操作系统提供的命令来实现。这些自定义命令本质上适用于特定的平台，许多项目使用**CMake**的一个主要原因是为了抽象平台差异，或者至少以最小的成本支持了一系列平台。

大部分自定义任务都与文件系统操作相关。创建、删除、重命名或移动文件和目录构成了这些任务的大部分，但是执行这些任务的命令因操作系统的不同而有所不同。因此，项目经常使用 `if-else` 条件来定义同一命令的不同平台版本，更糟的是，他们会费心费力为某些平台实现命令。许多开发人员不知道`cmake`命令本身提供了命令模式，它抽象了许多平台的特定任务：

```
cmake -E cmd [args...]
```

支持的完整的命令集可以使用 `cmake -E help` 列出，但更常用命令包括：

- `compare_files`
- `copy`

- `copy_directory`
- `copy_if_different`
- `echo`
- `env`
- `make_directory`
- `md5sum`
- `remove`
- `remove_directory`
- `rename`
- `tar`
- `time`
- `touch`

考虑一个自定义任务的例子，删除某个目录及其所有内容：

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")

# Naive platform specific implementation (not robust)
if(WIN32)
  add_custom_target(myCleanup
    COMMAND rmdir /S /Q "${discardDir}"
  )
elseif(UNIX)
  add_custom_target(myCleanup
    COMMAND rm -rf "${discardDir}"
  )
else()
  message(FATAL_ERROR "Unsupported platform")
endif()

# Platform independent equivalent
add_custom_target(myCleanup
  COMMAND "${CMAKE_COMMAND}" -E remove_directory "${discardDir}"
)
```

特定于平台的实现显示了项目通常如何尝试实现这样的场景，但是 `if-else` 条件测试的是目标平台，而不是主机平台。在交叉编译场景中，这可能会导致使用错误的平台命令。然而，平台独立版本总是为主机平台选择正确的命令。

该示例还展示了如何正确使用`cmake`命令。`CMAKE_COMMAND` 变量由CMake填充，它包含主构建中使用的CMake可执行文件的完整路径。以这种方式使用 `CMAKE_COMMAND` 可以确保定制命令也使用相同版本的CMake。`cmake` 可执行文件不需要在当前路径上，如果安装了多个版本的`cmake`，总会使用正确的版本，而不管用户的路径选择了哪个版本。即使用户的 `PATH` 环境变量发生了变化，它还会在构建阶段使用与在配置阶段相同的CMake版本。

本章的前面，我们注意到 `add_custom_target()` 和 `add_custom_command()` 的 `COMMENT` 选项并不可靠。与 `COMMENT` 不同，项目可以使用 `-E echo` 命令在自定义命令序列的任何地方进行注释：

```
set(discardDir "${CMAKE_CURRENT_BINARY_DIR}/private")
add_custom_target(myCleanup
  COMMAND ${CMAKE_COMMAND} -E echo "Removing ${discardDir}"
  COMMAND ${CMAKE_COMMAND} -E remove_directory "${discardDir}"
  COMMAND ${CMAKE_COMMAND} -E echo "Recreating ${discardDir}"
  COMMAND ${CMAKE_COMMAND} -E make_directory "${discardDir}"
)
```

**CMake**的命令模式非常有用，可以独立于平台的方式执行常见任务。然而，有时需要更复杂的逻辑，而此类自定义任务通常使用特定于平台的**Shell**脚本实现。另一种方法是使用**CMake**本身作为脚本引擎，提供一种独立于平台的语言来表达任意逻辑。`cmake`命令的-P选项可以使`cmake`进入脚本模式：

```
cmake [options] -P filename
```

文件名参数是要执行的**CMake**脚本文件的名称。支持**CMakeLists.txt**语法，但没有配置或生成步骤，而且不会更新**CMakeCache.txt**文件。脚本文件本质上只是作为一组命令处理，而不是作为一个项目，因此与构建目标或项目级特性相关的任何命令都不受支持。尽管如此，脚本模式允许实现复杂的逻辑，而且优点是不需要安装任何额外的解释器。

虽然脚本模式不像普通的**Shell**或命令解释器那样支持命令行选项，但它支持传递带有-D选项的变量，就像普通的`cmake`调用一样。由于没有以脚本模式更新**CMakeCache.txt**文件，因此可以自由使用-D选项，而不会影响主构建的缓存。所以，这些选项必须放在-P之前。

```
cmake -DOPTION_A=1 -DOPTION_B=foo -P myCustomScript.cmake
```

## 17.6. 结合不同的方法

下面的例子演示了本章中介绍的许多特性。特别地，展示了如何使用指定自定义任务的不同方法来完成重要的任务。

*CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.0)
project(Example)

# Define an executable which generates various files in a
# directory passed as a command line argument
add_program(generateFiles generateFiles.cpp)

# Create a custom target which invokes the above executable
# after creating an empty output directory for it to populate,
# then invoke a script to archive that directory's contents
# and print the MD5 checksum of that archive
set(outDir "foo")
add_custom_target(archiver
  COMMAND ${CMAKE_COMMAND} -E echo "Archiving generated files"
  COMMAND ${CMAKE_COMMAND} -E remove_directory "${outDir}"
  COMMAND ${CMAKE_COMMAND} -E make_directory "${outDir}"
  COMMAND generateFiles "${outDir}"
  COMMAND ${CMAKE_COMMAND} "-DTAR_DIR=${outDir}"
  -P "${CMAKE_CURRENT_SOURCE_DIR}/archiver.cmake"
)
```

*archiver.cmake*

```

cmake_minimum_required(VERSION 3.0)

if(NOT TAR_DIR)
  message(FATAL_ERROR "TAR_DIR must be set")
endif()

# Create an archive of the directory
set(archive archive.tar)
execute_process(COMMAND ${CMAKE_COMMAND} -E tar cf ${archive} "${TAR_DIR}"
  RESULT_VARIABLE result
)
if(result)
  message(FATAL_ERROR "Archiving ${TAR_DIR} failed: ${result}")
endif()

# Compute MD5 checksum of the archive
execute_process(COMMAND ${CMAKE_COMMAND} -E md5sum ${archive}
  OUTPUT_VARIABLE md5output
  RESULT_VARIABLE result
)
if(result)
  message(FATAL_ERROR "Unable to compute md5 of archive: ${result}")
endif()

# Extract just the checksum from the output
string(REGEX MATCH "^\ *[^\ ]*" md5sum "${md5output}")
message("Archive MD5 checksum: ${md5sum}")

```

## 17.7. 总结

需要执行自定义任务时，最好在构建阶段而不是配置阶段执行。快速配置阶段很重要，因为可以调用时自动修改一些文件(例如：项目的任何CMakeLists.txt文件，任何文件包含由CMakeLists.txt文件或任何文件列出的来源`configure_file()`命令会在下一章中讨论)。出于这个原因，如果有选择的话，最好使用`add_custom_target()`或`add_custom_command()`，而不是`execute_process()`。

与平台相关的命令经常与`add_custom_command()`、`add_custom_target()`和`execute_process()`一起使用，但这些命令通常可以使用CMake的命令模式(-E)以平台无关的方式表示。可能的情况下，最好使用与平台无关的命令。此外，CMake还可以用作一种独立于平台的脚本语言，当使用-P选项调用CMake命令序列时，可以将文件处理为CMake命令。使用CMake脚本，而不是特定于平台的Shell或单独安装的脚本引擎，可以降低项目的复杂性，并减少所需的额外依赖。具体来说，考虑CMake的脚本模式是否会比使用Unix Shell脚本或Windows批处理文件，甚至是Python、Perl等语言的脚本更好，因为这些语言可能在所有平台上默认都不可用。下一章将展示如何使用CMake直接操作文件，而不必求助于这些工具和方法。

实现自定义任务时，尽量避免那些不能跨平台提供的通用特性。

- 最好使用命令模式-E echo，而不是使用带有`COMMENT`关键字的`add_custom_command()`和`add_custom_target()`。
- 尽量避免在`add_custom_command()`中使用`PRE_BUILD`。
- 考虑`add_custom_command()`使用`IMPLICIT_DEPENDS`或`DEPFILE`选项时，是否会有特定于生成器的行为发生。
- 避免在`add_custom_command()`中将源文件作为`MAIN_DEPENDENCY`，除非替换了该源文件的默认构建规则。

注意自定义任务的输入和输出的依赖关系，确保`add_custom_command()`创建的所有文件都为`OUTPUT`文件。调用`add_custom_command()`或`add_custom_target()`时，将构建目标作为命令或参数，最好显式地设置为`DEPENDS`，而不是依赖于CMake的自动处理。较弱的自动依赖关系可能不会强制执行开发人员所期望的操作。如果列表中的文件`DEPENDS`于`add_custom_target()`或`add_custom_command()`，则始终使用绝对路径以避免路径错误。

调用 `execute_process()` 时，大多数时候应该通过 `RESULT_VARIABLE` 捕获结果，并使用 `if()` 命令测试该命令的状态。包括使用 `TIMEOUT` 选项时，因为 `TIMEOUT` 本身不会产生错误，所以只会保证命令运行的时间不会超过指定的时间。

# 第18章：文件处理

许多项目需要在构建过程中操作文件和目录。这些操作有些简单，有些复杂，常见的包括：

- 构造路径或提取组件的路径。
- 从目录中获取文件。
- 复制文件。
- 使用字符串内容生成文件。
- 使用另一个文件的内容生成文件。
- 读取文件的内容。
- 计算文件的校验和/或哈希值。

**CMake**提供各种相关处理文件和目录的操作。某些情况下，实现同一种操作可能有多种方式，因此了解不同的选择并了解如何有效地使用它们。这些操作有许多会误用，有些是由于在线教程和示例中普遍存在的误用，导致人们相信这是正确的做法。本章将讨论一些有问题的反模式。

**CMake**大部分与文件相关的功能由 `file()` 提供，其他命令也提供了替代方案，或者提供了相关功能。**CMake**的命令模式(前一章中介绍过)也提供了各种与文件相关的特性，这些特性与 `file()` 提供的特性有很大重叠，但只涵盖了 `file()` 的一些场景，而不是多数情况下的替代方案。

## 18.1. 配置路径

文件处理的最基本部分是文件名和路径。项目通常需要从路径中提取文件名、文件后缀等，或者在绝对路径和相对路径之间进行转换。执行这些操作的方法是 `get_filename_component()`，该命令有三种不同的形式。第一种形式允许提取路径或文件名的不同部分：

```
get_filename_component(outVar input component [CACHE])
```

调用的结果存储在 `outVar` 变量中。要从输入中提取的 `component` 由组件指定，组件必须是以下组件之一：

### DIRECTORY

提取输入的路径部分，但不包含文件名。**CMake 2.8.12**之前，这个选项是 `PATH`，为了保持与旧版本的兼容性，它仍然可以作为 `DIRECTORY` 的同义词使用。

### NAME

提取完整的文件名，包括扩展名。这实际上是丢弃了输入的目录部分。

### NAME\\_WE

只提取基本文件名。这是类似的名称，除了文件名的部分，但不包括提取的第一个“.”。

### EXT

这是 `NAME\_WE` 的补充。它从第一个“.”开始提取文件名的扩展部分。

`CACHE` 关键字可选。如果存在，则结果存储为缓存变量而不是常规变量。通常，不希望将结果存储在缓存中，因此不需要使用 `CACHE` 关键字。

```
set(input /some/path/foo.bar.txt)
get_filename_component(path1 ${input} DIRECTORY) # /some/path
get_filename_component(path2 ${input} PATH) # /some/path
get_filename_component(fullName ${input} NAME) # foo.bar.txt
get_filename_component(baseName ${input} NAME_WE) # foo
get_filename_component(extension ${input} EXT) # .bar.txt
```

`get_filename_component()` 的第二种形式用于获取绝对路径:

```
get_filename_component(outVar input component [BASE_DIR dir] [CACHE])
```

这种形式下，输入可以是相对路径，也可以是绝对路径。如果给定了 `BASE_DIR`，相对路径解释为相对于 `dir` 而不是当前源目录(即 `CMAKE_CURRENT_SOURCE_DIR`)。如果输入是绝对路径，则忽略 `BASE_DIR`。

组件决定如何处理符号链接时，会将得到的路径存储在 `outVar`:

#### *ABSOLUTE*

输入的绝对路径而不需要解析符号链接。

#### *REALPATH*

解析符号链接后，输入的绝对路径。

`file()` 命令提供了反操作，将绝对路径转换为相对路径:

```
file(RELATIVE_PATH outVar relativeToDir input)
```

下面演示用法:

```
set(basePath /base)
set(fooBarPath /base/foo/bar)
set(otherPath /other/place)

file(RELATIVE_PATH fooBar ${basePath} ${fooBarPath})
file(RELATIVE_PATH other ${basePath} ${otherPath})

# The variables now have the following values:
# fooBar = foo/bar
# other = ../other/place
```

`get_filename_component()` 的第三种形式可以提取命令行的部分内容:

```
get_filename_component(progVar input PROGRAM
[PROGRAM_ARGS argVar] [CACHE])
```

假设输入是一个可能包含参数的命令行。**CMake**将提取可执行文件的完整路径，该路径将由指定的命令行调用，必要时使用 `PATH` 环境变量解析可执行文件的位置，并将结果存储在 `progVar` 中。如果给定 `PROGRAM_ARGS`，命令行参数集也将作为列表存储在由 `argVar` 变量中。`CACHE` 关键字与 `get_filename_component()` 的其他形式具有相同含义。

**CMake**在文件处理中，项目可以在所有平台上使用正向斜杠作为目录分隔符，**CMake**在项目需要时转换为本地路径。然而，项目有时可能需要显式地在**CMake**和本机路径之间进行转换，比如：使用自定义命令时，需要向需要本机路径的脚本传递路径。对于这些情况，`file()` 提供了另外两种形式，帮助转换平台原生和**CMake**之间的路径:

```
file(TO_NATIVE_PATH input outVar)
file(TO_CMAKE_PATH input outVar)
```

`TO_NATIVE_PATH` 将输入转换为主机平台的本机路径。这相当于使用了正确的目录分隔符(Windows上是反斜杠，其他系统是正斜杠)。`TO_CMAKE_PATH` 将输入中的所有目录分隔符转换为正斜杠。这是CMake对所有平台上的路径使用的表示。输入也可以与平台的 `PATH` 环境变量兼容的方式指定的路径列表，所有冒号分隔符都替换为分号，从而将路径输入转换为CMake的路径。

```
# Unix example
set(customPath /usr/local/bin:/usr/bin:/bin)
file(TO_CMAKE_PATH ${customPath} outVar)
# outVar = /usr/local/bin;/usr/bin;/bin
```

## 18.2. 复制文件

配置阶段或构建过程中，复制文件是比较常见的需求。因为复制文件对大多数用户来说是一项熟悉的任务，所以对于CMake的开发新手来说，用他们已知的方法来实现文件复制很正常。不幸的是，这会导致使用特定于平台的Shell命令(`add_custom_target()` 和 `add_custom_command()`)，有时还会出现依赖问题，需要开发人员多次运行CMake和/或按特定顺序手动构建目标。不过，CMake提供了更好的方案。

本节中，将介绍一些复制文件的方法。有些是为了满足特定的需要，而另一些则是为了更加通用，可以在各种情况下使用。方法在所有平台上的工作方式完全相同。

在配置时常用复制文件的命令，这个命令也是不那么直观的命令。`configure_file()` 允许将单个文件从一个位置复制到另一个位置，可以选择在此过程中执行CMake变量替换。复制是立即执行的，因此是一个配置时操作。该命令的简化形式如下：

```
configure_file(source destination [COPYONLY | @ONLY] [ESCAPE_QUOTES])
```

源文件必须是已经存在的文件，可以是绝对路径，也可以是相对路径，后者相对于当前的源目录(即 `CMAKE_CURRENT_SOURCE_DIR`)。目标不能简单地将文件复制到一个目录中，它必须是文件名，可选路径可以是绝对路径或相对路径。如果目标不是绝对路径，可以相对于当前二进制目录(即 `CMAKE_CURRENT_BINARY_DIR`)。如果目标路径的部分不存在，CMake将创建丢失的目录。注意，经常会看到项目将 `CMAKE_CURRENT_SOURCE_DIR` 或 `CMAKE_CURRENT_BINARY_DIR` 分别作为源和目标路径的一部分，这只会增加不必要的混乱，应该避免。

如果修改源文件，构建则认为目标文件已经过期，并将自动重新运行`cmake`。如果配置和生成时间很长，并且源文件经常被修改，这可能会使开发人员感到沮丧。因此，`configure_file()` 最好只用于不需要经常更改的文件。

执行复制时，`configure_file()` 具有替换CMake变量的能力。如果没有 `COPYONLY` 或 `@ONLY`，源文件中任何看起来像是使用了CMake的变量(即  `${someVar}` )的内容都将替换为该变量的值。如果没有同名变量存在，则替换一个空字符串。表单 `@someVar@` 的字符串也以同样的方式替换。

*CMakeLists.txt*

```
set(FOO "String with spaces")
configure_file(various.txt.in various.txt)
```

*various.txt.in*

```
CMake version: ${CMAKE_VERSION}
Substitution works inside quotes too: "${FOO}"
No substitution without the $ and {}: FOO
Empty ${} specifier gets removed
Escaping has no effect: \${FOO}
@-syntax also supported: @FOO@
```

### *various.txt*

```
CMake version: 3.7.0
Substitution works inside quotes too: "String with spaces"
No substitution without the $ and {}: FOO
Empty specifier gets removed
Escaping has no effect: \String with spaces
@-syntax also supported: String with spaces
```

`ESCAPE_QUOTES` 关键字会为替换的引号前面加上反斜杠。

### *CMakeLists.txt*

```
set(BAR "Some \"quoted\" value")
configure_file(quoting.txt.in quoting.txt)
configure_file(quoting.txt.in quoting_escaped.txt ESCAPE_QUOTES)
```

### *quoting.txt.in*

```
A: @BAR@
B: "@BAR@"
```

### *quoting.txt*

```
A: Some "quoted" value
B: "Some "quoted" value"
```

### *quoting\_escaped.txt*

```
A: Some \"quoted\" value
B: "Some \"quoted\" value"
```

如上例所示，`ESCAPE_QUOTES` 选项会对所有引号进行转义。因此，文件复制时必须注意敏感空间，以及避免引用有替换可能的字符串。

一些文件类型需要保留 `someVar` 不进行替换。一个例子是，正在复制的文件是一个 Unix shell 脚本，其中 `someVar` 是引用 Shell 变量的一种有效且常见的方法。这种情况下，替换可以仅限于使用 `@ONLY` 关键字的 `@someVar@` 形式：

### *CMakeLists.txt*

```
set(USER_FILE whoami.txt)
configure_file(whoami.sh.in whoami.sh @ONLY)
```

### *whoami.sh.in*

```
#!/bin/sh  
  
echo ${USER} > "@USER_FILE@"
```

### *whoami.sh*

```
#!/bin/sh  
  
echo ${USER} > "whoami.txt"
```

还可以使用 `COPYONLY` 关键字完全禁用替换。如果知道不需要替换，那么指定 `COPYONLY` 是个好办法，可以防止不必要的处理和意外的替换。

使用 `configure_file()` 替换文件名或路径时，常见的错误是使用空格和引号。如果源文件需要单个路径或文件名处理，可能需要用引号将替换的变量括起来。这就是上面示例中的源文件使用 `"@USER_FILE@"`，而不是 `@USER_FILE@` 作为文件名写入输出的原因。

使用  `${someVar}` 或 `@someVar@` 替换CMake变量也可以在字符串上执行，而不仅是在文件上。`string()` 也提供相同的功能：

```
string(CONFIGURE input outVar [@ONLY] [ESCAPE_QUOTES])
```

这些选项的含义与 `configure_file()` 相同。如果要复制的内容需要更复杂的步骤，而不仅仅是简单的替换(下一节将给出一个替换示例)，则此方法很有用。

如果不需替换，另一种选择是使用 `file()` 复制或安装，二者都支持相同的选项：

```
file(<COPY|INSTALL> fileOrDir1 [fileOrDir2...]  
DESTINATION dir  
[NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |  
[FILE_PERMISSIONS permissions...]  
[DIRECTORY_PERMISSIONS permissions...]]  
[FILES_MATCHING]  
[PATTERN pattern | REGEX regex] [EXCLUDE]  
[PERMISSIONS permissions...]  
[...]  
)
```

可以将多个文件，甚至整个目录层次结构复制到选定的目录中，如果存在符号链接，也可以保留符号链接。没有指定绝对路径的任何源文件或目录，都视为相对于当前源目录的路径。如果目标目录地址不是绝对的，将解释为相对于当前二进制目录。CMake会根据需要，创建目标目录结构。

如果源文件是一个目录名，将复制到目标文件中。要将目录中的内容复制到目标目录中，需要在源目录中附加一个正斜杠( / )，如下所示：

```
file(COPY base/srcDir DESTINATION destDir) # --> destDir/srcDir  
file(COPY base/srcDir/ DESTINATION destDir) # --> destDir
```

`COPY` 也会复制源文件或目录的权限，而 `INSTALL` 则不保留原始权限。可以用 `NO_SOURCE_PERMISSIONS` 和 `USE_SOURCE_PERMISSIONS` 选项来覆盖这些默认值，也可以用 `FILE_PERMISSIONS` 和 `DIRECTORY_PERMISSIONS` 选项显式地指定权限。权限值依据基于Unix系统支持的权限值：

<b>OWNER_READ</b>	<b>OWNER_WRITE</b>	<b>OWNER_EXECUTE</b>
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

如果在给定的平台上不理解特定权限，则直接忽略它。可以(通常)将多个权限一起列出。例如，一个**Unix Shell**脚本可以复制到当前的二进制目录，如下所示：

```
file(COPY whoami.sh
DESTINATION .
FILE_PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_WRITE
)
```

**COPY** 和 **INSTALL** 还保留复制的文件和目录的时间戳。此外，如果源文件出现在目标文件中，并且具有相同的时间戳，该文件的复制将视为完成。除了默认权限之外，**COPY** 和 **INSTALL** 之间唯一的其他区别是，**INSTALL** 会显示每个复制项的状态消息，而 **COPY** 不显示。这是因为 **INSTALL** 通常用作**CMake**脚本的一部分，以脚本模式运行，通常会显示安装的每个文件名称。

**COPY** 和 **INSTALL** 还支持特定通配符模式和正则表达式，这可用于限制复制哪些文件，并仅对匹配的文件覆盖权限。可以在一个 **file()** 中给出多个模式和正则表达式。通过示例可以很好地展示这种用法。

下面从 **someDir** 复制所有头文件(**.h**)和脚本文件(**.sh**)，但文件名以 **\_private.h** 结尾的头文件除外。保留源文件的目录结构，头文件的权限和来源处一样，而脚本给的是读、写和执行权限。

```
file(COPY someDir
DESTINATION .
FILES_MATCHING
REGEX .*_private\\.h EXCLUDE
PATTERN *.h
PATTERN *.sh
PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
)
```

如果复制整个源，需要为匹配文件给与权限的一个子集，则可以忽略 **FILES\_MATCHING** 关键字，而使用模式和正则表达式用于权限覆盖。

```
file(COPY someDir
DESTINATION .
# Make Unix shell scripts executable by everyone
PATTERN *.sh PERMISSIONS
OWNER_READ OWNER_WRITE OWNER_EXECUTE
GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_EXECUTE
# Ensure only owner can read/write private key files
REGEX _dsa\$|_rsa\$ PERMISSIONS
OWNER_READ OWNER_WRITE
)
```

**CMake**为复制文件和目录提供了第三种选择。尽管 **configure\_file()** 和 **file()** 都在配置时使用，或者在安装时作为**CMake**脚本的一部分，但**CMake**的命令模式也可以用在构建时复制文件和目录。命令模式是作为 **add\_custom\_target()** 和 **add\_custom\_command()** 的一部分，因为它提供了平台独立性命令，所以是复制的首选方式。有三个与复制相关的命令，第一个用于复制单个文件：

```
cmake -E copy file1 [file2...] destination
```

如果只提供了单个源文件，那么 `destination` 会解释为要复制到的文件的名称，除非指定了一个存在目录。当目标是一个存在目录时，源文件将会复制到其中。这种行为与大多数操作系统的复制命令是一致的，但这也意味着依赖于复制操作前文件系统状态。由于这个原因，复制单个文件时始终显式地指定目标文件名更健壮，除非保证目标是一个存在的目录。

如果目的地包括路径(相对或绝对)，CMake将尝试创建目标路径和复制单个源文件一样。所以复制单个文件时，`copy` 命令不需要前面的步骤来确保目标目录存在。但如果列出了多个源文件，`destination` 必须引用存在的目录。CMake的命令模式 `make_directory` 可用于创建不存在(指定)的目录，从而确保这个目录可用(包括任何父目录)。下面展示了如何安全地将命令模式的命令放在一起：

```
add_custom_target(copyOne
  COMMAND ${CMAKE_COMMAND} -E copy a.txt output/textfiles/a.txt
)
add_custom_target(copyTwo
  COMMAND ${CMAKE_COMMAND} -E make_directory output/textfiles
  COMMAND ${CMAKE_COMMAND} -E copy a.txt b.txt output/textfiles
)
```

`copy` 命令始终将源文件复制到目标文件，即使目标文件与源文件相同。这将导致目标时间戳更新，这有时是不希望看到的。如果文件已经匹配，不应该更新时间戳，使用 `copy_if_different` 命令更合适：

```
cmake -E copy_if_different file1 [file2...] destination
```

该命令的功能与 `copy` 命令完全相同，除非目标中已经存在源文件，并且与源文件相同，否则不执行复制。

不仅可以复制个别文件，命令模式也可以复制整个目录：

```
cmake -E copy_directory dir1 [dir2...] destination
```

与文件相关的复制命令不同，会创建目标目录，包括任何中间路径。还要注意 `copy_directory` 将源目录的内容复制到目标目录中，而不是源目录本身。例如，假设一个目录 `myDir` 包含一个文件 `someFile.txt`，并执行以下命令：

```
cmake -E copy_directory myDir targetDir
```

该命令的结果将是 `targetDir` 下包含 `someFile.txt` 包含文件，而不是 `myDir/someFile.txt`。

通常，`configure_file()` 和 `file()` 最适合在配置时复制文件，而CMake的命令模式是在构建时复制的首选方式。虽然可以在配置时结合使用命令模式和 `execute_process()` 来复制文件，但没理由这样做，因为 `configure_file()` 和 `file()` 都更直接，而且还具有在出现错误时自动停止的优点。

## 18.3. 直接读写文件

CMake提供的不仅仅是复制文件的能力，还提供了许多用于读写文件内容的命令。`file()` 命令提供了大量的功能，最简单的是直接写入文件：

```
file(WRITE fileName content)
file(APPEND fileName content)
```

两个命令都将指定的内容写入指定的文件，两者之间的区别是，如果文件名已经存在，`APPEND` 将添加到现有的内容中，而 `WRITE` 将在写入之前丢弃现有的内容。内容就像任何其他函数参数一样，可以是变量或字符串的内容。

```
set(msg "Hello world")
file(WRITE hello.txt ${msg})
file(APPEND hello.txt " from CMake")
```

上面的示例将生成 `hello.txt` 文件，其中包含一行来自CMake的文本 `Hello world`。注意，新行不会自动添加，因此上面示例中的 `APPEND` 中的文本直接续在 `WRITE` 文本之后，而不会中断。要写入换行符，必须包含在传递给 `file()` 的内容中。一种方法是使用跨多行引用的值：

```
file(WRITE multi.txt "First line
Second line
")
```

如果使用CMake 3.0或更高版本，使用lua括号语法有时会更方便，因为可以防止对内容进行任何变量替换。

```
file(WRITE multi.txt [[
First line
Second line
]])

file(WRITE userCheck.sh [=[
#!/bin/bash

[[ -n "${USER}" ]] && echo "Have USER"
]=])
```

上面代码中，要写入到 `multi.txt` 的内容只包含没有特殊字符的简单文本，因此最简单的括号语法(可以省略=字符)就足够了，只留下一对方括号来标记内容的开始和结束。请忽略第一个换行符的行为是如何使命令更具可读性的。

`userCheck.sh` 的内容要有趣得多，突出显示了括号语法的特性。如果没有括号语法，CMake将看到  `${USER}` 部分，并将其视为CMake变量替换，但因为括号语法没有执行这样的替换，所以结果将保持原样。同样的原因，内容中的各种引用字符也不会解释为内容以外的东西。不需要对它们进行转义，防止它们解释为参数的开始或结束。此外，注意嵌入的内容包含一对方括号。这就是开始和结束标记中数量可变的 = 符号要处理的情况，允许选择标记。将多行写入文件且不需要执行替换时，括号语法通常是指定编写内容最便捷的方法。

有时，项目可能需要编写一个文件，其内容取决于构建类型。简单的方法是可以使用 `CMAKE_BUILD_TYPE` 变量替代，但是这对像Xcode或Visual Studio这样的多配置生成器不起作用。所以，可以使用 `file(GENERATE...)` 命令：

```
file(GENERATE
    OUTPUT outFile
    INPUT inFile | CONTENT content
    [CONDITION expression]
)
```

工作原理有点像 `file(WRITE...)`，除了会为当前CMake生成器支持的每种构建类型写入一个文件。`INPUT` 或 `CONTENT` 选项必须出现，但不能同时出现。它们定义了要写入指定输出文件的内容。所有的参数都支持生成器表达式，这就可以为每种构建类型定制文件名和内容。可以使用 `CONDITION` 选项跳过构建类型，对于要跳过的构建类型，表达式的计算值为0，对于生成的构建类型，表达式的计算值为1。

下面的示例，说明如何使用生成器表达式，根据构建类型自定义内容和文件名。

```

# Generate unique files for all but Release
file(GENERATE
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/outFile-$<CONFIG>.txt
  INPUT ${CMAKE_CURRENT_SOURCE_DIR}/input.txt.in
  CONDITION ${NOT:$<CONFIG:Release>>}
)

# Embedded content, bracket syntax does not
# prevent the use of generator expressions
file(GENERATE
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/details-$<CONFIG>.txt
  CONTENT [[
    Built as "<CONFIG>" for platform "<PLATFORM_ID>".
  ]])

```

上面的第一种情况中，`input.txt` 中的任何生成器表达式，将在写入输出文件时计算。这有点类似于 `configure_file()` 替换CMake变量的方式，只不过这次替换的是生成器表达式。第二个例子演示了如何将括号语法与内嵌内容结合起来使用，从而成为定义内联文件内容的方法，这会涉及到生成器表达式和引号。

通常，每种构建类型的输出文件是不同的。但在某些情况下，输出文件可能希望是相同的，比如文件内容不依赖于构建类型，而是依赖于其他生成器表达式。为了支持这样的用例，CMake允许输出文件对不同的构建类型是相同的，但也只有在生成文件内容对于那些构建类型也是相同的情况下。CMake不允许多个 `file(GENERATE...)`，生成相同的输出文件。

与 `file(COPY...)` 类似，`file(GENERATE...)` 只会在内容发生变化时修改输出文件。因此，只有在内容不同时，输出文件的时间戳才会更新。当生成的文件作为构建目标(如生成的头文件)的输入时，可以避免不必要的重新构建。

与大多数CMake命令相比，`file(GENERATE...)` 的行为方式上有一些区别。因为它会计算生成器表达式，所以不能立即写出文件。当这些文件是作为生成阶段的一部分时，生成阶段发生在所有CMakeLists.txt文件处理之后。当 `file(GENERATE...)` 返回时，生成的文件将不存在，因此在配置阶段，这些文件不能作为输入。特别是，生成的文件直到配置阶段结束时才存在，因此不能使用 `configure_file()`、`file(COPY...)` 等方法复制或读取它们。但仍然可以用作构建阶段的输入，例如：生成的源文件或头文件。

另一个需要注意的要点是，CMake 3.10之前，`file(GENERATE...)` 处理相对路径的方式与通常的CMake不同。相对路径的行为却无法确定，通常是相对于当调用cmake的工作目录。这就增大了不可靠性和不一致性，所以在CMake 3.10中，行为就改变了，使输入路径作为相对于当前源目录，输出路径作为相对于当前二进制目录，就像大多数其他路径处理的CMake命令一样。除非最小CMake版本设置为3.10或更高版本，否则项目应该考虑使用 `file(GENERATE...)` 时使用的相对路径靠不靠谱。

`file()` 不仅可以复制或创建文件，还可以用来读取文件的内容：

```

file(READ fileName outVar
  [OFFSET offset] [LIMIT byteCount] [HEX]
)

```

没有任何可选关键字的情况下，该命令将读取 `fileName` 的所有内容，并将它们作为单个字符串存储在 `outVar` 中。偏移量选项可用于从指定的偏移量中读取，该偏移量以字节为单位计算。还可以使用 `LIMIT` 选项限制读取的最大字节数。如果给出了十六进制选项，则内容将转换为十六进制表示，这对于包含二进制数据文件很有用。

如果希望逐行分解文件内容，则字符串的形式可能更方便。这个方式不是将整个文件的内容存储为单个字符串，而是将其存储为一个列表，每一行都是一个列表项。以下简化的显示了常用的选项：

```
file(STRINGS fileName outVar
    [LENGTH_MAXIMUM maxBytesPerLine]
    [LENGTH_MINIMUM minBytesPerLine]
    [LIMIT_INPUT maxReadBytes]
    [LIMIT_OUTPUT maxStoredBytes]
    [LIMIT_COUNT maxStoredLines]
    [REGEX regex]
)
```

上面没有显示的选项与编码、特殊文件类型的转换或换行字符的处理，这些在大多数情况下都不需要。详细信息请参考CMake文档。

`LENGTH_MAXIMUM` 和 `LENGTH_MINIMUM` 选项可用于排除长度大于或小于特定字节数的字符串。使用 `LIMIT_INPUT` 限制读取的总字节数，而使用 `LIMIT_OUTPUT` 限制存储的总字节数。然而，`LIMIT_COUNT` 选项是限制行存储的总数，而不是字节数。

`REGEX` 选项是从文件中提取感兴趣的特定行的特别有用的方法。例如，下面的列表包含 `myStory.txt` 中包含 `PKG_VERSION` 或 `MODULE_VERSION` 的行。

```
file(STRINGS myStory.txt versionLines
    REGEX "(PKG|MODULE)_VERSION"
)
```

还可以与 `LIMIT_COUNT` 结合使用，只获取第一个匹配项。下面的示例展示了如何组合 `file()` 和 `string()`，来提取第一行中与正则表达式匹配的部分。

```
set(regex "^\*FOO_VERSION *= *([^\ ]+) *$")
file(STRINGS config.txt fooVersion
    REGEX "${regex}"
)
string(REGEX REPLACE "${regex}" "\\\1" fooVersion "${fooVersion}")
```

如果 `config.txt` 包含这样的一行：

```
FOO_VERSION = 2.3.5
```

那么 `fooVersion` 中存储的值将是2.3.5。

## 18.4. 配置文件系统

除了读写文件之外，CMake还支持其他常见的文件系统操作。

```
file(RENAME source destination)
file(REMOVE files...)
file(REMOVE_RECURSE filesOrDirs...)
file(MAKE_DIRECTORY dirs...)
```

`RENAME` 重命名文件或目录，如果目标已经存在，则以静默方式替换。源和目标必须是相同类型，即两个文件或两个目录。不允许指定文件和目录为目的。若要将文件移动到目录中，必须将文件名指定为目标文件的一部分。此外，目标路径必须存在，`RENAME` 不会创建中间目录。

`REMOVE` 可用于删除文件。如果列出的任何文件不存在，`file()` 命令不会报告错误，试图用 `REMOVE` 删除目录将不起作用。要删除目录及其所有内容，请使用 `REMOVE_RECURSE` 形式。

`MAKE_DIRECTORY` 表单将确保列出的目录存在，根据需要创建中间路径，如果目录已经存在，不会报错。

**CMake**的命令模式也支持非常类似的能力，可以在构建时使用，而不是配置时：

```
cmake -E rename source destination
cmake -E remove [-f] files...
cmake -E remove_directory dir
cmake -E make_directory dirs...
```

这些命令的行为基本上与 `file()` 的对应命令相当，只有细微的变化。`remove_directory` 命令只能用于单个目录，而 `file(REMOVE_RECURSE...)` 可以删除多个项，并且文件和目录都可以列出。`remove` 命令接受可选的 `-f` 标志，当试图删除不存在的文件时，它会改变行为。如果没有 `-f`，将返回一个非零的退出码，而使用 `-f`，将返回一个零的退出码。这是模拟了 Unix `rm -f` 的行为。

**CMake**还支持用递归或非递归的方式列出一个或多个目录的内容：

```
file(GLOB outVar
  [LIST_DIRECTORIES true|false]
  [RELATIVE path]
  [CONFIGURE_DEPENDS] # Requires CMake 3.12 or later
  expressions...
)
```

```
file(GLOB_RECURSE outVar
  [LIST_DIRECTORIES true|false]
  [RELATIVE path]
  [FOLLOW_SYMLINKS]
  [CONFIGURE_DEPENDS] # Requires CMake 3.12 or later
  expressions...
)
```

这些命令会找到名称与所提供的表达式(可以看作是简化的正则表达式)匹配的所有文件，通过添加字符集，可以将它们视为普通的通配符。`GLOB_RECURSE` 还可以包括路径组件。一些例子可以说明基本用法：

<code>*.txt</code>	名称以 <code>.txt</code> 结尾的所有文件。
<code>foo?.txt</code>	比如 <code>foo2.txt</code> , <code>fooB.txt</code> 等
<code>bar[0-9].txt</code>	匹配所有 <code>barX.txt</code> 文件，其中 <code>X</code> 为单个数字。
<code>/images/*.png</code>	对于 <code>GLOB_RECURSE</code> ，只匹配扩展名为 <code>.png</code> ，并且位于名为 <code>images</code> 的子目录中的文件。这是在结构良好的目录层次结构中查找文件的良好方法。

对于 `GLOB` 与表达式匹配的文件和目录都存储在 `outVar` 中，对于 `GLOB_RECURSE` 默认情况下不包含目录名，但是可以使用 `LIST_DIRECTORIES` 选项进行控制。`GLOB_RECURSE` 指向目录的符号链接通常作为 `outVar` 中的内容，但 `FOLLOW_SYMLINKS` 选项指示**CMake**降至目录中，而不是将其列出。

默认情况下，返回的文件名集将是绝对路径，与使用的表达式无关。`RELATIVE` 选项可更改此行为，使所报告的路径相对于指定目录。

```
set(base /usr/share)

file(GLOB_RECURSE images
  RELATIVE ${base}
  ${base}/*/*.png
)
```

上面会找到 `/usr/share` 下面的所有图像，并包括这些图像的路径，除了去掉了 `/usr/share` 部分。注意表达式中的 `/*` 可以匹配下面的任何目录。

开发人员应该知道，`file(GLOB...)` 命令的速度不如 Unix 的 `find` 命令快。因此，如果使用运行时搜索文件系统中包含许多文件的部分，运行时长非常重要。

`file(GLOB)` 和 `file(GLOB_RECURSE)` 会经常误用。它们不应该用于收集用作源文件、头文件或充当构建输入的文件集。避免这种情况的原因之一是，如果文件被添加或删除，CMake 不会自动重新运行，因此构建不会意识到更改。如果开发人员使用了版本控制系统，并在分支之间进行切换，这就成问题了。在这些地方，文件集可能会发生变化，但不会导致 CMake 重新运行。`CMake 3.12` 中添加的 `CONFIGURE_DEPENDS` 选项试图解决这个缺陷，但会带来性能损失，并且只对某些生成器有效，所以最好不要使用这个选项。

不幸的是，经常可以看到教程和示例使用 `file(GLOB)` 和 `file(GLOB_RECURSE)` 收集要传递给 `add_executable()` 和 `add_library()` 等命令的源文件集。由于上述原因，CMake 文档明确不鼓励这样做。对于跨多个目录有许多文件的项目，有更好的方法来收集源文件集。`28.5.1` 节提出了一些备选策略，这些策略不仅避免了这些问题，而且鼓励采用更模块化和自包含的目录结构。

## 18.5. 下载和上传

`file()` 有许多其他使用方式，它们执行不同的任务。提供了从 `url` 下载文件和将文件上传到 `url` 的功能。

```
file(DOWNLOAD url fileName [options...])
file(UPLOAD fileName url [options...])
```

`DOWNLOAD` 从指定的 `url` 下载文件并将其保存为 `fileName`。如果给出了相对文件名，将解释为相对于当前二进制目录。`UPLOAD` 执行反向操作，将命名的文件上传到指定的 `url`。对于上传，相对路径解释为相对于当前源目录。`DOWNLOAD` 和 `UPLOAD` 都有一些常见选项：

***LOG outVar***

将操作的日志输出保存到指定变量。当下载或上传失败时，助于诊断问题。

***SHOW\_PROGRESS***

该选项出现时，会将进度信息作为状态消息记入日志。这可能会产生一个相当“嘈杂”的 CMake 配置阶段，所以可以使用这个选项作为测试失败的连接的调试方法。

***TIMEOUT seconds***

如果经过的时间超过设置的秒数，则中止操作。

***INACTIVITY\_TIMEOUT seconds***

更具体的超时设置。有些网络连接的质量可能很差，或者可能只是很慢。允许操作继续下去，只要它有速度，但是如果它停止的时间超过了可接受的时限，操作就会失败。`INACTIVITY_TIMEOUT` 选项提供了这种功能，而 `TIMEOUT` 只允许限制总时间。

`DOWNLOAD` 还支持更多的选择：

***EXPECTED\_HASH ALGO=value***

指定正在下载的文件的校验和，以便 CMake 可以验证内容。`ALGO` 可以是支持的任意一种哈希算法，最常用的是 `MD5` 和 `SHA1`。一些较老的项目可能使用 `EXPECTED_MD5` 作为 `EXPECTED_HASH MD5=...` 的替代，但是新的项目应该使用 `EXPECTED_HASH`。

### *TLS\_VERIFY value*

此选项接受一个布尔值，该值指示从 `https://url` 下载时是否执行服务器证书验证。如果没有提供此选项，则 **CMake** 将查找名为 `CMAKE_TLS_VERIFY` 的变量。如果选项和变量都没有定义，则默认不验证服务器证书。

### *TLS\_CAINFO fileName*

可以使用此选项指定自定义证书文件。它只影响 `https://url`。

对于 **CMake 3.7** 或更高版本，以下选项也可用于 `DOWNLOAD` 和 `UPLOAD`：

### *USERPWD username:password*

提供操作的身份验证细节。注意，硬编码密码是一个安全问题，应该避免使用。如果使用此选项提供密码，则内容应该来自项目外部，例如：配置时从用户本地机器读取的受保护的文件。

### *HTTPHEADER header*

操作的 **HTTP** 头部，可以根据需要重复多次，以提供多个头值。下面的部分示例演示了该选项的一个案例：

```
file(DOWNLOAD "https://somebucket.s3.amazonaws.com/myfile.tar.gz"
myfile.tar.gz
EXPECTED_HASH SHA1=${myfileHash}
HTTPHEADER "Host: somebucket.s3.amazonaws.com"
HTTPHEADER "Date: ${timestamp}"
HTTPHEADER "Content-Type: application/x-compressed-tar"
HTTPHEADER "Authorization: AWS ${s3key}:${signature}"
)
```

基于 `file()` 的下载和上传命令通常用作安装步骤、打包或测试报告的一部分，但偶尔也用于其他目的。例如，在配置时下载引导文件，或者将不能或不应该作为项目源的一部分存储的文件放入构建中(例如，只有特定开发人员才能访问的敏感文件，非常大的文件等等)。

## 18.6. 总结

本章介绍了一系列与文件处理相关的**CMake**功能。可以使用各种方法有效地执行一系列的任务。建立良好的习惯和模式，并在整个项目中始终如一地使用它们，将有助于开发新手接触到更好的实践。

`configure_file()` 是开发新手经常忽略的一个命令，它提供处理文件的关键方法，该文件的内容可以在配置时确定并进行调整，甚至只为了进行简单的文件复制。常见的命名约定是源和目标的文件名部分是相同的，除了源有额外的 `.in` 作为后缀。一些**IDE**环境理解这种约定，并且仍然会根据文件的扩展名(不带 `.in` 后缀)在源文件上提供适当的语法高亮显示。后缀不仅作为清晰的提醒，在文件需要转换/复制之前使用，如果 **CMake** 或编译器寻找文件在多个目录，也可以防止目标意外的使用。当目标文件是 C/C++ 头文件时，并且当前源目录和二进制目录都在头文件搜索路径上时，这就很重要了。

为复制文件选择最合适命令并不总是很明朗。在 `configure_file()`、`file(COPY)` 和 `file(INSTALL)` 之间进行选择时，参考以下内容作为指导：

- 如果需要用**CMake**变量替换文件内容，`configure_file()` 是合适的选择。
- 如果只需要复制一个文件，但文件名会发生变化，`configure_file()` 的语法比 `file(COPY...)` 稍短一些，两者都适合。
- 如果复制多个文件或整个目录结构，使用 `file(COPY)` 或 `file(INSTALL)` 更好。
- 如果需要控制文件或目录权限作为复制的一部分，必须使用 `file(COPY)` 或 `file(INSTALL)`。
- `file(INSTALL)` 通常作为安装脚本的一部分使用。其他情况下，最好使用 `file(COPY)`。

CMake 3.10之前，`file(GENERATE...)` 命令对相对路径的处理是不同的。项目不应该依赖于开发人员能意识到这种不同的行为，而是应该始终使用绝对路径指定输入和输出文件，以避免错误或文件在意外的位置生成。

使用`file(DOWNLOAD...)` 或`file(UPLOAD...)` 命令下载或上传文件时，要仔细考虑安全和效率的问题。尽量避免在项目源的版本控制系统中存储的任何文件中嵌入任何类型的认证细节(用户名、密码、私钥等)。这些细节应该来自项目外部，比如：通过环境变量(仍然不安全)、用户文件系统中找到的具有适当权限限制访问的文件或某种类型的密钥。下载时使用`EXPECTED_HASH` 选项，重用以前运行时下载的内容，避免可能耗时的远程操作。如果下载的文件的哈希值不能提前知道，那么强烈建议使用`TLS_VERIFY` 选项来确保内容的完整性。还可以考虑指定一个`TIMEOUT`、`INACTIVITY_TIMEOUT` 或两者同时指定，以防止在网络连接不佳或不可靠的情况下，配置时的无限阻塞。

# 第19章：指定版本

版本控制是一件应该重视的事，版本号的重要性经常被低估，导致用户无法满足预期，或者对版本变化感到困惑。市场营销和版本控制策略如何影响构建、打包等的技术之间，也存在着微妙的关系。早期思考和建立这些项目的定位，好于交付之后再思考。本章探讨了实现有效版本控制策略的方法，利用CMake特性来提供健壮、高效的过程。

## 19.1. 项目版本

通常需要在顶级CMakeLists.txt文件定义，以便构建的各个部分可以引用。源代码可能想要嵌入项目版本，以便显示给用户或记录在日志文件中，打包步骤可能需要它来定义发布版本的细节等等。可以简单地在CMakeLists.txt文件的开头附近设置一个变量，记录需要的版本号：

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION 2.4.7)
```

如果需要提取单独的组件，可能需要定义复杂的变量集：

```
cmake_minimum_required(VERSION 3.0)
project(FooBar)
set(FooBar_VERSION_MAJOR 2)
set(FooBar_VERSION_MINOR 4)
set(FooBar_VERSION_PATCH 7)
set(FooBar_VERSION
    ${FooBar_VERSION_MAJOR}.${FooBar_VERSION_MINOR}.${FooBar_VERSION_PATCH}
)
```

不同的项目可能对变量的命名使用不同的规定。版本号的结构也因项目而异，因此缺乏一致性使得将许多项目合并成一个更大的集合或超级构建(28.1节中讨论)，从而更加难以理解。CMake 3.0引入了新的功能，更容易地指定版本细节，并为项目版本编号带来了一致性。`VERSION`关键字添加到`project()`命令中，格式为`major.minor.patch.tweak`。根据该信息，可以自动填充一组变量，使完整的版本字符串以及每个版本组件对其余部分单独可用。如果提供的版本字符串省略了某些部分(例如，省略微调部分)，则相应的变量将保留为空。下表显示了当`VERSION`关键字与`project()`一起使用时，如何自动填充的版本变量：

PROJECT_VERSION	projectName_VERSION
PROJECT_VERSION_MAJOR	projectName_VERSION_MAJOR
PROJECT_VERSION_MINOR	projectName_VERSION_MINOR
PROJECT_VERSION_PATCH	projectName_VERSION_PATCH
PROJECT_VERSION_TWEAK	projectName_VERSION_TWEAK

这两组变量的用途略有不同，项目的`projectName_...`变量可以用于从当前目录范围或以下的任何地方。像`project(FooBar VERSION 2.7.3)`这样的调用会产生名为`FooBar_VERSION`、`FooBar_VERSION_MAJOR`等变量。不能以相同的参数`project()`调用两次，所以这些特定于项目的变量不会在其他`project()`调用时覆盖。另一方面，`PROJECT_...`变量在每次调用`project()`时都会更新，可以用来提供当前或获取`project()`最近一次调用的版本信息。CMake 3.12在顶级CMakeLists.txt文件调用`project()`时，还提供了一组相似的变量版本。这些变量是：

CMAKE_PROJECT_VERSION
CMAKE_PROJECT_VERSION_MAJOR
CMAKE_PROJECT_VERSION_MINOR
CMAKE_PROJECT_VERSION_PATCH
CMAKE_PROJECT_VERSION_TWEAK

同样的模式也用于为项目名称、描述和主页 url 提供变量，后两者分别在CMake版本3.9和3.12中添加。作为一个指南，PROJECT\_... 变量对于泛型代码(尤其是模块)很有用，可以用来为打包或文档定义默认值。CMAKE\_PROJECT\_... 变量有时也用作默认值，所以它们通常使用于项目顶部，可靠性没那么高。projectName\_... 变量是最健壮的，因为在提供项目信息时总是无误的。

当支持CMake 3.0之前版本时，需要定义自己的版本变量，这与CMake 3.0及以后版本自动定义的变量有冲突，可能导致CMP0048策略警告。下面是导致这种警告的代码示例：

```
cmake_minimum_required(VERSION 2.8.12)
set(FooBar_VERSION 2.4.7)
project(FooBar)
```

上面的示例中，显式设置了 FooBar\_VERSION 变量，但该变量名与 project() 自动定义的变量冲突。生成策略警告旨在鼓励项目使用不同的变量名，或者将CMake的最小版本更新到3.0，并在 project() 中设置版本。

## 19.2. 让源码了解版本信息

CMakeLists.txt文件中定义了版本信息后，常见的需求是用于编译的源码。可以使用许多不同的方法，每一种都有优点和缺点。CMake最常用的技术是在项目顶层添加编译器定义：

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)
add_definitions(-DFOOBAR_VERSION="\${FooBar_VERSION}\")
```

这使得版本信息作为一个原始字符串，可以这样使用：

```
void printVersion()
{
    std::cout << FOOBAR_VERSION << std::endl;
}
```

虽然这种方法简单，但将定义添加到项目中每个单独文件的编译中会有一些缺点。除了对每个文件的执行命令行编译，这也意味着版本号变化，会让整个项目重新构建。这可能是一个次要的问题，但是定期在源码控制系统的不同分支之间切换的开发人员，肯定会对不必要的重新编译感到恼火。稍好一点的方法是，使用源属性为需要 FOOBAR\_VERSION 符号的文件进行定义。例如：

```

cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

add_executable(foobar main.cpp src1.cpp src2.cpp ...)

get_source_file_property(defs src1.cpp COMPILE_DEFINITIONS)
list(APPEND defs "FOOBAR_VERSION=\"${FooBar_VERSION}\"")
set_source_files_properties(src1.cpp PROPERTIES
    COMPILE_DEFINITIONS ${defs}
)

```

避免了向每个文件添加编译器定义，只是将其添加到需要它的文件中。然而，当设置单独的源属性时，会对构建依赖关系产生负面影响，这将导致重新构建的文件比需要的多。因此，这种方法看起来可能是一种改进，但通常不可行。

与命令行上传递版本信息不同，另一种常见方法是使用 `configure_file()` 编写提供版本信息的头文件。例如：

*foobar\_version.h.in*

```

#include <string>
inline std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}
inline unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}
inline unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}
inline unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}
inline unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}

```

*main.cpp*

```

#include "foobar_version.h"
#include <iostream>
int main(int argc, char* argv[])
{
    std::cout << "VERSION = " << getFooBarVersion() << "\n"
    << "MAJOR = " << getFooBarVersionMajor() << "\n"
    << "MINOR = " << getFooBarVersionMinor() << "\n"
    << "PATCH = " << getFooBarVersionPatch() << "\n"
    << "TWEAK = " << getFooBarVersionTweak()
    << std::endl;
    return 0;
}

```

*CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.h.in foobar_version.h @ONLY)
add_executable(foobar main.cpp)
target_include_directories(foobar PRIVATE "${CMAKE_CURRENT_BINARY_DIR}")
```

`foobar_version.h.in` 中的`+0`，对于`minor`、`patch`和`tweak`部件来说是必要的，以便在忽略版本组件的情况下允许对应的变量为空。

通过头文件提供版本信息是对技术的改进。版本信息不包含在任何源文件编译的命令行中，只有 `#include "foobar_version.h"` 头文件的源文件，在版本信息更改时才需要重新编译。提供不同的版本组件，而版本字符串也不会对命令行产生影响。然而，如果在许多不同的源文件中都需要版本号，这仍然可能导致需要更多源文件进行重新编译。通过将实现从头文件移到 `.cpp` 文件中，并将其编译为库，可以做进一步的改进。

#### `foobar_version.h`

```
#include <string>
std::string getFooBarVersion();
unsigned getFooBarVersionMajor();
unsigned getFooBarVersionMinor();
unsigned getFooBarVersionPatch();
unsigned getFooBarVersionTweak();
```

#### `foobar_version.cpp.in`

```
#include "foobar_version.h"
std::string getFooBarVersion()
{
    return "@FooBar_VERSION@";
}
unsigned getFooBarVersionMajor()
{
    return @FooBar_VERSION_MAJOR@;
}
unsigned getFooBarVersionMinor()
{
    return @FooBar_VERSION_MINOR@ +0;
}
unsigned getFooBarVersionPatch()
{
    return @FooBar_VERSION_PATCH@ +0;
}
unsigned getFooBarVersionTweak()
{
    return @FooBar_VERSION_TWEAK@ +0;
}
```

#### `CMakeLists.txt`

```

cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
add_library(foobar_version STATIC ${CMAKE_CURRENT_BINARY_DIR}/foobar_version.cpp)

add_executable(foobar main.cpp)
target_link_libraries(foobar PRIVATE foobar_version)

add_library(fooToolkit mylib.cpp)
target_link_libraries(fooToolkit PRIVATE foobar_version)

```

这种安排没有前面方法的缺点。当版本信息发生变化时，只需要重新编译一个源文件(生成的 `foobar_version.cpp` 文件)，而 `foobar` 和 `fooToolkit` 目标需要重新链接。`foobar_version.h` 永远不会改变，所以当版本信息改变时，任何依赖于它的文件都不会重编。也没有向任何源文件的编译命令行添加任何选项，因此不会因更改版本信息而触发重新编译。

项目提供库和头文件作为发布包的一部分，上述方法也是健壮的。头文件不包含版本信息，而库中包含。因此，使用库的代码可以调用版本函数，并且确认它们接收到的细节是构建库时使用的。这在复杂的环境中很有帮助，这种环境中，可能会安装项目的多个版本，而不一定按照项目的预期结构进行构造。

这种方法的变体是 `foobar_version` 为对象库，而不是一个静态库。最终的结果或多或少是相同的，但是从中获得的好处并不多。使其成为动态库会失去一些健壮性优势，并且会增加复杂性，收效甚微，因此通常建议将这些版本库设置为静态。

### 19.3. 源代码控制提交

对于项目来说，记录与源代码控制系统相关的信息很正常，包括构建时版本或提交哈希、当前分支的名称或标记等等。上面介绍的方式是通过专用的 `.cpp` 文件提供版本详细信息，很适合添加函数来返回这些信息。例如，当前 `git` 的哈希值可以相对容易的提供：

`foobar_version.cpp.in`

```

std::string getFooBarGitHash()
{
    return "@FooBar_GIT_HASH@";
}
// Other functions as before...

```

`CMakeLists.txt`

```

cmake_minimum_required(VERSION 3.0)
project(FooBar VERSION 2.4.7)

# The find_package() command is covered later in the Finding Things chapter.
# Here, it provides the GIT_EXECUTABLE variable after searching for the
# git binary in some standard/well-known locations for the current platform.
find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-parse HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE FooBar_GIT_HASH
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get git hash: ${result}")
endif()

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc...

```

有趣一点的示例，查看指定文件自创建以来，产生了多少次提交。考虑将项目的版本信息会嵌入到单独的文件中，这个文件中只有项目版本号。这样就可以合理地假设文件只在版本号改变时才会改变。因此，查看自当前分支上的文件更改提交数量，通常是统计自上一个版本更新以来提交数量的好方法。

下面的示例将项目版本移到名为 `projectVersionDetails.cmake` 的单独文件中，并通过生成的 `foobar_version.cpp` 文件的新函数提供提交数量。它演示了一种模式，这种模式适用于通过顶级 `project()` 设置版本信息，但如果父项目合并到更大的项目层次结构中，这个操作不会干扰父项目。

#### `foobar_version.cpp.in`

```

unsigned getFooBarCommitsSinceVersionChange()
{
    return @FooBar_COMMITS_SINCE_VERSION_CHANGE@;
}
// Other functions as before...

```

#### `projectVersionDetails.cmake`

```

# This file should contain nothing but the following line
# setting the project version. The variable name must not
# clash with the FooBar_VERSION* variables automatically
# defined by the project() command.
set(FooBar_VER 2.4.7)

```

#### `CMakeLists.txt`

```

cmake_minimum_required(VERSION 3.0)
include(projectVersionDetails.cmake)
project(FooBar VERSION ${FooBar_VER})

find_package(Git REQUIRED)
execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list -1 HEAD projectVersionDetails.cmake
    RESULT_VARIABLE result
    OUTPUT_VARIABLE lastChangeHash
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get hash of last change: ${result}")
endif()

execute_process(
    COMMAND ${GIT_EXECUTABLE} rev-list ${lastChangeHash}..HEAD
    RESULT_VARIABLE result
    OUTPUT_VARIABLE hashList
    OUTPUT_STRIP_TRAILING_WHITESPACE
)
if(result)
    message(FATAL_ERROR "Failed to get list of git hashes: ${result}")
endif()
string(REGEX REPLACE "[\n\r]+;" ";" hashList "${hashList}")
list(LENGTH hashList FooBar_COMMITS_SINCE_VERSION_CHANGE)

configure_file(foobar_version.cpp.in foobar_version.cpp @ONLY)
# Targets, etc...

```

上述方法输出文件最后修改版本的git哈希，然后使用 `git rev-list` 获取整个存储库的提交列表。提交最初是作为哈希字符串找到的，然后通过使用列表分隔符(`;`)替换换行字符，转换为CMake列表。然后 `list()` 简单地计算列表中有多少项，给出提交的数量。一种更简单的方法是使用 `git rev-list --count` 直接获取数字，但是旧版本的git不支持 `--count` 选项，因此如果需要支持旧版本的git，最好使用上述方法。

一些项目使用git描述来提供各种细节，包括分支名称、最近的标签等等，但请注意，标签和分支细节可以在不改变提交的情况下改变。如果分支或标记移动或重命名，构建不可重复。如果版本信息只依赖于文件提交哈希，则不会产生这样的缺陷。构建确认提交没有错误之后，这也为项目提供了创建、重命名或删除标签的自由(想象一下在持续集成构建、测试等确认没有问题之后，应该要提交发布标签)。

像Subversion这样的源代码控制系统还面临其他挑战。一方面，Subversion为整个存储库维护一个全局修订号，因此不需要首先获取提交哈希值，然后对其进行计数。Subversion还有一个复杂之处，它允许混合不同文件的不同版本。因此，如果开发人员签出文件的不同版本，而不去管项目版本文件，上面为git列出的方法就会失效。这不是自动化持续集成预期的方式，但对于在自己的本地工作的开发人员则有可能，这取决于他们偏好的工作方式。

对上述技术的另一个考虑是，将什么强制更新到生成的 `.cpp` 文件版本。如果项目版本文件改变了，CMake可以确保配置步骤重新运行，因为它是以 `include()` 的形势进入主 `CMakeLists.txt` 文件的。如果对其他文件进行了提交，CMake将不会了解。在版本控制系统中实现钩子(比如git的 `post-commit` 钩子)来强制CMake重新运行是可能的，但这可能更让开发人员烦恼。最终，会在便利性和健壮性之间妥协。也就是说，源码控制信息的准确性可能只对发布至关重要，而且应该足够简单，以确保发布过程可以显式地调用CMake。

## 19.4. 总结

项目不需要遵循任何特定的版本控制系统，但是需要遵循 `major.minor.patch.tweak` 格式，开发新手可以更容易地理解项目使用的版本。后面的章节中，版本格式在打包发布时更加重要，但是许多项目在运行时报告他们自己的版本号，版本格式也会影响构建。

组成版本格式的每个数字的含义由项目决定，但有些约定是用户期望的。例如，`major`通常意味着一个重要的发布，通常涉及到不向后兼容的变更或者代表项目方向上的变更。`minor`发生了变化，用户会倾向于将其视为增量版本，可能是在不破坏现有行为的情况下添加新特性。修改`patch`时，用户不会认为这是特别重要的更改，希望是相对较小的更改，比如修复一些`bug`，但不引入新功能。`tweak`会经常忽略，除了比`patch`更不重要之外，往往没有常见的解释。请注意，这些只是一般性的观察结果，项目可以给予版本号完全不同的含义。为了简单，一个项目可能只使用一个单独的数字，有效地指定每个版本作为新的`major`。虽然这很容易实现，但无法给用户更多的指导，并且需要高质量的发布说明来管理每个版本之间的区别，从而满足用户的期望。

`project()` 的 `VERSION` 关键字是CMake在 `major.minor.patch.tweak` 中提供的例子。项目提供版本字符串，`project()` 自动定义一组变量，使版本号的各个部分可用。一些CMake模块可以使用这些变量作为某些元数据的默认值，所以通常建议使用 `project()` 中的 `VERSION` 关键字来设置项目版本。这个关键字是CMake 3.0 中添加，如果要支持旧的CMake版本，这个功能仍然需要考虑。项目不应该定义与自动定义的名称冲突的变量，否则之后的CMake版本会发出警告，避免显式地设置名称为 `xxx_VERSION` 或 `xxx_VERSION_yyy` 的变量。

定义版本号时，考虑在自己的专用文件中这样做，然后CMake通过 `include()` 将其拉入。这使得项目可以利用版本号，这些版本号与项目的源代码控制系统所看到的文件中的变化保持一致。为了减少版本更改时不必要的重新编译，可以生成一个 `.c` 或 `.cpp` 文件，其中包含返回版本细节的函数，而不是将这些细节嵌入生成在头文件中，或者作为编译器定义在命令行上传递。还要确保这些函数提供的名称包含特定于项目的内容，或者将它们放置在特定于项目的命名空间中。这允许在许多项目之间可以使用相同的模式，这些项目也可合并到单个构建中，而不会导致名称冲突。

项目的早期建立版本控制策略和实现模式，有助于开发人员清楚地了解如何更新版本细节，并鼓励在第一次交付的压力到来前就考虑发布的过程。还允许较低效率的方法(需要尽早淘汰)，以便在版本号变更，以及相似构建时间变得更重要前，最大化构建的效率。

# 第20章：库

与编写普通应用程序相比，创建和维护库通常更加复杂，尤其是动态库。所有关于代码正确性和可维护性的常见问题仍然存在，但动态库会带来与API一致性、版本之间的兼容性、符号可见性等相关的考虑。此外，每个平台通常都有自己的独特的特性和需求，这使得跨平台库的开发成为一项具有挑战的任务。

大多数情况下，所有主要平台都支持一组核心功能，只是定义或使用的方式不同而已。**CMake**抽象出这些差异的特性，以便开发人员可以更专注于功能，将实现细节留给构建系统。

## 20.1. 构建的基础知识

定义库的基本命令已经在前面的章节中介绍过了，形式如下：

```
add_library(  
    targetName [STATIC | SHARED | MODULE | OBJECT]  
    [EXCLUDE_FROM_ALL]  
    source1 [source2 ...])
```

如果提供 `SHARED` 或 `MODULE` 关键字，将生成动态库。另外，如果没有给出 `STATIC`、`SHARED`、`MODULE` 或 `OBJECT` 关键字， `BUILD_SHARED_LIBS` 变量在调用 `add_library()` 时的值为`true`，将生成动态库。

`SHARED` 和 `MODULE` 之间的区别在于，动态库用于链接其他目标，而 `MODULE` 库不是。`MODULE` 库通常用于插件或其他可选库，可以在运行时加载。此类库的加载，通常依赖于应用程序配置或某些系统特性。其他可执行程序和库通常不链接到 `MODULE` 库。

大多数基于 Unix 的平台上，`STATIC` 或 `SHARED` 库的文件名在默认情况下使用 `lib` 前缀，而模块可能不使用。`Apple` 平台还支持框架和可加载包，允许附加文件以良好的目录结构与库绑定。

`Windows` 平台上，不管库的类型是什么，库名称都没有前缀。`STATIC` 库目标产生单个的 `.lib`，而 `SHARED` 库目标会产生两个独立的文件，一个用于运行时 (`.dll` 或动态链接库)，另一个用于构建时链接 (即 `.lib` 导入库)。开发人员有时会混淆导入库和静态库，因为使用了相同的文件后缀，但 **CMake** 通常可以正确地处理它们，而不需要任何特殊的操作。

当在 `Windows` 下使用 `GNU` 工具 (例如 `MinGW` 或 `MSYS` 项目生成器)，**CMake** 有能力将 `GNU` 导入库 (`.dll` 和 `.a`) 以相同的格式导出成 `Visual Studio` 生成库 (`lib`)。如果发布用 `GNU` 工具构建的动态库，使其能够链接到 `Visual Studio` 构建的二进制文件，那么这很有用。请注意，要实现此转换，必须安装 `Visual Studio`。通过将动态库的 `GNUToMS` 目标属性设置为 `true` 来启用转换。这个目标属性在调用 `add_library()` 时由 `CMAKE_GNUtoMS` 值初始化。

## 20.2. 连接静态库

**CMake** 可以处理链接静态库的特殊情况。如果库 A 为静态库目标 B 的私有依赖项，就链接而言 (并且仅用于链接)，A 将视为公共依赖项。因为私有的 A 库仍然需要添加到链接到 B 的链接器命令行中，以便在链接时找到来自 A 的符号。如果 B 库是动态库，则不需要在链接器命令行中列出它所依赖的私有库 A。这一切都是由 **CMake** 处理，因此开发人员除了指定 `PUBLIC`、`PRIVATE` 和 `INTERFACE` 依赖 `target_link_libraries()` 之外的细节，无需关心其他。

项目中静态库不包含循环依赖，也就是相互依赖的两个或两个以上的库。然而，一些场景会出现这样的情  
况，只要指定了相关的链接关系(通过 `target_link_libraries()` )，**CMake**就会识别并处理循环依赖关系。一个  
修改过的**CMake**文档中的例子演示了这种行为：

```
add_library(A STATIC a.cpp)
add_library(B STATIC b.cpp)
target_link_libraries(A PUBLIC B)
target_link_libraries(B PUBLIC A)
add_executable(main main.cpp)
target_link_libraries(main A)
```

上面例子中，为**main**目标连接命令将包含**A B A B**。这种重复是**CMake**自动提供，不需要开发人员操作，在某  
些特殊的情况下，可能需要多个重复。虽然**CMake**为此提供了 `LINK_INTERFACE_MULTIPLICITY` 目标属性，但这种  
情况意味着重新构造项目。对象库可能是解决这种深度依赖关系的有用工具，因为实际上就是一组源，而  
不是实际的库。链接器命令行上对象文件的顺序通常不重要，而库的顺序却很重要。

## 20.3. 动态库的版本

如果**CMake**项目不希望库在项目之外使用，那么创建的任何动态库都不需要版本信息。整个项目倾向于在部  
署时更新，因此在确保版本之间的兼容性方面没有什么问题。但如果项目提供库，而其他软件可以链接到  
库，库的版本控制就变得非常重要。库的版本信息增强了库的健壮性，允许其他软件指定它们希望链接到的  
接口，并在运行时提供给它们。

大多数平台都提供了指定动态库版本号的功能，但是实现的方式有很大不同。平台通常有能力编码二进制版  
本信息到动态库中，这些信息有时用来决定二进制文件是否可以使用另一个可执行文件或动态库的链接。一  
些平台也在其名称中，使用不同级别版本号的文件集和符号链接。例如，在**Linux**上动态库的一组文件和符号  
链接可能是这样的：

```
libmystuff.so.2.4.3
libmystuff.so.2 --> libmystuff.so.2.4.3
libmystuff.so --> libmystuff.so.2
```

**CMake**负责处理与共享库版本处理相关的平台差异。将目标链接到动态库时，在决定链接哪个文件或符号链  
接时，遵循平台约定。构建动态库时，如果提供了版本信息，**CMake**将自动创建完整的文件集和符号链接。

动态库的版本信息由 `VERSION` 和 `SOVERSION` 目标属性定义。在平台上，这些属性的解释不同，但按照语义版本  
控制原则，这些差异可以无缝处理。版本化假设在 `major.minor.patch` 中每个版本组件都是整数。`VERSION` 属性  
将设置为完整的 `major.minor.patch` 格式，而 `SOVERSION` 仅设置为 `major`。随着项目的发展和发布，版本化意味  
着版本信息应该有如下修改：

- 当对不兼容的API进行更改时，增加版本的**major**部分，并将**minor**和**patch**部分重置为0。这意味着  
`SOVERSION` 属性将在每次出现API破坏时(且仅在出现API破坏时)更改。
- 当以向后兼容的方式添加功能时，将**minor**部分和**patch**部分重置为0。**major**部分保持不变。
- 当以向后兼容的bug进行修复时，增加**patch**值并保持**major**部分和**minor**部分不变。

如果根据这些原则修改了动态库的版本信息，在所有平台上运行时的API不兼容问题将最小化。考虑下面的例  
子，在**Linux**操作系统上产生的符号链接：

```
add_library(mystuff SHARED source1.cpp ...)
set_target_properties(mystuff PROPERTIES
  VERSION 2.4.3
  SOVERSION 2
)
```

**Apple**平台上，`otool -L`可以用来打印动态库中的版本信息。上面示例生成的动态库的输出将输出版本信息，说明其兼容版本为2.0.0，当前版本为2.4.3。

任何链接到`mystuff`库的都将命名为`libmystuff.2.dylib`，作为运行时要查找的库。**Linux**平台在动态库的符号链接中显示了类似的结构，通常做法是只使用库动态库名的**major**部分。

**Windows**上，**CMake**的行为是从`VERSION`属性中提取**major.minor**，并将其编码到**DLL**中作为**DLL**映射版本。**Windows**没有动态库名称的概念，因此不使用`SOVERSION`属性。然而，遵循版本化的原则是，至少确保使用**DLL**版本来确定，库与链接的二进制文件间的兼容性。

需要注意的是，版本化并不是平台严格要求的。它提供了良好的规范，为动态库及其使用对象间的关系管理带来了确定性。从而反映了库版本，在大多数基于**Unix**的平台上是如何解释的，**CMake**充分利用户`VERSION`和`SOVERSION`目标属性，来提供遵循本地平台约定的动态库。

项目应该知道，如果只设置了`VERSION`和`SOVERSION`目标属性中的一个，丢失的属性将视为与提供的那个属性具有相同的值。这不太可能产生良好的版本，除非版本号只使用单一的数字(即没有次要或补丁部分)。这样的版本号在某些情况下可能是合适的，但是为了更灵活和更健壮的运行时行为，项目应该努力遵循上面的原则。

## 20.4. 接口的兼容性

`VERSION`和`SOVERSION`目标属性，允许在操作系统级别以一种与平台无关的方式指定API版本控制。**CMake**还提供了其他属性，这些属性可用于定义**CMake**目标之间相互链接时的兼容性，可用于描述和强制执行版本编号无法捕获的信息。

考虑一个现实示例，如果有适当的**SSL**工具包，网络库只提供对<https://protocol>支持(相关的)安全功能。程序的其他部分需要基于**SSL**是否支持功能手动调整，而这个项目作为一个整体应该是一致的使用或不使用**SSL**特性。这可以通过接口兼容性属性来强制执行。

可以定义几种不同类型的接口兼容性属性，但最简单的是布尔属性。其基本思想是，库指定用于发布特定布尔状态的属性的名称，然后使用相关值定义该属性。当(链接在一起的)多个库为接口兼容性定义了相同的属性名时，**CMake**将检查它们是否指定了相同的值，如果不相同，则会报错。

```

add_library(networking net.cpp)
set_target_properties(networking PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_library(util util.cpp)
set_target_properties(util PROPERTIES
    COMPATIBLE_INTERFACE_BOOL SSL_SUPPORT
    INTERFACE_SSL_SUPPORT YES
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE networking util)
target_compile_definitions(myApp PRIVATE
    $<$<BOOL:$<TARGET_PROPERTY:SSL_SUPPORT>>:HAVE_SSL>
)

```

两个库目标都为属性名 `SSL_SUPPORT` 定义了接口兼容性。`COMPATIBLE_INTERFACE_BOOL` 属性包含一个名称列表，每个名称都需要在该目标上定义具有相同名称的 `INTERFACE_` prepended 属性。当这两个库作为 `myApp` 的链接依赖项使用时，**CMake** 检查这两个库是否用相同的值定义了 `INTERFACE_SSL_SUPPORT`。此外，**CMake** 还将自动用相同的值填充 `myApp` 目标的 `SSL_SUPPORT` 属性，然后将其用作生成器表达式的一部分，并将其作为编译宏提供给 `myApp` 的源代码。

这允许 `myApp` 代码根据是否将 SSL 支持编译到它使用的库中进行自我调整，`myApp` 不只是简单地检测 SSL 支持是否可用，它可以通过显式定义其 `SSL_SUPPORT` 属性来指定一个需求，以保存库必须兼容的值。这种情况下，**CMake** 不会自动填充 `myApp` 的 `SSL_SUPPORT` 属性，而是会比较这些值并确保库与指定的要求一致。

```

# Require libraries to have SSL support
set_target_properties(myApp PROPERTIES SSL_SUPPORT YES)

```

上面的例子在某种程度上是人为设计的，因为相同的约束可以通过其他方式进行。当项目变得更加复杂，目标分散在许多目录中，或者来自外部构建的项目时，接口兼容规范的优势就显现出来了。接口兼容性分配作为属性的目标，只需要在一个地方定义，然后就可以在任何目标上无副作用的使用。目标不需要知道接口兼容性的细节，只需要知道最终存储在目标的 `INTERFACE_` 属性。

**CMake** 还支持以字符串形式表示的接口兼容性。除了命名属性需要完全相同的值，并且可以容纳任意内容外，这些工作方式基本上与布尔值相同。前面的例子可以进行修改，要求库使用相同的 SSL 实现，而不仅仅在是否支持 SSL 上达成一致：

```

add_library(networking net.cpp)
set_target_properties(networking PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_library(util util.cpp)
set_target_properties(util PROPERTIES
    COMPATIBLE_INTERFACE_STRING SSL_IMPL
    INTERFACE_SSL_IMPL OpenSSL
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE networking util)
target_compile_definitions(myApp PRIVATE
    SSL_IMPL=$<TARGET_PROPERTY:SSL_IMPL>
)

```

上面例子中，`SSL_IMPL` 属性用作与 OpenSSL 作为 SSL 库的字符串接口兼容。就像布尔一样，`myApp` 目标可以定义 `SSL_IMPL` 属性来指定需求，而不是让 CMake 用库中的值填充它。

CMake 支持的另一种接口兼容性是数值。数值接口兼容性用于确定在一组库中，为某个属性定义的最小值或最大值，而不是要求属性具有相同的值。可以利用这一差异，让目标检测支持的最小协议版本，或者计算出链接到的不同库之间所需的最大缓冲区。

```
add_library(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 3
    INTERFACE_TMP_BUFFERS 200
)

add_library(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow PROPERTIES
    COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER
    COMPATIBLE_INTERFACE_NUMBER_MAX TMP_BUFFERS
    INTERFACE_PROTOCOL_VER 2
    INTERFACE_TMP_BUFFERS 15
)

add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE bigAndFast smallAndSlow)
target_compile_definitions(myApp PRIVATE
    MIN_API=$<TARGET_PROPERTY:PROTOCOL_VER>
    TMP_BUFFERS=$<TARGET_PROPERTY:TMP_BUFFERS>
)
```

上面例子中，`PROTOCOL_VER` 定义为一个最小的数字接口，所以 `myApp` 的 `PROTOCOL_VER` 属性将设置为 `INTERFACE_PROTOCOL_VER` 属性库的链接指定的最小值，在这种情况下就是 2。类似地，`TMP_BUFFERS` 定义为最大数值接口，`myApp` 的 `TMP_BUFFERS` 属性接收其链接库的 `INTERFACE_TMP_BUFFERS` 属性中最大的值，即 200。

此时，很自然地考虑对最小和最大数值接口使用相同的属性，从而允许在父接口中检测最小和最大的值。因为 CMake 不允许同一个属性使用多种接口，所以是不可能的。如果一个属性用于多种类型的接口，CMake 就不可能知道应该使用哪种类型来计算存储在父元素 `result` 属性中的值。例如，上面的例子中，如果 `PROTOCOL_VER` 同时是接口兼容性的最小值和最大值，CMake 就不能确定存储在 `myApp` 的 `PROTOCOL_VER` 属性中的值应该存储最小值还是最大值？所以必须使用单独的属性来实现：

```

add_library(bigAndFast strategy1.cpp)
set_target_properties(bigAndFast PROPERTIES
COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
INTERFACE_PROTOCOL_VER_MIN 3
INTERFACE_PROTOCOL_VER_MAX 3
)

add_library(smallAndSlow strategy2.cpp)
set_target_properties(smallAndSlow PROPERTIES
COMPATIBLE_INTERFACE_NUMBER_MIN PROTOCOL_VER_MIN
COMPATIBLE_INTERFACE_NUMBER_MAX PROTOCOL_VER_MAX
INTERFACE_PROTOCOL_VER_MIN 2
INTERFACE_PROTOCOL_VER_MAX 2
)

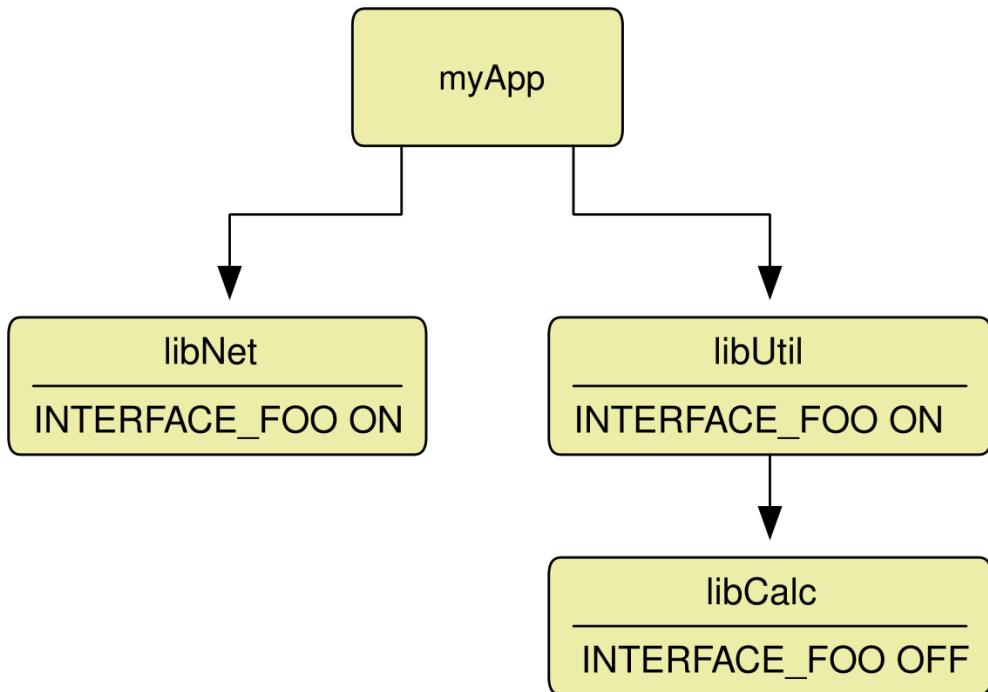
add_executable(myApp myapp.cpp)
target_link_libraries(myApp PRIVATE bigAndFast smallAndSlow)
target_compile_definitions(myApp PRIVATE
PROTOCOL_VER_MIN=${TARGET_PROPERTY:PROTOCOL_VER_MIN}
PROTOCOL_VER_MAX=${TARGET_PROPERTY:PROTOCOL_VER_MAX}
)

```

以上示例的结果是，`myApp` 根据链接库使用的协议，知道支持协议的版本范围。

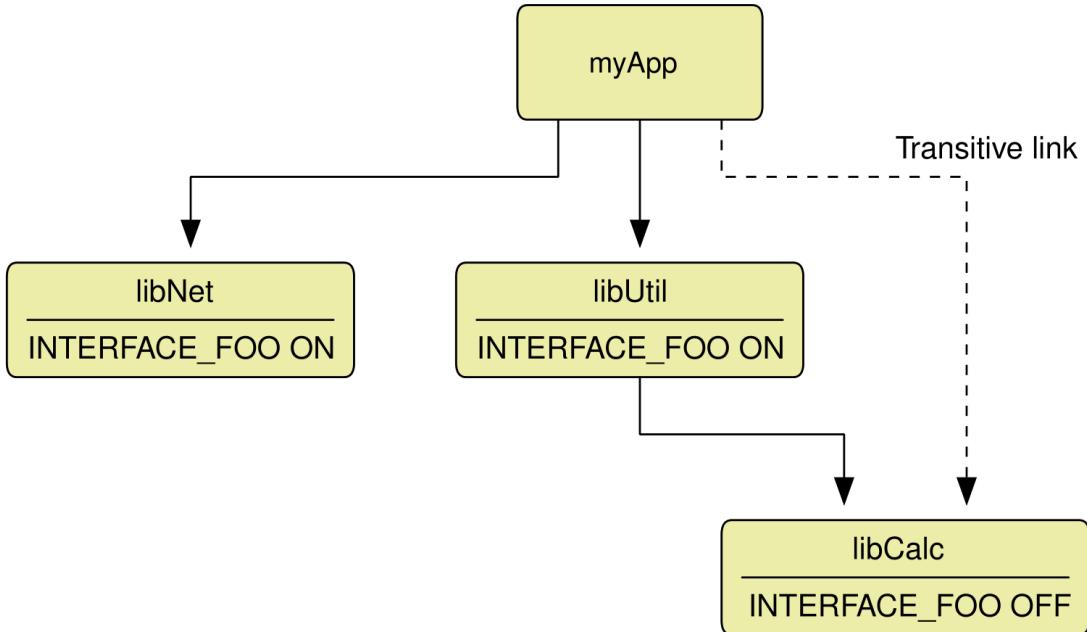
如果目标定义特定类型的兼容性接口，其他目标就不需要定义了。对于该特定属性，任何没有匹配兼容性接口的目标都将忽略。这确保了库只需要定义相关的兼容性接口。

当存在多个级别的库依赖时，处理兼容性接口的方式就有一些微妙。考虑下图的结构，包含多库和可执行目标，以及直接链接的依赖关系。



当所有的链接依赖都为 `PRIVATE`，只有 `libNet` 和 `libUtil` 是 `myApp` 的直接依赖项，因此这两个库的 `INTERFACE_FOO` 属性需要相同的值。`libCalc` 库不考虑这个属性值，因为与 `myApp` 不是直接依赖关系。此外，`libUtil` 库的唯一直接依赖的是 `libCalc` 库，因此 `libCalc` 库的 `INTERFACE_FOO` 属性不需要与其他库一致。尽管 `libUtil` 和 `libCalc` 都为相同的属性名定义了兼容性接口，但它们不是共同目标的直接依赖项，所以不需要兼容值。

现在考虑 `libCalc` 是 `libUtil` 的 `PUBLIC` 依赖关系。这种情况下，最终的链接关系实际上是这样的：



当 `libCalc` 是 `libUtil` 的 `PUBLIC` 依赖项时，任何链接到 `libUtil` 的目标也将链接到 `libCalc`。因此，`libCalc` 成为 `myApp` 的直接依赖项，因此需要参与 `libNet` 和 `libUtil` 的接口兼容性检查。因为影响范围可以超出公共链接关系的目标，所以在定义接口兼容性时，必须非常小心。

## 20.5. 符号可见性

简单地说，库可以看作是编译源码的容器，其他代码可以调用或使用的各种函数和全局数据。静态库容器实际上是对象文件的集合，将其组合在一起的工具称为归档器。另一方面，动态库由链接器生成，链接器处理目标代码、归档文件等，并决定在动态库文件中包含什么。一些函数和全局数据可隐藏，动态库之外的代码不能调用或使用。导出其他符号，这样动态库内部和外部的代码都可以访问，这称为符号的可见性。编译器有不同的方法来指定符号的可见性，也有不同的默认行为。一些默认情况下使所有符号可见，而另一些默认情况下隐藏所有符号。

编译器将单个函数、类和数据标记为可见或不可见的语法也有所不同，这增加了编写可移植动态库的复杂性。为了避免复杂，开发人员选择简单地使所有符号可见，从而避免显式地标记用于导出的任何符号。虽然这一招看起来不错，但有一些缺点：

- 每个函数、类、类型、全局变量等都可以使用。如果项目内容依赖于文档定义的公共标志，那么可以接受。
- 通过所有符号可见，调用代码不能禁止不应做的事。其他链接到库的代码可能会依赖内部符号，这使得动态库很难在不破坏使用项目的情况下更改实现或内部结构。

- 所有符号都视为可见时，链接器无法知道每个符号是否会被使用，必须将它们全部包含在最终的动态库中。只有一个子集的符号导出时，链接器可以识别代码不能使用的符号，将其可见性丢弃，会让生成的文件更小，在运行时加载的更快。
- 像C++支持模板的语言有可能定义了大量符号。默认情况下所有符号都可见，这可能会导致动态库的符号表变得非常大。极端情况下，这对运行时启动性能会产生影响。
- 函数的内部实现中，使用的库可以通过使用名称暴露。某些上下文中，这可能是一个安全问题，或者会暴露不应该对库使用者可见的商业IP。

上面强调了符号可见性是关于库API的公私性质，就像是关于动态库性能和包大小的底层机制一样。显然，只导出那些认为是公开符号是有好处的，但是编译器和平台特有的特性通常为多平台项目带来了障碍。**CMake** 将这些差异抽象为几个属性、变量和帮助模块后，简化了这个过程。

### 20.5.1. 指定默认的可见性

**Visual Studio** 编译器默认所有符号都隐藏，除非显式导出。其他编译器，如**GCC**和**Clang**相反，默认情况下所有符号可见，只有在显式地告知时才隐藏。如果项目希望在所有编译器和平台上具有相同的默认符号可见性，则必须选择这两种方法中的一种，但上一节强调的缺点为选择默认隐藏符号提供了有力的论据。实现隐藏默认可见性的第一步是在动态库目标上定义 `<LANG>_VISIBILITY_PRESET` 属性集。对于使用此功能的两种最常见语言，**C**和**C++**的属性名分别是 `C_VISIBILITY_PRESET` 和 `CXX_VISIBILITY_PRESET`。该属性值应该隐藏，这将改变默认的可见性，以隐藏所有符号。其他支持的值包括 `default`、`protected` 和 `internal`，这些值对跨平台项目不太有用。要么指定为默认行为，要么是隐藏，在某些上下文中可能具有更特殊的含义。

第二步是指定内联函数在默认情况下应该隐藏。对于大量使用模板的**C++**代码，这可极大地减少最终动态库文件的大小。此行为由目标属性 `VISIBILITY_INLINES_HIDDEN` 控制，并适用于所有语言。默认情况下，它应该为 `true` 来隐藏内联符号。

`<LANG>_VISIBILITY_PRESET` 和 `VISIBILITY_INLINES_HIDDEN` 可以在动态库目标上指定，或者通过适当的**CMake**变量设置默认值。创建目标时，`<LANG>_VISIBILITY_PRESET` 属性由 **CMake** 变量 `CMAKE_<LANG>_VISIBILITY_PRESET` 值初始化，其 `VISIBILITY_INLINES_HIDDEN` 属性由 `CMAKE_VISIBILITY_INLINES_HIDDEN` 初始化。这通常比为每个目标分别设置属性更为方便。

对于希望让所有符号在所有平台上默认可见的项目，只需要改变**Visual Studio**编译器的默认行为。**CMake 3.4** 版本中提供了 `WINDOWS_EXPORT_ALL_SYMBOLS` 目标属性，该属性提供了这种行为，但有一些注意事项。将这个属性定义为 `true` 值将导致**CMake** 编写一个 `.def` 文件，其中包含用于创建共享库的所有对象文件中的所有符号，并将该 `.def` 文件传递给链接器。这是相当粗暴的方法，可以防止源代码有选择地隐藏符号，所以应该只在所有符号都应该可见的地方使用。当创建动态库目标时，这个目标属性由 `CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS` 初始化。

### 20.5.2. 指定单个符号的可见性

大多数常见的编译器都支持指定单个符号的可见性，但是这样做的方式各不相同。**Visual Studio** 使用一种方法，而大多数其他编译器都遵循**GCC**使用的方法。两者有着相似的结构，但是使用不同的关键字。像**C**、**C++**和它们的衍生语言的源代码，可以使用预定义来进行可见性控制，项目可以指示**CMake** 提供适当的定义。

可以指定符号可见性的主要情况有三种：类、函数和变量。下面的例子包含了这三种情况的声明，请注意 `MYTOOLS_EXPORT` 的位置：

```
class MYTOOLS_EXPORT SomeClass {...}; // Export non-private members of a class
MYTOOLS_EXPORT void someFunction(); // Make a free function visible
MYTOOLS_EXPORT extern int myGlobalVar; // Make a global variable visible
```

构建上述实现的动态库时，需要用关键字替换 `MYTOOLS_EXPORT`，这些关键字指定是否导出该符号。另一方面，如果动态库之外的代码读取了相同的声明，必须用指定应该导入该符号的关键字替换 `MYTOOLS_EXPORT`。  
Windows 上，这些关键字采用 `_declspec(...)` 格式，而类 GCC 的编译器使用 `_attribute_(...)` 格式。

为所有编译器，以及导出和导入情况下的 `MYTOOLS_EXPORT` 提供正确的内容可能有些麻烦。再加上开发人员可能选择构建动态或静态库的混合库，复杂性就更大。幸运的是，CMake 提供了 `GenerateExportHeader` 模块，可以非常方便的处理这些细节。`GenerateExportHeader` 模块提供以下功能：

```
generate_export_header(  
    target  
    [BASE_NAME baseName]  
    [EXPORT_FILE_NAME exportFileName]  
    [EXPORT_MACRO_NAME exportMacroName]  
    [DEPRECATED_MACRO_NAME deprecatedMacroName]  
    [NO_EXPORT_MACRO_NAME noExportMacroName]  
    [STATIC_DEFINE staticDefine]  
    [NO_DEPRECATED_MACRO_NAME noDeprecatedMacroName]  
    [DEFINE_NO_DEPRECATED]  
    [PREFIX_NAME prefix]  
    [CUSTOM_CONTENT_FROM_VARIABLE var]  
)
```

通常不需要任何可选参数，只提供动态库目标名称即可。CMake 在当前二进制目录中写入头文件，使用小写的目标名称，并以 `_export.h` 作为头文件的名称。头文件为符号导出提供了定义，该定义具有类似的结构名称，这次使用的是附加了 `_EXPORT` 的目标名称。下面演示用法：

#### CMakeLists.txt

```
# Hide things by default  
set(CMAKE_CXX_VISIBILITY_PRESET hidden)  
set(CMAKE_VISIBILITY_INLINES_HIDDEN YES)  
  
# NOTE: myTools.cpp must #include myTools.h  
add_library(myTools myTools.cpp)  
target_include_directories(myTools PUBLIC  
    "${CMAKE_CURRENT_BINARY_DIR}"  
)  
  
# Write out mytools_export.h to the current binary directory  
include(GenerateExportHeader)  
generate_export_header(myTools)
```

#### myTools.h

```
#include "mytools_export.h"  
  
class MYTOOLS_EXPORT SomeClass  
{  
    // ...  
};  
MYTOOLS_EXPORT void someFunction();  
MYTOOLS_EXPORT extern int myGlobalVar;
```

当前的二进制目录并不是默认的头文件搜索路径，所以要将其添加为 `PUBLIC` 库的搜索路径，以确保 `mytools_export.h` 头文件可以找到库的源码和目标动态库。

如果不希望使用目标名称作为头文件名或预处理器定义名称的一部分，可以使用 `BASE_NAME` 选项。以同样的方式改变了，将 `_export.h` 附加到文件名中，将 `_EXPORT` 附加到预处理器定义中。

### CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools BASE_NAME fooBar)
```

### myTools.h

```
#include "foobar_export.h"

class FOOBAR_EXPORT SomeClass
{
// ...
};

FOOBAR_EXPORT void someFunction();
FOOBAR_EXPORT extern int myGlobalVar;
```

如果为文件和预处理器定义使用不同的名称，可以提供 `EXPORT_FILE_NAME` 和 `EXPORT_MACRO_NAME` 选项，而不是使用 `BASE_NAME`。与 `BASE_NAME` 不同，这两个选项提供的名称不需要进行任何修改。

### CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools
  EXPORT_FILE_NAME export_myTools.h
  EXPORT_MACRO_NAME API_MYTOOLS
)
```

### myTools.h

```
#include "export_myTools.h"

class API_MYTOOLS SomeClass
{
// ...
};

API_MYTOOLS void someFunction();
API_MYTOOLS extern int myGlobalVar;
```

`generate_export_header()` 函数提供的不仅是这个预处理定义，还提供其他的预处理，这些预处理可用于将符号标记为已弃用，或显式地指定不应该导出符号。后者可用于避免导出其他类的某些部分，例如：用于内部动态库使用，而不是由动态库外部代码使用的公共成员函数。默认情况下，这个预处理的名称由目标名称(如果指定了该名称，则为 `BASE_NAME` )组成，并附加了 `_NO_EXPORT`，如果需要可以通过 `NO_EXPORT_MACRO_NAME` 选项指定名称。

### CMakeLists.txt

```
include(GenerateExportHeader)
generate_export_header(myTools
  NO_EXPORT_MACRO_NAME REALLY_PRIVATE
)
```

### myTools.h

```

#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
    REALLY_PRIVATE void doInternalThings();
    // ...
};

```

函数弃用的方式类似，提供预处理，其中大写目标(或 `BASE_NAME`)名称后跟 `_DEPRECATED`，或者允许通过 `DEPRECATED_MACRO_NAME` 选项指定自定义名称。也可以给出 `DEFINE_NO_DEPRECATED` 选项，这可以提供额外的预处理定义，其名称通常为大写目标，或 `BASE_NAME` 后跟 `_NO_DEPRECATED` 组成。与其他预处理定义一样，这个名称也可以用 `NO_DEPRECATED_MACRO_NAME` 选项覆盖。一些编译器中，标记为 `deprecated` 的符号会导致编译时警告。这是一种有用的机制，鼓励开发人员更新代码，不再使用废弃符号。下面展示了如何使用废弃机制。

#### *CMakeLists.txt*

```

option(OMIT_DEPRECATED "Leave out deprecated parts of myTools")
if(OMIT_DEPRECATED)
    set(deprecatedOption "DEFINE_NO_DEPRECATED")
else()
    unset(deprecatedOption)
endif()
include(GenerateExportHeader)
generate_export_header(myTools
    NO_DEPRECATED_MACRO_NAME OMIT_DEPRECATED
    ${deprecatedOption}
)

```

#### *myTools.h*

```

#include "mytools_export.h"

class MYTOOLS_EXPORT SomeClass
{
public:
#ifndef OMIT_DEPRECATED
    MYTOOLS_DEPRECATED void oldImpl();
#endif
    // ...
};

```

#### *myTools.cpp*

```

#include "myTools.h"

#ifndef OMIT_DEPRECATED
void SomeClass::oldImpl() { ... }
#endif

```

上面的例子提供了CMake缓存变量来决定是否编译已弃用的项。开发人员可以在不编辑任何文件的情况下做出选择，因此验证使用或不使用过时API的行为很容易。如果持续集成构建需要设置为使用或不使用库的弃用部分进行测试，这就特别有用。当项目作为另一个项目的依赖项时，允许其他项目的开发人员通过更改CMake缓存变量来测试代码是否使用了废弃的符号。

不太常见但却很重要的案例也值得提及。有些项目可能希望同时构建库的动态版本和静态版本。本例中，同一组源代码需要允许为动态库的构建启用符号导出，但为静态库的构建禁用符号导出(请参阅下一节，了解为什么不总是这样)。在构建中需要这两种形式的库时，需要以不同的目标进行构建，`generate_export_header()` 函数会生成一个与单个目标紧密关联的头文件。为了支持此场景，生成的头文件包含在导出定义之前，检查是否存在预处理定义的逻辑中。这个特殊定义的名称遵循通常的模式，这一次是大写目标或 `BASE_NAME` 后跟 `_STATIC_DEFINE`，或者使用 `STATIC_DEFINE` 选项提供的自定义名称。定义这个特殊的预处理定义时，导出定义展开为无内容，这是将目标构建为静态库时所需要的。如果没有特殊的预处理器定义，导出定义通常具有内容，并在构建动态库时按预期进行。

当为同一组源文件构建动态库和静态库时，应该为 `generate_export_header()` 函数提供与动态库对应的目标。然后，只在静态库的目标上设置特殊的预处理定义。`BASE_NAME` 选项通常还用于使各种符号的展现形式都很直观，而不是只特定于动态库。下面演示了实现预期结果所需的结构：

```
# Same source list, different library types
add_library(myShared SHARED ${mySources})
add_library(myStatic STATIC ${mySources})

# Shared target used for generating export header
# with the name myTools_export.h, which will be suitable
# for both the shared and static targets
include(GenerateExportHeader)
generate_export_header(myShared BASE_NAME myTools)

# Static target needs special preprocessor define
# to prevent symbol import/export keywords being added
target_compile_definitions(myStatic PRIVATE
    MYTOOLS_STATIC_DEFINE
)
```

从前面的讨论中可以看出，`generate_export_header()` 函数定义了许多不同的预处理，不同的目标有可能对其中一些使用相同的名称。为了减少名称冲突，`PREFIX_NAME` 选项允许指定附加字符串，该字符串将前置到每个预处理器的名称中。这个选项通常与项目相关，有效地将项目生成的所有预处理器名称放入类似于特定于项目的命名空间中。

未讨论的选项是 `CUSTOM_CONTENT_FROM_VARIABLE`，它在CMake 3.7中添加。这个选项允许任意内容注入生成的头文件末尾，毕竟不同的预处理逻辑已添加。使用时必须为该选项提供变量的名称，该变量的内容应该是注入的，而不是内容本身。

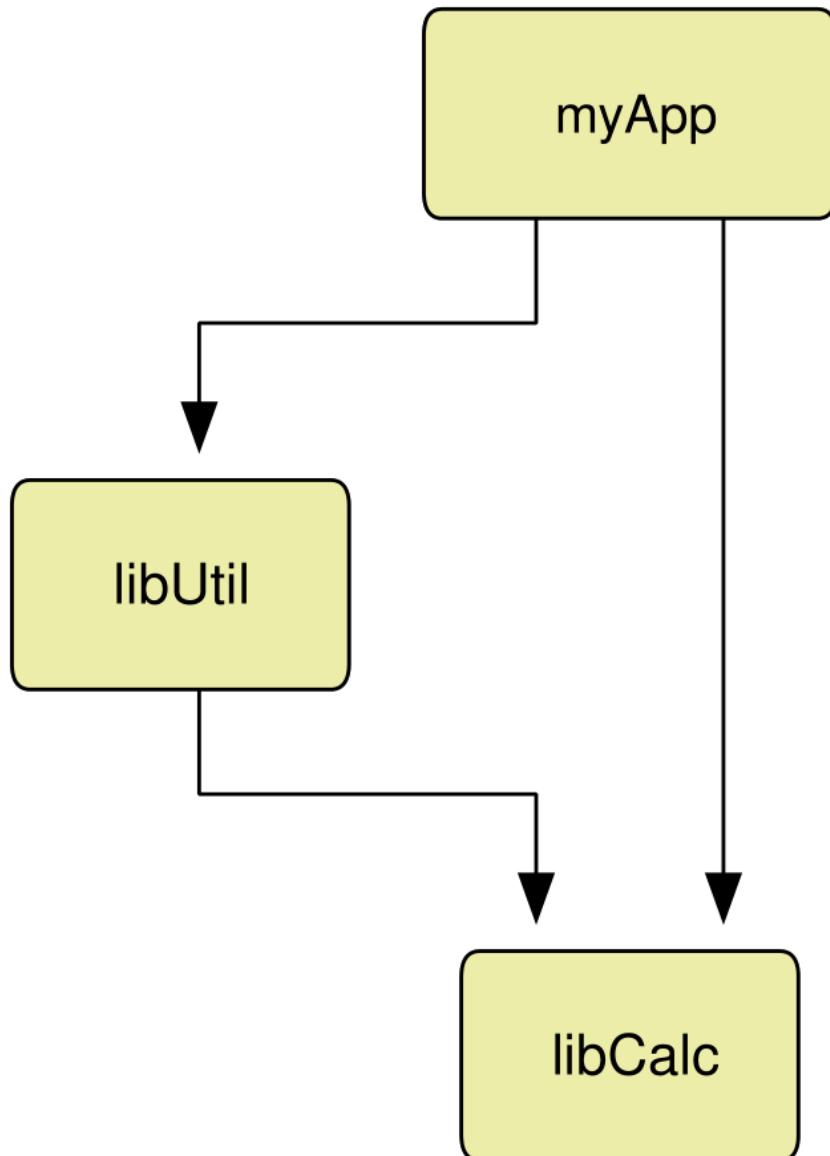
```
string(TIMESTAMP now)
set(customContents /* Generated: ${now} */)
generate_export_header(myTools
    CUSTOM_CONTENT_FROM_VARIABLE customContents
)
```

## 20.6. 混合静态和动态库

当项目将所有库构建为静态时，该构建可能会对库链接依赖关系宽容一些。项目可能会忽略目标间的依赖关系，但当各种静态库链接到最终的可执行文件中时，缺少的库依赖关系就会得到满足，因为它们会按照需要的顺序为可执行文件显式地列出。构建会成功，但要经过一段时间的试错构建，让链接器报出丢失的符号，添加更多丢失的库或重新排序现有的库等等。

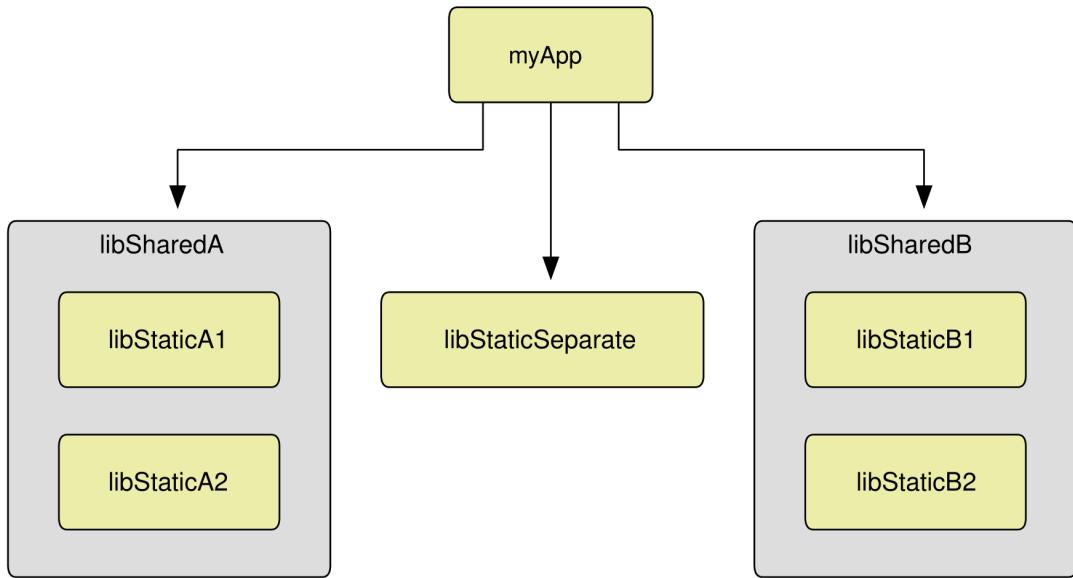
特别是在许多小型库的项目中，这种情况的成功更多的是靠运气，而不是良好的设计。如果链接指定的依赖项有静态库，CMake会自动处理并链接到这些依赖项。因此，即使 `PRIVATE / PUBLIC` 性质的指定依赖不正确，但静态库总是视为 `PUBLIC`，这有时会让构建工作中依赖关系的描述不准确，但也能正确构建。

库目标定义为动态和静态混合时，链接依赖关系的正确性就变得更为重要。考虑以下一系列目标：



如果 `libUtil` 和 `libCalc` 是静态库，上述链接依赖关系安全。如果 `libUtil` 是动态库，上述链接依赖关系可能会复制数据。如果 `libCalc` 定义了全局数据，比如对于类的单例或静态数据是通用的，`myApp` 和 `libUtil` 可能都有自己独立的数据实例。之所以成为可能，是因为 `myApp` 和 `libUtil` 都需要链接器来解析符号，所以两个调用都可能决定全局数据，并在可执行库或动态库中设置内部实例。如果不是全局数据导出的符号，链接器不会看到已经创建的实例在 `libUtil` 需要连接到 `myApp` 中。最终结果是在 `myApp` 中创建了第二个实例，这会导致难以跟踪的运行时问题。这种情况的典型表现是，变量神奇地出现在函数调用中，将值从可执行程序或动态库更改为另一个动态库使用的值。

类似于上述场景的情况可能以多种不同的形式出现，相同的原则适用于每一种情况。如果静态库链接到动态库中，这个动态库不应该与链接同一个静态库的任何其他库或可执行文件结合在一起。理想情况下，如果动态库和静态库混合在一起，静态库应该只链接到一个动态库中，而需要从这些静态库中获得符号的目标都应该链接到动态库。动态库本来就有自己的API，而静态库可能对这些API的实现有所贡献。



当涉及到符号可见性时，使用静态库来构建的动态库会带来一系列问题。通常，来自静态库的代码不会导出，因此不会作为动态库导出符号的一部分。解决这个问题的方法是在动态库上使用 `generate_export_header()` 函数，然后让静态库用相同方式的导出定义。实现此功能的关键是确保静态库具有动态库目标的编译定义，并附加 `_EXPORTS`，这就是生成的头文件检测代码是否构建为动态库的一种方式。

#### CMakeLists.txt

```

add_library(myShared SHARED shared.cpp)
add_library(myStatic STATIC static.cpp)

include(GenerateExportHeader)
generate_export_header(myShared BASE_NAME mine)

target_link_libraries(myShared PRIVATE myStatic)
target_include_directories(myShared PUBLIC ${CMAKE_CURRENT_BINARY_DIR})
target_include_directories(myStatic PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# This makes the static library code appear to be part of the shared library
# library as far as the generated export header is concerned
target_compile_definitions(myStatic PRIVATE myShared_EXPORTS)

```

#### shared.h

```

#include "mine_export.h"

MINE_EXPORT void sharedFunc();

```

#### static.h

```

#include "mine_export.h"

MINE_EXPORT void staticFunc();

```

考虑另一个情况，当链接动态库时，链接器是否会丢弃静态库中定义的代码或数据？如果确定没有使用特定符号，链接器可能会将其作为优化丢弃。可能需要采取特殊的方式避免这种情况。一种选择是让动态库显式地保留每个符号。这样做的好处是，适用于所有的编译器和链接器，但对于重要的项目是不可行。另一种方

法实际上需要添加特定于链接的标志，例如Unix系统上的 `ld` 链接器的 `--whole-archive` 和 `--no-whole-archive`，或者Visual Studio中的 `/WHOLEARCHIVE`，但是并非所有链接器都有这种功能。如果动态库使用静态库导出的每个符号是不切实际的，那么可以考虑将这些静态库转换为动态库。

如果动态库仅以私有方式链接到静态库(意味着不需要导出静态库的符号)，那么情况就简单得多。某些平台上，除了简单地将动态库链接到静态库之外，不需要进一步操作。其他情况下，可能会出现一两个小问题。例如，在许多64位Unix系统上，如果要将代码放入动态库，必须将其编译为位置独立的代码，而对于静态库则没有这样的要求。但是，如果动态库链接到静态库，则必须将静态库构建为与位置无关的。

CMake提供了 `POSITION_INDEPENDENT_CODE` 目标属性，作为需要在平台上处理位置无关行为的一种方式。当设置为`true`时，这将构建位置独立的代码。默认情况下，该属性对于 `SHARED` 和 `MODULE` 库是 `ON`，对于所有其他类型的目标是 `OFF`。可以通过设置 `CMAKE_POSITION_INDEPENDENT_CODE` 来覆盖默认值，当创建目标时，将用于初始化 `POSITION_INDEPENDENT_CODE` 目标属性。

```
add_library(myShared SHARED shared.cpp)
add_library(myStatic STATIC static.cpp)
target_link_libraries(myShared PRIVATE myStatic)

set_target_properties(myStatic PROPERTIES
  POSITION_INDEPENDENT_CODE ON
)

set(CMAKE_POSITION_INDEPENDENT_CODE ON)
add_library(myOtherStatic STATIC other.cpp)
target_link_libraries(myShared PRIVATE myOtherStatic)
```

## 20.7. 总结

构建按需加载的插件可以使用 `MODULE` 库，以及用于链接的 `SHARED` 库。使用 `SHARED` 库时，无论是出于暴露API的目的，还是为了隐藏敏感的实现细节，必须严格控制向库使用者公开的符号。如果目标是将库作为发布包的一部分进行交付，大多数情况下，动态库往往比静态库更受青睐。

如果目标使用来自库的内容，应该直接链接到库。即使库已经是其他的链接依赖，也不要依赖于目标直接使用的间接链接依赖。如果其他目标更改了实现，并且不再链接到库，主目标将不再构建。此外，表达正确的链接依赖类型：`PRIVATE`、`PUBLIC` 或 `INTERFACE`。这可确保了CMake正确地处理动态库和静态库的链接依赖关系。指定与正确可见级别的所有直接依赖关系，对于确保CMake构造的库顺序正确，且形成可靠链接器命令行来说至关重要。

使用正确的链接可见性，目标不需要知道内部使用的库依赖关系，只需要链接到一个库，并让库定义自己的依赖关系。然后，CMake会确保所有的库都在最终的链接器命令行中以正确的顺序指定。抵制简单地将所有链接依赖公开的诱惑，这会让私有库的可见性扩展到不需要的地方。当项目准备发布或发布时，这一点尤其重要。

考虑尽早使用库版本控制策略。当一个库发布，版本号就有了一些关于二进制兼容性的含义。使用 `VERSION` 和 `SOVERSION` 目标属性来指定库的版本(即使在项目早期这些属性设置为基本占位符)，在没有其他策略的情况下，一个合理的选择是将版本编号从0.1.0开始，人们倾向于将0.0.0认为是默认值，或者错误地认为没有设置版本，而1.0.0有时认为是第一个公开发行版本。强烈考虑采用语义版本控制来处理以后的版本变更，库版本的变化可能会对发布过程、打包等方面产生巨大影响，在这些库公开发布之前，开发人员需要花时间了解动态库版本号的含义。还要考虑项目版本和库版本之间是否应该有任何关系。一旦发布了第一个版本，就很难改变这种关系，所以要谨慎地连接它们，除非它们之间有很强的关联(作为SDK交付一组一致的库的项目就是这种强关联的一个例子)。

如果有特定的支持工具集、库可用，一些项目可以选择性地提供某些功能。为了允许构建的其他部分，或其他项目的检测，或检查与可选功能或特性的一致性，可以提供兼容性接口的详细信息。考虑是否有问题的特性需要在库之外具有可见性，比如允许目标检测是否支持该特性，或者确认所选的实现是否提供了所需的功能。还要考虑指定和使用兼容性接口带来的利益与增加的复杂性比例是否合适。因为库依赖层次结构越深，高效地使用兼容性接口就越困难。

项目生命周期的早期就要考虑符号的可见性，因为回头在考虑符号的可见性细节可能会翻新整个项目，实施起来会非常困难。创建库时，始终要养成这样一种习惯：考虑一个特定的类、函数或变量是否应该在库之外访问。认为任何外部可见性都很难改变，而内部之间的事情可以自由地修改。使用隐藏可见性作为默认值，并显式地标记要导出的每个实体，理想情况下使用 `generate_export_header()` 函数提供的宏，以便CMake处理各种平台差异。还要考虑使用该函数提供的废弃宏，以清楚地标识库API中已弃用，可能在未来版本中删除的那些部分API。

混合使用动态库和静态库时要格外小心。可能的话，最好使用其中之一，而不是两者都使用，这样可以避免一些与构建设置一致性和符号可见性控制相关的问题。如果混合使用这两种库类型是有意义的，请尝试确保静态库只链接到一个动态库中。将静态库视为动态库中的子组，外部目标仅链接到动态库。更好的方式是，考虑直接将代码从静态库中提取到动态库中，完全摆脱静态库。**28.5.1**节中会介绍相关的技术，并演示如何逐步向现有目标添加源，从而允许目标源在子目录中进行累积。

# 第21章：工具链与交叉编译

考虑构建软件和相关工具过程时，开发人员通常会考虑编译器和链接器。这些是开发人员接触到的主要工具，但是还有许多其他工具、库和文件也对这个过程有贡献。简单来说，这些工具和其他文件统称为工具链。

对于桌面或传统服务器的应用程序，通常不需要太深入地考虑工具链。大多数情况下，决定使用主流平台工具链的哪个版本是非常复杂的。**CMake**通常不需要太多帮助就能找到工具链，开发人员可以继续编写软件。然而，对于移动或嵌入式开发，情况就不同了，工具链通常需要由开发人员以某种方式指定。像指定不同的目标系统名称那样简单，也可以像指定单个工具和目标根文件系统的路径那样复杂。还可能需要设置特殊的标志，使工具生成支持芯片组、具有所需性能特征等的二进制文件。

选择了工具链，**CMake**就会在内部执行相当多的处理，测试工具链，确定它所支持的特性，设置各种属性和变量等等。即使使用默认工具链的构建也是如此，不仅仅是交叉编译的构建。这些测试的结果可以在**CMake**的输出中看到，第一次运行一个给定的构建目录，**macOS**的例子如下所示(为简便起见，显示的C和CXX编译器路径已经折叠)：

```
-- The C compiler identification is AppleClang 9.0.0.9000039
-- The CXX compiler identification is AppleClang 9.0.0.9000039
-- Check for working C compiler: /Applications/Xcode.app/.../cc
-- Check for working C compiler: /Applications/Xcode.app/.../cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++
-- Check for working CXX compiler: /Applications/Xcode.app/.../c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
```

通常在第一个 `project()`，缓存工具链测试结果时进行大量处理。`enable_language()` 在启用前，未启用的语言时也会触发这样的处理，另一个 `project()` 调用添加以前未启用的语言也会触发这样的处理。当启用一种语言，缓存信息则会使用，而不是重新测试工具链，甚至在随后的**CMake**运行时也是如此。这导致两个结果：

- 当构建目录配置了特定的工具链，就不能(安全地)更改它。某些情况下，**CMake**可能会检测到工具链的修改，并丢弃之前的结果，但这只会丢弃缓存的与工具链直接相关的信息。基于**CMake**所知道的缓存工具链信息之外的任何其他缓存都不会重置。因此，更改工具链之前，应该完全清除构建目录(仅仅删除 `CMakeCache.txt` 文件可能不够，其他细节可能缓存在不同的位置)。
- 不同的工具链不能混合在一个项目内。**CMake**从根本上认为一个项目始终使用单一工具链。为了使用多个工具链，必须构造项目来执行外部构建的部分(在27.1节和28.1节中进行讨论)。

## 21.1. 工具链文件

如果默认的工具链不合适，推荐使用工具链文件来指定工具链的信息。这是一个普通的**CMake**脚本，通常包含很多 `set(...)`。这些**CMake**变量用来描述目标平台的变量、各种工具链组件位置等等。工具链文件的名称通过特殊的缓存变量 `CMAKE_TOOLCHAIN_FILE` 传递给**CMake**，如下所示：

```
cmake -DCMAKE_TOOLCHAIN_FILE=myToolchain.cmake path/to/source
```

可以使用绝对路径，也可以使用相对路径，如上面的例子中所示，**CMake**首先查找与构建目录相关的路径，如果没有找到，则查找与源目录相关的路径。这个工具链文件必须在第一次运行**CMake**的构建目录时指定，不能在之后添加或更改指向不同的工具链。因为变量本身是缓存的，所以没有必要为后续的**CMake**运行重新指定它。

每次调用 `project()` 都会读取工具链文件，开发人员不需要考虑太多，但可能会导致一些微妙的意外行为。如果工具链文件设置或修改项目本身操纵的变量，或者工具链文件错误地假定为整个项目只处理一次，那么项目开发时使用的 `project()`，会损坏的工具链的设置，或没有让变量进行任何改变。因此，开发人员应该确保工具链最小化，对项目要做的事情尽可能少做假设。理想情况下，工具链文件应该从项目中解耦，甚至可以在不同的项目之间重用，因为它们应该只描述工具链，而不是如何与项目进行交互。

工具链文件的内容可以不同，但总体上需要做如下的几个事情：

- 描述目标系统的基本信息。
- 提供工具的路径(通常是编译器的路径)。
- 设置工具的默认标志(通常只针对编译器，也可能是链接器)。
- 交叉编译的情况下设置目标平台文件系统根目录的位置。

工具链文件中包含其他逻辑很常见，特别是各种 `find_…()` 命令。虽然在某些情况下，这样的逻辑是合适的，但在大多数情况下，这样的逻辑应该是项目的一部分。只有项目知道自己要寻找什么，所以工具链不应该假设项目想要做什么。

## 21.2. 定义目标系统

描述目标系统信息的基本变量：

- `CMAKE_SYSTEM_NAME`
- `CMAKE_SYSTEM_PROCESSOR`
- `CMAKE_SYSTEM_VERSION`

`CMAKE_SYSTEM_NAME` 是最重要的，它定义了目标平台的类型，而 `CMAKE_HOST_SYSTEM_NAME` 定义了执行构建的平台。`CMake`本身总是设置 `CMAKE_HOST_SYSTEM_NAME`，而 `CMAKE_SYSTEM_NAME` 可以(通常是)由工具链文件设置。如果 `CMake`能够直接在目标平台上运行，可以将 `CMAKE_SYSTEM_NAME` 设置为 `CMAKE_HOST_SYSTEM_NAME`。因此，典型的值包括Linux、Windows、QNX、Android或Darwin，但对于某些情况(例如：嵌入式设备)，可以使用Generic代替。某些情况下，平台名称也有一些变体，比如WindowsStore和WindowsPhone。如果在工具链文件中设置了 `CMAKE_SYSTEM_NAME`，那么`CMake`也会将 `CMAKE_CROSSCOMPILING` 变量设置为true，即使值与 `CMAKE_HOST_SYSTEM_NAME` 相同。如果没有设置 `CMAKE_SYSTEM_NAME`，将赋予与 `CMAKE_HOST_SYSTEM_NAME` 相同的值。

`CMAKE_SYSTEM_PROCESSOR` 旨在描述目标平台的硬件架构。如果未指定，将赋予与 `CMAKE_HOST_SYSTEM_PROCESSOR` 相同的值，该值由`CMake`自动填充。交叉编译场景中，或者在相同系统类型的64位主机上为32位平台构建时，这将导致 `CMAKE_SYSTEM_PROCESSOR` 设置不正确。因此，建议设置 `CMAKE_SYSTEM_PROCESSOR`，如果架构与主机不匹配，那么这个项目就构建的有问题。基于错误的 `CMAKE_SYSTEM_PROCESSOR` 值，错误决策可能会导致难以检测或诊断的问题。

根据 `CMAKE_SYSTEM_NAME` 的设置，`CMAKE_SYSTEM_VERSION` 变量有不同的含义。例如，系统名为WindowsStore、WindowsPhone或WindowsCE，系统版本将用于定义使用哪个Windows SDK。值可能是8.1或10.0，或者非常特定的版本，如10.0.10240.0。如果 `CMAKE_SYSTEM_NAME` 设置为Android，那么 `CMAKE_SYSTEM_VERSION` 通常会解释为Android API版本，并且必须是正整数。对于其他系统名，经常会看到 `CMAKE_SYSTEM_VERSION` 设置为1，或者根本没有设置。`CMake`文档的工具链部分提供了 `CMAKE_SYSTEM_VERSION` 不同用法的示例，但是变量的含义和允许值集并没明确定义。因此，建议项目在实现时，如有依赖 `CMAKE_SYSTEM_VERSION` 的逻辑需要谨慎。

通常，这三个 `CMAKE_SYSTEM_…` 变量描述了目标系统信息，但也有例外：

- 所有的Apple平台都使用Darwin作为 `CMAKE_SYSTEM_NAME`，甚至iOS、tvOS或watchOS也是如此。`CMAKE_SYSTEM_PROCESSOR` 和 `CMAKE_SYSTEM_VERSION` 对于Apple平台没有特别的意义。通过使用不同的变量 `CMAKE OSX_SYSROOT` 来指定目标系统，该变量选择要用于构建的基本SDK。根据选择的SDK确定目标设备，开发人员可以在构建时选择设备或模拟器。这是一个复杂的主题，将在22.5节中详细介绍。
- `CMAKE_SYSTEM_PROCESSOR` 变量通常针对Android平台时不设置。这将在下面的21.6.3节中进一步讨论。

## 21.3. 选择工具

构建使用的所有工具中，从开发人员的角度来看，编译器可能是最重要的。编译器的路径由 `CMAKE_<LANG>_COMPILER` 变量控制，可以在工具链文件或命令行中设置该变量来手动控制所使用的编译器，也可以忽略该变量允许CMake自动选择一个。如果可执行文件的名称是手动提供的，而没有路径，CMake将使用 `find_program()` 搜索它(在23.3节中介绍)。如果提供了编译器的完整路径，则将直接使用它。如果没有手动指定编译器，CMake将根据目标平台和生成器的内部默认设置选择一个编译器。

大多数语言还支持通过指定环境变量来设置编译器，而不是必须设置 `CMAKE_<LANG>_COMPILER`。它们通常遵循一些常见的约定，比如C编译器使用CC, C++编译器使用CXX, Fortran编译器使用FC等等。环境变量只有在CMake第一次在构建目录中运行时才会起作用，并且只有在相应的 `CMAKE_<LANG>_COMPILER` 变量没有使用工具链文件或CMake命令行设置时才会起作用。

当有了编译器，CMake就可以识别并尝试确定它的版本。该编译器信息分别通过 `CMAKE_<LANG>_COMPILER_ID` 和 `CMAKE_<LANG>_COMPILER_VERSION` 变量提供。编译器ID是一个简短的字符串，用于区分不同的编译器，通用的值是GNU、Clang、AppleClang、MSVC、Intel等等。CMake文档为 `CMAKE_<LANG>_COMPILER_ID` 提供了支持ID的完整列表。如果能够确定编译器的版本，通常会使用 `major.minor.patch.tweak` (不是所有的版本组件都需要出现)。

除了 `CMAKE_<LANG>_COMPILER_ID` 和 `CMAKE_<LANG>_COMPILER_VERSION` 外，还支持不带 `CMAKE_` 前缀的生成器表达式。变量或生成器表达式都可以仅针对特定编译器或编译器版本，有条件地添加内容。例如，GCC 7引入新选项 `-fcode-hoisting`，下面展示了在C++编译时的两种添加方法：

```
add_library(foo ...)

# Conditionally add -fcode-hoisting option using variables
if(CXX_COMPILER_ID STREQUAL GNU AND
NOT CXX_COMPILER_VERSION VERSION_LESS 7)
target_compile_options(foo PRIVATE -fcode-hoisting)
endif()

# Same thing using generator expressions instead
target_compile_options(foo PRIVATE
$<$<AND:$<CXX_COMPILER_ID:GNU>,
$<VERSION_GREATER_EQUAL:$<CXX_COMPILER_VERSION>,7>>:-fcode-hoisting>
)
```

编译器ID是识别使用编译器最健壮的方法。CMake 3.0之前，Apple Clang编译器和Clang一样，都有编译器ID Clang。从CMake 3.0开始，苹果的编译器编译器ID AppleClang，这样就可以与Clang区别开来。添加策略CMP0025是为了允许那些需要使用旧的行为的项目。

当确定了编译器的路径，CMake就能够为编译器和链接器找出一组默认标志。这些在项目中可见，如 `CMAKE_<LANG>_FLAGS`，`CMAKE_<LANG>_FLAGS_<CONFIG>`，`CMAKE_<TARGETTYPE>_LINKER_FLAGS` 和 `CMAKE_<TARGETTYPE>_LINKER_FLAGS_<CONFIG>` 变量，这些在14.3节中已经介绍过了。开发人员可以使用相同名称，附加 `_INIT` 的变量，将自己项目使用的标志添加到这些变量的默认值集中。这些 `_INIT` 变量只用于设置初始值的默认值。

常见的错误是在工具链文件中设置非 `...INIT` 变量(例如, 设置 `CMAKE_<LANG>_FLAGS` 而不是 `CMAKE_<LANG>_FLAGS_INIT`)。这将导致丢弃或隐藏开发人员对缓存中变量所做的任何更改。因为在 `project()` 调用时也会重新读取工具链文件, 所以可以会丢弃项目本身对这些变量所做的更改。使用 `...INIT` 设置变量, 而不是开始只用初始默认值, 之后再用非 `...INIT` 的变量对其进行修改的方式。

举个例子, 考虑一个工具链文件, 开发人员会使用特殊的编译器标记来设置构建, 以便进行调试(可以跨多个项目重用一些复杂的开发人员专用逻辑, 而不必将其添加到每个项目中)。以下选择了GNU编译器并启用警告标志:

```
set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)
set(extraOpts "-Wall -Wextra")
set(CMAKE_C_FLAGS_DEBUG_INIT ${extraOpts})
set(CMAKE_CXX_FLAGS_DEBUG_INIT ${extraOpts})
```

不幸的是, CMake在将开发人员指定的 `...INIT` 选项与它通常提供的默认选项组合的方式上存在一些不同。大多数情况下, CMake会给 `...INIT` 变量指定的选项添加更多选项, 但是对于一些平台/编译器组合(特别是较老的或不太常用的), 开发人员指定的 `...INIT` 值可以丢弃。这源于这些变量的历史, 过去只用于内部, 单方面设置 `...INIT` 值。从CMake 3.7开始, `...INIT` 变量设计为通用变量, 对于常用的编译器, 行为修改为追加而不是替换。对于非常旧的或不再维护的编译器来说, 行为保持不变。

有些编译器更多地相适编译器的驱动程序, 期望命令行参数来指定要编译的目标平台/架构。Clang和QNX qcc就是这种编译器。对于那些CMake认为需要这样参数的编译器, 可以在工具链文件中设置 `CMAKE_<LANG>_COMPILER_TARGET` 变量来指定目标。在支持的情况下应该使用, 而不是尝试手动添加 `CMAKE_<LANG>_FLAGS_INIT` 标志。

另一种不太常见的情况是, 编译器工具链不包括其他支持工具, 如归档器或链接器。这些编译器驱动程序通常支持命令行参数, 可用于指定这些工具的位置。CMake提供了 `CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN` 变量, 该变量可用于指定程序所在的目录。

## 21.4. 系统根目录

很多情况下工具链是需要的, 因为可以在目标平台上找到, 有时项目需要访问一组库、头文件等。处理这个问题的常见方法是为构建提供目标平台文件系统根目录的简化版本(甚至是完整版本), 称为系统根, 或为 `sysroot`。`sysroot`基本上就是目标平台的根文件系统, 可以挂载或复制到可以通过主机的文件系统访问的路径。工具链包通常提供一个最小的`sysroot`, 其中包含编译和链接所需的各种库。

CMake对`sysroot`有相当广泛的支持。工具链文件可以将 `CMAKE_SYSROOT` 变量设置为`sysroot`位置, 仅使用该信息, CMake就可以优先在`sysroot`中找到库、头文件等所需之物, 而不是主机上的同名文件。许多情况下, CMake还会自动向底层工具添加必要的编译器/链接器标志, 使它们知道`sysroot`的位置。对于更复杂的情况, 需要提供不同的`sysroot`来编译和链接(如使用统一头文件的Android NDK), 使用CMake 3.9或更高版本时, 工具链文件可以设置为 `CMAKE_SYSROOT_COMPILE` 和 `CMAKE_SYSROOT_LINK`。

开发人员可以选择在主机挂载点下挂载完整的目标文件系统, 并将其设置为`sysroot`。可以将其挂载为只读, 如果不是这样, 构建时不要修改它。因此, 在构建项目时, 可能需要将其安装到其他地方, 而不是写入 `sysroot`区域。CMake提供了 `CMAKE_STAGING_PREFIX` 变量, 可以用来设置分段点, 任何安装命令都将安装到该分段点以下(有关此区域的讨论, 请参阅25.1.2节)。这个分段区域可以是正在运行目标系统的挂载点, 可以在安装之后测试已安装的二进制文件。在主机上对目标系统进行交叉编译时, 这种方式特别有用, 否则在目标系统上构建会很慢(例如: 在桌面机器上构建树莓派目标)。在23.1.2节还讨论了 `CMAKE_STAGING_PREFIX` 如何影响 CMake搜索库、头文件的方式。

## 21.5. 检查编译器

`project()` 或 `enable_language()` 触发对编译器和语言特性的测试时，会在内部调用 `try_compile()` 来执行各种检查。如果提供了工具链文件，每次 `try_compile()` 调用都会进行读取，因此测试项目将以类似的方式配置。`CMake` 会自动传递相关的变量，比如 `CMAKE_<LANG>_FLAGS`，但是工具链文件可能希望传递其他变量到测试构建中。由于主构建将首先读取工具链文件，因此工具链文件本身可以定义应该传递哪些变量来进行构建测试。这是通过向 `CMAKE_TRY_COMPILE_PLATFORM_VARIABLES` 变量添加变量名来实现的(不要在项目中设置该变量，只能在工具链文件中设置)。使用 `list(APPEND)` 而不是 `set()`，这样 `CMake` 添加的任何变量都不会丢失。`CMAKE_TRY_COMPILE_PLATFORM_VARIABLES` 最后是否包含重复项并不重要，重要的是所需的变量名是否存在。

`try_compile()` 通常编译并链接测试代码以生成可执行文件，一些交叉编译场景中，如果运行链接器需要自定义标志或链接器脚本，或者不希望调用链接器，则会出现问题(交叉编译可能有这样的限制)。如果使用的是 `CMake 3.6` 或更高版本，可以通过将 `CMAKE_TRY_COMPILE_TARGET_TYPE` 设置为 `STATIC_LIBRARY` 来创建一个静态库。这就避免了对链接器的需要，但是仍然需要一个归档工具。`CMAKE_TRY_COMPILE_TARGET_TYPE` 也可以有 `EXECUTABLE`，如果没设置就为默认行为。`CMake 3.6` 之前，目前已弃用的 `CMakeForceCompiler` 模块可以用于避免 `try_compile()` 的调用，但现在 `CMake` 严重依赖这些测试来找出支持功能的编译器，所以不推荐使用 `CMakeForceCompiler`。

虽然在编译器检查期间不会调用 `try_run()`，但是 `try_run()` 与 `try_compile()` 密切相关，其行为会受到交叉编译的影响。`try_run()` 实际上是 `try_compile()` 之后尝试构建的可执行文件。当 `CMAKE_CROSSCOMPILING` 设置为 `true` 时，`CMake` 可以修改其运行测试可执行文件的逻辑。如果设置了 `CMAKE_CROSSCOMPILING_EMULATOR` 变量，`CMake` 将把它前置到命令中，否则该命令将用于在目标平台上运行可执行文件，并使用该命令在主机平台上运行可执行文件。如果 `CMAKE_CROSSCOMPILING` 为 `true` 时 `CMAKE_CROSSCOMPILING_EMULATOR` 未设置，`CMake` 要求工具链或项目手动设置一些缓存变量。这些变量提供了退出代码以及标准输出和标准错误输出，如果可执行文件能够在目标平台上运行，则将获得这些输出。必须手动设置显然不方便，并且容易出错，所以在 `CMAKE_CROSSCOMPILING_EMULATOR` 没有设置时，项目应避免调用 `try_run()` 做交叉编译。无法避免手动定义变量的情况下，`CMake` 文档为 `try_run()` 变量集提供了必要的信息，如何使用 `CMAKE_CROSSCOMPILING_EMULATOR` 也会在 24.6 节讨论。

## 21.6. 例子

选择下面的示例来突出本章讨论的概念，`CMake` 参考文档的工具链部分包含了对各种不同目标平台的进一步示例。

### 21.6.1. 树莓派 Raspberry Pi

交叉编译的树莓派应用程序是很好的使用方式，通常使用 `CMake` 处理交叉编译。第一步是获得编译器工具链，常见的方法是使用像 `crosstool-NG` 这样的程序。这个示例的其余部分将使用 `/path/to/toolchain` 引用工具链目录结构的顶部。

树莓派工具链文件可能是这样的：

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR ARM)

set(CMAKE_C_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /path/to/toolchain/bin/armv8-rpi3-linux-gnueabihf-g++)

set(CMAKE_SYSROOT /path/to/toolchain/armv8-rpi3-linux-gnueabihf/sysroot)
```

如果主机有用于运行的挂载点，可以用于测试项目构建的二进制文件。例如，假设 `/mnt/rpiStage` 是一个附加到正在运行的Raspberry Pi的挂载点(最好指向某个本地目录，而不是系统根目录，这样就可以以任意方式删除或修改它，而不会破坏正在运行的系统)。工具链文件会将此挂载点指定为一个暂存区域，如下所示：

```
set(CMAKE_STAGING_PREFIX /mnt/rpiStage)
```

项目的二进制文件可以安装到这个区域，并直接在设备上运行(参见25.1.2节)。

### 21.6.2. GCC在64位平台上构建32位目标

GCC通过在编译器和链接器命令中添加 `-m32` 标志，允许在64位主机上构建32位二进制文件。下面的工具链示例允许在路径中找到GCC编译器，只在编译器和链接器使用的初始设置中添加额外的标记。个人看来，这种安排可以视为交叉编译。因此，也可以设置 `CMAKE_SYSTEM_NAME`，因为设置它会强制 `CMAKE_CROSSCOMPILING` 的值为`true`。无论哪种方式，都应该设置 `CMAKE_SYSTEM_PROCESSOR`，因为这个工具链文件的目标是专门针对与主机不同的处理器。

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR i686)

set(CMAKE_C_COMPILER gcc)
set(CMAKE_CXX_COMPILER g++)

set(CMAKE_C_FLAGS_INIT -m32)
set(CMAKE_CXX_FLAGS_INIT -m32)

set(CMAKE_EXE_LINKER_FLAGS_INIT -m32)
set(CMAKE_SHARED_LINKER_FLAGS_INIT -m32)
set(CMAKE_MODULE_LINKER_FLAGS_INIT -m32)
```

确认构建是32位的一种方法是使用 `CMAKE_SIZEOF_VOID_P` 变量，该变量由CMake自动计算，作为其工具链设置的一部分。对于64位构建，它的值为8，而对于32位构建，它的值为4。

```
math(EXPR bitness "${CMAKE_SIZEOF_VOID_P} * 8")
message("${bitness}-bit build")
```

### 21.6.3. 安卓 Android

Android的交叉编译可能比目前介绍的基本情况更加复杂，并且在如何描述目标系统方面存在一些差异。`CMAKE_SYSTEM_NAME` 必须设置为Android，但通常不设置 `CMAKE_SYSTEM_PROCESSOR`，`CMAKE_SYSTEM_VERSION` 留给CMake来确定。不需要设置单个编译器和工具的路径，而是由一些Android变量控制工具链配置。使用的CMake生成器的类型也会影响可用选项，因为不同的生成器支持不同的开发环境。例如，当使用Visual Studio生成器时，CMake要求安装Nvidia Nsight Tegra Visual Studio Edition。另一方面，使用Ninja或Makefile生成器可以让开发者在使用Android NDK或独立工具链之间做出选择。

#### NDK和独立工具链

使用Ninja或Makefile生成器时，CMake会来决定是使用NDK还是独立的工具链。这些步骤在CMake工具链文档中有清楚的详细说明，但它可以帮助进一步分解这些步骤(使用第一个匹配)：

#### 指定开发环境

- 如果设置了 `CMAKE_ANDROID_NDK` 变量，则会使用该位置的NDK。

- 如果设置了 `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` 变量，将使用该位置的独立工具链。这个位置必须有一个 `sysroot` 子目录。

#### 设置 `CMAKE_SYSROOT`

- 如果 `CMAKE_SYSROOT` 设置为 `<ndk>/platforms/android-<api>/arch-<arch>`，就好像 `CMAKE_ANDROID_NDK` 置为该路径的 `<ndk>` 部分。如果工具链文件没有显式提供，默认的Android API级别将填充路径的 `<api>` 部分(见下面)。
- 如果 `CMAKE_SYSROOT` 设置为 `<somedir>/sysroot` 形式的目录，就好像 `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` 设置为 `<someDir>`。

#### 替代 `CMake` 变量

- 如果 `ANDROID_NDK` 设置了，会当作 `CMAKE_ANDROID_NDK`。新项目不应依赖于这个变量，而应该直接使用更规范的 `CMAKE_ANDROID_NDK` 变量。
- 类似地，如果 `ANDROID_STANDALONE_TOOLCHAIN` 设置了，会当作 `CMAKE_ANDROID_STANDALONE_TOOLCHAIN`。新的项目不应该依赖于这个变量，而应该直接使用更规范的 `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` 变量。

#### 环境变量

- 如果设置了 `ANDROID_NDK_ROOT` 或 `ANDROID_NDK` 环境变量，会作为 `CMAKE_ANDROID_NDK` 变量的值。
- 如果设置了 `ANDROID_STANDALONE_TOOLCHAIN` 环境变量，会作为 `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` 变量的值。

NDK为开发者提供了(比独立工具链)更多的灵活性。独立的工具链只针对单一的架构和API级别，而NDK可能包含对多个工具链都支持，从而支持一系列的架构、API级别等等。请注意，根据NDK路线图显示，在r19发布前后，独立工具链正在被淘汰，除了Clang工具链和一个STL实现之外，所有的工具链都将移除。以下是对NDK和独立工具链相关的变量的介绍：

#### `CMAKE_SYSTEM_VERSION`

使用NDK时，可以将其设置为Android API级别，也可以由CMake填充。当未设置时，CMake首先检查是否设置了 `CMAKE_ANDROID_API` 变量，如果可用就使用。否则，如果设置了 `CMAKE_SYSROOT`，CMake将尝试从NDK目录结构中检测API级别。如果再失败，将使用NDK支持的最新API级别。对于独立的工具链，`CMAKE_SYSTEM_VERSION` 的值总是由工具链确定。

#### `CMAKE_ANDROID_ARCH_ABI`

这个变量指定了Android ABI。对于NDK构建，如果没有设置，对于r16以下的NDK版本，默认为armeabi，或者对于以后的版本，默认为老的arm ABI。`CMAKE_ANDROID_ARCH_ABI` 可以赋予其他值，只要NDK有必要的架构支持(例如：arm64-v8a, armeabi-v7a, armeabi-v6, mips, mips64, x86或x86\_64)。这个变量在使用独立工具链时自动设置。`CMAKE_ANDROID_ARCH` 将有 `CMAKE_ANDROID_ARCH_ABI` 替代，从而提供相对更为通用的架构值：arm、arm64、mips、mips64、x86或x86\_64。

#### `CMAKE_ANDROID_ARM_MODE`

当 `CMAKE_ANDROID_ARCH_ABI` 设置为armeabi\*架构时，开发人员可以在32位ARM或16位Thumb处理器之间进行选择。如果 `CMAKE_ANDROID_ARM_MODE` 设置为true值，将选中ARM处理器，否则设置为false或完全不设置，Thumb将是目标处理器。这可以通过NDK或独立的工具链来设置。

#### `CMAKE_ANDROID_ARM_NEON`

当 `CMAKE_ANDROID_ARCH_ABI` 设置为armeabi-v7a时，`CMAKE_ANDROID_ARM_NEON` 可以设置为真值来启用NEON支持。这可以通过使用NDK或独立的工具链来设置。

#### `CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION`

这个NDK特定变量可以用来指定从NDK中选择的工具链。如果给定，值必须采用下列形式之一：

- X.Y - GCC 版本 X.Y
- clangX.Y - Clang 版本 X.Y
- clang - 最新可用的Clang版本

如果未设置此变量，则使用NDK中可用的最新GCC版本。请注意，NDK文档(r16)声明在NDK中不再支持GCC，并且NDK路线图计划在r18中完全删除GCC，因此强烈建议使用Clang工具链。

#### CMAKE\_ANDROID\_STL\_TYPE

使用独立的工具链时，通过给出其中一个支持的值来选择多种STL实现：

- none
- system
- gabi++\_static
- gabi++\_shared
- gnustl\_static
- gnustl\_shared
- c++\_static
- c++\_shared
- stlport\_static
- stlport\_shared

如果没有给出，默认值是gnustlstatic。但请注意，与gnustl STL实现紧密相连的GCC工具链在NDK r18中是不可用的，而且在旧的NDK中只支持C++11。stlport 实现甚至更旧、更原始，甚至不支持C++11。none选项根本不支持C++，system选项只有new和delete`操作，没有STL。

NDK r16文档指出，c++\_static 和 c++\_shared STL类型将是未来NDK版本中唯一可用的类型，NDK路线图显示这将出现在r18中。因此，建议项目使用一个c++\_\* STL实现(LLVM C++标准库实现)，并使用Clang工具链。

每个CMake目标都有自己的ANDROID\_STL\_TYPE 属性，CMAKE\_ANDROID\_STL\_TYPE 变量用于提供该属性的初始值。大多数情况下，整个构建过程中使用相同的STL，因此使用变量而不是设置单个目标属性可能更简单、更健壮。

NDK构建的工具链文件的最小示例如下：

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
```

使用带有最新GCC工具链的NDK中的API级别也最新。它将不支持neon的armeabi体系结构(Thumb处理器)，并将使用gnustl\_static STL实现。一个更现实的例子设置了更多的变量：

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 26) # API level
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_ABI arm64-v8a)
set(CMAKE_ANDROID_TOOLCHAIN_VERSION clang)
set(CMAKE_ANDROID_STL_TYPE c++_shared)
```

上面使用了最新的Clang工具链和一个共享的运行时STL，支持近期的C++标准。

相比之下，独立的工具链通常会非常简单，因为许多配置是由工具链本身决定的：

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_STANDALONE_TOOLCHAIN /path/to/android-toolchain)
```

某些工具可能强制使用内部工具链文件，这可能使开发人员更难指定上述设置。Android Studio就是个例子，它提供了自己的工具链文件，覆盖了CMake的大部分逻辑。`gradle`构建用于创建一个外部CMake构建，使用Ninja生成器和Android SDK管理器提供NDK。虽然对工具链文件的直接访问是不启用的，但`gradle`构建确实提供了一系列的`gradle`变量，这些变量会翻译成CMake中的等价物。开发人员应该参考该工具的文档，以确定是否/如何使用不同的CMake版本，以及如何影响CMake构建的行为。这似乎是Android Studio的一个活跃开发领域。

对于使用`ndk-build`而不是`gradle`的开发者来说，CMake 3.7引入了导出Android的能力。可以使用`export()`作为CMake构建的一部分，也可以使用`install()`作为安装步骤的一部分。构建期间的导出非常简单：

```
export(TARGETS target1 [target2...] ANDROID_MK fileName)
```

文件名通常为带路径的`Android.mk`，把它放到`ndk-build`需要的位置。每个命名的目标都将包括在生成的文件中，以及相关的使用要求，如包含标志、编译器定义等。如果项目需要支持成为父`ndk`构建的一部分，这就是它通常想要做的。如果CMake要打包项目，并且想让自己更容易合并到任何`ndk`构建中，`install()`提供了所需的功能(参见25.3节)。

### Visual Studio生成器

使用Visual Studio生成器时，CMake要求安装Nvidia Nsight Tegra Visual Studio Edition。最终的项目将驱动整个构建，而不是形成一个更大的`gradle`或`ndk`构建结构的一部分。支持是在CMake 3.1中首次添加的，但是很多选项直到CMake 3.4才添加。生成器通常在CMake上设置如下命令行：

```
cmake -G "Visual Studio 12 2013 Tegra-Android" \
-DCMAKE_TOOLCHAIN_FILE=/some/path/toolchain.cmake \
/path/to/source
```

最小的工具链文件只需要将`CMAKE_SYSTEM_NAME`设置为Android，就像NDK和独立的工具链一样，可以设置更多的变量来影响目标架构等等。在很多情况下，为Visual Studio构建设置的变量与NDK情况不同，但通常是相关的。

NDK和独立的工具链构建会设置`CMAKE_ANDROID_ABI`并允许`CMAKE_ANDROID_ARCH`的派生，Visual Studio的工具链文件会直接设置`CMAKE_ANDROID_ARCH`。Visual Studio情况下的允许值也不同：`armv7-a`、`armv7-a-hard`、`arm64-v8a`、`x86`和`x86_64`。

Visual Studio构建的工具链文件将设置`CMAKE_ANDROID_API`，而不是使用`CMAKE_SYSTEM_VERSION`来指定目标设备的Android API级别，而`CMAKE_ANDROID_API`作为`ANDROID_API`目标属性的默认值。此外，可以设置`CMAKE_ANDROID_API_MIN`来指定用于构建本机代码的API版本(遵循相同的模式，并充当`ANDROID_API_MIN`目标属性的默认值)。这有点类似于Apple平台的情况，其中用于构建的SDK可以单独指定到目标设备的最低操作系统级别(参见22.5节)。

`CMAKE_ANDROID_STL_TYPE`变量可以设置并接受NDK的值，但是不支持`c++_static`和`c++_shared`。它会用作`ANDROID_STL_TYPE`目标属性的默认值。

由于这种方式驱动了整个构建，所以必须设置比CMake构建本地代码更多的内容。还有许多其他目标属性与构建原生代码没有关联的构建部分相关，比如：JAR依赖项、Java源的设置等。其中一些目标属性还与CMake变量的默认值的定义相关。这些目标属性都有`ANDROID_...`的名称，而CMake的默认变量有`CMAKE_ANDROID_...`。这些信息超出了本文的讨论范围，因此感兴趣的读者可以参考CMake文档来了解支持的属性和变量的详细信息，然后根据项目的非原生部分来设置它们。

## 21.7. 总结

工具链文件乍一看有点吓人，但这大多来自于其中放置了太多逻辑和许多示例和项目。工具链文件应该尽可能的少，以支持所需的工具，并且它们通常应该在不同的项目之间重用。特定于项目的逻辑应该使用在项目自己的CMakeLists.txt文件中。

编写工具链文件时，开发人员应该确保内容不会只会执行一次。CMake可能会处理工具链文件多次，这取决于项目做了什么(例如：多次调用 `project()` 或 `enable_language()` )。工具链文件也可以用于临时构建，作为 `try_compile()` 的一部分，所以不应该对使用上下文做任何假设。

避免使用已弃用的 `CMakeForceCompiler` 模块在构建中设置编译器。这个模块在使用旧的CMake版本时很流行，但新版本严重依赖于测试工具链，并会找出它所支持的特性。`CMakeForceCompiler` 模块主要用于CMake未知的编译器，但是在最近的CMake版本中使用这样的编译器很可能会带来不小的限制。建议与CMake开发人员合作，为此类编译器添加所需的支持。

注意，不要丢弃或错误处理，可能在处理工具链文件时已经设置的变量的内容。常见的错误是修改像 `CMAKE_<LANG>_FLAGS`，而不是 `CMAKE_<LANG>_FLAGS_INIT` 变量，这可能会丢弃开发人员手动设置的值，或者多次处理工具链文件时与需要的值不同。

以Android平台为目标时，最好使用带有NDK和Ninja或Makefile生成器的简单工具链文件。这种组合CMake支持的，并且最容易使用。工具链文件可以非常简单，而且最近版本的IDE工具(如Android Studio)也开始使用这种方法。当开发人员使用他们自己的工具链文件时，不要使用流行的taka-no-me工具链文件，因为它过于复杂并且存在很多已知的问题。较新的CMake版本支持简单的工具链文件，这些工具链文件工作起来很流畅，而且工作量很小。

项目应该避免对任何逻辑使用 `CMAKE_CROSSCOMPILING` 变量。这个变量可能会引起误解，即使目标和主机平台相同，也可以将其设置为`true`，或者在目标和主机平台不同时设置为`false`。项目作者应该意识到，一些多配置生成器(例如Xcode)允许在建造时选择目标平台，所以CMake逻辑基于是否交叉编译，需要仔细地处理(该项目可能产生的)不同情况。

工具链文件通常包含命令可以用来修改CMake搜索程序、库和其他文件的位置。参见第23章，寻找与此领域相关的实践。

# 第22章：Apple专属特性

Apple平台有许多特点，这些特点直接影响软件的构建方式。尽管macOS的命令行应用程序可以与基于Unix的平台类似的方式构建，但具有图形用户界面的应用程序通常以特定的格式提供给Apple，称为应用程序包(application bundle，或简称为app bundle)。这些包不仅仅是可执行文件，它们是一个标准化的目录结构，包含与应用程序关联的各种文件。这些应用程序包是自包含的，可以够作为一个单元移动，可以放置在用户文件系统的任何位置。

库的情况也类似。独立的静态和动态库可以像基于Unix的平台上的库一样创建，但也可以作为框架的一部分构建，这在本质上相当于应用程序包的库。框架有自己的标准化目录结构，可包含库二进制文件以外的文件。甚至可以在该目录结构中支持多个版本。在运行时加载的库可以构建为可加载的包，这与Apple的CFBundle功能相对应。

捆绑包和框架是为Apple应用商店生成内容的重要组成部分。另一个关键方面是代码签名，这是验证软件完整性和来源的过程，是App Store发行的强制性标准。代码授权也是构建过程中不可或缺的部分，它控制着代码特性。这些授权是代码签名过程密封的信息，如果默认授权集(为空)不合适，则必须在构建时定义。

总之，这些特性给CMake项目带来了挑战。接下来的部分提供了用于理解和处理这些问题的工具，突出了CMake当前的局限性。虽然CMake正式支持macOS和iOS，但对tvOS和watchOS的支持还不完善。

## 22.1. CMake选择生成器

用于生成框架和捆绑包的技术和工具在不断发展，Apple操作系统发布版本经常引入新特性，并围绕签名、发布等改变需求。这些流程和技术会集成到Xcode中，作为Apple希望开发人员使用的主要工具，开发人员通常希望升级到当前的Xcode版本，而不是停留在过去的版本中。资源编译、代码签名等领域是作为构建应用程序和框架会自动处理的部分，其中许多特性是Apple生态所独有的。

对于CMake项目，这意味着Xcode生成器是使用Xcode工具链构建的最可靠、最方便的工具。其他生成器，如makefile或Ninja，往往缺乏Xcode生成器的一些自动化功能，或者对一些Xcode特性的支持方面比较落后。除了不打算通过App Store发布未签名的桌面应用程序外，开发人员也会需要使用Xcode生成器支持构建。还要注意Apple平台快速变化的特性，开发人员通常希望CMake版本能跟上变化。

Xcode生成器的独特优势之一是它支持设置任意的Xcode项目属性。大多数项目设置都可以通过使用`XCODE_ATTRIBUTE_XXX`格式的目标属性(其中`xxx`是一个Xcode属性的名称)对每个目标进行修改。这些名称在Apple文档中定义，但找到它们更方便的方法是打开Xcode项目，转到目标的构建设置，并单击感兴趣的构建设置。Quick Help助手编辑器会显示设置名称和描述。所有目标的默认值都可以通过`CMAKE_XCODE_ATTRIBUTE_XXX`设置。定义目标时，使用这些变量对的目标属性进行初始化。下面的例子演示了如何设置一些常用属性：

```
# Set the default signing identity and team ID to use for all targets
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer"
set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM XYZ123ABCD)

# Some target-specific settings
set_target_properties(myiOSApp PROPERTIES
  XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 1,2
  XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0
)
```

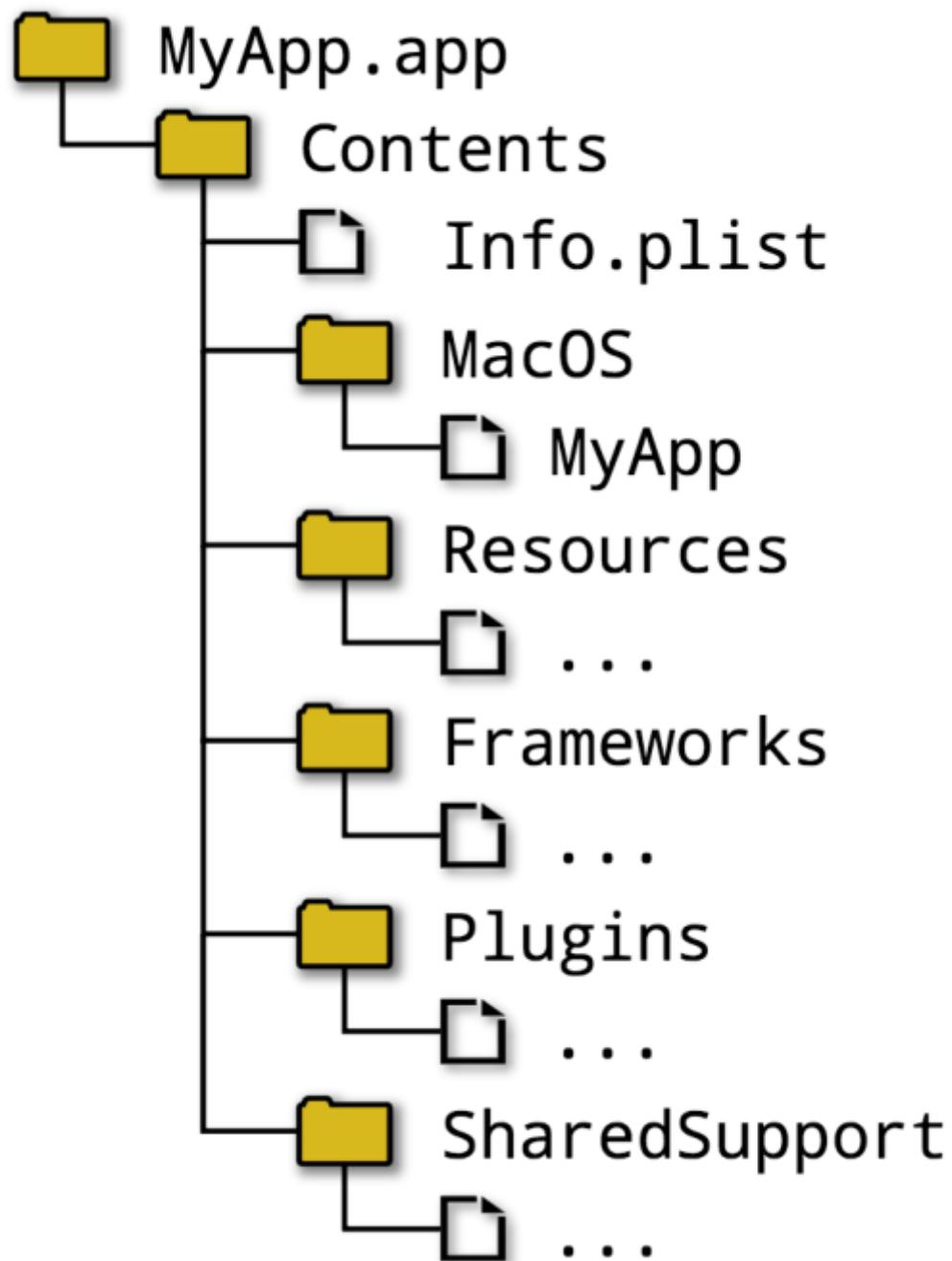
该特性还可以用于为特定构建类型设置**Xcode**属性，方法是在属性名后面附加 `[variant=ConfigName]`。对于更具体的属性设置，也可以将其他后缀类型附加到属性名中，但是并不常见。甚至 `[variant=...]` 后缀也不太需要。下面的例子给出了这个特性可用的用例：

```
set_target_properties(myiOSApp PROPERTIES
    XCODE_ATTRIBUTE_GCC_UNROLL_LOOPS[variant=Release] YES
    XCODE_ATTRIBUTE_ENABLE_TESTABILITY[variant=Debug] YES
)
```

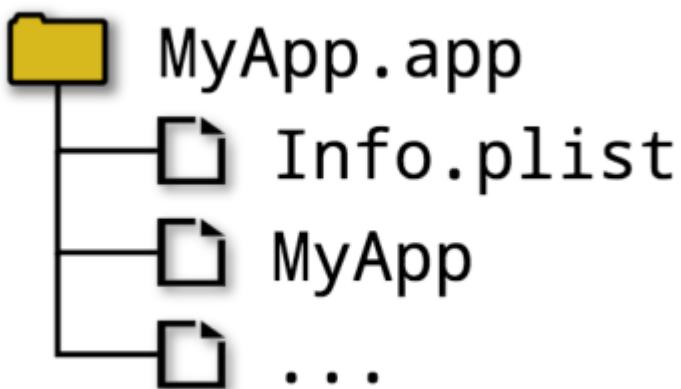
有些项目可能需要设置相当多的属性来获得所需的**Xcode**行为和特性，而其他项目可能非常简单，只需要很少的设置。有些属性只在非常特定的环境中需要，而其他属性则非常常见，它们几乎(或者应该)出现在每个 Apple平台的项目中。本章的其余部分将讨论其中的一些属性，包括上面例子中使用的属性。

## 22.2. 应用程序包

macOS的应用程序包结构与iOS、tvOS和watchOS的不同。macOS结构中不同类别的文件在不同的子目录中，看起来像以下结构(应用程序可能只有一些子目录显示):



相比之下，iOS、tvOS和watchOS的目录结构是扁平的，几乎没有定义结构：



构建App Bundle时，CMake在某种程度上抽象了这些结构上的差异，不管这个Bundle是为macOS、iOS、tvOS还是watchOS构建的，至少有一些可以用相同的方式处理。然而，开发人员应该意识到，直到最近的CMake版本(资源的处理是一个特定的例子)，这种抽象才能正确实现，所以强烈建议使用最新版的CMake。

通过向 `add_executable()` 添加 `MACOSX_BUNDLE` 关键字，会将应用程序识别为一个Bundle：

```
add_executable(myApp MACOSX_BUNDLE ...)
```

这将 `MACOSX_BUNDLE` 目标属性设置为`true`，非Apple平台会直接忽略该属性。一个项目可以将 `CMAKE_MACOSX_BUNDLE` 变量设置为`true`，并且随后定义可执行目标具有 `MACOSX_BUNDLE` 属性，可以说是清楚的描述了 `add_executable()` 使用 `MACOSX_BUNDLE` 关键字的特点(项目通常定义了小部分应用程序包，并且只有一个)。

`MACOSX_BUNDLE` 不仅适用于macOS，还适用于iOS、tvOS和watchOS。关键字先于非桌面Apple平台，因此有桌面专用的名称。而不是为其他平台创建新的关键字，现有关键字的使用扩大到覆盖所有Apple平台。这种扩展特定于OSX的关键字和变量，可以覆盖所有Apple平台的模式。其他情况下也可以看到，但这种模式并不适用于所有与OSX相关的变量和属性。

每个应用程序包至少有一个`Info.plist`文件和一个可执行文件(`MyApp`在上面的目录结构例子)。默认情况下，CMake将从一个模板文件中提供`Info.plist`文件。然而，大多数情况下，项目希望提供自己这个文件，这样就可以完全控制应用程序的配置。当应用程序使用故事板或界面构建器文件时，必须提供自定义`Info.plist`，这样相关的关键条目就会出现，比如 `NSMainStoryboardFile`。`MACOSX_BUNDLE_INFO_PLIST` 目标属性可以设置为`Info.plist`的文件模板(适用于所有Apple平台，不只是macOS)。默认模板文件名为 `MacOSXBundleInfo.plist.in`，并且可以在CMake的模块目录中找到。它可以作为定制模板的起点

不管目标是否使用默认`Info.plist`模板，CMake会将模板文件复制到App Bundle中，并在此过程中执行一些特定的替换。模板文件中，如果 `xxx` 是下表中的属性之一，则 `$(xxx)` 的任何内容都将使用 `xxx` 目标属性的值所替代。这些属性会在默认`Info.plist`中映射特定键值。如果项目提供了自己的模板文件，并使用了这些变量，通常应该遵循相同的映射方式。

属性	Info.plist键	例子
MACOSX_BUNDLE_NAME	CFBundleName	MyApp
MACOSX_BUNDLE_VERSION	CFBundleVersion	2.4.7rc1
MACOSX_BUNDLE_COPYRIGHT	NSHumanReadableCopyright	© 2018 MyCom
MACOSX_BUNDLE_GUI_IDENTIFIER	CFBundleIdentifier	com.example.myapp
MACOSX_BUNDLE_SHORT_VERSION_STRING	CFBundleShortVersionString	2.4.7
MACOSX_BUNDLE_LONG_VERSION_STRING	CFBundleLongVersionString	see below
MACOSX_BUNDLE_INFO_STRING	CFBundleGetInfoString	see below
MACOSX_BUNDLE_ICON_FILE	CFBundleIconFile	see below

Apple不再将 `CFBundleLongVersionString` 作为Info.plist的键值，所以项目可以选择不提供它。文档还声明 `NSHumanReadableCopyright` 已经取代了 `CFBundleGetInfoString`，并且 `CFBundleIconFile` 已弃用，建议使用 `CFBundleIconFiles` 或 `CFBundleIcons` 代替。如果没有设置其他选项，仍可以使用 `CFBundleIconFile`。

如果定义了多个App目标，项目可以设置与上表中的属性名称完全相同的变量，这些变量用于初始化目标属性。注意，这不同于通常的CMake约定，即在变量作为默认值的目标属性之前具有 `CMAKE_` 前缀。

当项目提供自己的 Info.plist 模板文件，不需要使用上面的目标属性。而且，硬编码是有效的。但请注意，`CFBundleVersion` 和 `CFBundleShortVersionString` 可能需要从 CMakeLists.txt 文件中指定的版本信息，因此通过 `MACOSX_BUNDLE_NAME` 和 `MACOSX_BUNDLE_SHORT_VERSION_STRING` 替换设置仍然是最方便的方法。随着时间的推移，Apple 对版本号的要求也在不断变化，现在基本上是强制性的 `major.minor.patch` 格式(有些例外)。下面展示了一种提供满足 Apple 要求的版本号的映射：

```
add_executable(myApp MACOSX_BUNDLE ...)
set_target_properties(myApp PROPERTIES
  MACOSX_BUNDLE_NAME "${PROJECT_NAME}"
  MACOSX_BUNDLE_VERSION "${PROJECT_VERSION}${BUILD_SUFFIX}"
  MACOSX_BUNDLE_SHORT_VERSION_STRING "${PROJECT_VERSION}"
)
```

上面的例子中，`BUILD_SUFFIX` 在最终将是一个空字符串，或者可以是一个或多个字母后跟一个1-255的数字。示例后缀可能是用于alpha版本的a17，或用于第二个候选版本的rc2等等，Info.plist文件将用于进一步包含使用的这些属性。

用了合适的Info.plist文件后，可以将注意力转向要编译和链接到包中的源文件。通常除了C/C++源文件，Apple平台还支持Objective C/C++源文件。通常具有 `.m` 或 `.mm` 后缀，可以在 `add_executable()` 和 `target_sources()` 中作为源文件，就像普通的C/C++文件一样。大多数CMake的生成器将识别这些文件后缀并编译文件(不仅仅是Xcode生成器)。

Apple平台特有的另一组源文件是用于定义用户界面的源文件。故事板或接口构建器文件类似于源文件，但需要一些额外的处理将它们编译为资源，并将编译后的结果放在App Bundle中适当的位置。Xcode生成器实现了这种自动编译，并将其复制到适当的位置，所以当应用程序包中有这些文件时，不推荐使用Makefile或Ninja生成器。故事板和接口构建器源需要作为 `add_executable()` 或 `target_sources()` 中的源。为了自动编译并复制到Bundle中的适当位置，还需要对 `RESOURCE` 目标属性进行设置。例如：

```

set(uiFiles
    Base.lproj/Main.storyboard
    Base.lproj/LaunchScreen.storyboard
)

add_executable(MyApp MACOSX_BUNDLE
    AppDelegate.m
    AppDelegate.h
    ViewController.m
    ViewController.h
    main.m
    ${uiFiles}
)

set_target_properties(MyApp PROPERTIES
    RESOURCE "${uiFiles}"
    MACOSX_BUNDLE_INFO_PLIST "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)

```

注意使用 `uiFiles` 变量处理接口构建的方式。这个变量值可以在 `add_executable()` 的源列表中不加引号使用，这使得接口构建文件仅作为源文件中的项出现。另一方面，`RESOURCE` 目标属性持有一个值，该值是以分号分隔的列表。因此，`RESOURCE` 属性要求引用 `uiFiles` 变量的值，而 `add_executable()` 不引用。

上面的例子中，`Info.plist` 文件包含键 `NSMainStoryboardFile`、`NSMainNibFile` 或 `UIStoryboardFile`（关于这些键的含义和使用信息，请参阅[Apple官方文档](#)），这些信息会告诉操作系统在启动应用程序时使用哪个UI元素。一个简单的`Info.plist`看起来像是这样：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>CFBundleDevelopmentRegion</key>
<string>en</string>
<key>CFBundleExecutable</key>
<string>$(EXECUTABLE_NAME)</string>
<key>CFBundleIconFile</key>
<string>${MACOSX_BUNDLE_ICON_FILE}</string>
<key>CFBundleIdentifier</key>
<string>${MACOSX_BUNDLE_GUI_IDENTIFIER}</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>${MACOSX_BUNDLE_NAME}</string>
<key>CFBundlePackageType</key>
<string>APPL</string>
<key>CFBundleShortVersionString</key>
<string>${MACOSX_BUNDLE_SHORT_VERSION_STRING}</string>
<key>CFBundleVersion</key>
<string>${MACOSX_BUNDLE_VERSION}</string>
<key>LSMinimumSystemVersion</key>
<string>$(MACOSX_DEPLOYMENT_TARGET)</string>
<key>NSHumanReadableCopyright</key>
<string>${MACOSX_COPYRIGHT}</string>
<key>NSMainStoryboardFile</key>
<string>Main</string>
<key>NSPrincipalClass</key>
<string>NSApplication</string>
</dict>
</plist>

```

在上面的例子中，`NSMainStoryboardFile` 字段的值是 `Main`，它指定 `Base.lproj/Main.storyboard UI` 将在应用程序启动时使用。还要注意一些字段值是如何使用 `{}$` 语法作为CMake变量的，`CFBundleExecutable` 和 `LSMinimumSystemVersion` 是使用Xcode变量替换 `$( )` 提供的。这两个字段由Xcode本身根据项目文件和正在构建的方案中的其他信息填充。`LSMinimumSystemVersion` 的值来自macOS的 `CMAKE OSX_DEPLOYMENT_TARGET` 变量，或为iOS目标属性设置的 `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` 变量(注意，CMake 3.11之后可以在所有平台使用 `CMAKE OSX_DEPLOYMENT_TARGET` )。如果方便的话，项目可以直接硬编码Info.plist文件。

包含在应用程序包的通常不是资源文件，这些文件仍然作为目标源列出，但不是将它们包含在资源目标属性中，而是将每个源的 `MACOSX_PACKAGE_LOCATION` 属性设置为在Bundle中(复制到)的位置。这些路径相对于Bundle顶部。这可将文件复制到非资源位置，或完全控制不需要编译的资源文件目标。也可以列出一个目录作为源，并设置 `MACOSX_PACKAGE_LOCATION` 将目录及其内容复制到包中，但CMake是否支持这一点没有文档说明(目录通常不能作为源)。下面的例子可以演示这些行为：

```
add_executable(MyApp MACOSX_BUNDLE
  AppDelegate.m
  AppDelegate.h
  ViewController.m
  ViewController.h
  main.m
  sharedConfig.xml
  nestedResource.dat
  someDir # Directory, CMake may not formally support this
)

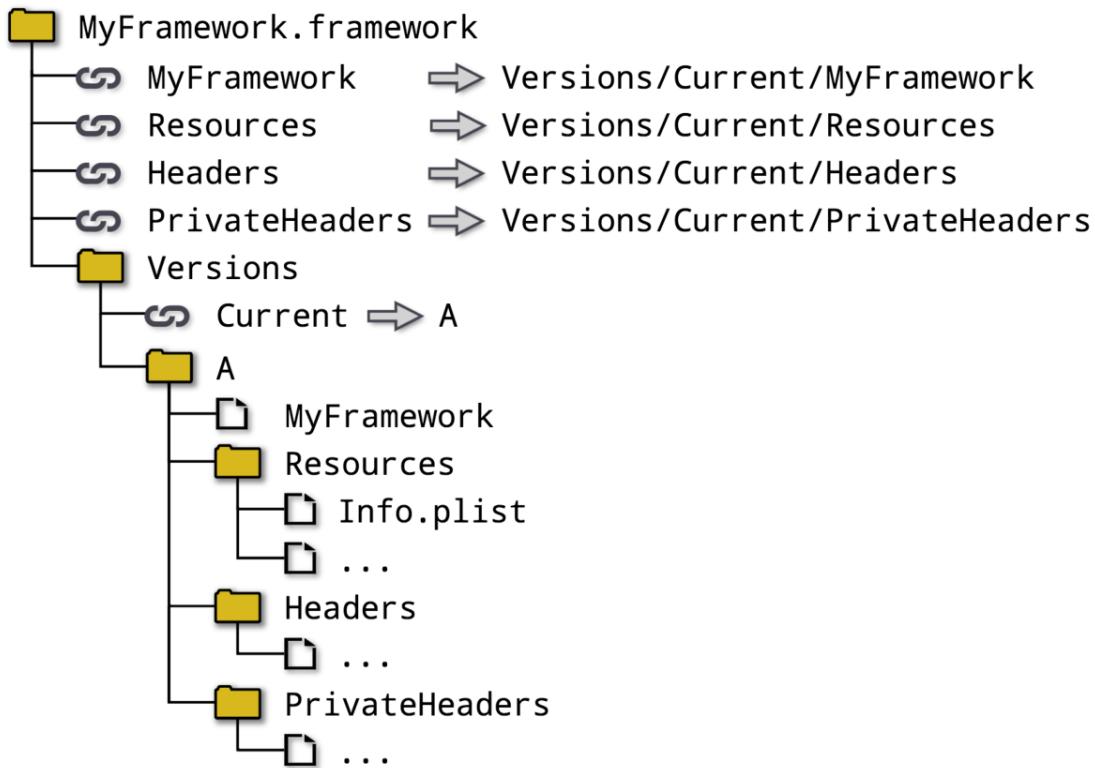
set_source_files_properties(sharedConfig.xml PROPERTIES
  MACOSX_PACKAGE_LOCATION SharedSupport/config
)
set_source_files_properties(nestedResource.dat PROPERTIES
  MACOSX_PACKAGE_LOCATION Resources/private/other
)

# Works, but might not be formally supported
set_source_files_properties(someDir PROPERTIES
  MACOSX_PACKAGE_LOCATION Resources/lotsOfThings
)
```

将 `MACOSX_PACKAGE_LOCATION` 设置为资源的起始路径，并且构建iOS目标时，会出现一种特殊情况。因为iOS应用程序包使用扁平化结构，CMake将剥离路径中的资源部分。CMake 3.9之前，这种行为实现是不正确的，并不是总能将文件放到想要的位置。

## 22.3. 框架

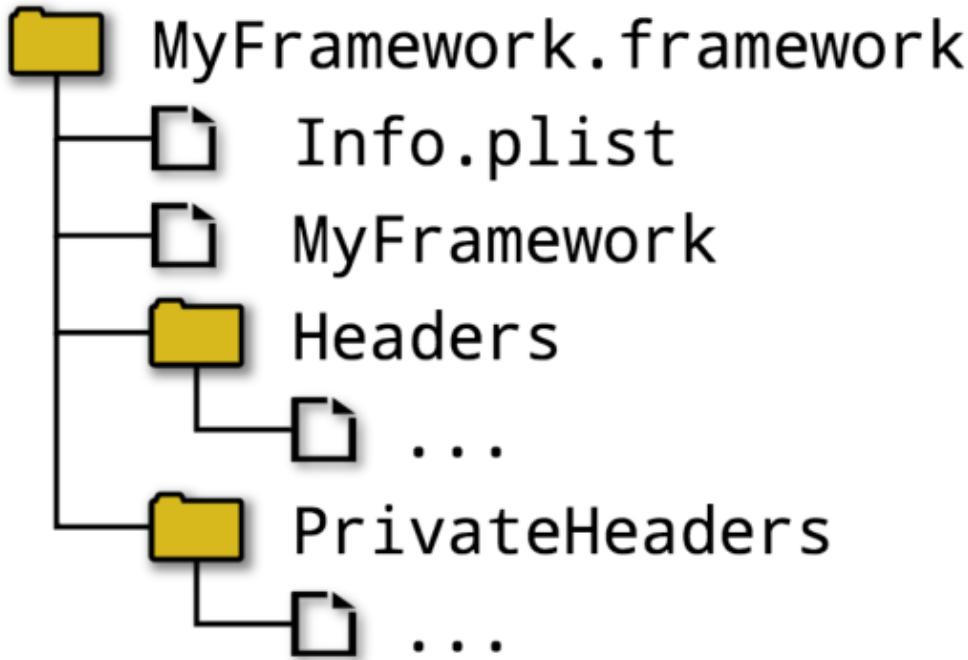
框架与应用程序包有一些相似之处，框架包含一个主库，但与应用程序包不同，macOS上可能有多个版本的库。除了资源之外，框架还支持头文件，在macOS中，资源和头文件都是特定于版本的。macOS的典型框架结构示例如下所示：



框架的顶层总有一个以 `.framework` 结尾的文件，通常该顶层目录中唯一的非符号链接是 `Versions` 子目录(伞形框架是例外，不在这里的框架支持范围)。该级别的其他内容通常指向当前版本子目录的符号链接。

版本目录中，库的每个版本都有自己的子目录。大多数情况下，这些目录名只是 `A`、`B` 等。数字版本使用的是另一种常见约定，它与动态库的版本控制方式结合在一起。不管版本控制的风格如何，名为 `Current` 的符号链接都指向最新的版本，它的作用类似于框架的默认版本。每个版本都有一个资源目录，其中至少包含一个 `Info.plist` 文件，提供关于特定版本的配置信息(后面将进一步讨论)。还会有一个库(通常是动态库，也可以是静态的)，还有公共头文件目录和私有头文件子目录。

相比之下，iOS、tvOS和watchOS上的结构是扁平的，通常不支持多版本：



**CMake**支持创建框架(macOS只支持单一版本)，并提供处理版本信息的特性。也支持 `Info.plist`与应用程序包使用的方法类似。第一步是按照常规方式定义一个库，然后通过设置框架属性标记为框架。大多数框架定义为动态库。在**CMake 3.8**中，静态库也可以构建为框架，框架属性在非Apple平台上可以忽略。仅对macOS而言，框架版本可以使用 `FRAMEWORK_VERSION` 属性指定，或者设置为默认版本。非macOS平台将忽略 `FRAMEWORK_VERSION` 属性，如果设置了，产出文件结构同样是扁平的，**Xcode**会为平台创建框架产生未版本化的框架结构。

```

add_library(MyFramework SHARED foo.cpp)
set_target_properties(MyFramework PROPERTIES
    FRAMEWORK TRUE
    FRAMEWORK_VERSION 5
)

```

`Info.plist`文件模板指定的方式与应用程序包相同，除了属性为 `MACOSX_FRAMEWORK_INFO_PLIST` (支持所有Apple平台，不仅仅是macOS):

```

set_target_properties(MyFramework PROPERTIES
    MACOSX_FRAMEWORK_INFO_PLIST "${CMAKE_CURRENT_SOURCE_DIR}/Info.plist"
)

```

至于应用程序包，如果框架没有显式提供`Info.plist`文件，该文件会自动生成默认的。项目是否提供自己`Info.plist`或依赖于默认生成的，**CMake**在将应用程序捆绑复制到框架中时，执行类似的替换。以下属性将替换为`Info.plist`文件的引用(信息期望的关联键值，也在`Info.plist`文件中列出):

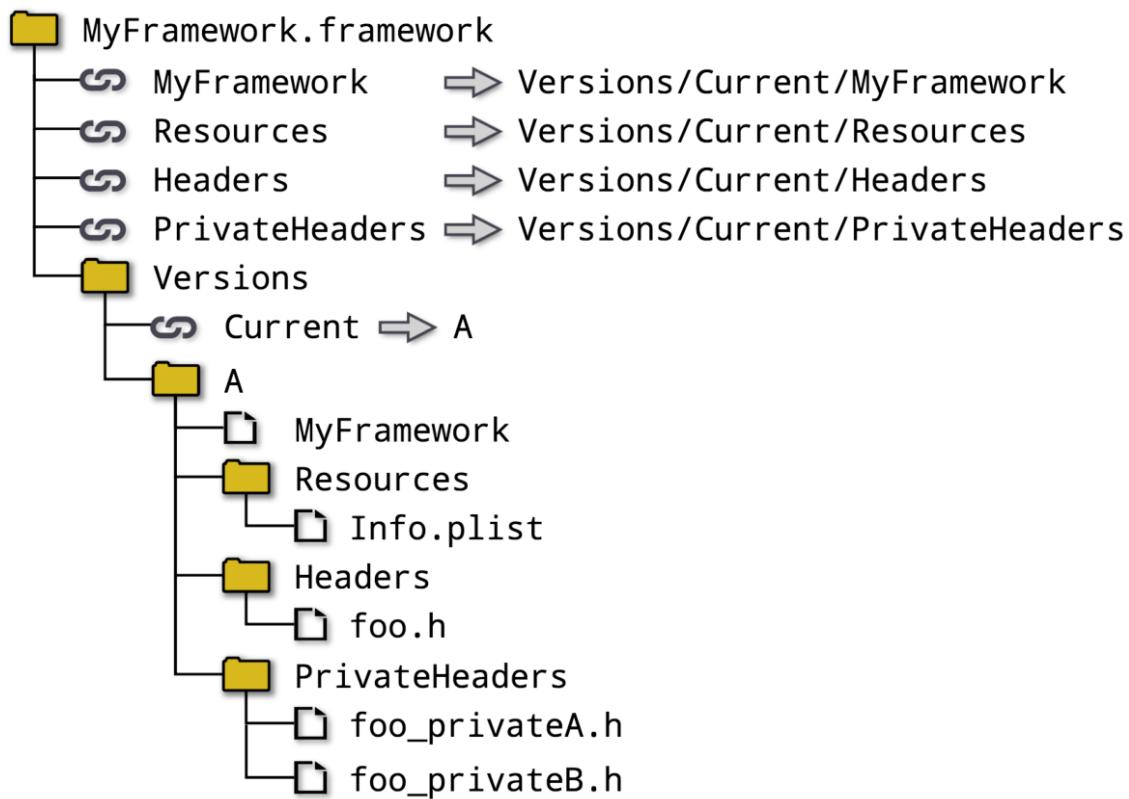
属性	Info.plist Key
MACOSX_FRAMEWORK_BUNDLE_VERSION	CFBundleVersion
MACOSX_FRAMEWORK_ICON_FILE	CFBundleIconFile
MACOSX_FRAMEWORK_IDENTIFIER	CFBundleIdentifier
MACOSX_FRAMEWORK_SHORT_VERSION_STRING	CFBundleShortVersionString

与应用程序包不同，默认的Info.plist在很多情况下，可能已经够用了，所以项目通常只需要设置上述四个属性，然后让CMake提供适当的Info.plist文件。框架通常包含与框架库相关联的头文件，这允许将框架视为自包含的包，其他软件可以根据它进行构建。

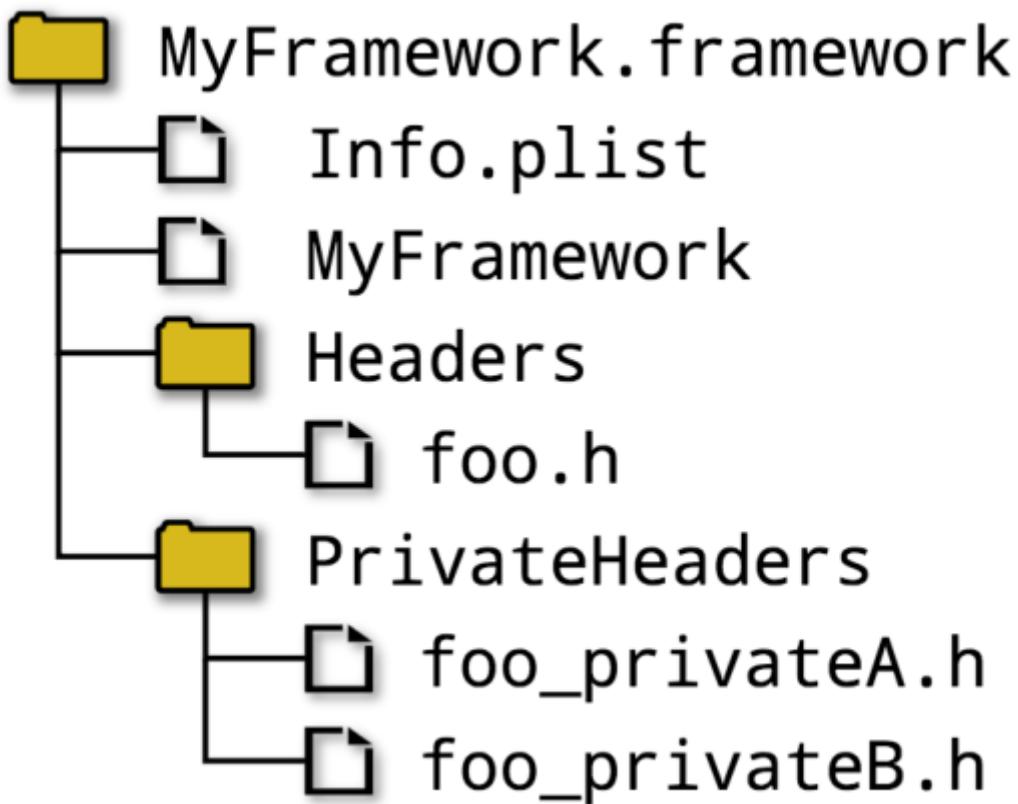
框架头文件划分为公共和私有，只有公共头文件可以直接包含或导入。私有头文件通常需要作为公共头的内部实现细节，而框架通常不包含任何私有头文件。CMake支持使用 PUBLIC\_HEADER 和 PRIVATE\_HEADER 属性指定公共和私有头文件。这两个属性都一个头文件列表，并且所有提到的文件也必须作为目标的源显式设置。PUBLIC\_HEADER 中列出的文件将复制到框架的 Headers 目录中，路径将会删除，而在 PRIVATE\_HEADER 中列出的文件将复制到 PrivateHeaders 目录中，路径也会删除。如果需要保留路径，则不能使用这些目标属性，而要通过使用 MACOSX\_PACKAGE\_LOCATION 来添加头文件。

```
add_library(MyFramework SHARED
foo.cpp
foo.h
foo_privateA.h
nested/foo_privateB.h
)
set_target_properties(MyFramework PROPERTIES
FRAMEWORK TRUE
PUBLIC_HEADER foo.h
PRIVATE_HEADER "foo_privateA.h;nested/foo_privateB.h"
)
```

上面的例子会在macOS上产生以下的目录结构:



在iOS上的相同例子会产生一个更扁平的结构:



注意，在非Apple平台上安装目标时，还要使用 `PUBLIC_HEADER` 和 `PRIVATE_HEADER` 属性。第25.2.3节中有更详细的介绍。

## 22.4. 可加载包

除了应用程序包和框架，Apple还支持macOS的可加载包。它们通常用作插件，或提供在运行时可能支持的可选特性。可加载包的结构与应用程序包的结构相同，但顶层目录的扩展名通常为 `.bundle` 或 `.plugin`。  
**CMake**通过 `MODULE` 库类型和 `BUNDLE` 目标属性，支持创建可加载的 `BUNDLE`。默认情况下，可加载包将放入扩展包中，也可以用 `BUNDLE_EXTENSION` 目标属性覆盖它。

```

add_library(MyBundle MODULE ...)
set_target_properties(MyBundle PROPERTIES
  BUNDLE TRUE
  BUNDLE_EXTENSION plugin
)

```

与应用程序包相关的所有属性，也可以用于可加载包。

## 22.5. 构建设置

为Apple平台构建项目时，许多属性统一决定为哪个平台构建，并为平台指定最低版本。**Xcode**生成器允许开发人员在构建时设置时间，这对于新手和有经验的**CMake**用户来说，都是一个比较难处理的特性。

对于单配置生成器，目标设备在配置时是已知的，但是对于Xcode同时支持设备和模拟器。此外，其中一些设备具有多个架构。iOS平台下，这可能意味着多达五种不同的目标平台组合。不同版本的Xcode附带不同版本的iOS SDK，一些开发人员甚至可能将旧的SDK移植到新的Xcode版本中。为了允许开发者在构建时在不同的目标设备和SDK之间切换，CMake项目必须正确地处理这些细节。

iOS、tvOS和watchOS SDK的选择，是许多在例中相当复杂的部分，经常让开发者重新运行CMake，从而在设备和模拟器之间进行切换构建。然而，对于最新版本的CMake和Xcode，指定SDK非常简单，只需将 CMAKE OSX\_SYSROOT 变量设置为某个iPhoneos、AppleTvOS或WatchOS即可。Xcode会为该平台选择最新的SDK，无需重新运行CMake，就可以在设备和模拟器构建之间切换构建。此外，Xcode会根据选的SDK自动填充架构支持，因此项目不需要添加任何额外的逻辑来指定架构。这让开发人员不需要重新运行CMake，就能对他们想要构建的内容进行控制。可用的SDK可以通过运行以下命令获得：

```
xcodebuild -showsdk
```

由于CMake执行编译器测试的方式，当对Apple平台而不是macOS时，需要设置缓存变量。CMake 3.12之前，代码签名会干扰编译器测试，这些测试并不使用正确的类型(例如，应该创建包而不创建)。要解决这些问题需要设置 CMAKE\_MACOSX\_BUNDLE 和 CMAKE\_XCODE\_ATTRIBUTE\_CODE\_SIGNING\_REQUIRED 变量。为了让编译器测试选择正确，CMAKE OSX\_SYSROOT 、 CMAKE\_MACOSX\_BUNDLE 和 CMAKE\_XCODE\_ATTRIBUTE\_CODE\_SIGNING\_REQUIRED 需要在配置阶段设置。最好使用工具链文件，类似这样：

```
set(CMAKE_MACOSX_BUNDLE YES)
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED NO)
set(CMAKE OSX_SYSROOT iphoneos)
```

默认情况下，项目将其部署目标设置为SDK或主机系统支持的部署目标。通常不会这样，因为项目通常希望能与特定的最小OS版本兼容。对于macOS，OSX\_DEPLOYMENT\_TARGET 属性将控制目标的最低macOS版本。可以使用 CMAKE OSX\_DEPLOYMENT\_TARGET 变量为这个属性指定默认值，需要在第一个使用 project() 之前设置。此外，如果CMakeLists.txt文件中直接设置了 CMAKE OSX\_DEPLOYMENT\_TARGET ，那么需要一个缓存变量，否则 project() 会在执行编译器检查时将其值覆盖。另一种策略是使用工具链文件，并在其中设置 CMAKE OSX\_DEPLOYMENT\_TARGET ，但是在macOS构建中使用工具链文件相当少见，这个变量应该由项目定义。还有一种方法是使用cmake命令行设置 CMAKE OSX\_DEPLOYMENT\_TARGET 缓存变量，但这也需要开发人员来设置，并且需要保证提供信息的正确性，所以降低了开发人员使用这种方式的积极性。

CMake 3.11之前，目标平台不是macOS时， CMAKE OSX\_DEPLOYMENT\_TARGET 变量没有效果。在CMake 3.11之前控制iOS的最小部署版本，可以使用 XCODE\_ATTRIBUTE\_IPHONEOS\_DEPLOYMENT\_TARGET 属性。这个目标属性的默认值可以使用 CMAKE\_XCODE\_ATTRIBUTE\_IPHONEOS\_DEPLOYMENT\_TARGET 变量来设置，与macOS不同的，这个变量可以在第一次 project() 调用后设置。从CMake 3.11开始， CMAKE OSX\_DEPLOYMENT\_TARGET 可以用来定义任何Apple平台的最小部署版本。如果目标同时定义了 OSX\_DEPLOYMENT\_TARGET 和 XCODE\_ATTRIBUTE\_IPHONEOS\_DEPLOYMENT\_TARGET 目标属性，在使用Xcode生成器时，后者优先。

```
# Set the deployment target for macOS with any CMake version, or all Apple
# platforms when using CMake 3.11 or later
cmake_minimum_required(VERSION 3.9)

# Must be before first call to project()
set(CMAKE OSX_DEPLOYMENT_TARGET 10.11)
project(AppleProject)
```

```

# Set the deployment target for iOS with any CMake version.

# Set defaults for all targets added hereafter within this directory scope or below
set(CMAKE_XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 9.0)

# Build an app with the deployment target explicitly set
add_executable(MyApp MACOSX_BUNDLE ...)
set_target_properties(MyApp PROPERTIES XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 10.0)

```

iOS项目还可能希望指定目标设备系列，Apple具有 `TARGETED_DEVICE_FAMILY` 属性，可用于指定设备系列。对于 iOS系统，有效值为iPhone的1(技术上也包括iPod touch)或iPad的2。如果应用程序同时支持iPhone和iPad，可以用逗号分隔指定两个值。如果未设置此属性，则默认为1。Xcode会使用这个值在应用程序的信息，在Info.plist中添加`UIDeviceFamily`条目，所以避免在Info.plist设置这个条目

```

# An app that supports only iPad
add_executable(MyiPadApp MACOSX_BUNDLE ...)
set_target_properties(MyiPadApp PROPERTIES
  XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 2
)

# An app that supports both iPhone and iPad
add_executable(RunEverywhereApp MACOSX_BUNDLE ...)
set_target_properties(RunEverywhereApp PROPERTIES
  XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY 1,2
)

```

以上介绍了大多数Apple项目需要定义与构建的主要设置。对于简单的未签名macOS应用足够了，但是大多数项目需要进一步的配置签名来构建产品。

## 22.6. 代码签名

过去几个的Xcode版本中，与代码签名相关的Xcode功能有了很大的发展。最近，代码签名和配置的自动化管理使得使用CMake构建签名应用程序变得更加容易，但仍需要理解签名过程，从而设置适当的属性和变量。Xcode 8中，自动签名和配置的工作方式发生了显著的变化，留下了许多演示Xcode 7和更早版本方法的示例，不再是最佳实践。本章主要介绍当前的自动签名和配置过程。

为了使自动签名和配置，应用程序必须有一个有效的包ID和另外两个需要提供的关键信息：开发团队ID和代码签名。这些属性需要指定为Xcode属性，通过属性或CMake变量在单个目标上设置，为相应的目标属性指定默认值。由于这两个数量在整个构建过程中通常需要相同，因此建议设置为项目顶部变量。

应该将 `XCODE_ATTRIBUTE DEVELOPMENT_TEAM` 目标属性或相应的 `CMAKE_XCODE_ATTRIBUTE DEVELOPMENT_TEAM` 变量设置为开发团队ID，这是通常是一个约10字符的短字符串。最方便的方法通常是在最顶层的CMakeLists.txt文件中的第一个`project()`命令之后设置 `CMAKE_XCODE_ATTRIBUTE DEVELOPMENT_TEAM`。根据项目的不同，开发人员可能需要更改此值。例如，如果是商业软件，并由一名员工构建，那么团队ID很可能永远不会改变，而对公众开放的开源项目，肯定是由开发人员使用自己的开发团队ID构建的。团队ID永远不会改变的情况下，只需要将 `CMAKE_XCODE_ATTRIBUTE DEVELOPMENT_TEAM` 定义为普通的变量是足够了，当开发人员需要更改时，应该将其定义为缓存变量，这样就可以给定默认值，开发人员可以在不编辑CMakeLists.txt文件的情况下进行覆盖。

类似地，`XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` 属性或相应的 `CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` 变量可以指定签名标识。Xcode 8中，这应该始终是字符串 `Mac Developer for macOS applications`、`iPhone Developer for iOS, tvOS 或 watchOS applications`。这些值将引导Xcode为开发团队选择最合适的签名标识。特殊情况下，签名标识可以设置为字符串，该字符串标识开发人员密钥链中的特定签名标识，但开发人员有责任确保该标识属于相应的开发团队。

下面的例子展示了CMakeLists.txt是如何为macOS应用程序构建的，该应用程序允许开发者更改团队ID和签名身份：

```
cmake_minimum_required(VERSION 3.9)
project(macOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM "ABC12345DE" CACHE STRING "")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "Mac Developer" CACHE STRING "")
```

对于一个iOS应用程序，团队ID预计不会改变，但开发者可能想要控制签名身份，只有身份需要缓存：

```
cmake_minimum_required(VERSION 3.9)
project(iOSexample)

set(CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM "ABC12345DE")
set(CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer" CACHE STRING "")
```

如上所述配置时，Xcode将自动选择适当的配置文件。如果合适的配置文件不存在，Xcode IDE可以自动创建一个(该功能是IDE的一个特性，不能用命令行构建)。这种自动配置是对早期Xcode版本的重大改进。早期版本中，配置文件必须通过开发人员手动创建。

前面的小节中，`CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED` 没有在iOS的工具链文件中设置，但该变量在`CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` 设置时忽略。移动代码签名信息在工具链文件中要避免`CMAKE_XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY` 的设置，这意味着`try-compile` 测试会作为目的执行的第一个`project()` 的一部分，这就需要一个有效的配置文件和有效的包ID。通常情况下，不推荐在团队帐户中创建这样的包ID和配置文件。`try-compile` 测试不需要执行代码签名，因此不应该使用工具链文件来启用全局签名。

Apple应用程序也有一组相关的权限，操作系统的这些控制功能将允许应用程序使用，比如Siri，推送通知等功能。Xcode IDE的项目设置中，用户可以转到App目标的Capabilities选项卡，打开所需的功能。启用相关的授权会自动生成到plist文件中，目标会链接到任何需要的框架，并将该功能添加到团队账户中应用的ID中。对于CMake生成的项目，这个Capabilities选项卡将会绕过。如果默认的权利不够，CMake项目应该直接提供相应的plist文件。项目本身必须处理所有需要的框架链接，并且App ID不做任何更改。实际上，对于许多应用程序来说，这些限制相当温和，只是会在框架链接阶段出现了一些问题。

通过`XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS` 目标属性指定合适的权限文件的名称，如下所示：

```
set_target_properties(myApp PROPERTIES
  XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS
  ${CMAKE_CURRENT_LIST_DIR}/myApp.entitlements
)
```

举个例子，将Siri添加到默认权限中：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>com.apple.developer.siri</key>
<true/>
</dict>
</plist>
```

如果应用程序链接到共享框架，而这些框架也是由项目构建的，不要为这些框架启用代码签名。推荐的方法是通过Xcode的Embed frameworks构建阶段，启用Code sign on copy选项，不幸的是CMake并不直接支持(关于CMake支持框架的限制的讨论，请参阅22.8节)。

## 22.7. 创建和导出归档

为了通过App Store、企业发布门户或临时发布来发布应用程序，首先要创建归档文件。虽然CMake不为创建这样的归档文件构建目标，但xcodebuild工具可以用CMake生成的项目来完成该任务。归档构建操作只需要几个选项，就可以构建发布所需的目标并创建归档。有几种方法可以指定要归档的内容，但相当简单的方法是命名项目、方案和输出的名称：

```
xcodebuild archive \
-project MyProject.xcodeproj \
-scheme MyApp \
-archivePath MyApp.xcarchive
```

使用Xcode生成器时，CMake会创建.xcodeproj文件。CMake 3.9之前，用户必须在Xcode IDE中加载项目来创建构建方案。这对于无法访问IDE的持续集成构建来说是一个问题，为了解决这个问题，CMake 3.9引入了CMAKE\_XCODE\_GENERATE\_SCHEME变量作为实验特性。这个变量设置为true时，CMake还会构建生成模式文件，然后允许为-scheme选项指定App目标的名称，归档任务就有所有需要的信息。上面的命令将为所有支持的架构构建发布配置的MyApp目标进行签署(仍然使用开发人员签名身份)，然后创建一个名为MyApp的归档文件，存档在当前目录中。

如果没有适当地设置某些安装属性，归档可能会失败。Apple developer文档包含一些故障诊断指南，这些有助于解决常见问题，其中一些相关的情况是确保目标类型正确地设置了目标的INSTALL\_PATH和SKIP\_INSTALL属性。CMake项目中，目标的XCODE\_ATTRIBUTE\_SKIP\_INSTALL属性对于库和嵌入式框架必须设置为YES，对于应用程序必须设置为NO。设置为NO的地方，还必须提供XCODE\_ATTRIBUTE\_INSTALL\_PATH，通常为\$(LOCAL\_APPS\_DIR)。如果不遵循这个建议，归档步骤通常会生成通用归档，而不是应用程序归档。

```
# Apps must have install step enabled
set_target_properties(macOSApp PROPERTIES
  XCODE_ATTRIBUTE_SKIP_INSTALL NO
  XCODE_ATTRIBUTE_INSTALL_PATH "${LOCAL_APPS_DIR}"
)
```

创建应用程序归档之后，需要将其导出以便分发。这是通过对xcodebuild工具的另一次调用实现，这次提供了刚刚创建的存档、一个选项plist文件和写入输出的位置。该命令的基本形式如下：

```
xcodebuild -exportArchive \
-archivePath myApp.xcarchive \
-exportOptionsPlist exportOptions.plist \
-exportPath Products
```

-archivepath选项指向由前面的xcodebuild调用创建的归档文件，而-exportPath选项指定创建最终输出文件的目录。关于导出步骤的其他内容，由-exportOptionsPlist指向的选项plist文件定义。支持的全部密钥可以在工具的帮助文档(xcodebuild -help)中找到，plist文件可能是这样的：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>method</key>
<string>app-store</string>
</dict>
</plist>
```

该方法指定了预期的发布渠道，并期望是以下渠道之一：

- app-store
- ad-hoc
- enterprise
- development
- developer-id
- package

默认情况下是development，但更有可能的主要方法是app-store、enterprise或ad-hoc。导出归档文件时，该工具将重新对应用程序进行签名，并根据所选方法选择合适的分发签名标识。开发人员应该创建/下载了适当的发行版签名身份和配置文件(很容易在Xcode IDE中完成，也可以为持续集成服务器等手工完成)。

## 22.8. 限制

至少在3.11版本之前，CMake对框架的支持有一些缺点。这在很大程度上是由于框架(甚至是常规库)合并到Xcode项目中，而不是在库构建阶段定义的链接文件，CMake将链接直接硬编码到另一个链接器标志的目标属性中。这与CMake在使用其他生成器类型时链接库和框架的方式相匹配，但是这些生成器不具有Xcode所具有的附加框架处理和代码签名特性。CMake确实会尝试检测它要链接的库是否是一个框架，使用`-framework someLib`而不是`-lsomeLib`或`/path/to/someLib`。对于那些识别为框架的，在链接器命令行中使用`dylib`，但是这并不能让Xcode了解其他框架。

对于静态框架，CMake是可用的，但是对于动态框架，则存在一些问题。将信息嵌入到链接器标志时，Xcode无法完全了解这个框架，创建应用程序存档或执行代码签名时也无法正确处理。因为没有定义关联的文件构建阶段，所以框架不会与目标链接一起安装，而嵌入式框架的代码签名通常是在构建期间执行的。

项目可用的选择受到CMake行为的限制。可以避免使用任何非系统的动态框架，这也有缺点。项目可能需要开发人员在运行CMake时和手动添加框架后，进行一些手工更改，这显然是不合理的，并排除了在无头文件环境中构建，例如：持续集成系统中。一个更可行的方法是在CMake运行后定义脚本来修改Xcode项目文件，或者在CMake项目中定义自定义命令，或后构建步骤来模拟Xcode项目在了解嵌入式框架的情况下通常会做的事情。这些选项都不是特别理想，而且都违背了CMake的本质。即使是自定义脚本或后构建步骤的方法也很有可能在未来与CMake的改进发生冲突。

CMake对权限的处理也相当简单，不如Xcode IDE在Capabilities target properties选项卡中提供的自动化处理方式。选项卡中，打开一个特定的功能还需要添加所需的框架，并自动更新应用ID信息。CMake的支持允许指定所有权限，但整个过程完全是手动的。项目负责以原始的plist格式定义权限，而且还必须手动链接到任何框架，这一点CMake没有很好地处理。尽管如此，在没有变通或让步骤变得过于繁琐的方法时，权限的处理至少是可能的。授权所需要的框架都是系统提供的，因此不需要嵌入到应用程序中，从而避免了大多数框架的处理缺陷。

对于不明显的CMake行为，使用Xcode生成器，会创建名为 `ZERO_CHECK` 的实用程序目标。项目中的大多数其他目标都依赖于 `ZERO_CHECK`，它的目的是在执行剩余构建之前确定是否需要重新运行CMake。不幸的是，如果由 `ZERO_CHECK` 重新运行CMake，该构建的其余部分仍然使用旧的项目信息，这可能会导致错误，因为目标是用“过时的设置”构建的。重建一次应该确保正确的目标重建，但这个目标很容易忽略。开发人员可能需要显式地构建 `ZERO_CHECK` 目标，或者修改后重新运行CMake第一CMakeLists.txt文件，或让CMake会自动重新运行，或简单地构建两次。

如果项目包含对 `project()` 的多个调用，则存在一个与 `ZERO_CHECK` 相关的问题。第一个以后的 `project()` 调用下面定义的目标，可能没有正确地设置对 `ZERO_CHECK` 的依赖关系。可以将 `CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY` 变量设置为true来避免这个问题，它还会加速CMake，在CMake 3.11添加了对该变量的支持。

## 22.9. 总结

CMake能够处理针对Apple平台的项目，但需要仔细考虑其局限性。如果必须用程序进行签名，那么使用任何非系统共享框架都需要手动编写脚本和自定义构建步骤，以获得所需结果。如果不需要动态框架，那CMake的功能就足够了，只要使用Xcode生成器，通常可以不费太多力气地自动完成。其他生成器(如Makefile或Ninja)对于构建未签名的macOS应用程序没什么问题，但对于其他平台或签名的应用程序，这些生成器通常缺乏能够轻松生成用于发布包所需的特性。除了未签名macOS应用程序开发，强烈建议使用Xcode生成器进行Apple平台开发。

当在Apple平台上使用CMake时，在线教程和示例中的很多信息相对来说已经过时。在iOS中看到复杂的工具链文件很常见，但这些工具链文件中包含的大部分逻辑若是有必要，最好由项目本身提供。对于Xcode 8或更高版本，如果项目致力于利用自动签名和配置，这会极大地简化了签名过程。为最小的工具链文件只需设置 `CMAKE_MACOSX_BUNDLE`、`CMAKE_XCODE_ATTRIBUTE_CODE_SIGNING_REQUIRED` 和 `CMAKE_OSX_SYSROOT` 就可以得到支持代码签名和分发的构建。其他与Xcode项目设置、特定于设备或平台的配置等相关的逻辑应该放在项目中。

教程和示例经常做的事是通过 `CMAKE_OSX_ARCHITECTURES` 来指定目标架构。针对iOS、watchOS或tvOS的项目中使用Xcode生成器时，这并不合适，这会妨碍开发人员在设备和模拟器构建之间自由切换。当在Xcode IDE中工作或在命令行构建时，目标架构是可选择的。因此，项目应该避免设置 `CMAKE_OSX_ARCHITECTURES`，而是让Xcode根据所选的SDK提供标准的框架集。SDK由 `CMAKE_OSX_SYSROOT` 确定，当选择设备SDK时，Xcode能够识别匹配的模拟器。例如，通过将 `CMAKE_OSX_SYSROOT` 设置为iphoneos，开发人员就可以使用设备和模拟器构建。此外，虽然可以将SDK版本指定为 `CMAKE_OSX_SYSROOT` 的一部分，但通常没有理由这样做。更有可能的是部署目标应设置通过 `MACOSX_DEPLOYMENT_TARGET` 或 `XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET` 设置SDK的版本。最终决定应用程序是否能够在目标上运行的是部署目标，这与构建的SDK无关(当然，假设SDK支持该部署目标)。由于默认情况下使用最新的SDK可用版本，因此构建使用特定的SDK版本几乎不会有什么好处，甚至是有害的。当指定了特定的SDK版本，并不是所有的开发人员机器都可以使用，因为这将取决于使用的是哪个Xcode版本。一些开发人员将旧的SDK移植到新的Xcode版本中，试图解决这个问题，但这显然没必要。

示例还将 `CMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH` 设置为true，以便只在Xcode IDE中构建当前选择的架构。同样，这是应该由开发人员在构建时决定，而不是由CMake强制执行。有时，也用于脚本构建，因为已知只应该构建一个特定的平台，架构可以指定为命令行选项。

如果项目包含的目标链接到不提供宽二进制的库或框架(即只针对单一目标平台构建)，那么将构建限制为一个架构是有意义的。这种情况下，由于这些库或框架只支持单个平台，因此只能为该平台构建项目。类似地，使用 `find_library()` 或 `find_package()` (下一章将介绍)时，这些命令本质上假设是为单个平台构建的，因此不尝试以支持在多个目标平台之间切换的方式，对平台进行定义。

有些项目会选择使用CMake的安装功能，而不是假设Xcode在构建时为一个可分发包完成所有需要的工作。对于这种情况，可以将 `IOS_INSTALL_COMBINED` 目标属性设置为true，以构建目标的设备和模拟器版本，并在安装步骤中合并为单个宽二进制文件。如果由于某些原因不使用Xcode生成器，或者项目的结构遵循了CMake更

不依赖于平台的构建-安装模型，那这可能是合适的选择。

**Xcode**的构建输出可能非常冗长，因此开发人员可能会选择使用**xcpretty**这样的工具来隐藏大部分细节(脚本构建中更常见，以减少日志大小)。不幸的是，这个特殊的工具通常会隐藏任何**CMake**的自定义后构建步骤的输出(即使这些自定义步骤会导致构建错误)。因此，当这类自定义步骤失败时，很难找出失败的原因，因此建议避免使用此工具，或者让它在脚本中易于关闭，以帮助诊断构建问题。**xcodebuild**命令的 `-quiet` 选项可以在不隐藏警告或错误的情况下减少日志的输出量。

## 第三部分：常用功能

对于少数幸运的人来说，项目会不需要依赖于任何其他东西。更有可能的情况是，项目需要超越自己的存在，并可与外部进行交互。这可以从两方面说起：

### 依赖性

项目可能依赖于其他外部提供的文件、库、可执行文件、包等等。

### 用户

其他项目可能希望使用该项目。一些人可能希望在源代码级别合并它，另一些人可能希望构建一个可用的二进制包。另一种可能是将项目安装在系统的某处。

让项目作为可用的独立包让其他项目使用，也会带来对包质量的期望。自动化测试通常是软件交付策略的关键部分，必须易于定义和执行测试，以及报告结果。

**CMake**工具套件提供了上述所有方面的功能。用于查找单个文件、库等的命令，还提供构建在这些命令之上的模块，为依赖关系管理提供更高级别的入口。**CTest**框架提供了一组丰富的自动化测试功能，而**CPack**大大简化了以各种格式创建包的过程。本书的这部分涵盖了这些外部关注的主题，展示了如何最大限度地利用**CMake**，同时也强调了常见的错误和陷阱。

本书的最后一章将讨论如何组织项目。做好这一工作需要对构建级特性，以及项目与其他项目之间的交互方式有所了解。借助前面章节中获得的知识，本文展示了如何构造和定义一个灵活、健壮且易于(开发人员)使用的项目。

# 第23章：查找

中等规模的项目很可能依赖于项目本身之外的东西较多。例如，期望有特定的库或工具可用，或者需要知道使用库的特定配置文件或头文件位置。更高的层次上，项目可能需要找到一个包，该包定义了一系列内容，包括目标、函数、变量，以及常规CMake项目可能定义的任何内容。

为了实现这一点，CMake提供了多种功能，让项目不用为找东西发愁，甚至可以很容易找到想要的东西，并能合并到其他项目中。`find_…()`命令提供了搜索文件、库或程序，甚至包的能力。CMake模块还添加了使用`pkg-config`来提供外部包信息的功能，其他模块则编写供其他项目使用的包文件。这一章涵盖了CMake对搜索文件系统中可用内容的支持。下载缺失的依赖在第27章中会来详细了解。可被其他项目发现的项目，会在第25.7节中介绍。

搜索的基本思想相对简单，但如何进行搜索则相当复杂。许多情况下，默认行为就足够了，但是对搜索位置和顺序的设置可以允许项目进行定制化搜索，以解释非标准行为和特殊情况。

## 23.1. 查找文件和路径

最基本的搜索任务是查找指定文件，使用`find_file()`实现。`find_…()`命令族中，有许多相同的选项和类似的行为。该命令的完整形式如下：

```
find_file(outVar
  name | NAMES name1 [name2...]
  [HINTS path1 [path2...] [ENV var]...]
  [PATHS path1 [path2...] [ENV var]...]
  [PATH_SUFFIXES suffix1 [suffix2 ...]]
  [NO_DEFAULT_PATH]
  [NO_PACKAGE_ROOT_PATH]
  [NO_CMAKE_PATH]
  [NO_CMAKE_ENVIRONMENT_PATH]
  [NO_SYSTEM_ENVIRONMENT_PATH]
  [NO_CMAKE_SYSTEM_PATH]
  [CMAKE_FIND_ROOT_PATH_BOTH |
  ONLY_CMAKE_FIND_ROOT_PATH |
  NO_CMAKE_FIND_ROOT_PATH]
  [DOC "description"]
)
```

该命令可以通过单个文件名搜索，也可以通过`NAMES`选项提供文件名列表。当要搜索文件的名称有一些变化时，列表就会很有用，比如：不同的操作系统发行版选择不同的命名方式、包含或不包含版本号、考虑文件从一个版本更改名称到另一个版本等。名称应该按优先顺序列出，搜索将在第一个匹配处停止。当指定文件名包含某种形式的版本号时，CMake文档建议列出没有版本细节的名称，这样就可以在本地构建文件中查找，没找到时再在操作系统提供的文件中查找。

搜索将根据明确的顺序在一组地点进行检查。大多数位置都有相关的选项，如果该选项存在，该选项将跳过该位置，从而允许根据需要定制搜索范围。下表总结了搜索顺序：

位置	“跳过”选项
包的root路径	NO_PACKAGE_ROOT_PATH
缓存变量 (CMake)	NO_CMAKE_PATH
环境变量 (CMake)	NO_CMAKE_ENVIRONMENT_PATH
通过HINTS选项指定路径	
环境变量 (系统)	NO_SYSTEM_ENVIRONMENT_PATH
缓存变量 (平台)	NO_CMAKE_SYSTEM_PATH
通过PATHS选项指定的路径	

### 包的root路径

第一个搜索的位置在 `find_file()` 调用Find模块时应用(本章后面将讨论)。最初在CMake 3.9.0中的添加了搜索位置，但在3.9.1中由于向后兼容性问题而删除。在问题解决后，CMake 3.12中又重新添加它。关于搜索位置的讨论将在第23.5节中进行，这里更加关注使用。

### 缓存变量 (CMake)

CMake搜索位置的缓存是从缓存变量 `CMAKE_PREFIX_PATH`、`CMAKE_INCLUDE_PATH`和`CMAKE_FRAMEWORK_PATH` 派生而来。其中，`CMAKE_PREFIX_PATH` 的设置不仅适用于 `find_file()`，而且适用于所有 `find_...()` 命令。在这个基准点下，需要bin、lib、include等目录结构，每个 `find_...()` 都会附加自己的子目录来构造搜索路径。`find_file()` 会搜索 `CMAKE_PREFIX_PATH` 的每个目录，所以目录 `<prefix>/include` 肯定会进行搜索。如果设置了 `CMAKE_LIBRARY_ARCHITECTURE`，那么将首先搜索架构的 `<prefix>/include/${CMAKE_LIBRARY_ARCHITECTURE}` 目录，以确保特定架构的位置优先于一般位置。`CMAKE_LIBRARY_ARCHITECTURE` 通常由CMake自动设置，项目不应该设置它。

对于搜索的头文件目录或框架路径，当不是标准目录布局或包的一部分时，可以使  
用 `CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH`。它们提供搜索目录列表，但与 `CMAKE_PREFIX_PATH` 不同，没有附  
加头文件目录的子目录。`find_file()` 和 `find_path()` 支持 `CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH`，并  
且 `find_library()` 也支持 `CMAKE_FRAMEWORK_PATH`。除此之外，这两组路径的处理方式相同。请参阅23.1.1节了解  
更多细节。

### 环境变量 (CMake)

CMake的环境变量位置与缓存变量位置非常相似。三个环境变  
量 `CMAKE_PREFIX_PATH`、`CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH` 的处理方式与同名缓存变量相同，只是在  
Unix平台上，每个列表项之间用冒号(:)而不是分号(;)分隔。这允许环境变量使用特定于平台的路径，这些路  
径与平台路径的样式相同。

### 环境变量 (平台)

系统的环境变量包括 `INCLUDE` 和 `PATH`。两者都可以包含路径分隔符(Unix系统上是冒号，Windows上是分号)  
分隔的各个路径，每项都会添加到搜索位置集中(`INCLUDE` 添加在 `PATH` 之前)。

在Windows上(包括Cygwin)，`PATH` 将以更复杂的方式处理。对于 `PATH` 环境变量中的每个项，通过从末尾删  
除bin或sbin子目录来计算基本路径。然后使用基本路径向搜索位置添加一个或两个路径。如果定义  
了 `CMAKE_LIBRARY_ARCHITECTURE`，则添加 `<base>/include/${CMAKE_LIBRARY_ARCHITECTURE}`。然后，不管是否定  
义 `CMAKE_LIBRARY_ARCHITECTURE`，`<base>/include` 都会添加到搜索路径集中。搜索路径排序中，这些路径会放在  
其他路径项的前面。如果 `PATH` 环境变量设置为 `C:\foo\bin;D:\bar`，并且 `CMAKE_LIBRARY_ARCHITECTURE` 设置  
为 `somearch`，那么搜索路径将按照如下顺序添加：

- C:\foo\include\somearch
- C:\foo\include
- C:\foo\bin
- D:\bar\include\somearch
- D:\bar\include
- D:\bar

### 缓存变量 (平台)

平台的缓存变量位置与CMake的缓存变量位置非常相似，名字稍有变化，但模式相同。变量名是 `CMAKE_SYSTEM_PREFIX_PATH`、`CMAKE_SYSTEM_INCLUDE_PATH` 和 `CMAKE_SYSTEM_FRAMEWORK_PATH`。这些特定于平台的变量不由项目或开发人员设置，作为设置平台工具链的一部分，以便反映特定于所使用的平台和编译器位置。例外情况是开发人员提供了工具链文件，这种情况下，在工具链文件中设置变量可能更合适。

### HINTS和PATHS

上面讨论的每组变量都在项目之外设置，但 `HINTS` 和 `PATHS` 是项目添加搜索路径的地方。`HINTS` 和 `PATHS` 的主要区别在于，`PATHS` 通常是固定的位置，永远不会改变，不依赖于任何其他东西，而 `HINTS` 通常是计算出来的，比如之前已经找到的位置，或者依赖于变量或属性值的路径。`PATHS` 是搜索的目录，`HINTS` 是搜索特定于平台或系统的位置之前的搜索路径。

`HINTS` 和 `PATHS` 都支持环境变量，这些环境变量可以包含主机本地格式的路径列表(例如，Unix系统使用冒号分隔，Windows使用分号分隔)。可以通过在变量名前面加 `ENV` 来实现，比如 `PATHS ENV FooDirs`。

除了 `HINTS` 和 `PATHS` 搜索位置之外，所有的搜索位置都有 `NO...PATH` 的跳过选项，该选项可用于跳过相应位置。此外，可以使用 `NO_DEFAULT_PATH` 选项绕过除了 `HINTS` 和 `PATHS` 位置之外的所有位置，强制命令只搜索特定位。

`PATH_SUFFIXES` 选项可用于提供子目录，以便检查每个搜索位置下的子目录。搜索位置依次与后缀一起使用，在转移到下一个搜索位置之前完全不带任何后缀。谨慎使用此选项，因为它扩展了搜索位置。

许多情况下，搜索顺序的复杂性并不重要，项目只需要指定搜索的文件名，只需要提供一些搜索路径(相当于路径选项)。这种情况下，可以使用该命令的简短形式：

```
find_file(outVar name [path1 [path2...]])
```

无论使用的是简短形式还是完整形式，搜索位置的顺序都设计为在具体的位置搜索之前搜索。虽然这是希望的行为，但在某些情况下可能不是这样。例如，项目可能希望在通过缓存或环境变量提供的搜索位置之前，先查找特定的路径。通过使用控制搜索位置的不同选项多次调用 `find_file()`，项目可以强制执行不同优先级的查找。一旦找到该文件，就缓存该位置，跳过后续搜索，这就是 `NO...PATH` 选项最有用的地方。例如，以下命令强制首先搜索 `/opt/foo/include`，只有在没有找到的情况下，才搜索默认位置：

```
find_file(FOO_HEADER foo.h PATHS /opt/foo/include NO_DEFAULT_PATH)
find_file(FOO_HEADER foo.h)
```

这样做的要求是每个调用都使用相同的结果变量，设置缓存变量，并在找到文件后跳过后续的查找。`DOC` 选项可用于将缓存变量存储为指定文档，但这个选项几乎没什么人用，可以自解释的缓存变量名不需要文档。按照惯例，这些缓存变量通常都是大写，并使用下划线分隔单词。

### 23.1.1. Apple平台特定的行为

尽管可以使用 `find_file()` 查找文件，但它最开始是用来搜索头文件的。这就是为什么一些默认搜索路径有 `include` 子目录。**Apple** 平台上，框架有时包含自己的头文件，并且 `find_file()` 还有其他行为，这些行为与相应子目录中的搜索相关。对于每个搜索位置可以视为框架或普通目录，或两者兼都有。行为由 `CMAKE_FIND_FRAMEWORK` 变量控制，该变量应该包含以下值之一：

- FIRST
- LAST
- ONLY
- NEVER

`FIRST` 是将搜索位置视为框架的项目目录，并添加适当的子目录以下移到头文件位置。如果找不到指定的文件，则将搜索位置视为普通目录而不是框架，并再次搜索。`LAST` 反转了这个顺序，`ONLY` 不会将位置视为普通目录，也不会跳过将位置视为框架的步骤。**Apple** 系统的默认值是 `FIRST`。

### 23.1.2. 控制交叉编译

对于交叉编译，搜索位置会变得相当复杂。交叉编译工具链通常收集自己的目录结构下的工具，这样就能与主机默认的构建工具区分开。在搜索特定文件时，通常先查看工具链的目录结构，再查看主机的目录结构，以便找到特定平台的目标文件。这在查找程序和库时尤其重要，即使是查找文件，也可能会出现文件内容在不同平台之间发生变化的情况(例如，特定于平台的头文件)。

为了支持交叉编译，可以将整个搜索位置集的起始位置与文件系统进行区分。可以使 `CMAKE_FIND_ROOT_PATH` 设置附加目录列表，可以在该列表中更新起始目录的搜索位置(即将列表中的项前置)。`CMAKE_SYSROOT` 变量也可以以类似的方式修改搜索起始目录。这个变量用于为交叉编译指定起始目录的位置，并且只能在工具链文件中设置，而不能由项目设置，还会影响编译使用的标志。**CMake 3.9** 中，`CMAKE_SYSROOT_COMPILE` 和 `CMAKE_SYSROOT_LINK` 也具有类似的效果。如果任何非起始位置已经位于 `CMAKE_FIND_ROOT_PATH`、`CMAKE_SYSROOT`、`CMAKE_SYSROOT_COMPILE` 或 `CMAKE_SYSROOT_LINK` 指定的位置，则不会更新起始目录，由 `CMAKE_STAGING_PREFIX` 指定路径下的非起始路径也不会进行更新。此外，所有 `find_...()` 的未定义行为不会更新任何以 ~ 字符开头的非起始路径(为了避免更新用户目录)。

更新起始置和非起始位置之间的默认搜索顺序由 `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE` 控制，为 `find_file()` 提供 `CMAKE_FIND_ROOT_PATH_BOTH`、`ONLY_CMAKE_FIND_ROOT_PATH` 或 `NO_CMAKE_FIND_ROOT_PATH` 选项，指定的行为也可以在每次调用时重写。下表总结了这个模式变量的影响，相关选项和最终的搜索顺序：

搜索模式	<code>find_file()</code> 选项	搜索顺序
BOTH	<code>CMAKE_FIND_ROOT_PATH_BOTH</code>	<code>CMAKE_FIND_ROOT_PATH</code> <code>CMAKE_SYSROOT_COMPILE</code> <code>CMAKE_SYSROOT_LINK</code> <code>CMAKE_SYSROOT</code> All non-rooted locations
NEVER	<code>NO_CMAKE_FIND_ROOT_PATH</code>	All non-rooted locations
ONLY	<code>ONLY_CMAKE_FIND_ROOT_PATH</code>	<code>CMAKE_FIND_ROOT_PATH</code> <code>CMAKE_SYSROOT_COMPILE</code> <code>CMAKE_SYSROOT_LINK</code> <code>CMAKE_SYSROOT</code> Any non-rooted locations already under one of the re-rooted locations or under <code>CMAKE_STAGING_PREFIX</code>

可以强制让 `find_file()` 忽略不合适的路径。在交叉编译场景中，需要忽略某些主机路径，以确保找到特定于目标的文件。项目可以将 `CMAKE_IGNORE_PATH` 设置为要从搜索中排除的目录列表。这些路径不是递归的，所以不能用于排除目录结构的整个部分，需要显式地指定每个目录。`CMAKE_SYSTEM_IGNORE_PATH` 做同样的事情，但这个变量由工具链设置。这两个 `...IGNORE_PATH` 不管是否在交叉编译环境下都可用，交叉编译时比较常见。

`find_file()` 只能提供一个位置，而一些交叉编译支持无需重新运行CMake，也可以在设备和模拟器构建之间切换。如果 `find_file()` 的结果取决于使用的是哪一个构建，那无疑是不可靠的行为。这对于查找库很重要，23.4节中有更详细的讨论。

## 23.2. 查找路径

项目可能希望找到包含特定文件的目录，而不是实际文件本身。`find_path()` 提供了这个功能，可以获得要找到的文件的目录信息，(其他方面都与 `find_file()` 相同)。

## 23.3. 查找程序

查找程序与查找文件略有不同，`find_program()` 使用与 `find_file()` 完全相同的参数，外加一个可选参数 `NAMES_PER_DIR`。还支持该命令的短形式。下面描述了 `find_program()` 与 `find_file()` 的区别，虽然看起来复杂，大多数情况下，只是描述了逻辑上预期的区别，这里会展示一些例外情况：

### 缓存变量 (CMake)

- `CMAKE_PREFIX_PATH` 下搜索时，`find_file()` 将 `include` 追加到每项。`find_program()` 将会追加 `bin` 和 `sbin` 作为检查搜索位置。`CMAKE_LIBRARY_ARCHITECTURE` 对 `find_program()` 没有影响。
- 可以用 `CMAKE_PROGRAM_PATH` 替换 `CMAKE_INCLUDE_PATH`，而且使用方式上完全相同。就是 `CMAKE_PROGRAM_PATH` 只能在 `find_program()` 中使用。
- 可以用 `CMAKE_APPBUNDLE_PATH` 替换 `CMAKE_FRAMEWORK_PATH`，而且使用方式上完全相同。它们只能在 `find_program()` 和 `find_package()` 中使用。

### 环境变量(系统)

- 系统环境变量的搜索位置处理起来相当简单。`INCLUDE` 对于 `find_program()` 没有任何意义，路径中的每项都会检查，而不会进行任何修改。这种行为在所有平台上都是相同的。

### 通用变量

- 当 `NAMES` 选项用于提供多个名称时，在搜索下一个前，会检查所有搜索位置。`find_program()` 支持 `NAMES_PER_DIR` 选项，该选项可以在继续搜索下一个位置之前，使用名称对目录文件进行匹配。`NAMES_PER_DIR` 选项在CMake 3.4中添加。
- Windows上(包括Cygwin和MinGW)，`.com` 和 `.exe` 文件也会自动进行检查，所以不需要提供这些扩展名作为程序名的一部分来查找。首先检查这些扩展，然后是没有扩展的名称。注意，CMake不会自动搜索`.bat` 和 `.cmd` 文件。
- `find_file()` 使用 `CMAKE_FIND_FRAMEWORK` 来确定框架和非框架路径之间的搜索顺序，而 `find_program()` 使用 `CMAKE_FIND_APPBUNDLE`，在应用包和非包路径之间进行控制。这两个变量支持的值是相同的，并且对包的预期有同等的意义。`find_file` 将在头文件夹子目录中查找，而 `find_program` 将在 `Contents/MacOS` 子目录中查找，并将结果设置为应用包中的可执行文件。
- `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE` 对 `find_program()` 没有影响，可以替换为具有等价效果的 `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM`，但只适用于 `find_program()`。交叉编译时，通常寻找的是主机平台工具，因此 `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM` 常设置为 `NEVER`。

## 23.4. 查找库

查找库也类似于查找文件，`find_library()` 支持与 `find_file()` 相同的选项，另外还支持 `NAMES_PER_DIR` 选项。差异如下：

### 缓存变量(CMake)

- 在 `CMAKE_PREFIX_PATH` 下搜索时，`find_file()` 将 `include` 追加到每项，而 `find_library()` 将追加 `lib`。
- `CMAKE_LIBRARY_PATH` 可以替换 `CMAKE_INCLUDE_PATH`，使用方式完全相同。而 `CMAKE_LIBRARY_PATH` 只用与 `find_library()`。`CMAKE_FRAMEWORK_PATH` 的使用方式与 `find_file()` 完全相同。

### 环境变量(系统)

- 系统环境变量的搜索位置的处理方法与 `find_file()` 相似。使用 `LIB` 环境变量代替 `INCLUDE`。而且，路径的搜索位置遵循与 `find_file()` 相同，只是 `lib` 附加到每个搜索位置后。与 `find_file()` 一样，路径逻辑仅适用于 Windows。

### 通用变量

- `NAMES_PER_DIR` 选项的含义与 `find_program()` 相同，在 CMake 3.4 中添加。
- `find_file()` 和 `find_library()` 使用 `CMAKE_FIND_FRAMEWORK` 来确定框架和非框架路径间的搜索顺序。在 `find_library()` 中，如果找到了框架，那么顶层 `.framework` 目录的名称就是存储在结果变量中的名称。
- `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE` 对 `find_library()` 没有影响，可以替换为具有同等的效果的 `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY`，但只能于 `find_library()`。在 Apple 平台上，在将 `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY` 设置为 `ONLY` 时要慎重，因为库可能为支持多个目标平台构建宽二进制文件。这些宽二进制文件可能并不位于特定于目标平台的路径下，因此需要搜索主机平台的路径来找到它们。

使用 `find_library()` 时还存一些行为差异。平台对库名称有不同的约定，比如在大多数 Unix 平台上将 `lib` 作为前缀。文件扩展名在不同的平台上也有很大的不同，Windows 上的 `dll` 也可以有一个相关联的导入库。`find_library()` 尽可能抽象出这些差异，允许项目只指定库的基本名称进行搜索。如果目录同时包含静态库和动态库，那么使用动态库。大多数时候，这个抽象做的很好，但是在某些情况下，需要重写这个行为。一种情况是，某些平台上将静态库设置为优先于动态库。下面这个例子希望在 Linux 上使用 `foobar` 的静态库，而不是动态库，但在 macOS 或 Windows 上还是使用默认的优先级：

```
# WARNING: Not robust!
find_library(FOOBAR_LIBRARY NAMES libfoobar.a foobar)
```

优先级覆盖仅适用于特定目录中找到的库。如果含动态库的目录在包含静态库的目录之前进行了搜索，那么上述方法将找不到静态库。确保静态库在所有搜索位置上优先于动态库的更健壮的方法是使用多个 `find_library()`：

```
# Better, static library now has priority across
# all search locations
find_library(FOOBAR_LIBRARY libfoobar.a)
find_library(FOOBAR_LIBRARY foobar)
```

这种技术不能在 Windows 上使用，因为静态库和动态库的导入库（即 `dll`）具有相同的文件名，包括后缀（如 `foobar.lib`）。因此，不能使用文件名来区分这两种类型的库。

库处理的另一个复杂性是许多平台同时支持 32 位和 64 位体系结构，可能会有 32 位和 64 位版本的库安装到不同的位置，但文件名相同的情况。这种多 `lib` 系统上，用于分离不同体系结构的目录结构可能会有所不同，甚至在对同一平台的不同发行版之间也是如此。例如，一些发行版将 64 位库放在 `lib` 目录下，32 位库放在 `lib32` 目录

下，而另一些发行版将64位库放在lib64目录下，32位库放在lib目录下，还有一些发行版使用另一种方式，放在libx32子目录。**CMake**在设置平台默认值时通常会知道这些变化，并填充全局属性 `FIND_LIBRARY_USE_LIB32_PATHS`、`FIND_LIBRARY_USE_LIB64_PATHS`、`FIND_LIBRARY_USE_LIBX32_PATHS` 和控制首先搜索特定于体系结构的目录(如果有的话)。项目可以使用 `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX`，来定制前缀，不过这样的需求非常少见。

当某个特定于体系结构的后缀(无论是从上面的一个全局属性还是从 `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX`)，用特定于体系结构的位置扩充搜索路径都很重要。搜索位置路径中任何位置以lib结尾的目录都将添加特定于体系结构的等效内容。这种情况在整个路径中递归发生，因此像`/opt/mylib/foo/lib`这样的搜索位置可能会在某些64位系统上对搜索位置集进行扩展`/opt/mylib64/foo/lib64`、`/opt/mylib64/foo/lib`、`/opt/mylib/foo/lib64`和`/opt/mylib/foo/lib`。即使搜索位置没有以lib结束，仍然会通过以体系结构为后缀的位置来增强，所以搜索位置`/opt/foo`可能会在一些64位系统上搜索`/opt/foo64`和`/opt/foo`位置。

特定于体系结构的搜索路径的细节通常不需要开发人员关注。在找到了不需要的库或丢失了需要的库的情况下，使用 `CMAKE_LIBRARY_PATH` 强制执行的结果可能比尝试操作特定于体系结构的逻辑更直接。通常不需要对其中的复杂性有详细的了解，如果只是为了减少**CMake**如何在特定位置找到库的神秘感，对以上几点有所了解就足够了。

使用在构建时可以在设备和模拟器配置之间切换的**CMake**生成器时，任何 `find_library()` 结果都不可用，因为只能找到设备或模拟器的库，不能同时找到两者都需要的库。即使重新运行**CMake**，也会保留缓存结果，因此不会更新库的位置，除非手动删除相关的缓存条目，这在**Xcode**构建中是特别常见。其中项目可能希望使用 `find_library()` 来定位各种框架或公共库(如**zlib**)。对于这些情况，只能直接指定链接器标志(而不指定路径)，让链接器在搜索路径上查找库。对于**Apple**框架，需要指定两个值，因为框架使用 `-framework <FrameworkName>` 添加。对于像**zlib**这样的普通库，`-lz`足矣。

## 23.5. 查找包

前面讨论的各种 `find_…()` 命令都关注于查找一个特定的项。然而，这些项通常只是包的一部分，作为一个包可能有它自己的项目可能感兴趣的特征，例如版本号或对某些特性的支持。项目通常希望将包作为一个单独的单元来查找，而不是手动地将不同的部分拼凑在一起。

**CMake**中有两种主要的方法来定义包，一种是将包定义为模块，另一种是通过配置来定义。配置通常作为包本身的一部分提供，它们与前面小节中讨论的各种 `find_…()` 命令的功能更加一致。另一方面，模块通常与包无关的东西一起定义(通常由**CMake**或项目本身定义)，当包随着时间的推移而发展时，就很难跟上进度。

加载模块或配置文件时，通常为包定义变量和导入目标。可以提供程序、库、目标所使用的标志的位置等等，包也可以定义函数和宏(尽管只有模块才会这样做)。对于提供什么，没有一组准确的需求，不过**CMake**开发人员手册中有一些约定，项目作者必须参考每个模块或包配置的文档来理解它提供了什么。一般来说，较老的模块倾向于提供遵循一致模式的变量，而较新的模块和配置实现通常定义导入目标。当变量和导入的目标都提供时，项目应该更喜欢后者，因为它们具有卓越的健壮性，并且更好地与**CMake**的传递依赖特性进行集成。

项目通常使用 `find_package()` 查找包，该命令有短格式和长格式。短格式通常应该是首选，因为它更简单，同时支持模块和配置包，而长格式不支持模块。然而，长格式提供了更多的搜索控制，使它在某些情况下灵活。短格式只有几个选项：

```
find_package(packageName
[version [EXACT]]
[QUIET] [REQUIRED]
[[COMPONENTS] component1 [component2...]]
[OPTIONAL_COMPONENTS component3 [component4...]]
[MODULE]
[NO_POLICY_SCOPE]
)
```

可选的版本参数表示包必须是比指定更高版本，如果给出了确切的选项，则需要版本完全匹配。包是可选的，所以可以包在可用的情况下使用，在找不到包或者没有合适的版本时不使用。如果包是强制性的，则需要提供必要的选项，以便在找不到包或无法满足版本要求时停止并出现错误信息。如果 `find_package()` 无法找到包，会有记录消息，可以使用 `QUIET` 选项来禁用记录，这对于可选包特别有用，因为缺少包时的警告，会让开发人员困惑。当找到包时，`QUIET` 选项也会阻止消息输出。

与组件相关的选项允许项目指定对包感兴趣的组件。并不是所有的包都支持组件，组件是否定义，以及组件代表什么，都取决于模块或配置。组件对于大型包(比如Qt)更有用，可以不安装所有的组件。`find_package()` 允许项目用 `COMPONENTS` 参数指定组件为必选，或用 `OPTIONAL_COMPONENTS` 参数指定组件为可选。例如，下面的调用需要找到Qt 5.9或更高版本，并且Gui组件是必选，DBus模块是可选。

```
find_package(Qt5 5.9 REQUIRED
COMPONENTS Gui
OPTIONAL_COMPONENTS DBus
)
```

当出现 `REQUIRED` 选项时，可以省略 `COMPONENTS` 关键字，并将强制性组件放置在 `REQUIRED` 之后。例如：

```
find_package(Qt5 5.9 REQUIRED Gui Widgets Network)
```

如果包定义了组件，但是没有提供给 `find_package()`，那么如何处理就取决于模块或配置了。对于某些，可能不需要任何组件(尽管仍然可以定义包的基本细节，例如基础库、包版本等等)。另一种可能性是，缺少组件可视为错误。考虑到行为上的差异，开发人员应该参考相关的文档。

短格式的其余选项使用频率较低。`NO_POLICY_SCOPE` 关键字是CMake 2.6遗留的，不建议使用。`MODULE` 关键字将调用限制为只搜索模块，不搜索配置包。项目应该避免使用这个选项，因为不应该关心包的实现细节，只陈述对包的需求即可。当 `MODULE` 不存在时，`find_package()` 的短格式将首先搜索模块，如果没有匹配的模块，将搜索配置包。

模块在第11章模块中讨论过。虽然使用 `include()` 将非包模块合并到项目中，如果包模块的文件名为 `Find<packageName>.cmake`，就可以通过 `find_package()` 来处理，所以通常称为Find模块。`include()` 和 `find_package()` 都将 `CMAKE_MODULE_PATH` 变量看作是CMake在模块集之前搜索的目录。

Find模块负责实现 `find_package()` 的所有调用，包括查找包、执行版本检查、满足组件需求，以及适当地记录或不记录消息。并不是所有查找模块都会做这些，它们选择忽略包名之外提供的部分或全部信息。因此，查阅模块文档以确认模块的行为。

Find模块通过调用各种 `find_…()` 来实现。因此，会受到与这些命令相关的缓存和环境变量的影响。`CMAKE_PREFIX_PATH` 变量会影响模块的查找，指定的每个路径作为基点，每个 `find_…()` 命令都会追加自己的子目录。对于遵循标准布局的包，只需向 `CMAKE_PREFIX_PATH` 添加包的基本安装位置，Find模块就能找到所需的所有组件。

与查找模块相比，带有配置细节的包为项目提供了更丰富、更健壮的方法来检索包的信息。配置模式下，可以使用 `find_package()` 选项集，该命令的长格式与其他 `find_…()` 相似：

```

find_package(packageName
[version [EXACT]]
[QUIET | REQUIRED]
[[COMPONENTS] component1 [component2...]]
[NO_MODULE | CONFIG]
[NO_POLICY_SCOPE]
[NAMES name1 [name2 ...]]
[CONFIGS fileName1 [fileName2...]]
[HINTS path1 [path2 ... ]]
[PATHS path1 [path2 ... ]]
[PATH_SUFFIXES suffix1 [suffix2 ... ]]
[CMAKE_FIND_ROOT_PATH_BOTH |
ONLY_CMAKE_FIND_ROOT_PATH |
NO_CMAKE_FIND_ROOT_PATH]
[<skip-options>] # See further below
)

```

当使用完整格式支持的选项调用 `find_package()` 时，将跳过对模块的搜索。`NO_MODULE` 或 `CONFIG` 关键字允许将短格式的匹配调用作为长格式处理，因此只搜索配置信息(两个关键字是等效的)。

搜索配置细节时，`find_package()` 查找名为 `<packageName>Config.cmake` 的文件，或 `<lowercasePackageName>-config.cmake` (默认)。`CONFIGS` 选项可以用来指定一组要搜索的文件名，但不推荐使用这个选项。

找到配置文件时，`find_package()` 还会在同一目录中查找相关的版本文件。版本文件将 `Version` 或 `-version` 附加到文件名中，因此如果是查找 `FooConfig.cmake` 文件，则会查找名为 `FooConfigVersion.cmake` 或 `FooConfig-version.cmake` 的版本文件，而 `foo-config.cmake` 将查找 `foo-configVersion.cmake` 或 `foo-config-version.cmake`。包不需要版本文件，但会提供。如果版本信息包含在 `find_package()` 中，而包没有版本文件时，则认为版本获取失败。

搜索的位置遵循与其他 `find_…()` 类似的模式，也支持包的注册。每个搜索位置视为一个包安装基点，以下各种子目录都可以搜索：

```

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

# The following are also checked on Apple platforms

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

```

上面 `<packageName>` 不区分大小写，只有在设置了 `CMAKE_LIBRARY_ARCHITECTURE` 后，才会搜索 `lib/<arch>` 子目录。`lib*` 子目录代表一组目录，包括 `lib64`、`lib32`、`libx32` 和 `lib`，`lib` 子目录总会去检查。如果将 `NAMES` 选项提供给 `find_package()`，将针对名称检查上述所有目录。

检查的搜索位置集遵循下表中的顺序，它与其他 `find_…()` 有许多相似之处。通过添加相关的 `NO_…` 关键字，可以禁用大多数搜索位置：

位置	跳过选项
包的起始位置	NO_PACKAGE_ROOT_PATH
缓存变量(CMake)	NO_CMAKE_PATH
环境变量(CMake)	NO_CMAKE_ENVIRONMENT_PATH
通过HINTS选项指定的路径	
环境变量(系统)	NO_SYSTEM_ENVIRONMENT_PATH
用户包注册位置	NO_CMAKE_PACKAGE_REGISTRY
缓存变量 (平台)	NO_CMAKE_SYSTEM_PATH
系统包注册位置	NO_CMAKE_SYSTEM_PACKAGE_REGISTRY
通过PATHS选项指定的路径	

### 包的起始位置

对于其他 `find_...()` 命令，CMake 3.9.0中添加了包起始目录的作为搜索位置，不过由于向后兼容问题删除了，并在CMake 3.12中重新添加。每次调用 `find_package()` 时，都会将 `<packageName>_ROOT` CMake和环境变量添加到搜索路径上。路径的使用方式与 `CMAKE_PREFIX_PATH` 完全相同，不仅用于对 `find_package()` 的调用，也可以用于 `find_package()` 过程中的所有 `find_...()`。如果 `find_package()` 加载了Find模块，那么Find模块内部调用的任何 `find_...()` 都将使用中的路径，就好像是 `CMAKE_PREFIX_PATH`，然后再检查其他路径。

例如，`find_package(Foo)` 会加载 `FindFoo.cmake`，`FindFoo.cmake` 中的任何 `find_...()`，将首先搜索  `${Foo_ROOT}`  和  `${ENV{Foo_ROOT}}`  (如果设置了)，然后再继续检查其他搜索位置。如果 `FindFoo.cmake` 包含了类似于 `find_package(Bar)` 的调用，则会加载 `FindBar.cmake`，堆栈将包含  `${Bar_ROOT}` 、 `${ENV{Bar_ROOT}}` 、 `${Foo_ROOT}`  和  `${ENV{Foo_ROOT}}` 。这个特性意味着嵌套的Find模块将首先搜索父Find模块的前缀位置，这样就不必通过 `CMAKE_PREFIX_PATH` 或其他方法手动传播。大多数情况下，项目可以忽略此功能，应该透明地工作，不需要进行任何操作。

### 缓存变量(CMake)

CMake的缓存变量 `location` 是从缓存变量 `CMAKE_PREFIX_PATH`、`CMAKE_FRAMEWORK_PATH` 和 `CMAKE_APPBUNDLE_PATH` 派生而来。这些命令的工作方式基本上与 `find_...()` 相同，只是 `CMAKE_PREFIX_PATH` 对应于包安装基点，所以不会追加 `bin`、`lib`、`include` 等目录。

### 环境变量(CMake)

上述缓存变量的关系与其他 `find_...()` 相同。环境变量 `CMAKE_PREFIX_PATH`、`CMAKE_INCLUDE_PATH` 和 `CMAKE_FRAMEWORK_PATH` 都使用特定于平台的路径分隔符( Unix 平台上的冒号， Windows 上的分号)。另外一个环境变量 `<packageName>_DIR` 也会在其他三个环境变量之前检查。

### 环境变量(系统)

唯一支持的特定于系统的环境变量是 `PATH`。每个入口都作为包安装的基点，任何 `bin` 或 `sbin` 后缀会删除。大多数系统中可能会搜索 `/usr` 等默认系统位置。

### 缓存变量(平台)

平台的缓存变量 `location` 与其他 `find_...()` 相同，提供与CMake的缓存变量等效的 `..._SYSTEM...` 系统变量的名称是 `CMAKE_SYSTEM_PREFIX_PATH`、`CMAKE_SYSTEM_FRAMEWORK_PATH` 和 `CMAKE_SYSTEM_APPBUNDLE_PATH`，这些变量不由项目设置。

## HINTS和PATHS

工作方式与其他 `find_…()` 相同，只是不支持环境变量的方式 `ENV someVar`。

### 注册包

`find_package()` 的特点是，为了用户和系统注册包提供一种方法，使包可以轻松查找，无需安装在系统位置。

各种 `NO_…` 选项的方式与其他 `find_…()` 相同，允许绕过每组搜索位置。`NO_DEFAULT_PATH` 关键字会传递除了 `HINTS` 和 `PATHS` 的所有路径。`PATH_SUFFIXES` 选项也有预期的效果，会检查每个搜索位置的子目录。

`find_package()` 还支持与其他 `find_…()` 相同的重定位根目录的搜索逻辑。`CMAKE_SYSROOT`、`CMAKE_STAGING_PREFIX` 和 `CMAKE_FIND_ROOT_PATH` 都是按照与相同的方式，并且 `CMAKE_FIND_ROOT_PATH_BOTH`、`ONLY_CMAKE_FIND_ROOT_PATH` 和 `NO_CMAKE_FIND_ROOT_PATH` 选项的含义也是等效的。当这三个选项不提供时，默认的重定位根目录模式由 `CMAKE_FIND_ROOT_PATH_MODE_PACKAGE` 变量控制，该变量有三个有效值(`ONLY`、`NEVER` 或 `BOTH`)。

与 `find_…()` 不同，查找配置文件时，`find_package()` 并不一定在找到的第一个匹配条件的包处停止搜索。搜索会考虑一系列搜索位置，搜索结果可能会返回多个匹配的特定搜索分支。如果公共目录下安装了多个版本的包，每个版本都在公共目录下有一个版本化的子目录，就会发生这种情况。这种情况下，使用 `CMAKE_FIND_PACKAGE_SORT_ORDER` 和 `CMAKE_FIND_PACKAGE_SORT_DIRECTION` 变量根据候选对象的版本信息，对它们进行排序。`CMAKE_FIND_PACKAGE_SORT_DIRECTION` 必须具有值 `DEC` 或 `ASC`，分别指降序或升序方向，而 `CMAKE_FIND_PACKAGE_SORT_ORDER` 控制排序的类型，并记录了 `NAME`、`NATURAL` 或 `NONE` 的值。如果设置为 `NONE` 或没有设置，则不执行排序，并且使用找到的第一个有效包。名称设置按字典序排序，而自然排序是将数字序列作为整数进行比较。下表演示了前两种方法在降序排序时的区别，这是没有设置 `CMAKE_FIND_PACKAGE_SORT_DIRECTION` 时的默认行为：

名称	默认排序
1.9	1.10
1.10	1.9
1.0	1.0

实际中，搜索逻辑的复杂性通常超出了使用 `find_package()` 所需的详细程度。只要包遵循更常见的目录布局，并且位于较高级别的安装位置下，`find_package()` 通常不需要进一步帮助，就能找到配置文件。

找到适合包的配置文件，`<packageName>_DIR` 缓存变量将设置为包含该文件的目录。然后，对 `find_package()` 的后续调用将首先查找该目录，如果配置文件存在，则直接使用配置文件，而不进一步搜索。如果在该位置没有包配置文件，忽略 `<packageName>_DIR`。这种方式比对同一个包的再次使用 `find_package()` 快很多，如果包删除了，搜索仍会执行。要注意，包位置的缓存可能意味着CMake没有机会在更好的位置上发现新添加的包。例如，操作系统预装了一个相当旧的软件包。第一次在项目上运行CMake时，会找到那个旧版本并将其位置存储在缓存中。用户看到使用的是旧版本，决定在其他目录下安装新版本的包，将该位置添加到 `CMAKE_PREFIX_PATH` 并重新运行CMake。这个场景中，仍然会使用旧版本，因为缓存仍然指向旧包的位置。考虑新版本的位置之前，需要删除 `<packageName>_DIR` 的缓存条目，或者卸载旧版本。

还有一种控制可以影响特定包的处理，可以通过在项目早期将 `CMAKE_DISABLE_FIND_PACKAGE_<packageName>` 变量设置为 `true`，可以禁用 `packageName` 对 `find_package()` 的非 `REQUIRED` 调用，理想情况下是在顶层CMake中或作为缓存变量。这可以看作是关闭可选包的一种方法，防止 `find_package()` 找到它。注意，如果这些调用包含 `REQUIRED` 关键字，就不可能阻止。

### 23.5.1. 注册包

包往往在系统位置，**CMake**可以在 `CMAKE_PREFIX_PATH` (或类似的)目录中找到。对于非系统包，如果每个包不都共享公共的安装前缀，就必须为每个包指定位置，不推荐这样做。**CMake**支持包注册表形式，允许对任意位置的引用收集在一起。这允许用户维护帐户或系统范围的注册，**CMake**将自动咨询。注册引用的位置不一定是完整的安装包，也可以是包构建树中的目录(或者任何其他目录)，只要有所需文件就行。

**Windows**上，提供了两个注册表。用户注册表存储在**Windows**注册表的 `HKEY_CURRENT_USER` 键下，而系统包注册表存储在 `HKEY_LOCAL_MACHINE` 键下：

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<packageName>\  
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<packageName>\
```

对于给定的 `packageName`，每个条目都包含 `REG_SZ`。这个值是一个目录，其中可以找到该包的配置文件。**Unix**平台上，没有系统包注册表，只有用户包注册信息存储在用户的主目录下，并且在该点下的含义与**Windows**相同：

```
~/.cmake/packages/<packageName>/
```

**CMake**对于如何在平台上创建这些没提供什么帮助。对于已安装的包，不提供自动机制，但是 `export()` 可以在项目的**CMakeLists.txt**文件中使用，将项目构建树的一部分添加到用户注册中：

```
export(PACKAGE packageName)
```

这将指定的包添加到用户注册表中，并将其指向与 `export()` 位置相关联的二进制目录。然后，由项目来确保该目录中是否存在适当的配置文件。如果不存在合适的配置文件，并且对项目包进行 `find_package()`，注册项将自动删除(如果权限允许)。包注册表中的条目名称都是指向目录路径的MD5值，这避免了名称冲突，也是 `export(PACKAGE)` 所采用的命名策略。

从构建树向注册表添加位置非常危险。虽然 `export(PACKAGE)` 可以将一个位置添加到注册表中，但是除了手动删除注册项或从构建目录中删除包配置文件之外，没有相应的机制来删除它。使用 `export(PACKAGE)` 还可能对持续集成系统造成破坏，它使项目获得在同一构建从属上的另一构建树。防止这种情况的一种方法是将 `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` 变量设置为 `ON`，这将禁用所有 `export(PACKAGE)`，防止项目向用户注册表添加构建树。作为补充，项目可以将 `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` 或 `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` 设置为 `ON`，以使所有的 `find_package()` 忽略用户和系统注册表。

实际中，注册表并不经常使用。为添加和删除条目提供的帮助有限，维护注册中心在某种程度上是手动的。当通过主机的包管理系统安装包时，可以适当地将自己添加到系统或用户注册表中，然后包的卸载程序可以删除相同的条目。虽然包位置定义得很好，而且在概念上很容易定义，但很少有包费心去注册和注销。包可以通过各种不同的方式进入用户的机器，这使得健壮而简单地实现这种注册/注销有些困难。

### 23.5.2. FindPkgConfig

`find_package()` 通常是查找包并将其合并到**CMake**项目中的首选方法，但在某些情况下，结果不太理想。一些 `Find`模块还没有更新，并且没有提供导入目标，而是依赖于一个变量集合，项目必须手动处理这些变量。其他模块可能落后于最新的包版本，导致不兼容或提供了不正确的信息。

某些情况下，包支持 `pkg-config`，与 `find_package()` 类似的工具。如果 `pkg-config` 可用，那么 `PkgConfig` `Find` 模块可以用来读取该信息，并可以以更友好的方式提供。导入的目标可以自动创建，使项目不必手动处理各种变量。`pkg-config` 信息也与软件包的安装版本相匹配，因为信息通常由软件包提供。

`FindPkgConfig` 模块定位 `pkg-config` 可执行文件，并定义几个函数来使用它，以查找和提取有关具有 `pkg-config` 支持包的详细信息。如果模块找到了可执行文件，将 `PKG_CONFIG_FOUND` 变量设置为 `true`，将 `PKG_CONFIG_EXECUTABLE` 变量设置为工具的位置。`PKG_CONFIG_VERSION_STRING` 也设置为工具的版本(2.8.8之前的CMake版本除外)。

实际中，因为模块定义了两个函数包装了工具，所以项目很少需要使用 `PKG_CONFIG_EXECUTABLE` 变量。`pkg_check_modules()` 和 `pkg_search_module()` 两个函数接收完全相同的选项，并具有类似的行为。两者之间区别是，`pkg_check_modules()` 检查其参数列表中给出的所有模块，而 `pkg_search_module()` 在第一个满足条件的模块处停止。这里使用“模块”，而不是“包”，是因为命令的历史原因，可能会造成一些混淆，但与常规的 CMake 模块没有直接关系，基本上可以认为是包。

```
pkg_check_modules(prefix
  [REQUIRED] [QUIET]
  [IMPORTED_TARGET]
  [NO_CMAKE_PATH]
  [NO_CMAKE_ENVIRONMENT_PATH]
  moduleSpec1 [moduleSpec2...]
)

pkg_search_module(prefix
  [REQUIRED] [QUIET]
  [IMPORTED_TARGET]
  [NO_CMAKE_PATH]
  [NO_CMAKE_ENVIRONMENT_PATH]
  moduleSpec1 [moduleSpec2...]
)
```

函数的行为与 `find_package()` 类似，这里的 `REQUIRED` 和 `QUIET` 参数与 `find_package()` 具有相同的效果。CMake 3.1或更高版本中，`CMAKE_PREFIX_PATH`、`CMAKE_FRAMEWORK_PATH` 和 `CMAKE_APPBUNDLE_PATH` 都是搜索位置，`NO_CMAKE_PATH` 和 `NO_CMAKE_ENVIRONMENT_PATH` 关键字在这里也具有相同的含义。`PKG_CONFIG_USE_CMAKE_PREFIX_PATH` 可以更改默认行为，以决定是否考虑这些搜索位置(视为布尔开关，打开或关闭搜索位置)，但是项目应该避免使用，除非需要支持早于3.1的CMake。

只有CMake 3.6或更高版本才支持 `IMPORTED_TARGET` 选项。如果找到了请求的模块，则会创建名为 `PkgConfig::<prefix>` 的导入目标。这个导入的目标将对模块的.pc文件进行接口信息的填充，提供头文件搜索路径、编译器标志等内容。如果项目需要的最小CMake版本是3.6或更高，强烈建议使用这个选项。

这些函数需要一个或多个 `moduleSpec` 参数来定义搜索内容。它们将查找的包/模块的名称与任何版本需求结合起来。支持的方式有：

- `moduleName`
- `moduleName=version`
- `moduleName>=version`
- `moduleName<=version`

注意，缺少 `>` 或 `<` 比较运算符，唯一支持的不等式运算符是 `>=` 和 `<=`。如果不包括版本需求，则接受任何版本。返回时，函数通过使用适当的选项调用 `pkg-config`，从而在调用范围内设置一些变量，以提取包相关的详细信息：

变量	<b>pkg-config</b> 使用的选项
prefix_LIBRARIES	--libs-only-l
prefix_LIBRARY_DIRS	--libs-only-L
prefix_LDFLAGS	--libs
prefix_LDFLAGS_OTHER	--libs-only-other
prefix_INCLUDE_DIRS	--cflags-only-l
prefix_CFLAGS	--cflags
prefix_CFLAGS_OTHER	--cflags-only-other
prefix_STATIC_LIBRARIES	--static --libs-only-l
prefix_STATIC_LIBRARY_DIRS	--static --libs-only-L
prefix_STATIC_LDFLAGS	--static --libs
prefix_STATIC_LDFLAGS_OTHER	--static --libs-only-other
prefix_STATIC_INCLUDE_DIRS	--static --cflags-only-l
prefix_STATIC_CFLAGS	--static --cflags
prefix_STATIC_CFLAGS_OTHER	--static --cflags-only-other

当多个项目由一组选项返回时(例如多个库或多个搜索路径), 相应的变量将包含一个CMake列表。

只有满足模块需求时, 才设置上述变量。检查这一点的标准方法是使用 `prefix_FOUND` 和 `prefix_STATIC_FOUND`。对于 `pkg_check_modules()`, 必须满足所有 `moduleSpec` 要求, 才能使这些变量的值为true, 而 `pkg_search_module()` 只需要找到一个匹配的 `moduleSpec`。

对于 `pkg_check_modules()`, 成功找到模块时还会设置一些额外的模块变量。下文中, 如果只给出一个 `moduleSpec`, 则 `YYY = prefix`, 否则 `YYY = prefix_moduleName`。

#### `YYY_VERSION`

找到的模块版本, 可以从 `--modversion` 选项的输出中获取。

#### `YYY_PREFIX`

模块的前缀目录。通过查询 `prefix` 获得, 这个变量是大多数.pc文件定义的, 也是 `pkg-config` 在默认情况下提供。

#### `YYY_INCLUDEDIR`

查询 `includedir` 的结果。这是一个常见的, 但不必需的变量。

#### `YYY_LIBDIR`

查询 `libdir` 的结果。同样, 这是一个常见的, 但不必需的变量。

CMake 3.4及以后版本中, `FindPkgConfig` 模块提供了额外的功能, 可以用来从.pc文件中提取任意变量:

```
pkg_get_variable(resultVar moduleName variableName)
```

`pkg_check_modules()` 在内部使用它来查询前缀、`includedir` 和 `libdir` 变量, 项目可以用来查询任意变量的值。

对于大多数系统，`FindPkgConfig` 模块提供的功能工作非常可靠。然而，这些函数的实现依赖于 `pkg-config` 版本0.20.0中引入的特性。一些较旧的系统(例如：**Solaris 10**)带有较旧的 `pkg-config`，这导致所有对 `FindPkgConfig` 不能找到任何模块，并且没有记录任何错误消息来提示这是因为 `pkg-config` 的版本太旧。

## 23.6. 总结

从CMake 3.0开始，已经有意识地转向使用导入的目标来表示外部库和程序，而不是变量。这允许将库和程序视为一个单位，而不仅是收集相关二进制文件的位置。但在库的情况下，相关的头文件的搜索路、编译器定义和库的依赖关系，以及目标也是导入目标的一部分。这使得项目使用外部库和程序时，就像项目定义的任何目标一样容易。这种关注点的转移意味着寻找包已经变得比寻找单个文件、路径等重要得多，而且越来越多的项目可以作为包来使用。寻找单独的文件仍然有用，而且有助于理解如何做到这一点，开发人员应该把它看作是包和/或导入目标的铺垫。可能的情况下，最好查找包，而不是包中的单个内容。

查找包时，出现的复杂情况与多个版本安装在不同位置有关。用户可能不知道所有已安装的版本，与其让项目预测这种情况，不如跳过默认的搜索路径，让用户通过缓存或环境变量提供相应的路径。`CMAKE_PREFIX_PATH` 是最方便的方法，因为CMake会自动搜索列出的每个前缀路径下的公共目录范围。

除了 `find_package()` 外，所有的 `find_...` 都以类似的方式工作，缓存成功后，会避免在下次 `find_...` 查找相同的内容时，再重复查找操作，甚至可以跨多个CMake调用缓存。因为在项目中有很多这样的 `find_...`，考虑到每个调用可能搜索大量的位置和目录条目，缓存机制可以节省大量时间。开发人员需要注意，当 `find_file()`、`find_path()`、`find_program()` 或 `find_library()` 调用成功则停止搜索，即使运行该命令会返回不同的结果，或者之前找到的实体不存在。如果删除了实体，则会导致构建错误，只有从缓存中删除过期的条目才能纠正这些错误。开发人员只需要删除整个缓存，然后重新从头开始构建，而不是试图找出哪些缓存变量需要删除。另一个方面是，开发人员应该注意，如果对这些 `find_...` 的调用未能找到所需的实体，那么即使在相同的项目中，也会对每个调用重复搜索。CMake会缓存成功的调用。如果一个项目有很多这样的调用，这可能会减慢配置步骤的时间。因此，开发人员应该仔细考虑项目如何使用 `find_...`，以增加成功搜索的可能性和数量。

`find_package()` 的情况复杂一些。如果通过Find模块找到包，很可能上述所有关注点都将应用于包，因为逻辑很可能构建在其他 `find_...` 之上。但是，如果通过配置模式找到了包，`find_package()` 将缓存成功的结果，并在后续调用时首先检查该位置。如果包在该位置不再有适当的配置文件，则继续执行正常的搜索逻辑。配置模式的这种独特行为更加健壮。

对于持续集成系统，`find_...` 结果的缓存可能导致一些问题。如果保留前一次运行的CMake缓存的地方使用增量构建，在项目中对其搜索方式的更改可能不会反映在构建中。只有当CMake缓存清除时，更改才会生效。缓存通常没有记录关于所找实体的详细信息，因此构建输出很少提供关于使用旧搜索详细信息的线索。因此，可能会要求所有的CI构建从头开始，但这对于耗时较长的构建不可行。可以帮助减少这个问题的策略是，在CI负载低的时候安排每日构建工作，这个时候清除构建树，按照正常的方式构建项目。这将保持常规的增量行为，并且通常会使任何缓存相关的问题得到解决。这种策略的有效性在一段时期内会降低，变更在一个分支上进行，CI构建在该分支和其他分支之间交替进行，但人们希望这样的情况是短期的，只要开发人员意识到这段时期的起因，就可以容忍带来的后果。

`find_package()` 的包注册特性应该谨慎处理。它们可能会给持续集成系统带来意想不到的结果，其中项目可能希望找到同样在一台机器上构建的包。不幸的是，没有环境变量可以禁用注册，项目本身可以通过设置 `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` 来进行(CI工作通常不会修改系统注册表所需的权限，所以设置 `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` 是不必要的)。实际中，很少有项目写入到注册表中，所以除非知道项目可能使用CI，否则不需要添加这个CMake变量。项目还应该避免在CI中使用 `export(PACKAGE)` (有争议的是，通常应该避免这样的使用)。

只有在 `find_package()` 不适合的情况下才能使用 `FindPkgConfig` 模块。通常，这是针对CMake提供查找模块的包，但查找模块相当旧，不提供导入目标。`FindPkgConfig`模块对于搜索CMake不知道的包，以及包没有提供自己的CMake配置文件时很有用(提供了 `pkg-config` (即`.pc`)文件)。

使用工具链文件进行交叉编译时，最好设置 `CMAKE_SYSROOT` 而不是 `CMAKE_FIND_ROOT_PATH`。虽然两者以相同的方式影响 `find_…()` 的搜索路径，但只有 `CMAKE_SYSROOT` 能保证编译器和链接器标志，以便头文件的包含和库链接能够正确工作。

交叉编译场景中，搜索程序通常希望找到主机上运行的二进制文件，而搜索文件和库通常希望找到目标文件。因此，通常会在工具链文件中看到以下默认强制执行的行为：

```
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

有人可能会说，这应该在项目中设置，而不是依赖于在工具链文件中的设置，从技术上讲，开发人员可以自由地使用任何工具链文件，而且项目会隐式地依赖默认行为，然后选择是否覆盖这些行为。这里的复杂性是对于每个 `project()` 或 `enable_language()` 都要重新读取工具链文件，如果项目想要强制执行特定的组合，就必须在每次这样的调用之后执行。因此，合理的折衷是让项目在第一个 `project()` 之前包含上面的块，并且让工具链编写器也包含它。如果工具链作者没有包含这样的块，至少项目能得到合理的默认值，如果工具链文件将默认值更改为其他东西，至少在整个项目中能一致地应用，所以开发人员在使用上面示例以外的设置时应该非常谨慎。

面对不需要重新运行CMake就可以在设备和模拟器构建之间切换的情况(例如在iOS项目中使用Xcode)时，需要避免使用 `find_library()`。通过该命令获得的任何结果只能指向设备或模拟器库中的一个，而不能同时指向两者。这种情况下，可以添加底层链接器标志，通过名称而不是通过路径链接，例如 `-framework ARKit` 或 `-lz`。如果在默认链接器搜索路径中找不到框架或库，项目还需要提供链接器选项来扩展搜索路径，以便能够找到它们。

对于使用 `CMAKE_MODULE_PATH` 或 `CMAKE_PREFIX_PATH` 控制CMake搜索内容的位置，一些在线示例和博客文章中经常出现冲突的情况。一个简单的区别是CMake只在搜索 `FindXXX.cmake` 时使用 `CMAKE_MODULE_PATH`。对于其他所有操作(包括搜索配置文件)使用 `CMAKE_PREFIX_PATH`。

# 第24章：测试

构建项目后的操作是测试创建的工程。**CMake**软件套件包括**CTest**工具，可用于自动化测试，可以完成配置、构建、测试，提交到仪表盘的整个过程。本章首先介绍了如何使用**CMake**来定义测试，并使用**ctest**工具来执行测试。自动化整个配置-构建-测试过程，本章后面将讨论其中使用到技术。

## 24.1. 定义和执行一个简单的测试

**CMake**项目中定义测试的第一步是在顶级**CMakeLists.txt**文件中使用 `enable_testing()`。这个函数的作用是让**CMake**在 `CMAKE_CURRENT_BINARY_DIR` 中生成**CTest**输入文件，其中包含项目中所有测试的详细信息(更准确地说，是当前目录范围内定义的测试)。可以在子目录中调用 `enable_testing()`，但是如果不在顶层，**CTest**输入文件不会在构建树的顶层创建。

使用 `add_test()` 定义测试：

```
add_test(NAME testName
         COMMAND command [arg...]
         [CONFIGURATIONS config1 [config2...]]
         [WORKING_DIRECTORY dir]
     )
```

这个命令添加了一个名为 `testName` 的测试，使用给定的参数运行命令。默认情况下，如果返回的退出码为0，测试则为通过。更灵活的通过/失败结果支持，将在下一节中讨论。

该命令是可执行文件的完整路径，也可以是项目中定义的可执行目标的名称。使用目标名称时，**CMake**将自动替换可执行文件的实际路径。这在使用多配置生成器(如**Xcode**或**Visual Studio**)时特别有用。下面展示一个项目示例：

```
cmake_minimum_required(VERSION 3.0)
project(CTestExample)
enable_testing()

add_executable(testapp testapp.cpp)
add_test(NAME noArgs COMMAND testapp)
```

使用目标的实际位置自动替换，并不会扩展到命令参数，只有命令本身支持这种替换。如果需要将目标位置作为命令行参数提供，可以使用生成器表达式。例如：

```
add_executable(app1 ...)
add_executable(app2 ...)

add_test(NAME withArgs COMMAND app1 ${TARGET_FILE:app2})
```

运行测试时，用户可以指定测试哪个配置。当项目使用单个配置生成器时，配置不必匹配生成类型。特别是，如果没有提供配置，则假设配置为空。如果没有可选配置关键字 `CONFIGURATIONS`，不管构建类型或用户请求配置是什么，程序将测试所有配置。如果给定了 `CONFIGURATIONS` 关键字，则仅对列出相应配置的测试。请注意，空配置仍然认为有效，因此要在该场景中运行测试，空字符串是 `CONFIGURATIONS` 的内容之一。

例如，要添加一个只对具有调试信息配置执行的测试，可以列出**Debug**和**RelWithDebInfo**的配置。添加空字符串也会在没有指定配置时运行：

```
add_test(NAME debugOnly
COMMAND testapp
CONFIGURATIONS Debug RelWithDebInfo "")
)
```

大多数情况下，不需要配置关键字 `CONFIGURATIONS`，测试将对所有配置执行，包括空配置。

默认情况下，测试将在 `CMAKE_CURRENT_BINARY_DIR` 目录中运行，可以用 `WORKING_DIRECTORY` 选项指定测试运行位置。在不同的目录中运行相同的可执行文件，以获得不同的输入文件集，从而不必指定相应的命令行参数。

```
add_test(NAME foo
COMMAND testapp
WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/foo
)
add_test(NAME bar
COMMAND testapp
WORKING_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/bar
)
```

如果指定工作目录，请使用绝对路径。如果给出了相对路径，将解释为相对于启动`ctest`的目录，不是构建树的顶部。为了确保工作目录的正确性，应该避免使用相对路径 `WORKING_DIRECTORY`。

如果在运行测试时指定的工作目录不存在，**CMake**版本3.11和更早版本将不会发出错误消息，仍然会运行测试。**CMake** 3.12及以后版本将捕获错误，并将测试视为失败。不管使用什么版本的**CMake**，确保工作目录存在并具有适当的权限是项目的责任。

由于向后兼容，还支持简化的 `add_test()`：

```
add_test(testName command [args...])
```

因为缺少全名，以及命令形式有些特殊，这种形式不该在新项目中使用。主要是不支持生成器表达式，如果 `command` 是目标的名称，**CMake**将不会自动替换为二进制文件的位置。

使用`ctest`命令行工具运行测试，通常从构建目录的顶部运行。当不带命令行参数运行时，将执行已定义的测试，并在每个测试启动和完成时记录状态，并隐藏所有测试输出。测试的输出将在最后打印：

```
Test project /path/to/build/dir
Start 1: fooWithBar
1/2 Test #1: fooWithBar..... Passed 0.00 sec
Start 2: fooWithoutBar
2/2 Test #2: fooWithoutBar..... Passed 0.00 sec
100% tests passed, 0 tests failed out of 2
Total Test time (real) = 0.02 sec
```

如果使用**Xcode**或**Visual Studio**这样的多配置生成器，`ctest`需要了解应该测试哪个配置。可以通过 `-c configType` 来实现，其中 `configType` 是支持的构建类型(Debug、Release等)。对于单配置生成器，`-c` 选项不是必须的，因为构建只能生成一种配置。

`ctest`用 `-v` 选项显示所有测试输出和关于运行的其他信息。`-vv` 和 `-vvv` 显示的信息冗余度越高，但通常只有研发`ctest`的开发人员才需要。即使是 `-v` 级别的信息，也比希望看到的信息更详细，并且用户可能只对失败的测试输出感兴趣。通过 `--output-on-failure` 选项，可以告诉`ctest`只显示失败测试的输出。开发人员可以将 `CTEST_OUTPUT_ON_FAILURE` 环境变量设置为任意值，以避免每次都指定它(该值没使用时，`ctest`只检查是否设置了 `CTEST_OUTPUT_ON_FAILURE`)。

默认情况下，每个测试将与ctest相同的环境中运行。如果测试需要更改环境，可以通过 ENVIRONMENT 测试属性来完成。此属性预期为 NAME=VALUE 的列表，用于定义在运行测试之前要设置的环境变量。更改仅针对该测试，不影响其他测试。

```
set_tests_properties(fooWithoutBar PROPERTIES
    ENVIRONMENT "FOO=bar;HAVE_BAZ=1"
)
```

环境变量需要修改，而替换现有值就不那么简单了。如果环境应该基于运行CMake的环境而不是ctest，那么 \$ENV{SOMEVAR} 可以用取现有值。一个例子就是在Windows上增加 PATH 环境变量，以确保测试可以找到需要链接的动态库：

```
# In this example, algo is assumed to be a shared library defined elsewhere
# in the project and whose binary will be in a different directory to fooTest
add_executable(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)
add_test(NAME fooWithAlgo COMMAND fooTest)
if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
        "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

用于调用ctest的实际环境修改环境更为复杂，通常也不严格。可以通过 cmake -E env 调用脚本的组合来实现， cmake -E env 将CMake提供的位置作为变量传递给 cmake -E env，然后该脚本使用这些值，调用测试可执行文件来执行扩充运行时环境的任务。这样的方法复杂，也很脆弱，除非没办法避免，否则应该避免使用。

为了方便IDE启用测试，CMake定义了一个自定义构建目标，该目标使用一组默认参数调用ctest。对于像 Xcode 和 Visual Studio 这样的多配置生成器，这个目标为 RUN\_TESTS，通过选择的构建类型作为ctest的配置。对于单配置生成器，目标简单地称为 test，并且在调用ctest时不指定任何配置。使用 RUN\_TESTS 或测试目标时，没有工具指定执行哪些测试，也没有任何其他传递给ctest的自定义选项。

## 24.2. 通过/失败标准和其他结果

根据测试命令的退出代码来确定测试结果，会有很大的限制，替代方法是指定正则表达式来匹配测试输出。 PASS\_REGULAR\_EXPRESSION 测试属性可用于指定一个正则表达式列表，测试输出至少匹配其中一个才能通过测试。这些正则表达式经常跨多行。类似地， FAIL\_REGULAR\_EXPRESSION 测试属性可以设置为正则表达式列表。如果其中任何一个匹配测试输出，即使输出也匹配 PASS\_REGULAR\_EXPRESSION 或退出代码为0，测试状态也为失败。测试可以同时拥有 PASS\_REGULAR\_EXPRESSION 和 FAIL\_REGULAR\_EXPRESSION 集。如果设置了 PASS\_REGULAR\_EXPRESSION (不为空)，则在确定测试通过或失败时不考虑退出码的状态。

```
# Ignore exit code, check output to determine the pass/fail status
set_tests_properties(fooTest PROPERTIES
    PASS_REGULAR_EXPRESSION
        "Checking some condition for fooTest: passed
        +.*
        All checks passed"
    FAIL_REGULAR_EXPRESSION "warning|Warning|WARNING"
)
```

有时需要跳过某个测试，可能只有测试本身可以确定。可以将 SKIP\_RETURN\_CODE 测试属性设置为测试返回的值，以指定跳过而不是失败。使用 SKIP\_RETURN\_CODE 退出的测试将覆盖其他通过/失败标准。

### *fooTest.cpp*

```
int main(int argc, char* argv[])
{
    if (shouldSkip())
        return 2; // Skipped

    if (runTest())
        return 0; // Passed

    return 1; // Failed
}
```

### *CMakeLists.txt*

```
add_executable(fooTest fooTest.cpp ...)
add_test(NAME foo COMMAND fooTest)

set_tests_properties(foo PROPERTIES
    SKIP_RETURN_CODE 2
)
```

以上测试的输出:

```
Test project /path/to/build/dir
Start 1: foo
1/1 Test #1: foo .....***Skipped 0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.01 sec

The following tests did not run:
  1 - foo (Skipped)
```

当有测试失败或由于某种原因没有运行时，将在末尾打印所有测试的状态。通过返回代码指示跳过的测试不会认为是失败，但仍然计算在测试总数中。测试可能会因为其他原因而跳过，比如依赖项不满足。

**CMake 3.9**或更高版本中，还支持禁用的测试属性。可用于将测试标记为临时禁用，这允许对其进行定义，但不执行，甚至不计入测试总数。并且不会认为是测试失败，但仍然会在测试结果中显示，并带有适当的状态信息。这类测试不应该长时间内保持禁用状态，该特性是用来临时禁用有问题的或不完整的测试，直到问题修复为止。

下面的简单示例演示了禁用测试的方法:

```
add_test(NAME fooWithBar ...)
add_test(NAME fooWithoutBar ...)
set_tests_properties(fooWithoutBar PROPERTIES DISABLED YES)
```

上面的`ctest`输出:

```
Test project /path/to/build/dir
Start 1: fooWithBar
1/2 Test #1: fooWithBar ..... Passed 0.00 sec
Start 2: fooWithoutBar
2/2 Test #2: fooWithoutBar .....***Not Run (Disabled) 0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.01 sec

The following tests did not run:
  2 - fooWithoutBar (Disabled)
```

某些情况下，测试可能会失败。与禁用测试相比，将测试标记为预期失败可能更合适，这样它就可以继续执行。可以将 `WILL_FAIL` 测试属性设置为 `true` 来表示这一点。这样做的另一个好处是，如果测试开始通过了，`ctest` 会认为这个测试失败了，开发人员会立即意识到行为上的变化。

测试通过/失败的另一个方面是完成时长。如果设置了 `TIMEOUT` 测试属性，则指定在终止测试并标记为失败之前，允许测试运行的秒数。`ctest` 还接受 `--timeout` 选项，对任何没有设置 `TIMEOUT` 属性的测试具有相同的效果。此外，还可以通过为 `ctest` 指定 `--stop-time` 选项，将时限应用于测试集。停止时间之后的参数必须是实时时间，而不是秒数，如果没有给出时区，则假定为本地时间。

```
add_test(NAME t1 COMMAND ...)
add_test(NAME t2 COMMAND ...)
set_tests_properties(t2 PROPERTIES TIMEOUT 10)
```

```
ctest --timeout 30 --stop-time 13:00
```

上面的示例中，每个测试的默认超时设置为 30 秒。因为 `t1` 没有设置 `TIMEOUT` 属性，所以它的超时时间为 30 秒，而 `t2` 的 `TIMEOUT` 属性设置为 10，这将覆盖 `ctest` 的默认设置。测试将在当地时间下午 1 点前完成。

某些情况下，测试可能需要等待特定的条件满足才开始测试。满足该条件并开始测试之后，可能需要对运行的部分应用超时。`CMake 3.6` 或更高版本中，`TIMEOUT_AFTER_MATCH` 测试属性可用来支持这种行为。需要包含两个条目的列表，第一个是满足条件后作为超时使用的秒数，第二个是正则表达式，用于匹配测试输出。找到正则表达式后，将重置测试的超时倒计时和开始时间，并将超时值设置为第一项。例如，下面将对测试应用 30 秒超时，一旦字符串条件在测试输出中出现，测试将有 10 秒的时间来完成，而原来的 30 秒超时将不再适用。

```
set_tests_properties(t2 PROPERTIES
  TIMEOUT 30
  TIMEOUT_AFTER_MATCH "10;Condition met"
)
```

如果测试需要 25 秒来满足条件，那么测试的总时间可能长达 35 秒，但是因为测试的开始时间进行了重置，所以 `ctest` 将报告 0 到 10 秒之间的时间（即满足条件所花费的时间不统计）。另一方面，如果在 30 秒内没有满足条件，测试将总体测试时间显示为 30 秒。

可能的情况下，应该避免使用 `TIMEOUT_AFTER_MATCH`，以便其他方法处理前置条件。第 24.5 节和第 24.4 节将讨论更好的替代方案。

## 24.3. 测试分组与选择

较大的项目中，运行子测试集很正常。开发人员可能只关注特定的失败测试，而对其他测试不感兴趣。只执行测试的特定子集的一种方法是提供 `-R` 和 `-E` 选项。这些选项指定一个正则表达式来匹配测试名称。`-R` 选择要包含在测试集中的测试，而 `-E` 则是排除，可以指定这两个选项进行组合。

```
add_test(NAME fooOnly COMMAND ...)
add_test(NAME barOnly COMMAND ...)
add_test(NAME fooWithBar COMMAND ...)
add_test(NAME fooSpecial COMMAND ...)
add_test(NAME other_foo COMMAND ...)
```

```
ctest -R Only # Run just fooOnly and barOnly
ctest -E Bar # Run all but fooWithBar
ctest -R '^foo' -E fooSpecial # Run all tests starting with foo except fooSpecial
ctest -R 'fooSpecial|other_foo' # Run only fooSpecial and other_foo
```

编写正则表达式来捕获所需的测试并不容易，开发人员可能只想查看所有测试而不运行。`-N` 选项只打印测试而不运行，可以用来检查正则表达式是否能提供所需测试集。

```
ctest -N

Test project /path/to/build/dir
Test #1: fooOnly
Test #2: barOnly
Test #3: fooWithBar
Test #4: fooSpecial
Test #5: other_foo

Total Tests: 5
```

```
ctest -N -R 'fooSpecial|other_foo'

Test project /path/to/build/dir
Test #4: fooSpecial
Test #5: other_foo

Total Tests: 2
```

添加测试会给定测试号，测试号在运行时保持不变，除非在项目中添加或删除一个测试。当使用 `-N` 时，按照项目定义的顺序列出测试，但不一定按照该顺序执行测试。可以通过测试号，而不是使用 `-I` 来选择要运行的测试。此方法相当脆弱，因为添加或删除单个测试，可能会改变分配给其他测试的数量。即使通过 `-c` 将不同的配置传递给 `ctest`，也会导致测试编号的更改。大多数情况下，最好按名称匹配。

两个测试的名称完全相同时，测试号可能会有用。如果在同一目录中定义，两个测试都会运行，不会发出任何警告。虽然重复的测试名称应该避免，但在涉及外部提供的分层测试项目中，总会碰到这样的情况。

`-I` 期望参数具有某种复杂的形式。最直接的是在命令行上指定测试号，用逗号分隔，不带空格：

```
ctest -I [start[,end[,stride[,testNum[,testNum...]]]]]
```

要指定单独的测试号，开始、结束和跨越可以保持空白，如下所示：

```
ctest -I ,,,3,2 # Selects tests 2 and 3 only
```

可以从文件中读取相同的细节，而不必在命令行中将文件的名称指定给 `-I` 来指定。如果运行相同复杂度的测试集，并且没有添加或删除测试，`-I` 就非常有用了：

`testNumbers.txt`

```
,,3,2
```

```
ctest -I testNumbers.txt
```

如果执行大量的测试，按名称或编号单独选择就会变得很麻烦。可以使用 `LABELS` 测试属性为测试分配标签列表，根据标签选择测试。`-L` 和 `-LE` 类似于 `-R` 和 `-E`，只是操作的是测试标签而不是测试名称。继续前面的示例，定义相同的测试：

```
set_tests_properties(fooOnly PROPERTIES LABELS "foo")
set_tests_properties(barOnly PROPERTIES LABELS "bar")
set_tests_properties(fooWithBar PROPERTIES LABELS "foo;bar;multi")
set_tests_properties(fooSpecial PROPERTIES LABELS "foo")
set_tests_properties(other_foo PROPERTIES LABELS "foo")
```

```
ctest -L bar

Test project /path/to/build/dir
Start 2: barOnly
1/2 Test #2: barOnly ..... Passed 1.52 sec
Start 3: fooWithBar
2/2 Test #3: fooWithBar ..... Passed 1.02 sec

100% tests passed, 0 tests failed out of 2

Label Time Summary:
bar = 2.53 sec*proc (2 tests)
foo = 1.02 sec*proc (1 test)
multi = 1.02 sec*proc (1 test)

Total Test time (real) = 2.54 sec
```

标签不仅可以方便地对测试进行分组，还可以对执行时间统计进行分组。如上面的示例输出，当测试集中的测试都设置了 `LABELS` 属性时，`ctest` 命令会打印标签信息。这允许开发人员了解每个标签组占总测试时间的情况。`sec*proc` 的 `proc` 部分是指处理器数量(24.4节中描述)。运行3秒需要4个处理器的测试，报告值为12。可以使用 `--no-label-summary` 禁用标签的时间摘要。

重新运行上次失败的测试，可以在进行修复后重新检查相关的测试。`ctest` 命令支持 `--rerun-failed`，该选项不需要给出任何测试名称、数字或标签。

有时，某个测试或一组测试只是间歇性地失败，因此需要运行多次测试，以重现失败。与反复运行 `ctest` 不同，可以使用 `--repeat-until-fail` 提供每个测试可以重复的次数上限。如果测试失败，`ctest` 将不会再重新运行这个测试。

```

ctest -L bar --repeat-until-fail 3

Test project /path/to/build/dir
Start 2: barOnly
Test #2: barOnly ..... Passed 1.52 sec
Start 2: barOnly
Test #2: barOnly .....***Failed 0.00 sec
Start 3: fooWithBar
Test #3: fooWithBar ..... Passed 1.02 sec
Start 3: fooWithBar
Test #3: fooWithBar ..... Passed 1.02 sec
Start 3: fooWithBar
2/2 Test #3: fooWithBar ..... Passed 1.02 sec

50% tests passed, 1 tests failed out of 2

Label Time Summary:
bar = 1.02 sec*proc (2 tests)
foo = 1.02 sec*proc (1 test)
multi = 1.02 sec*proc (1 test)

Total Test time (real) = 4.59 sec

The following tests FAILED:
  2 - barOnly (Failed)
Errors while running CTest

```

标签标签不会累积重复测试的总时间，只使用测试最后一次执行的时间。但是，总测试时间会计算所有重复的次数。

## 24.4. 并行执行

对于大型项目或者测试需要大量时间的项目来说，最大化测试吞吐量是一个重要的因素。并行运行测试是 `ctest` 的关键特性，可以使用与标准 `make` 工具非常相似的命令行选项来启用。可以使用 `-j` 指定同时运行测试数量的上限。与大多数 `make` 实现不同的是，必须提供一个值，否则选项将不起作用。另一种方法是，`CTEST_PARALLEL_LEVEL` 环境变量可以用来指定作业的数量，但如果同时使用这两个变量，则命令行选项优先。这种安排对于持续集成服务器特别有用，因为 `CTEST_PARALLEL_LEVEL` 可以设置为服务器上的 CPU 核数，从而使项目不必自己计算并行任务的数量。对于需要限制并行作业数量的项目，仍然可以使用 `-j` 覆盖 `CTEST_PARALLEL_LEVEL`。

相关的选项是 `-1`，用于指定 CPU 负载的上限。如果负载超过此限制，`ctest` 将停止新的测试。不过，这个选项的缺点也很明显。通常，`ctest` 最初会启动 `-j` 或 `CTEST_PARALLEL_LEVEL` 设置所允许的任务限制范围内的测试，超过 `-1` 指定的限制。测量的 CPU 负载通常有延迟，所以在测量负载增加之前开启了太多的测试。为了防止这种情况，应该将 `-j` 或 `CTEST_PARALLEL_LEVEL` 指定的并行任务数量设置为不超过 `-1` 的限制。如果既没有设置 `-j`，也没有设置 `CTEST_PARALLEL_LEVEL`，`-1` 将不起作用。尽管存在这些限制，但 `-1` 在帮助减少共享系统上的 CPU 过载时仍然有用，因为其他进程可能会竞争 CPU 资源。

默认情况下，假定每个测试消耗一个 CPU。对于使用多个 CPU 的测试用例，可以设置 `PROCESSORS` 测试属性，表示预期使用多少个 CPU。开始测试之前，将使用该值来确定是否有足够的 CPU 资源可用。如果 `PROCESSORS` 设置为高于任务限制的值，在确定是否可以启动测试时，将设置为任务的限制。

这些选项的效果可以在下面的示例输出中看到，这里使用的是之前的测试集。

```

ctest -j 5

Test project /path/to/build/dir
Start 5: other_foo
Start 2: barOnly
Start 3: fooWithBar
Start 1: fooOnly
Start 4: fooSpecial
1/5 Test #4: fooSpecial ..... Passed 0.12 sec
2/5 Test #1: fooOnly ..... Passed 0.52 sec
3/5 Test #3: fooWithBar ..... Passed 1.01 sec
4/5 Test #2: barOnly ..... Passed 1.52 sec
5/5 Test #5: other_foo ..... Passed 2.02 sec
100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar = 2.53 sec*proc (2 tests)
foo = 1.65 sec*proc (3 tests)
multi = 1.01 sec*proc (1 test)

Total Test time (real) = 2.03 sec

```

定义了5个测试，命令行上的任务限制为5，因此能够立即启动所有测试。每个测试的结果在完成时记录，而不是以它们的启动顺序。将任务限制减少到2显示如下输出：

```

ctest -j 2

Test project /path/to/build/dir
Start 5: other_foo
Start 2: barOnly
1/5 Test #2: barOnly ..... Passed 1.52 sec
Start 3: fooWithBar
2/5 Test #5: other_foo ..... Passed 2.01 sec
Start 1: fooOnly
3/5 Test #1: fooOnly ..... Passed 0.52 sec
Start 4: fooSpecial
4/5 Test #3: fooWithBar ..... Passed 1.02 sec
5/5 Test #4: fooSpecial ..... Passed 0.12 sec

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar = 2.54 sec*proc (2 tests)
foo = 1.65 sec*proc (3 tests)
multi = 1.02 sec*proc (1 test)

Total Test time (real) = 2.66 sec

```

对于大量的测试和较高的任务限制，很难跟踪单独测试的情况，运行结束时的测试总结变得更加重要。

测试有时需要确保没有其他测试与它们并行运行。可能正在执行对机器上的其他活动敏感的操作，或者会干扰其他测试的条件。要进行约束，可以将测试的 `RUN_SERIAL` 属性设置为`true`。这是相当严格的约束，可能对测试吞吐量有很大的影响，所以应该有节制地使用。更好的替代方法是使用 `RESOURCE_LOCK` 测试属性，用于提供测试需要独占访问的资源列表。这些资源是字符串，`ctest`不以任何方式解释它们，除非 `RESOURCE_LOCK` 属性中有这些资源，从而使得其他测试不会同时使用相应的资源。这种方式可以解决需要独占访问（例如数据库、共享内存）的测试，而不会阻塞与该资源不相关的测试。

```
set_tests_properties(fooOnly fooSpecial other_foo PROPERTIES RESOURCE_LOCK foo)
set_tests_properties(barOnly PROPERTIES RESOURCE_LOCK bar)
set_tests_properties(fooWithBar PROPERTIES RESOURCE_LOCK "foo;bar")
```

下面的示例输出显示，尽管作业限制为5允许同时执行所有测试，但ctest延迟启动了某些测试，直到所需的资源可用为止。

```
ctest -j 5

Test project /path/to/build/dir
Start 5: other_foo
Start 2: barOnly
1/5 Test #2: barOnly ..... Passed 1.52 sec
2/5 Test #5: other_foo ..... Passed 2.02 sec
Start 3: fooWithBar
3/5 Test #3: fooWithBar ..... Passed 1.01 sec
Start 1: fooOnly
4/5 Test #1: fooOnly ..... Passed 0.52 sec
Start 4: fooSpecial
5/5 Test #4: fooSpecial ..... Passed 0.12 sec

100% tests passed, 0 tests failed out of 5

Label Time Summary:
bar = 2.53 sec*proc (2 tests)
foo = 1.65 sec*proc (3 tests)
multi = 1.01 sec*proc (1 test)

Total Test time (real) = 3.67 sec
```

## 24.5. 测试依赖

测试不仅用于验证特定条件，还可用于执行这些条件。例如，测试可能需要连接服务器，以便验证客户机实现。不必依赖开发人员来确定服务器的可用性，可以创建另一个测试用例来确保服务器正在运行。然后，客户端测试需要对服务器进行某种依赖性，以确保测试以正确的顺序运行。

DEPENDS 测试属性允许在测试运行前完成的其他测试，从而对表示进行约束。以上“客户机/服务器”示例可以简单地表示为：

```
set_tests_properties(clientTest1 clientTest2 PROPERTIES DEPENDS startServer)
set_tests_properties(stopServer PROPERTIES DEPENDS "clientTest1;clientTest2")
```

DEPENDS 测试属性的缺点是，定义测试顺序时，不考虑先决测试是通过还是失败。上面的示例中，如果 `startServer` 测试用例失败，仍然会运行 `clientTest1`、`clientTest2` 和 `stopServer` 测试。这些测试很可能失败，测试将显示所有测试都失败，而实际上只有 `startServer` 失败，其他测试应该跳过。

CMake 3.7增加了对测试固件的支持，这个概念允许表达测试之间的依赖关系。测试可以通过 `FIXTURES_REQUIRED` 测试属性列出该固件的名称，来表示需要某个特定的固件。`FIXTURES_SETUP` 测试属性中具有相同固件名称的任何测试必须成功完成，然后才会启动相关测试。如果某个固件的设置测试失败，则需要该固件的所有测试都标记为跳过。类似地，测试可以在 `FIXTURES_CLEANUP` 测试属性中列出固件，以指示它必须在其他测试之后运行，该测试在 `FIXTURES_SETUP` 或 `FIXTURES_REQUIRED` 属性中有相同的固件。这些清理测试不需要通过设置或修复程序的测试，因为即使早期测试失败，也可能需要清理。

所有三个与固件相关的测试属性都接受固件名称列表。这些名称是任意的，不必与所使用的测试名称、资源或任何其他属性相关。固件名称能让开发人员清楚它们表示什么，因此通常与 `RESOURCE_LOCK` 属性使用的值相同。

考虑前面的“客户机/服务器”示例。这可以用具有以下属性的固件表示：

```
set_tests_properties(startServer PROPERTIES FIXTURES_SETUP server)
set_tests_properties(clientTest1 clientTest2 PROPERTIES FIXTURES_REQUIRED server)
set_tests_properties(stopServer PROPERTIES FIXTURES_CLEANUP server)
```

上面的例子中，`server` 是固件的名称，`clientTest1` 和 `clientTest2` 会在 `startServer` 后和 `stopServer` 前运行。如果启用了并行执行，`startServer` 将首先运行，两个客户机测试将同时运行，而 `stopServer` 将在两个客户机测试都完成或跳过后运行。

固件的另一个好处是，可以让开发人员只运行测试的子集。考虑这样一个场景：开发人员正在处理 `clientTest2`，但对运行 `clientTest1` 不感兴趣。当使用 `DEPENDS` 表示测试的依赖关系时，开发人员负责确保在测试集中包括所需的测试，这需要理解所有相关的依赖关系。示例如下所示：

```
ctest -R "startServer|clientTest2|stopServer"
```

使用固件时，`ctest` 会自动设置或清理要添加到要执行测试集的测试，以满足固件需求。这意味着开发者只需要指定关注的测试，就可以把依赖关系交给 `ctest` 处理：

```
ctest -R clientTest2
```

使用 `--rerun-failed` 时，相同的机制将设置和清理要自动添加到测试集中的测试，以满足之前失败测试的固件依赖关系。

固件可以有零个或多个安装测试和零个或多个清理测试。固件可以定义设置测试，而不定义清理测试，反之亦然。固件可以不包含设置或清理测试。这种情况下，固件对要执行的测试或测试将运行的时间没有影响。固件可以有与之关联的设置和/或清理测试，但不进行测试。开发过程中定义测试或禁用测试时，可能会出现这些情况。对于不需要测试的情况，CMake 3.7 中的 Bug 允许固件的清理测试在安装测试前运行，这个 Bug 在 3.8.0 版本中得到了修复。

下面的示例演示了，如何使用固件来表示更复杂的测试依赖关系。扩展前面的示例，假设一个客户机测试只需要一个服务器，而另一个客户机测试需要服务器和数据库都可用。可以定义两个固件来表示：`server` 和 `database`。对于后者，可以简单地检查是否有可用的数据库，如果没有可用的数据库则失败，因此数据库固件不需要清理测试。服务器和数据库固件不相关，因此不需要依赖关系。这些可以这样表达：

```
# Setup/cleanup
set_tests_properties(startServer PROPERTIES FIXTURES_SETUP server)
set_tests_properties(stopServer PROPERTIES FIXTURES_CLEANUP server)
set_tests_properties(ensureDbAvailable PROPERTIES FIXTURES_SETUP database)

# Client tests
set_tests_properties(clientNoDb PROPERTIES FIXTURES_REQUIRED server)
set_tests_properties(clientWithDb PROPERTIES FIXTURES_REQUIRED "server;database")
```

虽然让 `ctest` 自动将固件依赖项添加到测试执行集中通常是有用的特性，但有时也不希望这样做。继续上面的示例，开发人员可能想让服务器继续运行，并多次执行客户机测试。他们可能正在进行更改，重新编译代码，并检查每次更改是否通过客户机测试。为了支持这种级别的控制，CMake 3.9 向 `ctest` 引入了 `-FS`、`-FC` 和 `-FA` 选项，每个选项都需要一个正则表达式，该正则表达式与固件名称匹配。`-FS` 用于禁用为正则表达式

式匹配的固件添加依赖项。`-FC` 对清除测试执行相同的操作，而 `-FA` 将两者结合，禁用匹配的设置和清除测试。常见的情况是完全禁用添加任何安装/清理依赖项，这可以通过给出句点(.) 的正则表达式来完成。下面的例子演示了这些选项的效果：

命令行	执行的测试集
<code>ctest -FS server -R clientNoDb</code>	<code>clientNoDb, stopServer</code>
<code>ctest -FC server -R clientNoDb</code>	<code>clientNoDb, startServer</code>
<code>ctest -FA server -R clientNoDb</code>	<code>clientNoDb</code>
<code>ctest -FS . -R client</code>	<code>clientNoDb, clientWithDb, stopServer</code>
<code>ctest -FA . -R client</code>	<code>clientNoDb, clientWithDb</code>

## 24.6. 交叉编译和模拟器

当项目定义的可执行目标用于 `add_test()` 时，CMake会自动替换构建可执行文件的位置。对于交叉编译来说，这不起作用，因为主机不能直接运行为不同平台的二进制文件。为了实现这一点，CMake提供了一个 `CROSSCOMPILING_EMULATOR` 目标属性，可以将目标设置为启动脚本或可执行文件。如果设置了属性，CMake将把它放在目标二进制文件之前，并将其作为命令来运行(也就是说，真正的目标二进制文件将成为 `CROSSCOMPILING_EMULATOR` 提供的模拟器命令的第一个参数)。这样，即使在交叉编译时也可以运行测试。

`CROSSCOMPILING_EMULATOR` 不必是实际的模拟器，只要是可以在主机上运行目标可执行文件的命令即可。可以在目标平台的模拟器上执行用例，也可以将其设置为脚本，或将可执行文件复制到目标机器并远程运行(例如：通过SSH连接)。无论使用哪种方法，模拟器或二进制文件的启动时间可能非常重要，可能会对测试计时标准产生影响。反过来，可能需要修改超时设置。

`CROSSCOMPILING_EMULATOR` 目标属性的默认值取自 `CMAKE_CROSSCOMPILING_EMULATOR` 变量，这是获取模拟器详细信息的常用方式，而不是设置每个目标的属性。这个变量通常在工具链中设置，因为它影响 `try_run()` 的方式，与上面影响测试和定制的方式类似。

即使没有交叉编译，CMake仍然会使用非空的 `CROSSCOMPILING_EMULATOR` 目标属性，并将它前置到命令行中，用于测试和执行该目标的自定义命令。这可能非常有用，允许将属性临时设置为启动脚本，从而可以进行调试或数据收集等工作。不建议将此技术作为项目构建的永久特性。

## 24.7. 构建和测试模式

`ctest`不仅可以用来执行一组测试，它还可以驱动整个配置、构建和测试。主要有两种方法：一种最基本、独立的方法，另一种与仪表工具紧密关联的方法。基本的方法是使用 `--build-and-test` 选项调用 `ctest` 工具，其有自己预期的形式：

```
ctest --build-and-test sourceDir buildDir
--build-generator generator
[options...]
[--test-command testCommand [args...]]
```

如果没有任何选项，上面的代码将使用 `sourceDir` 和 `binaryDir` 运行CMake，并使用指定的生成器。所以这三个选项都必须指定。如果CMake运行成功，`ctest`将构建`clean`目标，最后构建默认的`all`目标。构建步骤之后运行测试，命令行上的最后一个选项必须是 `--test-command`，及其关联的 `testCommand` 和一些可选参数。这是一种调用 `ctest` 运行所有测试的方式。

```
ctest --build-and-test sourceDir buildDir  
--build-generator Ninja  
--test-command ctest -j 4
```

以上命令执行了完整的配置-清理-建造-测试流水。提供了各种选项，可用于修改运行管道的相应部分，以及如何运行。例如，`--build-nocmake` 和 `--build-noclean` 分别禁用 `configure` 和 `clean` 步骤。`--build-two-config` 选项将调用 `CMake` 两次，用来处理特殊情况，即需要通过第二次 `CMake` 来完全配置项目。使用像 `Visual Studio` 这样的生成器时，需要使用 `--build-generator-platform` 和 `--build-generator-toolset` 指定额外的生成器信息，它们将分别作为 `-A` 和 `-T` 传递给 `cmake`，并用于配置步骤。像 `Xcode` 这样的生成器可能需要给指定项目名称，以便找到配置阶段生成的项目文件，这可以通过 `--build-project` 来完成。可以使用 `--build-target` 设置构建步骤中要构建的目标，并且可以通过使用工具传递 `--build-makeprogram` 覆盖构建工具。

可以看到，与 `--build-and-test` 模式相关的选项都以 `--build` 开始。虽然大多数选项都有直观的名称，但是 `--build` 前缀可能会一些异常情况。存在一个名为 `--build-options` 的选项，最初看起来可能与构建步骤相关，但实际上用于将命令行选项传递给 `cmake` 命令。还有个附加的约束，即必须在命令行的末尾（除非同时指定了 `--test-command`，这种情况下 `--build-options` 必须在 `--test-command` 之前）。下面的示例将展示这些，向 `cmake` 调用添加了两个缓存变量，并在构建步骤后运行完整的测试套件。

```
ctest --build-and-test sourceDir buildDir  
--build-generator Ninja  
--build-options -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON  
--test-command ctest -j 4
```

还有一些其他的 `--build-` 选项，不过上面的内容已经涵盖了最常用的选项。还有一个需要说明的选项是 `--test-timeout`，它对 `test` 命令运行的时间设置了时间限制（以秒为单位）。

使用 `ctest` 控制整个流水，与显式调用每个阶段所需的工具相比，优劣要取决于特定的情况。上面的最后一个例子可以在 `Unix` 上使用以下等价的命令实现：

```
mkdir -p buildDir  
cd buildDir  
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON sourceDir  
cmake --build . --target clean  
cmake --build .  
ctest -j 4
```

单独调用工具允许以完整的选项集运行，而 `--build-and-test` 对于控制构建阶段的能力非常有限。

构建和测试模式特别方便，项目需要执行完整的配置-构建-测试，从而与主构建分离。整个循环可以通过 `ctest` 来控制，因此可以用作对 `add_test()` 调用的 `COMMAND`，添加测试到项目的测试套件中会变得简单。

下面的例子展示了，如何使用单独的构建来测试项目构建库提供的 API：

```
add_library(decoder foo.c bar.c)  
  
add_test(NAME decoder.api  
COMMAND ${CMAKE_CTEST_COMMAND}  
--build-and-test ${CMAKE_CURRENT_LIST_DIR}/test_api  
${CMAKE_CURRENT_BINARY_DIR}/test_api  
--build-generator ${CMAKE_GENERATOR}  
--build-options -DDECODER_LIB=${TARGET_FILE:decoder}>  
--test-command ${CMAKE_CTEST_COMMAND}  
)
```

`test_api` 源目录将包含自己的`CMakeLists.txt`文件，目的是配置针对解码库链接的构建，并在 `DECODER_LIB` 中设置绝对路径(是将库位置传递给测试项目的方法之一)。关于这种类型的测试，还可以用来验证特定的测试项目有没有构建，或者验证因特定的错误导致的配置失败(例如丢失符号)。这种预期的构建错误不能在主项目中测试，因为会导致主项目构建失败。

此类测试可能有用的场景是测试主项目创建的代码生成器输出。测试固件可用于设置一对测试，一个用于生成代码，另一个用于执行测试构建。例如：

```
add_executable(codegen generator.cpp)

add_test(NAME generate_code COMMAND codegen)
add_test(NAME build_generated_code
COMMAND ${CMAKE_CTEST_COMMAND}
--build-and-test ${CMAKE_CURRENT_LIST_DIR}/test_generation
${CMAKE_CURRENT_BINARY_DIR}/test_generation
--build-generator ${CMAKE_GENERATOR}
--test-command ${CMAKE_CTEST_COMMAND}
)

set_tests_properties(generate_code PROPERTIES FIXTURES_SETUP generator)
set_tests_properties(build_generated_code PROPERTIES FIXTURES_REQUIRED generator)
```

构建和测试模式也可以用于验证`CMake`脚本，方法是将它们包含在一个小型的测试项目中，并在适当的情况下调用其功能。实际上，这提供了相当方便的方法来实现`CMake`脚本的单元测试，从而避免了将此类测试放到主项目的配置阶段。

虽然构建和测试模式对于上面提到的情况肯定有用，但缺乏完全脚本化运行的灵活性，这种情况下，每个命令都可以使用完整的选项集。下一节将介绍调用`ctest`的另一种方法，提供了对整个流水的更强大处理，包括一些报告功能。

## 24.8. 集成CDash

`CTest`与另一款名为`CDash`的产品有着悠久的历史和密切的合作关系，`CDash`也是开发`CMake`和`CTest`的同一家公司开发的。`CDash`是基于`Web`的仪表板，它从`ctest`驱动的软件构建和测试管道收集结果。从管道的每个阶段收集警告和错误，并显示每个阶段的概要信息，可以看到每个单独的警告或错误。通过对过去流水的历史记录，可以观察一段时间内的趋势并比较运行情况。`CMake`本身有自己的仪表盘，跟踪夜间构建，与合并请求相关的构建等等。花几分钟研究示例仪表盘将有助于理解本节所涵盖的内容：

<https://open.cdash.org/index.php?project=CMake>

### 24.8.1. CDash的重要概念

`CTest`和`CDash`如何执行流水和报告结果联系在一起，这里有三个重要的概念：步骤(也称为动作)、模型(也称为模式)和跟踪。步骤是流水执行的一系列操作。按照通常的调用顺序，定义主要的动作集是：

- Start
- Update
- Configure
- Build
- Test
- Coverage
- MemCheck
- Submit

并不是所有的操作都必须执行，有些操作可能不受支持或不需要运行。简单地说，CDash仪表盘中的每一行对应一个管道，通常会显示所执行操作的信息(提交哈希、警告、错误、失败等)。

每个流水都必须与一个模型相关联，该模型用于定义某些行为，例如在特定步骤失败后是否继续后续步骤。当没有请求特定操作时，该模型还提供一组默认操作。支持的模型有：

#### 夜间性

每天调用一次，通常在执行机器不太忙的时候由自动作业。默认的操作集包括上面列出的所有步骤，**MemCheck**除外。如果**Update**步骤失败，将执行其余步骤。

#### 持续性

与夜间性构建非常相似，不同之处在于在一天中根据需要运行多次，通常是为了响应提交的更改。它定义了与夜间性构建相同的一组操作，但是如果**Update**步骤失败，则不会执行后面的步骤。

#### 实验性

该模型用于由开发人员根据需要执行特殊的实验。默认操作集包括**Update**和**MemCheck**之外的所有步骤。如果指定了三种已定义模型之外的模型类型，或者根本没有指定任何模型类型，则该模型类型将视为实验性的。

跟踪控制在仪表盘结果中会显示相应流水的结果，跟踪名称可以是项目或开发人员希望使用的任何名称，但如果没有指定跟踪，将设置为与模型相同的名称。有一种常见的误解，即模型在仪表盘中控制分组，但这部分是由跟踪完成的。覆盖率和**MemCheck**操作是一种特殊的情况，有效地忽略了跟踪，并且在仪表盘上专用组的部分进行显示(分别是覆盖率和动态分析)。

### 24.8.2. 执行管道和操作

对于有配置文件的项目，可以使用以下ctest命令形式使用整个或单个流水步骤：

```
ctest [-M Model] [-T Action] [--track Track] [otherOptions...]
```

必须指定至少一个或两个Model和Action。为了方便，**-M** 和 **-T** 可以组合成 **-D**，如下所示：

```
ctest -D Model[Action] [--track Track] [otherOptions...]
```

**-D** 可以忽略该操作或将其附加到Model中。有效参数例子包括**NightlyConfigure**, **NightlyConfigure**, **ExperimentalBuild**等。如果需要，可以多次指定 **-T** 和 **-D**，以便配置多个步骤。注意 **-D** 也可以用于定义ctest变量，ctest命令将处理不能识别的Model或ModelAction作为测试设置。因此，使用 **-M** 和 **-T** 可能比使用 **-D** 更安全。

夜间执行使用默认步骤，并在默认组中报告结果。夜间执行可以简单写为：

```
ctest -M Nightly
```

同样的事情，但是在另一个叫做“夜间管理员”的小组中，结果是这样的：

```
ctest -M Nightly --track "Nightly Master"
```

考虑由配置、构建和测试步骤组成的自定义流水，将结果分组在简单测试下。这需要显式地指定步骤集，不同于为实验模型定义的默认操作集(没有执行覆盖步骤)。这可以使用ctest的序列来完成，也可以使用命令行上多个 **-T** 来完成。两种形式如下所示：

```
# Separate commands
ctest -T Start -M Experimental --track "Simple Tests"
ctest -T Configure
ctest -T Build
ctest -T Test
ctest -T Submit

# One command
ctest -M Experimental --track "Simple Tests" \
-T Start -T Configure -T Build -T Test -T Submit
```

第一步应该是**Start**操作，用于初始化流水细节，并记录后续步骤将使用的模型和跟踪名称。如果将操作拆分为单独的**ctest**调用，对于后面的步骤，不需要重复这些细节。最后是提交操作，这里假设目标是向仪表盘提交最终结果。

上面的所有输出都收集在调用**ctest**目录下的测试子目录中。**Start**操作输出一个名为**TAG**的文件，该文件至少包含两行，第一行是运行开始的日期-时间，格式为`YYYYMMDD-hhmm`，第二行是跟踪名称。**CMake 3.12**将模型名称作为第三行。**Start**后的每一步行动，它将创建自己的输出文件在 `Testing/YYYYMMDD-hhmm-`  
`hhmm/<Action>.xml` 中，并将测试日志记录在 `Testing/Temporary/Last<Action>_YYYYMMDD-hhmm.log` 文件中 (在  
**MemCheck**步骤的情况下，`<action>` 部分将是动态分析，而不是这些文件名中的**MemCheck**)。**Submit**操作收集**XML**输出文件和日志文件，并提交到指定的指示板上。

要将构建注释附加到整个流水中，请在提交步骤中使用 `-A` 或 `--add-notes` 来指定要上传的文件名，如果要添加多个文件，可以用分号分隔。这是记录有关特定流水信息的方法，例如来自持续集成系统的信息。

```
ctest -T Submit --add-note JobNote.txt
```

还有一个 `--extra-submit` 选项，主要在**ctest**内部使用，开发人员或项目不应该直接使用。

虽然上述功能主要用于与**CDash**集成，也可以用于其他场景。例如，**Jenkins CI**系统有一个插件，允许读取测试操作的**Test.xml**输出文件，以类似于**CDash**的方式记录测试结果。与普通方式运行**ctest**不同，可以仅使用**Test**操作的仪表盘。**Jenkins**插件只需要知道在哪里可以找到**Test.xml**文件，就可以读取测试结果。使用这种方式时，启动操作都可以省略，如果其他步骤没有在启动操作前执行，**ctest**将使用**Experimental**模式，静默地执行启动操作。项目可能希望在这样做之前清除测试目录中的内容，以确保**Jenkins**只获取当前运行的结果。

将**XML**输出文件传递给**CDash**以外的工具时，可能不需要**ctest**压缩输出。默认情况下，操作的输出压缩成**ascii**编码的形式写入**XML**文件，可以通过 `--no-compress-output` 选项传递给**ctest**来避免压缩。只有在必要时才使用此选项，因为这会产生更大的输出文件。

仪表盘步骤在没有**CDash**的情况下，可以支持对代码覆盖或内存的检查(**Valgrind**, **Purify**, 各种杀毒软件等等)。这些指示盘操作可以使相关工具的使用和结果收集变得更容易。

### 24.8.3. 配置CTest

为集成**CDash**准备一个项目，主要是由**CMake**提供的**CTest**模块完成。这个模块应该包含在 `project()` 之后的顶层**CMakeLists.txt**文件中。

```

cmake_minimum_required(VERSION 3.0)
project(CDashExample)

# ... set any variables to customize CTest behavior

include(CTest)

# ... Define targets and tests as usual

```

因为**CTest**模块会在构建目录中写入各种文件，需要包含在顶层**CMakeLists.txt**文件中，而那些生成文件通常位于构建树的顶层。如果该项目通过**add\_subdirectory()**合并到父项目中，父项目还应该将**include(CTest)**放在其顶层**CMakeLists.txt**中，以便在正确位置生成必要的文件。

**CTest**模块定义了**BUILD\_TESTING**缓存变量，默认值为**true**。用于决定模块是否调用**enable\_testing()**，因此项目不必调用**enable\_testing()**。只有在启用测试时，项目可以使用此缓存变量来处理。如果项目有很多测试，这些测试需要很长时间来构建，这个变量可以避免测试在不需要时添加到构建中。

```

cmake_minimum_required(VERSION 3.0)
project(CDashExample)
include(CTest)

# ... define regular targets

if(BUILD_TESTING)
    # ... define test targets and add tests
endif()

```

**CTest**模块为每个**Model**和**ModelAction**组合定义构建目标。这些目标在执行**ctest**时，可用**-D**设置目标名称，目的是为在**IDE**中执行整个流水或仅执行仪表盘操作。如果使用命令行方式，直接用**ctest**就好。

**CTest**模块执行时，非常重要的任务会写在构建目录中名为**DartConfiguration.tcl**的配置文件中。该文件的名称很有历史性，**Dart**是**CDash**最开始的名称。该文件记录很多细节信息，如源目录和构建目录位置、执行构建设备的信息、使用的工具链、各种工具的位置和其他默认值。还包含**CDash**服务器的详细信息，为了做到这一点，项目需要在源文件树顶层提供一个**CTestConfig.cmake**文件，并且带有相关内容。合适的**CTestConfig.cmake**文件可以从**CDash**中获得(需要管理员权限)，手动创建这个文件也并不困难。看起来像这样：

```

# Name used by CDash to refer to the project
set(CTEST_PROJECT_NAME "MyProject")

# Time to use for the start of each day. Used by
# CDash to group results by day, usually set to
# midnight in the local timezone of the CDash server.
set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")

# Details of the CDash server to submit to
set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=${CTEST_PROJECT_NAME}")
set(CTEST_DROP_SITE_CDASH YES)

# Optional, but recommended so that command lines
# can be seen in the CDash logs
set(CTEST_USE_LAUNCHERS YES)

```

`DartConfiguration.tcl`文件由`CTest`模块输出，包含对每个仪表盘的可配置选项。默认情况下，大多数已经设置为适当的值，但是`Coverage`和`MemCheck`步骤中，有些项开发人员可能特别感兴趣。这可通过`CMake`变量控制，开发者可以在`CMake`缓存中检查和修改这些变量，也可以在包含`CTest`模块之前在`CMakeLists.txt`文件中直接设置这些变量。

假设测试覆盖率步骤调用了`gcov`，`CTest`模块将通过名字索引命令。`COVERAGE_COMMAND`缓存变量保存了该搜索的结果，开发人员可以修改。第二个缓存变量`COVERAGE_EXTRA_FLAGS`用于`COVERAGE_COMMAND`的参数，因此开发人员能够控制命令和传递的参数。

`MemCheck`步骤更有趣。支持许多不同的内存检查工具，包括`Valgrind`, `Purify`, `BoundsChecker`等等。对于前三个，可以通过`MEMORYCHECK_COMMAND`设置可执行文件的位置来选择，`ctest`将从可执行文件名称中进行识别。对于`Valgrind`，还可以设置`VALGRIND_COMMAND_OPTIONS`变量来覆盖`Valgrind`原始的选项。要使用其中的工具，请将`MEMORYCHECK_TYPE`设置为以下字符串之一(会忽略`MEMORYCHECK_COMMAND`)：

- `AddressSanitizer`
- `LeakSanitizer`
- `MemorySanitizer`
- `ThreadSanitizer`
- `UndefinedBehaviorSanitizer`

然后`ctest`会启动测试可执行文件，但相关的环境变量会设置为启用的软件。注意，软件工具需要使用相关的编译器和链接器标志(通常是`-fsanitize=XXX`，可能还有`-fno-omit-frame-pointer`)对目标项目进行构建。相关标志和工具的详细信息，请参阅`Clang`或`GCC`文档。

以上细节足够执行各种仪表盘操作，并将结果提交到`CDash`服务器，但是存在一个先有鸡还是先有蛋的问题。更新和配置步骤需要执行，以获`DartConfiguration.tcl`文件。因此，无法捕获这两步的详细信息，对于配置步骤，第一次运行`cmake`的输出将丢失，只能通过在已经配置的构建目录中重新运行`cmake`获得输出。不过，其他步骤的输出是可以获取的，某些情况下这可能就足够了。例如，使用像`Gitlab CI`或`Jenkins`这样的持续集成系统时，源码的克隆或更新可以由CI系统来处理。可以执行初始的`cmake`运行，然后其他步骤可以作为仪表盘操作运行。最终结果可以提交给`CDash`服务器，也可以由CI系统直接读取，或者两者都可以。

为了捕获完整的输出，包括现有源码的克隆或更新，以及第一个配置步骤，必须编写自定义`ctest`脚本来建立必需设置的信息，并调用相关的`ctest`函数。这可能是非常复杂的过程，如果已经在使用另一个CI系统，就不需要这样做。如果不克隆/更新步骤，定制脚本的复杂性就会降低。使用这种方式时，`ctest`使用`-S`或`-SP`选项调用(它们是相同的，只是后者创建新进程，而前者不会)。下面演示了一个相当简单的示例。

```
ctest -S MyCustomCTestJob.cmake
```

*MyCustomCTestJob.cmake*

```

# Re-use CDash server details we already have
include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

# Basic information every run should set, values here are just examples
site_name(CTEST_SITE)
set(CTEST_BUILD_NAME ${CMAKE_HOST_SYSTEM_NAME})
set(CTEST_SOURCE_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)

# Dashboard actions to execute, always clearing the build directory first
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental)
ctest_configure()
ctest_build()
ctest_test()
ctest_submit()

```

虽然上面的脚本相当简单，但下面的示例更有趣，说明了自定义脚本是如何定义行的。不是等到运行的最后才将结果提交到仪表盘，而是在每个步骤之后逐步提交(如果某些步骤花费了很长时间，这很有用)。这些可执行文件是工具构建的，通过运行工具检查，而不是常规测试。最后再上传了一些额外的文件。

```

include(${CTEST_SCRIPT_DIRECTORY}/CTestConfig.cmake)

site_name(CTEST_SITE)
set(CTEST_BUILD_NAME "${CMAKE_HOST_SYSTEM_NAME}-ASan")
set(CTEST_SOURCE_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}")
set(CTEST_BINARY_DIRECTORY "${CTEST_SCRIPT_DIRECTORY}/build")
set(CTEST_CMAKE_GENERATOR Ninja)
set(CTEST_CONFIGURATION_TYPE RelWithDebInfo)
set(CTEST_MEMORYCHECK_TYPE AddressSanitizer)
set(configureOpts
    "-DCMAKE_CXX_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
    "-DCMAKE_EXE_LINKER_FLAGS_INIT=-fsanitize=address -fno-omit-frame-pointer"
)
ctest_empty_binary_directory(${CTEST_BINARY_DIRECTORY})
ctest_start(Experimental TRACK Sanitizers)
ctest_configure(OPTIONS "${configureOpts}")
ctest_submit(PARTS Start Configure)
ctest_build()
ctest_submit(PARTS Build)
ctest_memcheck()
ctest_submit(PARTS MemCheck)
ctest_upload(FILES ${CTEST_BINARY_DIRECTORY}/mytest.log
    ${CTEST_BINARY_DIRECTORY}/anotherFile.txt
)
ctest_submit(PARTS Upload Submit)

```

**CMake**文档中详细介绍了各种 `ctest_...` 命令，以及**CTest**和**CMake**变量，这些变量可用于定制每个步骤或方法。上面的脚本是基础脚本，可以用来试验不同的参数和变量。

同时处理克隆/更新项目的脚本会比较复杂。项目通常有特殊的方法来实现这一点，通常需要决定如何安排夜间和持续构建之类的事情。对合并请求的自动化构建支持，将在很大程度上依赖于承载项目存储库的功能。对于那些感兴趣的人，推荐的入门方法是找到使用类似存储库托管的项目，将其作为指导项目。一些项目在其存储库中包含定制脚本，以便于访问(许多**Kitware**的项目都是这样做的，并且这些脚本已经有相当好的文档记录)。

#### 24.8.4. 测试标准和结果

上面的例子简单地展示了文件上传可以合并到自定义的CTest脚本中。`ctest_upload()` 提供了用于记录上传文件的机制，并将其附加到CDash的构建结果中，随后的 `ctest_submit()` 中执行上传操作。但有时文件上传应该与特定的测试相关联，而不是与整个脚本相关联。为此，CMake提供了 `ATTACHED_FILES` 和 `ATTACHED_FILES_ON_FAIL` 测试属性。两者都包含一个要上传的文件列表，并与特定的测试相关联，唯一的区别是后者包含的文件只在测试失败时才上传。这是一种非常有用的方法，可以记录有关故障的冗余信息，以便进行进一步的调试。

```
add_executable(doGen ...)  
add_test(NAME generateFile COMMAND doGen)  
set_tests_properties(generateFile PROPERTIES  
    ATTACHED_FILES_ON_FAIL  
    ${CMAKE_CURRENT_BINARY_DIR}/generated.c  
    ${CMAKE_CURRENT_BINARY_DIR}/generated.h  
)
```

测试还可以记录单个测量值，CDash中的每次测试提交都会进行记录和跟踪。形式通常是`key=value`，虽然`=value`部分可以省略，使用默认值1。测量会记录成测试特性，如下所示：

```
set_tests_properties(perfRun PROPERTIES  
    MEASUREMENT mySpeed=${someValue}  
)
```

因为测量值必须在测试运行前定义，所以这样做的用处有限。更有用的是Vtk等项目大量使用的未文档化特性，以及围绕它构建的项目。这些项目中，测量值以类似于HTML标记的形式嵌入到测试输出本身中。`ctest`扫描这些度量值的输出，提取相关数据并将其作为测试结果的一部分上传到CDash。这些测量结果随后显示在 `test details` 页面顶部的结果表中。最简单的测量类型定义如下：

```
<DartMeasurement name="key" type="someType">value</DartMeasurement>
```

`name`作为结果表中测量值的标签，`type`类似于文本/字符串或数值/双引号的属性。值是对测量有意义的文本或数字内容。对于数值，CDash提供了工具来绘制最近测试运行中每次测量的历史，这对于随时间变化的行为非常有用。

另一种形式可以嵌入一个文件，而不是特定值：

```
<DartMeasurementFile name="key" type="someType">filePath</DartMeasurementFile>
```

第二种形式对于上传图片最有用，类型属性应该是`image/png`或`image/jpeg`之类的。`filePath`是要上传文件的绝对路径。

CDash可以识别一些特殊的图像测量名称，可用于帮助比较预期和实际的图像，CDash甚至为重叠比较提供了交互式UI。识别的名称属性及含义包括：

#### *TestImage*

为测试生成的图像。它可以认为是测试输出，并将单独显示，也将作为交互式比较图像的一部分。

#### *ValidImage*

相当于测试的期望图像。应该与测量值具有相同的尺寸，但不一定要求具有相同的图像格式。只包括交互式图像。

#### *DifferenceImage2*

可以使用各种工具来生成两个图像之间的差异。当测试提供这样的图像文件时，可以使用这个名称将其包含在上载到CDash的测试输出中。比较图像将纳入交互式比较图像中。

## 24.9. GoogleTest

CMake/Ctest为构建、执行和确定测试通过/失败状态提供支持。该项目负责提供测试代码本身，就可以使用类似GoogleTest这样的测试框架。这样的框架补充了CMake和Ctest所提供的特性，便于编写清晰、结构良好的测试用例，从而很好地集成到CMake和Ctest的工作方式中。

CMake通过FindGTest模块支持GoogleTest。该模块搜索预先构建的GoogleTest的位置，并创建项目可以使用的变量将GoogleTest合并到他们的构建中。CMake 3.5中，还提供了导入目标，这比使用变量更可取。使用导入目标可以更健壮地处理使用需求和属性。下面是使用CMake 3.5或更高版本的示例：

```
add_executable(myGTestCases ...)

find_package(GTest REQUIRED)
target_link_libraries(myGTestCases PRIVATE GTest::GTest)

add_test(NAME myGTestCases COMMAND myGTestCases)
```

导入目标负责确保在构建myGTestCases时使用相关的头文件搜索路径，以及在需要时链接线程库等内容。上面的代码可以在所有平台上运行，隐藏了与不同平台和编译器上的不同名称、运行时、标志等相关的复杂性。如果使用模块定义的变量，不是导入目标，这些事情大多必须手动处理，那么这就是一项相当危险的任务了。

更健壮的方法是将GoogleTest的源代码直接合并到构建中，而不是依赖于已有的预构建二进制文件。这确保GoogleTest与项目的其余部分使用完全相同的编译器和链接器构建，从而避免了在使用预构建的GoogleTest二进制文件时可能出现的许多问题。项目可以通过多种方式做到这一点，每种方式都有其优缺点。在项目中创建源文件和头文件的副本是最简单的，但它会使项目与将来可能对GoogleTest进行的改进失去联系。

GoogleTest git存储库可以作为git子模块添加到项目中，但这也有其自身的健壮性问题。作为配置步骤的一部分，下载GoogleTest源代码的第三个选项在27.2节中详细讨论“FetchContent”，它也有一些缺点(CMake 3.11中添加的特性也使它用起来变得非常容易)。

使用GoogleTest的测试可执行文件通常定义多个测试用例。通常只运行一次可执行文件，并假设是一个单独的测试用例，这种模式并不合适。理想情况下，每个GoogleTest测试用例都应该对Ctest可见，这样每个测试用例都可以单独运行和评估。FindGTest模块提供了`gtest_add_test()`函数，该函数扫描源代码，查找相关GoogleTest宏的使用情况，并提取出单独的测试用例作为自己的Ctest测试。这个命令的示例：

```
gtest_add_tests(executable "extraArgs" sourceFiles..)
```

CMake 3.1中，扫描的`sourceFiles`列表可以用关键字`AUTO`代替，通过假设可执行文件是CMake目标，并使用它的`SOURCES`目标属性来获得源文件列表。

CMake 3.9中，项目可以使用`gtest_add_tests()`函数和项目本身构建的GoogleTest。这意味着该项目不需要Find模块，因此该功能移到新的GoogleTest模块，使用FindGTest将其包含进来，以保持向后兼容性。作为完成这项工作的一部分，还改进了使用形式：

```

gtest_add_tests(
    TARGET target
    [SOURCES src1...]
    [EXTRA_ARGS arg1...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [SKIP_DEPENDENCY]
    [TEST_LIST outVar]
)

```

仍然支持之前的形式，但项目应该在可能的情况下使用新形式，因为它更灵活、更健壮。例如，可以对带有不同参数的 `gtest_add_tests()` 的多次调用提供相同的目标，每个调用不同的 `TEST_PREFIX` 和/或 `TEST_SUFFIX`，以区分生成的测试集。新形式还为 `TEST_LIST` 选项添加了一组测试。有了可用的测试名称，项目就能够根据需要修改测试的属性。下面的示例演示了这些功能：

```

# Assume GoogleTest is already part of the build, so we don't need
# FindGTest and can reference the gtest target directly
include(GoogleTest)
add_executable(testDriver ...)
target_link_libraries(testDriver PRIVATE gtest)

# Run the testDriver twice with two different arguments
gtest_add_tests(
    TARGET testDriver
    EXTRA_ARGS --algo=fast
    TEST_SUFFIX .Fast
    TEST_LIST fastTests
)
gtest_add_tests(
    TARGET testDriver
    EXTRA_ARGS --algo=accurate
    TEST_SUFFIX .Accurate
    TEST_LIST accurateTests
)
set_tests_properties(${fastTests} PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)

```

上面的示例创建了两组测试，对它们设置了不同的超时限制。每组测试的名称都有不同的后缀。如果没有 `TEST_SUFFIX` 选项，对 `gtest_add_tests()` 的第二次调用将失败，因为这将尝试创建与第一次调用同名的测试。该示例还为某些测试设置了 `Beta` 标签，而不管它们属于哪个测试集。

虽然 `gtest_add_tests()` 对于没有特殊格式的简单用例和源文件处理得很好，但不能处理参数化测试或通过自定义宏定义的测试。还需要重新运行 `CMake`，在测试源更改时重新扫描源文件。如果 `CMake` 的步骤不够快，编写测试代码可能会令人沮丧，因为在每次更改之后 `CMake` 将为新的测试代码重新运行。`SKIP_DEPENDENCY` 选项可以防止这种行为，并依赖于开发人员手动重新运行 `CMake` 来更新测试集，这更多针对的是测试时的临时解决方案。

`CMake 3.10` 中，添加了新函数来解决 `gtest_add_tests()` 的缺点，要求可执行文件在运行 `Ctest` 时列出测试，而不是在 `CMake` 时扫描源代码时。因此，每当测试源发生更改时，不需要重新运行 `CMake`，参数化测试将得到处理，并且对测试的格式或定义没有任何限制。唯一的权衡是，测试列表在 `CMake` 运行期间不可用，只有在实际运行 `Ctest` 时才能获得。

```

gtest_discover_tests(target
  [EXTRA_ARGS arg1...]
  [WORKING_DIRECTORY dir]
  [TEST_PREFIX prefix]
  [TEST_SUFFIX suffix]
  [NO_PRETTY_TYPES]
  [NO_PRETTY_VALUES]
  [PROPERTIES name1 value1...]
  [TEST_LIST var]
  [DISCOVERY_TIMEOUT seconds] # See notes below
)

```

默认情况下，生成参数化测试的名称时，函数将尝试使用类型或值名称，而不是数字索引。这通常会产生可读性更强、更有用的名称，但对于不希望这样做的情况，可以使用 `NO_PRETTY_TYPES` 和 `NO_PRETTY_VALUES` 来禁止替换，只使用索引值。

`DISCOVERY_TIMEOUT` 指的是运行可执行文件，以获得测试列表所花费的时间。默认的5秒对于可执行程序来说已经足够了，但是对于需要大量测试时间的可执行程序，或者其他一些导致返回测试列表花费很长时间的行为来说就不够了。这个特殊的选项最初是在CMake 3.10.1中添加的，其关键字名称为 `TIMEOUT`，但发现会与 `TIMEOUT` 测试属性冲突，从而导致意外行为。在CMake 3.10.3中，关键字被更改为 `DISCOVERY_TIMEOUT`，以防止出现意外。

由于测试列表没有返回给调用者，因此不可调用 `set_tests_properties()` 或 `set_property()` 来修改测试属性。相反，`gtest_discover_tests()` 允许将属性值指定为调用的一部分，然后将其写入Ctest输入文件，以便在运行Ctest时应用。虽然不能在CMake中遍历已发现的测试集，并单独处理，但将已发现测试的属性作为整体设置的能力是必须的。这方面的例外是，不能设置具有与 `gtest_discover_tests()` 中的关键字相同名称的测试属性，也不能设置属性列表的值。可以使用自定义的Ctest脚本来处理这种情况，下面给出了一个示例。

`TEST_LIST` 选项对于 `gtest_discover_tests()` 和 `gtest_add_tests()` 是不同的。本例中，此选项给出的变量名用于CMake输出(作为Ctest的输入)文件中，而不是直接用于CMake。`TEST_LIST` 选项只在项目向生成的Ctest输入文件中添加自定义逻辑，并引用生成的测试列表时才需要。即使这样，也只有在对 `gtest_discover_tests()` 的多次调用中使用相同目标时，才有必要这样做。如果没有对 `TEST_LIST` 选项进行设置，则使用 `<target>_TESTS` 的默认变量名。

可以通过将文件名附加到 `TEST_INCLUDE_FILES` 目录属性中保存的文件列表来添加定制代码。因为 `gtest_discover_tests()` 使用该属性来构建要由Ctest读取的文件集，所以项目不能覆盖这个目录属性，只能进行追加。下面的示例展示了如何使用自定义文件操作测试的属性，并实现与前面的 `gtest_add_tests()` 示例相同的逻辑，包括解决超时名称冲突的解决方案：

```

gtest_discover_tests(
  testDriver
  EXTRA_ARGS --algo=fast
  TEST_SUFFIX .Fast
  TEST_LIST fastTests
)
gtest_discover_tests(
  testDriver
  EXTRA_ARGS --algo=accurate
  TEST_SUFFIX .Accurate
  TEST_LIST accurateTests
)
set_property(DIRECTORY APPEND PROPERTY
  TEST_INCLUDE_FILES ${CMAKE_CURRENT_LIST_DIR}/customTestManip.cmake
)

```

*customTestManip.cmake*

```

# Set here to work around the TIMEOUT keyword clash with the
# gtest_discover_tests() call, works with all CMake versions
set_tests_properties(${fastTests} PROPERTIES TIMEOUT 3)
set_tests_properties(${accurateTests} PROPERTIES TIMEOUT 20)

set(betaTests ${fastTests} ${accurateTests})
list(FILTER betaTests INCLUDE REGEX Beta)
set_tests_properties(${betaTests} PROPERTIES LABELS Beta)

```

使用自定义的Ctest脚本给项目增加了复杂性，允许完全控制测试属性。不需要担心与 `gtest_discover_tests()` 的名称冲突，并且可以安全地处理列表值的属性。

## 24.10. 总结

每个测试的名称应该简短，足够描述于测试的性质，以便使用正则表达式和Ctest的 `-R` 和 `-E` 缩小测试集的范围。避免在名称中包含 `test`，因为只用于向测试输出添加额外内容，除此之外没有任何用处。

假设项目可能合并到更大的层次结构中，其中可能有许多其他测试。所有项目中保持测试名的唯一非常困难，但是与其在每个测试名中包含一个特定于项目的字符串，不如考虑使用 `LABELS` 测试属性来为每个测试包含特定于项目的标签。这些标签允许通过正则表达式 `-L` 和 `-LE` 包含或排除测试。测试可以有多个标签，因此这对如何使用其他标签没有限制，但可能使测试使用更方便。

标签的另一个好用法是识别预期要运行很长时间的测试。开发人员和持续集成系统可能不希望频繁地运行它们，因此能够基于测试标签非常方便的排除。考虑向运行时间较长且不需要经常运行的测试添加标签，没有任何其他现有惯例的情况下，使用“长期”(LongRunning)这个标签是一个不错的选择。

除了使用正则表达式匹配测试名称和标签外，还可以将测试集缩小到特定目录。不用从构建树的顶层运行Ctest，可以从下面的子目录运行。Ctest只知道从该目录的源目录及其下的目录中定义的那些测试。为了充分利用这一点，不应该将所有测试都收集在一个地方，并且不使用目录结构进行定义。让测试靠近其源码可能会很有用，这样可以重用源代码的自然目录结构，也为测试提供结构。如果要移动源代码，这种方法还可以更容易地移动与其关联的测试。

编写测试很容易，打开日志记录，然后使用pass/fail的正则表达式来确定是否成功。这是一种相当粗糙的方法，因为开发人员经常更改日志输出，这里假设它只是为了提供信息。向日志输出中添加时间戳将使这种方法更加复杂。如果可能的话，最好让测试代码本身通过显式地测试预期的前置和后置条件、中间值等方式来确定成功或失败状态，而不是依赖于匹配日志的输出。像GoogleTest这样的测试框架使得编写和维护这样的测试变得相当容易，并且强烈推荐使用这些框架(使用哪个框架不重要，重要的是使用适合自己的)。

如果使用GoogleTest框架，可以考虑使用GoogleTest模块提供的 `gtest_add_tests()` 和 `gtest_discover_tests()` 函数。如果测试代码足够简单，`gtest_add_tests()` 可以找到所有测试，提供了操作单个测试属性的方法，但是在处理测试代码本身时可能不太方便，因为可能需要频繁地重新运行CMake。如果项目需要CMake 3.10.3或更高版本作为最低版本，那么 `gtest_discover_tests()` 可能更合适。如果遵循上面关于使用测试标签的建议，此函数的主要缺点是将测试属性设置为列表，所以需要做很多额外的工作，这一点尤其重要。如果需要支持3.9之前的CMake版本，则只能使用 `gtest_add_tests()`，并且只能使用更简单形式。该项目还将需要使用FindGTest模块，而不是GoogleTest模块，如果GoogleTest是作为项目本身的一部分构建的，将进一步增加复杂性。因此强烈建议使用GoogleTest的项目迁移到CMake 3.9或更高版本，最好是3.10.3或更高版本。

对于不同目标平台进行交叉编译的项目，考虑是否可以将测试编写为在模拟器下运行，或者通过脚本或等效机制在远程系统上执行。CMake的 `CMAKE_CROSSCOMPILING_EMULATOR` 变量和相关的 `CROSSCOMPILING_EMULATOR` 目标属性可以用于实现这两种策略。理想情况下，`CMAKE_CROSSCOMPILING_EMULATOR` 应该在交叉编译工具链文件中设置

充分利用**Ctest**中对并行测试执行的支持。如果已知测试使用多个CPU，请设置这些测试的处理器属性，以便为**Ctest**提供调度指导。如果测试需要独占访问共享资源，请使用 `RESOURCE_LOCK` 属性来控制对该资源的访问，避免使用 `RUN_SERIAL` 测试属性，除非没有其他选择。`RUN_SERIAL` 可能会对并行测试性能产生很大的负面影响，除了快速的、临时的开发人员实验之外，不推荐使用 `RUN_SERIAL`。如果运行**Ctest**的机器上可能有其他进程造成CPU负载，可以考虑使用 `-1` 来限制CPU上的过度提交。这在开发人员的机器上特别有用，因为开发人员可能同时为多个项目构建和运行测试。

如果最低CMake版本可以设置为3.7或更高，最好使用测试固件来定义测试之间的依赖关系。定义测试用例，以设置和清理其他测试所需的资源，启动和停止服务等等。减少运行时测试集的正则表达式匹配或使用选项 `-rerun-failed`，**Ctest**会自动添加所需的固件测试集。固件也确保跳过测试依赖关系的失败，与 `DEPENDS` 测试属性不同，固件只控制测试顺序。要获得对将自动添加到测试集中，且满足固件依赖关系的测试细粒度控制，可以使用CMake 3.9或更高版本，使用**Ctest**选项 `-FS`、`-FC` 和 `-FA`，项目只需要CMake 3.7作为最低版本。另外，由于依赖关系和时间控制很清楚，可以将 `TIMEOUT_AFTER_MATCH` 测试属性和固件一起使用。

**Ctest**构建和测试模式可以将小型测试构建合并到主项目的测试套件中，作为测试用例。当测试构建需要验证某些情况会导致配置或构建错误时，这些方法非常有效。因为测试用例可以预期为失败，所以可以验证这样的条件，而不会导致主项目的构建失败。考虑使用**Ctest**构建和测试模式作为调用 `add_test()` 的 `COMMAND` 参数，从而定义这样的测试用例。

为了主项目的完整配置、构建和测试，考虑**CDash**集成特性提供的功能，而不是使用**Ctest**构建和测试模式。**CDash**集成特性可以更好地捕获整个过程的输出，并提供每个步骤的自定义机制。还有一些额外的特性，可以方便地使用代码覆盖和动态分析工具，比如内存检查器等，而且是否将结果提交给**CDash**服务器，这些特性都可以使用。实际上，驱动整个**CDash**流程的定制**Ctest**脚本功能可以在不使用**CDash**的情况下使用，这使得它可以独立于平台的方式，可以为其他持续集成系统编写整个构建和测试流程的脚本。**CDash**服务器还可以与其他CI系统一起使用，以提供更丰富的特性，用于记录和比较构建历史、失败趋势等。

# 第25章：安装

在使用项目的源代码、创建各种资源、使构建健壮，并实现自动化测试等工作之后，发布已构建的软件是非常重要的一步。它直接影响用户对项目的第一印象，如果做得不好，可能会导致软件在使用之前就遭到拒绝。

开发人员和用户对项目的期望不同。对于某些人来说，仅仅提供对源码库的访问，并让用户自己构建出软件就已经足够了。虽然这可能是交付的一部分，但并非所有用户都希望有如此级别的参与度。相反，他们通常会期望预先构建的二进制包，以便在他们的机器上安装和使用，最好是通过熟悉的包管理系统。给定各种包管理器和交付格式，将给项目维护者带来令人生畏的挑战。然而，它们之间有足够的共同元素，通过一些规划，可以支持大多数主流平台。

项目的生命周期中，交付阶段考虑得越早，最终的打包和部署阶段就越顺利。好的出发点是在开发开始前，或在项目开始前，尽早提出以下问题：

- 应该支持哪些平台，包括最初支持的平台和将来可能支持的平台吗？为了支持项目的特性，是否有最低的平台API或SDK版本要求？
- 每个平台上，用户熟悉的包格式是什么？项目可以以这些格式交付吗？有没有什么特定的软件包格式比其他格式更重要，或者是有强制性的格式需求？
- 是否需要或期望的软件包格式？对软件必须如何布局、构建或交付有需求？项目资源是否必须以特定的格式、方案、位置等方式提供？
- 终端用户想要同时安装软件的多个版本吗？
- 是否应该在没有管理权限的情况下支持安装软件？
- 软件是否可以重定位，以便用户可以将其安装在系统的任何位置（包括在Windows的任何盘符上）？
- 项目是否希望通过PATH环境变量在部署计算机上？提供一个或多个可执行文件？项目中是否有不应该在PATH上暴露的部分？
- 项目是否提供了其他CMake项目构建中使用的东西（库、可执行文件、头文件、资源等）？

这些问题将影响软件在安装时的布局，这反过来又会影响源代码访问自身资源的方式等等。甚至可能影响软件可用的功能，所以尽早理解这些事情可以节省时间，避免浪费精力。

本章着重于布局方面，以及如何将必要的文件组装到需要的位置。还演示了如何通过提供配置包来简化其他CMake项目的使用。具有某些背景的开发人员可能认为，这些方面属于make安装领域。下一章通过讨论CMake和CPack可以产生的各种包格式来完成这个拼图。使用这里描述的安装功能，可以将软件安装到一个干净的区域内，然后将这些内容生成最终的软件包。

## 25.1. 目录结构

决定如何部署产品之前，了解部署平台的约束是一个必要的步骤。只有清楚了这些细节，CMake项目才能开始定义要安装到哪里。可以进行鸟瞰式观察，这些观察可能对一个项目的安装布局有很大的影响。

- Apple对格式（捆绑包、框架等）的规定很严格，这样灵活性就很低，不过也清楚地说明项目需要如何交付。正如在第22章提到的，CMake已经在构建阶段自动处理大部分内容，使应用程序为进行Xcode的最后一步做好准备，即执行最终的应用签名、创建包和提交应用程序。如果使用CMake/CPack的安装阶段，就会是简单地将遵循指定布局的方式进行打包。
- 对于Linux发行版，肯定会有每种类型文件应该安装在何处的指导原则。文件系统层次标准构成了大多数应用布局的基础，其他基于Unix的系统遵循类似的结构。即使不打算直接包含在发行版中，FHS仍然可以作为很好的指南，指导如何构造一个包，以便在许多基于Unix的系统上进行平稳和健壮的安装。

- 有些项目可能希望在用户的PATH环境变量提供一个或多个可执行文件，这样就可以从终端或命令行轻松地调用。**Windows**上，如果项目安装通过添加一个包含它自己的**DLL**的目录来修改**PATH**，那么其他应用程序可能会选择相应**DLL**，而不是之前预装的**DLL**(例如，从自己的私有目录或标准系统的位置)。工具包(如**Qt**)的**DLL**经常成为这种情况的受害者，因为包以不应该的方式修改了**PATH**。如果项目想要将自己的可执行文件增加到**PATH**中，应该确保该目录中不存在无用的**DLL**，并将**DLL**与可执行文件放在同一目录中以便**Windows**在运行时找到它们，从而避免**DLL**冲突。典型的解决方案是创建一个只包含启动脚本的目录，然后安全地将这些脚本添加到**PATH**环境变量中。

### 25.1.1. 相对目录

除了部署到**Apple**平台之外，所有主流平台之间存在很大的共性(或者至少是潜在的共性)。可以认为安装位置由基本路径和该路径下的相对路径组成。基本路径可能类似 `/usr/...`，`/opt/...` 或者 `C:\Program Files`，不同平台之间的差异很大，但在这个基础上的相对路径非常相似。一种常见的安排是将可执行文件(对于**Windows**是 `dll`)安装到 `bin` 目录，将库文件安装到 `lib`，并将头文件目录安装在 `include` 下。其他文件的安装位置可变，但是这三种已经涵盖了项目将要安装的一些重要的文件类型。

**Windows**上，将可执行文件和**DLL**放在基本安装位置，而不是 `bin` 目录下。虽然这是一种常见的做法，但可能导致基目录特别臃肿，使用户很难找到其他组件。另一种是将启动脚本放在 `cmd` 的子目录中，这使它们可以与 `bin` 等其他位置的 **DLL** 分隔开。

找到适用于大多数平台的目录结构不太现实，因为它最小化了由项目源代码实现特定于平台的逻辑。如果项目在所有平台上使用相同的相对布局，那么程序在运行时更容易找到需要的东西。在没有任何其他需求的情况下，**CMake**的 `GNUInstallDirs` 模块提供了非常方便的方法来使用标准目录布局。与上面提到的常见情况一致，还提供了各种符合**GNU**编码标准和**FHS**的标准位置。撇开与基本安装路径相关的部分，甚至可以在 **Windows** 上使用。

使用 `GNUInstallDirs` 模块相当简单：

```
# Minimal inclusion, but see caveat further below
include(GNUInstallDirs)
```

这会创建 `CMAKE_INSTALL_<dir>` 缓存变量，其中 `<dir>` 表示特定的位置。模块文档给出了定义位置的完整信息，但一些常用方式包括：

#### *BINDIR*

直接运行的可执行文件、脚本和符号链接的位置。默认为 `bin`。

#### *SBINDIR*

与 `BINDIR` 相似，不过是针对有系统管理权限的情况。默认为 `sbin`。

#### *LIBDIR*

库和编译文件的路径。根据主机/目标平台，默认设置为 `lib` 或其他(可能包括特定于体系结构的子目录)。

#### *LIBEXECDIR*

不直接由用户调用的可执行文件，但可以通过启动脚本或位于 `BINDIR` 中的符号链接的方式运行。默认为 `libexec`

#### *INCLUDEDIR*

头文件目录。默认为 `include`。

#### *DATAROOTDIR*

只读与结构无关的数据点。为了避开`DOCDIR`的警告，通常不直接引用。

#### `DATADIR`

与结构无关的只读数据，如图像和其他资源。默认值与`DATAROOTDIR`相同，用于覆盖项目数据位置的首选方法。

#### `MANDIR`

`man`格式文档的路径。默认为`DATAROOTDIR/man`。

#### `DOCDIR`

通用文档路径。默认值为`DATAROOTDIR/doc/PROJECT_NAME`(参见下面的注释，了解为什么依赖这个默认值是不安全的)。

每个位置都定义为缓存变量，因此可以重写。开发人员通常不会更改它们，因为安装位置应该在项目的控制之下。对于项目来说，更改默认位置也不可取，但如果项目希望遵循标准布局，需要做一些调整，那么更改位置没什么问题。

`DOCDIR` 位置需要特别提及，因为它默认包含 `PROJECT_NAME` 变量。`PROJECT_NAME` 通过对 `project()` 的调用进行更新，可以在项目结构中变化。`GNUInstallDirs` 模块仅在尚未定义缓存变量时才设置它们，因此 `CMAKE_INSTALL_DOCDIR` 的值将由 `GNUInstallDirs` 模块决定。为了防止这种情况，允许默认的文档目录遵循项目的结构，项目可能需要在每次包含模块时显式地设置 `DOCDIR` 的位置(非缓存变量将覆盖缓存变量)：

```
# Explicitly set DOCDIR location each time
include(GNUInstallDirs)
set(CMAKE_INSTALL_DOCDIR ${CMAKE_INSTALL_DATAROOTDIR}/doc/${PROJECT_NAME})
```

本章的剩余部分中，示例中将使用 `CMAKE_INSTALL_<dir>` 作为安装路径。

### 25.1.2. 安装位置

确定了安装文件的布局之后，必须确定该布局的安装位置。有许多因素会影响这一决定，首先要回答安装是否可重定位。这只是意味着可以使用安装基点，只要保留相对布局，已安装的项目仍按预期工作。可重定位是非常重要，并且应该是大多数项目的目标，例如：

- 可同时安装多个版本。
- 重定位包可以安装到共享驱动器上，这些共享驱动器在不同终端用户的机器上可能有不同的挂载点。
- 一组自包含的可重定位文件，可以更容易地打包成系统包。
- 非管理员用户可以重定位项目在自己本地目录进行安装。

不是所有的项目都可以重定位，需要把文件放在非常特定的位置(例如内核包)。除了少数配置文件外，有些项目可以重定位，有时是将这些特定的文件作为安装后的步骤进行处理(下一章将针对特定的打包系统进行讨论)。

基本安装位置的选择与目标平台密切相关，每个平台都有自己的通用实践和指导原则。`Windows`上，基本安装位置通常是 `C:\Program Files` 目录，而在大多数其他系统上是 `/usr/local` 或 `/opt` 目录。`CMake`提供了许多用于管理安装位置的控件，以抽象出这些平台差异。最重要的是 `CMAKE_INSTALL_PREFIX` 变量，它控制用户构建安装目标时的基本安装位置(该目标可以通过一些生成器类型调用`install`)。`CMAKE_INSTALL_PREFIX` 的默认值是 `C:\Program Files\${PROJECT_NAME}`，基于`Unix`的平台上是 `/usr/local`。

`Linux`上安装时，默认值不符合文件系统层次结构标准。`FHS`要求系统包使用 `/` 或 `/usr` 为基本位置，后者更可能是理想的选择。对于附加包，应该安装到 `/opt/<package>` 或 `/opt/<provider>`，建议使用 `/opt/<provider>`。如果使用的是 `<provider>`，在形式上应该是`LAMA`-注册的名称，或者仅提供包的小写域名。为了避免使用

相同基本安装位置的不同包之间的冲突。对于大多数项目，建议显式地在非Windows平台上设置 `CMAKE_INSTALL_PREFIX` 以兼用FHS路径 `/opt/...`，但是这通常只在顶层CMakeLists.txt中完成，并检查项目是否源码树的顶层(支持分级项目安排)。

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
endif()
```

对于交叉编译的场景，可以定义 `CMAKE_STAGING_PREFIX` 来为安装规则提供安装到的位置。这可以许安装到文件系统，同时保留 `CMAKE_INSTALL_PREFIX` 的其他效果，比如在已安装的二进制文件中嵌入路径(25.2.2节)。`CMAKE_STAGING_PREFIX` 还会影响大多数 `find_...()` 命令。

对于一些打包场景和位置测试安装过程，CMake支持用于非Windows平台的 `DESTDIR` 功能。`DESTDIR` 不是CMake变量，而是传递给构建工具的变量，或者作为构建工具的环境变量。它允许将安装基础位置放置在任意位置，而不是文件系统的根目录。当直接调用构建工具时，例如：

```
make DESTDIR=/home/me/staging install
env DESTDIR=/home/me/staging ninja install
```

`DESTDIR` 功能在概念上有点类似于 `CMAKE_STAGING_PREFIX`，但是 `DESTDIR` 只在安装时指定，不会影响 `find_...()` 命令。`CMAKE_STAGING_PREFIX` 保存为缓存变量，而 `DESTDIR` 是环境变量，构建工具调用之间不保存。

`CMAKE_INSTALL_PREFIX`、`CMAKE_STAGING_PREFIX` 和 `DESTDIR` 的组合为项目和开发人员提供了设置基本安装位置的灵活性，并可以测试安装，而不必实际接触最终的安装位置。但是要注意，不同的打包格式可能有自己的默认安装位置，并且可能会忽略这三个变量，而不是特定于包的变量。

## 25.2. 安装目标

确定了安装区域的目录结构后，可以将注意力转到安装内容本身了。项目使用 `install()` 来定义安装内容，以及放置位置等。这个命令有许多不同的形式，每个形式都作用于第一个参数指定的实体目标。其中一种用于安装的方式：

```
install(TARGETS targets...
[EXPORT exportName]
[CONFIGURATIONS configs...]
# One or more blocks of the following
[ [entityType]
DESTINATION dir
[PERMISSIONS permissions...]
[NAMELINK_ONLY | NAMELINK_SKIP]
[COMPONENT component]
[NAMELINK_COMPONENT component] # CMake 3.12 or later only
[EXCLUDE_FROM_ALL]
[OPTIONAL]
[CONFIGURATIONS configs...]
]...
# Special case
[INCLUDES DESTINATION incDirs...]
)
```

提供一个或多个目标，`entityType` 指定如何安装目标的各个部分。每个目标都必须与 `install()` 在相同的目录范围内，并且 `entityType` 必须是以下类型之一：

**RUNTIME**

安装可执行的二进制文件。**Windows**上还会安装库目标的**DLL**。**Apple**捆绑包不包含在其中。

### **LIBRARY**

除**Windows**和**Apple**框架之外的所有平台上安装动态库。

### **ARCHIVE**

除了**Apple**框架外，安装静态库(所有平台)。**Windows**上，这还会安装动态库的导入库(即.**.lib**)部分。

### **OBJECTS**

安装与对象库关联的对象(仅适用于**CMake 3.9**或更高版本)。

### **FRAMEWORK**

**Apple**平台上，安装框架(动态或静态)，包括复制到框架中的任何内容(例如，通过 `POST_BUILD` 自定义规则)。

### **BUNDLE**

**Apple**平台上，安装捆绑包，包括复制到捆绑包中的内容。

### **PUBLIC\_HEADER**

非**Apple**平台上，这将安装框架库目标的 `PUBLIC_HEADER` 属性中的文件。**Apple**平台上，这些头文件看作为框架的一部分，但是对于非**Apple**平台，这些目标视为普通的动态库，并且头文件需要单独安装。

### **PRIVATE\_HEADER**

类似于 `PUBLIC_HEADER`，但会受 `PRIVATE_HEADER` 属性的影响。

### **RESOURCE**

非**Apple**平台上，这将安装框架或捆绑目标的 `RESOURCE` 属性中的文件。**Apple**平台上，这些文件作为 `FRAMEWORK` 或 `BUNDLE` 的一部分。

`entityType` 之后，可以列出各种选项。例如，下面展示了如何在所有平台(不包括**Apple**框架)上，以某种方式安装库，将各自的部分放在期望的位置上：

```
install(TARGETS mySharedLib myStaticLib
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
    )
```

上面的示例展示了 `DESTINATION` 如何为同一个目标的不同部分指定不同的位置。该命令足够灵活，可以同时处理多个不同类型的目标。

- 对于**mySharedLib**，**Windows**上，**DLL**将转到 `RUNTIME`，导入库转到 `ARCHIVE`。其他平台上，动态库将安装到 `LIBRARY`。
- **myStaticLib**目标的静态库将安装到 `ARCHIVE`。

如果目标没有对应的 `entityType`，**CMake**通常会发出警告或错误(例如，其中一个目标是静态库，但没有提供 `ARCHIVE`)。`entityType` 可以省略，目标列表后面的选项将应用于所有实体类型。通常目标只有一种实体类型时，才会这样做：

```
# Targets are both executables, so specifying the entity type isn't needed
install(TARGETS exe1 exe2
        DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

后面的选项可以指定的不仅仅是目标。还可以使用权限选项覆盖默认权限，指定一个或多个与18.2节中的文件(复制)命令相同的值：

OWNER_READ	OWNER_WRITE	OWNER_EXECUTE
GROUP_READ	GROUP_WRITE	GROUP_EXECUTE
WORLD_READ	WORLD_WRITE	WORLD_EXECUTE
SETUID	SETGID	

对于文件(复制)，平台不支持的权限将忽略。请注意，CMake在默认情况下会为所有目标设置适当的权限，所以通常只需要明确地提供权限。如果安装的位置需要比正常情况下更多的权限，或者需要添加 SETUID 或 SETGID，类似如下情况：

```
# Intended to only be run by an administrator, so only allow the owner to have access
install(TARGETS onlyOwnerCanRunMe
        DESTINATION ${CMAKE_INSTALL_SBINDIR}
        PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
)

# Install with set-group permission
install(TARGETS runAsGroup
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
        GROUP_READ GROUP_EXECUTE SETGID
)
```

对于 LIBRARY 类型，当为库目标提供了版本信息时，一些平台支持创建符号链接(参见20.3节)。动态库可能存在的文件和符号链接通常是这样的：

```
libmyShared.so.1.3.2 ①
libmyShared.so.1 --> libmyShared.so.1.3.2 ②
libmyShared.so --> libmyShared.so.1 ③
```

① 由项目构建的实际二进制文件。② 符号链接文件，其名称是实际库的名称。当遵循语义版本控制时，将只在其名称中包含版本的主要部分。③ 文件名中没有嵌入版本信息的Namalink。当链接器命令行有 -lmyShared 这样的选项时，就可以通过该文件找到实际的库文件。

安装库实体文件时，可以提供 NAMELINK\_ONLY 或 NAMELINK\_SKIP 选项。NAMELINK\_ONLY 选项将只安装namalink文件，而 NAMELINK\_SKIP 将安装除了namalink之外的所有内容。如果库目标没有版本信息，或者平台不支持 namalink，这两个选项的行为就会改变。NAMELINK\_ONLY 将不安装任何东西，而 NAMELINK\_SKIP 将安装真正的库文件。当创建单独的运行时和开发包时，这些选项特别有用，namalink安装入开发包后，其他文件/链接会安装入运行时包。当给出 NAMELINK\_ONLY 选项时，CMake不会警告 install() 中缺少的内容。之所以这样做，是因为不能在同一个 install() 中同时使用 NAMELINK\_SKIP 和 NAMELINK\_ONLY，因此必须在不同的调用中对两者进行拆分(参见下面的示例)。

每个 entityType 都可以指定 COMPONENT 选项。组件用于打包的逻辑，将在下一章详细讨论，现在可以看作是分离不同安装集的一种方式。上面提到的独立运行时和开发包场景，可以设置成如下方式：

```

install(TARGETS myShared myStatic
RUNTIME
  DESTINATION ${CMAKE_INSTALL_BINDIR}
  COMPONENT MyProj_Runtime
LIBRARY
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  NAMELINK_SKIP
  COMPONENT MyProj_Runtime
ARCHIVE
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  COMPONENT MyProj_Development
)

# Because NAMELINK_ONLY is given, CMake won't complain about a missing RUNTIME block
install(TARGETS myShared
LIBRARY
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  NAMELINK_ONLY
  COMPONENT MyProj_Development
)

```

CMake 3.12中，使用 `NAMELINK_COMPONENT` 选项可以更简单地将nameLink拆分为不同的组件。此选项可以与 `COMPONENT` 一起使用，但只能在 `LIBRARY` 中使用。使用这种方式，可以更简明地描述上述情况：

```

install(TARGETS myShared myStatic
RUNTIME
  DESTINATION ${CMAKE_INSTALL_BINDIR}
  COMPONENT MyProj_Runtime
LIBRARY
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  COMPONENT MyProj_Runtime
  NAMELINK_COMPONENT MyProj_Development # Requires CMake 3.12 or later
ARCHIVE
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  COMPONENT MyProj_Development
)

```

如果没有为块提供任何 `COMPONENT`，那么将与一个默认组件相关联，该组件的名称由 `CMAKE_INSTALL_DEFAULT_COMPONENT_NAME` 提供。如果未设置该变量，则将未指定的变量用作默认组件名称。第三方子项目不使用任何安装组件的情况下，更改默认组件名可能会有用。为了使子项目的安装构件与主项目分离，可以用 `add_subdirectory()` 将子项目拉入主构建之前更改默认名称。

可以使用 `EXCLUDE_FROM_ALL` 选项仅安装特定的组件。默认情况下，安装不指定组件，并且会安装所有组件，但是打包实现可以单独安装特定组件。CMake 3.12中还添加了文档来演示如何从命令行执行此操作。对于大多数项目，可能不太需要 `EXCLUDE_FROM_ALL`。

`OPTIONAL` 关键字也很少使用。如果目标存在，但文件缺失(例如，Windows DLL中 `ARCHIVE` 类型的导入库)，而CMake不会认为是错误的。因为可以隐藏构建/安装逻辑的错误配置，所以请谨慎使用此选项。

还可以通过添加 `CONFIGURATIONS` 选项使用特定的配置。只有当前构建类型是所列出的类型之一时，才会安装该实体类型。对于单个 `install()`，实体类型不能列出多次，因此如果不同的配置需要不同的信息，则需要多次调用。下面的例子展示了如何在不同的目录中安装调试版本和发布版本的静态库：

```

install(TARGETS myStatic
ARCHIVE
DESTINATION ${CMAKE_INSTALL_LIBDIR}/Debug
CONFIGURATIONS Debug
)
install(TARGETS myStatic
ARCHIVE
DESTINATION ${CMAKE_INSTALL_LIBDIR}/Release
CONFIGURATIONS Release RelWithDebInfo MinSizeRel
)

```

`CONFIGURATIONS` 关键字还可以位于所有参数的前面，并作为没提供配置的默认值的默认值。下面的示例中，除了为调试和发布安装的 `ARCHIVE` 块外，所有的块都只为发布版安装。

```

install(TARGETS myShared myStatic
CONFIGURATIONS Release
RUNTIME
DESTINATION ${CMAKE_INSTALL_BINDIR}
LIBRARY
DESTINATION ${CMAKE_INSTALL_LIBDIR}
ARCHIVE
DESTINATION ${CMAKE_INSTALL_LIBDIR}
CONFIGURATIONS Debug Release
)

```

### 25.2.1. 接口属性

如果导出目标，就要设置接口属性，供其他项目的目标使用。各种接口目标属性将自动将已安装的目标细节导出，但是需要特殊处理，以满足构建目标和使用已安装目标的需求。考虑以下代码示例：

```

add_library(foo STATIC ...)
target_include_directories(foo
INTERFACE ${CMAKE_CURRENT_BINARY_DIR}/somewhere
${MyProject_BINARY_DIR}/anotherDir
)
install(TARGETS foo
DESTINATION ...
)

```

构建中，任何链接到`foo`的内容都会有某个位置的绝对路径，并且在头文件搜索路径中添加一个 `anotherDir`。当`foo`安装时，可以打包并部署到完全不同的机器上。显然，到某个位和 `anotherDir` 的路径将不再有意义，但上面的例子将添加到目标的头文件搜索路径中。需要提一下的是，“构建时使用路径`xxx`，安装时使用路径`yyy`”的方式，这正是 `BUILD_INTERFACE` 和 `INSTALL_INTERFACE` 生成器表达式所做的：

```

include(GNUInstallDirs)
target_include_directories(foo
INTERFACE
$<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/somewhere>
$<BUILD_INTERFACE:${MyProject_BINARY_DIR}/anotherDir>
$<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)

```

`$<BUILD_INTERFACE:xxx>` 将构建树扩展为`xxx`，在安装时扩展为空，而 `$<INSTALL_INTERFACE:yyy>` 则相反，确保`yyy`只针对已安装目标添加。`INSTALL_INTERFACE` 中，`yyy`通常是相对路径，可视为相对于基本安装位置的路径。

虽然构建树中的头文件搜索路径可能因目标而异，但在安装之后，目标通常都共享相同的头文件搜索路径。上面的示例中，使用了 `CMAKE_INSTALL_INCLUDEDIR`，并且可能会对每个可安装的目标重复使用，但是为每个目标分别指定并不是最方便的方法。可以使用 `install()` 命令的 `include` 选项为一组目标指定相同的信息。`INCLUDES DESTINATION` 之后给出的所有目录都添加到列出目标的 `INTERFACE_INCLUDE_DIRECTORIES` 属性中。这使得头文件搜索路径的描述更加简洁。

```
add_library(myStatic STATIC ...)
add_library(myHeaderOnly INTERFACE ...)

target_include_directories(myStatic
    PUBLIC ${CMAKE_CURRENT_BINARY_DIR}/static_exports
)
target_include_directories(myHeaderOnly
    INTERFACE ${CMAKE_CURRENT_LIST_DIR}
)

install(TARGETS myStatic myHeaderOnly
    ARCHIVE
        DESTINATION ${CMAKE_INSTALL_LIBDIR}
    INCLUDES
        DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

与其他参数不同，可以为 `INCLUDES DESTINATION` 列出多个目录，尽管在实践中这可能不太常见。还要注意，`INCLUDES` 块不支持其他 `entityType` 块支持的信息，只指定 `DESTINATION` 关键字，之后可跟一个或多个位置。

## 25.2.2. RPATH

当操作系统加载库或可执行文件时，必须找到二进制文件链接到的所有动态库。不同的平台有不同的处理方式，**Windows**依赖于 `PATH` 环境变量中的位置，以及二进制文件所在的目录来查找所需要的库。其他平台使用专用于此目的的环境变量，比如 `LD_LIBRARY_PATH`，比如在 `conf` 文件中列出的库。环境变量的缺点是，它依赖于加载二进制文件的人或进程正确地配置了环境。

许多情况下，提供二进制文件的包已经知道在哪里可以找到依赖库，因为它们可能是在同一个包里。大多数非**Windows**平台都支持二进制文件将库搜索路径直接硬编码到文件中。此特性的通用名称是 `run path` 或 `RPATH`，实际名称可能与平台有关。通过嵌入 `RPATH` 信息，二进制文件可以是自包含的，并且不必依赖于环境或系统配置提供的任何路径。此外，`RPATH` 可以包含某些占位符，这些占位符允许定义只在运行时解析为绝对路径的相对路径。占位符允许基于二进制文件的位置进行解析，因此可重定位包，可以定义 `RPATH` 细节，这些信息只能基于包的相对布局对路径进行硬编码。

与上一节中接口属性一样，构建中的 `RPATH` 与已安装的二进制文件存在需求冲突。构建中开发人员需要二进制文件能够找到需要链接的动态库，这样可执行文件才能运行(调试、测试执行等)。支持 `RPATH` 的平台，**CMake**将默认嵌入所需的路径，从而为开发人员提供最方便的体验，而不需要进行任何设置。但 `RPATH` 只适合特定的构建，因此在安装目标时，**CMake**会用替换路径重写(默认替换成空的 `RPATH`)。

使用 `RPATH` 默认值还是合理的，但是不太适合需要安装的目标。项目将会覆盖默认行为，以确保构建树和安装目标得到适当的满足。**CMake**允许单独控制构建和安装 `RPATH` 位置，因此项目可以实现适合自己的策略。以下目标属性和变量可以用于影响 `RPATH`：

### `BUILD_RPATH`

此目标属性可用于嵌入到构建树的二进制文件中的搜索路径。这是**CMake**为二进制文件的链接依赖，自动添加的附加路径，因此指定**CMake**的路径，而不是使用外部路径。只有当二进制文件在运行时使用 `dlopen()` 或其他等效机制加载非链接库时(比如加载可选插件模块)，才需要这个属性。这个属性

在 `add_library()` 或 `add_executable()` 创建目标时由 `CMAKE_BUILD_RPATH` 的值初始化。`BUILD_RPATH` 属性和 `CMAKE_BUILD_RPATH` 变量在CMake 3.8中添加。

#### `INSTALL_RPATH`

此目标属性指定安装二进制文件时的 `RPATH`。与 `BUILD_RPATH` 不同，CMake在默认情况下不提供 `INSTALL_RPATH`，因此项目应该将此属性设置为已安装的路径。当创建目标时，该属性由 `CMAKE_INSTALL_RPATH` 变量值初始化。

#### `INSTALL_RPATH_USE_LINK_PATH`

当此目标属性设置为`true`时，目标链接的每个库的路径将添加到安装 `RPATH` 位置集中，当该路径指向项目的源目录和二进制目录外的位置时才如此。这主要将绝对路径嵌入到不属于项目的外部库，这些库应该位于项目将部署到的位置。要谨慎使用这个属性，这样的假设可能会降低已安装包的健壮性(路径可能会随着外部库的版本而改变，系统管理员可能会选择非默认的安装配置等)。这个属性在创建目标时由 `CMAKE_INSTALL_RPATH_USE_LINK_PATH` 值初始化。

#### `BUILD_WITH_INSTALL_RPATH`

有些项目的安装布局与构建布局一样，安装 `RPATH` 也适合于构建树。通过将这个目标属性设置为`true`，就不会使用 `BUILD_RPATH`，而在构建时将 `INSTALL_RPATH` 嵌入到二进制文件中。注意，当使用加载器支持而链接器不支持的占位符(后面将进一步讨论)时，这可能会导致链接过程中的构建问题。这个属性在创建目标时由 `CMAKE_BUILD_WITH_INSTALL_RPATH` 值初始化。

#### `SKIP_BUILD_RPATH`

当此目标属性设置为`true`时，不设置 `BUILD_RPATH`。将忽略 `BUILD_RPATH`，CMake不会自动为目标链接到的库添加 `RPATH`。请注意，如果依赖库链接到其他库，可能导致构建失败，请谨慎使用。创建目标时，该属性由 `CMAKE_SKIP_BUILD_RPATH` 值初始化。如果 `BUILD_WITH_INSTALL_RPATH` 属性设置为`true`，也会重载。

#### `CMAKE_SKIP_INSTALL_RPATH`

安装时相当于 `CMAKE_SKIP_BUILD_RPATH`。将其设置为`true`会直接忽略 `INSTALL_RPATH` 目标属性，并可能导致已安装的目标在运行时无法找到依赖库，因此有用性值得怀疑。注意，这里没有 `SKIP_INSTALL_RPATH` 目标属性，只有 `CMAKE_SKIP_INSTALL_RPATH`。

#### `CMAKE_SKIP_RPATH`

将该变量设置为`true`将禁用所有 `RPATH` 支持，并忽略上述所有属性和变量。除非项目以其他方式管理运行时库的加载，否则通常不希望这样做。

安装 `RPATH` 位置应该基于相对路径。在大多数基于Unix的平台上通过使用 `$ORIGIN` 占位符，来表示嵌入 `RPATH` 的二进制文件的位置。例如，下面是定义安装 `RPATH` 的常用方法，该方法适用于与默认 `GNUInstallDirs` 模块布局类似的项目：

```
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/../lib)
```

要使其更加健壮并考虑默认布局的潜在变化，还需要做一些工作。需要计算出从可执行文件目录到库文件目录的相对路径，实现如下：

```
include(GNUInstallDirs)
file(RELATIVE_PATH relDir
${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
set(CMAKE_INSTALL_RPATH $ORIGIN $ORIGIN/${relDir})
```

上面定义的所有目标都有 `INSTALL_RPATH`，引导加载程序查找与二进制文件相同的目录，以及类似于 `../lib` 或二进制文件的位置。因此，对于安装到`bin`的可执行文件和安装到`lib`的动态库，这将确保两者都能找到项目提供的其他库。第一次向项目添加 `RPATH` 支持时，建议将此作为起点。注意，Apple的目标工作方式略有不同，可能会有不同的布局，因此上面的内容需要调整以覆盖该平台(下一节将讨论)。

需要注意的问题是，加载器可以理解 `$ORIGIN`，而链接器很可能不能。当链接到某个库，而该库本身又链接到另一个库时，可能会出问题。第一级的链接不会出现问题，因为库会直接在链接器命令行上列出，但是第二级的库依赖关系必须由链接器找到。当链接器不理解 `$ORIGIN` 时，无法通过 `RPATH` 信息找到第二层库。因此，除非路径也由其他选项(如 `-L`)指定，否则就算第一级库技术上包含了所需的所有信息，链接也会失败。这是一个众所周知的问题，并不是CMake特有的，它是链接器(尤其是GNU ld链接器)的一个弱点。

根据上面提到的各种属性和变量，可能需要CMake在安装目标时更改嵌入的 `RPATH` 信息。有两种方法可以做到这一点。如果二进制文件是ELF格式的，默认情况下CMake使用内部工具在安装的二进制文件中直接重写 `RPATH`。ELF头文件中的 `RPATH` 值大小固定，CMake在必要时会填充 `BUILD_RPATH`，需要确保有足够的空间用于 `INSTALL_RPATH`。除了构建时链接器的选项外，开发人员基本上不知道实现的具体细节。对于非ELF平台，CMake在安装时重新链接二进制文件，而不是指定 `INSTALL_RPATH` 信息。这有时会使开发人员感到困惑，他们想知道为什么构建的东西需要再次链接，但最终重新链接是获得预期结果的一种方法。ELF平台也可以通过将 `CMAKE_NO_BUILTIN_CRPATH` 变量设置为`true`，来强制重链接行为，除非内部 `RPATH` 重写失败，否则不应该使用这种方法。

交叉编译时，其他一些变量可以修改嵌入到二进制文件中的 `RPATH`。任何以 `CMAKE_STAGING_PREFIX` 开始的 `RPATH` 位置都自动的将前缀替换为 `CMAKE_INSTALL_PREFIX`，对于构建和安装 `RPATH` 位置都是如此。任何以 `CMAKE_SYSROOT` 开头的安装 `RPATH` 位置都将去掉这个前缀。

### 25.2.3. Apple平台上的特定目标

Apple的加载器和链接器与其他Unix平台的工作方式不同。Linux等平台上的库只将库名编码为动态库(例如 `soname`)，而Apple平台则编码库的完整路径。这个完整路径称为 `install_name`，而 `install_name` 的路径部分称为 `install_name_dir`。对该库的链接都会将完整的 `install_name` 编码为要搜索的库。当所有的东西都安装到预期位置时没问题，但是对于可重定位包(包括大多数应用程序包)，这就太不灵活了。作为一种处理方法，Apple支持类似 `$ORIGIN` 的基点，但是占位符不同：

`@loader_path`

这相当于Apple的 `$ORIGIN`，但链接器能够理解，因此不会遇到其他链接器无法解码 `$ORIGIN` 的问题。

`@executable_path`

这个变量会使用正在执行程序的位置。对于依赖项引入的库，这种方法的帮助不大，因为要知道使用可执行文件的位置。这通常是不确定的，因此 `@loader_path` 通常是更好的选择。

`@rpath`

可以作为 `install_name_dir` 部分的占位符，也可以完全替换 `install_name_dir`。

`@loader_path` 和 `@rpath` 的组合可以用于实现与其他支持 `$ORIGIN` 的Unix平台相似的行为。CMake提供Apple特定的控制项，以便在Apple平台进行适当的设置：

`MACOSX_RPATH`

这个目标属性设置为`true`时，CMake在为Apple平台构建时自动将 `install_name_dir` 设置为`@rpath`。这是CMake 3.0之后默认的行为。该变量可以使用 `INSTALL_NAME_DIR` 覆盖。如果创建目标时设置了 `CMAKE_MACOSX_RPATH`，则使用 `MACOSX_RPATH` 的属性值初始化。

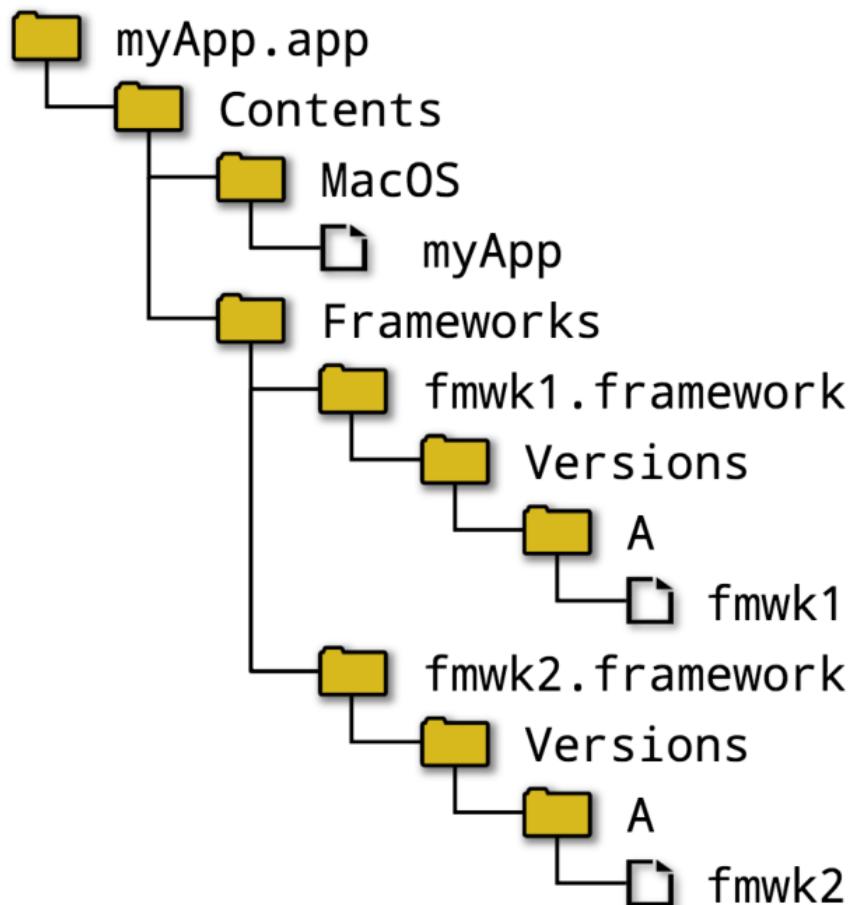
`INSTALL_NAME_DIR`

这个目标属性用于显式设置库的 `install_name` 中的 `install_name_dir` 部分。默认的 `install_name` 的形式通常 是 `@rpath/libsomename`。但是对于 `@rpath` 的情况不合适，`INSTALL_NAME_DIR` 可以指定替代方案。该属性在创建时使用 `CMAKE_INSTALL_NAME_DIR` 初始化。该属性会在非Apple平台上无效化。

对于非包布局，`$ORIGIN` 也可以扩展到Apple平台：

```
if(APPLE)
set(basePoint @loader_path)
else()
set(basePoint $ORIGIN)
endif()
include(GNUInstallDirs)
file(RELATIVE_PATH relDir
${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}
${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR}
)
set(CMAKE_INSTALL_RPATH ${basePoint} ${basePoint}/${relDir})
```

当使用了Apple包或框架，Apple的布局就会与其他平台完全不同，上面的策略就没用了。这种情况，有不同的策略来定义运行时搜索路径。例如，macOS应用包在安装了相关目标后，或者在构建后复制了框架(只显示了包结构的相关部分)，最终可能会形成如下结构：



上面 `RPATH` 的信息可以通过将 `myApp` 的 `INSTALL_RPATH` 目标属性设置为 `@executable_path/..` 来实现。`fmwk1` 和 `fmwk2` 会设置为 `@loader_path/../../`。要支持构建后框架的复制，还可以在构建时使用安装信息 `RPATH`。省略了构建后框架复制和代码签名的信息，这样的安排可能看起来像这样：

```
set(CMAKE_BUILD_WITH_INSTALL_RPATH YES)
set(CMAKE_BUILD_WITH_INSTALL_NAME_DIR YES)

add_executable(myApp MACOSX_BUNDLE ...)
add_library(fmwk1 SHARED ...)
add_library(fmwk2 SHARED ...)
target_link_libraries(myApp PRIVATE fmwk1) # Only needs fmwk1 directly...
target_link_libraries(fmwk1 PRIVATE fmwk2) # ... but fmwk1 needs fmwk2

set_target_properties(myApp PROPERTIES
  INSTALL_RPATH @executable_path/../Frameworks
)
set_target_properties(fmwk1 fmwk2 PROPERTIES
  FRAMEWORK TRUE
  INSTALL_RPATH @loader_path/../../..
)
```

如果项目的策略是只在安装时嵌入框架，那么像下面这样就行了：

```
install(TARGETS fmwk1 fmwk2 myApp
  BUNDLE DESTINATION .
  FRAMEWORK DESTINATION myApp.app/Contents/Frameworks
)
```

另一方面，如果项目希望在构建时嵌入框架，可以相对容易地实现构建后步骤，如下示例所示。但是请注意，`TARGET_BUNDLE_DIR` 和 `TARGET_BUNDLE_CONTENT_DIR` 生成器表达式仅在CMake 3.9或更高版本中可用。

```
add_custom_command(TARGET myApp POST_BUILD
  COMMAND rsync -a
    ${TARGET_BUNDLE_DIR:fmwk1}
    ${TARGET_BUNDLE_DIR:fmwk2}
    ${TARGET_BUNDLE_CONTENT_DIR:myApp}/Frameworks/
)
```

上面的复制步骤有健壮性问题，比如不能删除旧内容，但是对于某些情况来说，已经足够好了，至少是好的起点。

如果包需要签名，CMake通常不会很好地支持嵌入框架。如第22章中强调的那样，Apple假设代码签名是由Xcode作为构建过程的一部分，而不是作为安装后的步骤，CMake在签名过程中提供的帮助很少。当前，如果项目希望使用嵌入式框架签署包，必须使用自己的逻辑实现。

如果项目希望为iOS创建通用二进制文件(有时也称为宽二进制文件)，则会出现另一个复杂问题。构建可以是设备端的，也可以是模拟器端的。通常只安装一种架构，可以通过CMake 3.5及以后版本提供了`IOS_INSTALL_COMBINED` 目标属性。如果此属性为`true`，当为设备构建安装目标时，还将构建模拟器体系结构，安装时将两者合并为单个二进制文件。反之亦然，安装模拟器构建也会构建和安装。如果该目标属性有关，则该特性依赖于项目实现的代码签名逻辑。

涉及到在框架中嵌入头文件时，CMake提供了更多的帮助。如22.3节所述，目标可以在`PUBLIC_HEADER` 和 `PRIVATE_HEADER` 目标属性中列出公共和私有头文件。在安装框架本身的过程中安装它们，不需要进一步配置。当这些相同的目标在非Apple平台上构建时，不会有任何框架结构来容纳头文件(目标文件将视为动态库)，但头文件仍然可以安装到指定位置：

```
install(TARGETS myShared
FRAMEWORK # Apple framework case
DESTINATION ...
LIBRARY # Non-Apple case
DESTINATION ...
PUBLIC_HEADER
DESTINATION ...
PRIVATE_HEADER
DESTINATION ...
)
```

## 25.3. 安装导出

安装目标时，可以使用带有 `EXPORT` 选项的 `install(TARGETS)`，`EXPORT` 选项会指定导出集的名称，然后使用命令的另一种形式安装导出集：

```
install(EXPORT exportName
DESTINATION dir
[FILE name.cmake]
[NAMESPACE namespace]
[PERMISSIONS permissions...]
[EXPORT_LINK_INTERFACE_LIBRARIES]
[COMPONENT component]
[EXCLUDE_FROM_ALL]
[CONFIGURATIONS configs...]
)
```

安装导出集会在指定的目标目录中创建具有指定名称的 `name.cmake` 文件(必须以`.cmake`结尾)。如果没有提供 `FILE` 选项，则使用基于 `exportName` 的默认文件名。生成的文件将包含CMake为每个导入目标的出口集合。此文件的目的是让其他项目包含它，以便引用此项目的目标，并拥有关于接口属性和目标间关系的完整信息。通过一些限制，项目可以像对待常规目标一样对待导入的目标。这些导出文件通常不能直接包含，它们用于配置包，然后使用 `find_package()` 来找到配置包(这在25.7节中有更详细的介绍)。

如果指定了 `NAMESPACE` 选项，在创建与之关联的导入目标时，每个目标都将在其名称前加上命名空间。考虑下面的例子：

```
add_library(myShared SHARED ...)
add_library(BagOfBeans::myShared ALIAS myShared)

install(TARGETS myShared
EXPORT BagOfBeans
DESTINATION ${CMAKE_INSTALL_LIBDIR}
)
install(EXPORT BagOfBeans
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/BagOfBeans
NAMESPACE BagOfBeans::
)
```

上面的示例遵循了16.4节的建议，其中每个常规目标有一个与之关联的命名空间别名。当为非别名 `myShared` 目标安装导出时，可以使用与别名目标相同的名称空间(即 `BagOfBeans::`)。这允许导出引用目标的详细信息，就像这个项目引用别名一样(`BagOfBeans::myShared`)。项目可以选择通过 `add_subdirectory()` 直接添加这个项目，或者通过 `find_package()` 拉入导出文件，不管选择了哪个方法，仍然可以使用相同的 `BagOfBeans::myShared` 目标名。在CMake社区中，这个重要的模式正在成为普遍的方式，因此尝试遵循它，能更好的兼容大多数项目。

`EXPORT` 关键字之后给出的导出集名称不必与 `NAMESPACE` 相关。命名空间通常与项目名称紧密关联，但是可以使用一系列不同的策略来命名导出集。例如，项目可以定义多个导出集，其目标共享单个命名空间，并且导出集可能对应于整个安装逻辑单元。这些导出集可能与安装 `COMPONENT` 一一对应，也可能收集多个组件。以下是案例演示：

```
# Single component export
install(TARGETS algo1 EXPORT MyProj_algoFree
DESTINATION ... COMPONENT MyProj_free
)
install(EXPORT MyProj_algoFree
DESTINATION ... COMPONENT MyProj_free
)
```

```
# Multi component export
install(TARGETS algo2 EXPORT MyProj_algoPaid
DESTINATION ... COMPONENT MyProj_licensed_A
)
install(TARGETS algo3 EXPORT MyProj_algoPaid
DESTINATION ... COMPONENT MyProj_licensed_B
)
install(EXPORT MyProj_algoPaid
DESTINATION ... COMPONENT MyProj_licensed_dev
)
```

上面的示例中，导出集只包含 `algo1` 目标，它是 `MyProj_free` 组件的成员。导出文件也是 `MyProj_free` 组件的成员，因此在安装该组件时，库和导出文件将一起安装。对于多组件导出，情况有所不同，其中导出集包含来自 `MyProj_licensed_A` 组件的 `algo2` 和来自 `MyProj_licensed_B` 组件的 `algo3`，但导出文件在它自己的单独组件中。因此，可以使用或不使用导出文件安装目标，这取决于是否安装 `MyProj_licensed_dev` 组件。

上面的多组件导出案例，强调了如何安装导出集和组件。安装导出文件，而不同时安装导出文件指向的实际目标是错误的。因此，如果用户安装了 `MyProj_licensed_dev` 组件，也必须安装 `MyProj_licensed_A` 和 `MyProj_licensed_B` 组件。

`install(EXPORT)` 的其余选项中，一些选项具有与 `install(TARGETS)` 的效果类似。`PERMISSIONS`、`EXCLUDE_FROM_ALL` 和 `CONFIGURATIONS` 选项应用于已安装的导出文件，而不是目标本身，但在其他方面等效。`install(EXPORT)` 使用的目标由项目决定，但遵循一些约定可能会有用。这样做的动机与导出文件用作配置包的一部分有关，所以关于这个主题的讨论会在25.7节之后进行。

`EXPORT_LINK_INTERFACE_LIBRARIES` 选项用于支持CMake 3.0之前的行为，并与链接接口库相关。不鼓励使用它，建议项目至少以3.0作为CMake的最低版本。

有一个非常类似的 `install()` 形式，专门用于导出Android ndk-build项目的目

```
install(EXPORT_ANDROID_MK exportName
DESTINATION dir
[FILE name.mk]
[NAMESPACE namespace]
[PERMISSIONS permissions...]
[EXPORT_LINK_INTERFACE_LIBRARIES]
[COMPONENT component]
[EXCLUDE_FROM_ALL]
[CONFIGURATIONS configs...]
)
```

`install(EXPORT)` 创建供其他CMake项目使用的文件，`install(EXPORT_ANDROID_MK)` 创建一个Android.mk，NDK构建可以包含该文件。Android.mk提供了附加到导出目标的所有使用要求，因此ndk-build项目将了解所有编译器定义、头文件搜索路径等信息。可以使用 `FILE` 选项更改导出文件的名称，但必须以.mk结尾。其他选项与 `install(EXPORT)` 方式具有相同的行为。`install(EXPORT_ANDROID_MK)` 需要CMake 3.7或更高版本，但项目至少需要3.11版本，以避免受私有依赖关系静态库的Bug影响。

某些情况下，无法导出文件进行安装。示例场景包括针对主构建的不同平台编译的子构建，或者由于目标名称冲突、滥用 `CMAKE_SOURCE_DIR` 等变量而不能直接添加到主构建中的第三方项目。对于这些情况，CMake提供了 `export()`，可以直接将导出文件写入到构建树中：

```
export(EXPORT exportName
      [NAMESPACE namespace]
      [FILE fileName]
      )
```

除了要编写导出文件外，上面的操作基本上相当于简化的 `install(EXPORT)`。尽管文件名可以包含路径(仍然以.cmake结尾)，但可用选项的简化与 `install(EXPORT)` 具有相同的含义。其他形式的 `export()` 允许导出单个目标，如果定义了导出集，那么上面的是最容易使用和维护的方式。

## 25.4. 安装文件和目录

与目标相比，安装单个文件和目录不那么复杂。文件安装可以使用以下形式：

```
install(<FILES | PROGRAMS> files...
       DESTINATION dir
       [RENAME newName]
       [PERMISSIONS permissions...]
       [COMPONENT component]
       [EXCLUDE_FROM_ALL]
       [OPTIONAL]
       [CONFIGURATIONS configs...]
       )
```

大多数选项已经很熟悉了，含义与 `install(TARGETS)` 相同。`install(FILES)` 和 `install(PROGRAMS)` 的区别是，如果没有 `PERMISSIONS`，后者会在默认情况下添加执行权限。这是为了安装可执行的脚本或程序(Shell脚本)，但不是CMake的目标。`RENAME` 选项只能操作单个文件，允许在安装时给该文件一个新名称。

某些情况下，项目可能希望安装与导入的目标相关联的二进制文件，但是 `install(TARGETS)` 不允许直接安装导入目标。解决这个问题的方法是将与导入目标关联的文件安装为普通文件。与目标相关联的使用需求将不会保留，但至少可以安装二进制文件，使用 `$<TARGET_FILE:>` 生成器表达式和其他类似的表达式特别有用。这样的缺点是，将处理所有平台差异的责任放回到项目身上，这对于导入的库目标来说是有问题的。

```
# Assume myImportedExe is an imported target for an executable not built by this project
install(PROGRAMS $<TARGET_FILE:myImportedExe>
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        )
```

安装目录遵循与文件类似的模式，扩展了支持的选项集：

```
install(DIRECTORY dirs...
DESTINATION dir
[FILE_PERMISSIONS permissions... | USE_SOURCE_PERMISSIONS]
[DIRECTORY_PERMISSIONS permissions...]
[COMPONENT component]
[EXCLUDE_FROM_ALL]
[OPTIONAL]
[CONFIGURATIONS configs...]
[MESSAGE_NEVER]
[FILES_MATCHING]
# The following block can be repeated as many times as needed
[ [PATTERN pattern | REGEX regex]
[EXCLUDE]
[PERMISSIONS permissions...] ]
)
)
```

没有任何可选参数的情况下，对于每个 `dirs` 位置，从该点开始的整个目录树将安装到目标目录中。如果源名称以斜杠结尾，则复制源目录的内容，而不是复制源目录本身。

```
# Results in somewhere/foo/...
install(DIRECTORY foo DESTINATION somewhere)

# Results in somewhere/...
install(DIRECTORY foo/ DESTINATION somewhere)
```

`COMPONENT`、`EXCLUDE_FROM_ALL`、`OPTIONAL` 和 `CONFIGURATIONS` 选项与其他 `install()` 有相同的含义。`MESSAGE_NEVER` 选项禁止为已安装文件提供日志消息。

支持选项来控制文件和目录的权限。如果给定了 `USE_SOURCE_PERMISSIONS`，则安装的每个文件将保留与其源文件相同的权限。`FILE_PERMISSIONS` 将覆盖该权限并使用指定的权限。如果两个选项都没有给出，文件将具有相同的默认权限，就像使用了 `install(FILE)` 一样。对于安装过程中创建的目录，可以使用 `DIRECTORY_PERMISSIONS` 选项来覆盖，除了添加执行权限之外，默认值与文件操作相同。

其余选项允许根据一个或多个通配符模式，或正则表达式，对文件集进行筛选。针对每个文件和目录的完整路径(总是用斜杠指定，即使在Windows上也是如此)测试每个模式或正则表达式。通配符模式必须匹配完整路径的末端，而不仅仅是中间某个部分，而正则表达式可以匹配路径的任何部分，因此更加灵活。如果模式或正则表达式后面跟着 `EXCLUDE` 关键字，将不会安装所有匹配的文件和目录。这是一种只从目录树中排除少数特定内容的方法，也可以通过在模式或 `REGEX` 块之前提供 `FILES_MATCHING` 关键字来实现相反的方法，这意味着将只安装那些确实匹配模式或正则表达式的文件和目录。如果既没有给出 `FILES_MATCHING`，也没有给出 `EXCLUDE`，那么模式或正则表达式的效果就是使用 `PERMISSIONS` 覆盖权限。

例子有助于澄清以上几点。下面的示例改编自CMake文档，安装src目录及其以下的所有头文件，并保留目录结构。

```
install(DIRECTORY src/
DESTINATION include
FILES_MATCHING
PATTERN *.h
)
```

下一个例子复制样本代码和一些脚本，覆盖后者的权限，以确保它们是可执行的：

```
install(DIRECTORY src/
DESTINATION samples
FILES_MATCHING
REGEX "example\\.(h|c|cpp|cxx)"
PATTERN *.txt
PATTERN *.sh
PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_EXECUTE
)
```

安装文档，跳过一些隐藏文件：

```
install(DIRECTORY doc/ todo/ licenses
DESTINATION doc
FILES_MATCHING
REGEX \\.(DS_Store|svn) EXCLUDE
)
```

下一个示例忽略了 `FILES_MATCHING` 或 `EXCLUDE` 选项，这样模式和正则表达式只会修改权限，而不过滤文件和目录列表：

```
install(DIRECTORY admin_scripts
DESTINATION private
PATTERN *.sh
PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
GROUP_READ GROUP_EXECUTE
)
```

所有情况下，`install(DIRECTORY)` 都保留源文件的目录结构。若在安装区域中创建一个空目录，源列表可以为空，但仍创建 `DESTINATION`。

```
install(DIRECTORY DESTINATION somewhere/emptyDir)
```

## 25.5. 自定义安装逻辑

某些情况下，仅仅将内容复制到安装区域是不够的。可能需要在安装过程中执行处理，例如重写文件的某些部分或以编程方式生成内容。对于这些情况，CMake提供了向安装步骤添加自定义逻辑的能力。

```
install(SCRIPT fileName | CODE cmakeCode
[COMPONENT component]
[EXCLUDE_FROM_ALL]
)
```

`CODE` 形式可将CMake命令直接作为单个字符串嵌入，而 `SCRIPT` 形式将使用 `include()` 在安装时读取脚本。请注意，在安装过程中调用自定义代码的位置未指定，但目前的行为是 `install()` 通常在目录范围内处理(但这并不扩展 `install()` 调用内嵌套的子目录)。

多个 `SCRIPT` 和/或 `CODE` 可以组合在一起，将按照指定的顺序执行。`COMPONENT` 和 `EXCLUDE_FROM_ALL` 选项不能重复给出。

```
install(CODE [[ message("Starting custom script") ]]
       SCRIPT myCustomLogic.cmake
       CODE [[ message("Finished custom script") ]])
COMPONENT MyProj_Runtime
)
```

## 25.6. 安装的依赖关系

创建包时，常见的是自包含。这可以扩展到项目构建的构件中，还包括外部依赖项，如编译器的运行时库。**CMake**提供了一些模块，使这个任务更容易。

`InstallRequiredSystemLibraries` 模块旨在为项目提供主要编译器的相关运行时库的详细信息。包括**Intel**(所有主要平台)和**Visual Studio**(仅适用于**Windows**)。使用模块相当简单，项目可以选择让模块定义 `install()`，也可以请求填充相关的变量，以便创建必要的命令。最简单的情况下，建议为 `install()` 设置组件，但项目也可以依赖默认值。

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

默认安装位置是**Windows**的`bin`，和其他平台的`lib`。这可能与大多数项目的典型安装布局相匹配，这里可以用 `CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION` 覆盖：

```
include(GNUInstallDirs)
if(WIN32)
  set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_BINDIR})
else()
  set(CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION ${CMAKE_INSTALL_LIBDIR})
endif()
set(CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT MyProj_Runtime)
include(InstallRequiredSystemLibraries)
```

如果项目希望使用自己定义 `install()`，则需要在包含模块之前将 `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP` 设置为`true`。然后项目可以使用 `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS` 变量访问运行时库列表：

```
set(CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP TRUE)
include(InstallRequiredSystemLibraries)
include(GNUInstallDirs)
if(WIN32)
  install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
  DESTINATION ${CMAKE_INSTALL_BINDIR}
  )
else()
  install(FILES ${CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS}
  DESTINATION ${CMAKE_INSTALL_LIBDIR}
  )
endif()
```

当使用**Intel**编译器时，默认的 `install()` 安装的不仅仅是 `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS` 的内容。还通过文档变量安装了一些没有提供给项目的目录。对于那些有兴趣研究这些附加内容是否合适的开发人员，可以在模块的实现中搜索 `CMAKE_INSTALL_SYSTEM_RUNTIME_DIRECTORIES`，看看附加内容是如何构造的。

使用**Visual Studio**编译器安装其他运行时组件(如**Windows Universal CRT**、**MFC**和**OpenMP**库)时，还可以使用其他一些控件。还可以强制安装运行时库的调试版本。这些在模块文档中都有清楚的描述，有兴趣的读者可以参考那里了解更多的细节。

另一对模块也可以用于安装项目的运行时依赖项。`BundleUtilities` 和 `GetPrerequisites` 模块采用不同的方法，使用特定于平台的工具直接查询已安装的二进制文件，并递归地复制丢失的库。这些模块非常难用，并且不适合处理编译器运行时依赖关系。它们有时可以有效地查找和安装不可预测的依赖项，例如对于复杂的跨平台工具包，如Qt (`DeployQt4`模块广泛地使用这两个模块)。大多数项目最好花精力找出实际的依赖关系，并直接安装，以确保构建过程更可控和可靠，也可以选择使用 `InstallRequiredSystemLibraries` 来处理编译器运行时的依赖关系。

## 25.7. 编写配置包文件

安装的项目要想让其他CMake项目使用它，首选的方法是提供配置包文件。这个文件可以通过 `find_package()` 找到，如在23.5节中介绍的那样。配置文件的名称必须匹配以下两种形式之一：

- `<packageName>Config.cmake`
- `<lowercasePackageName>-config.cmake`

第一种形式可能更常见一些，该文件将为安装的项目提供的库和可执行文件提供导入目标。如果安装的基点添加到 `CMAKE_PREFIX_PATH` 中，配置文件安装到的目录应该是 `find_package()` 搜索的默认位置，这样可以很容易找到配置文件。第23.5节中，将搜索的位置为：

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

在Apple平台上，还可以搜索以下目录：

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

显然，这是一组很大的备选项，但最佳选择在某种程度上取决于项目如何安装。在打包到Linux发行版中时，发行版本本身可能有关于这些文件应该放在哪里的策略。与其强迫发行版向项目提供自己的补丁，以确保配置文件根据其策略安装，不如提供一种将所需的详细信息传递到构建的方法。缓存变量是个理想的选择，因为项目可以指定默认值，可以在不更改项目的情况下重写。在没有任何其他约束的情况下，两个非常简单且常用的位置是 `<prefix>/cmake` 和 `<prefix>/lib/cmake/<packageName>`，后者对多体系结构的部署更友好一些(参见下面的示例)。

`install(EXPORT_ANDROID_MK)` 会为项目提供一个Android.mk文件，CMake对其位置没有特定约定。可以合理安排包布局中专用的ndk-build目录，但这最终要取决于项目。

### 25.7.1. CMake项目的配置文件

对于简单的CMake项目，使用单一的导出集，没有依赖，`install(EXPORT)` 可以直接创建基本的配置文件：

```

include(GNUInstallDirs)
install(EXPORT myProj
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
NAMESPACE MyProj::
FILE MyProjConfig.cmake
)

```

这里使用了 `GNUInstallDirs` 模块提供的 `CMAKE_INSTALL_LIBDIR` 缓存变量，来增加Linux发行版不需要进行任何更改的可能性。`GNUInstallDirs` 模块已经解决了常见的情况，通过定义缓存变量，允许在需要时进行简单的定制。

实践中，配置文件通常不是这样生成。更常见的情况是，准备单独的配置文件，通过 `include()` 引入导出文件。一个扩展示例使用两个导出集演示了该方案：

#### *MyProjConfig.cmake*

```

include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake")

```

#### *CMakeLists.txt*

```

# Define targets, etc...

# Create two separate export sets installed to the same place
# and a manually written config file that will include them
include(GNUInstallDirs)
install(EXPORT MyProj_Runtime
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
NAMESPACE MyProj::
FILE MyProj_Runtime.cmake
COMPONENT MyProj_Runtime
)
install(EXPORT MyProj_Development
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
NAMESPACE MyProj::
FILE MyProj_Development.cmake
COMPONENT MyProj_Development
)
install(FILES MyProjConfig.cmake
DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
)

```

`MyProjConfig.cmake`非常简单，不需要外部提供依赖关系，并且配置文件假设运行时和开发组件都是安装的。然后考虑一个场景，其中运行时组件依赖于其他名为`BagOfBeans`的包。配置文件负责确保`BagOfBeans`中所需的目标可用，通常通过 `find_package()` 来实现。方便起见，有时候可以使用 `CMakeFindDependencyMacro` 模块中的 `find_dependency()` 作为 `find_package()` 的包装器，处理 `QUIET` 和 `REQUIRED` 关键字。`find_dependency()` 还有一个附加行为，如果没有找到所请求的包，配置文件的处理将立即停止，控制权返回给调用者。这就好像是在 `find_dependency()` 失败后立即进行了 `return()`。简单、干净的依赖项规范，以及对依赖项失败的优雅处理。

#### *MyProjConfig.cmake*

```

include(CMakeFindDependencyMacro)
find_dependency(BagOfBeans)
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Runtime.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/MyProj_Development.cmake")

```

项目作者应该知道，`find_dependency()` 包含一个优化，如果检测到请求的包已经在前面找到，就会跳过调用。除非以后请求不同的包组件，否则这种方法可以正常工作。`find_dependency()` 第一次成功时，会锁定找到的组件集。如果稍后调用`find_dependency()` 传递不同的组件，则忽略。因此，如果依赖项支持包组件，项目应该直接调用`find_package()`，并自己处理`QUIET` 和`REQUIRED` 选项。这些选项作为变量`_${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIET` 和`_${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED` 传递到配置文件。因为可能存在大小写差异，所以始终使用`_${CMAKE_FIND_PACKAGE_NAME}` 而不是硬编码包名。

```
unset(extraArgs)
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_QUIETLY)
    list(APPEND extraArgs QUIET)
endif()
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED)
    list(APPEND extraArgs REQUIRED)
endif()
find_package(BagOfBeans COMPONENTS Foo Bar ${extraArgs})
```

如果项目的组件是可选的，那么配置文件的复杂性就会增加。支持这种功能所涉及的步骤归纳如下：

- 构建需要找到的项目组件集。从`find_package()` 中的必需和可选组件集开始，添加满足项目依赖关系的组件。
- 计算出该项目组件集所需的外部依赖集。有些是强制的，有些是可选的，因此需要派生两个独立的外部依赖集。
- 查找外部依赖项，如果任何必需的依赖项未能加载，项目查找操作也必须失败，控制应该立即返回适当的错误消息。缺少可选的外部依赖项不会出现失败或错误消息。
- 更新项目组件集，以删除依赖于缺少可选外部依赖项的组件。如果删除的组件本身是其他组件的依赖项，则可能需要进一步的筛选项目组件集。
- 加载剩余的项目组件。

如果没有指定组件，项目需要决定要做什么。这可以视为所有组件都指定为可选组件或是必需组件。另一种策略是加载基本组件的最小集合，并忽略其他组件。最合适的策略将取决于项目组成部分的性质。请求组件集将在`_${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS` 中可用，如果组件指定为必需的，而不是可选的，那么`_${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_<comp>` 对该组件的值为`true`。

配置文件不应该使用`message()` 报告错误，而是应该将错误消息存储在名为`_${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE` 的变量中。然后，`find_package()` 将获取该错误，并将其包装为在项目何生错误处的详细信息，`_${CMAKE_FIND_PACKAGE_NAME}_FOUND` 设置为`false` 以表示失败。这允许`find_package()` 正确地实现不使用`REQUIRED` 的调用。如果包配置文件使用了`message(FATAL_ERROR...)`，那么调用者永远不能将包视为可选。

```

# Work out the set of components to load
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS)
set(comps ${CMAKE_FIND_PACKAGE_NAME}_FIND_COMPONENTS)
# Ensure Runtime is included if Development was specified
if(Development IN_LIST comps AND NOT Runtime IN_LIST comps)
list(APPEND comps Runtime)
endif()
else()
# No components given, look for all components
set(comps Runtime Development)
endif()

# Find external dependencies, storing comps in a safer variable name.
# In this example, BagOfBeans is only needed by the Development component.
set(${CMAKE_FIND_PACKAGE_NAME}_comps ${comps})
if(Development IN_LIST ${comps})
find_dependency(BagOfBeans)
endif()

# Check all required components are available before trying to load any
foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
if(${CMAKE_FIND_PACKAGE_NAME}_FIND_REQUIRED_${comp} AND
NOT EXISTS ${CMAKE_CURRENT_LIST_DIR}/MyProj${comp}.cmake)
set(${CMAKE_FIND_PACKAGE_NAME}_NOT_FOUND_MESSAGE
"MyProj missing required dependency: ${comp}")
set(${CMAKE_FIND_PACKAGE_NAME}_FOUND FALSE)
return()
endif()
endforeach()

foreach(comp IN LISTS ${CMAKE_FIND_PACKAGE_NAME}_comps)
# All required components are known to exist. The OPTIONAL keyword
# allows the non-required components to be missing without error.
include(${CMAKE_CURRENT_LIST_DIR}/MyProj${comp}.cmake OPTIONAL)
endforeach()

```

上面的示例演示了，检查是否能够满足所需的组件之前不创建任何导入目标。

与配置文件密切相关的是版本文件。如果提供了版本文件，名字应该符合两种形式之一，并且必须和配置文件在同一目录下：

- <packageName>ConfigVersion.cmake
- <lowercasePackageName>-config-version.cmake

版本文件名的形式通常遵循与其相关配置文件相同的形式(例如，`FooConfigVersion.cmake`会和`FooConfig.cmake`搭配，而`foo-config-version.cmake`通常与`foo-config.cmake`配对)。版本文件是通知`find_package()`包是否满足指定的版本要求。`find_package()`在加载版本文件前会设置一些变量：

- PACKAGE\_FIND\_NAME
- PACKAGE\_FIND\_VERSION
- PACKAGE\_FIND\_VERSION\_MAJOR
- PACKAGE\_FIND\_VERSION\_MINOR
- PACKAGE\_FIND\_VERSION\_PATCH
- PACKAGE\_FIND\_VERSION\_TWEAK
- PACKAGE\_FIND\_VERSION\_COUNT

这些变量包含`find_package()`的`VERSION`版本信息。如果没有给出这样的参数，那么`PACKAGE_FIND_VERSION`将为空，而其他`PACKAGE_FIND_VERSION_*`变量将为0。`PACKAGE_FIND_VERSION_COUNT`对版本组件进行计数，其余变量有明显的含义。版本文件需要检查请求的信息与实际版本的包，然后设置以下变量：

### *PACKAGE\_VERSION*

这是实际的包版本，通常是*major.minor.patch.tweak*(不是所有的组件都是必需的)。

### *PACKAGE\_VERSION\_EXACT*

只有在包版本和请求版本完全匹配时才设置为*true*。

### *PACKAGE\_VERSION\_COMPATIBLE*

只有在包版本与请求版本兼容时才设置为*true*。如何决定兼容性取决于包本身。对于20.3节中提到的语义版本化原则的项目，变量将按照以下规则设置：

- 如果缺少任何版本组件，将其视为0。
- 如果主版本号不同，则结果为*false*。
- 如果主版本号相同，次版本号小于所需的版本，则结果为*false*。
- 如果主版本号和次版本号相同，包的补丁版本号小于所需的版本，则结果为*false*。
- 如果主版本号、次版本号和补丁版本号相同，软件包的微调版本号小于所需的版本，则结果为*false*。
- 对于所有其他情况，结果为*true*。

### *PACKAGE\_VERSION\_UNSUITABLE*

只有在版本文件需要表明包不能满足任何版本要求时，才设置为*true*(基本上包没有版本号，所以任何版本要求都应该视为失败)。

`find_package()` 将使用此信息，将以下变量返回给调用者(这里的返回值将实际的包的版本，而不是版本需求传递到 `find_package()` 中)：

- `<packageName>_VERSION`
- `<packageName>_VERSION_MAJOR`
- `<packageName>_VERSION_MINOR`
- `<packageName>_VERSION_PATCH`
- `<packageName>_VERSION_TWEAK`
- `<packageName>_VERSION_COUNT`

虽然项目可以自由创建版本文件，但更简单、更健壮的方法是使用 `CMakePackageConfigHelpers` 模块提供的 `write_basic_package_version_file()` 命令：

```
write_basic_package_version_file(outFile
[VERSION requiredVersion]
COMPATIBILITY compat
)
```

如果给定了 `VERSION`，那么 `requiredVersion` 应该是 `major.minor.patch.tweak` 形式，但只有主部分是强制性的。如果没有给出 `VERSION` 选项，则使用 `PROJECT_VERSION` 变量(由 `project()` 设置)。兼容性选项指定了如何确定兼容性的策略。`compat` 参数必须是下列值之一(注意，大多数名称有一点误导)：

#### *AnyNewerVersion*

包版本必须等于或大于指定的版本。

#### *SameMajorVersion*

包版本必须等于或大于指定的版本，并且包版本号的主部分必须与 `requiredVersion` 中的相同。这与语义版本化的兼容性要求相同。

#### *SameMinorVersion*

包版本必须等于或大于指定的版本，并且包版本号的主要部分和次要部分必须与 `requiredVersion` 中的相同。只有 CMake 3.11 或更高版本才支持这种选择。

#### *ExactVersion*

软件包版本号的主要、次要和补丁部分必须与 `requiredVersion` 中的相同，忽略调整部分。这一策略尤其具有误导性，目前正在讨论，会将其废弃，以支持更清晰的策略。

`CMakePackageConfigHelpers` 模块还提供了 `configure_package_config_file()`，目的是为了给路径处理提供一些便利，使项目更容易定义可重定位的包。大多数项目通常不需要它，但是当包配置文件需要相对于基本安装位置，而不是配置文件本身的位置来引用安装文件时，该模块提供了一种更简单的方法。该命令的形式如下：

```
configure_package_config_file(inputFile outputFile
  INSTALL_DESTINATION path
  [INSTALL_PREFIX prefix]
  [PATH_VARS var1 [var2...] ]
  [NO_SET_AND_CHECK_MACRO]
  [NO_CHECK_REQUIRED_COMPONENTS_MACRO]
)
```

应该使用该命令替换 `configure_file()`，以复制 `<Project>Config.cmake.in`，并将其替换。它将用做转换为绝对路径 `<somenvar>` 的内容，替换 `@PACKAGE_<somenvar>@` 中的变量。原始内容视为相对于基本安装位置的内容，需要用 `PATH_VARS` 选项。以这种方式转换的每个变量，要使此功能发挥作用，要在替换的变量之前使用，输入文件的顶部或附近必须有 `@PACKAGE_INIT@`。

相对于 `INSTALL_PREFIX`，`INSTALL_DESTINATION` 是 `outputFile` 将安装到的目录。当省略 `INSTALL_PREFIX` 时，默认为 `CMAKE_INSTALL_PREFIX`。`INSTALL_PREFIX` 通常只在 `outputFile` 用于构建树而不是安装的情况下提供(例如，它会与 `export(EXPORT)` 一起使用)。

`NO_SET_AND_CHECK_MACRO` 和 `NO_CHECK_REQUIRED_COMPONENTS_MACRO` 选项阻止 `@PACKAGE_INIT@` 定义辅助函数。在导入目标成为提供包目标的首选方式之前，需要使用变量。为了实现这点，`configure_package_config_file()` 提供了一个 `set_and_check()`，只会在尚未定义变量的情况下设置变量。提供导入目标的项目不需要这个宏，可以添加 `NO_SET_AND_CHECK_MACRO` 来防止定义。同样，当所有信息都通过变量提供时，在返回之前检查是否在末尾设置了必需的变量。为此定义了名为 `check_required_components()` 的宏，但是提供导入目标的项目本身应该执行这些检查，并且只有在找到所有组件时才定义导入目标，这会使得 `check_required_components()` 宏变得多余。

下面的例子有助于阐明这个命令的用法：

#### *CMakeLists.txt*

```
include(GNUInstallDirs)
include(CMakePackageConfigHelpers)
set(cmakeModulesDir cmake)
configure_package_config_file(MyProjConfig.cmake.in MyProjConfig.cmake
  INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
  PATH_VARS cmakeModulesDir
  NO_SET_AND_CHECK_MACRO
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
)
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/MyProjConfig.cmake
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MyProj
  COMPONENT ...
)
```

#### *MyProjConfig.cmake.in*

```

@PACKAGE_INIT@

list(APPEND CMAKE_MODULE_PATH "@PACKAGE_cmakeModulesDir@")

# Include the project's export files, etc...

```

## 25.7.2. 非cmake项目的配置文件

配置文件机制并不仅限于CMake构建项目，也可以用于非CMake项目(尽管这还不是很常见)。虽然CMake项目可以利用各种CMake功能创建所需的文件，非CMake项目必须手动定义它们。对于这样的项目，保持文件的简单性也很重要，因为它们很可能由不太熟悉CMake的人维护。第一步是放弃对组件的支持，只将包作为导入目标的简单集合提供。对于只需要提供库的项目，下面的例子展示了一个小的配置文件，可以作为一个起点：

```

# Compute the base point of the install by getting the directory of this
# file and moving up the required number of directories
set(_IMPORT_PREFIX "${CMAKE_CURRENT_LIST_DIR}")
foreach(i RANGE 1 NumSubdirLevels) ①
    get_filename_component(_IMPORT_PREFIX "${_IMPORT_PREFIX}" PATH)
    if(_IMPORT_PREFIX STREQUAL "/")
        set(_IMPORT_PREFIX "")
        break()
    endif()
endforeach()

# Use a prefix specific to this project
set(projPrefix MyProj)

# Example of defining a static library imported target
add_library(${projPrefix}::myStatic STATIC IMPORTED)
set_target_properties(${projPrefix}::myStatic PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyStatic.a" ②
)

# Example of defining a shared library imported target with version details
add_library(${projPrefix}::myShared SHARED IMPORTED)
set_target_properties(${projPrefix}::myShared PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyShared.so.1.6.3" ③
    IMPORTED SONAME "libmyShared.so.1" ④
)

# Another example of a shared library, this time for Windows
add_library(${projPrefix}::myDLL SHARED IMPORTED)
set_target_properties(${projPrefix}::myDLL PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/bin/myShared.dll"
    IMPORTED_IMPLIB "${_IMPORT_PREFIX}/lib/myShared.lib" ⑤
)

```

① `NumSubdirLevels` 是该配置文件在基本安装点的子目录级别数。例如，如果文件在 `lib/cmake/Foo/FooConfig.cmake` 中找到，那么 `NumSubdirLevels` 是3。

②指定库相对于基本安装点的路径，在安装点之前找到，并存储在 `_IMPORT_PREFIX` 中。

③这个示例展示了Linux等平台，动态库版本号如何放在文件名的末尾。

④对于支持 `sonames` 的平台，`IMPORTED SONAME` 实际上是将链接嵌入到这个目标的二进制文件中的名称。Apple 平台上，通常会有包含 `@rpath` 和一些子目录的组件。

⑤还必须提供DLL导入库的位置。如果只是提供DLL(例如，它在运行时可用，但不用于直接链接)，`IMPORTED_IMPLIB`可以省略，但这种情况不太常见

上面是基本内容，各种`IMPORTED_...`属性需要为每个平台进行定制，但是非CMake项目可以自由地使用它认为方便的机制来生成安装配置文件。为了增加健壮性，每个导入库只有在不存在的情况下才会添加，如下所示：

```
if(NOT TARGET ${projPrefix}::myStatic)
add_library(${projPrefix}::myStatic STATIC IMPORTED)
set_target_properties(${projPrefix}::myStatic PROPERTIES
    IMPORTED_LOCATION "${_IMPORT_PREFIX}/lib/libmyStatic.a"
)
endif()
```

## 25.8. 总结

安装是一个重要的主题，需要良好的计划和对预期部署平台的理解。对于项目来说，最初只关注一个平台，或者只关注最终打算使用的一组平台的子集是很常见的，但是任何安装和部署计划会导致在项目发布周期后期处理意外的复杂性和平台差异。项目应该清楚地了解所安装的文件和目录结构，以及最终将支持的一整套打包方案。这在很大程度上影响了项目结构，包括如何在库之间划分功能，以及二进制文件中需要显示哪些符号等基本内容。

项目应该尽可能遵循标准的布局。使用像`GNUInstallDirs`这样的模块可以极大地简化这个任务，即使对于Windows上的包也是如此。如果不可能，或者不希望使用，项目仍然可以考虑相同的目录结构是否可以在不同的平台上使用，以简化应用程序开发。

强烈鼓励项目使其包可重定位。除非软件包需要安装到非常特定的位置，可重定位软件包有显著的优势。为最终用户提供了更大的灵活性，更容易支持广泛的打包系统，并且在开发期间更容易测试。

默认安装点的选择特定于平台，CMake提供默认设置并不总是理想的，但是包的创建通常会覆盖它们。避免在安装路径中包含包版本号，特别是对于可重定位的包。由于不同的使用场景会调用不同的目录结构，而这些目录结构可能与特定于版本的路径不兼容，因此倾向于让用户自行决定安装。项目也应该倾向于遵循适当的标准，比如Linux的文件系统层次标准(通常也适用于大多数其他基于Unix的平台，Apple除外)。

当定义目标有使用需求时，使用`$<BUILD_INTERFACE:>`生成器表达式来正确地表示构建使用的头文件搜索路径。对于任何安装的库目标，对于`install(TARGETS)`最好使用`INCLUDES DESTINATION`来设置头文件搜索路径，而不是在目标本身上使用`$<INSTALL_INTERFACE:>`生成器表达式，并确保`INCLUDES DESTINATION`使用相对于安装点的相对路径。

```
add_library(foo ...)

# Not ideal: embeds build paths in installed export files
target_include_directories(foo PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# Better: separate paths for build and install, with the latter
# added as part of the install() command rather than with the target
include(GNUInstallDirs)
target_include_directories(foo PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
)
install(TARGETS foo ...
    INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

始终为已安装的目标分配组件，并使用特定于项目的组件名称。当项目作为大型项目的一部分使用时，允许父项目控制子组件的处理方式。一个好模式的例子是 `<ProjectName>_<componentname>`，例如 `MyProj_Runtime`。安装导出集时，使用相同的项目名称作为命名空间，并附加两个冒号(即 `MyProj::`)。遵循这些命名约定将使已安装的项目更加直观，还可以避免与其他项目包名称的冲突。

如果项目提供了其他项目需要链接的库，最好为运行时支持和开发单独定义的组件。这允许父项目重用运行时组件来打包动态库和执行所需的内容，并避免打包仅用于开发的目标，如静态库、头文件等。还减少了包维护人员(例如Linux发行版)的工作，因为包经常分解为运行时包和开发包。

包配置文件中，始终没有创建导入的目标，除非 `find_package()` 成功。这意味着在创建导入目标之前，所有必需组件都要可用，所有必需的目标依赖项都应该存在。要引入依赖项，使用 `CMakeFindDependencyMacro` 模块中的 `find_dependency()`，而不是从包配置文件中调用 `find_package()`，除非依赖项支持包组件。如果调用 `find_package()` 引入依赖项，请确保将 `QUIET` 和 `REQUIRED` 选项正确地传递到依赖项的 `find_package()` 中。还可以使用适当的变量来定义成功/失败，并将错误消息报告给原始的 `find_package()`，而不是调用 `message(FATAL_ERROR...)` 或类似的命令。

请使用 `InstallRequiredSystemLibraries` 模块来处理编译器运行时依赖项的安装，这使得项目避免了为不同的 Visual Studio版本、SDK、工具集选择等寻找合适的文件，而重复所有复杂的逻辑。如果对Intel编译器的支持很重要，请理解此模块默认情况下安装的各种库，并决定是否需要这些库。特别是使用OpenMP的项目，很可能希望使用默认的安装命令，而不是定义自己的安装命令，这样就不必手动定义所需的库。

# 第26章： 打包

创建发布包是开发人员经常感到无力的部分。对于希望掌握跨平台创建健壮的、表现良好的包来说，各种打包系统、平台差异和约定都有一条非常陡峭的学习曲线。每个包管理系统总是使用自己独特的输入规范，对包含的内容、应该如何安装、包组件之间的关联、如何与操作系统集成等。平台之间，甚至同一平台的不同发行版之间的差异并不明显，通常只有遇到不可预见的行为或约束时才会了解(Windows路径长度限制和Linux上不同的系统库目录的名称就是很好的例子)。

尽管有这些差异，在使用的包方面，也有共同点。虽然每个系统或平台可能有不同的实现，但许多打包功能可以以通用的方式描述。**CMake**和**CPack**利用了这一点，并提供了定义良好的接口来指定这些共通方面，然后将其转换为系统输入文件或命令，以生成各种格式的包。这为开发人员提供了更短的学习曲线，从而为跨相关平台生成包提供了捷径。

**CPack**不仅抽象了打包的共同点，还简化了许多打包器的特性。通过这些简单的接口，**CMake**和**CPack**使开发人员能够以更熟悉的方式开发更高级的打包特性。大多数情况下，这是通过设置几个相关的变量，或使用适当的参数调用函数来实现的，所有这些参数都在每个包程序的**CMake**模块文档中定义。

**CPack**打包在内部实现为一个或多个暂存区域的安装，然后使用该区域生成最终的包。这些安装是通过 `install()` 来控制的，上一章介绍了这个过程的第一部分，本章会介绍了这个过程的第二部分，描述了指定包数据和包本身配置的变量和命令。

## 26.1. 打包的基础知识

打包设置和执行与测试的处理方式类似。**CPack**读取一个输入文件，并根据该文件的内容生成相应的包。如果命令行上没有显式地提供输入文件，**CPack**将使用当前目录中的 `CPackConfig.cmake`。这个输入文件通常由**CMake**通过**CPack**模块生成，就像包含**CTest**模块为**CTest**生成输入文件一样。项目可以通过**CMake**变量和命令自定义打包输入文件的内容。

**CPack**模块支持基于目标平台的默认格式，要创建的格式集可以在**CPack**命令行上使用 `-G` 覆盖。如果需要建立多种格式时，可以使用分号分隔：

```
cpack -G "ZIP;RPM"
```

如果**CMake**配置为使用**Xcode**或**Visual Studio**这样的多配置生成器，**CPack**需要知道应该打包哪个配置的可执行文件。可以通过给**CPack**一个 `-c` 选项来完成的(单配置**CMake**生成器会忽略 `-c` 选项)：

```
cpack -C Release
```

**CPack**命令还支持其他选项，`-G` 和 `-c` 是两个常用的选项，其他信息通过输入文件提供。这是因为开发人员可以构建包目标，而不是直接调用**CPack**，该目标将首先构建默认的**all**目标，然后使用最少的选项调用**CPack**。因此，对于项目来说，确保**CPack**输入文件定义了必需的设置，会更加方便一些。如果构建树的顶层包含名为 `CPackConfig.cmake` 的文件，**CMake**将自动创建包目标。

创建**CPack**输入文件的最简单方法是包含**CPack**模块，对于整个**CMake**项目只执行一次。这种包含通常在顶层**CMakeLists.txt**文件的末尾附近，直接或通过子目录的**CMakeLists.txt**完成。以项目是否有父项目作为包含条件，还可以确保项目仅在本身为顶层项目时才会打包。例如：

```

cmake_minimum_required(VERSION 3.0)
project(MyProj)
# ...Define targets, add subdirectories, etc...

# End of the CMakeLists.txt file
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
  add_subdirectory(packaging) # include(CPack) will happen in here
endif()

```

包含CPack模块时，`CPackConfig.cmake`文件保存到构建树的顶层(即`CMAKE_BINARY_DIR`)。由于CMake只在处理完CMakeLists.txt文件后才检查这个文件。如果包含CPack模块，便会创建包的构建目标。

虽然为打包配置的大多数提供了默认值，但这些默认值并不总合适。大多数项目希望在包含CPack模块之前设置一些基本信息，以提供更好的替代方案。特别是建议在`include(CPack)`之前设置以下变量：

#### `CPACK_PACKAGE_NAME`

包名是数据中基本的部分之一。用作包的默认文件名的一部分，可能出现在UI安装程序的不同位置，并且很可能是用户用来引用项目的名称。理想情况下，不会包含空格，因为在某些上下文中，空格会替换成其他字符。如果没有显式设置该变量，则使用`CMAKE_PROJECT_NAME`作为默认值。

#### `CPACK_PACKAGE_DESCRIPTION_SUMMARY`

这个变量提供了关于项目的简短描述，适合在空间受限的包列表中显示，并且让终端用户了解该包的用处。也在其他情况下显示给用户，仅用于显示信息。CMake 3.9中，默认值取自`CMAKE_PROJECT_DESCRIPTION`，而在早期的CMake版本中，默认值是空字符串。

#### `CPACK_PACKAGE_VENDOR`

供应商通常只用于获取信息，而不会影响包的结构或行为。默认的Humanity通常不适合做任何事情，除非正确设置。这里可以使用真实的公司或组织名称，而不是域名。

#### `CPACK_PACKAGE_VERSION_MAJOR`, `CPACK_PACKAGE_VERSION_MINOR`, `CPACK_PACKAGE_VERSION_PATCH`

用于构造整个包版本，可作为包文件名的一部分出现在包的元数据和安装程序UI中。版本信息是打包的关键部分，项目应该始终显式地设置。默认值为0,1，1只作为占位符，不应该用于正式的发布包。一种方便的使用模式是，给`project()`提供相关的版本信息：

```

set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})

```

CPack根据这三个变量填充`CPACK_PACKAGE_VERSION`，但这只会在CPack运行时发生，因此CMake处理期间不会填充`CPACK_PACKAGE_VERSION`。CMake 3.12中，这些变量的默认值来

自`CMAKE_PROJECT_VERSION_MAJOR`、`CMAKE_PROJECT_VERSION_MINOR`和`CMAKE_PROJECT_VERSION_PATCH`变量，这些变量在CMake 3.12中添加。这些变量是通过顶层CMakeLists.txt文件中`project()`的版本信息设置，因此可以提供合理的默认值，而不是随意的0、1和1。当项目为顶层项目，可以依赖这些变量来提供默认值，但实际情况可能并不总是这样。因此，始终显式地将设置为项目真正想要值则更为安全。

#### `CPACK_PACKAGE_INSTALL_DIRECTORY`

一些包程序会将其附加到基本安装中，以创建特定于包的目录。其默认值可以不同，可能包括包名和版本。默认值中不太合适出现版本号，例如对于能够升级项目的安装程序。为了确保更好的默认行为，项目可能将其设置为与`CPACK_PACKAGE_NAME`相同的值。

### ***CPACK\_VERBATIM\_VARIABLES***

应该始终显式地将该变量设置为**true**。确保写入CPack配置文件的内容正确地转义。默认值为**false**只是为了保持与早期CMake版本的兼容，但是旧的行为会导致配置文件的格式不正确，因此不推荐使用。

通常会设置更多的变量来改善用户体验，特别是对于UI安装。

### ***CPACK\_PACKAGE\_DESCRIPTION\_FILE***

这是一个文本文件的名称，其中包含对项目较长的描述。文件的内容可能会在安装程序的介绍性页面中显示，或者添加到包的元数据中。始终使用文件的绝对路径。作为另一种选择，描述可以直接作为名为 `CPACK_PACKAGE_DESCRIPTION` 变量的内容提供。虽然在CMake 3.11或更早的版本中没有文档说明这一点，但在CMake的早期版本中就提供了支持。

### ***CPACK\_RESOURCE\_FILE\_WELCOME***

安装程序在开始页面上会显示欢迎信息。此变量指定一个文件名，指定在欢迎页面上应该显示的内容。如果没有设置，那么对于那些显示欢迎消息的安装程序，CPack会提供默认值作为占位符，这是一个相对较差的方案，不适合正式发布版本。如果发布有欢迎页面的安装程序，项目应该需要设置这个选项。并且始终使用文件的绝对路径。

### ***CPACK\_RESOURCE\_FILE\_LICENSE***

大多数UI安装程序都会向用户显示许可页面，并要求用户在继续安装前接受许可。许可显示的文本取自这个变量命名的文件。如果没有设置变量，则使用一些通用的占位符文本，但是强烈建议项目提供自己的许可信息。始终使用文件的绝对路径。

### ***CPACK\_RESOURCE\_FILE\_README***

一些UI安装程序会提供单独的页面，显示由该变量命名的文件的内容。可以在用户继续安装前向他们提供一些信息，并且默认情况下会使用通用的文本。如果想要创建显示这些页面的安装程序，项目应该通过这个变量提供一个包含更合适内容的文件。始终使用文件的绝对路径。

### ***CPACK\_PACKAGE\_ICON***

通常可以设置这个变量，但是请注意，大多数包生成器对于包内关联位置的图标的格式和使用都有自己的不同要求。有些生成器会忽略这个变量，而有些则不会。

遵循上述准则的示例：

```
set(CPACK_PACKAGE_NAME MyProj)
set(CPACK_PACKAGE_VENDOR MyCompany)
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "CPack example project")
set(CPACK_PACKAGE_INSTALL_DIRECTORY ${CPACK_PACKAGE_NAME})
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
set(CPACK_VERBATIM_VARIABLES YES)
set(CPACK_PACKAGE_DESCRIPTION_FILE ${CMAKE_CURRENT_LIST_DIR}/Description.txt)
set(CPACK_RESOURCE_FILE_WELCOME ${CMAKE_CURRENT_LIST_DIR}/Welcome.txt)
set(CPACK_RESOURCE_FILE_LICENSE ${CMAKE_CURRENT_LIST_DIR}/License.txt)
set(CPACK_RESOURCE_FILE_README ${CMAKE_CURRENT_LIST_DIR}/Readme.txt)
include(CPack)
```

为了方便在不带参数的情况下运行CPack，并使用包构建目标，应该将 `CPACK_GENERATOR` 设置为所需的格式。如果没有设置，将使用默认生成器支持的格式。由于并不是所有平台都支持或适合所有格式，因此此变量需要指定格式。下面的示例为目标平台选择了通用的存档格式和原生包的格式：

```
if(WIN32)
    set(CPACK_GENERATOR ZIP WIX)
elseif(APPLE)
    set(CPACK_GENERATOR TGZ productbuild)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
    set(CPACK_GENERATOR TGZ RPM)
else()
    set(CPACK_GENERATOR TGZ)
endif()
```

CPack模块还定义了生成源码包的信息。可以使用 `CPackConfig.cmake` 代替创建的 `CPackSourceConfig.cmake`，当项目配置为使用Makefile或Ninja生成器时，还会定义 `package_source` 构建目标。生成的源码包相对简单，下面两个命令都可以实现相同的效果。

```
# All build generators
cpack -G TGZ --config CPackSourceConfig.cmake

# Makefile and Ninja build generators only
cmake --build . --target package_source
```

源码包包含整个源目录树。可以使用 `CPACK_SOURCE_IGNORE_FILES` 过滤掉某些部分，其中包含正则表达式列表，用于比较完整的文件路径，所有匹配的文件将从源包中删除。这个变量默认忽略`.git`、`.svn`等存储库目录，以及常见的临时文件。如果项目覆盖了 `CPACK_SOURCE_IGNORE_FILES`，需要指定相关的模式。为了避免正则表达式中的转义和引号问题，强烈建议将 `CPACK_VERTIM_VARIABLES` 设置为true。

```
set(CPACK_VERTIM_VARIABLES YES)
set(CPACK_SOURCE_IGNORE_FILES
    \\.git/
    \\.swp
    \\.orig
    /CMakeLists\\.txt\\user
    /privateDir/
)
```

## 26.2. 组件

如果项目的 `install()` 中没有定义任何组件，那么所有包生成器都将生成一个包含所有已安装内容的包。当项目定义了组件时，就为如何打包提供了更大的灵活性。还可以在组件之间指定关系，从而允许定义分层组件结构，并在安装时强制执行组件之间的依赖关系。每个包生成器都以不同的方式使用这些组件的信息，一些生成器为不同的组件创建包，而另一些生成器则在单个UI安装程序中显示用户可选择的组件。一些安装程序甚至支持在安装时按需下载组件。

上一章演示了如何将组件定义为 `install()` 的一部分。这些命令只将内容分配给组件，而不定义任何组件信息。组件之间的关系是使用 `CPackComponent` 模块指定的，`CPackComponent` 模块作为包含CPack模块的一部分会自动包含。这些命令还为组件提供额外的数据，一些安装程序在安装期间使用这些组件向用户显示信息。

CPackComponent 模块中最重要的命令是 `cpack_add_component()`，它描述了单个组件：

```
cpack_add_component(componentName
[DISPLAY_NAME name]
[DESCRIPTION description]
[DEPENDS comp1 [comp2...] ]
[GROUP group]
[REQUIRED | DISABLED]
[HIDDEN]
[INSTALL_TYPES type1 [type2...] ]
[DOWNLOADED]
[ARCHIVE_FILE archiveFileName]
[PLIST plistFileName]
)
```

虽然所有的关键字都是可选的，但至少要提供 `DISPLAY_NAME` 和 `DESCRIPTION`，以便在安装期间向用户提供有意义的信息，并使非UI安装程序有足够的数据供用户理解包的用途。如果只在安装一个或多个组件时才安装该组件，应该将这些组件与 `DEPENDS` 一起列出。注意，并不是所有包类型都满足这些依赖关系。可以使用 `GROUP` 将组件放在特定的组下，可以使用 `cpack_add_component_group()` 进行描述(后面将进一步讨论)。

如果是必安组件，则应该给出 `REQUIRED` 关键字。用户将无法通过安装程序的UI禁用该组件。如果没有这个关键字，用户可以启用或禁用组件，默认状态是启用的。若要将此默认值更改为禁用，需要添加 `DISABLED` 关键字。无论组件是否必需，可以通过添加 `HIDDEN` 关键字对安装程序UI进行隐藏。非必需的隐藏组件通常会禁用，如果有另一个启用的组件对其有依赖，安装程序将安装该组件。

其余选项具有更专业的效果，只适用于少数包生成器。安装类型是组件的预置选择，可用于简化用户的安装过程。组件可以通过 `INSTALL_TYPES` 选项分配任意数量的安装类型，其中每个类型都是由 `cpack_add_install_type()` 定义，如下所示：

```
cpack_add_install_type(typeName [DISPLAY_NAME uiName])
```

如果 `typeName` 已经具有足够的描述性，可以省略 `DISPLAY_NAME` 选项，但对于使用多个显示的安装类型，必须使用 `DISPLAY_NAME`，而 `uiName` 将是带引号的字符串。没有预定义的安装类型，但可以看到包提供了 `Full`、`Minimal` 或 `Default` 这样的安装类型。在 CMake 提供的维护的包生成器中，只有 NSIS 支持安装类型。

对于那些支持可下载组件的生成器，将 `DOWNLOADED` 关键字添加到 `cpack_add_component()` 可以按需下载组件，而不是直接包含在包中。`ARCHIVE_FILE` 选项可用于定制下载组件的文件名。CMake 提供的下载组件生成器是 IFW，所以关于这个特性的讨论会在 26.4.2 中进行。类似地，`PLIST` 选项(仅在 CMake 3.9 或更高版本中可用)只能用于 `productbuild` 包生成器(参见 26.4.6 节)。

如果没有用 `GROUP` 定义组件的详细信息，那么在大多数 UI 安装程序中，组件会作为一个简单的列表。当使用分组时，允许定义任意深度的层次，可以包含组件和其他组。使用 CPackComponent 模块中的以下命令，可以定义一个组：

```
cpack_add_component_group(groupName
[DISPLAY_NAME name]
[DESCRIPTION description]
[PARENT_GROUP parent]
[EXPANDED]
[BOLD_TITLE]
)
```

这个命令可以在 `groupName` 的 `cpack_add_component()` 之前或之后使用。`DISPLAY_NAME` 和 `DESCRIPTION` 选项的作用与 `cpack_add_component()` 中的对应选项相同。`PARENT_GROUP` 是组的等价选项，允许将其放在另一个组下，以支持任意组层次。当使用 `EXPANDED` 关键字时，组将在安装程序 UI 中展开，而 `BOLD_TITLE` 关键字将使该组以粗体显示。

理想情况下，组件名称应该特定于项目，以分层的项目方式来选择要打包的组件，以及如何在安装程序中显示它们(或者在非UI安装程序中，如何构造特定于组件的包)。组名的限制较少，因为它们可能包含来自不同项目的组件和组。组名不能与组件名相同。

`cpack_add_component()` 和 `cpack_add_component_group()` 的作用是在当前范围内定义一组特定于组件的变量。`CPackComponent` 文档列出了其中一些变量，并建议可以直接设置这些变量，但不建议这样做。命令提供了一种更健壮、可读性更好的定义组件和组信息的方式。它们应该与 `include(CPack)` 在相同的作用域中，最好是在 `include(CPack)` 之后立即调用。从技术上来说，约束没有这么严格，但是在不同的范围中定义组件会让关联性变得脆弱。

一个示例有助于说明上面的概念和讨论：

```
set(CPACK_PACKAGE_NAME ...)  
# ... set other variables as per earlier example  
  
include(CPack)  
  
cpack_add_component(MyProj_Runtime  
DISPLAY_NAME Runtime  
DESCRIPTION "Shared libraries and executables"  
REQUIRED  
INSTALL_TYPES Full Developer Minimal  
)  
cpack_add_component(MyProj_Development  
DISPLAY_NAME "Developer pre-requisites"  
DESCRIPTION "Static libraries and headers needed for building apps"  
DEPENDS MyProj_Runtime  
GROUP MyProj_SDK  
INSTALL_TYPES Full Developer  
)  
cpack_add_component(MyProj_Samples  
DISPLAY_NAME "Code samples"  
GROUP MyProj_DevHelp  
INSTALL_TYPES Full Developer  
DISABLED  
)  
cpack_add_component(MyProj_ApiDocs  
DISPLAY_NAME "API documentation"  
GROUP MyProj_DevHelp  
INSTALL_TYPES Full Developer  
DISABLED  
)  
cpack_add_component_group(MyProj_SDK  
DISPLAY_NAME SDK  
DESCRIPTION "Developer tools, libraries, etc."  
)  
cpack_add_component_group(MyProj_DevHelp  
DISPLAY_NAME Documentation  
DESCRIPTION "Code samples and API docs"  
PARENT_GROUP MyProj_SDK  
)  
cpack_add_install_type(Full)  
cpack_add_install_type(Minimal)  
cpack_add_install_type(Developer DISPLAY_NAME "SDK Development")
```

项目生成器可以要求以三种方式的其中一种处理组件，选择由 `CPACK_COMPONENTS_GROUPING` 控制，可以设置为以下值：

#### `ALL_COMPONENTS_IN_ONE`

将创建单个包的所有请求组件，组件和组结构忽略。

### *ONE\_PER\_GROUP*

每个顶层组件组应该创建一个包，那些不属于组的组件将创建自己的包。如果没有设置 `CPACK_COMPONENTS_GROUPING` (这是默认行为)，并且通常是理想的安排，但是对于UI安装程序，隐藏了项目想要显示的组件。

### *IGNORE*

每个组件会创建自己的包，而不考虑任何组件组。此设置可能更适合UI安装程序，以确保没有隐藏组件(除非显式地配置为隐藏)。

还有两个变量也会影响生成器解释组件的方式。如果 `CPACK_MONOLITHIC_INSTALL` 设置为`true`，组件将完全禁用，所有组件将安装到一个包中。因此要在所有平台上仔细测试结果，并特别注意寻找任何意外的情况。由于传统的原因，每个生成器有自己的设置，用于在默认情况下确定组件的支持程度。这个设置可以由 `CPACK_<GENNAME>_COMPONENT_INSTALL` 在每个生成器的基础上覆盖，该变量可以根据需要设置为`true`或`false`。

执行基于组件的安装时，项目不需要包括最终包中的所有组件。包含的组件集由 `CPACK_COMPONENTS_ALL` 控制，必须在 `include(CPack)` 之前设置该变量。如果没有设置，CPack将打包所有组件，项目可以显式地设置变量，可以只列出打包的组件。例如，如果项目想要控制文档和代码是否应该打包，可以这样来实现：

```
if(NOT MYPROJ_PACKAGE_HELP)
set(CPACK_COMPONENTS_ALL
MyProj_Runtime
MyProj_Development
)
endif()
include(CPack)
```

项目可能希望安装除少数特定组件外的所有特定组件，而不是显式地列出要打包的所有组件。只读伪属性 `COMPONENTS` 中有完整的组件集，可以通过 `get_cmake_property()` 检索。项目可以从该组件列表删除不需要的条目。

```
if(NOT MYPROJ_PACKAGE_HELP)
get_cmake_property(CPACK_COMPONENTS_ALL COMPONENTS)
list(REMOVE_ITEM CPACK_COMPONENTS_ALL
MyProj_Samples
MyProj_ApiDocs
)
endif()
include(CPack)
```

开始选择要安装的组件集，以及如何处理组件似乎有点复杂。造成困难的主要原因是理解每个包生成器如何处理 `CPACK_COMPONENTS_GROUPING`。本章后面的内容将解释每种生成器类型的行为，但在测试项目中会进行一些快速实验，对于了解不同设置的效果通常有指导意义。

## 26.3. 多配置包

CPack本质上为单个构建配置生成包。大多数情况下，包为发布构建类型创建，但对于像SDK这样的项目，可能需要同时包含库的调试版本和发布版本。要将这两种配置都构建，并打包到一组包中，需要多做一些工作。

CPack提供了高级变量 `CPACK_INSTALL_CMAKE_PROJECTS`，可以将多个构建树合并到一个包中。预计每个包中会有四个部分，这四个部分是：

- 构建目录。

- 项目名称(对多配置生成器很重要)。
- 安装组件。 ALL 会列出 CMAKE\_COMPONENTS\_ALL 变量中的所有组件。其他值需要定义类似于 CMAKE\_COMPONENTS\_XXX 的变量，该变量只保存组件名。例如，如果要安装的组件名为 Runtime，则需要定义变量 CMAKE\_COMPONENTS\_RUNTIME，并将其值设置为 Runtime。
- 软件包中的相对位置。安全值是正斜杠(/)，因为不同的包生成器会以不同的方式使用它。

项目的四部分中，一部分用于发布，其余部分用于调试。发行版的构建目录可以定义为 CMAKE\_BINARY\_DIR，对于调试构建，需要创建并构建单独的构建目录。

调试组只需要添加两种构建配置之间不同的组件，但无论是使用默认的 ALL 还是使用特定组件，都需要特别小心，确保安装的文件不会相互覆盖。最后发布组件将确保所有具有相同名称和安装位置的文件，并在打包时以发布版本进行。

```
set(CPACK_COMPONENTS_MYPProj_RUNTIME MyProj_Runtime)
set(CPACK_COMPONENTS_MYPProj_DEVELOPMENT MyProj_Development)

unset(CPACK_INSTALL_CMAKE_PROJECTS)
if(MYPProj_DEBUG_BUILD_DIR)
list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
${MYPProj_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Runtime /
${MYPProj_DEBUG_BUILD_DIR} ${CMAKE_PROJECT_NAME} MyProj_Development /
)
endif()
list(APPEND CPACK_INSTALL_CMAKE_PROJECTS
${CMAKE_BINARY_DIR} ${CMAKE_PROJECT_NAME} ALL /
)
include(CPack)
```

当使用多配置生成器 Xcode 或 Visual Studio，MYPProj\_DEBUG\_BUILD\_DIR 目录在上面的例子中需要配置为只支持调试类型，而不是默认设置。这是唯一使安装调试构建输出，而不是以相同的配置为主要项目的方式。在该调试构建目录中运行 cmake 时，可以显式地将 CMAKE\_CONFIGURATION\_TYPES 缓存设置为 Debug。

虽然可以只对多配置生成器使用一个构建目录，但这样更加脆弱和复杂。相反，上述技术适用于所有构建和包生成器类型。此外，还可以扩展合并不同体系结构的构建，甚至将完全独立的项目合并到统一的包或一组包中。

## 26.4. 包生成器

CPack 支持创建各种格式的包，所有这些为以下类别之一：

### 简单的打包

打包文件可以是各种格式，比如 zip、tarball、bz2 等等。它们是包格式中最基本的，因为这只是用户在文件系统，某个位置文件的打包。它们是所有包格式中支持程度最好的，并且当最终用户想要多个不同版本可用或同时安装时，它们简单易用。

### UI 安装器

倾向于与支持的目标平台深度集成，提供诸如在安装后添加和删除组件、与桌面菜单集成等特性。通常为用户提供选择安装哪些组件的方法，而且非常直观，因此新手用户往往更喜欢。CMake 支持 Windows 上的 NSIS 和 WIX 安装程序，Mac 上的 DragNDrop(比如 DMG) 和 productbuild，以及 Windows、Mac 和 Linux 上的 Qt 安装程序框架(IFW)。Mac 的一些旧的安装程序类型仍然支持，后面的章节会简要提及。

### 非 UI 包

针对特定的包管理器。RPM和DEB在Linux上非常流行，FreeBSD和Cygwin的包在各自的平台上能很好的支持。

### 特定于产品的包

CMake 3.12为.NET添加了NuGet格式的支持。

无论生成哪种类型的包，使用的是相同的 `cPackConfig.cmake`，通常不会出现问题，因为特定于生成器的配置通常是通过特定于生成器变量实现的。如果需要为特定的生成器添加某些逻辑，并且CMake和CPack提供的现有变量不够，可以将 `CPACK_PROJECT_CONFIG_FILE` 设置为文件的名称，该文件将在调用每个包生成器时包含一次。每次读取时，`CPACK_GENERATOR` 将保存生成器的名称，而不是整个生成器列表。生成器需要时，可以通过该文件覆盖 `CPackConfig.cmake` 中的设置。完整的CPack按照以下伪代码步骤运行：

```
include(CPackConfig.cmake)

function(generate CPACK_GENERATOR)
    # CPACK_GENERATOR is a single generator local to this function scope
    if(CPACK_PROJECT_CONFIG_FILE)
        include(${CPACK_PROJECT_CONFIG_FILE})
    endif()

    # ...invoke package generator
    endfunction()

# Here CPACK_GENERATOR is the list of generators to be processed,
# as set by CPackConfig.cmake or on the cpack command line
foreach(generator IN LISTS CPACK_GENERATOR)
    generate(${generator})
endforeach()
```

上面的例子是将 `CPACK_PACKAGE_ICON` 设置为生成器特定的值，因为不同的生成器希望这个图标具有不同的格式，因此文件名需要区分生成器。

本章的其余部分将讨论CMake/CPack提供的包生成器。

#### 26.4.1. 打包文件

CPack支持以多种不同格式创建打包文件。最受广泛支持的是ZIP和TGZ，前者在Windows平台上很常见，后者生成gzip压缩的tar包(.tar.gz或.TGZ)，在其他平台也都受支持。其他可用的存档格式包括TBZ2 (.tar.bz2)、TXZ (.tar.xz)、TZ (.tar.z)和7Z (7zip archives .7Z)。为了可移植性最大化，通常应该选ZIP和TGZ，但其他格式可能生成较小的存档，并且可能适合那些支持该格式的平台。

CPack还支持解压打包格式。这可以使用STGZ生成器进行，它会生成Unix Shell脚本，该脚本末尾嵌入了打包文件。这可以看作是一种基于控制台的UI安装程序，但实际上只提供非常基本的功能，用户可能更喜欢可以自己解包的打包文件。

由于历史原因，打包生成器在默认情况下禁用了组件。要启用基于组件的打包格式，必须将 `CPACK_ARCHIVE_COMPONENT_INSTALL` 设置为true，然后 `CPACK_COMPONENTS_GROUPING` 确定要生成的打包文件集。

执行非组件安装时，可以使用 `CPACK_ARCHIVE_FILE_NAME` 控制包的文件名。对于基于组件的安装，每个组件包的名称由 `CPACK_ARCHIVE_<COMP>_FILE_NAME` 控制，其中 `<COMP>` 是大写的组件名或组名。适当的打包扩展名将附加到指定的文件名后(例如.tar.gz,.zip等)。

打包文件的常用约定是，提取目录结构的顶层与打包文件的名称相同，但不带文件扩展名(即与 `CPACK_PACKAGE_FILE_NAME` 相同)。对于非组件安装，这是打包生成器的默认行为，但是对于多组件包，默认情况下不使用顶层目录，通过将 `CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY` 设置为true，项目可以为组件打包指

定公共顶层目录。由于这个变量是所有包生成器共享的，特定生成器可以很容易做到这点：

#### CMakeLists.txt

```
set(CPACK_PROJECT_CONFIG_FILE
${CMAKE_CURRENT_LIST_DIR}/cpackGeneratorOverrides.cmake
)
```

#### cpackGeneratorOverrides.cmake

```
if(CPACK_GENERATOR MATCHES "^(7Z|TBZ2|TGZ|TXZ|TZ|ZIP)$")
set(CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY YES)
endif()
```

开发人员应该注意，一些打包格式、平台和文件系统对文件名和路径长度有限制。例如，**POSIX.2**要求文件名不超过100个字符，对于扩展的**tar**格式路径不超过255个字符，而旧的**tar**格式可能将整个路径限制为100个字符或更少。当解压缩到**eCryptFS**文件系统时，根据经验，文件名有大概140个字符的限制。**Windows**上解压的路径长度可以限制为260个字符，这取决于某些设置和操作系统版本。**UTF-8**文件名和路径使情况更加复杂，并可能会缩短有效字符的长度。

考虑到这些约束，项目应该避免在安装包中使用长路径和文件名。这些限制在包类型中最为明显，由于其他非打包格式也在内部使用打包方式，并将其部署到具有这些限制的系统中，因此通常应该选较短的路径和文件名。

### 26.4.2. Qt安装程序框架(IFW)

**IFW**包生成器为**CPack**提供基于UI的打包格式，可以使用相同的配置信息为**Windows**、**Mac**和**Linux**提供安装程序，这使得想要在主要桌面平台上拥有一致的UI安装程序时，可以选择该生成器。还可以显示组件和组的本地化名称和描述，以及灵活的自定义能力。

使用UI外观和安装程序图标的默认，通常就足够了，但有些项目可能希望定制化的图标来表明品牌。生成器会忽略 `CPACK_PACKAGE_ICON`，依赖于三个独立的**IFW**变量来控制不同的图标：

- `CPACK_IFW_PACKAGE_ICON` (**Windows**是.ico, **Mac**是.icns, **Linux**没有)
- `CPACK_IFW_PACKAGE_WINDOW_ICON` (.png)
- `CPACK_IFW_PACKAGE_LOGO` (最好是.png)

这些变量在不同平台之间无法统一处理，因此很难正确地设置。简单起见，最好将这三个图像设置为相同的图像，尽管可能采用不同的格式和/或大小。建议在感兴趣的平台上进行测试，以确保安装程序按照预期的方式呈现。下面的示例展示了如何进行这样的配置：

```
# Define generic setup for all generator types...

# IFW-specific configuration
if(WIN32)
set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
elseif(APPLE)
set(CPACK_IFW_PACKAGE_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.icns)
endif()
set(CPACK_IFW_PACKAGE_WINDOW_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.png)
set(CPACK_IFW_PACKAGE_LOGO ${CMAKE_CURRENT_LIST_DIR}/Logo.png)

include(CPack)
include(CPackIFW)
# Define components and component groups...
```

**IFW**生成器默认启用基于组件安装。生成单独的安装程序，但 `CPACK_COMPONENTS_GROUPING` 可以控制为用户显示的层数次：

#### `ALL_COMPONENTS_IN_ONE`

不显示组件层次结构，始终安装默认启用的组件。

#### `ONE_PER_GROUP`

只显示第一级组，以及不属于任何组的组件。任何组下的子组和组件将隐藏。

#### `IGNORE`

显示所有隐藏组件，而不管在组层次结构中的位置。这可能是大多数项目想要使用的选项。

命令可以提供组件和组的更详细配置：

```
cpack_ifw_configure_component(componentName
  [NAME componentNameId]
  [DISPLAY_NAME displayName...]
  [DESCRIPTION description...]
  [VERSION <version>]
  [DEPENDS compId1 [compId2...] ]
  [REPLACES compId3 [compId4...] ]
  # Other options not shown
)

# The cpack_ifw_configure_component() command supports all
# of the above options too
```

可以为每个组件或每组件组的 `DISPLAY_NAME` 和 `DESCRIPTION` 提供不同语言和地区的内容。这两个选项为一个对列表，其中第一个值是语言或区域ID，第二个值是该语言的文本内容。列表中的第一个值可以在没有前面的语言或语言环境ID的情况下给出，如果在安装时没有任何语言或语言环境ID与用户当前的设置匹配，则该值将当作默认文本。

```
cpack_ifw_configure_component(MyProj_Docs
  DISPLAY_NAME Documentation
  de Dokumentation
  pl Dokumentacja
)

cpack_ifw_configure_component_group(MyProj_Colors
  DISPLAY_NAME en Colors
  en_AU Colours
  DESCRIPTION en "Available color palettes"
  en_AU "Available colour palettes"
)
```

`VERSION` 选项允许指定组件和组的版本号。在线安装程序使用它来确定更新是否可用(参见后面的部分)。如果没有给出 `VERSION`，则默认为 `CPACK_PACKAGE_VERSION`。

`DEPENDS` 类似于 `cpack_add_component()` 的选项，不同的是 `compId1...` 项的形式不同。需要遵循**QtIFW**样式，这是一个层次字符串，而不是 `componentName`。分组层次结构的每一层都用点分隔，如下面的例子所示：

```

include(CPack)
include(CPackIFW).

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar DEPENDS groupA.foo)

```

组件安装程序内部使用的名称可以用 `NAME` 覆盖。此名称将用于标识 `DEPENDS` 参数中的组件，也用于检查组件的新版本是否可用。顶层组名可以用 `CPACK_IFW_PACKAGE_GROUP` 设置，通常设置为反向域名，以确保组件名在大型、多供应商的安装程序中不会发生冲突。当列出与 `DEPENDS` 选项的依赖项时，必须包含顶层组名，如下面修改示例所示：

```

set(CPACK_IFW_PACKAGE_GROUP com.examplecompany.product)

include(CPack)
include(CPackIFW)

cpack_add_component(foo GROUP groupA)
cpack_add_component(bar GROUP groupB)

cpack_add_component_group(groupA)
cpack_add_component_group(groupB)

cpack_ifw_configure_component(bar
    DEPENDS com.examplecompany.product.groupA.foo
)

```

`CPACK_IFW_PACKAGE_GROUP` 只是额外变量的一个，可以用来设置特定于 IFW 的配置。这些变量应该在 `include(CPackIFW)` 之前设置，并且可以以各种方式修改安装程序的显示和行为。`CPackIFW` 模块文档提供了支持变量的完整清单，其中许多变量在 QtIFW 产品的本机配置中有类似设置。大多数变量都有合理的默认值，应该更多地将其视为可定制的选项。例外是（需要安装的）维护工具名称的相关变量，它允许用户修改已安装的组件集或删除产品。一些平台上，工具名可能出现在桌面或应用程序菜单中，默认名称可能会让用户感到困惑。因此，项目应该提供具体的名称，可以这样做：

```

set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME ${PROJECT_NAME}_MaintenanceTool)
set(CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_INI_FILE
    ${CPACK_IFW_PACKAGE_MAINTENANCE_TOOL_NAME}.ini)
include(CPackIFW)

```

`.ini` 文件用于提供安装程序的状态信息。`.ini` 文件的名称选择性设置，最好与安装程序一致。通过设置，当工具出现在桌面或应用程序菜单中，用户将看到与项目相关的名称。

IFW 生成器的特性是能创建在线安装，部分或所有组件都可以按需下载，而不是打包成安装程序的一部分。如果可选组件非常大，这就特别方便。在线安装的另一个好处是，可以从在线存储库获得更新的版本，这提供了一个非常方便的升级方式。用户运行工具时，工具会联系在线存储库集，确定可用的组件及其版本，然后根据需要添加、删除或升级单个组件。

将项目配置为支持可下载组件的第一步，是指定安装程序从何处下载这些组件。主默认存储库使用 `cpack_configure_downloads()` 指定：

```
cpack_configure_downloads(baseUrl
[ALL]
[ADD_REMOVE | NO_ADD_REMOVE]
[UPLOAD_DIRECTORY dir]
)
```

`baseUrl` 是安装程序寻找可下载组件的位置。安装程序希望在该位置下找到一个名为 `updates.xml` 的文件。如果存在 `ALL` 关键字，所有组件都视为可下载，而不管是否显式标记为可下载，这是制作在线安装程序的一种方法。因为不需要嵌入包，从而生成最小的安装程序。

关键字 `ADD_REMOVE` 引导安装程序，使用 Windows 的“添加/删除程序”功能。当用户通过 Windows 系统设置选择修改包时，将运行工具。`ALL` 关键字意味着 `ADD_REMOVE`，可以通过 `NO_ADD_REMOVE` 进行覆盖。

`UPLOAD_DIRECTORY` 选项支持可下载组件的 CPack 生成器类型(尽管这些组件都没有积极维护)，IFW 生成器会忽略该选项。当 CPack 运行时，在单独的目录中创建可下载包，这样整个目录的内容就可以上传到 `baseUrl` 位置(必须手动完成)。`UPLOAD_DIRECTORY` 选项旨在允许项目覆盖此目录所在的位置，但 IFW 生成器总会创建名为 `repository` 的目录，该目录位于基 `_CPack_Packages` 目录下。

IFW 生成器允许项目指定维护工具和安装程序访问的附加存储库。如果不同的组件是由不同的供应商提供，或者某些组件与其他组件有不同的发布计划，那这个功能会非常有用。

```
cpack_ifw_add_repository(repoName
URL baseUrl
[DISPLAY_NAME displayName]
[DISABLED]
[USERNAME username]
[PASSWORD password]
)
```

`repoName` 是内部跟踪名称，`baseUrl` 与 `cpack_configure_downloads()` 有类似的含义。`DISPLAY_NAME` 选项通常用于提供名称，否则 `baseUrl` 将显示为存储库名称，这对用户不太友好。如果存储库需要用户名和密码，可以提供，但是请注意，密码将以未加密的方式存储是不安全的。`DISABLED` 关键字表示存储库在默认情况下禁用，但用户可以在安装程序或维护工具的 UI 中启用它。

用于发布包的主存储库和用于预览包的辅助存储库(默认禁用)的例子可以这样配置：

```
include(CPack)
include(CPackIFW)

cpack_configure_downloads("https://example.com/packages/product/release ALL")
cpack_ifw_add_repository(secondaryRepo
DISPLAY_NAME "Preview features"
URL "https://example.com/packages/product/preview"
DISABLED
)
```

不幸的是，`cpack_configure_downloads()` 目前不支持指定名称，因此主 `URL` 将显示为纯 `URL`，而不是对用户更友好的名称。

包生成器的缺点是，生成的安装程序不能为用户提供无交互的命令行安装方法。这是 Qt 安装程序框架本身的限制，而不是 CMake 或 CPack 的限制。与大多数其他生成器类型相比，因为包含安装程序的接口、网络等所需的 Qt 支持，所以安装程序有额外的开销。与其他生成器的几百 kB 相比，这会使普通安装程序的大小达到 18MB 或更大。

上面的讨论只涵盖了IFW生成器的主要方面，还有相当多的可用功能允许项目定制安装程序和维护工具。对于许多项目来说，上面的功能允许创建灵活、健壮和跨平台的安装程序。如果需要进一步的裁剪，所提供的特性将作为扩展的基础。

### 26.4.3. WIX

WIX包生成器使用WIX工具集为Windows生成.msi安装程序。与IFW包生成器相比，具有类似程度的UI可定制性，有以下优势：

- 通过msiexec工具的选项，可以直接支持命令行(即无人值守)安装。
- 安装程序与Windows的添加/删除功能直接集成。
- 默认的界面外观为大多数用户所熟悉。

与IFW相比，有以下缺点：

- 没有简单、直接的方法来提供本地化的组件名称和描述。
- 会忽略 CPACK\_WIX\_COMPONENT\_INSTALL 和 CPACK\_COMPONENTS\_GROUPING 选项。
- 不支持可下载组件。
- 不能同时安装具有相同级别GUID的多个版本。每个安装都会替换前一个，即使是在完全不同的目录下。

默认情况下，WIX生成器生成一个基于组件的包，会在UI中显示，就好像将 CPACK\_COMPONENTS\_GROUPING 设置为 IGNORE 一样。如果不希望使用基于组件的包，可以将 CPACK\_MONOLITHIC\_INSTALL 设置为true，安装所有已定义的组件。独立的安装程序中不可能只包含一些组件，如果设置了 CPACK\_COMPONENTS\_ALL，CMake会发出警告并忽略 CPACK\_COMPONENTS\_ALL。

WIX安装程序的关键部分是包含产品GUID和升级GUID。如果其他已安装的包具有相同的升级GUID，则升级该包，而不是将新包作为单独的产品安装。如果升级GUID是相同的，但是产品的GUID是不同的，那么就认为是一次重大的升级，新的安装程序将完全替换旧的包。如果产品GUID也相同，只要安装程序的版本号比当前安装的包更新，那么新的安装程序将执行升级。服务包就是这样一个示例，其中将相同的产品GUID应用到基本版本进行维护。除非创建高级的安装程序或打包策略，否则项目通常需要在每个版本中更改产品GUID，因为Windows本身对于在不同包之间的产品GUID约束相当严格。

CPack提供了对设置产品和升级GUID的支持。CPACK\_WIX\_PRODUCT\_GUID 和 CPACK\_WIX\_UPGRADE\_GUID 可以在 include(CPack) 之前设置(以手动控制)，或者可以不设置，允许CPack在每次调用时生成新值。对于产品GUID，这种自动生成可能是期望行为，但是升级GUID在理想情况下在产品生命周期中应该永远不变。项目应该获取GUID，并将 CPACK\_WIX\_UPGRADE\_GUID 设置为该值，不要再更改。这将确保所有未来版本都能够无缝地升级。GUID可以通过各种方法获得，比如命令行工具、基于Web的UUID生成器，甚至使用 string(UUID) 命令与CMake本身一起获得。对于某些产品，升级GUID随着每个主要版本的变化而变化，以允许较旧版本与新版本共存。

产品GUID必须更改的条件之一是.msi文件名称的更改。安装程序的文件名通常会包含一些版本细节，这意味着每个版本都将视为一次重大升级。如果用户安装了新版本，将完全取代以前安装的版本。可以将新版本安装到不同的目录，并将旧的目录删除。使用包含版本号的默认安装目录(由 CPACK\_PACKAGE\_INSTALL\_DIRECTORY 控制)可能很有诱惑力，但是用户很可能希望在升级时默认目录保持不变。理想情况下，只有升级GUID更改时，默认目录才应该更改，因为GUID是提供从一个版本到另一个版本的标识。

安装新包，并且已经安装了具有相同升级GUID的另一个包时，会在版本之间进行检查。只有当新包的版本更新时，升级才进行。测试中只考虑了前三个版本号组件，因此版本2.7.4.3和2.7.4.9从升级的角度来看是相同的版本。因此，打算使用WIX生成器的项目应该避免使用超过三个版本号组件。如果 cpackage\_version\_xxx 的版本部分自动设置为 CPACK\_PACKAGE\_VERSION，则会强制执行升级。

对于基本的WIX包，UI大多数默认值都还不错。项目可能希望提供产品图标来代替通用的MSI安装程序图标，用于在添加/删除区域显示品牌，其他的默认值可以接受。下面的示例展示了WIX安装程序的基本配置。

```

# Define generic setup for all generator types...

# WIX-specific configuration
set(CPACK_WIX_PRODUCT_ICON ${CMAKE_CURRENT_LIST_DIR}/Logo.ico)
set(CPACK_WIX_UPGRADE_GUID XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX)

include(CPack)
# Define components and component groups...

```

## 26.4.4. NSIS

**NSIS**包生成器使用Nullsoft脚本安装系统，为Windows生成安装程序可执行文件。它与IFW和WIX生成器有许多相似的特性，包括一定程度的UI可定制性和对组件层次结构的支持。**NSIS**生成器的优点有：

- 安装程序的可执行文件，可以通过专用的命令行选项直接支持无人值守的安装。
- 是唯一积极维护的安装类型的CPack生成器。
- 支持安装前/安装后和卸载前命令，尽管这些命令必须实现为**NSIS**命令。

**NSIS**生成器有几个缺点：

- 忽略 CPACK\_NSIS\_COMPONENT\_INSTALL 和 CPACK\_COMPONENTS\_GROUPING 选项。**NSIS**生成器与WIX生成器具有相同的限制。
- 不支持可下载的组件。
- 安装了产品后，用户只能在重新安装的情况下更改已安装的组件集。
- 只支持基本的UI定制，不直接支持任何UI内容的本地化。这些是CPack生成器的限制，而不是**NSIS**本身的限制，**NSIS**通过本地脚本语言提供了一些工具。
- 尽管可以将不同的版本安装到不同的位置，但它们会共享注册表信息，因此不能完全隔离。所以，只有一个版本会出现在Windows设置的添加/删除区域。

默认情况下，如果新包与旧包安装在同一目录下，这些安装程序将只执行对现有产品安装的升级。项目可以将 CPACK\_NSIS\_ENABLE\_UNINSTALL\_BEFORE\_INSTALL 变量设置为true，强制安装程序检查注册表是否已有安装。这种检查不依赖于安装位置，因此是检查要升级的现有安装的可靠方法。因此，建议对大多数项目将该变量设置为true。

**NSIS**安装程序可以覆盖默认外观。安装程序、卸载程序和产品本身在添加/删除区域显示的图标都可以设置，因为默认的显示太差强人意，显式地设置产品显示的名称，以避免CPack提供不适当的默认名称。下面的示例展示了一个覆盖的基本配置，以避免使用不合适默认值的情况。

```

# Define generic setup for all generator types...

# NSIS-specific configuration
set(CPACK_NSIS_MUI_ICON ${CMAKE_CURRENT_LIST_DIR}/InstallerIcon.ico) ①
set(CPACK_NSIS_MUI_UNIICON ${CMAKE_CURRENT_LIST_DIR}/UninstallerIcon.ico) ②
set(CPACK_NSIS_INSTALLED_ICON_NAME bin/MainApp.exe) ③
set(CPACK_NSIS_DISPLAY_NAME "My Project Suite") ④
set(CPACK_NSIS_PACKAGE_NAME "My Project") ⑤

set(CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL YES)

include(CPack)
# Define components and component groups...

```

① 安装程序的图标。Windows可能会有更多内容，以表明安装程序需要管理员权限。使用绝对路径确保**NSIS**在创建安装程序时能够找到图标。

② 用于安装/卸载程序的图标。同样，使用图标文件的绝对路径。

③ 这将控制在添加/删除区域中使用的图标。必须是图标文件(.ico)或有嵌入图标应用的可执行文件路径。路径应该是相对于安装位置的基点。

④ 仅在添加/删除区域中显示的包的名称。

⑤ 安装程序的UI和安装期间在标题栏中使用的名称。在某些上下文中，可能会加上“Setup”。

## 26.4.5. DragNDrop

Mac上产品通常以.dmg文件的形式发布，作用类似于磁盘镜像，可以包含从单个应用程序到整个应用程序套件、文档链接等任何内容。/Applications 区域的符号链接经常作为镜像的一部分，以便用户可以轻松地将应用程序拖放到镜像中进行安装，因此这种生成器类型的名称为DragNDrop。此生成器类型的配置变量在名称中使用DMG，而不是DRAGNDROP，但是请注意，CPack只将DRAGNDROP识别为生成器的名称。

dmg格式更接近于打包，而不是UI安装程序。组件用于控制创建一个或多个.dmg文件，以及每个.dmg文件包含什么，但没有安装时UI来选择组件。用户需要打开.dmg文件并将内容拖到所需的位置来安装它们。CPACK\_COMPONENTS\_ALL 控制安装了哪些组件，CPACK\_COMPONENTS\_GROUPING 变量控制这些组件如何在.dmg文件中分布，如下所示：

### ALL\_COMPONENTS\_IN\_ONE

所有组件都包含在一个.dmg文件中。

### ONE\_PER\_GROUP

每个顶层组件组和每个不在组中的组件，都将放在单独的.dmg文件中。

### IGNORE

每个组件将放在自己单独的.dmg文件中，并忽略所有组件组。

除了默认值之外，这种生成器类型几乎不需要自定义。打开磁盘镜像时，Finder窗口显示的大小和布局可以通过提供自定义的.DS\_Store文件来控制。该项目需要使用示例文件夹，需要手工准备这样的文件，其中包含与最终磁盘映像相同的内容，或者可以使用AppleScript编程创建。CPACK\_DMG\_DS\_STORE 可以用来命名预先准备好的.DS\_Store文件，或者CPACK\_DMG\_DS\_STORE\_SETUP\_SCRIPT 可以指向包生成时运行的AppleScript文件。对于这两种情况，可以使用CPACK\_DMG\_BACKGROUND\_IMAGE 设置背景图像，但是将背景保留为默认的空白比较常见。对于磁盘镜像不应该提供/Applications 文件夹的符号链接，项目应该将CPACK\_DMG\_DISABLE\_APPLICATIONS\_SYMLINK 设置为true。

将CPACK\_PACKAGE\_ICON 设置为.icns格式的图标，可以为磁盘镜像指定图标。此图标仅用于挂载时表示.dmg文件，而不用于.dmg文件本身。指定的图标可能显示在Finder标题栏或某些Finder视图中。

语言本地化通过CPACK\_DMG\_SLA\_DIR 和CPACK\_DMG\_SLA\_LANGUAGES 提供。这些可用于提供在许可协议阶段，打开磁盘镜像时，使用特定短语，并提供许可协议的本地化版本。请参阅CPackDMG模块文档，了解如何使用这两个变量，以及需要提供的语言文件的相关要求。

Bundle生成器类型与DragNDrop生成器相关。使用相同的DMG变量集，加上自己的一些变量。Bundle生成器最初目的是生成提交到Apple应用程序商店的应用程序包。如今，这样的应用程序包在使用CMake的Xcode生成器时，可以更好的构建，更接近Apple的期望。请参阅第22章，了解此类应用程序包的推荐方法。

## 26.4.6. productbuild

DragNDrop生成器的替代方法是productbuild。它不生成.dmg磁盘镜像，而是生成用于macOS安装程序应用程序的.pkg包。生成器不应该将CPACK\_MONOLITHIC\_INSTALL 设置为true，因为这样做会损坏的安装程序。安装程序类型不受支持，也很少能够定制UI。

与IFW生成器相比，`productbuild`的主要优点是能够为安装程序签名。通过将`CPACK_PRODUCTBUILD_IDENTITY_NAME`(还可以将`CPACK_PRODUCTBUILD_KEYCHAIN_PATH`)设置为签名信息，可以很容易地进行配置。通常只指定默认标识就足够了：

```
set(CPACK_PRODUCTBUILD_IDENTITY_NAME "Developer ID Installer")
include(CPack)
```

`productbuild`生成器缺乏对可下载组件的支持，因此无法创建在线安装程序。升级是通过替换现有安装的内容。与NSIS安装程序一样，只能在安装时，修改已安装的组件集。并且，不可能同时将多个版本安装到不同的目录。

默认情况下，`productbuild`生成器生成的安装程序可重定位，当包安装到用户的机器上时，如果操作系统知道一个与包中提供的应用程序同名的应用程序包，则安装程序将覆盖现有的应用程序，无论在文件系统的哪个位置。这种情况下，应用程序不会安装到默认的`/Applications`区域，这也不会出现在用户期望的地方，这种情况通常出现在开发人员用来构建和测试包的机器上。构建产生的应用程序包是操作系统已知的，所以在安装包时，构建树的应用程序包会用作该应用程序的安装位置，而不是`/Applications`中预期的位置。构建树的`_CPack_Packages`级目录中也会有该应用程序的副本，它可以产生类似的行为。为了正确地测试安装程序，在运行安装程序之前，需要先从开发人员的机器上删除应用程序包的所有副本。

解决上述重定位问题的方法是将组件标记为不可重定位。这就阻止了安装程序选择应用程序包的位置，但代价是阻止了用户移动应用程序包。要使组件不可重定位，需要使用`cpack_add_component()`的`PLIST`选项为每个组件提供一个定制的`plist`文件。`plist`文件应该使用`pkgbuild`命令的`--analyze`选项来获得，其他选项可以通过查看项目的`CPack`输出找到：

```
cpack -G productbuild -v
```

常规的`plist`文件看起来是这样的：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
<dict>
<key>BundleHasStrictIdentifier</key>
<true/>
<key>BundleIsRelocatable</key>
<true/>
<key>BundleIsVersionChecked</key>
<true/>
<key>BundleOverwriteAction</key>
<string>upgrade</string>
<key>RootRelativeBundlePath</key>
<string>Applications/MyApp.app</string>
</dict>
</array>
</plist>
```

将`BundleIsRelocatable`字典项改为`false`，以防止操作系统在安装时重新定位应用程序。组件中的每个应用程序包将有一个`<dict></dict>`节。当`plist`文件生成和更新时，可以这样使用：

```
cpack_add_component(MyProj_Runtime
... # Other options
PLIST runtime.plist
)
```

应该将productbuild生成器视为旧的不再支持PackageMaker生成器的替代品。Apple不再提供PackageMaker应用程序，因此使用新版macOS的开发者必须使用productbuild。

## 26.4.7. RPM

Linux系统上，RPM是两种主要的包管理格式之一。RPM包没有自己的UI特性，本质上只是带有数据集和一些脚本特性的打包文件。系统的包管理器使用这些来管理包之间的依赖关系，向用户提供信息，触发安装前/安装后脚本和卸载脚本等。

由于包本身没有UI特性，因此不需要对UI进行定制，但是RPM生成器通过变量提供了数据的可定制性。许多变量不需要显式地设置，因为大多数默认值都适合于不需要执行复杂操作的项目。对于不需要调用安装前/安装后或卸载脚本的包，并且可以由底层的包创建工具，自动确定包之间的依赖关系，定制的数量与其他包生成器类似。

RPM生成器支持组件安装，但默认情况下禁用组件。当组件禁用时，只生成一个.rpm，行为就好像 `CPACK_MONOLITHIC_INSTALL` 设置为true。这种情况下，所有组件都包含在包中。如果启用了组件，那么 `CPACK_COMPONENTS_GROUPING` 就有意义了，并且会创建多个.rpm文件。通过将 `CPACK_RPM_COMPONENT_INSTALL` 设置为true来启用组件，已安装的组件集通常由 `CPACK_COMPONENTS_ALL` 控制。

```
# Define generic setup for all generator types...
set(CPACK_COMPONENTS_GROUPING ONE_PER_GROUP)

# RPM-specific configuration
set(CPACK_RPM_COMPONENT_INSTALL YES)

include(CPack)
# Define components and component groups...
```

组件名或组名可能不适合用作包名，这些包名通常在RPM package manager UI应用程序中作为.RPM文件名的一部分对用户可见。可以使用 `CPACK_RPM_<COMP>_PACKAGE_NAME` 在每个组件的基础上设置这些名称，其中 `<COMP>` 是大写组件名称。当创建禁用组件的包时，可以通过设置 `CPACK_RPM_PACKAGE_NAME` 覆盖单个包名。

```
add_executable(sometool ...)
install(TARGETS sometool ... COMPONENT MyProjUtils)

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
include(CPack)
```

.rpm文件的名称也可以定制，每个组件的.rpm文件的名称是由 `CPACK_RPM_<COMP>_FILE_NAME` 变量控制(对于非组件打包，只需 `CPACK_RPM_FILE_NAME`)。这些变量的默认值遵循以下模式：

```
<CPACK_PACKAGE_FILE_NAME>[-<component>].rpm
```

`<component>` 部分是原始组件的名称(大写/小写没有变化)。这种默认文件名的缺点是它不包含任何版本或架构细节，但是这些信息通常是必需的(至少是希望有的)。最好是CPack让底层的包创建工具选择一个默认包名，这可以通过将 `CPACK_RPM_<COMP>_FILE_NAME` 设置为一个特殊的字符串RPM-DEFAULT来完成。

.rpm默认的包文件名将自动包含架构。如果需要显式地指定架构，比如将一个包标记为noarch，表明不特定于架构，`CPACK_RPM_<COMP>_PACKAGE_ARCHITECTURE` 变量可以设置为所需的值，如果没有设置特定于组件的覆盖，可以将 `CPACK_RPM_PACKAGE_ARCHITECTURE` 设置为默认值(也用于集成包)。体系结构的默认值由CPack作为uname -m的输出，但是如果在64位主机上构建32位包，这将会导致错误，因此项目最好显式地设置体系结构。

RPM文件需要有版本信息。RPM生成器将默认使用 `CPACK_PACKAGE_VERSION`，也可以使用 `CPACK_RPM_PACKAGE_VERSION` 设置RPM的版本号(这种情况应该很少)。注意，目前还不能指定每个组件的版本，CPack的 RPM生成器目前仅限于为所有组件设置相同的版本。除了包版本之外，RPM包还有单独的发布号，使用 `CPACK_RPM_PACKAGE_RELEASE` 指定。这个发布号是包本身的发布，而不是产品的发布，如果发布号增加了(例如，为了修复打包问题)，包版本通常会保持不变。如果包版本发生了变化，发布号会重新设置为1，这是没有指定 `CPACK_RPM_PACKAGE_RELEASE` 时的默认值。还可以通过 `CPACK_RPM_PACKAGE_EPOCH` 指定epoch，它在某些系统或存储库中的使用可能比其他系统或存储库更常见。完整版本的格式是 `E:X.Y.Z-R`，其中E是历元，如果提供的话必须是一个数字。如果没有设置历元，完整版本的格式为`X.Y.Z-R`。

除非知道需要的epoch值，否则项目通常不设置epoch。除非项目覆盖了 `CPACK_PACKAGE_VERSION` 和 `CPACK_RPM_PACKAGE_ARCHITECTURE` 的值，否则它们的值在CMakeLists.txt文件中是不可用的，因为这些变量的默认值只在CPack处理输入文件时会用到，而不是在CMake运行时。这意味着直接设置包文件名，要比使用RPM-DEFAULT做更多的工作。下面的例子展示了如何使用rpm默认特性：

```
set(CPACK_RPM_PACKAGE_RELEASE 5) # Optional, default of 1 is often okay
if(CMAKE_SIZEOF_VOID_P EQUAL 4)
    set(CPACK_RPM_PACKAGE_ARCHITECTURE i686)
endif()

set(CPACK_RPM_MYPROJUTILS_PACKAGE_NAME myproj-tools)
set(CPACK_RPM_MYPROJUTILS_FILE_NAME RPM-DEFAULT)
include(CPack)
```

对于上面的例子，假设 `CPACK_PACKAGE_VERSION` 计算结果是X.Y.Z格式的字符串，这个例子通常会得到这样的包文件名：

```
myproj-tools-X.Y.Z-5.i686.rpm
myproj-tools-X.Y.Z-5.x86_64.rpm
```

正如前一章所讨论的，Linux系统上不太可能需要默认的基本安装点，这也会扩展到了RPM包的创建。实际上，除了Windows系统之外，通常应该通过显式地设置 `CPACK_PACKAGING_INSTALL_PREFIX` 来为打包设置合适的基准点。扩展前一章的示例，该项目可能需要做如下操作：

```
if(NOT WIN32 AND CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    set(CMAKE_INSTALL_PREFIX "/opt/mycompany.com/${PROJECT_NAME}")
    set(CPACK_PACKAGING_INSTALL_PREFIX ${CMAKE_INSTALL_PREFIX})
endif()
```

RPM包的独特性是可以包含重新定位路径。包可以指定一个或多个路径前缀，然后用户可以选择在安装时将其重新定位到文件系统的某处。要支持这个特性，必须将 `CPACK_RPM_PACKAGE_RELOCATABLE` 变量设置为true，然后 `CPACK_RPM_RELOCATION_PATHS` 可以允许用户重新定位的路径前缀列表。如果使用这个特性，开发人员应该参考 CPackRPM模块文档，了解如何处理相对路径，以及适用于这两个变量的各种默认行为。还要注意的是，如果该项目是作为Linux发行版的一部分，发行版维护人员可能要重写安装前缀变量和重新定位目录。

RPM包创建工具通常会去除可执行文件和动态库中的调试符号。其基本原理是应该尽量减小发布二进制文件的大小，并且通常会隐藏实现细节，不提供调试工具。通常，剥离是由 `CPACK_STRIP_FILES` 控制，决定了在打包期间剥离是否作为阶段安装的一部分，在RPM生成器的情况下，RPM包创建工具通常在默认情况下执行自己的剥离操作。因此，即使 `CPACK_STRIP_FILES` 为false或未设置，剥离操作仍然可能发生。潜在的问题是包创建工具rpmbuild通常会在安装后进行的操作，会在创建最终的.rpm包之前删除二进制文件并执行其他任务。通常CPack提供的解决方案是通过设置 `CPACK_RPM_SPEC_INSTALL_POST`，内容类似于 `/bin/true` 的路径。不过这种方式已经废弃，推荐使用 `CPACK_RPM_SPEC_MORE_DEFINE` 进行代替：

```
# Prevent stripping and other post-install steps during package creation
set(CPACK_RPM_SPEC_MORE_DEFINE "%define __spec_install_post /bin>true")
```

虽然上述的防剥离技术有效，但也抛弃了通常会使用的其他操作(例如，`python`文件的自动字节码编译，特定于体系结构的后处理)。更好的替代方案是允许剥离.rpm中的二进制文件，并生成一个单独的`debuginfo`包。对生成`debuginfo`包的支持是在CMake 3.7中添加的，在3.8和3.9中得到了进一步的改进。要启用这个特性，通常只需要将`CPACK_RPM_DEBUGINFO_PACKAGE`或组件特定的`PACK_RPM_<COMP>_DEBUGINFO_PACKAGE`设置为`true`。生成的`debuginfo`包包含源文件和调试信息。默认情况下，源文件取自`CMAKE_SOURCE_DIR`和`CMAKE_BINARY_DIR`，也可以用`CPACK_BUILD_SOURCE_DIRS`覆盖。可以使

用`CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS`和`cpack_rpm_debuginfo_exclude_dirs_add`排除源目录层次结构的部分。前者通常用于排除系统目录，并具有适当的默认值。发行版维护人员可能希望独立于项目，在`CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION`中设置的内容重写`CPACK_RPM_DEBUGINFO_EXCLUDE_DIRS_ADDITION`，因此需要使用两个单独的变量。

生成`debuginfo`包时，有时会遇到如下错误：

```
CPackRPM: source dir path '/path/to/source/dir' is shorter
than debuginfo sources dir path
'/usr/src/debug/SomeProject-X.Y.Z-Linux/src_0'! Source dir path must be
longer than debuginfo sources dir path. Set
CPACK_RPM_BUILD_SOURCE_DIRS_PREFIX variable to a shorter value or make
source dir path longer. Required for debuginfo packaging. See
documentation of CPACK_RPM_DEBUGINFO_PACKAGE variable for details.
```

由于在`debuginfo`处理中重写路径的方式，到源码的路径需要比预期安装的源位置更长。注意，这会影响持续集成系统，其中源码的位置固定。这种对路径长度的需求可能与其他限制冲突，其中路径长度可能需要最小化，所以要仔细考虑这些限制是否适用于项目。

源码版本的RPM也可以由RPM生成器生成。它们类似于`debuginfo`包，但只包含源代码，不包含调试信息。生成方式与其他包生成器的源代码包相同，CPackRPM模块文档包括如何从源代码RPM构建二进制RPM的基本说明。

```
# Create source RPM
cpack -G RPM --config CPackSourceConfig.cmake

# Verify that a binary RPM can be produced from it
mkdir -p build_dir/{BUILD,BUILDROOT,RPMS,SOURCES,SPECS,SRPMS}
rpmbuild --define "_topdir build_dir" --rebuild <source-RPM-filename>
```

CPackRPM模块提供了比上面讨论更多的变量。可以指定包提供相应的信息，或者指导包创建工具自动计算。如果包替换有冲突，也可以指定。可以提供在包安装和卸载之前或之后运行的脚本，或者项目可以提供自己定制的`.spec`文件模板，而不是使用CPack提供的默认模板(这应该尽可能避免，因为定制化会否决CPack提供的大部分功能)。

## 26.4.8. DEB

DEB格式是Linux系统的另一种主流包格式，DEB和RPM都有许多相似的特性。DEB包只包含相关元数据信息，系统包管理器使用这些元数据来描述依赖关系、触发脚本等。

DEB和RPM之间的区别是DEB包不需要特殊的工具。这使得DEB包可以在不使用DEB格式的系统上创建，这可以在基于RPM的系统(如RedHat、SuSE等)上同时生成RPM和DEB包。当在非DEB系统上创建DEB包时，像`dpkg-shlibdeps`这样的工具不可用，因此依赖关系就无法计算了。

组件的处理方式与RPM类似，并具有类似的配置。组件是通过设置 `CPACK_DEB_COMPONENT_INSTALL` 为true来启用的(这个变量不遵循特定于DEB变量的命名，这些变量的名称前缀是 `CPACK_DEBIAN_` 而不是 `CPACK_DEB_`)。包名类似于 `CPACK_DEBIAN_PACKAGE_NAME` 和 `CPACK_DEBIAN_<COMP>_PACKAGE_NAME` 变量，文件名由 `CPACK_DEBIAN_FILE_NAME` 和 `CPACK_DEBIAN_<COMP>_FILE_NAME` 变量控制。DEB和RPM存在相同的文件命名问题，只是应该使用DEB-DEFAULT，而不是RPM-DEFAULT。如果提供其他值，文件名必须以.deb或.ipk结尾。`DEB`的版本控制与RPM非常相似，即指定架构。提供了等效的`DEB`变量，在变量名中使用`DEBIAN`替换`RPM`。

与RPM相比，DEB包生成器没什么变量来影响依赖项的处理。如果打包是在基于DEB的主机上执行，则`dpkg-shlibdeps`工具可用，可以通过设置 `CPACK_DEBIAN_PACKAGE_SHLIBDEPS` 或组件特定的 `CPACK_DEBIAN_<COMP>_PACKAGE_SHLIBDEPS` 为true来自动计算动态库的依赖关系。手动指定的依赖项可以通过 `CPACK_DEBIAN_PACKAGE_DEPENDS` 和 `CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS` 提供，如果同时使用手动和自动依赖项，则与依赖项合并。请注意，如果设置了特定于组件的依赖变量，则不会为该组件使用非组件变量。如果启用依赖计算，则将填充特定的组件变量，因此如果只设置 `CPACK_DEBIAN_PACKAGE_DEPENDS`，对于自动依赖的填充组件，会忽略该变量。因此，自动依赖启用时总会填充 `CPACK_DEBIAN_<COMP>_PACKAGE_DEPENDS`，而不是 `CPACK_DEBIAN_PACKAGE_DEPENDS`，这样会让项目更健壮。如果通过 `cpack_add_component()` 的 `DEPENDS` 选项指定组件间依赖关系，项目也应该将 `CPACK_DEBIAN_ENABLE_COMPONENT_DEPENDS` 设置为true，然后该选项将在生成的组件包中强制执行这些依赖关系。

每个包还可以指定需要的动态库。在提供`readelf`的平台上，可以通过将 `CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS` 设置为true来自动确定这些库的依赖关系。然后使用`readelf`工具确定每个共享对象需要的动态库，并将这些信息添加到包中。`CPACK_DEBIAN_PACKAGE_GENERATE_SHLIBS_POLICY` 变量控制是否执行精确(=)或最小(>=)需求。

CPackDeb模块文档详细介绍了许多前面没有提到特定于DEB的变量。特别是，一些变量可以用来指定包需要什么、提供什么、替换什么等。还可以设置一些特定于DEB的元数据项，比如维护人员信息、包组或类别等。开发人员应该参考模块文档，以获得更多信息。

## 26.4.9. FreeBSD

FreeBSD包生成器相对不成熟，在CMake 3.10中添加。不支持组件，生成一个.pkg文件。一些FreeBSD特定的变量可以指定包的元数据，还有一些和DEB或RPM特定变量相同。大部分包配置可以由通用的 `CPACK_...` 来指定，而不是由特定生成器变量来指定。建议项目开发人员查阅CPackFreeBSD模块文档，以了解可用的特性和限制。

## 26.4.10. Cygwin

基本的包生成器是针对Cygwin的。它本质上是一个BZip2打包文件的包装器，除了通用变量之外不提供任何配置。项目可能只希望使用一种简单的打包格式。

## 26.4.11. NuGet

CMake 3.12中增加了对NuGet包格式的支持。新模块支持许多与其他包生成器类似的选项，这些选项都在模块文档中列出。这些选项大部分能自解释，它们遵循与上面讨论的其他生成器模式。开发人员应该意识到，由于这个包生成器是CMake的新版本添加的，所以还没有进行广泛的测试，所以建议注意以后的CMake版本的补丁和更新。

# 26.5. 总结

关于打包要做的第一个决定是项目为发行版提供哪种包格式。一个好的出发点是考虑为每个目标平台提供至少一种简单的打包格式和一种本机格式。当终端用户希望同时安装产品的多个版本时，打包格式非常方便，可以将发布打包解压到不同的目录中。只要包是完全可重定位的，这是一个简单而有效的策略。为了获得最广泛的兼容性，建议对Windows使用ZIP打包，对基于unix的系统使用TGZ打包。

非打包格式取决于目标平台。如果UI安装程序适用于所有平台，考虑使用IFW生成器来获得一致的用户体验。这些安装程序还提供了最大的可定制性、本地化和可下载组件选项。如果本机安装程序是首选，对于Windows来说，WIX或NSIS都挺好，而且功能相似；对于Mac，多组件项目可能更适合使用productbuild生成器来获得干净的安装体验，但对于非组件项目，用户更倾向于使用DragNDrop生成器，因为提供了更多的简单性和灵活性；Linux上，如果不使用IFW生成器实现跨平台一致性，可以考虑同时提供RPM和DEB包以供用户使用。

要特别考虑用户是否应该能够在“无头系统”(无界面显示的系统)上安装产品。这直接影响包格式的选择，以及需要定义和打包组件的方式。对于“无头系统”，非UI安装方法必须可用，包不应该需要与UI有依赖，这样UI组件需要与非UI组件分开。这对于RPM和DEB包格式尤其重要，因为包之间的依赖关系通常由包管理器强制执行，所以需要UI依赖关系的组件包可能会为“无头系统”安装大量不需要的UI相关包。

定义组件名称时，考虑到项目可作为更大项目的子项目。在组件名称中包含项目名称，以防止项目之间的名称冲突。UI安装程序中显示组件名、包文件名等可以设置为不同的名称，而不是依赖于CMake项目内部使用的组件名。事实上，鼓励为组件设置自定义的显示名称和描述，包括为包格式提供本地化支持。

设置组件细节时，最好使用相关CMake模块定义的命令，而不是直接设置变量。诸如 `cpack_add_component()`、`cpack_add_component_group()` 等命令使用了命名参数，这使得设置各种选项容易阅读和维护。因为参数名称中的任何错误都会捕获，而如果变量名称拼写错误，则直接忽略掉

为各种生成器配置详细信息时，可能有大量的变量会影响打包的方式。许多情况下，默认值可以接受，但项目应该设置一些细节。项目应该显式地设置的变量有三个  
个 `CPACK_PACKAGE_VERSION_MAJOR`、`CPACK_PACKAGE_VERSION_MINOR` 和 `CPACK_PACKAGE_VERSION_PATCH`，因为默认的版本信息很少适合或者可能并不总是可靠的。包名、描述和供应商信息也应该设置。为了确保生成的输入文件中的变量值进行转义，应该显式地将 `CPACK_VERBATIM_VARIABLES` 设置为true。

大多数情况下，项目都希望避免在默认安装目录名称中包含版本号。许多安装程序都支持更新现有的安装，因此在产品升级之后，目录名中不适合放置版本号。用户可能喜欢目录名在升级时保持不变，这样就可以编写跨版本的包装器脚本、启动器等。简单的压缩包是例外，这就是为什么非组件打包生成的默认行为大多遵循常规，即将提取的内容放在适当的子目录下，该子目录包括包名和版本。对于基于组件的包，项目需要将 `CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY` 设置为true以获得类似的行为。

RPM和DEB包应该倾向于将包文件名设置为RPM-DEFAULT和DEB-DEFAULT。这确保了包文件名遵循通用的命名约定，而且将包版本和体系结构信息合并到包文件名中是一种更简单的方法。不要依赖于CPack提供的默认RPM或DEB包文件名，因为它们忽略了版本和架构信息。

使用RPM生成器时应该为发布包保留调试信息，请考虑使用debuginfo功能，而不是阻止包创建的剥离步骤。防止剥离需要禁用包生成的其他需求，并需要将调试信息作为发布包的一部分。debuginfo功能允许提供适当的发布包，包括在单独的包中捕获的调试信息，这些包可以分发给用户，也可以不分发。

如果需要从单个CPack调用生成多架构或调试和发布包，请使用 `CPACK_INSTALL_CMAKE_PROJECTS` 来合并来自多个构建树的组件。使用这样的安排时，总是最后安装发布组件，以避免调试和发布组件安装到相同的文件名和目录中。

探索和理解项目支持的每个UI安装程序提供的定制选项。强烈建议定义产品图标，以确保专业的外观。项目还应该提供自述文件、欢迎信息和许可详情，这样安装程序或包的元数据就可以不用使用CPack的占位符了。

# 第27章：扩展

对于中等复杂度的项目，可能依赖于一个或多个外部依赖项。这些依赖通常是工具包，如zlib、OpenSSL、Boost等，以及由同一组织或作为资源、测试数据等使用的私有项目。某些情况下，项目可以期望操作系统提供必要依赖。例如，项目是作为操作系统的一部分，那么是合适的。对于独立项目，应该控制其依赖项的确切版本，以确保构建可重复，并且发布包有已知的来源。当在持续集成系统上构建时，这一点尤其重要。

CMake为外部内容引入构建提供了一些选择。`file(DOWNLOAD)` 可以用于检索特定文件，可以在配置阶段使用，也可以作为脚本模式(即CMake -P)进行。但这个命令通常还不够集成整个项目所需的功能级别。为了下载和构建完整的依赖关系，CMake中的传统方法是使用ExternalProject模块。这是CMake的一部分，除了简单地下载和构建之外，还有很多用途。CMake 3.11中添加的FetchContent模块构建在ExternalProject之上，开启了各种新的用例，包括处理项目之间共享的依赖关系，以及在一个构建中支持整个项目层次结构。

ExternalData模块为在构建时处理外部内容提供了另一种选择，它会关注测试用例使用的数据。

## 27.1. ExternalProject

ExternalProject模块的主要目的是支持下载和构建外部项目，这些外部项目不能直接成为主项目的一部分。外部项目可以添加自己独立的子构建，有效地从主项目中分离出来，并当作黑盒处理。这意味着可以用于不同的体系结构、不同的构建项目，甚至可以使用CMake以外的系统构建项目。还可以用于处理定义目标或安装与主项目冲突的组件。

ExternalProject通过在主要项目中定义一组构建目标来工作，这些目标代表了获取和构建外部项目的不同阶段。然后在执行序列时，CMake会使用相应的目标收集这些信息。时间戳用于跟踪已执行阶段，并且不需要重复，除非相关信息发生更改。默认的阶段设置如下：

### Download

可以使用多种方法获取外部项目。这包括从URL下载打包文件，并自动解压缩它，或者从源代码存储库(如git、subversion、mercurial或CVS)复制/签出。另外，如果支持的下载选项都不合适，项目也可以定义自己的命令。

### Update/Patch

下载了源码，就可以对其应用补丁(打包文件下载)，或者将其更新(对于源代码存储库)。如果需要，可以提供定制命令覆盖默认行为。

### Configure

如果外部项目使用CMake作为构建系统，这一步在下载的源码上执行CMake。一些信息从主构建中传递过来，使外部CMake项目的配置无缝。对于非CMake的外部项目，可以提供自定义命令来运行等效的步骤，比如运行配置脚本和适当的选项。

### Build

默认情况下，使用CMake配置构建，则使用与主项目相同的构建工具构建已配置的外部项目。可以为构建阶段提供定制命令，以使用不同的构建工具或执行其他任务。

### Install

外部项目可以安装到本地目录中，通常安装到主项目构建树中的某个位置。然后，主项目知道外部项目的构建件在哪里，可以将它们合并到自己的构建中。默认行为取决于配置阶段是否使用CMake构建。

## Test

外部项目可能带有自己的测试，主项目可能希望或不希望运行这些测试。`ExternalProject`模块在是否运行测试阶段(默认情况下不运行)，并且测试应该在安装阶段之前还是之后运行也可以选择。如果启用测试阶段，默认的测试目标将假定存在于外部项目中，也可以指定自定义命令来对测试阶段进行控制。

该模块允许自定义阶段，并插入到上述工作流中，但是默认的阶段集对于大多数项目来说已经够用。默认阶段的详细信息都由模块提供的主函数 `ExternalProject_Add()` 设置。这个函数接受许多选项，在模块的文档中有详细说明。下面给出了一些常用的场景和一些典型的场景，以指导读者如何充分利用`ExternalProject`提供的功能。

### 27.1.1. 主要特性

最简单的情况包括从 `URL` 下载源归档文件，并将其构建为CMake项目。实现这一点所需的信息就是 `URL`，如下所示：

```
include(ExternalProject)
ExternalProject_Add(someExtProj
    URL http://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

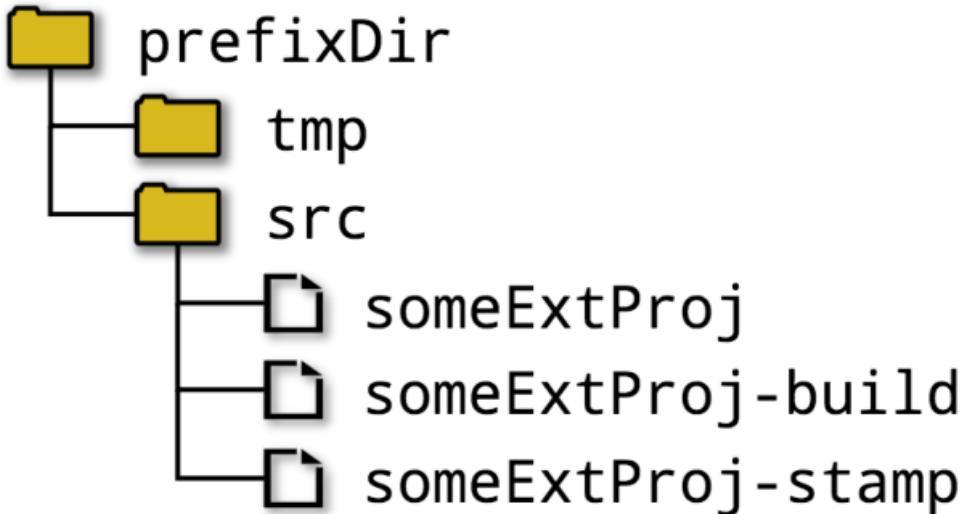
该函数的第一个参数要在主项目中创建构建目标的名称。此目标将用于记录引用外部项目的整个构建过程。默认情况下，会添加到主项目的`all`目标中，也可以通过添加 `EXCLUDE_FROM_ALL` 选项来禁用，该选项与 `add_executable()`、`add_custom_target()` 具有相同的效果。上面的例子中，构建 `someExtProj` 目标将导致在主项目的构建阶段执行以下操作：

- 下载压缩包并解压缩它。
- 基于主版本使用默认选项运行`cmake`。
- 为默认目标与主项目相同的构建工具。
- 构建外部项目的安装目标。

这些步骤都会在构建目录中创建的一组单独的目录来保存源文件、构建输出、时间戳和与外部项目构建相关的其他临时文件。这些目录的结构取决于几个不同的因素，模块文档提供了如何选择目录结构的详细说明。更简单的是展示主项目如何控制位置，而不依赖于默认值。可以使用 `PREFIX` 选项设置目录的基本位置。

```
ExternalProject_Add(someExtProj
    PREFIX prefixDir
    URL http://somecompany.com/releases/myproj_1.2.3.tar.gz
)
```

使用这种方式时，目录布局将基于 `prefixDir`，通常应该使用绝对路径，并且位于主项目的构建区域，这个位置下创建的默认相对目录布局如下所示。解压缩的归档文件将放在 `prefixDir/src/someExtProj` 中，CMake构建将使用 `prefixDir/src/someExtProj-build` 作为构建目录。



可以设置 `EP_PREFIX` 和 `EP_BASE` 目录属性来修改上述布局，前缀和这些目录属性只提供对目录结构的粗略控制。对于那些需要的情况，`ExternalProject_Add()` 允许直接设置部分或所有目录：

```

ExternalProject_Add(someExtProj
    DOWNLOAD_DIR downloadDir
    SOURCE_DIR sourceDir
    BINARY_DIR binaryDir
    INSTALL_DIR installDir
    TMP_DIR tmpDir
    STAMP_DIR stampDir
    URL http://somecompany.com/releases/myproj_1.2.3.tar.gz
)

```

实际上，很少使用 `TMP_DIR` 和 `STAMP_DIR`。默认安装位置由外部项目决定，这通常是系统范围的位置，所以可以指定 `INSTALL_DIR` 来收集所有外部项目的构建目录(要使外部项目使用指定的 `INSTALL_DIR`，还需要执行进一步的设置，稍后的示例将会展示)。

另一个是提供 `SOURCE_DIR`，并给出现有目录的位置。当使用这种方式时，不需要提供下载方法，命令将简单地使用指定源目录的现有内容。这是为不同平台构建主项目源码的一种方法。例如：

```

ExternalProject_Add(firmware
    SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    #... other options to configure differently
)

```

当外部项目也使用CMake作为其构建系统时，可以添加CMake命令行选项来修改配置。实现这一点的最直接方法是使用 `CMAKE_ARGS` 选项，后面要传递给外部项目的cmake命令的参数。上面的例子可以扩展到工具链文件中，配置版本构建和使用指定的安装目录，像这样：

```

ExternalProject_Add(firmware
    SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
    -D CMAKE_BUILD_TYPE=Release
    -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR> # See further below
)

```

如果需要设置多个CMake选项，则生成的CMake命令的长度可能会成为问题。另一种方法是使用 `CMAKE_CACHE_ARGS` 指定缓存变量，而不是通过 `CMAKE_ARGS` 定义。这些参数的形式应该是 `-Dvariable:TYPE=value`，并转换为这样的方式 `set(variable value CACHE TYPE "" FORCE)`。然后使用 `-c` 将该文件传递给 `cmake` 命令行。其效果与通过 `-D` 在 `cmake` 命令行上直接设置变量是一样的。还有其他选项可以用来更改 CMake 生成器，以及使用 CMake 不太常见的功能。可以参阅模块文档以了解更多信息。

如果外部项目不使用 CMake 作为构建系统，可以给出 `CONFIGURE_COMMAND` 来提供可执行的自定义命令，而不是运行 CMake。例如，许多项目提供一个配置脚本，可以这样设置：

```
ExternalProject_Add(someAutotoolsProj
  URL someUrl
  CONFIGURE_COMMAND <SOURCE_DIR>/configure
  ...
)
```

`configure` 命令在构建目录中运行，但 `configure` 脚本将在源目录中运行。上面演示了另一种策略，使用默认的结构，而不是显式地定义用于外部项目的目录布局，但是命令的占位符支持提供了源目录的位置。前面的示例还使用了占位符，作为 `CMAKE_INSTALL_PREFIX` 的值传递安装目录。占位符就是用尖括号括起来的特定目录的选项名，最常用的是 `<SOURCE_DIR>`，`<BINARy_DIR>` 和 `<INSTALL_DIR>`。`<DOWNLOAD_DIR>` 可以在 CMake 3.11 或更高版本中使用。占位符的更多信息可以通过模块文档进行了解。

如果没有使用 `CONFIGURE_COMMAND` 选项，则假定该项目使用 CMake 构建，并且外部项目的构建步骤将使用与主项目相同的构建工具。对于这种情况，默认的构建步骤没问题，不需要特殊的处理。当提供 `CONFIGURE_COMMAND` 时，默认构建工具假定为 `make`。如果构建非默认目标，或者需要构建工具而不是 `make`，必须提供自定义构建命令。例如：

```
find_program(MAKE_EXECUTABLE NAMES nmake gmake make)
ExternalProject_Add(someAutotoolsProj
  URL someUrl
  CONFIGURE_COMMAND <SOURCE_DIR>/configure
  BUILD_COMMAND ${MAKE_EXECUTABLE} specialTool
)
```

自定义构建命令可以做任何事情，甚至可以设置为空字符串，以有效地绕过构建阶段。可以预见的是，同样的模式在安装阶段也可行。对于 CMake 项目，主项目的构建工具用来构建默认名为 `install` 的目标，而对于非 CMake 项目，默认命令是简单的 `make install`。`INSTALL_COMMAND` 可用于自定义安装命令，也可以将其设置为空字符串，以禁用安装。当主项目不需要安装时，通常会使用这种方法。

```
ExternalProject_Add(someAutotoolsProj
  URL someUrl
  CONFIGURE_COMMAND <SOURCE_DIR>/configure
  BUILD_COMMAND ${MAKE_EXECUTABLE} specialTool
  INSTALL_COMMAND "" # Effectively disable the install stage
)
```

注意正确的处理安装阶段。如果外部项目使用 CMake 作为其构建系统，则默认安装规则的目的地由 `CMAKE_INSTALL_PREFIX` 缓存变量控制。如果没有设置此变量，则将使用默认位置，这会导致将外部项目安装到系统目录中，而这通常不是期望的结果（如果项目是在持续集成系统中构建的，则肯定不是）。类似地，如果外部项目使用 CMake 以外的构建系统，则默认安装命令将是 `make install`，也可能尝试将目标安装到系统目录。对于 CMake，通过 `CMAKE_ARGS` 设置缓存变量（如前面示例中所示）就可以解决这种情况。而对于基于 `Makefile` 的项目，下面这样的设置通常是可用的：

```
ExternalProject_Add(otherProj
    URL ...
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/otherProj-install
    CONFIGURE_COMMAND <SOURCE_DIR>/configure
    INSTALL_COMMAND ${MAKE_EXECUTABLE} DESTDIR=<INSTALL_DIR> install
)
```

`INSTALL_DIR` 选项除了为 `<INSTALL_DIR>` 占位符定义值外，不做任何事情。调用者可以使用 `<INSTALL_DIR>` 占位符将信息传递到需要的地方。项目应该使用 `INSTALL_DIR` 定义位置，然后使用 `<INSTALL_DIR>` 占位符，而不是直接将路径嵌入到 `INSTALL_COMMAND` 选项中。

测试阶段的处理方式略有不同，默认情况下不执行任何操作。必须给出特定于测试的选项进行启用，比如 `TEST_BEFORE_INSTALL YES` 或 `TEST_AFTER_INSTALL YES`。启用后与构建和安装阶段相同，使用适当的构建工具调用测试目标，也可以提供 `TEST_COMMAND` 来替代。

当然，`ExternalProject` 提供的下载支持远不止 `URL`。对于压缩包，支持主项目给出要下载文件的哈希值。这不仅有验证下载内容的优势，而且还允许模块检查已下载过的文件，如果知道正确哈希值的文件，就可以避免再次重新下载。哈希值可以是 `file()` 支持的算法，`MD5` 或 `SHA1` 更为常见。哈希值用 `URL_HASH` 给出，如下面的例子所示：

```
ExternalProject_Add(someAutotoolsProj
    URL someUrl
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

强烈建议指定哈希值。如果使用的 `URL` 选项没有附带 `URL_HASH` 选项，`CMake` 将发出警告（作为与旧版本 `CMake` 保持向后兼容性的特殊情况，可以使用 `URL_MD5` 选项来提供 `MD5` 值，不过项目应该避免使用 `URL_MD5`，而使用更灵活的 `URL_HASH`）。

还可以指定多个 `URL`，让项目依次尝试每个 `URL`，直到成功为止。当连接的可用服务器因网络连接、VPN 设置等的不同而改变时，或者在可能比较慢的远程服务器之前尝试本地服务器时，这个功能非常有用。该特性不能用于 `file://url`。

```
ExternalProject_Add(someProj
    URL http://mirrors.mycompany.com/releases/someproj-1.2.3.tar.gz
    https://somewhereelse.com/artifacts/someproj-1.2.3.tar.gz
    URL_HASH MD5=b4a78fe5c9f2ef73cd3a6b07e79f2283
    #... other options
)
```

下载打包文件时，根据下载后的文件内容检测打包格式，自动解压文件。如果需要，可以禁用自动解包，并且可以控制下载配置。有关这些不太常见的相关选项，请参阅模块文档。

该模块还可以直接与 `git`、`subversion`、`mercurial` 或 `CVS` 的源代码存储库一起工作，每个选项都要求使用 `<REPOTYPE>_REPOSITORY` 选项来命名存储库，还可以提供其他特定于存储库的选项。

```
ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG 3a281711d1243351190bdee50a40d81694aa630a
)
```

上面的示例显示了克隆git存储库和切换特定提交所需的信息。如果省略了 `GIT_TAG` 选项，则使用默认分支(通常是主分支)上的最新提交。标记或分支的名称也可以通过 `GIT_TAG` (而不是提交哈希值)给定。虽然 `GIT_TAG` 支持这些不同的选择，只有提交哈希值才是明确的。git中分支或标记名称引用的提交会随着时间的推移而移动，因此并不能保证可重复构建。类似地，完全省略 `GIT_TAG` 与提供默认分支的名称相同，因此也不会总是指向同一提交。

对于 `GIT_TAG` 只使用提交哈希值还有另一个原因。因为标签或分支名称可能会随时间变化，所以每次运行 CMake 时，`ExternalProject_Add()` 都需要与远端联系，即使已经克隆并切换了标签或分支。如果不从远程获取，就不能确保标记或分支没有移动。每次重新运行 CMake 时，这种确认的代价可能很大，特别是当项目使用许多外部项目时。如果使用了提交哈希值，那么 `ExternalProject_Add()` 可以确定它是否已经在本地完成了提交，而不需要与远程服务器联系。当成功获取了提交，CMake 的后续运行都不需要连接网络。

可以使用其他选项来定制git行为，包括指定不同的默认远程名称、控制git子模块、浅克隆和任意的git配置选项。请参阅模块文档以了解更多细节。

subversion库类似于git:

```
ExternalProject_Add(myProj
  SVN_REPOSITORY svn+ssh@somecompany.com/svn/myproj/trunk
  SVN_REVISION -r31227
)
```

`SVN_REVISION` 选项指定svn命令行选项，该选项将指定需要切换的提交。通常是用 `-r` 指定修订号，但在技术上可以是任何有效的命令行选项。如果省略 `SVN_REVISION`，则使用最新的修订，但是项目应该提供此选项，以确保构建可重复。`ExternalProject_Add()` 还支持与安全相关的subversion选项，比如对存储库进行身份验证和指定证书信任设置。请参考ExternalProject模块文档，了解这些这些选项的详细信息。

相比之下，对Mercurial和CVS的支持非常基础。对于Mercurial，只能指定存储库和标记，而对于CVS，还需要指定模块:

```
ExternalProject_Add(myProjHg
  HG_REPOSITORY http://somecompany.com/hg/myproj
  HG_TAG dd2ce38a6b8a
)
ExternalProject_Add(myProjCVS
  CVS_REPOSITORY http://somecompany.com/cvs/myproj
  CVS_MODULE someModule
  CVS_TAG -rsomeTag
)
```

`CVS_TAG` 选项类似于 `SVN_REVISION` 选项，因为直接放在cvs命令行上，所以必须包含必需的命令前缀。

### 27.1.2. 步骤管理

有时引用ExternalProject序列中的步骤是有用的，甚至是必要的，比如添加对另CMake目标的依赖，该目标为特定的步骤提供了输入。可以将 `STEP_TARGETS` 选项交给 `ExternalProject_Add()`，为指定的步骤集创建CMake 目标。这些目标具有`mainName-step`的名称，其中`mainName`作为 `ExternalProject_Add()` 的第一个参数给定，`step`是目标的步骤。例如，以下操作将定义myProj-configure和myProj-install目标:

```
ExternalProject_Add(myProj
  GIT_REPOSITORY git@somecompany.com/git/myproj.git
  GIT_TAG 3a281711d1243351190bdee50a40d81694aa630a
  STEP_TARGETS configure install
)
```

为这些步骤目标添加依赖关系时需要多加小心。要使步骤目标依赖于其他CMake目标，项目应该使用模块提供的 `ExternalProject_Add_StepDependencies()` 函数，而不是调用 `add_dependencies()`。该命令的形式如下：

```
ExternalProject_Add_StepDependencies(mainName step otherTarget1...)
```

下面的例子展示了如何使用这个功能，使配置步骤上的例子依赖于可执行的主项目：

```
add_executable(preConfigure ...)
ExternalProject_Add_StepDependencies(myProj configure preConfigure)
```

要使普通的CMake目标依赖于`step`目标，可以使用 `add_dependencies()`：

```
add_executable(postInstall ...)
add_dependencies(postInstall myProj-install)
```

如果外部项目的特定步骤需要依赖于另一个外部项目的步骤，则必须再次使用 `ExternalProject_Add_StepDependencies()`：

```
ExternalProject_Add(earlier
    STEP_TARGETS build
    ...
)
ExternalProject_Add(later
    STEP_TARGETS build
    ...
)
ExternalProject_Add_StepDependencies(later build earlier-build)
```

如果前面定义的测试运行起来很耗时，上面的安排可能很有用，但在并行构建中，后面的项目不需要等待这些测试，只需要构建前面的测试。

当需要为多个外部项目定义相同的`step`目标集，可以通过设置 `EP_STEP_TARGETS` 目录属性将它们设置为默认值。

```
set_property(DIRECTORY PROPERTY EP_STEP_TARGETS build)
ExternalProject_Add(earlier ...)
ExternalProject_Add(later ...)
ExternalProject_Add_StepDependencies(later build earlier-build)
```

对于许多项目来说，这种依赖的粒度在复杂性过高的项目中可能不值得。通过使用 `ExternalProject_Add()` 的 `DEPENDS` 选项，整个外部项目可以依赖于另一个目标，这个操作要简单得多：

```
add_executable(preConfigure ...)
ExternalProject_Add(myProj
    DEPENDS preConfigure
    ...
)
```

`DEPENDS` 选项负责确保所有`step`依赖项都得到正确处理，就像设置更细粒度依赖项时 `ExternalProject_Add_StepDependencies()` 所做的那样。

项目并不局限于默认的步骤，可以创建自定义步骤，并插入到工作流中，并根据需要建立依赖关系。`ExternalProject_Add_Step()` 函数提供了以下功能：

```
ExternalProject_Add_Step(mainName step
[COMMAND command [args...]]
[COMMENT comment]
[WORKING_DIRECTORY dir]
[DEPENDS filesWeDependOn...]
[DEPENDEES stepsWeDependOn...]
[DEPENDERS stepsDependOnUs...]
[BYPRODUCTS byproducts...]
[ALWAYS bool]
[EXCLUDE_FROM_MAIN bool]
[LOG bool]
[USES_TERMINAL bool]
)
```

`COMMAND` 用于定义执行步骤时要采取的操作，类似于为每个默认步骤指定的自定义命令。可以在执行该步骤时提供注释来自定义消息，这样的注释并不总是显示出来，也不是必需的。`WORKING_DIRECTORY` 选项与 `add_custom_target()` 具有相同的含义。

可以通过自定义步骤提供全面的依赖信息。如果命令依赖于特定文件或一组文件，则该使用 `DEPENDS` 选项。对于构建生成的文件，必须由相同目录范围中创建的自定义命令生成。`DEPENDEES` 和 `DEPENDERS` 选项定义了此自定义步骤在外部项目工作流中的位置。必须注意完全指定所有直接依赖项，否则自定义步骤不会按顺序执行。如果自定义步骤生成了外部项目或主项目中其他项目所依赖的文件，也应该使用 `BYPRODUCTS` 选项。如果不这样做，可能会使Ninja生成器缺少构建规则。

将 `ALWAYS` 设置为`true`，可以使自定义步骤显示为过期。项目通常只在没有其他`step`依赖的情况下才这样做，因为任何依赖都认为是过时的，这可能会导致构建要做更多的工作。如果定制`step`仅按需构建，通常需要将 `ALWAYS` 和 `EXCLUDE_FROM_MAIN` 同时设置为`true`。选项 `LOG` 和 `USES_TERMINAL` 将在下一节中讨论。

所有默认步骤都通过从 `ExternalProject_Add()` 内部调用 `ExternalProject_Add_Step()` 创建。项目不能重新定义它们，这意味着定制步骤不能命名为创建文件夹、下载、更新、跳过更新、打补丁、配置、构建、安装或测试。

操作和步骤间的依赖关系由 `ExternalProject_Add_Step()` 定义，但为自定义步骤创建目标时，必须调用 `ExternalProject_Add_StepTargets()` 函数。`ExternalProject_Add()` 在内部调用该函数，为 `STEP_TARGETS` 中的 `step`，或通过 `EP_STEP_TARGETS` 目录属性设置的`step`创建目标。

```
ExternalProject_Add_StepTargets(mainName [NO_DEPENDS] steps...)
```

`NO_DEPENDS` 在大多数情况下不推荐使用(请参阅模块文档中对该选项的详细讨论)。下面的示例演示如何定义包含的自定义步骤，该步骤依赖于构建步骤，但仅在显式请求时执行。

```
ExternalProject_Add_Step(myProj package
COMMAND ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
DEPENDEES build
ALWAYS YES
EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)
```

### 27.1.3. 其他特性

对于默认或自定义步骤，可以指定自定义命令。对于 `ExternalProject_Add()`，相关选项以 `_COMMAND` 结尾，而对于 `ExternalProject_Add_Step()`，选项要提供自定义命令。这两个函数都允许通过在第一个命令后面附加更多选项来给出多个命令。然后按照顺序执行每个命令。

```

ExternalProject_Add(myProj
    CONFIGURE_COMMAND ${CMAKE_COMMAND} -E echo "Starting custom configure"
    COMMAND ./configure
    COMMAND ${CMAKE_COMMAND} -E echo "Custom configure completed"
    BUILD_COMMAND ${CMAKE_COMMAND} -E echo "Starting custom build"
    COMMAND ${MAKE_EXECUTABLE} mySpecialTarget
    COMMAND ${CMAKE_COMMAND} -E echo "Custom build completed"
)
ExternalProject_Add_Step(myProj package
    COMMAND ${CMAKE_COMMAND} -E echo "Starting packaging step"
    COMMAND ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    COMMAND ${CMAKE_COMMAND} -E echo "Packaging completed"
    DEPENDEES build
    ALWAYS YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)

```

命令的另一个特性是访问终端的能力，对需要用户提供密码来访问存储库访问来说非常重要。虽然这并不适合在没有终端可用的情况下进行持续集成，但有时对开发人员很有用。对于默认步骤，通过使用 `ExternalProject_Add()` 的 `USES_TERMINAL_<STEP>` 的选项来控制对终端的访问，其中 `<STEP>` 是大写的步骤名，选项给定的值是 `true` 或 `false`。对于自定义步骤，`ExternalProject_Add_Step()` 的 `USES_TERMINAL` 选项具有相同的效果。如果使用 `git` 或 `subversion` 存储库下载，建议允许下载和更新步骤访问终端。

```

ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG 3a281711d1243351190bdee50a40d81694aa630a
    USES_TERMINAL_DOWNLOAD YES
    USES_TERMINAL_UPDATE YES
)

```

只有在需要时，才允许访问终端。这样做的效果主要与 `Ninja` 生成器相关，其中自定义步骤将放置到控制台任务池中。分配给控制台的所有目标必须串行运行，并在其他任务池中并行运行任务，直到当前控制台作业完成为止。除非绝对必要，否则不要让构建步骤访问终端，这可能对项目的总体构建产生严重的负面影响。

某些情况下，捕获文件各个步骤的输出，而不是将其放到终端(或重定向到的任何位置)会很有用。在存在大量输出的情况下特别有用，这些输出只有在出现错误或其他意外问题时才有意义。要将步骤的输出重定向到文件，可以将 `LOG_<STEP>` 选项设置为 `ExternalProject_Add()`，或将 `LOG` 选项设置为 `ExternalProject_Add_Step()` 为 `true`。然后，终端输出将只显示一条短消息，指示该步骤是否成功，以及日志文件可以在哪里找到，这些日志文件将保存在 `timestamp` 目录中(即 `STAMP_DIR`)。

```

ExternalProject_Add(myProj
    GIT_REPOSITORY git@somecompany.com/git/myproj.git
    GIT_TAG 3a281711d1243351190bdee50a40d81694aa630a
    LOG_BUILD YES
    LOG_TEST YES
)

```

某些情况下，项目想知道是否向 `ExternalProject_Add()` 提供了某个特定选项，或者某个特定选项的值是多少。诸如 `<SOURCE_DIR>` 等占位符涵盖了许多需要在 `ExternalProject_Add()` 中引用的常见场景，但是对于其他情况，模块提供了 `ExternalProject_Get_Property()`。语法明显不同于其他属性检索命令，如 `get_property()`：

```
ExternalProject_Get_Property(mainName propertyName...)
```

没有给出输出变量名，而是创建与要检索的属性名匹配的变量，允许在调用中检索多个属性。

```

ExternalProject_Get_Property(myProj SOURCE_DIR LOG_BUILD)
set(msg "myProj source will be in ${SOURCE_DIR}")
if(LOG_BUILD)
  string(APPEND msg " and its build output will be redirected to log files")
endif()
message(STATUS "${msg}")

```

## 27.1.4. 常见问题

ExternalProject模块既强大又有效，但有时也会导致问题难以跟踪。最常见的问题是在尝试设置多个外部项目时，其中个项目希望使用来自另一个项目的生成输出。通常需要主项目做两件事：

- 指定两个项目之间的依赖关系。
- 向**depend**er项目提供查找到的信息。

通过为依赖项目的配置步骤，并在被依赖项目的主要目标上创建依赖关系，可以很容易完成第一件事。第二件事要求理解依赖项目如何知道被依赖项目的位置。例如，使用 `find_package()`、`find_library()` 等来定位依赖项目，那么设置 `CMAKE_PREFIX_PATH` 就足够了。下面的示例演示了这种方式，将 zlib 和 libpng 构建为外部项目，并安装到相同的目录中。因为 libpng 需要 zlib，所以将公共安装区域赋给 `CMAKE_PREFIX_PATH` 就可以找到 zlib。当 libpng 使用 `CMAKE_PREFIX_PATH` 时，需要保证在 libpng 运行其配置步骤之前安装 zlib。

```

cmake_minimum_required(VERSION 3.0)
project(ExtProjDeps)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

ExternalProject_Add(zlib
  INSTALL_DIR ${installDir}
  URL https://zlib.net/zlib-1.2.11.tar.gz
  URL_HASH SHA256=c3e5e9fdd5004dcb542feda5ee4f0ff0744628baf8ed2dd5d66f8ca1197cb1a1
  CMAKE_ARGS -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(libpng
  INSTALL_DIR ${installDir}
  URL ftp://ftp-osl.osuosl.org/pub/libpng/src/libpng16/libpng-1.6.34.tar.gz
  URL_HASH MD5=03fbc5134830240104e96d3cda648e71
  CMAKE_ARGS -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
  -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR>
)
ExternalProject_Add_StepDependencies(libpng configure zlib)

```

上面的项目除了定义了一组外部项目之外，什么也没做的方式，被称为“超级构建”，这个主题将在下一章进一步讨论。

使用 Ninja 生成器时可能出现的另一个与依赖相关的问题，Ninja 不知道如何构建外部项目时，应该提供的特定文件。下面的示例演示了这种情况：

```

ExternalProject_Add(myProj
    # Relevant options to download and build a library "someLib"
    ...
)

ExternalProject_Get_Property(myProj INSTALL_DIR)

add_library(MyProj::someLib STATIC IMPORTED)

set_target_properties(MyProj::someLib PROPERTIES
    # Platform-specific due to hard-coded library location and file name
    IMPORTED_LOCATION ${INSTALL_DIR}/lib/libsomeLib.a
)

add_dependencies(MyProj::someLib myProj)

```

Ninja生成器会尝试查找`libsomeLib`。第一次构建`myProj`外部项目之前，它还不存在。Ninja会暂停，并出现错误，说明它不知道如何构建缺失的依赖项。其他生成器可能轻松地进行依赖项检查而不抱怨，但不应将其视为对依赖项的正确指定。解决上述问题的一种方法是在 `ExternalProject_Add()` 中添加 `BUILD_BYPRODUCTS` 选项，以指定构建输出(在CMake 3.2或更高版本中可用)。Ninja将得到所有的信息，以及需要满足的依赖关系。

```

ExternalProject_Add(myProj
    BUILD_BYPRODUCTS <INSTALL_DIR>/lib/libsomeLib.a
    # Relevant options to download and build the above library
    ...
)

```

上述情况就是当外部项目与主项目中的目标混合在一起时，出现问题的一个例子。通常需要手工指定CMake通常代表项目处理的平台特定信息(例如库名和位置)。使用ExternalProject时，项目应该考虑超级构建是否更合适，而不是尝试创建自己的构建目标。

其他情况下也会出现依赖问题。请考虑前面的示例，其中ExternalProject用于与主构建不同的工具链来进行的构建。

```

ExternalProject_Add(firmware
    SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
    -D CMAKE_BUILD_TYPE=Release
    -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
)

```

上面的代码将成功构建，并且看起来一切正常。如果开发人员随后对源目录中的源代码进行了更改，主项目将不会重新构建目标。这是因为ExternalProject使用时间戳来记录这些步骤的成功完成，所以除非计算依赖关系的方式发生了变化，否则主项目认为项目仍然是最新。这可以通过使用 `BUILD_ALWAYS` 选项强制构建目标进行构建：

```

ExternalProject_Add(firmware
    SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/firmware
    INSTALL_DIR ${CMAKE_CURRENT_BINARY_DIR}/firmware-artifacts
    CMAKE_ARGS -D CMAKE_TOOLCHAIN_FILE=${CMAKE_CURRENT_LIST_DIR}/fwtoolchain.cmake
    -D CMAKE_BUILD_TYPE=Release
    -D CMAKE_INSTALL_PREFIX=<INSTALL_DIR>
    BUILD_ALWAYS YES
)

```

这将导致在每次构建主项目时，要调用项目的构建工具。如果项目中没有任何更改，则构建步骤将不起作用；如果有更改，则将按照预期重新构建。

## 27.2. FetchContent

`ExternalProject`的优点，也是它的弱点。它允许外部项目构建与主项目完全隔离，因此可以使用不同的工具链、针对不同的平台、使用不同的构建类型，甚至是完全不同的构建系统。这些收益的代价是，项目对外部项目产生的产品一无所知。如果主构建中的任何内容需要引用外部项目的输出，则必须手动向主构建提供该信息。这是使用`CMake`项目要做的事情，以这种方式使用`ExternalProject`可能不太合适。

对于使用`CMake`作为构建系统的外部项目，没必要使用不同设置来构建主项目和外部项目。事实上，更常见的情况是外部项目使用与主项目相同的设置来构建，但是使用`ExternalProject`并不那么容易做到的。更方便的方法是使用`add_subdirectory()`将其添加到主构建中，就好像是主项目源码的一部分。使用`ExternalProject`无法做到这一点，因为源码直到构建时才进行下载。项目可以使用其他策略(如git子模块)来避免这一问题，但也有不足。

`CMake 3.11`中添加了`FetchContent`模块来解决上述问题。在内部使用`ExternalProject`来设置子构建，子构建在配置阶段将下载和更新外部内容。这意味着下载的内容立即可用，因此主项目可以通过`add_subdirectory()`将其导入主构建，并将其作为资源使用。

依赖于许多外部项目的项目中，这些外部项目有时会共享一些依赖关系。没必要多次下载和构建这些公共依赖关系，但是`ExternalProject`本身并没有直接提供处理这种情况的方法。`FetchContent`模块为这个场景提供了解决方案，允许将外部项目的依赖项信息与初始化下载命令分开定义。第一次为给定的依赖项指定下载信息时，会在内部保存，之后的任何尝试都会忽略。当项目要求填充依赖项时，项目会使用保存的信息，并且项目可以简单地重用这些内容，而不是再次下载。这种“首选设置胜出”的方法意味着父项目可以覆盖`add_subdirectory()`拉入的外部子项目的依赖信息。

下面的示例演示了`FetchContent`模块的使用方式：

```
include(FetchContent)
FetchContent_Declare(googletest ①
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG ec44c6c1675c25b9827aacd08c02433ccde7780 # release-1.8.0
)

FetchContent_GetProperties(googletest) ②
if(NOT googletest_POPULATED)
  FetchContent_Populate(googletest)
  add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR}) ③
endif()
```

① 记录从`GoogleTest`获得的信息。如果项目中的其他地方已经这样做了，这里声明的信息将会忽略。

② 只有在项目的其他部分还没有这样做的情况下，填充`GoogleTest`的内容。

③ 为`add_subdirectory()`提供`xxx_SOURCE_DIR` 和 `xxx_BINARY_DIR`。当`xxx_SOURCE_DIR`不指向在当前二进制目录位置时(通常会出现这种情况)，`add_subdirectory()`会要求给出相关的二进制目录。

`FetchContent_Declare()`的第一个参数是依赖项的名称(这个名称大小写不敏感)。名称后面的参数是`ExternalProject_Add()`所支持的任何选项，除了与配置、构建、安装或测试步骤相关的选项。实际上，给出的选项只有定义下载方法的选项，比如上面`GoogleTest`示例中的`git`细节。

`FetchContent_GetProperties()`会检查是否填充了依赖项，还可以检索一些目录信息。该命令的详情如下：

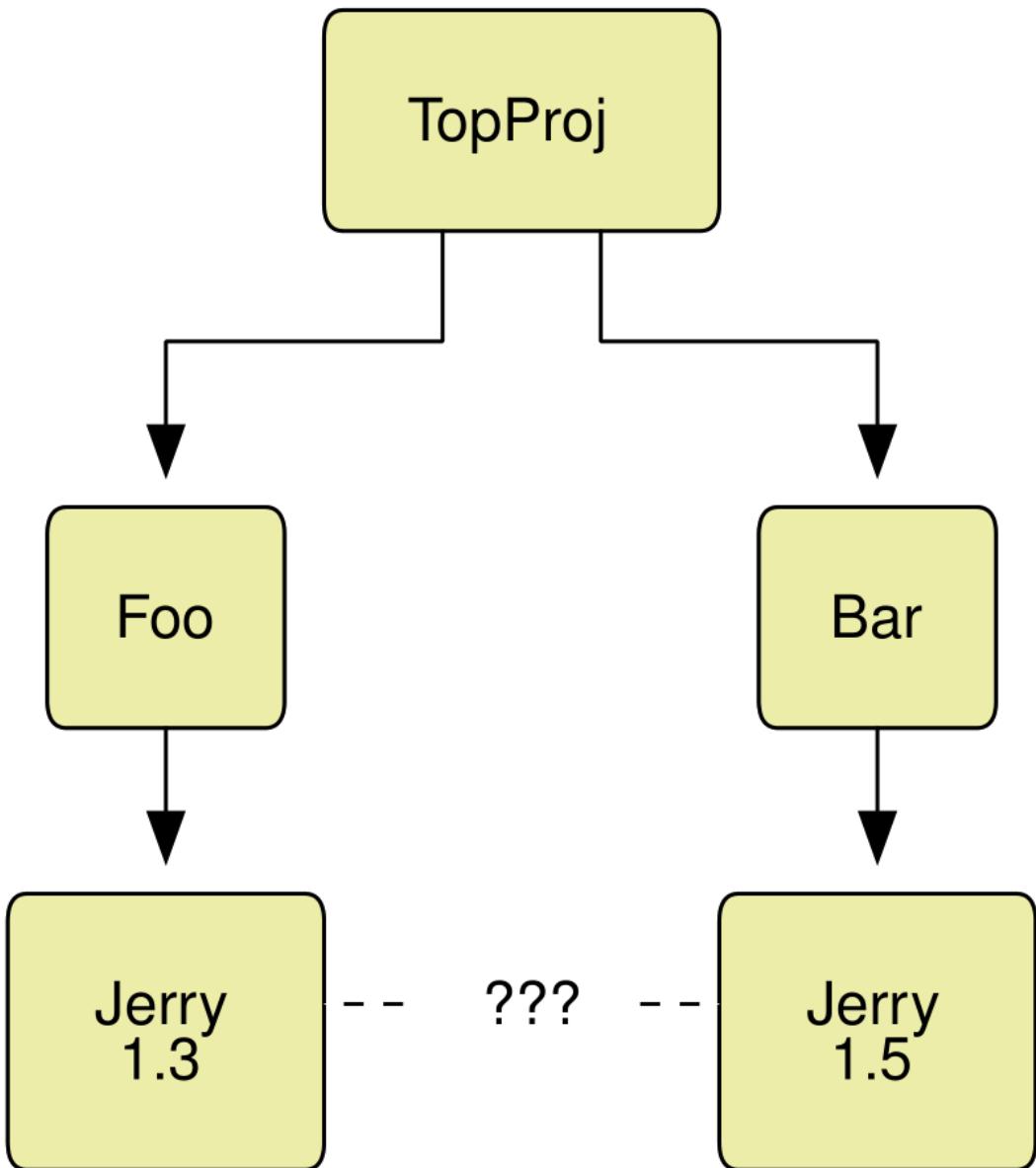
```
FetchContent_GetProperties(name
    [SOURCE_DIR srcDirVar]
    [BINARY_DIR binDirVar]
    [POPULATED doneVar]
)
```

可以使用 `SOURCE_DIR`、`BINARY_DIR` 和填充选项指定变量的名称，其中属性与存储名称依赖项关联。如果这些选项没有给出，命令将在调用作用域中设置变量 `<lcname>_SOURCE_DIR`、`<lcname>_BINARY_DIR` 和 `<lcname>_POPULATED`，其中 `<lcname>` 是转换为小写的名称。如果遵循规范模式，则不需要可选参数。

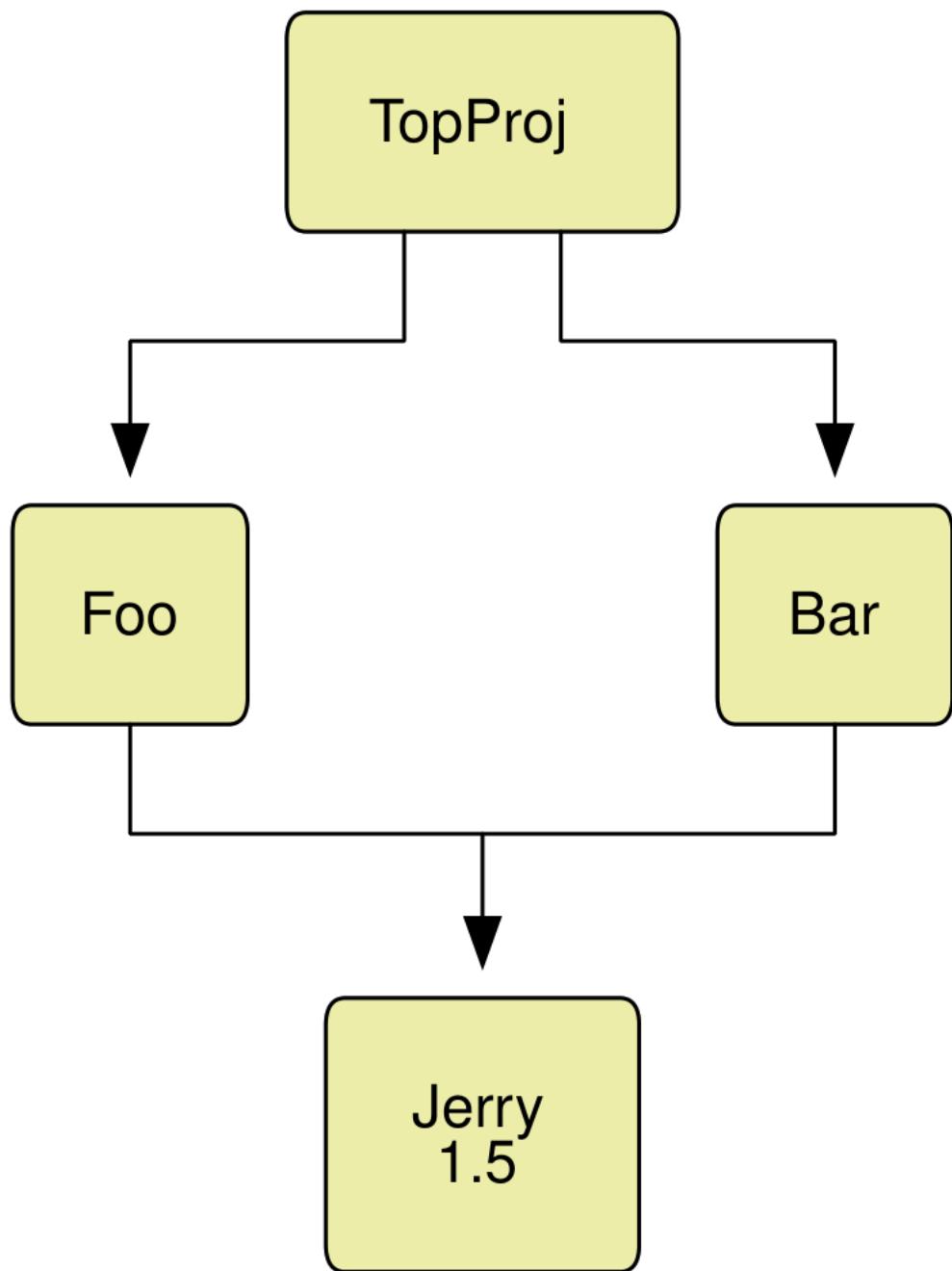
如果 `FetchContent_Populate()` 已经在项目的某个地方调用了，填充的属性将为 `true`，则 `SOURCE_DIR` 属性指定下载内容的位置。由于下载内容可能不是 `FetchContent_Populate()` 调用位置的直接子目录，因此对于 `add_subdirectory()` 的调用，需要使用 `BINARY_DIR` 属性。

如果 `FetchContent_GetProperties()` 确认指定的内容尚未填充，则调用 `FetchContent_Populate()` 来填充内容。当使用上面所示的规范形式作为项目的一部分时，将只接受一个参数，即要填充的依赖项名称。使用前面声明的详细信息，如果之前的 `cmake` 运行还没有填充内容，则填充内容。`<lcname>_POPULATED`，`<lcname>_SOURCE_DIR` 和 `<lcname>_BINARY_DIR` 也将再调用范围内设置，与调用 `FetchContent_GetProperties(name)` 的方式完全相同。

下面的示例展示了 `FetchContent` 模块允许顶层项目覆盖较低级别依赖项，并设置信息的方式。考虑顶层项目 `TopProj`，依赖于外部项目 `Foo` 和 `Bar`。`Foo` 和 `Bar` 都依赖于另一个外部项目 `Jerry`，但是他们想要的版本不同。



只需要下载并构建一个Jerry的副本，**Foo**和**Bar**就可以使用这个副本。当这些项目合并到一个构建中，所选的Jerry版本必须覆盖**Foo**或**Bar**通常使用的版本，或者可能两者都使用。顶级项目负责确保选择了一个有效的版本，这样**Foo**和**Bar**就可以根据它进行构建。这个例子假设**Foo**在自己构建时使用1.3版本，可以安全地使用后续版本。实现的示例如下：



*TopProj CMakeLists.txt*

```

# Declare the direct dependencies
include(FetchContent)
FetchContent_Declare(foo GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_Declare(bar GIT_REPOSITORY ... GIT_TAG ...)

# Override the Jerry dependency to ensure we get what we want
FetchContent_Declare(jerry
  URL https://somecompany.com/releases/jerry-1.5.tar.gz
  URL_HASH ...
)

# Populate the direct dependencies but leave Jerry to be populated by foo
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
  FetchContent_Populate(foo)
  add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()
FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
  FetchContent_Populate(bar)
  add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()

```

#### Foo CMakeLists.txt

```

include(FetchContent)
FetchContent_Declare(jerry
  URL https://somecompany.com/releases/jerry-1.3.tar.gz
  URL_HASH ...
)
FetchContent_GetProperties(jerry)
if(NOT jerry_POPULATED)
  FetchContent_Populate(jerry)
  add_subdirectory(${jerry_SOURCE_DIR} ${jerry_BINARY_DIR})
endif()

```

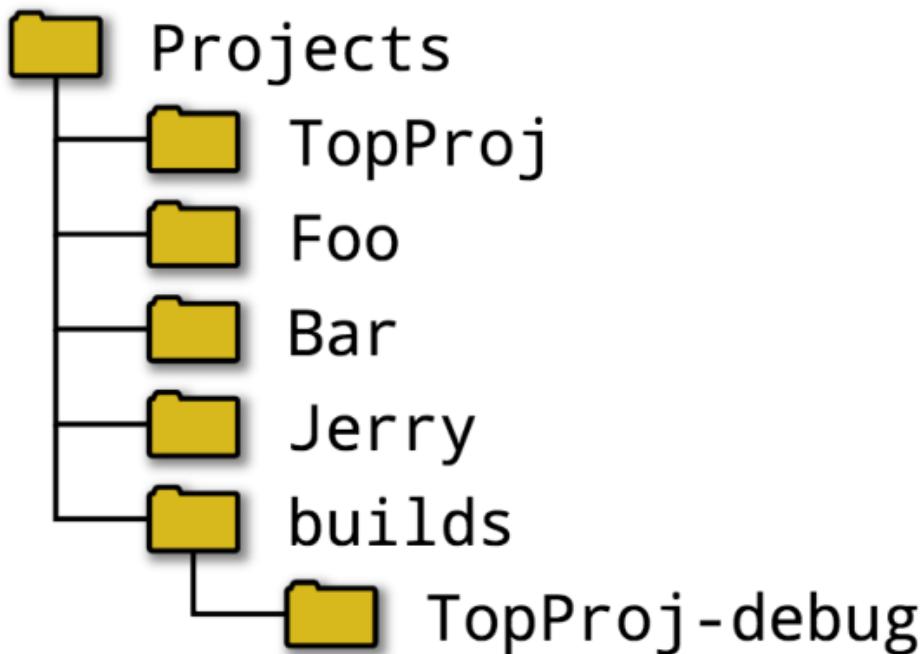
Bar的CMakeLists.txt文件与Foo的相同，只是 URL 指定的是jerry-1.5.tar.gz，而不是jerry-1.3.tar.gz。上面的框架示例允许Foo和Bar独立进行项目构建，或者可以合并到另一个项目中，比如TopProj。

### 27.2.1. 开发人员须知

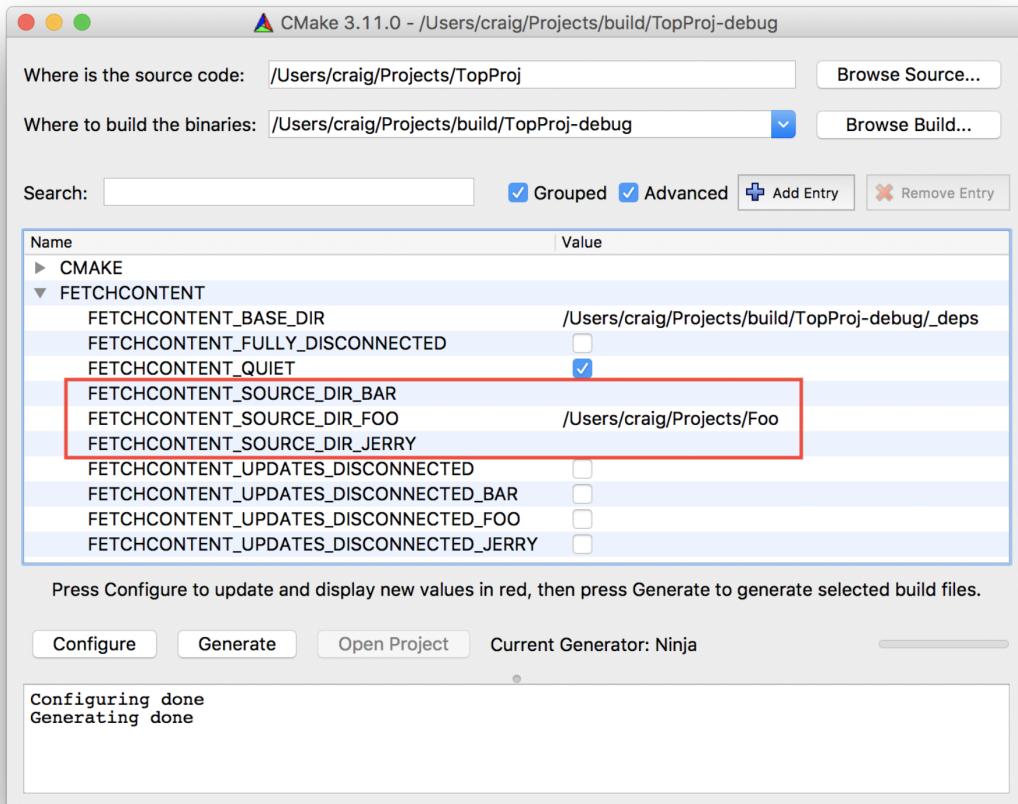
有时开发人员可能想要同时处理多个项目，在主项目及其依赖项中或者跨多个依赖项进行更改。当更改外部项目的某些部分时，开发人员希望使用本地副本，而不是每次都更新下载。FetchContent模块允许使用CMake缓存变量覆盖任何外部依赖项的源目录，为这种操作模式提供了直接支持。这些变量的名称形式为 `FETCHCONTENT_SOURCE_DIR_<DEPNAME>`，其中 `<DEPNAME>` 是依赖项的大写名称。

前面的示例中，可以考虑这样一种情况：开发人员想要对Foo进行更改，看看如何影响主项目的。可以在 mainproject之外创建一个单独的Foo克隆，然后将相应的位置设置到 `FETCHCONTENT_SOURCE_DIR_FOO` 中。TopProj 项目将使用该本地副本的源文件，并且不会以任何方式修改它，它仍然会在TopProj构建区域中为它使用相同的构建目录。唯一的区别是源来自哪里，通过设置 `FETCHCONTENT_SOURCE_DIR_FOO`，开发人员将接管对内容的控制。可以自由地更改本地副本中的任何内容、进行进一步的提交、切换分支或其他任何可能需要的内容，然后重新构建主TopProj项目，而不需要更改TopProj。

适合上述用法的方式是设置共同目录，开发人员可以在该目录下切换想要使用的不同项目。需要时，可以将主项目指向这些本地签出，但是仍然使用默认的下载内容。上面的例子中，这样的方式看起来是这样的：



如果开发人员想对 Foo 做一些更改，使用 TopProj 的构建来测试，可以将 `FETCHCONTENT_SOURCE_DIR_FOO` 设置为 `/../Projects/Foo`，但是 Foo 依赖项的所有构建输出仍然在 `Projects/build/TopProj-debug` 下。如果 `FETCHCONTENT_SOURCE_DIR_BAR` 未设置，Bar 仍然会下载，而不是使用项目 /Bar 中的已下载代码。开发人员可以随时通过设置 `FETCHCONTENT_SOURCE_DIR_BAR` 切换到本地内容。因为相关的缓存变量都共享相同的前缀，所以很容易在 CMake GUI 或 ccmake 工具中找到。这使得查看哪些项目当前正在使用本地副本变得非常简单。



上述场景的显著优势是，特性很好地与IDE集成，如代码重构工具等。IDE可以看到整个项目，包括依赖项，因此使用这些依赖项进行本地切换时，可以透明地跨多个项目执行重构，就像是同一个项目那样容易。即使不使用任何本地切换依赖关系，IDE也有机会来自动完成构建更完整的代码模型、跟随符号等等功能。

### 27.2.2. FetchContent的其他用途

FetchContent支持的不仅仅是下载外部项目的源代码，可以通过`add_subdirectory()`将其添加到主项目中。另一中用法是将CMake模块收集到中央存储库中，并在许多项目中重用。通过这种机制可以引入多个集合，这使得其他项目合并有用的CMake脚本会相对简单，而无需在主项目源中嵌入副本。下面的示例演示了下载外部git存储库，并将其cmake子目录添加到主项目的cmake模块搜索路径的示例。

```
include(FetchContent)
FetchContent_Declare(JoeSmithUtils GIT_REPOSITORY ... GIT_TAG ...)
FetchContent_GetProperties(JoeSmithUtils)
if(NOT joesmithutils_POPULATED)
  FetchContent_Populate(JoeSmithUtils)
  list(APPEND CMAKE_MODULE_PATH ${joesmithutils_SOURCE_DIR}/cmake)
endif()
```

FetchContent模块可以在第一个`project()`之前使用。该特性允许模块提供工具链文件，开发人员可以在主项目中使用这些文件。

```

cmake_minimum_required(VERSION 3.11)

include(FetchContent)
FetchContent_Declare(CompanyXToolchains
    GIT_REPOSITORY ...
    GIT_TAG ...
    SOURCE_DIR ${CMAKE_BINARY_DIR}/toolchains
)

FetchContent_GetProperties(CompanyXToolchains)
if(NOT companyxtoolchains_POPULATED)
    FetchContent_Populate(CompanyXToolchains)
endif()

project(MyProj)

```

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchains/toolchain_betacxx.cmake ...
```

上面的示例中，使用 `SOURCE_DIR` 选项显式覆盖工具链的下载目录。假设 `CompanyXToolchains` 项目没有子目录工具链文件的集合，则位置可以预测，并且易于开发人员使用。如果使用非常特定的工具链，并且希望这些工具链安装在相同的位置，这是一种非常有效的方法，可以帮助整个团队使用通用的构建设置。这项技术甚至可以扩展到下载实际的工具链。

### 27.2.3. 限制

大多数情况下，`FetchContent` 模块具有强大的优势，但也有一些需要注意的限制。主要的限制是 `CMake` 目标名称在合并项目的集合中必须是唯一的，所以如果两个外部项目定义了相同名称的目标，则不能通过 `add_subdirectory()` 同时添加。如果项目遵循的命名约定，使用了特定于项目的前缀或类似的东西，这个限制很容易规避。困难来自于从未想过以这种方式使用的项目，以及使用普通名称的项目。对于那些使用特定于项目目标名称的项目，可以使用 `OUTPUT_NAME` 目标属性控制所创建的二进制文件的名称。例如：

```

add_library(BagOfBeans_varieties ...)
set_target_properties(BagOfBeans_varieties PROPERTIES
    OUTPUT_NAME beantypes
)

add_executable(BagOfBeans_planter )
set_target_properties(BagOfBeans_planter PROPERTIES
    OUTPUT_NAME planter
)

```

`OUTPUT_NAME` 属性和其他相关属性将在 28.5.2 节中更详细地介绍。

安装组件也类似，但不那么严格的限制。每个项目使用特定于项目的前缀或相同的名称来命名它们的安装组件，这允许父项目可以挑选出想要包含的组件。如果两个或多个外部项目依赖项使用相同的安装组件名称，父项目则不能区分。这是否重要取决于具体情况，但是通过确保项目对其安装组件使用良好的命名约定，可以很容易地避免该情况的发生。

通过 `add_subdirectory()` 将外部依赖项吸收到更大的父构建中的做法不是很普遍，许多项目从来没有考虑过，并且遇到使项目难以合并也很常见。常见的例子是，项目假设是顶层项目，在使用像 `CMAKE_SOURCE_DIR` 和 `CMAKE_BINARY_DIR` 这样的变量地方，使用 `CMAKE_CURRENT_SOURCE_DIR` 和 `CMAKE_CURRENT_BINARY_DIR` 替代可能更合适。这样问题通常很容易修复，但需要对项目进行写访问，让项目维护人员接受变更，或者保留项目副本，在副本中进行相关修复或实行类似的措施。

## 27.3. ExternalData

另一个名为**ExternalData**的模块提供了在构建时下载的文件的方法，此模块的重点是在构建特定目标时下载测试数据。类似于**ExternalProject**的工作方式，但两个模块定义要下载内容的方式有很大不同。**ExternalProject**模块允许显式定义下载信息，并支持多种方法。**ExternalData**模块采用一种不同的方法，单个文件在一组项目定义的基本 URL 位置下可用，路径和文件名使用特定的哈希算法编码。实际的文件在项目的源中由同名的占位符文件表示，除非附加了哈希算法的名称作为文件名后缀。该模块提供了一个函数，用于将特殊的字符串参数转换为最终的下载位置和名称，以及 `add_test()` 函数的包装器，以便更容易地将解析后的位置传递给测试。

实践中，为**ExternalData**设置必要的支持所涉及的步骤往往会降低其吸引力。要从其中下载数据的服务器必须分别处理每个文件，每次添加新文件或更新现有文件时，都必须手动对其进行哈希计算，并将其上传到与该哈希值匹配的路径和文件名中。如果文件很大，但是与前一次迭代只有很小的区别，仍需要完全复制该文件。相比之下，**ExternalProject**模块可以通过基于存储库的下载方法实现相同的功能，但是所涉及的步骤对于大多数开发人员来说比较简单。选择适当的存储库方法还可以有效地处理大文件中的小更改。

考虑使用**ExternalData**的原因是它支持一系列文件，而不仅仅是单个文件。这更像是处理一系列文件时会出现的场景。即使这样，也可以通过**ExternalProject**和**foreach()**循环实现类似的功能，这可能比**ExternalData**更容易设置。如果项目的测试主要关注时间序列数据或其他类似的顺序数据集，至少评估一下**ExternalData**是否是在构建时按需获取数据的更好方法。请参考模块文档以获得更多信息，或获得更实用的介绍，与本书相同的站点上提供的关于这个主题的[文章](#)，可能会对使用者有所帮助。

## 27.4. 总结

**ExternalProject**和**FetchContent**提供了将外部内容合并到父项目中的方法。**ExternalProject**很适合引入成熟的外部项目，这些项目具有良好的打包功能，并且提供了定义良好的配置文件，`find_package()` 可以使用这些配置文件导入相关的目标。它还有一个优点，即只有在构建需要(外部依赖项)时才下载它们，而且下载可以与其他构建任务并行完成。当开发人员需要跨多个项目进行工作并进行更改时，就不太方便了，特别是涉及到少量的重构。由于**ExternalProject**是CMake的一部分，网上也有现成的资料，但而已经经常会看到开发人员为如何使用它进行挣扎。常见的缺点是在特定平台中硬编码库的路径和文件名，由于在主项目中混合了**ExternalProject**和其他目标，而不是传统的超级构建。选择使用外部项目前，仔细考虑外部依赖打包的成熟度和质量，以及主项目是否可以使用超级构建。如果主项目不能转化为超级构建，就不要使用它。

**FetchContent**模块是一个很好的选择，将其他项目添加到构建中，允许同时处理。为开发人员提供了跨项目的自由，可以临时切换到本地修改、更改分支、使用不同的发布版本和各种其他用例进行无缝测试。对IDE工具也很友好，因为整个构建看起来就像单独的项目，所以像代码完成之类的东西通常提供了更好的洞察力，可能比单独加载项目更可靠。如果将依赖关系添加到成熟项目中，**FetchContent**的破坏性要比**ExternalProject**小得多，因为不需要对主项目进行任何重组。还非常适合于合并相对不成熟的、还没有实现安装组件和打包的外部项目。**FetchContent**的另一个优点是，会在整个项目层次结构中使用相同的编译器和设置。如果可以接受3.11或更高的CMake为最低版本，请考虑**FetchContent**是否比**ExternalProject**更方便、更自然地适合这个项目。强烈建议开发者熟悉ccache之类的工具，以加快构建速度，这样使用**FetchContent**的好处更加明显。

无论使用**ExternalProject**还是**FetchContent**，如果下载信息是为git存储库定义，最好将 `GIT_TAG` 设置为提交哈希值，而不是分支或标记名称。这样做效率更高，因为如果本地克隆已经有了提交，可以避免建立任何网络连接。

如果项目想按需下载测试数据，请检查**ExternalData**模块是否是合适的选择。**ExternalProject**模块可能使用起来更简单，也更容易被大多数开发人员理解，但是在特定的情况下，比如处理文件序列，**ExternalData**可能会更简单。如果有疑问，请选择**ExternalProject**，因为它的接口更简单，并且可以有效地处理对大数据集的小更改。

处理项目时，考虑总是假设有一天会作为其他项目的子项目包含，这为将来如何使用项目提供了最大的灵活性。常见的问题包括：

- 不要假设这个项目是最高级别的项目。使用像 `CMAKE_CURRENT_SOURCE_DIR` 和 `CMAKE_CURRENT_BINARY_DIR` 这样的变量来引用相对于项目自身目录结构的位置，而不是 `CMAKE_SOURCE_DIR` 和 `CMAKE_BINARY_DIR`。
- 使用特定于项目的目标名称。避免使用通用名称，即使是内部实用目标，因为CMake需要在整个项目层次结构中对所有非全局导入目标使用全局唯一的目标名称。
- 使用特定于项目的安装组件名称，避免通用名称。
- 最好在项目定义的安装组件集中提供多粒度选择，以便父项目可以选择想要安装的部件。考虑整个或部分部署项目的不同方式，并确保安装组件可以捕获不同组合。
- 如果别名目标可用，那么使用目标的名称空间别名链接(例如，更倾向链接到 `MyProj::mpfoo` 而不是只链接 `mpfoo`)。这允许项目在ExternalProject和FetchContent场景中使用。
- 避免强制设置缓存变量。不使用常规的CMake变量来覆盖当前作用域，替代方案是使用目标或目录属性。

# 第28章：项目结构

项目结构有效的因素多种多样。对一个项目有效的东西可能对另一个项目无效，但是有些比较统一。在项目生命周期的早期，选择灵活但可预测的目录结构，使其能够以最小的摩擦和重组进行发展。

最重要的决策是，项目应该构建为超级构建还是常规项目。两者是不同的，各有优缺点。决定很大程度上取决于项目想要如何对待依赖关系，以及是直接吸收它们，还是隔离在子构建中。对于那些没有任何依赖关系的项目，常规项目是显而易见的选择。当存在依赖关系时，可能需要在正确的项目结构和顺利的构建间做权衡。

邮件列表、问题跟踪器和Q&A站点中最常见的主题是，试图使用一种项目结构，却期望具有另一种结构的功能，从而产生问题。出现这种情况是因为项目以特定的结构开始，但随着依赖项的添加，该结构不再支持开发人员希望项目实现的功能。参与其中的人已经习惯了在现有的结构下工作，所以改变它可能造成非常混乱的局面，而且经常会遇到相当大的阻力。项目越老，这样的改变可能就越难。因此，需要在项目生命周期的早期处理依赖关系，并适当考虑未来的期望。

## 28.1. 超级构建结构

当依赖项不使用CMake作为构建系统时，超级构建往往是首选结构。将每个依赖项视为单独构建，主项目将指导整个序列，以及从一个依赖项的构建传递到另一个依赖项的方式。每个单独的构建都使用ExternalProject添加到主构建中，这样允许CMake查看每个构建产生的内容，并自动检测可以传递给其他依赖项的信息，从而避免在主构建中手动硬编码这些信息。即使所有的依赖项都使用CMake，超级构建仍然是首选，比如为了避免目标名称冲突或假设总是顶层项目的问题。

超级构建允许对独立依赖构建的顺序进行精确控制。例如，其他依赖项运行自己的配置阶段前，可能需要一个或多个依赖项完全完成构建，包括安装步骤。对于这样的示例，后面的配置步骤可以查看已安装的组件，并自动计算出适当的文件名、位置等，这在常规构建中是不可能的。

超级构建可以使用顶层CMakeLists.txt文件实现，该文件遵循可预测的模式。一种为所有依赖项使用公共安装区域，而另一种是将每个依赖项安装到自己的安装区域。两者相似，所以使用通用安装区域的定义会简单一些：

```
cmake_minimum_required(VERSION 3.0)
project(SuperbuildExample)
include(ExternalProject)

set(installDir ${CMAKE_CURRENT_BINARY_DIR}/install)

ExternalProject_Add(someDep1 ①
...
  INSTALL_DIR ${installDir}
  CMAKE_ARGS -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
)
ExternalProject_Add(someDep2
...
  INSTALL_DIR ${installDir}
  CMAKE_ARGS -DCMAKE_INSTALL_PREFIX:PATH=<INSTALL_DIR>
  -DCMAKE_PREFIX_PATH:PATH=<INSTALL_DIR> ②
)
ExternalProject_Add_StepDependencies(someDep2 configure someDep1) ③
```

① 至少有一个依赖项不需要其他依赖项。

② 使用 `find_package()` 定位其依赖关系，通常只需将 `CMAKE_PREFIX_PATH` 设置为通用安装目录即可。

③ 添加步骤依赖项，以确保配置步骤只在安装了其他必需的依赖项之后运行。

如果每个依赖项都安装到自己的安装区域，与上面的唯一区别是，为后面的依赖项提供的 `CMAKE_PREFIX_PATH` 可能需要是之前所有依赖项安装的目录列表，而不仅仅是公共的安装目录。

如果依赖项不使用CMake作为构建系统，那么整体结构不会改变，只会改变依赖项构建的方式。例如，使用像autotools这样的构建系统的依赖项可能会这样指定：

```
ExternalProject_Add(someDep3
    INSTALL_DIR ${installDir}
    CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix <INSTALL_DIR>
    ...
)
```

可能还需要将其他选项传递给配置脚本，以更具体的方式告诉它在哪里找到依赖项，显然会根据依赖项的配置功能而有所不同。

超级构建中打包就比较困难了。每个依赖项都可以控制自己的打包，因此顶层项目最终不打包。相反，如果确实需要打包，可能会给 `ExternalProject_Add()` 调用一个或多个自定义打包步骤。上一章演示了如何使用 `ExternalProject_Add_Step()` 来实现（类似的方法也可以用于非cmake的子项目）：

```
ExternalProject_Add_Step(myProj package
    COMMAND ${CMAKE_COMMAND} --build <BINARY_DIR> --target package
    DEPENDS build
    ALWAYS YES
    EXCLUDE_FROM_MAIN YES
)
ExternalProject_Add_StepTargets(myProj package)
```

超级构建所做的一切只是将外部项目组合在一起时，可以很好地工作。它们依赖于所有具有良好安装规则的外部项目，如果知道外部项目的位置，那么每个项目都应该能够找到自己的依赖项。如果这些都做不到，那顶层项目将不可避免地硬编码关于平台特定信息，这时超级构建就不好用了。

## 28.2. 非超级构建结构

如果项目没有依赖项，或者依赖项使用FetchContent或类似于git子模块的机制引入到主构建中，那么一些前瞻性计划将有助于避免以后的困难。真正帮助项目保持易于理解和使用的做法是，其顶层CMakeLists.txt更像一个目录。该结构可分为以下几个部分：

### 序言

包括最基本的设置，比如对 `cmake_minimum_required()` 和 `project()` 的调用。还包括使用FetchContent模块引入工具链文件和CMake助手库。这个部分通常非常短。

### 项目设置

这个高级的部分会做一些事情，比如设置一些全局属性和默认变量，会在CMake缓存中定义构建选项，可能会包含一些逻辑，以解决整个构建的需求。设置默认语言标准、构建类型和各种搜索路径。

### 依赖关系

引入外部依赖关系，以便对项目的其余部分可用。与其在顶层CMakeLists.txt文件中定义它们，不如将它们放在专门的目录中，这样更干净、更健壮。

## 主要构建目标

理想情况下，这个部分应该是由一个或多个 `add_subdirectory()` 组成。

### 测试

虽然单元测试可以作为主源嵌入在相同的目录结构中，但集成测试可以位于主源之外的单独区域中(它们将添加到主构建目标之后)。

### 打包

这通常应该是项目定义的最后一件事，最好在它自己的目录中，以保持顶层的整洁性。

上面重复出现的模式是，除了序言和项目设置外，大多数内容最好在通过 `add_subdirectory()` 添加的子目录中定义。这不仅使顶层 **CMakeLists.txt** 文件更易于阅读和理解，而且允许每个子目录关注特定的区域。这有助于简化查找，还意味着可以使用目录作用域，最小化不相关区域的变量暴露给不需要了解它们的人。下面是一个简单的顶层 **CMakeLists.txt** 的例子，遵循了上面的指导意见，看起来像这样：

```
# Preamble
cmake_minimum_required(VERSION 3.1)
project(MyProj)
enable_testing()

# Project wide setup
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR}/cmake)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED YES)
set(CMAKE_CXX_EXTENSIONS NO)

# Externally provided content
add_subdirectory(dependencies)

# Main targets built by this project
add_subdirectory(src)

# Things typically only needed if we are the top level project
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
    add_subdirectory(tests)
    add_subdirectory(packaging)
endif()
```

实践中，项目设置可能会包含比上面显示的更多的内容，并且可能会有其他目录用于项目构建(例如，用于文档，添加其他可安装内容，如脚本、图像等)。

如果按照上面关于在子目录中定义大多数内容的建议，项目源的顶部目录只包含大部分管理文件。其中可能包括自述文件、许可信息、贡献说明等等。持续集成系统还经常在顶层目录中查找特定的文件名。将源文件保存在顶层目录之外，可以确保脚本专注于项目的高级描述。

将依赖项的处理委托给子目录，可以实现两件事情。首先，确保任何依赖项都不能看到相同的 `CMAKE_SOURCE_DIR` 和 `CMAKE_CURRENT_SOURCE_DIR`，因此可以通过比较这两个变量来检测是合并到更大的项目结构中，还是独立构建。上面的简单示例显示了不是顶层项目时，如何经常使用此方法来避免定义测试和打包信息。还可以确保用于设置依赖项的非缓存变量，不会意外地泄漏到构建的其他部分。这样做的结果是，作为项目的其他部分利用依赖关系的手段，鼓励使用**CMake**目标，而不是变量。

使用**FetchContent**模块将项目依赖项合并到构建中的示例如下：

*dependencies/CMakeLists.txt*

```

include(FetchContent)

# Declare all the dependency details first in case any dependency wants
# to pull in some of the same ones (this keeps us in control)
FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

# Add each dependency if not already part of the build
FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
  FetchContent_Populate(foo)
  add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()

FetchContent_GetProperties(bar)
if(NOT bar_POPULATED)
  FetchContent_Populate(bar)
  add_subdirectory(${bar_SOURCE_DIR} ${bar_BINARY_DIR})
endif()

```

依赖关系可能需要在 `add_subdirectory()` 之前设置某些变量。这应该在自己的范围内完成，这样就不会影响在同一范围内添加的其他依赖项。因此，把每个依赖填充放到自己的子目录中也是很有用的，这将使上面的例子看起来像这样：

*dependencies/CMakeLists.txt*

```

include(FetchContent)

FetchContent_Declare(jerry ...)
FetchContent_Declare(foo ...)
FetchContent_Declare(bar ...)

add_subdirectory(foo)
add_subdirectory(bar)

```

子目录会看起来像这样：

*dependencies/foo/CMakeLists.txt*

```

FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
  FetchContent_Populate(foo)

  # Add any customizations needed before actually pulling in the dependency.
  # For example, build static libs by default and only build those targets
  # that another target depends on.
  set(BUILD_SHARED_LIBS NO)
  set_directory_properties(PROPERTIES EXCLUDE_FROM_ALL YES)

  # Now add the dependency
  add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
endif()

```

`bar`子目录在结构上类似。以上的示例可以扩展处理预构建的二进制包或源代码包：

```

FetchContent_GetProperties(foo)
if(NOT foo_POPULATED)
FetchContent_Populate(foo)

if(EXISTS ${foo_SOURCE_DIR}/CMakeLists.txt)
# Probably source, but could still be a binary package that
# provides itself through a top level CMakeLists.txt file
add_subdirectory(${foo_SOURCE_DIR} ${foo_BINARY_DIR})
else()
# Must be a binary package, assume it provides a config file in a
# standard location within its directory layout
find_package(foo REQUIRED
NO_DEFAULT_PATH
PATHS ${foo_SOURCE_DIR}
)
# For this to be useful, imported targets must be promoted to global
# so that other parts of the project can access them
set_target_properties(foo::foo PROPERTIES IMPORTED_GLOBAL TRUE)
endif()
endif()

```

`FetchContent`模块和`IMPORTED_GLOBAL`目标属性只在CMake 3.11之后可用。没有这些特性的情况下添加依赖关系会非常困难，需要做出让步，或者放弃添加预构建的二进制包。由于不能将局部目标提升为全局目标，替代方法通常依赖于变量将信息传递回主构建，或者将全局目标定义为本地导入目标。另一种不太理想的方法是直接从顶层CMakeLists.txt文件中添加依赖项，这会使项目难以合并到更大的项目层次结构中。如果不需要支持预构建的二进制包，则不需要`IMPORTED_GLOBAL`目标属性。为了支持3.11之前的CMake，像git子模块或`file(DOWNLOAD)`这样的方式可以作为`FetchContent`的替代品。

其他主项目的顶层子目录中，添加测试和打包并不需要特殊的东西，只需要遵循前面章节中已经介绍过的方式即可。`test`子目录的内容和结构将特定于项目，而打包通常只需要一个CMakeLists.txt文件，可能还需要配置到构建目录中供CPack使用的其他文件，还可能包含包生成器使用的资源。`src`目录的结构是更大的主题，在28.5节中会进一步介绍。

## 28.3. 常见的顶层子目录

上一节提到了一些目录名，通常是在源顶层下面的子目录。经常使用的目录如下：

- `cmake`
- `dependencies`
- `doc`
- `src`
- `tests`
- `packaging`

没有任何其他约定的情况下，鼓励项目使用这些目录名。CMake子目录中收集CMake助手脚本可以很容易地找到它们，允许开发人员浏览该目录的内容，并发现他们可能不知道的有用工具。顶层CMakeLists.txt的项目范围设置中的单个`list(APPEND CMAKE_MODULE_PATH..)`使它们对整个项目可用。`doc`子目录可以方便地收集文档，如果使用像Markdown或Asciidoc这样的格式，并且文件包含彼此之间的相对链接，那么`doc`子目录将非常有用。

有一些项目应该避免的子目录名称，仅使用一个参数调用`add_subdirectory()`将在构建目录中产生一个同名的对应目录。项目应该避免使用可能导致与在构建区域中创建的预定义目录冲突的源目录名称，包括以下内容：

- `Testing`
- `CMakeFiles`

- CMakeScripts
- 任何默认构建类型(即 `CMAKE_CONFIGURATION_TYPES` 的值)。
- 任何以下划线开头的目录名。

由于某些文件系统可能不区分大小写，所以上面的名称都不应该用于任何大小写组合。其他用作安装目的地的常见目录名也可能出现在构建目录中，具体取决于构建二进制位置所使用的策略(28.5.2节中进一步讨论)。因此，避免源目录名(如`bin`、`lib`、`share`、`man`等)也是明智的选择。

一些项目选择顶层的`include`目录并在那里收集公共头文件，而不是将它们放在实现文件旁边。请注意，如果像这样分开头文件，一些IDE工具可能无法自动找到头文件，因此这样的方式可能对一些开发人员不太方便。它还倾向于对某个特定特性进行更改或不太本地化的Bug修复。另一方面，专用的`include`目录清楚地告知哪些头文件是公开的，可以拥有与安装时相同的目录结构。这两种方法都有各自的优点，但对于开发新人来说，将头文件与相关的实现文件放在一起可能会更简单一些。

## 28.4. IDE项目

当使用Xcode或Visual Studio等项目生成器时，会在构建目录的顶部创建一个项目或解决方案文件。这个文件可以在IDE中打开，就像这个应用程序的其他项目文件一样，并仍然在CMake的控制之下。重要的是，这些项目文件作为构建的一部分生成，所以不应该用版本控制系统跟踪。IDE中对项目所做的更改将在下一次CMake运行时丢失。

因为Xcode或Visual Studio项目文件由CMake生成，项目的目标和文件在项目层次结构或文件树中显示的方式也在CMake项目的控制之下。CMake提供了许多属性，这些属性可以影响某些IDE环境中如何对目标和文件进行分组和标记。第一级分组是针对目标的，可以通过将`USE_FOLDERS`全局属性设置为`true`来启用目标。然后可以使用`FOLDER`目标属性指定每个目标的位置，该属性区分大小写的名称，可以将目标放置在该名称下。要创建类似树的层次结构，可以使用前斜杠来分隔嵌套级别。如果文件夹属性为空或未设置，目标将保持在项目的顶层。Xcode和Visual Studio生成器都遵循文件夹目标属性。

```
set_property(GLOBAL PROPERTY USE_FOLDERS YES)

add_executable(foo ...)
add_executable(bar ...)
add_executable(test_foo ...)
add_executable(test_bar ...)

set_target_properties(foo bar PROPERTIES FOLDER "Main apps")
set_target_properties(test_foo test_bar PROPERTIES FOLDER "Main apps/Tests")
```

CMake 3.11之前，默认情况下文件夹目标属性是空，而在CMake 3.12之前，由`CMAKE_FOLDER`初始化。

IDE中为目标显示的名称默认与CMake使用的目标名称相同。Visual Studio生成器允许通过设置`PROJECT_LABEL`目标属性来覆盖显示名称，但是Xcode生成器不支持这个设置。

```
set_target_properties(foo PROPERTIES PROJECT_LABEL "Foo Tastic")
```

有些目标是由CMake本身创建的，比如安装、打包、运行测试等。

对于Xcode，大多数文件没有显示在文件/目标树中，但是对于Visual Studio，默认情况下是分组在一个名为`CMakePredefinedTargets`的文件夹下。可以用`PREDEFINED_TARGETS_FOLDER`全局属性覆盖它，但通常不会这样做。

每个目标对单个文件的分组也可以由CMake项目控制，可以使用`source_group()`完成，并且独立于目标文件夹分组(即使`USE_FOLDERS`全局属性为`false`或`unset`)。该命令有两种形式，第一种用于定义单个组：

```
source_group(group
[FILES src...]
[REGULAR_EXPRESSION regex]
)
```

组可以是简单的名称，用于对相关文件进行分组，也可以指定类似于目标的层次结构。由于历史原因，嵌套级别是由反斜杠而不是正斜杠定义的。为了正确的解析CMake，反斜杠必须转义，所以组`foo`下面的嵌套将以如下方式指定：

```
source_group(foo\\bar ...)
```

可以使用 `FILES` 参数指定单个文件，并假设路径相对于 `CMAKE_CURRENT_SOURCE_DIR`。因为该命令不是特定于某个目标，所以选项可以确保只有特定的文件受到分组的影响。如果项目定义一个更灵活的分组结构，使用 `REGULAR_EXPRESSION` 更合适，可有效地应用于项目中所有目标的分组规则。特定文件可以匹配多个分组的情况，文件优先于 `REGULAR_EXPRESSION`，后定义的正则 `REGULAR_EXPRESSION` 组优先于前面定义的正则表达式的组。

下面的示例为所有目标设置了通用规则，将具有常用源和头文件扩展名的文件分组到 `Sources` 之下。测试源和头文件将覆盖该分组，并放在 `Tests` 组下，而特殊情况是 `special.cxx` 将放在自己的专用子组下面。

```
source_group(Sources REGULAR_EXPRESSION "\\\.(c(xx|pp)?|hh?)$")
source_group(Tests REGULAR_EXPRESSION "test.*") # Overrides the above
source_group(Sources\\Special FILES special.cxx) # Overrides both of the above
```

CMake 为源文件提供默认的源文件组，为头文件提供默认的头文件组，这些都可以覆盖。还可以定义其他默认组，如 `Resources` 和 `Object Files`。

`source_group()` 的第二种形式允许组层次结构遵循特定文件的目录结构。可以在 CMake 3.8 或更高版本中使用。

```
source_group(TREE root
[PREFIX prefix]
[FILES src...]
)
```

`TREE` 指示命令根据根目录下的目录结构，对指定的文件进行分组。`PREFIX` 可用于将该分组结构置于前缀父组或组层次结构之下。这可以与 `SOURCES` 目录属性一起使用，以重新生成组成目标源的目录结构，但前提是所有源都低于公共点（例如，没有从 `build` 目录生成的源）。许多目标都满足这些条件，因此通常可以使用下面的示例模式，快速而轻松地在 IDE 中以目标的表示方式提供某种结构。

```
# Only suitable if SOURCES does not contain generated files in this example
get_target_property(sources someTarget SOURCES)

source_group(TREE ${CMAKE_CURRENT_SOURCE_DIR}
PREFIX "Magic\\Sources"
FILES ${sources}
)
```

IDE 通常只显示作为目标源显式添加的文件。如果目标定义为仅添加实现文件作为源，那么头文件通常不会出现在 IDE 文件列表中。因此，常见的做法是显式地列出头文件，即使它们不会进行编译，CMake 会有忽略它们，而不是将它们添加到 IDE 源文件列表中。这不仅可以扩展到头文件，还可以用于添加其他未编译的文件，如图像、脚本和其他资源。一些特性，比如与 `MACOSX_PACKAGE_LOCATION` 源属性相关的特性，需要将文件作为源文件列出，才能产生任何效果。

某些情况下，可能希望源文件出现在IDE文件列表中，但不进行编译，特定于平台的文件应该只在其他目标平台上编译和链接。为了防止CMake试图编译特定的文件，源文件的 `HEADER_FILE_ONLY` 源属性可以设置为`true`(不要被属性名称所迷惑，它的范围不仅仅是头文件)。

```
add_executable(myApp main.cpp net.cpp net_win.cpp)

if(NOT WIN32)
    # Don't need to compile this file for non-Windows platforms
    set_source_files_properties(net_win.cpp PROPERTIES
        HEADER_FILE_ONLY YES
    )
endif()
```

## 28.5. 定义目标

前面已经介绍了一系列CMake特性，允许详细定义目标。包括构建目标的源和其他文件、应该如何构建目标，以及目标间如何进行交互。本节的重点是演示如何使用这些技术，使项目易于理解，生成健壮的构建，并提高可维护性。

对于简单的项目，源文件和目标的数量可能很少，所有相关信息都在CMakeLists.txt文件中给出。如果遵循前面推荐的目录结构，这将意味着src目录将没有子目录，其CMakeLists.txt文件将定义所有内容。最初，看起来像这样：

*src/CMakeLists.txt*

```
add_executable(planter main.cpp soy.cpp coffee.cpp)

target_compile_definitions(planter PUBLIC COFFEE_FAMILY=Robusta)

add_test(NAME NoArgs COMMAND planter)
add_test(NAME WithArgs COMMAND planter beanType=soy)
```

这就对项目将如何使用做出了许多假设，最大的假设是项目不会安装或打包，并且不会吸收到更大的项目层次结构中。这些限制是可以避免的。上述简单案例的缺点有：

- 目标名称不特定于项目，因此如果合并到更大的父项目中，目标名称可能与其他目标冲突。在目标名称上使用特定于项目的前缀是解决这个缺点的简单方法。
- 由于没有安装规则，无法轻松地安装目标或将其包含在包中。
- 没有定义带命名空间的别名目标，即使后来添加了 `install()` 并实现了打包，其他项目对于预构建的二进制文件和源码包必须使用不同的目标名称。
- 测试名称不是特定于项目的，如果这个项目吸收到更大的项目层次结构中，那么可能会与其他项目的测试名称发生冲突。同样，将项目名称或其他字符串合并到测试名称中，可以解决这个问题。
- 总是添加测试，即使这不是顶级项目。对于具有许多测试的大型项目，这可能会增加不必要的构建时间。
- 头文件没有作为源文件列出，所以不会出现在某些IDE中。

针对以上几点，并遵循前几章推荐的实践，该示例需要扩展到以下内容：

*src/CMakeLists.txt*

```

=====
# Define targets
=====

add_executable(BagOfBeans_planter main.cpp soy.cpp soy.h coffee.cpp coffee.h)
add_executable(BagOfBeans::BagOfBeans_planter ALIAS BagOfBeans_planter)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
target_compile_definitions(BagOfBeans_planter PUBLIC COFFEE_FAMILY=Robusta)

=====
# Testing
=====

add_test(NAME BagOfBeans.planter.NoArgs COMMAND BagOfBeans_planter)
add_test(NAME BagOfBeans.planter.WithArgs COMMAND BagOfBeans_planter beanType=soy)

=====
# Packaging
=====

include(GNUInstallDirs)
install(TARGETS BagOfBeans_planter
        EXPORT BagOfBeans_apps
        DESTINATION ${CMAKE_INSTALL_BINDIR}
        COMPONENT BagOfBeans_apps
)

```

对于相当简单的可执行文件来说，这个CMakeLists.txt的信息量太大了。因为其强调了在实际项目中，除了单独构建二进制文件之外，还需要考虑更多事情。增加的复杂性主要使用了较长的名称，减少了冲突的可能。添加打包逻辑会增加大量的信息，这些细节对于没有经验的开发人员来说是很少能接触到的。如上所示，向文件中添加分割符，可以帮助新开发人员更容易理解文件，并且随着项目的发展，还可以使文件保持组织结构。

## 28.5.1. 目标源

源文件的数量增加时，全部放在一个目录中会使处理变得更加困难。通常将它们放在按功能分组的子目录下，这还有一些其他的好处。不仅有助于避免变得过于混乱，还可以使基于CMake缓存选项，或其他配置的逻辑更容易打开或关闭。例如：

```

add_executable(BagOfBeans_planter main.cpp)

option(BAGOFBEANS_SOY "Support planting soy beans" ON)
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)
if(BAGOFBEANS_SOY)
  add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
  add_subdirectory(coffee)
endif()

```

前面的章节中，可执行文件和库总在一个目录中定义，所以完整的文件列表可以直接提供给add\_executable()或add\_library()。子目录使用target\_sources() (CMake 3.1或更高版本可以使用)定义目标之后，将源添加到目标中。就像其他target\_...()一样工作，并且有非常相似的形式：

```

target_sources(targetName
  <PRIVATE|PUBLIC|INTERFACE> src...
  # Repeat with more sections as needed
  ...
)

```

提供了一个或多个 `PRIVATE`、`PUBLIC` 或 `INTERFACE` 部分，每个部分都列出了要保存的源文件

添加相关目标。将 `PRIVATE` 源添加到 `targetName` 的 `SOURCES` 属性，而将 `INTERFACE` 源添加到 `INTERFACE_SOURCES` 属性。`PUBLIC` 源会同时添加到这两个属性中。考虑这一点的更实用的方法是将 `PRIVATE` 源编译到 `targetName` 中，将 `INTERFACE` 源添加到链接到 `targetName` 中，而将 `PUBLIC` 源添加到两者中。

除了 `PRIVATE` 以外都是不常见，因为向所有通过 `targetName` 链接的目标添加源文件的作用有限。可以使用它来添加需要作为转换单元的部分资源，或者嵌入不应该通过接口公开的内容，这些情况都不常见。

`target_sources()` 的特性是，如果用相对路径指定了源，则该路径是相对于添加目标的源目录，这就产生了许多问题。第一个问题是，如果作为 `INTERFACE` 源添加的，则该路径将视为相对于其他目标的，而不是 `targetName`。显然，这可能会创建不正确的路径，因此任何非 `PRIVATE` 源都必须使用绝对路径指定。第二个问题是，当从定义 `targetName` 的目录以外的目录调用 `target_sources()` 时，相对路径并不直观。考虑如何指定前面示例中目录中的 `CMakeLists.txt` 文件：

`src/coffee/CMakeLists.txt`

```
target_sources(BagOfBeans_planter
PRIVATE
# WARNING: These will be wrong
coffee.cpp
coffee.h
)
...
```

上面的例子是添加来自相同目录的源文件，它们解释为相对于 `src` 而不是相对于 `src/coffee`。解决这个问题最健壮的方法是在前面加上 `CMAKE_CURRENT_SOURCE_DIR` 或 `CMAKE_CURRENT_LIST_DIR`，以确保始终使用正确的路径。

`src/coffee/CMakeLists.txt`

```
target_sources(BagOfBeans_planter
PRIVATE
${CMAKE_CURRENT_LIST_DIR}/coffee.cpp
${CMAKE_CURRENT_LIST_DIR}/coffee.h
)

target_compile_definitions(BagOfBeans_planter
PUBLIC COFFEE_FAMILY=Robusta
)

target_include_directories(BagOfBeans_planter
PUBLIC ${BUILD_INTERFACE}:${CMAKE_CURRENT_LIST_DIR}
)
```

这种情况下，在每个源文件前面加上 `${CMAKE_CURRENT_LIST_DIR}` 或 `$<...:${CMAKE_CURRENT_LIST_DIR}>` 不是很方便，也不直观。这方面的改进可能会在即将发布的版本中出现(必要的更改已经完成)。

上面还演示了如何将其他 `target_...` 命令移动到子目录中，而不仅仅是 `target_sources()`，这有助于将内容保留在本地。例如，只有在启用某个特性时，才添加特定于该特性的编译定义、编译器标志和头文件搜索路径。如果需要重新组织目录结构，并将该目录移到其他位置，则该文件中的任何内容都不需要更改，目标中有 `#include "coffee.h"` 的源文件不需要修改。

这种信息本地化的例外是 `target_link_libraries()`，它只能在同一目录中定义的目标上使用。如果子目录需要使目标链接到某个内容，就不能在子目录中这样做。对 `target_link_libraries()` 的调用必须在调用 `add_executable()` 或 `add_library()` 的目录中进行。例如，如果 `BagOfBeans_planter` 目标需要链接到名为 `weather` 的库，则必须在 `src/CMakeLists.txt` 中而不是在 `src/coffee/CMakeLists.txt` 中添加 `target_link_libraries()`。这将会有如下结果：

```

option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)

if(BAGOFBEANS_COFFEE)
  add_subdirectory(coffee)
  target_link_libraries(BagOfBeans_planter PRIVATE weather)
endif()

```

**CMake**开发人员正在积极讨论这一限制，并在未来的**CMake**版本中删除或放宽该限制。对于3.1到3.12的**CMake**版本，除了添加目标应该链接到的库之外，子目录可以自包含。**CMake 3.1**之前，需要完全不同的方法，依赖于在变量中建立源列表，只有在所有子目录都添加之后才创建目标。可能是这样：

```

# Pre-CMake 3.1 method, avoid using this approach
unset(planterSources)
unset(planterDefines)
unset(planterOptions)
unset(planterLinkLibs)

# Subdirs are expected to add to the above variables using PARENT_SCOPE
option(BAGOFBEANS_SOY "Support planting soy beans" ON)
option(BAGOFBEANS_COFFEE "Support planting coffee beans" ON)
if(BAGOFBEANS_SOY)
  add_subdirectory(soy)
endif()
if(BAGOFBEANS_COFFEE)
  add_subdirectory(coffee)
endif()

# Lastly define the target and its other details. All variables
# are assumed to name PRIVATE items.
add_executable(BagOfBeans_planter ${planterSources})
target_compile_definitions(BagOfBeans_planter PRIVATE ${planterDefines})
target_compile_options(BagOfBeans_planter PRIVATE ${planterOptions})
target_link_libraries(BagOfBeans_planter PRIVATE ${planterLinkLibs})

```

如果有些项不是 `PRIVATE`，上述问题会变得更加复杂。这样的变量使用非常脆弱，因为不依赖于子目录使用的变量，所以目标在变量中写错时**CMake**并不会发现这个错误。这些项还加强了父目录和子目录之间的耦合性，因为每个子目录都必须使用 `set(...PARENT_SCOPE)` 将相关的变量返回给父目录。对于嵌套很深的目录，这样做会出错。

## 28.5.2. 目标的输出

构建库或可执行文件时，其默认位置是 `CMAKE_CURRENT_BINARY_DIR`，或者是下面特定配置的子目录，具体取决于所使用的项目生成器类型。对于有许多子目录或深度嵌套层次结构的项目，这会给开发人员带来不便。对于这种情况，**CMake**提供了目标属性，让项目在一定程度上控制每个构建二进制文件的输出位置：

### `RUNTIME_OUTPUT_DIRECTORY`

用于所有平台上的可执行文件和Windows的DLL。

### `LIBRARY_OUTPUT_DIRECTORY`

用于非Windows 平台上的动态库。

### `ARCHIVE_OUTPUT_DIRECTORY`

用于所有平台上的静态库和Windows上与DLL库关联的导入库。

对于以上三种方式，Visual Studio和Xcode这样的多配置生成器将自动为每个值添加特定于配置的子目录(除非包含生成器表达式)。由于历史原因，也支持附加 `_<CONFIG>` 的配置属性，但应该避免使用需要配置的生成器表达式。

这些目标属性常见的用途是，将库和可执行文件一起收集到与安装时类似的目录结构中。如果应用程序希望各种资源位于相对于可执行文件的特定位置，这也很有用。**Windows**上可以简化调试，因为可执行文件和**DLL**可以收集到同一个目录中，允许可执行文件自动找到它们的**DLL**依赖项(其他平台上不需要这样做，`RPATH`可以将必要的位置嵌入到二进制文件中)。

按照通常的模式，这些目标属性由同名的**CMake**变量初始化。当所有目标都使用相同的输出位置时，可以在项目的顶层设置这些变量，这样就不必为每个目标单独设置属性。为了将项目合并到更大的项目中，只有在还没有设置这些变量时，才设置它们，以便父项目可以覆盖输出位置。还应该使用相对于 `CMAKE_CURRENT_BINARY_DIR` 的位置，而不是 `CMAKE_BINARY_DIR`。下面的示例展示了如何安全地收集当前构建目录下的**stage**子目录下的二进制文件(除非父项目进行了覆盖)。

```
include(GNUInstallDirs)
if(NOT CMAKE_RUNTIME_OUTPUT_DIRECTORY)
  set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
    ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_BINDIR})
endif()
if(NOT CMAKE_LIBRARY_OUTPUT_DIRECTORY)
  set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
    ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
if(NOT CMAKE_ARCHIVE_OUTPUT_DIRECTORY)
  set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
    ${CMAKE_CURRENT_BINARY_DIR}/stage/${CMAKE_INSTALL_LIBDIR})
endif()
```

避免创建 `CMAKE_OUTPUT_DIRECTORY` 作为缓存变量，因为它们不在开发人员的控制之下，应该由项目控制，项目的各个部分可能会对二进制文件的相对布局做出设定。更重要的是，将它们作为普通变量，意味着可以在定义测试可执行文件的子目录中取消设置，从而避免与其他二进制文件一起收集，使该区域变得混乱。

二进制文件的名称也可以由项目控制。默认情况下，二进制文件的名称与目标名称相同。当目标名称遵循合并项目名称的约定时(当作为较大项目层次结构的一部分时，可以保持它们的惟一性)，目标名称可能不适合作为二进制文件的名称，因此可能需要重写此默认值。`OUTPUT_NAME` 目标属性可以设置为二进制文件使用的名称，或者对于特殊情况，可以设置更具体的是 `RUNTIME_OUTPUT_NAME`、`LIBRARY_OUTPUT_NAME` 和 `ARCHIVE_OUTPUT_NAME` 属性。大多数情况下，`OUTPUT_NAME` 就足够了，也是首选的。

```
add_executable(BagOfBeans_planter ...)
set_target_properties(BagOfBeans_planter PROPERTIES OUTPUT_NAME planter)
```

由于历史原因，也支持如 `OUTPUT_NAME_<CONFIG>` 这样的特定配置变量，但是项目应该更倾向于使用生成器表达式。

较老的项目有时会尝试读取 `LOCATION` 目标属性，以确定二进制文件的输出位置和名称，并在自定义目标命令或其他类似逻辑中使用。这对于多配置生成器是有问题的，因为位置取决于配置，但 `LOCATION` 目标属性没有考虑到这一点，**CMake 3.0**及以后版本会在项目试图设置此目标属性时发出警告。项目应该使用像 `$<TARGET_FILE:...>` 这样的生成器表达式。

### 28.5.3. Windows平台的问题

Windows不支持 `RPATH` 会开发人员带来了许多问题。开发期间运行可执行文件时，可执行文件需要的DLL必须位于同一目录中，或者位于 `PATH` 环境变量中。为项目的主要二进制文件设置各种 `..._OUTPUT_PATH` 属性将可执行文件和库放在相同的位置，但是这种技术对测试不太方便，因为可能有许多可执行文件，并且将它们全部放在一个输出目录中使用起来也比较困难。

对于通过CTest执行的测试，可以使用 `ENVIRONMENT` 测试属性将所需的DLL目录添加到路径，如下所示：

```
add_executable(fooTest ...)
target_link_libraries(fooTest PRIVATE algo)
add_test(NAME fooWithAlgo COMMAND fooTest)
if(WIN32)
    set_tests_properties(fooWithAlgo PROPERTIES ENVIRONMENT
    "PATH=$<SHELL_PATH:$<TARGET_FILE_DIR:algo>>$<SEMICOLON>$ENV{PATH}"
    )
endif()
```

这对测试可执行文件在调试器下的Visual Studio IDE中运行没有帮助，所以需要更多的操作。CMake 3.8增加了对 `VS_USER_PROPS` 目标属性的支持，该属性可以在每个目标基础上覆盖用户属性文件的位置。创建自定义属性文件时，可以将 `LocalDebuggerEnvironment` 项添加到默认PATH中。如果测试所需的DLL都集中在少数的几个位置，可以为每个测试生成一个用户属性文件并重用(如果需要，仍然可以为每个目标生成和使用自定义用户属性文件)。可以使用 `configure_file()` 自动填充输出目录。

```
file(TO_NATIVE_PATH ${CMAKE_RUNTIME_OUTPUT_DIRECTORY} baseDir)
configure_file(user.props.in user.props @ONLY)
```

用户属性文件有点复杂，但是一个相当基础的例子，使用是示例可能是这样：

`user.props.in`

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Debug</LocalDebuggerEnvironment>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <LocalDebuggerEnvironment>PATH=@baseDir@\Release</LocalDebuggerEnvironment>
  </PropertyGroup>
</Project>
```

用户属性文件不仅可用于设置调试器环境，上面的内容还可以为希望进一步研究此技术的开发者提供起点。

对于Windows的可执行文件和DLL，通常会生成一个PDB(程序数据库)文件，以便在开发期间提供调试信息。有两种PDB文件，CMake为这两种文件提供了特性。对于动态库和可执行文件，可以使用 `PDB_NAME` 和特定于配置的 `PDB_NAME_<config>` 目标属性来覆盖PDB文件的名称。默认名称通常就挺好，因为它匹配DLL或可执行文件的名称，后缀为.pdb，而不是.dll或.exe。默认情况下，PDB文件放置在与DLL或可执行文件相同的目录中，也可用 `PDB_OUTPUT_DIRECTORY` 和配置特定的 `PDB_OUTPUT_DIRECTORY_<config>` 目标属性覆盖。注意，与其他 `_OUTPUT_DIRECTORY` 属性不同，`PDB_OUTPUT_DIRECTORY` 不支持CMake 3.11或更早版本的生成器表达式。

CMake还会创建第二种PDB文件，它保存为目标构建的各个对象文件的信息。这个PDB文件在开发过程中用处不大，除了用于静态库之外。对于C++，这个PDB文件的默认名称是VCxx.pdb。其中xx代表正在使用的Visual C++版本(例如VC14.pdb)。因为默认名称不是特定于目标的，所以在某些情况下很容易出错，并可能将不同目标的PDB混淆。CMake允许使用 `COMPILE_PDB` 和特定的 `COMPILE_PDB_<config>` 目标属性来控制每个目标PDB文件的名称。这些PDB文件的位置也可以

用 `COMPILE_PDB_OUTPUT_DIRECTORY` 和 `COMPILE_PDB_OUTPUT_DIRECTORY_<config>` 目标属性覆盖。

请注意，这些对象PDB文件对于DLL和可执行目标没什么用处，因为主PDB已经包含了所需的所有调试信息。

## 28.6. 杂项工程特性

项目生成器通常提供清理目标，可以用来删除所有生成的文件、构建输出等。IDE工具有时使用它来进行重新构建，作为在构建之后进行清理，或者开发人员使用它来删除构建输出，以便在下次构建时强制重新构建所有内容。有时候，项目定义了一个规则，创建了CMake不知道的文件，所以不包括在清理步骤中，并且仍然有可能影响下个构建。项目可以将这些文件添加到 `ADDITIONAL_MAKE_CLEAN_FILES` 目录属性来告诉CMake有关这些文件的信息，该属性保存该目录范围的文件列表，该目录范围应该是清理目标的一部分。只有Makefile系列生成器支持这一点。Ninja生成器不支持该属性，但可以通过 `add_custom_command()` 和 `add_custom_target()` 等命令提供的选项提供更健壮的替代方法。通过将这些文件作为副产品列出，Ninja知道在构建清理目标时要删除它们。其他项目生成器也有这样的功能。

如果特定的文件进行更改，某些高级的方法可能需要CMake重新运行。CMake在自动跟踪依赖关系方面做得很好，比如使用 `configure_file()` 复制文件，但是自定义命令和其他任务可能依赖于CMake没有跟踪到的文件。可以将这些文件添加到 `CMAKE_CONFIGURE_DEPENDS` 目录属性中，如果列出的文件发生了更改，将在下一次构建之前重新运行CMake。如果用相对路径指定文件，则文件视为相对于与目录属性相关联的源目录。大多数项目通常不需要使用 `CMAKE_CONFIGURE_DEPENDS` 目录属性，当CMake没有机会了解作为配置或生成步骤的输入文件时，应该使用它。大多数文件依赖关系是构建时依赖关系，而不是配置或生成时依赖关系，因此在使用此属性之前，请检查项目是否需要重新运行CMake，而不是作为常规构建的一部分简单地重新编译源文件或目标。

有时会出现这样的情况：需要将来自外部的源项目添加到构建中，但存在一些问题，无法正常工作。一些常见的例子包括没设置相关的变量或属性，特别是当项目支持非常老的CMake版本，并且没有更新来处理新的CMake特性和检查时。对于其中的一些问题，注入CMake代码解决问题是可能的(不修改外部项目)。`project()` 有个特性，会检查名为 `CMAKE_PROJECT_<PROJNAME>_INCLUDE` 的变量，其中 `<PROJNAME>` 是 `project()` 命令的项目名。如果定义了该变量，则假定该变量包含文件的名称，CMake应该将该文件作为 `project()` 返回之前执行的最后一项内容包含进来。实际上，`project()` 是这样工作的：

```
project(SomeProj)
if(CMAKE_PROJECT_SomeProj_INCLUDE)
    include(${CMAKE_PROJECT_SomeProj_INCLUDE})
endif()
```

因为每个 `project()` 调用都支持这种行为，因此每个 `project()` 调用都成为CMake代码注入的点。可以用来更改项目中目标属性的默认值，也可以用于添加额外的编译器或链接器标志等。该特性的另一个特别方便的用法是，可以安全地为持续集成构建设置选项，而不必将它们保存在CMake缓存中。这意味着增量构建不会受到旧的CMake缓存选项的影响，旧的CMake缓存选项会在项目进行更改后删除或不再设置。

例如，开发人员正在处理集成分支，其中应该临时启用额外的检查。一种简单的方法是显式地设置像 `CMAKE_C_FLAGS` 或 `CMAKE_CXX_FLAGS` 这样的变量，但是由于CI脚本不应该更改项目本身，唯一的选择就是将它们设置为缓存选项。这时惟一的操作是清除缓存，这可能会强制进行完整的重新构建。更好的替代方法是使用 `CMAKE_PROJECT_<PROJNAME>_INCLUDE`，在最顶层的 `project()` 调用结束时，处理特定于CI的文件。这个文件会像项目的其他部分一样处于源代码控制下。分支合并前，该文件会恢复正常，并且构建不会保留临时标志。

### CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(MyProj)
...
```

CI系统会像这样调用CMake:

```
cmake -D CMAKE_PROJECT_MyProj_INCLUDE:FILEPATH=path/to/ciOptions.cmake ...
```

文件ciOptions.cmake可能是空的，或者只是包含一些常见的设置，比如打开可选特性。对于分支，它可能包含如下内容：

*ciOptions.cmake*

```
compile_definitions(DO_EXTRA_CI_CHECKS=1)
set(ENABLE_SANITIZERS YES)
```

像这样将文件注入到`project()`中不应该是正常开发的一部分。它有特定的用途，用于克服较旧项目中的缺陷，以及一些非常的情况，比如在持续集成构建中，但在这些情况之外，开发人员通常更喜欢直接添加或修改项目的CMakeLists.txt文件。

## 28.7. 总结

项目的结构和使用方式可以不同。一些过去常见的惯例现在看起来就非常糟糕，因为新特性和经验使旧方法需要被更健壮、更灵活的方法取代，并可以完成以前完成不了的事情。工具在升级，语言在发展，依赖关系在变化——所有这些都意味着项目也需要随着时间的推移而更新。特别是对于CMake项目来说，以3.0之前的旧CMake版本为目标的项目将面临越来越多的坎坷。现在正朝着以目标为中心的模式迈进，CMake的大部分开发都是朝着这个方向发展。因此，最好使用支持这些特性的CMake版本。低于CMake 3.1的内容都可能过于严格，由于更新了语言支持和新特性，尽可能考虑CMake 3.7。如果使用较新的工具，如CUDA或新的语言标准，强烈建议使用最新的CMake版本。Visual Studio或Xcode的新版本也倾向于需要新的CMake版本，以便为工具链中的变化进行修补。

每个项目都需要做出的选择，将自己构建为超级构建还是常规构建。如果项目将CMake的最小版本设置为3.11，那么非超级构建的方式将有更强大的特性进行依赖管理，这可能会使超级构建的需求变得没有必要。考虑FetchContent模块和将本地导入目标推广到全局范围，是否为开发人员提供了更大的灵活性和更好的体验。当项目的所有依赖项都相对成熟，并且有良好的安装规则时，超级构建可能是一个合适的选择，其优势是可以与旧的CMake版本一起使用。这两种方法都有自己的定位，但在项目的生命周期中越早决定是否使用超级构建，项目就越有可能避免以后的大规模破坏性重组。

不管项目是不是超级构建，目标都是让项目的顶层集中在高层上。可以将顶层CMakeLists.txt文件看作是项目的目录。顶层目录应该包含管理文件和一组子目录，每个子目录都关注特定的区域。避免子目录名称与在构建目录中自动创建的名称发生冲突。使用相当标准的名称，除非有必须遵守的协议。

对于常规项目，目标是使顶层CMakeLists.txt文件遵循以下常用模式：

- 序言
- 项目设置
- 依赖关系
- 构建目标
- 测试
- 打包

用注释块清楚地描述每个部分，将有助于项目的开发人员维护该结构。跨项目建立此模式将有助于加强对顶层CMakeLists.txt文件流水的关注，可以看作为流程概述。

定义分散在不同目录中的源构建目标时，最好先创建目标，然后使用 `target_sources()` 添加目录源。适当的情况下，按功能或特性对子目录进行分组，以便可以容易地移动或作为一个单元启用/禁用。其他以目标为中心的命令(即 `target_compile_definitions()`、`target_compile_options()` 和 `target_include_directory()` )也可以在相关的子目录中使用。这有助于将信息保持在与其相关的位置，而不是分散到不同的目录中。避免使用变量来建立源列表，通过目录层次结构向上传递，并最终用于创建目标、定义编译器标志等。使用变量而不是直接操作目标会让项目更脆弱、更冗长，而且会让CMake捕捉不到拼写错误或其他错误。

按照上面的建议，构建简单的目标，避免使用不必要的变量来保存目标或项目的名称。应该特别避免下列模式：

```
set(projectName ...)
project(${projectName})
add_executable(${projectName} ...)
```

上面的例子把不应该有的情况放在一起。项目名称很少更改，在 `project()` 中直接指定项目的名称，如果需要在项目的其他地方引用，使用CMake提供的标准变量。对于目标，目标名称会广泛的使用，在变量中携带既麻烦又容易出错。为目标指定名称，并在整个项目中始终使用该名称。即使整个项目中只有一个目标，也不一定必须与项目名称相同，两者应该视为独立的，而非捆绑在一起。

添加测试时，考虑将测试代码放在与被测试代码较近的位置。这有助于将逻辑相关的代码放在一起，并鼓励开发人员使测试保持最新。源目录其他部分的测试很容易遗忘，对于涉及多个领域的测试(如集成测试)，局部性不是很强，因此在公共位置收集这些较高级别的测试可能更合适。顶层测试子目录可用于此类情况。

对于较大的项目，请考虑是否值得用IDE工具中表示项目的方式组织项目。如果有许多目标，除非使用文件夹目标属性添加了一些结构，否则很难处理该项目。对于那些有许多源的目标，可以使用 `source_group()` 对结构进行组织，该命令可用于围绕任何有意义的概念或特性定义层次结构。

应该特别考虑那些在Windows上构建的项目，开发人员可能使用Visual Studio IDE的项目。缺少 `RPATH` 支持意味着可执行文件依赖于能够在同一目录中，或通过PATH环境变量找到DLL依赖项。这既影响了通过CTest运行的测试程序，也影响了开发人员在Visual Studio IDE中运行可执行文件的能力。强制所有可执行文件和DLL在相同的输出目录中是解决问题的一种方法，可以通过各种 `...OUTPUT_DIRECTORY` 目标属性，及关联的 `CMAKE_...OUTPUT_DIRECTORY` 实现。避免在后期构建规则或自定义任务中复制DLL，可以将它们放在多个位置，以便其他可执行程序找到它们。但这种方式是脆弱的，很容易导致错误地使用DLL。

测试程序最好不要收集到与主程序和DLL相同的位置。一些测试代码可能要找到相对于它们自己位置的其他文件，因此需要将它们分开。使用 `ENVIRONMENT` 测试属性指定适当的PATH，以确保测试在通过CTest运行时能够找到所需的DLL。还可以考虑使用CMake 3.8或更高版本，并定义一个用户属性文件，然后使用 `VS_USER_PROPS` 目标属性来识别测试目标。这可用于具有调试器的环境，以便直接在Visual Studio IDE中运行测试。

使用Visual Studio生成器时，最好保持PDB的默认设置。这会让PDB文件出现在开发人员期望的位置，并具有与可执行文件或库匹配的名称。当使用生成器表达式时，尝试更改PDB文件的输出目录实现起来非常复杂，某些情况下，很难将PDB文件放入所需目录。