目录

1. Introduction 1.1
2. 现代的 CMake 的介绍 1.2
    1. 安装 CMake 1.2.1
    2. 运行 CMake 1.2.2
    3. CMake 行为准则 1.2.3
    4. CMake 各个版本添加的新特性 1.2.4
3. 基础知识简介 1.3
    1. 变量与缓存 1.3.1
    2. 用 CMake 进行编程 1.3.2
    3. 与你的代码交互 1.3.3
    4. 如何组织你的项目 1.3.4
    5. 在 CMake 中运行其他程序 1.3.5
    6. 一个简单的例子 1.3.6
4. 为 CMake 项目添加特性 1.4
    1. C++11 及后续版本 1.4.1
    2. 一些小而常见的需求 1.4.2
    3. 一些实用的工具 1.4.3
    4. 一些有用的模组 1.4.4
    5. CMake 对 IDE 的支持 1.4.5
    6. 调试 1.4.6
5. 包含子项目 1.5
    1. 子模组 1.5.1
    2. 使用 CMake 下载项目 1.5.2
    3. 获取软件包（FetchContent）(CMake 3.11+) 1.5.3
6. 测试 1.6
    1. GoogleTest 1.6.1
    2. Catch 1.6.2
7. 导出与安装 1.7
    1. 安装 1.7.1
    2. 导出 1.7.2
    3. 打包 1.7.3
8. 查找库（或包） 1.8
    1. CUDA 1.8.1
    2. OpenMP 1.8.2
    3. Boost 1.8.3
    4. MPI 1.8.4
    5. ROOT 1.8.5
        1. 使用文件系统的用例 1.8.5.1
        2. 使用目标系统的用例 1.8.5.2
        3. 使用生成字典的用例 1.8.5.3
    6. Minuit2 1.8.6

Introduction

Modern CMake 简体中文版

概述

这是著名 CMake 教程 Modern CMake 的简体中文翻译版。

你可以在 这里 找到它的原版。

它致力于解决网络上随处可见的糟糕例子以及所谓的“最佳实践”中存在的问题。

如果你想要学好 CMake ， 那你应该会从这本书中受益！

英文原版链接： https://cliutils.gitlab.io/modern-cmake/

简体中文版链接： https://modern-cmake-cn.github.io/Modern-CMake-zh_CN/

许可协议

贡献

本书是一篇持续维护的文档，你可以点击文档右上角的编辑按钮来对文章进行编辑。

同时，受限于译者的水平，不足之处敬请谅解，欢迎提出 Issue 或 Pull Request！

本文档遵循 中文文案排版指南，为了确保排版的美观以及风格的统一，请确保贡献内容符合指南规范。

现代的 CMake 的介绍

现代的 CMake 的介绍

人们喜欢对构建系统抱有敌意。 CppCon17 的讲座就是一个开发者们将构建系统当成头等笑话的例子。 这引出了一个问题：为什么（人们这样认为）？ 确实，使用构建系统构建项目时不乏这样那样的问题。 但我认为，在 2020 年，我们有一个非常好的解决方案来消除其中的一些问题。 那就是 CMake 。但不是 CMake 2.8 ，它比 C++11 还要早出现。 也不是那些糟糕的 CMake 例程（甚至包括那些 KitWare 自己的教程里发布的例子）。 我指的是现代的 CMake 。是 CMake 3.4+ ，甚至是 CMake 3.21+ ！它简洁、强大、优雅，所以你能够花费你的大部分时间在编写代码上，而不是在一个不可读、不可维护的 Make （或 CMake 2） 文件上浪费时间。 并且 CMake 3.11+ 的构建速度应该也会更加的快！！！

本书是一篇持续维护的文档。你可以在 GitLab 上提 issue 或是 合并请求。 你也可以 下载PDF 格式的副本。请务必查看一下 HSF CMake Training （也是一个 CMake 教程）！

简而言之，如果你正在考虑使用 Modern CMake，以下是你心中最可能存在的问题:

为什么我需要一个好的构建系统?

以下情况是否适用于你?

- 你想避免将路径硬编码
- 你需要在不止一台电脑上构建软件包
- 你想在项目中使用 CI（持续集成）
- 你需要支持不同的操作系统（甚至可能只是 Unix 的不同版本）
- 你想支持多个编译器
- 你想使用 IDE，但也许不总是使用
- 你想从逻辑上描述你的程序是如何结构的，而不是通过某些标志和命令
- 你想使用一个第三方库
- 你想使用工具，比如 Clang-Tidy，来帮助你编码
- 你想使用调试器来 Debug

如果是这样，你会从类似 CMake 的构建系统中受益。

为什么答案一定是CMake?

构建系统是一个热门话题。当然，有很多构建系统可选。但是，即使是一个真的非常好的构建系统，或者一个使用类似（CMake）的语法的，也不能达到 CMake 的使用体验。 为什么？ 因为生态。 每个 IDE 都支持 CMake（或者是 CMake 支持那个 IDE）。 使用 CMake 构建的软件包比使用其他任何构建系统的都多。 所以，如果你想要在你的代码中包含一个库，你有两个选择，要么自己写一个构建系统，要么使用该库支持的构建系统中的某个。而那通常包含 CMake。 如何你的工程包含的库多了，CMake 或很快成为那些库所支持的构建系统的交集。并且，如果你使用一个预装在系统中的库，它有很大可能有一个 find CMake 或者是一个 config CMake 的脚本。

为什么使用现代的 CMake ?

大概在 CMake 2.6-2.8 时， CMake 开始成为主流。它出现在大多数 Linux 操作系统的包管理器中，并被用于许多包中。

接着 Python 3 出现了。

这是一个直到现在某些的工程中进行的非常艰难、丑陋的迁移。

我知道，这和 CMake 没有任何关系。

但它们有一个 3 ,并且都跟在 2 后面。

所以我相信 CMake 3 跟在 Python 3 后面真是十分倒霉。[1] 因为尽管每一个版本的 CMake 都有良好的向后兼容性，但 CMake 3 却总是被当作新事物来对待。 你会发现像 CentOS7 这样的操作系统，其上的 GCC 4.8 几乎完全支持 C++14 ，而 CMake 则是在 C++11 之前几年就已经出现的 CMake 2.8 。

你应该至少使用在你的编译器之后出现的 CMake 版本，因为它需要知道该版本的编译器标志等信息。 而且，由于 CMake 只会启用 CMakeLists.txt 中包含的最低 CMake 版本所对应的特性，所以即使是在系统范围内安装一个新版本的 CMake 也是相当安全的。 你至少应该在本地安装它。这很容易（在许多情况下是 1-2 行命令），你会发现 5 分钟的工作将为你节省数百行和数小时的 CMakeLists.txt 编写，而且从长远来看，将更容易维护。

本书试图解决那些网上泛滥的糟糕例子和所谓”最佳实践“存在的问题。

其他资料

本书原作者的其他资料:

- HSF CMake Training
- Interactive Modern CMake talk

在网上还有一些其他的地方可以找到好的资讯。下面是其中的一些:

- The official help: 非常棒的文档。组织得很好，有很好的搜索功能，而且你可以在顶部切换版本。它只是没有一个很好的 “最佳实践教程”，而这正是本书试图解决的内容。
- Effective Modern CMake: 一个很好的 do's and don'ts 的清单。
- Embracing Modern CMake: 一篇对术语有很好描述的文章。
- It's time to do CMake Right: 一些现代的 CMake 项目的最佳实践。
- The Ultimate Guide to Modern CMake: 一篇有着本书类似目的稍显过时的文章。
- More Modern CMake: 来自 Meeting C++ 2018 的一个很棒的演讲，推荐使用 CMake 3.12 以上版本。该演讲将 CMake 3.0+ 称为 “现代 CMake”，将 CMake 3.12+ 称为 “更现代的 CMake”。
- Oh No! More Modern CMake: More Modern CMake 的续篇。
- toeb/moderncmake: 关于 CMake 3.5+ 的很好的介绍和例子，包括从语法到项目组织的介绍。

制作

Modern CMake 最初由 Henry Schreiner 编写. 其他的贡献者可以在 Gitlab的列表 中找到.

[1]. CMake 3.0 同样从非常老的CMake版本中删除了几个早已废弃的功能，并对与方括号有关的语法做了一个非常微小的向后不兼容的修改，所以这个说法并不完全公正；可能有一些非常非常老的CMake文件会在 CMake 3.0+ 中停止工作，但我从未遇到过。 ↩

安装 CMake

安装 CMake

你的 CMake 版本应该比你的编译器要更新，它应该比你使用的所有库（尤其是 Boost）都要更新。新版本对任何一个人来说都是有好处的。

如果你拥有一个 CMake 的内置副本，这对你的系统来说并不特殊。你可以在系统层面或用户层面轻松地安装一个新的来代替它。如果你的用户抱怨 CMake 的要求被设置得太高，请随时使用这里的内容来指导他们。尤其是当他们想要 3.1 版本以上，甚至是 3.21 以上版本的时候.....

快速一览（下面有关于每种方法的更多信息)

按作者的偏好排序：

- 所有系统
  - Pip (官方的，有时会稍有延迟)
  - Anaconda / Conda-Forge
- Windows
  - Chocolatey
  - Scoop
  - MSYS2
  - Download binary (官方的)
- macOS
  - Homebrew
  - MacPorts
  - Download binary (官方的)
- Linux
  - Snapcraft (官方的)
  - APT repository (仅适用于Ubuntu/Debian) (官方的)
  - Download binary (官方的)

官方安装包

你可以从 KitWare 上下载 CMake。如果你是在 Windows 上，这可能就是你获得 CMake 的方式。在 macOS 上获得它的方法也不错（而且开发者还提供了支持 Intel 和 Apple Silicon 的 Universal2 版本），但如果你使用 Homebrew 的话，使用`brew install cmake`会带来更好的效果（你应该这样做；苹果甚至支持 Homebrew，比如在 Apple Silicon 的推出期间）。你也可以在大多数的其他软件包管理器上得到它，比如 Windows 的 Chocolatey 或 macOS 的 MacPorts 。

在 Linux 上，有几种选择。Kitware 提供了一个 Debian/Ubunutu apt 软件库，以及 snap 软件包。官方同时提供了 Linux 的二进制文件包，但需要你去选择一个安装位置。如果你已经使用`~/.local`存放用户空间的软件包，下面的单行命令[1]将为你安装 CMake[2]。

```
~ $ wget -qO- "https://cmake.org/files/v3.21/cmake-3.21.0-
linux-x86_64.tar.gz" | tar --strip-components=1 -xz -C
~/.local
```

上面的名字在 3.21 版本中发生了改变：在旧版本中，包名是`cmake-3.19.7-Linux-x86_64.tar.gz`。如果你只是想要一个仅有 CMake 的本地文件夹：

```
~ $ mkdir -p cmake-3.21 && wget -qO-
"https://cmake.org/files/v3.21/cmake-3.21.0-linux-
```

```
x86_64.tar.gz" | tar --strip-components=1 -xz -C cmake-3.21
~ $ export PATH=`pwd`/cmake-3.21/bin:$PATH
```

显然，你要在每次启动新终端都追加一遍 PATH，或将该指令添加到你的`.bashrc`或 LMod 系统中。

而且，如果你想进行系统安装，请安装到`/usr/local`；这在 Docker 容器中是一个很好的选择，例如在 GitLab CI 中。请不要在非容器化的系统上尝试。

```
docker $ wget -qO- "https://cmake.org/files/v3.21/cmake-
3.21.0-linux-x86_64.tar.gz" | tar --strip-components=1 -xz -
C /usr/local
```

如果你在一个没有 wget 的系统上，请使用`curl -s`代替`wget -qO-`。

你也可以在任何系统上构建 CMake，这很容易，但使用二进制文件通常是更快的。

CMake 默认版本

下面是一些常见的构建环境和你会在上面发现的 CMake 版本。请自行安装 CMake，它只有 1-2 行，而且内置的版本没有什么 "特殊 "之处。它们通常也是向后兼容的。

Windows

另外 Scoop 一般也是最新的。来自 CMake.org 的普通安装程序在 Windows 系统上通常也很常见。

macOS

至少根据 Google Trends 的调查，如今 Homebrew 在 macOS 上的流行程度是相当高的。

Linux

RHEL/CentOS

CentOS 8 上的默认安装包不算太差，但最好不要使用 CentOS 7 上的默认安装包。请使用 EPEL 包来代替它。

Ubuntu

你应该只在 18.04 以上的版本使用默认的 CMake；它是一个 LTS 版本，并且有一个相当不错的最低版本！

Debian

其它 Linux 发行版

常用工具

在许多系统上只需`pip install cmake`。如果需要的话，请添加`--user`（如果需要的话，modern pip 会为你做好这个）。然而它目前还没有提供 Universal2 的轮子（wheels）。

CI

分布情况	CMake 版本	说明
TravisCI Xenial	3.12.4	2018 年 11 月中旬，这一映像已准备好广泛使用
TravisCI Bionic	3.12.4	目前与 Xenial 一样
Azure DevOps 18.04	3.17.0	
GitHub Actions 18.04	3.17.0	大部分与 Azure DevOps 保持同步
GitHub Actions 20.04	3.17.0	大部分与 Azure DevOps 保持同步

如果你在使用 GitHub Actions，也可以查看 jwlawson/actions-setup-cmake 进行操作，它可以安装你选择的 CMake，即使是在 docker 中也可以操作运行。

完整列表

小于 3.10 的版本用更深的红色标记。

也可参见 pkgs.org/download/cmake。

Pip

这也是一个官方软件包，由 CMake 作者在 KitWare 进行维护。这是一种相当新的方法，在某些系统上可能会失败（在我最后一次检查时，Alpine 还不被支持，但它有当时最新的 CMake），但它工作的效果非常好（例如在 Travis CI 上）。如果你安装了 pip（Python 的软件包安装程序），你可以这样做：

```
gitbook $ pip install cmake
```

只要你的系统中存在二进制文件，你便可以立即启动并运行它。如果二进制文件不存在，它将尝试使用 KitWare 的`scikit-build`包来进行构建。目前它还无法在软件包系统中作为依赖项，甚至可能需要（较早的）CMake 副本来构建。因此，只有在二进制文件存在的情况下我们才能使用这种方式，大多数情况下都是这样的。

这样做的好处是能遵从你当前的虚拟环境。然而，当它被放置在`pyproject.toml`文件中时，它才真正发挥了作用--它只会被安装到构建你的软件包中，而不会在之后保留下来！这简直太棒了。

就我个人而言，在 Linux 上时，我会把 CMake 的版本放入文件夹名中，比如`/opt/cmake312`或`~/opt/cmake312`，然后再把它们添加到 [LMod][]。参见`envmodule setup`，它可以帮助你在 macOS 或 Linux 上设置 LMod 系统。这需要花点时间来学习，但这是管理软件包和编译器版本的一个好方法。

[1]. 我想这是显而易见的，但你现在正在下载和运行代码，这会使你暴露在其他人的攻击之下。如果你是在一个重要的环境中，你应该下载文件并检查校验码。(注意，简单地分两步做并不能使你更安全，只有校验和码更安全) ↩

[2]. 如果你的主目录中没有.local，想要开始也很容易。只要建立这个文件夹，然后把export PATH="$HOME/.local/bin:$PATH"添加到你的.bashrc或.bash_profile或.profile文件中。现在你可以把你构建的任何软件包安装到-DCMAKE_INSTALL_PREFIX=~/.local而不是/usr/local! ↩

运行 CMake

运行 CMake

在编写 CMake 之前，要确保你已经清楚了如何运行 CMake 来构建文件。 几乎所有 CMake 项目都一样。

构建项目

除非另行说明，你始终应该建立一个专用于构建的目录并在那里构建项目。从技术上来讲，你可以进行内部构建（即在源代码目录下执行 CMake 构建命令），但是必须注意不要覆盖文件或者把它们添加到 git，所以别这么做就好。

这是经典的 CMake 构建流程 （TM）：

```
~/package $ mkdir build
~/package $ cd build
~/package/build $ cmake ..
~/package/build $ make
```

你可以用 `cmake --build .` 替换 `make` 这一行。它会调用 `make` 或者任何你正在使用的构建工具。如果你正在使用版本比较新的 CMake（除非你正在检查对于老版本 CMake 的兼容性，否则应该使用较新的版本），你也可以这样做：

```
~/package $ cmake -S . -B build
~/package $ cmake --build build
```

以下**任何**一条命令都能够执行安装：

```
# From the build directory (pick one)
~/package/build $ make install
~/package/build $ cmake --build . --target install
~/package/build $ cmake --install . # CMake 3.15+ only

# From the source directory (pick one)
~/package $ make -C build install
~/package $ cmake --build build --target install
~/package $ cmake --install build # CMake 3.15+ only
```

所以你应该选择哪一种方法？只要你**别忘记**输入构建目录作为参数，在构建目录之外的时间较短，并且从源代码目录更改源代码比较方便就行。你应该试着习惯使用 `--build`，因为它能让你免于只用 `make` 来构建。需要注意的是，在构建目录下进行工作一直都非常普遍，并且一些工具和命令（包括 CTest）仍然需要在 build 目录中才能工作。

额外解释一下，你可以指定 CMake 工作在**来自构建目录**的源代码目录，也可以工作在任何**现有**的构建目录。

如果你使用 `cmake --build` 而不是直接调用更底层的构建系统（译者注：比如直接使用 `make`），你可以用 `-v` 参数在构建时获得详细的输出（CMake 3.14+），用 `-j N` 指定用 N 个 CPU 核心并行构建项目（CMake 3.12+），以及用 `--target`（任意版本的 CMake）或 `-t`（CMake 3.15+）来选择一个目标进行部分地构建。这些命令因不同的构建系统而异，例如 `VERBOSE=1 make` 和 `ninja -v`。你也可以使用环境变量替代它们，例如 `CMAKE_BUILD_PARALLEL_LEVEL` (CMake 3.12+) 和 `VERBOSE` (CMake 3.14+)。

指定编译器

指定编译器必须在第一次运行时在空目录中进行。这种命令并不属于 CMake 语法，但你仍可能不太熟悉它。如果要选择 Clang：

```
~/package/build $ CC=clang CXX=clang++ cmake ..
```

这条命令设置了 bash 里的环境变量 CC 和 CXX，并且 CMake 会使用这些参数。这一行命令就够了，你也只需要调用一次；之后 CMake 会继续使用从这些变量里推导出来的路径。

指定生成器

你可以选择的构建工具有很多；通常默认的是 `make`。要显示在你的系统上 CMake 可以调用的所有构建工具，运行：

```
~/package/build $ cmake --help
```

你也可以用 `-G"My Tool"`（仅当构建工具的名字中包含空格时才需要引号）来指定构建工具。像指定编译器一样，你应该在一个目录中第一次调用 CMake 时就指定构建工具。如果有好几个构建目录也没关系，比如 `build/` 和 `buildXcode`。你可以用环境变量 `CMAKE_GENERATOR` 来指定默认的生成器（CMake 3.15+）。需要注意的是，makefiles 只会在你明确地指出线程数目之时才会并行运行，比如 `make -j2`，而 Ninja 却会自动地并行运行。在较新版本的 CMake 中，你能直接传递并行选项，比如`-j2`，到命令 `cmake --build`。

设置选项

在 CMake 中，你可以使用 `-D` 设置选项。你能使用 `-L` 列出所有选项，或者用 `-LH` 列出人类更易读的选项列表。如果你没有列出源代码目录或构建目录，这条命令将不会重新运行 CMake（使用 `cmake -L` 而不是 `cmake -L .`）。

详细和部分的构建

同样，这不属于 CMake，如果你正使用像 `make` 一样的命令行构建工具，你能获得详细的输出：

```
~/package/build $ VERBOSE=1 make96
```

我们已经提到了在构建时可以有详细输出，但你也可以看到详细的 `CMake` 配置输出。`--trace` 选项能够打印出运行的 `CMake` 的每一行。由于它过于冗长，`CMake 3.7` 添加了 `--trace-source="filename"` 选项，这让你可以打印出你想看的特定文件运行时执行的每一行。如果你选择了要调试的文件的名称（在调试一个 `CMakeLists.txt` 时通常选择父目录，因为它们名字都一样），你就会只看到这个文件里运行的那些行。这很实用！

实际上你写成 `make VERBOSE=1`，make 也能正确工作，但这是 `make` 的一个特性而不是命令行的惯用写法。

你也可以通过指定一个目标来仅构建一部分，例如指定你已经在 CMake 中定义的库或可执行文件的名称，然后 make 将会只构建这一个目标。

选项

CMake 支持缓存选项。CMake 中的变量可以被标记为 "cached"，这意味着它会被写入缓存（构建目录中名为 `CMakeCache.txt` 的文件）。你可以在命令行中用 `-D` 预先设定（或更改）缓存选项的值。CMake 查找一个缓存的变量时，它就会使用已有的值并且不会覆盖这个值。

标准选项

大部分软件包中都会用到以下的 CMake 选项：

- `-DCMAKE_BUILD_TYPE=` 从 Release， RelWithDebInfo， Debug， 或者可能存在的更多参数中选择。
- `-DCMAKE_INSTALL_PREFIX=` 这是安装位置。UNIX 系统默认的位置是 `/usr/local`，用户目录是 `~/.local`，也可以是你自己指定的文件夹。
- `-DBUILD_SHARED_LIBS=` 你可以把这里设置为 `ON` 或 `OFF` 来控制共享库的默认值（不过，你也可以明确选择其他值而不是默认值）
- `-DBUILD_TESTING=` 这是启用测试的通用名称，当然不会所有软件包都会使用它，有时这样做确实不错。

调试你的 CMake 文件

我们已经提到了在构建时可以有详细输出，但你也可以看到详细的 CMake 配置输出。`--trace` 选项能够打印出运行的 CMake 的每一行。由于它过于冗长，CMake 3.7 添加了 `--trace-source="filename"` 选项，这让你可以打印出你想看的特定文件运行时执行的每一行。如果你选择了要调试的文件的名称（在调试 CMakeLists.txt 时通常选择父目录，因为它的名字在任何项目中都一样），你就会只看到这个文件里运行的那些行。这很实用！

CMake 行为准则

CMake 行为准则(Do's and Don'ts)

CMake 应避免的行为

接下来的两个列表很大程度上基于优秀的 gist Effective Modern CMake. 那个列表更长且更详细，也非常欢迎你去仔细阅读它。

- **不要使用具有全局作用域的函数**：这包含 `link_directories`、`include_libraries` 等相似的函数。
- **不要添加非必要的 PUBLIC 要求**：你应该避免把一些不必要的东西强加给用户（-Wall）。相比于 **PUBLIC**，更应该把他们声明为 **PRIVATE**。
- **不要在file函数中添加 GLOB 文件**：如果不重新运行 CMake，Make 或者其他的工具将不会知道你是否添加了某个文件。值得注意的是，CMake 3.12 添加了一个 `CONFIGURE_DEPENDS` 标志能够使你更好的完成这件事。
- **将库直接链接到需要构建的目标上**：如果可以的话，总是显式的将库链接到目标上。
- **当链接库文件时，不要省略 PUBLIC 或 PRIVATE 关键字**：这将会导致后续所有的链接都是缺省的。

CMake 应遵守的规范

- **把 CMake 程序视作代码**：它是代码。它应该和其他的代码一样，是整洁并且可读的。
- **建立目标的观念**：你的目标应该代表一系列的概念。为任何需要保持一致的东西指定一个 （导入型）INTERFACE 目标，然后每次都链接到该目标。
- **导出你的接口**：你的 CMake 项目应该可以直接构建或者安装。
- **为库书写一个 Config.cmake 文件**：这是库作者为支持客户的体验而应该做的。
- **声明一个 ALIAS 目标以保持使用的一致性**：使用 `add_subdirectory` 和 `find_package` 应该提供相同的目标和命名空间。
- **将常见的功能合并到有详细文档的函数或宏中**：函数往往是更好的选择。
- **使用小写的函数名**： CMake 的函数和宏的名字可以定义为大写或小写，但是一般都使用小写，变量名用大写。
- **使用 `cmake_policy` 和/或 限定版本号范围**： 每次改变版本特性 (policy) 都要有据可依。应该只有不得不使用旧特性时才降低特性 (policy) 版本。

CMake 各个版本添加的新特性

CMake 各个版本添加的新特性

CMake 修改记录的简化版本，这里仅挑了作者认为的重点。这里，每个版本的名称都由作者自行命名，不要太在意。

CMake 3.0：接口库

这个版本添加了大量内容，主要是为了填充目标接口。一些需要的功能遗弃了，并在 CMake 3.1 中重新实现。

- 首次发布于 2014年6月10日
- 更新了文档
- 添加了 INTERFACE 库类型
- 支持项目版本关键字 VERSION
- 导出构建树更容易
- 括号参数和支持注释(未广泛使用)
- 以及其他很多改进

CMake 3.1：支持 C++11 和编译特性

支持 C++11 的第一个版本，并针对 CMake 3.0 新特性进行了修复。如若需要使用旧版 CMake，该版本推荐作为最低。

- 首次发布于 2014年12月17日
- 支持 C++11
- 支持编译特性
- 源文件可以通过 `target_sources` 在创建目标之后添加
- 优化了生成器表达式和 INTERFACE 目标

CMake 3.2：UTF8

一个小版本，主要是添加了小功能和对之前功能缺陷的修复。还有一些内部变化有，我认为对 Windows 和 UTF8 支持更好这个很重要。

- 首次发布于 2015年3月11日
- 可以在循环中使用 `continue()`
- 新增文件和目录锁

CMake 3.3：if 中添加 IN_LIST

if中添加了 `IN_LIST` 选项，并且可以使用环境变量 `$PATH` (详见 CMake 3.6) 对库文件进行搜索，添加了 INTERFACE 库的依赖关系，还有其他一些改进。随着支持的语言越来越多， `COMPILE_LANGUAGE` 支持生成器表达式就很有必要了。并且，Makefile 在并行执行时的输出更好看了。

- 首次发布于2015年7月23日
- `if` 支持 `IN_LIST` 关键字
- 新增 `*_INCLUDE_WHAT_YOU_USE` 属性
- `COMPILE_LANGUAGE` 支持生成器表达式(只有某些生成器支持)

CMake 3.4：Swift & CCache

这个版本增加了许多有用的工具，对 Swift 语言的支持，以及常用功能的改进。也开始支持编译器启动器，比如 CCache。

- 首次发布于 2015年11月12日
- 支持 `Swift` 语言
- `get_filename_component` 添加`BASE_DIR` 选项
- 新增 `if(TEST ...)`
- 新增 `string(APPEND ...)`
- 为 make 和 ninja 添加了新的内置变量 `CMAKE_*_COMPILER_LAUNCHER`
- `TARGET_MESSAGES` 允许 Makefile 在目标完成后打印消息
- 导入目标开始出现在官方的 `Find*.cmake` 文件中

CMake 3.5：ARM

这个版本将 CMake 扩展到更多的平台，并且可以使用命令行来控制警告信息。

- 首次发布于 2016年3月8日
- 多个输入文件可以对应多个 `cmake -E` 命令。
- 内置 `cmake_parse_arguments` 解析指令
- Boost、GTest 等库支持导入目标
- 支持 ARMCC，优化对 iOS 的支持
- XCode 反斜杠问题修复

CMake 3.6：Clang-Tidy

这个版本增加了 Clang-Tidy 支持，添加了更多的工具和对原有功能的改进。 取消了在 Unix 系统上搜索 `$PATH` 的问题，取而代之的是使用 `$CMAKE_PREFIX_PATH`。

- 首次发布于 2016年7月7日
- 为工程安装时添加 `EXCLUDE_FROM_ALL`
- 新增 `list(FILTER`
- 工具链添加了 `CMAKE_*_STANDARD_INCLUDE_DIRECTORIES`和 `CMAKE_*_STANDARD_LIBRARIES`
- 改进了 Try-compile 功能
- 新增 `*_CLANG_TIDY` 属性
- 外部项目可以是浅克隆，以及其他改进

CMake 3.7：Android & CMake 的服务器模式

可以使用交叉编译，构建在 Android 平台运行的程序。if 的新选项可使代码可读性更好。新增的服务器模式是为了提高与 IDE 的集成(但 CMake 3.14+ 使用另一种方式取而代之)。优化了对 VIM 编辑器的支持。

- 首次发布于 November 11, 2016
- `cmake_parse_arguments` 新增了 `PARSE_ARGV` 模式
- 改进了在 32 位工程在 64 位环境中的构建
- if 增加了很多好用的比较选项，比如 `VERSION_GREATER_EQUAL` (真的需要这么久吗?)
- 新增 `LINK_WHAT_YOU_USE`
- 大量与文件和目录相关的自定义属性
- 新增 CMake 服务器模式
- 新增 `--trace-source="filename"`，用于监控某些文件

CMake 3.8：C# & CUDA

CUDA 作为一种语言加入了 CMake，使用 `cxx_std_11` 作为编译器元特性。若使用 CMake 3.8+，新的生成器表达式真的很好用!

- 首次发布于2017年4月10日
- 原生支持 C# 语言
- 原生支持 CUDA 语言
- 新增元特性 `cxx_std_11`(以及14和17)
- 优化 `try_compile` 对语言的支持
- 新增 `BUILD_RPATH` 属性
- `COMPILE_FLAGS` 支持生成器表达式
- 新增 `*_CPPLINT`
- 新增 `$<IF:cond,true-value,false-value>` (wow!)
- 新增 `source_group(TREE` (终于可以在 IDE 中显示项目的文件夹结构了!)

CMake 3.9：IPO

这个版本对 CUDA 支持进行了大量修复，包括对 `PTX` 和 MSVC 生成器的支持。过程间优化(IPO)已正确支持了。

甚至有更多模块提供导入的目标，包括 MPI。

- 首次发布于2017年7月18日
- CUDA 支持 Windows
- 优化部分情况下对对象库的支持
- `project` 新增 `DESCRIPTION` 关键字
- `separate_arguments` 新增 `NATIVE_COMMAND` 模式
- `INTERPROCEDURAL_OPTIMIZATION` 强制执行(以及添加 `CMAKE_*` 初始化器，新增 CheckIPOSupported，支持 Clang 和 GCC)
- 新增了`GoogleTest`模块
- 对`FindDoxygen`进行了大幅度改进

CMake 3.10：CppCheck

CMake 现在使用 C++11 编译器构建，许多改进有助于编写可读性更好的代码。

- 首次发布于2017年11月20日
- 支持 Fortran 编译器 flang
- 将编译器启动器添加到 CUDA
- `configure_file` 支持 `#cmakedefines`
- 新增 `include_guard()`，确保CMake源文件只包含一次
- 新增 `string(PREPEND`
- 新增 `*_CPPCHECK` 属性
- 目录添加了 `LABELS` 属性
- 极大地扩展了 FindMPI 模块
- 优化了 FindOpenMP 模块
- `GoogleTest` 可动态发现测试用例
- `cmake_host_system_information` 可获取更多信息。

CMake 3.11：更快 & IMPORTED INTERFACE

这个版本运行起来 应该会 快很多，还可以直接将 INTERFACE 目标添加到 IMPORTED 库(内部的 `Find*.cmake` 脚本会更加清晰)。

- 首次发布于 2018年3月28日
- Fortran 支持编译器启动器
- Xcode 和 Visual Studio 支持 `COMPILE_LANGUAGE` 的生成器表达式

- 可以直接将 INTERFACE 目标添加到 IMPORTED INTERFACE 库中(Wow!)
- 对源文件属性进行了扩展
- `FetchContent` 模块现在允许在配置时下载 (Wow)

CMake 3.12：版本范围和CONFIGURE_DEPENDS

非常牛的版本，包含了许多长期要求添加的小功能。其中一个是新增了版本范围，现在可以更容易地设置最低和最高的 CMake 版本了。也可以在一组使用 `GLOB` 获取的文件上设置 `CONFIGURE_DEPENDS`，构建系统将检查这些文件，并在需要时重新运行！还可以对 `find_package` 的搜索路径使用通用的 `PackageName_ROOT` 。对 string 和 list 大量的功能添加、模块更新、全新的 Python 查找模块(2 和 3 版本都有)等等。

- 首次发布于2018年7月17日
- 支持 `cmake_minimum_required` 的范围表示（向后兼容）
- 使用命令行 `--build` 构建时，支持 `-j,--parallel` 进行并行构建（传递给构建工具）
- 支持编译选项中的 `SHELL:` 字符串（不删除）
- 新增 `FindPython` 模块
- 新增 `string(JOIN`，`list(JOIN` 和 `list(TRANSFORM`
- 新增 `file(TOUCH` 和 `file(GLOB CONFIGURE_DEPENDS`
- 支持 C++20
- CUDA 作为语言的改进：支持 CUDA 7 和 7.5
- 支持 macOS 的 OpenMP (仅限命令行)
- 新增了几个新属性和属性初始化器
- CPack 可读取 `CMAKE_PROJECT_VERSION` 变量

CMake 3.13：连接控制

可以在Windows创建符号链接了！新增了许多新函数，响应了 CMake 的主流请求，如 `add_link_options`,`target_link_directories` 和 `target_link_options`。可以在源目录之外对目标进行更多的修改，可以更好的实现文件分离。`target_sources` 终于可以正确地处理相对路径（策略76）了。

- 首次发布于 2018年11月20日
- 新增 `ctest --progress` 选项，输出实时测试进度
- 新增 `target_link_options` 和 `add_link_options`
- 新增 `target_link_directories`
- 创建符号链接 `-E create_symlink`，只支持 Windows
- Windows 支持 IPO
- 可对源目录和构建目录使用 `-S` 和 `-B`
- 可对当前目标外的目录使用 `target_link_libraries` 和 `install`
- 新增 `STATIC_LIBRARY_OPTIONS` 属性
- `target_sources` 现在相对于当前源目录（CMP0076）
- 若使用 Xcode，可以实验性地设置 schema 字段

CMake 3.14：文件工具 (AKA CMake π)

进行了很多小清理，包括几个用于文件的工具。生成器表达式可以在更多的地方使用，使用 list 要优于使用空变量。很多的 find 包可以产生目标。Visual Studio 16 2019 生成器与旧版本略有不同。不支持 Windows XP 和Vista。

- 首次发布于 2019年3月14日

- `--build` 命令添加了 `-v/--verbose` 选项。若构建工具支持，可以使用冗余构建
- FILE指令新增了 `CREATE_LINK`，`READ_SYMLINK` 和 `SIZE` 选项
- `get_filename_component` 新增了 `LAST_EXT` 和 `NAME_WLE` 用于获取文件*最后的*扩展名，比如可以从文件名 `version.1.2.zip`，获取后缀名 `.zip`（非常方便!）
- 可以在 `if` 语句中使用 `DEFINED CACHE{VAR}`，查看是否在 CACHE 中定义了变量。
- 新增 `BUILD_RPATH_USE_ORIGIN`，以改进对构建目录中 RPath 的处理。
- CMake 服务器模式使用一个文件 API 所取代。从长远来看，这会影响 IDE。

CMake 3.15：升级CLI

这个版本有许多较小改进，包括对CMake命令行的改进，比如：通过环境变量控制默认生成器（现在很容易将默认生成器改为 Ninja）。`--build` 模式支持多个目标，添加了 `--install` 模式。CMake支持多级日志记录。可以使用一些方便的工具来测试生成器表达式。FindPython 模块持续改进，FindBoost 与 Boost 1.70 的新 CONFIG 模块有了更多的内联。`export(PACKAGE)` 发生了巨大变化，不再将默认目录设置为 `$HOME/.cmake` (若 cmake 最小版本为 3.15+)，若用户若想使用它，需要额外的设置步骤。

- 首次发布于 2019年7月17日
- 新增控制默认生成器的环境变量 `CMAKE_GENERATOR`
- 命令行可构建多个目标，`cmake . --build --target a b`
- `--target` 可缩写为 `-t`
- 项目安装支持命令行选项 `cmake . --install`，该过程不使用构建系统
- 支持日志级别参数 `--loglevel`，为 `message` 指令添加 `NOTICE`，`VERBOSE`，`DEBUG`和 `TRACE`选项
- `list` 指令新增了 `PREPEND`、`POP_FRONT`和 `POP_BACK` 选项
- `execute_process` 指令新增了 `COMMAND_ECHO` 选项 (`CMAKE_EXECUTE_PROCESS_COMMAND_ECHO`) 可以在运行命令之前自动显示具体命令
- Ninja 的几个改进，包括对 SWIFT 语言的支持
- 改进编译器和列表的生成器表达式

CMake 3.16：统一构建

添加了统一构建模式，允许源文件合并成单独的构建文件。增加了对预编译头文件的支持（可能是为 C++20 的模块做准备），完成了对许多小功能的修复，特别是对较新的特性，如 FindPython、FindDoxygen 等。

- 首次发布于 2019年11月26日
- 新增对 Objective C 和 Objective C++ 语言的支持
- 使用 `target_precompile_headers` 支持预编译头文件
- 支持使用 “Unity” 或 “Jumbo” 构建时(合并源文件)使用 `CMAKE_UNITY_BUILD`
- CTest：展开列表，可跳过基于正则表达式的方式
- 控制 RPath 的几个新特性。
- 生成器表达式可以在更多地方使用，比如构建和安装路径
- 可以通过新变量显式地控制查找位置

CMake 3.17：原生支持CUDA

添加了 FindCUDAToolkit，允许在不启用 CUDA 语言的情况下，查找和使用 CUDA 工具包！CUDA 现在更具可配置性，例如：链接到动态库。其他功能做了很多优化，比如：FindPython。并且，可以一次性遍历多个列表。

- 首次发布于 2020年3月20日
- `CUDA_RUNTIME_LIBRARY` 终于可以设置为 Shared！
- 新增 FindCUDAToolkit
- `cmake -E rm` 替换旧的删除命令
- 添加 CUDA 元特性，如`cuda_std_03`等。
- `--debug-find` 可跟踪包的搜索
- ExternalProject 可以禁用递归签出
- FindPython 更好地与 Conda 集成
- DEPRECATION 可以应用于目标
- 新增 rm 命令
- 几个新的环境变量
- foreach 新增 `ZIP_LISTS` 选项（一次性遍历多个列表）

CMake 3.18：CUDA与Clang & CMake宏特性

CUDA 现在支持 Clang （不可分离编译）。新增了 CUDA_ARCHITECTURES 属性，可以更好地支持针对 CUDA 硬件。cmake_language 命令支持从字符串中使用 cmake 命令和表达式。还有许多其他元特性的变化，可以使新功能可用：通过变量调用函数，解析字符串，并使用字符串配置文件。还有许多其他漂亮的小功能添加和功能修复，下面是其中的一些。

- 首次发布于 2020年7月15日
- `cmake` 命令可使用 `cat` 合并多个文件
- 新增 `cmake` 命令的分析模式
- `cmake_language` 新增 `CALL` 和 `EVAL` 选项
- 若需多次导出，可使用 `export` 的 `APPEND` 选项（CMake 3.18+）
- 可以使用 `file()` 进行打包
- 若需要替换文件中的字符串，`file(CONFIGURE` 是比 `configure_file` 更好的方式
- 其他 `find_*` 命令新增了 `find_package` 的 `REQUIRED` 标志
- 为 `list(SORT` 新增了 `NATURAL` 比较模式
- 新增处理 DIRECTORY 作用域属性的多个选项
- 新增 `CUDA_ARCHITECTURES`
- 新增 `LINK_LANGUAGE` 生成器表达式（包括 `DEVICE/HOST` 版本）
- 源目录可以成为 `FetchContent` 的子目录

CMake 3.19：预设

可以以 JSON 的方式添加预设，用户将获得预设的默认值。`find_package` 支持版本范围，特殊的查找模块，比如：FindPython，有对版本范围的自定义支持。添加了许多新的权限控制，进一步的普及生成器表达式。

- 首次发布于 2020年11月18日
- CMake预设文件 —— 可以为每个生成器的项目设置默认值，或者用户可以进行预设。即使当前项目没有使用 `CMakePresets.json`，也可将 `CMakeUserPresets.json` 添加到 `.gitignore` 中。
- XCode 12+ 中引入了新的构建系统
- 支持 MSVC 对 Android 的构建
- 新增 `cmake -E create_hardlink`
- `add_test` 正确地支持测试名称中的空格
- 可将 `cmake_language` 中标记为 `DEFER` 的目录放在最后进行处理

- 大量新 file 选项，如临时下载和 ARCHIVE_CREATE 的 COMPRESSION_LEVEL
- find_package 支持版本范围
- DIRECTORY 可以在属性命令中包含二进制目录
- string 新增 JSON 模式
- 新 OPTIMIZE_DEPENDENCIES 属性和 CMAKE_* 变量可智能地删除静态库和对象库的依赖项。
- PCH 支持 PCH_INSTANTIATE_TEMPLATES 属性和 CMAKE_* 变量。
- 检查模块支持 CUDA 和 ISPC 语言
- FindPython：新增 Python*_LINK_OPTIONS
- ctest 的 compute-sanitizer 支持 CUDA 的 memcheck

CMake 3.20：文档

CMake 文档通过添加 “new in” 标签来快速查看添加的内容，无需切换文档版本，提高了工作效率！新增 C++23 的支持。源文件必须列出扩展名，并且始终遵循设置的 LANGUAGE 规则。还做了相当多的清理工作（为了使 工程部署的阻碍最小化，最好使用版本 ...3.20 对源码进行测试），继续改进预设。

- 首次发布于 2021年3月23日
- 支持 C++23
- 新增 CUDAARCHS 环境变量，用于设置 CUDA 架构
- 支持新的 IntelLLVM 编译器（OneAPI 2021.1）和 NVHPC的 NVIDIA HPC SDK
- 一些扩展生成器表达式支持自定义命令/目标，可在安装时重命名
- 新增的 cmake_path 命令可用于路径
- try_run 新增了 WORKING_DIRECTORY 选项
- file(GENERATE 添加了很多特性
- 一些功能或特性的移除，如 cmake-server，WriteCompilerDetectionHeader（若策略设置为3.20+），以及一些可用新方法替代的东西。
- 源文件必须包含扩展名

CMake 3.21：配色

不同的消息类型有不同的颜色！现在有变量可以查看是否在顶级项目中。大量有关持续清理和特化的新特性，如添加HIP语言和C17和C23支持。继续改进预设。

- 首次发布于 2021年7月14日
- 初步支持 MSVC 2022
- 为 make 和 ninja 添加了 CMAKE_<LANG_LINKER_LAUNCHER
- HIP 作为语言添加
- 新增 C17 和 C23 支持
- 新增 --install -prefix <dir> 和 --toolchain <file>
- 消息根据消息类型着色！
- 支持 MSYS，包括 FindMsys
- file(指令更新，添加了 EXPAND_TILDE 属性
- 支持向 install 添加运行时的依赖项和工件
- 新增 PROJECT_IS_TOP_LEVEL 和 <PROJECT-NAME>_IS_TOP_LEVEL
- find_ 指令在缓存方面的改进

CMake 3.22：方便的环境变量

一个较小的版本，在常见的构建方面进行了一些不错的改进。可以在开发环境中设置 CMAKE_BUILD_TYPE 来设置默认的构建类型，还有其他几个新环境变量和变量

的添加。与标准相关的编译器标志进行了改进。
`cmake_host_system_information` 在操作系统信息方面得到了进一步的改进（从 3.10 开始）。

- 首次发布于 2021年11月18日
- 新的默认环境变量 `CMAKE_BUILD_TYPE` 和 `CMAKE_CONFIGURATION_TYPES`
- 新增环境变量 `CMAKE_INSTALL_MODE` 用于安装类型（symlink）
- 新增 `CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>`变量，将可选查找转换为必选查找
- 新增针对编译器的 `CMAKE_<LANG>_EXTENSIONS_DEFAULT` 变量
- `CMakeDependentOption` 可使用正常的条件语法
- CTest 可以修改环境变量
- 一些生成器可以在使用 MSVC 时包含外部（系统）头文件

CMake 3.23：纯头文件库

一个可靠的版本，只关注头文件库，更多的用户控件，CMake 预设，以及更好的 CUDA 支持。纯头文件库有一些强大的新特性，比如：各种 `*_SETS` 目标属性。有一些新的控件，可以限制 `find_` 查找路径，以及从现有目标中删除 `SYSTEM`。还可以获得了扩展的调试特性，以及将所有链接强制指向目标。预设可以包括其他文件。CUDA 和C# 部分进行了更新，并添加了几个编译器。

- 首次发布于 2022年3月29日
- CMake 预设的改进，可以包含其他文件。
- 两个新的编译器，以及更好的 C# 支持。
- `FILE_SET` 可用于 `install` 和 `target_sources` 纯头文件库。
- `<INTERFACE_>HEADER_SETS`,`<INTERFACE_>HEADER_DIRS` 为目标头文件。
- 新增 `CUDA_ARCHITECTURES` 对 all 和 all-major.a 的支持
- 可以为 `find_*` 或 find 模块启用 DEBUG 消息。
- `define_property()` 新增了 `INITIALIZE_FROM_VARIABLE` 选项。
- `CMAKE_<SYSTEM_>IGNORE_PREFIX_PATH` 可以控制 `find_*` 的查找路径。
- 新增 `<CMAKE_>LINK_LIBRARIES_ONLY_TARGETS` 强制只链接目标（非常适合查找错误！）
- `IMPORTED_NO_SYSTEM` 可强制从目标中删除 SYSTEM 的新属性。
- `FindGTest` 在找到 `GMock` 目标的情况下，会添加 `GMock` 目标。

CMake 3.24：包查找器

一个很棒的版本。软件包编写者正在实现`find_package`和`FetchContent`的集成，这可以完成“丢失时下载”的工作，并且可以由软件包编写者配置。类似地，作为错误的警告可以由包设置，也可以由打包器删除（最好不要这样做，除非当前项目作为主项目构建！）。

- 首次发布于 2022年8月4日
- `--fresh` 选项在运行时可删除旧缓存。
- `find_package` 和 `FetchContent` 现在集成在一起了 —— 可以选择下载缺失的依赖项。
- `find_package` 新增 `GLOBAL` 选项。
- `CMAKE_PROJECT_TOP_LEVEL_INCLUDES` 允许用户（像打包器一样）注入项目代码。
- 生成器表达式可管理 `PATH`。

- 新增 CMAKE_COLOR_DIAGNOSTICS 环境变量和变量，取代 CMAKE_COLOR_MAKEFILE。
- 可以禁用 find_* 搜索安装前缀（目录）。
- 新增 COMPILE_WARNING_AS_ERROR 属性和 CMAKE_ 变量，可使用 --compile-no-warning-as-error禁用。
- CUDA 支持对当前检测到的 GPU 进行 native 编译。
- SYSTEM 的包含路径可以在 MSVC 生成器上使用。
- 更好地支持 MSVC，XCode 等 IDE 。
- 支持 LLVMFlang 编译器。

CMake 3.25：块作用域和 SYSTEM

新增块作用域指令，可有选择地控制变量和策略，对 SYSTEM 也有更多的控制。可以在find_指令中使用 VALIDATOR 选项，并且工作流程也进行了升级。

- 首次发布于 2022年11月16日
- 支持 C++26
- CUDA 的 nvcc 可以使用 LTO
- 新增了工作流预设和包预设。
- SYSTEM 可作为目录属性添加到 add_subdirectory 和 FetchContent
- block()/endblock() 用于策略/变量范围， return() 中新增 PROPOGATE 选项
- 添加了 BSD 和 LINUX 变量
- find_* 新增 VALIDATOR 选项。
- 新增的 SYSTEM 目标/目录属性和 EXPORT_NO_SYSTEM，同样用于 FetchContent 。

CMake开发中：WIP(Work In Process)

- FindPython 可以生成正确的 PyPy SOABI（终于！）

基础知识简介

基础知识简介

最低版本要求

这是每个 `CMakeLists.txt` 都必须包含的第一行

```
cmake_minimum_required(VERSION 3.1)
```

顺便提一下关于 CMake 的语法。命令 `cmake_minimum_required` 是不区分大小写的，所以常用的做法是使用小写[1]。 `VERSION` 和它后面的版本号是这个函数的特殊关键字。在这本书中，你可以点击命令的名称来查看它的官方文档，并且可以使用下拉菜单来切换 `CMake` 的版本。

这一行很特殊[2]！ `CMake` 的版本与它的特性（policies）相互关联，这意味着它也定义了 `CMake` 行为的变化。因此，如果你将 `cmake_minimum_required` 中的 `VERSION` 设定为 `2.8`，那么你将会在 macOS 上产生链接错误，例如，即使在 `CMake` 最新的版本中，如果你将它设置为 `3.3` 或者更低，那么你将会得到一个隐藏的标志行为(symbols behaviour)错误等。你可以在 policies 中得到一系列 policies 与 versions 的说明。

从 CMake 3.12 开始，版本号可以声明为一个范围，例如 `VERSION 3.1...3.15`；这意味着这个工程最低可以支持 `3.1` 版本，但是也最高在 `3.15` 版本上测试成功过。这对需要更精确(better)设置的用户体验很好，并且由于一个语法上的小技巧，它可以向后兼容更低版本的 CMake （尽管在这里例子中虽然声明为 `CMake 3.1-3.15` 实际只会设置为 `3.1` 版本的特性，因为这些版本处理这个工程没有什么差异）。新的版本特性往往对 macOS 和 Windows 用户是最重要的，他们通常使用非常新版本的 CMake。

当你开始一个新项目，起始推荐这么写：

```
cmake_minimum_required(VERSION 3.7...3.21)

if(${CMAKE_VERSION} VERSION_LESS 3.12)
    cmake_policy(VERSION
${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
endif()
```

如果 CMake 的版本低于3.12，`if` 块条件为真，CMake 将会被设置为当前版本。如果 CMake 版本是 3.12 或者更高，`if` 块条件为假，将会遵守 `cmake_minimum_required` 中的规定，程序将继续正常运行。

WARNING: MSVC 的 CMake 服务器模式起初解析这个语法的时候有一个bug，所以如果你需要支持旧版本的 MSVC 的非命令行的 Windows 构建，你应该这么写：

```
cmake_minimum_required(VERSION 3.7)

if(${CMAKE_VERSION} VERSION_LESS 3.21)
    cmake_policy(VERSION
${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
else()
    cmake_policy(VERSION 3.21)
endif()
```

如果你真的需要在这里设置为一个低版本，你可以使用 `cmake_policy` 来有条件的提高特性级别或者设置一个特殊的特性。请至少为你的 macOS 用户进行设置！

设置一个项目

现在，每一个顶层 CMakelists 文件都应该有下面这一行：

```
project(MyProject VERSION 1.0
                  DESCRIPTION "Very nice project"
                  LANGUAGES CXX)
```

现在我们看到了更多的语法。这里的字符串是带引号的，因此内容中可以带有空格。项目名称是这里的第一个参数。所有的关键字参数都可选的。`VERSION` 设置了一系列变量，例如 `MyProject_VERSION` 和 `PROJECT_VERSION`。语言可以是 `C`,`CXX`,`Fortran`,`ASM`,`CUDA`(CMake 3.8+),`CSharp`(3.8+),`SWIFT`(CMake 3.15+ experimental)，默认是`C CXX`。在 CMake 3.9，可以通过`DESCRIPTION` 关键词来添加项目的描述。这个关于 `project` 的文档可能会有用。

你可以用 `#` 来添加注释。CMake 也有一个用于注释的内联语法，但是那极少用到。

项目名称没有什么特别的用处。这里没有添加任何的目标(target)。

生成一个可执行文件

尽管库要有趣的多，并且我们会将把大部分时间花在其上。但是现在，先让我们从一个简单的可执行文件开始吧！

```
add_executable(one two.cpp three.h)
```

这里有一些语法需要解释。`one` 既是生成的可执行文件的名称，也是创建的 `CMake` 目标(target)的名称(我保证，你很快会听到更多关于目标的内容)。紧接着的是源文件的列表，你想列多少个都可以。CMake 很聪明，它根据拓展名只编译源文件。在大多数情况下，头文件将会被忽略；列出他们的唯一原因是为了让他们在 IDE 中被展示出来，目标文件在许多 IDE 中被显示为文件夹。你可以在 buildsystem 中找到更多关于一般构建系统与目标的信息。

生成一个库

制作一个库是通过 `add_library` 命令完成的，并且非常简单：

```
add_library(one STATIC two.cpp three.h)
```

你可以选择库的类型，可以是 `STATIC`,`SHARED`, 或者`MODULE`.如果你不选择它，CMake 将会通过 `BUILD_SHARED_LIBS` 的值来选择构建 STATIC 还是 SHARED 类型的库。

在下面的章节中你将会看到，你经常需要生成一个虚构的目标，也就是说，一个不需要编译的目标。例如，只有一个头文件的库。这被叫做 `INTERFACE` 库，这是另一种选择，和上面唯一的区别是后面不能有文件名。

你也可以用一个现有的库做一个 `ALIAS` 库，这只是给已有的目标起一个别名。这么做的一个好处是，你可以制作名称中带有 `::` 的库（你将会在后面看到）[3]。

目标时常伴随着你

现在我们已经指定了一个目标，那我们如何添加关于它的信息呢？例如，它可能需要包含一个目录：

```
target_include_directories(one PUBLIC include)
```

`target_include_directories` 为目标添加了一个目录。 `PUBLIC` 对于一个二进制目标没有什么含义；但对于库来说，它让 CMake 知道，任何链接到这个目标

的目标也必须包含这个目录。其他选项还有 PRIVATE（只影响当前目标，不影响依赖），以及 INTERFACE（只影响依赖）。

接下来我们可以将目标之间链接起来：

```
add_library(another STATIC another.cpp another.h)
target_link_libraries(another PUBLIC one)
```

target_link_libraries 可能是 CMake 中最有用也最令人迷惑的命令。它指定一个目标，并且在给出目标的情况下添加一个依赖关系。如果不存在名称为 one 的目标，那他会添加一个链接到你路径中 one 库（这也是命令叫 target_link_libraries 的原因）。或者你可以给定一个库的完整路径，或者是链接器标志。最后再说一个有些迷惑性的知识：），经典的 CMake 允许你省略 PUBLIC 关键字，但是你在目标链中省略与不省略混用，那么 CMake 会报出错误。

只要记得在任何使用目标的地方都指定关键字，那么就不会有问题。

目标可以有包含的目录、链接库（或链接目标）、编译选项、编译定义、编译特性（见C++11 章节）等等。正如你将在之后的两个项目章节中看到的，你经常可以得到目标（并且经常是指定目标）来代表所有你使用的库。甚至有些不是真正的库，像 OpenMP，就可以用目标来表示。这也是为什么现代 CMake 如此的棒！

更进一步

看看你是否能理解以下文件。它生成了一个简单的 C++11 的库并且在程序中使用了它。没有依赖。我将在之后讨论更多的 C++ 标准选项，代码中使用的是 CMake 3.8。

```
cmake_minimum_required(VERSION 3.8)

project(Calculator LANGUAGES CXX)

add_library(calclib STATIC src/calclib.cpp
include/calc/lib.hpp)
target_include_directories(calclib PUBLIC include)
target_compile_features(calclib PUBLIC cxx_std_11)

add_executable(calc apps/calc.cpp)
target_link_libraries(calc PUBLIC calclib)
```

[1]. 在这本书中，我主要避免向你展示错误的做事方式。你可以在网上找到很多关于这个的例子。我偶尔会提到替代方法，但除非是绝对必要，否则不推荐使用这些替代的方法，通常他们只是为了帮助你阅读更旧的 CMake 代码。 ↩

[2]. 有时你会在这里看到 FATAL_ERROR，那是为了支持在 CMake < 2.6 时的错误，现在应该不会有问题了。 ↩

[3]. :: 语法最初是为了 INTERFACE IMPORTED 库准备的，这些库应该是在当前项目之外定义的。但是，因为如此，大多数的 target_* 命令对 IMPORTED 库不起作用，这使得它们难以自己设置。所以，暂时不要使用 IMPORTED 关键字，而使用 ALIAS 目标；它在你开始导出目标之前，都表现的很好。这个限制在 CMake 3.11 中得以修复。 ↩

变量与缓存

变量与缓存

本地变量

我们首先讨论变量。你可以这样声明一个本地 (local) 变量:

```
set(MY_VARIABLE "value")
```

变量名通常全部用大写，变量值跟在其后。你可以通过 `${}` 来解析一个变量，例如 `${MY_VARIABLE}`.[1] CMake 有作用域的概念，在声明一个变量后，你只可以在它的作用域内访问这个变量。如果你将一个函数或一个文件放到一个子目录中，这个变量将不再被定义。你可以通过在变量声明末尾添加 `PARENT_SCOPE` 来将它的作用域置定为当前的上一级作用域。

列表就是简单地包含一系列变量:

```
set(MY_LIST "one" "two")
```

你也可以通过 `;` 分隔变量，这和空格的作用是一样的:

```
set(MY_LIST "one;two")
```

有一些和 `list(` 进行协同的命令， `separate_arguments` 可以把一个以空格分隔的字符串分割成一个列表。需要注意的是，在 CMake 中如果一个值没有空格，那么加和不加引号的效果是一样的。这使你可以在处理知道不可能含有空格的值时不加引号。

当一个变量用 `${}` 括起来的时候，空格的解析规则和上述相同。对于路径来说要特别小心，路径很有可能会包含空格，因此你应该总是将解析变量得到的值用引号括起来，也就是，应该这样 `"${MY_PATH}"` 。

缓存变量

CMake 提供了一个缓存变量来允许你从命令行中设置变量。CMake 中已经有一些预置的变量，像 `CMAKE_BUILD_TYPE` 。如果一个变量还没有被定义，你可以这样声明并设置它。

```
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")
```

这么写不会覆盖已定义的值。这是为了让你只能在命令行中设置这些变量，而不会在 CMake 文件执行的时候被重新覆盖。如果你想把这些变量作为一个临时的全局变量，你可以这样做:

```
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "" FORCE)
mark_as_advanced(MY_CACHE_VARIABLE)
```

第一行将会强制设置该变量的值，第二行将使得用户运行 `cmake -L ..` 或使用 GUI 界面的时候不会列出该变量。此外，你也可以通过 `INTERNAL` 这个类型来达到同样的目的（尽管在技术上他会强制使用 STRING 类型，这不会产生任何的影响）:

```
set(MY_CACHE_VARIABLE "VALUE" CACHE INTERNAL "")
```

因为 `BOOL` 类型非常常见，你可以这样非常容易的设置它:

```
 option(MY_OPTION "This is settable from the command line"
OFF)
```

对于 BOOL 这种数据类型，对于它的 ON 和 OFF 有几种不同的说辞 (wordings) 。

你可以查看 cmake-variables 来查看 CMake 中已知变量的清单。

环境变量

你也可以通过 `set(ENV{variable_name} value)` 和 `$ENV{variable_name}` 来设置和获取环境变量，不过一般来说，我们最好避免这么用。

缓存

缓存实际上就是个文本文件，`CMakeCache.txt` ，当你运行 CMake 构建目录时会创建它。 CMake 可以通过它来记住你设置的所有东西，因此你可以不必在重新运行 CMake 的时候再次列出所有的选项。

属性

CMake 也可以通过属性来存储信息。这就像是一个变量，但它被附加到一些其他的物体 (item) 上，像是一个目录或者是一个目标。一个全局的属性可以是一个有用的非缓存的全局变量。许多目标属性都是被以 `CMAKE_` 为前缀的变量来初始化的。例如你设置 `CMAKE_CXX_STANDARD` 这个变量，这意味着你之后创建的所有目标的 `CXX_STANDARD` 都将被设为`CMAKE_CXX_STANDARD` 变量的值。

你可以这样来设置属性：

```
set_property(TARGET TargetName
             PROPERTY CXX_STANDARD 11)

set_target_properties(TargetName PROPERTIES
                      CXX_STANDARD 11)
```

第一种方式更加通用 (general) ，它可以一次性设置多个目标、文件、或测试，并且有一些非常有用的选项。第二种方式是为一个目标设置多个属性的快捷方式。此外，你可以通过类似于下面的方式来获得属性：

```
get_property(ResultVariable TARGET TargetName PROPERTY
CXX_STANDARD)
```

可以查看 cmake-properties 获得所有已知属性的列表。在某些情况下，你也可以自己定义一些属性[2]。

[1]. `if` 的条件部分语法有一些奇怪，因为 `if` 语法比 `${}` 出现的更早，所以它既可以加 `${}` 也可以不加 `${}`。 ↩

[2]. 对于接口类的目标，可能对允许自定义的属性有一些限制。 ↩

用 CMake 进行编程

用 CMake 进行编程

控制流程

CMake 有一个 `if` 语句，尽管经过多次版本迭代它已经变得非常复杂。这里有一些全大写的变量你可以在 `if` 语句中使用，并且你既可以直接引用也可以利用 `${}` 来对他进行解析（ `if` 语句在历史上比变量拓展出现的更早 ）。这是一个 `if` 语句的例子：

```
if(variable)
    # If variable is `ON`, `YES`, `TRUE`, `Y`, or non zero
number
else()
    # If variable is `0`, `OFF`, `NO`, `FALSE`, `N`,
`IGNORE`, `NOTFOUND`, `""`, or ends in `-NOTFOUND`
endif()
# If variable does not expand to one of the above, CMake
will expand it then try again
```

如果你在这里使用 `${variable}` 可能会有一些奇怪，因为看起来它好像 `variable` 被展开 (expansion) 了两次。在 CMake 3.1+ 版本中加入了一个新的特性 (CMP0054)，CMake 不会再展开已经被引号括起来的展开变量。也就是说，如果你的 CMake 版本大于 `3.1` ，那么你可以这么写：

```
if("${variable}")
    # True if variable is not false-like
else()
    # Note that undefined variables would be `""` thus false
endif()
```

这里还有一些关键字可以设置，例如：

- 一元的: `NOT`, `TARGET`, `EXISTS` (文件), `DEFINED`, 等。
- 二元的: `STREQUAL`, `AND`, `OR`, `MATCHES` (正则表达式), `VERSION_LESS`, `VERSION_LESS_EQUAL` (CMake 3.7+), 等。
- 括号可以用来分组

generator-expressions

generator-expressions 语句十分强大，不过有点奇怪和专业 (specialized)。大多数 CMake 命令在配置的时候执行，包括我们上面看到的 `if` 语句。但是如果你想要他们在构建或者安装的时候运行呢，应该怎么写？ 生成器表达式就是为此而生[1]。它们在目标属性中被评估（ evaluate ）：

最简单的生成器表达式是信息表达式，其形式为 `$<KEYWORD>`；它会评估和当前配置相关的一系列信息。信息表达式的另一个形式是 `$<KEYWORD:value>`，其中 `KEYWORD` 是一个控制评估的关键字，而 `value` 则是被评估的对象（ 这里的 `value` 中也允许使用信息表达式，如下面的 `${CMAKE_CURRENT_SOURCE_DIR}/include` ）。如果 `KEYWORD` 是一个可以被评估为0或1的生成器表达式或者变量，如果（`KEYWORD`被评估）为1则 `value` 会在这里被保留下来，而反之则不会。你可以使用嵌套的生成器表达式，你也可以使用变量来使得自己更容易理解嵌套的变量。一些表达式也可以有多个值，值之间通过逗号分隔[2]。

如果你有一个只想在配置阶段的 DEBUG 模式下开启的编译标志（ flag ），你可以这样做：

```
target_compile_options(MyTarget PRIVATE "$<$<CONFIG:Debug>:--my-flag>")
```

译者注：这里有点迷惑性，这里其实包含了两种 generator-expression，分别是 configuration-expression 和 conditional-expression，前者使用的形式是 `$<CONFIG:cfgs>`，这里的 cfgs 是一个 List，如果 CONFIG 满足 cfgs 列表中的任何一个值，这个表达式会被评估（evaluate）为 1，否则为 0。后者使用的形式是 `$<condition:true_string>`，如果 condition 值为 1，则表达式被评估为 true_string，否则为空值。因此这里表达的含义是，如果这里是一个 DEBUG 的 configuration，就设置 --my-flag。可参见官方文档。

这是一个相比与指定一些形如 `*_DEBUG` 这样的变量更加新颖并且更加优雅的方式，并且这对所有支持生成器表达式的设置都通用。需要注意的是，你永远不要在配置阶段（configuration phase）使用配置有关值（configure time value），因为在使用像 IDE 这种多配置生成器时你没法在配置阶段获取到这些值，只有在构建阶段使用生成器表达式或者形如 `*_<CONFIG>` 的变量才能获得。

一些生成器表达式的其他用途：

- 限制某个项目的语言，例如可以限制其语言为 CXX 来避免它和 CUDA 等语言混在一起，或者可以通过封装它来使得他对不同的语言有不同的表现。
- 获得与属性相关的配置，例如文件的位置。
- 为构建和安装生成不同的位置。

最后一个是常见的。你几乎会在所有支持安装的软件包中看到如下代码：

译者注：表示在目标对于直接 BUILD 使用的目标包含的头文件目录为 `${CMAKE_CURRENT_SOURCE_DIR}/include`，而安装的目标包含的头文件目录为 `include`，是一个相对位置（同时需要 install 对应的头文件才可以）。

```
target_include_directories(
    MyTarget
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)
```

宏定义与函数

你可以轻松地定义你自己的 CMake `function` 或 `macro` 。函数和宏只有作用域上存在区别，宏没有作用域的限制。所以说，如果你想让函数中定义的变量对外部可见，你需要使用 `PARENT_SCOPE` 来改变其作用域。如果是在嵌套函数中，这会变得异常繁琐，因为你必须在想要变量对外的可见的所有函数中添加 `PARENT_SCOPE` 标志。但是这样也有好处，函数不会像宏那样对外“泄漏”所有的变量。接下来用函数举一个例子：

下面十一个简单的函数的例子：

```
function(SIMPLE REQUIRED_ARG)
    message(STATUS "Simple arguments: ${REQUIRED_ARG},
followed by ${ARGN}")
    set(${REQUIRED_ARG} "From SIMPLE" PARENT_SCOPE)
endfunction()

simple(This Foo Bar)
message("Output: ${This}")
```

输出如下：

```
-- Simple arguments: This, followed by Foo;Bar
Output: From SIMPLE
```

如果你想要有一个指定的参数，你应该在列表中明确的列出，除此之外的所有参数都会被存储在 ARGN 这个变量中（ ARGV 中存储了所有的变量，包括你明确列出的）。CMake 的函数没有返回值，你可以通过设定变量值的形式来达到同样地目的。在上面的例子中，你可以通过指定变量名来设置一个变量的值。

参数的控制

你应该已经在很多 CMake 函数中见到过，CMake 拥有一个变量命名系统。你可以通过 cmake_parse_arguments 函数来对变量进行命名与解析。如果你想在低于 3.5 版本的CMake 系统中使用它，你应该包含 CMakeParseArguments 模块，此函数在 CMake 3.5 之前一直存在与上述模块中。这是使用它的一个例子：

```
function(COMPLEX)
    cmake_parse_arguments(
        COMPLEX_PREFIX
        "SINGLE;ANOTHER"
        "ONE_VALUE;ALSO_ONE_VALUE"
        "MULTI_VALUES"
        ${ARGN}
    )
endfunction()

complex(SINGLE ONE_VALUE value MULTI_VALUES some other
values)
```

在调用这个函数后，会生成以下变量：

```
COMPLEX_PREFIX_SINGLE = TRUE
COMPLEX_PREFIX_ANOTHER = FALSE
COMPLEX_PREFIX_ONE_VALUE = "value"
COMPLEX_PREFIX_ALSO_ONE_VALUE = <UNDEFINED>
COMPLEX_PREFIX_MULTI_VALUES = "some;other;values"
```

如果你查看了官方文档，你会发现可以通过 set 来避免在 list 中使用分号，你可以根据个人喜好来确定使用哪种结构。你可以在上面列出的位置参数中混用这两种写法。此外，其他剩余的参数（因此参数的指定是可选的）都会被保存在 COMPLEX_PREFIX_UNPARSED_ARGUMENTS 变量中。

[1]. 他们看起来像是在构建或安装时被评估的，但实际上他们只对每个构建中的配置进行评估。 ↩

[2]. CMake 官方文档中将表达式分为信息表达式，逻辑表达式和输出表达式。 ↩

与你的代码交互

与你的代码交互

通过 CMake 配置文件

CMake 允许你在代码中使用 `configure_file` 来访问 CMake 变量。该命令将一个文件（一般以 `.in` 结尾）的内容复制到另一个文件中，并替换其中它找到的所有 CMake 变量。如果你想要在你的输入文件中避免替换掉使用 `${}` 包含的内容，你可以使用 `@ONLY` 关键字。还有一个关键字 `COPY_ONLY` 可以用来作为 `file(COPY` 的替代字。

这个功能在 CMake 中使用的相当频繁，例如在下面的 `Version.h.in` 中：

Version.h.in

```
#pragma once

#define MY_VERSION_MAJOR @PROJECT_VERSION_MAJOR@
#define MY_VERSION_MINOR @PROJECT_VERSION_MINOR@
#define MY_VERSION_PATCH @PROJECT_VERSION_PATCH@
#define MY_VERSION_TWEAK @PROJECT_VERSION_TWEAK@
#define MY_VERSION "@PROJECT_VERSION@"
```

CMake lines:

```
configure_file (
    "${PROJECT_SOURCE_DIR}/include/My/Version.h.in"
    "${PROJECT_BINARY_DIR}/include/My/Version.h"
)
```

在构建你的项目时，你也应该包括二进制头文件路径。如果你想要在头文件中包含一些 `true/false` 类型的变量，CMake 对 C 语言有特有的 `#cmakedefine` 和 `#cmakedefine01` 替换符来完成上述需求。

你也可以使用（并且是常用）这个来生成 `.cmake` 文件，例如配置文件（见 installing）。

读入文件

另外一个方向也是行得通的，你也可以从源文件中读取一些东西（例如版本号）。例如，你有一个仅包含头文件的库，你想要其在无论有无 CMake 的情况下都可以使用，上述方式将是你处理版本的最优方案。可以像下面这么写：

```
# Assuming the canonical version is listed in a single line
# This would be in several parts if picking up from MAJOR,
MINOR, etc.
set(VERSION_REGEX "#define MY_VERSION[ \t]+\"(.+)\"")

# Read in the line containing the version
file(STRINGS
"${CMAKE_CURRENT_SOURCE_DIR}/include/My/Version.hpp"
    VERSION_STRING REGEX ${VERSION_REGEX})

# Pick out just the version
string(REGEX REPLACE ${VERSION_REGEX} "\\1" VERSION_STRING
"${VERSION_STRING}")
```

```
# Automatically getting PROJECT_VERSION_MAJOR,
My_VERSION_MAJOR, etc.
project(My LANGUAGES CXX VERSION ${VERSION_STRING})
```

如上所示，`file(STRINGS file_name variable_name REGEX regex)` 选择了与正则表达式相匹配的行，并且使用了相同的正则表达式来匹配出其中版本号的部分。

如何组织你的项目

如何组织你的项目

下面的说法可能存在一些偏见，但我认为这是一种好的组织方式。我将会讲解如何组织项目的目录结构，这是基于以往的惯例来写的，这么做对你有以下好处：

- 可以很容易阅读以相同模式组织的项目
- 避免可能造成冲突的组织形式
- 避免使目录结构变得混乱和复杂

首先，如果你创建一个名为 `project` 的项目，它有一个名为 `lib` 的库，有一个名为 `app` 的可执行文件，那么目录结构应该如下所示：

```
- project
  - .gitignore
  - README.md
  - LICENCE.md
  - CMakeLists.txt
  - cmake
    - FindSomeLib.cmake
    - something_else.cmake
  - include
    - project
      - lib.hpp
  - src
    - CMakeLists.txt
    - lib.cpp
  - apps
    - CMakeLists.txt
    - app.cpp
  - tests
    - CMakeLists.txt
    - testlib.cpp
  - docs
    - CMakeLists.txt
  - extern
    - googletest
  - scripts
    - helper.py
```

其中，文件的名称不是绝对的，你可能会看到关于文件夹名称为 `tests` 还是 `test` 的争论，并且应用程序所在的文件夹可能为其他的名称（或者一个项目只有库文件）。你也许也会看到一个名为 `python` 的文件夹，那里存储关于 python 绑定器的内容，或者是一个 `cmake` 文件夹用于存储如 `Find<library>.cmake` 这样的 `.cmake` 辅助文件。但是一些比较基础的东西都在上面包括了。

可以注意到一些很明显的问题，`CMakeLists.txt` 文件被分割到除了 `include` 目录外的所有源代码目录下。这是为了能够将 `include` 目录下的所有文件拷贝到 `/usr/include` 目录或其他类似的目录下（除了配置的头文件，这个我将会在另一章讲到），因此为了避免冲突等问题，其中不能有除了头文件外的其他文件。这也是为什么在 `include` 目录下有一个名为项目名的目录。顶层 `CMakeLists.txt` 中应使用 `add_subdirectory` 命令来添加一个包含 `CMakeLists.txt` 的子目录。

你经常会需要一个 `cmake` 文件夹，里面包含所有用到的辅助模块。这是你放置所有 `Find*.cmake` 的文件。你可以在 github.com/CLIUtils/cmake 找到一些常见的辅助模块集合。你可以通过以下语句将此目录添加到你的 CMake Path 中：

```
set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake"
${CMAKE_MODULE_PATH})
```

你的 `extern` 应该几乎只包含 git 子模块（ submodule ）。通过此方式，你可以明确地控制依赖的版本，并且可以非常轻松地升级。关于添加子模块的例子，可以参见 Testing 章节。

你应该在 `.gitignore` 中添加形如 `/build*` 的规则，这样用户就可以在源代码目录下创建 `build` 目录来构建项目，而不用担心将生成的目标文件添加到 `.git` 中。有一些软件包禁止这么做，不过这还是相比**做一个真正的外部构建并且针对不同的包来使用不同的构建**要好的多。

如果你想要避免构建目录在有效的（ valid ）源代码目录中，你可以在顶层 `CMakeLists.txt` 文件头部添加如下语句：

```
### Require out-of-source builds
file(TO_CMAKE_PATH "${PROJECT_BINARY_DIR}/CMakeLists.txt"
LOC_PATH)
if(EXISTS "${LOC_PATH}")
    message(FATAL_ERROR "You cannot build in a source
directory (or any directory with a CMakeLists.txt file).
Please make a build subdirectory. Feel free to remove
CMakeCache.txt and CMakeFiles.")
endif()
```

可以在这里查看 拓展代码样例

在 CMake 中运行其他程序

在 CMake 中运行其他的程序

在配置时运行一条命令

在配置时运行一条命令是相对比较容易的。可以使用 execute_process 来运行一条命令并获得他的结果。一般来说，在 CMake 中避免使用硬编码路径是一个好的习惯，你也可以使用 ${CMAKE_COMMAND} , find_package(Git) , 或者 find_program 来获取命令的运行权限。可以使用 RESULT_VARIABLE 变量来检查返回值，使用 OUTPUT_VARIABLE 来获得命令的输出。

下面是一个更新所有 git 子模块的例子：

```
find_package(Git QUIET)

if(GIT_FOUND AND EXISTS "${PROJECT_SOURCE_DIR}/.git")
    execute_process(COMMAND ${GIT_EXECUTABLE} submodule
update --init --recursive
                    WORKING_DIRECTORY
${CMAKE_CURRENT_SOURCE_DIR}
                    RESULT_VARIABLE GIT_SUBMOD_RESULT)
    if(NOT GIT_SUBMOD_RESULT EQUAL "0")
        message(FATAL_ERROR "git submodule update --init --
recursive failed with ${GIT_SUBMOD_RESULT}, please checkout
submodules")
    endif()
endif()
```

在构建时运行一条命令

在构建时运行一条命令有点难。主要是目标系统（target system）使这变的很难，你希望你的命令在什么时候运行？它是否会产生另一个目标需要的输出？记住这些需求，然后我们来看一个关于调用 Python 脚本生成头文件的例子：

```
find_package(PythonInterp REQUIRED)
add_custom_command(OUTPUT
"${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp"
    COMMAND "${PYTHON_EXECUTABLE}"
"${CMAKE_CURRENT_SOURCE_DIR}/scripts/GenerateHeader.py" --
argument
    DEPENDS some_target)

add_custom_target(generate_header ALL
    DEPENDS
"${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp")

install(FILES
${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp
DESTINATION include)
```

在这里，当你在 add_custom_target 命令中添加 ALL 关键字，头文件的生成过程会在 some_target 这些依赖目标完成后自动执行。当你把这个目标作为另一个目标的依赖，你也可以不加 ALL 关键字，那这样他会在被依赖目标构建时会自动执行。或者，你也可以显示的直接构建 generate_header 这个目标。

CMake 中包含的常用的工具

在编写跨平台的 CMake 工程时，一个有用的工具是 `cmake -E <mode>`（在 `CMakeLists.txt` 中被写作 `${CMAKE_COMMAND} -E`）。通过指定后面的 `<mode>` 允许 CMake 在不显式调用系统工具的情况下完成一系列事情，例如 `copy(复制)`，`make_directory(创建文件夹)`，和 `remove(移除)`。这都是构建时经常使用的命令。 需要注意的是，一个非常有用的 mode——`create_symlink`，只有在基于 Unix 的系统上可用，但是在 CMake 3.13 后的 Windows 版本中也存在此 `mode`。点击这里查看对应文档。

一个简单的例子

一个简单的例子

这是一个简单、完整并且合理的 `CMakeLists.txt` 的例子。对于这个程序，我们有一个带有头文件与源文件的库文件（ MyLibExample ），以及一个带有源文件的应用程序（ MyExample ）。

```
# Almost all CMake files should start with this
# You should always specify a range with the newest
# and oldest tested versions of CMake. This will ensure
# you pick up the best policies.
cmake_minimum_required(VERSION 3.1...3.21)

# This is your project statement. You should always list
languages;
# Listing the version is nice here since it sets lots of
useful variables
project(
  ModernCMakeExample
  VERSION 1.0
  LANGUAGES CXX)

# If you set any CMAKE_ variables, that can go here.
# (But usually don't do this, except maybe for C++ standard)

# Find packages go here.

# You should usually split this into folders, but this is a
simple example

# This is a "default" library, and will match the ***
variable setting.
# Other common choices are STATIC, SHARED, and MODULE
# Including header files here helps IDEs but is not
required.
# Output libname matches target name, with the usual
extensions on your system
add_library(MyLibExample simple_lib.cpp simple_lib.hpp)

# Link each target with other targets or add options, etc.

# Adding something we can run - Output name matches target
name
add_executable(MyExample simple_example.cpp)

# Make sure you link your targets with this command. It can
also link libraries and
# even flags, so linking a target that does not exist will
not give a configure-time error.
target_link_libraries(MyExample PRIVATE MyLibExample)
```

完整的例子可以在此查看 examples folder.

一个更大，并且包含多文件的例子可在此查看 also available.

为 CMake 项目添加特性

为 CMake 项目添加特性

本节将会涵盖如何为你的 CMake 项目添加特性。你将会学到如何为你的 C++ 项目添加一些常用的选项，如 C++11 支持，以及如何支持 IDE 工具等。

默认的构建类型

CMake 通常会设置一个 “既不是 Release 也不是Debug” 的空构建类型来作为默认的构建类型，如果你想要自己设置默认的构建类型，你可以参考 Kitware blog 中指出的方法。

```
set(default_build_type "Release")
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  message(STATUS "Setting build type to
'${default_build_type}' as none was specified.")
  set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
      STRING "Choose the type of build." FORCE)
  # Set the possible values of build type for cmake-gui
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS
    "Debug" "Release" "MinSizeRel" "RelWithDebInfo")
endif()
```

C++11 及后续版本

C++11 及后续版本

CMake 中支持 C++11，但是这是针对于 CMake 2.8 及以后的版本来说的。这是为什么？很容易可以猜到， C++11 在 2009年——CMake 2.0 发布的时候还不存在。只要你使用 CMake 的是 CMake 3.1 或者更新的版本，你将会得到 C++11 的完美支持，不过这里有两种不同的方式来启用支持。 并且你将看到，在 CMake 3.8+ 中对 C++11 有着更好的支持。我将会在 CMake 3.8+ 的基础上讲解，因为这才叫做 Modern CMake。

CMake 3.8+: 元编译器选项

只要你使用新版的 CMake 来组织你的项目，那你就能够使用最新的方式来启用 C++ 的标准。这个方式功能强大，语法优美，并且对最新的标准有着很好的支持。此外，它对目标 (target) 进行混合标准与选项设置有着非常优秀的表现。假设你有一个名叫 `myTarget` 的目标，它看起来像这样：

```
target_compile_features(myTarget PUBLIC cxx_std_11)
set_target_properties(myTarget PROPERTIES CXX_EXTENSIONS
OFF)
```

对于第一行，我们可以在 `cxx_std_11`、`cxx_std_14` 和 `cxx_std_17` 之间选择。第二行是可选的，但是添加了可以避免 CMake 对选项进行拓展。如果不添加它，CMake 将会添加选项 `-std=g++11` 而不是 `-std=c++11` 。第一行对 `INTERFACE` 这种目标 (target) 也会起作用，第二行只会对实际被编译的目标有效。

如果在目标的依赖链中有目标指定了更高的 C++ 标准，上述代码也可以很好的生效。这只是下述方法的一个更高级的版本，因此可以很好的生效。

CMake 3.1+: 编译器选项

你可以指定开启某个特定的编译器选项。这相比与直接指定 C++ 编译器的版本更加细化，尽管去指定一个包使用的所有编译器选项可能有点困难，除非这个包是你自己写的或者你的记忆力非凡。最后 CMake 会检查你编译器支持的所有选项，并默认设置使用其中每个最新的版本。因此，你不必指定所有你需要的选项，只需要指定那些和默认有出入的。设置的语法和上一部分相同，只是你需要挑选一个列表里面存在的选项而不像是 `cxx_std_*` 。这里有包含所有选项的列表。

如果你需要可选的选项，在 CMake 3.3+ 中你可以使用列表 `CMAKE_CXX_COMPILE_FEATURES` 及 `if(... INLIST ...)` 来查看此选项是否在此项目中被选用，然后来决定是否添加它。可以 在此 查看一些其他的使用情况。

CMake 3.1+: 全局设置以及属性设置

这是支持 C++ 标准的另一种方式，（在目标及全局级别）设置三个特定属性的值。这是全局的属性：

```
set(CMAKE_CXX_STANDARD 11 CACHE STRING "The C++ standard to
use")
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

第一行设置了 C++ 标准的级别， 第二行告诉 CMake 使用上述设置， 最后一行关闭了拓展，来明确自己使用了 `-std=c++11` 还是 `-std=g++11` 。这个方法中可以在最终包 (final package) 中使用，但是不推荐在库中使用。你应该总是把它设置为一个缓存变量，这样你就可以很容易地重写其内容来尝试新的标准（或者如果你在库中使用它的话，这是重写它的唯一方式。**不过再重申一遍**，不要在库中使用此方式）。你也可以对目标来设置这些属性：

```
set_target_properties(myTarget PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO
)
```

这种方式相比于上面来说更好，但是仍然没法对 `PRIVATE` 和 `INTERFACE` 目标的属性有明确的控制，所以他们也仍然只对最终目标 (final targets) 有用。

你可以在 Craig Scott's useful blog post 这里找到更多关于后面两种方法的信息。

不要自己设置手动标志。如果这么做，你必须对每个编译器的每个发行版设置正确的标志，你无法通过不支持的编译器的报错信息来解决错误，并且 IDE 可能不会去关心手动设置的标志。

一些小而常见的需求

为 CMake 项目添加选项

CMake 中有许多关于编译器和链接器的设置。当你需要添加一些特殊的需求，你应该首先检查 CMake 是否支持这个需求，如果支持的话，你就可以不用关心编译器的版本，一切交给 CMake 来做即可。 更好的是，你可以在 `CMakeLists.txt` 表明你的意图，而不是通过开启一系列标志 (flag) 。

其中最首要，并且最普遍的需求是对 C++ 标准的设定与支持，这个将会单独开一章节讲解。

地址无关代码(Position independent code)

用标志 `-fPIC` 来设置这个是最常见的。大部分情况下，你不需要去显式的声明它的值。CMake 将会在 `SHARED` 以及 `MODULE` 类型的库中自动的包含此标志。如果你需要显式的声明，可以这么写:

```
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
```

这样会对全局的目标进行此设置，或者可以这么写:

```
set_target_properties(lib1 PROPERTIES
POSITION_INDEPENDENT_CODE ON)
```

来对某个目标进行设置是否开启此标志。

Little libraries

如果你需要链接到 `dl` 库，在 Linux 上可以使用 `-ldl` 标志，不过在 CMake 中只需要在 `target_link_libraries` 命令中使用内置的 CMake 变量 `${CMAKE_DL_LIBS}`。这里不需要模组或者使用 `find_package` 来寻找它。（这个命令包含了调用 `dlopen` 与 `dlclose` 的一切依赖）

不幸的是，想要链接到数学库没这么简单。如果你需要明确地链接到它，你可以使用 `target_link_libraries(MyTarget PUBLIC m)`，但是使用 CMake 通用的 `find_library` 可能更好，如下是一个例子:

```
find_library(MATH_LIBRARY m)
if(MATH_LIBRARY)
    target_link_libraries(MyTarget PUBLIC ${MATH_LIBRARY})
endif()
```

通过快速搜索，你可以很容易地找到这个和其他你需要的库的 `Find*.cmake` 文件，大多数主要软件包都具有这个 CMake 模组的辅助库。更多信息请参见**包含现有软件包**的章节。

程序间优化(Interprocedural optimization)

`INTERPROCEDURAL_OPTIMIZATION`，最有名的是 *链接时间优化* 以及 `-flto` 标志，这在最新的几个 CMake 版本中可用。你可以通过变量 `CMAKE_INTERPROCEDURAL_OPTIMIZATION`（CMake 3.9+ 可用）或对目标指定 `INTERPROCEDURAL_OPTIMIZATION` 属性来打开它。在 CMake 3.8 中添加了对 GCC 及 Clang 的支持。如果你设置了 `cmake_minimum_required(VERSION 3.9)` 或者更高的版本（参考 CMP0069），当在编译器不支持 `INTERPROCEDURAL_OPTIMIZATION` 时，通过变量或属性启用该优化会产生报

错。你可以使用内置模块 CheckIPOSupported 中的 `check_ipo_supported()` 来检查编译器是否支持 IPO 。下面是基于 CMake 3.9 的一个例子：

```
include(CheckIPOSupported)
check_ipo_supported(RESULT result)
if(result)
  set_target_properties(foo PROPERTIES
INTERPROCEDURAL_OPTIMIZATION TRUE)
endif()
```

一些实用的工具

CCache 和一些其他的实用工具

在过去的一些版本中，一些能够帮助你写好代码的实用工具已经被添加到了 CMake 中。往往是通过为目标指定属性，或是设定形如 `CMAKE_*` 的初始化变量的值的形式启用相应工具。这个启用的规则不只是对某个特定的工具（program）起作用，一些行为相似的工具都符合此规则。

当需要启用多个工具时，所有的这些变量都通过 `;` 分隔（CMake 中列表的分隔标准）来描述你在目标源程序上需要使用的工具(program) 以及选项。

CCache[1]

通过设置变量 `CMAKE_<LANG>_COMPILER_LAUNCHER` 或设置目标的 `<LANG>_COMPILER_LAUNCHER` 属性来使用一些像 CCache 的方式来“封装”目标的编译。在 CMake 的最新版本中拓展了对 CCache 的支持。在使用时，可以这么写：

```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
    set(CMAKE_CXX_COMPILER_LAUNCHER "${CCACHE_PROGRAM}")
    set(CMAKE_CUDA_COMPILER_LAUNCHER "${CCACHE_PROGRAM}") #
CMake 3.9+
endif()
```

一些实用工具

设置以下属性或是在命令行中设置以 `CMAKE_*` 为起始的变量来启动这些功能。它们大部分只在 make 或 ninja 生成器生成 C 和 CXX 项目时起作用。

- `<LANG>_CLANG_TIDY`: CMake 3.6+
- `<LANG>_CPPCHECK`
- `<LANG>_CPPLINT`
- `<LANG>_INCLUDE_WHAT_YOU_USE`

Clang tidy[2]

这是在命令行中运行 clang-tidy 的方法，使用的是一个列表（记住，用分号分隔的字符串是一个列表）。

这是一个使用 Clang-Tidy 的简单例子：

```
~/package # cmake -S . -B build-tidy -
DCMAKE_CXX_CLANG_TIDY="$(which clang-tidy);-fix"
~/package # cmake --build build -j 1
```

这里的 `-fix` 部分是可选的，将会修改你的源文件来尝试修复 clang-tidy 警告 (warning) 的问题。如果你在一个 git 仓库中工作的话，使用 `-fix` 是相当安全的，因为你可以看到代码中哪部分被改变了。不过，请确保**不要同时运行你的 makefile/ninja 来进行构建**！如果它尝试修复一个相同的头文件两次，可能会出现预期外的错误。

如果你想明确的使用目标的形式来确保自己对某些特定的目标调用了 clang-tidy，为可以设置一个变量（例如像 `DO_CLANG_TIDY`，而不是名为 `CMAKE_CXX_CLANG_TIDY` 的变量），然后在创建目标时，将它添加为目标的属性。你可以通过以下方式找到路径中的 clang-tidy：

```
find_program(
    CLANG_TIDY_EXE
    NAMES "clang-tidy"
    DOC "Path to clang-tidy executable"
)
```

Include what you use[3]

这是一个使用 include what you use 的例子。首先，你需要确保系统中有这个工具，例如在一个 docker 容器中或者通过 macOS 上的 brew 利用 brew install include-what-you-use 来安装它。然后，你可以通过此方式使用此工具，而不需要修改你的源代码：

```
~/package # cmake -S . -B build-iwyu -
DCMAKE_CXX_INCLUDE_WHAT_YOU_USE=include-what-you-use
```

最后，你可以重定向输出到文件，然后选择是否应用此修复：

```
~/package # cmake --build build-iwyu 2> iwyu.out
~/package # fix_includes.py < iwyu.out
```

（你应该先检查一下这些修复的正确性，或者在修复后对代码进行润色！）

Link what you use

这是一个布尔类型的目标属性，LINK_WHAT_YOU_USE，它将会在链接时检查与目标不相干的文件。

Clang-format[4]

不幸的是，Clang-format 并没有真正的与 CMake 集成。你可以制作一个自定义的目标（参考 这篇文章，或者你可以尝试自己手动的去运行它。）一个有趣的项目/想法 在这里，不过我还没有亲自尝试过。它添加了一个格式化 (format) 的目标，并且你甚至没法提交没有格式化过的文件。

下面的两行可以在一个 git 仓库中，在 bash 中使用 clang-format 工具（假设你有一个 .clang-format 文件）：

```
gitbook $ git ls-files -- '*.cpp' '*.h' | xargs clang-format
-i -style=file
gitbook $ git diff --exit-code --color
```

译者注：以下所有的脚注说明都为译者添加，原文并不包含此信息。脚注的说明资料均来自于互联网。

[1]. Ccache（或 ”ccache“）是一个编译器缓存。它通过缓存之前的编译文件并且利用之前已经完成的编译过程来加速重编译)。Ccache是一个免费的软件，基于 GNU General Public License version 3 或之后更新的许可协议发布。可以查看这里的 许可协议页面 。 ↩

[2]. **clang-tidy** 是一个基于 clang 的 C++ 代码分析工具。它意图提供一个可扩展的框架，用于诊断和修复典型的编程错误，如样式违规、接口误用、或通过静态分析推断出的错误。**clang-tidy** 是一个模块化的程序，为编写新的检查规则提供了方便的接口。 ↩

[3]. 一个与 **clang** 一起使用，用于分析 C 和 C++ 源文件中 **#include** 的工具。 ↩

[4]. ClangFormat 描述了一套建立在 LibFormat 之上的工具。它可以以各种方式支持你的工作流程，包括独立的工具和编辑器的集成。 ↩

一些有用的模组

CMake 中一些有用的模组

在 CMake 的 modules 集合了很多有用的模组，但是有一些模块相比于其他的更有用。以下是一些比较出彩的：

CMakeDependentOption

这增加了命令 `cmake_dependent_option`，它根据另外一组变量是否为真来（决定是否）开启一个选项。下面是一个例子：

```
include(CMakeDependentOption)
cmake_dependent_option(BUILD_TESTS "Build your tests" ON
"VAL1;VAL2" OFF)
```

如上代码是下面的一个缩写：

```
if(VAL1 AND VAL2)
    set(BUILD_TESTS_DEFAULT ON)
else()
    set(BUILD_TESTS_DEFAULT OFF)
endif()

option(BUILD_TESTS "Build your tests"
${BUILD_TESTS_DEFAULT})

if(NOT BUILD_TESTS_DEFAULT)
    mark_as_advanced(BUILD_TESTS)
endif()
```

需要注意的是，如果你使用了 `include(CTest)`，用 `BUILD_TESTING` 来检测是否启用是更好的方式，因为它就是为此功能而生的。这里只是一个 `CMakeDependentOption` 的例子。

CMakePrintHelpers

这个模块包含了几个方便的输出函数。`cmake_print_properties` 可以让你轻松的打印属性，而 `cmake_print_variables` 将打印出你给它任意变量的名称和值。

CheckCXXCompilerFlag

这个模块允许你检查编译器是否支持某个标志，例如：

```
include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(-someflag OUTPUT_VARIABLE)
```

需要注意的是 `OUTPUT_VARIABLE` 也会出现在打印的配置输出中，所以请选个不错的变量名。

这只是许多类似模块中的一个，例如 `CheckIncludeFileCXX`、`CheckStructHasMember`、`TestBigEndian` 以及`CheckTypeSize`，它们允许你检查系统的信息（并且你可以在代码中使用这些信息）。

try_compile/try_run

准确的说，这不是一个模块，但是它们对上述列出的许多模块至关重要。通过它你可以在配置时尝试编译（也可能是运行）一部分代码。这可以让你在配置时获取关于系统能力的信息。基本的语法如下：

```
try_compile(
  RESULT_VAR
    bindir
  SOURCES
    source.cpp
)
```

这里有很多可以添加的选项，例如 `COMPILE_DEFINITIONS`。在 CMake 3.8+ 中，这将默认遵循 CMake 中 C/C++/CUDA 的标准设置。如果你使用的是 `try_run` 而不是 `try_compile`，它将运行生成的程序并将运行结果存储在 `RUN_OUTPUT_VARIABLE` 中。

FeatureSummary

这是一个十分有用但是也有些奇怪的模块。它能够让你打印出找到的所有软件包以及你明确设定的所有选项。它和 `find_package` 有一些联系。像其他模块一样，你首先要包括模块：

```
include(FeatureSummary)
```

然后，对于任何你已经运行或者将要运行的 `find_package`，你可以这样拓展它的默认信息：

```
set_package_properties(OpenMP PROPERTIES
    URL "http://www.openmp.org"
    DESCRIPTION "Parallel compiler directives"
    PURPOSE "This is what it does in my package")
```

你也可以将包的 `TYPE` 设置为 `RUNTIME`、`OPTIONAL`、`RECOMMENDED` 或者 `REQUIRED`。但是你不能降低包的类型，如果你已经通过 `find_package` 添加了一个 `REQUIRED` 类型的包，你将会看到你不能改变它的 `TYPE`：

并且，你可以添加任何选项让其成为 `feature summary` 的一部分。如果你添加的选项名与包的名字一样，他们之间会互相产生影响：

```
add_feature_info(WITH_OPENMP OpenMP_CXX_FOUND "OpenMP
(Thread safe FCNs only)")
```

然后，你可以将所有特性 (features) 的集合打印到屏幕或日志文件中：

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    feature_summary(WHAT ENABLED_FEATURES DISABLED_FEATURES
PACKAGES_FOUND)
    feature_summary(FILENAME
${CMAKE_CURRENT_BINARY_DIR}/features.log WHAT ALL)
endif()
```

你可以建立一个 `WHAT` 目标来集合任何你想查看的特性 (features)，或者直接使用 `ALL` 目标也行。

CMake 对 IDE 的支持

CMake 对 IDE 的支持

一般来说，IDE 已经被标准的 CMake 的项目支持。不过这里有一些额外的东西可以帮助 IDE 表现得更好：

用文件夹来组织目标 (target)

一些 IDE，例如 Xcode，支持文件夹。你需要手动的设定 `USE_FOLDERS` 这个全局属性来允许 CMake 使用文件夹组织你的文件：

```
set_property(GLOBAL PROPERTY USE_FOLDERS ON)
```

然后，你可以在创建目标后，为目标添加文件夹属性，即将其目标 `MyFile` 归入到 `Scripts` 文件夹中：

```
set_property(TARGET MyFile PROPERTY FOLDER "Scripts")
```

文件夹可以使用 `/` 进行嵌套。

你可以使用正则表达式或在 `source_group` 使用列表来控制文件在文件夹中是否可见。

用文件夹来组织文件

你也可以控制文件夹对目标是否可见。有两种方式，都是使用 `source_group` 命令，传统的方式是：

```
source_group("Source Files\\New Directory"
REGULAR_EXPRESSION ".*\\.c[ucp]p?")
```

你可以用 `FILES` 来明确的列出文件列表，或者使用 `REGULAR_EXPRESSION` 来进行筛选。通过这个方式你可以完全的掌控文件夹的结构。不过，如果你的文件已经在硬盘中组织的很好，你可能只是想在 CMake 中复现这种组织。在 CMake 3.8+ 中，你可以用新版的 `source_group` 命令非常容易的做到上述情形：

```
source_group(TREE "${CMAKE_CURRENT_SOURCE_DIR}/base/dir"
PREFIX "Header Files" FILES ${FILE_LIST})
```

对于 `TREE` 选项，通常应该给出一个以 `${CMAKE_CURRENT_SOURCE_DIR}` 起始的完整路径（因为此命令的文件解析路径是相对于构建目录的）。这个 `PREFIX` 设置文件将在 IDE 结构中的位置，而 `FILES` 选项是包含一些文件的列表 (FILE_LIST)。CMake 将会解析 `TREE` 路径下 `FILE_LIST` 中包含的文件，并将每个文件添加到 `PREFIX` 结构下，这构成了 IDE 的文件夹结构。

> 注意：如果你需要支持低于 3.8 版本的CMake，我不建议你使用上述命令，只建议在 CMake 3.8+ 中使用上述文件夹布局。对于做这种文件夹布局的旧方法，请参见 这篇博文。

在 IDE 中运行CMake

要使用 IDE，如果 CMake 可以生成对应 IDE 的文件（例如 Xcode，Visual Studio），可以通过 `-G"name of IDE"` 来完成，或者如果 IDE 已经内置了对 CMake 的支持（例如 CLion，QtCreator 和一些其他的 IDE），你可以直接在 IDE 中打开 `CMakeLists.txt` 来运行 CMake。

调试

调试代码

你可能需要对你的 CMake 构建过程或你的 C++ 代码进行调试。本文将介绍这两者。

调试 CMake

首先，让我们来盘点一下调试 CMakeLists 和其他 CMake 文件的方法。

打印变量

通常我们使用的打印语句如下:

```
message(STATUS "MY_VARIABLE=${MY_VARIABLE}")
```

然而，通过一个内置的模组 `CMakePrintHelpoers` 可以更方便的打印变量:

```
include(CMakePrintHelpers)
cmake_print_variables(MY_VARIABLE)
```

如果你只是想要打印一个变量，那么上述方法已经很好用了！如果你想要打印一些关于某些目标 (或者是其他拥有变量的项目，比如 `SOURCES`、`DIRECTORIES`、`TESTS` , 或 `CACHE_ENTRIES` - 全局变量好像因为某些原因缺失了) 的变量，与其一个一个打印它们，你可以简单的列举并打印它们:

```
cmake_print_properties(
    TARGETS my_target
    PROPERTIES POSITION_INDEPENDENT_CODE
)
```

跟踪运行

你可能想知道构建项目的时候你的 CMake 文件究竟发生了什么，以及这些都是如何发生的？用 `--trace-source="filename"` 就很不错，它会打印出你指定的文件现在运行到哪一行，让你可以知道当前具体在发生什么。另外还有一些类似的选项，但这些命令通常给出一大堆输出，让你找不着头脑。

例子:

```
cmake -S . -B build --trace-source=CMakeLists.txt
```

如果你添加了 `--trace-expand` 选项，变量会直接展开成它们的值。

以 debug 模式构建

对于单一构建模式的生成器 (single-configuration generators)，你可以使用参数 `-DCMAKE_BUILD_TYPE=Debug` 来构建项目，以获得调试标志 (debugging flags)。对于支持多个构建模式的生成器 (multi-configuration generators)，像是多数IDE，你可以在 IDE 里打开调试模式。这种模式有不同的标志（变量以 `_DEBUG` 结尾，而不是 `_RELEASE` 结尾），以及生成器表达式的值 `CONFIG:Debug` 或 `CONFIG:Release`。

如果你使用了 debug 模式构建，你就可以在上面运行调试器了，比如 gdb 或 lldb。

包含子项目

包含子项目

这就是将一个好的 Git 系统与 CMake 共同使用的优势所在。虽然靠这种方法无法解决世界上所有的问题，但可以解决大部分基于 C++ 的工程包含子项目的问题！

本章中列出了几种包含子项目的方法。

子模组

Git 子模组（Submodule）

如果你想要添加一个 Git 仓库，它与你的项目仓库使用相同的 Git 托管服务（诸如 GitHub、GitLab、BitBucker 等等），下面是正确的添加一个子模组到 `extern` 目录中的命令：

```
gitbook $ git submodule add ../../owner/repo.git extern/repo
```

此处的关键是使用相对于你的项目仓库的相对路径，它可以保证你使用与主仓库相同的访问方式（ ssh 或 https ）访问子模组。这在大多数情况都能工作得相当好。当你在一个子模组里的时候，你可以把它看作一个正常的仓库，而当你在主仓库里时，你可以用 `add` 来改变当前的提交指针。

但缺点是你的用户必须懂 git submodule 命令，这样他们才可以 `init` 和 `update` 仓库，或者他们可以在最开始克隆你的仓库的时候加上 `--recursive` 选项。针对这种情况，CMake 提供了一种解决方案：

```
find_package(Git QUIET)
if(GIT_FOUND AND EXISTS "${PROJECT_SOURCE_DIR}/.git")
# Update submodules as needed
    option(GIT_SUBMODULE "Check submodules during build" ON)
    if(GIT_SUBMODULE)
        message(STATUS "Submodule update")
        execute_process(COMMAND ${GIT_EXECUTABLE} submodule
update --init --recursive
                        WORKING_DIRECTORY
${CMAKE_CURRENT_SOURCE_DIR}
                        RESULT_VARIABLE GIT_SUBMOD_RESULT)
        if(NOT GIT_SUBMOD_RESULT EQUAL "0")
            message(FATAL_ERROR "git submodule update --init
--recursive failed with ${GIT_SUBMOD_RESULT}, please
checkout submodules")
        endif()
    endif()
endif()

if(NOT EXISTS
"${PROJECT_SOURCE_DIR}/extern/repo/CMakeLists.txt")
    message(FATAL_ERROR "The submodules were not downloaded!
GIT_SUBMODULE was turned off or failed. Please update
submodules and try again.")
endif()
```

第一行使用 CMake 自带的 `FindGit.cmake` 检测是否安装了 Git 。然后，如果项目源目录是一个 git 仓库，则添加一个选项（默认值为 `ON`），用户可以自行决定是否打开这个功能。然后我们运行命令来获取所有需要的仓库，如果该命令出错了，则 CMake 配置失败，同时会有一份很好的报错信息。最后无论我们以什么方式获取了子模组，CMake 都会检查仓库是否已经被拉取到本地。你也可以使用 `OR` 来列举其中的几个。

现在，你的用户可以完全忽视子模组的存在了，而你同时可以拥有良好的开发体验！唯一需要开发者注意的一点是，如果你正在子模组里开发，你会在重新运行 CMake 的时候重置你的子模组。只需要添加一个新的提交到主仓库的暂存区，就可以避免这个问题。

然后你就可以添加对 CMake 有良好支持的项目了：

```
add_subdirectory(extern/repo)
```

或者，如果这是一个只有头文件的库，你可以创建一个接口库目标 (interface library target) 。或者，如果支持的话，你可以使用`find_package`，可能初始的搜索目录就是你所添加的目录（查看文档或你所使用的`Find*.cmake`文件）。如果你追加到你的`CMAKE_MODULE_PATH`，你也可以包括一个CMake帮助文件目录，例如添加`pybind11`改进过的`FindPython*.cmake`文件。

小贴士：获取 Git 版本号

将下面的命令加入到上述 Git 更新子仓库的那段中：

```
execute_process(COMMAND ${GIT_EXECUTABLE} rev-parse --short
HEAD
                WORKING_DIRECTORY
"${CMAKE_CURRENT_SOURCE_DIR}"
                OUTPUT_VARIABLE PACKAGE_GIT_VERSION
                ERROR_QUIET
                OUTPUT_STRIP_TRAILING_WHITESPACE)
```

使用 CMake 下载项目

使用 CMake 下载项目

在构建时 (build time) 下载

直到 CMake 3.11， 主流的下载包的方法都在构建时进行。这（在构建时下载）会造成几个问题；其中最主要问题的是 `add_subdirectory` 不能对一个尚不存在的文件夹使用！因此，我们导入的外部项目内置的工具必须自己构建自己（这个外部项目）来解决这个问题。（同时，这种方法也能用于构建不支持 CMake 的包）[1]

[1]. 注意，外部数据就是不在包内的数据的工具。 ↩

在配置时 (configure time) 下载

如果你更喜欢在配置时下载，看看这个仓库 Crascit/DownloadProject ，它提供了插件式（不需要改变你原有的 CMakeLists.txt）的解决方案。但是，子模块 (submodules) 很好用，以至于我已经停止了使用 CMake 对诸如 GoogleTest 之类的项目的下载，并把他们加入到了子模块中。自动下载在没有网络访问的环境下也是难以实现的，并且外部项目经常被下载到构建目录中，如果你有多个构建目录，这就既浪费时间又浪费空间。

获取软件包（FetchContent）(CMake 3.11+)

获取软件包（FetchContent）(CMake 3.11+)

有时你想要在配置的时候下载数据或者是包，而不是在编译的时候下载。这种方法已经被第三方包重复“发明”了好几次。最终，这种方法在 CMake 3.11 中以 FetchContent 模块的形式出现。

FetchContent 模块有出色的文档，我在此不会赘述。我会阐述这样几个步骤：

- 使用 `FetchContent_Declare(MyName)` 来从 URL、Git 仓库等地方获取数据或者是软件包。
- 使用 `FetchContent_GetProperties(MyName)` 来获取 `MyName_*` 等变量的值，这里的 `MyName` 是上一步获取的软件包的名字。
- 检查 `MyName_POPULATED` 是否已经导出，否则使用 `FetchContent_Populate(MyName)` 来导出变量（如果这是一个软件包，则使用 `add_subdirectory("${MyName_SOURCE_DIR}" "${MyName_BINARY_DIR}")`）

比如，下载 Catch2：

```
FetchContent_Declare(
  catch
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        v2.13.6
)

# CMake 3.14+
FetchContent_MakeAvailable(catch)
```

如果你不能使用 CMake 3.14+，可以使用适用于低版本的方式来加载：

```
# CMake 3.11+
FetchContent_GetProperties(catch)
if(NOT catch_POPULATED)
  FetchContent_Populate(catch)
  add_subdirectory(${catch_SOURCE_DIR} ${catch_BINARY_DIR})
endif()
```

当然，你可以将这些语句封装到一个宏内：

```
if(${CMAKE_VERSION} VERSION_LESS 3.14)
    macro(FetchContent_MakeAvailable NAME)
        FetchContent_GetProperties(${NAME})
        if(NOT ${NAME}_POPULATED)
            FetchContent_Populate(${NAME})
            add_subdirectory(${${NAME}_SOURCE_DIR}
${${NAME}_BINARY_DIR})
        endif()
    endmacro()
endif()
```

这样，你就可以在 CMake 3.11+ 里使用 CMake 3.14+ 的语法了。

可以在这里查看例子。

测试

测试

General Testing Information

你需要在你的主 CMakeLists.txt 文件中添加如下函数调用（而不是在子文件夹 CMakeLists.txt 中）：

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    include(CTest)
endif()
```

这么做将可以使得具有 CMake 测试功能，并且具有一个 `BUILD_TESTING` 选项使得用户可以选择开启或关闭测试（还有一些其他的设置）。或者你可以直接通过调用 `enable_testing()` 函数来开启测试。

当你添加你自己的测试文件夹时，你应该这么做：

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME AND
BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

这么做的（译者注：需要添加 `CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME`）的原因是，如果有他人包含了你的包，并且他们开启了 `BUILD_TESTING` 选项，但他们并不想构建你包内的测试单元，这样会很有用。在极少数的情况下他们可能真的想要开启所有包的测试功能，你可以提供给他们一个可以覆盖的变量（如下例的 `MYPROJECT_BUILD_TESTING`，当设置 `MYPROJECT_BUILD_TESTING` 为 ON 时，会开启该项目的测试功能）：

```
if((CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME OR
MYPROJECT_BUILD_TESTING) AND BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

本书中的示例就使用了覆盖变量的形式来开启所有测试，因为主 CMake 项目确实想要运行所有子项目的测试功能。

你可以这样注册一个测试目标(targets)：

```
add_test(NAME TestName COMMAND TargetName)
```

如果你在 `COMMAND` 后写了除 `TargetName` 之外的东西，他将会被注册为在命令行运行的指令。在这里写生成器表达式(generator-expression)也是有效的：

```
add_test(NAME TestName COMMAND $<TARGET_FILE:${TESTNAME}>)
```

这么写将会使用该目标生成的文件（也就是生成的可执行文件）的路径作为参数。

将构建作为测试的一部分

如果你想在测试时运行 CMake 构建一个项目，这也是可以的（事实上，这也是 CMake 如何进行自我测试的）。例如，如果你的主项目名为 `MyProject` 并且你有一个 `examples/simple` 项目需要在测试时构建，那么可以这么写：

```
add_test(
  NAME
    ExampleCMakeBuild
```

```
  COMMAND
    "${CMAKE_CTEST_COMMAND}"
            --build-and-test
"${My_SOURCE_DIR}/examples/simple"

"${CMAKE_CURRENT_BINARY_DIR}/simple"
            --build-generator "${CMAKE_GENERATOR}"
            --test-command "${CMAKE_CTEST_COMMAND}"
)
```

测试框架

可以查看子章节了解主流测试框架的使用方式(recipes)：

- GoogleTest: 一个 Google 出品的主流测试框架。不过开发可能有点慢。
- Catch2: 一个现代的，具有灵巧的宏的 PyTest-like 的测试框架。
- DocTest: 一个 Catch2 框架的替代品，并且编译速度更快、更干净(cleaner)。See Catch2 chapter and replace with DocTest.

GoogleTest

GoogleTest

GoogleTest 和 GoogleMock 是非常经典的选择；不过就我个人经验而言，我会推荐你使用 Catch2，因为 GoogleTest 十分遵循谷歌的发展理念；它假定用户总是想使用最新的技术，因此会很快的抛弃旧的编译器（不对其适配）等等。添加 GoogleMock 也常常令人头疼，并且你需要使用 GoogleMock 来获得匹配器(matchers)，这在 Catch2 是一个默认特性，而不需要手动添加（但 docstest 没有这个特性）。

子模块(Submodule)的方式（首选）

当使用这种方式，只需要将 GoogleTest 设定(checkout) 为一个子模块：[1]

```
git submodule add --branch=release-1.8.0
../../google/googletest.git extern/googletest
```

然后，在你的主 `CMakeLists.txt` 中：

```
option(PACKAGE_TESTS "Build the tests" ON)
if(PACKAGE_TESTS)
    enable_testing()
    include(GoogleTest)
    add_subdirectory(tests)
endif()
```

我推荐你使用一些像 `PROJECT_NAME STREQUAL CMAKE_PROJECT_NAME` 来设置 `PACKAGE_TEST` 选项的默认值，因为这样只会在项目为主项目时才构建测试单元。

像之前提到的，你必须在你的主 `CMakeLists.txt` 文件中调用 `enable_testing()` 函数。 现在，在你的 `tests` 目录中：

```
add_subdirectory("${PROJECT_SOURCE_DIR}/extern/googletest"
"extern/googletest")
```

如果你在你的主 `CMakeLists.txt` 中调用它，你可以使用普通的 `add_subdirectory`；这里因为我们是从子目录中调用的，所以我们需要一个额外的路径选项来更正构建路径。

下面的代码是可选的，它可以让你的 `CACHE` 更干净：

```
mark_as_advanced(
    BUILD_GMOCK BUILD_GTEST BUILD_SHARED_LIBS
    gmock_build_tests gtest_build_samples gtest_build_tests
    gtest_disable_pthreads gtest_force_shared_crt
gtest_hide_internal_symbols
)
```

If you are interested in keeping IDEs that support folders clean, I would also add these lines:

```
set_target_properties(gtest PROPERTIES FOLDER extern)
set_target_properties(gtest_main PROPERTIES FOLDER extern)
set_target_properties(gmock PROPERTIES FOLDER extern)
set_target_properties(gmock_main PROPERTIES FOLDER extern)
```

然后，为了增加一个测试，推荐使用下面的宏：

```
macro(package_add_test TESTNAME)
    # create an exectuable in which the tests will be stored
    add_executable(${TESTNAME} ${ARGN})
    # link the Google test infrastructure, mocking library,
and a default main fuction to
    # the test executable.  Remove g_test_main if writing
your own main function.
    target_link_libraries(${TESTNAME} gtest gmock
gtest_main)
    # gtest_discover_tests replaces gtest_add_tests,
    # see
https://cmake.org/cmake/help/v3.10/module/GoogleTest.html
for more options to pass to it
    gtest_discover_tests(${TESTNAME}
        # set a working directory so your project root so
that you can find test data via paths relative to the
project root
        WORKING_DIRECTORY ${PROJECT_DIR}
        PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY
"${PROJECT_DIR}"
    )
    set_target_properties(${TESTNAME} PROPERTIES FOLDER
tests)
endmacro()

package_add_test(test1 test1.cpp)
```

这可以简单、快速的添加测试单元。你可以随意更改来满足你的需求。如果你之前没有了解过 ARGN，ARGN 是显式声明的参数外的所有参数。如 `package_add_test(test1 test1.cpp a b c)`，ARGN 包含除 test1 与 test1.cpp 外的所有参数。

可以更改宏来满足你的要求。例如，如果你需要链接不同的库来进行不同的测试，你可以这么写：

```
macro(package_add_test_with_libraries TESTNAME FILES
LIBRARIES TEST_WORKING_DIRECTORY)
    add_executable(${TESTNAME} ${FILES})
    target_link_libraries(${TESTNAME} gtest gmock gtest_main
${LIBRARIES})
    gtest_discover_tests(${TESTNAME}
        WORKING_DIRECTORY ${TEST_WORKING_DIRECTORY}
        PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY
"${TEST_WORKING_DIRECTORY}"
    )
    set_target_properties(${TESTNAME} PROPERTIES FOLDER
tests)
endmacro()

package_add_test_with_libraries(test1 test1.cpp lib_to_test
"${PROJECT_DIR}/european-test-data/")
```

下载的方式

你可以通过 CMake 的 `include` 指令使用使用我在 CMake helper repository 中的下载器，

这是一个 GoogleTest 的下载器，基于优秀的 DownloadProject 工具。为每个项目下载一个副本是使用 GoogleTest 的推荐方式（so much so, in fact, that they have disabled the automatic CMake install target），so this respects that design decision. 这个方式在项目配置时下载 GoogleTest，所以 IDEs 可以正确的找到这些库。这样使用起来很简单：

```
cmake_minimum_required(VERSION 3.10)
project(MyProject CXX)
list(APPEND CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)

enable_testing() # Must be in main file

include(AddGoogleTest) # Could be in /tests/CMakeLists.txt
add_executable(SimpleTest SimpleTest.cu)
add_gtest(SimpleTest)
```

> 提示：add_gtest 只是一个添加 gtest，gmock 以及 gtest_main 的宏，然后运行 add_test 来创建一个具有相同名字的测试单元
>
> ```
> target_link_libraries(SimpleTest gtest gmock gtest_main)
> add_test(SimpleTest SimpleTest)
> ```

FetchContent: CMake 3.11

这个例子是用 FetchContent 来添加 GoogleTest：

```
include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        release-1.8.0
)

FetchContent_GetProperties(googletest)
if(NOT googletest_POPULATED)
  FetchContent_Populate(googletest)
  add_subdirectory(${googletest_SOURCE_DIR}
${googletest_BINARY_DIR})
endif()
```

[1]. 在这里我假设你在 Github 仓库中使用 googletest，然后使用的是 googletest 的相对路径。↩

Catch

Catch

Catch2（只有 C++11 版本）是一个独立且强大的测试工具，它的理念（philosophy）类似于 Python 中的 Pytest。他比 GTest 支持更多的编译器版本，并且会紧跟潮流支持新的事物，比如支持在 M1 版本 MacOS 上使用 Catch。他也有一个相似但是更加快速的双胞胎兄弟，doctest，他编译十分迅速但是缺少了一些类似于匹配器（features）的特性。为了在 CMake 项目中使用 Catch，下面是一些可选的方式：

如何配置

Catch 对 CMake 支持很友好，不过你还是需要下载整个仓库来使用他。无论是使用 submodules 还是FetchContent 都可以。`extended-project` 与 `fetch` 这两个示例用的都是 FetchContent 的方式。更多的可以参考官方文档。

Quick download

这可能是最简单并且对老版本 CMake 适配性更好的方式。你可以一步到位地直接下载一个 All-in-one 的头文件：

```
add_library(catch_main main.cpp)
target_include_directories(catch_main PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
set(url
https://github.com/philsquared/Catch/releases/download/v2.13
.6/catch.hpp)
file(
  DOWNLOAD ${url} "${CMAKE_CURRENT_BINARY_DIR}/catch.hpp"
  STATUS status
  EXPECTED_HASH
SHA256=681e7505a50887c9085539e5135794fc8f66d8e5de28eadf13a30
978627b0f47)
list(GET status 0 error)
if(error)
  message(FATAL_ERROR "Could not download ${url}")
endif()
target_include_directories(catch_main PUBLIC
"${CMAKE_CURRENT_BINARY_DIR}")
```

在 Catch 3 发布后，你可能需要下载两个文件，因为现在需要两个文件进行测试（但是你不再需要自己写 main.cpp 文件）。这个 `main.cpp` 文件看起来像这样：

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

Vendoring

如果你已经把 Catch 加入到你项目的一部分（放到了一个单独的文件夹中），你可以这样来使用 Catch：

```
# Prepare "Catch" library for other executables
set(CATCH_INCLUDE_DIR
${CMAKE_CURRENT_SOURCE_DIR}/extern/catch)
add_library(Catch2::Catch IMPORTED INTERFACE)
set_property(Catch2::Catch PROPERTY
INTERFACE_INCLUDE_DIRECTORIES "${CATCH_INCLUDE_DIR}")
```

然后，你需要链接到 Catch2::Catch。你也可以把它作为一个 INTERFACE 目标，因为你不会导出你的测试模块。

Direct inclusion

如果你使用 ExternalProject，FetchContent 或者 git submodules 的形式来添加库，你也可以使用 `add_subdirectory` 。（CMake 3.1+）

Catch 还提供了两个 CMake 模块（modules），你可以通过这个来注册独立的测试。

导出与安装

导出与安装

让别人使用库有三种好方法和一种坏方法:

查找模块（不好的方式）

`Find<mypackage>.cmake` 脚本是为那些不支持 CMake 的库所设计，所以已经使用 CMake 的库，不要创建这个脚本文件！可以使用 `Config<mypackage>.cmake`，具体方式如下所示。

添加子项目

可以将项目作为一个子目录放置于包中，接着使用 `add_subdirectory` 添加相应的子目录，这适用于纯头文件和快速编译的库。还需要注意的是，安装命令可能会干扰父项目，因此可以使用 `add_subdirectory` 的`EXCLUDE_FROM_ALL`选项；当显式使用的目标时，仍然会进行构建。

作为库的作者，请使用 `CMAKE_CURRENT_SOURCE_DIR` 而非 `PROJECT_SOURCE_DIR` (对于其他变量也是如此，比如 `CMAKE_CURRRENT_BINARY_DIR`)。通过检查 `CMAKE_PROJECT_NAME` 和 `PROJECT_NAME` 的内容是否相同 （STREQUAL），可以只添加对项目有意义的选项或默认值。

此外，使用命名空间也是不错的方式。使用库的方式应该与下面的一致，应该对所有方法的使用进行标准化。

```
add_library(MyLib::MyLib ALIAS MyLib)
```

这里的 ALIAS（别名）目标不会在后面导出。

导出

第三种方法是 `*Config.cmake` 脚本，这将是下一章的主题。

安装

安装

进行安装时，比如执行 make install，安装命令会将文件或目标“安装”到安装树中。简单使用目标安装指令的方式:

```
install(TARGETS MyLib
        EXPORT MyLibTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        RUNTIME DESTINATION bin
        INCLUDES DESTINATION include
        )
```

当有一个库、静态库或程序要安装时，才需要将不同的文件安装到不同的目的地。由于目标不安装包含目录，所以 包含（INCLUDES）目标是特殊的。只能在导出的目标上设置包含目录（通常由target_include_directories设置，若想要清理 cmake 文件，需要检查 MyLibTargets 文件，确定没有多次包含同一个包含目录）。

给定 CMake 可访问的版本是个不错的方式。使用 find_package 时，可以这样指定版本信息:

```
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
    MyLibConfigVersion.cmake
    VERSION ${PACKAGE_VERSION}
    COMPATIBILITY AnyNewerVersion
    )
```

接下来有两个选择。创建MyLibConfig.cmake，可以直接将目标导出放在这个文件中，或者手动写入，然后包目标文件。若有依赖项（可能只是 OpenMP），则需要添加相应的选项。下面是个例子:

首先，创建一个安装目标文件（类似于在构建目录中创建的文件）:

```
install(EXPORT MyLibTargets
        FILE MyLibTargets.cmake
        NAMESPACE MyLib::
        DESTINATION lib/cmake/MyLib
         )
```

该文件将获取导出目标，并将其放入文件中。若没有依赖项，只需使用 MyLibConfig.cmake 代替 MyLibTargets.cmake 即可。然后，在源码树的某处，创建一个自定义 MyLibConfig.cmake 文件。若想要捕获配置时的变量，可以使用 .in 文件，并且可以使用 @var@ 语法。具体方式如下所示:

```
include(CMakeFindDependencyMacro)

# Capturing values from configure (optional)
set(my-config-var @my-config-var@)

# Same syntax as find_package
find_dependency(MYDEP REQUIRED)

# Any extra setup

# Add the targets file
include("${CMAKE_CURRENT_LIST_DIR}/MyLibTargets.cmake")
```

现在，可以使用配置文件（若使用 .in 文件），然后安装已生成的文件。因为创建了ConfigVersion文件，所以可以在这里安装它。

```
configure_file(MyLibConfig.cmake.in MyLibConfig.cmake @ONLY)
install(FILES
"${CMAKE_CURRENT_BINARY_DIR}/MyLibConfig.cmake"

"${CMAKE_CURRENT_BINARY_DIR}/MyLibConfigVersion.cmake"
        DESTINATION lib/cmake/MyLib
        )
```

就是这样！现在，当包安装完成后，lib/cmake/MyLib 中就出现了 CMake 搜索所需的文件（特别是MyLibConfig.cmake和MyLibConfigVersion.cmake），配置时使用的目标文件应该也在那里。

当 CMake 搜索包时，将在当前安装目录，以及几个标准位置中进行查找。可以手动将相应的目录添加到搜索路径中，包括 MyLib_PATH。若没有找到配置文件，CMake 会输出相应的信息，告知用户当前的情况。

导出

导出

CMake 3.15 中，导出的默认行为发生了变化。由于更改用户主目录中的文件是“令人惊讶的”（确实如此，这就是本章存在的原因），因此不再是默认行为。若将 CMake 的最小或最大版本设置为 3.15+，这种情况将不再发生，除非将 `CMAKE_EXPORT_PACKAGE_REGISTRY` 设置为`ON`。

CMake 访问项目有三种方式：子目录、导出构建目录和安装。要使用项目的构建目录，就需要导出目标。正确的安装需要导出目标，使用构建目录只需要再增加了两行代码，但这并不是我推荐的工作方式。不过，对于开发和安装过程来说的确好用。

还需要创建导出集，可能要放在主 `CMakeLists.txt` 文件的末尾:

```
export(TARGETS MyLib1 MyLib2 NAMESPACE MyLib:: FILE
MyLibTargets.cmake)
```

这将把列出的目标放到构建目录的文件中，还可以给添加一个命名空间作为前缀。现在，CMake 可以找到这个包了，并将这个包导出到 `$HOME/.cmake/packages` 文件夹下:

```
set(CMAKE_EXPORT_PACKAGE_REGISTRY ON)
export(PACKAGE MyLib)
```

现在，`find_package(MyLib)`就可以找到构建文件夹了。来看看生成的 `MyLibTargets.cmake`文件到底做了什么。它只是一个普通的CMake文件，但带有导出的目标。

注意，这种方式有一个缺点：若导入了依赖项，则需要在 `find_package` 之前导入它们。这个问题将在后面的章节中解决。

打包

打包

CMake 有两种打包方式：一是使用CPackConfig.cmake文件；二是将 CPack 变量放置在 CMakeLists.txt 文件中。若想要包含主构建的相关变量（比如：版本号），可以使用配置文件的方式。这里，我将展示第二种方式：

```
# Packaging support
set(CPACK_PACKAGE_VENDOR "Vendor name")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "Some summary")
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
set(CPACK_RESOURCE_FILE_LICENSE
"${CMAKE_CURRENT_SOURCE_DIR}/LICENCE")
set(CPACK_RESOURCE_FILE_README
"${CMAKE_CURRENT_SOURCE_DIR}/README.md")
```

这些是生成二进制包时最常见的变量。二进制包使用 CMake 的安装机制，已经安装的东西都会显示出来。

当然，还可以制作源码包。可以将相应的正则表达式添加到 CMAKE_SOURCE_IGNORE_FILES 中，以确保只打包期望的文件（排除构建目录或 git 信息）；否则，package_source 会将源目录中的所有内容打包在一起。这里，也可以根据自己的喜欢的文件类型，对源码包生成器进行设置：

```
set(CPACK_SOURCE_GENERATOR "TGZ;ZIP")
set(CPACK_SOURCE_IGNORE_FILES
    /.git
    /dist
    /.*build.*
    /\\\\.DS_Store
)
```

注意，这种方式无法在 Windows 系统中正常运行，但是生成的源码包可以在 Windows 系统中正常使用。

最后，需要包含一下 CPack 模块:

```
include(CPack)
```

查找库（或包）

查找包

CMake 中有两种方式查找包：“模块”模式（Module）和“配置”模式（Config）。

CUDA

CUDA

使用 CUDA 有两种方式：CMake 3.8（Windows为3.9）中引入的新方法，应该比旧的方法更受欢迎 —— 可能会在旧包中使用这种方法，所以本节会提一下。与旧语言不同，CUDA 的支持一直在快速发展，因为构建 CUDA 非常困难，所以建议*使用最新版本*的 CMake！CMake 3.17 和 3.18 有很多直接针对 CUDA 的改进。

对于 CUDA 和现代 CMake 来说，一个很好的参考是 CMake 开发者 Robert Maynard 在 GTC 2017 的 演讲 ppt。

启用 CUDA 语言

有两种方法可以启用 CUDA（若 CUDA 的支持不可选）：

```
project(MY_PROJECT LANGUAGES CUDA CXX)
```

这里可能需要在将 `CXX` 一并列出。

若 CUDA 的支持可选，可以将其放在条件语句中：

```
enable_language(CUDA)
```

要检查 CUDA 是否可用，可使用 CheckLanuage：

```
include(CheckLanguage)
check_language(CUDA)
```

可以通过检查 `CMAKE_CUDA_COMPILER`（CMake 3.11 之前没有）来查看 CUDA 开发包是否存在。

可以检查 `CMAKE_CUDA_COMPILER_ID`（对于 nvcc，其值为 `"NVIDIA"`，Clang 将在 CMake 3.18 支持）。可以用 `CMAKE_CUDA_COMPILER_VERSION` 检查 CUDA 版本。

CUDA 的变量

CMake 中许多名称中带有 `CXX` 的变量都有 `CUDA` 版本。例如，要设置 CUDA 所需的 C++ 标准，

```
if(NOT DEFINED CMAKE_CUDA_STANDARD)
    set(CMAKE_CUDA_STANDARD 11)
    set(CMAKE_CUDA_STANDARD_REQUIRED ON)
endif()
```

若正在查找和设置 CUDA 的标准级别，CMake 3.17 中添加了一组新的编译器特性，比如：`cuda_std_11`。这些版本特性与 `cxx` 的版本特性使用方式相同。

添加库/可执行文件

这部分很简单；使用 CUDA 文件 `.cu`，*就像平常添加库一样*。

也可以使用分离编译的方式：

```
set_target_properties(mylib PROPERTIES
                            CUDA_SEPARABLE_COMPILATION ON)
```

也可以直接使用 CUDA_PTX_COMPILATION 属性创建一个 PTX（Parallel Thread eXecution）文件。

目标架构

构建 CUDA 代码时，应该以架构为目标。若没有明确的架构信息，也可以编译成 ptx，nvcc 编译器会提供基本的指令，所以 PTX 还需要在运行时进行编译，这会使 GPU kernel 的加载速度慢得多。

所有 NVIDA 显卡都有一个架构级别，比如：7.2。在处理架构时，有两个选择：

- 代码层：将向正在编译的代码预报一个版本（如：5.0），这将使用 5.0 之前的所有特性，但不会超过 5.0（假设代码/标准库编写良好）。
- 目标架构：必须等于或大于架构版本。这需要有与目标显卡相同的主版本号，并且等于或小于目标显卡。所以使用架构为 7.2 的显卡时，在编译时将代码架构版本设置为 7.0 将是首选。最后，还可以生成 PTX；PTX 将在架构版本大于当前架构的所有显卡上工作，不过需要在运行时再对 PTX 进行编译。

CMake 3.18 中，设置目标架构变得非常容易。若 CMake 的版本范围为 3.18+，可以对目标使用 CMAKE_CUDA_ARCHITECTURES 变量和 CUDA_ARCHITECTURES 属性。允许直接写值（不带 .），比如：架构 5.0，可以就写为 50。若设置为 OFF，将不会传递任何架构信息。

使用目标

使用目标与 CXX 类似，但有一个问题。若目标包含编译器选项（或标志），大多数情况下，这些选项将无法正确使用（很难正确的封装在 CUDA 包装宏或函数中）。正确的编译器选项设置应该如下所示：

```
"$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>>:-
fopenmp>$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>>:-
Xcompiler=-fopenmp>"
```

然而，不管是使用传统 CMake 的 find_package 方法，还是使用现代 CMake 的目标和继承方法，都不好使。这是我吃了不少苦头总结出来的经验。

目前，有一个合适的解决方案，*只要知道未别名的目标名称*即可。这是一个函数，若使用 CUDA 编译器，可以通过包装编译选项（标志）来修复仅处理 C++ 的目标：

```
function(CUDA_CONVERT_FLAGS EXISTING_TARGET)
    get_property(old_flags TARGET ${EXISTING_TARGET}
PROPERTY INTERFACE_COMPILE_OPTIONS)
    if(NOT "${old_flags}" STREQUAL "")
        string(REPLACE ";" "," CUDA_flags "${old_flags}")
        set_property(TARGET ${EXISTING_TARGET} PROPERTY
INTERFACE_COMPILE_OPTIONS

"$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>>:${old_flags}>$
<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>>:-
Xcompiler=${CUDA_flags}>"
            )
    endif()
endfunction()
```

内置变量

- CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES：指示 CUDA 开发包内置 Thrust 等工具的目录
- CMAKE_CUDA_COMPILER：NVCC 的具体路径

即使不启用 CUDA 语言，也可以使用 `FindCUDAToolkit` 来查找 CUDA 的各种目标和变量。

注意，FindCUDA 已弃用，但对于低于 3.18 的 CMake，以下函数需要 FindCUDA：

- CUDA 版本检查/选择版本
- 架构检测（注意：3.12 部分修复了这个问题）
- 为 CUDA 库连接非 `.cu` 文件

FindCUDA [警告：不要使用] (仅供参考)

若要支持旧版 CMake，建议至少在 CMake 文件夹中包含来自 CMake 3.9 版本的 FindCUDA（参见 CLIUtils github 组织中的 git库）。需要添加两个特性：`CUDA_LINK_LIBRARIES_KEYWORD` 和`cuda_select_nvcc_arch_flags`，以及较新的架构和 CUDA 版本。

要使用旧版 CUDA 支持方式，可以使用 `find_package`：

```
find_package(CUDA 7.0 REQUIRED)
message(STATUS "Found CUDA ${CUDA_VERSION_STRING} at
${CUDA_TOOLKIT_ROOT_DIR}")
```

可以用 `CUDA_NVCC_FLAGS`（使用列表添加的方式，`list(APPEND)`）控制 CUDA 标志，通过 `CUDA_SEPARABLE_COMPILATION` 控制分离编译。若想确保 CUDA 的正常工作，需要将关键字添加到目标中（CMake 3.9+）：

```
set(CUDA_LINK_LIBRARIES_KEYWORD PUBLIC)
```

若想让用户检查当前硬件的架构标志，可以使用以下方式：

```
cuda_select_nvcc_arch_flags(ARCH_FLAGS) # optional argument
for arch to add
```

OpenMP

OpenMP

CMake 3.9+ 中对 OpenMP 的支持进行了极大的改善。现代（TM）CMake 使用 OpenMP 链接到一个目标的方法如下：

```
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    target_link_libraries(MyTarget PUBLIC
OpenMP::OpenMP_CXX)
endif()
```

这不仅比传统方法简单，若需要的话，还可以将库链接与编译的设置分开。CMake 3.12+ 中，甚至支持了 macOS 系统中的 OpenMP（需要对库文件进行安装，例如 `brew install libomp`）。若需要支持旧版 CMake，下面的代码可以在 CMake 3.1+ 上正常运行：

```
# For CMake < 3.9, we need to make the target ourselves
if(NOT TARGET OpenMP::OpenMP_CXX)
    find_package(Threads REQUIRED)
    add_library(OpenMP::OpenMP_CXX IMPORTED INTERFACE)
    set_property(TARGET OpenMP::OpenMP_CXX
                 PROPERTY INTERFACE_COMPILE_OPTIONS
${OpenMP_CXX_FLAGS})
    # Only works if the same flag is passed to the linker;
use CMake 3.9+ otherwise (Intel, AppleClang)
    set_property(TARGET OpenMP::OpenMP_CXX
                 PROPERTY INTERFACE_LINK_LIBRARIES
${OpenMP_CXX_FLAGS} Threads::Threads)

endif()
target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
```

警告：CMake 小于 3.4 的版本中，Threads 包中有一个 bug，需要启用C语言。

Boost

Boost 库

CMake 提供 Boost 库的查找包，但其工作方式有些奇怪。FindBoost 中可获得完整的描述：这只是一个概述，并提供一个示例。务必查看页面上使用 CMake 的最低版本，然后再查看有哪些支持的选项。

首先，可以在搜索 Boost 之前设置的一组变量，自定义选定 Boost 库的行为。可设置的变量随着 CMake 的发展变得越来越多，这里仅设置三个常见的变量：

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
```

CMake 3.5 添加了导入目标。这些目标有助于处理依赖关系，所以这也是添加 Boost 库的好方法。然而，CMake 包含所有已知的 Boost 依赖信息，因此 CMake 的版本必须比 Boost 新，这样 CMake 才能正常工作。最近 CMake 的 合并请求 中，开始假设依赖项会从已知的上一个版本继承下来，并直接使用（同时给出警告）。这项功能也反向移植到了 CMake 3.9 中。

导入目标位于 `Boost::` 命名空间。只有 `Boost::boost` 组件有头文件，其他组件均是预编译的库文件。这里，可以根据需要包含相应的依赖项。

下面的例子中使用了 `Boost::filesystem` 库：

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
find_package(Boost 1.50 REQUIRED COMPONENTS filesystem)
message(STATUS "Boost version: ${Boost_VERSION}")

# This is needed if your Boost version is newer than your
CMake version
# or if you have an old version of CMake (<3.5)
if(NOT TARGET Boost::filesystem)
    add_library(Boost::filesystem IMPORTED INTERFACE)
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_INCLUDE_DIRECTORIES ${Boost_INCLUDE_DIR})
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_LINK_LIBRARIES ${Boost_LIBRARIES})
endif()

target_link_libraries(MyExeOrLibrary PUBLIC
Boost::filesystem)
```

MPI

MPI

添加 MPI 的方式类似于 OpenMP，所以最好使用 CMake 3.9+ 。

```
find_package(MPI REQUIRED)
message(STATUS "Run: ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG}
${MPIEXEC_MAX_NUMPROCS} ${MPIEXEC_PREFLAGS} EXECUTABLE
${MPIEXEC_POSTFLAGS} ARGS")
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

若需要支持旧版 CMake，可以使用 3.1+ 版本来模仿上面的方式：

```
find_package(MPI REQUIRED)

# For supporting CMake < 3.9:

if(NOT TARGET MPI::MPI_CXX)
    add_library(MPI::MPI_CXX IMPORTED INTERFACE)

    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_COMPILE_OPTIONS
${MPI_CXX_COMPILE_FLAGS})
    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_INCLUDE_DIRECTORIES
"${MPI_CXX_INCLUDE_PATH}")
    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_LINK_LIBRARIES
${MPI_CXX_LINK_FLAGS} ${MPI_CXX_LIBRARIES})
endif()

message(STATUS "Run: ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG}
${MPIEXEC_MAX_NUMPROCS} ${MPIEXEC_PREFLAGS} EXECUTABLE
${MPIEXEC_POSTFLAGS} ARGS")
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

ROOT

ROOT

ROOT 是用于高能物理学的 C++ 工具包，里面东西超级多。CMake 确实有很多使用它的方法，不过其中许多/大多数的例子可能是错误的。这里，说一下我推荐的方式。

最新版本的 ROOT，对 CMake 的支持*改进了很多* —— 使用 6.16+ 就变得很容易！若需要支持 6.14 或更早版本，可以继续往下阅读。6.20 版中有进一步的改进，更像一个合适的 CMake 项目，具有为目标导出 C++ 标准特性等功能。

查找 ROOT

ROOT 6.10+ 支持配置文件查找的方式，所以可以这样做：

```
find_package(ROOT 6.16 CONFIG REQUIRED)
```

尝试找到 ROOT 包。若没有设置 ROOT 的路径，可以将 `-DROOT_DIR=$ROOTSYS/cmake` 传递给 CMake，从而找到ROOT。(实际上，应该使用 `source thisroot.sh` 的方式）

正确的方式（使用目标）

ROOT 6.12 及更早版本，没有为导入的目标添加包含目录。ROOT 6.14+ 已经修正了这个错误，所需的目标属性已经变得更好用了。这种方法使得 ROOT 变得越来越容易使用（有关旧版 ROOT 的内容，请参阅本节页末尾的示例）。

链接时，只需选择想要使用的库：

```
add_executable(RootSimpleExample SimpleExample.cxx)
target_link_libraries(RootSimpleExample PUBLIC
ROOT::Physics)
```

若想查看默认列表，可以在命令行上运行 `root-config --libs`。若已使用 Homebrew 安装了 ROOT 6.18，那么就会显示如下信息：

- `ROOT::Core`
- `ROOT::Gpad`
- `ROOT::Graf3d`
- `ROOT::Graf`
- `ROOT::Hist`
- `ROOT::Imt`
- `ROOT::MathCore`
- `ROOT::Matrix`
- `ROOT::MultiProc`
- `ROOT::Net`
- `ROOT::Physics`
- `ROOT::Postscript`
- `ROOT::RIO`
- `ROOT::ROOTDataFrame`
- `ROOT::ROOTVecOps`
- `ROOT::Rint`
- `ROOT::Thread`
- `ROOT::TreePlayer`
- `ROOT::Tree`

旧的全局方式

ROOT 提供了一个配置 ROOT 项目的工具，可以使用 `include("${ROOT_USE_FILE}")` 激活它。这将自动创建不美观的目录群和全局变量。这个工具的确可以节省一些配置时间，但若想通过修改这一过程来解决一些棘手的问题，则需要花费大量的时间。若不需要创建库，那么直接使用脚本也无所谓。这里，包含目录和编译连接标志使用的是全局设置，但仍然需要为目标手动链接 `${ROOT_LIBRARIES}`，可能还需要 `ROOT_EXE_LINKER_FLAGS`（例如使用 macOS 操作系统时，必须在链接之前使用 `separate_arguments`，若有多个标志，就会报错）。另外，在 6.16 之前，必须手动修复一个空格错误。

就像下面这样：

```
# Sets up global settings
include("${ROOT_USE_FILE}")

# This is required for ROOT < 6.16
# string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS
"${ROOT_EXE_LINKER_FLAGS}")

# This is required on if there is more than one flag (like
on macOS)
separate_arguments(ROOT_EXE_LINKER_FLAGS)

add_executable(RootUseFileExample SimpleExample.cxx)
target_link_libraries(RootUseFileExample PUBLIC
${ROOT_LIBRARIES}

${ROOT_EXE_LINKER_FLAGS})
```

组件

CMake 查找 ROOT 时，允许指定相应的组件。CMake 会将列出的内容添加到 `${ROOT_LIBRARIES}`，因此若想使用相应的组件来构建自己的目标，使用这种方式可以避免多次引用。不过，这并没有解决依赖关系的问题；只使用 `RooFit` 而没有 `RooFitCore` 也是错误的。若链接的是 `ROOT::RooFit`，而不是 `${ROOT_LIBRARIES}`，则无需链接`RooFitCore`。

生成字典

生成字典是 ROOT 解决 C++ 中缺少反射特性的一种方式。需要 ROOT 了解自定义类的细节，这样 ROOT 就可以保存它，并且可以在 Cling 解释器中显示相应的方法，等等。源代码需要进行如下修改来支持生成字典：

- 类定义应该以 `ClassDef(MyClassName, 1)` 结束
- 类实现中应该有 `ClassImp(MyClassName)`

ROOT 提供了 `rootcling` 和 `genreflex`（`rootcling` 的遗留接口），这两个二进制文件可生成构建字典所需的源文件，还定义了一个CMake函数 `root_generate_dictionary`，其在构建过程中会使用 `rootling` 。

要加载函数，首先要包含 ROOT 的宏：

```
include("${ROOT_DIR}/modules/RootNewMacros.cmake")
# For ROOT versions than 6.16, things break
# if nothing is in the global include list!
if (${ROOT_VERSION} VERSION_LESS "6.16")
    include_directories(ROOT_NONEXISTENT_DIRECTORY_HACK)
endif()
```

为了修复 `RootNewMacros.cmake` 文件中的错误，这里需要 `if(…)`。若没有全局包含目录或 `inc` 文件夹，该错误会导致字典生成失败。这里我包含了一个不存在的

目录，只是为了让字典能够顺利生成， ROOT_NONEXISTENT_DIRECTORY_HACK 目录其实并不存在。

rootcling 使用带有 特定公式 的特殊头文件来确定为哪部分用于生成字典。文件名可以有前缀，但必须以 LinkDef.h 结尾。编写完成了这个头文件，就可以调用字典中生成的函数了。

手动生成字典

有时，可能想让 ROOT 生成字典，然后自己将源文件添加到库目标中。可以用字典名来调用 root_generate_dictionary 函数（例如：G__Example），任何需要的头文件（以 LINKDEF.h 结尾的头文件）可在 LINKDEF 后列出：

```
root_generate_dictionary(G__Example Example.h LINKDEF
ExampleLinkDef.h)
```

该命令将创建三个文件:

- ${NAME}.cxx：创建库时，这个文件应该包含在源代码中。
- lib{NAME}.rootmap（移除 G__ 前缀）：纯文本的 rootmap 文件
- lib{NAME}_rdict.pcm（移除G__ 前缀）：ROOT 预编译的模块文件 目标名（${NAME}）由字典名决定；若字典名以 G__ 为前缀，会将前缀删除。否则，直接使用目标名。

最后两个输出文件在生成库后产生。可以通过 CMAKE_LIBRARY_OUTPUT_DIRECTORY 进行检查（不会获取本地目标的设置信息）。设置了 libdir，但没有设置（全局）安装位置时，还需要将 ARG_NOINSTALL 设置为 TRUE。

使用现有目标构建字典

可以通过 MODULE 参数，要求 ROOT 来执行相应的操作，而不是手动将生成的内容添加到库源代码中。此参数应该指定构建目标名:

```
add_library(Example)
root_generate_dictionary(G__Example Example.h MODULE Example
LINKDEF ExampleLinkDef.h)
```

字典全名（例如：G__Example）不应该与 MODULE 参数相同。

使用旧版 ROOT

若有必要使用旧版 ROOT，就需要像这样编写 CMake 脚本:

```
# ROOT targets are missing includes and flags in ROOT 6.10
and 6.12
set_property(TARGET ROOT::Core PROPERTY
    INTERFACE_INCLUDE_DIRECTORIES "${ROOT_INCLUDE_DIRS}")

# Early ROOT does not include the flags required on targets
add_library(ROOT::Flags_CXX IMPORTED INTERFACE)


# ROOT 6.14 and earlier have a spacing bug in the linker
flags
string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS
"${ROOT_EXE_LINKER_FLAGS}")

# Fix for ROOT_CXX_FLAGS not actually being a CMake list
separate_arguments(ROOT_CXX_FLAGS)
```

```
set_property(TARGET ROOT::Flags_CXX APPEND PROPERTY
    INTERFACE_COMPILE_OPTIONS ${ROOT_CXX_FLAGS})

# Add definitions
separate_arguments(ROOT_DEFINITIONS)
foreach(_flag ${ROOT_EXE_LINKER_FLAG_LIST})
    # Remove -D or /D if present
    string(REGEX REPLACE [=[^[-//]D]=] "" _flag ${_flag})
    set_property(TARGET ROOT::Flags APPEND PROPERTY
INTERFACE_LINK_LIBRARIES ${_flag})
endforeach()

# This also fixes a bug in the linker flags
separate_arguments(ROOT_EXE_LINKER_FLAGS)
set_property(TARGET ROOT::Flags_CXX APPEND PROPERTY
    INTERFACE_LINK_LIBRARIES ${ROOT_EXE_LINKER_FLAGS})

# Make sure you link with ROOT::Flags_CXX too!
```

使用文件系统的用例

简单的 ROOT 工程

这是使用文件系统，但不使用字典的最小 ROOT 项目示例。

examples/root-usefile/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1...3.21)

project(RootUseFileExample LANGUAGES CXX)

find_package(ROOT 6.16 CONFIG REQUIRED)
# Sets up global settings
include("${ROOT_USE_FILE}")

# This is required for ROOT < 6.16
# string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS
"${ROOT_EXE_LINKER_FLAGS}")

# This is required on if there is more than one flag (like
on macOS)
separate_arguments(ROOT_EXE_LINKER_FLAGS)

add_executable(RootUseFileExample SimpleExample.cxx)
target_link_libraries(RootUseFileExample PUBLIC
${ROOT_LIBRARIES}

${ROOT_EXE_LINKER_FLAGS})
```

examples/root-usefile/SimpleExample.cxx

```
#include <TLorentzVector.h>

int main() {
    TLorentzVector v(1,2,3,4);
    v.Print();
    return 0;
}
```

使用目标系统的用例

简单的 ROOT 工程

这是使用目标系统，但不使用字典的最小 ROOT 项目示例。

examples/root-simple/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1...3.21)

project(RootSimpleExample LANGUAGES CXX)

# Finding the ROOT package
find_package(ROOT 6.16 CONFIG REQUIRED)

# Adding an executable program and linking to needed ROOT
libraries
add_executable(RootSimpleExample SimpleExample.cxx)
target_link_libraries(RootSimpleExample PUBLIC
ROOT::Physics)
```

examples/root-simple/SimpleExample.cxx

```
#include <TLorentzVector.h>

int main() {
    TLorentzVector v(1,2,3,4);
    v.Print();
    return 0;
}
```

使用生成字典的用例

字典用例

这是在 CMake 中构建包含字典模块的例子。通过 `find_package` 手动添加 ROOT，而非使用 ROOT 建议的标志。在大多数系统中，CMake 文件中唯一重要的标志。

examples/root-dict/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.4...3.21)

project(RootDictExample LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 11 CACHE STRING "C++ standard to
use")
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_PLATFORM_INDEPENDENT_CODE ON)

find_package(ROOT 6.20 CONFIG REQUIRED)
# If you want to support <6.20, add this line:
# include("${ROOT_DIR}/modules/RootNewMacros.cmake")
# However, it was moved and included by default in 6.201

root_generate_dictionary(G__DictExample DictExample.h
LINKDEF DictLinkDef.h)

add_library(DictExample SHARED DictExample.cxx DictExample.h
G__DictExample.cxx)
target_include_directories(DictExample PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
target_link_libraries(DictExample PUBLIC ROOT::Core)

## Alternative to add the dictionary to an existing target:
# add_library(DictExample SHARED DictExample.cxx
DictExample.h)
# target_include_directories(DictExample PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}")
# target_link_libraries(DictExample PUBLIC ROOT::Core)
# root_generate_dictionary(G__DictExample DictExample.h
MODULE DictExample LINKDEF DictLinkDef.h)
```

支持文件

这是一个极简的类定义，只有一个成员函数：

examples/root-dict/DictExample.cxx

```
#include "DictExample.h"

Double_t Simple::GetX() const {return x;}

ClassImp(Simple)
```

examples/root-dict/DictExample.h

```
#pragma once

#include <TROOT.h>
```

```
class Simple {
    Double_t x;

public:
    Simple() : x(2.5) {}
    Double_t GetX() const;

    ClassDef(Simple,1)
};
```

还需要一个 LinkDef.h。

examples/root-dict/DictLinkDef.h

```
// See: https://root.cern.ch/selecting-dictionary-entries-
linkdefh
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ nestedclasses;

#pragma link C++ class Simple+;

#endif
```

进行测试

这是一个宏示例，用于测试上面文件生成的结果是否正确。

examples/root-dict/CheckLoad.C

```
{
gSystem->Load("libDictExample");
Simple s;
cout << s.GetX() << endl;
TFile *_file = TFile::Open("tmp.root", "RECREATE");
gDirectory->WriteObject(&s, "MyS");
Simple *MyS = nullptr;
gDirectory->GetObject("MyS", MyS);
cout << MyS->GetX() << endl;
_file->Close();
}
```

Minuit2

Minuit2

Minuit2 在独立模式下可用，用于 ROOT 不可用或未启用 Minuit2 构建的情况。本节会介绍推荐的用法，以及讨论一下集成设计方面的事宜。

用法

无论是使用 ROOT 源码，还是使用单独发行的源码包，Minuit2 都可以使用 CMake 的标准用法：

```
# Check for Minuit2 in ROOT if you want
# and then link to ROOT::Minuit2 instead

add_subdirectory(minuit2) # or root/math/minuit2
# OR
find_package(Minuit2 CONFIG) # Either build or install

target_link_libraries(MyProgram PRIVATE Minuit2::Minuit2)
```

开发

Minuit2 是将现代 CMake（CMake 3.1+）构建，集成到现有框架的解决方案。同时，也是一个很好的例子。

要处理两个不同的 CMake 系统，主 `CMakeLists.txt` 定义一些公共选项，然后使用 `Standalone.cmake` （不作为 ROOT 的一部分构建的情况下）。

ROOT 案例中最困难的部分是，Minuit2 需要 `math/minuit2` 目录之外的文件。这个问题，通过使用`copy_standalone.cmake` 解决了。文件中使用一个函数，该函数接受一个文件名列表，然后返回原始源中的文件名列表；或者将文件复制到本地源中，并返回新位置的列表；亦或者，若原始源不存在（独立模式），则只返回新位置的列表。

```
# Copies files into source directory
cmake /root/math/minuit2 -Dminuit2-standalone=ON

# Makes .tar.gz from source directory
make package_source

# Optional, clean the source directory
make purge
```

但这仅适用于使用源码包的开发人员 —— 普通用户*不会选择该选项*，也不会创建源码的副本。

使用 ROOT 源码，还是使用单独发行的源码包，都可以使用`make install` 或 `make package` （生成二进制包），而无需添加 `standalone` 选项。