

# 目录

Introduction	1.1
现代的 CMake 的介绍	1.2
安装 CMake	1.2.1
运行 CMake	1.2.2
CMake 行为准则	1.2.3
What's new in CMake	1.2.4
基础知识简介	1.3
变量与缓存	1.3.1
用 CMake 进行编程	1.3.2
与你的代码交互	1.3.3
如何组织你的项目	1.3.4
在 CMake 中运行其他程序	1.3.5
一个简单的例子	1.3.6
为 CMake 项目添加特性	1.4
C++11 及后续版本	1.4.1
一些小而常见的需求	1.4.2
一些实用的工具	1.4.3
一些有用的模组	1.4.4
CMake 对 IDE 的支持	1.4.5
调试	1.4.6
包含子项目	1.5
子模组	1.5.1
使用 CMake 下载项目	1.5.2
获取软件包 (FetchContent) (CMake 3.11+)	1.5.3
测试	1.6
GoogleTest	1.6.1
Catch	1.6.2
Exporting and Installing	1.7
Installing	1.7.1
Exporting	1.7.2
Packaging	1.7.3
Looking for Libraries (Packages)	1.8
CUDA	1.8.1
OpenMP	1.8.2
Boost	1.8.3

## UseFile Example

MPI	1.8.4
ROOT	1.8.5
UseFile Example	1.8.5.1
Simple Example	1.8.5.2
Dictionary Example	1.8.5.3
Minuit2	1.8.6

# 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。

它致力于解决网上那些泛滥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/](https://modern-cmake-cn.github.io/Modern-CMake-zh_CN/)

因为参与翻译的同学们都还在上学，因此学期内翻译不会有太多的进展，我们将在假期继续推进。

## 许可协议

本书采用与原书相同的 [LICENSE](#)

## 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](#)！

## 现代的 CMake 的介绍

人们喜爱讨厌构建系统。 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。<sup>1</sup> 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，由于只会 CMake 会启用 CMakeLists.txt 中的生命力的最低 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。

<sup>1</sup>. 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

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/Ubuntu 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<sup>1</sup>将为你安装CMake<sup>2</sup>。

```
~ $ 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

[Chocolatey package](#) 3.23.1 [MSYS2 mingw package](#) 3.23.1

[MSYS2 msys2 package](#) 3.22.1

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

[Homebrew package](#) 3.23.1 [Homebrew Casks package](#) 3.23.1

[MacPorts package](#) 3.22.4

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

Debian 10 package	3.13.4	Debian 10 Backports package	3.18.4
Debian 11 package	3.18.4	Debian 11 Backports package	3.23.1
Debian Unstable package			3.23.1

### 其它Linux发行版

Alpine Linux 3.15 package	3.21.3	Arch package	3.23.1
Fedora 35 package	3.22.2	FreeBSD port	3.22.2
OpenBSD port	3.20.3	Gentoo package	3.23.1
openSUSE Tumbleweed package	3.23.1	Homebrew package	3.23.1

### 常用工具

ConanCenter package	3.23.1		conda-forge v3.23.1	
---------------------	--------	--	---------------------	--

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` ! ↵

[下载]:

# 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` 或这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(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，但是通常都使用小写，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。

## What's new in CMake

This is an abbreviated version of the CMake changelog with just the highlights for authors. Names for each release are arbitrarily picked by the author.

### CMake 3.0 : Interface libraries

There were a ton of additions to this version of CMake, primarily to fill out the target interface. Some bits of needed functionality were missed and implemented in CMake 3.1 instead.

- Initially released [June 10, 2014](#)
- New documentation
- INTERFACE libraries
- Project VERSION support
- Exporting build trees easily
- Bracket arguments and comments available (not widely used)
- Lots of improvements

### CMake 3.1 : C++11 and compile features

This is the first release of CMake to support C++11. Combined with fixes to the new features of CMake 3.0, this is currently a common minimum version of CMake for libraries that want to support old CMake builds.

- Initially released [December 17, 2014](#)
- C++11 Support
- Compile features support
- Sources can be added later with `target_sources`
- Better support for generator expressions and INTERFACE targets

### CMake 3.2 : UTF8

This is a smaller release, with mostly small features and fixes. Internal changes, like better Windows and UTF8 support, were the focus.

- Initially released [March 11, 2015](#)
- `continue()` inside loops
- File and directory locks added

### CMake 3.3 : if IN\_LIST

This is notable for the useful `IN_LIST` option for `if`, but it also added better library search using `$PATH` (See CMake 3.6), dependencies for INTERFACE libraries, and several other useful improvements. The addition of a `COMPILER_LANGUAGE`

generator expression would prove very useful in the future as more languages are added. Makefiles now produce better output in parallel.

- Initially released [July 23, 2015](#)
- `IN_LIST` added to `if`
- `*_INCLUDE_WHAT_YOU_USE` property added
- `COMPILE_LANGUAGE` generator expression (limited support in some generators)

## CMake 3.4 : Swift & CCache

This release adds lots of useful tools, support for the Swift language, and the usual improvements. It also started supporting compiler launchers, like CCache.

- Initially released [November 12, 2015](#)
- Added `Swift` language
- Added `BASE_DIR` to `get_filename_component`
- `if(TEST ...)` added
- `string(APPEND ...)` added
- `CMAKE_*_COMPILER_LAUNCHER` added for make and ninja
- `TARGET_MESSAGES` allow makefiles to print messages after target is completed
- Imported targets are beginning to show up in the official `Find*.cmake` files

## CMake 3.5 : ARM

This release expanded CMake to more platforms, and make warnings easier to control from the command line.

- Initially released [March 8, 2016](#)
- Multiple input files supported for several of the `cmake -E` commands.
- `cmake_parse_arguments` now builtin
- Boost, GTest, and more now support imported targets
- ARMCC now supported, better support for iOS
- XCode backslash fix

## CMake 3.6 : Clang-Tidy

This release added Clang-Tidy support, along with more utilities and improvements. It also removed the search of `$PATH` on Unix systems due to problems, instead users should use `$CMAKE_PREFIX_PATH`.

- Initially released [July 7, 2016](#)
- `EXCLUDE_FROM_ALL` for install
- `list(FILTER` added
- `CMAKE_*_STANDARD_INCLUDE_DIRECTORIES` and `CMAKE_*_STANDARD_LIBRARIES` added for toolchains
- Try-compile improvements
- `*_CLANG_TIDY` property added
- External projects can now be shallow clones, and other improvements

## CMake 3.7 : Android & CMake Server

You can now cross-compile to Android. Useful new if statement options really help clarify code. And the new server mode was supposed to improve integration with IDEs (but is being replaced by a different system in CMake 3.14+). Support for the VIM editor was also improved.

- Initially released [November 11, 2016](#)
- `PARSE_ARGV` mode for `cmake_parse_arguments`
- Better 32-bit support on 64-bit machines
- Lots of useful new if comparisons, like `VERSION_GREATER_EQUAL` (really, why did it take this long?)
- `LINK_WHAT_YOU_USE` added
- Lots of custom properties related to files and directories
- CMake Server added
- Added `--trace-source="filename"` to monitor certain files only

## CMake 3.8 : C# & CUDA

This adds CUDA as a language, as well as `cxx_std_11` as a compiler meta-feature. The new generator expression could be really useful if you can require CMake 3.8+!

- Initially released [April 10, 2017](#)
- Native support for C# as a language
- Native support for CUDA as a language
- Meta features `cxx_std_11` (and 14, 17) added
- `try_compile` has better language support
- `BUILD_RPATH` property added
- `COMPILE_FLAGS` now supports generator expression
- `*_CPPLINT` added
- `$<IF:cond,true-value,false-value>` added (wow!)
- `source_group(TREE` added (finally allowing IDEs to reflect the project folder structure!)

## CMake 3.9 : IPO

Lots of fixes to CUDA support went into this release, including `PTX` support and MSVC generators. Interprocedural Optimizations are now supported properly.

Even more modules provide imported targets, including MPI.

- Initially released [July 18, 2017](#)
- CUDA supported for Windows
- Better object library support in several situations
- `DESCRIPTION` added to `project`
- `separate_arguments` gets `NATIVE_COMMAND`
- `INTERPROCEDURAL_OPTIMIZATION` enforced (and `CMAKE_*` initializer added, CheckIPOSupported added, Clang and GCC support)

- New `GoogleTest` module
- `FindDoxygen` drastically improved

## CMake 3.10 : CppCheck

CMake now is built with C++11 compilers. Lots of useful improvements help write cleaner code.

- Initially released [November 20, 2017](#)
- Support for flang Fortran compiler
- Compiler launcher added to CUDA
- Indented `#cmakedefines` now supported for `configure_file`
- `include_guard()` added to ensure a file gets included only once
- `string(PREPEND` added
- `*_CPPCHECK` property added
- `LABELS` added to directories
- FindMPI vastly expanded
- FindOpenMP improved
- Dynamic test discovery for `GoogleTest`

## CMake 3.11 : Faster & IMPORTED INTERFACE

This release is [supposed to be](#) much faster. You can also finally directly add INTERFACE targets to IMPORTED libraries (the internal `Find*.cmake` scripts should become much cleaner eventually).

- Initially released [March 28, 2018](#)
- Fortran supports compiler launchers
- Xcode and Visual Studio support `COMPILE_LANGUAGE` generator expressions finally
- You can now add INTERFACE targets directly to IMPORTED INTERFACE libraries (Wow!)
- Source file properties have been expanded
- `FetchContent` module now allows downloads to happen at configure time (Wow)

## CMake 3.12 : Version ranges and CONFIGURE\_DEPENDS

Very powerful release, containing lots of smaller long-requested features. One of the smaller but immediately noticeable changes is the addition of version ranges; you can now set both the minimum and maximum known CMake version easily. You can also set `CONFIGURE_DEPENDS` on a `GLOB` ed set of files, and the build system will check those files and rerun if needed! You can use the general

`PackageName_ROOT` for all `find_package` searches. Lots of additions to strings and lists, module updates, shiny new Python find module (2 and 3 versions too), and many more.

- Initially released [July 17, 2018](#)
- Support for `cmake_minimum_required` ranges (backward compatible)
- Support for `-j,--parallel` in `--build` mode (passed on to build tool)
- Support for `SHELL:` strings in compile options (not deduplicated)
- New FindPython module
- `string(JOIN` and `list(JOIN`, and `list(TRANSFORM`
- `file(TOUCH` and `file(GLOB CONFIGURE_DEPENDS`
- C++20 support
- CUDA as a language improvements: CUDA 7 and 7.5 now supported
- Support for OpenMP on macOS (command line only)
- Several new properties and property initializers
- CPack finally reads `CMAKE_PROJECT_VERSION` variables

## CMake 3.13 : Linking control

You can now make symbolic links on Windows! Lots of new functions that fill out the popular requests for CMake, such as `add_link_options`, `target_link_directories`, and `target_link_options`. You can now do quite a bit more modification to targets outside of the source directory, for better file separation. And, `target_sources` *finally* handles relative paths properly (policy 76).

- Initially released [November 20, 2018](#)
- New `ctest --progress` option for live output
- `target_link_options` and `add_link_options` added
- `target_link_directories` added
- Symbolic link creation, `-E create_symlink`, supported on Windows
- IPO supported on Windows
- You can use `-s` and `-B` for source and build directories
- `target_link_libraries` and `install` work outside the current target directory
- `STATIC_LIBRARY_OPTIONS` property added
- `target_sources` is now relative to the current source directory (CMP0076)
- If you use Xcode, you now can experimentally set schema fields

## CMake 3.14 : File utilities (AKA CMake π)

This release has lots of small cleanups, including several utilities for files. Generator expressions work in a few more places, and list handling is better with empty variables. Quite a few more find packages produce targets. The new Visual Studio 16 2019 generator is a bit different than older versions. Windows XP and Vista support has been dropped.

- Initially released [March 14, 2019](#)

- The `cmake --build` command gained `-v/-verbose`, to use verbose builds if your build tool supports it
- The `FILE` command gained `CREATE_LINK`, `READ_SYMLINK`, and `SIZE`
- `get_filename_component` gained `LAST_EXT` and `NAME_WLE` to access just the *last* extension on a file, which would get `.zip` on a file such as `version.1.2.zip` (very handy!)
- You can see if a variable is defined in the `CACHE` with `DEFINED CACHE{VAR}` in an `if` statement.
- `BUILD_RPATH_USE_ORIGIN` and CMake version were added to improve handling of RPath in the build directory.
- The CMake server mode is now being replaced with a file API, starting in this release. Will affect IDEs in the long run.

## CMake 3.15 : CLI upgrade

This release has many smaller polishing changes, include several of improvements to the CMake command line, such as control over the default generator through environment variables (so now it's easy to change the default generator to Ninja). Multiple targets are supported in `--build` mode, and `--install` mode added. CMake finally supports multiple levels of logging. Generator expressions gained a few handy tools. The still very new FindPython module continues to improve, and FindBoost is now more inline with Boost 1.70's new `CONFIG` module. `export(PACKAGE)` has drastically changed; it now no longer touches `$HOME/.cmake` by default (if CMake Minimum version is 3.15 or higher), and requires an extra step if a user wants to use it. This is generally less surprising.

- Initially released [July 17, 2019](#)
- `CMAKE_GENERATOR` environment variable added to control default generator
- Multiple target support in build mode, `cmake . --build --target a b`
- Shortcut `-t` for `--target`
- Install support, `cmake . --install`, does not invoke the build system
- Support for `--loglevel` and `NOTICE`, `VERBOSE`, `DEBUG`, and `TRACE` for message
- The `list` command gained `PREPEND`, `POP_FRONT`, and `POP_BACK`
- `execute_process` gained `COMMAND_ECHO` option  
(`CMAKE_EXECUTE_PROCESS_COMMAND_ECHO`) allows you to automatically echo commands before running them
- Several Ninja improvements, include SWIFT language support
- Compiler and list improvements to generator expressions

## CMake 3.16 : Unity builds

A new unity build mode was added, allowing source files to be merged into a single build file. Support for precompiled headers (possibly preparing for C++20 modules, perhaps?) was added. Lots of other smaller fixes were implemented, especially to newer features, such as to FindPython, FindDoxygen, and others.

- Initially released [November 26, 2019](#)
- Added support for Objective C and Objective C++ languages
- Support for precompiling headers, with `target_precompile_headers`
- Support for "Unity" or "Jumbo" builds (merging source files) with `CMAKE_UNITY_BUILD`
- CTest: Can now skip based on regex, expand lists
- Several new features to control RPath.
- Generator expressions work in more places, like build and install paths
- Find locations can now be explicitly controlled through new variables

## CMake 3.17 : More CUDA

A FindCUDA Toolkit was finally added, which allows finding and using the CUDA toolkit without enabling the CUDA language! CUDA now is a bit more configurable, such as linking to shared libraries. Quite a bit more polish in the expected areas, as well, like FindPython. Finally, you can now iterate over multiple lists at a time.

- Initially released [March 20, 2020](#)
- `CUDA_RUNTIME_LIBRARY` can finally be set to Shared!
- FindCUDA Toolkit finally added
- `cmake -E rm` replaces older remove commands
- CUDA has meta features like `cuda_std_03`, etc.
- You can track the searches for packages with `--debug-find`
- ExternalProject can now disable recursive checkouts
- FindPython better integration with Conda
- DEPRECATION can be applied to targets
- CMake gained a `rm` command
- Several new environment variables
- `foreach` can now do `ZIP_LISTS` (multiple lists at a time)

## CMake 3.18 : CUDA with Clang & CMake macro language

CUDA now supports Clang (without separable compilation). A new `CUDA_ARCHITECTURES` property was implemented to better support targeting CUDA hardware. A new `cmake_language` command supports calling `cmake` commands and expressions from strings. Lots of other meta changes that could make new designs available; calling functions by variable, evaluating arbitrary CMake by string, and configure files directly from strings. Many other nice tiny features and papercut fixes are sprinkled throughout, a small selection is below.

- Initially released [July 15, 2020](#)
- `cmake` can `cat` files together now
- New profiling mode for `cmake`
- `cmake_language` with `CALL` and `EVAL`
- `export` requires `APPEND` if used multiple times (in CMake language level 3.18+)

- You can archive directly from `file()`
- `file(CONFIGURE` is a nicer form of `configure_file` if you already have a string to produce
- Other `find_*` commands gain `find_package`'s `REQUIRED` flag
- NATURAL SORTING in `list(SORT` added
- More options for handling properties with DIRECTORY scope
- `CUDA_ARCHITECTURES` was added
- New `LINK_LANGUAGE` generator expressions (`DEVICE / HOST` versions too)
- Source can be a subdirectory for `FetchContent`

## CMake 3.19 : Presets

You can now add presets in JSON form, and users will get the preset default.

`find_package` can now take a version range, and some specialty find modules, like `FindPython`, have custom support for it. A lot of new controls were added for permissions. Further support for generator expressions in more places.

- New [CMake presets files](#) now supported - you can set defaults for your project per generator, or you can make User presets. PSA: Please add `CMakeUserPresets.json` to your `.gitignore`, even if you do not use `CMakePresets.json`.
- CMake now uses the new build system introduced in XCode 12+
- MSVC for Android now supported
- `cmake -E create_hardlink` was added
- `add_test` finally properly supports whitespace in test names
- You can now `DEFER cmake_language` to run at the end of the directory processing
- Lots of new `file` options, like temporary downloads and `COMPRESSION_LEVEL` for `ARCHIVE_CREATE`
- `find_package` supports a version range
- `DIRECTORY` can now include a binary directory in property commands
- New `JSON` commands for `string`
- New `OPTIMIZE_DEPENDENCIES` property and `CMAKE_*` variable for smartly dropping dependencies of static and object libraries.
- PCH support expanded with `PCH_INSTANTIATE_TEMPLATES` property and `CMAKE_*` variable.
- Check modules have been expanded with `CUDA` and `ISPC` languages
- `FindPython`: `Python*_LINK_OPTIONS` added
- `compute-sanitizer` for `ctest` now supports CUDA for memcheck

## CMake 3.20 : Docs

The CMake docs received a major boost in productivity by adding "new in" tags to quickly see what was added without having to toggle documentation versions!

C++ 23 support added. Source files must have the extension listed now, and `LANGUAGE` is always respected. Quite a bit of cleanup was done; make sure your code is tested with `...3.20` before deploying that as your maximum.

Presets continue to be improved.

- Support added for C++23
- CUDAARCHS environment variable for setting CUDA architectures
- The new `IntelLLVM` compilers are now supported (OneAPI 2021.1), and `NVHPC` NVIDIA HPC SDK, as well
- Some expanded generator expression support in custom commands/targets, install renaming
- New `cmake_path` command for working with paths
- `try_run` now has a `WORKING_DIRECTORY`
- More features for the `file(GENERATE` command
- Several removals, like `cmake-server`, `writeCompilerDetectionHeader` (if policy set to 3.20+), and a few things that have newer methods now.
- Source files must include the extension

## CMake 3.21 : Colors

Different message types now have different colors! There's now a nice variable to see if you are in the top level project. Lots of continued cleanup and specialized new features, such as adding the HIP language and C17 and C23 support. Presets continue to be improved.

- Preliminary support for MSVC 2022
- `CMAKE_<LANG_LINKER_LAUNCHER` added for make and ninja
- HIP added as a language
- C17 and C23 support added
- `--instal-prefix <dir>` and `--toolchain <file>` added when running CMake
- Messages printed are colored by message type!
- Support for MSYS, including `FindMsys`
- The `file()` command got several updates, including `EXPAND_TILDE`
- Support for runtime dependencies and artifacts added to `install`
- `PROJECT_IS_TOP_LEVEL` and `<PROJECT-NAME>_IS_TOP_LEVEL` finally added
- Caching improvements for the `find_` commands

# 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<sup>1</sup>。`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<sup>2</sup>！`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）<sup>3</sup>。

## 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)
```

<sup>1</sup>. 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。 ↵

<sup>2</sup>. 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，现在应该不会有这些问题了。 ↵

<sup>3</sup>. `::` 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

## 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}`。<sup>1</sup> CMake 有作用域的概念，在声明一个变量后，你只可以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<sup>2</sup>。

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

## 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', ''
endif()
# If variable does not expand to one of the above, CMake will expand it then t
```

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。但是如果你想让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` 则是需要进行比较的值 (这里的 `KEYWORD` 也允许使用信息表达式)。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则将会被替换

(成 `value` ) , 如果是0则不会替换。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<sup>2</sup>。

译者注：这里有点类似于 C 语言中的条件运算符。这里由于译者英语水平的问题，翻译的不够清楚，后续会改善。

如果你有一个只想在 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>")
```

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, 你应该永远都不要使用配置时间的值作为当前的配置, 因为像 IDE 这种多配置生成器不会在配置过程中生成配置时间, 只有在构建时可以通过生成器表达式和 `*_<CONFIG>` 这类变量可以获得。

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：

```
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。<sup>1</sup>

<sup>1</sup>. 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

## 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。

译者注：这里原文讲的有些太略，后续补充内容。

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，上述方式将是处理版本的最优方案。可以像下面这么写：

## UseFile Example

```
# 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

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](https://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：

## UseFile Example

```
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 containing one) that contains CMakeLists.txt")
endif()
```

可以在这里查看 [拓展代码样例](#)

# 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 CMkae 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 result ${GIT_SUBMOD_RESULT}")
    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/generate_header.py"
                  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。

译者注：这里翻译的有一些拗口，后续会改善。

## 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，一个非常有用的是 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

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
# even flags, so linking a target that does not exist will not give a configuration 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

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

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。

## 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，你就可以不用关系编译器的版本，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 `-fIto` 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()
```

## 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<sup>1</sup>

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<sup>2</sup>

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<sup>3</sup>

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<sup>4</sup>

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。

<sup>1</sup>. 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<sup>3</sup>。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](#)。<sup>4</sup>

<sup>2</sup>. **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。<sup>5</sup>

<sup>3</sup>. 一个与 **clang** 一起使用，用于分析 C 和 C++ 源文件中 `#include` 的工具。<sup>6</sup>

<sup>4</sup>. 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。<sup>7</sup>

## 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：

## UseFile Example

```
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

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。通过这种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")
```

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 `cmakePrintHelpers` 可以更方便的打印变量：

```
include(cmakePrintHelpers)
cmake_print_variables(MY_VARIABLE)
```

如何你只是想要打印一个变量，那么上述方法已经很好用了！如何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

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。

## 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")
    endif()
endif()
endif()

if(NOT EXISTS "${PROJECT_SOURCE_DIR}/extern/repo/CMakeLists.txt")
    message(FATAL_ERROR "The submodules were not downloaded! GIT_SUBMODULE was"
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

### 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）<sup>1</sup>

<sup>1</sup>. 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+)

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

## 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 BL
    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 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：<sup>1</sup>

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

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 executable in which the tests will be stored
    add_executable(${TESTNAME} ${ARGN})
    # link the Google test infrastructure, mocking library, and a default main
    # the test executable. Remove g_test_main if writing your own main function
    target_link_libraries(${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
    gtest_discover_tests(${TESTNAME})
    # set a working directory so your project root so that you can find test files
    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}/external/gtest")
```

## 下载的方式

你可以通过 CMake 的 `include` 指令使用我在 [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()
```

<sup>1</sup>. 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

[Catch2](#) (C++11 only version) is a powerful, idiomatic testing solutions similar in philosophy to PyTest for Python. It supports a wider range of compilers than GTest, and is quick to support new things, like M1 builds on macOS. It also has a smaller but faster twin, [doctest](#), which is quick to compile but misses features like matchers. To use Catch in a CMake project, there are several options.

## Configure methods

Catch has nice CMake support, though to use it, you need the full repo. This could be with submodules or FetchContent. Both the [extended-project](#) and [fetch](#) examples use FetchContent. See [the docs](#).

## Quick download

This is likely the simplest method and supports older versions of CMake. You can download the all-in-one header file in one step:

```
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=681e7505a50887c9085539e5135794fc8f66d8e5de28eadf13a30978
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}")
```

This will two downloads when Catch 3 is released, as that now requires two files (but you no longer have to write a main.cpp). The `main.cpp` looks like this:

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

## Vendoring

If you simply drop in the single include release of Catch into your project, this is what you would need to add Catch:

## UseFile Example

```
# 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}"
```

Then, you would link to Catch2::Catch. This would have been okay as an INTERFACE target since you won't be exporting your tests.

## Direct inclusion

If you add the library using ExternalProject, FetchContent, or git submodules, you can also `add_subdirectory Catch (CMake 3.1+)`.

Catch also provides two CMake modules that you can use to register the individual tests with CMake.

# Exporting and Installing

There are three good ways and one bad way to allow others use your library:

## Find module (the bad way)

If you are the library author, don't make a `Find<mypackage>.cmake` script! These were designed for libraries whose authors did not support CMake. Use a `Config<mypackage>.cmake` instead as listed below.

## Add Subproject

A package can include your project in a subdirectory, and then use `add_subdirectory` on the subdirectory. This useful for header-only and quick-to-compile libraries. Note that the install commands may interfere with the parent project, so you can add `EXCLUDE_FROM_ALL` to the `add_subdirectory` command; the targets you explicitly use will still be built.

In order to support this as a library author, make sure you use `CMAKE_CURRENT_SOURCE_DIR` instead of `PROJECT_SOURCE_DIR` (and likewise for other variables, like binary dirs). You can check `CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME` to only add options or defaults that make sense if this is a project.

Also, since namespaces are a good idea, and the usage of your library should be consistent with the other methods below, you should add

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

to standardise the usage across all methods. This ALIAS target will not be exported below.

## Exporting

The third way is `*Config.cmake` scripts; that will be the topic of the next chapter in this session.

# Installing

Install commands cause a file or target to be "installed" into the install tree when you `make install`. Your basic target install command looks like this:

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

The various destinations are only needed if you have a library, static library, or program to install. The includes destination is special; since a target does not install includes. It only sets the includes destination on the exported target (which is often already set by `target_include_directories`, so check the `MyLibTargets` file and make sure you don't have the include directory included twice if you want clean cmake files).

It's usually a good idea to give CMake access to the version, so that `find_package` can have a version specified. That looks like this:

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

You have two choices next. You need to make a `MyLibConfig.cmake`, but you can do it either by exporting your targets directly to it, or by writing it by hand, then including the targets file. The later option is what you'll need if you have any dependencies, even just OpenMP, so I'll illustrate that method.

First, make an install targets file (very similar to the one you made in the build directory):

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

This file will take the targets you exported and put them in a file. If you have no dependencies, just use `MyLibConfig.cmake` instead of `MyLibTargets.cmake` here. Then write a custom `MyLibConfig.cmake` file in your source tree somewhere. If you want to capture configure time variables, you can use a `.in` file, and you will want to use the `@var@` syntax. The contents that look like this:

## UseFile Example

```
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")
```

Now, you can use `configure` file (if you used a `.in` file) and then install the resulting file. Since we've made a `ConfigVersion` file, this is a good place to install it too.

```
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
       )
```

That's it! Now once you install a package, there will be files in `lib/cmake/MyLib` that CMake will search for (specifically, `MyLibConfig.cmake` and `MyLibConfigVersion.cmake`), and the targets file that config uses should be there as well.

When CMake searches for a package, it will look in the current install prefix and several standard places. You can also add this to your search path manually, including `MyLib_PATH`, and CMake gives the user nice help output if the `configure` file is not found.

# Exporting

The default behavior for exporting changed in CMake 3.15. Since changing files in a user's home directory is considered "surprising" (and it is, which is why this chapter exists), it is no longer the default behavior. If you set a minimum or maximum CMake version of 3.15 or later, this will no longer happen unless you set `CMAKE_EXPORT_PACKAGE_REGISTRY` as shown below.

There are three ways to access a project from another project: subdirectory, exported build directories, and installing. To use the build directory of one project in another project, you will need to export targets. Exporting targets is needed for a proper install; allowing the build directory to be used is just two added lines. It is not generally a way to work that I would recommend, but can be useful for development and as way to prepare the installation procedure discussed later.

You should make an export set, probably near the end of your main

`CMakeLists.txt` :

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

This puts the targets you have listed into a file in the build directory, and optionally prefixes them with a namespace. Now, to allow CMake to find this package, export the package into the `$HOME/.cmake/packages` folder:

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

Now, if you `find_package(MyLib)`, CMake can find the build folder. Look at the generated `MyLibTargets.cmake` file to help you understand exactly what is created; it's just a normal CMake file, with the exported targets.

Note that there's a downside: if you have imported dependencies, they will need to be imported before you `find_package`. That will be fixed in the next method.

# Packaging

There are two ways to instruct CMake to build your package; one is to use a CPackConfig.cmake file, and the other is to integrate the CPack variables into your CMakeLists.txt file. Since you want variables from your main build to be included, like version number, you'll want to make a configure file if you go that route. I'll show you the integrated version:

```
# 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")
```

These are the most common variables you'll need to make a binary package. A binary package uses the install mechanism of CMake, so anything that is installed will be present.

You can also make a source package. You should set `CMAKE_SOURCE_IGNORE_FILES` to regular expressions that ensure you don't pick up any extra files (like the build directory or git details); otherwise `make package_source` will bundle up literally everything in the source directory. You can also set the source generator to make your favorite types of files for source packages:

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

Note that this will not work on Windows, but the generated source packages work on Windows.

Finally, you need to include the CPack module:

```
include(CPack)
```

## Finding Packages

There are two ways to find packages in CMake: "Module" mode and "Config" mode.

## CUDA

CUDA support is available in two flavors. The new method, introduced in CMake 3.8 (3.9 for Windows), should be strongly preferred over the old, hacky method - I only mention the old method due to the high chances of an old package somewhere having it. Unlike the older languages, CUDA support has been rapidly evolving, and building CUDA is hard, so I would recommend you *require a very recent version* of CMake! CMake 3.17 and 3.18 have a lot of improvements directly targeting CUDA.

A good resource for CUDA and Modern CMake is [this talk](#) by CMake developer Robert Maynard at GTC 2017.

## Adding the CUDA Language

There are two ways to enable CUDA support. If CUDA is not optional:

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

You'll probably want `cxx` listed here also. And, if CUDA is optional, you'll want to put this in somewhere conditionally:

```
enable_language(CUDA)
```

To check to see if CUDA is available, use `CheckLanguage`:

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

You can see if CUDA is present by checking `CMAKE_CUDA_COMPILER` (was missing until CMake 3.11).

You can check variables like `CMAKE_CUDA_COMPILER_ID` (for nvcc, this is `"NVIDIA"`, Clang was added in CMake 3.18). You can check the version with

```
CMAKE_CUDA_COMPILER_VERSION
```

## Variables for CUDA

Many variables with `cxx` in the name have a CUDA version with `cuda` instead. For example, to set the C++ standard required for CUDA,

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

If you are looking for CUDA's standard level, in CMake 3.17 a new collection of compiler features were added, like `cuda_std_11`. These have the same benefits that you are already used to from the `cxx` versions.

## Adding a library / executable

This is the easy part; as long as you use `.cu` for CUDA files, you can just add libraries *exactly like you normally would*.

You can also use separable compilation:

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

You can also directly make a PTX file with the `CUDA_PTX_COMPILATION` property.

## Targeting architectures

When you build CUDA code, you generally should be targeting an architecture. If you don't, you compile 'ptx', which provide the basic instructions but is compiled at runtime, making it potentially much slower to load.

All cards have an architecture level, like "7.2". You have two choices; the first is the code level; this will report to the code being compiled a version, like "5.0", and it will take advantage of all the features up to 5.0 but not past (assuming well written code / standard libraries). Then there's a target architecture, which must be equal or greater to the code architecture. This needs to have the same major number as your target card, and be equal to or less than the target card. So 7.0 would be a common choice for our 7.2 card. Finally, you can also generate PTX; this will work on all future cards, but will compile just in time.

In CMake 3.18, it became very easy to target architectures. If you have a version range that includes 3.18 or newer, you will be using `CMAKE_CUDA_ARCHITECTURES` variable and the `CUDA_ARCHITECTURES` property on targets. You can list values (without the `.`), like 50 for arch 5.0. If set to OFF, it will not pass architectures.

## Working with targets

Using targets should work similarly to CXX, but there's a problem. If you include a target that includes compiler options (flags), most of the time, the options will not be protected by the correct includes (and the chances of them having the correct CUDA wrapper is even smaller). Here's what a correct compiler options line should look like:

```
"$<<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>>; -fopenmp>$<<BUILD_INTERFACE:$<C
```

However, if you using almost any `find_package`, and using the Modern CMake methods of targets and inheritance, everything will break. I've learned that the hard way.

For now, here's a pretty reasonable solution, as long as you know the un-aliased target name. It's a function that will fix a C++ only target by wrapping the flags if using a CUDA compiler:

```
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:CXX>}:${old_flags}}")
    endif()
endfunction()
```

## Useful variables

- `CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES` : Place for built-in Thrust, etc
- `CMAKE_CUDA_COMPILER` : NVCC with location

You can use `FindCUDAToolkit` to find a variety of useful targets and variables even without enabling the CUDA language.

**Note that FindCUDA is deprecated, but for versions of CMake < 3.18, the following functions required FindCUDA:**

- CUDA version checks / picking a version
- Architecture detection (Note: 3.12 fixes this partially)
- Linking to CUDA libraries from non-.cu files

## Classic FindCUDA [WARNING: DO NOT USE] (for reference only)

If you want to support an older version of CMake, I recommend at least including the `FindCUDA` from CMake version 3.9 in your `cmake` folder (see the `CL_Utils` github organization for a [git repository](#)). You'll want two features that were added:

`CUDA_LINK_LIBRARIES_KEYWORD` and `cuda_select_nvcc_arch_flags`, along with the newer architectures and CUDA versions.

To use the old CUDA support, you use `find_package` :

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

You can control the CUDA flags with `CUDA_NVCC_FLAGS` (list append) and you can control separable compilation with `CUDA_SEPARABLE_COMPILATION`. You'll also want to make sure CUDA plays nice and adds keywords to the targets (CMake 3.9+):

```
set(CUDA_LINK_LIBRARIES_KEYWORD PUBLIC)
```

## UseFile Example

You'll also might want to allow a user to check for the arch flags of their current hardware:

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

# OpenMP

OpenMP support was drastically improved in CMake 3.9+. The Modern(TM) way to add OpenMP to a target is:

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

This not only is cleaner than the old method, it will also correctly set the library link line differently from the compile line if needed. In CMake 3.12+, this will even support OpenMP on macOS (if the library is available, such as with `brew install libomp`). However, if you need to support older CMake, the following works on CMake 3.1+:

```
# 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
    set_property(TARGET OpenMP::OpenMP_CXX
        PROPERTY INTERFACE_LINK_LIBRARIES ${OpenMP_CXX_FLAGS} Threads)

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

Warning: CMake < 3.4 has a bug in the Threads package that requires you to have the `c` language enabled.

## Boost library

The Boost library is included in the find packages that CMake provides, but it has a couple of oddities in how it works. See [FindBoost](#) for a full description; this will just give a quick overview and provide a recipe. Be sure to check the page for the minimum required version of CMake you are using and see what options you have.

First, you can customize the behavior of the Boost libraries selected using a set of variables that you set before searching for Boost. There are a growing number of settings, but here are the three most common ones:

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

In CMake 3.5, imported targets were added. These targets handle dependencies for you as well, so they are a very nice way to add Boost libraries. However, CMake has the dependency information baked into it for all known versions of Boost, so CMake must be newer than Boost for these to work. In a recent [merge request](#), CMake started assuming that the dependencies hold from the last version it knows about, and will use that (along with giving a warning). This functionality was backported into CMake 3.9.

The import targets are in the `Boost::` namespace. `Boost::boost` is the header only part. The other compiled libraries are available, and include dependencies as needed.

Here is an example for using the `Boost::filesystem` library:

```
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

To add MPI, like OpenMP, you'll be best off with CMake 3.9+.

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

However, you can imitate this on CMake 3.1+ with:

```
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_LINK_FLAGS})
endif()

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

# ROOT

ROOT is a C++ Toolkit for High Energy Physics. It is huge. There are really a lot of ways to use it in CMake, though many/most of the examples you'll find are probably wrong. Here's my recommendation.

Most importantly, there are *lots of improvements* in CMake support in more recent versions of ROOT - Using 6.16+ is much, much easier! If you really must support 6.14 or earlier, see the section at the end. There were further improvements in 6.20, as well, it behaves much more like a proper CMake project, and exports C++ standard features for targets, etc.

## Finding ROOT

ROOT 6.10+ supports config file discovery, so you can just do:

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

to attempt to find ROOT. If you don't have your paths set up, you can pass `-DROOT_DIR=$ROOTSYS/cmake` to find ROOT. (But, really, you should source `thisroot.sh`).

## The right way (Targets)

ROOT 6.12 and earlier do not add the include directory for imported targets. ROOT 6.14+ has corrected this error, and required target properties have been getting better. This method is rapidly becoming easier to use (see the example at the end of this page for the older ROOT details).

To link, just pick the libraries you want to use:

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

If you'd like to see the default list, run `root-config --libs` on the command line. In Homebrew ROOT 6.18 this would be:

- `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

## The old global way

ROOT provides a utility to set up a ROOT project, which you can activate using `include("${ROOT_USE_FILE}")`. This will automatically make ugly directory level and global variables for you. It will save you a little time setting up, and will waste massive amounts of time later if you try to do anything tricky. As long as you aren't making a library, it's probably fine for simple scripts. Includes and flags are set globally, but you'll still need to link to  `${ROOT_LIBRARIES}` yourself, along with possibly `ROOT_EXE_LINKER_FLAGS` (You will have to `separate_arguments` first before linking or you will get an error if there are multiple flags, like on macOS). Also, before 6.16, you have to manually fix a bug in the spacing.

Here's what it would look like:

```
# 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(RootUseFileDialog SimpleExample.cxx)
target_link_libraries(RootUseFileDialog PUBLIC ${ROOT_LIBRARIES}
                      ${ROOT_EXE_LINKER_FLAGS})
```

## Components

Find ROOT allows you to specify components. It will add anything you list to  `${ROOT_LIBRARIES}`, so you might want to build your own target using that to avoid listing the components twice. This did not solve dependencies; it was an error to list `RooFit` but not `RooFitCore`. If you link to `ROOT::RooFit` instead of  `${ROOT_LIBRARIES}`, then `RooFitCore` is not required.

## Dictionary generation

Dictionary generation is ROOT's way of working around the missing reflection feature in C++. It allows ROOT to learn the details of your class so it can save it, show methods in the Cling interpreter, etc. Your source code will need the following modifications to support dictionary generation:

- Your class definition should end with `ClassDef(MyClassName, 1)`
- Your class implementation should have `ClassImp(MyClassName)` in it

ROOT provides `rootcling` and `genreflex` (a legacy interface to `rootcling`) binaries which produce the source files required to build the dictionary. It also defines `root_generate_dictionary`, a CMake function to invoke `rootcling` during the build process.

To load this function, first include the ROOT macros:

```
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()
```

The `if(...)` condition is added to fix a bug in the NewMacros file that causes dictionary generation to fail if there is not at least one global include directory or a `inc` folder. Here I'm including a non-existent directory just to make it work. There is no `ROOT_NONEXISTENT_DIRECTORY_HACK` directory.

`rootcling` uses a special header file with a [specific formula](#) to determine which parts to generate dictionaries for. The name of this file may have any prefix, but **must** end with `LinkDef.h`. Once you have written this header file, the dictionary generation function can be invoked.

## Manually building the dictionary

Sometimes, you might want to ask ROOT to generate the dictionary, and then add the source file to your library target yourself. You can call the

`root_generate_dictionary` with the name of the dictionary, e.g. `G_Example`, any required header files, and finally the special `LinkDef.h` file, listed after `LINKDEF`:

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

This command will create three files:

- `${NAME}.cxx` : This file should be included in your sources when you make your library.
- `lib${NAME}.rootmap` ( `G__` prefix removed): The rootmap file in plain text
- `lib${NAME}_rdict.pcm` ( `G__` prefix removed): A [ROOT pre-compiled module file](#) The name ( `${NAME}`) of the target that you must create is determined by the dictionary name; if the dictionary name starts with `G__`, it will be removed. Otherwise, the name is used directly.

The final two output files must sit next to the library output. This is done by checking `CMAKE_LIBRARY_OUTPUT_DIRECTORY` (it will not pick up local target settings). If you have a `libdir` set but you don't have (global) install locations set, you'll also need to set `ARG_NOINSTALL` to `TRUE`.

## Building the dictionary with an existing target

Instead of manually adding the generated to your library sources, you can ask ROOT to do this for you by passing a `MODULE` argument. This argument should specify the name of an existing build target:

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

The full name of the dictionary (e.g. `G__Example`) should not be identical to the `MODULE` argument.

## Using Old ROOT

If you really have to use older ROOT, you'll need something like this:

```
# 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_CXX 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!
```

# A Simple ROOT Project

This is a minimal example of a ROOT project using the UseFile system and without a dictionary.

## 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;
}
```

# A Simple ROOT Project

This is a minimal example of a ROOT project using the target system and without a dictionary.

## 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;
}
```

## Dictionary Example

This is an example of building a module that includes a dictionary in CMake. Instead of using the ROOT suggested flags, we will manually add threading via `find_package`, which is the only important flag in the list on most systems.

### **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)
```

## Supporting files

This is just a simple-as-possible class definition, with one method:

### **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)
};
```

We need a `LinkDef.h`, as well.

## examples/root-dict/DictLinkDef.h

```
// See: https://root.cern.ch/selecting-dictionary-entries-linkdefh
#ifndef __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
```

## Testing it

This is an example of a macro that tests the correct generation from the files listed above.

## 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 is available in standalone mode, for use in cases where ROOT is either not available or not built with Minuit2 enabled. This will cover recommended usages, as well as some aspects of the design.

## Usage

Minuit2 can be used in any of the standard CMake ways, either from the ROOT source or from a standalone source distribution:

```
# 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)
```

## Development

Minuit2 is a good example of potential solutions to the problem of integrating a modern (CMake 3.1+) build into an existing framework.

To handle the two different CMake systems, the main `CMakeLists.txt` defines common options, then calls a `Standalone.cmake` file if this is not building as part of ROOT.

The hardest part in the ROOT case is that Minuit2 requires files that are outside the `math/minuit2` directory. This was solved by adding a `copy_standalone.cmake` file with a function that takes a filename list and then either returns a list of filenames inplace in the original source, or copies files into the local source and returns a list of the new locations, or returns just the list of new locations if the original source does not exist (standalone).

```
# 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
```

This is only intended for developers wanting to produce source packages - a normal user *does not pass this option* and will not create source copies.

## UseFile Example

You can use `make install` or `make package` (binary packages) without adding this `standalone` option, either from inside the ROOT source or from a standalone package.