## CMake入门07-管理外部依赖(使用第三方库)

### 测试环境：

**Windows 10（Ubuntu）+MinGw（gcc/g++/Ninja.exe）+ Visual Studio 2019 +Clion（克利翁）+QT**

**插图以及内容来自书籍《Modern CMake For C++》**

**辅助工具：ChatGPT**

### 前言：

这节课极其重要，而且偏向实战，内容几乎涵盖了所有使用第三方库的方法

一定要反复认真的观看视频和阅读原书籍，

与许多其他语言相比，C++ 世界中依赖管理和包管理非常困难。

为此微软还有个团队专门在做这个的处理，他们出的工具叫vcpkg；

可见对于大多数c++开发者来说，这的确不容易；

C++20试图通过module来解决这个问题，不过需要时间；

使用CMake的最大优势之一是它有内置的依赖管理，可以发现许多第三方库。

### 本节主要的内容：

* 为什么要使用第三方库；
* 使用库的底层逻辑是什么
* 直接使用第三库的源代码（不推荐）
* 最简单的使用第三方库的方法。
* 如何查找已安装的软件包find\_ package（重点介绍）
* 使用 FindPkgConfig 发现旧版软件包（过时，仅参考）
* 编写自己的查找模块（重点介绍）
* 使用 Git 存储库 （只是了解）
* 使用 ExternalProject 和 FetchContent 模块 （大致了解）
* 有没有更好的依赖解决方案？（推荐）

### 涉及到的cmake命令：

* include\_directories 指定系统包含目录，以供目录范围内的目标使用。
* link\_directories 指定链接器在链接目标时应在其中搜索库的目录。
* target\_include\_directories 指定编译给定目标时要使用的包含目录。这可用于为特定目标或目标指定包含目录。
* target\_link\_libraries 指定要与目标链接的库或目标。默认情况下，链接器将在系统库目录和任何由 link\_directories 命令指定的目录中搜索库。
* find\_file:查找单个文件。
* find\_path:查找包含特定文件的目录。（一般是指头文件）
* find\_library:查找库文件。
* find\_program:查找可执行程序。
* find\_package:查找完整的包集。

### 为什么要使用第三方库？

首先我们看看我写的字幕软件依赖了多少第三方库；

图形用户界面, 文本, 应用程序

描述已自动生成

### 思考：使用库底层的逻辑？

如果我们要使用一个库，我们需要哪些文件；

二进制依赖：

* 头文件；
* 静态库，动态库；

源码依赖：

这里不考虑直接把源码放进来的情况；

为什么需要头文件：（我们之前的课程中说道）

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

### 直接使用第三方库源码（不推荐）

一种相对直接但危险的方法是手动下载或克隆它们到项目中的子文件夹中，然后使用 add\_subdirectory 添加该文件夹。

虽然这有效并且非常快，但它很快就会变得乏味且难以维护。

### 最简单的使用第三方库的方法：

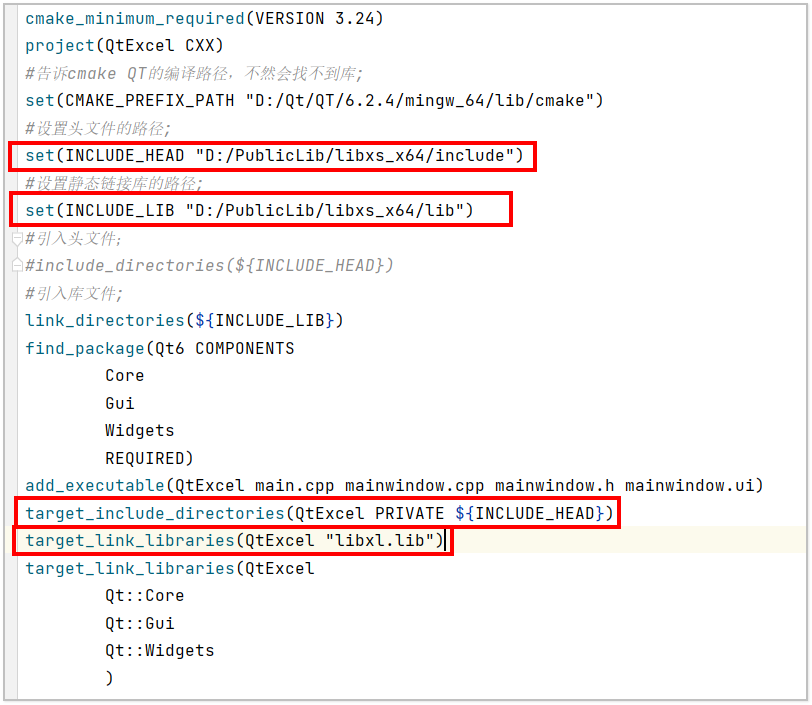
比如我们要使用一个libxl读写Excel的c++库；

官网下载后的Windows版本：

https://www.libxl.com/download.html



CMakelists.txt



思考：动态库怎么链接进来呢？把库里面的dll文件复制到可执行文件下：

文本

描述已自动生成

### 如何查找已安装的软件（find\_ package）-Modern CMake



#### 1.find\_ package命令说明：

图形用户界面, 文本

描述已自动生成

PackageName：表示包名；

MODULE：MODULE关键字意味着只能使用Module模式来查找软件包，没有回退到Config模式

REQUIRED：表示是否必须，如果加了这个关键字，没有找到构建直接失败；

COMPONENTS：在COMPONENTS关键字之后可以列出一个特定于软件包的必需组件列表。 如果其中任何组件无法被满足，则认为未找到软件包。

OPTIONAL\_COMPONENTS：在OPTIONAL\_COMPONENTS之后可以列出其他可选组件。 如果无法满足这些组件，则整个软件包仍然可以视为已找到，只要满足了所有必需组件即可。

图形用户界面, 文本, 应用程序

描述已自动生成

文本

描述已自动生成

CONFIG：选项表示使用指定的配置文件来查找并加载包，通常结合--config选项使用；

HINTS：选项表示提供额外的查找提示路径；

PATHS：选项表示指定查找的路径列表，会在默认路径之前搜索；

NO\_DEFAULT\_PATH：选项表示不使用默认路径，只查找指定路径。

QUIET关键字关闭所有关于已找到/未找到包的消息

文本

描述已自动生成

#### 2.该命令有几种模式来搜索包:

使用find\_package基本签名时，命令先搜索“模块”模式。如果没有找到该包，搜索将回落到Config模式。

##### 第一种：MODULE模块模式

MODULE关键字意味着只能使用MODULE模式来查找包，而不能退回到Config模式。

在这种模式下，CMake搜索名为Find<PackageName>.cmake的文件

首先在CMAKE\_MODULE\_PATH中列出的位置中查找，然后在CMake安装提供的Find模块中查找。

如果找到文件，则由CMake读取和处理。 它负责找到软件包，检查版本并生成任何所需的消息。

Find<PackageName>.cmake文件通常不是由软件包本身提供的，

而是通常由软件包之外的某个东西（例如操作系统，CMake本身或甚至调用find\_package()命令的项目）提供。

###### 在Ubuntu下演示：protobuf

安装 protobuf 库：

$ apt update

$ apt install protobuf-compiler libprotobuf-dev



程序解释：

我已经包含了message.pb.h标题。这个文件还不存在;它需要在message.proto编译期间由Protobuf编译器protoc创建

protobuf\_generate\_cpp是在Protobuf findmodule中定义的自定义函数。在底层，它调用add\_custom\_command()，后者使用适当的参数调用协议编译器。

Protobuf有一个特殊的编译器，它可以读取这些文件并生成c++源文件和头文件，然后供我们的应用程序使用。这

find\_package(Protobuf REQUIRED) 要求CMake 查找指定的FindProtobuf.cmake模块，并为我们设置Protobuf 库

Target\_include\_directories()添加必要的包含路径

由包提供的INCLUDE\_DIRS和CMAKE\_CURRENT\_BINARY\_DIR。

后者是必需的，这样编译器才能找到生成的message.pb.h头文件。

##### 第二种：CONFIG配置模式

find\_package()使用CONFIG 模式查找：

* <CamelCasePackageName>Config.cmake 使用驼峰式命名法
* <kebab-case-package-name>-config.cmake 使用烤串式命名法

举例：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

如果系统下有这两类文件，表示通过CONFIG 找到了；

###### QT的例子：

您可以通过不同的方式告诉 CMake 关于 Qt 的信息，

FindQt6.cmake

Qt6-config.cmake

Qt6Config.cmake

但最常见和推荐的方法是设置 CMake 缓存变量 CMAKE\_PREFIX\_PATH 以包含 Qt 6 安装前缀。

*set(CMAKE\_PREFIX\_PATH "D:/Qt/QT/6.2.4/mingw\_64/lib/cmake")*

图形用户界面, 文本, 应用程序

描述已自动生成

###### 在Windows下演示：protobuf

安装 protobuf 库：

vcpkg install protobuf:x64-windows

*方式1：#set(protobuf\_DIR "D:/PublicLib/vcpkg/installed/x64-windows/share/protobuf")  
方式2：#set(CMAKE\_PREFIX\_PATH "D:/PublicLib/vcpkg/packages/protobuf\_x64-windows")*

*方式3：*

*Clion下面：*

*-DCMAKE\_TOOLCHAIN\_FILE=D:/PublicLib/vcpkg/scripts/buildsystems/vcpkg.cmake*

*CMAKE\_TOOLCHAIN\_FILE* 通常，该变量的值是一个路径，指向一个包含工具链设置的CMake脚本。该脚本中通常包含了编译器、链接器、库文件路径、预定义宏等工具链设置

*微软的vcpkg的官方文档:*

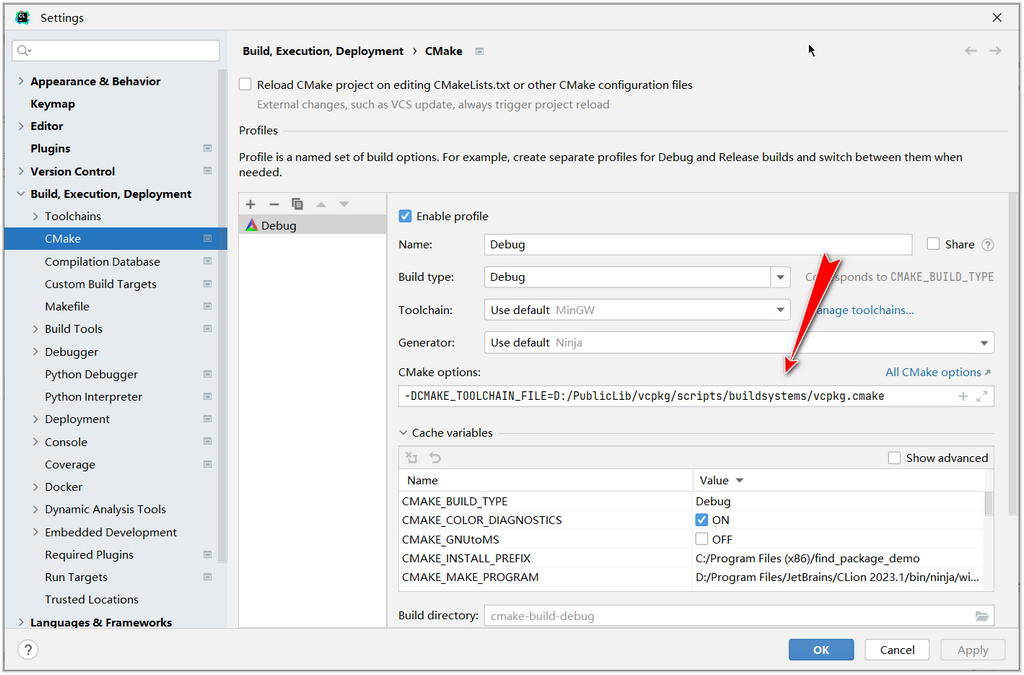
*https://learn.microsoft.com/en-us/vcpkg/users/buildsystems/cmake-integration*

*CMAKE\_PREFIX\_PATH:指定用于指定CMake在查找库文件、头文件和其他依赖项时需要搜索的前缀路径列表；*

*<CamelCasePackageName>\_DIR：指定<CamelCasePackageName>Config.cmake或<kebab-case-package-name>-config.cmake的存放路径；*

图形用户界面, 文本, 应用程序

描述已自动生成



主要如果不加CONFIG关键字，默认是从MODULE到CONFIG;

第三种：FetchContent重定向模式

自版本3.24开始：find\_package()调用可以在内部重定向到FetchContent模块提供的软件包。

对于调用者而言，行为类似于Config模式，但搜索逻辑被绕过，不使用组件信息；

#### 3.查找结果：

在大多数情况下，您可以期望在调用 find\_package() 时设置一些变量，无论您使用的是内置查找模块还是与包捆绑在一起的配置文件（假设已找到包）：

* <PKG\_NAME>\_FOUND 查找结果；
* <PKG\_NAME>\_INCLUDE\_DIRS or <PKG\_NAME>\_INCLUDES 头文件；
* <PKG\_NAME>\_LIBRARY or <PKG\_NAME>\_LIBRARIES or <PKG\_NAME>\_LIBS 库文件；
* <PKG\_NAME>\_DEFINITIONS
* IMPORTED targets specified by the find-module or config-file 由查找模块或配置文件指定的导入目标

#### 4.现代的cmake，直接导入目标

最后一点非常有趣——如果一个包支持所谓的“现代 CMake”（围绕目标构建），

它将提供那些 IMPORTED 目标而不是（或与这些变量一起），

从而允许更清晰、更简单的代码。建议优先考虑目标而不是变量。

Protobuf 是一个很好的例子，因为它同时提供变量和导入目标

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

protobuf::libprotobuf 导入目标隐式指定包含目录，并且由于传递依赖性（或我称之为传播属性），它们与我们的主要目标共享。

### 使用 FindPkgConfig 发现旧版包-过时的方法；

管理依赖关系和发现它们所需的所有编译标志的问题与c++库本身一样古老。

有许多工具可以处理它，从非常小和最小的机制到作为构建系统和ide的一部分提供的非常通用的解决方案。其中一个(曾经非常流行的)工具叫做PkgConfig

(freedesktop.org/wiki/Software/pkg-config/)。

它通常在类unix系统上可用(尽管它也适用于macOS和Windows)。

python-3.9.pc

库的作者提供了一个小的 .pc 文件，其中包含编译和链接所需的详细信息，例如这个

Pkg-config的库举例：

图形用户界面, 文本, 应用程序

描述已自动生成

为了在构建场景中正确地使用PkgConfig，构建系统必须在操作系统中找到pkg-config可执行文件，

运行几次并提供适当的参数，并将响应存储在变量中，以便稍后将它们传递给编译器。

幸运的是，CMake提供了一个方便的内置查找模块

FindPkgConfig。它遵循常规查找模块的大多数规则

提供 .pc 文件的比较流行的库的一个示例是 PostgreSQL 数据库的客户端 – libpqxx。

举例：PostgreSQL database – libpqxx

Ubuntu下安装libpqxx

因为这个包刚好是旧的机制打包的；

apt-get install libpqxx-dev

使用如下：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

1. 我们要求CMake用find\_package()命令找到PkgConfig可执行文件。如果pkg-config因为REQUIRED关键字而不存在，则会失败。
2. 调用FindPkgConfig查找模块中定义的pkg\_check\_modules()自定义宏，以创建一个新的IMPORTED目标，选择PQXX作为名称。find模块将搜索名为libpxx的依赖项，同样，如果库由于REQUIRED关键字而不可用，它将失败。注意IMPORTED\_TARGET关键字——如果没有它，就不会自动创建目标，我们必须用宏创建的变量手动定义它。
3. 我们通过打印PQXX\_确认一切都是正确的诊断消息。如果我们在前面的命令中没有指定REQUIRED，我们可以在这里检查是否设置了这个变量(也许是为了允许其他回退机制发挥作用)；
4. 我们创建主可执行文件；
5. 我们链接由pkg\_check\_ modules()创建的PkgConfig::PQXX IMPORTED目标。请注意，PkgConfig::是一个常量前缀，PQXX来自传递给该命令的第一个参数。

这是一个相当方便的方法，可以引入还不支持CMake的依赖项。

如果以上两个都没有怎么办?

### 编写自己的查找模块

虽然 CMake 几乎是一个行业标准，但仍然有很多库没有使用 CMake 进行管理，或者使用 CMake 进行管理但不导出 CMake 包。

如果它们可以安装在系统的默认位置，找到这些库通常不是问题，

但这并不总是可能或需要的。

一种常见的情况是，当使用仅特定项目需要的专有第三方库或使用不同版本的库来构建系统包管理器安装的库时。

查找模块是 CMake 的说明，用于指导如何为库查找必要的头文件和二进制文件，以及创建供 CMake 使用的导入目标。

在模块模式下调用 find\_package 时，CMake 在 CMAKE\_MODULE\_PATH 中搜索名为 Find<PackageName>.cmake 的文件。

* 查找头文件；
* 查找库文件；
* 设置目标Target；

查找包的流程：

通常，find模块按照以下顺序执行:

1. 它查找属于包的文件。
2. 它设置包含包的【包含】目录和【库目录】的变量。
3. 它为导入的包设置了目标Target。
4. 它为目标Target设置属性。

#### 例子1-OBSCURE库：

我们要做的第一件事是将cmake文件夹添加到CMAKE\_MODULE\_PATH，这是一个列表。所以，首先，我们添加以下行到CMakeLists.txt文件:

图形用户界面, 文本, 应用程序, 电子邮件

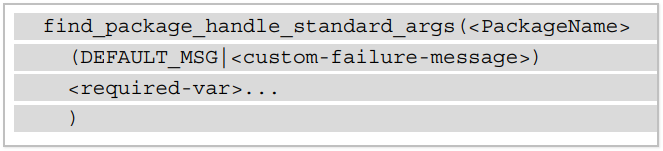
描述已自动生成

程序解析：

1.首先，使用 find\_library 命令搜索属于依赖项的实际库文件。如果找到，它的路径（包括实际文件名）将存储在 OBSCURE\_LIBRARY 变量中。命名 <PACKAGENAME>\_LIBRARY 变量是一种常见的做法。 NAMES 参数是库的可能名称列表。名称会自动使用通用前缀和扩展名进行扩展。因此，尽管在前面的示例中，我们查找名称“obscure”，但会找到一个名为 libobscure.so 或 obscure.dll 的文件。有关搜索顺序、提示和路径的更多详细信息将在本节后面介绍。

2.接下来，查找模块尝试定位包含路径。这是通过查找库的已知路径模式来完成的，通常是公共头文件之一。结果存储在 OBSCURE\_INCLUDE\_DIR 变量中。同样，通常的做法是将此变量命名为 <PACKAGENAME>\_INCLUDE\_DIR。

3.由于处理查找模块的所有要求可能很乏味并且通常非常重复，因此 CMake 提供了 FindPackageHandleStandardArgs 模块，它提供了一个方便的函数来处理所有常见情况。它提供了 find\_package\_handle\_standard\_args 函数，处理 REQUIRED、QUIET 和 find\_package 的版本相关参数。 find\_package\_handle\_standard\_args 有一个短签名和一个长签名。在示例中，使用短签名



4.对于大多数情况，find\_package\_handle\_standard\_args 的缩写形式就足够了。简而言之，find\_package\_handle\_standard\_args 函数将包名称作为第一个参数和包所需的变量列表。 DEFAULT\_MSG 参数告诉它在成功或失败时打印默认消息，这取决于 find\_package 是用 REQUIRED 还是 QUIET 调用的。该消息可以自定义，但我们建议您尽可能使用默认消息。这样，所有 find\_package 命令的消息都是一致的。在前面的示例中，find\_package\_handle\_standard\_args 检查传递的 OBSCURE\_LIBRARY 和 OBSCURE\_INCLUDE\_DIR 变量是否有效。如果是这种情况，则设置 <PACKAGENAME>\_FOUND 变量

5.如果一切顺利，查找模块将定义目标。在我们这样做之前，检查我们尝试创建的目标是否不存在是很有帮助的（以避免在我们多次调用相同依赖项的 find\_package 的情况下覆盖它）。创建目标是使用 add\_library 完成的。由于我们无法确定它是静态库还是动态库，因此类型为 UNKNOWN 并设置了 IMPORTED 标志。

最后，设置库的属性。我们推荐的最低设置是 MPORTED\_LOCATION 属性和包含文件在 INTERFACE\_INCLUDE\_DIR 中的位置。

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

#### 例子2-PQXX：

我们首先编写一个新的FindPQXX.cmake也就是利用上面讲的find\_package 模块方式，我们将把它存储在项目源代码树的Cmake /module目录中。

我们需要确保当find\_package()被调用时，find-模块被CMake发现，

所以我们将这个路径添加到CMakeLists.txt with list(APPEND)中的CMAKE\_MODULE\_PATH变量中。

FindPQXX.cmake

文本

描述已自动生成

要使用PQXX，主机还需要PostgreSQL；

函数详解：

1. 在 CMake 项目中找到并添加 PQXX 库的。
2. 首先，它定义了一个自定义函数 add\_imported\_library()，用于添加一个导入库。
3. 该函数先判断是否已经缓存了 PQXX 库的路径，如果已经缓存了，就直接调用 add\_imported\_library() 添加该库，并立即返回。
4. 如果没有缓存，那么它会尝试使用 find\_library() 和 find\_path() 函数来寻找 PQXX 库和头文件的路径，并且缓存这些路径。
5. 最后，在使用 find\_package\_handle\_standard\_args() 验证 PQXX 库是否可用后，
6. 如果 PQXX 库的路径是有效的，
7. 就调用 add\_imported\_library() 函数添加该库。
8. 如果 PQXX 库没有找到，该函数返回一个错误消息。

现在是时候检查 PQXX\_LIBRARY\_PATH 和 PQXX\_HEADER\_PATH 变量是否包含任何 -NOTFOUND 值。

同样，我们可以手动执行此操作，然后根据惯例打印诊断消息或终止构建执行，

或者我们可以使用 CMake 提供的 FindPackageHandleStandardArgs 列表文件中的 find\_package\_handle\_standard\_args() 辅助函数。

这是一个辅助命令，如果路径变量已填充，则将 <PKG\_NAME>\_FOUND 变量设置为 1，

并提供有关成功和失败的正确诊断消息（它将遵守 QUIET 关键字）。

如果在将 REQUIRED 关键字传递给查找模块时提供的路径变量之一为空，它也会以 FATAL\_ERROR 终止执行

CMakeLists.txt

图形用户界面, 文本, 应用程序

描述已自动生成

#### 涉及到的新命令详解：

涉及到的新的命令：

UNKNOWN 属性表示该导入库的类型（静态库或是动态库）无法被确定。

mark\_as\_advanced() 函数：将 **PQXX\_LIBRARIES** 和 **PQXX\_INCLUDES** 标记为高级选项，使它们在 CMake 的菜单中不会显示出来。

find\_dependency 是一个包含在 CMakeFindDependencyMacro 模块中的 CMake 宏。它被用于通过 **find\_package()** 命令查找一个依赖，并在找到该依赖时，将其缓存。

CMAKE\_PREFIX\_PATH 环境变量可以设置为目录列表，指定要由 find\_package()、find\_program()、find\_library()、find\_file() 和 find\_path() 命令搜索的安装前缀。每个命令都将添加其自己的文档中指定的适当子目录（如 bin、lib 或 include）。

find\_library用法：

图形用户界面, 文本, 应用程序

描述已自动生成

find\_library 自动将特定于平台的命名约定应用于文件名。在 Unix 平台上，名称将以 lib 为前缀，而在 Windows 上，将添加 .dll 或 .lib 扩展名

find\_path用法



find\_package\_handle\_standard\_args的用法

图形用户界面, 文本

描述已自动生成

如果以上都没有怎么办，我们自己编译；

### 使用 Git 存储库

许多项目依赖Git作为版本控制系统。假设我们的项目和外部库都在使用它，

是否有某种Git魔法可以让我们将这些存储库链接在一起?

我们能否构建一个特定的(或最新的)库版本作为构建项目的一步?如果有，怎么做?

#### 通过 Git 子模块提供外部库

一个可能的解决方案是使用一种内置在Git中的机制，称为Git子模块。

子模块允许项目存储库使用其他Git存储库，而无需实际将引用的文件添加到项目存储库。它们的工作原理类似于软链接

-它们指向一个特定的分支或在外部存储库中提交(但你需要显式地更新它们)

举例：

git init

git submodule add <https://github.com/jbeder/yaml-cpp.git>

注意，这里要先删除yaml-cpp文件夹；

图形用户界面, 应用程序

描述已自动生成

程序解读：

该代码使用 configure\_file() 函数将 config.yaml 文件复制到构建目录中。

然后，它尝试查找 yaml-cpp 包。如果找不到，它将使用 git 命令初始化子模块，

并添加 yaml-cpp 的子目录以构建该包。

最后，它使用 target\_link\_libraries() 函数将 yaml-cpp 库链接到 welcome 可执行文件中。

添加 yaml-cpp 存储库的子目录。 CMake 会将其视为项目的一部分并递归执行任何嵌套的 CMakeLists.txt 文件。

git submodule add <https://github.com/jbeder/yaml-cpp.git> 用法：

图形用户界面, 文本, 应用程序

描述已自动生成

git submodule update --init -- external/yaml-cpp的解读：

图形用户界面, 文本

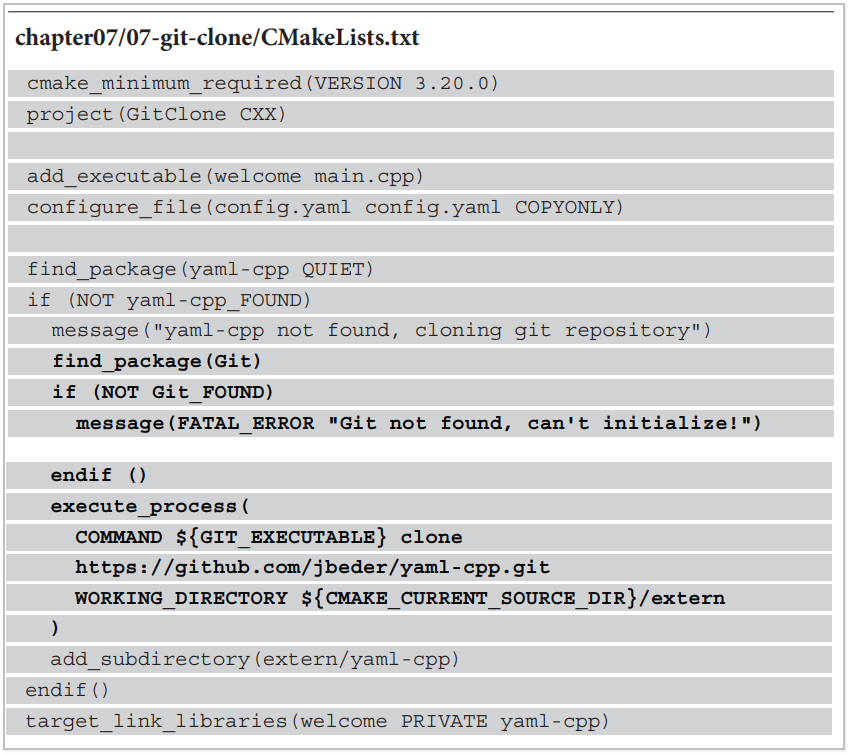
描述已自动生成

execute\_process的用法：

图形用户界面, 文本

描述已自动生成

#### 使用Git克隆项目依赖项



程序理解：

1. 我们首先通过FindGit查找模块检查Git是否可用。
2. 如果不是，我们就会陷入困境。我们将发出FATAL\_ERROR，并希望用户知道下一步要做什么。
3. 否则，我们将使用由find\_package()设置的GIT\_EXECUTABLE变量调用execute\_process()并克隆我们感兴趣的存储库。

如果不是GIT项目怎么办？

### 使用 ExternalProject 和 FetchContent 模块

在 FetchContent 之前，您将使用 Git 子模块手动下载依赖项，然后使用 add\_subdirectory 添加它们。

这在某些情况下有效，但维护起来可能相当不方便和麻烦

对于使用 CMake 生成的外部项目，使用 FetchContent 模块是添加源依赖项的首选方法。

对于二进制依赖项，使用find\_包和查找模块仍然是首选方法。

CMake 上的在线参考书将建议 ExternalProject 和 FetchContent 模块来处理更复杂的项目中的依赖关系管理。

ExternalProject 和 FetchContent 之间的主要区别之一是，FetchContent 在配置期间下载和配置外部项目，

而 ExternalProject 在构建步骤中执行所有操作。这样做的缺点是源代码及其配置在配置期间不可用。

#### 使用ExternalProject

CMake 3.0.0引入了一个名为ExternalProject的模块。正如您可以猜到的，它的目的是添加对在线存储库中可用的外部项目的支持。

ExternalProject模块在构建阶段填充依赖项。对于使用ExternalProject\_Add()添加的每个外部项目，CMake将执行以下步骤:

1. Mkdir -为外部项目创建一个子目录
2. 下载-从存储库或URL获取项目文件
3. 更新-在重新运行时刷新文件以获得支持增量更新的下载方法
4. 补丁-可选地执行一个补丁命令，根据项目的需要更改下载的文件
5. configure -对CMake项目执行configure阶段，对非CMake依赖项执行手动指定的命令
6. build -执行CMake项目的构建阶段，对于其他依赖项，执行make命令
7. install -安装CMake项目，对于其他依赖项，执行make install命令
8. test -如果TEST\_…定义选项

在构建阶段填充依赖项这一事实非常重要，它有两个效果 – 项目的命名空间是完全独立的，任何外部项目定义的目标在主项目中都不可见。后者特别痛苦，因为我们不能像使用 find\_package（） 命令后那样使用 target\_link\_libraries（）。这是因为两个配置阶段不相交。 主项目必须完成配置阶段并开始构建阶段，然后才能下载和配置依赖项。

可以使用多种方式下载源文件，这里依然使用的Git；

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

ExternalProject\_Add的用法：

图形用户界面, 文本, 应用程序

描述已自动生成

我们在这里需要谨慎，并理解这个示例之所以能够工作，

是因为yaml-cpp库在其CMakeLists.txt中定义了一个安装阶段。

这个阶段将库文件复制到系统中的标准位置。

target\_link\_libraries()命令的yaml-cpp参数被CMake解释为链接器- -lyaml-cpp的直接参数。

图形用户界面, 文本, 应用程序

描述已自动生成

文本

描述已自动生成

#### 使用FetchContent

现在，建议使用FetchContent模块来导入外部项目。这个模块在CMake 3.11版就已经提供了，但是我们建议至少使用3.14才能有效地使用它

本质上，FetchContent是ExternalProject的一个高级包装，提供类似的功能并且更加方便。

关键的区别在于执行的阶段——与ExternalProject不同，FetchContent在配置阶段填充依赖项，

将外部项目声明的所有目标带入主项目的范围内。这样，我们就可以像使用自己定义的目标一样使用它们。

FetchContent实现的是外部依赖的内部化，使得我们可以像编写自己的代码一样使用外部依赖。

FetchContent模块的使用需要三个步骤:

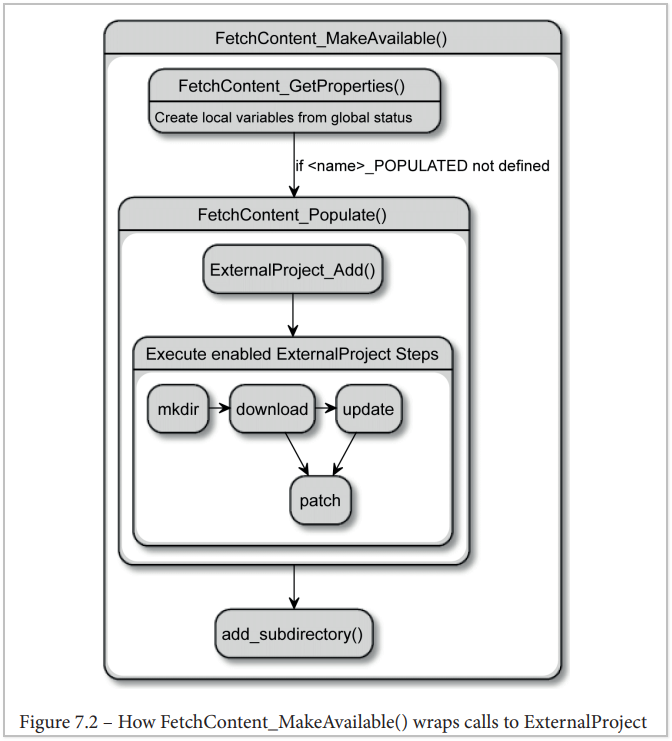
1. 使用Include (FetchModule)在项目中包含该模块。
2. 使用FetchContent\_Declare()命令配置依赖关系。
3. 使用FetchContent\_MakeAvailable()命令填充依赖项-下载，构建，安装，并将其列表文件添加到主项目和解析。

FetchContent 提供了用于拉取源依赖项（主要是FetchContent\_Declare）的函数列表，

它定义了用于下载和构建FetchContent\_MakeAvailable的参数，

这些参数填充依赖项的目标并使它们可用于构建。

FetchContent\_MakeAvailable的执行流程：



执行流程：

1.调用 FetchContent\_GetProperties() 将 FetchContent\_Declare() 设置的配置从全局变量读取到局部变量。

2.检查（不区分大小写）具有此名称的依赖项是否已填充以避免下载两次。如果是这样，请到此为止。

3.调用 FetchContent\_Populate()。它将通过转发我们设置的选项（但跳过禁用的选项）和下载依赖项来配置包装的 ExternalProject 模块。它还将设置一些变量以防止在后续调用中重新下载并将必要的路径转发到下一个命令

4.最后，add\_subdirectory() 以源代码和构建树作为参数调用，以告知父项目列表文件的位置以及构建工件的放置位置。

通过调用 add\_subdirectory()，CMake 有效地执行获取项目的配置阶段并检索在当前范围内定义的任何目标。多方便啊！

举例说明：

表格

低可信度描述已自动生成

目标存在；它是一个静态库，其源目录位于构建树中。使用相同的帮助程序调试 ExternalProject 示例中的目标只会返回：

在配置阶段无法识别目标。这就是为什么 FetchContent 更好并且应该尽可能使用的原因。

文本

描述已自动生成

#### FetchContent和ExternalProject的区别？

ExternalProject是旧的方法，它允许在CMake文件中定义外部项目，这些项目可以独立地构建，

并且在构建期间可以添加一些自定义命令和参数。然后在构建本地项目时，可以从外部项目构建的输出中导入文件，如头文件和库。

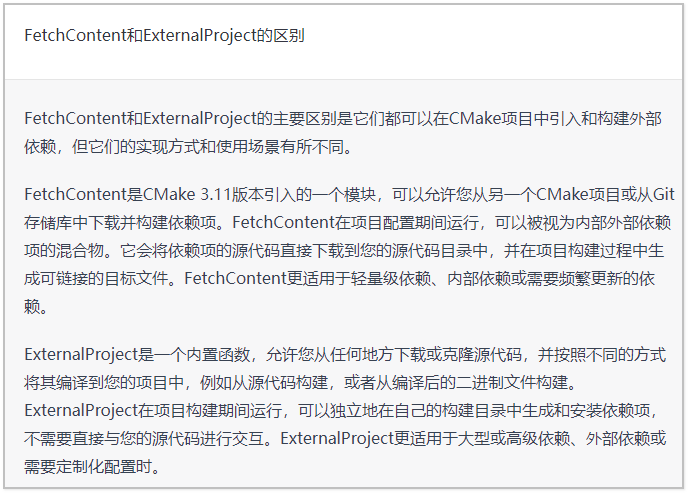
FetchContent是新的方法，它允许在CMake文件中定义和下载外部项目，可以在构建本地项目时直接编译这些项目。

与ExternalProject不同的是，FetchContent将所有外部项目的代码合并到您的项目中，

并使用add\_subdirectory添加，从而将外部项目作为本地项目的一部分来处理。

因此，两者的最大区别在于ExternalProject是独立的项目，而FetchContent将外部项目视为本地项目的一部分。

并且，FetchContent具有自己的一些优势，例如可以更轻松地管理依赖性、更简化的 CMakeLists.txt 和更简单的使用方式。



举例：

比如说，一个项目可能需要使用一个名为 "libfoo" 的外部库，通常情况下，如果使用 ExternalProject，

则必须先将 libfoo作为独立的项目进行编译，这代表着 libfoo与本地项目是两个独立的实体。

而使用 FetchContent 则相当于直接将 libfoo的源代码放入到本地项目中进行管理，

这样 libfoo和本地代码就处于同一个代码库中。

例如，当 cmake 构建时，

在本地项目中调用 libfoo 的函数和使用本地项目自己的代码一样，不需要任何额外链接和库文件配置。

总的来说，FetchContent 将外部库的源代码作为本地项目的一部分处理，

从而简化了管理和构建的过程。同时，也使得我们不再需要管理额外的外部项目。



思考：如果是非cmake的项目，如何下载，编译呢。

### 有没有更好的依赖解决方案？

使用包管理工具：

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

Conan 和 vcpkg 是处理 C++ 依赖项的两个比较流行的工具。它们都可以处理复杂的构建系统

图表, 条形图

描述已自动生成

比如：如果要用vcpkg安装一个Windows下的库，怎么处理，只需要一行命令就可以了；

vcpkg install ffmpeg:x64-windows

表格

描述已自动生成

更多的介绍可以查看官网资料；

Clin中使用vcpkg的方法：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

通过前面的解释，我们是不是已经搞清楚了vcpkg的原理，就是我们上面说的构建方法，

只是有些库不能直接构建，需要打补丁；

更进一步：

CLion在 2023.1 版本中添加了对 vcpkg 的支持；

图形用户界面, 网站

描述已自动生成

原文链接：https://blog.jetbrains.com/clion/2023/01/support-for-vcpkg-in-clion/

### 总结

依赖管理是一个很难涵盖的话题，有时可能很乏味。然而，花时间使用本章中描述的技术正确设置它是值得的。

CMake 的多功能性及其各种查找依赖项的方法是它最大的优势，但也是它最大的弱点。

通过使用各种 find\_ 命令、FetchContent、ExternalProject，或将任何可用的包管理器与 CMake 集成，

几乎任何依赖项都可以集成到项目中。然而，有这么多方法可供选择，要找到最好的方法可能很困难。

尽管如此，我们还是建议尽可能使用 find\_package。 CMake 越流行，其他项目无缝集成的可能性就越大。

# 参考文献：

## 《Modern CMake For C++》

## 《CMake Best Practices (2022) 》

## CMake Integration（微软写的）

https://learn.microsoft.com/en-us/vcpkg/users/buildsystems/cmake-integration

## C++ Ecosystem in 2022

<https://blog.jetbrains.com/clion/2023/01/cpp-ecosystem-in-2022/>

## Support for vcpkg in CLion

https://blog.jetbrains.com/clion/2023/01/support-for-vcpkg-in-clion/