在项目中使用第三方库

随着计算机技术的发展，现在基本不可能从头开始编写一个复杂的项目，它们多少都需要依赖其他的库。然而，C++ 社区并没有像 Go、Rust 等语言提供了统一的包管理系统，每个项目都有自己偏好的第三方依赖管理方式。因此，如何管理以及使用第三方 C++ 依赖是一件十分复杂的事情。我们需要根据第三方项目提供的配置文件和自己的喜好，来确定合适的引用方式。

为什么大家都觉得 C++ 编程麻烦，其中也有一点是因为 C++ 不像 Python 等编程语言有标准以及完善的包管理器，可以快速地在项目中引入依赖。C++ 中使用其他人的依赖多多少少需要掌握程序的编译链接知识，而且需要自己手动解决传递依赖问题。如果能方便地引用其他人的代码以及使用库，C++ 编程会简单很多。

依赖管理方式概览

总的来说，第三方的依赖管理有几种方式，有的方便开发者开箱即用，但是会占用更多空间，有的需要开发者自行处理依赖，好处是多个项目可以共享依赖节省空间。

项目自行管理依赖

项目自行管理依赖的意思是，项目需要什么依赖，都在项目自己包含依赖的源代码等，做到其他开发者克隆代码后仅需要安装基本的构建工具链即可构建。这种模式也称为 vendor 模式。

1. 直接将源代码拷贝到项目目录下，一般这个目录会叫 `third_party`。例如，我们需要依赖 Google Test，就将它的源代码放在 `third_party/googletest-1.0` 下。
2. 将第三方项目作为 Git Submodule 保存到项目目录下。这种方式和第一种方式类似，同样也是把源代码保存在了项目目录里，不同的是使用 Git Submodule 的话可以更方便地管理版本。例如，我们需要依赖 Google Test，就使用 `git submodule add https://github.com/google/googletest third_party/googletest` 命令，将依赖保存到 `third_party` 目录下。需要注意的是，如果使用了 submodule，那么需要提醒其他开发者记得初始化 submodule。例如使用 `git clone --recurse-submodules` 克隆项目。

> **▼ Git Subtree**
>
> Git Submodule 会引入 `.gitmodule` 文件，在开发者克隆仓库的时候也需要额外的选项或者操作才能将子项目完整下载下来，而且 Git Submodule 需要的 Git 版本比较高。
>
> 作为替代，可以使用 Git Subtree 指令。严格来说它其实是一个 Git 脚本，而不是 Git 的原生功能，它不创建额外的元信息文件，而是将元信息保存到 Commit 记录中。使用 Git Subtree 的好处和直接将源代码拷贝到项目目录里一样，其他开发者开箱即用，意识不到 Submodule 的存在。缺点就是比较慢，而且需要注意不要在同一个 Commit 中提交子项目和父项目的代码修改。考虑到依赖很少修改，这些代价也可以接受。

开发者管理依赖

仅在文档中说明需要什么库，让开发者自行安装，并提供编译选项让开发者指定路径。基本上使用 Autotools 的都是这种模式，比如 `./configure` 脚本默认会去查找项目所需的库，找不到就直接报错。开发者要么将库安装到系统目录里，要么在查找路径里添加库的目录，要么在 `./configure` 时使用 `--with-***` 选项来指定项目所需库的路径。为了方便开发，这种方式一般会在文档中提供依赖安装的方式，让开发者手动安装。

- 使用系统包管理器。例如，我们可以在项目文档中说明需要安装 `libgoogletest-dev` 这个包。这种方式会污染全局环境，一般不建议使用。
- 仅说明依赖版本，让开发者自行下载依赖编译安装。这种方式比较麻烦，如果其他依赖也有依赖的话，需要人手工解决。

- 使用第三方包管理器。例如 vcpkg 、 conan 、 pkg-config 等，这种既不会污染全局环境，也能自动处理依赖关系，缺点是可能包并不是很全或者最新。

混合管理

当然，由于 C++ 项目的复杂，一般会出现多种管理方式混用的情况。比如说，像 boost 这种庞大而常用的库，项目就没必要将它的代码也放到自己仓库中，让开发者自己安装即可。而有些系统库，例如 libcg ，则最好是使用包管理器安装适合系统的版本。

示例

下面提供几种不同引用项目依赖的方式，来让你体验一下 C++ 中包管理的复杂和多样。

.cmake 文件

如果你是用系统自带的包管理器安装的开发库，那么这个库有可能已经提供了 .cmake 文件，里面内置了使用这个库所需要的命令。这时候可以简单地使用 find_package 来让 CMake 帮我们加载配置文件。我们以 fmt 库为例：

```
$ sudo apt install libfmt-dev
```

CMakeLists.txt

```
find_package(fmt)
target_link_libraries(my-server fmt)
```

CMakeLists.txt

有一些开源项目可能提供了 CMakeLists.txt ，或者我们想引用我们自己或者同事、同学写的 CMake 项目，这时候可以用 add_subdirectory 来添加依赖，根据第三方依赖不同的放置位置，引用的方式也不太一样。

依赖是项目的子文件夹

假如我们的项目位于 my-server 文件夹，如果第三方依赖放在了 my-server/third_party ，是项目的子文件夹，那么我们直接使用 add_subdirectory ，然后使用对应的 target 名称即可。例如，假如我们使用了一个叫 cpr 的 CMake 项目，放在了 my-server/third_party/cpr 下，我们可以这么引用：

CMakeLists.txt

```
add_subdirectory("${CMAKE_CURRENT_SOURCE_DIR}/third_party/cpr")
target_link_libraries(my-server cpr)
```

依赖和项目放在同一个层级

假如我们编写了几个项目 my-server 、 another-server ，如果都用到了同一个依赖，那么在每个项目里都放一份代码可能会浪费空间，而且更新依赖也需要分别更新，这时候我们希望将第三方依赖放到和他们同一个级别的 third_party 目录中，还是以 cpr 项目为例，假如我们把依赖放在了 third_party/cpr 下，那么在 my-server 中，我们需要这么写：

CMakeLists.txt

```
add_subdirectory("${CMAKE_CURRENT_LIST_DIR}/../third_party/cpr" cpr.build)
target_link_libraries(my-server cpr)
```

▼ **Try it**

尝试不指定 add_subdirectory 的第二个参数，看看 CMake 的报错。

和前面的差不多，我们同样通过 add_subdirectory 指定项目的路径，但是，由于现在包含的路径不是项目的子文件夹了，需要我们手动指定依赖编译存放的路径，也就是 add_subdirectory 的第二个参数。

构建时再下载

如果不想将代码下载到自己的代码库中，可以使用 FetchContent 在构建的时候再下载，例如：

CMakeLists.txt

```
include(FetchContent)
FetchContent_Declare(cpr GIT_REPOSITORY https://github.com/whoshuu/cpr.git GIT_TAG c8d33915dbd88ad6c92b258869b
03aba06587ff9)
FetchContent_MakeAvailable(cpr)

target_link_libraries(my-server cpr)
```

pkg-config

有一些库提供了 pkg-config 文件，后缀名是 .pc，我们可以使用 pkg-config 命令来在编译的时候输出合适的编译参数。例如：

```
$ sudo apt install libmysqlclient-dev
$ pkg-config --libs --cflags mysqlclient
-I/usr/include/mysql -lmysqlclient
$ g++ main.cpp `pkg-config --libs --cflags mysqlclient`
```

在 CMake 中，可以通过 PkgConfig 这个包及其提供的 pkg_check_modules 命令方便地引用已经存在的 pkg-config 文件。以 mysqlclient 这个库为例，它没有提供 .cmake 文件，也没有 CMakeLists.txt，但是提供了 pkg-config 文件：

CMakeLists.txt

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(MySQL REQUIRED IMPORTED_TARGET mysqlclient>=21.0)
target_link_libraries(my-server PkgConfig::MySQL)
```

其中 pkg_check_modules 第一个参数为引入后的变量前缀，这个例子里，会引入 MySQL_LIBRARIES 等变量。为了方便使用，还可以通过 IMPORTED_TARGET 来提供 PkgConfig::MySQL 这样的引入形式，最后便是指定包名以及包版本号。

vcpkg

vcpkg 是微软出品的一款跨平台 C++ 包管理软件，主要服务 CMake 用户。vcpkg 和系统包管理系统类似，直接使用 vcpkg install *** 就可以全自动地安装一个第三方库。不同的是，vcpkg 社区会维护项目的 CMake 文件，这样使用 vcpkg 安装的包都只需要在 CMakeLists.txt 中使用 find_package 指令即可轻松使用，同时 vcpkg 也不会将库安装到系统中，在使用 CMake 的时候需要先导入 vcpkg 提供的 CMake 文件。

使用 vcpkg 首先需要从源码编译 vcpkg：

```
$ git clone https://github.com/Microsoft/vcpkg.git && cd vcpkg
$ ./vcpkg/bootstrap-vcpkg.sh
$ ./vcpkg integrate install
Applied user-wide integration for this vcpkg root.
```

```
CMake projects should use: "-DCMAKE_TOOLCHAIN_FILE=/home/howard/vcpkg/scripts/buildsystems/vcpkg.cmake"
```

然后，就可以使用 vcpkg install 指令安装第三方包：

```
$ ./vcpkg install fmt
Computing installation plan...
The following packages will be built and installed:
    fmt[core]:x64-linux -> 7.1.3#5
  * vcpkg-cmake[core]:x64-linux -> 2021-06-25#5
  * vcpkg-cmake-config[core]:x64-linux -> 2021-05-22#1
# ...
Total elapsed time: 12.03 s

The package fmt provides CMake targets:

    find_package(fmt CONFIG REQUIRED)
    target_link_libraries(main PRIVATE fmt::fmt)

    # Or use the header-only version
    find_package(fmt CONFIG REQUIRED)
    target_link_libraries(main PRIVATE fmt::fmt-header-only)
```

这样，我们只需要按照指示，在 CMakeLists.txt 中添加 find_package 指令，然后在使用 CMake 配置项目的时候，指定 -DCMAKE_TOOLCHAIN_FILE 选项，就可以使用 vcpkg 提供的包了。

CMakeLists.txt

```
find_package(fmt CONFIG REQUIRED)
target_link_libraries(my-server fmt)


$ mkdir build && cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=/home/howard/vcpkg/scripts/buildsystems/vcpkg.cmake
```

最后更新: 2021-09-08 22:29:01