引例

```
# REQUIRED 说明必须找到，否则停止
# 默认采用module模式，如果没有找到相应的配置文件就使用config模式
find_package(OpenCV 4 REQUIRED)

# 当加载了配置文件后，一般都会定义两个变量 OpenCV_INCLUDE_DIRS 和
OpenCV_LIBRARIES
include_directories(${OpenCV_INCLUDE_DIRS}) # 即 OpenCV_INCLUDE_DIRS
是头文件所在的路径，注意s复数

add_exexcutable(main src/main.cpp)

target_link_libraries(main ${OpenCV_LIBRARIES}) # 即 OpenCV_LIBRARIES
是库文件所在的路径，s：多个
```

作用

项目需要使用一（多）个已经构建好的package。这个package可能是：

1. 由cmake构建的
2. 别的build system构建好的
3. 一些无需编译的文件的集合

cmake通过提供 `find_package` 这个命令来处理上述情况，它会搜寻默认的路径和一些项目或者用户指定的路径。

以下两种搜索方法，默认是使用module，找不到文件之后会自动进入config模式

搜索方法1--config mode[1]

> *相对来说更可靠的方法，因为软件包的信息总是和软件包本身保持同步*

config files 通常都能在类似于 XXX/lib/cmake/<PackageName> 目录下找到

cmake配置文件的名称并没有固定的要求。但是如果有些其他奇怪名字的.cmake文件，应该都会已经在<PackageName>config.cmake中通过 `include()` 命令将它们的配置也载入好，所以用户一般只需要 `find_package()` 即可

配置文件搜索过程[1.2]

配置文件的明细必须是 <PackageName>Config.cmake 或者 <LowerCasePackageName>-config.cmake。后面可能还有version，比如<PackageName>ConfigVersion.cmake。这一配置文件是CMake进入包的 entry point。

1. 默认的系统包管理目录 CMAKE_SYSTEM_PREFIX_PATH：`/usr` `/usr/local`
2. cmkae包管理目录 CMAKE_PREFIX_PATH。举例某配置文件的路径是：`/opt/somepackage/lib/cmake/somePackage/SomePackageConfig.cmake`。那么我就需要 `list(append CMAKE_PREFIX_PATH /opt/somepackage)`。它会自动寻找 `/opt/somepackage/` 内子目录 `lib/cmake/`
    a. 上面所描述的情况是为什么呢?
    b. 见参考资料[3]：CMAKE_PREFIX_PATH is Semicolon-separated list of directories specifying installation prefixes to be searched by the find_package(), find_program(), find_library(), find_file(), and find_path() commands。简单来说就是CMAKE_PREFIX_PATH就是由一

些由`;`隔开的安装路径前缀组成的列表。那么这个安装路径前缀又是什么呢?

c. 那就得回到make install这个命令。一般我们想指定安装位置时，会`set(CMAKE_INSTALL_PREFIX XXX)`。运行完 make install 这个命令之后，XXX这个目录就会变成类似于这样:

```
XXX
├── bin/
│   ├── my_application
│   └── my_other_application
├── lib/
│   ├── libmy_library.so
│   ├── libmy_library.a
│   ├── cmake/
│   │   ├── my_library/
│   │   │   ├── my_libraryConfig.cmake
│   │   │   ├── my_libraryConfigVersion.cmake
│   │   │   ├── my_libraryTargets.cmake
│   │   │   └── my_libraryTargets-noconfig.cmake
│   │   └── MyProjectConfig.cmake
│   └── pkgconfig/
│       └── my_library.pc
├── include/
```

d. 可以看到所有的配置文件都会自动生成在XXX的子目录`lib/cmake`下。因此CMAKE_PREFIX_PATH也是添加XXX即可，不需要具体指定到包含`.cmake`的目录`XXX/lib/cmake/`

3. 设置cmake变量 <PackageName>_DIR。注意此时这个路径不是一个prefix。而是要具体指向一个包含配置文件的目录,比如说`set(SomePackage_DIR /opt/seompackage/lib/cmake/SomePackage)`。注意它和上面两个prefix是不一样的!!!。注意这里的<PackageName>_DIR 是cmake变量，和下面同名的cmake环境变量区分开❗❗❗

搜索方法2--module mode

> *并非所有的软件包都支持cmake，这时候就需要单独提供<findPackageName>.cmake文件。由于这个文件通常和软件包独立维护，所以不一定可靠。*

如果包没有提供config-style file，可以通过寻找 FindSomePackage.cmake 文件来配置包。相比于config模式，module模式：

1. find module file 不是由包本身提供的，因此即使找到了文件也不代表能够正确的配置包
2. 类似于config模式的CMAKE_PREFIX_PATH，cmake在CMAKE_MODULE_PATH中找寻find module file。注意这不是PREFIX，所以要指定到直接包含 Find<PackageName>.cmake 的目录。
3. cmake可能跳过某些第三方库提供的 Find<PackageName>.cmake 文件。因为有些软件包的Find<PackageName>.cmake 文件是由 CMake 社区维护的，而不是由每个软件包的开发者提供的。很容易出现第三方软件包的更新速度比 Find 模块的更新速度更快，所以对于CMake社区来说负担很重，所以不提供标准的find module。

查询路径

Q: find_package 是去哪里找第三方包的头文件目录和库文件的?

A: 所有查询的规则都写在了 `.cmake` 文件中，因此我们要做的就是找到 `.cmake` 文件在哪。并且一般来说有两种命名的方式： 1. `Find<package>.cmake` 2. `<package>Config.cmake`

`.cmake` 文件所在的目录一般：

1. 通过编译安装第三方包时，会自动安装到 `$CMAKE_PREFIX_PATH/lib/cmake` 目录下，其中 `CMAKE_PREFIX_PATH` 就是你编译安装时指定的路径。如何指定安装路径，可以查询这篇笔记的 `CMAKE_INSTALL_PREFIX` 部分。
2. 系统默认库 `/usr/lib/cmake` `/usr/local/lib/cmake`

find_package 用法

```
# 基础用法
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
             [REQUIRED] [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [REGISTRY_VIEW  (64|32|64_32|32_64|HOST|TARGET|BOTH)]
             [GLOBAL]
             [NO_POLICY_SCOPE]
             [BYPASS_PROVIDER])
             # 以下是高级用法，后续我们会讲的是NO_*参数，可以稍微看一眼
             [NAMES name1 [name2 ...]]
             [CONFIGS config1 [config2 ...]]
             [HINTS path1 [path2 ... ]]
             [PATHS path1 [path2 ... ]]
             [REGISTRY_VIEW  (64|32|64_32|32_64|HOST|TARGET|BOTH)]
             [PATH_SUFFIXES suffix1 [suffix2 ...]]
             [NO_DEFAULT_PATH]
             [NO_PACKAGE_ROOT_PATH]
             [NO_CMAKE_PATH]
             [NO_CMAKE_ENVIRONMENT_PATH]
             [NO_SYSTEM_ENVIRONMENT_PATH]
             [NO_CMAKE_PACKAGE_REGISTRY]
             [NO_CMAKE_BUILDS_PATH] # Deprecated; does nothing.
             [NO_CMAKE_SYSTEM_PATH]
             [NO_CMAKE_INSTALL_PREFIX]
             [NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]
             [CMAKE_FIND_ROOT_PATH_BOTH |
              ONLY_CMAKE_FIND_ROOT_PATH |
              NO_CMAKE_FIND_ROOT_PATH])
# verision 版本合适就好，即大版本号相同
# EXACT 版本必须一致 如：4.1.20
# QUEIET 没找到包就算了
# MODULE 指定后就不会fallback到config模式了
# REQUIRED 一定要找到包，否则fatal error
# 其他关键字，自己看官方文档吧，基础掌握这些已经够用了

# 例
set(protobuf_DIR
"/home/<user_name>/temp_work/install_aarch64/grpc/lib/cmake/protobuf"
)
```

```
find_package(protobuf REQUIRED)
```

Config模式下搜索配置文件过程[2]

CMake 3.24 新功能： 两种模式下都会先在 `CMAKE_FIND_PACKAGE_REDIRECTS_DIR` 目录搜索config file。如果没有找到，就按照如下的路径继续寻找（只考虑Unix）：

前置补充：此表格说明了当指定了路径作为prefix之后，其会递归寻找的子目录

ENTRY	PLATFORM
`<prefix>/(lib/<arch>|lib*|share)/cmake/<name>*/`	U
`<prefix>/(lib/<arch>|lib*|share)/<name>*/`	U
`<prefix>/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/`	U
`<prefix>/<name>*/(lib/<arch>|lib*|share)/cmake/<name>*/`	U
`<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/`	U
`<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/`	U

上面路径补充解读：

1. (lib/<arch>|lib*|share)：表示三个可能的选项：
    a. lib/<arch>: 表示库文件安装在 `<prefix>/lib/<arch>` 目录下，其中 `<arch>` 为体系结构名称，例如 x86、x64 等
    b. lib*: 表示库文件安装在 `<prefix>` 目录下的任意名为 `lib*` 的目录中，例如 `lib`、`lib64` 等。
    c. share：表示库文件安装在 `<prefix>/share` 目录下。
2. <name>*: 表示name开头的任意名称
3. (cmake|CMake)：两个可能的选项：cmake和CMake

<prefix> 即 installation prefix 按照如下的顺序搜寻（如果设定了 `NO_DEFAULT_PATH` 那么所有的 `NO_*` 都会enable）：

1. 软件包独有的安装路径前缀（此过程会跳过，如果传入了 `NO_PACKAGE_ROOT_PATH` or `set(CMAKE_FIND_USE_PACKAGE_ROOT_PATH FALSE)`：

    a. cmake 变量: <PackageName>_ROOT
    b. cmake 变量: <PACKAGENAME>_ROOT
    c. 环境变量: <PackageName>_ROOT
    d. 环境变量: <PACKAGENAME>_ROOT
2. cmake变量<VAR>（可以使用-DVAR=value 来指定, value是用分号分隔的list。但是还是在CMkaeLists.txt里面set变量值比较清晰明了）（此过程会跳过，如果传入了 `NO_CMAKE_PATH` or `set(CMAKE_FIND_USE_CMAKE_PATH FALSE)`）:
    a. CMAKE_PREFIX_PATH
    b. CMAKE_FRAMEWORK_PATH
    c. CMAKE_APPBUNDLE_PATH
3. cmake环境变量（此过程会跳过，如果传入了 `NO_CMAKE_ENVIRONMENT_PATH` or `set(CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH FALSE)`）：
    a. `<PackageName>_DIR` 这里设定的<PackageName>_DIR是cmake环境变量，搜索的过程其被当作prefix，所以在Unix系统下，若它直接包含了config-style file，根据上面表格的表格，是找不到的!! !!
    b. CMAKE_PREFIX_PATH
    c. CMAKE_FRAMEWORK_PATH
    d. CMAKE_APPBUNDLE_PATH
4. `HINT` 选项指定的路径
5. 系统环境变量（此过程会跳过，如果传入了 `NO_SYSTEM_ENVIRONMENT_PATH` or `set(CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH FALSE)`）：
    a. PATH
6. Search paths stored in the CMake User Package Registry.(没用过，不了解，日后再说)（此过程会跳过，如果传入了 `NO_CMAKE_PACKAGE_REGISTRY` or `set(CMAKE_FIND_USE_PACKAGE_REGISTRY FALSE)`）
7. 当前系统的平台文件(platform files)定义的cmake变量（跳过 `CMAKE_INSTALL_PREFIX` and `CMAKE_STAGING_PREFIX` 如果传入了 `NO_CMAKE_INSTALL_PREFIX`）（此过程会跳过，如果传入了 `NO_CMAKE_SYSTEM_PATH` or `set(CMAKE_FIND_USE_CMAKE_SYSTEM_PATH FALSE)`）：
    a. CMAKE_SYSTEM_PREFIX_PATH
    b. CMAKE_SYSTEM_FRAMEWORK_PATH
    c. CMAKE_SYSTEM_APPBUNDLE_PATH
8. Search paths stored in the CMake System Package Registry.（和6类似，不说了）
9. `PATHS` 选项指定的路径

有一个问题是之前说的是<PackageName>_DIR要直接包含 config-style file，但是这边条目3又把它当成prefix来使用，需要明确一下到底以哪个为准，或者哪里理解有偏差

使用方法

```
# 方法1
list(APPEND CMAKE_MODULE_PATH "
<prefix_path>/lib/cmake/<third_party>") # 此路径下包含了 .cmake 文件
## 设定完后，module模式会优先查找此路径

# 方法2
set(OpenCV_DIR "<prefix_path>/lib/cmake/<third_party>")

# 而后
find_package(OpenCV)
```

链接库问题

`find_package` 之后，自动生成 了头文件所在的目录 `<package_name>_INCLUDE_DIRS` 和库文件路径`<package_name>_LIBRARIES`

此时 `include_directories` 一般是不会出现问题的，但是 `target_link_libraries` 可能会出现问题。最后报错显示 ld找不到包。

这是因为，部分第三方库 `.cmake` 提供给 `cmake` 的 `${<package_name>_LIBRARIES}` 只是一个库的名字，而不是库文件的绝对路径。关于如何解决链接库的问题，可以查看这篇笔记

参考资料

1. cmake.org--use dependencies guide
2. cmake--find_package
3. cmake.org--CMAKE_PREFIX_PATH