Cmake 实践

Cmake Practice --Cjacker

前言：

cmake 已经开发了 5,6 年的时间，如果没有 KDE4，也许不会有人或者 Linux 发行版本重视 cmake，因为除了 Kitware 似乎没有人使用它。通过 KDE4 的选型和开发，cmake 逐渐进入了人们的视线，在实际的使用过程中，cmake 的优势也逐渐的被大家所认识，至少 KDE 的开发者们给予了 cmake 极高的评价，同时庞大的 KDE 项目使用 cmake 来作为构建工具也证明了 cmake 的可用性和大项目管理能力。

所以，cmake 应该感谢 KDE，也正因为如此，cmake 的开发者投入了 KDE 从 autotools 到 cmake 的迁移过程中，并相当快速和顺利的完成了迁移，现在整个 KDE4 开发版本全部使用 cmake 构建。

这也是促使我们学习 cmake 的原因，首先 cmake 被接受并成功应用，其次，cmake 的优势在实际使用中不断的体现出来。

我们为什么不来认识一下这款优秀的工程构建工具呢？

在 2006 年 KDE 大会，听 cmake 开发者当面介绍了 cmake 之后，我就开始关注 cmake，并将 cmake 纳入了 Everest 发行版，作为系统默认组件。最近 QT-4.3 也正式进入了 Everest 系统，为 KDE4 构建完成了准备工作。

但是，在学习 cmake 的过程中，发现官方的文档非常的少，而且错误也较多，比如：

在介绍 Find<Name>模块编写的文档中，模块名称为 FOO，但是后面却出现了 Foo_FIND_QUIETLY 的定义，这显然是错误的，这样的定义永远不可能有效，正确的定义是 FOO_FIND_QUIETLY。种种原因，促使我开始写一份“面向使用和实用”的 cmake 文档，也就是本教程《cmake 实践》(Cmake Practice)

本文档是边学习边编写的成果，更像是一个学习笔记和 Tutorial，因此难免有失误或者理解不够透彻的地方，比如，我仍然不能理解为什么绝大部分使用变量的情况要通过${}引用，而在 IF 语句中却必须直接使用变量名。也希望能够有 cmake 的高手来指点迷津。

补：从 cmake 的 maillist,我找到了一些答案，原文是：

The `IF(var)' or `IF(NOT var)' command expects `var' to be the name of a variable. This is stated in CMake's manual. So, for your situation `IF(${libX})' is the same as `IF(/usr/lib/xorg)' and then CMake will check the value of the variable named `/usr/lib/xorg'.也就是说 IF 需要的是变量名而不是变量值

这个文档是开放的，开放的目的是为了让更多的人能够读到并且能够修改，任何人都可以对它作出修改和补充，但是，为了大家都能够获得你关于 cmake 的经验和积累，如果你现错误或者添加了新内容后，请务必 CC 给我一份，让我们共同把 cmake 掌握的更好。

一，初识 cmake

Cmake 不再使你在构建项目时郁闷地想自杀了.

--一位 KDE 开发者

1，背景知识：

cmake 是 kitware 公司以及一些开源开发者在开发几个工具套件(VTK)的过程中衍生品，最终形成体系，成为一个独立的开放源代码项目。项目的诞生时间是 2001 年。其官方网站是 www.cmake.org，可以通过访问官方网站获得更多关于 cmake 的信息。cmake 的流行其实要归功于 KDE4 的开发(似乎跟当年的 svn 一样，KDE 将代码仓库从 CVS 迁移到 SVN，同时证明了 SVN 管理大型项目的可用性)，在 KDE 开发者使用了近 10 年 autotools 之后，他们终于决定为 KDE4 选择一个新的工程构建工具，其根本原因用 KDE 开发者的话来说就是：只有少数几个“编译专家”能够掌握 KDE 现在的构建体系(admin/Makefile.common)，在经历了 unsermake， scons 以及 cmake 的选型和尝试之后，KDE4 决定使用 cmake 作为自己的构建系统。在迁移过程中，进展异常的顺利，并获得了 cmake 开发者的支持。所以，目前的 KDE4 开发版本已经完全使用 cmake 来进行构建。像 kdesvn,rosegarden 等项目也开始使用 cmake，这也注定了 cmake 必然会成为一个主流的构建体系。

2，特点：

cmake 的特点主要有：

1，开放源代码，使用类 BSD 许可发布。http://cmake.org/HTML/Copyright.html

2，跨平台，并可生成 native 编译配置文件，在 Linux/Unix 平台，生成 makefile，在苹果平台，可以生成 xcode，在 Windows 平台，可以生成 MSVC 的工程文件。

3，能够管理大型项目，KDE4 就是最好的证明。

4，简化编译构建过程和编译过程。Cmake 的工具链非常简单：cmake+make。

5，高效虑，按照 KDE 官方说法，CMake 构建 KDE4 的 kdelibs 要比使用 autotools 来构建 KDE3.5.6 的 kdelibs 快 40%，主要是因为 Cmake 在工具链中没有 libtool。

6，可扩展，可以为 cmake 编写特定功能的模块，扩充 cmake 功能。

3，问题，难道就没有问题?

1，cmake 很简单，但绝对没有听起来或者想象中那么简单。

2，cmake 编写的过程实际上是编程的过程，跟以前使用 autotools 一样，不过你需要编写的是 CMakeLists.txt(每个目录一个)，使用的是”cmake 语言和语法”。

3，cmake 跟已有体系的配合并不是特别理想，比如 pkgconfig，您在实际使用中会有所体会，虽然有一些扩展可以使用，但并不理想。

4，个人的建议：

1，如果你没有实际的项目需求，那么看到这里就可以停下来了，因为 cmake 的学习过程就是实践过程，没有实践，读的再多几天后也会忘记。

2，如果你的工程只有几个文件，直接编写 Makefile 是最好的选择。

3，如果使用的是 C/C++/Java 之外的语言，请不要使用 cmake(至少目前是这样)

4，如果你使用的语言有非常完备的构建体系，比如 java 的 ant，也不需要学习 cmake，

虽然有成功的例子，比如 QT4.3 的 csharp 绑定 qyoto。

5，如果项目已经采用了非常完备的工程管理工具，并且不存在维护问题，没有必要迁移到 cmake

4，如果仅仅使用 qt 编程，没有必要使用 cmake，因为 qmake 管理 Qt 工程的专业性和自动化程度比 cmake 要高很多。

二，安装 cmake

还需要安装吗?

cmake 目前已经成为各大 Linux 发行版提供的组件，比如 Everest 直接在系统中包含，Fedora 在 extra 仓库中提供，所以，需要自己动手安装的可能性很小。如果你使用的操作系统(比如 Windows 或者某些 Linux 版本)没有提供 cmake 或者包含的版本较旧，建议你直接从 cmake 官方网站下载安装。

http://www.cmake.org/HTML/Download.html

在这个页面，提供了源代码的下载以及针对各种不同操作系统的二进制下载，可以选择适合自己操作系统的版本下载安装。因为各个系统的安装方式和包管理格式有所不同，在此就不再赘述了，相信一定能够顺利安装 cmake。

三，初试 cmake – cmake 的 helloworld

Hello world，世界 你好

本节选择了一个最简单的例子 Helloworld 来演练一下 cmake 的完整构建过程，本节并不会深入的探讨 cmake，仅仅展示一个简单的例子，并加以粗略的解释。我们选择了 Everest Linux 作为基本开发平台，因为这个只有一张 CD 的发行版本，包含了 gcc-4.2/gtk/qt3/qt4 等完整的开发环境，同时，系统默认集成了 cmake 最新版本 2.4.6。

1，准备工作：

首先，在/backup 目录建立一个 cmake 目录，用来放置我们学习过程中的所有练习。

```
mkdir -p /backup/cmake
```

以后我们所有的 cmake 练习都会放在/backup/cmake 的子目录下(你也可以自行安排目录，这个并不是限制，仅仅是为了叙述的方便)

然后在 cmake 建立第一个练习目录 t1

```
cd /backup/cmake
mkdir t1
cd t1
```

在 t1 目录建立 main.c 和 CMakeLists.txt(注意文件名大小写)：

main.c 文件内容：

```
//main.c
#include <stdio.h>
int main()
{
     printf(“Hello World from t1 Main!\n”);
     return 0;
}
```

CmakeLists.txt 文件内容：

```
PROJECT (HELLO)
SET(SRC_LIST main.c)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir "${HELLO_SOURCE_DIR})
ADD_EXECUTABLE(hello SRC_LIST)
```

2，开始构建

所有的文件创建完成后，t1 目录中应该存在 main.c 和 CMakeLists.txt 两个文件

接下来我们来构建这个工程，在这个目录运行：

cmake . (注意命令后面的点号，代表本目录)。

输出大概是这个样子：

```
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- This is BINARY dir /backup/cmake/t1
-- This is SOURCE dir /backup/cmake/t1
-- Configuring done
-- Generating done
-- Build files have been written to: /backup/cmake/t1
```

再让我们看一下目录中的内容：

你会发现，系统自动生成了：

CMakeFiles, CMakeCache.txt, cmake_install.cmake 等文件，并且生成了 Makefile.

现在不需要理会这些文件的作用，以后你也可以不去理会。最关键的是，它自动生成了 Makefile.

然后进行工程的实际构建，在这个目录输入 make 命令，大概会得到如下的彩色输出：

```
Scanning dependencies of target hello
[100%] Building C object CMakeFiles/hello.dir/main.o
Linking C executable hello
[100%] Built target hello
```

如果你需要看到 make 构建的详细过程，可以使用 make VERBOSE=1 或者 VERBOSE=1 make 命令来进行构建。

这时候，我们需要的目标文件 hello 已经构建完成，位于当前目录，尝试运行一下：

```
./hello
```

得到输出：

```
Hello World from Main
```

恭喜您，到这里为止您已经完全掌握了 cmake 的使用方法。

3，简单的解释：

我们来重新看一下 CMakeLists.txt，这个文件是 cmake 的构建定义文件，文件名是大小写相关的，如果工程存在多个目录，需要确保每个要管理的目录都存在一个 CMakeLists.txt。(关于多目录构建，后面我们会提到，这里不作过多解释)。

上面例子中的 CMakeLists.txt 文件内容如下：

```
PROJECT (HELLO)
SET(SRC_LIST main.c)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir "${HELLO_SOURCE_DIR})
ADD_EXECUTABLE(hello ${SRC_LIST})
```

PROJECT 指令的语法是：

```
PROJECT(projectname [CXX] [C] [Java])
```

你可以用这个指令定义工程名称，并可指定工程支持的语言，支持的语言列表是可以忽略的，默认情况表示支持所有语言。这个指令隐式的定义了两个 cmake 变量：<projectname>_BINARY_DIR 以及<projectname>_SOURCE_DIR，这里就是 HELLO_BINARY_DIR 和 HELLO_SOURCE_DIR(所以 CMakeLists.txt 中两个 MESSAGE 指令可以直接使用了这两个变量)，因为采用的是内部编译，两个变量目前指的都是工程所在路径/backup/cmake/t1，后面我们会讲到外部编译，两者所指代的内容会有所不同。

同时 cmake 系统也帮助我们预定义了 PROJECT_BINARY_DIR 和 PROJECT_SOURCE_DIR 变量，他们的值分别跟 HELLO_BINARY_DIR 与 HELLO_SOURCE_DIR 一致。

为了统一起见，建议以后直接使用 PROJECT_BINARY_DIR，PROJECT_SOURCE_DIR，即使修改了工程名称，也不会影响这两个变量。如果使用了 <projectname>_SOURCE_DIR，修改工程名称后，需要同时修改这些变量。

SET 指令的语法是：

```
SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
```

现阶段，你只需要了解 SET 指令可以用来显式的定义变量即可。

比如我们用到的是 SET(SRC_LIST main.c)，如果有多个源文件，也可以定义成：

SET(SRC_LIST main.c t1.c t2.c)。

MESSAGE 指令的语法是：

```
MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] "message to display"
...)
```

这个指令用于向终端输出用户定义的信息，包含了三种类型：

SEND_ERROR，产生错误，生成过程被跳过。

SATUS，输出前缀为—的信息。

FATAL_ERROR，立即终止所有 cmake 过程.

我们在这里使用的是 STATUS 信息输出，演示了由 PROJECT 指令定义的两个隐式变量 HELLO_BINARY_DIR 和 HELLO_SOURCE_DIR。

ADD_EXECUTABLE(hello ${SRC_LIST})

定义了这个工程会生成一个文件名为 hello 的可执行文件，相关的源文件是 SRC_LIST 中定义的源文件列表， 本例中你也可以直接写成 ADD_EXECUTABLE(hello main.c)。

在本例我们使用了${}来引用变量，这是 cmake 的变量应用方式，但是，有一些例外，比如在 IF 控制语句，变量是直接使用变量名引用，而不需要${}。如果使用了${}去应用变量，其实 IF 会去判断名为${}所代表的值的变量，那当然是不存在的了。

将本例改写成一个最简化的 CMakeLists.txt：

PROJECT(HELLO)

ADD_EXECUTABLE(hello main.c)

4，基本语法规则

前面提到过，cmake 其实仍然要使用”cmake 语言和语法”去构建，上面的内容就是所谓的”cmake 语言和语法”，最简单的语法规则是：

1，变量使用${}方式取值，但是在 IF 控制语句中是直接使用变量名

2，指令(参数 1 参数 2...)

参数使用括弧括起，参数之间使用空格或分号分开。

以上面的 ADD_EXECUTABLE 指令为例，如果存在另外一个 func.c 源文件，就要写成：

ADD_EXECUTABLE(hello main.c func.c)或者

ADD_EXECUTABLE(hello main.c;func.c)

3，指令是大小写无关的，参数和变量是大小写相关的。但，推荐你全部使用大写指令。

上面的 MESSAGE 指令我们已经用到了这条规则：

MESSAGE(STATUS “This is BINARY dir” ${HELLO_BINARY_DIR})

也可以写成：

MESSAGE(STATUS “This is BINARY dir ${HELLO_BINARY_DIR}”)

这里需要特别解释的是作为工程名的 HELLO 和生成的可执行文件 hello 是没有任何关系的。

hello 定义了可执行文件的文件名，你完全可以写成：

ADD_EXECUTABLE(t1 main.c)

编译后会生成一个 t1 可执行文件。

5，关于语法的疑惑

cmake 的语法还是比较灵活而且考虑到各种情况，比如

SET(SRC_LIST main.c)也可以写成 SET(SRC_LIST “main.c”)

是没有区别的，但是假设一个源文件的文件名是 fu nc.c(文件名中间包含了空格)。

这时候就必须使用双引号，如果写成了 SET(SRC_LIST fu nc.c)，就会出现错误，提示你找不到 fu 文件和 nc.c 文件。这种情况，就必须写成：

SET(SRC_LIST “fu nc.c”)

此外，你可以可以忽略掉 source 列表中的源文件后缀，比如可以写成

ADD_EXECUTABLE(t1 main)，cmake 会自动的在本目录查找 main.c 或者 main.cpp 等，当然，最好不要偷这个懒，以免这个目录确实存在一个 main.c 一个 main.

同时参数也可以使用分号来进行分割。

下面的例子也是合法的：

ADD_EXECUTABLE(t1 main.c t1.c)可以写成 ADD_EXECUTABLE(t1 main.c;t1.c).

我们只需要在编写 CMakeLists.txt 时注意形成统一的风格即可。

6，清理工程：

跟经典的 autotools 系列工具一样，运行：

make clean

即可对构建结果进行清理。

7，问题?问题！

“我尝试运行了 make distclean，这个指令一般用来清理构建过程中产生的中间文件的，如果要发布代码，必然要清理掉所有的中间文件，但是为什么在 cmake 工程中这个命令是无效的？”

是的，cmake 并不支持 make distclean，关于这一点，官方是有明确解释的：

因为 CMakeLists.txt 可以执行脚本并通过脚本生成一些临时文件，但是却没有办法来跟踪这些临时文件到底是哪些。因此，没有办法提供一个可靠的 make distclean 方案。

Some build trees created with GNU autotools have a "make distclean" target that cleans the build and also removes Makefiles and other parts of the generated build system. CMake does not generate a "make distclean" target because CMakeLists.txt files can run scripts and arbitrary commands; CMake has no way of tracking exactly which files are generated as part of running

CMake. Providing a distclean target would give users the false impression that it would work as expected. (CMake does generate a "make clean" target to remove files generated by the compiler and linker.)

A "make distclean" target is only necessary if the user performs an in-source build. CMake supports in-source builds, but we strongly encourage users to adopt the notion of an out-of-source build. Using a build tree that is separate from the source tree will prevent CMake from generating any files in the source tree. Because CMake does not change the source tree, there is no need for a distclean target. One can start a fresh build by deleting the build tree or creating a separate build tree.

同时，还有另外一个非常重要的提示，就是：我们刚才进行的是内部构建(in-source build)，而 cmake 强烈推荐的是外部构建(out-of-source build)。

8，内部构建与外部构建：

上面的例子展示的是“内部构建”，相信看到生成的临时文件比您的代码文件还要多的时候，估计这辈子你都不希望再使用内部构建:-D

举个简单的例子来说明外部构建，以编译 wxGTK 动态库和静态库为例，在 Everest 中打包方式是这样的：

解开 wxGTK 后。

在其中建立 static 和 shared 目录。

进入 static 目录，运行../configure –enable-static;make 会在 static 目录生成 wxGTK 的静态库。

进入 shared 目录，运行../configure –enable-shared;make 就会在 shared 目录生成动态库。

这就是外部编译的一个简单例子。

对于 cmake，内部编译上面已经演示过了，它生成了一些无法自动删除的中间文件，所以，引出了我们对外部编译的探讨，外部编译的过程如下：

1，首先，请清除 t1 目录中除 main.c CmakeLists.txt 之外的所有中间文件，最关键的是 CMakeCache.txt。

2，在 t1 目录中建立 build 目录，当然你也可以在任何地方建立 build 目录，不一定必须在工程目录中。

3，进入 build 目录，运行 cmake ..(注意,..代表父目录，因为父目录存在我们需要的 CMakeLists.txt，如果你在其他地方建立了 build 目录，需要运行 cmake <工程的全路径>)，查看一下 build 目录，就会发现了生成了编译需要的 Makefile 以及其他的中间文件.

4，运行 make 构建工程，就会在当前目录(build 目录)中获得目标文件 hello。

上述过程就是所谓的 out-of-source 外部编译，一个最大的好处是，对于原有的工程没有任何影响，所有动作全部发生在编译目录。通过这一点，也足以说服我们全部采用外部编译方式构建工程。

这里需要特别注意的是：

通过外部编译进行工程构建，HELLO_SOURCE_DIR 仍然指代工程路径，即 /backup/cmake/t1

而 HELLO_BINARY_DIR 则指代编译路径，即/backup/cmake/t1/build

9，小结：

本小节描述了使用 cmake 构建 Hello World 程序的全部过程，并介绍了三个简单的指令：

PROJECT/MESSAGE/ADD_EXECUTABLE 以及变量调用的方法，同时提及了两个隐式变量 <projectname>_SOURCE_DIR 及<projectname>_BINARY_DIR，演示了变量调用的方法，从这个过程来看，有些开发者可能会想，这实在比我直接写 Makefile 要复杂多了，甚至我都可以不编写 Makefile，直接使用 gcc main.c 即可生成需要的目标文件。是的，正如第一节提到的，如果工程只有几个文件，还是直接编写 Makefile 最简单。但是，kdelibs 压缩包达到了 50 多 M，您认为使用什么方案会更容易一点呢？

下一节，我们的任务是让 HelloWorld 看起来更像一个工程。

四，更好一点的 Hello World

没有最好，只有更好

从本小节开始，后面所有的构建我们都将采用 out-of-source 外部构建，约定的构建目录是工程目录下的 build 自录。

本小节的任务是让前面的 Hello World 更像一个工程，我们需要作的是：

1，为工程添加一个子目录 src，用来放置工程源代码；

2，添加一个子目录 doc，用来放置这个工程的文档 hello.txt

3，在工程目录添加文本文件 COPYRIGHT, README;

4，在工程目录添加一个 runhello.sh 脚本，用来调用 hello 二进制

4，将构建后的目标文件放入构建目录的 bin 子目录；

5，最终安装这些文件：将 hello 二进制与 runhello.sh 安装至/usr/bin，将 doc 目录的内容以及 COPYRIGHT/README 安装到/usr/share/doc/cmake/t2，将

1，准备工作：

在/backup/cmake/目录下建立 t2 目录。

将 t1 工程的 main.c 和 CMakeLists.txt 拷贝到 t2 目录中。

2，添加子目录 src：

```
mkdir src
mv main.c src
```

现在的工程看起来是这个样子：

一个子目录 src，一个 CMakeLists.txt。

上一节我们提到，需要为任何子目录建立一个 CMakeLists.txt，

进入子目录 src，编写 CMakeLists.txt 如下：

```
ADD_EXECUTABLE(hello main.c)
```

将 t2 工程的 CMakeLists.txt 修改为：

```
PROJECT(HELLO)
ADD_SUBDIRECTORY(src bin)
```

然后建立 build 目录，进入 build 目录进行外部编译。

```
cmake  ..
make
```

构建完成后，你会发现生成的目标文件 hello 位于 build/bin 目录中。

语法解释：

ADD_SUBDIRECTORY 指令

ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])

这个指令用于向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置。EXCLUDE_FROM_ALL 参数的含义是将这个目录从编译过程中排除，比如，工程的 example，可能就需要工程构建完成后，再进入 example 目录单独进行构建(当然，你也可以通过定义依赖来解决此类问题)。

上面的例子定义了将 src 子目录加入工程，并指定编译输出(包含编译中间结果)路径为 bin 目录。如果不进行 bin 目录的指定，那么编译结果(包括中间结果)都将存放在 build/src 目录(这个目录跟原有的 src 目录对应)，指定 bin 目录后，相当于在编译时将 src 重命名为 bin，所有的中间结果和目标二进制都将存放在 bin 目录。

这里需要提一下的是 SUBDIRS 指令，使用方法是：

SUBDIRS(dir1 dir2...)，但是这个指令已经不推荐使用。它可以一次添加多个子目录，并且，即使外部编译，子目录体系仍然会被保存。

如果我们在上面的例子中将 ADD_SUBDIRECTORY (src bin)修改为 SUBDIRS(src)。

那么在 build 目录中将出现一个 src 目录，生成的目标代码 hello 将存放在 src 目录中。

3，换个地方保存目标二进制

不论是 SUBDIRS 还是 ADD_SUBDIRECTORY 指令(不论是否指定编译输出目录)，我们都可以通过 SET 指令重新定义 EXECUTABLE_OUTPUT_PATH 和 LIBRARY_OUTPUT_PATH 变量来指定最终的目标二进制的位置(指最终生成的 hello 或者最终的共享库，不包含编译生成的中间文件)

SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)

SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)

在第一节我们提到了<projectname>_BINARY_DIR 和 PROJECT_BINARY_DIR 变量，他们指的编译发生的当前目录，如果是内部编译，就相当于 PROJECT_SOURCE_DIR 也就是工程代码所在目录，如果是外部编译，指的是外部编译所在目录，也就是本例中的 build 目录。

所以，上面两个指令分别定义了：

可执行二进制的输出路径为 build/bin 和库的输出路径为 build/lib.

本节我们没有提到共享库和静态库的构建，所以，你可以不考虑第二条指令。

问题是，我应该把这两条指令写在工程的 CMakeLists.txt 还是 src 目录下的 CMakeLists.txt，把握一个简单的原则，在哪里 ADD_EXECUTABLE 或 ADD_LIBRARY，如果需要改变目标存放路径，就在哪里加入上述的定义。

在这个例子里，当然就是指 src 下的 CMakeLists.txt 了。

4，如何安装。

安装的需要有两种，一种是从代码编译后直接 make install 安装，一种是打包时的指定目录安装。

所以，即使最简单的手工编写的 Makefile，看起来也是这个样子的：

```
DESTDIR=
install:
     mkdir -p $(DESTDIR)/usr/bin
     install -m 755 hello $(DESTDIR)/usr/bin
```

你可以通过：

```
make install
```

将 hello 直接安装到/usr/bin 目录，也可以通过 make install DESTDIR=/tmp/test 将他安装在

/tmp/test/usr/bin 目录，打包时这个方式经常被使用。

稍微复杂一点的是还需要定义 PREFIX，一般 autotools 工程，会运行这样的指令：

./configure –prefix=/usr 或者./configure --prefix=/usr/local 来指定 PREFIX

比如上面的 Makefile 就可以改写成：

```
DESTDIR=
PREFIX=/usr
install:
     mkdir -p $(DESTDIR)/$(PREFIX)/bin
     install -m 755 hello $(DESTDIR)/$(PREFIX)/bin
```

那么我们的 HelloWorld 应该怎么进行安装呢？

这里需要引入一个新的 cmake 指令 INSTALL 和一个非常有用的变量 CMAKE_INSTALL_PREFIX。

CMAKE_INSTALL_PREFIX 变量类似于 configure 脚本的 –prefix，常见的使用方法看起来是这个样子：

```
cmake -DCMAKE_INSTALL_PREFIX=/usr .
```

INSTALL 指令用于定义安装规则，安装的内容可以包括目标二进制、动态库、静态库以及文件、目录、脚本等。

INSTALL 指令包含了各种安装类型，我们需要一个个分开解释：

目标文件的安装：

```
        INSTALL(TARGETS targets...
            [[ARCHIVE|LIBRARY|RUNTIME]
                        [DESTINATION <dir>]
                        [PERMISSIONS permissions...]
                        [CONFIGURATIONS
          [Debug|Release|...]]
                        [COMPONENT <component>]
                        [OPTIONAL]
                       ] [...])
```

参数中的 TARGETS 后面跟的就是我们通过 ADD_EXECUTABLE 或者 ADD_LIBRARY 定义的目标文件，可能是可执行二进制、动态库、静态库。

目标类型也就相对应的有三种，ARCHIVE 特指静态库，LIBRARY 特指动态库，RUNTIME 特指可执行目标二进制。

DESTINATION 定义了安装的路径，如果路径以/开头，那么指的是绝对路径，这时候 CMAKE_INSTALL_PREFIX 其实就无效了。如果你希望使用 CMAKE_INSTALL_PREFIX 来定义安装路径，就要写成相对路径，即不要以/开头，那么安装后的路径就是

${CMAKE_INSTALL_PREFIX}/<DESTINATION 定义的路径>

举个简单的例子：

```
INSTALL(TARGETS myrun mylib mystaticlib
     RUNTIME DESTINATION bin
     LIBRARY DESTINATION lib
     ARCHIVE DESTINATION libstatic
     )
```

上面的例子会将：

可执行二进制 myrun 安装到${CMAKE_INSTALL_PREFIX}/bin 目录

动态库 libmylib 安装到${CMAKE_INSTALL_PREFIX}/lib 目录

静态库 libmystaticlib 安装到${CMAKE_INSTALL_PREFIX}/libstatic 目录

特别注意的是你不需要关心 TARGETS 具体生成的路径，只需要写上 TARGETS 名称就可以了。

普通文件的安装：

```
INSTALL(FILES files... DESTINATION <dir>
    [PERMISSIONS permissions...]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [RENAME <name>] [OPTIONAL])
```

可用于安装一般文件，并可以指定访问权限，文件名是此指令所在路径下的相对路径。如果默认不定义权限 PERMISSIONS，安装后的权限为：

OWNER_WRITE, OWNER_READ, GROUP_READ,和 WORLD_READ，即 644 权限。

非目标文件的可执行程序安装(比如脚本之类)：

```
INSTALL(PROGRAMS files... DESTINATION <dir>
    [PERMISSIONS permissions...]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [RENAME <name>] [OPTIONAL])
```

跟上面的 FILES 指令使用方法一样，唯一的不同是安装后权限为：

OWNER_EXECUTE, GROUP_EXECUTE, 和 WORLD_EXECUTE，即 755 权限

目录的安装：

```
INSTALL(DIRECTORY dirs... DESTINATION <dir>
    [FILE_PERMISSIONS permissions...]
    [DIRECTORY_PERMISSIONS permissions...]
    [USE_SOURCE_PERMISSIONS]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [[PATTERN <pattern> | REGEX <regex>]
     [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

这里主要介绍其中的 DIRECTORY、PATTERN 以及 PERMISSIONS 参数。

DIRECTORY 后面连接的是所在 Source 目录的相对路径，但务必注意：

abc 和 abc/有很大的区别。

如果目录名不以/结尾，那么这个目录将被安装为目标路径下的 abc，如果目录名以/结尾，代表将这个目录中的内容安装到目标路径，但不包括这个目录本身。

PATTERN 用于使用正则表达式进行过滤，PERMISSIONS 用于指定 PATTERN 过滤后的文件权限。

我们来看一个例子：

```
INSTALL(DIRECTORY icons scripts/ DESTINATION share/myproj
```

```
        PATTERN "CVS" EXCLUDE
        PATTERN "scripts/*"
        PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
                GROUP_EXECUTE GROUP_READ)
```

这条指令的执行结果是：

将 icons 目录安装到 <prefix>/share/myproj，将 scripts/中的内容安装到 <prefix>/share/myproj

不包含目录名为 CVS 的目录，对于 scripts/*文件指定权限为 OWNER_EXECUTE OWNER_WRITE OWNER_READ GROUP_EXECUTE GROUP_READ.

安装时 CMAKE 脚本的执行：

INSTALL([[SCRIPT <file>] [CODE <code>]] [...])

SCRIPT 参数用于在安装时调用 cmake 脚本文件（也就是<abc>.cmake 文件）

CODE 参数用于执行 CMAKE 指令，必须以双引号括起来。比如：

INSTALL(CODE "MESSAGE(\"Sample install message.\")")

安装还有几个被标记为过时的指令，比如 INSTALL_FILES 等，这些指令已经不再推荐使用，所以，这里就不再赘述了。

下面，我们就来改写我们的工程文件，让他来支持各种文件的安装，并且，我们要使用 CMAKE_INSTALL_PREFIX 指令。

5，修改 Helloworld 支持安装

在本节开头我们定义了本节的任务如下：

1，为工程添加一个子目录 src，用来存储源代码；

2，添加一个子目录 doc，用来存储这个工程的文档 hello.txt

3，在工程目录添加文本文件 COPYRIGHT, README;

4，在工程目录添加一个 runhello.sh 脚本，用来调用 hello 二进制

4，将构建后的目标文件放入构建目录的 bin 子目录；

5，最终安装这些文件：将 hello 二进制与 runhello.sh 安装至/<prefix>/bin，将 doc 目录中的 hello.txt 以及 COPYRIGHT/README 安装到 /<prefix>/share/doc/cmake/t2，将

首先我们先补上为添加的文件。

添加 doc 目录及文件：

```
cd /backup/cmake/t2
mkdir doc
vi doc/hello.txt
```

随便填写一些内容并保存

在工程目录添加 runhello.sh 脚本，内容为：

```
hello
```

添加工程目录中的 COPYRIGHT 和 README

```
touch COPYRIGHT
touch README
```

下面改写各目录的 CMakeLists.txt 文件。

1，安装 COPYRIGHT/README，直接修改主工程文件 CMakelists.txt，加入以下指令：

```
INSTALL(FILES COPYRIGHT README DESTINATION share/doc/cmake/t2)
```

2，安装 runhello.sh，直接修改主工程文件 CMakeLists.txt，加入如下指令：

```
INSTALL(PROGRAMS runhello.sh DESTINATION bin)
```

3，安装 doc 中的 hello.txt，这里有两种方式：一是通过在 doc 目录建立 CMakeLists.txt 并将 doc 目录通过 ADD_SUBDIRECTORY 加入工程来完成。另一种方法是直接在工程目录通过

```
INSTALL(DIRECTORY
```

来完成)，前者比较简单，各位可以根据兴趣自己完成，我们来尝试后者，顺便演示以下 DIRECTORY 的安装。

因为 hello.txt 要安装到/<prefix>/share/doc/cmake/t2，所以我们不能直接安装整个 doc 目录，这里采用的方式是安装 doc 目录中的内容，也就是使用”doc/”

在工程文件中添加

```
INSTALL(DIRECTORY doc/ DESTINATION share/doc/cmake/t2)
```

6，尝试我们修改的结果：

现在进入 build 目录进行外部编译，注意使用 CMAKE_INSTALL_PREFIX 参数，这里我们将它安装到了/tmp/t2 目录：

```
cmake -DCMAKE_INSTALL_PREFIX=/tmp/t2/usr ..
```

然后运行

```
make
make install
```

让我们进入/tmp/t2 目录看一下安装结果：

```
./usr
./usr/share
./usr/share/doc
```

./usr/share/doc/cmake

./usr/share/doc/cmake/t2

./usr/share/doc/cmake/t2/hello.txt

./usr/share/doc/cmake/t2/README

./usr/share/doc/cmake/t2/COPYRIGHT

./usr/bin

./usr/bin/hello

./usr/bin/runhello.sh

如果你要直接安装到系统，可以使用如下指令：

cmake -DCMAKE_INSTALL_PREFIX=/usr ..

7，一个疑问

如果我没有定义 CMAKE_INSTALL_PREFIX 会安装到什么地方？

你可以尝试以下，cmake ..;make;make install，你会发现 CMAKE_INSTALL_PREFIX 的默认定义是/usr/local

8，小结：

本小节主要描述了如何在工程中使用多目录、各种安装指令以及 CMAKE_INSTALL_PREFIX 变量(你真够牛的，这么点东西居然罗唆了这么多文字)

在下一小节，我们将探讨如何在 cmake 中构建动态库和静态库，以及如何使用外部头文件和外部共享库，毕竟，这是程序编写中最长使用的（对了，你知道用怎样的 gcc 参数可以直接构建静态库和动态库吗？）

五，静态库与动态库构建

读者云，太能罗唆了，一个 Hello World 就折腾了两个大节。OK，从本节开始，我们不再折腾 Hello World 了，我们来折腾 Hello World 的共享库。

本节的任务：

１，建立一个静态库和动态库，提供 HelloFunc 函数供其他程序编程使用，HelloFunc 向终端输出 Hello World 字符串。

２，安装头文件与共享库。

一，准备工作：

在/backup/cmake 目录建立 t3 目录，用于存放本节涉及到的工程

二，建立共享库

```
cd /backup/cmake/t3
mkdir lib
```

在 t3 目录下建立 CMakeLists.txt，内容如下：

```
PROJECT(HELLOLIB)
ADD_SUBDIRECTORY(lib)
```

在 lib 目录下建立两个源文件 hello.c 与 hello.h

hello.c 内容如下：

```
#include “hello.h”
void HelloFunc()
{
     printf(“Hello World\n”);
}
```

hello.h 内容如下：

```
#ifndef HELLO_H
#define HELLO_H
#include <stdio.h>
void HelloFunc();
#endif
```

在 lib 目录下建立 CMakeLists.txt，内容如下：

SET(LIBHELLO_SRC hello.c)

ADD_LIBRARY(hello SHARED ${LIBHELLO_SRC})

三，编译共享库：

仍然采用 out-of-source 编译的方式，按照习惯，我们建立一个 build 目录，在 build 目录中

cmake ..

make

这时，你就可以在 lib 目录得到一个 libhello.so，这就是我们期望的共享库。

如果你要指定 libhello.so 生成的位置，可以通过在主工程文件 CMakeLists.txt 中修改 ADD_SUBDIRECTORY(lib)指令来指定一个编译输出位置或者

在 lib/CMakeLists.txt 中添加

SET(LIBRARY_OUTPUT_PATH <路径>)来指定一个新的位置。

这两者的区别我们上一节已经提到了，所以，这里不再赘述，下面，我们解释一下一个新的指令 ADD_LIBRARY

```
ADD_LIBRARY(libname    [SHARED|STATIC|MODULE]
  [EXCLUDE_FROM_ALL]
        source1 source2 ... sourceN)
```

你不需要写全 libhello.so，只需要填写 hello 即可，cmake 系统会自动为你生成 libhello.X

类型有三种：

SHARED，动态库

STATIC，静态库

MODULE，在使用 dyld 的系统有效，如果不支持 dyld，则被当作 SHARED 对待。

EXCLUDE_FROM_ALL 参数的意思是这个库不会被默认构建，除非有其他的组件依赖或者手工构建。

四，添加静态库：

同样使用上面的指令，我们在支持动态库的基础上再为工程添加一个静态库，按照一般的习惯，静态库名字跟动态库名字应该是一致的，只不过后缀是.a 罢了。

下面我们用这个指令再来添加静态库：

ADD_LIBRARY(hello STATIC ${LIBHELLO_SRC})

然后再在 build 目录进行外部编译，我们会发现，静态库根本没有被构建，仍然只生成了一个动态库。因为 hello 作为一个 target 是不能重名的，所以，静态库构建指令无效。

如果我们把上面的 hello 修改为 hello_static:

```
ADD_LIBRARY(hello_static STATIC ${LIBHELLO_SRC})
```

就可以构建一个 libhello_static.a 的静态库了。

这种结果显示不是我们想要的，我们需要的是名字相同的静态库和动态库，因为 target 名称是唯一的，所以，我们肯定不能通过 ADD_LIBRARY 指令来实现了。这时候我们需要用到另外一个指令:

SET_TARGET_PROPERTIES，其基本语法是:

```
SET_TARGET_PROPERTIES(target1 target2 ...
                      PROPERTIES prop1 value1
                      prop2 value2 ...)
```

这条指令可以用来设置输出的名称，对于动态库，还可以用来指定动态库版本和 API 版本。

在本例中，我们需要作的是向 lib/CMakeLists.txt 中添加一条:

```
SET_TARGET_PROPERTIES(hello_static PROPERTIES OUTPUT_NAME "hello")
```

这样，我们就可以同时得到 libhello.so/libhello.a 两个库了。

与他对应的指令是:

```
GET_TARGET_PROPERTY(VAR target property)
```

具体用法如下例，我们向 lib/CMakeListst.txt 中添加:

```
GET_TARGET_PROPERTY(OUTPUT_VALUE hello_static OUTPUT_NAME)
MESSAGE(STATUS “This is the hello_static
OUTPUT_NAME:”${OUTPUT_VALUE})
```

如果没有这个属性定义，则返回 NOTFOUND.

让我们来检查一下最终的构建结果，我们发现，libhello.a 已经构建完成，位于 build/lib 目录中，但是 libhello.so 去消失了。这个问题的原因是：cmake 在构建一

个新的 target 时，会尝试清理掉其他使用这个名字的库，因为，在构建 libhello.a 时，就会清理掉 libhello.so.

为了回避这个问题，比如再次使用 SET_TARGET_PROPERTIES 定义 CLEAN_DIRECT_OUTPUT 属性。

向 lib/CMakeLists.txt 中添加：

```
SET_TARGET_PROPERTIES(hello PROPERTIES CLEAN_DIRECT_OUTPUT 1)
SET_TARGET_PROPERTIES(hello_static PROPERTIES CLEAN_DIRECT_OUTPUT 1)
```

这时候，我们再次进行构建，会发现 build/lib 目录中同时生成了 libhello.so 和 libhello.a

五，动态库版本号

按照规则，动态库是应该包含一个版本号的，我们可以看一下系统的动态库，一般情况是

```
libhello.so.1.2
libhello.so ->libhello.so.1
libhello.so.1->libhello.so.1.2
```

为了实现动态库版本号，我们仍然需要使用 SET_TARGET_PROPERTIES 指令。

具体使用方法如下：

```
SET_TARGET_PROPERTIES(hello PROPERTIES VERSION 1.2 SOVERSION 1)
```

VERSION 指代动态库版本，SOVERSION 指代 API 版本。

将上述指令加入 lib/CMakeLists.txt 中，重新构建看看结果。

在 build/lib 目录会生成：

```
libhello.so.1.2
libhello.so.1->libhello.so.1.2
libhello.so ->libhello.so.1
```

六，安装共享库和头文件

以上面的例子，我们需要将 libhello.a，libhello.so.x 以及 hello.h 安装到系统目录，才能真正让其他人开发使用，在本例中我们将 hello 的共享库安装到<prefix>/lib 目录，将 hello.h 安装到<prefix>/include/hello 目录。

利用上一节了解到的 INSTALL 指令，我们向 lib/CMakeLists.txt 中添加如下指令：

```
INSTALL(TARGETS hello hello_static
```

```
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib)

INSTALL(FILES hello.h DESTINATION include/hello)
```

注意，静态库要使用 ARCHIVE 关键字

通过：

```
cmake -DCMAKE_INSTALL_PREFIX=/usr ..
make
make install
```

我们就可以将头文件和共享库安装到系统目录/usr/lib 和/usr/include/hello 中了。

七，小结：

本小节，我们谈到了：

如何通过 ADD_LIBRARY 指令构建动态库和静态库。

如何通过 SET_TARGET_PROPERTIES 同时构建同名的动态库和静态库。

如何通过 SET_TARGET_PROPERTIES 控制动态库版本

最终使用上一节谈到的 INSTALL 指令来安装头文件和动态、静态库。

在下一节，我们需要编写另一个高级一点的 Hello World 来演示怎么使用我们已经构建的构建的共享库 libhello 和外部头文件。

六，如何使用外部共享库和头文件

抱歉，本节仍然继续折腾 Hello World.

上一节我们已经完成了 libhello 动态库的构建以及安装，本节我们的任务很简单：
编写一个程序使用我们上一节构建的共享库。

1，准备工作：
请在/backup/cmake 目录建立 t4 目录，本节所有资源将存储在 t4 目录。

2，重复以前的步骤，建立 src 目录，编写源文件 main.c，内容如下：

```
#include <hello.h>
int main()
{
     HelloFunc();
     return 0;
}
```

编写工程主文件 CMakeLists.txt

```
PROJECT(NEWHELLO)
ADD_SUBDIRECTORY(src)
```

编写 src/CMakeLists.txt

```
ADD_EXECUTABLE(main main.c)
```

上述工作已经严格按照我们前面季节提到的内容完成了。

３，外部构建
按照习惯，仍然建立 build 目录，使用 cmake ..方式构建。
过程：

```
cmake ..
make
```

构建失败，如果需要查看细节，可以使用第一节提到的方法
make VERBOSE=1 来构建
错误输出为是：

```
/backup/cmake/t4/src/main.c:1:19: error: hello.h: 没有那个文件或目录
```

４，引入头文件搜索路径。

hello.h 位于/usr/include/hello 目录中，并没有位于系统标准的头文件路径，

(有人会说了，白痴啊，你就不会 include <hello/hello.h>，同志，要这么干，我这一节就没什么可写了，只能选择一个 glib 或者 libX11 来写了，这些代码写出来很多同志是看不懂的)

为了让我们的工程能够找到 hello.h 头文件，我们需要引入一个新的指令 INCLUDE_DIRECTORIES，其完整语法为：

INCLUDE_DIRECTORIES([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)

这条指令可以用来向工程添加多个特定的头文件搜索路径，路径之间用空格分割，如果路径中包含了空格，可以使用双引号将它括起来，默认的行为是追加到当前的头文件搜索路径的后面，你可以通过两种方式来进行控制搜索路径添加的方式：

１，CMAKE_INCLUDE_DIRECTORIES_BEFORE，通过 SET 这个 cmake 变量为 on，可以将添加的头文件搜索路径放在已有路径的前面。

２，通过 AFTER 或者 BEFORE 参数，也可以控制是追加还是置前。

现在我们在 src/CMakeLists.txt 中添加一个头文件搜索路径，方式很简单，加入：

INCLUDE_DIRECTORIES(/usr/include/hello)

进入 build 目录，重新进行构建，这是找不到 hello.h 的错误已经消失，但是出现了一个新的错误：

main.c:(.text+0x12): undefined reference to `HelloFunc'

因为我们并没有 link 到共享库 libhello 上。

5，为 target 添加共享库

我们现在需要完成的任务是将目标文件链接到 libhello，这里我们需要引入两个新的指令

LINK_DIRECTORIES 和 TARGET_LINK_LIBRARIES

LINK_DIRECTORIES 的全部语法是：

LINK_DIRECTORIES(directory1 directory2 ...)

这个指令非常简单，添加非标准的共享库搜索路径，比如，在工程内部同时存在共享库和可执行二进制，在编译时就需要指定一下这些共享库的路径。这个例子中我们没有用到这个指令。

TARGET_LINK_LIBRARIES 的全部语法是：

```
TARGET_LINK_LIBRARIES(target library1
                      <debug | optimized> library2
                      ...)
```

这个指令可以用来为 target 添加需要链接的共享库，本例中是一个可执行文件，但是同样

可以用于为自己编写的共享库添加共享库链接。

为了解决我们前面遇到的 HelloFunc 未定义错误，我们需要作的是向 src/CMakeLists.txt 中添加如下指令：

```
TARGET_LINK_LIBRARIES(main hello)
```

也可以写成

```
TARGET_LINK_LIBRARIES(main libhello.so)
```

这里的 hello 指的是我们上一节构建的共享库 libhello.

进入 build 目录重新进行构建。

```
cmake ..
make
```

这是我们就得到了一个连接到 libhello 的可执行程序 main，位于 build/src 目录，运行 main 的结果是输出：

```
Hello World
```

让我们来检查一下 main 的链接情况：

```
ldd src/main
        linux-gate.so.1 =>  (0xb7ee7000)
        libhello.so.1 => /usr/lib/libhello.so.1 (0xb7ece000)
        libc.so.6 => /lib/libc.so.6 (0xb7d77000)
        /lib/ld-linux.so.2 (0xb7ee8000)
```

可以清楚的看到 main 确实链接了共享库 libhello，而且链接的是动态库 libhello.so.1

那如何链接到静态库呢?

方法很简单：

将 TARGET_LINK_LIBRRARIES 指令修改为：

```
TARGET_LINK_LIBRARIES(main libhello.a)
```

重新构建后再来看一下 main 的链接情况

```
ldd src/main
        linux-gate.so.1 =>  (0xb7fa8000)
        libc.so.6 => /lib/libc.so.6 (0xb7e3a000)
        /lib/ld-linux.so.2 (0xb7fa9000)
```

说明，main 确实链接到了静态库 libhello.a

6，特殊的环境变量 CMAKE_INCLUDE_PATH 和 CMAKE_LIBRARY_PATH

务必注意，这两个是环境变量而不是 cmake 变量。

使用方法是要在 bash 中用 export 或者在 csh 中使用 set 命令设置或者

CMAKE_INCLUDE_PATH=/home/include cmake ..等方式。

这两个变量主要是用来解决以前 autotools 工程中

--extra-include-dir 等参数的支持的。

也就是，如果头文件没有存放在常规路径(/usr/include, /usr/local/include 等)，则可以通过这些变量就行弥补。

我们以本例中的 hello.h 为例，它存放在/usr/include/hello 目录，所以直接查找肯定是找不到的。

前面我们直接使用了绝对路径 INCLUDE_DIRECTORIES(/usr/include/hello)告诉工程这个头文件目录。

为了将程序更智能一点，我们可以使用 CMAKE_INCLUDE_PATH 来进行，使用 bash 的方法如下:

```
export CMAKE_INCLUDE_PATH=/usr/include/hello
```

然后在头文件中将 INCLUDE_DIRECTORIES(/usr/include/hello)替换为:

```
FIND_PATH(myHeader hello.h)
IF(myHeader)
INCLUDE_DIRECTORIES(${myHeader})
ENDIF(myHeader)
```

上述的一些指令我们在后面会介绍。

这里简单说明一下，FIND_PATH 用来在指定路径中搜索文件名，比如:

```
FIND_PATH(myHeader NAMES hello.h PATHS /usr/include
/usr/include/hello)
```

这里我们没有指定路径，但是，cmake 仍然可以帮我们找到 hello.h 存放的路径，就是因为我们设置了环境变量 CMAKE_INCLUDE_PATH。

如果你不使用 FIND_PATH，CMAKE_INCLUDE_PATH 变量的设置是没有作用的，你不能指望它会直接为编译器命令添加参数-I<CMAKE_INCLUDE_PATH>。

以此为例，CMAKE_LIBRARY_PATH 可以用在 FIND_LIBRARY 中。

同样，因为这些变量直接为 FIND_指令所使用，所以所有使用 FIND_指令的 cmake 模块都会受益。

7，小节：

本节我们探讨了：

如何通过 INCLUDE_DIRECTORIES 指令加入非标准的头文件搜索路径。

如何通过 LINK_DIRECTORIES 指令加入非标准的库文件搜索路径。

如果通过 TARGET_LINK_LIBRARIES 为库或可执行二进制加入库链接。

并解释了如果链接到静态库。

到这里为止，您应该基本可以使用 cmake 工作了，但是还有很多高级的话题没有探讨，比如编译条件检查、编译器定义、平台判断、如何跟 pkgconfig 配合使用等等。

到这里，或许你可以理解前面讲到的“cmake 的使用过程其实就是学习 cmake 语言并编写 cmake 程序的过程”，既然是“cmake 语言”，自然涉及到变量、语法等.

下一节，我们将抛开程序的话题，看看常用的 CMAKE 变量以及一些基本的控制语法规则。

七，cmake 常用变量和常用环境变量

一，cmake 变量引用的方式：

前面我们已经提到了，使用${}进行变量的引用。在 IF 等语句中，是直接使用变量名而不通过${}取值

二，cmake 自定义变量的方式：

主要有隐式定义和显式定义两种，前面举了一个隐式定义的例子，就是 PROJECT 指令，他会隐式的定义<projectname>_BINARY_DIR 和<projectname>_SOURCE_DIR 两个变量。

显式定义的例子我们前面也提到了，使用 SET 指令，就可以构建一个自定义变量了。

比如：

SET(HELLO_SRC main.SOURCE_PATHc)，就 PROJECT_BINARY_DIR 可以通过${HELLO_SRC}来引用这个自定义变量了.

三，cmake 常用变量：

1，CMAKE_BINARY_DIR

PROJECT_BINARY_DIR

<projectname>_BINARY_DIR

这三个变量指代的内容是一致的，如果是 in source 编译，指得就是工程顶层目录，如果是 out-of-source 编译，指的是工程编译发生的目录。PROJECT_BINARY_DIR 跟其他指令稍有区别，现在，你可以理解为他们是一致的。

２，CMAKE_SOURCE_DIR

PROJECT_SOURCE_DIR

<projectname>_SOURCE_DIR

这三个变量指代的内容是一致的，不论采用何种编译方式，都是工程顶层目录。

也就是在 in source 编译时，他跟 CMAKE_BINARY_DIR 等变量一致。
PROJECT_SOURCE_DIR 跟其他指令稍有区别，现在，你可以理解为他们是一致的。

３，CMAKE_CURRENT_SOURCE_DIR

指的是当前处理的 CMakeLists.txt 所在的路径，比如上面我们提到的 src 子目录。

４，CMAKE_CURRRENT_BINARY_DIR

如果是 in-source 编译，它跟 CMAKE_CURRENT_SOURCE_DIR 一致，如果是 out-of-source 编译，他指的是 target 编译目录。

使用我们上面提到的 ADD_SUBDIRECTORY(src bin)可以更改这个变量的值。

使用 SET(EXECUTABLE_OUTPUT_PATH <新路径>)并不会对这个变量造成影响，它仅仅修改了最终目标文件存放的路径。

５，CMAKE_CURRENT_LIST_FILE

输出调用这个变量的 CMakeLists.txt 的完整路径

６，CMAKE_CURRENT_LIST_LINE

输出这个变量所在的行

7，CMAKE_MODULE_PATH

这个变量用来定义自己的 cmake 模块所在的路径。如果你的工程比较复杂，有可能会自己编写一些 cmake 模块，这些 cmake 模块是随你的工程发布的，为了让 cmake 在处理 CMakeLists.txt 时找到这些模块，你需要通过 SET 指令，将自己的 cmake 模块路径设置一下。

比如

SET(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)

这时候你就可以通过 INCLUDE 指令来调用自己的模块了。

８，EXECUTABLE_OUTPUT_PATH 和 LIBRARY_OUTPUT_PATH

分别用来重新定义最终结果的存放目录，前面我们已经提到了这两个变量。

9，PROJECT_NAME

返回通过 PROJECT 指令定义的项目名称。

四，cmake 调用环境变量的方式

使用$ENV{NAME}指令就可以调用系统的环境变量了。

比如

MESSAGE(STATUS “HOME dir: $ENV{HOME}”)

设置环境变量的方式是:

SET(ENV{变量名} 值)

1,CMAKE_INCLUDE_CURRENT_DIR

自动添加 CMAKE_CURRENT_BINARY_DIR 和 CMAKE_CURRENT_SOURCE_DIR 到当前处理的 CMakeLists.txt。相当于在每个 CMakeLists.txt 加入:

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_BINARY_DIR}
${CMAKE_CURRENT_SOURCE_DIR})

2,CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE

将工程提供的头文件目录始终至于系统头文件目录的前面，当你定义的头文件确实跟系统发

生冲突时可以提供一些帮助。

3,CMAKE_INCLUDE_PATH 和 CMAKE_LIBRARY_PATH 我们在上一节已经提及。

五，系统信息

1,CMAKE_MAJOR_VERSION，CMAKE 主版本号，比如 2.4.6 中的 2

2,CMAKE_MINOR_VERSION，CMAKE 次版本号，比如 2.4.6 中的 4

3,CMAKE_PATCH_VERSION，CMAKE 补丁等级，比如 2.4.6 中的 6

4,CMAKE_SYSTEM，系统名称，比如 Linux-2.6.22

5,CMAKE_SYSTEM_NAME，不包含版本的系统名，比如 Linux

6,CMAKE_SYSTEM_VERSION，系统版本，比如 2.6.22

7,CMAKE_SYSTEM_PROCESSOR，处理器名称，比如 i686.

8,UNIX，在所有的类 UNIX 平台为 TRUE，包括 OS X 和 cygwin

9,WIN32，在所有的 win32 平台为 TRUE，包括 cygwin

六，主要的开关选项：

1，CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS，用来控制 IF ELSE 语句的书写方式，在下一节语法部分会讲到。

2，BUILD_SHARED_LIBS

这个开关用来控制默认的库编译方式，如果不进行设置，使用 ADD_LIBRARY 并没有指定库类型的情况下，默认编译生成的库都是静态库。

如果 SET(BUILD_SHARED_LIBS ON)后，默认生成的为动态库。

３，CMAKE_C_FLAGS

设置 C 编译选项，也可以通过指令 ADD_DEFINITIONS()添加。

4，CMAKE_CXX_FLAGS

设置 C++编译选项，也可以通过指令 ADD_DEFINITIONS()添加。

小结：

本章介绍了一些较常用的 cmake 变量，这些变量仅仅是所有 cmake 变量的很少一部分，目前 cmake 的英文文档也是比较缺乏的，如果需要了解更多的 cmake 变量，更好的方式是阅读一些成功项目的 cmake 工程文件，比如 KDE4 的代码。

八，cmake 常用指令

前面我们讲到了 cmake 常用的变量，相信“cmake 即编程”的感觉会越来越明显，无论如何，我们仍然可以看到 cmake 比 autotools 要简单很多。接下来我们就要集中的看一看 cmake 所提供的常用指令。在前面的章节我们已经讨论了很多指令的用法，如 PROJECT，ADD_EXECUTABLE，INSTALL，ADD_SUBDIRECTORY，SUBDIRS，INCLUDE_DIRECTORIES，LINK_DIRECTORIES，TARGET_LINK_LIBRARIES，SET 等。

本节会引入更多的 cmake 指令，为了编写的方便，我们将按照 cmake man page 的顺序来介绍各种指令，不再推荐使用的指令将不再介绍，INSTALL 系列指令在安装部分已经做了非常详细的说明，本节也不在提及。*(你可以将本章理解成选择性翻译，但是会加入更多的个人理解)*

一，基本指令

1，ADD_DEFINITIONS

向 C/C++编译器添加-D 定义，比如：

ADD_DEFINITIONS(-DENABLE_DEBUG -DABC)，参数之间用空格分割。

如果你的代码中定义了#ifdef ENABLE_DEBUG #endif，这个代码块就会生效。

如果要添加其他的编译器开关，可以通过 CMAKE_C_FLAGS 变量和 CMAKE_CXX_FLAGS 变量设置。

２，ADD_DEPENDENCIES

定义 target 依赖的其他 target，确保在编译本 target 之前，其他的 target 已经被构建。

```
ADD_DEPENDENCIES(target-name depend-target1
                 depend-target2 ...)
```

３，ADD_EXECUTABLE、ADD_LIBRARY、ADD_SUBDIRECTORY 前面已经介绍过了，这里不再罗唆。

４，ADD_TEST 与 ENABLE_TESTING 指令。

ENABLE_TESTING 指令用来控制 Makefile 是否构建 test 目标，涉及工程所有目录。语法很简单，没有任何参数，ENABLE_TESTING()，一般情况这个指令放在工程的主 CMakeLists.txt 中.

ADD_TEST 指令的语法是：

ADD_TEST(testname Exename arg1 arg2 ...)

testname 是自定义的 test 名称，Exename 可以是构建的目标文件也可以是外部脚本等等。后面连接传递给可执行文件的参数。如果没有在同一个 CMakeLists.txt 中打开 ENABLE_TESTING()指令，任何 ADD_TEST 都是无效的。

比如我们前面的 Helloworld 例子，可以在工程主 CMakeLists.txt 中添加

```
ADD_TEST(mytest ${PROJECT_BINARY_DIR}/bin/main)
ENABLE_TESTING()
```

生成 Makefile 后，就可以运行 make test 来执行测试了。

5，AUX_SOURCE_DIRECTORY

基本语法是：

```
AUX_SOURCE_DIRECTORY(dir VARIABLE)
```

作用是发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表。因为目前 cmake 还不能自动发现新添加的源文件。

比如

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
ADD_EXECUTABLE(main ${SRC_LIST})
```

你也可以通过后面提到的 FOREACH 指令来处理这个 LIST

6，CMAKE_MINIMUM_REQUIRED

其语法为 CMAKE_MINIMUM_REQUIRED(VERSION versionNumber [FATAL_ERROR])

比如 CMAKE_MINIMUM_REQUIRED(VERSION 2.5 FATAL_ERROR)

如果 cmake 版本小与 2.5，则出现严重错误，整个过程中止。

7，EXEC_PROGRAM

在 CMakeLists.txt 处理过程中执行命令，并不会在生成的 Makefile 中执行。具体语法为：

```
EXEC_PROGRAM(Executable [directory in which to run]
                  [ARGS <arguments to executable>]
                  [OUTPUT_VARIABLE <var>]
                  [RETURN_VALUE <var>])
```

用于在指定的目录运行某个程序，通过 ARGS 添加参数，如果要获取输出和返回值，可通过 OUTPUT_VARIABLE 和 RETURN_VALUE 分别定义两个变量.

这个指令可以帮助你在 CMakeLists.txt 处理过程中支持任何命令，比如根据系统情况去修改代码文件等等。

举个简单的例子，我们要在 src 目录执行 ls 命令，并把结果和返回值存下来。

可以直接在 src/CMakeLists.txt 中添加：

```
EXEC_PROGRAM(ls ARGS "*.c" OUTPUT_VARIABLE LS_OUTPUT RETURN_VALUE LS_RVALUE)
IF(not LS_RVALUE)
MESSAGE(STATUS "ls result: " ${LS_OUTPUT})
ENDIF(not LS_RVALUE)
```

在 cmake 生成 Makefile 的过程中，就会执行 ls 命令，如果返回 0，则说明成功执行，那么就输出 ls *.c 的结果。关于 IF 语句，后面的控制指令会提到。

8，FILE 指令

文件操作指令，基本语法为：

```
        FILE(WRITE filename "message to write"... )
        FILE(APPEND filename "message to write"... )
        FILE(READ filename variable)
        FILE(GLOB  variable [RELATIVE path] [globbing expressions]...)
        FILE(GLOB_RECURSE variable [RELATIVE path]
             [globbing expressions]...)
        FILE(REMOVE [directory]...)
        FILE(REMOVE_RECURSE [directory]...)
        FILE(MAKE_DIRECTORY [directory]...)
        FILE(RELATIVE_PATH variable directory file)
        FILE(TO_CMAKE_PATH path result)
        FILE(TO_NATIVE_PATH path result)
```

这里的语法都比较简单，不在展开介绍了。

9，INCLUDE 指令，用来载入 CMakeLists.txt 文件，也用于载入预定义的 cmake 模块.

```
        INCLUDE(file1 [OPTIONAL])
        INCLUDE(module [OPTIONAL])
```

OPTIONAL 参数的作用是文件不存在也不会产生错误。

你可以指定载入一个文件，如果定义的是一个模块，那么将在 CMAKE_MODULE_PATH 中搜索这个模块并载入。

载入的内容将在处理到 INCLUDE 语句是直接执行。

二，INSTALL 指令

INSTALL 系列指令已经在前面的章节有非常详细的说明，这里不在赘述，可参考前面的安装部分。

三，FIND_指令

FIND_系列指令主要包含一下指令：

FIND_FILE(<VAR> name1 path1 path2 ...)

VAR 变量代表找到的文件全路径，包含文件名

FIND_LIBRARY(<VAR> name1 path1 path2 ...)

VAR 变量表示找到的库全路径，包含库文件名

FIND_PATH(<VAR> name1 path1 path2 ...)

VAR 变量代表包含这个文件的路径。

FIND_PROGRAM(<VAR> name1 path1 path2 ...)

VAR 变量代表包含这个程序的全路径。

FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE]
[[REQUIRED|COMPONENTS] [componets...]])

用来调用预定义在 CMAKE_MODULE_PATH 下的 Find<name>.cmake 模块，你也可以自己定义 Find<name>模块，通过 SET(CMAKE_MODULE_PATH dir)将其放入工程的某个目录中供工程使用，我们在后面的章节会详细介绍 FIND_PACKAGE 的使用方法和 Find 模块的编写。

FIND_LIBRARY 示例：

```
FIND_LIBRARY(libX X11 /usr/lib)
IF(NOT libX)
MESSAGE(FATAL_ERROR “libX not found”)
ENDIF(NOT libX)
```

四，控制指令：

1,IF 指令，基本语法为：

```
IF(expression)
  # THEN section.
  COMMAND1(ARGS ...)
```

```
    COMMAND2(ARGS ...)
    ...
  ELSE(expression)
    # ELSE section.
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
  ENDIF(expression)
```

另外一个指令是 ELSEIF，总体把握一个原则，凡是出现 IF 的地方一定要有对应的 ENDIF.出现 ELSEIF 的地方，ENDIF 是可选的。

表达式的使用方法如下：

IF(var)，如果变量不是：空，0，N，NO，OFF，FALSE，NOTFOUND 或 <var>_NOTFOUND 时，表达式为真。

IF(NOT var)，与上述条件相反。

IF(var1 AND var2)，当两个变量都为真是为真。

IF(var1 OR var2)，当两个变量其中一个为真时为真。

IF(COMMAND cmd)，当给定的 cmd 确实是命令并可以调用是为真。

IF(EXISTS dir)或者 IF(EXISTS file)，当目录名或者文件名存在时为真。

IF(file1 IS_NEWER_THAN file2)，当 file1 比 file2 新，或者 file1/file2 其中有一个不存在时为真，文件名请使用完整路径。

IF(IS_DIRECTORY dirname)，当 dirname 是目录时，为真。

IF(variable MATCHES regex)

IF(string MATCHES regex)

当给定的变量或者字符串能够匹配正则表达式 regex 时为真。比如：

```
IF("hello" MATCHES "ell")
MESSAGE("true")
ENDIF("hello" MATCHES "ell")
```

IF(variable LESS number)

IF(string LESS number)

IF(variable GREATER number)

IF(string GREATER number)

IF(variable EQUAL number)

IF(string EQUAL number)

数字比较表达式

IF(variable STRLESS string)

IF(string STRLESS string)

IF(variable STRGREATER string)

IF(string STRGREATER string)

IF(variable STREQUAL string)

IF(string STREQUAL string)

按照字母序的排列进行比较.

IF(DEFINED variable)，如果变量被定义，为真。

一个小例子，用来判断平台差异:

```
IF(WIN32)
     MESSAGE(STATUS “This is windows.”)
     #作一些 Windows 相关的操作
ELSE(WIN32)
     MESSAGE(STATUS “This is not windows”)
     #作一些非 Windows 相关的操作
ENDIF(WIN32)
```

上述代码用来控制在不同的平台进行不同的控制，但是，阅读起来却并不是那么舒服，ELSE(WIN32)之类的语句很容易引起歧义。

这就用到了我们在“常用变量”一节提到的 CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS 开关。

可以 SET(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS ON)

这时候就可以写成:

```
IF(WIN32)
ELSE()
ENDIF()
```

如果配合 ELSEIF 使用，可能的写法是这样：

```
IF(WIN32)
#do something related to WIN32
ELSEIF(UNIX)
#do something related to UNIX
ELSEIF(APPLE)
#do something related to APPLE
ENDIF(WIN32)
```

2,WHILE

WHILE 指令的语法是：

```
        WHILE(condition)
          COMMAND1(ARGS ...)
          COMMAND2(ARGS ...)
          ...
        ENDWHILE(condition)
```

其真假判断条件可以参考 IF 指令。

3,FOREACH

FOREACH 指令的使用方法有三种形式：

1，列表

```
        FOREACH(loop_var arg1 arg2 ...)
          COMMAND1(ARGS ...)
          COMMAND2(ARGS ...)
          ...
        ENDFOREACH(loop_var)
```

像我们前面使用的 AUX_SOURCE_DIRECTORY 的例子

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
FOREACH(F ${SRC_LIST})
     MESSAGE(${F})
ENDFOREACH(F)
```

2，范围

```
FOREACH(loop_var RANGE total)
ENDFOREACH(loop_var)
```

从 0 到 total 以 1 为步进

举例如下：

```
FOREACH(VAR RANGE 10)
MESSAGE(${VAR})
ENDFOREACH(VAR)
```

最终得到的输出是：

```
0
1
2
3
4
5
6
7
8
9
10
```

３，范围和步进

```
FOREACH(loop_var RANGE start stop [step])
ENDFOREACH(loop_var)
```

从 start 开始到 stop 结束，以 step 为步进，

举例如下

```
FOREACH(A RANGE 5 15 3)
MESSAGE(${A})
ENDFOREACH(A)
```

最终得到的结果是：

```
5
8
11
14
```

这个指令需要注意的是，知道遇到 ENDFOREACH 指令，整个语句块才会得到真正的执行。

小结：

本小节基本涵盖了常用的 cmake 指令，包括基本指令、查找指令、安装指令以及控制语句等，特别需要注意的是，在控制语句条件中使用变量，不能用${}引用，而是直接应用变量名。

掌握了以上的各种控制指令，你应该完全可以通过 cmake 管理复杂的程序了，下一节，我们将介绍一个比较复杂的例子，通过他来演示本章的一些指令，并介绍模块的概念。

九，复杂的例子：模块的使用和自定义模块

你现在还会觉得 *cmake* 简单吗?

本章我们将着重介绍系统预定义的 Find 模块的使用以及自己编写 Find 模块，系统中提供了其他各种模块，一般情况需要使用 INCLUDE 指令显式的调用，FIND_PACKAGE 指令是一个特例，可以直接调用预定义的模块。

其实使用纯粹依靠 cmake 本身提供的基本指令来管理工程是一件非常复杂的事情，所以，cmake 设计成了可扩展的架构，可以通过编写一些通用的模块来扩展 cmake.

在本章，我们准备首先介绍一下 cmake 提供的 FindCURL 模块的使用。然后，基于我们前面的 libhello 共享库，编写一个 FindHello.cmake 模块。

一，使用 FindCURL 模块

在/backup/cmake 目录建立 t5 目录，用于存放我们的 CURL 的例子。

建立 src 目录，并建立 src/main.c，内容如下：

```
#include <curl/curl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
FILE *fp;
int write_data(void *ptr, size_t size, size_t nmemb, void *stream)
{
int written = fwrite(ptr, size, nmemb, (FILE *)fp);
return written;
}
int main()
{
const char * path = “/tmp/curl-test”;
const char * mode = “w”;
fp = fopen(path,mode);
curl_global_init(CURL_GLOBAL_ALL);
CURLcode res;
CURL *curl = curl_easy_init();
curl_easy_setopt(curl, CURLOPT_URL, “http://www.linux-ren.org”);
curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
curl_easy_setopt(curl, CURLOPT_VERBOSE, 1);
res = curl_easy_perform(curl);
curl_easy_cleanup(curl);
}
```

这段代码的作用是通过 curl 取回 www.linux-ren.org 的首页并写入/tmp/curl-test 文件中。

建立主工程文件 CMakeLists.txt

```
PROJECT(CURLTEST)
ADD_SUBDIRECTORY(src)
```

建立 src/CMakeLists.txt

```
ADD_EXECUTABLE(curltest main.c)
```

现在自然是没办法编译的，我们需要添加 curl 的头文件路径和库文件。

方法 1:

直接通过 INCLUDE_DIRECTORIES 和 TARGET_LINK_LIBRARIES 指令添加:

我们可以直接在 src/CMakeLists.txt 中添加:

```
INCLUDE_DIRECTORIES(/usr/include)
TARGET_LINK_LIBRARIES(curltest curl)
```

然后建立 build 目录进行外部构建即可。

现在我们要探讨的是使用 cmake 提供的 FindCURL 模块。

方法 2，使用 FindCURL 模块。

向 src/CMakeLists.txt 中添加:

```
FIND_PACKAGE(CURL)
IF(CURL_FOUND)
   INCLUDE_DIRECTORIES(${CURL_INCLUDE_DIR})
   TARGET_LINK_LIBRARIES(curltest ${CURL_LIBRARY})
ELSE(CURL_FOUND)
     MESSAGE(FATAL_ERROR ”CURL library not found”)
ENDIF(CURL_FOUND)
```

对于系统预定义的 Find<name>.cmake 模块，使用方法一般如上例所示:

每一个模块都会定义以下几个变量

- <name>_FOUND
- <name>_INCLUDE_DIR or <name>_INCLUDES
- <name>_LIBRARY or <name>_LIBRARIES

你可以通过<name>_FOUND 来判断模块是否被找到，如果没有找到，按照工程的需要关闭某些特性、给出提醒或者中止编译，上面的例子就是报出致命错误并终止构建。

如果<name>_FOUND 为真，则将<name>_INCLUDE_DIR 加入 INCLUDE_DIRECTORIES，将<name>_LIBRARY 加入 TARGET_LINK_LIBRARIES 中。

我们再来看一个复杂的例子，通过<name>_FOUND 来控制工程特性:

```
SET(mySources viewer.c)
SET(optionalSources)
```

```
SET(optionalLibs)
FIND_PACKAGE(JPEG)
IF(JPEG_FOUND)
   SET(optionalSources ${optionalSources} jpegview.c)
   INCLUDE_DIRECTORIES( ${JPEG_INCLUDE_DIR} )
   SET(optionalLibs ${optionalLibs} ${JPEG_LIBRARIES} )
   ADD_DEFINITIONS(-DENABLE_JPEG_SUPPORT)
ENDIF(JPEG_FOUND)

IF(PNG_FOUND)
   SET(optionalSources ${optionalSources} pngview.c)
   INCLUDE_DIRECTORIES( ${PNG_INCLUDE_DIR} )
   SET(optionalLibs ${optionalLibs} ${PNG_LIBRARIES} )
   ADD_DEFINITIONS(-DENABLE_PNG_SUPPORT)
ENDIF(PNG_FOUND)

ADD_EXECUTABLE(viewer ${mySources} ${optionalSources} )
TARGET_LINK_LIBRARIES(viewer ${optionalLibs}
```

通过判断系统是否提供了 JPEG 库来决定程序是否支持 JPEG 功能。

二，编写属于自己的 FindHello 模块。

我们在此前的 t3 实例中，演示了构建动态库、静态库的过程并进行了安装。

接下来，我们在 t6 示例中演示如何自定义 FindHELLO 模块并使用这个模块构建工程：

请在建立/backup/cmake/中建立 t6 目录，并在其中建立 cmake 目录用于存放我们自己定义的 FindHELLO.cmake 模块，同时建立 src 目录，用于存放我们的源文件。

1，定义 cmake/FindHELLO.cmake 模块

```
FIND_PATH(HELLO_INCLUDE_DIR hello.h /usr/include/hello
/usr/local/include/hello)
FIND_LIBRARY(HELLO_LIBRARY NAMES hello PATH /usr/lib
/usr/local/lib)
IF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
   SET(HELLO_FOUND TRUE)
ENDIF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
IF (HELLO_FOUND)
   IF (NOT HELLO_FIND_QUIETLY)
      MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
```

```
   ENDIF (NOT HELLO_FIND_QUIETLY)
ELSE (HELLO_FOUND)
   IF (HELLO_FIND_REQUIRED)
      MESSAGE(FATAL_ERROR "Could not find hello library")
   ENDIF (HELLO_FIND_REQUIRED)
ENDIF (HELLO_FOUND)
```

针对上面的模块让我们再来回顾一下 FIND_PACKAGE 指令：

```
        FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE]
                [[REQUIRED|COMPONENTS] [componets...]])
```

前面的 CURL 例子中我们使用了最简单的 FIND_PACKAGE 指令，其实他可以使用多种参数，

QUIET 参数，对应与我们编写的 FindHELLO 中的 HELLO_FIND_QUIETLY，如果不指定这个参数，就会执行：

```
MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
```

REQUIRED 参数，其含义是指这个共享库是否是工程必须的，如果使用了这个参数，说明这个链接库是必备库，如果找不到这个链接库，则工程不能编译。对应于 FindHELLO.cmake 模块中的 HELLO_FIND_REQUIRED 变量。

同样，我们在上面的模块中定义了 HELLO_FOUND, HELLO_INCLUDE_DIR,HELLO_LIBRARY 变量供开发者在 FIND_PACKAGE 指令中使用。

OK，下面建立 src/main.c，内容为：

```
#include <hello.h>
int main()
{
     HelloFunc();
     return 0;
}
```

建立 src/CMakeLists.txt 文件，内容如下：

```
FIND_PACKAGE(HELLO)
IF(HELLO_FOUND)
    ADD_EXECUTABLE(hello main.c)
    INCLUDE_DIRECTORIES(${HELLO_INCLUDE_DIR})
    TARGET_LINK_LIBRARIES(hello ${HELLO_LIBRARY})
ENDIF(HELLO_FOUND)
```

为了能够让工程找到 FindHELLO.cmake 模块(存放在工程中的 cmake 目录)

我们在主工程文件 CMakeLists.txt 中加入：

```
SET(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)
```

三，使用自定义的 FindHELLO 模块构建工程

仍然采用外部编译的方式，建立 build 目录，进入目录运行：

cmake ..

我们可以从输出中看到：

Found Hello: /usr/lib/libhello.so

如果我们把上面的 FIND_PACKAGE(HELLO)修改为 FIND_PACKAGE(HELLO QUIET),则不会看到上面的输出。

接下来就可以使用 make 命令构建工程，运行：

./src/hello 可以得到输出

Hello World。

说明工程成功构建。

四，如果没有找到 hello library 呢?

我们可以尝试将/usr/lib/libhello.x 移动到/tmp 目录，这样，按照 FindHELLO 模块的定义，就找不到 hello library 了，我们再来看一下构建结果：

cmake ..

仍然可以成功进行构建，但是这时候是没有办法编译的。

修改 FIND_PACKAGE(HELLO)为 FIND_PACKAGE(HELLO REQUIRED)，将 hello library 定义为工程必须的共享库。

这时候再次运行 cmake ..

我们得到如下输出：

CMake Error: Could not find hello library.

因为找不到 libhello.x，所以，整个 Makefile 生成过程被出错中止。

小结：

在本节中，我们学习了如何使用系统提供的 Find<NAME>模块并学习了自己编写 Find<NAME>模块以及如何在工程中使用这些模块。

后面的章节，我们会逐渐学习更多的 cmake 模块使用方法以及用 cmake 来管理 GTK 和 QT4 工程。

目录

一、**CMake2.8.3** 选项 4

用法 4

描述 5

选项 5

生成器 10

二、CMake 命令 12

CMD#1 : add_custom_command 12

CMD#2: add_custom_target 14

CMD#3:add_definitions 15

CMD#4:add_dependencies 15

CMD#5:add_executable: 16

CMD#6:add_library 16

CMD#7:add_subdirectory 17

CMD#8:add_test 18

CMD#9:aux_source_directory 19

CMD#10:break 19

CMD#11:build_command 19

CMD#12:cmake_minimum_required 20

CMD#13:cmake_policy 20

CMD#14:configure_file: 21

CMD#15:create_test_sourcelist: 22

CMD#16:define_property: 22

CMD#17: else 23

CMD#18: elseif 23

CMD#19: enable_language 23

CMD#20: enable_testing 23

CMD#21: endforeach 24

CMD#22: endfunction 24

CMD#23: endif 24
CMD#24: endmacro 24
CMD#25: endwhile 24
CMD#26: execute_process 25
CMD#27:export 25
CMD#28: file 26
CMD#29:find_file 28
CMD#30:find_library 30
CMD#31:find_package 33
CMD#32 : find_path 38
CMD#33:find_program 40
CMD#34:fltk_wrap_ui 42
CMD#35 : foreach 42
CMD#36 : function 43
CMD#37 : get_cmake_property 43
CMD#38 : get_directory_property 44
CMD#39 : get_filename_component 44
CMD#40 : get_property 44
CMD#41 : get_source_file_property 45
CMD#42 : get_target_property 45
CMD#43 : get_test_property 46
CMD#44 : if 46
CMD#45 : include 49
CMD#46 : include_directories 50
CMD#47 : include_external_msproject 50
CMD#48 : include_regular_expression 50
CMD#49 : install 50
CMD#50 : link_directories 指定连接器查找库的路径。 55
CMD#51: list 55
CMD#52： load_cache 56

CMD#53：load_command 56
CMD#54：macro 57
CMD#55：mark_as_advanced 57
CMD#56：math 57
CMD#57：message 58
CMD#58: option 58
CMD#59: output_required_files 58
CMD#60: project 58
CMD#61: qt_wrap_cpp 59
CMD#62: qt_wrap_ui 59
CMD#63: remove_definitions 59
CMD#64: return 59
CMD#65: separate_arguments 60
CMD#66: set 60
CMD#67: set_directory_properties 61
CMD#68: set_property 61
CMD#69: set_source_files_properties 62
CMD#70: set_target_properties 62
CMD#71: set_tests_properties 64
CMD#72: site_name 64
CMD#73: source_group 64
CMD#74: string 64
CMD#75: target_link_libraries 66
CMD#76: try_compile 67
CMD#77: try_run 68
CMD#78 unset 69
CMD#79 : variable_watch 69
CMD#80: while 69

公司的一个项目使用 CMake 作为跨平台构建工具；业务有需求，当然要好好研读一下官方的技术手册。目前的计划是先把官方手册翻译一下，了解清楚 CMake 中的各种命令、属性和变量的用法。同时在工作中也会阅读 CMake 的真实源码，后续会基于此陆续写一些工程中使用 CMake 的心得。CMake 的版本也在不停更新，有些新的命令和变量会随着版本更新添加进来，这是后事了，暂且不管；现在锁定 CMake 2.8.3 作为手册翻译的版本。

作为园子里的新丁，文章在术语和表达等等方面会有欠缺的地方，还请大侠们慷慨指点。另外，罗马不是一天建成的，长长的手册翻译完也不知道要经历多少日升月落；不过还是希望自己能够坚持下去:-)。

一、 CMake2.8.3 选项

- 命令名称
- 用法
- 描述
- 命令选项
- 生成器
- 命令
- 属性
- 全局域属性
- 目录属性
- 目标属性
- 测试属性
- 源代码属性
- **Cache Entries** 属性
- 兼容性命令
- **CMake** 标准模块
- **CMake** 策略
- 变量
- 改变行为的变量
- 描述系统的变量
- 语言变量
- 控制构建的变量
- 提供信息的变量
- 版权
- 其他参考资料

命令名称

cmake - 跨平台 Makefile 生成工具。

用法

cmake [选项] <源码路径>

cmake [选项] <现有构建路径>

描述

cmake 可执行程序是 CMake 的命令行界面。它可以用脚本对工程进行配置。工程配置设置可以在命令行中使用-D 选项指定。使用-i 选项，cmake 将通过提示交互式地完成该设置。

CMake 是一个跨平台的构建系统生成工具。它使用平台无关的 CMake 清单文件 CMakeLists.txt，指定工程的构建过程；源码树的每个路径下都有这个文件。CMake 产生一个适用于具体平台的构建系统，用户使用这个系统构建自己的工程。

选项

-C <initial-cache>: 预加载一个脚本填充缓存文件。

当 cmake 在一个空的构建树上第一次运行时，它会创建一个 CMakeCache.txt 文件，然后向其中写入可定制的项目设置数据。-C 选项可以用来指定一个文件，在第一次解析这个工程的 cmake 清单文件时，从这个文件加载缓存的条目(cache entries)信息。被加载的缓存条目比项目默认的值有更高的优先权。参数中给定的那个文件应该是一个 CMake 脚本，其中包含有使用 CACHE 选项的 SET 命令；而不是一个缓存格式的文件。

-D <var>:<type>=<value>: 创建一个 CMake 的缓存条目。

当 cmake 第一次运行于一个空的构建数时，它会创建一个 CMakeCache.txt 文件，并且使用可定制的工程设置来填充这个文件。这个选项可以用来指定优先级高于工程的默认值的工程设置值。这个参数可以被重复多次，用来填充所需要数量的缓存条目(cache entries)。

-U <globbing_expr>: 从 CMake 的缓存文件中删除一条匹配的条目。

该选项可以用来删除 CMakeCache.txt 文件中的一或多个变量。文件名匹配表达式(globbing expression)支持通配符*和?的使用。该选项可以重复多次以删除期望数量的缓存条目。使用它时要小心，你可能因此让自己的 CMakeCache.txt 罢工。

-G <generator-name>: 指定一个 makefile 生成工具。

在具体的平台上，CMake 可以支持多个原生的构建系统。makefile 生成工具的职责是生成特定的构建系统。可能的生成工具的名称将在生成工具一节给出。

-Wno-dev: 抑制开发者警告。

抑制那些为 CMakeLists.txt 文件的作者准备的警告信息。

-Wdev: 使能开发者警告信息输出功能。

允许那些为 CMakeLists.txt 文件的作者准备的警告信息。

-E: CMake 命令行模式。

为了真正做到与平台无关，CMake 提供了一系列可以用于所有系统上的的命令。以-E 参数运行 CMake 会帮助你获得这些命令的用法。可以使用的命令有：chdir, copy, copy_if_different copy_directory, compare_files, echo, echo_append, environment,

make_directory, md5sum, remove_directory, remove, tar, time, touch, touch_nocreate, write_regv, delete_regv, comspec, create_symlink。

-i：以向导模式运行 CMake。

向导模式是在没有 GUI 时，交互式地运行 cmake 的模式。cmake 会弹出一系列的提示，要求用户回答关于工程配置的一行问题。这些答复会被用来设置 cmake 的缓存值。

-L[A][H]：列出缓存的变量中的非高级的变量。

-L 选项会列出缓存变量会运行 CMake，并列出所有 CMake 的内有被标记为 INTERNAL 或者 ADVANCED 的缓存变量。这会显示当前的 CMake 配置信息，然后你可以用-D 选项改变这些选项。修改一些变量可能会引起更多的变量被创建出来。如果指定了 A 选项，那么命令也会显示高级变量。如果指定了 H 选项，那么命令会显示每个变量的帮助信息。

--build <dir>：构建由 CMake 生成的工程的二进制树。（这个选项的含义我不是很清楚—译注）

该选项用以下的选项概括了内置构建工具的命令行界面

<dir> = 待创建的工程二进制路径。
--target <tgt> = 构建<tgt>，而不是默认目标。
--config <cfg> = 对于多重配置工具，选择配置<cfg>。
--clean-first = 首先构建目标的 clean 伪目标，然后再构建。
（如果仅仅要 clean 掉，使用--target 'clean'选项。）

-- = 向内置工具（native tools）传递剩余的选项。

运行不带选项的 cmake --build 来获取快速帮助信息。

-N：查看模式。

仅仅加载缓存信息，并不实际运行配置和生成步骤。

-P <file>：处理脚本模式。

将给定的 cmake 文件按照 CMake 语言编写的脚本进行处理。不执行配置和生成步骤，不修改缓存信息。如果要使用-D 选项定义变量，-D 选项必须在-P 选项之前。

--graphviz=[file]：生成依赖的 graphviz 图。

生成一个 graphviz 软件的输入文件，其中包括了项目中所有库和可执行文件之间的依赖关系。

--system-information [file]：输出与该系统相关的信息。

输出范围比较广的、与当前使用的系统有关的信息。如果在一个 CMake 工程的二进制构建树的顶端运行该命令，它还会打印一些附加信息，例如缓存，日志文件等等。

--debug-trycompile: 不删除"尝试编译"路径。

不删除那些为 try_compile 调用生成的路径。这在调试失败的 try_compile 文件时比较有用。不过，因为上一次"尝试编译"生成的旧的垃圾输出文件也许会导致一次不正确通过/不通过，且该结果与上次测试的结果不同，所以该选项可能会改变"尝试编译"的结果。对于某一次"尝试编译"，该选项最好只用一次；并且仅仅在调试时使用。

--debug-output: 将 cmake 设置为调试模式。

在 cmake 运行时，打印额外的信息；比如使用 message(send_error)调用得到的栈跟踪信息。

--trace: 将 cmake 设置为跟踪模式。

用 message(send_error)调用，打印所有调用生成的跟踪信息，以及这些调用发生的位置。（这句话含义不是很确定—译注。）

--help-command cmd [file]: 打印单个命令 cmd 的帮助信息，然后退出。

显示给定的命令的完整的文档。如果指定了[file]参数，该文档会写入该文件，其输出格式由该文件的后缀名确定。支持的文件类型有：man page，HTML，DocBook 以及纯文本。

--help-command-list [file]: 列出所有可用命令的清单，然后退出。

该选项列出的信息含有所有命令的名字；其中，每个命令的帮助信息可以使用 --help-command 选项后跟一个命令名字得到。如果指定了[file]参数，帮助信息会写到 file 中，输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-commands [file]: 打印所有命令的帮助文件，然后退出。

显示所有当前版本的命令的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-compatcommands [file]: 打印兼容性命令（过时的命令—译注）的帮助信息。

显示所有关于兼容性命令的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-module module [file]: 打印某单一模块的帮助信息，然后退出。

打印关于给定模块的完整信息。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-module-list [file]: 列出所有可用模块名，然后退出。

列出的清单包括所有模块的名字；其中，每个模块的帮助信息可以使用--help-module 选项，后跟模块名的方式得到。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-modules [file]: 打印所有模块的帮助信息，然后退出。

显示关于所有模块的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-custom-modules [file]: 打印所有自定义模块名，然后退出。

显示所有自定义模块的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-policy cmp [file]: 打印单个策略的帮助信息，然后退出。

显示给定的策略的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-policies [file]: 打印所有策略的帮助信息，然后退出。

显示所有策略的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-property prop [file]: 打印单个属性的帮助信息，然后退出。

显示指定属性的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-property-list [file]: 列出所有可用的属性，然后退出。

该命令列出的清单包括所有属性的名字；其中，每个属性的帮助信息都可以通过 --help-property 选项后跟一个属性名的方式获得。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-properties [file]: 打印所有属性的帮助信息，然后退出。

显示所有属性的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-variable var [file]: 打印单个变量的帮助信息，然后退出。

显示指定变量的完整文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-variable-list [file]: 列出文档中有记录的变量，然后退出。

该命令列出的清单包括所有变量的名字；其中，每个变量的帮助信息都可以通过 --help-variable 选项后跟一个变量名的方式获得。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--help-variables [file]: 打印所有变量的帮助信息，然后退出。

显示所有变量的完整帮助文档。如果指定了[file]参数，帮助信息会写到 file 中，且输出格式依赖于文件名后缀。支持的文件格式包括：man page，HTML，DocBook 以及纯文本。

--copyright [file]: 打印 CMake 的版权信息，然后退出。

如果指定了[file]参数，版权信息会写到这个文件中。

--help: 打印用法信息，然后退出。

用法信息描述了基本的命令行界面及其选项。

--help-full [file]: 打印完整的帮助信息，然后退出。

显示大多数 UNIX man page 提供的帮助信息。该选项是为非 UNIX 平台提供的；但是如果 man 手册页没有安装，它也能提供便利。如果制定了[file]参数，帮助信息会写到这个文件中。

--help-html [file]: 以 HTML 格式打印完整的帮助信息，然后退出。

CMake 的作者使用该选来帮助生成 web 页面。如果指定了[file]参数，帮助信息会写到这个文件中。

--help-man [file]: 以 UNIX 的 man 手册页格式打印完整的帮助信息，然后退出。

cmake 使用该选生成 UNIX 的 man 手册页。如果指定了[file]参数，帮助信息会写到这个文件中。

--version [file]: 显示程序名/版本信息行，然后退出。

如果指定了[file]参数，版本信息会写到这个文件中。

==========

对于 CMake 的语言要素，比如命令，属性和变量，帮助命令选项也是很有规律的，一般是用 --help-xxx-list 查看所有值的名字，找出感兴趣的项，然后用--help-xxx name 查看该名字的详细信息；也可以用--help-xxxs 获得相关语言要素的完整帮助信息。

生成器

生成器这一节确实没有很多料，纯粹的流水账；不过为了完整，也还是给它一点篇幅吧。下一章将开始我们真正的主题:-)

==========

在 CMake 2.8.3 平台上，CMake 支持下列生成器：

Borland Makefiles: 生成 Borland makefile。

MSYS Makefiles: 生成 MSYS makefile。

生成的 makefile 用 use /bin/sh 作为它的 shell。在运行 CMake 的机器上需要安装 msys。

MinGW Makefiles: 生成供 mingw32-make 使用的 make file。

生成的 makefile 使用 cmd.exe 作为它的 shell。生成它们不需要 msys 或者 unix shell。

NMake Makefiles: 生成 NMake makefile。

NMake Makefiles JOM: 生成 JOM makefile。

Unix Makefiles: 生成标准的 UNIX makefile。

在构建树上生成分层的 UNIX makefile。任何标准的 UNIX 风格的 make 程序都可以通过默认的 make 目标构建工程。生成的 makefile 也提供了 install 目标。

Visual Studio 10: 生成 Visual Studio 10 工程文件。

Visual Studio 10 Win64: 生成 Visual Studio 10 Win64 工程文件。

Visual Studio 6: 生成 Visual Studio 6 工程文件。

Visual Studio 7: 生成 Visual Studio .NET 2002 工程文件。

Visual Studio 7 .NET 2003: 生成 Visual Studio .NET 2003 工程文件。

Visual Studio 8 2005: 生成 Visual Studio .NET 2005 工程文件。

Visual Studio 8 2005 Win64: 生成 Visual Studio .NET 2005 Win64 工程文件。

Visual Studio 9 2008: 生成 Visual Studio 9 2008 工程文件。

Visual Studio 9 2008 Win64: 生成 Visual Studio 9 2008 Win64 工程文件。

Watcom WMake: 生成 Watcom WMake makefiles。

CodeBlocks - MinGW Makefiles: 生成 CodeBlock 工程文件。

> 在顶层目录以及每层子目录下为 CodeBlocks 生成工程文件，生成的 CMakeList.txt 的特点是都包含一个 PROJECT()调用。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

CodeBlocks - NMake Makefiles: 生成 CodeBlocks 工程文件。

> 在顶层目录以及每层子目录下为 CodeBlocks 生成工程文件，生成的 CMakeList.txt 的特点是都包含一个 PROJECT()调用。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

CodeBlocks - Unix Makefiles: 生成 CodeBlocks 工程文件。

> 在顶层目录以及每层子目录下为 CodeBlocks 生成工程文件，生成的 CMakeList.txt 的特点是都包含一个 PROJECT()调用。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

Eclipse CDT4 - MinGW Makefiles: 生成 Eclipse CDT 4.0 工程文件。

> 在顶层目录下为 Eclipse 生成工程文件。在运行源码外构建时，一个连接到顶层源码路径的资源文件会被创建。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

Eclipse CDT4 - NMake Makefiles: 生成 Eclipse CDT 4.0 工程文件。

在顶层目录下为 Eclipse 生成工程文件。在运行源码外构建时，一个连接到顶层源码路径的资源文件会被创建。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

Eclipse CDT4 - Unix Makefiles: 生成 Eclipse CDT 4.0 工程文件。

在顶层目录下为 Eclipse 生成工程文件。在运行源码外构建时，一个连接到顶层源码路径的资源文件会被创建。除此之外还会在构建树上生成一套层次性的 makefile。通过默认的 make 目标，正确的 make 程序可以构建这个工程。makefile 还提供了 install 目标。

二、CMake 命令

CMake 手册的客套话总算说完了，开始进入正题。第一部分是 CMake 命令。命令就相当于命令行下操作系统提供的各种命令，重要性不言而喻；可以说，这些命令是 CMake 构建系统的骨架。CMake 2.8.3 共有 80 条命令，分别是：add_custom_command, add_custom_target, add_definitions, add_dependencies, add_executable, add_library, add_subdirectory, add_test, aux_source_directory, break, build_command, cmake_minimum_required, cmake_policy, configure_file, create_test_sourcelist, define_property, else, elseif, enable_language, enable_testing, endforeach, endfunction, endif, endmacro, endwhile, execute_process, export, file, find_file, find_library, find_package, find_path, find_program, fltk_wrap_ui, foreach, function, get_cmake_property, get_directory_property, get_filename_component, get_property, get_source_file_property, get_target_property, get_test_property, if, include, include_directories, include_external_msproject, include_regular_expression, install, link_directories, list, load_cache, load_command, macro, mark_as_advanced, math, message, option, output_required_files, project, qt_wrap_cpp, qt_wrap_ui, remove_definitions, return, separate_arguments, set, set_directory_properties, set_property, set_source_files_properties, set_target_properties, set_tests_properties, site_name, source_group, string, target_link_libraries, try_compile, try_run, unset, variable_watch, while。这些命令在手册中是字典序排列的；为了便于查找，翻译也按照字典序来组织。但是在翻译结束后，会对命令进行小结，与大家讨论一下这些命令的使用方法和使用时机。

==========

CMD#1 : add_custom_command

为生成的构建系统添加一条自定义的构建规则。

add_custom_command 命令有两种主要的功能；第一种是为了生成输出文件，添加一条自定义命令。

```
add_custom_command(OUTPUT output1 [output2 ...]
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [MAIN_DEPENDENCY depend]
                   [DEPENDS [depends...]]
                   [IMPLICIT_DEPENDS <lang1> depend1 ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment] [VERBATIM] [APPEND])
```

这种命令格式定义了一条生成指定的文件（文件组）的生成命令。在相同路径下创建的目标（CMakeLists.txt 文件）——任何自定义命令的输出都作为它的源文件——被设置了一条规则：在构建的时候，使用指定的命令来生成这些文件。如果一个输出文件名是相对路径，它将被解释成相对于构建树路径的相对路径，并且与当前源码路径是对应的。注意，MAIN_DEPENDENCY 完全是可选的，它用来向 visual studio 建议在何处停止自定义命令。对于各种类型的 makefile 而言，这条命令创建了一个格式如下的新目标：

```
OUTPUT: MAIN_DEPENDENCY DEPENDS
      COMMAND
```

如果指定了多于一条的命令，它们会按顺序执行。ARGS 参数是可选的，它的存在是为了保持向后兼容，以后会被忽略掉。

第二种格式为一个目标——比如一个库文件或者可执行文件——添加一条自定义命令。这种格式可以用于目标构建前或构建后的一些操作。这条命令会成为目标的一部分，并且只有目标被构建时才会执行。如果目标已经构建了，该目标将不会执行。

```
add_custom_command(TARGET target
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment] [VERBATIM])
```

这条命令定义了一个与指定目标的构建过程相关的新命令。新命令在何时执行，由下述的选项决定：

```
PRE_BUILD  - 在所有其它的依赖之前执行；
PRE_LINK   - 在所有其它的依赖之后执行；
POST_BUILD - 在目标被构建之后执行；
```

注意，只有 Visual Studio 7 或更高的版本才支持 PRE_BUILD。对于其他的生成器，PRE_BUILD 会被当做 PRE_LINK 来对待。

如果指定了 WORKING_DIRECTORY 选项，这条命令会在给定的路径下执行。如果设置了 COMMENT 选项，后跟的参数会在构建时、以构建信息的形式、在命令执行之前显示出来。

如果指定了 APPEND 选项，COMMAND 以及 DEPENDS 选项的值会附加到第一个输出文件的自定义命令上。在此之前，必须有一次以相同的输出文件作为参数的对该命令的调用。在当前版本下，如果指定了 APPEND 选项，COMMENT, WORKING_DIRECTORY 和 MAIN_DEPENDENCY 选项会被忽略掉，不过未来有可能会用到。

如果指定了 VERBATIM 选项，所有该命令的参数将会合适地被转义，以便构建工具能够以原汁原味的参数去调用那些构建命令。注意，在 add_custom_command 能看到这些参数之前，CMake 语言处理器会对这些参数做一层转义处理。推荐使用 VERBATIM 参数，因为它能够保证正确的行为。当 VERBATIM 未指定时，CMake 的行为依赖于平台，因为 CMake 没有针对某一种工具的特殊字符采取保护措施。

如果自定义命令的输出并不是实际的磁盘文件，应该使用 SET_SOURCE_FILES_PROPERTIES 命令将该输出的属性标记为 SYMBOLIC。

IMPLICIT_DEPENDS 选项请求扫描一个输入文件的隐含依赖关系。给定的语言参数（文中的 lang1—译注）指定了应该使用哪种编程语言的依赖扫描器。目前为止，仅支持 C 和 CXX 语言扫描器。扫描中发现的依赖文件将会在编译时添加到自定义命令中。注意，IMPLICIT_DEPENDS 选项目前仅仅直至 Makefile 生成器，其它的生成器会忽略之。

如果 COMMAND 选项指定了一个可执行目标(由 ADD_EXECUTABLE 命令创建的目标)，在构建时，它会自动被可执行文件的位置所替换。而且，一个目标级的依赖性将会被添加进去，这样这个可执行目标将会在所有依赖于该自定义命令的结果的目标之前被构建。不过，任何时候重编译这个可执行文件，这种特性并不会引入一个会引起自定义命令重新运行的文件级依赖。

DEPENDS 选项指定了该命令依赖的文件。如果依赖的对象是同一目录（CMakeLists.txt 文件）下另外一个自定义命令的输出，CMake 会自动将其它自定义命令带到这个命令中来。如果 DEPENDS 指定了任何类型的目标（由 ADD_*命令创建），一个目标级的依赖性将会被创建,以保证该目标在任何其它目标使用这个自定义命令的输出之前,该目标已经被创建了。而且，如果该目标是可执行文件或库文件，一个文件级依赖将会被创建，用来引发自定义命令在目标被重编译时的重新运行。

==========

在 Unix Makefile 中，这条命令相当于增加了一个依赖关系和一条显式生成命令。

CMD#2: add_custom_target

添加一个目标，它没有输出；这样它就总是会被构建。

```
add_custom_target(Name [ALL] [command1 [args1...]]
                  [COMMAND command2 [args2...] ...]
                  [DEPENDS depend depend depend ... ]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment] [VERBATIM]
                  [SOURCES src1 [src2...]])
```

用 Name 选项给定的名字添加一个目标，这个目标会引发给定的那些命令。这个目标没有输出文件，并且总是被认为是过时的，即使那些命令试图去创建一个与该目标同名的文件。使用 ADD_CUSTOM_COMMAND 命令可以生成一个带有依赖性的文件。默认情况下，没有目标会依赖于自定义目标。使用 ADD_DEPENDENCIES 命令可以添加依赖于该目标或者被该目标依赖的目标。如果指定了 ALL 选项，这表明这个目标应该被添加到默认的构建目标中，这样它每次都会被构建（命令的名字不能是 ALL）。命令和选项是可选的；如果它们没有被指定，将会产生一个空目标。如果设定了 WORKING_DIRECTORY 参数，该命令会在它指定的路径下执行。如果指定了 COMMENT 选项，后跟的参数将会在构件的时候，在命令执行之前，被显示出来。DEPENDS 选项后面列出来的依赖目标可以引用 add_custom_command 命令在相同路径下（CMakeLists.txt）生成的输出和文件。

如果指定了 VERBATIM 选项，所有传递到该命令的选项将会被合适地转义；这样，该命令调用的构建工具会接收到未经改变的参数。注意，CMake 语言处理器会在 add_custom_target 命令在看到这些参数之前对它们进行一层转义。推荐使用该参数，因为它保证了正确的行为。当未指定该参数时，转义的行为依赖于平台，因为 CMake 没有针对于特定工具中特殊字符的保护措施。

SOURCES 选项指定了会被包含到自定义目标中的附加的源文件。指定的源文件将会被添加到 IDE 的工程文件中，方便在没有构建规则的情况下能够编辑。

CMD#3:add_definitions

为源文件的编译添加由-D 引入的 define flag。

```
add_definitions(-DFOO -DBAR ...)
```

在编译器的命令行上，为当前路径以及下层路径的源文件加入一些 define flag。这个命令可以用来引入任何 flag，但是它的原意是用来引入预处理器的定义。那些以-D 或/D 开头的、看起来像预处理器定义的 flag，会被自动加到当前路径的 COMPILE_DEFINITIONS 属性中。为了后向兼容，非简单值（non-trival，指的是什么？——译注）的定义会被留在 flags 组（flags set）里，而不会被转换。关于在特定的域以及配置中增加预处理器的定义，参考路径、目标以及源文件的 COMPILE_DEFINITIONS 属性来获取更多的细节。

CMD#4:add_dependencies

为顶层目标引入一个依赖关系。

```
add_dependencies(target-name depend-target1
                 depend-target2 ...)
```

让一个顶层目标依赖于其他的顶层目标。一个顶层目标是由命令 ADD_EXECUTABLE，ADD_LIBRARY，或者 ADD_CUSTOM_TARGET 产生的目标。为这些命令的输出引入依赖性可以保证某个目标在其他的目标之前被构建。查看 ADD_CUSTOM_TARGET 和 ADD_CUSTOM_COMMAND 命令的 DEPENDS 选项，可以了解如何根据自定义规则引入文件级的依赖性。查看 SET_SOURCE_FILES_PROPERTIES 命令的 OBJECT_DEPENDS 选项，可以了解如何为目标文件引入文件级的依赖性。

CMD#5:add_executable:

使用给定的源文件，为工程引入一个可执行文件。

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               source1 source2 ... sourceN)
```

引入一个名为<name>的可执行目标，该目标会由调用该命令时在源文件列表中指定的源文件来构建。<name>对应于逻辑目标名字，并且在工程范围内必须是全局唯一的。被构建的可执行目标的实际文件名将根据具体的本地平台创建出来（比如<name>.exe 或者仅仅是<name>）。

默认情况下，可执行文件将会在构建树的路径下被创建，对应于该命令被调用的源文件树的路径。如果要改变这个位置，查看 RUNTIME_OUTPUT_DIRECTORY 目标属性的相关文档。如果要改变最终文件名的<name>部分，查看 OUTPUT_NAME 目标属性的相关文档。

如果指定了 MACOSX_BUNDLE 选项，对应的属性会附加在创建的目标上。查看 MACOSX_BUNDLE 目标属性的文档可以找到更多的细节。

如果指定了 EXCLUDE_FROM_ALL 选项，对应的属性将会设置在被创建的目标上。查看 EXCLUDE_FROM_ALL 目标属性的文档可以找到更多的细节。

使用下述格式，add_executable 命令也可以用来创建导入的（IMPORTED）可执行目标：

```
add_executable(<name> IMPORTED)
```

一个导入的可执行目标引用了一个位于工程之外的可执行文件。该格式不会生成构建这个目标的规则。该目标名字的作用域在它被创建的路径以及底层路径有效。它可以像在该工程内的其他任意目标一样被引用。导入可执行文件为类似于 add_custom_command 之类的命令引用它提供了便利。

关于导入的可执行文件的细节可以通过设置以 IMPORTED_开头的属性来指定。这类属性中最重要的是 IMPORTED_LOCATION（以及它对应于具体配置的版本 IMPORTED_LOCATION_<CONFIG>）；该属性指定了执行文件主文件在磁盘上的位置。查看 IMPORTED_*属性的文档来获得更多信息。

CMD#6:add_library

使用指定的源文件向工程中添加一个库。

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            source1 source2 ... sourceN)
```

添加一个名为<name>的库文件，该库文件将会根据调用的命令里列出的源文件来创建。<name>对应于逻辑目标名称，而且在一个工程的全局域内必须是唯一的。待构建的库文件的

实际文件名根据对应平台的命名约定来构造（比如 lib<name>.a 或者<name>.lib）。指定 STATIC，SHARED，或者 MODULE 参数用来指定要创建的库的类型。STATIC 库是目标文件的归档文件，在链接其它目标的时候使用。SHARED 库会被动态链接，在运行时被加载。MODULE 库是不会被链接到其它目标中的插件，但是可能会在运行时使用 dlopen-系列的函数动态链接。如果没有类型被显式指定，这个选项将会根据变量 BUILD_SHARED_LIBS 的当前值是否为真决定是 STATIC 还是 SHARED。

默认状态下，库文件将会在于源文件目录树的构建目录树的位置被创建，该命令也会在这里被调用。查阅 ARCHIVE_OUTPUT_DIRECTORY，LIBRARY_OUTPUT_DIRECTORY，和 RUNTIME_OUTPUT_DIRECTORY 这三个目标属性的文档来改变这一位置。查阅 OUTPUT_NAME 目标属性的文档来改变最终文件名的<name>部分。

如果指定了 EXCLUDE_FROM_ALL 属性，对应的一些属性会在目标被创建时被设置。查阅 EXCLUDE_FROM_ALL 的文档来获取该属性的细节。

使用下述格式，add_library 命令也可以用来创建导入的库目标：

```
add_library(<name> <SHARED|STATIC|MODULE|UNKNOWN> IMPORTED)
```

导入的库目标是引用了在工程外的一个库文件的目标。没有生成构建这个库的规则。这个目标名字的作用域在它被创建的路径及以下有效。他可以向任何在该工程内构建的目标一样被引用。导入库为类似于 target_link_libraries 命令中引用它提供了便利。关于导入库细节可以通过指定那些以 IMPORTED_的属性设置来指定。其中最重要的属性是 IMPORTED_LOCATION（以及它的具体配置版本，IMPORTED_LOCATION_<CONFIG>），它指定了主库文件在磁盘上的位置。查阅 IMPORTED_*属性的文档获取更多的信息。

CMD#7:add_subdirectory

为构建添加一个子路径。

```
add_subdirectory(source_dir [binary_dir]
                 [EXCLUDE_FROM_ALL])
```

这条命令的作用是为构建添加一个子路径。source_dir 选项指定了 CMakeLists.txt 源文件和代码文件的位置。如果 source_dir 是一个相对路径，那么 source_dir 选项会被解释为相对于当前的目录，但是它也可以是一个绝对路径。binary_dir 选项指定了输出文件的路径。如果 binary_dir 是相对路径，它将会被解释为相对于当前输出路径，但是它也可以是一个绝对路径。如果没有指定 binary_dir，binary_dir 的值将会是没有做任何相对路径展开的 source_dir，这也是通常的用法。在 source_dir 指定路径下的 CMakeLists.txt 将会在当前输入文件的处理过程执行到该命令之前，立即被 CMake 处理。

如果指定了 EXCLUDE_FROM_ALL 选项，在子路径下的目标默认不会被包含到父路径的 ALL 目标里，并且也会被排除在 IDE 工程文件之外。用户必须显式构建在子路径下的目标，比如一些示范性的例子工程就是这样。典型地，子路径应该包含它自己的 project()命令调用，这样会在子路径下产生一份完整的构建系统（比如 VS IDE 的 solution 文件）。注意，目标间的

依赖性要高于这种排除行为。如果一个被父工程构建的目标依赖于在这个子路径下的目标，被依赖的目标会被包含到父工程的构建系统中，以满足依赖性的要求。

CMD#8:add_test

以指定的参数为工程添加一个测试。

```
add_test(testname Exename arg1 arg2 ... )
```

如果已经运行过了 ENABLE_TESTING 命令，这个命令将为当前路径添加一个测试目标。如果 ENABLE_TESTING 还没有运行过，该命令啥事都不做。测试是由测试子系统运行的，它会以指定的参数执行 Exename 文件。Exename 或者是由该工程构建的可执行文件，也可以是系统上自带的任意可执行文件（比如 tclsh）。该测试会在 CMakeList.txt 文件的当前工作路径下运行，这个路径与二进制树上的路相对应。

```
add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
         COMMAND <command> [arg1 [arg2 ...]])
```

如果 COMMAND 选项指定了一个可执行目标（用 add_executable 创建），它会自动被在构建时创建的可执行文件所替换。如果指定了 CONFIGURATIONS 选项，那么该测试只有在列出的某一个配置下才会运行。

在 COMMAND 选项后的参数可以使用"生成器表达式"，它的语法是"$<...>"。这些表达式会在构建系统生成期间，以及构建配置的专有信息的产生期间被评估。合法的表达式是：

```
$<CONFIGURATION>           = 配置名称
$<TARGET_FILE:tgt>         = 主要的二进制文件(.exe, .so.1.2, .a)
$<TARGET_LINKER_FILE:tgt> = 用于链接的文件(.a, .lib, .so)
$<TARGET_SONAME_FILE:tgt> = 带有.so.的文件(.so.3)
```

其中，"tgt"是目标的名称。目标文件表达式 TARGET_FILE 生成了一个完整的路径，但是它的_DIR 和_NAME 版本可以生成目录以及文件名部分：

```
$<TARGET_FILE_DIR:tgt>/$<TARGET_FILE_NAME:tgt>
$<TARGET_LINKER_FILE_DIR:tgt>/$<TARGET_LINKER_FILE_NAME:tgt>
$<TARGET_SONAME_FILE_DIR:tgt>/$<TARGET_SONAME_FILE_NAME:tgt>
```

用例：

```
1  add_test(NAME mytest
2           COMMAND testDriver --config $<CONFIGURATION>
3                              --exe $<TARGET_FILE:myexe>)
```

这段代码创建了一个名为 mytest 的测试，它执行的命令是 testDriver 工具，传递的参数包括配置名，以及由目标生成的可执行文件 myexe 的完整路径。

CMD#9:aux_source_directory

查找在某个路径下的所有源文件。

```
aux_source_directory(<dir> <variable>)
```

搜集所有在指定路径下的源文件的文件名，将输出结果列表储存在指定的<variable>变量中。该命令主要用在那些使用显式模板实例化的工程上。模板实例化文件可以存储在 Templates 子目录下，然后可以使用这条命令自动收集起来；这样可以避免手工罗列所有的实例。

使用该命令来避免为一个库或可执行目标写源文件的清单，是非常具有吸引力的。但是如果该命令貌似可以发挥作用，那么 CMake 就不需要生成一个感知新的源文件何时被加进来的构建系统了（也就是说，新文件的加入，并不会导致 CMakeLists.txt 过时，从而不能引起 CMake 重新运行。——译注）。正常情况下，生成的构建系统能够感知它何时需要重新运行 CMake，因为需要修改 CMakeLists.txt 来引入一个新的源文件。当源文件仅仅是加到了该路径下，但是没有修改这个 CMakeLists.txt 文件，使用者只能手动重新运行 CMake 来产生一个包含这个新文件的构建系统。

CMD#10:break

从一个包围该命令的 foreach 或 while 循环中跳出。

```
break()
```

从包围它的 foreach 循环或 while 循环中跳出。

CMD#11:build_command

获取构建该工程的命令行。

```
build_command(<variable>
              [CONFIGURATION <config>]
              [PROJECT_NAME <projname>]
              [TARGET <target>])
```

把给定的变量<variable>设置成一个字符串，其中包含使用由变量 CMAKE_GENERATOR 确定的项目构建工具，去构建某一个工程的某一个目标配置的命令行。

对于多配置生成器，如果忽略 CONFIGURATION 选项，CMake 将会选择一个合理的默认值；而对于单配置生成器，该选项会被忽略。

如果 PROJECT_NAME 选项被忽略，得到的命令行用来构建当前构建树上的顶层工程。

如果 TARGET 选项被忽略，得到的命令行可以用来构建所有目标，比较高效的用法是构建目标 all 或者 ALL_BUILD。

build_command(<cachevariable> <makecommand>)

不推荐使用以上的这种格式,但对于后相兼容还是有用的。只要可以,就要使用第一种格式。

这种格式将变量<cachevariable>设置为一个字符串，其中包含从构建树的根目录，用<makecommand>指定的构建工具构建这个工程的命令。<makecommand>应该是指向msdev，devenv，nmake，make 或者是一种最终用户指定的构建工具的完整路径。

CMD#12:cmake_minimum_required

设置一个工程所需要的最低 CMake 版本。

```
cmake_minimum_required(VERSION major[.minor[.patch[.tweak]]]
                       [FATAL_ERROR])
```

如果 CMake 的当前版本低于指定的版本，它会停止处理工程文件，并报告错误。当指定的版本高于 2.4 时，它会隐含调用：

```
cmake_policy(VERSION major[.minor[.patch[.tweak]]])
```

从而将 cmale 的策略版本级别设置为指定的版本。当指定的版本是 2.4 或更低时，这条命令隐含调用：

```
cmake_policy(VERSION 2.4)
```

这将会启用对于 CMake 2.4 及更低版本的兼容性。

FATAL_ERROR 选项是可以接受的，但是 CMake 2.6 及更高的版本会忽略它。如果它被指定，那么 CMake 2.4 及更低版本将会以错误告终而非仅仅给出个警告。

CMD#13:cmake_policy

管理 CMake 的策略设置。

随着 CMake 的演变，有时为了搞定 bug 或改善现有特色的实现方法，改变现有的行为是必须的。CMake 的策略机制是在新的 CMake 版本带来行为上的改变时，用来帮助保持现有项目的构建的一种设计。每个新的策略（行为改变）被赋予一个"CMP<NNNN>"格式的识别符，其中"<NNNN>"是一个整数索引。每个策略相关的文档都会描述"旧行为"和"新行为"，以及引入该策略的原因。工程可以设置各种策略来选择期望的行为。当 CMake 需要了解要用哪种行为的时候，它会检查由工程指定的一种设置。如果没有可用的设置，工程假定使用"旧行为"，并且会给出警告要求你设置工程的策略。

cmake_policy 是用来设置"新行为"或"旧行为"的命令。如果支持单独设置策略，我们鼓励各项目根据 CMake 的版本来设置策略。

```
cmake_policy(VERSION major.minor[.patch[.tweak]])
```

上述命令指定当前的 CMakeLists.txt 是为给定版本的 CMake 书写的。所有在指定的版本或更早的版本中引入的策略会被设置为使用"新行为"。所有在指定的版本之后引入的策略将会变为无效（unset）。该命令有效地为一个指定的 CMake 版本请求优先采用的行为，并且告知更新的 CMake 版本给出关于它们新策略的警告。命令中指定的策略版本必须至少是 2.4，否则命令会报告一个错误。为了得到支持早于 2.4 版本的兼容性特性，查阅策略 CMP0001 的相关文档。

```
cmake_policy(SET CMP<NNNN> NEW)

cmake_policy(SET CMP<NNNN> OLD)
```

对于某种给定的策略，该命令要求 CMake 使用新的或者旧的行为。对于一个指定的策略，那些依赖于旧行为的工程，通过设置策略的状态为 OLD，可以禁止策略的警告。或者，用户可以让工程采用新行为，并且设置策略的状态为 NEW。

```
cmake_policy(GET CMP<NNNN> <variable>)
```

该命令检查一个给定的策略是否设置为旧行为或新行为。如果策略被设置，输出的变量值会是"OLD"或"NEW"，否则为空。

CMake 将策略设置保存在一个栈结构中，因此，cmake_policy 命令产生的改变仅仅影响在栈顶端的元素。在策略栈中的一个新条目由各子路径自动管理，以此保护它的父路径及同层路径的策略设置。CMake 也管理通过 include()和 find_package()命令加载的脚本中新加入的条目，除非调用时指定了 NO_POLICY_SCOPE 选项（另外可参考 CMP0011）。cmake_policy 命令提供了一种管理策略栈中自定义条目的接口：

```
cmake_policy(PUSH)
cmake_policy(POP)
```

每个 PUSH 必须有一个配对的 POP 来去掉撤销改变。这对于临时改变策略设置比较有用。

函数和宏会在它们被创建的时候记录策略设置，并且在它们被调用的时候使用记录前的策略。如果函数或者宏实现设置了策略，这个变化会通过调用者(caller)一直上传，自动传递到嵌套的最近的策略栈条目。

CMD#14:configure_file:

将一份文件拷贝到另一个位置并修改它的内容。

```
configure_file(<input> <output>
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY])
```

将文件<input>拷贝到<output>然后替换文件内容中引用到的变量值。如果<input>是相对路径，它被评估的基础路径是当前源码路径。<input>必须是一个文件，而不是个路径。如果<output>是一个相对路径，它被评估的基础路径是当前二进制文件路径。如果<output>是一个已有的路径，那么输入文件将会以它原来的名字放到那个路径下。

该命令替换掉在输入文件中，以${VAR}格式或@VAR@格式引用的任意变量，如同它们的值是由 CMake 确定的一样。 如果一个变量还未定义，它会被替换为空。如果指定了 COPYONLY 选项，那么变量就不会展开。如果指定了 ESCAPE_QUOTES 选项，那么所有被替

换的变量将会按照 C 语言的规则被转义。该文件将会以 CMake 变量的当前值被配置。如果指定了@ONLY 选项，只有@VAR@格式的变量会被替换而${VAR}格式的变量则会被忽略。这对于配置使用${VAR}格式的脚本文件比较有用。任何类似于#cmakedefine VAR 的定义语句将会被替换为#define VAR 或者/* #undef VAR */，视 CMake 中对 VAR 变量的设置而定。任何类似于#cmakedefine01 VAR 的定义语句将会被替换为#define VAR 1 或#define VAR 0，视 VAR 被评估为 TRUE 或 FALSE 而定。

（configure_file 的作用是让普通文件也能使用 CMake 中的变量。——译注）

CMD#15:create_test_sourcelist:

为构建测试程序创建一个测试驱动器和源码列表。

```
create_test_sourcelist(sourceListName driverName
                       test1 test2 test3
                       EXTRA_INCLUDE include.h
                       FUNCTION function)
```

测试驱动器是一个将很多小的测试代码连接为一个单一的可执行文件的程序。这在为了缩减总的需用空间而用很多大的库文件去构建静态可执行文件的时候，特别有用。构建测试驱动所需要的源文件列表会在变量 sourceListName 中。DriverName 变量是测试驱动器的名字。其它的参数还包括一个测试源代码文件的清单，中间可以用分号隔开。每个测试源码文件中应该有一个与去掉扩展名的文件名同名的函数（比如 foo.cxx 文件里应该有 int foo(int, char*[]);）（和 main 的函数签名一样——译注）。DriverName 可以在命令行中按名字调用这些测试中的每一个。如果指定了 EXTRA_INCLUDE，那么它后面的参数（即 include.h——译注）会被包含到生成的文件里。如果指定了 FUNCTION 选项，那么它后面的参数（即 function——译注）会被认为是一个函数名，传递给它的参数是一个指向 argc 的指针和 argv。这个选项可以用来为每个测试函数添加额外的命令行参数处理过程。CMake 变量 CMAKE_TESTDRIVER_BEFORE_TESTMAIN 用来设置在调用测试的 main 函数之前调用的代码。

CMD#16:define_property:

定义并描述（Document）自定义属性。

```
define_property(<GLOBAL | DIRECTORY | TARGET | SOURCE |
                 TEST | VARIABLE | CACHED_VARIABLE>
                 PROPERTY <name> [INHERITED]
                 BRIEF_DOCS <brief-doc> [docs...]
                 FULL_DOCS <full-doc> [docs...])
```

在一个域（scope）中定义一个可以用 set_property 和 get_property 命令访问的属性。这个命令对于把文档和可以通过 get_property 命令得到的属性名称关联起来非常有用。第一个参数确定了这个属性可以使用的范围。它必须是下列值中的一个：

GLOBAL = 与全局命名空间相关联
DIRECTORY = 与某一个目录相关联
TARGET = 与一个目标相关联
SOURCE = 与一个源文件相关联
TEST = 与一个以 add_test 命名的测试相关联
VARIABLE = 描述（document）一个 CMake 语言变量
CACHED_VARIABLE = 描述（document）一个 CMake 语言缓存变量

注意，与 set_property 和 get_property 不相同，不需要给出实际的作用域；只有作用域的类型才是重要的。PROPERTY 选项必须有，它后面紧跟要定义的属性名。如果指定了 INHERITED 选项，那么如果 get_property 命令所请求的属性在该作用域中未设置，它会沿着链条向更高的作用域去搜索。DIRECTORY 域向上是 GLOBAL。TARGET，SOURCE 和 TEST 向上是 DIRECTORY。

BRIEF_DOCS 和 FULL_DOCS 选项后面的参数是和属性相关联的字符串，分别作为变量的简单描述和完整描述。在使用 get_property 命令时，对应的选项可以获取这些描述信息。

CMD#17: else

开始一个 if 语句块的 else 部分。

```
else(expression)
```

参见 if 命令。

CMD#18: elseif

开始 if 块的 elseif 部分。

```
elseif(expression)
```

参见 if 命令。

CMD#19: enable_language

支持某种语言（CXX/C/Fortran/等）

```
enable_language(languageName [OPTIONAL] )
```

该命令打开了 CMake 对参数中指定的语言的支持。这与 project 命令相同，但是不会创建任何 project 命令会产生的额外变量。可以选用的语言的类型有 CXX，C，Fortran 等。如果指定了 OPTIONAL 选项，用 CMAKE_<languageName>_COMPILER_WORKS 变量来判断该语言是否被成功支持。

CMD#20: enable_testing

打开当前及以下目录中的测试功能。

```
enable_testing()
```

为当前及其下级目录打开测试功能。也可参见 add_test 命令。注意，ctest 需要在构建跟目录下找到一个测试文件。因此，这个命令应该在源文件目录的根目录下。

CMD#21: endforeach

结束 foreach 语句块中的一系列命令。

```
endforeach(expression)
```

参见 FOREACH 命令。

CMD#22: endfunction

结束一个 function 语句块中的一系列命令。

```
endfunction(expression)
```

参见 function 命令。

CMD#23: endif

结束一个 if 语句块中的一系列命令。

```
endif(expression)
```

参见 if 命令。

CMD#24: endmacro

结束一个 macro 语句块中的一系列命令。

```
endmacro(expression)
```

参见 macro 命令。

CMD#25: endwhile

结束一个 while 语句块中的一系列命令。

```
endwhile(expression)
```

参见 while 命令。

CMD#26: execute_process

执行一个或更多个子进程。

```
execute_process(COMMAND <cmd1> [args1...]]
                [COMMAND <cmd2> [args2...] [...]]
                [WORKING_DIRECTORY <directory>]
                [TIMEOUT <seconds>]
                [RESULT_VARIABLE <variable>]
                [OUTPUT_VARIABLE <variable>]
                [ERROR_VARIABLE <variable>]
                [INPUT_FILE <file>]
                [OUTPUT_FILE <file>]
                [ERROR_FILE <file>]
                [OUTPUT_QUIET]
                [ERROR_QUIET]
                [OUTPUT_STRIP_TRAILING_WHITESPACE]
                [ERROR_STRIP_TRAILING_WHITESPACE])
```

运行一条或多条命令，使得前一条命令的标准输出以管道的方式成为下一条命令的标准输入。所有进程公用一个单独的标准错误管道。如果指定了 WORKING_DIRECTORY 选项，后面的路径选项将会设置为子进程的当前工作路径。如果指定了 TIMEOUT 选项，如果子进程没有在指定的秒数（允许分数）里完成，子进程会自动终止。如果指定了 RESULT_VARIABLE 选项，该变量将保存为正在运行的进程的结果；它可以是最后一个子进程的整数返回代码，也可以是一个描述错误状态的字符串。如果指定了 OUTPUT_VARIABLE 或者 ERROR_VARIABLE，后面的变量将会被分别设置为标准输出和标准错误管道的值。如果两个管道都是用了相同的变量，它们的输出将会按产生的顺序被合并。如果指定了 INPUT_FILE，OUTPUT_FILE 或 ERROR_FILE 选项，其后的文件将会分别被附加到第一个进程的标准输入、最后一个进程的标准输出，或者所有进程的标准错误管道上。如果指定了 OUTPUT_QUIET 后者 ERROR_QUIET 选项，那么标准输出或标准错误的结果将会被静静的忽略掉。如果为同一个管道指定了多于一个的 OUTPUT_*或 ERROR_* 选项，优先级是没有指定的。如果没有指定 OUTPUT_*或者 ERROR_*选项，输出将会与 CMake 进程自身对应的管道共享。

execute_process 命令是 exec_program 命令的一个较新的功能更加强大的版本。但是为了兼容性的原因，旧的 exec_program 命令还会继续保留。

CMD#27:export

从构建树中导出目标供外部使用。

```
export(TARGETS [target1 [target2 [...]]] [NAMESPACE <namespace>]
       [APPEND] FILE <filename>)
```

创建一个名为<filename>的文件，它可以被外部工程包含进去，从而外部工程可以从当前工程的构建树中导入目标。这对于交叉编译那些可以运行在宿主平台的的 utility 可执行文件，

然后将它们导入到另外一个编译成目标平台代码的工程中的情形，特别有用。如果指定了 NAMESPACE 选项，<namespace>字符串将会被扩展到输出文件中的所有目标的名字中。如果指定了 APPEND 选项，生成的代码将会续接在文件之后，而不是覆盖它。如果一个库目标被包含在 export 中，但是连接成它的目标没有被包含，行为没有指定。

由该命令创建的文件是与指定的构建树一致的，并且绝对不应该被安装。要从一个安装树上导出目标，参见 install(EXPORT)命令。

export(PACKAGE <name>)

在 CMake 的用户包注册表中，为<name>包(package)存储当前的构建目录。这将有助于依赖于它的工程从当前工程的构建树中查找并使用包而不需要用户的介入。注意，该命令在包注册表中创建的条目，仅仅在与跟构建树一起运行的包配置文件(<name>Config.cmake)一起使用时才会起作用。

CMD#28: file

文件操作命令

```
file(WRITE filename "message to write"... )
file(APPEND filename "message to write"... )
file(READ filename variable [LIMIT numBytes] [OFFSET offset] [HEX])
file(STRINGS filename variable [LIMIT_COUNT num]
     [LIMIT_INPUT numBytes] [LIMIT_OUTPUT numBytes]
     [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUM numBytes]
     [NEWLINE_CONSUME] [REGEX regex]
     [NO_HEX_CONVERSION])
file(GLOB variable [RELATIVE path] [globbing expressions]...)
file(GLOB_RECURSE variable [RELATIVE path]
     [FOLLOW_SYMLINKS] [globbing expressions]...)
file(RENAME <oldname> <newname>)
file(REMOVE [file1 ...])
file(REMOVE_RECURSE [file1 ...])
file(MAKE_DIRECTORY [directory1 directory2 ...])
file(RELATIVE_PATH variable directory file)
file(TO_CMAKE_PATH path result)
file(TO_NATIVE_PATH path result)
file(DOWNLOAD url file [TIMEOUT timeout] [STATUS status] [LOG log]
     [EXPECTED_MD5 sum] [SHOW_PROGRESS])
```

WRITE 选项将会写一条消息到名为 filename 的文件中。如果文件已经存在，该命令会覆盖已有的文件；如果文件不存在，它将创建该文件。

APPEND 选项和 WRITE 选项一样，将会写一条消息到名为 filename 的文件中，只是该消息会附加到文件末尾。

READ 选项将会读一个文件中的内容并将其存储在变量里。读文件的位置从 offset 开始，最多读 numBytes 个字节。如果指定了 HEX 参数，二进制代码将会转换为十六进制表达方式，并存储在变量里。

STRINGS 将会从一个文件中将一个 ASCII 字符串的 list 解析出来，然后存储在 variable 变量中。文件中的二进制数据会被忽略。回车换行符会被忽略。它也可以用在 Intel 的 Hex 和 Motorola 的 S-记录文件；读取它们时，它们会被自动转换为二进制格式。可以使用 NO_HEX_CONVERSION 选项禁止这项功能。LIMIT_COUNT 选项设定了返回的字符串的最大数量。LIMIT_INPUT 设置了从输入文件中读取的最大字节数。LIMIT_OUTPUT 设置了在输出变量中存储的最大字节数。LENGTH_MINIMUM 设置了要返回的字符串的最小长度；小于该长度的字符串会被忽略。LENGTH_MAXIMUM 设置了返回字符串的最大长度；更长的字符串会被分割成不长于最大长度的字符串。NEWLINE_CONSUME 选项允许新行被包含到字符串中，而不是终止它们。REGEX 选项指定了一个待返回的字符串必须满足的正则表达式。典型的使用方式是：

```
file(STRINGS myfile.txt myfile)
```

该命令在变量 myfile 中存储了一个 list，该 list 中每个项是输入文件中的一行文本。

GLOB 选项将会为所有匹配查询表达式的文件生成一个文件 list，并将该 list 存储进变量 variable 里。文件名查询表达式与正则表达式类似，只不过更加简单。如果为一个表达式指定了 RELATIVE 标志，返回的结果将会是相对于给定路径的相对路径。文件名查询表达式的例子有：

```
*.cxx       - 匹配所有扩展名为 cxx 的文件。
*.vt?       - 匹配所有扩展名是 vta,...,vtz 的文件。
f[3-5].txt - 匹配文件 f3.txt, f4.txt, f5.txt。
```

GLOB_RECURSE 选项将会生成一个类似于通常的 GLOB 选项的 list，只是它会寻访所有那些匹配目录的子路径并同时匹配查询表达式的文件。作为符号链接的子路径只有在给定 FOLLOW_SYMLINKS 选项或者 cmake 策略 CMP0009 被设置为 NEW 时，才会被寻访到。参见 cmake --help-policy CMP0009 查询跟多有用的信息。

使用递归查询的例子有：

```
/dir/*.py  - 匹配所有在/dir 及其子目录下的 python 文件。
```

MAKE_DIRECTORY 选项将会创建指定的目录，如果它们的父目录不存在时，同样也会创建。（类似于 mkdir 命令——译注）

RENAME 选项对同一个文件系统下的一个文件或目录重命名。(类似于 mv 命令——译注）

REMOVE 选项将会删除指定的文件，包括在子路径下的文件。（类似于 rm 命令——译注）

REMOVE_RECURSE 选项会删除给定的文件以及目录，包括非空目录。（类似于 rm -r 命令——译注）

RELATIVE_PATH 选项会确定从 direcroty 参数到指定文件的相对路径。

TO_CMAKE_PATH 选项会把 path 转换为一个以 unix 的 / 开头的 cmake 风格的路径。输入可以是一个单一的路径，也可以是一个系统路径，比如"$ENV{PATH}"。注意，在调用 TO_CMAKE_PATH 的 ENV 周围的双引号只能有一个参数(Note the double quotes around

the ENV call TO_CMAKE_PATH only takes one argument. 原文如此。quotes 和后面的 takes 让人后纠结，这句话翻译可能有误。欢迎指正——译注)。

TO_NATIVE_PATH 选项与 TO_CMAKE_PATH 选项很相似，但是它会把 cmake 风格的路径转换为本地路径风格：windows 下用\，而 unix 下用/。

DOWNLOAD 将给定的 URL 下载到指定的文件中。如果指定了 LOG var 选项，下载日志将会被输出到 var 中。如果指定了 STATUS var 选项，下载操作的状态会被输出到 var 中。该状态返回值是一个长度为 2 的 list。list 的第一个元素是操作的数字返回值，第二个返回值是错误的字符串值。错误信息如果是数字 0，操作中没有发生错误。如果指定了 TIMEOUT time 选项，在 time 秒之后，操作会超时退出；time 应该是整数。如果指定了 EXPECTED_MD5 sum 选项，下载操作会认证下载的文件的实际 MD5 和是否与期望值匹配。如果不匹配，操作将返回一个错误。如果指定了 SHOW_PROGRESS 选项，进度信息会以状态信息的形式被打印出来，直到操作完成。

file 命令还提供了 COPY 和 INSTALL 两种格式：

```
file(<COPY|INSTALL> files... DESTINATION <dir>
     [FILE_PERMISSIONS permissions...]
     [DIRECTORY_PERMISSIONS permissions...]
     [NO_SOURCE_PERMISSIONS] [USE_SOURCE_PERMISSIONS]
     [FILES_MATCHING]
     [[PATTERN <pattern> | REGEX <regex>]
      [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

COPY 版本把文件、目录以及符号连接拷贝到一个目标文件夹。相对输入路径的评估是基于当前的源代码目录进行的，相对目标路径的评估是基于当前的构建目录进行的。复制过程将保留输入文件的时间戳；并且如果目标路径处存在同名同时间戳的文件，复制命令会把它优化掉。赋值过程将保留输入文件的访问权限，除非显式指定权限或指定 NO_SOURCE_PERMISSIONS 选项（默认是 USE_SOURCE_PERMISSIONS）。参见 install(DIRECTORY)命令中关于权限（permissions），PATTERN，REGEX 和 EXCLUDE 选项的文档。

INSTALL 版本与 COPY 版本只有十分微小的差别：它会打印状态信息，并且默认使用 NO_SOURCE_PERMISSIONS 选项。install 命令生成的安装脚本使用这个版本（它会使用一些没有在文档中涉及的内部使用的选项。）

CMD#29:find_file

查找一个文件的完整路径。

```
find_file(<VAR> name1 [path1 path2 ...])
```

这是该命令的精简格式,对于大多数场合它都足够了。它与命令 `find_file(<VAR> name1 [PATHS path1 path2 ...])`是等价的。

```
find_file(
          <VAR>
          name | NAMES name1 [name2 ...]
```

```
  [HINTS path1 [path2 ... ENV var]]
  [PATHS path1 [path2 ... ENV var]]
  [PATH_SUFFIXES suffix1 [suffix2 ...]]
  [DOC "cache documentation string"]
  [NO_DEFAULT_PATH]
  [NO_CMAKE_ENVIRONMENT_PATH]
  [NO_CMAKE_PATH]
  [NO_SYSTEM_ENVIRONMENT_PATH]
  [NO_CMAKE_SYSTEM_PATH]
  [CMAKE_FIND_ROOT_PATH_BOTH |
   ONLY_CMAKE_FIND_ROOT_PATH |
   NO_CMAKE_FIND_ROOT_PATH]
)
```

这条命令用来查找指定文件的完整路径。一个名字是<VAR>的缓存条目（参见 CMakeCache.txt 的介绍——译注）变量会被创建，用来存储该命令的结果。如果发现了文件的一个完整路径，该结果会被存储到该变量里并且搜索过程不会再重复，除非该变量被清除。如果什么都没发现，搜索的结果将会是<VAR>-NOTFOUND；并且在下一次以相同的变量调用 find_file 时，该搜索会重新尝试。被搜索的文件的文件名由 NAMES 选项后的名字列表指定。附加的其他搜索位置可以在 PATHS 选项之后指定。如果 ENV var 在 HINTS 或 PATHS 段中出现，环境变量 var 将会被读取然后被转换为一个系统级环境变量，并存储在一个 cmake 风格的路径 list 中。比如，使用 ENV PATH 将会将系统的 path 变量列出来。在 DOC 之后的变量将会用于 cache 中的文档字符串（documentation string）。PATH_SUFFIXES 指定了在每个搜索路径下的需要搜索的子路径。

如果指定了 NO_DEFAULT_PATH 选项，那么在搜索时不会附加其它路径。如果没有指定 NO_DEFAULT_PATH 选项，搜索过程如下：

1、在 cmake 特有的 cache 变量中指定的搜索路径搜索。这些路径用于在命令行里用 -DVAR=value 被设置。如果使用了 NO_CMAKE_PATH 选项，该路径会被跳过。（此句翻译可能有误——译注。）搜索路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的路径<prefix>，<prefix>/include
变量：CMAKE_INCLUDE_PATH
变量：CMAKE_FRAMEWORK_PATH

2、在 cmake 特定的环境变量中指定的搜索路径搜索。该路径会在用户的 shell 配置中被设置。如果指定了 NO_CMAKE_ENVIRONMENT_PATH 选项，该路径会被跳过。搜索路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的路径<prefix>，<prefix>/include
变量：CMAKE_INCLUDE_PATH
变量：CMAKE_FRAMEWORK_PATH

3、由 HINTS 选项指定的搜索路径。这些路径是由系统内省（introspection）时计算出来的路径，比如已经发现的其他项的位置所提供的痕迹。硬编码的参考路径应该使用 PATHS 选项指定。（HINTS 与 PATHS 有何不同？比后者的优先级高？有疑问。——译注）

4、搜索标准的系统环境变量。如果指定 NO_SYSTEM_ENVIRONMENT_PATH 选项，搜索路径将跳过其后的参数。搜索路径包括环境变量 PATH 个 INCLUDE。

5、查找在当前系统的平台文件中定义的 cmake 变量。如果指定了 NO_CMAKE_SYSTEM_PATH 选项，该路径会被跳过。其他的搜索路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的路径<prefix>，<prefix>/include
变量：CMAKE_SYSTEM_INCLUDE_PATH
变量：CMAKE_SYSTEM_FRAMEWORK_PATH

6、搜索由 PATHS 选项指定的路径或者在命令的简写版本中指定的路径。这一般是一些硬编码的参考路径。在 Darwin 后者支持 OS X 框架的系统上，cmake 变量 CMAKE_FIND_FRAMWORK 可以设置为空或者下述值之一：

"FIRST" - 在标准库或者头文件之前先查找框架。对于 Darwin 系统，这是默认的。
"LAST" - 在标准库或头文件之后再查找框架。
"ONLY" - 只查找框架。
"NEVER" - 从不查找框架。

在 Darwin 或者支持 OS X Application Bundles 的系统上，cmake 变量 CMAKE_FIND_APPBUNDLE 可以被设置为空，或者下列值之一：

"FIRST" - 在标准程序之前查找 application bundles，这也是 Darwin 系统的默认选项。
"LAST" - 在标准程序之后查找 application bundlesTry。
"ONLY" - 只查找 application bundles。
"NEVER" - 从不查找 application bundles。

CMake 的变量 CMAKE_FIND_ROOT_PATH 指定了一个或多个在所有其它搜索路径之前的搜索路径。该选项很有效地将给定位置下的整个搜索路径的最优先路径进行了重新指定。默认情况下，它是空的。当交叉编译一个指向目标环境下的根目录中的目标时，CMake 也会搜索那些路径；该变量这时显得非常有用。默认情况下，首先会搜索在 CMAKE_FIND_ROOT_PATH 变量中列出的路径，然后才是非根路径。设置 CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 变量可以调整该默认行为。该行为可以在每次调用时被手动覆盖。通过使用 CMAKE_FIND_ROOT_PATH_BOTH 变量，搜索顺序将会是上述的那样。如果使用了 NO_CMAKE_FIND_ROOT_PATH 变量，那么 CMAKE_FIND_ROOT_PATH 将不会被用到。如果使用了 ONLY_CMAKE_FIND_ROOT_PATH 变量，那么只有 CMAKE_FIND_ROOT_PATH 中的路径（即 re-rooted 目录——译注）会被搜索。

一般情况下，默认的搜索顺序是从最具体的路径到最不具体的路径。只要用 NO_*选项多次调用该命令，工程就可以覆盖该顺序。

```
find_file(<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_file(<VAR> NAMES name)
```

只要这些调用中的一个成功了，返回变量就会被设置并存储在 cache 中；然后该命令就不会再继续查找了。

CMD#30:find_library

查找一个库文件

```
find_library(<VAR> name1 [path1 path2 ...])
```

这是该命令的简写版本，在大多数场合下都已经够用了。它与命令 find_library(<VAR> name1 [PATHS path1 path2 ...])等价。

```
find_library(
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_CMAKE_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

该命令用来查找一个库文件。一个名为<VAR>的 cache 条目会被创建来存储该命令的结果。如果找到了该库文件，那么结果会存储在该变量里，并且搜索过程将不再重复，除非该变量被清空。如果没有找到，结果变量将会是<VAR>-NOTFOUND，并且在下次使用相同变量调用 find_library 命令时，搜索过程会再次尝试。在 NAMES 参数后列出的文件名是要被搜索的库名。附加的搜索位置在 PATHS 参数后指定。如果再 HINTS 或者 PATHS 字段中设置了 ENV 变量 var,环境变量 var 将会被读取并从系统环境变量转换为一个 cmake 风格的路径 list。例如，指定 ENV PATH 是获取系统 path 变量并将其转换为 cmake 的 list 的一种方式。在 DOC 之后的参数用来作为 cache 中的注释字符串。PATH_SUFFIXES 选项指定了每个搜索路径下待搜索的子路径。

如果指定了 NO_DEFAULT_PATH 选项，那么搜索的过程中不会有其他的附加路径。如果没有指定该选项，搜索过程如下：

1、搜索 cmake 特有的 cache 变量指定的路径。这些变量是在用 cmake 命令行时，通过 -DVAR=value 指定的变量。如果指定了 NO_CMAKE_PATH 选项，这些路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>，路径<prefix>/lib
CMAKE_LIBRARY_PATH
CMAKE_FRAMEWORK_PATH

2、搜索 cmake 特有的环境变量指定的路径。这些变量是用户的 shell 配置中设置的变量。如果指定了 NO_CMAKE_ENVIRONMENT_PATH 选项，这些路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>，路径<prefix>/lib
CMAKE_LIBRARY_PATH
CMAKE_FRAMEWORK_PATH

3、搜索由 HINTS 选项指定的路径。这些路径是系统内省（introspection）估算出的路径，比如由另一个已经发现的库文件的地址提供的参考信息。硬编码的推荐路径应该通过 PATHS 选项指定。

4、查找标准的系统环境变量。如果指定了 NO_SYSTEM_ENVIRONMENT_PATH 选项，这些路径会被跳过。搜索的路径还包括：

PATH
LIB

5、查找在为当前系统的平台文件中定义的 cmake 变量。如果指定了 NO_CMAKE_SYSTEM_PATH 选项，该路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_SYSTEM_PREFIX_PATH 中的<prefix>，路径<prefix>/lib
CMAKE_SYSTEM_LIBRARY_PATH
CMAKE_SYSTEM_FRAMEWORK_PATH

6、搜索 PATHS 选项或者精简版命令指定的路径。这些通常是硬编码的推荐搜索路径。

在 Darwin 或者支持 OS X 框架的系统上，cmake 变量 CMAKE_FIND_FRAMEWORK 可以用来设置为空，或者下述值之一：

"FIRST" - 在标准库或头文件之前查找框架。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找框架。
"ONLY" - 仅仅查找框架。
"NEVER" - 从不查找框架。

在 Darwin 或者支持 OS X Application Bundles 的系统，cmake 变量 CMAKE_FIND_APPBUNDLE 可以被设置为空或者下面这些值中的一个：

"FIRST" - 在标准库或头文件之前查找 application bundles。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找 application bundles。
"ONLY" - 仅仅查找 application bundles。
"NEVER" - 从不查找 application bundles。

CMake 变量 CMAKE_FIND_ROOT_PATH 指定了一个或者多个优先于其他搜索路径的搜索路径。该变量能够有效地重新定位在给定位置下进行搜索的根路径。该变量默认为空。当使用交叉编译时，该变量十分有用：用该变量指向目标环境的根目录，然后 CMake 将会在那里查找。默认情况下，在 CMAKE_FIND_ROOT_PATH 中列出的路径会首先被搜索，然后是"非根"路径。该默认规则可以通过设置 CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 做出调整。在每次调用该命令之前，都可以通过设置这个变量来手动覆盖默认行为。如果使用了 NO_CMAKE_FIND_ROOT_PATH 变量，那么只有重定位的路径会被搜索。

默认的搜索顺序的设计逻辑是按照使用时从最具体到最不具体。通过多次调用 find_library 命令以及 NO_*选项，可以覆盖工程的这个默认顺序：

```
find_library(<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_library(<VAR> NAMES name)
```

只要这些调用中的一个成功返回，结果变量就会被设置并且被存储到 cache 中；这样随后的调用都不会再行搜索。如果那找到的库是一个框架，VAR 将会被设置为指向框架"<完整路径>/A.framework" 的完整路径。当一个指向框架的完整路径被用作一个库文件，CMake 将使用-framework A，以及-F<完整路径>这两个选项将框架连接到目标上。

CMD#31:find_package

为外部工程加载设置。

```
find_package(<package> [version] [EXACT] [QUIET]
             [[REQUIRED|COMPONENTS] [components...]]
             [NO_POLICY_SCOPE])
```

查找并加载外来工程的设置。该命令会设置<package>_FOUND 变量，用来指示要找的包是否被找到了。如果这个包被找到了，与它相关的信息可以通过包自身记载的变量中得到。QUIET 选项将会禁掉包没有被发现时的警告信息。REQUIRED 选项表示如果报没有找到的话，cmake 的过程会终止，并输出警告信息。在 REQUIRED 选项之后，或者如果没有指定 REQUIRED 选项但是指定了 COMPONENTS 选项，在它们的后面可以列出一些与包相关的部件清单（components list）。[version]参数需要一个版本号，它是正在查找的包应该兼容的版本号（格式是 major[.minor[.patch[.tweak]]]）。EXACT 选项要求该版本号必须精确匹配。如果在 find-module 内部对该命令的递归调用没有给定[version]参数，那么[version]和 EXACT 选项会自动地从外部调用前向继承。对版本的支持目前只存在于包和包之间(详见下文)。

用户代码总体上应该使用上述的简单调用格式查询需要的包。本命令文档的剩余部分则详述了 find_package 的完整命令格式以及具体的查询过程。期望通过该命令查找并提供包的项目维护人员，我们鼓励你能继续读下去。

该命令在搜索包时有两种模式："模块"模式和"配置"模式。当该命令是通过上述的精简格式调用的时候，合用的就是模块模式。在该模式下，CMake 搜索所有名为 Find<package>.cmake 的文件，这些文件的路径由变量由安装 CMake 时指定的 CMAKE_MODULE_PATH 变量指定。如果查找到了该文件，它会被 CMake 读取并被处理。该模式对查找包，检查版本以及生成任何别的必须信息负责。许多查找模块（find-module）仅仅提供了有限的，甚至根本就没有对版本化的支持；具体信息查看该模块的文档。如果没有找到任何模块，该命令会进入配置模式继续执行。

完整的配置模式下的命令格式是：

```
find_package(<package> [version] [EXACT] [QUIET]
             [[REQUIRED|COMPONENTS] [components...]] [NO_MODULE]
             [NO_POLICY_SCOPE]
             [NAMES name1 [name2 ...]]
             [CONFIGS config1 [config2 ...]]
             [HINTS path1 [path2 ... ]]
             [PATHS path1 [path2 ... ]]
```

```
[PATH_SUFFIXES suffix1 [suffix2 ...]]
[NO_DEFAULT_PATH]
[NO_CMAKE_ENVIRONMENT_PATH]
[NO_CMAKE_PATH]
[NO_SYSTEM_ENVIRONMENT_PATH]
[NO_CMAKE_PACKAGE_REGISTRY]
[NO_CMAKE_BUILDS_PATH]
[NO_CMAKE_SYSTEM_PATH]
[CMAKE_FIND_ROOT_PATH_BOTH |
 ONLY_CMAKE_FIND_ROOT_PATH |
 NO_CMAKE_FIND_ROOT_PATH])
```

NO_MODULE 可以用来明确地跳过模块模式。它也隐含指定了不使用在精简格式中使用的那些选项。

配置模式试图查找一个由待查找的包提供的配置文件的位置。包含该文件的路径会被存储在一个名为<package>_DIR 的 cache 条目里。默认情况下，该命令搜索名为<package>的包。如果指定了 NAMES 选项，那么其后的 names 参数会取代<package>的角色。该命令会为每个在 names 中的 name 搜索名为<name>Config.cmake 或者<name 全小写>-config.cmake 的文件。通过使用 CONFIGS 选项可以改变可能的配置文件的名字。以下描述搜索的过程。如果找到了配置文件，它将会被 CMake 读取并处理。由于该文件是由包自身提供的，它已经知道包中内容的位置。配置文件的完整地址存储在 cmake 的变量<package>_CONFIG 中。

所有 CMake 要处理的配置文件将会搜索该包的安装信息，并且将该安装匹配的适当版本号（appropriate version）存储在 cmake 变量<package>_CONSIDERED_CONFIGS 中，与之相关的版本号（associated version）将被存储在<package>_CONSIDERED_VERSIONS 中。

如果没有找到包配置文件，CMake 将会生成一个错误描述文件，用来描述该问题——除非指定了 QUIET 选项。如果指定了 REQUIRED 选项，并且没有找到该包，将会报致命错误，然后配置步骤终止执行。如果设置了<package>_DIR 变量被设置了，但是它没有包含配置文件信息，那么 CMake 将会直接无视它，然后重新开始查找。

如果给定了[version]参数，那么配置模式仅仅会查找那些在命令中请求的版本（格式是 major[.minor[.patch[.tweak]]]）与包请求的版本互相兼容的那些版本的包。如果指定了 EXACT 选项，一个包只有在它请求的版本与[version]提供的版本精确匹配时才能被找到。CMake 不会对版本数的含义做任何的转换。包版本号由包自带的版本文件来检查。对于一个备选的包配置文件<config-file>.cmake，对应的版本文件的位置紧挨着它，并且名字或者是<config-file>-version.cmake 或者是<config-file>Version.cmake。如果没有这个版本文件，那么配置文件就会认为不兼容任何请求的版本。当找到一个版本文件之后，它会被加载然后用来检查(find_package)请求的版本号。版本文件在一个下述变量被定义的嵌套域中被加载：

```
PACKAGE_FIND_NAME          = <package>名字。
PACKAGE_FIND_VERSION       = 请求的完整版本字符串
PACKAGE_FIND_VERSION_MAJOR = 如果被请求了，那么它是 major 版本号，否则是 0。
PACKAGE_FIND_VERSION_MINOR = 如果被请求了，那么它是 minor 版本号，否则是 0。
```

PACKAGE_FIND_VERSION_PATCH = 如果被请求了，那么它是 patch 版本号，否则是 0。
PACKAGE_FIND_VERSION_TWEAK = 如果被请求了，那么它是 tweak 版本号，否则是 0。
PACKAGE_FIND_VERSION_COUNT = 版本号包含几部分，0 到 4。

版本文件会检查自身是否满足请求的版本号，然后设置了下面这些变量：

PACKAGE_VERSION = 提供的完整的版本字符串。
PACKAGE_VERSION_EXACT = 如果版本号精确匹配，返回 true。
PACKAGE_VERSION_COMPATIBLE = 如果版本号相兼容，返回 true。
PACKAGE_VERSION_UNSUITABLE = 如果不适合任何版本，返回 true。

下面这些变量将会被 find_package 命令检查，用以确定配置文件是否提供了可接受的版本。在 find_package 命令返回后，这些变量就不可用了。如果版本可接受，下述的变量会被设置：

<package>_VERSION = 提供的完整的版本字符串。
<package>_VERSION_MAJOR = 如果被请求了，那么它是 major 版本号，否则是 0。
<package>_VERSION_MINOR = 如果被请求了，那么它是 minor 版本号，否则是 0。
<package>_VERSION_PATCH = 如果被请求了，那么它是 patch 版本号，否则是 0。
<package>_VERSION_TWEAK = 如果被请求了，那么它是 tweak 版本号，否则是 0。
<package>_VERSION_COUNT = 版本号包含几部分，0 到 4。

然后，对应的包配置文件才会被加载。当多个包配置文件都可用时，并且这些包的版本文件都与请求的版本兼容，选择哪个包将会是不确定的。不应该假设 cmake 会选择最高版本或者是最低版本。（以上的若干段是对 find_package 中版本匹配步骤的描述，并不需要用户干预——译注。）

配置模式提供了一种高级接口和搜索步骤的接口。这些被提供的接口的大部分是为了完整性的要求，以及在模块模式下，包被 find-module 加载时供内部使用。大多数用户仅仅应该调用：

find_package(<package> [major[.minor]] [EXACT] [REQUIRED|QUIET])

来查找包。鼓励那些需要提供 CMake 包配置文件的包维护人员应该命名这些文件并安装它们，这样下述的整个过程将会找到它们而不需要使用附加的选项。

CMake 为包构造了一组可能的安装前缀。在每个前缀下，若干个目录会被搜索，用来查找配置文件。下述的表格展示了待搜索的路径。每个条目都是专门为 Windows(W)，UNIX(U) 或者 Apple(A)约定的安装树指定的。

<prefix>/ (W)
<prefix>/(cmake|CMake)/ (W)
<prefix>/<name>*/ (W)
<prefix>/<name>*/(cmake|CMake)/ (W)
<prefix>/(share|lib)/cmake/<name>*/ (U)
<prefix>/(share|lib)/<name>*/ (U)
<prefix>/(share|lib)/<name>*/(cmake|CMake)/ (U)

在支持 OS X 平台和 Application Bundles 的系统上，包含配置文件的框架或者 bundles 会在下述的路径中被搜索：

<prefix>/<name>.framework/Resources/ (A)
<prefix>/<name>.framework/Resources/CMake/ (A)
<prefix>/<name>.framework/Versions/*/Resources/ (A)
<prefix>/<name>.framework/Versions/*/Resources/CMake/ (A)
<prefix>/<name>.app/Contents/Resources/ (A)
<prefix>/<name>.app/Contents/Resources/CMake/ (A)

在所有上述情况下，<name>是区分大小写的，并且对应于在<package>或者由 NAMES 给定的任何一个名字。

这些路径集用来与那些在各自的安装树上提供了配置文件的工程协作。上述路径中被标记为(W)的是专门为 Windows 上的安装设置的，其中的<prefix>部分可能是一个应用程序的顶层安装路径。那些被标记为(U)的是专门为 UNIX 平台上的安装设置的，其中的<prefix>被多个包共用。这仅仅是个约定，因此，所有(W)和(U)路径在所有平台上都仍然会被搜索。那些被标记为(A)的路径是专门为 Apple 平台上的安装设置的。CMake 变量 CMAKE_FIND_FRAMEWORK 和 CMAKE_FIND_APPBUNDLE 确定了偏好的顺序，如下所示：

安装前缀是通过以下步骤被构建出来的。如果指定了 NO_DEFAULT_PATH 选项，所有 NO_*选项都会被激活。

1、搜索在 cmake 特有的 cache 变量中指定的搜索路径。这些变量是为了在命令行中用 -DVAR=value 选项指定而设计的。通过指定 NO_CMAKE_PATH 选项可以跳过该搜索路径。搜索路径还包括：

CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
CMAKE_APPBUNDLE_PATH

2、搜索 cmake 特有的环境变量。这些变量是为了在用户的 shell 配置中进行配置而设计的。通过指定 NO_CMAKE_ENVIRONMENT_PATH 选项可以跳过该路径。搜索的路径包括：

<package>_DIR
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
CMAKE_APPBUNDLE_PATH

3、搜索 HINTS 选项指定的路径。这些路径应该是由操作系统内省时计算产生的，比如由其它已经找到的项的位置而提供的线索。硬编码的参考路径应该在 PATHS 选项中指定。

4、搜索标准的系统环境变量。如果指定了 NO_SYSTEM_ENVIRONMENT_PATH 选项，这些路径会被跳过。以"/bin"或"/sbin"结尾的路径条目会被自动转换为它们的父路径。搜索的路径包括：

PATH

5、搜索在 CMake GUI 中最新配置过的工程的构建树。可以通过设置 NO_CMAKE_BUILDS_PATH 选项来跳过这些路径。这是为了在用户正在依次构建多个相互依赖的工程时而准备的。

6、搜索存储在 CMake 用户包注册表中的路径。通过设置 NO_CMAKE_PACKAGE_REGISTRY 选项可以跳过这些路径。当 CMake 嗲用 export(PACKAGE<name>)配置一个工程时，这些路径会被存储在注册表中。参见 export(PACKAGE)命令的文档阅读更多细节。

7、搜索在当前系统的平台文件中定义的 cmake 变量。可以用 NO_CMAKE_SYSTEM_PATH 选项跳过这些路径。

CMAKE_SYSTEM_PREFIX_PATH

CMAKE_SYSTEM_FRAMEWORK_PATH

CMAKE_SYSTEM_APPBUNDLE_PATH

8、搜索由 PATHS 选项指定的路径。这些路径一般是硬编码的参考路径。

在 Darwin 或者支持 OS X 框架的系统上，cmake 变量 CMAKE_FIND_FRAMEWORK 可以用来设置为空，或者下述值之一：

"FIRST" - 在标准库或头文件之前查找框架。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找框架。
"ONLY" - 仅仅查找框架。
"NEVER" - 从不查找框架。

在 Darwin 或者支持 OS X Application Bundles 的系统，cmake 变量 CMAKE_FIND_APPBUNDLE 可以被设置为空或者下面这些值中的一个：

"FIRST" - 在标准库或头文件之前查找 application bundles。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找 application bundles。
"ONLY" - 仅仅查找 application bundles。
"NEVER" - 从不查找 application bundles。

CMake 变量 CMAKE_FIND_ROOT_PATH 指定了一个或者多个优先于其他搜索路径的搜索路径。该变量能够有效地重新定位在给定位置下进行搜索的根路径。该变量默认为空。当使用交叉编译时，该变量十分有用：用该变量指向目标环境的根目录，然后 CMake 将会在那里查找。默认情况下，在 CMAKE_FIND_ROOT_PATH 中列出的路径会首先被搜索，然后是"非根"路径。该默认规则可以通过设置 CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 做出调整。在每次调用该命令之前，都可以通过设置这个变量来手动覆盖默认行为。如果使用了 NO_CMAKE_FIND_ROOT_PATH 变量，那么只有重定位的路径会被搜索。

默认的搜索顺序的设计逻辑是按照使用时从最具体到最不具体。通过多次调用 find_library 命令以及 NO_*选项，可以覆盖工程的这个默认顺序：

```
 find_library(<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_library(<VAR> NAMES name)
```

只要这些调用中的一个成功返回，结果变量就会被设置并且被存储到 cache 中；这样随后的调用都不会再行搜索。如果那找到的库是一个框架，VAR 将会被设置为指向框架"<完整路

径>/A.framework" 的完整路径。当一个指向框架的完整路径被用作一个库文件，CMake 将使用-framework A，以及-F<完整路径>这两个选项将框架连接到目标上。

参见 cmake_policy()命令的文档中关于 NO_POLICY_SCOPE 选项讨论。

CMD#32 : find_path

搜索包含某个文件的路径

```
find_path(<VAR> name1 [path1 path2 ...])
```

在多数情况下，使用上述的精简命令格式就足够了。它与命令 find_path(<VAR> name1 [PATHS path1 path2 ...])等价。

```
find_path(
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_CMAKE_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

该命令用于给定名字文件所在的路径。一条名为<VAR>的 cache 条目会被创建，并存储该命令的执行结果。如果在某个路径下发现了该文件，该结果会被存储到该变量中；除非该变量被清除，该次搜索不会继续进行。如果没有找到，存储的结果将会是<VAR>-NOTFOUND，并且当下一次以相同的变量名调用 find_path 命令时，该命令会再一次尝试搜索该文件。需要搜索的文件名通过在 NAMES 选项后面的列出来的参数来确定。附加的搜索位置可以在 PATHS 选项之后指定。如果在 PATHS 或者 HINTS 命令中还指定了 ENV var 选项，环境变量 var 将会被读取并从一个系统环境变量转换为一个 cmake 风格的路径 list。比如，ENV PATH 是列出系统 path 变量的一种方法。参数 DOC 将用来作为该变量在 cache 中的注释。PATH_SUFFIXES 指定了在每个搜索路径下的附加子路径。

如果指定了 NO_DEFAULT_PATH 选项，那么没有其它附加的路径会被加到搜索过程中。如果并未指定 NO_DEFAULT_PATH 选项，搜索的过程如下：

1、搜索 cmake 专有的 cache 变量中的路径。这种用法是为了在命令行中用选项 -DVAR=value 指定搜索路径。如果指定了 NO_CMAKE_PATH 选项，该路径会被跳过。搜索路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>/，路径<prefix>/include

CMAKE_INCLUDE_PATH
CMAKE_FRAMEWORK_PATH

2、搜索 cmake 专有的环境变量中指定的路径。这种用法是为了在用户的 shell 配置中设置指定的搜索路径。如果指定了 NO_CMAKE_ENVIRONMENT_PATH 选项，该路径会被跳过。搜索路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>/，路径<prefix>/include
CMAKE_INCLUDE_PATH
CMAKE_FRAMEWORK_PATH

3、搜索由 HINTS 选项指定的路径。这些路径应该是由系统内省时计算得出的路径，比如由其它已经发现的项目提供的线索。硬编码的参考路径应该在 PATHS 选项中指定。

4、搜索标准的系统环境变量。通过指定选项 NO_SYSTEM_ENVIRONMENT_PATH 可以跳过搜索环境变量。搜索的路径还包括：

PATH
INCLUDE

5、查找在为当前系统的平台文件中定义的 cmake 变量。如果指定了 NO_CMAKE_SYSTEM_PATH 选项，该路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_SYSTEM_PREFIX_PATH 中的<prefix>，路径<prefix>/include
CMAKE_SYSTEM_LIBRARY_PATH
CMAKE_SYSTEM_FRAMEWORK_PATH

6、搜索 PATHS 选项或者精简版命令指定的路径。这些通常是硬编码的推荐搜索路径。

在 Darwin 或者支持 OS X 框架的系统上，cmake 变量 CMAKE_FIND_FRAMEWORK 可以用来设置为空，或者下述值之一：

"FIRST" - 在标准库或头文件之前查找框架。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找框架。
"ONLY" - 仅仅查找框架。
"NEVER" - 从不查找框架。

在 Darwin 或者支持 OS X Application Bundles 的系统，cmake 变量 CMAKE_FIND_APPBUNDLE 可以被设置为空或者下面这些值中的一个：

"FIRST" - 在标准库或头文件之前查找 application bundles。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找 application bundles。
"ONLY" - 仅仅查找 application bundles。
"NEVER" - 从不查找 application bundles。

CMake 变量 CMAKE_FIND_ROOT_PATH 指定了一个或者多个优先于其他搜索路径的搜索路径。该变量能够有效地重新定位在给定位置下进行搜索的根路径。该变量默认为空。当使用交叉编译时，该变量十分有用：用该变量指向目标环境的根目录，然后 CMake 将会在那里查找。

默认情况下，在 CMAKE_FIND_ROOT_PATH 中列出的路径会首先被搜索，然后是"非根"路径。该默认规则可以通过设置 CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 做出调整。在每次调用该命令之前，都可以通过设置这个变量来手动覆盖默认行为。如果使用了 NO_CMAKE_FIND_ROOT_PATH 变量，那么只有重定位的路径会被搜索。

默认的搜索顺序的设计逻辑是按照使用时从最具体到最不具体的路径。通过多次调用 find_path 命令以及 NO_*选项，可以覆盖工程的这个默认顺序：

```
 find_path(<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_path(<VAR> NAMES name)
```

只要这些调用中的一个成功返回，结果变量就会被设置并且被存储到 cache 中；这样随后的调用都不会再行搜索。在搜索框架时，如果以 A/b.h 的格式指定文件，那么该框架搜索过程会搜索 A.framework/Headers/b.h。如果找到了该路径，它将会被设置为框架的路径。CMake 将把它转换为正确的-F 选项来包含该文件。

CMD#33:find_program

查找可执行程序

```
find_program(<VAR> name1 [path1 path2 ...])
```

这是该命令的精简格式，它在大多数场合下都够用了。命令 find_program(<VAR> name1 [PATHS path1 path2 ...])是它的等价形式。

```
find_program(
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_CMAKE_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

该命令用于查找程序。一个名为<VAR>的 cache 条目会被创建用来存储该命令的结果。如果该程序被找到了，结果会存储在该变量中，搜索过程将不会再重复，除非该变量被清除。如果没有找到，结果将会是<VAR>-NOTFOUND，并且下次以相同的变量调用该命令时，还会做搜索的尝试。被搜索的程序的名字由 NAMES 选项后列出的参数指定。附加的搜索位置可以在 PATHS 参数后指定。如果在 HINTS 或者 PATHS 选项后有 ENV var 参数，环境变量 var 将会

被读取并从系统环境变量转换为 cmake 风格的路径 list。比如 ENV PATH 是一种列出所有系统 path 变量的方法。DOC 后的参数将会被用作 cache 中的注释字符串。PATH_SUFFIXES 指定了在每个搜索路径下要检查的附加子路径。

如果指定了 NO_DEFAULT_PATH 选项，那么搜索的过程中不会有其他的附加路径。如果没有指定该选项，搜索过程如下：

1、搜索 cmake 特有的 cache 变量指定的路径。这些变量是在用 cmake 命令行时，通过 -DVAR=value 指定的变量。如果指定了 NO_CMAKE_PATH 选项，这些路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>，路径<prefix>/[s]bin
CMAKE_PROGRAM_PATH
CMAKE_APPBUNDLE_PATH

2、搜索 cmake 特有的环境变量指定的路径。这些变量是用户的 shell 配置中设置的变量。如果指定了 NO_CMAKE_ENVIRONMENT_PATH 选项，这些路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_PREFIX_PATH 中的<prefix>，路径<prefix>/[s]bin
CMAKE_PROGRAM_PATH
CMAKE_APPBUNDLE_PATH

3、搜索由 HINTS 选项指定的路径。这些路径是系统内省（introspection）估算出的路径，比如由另一个已经发现的程序的地址提供的参考信息。硬编码的推荐路径应该通过 PATHS 选项指定。

4、查找标准的系统环境变量。如果指定了 NO_SYSTEM_ENVIRONMENT_PATH 选项，这些路径会被跳过。搜索的路径还包括：

PATH

5、查找在为当前系统的平台文件中定义的 cmake 变量。如果指定了 NO_CMAKE_SYSTEM_PATH 选项，该路径会被跳过。搜索的路径还包括：

对于每个在 CMAKE_SYSTEM_PREFIX_PATH 中的<prefix>，路径<prefix>/[s]bin
CMAKE_SYSTEM_PROGRAM_PATH
CMAKE_SYSTEM_APPBUNDLE_PATH

6、搜索 PATHS 选项或者精简版命令指定的路径。这些通常是硬编码的推荐搜索路径。

在 Darwin 或者支持 OS X 框架的系统上，cmake 变量 CMAKE_FIND_FRAMEWORK 可以设置为空，或者下述值之一：

"FIRST" - 在标准库或头文件之前查找框架。在 Darwin 系统上这是默认选项。
"LAST" - 在标准库或头文件之后查找框架。
"ONLY" - 仅仅查找框架。
"NEVER" - 从不查找框架。

在 Darwin 或者支持 OS X Application Bundles 的系统，cmake 变量 CMAKE_FIND_APPBUNDLE 可以被设置为空或者下面这些值中的一个：

"FIRST" - 在标准程序之前查找 application bundles。在 Darwin 系统上这是默认选项。
"LAST" - 在标准程序之后查找 application bundles。
"ONLY" - 仅仅查找 application bundles。
"NEVER" - 从不查找 application bundles。

CMake 变量 CMAKE_FIND_ROOT_PATH 指定了一个或者多个优先于其他搜索路径的搜索路径。该变量能够有效地重新定位在给定位置下进行搜索的根路径。该变量默认为空。当使用交叉编译时，该变量十分有用：用该变量指向目标环境的根目录，然后 CMake 将会在那里查找。默认情况下，在 CMAKE_FIND_ROOT_PATH 中列出的路径会首先被搜索，然后是"非根"路径。该默认规则可以通过设置 CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 做出调整。在每次调用该命令之前，都可以通过设置这个变量来手动覆盖默认行为。如果使用了 NO_CMAKE_FIND_ROOT_PATH 变量，那么只有重定位的路径会被搜索。

默认的搜索顺序的设计逻辑是按照使用时从最具体到最不具体。通过多次以 NO_*选项调用 find_program 命令，可以覆盖工程的这个默认顺序：

```
find_library(<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_library(<VAR> NAMES name)
```

只要这些调用中的一个成功返回，结果变量就会被设置并且被存储到 cache 中；这样随后的调用都不会再行搜索。

CMD#34:fltk_wrap_ui

创建 FLTK 用户界面包装器。

```
fltk_wrap_ui(resultingLibraryName source1
             source2 ... sourceN )
```

为所有列出的.fl 和.fld 文件生成.h 和.cxx 文件。这些生成的.h 和.cxx 文件将会加到变量 resultingLibraryName_FLTK_UI_SRCS 中，它也会加到你的库中。

如果现在有同一个工程构建出的Debug版和Release版可执行文件projectD和projectR，如下的命令可以帮助你把Debug版中的符号表加到Release版中，实现对Release版的调试。

1、objcopy --only-keep-debug projectD projectsymbol.dbg #生成符号表；

2、gdb -q --symbol=projectsymbol.dbg -exec=projectR #加载符号表；

这下在 gdb 中就可以看到源代码了。

CMD#35 : foreach

对一个 list 中的每一个变量执行一组命令。

```
foreach(loop_var arg1 arg2 ...)
  COMMAND1(ARGS ...)
```

```
  COMMAND2(ARGS ...)
  ...
endforeach(loop_var)
```

所有的 foreach 和与之匹配的 endforeach 命令之间的命令会被记录下来而不会被调用。等到遇到 endforeach 命令时，先前被记录下来的命令列表中的每条命令都会为 list 中的每个变量调用一遍。在每次迭代中，循环变量${loop_var}将会被设置为 list 中的当前变量值。

```
foreach(loop_var RANGE total)
foreach(loop_var RANGE start stop [step])
```

foreach 命令也可以遍历一个人为生成的数据区间。遍历的方式有三种：

*如果指定了一个数字，区间是[0, total]。

*如果指定了两个数字，区间将会是第一个数字到第二个数字。

*第三个数字是从第一个数字遍历到第二个数字时的步长。

```
foreach(loop_var IN [LISTS [list1 [...]]]
                    [ITEMS [item1 [...]]])
```

该命令的含义是：精确遍历一个项组成的 list。LISTS 选项后面是需要被遍历的 list 变量的名字，包括空元素（一个空字符串是一个零长度 list）。ITEMS 选项结束了 list 参数的解析，然后在迭代中引入所有在其后出现的项。（猜测是用 list1 中的项 item1，依次类推，为循环变量赋值。——译注）

CMD#36 : function

开始记录一个函数，为以后以命令的方式调用它做准备。

```
function(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endfunction(<name>)
```

定义一个名为<name>的函数，它以 arg1 arg2 arg3 (...)为参数。在 function 之后，对应的 endfunction 之前列出的命令，在函数被调用之前，是不会被调用的。当函数被调用时，在函数中记录的那些命令首先会用传进去的参数替换掉形参（${arg1}）；然后跟正常命令一样去调用这些命令。除了形参，你还可以引用这些变量：ARGC 为传递给函数的变量个数，ARGV0 ARGV1 ARGV2 ...表示传到函数中的实参值。这些变量为编写可选参数函数提供了便利。此外，ARGV 保留了一个该函数所有实参的 list，ARGN 保留了函数形参列表以后的所有参数列表。

参见 cmake_policy()命令文档中 function 内部策略行为的相关行为。

CMD#37 : get_cmake_property

获取一个 CMake 实例的属性。

```
get_cmake_property(VAR property)
```

从指定的 CMake 实例中获取属性。属性的值存储在变量 VAR 中。如果属性不存在，CMake 会报错。一些会被支持的属性包括：VATIABLES，COMMANDS，MACROS 以及 COMPONENTS。

CMD#38 : get_directory_property

获取 DIRECTORY 域中的某种属性。

```
get_directory_property(<variable> [DIRECTORY <dir>] <prop-name>)
```

在指定的变量中存储路径（directory）域中的某种属性。如果该属性没有被定义，将会返回空字符串。DIRECTORY 参数指定了要取出的属性值的另一个路径。指定的路径必须已经被 CMake 遍历过了。

```
get_directory_property(<variable> [DIRECTORY <dir>]
                       DEFINITION <var-name>)
```

该命令从一个路径中获取一个变量的定义。这种格式在从另一个路径中获取变量的定义时比较有用。

CMD#39 : get_filename_component

得到一个完整文件名中的特定部分。

```
get_filename_component(<VAR> FileName
                       PATH|ABSOLUTE|NAME|EXT|NAME_WE|REALPATH
                       [CACHE])
```

将变量<VAR>设置为路径(PATH)，文件名(NAME)，文件扩展名(EXT)，去掉扩展名的文件名(NAME_WE)，完整路径(ABSOLUTE)，或者所有符号链接被解析出的完整路径(REALPATH)。注意，路径会被转换为 Unix 的反斜杠(/)，并且没有结尾的反斜杠。该命令已经考虑了最长的文件扩展名。如果指定了 CACHE 选项，得到的变量会被加到 cache 中。

```
get_filename_component(<VAR> FileName
                       PROGRAM [PROGRAM_ARGS <ARG_VAR>]
                       [CACHE])
```

在 FileName 中的程序将会在系统搜索路径中被查找，或者是一个完整路径。如果与 PRPGRAM 一起给定了 PROGRAM_ARGS 选项，那么任何在 Filename 字符串中出现的命令行中选项将会从程序名中分割出来并存储在变量<ARG_VAR>中。这可以用来从一个命令行字符串中分离程序名及其选项。

CMD#40 : get_property

获取一个属性值

```
get_property(<variable>
             <GLOBAL             |
              DIRECTORY [dir]    |
              TARGET    <target> |
              SOURCE    <source> |
              TEST      <test>   |
              CACHE     <entry>  |
              VARIABLE>
             PROPERTY <name>
             [SET | DEFINED | BRIEF_DOCS | FULL_DOCS])
```

获取在某个域中一个对象的某种属性值。第一个参数指定了存储属性值的变量。第二个参数确定了获取该属性的域。域的选项仅限于：

- GLOBAL 域是唯一的，它不接受域名字。
- DIRECTORY 域默认为当前目录，但是其他的路径（已经被 CMake 处理过）可以以相对路径或完整路径的方式跟在该域后面。
- TARGET 域后面必须跟有一个已有的目标名。
- SOURCE 域后面必须跟有一个源文件名。
- TEST 域后面必须跟有一个已有的测试。
- CACHE 域后面必须跟有一个 cache 条目。
- VARIABLE 域是唯一的，它不接受域名字。

PROPERTY 选项是必须的，它后面紧跟要获取的属性名。如果该属性没有被设置，该命令将返回空值。如果给定了 SET 选项，那么返回值会被设置为一个布尔值，用来指示该属性是否被设置过。如果给定了 DEFINED 选项，那么返回值会被设置为一个布尔值，用来指示该属性是否被类似于 define_property 的命令定义过。如果指定了 BRIEF_DOCS 或者 FULL_DOCS 选项，那么该变量将会被设置为被查询属性的文档的字符串。如果被请求的属性的文档没有被定义，将返回 NOTFOUND。

CMD#41 : get_source_file_property

为一个源文件获取一种属性值。

```
get_source_file_property(VAR file property)
```

从一个源文件中获取某种属性值。这个属性值存储在变量 VAR 中。如果该属性没有被找到，VAR 会被设置为 NOTFOUND。使用 set_source_files_proterties 命令来设置属性值。源文件属性通常用来控制文件如何被构建。一个必定存在的属性是 LOCATION。

CMD#42 : get_target_property

从一个目标中获取一个属性值。

```
get_target_property(VAR target property)
```

从一个目标中获取属性值。属性的值会被存储在变量 VAR 中。如果该属性没有被发现，VAR 会被设置为 NOTFOUND。使用 set_target_properties 命令来设置属性值。属性值一般用于控制如何去构建一个目标，但是有些属性用来查询目标的信息。该命令可以获取当前已经被构建好的任意目标的属性。该目标不一定存在于当前的 CMakeLists.txt 文件中。

CMD#43 : get_test_property

获取一个测试的属性。

```
get_test_property(test VAR property)
```

从指定的测试中获取某种属性。属性值会被存储到变量 VAR 中。如果没有找到该属性，CMake 将会报错。你可以使用命令 cmake --help-property-list 来获取标准属性的清单。

CMD#44 : if

条件执行一组命令。

```
if(expression)
  # then section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
elseif(expression2)
  # elseif section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
else(expression)
  # else section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endif(expression)
```

评估给定的表达式。如果结果是 true，在 THEN 段的命令就会被调用。否则，在 ELSE 区段的命令会被调用。ELSEIF 和 ELSE 区段是可选的 。可以有多个 ELSEIF 子句。注意，在 else 和 elseif 子句中的表达式也是可选的。判断条件可以用长表达式，并且表达式有约定的优先级顺序。括号中的表达式会首先被调用；然后是一元运算符，比如 EXISTS，COMMAND 以及 DEFINED；然后是 EQUAL，LESS，GREATER，STRLESS，STRGREATER，STREQUAL，MATCHES；然后是 NOT 运算符，最后是 AND，OR 运算符。几种可能的表达式是：

```
if(<常量>)
```

如果<常量>是 1，ON，YES，TRUE，Y 或者非 0 数值，那么表达式为真；如果<常量>是 0，OFF，NO，FALSE，N，IGNORE，""，或者以'-NOTFOUND'为后缀，那么表达式为假。这些布尔常量值是大小写无关的。

if(<变量>)

如果<变量>的值不是一个 false 常量，表达式为真。

if(NOT <表达式>)

如果<表达式>的值是 false 的话，真个表达式为真。

if(<表达式 1> AND <表达式 2>)

如果两个表达式都为真，整个表达式为真。

if(<表达式 1> OR <表达式 2>)

只要有一个表达式为真，整个表达式为真。

if(COMMAND command-name)

如果给出的名字是一个可以被调用的命令，宏，或者函数的话，整个表达式的值为真。

if(POLICY policy-id)

如果给出的名字是一个已有的策略（格式是 CMP<NNNN>），表达式为真。

if(TARGET 目标名)

如果给出的名字是一个已有的构建目标或导入目标的话，表达式为真。

if(EXISTS 文件名)

if(EXISTS 路径名)

如果给出的文件名或路径名存在，表达式为真。该命令只对完整路径有效。

if(file1 IS_NEWER_THAN file2)

如果 file1 比 file2 更新或者其中的一个文件不存在，那么表达式为真。该命令只对完整路径有效。

if(IS_DIRECTORY directory-name)

如果给定的名字是一个路径，表达式返回真。该命令只对完整路径有效。

if(IS_SYMLINK file-name)

如果给定的名字十一个符号链接的话，表达式返回真。该命令只对完整路径有效。

if(IS_ABSOLUTE path)

如果给定的路径是一个绝对路径的话，表达式返回真。

if(variable MATCHES regex)

if(string MATCHES regex)

如果给定的字串或变量值域给定的正则表达式匹配的话，表达式返回真。

if(variable LESS number)
if(string LESS number)
if(variable GREATER number)
if(string GREATER number)
if(variable EQUAL number)
if(string EQUAL number)

如果给定的字串或变量值是一个有效的数字并且不等号或等号满足的话，表达式返回真。

if(variable STRLESS string)
if(string STRLESS string)
if(variable STRGREATER string)
if(string STRGREATER string)
if(variable STREQUAL string)
if(string STREQUAL string)

如果给定的字串或变量值依字典序小于（或者大于，或者等于）右边给出的字串或变量值的话，表达式返回真。

if(version1 VERSION_LESS version2)
if(version1 VERSION_EQUAL version2)
if(version1 VERSION_GREATER version2)

对版本号的各部分依次比较（版本号格式是 major[.minor[.patch[.tweak]]]）version1 和 version2 的大小。

if(DEFINED variable)

如果给定的变量被定义了的话，该表达式为真。如果变量被设置了，它的值是真是假都无所谓。

if((expression) AND (expression OR (expression)))

在小括号内的表达式会首先被计算，然后才按照先前介绍的运算来计算。有内嵌的括号时，最里的括号会作为包含它们的表达式的计算过程的一部分。IF 语句在 CMake 的历史上出现的相当早，它拥有一些需要特殊介绍的便捷特性。IF 表达式只有在其中有一个单一的保留值的时候，才会精简操作（即不做变量展开——译注）；这些保留值包括：如果是大小写无关的 ON，1，YES，TRUE，Y，它返回真；如果是 OFF，0，NO，FALSE，N，NOTFOUND，*-NOTFOUND，IGNORE，它返回假。这种特性非常合理，它为新作者提供了一种不需要精确匹配 true 或者 false 的便利性。这些值会被当做变量处理，即使它们没有使用${}语法的时候，也会被解引用。这意味着，如果你写下了这样的语句：

if (boobah)

CMake 将会把它当做你写了

if (${boobah})

来处理。类似地，如果你写了

```
if (fubar AND sol)
```

CMake 将会便捷地把它解释为

```
if ("${fubar}" AND "${sol}")
```

上述两例的后者确实是正确的书写方式，但是前者也是可行的。if 语句中只有某些操作有这种特殊的变量处理方式。这些特殊的语句包括：

对于 MATCHES 运算符，待匹配的左边的参数首先被检查，用来确认它是否是一个已经定义的变量；如果是，该变量的值会被使用，否则就会用它的原始值。

如果 MATCHES 运算符没有左边的参数，它返回 false，但不产生错误。

LESS，GREATER，EQUAL 运算符的左边的参数和右边的参数会被独立测试，用来确认它们是否是被定义的变量；如果是，使用它们被定义的值，否则使用它们的原始值。

STRLESS，STRGREATER，STREQUAL 运算符的左边的参数和右边的参数会被独立测试，用来确认它们是否是被定义的变量；如果是，使用它们被定义的值，否则使用它们的原始值。

VERSIONLESS，VERSIONGREATER，VERSIONEQUAL 运算符的左边的参数和右边的参数会被独立测试，用来确认它们是否是被定义的变量；如果是，使用它们被定义的值，否则使用它们的原始值。

NOT 运算符右边的参数会被测试用来确定它是否是布尔常量，如果是，就用这个常量；否则它会被当做一个变量然后被解引用。

AND 和 OR 运算符的左边的参数和右边的参数会被独立测试，用来确认它们是否是布尔常量；如果是，就用这个常量，否则它们会被当做变量然后被解引用。

CMD#45 ： include

从给定的文件中读取 CMake 的列表文件。

```
include(<file|module> [OPTIONAL] [RESULT_VARIABLE <VAR>]
                      [NO_POLICY_SCOPE])
```

从给定的文件中读取 CMake 的清单文件代码。在清单文件中的命令会被立即处理，就像它们是写在这条 include 命令展开的地方一样。如果指定了 OPTIONAL 选项，那么如果被包含文件不存在的话，不会报错。如果指定了 RESULT_VARIABLE 选项，那么 var 或者会被设置为被包含文件的完整路径，或者是 NOTFOUND，表示没有找到该文件。

如果指定的是一个模块（module）而不是一个文件，查找的对象会变成路径 CMAKE_MODULE_PATH 下的文件<modulename>.camke。

参考 cmake_policy()命令文档中关于 NO_POLICY_SCOPE 选项的讨论。

CMD#46 : include_directories

为构建树添加包含路径。

```
include_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)
```

将给定的路径添加到编译器搜索包含文件（.h 文件）的路径列表中。缺省情况下，该路径会被附加在当前路径列表的后面。这种缺省行为可以通过设置 CMAKE_include_directories_BEFORE 变量为 ON 被改变。通过将该变量改变为 BEFORE 或 AFTER，你可以在追加和附加在前端这两种方式中选择，而不用理会缺省设置。如果指定了 SYSTEM 选项，编译器将会认为该路径是某种平台上的系统包含路径。

CMD#47 : include_external_msproject

在一个 workspace 中包含一个外部的 Microsoft 工程。

```
include_external_msproject(projectname location dep1 dep2 ...)
```

在生成的 workspace 文件中包含一个外部的 Microsoft 工程。它会创建一个名为 [projectname]的目标。这个目标可以用在 add_dependencies 命令中让其他工程依赖于这个外部工程。当前版本下，该命令在 UNIX 平台上不会做任何事情。

CMD#48 : include_regular_expression

设置用于依赖性检查的正则表达式。

```
include_regular_expression(regex_match [regex_complain])
```

设置依赖性检查的正则表达式。这有匹配正则表达式 regex_match 的文件会成为依赖性跟踪的对象。只有匹配 regex_complain 的文件，在找不到它们的时候才会给出警告（标准头文件不会被搜索）。正则表达式的默认值是：

```
regex_match    = "^.*$" (匹配所有文件)
regex_complain = "^$" (仅匹配空字符串)
```

CMD#49 : install

指定在安装时要运行的规则。

该命令为一个工程生成安装规则。在某一源文件路径中，调用这条命令所指定的规则会在安装时按顺序执行。在不同路径之间的顺序未定义。

该命令有诸多版本。其中的一些版本定义了文件以及目标的安装属性。这多个版本的公共属性都有所涉及，但是只有在指定它们的版本中，这些属性才是合法的（下面的 DESTIONATION 到 OPTIONAL 的选项列表是公共属性。——译注）。

DESTINATION 选项指定了一个文件会安装到磁盘的哪个路径下。若果给出的是全路径（以反斜杠或者驱动器名开头），它会被直接使用。如果给出的是相对路径，它会被解释为相对于 CMAKE_INSTALL_PREFIX 的值的相对路径。

PERMISSIONS 选项制定了安装文件需要的权限。合法的权限有：OWNER_READ，OWNER_WRITE，OWNER_EXECUTE，GROUP_READ，GROUP_WRITE，GROUP_EXECUTE，WORLD_READ，WORLD_WRITE，WORLD_EXECUTE，SETUID 和 SETGID。对于在某些特定的平台上没有意义的权限，在这些平台上会忽略这些选项。

CONFIGURATIONS 选项指定了该安装规则将会加诸之上的一系列的构建配置（Debug，Release，等等）。

COMPONENT 选项指定了该安装规则相关的一个安装部件的名字，比如“runtime”或“development”。对于那些指定安装部件的安装过程来说，在安装时只有与给定的部件名相关的安装规则会被执行。对于完整安装，所有部件都会被安装。

RENAME 选项为一个可能不同于原始文件的已经安装的文件指定另一个名字。重命名只有在该命令正在安装一个单一文件时才被允许（猜测是为了防止文件名冲突时覆盖掉旧文件。——译注）。

OPTIONAL 选项表示要安装的文件不存在不会导致错误。

TARGETS 版本的 install 命令

```
install(TARGETS targets... [EXPORT <export-name>]
        [[ARCHIVE|LIBRARY|RUNTIME|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [OPTIONAL] [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...])
```

TARGETS 格式的 install 命令规定了安装工程中的目标（targets）的规则。有 5 中可以被安装的目标文件：ARCHIVE，LIBRARY，RUNTIME，FRAMEWORK，和 BUNDLE。除了被标记为 MACOSX_BUNDLE 属性的可执行文件被当做 OS X 上的 BUNDLE 目标外，其他的可执行文件都被当做 RUNTIME 目标。静态链接的库文件总是被当做 ARCHIVE 目标。模块库总是被当做 LIBRARY 目标。对于动态库不是 DLL 格式的平台来说，动态库会被当做 LIBRARY 目标来对待，被标记为 FRAMEWORK 的动态库是例外，它们被当做 OS X 上的 FRAMEWORK 目标。对于 DLL 平台而言，动态库的 DLL 部分被当做一个 RUNTIME 目标而对应的导出库被当做是一个 ARCHIVE 目标。所有基于 Windows 的系统，包括 Cygwin，都是 DLL 平台。ARCHIVE，LIBRARY，RUNTIME 和 FRAMEWORK 参数改变了后续属性会加诸之上的目标的类型。如果只给出了一种类型，那么只有那种类型的目标会被安装（这样通常只会安装一个 DLL 或者一个导出库。）

PRIVATE_HEADER，PUBLIC_HEADER，和 RESOURCE 选项的功能是，在非苹果平台上，将后续的属性应用在待安装的一个 FRAMEWORK 共享库目标的相关文件上。这些选项定义的规则在苹果系统上会被忽略掉，因为相关的文件将会被安装到 framework 文件

夹内的合适位置。参见 PRIVATE_HEADER，PUBLIC_HEADER 和 RESOURCE 目标属性中更为详细的解释。

可以指定 NAMELINK_ONLY 或者 NAMELINK_SKIP 选项作为 LIBRARY 选项。在一些平台上，版本化的共享库有一个符号链接，比如 lib<name>.so -> lib<name>.so.1，其中"lib<name>.so.1"是 so 库文件名（soname）而"lib<name>.so"是一个符号链接，当指定"-l<name>"选项时，链接器将会查找这个符号链接。如果一个库目标已经被安装，NAMELINK_ONLY 选项表示仅仅安装符号链接；而 NAME_SKIP 选项则表示仅仅安装库文件而不是符号链接。当两种选项都没有给出时，动态库的两个部分都会被安装。在那些版本化的共享库没有符号链接或者库没有被版本化的平台，选项 NAMELINK_SKIP 安装这个库，而 NAMELINK_ONLY 选项什么都不会安装。参见 VERSION 和 SOVERSION 目标属性，获取关于创建版本化共享库的更多细节。

在该命令的 TARGETS 版本的一次调用中，可以一次性指定一个或多个属性组。一个目标也可以被多次安装到不同的位置。假设有三个目标 myExe，mySharedLib 和 myStaticLib，下面的代码

```
install(TARGETS myExe mySharedLib myStaticLib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib/static)
install(TARGETS mySharedLib DESTINATION /some/full/path)
```

将会把 myExe 安装到<prefix>/bin 目录下，把 myStaticLib 安装到<prefix>/lib/static 目录下。在非-DLL 平台上，mySharedLib 将会被安装到<prefix>/lib 和/some/full/path 下。在 DLL 平台上，mySharedLib DLL 将会被安装到<prefix>/bin 和/some/full/path 路径下，它的导出库会被安装到<prefix>/lib/static 和/some/full/path 路径下。

EXPORT 选项将已经安装的目标文件和一个名为<export-name>的导出文件关联起来。它必须出现在所有 RUNTIME，LIBRARY 或者 ARCHIVE 选项之前。为了实际安装导出文件本身（export file），调用 install(EXPORT)。参见下述 install 命令 EXPORT 版本的文档获取更多的细节。

将 EXCLUDE_FROM_ALL 设置为 true 时，安装一个目标会造成未定义的行为。

FILES 版本的 install 命令

```
install(FILES files... DESTINATION <dir>
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>]
        [RENAME <name>] [OPTIONAL])
```

FILES 版本的 install 命令指定了为一个工程安装文件的规则。在命令中，以相对路径方式给出的文件名是相对于当前源代码路径而言的。以这个版本安装的文件，如果没有指定 PERMISSIONS 选项，默认会具有 OWNER_WRITE，OWNER_READ，GROUP_READ，和 WORLD_READ 的权限。

PROGRAMS 版本的 install 命令

```
install(PROGRAMS files... DESTINATION <dir>
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>]
        [RENAME <name>] [OPTIONAL])
```

PROGRAMS 版本与 FILES 版本一样，只在默认权限上有所不同：它还包括了 OWNER_EXECUTE，GROUP_EXECUTE 和 WORLD_EXECUTE 选项。INSTALL 的这个版本用来安装不是目标的程序，比如 shell 脚本。使用 TARGETS 格式安装该工程内部构建的目标。

DIRECTORY 版本的 install 命令

```
install(DIRECTORY dirs... DESTINATION <dir>
        [FILE_PERMISSIONS permissions...]
        [DIRECTORY_PERMISSIONS permissions...]
        [USE_SOURCE_PERMISSIONS] [OPTIONAL]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>] [FILES_MATCHING]
        [[PATTERN <pattern> | REGEX <regex>]
         [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

INSTALL 的 DIRECTORY 版本将一个或者多个路径下的内容安装到指定的目标地址下。目录结构会原封不动地（verbatim）拷贝到目标地址。每个路径名的最后一部分会追加到目标路径下，但是结尾反斜杠（trailing slash）可以用来避免这一点，因为这样最后一部分就是空的。给定的相对路径名被解释成相对于当前源路径的路径。如果没有指定输入目录名字，目标目录会被创建，但是不会安装任何东西。FILE_PERMISSIONS 和 DIRECTORY_PERMISSIONS 选项指定了赋予目标路径和目标文件的权限。如果指定了 USE_SOURCE_PERMISSIONS 选项，但没有指定 FILE_PERMISSIONS 选项，文件权限将沿袭源目录结构的权限,而且这个路径会被赋予 PAROGRAMS 版本中指定的默认权限。

通过使用 PATTERN 或 REGEX 选项可以对路径安装做出细粒度的控制。这些用于匹配的选项指定了一个查询模式或正则表达式来匹配输入路径内的路径或文件。它们可以用来将特定的选项（见下文）加诸于遇到的文件和路径的一个子集上。每个输入文件或路径的完整路径（反斜杠/开头的路径）将用来匹配该表达式。PATTERN 仅仅用来匹配完全文件名：匹配该模式的全路径的那部分必须出现在文件名的结尾，并且必须以一个反斜杠开始。

正则表达式会用来匹配一个完全路径的任何部分，但是它也可以使用'/'和'$'模仿 PATTERN 的行为。默认情况下，所有文件和路径不管是否匹配都会被安装。可以在第一个匹配选项之前指定 FILE_MATCHING 选项，这样就能禁止安装那些不与任何表达式匹配的文件。比如，代码

```
install(DIRECTORY src/ DESTINATION include/myproj
        FILES_MATCHING PATTERN "*.h")
```

将会精确匹配并安装从源码树上得到的头文件。

有些选项后面可以跟在 PATTERN 或者 REGEX 表达式的后面，这样这些选项只能加诸于匹配 PATTERN/REGEX 的文件或路径上。EXCLUDE 选项将会指示安装过程跳过那些匹

配的文件或者路径。PERMISSIONS 选项可以覆盖那些匹配 PATTERN/REGEX 的文件的权限设定。例如，代码

```
install(DIRECTORY icons scripts/ DESTINATION share/myproj
        PATTERN "CVS" EXCLUDE
        PATTERN "scripts/*"
        PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
                    GROUP_EXECUTE GROUP_READ)
```

会将 icons 路径安装到 share/myproject/icons 下，同时把 scripts 目录安装到 share/myproj 路径下。icons 将具备默认的文件权限，scripts 将会被给与指定的权限，但是所有 CVS 路径排除在外。

SCRIPT 和 CODE 版本的 install 命令

```
install([[SCRIPT <file>] [CODE <code>]] [...])
```

SCRIPT 格式将会在安装期调用给定的脚本文件。如果脚本文件名是一个相对路径，它会被解释为相对于当前的源路径。CODE 格式将会在安装期调用给定的 CMake 代码。code 被指定为一个双引号括起来的单独的参数。例如，代码

```
install(CODE "MESSAGE(\"Sample install message.\")")
```

会在安装时打印一条消息。

EXPORT 版本的 install 命令

```
install(EXPORT <export-name> DESTINATION <dir>
        [NAMESPACE <namespace>] [FILE <name>.cmake]
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>])
```

EXPORT 格式的 install 命令生成并安装一个包含将安装过程的安装树导入到另一个工程中的 CMake 文件。Target 格式的安装过程与上文提及的使用 EXPORT 选项的 install(TARGET ...)格式的命令中的 EXPORT <export-name>选项是相关的。NAMESPACE 选项会在它们被写入到导入文件时加到目标名字之前。缺省时，生成的文件就是<export-name>.cmake；但是 FILE 选项可以用来指定不同于次的文件名。FILE 选项后面的参数必须是一".cmake"为扩展名的文件。如果指定了 CONFIGURATIONS 选项，那么只有那些具名的配置中的一个被安装时，这个文件才会被安装。而且，生成的导入文件只能涉及到匹配的目标配置版本。如果指定了一个 COMPONENT 选项，并且<component>与那个<export-name>相关的目标指定的部件不匹配，那么行为是未定义的。如果一个库目标被包含在 export 之中，但是与之关联的库却没有背包含，那么结果是未指定的。

EXPORT 格式可以协助外部工程使用当前工程构建出来并安装的目标。例如，代码

```
install(TARGETS myexe EXPORT myproj DESTINATION bin)
install(EXPORT myproj NAMESPACE mp_ DESTINATION lib/myproj)
```

将会把可执行文件 myexe 安装到<prefix>/bin 下，并且将导入它的代码写到文件"<prefix>/lib/myproj/myproj.cmake"中。一个外部工程可以用 include 命令加载这个文件，并且可以在安装树上使用导入的目标名 mp_myexe（前缀_目标名——译注）引用 myexe 可执行文件，如同这个目标是它自身的构建树的内置目标一样。

注意：这个命令会取代 INSTALL_TARGETS 命令以及 PRE_INSTALL_SCRIPT 和 POST_INSTALL_SCRIPT 两个目标属性。它也可以取代 FILES 格式的 INSTALL_FILES 命令和 INSTALL_PROGRAMS 命令。由 INSTALL 命令生成的安装规则相对于那些由 INSTALL_TARGETS，INSTALL_FILES 和 INSTALL_PROGRAMS 命令生成的安装规则处理顺序是未定义的。

CMD#50 ： link_directories 指定连接器查找库的路径。

```
link_directories(directory1 directory2 ...)
```

指定连接器搜索库文件时的路径。该命令仅仅能用在那些在它被调用后才生成的目标上。由于历史上的原因，为该命令指定的相对路径将会不加改变地传递给连接器（不像许多其他 CMake 命令那样解释为相对于当前源路径的相对路径。）

CMD#51: list

列表操作命令。

```
list(LENGTH <list> <output variable>)
list(GET <list> <element index> [<element index> ...] <output variable>)
list(APPEND <list> <element> [<element> ...])
list(FIND <list> <value> <output variable>)
list(INSERT <list> <element_index> <element> [<element> ...])
list(REMOVE_ITEM <list> <value> [<value> ...])
list(REMOVE_AT <list> <index> [<index> ...])
list(REMOVE_DUPLICATES <list>)
list(REVERSE <list>)
list(SORT <list>)
```

使用 LENGTH 选项时，该命令会返回给定 list 的长度。

使用 GET 选项时，该命令返回 list 中所有被 index 索引的元素构成的 list。

使用 APPEND 选项时，该命令将会在该 list 之后追加若干元素。

使用 FIND 选项时，该命令将返回 list 中指定的元素的索引；若果未找到，返回-1。

使用 INSERT 选项时，该命令将在 list 中指定的位置插入若干元素。

使用 REMOVE_AT 和 REMOVE_ITEM 选项将会从 list 中删除一些元素。它们之间的区别是：REMOVE_ITEM 删除的是指定的项，而 REMOVE_AT 删除的是在指定索引处的项。

使用 REMOVE_DUPLICATES 选项时，该命令将删除 list 中的重复项。

使用 REVERSE 选项时，该命令将把 list 的内容就地前后倒换。

使用 SORT 选项时，该命令将按字母序对 list 总的内容就地排序。

注意：在 CMake 中，一个 list 是一个由封号;分割的一组字符串。使用 set 命令可以创建一个 list。例如，set(var a b c d e)命令将会创建一个 list：a;b;c;d;e；而 set(var "a b c d e")命令创建的只是一个字符串，或者说是只有一个项的 list。

当使用指定索引的命令格式时，如果<element index>是大于等于 0 的数，<element index>是从 list 第一个项开始的序号，list 的第一项的索引是 0。如果<element index>小于等于-1，这个索引是从结尾开始的逆向索引，其中-1 表示的是 list 的最后一项。当使用负数索引时，注意它们不是从 0 开始！-0 与 0 等价，它指向 list 的第一个成员。

CMD#52：load_cache

从另一个工程的 CMake cache 中加载值。

```
load_cache(pathToCacheFile READ_WITH_PREFIX
      prefix entry1...)
```

该命令读取指定的 cache 文件，并将以请求的前缀为其前缀的那些 cache 文件中的 entry(ies)保存到变量中。这个格式仅仅读取值，但是不在本地工程的 cache 中创建 entry(ies)。

```
load_cache(pathToCacheFile [EXCLUDE entry1...]
      [INCLUDE_INTERNALS entry1...])
```

从另一个 cache 文件中加载值并像内部 entry(ies)那样，将它们存储到本地工程的 cache 中。这条命令对于一个依赖于另一个不同构建树上的另一个工程的工程比较有用。EXCLUDE 选项给出了那些需要排除在外的 entry(ies)的一个 list。INCLUDE_INTERNALS 选项给出了需要包含的 entry(ies)的内部 entry(ies)的一个 list。通常情况下，不需要引入内部 entry(ies)。强烈不推荐使用该命令的这种格式，但是它可以被用来维持向后兼容性。

CMD#53：load_command

将一条命令加载到一个运行中的 CMake。

```
load_command(COMMAND_NAME <loc1> [loc2 ...])
```

该命令将在给定的路径下查找名字为 cmCOMMAND_NAME 的一个库。如果找到了，它将会以模块的方式被加载，然后该命令将会被添加到可用的 CMake 命令集中。通常，TRY_COMPILE 选项被用在这个命令之前来编译这个模块。如果该命令被成功加载，一个名为CMAKE_LOADED_COMMAND_<COMMAND_NAME>的变量将会被设置为这个加载模块的完整路径。否则，这个变量就不会被设置。

CMD#54：macro

为后续以命令方式调用而开始记录一组宏。

```
macro(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endmacro(<name>)
```

定义一个名为<name>的宏，它以 arg1 arg2 arg3 (...)为参数。在 macro 命令之后，在与之配对的 endmacro 命令之前出现的命令，只有在宏被调用的时候才会被调用。当被调用的时候，这些被记录的命令首先以传进来的实参替换掉形参(如${arg1})，然后像正常的命令那样执行。除了形参之外，你还可以引用变量${ARGC}，它表示传递到宏里的参数的数量；${ARG0}, ${ARG1}, ${ARG2} ...等等则是传进来的实参值。这些变量使得创建带可选参数的宏变得很便捷。此外，变量${ARGV}保留了所有传递到宏里的所有参数组成的一个 list，变量${ARGN}保留了在最后一个形参之后的参数组成的一个 list。注意：传递到宏内部的参数和值，比如 ARGN 不是 CMake 通常意义下的变量；它们只是字符串替换，这一点非常像 C 预处理器对 C 语言宏的处理过程。如果你想要用真正的 CMake 变量，你应该查看一下 function 命令的说明。

关于在 macro 内部的策略的行为，参见 cmake_policy()命令的相关文档。

CMD#55：mark_as_advanced

将 CMake 的缓存变量标记为高级。

```
mark_as_advanced([CLEAR|FORCE] VAR VAR2 VAR...)
```

将缓存的变量标记为高级变量。其中，高级变量指的是那些在 cmake GUI 中，只有当“显示高级选项”被打开时才会被显示的变量。如果 CLEAR 是第一个选项，参数中的高级变量将变回非高级变量。如果 FORCE 是第一个选项，参数中的变量会被提升为高级变量。如果两者都未出现，新的变量会被标记为高级变量；如果这个变量已经是高级/非高级状态的话，它将会维持原状。

该命令在脚本中无效。

CMD#56：math

数学表达式。

```
math(EXPR <output variable> <math expression>)
```

EXPR 计算数学表达式然后通过 output 变量返回计算结果。数学表达式的一个例子是 "5*(10+13)"。该命令支持的运算符包括：+ - * / % ^ ~ << >> ；它们的含义与 C 语言中的完全一致。

CMD#57：message

为用户显示一条消息。

```
message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR]
    "message to display" ...)
```

可以用下述可选的关键字指定消息的类型：

```
(无)            = 重要消息；
STATUS          = 非重要消息；
WARNING         = CMake 警告, 会继续执行；
AUTHOR_WARNING = CMake 警告 (dev), 会继续执行；
SEND_ERROR      = CMake 错误, 继续执行，但是会跳过生成的步骤；
FATAL_ERROR     = CMake 错误, 终止所有处理过程；
```

CMake 的命令行工具会在 stdout 上显示 STATUS 消息，在 stderr 上显示其他所有消息。CMake 的 GUI 会在它的 log 区域显示所有消息。交互式的对话框（ccmake 和 CMakeSetup）将会在状态行上一次显示一条 STATUS 消息，而其他格式的消息会出现在交互式的弹出式对话框中。

CMake 警告和错误消息的文本显示使用的是一种简单的标记语言。文本没有缩进，超过长度的行会回卷，段落之间以新行做为分隔符。

CMD#58： option

为用户提供一个可选项。

```
option(<option_variable> "描述选项的帮助性文字" [initial value])
```

该命令为用户提供了一个在 ON 和 OFF 中做出选择的选项。如果没有指定初始值，将会使用 OFF 作为初值。如果有些选项依赖于其他选项的值，参见 CMakeDependentOption 模块的帮助文件。

CMD#59： output_required_files

输出一个 list，其中包含了一个给定源文件所需要的其他源文件。

```
output_required_files(srcfile outputfile)
```

输出一个指定的源文件所需要的所有源文件的 list。这个 list 会写到 outputfile 变量中。该命令的功能是将 srcfile 的依赖性写出到 outputfile 中，不过该命令将尽可能地跳过.h 文件，搜索依赖中的.cxx，.c 和.cpp 文件。

CMD#60： project

为整个工程设置一个工程名。

```
project(<projectname> [languageName1 languageName2 ... ] )
```

为本工程设置一个工程名。而且，该命令还将变量<projectName>_BINARY_DIR 和<projectName>_SOURCE_DIR 设置为对应值。后面的可选项还可以让你指定你的工程可以支持的语言。比如 CXX(即 C++)，C，Fortran，等等。在默认条件下，支持 C 和 CXX 语言。例如，如果你没有 C++编译器，你可以通过列出你想要支持的语言，例如 C，来明确地禁止对它的检查。使用特殊语言"NONE"，针对任何语言的检查都会被禁止。

CMD#61: qt_wrap_cpp

创建 Qt 包裹器。

```
qt_wrap_cpp(resultingLibraryName DestName SourceLists ...)
```

为所有在 SourceLists 中列出的.h 文件生成 moc 文件。这些 moc 文件将会被添加到那些使用 DestName 源文件列表的库文件中。

Produce moc files for all the .h files listed in the SourceLists. The moc files will be added to the library using the DestName source list.

CMD#62: qt_wrap_ui

创建 Qt 的 UI 包裹器。

```
qt_wrap_ui(resultingLibraryName HeadersDestName SourcesDestName
SourceLists ...)
```

为所有在 SourceLists 中列出的.ui 文件生成.h 和.cxx 文件。这些.h 文件会被添加到使用 HeadersDestNamesource 列表的库中。这些.cxx 文件会被添加到使用 SourcesDestNamesource 列表的库中。

CMD#63: remove_definitions

取消由 add_definitions 命令添加的-D 定义标志。

```
remove_definitions(-DFOO -DBAR ...)
```

在当前及以下的路径，从编译命令行中取消（由 add_definitions 命令添加的）标志。

CMD#64: return

从一个文件，路径或函数内返回。

```
return()
```

从一个文件，路径或函数中返回。若出现在一个 include 文件里（经由 include()或 find_package()命令），该命令会导致当前文件的处理过程停止，并且将控制权转移到试图包含它的文件中。若出现在一个不被任何文件包含的文件中，例如，一个 CMakeLists.txt 中，那么该命令将控制权转移到父目录下，如果存在这样的父目录的话。如果在一个函数中调用 return 函数，控制权会返回到该函数的调用函数那里。注意，宏不是函数，它不会像函数那样去处理 return 命令。

CMD#65: separate_arguments

将空格分隔的参数解析为一个分号分隔的 list。

```
separate_arguments(<var> <UNIX|WINDOWS>_COMMAND "<args>")
```

解析一个 unix 或者 windows 风格的命令行字符串"<args>"，并将结果以分号分隔的 list 的形式存储到<var>中。整个命令行都必须从这个"<args>"参数中给出。

UNIX_COMMAND 模式以没有被括起来的白字符为参数的分隔符。它可以识别单引号和双引号的引号对。反斜杠可以对下一个字符的字面值转义（\"，就是"）;没有其他特殊的转义字符（例如\n 就是 n）。

WINDOWS_COMMAND 模式按照与运行时库相同的语法解析一个 windows 命令行，在启动(starrtup)时构造 argv。它使用没有被双引号括起来的白字符来分隔参数。反斜杠维持其字面含义，除非它们在双引号之前。更多细节，参见 MSDN 的文章："Parsing C Command-Line Arguments"。

```
separate_arguments(VARIABLE)
```

将 VARIABLE 的值转换为一个分号分隔的 list。所有的空格会被替换为';'。该命令可以用来辅助生成命令行。

CMD#66: set

将一个 CMAKE 变量设置为给定值。

```
set(<variable> <value> [[CACHE <type> <docstring> [FORCE]] | PARENT_SCOPE])
```

将变量<variable>的值设置为<value>。在<variable>被设置之前，<value>会被展开。如果有 CACHE 选项，那么<variable>就会添加到 cache 中；这时<type>和<docstring>是必需的。<type>被 CMake GUI 用来选择一个窗口，让用户设置值。<type>可以是下述值中的一个：

```
FILEPATH = 文件选择对话框。
PATH     = 路径选择对话框。
STRING   = 任意的字符串。
BOOL     = 布尔值选择复选框。
INTERNAL = 不需要 GUI 输入端。(适用于永久保存的变量)。
```

如果<type>是内部的(INTERNAL)，那么<value>总是会被写入到 cache 中，并替换任何已经存在于 cache 中的值。如果它不是一个 cache 变量，那么这个变量总是会写入到当前的 makefile 中。FORCE 选项将覆盖 cache 值，从而去掉任何用户带来的改变。

如果指定了 PARENT_SCOPE 选项，变量<variable>将会被设置为当前作用域之上的作用域中。每一个新的路径或者函数都可以创建一个新作用域。该命令将会把一个变量的值设置到父路径或者调用函数中（或者任何类似的可用的情形中。）

如果没有指定<value>，那么这个变量就会被撤销而不是被设置。另见：unset()命令。

```
set(<variable> <value1> ... <valueN>)
```

在这种情形下，<variable>被设置为一个各个值之间由分号分隔的 list。

<variable>可以是环境变量，比如：

```
set( ENV{PATH} /home/martink )
```

在这种情形下，环境变量将会被设置。

CMD#67: set_directory_properties

设置某个路径的一种属性。

```
set_directory_properties(PROPERTIES prop1 value1 prop2 value2)
```

为当前的路径及其子路径设置一种属性。如果该属性不存在，CMake 将会报告一个错误。属性包括：INCLUDE_DIRECTORIES, LINK_DIRECTORIES, INCLUDE_REGULAR_EXPRESSION, 以及 ADDITIONAL_MAKE_CLEAN_FILES 共四种。ADDITIONAL_MAKE_CLEAN_FILES 是一个文件名的 list，其中包含有"make clean"阶段会被清除掉的文件。

CMD#68: set_property

在给定的作用域内设置一个命名的属性。

```
set_property(<GLOBAL                        |
             DIRECTORY [dir]                |
             TARGET    [target1 [target2 ...]] |
             SOURCE    [src1 [src2 ...]]     |
             TEST      [test1 [test2 ...]]   |
             CACHE     [entry1 [entry2 ...]]>
            [APPEND]
            PROPERTY <name> [value1 [value2 ...]])
```

为作用域里的 0 个或多个对象设置一种属性。第一个参数决定了属性可以影响到的作用域。他必须是下述值之一：GLOBAL，全局作用域，唯一，并且不接受名字。DIRECTORY，路径作用域，默认为当前路径，但是也可以用全路径或相对路径指定其他值。TARGET，目

标作用域，可以命名 0 个或多个已有的目标。SOURCE，源作用域，可以命名 0 个或多个源文件。注意，源文件属性只对加到相同路径（CMakeLists.txt）中的目标是可见的。TEST 测试作用域可以命名 0 个或多个已有的测试。CACHE 作用域必须指定 0 个或多个 cache 中已有的条目。

PROPERTY 选项是必须的，并且要紧跟在待设置的属性的后面。剩余的参数用来组成属性值，该属性值是一个以分号分隔的 list。如果指定了 APPEND 选项，该 list 将会附加在已有的属性值之后。

CMD#69: set_source_files_properties

源文件有一些属性来可以改变它们构建的方式。

```
set_source_files_properties([file1 [file2 [...]]]
                            PROPERTIES prop1 value1
                            [prop2 value2 [...]])
```

以键/值对的方式设置与源文件相关的那些属性值。那些 CMake 中的源文件属性，参见关于属性的相关文档。不能被识别的属性将会被忽略。源文件属性只对同一路径（CMakeLists.txt）中添加的目标可见。

CMD#70: set_target_properties

设置目标的一些属性来改变它们构建的方式。

```
set_target_properties(target1 target2 ...
                      PROPERTIES prop1 value1
                      prop2 value2 ...)
```

为一个目标设置属性。该命令的语法是列出所有你想要变更的文件，然后提供你想要设置的值。你能够使用任何你想要的属性/值对，并且在随后的代码中调用 GET_TARGET_PROPERTY 命令取出属性的值。

影响一个目标输出文件的名字的属性详述如下。PREFIX 和 SUFFIX 属性覆盖了默认的目标名前缀（比如 lib）和后缀（比如.so）。IMPORT_PREFIX 和 IMPORT_SUFFIX 是与之等价的属性，不过针对的是 DLL（共享库目标）的导入库。在构建目标时，OUTPUT_NAME 属性设置目标的真实名字，并且可以用来辅助创建两个具有相同名字的目标，即使 CMake 需要唯一的逻辑目标名。<CONFIG>_OUTPUT_NAME 可以为不同的配置设置输出的目标名字。当目标在指定的配置名<CONFIG>（全部大写，例如 DEBUG_POSTFIX）下被构建时，<CONFIG>_POSTFIX 为目标的真实名字设置一个后缀。该属性的值在目标创建时被初始化为 CMAKE_<CONFIG>_POSTFIX 的值（可执行目标除外，因为较早的 CMake 版本不会为可执行文件使用这个属性。）

LINK_FLAGS 属性可以用来为一个目标的链接阶段添加额外的标志。LINK_FLAGS_<CONFIG>将为配置<CONFIG>添加链接标志，例如 DEBUG，RELEASE，MINSIZEREL，RELWITHDEBINFO。DEFINE_SYMBOL 属性设置了编译一个共享库中

的源文件时才会被定义的预处理器符号名。如果这个值没有被设置的话，那么它会被设置为默认值 target_EXPORTS（如果目标不是一个合法的 C 标示符的话可以用一些替代标志）。这对于检测头文件是包含在它们的库以内还是以外很有帮助，从而可以合理设置 dllexport/dllimport 修饰符（注意，只有在编译到的时候，这个符号才会被定义；因此猜测在代码中，判断预处理符号是否被定义可以知道依赖库是导入的还是导出的——译注）。COMPILE_FLAGS 属性可以设置附加的编译器标志，它们会在构建目标内的源文件时被用到。它也可以用来传递附加的预处理器定义。

LINKER_LANGUAGE 属性用来改变链接可执行文件或共享库的工具。默认的值是设置与库中的文件相匹配的语言。CXX 和 C 是这个属性的公共值。

对于共享库，VERSION 和 SOVERSION 属性分别可以用来指定构建的版本号以及 API 版本号。当构建或者安装时，如果平台支持符号链接并且链接器支持 so 名字，那么恰当的符号链接会被创建。如果只指定两者中的一个，缺失的另一个假定为具有相同的版本号。对于可执行文件，VERSION 可以被用来指定构建版本号。当构建或者安装时，如果该平台支持符号链接，那么合适的符号链接会被创建。对于在 Windows 系统而言，共享库和可执行文件的 VERSION 属性被解析成为一个"major.minor"的版本号。这些版本号被用做该二进制文件的镜像版本。

还有一些属性用来指定 RPATH 规则。INSTALL_RPATH 是一个分号分隔的 list，它指定了在安装目标时使用的 rpath（针对支持 rpath 的平台而言）（-rpath 在 gcc 中用于在编译时指定加载动态库的路径；优先级较系统库路径要高。详情参见 http://www.cmake.org/Wiki/CMake_RPATH_handling#What_is_RPATH_.3F——译注）。INSTALL_RPATH_USE_LINK_PATH 是一个布尔值属性，如果它被设置为真，那么在链接器的搜索路径中以及工程之外的目录会被附加到 INSTALL_RPATH 之后。SKIP_BUILD_RPATH 是一个布尔值属性，它指定了是否跳过一个 rpath 的自动生成过程，从而可以从构建树开始运行。BUILD_WITH_INSTALL_RPATH 是一个布尔值属性，它指定了是否将在构建树上的目标与 INSTALL_RPATH 链接。该属性要优先于 SKIP_BUILD_RPATH，因此避免了安装之前的重新链接。INSTALL_NAME_DIR 是一个字符串属性，它用于在 Mac OSX 系统上，指定了被安装的目标中使用的共享库的"install_name"域的目录部分。如果目标已经被创建，变量 CMAKE_INSTALL_RPATH, CMAKE_INSTALL_RPATH_USE_LINK_PATH, CMAKE_SKIP_BUILD_RPATH, CMAKE_BUILD_WITH_INSTALL_RPATH 和 CMAKE_INSTALL_NAME_DIR 的值会被用来初始化这个属性。

PROJECT_LABEL 属性可以用来在 IDE 环境，比如 visual studio，中改变目标的名字。 VS_KEYWORD 可以用来改变 visual studio 的关键字，例如如果该选项被设置为 Qt4VSv1.0 的话，QT 集成将会运行得更好。

VS_SCC_PROJECTNAME, VS_SCC_LOCALPATH, VS_SCC_PROVIDER 可以被设置，从而增加在一个 VS 工程文件中对源码控制绑定的支持。

PRE_INSTALL_SCRIPT 和 POST_INSTALL_SCRIPT 属性是在安装一个目标之前及之后指定运行 CMake 脚本的旧格式。只有当使用旧式的 INSTALL_TARGETS 来安装目标时，才能使用这两个属性。使用 INSTALL 命令代替这种用法。

EXCLUDE_FROM_DEFAULT_BUILD 属性被 visual studio 生成器使用。如果属性值设置为 1，那么当你选择"构建解决方案"时，目标将不会成为默认构建的一部分。

CMD#71: set_tests_properties

设置若干个测试的属性值。

```
set_tests_properties(test1 [test2...] PROPERTIES prop1 value1 prop2 value2)
```

为若干个测试设置一组属性。若属性未被发现，CMake 将会报告一个错误。这组属性包括：WILL_FAIL， 如果设置它为 true，那将会把这个测试的"通过测试/测试失败"标志反转。PASS_REGULAR_EXPRESSION，如果它被设置，这个测试的输出将会被检测是否违背指定的正则表达式，并且至少要有一个正则表达式要匹配；否则测试将会失败。

例子: PASS_REGULAR_EXPRESSION "TestPassed;All ok"

FAIL_REGULAR_EXPRESSION: 如果该属性被设置，那么只要输出匹配给定的正则表达式中的一个，那么测试失败。

例子: PASS_REGULAR_EXPRESSION "[^a-z]Error;ERROR;Failed"

PASS_REGULAR_EXPRESSION 和 FAIL_REGULAR_EXPRESSION 属性都期望一个正则表达式列表（list）作为其参数。

TIMEOUT: 设置该属性将会限制测试的运行时长不超过指定的秒数。

CMD#72: site_name

将给定的变量设定为计算机名。

```
site_name(variable)
```

CMD#73: source_group

为 Makefile 中的源文件定义一个分组。

```
source_group(name [REGULAR_EXPRESSION regex] [FILES src1 src2 ...])
```

为工程中的源文件中定义一个分组。这主要用来在 Visual Studio 中建立文件组按钮 (file tabs)。所有列出来的文件或者匹配正则表达式的文件都会被放到这个文件组中。如果一个文件匹配多个组，那么**最后**明确地列出这个文件的组将会包含这个文件，如果有这样的组的话。如果没有任何组明确地列出这个文件，那么**最后**那个其正则表达式与该文件名匹配的组，将会成为最终候选者。

组名中可以包含反斜杠，以指定子文件组：source_group(outer\\inner ...)

为了保持后向兼容性，这个命令也支持这种格式：source_group(name regex)

CMD#74: string

字符串操作函数。

```
  string(REGEX MATCH <regular_expression> <output variable> <input> [<input>...])
  string(REGEX MATCHALL <regular_expression> <output variable> <input>
[<input>...])
  string(REGEX REPLACE <regular_expression> <replace_expression> <output variable>
<input> [<input>...])
  string(REPLACE <match_string> <replace_string> <output variable> <input>
[<input>...])
  string(COMPARE EQUAL <string1> <string2> <output variable>)
  string(COMPARE NOTEQUAL <string1> <string2> <output variable>)
  string(COMPARE LESS <string1> <string2> <output variable>)
  string(COMPARE GREATER <string1> <string2> <output variable>)
  string(ASCII <number> [<number> ...] <output variable>)
  string(CONFIGURE <string1> <output variable> [@ONLY] [ESCAPE_QUOTES])
  string(TOUPPER <string1> <output variable>)
  string(TOLOWER <string1> <output variable>)
  string(LENGTH <string> <output variable>)
  string(SUBSTRING <string> <begin> <length> <output variable>)
  string(STRIP <string> <output variable>)
  string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>] [RANDOM_SEED <seed>]
<output variable>)
```

REGEX MATCH：匹配正则表达式一次，然后将匹配的值存储到输出变量中。

REGEX MATCHALL：尽可能多次地匹配正则表达式，然后将匹配的值以 list 的形势存储到输出变量中。

REGEX REPLACE：尽可能多次地匹配正则表达式，并且将匹配的值用 replacement expression 替换掉，然后存储到输出变量中。这个 replace expression 可以引用包含匹配字符串的子表达式，这些匹配的字符串用圆括号隔开的\1，\2，...，\9 等加以引用。注意：在 CMake 代码里，如果要使用一个反斜杠，必须要用两个反斜杠(\\1)转义，才能通过参数解析。

REPLACE：将输入字符串内所有出现 match_string 的地方都用 replace_string 代替，然后将结果存储到输出变量中。

COMPARE EQUAL/NOTEQUAL/LESS/GREATER：将会比较两个字符串，然后将比较的结果（true/false）存储到输出变量中。

ASCII：将会把所有数字转换为对应的 ASCII 字符。

CONFIGURE：将一个字符串进行变换，这种变换与将一个 FILE 变换为 CONFIGURE_FILE 相似。

TOUPPER/TOLOWER：将字符串转换为大写/小写字符。

LENGTH：返回给定字符串的长度。

SUBSTRING：返回给定字符串的子串。

STRIP：返回一个给定字符串的子串，它会去掉原先字符串开始和结尾的空格。

RANDOM：将会返回一个给定长度的随机字符串，它由给定的字母表中的字母组成。默认的长度是 5 个字符，默认的字母表是全部的大小写字母以及数字。如果指定了一个整数 RANDOM_SEED，它的值将会被用做随机数发生器的种子。

在正则表达式中，下述字符有特殊含义：

^	在行首匹配。
$	在行尾匹配。
.	匹配任意单个字符。
[ ]	匹配在中括号中的任意字符。
[^]	匹配不在中括号中的任意字符。
-	匹配任意在短横线两端字符闭区间中间的任意一个字符。
*	匹配先前模式零次或多次。
+	匹配先前模式一次或多次。
?	匹配先前模式零次或一次。
|	匹配|两侧的任意一种模式。
()	保存一个匹配的子表达式，这个子表达式后续可以在 REGEX REPLACE 操作中以\n 的方式引用。 它也会被所有正则表达式相关的命令所保存；包括，比如，如果用到 if(MATCHES)命令的话，这些匹配的值被保存在变量 CMAKE_MATCH_(0..9)中。

CMD#75: target_link_libraries

将给定的库链接到一个目标上。

target_link_libraries(<target> [item1 [item2 [...]]] [[debug|optimized|general] <item>] ...)

为给定的目标设置连接时使用的库或者标志(flags)。如果一个库名字与工程中的另外一个目标相匹配，一个依赖关系会自动添加到构建系统中来，这样就可以在链接目标之前，保证正在被链接的库是最新的。以"-"开始，但不是"-l"或"-framework"的那些项，将会被当作链接器标志来处理。

关键字"debug"，"optimized"或者"general" 表示紧随关键字之后的库仅仅会被用到相应的构建配置上。"debug"关键字对应于调试配置（或者，如果全局属性 DEBUG_CONFIGURATIONS 被设置的话，就是 DEBUG_CONFIGURATIONS 中的名字所指定的配置）。"optimized"关键字对应于所有其他的配置类型。"general"关键字对应于所有的配置，并且纯粹是可选的（它是默认配置，可以省略）。通过创建并链接到导入库目标，可以对每种配置规则进行更细致的粒度控制。更多内容参见 add_library 命令的 IMPORTED 模式。

默认时，库之间的依赖性是可传递的。当这个目标被链接到其他目标上时，那么链接到这个目标上的库也会出现在其他目标的链接依赖上。参见 LINK_INTERFACE_LIBRARIES 属性的相关文档，其中有关于如何覆盖一个目标的链接依赖性传递设置的介绍。

```
target_link_libraries(<target> LINK_INTERFACE_LIBRARIES
[[debug|optimized|general] <lib>] ...)
```

对于 LINK_INTERFACE_LIBRARIES 模式，它将会把库附加在 LINK_INTERFACE_LIBRARIES 以及 LINK_INTERFACE_LIBRARIES 在不同配置下的等价目标属性，而不是用这些库去链接。指定为“debug”的库将会被附加到 LINK_INTERFACE_LIBRARIES_DEBUG 属性(或者是在 DEBUG_CONFIGURATIONS 全局属性中列出的配置，如果 DEBUG_CONFIGURATIONS 被设置的话）。指定为“optimized”库将会被附加到 LINK_INTERFACE_LIBRARIES 属性上。指定为“general”的库(或者没有任何关键字的库)，将会被当做即被指定为“debug”又被指定为“optimized”对待。

库之间的依赖图通常是非循环图（DAG），但是如果出现互相依赖的**静态**库，CMake 会允许依赖图中包含循环依赖(强连通分支)。当其它目标链接到这些库中的一个时，CMake 会重复整个连通分支。例如，代码：

```
1 add_library(A STATIC a.c)
2 add_library(B STATIC b.c)
3 target_link_libraries(A B)
4 target_link_libraries(B A)
5 add_executable(main main.c)
6 target_link_libraries(main A)
```

将“main”链接到了“A B A B”。（虽然通常一次重复就足够了，但是病态对象文件以及符号排布可能需要多次重复。你可以通过在上一次 target_link_libraries 调用中手动重复该分支来处理这种情况。不过，如果两个归档文件确实是如此紧密的相互关联，它们可能会被合并为一个单一的归档文件。）

CMD#76: try_compile

尝试编译一些代码。

```
try_compile(RESULT_VAR bindir srcdir
            projectName <targetname> [CMAKE_FLAGS <Flags>]
            [OUTPUT_VARIABLE var])
```

尝试编译一个程序。在这种格式时，srcdir 路径下应该包含一个完整的 CMake 工程，包括 CMakeLists.txt 文件以及所有的源文件。在该命令运行完之后，路径 bindir 和 srcdir 不会被删除。如果指定了<target name>，那么 CMake 将只构建那个目标；否则，目标 all 或 ALL_BUILD 将会被构建。

```
try_compile(RESULT_VAR bindir srcfile
            [CMAKE_FLAGS <Flags>]
            [COMPILE_DEFINITIONS <flags> ...]
            [OUTPUT_VARIABLE var]
            [COPY_FILE <filename> )
```

尝试编译一个 srcfile。在这种情况下，用户仅仅需要提供源文件。CMake 会创建合适的 CMakeLists.txt 文件来构建源文件。如果使用了 COPY_FILE 选项，编译出的文件将会被拷贝到给定的文件那里。

在这个版本里，所有在 bindir/CMakeFiles/CMakeTmp 文件夹下的文件，将会被自动清除；通过向 CMake 传递调试选项--debug-trycompile 可以避免这个清除步骤。另外一些可以包含的额外标志有：INCLUDE_DIRECTORIES, LINK_DIRECTORIES, 和 LINK_LIBRARIES。COMPILE_DEFINITIONS 是通过-Ddefinations 选项设置的预定义符号，这会传递到编译器命令行。try_compile 命令在构建过程中伴随创建出的 CMakeLists.txt 看起来像这样：

```
add_definitions( <expanded COMPILE_DEFINITIONS from calling cmake>)
include_directories(${INCLUDE_DIRECTORIES})
link_directories(${LINK_DIRECTORIES})
add_executable(cmTryCompileExec sources)
target_link_libraries(cmTryCompileExec ${LINK_LIBRARIES})
```

在该命令的这两种版本里，如果指定了 OUTPUT_VARIABLE，那么构建过程的输出会存储到给定的变量里。编译成功或失败的结果，会通过 RESULT_VAR 返回。CMAKE_FLAGS 可以用来向正在构建的 CMake 传递-DVAR:TYPE = VALUE 符号。

CMD#77: try_run

尝试编译并运行某些代码。

```
try_run(RUN_RESULT_VAR COMPILE_RESULT_VAR
        bindir srcfile [CMAKE_FLAGS <Flags>]
        [COMPILE_DEFINITIONS <flags>]
        [COMPILE_OUTPUT_VARIABLE comp]
        [RUN_OUTPUT_VARIABLE run]
        [OUTPUT_VARIABLE var]
        [ARGS <arg1> <arg2>...])
```

尝试编译一个源文件 srcfile。通过变量 COMPILE_RESULT_VAR 返回 TRUE 或者 FALSE 来反应编译是否失败。如果构建出了可执行文件，但是不能运行，那么 RUN_RESULT_VAR 会被设置为 FAILED_TO_RUN。COMPILE_OUTPUT_VARIABLE 变量指定了一个变量，这个变量存储了构建步骤输出的信息。RUN_OUTPUT_VARIABLE 指定了一个变量，这个变量存储了运行可执行文件时的输出。出于兼容性的考虑，OUTPUT_VARIABLE 还会被支持，它包含了包含编译和运行阶段的输出信息。

交叉编译相关问题

当运行交叉编译时，第一步中编译出的可执行文件通常不能在编译宿主机上直接运行。try_run()函数会检查 CMAKE_CROSSCOMPILING 变量来检测 CMake 是否是交叉编译模式。如果是的话，CMake 还是会尝试编译可执行文件，但是它不会尝试运行可执行文件。相反，他会创建一些 cache 变量，这些变量必须由用户填充，或者在某个 CMake 脚本中预先设置为那些在真实目标机平台上执行的结果。这些变量有：RUN_RESULT_VAR (解释

参见上文)，以及如果使用了 RUN_OUTPUT_VARIABLE (或者 OUTPUT_VARIABLE) ，还有一个附加的 cache 变量 RUN_RESULT_VAR__COMPILE_RESULT_VAR__TRYRUN_OUTPUT。该变量是为了保存执行过程中 stdout 和 stderr 的输出。

为了让交叉编译更加容易些，必要时再使用 try_run 命令。如果你使用了 try_run 命令，那么只有必要时才使用 RUN_OUTPUT_VARIABLE（或者 OUTPUT_VARIABLE）变量。在交叉编译时，使用这些变量需要 cache 变量必须被手动设置为可执行文件的输出。你也可以用 if(CMAKE_CROSSCOMPILING)将 try_run 的调用"保护"起来，同时还要为这种情形给定一个易于预先设置的备选方案。

CMD#78 unset

撤销对一个变量，cache 变量或者环境变量的设置。

```
unset(<variable> [CACHE])
```

删除一个指定的变量，让它变成未定义的。如果指定了 CACHE 选项，那么这个变量将会从 cache 中删除而不是当前作用域。<variable>可以是一个环境变量，比如：

```
unset(ENV{LD_LIBRARY_PATH})
```

在这个例子中，这个变量将会从当前的环境中被删除。

CMD#79 : variable_watch

监视 CMake 变量的改变。

```
variable_watch(<variable name> [<command to execute>])
```

如果给定的变量发生了变化，关于正在被改写的变脸的消息会被打印出来。如果指定了 command 选项，这条命令会被执行。这条命令会接受这样的参数：COMMAND(<variable> <access> <value> <current list file> <stack>)

CMD#80: while

当条件为真时，评估（执行）一组命令。

```
while(condition)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endwhile(condition)
```

所有在 while 和与之配对的 endwhile 之间的命令将会被记录，但并不会执行。只有当 endwhile 被评估，并且条件为真时，这个命令列表的记录才会被调用。条件值的评估与 if 命令使用相同的逻辑。

这段时间因为项目进展不顺，翻译手册的事情暂时被搁置；今天总算能抽出一点点时间略微弥补一下。CMake 的命令在上一节已经介绍完了，因为总结这些用法需要大块的时间去构思例子，暂且省去这段 loos ends，待以后有假期时补上；先进入另一主题：CMake 变量。

CMake 变量按功能分有主要有四种不同的类型：1.) 提供信息的变量[共 53 个]；2.) 改变行为的变量[共 23 个]；3.) 描述系统的变量[共 24 个]；4.)控制构建过程的变量[共 22 个]。此外还有一些变量因编译使用的语言不同而不同，将它们归为第五类[共 29 个]。 由于变量比较多，这里只给出变量的大概描述；具体作用可使用 cmake --help-variable variable_name 命令查看。

一、提供信息的变量

VAR#1-1 : CMAKE_AR 静态库的归档工具名字。

VAR#1-2 : CMAKE_BINARY_DIR 构建树的顶层路径。

VAR#1-3 : CMAKE_BUILD_TOOL 实际构建过程中使用的工具。

VAR#1-4 : CMAKE_CACHEFILE_DIR 文件 CMakeCache.txt 所在的路径。

VAR#1-5 : CMAKE_CACHE_MAJOR_VERSION 用于创建 CMakeCache.txt 文件的 CMake 的主版本号。

VAR#1-6 : VCMAKE_CACHE_MINOR_VERSION 用于创建 CMakeCache.txt 文件的 CMake 的子版本号。

VAR#1-7 : CMAKE_CACHE_PATCH_VERSION 用于创建 CMakeCache.txt 文件的 CMake 的补丁号。

VAR#1-8 : CMAKE_CFG_INTDIR 构建时，与构建配置相对应的输出子路径（只读）。

VAR#1-9 : CMAKE_COMMAND 指向 CMake 可执行文件的完整路径。

VAR#1-10 : CMAKE_CROSSCOMPILING 指出 CMake 是否正在交叉编译。

VAR#1-11 : CMAKE_CTEST_COMMAND 与 cmake 一起安装的 ctest 命令的完整路径。

VAR#1-12 : CMAKE_CURRENT_BINARY_DIR 当前正在被处理的二进制目录的路径。

VAR#1-13 : CMAKE_CURRENT_LIST_DIR 当前正在处理的 listfile 的完整目录。

VAR#1-14 : CMAKE_CURRENT_LIST_FILE 当前正在处理的 listfile 的完整路径。

VAR#1-15 : CMAKE_CURRENT_LIST_LINE 当前正在处理的 listfile 的行号。

VAR#1-16 : CMAKE_CURRENT_SOURCE_DIR 指向正在被处理的源码目录的路径。

VAR#1-17 : CMAKE_DL_LIBS 包含 dlopen 和 dlclose 函数的库的名称。

VAR#1-18 : CMAKE_EDIT_COMMAND 指向 cmake-gui 或 ccmake 的完整路径。

VAR#1-19：CMAKE_EXECUTABLE_SUFFIX(_<LANG>) 本平台上可执行文件的后缀。

VAR#1-20 ： CMAKE_EXTRA_GENERATOR 构建本工程所需要的额外生成器。

VAR#1-21 ： CMAKE_EXTRA_SHARED_LIBRARY_SUFFIXES 附加的共享库后缀（除 CMAKE_SHARED_LIBRARY_SUFFIX 以外，其他可以识别的共享库的后缀名。）

VAR#1-22 ： CMAKE_GENERATOR 用于构建该工程的生成器。

VAR#1-23 ： CMAKE_HOME_DIRECTORY 指向源码树顶层的路径。

VAR#1-24 ： CMAKE_IMPORT_LIBRARY_PREFIX(_<LANG>) 需要链接的导入库的前缀。

VAR#1-25 ： CMAKE_IMPORT_LIBRARY_SUFFIX(_<LANG>) 需要链接的导入库的后缀。

VAR#1-26 ： CMAKE_LINK_LIBRARY_SUFFIX 需要链接的库的后缀。

VAR#1-27：CMAKE_MAJOR_VERSION cmake 的主版本号（例如 2.X.X 中的 2）。

VAR#1-28 ： CMAKE_MAKE_PROGRAM 参见 CMAKE_BUILD_TOOL。

VAR#1-29：CMAKE_MINOR_VERSION cmake 的次版本号（例如 X.4.X 中的 4）。

VAR#1-30 ： CMAKE_PARENT_LIST_FILE 当前正在被处理 listfile 的父 listfile 的全路径。

VAR#1-31 ： CMAKE_PATCH_VERSION cmake 的补丁号(例如 X.X.3 中的 3)。

VAR#1-32 ： CMAKE_PROJECT_NAME 当前工程的工程名。

VAR#1-33：CMAKE_RANLIB 静态库的随机化工具的名字(比如 linux 下的 ranlib)。

VAR#1-34 ： CMAKE_ROOT CMake 的安装路径。

VAR#1-35 ： CMAKE_SHARED_LIBRARY_PREFIX(_<LANG>) 被链接的共享库的前缀。

VAR#1-36 ： CMAKE_SHARED_LIBRARY_SUFFIX(_<LANG>) 被链接的共享库的后缀。

VAR#1-37 ： CMAKE_SHARED_MODULE_PREFIX(_<LANG>) 被链接的可加载模块的前缀。

VAR#1-38 ： CMAKE_SHARED_MODULE_SUFFIX(_<LANG>) 被链接的共享库的后缀。

VAR#1-39 ： CMAKE_SIZEOF_VOID_P void 指针的长度。

VAR#1-40： CMAKE_SKIP_RPATH 如果变量为真，不为编译出的可执行文件添加运行时的路径信息。默认添加。

VAR#1-41 ： CMAKE_SOURCE_DIR 源码树的顶层路径。

VAR#1-42 ： CMAKE_STANDARD_LIBRARIES 链接到所有可执行文件和共享库上的库。这是一个 list。

VAR#1-43 ： CMAKE_STATIC_LIBRARY_PREFIX(_<LANG>) 被链接的静态库的前缀。

VAR#1-44 ： CMAKE_STATIC_LIBRARY_SUFFIX(_<LANG>) 被链接的静态库的后缀。

VAR#1-45： CMAKE_TWEAK_VERSION cmake 的 tweak 版本号(例如 X.X.X.1 中的 1)。

VAR#1-46 ： CMAKE_USING_VC_FREE_TOOLS 如果用到了免费的 visual studio 工具，设置为真。

VAR#1-47 ： CMAKE_VERBOSE_MAKEFILE 设置该变量为真将创建完整版本的 makefile。

VAR#1-48 ： CMAKE_VERSION cmake 的完整版本号；格式为 major.minor.patch[.tweak[-id]]。

VAR#1-49 ： PROJECT_BINARY_DIR 指向工程构建目录的全路径。

VAR#1-50 ： PROJECT_NAME 向 project 命令传递的工程名参数。

VAR#1-51 ： PROJECT_SOURCE_DIR 当前工程的源码路径。

VAR#1-52： [Project name]_BINARY_DIR 给定工程的二进制文件顶层路径。

VAR#1-53 ： [Project name]_SOURCE_DIR 给定工程的源码顶层路径。

Preface： 本文是 CMake 官方文档 CMake Tutorial (http://www.cmake.org/cmake/help/cmake_tutorial.html) 的翻译。通过一个样例工程从简单到复杂的完善过程，文档介绍了 CMake 主要模块（cmake, ctest, cpack）的功能和使用环境；从中可以一窥 cmake 的大体形貌。正文如下：

本文下述内容是一个手把手的使用指南；它涵盖了 CMake 需要解决的公共构建系统的一些问题。这些主题中的许多主题已经在 Mastering CMake 一书中以单独的章节被介绍过，但是通过一个样例工程看一看它们如何工作也是非常有帮助的。本指南可以在 CMake 源码树的 Tests/Tutorial 路径下找到。每一步都有它自己的子路径，其中包含该步骤的一个完整的指南。

作为基础的起始点（步骤 1）

最基本的工程是一个从源代码文件中构建可执行文件的例子。对于简单工程，只要一个两行的 CMakeLists 文件就足够了。这将会作为我们指南的起点。这份 CMakeLists 文件看起来像是这样：

```
1 cmake_minimum_required (VERSION 2.6)
2 project (Tutorial)
3 add_executable(Tutorial tutorial.cxx)
```

注意到这个例子在 CMakeLists 文件中使用了小写。CMake 支持大写、小写、混合大小写的命令。tutorial.cxx 中的源代码用来计算一个数的平方根，并且它的第一版非常简单，如下所示：

```
// A simple program that computes the square root of a number
// 计算一个数的平方根的简单程序
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int  main (int  argc, char  *argv[])
{
    if  (argc < 2)
        {
        fprintf(stdout,"Usage: %s number\n",argv[0]);
        return  1;
        }
    double  inputValue = atof(argv[1]);
    double  outputValue = sqrt(inputValue);
    fprintf(stdout,"The square root of %g is %g\n",
                    inputValue, outputValue);
    return  0;
}
```

我们添加的第一个特性用来为工程和可执行文件指定一个版本号。虽然你可以在源代码中唯一指定它，但是你在 CMakeLists 文件中指定它可以提供更好的灵活性。如下所示，我么可以通过添加一个版本号来修改 CMakeLists 文件：

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
# 版本号
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)
# 配置一个头文件，通过它向源代码中传递一些 CMake 设置。
configure_file (
    "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
    "${PROJECT_BINARY_DIR}/TutorialConfig.h"
    )
# 将二进制文件树添加到包含文件的搜索路径中，这样我们可以找到 TutorialConfig.h
include_directories("${PROJECT_BINARY_DIR}")
# 添加可执行文件
add_executable(Tutorial tutorial.cxx)
```

由于配置过的文件将会被写到二进制文件目录下，我们必须把该目录添加到包含文件的搜索路径清单中。然后，以下的代码就可以在源目录下创建一份 TotorialConfig.h.in 文件：

```
// 与 tutorial 相关的配置好的选项与设置；
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

当 CMake 配置这份头文件时，@Tutorial_VERSION_MAJOR@和 @Tutorial_VERSION_MINOR@的值将会被从 CMakeLists 文件中传递过来的值替代。下一步，我们要修改 tutorial.cxx 来包含 configured 头文件然后使用其中的版本号。修改过的源代码展列于下：

```
// 计算平方根的简单程序。
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "TutorialConfig.h"

int  main (int  argc, char  *argv[])
{
    if  (argc < 2)
        {
        fprintf(stdout,"%s Version %d.%d\n",
                        argv[0],
                        Tutorial_VERSION_MAJOR,
                        Tutorial_VERSION_MINOR);
        fprintf(stdout,"Usage: %s number\n",argv[0]);
        return  1;
        }
    double  inputValue = atof(argv[1]);
    double  outputValue = sqrt(inputValue);
    fprintf(stdout,"The square root of %g is %g\n",
                    inputValue, outputValue);
    return  0;
}
```

引入库（步骤 2）

现在我们将会在我们的工程中引入一个库。这个库会包含我们自己实现的计算一个数的平方根的函数。可执行文件随后可以使用这个库文件而不是编译器提供的标准开平方函数。在本指南中，我们将会把库文件放到一个子目录 MathFunctions 中。它包含下述的单行 CMakeLists 文件：

```
add_library(MathFunctions mysqrt.cxx)
```

源文件 mysqrt.cxx 有一个叫做 mysqrt 的函数，它提供了与编译器的 sqrt 函数类似的功能。为了使用新的库，我们在顶层的 CMakeLists 中增加一个 add_subrirectory 调用，这样这个库也会被构建。我们也要向可执行文件中增加另一个头文件路径，这样就可以从

MathFunctions/mysqrt.h 头文件中找到函数的原型。最后的一点更改是在向可执行文件中引入新的库。顶层 CMakeLists 文件的最后几行现在看起来像是这样：

```
include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
add_subdirectory (MathFunctions)
# 引入可执行文件
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial MathFunctions)
```

现在，让我们考虑下让 MathFunctions 库变为可选的。在本指南中，确实没有必要这样画蛇添足；但是对于更大型的库或者依赖于第三方代码的库，你可能需要这种可选择性。第一步是为顶层的 CMakeLists 文件添加一个选项：

```
# 我们应该使用我们自己的数学函数吗？
option (USE_MYMATH
                "Use tutorial provided math implementation"  ON)
```

这将会在 CMake 的 GUI 中显示一个默认的 ON 值，并且用户可以随需改变这个设置。这个设置会被存储在 cache 中，那么用户将不需要在 cmake 该工程时，每次都设置这个选项。第二处改变是，让链接 MathFunctions 库变为可选的。要实现这一点，我们修改顶层 CMakeLists 文件的结尾部分：

```
# 添加 MathFunctions 库吗？
if  (USE_MYMATH)
    include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
    add_subdirectory (MathFunctions)
    set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)
# 添加可执行文件
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial   ${EXTRA_LIBS})
```

这里用 USE_MYMATH 设置来决定是否 MathFunctions 应该被编译和执行。注意到，要用一个变量（在这里是 EXTRA_LIBS）来收集所有以后会被连接到可执行文件中的可选的库。这是保持带有许多可选部件的较大型工程干净清爽的一种通用的方法。源代码对应的改变相当直白，如下所示：

```
// 计算一个数平方根的简单程序
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "TutorialConfig.h"
#ifdef USE_MYMATH
#include "MathFunctions.h"
#endif

```

```
10      int  main (int  argc, char  *argv[])
11      {
12          if  (argc < 2)
13              {
14              fprintf(stdout,"%s Version %d.%d\n", argv[0],
15                                  Tutorial_VERSION_MAJOR,
16                                  Tutorial_VERSION_MINOR);
17              fprintf(stdout,"Usage: %s number\n",argv[0]);
18              return  1;
19              }
20
21          double  inputValue = atof(argv[1]);
22
23      #ifdef USE_MYMATH
24          double  outputValue = mysqrt(inputValue);
25      #else
26          double  outputValue = sqrt(inputValue);
27      #endif
28
29          fprintf(stdout,"The square root of %g is %g\n",
30                              inputValue, outputValue);
31          return  0;
32      }
```

在源代码中，我们也使用了 USE_MYMATH。这个宏是由 CMake 通过 TutorialConfig.h.in 配置文件中的下述语句行提供给源代码的：

+ View Code

安装与测试（步骤 3）

下一步我们会为我们的工程引入安装规则以及测试支持。安装规则相当直白，对于 MathFunctions 库，我们通过向 MathFunctions 的 CMakeLists 文件添加如下两条语句来设置要安装的库以及头文件：

```
1   install (TARGETS MathFunctions DESTINATION bin)
2   install (FILES MathFunctions.h DESTINATION include)
```

对于应用程序，在顶层 CMakeLists 文件中添加下面几行，它们用来安装可执行文件以及配置头文件：

```
1   # 添加安装目标
2   install (TARGETS Tutorial DESTINATION bin)
3   install (FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
4                    DESTINATION include)
```

这就是要做的全部;现在你应该可以构建 tutorial 工程了。然后,敲入命令 make install（或者从 IDE 中构建 INSTALL 目标）然后它就会安装需要的头文件，库以及可执行文件

CMake 的变量 CMAKE_INSTALL_PREFIX 用来确定这些文件被安装的根目录。添加测试同样也只需要相当浅显的过程。在顶层 CMakeLists 文件的的尾部补充许多基本的测试代码来确认应用程序可以正确工作。

+ View Code

第一个测试用例仅仅用来验证程序可以运行，没有出现段错误或其他的崩溃，并且返回值必须是 0。这是 CTest 所做测试的基本格式。余下的几个测试都是用 PASS_REGULAR_EXPRESSION 测试属性来验证测试代码的输出是否包含有特定的字符串。在本例中，测试样例用来验证计算得出的平方根与预定值一样；当指定错误的输入数据时，要打印用法信息。如果你想要添加许多测试不同输入值的样例，你应该考虑创建如下所示的宏：

+ View Code

对于每个 do_test 宏调用，都会向工程中添加一个新的测试用例；宏参数是测试名、函数的输入以及期望结果。

增加系统内省（步骤 4）

下一步，让我们考虑向我们的工程中引入一些依赖于目标平台上可能不具备的特性的代码。在本例中，我们会增加一些依赖于目标平台是否有 log 或 exp 函数的代码。当然，几乎每个平台都有这些函数；但是对于 tutorial 工程，我们假设它们并非如此普遍。如果该平台有 log 函数，那么我们会在 mysqrt 函数中使用它去计算平方根。我们首先在顶层 CMakeLists 文件中使用宏 CheckFunctionExists.cmake 测试这些函数的可用性：

+ View Code

下一步，如果 CMake 在对应平台上找到了它们，我们修改 TutorialConfig.h.in 来定义这些值；如下：

```
// 该平台提供 exp 和 log 函数吗?
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

这些 log 和 exp 函数的测试要在 TutorialConfig.h 的 configure_file 命令之前被处理，这一点很重要。最后，在 mysqrt 函数中，如果 log 和 exp 在当前系统上可用的话，我们可以提供一个基于它们的可选的实现：

```
1 // 如果我们有 log 和 exp 两个函数，那么使用它们<br>#if defined (HAVE_LOG) && define
2     result = exp(log(x)*0.5);
3 #else // 否则使用替代方法
```

添加一个生成文件以及生成器（步骤 5）

在本节，我们会展示你应该怎样向一个应用程序的构建过程中添加一个生成的源文件。在本范例中，我们会创建一个预先计算出的平方根表作为构建过程的一部分。MathFunctions 子路径下，一个新的 MakeTable.cxx 源文件来做这件事。

```
1 // 一个简单的用于构建平方根表的程序
2 #include <stdio.h>
```

```
3   #include <stdlib.h><br>#include <math.h>
4
5   int  main (int  argc, char  *argv[])
6   {
7       int  i;
8       double  result;
9
10      // 确保有足够多的参数
11      if  (argc < 2)
12          {
13          return  1;
14          }
15
16      // 打开输出文件
17      FILE  *fout = fopen(argv[1],"w");
18      if  (!fout)
19          {
20          return  1;
21          }
22
23      // 创建一个带有平方根表的源文件<br>   fprintf(fout,"double sqrtTable[] = {\
24      for  (i = 0; i < 10; ++i)
25          {
26          result = sqrt(static_cast<double>(i));
27          fprintf(fout,"%g,\n",result);
28          }
29
30      // 该表以 0 结尾
31      fprintf(fout,"0};\n");
32      fclose(fout);
33      return  0;
34  }
```

注意到这个表是由合法的 C++代码生成的，并且被写入的输出文件的名字是作为一个参数输入的。下一步是将合适的命令添加到 MathFunction 的 CMakeLists 文件中，来构建 MakeTable 可执行文件，然后运行它，作为构建过程的一部分。完成这几步，需要少数的几个命令，如下所示：

```
1   # 首先，我们添加生成该表的可执行文件<br>add_executable(MakeTable MakeTable.cxx)
2   # 然后添加该命令来生成源文件
3   add_custom_command (
4       OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
5       COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
6       DEPENDS MakeTable
7       )
```

```
8
9   # 为包含文件，向搜索路径中添加二进制树路径
10  include_directories( ${CMAKE_CURRENT_BINARY_DIR} )
11  <br># 添加 main 库
12  add_library(MathFunctions mysqrt.cxx ${CMAKE_CURRENT_BINARY_DIR}/Table.h   )
```

首先，MakeTable 的可执行文件也和其他被加入的文件一样被加入。然后，我们添加一个自定义命令来指定如何通过运行 MakeTable 来生成 Table.h。这是通过将生成 Table.h 增加到 MathFunctions 库的源文件列表中来实现的。我们还必须增加当前的二进制路径到包含路径的清单中，这样 Table.h 可以被找到并且可以被 mysqrt.cxx 所包含。当该工程被构建后，它首先会构建 MakeTable 可执行文件。然后它会运行 MakeTable 来生成 Table.h 文件。最后，它会编译 mysqrt.cxx（其中包含 Table.h）来生成 MathFunctions 库。到目前为止，拥有我们添加的完整特性的顶层 CMakeLists 文件看起来像是这样：

```
1   cmake_minimum_required (VERSION 2.6)
2   project (Tutorial)
3
4   # 版本号
5   set (Tutorial_VERSION_MAJOR 1)
6   set (Tutorial_VERSION_MINOR 0)
7
8   # 本系统是否提供 log 和 exp 函数?
9   include (${CMAKE_ROOT}/Modules/CheckFunctionExists.cmake)
10
11  check_function_exists (log  HAVE_LOG)
12  check_function_exists (exp  HAVE_EXP)
13
14  # 我们应该使用自己的 math 函数吗?
15  option(USE_MYMATH
16      "Use tutorial provided math implementation"  ON)
17
18  # 配置一个头文件来向源代码传递一些 CMake 设置。
19  configure_file (
20      "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
21      "${PROJECT_BINARY_DIR}/TutorialConfig.h"
22      )
23
24  # 为包含文件的搜索路径添加二进制树，这样才能发现 TutorialConfig.h 头文件。
25  include_directories ("${PROJECT_BINARY_DIR}")
26
27  # 添加 MathFunctions 库吗?
28  if  (USE_MYMATH)
29      include_directories ("${PROJECT_SOURCE_DIR}/MathFunctions")
30      add_subdirectory (MathFunctions)
```

```
    set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)

# 添加可执行文件
add_executable (Tutorial tutorial.cxx)
target_link_libraries (Tutorial   ${EXTRA_LIBS})

# 添加安装的目标
install (TARGETS Tutorial DESTINATION bin)
install (FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
                  DESTINATION include)

# 测试 1 ： 应用程序可以运行吗?
add_test (TutorialRuns Tutorial 25)

# 测试 2 ：  使用信息可用吗?
add_test (TutorialUsage Tutorial)
set_tests_properties (TutorialUsage
    PROPERTIES
    PASS_REGULAR_EXPRESSION "Usage:.*number"
    )

# 定义一个可以简化引入测试过程的宏
macro (do_test arg result)
    add_test (TutorialComp${arg} Tutorial ${arg})
    set_tests_properties (TutorialComp${arg}
        PROPERTIES PASS_REGULAR_EXPRESSION ${result}
        )
endmacro (do_test)

# do a bunch of result based tests
# 执行一系列基于结果的测试
do_test (4 "4 is 2")
do_test (9 "9 is 3")
do_test (5 "5 is 2.236")<br>do_test (7 "7 is 2.645")
do_test (25 "25 is 5")
do_test (-25 "-25 is 0")
do_test (0.0001 "0.0001 is 0.01")
```

TutorialConfig.h 文件看起来像是这样：

```
// Tutorial 的配置选项与设置如下
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
#cmakedefine USE_MYMATH
```

```
// 该平台提供 exp 和 log 函数吗?
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

然后，MathFunctions 工程的 CMakeLists 文件看起来像是这样：

+ View Code

构建一个安装器（步骤 6）

下一步假设我们想要向其他人分发我们的工程，这样他们就可以使用它。我们想同时提供在许多不同平台上的源代码和二进制文档发行版。这与之前我们在"安装与测试（步骤 3）"做过的安装有一点不同，那里我们仅仅安装我们从源码中构建出来的二进制文件。在本例子中，我们会构建支持二进制安装以及类似于 cygwin，debian，RPM 等具有包管理特性的安装包。为了完成这个目标，我们会使用 CPack 来创建 Packaging with CPack 一章中描述的特定平台的安装器。

```
1  # 构建一个 CPack 驱动的安装包
2  include (InstallRequiredSystemLibraries)
3  set (CPACK_RESOURCE_FILE_LICENSE
4           "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
5  set (CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
6  set (CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
7  include (CPack)
```

需要做的全部事情就这些。我们以包含 InstallRequiredSystemLibraries 开始。这个模块将会包含许多在当前平台上，当前工程需要的运行时库。第一步我们将一些 CPack 变量设置为保存本工程的许可证和版本信息的位置。版本信息使用了我们在本指南中先前设置的变量。最后，我们要包含 CPack 模块，它会使用这些变量以及你所处的系统的一些别的属性，然后来设置一个安装器。下一步是以通常的方式构建该工程然后随后运行 CPack。如果要构建一个二进制发行包，你应该运行：

```
1  cpack -C CPackConfig.cmake
```

为了创建一个源代码发行版，你应该键入：

```
1  cpack -C CPackSourceConfig.cmake
```

增加对 Dashboard 的支持（步骤 7）

增加对向一个 dashboard 提交我们的测试结果的功能的支持非常简单。我们在本指南的先前步骤中已经定义了我们工程中的许多测试样例。我们仅仅需要运行这些测试样例然后将它们提交到 dashboard 即可。为了包含对 dashboards 的支持，我们需要在顶层 CMakeLists 文件中包含 CTest 模块。

```
1  # 支持 dashboard 脚本
2  include (CTest)
```

我们也可以创建一个 CTestConfig.cmake 文件，在其中来指定该 dashboard 的工程名。

```
1 set (CTEST_PROJECT_NAME "Tutorial")
```

CTest 将会在运行期间读取这个文件。为了创建一个简单的 dashboard，你可以在你的工程下运行 CMake，然后切换到二进制树，然后运行 ctest -DExperimental. 你的 dashboard 将会被更新到 Kitware 的公共 dashboard.