**[SirDigit](http://www.cnblogs.com/coderfenghc/)**

[**CMake 手册详解（一）**](http://www.cnblogs.com/coderfenghc/archive/2012/06/16/CMake_ch_01.html)

      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:-)。

**CMake2.8.3 主索引**

* **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。

=========================================================================

好了，今天先翻到这里。开饭喽~~

[**SirDigit**](http://www.cnblogs.com/coderfenghc/)

[**CMake 手册详解（二）**](http://www.cnblogs.com/coderfenghc/archive/2012/06/17/2552899.html)

书接上回:-)

====================================================

* --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。

下一章将介绍“生成器”。

[**SirDigit**](http://www.cnblogs.com/coderfenghc/)

[**CMake 手册详解（三）**](http://www.cnblogs.com/coderfenghc/archive/2012/06/18/2553964.html)

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:-)

=================================================

**生成器**

　　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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake 手册详解（四）](http://www.cnblogs.com/coderfenghc/archive/2012/06/19/2555388.html)

       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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake 手册详解（五）](http://www.cnblogs.com/coderfenghc/archive/2012/06/20/2556996.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake 手册详解（六）](http://www.cnblogs.com/coderfenghc/archive/2012/06/23/2559603.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake 手册详解（七）](http://www.cnblogs.com/coderfenghc/archive/2012/06/25/2562778.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake 手册详解（八）](http://www.cnblogs.com/coderfenghc/archive/2012/07/04/2576862.html)

上周正在准备转正答辩，手册翻译的事情暂时搁浅了一段时间；现在继续《CMake手册详解》之第八篇：

=======================================================

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （九）](http://www.cnblogs.com/coderfenghc/archive/2012/07/05/2578451.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十）](http://www.cnblogs.com/coderfenghc/archive/2012/07/08/2581734.html)

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。）

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十一）](http://www.cnblogs.com/coderfenghc/archive/2012/07/11/2585616.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十二）](http://www.cnblogs.com/coderfenghc/archive/2012/07/14/2591135.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十三）](http://www.cnblogs.com/coderfenghc/archive/2012/07/15/2592758.html)

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。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十四）](http://www.cnblogs.com/coderfenghc/archive/2012/07/19/2599988.html)

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。

===========================分==割==线=========================

最近工作比较忙，所以翻译没以前那么勤了，翻译的词句也只是保证能够勉强看懂而已。以后有时间了一定回过头来认真修订下，统一风格和措辞。

# [SirDigit](http://www.cnblogs.com/coderfenghc/)

## [CMake手册详解 （十五）](http://www.cnblogs.com/coderfenghc/archive/2012/07/22/2604041.html)

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。