# 精通cmake

1. 为什么用cmake

1.1 cmake的历史

1.2 为什么不使用Autoconf？

1.3 为什么不使用 JAM,qmake,SCons,或者ANT？

1.4 为什么不自己写脚本？

1.5 CMake能运行在什么平台上？

1. 开始

2.1 在你的电脑上(Unix和苹果系统，视窗系统)获取并安装CMake。

2.2 自己编译CMake

2.3 CMake基本用法和语法

2.4 Hello World

2.5 如何运行CMake？

使用Qt界面

使用ccmake界面

在命令行运行CMake

为CMake指定编译器

依赖检查

2.6 编辑CMakeLists文件

2.7 为CMake设定初始值

2.8 构建你的工程

1. 关键概念

3.1 主结构

3.2 目标

3.3 源文件

3.4 目录，生成器，测试和属性

3.5 变量和缓存实体

3.6 构建配置

1. 编写CMakeLists文件

4.1 CMake语法

4.2 基本命令

4.3 控制流

4.4 正则表达式

4.5 检查CMake版本

4.6 使用组件

使用SWIG组件

使用Qt组件

使用FLTK组件

4.7 策略

更新工程到新版本的CMake

4.8 链接程序库

通过某个目标的优化或Debug程序库

4.9 共享程序库和可加载组件

4.10共享程序库版本控制

4.11 安装文件

安装依赖共享库

4.12 高级命令

1. 系统检测

5.1 使用头文件和程序库

5.2 系统属性

5.3 查找包

5.4 内置组建查找

5.5 如何传递编译参数

5.6 如何配置头文件

5.7 生成CMake包配置文件

1. 定制命令及目标

6.1 可移植的定制命令

6.2 对目标使用add\_custom\_command

当一个可执行文件构建好以后如何拷贝？

6.3 使用add\_custom\_command创建文件

使用可执行文件构建源文件

6.4 添加定制目标

6.5 指定依赖和输出

6.6 当对于某个输出没有对应规则

单个命令产生多个输出

由不同命令产生相同输出

1. 转换现有构建系统到CMake

7.1 源文件目录结构

7.2 构建目录

7.3 使用CMake命令转换工程

7.4 转换UNIX Makefiles

7.5 转换基于Autoconf的工程

7.6 转换基于windows工程

1. 使用CMake交叉编译

8.1 工具链文件

查找外部程序库，程序，和其他文件

8.2 系统检测

使用编译检测

8.3 运行工程中的可执行文件

8.4 交叉编译Hello World

8.5 为微控制器交叉编译

8.6 对已存在工程交叉编译

8.7 对复杂工程交叉编译-VTK

8.8 提示及小技巧

1. 使用CPACK打包

9.1 CPack基础

当包含CPack.cmake时，发生了什么？

添加定制的CPack选项

使用CPack添加选项

9.2 CPack源码包

9.3 CPack安装命令

9.4 视窗系统的CPack安装器NSIS

CMake为NSIS使用CPack变量

在开始菜单创建视窗系统快捷方式

高级的NSIS CPack选项

设置文件扩展关联到NSIS

安装微软运行时库

CPack组件安装支持

9.5 Cygwin中的CPack安装

9.6 Mac OSX PackageMaker下的CPack

9.7 Mac OSX拖拽下的CPack

9.8 MAX OSX X11 Applications下的CPack

9.9 Debian下的CPack包

9.10 CPack RPM包

9.11 CPack文件

1. 使用CMake进行自动化测试

10.1 使用CMake，CTest和CDash测试

10.2 CMake如何帮助测试

10.3 额外的测试属性项

10.4 使用CTest测试

10.5 使用CTest驱动复杂测试

10.6 处理大量测试

10.7 产生测试主控面板

为工程添加测试主控面板

安装设置客户端

10.8 为工程定制主控面板

主控面板的子任务设置

过滤错误和警告

为主控面板添加注释

10.9 设置自动的主控面板客户端

设置持续的主控面板

CTest脚本中的变量

10.10 高级CTest脚本

传统CTest脚本中的限制

扩展的CTest脚本

10.11 设置主控面板服务器

CDash服务器

高级服务器管理

构建组

电子邮件

地址

图表

对构建添加注释

日志

测试时机

移动支持

备份CDash

升级CDash

CDash维护

10.12 子工程

使用ctest\_submit提交PARTS和FILES

将你的工程划分为若干子工程

1. 移植CMake工程到新平台和新语言

11.1 确定系统

11.2 设置语言支持

11.3 移植到新平台

11.4 添加新的语言

11.5 管理变量列表

生成标签变量

语言特定信息

11.6 编译器和平台的例子

Como编译器

Borland编译器

11.7 扩展CMake

创建加载的命令

使用加载的命令

1. 教程

12.1 基础（第一步）

添加版本号和配置头文件

12.2 添加程序库（第二步）

12.3 安装和测试（第三步）

12.4 添加系统内省（第四步）

12.5 添加生成的文件和生成器（第五步）

12.6 构建安装器（第六步）

12.7 添加主控面板的支持（第七步）

附件A 变量

改变行为的变量

描述系统的变量

关于语言的变量

控制构建的变量

提供信息的变量

附件B 命令行相关

CMake命令行选项

CMake生成器

CTest命令行选项

CPack命令行选项

CPack生成器

附件C列出文件命令

当前命令

兼容命令

附件D挑选的组件

CMake组件

附件E属性

全局属性项

目录属性项

目标属性项

测试属性项

源文件属性项

缓存属性

附件F策略

# 第一章

Why CMake?

为什么使用CMake？

If you have ever maintained the build and installation process for a software package, you will be interested in CMake. CMake is an open source build manager for software projects that allows developers to specify build parameters in a simple portable text ﬁle format. This ﬁle is then used by CMake to generate project ﬁles for native build tools including Integrated Development Environments such as Microsoft Visual Studio or Apple’s Xcode, as well as UNIX, Linux, NMake, and Borland style Makeﬁles. CMake handles the difﬁcult aspects of building software such as cross platform builds, system introspection, and user customized builds, in a simple manner that allows users to easily tailor builds for complex hardware and software systems.

如果你曾经维护过软件包的构建和安装过程，你就会对CMake感兴趣。CMake是一个开源的软件工程的构建管理工具，它允许开发者在一个可移植的文本格式的文件中指定构建参数。CMake使用这个文件生成使用开发环境中集成的本地构建工具的工程文件，比如微软的Visual Studio或者苹果的Xcode，在UNIX，Linux系统中，生成NMake，Borland风格的Makefile文件。CMake处理构建软件中复杂的部分比如（跨平台构建 （交叉编译）），系统内省，和定制构建，只用简单的操作就可以方便用户剪裁对应复杂硬件和软件系统的构建。

For any project, and especially cross platform projects, there is a need for a uniﬁed build system. Many projects today ship with both a UNIX Makeﬁle (or Makeﬁle.in) and a Microsoﬁ Visual Studio workspace. This requires that developers constantly try to keep both build systems up to date and consistent with each other. To target additional build systems such as Borland or Xcode requires even more custom copies of these ﬁles, creating an even bigger problem. This problem is compounded if you try to support optional components, such as including JPEG support if libjpeg is available on the system. CMake solves this by consolidating these different operations into one simple easy to understand ﬁle format.

对于任意工程，特别是跨平台工程，需要一个统一的构建系统。现在，许多工程都提供UNIX Makefile和微软Visual Studio构建环境。这就需要开发人员持续维护更新两套构建系统并使它们保持一致。如果要添加对应Borland或者Xcode的构建系统，就需要修改更多的文件，这将会产生更多的问题。如果要支持可选的组件，比如如果系统有libjpeg库，就包含JPEG，这将会使得问题变得更加复杂。CMake通过将这些不同的操作编写到一个易于理解的文件中解决这个问题。

If you have multiple developers working on a project, or multiple target platforms, then the software will have to be built on more than one computer. Given the wide range of installed software and custom options that are involved with setting up a modern computer, the chances are that two computers running the same OS will be slightly different. CMake provides many beneﬁts for single platform multi-machine development environments including:

如果你的工程有多个开发人员，或者多个目标平台，那么你的软件会在不止一台电脑上构建。因为不同的电脑安装的软件不同，配置不同，那么不同的电脑运行相同的操作系统也会有些许不同，CMake对于这种单一平台多个电脑的运行环境提供一些帮助。这包括：

* The ability to automatically search for programs, libraries, and header files that may be required by the software being built. This includes the ability to consider environment variables and Window’s registry settings when searching.
* 自动查找软件构件所需程序，程序库，头文件。这包含查找系统环境变量和视窗系统注册表的能力。
* The ability to build in a directory tree outside of the source tree, This is a useful feature found on many UNIX platforms; CMake provides this feature on Windows as well. This allows a developer to remove an entire build directory without fear of removing source ﬁles.
* 在源代码树之外的文件夹中构建的能力，这在UNIX平台下是很有用的特性；CMake也为视窗系统提供这种特性。也就允许开发者删除整个构建文件夹而不用担心误删源文件。
* The ability to create complex custom commands for automatically generated ﬁles such as Qt's moc (qt.nokia.com), The Insight Toolkit’s CABLE wrappers(publickitware.com/Cable/HTML/Index.html) and SWIG (www.swig.org) wrapper generators. These commands are used to generate new source ﬁles during the build process that are in turn compiled into the software.
* \*\*\*\*todo\*\*\*\*
* The ability to select optional components at conﬁguration time. For example, several of VTK’s libraries are optional, and CMake provides an easy way for users to select which libraries are built.
* 在配置时选择可选组建的能力。比如一些VTK库是可选的，CMake提供一种简单的方式让用户选择构建哪个程序库。
* The ability to automatically generate workspaces and projects from a simple text ﬁle. This can be very handy for systems that have many programs or test cases, each of which requires a separate project ﬁle, typically a tedious manual process to create using an IDE.
* 通过简单的文本文件自动生成工作区和工程的能力。这对于那些有很多程序和测试用例，每个程序和测试用例需要单独的工程文件的工程是非常方便的，一般情况下，需要IDE手动生成这些工程。
* The ability to easily switch between static and shared builds. CMake knows how to create shared libraries and modules on all platforms supported. Complicated platform-speciﬁc linker ﬂags are handled, and advanced features like built in run time search paths for shared libraries are supported on many UNIX systems.
* 方便切换创建静态库或共享库的能力。CMake知道如何在所支持的系统上创建共享库和模块。能够处理复杂的平台相关的链接标志位，和一些高级特性，比如许多UNIX系统都支持的内嵌的对于共享库路径的运行时查找。
* Automatic generation of ﬁle dependencies and support for parallel builds on most platforms.

When developing cross platform software, CMake provides a number of additional features:

* 在大多数平台支持文件依赖及并行构建。

当开发跨平台软件时，CMake提供了一系列额外的特性：

* The ability to test for machine byte order and other hardware speciﬁc characteristics.
* 检测机器字节顺序和其他硬件特性的能力。
* A single set of build conﬁguration ﬁles that work on all platforms. This avoids the problem of developers having to maintain the same information in several different formats inside a project.
* 适用于全平台的一套构建配置文件。这可以避免开发者维护记录相同信息的不同格式的文件。
* Support for building shared libraries on all platforms that support it.
* 在支持的平台上构建共享库的能力。
* The ability to conﬁgure ﬁles with system dependent information such as the location of data ﬁles and other information. CMake can create header ﬁles that contain information such as paths to data ﬁles and other information in the form of #deﬁne macros. System speciﬁc ﬂags can also be placed in conﬁgured header ﬁles. This has advantages over command line -D options to the compiler because it allows other build systems to use the CMake built library without having to specify the exact same command line options used during the build.
* 使用系统相关信息如数据文件的位置或其他信息来配置文件的能力。CMake能够生成包含数据文件路径或其他信息的头文件（以#define宏的形式）比命令行中传递给编译器-D可选项要好，因为他允许其他的构建系统使用CMake的构建库而不必在构建中指明该命令行可选项。

1.1 The History of CMake

1.1 CMake的历史

CMake development began in 1999 as part of the Insight Toolkit (ITK, www.itk.org) funded by the US National Library of Medicine. ITK is a large software project that works on many platforms and can interact with many other software packages. To support this, a powerful, yet easy to use, build tool was required. Having worked with build systems for large projects in the past, the developers designed CMake to address these needs. Since then CMake has continuously grown in popularity, with many projects and developers adopting it for its ease of use and ﬂexibility. Since 1999 CMake has been under active development and has matured to the point where it is a proven solution for a wide range of build issues. The most telling example of this is the successful adoption of CMake as the build system of the K Desktop Environment (KDE), arguably the largest open source software project in existence.

CMake项目开始于1999年，它是作为由美国国家医学图书馆赞助的项目（Insight Toolkit ITK，www.itk.com）的一部分出现的。ITK是一个大型的软件工程，能够在许多不同的平台上运行，可以和许多其他软件配合使用。为了支持这一特性，就需要一个强大的易于使用的构建工具。基于使用过往的大型工程的构建工具的经验，开发者们按上述需求设计CMake。之后，CMake逐渐流行，许多工程和开发者因其易用性和灵活性接受了它。之后CMake不断发展，在解决广泛的构建方面的问题上日渐成熟。一个非常有说服力的例子是CMake作为构建系统在K Desktop Environment（KDE）这个庞大的开源软件项目中被使用至今。

1.2 Why Not Use Autoconf?

1.2 为什么不使用Autoconf？

Before developing CMake its authors had experience with the existing set of available tools. Autoconf combined with automake provides some of the same functionality as CMake, but to use these tools on a Windows platform requires the installation of many additional tools not found natively on a Windows box. In addition to requiring a host of tools, autoconf can be difﬁcult to use or extend and impossible for some tasks that are easy in CMake. Even if you do get autoconf and its required environment running on your system, it generates Makeﬁles that will force users to the command line. CMake on the other hand provides a choice, allowing developers to generate project ﬁles that can be used directly from the IDE to which Windows and Xcode developers are accustomed.

在开发CMake之前，它的作者使用过一系列其他的构建工具。Autoconf连同automake部分提供了和CMake相同的功能，但是要是在Windows平台使用这些工具，就需要安装额外的工具。除了依赖其他工具，autoconf难于使用和扩展，一些在CMake中易于使用的tasks（任务），在autoconf中无法使用。即使你你安装了autoconf依赖的环境，并在你的系统中运行了autoconf，它所生成的Makefiles强迫用户使用命令行执行。CMake提供了另外一种选择，它允许开发者生成能够直接被IDE导入的工程文件，这些IDE是开发者所熟悉的。

While autoconf supports user speciﬁed options, it does not support dependent options where one option depends on some other property or selection. For example, in CMake you could have a user option to enable multithreading be dependent on ﬁrst determining if the user’s system has multithreading support. CMake provides an interactive user interface, making it easy for the user to see what options are available and how to set them.

虽然autoconf支持用户指定选项，但它不支持依赖于其他属性或选择的选项。比如，在CMake中你可以指定一个用户选项去开启多线程，这个选项依赖于检测用户系统是否支持多线程的结果。CMake提供了一个交互界面，方便使用者确认哪些选项是开启的，以及如何设置这些选项。

For UNIX users, CMake also provides automated dependency generation that is not done directly by autoconf. CMake’s simple input format is also easier to read and maintain than a combination of Makeﬁle.in and conﬁgure in ﬁles. The ability of CMake to remember and chain library dependency information has no equivalent in autoconf/automake.

对于UNIX的用户，CMake也支持自动依赖生成，而autoconf并不直接支持。CMake的配置文件也比Makefile.in和configure文件易于理解和维护。CMake记录和链接程序库依赖信息的能力也不是autoconf/automake可比的。

1.3 Why Not Use JAM, qmake, SCons, or ANT?

1.3 为什么不使用JAM，qmake，SCons或ANT

Other tools such as ANT, qmake, SCons, and JAM have taken different approaches to solving these problems and they have helped us to shape CMake. Of the four, qmake, is the most similar to CMake although it lacks much of the system interrogation that CMake provides. Qmake‘s input format more is closely related to a traditional Makeﬁle. ANT, JAM and SCons are also cross-platform although they do not support generating native project ﬁles. They do break away from the traditional Makeﬁle oriented input with ANT using XML, JAM using its own language, and SCons using Python. A number of these tools run the compiler directly, as opposed to letting the system’s build process perform that task. Many of these tools require other tools such as Python or Java to be installed before they will work.

其他工具比如ANT，qmake，SCons，和JAM通过不同的方式解决了这些问题，这帮助我们更加了解CMake。这四个工具中，qmake和CMake最像，虽然它不支持CMake提供的那种系统查询功能。Qmake的输入格式类似于传统的Makefile。ANT，JAM和SCons也是跨平台的，虽然它们不支持生成工程文件。它们和传统的输入向Makefile完全不同，ANT使用XML，JAM使用自己的语音，SCons使用Python。其中很多工具直接运行编译器，这和构建过程执行任务（task）截然不同。这些工具需要安装其他软件，如Python或者Java。

1.4 Why Not Script It Yourself?

1.4 为什么不自己写脚本

Some projects use existing scripting languages such as Perl or Python to conﬁgure build processes. Although similar functionality can be achieved with systems like this, over-use of tools can make the build process more of an Easter egg hunt than a simple—to-use build system. When building your software package users are forced to ﬁnd and install version 4.3.2 of this, and 3.2.4 of that, before they can even start the build process. To avoid that problem, it was decided that CMake would require no more tools than the software it was being used to build would require. At a minimum using CMake requires a C compiler, that compiler's native build tools, and a CMake executable. CMake was written in C++, requires only a C++ compiler to build and precompiled binaries are available for most systems. Scripting it yourself also typically means you will not be generating native Xcode or Visual Studio workspaces, making Mac and Windows builds limited.

有些工程使用脚本语言如Perl或者Python来配置构建过程。尽管也能实现CMake类似的功能，但是过度使用工具会造成构建过程更加复杂。当构建软件包时，用户必须查找并安装对应版本的工具，然后才能开始构建过程。为了避免这一问题，CMake被设计成只依赖软件构建所必需的工具。最小情况下CMake只依赖C编译器，和CMake可执行文件。CMake是由C++编写的，对于大多数系统，只需要C++编译器和预编译二进制文件。自己编写脚本文件意味着无法生成Xcode或Visual Studio工程文件，这在Mac和Windows系统下是受限制的。

1.5 On What Platforms Does CMake Run?

1.5 CMake能运行在哪些平台上？

CMake runs on a wide variety of platforms including Microsoft Windows, Apple Mac OS X, and most UNIX or UNIX-like platforms. At the time of the writing of this book CMake was tested nightly on the following platforms: Windows 98/2000/XP/Vista/7, AIX, HPUX, IRIX, Linux, Mac OS X, Solaris, OSF, QNX, CYGWIN, MinGW, and FreeBSD. You can check www.cmake.org for a current list of tested platforms.

CMake可以运行在广泛的平台上，包括微软视窗系统，苹果Mac OS X，和大多数UNIX或类UNIX系统。在写本书的时候，CMake每日测试版可以运行在下列平台：Windows 98/2000/XP/Vista/7, AIX, HPUX, IRIX, Linux, Mac OS X, Solaris, OSF, QNX, CYGWIN, MinGW, and FreeBSD。你可以访问www.cmake.org查看当前适配的平台。

Likewise, CMake supports most common compilers. It supports the GNU compiler on all CMake supported platforms. Other tested compilers include Visual Studio 6 through 10, Intel C, SGI CC, Mips Pro, Borland, Sun CC and HP aCC. CMake should work for most UNIX-style compilers out of the box. If the compiler takes arguments in a strange way, then see the section Porting CMake to New Platform on page 241 for information on how to customize CMake for a new compiler.

同样的，CMake支持大多数编译器。在所有适配的平台上CMake都支持GNU编译器。其他测试过的编译器包括Visual Studio 6-10，Intel C, SGI CC, Mips Pro, Borland, Sun CC and HP aCC。CMake还应该支持大多数UNIX风格的编译器。如果编译器以古怪的方式接受参数，请参考第241页的Porting CMake to New Platform章节获得如何为新编译器定制CMake的信息。

Chapter 2

第二章

Getting Started

新手入门

2.1 Getting and Installing CMake on Your Computer

2.1 在你的计算机中获取并安装CMake

Before using CMake you will need to install or build the CMake binaries on your system. On many systems you may ﬁnd that CMake is already installed, or is available for install with the standard package manager tool for the system. Cygwin, Debian, FreeBSD, Mac OS X Fink, and many others all have CMake distributions. If your system does not have a CMake package, you can ﬁnd CMake precompiled for most common architectures at www.cmake.org. If you do not ﬁnd binaries for your system precompiled, then you can build CMake from source. To build CMake you will need a modern C++ compiler.

在使用CMake之前你需要在你的系统中安装或构建CMake二进制文件。你会发现在许多系统中已经安装了CMake，或者可以通过系统的标准包管理器安装。Cygwin, Debian, FreeBSD, Mac OS X Fink，和许多其他的系统都有对应的CMake版本。如果你的系统没有CMake软件包，你能在www.cmake.org找到对应大多数架构的预编译好的CMake。如果没有找到对应你的系统的CMake版本，你也可以通过源代码构建CMake。如果这样你需要一个现代的编译器。

If your system provides CMake as one of its standard packages, follow your system’s package installation instructions. If your system does not have CMake, or has an out of date version of CMake, you can download precompiled binaries from www.cmake.org. The binaries from www.cmake.org come in the form of a compressed tar ﬁle. The tar ﬁle contains a README ﬁle and an enclosed tar ﬁle. The README ﬁle contains a manifest of the ﬁles contained in the enclosed tar ﬁle, and some instructions. To install, simply extract the enclosed tar ﬁle into a destination directory (typically /usr/local). However, it can be any directory, and does not require root privileges for installation.

如果你的系统提供CMake作为标准包，按照你的系统的包安装指令操作。如果你的系统没有CMake，或者CMake的版本太老，你可以从www.cmake.org下载预编译好的二进制文件。这些二进制文件被达成了一个tar格式的压缩包。其中包括一个README文件。和tar文件。README文件描述了在tar文件中的文件的清单文件，和一些指令。要安装CMake，解压缩tar文件到目标目录（一般是/usr/local）。也可以是任意目录，这并不需要root权限来安装。

For Windows CMake has a NullSoﬁ install ﬁle available for download from www.cmake.org. To install this ﬁle, simply run the executable on the windows machine on which you want to install CMake. You will be able to run CMake from the Start Menu after it is installed.

对于视窗系统，CMake在www.cmake.org提供一个NullSofi安装文件，只需在想要安装的视窗系统中执行这个可执行文件，即可安装。装好后，你可以从开始菜单中执行CMake。

2.2 Building CMake Yourself

2.2 自己构建CMake

If binaries are not available for your system, or if binaries are not available for the version of CMake you wish to use, you can build CMake from the source code. You can obtain the CMake source code by following the instructions at www.cmake.org. Once you have the source code it can be built in two different ways. If you have a version of CMake on your system you can use it to build other versions of CMake. Generally the current development version of CMake can always be built from the previous release of CMake. This is how new versions of CMake are built on most Windows systems.

如果没有对应你的系统的二进制文件，或者二进制文件的版本号和你希望使用的CMake版本号不一致，你可以通过源码构建CMake。你可以在www.cmake.org按指令下载源码。一旦拥有的源码，你可以通过两种方式构建CMake。如果你的系统中安装了CMake，你可以使用它构建其他版本的CMake。一般来说，当前开发版的CMake都是由之前发布的CMake构建的。这也是大多数视窗系统CMake的构建方式。

The second way to build CMake is by running its bootstrap build script. To do this you change directory into your CMake source directory and type

第二种构建方式是执行CMake的引导构建脚本。要这样做，你需要切换到CMake源码目录，并执行

./bootstrap

make

make install

The make install step is optional since CMake can run directly from the build directory if desired. On UNIX, if you are not using the GNU C++ compiler, you need to tell the bootstrap script which compiler you want to use. This is done by setting the environment variable CXX before running bootstrap. If you need to use any special ﬂags with your compiler, set the CXXFLAGS environment variable. For example, on the SGI with the 7.3X compiler, you would build CMake like this:

第三步是可选的，因为CMake能够直接在构建目录中执行。在UNIX系统中，如果你没有使用CNUC++编译器，你需要告诉引导脚本你想要使用的编译器。这通过在执行引导脚本前设置环境变量CXX来实现。如果你需要使用特殊的编译器标志位，设置CXXFLAGS环境变量。比如说，在SGI中使用7.3x版本编译器。你需要这样构建CMake：

cd CMake

(setenv CXX CC; setenv CXXFLAGS "—LANsttd"; ./bootstrap)

make

make install

2.3 Basic CMake Usage and Syntax

2.3 CMake的基本用法及语法

Using CMake is simple. The build process is controlled by creating one or more CMakeLists ﬁles (actually CMakeLists.txt but this guide will leave off the extension in most cases) in each of the directories that make up a project. The CMakeLists ﬁles should contain the project description in CMake's simple language. The language is expressed as a series of commands. Each command is evaluated in the order that it appears in the CMakeLists ﬁle. The commands have the form

command (args...)

使用CMake很简单。构建过程由工程每个文件夹中的一个或多个CMakeLists（事实上是CMakeLists.txt文件，不过大多数情况下省略扩展名）文件控制。CMakeLists文件应该包含由CMake的简单语言写的工程描述。这个语言由一系列的命令组成。每条命令以它在CMakeLists文件中的顺序执行。命令以

command(args...)的格式书写。

where command is the name of the command, and args is a white-space separated list of arguments. (Arguments with embedded white-space should be double quoted.) CMake is case insensitive to command names as of version 2.2. So where you see command you could use COMMAND or Command instead. Older versions of CMake only accepted uppercase commands.

在这里command指命令的名称，args是由空格分开的参数列表。（参数中包含空格需要用双引号括起来。）CMake从2.2版开始对于命令大小写不敏感。所以你可以看到它的命令可能写成COMMAD或者Command。老版本的CMake只接受大写命令。

CMake supports simple variables that can be either strings or lists of strings. Variables are referenced using a ${VAR} syntax. Multiple arguments can be grouped together into a list using the set command. All other commands expand the lists as if they had been passed into the command with white-space separation. For example, set (Foo a b c) will result in setting the variable Foo to a b c, and if Foo is passed into another command command ($ {Foo} ) it would be equivalent to command (a b c) . If you want to pass a list of arguments to a command as if it were a single argument simply double quote it. For example command ( "$ {Foo} ") would be invoked passing only one argument equivalent to command(

"a b c" ).

CMake支持以字符串或字符串数组列表的形式的简单变量。变量使用${VAR}这样的语法来引用。多个参数可以通过使用set命令聚集成一个列表。当它被以空格分隔传入其他命令时将展开这个列表。比如，set(Foo a b c)会设置变量Foo的值为 a b c，如果Foo被传入其他命令command command(${Foo})就等同于command(a b c)。如果你想要传入参数列表并把它当做一个参数，可以使用双引号括起来。比如command("${Foo}")，这将等同于传入一个参数的command("a b c")。

System environment variables and Windows registry values can be accessed directly in CMake. To access system environment variables the syntax $ENV{VAR} is used. CMake can also reference registry entries in many commands using a syntax of the form [HKEYiCURRENTiUSER\\Software\\pathl\\path2;key], where the paths are built from the registry tree and key.

系统环境变量和Windows注册表的值可以被CMake直接访问。在CMake中以$ENV{VAR}的形式获得环境变量的值。在许多命令中，CMake也可以以[HKEYiCURRENTiUSER\\Software\\pathl\\path2;key]的形式引用注册表中的表项，这里paths是指注册表中的路径和键。

2.4 Hello World for CMake

2.4 Hello World

For starters let us consider the simplest possible CMakeLists ﬁle. To compile an executable from one source ﬁle the CMakeLists ﬁle would contain two lines:

作为初学者，我们从一个最简单的CMakeLists文件开始。想要从一个源文件编译成一个可执行文件只需要两行代码：

project (Hello)

add\_executable (Hello Hello.c)

To build the Hello executable you follow the process described in Running CMake (See section 2.5) to generate the Makeﬁles or Microsoft project ﬁles. The project command indicates what the name of the resulting workspace should be and the add\_executable command adds executable an target to the build process. That’s all there is to it for this simple example. If your project requires a few ﬁles it is also quite easy, just modify the add\_executable line as shown below.

要编译可执行文件Hello，你需要按第2.5小节的步骤生成Makefiles或Microsoft工程文件。project命令指明工程（工作区）的名字，add\_executable命令将生成一个可执行文件的目标添加到构建过程。这就是这个简单例子的全部了。如果你的工程有几个源文件也很简单，只需按下面的方式修改add\_executable命令。

add\_executable (Hello Hello.c File2.c File3.c File4.c)

add\_executable is just one of many commands available in CMake. Consider the more

complicated example below.

add\_executable是CMake众多命令中的一条。下面是更复杂的例子。

cmake\_minimum\_required (2.6)

project (HELLO)

set (HELLO\_SRCS Hello.c File2.c File3.c)

if (WIN32)

Set(HELLO\_SRCS ${HELLO\_SRCS} WinSupport.c)

else()

set(HELLO\_SRCS ${HELLO\_SRCS} UnixSupport.c)

endif()

add\_executable (Hello ${HELLO\_SRCS})

# look for the Tcl library

find\_library (TCL\_LIBRARY

NAMES tcl tc184 tcl83 tcl82 tcl80

PATHS /usr/lib /usr/local/lib

)

if (TCL\_LIBRARY)

target\_link\_library (Hello ${TCL\_LIBRARY})

endif()

In this example the set command is used to group together source ﬁles into a list. The if command is used to add either WinSupport.c or UnixSupport.c to this list based on whether or not CMake is running on Windows. Finally, the add\_executable command is used to build the executable with the ﬁles listed in the variable HELLO\_SRCS. The find\_library command looks for the Tcl library under a few different names and in a few different paths. An if command checks if the TCL\_LIBRARY was found and if so adds it to the link line for the Hello executable target. Note the use of the # character to denote a comment line. All characters from the # to the end of the line are considered to be part of the comment.

在这个例子中，set命令将若干源文件组成一个列表，if命令通过判断CMake是否运行在Windows中，选择加入源文件WinSupport.c或UnixSupport.c到列表中。最后add\_executable命令使用变量HELLO\_SRCS列表中的源文件构建可执行文件。find\_library命令在指定名字和路径下查找Tcl程序库。如果if命令判断找到TCL\_LIBRARY，就将它同Hello可执行目标链接。请注意，字符#表示一个注释行。它后面的字符都被认为是注释。

2.5 How to Run CMake?

2.5 如何运行CMake？

Once CMake has been installed on your system, using it to build a project is easy. There are two main directories CMake uses when building a project: the source directory and the binary directory. The source directory is where the source code for your project is located. This is also where the CMakeLists ﬁles will be found. The binary directory is where you want CMake to put the resulting object ﬁles, libraries, and executables. Typically CMake will not write any ﬁles to the source directory, only the binary directory. If you want to you can set the source and binary directories to be the same. This is known as an in-source build, in contrast to an out-of-source build where they are different.

如果已经安装好CMake，使用它构建工程就简单了。当构建工程时，CMake主要使用两个文件目录：源文件目录和二进制文件目录。源文件目录是工程源码所在目录。CMakeLists文件也放在此目录。二进制文件目录是CMake存放结构目标文件，程序库，和可执行文件的目录。一般来说CMake不在源文件目录，而只在二进制目录创建文件。如果你需要，可以将源文件目录和二进制目录设置成一样的。这就是所谓的源码内构建，与之相对的是源码外构建。

CMake supports both in-source and out-of-source builds on all operating systems. This means that you can conﬁgure your build to be completely outside of the source code tree which makes it very easy to remove all of the ﬁles generated by a build. Having the build tree differ from the source tree also makes it easy to support having multiple builds of a single source tree. This is useful when you want to have multiple builds with different options but just one copy of the source code. Now let us consider the specifics of running CMake using its Qt based GUI and command line interfaces.

CMake在所有系统中支持源码内构建和源码外构建。这就意味着你可以配置使它完全在源代码外部构建，从而是删除构建结构变得非常容易。在源代码外部构建也使得支持使用一套源码生成多个构建得到支持。这对于通过选项使用一套源码生成多个构建是很有用的。下面我们来看如何通过基于Qt的交互界面和命令行运行CMake。

Running CMake’s Qt Interface

运行CMake Qt交互界面

CMake includes a Qt based user interface developed by Clinton Stimpson that can be used on most platforms, including UNIX, Mac OS X, and Windows. This interface is included in the CMake source code, but you will need an installation of Qt on your system in order to build it.

CMake包含一个由Clinton Stimpson开发的可以再大多数系统（包括UNIX，Mac和Windows）上运行的基于Qt的交互界面。这个交互界面包含在CMake的源码中，但是要运行这个交互界面，你需要在系统中安装Qt库来构建它。

On Windows the executable is named cmake-gui.exe and should be in your Start menu under Program Files. There may also be a shortcut on your desktop, or if you built CMake from source, it will be in the build directory. For UNIX and Mac users the executable is named cmake-gui and it can be found where you installed the CMake executables.A GUI will appear similar to what is shown in Figure l. The top two entries are the source code and binary directories. They allow you to specify where the source code is for What you want to compile and where the resulting binaries should be placed. You should set these two values ﬁrst. If the binary directory you specify does not exist, it will be created for you. If the binary directory has been conﬁgured by CMake before then it will automatically set the source tree.

在Windows中这个可执行文件是cmake-gui.exe，它应该在你的开始菜单的Program Files项中。在你的桌面上也应该有它的快捷方式，如果你从源代码构建CMake，它也应该在你的构建目录中。对于UNIX和苹果系统的用户，这个可执行文件是cmake-gui，它可以在你的安装目录中被找到。这个图形交互界面和图一类似。顶部是源文件目录和构建目录。你可以指定源文件目录和构建的二进制目录。你应该先设置这两个目录。如果你指定的构建目录不存在，它会为你生成一个。如果你之前配置过CMake的构建目录，它会自动找到对应的源文件目录。

The middle area is where you can specify different options for the build process. More obscure variables may be hidden, but can be seen if you select "Advanced View" from the View pulldown. You can search for values in the middle area by typing all or part of the name into the Search box. This can be handy for ﬁnding speciﬁc settings or options in a large project. The bottom area of the window includes the Conﬁgure and Generate buttons as well as a progress bar and scrollable output window.

在中间部分你可以为构建过程指定不同的可选项。一些隐藏变量不会显示，你可以勾选"Advanced View"选项显示它们。你可以在搜索框中输入关键字搜索选项。这对于在一个庞大的工程中查找指定设置是很有用的。底部区域包含配置和生成按钮，进度条，可一个可滚动的输出窗口。

Once you have speciﬁed the source code and binary directories you should click the Conﬁgure button. This will cause CMake to read in the CMakeLists ﬁles from the source code directory and then update the cache area to display any new options for the project. If you are running cmake-gui for the ﬁrst time on this binary directory it will prompt you to determine what generator you wish to use, as shown in Figure 2. This dialog also presents options for customizing and tweaking the compilers you wish to use for this build.

一旦你指定了源文件目录和构建目录，你需要点击Configure按钮，使CMake读取源文件目录中的CMakeLists文件，更新缓存区域，显示工程的新选项。对应指定的构建目录，如果你是第一次在运行cmake-gui，它会提示你选择使用哪种生成器（如图2所示）。这个对话框也会显示你希望构建时使用的定制和调整编译器的可选项。

After the ﬁrst conﬁgure you can adjust your cache settings if desired and click the Conﬁgure button again. New values that were created by the conﬁgure process will be colored red. To be sure you have seen all possible values you should click Conﬁgure until no values are red and you are happy with all the settings. Once you are done conﬁguring, click the Generate button, this will produce the appropriate ﬁles.

第一次配置完后，如果需要你可以调整缓存设置，并再次点击Configure按钮。配置过程中新的值会被标记成红色。你需要确认所有的选项，直到它们不再显示成红色。一旦配置完成，点击Generate按钮，生成恰当的文件。

It is important that you make sure that your environment is suitable for running cmake-gui. If you are using an IDE such as Visual Studio then your environment will be setup correctly for you. If you are using NMake or MinGW then you need to make sure that the compiler can run from your environment. You can either directly set the required environment variables for your compiler or use a shell in which they are already set. For example, Microsoft Visual Studio has an option on the start menu for creating a Visual Studio Command Prompt. This opens up a command prompt window that has its environment already setup for Visual Studio. You should run cmake-gui from this command prompt if you want to use NMake Makeﬁles. The same approach applies to MinGW, you should run cmake-gui from a MinGW shell that has a working compiler in its path.

确认你的环境适合运行cmake-gui是非常重要的。如果你使用集成开发环境，如VIsual Studio，那么你的环境将被正确设置。如果你使用NMake或者MinGW，那么你需要确认在你的环境中编译器可以运行。你既可以为你的编译器设置所需的环境变量，也可以使用已经设置好环境变量的shell。比如说Microsoft Visual Studio在开始菜单有一个选项生成命令提示符。这会打开一个命令提示符窗口，并为Visual Studio设置好相应的环境变量。如果你想使用NMake Makefiles，你可以在这个命令提示符下运行cmake-gui。同样的，你也可以在MinGW shell中运行cmake-gui，它已经在path中配置好了一个可用的编译器。

When cmake-gui ﬁnishes it will have generated the build ﬁles in the binary directory you speciﬁed. If Visual Studio was selected as the generator, a MSVC workspace (or solution) ﬁle is created. This ﬁle's name is based on the name of the project you speciﬁed in the PROJECT command at the beginning of your CMakeLists ﬁle. For many other generator types, Makeﬁles are generated. The next step in this process is to open the workspace with MSVC. Once open, the project can be built in the normal manner of Microsoft Visual C++. The ALL\_BUILD target can be used to build all of the libraries and executables in the package. If you are using a Makeﬁle build type, then you would build by running make or nmake on the resulting Makeﬁles.

当使用cmake-gui后它会在你指定的构建目录中生成构建文件。如果指定Visual Studio生成器，则生成MSVC工作区文件。这个文件的名字基于你在CMakeLists文件开头使用PROJECT命令指定的工程名。大多数生成器生成Makefiles文件。下一步，使用MSVC打开工作区，然后就可以使用Microsoft Visual C++构建。ALL\_BUILD目标可以被用来构建所有的程序库和可执行文件。如果你使用Makefile构建，需要使用生成的Makefiles文件执行make或者nmake。

Running the ccmake Curses Interface

运行ccmake Curses界面

On most UNIX platforms, if the curses library is supported, CMake provides an executable called ccmake. This interface is a terminal-based text application that is very similar to the Qt based GUT. To run ccmake, change directory (Cd) to the directory where you want the binaries to be placed. This can be the same directory as the source code for what we call in-source builds or it can be a new directory you create. Then run ccmake with the path to the source directory on the command line. For in-source builds use "." for the source directory. This will start the text interface as shown in Figure 3 (in this case the cache variables are from VTK and most are set automatically).

在大多数UNIX平台上，如果支持curses 程序库，CMake提供了一个可执行文件ccmake。这个交互界面是基于终端的文本程序这类似于基于Qt的GUT。要运行ccmake，使用cd命令切换到构建目录。要是使用源码内构建，构建目录可以和源码目录一致。然后运行ccmake，使用源码目录作为参数。对于源码内构建，使用"."指定源码目录。然后会显示如图3所示文本交互界面（这里缓存变量来自于VTK，大部分是自动设置的）。

Brief instructions are displayed in the bottom of the window. If you hit the "c" key, it will conﬁgure the project. You should always conﬁgure after changing values in the cache. To change values, use the arrow keys to select cache entries, and then the enter key to edit them. Boolean values will toggle with the enter key. Once you have set all the values as you like, you can hit the "g" key to generate the Makeﬁles and exit. You can also hit "h" for help, "q" to quit, and "t" to toggle the Viewing of advanced cache entries. Two examples of CMake usage on the UNIX platform follow for a hello world project called Hello. In the ﬁrst example, an in-source build is performed.

在窗口的底部是指令的简要介绍。如果按"c"键，将配置工程。你应该在缓存中修改了值后，重新配置。要修改值，使用方向键选择缓存项，然后点击回车键编辑它们。布尔型的值可以通过回车键切换。一旦你设置好所有的值，按"g"键生成Makefiles并退出。你也可以按"h"键，显示帮助，"q"键退出，"t"键切换高级显示模式。下面是UNIX平台上，两个使用CMake的例子，它们是hello world工程，叫Hello。第一个例子是源码内构建的。

cd Hello

ccmake

make

In the second example, an out-of-source build is performed.

第二个例子是源码外构建。

mkdir Hello-Linux

cd Hello-Linux

ccmake ../Hello

make

Running CMake from the Command Line

使用命令行运行CMake

From the command line, CMake can be run as an interactive question and answer session or as a non-interactive program. To run in interactive mode, just pass the "-i" option to CMake. This will cause CMake to ask you for a value for each entry in the cache file for the project. CMake will provide reasonable defaults, just like it does in the GUI and curses based interfaces. The process stops when there are no longer any more questions to ask. An example of using the interactive mode of CMake is provided below.

在命令行下，CMake可以以问答的交互方式运行，也可以以非交互方式运行。使用"-i"可选项，CMake将以交互方式运行。CMake将会询问工程中缓存文件每项的值。CMake会提供一个合理的默认值，就像基于GUI和curses的交互中那样。当问完所有问题这个过程就结束了。下面是一个使用CMake交互方式的例子。

$ cmake -i eG "NMake Makefiles" ../CMake

Would you like to see advanced options? [No]:

Please wait while cmake processes CMakeLists.txt files....

Variable Name: BUILD\_TESTING

Description: Build the testing tree.

Current Value: ON

New Value (Enter to keep current value):

Variable Name: CMAKE\_INSTALL PREFIX

Description: Install path prefix, prepended onto install

directories.

Current Value: C:/Program Files/CMake

New Value (Enter to keep current value):

Please wait while cmake processes CMakeLists.txt files....

CMake complete, run make to build project.

Using CMake to build a project in non—interactive mode is a simple process if the project has few or no options. For larger projects like VTK, using ccmake, cmake -i, or cmake-gui is recommended. To build a project with a non—interactive CMake, first change directory to where you want the binaries to be placed. For an in-source build you then run cmake . and pass in any options using the -D ﬂag. For out-of-source builds the process is the same except you run cmake and also provide the path to the source code as its argument. Then type make and your project should compile. Some projects will have install targets as well, you can type make install to install them.

如果工程只有若干选择项或者没有，使用CMake非交互方式构建会很简单。对于庞大的工程，比如VTK，推荐使用ccmake，cmake -i，或者cmake-gui。使用非交互方式构建工程，首先切换到构建目录，如果是源码内构建，运行cmake .后面使用-D标示传入可选项。对于源码外构建，运行cmake后面传入源码路径作为参数。然后输入make编译工程。有些工程有install目标，你可以输入make install安装。

Specifying the Compiler to CMake

指定CMake编译器

On some systems you may have more than one compiler to choose from or your compiler may be in a non-standard place. In these cases you will need to specify to CMake where your desired compiler is located. There are three ways to specify this; the generator can specify the compiler, an environment variable can be set, or a cache entry can be set. Some generators are tied to a speciﬁc compiler, for example the Visual Studio 6 generator always uses the Microsoft Visual Studio 6 compiler. For Makeﬁle based generators CMake will try a list of usual compilers until it ﬁnds a working compiler. The list can be found in the ﬁles:

在某些系统中，可能存在多个编译器或者编译器装在非标准位置。在这种情况下你需要指定编译器所在位置。有三种方法指定；可以使用生成器指定编译器，可以设置环境变量，或者设置缓存项。有些生成器同特定的编译器绑定在一起。比如Visual Studio 6生成器总是使用Microsoft Visual Studio 6 编译器。对于基于Makefile的生成器，CMake会尝试在一些列常用编译器中找到一个可用的。这些常用编译器能够在下列文件中找到：

Modules/CMakeDeterminCCompiler.cmake 和

Modules/CMakeDeterminCXXCompiler.cmake

The lists can be preempted with environment variables that can be set before CMake is run. The CC environment variable speciﬁes the C compiler while CXX speciﬁes the C++ compiler. You can specify the compilers directly on the command line by using -DCMAKE\_CXX\_COMPILER=cl for example. If those are not set, CMake will try the following list of compilers:

这个列表可以由在CMake运行之前设置的环境变量取代。环境变量CC指定C编译器，CXX指定C++编译器。你可以在命令行中通过-DCMAKE\_CXX\_COMPILER=cl可选项直接指定编译器。如果没有指定，CMake会测试下列编译器列表：

c++ g++ CC aCC cl bcc xlC.

Once CMake has been run and picked a compiler, you can change the selection by changing the cache entries CMAKE\_CXX\_COMPILER and CMAKE\_C\_COMPILER, although this is not recommended. The problem with doing this is that the project you are conﬁguring may have already run some tests on the compiler to determine what it supports. Changing the compiler does not normally cause these tests to be rerun which can lead to incorrect results. If you must change the compiler, start over with an empty binary directory. The ﬂags for the compiler and the linker can also be changed by setting environment variables. Setting LDFLAGS will initialize the cache values for link ﬂags, while CXXFLAGS and CFLAGS will initialize CMAKE\_CXX\_FLAGS and CMAKE\_C\_FLAGS respectively.

一旦CMake开始运行并选择了一个编译器，你可以通过改变缓存项CMAKE\_CXX\_COMPILER和CMAKE\_C\_COMPILER改变指定的编译器，虽然这么做并不推荐。这样做的问题是配置中的工程可能正在执行编译器支持功能的测试。更换编译器并不会重新执行这些测试，这可能导致不正确的结果。如果你必须要更换编译器，从一个空的构建目录开始。通过设置环境变量也可以改变编译器和连接器的标志位。设置LDFLAGS将初始化连接标志位的缓存项，CXXFLAGS和CFLAGS将分别初始化CMAKE\_CXX\_FLAGS和CMAKE\_C\_FLAGS变量。

Dependency Analysis

依赖分析

CMake has powerful built-in dependency analysis capabilities for C and C++ source code ﬁles. CMake also has limited support for Fortran and Java dependencies. Since Integrated Development Environments (IDES) support and maintain dependency information, CMake skips this step for those build systems. However, Makeﬁles with a make program do not know how to automatically compute and keep dependency information up-to-date. For these builds, CMake automatically computes dependency information for C, C++ and Fortran ﬁles. Both the generation and maintenance of these dependencies are automatically done by CMake. Once a project is initially configured by CMake, users only need to run make, and CMake does the rest of the work. CMake’s dependencies fully support parallel builds for multiprocessor systems.

对于C和C++源文件，CMake具有强大的依赖分析能力。而且对于Fortran和Java语言CMake也有一定的支持。因为集成开发环境支持并维护依赖信息，所以CMake跳过了这一步，而Makefiles和make程序并不能自动的计算和维护更新依赖信息。对于使用这些方式的构建，CMake为C，C++和Fortran自动计算依赖信息，这些依赖信息的生成和维护都是自动的。一个工程由CMake配置后，用户只需运行make，而CMake做了其他的工作。对于多处理器系统，CMake的依赖支持并行构建。

Although users do not need to know how CMake does this work, it may be useful to look at the dependency information ﬁles for a project. This information for each target is stored in four ﬁles called depend.make, flags.make, build.make, and Dependlnfo.cmake. depend.make stores the depend information for all the object ﬁles in the directory. flags . make contains the compile ﬂags used for the source ﬁles of this target. If they change then the ﬁles will be recompiled. Dependlnfo.cmake is used to keep the dependency information up-to-date and contains information about what ﬁles are part of the project and what languages they are in. Finally, the rules for building the dependencies are stored in build.make. If a dependency is out of date then all of the dependencies for that target will be recomputed, keeping the dependency information current.

尽管用户并不知道CMake是如何做的，不过能看到工程的文件依赖信息可能是有用的。每个目标的信息存储在depend.make，flags.make，build.make，和DependInfo.make等四个文件中。depend.make存储目中所有目标文件的依赖信息。flags.make包含这个目标的编译标志位。如果这些标志位改变了那么这些文件会被重新编译。DependInfo.cmake用来更新依赖信息，它包含了工程所包含的文件，和它们所使用的语言。最后，构建这些依赖的规则保存在build.make。如果一个依赖过时了，那么这个目标的所有依赖将被重新计算，保持依赖信息最新。

2.6 Editing CMakeLists Files

2.6 编辑CMakeLists文件

CMakeLists ﬁles can be edited in almost any text editor. Some editors, such as Notepad++, come with CMake syntax highlighting and indentation support built in. For editors such as Emacs or Vim CMake includes indentation and syntax highlighting modes. These can be found in the Docs directory of the source distribution, or downloaded from the CMake web site. The ﬁle Cmake-mode.el is the Emacs mode, and cmake-indent.vim and cmake-syntax.vim are used by Vim. Within Visual Studio the CMakeLists ﬁles are listed as part of the project and you can edit them simply by double clicking on them. Within any of the supported generators (Makeﬁles, Visual Studio, etc) if you edit a CMakeLists ﬁle and rebuild, there are rules that will automatically invoke CMake to update the generated ﬁles (eg. Makeﬁles or project ﬁles) as required. This helps to assure that your generated ﬁles are always in sync with your CMakeLists ﬁles.

CMakeLists文件几乎可以用任意文本编辑器编辑。有些编辑器，如Notepad++，支持内置的CMake语法高亮和缩进。对于Emacs或者Vim编辑器，CMake包含缩进和语法高亮模式。这些可以再源文件家的Docs目录找到，或者在CMake网站下载。Cmake-mode.el是Emacs使用的，cmake-indent.vim和cmake-syntax.vim是Vim使用的。对于Visula Studio工程，CMakeLists文件是工程文件，你可以通过双击编辑它。对于所支持的生成器（Makefiles，Visual Studio，等等）如果你编辑了CMakeLists文件，规则会自动调用CMake更新所需生成的文件（如Makefiles或者工程文件）。这确保生成的文件和你的CMakeLists文件保持同步。

Since CMake computes and maintains dependency information, the CMake executables must always be available (though they don’t have to be in your PATH) when make or an IDE is being run on CMake generated ﬁles. This means that if a CMake input ﬁle changes on disk, your build system will automatically re-run CMake and produce up-to-date build ﬁles. For this reason you generally should not generate Makeﬁles or projects with CMake and move them to another machine that does not have CMake installed.

因为CMake计算并维护依赖信息，当make或者IDE运行时，CMake可执行文件应该总是可以执行的（尽管它们不必须在PATH路径下）。也就是说如果CMake输入文件改变了，你的构建系统会自动重新执行CMake并生成更新的构建文件。因此你不应该使用CMake生成Makefiles或者工程文件，并把它们移动到没有安装CMake的机器中。

2.7 Setting Initial Values for CMake

2.7 设置CMake的初始值

While CMake works well in an interactive mode, sometimes you will need to setup cache entries without running a GUI. This is common when setting up nightly dashboards or if you will be creating many build trees with the same cache values. In these cases the CMake cache can be initialized in two different ways. The ﬁrst way is to pass the cache values on the CMake command line using -DCACHE\_VAR: TYPE=VALUE arguments. For example, consider the following nightly dashboard script for a UNIX machine:

由于CMake在互动模式下工作的很好，有时你不需要在GUI模式下设置缓存项。这在设置每日构建或使用一套缓存项创建多个构建树时是很正常的。在这种情况下，CMake缓存可以以两种方式初始化。第一种是执行CMake命令式通过-DCACHE\_VAR: TYPE=VALUE参数传入。例如，下面是一个UNIX系统上的每日构建脚本：

#!/bin/tcsh

cd ${HOME}

# wipe out the old binary tree and then create it again

rm -rf Foo-Linux

mkdir Foo-Linux

cd Foo-Linux

# run cmake to setup the cache

cmake -DBUILD\_TESTING:BOOL=ON <etc...> ../Foo

# generate the dashboard

ctest -D Nightly

The same idea can be used with a batch ﬁle on Windows. The second way is to create a ﬁle to be loaded using CMake's -c option. In this case instead of setting up the cache with -D options it is done though a ﬁle that is parsed by CMake. The syntax for this ﬁle is standard CMakeLists syntax and it is typically just a series of set commands such as:

同样的方式也可以应用到Windows的批处理文件。第二种方式是创建一个文件使用-c可选项传入到CMake。这种方式可以代替第一种使用-D可选项的方式。这个文件的语法是标准的CMakeLists语法，其实就是一系列的set命令：

#Build the vtkHybrid kit.

set (VTK\_USE\_HYBRID ON CACHE BOOL "doc string")

In some cases there might be an existing cache and you want to force the cache values to be set a certain way. For example say you want to turn Hybrid on even if the user has previously run CMake and turned it off. Then you can do:

某种情况下，可能缓存已经存在，而你想要强制设置缓存项。比如你想设置Hybrid项为启用，即使用户之前已经运行了CMake并设置该项为关闭。那么你可以这样做：

#Build the vtkHybrid kit always.

set (VTK\_USE\_HYBRID ON CACHE BOOL "doc" FORCE)

Another option is that you want to set and then hide options so the user will not be tempted to adjust them later on. This can be done using the following commands:

另一种可选项是你想要设置并隐藏它，让用户以后无法改变它。你可以这样做：#Build the vtkHybrid kit always and don’t distract

#the user by showing the option.

set (VTK\_USE\_HYBRID ON CACHE INTERNAL "doc" FORCE)

mark\_as\_advanced (VTK\_USE\_HYBRID)

You might be tempted to edit the cache ﬁle directly, or to "initialize" a project by giving it an initial cache ﬁle. This may not work and could cause additional problems in the future. First, the syntax of the CMake cache is subject to change. Second, cache ﬁles have full paths in them that make them unsuitable for moving between binary trees. So if you want to initialize a cache ﬁle use one of the two standard methods described above.

你可能想要直接编辑缓存文件，或者给定缓存文件初始化工程。这可能行不通，或者产生额外的问题。第一，CMake缓存的语法有可能变化。第二，缓存文件中存在绝对路径，在构建树之间移动它们是不合适的。所以如果你想要初始化缓存文件，使用上述两种方法。

2.8 Building Your Project

2.8 构建你的工程

After you have run CMake your project will be ready to be built. If your target generator is based on Makeﬁles then you can build your project by changing directory to your binary tree and typing make (or gmake or nmake as appropriate). If you generated ﬁles for an IDE such as Visual Studio, you can start your lDE, load the project ﬁles into it, and build as you normally would.

当执行完CMake你的工程就可以开始构建了。如果你的目标生成器是基于Makefiles的，那么你可切换目录到构建目录，输入make（或者gmake或者nmake）来构建你的工程。如果你生成的文件是IDE工程文件，如Visual Studio，你可以启动IDE，载入工程文件，构建工程。

Another option is to use CMake’s -build option from the command line. This option is simply a convenience that allows you to build your project from the command line, even if that requires launching an IDE. The command line options for -build include:

另外也可以通过命令行使用CMake的 -build可选项。这个可选项只是为了方便你使用命令行构建你的工程，这可能需要启动IDE。命令行可选项-build包括：

Usage: cmake --build <dir> [options] [-- [native-options]]

Options:

<dir> = Project binary directory to be built.

--target <tgt> = Build <tgt> instead of default targets.

--config <cfg> = For multi-configuration tools, choose <cfg>.

--clean-first = Build target 'clean' first, then build.

(To clean only, use --target 'clean'.)

-- = Pass remaining options to the native tool.

So even if you are using Visual Studio as your generator you can type the following to build your project from the command line if you wish.

所以即使你使用Visual Studio作为生成器，你也可以输入下列命令从命令行构建你的工程。

cmake --build <your binary dir>

That is all there is to installing and running CMake for simple projects. In the following chapters we will consider CMake in more detail and how to use it on more complex software projects.

这就是安装运行CMake的全部内容。之后的章节会讨论CMake的更多使用细节及如何将它使用在复杂工程中。

Chapter 3

第三章

Key Concepts

核心概念

3.1 Main Structures

3.1 主结构

This chapter provides an introduction to CMake's key concepts. As you start working with CMake you will run into a variety of concepts such as targets, generators, and commands. In CMake these concepts are implemented as C++ classes and are referenced in many of CMake's commands. Understanding these concepts will provide you with the working knowledge you need to create effective CMakeLists ﬁles.

本章介绍了CMake的核心概念。当你开始使用CMake时，你会接触到很多新概念例如目标，生成器，命令。在CMake里，这些概念由C++的类实现，并在很多命令中用到。理解这些概念可以帮助你编写高效的CMakeLists文件。

Before going into detail about CMake’s classes it is worth understanding their basic relationships. At the lowest level there are source files. These correspond to typical C or C++ source code files. Source files are combined into targets. A target is typically an executable or library. A directory represents a directory in the source tree and typically has a CMakeLists file and one or more targets associated with it. Every directory has a local generator that is responsible for generating the Makeﬁles or project ﬁles for that directory. All of the local generators share a common global generator that oversees the build process. Finally, the global generator is created and driven by the cmake class itself.

在深入了解这些类的实现细节之前，理解它们之间的关系是很有帮助的。最底层是源文件，对应C或者C++的源码文件。源文件组成目标。目标一般指代一个可执行文件或者程序库。目录表示源代码树所在文件夹，一般包含一个CMakeLists文件。里面定义了一个或多个目标。每个目录有个一个本地生成器来生成Makefiles或对应本目录的工程文件。所有的本地生成器共享一个全局生成器，监控构建过程。最后，全局生成器有cmake创建并驱动。

Figure 4 shows the basic class structure of CMake. We will now consider CMake's concepts in a bit more detail. CMake's execution begins by creating an instance of the cmake class and passing the command line arguments to it. This class manages the overall conﬁguration process and holds information that is global to the build process such as the cache values. One of the ﬁrst things the cmake class does is to create the correct global generator based on the user's selection of what generator to use (such as Visual Studio 10, Borland Makeﬁles, or UNIX Makefiles). At this point the cmake class passes control to the global generator it created by invoking the conﬁgure and generate methods.

图4显示了CMake类的基本结构。我们现在可以深入了解CMake的概念的更多细节。CMake执行时，先创建一个cmake类的对象，把命令行参数传递给它。这个类管理全部的配置过程并在构建过程中保存全局数据，如缓存项。首先cmake类根基用户选择的生成器（如Visual Studio 10，Borland Makefiles，或者UNIX Makefiles）创建全局生成器。然后cmake类将控制权传给它调用配置和生成方法生成的全局生成器。

The global generator is responsible for managing the conﬁguration and generation of all of the Makeﬁles (or project ﬁles) for a project. In practice most of the work is actually done by local generators which are created by the global generator. One local generator is created for each directory of the project that is processed. So while a project will have only one global generator it may have many local generators. For example, under Visual Studio 7 the global generator creates a solution ﬁle for the entire project while the local generators create a project ﬁle for each target in their directory.

全局生成器负责管理对应所有工程的Makefiles（或者工程文件）的配置和生成。实际上，由全局生成器创建的本地生成器完成了大部分的工作。每个本地生成器对应工程中的一个目录。所以一个工程有一个全局生成器和多个本地生成器。例如，在Visual Studio 7中全局生成器创建一个对应整个工程的解决方案文件，而本地生成器创建对应每个目标所在目录的工程文件。

In the case of the "Unix Makeﬁles" generator, the local generators create most of the Makeﬁles and the global generator simply orchestrates the process and creates the main top-level Makeﬁle. Implementation details vary widely among generators. The Visual Studio 6 generators make use of .dsp and .dsw ﬁle templates and perform variable replacements on them. The generators for Visual Studio 7 and later directly generate the XML output without using any ﬁle templates. The Makeﬁle generators including UNIX, NMake, Borland, etc use a set of rule templates and replacements to generate their Makeﬁles.

例如"Unix Makefiles"生成器，本地生成器生成大部分的Makefiles而全局生成器只管理整个过程并创建最顶部的Makefile。生成器的实现多种多样。VIsual Studio 6生成器使用.dsp和。dsw文件作为模板，替换它们中间的变量。Visual Studio 7及更高版本不使用模板直接生成XML文件。而Makefile生成器（包括UNIX，NMAKE，Borland，等等）使用一套规则模板，替换生成它们自己的Makefiles。

Each local generator has an instance of the class cmMakeﬁle, cmMakeﬁle is where the results of parsing the CMakeLists ﬁles are stored. Speciﬁcally, for each directory in a project there will be a single cmMakeﬁle instance which is why the cmMakeﬁle class is often referred to as the directory. This is clearer for build systems that do not use Makeﬁles. That instance will hold all of the information from parsing that directory's CMakeLists ﬁle (see Figure 5). One way to think of the cmMakeﬁle class is as a structure that starts out initialized with a few variables from its parent directory, and is then ﬁlled in as the CMakeLists ﬁle is processed. Reading in the CMakeLists ﬁle is simply a matter of CMake executing the commands it ﬁnds in the order it encounters them.

每个本地生成器有一个cmMakefile实例，它分析CMakeLists文件并保存结果。特别的，工程中的每个目录都会有一个cmMakefile实例，所以通常cmMakefile指代一个目录。这对于不使用Makefiles的构建系统是明确的。这个实例保存通过分析对应目录中的CMakeLists文件（参考图5）获得的信息。可以这样理解，cmMakefile类是一个结构体，它通过父目录的一些变量初始化，通过分析CMakeLIst文件保存信息。CMake按CMakeList文件中的命令顺序执行。

Each command in CMake is implemented as a separate C++ class, and has two main parts. The ﬁrst part of a command is the InitialPass method. The lnitialPass method receives the arguments and the cmMakefile instance for the directory currently being processed, and then performs its operations. In the case of the set command, it processes its arguments and if the arguments are correct it calls a method on the cmMakefile to set the variable. The results of the command are always stored in the cmMakefile instance. Information is never stored in a command. The last part of a command is the FinalPass. The FinalPass of a command is executed after all commands (for the entire CMake project) have had their InitialPass invoked. Most commands do not have a FinalPass, but in some rare cases a command must do something with global information that may not be available during the initial pass.

CMake中的每个命令由不同的C++类实现，主要分为两部分。命令的第一部分是InitialPass (初始化传递)方法。该方法接收参数，当前目录的cmMakefile实例开始执行操作。例如set命令，它处理传递进来的参数，如果参数正确，就调用cmMakefile去设置变量的值。命令执行的结果总是保存在cmMakefile实例中。信息不会保存在命令中。命令的第二部分是FinalPass(最终传递)，它执行在所有命令（整个CMake工程中的）执行完InitalPass之后。大多数命令没有FinalPass部分，但是有些特殊情况命令必须做某些设计全局信息的事情，而它在InitialPass阶段无法完成。

Once all of the CMakeLists ﬁles have been processed the generators use the information collected into the cmMakefile instances to produce the appropriate ﬁles for the target build system (such as Makeﬁles).

一旦所有的CMakeLists文件被处理完，生成器将收集到的信息传入cmMakefile实例，产生适用于目标构建系统的文件（如Makefiles）。

3.2 Targets

3.2 目标

Now that we have discussed the overall process of CMake, let us consider some of the key items stored in the cmMakefile instance. Probably the most important item is targets. Targets represent executables, libraries, and utilities built by CMake. Every add\_l ibrary, add\_executable, and add\_custom\_target command creates a target. For example, the following command will create a target named foo that is a static library, with foo1.c and foo2.c as source ﬁles.

现在我们开始全面讨论CMake的工作过程，cmMakefile实例保存了一些重要的条目。这里最重要的部分就是目标。目标指代由CMake构建的可执行文件，程序库，或者工具。每个add\_library，add\_executable和add\_custom\_targe命令创建一个目标。例如下面的命令创建一个叫做foo的目标，它使用源文件foo1.c和foo2.c构建静态程序库。

add\_library (foo STATIC foo1.c foo2.c)

The name foo is now available for use as a library name everywhere else in the project, and CMake will know how to expand the name into the library when needed. Libraries can be declared to be of a particular type such as STATIC, SHARED, MODULE, or left undeclared. STATIC indicates that the library must be built as a static library. Likewise SHARED indicates it must be built as a shared library. MODULE indicates that the library must be created so that it can be dynamically loaded into an executable. On many operating systems this is the same as SHARED, but on other systems such as Mac OS X it is different. If none of these options are speciﬁed this indicates that the library could be built as either shared or static. In that case CMake uses the setting of the variable BUILD\_SHARED\_LIBS to determine if the library should be SHARED or STATIC. If it is not set, then CMake defaults to building static libraries.

在整个工程中foo被用作程序库的名字，CMake知道怎样展开程序库的名字。程序库能够被声明成不同的类型，如静态库，共享库，模块，或者未声明。静态库指明程序库被构建为静态程序库。共享库是指构建成共享程序库。模块是指构建成程序库可以被可执行文件动态调用。在许多操作系统中，它和共享程序库是一样的，但是在Max OS X这类操作系统中是不同的。如果这个选项没有被指明，则表示它可以被构建成动态或者共享的。也就是说CMake通过变量BUILD\_SHARED\_LIBS判断程序库应该被构建成共享的还是静态的。如果该变量没有设置，CMake默认构建静态程序库。

Likewise executables have some options. By default an executable will be a traditional console application that has a main (int argc, const char\*argv[] ). If WIN32 is speciﬁed after the executable name then the executable will be compiled as a MS Windows executable and the operating system will call WinMain instead of main at startup. WIN32 has no effect on non-Windows systems.

另外，可执行文件文件也有一些可选项。默认情况下，可执行文件会被构建成传统的控制台程序，它有一个main(int argc, const char\* argv[])函数作为入口。如果可执行文件的把名字后面跟着WIN32，那么它将被编译成MS Windows可执行程序，入口函数就变成了WinMain。这在非Windows系统中无效。

In addition to storing their type, targets also keep track of general properties. These properties can be set and retrieved using the set\_target\_properties and get\_target\_property commands, or the more general set\_property and get\_property commands. The most commonly used property is LINK\_FLAGS, which is used to specify link ﬂags for a speciﬁc target. Targets store a list of libraries that they link against which are set using the target\_link\_libraries command. Names passed into this command can be libraries, full paths to libraries, or the name of a library from an add\_library command. They also store the link directories to use when linking, the install location for the target, and custom commands to execute after linking.

除了保存类型，目标也保存通用属性项。这些属性项可以通过命令被set\_target\_properties和get\_target\_property命令设置或取得，也可以使用set\_property和get\_property命令。最常用的属性项是LINK\_FLAGS，它被用来指明特定目标的连接标志位。目标保存由target\_link\_libraries命令设置的连接时使用的程序库列表。传入这个命令的名字可以是程序库的名字，可以是程序库的绝对地址，或者是使用add\_library命令指定的名字。它们也保存连接时使用的目录，目标安装路径，以及连接后执行的用户命令。

For each library CMake creates, it keeps track of all the libraries on which that library depends. Since static libraries do not link to the libraries on which they depend, it is important for CMake to keep track of the libraries so they can be speciﬁed on the link line of the executable being created. For example,

对于CMake创建的每个程序库，它都跟踪它们所依赖的程序库。因为静态程序库并不连接它们的依赖，所以当使用这些程序库连接可执行文件时保持跟踪它们所依赖的程序库就非常重要了。例如，

add‘library (foo foo.cxx)

target‘linkﬁlibraries (foo bar)

addiexecutable (foobar foobar.cxx)

target\_link\_libraries (foobar foo)

This will link the libraries foo and bar into the executable foobar even, although only foo was explicitly linked into foobar. With shared or DLL builds this linking is not always needed, but the extra linkage is harmless. For static builds this is required. Since the foo library uses symbols from the bar library, foobar will most likely also need bar since it uses foo.

这将会连接程序库foo和bar到可执行文件foobar，尽管只写明程序库foo连接到foobar。要是使用动态共享库或者动态链接库就不需要了，不过加上也没有坏处。对于静态程序库这是必须的。因为程序库foo使用程序库bar定义的标志，foobar很可能也需要bar。

3.3 Source Files

3.3 源文件

The source ﬁle structure is in many ways similar to a target. It stores the ﬁlename, extension, and a number of general properties related to a source ﬁle. Like targets you can set and get properties using set\_source\_files\_properties and get\_source\_file\_property, or the more generic versions. The most common properties include:

源文件的结构和目标的类似。它保存文件名，扩展名，和一系列的源文件相关通用属性项。和目标类似，你可以通过set\_source\_files\_properties和get\_source\_file\_property来设置和获得这些属性项。常用的属性项包括：

COMPILE\_FLAGS

编译标志位

Compile ﬂags speciﬁc to this source ﬁle. These can include source speciﬁc -D and -I ﬂags.

编译标志位是源码文件的属性项。可以通过-D和-I加入这些标志位。

GENERATED

生成的

The GENERATED property indicates that the source ﬁle is generated as part of the build process. In this case CMake will treat it differently for computation of dependencies because the source ﬁle may not exist when CMake is ﬁrst run.

"生成的"属性表示源码文件是在构建过程中生成的。CMake不会计算它的依赖，因为源码文件在CMake第一次运行时可能不存在。

OBJECT\_DEPENDS

对象依赖

Adds additional ﬁles on which this source ﬁle should depend. CMake automatically performs dependency analysis to determine the usual C, C++ and Fortran dependencies, This parameter is used rarely in cases where there is an unconventional dependency or the source ﬁles do not exist at dependency analysis time.

加入该源码文件的依赖。CMake自动进行依赖分析，决定是C，C++还是Fortran的依赖，这个参数并不常用如果源码文件在依赖分析时不存在，或者存在非常规依赖。

ABSTRACT

WRAP\_EXCLUDE

抽象包装排除

CMake doesn't directly use these properties. Some loaded commands and extensions to CMake look at these properties to determine how and when to wrap a C++ class into languages such as Tcl, Python, etc.

CMake并不直接使用这些属性项。有些CMake加载的命令和扩展查找这些属性项决定合适及如何包装一个C++类到其他语言，如Tcl，Python等等。

3.4 Directories, Generators, Tests, and Properties

3.4 目录，生成器，测试和属性项

in addition to targets and source ﬁles you may ﬁnd yourself occasionally working with other classes such as directories, generators, and tests. Normally such interactions take the shape of setting or getting properties from these objects. All of these Classes have properties associated with them, as do source ﬁles and targets. A property is a key—value pair attached to a speciﬁc object such as a target. The most generic way to access properties is through the set\_property and get\_property commands. These commands allow you to set or get a property from any class in CMake that has properties. Some of the properties for targets and source ﬁles have already been covered. Some useful properties for a directory include:

除了目标和源码文件，你可能还会遇到其他的概念如目录，生成器，和测试。一般的使用设置和获取方法属性项。所有这些类都有与之相关的属性项，这和源码文件及目标一样。属性项就是某些指定对象的一个键值对，如目标。访问属性项最通用的方式是使用set\_property和get\_property命令。这些命令允许你设置或获取CMake中的任何类的属性项。目标和源码文件的属性项有些已经介绍过了。下面是目录的一些比较有用的属性项：

ADDITIONAL\_MAKE\_CLEAN\_FILES：

额外需要清理的文件：

This property speciﬁes a list of additional ﬁles that will be cleaned as a part of the "make clean" stage. By default CMake will clean up any generated ﬁles that it knows about, but your build process may use other tools that leave ﬁles behind. This property can be set to a list of those ﬁles so that they also will be properly cleaned up.

这个属性项指定了一个在"清理"阶段额外需要清理的文件列表。默认情况下，CMake将清除它所知的生成的所有文件，但是构建过程可能使用其他工具生成文件，这些文件不被删除。该属性项设置一个额外需要删除的文件列表。

EXCLUDE\_FROM\_ALL

排除属性

This property indicates if all the targets in this directory and all sub directories should be excluded from the default build target. If it is not, then with a Makeﬁle for example typing make will cause these targets to be built as well. The same concept applies to the default build of other generators.

这个属性项表示该目录及其子目录中的所有目标将被排除在默认构建目标之外。如果没有设置该属性项，则对于是Makefile生成器，输入make项构建这些目标。对于其他生成器，该概念同样适用。

LISTFILE\_STACK

文件列表栈

This property is mainly useful when trying to debug errors in your CMake scripts. It returns a list of what list ﬁles are currently being processed, in order. So if one CMakeLists ﬁle does an include command then that is effectively pushing the included CMakeLists ﬁle onto the stack.

该属性项在检查CMake脚本错误时非常有用。它返回一个已经处理完的有序的文件列表。所以如果一个CMakeLists文件使用了include命令，它会有效的将包含的CMakeLists文件加入这个栈。

A full list of properties supported in CMake can be obtained by running cmake with the --help-property-list option. The generators and directories are automatically created for you as CMake processes your source tree.

运行cmake时使用--help-property-list选项将显示CMake所支持的完整的属性项列表。生成器和目录将在CMake处理你的源代码树时自动创建。

3.5 Variables and Cache Entries

3.5 变量和缓存项

CMakeLists files use variables much like any programming language. Variables are used to store values for later use, and can be a single value such as "ON" or "OFF", or they can represent a list such as(/usr/include /home/foo/include /usr/local/include). A number of useful variables are automatically deﬁned by CMake and are discussed in Appendix A - Variables.

CMakeLists文件像编程语言一样使用变量。变量用来存储之后要用到的值，它可以是简单的"ON"或"OFF"，或者可以是一个列表如(/usr/include /home/foo/include /usr/local/include)。一些有用的变量已经由CMake自动定义好了，我们在附件A-变量章节会继续讨论。

Variables in CMake are referenced using a ${VARIABLE} notation, and they are deﬁned in the order of execution of the set commands. Consider the following example:

变量在CMake中可以使用￥{VARIABLE}重新定义，它们按set命令执行的顺序定义。请参考下面的例子：

# FOO is undefined

set (FOO 1)

# FOO is now set to 1

set (FOO 0)

# FOO is now set to 0

This may seem straightforward, but consider the following example:

这可能有点简单，请看下面的例子：

set (FOO 1)

if (${FOO} LESS 2)

set (FOO 2)

else (${FOO} LESS 2)

set (FOO 3)

endif (${FOO} LESS 2)

Clearly the if statement is true, which means that the body of the if statement will be executed. That will set the variable FOO to 2, and so when the else statement is encountered FOO will have a value of 2. Normally in CMake the new value of FOO would be used, but the else statement is a rare exception to the rule and always refers back to the value of the variable when the if statement was executed. So in this case the body of the else clause will not be executed. To further understand the scope of variables consider this example:

首先，如果if判断为真，那么条件语句体将被执行。变量FOO的值会被设成2，else语句判断时FOO的值就会是2。正常情况下，在CMake中会使用FOO变量的新值，但是else语句使用特殊规则，它使用if语句时变量的值判断。所以这种情况下else后面的语句体将不被执行。下面的例子有助于理解变量的作用域：

set (foo 1)

# process the dir1 subdirectory

add\_subdirectory (dir1)

# include and process the commands in file1.cmake

include (file1.cmake)

set (bar 2)

# process the dir2 subdirectory

add\_subdirectory (dir2)

# include and process the commands in file2.cmake

include (file2.cmake)

In this example because the variable foo is deﬁned at the beginning, it will be deﬁned while processing both dir1 and dir2. In contrast bar will only be deﬁned when processing dir2. Likewise foo will be deﬁned when processing both ﬁle1.cmake and ﬁle2.cmake, whereas bar will only be deﬁned while processing ﬁle2.cmake.

在这个例子中变量foo在一开始就定义了，在处理dir1和dir2的过程中，它一直有效。相反的，变量bar只在处理dir2的过程中有效。同样的foo在处理file1.cmake和file2.cmake时有效，而bar只在处理file2.cmake时有效。

Variables in CMake have a scope that is a little different from most languages. When you set a variable it is visible to the current CMakeLists ﬁle or function, as well as any subdirectory’s CMakeLists ﬁles, any functions or macros that are invoked, and any ﬁles that are included using the INCLUDE command. When a new subdirectory is processed (or a function called) a new variable scope is created and initialized with the current value of all variables in the calling scope. Any new variables created in the child scope, or changes made to existing variables, will not impact the parent scope. Consider the following example:

CMake中变量的作用域同大多数语言有一些区别。当你设置了一个变量，它对于当前CMakeList文件或函数是可见的，而且对于子目录CMakeLists文件，函数，宏，以及使用include命令包含的文件也可见。当处理一个新的子目录时（或者函数调用时），生成一个新的变量的作用域，使用调用时的变量的值对它进行初始化。新的变量在子作用域中被创建，或者转存到已经存在的变量中，它不会影响父作用域。请参考下面的例子：

function (foo)

message (${test}) # test is 1 here

set (test 2)

message (${test}) # test is 2 here, but only in this scope

endfunction()

set (test 1)

foo()

message (${test}) # test will still be 1 here

In some cases you might want a function or subdirectory to set a variable in its parent’s scope. This is one way for CMake to return a value from a function, and it can be done by using the PARENT\_SCOPE option with the set command. We can modify the prior example so that the function foo changes the value of test in its parent’s scope as follows:

某些时候你可能想要一个函数或者子目录设置父作用域中的变量。这是CMake使用函数返回值的方式，可以通过set命令使用PARENT\_SCOPE选项来实现。我们可以修改之前的例子使得函数foo改变父作用域变量test的值，例子如下：

function (foo)

message (${test}) # test is 1 here

set (test 2 PARENT\_SCOPE)

message (${test}) # test still 1 in this scope

endfunction()

set (test 1)

foo()

message (${test}) # test will now be 2 here

Variables can also represent a list of values. In these cases when the variable is expanded it will be expanded into multiple values. Consider the following example:

变量也可以表示为值的列表。在这种情况下当变量将展开成多个值的形式。请看下面的例子：

# set a list of items

set (items\_to\_buy apple orange pear beer)

# loop over the items

foreach (item ${items\_to\_buy})

message ( "Don’t forget to buy one ${item}" )

endforeach ()

In some cases you might want to allow the user building your project to set a variable from the CMake user interface. In that case the variable must be a cache entry. Whenever CMake is run it produces a cache file in the directory where the binary ﬁles are to be written. The values of this cache file are displayed by the CMake user interface. There are a few purposes of this cache. The first is to store the user's selections and choices, so that if they should run CMake again they will not need to reenter that information. For example, the option command creates a Boolean variable and stores it in the cache.

某种情况下你可能允许用户使用CMake用户界面去设置一些变量，来构建你的工程。这种情况下，变量必须是缓存项。每次CMake运行，它都在构建目录下产生一个缓存文件。这个缓存文件的内容显示在CMake用户界面中。这样做有几个目的。 首先是保存用户选项，再执行CMake时不必重新选一次。例如，option命令生成一个布尔型变量，并保存在缓存中。

option (USE\_JPEG "Do you want to use the jpeg library")

The above line would create a variable called USE\_JPEG and put it into the cache. That way the user can set that variable from the user interface and its value will remain in case the user should run CMake again in the future. To create a variable in the cache you can use commands like option, find\_file, or you can use the standard set command with the CACHE option.

上面这一行生成一个变量USE\_JPEG并把它放入缓存中。这样用户可以通过用户交互界面设置变量的值，在下次运行时获得正确的值。你可以使用命令option，find\_file，或者set CACHE选项来生成一个变量。

set (USE\_JPEG ON CACHE BOOL "include jpeg support?")

When you use the cache option you must also provide the type of the variable and a documentation string. The type of the variable is used by the GUI to control how that variable is set and displayed. Variable types include BOOL, PATH, FILEPATH, and STRING. The documentation string is used by the GUI to provide online help.

当你使用cache选项，你必须给出变量类型和文档描述。变量类型用来在可视化交互界面中控制变量的设置和显示。变量类型包括BOOL，PATH，FILEPATH和STRING。文档描述被用来提供帮助文档。

The other purpose of the cache is to store key variables that are expensive to determine. These variables may not be visible or adjustable by the user. Typically these values are system dependent variables such as CMAKE\_WORDS\_BIGENDIAN, which require CMake to compile and run a program to determine their value. Once these values have been determined, they are stored in the cache to avoid having to recompute them every time CMake is run. Generally CMake tries to limit these variables to properties that should never change (such as the byte order of the machine you are on). If you signiﬁcantly change your computer, either by changing the operating system, or switching to a different compiler, you will need to delete the cache ﬁle (and probably all of your binary tree's object ﬁles, libraries, and executables).

另一个使用缓存保存重要变量的目的是决定变量值的成本很高。这些变量可能对于用户不可见，不可修改。一般来讲它们的值是系统相关的，比如CMAKE\_WORDS\_BIGENDIAN，需要CMake编译并执行一个程序去判断它的值。一旦这些值确定后，它们就被保存在缓存中以避免每次CMake运行时重复运算。一般来讲，CMake限制这些变量的值的修改（比如当前运行机器的字节顺序）。如果你更换了机器，或者切换了操作系统，你就需要删除缓存文件（很可能也需要删除二进制树的目标文件，程序库，和可执行文件）。

Variables that are in the cache also have a property indicating if they are advanced or not. By default when a CMake GUI is run (such as ccmake or cmake-gui) the advanced cache entries are not displayed. This is so that the user can focus on the cache entries that they should consider changing. The advanced cache entries are other options that the user can modify, but typically will not. It is not unusual for a large software project to have ﬁfty or more options, and the advanced property lets a software project divide them into key options for most users and advanced options for advanced users. Depending on the project there may not be any non-advanced cache entries. To make a cache entry advanced the mark\_as\_advanced command is used with the name of the variable (a.k.a. cache entry) to make advanced.

缓存中的变量有一个属性表示它们是否是高级缓存项。默认情况下当CMake可视化用户界面运行时（如ccmake或者cmake-gui）不显示高级缓存项。这可以让使用者集中精力在可修改的缓存项上。高级缓存项可以使用其他选项修改，但是一般不这样做。一般大型软件可能有几十个选项，高级属性项把一个软件工程分为重要选项（适用于大多数用户）和高级选项适用于高级用户。一个工程可能没有任何非高级缓存项。可以使用mark\_as\_advanced命令把一个缓存项设置为高级。

In some cases you might want to restrict a cache entry to a limited set of predeﬁned options. You can do this by setting the STRINGS property on the cache entry. The following CMakeLists code illustrates this by creating a property named CRYPTOBACKEN as usual, and then setting the STRINGS property on it to a set of three options.

有时你可能想限制缓存项为一组预定义选项中的一项。你可以对缓存项使用STRINGS属性。下面的CMakeLists代码通过正常创建属性项CRYPTOBACKEN，然后通过设置STRINGS属性项为三个选项，展示了这种方式。

set (CRYPTOBACKEND "OpenSSL" CACHE STRING

"Select a cryptography backend")

set\_property (CACHE CRYPTOBACKEND PROPERTY STRINGS

"OpenSSL" "LibTomCrypt" "LibDES")

When cmake-gui is run and the user selects the CRYPTOBACKEND cache entry, they will be presented with a pulldown to select which option they want, as shown in Figure 6.

当cmake-gui运行时，用户点击CRYPTOBACKEND缓存项，将会弹出一个下拉列表让用户选择。如图6。

A few ﬁnal points should be made concerning variables and their interaction with the cache. If a variable is in the cache, it can still be overridden in a CMakeLists ﬁle using the set command without the CACHE option. Cache values are checked only if the variable is not found in the current cmMakefile instance before CMakeLists ﬁle processing begins. The set command will set the variable for processing the current CMakeLists ﬁle (and subdirectories as usual) without changing the value in the cache.

最后会介绍变量和缓存的交互关系。如果变量在缓存中，通过set命令，它仍能被CMakeLists文件修改，而不需要添加CACHE选项。只有在CMakeLists文件处理前cmMakefile实例中没有变量，缓存项才会被使用。set命令在处理当前CMakeLists文件（及其子目录）时，设置变量，而且不改变变量在缓存中的值。

# assume that FOO is set to ON in the cache

set (FOO OFF)

# sets foo to OFF for processing this CMakeLists file

# and subdirectories; the value in the cache stays ON

Once a variable is in the cache, its "cache" value cannot normally be modiﬁed from a CMakeLists ﬁle. The reasoning behind this is that once CMake has put the variable into the cache with its initial value, the user may then modify that value from the GUI. If the next invocation of CMake overwrote their change back to the set value, the user would never be able to make a change that CMake wouldn’t overwrite. So a set (FOO ON CACHE BOOL "cloc") command will typically only do something when the cache doesn‘t have the variable in it. Once the variable is in the cache, that command will have no effect.

一旦一个变量被放入缓存中，它对应的缓存项通常不能被CMakeLists文件修改。因为一旦CMake把变量放入缓存中，并初始化以后，用户将只能使用GUI修改它的值。如果下次运行CMake时改变了缓存项的值，用户将无法只通过GUI修改它的值。所以一条set (FOO ON CACHE BOOL "cloc")命令一般只有当缓存中没有这个变量时才有效，一旦这个变量在缓存中，该命令将失效。

In the rare event that you really want to change a cached variable's value you can use the FORCE option in combination with the CACHE option to the set command. The FORCE option will cause the set command to override and change the cache value of a variable.

如果确实有场合需要改变缓存中变量的值，你可以使用set命令组合FORCE选项和CACHE选项。FORCE选项将使得set命令修改缓存中变量的值。

3.6 Build Configurations

3.6 构建配置

Build conﬁgurations allow a project to be built in different ways for debug, optimized, or any other special set of ﬂags. CMake supports, by default, Debug, Release, MinSizeRel, and RelWithDeblnfo conﬁgurations. Debug has the basic debug ﬂags turned on. Release has the basic optimizations turned on. MinSizeRel has the ﬂags that produce the smallest object code, but not necessarily the fastest code. RelWithDebInfo builds an optimized build with debug information as well.

构建配置允许一个工程按调试，优化，或者其他特殊的标志位等不同需求构建。CMake默认支持构建调试，发布，最小发布，和带调试信息的发布版本的构建。调试版打开几本调试标志位。发布版打开基本优化功能。最小发布版使用生成最小对象代码的标志位，但不保证速度最快。带调试信息的发布版使用带调试信息的优化构建。

CMake handles the conﬁgurations in slightly different ways depending on what generator is being used. The conventions of the native build system are followed when possible. This means that conﬁgurations impact the build in different ways when using Makeﬁles versus using Visual Studio project ﬁles.

根据生成器的不同，CMake的配置有些许变化。它根据本地构建系统的不同也有变化。这意味着配置根据本地系统的不同以及使用Makefiles还是Visual Studio工程而变化。

The Visual Studio IDE supports the notion of Build Conﬁgurations. A default project in Visual Studio usually has Debug and Release conﬁgurations. From the IDE you can select build Debug, and the ﬁles will be built with Debug ﬂags. The IDE puts all of the binary ﬁles into directories with the name of the active conﬁguration. This brings about an extra complexity for projects that build programs that need to be run as part of the build process from custom commands. See the CMAKE\_CEG\_INTDIR variable and the custom commands section for more information about how to handle this issue. The variable CMAKE\_CONFIGURATION\_TYPES is used to tell CMake which conﬁgurations to put in the workspace.

Visual Studio IDE支持构建配置的概念。一个默认的Visual Studio工程通常有调试和发布两种配置。使用IDE你可以选择构建调试版，使用调试标志位构建。IDE将所有构建好的二进制文件放到使用当前配置命名的文件夹中。这将使得构建一个工程更加复杂，构建程序需要使用指定的命令构建。了解CMAKE\_CEG\_INTDIR变量并且阅读指定命令章节，获得此问题的更多信息。CMAKE\_CONFIGURATION\_TYPES变量告诉CMake使用哪个配置。

With Makeﬁle based generators, only one conﬁguration can be active at the time CMake is run, and it is speciﬁed by the CMAKE\_BUILD\_TYPE variable. if the variable is empty then no ﬂags are added to the build. If the variable is set to the name of a conﬁguration, then the appropriate variables and rules (such as CMAKEWCXX\_FLAGS\_<ConfigName>) are added to the compile lines. Makeﬁles do not use special conﬁguration subdirectories for object ﬁles. To build both debug and release trees, the user is expected to create multiple build directories using the out of source build feature of CMake, and to set the CMAKE\_BUILD\_TYPE to the desired selection for each build. For example,

使用基于Makefile的生成器，在CMake运行时，同时只能有一种配置，它由CMAKE\_BUILD\_TYPE指定。如果变量为空则不添加任何标志位构建。如果这个变量设置为配置的名字，那么将添加适合的变量和规则（如CMAKE\_CXX\_FLAGS\_<ConfigName>）到编译命令行。Makefile不为目标。文件使用特别的配置子目录。构建调试版和发行版，使用者希望根据构建类型创建不同的目录，并设置CMAKE\_BUILD\_TYPE想要的类型。例如，

# With source code in the directory MyProject

# to build MyProject—debug create that directory, cd into it and

(ccmake ../MyProject -DCMAKE\_BUILDWTYPE:STRING=Debug)

# the same idea is used for the release tree MyProject—release

(ccmake ../MyProject -DCMAKE\_BUILD\_TYPE:STRING=Release)

Writing CMakeLists Files

编写CMakeLists文件

This chapter will cover the basics of writing effective CMakeLists ﬁles for your software. It will cover all of the basic commands and issues you will need to handle most projects. It will also discuss how to convert existing UNIX or Windows projects into CMakeLists ﬁles. While CMake can handle extremely complex projects, for most projects you will ﬁnd this chapter’s contents will tell you all you need to know. CMake is driven by the CMakeLists.txt ﬁles written for a software project. The CMakeLists ﬁles determine everything from what options to put into the cache, to what source ﬁles to compile. ln addition to discussing how to write a CMakeLists ﬁle this chapter will also cover how to make them robust and maintainable. The basic syntax of a CMakeLists.txt ﬁle and key concepts of CMake have already been discussed in chapters 2 and 3. This chapter will expand on those concepts and introduce a few new ones.

本章包含如果编写高效CMakeLists文件的基本方法。这包括基本命令介绍和对于大多数工程所要面对的问题。而且还包括如何转换UNIX或者Windows工程到CMakeLists文件。CMake能处理极为复杂的工程，对于大多数工程，本章介绍的知识涵盖了你所需要的所有内容。CMake由CMakeLists.txt文件驱动。CMakeLists文件确定了所有放入缓存的选项，以及所有需要编译的源文件。本章除了讨论如何编写CMakeLIsts文件还涉及如何使CMakeLists健壮，可维护。在第2，3章已经讨论过了CMakeLists.txt文件的基本语法和CMake的核心概念。本章将扩展这些概念并介绍一些新的知识。

4.1 CMake Syntax

4.1 CMake语法

CMakeLists ﬁles follow a simple syntax consisting of comments, commands, and white space. A comment is indicated using the # character and runs from that character until the end of the line. A command consists of the command name, opening parenthesis, white space separated arguments and a closing parenthesis. All white space (spaces, line feeds, tabs) are ignored except to separate arguments. Anything within a set of double quotes is treated as one argument as is typical for most languages. The backslash can be used to escape characters preventing the normal interpretation of them. The subsequent examples in this chapter will help to clear up some of these syntactic issues. You might wonder why CMake decided to have its own language instead of using an existing one such as Python, Java, or Tcl. The main reason is that we did not want to make CMake require an additional tool to run. By requiring one of these other languages all users of CMake would be required to have that language installed, and potentially a speciﬁc version of that language. This is on top of the language extensions that would be required to do some of the CMake work, for both performance and capability reasons.

CMakeLists文件使用简单的语法，包括注释，命令，和空白符。一个注释是以字符#开始，直到行尾。命令由命令名，小括号，空格分隔的参数组成。所有的空白符（空格，换行符，制表符）除了分隔参数用都会被忽略。可以使用反斜杠作为转义符转义。本章的一些例子会明确这些语法问题。你可能有疑问，为什么CMake要使用自己的语言，而不是Python，Java，或者Tcl这些已经存在的语言。一个主要的原因是我们不想让CMake依赖外部工具运行。如果使用上述语言之一，CMake将需要安装这些语言，而且可能是某些特定版本。CMake如果在这些语言之上扩展，将会有适配问题和效率问题。

4.2 Basic Commands

4.2 基本命令

While the previous chapters have already introduced many of the basic commands for CMakeLists ﬁles, this chapter will review and expand on them. The ﬁrst command the top-level CMakeLists ﬁle should have is the PROJECT command. This command both names the project and optionally speciﬁes what languages will be used by it. Its syntax is as follows:

之前的章节已经介绍了一些CMake的一些基本命令，本章将回顾并详细描述这些命令。CMakeLists文件中第一条命令是PROJECT命令。这条命令指定了工程名和工程所使用的语言（可选）。语法如下：

project (projectname [CXX] [C] [Java] [NONE])

If no languages are speciﬁed then CMake defaults to supporting C and C++. If the NONE language is passed then CMake includes no language speciﬁc support. Whenever C++ language support is speciﬁed then C language support will also be loaded.

如果没有指定使用的语言，CMake默认支持C和C++。如果设置语言为NONE，CMake不支持任何语言。当指定语言为C++时，也将支持C语言。

For each project command that appears in a project, CMake will create a top level IDE project ﬁle. The project will contain all targets that are in the CMakeLists.txt ﬁle, and any of its subdirectories as speciﬁed by the add\_subdirectory command. If the EXCLUDE\_FROM\_ALL option is used in the add\_subdirectory command, then the generated project will not appear in the top level Makeﬁle or IDE project ﬁle. This is useful for generating sub projects that do not make sense as part of the main build process. Consider that a project with a number of examples could use this feature to generate the build ﬁles for each example with one run of CMake, but not have the examples built as part of the normal build process.

对于工程中的每个project命令，CMake将生成一个顶层的IDE工程文件。这个工程会包括CMakeLists.txt文件中的所有目标，和通过add\_subdirectory命令添加的子目录中的所有CMakeLists.txt文件中的所有目标。如果使用add\_subdirectory命令时使用的EXCLUDE\_FROM\_ALL选项，那么对应的工程将不出现在生成的Makefile或IDE工程文件中。如果子目录中的工程对于主构建过程没有太多影响的时候（和主工程没有多大关系）。这回很有用。考虑一下当一个工程有很多例子时就可以使用这一特性运行CMake为每个例子成成构建文件，而不将这些例子作为主构建过程的一部分。

The set command is probably one of the most used commands since it is used for deﬁning and modifying variables and lists. Complimenting the set command are the remove and separate\_arguments commands. The remove command can be used to remove a value from a variable list, while the separate\_arguments command can be used to take a single variable value (as opposed to a list) and break it into a list based on spaces.

set命令用来定义和修改变量和列表，它可能是最常用的命令。remove和separate\_arguments命令作为set命令的补充。remove命令可以用来删除变量列表中的值，separate\_arguments可以将一个值分拆成由空格分隔的列表。

The add\_executable and add\_library commands are the main commands for deﬁning what libraries and executables to build, and what source ﬁles comprise them. For Visual Studio projects the source ﬁles will show up in the IDE as usual, but any header ﬁles the project uses will not be there. To have the header ﬁles show up as well you simply add them to the list of source ﬁles for the executable or library. This can be done for all generators. Any generators that do not use the header ﬁles directly (such as Makeﬁle based generators) will simply ignore them.

add\_executable和add\_library命令主要用来定义程序库和可执行文件，以及它们由哪些源代码组成。对于Visual Studio工程，源码会显示在IDE中，但是工程中的头文件不显示。你需要将头文件添加到源码列表中。对于所有的生成器都可以这样做。不使用头文件目录的生成器（如基于Makefile的生成器）会直接忽略它们。

4.3 Flow Control

4.3 流程控制

In many ways writing a CMakeLists ﬁle is like a writing a program in a simple language. Like most languages CMake provides ﬂow control structures to help you along your way. CMake provides three ﬂow control structures;

在很多方面，编写CMakeList文件就好像使用简单语言编写程序。和大多数语言一样CMake提供控制流结构；

* conditional statements (e.g. if)
* 条件语句（如if）
* looping constructs (e.g. foreach and while)
* 循环结构（如foreach和while）
* procedure deﬁnitions (e.g. macro and function)
* 过程定义（如宏和函数）

First we will consider the if command. In many ways the if command in CMake is just like the if command in any other language. It evaluates its expression and based on that either executes the code in its body or optionally the code in the else clause. For example:

首先我们来看if命令。CMake中的if命令和很多语言中的if语句基本一致。它计算表达式的值并以此判断执行if语句体中的代码还是else从句。比如：

if (FOO)

# do something here

else (FOO)

# do something else

endif (FOO)

One difference you might notice is that the conditional of the if statement is repeated in the else and endif clauses. This is optional and in this book you will see examples of both styles. You could just as well choose to write:

可能你会注意到一点不同，if语句的条件重复出现在else和endif从句中。这是可选的，在本书中你会看到两种写法的例子。选择哪种都可以：

if (FOO)

# do something here

else ()

# do something else

endif ()

When you include conditionals in the else and endif clause they are used to provide additional error checking. As such they must exactly match the original conditional of the if statement. The following code would not work:

当else和endif从句包含条件时，通常会有错误检查。它们必须和if语句的条件一致。下面的例子是错误的：

set (FOO 1)

if (${FOO})

# do something

endif (1)

# ERROR, it doesn't match the original if conditional

Fortunately CMake provides verbose error messages in the case where an if statement is not properly matched with an endif. This should help you to track down any problems with matching conditionals. Providing the conditionals on the else and endif commands also has the added beneﬁt of helping to document your CMakeLists ﬁle. With a long if statement it can be easy to lose track of what if statement the endif is closing. if statements can be nested to any depth, and any command can be used inside of an if or else clause.

幸运的是一旦if语句和endif不匹配，CMake将提供详细的错误信息。这能帮助你找到这类匹配错误。在else和endif中重复条件可以帮助完善CMakeLists文档。比如一个长的if语句，如果没有重复条件，可能不容易找到与之匹配的endif从句。if语句可以嵌套任意层，在if和else从句中可以使用任何命令。

As with many other languages, CMake supports elseif so that you can sequentially test for multiple conditions. For example:

和其他语言一样，CMake支持elseif从句，使你可以顺序测试多个条件。例如：

if (MSVCSO)

# do something here

elseif (MSVC90)

# do something else

elseif (APPLE)

# do something else

endif ()

The if command has a limited set of operations that you can use. It does not support general purpose C style expressions such as ${FOO} && ${BAR} || ${FUBAR}, instead it supports a limited subset of expressions that should work for most cases. Speciﬁcally if supports:

If命令只支持一部分的操作。它不支持C风格的通用表达式如${FOO} && ${BAR} || ${FUBAR}，相反，它支持有限的表达式，覆盖大部分情况。if支持：

if (variable)

True if the variable‘s value is not empty, 0, FALSE, OFF, or NOTFOUND.

当变量的值不是空，0，FALSE，OFF，或者NOTFOUND时为真。

if (NOT variable)

True if the variable‘s value is empty, 0, FALSE, OFF, 0r NOTFOUND

当变量的值是空，0，FALSE，OFF，或者NOTFOUND时为真。

if (variable1 AND variable2)

True if both variables would be considered true individually.

当变量的值都为真时为真。

if (variable1 OR variable2)

True if either variable would be considered true individually.

当有一个变量的值为真时为真。

if (COMMAND command-name)

True if the given name is a command that can be invoked.

当command-name的值是一个可以被调用的命令名时为真。

if (DEFINED variable)

True if the given variable has been set, regardless of what value it was set to.

当变量已经定义时，不管变量的值，均为真。

if (EXISTS ﬁle-name)

if (EXISTS directory-name)

True if the named ﬁle or directory exists.

当文件存在或目录存在时为真。

if (IS\_DIRECTORY name)

if (IS\_ABSOLUTE name)

True if the given name is a directory, or absolute path respectively.

当所给出的是一个目录或者绝对路径时为真。

if (name1 IS\_NEWER\_THAN name2)

True if the ﬁle speciﬁed by name1 has a more recent modiﬁcation time than the ﬁle

speciﬁed by name2.

当文件1的修改日期比文件2新时为真。

if (variable MATCHES regex)

if (string MATCHES regex)

True if the given string or variable's value matches the given regular expression.

当string或者变量的值匹配正则表达式时为真。

Options such as EQUAL, LESS, and GREATER are available for numeric comparisons. STRLESS, STREQUAL, and STRGREATER can be used for lexicographic comparisons. VERSION\_LESS, VERSION\_EQUAL, and VERSION\_GREATER can be used to compare versions of the form major [ .minor[ .patch [ . tweak] ] 1. Similar to C and C++ these expressions can be combined to create more powerful conditionals. For example consider the following conditionals:

选项EQUAL，LESS，和GREATER适用于数字比较。STRLESS, STREQUAL，和STRGREATER适用于文字比较。VERSION\_LESS，VERSION\_EQUAL，和VERSION\_GREATER可以用于比较major[.minor[.patch[.tweak]]形式的版本号。和C，C++类似，这些表达式可以组合成更强大的条件。请看如下例子：

if ((1 LESS 2) AND (3 LESS 4))

message ("sequence of numbers")

endif 0

if (1 AND 3 AND 4)

message ("series of true values")

endif (1 AND 3 AND 4)

if (NOT 0 AND 3 AND 4)

message ("a false value")

endif (NOT 0 AND 3 AND 4)

if (0 OR 3 AND 4)

message ("or statements")

endif (0 OR 3 AND 4)

if (EXISTS ${PROJECT\_SOURCE\_DIR}/Help.txt AND COMMAND IF)

message ("Help exists")

endif (EXISTS ${PROJECT\_SOURCE\_DIR}/Help.txt AND COMMAND IF)

set (fooba 0)

if (NOT DEFINED foobar)

message ("foobar is not defined")

endif (NOT DEFINED foobar)

if (NOT DEFINED fooba)

message ("fooba not defined")

endif (NOT DEFINED fooba)

In compound if statements there is an order of precedence that specifies the order that the operations will be evaluated. For example, in the statement below, the NOT will be evaluated ﬁrst then the AND, not the other way around. Thus the statement will be false and the message never printed. Had the AND been evaluated ﬁrst the statement would be true.

在复杂的if语句中执行顺序决定了表达式的值。下面的例子中，NOT运算比AND运算先执行，所以条件为假，消息不会打印输出。如果AND先执行，则语句为真。

if (NOT 0 AND 0)

message ("This line is never executed")

endif (NOT 0 AND 0)

CMake deﬁnes the order of operations such that parenthetical groups are evaluated ﬁrst, then EXISTS, COMMAND, DEFINED and similar preﬁx operators are evaluated, then any EQUAL, LESS, GREATER, STREQUAL, STRLESS, STRGREATER, and MATCHES operators. The NOT operators are evaluated next, and ﬁnally the AND and OR expressions will be evaluated. With operations that have the same level of precedence, such as AND and OR, they will be evaluated from left to right. Once all of the expressions have been evaluated the ﬁnal result will be tested to see if it is true or false. CMake considers any of the following values to be true: ON, 1, YES, TRUE, Y. The following values are all considered to be false: OFF, 0, NO, FALSE, N, NOTEOUND, \*-NOTFOUND, IGNORE. This test is case insensitive so true, True, and TRUE are all treated the same.

CMake定义了操作的执行顺序，括号内的将先执行，然后是EXIST，COMMAND，DEFINED和类似前缀的操作符，然后是EQUAL，LESS，GREATER，STREQUAL，STRLESS，STRGREATER，和MATCHES操作符。然后是NOT操作符，最后是AND和OR表达式。同样优先级的操作符，从左到右顺序执行。当所有表达式执行完，会判断最终结果是true或者false。CMake认定下面的值为true：ON，1，YES，TRUE，Y。认定下列值为false：OFF，0，NO，FALSE，N，NOTFOUND，\*-NOTFOUND，IGNORE。这些是大小写不敏感的，所以true，True，和TRUE都当做真。

Now let us consider the other ﬂow control commands. The foreach, while, macro, and function commands are the best way to reduce the size of your CMakeLists ﬁles and keep them maintainable. The foreach command enables you to execute a group of CMake commands repeatedly on the members of a list. Consider the following example adapted from VTK:

现在让我们来看一下另一种流程控制。foreach，while，宏和函数命令是节省CMakeLists文件大小和保持可维护性最好的方式。foreach命令允许你对每个列表项重复执行一组CMake命令。请看下面的例子：

foreach (tfile

TestAnisotropicDiffusion2D

TestButterworthLowPass

TestButterworthHighPass

TestCityBIockDistance

TestConvolve

)

additest(${tfile}-image ${VTK\_EXECUTABLE}

${VTK\_SOURCE\_DIR}/Tests/rtlmageTest.tcl

${VTK\_SOURCE\_DIR}/Tests/${tflle}.tcl

-D ${VTK\_DATA\_ROOT}

-V Baseline/Imaging/${tfile}.png

-A ${VTK\_SOURCE\_DIR}/Wrapping/Tcl

)

endforeach ( tfile )

The ﬁrst argument of the foreach command is the name of the variable that will take on a different value with each iteration of the loop. The remaining arguments are the list of values over which to loop. In this example the body of the foreach loop is just one CMake command, add\_test. In the body of the foreach loop any time the loop variable (tfile in this example) is referenced it will be replaced with the current value from the list. In the ﬁrst iteration, occurrences of ${tfile} will be replaced with TestAnisotropicDiffusion2D. In the next iteration, ${tfile} will be replaced with TestButterworthLowPass. The foreach loop will continue to loop until all of the arguments have been processed.

Foreach命令的第一个参数是变量名，它的值是循环中每一项的值。剩下的参数是用于循环的列表。在这个例子中foreach循环执行一条CMake命令，add\_test。在foreach的每次循环中变量（例子中是tfile）的值会被替换为当前列表项的值，在首次循环中${tfile}的值被替换为TestAnisotropicDiffusion2D。下次循环时${tfile}将被替换为TestButterworthLowPass。foreach循环将继续知道所有的参数都被执行。

It is worth mentioning that foreach loops can be nested and that the loop variable is replaced prior to any other variable expansion. This means that in the body of a foreach loop you can construct variable names using the loop variable. In the code below the loop variable tfile is expanded, and then concatenated with \_TEST\_RESULT. That new variable name is then expanded and tested to see if it matches FAILED.

需要注意的是foreach循环可以嵌套，变量可以被之前的其它变量替换。也就是说你可以在foreach循环体中使用循环变量的名字构造变量。在下面的代码中，循环变量tfile被扩展了，后面接上了\_TEST\_RESULT。新变量的名字被用来测试是否匹配FAILED。

if (${${tfile}\_TEST\_RESULT} MATCHES FAILED)

message ("Test ${tfile} failed.")

endif ()

The while command provides for looping based on a test condition. The format for the test expression in the while command is the same as that for the if command described earlier. Consider the following example, which is used by CTest. Note that CTest updates the value of CTEST\_ELAPSED\_TIME internally.

While命令提供了基于条件循环的机制。用于测试的表达式的格式和之前介绍的if命令一致。请看下面CTest中的例子。请注意CTest内部更新CTEST\_ELAPSED\_TIME的值。

#####################################################

# run paraview and ctest test dashboards for 6 hours

#

while (${CTEST\_ELAPSED\_TIME} LESS 36000)

set (START\_TIME ${CTEST\_ELAPSED\_TIME})

ctest\_run\_script ( "dash1\_ParaView\_vs71continuous.cmake" )

ctest\_run\_script ( "dash1\_cmake\_vs71continuous.cmake" )

endwhile ()

The foreach and while commands allow you to handle repetitive tasks that occur in sequence, whereas the macro and function commands support repetitive tasks that may be scattered throughout your CMakeLists ﬁles. Once a macro or function is deﬁned it can be used by any CMakeLists ﬁles processed after its deﬁnition.

Foreach和while命令允许你处理顺序执行的重复任务，而宏和函数命令支持重复任务，但是可能在CMakeLists文件中到处都是。一旦一个宏或者函数定义好，就能在CMakeLists文件处理过程中被使用。

A function in CMake is very much like a function in C or C++. You can pass arguments into it, and the arguments passed in become variables within the function. Likewise some standard variables such as ARGC, ARGV, ARGN, and ARGVO, ARGVi, etc are deﬁned. Within a function you are in a new variable scope, much like when you drop into a subdirectory using the add\_subdirectory command you are in a new variable scope. All the variables that were deﬁned when the function was called are still deﬁned, but any changes to variables or new variables only exist within the function. When the function returns those variables will go away. Put more simply, when you invoke a function a new variable scope is pushed and when it returns that variable scope is popped.

CMake中的函数和C或者C++中的函数很像。你能向它传递参数，参数在函数内变成了变量。在函数中，你处于新的变量作用域中，就好像当你进入使用add\_subdirectory命令进入子目录，你处于新的变量作用域中。当函数被调用时所有的定义的变量还是定义好的，但是变量的改变只在函数内部有效。当函数返回时这些变量消失，更简单的说，当你调用函数，变量作用域压栈，返回时变量作用域出栈。

The ﬁrst argument is the name of the function to deﬁne. All additional arguments are formal parameters to the function.

函数命令的第一个参数是函数名。剩下的参数将作为函数的参数。

function(DetermineTime \_time)

# pass the result up to whatever invoked this

set (${\_time} "1:23:45" PARENT\_SCOPE)

endfunction()

# now use the function we just defined

DetermineTime( current\_time )

if( DEFINED current\_time )

message(STATUS "The time is now: ${current\_time}")

endif()

Note that in this example \_time is used to pass the name of the return variable. The set command is invoked with the value of \_time, which in this example will be current\_time. Finally the set command uses the PARENT\_SCOPE option to set that variable in the parent’s scope instead of the local scope.

请注意这个例子中\_time用于传递返回变量的名字，set命令设置\_time值作为变量名的值，在例子中是current\_time。最后set命令使用PARENT\_SCOPE选项在父作用域中设置变量。

Macros are deﬁned and called in the same manner as functions. The main differences are that a macro does not push and pop a new variable scope, and the arguments to a macro are not treated as variables but are string replaced prior to execution. This is very much like the differences between a macro and a function in C or C++. The ﬁrst argument is the name of the macro to create. All additional arguments are formal parameters to the macro.

宏的定义和调用和函数一致。最主要的区别是宏宏不入栈出栈作用域，传递给宏的参数不会被当成变量，而是字符串。这个C或者C++中的宏和函数的区别非常像。第一个参数是宏的名字。剩下的参数是传入宏的参数。

# define a simple macro

macro (assert TEST COMMENT)

if (NOT ${TEST))

message ("Assertion failed: ${COMMENT}")

endif (NOT ${TEST})

endmacro (assert)

# use the macro

find\_library (FOO\_LIB foo /usr/local/lib)

assert ( ${FOOiLIB) "Unable to find library foo" )

The simple example above creates a macro called assert. The macro is deﬁned to take two arguments. The ﬁrst argument is a value to test and the second argument is a comment to print out if the test fails. The body of the macro is a simple if command with a message command inside of it. The macro body ends when the endmacro command is found. The macro can be invoked simply by using its name as if it were a command. In the above example if FOO\_LIB was not found a message would be displayed indicating the error condition.

上面这个简单的例子定义了一个叫assert的宏。宏被定义有两个参数。第一个参数是要测试的数，第二个参数是当测试失败打印的解释。宏的语句体是if命令在语句体中是message命令。宏的语句体由endmacro命令结束。可以像使用命令那样通过名字调用宏。上面的例子中，如果FOO\_LIB没有找到，则输出一条消息指明错误。

The macro command also supports deﬁning macros that take variable argument lists. This can be useful if you want to deﬁne a macro that has optional arguments or multiple signatures. Variable arguments can be referenced using ARGC and ARGVO, ARGVT, etc., instead of the formal parameters. ARGV0 represents the ﬁrst argument to the macro, ARGV1 represents the next, and so forth. You can even use a mixture of formal arguments and variable arguments, as shown in the example below.

宏命令支持定义一个宏接受参数列表。如果你想要定义一个宏有可选的参数或者多个参数，这将很有用。变量参数可以使用ARGC和ARGVO，ARGVT等等引用，代替正常的参数。ARGV0表示宏的第一个参数，ARGV1表示下一个参数，以此类推。你也可以混合使用正常的参数和变量参数，如下面的例子。

# define a macro that takes at least two arguments

# (the formal arguments) plus an optional third argument

macro (assert TEST COMMENT)

if (NOT ${TEST})

message ("Assertion failed: ${COMMENT}")

# if called with three arguments then also write the

# message to a file specified as the third argument

if (${ARGC} MATCHES 3)

file (APPEND ${ARGV2} "Assertion failed: ${COMMENT}")

endif (${ARGC} MATCHES 3)

endif (NOT ${TEST})

endmacro (ASSERT)

# use the macro

find\_library (FOO\_LIB foo /usr/local/lib)

assert ( ${FOO\_LIB} "Unable to find library foo" )

In this example the two required arguments are TEST and COMMENT. These required arguments can be referenced by name, as they are in this example, or they can be referenced ‘using ARGV0 and ARGV1. If you want to process the arguments as a list you can use the ARGV and ARGN variables. ARGV (as opposed to ARGV0, ARGV1, etc) is a list of all the arguments to the macro, while ARGN is a list of all the arguments after the formal arguments. Inside your macro you can use the foreach command to iterate over ARGV or ARGN as desired.

在这个例子中宏assert有两个参数TEST和COMMENT。你可以像例子中那样使用名字引用这两个参数（${TEST}，${COMMENT}），或者使用ARGV0和ARGV1。如果你想把参数当做一个列表，你可以使用ARGV和ARGN变量。ARGV（与ARGV0和ARGV1相对）表示宏的参数列表，ARGN表示正常参数后的所有参数列表。在宏中你可以使用foreach命令访问ARGV和ARGN。

CMake has two commands for interrupting the processing flow. The break command will break out of a foreach or while loop before it would normally end. The return command will return from a function or listﬁle before the function or listfile has reached its end.

CMake有两个命令可以中断循环。break命令可以在正常结束前跳出foreach或者loop循环。返回命令将在函数结尾前从函数返回。

4.4 Regular Expressions

4.4 正则表达式

A few CMake commands, such as if and string, make use of regular expressions, or can take a regular expression as an argument. In its simplest form, a regular-expression is a sequence of characters used to search for exact character matches. However, many times the exact sequence to be found is not known, or only a match at the beginning or end of a string is desired. Since there are several different conventions for specifying regular expressions CMake’s standard is described below. The description is based on the open source regular expression class from Texas Instruments, which is used by CMake for parsing regular expressions.

有些CMake命令，比如if和string，会使用正则表达式，或者将正则表达式作为一个参数。正则表达式最简单的方式是按每个字符顺序匹配。但是很多时候精确查找并不能满足需求。下面介绍一些CMake中标准正则表达式的用法。这些介绍基于德州仪器的正则表达式开源项目，CMake中使用它解析正则表达式。

Regular expressions can be speciﬁed by using combinations of standard alphanumeric characters and the following regular expression meta characters:

正则表达式由标准的文字数字字符和下面介绍的元字符组成：

^ Matches at beginning of a line or string.匹配行起始。

$ Matches at end of a line or string.匹配行结束。

. Matches any single character other than a newline.匹配一行内的单个字符。

[ ] Matches any character(s) inside the brackets.匹配括号内的一个或多个字符。

[^] Matches any character(s) not inside the brackets.匹配不在括号内的一个或多个字符。

[-] Matches any character in range on either side of a dash.匹配范围内的一个字符。

\* Matches preceding pattern zero or more times.匹配零个或多个字符。

+ Matches preceding pattern one or more times.匹配一个或多个字符。

? Matches preceding pattern zero or once only.匹配零个或一个字符

() Saves a matched expression and uses it in a later replacement.保存小括号中匹配的内容，后面引用。

(|) Matches either the left or right side of the bar. 匹配左边或右边。

Note that more than one of these meta-characters can be used in a single regular expression in order to create complex search patterns. For example, the pattern [^ab1-9] indicates to match any character sequence that does not begin with the characters a or "b" or numbers in the series one through nine. The following examples may help clarify regular expression usage:

请注意在一个正则表达式中可以有多个元字符，组成复杂的搜索模式。例如[^ab1-9]表示匹配字符非a或者b或者1到9数字。下面的例子明确了正则表达式的用法：

* The regular expression "^hello" matches a "hello" only at the beginning of a search string. It would match "hello there", but not "hi,\nhello there".
* 正则表达式"^hello"匹配以hello开始的字符串。它匹配"hello there"，不匹配"hi,\nhello there"。
* The regular expression "long$" matches a "long" only at the end of a search string. It would match "so long", but not "long ago".
* 正则表达式"long$"匹配以long结尾的字符串。它匹配"so long"，不匹配"long ago"。
* The regular expression "t..t..g" will match anything that has a "t", then any two characters, another "t", any two characters, and then a "g". It would match "testing" or "test again", but would not match "toasting".
* 正则表达式"t..t..g"匹配"t"后接两个任意字符，接"t"，后接两个任意字符，后接"g"，它匹配"testing"或"test again"，不匹配"toasting"。
* The regular expression "[l-9ab]" matches any number one through nine, and the characters "a" and "b". It would match "hello 1" or "begin", but would not match "no-match".
* 正则表达式"[l-9ab]"匹配1到9任意数字或者a或者b，它匹配"hello 1"或者"begin"，不匹配"no-match"。（错误）
* The regular expression "[^1-9ab]" matches any character that is not a number one through nine, or an "a" or "b". It would NOT match "lab2" or "b2345a", but would match "no-match".
* 正则表达式"[^1-9ab]"匹配非1到9的数组和非字符a或者b。它不匹配"lab2"和"b2345a"，匹配"no-match"。（错误）
* The regular expression "br\* " matches something that begins with a "b", is followed by zero or more "r"s, and ends in a space. It would match "brrrrr " and "b ", but would not match "brrh ".
* 正则表达式"br\* "匹配字符"b"，后面接零个或多个"r"，后接空格。它匹配"brrrrr "和"b "，不匹配"brrh "。
* The regular expression "br+ " matches something that begins with a "b", is followed by one or more "r"s, and ends in a space. It would match "brrrrr ", and "br ", but would not match "b " or "brrh ".
* 正则表达式"br+ "匹配字符"b"，后面接一个或多个"r"，后接空格。它匹配"brrrrr "和"br "，不匹配"b "或者"brrh "。
* The regular expression "br? " matches something that begins with a "b", is followed by zero or one "r"s, and ends in a space. It would match "br ”, and "b ”, but would not match "brrrr " or "brrh ".
* 正则表达式"br？ "匹配字符"b"，后面接零个或一个"r"，后接空格。它匹配"br "和"b ”，不匹配"brrrr "或者"brrh "。
* The regular expression "(..p)b" matches something ending with pb and beginning with Whatever the two characters before the ﬁrst p encountered in the line were. It would ﬁnd "repb" in "rep drepaqrepb". The regular expression "(..p)a" would find "repa qrepb” in "rep drepa qrepb
* 正则表达式"(..p)b"匹配pb结尾，前面两个任意字符。它能找到"rep drepaqrepb"中的"repb"，正则表达式"(..p)a"能找到"rep drepa qrepb"中的"repa qrepb"
* The regular expression "d(..p)" matches something ending with p, beginning with d, and having two characters in between that are the same as the two characters before the first p encountered in the line. It would match "drepa qrepb" in "rep drepa qrepb".
* 正则表达式"d(..p)"匹配d开始，接两个字符接p。它匹配"rep drepa qrepb"中的"drepa qrepb"

4.5 Checking Versions of CMake

4.5 检测CMake版本号

CMake is an evolving program and as new versions are released, new features or commands may be introduced. As a result, there may be instances where you might want to use a command that is in a current version of CMake but not in previous versions. There are a couple of ways to handle this. One option is to use the if command to check whether a new command exists. For example:

CMake是一个不断迭代新版本的程序，会有新的特性和新的命令出现。所以你使用的命令可能对CMake版本有限制。有很多方法处理这种问题。其中一个是使用if命令判断命令是否存在。例如：

# test if the command exists

if (COMMAND some\_new\_command)

# use the command

some\_new\_command ( ARGS...)

endif (COMMAND some\_new\_command)

The above approach should work in most cases, but if you need more information you can test against the actual version of CMake that is being run by evaluating the CMAKE\_VERSION variables, as in the following example:

上述方式适用于大多数情况，如果你需要具体的CMake版本号，你可以使用变量CMAKE\_VERSION，请看下面的例子：

# look for newer versions of CMake

if (${CMAKE\_VERSION} VERSION\_GREATER l.6.l)

# do something special here

Endif ()

When writing your CMakeLists files you might decide that you do not want to support old versions of CMake. To do this you can place the following command at the top of your CMakeLists file:

当编写你CMakeLists文件时你可能不打算支持老版本的CMake。你可以在CMakeLists文件最开始使用下面的命令：

cmake\_minimum\_required (VERSION 2.2)

This indicates that the person running CMake on your project must have at least CMake version 2.2. If they are running an older version of CMake then an error message will be displayed telling them that the project requires at least the speciﬁed version of CMake.

这表示运行在你的工程上运行CMake的用户至少要使用2.2以上版本的CMake。如果你使用低于此版本的CMake，将会收到一条错误信息显示该工程至少需要指定版本的CMake。

Finally, in some cases a new release of CMake might come out that no longer supports some commands you were using (although we try to avoid this). In these cases you can use CMake policies, as discussed in section 4.7.

最后，某些情况下新的发布版本的CMake可能不在支持某种你所用到的命令。你是你需要用到CMake策略，这会在第4.7节讨论。

4.6 Using Modules

4.6 使用模块

Code reuse is a valuable technique in software development and CMake has been designed to support it. Allowing CMakeLists ﬁles to make use of reusable modules enables the entire CMake community to share reusable sections of code. For CMake these sections of code are called modules and can be found in the Modules subdirectory of your CMake installation. Modules are simply sections of CMake commands put into a ﬁle. They can then be included into other CMakeLists ﬁles using the include command. For example, the following commands will include the FindTCL module from CMake and then add the Tcl library to the target FOO.

代码重用在软件开发中是一项很有用的技术。CMake也支持。允许CMakeLists文件使用可重用的模块可以让整个CMake社区共享可重用代码。对于CMake来讲这部分代码称为模块，你可以在CMake安装的文件夹的子目录中找到。模块是一些放到一个文件中的简单的CMake命令代码块。你能够使用include命令在其他CMakeLists文件中包含它们。下面的命令在CMake中包含FindTCL模块，并为FOO目标添加Tcl程序库。

include (FindTCL)

target\_link\_libraries (FOO ${TCL\_LTBRARY})

A module’s location can be speciﬁed using the full path to the module ﬁle, or by letting CMake ﬁnd the module by itself. CMake will look for modules in the directories speciﬁed by CMAKE\_MODULE\_PATH and if it cannot ﬁnd it there, it will look in the Modules subdirectory of CMake. This way projects can override modules that CMake provides, to customize them for their needs. Modules can be broken into a few main categories:

可以使用模块文件的绝对地址指定一个模块，或者让CMake自己查找模块。CMake会在CMAKE\_MODULE\_PATH指定的目录中查找模块，如果没有找到，它会在指定目录的子目录中查找。通过这种方式，可以覆盖CMake提供的模块，按需求定制它们。模块可以分成几大类：

Find Modules 查找模块

These modules determine the location of software elements such as header ﬁles or libraries.

这类模块确定工程中头文件或者程序库的位置。

System lntrospection Modules 系统自检模块

These modules test the system for properties such as the size of a ﬂoat, support for ANSI C++ streams, etc.

这类模块检测系统属性，如float字节数，是否支持ANSI C++ streams，等等。

Utility Modules 工具模块

These modules provide added functionality such as support for situations where one CMake project depends on another and other convenience routines.

这类模块提供额外功能，如支持CMake依赖工程的定位，和其他一些有用的功能。

Now let us consider these three types of modules in more detail. CMake includes a large number of Find modules. The purpose of a Find module is to locate software elements such as header or library ﬁles. If they cannot be found then they provide a cache entry so that the user can set the required properties. Consider the following module that ﬁnds the PNG library.

现在让我们具体看看这三类模块。CMake包含一大批查找模块。查找模块是用来定位头文件或者程序库的。如果无法找到，它们提供了缓存项供用户自己设置所需属性项。请看下面查找PNG程序库的模块。

#

# Find the native PNG includes and library

#

# This module defines

# PNG\_INCLUDE\_DIR, where to find png.h, etc.

# PNG\_LIBRARIES, the libraries to link against to use PNG.

# PNG\_DEFINITIONS - You should call

# add\_definitions (${PNG\_DEFINITIONS}) before compiling code

# that includes png library files.

# PNG\_FOUND, If false, do not try to use PNG.

# also defined, but not for general use are

# PNG\_LIBRARY, where to find the PNG library.

# None of the above will be defined unless zlib can be found.

# PNG depends on Zlib

include (FindZLIB.cmake)

if (ZLIB\_FOUND)

find\_path (PNG\_PNG\_INCLUDE\_DIR png.h

/usr/local/include

/usr/include

)

find\_library (PNG\_LIBRARY png

/usr/lib

/usr/local/lib

)

if (PNG\_LIBRARY)

if (PNG\_PNG\_INCLUDE\_DIR)

# png.h includes zlib.h. Sigh.

set (PNG\_INCLUDE\_DIR

${PNG\_PNG\_INCLUDE\_DIR} ${ZLIB\_INCLUDE\_DIR} )

set (PNG\_LIBRARIES ${PNG\_LIBRARY} ${ZLIB\_LIBRARY})

set (PNG\_FOUND "YES")

if (CYGWIN)

if (BUILD\_SHARED\_LIBS)

# No need to define PNGﬁUSE\_DLL here, because

# it's default for Cygwin.

(BUILD\_SHARED\_LIBS)

set (PNG\_DEFINITIONS -DPNG\_STATIC)

endif (BUILD\_SHARED\_LIBS)

endif (CYGWIN)

endif ()

endif ()

endif ()

The top of the module clearly documents what the module will do and what variables it will set. Next it includes another module, the FindZLIB module, that determines if the ZLib library is installed. Next, if ZLib is found, the find\_path command is used to locate the PNG include ﬁles. The ﬁrst argument is the name of the variable to store the result in, the second argument is the name of the header ﬁle to look for, the remaining arguments are paths to search for the header ﬁle. If it is not found in the system path then the variable is set to PNG\_PNG\_INCLUDE\_DIR-NOTFOUND, allowing the user to set it.

这个模块首先声明模块的功能以及将要设置的变量。之后它包含了另一个模块，FindZLIB模块，该模块判断是否安装了ZLib程序库，如果找到了ZLib程序库，就使用find\_path命令定位PNG include文件。该变量的第一个参数是返回值变量的名字，第二个参数是需要查找的头文件的名字，剩下的参数是查找路径。如果没有在系统路径下找到，则设置PNG\_PNG\_INCLUDE\_DIR为NOTFOUND，允许用户去设置它。

Note that the paths to search for the PNG library can include hard coded directories, registry entries, and directories made up of other CMake variables. The next command ﬁnds the actual PNG library using the find\_library command. This command performs additional checks to ﬁnd a proper library name, such as adding "lib" in front of the name and ".so" at the end of the name on Linux systems.

这里，查找路径可以包含硬编码路径，注册项，和其它CMake变量。然后使用find\_library命令查找PNG程序库。该命令会添加前缀后缀查找对应的程序库，如在linux系统中添加"lib"前缀和".so"后缀查找。

After the ﬁnd calls, some CMake variables are set that developers using FindPNG can use in their projects (such as the include paths, and library name). Finally PNG\_FOUND is set correctly, which lets developers know that the PNG library was properly found.

开发者在自己的工程中使用FindPNG，当调用后会设置一些CMake变量，如果找到PNG程序库，PNG\_FOUND将被正确设置。

This structure is fairly common to all Find modules in CMake. Usually they are fairly short, but in some cases, such as FindOpenGL they can be a few pages long. They are normally independent of other modules, but there is no restriction on the use of other modules.

这种结构在查找模块中很常见。一般的它们都很短，但是像FindOpenGL 这种也可能有几页长。它们一般不依赖于其他模块，但也不限定不能使用其他模块。

System introspection modules provide information about the target platform or compiler. Many of these modules have names preﬁxed with Test or Check, such as TestBigEndian and CheckTypeSize. Many of the system introspection modules actually try to compile code in order to determine the correct result. In these cases the source code is usually named the same as the module, but with a .c or .cxx extension. System introspection modules are covered in more detail in chapter 5.

系统自检模块提供目标平台或者编译器的相关信息。这类模块一般都以Text或者Check等前缀命名，如TestBigEndian和CheckTypeSize。许多系统自检模块实际需要编译代码去确认结果。这种情况下需要编译的源代码通常和模块同名，使用.c或.cxx扩展名。系统自检模块将会在第五章详细讨论。

CMake includes a few Utility modules to help make using CMake a little easier. CMakeExportBuildSettings and CMakeImportBuildSettings provide tools to help verify that two C++ projects are compiled with the same compiler and key ﬂags. The CMakePrintSystemInformation module prints out a number of key CMake settings to aid in debugging.

CMake包含一些工具模块以方便使用。CMakeExportBuildSettings和CMakeImportBuildSettings模块提供确认两个C++工程使用同意编译器和标志位的工具。CMakePrintSystemInformation模块打印CMake设置的用于调试的一组信息。

Using CMake with SWIG

使用SWIG

One example of how modules can be used is to look at wrapping your C/C++ code in another language using SWIG. SWIG (Simpliﬁed Wrapper and Interface Generator) www.swig.org is a tool that reads annotated C/C++ header ﬁles, and creates wrapper code (glue code) in order to make the corresponding C/C++ libraries available to other programming languages such as Tcl, Python, or Java. CMake supports SWIG with the find\_package command. Although SWIG can be used from CMake using custom commands, the SWIG package provides several macros that make building SWIG projects with CMake simpler. To use the SWIG macros, ﬁrst you must call the find\_package command with the name SWIG. Then you need to include the ﬁle referenced by the variable SWIG\_USE\_FILE. This will deﬁne several macros and set up CMake to easily build SWIG based projects.

一个如何使用模块的例子是使用SWIG包装C/C++代码成其他语言。SWIG(Simplified Wrapper and Interface Generator) [www.sqig.org是一个工具，它通过读取带注解的C/C++头文件，生成包装代码（胶水代码），使得C/C++程序库可以被其他语言（Tcl，Python，或者Java）使用。CMake使用find\_package命令支持SWIG。SWIG可以通过相对应的命令在CMake中使用，它也提供了一些宏定义使得使用CMake构建SWIG工程更加容易。要使用SWIG宏，首先要使用find\_package命令查找SWIG。然后要include](http://www.sqig.org是一个工具，它通过读取带注解的C/C++头文件，生成包装代码（胶水代码），使得C/C++程序库可以被其他语言（Tcl，Python，或者Java）使用。CMake使用find_package命令支持SWIG。SWIG可以通过相对应的命令在CMake中使用，它也提供了一些宏定义使得使用CMake构建SWIG工程更加容易。要使用SWIG宏，首先要使用find_package命令查找SWIG。然后要include) 变量SWIG\_USE\_FILE对应的文件。这将定义SWIG宏方便CMake构建基于SWIG的工程。

Two very useful macros are SWIG\_ADD\_MODULE and SWIG\_LINK\_LIBRARIES. SWIG\_ADD\_MODULE works much like the add\_library command in CMake. The command is invoked like this:

两个非常有用的SWIG宏是SWIG\_ADD\_MODULE和SWIG\_LINK\_LIBRARIES。SWIG\_ADD\_MODULE用起来很像add\_library命令。调用方式如下：

SWIG\_ADD\_MODULE (module\_name language source1 source2 ... sourceN)

The ﬁrst argument is the name of the module to create. The next argument is the target language SWIG is producing a wrapper for. The rest of the arguments consist of a list of source ﬁles used to create the shared module. The big difference is that SWIG . i interface ﬁles can be used directly as sources. The macro will create the correct custom commands to run SWIG, and generate the C or C++ wrapper code from the SWIG interface ﬁles. The sources can also be regular C or C++ ﬁles that need to be compiled in with the wrappers.

第一个参数是要创建的模块名。第二个参数是要支持的目标语言。后面的参数是用于生成共享模块的源代码列表。最大的不同是SWIG .i接口文件可以直接当成源文件使用。宏会创建正确的命令去运行SWIG，并且生成从SWIG接口文件生成包装代码。源文件也包含需要和包装代码一起编译的C或者C++文件。

The SWIGALINK\_LIBRARIES macro is used to link support libraries to the module. This macro is used because depending on the language being wrapped by SWIG, the name of the module may be different. The actual name of the module is stored in a variable called SWIG\_MODULE\_${name}\_REAL\_NAME where ${name} is the name passed into the SWIG\_ADD\_MODULE macro. For example, SWIG\_ADD\_MODULE (foo tcl foo.i) would create a variable called SWIG\_MODULE\_foo\_REAL\_NAME which would contain the name of the actual module created.

宏SWIGALINK\_LIBRARIES用来将支持的程序库连接到模块。需要使用这个宏因为根基包装适配的语言不同，模块的名字也不同。真正的名字保存在变量SWIG\_MODULE\_${name}\_REAL\_NAME中，${name}是传入宏SWIG\_ADD\_MODULE中的名字。例如，SWIG\_ADD\_MODULE (foo tcl foo.i)将会创建一个变量SWIG\_MODULE\_foo\_REAL\_NAME，它对应真正创建的模块的名字。

Now consider the following example that uses the SWIG example found in SWIG under Examples/python/Class.

请参考下面使用SWIG的例子 它在SWIG的Examples/python/Class目录下。

# Find SWIG and include the use swig file

find\_package (SWIG REQUIRED)

include (${SWIG\_USE\_FILE})

# Find python library and add include path for python headers

find\_package (PythonLibs)

Include\_directories (${PYTHON\_INCLUDE\_PATH})

# set the global swig flags to empty

set (CMAKE\_SWIG\_FLAGS "")

# let swig know that example.i is c++ and add the -includeall

# flag to swig

set\_source\_files\_properties (example.i PROPERTIES CPLUSPLUS ON)

set\_source\_files\_properties (example.i

PROPERTIES SWIG\_FLAGS "-includeall")

# Create the swig module called example

# using the example.i source and example.cxx

# swig will be used to create wrap\_example.cxx from example.i

SWIG\_ADD\_MODULE (example python example.i example.cxx)

SWIG\_LINK\_LIBRARIES (example ${PYTHON\_LIBRARIES})

This example ﬁrst uses find\_package to locate SWIG. Next it includes the SWIG\_USE\_FILE deﬁning the SWIG CMake macros. Then it ﬁnds the Python libraries and sets up CMake to build with the Python library. Notice that the SWIG input ﬁle example.i is used like any other source ﬁle in CMake, and properties are set on the ﬁle telling SWIG that the ﬁle is C++ and that the SWIG ﬂag -includeall should be used when running SWIG on that source ﬁle. The module is created by telling SWIG the name of the module, the target language and the list of source ﬁles. Finally, the Python libraries are linked to the module.

这个例子首先使用find\_package查找SWIG。之后include SWIG\_USE\_FILE文件定义SWIG CMake宏。然后查找Python程序库并且设置CMake供构建时使用。请注意SWIG输入文件example.i在CMake中被当做源文件使用，并且设置属性项告诉SWIG该文件是C++文件。在使用SWIG编译源文件时使用-includeall标志位。使用SWIG\_ADD\_MODULE创建模块，并告诉SWIG模块名，目标语言和源文件列表。最后Python程序库连接到模块。

Using CMake with Qt

使用Qt

Projects using the popular widget toolkit Qt from Nokia, qt.nokia.com, can be built with CMake. CMake supports multiple versions of Qt, including versions 3 and 4. The ﬁrst step is to tell CMake what version(s) of Qt to look for. Many Qt applications are designed to work with Qt3 or Qt4, but not both. If your application is designed for Qt4 then you can use the FindQt4 module, for Qt3 you should use the FindQt3 module. If your project can work with either version of Qt then you can use the generic FindQt module. All of the modules provide helpful tools for building Qt projects. The following is a simple example of building a project that uses Qt4.

工程中使用流行的Nokia的小部件工具包Qt，qt.nokia.com，可以使用CMake构建。CMake支持多版本的Qt，包括Qt3和Qt4。首先需要告诉CMake使用什么版本的Qt。去多Qt应用设计为既支持Qt3又支持Qt4，但是不能同时支持。如果你的应用基于Qt4设计，那你可以使用FindQt4模块,对于Qt3使用FindQt3模块。如果你的工程支持任意Qt版本，你可以使用通用的FindQt模块。所有的模块都提供帮助工具构建Qt工程。下面是使用Qt4构建个孔撑得简单的例子。

find\_package ( Qt4 )

if (QT4\_FOUND)

include (${QT\_USE\_FILE})

# what are our ui files?

set (QTUI\_SRCS qtwrapping.ui)

QT4\_WRAP\_UI (QTUI\_H\_SRCS ${QTUI\_SRCS})

QT4\_WRAP\_CPP (QT\_MOC\_SRCS TestMoc.h)

add\_library (myqtlib ${QTUI\_H\_SRCS} ${QT\_MOC\_SRCS})

target\_link\_libraries (myqtlib ${QT\_LIBRARIES} )

add\_executable (qtwrapping qtwrappingmain.cxx)

target\_link\_libraries (qtwrapping myqtlib)

endif (QT4\_FOUND)

Using CMake with FLTK

使用FLTK

CMake also supports the The Fast Light Toolkit (FLTK) with special FLTK CMake commands. The FLTK\_WRAP\_UI command is used to run the ﬂtk ﬂuid program on a .ﬂ ﬁle and produce a C++ source ﬁle as part of the build. The following example shows how to use FLTK with CMake.

CMake也支持快速轻量级工具包（FLTK），使用对应的CMake命令。FLTK\_WRAP\_UI命令用来对.fl文件运行fltk程序，作为构建的一部分，它生成一个C++源文件。下面的例子显示如何在CMake中使用FLTK。

find\_package (FLTK)

if (FLTK\_FOUND)

set (FLTK\_SRCS

fltkl.fl

)

FLTK\_WRAP\_UI (wraplibFLTK ${FLTKiSRCSH})

add\_library (wraplibFLTK ${wraplibFLTK\_UI\_SRCS})

endif (FLTK\_FOUND)

4.7 Policies

4.7 策略

For various reasons, sometimes a new feature or change is made to CMake that is not fully backwards compatible with older versions of CMake. This can create problems when someone tries to use an old CMakeLists ﬁle with a new version of CMake. To help both end users and developers through such issues, we have introduced policies. Policies are a mechanism in CMake to help improve backwards compatibility and track compatibility issues between different versions of CMake.

基于某种原因，有时CMake新的特性或变动可能不向下兼容老的版本。当有些人在新版本CMake下使用老的CMakeLIsts文件，可能产生错误。为了处理这种情况，我们需要介绍策略。策略是CMake中的一种机制，它能改善版本兼容性，并追踪不同版本的兼容问题。

Design Goals 设计目标

There were four main design goals for the CMake policy mechanism:

CMake策略机制基于4个目标设计：

1. Existing projects should build with versions of CMake newer than that used by the

project authors.

1. 当前工程应该使用比工程作者新的CMake版本构建。

* Users should not need to edit code to get the projects to build.
* 用户构建工程不用修改代码。
* Warnings may be issued but the projects should build.
* 可以有警告，但工程可以构建。

1. Correctness of new interfaces or bugs ﬁxes in old interfaces should not be inhibited by compatibility requirements. Any reduction in correctness of the latest interface is not fair on new projects.

3. Every change made to CMake that may require changes to a project’s CMakeLists ﬁles should be documented.

3. 任何CMakeLists文件的修改都应该有文档描述。

* Each change should also have a unique identiﬁer that can be referenced by warning and error messages.
* 每个修改都应该有唯一的可以标识符，以便它可以被警告或出错信息引用。
* The new behavior is enabled only when the project has somehow indicated it is supported.
* 只有当工程有明确标识支持某些功能时，它才是可用的。

1. We must be able to eventually remove code that implements compatibility with ancient CMake versions.
2. 支配老旧版本的CMake代码最后必须能够被移除。

* Such removal is necessary to keep the code clean and to allow for internal refactoring.
* 这种移除对于保证代码整洁和内部重构是必须的。
* After such removal, attempts to build projects written for ancient versions must fail with an informative message.
* 移除后，使用老旧版本CMake构建工程失败时，必须有相关信息。

All policies in CMake are assigned a name of the form CMPNNNN where NNNN is an integer value. Policies typically support both an old behavior that preserves compatibility with earlier versions of CMake, and a new behavior that is considered correct and preferred for use by new projects. Every policy has documentation detailing the motivation for the change, and the old and new behaviors Projects may conﬁgure the setting of each policy to request old or new behavior. When CMake encounters user code that may be affected by a particular policy it checks to see whether the project has set the policy. If the policy has been set (to OLD or NEW) then CMake follows the behavior speciﬁed. If the policy has not been set then the old behavior is used, but a warning is issued telling the project author to set the policy.

CMake中所有的策略都已CMPNNNN的形式命名，NNNN有数字组成。策略一般支持旧的行为（保证CMake旧版本的兼容性）和新的行为（推荐用户在新的工程中使用）。每个策略都有文档详细描述修改的目的，以及工程中新的或者老的行为，以及如何配置策略。当CMake遇到用户代码受特定策略的影响时，它会检测该工程是否设置了这个策略。如果设置（是OLD还是NEW）了那么CMake就按照设置的行为工作。如果没有设置，默认使用老的行为，然后提示工程作者指明策略。

There are a couple ways to set the behavior of a policy. The quickest way is to set all policies to a version corresponding to the release version of CMake for which the project was written. Setting the policy version requests the new behavior for all policies introduced in the corresponding version of CMake or earlier. Policies introduced in later versions are marked as not set in order to produce proper warning messages. The policy version is set using the cmake\_policy command's VERSION signature. For example, the code

cmake\_policy(VERSION 2.6)

有很多种方法设置策略的行文。最快速的方式是对于某一工程使用某一特定CMake版本设置左右的策略。设置版本对应的策略需要知道对应版本和以前版本的所有新的行为。对于老版本的策略会被标志位没有设置，这样可以产生适合的警告消息。使用cmake\_policy命令设置策略版本。例如，

cmake\_policy(VERSION 2.6)

will request the new behavior for all policies introduced in CMake 2.6 or earlier. The cmake\_minimum\_required command will also set the policy version, which is convenient for use at the top of projects. A project should typically begin with the lines

对于CMake2.6或者更老版本的所有策略需要新的行为。cmake\_minimum\_required命令也会设置策略版本，这在工程开始就能看到。一个CMakeLists文件应该以这几行开始。

cmake\_minimum\_required (VERSION 2.6)

project (MyProject)

# ...code using CMake 2.6 policies

Of course one should replace "2.6" with whatever version of CMake you are currently writing to. You can also set each policy individually if you wish. This is sometimes helpful for project authors who want to incrementally convert their projects to use the new behavior, or silence warnings about dependence on old behavior. The cmake\_policy command's SET option may be used to explicitly request old or new behavior for a particular policy.

你需要将版本号"2.6"替换为你想要支持的版本号。如果愿意你也可以单个设置策略。这对于想要慢慢支持新特性的或者对于旧特性的依赖不与警告的工程作者来说是很有用的。cmake\_policy命令设置可以明确对于某些策略的新或者旧特性的需求。

For example, CMake 2.6 introduced policy CMP0002, which requires all logical target names to be globally unique (duplicate target names previously worked in some cases by accident but were not diagnosed). Projects using duplicate target names and working accidentally will receive warnings referencing the policy. The warnings may be silenced by the code

举例来说，CMake 2.6版本的策略CMP0002，需要所有逻辑目标以全局唯一名字命名（重名的目标在之前的版本某种情况下可能能正常工作，但是不会被发现）。工程使用重名的目标有时能正常工作，但会收到警告。警告可能被设置成静默

cmake\_policy (SET CMP0002 OLD)

which explicitly tells CMake to use the old behavior for the policy (silently accept duplicate target names). Another option is to use the code

上面的代码明确设置策略CMP0002使用旧行为（对重名目标保持沉默）。另外一个选项是

cmake\_policy (SET CMP0002 NEW)

to explicitly tell CMake to use new behavior and produce an error when a duplicate target is created. Once this is added to the project it will not build until the author removes any duplicate target names.

上述代码明确告知CMake使用新行为，当创建重名目标时给出报错。一旦这条加入到CMakeLists文件，当有重名目标时将无法构建。

When a new version of CMake is released that introduces new policies it will still build old projects, because by default they do not request NEW behavior for any of the new policies. When starting a new project one should always specify the most recent release of CMake to be supported as the policy version level. This will make sure that the project is written to work using policies from that version of CMake and not using any old behavior. If no policy version is set CMake will warn and assume a policy version of 2.4. This allows existing projects that do not specify cmake\_minimum\_required to build as they would have with CMake 2.4.

当CMake发布了一个新版本，使用新的策略仍然可以构建旧工程，默认情况下对于新的策略不使用新行为。当创建一个新工程时应该指定使用最新版本的CMake对应的策略。这样可以确保使用的策略从该CMake版本后不使用旧行为。如果没有设置策略版本，CMake将警告并假定策略版本是2.4。这将允许已经存在的工程不是使用cmake\_minimum\_required时默认使用2.4以上版本构建。

4.8 Linking Libraries

4.8 连接程序库

In CMake 2.6 and later a new approach to generating link lines for targets has been implemented. Consider these libraries:

在CMake 2.6和之后的版本中已经实现了新的为目标生成连接命令行的方法。请看下面的程序库：

/path/to/libfoo.a

/path/to/libfoo.so

Previously if someone wrote

target\_link\_libraries (myexe /path/to/libfoo.a)

CMake would generate this code to link it:

CMake会生成如下代码连接它：

... -L/path/to -Wl,-Bstatic -lfoo -Wl,-Bdynamic ...

This worked most of the time, but some platforms (such as Mac OS X) do not support the -Bstatic or equivalent ﬂag. This made it impossible to link to the static version of a library without creating a symlink in another directory and using that one instead. Now CMake will generate this code:

这在大多数时候有效，但是某些平台（如Mac OS X）不支持-Bstatic或者等效的标志位。这使得不在其他目录下生成软连接作为替换而连接静态程序库是不可能的。这时CMake生成下面的代码：

... /path/to/libfoo.a ...

This guarantees that the correct library is chosen. However there are some caveats to keep in mind. In the past a project could write this (incorrect) code, and it would work by accident:

这确保选择正确的程序库。但是需要意识到，之前工程这样写有时也可以正确工作（尽管写法是错误的）：

add\_executable (myexe myexe.c)

target\_link\_libraries (myexe /path/to/libA.so B)

where "B" is meant to link "/path/to/libB.so". This code is incorrect because it asks CMake to link to B but does not provide the proper linker search path for it. It used to work by accident because the -L/path/to would get added as part of the implementation of linking to A. The correct code would be either

在这里"B"表示连接"/path/to/libB.so"。这段代码是错误的，因为它让CMake连接到B，但并没有提供搜索连接库的路径。当-L/path/to中包含连接路径时，它能正常工作。正确的写法是：

link\_directories (/path/to)

add\_executable (myexe myexe.c)

target\_link\_libraries (myexe /path/to/libA.so B)

下面的写法更好：

add\_executable (myexe myexe.c)

target\_link\_libraries (myexe /path/to/libA.so /path/to/libB.so)

Linking to System Libraries

连接系统程序库

System libraries on UNIX-like systems are typically provided in /usr/lib or /lib. These directories are considered implicit linker search paths because linkers automatically search these locations, even without a ﬂag like -L/usr/lib. Consider the code

在类UNIX系统中，系统程序库放置在/usr/lib或者/lib目录下。这些目录是连接器默认的搜索路径。即便不加入-L/usr/lib标识符。请看下面的代码

find\_library (M\_LIB m)

target\_link\_libraries (myexe ${M\_LIB})

Typically the find\_library command would ﬁnd the math library /usr/lib/libm.so, but some platforms provide multiple versions of libraries correesponding to different architectures. For example, on an IRIX machine one might ﬁnd the libraries

一般来讲find\_library命令将找到数学运算程序库/usr/lib/libm.so，但是有些平台会根据架构不同，提供多个版本的程序库。比如，IRIX可能找到程序库

/usr/lib/libm.so (ELF O32)

/usr/lib32/libm.so (ELF n32)

/usr/lib64/libm.so (ELF 64)

On a Solaris machine one might ﬁnd

在Solaris上可能找到

/usr/lib/libm.so (sparcv8 architecture)

/usr/lib/sparcv9/libm.so (sparcv9 architecture)

Unfortunately, find\_library may not know about all of the architecture-speciﬁc system search paths used by the linker. In fact, when it ﬁnds /usr/lib/libm.so, it may be ﬁnding a library with the incorrect architecture. If the link computation were to produce the line

不幸的是，find\_library命令可能不知道所有架构的系统的连接查找路径。事实上，它找到的/usr/lib/libm.so可能对应错误的架构。如果生成的连接命令是这样的

/usr/lib/libm.so

the linker might complain if /usr/lib/libm.so does not match the architecture it wants. One solution to this problem is for the link computation to recognize that the library is in a system directory and ask the linker to search for the library. It could produce the link line

连接器可能会报错/usr/lib/libm.so并不和所需的架构匹配。一个解决方案是指明连接的是系统程序库，并使连接器查找。它会生成对应的连接命令

... -lm ...

and the linker would search through its architecture-speciﬁc implicit link directories to ﬁnd the correct library. Unfortunately, this solution suffers from the original problem of distinguishing between static and shared versions. In order to ask the linker to ﬁnd a static system library with the correct architecture it must produce the link line

连接器会找到对应它架构的程序库。不幸的是，这种解决方案也要明确是静态程序库还是共享程序库。要想使连接器找到对应架构下正确的静态系统程序库，需要生成连接命令

... -Wl,-Bstatic -lm ... -Wl,-Bshared ...

Since not all platforms support such ﬂags CMake compromises. Libraries that are not in implicit system locations are linked by passing the full library path to the linker. Libraries that are in implicit system locations (such as /usr/lib) are linked by passing the -l option if a ﬂag like -Bstatic is available, and by passing the full library path to the linker otherwise.

因为有些平台不支持CMake生成的标志位。连接器使用绝对地址连接不在系统程序库目录下的程序库。在系统程序库目录下的程序库通过-l选项指定，如果支持-Bstatic标志位，可以传递给连接器绝对路径。

Specifying Optimized or Debug Libraries with a Target

指定目标生成优化的或者调试用程序库

On Windows platforms it is often required to link debug libraries with debug libraries, and optimized libraries with optimized libraries. CMake helps satisfy this requirement with the target\_link\_libraries command, which accepts an optional ﬂag that is debug or optimized. So, if a library is preceded with either debug or optimized, then that library will only be linked in with the like conﬁguration type. For example:

在Windows平台，一般需要生成对应调试或优化的程序库。使用target\_link\_libraries CMake命令可以满足这种需求，这需要传递可选标志位指明用于调试还是优化。如果一个构建标记为调试或者优化，那么它的程序库将按此类型连接。例如：

add\_executable (foo foo.c)

target\_link\_libraries (foo debug libdebug optimized libopt)

In this case foo will be linked against libdebug if a debug build was selected, or against libopt if an optimized build was selected.

本例中foo将被连接为libdebug（使用debug构建）或者libopt（使用optimized构建）。

4.9 Shared Libraries and Loadable Modules

4.9 共享程序库和可加载模块

Shared libraries and loadable modules are very powerful tools for software developers. They can be used to create extension modules or plugins for off-the-shelf software, and can be used to decrease the compile/link/run cycles for C and C++ programs. However, despite years of use, the cross platform creation of shared libraries and modules remains a black art understood by only a few developers. CMake has the ability to aid developers in the creation of shared libraries and modules. CMake knows the correct tools and ﬂags to use in order to produce the shared libraries for most modern operating systems that support them. Unfortunately, CMake cannot do all the work, and developers must sometimes alter source code and understand the basic concepts and common pitfalls associated with shared libraries before they can be used effectively. This section will describe many of the issues required to take advantage of shared libraries and loadable modules.

共享程序库和可加载模块是软件开发中非常强大的工具。它们被用来创建现有软件的扩展模块或者插件，而且可以减少C和C++程序的编译/连接/执行时间。但是，这么多年过去了，跨平台创建共享程序库和模块依然被看做是一门黑暗艺术，只被少数开发者掌握。CMake可以让开发者掌握创建共享程序库和模块的能力。CMake知道如何使用正确的工具和标志位为支持的大多数现代操作系统创建共享程序库。不幸的是，CMake并不能做所有的事，开发者有时必须修改源代码并且在有效使用共享程序库之前理解一些基本概念和已知的陷阱。本节将详细讨论如何使用共享程序库和可加载模块。

A shared library should be thought of more like an executable than a static library, and on most systems actually requires executable permissions to be set on the shared library ﬁle. This means that shared libraries can link to other shared libraries when they are created in the same way as an executable. Unlike a static library where the atomic unit is the object ﬁle, for shared libraries, the entire library is the atomic unit. This can cause some unexpected linker errors when converting from static to shared libraries. If an object ﬁle is part of a static library, but the executable linking to the library does not use any of the symbols in that object ﬁle, then the ﬁle is simply excluded from the ﬁnal linked executable. With shared libraries, all the object ﬁles that make up the library and all of the dependencies that they require come as one unit. For example, suppose you had a library with an object ﬁle deﬁning the function DisplayOnXWindow() which required the X11 library. If you linked an executable to that library, but did not call the DisplayOnXWindow() function, the static library version would not require X11, but the shared library version would require the X11 library. This is because a shared library has to be taken as one unit, and a static library is only an archive of object ﬁles from which linkers can choose which objects are needed. This means that static linked executables can be smaller, as they only contain the object code actually used.

一个共享程序库更像是一个可执行的而不是一个静态程序库，在大多数操作系统中，共享程序库也需要可执行权限。这表明当两个共享程序库以同样方式创建为一个可执行的时，共享程序库可以连接到其他共享程序库。不同于静态程序库，由目标文件（原子单元）组成，而共享程序库整个是一个原子单元。将静态程序库转为动态程序库可能引起一些未知的连接错误。如果一个静态程序库中的目标文件，以可执行形式连接到一个程序库而且没有使用这个目标文件中的任何标识符，那么这个目标文件会最终排除在可执行文件之外。使用共享程序库，所有的目标文件和依赖文件组成一个单元文件。例如，假设你有一个依赖X1程序库的程序库，当中有一个目标文件定义了一个方法DisplayOnXWindow()。如果你连接一个可执行的到这个共享程序库，但是没有调用DisplayOnXWindow()方法，静态程序库的版本将不需要X11，而动态程序库需要X11程序库。这是因为共享程序库被认为是一个单元，而静态程序库是包含很多目标文件的档案文件，连接器可以选择需要哪些目标文件。这表示静态连接的可执行文件可以更小，因为它们只包含需要的对象代码。

Another difference between shared and static libraries is library order. With static libraries the order on the link line can make a difference. This is because most linkers only use the symbols that are needed in a single pass over all the given libraries. So, the library order should go from the library that uses the most other libraries to the library that uses no other libraries. CMake will preserve and remember the order of libraries and library dependencies of a project. This means that each library in a project should use the target\_link\_libraries command to specify all of the libraries that it directly depends on. The libraries will be linked with each other for shared builds, but not static builds. However, the link information is used in static builds when executables are linked. An executable that only links library libA will get libA plus libB and libC as long as libA’s dependency on libB and libC was properly speciﬁed using target\_link\_libraries(libA libB libC).

共享程序库和静态程序库的另一个不同是程序库顺序。静态程序库的连接命令的顺序是有区别的。这是因为大多数连接器只在单通道中使用所给出的程序库的标识符。所以程序库顺序应该从依赖其它程序库最多的程序库开始到没有依赖的程序库结束。CMake将保存工程中程序库的顺序和依赖。也就是说每个工程中的程序库应该使用target\_link\_libraries命令指明它的直接依赖。对于共享构建，程序库会彼此连接。但是，连接信息在静态构建可执行文件时使用。一个可执行文件只连接程序库libA会将libA的依赖libB和libC使用target\_link\_libraries(libA libB libC)指明。

At this point, one might wonder why shared libraries would be preferred over static libraries. There are several reasons. First, shared libraries can decrease the compile/link/run cycle time. This is because the linker does not have to do as much work when linking to shared libraries because there are fewer decisions to be made about which object ﬁles to keep. Also, often times, the executable does not even need to be re-linked after the shared library is rebuilt. So, developers can work on a library compiling and linking only the small part of the program that is currently being developed, and then re—run the executable after each build of the shared library. Also, if a library is used by many different executables on a system, then there only needs to be one copy of the library on disk, and often in memory too.

这时，你可能会想为什么要优先考虑共享程序库。这里有几点原因。第一，共享程序库可以减少编译/连接/执行时间。这是因为连接器连接共享程序库时不需要做太多的工作，它不用考虑哪些目标文件需要保留。另外在共享程序库重新构建后，不需要重新连接。所以，开发人员可以关注于正在开发的一小部分程序的编译和连接。另外，如果一个程序库被系统中的不同程序使用，它在内存和磁盘中只需要一份拷贝。

In addition to the concept of a software library, shared libraries can also be used on many systems as run time loadable modules. This means that a program can at run time, load and execute object code that was not part of the original software. This allows developers to create software that is both open and closed. (For more information see Object Oriented Software Construction by Bertrand Meyer.) Closed software is software that cannot be modiﬁed. It has been through a testing cycle and can be certiﬁed to perform speciﬁc tasks with regression tests. However, a seemingly opposite goal is sought after by developers of object oriented software. This is the concept of Open software that can be extended by future developers. This can be done via inheritance and polymorphism with object systems. Shared libraries that can be loaded at run time, allow for these seemingly opposing goals to be achieved in the same software package. Many common applications support the idea of plugins. The most common of these applications is the web browser. Internet Explorer uses plugins to support video over the web and 3D visualization. In addition to plugins, loadable factories can be used to replace C++ objects at run time, as is done in VTK.

除了软件程序库的概念，在许多系统中，共享程序库还可以被用作运行时可加载模块。这意味着一个程序可以在运行时加载并运行目标代码，这些目标代码不在原始程序中。这就允许开发者开发出既开放又封闭的软件。（更多的信息请看Bertrand Meyer的面向对象软件构造Object Oriented Software Construction。）封闭是指软件不能被修改。通过测试后它可以执行某些工作。但是，显然这个面向对象软件开发者的目标相违背。开放软件的概念是它可以在后续开发中扩展。这可以通过对对象系统继承和使用多态完成。共享程序库可以在运行时加载。许多程序都支持这一种插件的思想。其中最常用的是网页浏览器。Internet Explorer使用插件支持网页视频回放和3D可视化。除了插件，可加载模块也被用在运行时替换C++目标文件，这在VTK中已经实现。

Once it is decided that shared libraries or loadable modules are the right choice for a particular project, there are a few issues that developers need to be aware of. The ﬁrst question that must be answered is which symbols are exported by the shared library? This may sound like a simple question, but the answer is different from platform to platform. On many, but not all UNIX systems, the default behavior is to export all the symbols much like a static library. However, on Windows systems, developers must explicitly tell the linker and compiler which symbols are to be exported and imported from shared libraries. This is often a big problem for UNIX developers moving to Windows. There are two ways to tell the compiler/linker which symbols to export/import on Windows. The most common approach is to decorate the code with a MicrosoftTM C/C++ language extension. An alternative is to create an extra ﬁle called a .def ﬁle. This ﬁle is a simple ASCII ﬁle containing the names of all the symbols to be exported from a library.

一旦决定在某个工程中使用共享程序库或者可加载模块，有几点问题需要开发者注意。首先需要明确共享程序库暴露哪些接口，这听起来很简单，但是平台不同答案也不尽相同。在很多系统上，但不局限于UNIX系统，默认会像静态程序库那样暴露所有接口。但是在Windows系统上，开发者必须明确告诉连接器和编译器暴露哪些接口。这导致UNIX开发者转为Windows开发者时，遇到很大问题。Windows中，有两种方式告诉编译器/连接器暴露哪些接口。最常用的方式使用MicrosoftTM C/C++语言扩展修饰代码。另一种是创建一个.def文件。该文件是一个ASCII文件，包含导出接口的名字。

The MicrosoftTM extension uses the \_declspec directive. If a symbol has \_declspec(dllexport)in front of it,it will be exported, and if it has \_declspec(dllimport) it will be imported. Since the same ﬁle may be shared during the creation and use of a library, it must be both exported and imported in the same source ﬁle. This can only be done with the preprocessor. The developer can create a macro called LIBRARY\_EXPORT that is deﬁned to dllexport when building the library and dllimport when using the library. CMake helps this process by automatically deﬁning ${LIBNAME}\_EXPORTS when building a DLL (dynamic link library, aka. a shared library) on Windows.

MicrosoftTM扩展使用\_declspec指令。如果一个标识符前面有\_declspec(dllexport)，它将被导出，如果是\_declspec(dllimport)，将被导入。因为一个文件即可能是共享的，又使用其它程序库，在源码文件中它必须是导出和导出的。这会被预处理器执行。开发者能创建宏LIBRARY\_EXPORT，定义dllexport，也可以定义dllimport用于构建。当在Windows中构建dll(动态链接库，既共享程序库)时，CMake帮助自动定义${LIBNAME}\_EXPORTS。

The following code snippet is from the VTK library vtkCommon, and is included by all ﬁles in the vtkCommon library:

下面是VTK程序库vtkCommon的代码片段，它包含在所有vtkCommon程序库中：

#if defined(WIN32)

#if defined(vtkCommon\_EXPORTS)

#define VTK\_COMMON\_EXPORT \_declspec(dllexport)

#else

#define VTK\_COMMON\_EXPORT \_declspec(dllimport)

#endif

#else

#define VTK\_COMMON#EXPORT

#endif

The example checks for Windows and checks the vtkCommon\_EXPORTS macro provided by CMake. So, on UNIX VTK\_COMMON\_EXPORT is deﬁned to nothing, and on Windows during the building of vtkCommon.dll it is deﬁned as \_declspec(dllexport) , and when the ﬁle is being used by another ﬁle, it is deﬁned to \_declspec(dllimport).

这个例子检测Windows并且检测Cmake是否提供vtkCommon\_EXPORTS宏。所以，UNIX系统上VTK\_COMMON\_EXPORT定义为nothing，而在Windows上构建vtkCommon.dll时，定义为\_declspec(dllexport)，当被另外的文件使用时，定义为\_declspec(dllimport)。

The second approach requires a .def ﬁle to specify the symbols to be exported. This ﬁle could be created by hand, but for a large and changing C++ library that could be time consuming and error prone. CMake’s custom commands can be used to run a pre-link program that will create a .def ﬁle from the compiled object ﬁles automatically. In the following example, a Perl script called makedef.pl is used, the script runs the DUMPBIN program on the .obj ﬁles and extracts all of the exportable symbols and writes a .def ﬁle with the correct exports for all the symbols in the library mylib.

第二种方法需要一个.def文件指明需要导出的标识符。这个文件可以手动创建，但是对于大的，迭代中的C++程序库，手动创建消耗时间并且容易出错。CMake的定制命令可以自动运行预连接的程序从编译后的目标文件生成.def文件。在下面的例子中，一个叫makedef.pl的Perl脚本对.obj文件执行DUMPBIN程序，并抽取所有可导出标志位，为程序库mylib写入.def文件。

----CMakeLists.txt-----

cmake\_minimum\_required (VERSION 2.6)

project (myexe)

set (SOURCES mylib.cxx mylib2.cxx)

# create a list of all the object files

string (REGEX REPLACE "\\.CXX" ".obj" OBJECTS "${SOURCES}“)

# create a shared library with the .def file

add\_library (mylib SHARED ${SOURCES}

${CMAKEiCURRENTWBINARY\_DIR}/mylib.def

)

# set the .def file as generated

set\_sourcegfiles\_properties (

${CMAKE\_CURRENT\_BINARY\_DIR}/mylib.def

PROPERTIES GENERATED 1

)

# create an executable

add\_executable (myexe myexe.cxx)

# link the executable to the dll

target\_link\_libraries (myexe mylib)

#convert to windows slashes

set (OUTDIR

${CMAKE\_CURRENT\_BINARY\_DIR}/${CMAKE\_CFG\_INTDIR}

)

string (REGEX REPLACE "/" "\\\\" OUTDIR ${OUTDIR})

# create a custom pre link command that runs

# a perl script to create a .def file using dumpbin

add\_custom\_command (

TARGET mylib PRE\_LINK

COMMAND perl

ARGS ${CMAKE#CURRENT750URCE7DIR}/makedef.pl

${CMAKE\_CURRENT\_BINARYADIR}\\mylib.def mylib

${OUTDIR} ${OBJECTS}

COMMENT "Create .def file"

)

---myexe.cxx----

#include <iostream>

#include "mylib.h"

int main()

{

std::cout << myTen() << "\n";

std::cout << myEight() << "\n";

---mylib.cxx--

int myTen()

{

return 10;

}

--mylib2.cxx---

int myEight()

{

return 8;

}

There is a signiﬁcant difference between Windows and most UNIX systems with respect to the requirements of symbols. DLLs on Windows are required to be fully resolved, this means that they must link every symbol at creation. UNIX systems allow shared libraries to get symbols from the executable or other shared libraries at run time. On UNIX systems that support this feature, CMake will compile with the ﬂags that allow executable symbols to be used by shared libraries. This small difference can cause large problems. A common, but hard to track down bug with DLLs happens with C++ template classes and static members. Two DLLs can end up with separate copies of what is supposed to be a single global static member of a class. There are also problems with the approach taken on most UNIX systems. The start up time for large applications with many symbols can be long since much of the linking is deferred to run time.

对于所需的标识符，Windows系统和大多数UNIX系统有显著的不同。Windows的DLLs需要在创建时连接所有标识符。UNIX系统语序共享程序库在运行时从可执行文件或者其他共享程序库中获得标志位。在支持这个特性的系统中，CMake将使用标志位编译可以执行文件的标识符可以被共享程序库使用。这个小小的区别导致很大的问题。当使用C++模板类和静态成员时DLLs会产生普遍并且难于追踪的问题。两个DLLs可能有一个全局静态成员的两份拷贝。大多数UNIX系统也有这个问题。大型软件使用过多标识符因为需要连接，启动也很耗时。

Another common pitfall occurs with C++ global objects. These objects require that constructors must be called before they can be used. The main that links or loads C++ shared libraries MUST be linked with the C++ compiler, or globals like cout may not be initialized before they are used, causing strange crashes at start up time.

另一个常见陷阱发生在C++全局对象上。这些对象需要在调用之前使用构造器构造。main函数中的连接和加载C++共享库必须被C++编译器连接，否则全局对象如cout在使用时可能没有初始化，这导致开始运行时异常崩溃。

Since executables that link to shared libraries must be able to ﬁnd the libraries at run time, special environment variables and linker ﬂags must be used. There are tools that can be used to show which libraries an executable is actually using. On many UNIX systems there is a tool called ldd (otool -L on Mac OS X) that shows which libraries are used by an executable. On Windows, a program called depends can be used to ﬁnd the same type of information. On many UNIX systems there are also environment variables like LD\_LIBRARY\_PATH that tell the program where to ﬁnd the libraries at run time. Where supported CMake will add run time library path information into the linked executables, so that LDﬁLIBRARY\_PATH is not required. This feature can be turned off by setting the cache entry CMAKE\_SKIP\_RPATH to false. This may be desirable for installed software that should not be looking in the build tree for shared libraries. On Windows there is only one PATH environment variable that is used for both DLLs and ﬁnding executables.

因为连接共享程序库的可执行文件必须在运行时找到程序库，必须使用指定环境变量和连接标志位。有工具能够显示可执行文件使用什么程序库。在许多UNIX系统下，工具ldd(Mac OS X下是otool -L)做这项工作。Windows下工具depends做这项工作。在许多UNIX系统下环境变量如LD\_LIBRARY\_PATH告诉程序在运行时去哪里寻找程序库。

4.10 Shared Library Versioning

4.10 共享程序库版本

When an executable is linked to a shared library, it is important that the copy of the shared library loaded at runtime matches that expected by the executable. On some UNIX systems, a shared library has an associated "soname" intended to solve this problem. When an executable links against the library, its soname is copied into the executable. At runtime, the dynamic linker uses this name from the executable to search for the library.

当一个可执行程序连接到一个共享程序库，在运行时加载匹配的程序库是很重要的。在一些UNIX系统中，共享程序库有一个soname以解决这个问题。当一个可执行文件连接程序库时，它的soname被写入可执行文件中。在运行时，动态连接器使用这个name查找程序库。

Consider a hypothetical shared library "foo" providing a few C functions that implement some functionality. The interface to foo is called an Application Programming Interface (API). If the implementation of these C functions changes in a new version of foo, but the API remains the same, then executables linked against foo will still run correctly. When the API changes, old executables will no longer run with a new copy of foo, so a new API version number must be associated with foo.

假设一个共享程序库"foo"提供了一些C函数。foo的接口叫做Application Programming Interface (API)。如果这些C函数的实现在foo的新版本中改变了，但是API没有改变，连接到foo的可执行文件还可以正确运行。当API改变，旧的可执行文件使用新版本的foo时将无法运行，所以API版本号是必须的。

This can be implemented by creating the original version of foo with a soname and ﬁle name such as libfoo.so.1. A symbolic link such as libfoo.so -> libfoo.so.1 will allow standard linkers to work with the library and create executables. The new version of foo can be called libfoo.so.2 and the symbolic link updated so that new executables use the new library. When an old executable runs, the dynamic linker will look for libfoo.so.1, ﬁnd the old copy of the library, and run correctly. When a new executable runs, the dynamic linker will look for libfoo.so.2 and correctly load the new version.

可以使用soname和文件名如(libfoo.so.1)组成程序库的文件名。软链接如libfoo.so -> libfoo.so.1可以使标准的连接器连接程序库和可执行文件。foo的新版本可以是libfoo.so.2，更新软链接可以使新的可执行文件使用新的程序库。当老的可执行文件运行时，动态连接器查找libfoo.so.1，发现老版本的程序库，并正确运行。当新的可执行文件运行时，动态连接器查找libfoo.so.2并加载正确的版本。

This scheme can be expanded to handle the case of changes to foo that do not modify the API. We introduce a second set of version numbers that is totally independent of the ﬁrst. This new set corresponds to the software version providing foo. For example, some larger project may have introduced the existence of library foo starting in version 3.4. In this case, the ﬁle name for foo might be libfoo.so.3.4, but the soname would still be libfoo.so.1 because the API for foo is still on its ﬁrst version. A symbolic link from libfoo.so.1 -> libfoo.so.3.4 will allow executables linked against the library to run. When a bug is ﬁxed in the software without changing the API to foo, then the new library ﬁle name might be libfoo.so.3.5, and the symbolic link can be updated to allow existing executables to run.

这个设计可以扩展到处理程序库改变API不变的情况。我们将介绍本号命名。