# 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）这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