链滴

[翻译] CMake 和 Make 之间的区别

作者：lixiang0

原文链接：https://ld246.com/article/1469082435014

来源网站：链滴

本文翻译的是一篇英文文档，主要讲述的是 CMake 和 Make 之间的区别。下文中首先列出文章中文翻译，然后紧接着的是英文原文。
<hr>
下面是中文翻译部分：
编程人员已经使用 CMake 和 Make 很长一段时间了。当你加入一家大公司或者开始在一个具有量代码的工程上开展工作时，你需要注意所有的构建。你需要看到处跳转的"CMakeLists.txt"文件。应该会在终端使用"cmake"和"make"。很多人都是盲目的跟着操作说明而并不在意我们已何种方式做我们需要做的事。构建的整个过程是什么？为什么要用这种方式去组织？Cmake 和 Make 之间有么区别？这有关系吗？他们可以互相转换吗？
事实证明，它们之间有很多的不同。理解它们之间的不同很重要，这样你可以确保在使用的过程不会陷入麻烦。在讲述它们之间的区别之前，我们首先来看看它们是什么。
Make
要设计一个软件系统，我们首先编写源码，然后通过编译器编译和创建可执行文件。可执行文件是要实现最终功能的文件。“Make”是一个工具，它控制可执行程序和程序源文件中非源码文件的成。
“Make”工具需要清楚的知道如何构建程序。 它通过一个叫做“makefile”的文件知晓如何构你的程序。这个文件列出了所有的非源码文件以及如何由别的文件来计算它。当你编写了一个程序，应该为它写一个 makefile 文件，这样才有可能通过使用“Make”来构建和安装你的程序。 很简单事情。如果你不理解的话，多读几遍这一段文字，因为理解这一段文字对于接下来的篇幅很重要。<b>
为什么需要“Make”
需要”make”的一个原因是，它可以使得终端用户构建和安装你的应用包，而不用去详细的了解具体是如何做到的。每一个工程都有它自己的规则和细微的差别，这会使得每次在复用的时候会变得痛苦。这就是我们创建这个 makefile 文件的原因。 构建步骤精确的记录在你提供的这个 makefile 件中。 “Make” 当源码文件发生变化时自动的指出哪一个文件需要更新。 同时它也自动确定以适当顺序进行更新文件，当一个非源码文件依赖的另一个非源码文件发生改变时。
每次当我们改变了系统中的一小部分源码的时候，重新编译整个程序的效率是很低的。因此，觉们改变了一小部分的源码文件的时候重新执行“Make”，它将不会重新编译整个程序。它仅仅更新些直接或者间接依赖这些改变了的源码文件的非源码文件。很酷吧！ “Make” 不局限于具体的语言对于程序中的每一个非源码文件，makefile 文件详细的说明了执行需要的 shell 命令。这些 shell 命能够启动编译器产生目标文件，链接器产生可执行文件，ar 更新库，镜像生成器格式化文档，等等。。Make 不仅仅局限于构建一个包。你也可以安装或者卸载一个包，生成索引表或者其他一些你经常的值得你写下来怎么去做的事情。
CMake
CMake 支持跨平台 Make。 CMake 辨别使用那种编译器去编译给出的源码种类。如果你不知道用何种编译器，你不能使用相同的编译器去编译所有不同种类的源码。你可以手动的指用何种编译器但是这将变得繁琐和痛苦。CMake 为每一种类型的目标平台按照正确的顺序调用命令。因此，将有多非显式的命令，比如 $(CC).
如果你是代码强迫症，请继续往下读。如果你不喜欢这一切，你可以跳过这一部分。一般的编译/接标识处理头文件、库文件、以及重定位其他平台无关和构建系统独立命令。调试标识被包含，通过置变量 CMAKE_BUILD_TYPE 为“debug”，或者在调用程序的使用传递给 CMake：cmake -DCMKE——BUILD——TYPE：STRING=Debug。
CMake 也提供平台无关的包含，通过‘-fPIC’标志（POSITION_INDEPENDENT_CODE 属性）因此，更多隐式的设置能够在 CMake 命令中实现，在 makefile 文件中也可以（通过使用 COMPILE_LAGS 或者相关的属性）。当然，CMake 在集成第三方库（像 OpenGL）方面也变得更加轻便。

它们之间有什么不同点？
如果你要使用编译脚本，构建的过程中有一个步骤，也就是需要在命令行中输入”make”。 对于Make，需要进行 2 步：第一，你需要配置你的编译环境（可以通过在你的编译目录中输入 cmake &lt source_dir>，也可以通过使用 GUI 客户端）。这将创建一个等效的，依赖你选择的编译环境的译脚本或其他。编译系统可以传递给 CMake 一个参数。总之，CMake 根据你的系统配置选择合理的默认的选择。第二，你在你选择的编译系统中执行实际的构建。
你将进入 GNU 构建系统领域。 如果你不熟悉这个，这一段对于你来说就是碎碎念了。现在，说了一些严肃的警告，往下走把！ 我们可以使用 Autotool 来比较 CMake。当我们这样做的时候，我

会发现 Make 的缺点，而且这就是 Autotool 产生的理由了。我们可以看到 CMake 明显比 Make 优的理由了。Autoconf 解决了一个重要的问题，也就是说与系统有关的构建和运行时信息的的可信赖现。但是，这仅仅是轻便软件的开发中的一小部分。作为结尾，GNU 工程已经开发了一系列集成的用程序，用于完成 Autoconf 开始之后的工作：GNU 构建系统中最重要的组件是 Autoconf, Automae, and Libtool.
□□□□“Make”就不能那样做了，至少不修改任何东西是做不到的。你可以自己做所有的跨平台工作，是这将花费很多时间。CMake 解决了这个问题，但是与此同时，它比 GNU 构建系统更有优势：</p

用于编写 CMakeLists.txt 文件的语言具有可读性和很容易理解。
不仅可以使用“Make” 来构建工程。
支持多种生产工具，比如 Xcode, Eclipse, Visual Studio, etc.
□□□□CMake 与 Make 对比具有以下优点：
自动发现跨平台系统库。
自动发现和管理的工具集
更容易将文件编译进共享库， 以一种平台无关的方式或者以比 make 更容易使用的的生成方式。br>
□□□□CMake 不仅仅只“make”，所以它变得更复杂。从长远来看，最好能够学会使用它。如果你仅在一个平台上构建小的工程，“Make”更适合完成这部分工作。

<hr>
<p>下面是本文的英文原文部分：
CMake vs Make
□□□□Programmers have been using CMake and Make for a long time now. When you join a big company or start working on a project with a large codebase, there are all these builds that yu need to take care of.You must have seen those “CMakeLists.txt” files floating around. You are supposed to run “cmake” and “make” commands on the terminal. A lot of people just follow the instructions blindly, not really caring about why we need to do things in a certain ay. What is this whole build process and why is it structured this way? What are the difference between CMake and Make? Does it matter? Are they interchangeable?
□□□□As it turns out, they are quite different. It is important to understand the differences betwen them to make sure you don’t get yourself in trouble. Before getting into the differences, le’s first see what they are.
Make
□□□□The way in which we design a software system is that we first write code, then the compiler compiles it and creates executable files. These executable files are the ones that carry out the ctual task. “Make” is a tool that controls the generation of executables and other non-soure files of a program from the program’s source files.
□□□□The “Make” tool needs to know how to build your program.It gets its knowledge of how to build your program from a file called the “makefile”. This makefile lists each of the non-source files and how to compute it from other files. When you write a program, you should writ a makefile for it, so that it is possible to use “Make” to build and install the program.Simple stuff! If you didn’t understand it, go back and read the paragraph again because it’s important for the next part.
Why do we need “Make”?
□□□□The reason we need “Make” is because it enables the end user to build and install your ackage without knowing the details of how it’s done. Every project comes with its own rules and nuances, and it can get quite painful every time you have a new collaborator. That’s the eason we have this makefile.The details of the build process are actually recorded in the make ile that you supply. “Make” figures out automatically which files it needs to update, based n which source files have changed.It also automatically determines the proper order for updatng the files, in case one non-source file depends on another non-source file.
□□□□Recompiling the entire program every time we change a small part of the system would be inefficient.Hence, if you change a few source files and then run “Make”, it doesn’t recompi

e the whole thing. It updates only those non-source files that depend directly or indirectly on he source files that you changed.Pretty neat! “Make” is not limited to any particular langua e. For each non-source file in the program, the makefile specifies the shell commands to com ute it. These shell commands can run a compiler to produce an object file, the linker to produ e an executable, ar to update a library, Makeinfo to format documentation, etc. “Make” is no limited to just building a package either. You can also use “Make” to control installing or u installing a package, generate tags tables for it, or anything else you want to do often enough to make it worth while writing down how to do it.
CMake
CMake stands for Cross-platform Make. CMake recognizes which compilers to use for a gi en kind of source.In case you didn’t know, you can’t use the same compiler to build all the different kinds of sources. You can do this manually every time you want to build your project, but it would be tedious and painful. CMake invokes the right sequence of commands for each type of target. Therefore, there is no explicit specification of commands like $(CC).
For coding junkies who really want the gory details, read on. If you are not into all that, you can skip to the next section. All the usual compiler/linker flags dealing with the inclusion of h ader files, libraries, etc are replaced by platform independent and build system independent ommands.Debugging flags are included by either setting the variable CMAKE_BUILD_TYPE to “Debug”, or by passing it to CMake when invoking the program:cmake -DCMAKE_BUILD_T PE:STRING=Debug.
CMake also offers the platform independent inclusion of the ‘-fPIC’ flag (via the POSITI N_INDEPENDENT_CODE property) and many others. Still, more obscure settings can be impl mented by hand in CMake just as well as in a Makefile (by using COMPILE_FLAGS and similar roperties). Of course CMake really starts to shine when third party libraries (like OpenGL) are i cluded in a portable manner.
What is the difference?
The build process has one step if you use a Makefile, namely typing “make” at the com and line.For CMake, there are two steps: First, you need to setup your build environment (eith r by typing cmake <source_dir> in your build directory or by running some GUI client).Th s creates a makefile or something equivalent, depending on the build system of your choice (e g. Make on *nix, VC++ or MinGW on Windows, etc). The build system can be passed to CMak as a parameter. However, CMake makes reasonable default choices depending on your syst m configuration. Second, you perform the actual build in the selected build system.
We are going to jump into the GNU build system territory here.If you are not familiar with hat, this paragraph might look like jibber-jabber to you. Alright, now that I have given the sta utory warning, let’s move on!We can compare CMake with Autotools. When we do that, we an see the shortcomings of Make, and they form the reason for the creation of Autotools. We can also see the obvious advantages of CMake over Make. Autoconf solves an important prob em i.e. reliable discovery of system-specific build and runtime information. But this is only a mall part in the development of portable software. To this end, the GNU project has develop d a suite of integrated utilities to finish the job Autoconf started: the GNU build system, whos most important components are Autoconf, Automake, and Libtool.
“Make” can’t do that, at least not without modifying it anyway!You can make it do all t at stuff but it would take a lot of time maintaining it across platforms. CMake solves the same problem, but at the same time, it has a few advantages over the GNU Build System:

The language used to write CMakeLists.txt files is readable and easier to understand.
It doesn’t only rely on “Make” to build the project.
It supports multiple generators like Xcode, Eclipse, Visual Studio, etc.

When comparing CMake with Make, there are several advantages of using Cmake:</
>

Cross platform discovery of system libraries.

Easier to compile your files into a shared library in a platform agnostic way, and in general easier to use than make.
Automatic discovery and configuration of the toolchain.

□□□□CMake does more than just “make”, so it can be more complex. In the long run, it’ better to learn how to use it. If you have just a small project on only one platform, then may e “Make” can do a better job.