01 将单个文件编译为可执行文件

将单个源文件编译为可执行文件

准备工作 我们希望将以下源代码编译为单个可执行文件：

```
1. #include
2. #include
3. #include
4.
5. std::string say_hello() { return std::string("Hello, CMake
   world!"); } 6.
6. int main() {
7. std::cout << say_hello() << std::endl;
8. return EXIT_SUCCESS;
9. }
```

具体实施
我们把CMake指令放入一个名为 CMakeLists.txt 的文件中。
NOTE:文件的名称区分大小写，必须命名为 CMakeLists.txt ，CMake才能够解析。
具体步骤如下：

1. 用编辑器打开一个文本文件，将这个文件命名为 CMakeLists.txt 。
2. 第一行，设置CMake所需的最低版本。如果使用的CMake版本低于该版本，则会发出致命错误：
   `cmake_minimum_required(VERSION 3.5 FATAL_ERROR)`
3. 第二行，声明了项目的名称(recipe-01)和支持的编程语言(CXX代表C++)：
   `project(recipe-01 LANGUAGES CXX)`
4. 指示CMake创建一个新目标:可执行文件 hello-world 。这个可执行文件是通过编译和链接源 文件 hello-world.cpp 生成的。CMake将为编译器使用默认设置，并

自动选择生成工具：
add_executable(hello-world hello-world.cpp)

5. 将该文件与源文件 hello-world.cpp 放在相同的目录中。记住，它只能被命名 为 CMakeLists.txt 。
6. 现在，可以通过创建 build 目录，在 build 目录下来配置项目：

i. $ mkdir -p build
ii. $ cd build
iii. $ cmake ..
iv.
v. -- The CXX compiler identification is GNU 8.1.0
vi. -- Check for working CXX compiler: /usr/bin/c++
vii. -- Check for working CXX compiler: /usr/bin/c++ -- works
viii. -- Detecting CXX compiler ABI info
ix. -- Detecting CXX compiler ABI info - done
x. -- Detecting CXX compile features
xi. -- Detecting CXX compile features - done
xii. -- Configuring done
xiii. -- Generating done
-- Build files have been written to: /home/user/cmake-cookbook/chapter-
xiv. 01/recipe-01/cxx-example/build

7. 如果一切顺利，项目的配置已经在 build 目录中生成。我们现在可以编译可执行文件：

i. $ cmake --build . 2.
ii. Scanning dependencies of target hello-world
iii. [50%] Building CXX object CMakeFiles/hello-world.dir/hello-world.cpp.o
iv. [100%] Linking CXX executable hello-world
v. [100%] Built target hello-world

工作原理
示例中，我们使用了一个简单的 CMakeLists.txt 来构建“Hello world”可执行文件：

1. cmake_minimum_required(VERSION 3.5 FATAL_ERROR) 2. project(recipe-01 LANGUAGES CXX)
2. add_executable(hello-world hello-world.cpp)
NOTE:CMake语言不区分大小写，但是参数区分大小写。 TIPS:CMake中，C++是默认的编程语言。不过，我们还是建议使用 LANGUAGES 选项

在 project 命令中显式地声明项目的语言。
要配置项目并生成构建器，我们必须通过命令行界面(CLI)运行CMake。CMake CLI提供了许多选 项， cmake -help 将输出以显示列出所有可用选项的完整帮助信息，我们将在书中对这些选项进行 更多地了解。正如您将从 cmake -help 的输出中显示的内容，它们中的大多数选项会让你您访问 CMake手册，查看详细信息。通过下列命令生成构建器：

```
1. $ mkdir -p build
2. $ cd build
3. $ cmake ..
```

这里，我们创建了一个目录 build (生成构建器的位置)，进入 build 目录，并通过指定 CMakeLists.txt 的位置(本例中位于父目录中)来调用CMake。可以使用以下命令行来实现相同的 效果：

```
1. $ cmake -H. -Bbuild
```

该命令是跨平台的，使用了 -H 和 -B 为CLI选项。 -H 表示当前目录中搜索根 CMakeLists.txt 文件。 -Bbuild 告诉CMake在一个名为 build 的目录中生成所有的文件。
NOTE: cmake -H. -Bbuild 也属于CMake标准使用方式：https://cmake.org/pipermail/cmake-developers/2018-January/030520.html 。
不过，我们将在本书中使用传统方法(创建一个构建目录，进入其中，并通过将CMake指向 CMakeLists.txt 的位置来配置项目)。
运行 cmake 命令会输出一系列状态消息，显示配置信息：

```
1. $ cmake .. 2.
3. -- The CXX compiler identification is GNU 8.1.0
4. -- Check for working CXX compiler: /usr/bin/c++
5. -- Check for working CXX compiler: /usr/bin/c++ -- works
6. -- Detecting CXX compiler ABI info
7. -- Detecting CXX compiler ABI info - done
8. -- Detecting CXX compile features
9. -- Detecting CXX compile features - done
10. -- Configuring done
11. -- Generating done
-- Build files have been written to: /home/user/cmake-cookbook/chapter-
12. 01/recipe-01/cxx-example/build
```

NOTE:在与 CMakeLists.txt 相同的目录中执行 cmake . ，原则上足以配置一个项目。然而， CMake会将所有生成的文件写到项目的根目录中。这将是一个源代码内构建，通常是不推荐的，因为这 会混合源代码和项目的目录树。我们首选的是源外构建。
CMake是一个构建系统生成器。将描述构建系统(如:Unix Makefile、Ninja、Visual Studio 等)应当如何操作才能编译代码。然后，CMake为所选的构建系统生成相应的指令。默认情况下，在 GNU/Linux和macOS系统上，CMake使用Unix Makefile生成器。Windows上，Visual Studio是 默认的生成器。在下一个示例中，我们将进一步研究生成器，并在第13章中重新讨论生成器。
GNU/Linux上，CMake默认生成Unix Makefile来构建项目：
Makefile : make 将运行指令来构建项目。

CMakefile ：包含临时文件的目录，CMake用于检测操作系统、编译器等。此外，根据所选的生 成器，它还包含特定的文件。
cmake_install.cmake ：处理安装规则的CMake脚本，在项目安装时使用。
CMakeCache.txt ：如文件名所示，CMake缓存。CMake在重新运行配置时使用这个文件。
要构建示例项目，我们运行以下命令：

```
$ cmake --build .
```

最后，CMake不强制指定构建目录执行名称或位置，我们完全可以把它放在项目路径之外。这样做同样 有效：

```
$ mkdir -p /tmp/someplace
$ cd /tmp/someplace
$ cmake /path/to/source
$ cmake --build .
```