# CMake 简单教程

## CLion与CMakefile

在CLion中创建项目之后，我们会得到一个CMakelist.txt。在编译过程中，CLion会根据这个文件生成Makefile，同时根据里面的target提供相应选项。

## CMakeList.txt的编写方法

这里简单介绍一下CMakelist中常用的几个指令的作用和用法，大家也可以参考<https://www.hahack.com/codes/cmake/> 上的教程。这里仅仅介绍一些基本命令的用法，如需深入了解，请自行查阅官方文档。

1. **cmake\_minimum\_required**

设置一个工程所需要的最低CMake版本,其语法为：

|  |
| --- |
| cmake\_minimum\_required(VERSION major[.minor[.patch[.tweak]]] |

如果CMake的当前版本低于指定的版本，它会停止处理工程文件，并报告错误。

例如，如果我希望指定自己的工程需要的CMake最低版本为3.10，那么可以这么写：

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.10) |

1. **project**

为整个工程设置工程名其语法为：

|  |
| --- |
| project(<projectname> [languageName1 languageName2 ... ] ) |

为本工程设置一个工程名。而且，该命令还将变量<projectName>\_BINARY\_DIR和<projectName>\_SOURCE\_DIR设置为对应值。后面的可选项允许你指定工程可以支持的语言。例如，希望将自己的工程名设置为Hello，并且指定我的工程支持C 和C++，则可以这么写：。

|  |
| --- |
| project(Hello C C++) |

1. **add\_executable**

使用给定的源文件，为工程引入一个可执行文件。其语法为：

|  |
| --- |
| add\_executable(<name> [WIN32] [MACOSX\_BUNDLE]  [EXCLUDE\_FROM\_ALL]  source1 source2 ... sourceN) |

此命令的作用是，告诉CMake使用文件source1、source2、… sourceN来生成可执行文件。这里的WIN32、MACOS\_BUNDLE以及EXCLUDE\_FROM\_ALL选项在本课程中不会用到，所以不做解释。

例如，我希望用main.c 以及 math.c 生成main.exe。那么可以这么写：

|  |
| --- |
| add\_executable(main main.c math.c) |

1. target\_compile\_options

只需要掌握下面这个用法即可。下面的例子对main这一target，设置其C编译参数为后面那一串东西。如果我们设置C++语言的编译参数，将其中的C改为CXX就可以了。大家没必要记住这串命令，用的时候直接在例子的基础上修改即可。

但是要注意的时，target\_compile\_options 修改的是编译命令，不影响链接命令。要修改链接参数的话需要使用set\_target\_properties。

|  |
| --- |
| target\_compile\_options(main  PRIVATE $<$<COMPILE\_LANGUAGE:C>:-m32 -masm=intel -mno-red-zone -ffreestanding -fno-stack-protector -fno-exceptions -fno-rtti>  ) |

1. set\_target\_properties

此命令可以用于修改链接参数。下面这个例子对main这一target，设置其链接参数为“-nostdlib -m32 -ffreestanding “。

|  |
| --- |
| set\_target\_properties(kernel PROPERTIES  LINK\_FLAGS "-nostdlib -m32 -ffreestanding ") |

1. enable\_language

使CMake支持某种语言（CXX/C/Fortran/等）。语法为：

|  |
| --- |
| enable\_language(languageName [OPTIONAL] ) |

该命令打开了CMake对参数中指定的语言的支持。这与project命令相同，但是不会创建任何project命令会产生的额外变量。默认情况下，CMake仅支持C和C++，本课程中需要增加对NASM的支持，为了实现对NASM的支持，我们可以在CMakelist中添加：

|  |
| --- |
| enable\_language(ASM\_NASM ) |

1. set

用来显式的定义变量并给变量赋值。语法如下：

|  |
| --- |
| set (VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]]) |

此命令，设置了一个名为VAR的变量，其内容为VALUE。如果后面要使用这个变量，我们将使用${VAR}的形式来实现。

例如，我希望定义一个变量source，并在add\_executable中使用在则可以这么写：

|  |
| --- |
| set (source “main.c math.c”)  add\_executable(main ${source}) |

这等价于：

|  |
| --- |
| add\_executable(main main.c math.c) |

CMAKE中有一些自己用的变量，我们可以通过设置这些变量来达到一些目的。

|  |  |
| --- | --- |
| 变量名 | 作用 |
| CMAKE\_ASM\_NASM\_OBJECT\_FORMAT | 设置nasm编译后得到文件格式。 |
| CMAKE\_ASM\_NASM\_COMPILE\_OBJECT | 设置nasm编译asm文件使用的命令。 |
| CMAKE\_ASM\_NASM\_LINK\_EXECUTABLE | 设置nasm链接时执行的命令。如果我们仅仅想得到bin文件，我们只需要编译而不需要链接，因此我们可以通过将该变量设置为“cp <OBJECTS> <TARGET>”来忽略链接过程。 |
| CMAKE\_C\_FLAGS | 设置编译C文件时的选项。 |
| PROJECT\_SOURCE\_DIR | 此CMakeList.txt文件所在的目录。不要修改。 |

1. add\_custom\_command

在本课程中add\_custom\_command命令主要是是为了生成输出文件。这里生成的文件一般是由命令行命令产生的。

|  |
| --- |
| add\_custom\_command(OUTPUT output1 [output2 ...]  COMMAND command1 [ARGS] [args1...]  [COMMAND command2 [ARGS] [args2...] ...]  [DEPENDS [depends...]]  [WORKING\_DIRECTORY dir]) |

其中，OUTPUT 后面跟上我们要生成的文件，COMMAND 后面跟上我们要运行的命令行命令；DEPENDS 后面跟上生成该文件需要的依赖文件；WORING\_DIRECTORY 后面紧跟这些命令的工作目录。下面举几个例子：

如果希望产生一个名为test.txt的文件，并向其中写入“helloworld”,那么可以这么写：

|  |
| --- |
| add\_custom\_command(OUTPUT test.txt  COMMAND touch text.txt  COMMAND echo “helloworld” > test.txt) |

如果我们希望，从已有的test.txt文件复制产生一个新的my.txt文件，可以这么写：

|  |
| --- |
| add\_custom\_command(OUTPUT my.txt  DEPENDS test.txt  COMMAND cp test.txt my.txt) |

1. add\_custom\_target

|  |
| --- |
| add\_custom\_target(Name [ALL]  [COMMAND command2 [args2...] ...]  [DEPENDS depend depend depend ... ]  [WORKING\_DIRECTORY dir]) |

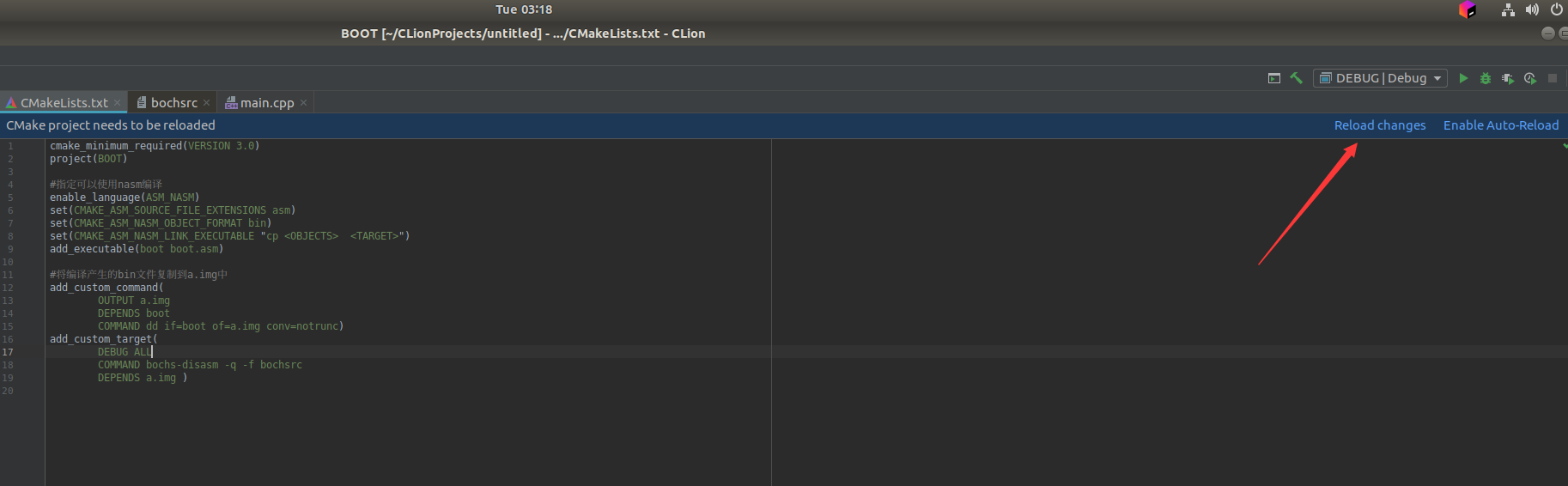
此命令是用于添加target的，其中Name要填写target的名字，ALL是可选的，其他的参数含义与add\_custom\_command中的相同。CLion会为每个taregt增加一个选项。

## 一个例子

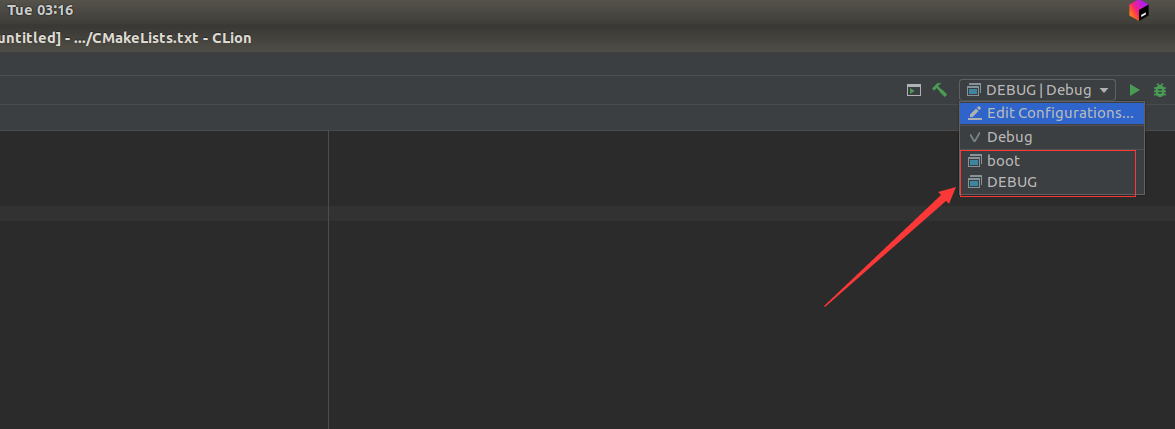
这里举个简单的例子，我们可以通过编写CMakeList将引导扇区的编译、生成、调试过程自动化。CMakeList.txt内容如下所示，整个项目在提供的hello.zip中：

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.0)  project(BOOT)  #指定可以使用nasm编译  enable\_language(ASM\_NASM)  set(CMAKE\_ASM\_SOURCE\_FILE\_EXTENSIONS asm)  #这里修改了变量CMAKE\_NASM\_COMPILE\_OBJECT，这个变量的作用是告诉CMake使用我们设定的命令来编译asm文件。这里的命令将asm文件编译为bin格式。  set(CMAKE\_ASM\_NASM\_COMPILE\_OBJECT "nasm <INCLUDES> <FLAGS> -f bin -o <OBJECT> <SOURCE>")  #这里修改了变量CMAKE\_ASM\_NASM\_LINK\_EXECUTABLE，它的作用是告诉CMake使用我们设定的命令来链接文件。由于我们最终需要的就是bin格式文件，不需要进行链接，所以这里就直接复制了。  set(CMAKE\_ASM\_NASM\_LINK\_EXECUTABLE "cp <OBJECTS> <TARGET>")  add\_executable(boot boot.asm)  #将编译产生的bin文件复制到a.img中  #在CLion中，以下命令的工作目录与CMakeList所在的目录是不同的，因此要先把需要的文件拷贝过去  add\_custom\_command(  OUTPUT a.img  DEPENDS boot  COMMAND cp ${PROJECT\_SOURCE\_DIR}/a.img a.img  COMMAND dd if=boot of=a.img conv=notrunc )  add\_custom\_target(  DEBUG ALL  COMMAND cp ${PROJECT\_SOURCE\_DIR}/bochsrc bochsrc  COMMAND bochs-disasm -q -f bochsrc  DEPENDS a.img) |

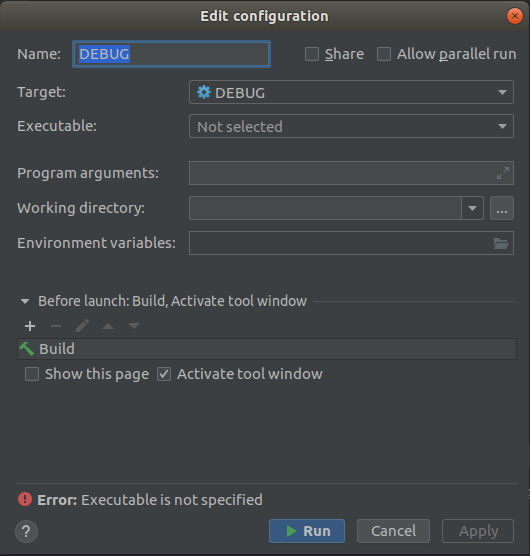
将asm文件和CMakeList文件编写好之后。如果CLion显示如下的提示，那么点击“Reload Changes“。



然后，打开下图中的下拉列表，选择Debug。



然后在弹出的对话框中直接点“Run“，之后可能会出现警告，我们点击”Continue any way”就可以了。



最后，就会出现bochs-disasm的调试界面。