

GCC/Make/CMake 之 GCC



FRONTI...

Formal了吗?

544 人赞同了该文章

本系列教程将介绍现代C/C++项目的构建编译常用工具链，GCC，Make，以及CMake。其中，GCC是C/C++语言的编译工具，Make是增量式（编译）批处理工具，CMake是Make脚本生成工具。在现代C/C++项目的构建中，它们的关系如下。

```
cmake          make          gcc
CMakeList.txt  ----> Makefile ----> Cmake ----> Binary
```

开发者需要编写 CMakeList.txt 文件，来配置项目相关的CMake参数。通过运行 cmake 命令，自动生成对应平台的Make工具自动构建脚本 Makefile 文件。当然，CMake也支持生成其他的构建工具的配置文件，比如Xcode的 xxxx.xcodeproj，Visual Studio的 xxxx.sln，Ninja的 xxxx.ninja 等等。目前，大多数开源的C/C++项目都支持使用CMake生成 Makefile 文件，再调用 make 命令，使用Make工具进行自动构建。Makefile 文件可以看成是一系列依赖于文件的Shell命令。它基于文件修改的时间戳来实现增量式处理。具体规则大致如下，若生成的目标文件的时间戳早于依赖文件的时间戳时，则执行对应的命令，重新生成目标文件。这实际上暗示了，Make工具不只用于编译，还可以用于其他的增量式文件生成任务。使用Make工具来编译C/C++项目时，一般会使用Shell命令来调用 gcc，自动化且增量式地实现C/C++源代码的编译链接等一系列工作。

GCC

在早期，GCC为GNU C Compiler的简写，即GNU计划中的C语言编译器。但经过多年的扩展和迭代，GCC逐渐支持C、C++、Objective-C、Fortran、Java、Ada和Go等越来越多语言的编译。因此，其GCC被重新定义为GNU Compiler Collection，即GNU编译器套件。在本篇中，我们仅介绍使用GCC编译C/C++项目。

值得注意的是，Apple公司曾经一直使用GCC作为官方的编译器。但是由于GCC开发社区对Apple所提的需求，给予的优先级始终不高，甚至很多Apple的重要需求基本不做考虑。于是，财大气粗的Apple一怒之下，决定放弃GCC，基于LLVM重新开发了编译工具Clang，支持C、C++、Objective-C等语言。因此，目前macOS上自带的默认 gcc 命令，实际上调用的是 clang。希望在macOS上使用GCC，需要自行安装，如使用macOS上常用的包管理工具 Homebrew，（brew install gcc）。比较幸运的一点是，clang 在使用上（调用方式，参数等方面），基本复刻了 gcc。所以，在本篇中，笔者虽然讲GCC，但实际上给出的例子，都是使用的Clang，但基本上差异不大，不会导致太大的问题。

编译流程

使用 gcc 编译C/C++程序时，主要的编译流程如下，包含预处理、编译、汇编、链接等四个步骤。以输入C语言程序源码文件 b.c 为例，直接调用命令 gcc b.c，将会完整执行以下流程，并生成对应的可执行的二进制文件 a.out。注意，这里 gcc 的默认输出就是固定的 a.out。在GCC工具链中，汇编由工具 as 完成，链接则由工具 ld 完成。

```
      -E          -S          -c
b.c ----> b.i ----> b.s ----> b.o ----> a.out
gcc      gcc      as      ld
```

对 gcc 使用以下指令，将会使其编译流程停止在对应位置：

- -E，（prEprocessing），执行到预处理步骤之后，即处理C/C++源码中 # 开头的指令，包括宏展开以及 #include 头文件引入等等。该指令默认不输出文件。可以使用 -o 指令输出约定后

▲ 赞同 544 ▼

● 21 条评论

➤ 分享

♥ 喜欢

★ 收藏

📄 申请转载

...



- `-S` , (**a**Ssembly) , 执行到**编译**步骤之后, 生成汇编文件, 但不生成二进制机器码。该指令默认的输出文件后缀为 `*.s` 。
- `-c` , (**c**ompilation) , 执行到**汇编**步骤之后, 调用工具 `as` , 从汇编码生成二进制机器码, 但不进行链接。该指令默认的输出文件后缀为 `*.o` (**o**bject) 。
- 不带以上参数调用 `gcc` 将会完整执行以上流程, 即执行到**链接**(linking) 步骤之后。链接步骤实际上调用链接工具 `ld` 来执行, 会将源码生成的二进制文件, 库文件, 以及程序的启动部分进行组合, 从而形成一个完整的二进制可执行文件。

特别的, 使用指令 `-o` , (**o**utput) , 可以指定输出文件的名称。例如 `gcc b.c -o b.bin` , 将生成可执行文件 `b.bin` , 而不是默认的 `a.out` 。

以上指令都可以在编译流程任意环节的基础上进行调用, 例如:

```
> gcc -E b.c -o b.i
> ls
b.c b.i
> gcc -S b.i
b.c b.i b.s
> gcc -c b.s
b.c b.i b.o b.s
> gcc b.o
b.c b.i b.o a.out b.s
```

编译参数

对于一个实际的C/C++项目而言, 源文件一般不会只有一个, 而且绝大多数情况下会使用到**第三方库** (Third-party Library) 。由于C/C++没有官方的**包管理工具** (Package Manager) , 如Python的 `pip` , Java的 `maven` , Nodejs的 `npm` 等等, 所以, 在C/C++项目中使用第三方库时, 一般使用系统自带的包管理器来进行第三方库的安装, 例如Ubuntu下的 `apt-get` , macOS的 `brew` (Homebrew) 等等。同时, 也可以选择自行获取第三方库的源码进行编译安装。

第三方库主要由两个部分组成, 即 a) **头文件**, b) **库文件**。头文件一般是一系列名为 `xxx.h` (head) 的文件, 相当于暴露出第三方库所提供的API接口(函数签名)。库文件一般会包含**静态库文件**和**动态库文件**, 相当于第三方库在功能上的二进制实现。其中, 静态库文件是一系列名为 `libxxx.a` (archive) 的文件 (Windows下为 `libxxx.lib` , **library**) 。动态库文件则是一系列名为 `libxxx.so` (shared object) 的文件 (Windows下为 `libxxx.dll` , **dynamic link library**, macOS下为 `libxxx.dylib` , **dynamic library**) 。系统自带的, 以及由系统包管理器安装的第三方库, 其头文件一般在 `/usr/include` 或 `/usr/local/include` 路径下, 库文件一般在 `/lib` , `/usr/lib` 和 `/usr/local/lib` 目录下。

正是由于以上因素的影响, GCC工具链不负责管理第三方库, 因此无法判定C/C++项目具体需要使用哪些库, 以及这些库的准确信息, 如位置、版本等。所以, 仅使用GCC, 无法完全自动地解决C/C++项目第三方库的依赖问题。我们需要人工地添加各种编译参数, 如 `-I` , `-l` 以及 `-L` , 将所依赖的第三方库的相关信息, 传递给 `gcc` 编译器。其中 `-I` 传递的是头文件所在的目录, `-l` 传递的是需要链接的库的名称, `-L` 传递的是库文件所在的目录。

-I 参数

回顾之前所介绍的GCC编译流程, 在**预处理**阶段需要处理 `#include` 指令, 将包含的头文件替换进源码。一般来说, 在进行预处理时, `gcc` 会自动在当前工程目录下, 以及 `/usr/include` 目录下寻找对应的头文件。

但对于位于其他目录下的第三方库的头文件, `gcc` 无法自动找到所需头文件的位置, 会报出形如 `xxx.h: file not found` 的错误。我们需要使用 `-I` 参数来指定第三方库头文件的位置。例如, 在macOS下, 使用Homebrew包管理器安装 `llvm` , 会相应地安装LLVM项目所包含的第三方库, 其对应的头文件位于 `/usr/local/opt/llvm/include` 目录。

五我们在使用LLVM提供的库时, 可以使用 `-I /usr/local/opt/llvm/include` (或者 `-I`

`gcc` 会额外在 `-I` 参



数指定的目录下搜索对应的头文件。 `-I` 参数可以**重复多次**使用，从而指定多个额外的头文件目录。 `-I` 参数一般指定**绝对路径**，但也可以用**相对路径**，比如头文件在当前目录，可以用 `-I.` 来指定。

需要注意的是，在C/C++源码中，使用 `#include"xxxx.h"` 语句时，其中的 `xxxx.h` 可以带上路径。我们甚至可以使用绝对路径来引用头文件。比如说，存在头文件 `/usr/local/opt/llvm/include/llvm/Pass.h`，我们在使用它时，可以直接通过这样的方式引用 `#include"/usr/local/opt/llvm/include/llvm/Pass.h"`。

不过，在C/C++工程中，并不推荐这种做法。比较推荐的做法是，使用相对路径加参数 `-I include_dir` 的方法来引用头文件。比如以上的例子中，我们会直接在源码中使用 `#include"llvm/Pass.h"`，并且将llvm库的头文件所在目录，通过参数 `-I /usr/local/opt/llvm/include` 传递给 `gcc`。这样做能够灵活地管理第三方库版本，也便于不同机器下的多人协作开发，比直接包含绝对路径头文件要好很多。

总而言之，`gcc` 在进行预处理时，会将库文件目录（如 `-I` 参数传递进来的目录，以及默认的 `/usr/include`，`/usr/local/include` 等目录），与程序源码中 `#include"xxxx.h"` 语句的 `xxxx.h` 进行组合拼接。倘若某个组合，得到的路径存在实际的头文件，那么就会将该头文件包含进来。

-l 参数

在GCC编译流程的链接阶段，会默认链接标准库，如 `libc.a`，但是对于第三方库，就需要手动添加。倘若在编译中报出如下的错误： `Undefined symbols for architecture x86_64: xxx...xxx ld: symbol(s) not found for architecture x86_64` 这一般是由**未正确指定**需要链接的第三方库导致的。

在使用 `gcc` 时，一般会选择使用 `-l` 参数来指定需要链接的库。例如，假定我们使用了 `math` 库（即 `#include<math.c>`），在进行编译时，便会报出如上的 `Undefined` 错误。这时，我们可以使用 `-lm`（或者 `-l m`）参数来指定需要链接 `math` 库。

注意，某些gcc编译器会把 `math` 库视为标准库进行自动链接。这时我们需要加上 `-nostdlib` 参数，使其不自动链接标准库，才会报出如上的 `Undefined` 错误。

初看 `-lm` 参数，可能会感觉有些诡异。那么，`-l` 参数具体是如何使用的呢？`-l` 参数后需要接**库名**（如 `m`），而不是**库文件名**（如 `libm.so`）。但库名和库文件名之间，存在非常直观的联系。以 `math` 库为例，其库文件名是 `libm.so`，而库名是 `m`。从中很容易看出，库名就是把库文件名的前缀 `lib` 和后缀名 `.so` 去掉后得到的。再比如说，LLVM包含的库文件 `libLLVMCore.a`，其对应的库名就是 `LLVMCore`，而链接它的参数为 `-lLLVMCore`。

-L 参数

位于 `/lib`，`/usr/lib`，`/usr/local/lib` 等目录下的库文件，例如 `libm.so`，在使用 `-l` 参数后，可以直接被链接。但如果库文件不在这些目录里，只用 `-l` 参数，进行链接时仍会报错，`ld: library not found for -lxxx`。这意味着链接程序 `ld` 在当前的库文件路径中，无法找到 `libxxx.so` 或 `libxxx.a`。

这时，我们需要使用 `-L` 参数，将所要链接的库文件所在的路径告诉 `gcc`。`-L` 参数后需要跟库文件所在的路径。例如，在macOS下，使用Homebrew包管理器安装 `llvm`，其对应的库文件位于 `/usr/local/opt/llvm/lib` 目录。倘若我们需要使用库 `LLVMCore`，即链接库文件 `libLLVMCore.a`，除了添加 `-lLLVMCore` 参数外，还需要使用参数 `-L/usr/local/opt/llvm/lib`，告诉 `gcc` 库文件所在的目录。

其他编译参数

除了以上的这些参数外，`gcc` 还有一些其他的参数，也是比较重要的，在此分别简要介绍。



在前面讲库文件的时候，我们提到了**静态链接库文件**（`libxxx.a`）和**动态链接库文件**（`libxxx.so`）。我们并未提及两者的区别。其实，我们通过如下的方式简单进行理解。gcc 链接静态库文件，会将静态库文件中用到的部分，**拷贝**到生成的二进制程序中，从而导致生成的文件比较大；而链接动态库文件，则不会进行拷贝，所以生成的二进制程序会比较小。链接动态库文件的缺点是，在其他机器上运行该程序时，要求其上正确安装了对应的动态库文件。相应的，链接静态库文件生成的程序，则没有这个要求。

在使用 gcc 进行链接时，**默认优先**使用动态链接库文件。仅当动态链接库文件不存在时，才使用静态链接库文件。如果需要使用静态链接的方式，则需要在编译时加上 `-static` 参数，强制使用静态链接库文件。例如，在 `/usr/local/opt/llvm/lib` 目录下，同时存在库文件 `libunwind.so` 和 `libunwind.a`。为了让 gcc 在链接时使用静态链接库文件 `libunwind.a`，我们可以添加 `-static` 参数，使用如下编译命令 `gcc hello.o -static -L/usr/local/opt/llvm/lib -lunwind`。

B. 优化参数

编译优化也是编译器的重要功能，适当的编译优化能大大加速程序的执行效率。gcc 提供了4级优化参数，分别是 `-O0`、`-O1`、`-O2`、`-O3`。一般来说，数字越大，所包含的编译优化策略就越多。此外，gcc 还提供了特殊的 `-Os` 参数。

- `-O0` 参数表示不使用任何优化策略，是 gcc 默认的优化参数。因为没有使用任何优化策略，编译得到的机器码与程序源码**高度对应**，两者之间基本可以建立一一对应的关系。所以，`-O0` 优化非常适合用于程序调试，并且通常和生成调试信息的参数 `-g`（generate debug information）配合使用。`-g` 参数会在编译时给生成的二进制文件附加一些用于代码调试的信息，比如符号表和程序源码。
- `-O1` 会尽量采用一些不影响编译速度的优化策略，**降低**生成的二进制文件的大小，以及**提高**程序执行的速度。
- `-O2` 使用 `-O1` 中的所有优化策略，还会采用一些会**降低**编译速度的优化策略，以**提高**程序的执行速度。
- `-O3` 在 `-O2` 的基础上，使用更多的优化策略。这些额外的优化策略会进一步**降低**编译速度，而且会**增加**生成的二进制文件的大小，但程序的执行速度则会进一步**提高**。
- `-Os` 则和 `-O3` 优化的方向相反。它在 `-O2` 的基础上，采用额外的优化策略，尽量的**降低**生成的二进制文件的大小。

倘若对各优化参数下，所开启的优化策略感兴趣，或者希望了解其他的优化参数，可以参考[1]。

C. 宏相关参数

有时，为了保证C/C++项目的跨平台性，或者在编译时，能比较灵活地在多个相似的库中作出选择，需要在源码中使用**条件编译**。条件编译即使用 `#ifdef M`，`#else`，`#endif`（或 `#ifndef M`，`#else`，`#endif`，以及 `#if`，`#elif`，`#else`，`#endif`）等指令，通过**宏定义**来控制需要编译的代码。

C/C++语言中，可以使用 `#define M` 语句在源码中定义宏 `M`。但是条件编译一般需要从外界，如编译器，传入一个宏定义。因此，gcc 提供了宏定义参数 `-D` 以及取消宏定义参数 `-U`。在使用 gcc 进行编译时，可以通过如下的方式，来进行相应的宏操作：

- `-Dmacro` 定义宏 `macro`，默认将其定义为 1，相当于在程序源码中使用 `#define macro` 语句。
- `-Dmacro=def` 定义宏 `macro` 为 `def`，相当于在程序源码中使用 `#define macro=def` 语句。
- `-Umacro` 取消宏 `macro` 的定义，相当于在程序源码中使用 `#undef macro` 语句。
- `-undef` 取消所有非标准宏的定义。

D. 其他

此外，还有一些其他的参数，也很重要，例如：

- `-std` 参数可以指定编译使用的C/C++标准。例如，`-std=c++11` 表示使用C++11标准，`-std=c99` 表示使用C99标准。特殊的，`-ansi` 表示使用ANSI C标准，一般等同于 `-std=c90`。
- `-Werror` 参数要求 gcc 将产生的**警告**（Warning）当成**错误**（Error）进行显示。



- `-wl` 参数告诉 `gcc`，将后面跟随的参数传递给链接器 `ld`。
- `-v` 参数可以显示 `gcc` 编译过程中一些额外输出信息。

倘若希望了解 `gcc` 的其他参数，可以通过 `gcc --help` 或者 `man gcc` 查看，也可以直接参考 [GCC 手册\[1\]](#)。

pkg-config

一般来说，人工编辑第三方库的编译链接参数是比较麻烦的。我们需要查找第三方库的头文件、库文件的安装路径，了解第三方库需要链接哪些其他的库，了解第三方库需要哪些编译参数等等。这些都不利于第三方库的快速集成。目前，很多现代的第三方库都提供了其对应的编译参数**自动生成工具**，一般名为 `xxx-config`。比如 `llvm` 就提供了 `llvm-config` 工具。在使用系统包管理器，或者自行编译安装了 `llvm` 后，可以直接调用 `llvm-config` 命令。我们以 `llvm 10.0` 为例，进行说明。

- 执行 `llvm-config --cxxflags`，可以得到 `-I/usr/local/Cellar/llvm/11.0.0/include -std=c++14 -stdlib=libc++ -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS`。这是编译 `llvm 10.0` 提供的库，所需的编译参数。它说明 `llvm 10.0` 的头文件目录是 `/usr/local/Cellar/llvm/11.0.0/include`，并且要求使用 C++14 标准，使用 C++ 标准库，还定义了一些编译时需要的宏。
- 执行 `llvm-config --ldflags`，可以得到 `-L/usr/local/Cellar/llvm/11.0.0/lib -Wl,-search_paths_first -Wl,-headerpad_max_install_names`。这是链接 `llvm 10.0` 提供的第三方库所需要的链接参数。它告诉编译器，第三方库的位置在 `/usr/local/Cellar/llvm/11.0.0/lib`，并会传递一些其他的参数给链接器 `ld`。
- 执行 `llvm-config --libs` 会得到 `-lLLVMXRay -lLLVMWindowsManifest ... -lLLVMDemangle`。这是 `llvm 10.0` 可以链接的全部库。一般我们不会选择链接所有的库。而是会使用形如以下的命令 `llvm-config --libs core`，得到 `-lLLVMCore -lLLVMRemarks -lLLVMBitstreamReader -lLLVMBinaryFormat -lLLVMSupport -lLLVMDemangle`。这是使用 `core` 模块所需要链接的库。
- 执行 `llvm-config --system-libs` 会得到 `-lm -lz -lcurses -lxml2`。这是 `llvm 10.0` 所需用到的系统库。

一般来说，我们会将以上命令的参数进行组合使用，例如调用 `llvm-config --cxxflags --ldflags --system-libs --libs core`，就可以得到我们所需的全部编译参数。

除了第三方库自带的 `xxx-config` 以外，很多现代的第三方库都可以使用工具 `pkg-config` 来生成编译参数。我们可以用 `pkg-config --list-all` 命令，来查看其所支持的所有第三方库。`pkg-config` 的一般使用方法是调用形如 `pkg-config pkg-name --libs --cflags` 的命令。例如，倘若要使用 `gmp` 库，我们可以执行 `pkg-config gmp --libs --cflags`，得到如下输出 -
`I/usr/local/Cellar/gmp/6.2.1/include -L/usr/local/Cellar/gmp/6.2.1/lib -lgmp`。

我们可以直接复制这些输出，再粘贴到 `gcc` 命令后，也可以使用形如 `"gcc a.c `pkg-config gmp --libs --cflags`"` 的命令，通过内嵌 shell 命令的方式，将第三方库的编译参数传递给 `gcc`。

References

- [1] Using the GNU Compiler Collection (GCC), 3.11 Options That Control Optimization, [gcc.gnu.org/onlinedocs/...](http://gcc.gnu.org/onlinedocs/)
- [2] GCC 10.1 Manuals, [gcc.gnu.org/onlinedocs/...](http://gcc.gnu.org/onlinedocs/)
- [3] 本篇的某些段落，借鉴和参考了网络上的博客资料，但实在难以溯源，故未列为参考。先在此表示感谢！如有侵权，请联系我删除。

编辑于 01-06

[GCC](#) [make](#) [CMake](#)

▲ 赞同 544 ▼ ● 21 条评论 ➦ 分享 ♥ 喜欢 ★ 收藏 📄 申请转载 ...



推荐阅读



CMake应用

从编译过程理解CMake

cmake应用：从编译过程理解CMake

很酷的程序... 发表于CMake...

使用CMake构建C++项目

背景本文中引用了libdeepvac项目的CMakeLists.txt来作为参考：
<https://github.com/DeepVAC/libdeepvac>
本文的结构如下：CMake语句；CMake中的流程控制；CMake中的内置变量；CMake...

Gemfi... 发表于SYSZU...



5分钟理make/

Error

21 条评论

⇌ 切换为时间排序

- 写下你的评论...  
-  黄点点

03-13

太干了，对于一个新手来说太足了，我是初学者，这两周每天必刷一遍。非常感谢作者

 4
-  陈伯伯

03-09

写得太好了，很有条理！感谢！

 1
-  雷贤

03-08

期待楼主后的总结，为不熟悉编程工具链的小伙伴们释疑

 1
-  ddd

03-06

感谢大佬的总结，会持续关注大佬，希望大佬能出一期cmakelist的常见用法。

 1
-  Leeeeee

07-10

有点小疑问，将预处理后的文件变成汇编代码这个过程不应该是叫compilation编译吗，为什么是assembly呢？

 赞
-  推杯问盏

05-21

条理清晰，细节到位。期待更新 cmake

 赞
-  美国银虎斑猫

04-09

写的很好，期待更新

 赞
-  龙猫猫

03-28

博士总结的真好👍👍👍
太实用了
怎么训练出来的哦

-  inccc

盼更新呀，求cmake系列~🙏🙏

👍 赞

03-26
-  Hello World

写的很好，点赞喜欢收藏一气呵成
催更催更😓😓

👍 赞

03-25
-  迪吧个神

学习了

👍 赞

03-24
-  迪吧个神

收益很大，谢谢

👍 赞

03-21
-  Christopher.Zhu

非常非常非常非常好！

👍 赞

03-17
-  知乎用户

清华的软件工程博士一般研究啥课题？

👍 赞

02-20
-  知乎用户 回复 知乎用户

清华软院四个所，图形所，系统所，信息所，网络所。研究内容就是所名字对应的板块些，其实主要看导师的方向。

👍 赞

02-25
-  给我一瓶解药

[拥抱]整理得很好哦，不知道后续会不会出cmake和makefile的总结🤔

👍 赞

02-17
-  zhkdleoxywk

感谢大佬的分享总结👍👍

👍 赞

02-17
-  王晓东

十分感谢您的分享，真的很详细

👍 赞

02-16
-  小二哥

大佬我咨询下我最近想交叉编译一个openwrt下使用的cmake，我是不是需要重写makefile啊

👍 赞

02-16
-  NaiveTomcat 回复 小二哥

如果是autotools工具链的话，configure脚本是有方便的交叉编译相关选项的，可以查一下

👍 赞

02-17
-  小二哥 回复 NaiveTomcat

好的，我看看，谢谢啦🙏

👍 赞

02-18