22 | 生产加速：如何使用结构化编译加速 C 项目构建？

2022-02-09 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 12:11　大小 11.16M

你好，我是于航。

在之前的课程中，我们曾遇到过很多段示例代码。而这些代码有一个共性，就是它们都十分短小，以至于可以被整理在一个单独的 .c 文件中。并且，通过简短的一行命令，我们就可以同时完成对代码的编译和程序的运行。

但现实情况中的 C 项目却往往没这么简单，动辄成百上千的源文件、各种各样的外部依赖与配置项，这些都让事情变得复杂了起来。因此，当 C 项目的体量由小变大时，如何组织其源代码的目录结构与编译流程，就成了我们必须去着重考虑的两个问题。而今天我们就来聊一聊，应该从哪些角度看待这两个问题。

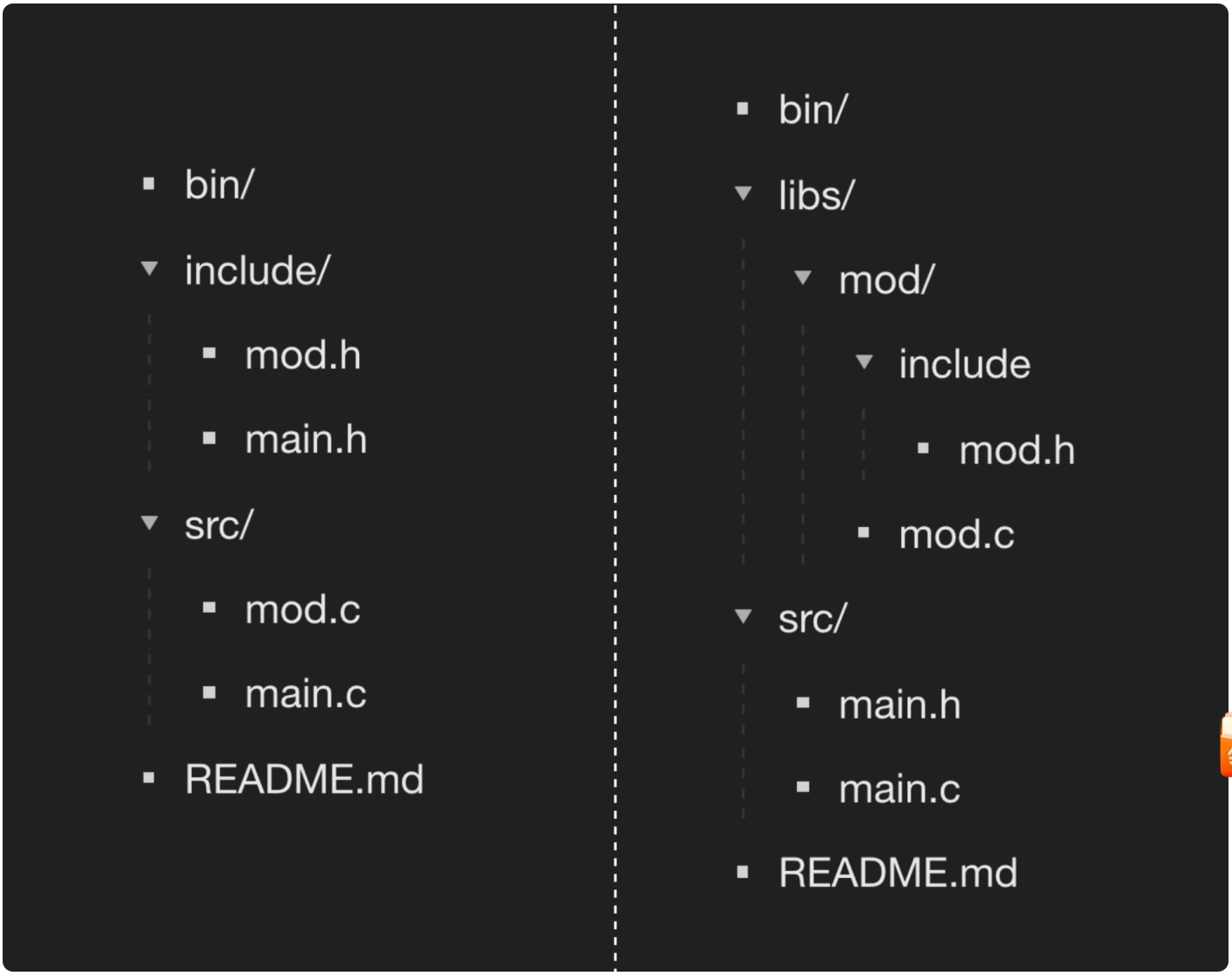
如何组织 C 项目的源代码目录结构？

我们先来看与源码目录结构相关的话题。其实，对于 C 项目的源代码目录结构，应该使用哪种组织方式，通常没有所谓的“最佳实践”，而是要具体问题具体分析。

对于小型项目，我们可以简单地将 .h 与 .c 这两类源文件分别归纳在两个独立的目录 include 与 src 中，甚至是全部混放在同一个目录下。而当项目逐渐变大时，不同的 C 源文件就可以按照所属功能，再进行更细致的划分。

比如，能够以模块为单位，以库的形式进行抽象的实现，可以统一放在名为 libs 的目录下进行管理。而使用库接口实现的应用程序代码，则可放置在名为 src 的目录中。其他与 C 源代码没有直接关系的文件，可以自由保存在项目根目录，或放置在以对应分类命名的独立目录内。

在下图中，我给出了两种你可以参考的目录结构。但需要注意的是，并没有默认的或最好的 C 项目目录结构，无论采用哪种形式，你都要随着项目的发展而学会不断变通。

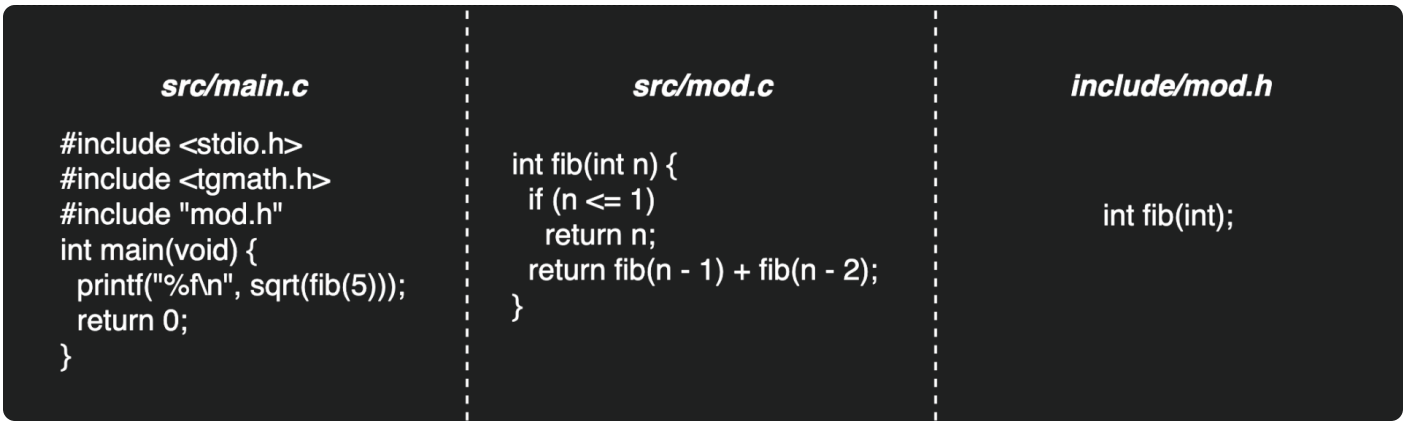


对于源代码目录结构的组织，一个基本原则是“清晰易懂”。其中，“清晰”是指即使在不了解具体实现的情况下，仅通过一层层展开项目代码的目录树，我们也能够以自顶向下的方式，来了解它在代码层面的基本组成结构。而“易懂”则是指在上面这个过程中，通过观察文件夹和文件的名字，我们可以对项目的基本功能与模块化实现有一个大致印象。

如何组织 C 项目的编译流程？

随着源代码目录被不断调整，项目的编译流程也相应地发生了变化。

假设有一个简单的 C 项目，它一共包含有三个源文件。按照我在上面介绍的第一种目录组织方式，这些文件被分别整理在项目根目录下的 src 与 include 文件夹内。而它们各自包含的内容则如下图所示：



其中，文件 src/main.c 为程序入口 main 函数的所在文件。而 src/mod.h 与 include/mod.c 两个文件，则一同为模块 mod 提供了相应的外部接口声明与具体实现。

按照我们之前的习惯，通过下面这行命令，便能够借助 GCC 编译器来完成对这个项目的编译过程。其中，我们指定了所有需要参与编译的 .c 文件。使用 `-I` 选项，我们为编译器指定了在查找头文件时，需要搜索的目录，即 “./include”。而使用 `-l` 选项，程序运行时依赖的 math 数学库可以在运行时被顺利链接。

复制代码

```
gcc src/main.c src/mod.c -I./include -lm -o bin/main
```

到这里，你可能会觉得相较于单文件 C 应用来说，多文件 C 应用的编译也不过如此，只是命令中参与编译的源文件数量和使用到的配置项多了一些。

但随着项目体量的逐渐增大，这种编译方式会面临两个重要问题。首先便是如何对冗长的编译命令进行管理。这个问题关系到，**我们是否可以清楚地知道项目每次编译时的具体状态，以及能否快速准确地对这些配置项进行相应修改**。

其次，上述命令在每一次执行时，都仅会生成最终的二进制可执行文件，这使得编译的中间结果无法被有效利用。因此，代码在每一次修改后，都需要再次经历完整的编译流程。对于大型项目来说，这无疑降低了开发效率。

那有没有办法来解决这两个问题呢？答案是肯定的。首先来看，我们可以如何利用 Makefile 来进行结构化的 C 项目编译。

使用 Makefile 进行结构化编译

Makefile 是一种在（类）Unix 操作系统中常用的，用于组织项目代码编译流程的方式，它通常需要配合名为 make 的构建自动化工具一起使用。

make 最初由贝尔实验室的 Stuart Feldman 于 1976 年实现，后来被整合到了 Unix 系统中。make 在执行时，会去搜索当前目录下名为 Makefile 的文本文件，并按照其内部指定的一系列规则，有序地对项目进行编译。

比如，对于上面提到的例子，我们可以编写如下所示的一段文本内容，并将它保存到项目根目录下名为 Makefile 的文件内。紧接着，通过在该文件所在目录下直接执行 make 命令，项目得以被正确编译。


```
bin/main: src/main.c src/mod.c
  gcc src/main.c src/mod.c -I./include -lm -o bin/main
```

Makefile 使用了一种与声明式编程语言类似的简化语法，以方便开发者灵活配置项目的编译流程。这里，上述配置文本的第一行指定了一个编译目标（bin/main），以及与该目标相关的依赖文件（src/main.c 与 src/mod.c）。而接下来以 Tab 键缩进的所有行（这里即第二行），均用于配置依赖文件到目标文件的编译转换细节。可以看到，我们使用了与之前完全相同的命令来实现这个过程，但是两者在编译时的差异已逐渐显现。

通过这种方式，我们已经部分解决了之前提到的问题，即每一次代码修改后，由于直接运行编译命令导致“全量编译”，进而带来的开发效率下降。make 命令在每次实际进行编译前，都会首先追踪各个编译目标与其依赖项的版本信息（通常为“最后修改时间”）。而只有当相关依赖的内容在上一次编译后发生改变，或目标文件不存在时，才会再次编译该目标。通过这种方式，我们可以**将大部分时间内的项目编译过程都集中在必要的几个源文件上**，而不用“浪费”已编译好的其他中间目标文件。

接下来，我们尝试进一步优化 Makefile 中的配置项，来让最终的二进制编译目标与各个中间依赖项作进一步分离。并且，通过抽离编译命令中的可配置部分，我们也可以让整个编译脚本变得更具可读性与可用性。优化后的文件内容如下所示：

```
# 用于控制编译细节的自定义宏；
CC = gcc
CFLAGS = -I./include
LDFLAGS = -lm
TARGET_FILE = bin/main

# 描述各个目标的详细编译步骤；
$(TARGET_FILE): $(patsubst src/%.c,src/%.o,$(wildcard src/*.c))
  $(CC) $^ $(LDFLAGS) -o $@

src/%.o: src/%.c include/%.h
  $(CC) $< $(CFLAGS) -c -o $@
```

可以看到，通过以 “#” 开头的注释信息，我们将整个 Makefile 文件的内容划分成了两个部分。

第一部分包含用户可配置的一些宏常量，这些宏将在 make 运行时被替换到下面已经配置好的具体编译命令中。这样，用户可以通过修改这些量值来在一定范围内自定义期望使用的编译流程。

而第二部分则对应于各个编译目标的具体编译细节，这里我们将最初的那条编译命令拆分成了如下两步：

1. 编译器将 src 与 include 文件夹内同名的 .c 与 .h 文件编译为对应的 .o 对象文件；
2. 编译器将所有的 .o 文件一次性编译，并生成最后的二进制可执行文件。

利用这种方式，我们增加了可复用的中间编译结果，使通过 make 命令进行的每一次编译过程，都仅局限在被修改的 .c 或同名的 .h 文件上。如此一来，我们便可以做到最大程度上的“中间结果复用化”。

为了帮你理解这部分配置代码，我将代码中你可能不太熟悉的 Makefile 语法元素的含义进行了整理，并放在了下面的表格中，你可以参考：

Makefile 语法元素	含义
%	通配符，可以匹配任何数量的任意字符
$(wildcard pattern…)	将指定的 pattern（可包含有通配符）扩展为包含有真实元素的完整列表
$(patsubst pattern,replacement,text)	从指定的 text 中筛选出满足 pattern 的项，并将其替换为 replacement 指定的形式
$(macro)	在当前位置展开宏 macro
$^	特殊宏，在当前位置展开为所在编译目标的依赖项列表（“:” 右侧元素）
$@	特殊宏，在当前位置展开为所在编译目标（“:” 左侧元素）
$<	特殊宏，在当前位置展开为所在编译目标的依赖项列表中的第一个元素

Makefile 帮助我们很好地解决了单一编译命令具有的可读性低、中间结果复用性差等诸多问题。你可以点击这个链接，来了解与它的可用语法和 make 工具相关的更多信息。

不过，仔细观察后你会发现，我们在 Makefile 中使用的各类命令与参数选项，都与程序当前运行所在的操作系统和平台直接相关。那么，当同一个 Makefile 文件被拷贝到其他环境中时，它是否还能正常工作呢？答案是 “it depends”。但很明确的是，“Makefile + make” 这种方式，本身就无法直接在除了（类）Unix 以外的其他操作系统上使用。因此，如何进一步满足 C 项目的跨平台自动化编译，便成了社区思考的另一个重要方向。接下来我们看看这个问题是如何被解决的。

使用 CMake 进行跨平台的自动化构建

“抽象”通常是用来解决这类问题的一大法宝。为了保证项目编译脚本的可移植性，我们便不能使用与具体软硬件实现相关的各类信息。因此，我们可以采取这样一种简单的方式：通过提供

平台无关的中立配置选项，把与项目构建相关的所有重要特征“抽离”出来。并且，在项目开始真正编译之前，再根据目标平台的具体情况对项目进行构建。

接下来我为你介绍的工具 CMake（Cross-platform Make）便是按照这样的思路实现的。只不过相较于直接编译代码，CMake 会根据所在平台的具体情况，生成相应的“平台本地构建项目”。比如，在（类）Unix 系统上，它会生成项目对应的 Makefile 文件；而在 Windows 系统上，则会生成项目对应的 Visual Studio 工程文件。在此基础上，再利用所在平台上的相关工具，CMake 便可完成项目的真正构建。

同 Makefile 类似，CMake 规定用于描述项目编译细节的配置信息，也需要被保存在名为 CMakeLists.txt 的文本文件中。作为对比，你可以使用如下所示的这段 CMake 配置信息，来编译我们在这一讲开头处介绍的那个 C 项目。关于其中每一行“代码”的具体作用，你可以参考它们上方的注释来进一步理解。

复制代码

```
cmake_minimum_required(VERSION 3.10)
# 设置项目名称；
project(Test)
# 设置二进制目标文件名称；
set(TARGET_FILE "main")
# 添加源文件目录；
aux_source_directory(./src DIR_SRCS)
# 设置二进制目标文件的依赖；
add_executable(${TARGET_FILE} ${DIR_SRCS})
# 设置头文件查找目录；
target_include_directories(${TARGET_FILE} PUBLIC "${PROJECT_SOURCE_DIR}/include
# 设置需要链接的库；
target_link_libraries(${TARGET_FILE} PUBLIC m)
```

可以很明显地看到，相较于 Makefile，CMake 的配置信息更加清晰易懂。比如，关于命令 target_include_directories 的具体用途，我们从它的名字上就能猜个大概。实际上，它便对应于 GCC 编译器的 -I 参数，可用于指定查找头文件时的搜索目录。你可以点击这个链接，来查看与 CMake 有关的更多信息。

当项目的 CMakeLists.txt 文件编写完毕后，通过下面这几个步骤，我们便能够完成项目的编译：

1. 使用命令 `mkdir build && cd build` 创建并进入用于存放编译结果文件的临时目录；

2. 使用命令 `cmake ..` 生成本地化构建项目。这里，CMake 会根据用户在 CMakeLists.txt 中指定的信息，来对当前环境进行相关检查，其中包括针对编译器的 ABI、可用性，以及支持特性的检查等。而当检查结束后，CMake 便会根据检查结果，动态生成**可用于支持项目在当前环境下进行编译的本地化构建项目**；
3. 使用命令 `cmake --build .` 让 CMake 利用本地的相关工具，完成项目的最终编译过程。

其他可选用工具

其实，早在 CMake 出现之前，GNU 旗下就已经出现了类似的跨平台自动化构建工具，即 Autotools 工具集。它们可以帮助我们把各类项目的源代码移植到多种不同的 Unix 系统上。但由于其学习成本较高，使用较为繁琐，因此它们在逐渐被 CMake 取代。

当然，除此之外，还有 Meson、Tup、Bazel 等构建工具可供你选择。它们在使用方式上都不尽相同，你可以点击相应链接来了解更多信息。但从实际情况来看，具有完善的功能、成熟的社区及解决方案的 CMake ，无疑仍是目前进行 C 项目跨平台自动化构建的最佳选择。

总结

讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要为你介绍了如何组织 C 项目的源代码目录结构，以及如何在（类）Unix 系统上使用 Makefile 和 CMake 等工具，来进行 C 项目的结构化编译与跨平台自动化构建。

其中，对于如何组织 C 项目的源代码目录结构，社区并没有所谓的最佳实践。正确的方式是结合项目的实际情况，在保证清晰易懂的前提下，再对项目结构进行及时、动态的调整。

相较于每次使用完整的编译命令，结构化编译可以通过复用各类中间编译结果，进一步提升编译效率。同时，“可编程”的编译配置脚本也使得项目的编译细节更具可读性与可用性。

而借助 CMake 等工具，我们可以在此基础之上实现跨平台的项目自动化构建。CMake 通过抽象平台相关的配置信息，让开发者可以通过中立的方式描述项目的构建细节。而在项目被真正编译前，CMake 会进行一系列检查与分析，并最终生成适合在当前环境下使用的本地化构建项目。

思考题

你所在团队的项目使用了哪种自动化构建工具？你觉得它们有哪些优点和不足呢？欢迎在评论区跟同学们一起交流。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　21丨生产加速：如何使用自动化测试确保 C 项目质量？

下一篇　期中测试丨来检验下你的学习成果吧！

精选留言 (2)

写留言

Geek_4c94d2
2022-02-23

cmake，在linux下可以直接生成makefile，所以可以cmake .. ; make

1

Frank
2022-03-10

Cmake 可以用于windows 么？

作者回复: 看了下是支持的哈，具体可以参考这里：https://cmake.org/download/

共 3 条评论 >