GCC 编译器 30 分钟入门教程

作者：C 语言中文网

GCC 编译器 30 分钟入门教程

作者：C 语言中文网

GPL协议和自由软件

计算机软件作为人类的知识财富，为人类社会的发展起到了巨大的作用，但长期以来软件源码作为个人或公司的私有财产受到严格的保密，很难做到像文学艺术作品一样地进行公开的交流，很大程度上造成软件的低水平，重复劳动严重，在一定意义上制约了软件的发展。

GPL 的诞生

直到 1985 年由 MIT 教授理查德·斯托曼（Richard Stallman）提出应将软件源码看成人类共同拥有的知识财富，应该公开地自由交换、修改，提出了 GNU 计划（因英文名相同，GNU 的 logo 就是一只牛羚），并建立了自由软件基金会；同时，发布了一份举足轻重的法律文件，GNU 通用公共授权书（GNU GPL, GNU General Public License）。

该授权书主要有以下几点：

- 自由软件（free software）指的是源码自由，不是价格；
- 自由软件必须附带程序源代码，但可收取费用；
- 任何人都可以自由分发自由软件并收取费用，但必须列明原创者姓名；
- 任何人都可以修改源代码，但必须列明修改人名字，以保护原创者名誉；
- 任何人都可以采用源代码中的某一段，但其开发之软件必须也为自由软件（例如，如果 Netscap 是自由软件，而 IE 采用了其中的部份源代码，则 IE 也必须成为自由软件）；
- 任何自由软件的衍生品也必须是自由软件；
- 自由软件没有担保，以保护分发者。

1991 年，Richard Stallman 对授权做了微小的修改，即所谓的通用公共授权第 2 版。同时，他也推出了更宽松的通用公共授权，用于自由程序库。这一系列的授权有效地保护了自由软件不受商业软件的非法侵犯，例如，1998 年 Netscap 决定采用与 GPL 差不多的 NPL（Netscap Public Liscense），这样一来，Microsoft 就无法将 Netscap 中的源代码运用在 IE 上，除非它们也要成为自由软件。

至此，在 GPL 下人们就可以自由交流、修改软件源码了，这一协议极大地推动了整个计算机软件行业的发展，并带来了以下明显的益处：

- 对于广大计算机软件的学习者来说，可以直接从源码中吸取营养，缩短学习的时间，提高学习的效率，少走弯路，再也不必花大量时间去看那些不知正确与否的“未解之谜”了，学习在某种程度上变成了一件轻松愉快的事情了。
-
- 可以集中大家的智慧发展软件，避免重复劳动。一个软件只有公开源码，通过很多人的研究才有可能发现其中深藏的错误，大家才能公开探讨相关的问题，并进行改进，在大家的共同“挑剔与监督”下才有可能编写出尽善尽美的软件来。

GPL 协议的核心就是要对源码进行公开，并且允许任何人修改源码，但是只要使用了 GPL 协议的软件源码，其衍生软件也必须公开源码，准许其他人阅读和修改源码，即 GPL 协议具有继承性。

另一个问题就是 GPL 软件并非就是免费软件，这里所说的自由软件

是指对软件源码的自由获得与自由使用、修改，软件开发者不但可以通过服务来收费，而且还可以通过出售 GPL 软件来获利。

适应 GPL 协议的软件一般都是自由软件，自由软件是指一件可以让用户自由复制、使用、研究、修改、分发等，而不附带任何条件的软件。

copyleft 授权

Stallman 为了停止中间人对自由软件权利的侵害，提出了 copyleft 授权，因为自由软件在发布过程中可能会有一些不合作的人通过对程序的修改而将软件变成私有软件，将程序变成 copyleft 授权。

我们首先声明它是有版权的，而后加人了分发条款，这些条款是法律指导，使得任何人都拥有对这一程序代码或者任何这一程序的衍生品的使用、修改和重新发布的权力，但前提是这些发布条款不能被改变。这样在法律上，代码和自由就不可分割了。

自由软件的支持者相信，总有一天，随着自由软件的日渐成熟，自由软件终将主宰整个软件行业，人们不再受少数商业软件公司的控制，真正实现“市集式开发模式”。

GCC的发展历史

GNU 项目计划的主要目的是创建一个名叫 GNU's Not Unix(GNU) 的完全免费的操作系统。该操作系统将包括绝大多数自由软件基金会所开发的其他软件，以对抗所有商业软件，而这个操作系统的核心（kernel）就叫 HURD。

但是 GNU 在开发完全免费的操作系统上并未取得成功，直到 20 世纪 90 年代由林纳斯·本纳第克特·托瓦兹（Linus Benedict Torvalds）开发了 Linux 操作系统，GNU 才算在免费操作系统上完成了任务。

虽然 GNU 计划在开发免费操作系统上不成功，但是却成功开发几个广为流传的 GNU 软件，其中最著名的是 GNU C Complier(gcc)。

这个软件成为历史上最优秀的C语言编译器， 其执行效率与一般的编译器相比平均效率要高 20%~30%，使得那些靠贩卖编译器的公司大吃苦头，因为它们无法研制出与 gcc 同样优秀，却又完全免费、并开放源代码的编译器来。

而由于它又是 copylefted，所以一旦有用户发现错误，就会通知 Richard Stallman，所以几乎每个月都可以推出新版本。然而，它还有一个十分特殊而且不同寻常的意义：几乎所有的自由软件都是通过它编译的。可以说，它是自由软件发展的基石与标杆。

现在，gcc 已经可以支持 7 种编程语言和 30 种编程结构，是学术界最受欢迎的编译工具。

其他 GNU 软件还包括 GNU emacs、GNU Debugger(GDB)、GNU Bash 以及大部分 Linux 系统的程序库和工具等。

目前，gcc 已发展到了 8.x 的版本，几乎所有开源软件和自由软件中都会用到，因此它的编译性能会直接影响到 Linux、Firefox、OpenOffice.org、Apache 以及一些数不清的小项目的开发。gcc 无疑处在开源软件的核心地位。

作为自由软件的旗舰项目，Richard Stallman 在十多年前刚开始写作 gcc 的时候，还只是把它当作一个C程序语言的编译器；gcc 的意思也只是 GNU C Compiler 而已。经过这么多年的发展，gcc 已经不仅仅能支持C语言，它现在还支持 Ada、C++、Java、Objective-C、Pascal、COBOL 以及函数式编程和逻辑编程的 Mercury 语言等。因此，现在的 gcc 已经变成了 GNU Compiler Collection，也即是 GNU 编译器家族的意思了。这个名称同时也说明了 gcc 对于各种硬件平台无所不在的支持，甚至包括一些生僻的硬件平台。

gcc 不仅功能非常强大，结构也异常灵活。最值得称道的一点就是，它可以通过不同的前端模块来支持各种语言，如 Java、Fortran、Pascal、Modula-3 和 Ada 语言等。

总结

GUN 虽然没有开发出操作系统，但是却开发出了很多系统级的自由软件，GCC 就是其中之一。

GNU编译器套件

GCC 原来代表“GNU C Compiler”的意思。自从面世后，GCC 逐渐扩充、发展，现在不仅仅支持C语言，还支持其他很多语言，包括 C++、Ada、Objective-C、Fortran 和 Java 等。因此，GCC 的意思被重新定义为“GNU Compiler Collection”，也即“GUN 编译器套件”。

GUN 编译器套件包含多种前端处理器，以翻译各种不同语言。当然，在本教程中我们重点讨论的是基于C语言的前端处理器 GCC。

GCC 也是一种多目标（multitarget）编译器；换句话说，它通过使用可互换的后端处理器，为多种不同的计算机架构生成相应的可执行程序。

正如模块化概念所提倡的，GCC 可被用作交互式编译器；也就是说，可以使用 GCC 对所有设备与操作系统创建可执行程序，不需要局限于仅仅是运行 GCC 的平台。然而，这么做需要特殊的配置和安装，大多数 GCC 的安装，仅能针对它们的宿主系统编译程序。

GCC 不仅支持C的许多“方言”，也可以区别不同的C语言标准；也就是说，可以使用命令行选项来控制编译器在翻译源代码时应该遵循哪个C标准。

例如，当使用命令行参数`-std=c99`启动 GCC 时，编译器支持 C99 标准。GCC 对 C11 标准的支持是不完整的，尤其是涉及定义在头文件 threads.h 中的多线程函数。这是因为，GCC 的C链接库长期以来支

持 POSIX 标准下与 C11 标准非常相似的多线程功能。针对更多这方面的细节，请参考 GCC 开发者 wiki 页面：
https://gcc.gnu.org/wiki/C11Status

GCC所支持的平台

GCC 编译程序集合可以在很多平台上运行。平台是指特定计算机芯片（CPU）及其运行的操作系统的组合。

尽管 GCC 已经被移植到数以千计的平台（硬件/软件的组合）上，但只有一些基木平台可以用来测试发布的正确性。表1中列出的平台是最流行的，并且它们对 GCC 支持的最好。

表1：GCC 支持的主要平台

硬件	操作系统
Alpha	Red Hat Linux 7.1
HPPA	HPUX 11.0
Intel x86	Debian Linux 2.2、Red Hat Linux 6.2 和 FreeBSD 4.5
MIPS	IRIX 6.5
PowerPC	AIX 4.3.3
Sparc	Solaris 2.7

GCC 除了可以保证在表 1 中列出的主要平台上正确运行，也能很好地支持表 2 中列出的次要平台。

表2：GCC 支持的次要平台

硬件	操作系统
PowerPC	Linux
Sparc	Linux
ARM	Linux
Intel x86	Cygwin

以上平台都是 GCC 进行了良好测试的平台。相比数以千计的平台，这些平台显然很少，这主要是因为人力和资金问题。

即使你的平台没有被列在上面，GCC 还是可能会运行地很好。GCC 的源代码中提供了完整的测试集合，你可以在自己的平台上自行测试。

另一种方法就是作为志愿者来测试你的平台，这样 GCC 就可以在每次发布之前得到测试。

GCC的组成部分以及使用到的软件

GCC 是由许多组件组成的。表 1 列出了 GCC 的各个部分，但它们也并不总是出现 的。有些部分是和语言相关的，所以如果没有安装某种特定语言，系统:中就不会出现相关的文件。

表1：GCC 安装的各个部分

部分	描述
c++	gcc 的一个版木，默认语言设置为 C++，而且在连接的时候自动包含标准 C++ 库。这和 g++ 一样
ccl	实际的C编译程序
cclplus	实际的 C++ 编泽程序
collect2	在不使用 GNU 连接程序的系统上，有必要运行 collect2 来产生特定的全局初始化代码（例如 C++ 的构造函数和析构函数）
configure	GCC 源代码树根目录中的一个脚木。用于设置配置值和创建GCC 编译程序必需的 make 程序的描述文件
crt0.o	这个初始化和结束代码是为每个系统定制的，而且也被编译进该文件，该文件然后会被连接到每个可执行文件中来执行必要的启动和终止程序
cygwin1.dll	Windows 的共享库提供的 API，模拟 UNIX 系统调

	用
f77	该驱动程序可用于编译 Fortran
f771	实际的 Fortran 编译程序
g++	gcc 的一个版木，默认语言设置为 C++，而且在连接的时候自动包含标准 C++ 库。这和 c++ 一样
gcc	该驱动程序等同于执行编译程序和连接程序以产生需要的输出
gcj	该驱动程序用于编译 Java
gnat1	实际的 Ada 编译程序
gnatbind	一种工具，用于执行 Ada 语言绑定
gnatlink	一种工具，用于执行 Ada 语言连接
jc1	实际的 Java 编译程序
libgcc	该库包含的例程被作为编泽程序的一部分，是因为它们可被连接到实际的可执行程序中。 它们是特殊的例程，连接到可执行程序，来执行基木的任务，例如浮点运算。这些库中的例程通常都是平台相关的
libgcj	运行时库包含所有的核心 Java 类
libobjc	对所有 Objective-C 程序都必须的运行时库
libstdc++	运行时库，包括定义为标准语言一部分的所有的

	C++ 类和函数

表 2 列出的软件和 GCC 协同工作，目的是实现编译过程。有些是很基本的（例如 as 和 ld），而其他一些则是非常有用但不是严格需要的。尽管这些工具中的很多都是各种 UNIX 系统的本地共具，但还是能够通过 GNU 包 binutils 得到大多数工具。

表2：GCC 使用的软件工具

工具	描述
addr2line	给出一个可执行文件的内部地址，addr2line 使用文件中的调试信息将地址翻泽成源代码文 件名和行号。该程序是 binutils 包的一部分
ar	这是一个程序，可通过从文档中增加、删除和析取文件来维护库文件。通常使用该工具是为了创建和管理连接程序使用的目标库文档。该程序是 binutils 包的一部分
as	GNU 汇编器。实际上它是一族汇编器，因为它可以被编泽或能够在各种不同平台上工作。 该程序是 binutils 包的一部分
autoconf	产生的 shell 脚木自动配置源代码包去编泽某个特定版木的 UNIX
c++filt	程序接受被 C++ 编泽程序转换过的名字（不是被重载的），而且将该名字翻泽成初始形式。 该程序是

	binutils 包的一部分
f2c	是 Fortran 到C的翻译程序。不是 GCC 的一部分
gcov	gprof 使用的配置工具，用来确定程序运行的时候哪一部分耗时最大
gdb	GNU 调试器，可用于检查程序运行时的值和行为
GNATS	GNU 的调试跟踪系统（GNU Bug Tracking System）。一个跟踪 GCC 和其他 GNU 软件问题的在线系统
gprof	该程序会监督编泽程序的执行过程，并报告程序中各个函数的运行时间，可以根据所提供 的配置文件来优化程序。该程序是 binutils 包的一部分
ld	GNU 连接程序。该程序将目标文件的集合组合成可执行程序。该程序是 binutils 包的一部
libtool	一个基本库，支持 make 程序的描述文件使用的简化共享库用法的脚木
make	一个工具程序，它会读 makefile 脚木来确定程序中的哪个部分需要编泽和连接，然后发布 必要的命令。它读出的脚木（叫做 makefile 或 Makefile）定义了文件关系和依赖关系
nlmconv	将可重定位的目标文件转换成 NetWare 可加载模块（NetWare Loadable Module, NLM）。该 程序是 binutils 的一部分

nm	列出目标文件中定义的符号。该程序是 binutils 包的一部分
objcopy	将目标文件从一种二进制格式复制和翻译到另外一种。该程序是 binutils 包的一部分
objdump	显示一个或多个目标文件中保存的多种不同信息。该程序是 binutils 包的一部分
ranlib	创建和添加到 ar 文档的索引。该索引被 ld 使用来定位库中的模块。该程序是 binutils 包的一部分
ratfor	Ratfor 预处理程序可由 GCC 激活，但不是标准 GCC 发布版的一部分
readelf	从 ELF 格式的目标文件显示信息。该程序是 binutils 包的一部分
size	列出目标文件中每个部分的名字和尺寸。该程序是 binutils 包的一部分
strings	浏览所有类型的文件，析取出用于显示的字符串。该程序是 binutils 包的一部分
strip	从目标文件或文档库中去掉符号表，以及其他调试所需的信息。该程序是 binutils 包的一部
vcg	Ratfor 浏览器从文木文件中读取信息，并以图表形式显示它们。而 vcg 工具并不是 GCC 发布中的一部分，但 -dv 选项可被用来产生 vcg 可以理解的优化数据的格式

windres	Window 资源文件编泽程序。该程序是 binutils 包的一部分

检测是否已经安装GCC编译器

如果读者使用的是类 Unix 操作系统（Unix、Linux、Mac OS 等），很有可能已经安装好 GCC。如想知道是否如此，在命令行提示符后键入`cc --version`命令。如果已经安装好 GCC，并链接到名为 cc 的默认C编译器，就会看到编译器的版本号和版权信息：

```
$ cc --version
cc (GCC) 4.9.2
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS
FOR A PARTICULAR PURPOSE.
```

> 在本教程示例中，如果输入行前面出现美元符号（$），表示这是命令行提示符。紧接其后的是命令行，可以通过控制台输入命令，启动 GCC（或者其他程序命令名称）。

有可能已经安装好 GCC，但是并没有链接到程序名称 cc。为解决这种情况，可以试着使用 GCC 的正式名称来调用它：

```
$ gcc --version
```

安装 GCC

如果还没有安装 GCC，可以咨询系统厂商，看看是否可以提供针对所用系统软件安装机制的 GCC 安装包。一些免费软件系统，如针对 Mac OS 的 MacPorts（https://www.macports.org/）和 Homebrew（http://brew.sh/），针对 Windows 的 Cygwin（http://cygwin.org）

和 MinGW（http://www.mingw.org/）也包括 GCC 安装包。

Cygwin 是 GNU 的一个扩展套件，它在 Windows 上提供一个面向 POSIX 标准的类 Unix 环境。

Cygwin 的基础是动态共享链接库 cygwin1.dll，当与底层 Windows 系统交互时，cygwin1.dll 向 Cygwin 程序提供类 Unix 系统函数。使用 GCC 编译的针对 Cygwin 的程序，也要求运行库 cygwin1.dll。

Cygwin 安装程序（https://cygwin.com）最初只安装基本包，并加载一个包管理器，使用包管理器可以选择安装其他 Cygwin 软件，例如 gcc、make 和 gdb。你可以在任何时候运行安装程序来添加、移除和更新程序。

MinGW 也为 Windows 提供 GCC 编译器，但不同于 Gygwin，GCC 的 MinGW 版本生成 32 位 Windows 程序，而不需要特别的运行库。它的变体版本 MinGW-w64，也称为 MinGW64，能生成 64 位程序。可以通过网址 http://sourceforge.net/projects/mingw 或 http://sourceforge.net/projects/mingw-w64 获取安装程序，安装最新版本的 MinGW。

注意，Cygwin 包管理器也允许安装 MinGW-GCC 包。GCC 因此具备类似交互编译器的能力，它在 Cygwin 上运行，但是生成不使用 cygwin1.dll 的 Windows 程序。

GCC 网站上列举了由第三方针对各种不同系统制作的 GCC 安装包，可以通过访问网址 http://gcc.gnu.org/install/binaries.html 来获取它

们。此外，如果所用的操作系统已有另一个C编译器，那么可以从自由软件基金会（Free Software Foundation）获得 GCC 源代码，依据网址 http://gcc.gnu.org/install/ 上的指南一步步安装 GCC。

GCC编译C语言程序完整演示

GCC 仅仅是一个编译器，没有界面，必须在命令行模式下使用。通过`gcc`命令就可以将源文件编译成可执行文件。

GCC 既可以一次性完成C语言源文件的编译，也可以分步骤完成。本节将完整演示如何一次性完成源文件的编译（初学者也经常会这么做），下节将演示分步骤编译源文件。

本节以下面的C语言代码为例进行演示：

```
#include <stdio.h>
int main()
{
    puts("C语言中文网");
    return 0;
}
```

1) 生成可执行程序

最简单的生成可执行文件的写法为：

$ cd demo #进入源文件所在的目录
$ gcc main.c #在 gcc 命令后面紧跟源文件名

> `#`表示注释，读者可以不写，我写上是为了让读者明白每个命令的含义。`#`是 Shell 中的注释格式。

打开 demo 目录，会看到多了一个名为 a.out 的文件，这就是最终生成的可执行文件，如下图所示：

这样就一次性完成了编译和链接的全部过程，非常方便。

> 注意：不像 Windows，Linux 不以文件后缀来区分可执行文件，Linux 下的可执行文件后缀理论上可以是任意的，这里的`.out`只是用来表明它是 GCC 的输出文件。不管源文件的名字是什么，GCC 生成的可执行文件的默认名字始终是`a.out`。

如果不想使用默认的文件名，那么可以通过`-o`选项来自定义文件名，例如：

$ gcc main.c -o main.out

这样生成的可执行程序的名字就是`main.out`。

因为 Linux 下可执行文件的后缀仅仅是一种形式上的，所以可执行文件也可以不带后缀，例如：

$ gcc main.c -o main

这样生成的可执行程序的名字就是`main`。

通过`-o`选项也可以将可执行文件输出到其他目录，并不一定非得在当前目录下，例如：

$ gcc main.c -o ./out/main.out

或者

$ gcc main.c -o out/main.out

表示将可执行文件输出到当前目录下的`out`目录，并命名为`main.out`。`./`表示当前目录，如果不写，默认也是当前目录。

> 注意：out 目录必须存在，如果不存在，gcc 命令不会自动创建，而是抛出一个错误。

2) 运行可执行程序

上面我们生成了可执行程序，那么该如何运行它呢？很简单，在控制台中输入程序的名字就可以，如下所示：

```
$ ./a.out
```

`./`表示当前目录，整条命令的意思是运行当前目录下的 a.out 程序。如果不写`./`，Linux 会到系统路径下查找 a.out，而系统路径下显然不存在这个程序，所以会运行失败。

> 所谓系统路径，就是环境变量指定的路径，我们可以通过修改环境变量添加自己的路径，或者删除某个路径。很多时候，一条 Linux 命令对应一个可执行程序，如果执行命令时没有指明路径，那么就会到系统路径下查找对应的程序。

输入完上面的命令，按下回车键，程序就开始执行了，它会将输出结果直接显示在控制台上，如下所示：

```
$ cd demo
$ gcc main.c
$ ./a.out
C语言中文网
$
```

下图演示了在控制台上的实际效果：

如果程序在其它目录下，运行程序时还要带上目录的名字，例如：

$ ./out/main.out

或者

$ out/main.out

这个时候加不加`./`都一样，Linux 能够识别出`out`是一个目录，而不是一个命令，它默认会在当前路径下查找该目录，而不是去系统路径下查找，所以不加`./`也不会出错。

注意，如果程序没有执行权限，可以使用`sudo`命令来增加权限，例如：

$ sudo chmod 777 a.out

完整的演示

为了让读者有一个更加全面的认识，我们不妨将上面两部分的内容连接起来，完整的演示一下从编辑源文件到运行可执行程序的全过程：

$ cd demo #进入源文件所在目录
$ touch main.c #新建空白的源文件
$ gedit main.c #编辑源文件
$ gcc main.c #生成可执行程序
$ ./a.out #运行可执行程序
C语言中文网
$ #继续等待输入其它命令

下图是在控制台上的实际效果：

GCC分步骤编译C语言程序

上节《GCC编译C语言程序完整演示》讲解的是通过gcc命令一次性完成编译和链接的整个过程，这样最方便，大家在学习C语言的过程中一般都这么做。实际上，gcc命令也可以将编译和链接分开，每次只完成一项任务。

本节将涉及到C语言程序的生成原理，如果你不了解，请转到《C语言入门教程》学习。

本节以下面的C语言代码为例进行演示：

```
#include <stdio.h>
int main()
{
    puts("C语言中文网");
    return 0;
}
```

1) 编译（Compile）

将源文件编译成目标文件需要使用-c选项，例如：

gcc -c main.c

就将 main.c 编译为 main.o。打开 demo 目录，就会看到 main.o：

main.o

对于微软编译器（内嵌在 Visual C++ 或者 Visual Studio 中），目标文件的后缀为.obj；对于 GCC 编译器，目标文件的后缀为.o。

一个源文件会生成一个目标文件，多个源文件会生成多个目标文件，源文件数目和目标文件数目是一样的。通常情况下，默认的目标文件名字和源文件名字是一样的。

如果希望自定义目标文件的名字，那么可以使用`-o`选项，例如：

gcc -c main.c -o a.o

这样生成的目标文件的名字就是 a.o。

2) 链接（Link）

在`gcc`命令后面紧跟目标文件的名字，就可以将目标文件链接成为可执行文件，例如：

gcc main.o

就将 main.o 链接为 a.out。打开 demo 目录，就会看到 a.out。

在`gcc`命令后面紧跟源文件名字或者目标文件名字都是可以的，`gcc`命令能够自动识别到底是源文件还是目标文件：如果是源文件，那么要经过编译和链接两个步骤才能生成可执行文件；如果是目标文件，只需要链接就可以了。

使用`-o`选项仍然能够自定义可执行文件的名字，例如：

gcc main.o -o main.out

这样生成的可执行文件的名字就是 main.out。

下面是一个完整的演示：

```
$ cd demo
$ gcc -c main.c
$ gcc main.o
$ ./a.out
C语言中文网
$
```

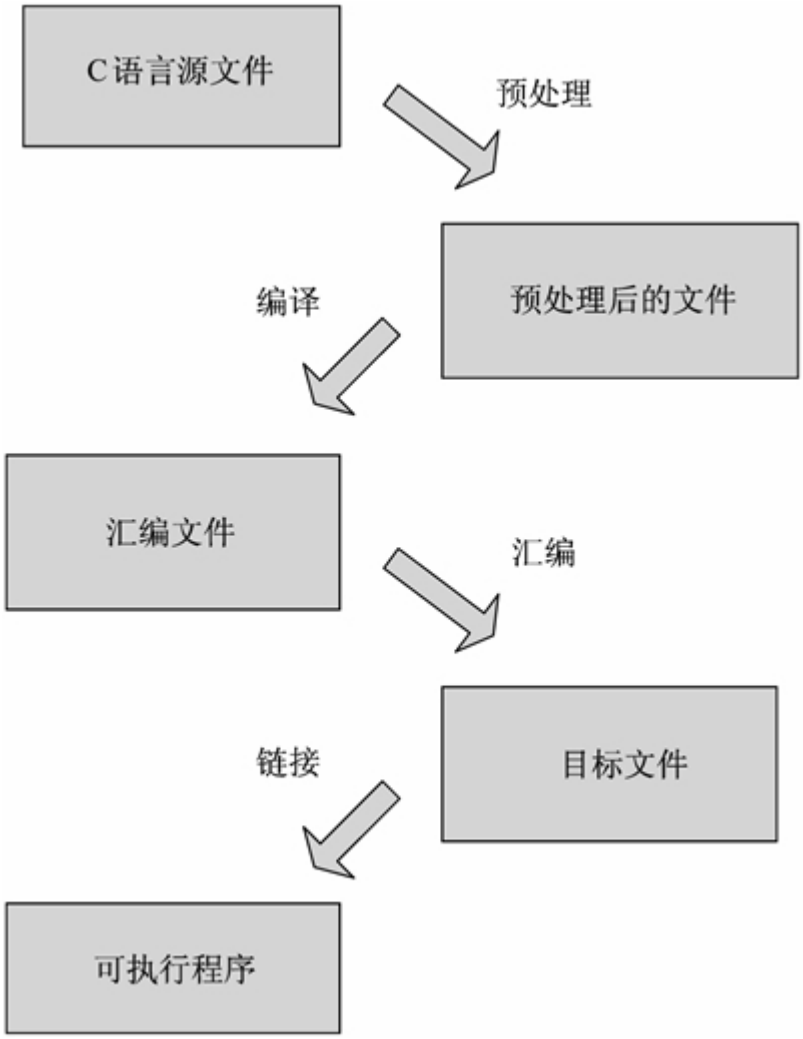
在控制台上的真实效果为：

GCC编译流程

GCC 编译器在编译一个C语言程序时需要经过以下 4 步：

1. 将C语言源程序预处理，生成`.i`文件。
2. 预处理后的.i文件编译成为汇编语言，生成`.s`文件。
3. 将汇编语言文件经过汇编，生成目标文件`.o`文件。
4. 将各个模块的`.o`文件链接起来生成一个可执行程序文件。

GCC 编译流程如下图所示：



`.i`文件、`.s`文件、`.o`文件可以认为是中间文件或临时文件，如果使用 GCC 一次性完成C语言程序的编译，那么只能看到最终的可执行文件，这些中间文件都是看不到的，因为 GCC 已经经它们删除了。

当然，可以使用 GCC 选项看到这些中间文件，下节我们会讲解 GCC 选项。

GCC常用选项

GCC 是一个功能强大的编译器，其编译选项非常多，有些选项通常不会用到。

CCC 从最初的一个试验型的玩具发展到现在 Linux 环境下的标准编译器，其命令选项也从最开始的 4 个发展到了现在的上千个。本节只介绍常用的 GCC 编译选项。

7.2.1 gcc 常用选项汇总

gcc 是一个功能强大的编译器，其编译选项非常多。有些选项一般程序员根本不会用到。因此将所有的编译选项全部列出讲解是不明智的。下面只对一些 gcc 编译器的常 用选项进行详细的讲解，这些选项在实际编程过程中非常实用。gcc 的常用选项如下表所示。

表：GCC常用的编译选项

gcc编译选项	选项的意义
-c	编译、汇编指定的源文件，但是不进行链接
-S	编译指定的源文件，但是不进行汇编
-E	预处理指定的源文件，不进行编译
-o [file1] [file2]	将文件 file2 编译成可执行文件 file1
-I directory	指定 include 包含文件的搜索目录

-g	生成调试信息，该程序可以被调试器调试

注意：gcc 编译选项会区分大小写。因此`-o`选项和`-O`选项的效果是不一样的。前者表示源文件编译成为可执行文件，后者表示将源文件编译成为可执行文件并且进行一级优化。

由于篇幅限制，本节只介绍几个简单的选项，复杂的选项会在后面几节中详细讲解。

-S

将C语言源文件编译为汇编语言，但是并不汇编该程序。使用该选项，我们可以查看C语言代码对应的汇编代码。

-E 选项

`-E`选项将C语言源文件进行预处理，但是并不编译该程序。对于一般的预处理问题，可以使用这个选项进行查看，例如，宏的展开问题、文件的包含问题等。

-I 选项

由于指定包含的头文件的目录，这一点对于大型的代码组织来说是很有用的。

-g 选项

`-g`选项可生成能被 gdb 调试器所使用的调试信息。只有使用了该选项后生成的可执行文件，才带有程序中引用的符号表。这时 gdb 调试程序才能对该可执行程序进行调试。

还有另一个 GCC 选项，可以方便地一次获得全部的中间输出文件，这就是-save-temps。当使用该选项时，GCC 会正常地编译和链接，但是会把预处理器输出、汇编语言和对象文件全部存储在当前目录下。使用 -save-temps 选项所生成的中间文件，与对应的源文件具有相同的文件名，但文件扩展名分别为.i、.s和.o，分别表示为预处理输出、汇编语言输出和对象文件。

GCC -c选项：只编译不链接，仅生成目标文件

`-c`选项表示编译、汇编指定的源文件（也就是编译源文件），但是不进行链接。使用`-c`选项可以将每一个源文件编译成对应的目标文件。

目标文件是一种中间文件或者临时文件，如果不设置该选项，gcc 一般不会保留目标文件，可执行文件生成完成后就自动删除了。

下面实例演示了 gcc -c 选项的用法。

```
$gcc -c test1.c test2.c test3.c
$ls -l *.o
-rwxr--r--  1  root  23  Feb  7  02:57  test1.o
-rwxr--r--  1  root  17  Feb  7  02:57  test2.o
-rwxr--r--  1  root  20  Feb  7  02:57  test3.o
```

如果不使用`-c`选项，则仅仅生成一个可执行文件，没有目标文件。

注意，使用`-c`选项表示只编译源文件，而不进行链接，因此，对于链接中的错误是无法发现的。

下面例子演示了 gcc 编译器在使用`-c`选项的时候不会发现链接错误。

1) 编写如下的两个源文件。

在 func.c 中定义了 func_a() 函数：

```
#include <stdio.h>
void func_a(){
    printf("FUNC_A\n");
}
```

在 main.c 中调用了 func_a() 和 func_b() 函数：

```
#include <stdio.h>
int main(void)
{
    func_a();
    func_b();
    return 0;
}
```

func_b() 函数并没有定义，所以在链接时会产生错误（编译时不会产生错误）。

2) 使用`-c`选项编译两个源文件，如下所示：

$gcc -c func.c main.c

编译器没有输出任何错误信息。

3) 不使用`-c`选项编译两个源文件：

$gcc func.c main.c

会看到如下的报错信息：

/tmp/ccLlOhvh.o：在函数‘main’中：
main.c:(.text+0x14)：对‘func_b’未定义的引用

collect2: 错误：ld 返回 1
由于没有找到 func_b() 函数的定义，所以发生了链接错误。

GCC -o选项：指定输出文件

GCG `-o`选项用来指定输出文件，它的用法为：

[infile] -o [outfile]

[infile] 表示输入文件（也即要处理的文件），它可以是源文件，也可以是汇编文件或者是目标文件；[outfile] 表示输出文件（也即处理的结果），它可以是预处理文件、目标文件、可执行文件等。

[infile] 和 [outfile] 可以是一个文件，也可以是一组文件：

- 如果 [infile] 是一组文件，那么就表示有多个输入文件；
- 如果 [outfile] 是一组文件，那么就表示有多个输出文件。

如果不使用 -o 选项，那么将采用默认的输出文件，例如，把可执行文件作为输出文件，它的名字默认为 a.out。

GCC -o选项使用举例

1) 将源文件作为输入文件，将可执行文件作为输出文件，也即完整地编译整个程序：

$ gcc main.c func.c -o app.out

将 main.c 和 func.c 两个源文件编译成一个可执行文件，其名字为 app.out。如果不使用 -o 选项，那么将生成名字为 a.out 的可执行文件。

2) 将源文件作为输入文件，将目标文件作为输出文件，也即只编译不链接：

$ gcc -c main.c -o a.o

将源文件 main.c 编译为目标文件 a.o。如果不使用 -o 选项，那么将生成名为 main.o 的目标文件。

3) 将源文件作为输入文件，将预处理文件作为输出文件，也即只进行预处理操作：

$ gcc -E main.c -o demo.i

对源文件 main.c 进行预处理操作，并将结果放在 demo.i 文件中。如果不使用 -o 选项，那么将生成名为 main.i 的预处理文件。

4) 将目标文件作为输入文件，将可执行文件作为输出文件：

$ gcc -c func.c main.c
$ gcc func.o main.o -o app.out

第一条命令只编译不链接，将生成 func.o 和 main.o 两个目标文件。第二条命令将生成的两个目标文件生成最终的可执行文件 app.out。如果不使用 -o 选项，那么将生成名字为 a.out 的可执行文件。

GCC -E选项：生成预处理文件

C语言代码在交给编译器之前，会先由预处理器进行一些文本替换方面的操作，例如宏展开、文件包含、删除部分代码等。

在正常的情况下，GCC 不会保留预处理阶段的输出文件，也即`.i`文件。然而，可以利用`-E`选项保留预处理器的输出文件，以用于诊断代码。`-E`选项指示 GCC 在预处理完毕之后即可停止。

默认情况下，预处理器的输出会被导入到标准输出流（也就是显示器），可以利用`-o`选项把它导入到某个输出文件：

```
$ gcc -E circle.c -o circle.i
```

表示把预处理的结果导出到 circle.i 文件。

因为头文件可能相当大，如果源文件包括了多个头文件，那么它的预处理器输出可能会庞杂难读。使用`-C`选项会很有帮助，这个选项可以阻止预处理器删除源文件和头文件中的注释：

```
$ gcc -E -C circle.c -o circle.c
```

> 注意，这里是大写的 -C，不是小写的 -c。小写的 -c 表示只编译不链接。

下面是 GCC 预处理器阶段常用的选项：

-Dname[=definition]

在处理源文件之前，先定义宏 name。宏 name 必须是在源文件和头文件中都没有被定义过的。将该选项搭配源代码中的`#ifdef name`命令使用，可以实现条件式编译。如果没有指定一个替换的值，该宏被定义为值 1。

-Uname

如果在命令行或 GCC 默认设置中定义过宏 name，则“取消”name 的定义。`-D`和`-U`选项会依据在命令行中出现的先后顺序进行处理。

-Idirectory[:directory[...]]

当通过 #include 命令把所需的头文件包括进源代码中时，除系统标准 include 目录之外，指定其他的目录对这些头文件进行搜索。

-iquote directory[:directory[...]]

这是在最近 GCC 版本中新增的选项，它为在 #include 命令中采用引号而非尖括号指定的头文件指定搜索目录。

-isystem directory[:directory[...]]

该选项在标准系统 include 目录以外为系统头文件指定搜索目录，且它指定的目录优先于标准系统 include 目录被搜索。在目录说明开头位置的等号，被视作系统根目录的占位符，可以使用`--sysroot`或`-isysroot`选项来修改它。

-isysroot directory

该选项指定搜索头文件时的系统根目录。例如，如果编译器通常在 /usr/include 目录及其子目录下搜索系统头文件，则该选项将引导到 directory/usr/include 及其子目录下进行搜索。

> `--sysroot`选项，采用一个连字符替代 i，它为链接库搜索而不是头文件搜索指定系统根目录以外的目录。如果 isysroot 不可用，则 sysroot 既为头文件又为链接库搜索指定目录。

-I-

在较新版本的 GCC 中，该选项被`-iquote`替代。在旧版本中，该选项用于将命令行的所有`-Idirectory`选项分割为两组。所有在`-I-`左边加上`-I`选项的目录，被视为等同于采用`-iquote`选项；这指的是，它们只对 #include 命令中采用引号的头文件名进行搜索。

所有在`-I-`右边加上`-I`选项的目录，将对所有 #include 命令中的头文件名进行搜索，无论文件名是在引号还是尖括号中。

而且，如果命令行中出现了`-I-`，那么包括源文件本身的目录不再自动作为搜索头文件的目录。

对于include目录而言，通常的搜索顺序是：

1. 包含指定源文件的目录（对于在 #include 命令中以引号包括的文件名）。
2. 采用`-iquote`选项指定的目录，依照出现在命令行中的顺序进行搜索。只对 #include 命令中采用引号的头文件名进行搜索。
3. 采用`-I`选型指定的目录，依照出现在命令行中的顺序进行搜索。
4. 采用环境变量 CPATH 指定的目录。
5. 采用`-isystem`选项指定的目录，依照出现在命令行中的顺序进行搜索。
6. 采用环境变量 C_INCLUDE_PATH 指定的目录。

7. 系统默认的 include 目录。

GCC -S选项：生成汇编文件

编译器的核心任务是把C程序翻译成机器的汇编语言（assembly language）。汇编语言是人类可以阅读的编程语言，也是相当接近实际机器码的语言。由此导致每种 CPU 架构都有不同的汇编语言。

> 实际上，GCC 是一个适合多种 CPU 架构的编译器，不会把C程序语句直接翻译成目标机器的汇编语言，而是在输入语言和输出汇编语言之间，利用一个中间语言，称为 RegisterTransfer Language（简称 RTL，寄存器传输语言）。借助于这个抽象层，在任何背景下，编译器可以选择最经济的方式对给定的操作编码。
>
> 而且，在交互文件中针对目标机器的抽象描述，为编译器重新定向到新架构提供了一个结构化的方式。但是，从 GCC 用户角度来看，我们可以忽略这个中间步骤。

通常情况下，GCC 把汇编语言输出存储到临时文件中，并且在汇编器执行完后立刻删除它们。但是可以使用`-S`选项，让编译程序在生成汇编语言输出之后立刻停止。

如果没有指定输出文件名，那么采用`-S`选项的 GCC 编译过程会为每个被编译的输入文件生成以`.s`作为后缀的汇编语言文件。如下例所示：

```
$ gcc -S circle.c
```

编译器预处理 circle.c，将其翻译成汇编语言，并将结果存储在 circle.s 文件中。

如果想把C语言变量的名称作为汇编语言语句中的注释，可以加上`-fverbose-asm`选项：

```
$ gcc -S -fverbose-asm circle.c
```

GCC -l选项：手动添加链接库

链接器把多个二进制的目标文件（object file）链接成一个单独的可执行文件。在链接过程中，它必须把符号（变量名、函数名等一些列标识符）用对应的数据的内存地址（变量地址、函数地址等）替代，以完成程序中多个模块的外部引用。

而且，链接器也必须将程序中所用到的所有C标准库函数加入其中。对于链接器而言，链接库不过是一个具有许多目标文件的集合，它们在一个文件中以方便处理。

> 当把程序链接到一个链接库时，只会链接程序所用到的函数的目标文件。在已编译的目标文件之外，如果创建自己的链接库，可以使用 ar 命令。

标准库的大部分函数通常放在文件 libc.a 中（文件名后缀`.a`代表“achieve”，译为“获取”），或者放在用于共享的动态链接文件 libc.so 中（文件名后缀`.so`代表“share object”，译为“共享对象”）。这些链接库一般位于 /lib/ 或 /usr/lib/，或者位于 GCC 默认搜索的其他目录。

当使用 GCC 编译和链接程序时，GCC 默认会链接 libc.a 或者 libc.so，但是对于其他的库（例如非标准库、第三方库等），就需要手动添加。

令人惊讶的是，标准头文件 <math.h> 对应的数学库默认也不会被链接，如果没有手动将它添加进来，就会发生函数未定义错误。

GCC 的`-l`选项可以让我们手动添加链接库。下面我们编写一个数学程序 main.c，并使用到了 cos() 函数，它位于 <math.h> 头文件。

```
#include <stdio.h>      /* printf */
#include <math.h>       /* cos */
#define PI 3.14159265
int main ()
{
    double param, result;
    param = 60.0;
    result = cos ( param * PI / 180.0 );
    printf ("The cosine of %f degrees is %f.\n", param, result
);
    return 0;
}
```

为了编译这个 main.c，必须使用`-l`选项，以链接数学库：

$ gcc main.c -o main.out -lm

数学库的文件名是 libm.a。前缀`lib`和后缀`.a`是标准的，`m`是基本名称，GCC 会在`-l`选项后紧跟着的基本名称的基础上自动添加这些前缀、后缀，本例中，基本名称为 m。

> 在支持动态链接的系统上，GCC 自动使用在 Darwin 上的共享链接库 libm.so 或 libm.dylib。

链接其它目录中的库

通常，GCC 会自动在标准库目录中搜索文件，例如 /usr/lib，如果想链接其它目录中的库，就得特别指明。有三种方式可以链接在 GCC 搜索路径以外的链接库，下面我们分别讲解。

1) 把链接库作为一般的目标文件，为 GCC 指定该链接库的完整路径

与文件名。

例如，如果链接库名为 libm.a，并且位于 /usr/lib 目录，那么下面的命令会让 GCC 编译 main.c，然后将 libm.a 链接到 main.o：

$ gcc main.c -o main.out /usr/lib/libm.a

2) 使用`-L`选项，为 GCC 增加另一个搜索链接库的目录：

$ gcc main.c -o main.out -L/usr/lib -lm

可以使用多个`-L`选项，或者在一个`-L`选项内使用冒号分割的路径列表。

3) 把包括所需链接库的目录加到环境变量 LIBRARYPATH 中。

GCC编译和链接多个文件（包括源文件、目标文件、汇编文件等）

编译多个源代码文件会生成多个目标文件，每个目标文件都包含一个源文件的机器码和相关数据的符号表。除非使用`-c`选项指示 GCC 只编译不链接，否则 GCC 会使用临时文件作为目标文件输出：

$ gcc -c main.c
$ gcc -c func.c

这些命令会在当前目录中生成两个目标文件，分别是 main.o 和 func.o。把两个源文件名放在同一个 GCC 命令中，也可以获得同样的结果：

$ gcc -c main.c func.c

然而，实际上编译器通常每次只会被调用来完成一件小型任务。大的程序包含许多源文件，在开发期间必须被编译、测试、编辑，然后再编译，很少会有在创建中的修改行为会影响所有的源文件。为了节省时间，可以使用 make 控制创建过程，由它调用编译器重新编译，而且只编译比对应的最新源文件旧的那些目标文件。

一旦所有当前源文件都被编译为目标文件，就可以使用 GCC 来链接它们：

$ gcc main.o func.o -o app.out -lm

GCC 假设扩展名为`.o`的文件是要被链接的目标文件。

文件类型

编译器支持许多和C语言程序相关的扩展名，对它们的说明如下：

扩展名（后缀）	说明
.c	C程序源代码，在编译之前要先进行预处理。
.i	C程序预处理输出，可以被编译。
.h	C程序头文件。（为了节省时间，许多源文件会包含相同的头文件，GCC 允许事先编译好头文件，称为“预编译头文件”，它合适情况下自动被用于编译。）
.s	汇编语言。
.S	有C命令的汇编语言，在汇编之前必须先进行预处理。

GCC 也支持以下扩展名：.ii、.cc、.cp、.cxx、.cpp、.CPP、.c++、.C、.hh、.H、.m、.mi、.f、.for、.FOR、.F、.fpp、.FPP、.r、.ads 和 .adb。这些文件类型涉及 C++、Objective-C、Fortran 或 Ada 等程序的编译。使用其他扩展名的文件会被视为对象文件，以作为链接之用。

如果使用其他命名惯例为输入文件命名，可以使用`-x file_type`选项来指示 GCC 应该如何对待这些文件。file_type 必须是下面其中之一：c、c-header、cpp-output、assembler（表示该文件包含汇编语言）、assembler-with-cpp 或 none。

在命令行中，`-x`后面所列的所有文件都会被视为所指定的类型。如果想改变类型，可以再次使用`-x`。如下例所示：

$ gcc -o bigprg mainpart.c -x assembler trickypart.asm -x c otherpart.c

可以在同一个命令行中多次使用`-x`选项，以指示不同类型的文件。`-x none`选项会全部取消这些指示，后续文件会按照 GCC 默认规则解释它们的扩展名。

混合输入类型

可以在 GCC 命令行中混合使用其他输入文件类型。如果编译器不能按照请求处理指定的文件，那么它会忽略它们。如下例所示：

$ gcc main.c func.o -o app.out -lm
借助于这个命令，假设指定的文件都存在，GCC 会编译和汇编 main.c，汇编 func.o，并且链接库文件 libm.a。

GCC生成动态链接库（.so文件）：-shared和-fPIC选项

Linux 下动态链接库（shared object file，共享对象文件）的文件后缀为`.so`，它是一种特殊的目标文件（object file），可以在程序运行时被加载（链接）进来。使用动态链接库的优点是：程序的可执行文件更小，便于程序的模块化以及更新，同时，有效内存的使用效率更高。

GCC 生成动态链接库

如果想创建一个动态链接库，可以使用 GCC 的`-shared`选项。输入文件可以是源文件、汇编文件或者目标文件。

另外还得结合`-fPIC`选项。-fPIC 选项作用于编译阶段，告诉编译器产生与位置无关代码（Position-Independent Code）；这样一来，产生的代码中就没有绝对地址了，全部使用相对地址，所以代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。

例如，从源文件生成动态链接库：

```
$ gcc -fPIC -shared func.c -o libfunc.so
```

从目标文件生成动态链接库：

```
$ gcc -fPIC -c func.c -o func.o
$ gcc -shared func.o -o libfunc.so
```

-fPIC 选项作用于编译阶段，在生成目标文件时就得使用该选项，以生成位置无关的代码。

GCC 将动态链接库链接到可执行文件

如果希望将一个动态链接库链接到可执行文件，那么需要在命令行中列出动态链接库的名称，具体方式和普通的源文件、目标文件一样。请看下面的例子：

```
$ gcc main.c libfunc.so -o app.out
```

将 main.c 和 libfunc.so 一起编译成 app.out，当 app.out 运行时，会动态地加载链接库 libfunc.so。

当然，必须要确保程序在运行时可以找到这个动态链接库。你可以将链接库放到标准目录下，例如 /usr/lib，或者设置一个合适的环境变量，例如 LIBRARY_PATH。不同系统，具有不同的加载链接库的方法。