**编译器的工作过程**

作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2014年11月11日](http://www.ruanyifeng.com/blog/2014/11/)

源码要运行，必须先转成二进制的机器码。这是编译器的任务。

比如，下面这段源码（假定文件名叫做test.c）。

#include <stdio.h>

int main(void)

{

fputs("Hello, world!\n", stdout);

return 0;

}

要先用编译器处理一下，才能运行。

$ gcc test.c

$ ./a.out

Hello, world!

对于复杂的项目，编译过程还必须分成三步。

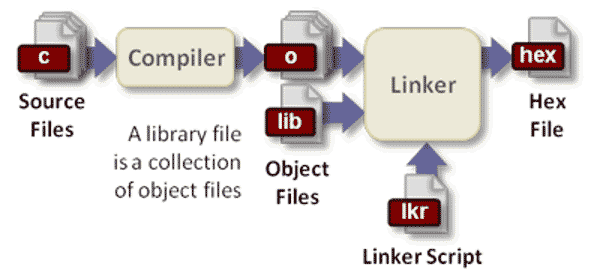
$ ./configure

$ make

$ make install

这些命令到底在干什么？大多数的书籍和资料，都语焉不详，只说这样就可以编译了，没有进一步的解释。

本文将介绍编译器的工作过程，也就是上面这三个命令各自的任务。我主要参考了Alex Smith的文章[《Building C Projects》](http://nethack4.org/blog/building-c.html)。需要声明的是，本文主要针对gcc编译器，也就是针对C和C++，不一定适用于其他语言的编译。



**第一步 配置（configure）**

编译器在开始工作之前，需要知道当前的系统环境，比如标准库在哪里、软件的安装位置在哪里、需要安装哪些组件等等。这是因为不同计算机的系统环境不一样，通过指定编译参数，编译器就可以灵活适应环境，编译出各种环境都能运行的机器码。这个确定编译参数的步骤，就叫做"配置"（configure）。

这些配置信息保存在一个配置文件之中，约定俗成是一个叫做configure的脚本文件。通常它是由[autoconf](http://zh.wikipedia.org/wiki/Autoconf)工具生成的。编译器通过运行这个脚本，获知编译参数。

configure脚本已经尽量考虑到不同系统的差异，并且对各种编译参数给出了默认值。如果用户的系统环境比较特别，或者有一些特定的需求，就需要手动向configure脚本提供编译参数。

$ ./configure --prefix=/www --with-mysql

上面代码是php源码的一种编译配置，用户指定安装后的文件保存在www目录，并且编译时加入mysql模块的支持。

**第二步 确定标准库和头文件的位置**

源码肯定会用到标准库函数（standard library）和头文件（header）。它们可以存放在系统的任意目录中，编译器实际上没办法自动检测它们的位置，只有通过配置文件才能知道。

编译的第二步，就是从配置文件中知道标准库和头文件的位置。一般来说，配置文件会给出一个清单，列出几个具体的目录。等到编译时，编译器就按顺序到这几个目录中，寻找目标。

**第三步 确定依赖关系**

对于大型项目来说，源码文件之间往往存在依赖关系，编译器需要确定编译的先后顺序。假定A文件依赖于B文件，编译器应该保证做到下面两点。

（1）只有在B文件编译完成后，才开始编译A文件。

（2）当B文件发生变化时，A文件会被重新编译。

编译顺序保存在一个叫做makefile的文件中，里面列出哪个文件先编译，哪个文件后编译。而makefile文件由configure脚本运行生成，这就是为什么编译时configure必须首先运行的原因。

在确定依赖关系的同时，编译器也确定了，编译时会用到哪些头文件。

**第四步 头文件的预编译（precompilation）**

不同的源码文件，可能引用同一个头文件（比如stdio.h）。编译的时候，头文件也必须一起编译。为了节省时间，编译器会在编译源码之前，先编译头文件。这保证了头文件只需编译一次，不必每次用到的时候，都重新编译了。

不过，并不是头文件的所有内容，都会被预编译。用来声明宏的#define命令，就不会被预编译。

**第五步 预处理（Preprocessing）**

预编译完成后，编译器就开始替换掉源码中bash的头文件和宏。以本文开头的那段源码为例，它包含头文件stdio.h，替换后的样子如下。

extern int fputs(const char \*, FILE \*);

extern FILE \*stdout;

int main(void)

{

fputs("Hello, world!\n", stdout);

return 0;

}

为了便于阅读，上面代码只截取了头文件中与源码相关的那部分，即fputs和FILE的声明，省略了stdio.h的其他部分（因为它们非常长）。另外，上面代码的头文件没有经过预编译，而实际上，插入源码的是预编译后的结果。编译器在这一步还会移除注释。

这一步称为"预处理"（Preprocessing），因为完成之后，就要开始真正的处理了。

**第六步 编译（Compilation）**

预处理之后，编译器就开始生成机器码。对于某些编译器来说，还存在一个中间步骤，会先把源码转为汇编码（assembly），然后再把汇编码转为机器码。

下面是本文开头的那段源码转成的汇编码。

.file "test.c"

.section .rodata

.LC0:

.string "Hello, world!\n"

.text

.globl main

.type main, @function

main:

.LFB0:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

movq stdout(%rip), %rax

movq %rax, %rcx

movl $14, %edx

movl $1, %esi

movl $.LC0, %edi

call fwrite

movl $0, %eax

popq %rbp

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

.LFE0:

.size main, .-main

.ident "GCC: (Debian 4.9.1-19) 4.9.1"

.section .note.GNU-stack,"",@progbits

这种转码后的文件称为对象文件（object file）。

**第七步 连接（Linking）**

对象文件还不能运行，必须进一步转成可执行文件。如果你仔细看上一步的转码结果，会发现其中引用了stdout函数和fwrite函数。也就是说，程序要正常运行，除了上面的代码以外，还必须有stdout和fwrite这两个函数的代码，它们是由C语言的标准库提供的。

编译器的下一步工作，就是把外部函数的代码（通常是后缀名为.lib和.a的文件），添加到可执行文件中。这就叫做连接（linking）。这种通过拷贝，将外部函数库添加到可执行文件的方式，叫做[静态连接](http://zh.wikipedia.org/zh-cn/%E9%9D%9C%E6%85%8B%E9%80%A3%E7%B5%90%E5%87%BD%E5%BC%8F%E5%BA%AB)（static linking），后文会提到还有[动态连接](http://zh.wikipedia.org/wiki/%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5%E5%BA%93)（dynamic linking）。

make命令的作用，就是从第四步头文件预编译开始，一直到做完这一步。

**第八步 安装（Installation）**

上一步的连接是在内存中进行的，即编译器在内存中生成了可执行文件。下一步，必须将可执行文件保存到用户事先指定的安装目录。

表面上，这一步很简单，就是将可执行文件（连带相关的数据文件）拷贝过去就行了。但是实际上，这一步还必须完成创建目录、保存文件、设置权限等步骤。这整个的保存过程就称为"安装"（Installation）。

**第九步 操作系统连接**

可执行文件安装后，必须以某种方式通知操作系统，让其知道可以使用这个程序了。比如，我们安装了一个文本阅读程序，往往希望双击txt文件，该程序就会自动运行。

这就要求在操作系统中，登记这个程序的元数据：文件名、文件描述、关联后缀名等等。Linux系统中，这些信息通常保存在/usr/share/applications目录下的.desktop文件中。另外，在Windows操作系统中，还需要在Start启动菜单中，建立一个快捷方式。

这些事情就叫做"操作系统连接"。make install命令，就用来完成"安装"和"操作系统连接"这两步。

**第十步 生成安装包**

写到这里，源码编译的整个过程就基本完成了。但是只有很少一部分用户，愿意耐着性子，从头到尾做一遍这个过程。事实上，如果你只有源码可以交给用户，他们会认定你是一个不友好的家伙。大部分用户要的是一个二进制的可执行程序，立刻就能运行。这就要求开发者，将上一步生成的可执行文件，做成可以分发的安装包。

所以，编译器还必须有生成安装包的功能。通常是将可执行文件（连带相关的数据文件），以某种目录结构，保存成压缩文件包，交给用户。

**第十一步 动态连接（Dynamic linking）**

正常情况下，到这一步，程序已经可以运行了。至于运行期间（runtime）发生的事情，与编译器一概无关。但是，开发者可以在编译阶段选择可执行文件连接外部函数库的方式，到底是静态连接（编译时连接），还是动态连接（运行时连接）。所以，最后还要提一下，什么叫做动态连接。

前面已经说过，静态连接就是把外部函数库，拷贝到可执行文件中。这样做的好处是，适用范围比较广，不用担心用户机器缺少某个库文件；缺点是安装包会比较大，而且多个应用程序之间，无法共享库文件。动态连接的做法正好相反，外部函数库不进入安装包，只在运行时动态引用。好处是安装包会比较小，多个应用程序可以共享库文件；缺点是用户必须事先安装好库文件，而且版本和安装位置都必须符合要求，否则就不能正常运行。

现实中，大部分软件采用动态连接，共享库文件。这种动态共享的库文件，Linux平台是后缀名为.so的文件，Windows平台是.dll文件，Mac平台是.dylib文件。

（文章完）