# [Autotool](https://segmentfault.com/a/1190000006915719)使用

# GNU构建系统和autotools

GNU开源软件的安装三部曲configure->make->make install，这就是GNU构建系统。

这个过程中， 首先需要有一个 configure 脚本，用来生成 Makefile 文件，然后才能使用Makefile文件进行代码的编译。

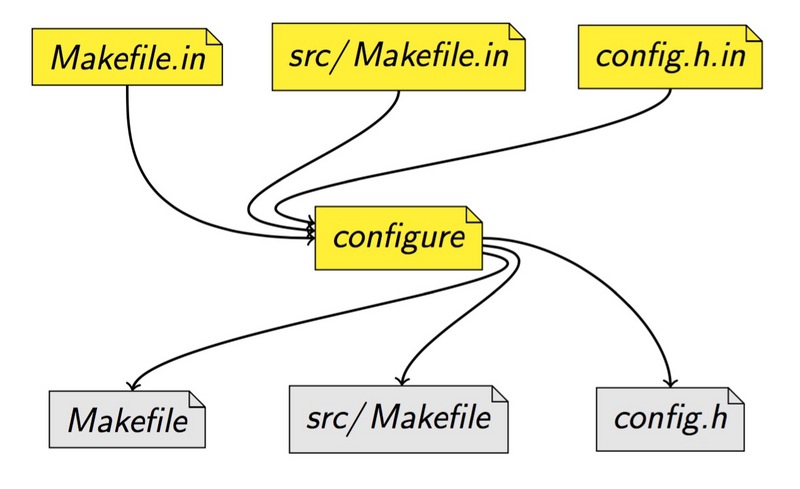
那么为什么要使用configure文件去生成Makefile文件？

最早的时候，程序员完成源代码开发以后，代码包里面一般会附带相应的 Makefile 文件。然后就可以 make && make install 来编译工程。当时并不需要这个运行 configure 的步骤。

但是如果一个程序被用的多了以后 ，可能需要被安装到不同的平台上使用。这个时候，在不同的平台上编译时，一方面可能需要对 Makefile 文件进行调整 (最常见的例子就是：不同的平台使用不同的编译器)。另外一方面，可能需要用一个替代函数来替换当前平台所不支持的函数 (例如：有的平台上不支持strdup这个调用)，需要在程序里面给每个平台写#define。

为了避免手工做这些调整，人们开始写 configure 脚本来自动做这些调整工作 (现在在 make 之前先运行 configure 是 GNU Code Style 标准所规定的)。

configure 脚本一般会先检查目前的环境，然后根据代码里的各级Makefile.in和config.h.in文件生成一个config.h 文件 (里面带了各种各样的#define) ，同时会生成一个 针对当前平台的 Makefile 文件，之后，make 命令就会使用到这个 Makefile文件。



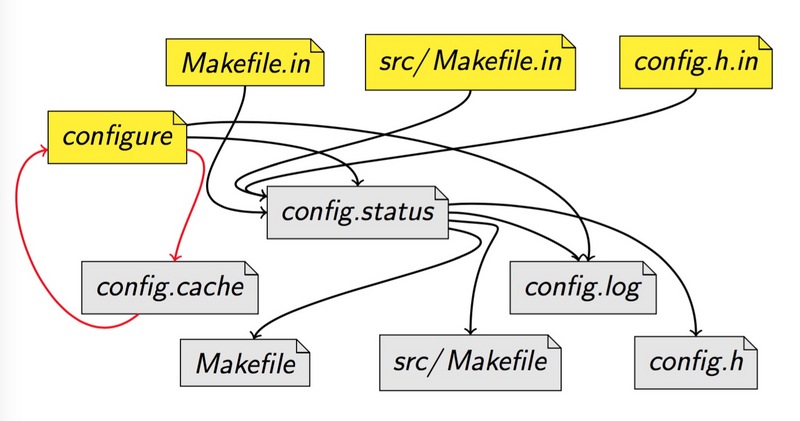
另外，GNU的 build 系统还有一些的功能，用户在使用 configure 这个脚本的时候，可能会使用到这些功能 (最常见的就是用 --prefix 来指定安装路径， --build指定编译平台，--host用来指定目标平台，用configure --help来查看说明等等)。

configure在执行过程中，除了生成Makefile外，还会生成的文件包括但不限于：

config.log 日志文件

config.cache 缓存，以提高下一次configure的速度，需通过-C来指定才会生成

config.status 实际调用编译工具构建软件的shell脚本



但是，过了一段时间以后，人们发现靠人手工写这个 configure 脚本工作量太巨大了，而且以后维护这个 configure也比较麻烦：一旦发现项目在某个平台的移植性有问题，就需要更新这个 configure 脚本，比较繁琐。于是，人们就开发了 Autotools 这个工具集来自动生成 configure 和Makefile脚本。

GNU Autotools 一般指的是3个 GNU 工具包：Autoconf，Automake 和 Libtool (本文先介绍前两个工具，Libtool留到今后介绍)。

# 使用流程

安装Autoconf和Automake工具包：yum install autoconf，yum install automake，包括以下工具:

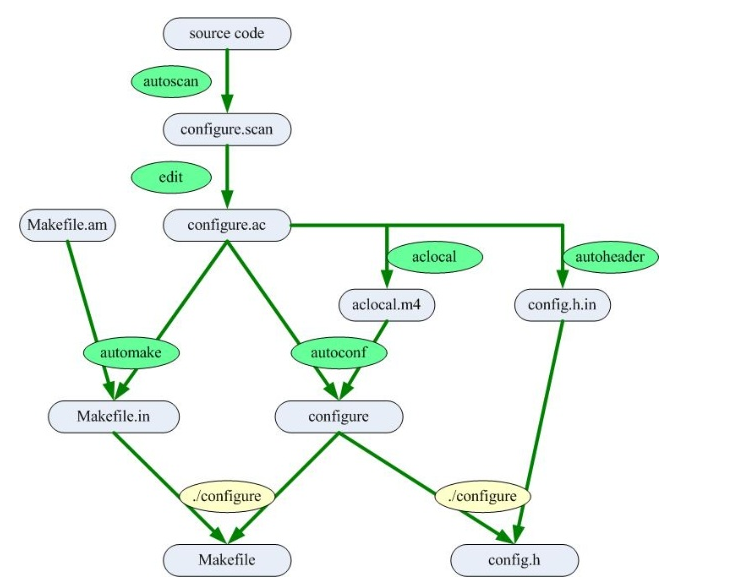
aclocal

autoscan

autoheader

automake

autoconf



简单总结一下这张图，分为八步：

# 运行autoscan命令，将configure.scan 文件重命名为configure.ac

# 运行aclocal命令，生成aclocal.m4

# 运行autoconf命令，生成configure

# 运行autoheader,生成config.h.in

# 创建Makefile.am，修改

# 运行automake -a命令

# 运行./confiugre脚本

1、**autoscan**

这个命令主要用于扫描工作目录，并且生成configure.scan文件。并且configure.scan需要重命令成configure.ac，然后编辑这个配置

**configure.ac标签说明**

|  |  |
| --- | --- |
| 标签 | 说明 |
| AC\_PREREQ | 声明autoconf要求的版本号 |
| AC\_INIT | 定义软件名称、版本号、联系方式 |
| AM\_INIT\_AUTOMAKE | 必须要的，参数为软件名称和版本号 |
| AC\_CONFIG\_SCRDIR | 宏用来侦测所指定的源码文件是否存在, 来确定源码目录的有效性.。此处为当前目录下main.c。 |
| AC\_CONFIG\_HEADER | 生成config.h文件保存configure.ac定义的宏，此文件可被源文件包含，以便 autoheader 命令使用。 |
| AC\_PROG\_CC | 指定编译器，默认GCC |
| AC\_CONFIG\_FILES | 运行configure后生成的config脚本需要处理的文件，将Makefile.in变成Makefile  生成相应的Makefile文件，不同文件夹下的Makefile通过空格分隔。例如：AC\_CONFIG\_FILES([Makefile, src/Makefile]) |
| AC\_OUTPUT | 用来设定 configure 所要产生的文件，如果是makefile，configure 会把它检查出来的结果带入makefile.in文件产生合适的makefile。 |

2、aclocal

扫描 configure.ac 文件生成 aclocal.m4文件, 该文件主要处理本地的宏定义，它根据已经安装的宏、用户定义宏和 acinclude.m4 文件中的宏将 configure.ac 文件需要的宏集中定义到文件 aclocal.m4 中(aclocal是一个perl 脚本程序).

3、autoconf

执行autoconf命令。这个命令将 configure.ac 文件中的宏展开，生成 configure 脚本。这个过程要用到aclocal.m4中定义的宏

1. Autoheader

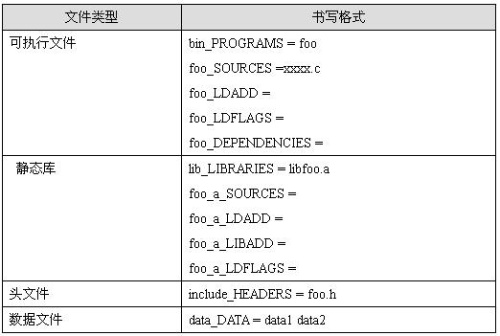
执行autoheader命令。该命令生成 config.h.in 文件。该命令通常会从 "acconfig.h” 文件中复制用户附加的符号定义。

1. **Makefile.am**

创建Makefile.am文件。Automake工具会根据 configure.in 中的参量把 Makefile.am 转换成 Makefile.in 文件。最终通过Makefile.in生成Makefile文件，所以Makefile.am这个文件非常重要，定义了一些生成Makefile的规则。

Makefile.am是一种比Makefile更高层次的规则。只需指定要生成什么目标，它由什么源文件生成，要安装到什么目录等构成。

表一列出了可执行文件、静态库、头文件和数据文件，四种书写Makefile.am文件个一般格式。

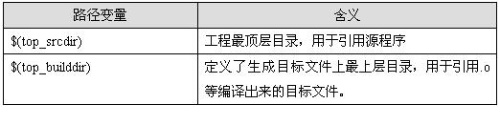


对于可执行文件和静态库类型，如果只想编译，不想安装到系统中，可以用noinst\_PROGRAMS代替bin\_PROGRAMS，noinst\_LIBRARIES代替lib\_LIBRARIES。

Makefile.am还提供了一些全局变量供所有的目标体使用：

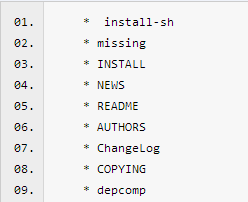


在Makefile.am中尽量使用相对路径，系统预定义了两个基本路径：

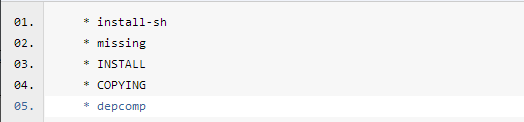


1. automake: automake将Makefile.am中定义的结构建立Makefile.in

Automake 所需要的文件：



用automake –a可以自动生成以下文件



要手动生成以下文件

touch NEWS README AUTHORS ChangeLog

1. ****./configure****   
    生成Makefile和config.h
2. Make

生成目标文件

### Libtool

使用 GNU Libtool 可以容易的在不同的系统中建立动态链接库。它通过一个称为 Libtool 库的抽象，隐藏了不同系统之间的差异，给开发人员提供了一致的的接口。对于大部分情况，开发人员甚至不用去查看相应的系统手册，只需要掌握 GNU Libtool 的用法就可以了。并且，使用 Libtool 的 Makefile 也只需要编写一次就可以在多个系统上使用。

Libtool 库可以是一个静态链接库，可以是一个动态链接库，也可以同时包含两者。在这篇文档中，我们围绕 Libtool 库的建立和使用，只是在适当的说明 Libtool 库和系统动态或者静态链接库之间的映射关系。

##### **Libtool 是一个工具**

虽然 Libtool 隐藏了在不同平台创建链接库的复杂性，但其最终还是需要底层系统对链接库的支持，它不能超越系统的限制，例如，Libtool 并不能在不支持动态链接库的系统中创建出动态链接库。

#### **Libtool 基本用法**

这一节以实例来说明如何使用 Libtool 从源代码创建最终链接库以及执行程序的完整步骤，这是软件开发过程中经常使用的内容，包括 :

* 创建 Libtool 对象文件 ;
* 创建 Libtool 库；
* 安装 Libtool 库 ;
* 使用 Libtool 库 ;
* 卸载 Libtool 库 ;

首先需要准备一个源文件 compress.c，代码如下：

###### **清单 1: compress.c**

#include <sys/mman.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <limits.h>

#include <assert.h>

#include <zlib.h>

/\* 一个简单的文件压缩函数 \*/

int compress\_file (const char \*filename)

{

int src\_fd, dest\_fd;

struct stat sb;

Bytef \*src, \*dest;

uLong dest\_len;

char dest\_file[PATH\_MAX];

src\_fd = open (filename, O\_RDONLY);

assert (dest\_fd != -1);

assert (fstat (src\_fd, &sb) != -1);

src = mmap (NULL, sb.st\_size, PROT\_READ, MAP\_PRIVATE, src\_fd, 0);

assert (src != MAP\_FAILED);

dest\_len = compressBound (sb.st\_size);

dest = malloc (dest\_len);

assert (dest);

assert (compress (dest, &dest\_len, src, sb.st\_size) == Z\_OK);

munmap (src, sb.st\_size);

close (src\_fd);

snprintf (dest\_file, sizeof (dest\_file), "%s.z", filename);

dest\_fd = creat (dest\_file, S\_IRUSR | S\_IWUSR);

assert (dest\_fd != -1);

write (dest\_fd, dest, dest\_len);

close (dest\_fd);

free (dest);

return 0;

}

这个文件实现了一个函数 compress\_file()，它接收一个文件名作为参数，然后对文件进行压缩，生成一个 .z结尾的压缩文件。在这个文件中使用了 compress()函数，这个函数是有由 libz 提供的。

从源文件建立 Libtool 库需要经过两个步骤，先建立 Libtool 对象文件，再建立 Libtool 库。

##### **建立 Libtool 对象文件**

如果使用传统的方式，建立对象文件通常使用下面的命令 :

$ gcc -c compress.c

使用 Libtool 则使用下面的命令 :

$ **libtool --mode=compile gcc -c foo.c**

可以看到，使用 Libtool 只需要将“传统”的命令 (gcc -c foo.c) 作为参数传递给 Libtool 即可。

在上面的命令中，libtool 使用 **compile**模式 (--mode=compile 选项 )，这是建立对象文件的模式，Libtool 还有其它的模式，后面将介绍。

上面的命令输出如下 :

mkdir .libs

gcc -c compress.c -fPIC -DPIC -o .libs/compress.o

gcc -c compress.c -o compress.o >/dev/null 2>&1

它建立了两个文件，一个是 **.libs/compress.o**，在建立这个文件时，Libtool 自动插入了 -fPIC和 -DPIC选项，告诉编译器生成位置独立的代码，之后将用这个文件来建立动态链接库。生成第二个文件**compress.o**没有添加额外的选项，它准备用来建立静态链接库。

除了上面的两个文件之外，Libtool 还建立了一个文件 **compress.lo**，这个文件就是 Libtool 对象文件，实际上也就是一个文本文件，里面记录了建立动态链接库和静态链接库分别所需要的真实文件名称，后面 Libtool 将使用这个文件而不是直接的使用 .libs/compress.o 和 compress.o。

##### **建立 Libtool 库**

用下面的命令建立 Libtool 库 :

$ **libtool --mode=link gcc -o libcompress.la compress.lo -rpath /tmp -lz**

注意这里使用 compress.lo 作为输入文件，并且告诉 Libtool 生成的目标文件为 **libcompress.la**，.la 是 Libtool 的库文件后缀。

-rpath选项告诉 Libtool 这个库将被安装到什么地方，如果省略了 -rpath选项，那么不会生成动态链接库。

因为我们的库中使用了 libz 提供的 compress 函数，所以也提供了 -lz 选项，Libtool 会记住这个依赖关系，后续在使用我们的库时自动的将依赖的库链接进来。

上面的命令输出如下 :

gcc -shared .libs/compress.o -lz -Wl,-soname -Wl,libcompress.so.0

-o .libs/libcompress.so.0.0.0

(cd .libs && rm -f libcompress.so.0 &&

ln -s libcompress.so.0.0.0 libcompress.so.0)

(cd .libs && rm -f libcompress.so &&

ln -s libcompress.so.0.0.0 libcompress.so)

ar cru .libs/libcompress.a compress.o

ranlib .libs/libcompress.a

creating libcompress.la

(cd .libs && rm -f libcompress.la &&

ln -s ../libcompress.la libcompress.la)

可以看到，Libtool 自动的插入了建立动态链接库需要的编译选项 -shared。并且，它也建立了静态链接库 **.libs/libcompress.a**，后面我们将会介绍如何控制 Libtool 只建立需要的库。

你可能会奇怪为什么建立的动态链接库有 .0 和 .0.0.0 这样的后缀，这里先不用理会它，后面在介绍 Libtool 库版本信息时将会解释这点。

值得注意的是，Libtool 希望后续使用 libcompress.la 文件而不是直接使用 libcompress.a 和 **libcompress.so** 文件，如果你这样做，虽然可以，但会破坏 Libtool 库的可移植性。

##### **安装 Libtool 库**

如果打算发布建立好的 Libtool 库，可以使用下面的命令安装它 :

$ libtool --mode=install install -c libcompress.la /tmp

我们需要告诉 Libtool 使用的安装命令，Libtool 支持 install 和 cp，这里使用的是 install。

虽然前面我们在建立库时，通过 -rpath 选项指定了库准备安装的路径 (/tmp)，但是这里我们还得要提供安装路径。请确保它们一致。

这个命令的输出如下 :

install .libs/libcompress.so.0.0.0 /tmp/libcompress.so.0.0.0

(cd /tmp && { ln -s -f libcompress.so.0.0.0 libcompress.so.0 ||

{ rm -f libcompress.so.0 &&

ln -s libcompress.so.0.0.0 libcompress.so.0; }; })

(cd /tmp && { ln -s -f libcompress.so.0.0.0 libcompress.so ||

{ rm -f libcompress.so && ln -s libcompress.so.0.0.0 libcompress.so; }; })

install .libs/libcompress.lai /tmp/libcompress.la

install .libs/libcompress.a /tmp/libcompress.a

chmod 644 /tmp/libcompress.a

ranlib /tmp/libcompress.a

...

可以看到它安装了真实的动态链接库和静态链接库，同时也安装了 Libtool 库文件 libcompress.la，这个文件可以被后续的 Libtool 命令使用。

在安装完成之后，可能还需要做一些配置才能正确使用，Libtool 的 finish 模式可以在这方面给我们一些提示 :

$ libtool -n --mode=finish /tmp

这个命令的输出有点长，所以不在这里列出，如果不能正常的使用安装好的库，请运行这个命令。

##### **使用 Libtool 库**

要在应用程序中使用前面创建的 Libtool 库很简单，准备一个源文件 main.c，它将使用 libcompress.la 库中定义的函数，代码如下 :

###### **清单 2: main.c**

#include <stdio.h>

extern int compress\_file (const char \*filename);

int main (int argc, char \*argv[])

{

if (argc < 2)

{

printf ("usage : %s file\n", argv[0]);

return 1;

}

return compress\_file (argv[1]);

}

我们还是要先为 main.c 建立 Libtool 对象文件，这和前面的方法一样 :

$ libtool --mode=compile gcc -c main.c

###### **使用安装的库**

然后使用下面的命令链接执行文件 :

$ libtool --mode=link gcc -o main main.lo /tmp/libcompress.la

我们也可以直接使用 libcompress.a 或者 libcompress.so，但是使用 Libtool 更加简单，因为它会将帮助你解决依赖关系，例如我们的 libcompress 依赖 libz。

上面命令的输出如下 :

gcc -o main .libs/main.o /tmp/libcompress.so -lz

-Wl,--rpath -Wl,/tmp -Wl,--rpath -Wl,/tmp

这里，Libtool 自动选择链接动态链接库，并且加上了运行时需要的 --rpath 选项，以及依赖的库 -lz。

如果要使用静态链接库，只需要加上 -static-libtool-libs选项即可，如下 :

$ libtool --mode=link gcc -o main main.lo /tmp/libcompress.la -static-libtool-libs

这个命令的输出如下 :

gcc -o main .libs/main.o /tmp/libcompress.a -lz

###### ****使用未安装的库****

也可以使用还没有安装的库，这和使用安装好的库几乎相同，只是指定的输入文件位置不一样，假如我们在同一个目录中开发 compress.c 和 main.c，那么使用下面的命令 :

$ libtool --mode=link gcc -o main main.lo ./libcompress.la

和使用安装的库不一样，这个时候建立的 main 程序只是一个封装脚本，如果你直接执行它不会有什么问题，但是如果你想调试它，例如 :

$ gdb main

gdb 会报怨 main 不是可执行格式，不能接受。这个时候我们需要使用 Libtool 的执行模式，使用下面的命令调试程序 :

$ libtool --mode=execute gdb main

##### **卸载 Libtool 库**

使用下面的命令可以卸载安装的库 :

$ libtool --mode=uninstall rm /tmp/libcompress.la

这个命令的输出如下 :

rm /tmp/libcompress.la /tmp/libcompress.so.0.0.0 /tmp/libcompress.so.0

/tmp/libcompress.so /tmp/libcompress.a

这将删除所有安装的库文件。

[https://segmentfault.com/a/1190000006915719#articleHeader5](https://segmentfault.com/a/1190000006915719" \l "articleHeader5)

<https://blog.csdn.net/tmxkwzy/article/details/53317094>

<https://blog.csdn.net/thalo1204/article/details/49183911>

<https://blog.csdn.net/qq_34924407/article/details/82917391>

<https://darktea.github.io/notes/2012/06/24/autotools.html>

<https://blog.csdn.net/kinbo88/article/details/36006509>

<https://www.cnblogs.com/xmphoenix/p/4065950.html>