# autotools 的使用

## 一、 autotools 简介

make 项目管理器具有着强大的功能。的确，Makefile 可以帮助 make 完成它的使命，但要承认的是，编写 Makefile 不是一件容易的事情，尤其是对于一个较大的项目而言更是如此。而且 Makefile 很多情况下并不能在众多版本的 Linux 操作系统下通用。

那么，有没有一种轻松的手段生成 Makefile 而同时又能让用户享受 make 的优越性呢？autotool 是系列工具正是为此而设计的，它只需要用户输入简单的目标文件、依赖文件、文件目录等，就可以轻松地生成 Makefile 了。另外，这些工具还可以完成系统配置信息的收集，从而可以方便地处理各种移植性的问题。也正是基于此，现在的 Linux 上的软件开发一般都用 autotools 来制作 Makefile 文件。

autotools 是系列工具，首先要确认系统是否安装了以下工具（可以用 which 命令进行查看）：

1) autoscan
2) aclocal
3) autoconf
4) autoheader
5) automake

记住，这一系列工具看着复杂，最终的目标还是生成 Makefile。

一般情况下，系统会默认安装这一系列工具的，如果未安装，则在 Ubuntu 中可以通过以下命令安装：

```
sudo apt-get install automake
```

用 autotools 产生 Makefile 文件的总体流程图如图 1 所示：

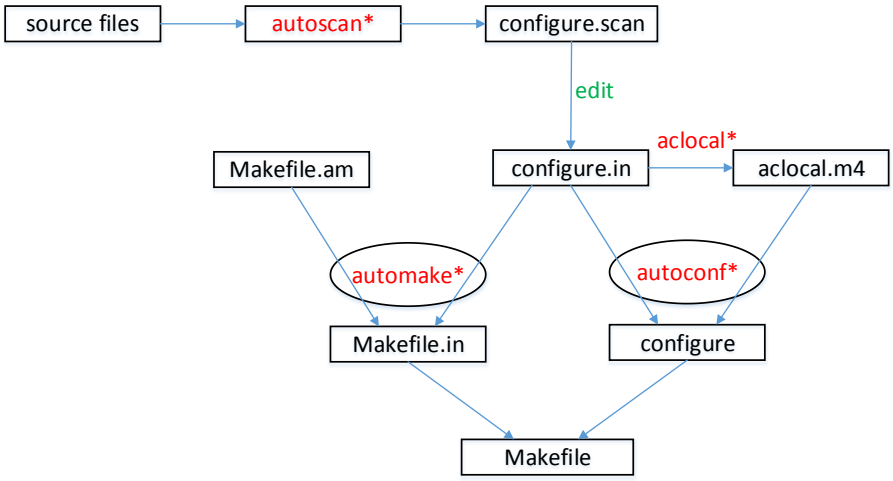


图 1 autotools 生成 Makefile 文件流程图

利用 autotools 系列工具生成“Makefile”文件的基本步骤如下所示：

| 步骤 | 功能 |
| --- | --- |
| 创建源文件与库文件 | 可以把所有的源文件 |
| 执行命令 autoscan | 在给定目标及其子目录树中检查源文件 |
| 编辑 configure.scan | “configure.scan”是“configure.in”的原型文件 |
| 执行命令 aclocal | 生成“aclocal.m4”文件 |
| 执行命令 autoconf | 利用“configure.in 和“aclocal.m4”文件生成“configure”文件 |
| 执行命令 autoheader | 生成“config.h.in”文件 |
| 编辑 makefile.am 文件 | 是 automake 的脚本配置文件 |
| 执行命令 automake | 生成“configure.in”文件 |
| 执行命令 make dist | 用 make 把程序和相关的文档打包，压缩包形式以供发布 |
| 执行命令./configure | 生成 Makefile 文件 |
| 执行命令 make | 编译源文件，生成可执行文件 |
| 执行命令 make install | 把程序安装到系统目录 |

# 二、 C 源文件同一目录下 autotools 的使用

如果你的源文件都放在同一个目录下面，那么使用 Autotools 的时候会相对简单很多。比较著名的开源软件 Memcached 也是放在同一目录下的，你可以去看下它的源码包。

下面会按照步骤来实现同一目录下的 Autotools 工具的使用。

1. 源代码例子

入口文件 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "sum.h"
#include "get.h"

//入口主函数
int main() {
    int x = 10;
    int y = 20;
    int z = sum(&x, &y);
    puts("This is Main");
    printf("Z:%d\n", z);
    x = 20;
    z = get(&x, &y);
    printf("Z:%d\n", z);
    return 1;
}
```

sum.h 和 sum.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int sum(int *x, int *y);

/*==========================*/
#include "sum.h"
#include "val.h"

int sum(int *x, int *y) {
    val(x);
    puts("This is SUM Method!=========HDH");
    return *x + *y;
}
```

val.h 和 val.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int val(int *x);

/*================*/
#include "val.h"
int val(int *x) {
    puts("This is Value==");
    printf("X:%d \n", *x);
    return 0;
}
```

get.h 和 get.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int get(int *x, int *y);
/*================*/
 #include "get.h"
int get(int *x, int *y) {
    puts("This is get");
    return (*x) * (*y);
}
```

2. autoscan 命令

第一步，我们需要在我们的项目目录下执行 autoscan 命令。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 需要重命名成 configure.ac，然后编辑这个配置，我们才能继续执行后面的命令。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 | 宏用于生成 config.h 文件，以便 autoheader 命令使用。 |
| AC_PROG_CC | 指定编译器，默认 GCC |
| AC_CONFIG_FILES | 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。 |

3. aclocal 命令

第二步，执行 aclocal 命令。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。

4. autoconf 命令

第三步，执行 autoconf 命令。这个命令将 configure.ac 文件中的宏展开，生成 configure 脚本。这个过程可能要用到 aclocal.m4 中定义的宏。

5. autoheader 命令

第四步，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。该例子中没有附加的符号定义，所以不需要创建“acconfig.h”文件。

6. 创建 Makefile.am 文件

第五步，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 更高层次的编译规则，可以和 configure.in 文件一起通过调用 automake 命令，生成 Makefile.in 文件，再调用./configure 的时候，就将 Makefile.in 文件自动生成 Makefile 文件了。所以 Makefile.am 文件是比 Makefile 文件更高的抽象。

下面我根据自己的工作中的一些应用，来讨论 Makefile.am 的编写。我觉得要注意的问题主要是将编译什么文件？这个文件会不会安装？这个文件被安装到什么目录下？可以将文件编译成可执行文件来安装，也可以编译成静态库文件安装，常见的文件编译类型有下面几种：

PROGRAMS---表示可执行文件

LIBRARIES---表示库文件

LTLIBRARIES---这也是表示库文件，前面的 LT 表示 libtool。

HEADERS---头文件。

SCRIPTS---脚本文件，这个可以被用于执行。如：example_SCRIPTS，如果用这样的话，需要我们自己定义安装目录下的 example 目录，很容易的，往下看。

DATA---数据文件，不能执行。

1） 编译可执行文件

```
#Makefile.am 文件
bin_PROGRAMS = xxx
#bin_PROGRAMS 表示指定要生成的可执行应用程序文件，这表示可执行文件在安装时需要被安装到系统
#中；如果只是想编译，不想被安装到系统中，可以用 noinst_PROGRAMS 来代替

xxx_SOURCES = a.c b.c c.c main.c d.c xxx.c
#xxx_SOURCES 表示生成可执行应用程序所用的源文件，这里注意，xxx_是由前面的 bin_PROGRAMS
#指定的，如果前面是生成 example,那么这里就是 example_SOURCES，其它的类似标识也是一样

xxx_CPPFLAGS = -DCONFIG_DIR=\"$(sysconfdir)\" -DLIBRARY_DIR=\"$(pkglibdir)\"
#xxx_CPPFLAGS 这和 Makefile 文件中一样，表示 C 语言预处理参数，这里指定了 DCONFIG_DIR，以后
#在程序中，就可以直接使用 CONFIG_DIR。不要把这个和另一个 CFLAGS 混淆，后者表示编译器参数

xxx_LDFLAGS = -export-dynamic -lmemcached
#xxx_LDFLAGS 连接的时候所需库文件的标识，这个也就是对应一些如-l,-shared 等选项

noinst_HEADERS = xxx.h
#这个表示该头文件只是参加可执行文件的编译，而不用安装到安装目录下。如果需要安装到系统中，
#可以用 include_HEADERS 来代替

INCLUDES = -I/usr/local/libmemcached/include/
#INCLUDES  链接时所需要的头文件
```

```
xxx_LDADD = $(top_builddir)/sx/libsession.a \
                $(top_builddir)/util/libutil.a
#xxx_LDADD 链接时所需要的库文件，这里表示需要两个库文件的支持
```

2） 编译动态库文件

要生成 xxx.so 文件

```
#Makefile.am 文件
xxxlibdir=$(libdir)//新建一个目录，该目录就是 lib 目录，运行后 xxx.so 将安装在该目录下
xxxlib_PROGRAMS=xxx.so
xxx_so_SOURCES=xxx.c
xxx_so_LDFLAGS=-shared -fpic //GCC 编译动态库的选项
```

3） 编译静态库文件

要生成 xxx.a 文件

```
#Makefile.am 文件
noinst_LTLIBRARIES = xxx.a
noinst_HEADERS = a.h b.h
xxx_a_SOURCES = a.c b.c xxx.c
```

在本文的例子中，Makefile.am 如下：

```
AUTOMARK_OPTIONS = foreign
bin_PROGRAMS = hello
hello_SOURCES = main.c val.h val.c get.h get.c sum.h sum.c
```

A. AUTOMAKE_OPTIONS：由于 GNU 对自己发布的软件有严格的规范，比如必须附带许可证声明文件 COPYING 等，否则 automake 执行时会报错. automake 提供了 3 中软件等级:foreign, gnu 和 gnits, 供用户选择。默认级别是 gnu. 在本例中，使用了 foreign 等级,它只检测必须的文件。

B. bin_PROGRAMS=hello：生成的可执行文件名称，生成多个可执行文件，可以用空格隔开。

C. hello_SOURCES：生成可执行文件 hello 需要依赖的源文件。其中 hello_为可执行文件的名称。

7. automake 命令

第六步，执行 automake --add-missing 命令。该命令生成 Makefile.in 文件。使用选项 “--add-missing” 可以让 automake 自动添加一些必需的脚本文件。如果发现一些文件不存在，可以通过手工 touch 命令创建。

8. configure 命令

第七步，估计大家都对 ./congigure 这个命令很熟悉吧。大部分 Linux 软件安装都先需要执行./configure，然后执行 make 和 make install 命令。

./configure 主要把 Makefile.in 变成最终的 Makefile 文件。configure 会把一些配置参数配置到 Makefile 文件里面。

9. make 命令

第八步，执行 make 命令，执行 make 命令后，就生成了可执行文件 hello。

# 三、 C 源文件不同一目录下 autotools 的使用

如果你的入口文件 main.c 和依赖的文件不是在同一个目录中的，使用 autotools 来管理项目的时候会稍微复杂一下。

在不同的目录下，项目会生成*.a 文件的静态连接（静态连接相当于将多个.o 目标文件合成一个）。最外层的 main.c 会通过静态连接方式来实现连接。

1. 源代码例子

这个例子中会加入 libevent 和 pthread，让例子稍显复杂，这样可以详细的介绍不同目录下的 Autotools 的使用。

我们创建两个目录：include/：放置.h 头文件；src/：放置.c 源文件

入口文件 main.c

```
#include "include/common.h"

//入口主函数
int main() {
    puts("当前线程 sleep 2 秒");
    sleep(2);
    int x = 10;
    int y = 20;
    int z = sum(&x, &y);
    puts("This is Main");
    printf("Z:%d\n", z);
    x = 20;
    z = get(&x, &y);
    printf("Z:%d\n", z);
    return 1;
}
```

common.h 文件：

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <pthread.h>
```

get.h：

```
int get(int *x, int *y);
```

sum.h

```
int sum(int *x, int *y);
```

val.h

```
#include "common.h"
int val(int *x);
```

get.c

```
#include "../include/get.h"
int get(int *x, int *y) {
        puts("This is get");
        return (*x) * (*y);
}
```

sum.c

```
#include "../include/sum.h"
#include "../include/val.h"

int sum(int *x, int *y) {
        val(x);
        puts("This is SUM Method!=========HDH");
        return *x + *y;
}
```

val.c

```
#include "../include/val.h"
int val(int *x) {
        //引入 libevent 的方法
        struct event_base *base; //定义一个 event_base
        base = event_base_new(); //初始化一个 event_base
        *x =  event_base_get_method(base); //查看用了哪个 IO 多路复用模型，linux 一下用 epoll
        printf("METHOD:%s\n", x);
        event_base_free(base);  //销毁 libevent
```

```
        puts("This is Value==");
        printf("X:%d \n", *x);
        return 0;
}
```

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
include  main.c  src
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls ./include/
common.h  get.h  sum.h  val.h
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls ./src/
get.c  sum.c  val.c
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

2. 创建 Makefile.am 文件

在项目根目录下先创建 Makefile.am 文件。

```
AUTOMAKE_OPTIONS=foreign #软件等级
SUBDIRS=src  #先扫描子目录
bin_PROGRAMS=hello #软件生成后的可执行文件名称
hello_SOURCES=main.c #当前目录源文件
hello_LDADD=src/libpro.a #静态连接方式 连接 src 下生成的 libpro.a 文件
LIBS = -l pthread -l event #因为我们项目中用到了 libevent 和 pthread，这个是动态连接
```

在 src/目录下创建 Makefile.am 文件。

```
noinst_LIBRARIES=libpro.a  #生成的静态库文件名称，noinst 加上之后是只编译，不安装到系统中。
libpro_a_SOURCES=sum.c get.c val.c #这个静态库文件需要用到的依赖
include_HEADERS=../include/common.h ../include/sum.h ../include/get.h ../include/val.h #导入需要依赖的头文件
```

说明：src/目录下面不加 include_HEADERS 也是可以运行的，但是在使用 make dist 打包命令后，并不会将 include/文件夹打包进去，所以还是需要加上 include_HEADERS。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# gedit Makefile.am
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# cd src/
root@ubuntu:~/Documents/linux-c-learn/automake-learn3/src# ls
get.c  sum.c  val.c
root@ubuntu:~/Documents/linux-c-learn/automake-learn3/src# gedit Makefile.am
root@ubuntu:~/Documents/linux-c-learn/automake-learn3/src# ls
get.c  Makefile.am  sum.c  val.c
root@ubuntu:~/Documents/linux-c-learn/automake-learn3/src# cd ..
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
include  main.c  Makefile.am  src
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

3. 执行 autoscan 命令

第一步，我们需要在我们的项目目录下执行 autoscan 命令。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，我们才能继续执行后面的命令。

修改 configure.ac 文件，主要添加 AC_PROG_RANLIB（生成静态库）；AC_PROG_LIBTOOL（用来生成动态库）。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# autoscan
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
autoscan.log  configure.scan  include  main.c  Makefile.am  src
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# mv configure.scan configu
re.ac
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
autoscan.log  configure.ac  include  main.c  Makefile.am  src
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

原来的 configure.ac 文件内容：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.69])
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
AC_CONFIG_SRCDIR([main.c])
AC_CONFIG_HEADERS([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.
AC_CHECK_HEADERS([stdlib.h unistd.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_OUTPUT
```

修改为：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.69])
AC_INIT([hello], [1.0], [769413715@qq.com])
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([main.c])
AC_PROG_RANLIB
AC_CONFIG_HEADERS([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.
AC_CHECK_HEADERS([stdlib.h unistd.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_OUTPUT
```

生成静态库

4. 执行 aclocal 命令

第二步，执行 aclocal 命令。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。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# aclocal
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
aclocal.m4      autoscan.log  configure.ac~  main.c       src
autom4te.cache  configure.ac  include        Makefile.am
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

5. 执行 autoconf 命令

第三步，执行 autoconf 命令。这个命令将 configure.ac 文件中的宏展开，生成 configure 脚本。这个过程可能要用到 aclocal.m4 中定义的宏。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# autoconf
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
aclocal.m4      autoscan.log  configure.ac   include  Makefile.am
autom4te.cache  configure     configure.ac~  main.c   src
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

6. autoheader 命令

第四步，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。该例子中没有附加的符号定义，所以不需要创建“acconfig.h”文件。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# autoheader
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
aclocal.m4      autoscan.log  configure     configure.ac~  main.c       src
autom4te.cache  config.h.in   configure.ac  include        Makefile.am
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

7. automake 命令

第五步，执行 automake --add-missing 命令。该命令生成 Makefile.in 文件。使用选项“--add-missing”可以让 automake 自动添加一些必需的脚本文件。如果发现一些文件不存在，可以通过手工 touch 命令创建。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# automake --add-missing
configure.ac:6: warning: AM_INIT_AUTOMAKE: two- and three-arguments forms are de
precated.  For more info, see:
configure.ac:6: http://www.gnu.org/software/automake/manual/automake.html#Modern
ize-AM_005fINIT_005fAUTOMAKE-invocation
configure.ac:12: installing './compile'
configure.ac:6: installing './install-sh'
configure.ac:6: installing './missing'
Makefile.am: installing './depcomp'
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

8. configure 命令

第六步，执行./configure 命令。./configure 主要把 Makefile.in 变成最终的 Makefile 文件。configure 会把一些配置参数配置到 Makefile 文件里面。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for ranlib... ranlib
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc understands -c and -o together... yes
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
```

```
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
checking for unistd.h... (cached) yes
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: executing depfiles commands
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
aclocal.m4      compile      config.log     configure.ac   include     Makefile     missing
autom4te.cache  config.h     config.status  configure.ac~  install-sh  Makefile.am  src
autoscan.log    config.h.in  configure      depcomp        main.c      Makefile.in  stamp-h1
```

9. make 命令

第七步，执行 make 命令。make 执行后，会生成 hello 的可执行文件。

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# make
make  all-recursive
make[1]: Entering directory `/root/Documents/linux-c-learn/automake-learn3'
Making all in src
make[2]: Entering directory `/root/Documents/linux-c-learn/automake-learn3/src'
gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT sum.o -MD -MP -MF .deps/sum.Tpo -c -o sum.o sum.c
mv -f .deps/sum.Tpo .deps/sum.Po
gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT get.o -MD -MP -MF .deps/get.Tpo -c -o get.o get.c
mv -f .deps/get.Tpo .deps/get.Po
gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT val.o -MD -MP -MF .deps/val.Tpo -c -o val.o val.c
val.c: In function ‘val’:
val.c:7:12: warning: assignment makes integer from pointer without a cast [enabled by default
]
        *x =  event_base_get_method(base); //查看用了哪个IO多路复用模型，linux一下用epoll

            ^
val.c:8:9: warning: format ‘%s’ expects argument of type ‘char *’, but argument 2 has type ‘i
nt *’ [-Wformat=]
        printf("METHOD:%s\n", x);
        ^
mv -f .deps/val.Tpo .deps/val.Po
rm -f libpro.a
ar cru libpro.a sum.o get.o val.o
ranlib libpro.a
make[2]: Leaving directory `/root/Documents/linux-c-learn/automake-learn3/src'
make[2]: Entering directory `/root/Documents/linux-c-learn/automake-learn3'
gcc -DHAVE_CONFIG_H -I.     -g -O2 -MT main.o -MD -MP -MF .deps/main.Tpo -c -o main.o main.c
mv -f .deps/main.Tpo .deps/main.Po
gcc  -g -O2   -o hello main.o src/libpro.a  -l pthread -l event
make[2]: Leaving directory `/root/Documents/linux-c-learn/automake-learn3'
make[1]: Leaving directory `/root/Documents/linux-c-learn/automake-learn3'
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

```
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ls
aclocal.m4      config.h       configure     hello      main.o       missing
autom4te.cache  config.h.in    configure.ac  include    Makefile     src
autoscan.log    config.log     configure.ac~ install-sh Makefile.am  stamp-h1
compile         config.status  depcomp       main.c     Makefile.in
root@ubuntu:~/Documents/linux-c-learn/automake-learn3# ./hello
当前线程sleep 2秒
METHOD:epoll
This is Value==
X:-620653291
This is SUM Method!=========HDH
This is Main
Z:-620653271
This is get
Z:400
root@ubuntu:~/Documents/linux-c-learn/automake-learn3#
```

# 四、 autotools 运行流程

1. 流程总结：

1) 执行 autoscan 命令。这个命令主要用于扫描工作目录，并且生成 configure.scan 文件。

2) 修改 configure.scan 为 configure.ac 文件，并且修改配置内容。

3) 执行 aclocal 命令。扫描 configure.ac 文件生成 aclocal.m4 文件。

4) 执行 autoconf 命令。这个命令将 configure.ac 文件中的宏展开，生成 configure 脚本。

5) 执行 autoheader 命令。该命令生成 config.h.in 文件。

6) 新增 Makefile.am 文件，修改配置内容

7) 执行 automake --add-missing 命令。该命令生成 Makefile.in 文件。

8） 执行 ./congigure 命令。将 Makefile.in 命令生成 Makefile 文件。

9） 执行 make 命令。生成可执行文件。

2. make 命令详解

1） make 命令：编译文件。make 命令主要通过 Makefile 文件生成可执行文件。

2） make clean 命令。清楚编译的文件，包括目标文件*.o 和可执行文件。

3） make install 命令把目标文件安装到系统中。默认安装到/usr/local/bin 目录下面。

4） make uninstall 命令，把目标文件从系统中卸载。

5） make dist 命令，打包发布。

如何使用发布的文件：

1） 下载到“hello-1.0.tar.gz”压缩文档。

2） 使用“tar -zxvf hello-1.0.tar.gz”命令解压。

3） 使用 “./configure” 命令，主要是生成 Makefile 命令，已经一些配置初始化。

4） 使用 “make” 命令编译源代码文件生成软件包。

5） 使用“make install ”命令来安装编译后的软件包到系统中。

3. Makefile.am 解读

1） 可执行文件类型

可执行文件类型主要是指最终生成的可执行的文件。例如我们上面“c 源文件同一目录下 autotools 的使用”中的例子。

| 书写格式 | 说明 |
| --- | --- |
| bin_PROGRAMS | 生成的可执行文件名称。如果生成的可执行文件名称为多个，则可以通过空格的方式分隔。<br>bin_PROGRAMS：当运行 make install 命令的时候，可执行文件会默认安装到 linux 系统的/usr/local/bin 目录下面<br>noinst_PROGRAMS：如果 make install 的时候不想被安装，可以使用 noinst_PROGRAMS 命令。<br>例子：bin_PROGRAMS=hello |
| hello_SOURCES | 编译成可执行文件所依赖的.c 源文件。多个源文件之间用空格分隔。hello 为可执行文件名称。 |
| hello_LDADD | 编译成可执行文件过程中，连接所需的库文件，包括*.so 的动态库文件和.a 的静态库文件。 |
| hello_LDFLAGS | 连接的时候所需库文件的标识 |

```
bin_PROGRAMS=hello #软件生成后的可执行文件名称为 hello
```

```
hello_SOURCES=main.c #当前目录源文件，如果当前目录有多个源文件，通过空格进行分隔
hello_LDADD=src/libpro.a #连接的时候所需的库文件
hello_LDFLAGS=    #连接的时候所需库文件的标识
LIBS= -l pthread -l event #第三方的库
```

2) 静态库文件类型

静态库文件类型，一般会将 c 源码放在不同的文件夹中，并且每个文件夹中都会有各自的 Makefile.am 文件，并且会被编译成静态链接库 *.a 格式的文件。

注意：静态库使用中，需要对 configure.ac 中加入 AC_PROG_RANLIB。

| 书写格式 | 说明 |
| --- | --- |
| noinst_LIBRARIES | 生成静态库（*.a）或者动态库(*.so)的名称。<br>库文件一般以 lib*.a 或者 lib*.so 来命名。<br>noinst_LIBRARIES:当运行 make install 的时候，库文件不会被安装到 linux 默认的/usr/local/lib 目录下。<br>lib_LIBRARIES:当运行 make intsall 的时候，则会被安装到/usr/local/lib 目录下。<br>下面的例子：noinst_LIBRARIES=libpro.a |
| libpro_a_SOURCES | c 的源文件，libpro_a 即上面的 livpro.a。多个文件用空格分开。 |
| libpro_a_LDADD | 加载所需的库文件。 |
| libpro_a_LDFLAGS | 编译的时候的连接标识。 |

```
noinst_LIBRARIES=libpro.a  #生成的静态库文件名称，noinst 加上之后是只编译，不安装到系统中。
libpro_a_SOURCES=sum.c get.c val.c #这个静态库文件需要用到的源文件。
libpro_a_LDADD = #加载库文件
libpro_a_LDFLAGS= #连接的时候所需库文件的标识
```

3) 头文件

我们一般需要导入一些*.h 的头文件，如果你在 Makefile.am 中没有标识需要导入的头文件，可能在 make dist 打包的时候出现问题，头文件可能不会被打进包里面。

```
include_HEADERS=../include/common.h ../include/sum.h ../include/get.h ../include/val.h  #可以将头文件引入
```

make install，头文件默认会被安装到 linux 系统/usr/local/include。

4) 数据文件

```
data_DATA = data1 data2
```

5) 常用变量

| 变量 | 含义 |
| --- | --- |
| INCLUDE | 比如链接时所需要的头文件 |
| LDADD | 比如链接时所需要的库文件 |

| | |
|---|---|
| LDFLAGS | 比如链接时所需要的库文件选项标志 |
| SUBDIRS | 源程序和一些默认的文件将自动打入.tar.gz 包，其他文件若要进入.tar.gz 包可以用这种办法 |

```
AUTOMAKE_OPTIONS=foreign #软件等级
SUBDIRS=src  #先扫描子目录，多个目录用空格隔开
LIBS = -l pthread -l event #因为我们项目中用到了 libevent 和 pthread，这个是动态连接，在编译的时候会自动加上 -l pthread -l event
EXTRA_DIST = conf   #打包一些配置文件
```

6） 安装目录

我们知道，默认情况下，执行 make install 命令，则会将文件安装到/usr/local/bin /usr/local/include /usr/local/lib 目录下面。我们可以通过命令./configure --prefix= 生成 Makefile 文件的时候，配置 make install 命令执行的时候的文件安装路径。下面这个例子，我们在执行 make install 的时候，程序会被安装到/home/test 目录下面。

```
./configure --prefix=/home/test
make
sudo make install
```

下面这些变量是已经定义好的安装路径的变量。用户也可以修改这些变量。例如将 bindir 修改成$(prefix)/bin。

```
bindir = $(prefix)/bin
libdir = $(prefix)/lib
datadir=$(prefix)/share
sysconfdir=$(prefix)/etc
includedir=$(prefix)/include
```

假如我们有自定义的文件夹，我们需要将这个文件夹下的内容安装到安装目录，则需要配置一个自定义的文件夹目录 confdir。

```
confdir=${prefix}/conf   #conf 为名称  dir 为每个文件夹变量必须带上
conf_DATA=conf/*      #这个是将 conf/目录下的内容安装到 confdir 目录下
EXTRA_DIST=conf       #在 make dist 打包的时候 也要将扩展文件夹打包进去
```

confdir 为需要创建的文件夹目录。conf_DATA 将需要拷贝的文件内容拷贝到${prefix}/conf 目录中去。