autotools的使用

1. 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 | |

1. 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);  } |

1. 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。 | |

1. 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。

1. 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。

1. 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。

1. 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 |

1. 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 |

1. 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 |

1. 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。
2. 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。
3. 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。
4. 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。

1. 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。

1. 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。

1. 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

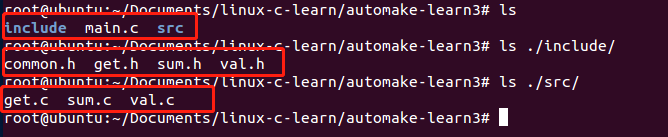
|  |
| --- |
| #include "../include/get.h"  int get(int \*x, int \*y) {  puts("This is get");  return (\*x) \* (\*y);  } |

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;  } |



1. 创建Makefile.am文件

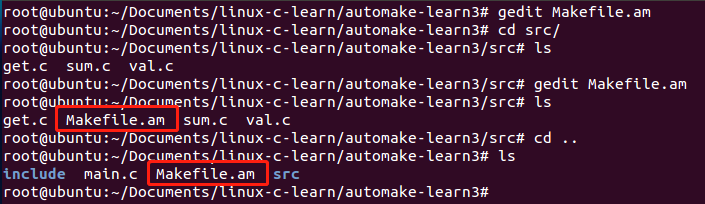
在项目根目录下先创建Makefile.am文件。

|  |
| --- |
| 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 |

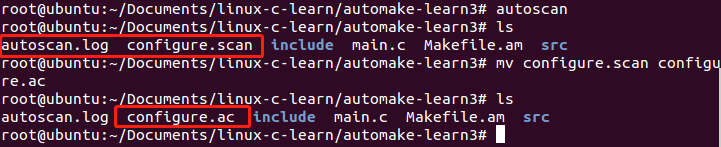
说明：src/目录下面不加include\_HEADERS也是可以运行的，但是在使用make dist打包命令后，并不会将include/文件夹打包进去，所以还是需要加上include\_HEADERS。



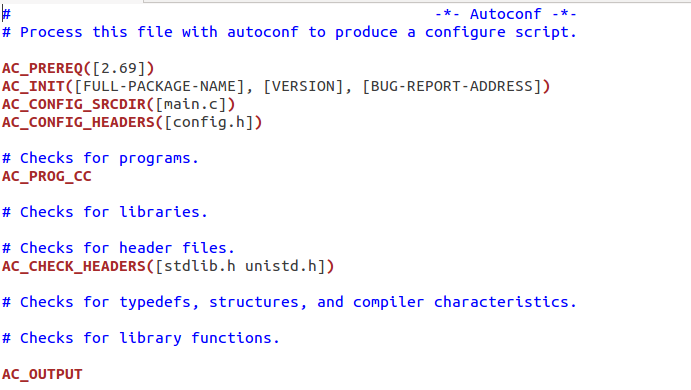
1. 执行autoscan命令

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。

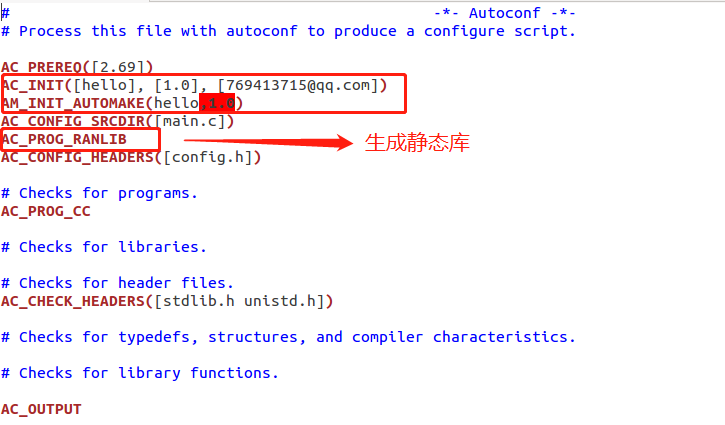
修改configure.ac文件，主要添加AC\_PROG\_RANLIB（生成静态库）；AC\_PROG\_LIBTOOL（用来生成动态库）。



原来的configure.ac文件内容：

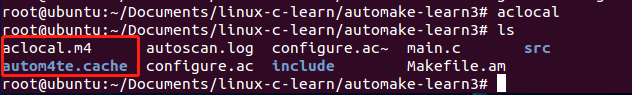


修改为：



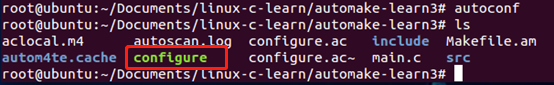
1. 执行aclocal命令

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。



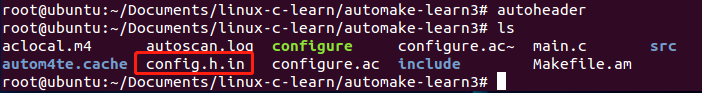
1. 执行autoconf命令

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。



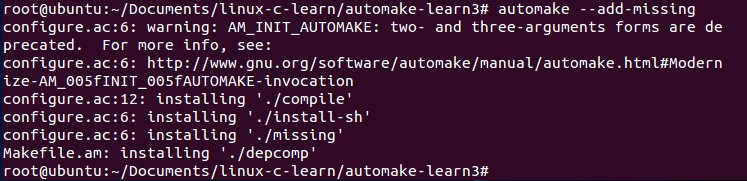
1. autoheader命令

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。



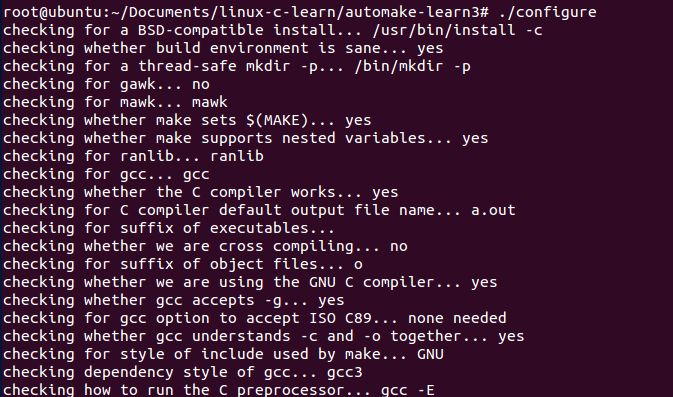
1. automake命令

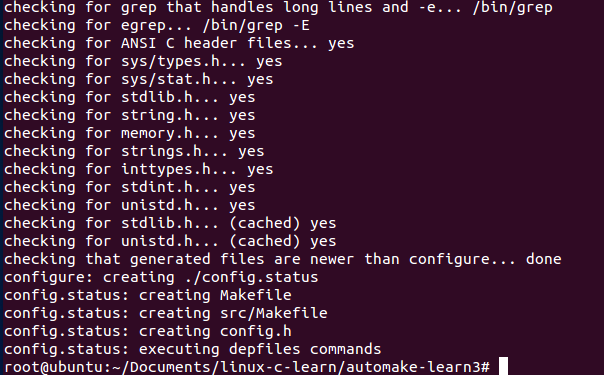
第五步，执行automake --add-missing命令。该命令生成 Makefile.in 文件。使用选项“--add-missing”可以让automake自动添加一些必需的脚本文件。如果发现一些文件不存在，可以通过手工touch命令创建。

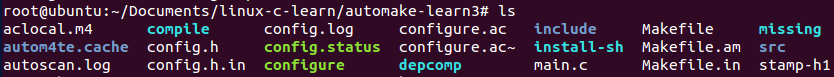


1. configure命令

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。

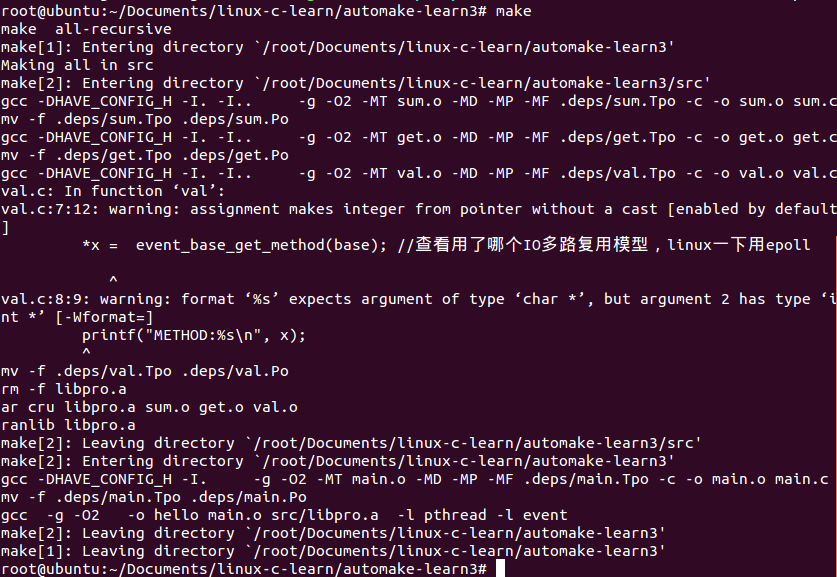


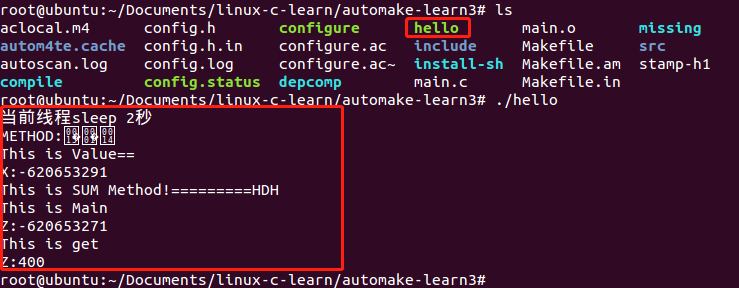




1. make命令

第七步，执行make命令。make执行后，会生成hello的可执行文件。





1. autotools运行流程
2. 流程总结：
3. 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。
4. 修改configure.scan为configure.ac文件，并且修改配置内容。
5. 执行aclocal命令。扫描 configure.ac 文件生成 aclocal.m4文件。
6. 执行autoconf命令。这个命令将 configure.ac 文件中的宏展开，生成 configure 脚本。
7. 执行autoheader命令。该命令生成 config.h.in 文件。
8. 新增Makefile.am文件，修改配置内容
9. 执行automake --add-missing命令。该命令生成 Makefile.in 文件。
10. 执行 ./congigure命令。将Makefile.in命令生成Makefile文件。
11. 执行make命令。生成可执行文件。
12. make命令详解
13. make命令：编译文件。make命令主要通过Makefile文件生成可执行文件。
14. make clean命令。清楚编译的文件，包括目标文件\*.o和可执行文件。
15. 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。
16. make uninstall 命令，把目标文件从系统中卸载。
17. 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。
6. Makefile.am解读
7. 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。  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  noinst\_PROGRAMS：如果make install的时候不想被安装，可以使用noinst\_PROGRAMS命令。  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 |

1. 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。  库文件一般以lib\*.a或者lib\*.so来命名。  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。  lib\_LIBRARIES:当运行make intsall的时候，则会被安装到/usr/local/lib目录下。  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 |

1. 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。

1. 数据文件

|  |
| --- |
| data\_DATA = data1 data2 |

1. 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 |

1. 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。