Auto Make好文

from: http://grid.tsinghua.edu.cn/home/liulk/publish/computer/AutoMake.html

Auto Make

版本问题

不同的automake，autoconf，autoscan之间存在着一定的不兼容性。这里推荐 autoscan 2.59和 automake 1.9配合使用。如果你是debian或者ubuntu，那么可你可以简单按照下面的方式安装

```
apt-get install autoconf
apt-get remove automake1.4
apt-get install automake1.9
```

Auto Make 例子

现在google Makefile 和 automake就能找到一些文章。以一个Hello 程序描述为一个project生成Makefile的过程。

这个例子其实在 Info automake 里能看到。大家把它翻成中文的，不错。

但实际上按照这个例子来做的话，步骤都对，就是太简单，一些常用的设置需要写进去，但是没有提到，还是要自己info, google, try.

步骤总述

- autoscan 生成configure.scan .
- 在configure.scan基础上手动编辑，主要要添加的：

```
AM_INIT_AUTOMAKE(myprojectname , version)

AC_OUTPUT( 最后要生成的Makefile , 包括 子目录中的，中间用空格隔开) ,
          例如 AC_OUTPUT(Makefile  subdir/Makefile subdir1/Makefile)

AC_PROG_RANLIB (意义见第四条末尾)
```

aclocal
- autoconf 生成configure脚本。
- 就是在每个最后需要生成Makefile的目录中，写一个Makefile.am .

最上层的要写明

```
AUTOMAKE_OPTIONS = foreign
```

如果这个目录没有要编译的文件 ，只包含了子目录，则只写个

```
SUBDIRS  = dir1
```

就ok了。

例如我的工程，最上层只是包含了源码目录，于是就写了

```
AUTOMAKE_OPTIONS=foreign
SUBDIRS=src
```

如果有文件要编译，则要指明target 先。比如我的src目录底下既有文件，又有目录，而src的这层目录中的文件最后是要编译成一个 可执行文件，则src目录下的Makefile.am这么写。

```
bin_PROGRAMS= myprogram
SUBDIRS= sub1

 myprogram_SOURCES= \
   a.cpp\
  b.cpp\
# 要编译的源文件。这儿的_SOURCES是关键字
EXTRA_DIST= \
 a.h \
 b.h
# 不用编成.o，但生成target myprogram也需要给编译器处理的头文件放这里

       myprogram_LDADD = libsub1.a 这个_LDADD是关键字，
# 最后生成myprogram这个执行文件，还要link src/sub1这个目录中的内容编成的一个lib :libsub1.a，

myprogram_LDFLAGS = -lpthread -lglib-2.0 -L/usr/bin $(all_libraries)
# myprogram还要link系统中的动态so，以此类推，需要连自编译的so,也写到这个关键字 _LDFLAGS后面就好了。

AM_CXXFLAGS = -D_LINUX
# 传递给g++编译器的一些编译宏定义，选项，

INCLUDES=-IPassport -Isub1/ -I/usr/include/glib-2.0\
   -I/usr/lib/glib-2.0/include $(all_includes)
#   传递给编译器的头文件路径。
```

下面是sub1种生成lib的Makefile.am

```
noinst_LIBRARIES = libprotocol.a
# 不是生成可执行文件，而是静态库，target用noinst_LIBRARIES
libprotocol_a_SOURCES = \
 alib.cpp
EXTRA_DIST = mylib.h\
  alib.h
INCLUDES= -I../ $(all_includes)
AM_CXXFLAGS = -D_LINUX -DONLY_EPOLL -D_SERVER
```

ok ,最后补上AC_PROG_RANLIB涵义，如果要自己生成lib，然后link到最终的可执行文件中，则要加上这个宏，否则不用。

[讨论] 每个目录至少都要有一个target，或者是可执行文件或者是lib，似乎对目录的划分带来点局限。

比如我的目录结构如果是这样

```
./Src
./Src/sub1
./Src/sub2
```

而我想这样，sub1,sub2都没有target，目录划分只是为了区别代码的不同模块，然后把两个目录中编译出的中间文件一起link ，得到最后需要的myprogram 。

似乎在Src/Makefile.am中要这么写

```
myprogram_SOURCES = sub1/a.cpp \
   sub2/b.cpp
```

可以实现，但我没试,: P

当然和设成先编译出libsub1.a libsub2.a 最后Link得到myprogram 没有本质区别了。

- automake --add-missing

Ok，Makefile.in应该放到各个目录下了。

- 最后，运行configure脚本，生成 各个目录下的Makefile .....
- 再最后 ， make .....

新版的**automake**变化

软件版本间的不兼容总是会带来不少麻烦。

首先我对比了网上的一些文章和gnu的autoconf文档，发现差别很大，新版本和老版本有很多不同，当然大部分特性新版本还是兼容老版本的，令人郁闷的是automake的文档中对configure.in的描述还是针对autoconf老版本的。

软件版本:

```
autoscan 2.59
automake 1.9.5
参考文档:
automake 1.7.8 doc http://www.gnu.org/software/automake/manual/html_node/
autoconf 2.59 doc http://www.gnu.org/software/autoconf/manual/
```

□□□ 建立程序:

```
cd ~/src
mkdir hello
编辑文件hello.c
```

□□□ 运行autoscan，产生错误： autom4te: configure.ac: no such file or directory

不予理会，生成了文件configure.scan

cp configure.scan configure.ac

修改configure.ac的内容如下:

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

AC_PREREQ(2.59)
AC_INIT(hello, 1.0, beyondwdq@gmail.com)
AC_CONFIG_SRCDIR([configure.ac])
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

这里说明一下的是，老版本的autotools使用configure.in作为输入，后来新版本改为configure.ac，但configure.in也可以。

□□□ 运行aclocal,生成文件aclocal.m4
□□□ 运行autoheader,声称文件config.h.in
□□□ 编辑文件Makefile.am 内容如下:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

□□□ 运行automake --add-missing,提示少了文件README NEWS AUTHORS ChangeLog

```
touch README NEWS AUTHORS ChangeLog 建立这些文件
```

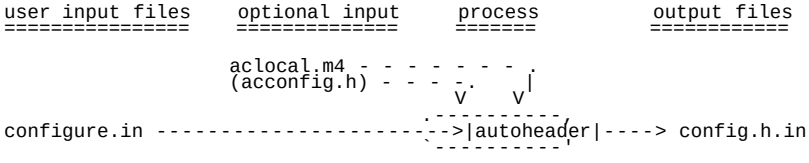
□□□ 再次运行automake --add-missing [注意] 若在Makefile.am中加上AUTOMAKE_OPTIONS= foreign 则不会提示缺少文件。
□□□ 运行autoconf, 生成了configure脚本
□□□ 执行./configure,生成Makefile
□□□ 运行make,编译通过。

autotools系列工具作用

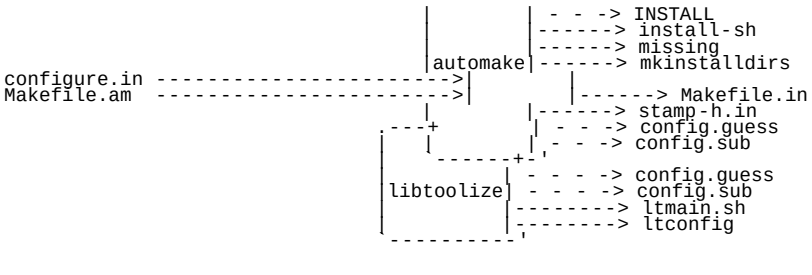
1.autoscan (autoconf): 扫描源代码以搜寻普通的可移植性问题，比如检查编译器，库，头文件等，生成文件configure.scan,它是configure.ac的一个雏形。

2.aclocal (automake):根据已经安装的宏，用户定义宏和acinclude.m4文件中的宏将configure.ac文件所需要的宏集中定义到文件aclocal.m4中。aclocal是一个perl 脚本程序，它的定义是: “aclocal - create aclocal.m4 by scanning configure.ac”

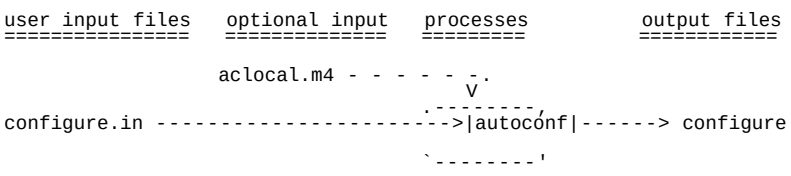
3.autoheader(autoconf): 根据configure.ac中的某些宏，比如cpp宏定义，运行m4，声称config.h.in



4.automake: automake将Makefile.am中定义的结构建立Makefile.in，然后configure脚本将生成的Makefile.in文件转换 为Makefile。如果在configure.ac中定义了一些特殊的宏，比如AC_PROG_LIBTOOL，它会调用libtoolize，否则它 会自己产生config.guess和config.sub



5.autoconf:将configure.ac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。



References: http://sourceware.org/autobook/autobook/autobook_276.html

怎么能让**automake**生成的**makefile**里面包含有我指定的库

```
INCLUDES = -I/include
LIBS = -lm -lcrypt
```

制作 **configure** 脚本

autoconf 是用来生成自动配置软件源代码脚本（configure）的工具。configure 脚本独立于 autoconf 运行，而且在运行的过程中，不需要用户的干预，通常不需要 附带参数。它是用来检验软件必须的参数的。autoconf 从一个列举编译软件时所 需要各种参数的模板文件中创建 configure。autoconf 需要 GNU m4 来生成该脚 本。

由 autoconf 生成的脚本一般起名为 configure。当运行时，configure 创建了多 个文件，并对这些文件中的配置参数赋予适当的值。由 configure 创建生成的文 件有:

- 一个或多个 Makefile，在软件源代码的每个目录中都生成一个 Makefile。
- 还可选的生成 C 头文件——configurable，包含了各种 #define 声明。
- 一个名为 config.status 的脚本，当运行时，重新生成上面的文件。
- 一个名为 config.cache 的脚本，保存运行检测的结果。
- 一个名为 config.log 的文件，保存有编译器生成的信息，用于调试 configure。

为了让 autoconf 生成 configure 脚本，需要以 configure.in 为参数调用 autoconf。如果要检测自己的各种参数，以作为对 autoconf 的补充，则需要写 aclocal.m4 和 acsite.m4 的文件。如果要使用 C 头文件，需要写 acconfig.h， 并且将 autoconf 生成的 config.h.in 同软件一起发行。

```
your source files --> [autoscan*] --> [configure.scan] --> configure.in

configure.in --. .------> autoconf* -----> configure

                +---+

[aclocal.m4] --+   `---.

[acsite.m4] ---'       |

                    +--> [autoheader*] -> [config.h.in]

[acconfig.h] ----.     |

                +-----'

[config.h.top] --+

[config.h.bot] --'

Makefile.in -------------------------------> Makefile.in

Files used in configuring a software package:

                     .--------------> config.cache

configure* ------------+-------------> config.log

                     |

[config.h.in] -.       v            .-> [config.h] -.

               +--> config.status* -+               +--> make*

Makefile.in ---'                    `-> Makefile ---'
```

编辑 configure.in 文件:

configure.in 文件中包含了对 autoconf 宏的调用，这些宏是用来检测软件所必 须的各项参数的。为了能够得到 configure.in 文件，需要使用 autoscan。 configure.in 文件中，在进行各项检测前，必须在最开始调用 AC_INIT，在最后调 用 AC_OUTPUT。另外有些宏由于检测的关系是和在文件中的位置相关的。最好每 一个宏占用一行。

使用 autoscan 创建 configure.in 文件

可以将目录做为参数调用 autoscan，如果不使用参数的化，则认为是当前目录。 autoscan 将检查指定目录中的源文件，并创建 configure.scan 文件。在将 configure.scan 改名为 configure.in 文件前，需要手工改动它以进行调整。

使用 autoconf 创建 configure 脚本

不带任何参数的运行 autoconf。autoconf 将使用 m4 宏处理器和 autoconf 宏，来 处理处理 configure.in 中的宏。

configure.in 中的宏:

AC_INIT（在源代码中唯一的一个文件）: configure 将检查该文件是否存在，并检查包含它的目录是否存在。

AC_OUTPUT（文件）: 指定创建的输出文件。在 configure.in 文件中调用一次。文件名间用空格分开。比如: AC_OUTPUT(Makefile:templates/top.mk lib/Makefile:templates/lib.mk)

在 configure.in 中，有一些被 autoconf 宏预先定义的变量，重要的有如下几个:

bindir: 安装可执行文件的目录。

includedir: C 头文件目录。

infodir: info 页安装目录。

mandir: 安装手册页的目录。

sbindir: 为管理员运行该该程序提供的安装路径。

srcdir: 为 Makefile 提供的源代码路径。

top_srcdir: 源代码的最上层目录。

LIBS: 给连接程序的 -l 选项

LDFLAGS: 给连接程序的 stripping（-s）和其他一些选项。

DEFS: 给 C 编译器的 -D 选项。

CFLAGS: 给 C 编译器的 debug 和优化选项。当调用了 AC_PROG_CC 才有效。

CPPFLAGS: 头文件搜索路径（-I)和给 C 预处理器和编译器的其他选项。

CXXFLAGS: 给 C++ 编译器的 debug 和优化选项。当调用了 AC_PROG_CXX 才有效。

如果在同一个目录下编译多个程序的话，使用 AC_CONFIG_SUBDIRS 宏，它的语法是:

AC_CONFIG_SUBDIRS(DIR....):

其他重要的宏:

AC_PROG_CC: 选择 C 编译器。如果在环境中不设置 CC 的话，则检测 gcc。
AC_PROG_CXX: 选择 C++ 编译器。

使用 **automake**

一般操作

Automake 工作时，读取一个叫'Makefile.am'的文件，并生成一个'Makefile.in'文件。Makefile.am中定义的宏和目标,会指导automake生成指定的代码。例如，宏'bin_PROGRAMS'将导致编译和连接的目标被生成。

Makefile.am中包含的目标和定义的宏被拷贝到生成的文件中去，这允许你添加任意代码到生成的Makefile.in文件中去。例如，使一个Automake发布中包含一个非标准的dvs-dist目标，Automake的维护者用它来从它的源码控制系统制作一个发布。

请注意，GNU生成的扩展名不被Automake所识别，在一个'Makefile.am'中使用这样一个扩展名会导致错误。

Automake试图以一种聪明的方式将相邻的目标（或变量定义）注释重组。

通常,Makefile.am中定义的目标会覆盖任何由automake自动生成的有相似名字的这样的目标。尽管这是种被支持的属性，但最好避免这么做，因为有些时候，生成的规则很严格。

类似的，Makefile.am中定义的变量会覆盖任何由automake自动生成的变量定义。这一特性经常要比目标定义的覆盖能力更常用。请注意，很多automake生成的变量只用于内部使用，在将来发布时他们的名字可能会变化。

当测试一个变量定义时，Automake降递归的测试在定义中引用的变量。例如，如果Automake看到这段snippet程序中的'foo_SOURCES':

```
xs = a.c b.c

foo_SOURCES = c.c $(xs)
```

它将使用文件: 'a.c','b.c'和'c.c' 作为foo_SOURCES的内容.

Automake 也允许不被拷贝到输出的注释形式,所有以'##'开头的行将被Automake 完全忽略.

深度

Automake 支持三种目录层次:'flat', 'shallow', 'deep'.

flat: 所有的文件都在一个目录中. 相应的Makefile.am中缺少SUBDIRS宏. termutils 是一个例子.

deep: 所有的资源都在子目录中,指定曾目录主要包含配置信息.GNU cpio 是一个很好的例子.GNU tar.相应的最顶层Makefile.am中将包含一个SUBDIR宏,但没有其他的宏来定义要创建的对象.

shallow: 主资源存在于最顶层目录,而不同的部分(典型的,库函数)在子目录中.Automake 就是这样的一个包.

严格性

当Automake 被GNU包维护者使用时,它的确努力去适应,但不要试图使用所有的GNU惯例.

目前,Automake 支持三种严格性标准:

foreign:Automake 将只检查绝对必须的东西.

gnu:Automake 将尽可能多的检查以适应GNU标准, 这是默认项.

gnits:Automake 将进行检查, 以适应“尚未成文”的Gnits标准。 他们基于GNU标准, 但更详尽。除非您是Gnits标准的制定者。建议您最好避免这个选项, 指导该标准正式发布。

统一命名规范

Automake变量一般遵循一套统一的命名规范以很容易的决定如何创建和安装程序（和其他派生对象）。给规范还支持configure时动态决定创建规则。

在make时, 一些变量被用于决定那些对象要被创建。写变量叫做primary variables。例如, PROGRAM变量包括一个要被编译和连接的程序列表。

另一个变量集用于决定被创建变量被安装到哪里。这些变量以相应的主变量命名, 但加一个前缀, 表示那些标准目录应被用作安装路径。这些标准目录的名称规定在GNU标准中。Automake用pkglibdir, pkgincludedir 和 pkgdatadir来展开这一列表。他们和没有pkg前缀的版本一样, 只不过有‘@PACHAGE@’扩展, PKGLIBDIR被定义为 $(DATADIR)/@PACKAGE@.

对每一个主变量, 有一个EXTRA_前缀的变量。这个变量用于列出所有对象, 至于哪些变量被创建, 哪些变量不被创建则取决于 configure。之所以需要这个变量, 是因为Automake必须静态的指导要创建对象的完整列表以便生成一个‘Makefile。in’文件。

例如, cpio 在configure时决定创建那些程序。一些程序被安装在bindir, 一些被安装在sbindir:

```
EXTRA_PROGRAMS = mt rmt

bin_PROGRAMS = cpoi pax

sbin_PRGRAMS = @PROGRAMS@
```

定义没有前缀的主变量是错误的（如: PROGRAMS）。值得注意的是, “dir”在作为构成变量名时会被忽略。一次, 我们写成bin_PROGRAMS 而不是bindir_PROGRAMS.

不是每一种对象都得以安装在每个目录下。Automake 将标记那些他认为是错误的尝试, 他也能够诊断一些明显的目录名拼写错误。

有时标准目录--被Automake使用的--不过用。特别的, 有时为了清晰, 将对象安装在一些预定义的子目录, 是很有用的。 Automake允许你增加安装目录。如果以一个变量名（如, zar）加上dir的另一个变量（如, zardir）被定义了, 那么他也是合法的。

例如, 如果HTML没支持Automake的一下部分, 你就可以用他来安装HTML源文件:

```
htmldir = $(prefix)/html

html_DATA = automake.html
```

“noinst”前缀专门指定有问题的对象不被安装。

“check”前缀表示有问题的对象知道make check命令被执行猜被创建。

可用的主变量是 'PROGRAMS','LIBRARIES','LISP','SCRIPTS','DATA','HEADERS','MANS'和'TEXINFOS' 导出变量是如何命名的

有时一个Makefile变量名有一些用户支持的文本导出。例如程序名被重写进Makefile宏名称。Automake读取这些文本, 所以他不必遵循命名规则。当生成后引用时名称中的字符除了字母, 数字, 下划线夺回被转换为下划线。例如, 如果你的程序里有sniff-glue,则导出变量名将会是sniff_glue_SOURCES,而不是sniff-glue_SOURCES.

一些例子

一个完整简单的例子

假设你写了一个名为zardoz的程序。

第一步, 更新你的configure.in文件以包含automake所需的命令。最简单的办法就是在AC_INIT后加一个AM_INIT_AUTOMAKE调用:

```
AM_INIT_AUTOMAKE(zardoz, 1.0)
```

如果你的程序没有任何复杂的因素。这是最简单的办法。

现在, 你必须重建‘configure’文件。这样做, 你必须告诉autoconf如何找到你所用的新宏。最简单的方式是使用 aclocal程序来 生成你的‘aclocal.m4’.aclocal让你将你的宏加进‘acincluide.m4’,所以你只需重命名并运行他

```
mv aclocal.m4 acinclude.m4

aclocal

autoconf
```

现在是为你的zardoz写Makefile.am的时候了。zardoz是一个用户程序, 所以逆向将他安装在其他用户程序安装的目录。zardoz还有一些Texinfo文档。你的configure.in脚本使用AC_REPLACE_FUNCS, 所以你需要链接‘@LIBOBJS@’

```
bin_PROGRAMS = zardoz

zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c

zardoz_LDADD = @LIBOBJS@

info_TEXINFOS = zardoz.texi
```

现在你可以运行Automake以生成你的Makefile.in文件。

一个经典的程序

hello 以其简单和多面幸而闻名。着一段将显示在Hello包Automake如何被使用。

下面是

dnl用 autoconf 处理它以产生一个 configure 脚本.

AC_INIT(src/hello.c)

AM_INIT_AUTOMAKE(hello, 1.3.11)

AM_CONFIG_HEADER(config.h)

dnl Set of available languages

ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.

AC_PROG_CC

AC_ISC_POSIX

dnl Checks for libraries.

dnl Checks for header files.

AC_STDC_HEADERS

AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.

AC_FUNC_ALLOCA

dnl Check for st_blksize in struct stat

AC_ST_BLKSIZE

dnl internationalization macros

AM_GNU_GETTEXT

AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \ src/Makefile tests/Makefile tests/hello], [chmod +x tests/hello])

'AM_'宏由Automake（或Gettext 库）提供；其余的是Autoconf标准宏。

top-level ‘Makefile.am’:

EXTRA_DIST = BUGS ChangeLog.O

SUBDIRS = doc intl po src tests

--如你所见,这里所有的工作时在子目录中真正完成的.

--'po' 和 'intl'目录是用 gettextize自动生成的,这里不做进一步讨论.

在'doc/Makefile.am'文件中我们看到:

info_TEXINFOS = hello.texi

hello_TEXINFOS = gpl.texi

--这已足以创建,安装和发布手册.

这里是'tests/Makefile.am'文件:

TESTS = hello

EXTRA_DIST = hello.in testdata

--脚本'hello'被configure创建,并且是唯一的测试.make check将运行它.

最后是,'src/Makefile.am',所有的真正的工作是在这里完成的:

bin_PROGRAMS = hello

hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h

hello_LDADD = @INTLLIBS@ @ALLOCA@

localedir = $(datadir)/locale

INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"

创建一个'Makefile.in'文件

要为一个包创建所有的'Makefile.in',在顶层目录运行automake不带参数的程序.automake将自动查找每个适当的'Makefile.am'文件,并声称相应的'Makefile.in'文件.请注意,automake会简要的察看一下包的构成;它假定一个包在顶层只有一个'configure.in'文件.如果你的包有多个'configure.in'文件,那么你必须在每一个含有 'configure.in'文件的目录运行一次'automake'程序.

你可以随意送给automake一个参数;'.am'会加到变量名的后面,并且作为输入文件的文件名.这一特性通常被用来自动重建一个'过时的''Makefile.in'文件.注意,automake必须在最顶级的目录运行,及时仅仅为了重新生成某个子目录中的 'Makefile.in'文件.

automake接受以下可选参数:

-a

--add-missing

在某些特殊情况下,Automake需要某些公共文件存在.例如,如果'configure.in'运行 AC_CANONICAL_HOST时,就需要'config.guess'.Automake需要几个这样的文件,这一选项会让automake自动的将一些缺失文件加进包中(如果可能的话).一般的,如果Automake告诉你某个文件不存在,你可以试一试这个选项.

--amdir = dir

在dir所制定的目录而不是安装目录下寻找Automake的数据文件,经常用于调试.

--build-dir = dir

告诉Automake创建目录在哪里.这一选项只用于将'dependencies'加进一个由make dist生成的'Makefile.in'文件中.

--cygnus

将使得所生成的'Makefile.in'文件遵循Cygnus规则,而不是GNU或Gnits规则.

--foreign

设置全局严格性为'gnits'.

--gnu

设置全局严格性为'gnu'.

--help

帮助

-i

--include-deps

将所有自动创建时的依赖信息包括在'Makefile.in'中.通常在制作发行时用到.

--generate-deps

创建一个'.dep_segment'文件,它合并了所有的自动生成时的依赖信息,通常用于制作发行.这在维护'SMakefile'或其他平台上的'Makefile'(如Makefile.Dos)时非常有用.它只能与--include-deps, --srcdir-name和 --build-dir 联合使用.注意这一选项不作其他任何处理.

--no-force

一般的,automake创建在'configure.in'中提及的所有'Makefile.in'文件.这一选项将导致,automake只更新那些对于与自身相关的东西过了时的'Makefile.in'文件.

-o dir

--output-dir = dir

将生成的'Makefile.in'放在指定的dir目录中.通常,'Makefile.in'放在与之相对应的'Makefile.am'所在目录中的.在制作发布时使用.

--srcdir-name = dir

告诉Automake与当前创建动作相关的源文件目录的名字.个选项仅被用于将自动创建时的依赖信息包括在'Makefile.in'中.

-v

--verbose

时Automake打印出正在读取或创建的文件的信息.

--version

打印Automake的版本号.

扫描'configure.in'文件

Automake扫描包的'configure.in'文件来决定某些关于包的信息.'configure.in'中必须定义一些变量并且需要一些autoconf宏.Automake也会利用'configure.in'文件的信息来定制它的输出.

Automake还支持一些autoconf宏以使维护更为简单.

配置需求

满足Automake基本要求的最简单方法是使用'AM_INIT_AUTOMAKE'宏.但如果你愿意,你可以手工来做.

自动生成 aclocale.m4

Automake包括很多Autoconf宏,其中有些在某些情况下游Automake使用,这些宏必须被定义在你的'aclocal.m4'中.否则,autoconf将看不到他们.

acloal程序将根据'configure.in'自动生成'aclocal.m4'.这提供了一个方便的途径来得到Automake提供的宏.

Automake所支持的Autoconf宏

AM_CONFIG_HEADER

AM_CYGWIN32

AM_FUNC_STRTOD

AM_FUNC_ERROR_AT_LINE

AM_FUNC_MKTIME

AM_FUNC_OBSTACK

AM_C_PROTOTYPES

AM_HEADER_TOCGWINSZ_NEEDS_SYS_IOCTL

AM_INIT_AUTOMAKE

AM_PATH_LISPDIR

AM_PROG_CC_STDC

AM_PROG_INTALL

AM_PROG_LEX

AM_SANITY_CHECK

AM_SYS_POSIX_TERMIOS

AM_TYPE_PTRDIFF_T

AM_WITH_DMALLOC

AM_WITH_REGEX

编写你自己的aclocal宏

Aclocal并没有特定的宏机制,因此你可以用你自己的宏来扩展它.这以特性经常被用以制作做那些想让自己的Autoconf宏被其他应用程序使用的库.例如,gettext库支持一个AM_GNU_GETTEXT宏,它可以被任何使用gettext库的包所使用.当该库被安装后,它会安装这个宏,这样,aclocal就能够找到他了.

宏的名称应以'.m4'结尾,这样的文件将被安装在'$(datadir)/aclocal'中.

最顶层'Makefile.am'

在non-flat包中,顶层'Makefile.am'必须告诉Automake哪些子目录将被创建.这是通过SUBDIRS变量定义的.

SUBDIRS宏包含一个子目录列表,以规定各种创建的过程.'Makefile'中很多目标(如,all)将不止运行所在目录中,还要运行于所有指定的子目录中.注意,在SUBDIRS中列出的目录并不需要包含'Makefile.am'文件而只需要'Makefile'文件. 这将允许包含位于不使用Automake的包(如,gettext)中的库.另外,SUBDIRS之中的目录必须是当前目录的直接子目录.比如,你不能在 SUBDIRS中指定'src/subdir'.

在deep型的包中,顶层'Makefile.am'文件通常非常短.例如,下面是HELLO发布中的'Makefile.am':

```
EXTRA_DIST = BUGS ChangeLog.O README-alpha

SUBDIRS = doc intl po src tests
```

如果你只想创建一个包的子集,你可以覆盖SUBDIRS(就如同GNU inetutils的情形)在你的'Makefile.am'中包括:

```
SUBDIRS = @SUBDIRS@
```

让后在你的'configure.in'中你可以指定:

```
SUBDIRS = "src doc lib po"

AC_SUBST(SUBDIRS)
```

这样做的结果是automake被欺骗了,它会在指定目录创建包,但并不真正绑定那个列表直到运行configure.

SUBDIRS可以包含配置的替代(如,'@DIRS@');Automake 自己并不真正检测这一变量的内容.

如果 SUBDIRS 被定义了,那么你的'configure.in' 就必须包含AC_PROG_MAKE_SET.

SUBDIRS 的使用并不局限于顶层'Makefile.am'.Automake 可被用来构造任意深度的包.

参考文献

Automake.htm

autoconf手册

对普通函数的检查
这些宏被用于寻找没有包括在特定函数测试宏中的函数。如果函数可能出现在除了缺省C库以外的库中，就要首先为这些库调用AC_CHECK_LIB。如果你除了需要检查函数是否存在之外，还要检查函数的行为，你就不得不为此而编写你自己的测试（参见编写测试）。
宏： AC_CHECK_FUNC (function, [action-if-found [, action-if-not-found]])

如果可以使用C函数function，就运行shell命令action-if-found，否则运行 action-if-not-found。如果你只希望在函数可用的时候定义一个符号，就考虑使用 AC_CHECK_FUNCS。由于C++比C更加标准化，即使在调用了AC_LANG_CPLUSPLUS 的时候，本宏仍然用C的连接方式对函数进行检查。（关于为测试选择语言的详情，请参见 对语言的选择）
宏： AC_CHECK_FUNCS (function... [, action-if-found [, action-if-not-found]])
对于每个在以空格分隔的函数列表function中出现的函数，如果可用，就定义HAVE_function （全部大写）。如果给出了action-if-found，它就是在找到一个函数的时候执行的附加的shell代码。你可以给出 `break'以便在找到第一个匹配的时候跳出循环。如果给出了action-if-not-found，它就在找不到某个函数的时候执行。
宏： AC_REPLACE_FUNCS (function...)
本宏的功能就类似于以将`function.o'添加到输出变量LIBOBJS的shell 代码为参数action-if-not-found，调用AC_CHECK_FUNCS。你可以通过用 `#ifndef HAVE_function'包围你为函数提供的替代版本的原型来声明函数。如果系统含有该函数，它可能在一个你应该引入的头文件中进行声明，所以你不应该重新声明它，以避免声明冲突。

头文件
下列宏检查某些C头文件是否存在。如果没有为你需要检查的头文件定义特定的宏，而且你不需要检查它的任何特殊属性，那么你就可以使用一个通用的头文件检查宏。

对特定头文件的检查
这些宏检查特定的系统头文件--它们是否存在，以及在某些情况下它们是否定义了特定的符号。
宏： AC_DECL_SYS_SIGLIST
如果在系统头文件，`signal.h'或者`unistd.h'，中定义了变量sys_siglist，就定义SYS_SIGLIST_DECLARED。
宏： AC_DIR_HEADER
类似于调用AC_HEADER_DIRENT和AC_FUNC_CLOSEDIR_VOID，但为了指明找到了哪个头文件而定义了不同的一组C预处理器宏。本宏和它定义的名字是过时的。它定义的名字是：

`dirent.h'
DIRENT
`sys/ndir.h'
SYSNDIR
`sys/dir.h'
SYSDIR
`ndir.h'
NDIR
此外，如果closedir不能返回一个有意义的值，就定义VOID_CLOSEDIR。

宏： AC_HEADER_DIRENT
对下列头文件进行检查，并且为第一个找到的头文件定义`DIR'，以及列出的C预处理器宏：

`dirent.h'
HAVE_DIRENT_H
`sys/ndir.h'
HAVE_SYS_NDIR_H
`sys/dir.h'
HAVE_SYS_DIR_H
`ndir.h'
HAVE_NDIR_H
源代码中的目录库声明应该以类似于下面的方式给出：

```
#if HAVE_DIRENT_H
# include
# define NAMLEN(dirent) strlen((dirent)->d_name)
#else
# define dirent direct
# define NAMLEN(dirent) (dirent)->d_namlen
# if HAVE_SYS_NDIR_H
#  include
# endif
# if HAVE_SYS_DIR_H
#  include
# endif
# if HAVE_NDIR_H
#  include
# endif
#endif
```

使用上述声明，程序应该把变量定义成类型struct dirent，而不是struct direct，并且应该通过把指向struct direct的指针传递给宏NAMLEN来获得目录项的名称的长度。

本宏还为SCO Xenix检查库`dir'和`x'。

宏： AC_HEADER_MAJOR
如果`sys/types.h'没有定义major、minor和makedev，但`sys/mkdev.h'定义了它们，就定义MAJOR_IN_MKDEV；否则，如果`sys/sysmacros.h'定义了它们，就定义MAJOR_IN_SYSMACROS。
宏： AC_HEADER_STDC
如果含有标准C（ANSI C）头文件，就定义STDC_HEADERS。特别地，本宏检查`stdlib.h'、`stdarg.h'、`string.h'和 `float.h'；如果系统含有这些头文件，它可能也含有其他的标准C头文件。本宏还检查`string.h'是否定义了memchr （并据此对其他mem函数做出假定），`stdlib.h'是否定义了free（并据此对malloc和其他相关函数做出假定），以及`ctype.h' 宏是否按照标准C的要求而可以用于被设置了高位的字符。

因为许多含有GCC的系统并不含有标准C头文件，所以用STDC_HEADERS而不是__STDC__ 来决定系统是否含有服从标准（ANSI-compliant）的头文件（以及可能的C库函数）。

在没有标准C头文件的系统上，变种太多，以至于可能没有简单的方式对你所使用的函数进行定义以使得它们与系统头文件声明的函数完全相同。某些系统包含了 ANSI和BSD函数的混合；某些基本上是标准（ANSI）的，但缺少`memmove'；有些系统在`string.h'或者`strings.h'中以宏的方式定义了BSD函数；有些系统除了含有`string.h'之外，只含有BSD函数；某些系统在`memory.h' 中定义内存函数，有些在`string.h'中定义；等等。对于一个字符串函数和一个内存函数的检查可能就够了；如果库含有这些函数的标准版，那么它就可能含有其他大部分函数。如果你在`configure.in'中安放了如下代码：

```
AC_HEADER_STDC
AC_CHECK_FUNCS(strchr memcpy)
```

那么，在你的代码中，你就可以像下面那样放置声明：

```
#if STDC_HEADERS
# include
#else
# ifndef HAVE_STRCHR
#  define strchr index
#  define strrchr rindex
# endif
char *strchr (), *strrchr ();
# ifndef HAVE_MEMCPY
#  define memcpy(d, s, n) bcopy ((s), (d), (n))
#  define memmove(d, s, n) bcopy ((s), (d), (n))
# endif
#endif
```

如果你使用没有等价的BSD版的函数，诸如memchr、memset、strtok 或者strspn，那么仅仅使用宏就不够了：你必须为每个函数提供一个实现。以memchr为例，一种仅仅在需要的时候（因为系统C库中的函数可能经过了手工优化）与你的实现协作的简单方式是把实现放入 `memchr.c'并且使用`AC_REPLACE_FUNCS(memchr)'。

宏: AC_HEADER_SYS_WAIT
如果`sys/wait.h'存在并且它和POSIX.1相兼容，就定义HAVE_SYS_WAIT_H。如果`sys/wait.h'不存在，或者如果它使用老式BSD union wait，而不是 int来储存状态值，就可能出现不兼容。如果`sys/wait.h'不与POSIX.1兼容，那就不是引入该头文件，而是按照它们的常见解释定义POSIX.1宏。下面是一个例子:

```
#include
#if HAVE_SYS_WAIT_H
# include
#endif
#ifndef WEXITSTATUS
# define WEXITSTATUS(stat_val) ((unsigned)(stat_val) >> 8)
#endif
#ifndef WIFEXITED
# define WIFEXITED(stat_val) (((stat_val) & 255) == 0)
#endif
```

宏: AC_MEMORY_H
在`string.h' 中，如果没有定义memcpy, memcmp等函数，并且`memory.h' 存在，就定义NEED_MEMORY_H。本宏已经过时；可以用AC_CHECK_HEADERS(memory.h)来代替。参见为 AC_HEADER_STDC提供的例子。
宏: AC_UNISTD_H
如果系统含有`unistd.h'，就定义HAVE_UNISTD_H。本宏已经过时；可以用 `AC_CHECK_HEADERS(unistd.h)'来代替。

检查系统是否支持POSIX.1的方式是:

```
#if HAVE_UNISTD_H
# include
# include
#endif

#ifdef _POSIX_VERSION
/* Code for POSIX.1 systems.  */
#endif
```

在POSIX.1系统中包含了`unistd.h'的时候定义_POSIX_VERSION。如果系统中没有`unistd.h'，那么该系统就一定不是POSIX.1系统。但是，有些非POSIX.1（non-POSIX.1）系统也含有`unistd.h'。

宏: AC_USG
如果系统并不含有`strings.h'、rindex、bzero等头文件或函数，就定义USG。定义USG就隐含地表明了系统含有`string.h'、strrchr、memset等头文件或函数。

符号USG已经过时了。作为本宏的替代，参见为AC_HEADER_STDC提供的例子。
对普通头文件的检查
这些宏被用于寻找没有包括在特定测试宏中的系统头文件。如果你除了检查头文件是否存在之外还要检查它的内容，你就不得不为此而编写你自己的测试（参见编写测试）。
宏: AC_CHECK_HEADER (header-file, [action-if-found [, action-if-not-found]])
如果系统头文件header-file存在，就执行shell命令action-if-found，否则执行action-if-not-found。如果你只需要在可以使用头文件的时候定义一个符号，就考虑使用 AC_CHECK_HEADERS。
宏: AC_CHECK_HEADERS (header-file... [, action-if-found [, action-if-not-found]])
对于每个在以空格分隔的参数列表header-file出现的头文件，如果存在，就定义 HAVE_header-file（全部大写）。如果给出了action-if-found，它就是在找到一个头文件的时候执行的附加shell代码。你可以把`break'作为它的值以便在第一次匹配的时候跳出循环。如果给出了action-if-not-found，它就在找不到某个头文件的时候被执行。

结构
以下的宏检查某些结构或者某些结构成员。为了检查没有在此给出的结构，使用AC_EGREP_CPP （参见检验声明）或者使用AC_TRY_COMPILE （参见检验语法）。
宏: AC_HEADER_STAT
如果在`sys/stat.h'中定义的S_ISDIR、S_ISREG等宏不能正确地工作（返回错误的正数），就定义 STAT_MACROS_BROKEN。这种情况出现在Tektronix UTekV、 Amdahl UTS和Motorola System V/88上。
宏: AC_HEADER_TIME
如果程序可能要同时引入`time.h'和`sys/time.h'，就定义 TIME_WITH_SYS_TIME。在一些老式系统中，`sys/time.h'引入了`time.h'，但`time.h'没有用多个包含保护起来，所以程序不应该显式地同时包含这两个文件。例如，本宏在既使用struct timeval或 struct timezone，又使用struct tm程序中有用。它最好和 HAVE_SYS_TIME_H一起使用，该宏可以通过调用AC_CHECK_HEADERS(sys/time.h)来检查。

```
#if TIME_WITH_SYS_TIME
# include
# include
#else
# if HAVE_SYS_TIME_H
#  include
# else
#  include
# endif
#endif
```

宏: AC_STRUCT_ST_BLKSIZE
如果struct stat包含一个st_blksize成员，就定义HAVE_ST_BLKSIZE。
宏: AC_STRUCT_ST_BLOCKS
如果struct stat包含一个st_blocks成员，就定义HAVE_ST_BLOCKS。否则，就把`fileblocks.o'添加到输出变量LIBOBJS中。
宏: AC_STRUCT_ST_RDEV
如果struct stat包含一个st_rdev成员，就定义HAVE_ST_RDEV。
宏: AC_STRUCT_TM
如果`time.h'没有定义struct tm，就定义TM_IN_SYS_TIME，它意味着引入`sys/time.h'将得到一个定义得更好的struct tm。
宏: AC_STRUCT_TIMEZONE
确定如何获取当前的时区。如果struct tm有tm_zone成员，就定义HAVE_TM_ZONE。否则，如果找到了外部数组tzname，就定义HAVE_TZNAME。

类型定义
以下的宏检查C typedefs。如果没有为你需要检查的typedef定义特定的宏，并且你不需要检查该类型的任何特殊的特征，那么你可以使用一个普通的typedef检查宏。

对特定类型定义的检查
这些宏检查在`sys/types.h'和`stdlib.h'（如果它存在）中定义的特定的C typedef。
宏: AC_TYPE_GETGROUPS
把GETGROUPS_T定义成getgroups的数组参数的基类型gid_t或者int。
宏: AC_TYPE_MODE_T
如果没有定义mode_t，就把mode_t定义成int。
宏: AC_TYPE_OFF_T
如果没有定义off_t，就把off_t定义成long。
宏: AC_TYPE_PID_T
如果没有定义pid_t，就把pid_t定义成int。
宏: AC_TYPE_SIGNAL
如果`signal.h'把signal声明成一个指向返回值为void的函数的指针，就把RETSIGTYPE定义成void；否则，就把它定义成int。

把信号处理器（signal handler）的返回值类型定义为RETSIGTYPE:

```
RETSIGTYPE
hup_handler ()
{
...
}
```

宏: AC_TYPE_SIZE_T

如果没有定义size_t，就把size_t定义成unsigned。
宏： AC_TYPE_UID_T
如果没有定义uid_t，就把uid_t定义成int并且把 gid_t定义成int。

对普通类型定义的检查
本宏用于检查没有包括在特定类型测试宏中的typedef。
宏： AC_CHECK_TYPE (type, default)
如果`sys/types.h'或者`stdlib.h'或者`stddef.h'存在，而类型 type没有在它们之中被定义，就把type定义成C（或者C++）预定义类型default；例如，`short'或者`unsigned'。

C编译器的特征
下列宏检查C编译器或者机器结构的特征。为了检查没有在此列出的特征，使用AC_TRY_COMPILE （参见检验语法）或者AC_TRY_RUN （参见检查运行时的特征）
宏： AC_C_BIGENDIAN
如果字（word）按照最高位在前的方式储存（比如Motorola和SPARC，但不包括Intel和VAX，CPUS），就定义 WORDS_BIGENDIAN。
宏： AC_C_CONST
如果C编译器不能完全支持关键字const，就把const定义成空。有些编译器并不定义 __STDC__，但支持const；有些编译器定义__STDC__，但不能完全支持 const。程序可以假定所有C编译器都支持const，并直接使用它；对于那些不能完全支持const的编译器，`Makefile'或者配置头文件将把const定义为空。
宏： AC_C_INLINE
如果C编译器支持关键字inline，就什么也不作。如果C编译器可以接受__inline__或者__inline，就把inline定义成可接受的关键字，否则就把inline定义为空。
宏： AC_C_CHAR_UNSIGNED
除非C编译器预定义了__CHAR_UNSIGNED__，如果C类型char是无符号的，就定义 __CHAR_UNSIGNED__。
宏： AC_C_LONG_DOUBLE
如果C编译器支持long double类型，就定义HAVE_LONG_DOUBLE。有些C编译器并不定义__STDC__但支持long double类型；有些编译器定义__STDC__但不支持long double。
宏： AC_C_STRINGIZE
如果C预处理器支持字符串化操作符（stringizing operator），就定义HAVE_STRINGIZE。字符串化操作符是 `#'并且它在宏定义中以如下方式出现：

#define x(y) #y
宏： AC_CHECK_SIZEOF (type [, cross-size])
把SIZEOF_uctype 定义为C（或C++）预定义类型type的，以字节为单位的大小，例如`int' or `char *'。如果编译器不能识别`type'，它就被定义为0。 uctype就是把type中所有小写字母转化为大写字母，空格转化成下划线，星号转化成`P' 而得到的名字。在交叉编译中，如果给出了cross-size，就使用它，否则configure就生成一个错误并且退出。

例如，调用
AC_CHECK_SIZEOF(int *)

在DEC Alpha AXP系统中，把SIZEOF_INT_P定义为8。

宏： AC_INT_16_BITS
如果C类型int是16为宽，就定义INT_16_BITS。本宏已经过时；更常见的方式是用 `AC_CHECK_SIZEOF(int)'来代替。
宏： AC_LONG_64_BITS
如果C类型long int是64位宽，就定义LONG_64_BITS。本宏已经过时；更常见的方式是用`AC_CHECK_SIZEOF(long)'来代替。

Fortran 77编译器的特征
下列的宏检查Fortran 77编译器的特征。为了检查没有在此列出的特征，使用AC_TRY_COMPILE （参见检验语法）或者AC_TRY_RUN （参见检验运行时的特征），但首先必须确认当前语言被设置成 Fortran 77 AC_LANG_FORTRAN77（参见对语言的选择）。
宏： AC_F77_LIBRARY_LDFLAGS
为成功地连接Fotran 77或者共享库而必须的Fortran 77内置函数（intrinsic）和运行库确定连接选项（例如，`-L'和`-l'）。输出变量FLIBS被定义为这些选项。

本宏的目的是用于那些需要把C++和Fortran 77源代码混合到一个程序或者共享库中的情况（参见GNU Automake中的`Mixing Fortran 77 With C and C++'节）。

例如，如果来自C++和Fortran 77编译器的目标文件必须被连接到一起，那么必须用C++编译器/连接器来连接（因为有些C++特定的任务要在连接时完成，这样的任务有调用全局构造函数、模板的实例化、启动例外（exception）支持，等等）。

然而，Fortran 77内置函数和运行库也必须被连接，但C++编译器/连接器在缺省情况下不知道如何添加这些 Fortran 77库。因此，就创建AC_F77_LIBRARY_LDFLAGS宏以确认这些Fortran 77库。
系统服务
下列宏检查操作系统服务或者操作系统能力。
宏： AC_CYGWIN
检查Cygwin环境。如果存在，就把shell变量CYGWIN设置成`yes'。如果不存在，就把CYGWIN设置成空字符串。
宏： AC_EXEEXT
根据编译器的输出，定义替换变量EXEEXT，但不包括.c、.o和.obj文件。对于Unix来说典型的值为空，对Win32来说典型的值为`.exe'或者`.EXE'。
宏： AC_OBJEXT
根据编译器的输出，定义替换变量OBJEXT，但不包括.c文件。对于Unix来说典型的值为`.o'，对Win32来说典型的值为`.obj'。
宏： AC_MINGW32
检查MingW32编译环境。如果存在，就把shell变量MINGW32设置成`yes'。如果不存在，就把MINGW32设置成空。
宏： AC_PATH_X
试图找到X Window系统的头文件和库文件。如果用户给出了命令行选项`--x-includes=dir'和 `--x-libraries=dir'，就使用这些目录。如果没有给出任一个选项，或者都没有给出，就通过运行xmkmf以处理一个测试 `Imakefile'，并且检查它所生成的`Makefile'，来得到没有给出的目录。如果这失败了（比如说，xmkmf不存在），就在它们通常存在的几个目录中寻找。如果任何一种方法成功了，就把shell变量x_includes和x_libraries设置成相应的位置，除非这些目录就在编译器搜索的缺省目录中。
如果两种方法都失败了，或者用户给出命令行选项`--without-x'，就把shell变量no_x 设置成`yes'；否则就把它设置成空字符串。

宏： AC_PATH_XTRA
AC_PATH_X的增强版。它把X需要的C编译器选项添加到输出变量X_CFLAGS，并且把 X的连接选项添加到X_LIBS。如果不能使用X系统，就把`-DX_DISPLAY_MISSING' 设置成X_CFLAGS。
本宏还检查在某些系统中为了编译X程序而需要的特殊库。它把所有系统需要的库添加到输出变量X_EXTRA_LIBS。并且它检查需要在`-lX11'之前被连接的特殊X11R6库，并且把找到的所有库添加到输出变量X_PRE_LIBS。
宏： AC_SYS_INTERPRETER
检查系统是否支持以形式为`#! /bin/csh'的行开头的脚本选择执行该脚本的解释器。在运行本宏之后，configure.in中的shell代码就可以检查shell变量 interpval；如果系统支持`#!'，interpval将被设置成`yes'，如果不支持就设置成`no'。
宏： AC_SYS_LONG_FILE_NAMES
如果系统支持长于14个字符的文件名，就定义HAVE_LONG_FILE_NAMES。
宏： AC_SYS_RESTARTABLE_SYSCALLS
如果系统自动地重新启动被信号所中断的系统调用，就定义HAVE_RESTARTABLE_SYSCALLS。

UNIX变种
下列宏检查对于有些程序来说需要特殊处理的一些操作系统，这是因为它们的头文件或库文件中含有特别怪异的东西。这些宏不讨人喜欢；它们将根据它们所支持的函数或者它们提供的环境，被更加系统化的方法所代替。
宏： AC_AIX
如果在AIX系统中，就定义_ALL_SOURCE。允许使用一些BSD函数。应该在所有运行C编译器的宏之前调用本宏。
宏： AC_DYNIX_SEQ
如果在Dynix/PTX (Sequent UNIX)系统中，就把`-lseq'添加到输出变量LIBS中。本宏已经过时；用AC_FUNC_GETMNTENT来代替。
宏： AC_IRIX_SUN
如果在IRIX(Silicon Graphics UNIX)系统中，就把`-lsun'添加到输出变量LIBS中。本宏已经过时。如果你用本宏来获取getmntent，就用AC_FUNC_GETMNTENT来代替。如果你为了口令（password）和组函数的NIS版本而使用本宏，就用`AC_CHECK_LIB (sun, getpwnam)'来代替。
宏： AC_ISC_POSIX
如果在POSIX化（POSIXized） ISC UNIX系统中，就定义_POSIX_SOURCE，并且把`-posix' （对于GNU C编译器）或者`-Xp'（对于其他C编译器）添加到输出变量CC中。本宏允许使用 POSIX工具。必须在调用AC_PROG_CC之后，在调用其他任何运行C编译器的宏之前，调用本宏。

宏: AC_MINIX
如果在Minix系统中，就定义_MINIX和_POSIX_SOURCE，并且把_POSIX_1_SOURCE 定义成2。本宏允许使用POSIX工具。应该在所有运行C编译器的宏之前调用本宏。
宏: AC_SCO_INTL
如果在SCO UNIX系统中，就把`-lintl'添加到输出变量LIBS。本宏已经过时；用AC_FUNC_STRFTIME来代替。
宏: AC_XENIX_DIR
如果在Xenix系统中，就把`-lx'添加到输出变量LIBS。还有，如果使用了`dirent.h'，就把`-ldir'添加到LIBS。本宏已经过时；用AC_HEADER_DIRENT来代替。

利用**libtool**自动生成动态库的**Makefile**的生成方法

```
#
# 利用libtool自动生成动态库
#
1. autoscan命令在当前目录生成configure.scan文件, 内容为:
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.57)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/bot.h])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CXX
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([limits.h malloc.h stdlib.h string.h unistd.h])
# Checks for typedefs, structures, and compiler characteristics.
AC_HEADER_STDBOOL
AC_C_CONST
AC_C_INLINE
# Checks for library functions.
AC_FUNC_MALLOC
AC_FUNC_REALLOC
AC_CHECK_FUNCS([memset strcasecmp strchr strdup])
AC_OUTPUT
将其该名为configure.ac 然后修改:
configure.ac 文件是 autoconf 的输入文件，经过 autoconf 处理，展开里面的 m4宏，
输出的是 configure 脚本。
第 4 行声明本文件要求的 autoconf 版本，因为本例使用了新版本 2.57，所以在此注明。
第 5 行 AC_INIT 宏用来定义软件的名称和版本等信息
AC_INIT([test], 1.0, [email]linhanzu@gmail.com[/email])
增加版本信息(为生成lib库做准备)
lt_major=1
lt_age=1
lt_revision=12
dist_version=0.1.12
AM_INIT_AUTOMAKE(test, $dist_version) //自动生成Makefile文件
增加宏, 打开共享库
AC_PROG_LIBTOOL
# Check for dl
DL_PRESENT=""
AC_CHECK_LIB( dl, dlopen, DL_PRESENT="yes",, $DL_LIBS -ldl )
if test "x$DL_PRESENT" = "xyes"; then
```

```
AC_DEFINE(HAVE_LIBDL, 1, [Define if DL lib is present])
DL_LIBS="-ldl"
AC_SUBST(DL_LIBS)
fi
# Check for libm
M_PRESENT=""
AC_CHECK_LIB( m, sin, M_PRESENT="yes",, $M_LIBS -lm )
if test "x$M_PRESENT" = "xyes"; then
AC_DEFINE(HAVE_LIBM, 1, [Define if libm is present])
M_LIBS="-lm"
AC_SUBST(M_LIBS)
fi
```

增加依赖库

```
# Check for pthread
PTHREAD_PRESENT=""
AC_CHECK_LIB( pthread, pthread_create, PTHREAD_PRESENT="yes",, $PTHREAD_LIBS
-lpthread )
if test "x$PTHREAD_PRESENT" = "xyes"; then
AC_DEFINE(HAVE_LIBPTHREAD, 1, [Define if libpthread is present])
PTHREAD_LIBS="-lpthread"
AC_SUBST(PTHREAD_LIBS)
fi
```

要生成项目工程目录和其它目录下的Makefile 文件, 必需加入

AM_CONFIG_FILES的宏:

例如: AC_CONFIG_FILES([Makefile

```
src/Makefile
data/Makefile
docs/Makefile])
```

修改完后Makefile.ac如下:

```
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.57)
AC_INIT([test],[1.0],[[email]arne_caspari@users.sourceforge.net[/email]])
AM_CONFIG_HEADER(config.h)
lt_major=1
lt_age=1
lt_revision=12
dist_version=0.1.12
AM_INIT_AUTOMAKE(test, $dist_version)
AC_SUBST(lt_major)
AC_SUBST(lt_revision)
AC_SUBST(lt_age)
# Checks for programs.
#AC_PROG_CC
#AC_PROG_INSTALL
#AC_PROG_LN_S
#AC_PROG_LIBTOOL
AM_PROG_LIBTOOL
# Checks for libraries.
pkg_modules="gtk+-2.0 >= 2.0.0"
```

```
PKG_CHECK_MODULES(GTK_PACKAGE, [$pkg_modules], HAVE_GTK2="yes", HAVE_GTK2="no" )
AC_SUBST(GTK_PACKAGE_CFLAGS)
AC_SUBST(GTK_PACKAGE_LIBS)
# Check for dl
DL_PRESENT=""
AC_CHECK_LIB( dl, dlopen, DL_PRESENT="yes",, $DL_LIBS -ldl )
if test "x$DL_PRESENT" = "xyes"; then
AC_DEFINE(HAVE_LIBDL, 1, [Define if DL lib is present])
DL_LIBS="-ldl"
AC_SUBST(DL_LIBS)
fi
# Check for libm
M_PRESENT=""
AC_CHECK_LIB( m, sin, M_PRESENT="yes",, $M_LIBS -lm )
if test "x$M_PRESENT" = "xyes"; then
AC_DEFINE(HAVE_LIBM, 1, [Define if libm is present])
M_LIBS="-lm"
AC_SUBST(M_LIBS)
fi
# Check for pthread （如示例中检测pthread，在生成的Makefile就会自动添加-lpthread）
PTHREAD_PRESENT=""
AC_CHECK_LIB( pthread, pthread_create, PTHREAD_PRESENT="yes",, $PTHREAD_LIBS
-lpthread )
if test "x$PTHREAD_PRESENT" = "xyes"; then
AC_DEFINE(HAVE_LIBPTHREAD, 1, [Define if libpthread is present])
PTHREAD_LIBS="-lpthread"
AC_SUBST(PTHREAD_LIBS)
fi
# Checks for header files.
#AC_HEADER_DIRENT
#AC_HEADER_STDC
#AC_CHECK_HEADERS([fcntl.h stdlib.h string.h sys/time.h unistd.h])
# Checks for typedefs, structures, and compiler characteristics.
#AC_TYPE_PID_T
#AC_TYPE_SIZE_T
#AC_HEADER_TIME
# Checks for library functions.
#AC_FUNC_CLOSEDIR_VOID
#AC_FUNC_MALLOC
#AC_CHECK_FUNCS([memset strstr])
AC_CONFIG_FILES([Makefile
src/Makefile
data/Makefile
doc/Makefile])
AC_OUTPUT
```

2.生成各目录下的Makefile.am文件

```
./Makefile.am   //工程目录下
SUBDIR = src data doc
../src/Makefile.am  //源码目录下
```

```
MAINTAINERCLEANFILES = Makefile.in
INCLUDES = -I../include
CPPFLAGS=-DINSTALL_PREFIX="\"$(prefix)\""
lib_LTLIBRARIES = libtest.la
libtest_la_LDFLAGS = -version-info @lt_major@:@lt_revision@:@lt_age@
libtest_la_SOURCES = \
          test.c \
          test_private.h \
          check_match.c \
          check_match.h \
          test_helpers.c \
          test_helpers.h \
          debug.h
libtest_la_LIBADD = \
             @DL_LIBS@ \
             @M_LIBS@
```

3. 生成autogen.sh脚本, 内容

```
#! /bin/sh
set -x
aclocal
autoheader
automake --foreign --add-missing --copy
autoconf
```

保存后修改权限 chmod a+x autogen.sh

3.运行脚本./autogen.sh, 生成configure脚本. 这里可能会遇到错误, 可以根据错误提示作相应修改.

4.运行./configure脚本.自动生成src目录下的makefile文件

5. 切换到目录src, 运行make 自动在当前目录下建立.libs文件, 编程生成的库文件就保存在该目录下.
    make install 安装在默认目录 /usr/local/lib/下.

6.如果要生成其它的安装目录, Makefile.am就要这样写

```
MAINTAINERCLEANFILES = Makefile.in
INCLUDES = -I../include
lib_LTLIBRARIES = libtt.la
libdir = $(prefix)/lib/test
libtt_la_LDFLAGS = -version-info @lt_major@:@lt_revision@:@lt_age@
libtt_la_LIBADD = @PTHREAD_LIBS@
libtt_la_SOURCES = \
               tt.c \
               video.c \
               video.h
```

autoconf 和**automake**生成**Makefile**文件

日期: 2006-09-22 作者: 杨小华、苏春艳 来自: IBM DW中国

本文介绍了在 linux 系统中，通过 Gnu autoconf 和 automake 生成 Makefile 的方法。主要探讨了生成 Makefile 的来龙去脉及其机理，接着详细介

绍了配置 Configure.in 的方法及其规则。

引子

无论是在Linux还是在Unix环境中，make都是一个非常重要的编译命令。不管是自 己进行项目开发还是安装应用软件，我们都经常要用到 make或make install。利 用make工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一 个包括几百个源文件的应用程序，使用make和makefile工具就可以轻而易举的理 顺各个源文件之间纷繁复杂的相互关系。

但是如果通过查阅make的帮助文档来手工编写Makefile,对任何程序员都是一场挑 战。幸而有GNU 提供的Autoconf及Automake这两套工具使得编写makefile不再是 一个难题。

本文将介绍如何利用 GNU Autoconf 及 Automake 这两套工具来协助我们自动产 生 Makefile文件，并且让开发出来的软件可以像大多数源码包那样，只需 "./configure", "make","make install" 就可以把程序安装到系统中。

模拟需求

假设源文件按如下目录存放，如图1所示，运用autoconf和automake生成makefile文件。

图 1文件目录结构

 j

假设src是我们源文件目录，include目录存放其他库的头文件，lib目录存放用到的库文件，然后开始按模块存放，每个模块都有一个对应的目 录，模块下再分子模块，如apple、orange。每个子目录下又分core，include，shell三个目录，其中core和shell目录存 放.c文件，include的存放.h文件，其他类似。

样例程序功能：基于多线程的数据读写保护（联系作者获取整个autoconf和automake生成的Makefile工程和源码，

工具简介

所必须的软件：autoconf/automake/m4/perl/libtool（其中libtool非必须）。

autoconf是一个用于生成可以自动地配置软件源码包，用以适应多种UNIX类系统的shell脚本工具，其中autoconf需要用到 m4，便于生成脚本。automake是一个从Makefile.am文件自动生成Makefile.in的工具。为了生成Makefile.in， automake还需用到perl，由于automake创建的发布完全遵循GNU标准，所以在创建中不需要perl。libtool是一款方便生成各种 程序库的工具。

目前automake支持三种目录层次：flat、shallow和deep。

1) flat指的是所有文件都位于同一个目录中。

就是所有源文件、头文件以及其他库文件都位于当前目录中，且没有子目录。Termutils就是这一类。

2) shallow指的是主要的源代码都储存在顶层目录，其他各个部分则储存在子目录中。

就是主要源文件在当前目录中，而其它一些实现各部分功能的源文件位于各自不同的目录。automake本身就是这一类。

3) deep指的是所有源代码都被储存在子目录中；顶层目录主要包含配置信息。

就是所有源文件及自己写的头文件位于当前目录的一个子目录中，而当前目录里没有任何源文件。 GNU cpio和GNU tar就是这一类。

flat类型是最简单的，deep类型是最复杂的。不难看出，我们的模拟需求正是基于第三类deep型，也就是说我们要做挑战性的事情：)。注：我们的测试程序是基于多线程的简单程序。

生成 **Makefile** 的来龙去脉

首先进入 project 目录，在该目录下运行一系列命令，创建和修改几个文件，就可以生成符合该平台的Makefile文件，操作过程如下：

1) 运行autoscan命令

2) 将configure.scan 文件重命名为configure.in，并修改configure.in文件

3) 在project目录下新建Makefile.am文件，并在core和shell目录下也新建makefile.am文件

4) 在project目录下新建NEWS、 README、 ChangeLog 、AUTHORS文件

5) 将/usr/share/automake-1.X/目录下的depcomp和complie文件拷贝到本目录下

6) 运行aclocal命令

7) 运行autoconf命令

8) 运行automake -a命令

9) 运行./confiugre脚本

可以通过图2看出产生Makefile的流程，如图所示：

图 2生成Makefile流程图

Configure.in的八股文

当我们利用autoscan工具生成confiugre.scan文件时，我们需要将confiugre.scan重命名为confiugre.in文件。confiugre.in调用一系列autoconf宏来测试程序需要的或用到的特性是否存在，以及这些特性的功能。

下面我们就来目睹一下confiugre.scan的庐山真面目:

```
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([config.h.in])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [main])
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_OUTPUT
```

每个configure.scan文件都是以AC_INIT开头，以AC_OUTPUT结束。我们不难从文件中看出confiugre.in文件的一般布局:

```
AC_INIT
  测试程序
  测试函数库
  测试头文件
  测试类型定义
  测试结构
  测试编译器特性
  测试库函数
  测试系统调用
AC_OUTPUT
```

上面的调用次序只是建议性质的，但我们还是强烈建议不要随意改变对宏调用的次序。

现在就开始修改该文件:

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

AC_PREREQ(2.59)
AC_INIT(test, 1.0, normalnotebook@126.com)
AC_CONFIG_SRCDIR([src/ModuleA/apple/core/test.c])
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(test,1.0)

# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [pthread_rwlock_init])
AC_PROG_RANLIB
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_OUTPUT([Makefile
               src/lib/Makefile
               src/ModuleA/apple/core/Makefile
               src/ModuleA/apple/shell/Makefile
               ])
```

其中要将AC_CONFIG_HEADER([config.h])修改为：AM_CONFIG_HEADER(config.h)，并加入AM_INIT_AUTOMAKE(test,1.0)。由于我们的测试程序是基于多线程的程序，所以要加入AC_PROG_RANLIB，不然运 行automake命令时会出错。在AC_OUTPUT输入要创建的Makefile文件名。

由于我们在程序中使用了读写锁，所以需要对库文件进行检查，即AC_CHECK_LIB([pthread], [main])，该宏的含义如下:

其中，LIBS是link的一个选项，详细请参看后续的Makefile文件。由于我们在程序中使用了读写锁，所以我们测试pthread库中是否存在pthread_rwlock_init函数。

由于我们是基于deep类型来创建makefile文件，所以我们需要在四处创建Makefile文件。即：project目录下，lib目录下，core和shell目录下。

Autoconf提供了很多内置宏来做相关的检测，限于篇幅关系，我们在这里对其他宏不做详细的解释，具体请参看参考文献1和参考文献2，也可参看autoconf信息页。

实战**Makefile.am**

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。

表 1Makefile.am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:

表 2 Makefile.am中可用的全局变量

在Makefile.am中尽量使用相对路径，系统预定义了两个基本路径:

表 3Makefile.am中可用的路径变量

在上文中我们提到过安装路径，automake设置了默认的安装路径：

1) 标准安装路径

默认安装路径为：$(prefix) = /usr/local，可以通过./configure --prefix=的方法来覆盖。

其它的预定义目录还包括：bindir = $(prefix)/bin, libdir = $(prefix)/lib, datadir = $(prefix)/share, sysconfdir = $(prefix)/etc等等。

2) 定义一个新的安装路径

比如test，可定义testdir = $(prefix)/test，然后test_DATA test1 test2，则test1，test2会作为数据文件安装到$(prefix)/ /test目录下。我们首先需要在工程顶层目录下（即project/）创建一个Makefile.am来指明包含的子目录：

```
"example">SUBDIRS=src/lib src/ModuleA/apple/shell src/ModuleA/apple/core
CURRENTPATH=$(shell /bin/pwd)
INCLUDES=-I$(CURRENTPATH)/src/include -I$(CURRENTPATH)/src/ModuleA/apple/include
export INCLUDES
```

由于每个源文件都会用到相同的头文件，所以我们在最顶层的Makefile.am中包含了编译源文件时所用到的头文件，并导出，见蓝色部分代码。

我们将lib目录下的swap.c文件编译成libswap.a文件，被apple/shell/apple.c文件调用，那么lib目录下的Makefile.am如下所示：

```
noinst_LIBRARIES=libswap.a
libswap_a_SOURCES=swap.c
INCLUDES=-I$(top_srcdir)/src/includ
```

细心的读者可能就会问：怎么表1中给出的是bin_LIBRARIES，而这里是noinst_LIBRARIES？这是因为如果只想编译，而不想 安装到系统中，就用noinst_LIBRARIES代替bin_LIBRARIES，对于可执行文件就用noinst_PROGRAMS代替 bin_PROGRAMS。对于安装的情况，库将会安装到$(prefix)/lib目录下，可执行文件将会安装到${prefix}/bin。如果想安 装该库，则Makefile.am示例如下：

```
bin_LIBRARIES=libswap.a
libswap_a_SOURCES=swap.c
INCLUDES=-I$(top_srcdir)/src/include
swapincludedir=$(includedir)/swap
swapinclude_HEADERS=$(top_srcdir)/src/include/swap.h
```

最后两行的意思是将swap.h安装到${prefix}/include /swap目录下。

接下来，对于可执行文件类型的情况，我们将讨论如何写Makefile.am？对于编译apple/core目录下的文件，我们写成的Makefile.am如下所示：

```
noinst_PROGRAMS=test
test_SOURCES=test.c
test_LDADD=$(top_srcdir)/src/ModuleA/apple/shell/apple.o $(top_srcdir)/src/lib/libswap.a
test_LDFLAGS=-D_GNU_SOURCE
DEFS+=-D_GNU_SOURCE
#LIBS=-lpthread
```

由于我们的test.c文件在链接时，需要apple.o和libswap.a文件，所以我们需要在test_LDADD中包含这两个文件。对于 Linux下的信号量/读写锁文件进行编译，需要在编译选项中指明-D_GNU_SOURCE。所以在test_LDFLAGS中指明。而 test_LDFLAGS只是链接时的选项，编译时同样需要指明该选项，所以需要DEFS来指明编译选项，由于DEFS已经有初始值，所以这里用+=的形 式指明。从这里可以看出，Makefile.am中的语法与Makefile的语法一致，也可以采用条件表达式。如果你的程序还包含其他的库，除了用 AC_CHECK_LIB宏来指明外，还可以用LIBS来指明。

如果你只想编译某一个文件，那么Makefile.am如何写呢？这个文件也很简单，写法跟可执行文件的差不多，如下例所示：

```
noinst_PROGRAMS=apple
apple_SOURCES=apple.c
DEFS+=-D_GNU_SOURCE
```

我们这里只是欺骗automake，假装要生成apple文件，让它为我们生成依赖关系和执行命令。所以当你运行完automake命令后，然后修改apple/shell/下的Makefile.in文件，直接将LINK语句删除，即：

```
……
clean-noinstPROGRAMS:
        -test -z "$(noinst_PROGRAMS)" || rm -f $(noinst_PROGRAMS)
apple$(EXEEXT): $(apple_OBJECTS) $(apple_DEPENDENCIES)
        @rm -f apple$(EXEEXT)
#$(LINK) $(apple_LDFLAGS) $(apple_OBJECTS) $(apple_LDADD) $(LIBS)
…….
```

通过上述处理，就可以达到我们的目的。从图1中不难看出为什么要修改Makefile.in的原因，而不是修改其他的文件。

原文链接：http://www-128.ibm.com/developerworks/cn/linux/l-makefile/

几个重要的宏

几个重要的宏 从前面可以看出重点有两个：修改configure.in和编辑Makefile.am 事实上，它们都是用一些宏来获取相关信息。重要的宏有：

configure.in中

- AC_INIT([xxx],[yyy],[zzz],[aaa])
  可以有四个参数，前三个必须。第一个是软件包的名字，一个字符串 (可以用空格，不需用引号)；第二个是版本，如0.6；地三个是报告bug的email；地四个是make dist时创建的tar的名字。如果不填(前面的逗号也去掉)的话默认为xxx(但是会有变化，例如xxx为My hello则创建my-hello.tar.gz)
- AC_OUTPUT([xxx])
  指出运行configure后输出那些文件，一般就是每个文件夹下的Makefile，如AC_OUTPUT([Makefile, src/Makefile, doc/Makefile])
- AC_CHECK_LIB([xxx],[yyy],[zzz],[aaa])
  这个用来检查系统中是否安装了某个库。xxx是库名去掉前 面的lib后面的扩展名；yyy是库xxx中的任一函数名；zzz是当存在时做什么操作，建议留空([] 也去掉)，因为默认情况就是把该库加入到LIBS 变量中，即加入-lxxx，如果改了的话就不加了；aaa是当不存在时的操作，可以用AC_MSG_ERROR([xxx is needed])，它将在运行configure时打印“xxx is needed”并退出。
- AM_INIT_AUTOMAKE(xxx, yyy)
  可以不加参数(同时去掉括号)，xxx指包名(这里的包名不能有空格，就算用引号也不管用)，yyy指版本号

Makefile.am中

- AUTOMAKE_OPTIONS=xxx
  xxx为gnu，foreign，和gnits。指用什么风格的工程，如果是gnu则必须要自己写AUTHOR，NEWS等文件；一般用foreign，在automake时会给你照搬一套默认的AUTHOR，NEWS等文件。
- bin_PROGRAMS=xxx
  指定最后生成的可执行文件的名字，即Makefile的目标，不一定要和包名相同。指定了这个之后， make install会把该目标拷贝到prefix/bin目录下。很明显类似的有sbin_PROGRAMS,lib_LIBRARIES, sysconf_DATA,man_MANS等，这个在《GNU编码标准》中有列表，automake做了一些扩展，但是一些简单应用只要知道这几个就行 了；例如指定man_MANS=hello.man.3则会把该文件安装到prefix/man目录下，指定sysconf_DATA= hello.conf就会把配置文件放到prefix/etc目录下。xxx_SOURCES=yyy 这里指定所有与xxx有关的源文件。注意，不要把其他目录下的文件放进去。例如，如果xxx用到了../comm目录下的debug.c，不要加到后面， 这样会在本目录下产生debug.o文件。标准做法是把../comm中的东西做成libcomm.a，然后用xxx_LDADD=.. /comm/libcomm.a加入。
- xxx_LDADD=path/libyyy.a
  如果xxx用到了某个lib，则用这个来指定。如果想指定全局lib则直接用LDADD＝就行了。

三种一般需求

源代码的目录结构一般有三种：flat型，即所有的文件都在一个目录下；deep型，即顶层目录没有源文件，源文件分装在子目录如src，doc，test等；shallow型，即顶层目录中也有源文件，但大部分源文件在子目录中，例如lib，include等

- 第一种： 前面已经提到。
- 第二种： 目录结构如下：其中src中需要用到comm中的东西

```
test
|-- comm
| |-- debug.c
| `-- debug.h
|-- doc
| |-- test.conf
| `-- test.man.3
`-- src
|-- test.c
`-- test.h
```

□□□ 进入test, 运行autoscan得到configure.scan
□□□ mv configure.scan configure.in,不一样的改动是

```
AC_OUTPUT([Makefile comm/Makefile src/Makefile doc/Makefile])
AC_PROG_RANLIB 因为用了lib
```

□□□ 运行aclocal
□□□ 运行autoconf
□□□ 为每个需要Makefile的文件夹创建文件Makefile.am。这里是Makefile.am,comm/Makefile.am, src/Makefile.am,doc/Makefile.am
顶层的Makefile.am内容如下

```
SUBDIRS = doc comm src test 这里注意把comm放在src前面
```

comm/Makefile.am内容如下:

```
noinst_LIBRARIES=libcomm.a noinst指的是该库不要install到prefix/lib目录下，因为只是一个临时的
libcomm_a_lib=debug.h debug.c 注意命名
```

src/Makefile.am内容如下

```
bin_PROGRAMS=test
hello_SOURCES=test.h test.c
```

doc/Makefile.am内容如下

```
man_MANS=test.man.3
sysconf_DATA=test.conf
```

□□□ autoheader
□□□ automake --add-missing --copy
□□□ 完成。
需要说明的是，如果还有一个目录lib里装的是一些用于做成libtest.a的文件，而且libtest.a还用到了comm中 的东西，这时不能在lib/Makefile.am中使用libtest_a_LIBADD=../comm/libcomm.a。这样回到之 libtest.a中有未解析符号，应该用libtest_a_LIBADD=../comm/debug.o(原因不明，但感觉不应该是这样的，这样太 土了)

- 第三种： 感觉于第二种没有什么区别——直接把src目录中的东西放到顶层目录就是了。但我想应该不是如此简单。

Updated: 2007-01-04 Home / Index

原文地址 http://grid.tsinghua.edu.cn/home/liulk/publish/computer/AutoMake.html

验证码： 4JBD 匿名

提交