

使用 Automake,Autoconf 生成 Makefile

使用 Automake,Autoconf 生成 Makefile

在 Unix 上写程序的人尤其是用 C 来开发程序的一般都遇到过 Makefile，用 make 来开发和编译程序的确很方便，可是要写出一个 Makefile 就不那么简单了。GNU Make 那份几百页的文件，让许多人害怕。当然，现在关于 make 的文档比较多，不过写一个 Makefile 总是一件很烦人的事情，GNU Autoconf 及 Automake 这两个软件就是帮助程序开发者轻松产生 Makefile 文件的。现在的 GNU 软件如 Apache, MySQL Minigui 等都是利用 Autoconf, Automake 实现自动编译的。用户只要使用 “./configure”, “make”, “make install” 就可以把程序安装到系统中。

简介

Makefile 基本上就是『目标』(target), 『关联』(dependencies) 和『动作』三者所组成的一系列规则。而 make 就是根据 Makefile 的规则决定如何编译 (compile) 和连接 (link) 程序或者其它动作。当然, make 可做的不只是编译和连接程序, 例如 FreeBSD 的 port collection 中, Makefile 还能做到自动下载远程程序, 解压缩 (extract), 打补丁 (patch), 设定, 然后编译, 安装到系统中。

Makefile 基本结构虽然很简单, 但是妥善运用这些规则就可以变换出许多不同的花样。却也因为这样, 许多人刚开始学写 Makefile 时会觉得没有规范可以遵循, 每个人写出来的 Makefile 都不大一样, 不知道从哪里下手, 而且常常会受到开发环境的限制, 只要环境参数不同或者路径更改, 可能 Makefile 就得跟着修改。虽然有 GNU Makefile Conventions (GNU Makefile 惯例) 制订出一些在进行 GNU 程序设计时写 Makefile 的一些标准和规范, 但是其内容很长而且很复杂, 并且经常作一些调整, 为了减轻程序开发人员维护 Makefile 的负担, 就出现了 Automake。

利用 Automake, 编程者只需要写一些预先定义好的宏 (macro), 提交给 Automake 处理, 就会产生一个可供 Autoconf 使用的 Makefile.in 文件。再配合使用 Autoconf 产生的自动配置文件 configure 即可产生一份符合 GNU Makefile 惯例的 Makefile 了。

需要的软件

在开始使用 Automake 之前, 首先确认你的系统安装有如下软件:

1. GNU Automake
2. GNU Autoconf
3. GNU m4
4. perl
5. GNU Libtool (如果你需要产生 shared library)

最好也使用 GNU C/C++ 编译器、GNU Make 以及其它 GNU 的工具程序来作为开发的环境, 这些工具都是属于 Open Source Software 不但免费而且功能强大。如果你是使用 Red Hat Linux 可以找到所有上述软件的 rpm 文件。

一个简单的例子

Automake 所产生的 Makefile 除了可以做到程序的编译和连接, 也可以用来生成文档(如 manual page, info 文件

等), 还可以有把源码文件包装起来以供发布, 所以程序源代码所存放的目录结构最好符合 GNU 的标准惯例, 接下来就用一个 `hello.c` 来做为例子。

在工作目录下建立一个新的子目录 `devel`, 再在 `devel` 下建立一个"hello"的子目录, 这个目录将作为存放 `hello` 这个程序及其相关文件的地方:

```
% mkdir devel;cd devel;mkdir hello;cd hello
```

用编辑器写一个 `hello.c` 文件,

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

```
int main(int argc, char** argv)
```

```
{
```

```
printf("Hello, GNU!\n");
```

```
return 0;
```

```
}
```

接下来就要用 `Autoconf` 及 `Automake` 来产生 `Makefile` 文件了,

1. 用 `autoscan` 产生一个 `configure.in` 的原型, 执行 `autoscan` 后会产生一个 `configure.scan` 的文件, 可以用它作为 `configure.in` 文件的蓝本。

```
% autoscan
```

```
% ls
```

```
configure.scan hello.c
```

2. 编辑 `configure.scan` 文件, 如下所示, 并且改名为 `configure.in`

```
dnl Process this file with Autoconf to produce a configure script.
```

```
AC_INIT(hello.c)
```

```
AM_INIT_AUTOMAKE(hello, 1.0)
```

```
dnl Checks for programs.
```

```
AC_PROG_CC
```

```
dnl Checks for libraries.
```

```
dnl Checks for header files.
```

```
dnl Checks for typedefs, structures, and compiler characteristics.
```

```
dnl Checks for library functions.
```

```
AC_OUTPUT(Makefile)
```

3. 执行 `aclocal` 和 `Autoconf`, 分别会产生 `aclocal.m4` 及 `configure` 两个文件

```
% aclocal
```

```
% Autoconf
```

```
% ls
```

```
aclocal.m4 configure configure.in hello.c
```

4. 编辑 `Makefile.am` 文件, 内容如下

```
AUTOMAKE_OPTIONS= foreign
```

```
bin_PROGRAMS= hello
```

```
hello_SOURCES= hello.c
```

5. 执行 `Automake --add-missing`, `Automake` 会根据 `Makefile.am` 文件产生一些文件, 包含最重要的 `Makefile.in`

```
% Automake --add-missing
```

```
Automake: configure.in: installing `./install-sh'
```

```
Automake: configure.in: installing `./mkinstalldirs'  
Automake: configure.in: installing `./missing'
```

注意，在执行automake --add-missing时可能会出现错误提示

required file `config.h.in' not found

可以在configure.in中找到以下宏，在前面加#，

#AC_CONFIG_HEADER([config.h])

也可以生成一个空的config.h.in

```
$touch config.h.in
```

6. 最后执行 ./configure:

```
% ./configure  
creating cache ./config.cache  
checking for a BSD compatible install... /usr/bin/install -c  
checking whether build environment is sane... yes  
checking whether make sets ${ MAKE} ... yes  
checking for working aclocal... found  
checking for working Autoconf... found  
checking for working Automake... found  
checking for working autoheader... found  
checking for working makeinfo... found  
checking for gcc... gcc  
checking whether the C compiler (gcc ) works... yes  
checking whether the C compiler (gcc ) is a cross-compiler... no  
checking whether we are using GNU C... yes  
checking whether gcc accepts -g... yes  
updating cache ./config.cache  
creating ./config.status  
creating Makefile
```

```
$ ls  
Makefile aclocal.m4 config.status hello.c mkinstalldirs  
Makefile.am config.cache configure install-sh
```

```
Makefile.in config.log configure.in missing
```

现在你的目录下已经产生了一个 Makefile 文件，输入 make 指令就可以编译 hello.c 了！

```
% make  
gcc -DPACKAGE="hello" -DVERSION="1.0" -I. -I. -g -O2 -c hello.c  
gcc -g -O2 -o hello hello.o
```

你还可以试试 “make clean”， “make install”， “make dist”：

```
[root@localhost hello]# make clean  
test -z "hello" || rm -f hello  
rm -f *.o core *.core  
[root@localhost hello]# make install  
gcc -DPACKAGE="hello" -DVERSION="1.0" -I. -I. -g -O2 -c hello.c  
gcc -g -O2 -o hello hello.o  
make[1]: Entering directory `/home/joe/devel/hello'  
/bin/sh ./mkinstalldirs /usr/local/bin  
/usr/bin/install -c hello /usr/local/bin/hello make[1]:
```

```
Nothing to be done for `install-data-am'. make[1]:  
Leaving directory `/home/joe/devel/hello'  
[root@localhost hello]# make dist
```

```

rm -rf hello-1.0 mkdir
hello-1.0 chmod 777
hello-1.0 here= ` cd .
&& pwd`;
top_distdir= ` cd hello-1.0 && pwd`;
distdir= ` cd hello-1.0 && pwd`;
cd .
&& Automake --include-deps --build-dir=$here --srcdir-name=. --output-dir=$top_distdir --foreign
Makefile
chmod -R a+r hello-1.0
GZIP=--best gtar chozf hello-1.0.tar.gz hello-1.0
rm -rf hello-1.0

```

一切工作得很好！当然，在 `make install` 时由于需要向系统目录拷贝文件，您需要有 `root` 权限。

更进一步

上述产生 `Makefile` 的过程和以往自行编写的方式非常不一样，使用 `Automake` 只需用到一些已经定义好的宏就可以了。我们把宏及目标 (`target`)写在 `Makefile.am` 文件内，`Automake` 读入 `Makefile.am` 文件后会把这一串已经定义好的宏展开并产生相对应的 `Makefile.in` 文件，然后再由 `configure` 这个 `shell script` 根据 `Makefile.in` 产生合适的 `Makefile`。

具体流程如下所示：

```

代码 --> [autoscans*] --> [configure.scan] --> configure.in
configure.in --. .----> Autoconf* -----> configure
+---+
[aclocal.m4] ---`---.
[acsuite.m4] ---' |
+--> [autoheader*] -> [config.h.in]
[acconfig.h] ----. |
+----'
[config.h.top] --+
[config.h.bot] --'

```

`Makefile.am` --# 61664; [Autoconf*] -----> `Makefile.in`

```

.-----> config.cache
configure* -----+-----> config.log
|
[config.h.in] .. v .-> [config.h] -.
+--> config.status* -+ +--> make*
Makefile.in ---`-> Makefile ---'

```

上图表示在整个过程中要使用的文件及产生出来的文件，有星号 (*) 代表可执行文件。在此示例中可由 `Autoconf` 及 `Automake` 工具所产生的额外文件有 `configure.scan`、`aclocal.m4`、`configure`、`Makefile.in`，需要加入设置的有 `configure.in` 及 `Makefile.am`。`开发者要书写的文件集中为 configure.in 和 Makefile.am`，在 `minigui` 项目中，我们把一系列的命令集中到一个批处理文件中：`autogen.sh`:

```
#!/bin/sh  
aclocal  
autoheader  
Automake --add-missing  
Autoconf
```

只要执行该批处理文件，结合 `configure.in` 和 `Makefile.am`，就可以生成需要的 `Makefile` 了。

编辑 `configure.in` 文件

`Autoconf` 是用来产生 '`configure`' 文件的工具。`'configure'` 是一个 `shell script`，它可以自动设定一些编译参数使程序能够条件编译以符合各种不同平台的 Unix 系统。`Autoconf` 会读取 `configure.in` 文件然后产生`'configure'` 这个 `shell script`。

`configure.in` 文件内容是一系列 GNU m4 的宏，这些宏经 `Autoconf` 处理后会变成检查系统特性的 `shell scripts`。`configure.in` 文件中宏的顺序并没有特别的规定，但是每一个 `configure.in` 文件必须在所有其它宏前加入 `AC_INIT` 宏，然后在所有其它宏的最后加上 `AC_OUTPUT` 宏。一般可先用 `autoscan` 扫描原始文件以产生一个 `configure.scan` 文件，再对 `configure.scan` 做些修改成 `configure.in` 文件。在例子中所用到的宏如下：

`dnl`

这个宏后面的内容不会被处理，可以视为注释

`AC_INIT(FILE)`

该宏用来检查源代码所在路径，`autoscan` 会自动产生，一般无须修改它。

`AM_INIT_AUTOMAKE(PACKAGE,VERSION)`

这是使用 `Automake` 所必备的宏，`PACKAGE` 是所要产生软件的名称，`VERSION` 是版本编号。

`AC_PROG_CC`

检查系统可用的 C 编译器，若源代码是用 C 写的就需要这个宏。

`AC_OUTPUT(FILE)`

设置 `configure` 所要产生的文件，若是 `Makefile`，`configure` 便会把它检查出来的结果填充到 `Makefile.in` 文件后产生合适的 `Makefile`。

实际上，在使用 `Automake` 时，还需要一些其他的宏，这些额外的宏我们用 `aclocal` 来帮助产生。执行 `aclocal` 会产生 `aclocal.m4` 文件，如果没有特别的用途，不需要修改它，用 `aclocal` 所产生的宏会告诉 `Automake` 如何动作。

有了 `configure.in` 及 `aclocal.m4` 两个文件以后，便可以执行 `Autoconf` 来产生 `configure` 文件了。

编辑 `Makefile.am` 文件

接下来要编辑 `Makefile.am` 文件，`Automake` 会根据 `configure.in` 中的宏并在 perl 的帮助下把 `Makefile.am` 转成 `Makefile.in` 文件。`Makefile.am` 文件定义所要产生的目标：

`AUTOMAKE_OPTIONS`

设置 `Automake` 的选项。`Automake` 主要是帮助开发 GNU 软件的人员来维护软件，所以在执行 `Automake` 时，

会检查目录下是否存在标准 GNU 软件中应具备的文件，例如 'NEWS'、'AUTHOR'、'ChangeLog' 等文件。设置为 `foreign` 时，`Automake` 会改用一般软件的标准来检查。

`bin_PROGRAMS`

定义要产生的执行文件名。如果要产生多个执行文件，每个文件名用空白符隔开。

`hello_SOURCES`

定义 'hello' 这个执行程序所需要的原始文件。如果 'hello' 这个程序是由多个原始文件所产生，必须把它所用到的所有原始文件都列出来，以空白符隔开。假设 'hello' 还需要 'hello.c'、'main.c'、'hello.h' 三个文件的话，则定义

`hello_SOURCES= hello.c main.c hello.h`

如果定义多个执行文件，则对每个执行程序都要定义相对的 `filename_SOURCES`。

编辑好 `Makefile.am` 文件，就可以用 `Automake --add-missing` 来产生 `Makefile.in`。加上 `--add-missing` 选项来告诉 `Automake` 顺便加入包装一个软件所必须的文件，如果你不使用该选项，`Automake` 可能会抱怨缺少了什么文件。`Automake` 产生的 `Makefile.in` 文件是完全符合 GNU `Makefile` 惯例的，只要执行 `configure` 这个 shell script 便可以产生合适的 `Makefile` 文件了。

使用 `Makefile`

利用 `configure` 所产生的 `Makefile` 文件有几个预先设定的目标可供使用，这里只用几个简述如下：

`make all`

产生设定的目标，既范例中的可执行文件。只敲入 `make` 也可以，此时会开始编译源代码，然后连接并产生执行文件。

`make clean`

清除之前所编译的可执行文件及目标文件(`object file, *.o`)。

`make distclean`

除了清除可执行文件和目标文件以外，也把 `configure` 所产生的 `Makefile` 清除掉。通常在发布软件前执行该命令。

`make install` 将程序安装到系统中，若源码编译成功，且执行结果正确，便可以把程序安装到系统预先设定的执行文件存放路径中，若用 `bin_PROGRAMS` 宏的话，程序会被安装到 `/usr/local/bin` 下。

`make dist`

将程序和相关的文档包装为一个压缩文档以供发布 (`distribution`)。执行完在目录下会产生一个以 `PACKAGE-VERSION.tar.gz` 为名称的文件。`PACKAGE` 和 `VERSION` 这两个参数是根据 `configure.in` 文中 `AM_INIT_AUTOMAKE(PACKAGE, VERSION)` 的定义。在我们的例子中会产生 '`hello-1.0.tar.gz`' 的文件。

`make distcheck`

和 `make dist` 类似，但是加入检查包装以后的压缩文件是否正常，这个目标除了把程序和相关文档包装成 `tar.gz` 文件外，还会自动把这个压缩文件解开，执行 `configure`，并执行 `make all`，确认编译无错误以后，方显示这个 `tar.gz` 文件已经准备好并可以发布了。当你看到：

=====

hello-1.0.tar.gz is ready for distribution

=====

就可以放心地发布您的软件了，检查过关的套件，基本上可以给任何具备 GNU 开发环境的人去重新编译成功。要注意的是，利用 Autoconf 及 Automake 所产生出来的软件套件是可以在没有安装 Autoconf 及 Automake 的环境使用的，因为 configure 是一个 shell script，它已被设计为可以在一般 Unix 的 sh 这个 shell 下执行。但是如果要修改 configure.in 及 Makefile.am 文件再产生新的 configure 及 Makefile.in 文件时就一定要有 Autoconf 及 Automake 了。

相关资料 通常我们掌握了一些入门知识就可以开始实践了，在有新的需求时，参照相关的文档和别人的例子解决问题，在实践中不断提高。

Autoconf 和 Automake 功能十分强大，可以从它们附带的 info 文档中找到详细的使用说明。或者您喜欢 html，可以从 gun 站点上下载 hmtl 版本。你也可以从许多现有的 GNU 软件或 Open Source 软件如 Minigui 中找到相关的 configure.in 或 Makefile.am 文件，他们是学习 Autoconf 及 Automake 更多技巧的最佳范例。

Another simple example

制作一个最简单的 helloworld 程序:

现有目录 test

mkdir src 建立 src 目录存放 源代码
在 src 下。

编辑 hello.c 文件

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

```
int main()
{
    printf("hello world\n");
    return 0;
}
```

src 目录下建立 Makefile.am 文件 (src/Makefile.am)
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS = hello
hello_SOURCES = hello.c
hello_LDADD = -lpthread (只是测试, 实际不需要连接该库)

保存退出

退到 test 目录

编辑 Makefile.am 文件 (Makefile.am)

SUBDIRS = src

退出保存

然后执行

autoscans

生成 configure.scan 文件

按此编辑此文件

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

AC_PREREQ(2.59)
AC_INIT(hello, 1.0, [miaoquan@nou.com.cn])
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([src/hello.c])
```

```
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_CONFIG_FILES([Makefile
#                 src/Makefile])
AC_OUTPUT(Makefile src/Makefile)
```

退出保存

将此文件更名 mv configure.scan configure.in

然后执行

touch NEWS README AUTHORS ChangeLog

然后执行

autoreconf -fvi

至此生成 `configure` 文件

执行 `configure` 文件

生成 `Makefile` 文件

```
make
make install
make uninstall
make dist
试验一下吧。
```

制作静态库文件

继续完善这个例子,论坛里有人问,如何生成静态库,并连接.

完善 `hello.c` 这个例子

当前目录

```
| - src 目录  
|   | - hello.c 文件  
| - include 目录  
|   | - hello.h 文件  
| - lib 目录  
|   | - test.c 文件 此文件用来生成 libhello.a
```

1. 在当前目录 编写 `Makefile.am`

```
SUBDIRS = lib src
```

在 `include` 目录下 编写 `hello.h`

```
extern void print(char *);
```

2. 在 `lib` 目录下编写 `test.c`

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

```
void print(char *msg)  
{  
    printf("%s\n",msg);  
}
```

在 `lib` 目录下编写 `Makefile.am`

```
noinst_LIBRARIES=libhello.a  
libhello_a_SOURCES=test.c
```

这里 `noinst_LIBRARIES` 的意思是生成的静态库 ,不会被 `make install` 安装
然后指定 `libhello.a` 的源文件 `test.c`

3. 在 `src` 目录下编写 `hello.c`

```
#include "hello.h"
```

```
int main()
```

```
{  
    print("haha"); //这里是静态库里的 print 函数  
    return 0;  
}
```

在 `src` 目录下编写 `Makefile.am`

```
INCLUDES= -I../include
```

```
bin_PROGRAMS=hello  
hello_SOURCES=hello.c  
hello_LDADD=../lib/libhello.a
```

首先指定头文件的位置 `../include` 然后
指定要生成执行文件 `hello` 然后指定源代
码文件 `hello.c` 最后添加静态库的位
置 `../lib/libhello.a`

4. 按照我首篇帖子的方式.

执行 `autoscans` 生成 `configure.scan`

修改该文件 按照首

篇帖子修改. 然后不

同之处

需要添加一行: `AC_PROG_RANLIB`

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

```
AC_PREREQ(2.59)  
AC_INIT(hello,1.1,[miaoquan@nou.com.cn])  
AM_INIT_AUTOMAKE  
AC_CONFIG_SRCDIR([src/hello.c])  
AC_CONFIG_HEADER([config.h])
```

```
# Checks for programs.  
AC_PROG_CC
```

```
# Checks for libraries.  
AC_PROG_RANLIB  
# Checks for header files.
```

```
# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

# AC_CONFIG_FILES([Makefile
#                   lib/Makefile
#                   src/Makefile])
AC_OUTPUT([Makefile
          lib/Makefile
          src/Makefile])
```

mv configure.scan configure.in

touch NEWS README AUTHORS ChangeLog

执行 autoreconf -fvi

生成 configure. 执行 configure 生成 Makefile..

后面同上...

make

make install

make uninstall

make dist

试验一下吧。

使用 **automake** 编译共享库的两种方法

用 **automake** 编译共享库有多种方法，**automake** 本身提供了对编译共享库支持，当然我们也可以不使用它的这个功能，下面分别介绍这两种方法：

一、不使用 **automake** 编译共享库功能

`configure.ac` 和 `Makefile.am` 的配置和可执行文件基本相同，仅仅 `Makefile.am` 中的变量名有些区别，另外需要再加上“`-fPIC -shared`”链接选项，如可以将“`bin_`”改成其它名字，如“`module_`”，这样就变成了：

```
LDFLAGS=-fPIC -shared  
moduledir=$(prefix)/lib # 请注意由于 module 不是 automake 标准的名称，所以需要自己加上安装目前  
module_PROGRAMS = libfoo.so  
libfoo_so_SOURCES = foo.c foo.h
```

二、使用 **automake** 编译共享库功能

automake 提供的编译共享库功能比较完善，支持同时编译出静态和共享两个，及带版号的多个版本，方法如下：

- 1、需要在 `configure.ac` 或 `configure.in` 文件中增加如下一句：

```
AC_PROG_LIBTOOL
```

- 2、在运行 `automake -a` 之前，需要执行

```
libtoolize -f -c
```

- 3、`Makefile.am` 的格式有点区别，如下：

```
lib_LTLIBRARIES = libfoo.la # 注意不是 libfoo.so  
libfoo_la_SOURCES = foo.cpp foo.h
```

这样编译成功之后，共享库将生成在`.libs` 目录下，包括如下一些文件：

```
-rw-r--r-- 1 jayyi users 11092 2007-06-08 16:48 libfoo.a  
lrwxr-xr-x 1 jayyi users   10 2007-06-08 16:48 libfoo.la -> ../libfoo.la  
-rw-r--r-- 1 jayyi users   989 2007-06-08 16:48 libfoo.lai  
lrwxr-xr-x 1 jayyi users   13 2007-06-08 16:48 libfoo.so -> libfoo.so.0.0.0*  
lrwxr-xr-x 1 jayyi users   13 2007-06-08 16:48 libfoo.so.0 -> libfoo.so.0.0.0*  
-rwxr-xr-x 1 jayyi users 13361 2007-06-08 16:48 libfoo.so.0.0.0*  
-rw-r--r-- 1 jayyi users 11044 2007-06-08 16:48 foo.o
```

利用 libtool 自动生成动态库的 Makefile 生成方法

```
#  
#  
# 利用 libtool 自动生成动态库  
#  
# -*- 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，这里就仅仅列举了 pthread 库，生成的 Makefile 会自动加上-pthread

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](#)—> [Configure.in??](#) 如下:

-*- Autoconf -*-

Process this file with autoconf to produce a configure script.

AC_PREREQ(2.57)

AC_INIT([test],[1.0],[[email]jarne_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
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_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. (注意: 如果您修改了 Makefile.am 中的项, 那么就得重新执行这一步)

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 //这个就是安装目录  
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
```

当然, Makefile 中的语法规则中还有很多宏定义, 您可以在 Makefile 的官方网站找到说明。
下一篇就打算写写条件编译的 Makefile 写法。

[/code]

Linux 下 Makefile 的 automake 生成全攻略

作为 Linux 下的程序开发人员，大家一定都遇到过 Makefile，用 make 命令来编译自己写的程序确实是很方便。一般情况下，大家都是手工写一个简单 Makefile，如果要想写出一个符合自由软件惯例的 Makefile 就不容易了。

在本文中，将给大家介绍如何使用 autoconf 和 automake 两个工具来帮助我们自动地生成符合自由软件惯例的 Makefile，这样就可以象常见的 GNU 程序一样，只要使用 “`./configure`” , “`make`” , “`make install`” 就可以把程序安装到 Linux 系统中去了。这将特别适合想做开放源代码软件的程序开发人员，又或如果你只是自己写些小的 Toy 程序，那么这个文章对你也会有很大的帮助。

一、Makefile 介绍

Makefile 是用于自动编译和链接的，一个工程有很多文件组成，每一个文件的改变都会导致工程的重新链接，但是不是所有的文件都需要重新编译，Makefile 中纪录有文件的信息，在 make 时会决定在链接的时候需要重新编译哪些文件。

Makefile 的宗旨就是：让编译器知道要编译一个文件需要依赖其他的哪些文件。当那些依赖文件有了改变，编译器会自动的发现最终的生成文件已经过时，而重新编译相应的模块。

Makefile 的基本结构不是很复杂，但当一个程序开发人员开始写 Makefile 时，经常会怀疑自己写的是否符合惯例，而且自己写的 Makefile 经常和自己的开发环境相关联，当系统环境变量或路径发生了变化后，Makefile 可能还要跟着修改。这样就造成了手工书写 Makefile 的诸多问题，automake 恰好能很好地帮助我们解决这些问题。

使用 automake，程序开发人员只需要写一些简单的含有预定义宏的文件，由 autoconf 根据一个宏文件生成 configure，由 automake 根据另一个宏文件生成 Makefile.in，再使用 configure 依据 Makefile.in 来生成一个符合惯例的 Makefile。下面我们将详细介绍 Makefile 的 automake 生成方法。

二、使用的环境

本文所提到的程序是基于 Linux 发行版本：Fedora Core release 1，它包含了我们要用到的 autoconf, automake,

三、从 helloworld 入手

我们从大家最常使用的例子程序 helloworld 开始。

下面的过程如果简单地说来就是：

新建三个文件：

```
helloworld.c
```

```
configure.in
```

```
Makefile.am
```

然后执行：

```
aclocal; autoconf; automake --add-missing; ./configure; make; ./helloworld
```

就可以看到 Makefile 被产生出来，而且可以将 helloworld.c 编译通过。很简单

吧，几条命令就可以做出一个符合惯例的 Makefile，感觉如何呀。现在开始介

绍详细的过程：

1、建目录

在你的工作目录下建一个 helloworld 目录，我们用它来存放 helloworld 程序及相关文件，如在 /home/my/build 下：

```
$ mkdir helloworld  
$ cd helloworld
```

2、 helloworld.c

然后用你自己最喜欢的编辑器写一个 helloworld.c 文件，如命令： vi helloworld.c。使用下面的代码作为 helloworld.c 的内容。

```
int main(int argc, char** argv)
{
    printf("Hello, Linux World!\n");
    return 0;
}
```

完成后保存退出。

现在在 helloworld 目录下就应该有一个你自己写的 helloworld.c 了。

3、生成 configure

我们使用 autoscan 命令来帮助我们根据目录下的源代码生成一个 configure.in 的模板文件。

命令：

```
$ autoscan
$ ls
configure.scan helloworld.c
```

执行后在 helloworld 目录下会生成一个文件：configure.scan，我们可以拿它作为 configure.in 的蓝本。

现在将 configure.scan 改名为 configure.in，并且编辑它，按下面的内容修改，去掉无关的语句：

```
=====configure.in 内容开始=====
# -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_INIT(helloworld)
AM_INIT_AUTOMAKE(helloworld 1.0)

# 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_OUTPUT(Makefile)  
=====configure.in 内容结束=====
```

然后执行命令 aclocal 和 autoconf，分别会产生 aclocal.m4 及 configure 两个文件：

```
$ aclocal  
$ ls  
aclocal.m4 configure.in helloworld.c  
$ autoconf  
$ ls  
aclocal.m4 autom4te.cache configure configure.in helloworld.c
```

大家可以看到 configure.in 内容是一些宏定义，这些宏经 autoconf 处理后会变成检查系统特性、环境变量、软件必须的参数的 shell 脚本。

autoconf 是用来生成自动配置软件源代码脚本 (configure) 的工具。configure 脚本能独立于 autoconf 运行，且在运行的过程中，不需要用户的干预。

要生成 configure 文件，你必须告诉 autoconf 如何找到你所用的宏。方式是使用 aclocal 程序来生成你的 aclocal.m4。

aclocal 根据 configure.in 文件的内容，自动生成 aclocal.m4 文件。aclocal 是一个 perl 脚本程序，它的定义是：“aclocal – create aclocal.m4 by scanning configure.ac”。

autoconf 从 configure.in 这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 宏处理器来处理 aclocal.m4，生成 configure 脚本。

m4 是一个宏处理器，将输入拷贝到输出，同时将宏展开。宏可以是内嵌的，也可以是用户定义的。除了可以展开宏，m4 还有一些内建的函数，用来引用文件，执行命令，整数运算，文本操作，循环等。m4 既可以作为编译器的前端，也可以单独作为一个宏处理器。

4、新建 Makefile.am

新建 Makefile.am 文件，命令：

```
$ vi Makefile.am
```

内容如下：

```
AUTOMAKE_OPTIONS=foreign  
bin_PROGRAMS=helloworld  
helloworld_SOURCES=helloworld.c
```

automake 会根据你写的 Makefile.am 来自动生成 Makefile.in

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。

5、运行 automake

命令：

```
$ automake --add-missing  
configure.in: installing `./install-sh'
```

```
configure: installing `./mkinstalldirs'
configure: installing `./missing'
Makefile.am: installing `./depcomp'
```

automake 会根据 `Makefile.am` 文件产生一些文件，包含最重要的 `Makefile.in`。

6、执行 `configure` 生成 `Makefile`

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
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 ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
$ ls -l Makefile
-rw-rw-r-- 1 yutao yutao 15035 Oct 15 10:40 Makefile
```

你可以看到，此时 `Makefile` 已经产生出来了。

7、使用 `Makefile` 编译代码

```
$ make

if gcc -DPACKAGE_NAME="" -DPACKAGE_TARNAME="" -DPACKAGE_VERSION="" -
DPACKAGE_STRING="" -DPACKAGE_BUGREPORT="" -DPACKAGE="helloworld" -DVERSION="1.0"

-L -L -g -O2 -MT helloworld.o -MD -MP -MF ".deps/helloworld.Tpo" \
-c -o helloworld.o `test -f 'helloworld.c' || echo '$^'` helloworld.c; \
then mv -f ".deps/helloworld.Tpo" ".deps/helloworld.Po"; \
else rm -f ".deps/helloworld.Tpo"; exit 1; \
fi

gcc -g -O2 -o helloworld helloworld.o
```

运行 `helloworld`

```
$ ./helloworld
Hello Linux world!
```

这样 `helloworld` 就编译出来了，你如果按上面的步骤来做的话，应该也会很容易地编译出正确的 `helloworld` 文件。你还可以试着使用一些其他的 `make` 命令，如 `make clean`, `make install`, `make dist`，看看它们会给你什么样的效果。感觉如何？自己也能写出这么专业的 `Makefile`，老板一定会对你刮目相看。

四、深入浅出

针对上面提到的各个命令，我们再做些详细的介绍。

1、 autoscan

autoscan 是用来扫描源代码目录生成 `configure.scan` 文件的。autoscan 可以用目录名做为参数，但如果你不使用参数的话，那么 autoscan 将认为使用的是当前目录。autoscan 将扫描你所指定目录中的源文件，并创建 `configure.scan` 文件。

2、 configure.scan

`configure.scan` 包含了系统配置的基本选项，里面都是一些宏定义。我们需要将它改名为 `configure.in`

3、 aclocal

`aclocal` 是一个 perl 脚本程序。`aclocal` 根据 `configure.in` 文件的内容，自动生成 `aclocal.m4` 文件。`aclocal` 的定义是：“`aclocal - create aclocal.m4 by scanning configure.ac`”。

4、 autoconf

`autoconf` 是用来产生 `configure` 文件的。`configure` 是一个脚本，它能设置源程序来适应各种不同的操作系统平台，并且根据不同的系统来产生合适的 `Makefile`，从而可以使你的源代码能在不同的操作系统平台上被编译出来。

`configure.in` 文件的内容是一些宏，这些宏经过 `autoconf` 处理后会变成检查系统特性、环境变量、软件必须的参数的 shell 脚本。`configure.in` 文件中的宏的顺序并没有规定，但是你必须在所有宏的最前面和最后面分别加上 `AC_INIT` 宏和 `AC_OUTPUT` 宏。

在 `configure.in` 中：

#号表示注释，这个宏后面的内容将被忽略。

`AC_INIT(FILE)` 这个宏用来检查源代码所在的

路径。

```
AM_INIT_AUTOMAKE(PACKAGE, VERSION)
```

这个宏是必须的，它描述了我们将要生成的软件包的名字及其版本号：PACKAGE 是软件包的名字，VERSION 是版本号。当你使用 make dist 命令时，它会给你生成一个类似 helloworld-1.0.tar.gz 的软件发行包，其中就有对应的软件包的名字和版本号。

```
AC_PROG_CC
```

这个宏将检查系统所用的 C 编译器。

```
AC_OUTPUT(FILE)
```

这个宏是我们要输出的 `Makefile` 的名字。

我们在使用 automake 时，实际上还需要用到其他的一些宏，但我们可以用 aclocal 来帮我们自动产生。执行 aclocal 后我们会得到 aclocal.m4 文件。

产生了 `configure.in` 和 `aclocal.m4` 两个宏文件后，我们就可以使用 autoconf 来产生 `configure` 文件了。

5、 `Makefile.am`

`Makefile.am` 是用来生成 `Makefile.in` 的，需要你手工书写。`Makefile.am` 中定义了一些内容：

```
AUTOMAKE_OPTIONS
```

这个是 automake 的选项。在执行 automake 时，它会检查目录下是否存在标准 GNU 软件包中应具备的各种文件，例如 AUTHORS、ChangeLog、NEWS 等文件。我们将其设置成 foreign 时，automake 会改用一般软件包的标准来检查。

`bin_PROGRAMS` 这个是指定我们所要产生的可执行文件的文件名。如果你要产生多个可执行文件，那么在各个名字间用空格隔开。

```
helloworld_SOURCES
```

这个是指定产生 “helloworld” 时所需要的源代码。如果它用到了多个源文件，那么请使用空格符号将它们隔开。比如需要 `helloworld.h`, `helloworld.c` 那么请写成 `helloworld_SOURCES= helloworld.h helloworld.c`。

如果你在 `bin_PROGRAMS` 定义了多个可执行文件，则对应每个可执行文件都要定义相对的 `filename_SOURCES`。

6、 automake

我们使用 `automake --add-missing` 来产生 `Makefile.in`。

选项`--add-missing` 的定义是 “`add missing standard files to package`”，它会让 `automake` 加入一个标准的软件包所必须的一些文件。

我们用 `automake` 产生的 `Makefile.in` 文件是符合 GNU `Makefile` 惯例的，接下来我们只要执行 `configure` 这个 shell 脚本就可以产生合适的 `Makefile` 文件了。

7、 Makefile

在符合 GNU `Makefile` 惯例的 `Makefile` 中，包含了一些基本的预先定义的操作：

`make`

根据 `Makefile` 编译源代码，连接，生成目标文件，可执行文件。

`make clean`

清除上次的 `make` 命令所产生的 `object` 文件（后缀为 “`.o`” 的文件）及可执行文件。

`make install`

将编译成功的可执行文件安装到系统目录中，一般为 `/usr/local/bin` 目录。

`make dist`

产生发布软件包文件（即 `distribution package`）。这个命令将会将可执行文件及相关文件打包成一个 `tar.gz` 压缩的文件用来作为发布软件的软件包。

它会在当前目录下生成一个名字类似 “`PACKAGE-VERSION.tar.gz`” 的文件。PACKAGE 和 VERSION，是我们在 `configure.in` 中定义的 `AM_INIT_AUTOMAKE(PACKAGE, VERSION)`。

```
make distcheck
```

生成发布软件包并对其进行测试检查，以确定发布包的正确性。这个操作将自动把压缩包文件解开，然后执行 `configure` 命令，并且执行 `make`，来确认编译不出现错误，最后提示你软件包已经准备好，可以发布了。

```
=====
```

```
helloworld-1.0.tar.gz is ready for distribution
```

```
=====
```

```
make distclean
```

类似 `make clean`，但同时也将 `configure` 生成的文件全部删除掉，包括 `Makefile`。

五、结束语

通过上面的介绍，你应该可以很容易地生成一个你自己的符合 GNU 惯例的 `Makefile` 文件及对应的项目文件。

如果你想写出更复杂的且符合惯例的 `Makefile`，你可以参考一些开放代码的项目中的 `configure.in` 和 `Makefile.am` 文件，比如：嵌入式数据库 `sqlite`，单元测试 `cppunit`。

(<http://www.fanqiang.com>)

Complex One:autoconf 和 automake 生成 Makefile 文件

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



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, E-mail: normalnotebook@126.com)。

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。

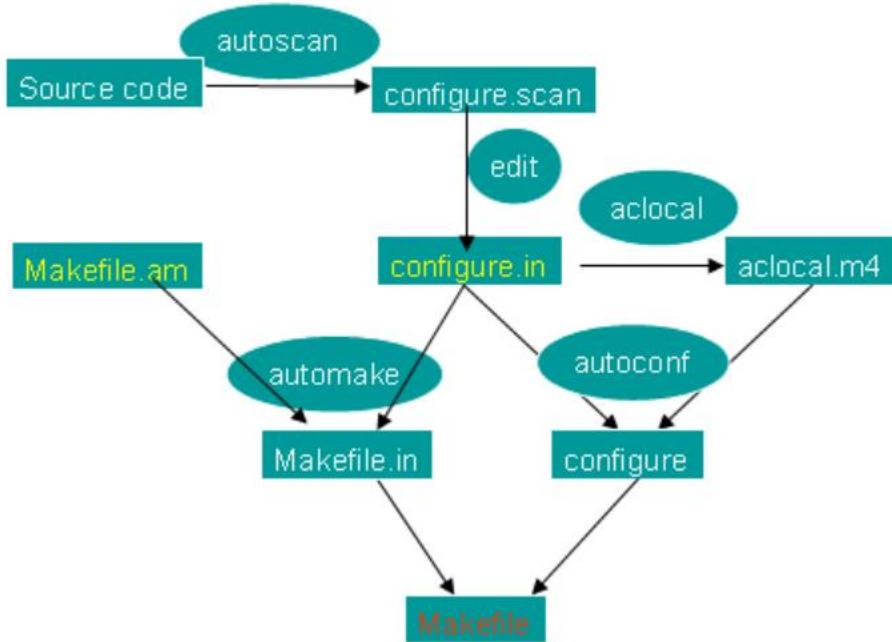
生成 `Makefile` 的来龙去脉

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 `autoscans` 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 `./configugre` 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 `autoscans` 工具生成 `configure.scan` 文件时，我们需要将 `configure.scan` 重命名为 `configure.in` 文件。
`configure.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 `configure.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 `--lthread`:
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 `config.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：

```
$mv configure.scan configure.in  
$vim configure.in
```

修改后的结果如下：

```
#                                     -*- 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 `pthread':  
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  
         ])D
```

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:

宏	含义
<code>AC_CHECK_LIB (lib,function[,action_if_found ,[action_if_not_found, ,[other_libs]]])</code>	该宏用来检查 lib 库中是否存在指定的函数。当测试成功时, 执行 shell 命令 <code>action_if_found</code> 或者当 <code>action_if_found</code> 为空时, 在输出变量 <code>LIBS</code> 中添加 <code>-l<code>lib</code></code> 。 <code>action_if_not_found</code> 把 <code>-l<code>other_libs</code></code> 选项传给 <code>link</code> 命令

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 1 Makefile.am 一般格式

文件类型	书写格式
可执行文件	bin_PROGRAMS = foo foo_SOURCES = xxxx.c foo_LDADD = foo_LDFLAGS = foo_DEPENDENCIES =
静态库	lib_LIBRARIES = libfoo.a foo_a_SOURCES = foo_a_LDADD = foo_a_LIBADD = foo_a_LDFLAGS =
头文件	include_HEADERS = foo.h
数据文件	data_DATA = data1 data2

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

变量	含义
INCLUDES	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
EXTRA_DIST	源程序和一些默认的文件将自动打入.tar.gz 包,其它文件若要进入.tar.gz 包可以用这种办法,比如配置文件,数据文件等等。
SUBDIRS	在处理本目录之前要递归处理哪些子目录

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

路径变量	含义
<code>\$(top_srcdir)</code>	工程最顶层目录，用于引用源程序
<code>\$(top_builddir)</code>	定义了生成目标文件上最上层目录，用于引用.o 等编译出来的目标文件。

尽量用相对路径引用源程序的位置，以下两个变量是预定义好的：

`$(top_srcdir)`无论在哪个目录层次，该变量定义了包含 `src` 目录的目录位置，用于引用源程序；

`$(top_builddir)` 定义了生成目标文件上最上层目录，用于引用.o 等编译出来的目标文件

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=<new_path>`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：

```
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：

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

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，我们写出的 `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/>

实际问题

特殊编译器的检查

首先在 `configure.in` 中加上对特殊编译器的检查，如果检查不到，则 `configure` 时会停止并给出“`Couldn't find mpicc`”的出错信息：

```
#检查 mpicc 编译器
AC_CHECK_PROG(MPICC, mpicc, yes, no)
if test "$MPICC" = no; then
```

```
AC_MSG_ERROR([Couldn't find mpicc])
fi
```

然后把需要用 mpicc 编译的源程序放在一个目录下面。在这个目录中先用上面的方法写 Makefile.am 文件。然后再加上下面这部分：

```
CC=mpicc
```

```
CFLAGS=
```

这样用自己定义的编译器和编译标志取代系统定义的编译器和编译标志。

由于我没有 mpicc 编译器，没有办法亲自做试验。可能使用 mpicc 还有其它的要求，直接照做不一定行得通。但解决问题的思想应该就是这样了，希望你能举一反三。

理解 AC_INIT

AC_INIT(dir1/code2.c)写一个要检查的源文件就够了。多写是没有用的。不信你可以作一下试验。只要把 AC_INIT 改为 AC_INIT(dir1/code2.c,dir1/vvv.c)，这个 vvv.c 文件是不存在的，但执行 configure 时，一样不会报错。

但这个要检查的文件应该是[程序的主文件](#)，少了它，程序将无法运行。

AM_INIT_AUTOMAKE(package, version)是项目文件打包时的名字。比如说，

AM_INIT_AUTOMAKE(mypro, 0.0.1)，如果项目要发布源代码，这时打包就可以执行一个 make dist，会自动生成一个 mypro-0.0.1.tar.gz 的文件。只要你从网上下载并编译过软件的源代码，对于这个 应该不会陌生。当然从技术的角度来说，项目名称和项目版本号可以随便取，但为了管理上的方便，一般都有规范可循。

如何生成动态库和静态库？

静态库是一个目标文件的简单集合。由 ar(archive，归档的意思)生成。

```
ar -cr libfoo.a foo.o bar.o
```

通常命名方式是 libxxx.a，但是你不遵守也没有太大的问题。应用程序在使用你的库的时候，通常只需要告诉 ld 你的库

名字即可，这个名字就是 libxxx.a 中的 xxx，例如 ld -lfoo。意思是告诉 ld，连接一个名字为 libfoo.a 或者 libfoo.so 的库。如果你的库名字不遵循 libxxx.a 的格式，ld 就找不到，给应用开发造成麻烦。

另外，静态的意思是每个用到该库的应用程序都拥有一份自己的库拷贝，应用程序运行的时候，即使将库删除也没有问题
因为应用程序自己已经有了自己的拷贝。

动态库结构复杂一些，通常是一个 ELF 格式的文件。可以有三种方法生成：

```
ld -G  
gcc -share  
libtool
```

用 ld 最复杂，用 gcc -share 就简单的多，但是-share 并非在任何平台都可以使用。GNU 提供了一个更好的工具 libtool，专门用来在各种平台上生成各种库。

动态库实际上应该叫做共享库，只是很多人从 windows 的 Dynamic Linked Library 这个词学习过来，把 unix 的共享库称做动态库。所有应用程序共享一份库拷贝，所以，即使连接完了，也不能将其删除。而且需要在 **LD_LIBRARY_PATH** 这个环境变量中正确的设置库所在的位置，否则程序运行会报告找不到这个库。

Linux 共享对象技术

在 Linux 操作系统中，采用了很多共享对象技术(Shared Object)，虽然它和 Windows 里的动态库相对应，但它并不称为动态库。相应的共享对象文件以 so 作为后缀，为了方便，在本文中，对该概念不进行专门区分。Linux 系统的 /Lib 以及标准图形界面的 /usr/X11R6/lib 等目录里面，就有许多以 so 结尾的共享对象。同样，在 Linux 下，也有静态函数库这种调用方式，相应的后缀以 a 结束。Linux 采用该共享对象技术以方便程序间共享，节省程序占有空间，增加程序的可扩展性和灵活性。Linux 还可以通过 LD-PRELOAD 变量让开发人员可以使用自己的程序库中的模块来替换系统模块。

同 Windows 系统一样，在 Linux 中创建和使用动态库是比较容易的事情，在编译函数库源程序时加上`-shared` 选项即可，这样所生成的执行程序就是动态链接库。通常这样的程序以 `so` 为后缀，在 Linux 动态库程序设计过程中，通常流程是编写用户的接口文件，通常是 `h` 文件，编写实际的函数文件，以 `c` 或 `cpp` 为后缀，再编写 `makefile` 文件。对于较小的动态库程序可以不用如此，但这样设计使程序更加合理。

编译生成动态连接库后，进而可以在程序中进行调用。在 Linux 中，可以采用多种调用方式，同 Windows 的系统目录(`C:\system32` 等)一样，可以将动态库文件拷贝到`/lib` 目录或者在`/lib` 目录里面建立符号连接，以便所有用户使用。下面介绍 Linux 调用动态库经常使用的函数，但在使用动态库时，源程序必须包含 `dlopen.h` 头文件，该文件定义调用动态链接库的函数的原型。

(1) 打开动态链接库: `dlopen`, 函数原型 `void *dlopen (const char *filename, int flag);`

`dlopen` 用于打开指定名字(`filename`)的动态链接库，并返回操作句柄。 (2) 取函数执行地址:

`dlSym`, 函数原型为: `void *dlSym(void *handle, char *symbol);`

`dlSym` 根据动态链接库操作句柄(`handle`)与符号(`symbol`)，返回符号对应的函数的执行代码地址。

(3) 关闭动态链接库: `dlClose`, 函数原型为: `int dlclose (void *handle);`

`dlClose` 用于关闭指定句柄的动态链接库，只有当此动态链接库的使用计数为 0 时，才会真正被系统卸载。

(4) 动态库错误函数: `dlError`, 函数原型为: `const char *dlError (void);` 当动态链接库操作函数执行失败时，`dlError` 可以返回出错信息，返回值为 `NULL` 时表示操作函数执行成功。

在取到函数执行地址后，就可以在动态库的使用程序里面根据动态库提供的函数接口声明调用动态库里面的函数。在编写调用动态库的程序的 `makefile` 文件时，需要加入编译选项`-rdynamic` 和`-ldl`。

除了采用这种方式编写和调用动态库之外，Linux 操作系统也提供了一种更为方便的动态库调用方式，也方便了其它程序调用，这种方式与 Windows 系统的隐式链接类似。其动态库命名方式为 “`lib* so *`”。在这个命名方式中，第一个*表示动态链接库的库名，第二个*通常表示该动态库的版本号，也可以没有版本号。在这种调用方式中，需要维护动态链接库的配置文件`/etc/ld.so.conf` 来让动态链接库为系统所使用，通常将动态链接库所在目录名追加到动态链接库 配置文件中。如具有 X window 窗口系统发行版该文件中都具有`/usr/X11R6/lib`，它指向 X window 窗口系统的动态链接库所在目录。为了使动态链接库能为系统所共享，还需运行动态链接库的管理命令 `/sbin/ldconfig`。在编译所引用的动

态库时，可以在 gcc 采用 `-fPIC` 或 `-L` 选项或直接引用所需的动态链接库方式进行编译。在 Linux 里面，可以采用 `ldd` 命令来检查程序依赖共享库。

完整的 Mingw 开发环境(使用 Autoconf,automake,libtool,libiconv and gettext)

转载自 LinuxAid 留言：

完整的 Mingw 开发环境(使用 Autoconf,automake,libtool,libiconv and gettext)

大家使用 gcc 在 windows 下开发软件，有两个途径，一、Cygwin,二、Mingw32

Cygwin 可以使用 autoconf,automake,libtool,libiconv and gettext(I18N/L10N) 进行完整的开发，但由于 Cygwin 封装了一层底层库，所以运行速度较慢，因此要想用 GCC 开发 Native Windows 程序，则

只有选择 Mingw 了，而 Mingw 却没有提供完整的 Autoconf,automake,libtool,libiconv and gettext 开发环境，并且很多包很陈旧，还有很多 bug。对一个新手来说，用 Mingw 开发是一件非常痛苦的事，我也是从这种痛苦中过来的，为了不让其它人也这样痛苦！我就整合了一个完整的 Mingw 开发环境 (使用最新的 Autoconf,automake,libtool,libiconv and gettext)，并且经过测试过的！

欢迎大家下载使用

下载地址：

http://sourceforge.net/project/showfiles.php?group_id=148008
(<http://sourceforge.net/projects/mingw-install>)

autoconf,automake,m4,libtool 的安装

一. 安装 autoconf

```
$ wget http://ftp.gnu.org/gnu/autoconf/autoconf-2.59.tar.gz  
$ tar -zvxf autoconf-2.59.tar.gz  
$ cd ./autoconf-2.59  
$ ./configure
```

```
$ sudo make  
$ sudo make install  
$ cd
```

二 安装 automake

```
$ wget http://ftp.gnu.org/gnu/automake/automake-1.9.6.tar.gz  
$ tar -zvxf automake-1.9.6.tar.gz  
$ cd ./automake-1.9.6  
$ ./configure  
$ sudo make  
$ sudo make install  
$ cd
```

三 安装 GNU m4

```
$ wget http://ftp.gnu.org/gnu/m4/m4-1.4.4.tar.gz  
$ tar -zvxf m4-1.4.4.tar.gz  
$ cd m4-1.4.4  
$ ./configure  
$ sudo make  
$ sudo make install  
$ cd
```

四 安装 GNU libtool

```
$ wget http://libtool.opendarwin.org/libtool.tar.gz (这个是 cvs 版本的)  
$ tar -zvxf libtool.tar.gz  
$ cd libtool-2.1a  
$ ./configure  
$ sudo make  
$ sudo make install
```

My Readme

Principle chart

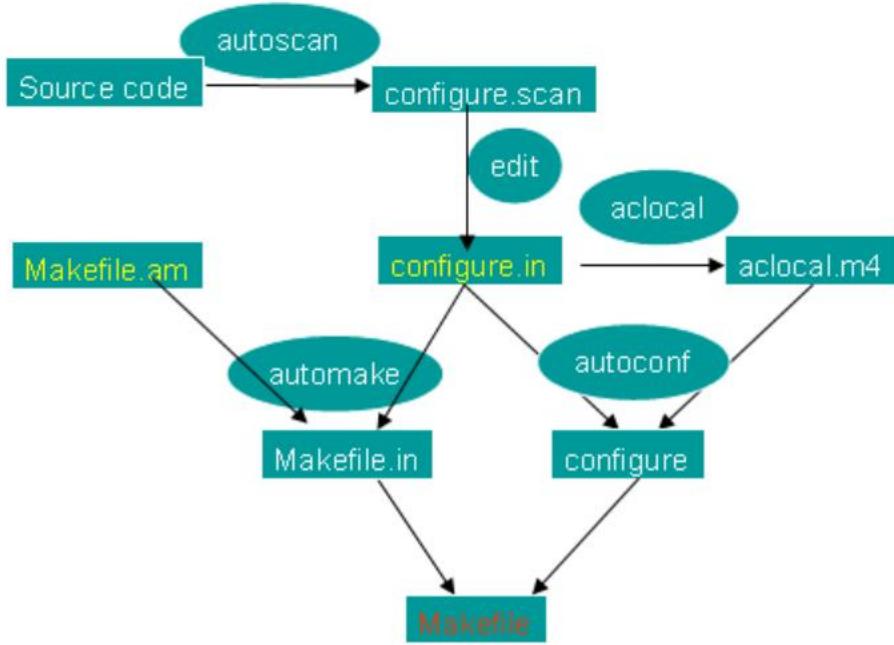


图 2 生成 Makefile 流程图

Script command:

(1) `autoscans`

(2)

```

#!/bin/sh
aclocal
autoheader
Automake --add-missing
Autoconf

```

Then we could run `./configure` to create final Makefile.

Run `make` could get the final executable file.

4.7.1 Preset Output Variables

Some output variables are preset by the Autoconf macros. Some of the Autoconf macros set additional output variables, which are mentioned in the descriptions for those macros. See [Section B.2 \[Output Variable Index\], page 257](#), for a complete list of output variables. See [Section 4.7.2 \[Installation Directory Variables\], page 23](#), for the list of the preset ones related to installation directories. Below are listed the other preset ones. They all are precious variables (see [Section 7.2 \[Setting Output Variables\], page 100](#), AC_ARG_VAR).

CFLAGS [Variable]

Debugging and optimization options for the C compiler. If it is not set in the environment when configure runs, the default value is set when you call AC_PROG_CC (or empty if you don't). configure uses this variable when compiling or linking programs to test for C features.

If a compiler option affects only the behavior of the preprocessor (e.g., '-D name'), it should be put into CPPFLAGS instead. If it affects only the linker (e.g., '-L directory'), it should be put into LDFLAGS instead. If it affects only the compiler proper, CFLAGS is the natural home for it. If an option affects multiple phases of the compiler, though, matters get tricky. One approach to put such options directly into CC, e.g., CC='gcc -m64'. Another is to put them into both CPPFLAGS and LDFLAGS, but not into CFLAGS.

configure_input [Variable]

A comment saying that the file was generated automatically by configure and giving the name of the input file. AC_OUTPUT adds a comment line containing this variable to the top of every makefile it creates. For other files, you should reference this variable in a comment at the top of each input file. For example, an input shell script should begin like this:

```
#!/bin/sh
# @configure_input@
```

The presence of that line also reminds people editing the file that it needs to be processed by configure in order to be used.

CPPFLAGS [Variable]

Preprocessor options for the C, C++, and Objective C preprocessors and compilers.

If it is not set in the environment when configure runs, the default value is empty.

configure uses this variable when preprocessing or compiling programs to test for C, C++, and Objective C features.

This variable's contents should contain options like '-I', '-D', and '-U' that affect only the behavior of the preprocessor. Please see the explanation of CFLAGS for what you can do if an option affects other phases of the compiler as well.

Currently, configure always links as part of a single invocation of the compiler that also preprocesses and compiles, so it uses this variable also when linking programs. However, it is unwise to depend on this behavior because the GNU coding standards do not require it and many packages do not use CPPFLAGS when linking programs.

See [Section 7.3 \[Special Chars in Variables\], page 102](#), for limitations that CPPFLAGS

might run into.

22 Autoconf

CXXFLAGS [Variable]

Debugging and optimization options for the C++ compiler. It acts like CFLAGS, but for C++ instead of C.

DEFS [Variable]

‘-D’ options to pass to the C compiler. If AC_CONFIG_HEADERS is called, configure replaces ‘@DEFS@’ with ‘-DHAVE_CONFIG_H’ instead (see [Section 4.8 \[Configuration Headers\], page 29](#)). This variable is not defined while configure is performing its tests, only when creating the output files. See [Section 7.2 \[Setting Output Variables\], page 100](#), for how to check the results of previous tests.

ECHO_C [Variable]

ECHO_N [Variable]

ECHO_T [Variable]

How does one suppress the trailing newline from echo for question-answer message pairs? These variables provide a way:

```
echo $ECHO_N "And the winner is... $ECHO_C"  
sleep 100000000000  
echo "${ECHO_T}dead."
```

Some old and uncommon echo implementations offer no means to achieve this, in which case ECHO_T is set to tab. You might not want to use it.

ERLCFLAGS [Variable]

Debugging and optimization options for the Erlang compiler. If it is not set in the environment when configure runs, the default value is empty. configure uses this variable when compiling programs to test for Erlang features.

FCFLAGS [Variable]

Debugging and optimization options for the Fortran compiler. If it is not set in the environment when configure runs, the default value is set when you call AC_PROG_FC (or empty if you don’t). configure uses this variable when compiling or linking programs to test for Fortran features.

FFLAGS [Variable]

Debugging and optimization options for the Fortran 77 compiler. If it is not set in the environment when configure runs, the default value is set when you call AC_PROG_F77 (or empty if you don’t). configure uses this variable when compiling or linking programs to test for Fortran 77 features.

LDFLAGS [Variable]

Options for the linker. If it is not set in the environment when configure runs, the default value is empty. configure uses this variable when linking programs to test for C, C++, Objective C, and Fortran features.

This variable’s contents should contain options like ‘-s’ and ‘-L’ that affect only the behavior of the linker. Please see the explanation of CFLAGS for what you can do if an option also affects other phases of the compiler.

Don’t use this variable to pass library names (‘-l’) to the linker; use LIBS instead.

LIBS [Variable]

‘-l’ options to pass to the linker. The default value is empty, but some Autoconf macros may prepend extra libraries to this variable if those libraries are found and provide necessary functions, see [Section 5.4 \[Libraries\], page 44](#). configure uses this variable when linking programs to test for C, C++, and Fortran features.

OBJCFLAGS [Variable]

Debugging and optimization options for the Objective C compiler. It acts like CFLAGS, but for Objective C instead of C.

builddir [Variable]

Rigorously equal to ‘.’. Added for symmetry only.

abs_builddir [Variable]

Absolute name of builddir.

top_builddir [Variable]

The relative name of the top level of the current build tree. In the top-level directory, this is the same as builddir.

abs_top_builddir [Variable]

Absolute name of top_builddir.

srcdir [Variable]

The name of the directory that contains the source code for that makefile.

abs_srcdir [Variable]

Absolute name of srcdir.

top_srcdir [Variable]

The name of the top-level source code directory for the package. In the top-level directory, this is the same as srcdir.

abs_top_srcdir [Variable]

Absolute name of top_srcdir.

4.7.2 Installation Directory Variables

The following variables specify the directories for package installation, see [section “Variables for Installation Directories” in The GNU Coding Standards](#), for more information. See the end of this section for details on when and how to use these variables.

bindir [Variable]

The directory for installing executables that users run.

datadir [Variable]

The directory for installing idiosyncratic read-only architecture-independent data.

datarootdir [Variable]

The root of the directory tree for read-only architecture-independent data files.

docdir [Variable]

The directory for installing documentation files (other than Info and man).

24 Autoconf

dvidir [Variable]

The directory for installing documentation files in DVI format.

exec_prefix [Variable]

The installation prefix for architecture-dependent files. By default it’s the same as prefix. You should avoid installing anything directly to exec prefix. However, the

default value for directories containing architecture-dependent files should be relative to exec prefix.

htmldir [Variable]

The directory for installing HTML documentation.

includedir [Variable]

The directory for installing C header files.

infodir [Variable]

The directory for installing documentation in Info format.

libdir [Variable]

The directory for installing object code libraries.

libexecdir [Variable]

The directory for installing executables that other programs run.

localedir [Variable]

The directory for installing locale-dependent but architecture-independent data, such as message catalogs. This directory usually has a subdirectory per locale.

localstatedir [Variable]

The directory for installing modifiable single-machine data.

mandir [Variable]

The top-level directory for installing documentation in man format.

oldincludedir [Variable]

The directory for installing C header files for non-GCC compilers.

pdfdir [Variable]

The directory for installing PDF documentation.

prefix [Variable]

The common installation prefix for all files. If exec prefix is defined to a different value, prefix is used only for architecture-independent files.

psdir [Variable]

The directory for installing PostScript documentation.

sbindir [Variable]

The directory for installing executables that system administrators run.

sharedstatedir [Variable]

The directory for installing modifiable architecture-independent data.

Chapter 4: Initialization and Output Files 25

sysconfdir [Variable]

The directory for installing read-only single-machine data.

Reference

<http://www.chinaunix.net> soft.soft/soft.soft
《automake autoconf & libtool》详细信息