**例解 autoconf 和 automake 生成 Makefile 文件**



杨 小华  
2006 年 9 月 21 日发布

[Weibo](http://service.weibo.com/share/share.php?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2Fl-makefile%2Findex.html%25&title=%E4%BE%8B%E8%A7%A3%20autoconf%20%E5%92%8C%20automake%20%E7%94%9F%E6%88%90%20Makefile%20%E6%96%87%E4%BB%B6&language=zh_cn)[Google+](https://plus.google.com/share?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2Fl-makefile%2Findex.html&t=%E4%BE%8B%E8%A7%A3%20autoconf%20%E5%92%8C%20automake%20%E7%94%9F%E6%88%90%20Makefile%20%E6%96%87%E4%BB%B6)[用电子邮件发送本页面](mailto:?subject=%E4%BE%8B%E8%A7%A3%20autoconf%20%E5%92%8C%20automake%20%E7%94%9F%E6%88%90%20Makefile%20%E6%96%87%E4%BB%B6&body=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Flinux%2Fl-makefile%2Findex.html)

[Comments](https://www.ibm.com/developerworks/cn/linux/l-makefile/#icomments)

[3](https://www.ibm.com/developerworks/cn/linux/l-makefile/" \l "icomments)

**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。

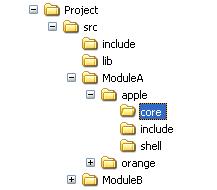
但是如果通过查阅make的帮助文档来手工编写Makefile,对任何程序员都是一场挑战。幸而有GNU 提供的Autoconf及Automake这两套工具使得编写makefile不再是一个难题。

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](mailto: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

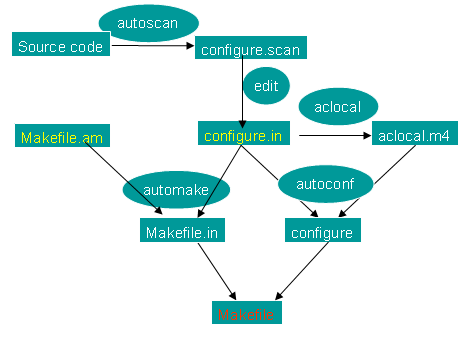
7) 运行autoconf命令

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：

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7  8  9  10  11  12  13  14 | # 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：**

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7  8  9  10 | 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：**

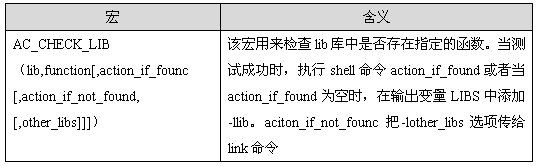
|  |  |
| --- | --- |
| **1**  2 | $mv configure.scan configure.in  $vim configure.in |

**修改后的结果如下：**

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #                                -\*- 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。**

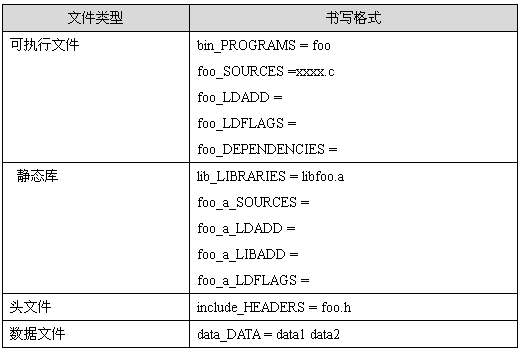
**Autoconf提供了很多内置宏来做相关的检测，限于篇幅关系，我们在这里对其他宏不做详细的解释，具体请参看参考文献1和参考文献2，也可参看autoconf信息页。**

**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。

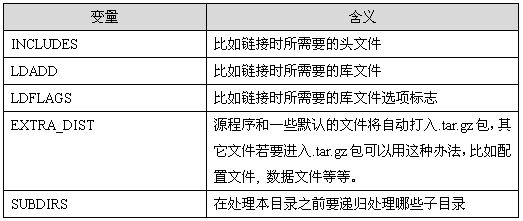
**表 1Makefile.am一般格式**



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。

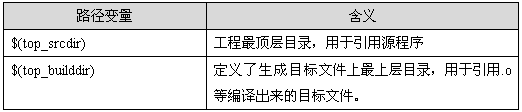
Makefile.am还提供了一些全局变量供所有的目标体使用：

**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=<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：

|  |  |
| --- | --- |
| **1**  2  3  4 | 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：**

|  |  |
| --- | --- |
| **1**  2  3 | 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：**

|  |  |
| --- | --- |
| **1**  2  3  4  5 | 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：**

|  |  |
| --- | --- |
| **1**  2  3  4  5  6 | 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：**

|  |  |
| --- | --- |
| **1**  2  3 | 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：**

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7 | …….  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。**