How to Use Make and Automake

Cui Tao
tcui@lsec.cc.ac.cn

The State Key Laboratory of Scientific and Engineering Computing
The Institute of Computational Mathematics and Scientific Engineering Computing

April 21, 2005

- Part I: Compile ,Link and Debug
- Part II: Make and Makefile
- Part III: Autoconf Automake

Outline of Part I

1. Compile and Link
2. 库依赖
3. 调试

Outline of Part I

1. Compile and Link
2. 库依赖
3. 调试

Outline of Part I

1. Compile and Link
2. 库依赖
3. 调试

Outline of Part II

4 Makefile简介
- 什么是Makefile?
- 作者简介
- Makefile文件名
- Makefile的构成
- Make是如何工作的

5 如何写Makefile
- A Simple Example
- Makefile的规则
- Makefile的变量
- Makefile的命令
- Makefile的伪目标

Outline of Part II

4. Makefile简介
   - 什么是Makefile?
   - 作者简介
   - Makefile文件名
   - Makefile的构成
   - Make是如何工作的
5. 如何写Makefile
   - A Simple Example
   - Makefile的规则
   - Makefile的变量
   - Makefile的命令
   - Makefile的伪目标

Outline of Part III

6 Autoconf 和Automake 简介
- 简介
- A Simple Example
- 使用autoconf生成configure
- 使用automake生成Makefile.in

Part I

Compile ,Link and Debug

Outline

1 Compile and Link

2 库依赖

3 调试

Outline

1 Compile and Link

2 库依赖

3 调试

Outline

1. Compile and Link
2. 库依赖
3. 调试

编译链接过程

- 预处理（Pre-Processing）
  - 展开宏
  - 插入include语句所包含的内容
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

编译链接过程

- 预处理（Pre-Processing）
  - 展开宏
  - 插入include语句所包含的内容
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

编译链接过程

- 预处理（Pre-Processing）
  - 展开宏
  - 插入include语句所包含的内容
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

编译链接过程

- 预处理（Pre-Processing）
  - 展开宏
  - 插入include语句所包含的内容
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

编译链接过程

- gcc -E hello.c -o hello.i
- gcc -c hello.i -o hello.o
- gcc hello.o -o hello

第三方函数库

- 头文件（.h）和库文件（.so或者.a）的集合
- 如何查找所需要的头文件和库文件？？

如何使用GCC

- -I: gcc foo.c -I /home/tcui/include -o foo
- -L:
- -l: gcc foo.c -L /home/xiaowp/lib -lfoo -o foo

如何使用GCC

- -I: gcc foo.c -I /home/tcui/include -o foo
- -L:
- -l: gcc foo.c -L /home/xiaowp/lib -lfoo -o foo

动态库和静态库

- 动态链接库（通常以.so结尾）— first choice if without '-static'
- 静态链接库（通常以.a结尾）

GDB

GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具。

一般来说，GDB主要帮忙你完成下面四个方面的功能：

- 启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。
- 可让被调试的程序在你所指定的调置的断点处停住。
- 当程序被停住时，可以检查此时你的程序中所发生的事。
- 动态的改变你程序的执行环境。

Example

Example: crash.c

```
#include <stdio.h>
int main(void)
int input =0;
printf(‘‘Input an integer:’’);
scanf(‘‘%d’’, input);
printf(‘‘The integer you input is %d ’’, input);
return 0;
```

Part II

Something about Makefile

Outline

4 Makefile简介
- 什么是Makefile?
- 作者简介
- Makefile文件名
- Makefile的构成
- Make是如何工作的

5 如何写Makefile
- A Simple Example
- Makefile的规则
- Makefile的变量
- Makefile的命令
- Makefile的伪目标

Outline

4 Makefile简介
- 什么是Makefile?
- 作者简介
- Makefile文件名
- Makefile的构成
- Make是如何工作的

5 如何写Makefile
- A Simple Example
- Makefile的规则
- Makefile的变量
- Makefile的命令
- Makefile的伪目标

什么是Makefile?

A makefile details the files, dependencies, and rules by which an executable application is built.

Richard Stallman

开源软件的领袖和先驱，从来没有领过一天工资，从来没有使用过Windows操作系统。这是他的主页：http://www.stallman.org/

这里贴上一张他的近照：



Roland McGrath

个人主页是：http://www.frob.com/ roland/ ，下面是他的一些事迹：

- 合作编写了并维护GNU make。
- 和Thomas Bushnell一同编写了GNU Hurd。
- 编写并维护着GNU C library。
- 合作编写并维护着部分的GNU Emacs。

Makefile文件名

- 默认：GNUmakefile, makefile, and Makefile
- 特定：make -f filename
- man make

Makefile的构成

- 显示规则: 显示规则说明了，如何生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
- 隐晦规则: 由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。
- 变量的定义: 在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点像C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
- 文件指示: 其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。
- 注释: Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，如果你要在你的Makefile中使用

引用其它的Makefile

在Makefile使用include关键字可以把别的Makefile包含进来。include的语法是：include ¡filename¿

Make的工作原理 I

- make会在当前目录下找名字叫'makefile'或'Makefile'的文件。
- 如果找到，它会找文件中的相应目标。如果make没有指定，默认找第一个目标。
- 如果目标不存在，或是目标所依赖的文件的文件修改时间要比目标文件新，那么，他就会执行后面所定义的命令。

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

make的执行步骤

- 读入所有的Makefile
- 读入被include的其a它Makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建依赖关系链
- 根据依赖关系，决定哪些目标要重新生成
- 执行生成命令

Example I

```
cc = gcc
INCLUDE = -I/usr/local/mysql/include/mysql
LIBS = -L/usr/local/mysql/lib/mysql
CLIENT = -lmysqlclient

.PHONY all
all: mysqlcli

main.o: main.c passwd.h
$(cc) $(INCLUDE) -c main.c
pro1.o: pro1.c
        $(cc) $(INCLUDE) -c pro1.c
```

Example II

```
mysqlcli: main.o pro1.o
$(cc) -o mysqlcli main.o pro1.o $(LIBS) $(CLIENT)
clean:
rm -f main.o pro1.o
        rm -f mysqlcli
```

规则中的语法

```
target : prerequisites
command
```

- target也就是一个目标文件，可以是Object File，也可以是执行文件。还可以是一个标签（Label）。
- prerequisites就是，要生成那个target所需要的文件或是目标。
- command也就是make需要执行的命令。（任意的Shell命令）
- 最后，还值得一提的是，在Makefile中的命令，必须要以

  Tab

  键开始。

Example-显示规则

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        main.o : main.c defs.h
cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
```

Example(续)

```
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
       rm -f edit main.o kbd.o command.o display.o \
         insert.o search.o files.o utils.o
```

什么是隐含规则

“隐含规则”也就是一种惯例，make会按照这种“惯例”心照不喧地来运行。例如，把.c文件编译成.o文件这一规则，你根本就不用写出来，make会自动推导出这种规则，并生成我们需要的.o文件。

常用隐含规则

- 编译C程序的隐含规则:
  $(CC) -c $(CPPFLAGS) $(CFLAGS)
- 编译C++程序的隐含规则:
  $(CXX) -c $(CPPFLAGS) $(CFLAGS)”。
  （建议使用“.cc”作为C++源文件的后缀，而不是“.C”）
- 编译Pascal程序的隐含规: $(PC) -c $(PFLAGS)
- 编译Fortran/Ratfor程序的隐含规则:
  “.f” “$(FC) -c $(FFLAGS)
  “.F” “$(FC) -c $(FFLAGS) $(CPPFLAGS)”
  “.f” “$(FC) -c $(FFLAGS) $(RFLAGS)”
- 预处理Fortran/Ratfor程序的隐含规则。
- .. .. ..

隐含规则中用到的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的makefile中改变这些变量的值，或是在make的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。

隐含规则中用到的变量 I

- AR : ar
- CC : cc
- CXX : g++
- CPP : $(CC) -E
- FC : f77
- PC : pc
- TEX : tex
- RM : rm -f
- CFLAGS : C语言编译器参数
- CXXFLAGS : C++语言编译器参数。

隐含规则中用到的变量 II

- CPPFLAGS：C预处理器参数。（C 和Fortran 编译器也会用到）。
- FFLAGS：Fortran语言编译器参数。
- LDFLAGS：链接器参数。（如：“ld”）
- .. .. ..

模式规则

使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有“%”字符。“%”的意思是表示一个或多个任意字符。

模式规则介绍

模式规则中，至少在规则的目标定义中要包含“%”，否则，就是一般的规则。目标中的“%”定义表示对文件名的匹配，“%”表示长度任意的非空字符串。例如：
“%.c”表示以“.c”结尾的文件名（文件名的长度至少为3），而“s.%.c”则表示以“s.”开头，“.c”结尾的文件名（文件名的长度至少为5）。

模式规则示例

```
%.o : %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@


.o.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
}
```

规则中的通配符 I

make支持三种通配符：“*”，“?”和“[.. ..]”。这是和Unix的B-Shell是相同的。

```
clean:
rm -f *.o

print: *.c
lpr -p $?
touch print
```

Makefile的变量

在Makefile中的定义的变量，就像是C/C++语言中的宏一样，他代表了一个文本字串，在Makefile中执行的时候其会自动原模原样地展开在所使用的地方。

变量名

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。变量是大小写敏感的。

环境变量

make运行时的系统环境变量可以在make开始运行时被载入到Makefile文件中，但是如果Makefile中已定义了这个变量，或是这个变量由make命令行带入，那么系统的环境变量的值将被覆盖。

如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的Makefile中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果Makefile中定义了CFLAGS，那么则会使用Makefile中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。

自动化变量I

$@
表示规则中的目标文件集。在模式规则中，如果有多个目标，那么“$@”就是匹配于目标中模式定义的集合。

自动化变量II

$%
仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是“foo.a(bar.o)”，那么，“$%”就是“bar.o”，“$@”就是“foo.a”。如果目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

自动化变量III

$ <
依赖目标中的第一个目标名字。如果依赖目标是以模式（即“%”）定义的，那么“$ <”将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

自动化变量IV

$?
所有比目标新的依赖目标的集合。以空格分隔。

变量高级运算 I

- 变量值的替换: $var:a=b
  把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”
- 把变量的值再当成变量

  x = y

  y = z

  a := $($(x))

简介

- 每条规则中的命令和操作系统Shell的命令行是一致的。
- make会按顺序逐条的执行命令，每条命令的开头必须以[Tab]键开头。
- make的命令默认是被“/bin/sh”解释执行的。

简介

- 每条规则中的命令和操作系统Shell的命令行是一致的。
- make会按顺序逐条的执行命令，每条命令的开头必须以[Tab]键开头。
- make的命令默认是被“/bin/sh”解释执行的。

简介

- 每条规则中的命令和操作系统Shell的命令行是一致的。
- make会按顺序逐条的执行命令，每条命令的开头必须以[Tab]键开头。
- make的命令默认是被“/bin/sh”解释执行的。

命令的执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。

Example I

示例一：

```
exec:
cd /home/hchen
pwd
```

示例二：

```
exec:
cd /home/hchen; pwd
```

命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

如何忽略命令出错???

——在命令前加上“-”

命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

如何忽略命令出错???

——在命令前加上“-”

命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

如何忽略命令出错???

——在命令前加上“- ”

什么是伪目标?

。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。
例如：

```
clean:
rm *.o temp
```

伪目标声明

为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”

```
.PHONY: clean
clean:
rm *.o temp
```

伪目标的依赖文件 I

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。例如：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
cc -o prog1 prog1.o utils.o
prog2 : prog2.o
cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
cc -o prog3 prog3.o sort.o utils.o
```

伪目标的依赖文件 II

Makefile中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。

Part III

Autoconf Automake

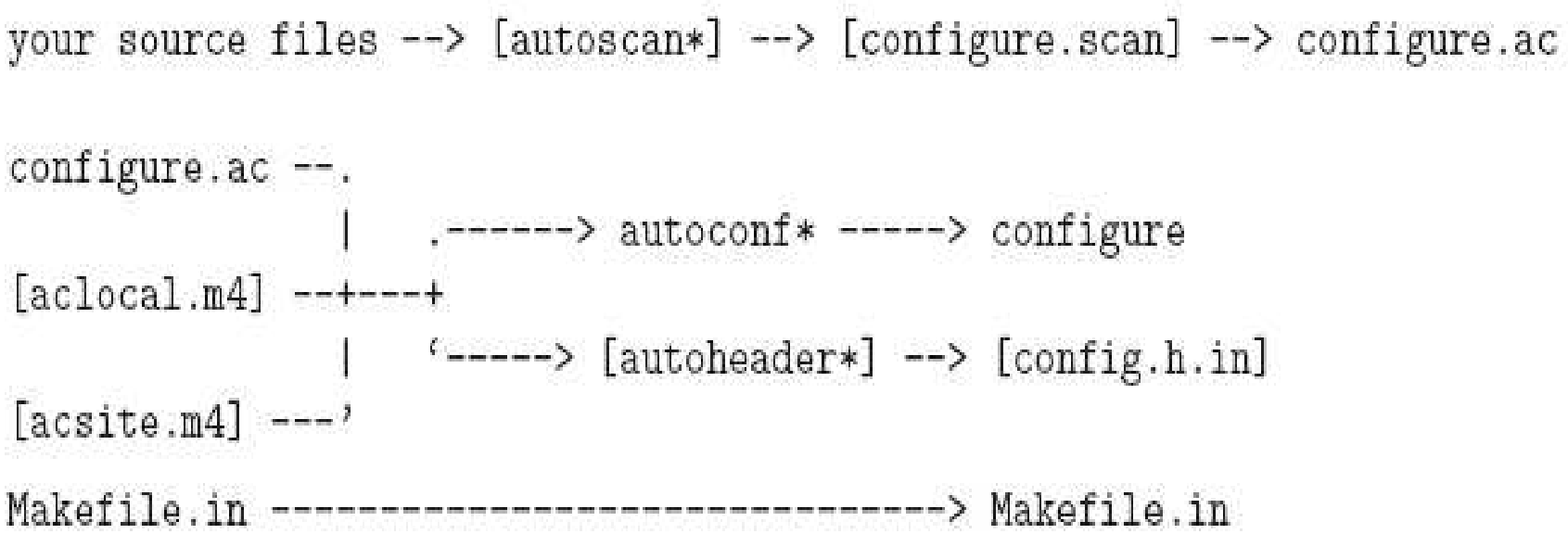
Outline

6 Autoconf 和Automake 简介
- 简介
- A Simple Example
- 使用autoconf生成configure
- 使用automake生成Makefile.in

简介

Autoconf是一个用于生成可以自动地配置软件源代码包以适应多种Unix类系统的shell脚本的工具。Autoconf生成的配置脚本通常被称为configure。Automake是一款用于自动生成Makefile.in文件的工具。

流程


your source files --> [autoscan*] --> [configure.scan] --> configure.ac
configure.ac --.
| .------> autoconf* -----> configure
[aclocal.m4] --+---+
| `-----> [autoheader*] --> [config.h.in]
[acsite.m4] ---'
Makefile.in ------------------------------> Makefile.in

简单的例子 I

Hello.c

- autoscan : 生成configure.scan
- configure.scan >>> configure.in

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

AC_PREREQ(2.59)
AC_INIT(HH, 1.0, tcui@lsec.cc.ac.cn)
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([hello.c])

# Checks for programs.
```

简单的例子 II

```
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

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

# Checks for library functions.
AC_OUTPUT(Makefile)
```

- aclocal : 生成aclocal.m4
- autoconf: 生成configure

简单的例子 III

- 创建Makefile.am

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS=hello
hello_SOURCES=hello.c
```

- automake :生成Makefile.in

创建configure.in I

使用autoscan命令生成configure.scan。然后手工修改成configure.in。configure.in中常用的宏有:

- AC_INIT
- AM_INIT_AUTOMAKE(PACKAGE,VERSION)
- AC_CONFIG_HEADER (header-to-create): 软件包使用一个配置头文件
- AC_OUTPUT_COMMANDS (extra-cmds [, init-cmds])
- AC_PREFIX_DEFAULT(prefix)
- AC_PREFIX_PROGRAM(program)
- AC_PREREQ(version) : 管理版本号，确保足够新的Autoconf版本

创建configure.in II

- AC_PROG_AWK：按顺序查找mawk、gawk、nawk和awk，并且把输出变量AWK 的值设置成第一个找到的程序名。首先寻找mawk是因为据说它是最快的实现。
- AC_PROG_CC：确定C的编译器。如果在环境中没有设定CC，就查找gcc，如果没有找到，就使用cc。把输出变量CC设置为找到的编译器的名字。
- AC_PROG_CXX
- AC_PROG_F77
- AC_CHECK_FILES(files[, action-if-found [, action-if-not-found]])
  为每个在files中给出的文件运行AC_CHECK_FILE。

创建configure.in III

- AC_CHECK_LIB(library,function[,action-if-found[,action-if-not-found[,other-libraries]]])
  依赖于当前的语言（参见对语言的选择），试图通过检查一个测试程序是否可以和库library进行连接以获取C、C++或者Fortran 77函数function，从而确认函数function 是可以使用的。library是库的基本名字；例如，为了检查'-lmp'，就把'mp'作为参数library。
  action-if-found是一个在与库成功地进行了连接的时候运行的shell命令列表；action-if-not-found是一个在与库的连接失败的时候运行的shell命令列表。如果没有给出action-if-found，缺省的动作就是把'-llibrary'添加到LIBS中，并且定义‘HAVE_LIBlibrary’（全¿ 部使用大写字母）。

创建configure.in IV

- .. .. ..
- .. .. ..
- AC_OUTPUT(filename)

创建configure I

- aclocal：产生aclocal.m4
- autoconf: 产生configure

使用automake生成Makefile.in I

- 生成Makefile.am
- automake –add-missing
- –add-missing：告诉automake帮我们加入发行一个软件包所必备的类似于install的工具。

Makefile.am I

- AUTOMAKE_OPTIONS
  设定automake 的选项。Automake 主要是帮助开发GNU 软体的人员维护软体套件，所以在执行automake 时，会检查目录下是否存在标准GNU 软体套件中应具备的文件档案，例如'NEWS'、'AUTHOR'、'ChangeLog' 等文件档。设成foreign 时，automake 会改用一般软体套件的标准来检查。
- bin_PROGRAMS
  定义我们所要产生的执行档档名。如果要产生多个执行档，每个档名用空白字元隔开。

Makefile.am II

- hello_SOURCES
  定义'hello' 这个执行档所需要的原始档。如果'hello' 这个程式是由多个原始档所产生，必须把它所用到的原始档都列出来，以空白字元隔开。假设'hello' 这个程式需要'hello.c'、'main.c'、'hello.h' 三个档案的话，则定义hello_SOURCES= hello.c main.c hello.h 如果我们定义多个执行档，则对每个执行档都要定义相对的filename_SOURCES。

使用Makefile I

- make all
  产生我们设定的目标，即此范例中的执行档。只打make 也可以，此时会开始编译原始码，然後连结，并且产生执行档。
- make clean
  清除之前所编译的执行档及目的档(object file, *.o)。
- make distclean
  除了清除执行档和目的档外，也把configure 所产生的Makefile也清除掉。

使用Makefile II

- make install
  将程式安装至系统中。如果原始码编译无误，且执行结果正确，便可以把程式安装至系统预设的执行档存放路径。如果我们用bin_PROGRAMS 巨集的话，程式会被安装至/usr/local/bin 这个目录。
- make dist
  将程式和相关的档案包装成一个压缩档以供散播(distribution) 。执行完在目录下会产生一个以PACKAGE-VERSION.tar.gz 为名称的档案。PACKAGE 和VERSION 这两个变数是根据configure.in 档中AM_INIT_AUTOMAKE(PACKAGE, VERSION) 的定义。

使用Makefile III

- make distcheck
  和make dist 类似，但是加入检查包装後的压缩档是否正常。这个目标除了把程式和相关档案包装成tar.gz 档外，还会自动把这个压缩档解开，执行configure，并且进行make all 的动作，确认编译无误後，会显示这个tar.gz 档已经准备好可供散播了。这个检查非常有用，检查过关的套件，基本上可以给任何一个具备GNU 发展环境的人去重新编译。

致谢

Thanks!

相关文档可在*http* : *//lsec.cc.ac.cn/* ∽ *tcui*下载