深入 Makefile

曹东刚

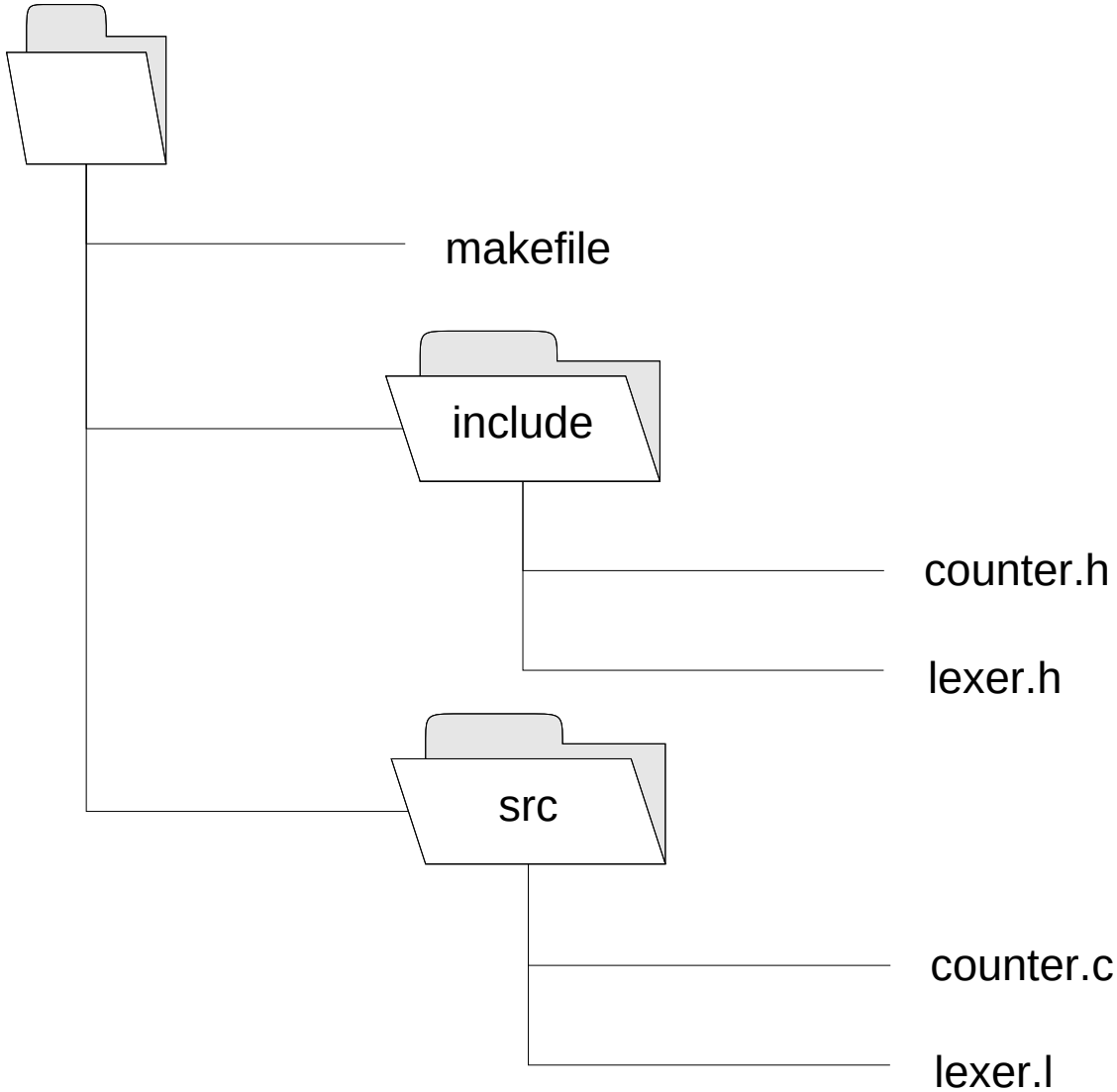
caodg@sei.pku.edu.cn

Linux 程序设计环境

http://c.pku.edu.cn/

内容提要

makefile 和源文件不在一个目录下



VPATH 和 vpath

```
1 VPATH    = src include
2 CPPFLAGS = -I include
3
4 count_words: counter.o lexer.o -lfl
5 count_words.o: counter.h
6 counter.o: counter.h lexer.h
7 lexer.o: lexer.h
```

也可以使用命令:

```
1 vpath %.c src
2 vpath %.h include
```

内置的模式规则 — 1

从.c 生成.o 文件

```
1 %.o:%.c
2     ⇥$(COMPILE.c) $(OUTPUT_OPTION) $<
```

从.l 生成.c 文件

```
1 %.c:%.l
2     ⇥@$(RM) $@
3     ⇥$(LEX.l) $< > $@
```

内置的模式规则 — 2

从.c 生成可执行文件

```
1 %:%.c
2 	$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

静态模式规则

静态模式规则：明确给出了目标列表

```
1 $(OBJECTS):%.o:%.c
2     →$(CC) -c $(CFLAGS) $< -o $@
```

应当尽可能使用静态模式规则，明确写出目标之间的依赖关系

后缀规则

后缀规则是定义隐式规则的初始方式.

```
1 .c.o:
2     $(COMPILE.c) $(OUTPUT_OPTION) $<
```

等价于:

```
1 %.o:%.c
2     $(COMPILE.c) $(OUTPUT_OPTION) $<
```

单后缀规则

单后缀规则常用于生成可执行文件

```
1 .p:
2 	$(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

内置规则的结构

```
1 %.o: %.c
2     ↦$(COMPILE.c) $(OUTPUT_OPTION) $<
```

变量:

```
1 COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) \
2     ↦$(TARGET_ARCH) -c
3 CC = gcc
4 OUTPUT_OPTION = -o $@
```

自动生成依赖

问题: C 语言源文件依赖头文件, 头文件之间也有依赖, 能否自动生成并管理这种依赖?

方法: 利用 gcc 的编译选项gcc -M

gcc 编译开关

```
~ $ echo "#include <stdio.h>" > stdio.c
~ $ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdio.h \
  /usr/include/_ansi.h \
  /usr/include/newlib.h \
  /usr/include/sys/config.h \
  /usr/include/machine/ieeefp.h \
  /usr/include/cygwin/config.h \
  /usr/lib/gcc/i686-pc-cygwin/3.4.4/include/stddef.h \
  /usr/lib/gcc/i686-pc-cygwin/3.4.4/include/stdarg.h \
  /usr/include/sys/reent.h \
  /usr/include/_ansi.h \
```

方法

- 为每个源文件 (filename.c) 生成一个依赖文件, 如 filename.d
- 该 filename.d 里面保存了 filename.c 和 filename.d 对头文件的依赖, 如:

```
counter.o counter.d : src/counter.c \
include/counter.h include/lexer.h
```

- 用包含命令 (include) 将所有 filename.d 包含在 makefile 中

生成 filename.d

```
1  include $(subst .c,.d,$(SOURCES))
2
3  %.d: %.c
4      →$(CC) -M $(CPPFLAGS) $< > $@.$$$$;          \
5      →sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' < $@.$$$$ > $@; \
6      →rm -f $@.$$$$
```

make 变量

- make 实际上包含两种语言：描述依赖关系的语言，以及进行文本替换的宏语言
- 变量名字几乎可以是任何字符，除了 `:` `#` `=`
- 变量大小写敏感
- 变量可以通过 `${CC}` 或者`$(CC)` 访问

make 变量

- 通常如果用户希望通过命令行或环境变量改变某变量的值，则这样的常量名字全部大写，单词之间通过下划线分隔
- 全部小写的变量只用于 makefile 文件内部
- 通常用变量引用外部程序名

变量扩展

- 简单扩展，通过“:=”赋值. 赋值语句被 make 读到的时候，即对赋值符右侧进行扩展

  MAKE_DEPEND := $(CC) -M

  如果 CC 没有定义，则赋值的结果为: ␣-M

- 递归扩展，通过“=”赋值. 只有当变量被 make 用到的时候，才对赋值符右侧进行扩展

  MAKE_DEPEND = $(CC) -M

  CC 的定义可以在MAKE_DEPEND 的后面

宏

```
1  define create-jar
2   @echo Creating $@...
3   $(RM) $(TMP_JAR_DIR)
4   $(MKDIR) $(TMP_JAR_DIR)
5   $(CP) -r  $^ $(TMP_JAR_DIR)
6   cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
7   $(JAR) -ufm $@ $(MANIFEST)
8   $(RM) $(TMP_JAR_DIR)
9  endef
10
11 $(UI_JAR): $(UI_CLASSES)
12     $(create-jar)
```

函数

GNU make 支持函数

- 函数的定义和宏的定义类似
- 函数的调用和变量的引用类似，只是要加上逗号分隔的参数列表

用户自定义函数

```
1  # cygwin script
2  AWK  := awk
3  KILL := kill
4
5  # $(kill-acroread)
6  define kill-acroread
7    @ ps -aw |                                \
8    $(AWK) ' /AcroRd32/ {                     \
9                print "Killing " $$3;         \
10               system( "$(KILL) " $$1 )      \
11           }'
12 endef
```

用户自定义函数

```
1  AWK  := awk
2  KILL := kill
3  # $(call kill-program, awk-pattern)
4  define kill-program
5      @ ps -aw |                          \
6      $(AWK) ' /$1/ {                     \
7          print "Killing " $$3;           \
8          system( "$(KILL)  " $$1 )       \
9      }'
10 endef
```

调用: `$(call kill-program, "AcroRd32")`

内置函数

GNU make 有若干内置函数，用于对变量进行操作. 函数使用语法: `$(function-name arg1[, argn])`

- 字符串操作函数
- 文件名操作函数
- 流程控制函数
- 用户自定义函数
- 其它函数

字符串函数

$(filter pattern...,text)，text: 空格分隔的单词串，返回完整匹配的单词. 例:

```
1 words := he the hen other the%
2 get-the:
3     @echo %he matches : $(filter %he,$(words))
```

$(subst search-string,replace-string,text)，将 text 中的 search-string 替换为 replace-string. 例:

```
1 sources := count_words.c counter.c lexer.c
2 objects := $(subst .c,.o,$(sources))
```

命令

- 目标之后以制表键 TAB 开头的行为命令
- 命令本质上是一个一行的 shell 脚本
- makefile 中的命令在子 shell 中执行
- 命令执行的 shell 缺省是/bin/sh, 由 make 的变量 SHELL 控制
- make 命令执行时继承父 shell 的除 SHELL 外的所有变量
- 需要由 shell 扩展的参数应该用$$n 的形式

命令解析

make 看到一个合法命令之后，即转入命令解析模式，建立一个一行脚本

- 以制表键缩进的行被认为是命令行
- 空行被忽略
- 以# 开始的行 (之前可能有空格，但没有制表键) 被认为是 makefile 注释，被忽略
- 条件处理指令，如 ifdef 和 ifeq在命令脚本中处理

注释

- makefile 注释: 不在命令中的以# 开头 (前面可有空格) 的行, 被 make 忽略处理
- shell 注释: 在命令中, 以制表符加# 开头的行, make 要对其进行扩展, 然后交给 shell 处理; 每个注释都会启动一个子 shell

注释示例

```
1  #this is make comment $(PWD)
2  print-pwd:
3      #
4      # this is shell comment
5      # PWD = $(PWD)
6      # $(findstring /e/home/Make,$(PWD))
7      #
```

长命令

由于每个命令在单独的一个子 shell 中执行，如果若干命令要在一起执行，需要用反斜杠\ 连接各行

例：错误的写法

```
1 filie_list:
2     for f in logic ui
3     do
4         echo $f/*.java
5     done > $@
```

长命令

正确的写法

```
1  filie_list:
2      →for f in logic ui; \
3      →do \
4      →   →echo $$f/*.java; \
5      →done > $@
```

内容提要

为什么要自动生成 Makefile

- 可移植性: 适应不同硬件平台和 Unix 系统
  - 机器字大小、工具、语言、服务器、设置等
    - 例如, bcopy 与 memcpy
- 派生依赖性: C 语言源文件之间的依赖关系

几种主要生成工具

- makedepend
- Imake
- autoconf
- automake

makedepend

- 随 X Window 系统发布
- 在解决源码依赖方面最快，最有效
- 只对 C 项目，分析 C 源文件的#include 等宏指令
- 将依赖关系生成到 Makefile 中

makedepend 示例

通常将 makedepend 作为 Makefile 的一个目标通过 make 调用

```
1  SRCS = Main.c Print.c
2
3  all: print
4  print: Main.o Print.o
5      gcc -o $@ $^
6
7  depend:
8      makedepend ${SRCS}
```

makedepend 更新后的 Makefile

```
1  SRCS = Main.c Print.c
2  all: print
3  print: Main.o Print.o
4      gcc -o $@ $^
5  depend:
6      makedepend $(SRCS)
7  # DO NOT DELETE
8  Main.o: Print.h
9  Print.o: /usr/include/stdio.h /usr/include/features.h
10 Print.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
11 Print.o: /usr/lib/gcc-lib/i486-linux/3.3.5/include/stddef.h
12 Print.o: /usr/include/bits/types.h /usr/include/bits/wordsize.h
13 Print.o: /usr/include/bits/wchar.h /usr/include/gconv.h
14 Print.o: Print.h
```

autoconf

GNU 项目发布约定

- 编译过程分两个步骤：产生编译配置、编译
- 项目根目录下有一个configure脚本，用于生成 Makefile
- configure 读取Makefile.in文件，生成 Makefile 及其他可能辅助的文件
- 调用标准的 make 进行编译

autoconf 支持的项目目录结构

- flat: 所有文件都位于同一个目录中，且没有子目录
  - 最简单
- shallow: 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
- deep: 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
  - 最复杂

autoconf 工作流程 — 1

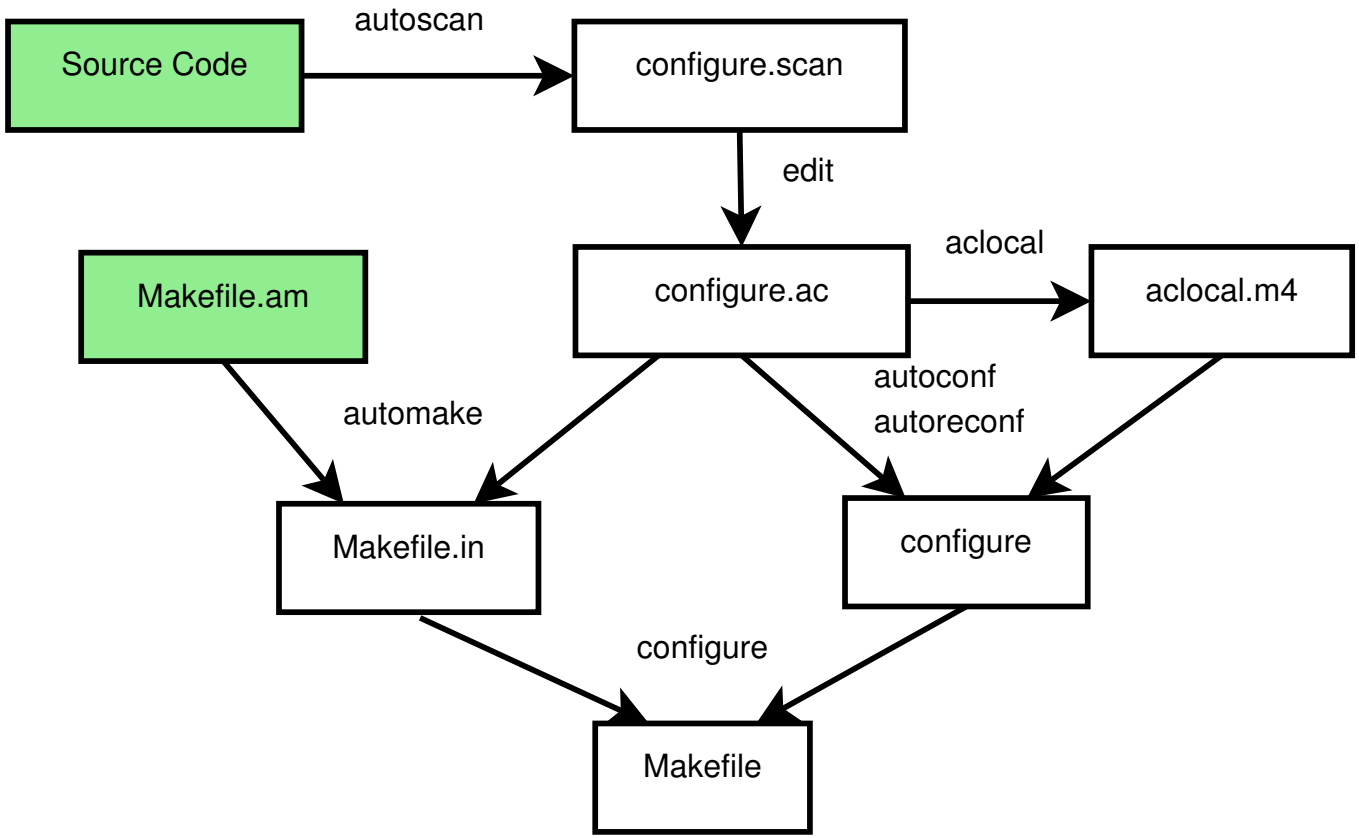
autoconf 最核心的文件是 configure.ac 文件和 Makefile.in 文件. 程序员可以直接手工创建, 也可以通过工具自动生成这两个文件的原型

- 建立 configure.in 文件
  1. 运行autoscan, 生成 configure.scan 文件
  2. 将 configure.scan 文件重命名为 configure.ac，并修改 configure.ac

autoconf 工作流程 — 2

- 建立 Makefile.in 文件
  1. 建立 Makefile.am 文件
  2. 运行automake, 生成 Makefile.in
- 生成 configure 文件
  1. 运行aclocal生成 aclocal.m4
  2. 运行autoconf生成 configure

autoconf 工作流程图



示例

一个简单示例，项目只包含 helloworld.c 文件

```
1  int main(int argc, char ** argv)
2  {
3      printf("Hello, Hello world!\n") ;
4      return 0 ;
5  }
```

示例：configure.ac

```
1  #                                               -*- Autoconf -*-
2  # Process this file with autoconf to produce a configure script.
3  AC_PREREQ(2.59)
4  AC_INIT(helloworld, 0.1, caodg@sei.pku.edu.cn)
5  AC_CONFIG_SRCDIR([helloworld.c])
6  # Checks for programs.
7  AC_PROG_CC
8  # Checks for libraries.
9  # Checks for header files.
10 # Checks for typedefs, structures, and compiler characteristics.
11 # Checks for library functions.
12 AC_OUTPUT(Makefile)
```

示例：Makefile.am

```
1 AUTOMAKE_OPTIONS=foreign
2 bin_PROGRAMS=helloworld
3 helloworld_SOURCES=helloworld.c
```