Makefile 教程

- 感谢徐海兵翻译
- 主讲人：秦风岭

主要内容

- 规则的命令
- Makefile 的变量
- 条件执行
- 隐含规则
- Make 项目管理实例

规则的命令

- 命令简介：

  规则的命令由一些 shell 命令行组成，它们被一条一条的执行。规则中除了第一条可以紧跟在依赖列表之后使用分号隔开的命令以外，其它的每一行命令行必须以 [Tab] 字符开始。多个命令行之间可以有空行和注释行。通常系统中可能存在多个不同的 shell。但在 make 处理 Makefile 过程时，如果没有明确指定，那么对所有规则中命令行的解析使用“ /bin/sh” 来完成。

- 示例：

  文件： Makefile

  命令： make command1

- 命令的执行

  规则中，当目标需要被重建时。此规则所定义的命令将会被执行，如果是多行命令，那么每一行命令将在一个独立的子 shell 进程中被执行。因此，多行命令之间的执行是相互独立的，相互之间不存在依赖。

- 示例：

  文件： Makefile

  命令： make command2 command3

- make 的递归执行

  make 的递归过程指的是：在 Makefile 中使用“ make” 作为一个命令来执行本身或者其它 makefile 文件的过程。递归调用在一个存在有多级子目录的项目中非常有用。

- 示例：

  文件： Makefile

  命令： make command_recusive1

  make command_recusive2

Makefile 中的变量

- 在 Makefile 中，变量是一个名字，代表一个文本字符串。可以用于目标，依赖和命令中。引用变量的地方，变量会被它的值所取代。类似 C 语言中的宏。
- 变量的特征：

  1.Makefile 中变量和函数的展开（除规则命令行中的变量和函数以外），是在 make 读取 makefile 文件时进行的。

  2. 变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表和所有我们能够想到的事物。

  3. 变量名是不包括“ :”、“ #”、“ =”、前置空白和尾空白的任何字符串。（示例 :Makefile 定义的规则 print_newvar ）

  4. 变量名是大小写敏感的。传统做法是变量名是全采用大写的方式。

  5. 另外有一些变量名只包含了一个或者很少的几个特殊的字符（符号）。称它们为自动化变量。

- 变量的引用

  当我们定义了一个变量之后，就可以在 Makefile 的很多地方使用这个变量。变量的引用方式是：“$(VARIABLE_NAME)”或者“ ${ VARIABLE_NAME }” 来引用一个变量的定义。美元符号“ $” 在 Makefile 中有特殊的含义，所有在命令或者文件名中使用“ $” 时需要用两个美元符号“ $$” 来表示。

  注意： Makefile 中对一些简单变量的引用，我们也可以不使用“ ()” 和“ {}” 来标记变量名，而直接使用“ $x” 的格式来实现，此种用法仅限于变量名为单字符的情况。另外自动化变量也使用这种格式。

- 变量的定义

变量的定义有递归展开，直接展开，条件赋值。

1. 递归展开式变量

这一类型变量的定义是通过“ =” 或者使用指示符“ define”定义的，这种变量的引用 , 在引用的地方是严格的文本替换过程 , 此变量值的字符串原模原样的出现在引用它的地方。

优点：

这种类型变量在定义时 , 可以引用之前没有定义的变量。

缺点：

1. 使用此类变量 , 可能会由于出现变量的递归定义而导致 make 陷入到无限的变量展开过程中 , 最终使 make 执行失败。

例如： CFLAGS = $(CFLAGS) –O

2. 此类变量定义中如果使用了函数 , 那么包含在变量值中的函数总会在变量被引用的地方执行。

示例：

文件： Makefile

命令： make print_curdir1

- 直接展开式变量

  这种风格的变量使用“ :=” 定义。在使用“ :=” 定义变量时，变量值中对其他量或者函数的引用在定义变量时被展开。所以变量被定义后就是一个实际需要的文本串，其中不再包含任何变量的引用。

  示例：

  文件： Makefile

  命令： make print_curdir2

- 条件赋值变量

  这种风格的变量使用“ ?=” 定义。只有此变量在之前没有赋值的情况下才会对这个变量进行赋值。

  示例：

  文件： compile_modules.mak

  变量： COMPILE_DIR?= $(shell ls $(LS_FLG) | grep ^d | awk '{print $$8}')

- 变量取值

1. 在运行 make 时通过命令行选项来取代一个已定义的变量值。例如 :stbconfig.mak 中 make MARKETNAME = SS01_DANDONG

2. 在 makefile 文件中通过赋值的方式或者使用“ define” 来为一个变量赋值。

3. 将变量设置为系统环境变量。所有系统环境变量都可以被 make 使用。

4. 自动化变量 , 在不同的规则中自动化变量会被赋予不同的值。

5. 一些变量具有固定的值。（ 在隐含规则介绍 ）

- 追加变量值

  通常，一个通用变量在定义之后的其他一个地方，可以对其值进行追加。在 Makefile 中使用“ +=” 来实现对一个变量值的追加操作。例如： objects += another.o 。

- override 指示符

  我们可以在执行 make 时通过命令行方式重新指定这个变量的值，命令行指定的值将替代出现在 Makefile 中此变量的值。如果不希望命令行指定的变量值替代在 Makefile 中的变量定义，那么我们需要在 Makefile 中使用指示符“ override” 来对这个变量进行声明。

Makefile 的条件判断

条件语句可以根据一个变量的值来控制 make 执行或者忽略 Makefile 的特定部分。条件语句可以是两个不同变量、或者变量和常量值的比较。

- 基本语法

格式 1：

CONDITIONAL-DIRECTIVE

TEXT-IF-TRUE

endif

格式 2：

CONDITIONAL-DIRECTIVE

TEXT-IF-TRUE

else

TEXT-IF-FALSE

endif

- 关键字“ ifeq”

  此关键字用来判断参数是否相等，格式如下：

  ifeq (ARG1, ARG2)

  ifeq 'ARG1' 'ARG2'

  ifeq "ARG1" "ARG2"

  ifeq "ARG1" 'ARG2'

  ifeq 'ARG1' "ARG2"

  替换展开“ ARG1” 和“ ARG1” 后，对它们的值进行比较。如果条件为真将“ TEXT-IF-TRUE” 作为 make 要执行的一部分，否则将“ TEXT-IF-FALSE” 作为 make 要执行的一部分。

- 示例：

  stbconfig.mak 中的 ifeq "$(MARKETNAME)" "GENERALMARKET"

- 关键字“ ifneq”

  此关键字是用来判断参数是否不相等 , 格式为 :

  ifneq (ARG1, ARG2)

  ifneq 'ARG1' 'ARG2'

  ifneq "ARG1" "ARG2"

  ifneq "ARG1" 'ARG2'

  ifneq 'ARG1' "ARG2"

  关键字“ ifneq” 实现的条件判断语句和“ ifeq” 相反。

  首先替换并展开“ ARG1” “ARG1” 和 , 对它们的值进行比较。如果不相同 (条件为真) 则将“ TEXT-IF-TRUE” 作为 make 要执行的一部分 , 否则将“ TEXT-IF-FALSE” 作为 make 要执行的一部分。

- 关键字“ ifdef” 和“ ifndef”

  1. 关键字“ ifdef” 用来判断一个变量是否已经定义。格式为 :

  ifdef VARIABLE-NAME

  2. 关键字“ ifndef” 实现的功能和“ ifdef” 相反。格式为 :

  ifdef VARIABLE-NAME

make 的隐含规则

在 Makefile 中重建一类目标的标准规则在很多场合需要用到。例如 : 根据 .c 源文件创建对应的 .o 文件。

“ 隐含规则”为 make 提供了重建一类目标文件通用方法 , 不需要在 Makefile 中明确地给出重建特定目标文件所需要的细节描述。例如 : 典型地 make 对 C 文件的编译过程是由 .c 源文件编译生成 .o 目标文件。 当 Makefile 中出现一个 .o 文件目标时 ,make 会使用这个通用的方式将后缀为 .c 的文件编译成为目标的 .o 文件。内嵌的“隐含规则”在其所定义的命令行中 , 会使用到一些变量 (通常也是内嵌变量)。我们可以通过改变这些变量的值来控制隐含规则命令的执行情况。例如 : 内嵌变量“ CFLAGS” 代表了 gcc 编译器编译源文件的编译选项 , 我们就可以在 Makefile 中重新定义它 , 来改变编译源文件所要使用的参数。尽管我们不能改变 make 内嵌的隐含规则 , 但是我们可以使用模式规则重新定义自己的隐含规则。

- make 的常见内嵌隐含规则

  1. 编译 C 程序

  “N.o” 自动由“ N.c” 生成 , 执行命令为 “$(CC) -c $(CPPFLAGS) $(CFLAGS)” 。

  2. 编译 C++ 程序

  “N.o” 自动由“ N.cc” “N.C” 生成 , 或者执行命令为 “$(CXX) -c $(CPPFLAGS)$(CFLAGS)” 。 建议使用 “ .cc” 作为 C++ 源文件的后缀 , 而不是“ .C”

- 隐含变量
- 内嵌隐含规则的命令中，所使用的变量都是预定义的变量。我们将这些变量称为“隐含变量”。这些变量允许被修改，在 Makefile 中通过命令行参数或者设置系统环境变量的方式来对它进行重定义。

  常见变量： AR， CC， CXX， RM
- 常见变量参数：
  ARFLAGS， CFLAGS， CXXFLAGS

- 模式规则

  模式规则类似于普通规则。只是在模式规则中，目标名中需要包含有模式字符“ %”，包含有模式字符“ %”的目标被用来匹配一个文件名，“%”可以匹配任何非空字符串。规则的依赖文件中同样可以使用“ %”，依赖文件中模式字符“ %”的取值情况由目标中的“ %”来决定。例如：对于模式规则“ %.o：%.c”，它表示的含义是：所有的 .o 文件依赖于对应的 .c 文件。

- 模式规则的格式

  %.o : %.c ; COMMAND...

  这个模式规则指定了如何由文件“ N.c” 来创建文件“ N.o”, 文件“ N.c” 应该是已存在的或者可被创建的。模式规则中依赖文件也可以不包含模式字符“ %” 。当依赖文件名中不包含模式字符“ %” 时 , 其含义是所有符合目标模式的目标文件都依赖于一个指定的文件 (例如 :%.o : debug.h, 表示所有的 .o 文件都依赖于头文件“ debug.h” 。)

  示例：

  文件： default_ovtc.mak

Make_Project

- 需求说明：

  1. 支持多模块，多项目，多平台配置，移植方便。
  2. 支持文件夹遍历。支持自定义和默认方式编译，支持单个库生成和多个库生成。
  3. 支持 C/C++ 文件自动包含，目录默认结构为 code 、 include ，支持自定义编译 C/C++ 文件方式和默认方式。
  4. 第三方移植库放置目录 ext/entire_lib 下面，后缀名 .lib
  5. 项目库的生成链接路径为 app/lib 下面，后缀名 .a
  6. 支持源码和库的编译方式。便于和 svn 代码权限管理整合。
  7. 支持文件系统和应用程序生成。

- 文件说明：

compile_modules.mak

用于编译多级子目录的编译模板，生成一个库。

default_recursive.mak

用于编译多级子目录的编译模板，生成单独的库。

default_ovtc.mak

用于编译 C/CPP 文件的编译模板。

generic.mak

编译选项，平台选项，包含头文件等通用的编译模板，编译时 Makefile 必须要包含的。

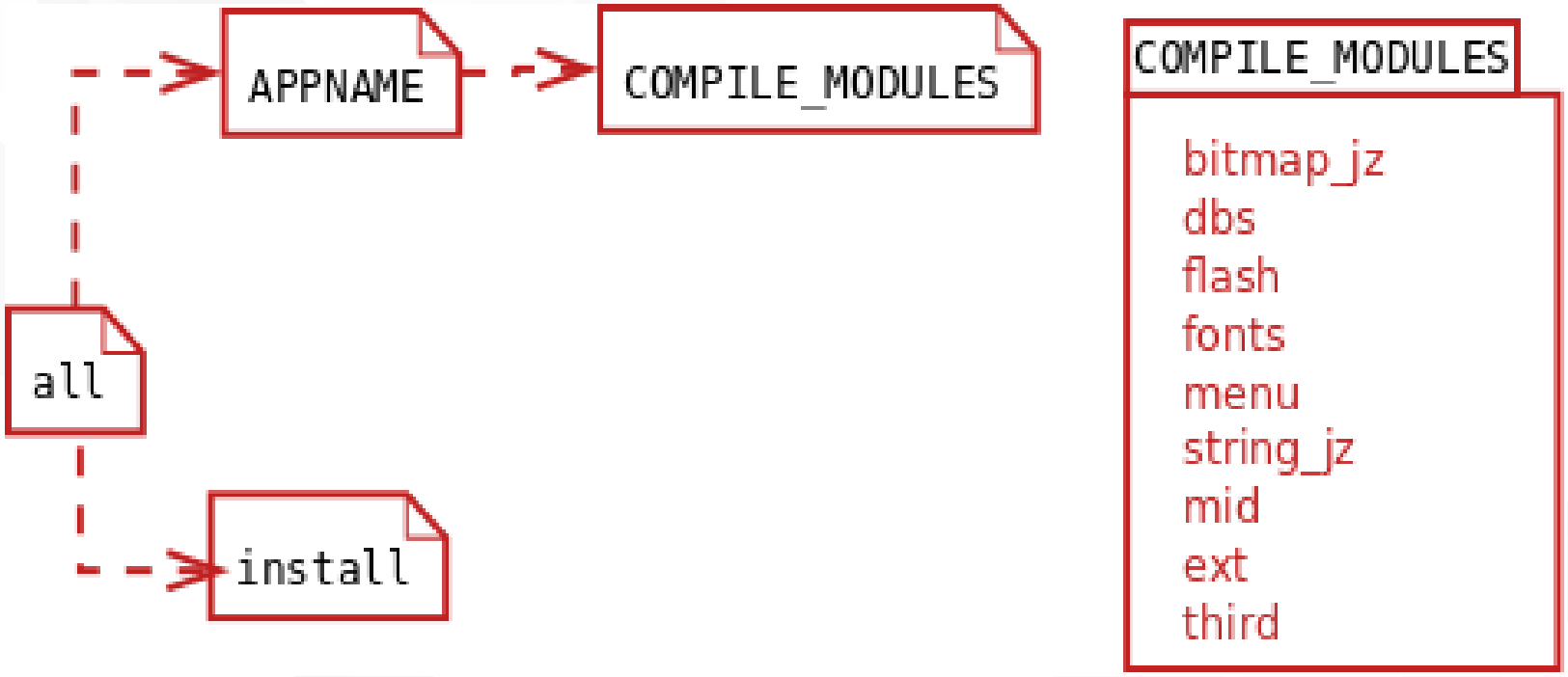
platform_xxxx.mak

和平台相关的编译选项、 binutils 、 SDK 头文件和库的编译模板。

stbconfig.mak

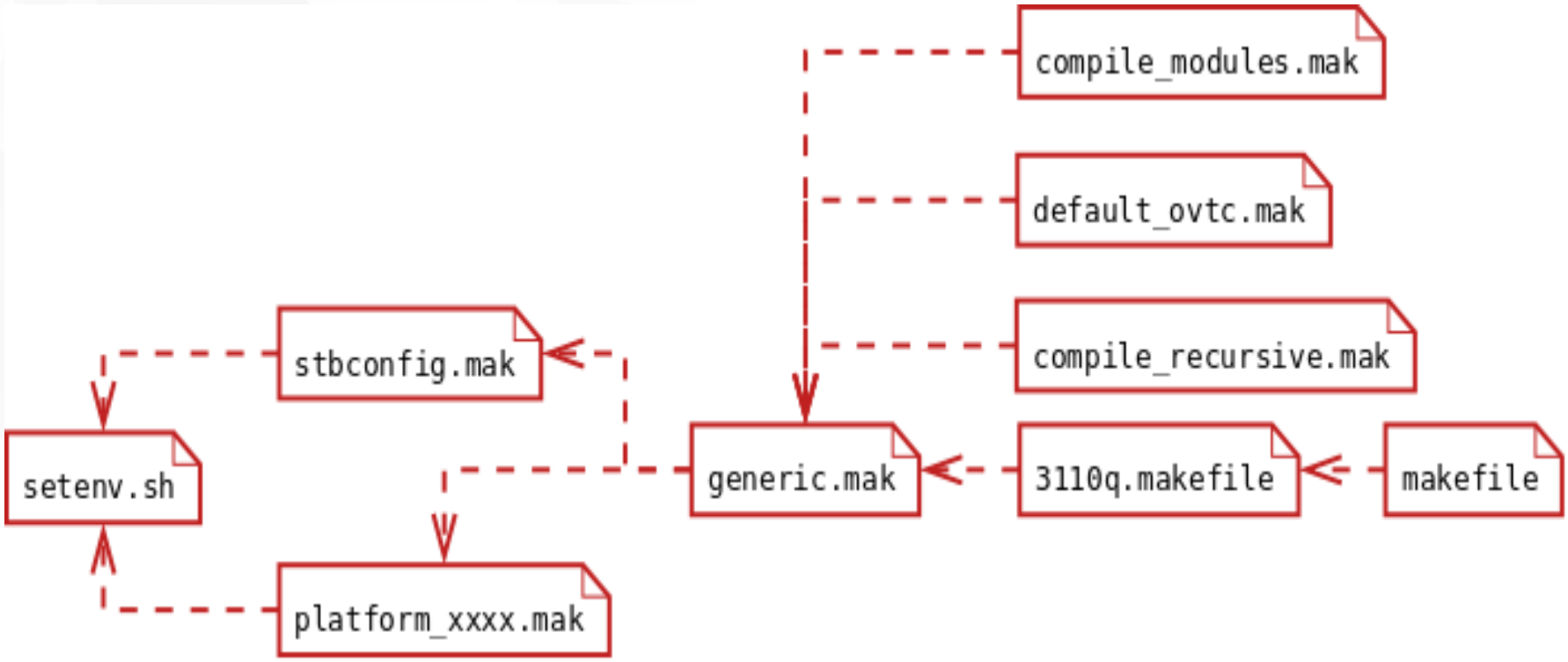
项目编译的配置模板。

应用程序生成依赖图


APPNAME
COMPILE_MODULES
all
install
COMPILE_MODULES
bitmap_jz
dbs
flash
fonts
menu
string_jz
mid
ext
third

ORIENT VIEW TECHNOLOGY

makefile 依赖关系图



- 使用方式：
  1. 配置 stbconfig.mak
  2. 执行 source setenv.sh
  3. 生成应用程序，执行 make
     生成文件系统，执行 make mkfs

谢谢大家！

- 联系方式
- 手机： 15010404682
- 邮箱： qin_fengling@126.com
- QQ： 415508683