最近一直在搞Makefile文件的编辑，一直想通过自己的心得体会与广大网友分享。Linux学习者只有参与的大多人当中去，才能体会到乐趣。同时慢慢培养自己的学习linux的兴趣爱好。与广大网上爱好者互动。

Linux的GNU计划：

Linux下构建自己的开源软件使用的是linux下自己带的强大的工具。Autoconf libtoolize 和automake .这是开源软件必须的基本工具。

如果使用了autoconf和automake，除了编译应用程序，用户并不需要有这些工具。使用这些工具的目的是创建能在用户环境使用的、可移植的shell脚本和Makefile文件。

Autoconf实际上是一个工具集，其中包含aclocal、autoheader和autoconf等可执行文件。这些工具生成一个可移植的shell脚本—configure，configure和软件包一起发布给用户。

它探查编译系统，生成 Makefile文件和一个特殊的头文件config.h。由configure生成的文件能适应用户系统的特定环境。

configure脚本从一个称为 Makefile.in的模板文件生成每个Makefile文件。

而Makefile.in 有Makefile.am 生成。

Configure.in 有开发者自己手动修改。

Makefile.am 是由开发者自己手写。

Libtool软件包是第三个重要的GNU工具，它的作用是确定共享库在特定平台上的特性。因为共享库在不同平台上可能会有所不同。

上述是自动生成Makefile的概括。以后再细讲。

手动书写Makefile：

手动书写顾名思义就是自己跳过configure.Scan configure.in configure Makefile.am Makefile.in 的生成过程。直接书写Makefile 这种方式只能用于相对简单的源代码。如有几个，几十个或者上百个源文件时，自己编写Makefile往往是可行的，但是如果我们所编写的源文件有几千，几万，几十万甚至更多时，显然手动书写Makefile不是个明智之举。

我们在手动书写Makefile时（此心得面向是略微看过Makefile的编写规则。如果没看过，可以参考这个文章“大家一起写Makefile”里面有蛮详细的介绍）一个简单的makefile的书写。

我们只需要把握编译选项，链接选项目标依赖文件等

给大家分享下自己的自己对编译和链接的领悟（勿拍砖）编译就是将检测各种源文件的语法是否正确。我们称之为是从.c🡪到.o的过程。这个过程中只是检测语法错误。而链接过程我称之为从.o🡪.a或者.la 然后从.a(la)到可执行文件的过程。其中有心的各位会发现.a或者.la是一个很大的文件。笔者以为这只是个中间连接文件。就是把所有的函数。调用子函数。以及嵌套调用的函数在中间文件里铺开的过程。

现在以笔者最近的一个例子作为分析进入到Makefile文件的编写中；

CROSSCOMPILE :=arm-linux-

CC :=$(CROSSCOMPILE)gcc

LD :=$(CROSSCOMPILE)ld

LDFLAGS := -lm

CLFAGS := -Wall -02 -c -g

CFLAGS += -I $(PWD)

OBJS :=bitstream.o\

mask.o\

mmask.o\

mqrspec.o\

qren.o\

qrencode.o\

qrinput.o\

qrspec.o\

rscode.o\

split.o

.PHONY:%.o

%.o:%.c

$(CC) $(CFLAGS) $< -o $@

all:$(OBJS)

$(CC) $(LDFLAGS) -o qrencode $^

.PHONY:clean

clean:

rm -rf $(OBJS) qrencode

很明显这是个属于极其简单的Makefile文件。有初学者会认为。很多我拿到的代码里已经有了Makefile 我干嘛还有学习Makefile编写呢？我觉得对于只想移植。不想开发的朋友来说或许足够了。但是对于我们要往深水区来游的小同志来说。显然Makefile的编写和shellscript 的编写。很明显是我们的必修课。这是我们对大工程把握水平的体现。

结束废话：开始分析Makefile的写法：

首先进come到我们面前的是CROSSCOMPILE :=arm-linux

CROSSCOMPILE :=这是个变量定义而且是一个可以传递到底层Makefile文件的定义变量。而且必须是传递到底层。然而一旦传递的底层。它必然是个环境变量。所以可以选择export linux下的命令来全局化。所以Makefile中变量的定义是如果需要引入到环境变量中请选择全程大写。如果仅是这个Makefile中使用。不会用到子目录Makefile中，请选择小写。当然即使大写我们也需要配合export后才可传递到底层Makefile

按照进程的思想，当前Makefile层是一个进程，调用下一层Makefile的过程就是开启一个新的进程。调用下一个进程使用include + Makefile文件

第二个符号：= 这是个环境变量的定义。

= 表示定义一个新的变量。：=表示是如果前边定义过了则选择是覆盖之前的定义。+=表示追加。？=则是表示如果询问定义，既是如果前边定义了，这里不在定义。

而arm-linux-gcc这里所说的是arm下交叉编译工具而gcc则是linux下的编译工具

当然如果选择的是C++ 则是g++ 和arm-linux-g++ 这里采用C语言。毕竟编译和链接过程跟C++ 类似。

上述二位是非常敬畏的了编译器。因为linux下它的出现解决了编译工具的问题。

而编译对应的是链接。链接分别对应的是arm-linux-ld 和ld 当然不同的交叉编译版本不同。可能略有差别。只有交叉编译的概念。如有误解请问伟大的度娘或者强大的360

CC以及LD这里定义的是编译器和链接器这里采用可以方便修改的做法。用的是变量替代。

往下走时是我们的参数传递。我们知道在linux命令行下使用gcc 或者arm-linux-gcc时我们也会有参数传入，这里就是我们所说的参数传递。参数传递的意思则是

-lm则是告诉编译器如果数学库找不到请到/libs下找。我们是需要数学库的

数学函数位于libm.so库文件中（这些库文件通常位于/lib目录下），-lm选项告诉编译器，我们程序中用到的数学函数要到这个库文件里找

一般参数很多我们这里只介绍必须的。很多是缺省的。就是了解有这回事。未必一定要用的

-c 告诉编译器你编译的对象是.c

-o 告诉编译器编译无错后输出文件格式为.o

-static告诉编译器不用动态库，只是使用静态库（动态库和静态库的去一个是.so一个是.a）这个编译后会产生很大的文件。一般不使用

与之对应的这是只是使用动态库–share 这一般是我们的选择。

还有就是-S生成汇编文件

-E生成预处理文件

-wall 启动警告检测

-02 是编译器的级别一共 00 01 02 03 四个编译器级别。-00表示没有优化,-01为缺省值，-03优化级别最高

-g 为在编译的时候，产生条是信息

-I指定头文件目录

$(PWD) 表示执行linux下命令 pwd　一个生成当前位置的变量。并替代了。

其余的参数如果有需要的可以自己查看拉。毕竟编译器是个很强大的。

参数的废话结束了那么我们进入到我们的目标定义

目标定义的一个显著符号则是：

如ALL： $(OBJS) 这里说明我们这就是编译的总目标

Makefile文件里可以定义编译一个目标，也可以定义编译多个目标（设计伪定义）

就是下面使用的关键字.PHONY

Makefile文件中总是默认第一个定义的目标作为总体的生成目标。使用上诉关键字后编译器就是知道了这里不是，只是一个伪目标。

通常配合着clean distclean 来使用。那就是后边的删除

Clean:

rm –rf $(OBJS) qrencode

这里指的是删除所有生成的目标文件

rm是linux下一个命令。不在叙述。

下边进入到奇怪的字符的解释：

$ 是一个用了很多遍的字符是一个具有替代功能的字符

如$(OBJS)就是替代我们定义的所有的变量。

还有就是$@ 是在编译过程中的所生成的目标文件

$<是我们所依赖的文件

$(CC) $(CFLAGS) $< -o $@

上述的语句就是 CC替代arm-linux-gcc CGLAGS 用来替代我们的参数传递

$<表示依赖所有的.C 文件–c的参数传递在CFLAGS -o目的是生成.o 文件

所以S@则是目标文件

还有语句

%.o:%.c

“%”的意思是匹配零或若干字符

就是自动匹配这里的o和c 文件

$(CC) $(LDFLAGS) -o qrencode $^

这句话的解释则是编译生成qrencode的可执行文件。

$^的意思则是自动依赖所有的文件

所以看者不知道是否明白了一个手写Makefile的语法写的过程？

其实就是把我们平时写的arm-linux-gcc –c mian.c –o mian.o

原理类似。

至于Makefile中的自动查询的方式：

我们知道上述是一个隐式的Makefile的编写方式。

Makefile中\* 为通配符这个玩过Linux肯定知道。还有就是换行符\

特别声明下如果在命令下加入@ 表示该条命令只是执行但不输出该条命令。

输出执行命令结果。

显式的编写方式往往看者很明白。但是很繁琐。

隐式的写法就是使用了自动匹配和静态编写的方式写成了上诉的Makefile

当然复杂的makefile还会有linux的shell命令的使用

还有就是我们的字符命令这里的简单书写个人感觉暂时用不着。

还有就是简单命令的使用

如 ifeq

If neq

#ifdefne

Else

#end if

VPATH 用于声明代码的路径

Vpath %.c file

Vpath %.c file1

路径声明优先搜索file 和file1

Makefile下每条命令下执行正确了都会返回0 错误返回非0 然后检测退出。

Make –t =touch 更新目标文件时间但不进行新的编译。

Make –q =question 如果目标不在。打印输出。目标在，不显示

Make –f=file指定编译文件

这里感觉使用的就是这么几个。

至于字符函数的处理。笔者的功力真的没有那么深。暂时手动写Makefile过程中未使用过。

自动生成Makefile的过程为：

1. 执行autoscane 生成configure.scan
2. 然后手动修改configure.scan为configure.in 有的平台直接为configure.ac
3. 修改configure.in 里面的内容。怎么修改。下文讲，这给给看客一个大致全过程。
4. 执行autoheader生成文件configure.h.in（现在一般改为configure.ac）。configure.in里有宏AC\_CONFIG\_HEADER()时用
5. 运行libtoolize生成一些libtool 的文件这些文件跟平台适应性有关系。
6. 运行autoconf 执行过程中会生成configure 文件
7. 手动写makefile.am 写的过程下文详细介绍
8. 执行automake –a 将makefile.am 生成Makefile.in 同时生成选项可以补齐文件config.guess，config.sub，install-sh，missing，depcomp
9. 然后执行./configure 后边会有缺省的参数比如

生成文件生成config.status，config.h，makefile

对于符合GNU的源码通常目录下包含那些文件又有哪些组成？

必须有一个README主要是介绍源码包的详细信息。

一个INSTALLIN文件介绍安装包的使用方法和安装方式

一个configure文件主要配合Makefile.in 生成Makefile文件。

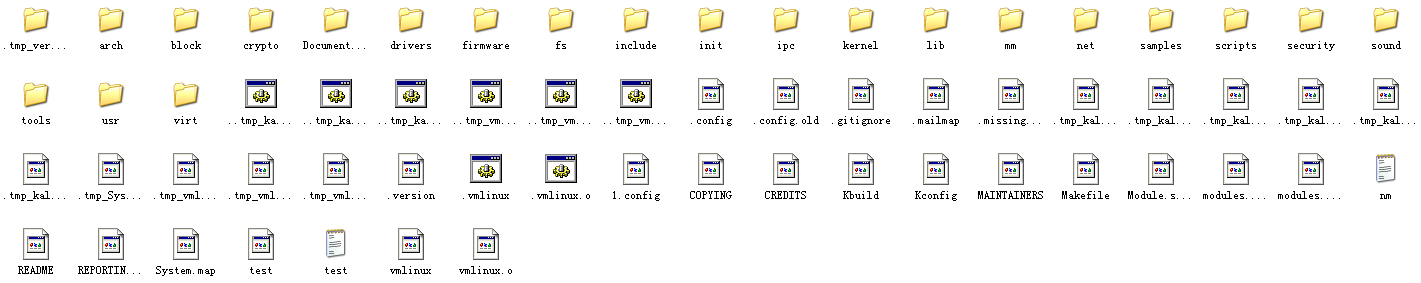
一个CONPYING 包含版本信息

一个CHANGELOG 包含软件的变化过程。

通常我们做顶层目录可以包含任何的信息，然后才是做底层的目录。分别设置其配置文件

我们用linux-2.26.32来分析：

顶层目录：



这里我们就可以看到一个大型工程里顶层目录的基本文件：

README CONPYING 等当然很多文件是我们后期执行make zImage时生成的。

但是我们通过这里见识到了源码的分类存放和基本的配置文件。

Arch架构下存放各种的关于架构的信息如：



当然linux系统下的跟我们这里讲的还是有差别的。Linux下不可能手动书写makefile 而是采用makemenuconfig 或者make Xconfig 来配置不同的执行对应着不同的脚本文件 如make menuconfig 对应着是Kconfig 最后会被选择后进入到.configure 然后配合顶层makefile使用。

而我们这里说的是应用软件的编写。

所以做GNU的文件源码我们一定要会源码分类放，同时学会修改configure.in

和手动编写Makefile.am

下文将着重介绍Makefile.am 编写和configure.in 修改

例如我们随便建立三个文件 一个是main.c

一个是hello目录下main.c

#include <stdio.h>

int main(int argc,int \*argv)

{

int num[100];

int val=0;

int val2=0;

printf("argv[1]===%d \n",argv[0]);

printf("argv[2]===%d \n",argv[2]);

val=add(argv[1],argv[2]);

val2=lsl(argv[1],argv[2]);

printf("val=%d \n",val);

printf("val2=%d \n",val2);

return 0;

}

我们在add目录下建立add.c

#include <stdio.h>

int add(int i,int j)

{

int num=i+j;

printf("here is add.c:\n");

return num;

}

在 hello 下建立lsl

Lsl.c

#include <stdio.h>

int lsl(int i,int j)

{

int val=i\*j;

printf("here is lsl\n");

return val;

}

我们执行完autoscan后生成 configure.scan 和autoscan.log

其实configure.scan就是configure.in的模板

于是我们的configure.scan内容就是

# -\*- Autoconf -\*-

# Process this file with autoconf to produce a configure script.

AC\_PREREQ([2.63])

AC\_INIT([FULL-PACKAGE-NAME],[VERSION],[BUG-REPORT-ADDRESS])

AC\_CONFIG\_SRCDIR([main.c])

AC\_CONFIG\_HEADERS([config.h])

# Checks for programs.

AC\_PROG\_CC

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC\_OUTPUT

而我们手动修改成configure.in后为

# -\*- Autoconf -\*-

# Process this file with autoconf to produce a configure script.

AC\_PREREQ(2.61)

AC\_INIT(hello, 1.0, [])

AM\_INIT\_AUTOMAKE(hello, 1.0)

AC\_CONFIG\_SRCDIR([main.c])

AC\_CONFIG\_HEADER([config.h])

# Checks for programs.

AC\_PROG\_CC

AC\_PROG\_LIBTOOL

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC\_OUTPUT(Makefile add/Makefile lsl/Makefile)

通过对比我们发现其实我们只是更改了部分文件。

AC\_PREREQ(2.61)

指的是我们使用的内核版本号

当然这里的输入接口AC\_INIT(hello, 1.0, [])

分别指的是输入接口，源码包，源码包版本号，缺省

AM\_INIT\_AUTOMAKE(hello, 1.0) 这句话指的是我们要使用automake 所传入的宏定义。

里面参数顾名思义也是源码包和版本号。

AC\_CONFIG\_SRCDIR([main.c])

AC\_CONFIG\_HEADER([config.h])

这两句话一个指的是需要检查的目下下的文件，一个是config.h文件

如果confi.h文件不在，我们需要手动输入很多参数，如下：

Gcc-DPACKAGE\_NAME=\"hellobb\"-DPACKAGE\_TARNAME=\"hellobb\"-DPACKAGE-VERSION=\"1.0\" -DPACKAGE\_STRING=\"hellobb\ 1.0\" -DPACKAGE\_BUGREPORT=\"\" -DPACKAGE=\"hellobb\" -DVERSION=\"1.0\" -DSTDC\_HEADERS=1 \_DHAVE\_SYS\_TYPES\_H=1 -DHAVE\_UNISTDLIB\_H=1 -DHAVE\_STING\_H=1 -DHAVE\_MEMORY\_H=1 -dHAVE\_STRINGS\_H=1 -DHAVE\_INTTYPES\_H=1 -DHAVE\_STDINT\_H=1 -DHAVE\_UNISTD\_H=1 -DHAVE\_DLFCN\_H=1 -DLT\_OBJDIR=\"./libs/\" -I. -g -O2 -MT main.o -MD -MP -MF .deps/main.Tpo -c -o main.c

所以离开了config.h 是何等的麻烦。还有就是我们使用的

AC\_PROG\_CC

AC\_PROG\_LIBTOOL

一个指定的是CC 所替代的编译工具，工具的指定可以在configure处指定，也可是make 处指定

下一行这是指定使用LIBTOOL编译成库

最后的输出接口则是

AC\_OUTPUT(Makefile add/Makefile lsl/Makefile)

指定生成的makefile文件

以上是简单地更改过程。

然后是makefile.am 书写过程

AUTOMAKE\_OPTIONS = foreign

SUBDIRS = add lsl

bin\_PROGRAMS = main

main\_SOURCES = main.c

main\_LDADD = add/libadd.la lsl/liblsl.la

一个简单地makefile.am 顶层的书写

而各个底层也需要有自己的makefile.am

如下：

lib\_LTLIBRARIES = liblsl.la

liblsl\_la\_SOURCES = lsl.c

书写完成后，进过上述几步后你会发现我们看到GNU的文件基本都有了

然后就可以放心的执行make make install 等

如果想制作一个基于arm 平台的我们只需要在./configure 处加上交叉编译选项即可：

那么我们所需要更改的是

--host =arm-linux

指定软件运行的系统平台.如果没有指定,将会运行`config.guess'来检测

-build=BUILD

指定软件包安装的系统平台.如果没有指定,默认值将是'--host'选项的值.

　--host=HOST

指定软件运行的系统平台.如果没有指定,将会运行`config.guess'来检测.

-srcdir=DIR

这个选项对安装没有作用.他会告诉'configure'源码的位置.一般来说不用指定此选项,因为'configure'脚本一般和源码文件在同一个目录下.

--program-prefix=PREFIX

指定将被加到所安装程序的名字上的前缀.例如,使用'--program-prefix=g'来configure一个名为'tar'的程序将会使安装的程序被命名为'gtar'.当和其他的安装选项一起使用时,这个选项只有当他被`Makefile.in'文件使用时才会工作.

有时候你可能不想让你的软件包与系统已有的软件包交互.例如,你可能不想让你的新编译器使用GNU ld.通过使用这个选项可以做到这一点:

$ ./configure --without-gnu-ld