welcome

- 第一部分　介绍 Makefile
- 第二部分　介绍 autoconf 和 automake
- 第三部分　介绍 libtool
- 第四部分　讨论

—— 预研部　黄晶晶

makefile 简介

- make 工具被用来进行工程编译和程序链接，而 make 程序又是根据 Makefile 中的规则描述执行相关的命令来完成指定的任务的。所以我们了解 makefile 很有必要。那么 makefile 究竟做了什么呢？makefile 主要做两件事情：

  （1）查看目标文件是否存在。如果不存在就会在当前目录下生成目标文件；如果存在就会进行事件（2）。

  （2）判断目标文件是否过期。如果文件过期就会在当前目录或设置的路径下重建文件；如果没有过期将不进行任何改动。

- 在写 makefile 之前需要注意，一般使用 Makefile 或 makefile 作为其文件名
- 下面用一个简单的示例带您进入 makefile 的世界

```
Foo.o: foo.c defs.h
        g++ -c -g foo.c
```

makefile 规则介绍

- 目标：依赖 目标：依赖；命令

  【tab 键】命令 或 【tab 键】命令

  1. 目标文件：每一个规则的第一个目标为最终目标，目标可以是多个文件（它们以空格隔开）。特殊目标——伪目标：该目标不代表一个文件。它的特点是：不管依赖文件是否有更新，规则中的命令一定会执行。将一个目标声明为伪目标的方式为“ .PHONY ：伪目标名 ”

  2. 依赖文件：依赖类型分两种，常规依赖和 order-only 依赖。常规依赖规则中，如果依赖文件中的任何一个比目标文件新，目标文件都需要更新。而在“ order-only”依赖中，在依赖文件更新的情况下可不需要更新规则的目标，该文件仅仅在目标文件不存在的情况下才参与规则的执行。但这种依赖类型不常使用。

  3. 命令：命令行前都要先打 tab 键；目标之前的命令不会执行，目标之后的命令才会执行。命令可以和目标在同一行。通过命令对依赖文件进行处理，从而完成目标

makefile 规则之目录搜索

makefile 中执行命令的对象是各种文件，执行命令的环境是当前目录，而在大工程中文件并不都是存放在当前目录下的，当文件在其他目录下时，就需要我们进行目录搜索，这样我们才能顺利进行相关操作。

- 目录搜索：在 Makefile 中，使用依赖文件的目录搜索功能，当工程的目录结构发生变化后，就可以做到不更改 Makefile 的规则，而只需更改依赖文件的搜索目录。搜索使用的特性主要是：

  （1） VPATH （变量），指定 makefile 中所有文件的搜索路径。文件首先在当前目录下查找，如果没找到，就会到指定路径下搜索。例如对变量的定义如下：VPATH=src:../headers 指定了两个搜索路径“ src” 和 ” ../headers”

  （2） vpath （关键字），选择性搜索。它可为不同类型的文件指定不同的搜索路径。例如在某目录下查找某一类型的文件： vpath %.h src:../header 它表明 makefile 中 .h 文件首先在当前目录下查找，然后在 src 和 ../header 目录下查找；vpath %.h 表示清除之前为 makefile 中 .h 文件设置的搜索路径； vpath 表示清除所有设置的文件搜索路径

- 库文件也可通过目录搜索查找，不过，其查找路径略有不同：

  库文件搜索顺序：当前目录 ->VPATH 指定目录 -> 系统中库文件目录

Makefile 规则之自动化变量

通过前面目录搜索的介绍，我们知道搜索得到的依赖文件名，是其路径全名，而规则中使用的却只是其单纯的文件名，故为了命令的正确执行，我们需要让文件保持一致性，这里就要用到自动化变量。

- 常用的自动化变量有：

$@ 表示规则的目标文件、 $^ 代表所有通过目录搜索得到的依赖文件的完整路径列表、 $< 表示通过目录搜索得到的依赖文件列表的第一个文件、 $? 表示所有比目标文件新的依赖文件列表等等

makefile 规则之静态模式

当规则存在多个目标,并且不同的目标、依赖文件是相似时，可以根据目标文件的名字来自动构造出依赖文件。这样就给我们的工作带来了很多方便。

- 静态模式：需要多个目标具有相似而非相同的依赖。
- 下面我们来看一个小例子：

```
objects = foo.o bar.o                          all: foo.o bar.o
all: $(objects)                                foo.o: foo.c
$(objects): %.o: %.c          <=====>          $(CC) -c $(CFLAGS) $< -o $@
$(CC) -c $(CFLAGS) $< -o $@                    bar.o: bar.c
(注：%.o为目标模式,%.c为依赖模式)               $(CC) -c $(CFLAGS) $< -o $@
```

- $(objects)为目标,(这个规则是多目标)。目标需要与目标模式文件类型一致。
- makefile中存在隐含规则，例：对未指明构建规则的目标文件foo.o ，系统会自动查找并使用foo.c文件对其进行构建。由此看来，静态模式的功能显得多此一举，那么我们为什么还要使用静态模式呢？静态模式比之隐含规则有以下优势：

（1）不能根据文件名通过词法分析进行分类的文件,我们可以明确列出这些文件,并使用静态模式规则来重建其隐含规则

（2）存在多个适合此文件的隐含规则,使用哪一个隐含规则具有不确定因素；

makefile 规则之自动产生依赖

- 在 C++ 中文件通过使用 include 来查找依赖的头文件，在 makefile 中使用“ -M” 选项来查找 .o 文件与头文件的依赖关系。一个小例子，假设 main.c 中包含头文件 defs.h ：

  g++ -M main.cpp

  其输出是： main.o: main.cpp defs.h

- “-M” 选项包括对所有头文件的依赖，若不需要在依赖关系中考虑标准库头文件则使用“ -MM” 选项
- 下面是一个小例子：

  foo.o: foo.c defs.h

  g++ -M foo.c

makefile 中命令

- 规则中的命令由一些shell命令行组成，对规则中命令行的解析使用/bin/sh来完成

（1）命令回显。make执行时会把要执行的命令行输出到标准输出设备（即“回显”），在命令行前加字符@，该行就不会回显

（2）命令的执行。同一行的命令为一个完整的shell命令，一个shell需要多个命令时可以用 && 或者分号隔开,如果命令行很长可以使用反斜线，书写在多个物理行上；不同行的命令为不同shell进程，相互独立、互不依赖。一个小例子：

```
foo : bar/lose                                                    cd bar;\
        cd bar; gobble lose > ../foo     或者   cd bar&&gobble lose > ../foo 或者     gobble lose > ../foo
```

（3）make的递归执行。在 Makefile 中使用“make”作为一个命令来执行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。一个小例子：

```
subsystem:
        cd subdir && $(MAKE)   #$(MAKE)是对make变量的引用
```

（4）定义命令包。在多个规则使用相同的一组命令的情况下，我们想能不能将这样一组命令进行类似c语言中函数一样的封装,以后在使用时通过它的名字来对其进行引用。这样就可减少重复工作。我们使用define定义一组命令，以endef结束定义。命令包与c语言中宏的使用方式一样。一个小例子：

```
define frobnicate
        @echo "frobnicating target $@"
        frob-step-1 $< -o $@-step-1
        frob-step-2 $@-step-1 -o $@
endef
```

在规则中使用时：

```
frob.out: frob.in
        $(frobnicate)
```

即可

Makefile 中的变量

- 在 Makefile 中,变量是一个名字(像是 C 语言中的宏),代表一个文本字符串(变量的值),变量有以下几个特征：

  （1）读取 makefile 时展开变量，它包括用“=”定义的变量以及用“define”定义的命令包

  （2）文件列表、选项列表、目录列表都可以用变量来表示

  （3）定义的变量名仅包含数字、字母、下划线，因为其他字符串会有特殊含义，如自动化变量 $@。变量名对大小写敏感。

- 变量的引用：引用方式为”${}”或“$()”，一般使用“$()”。一个小例子：

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)
        $(objects) : defs.h
```

  变量引用的展开过程是严格的文本替换过程，就是说变量值的字符串被精确的展开在变量被引用的地方

-

Makefile 中的变量

- 赋值变量定义：存在两种变量定义：

  （1）递归展开式变量。用“ =” 定义的变量，也是我们常用的定义方式，这种类型的变量在定义时，可以引用在其后才定义的变量。例子：
  A=2 ， B=A ， A=4 ，则 B=4

  （2）直接展开式变量。用“ :=” 定义的变量，变量被定义后是一个实际需要的文本串，其中不再包含任何变量的引用。例子：
  A=2 ， B:=A ， A=4 ，则 B=2

- 变量的用法：常用“替换引用”。对于一个已经定义的变量，可以将其值中的后缀字符 (串) 使用指定的字符 (串) 替换。例如，

  ```
  foo := a.o b.o c.o o.o

  bar := $(foo:.o=.c)    # 或者  bar:=$(foo:%.o=%.c)
  ```

  bar 的值就是 a.c b.c c.c o.c

makefile 中的内嵌函数

- GNU make 的函数提供了处理文件名、变量、文本和命令的方法。使用函数我们的 Makefile 可以书写的更加灵活。函数调用格式为：$(FUNCTION ARGUMENTS)。常用的几种函数：

  (1)文本处理函数。GNU make中有多个内嵌的文本(字符串)处理函数，如:$(subst FROM,TO,TEXT)，函数功能为:把字串“TEXT”中的“FROM”字符替换为“TO”

  (2)文件名处理函数。对一系列空格分割的文件名进行转换,函数的参数为若干个文件名，对文件名进行处理。如：$(dir NAMES...)

  从文件名序列“NAMES...”中取出各个文件名的目录部分。如果文件名中没有斜线,认为此文件为当前目录(“./”)下的文件。示例:$(dir src/foo.c hacks)，返回值为“src/ ./”

  (3)其他函数。foreach、if、call、shell等等

执行 make

- 一般描述整个工程编译规则的 Makefile 最简单直接的是使用不带任何参数的“ make” 命令来重新编译所有过时的文件。然而，为了达到某些特殊的目的时，就需要使用 make 的命令行参数来实现了：

  (1)-f 参数。前面我们提到直接“ make” 命令时，系统会在当前目录下依次搜索名为 “ GNUmakefile ”、“ makefile” 和“ Makefile” 的文件；当使用参数选项“ make -f filename” 时就指定 make 的执行文件，即文件名可任意。

  (2) 指定终极目标。任何出现在 Makefile 规则中的目标都可以被指定为终极目标，然后通过命令“ make+ 终极目标”，我们就能完成对它的执行。例如

  ```
  foo: foo.o
      g++ -o foo foo.o
  ```

  ，命令 make foo 即可执行下面命令行

make 的其他特性

- Make 的隐含规则
- 使用 make 更新静态库文件：静态库文件也称为“文档文件”，它是一些 .o 文件的集合。 linux 中使用工具“ ar” 对它进行维护管理。例如：下边这个规则用于创建、更新库“ foolib” ：

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

- 下面为综合上述 make 特征的一个例子：

```
# 递归展开式变量
export A="Test "
CC=g++
test="Hello world"
```

```
#命令包，一次定义多次使用，类似C++中的宏定义
define comp
@echo "several tests as follows:"
endef

#目录环境变量
ROT=/home/hjj/test/testmake
SUBDIRS=$(ROT)/test1 $(ROT)/test2 $(ROT)/test3

#终极目标#make参数-s，在命令运行时不输出
#$$表示shell进程ID，$$sub动态文件名，避免写文件时其他使用者用文件
all:
$(comp)
@sub='$(SUBDIRS)';for subdir in $$sub; do\
 (cd $$subdir && make -s);\
        done
@echo "there are three test files in $(ROT)"
@echo "they are: $(SUBDIRS)"
```

autoconf&&automake

- 通过上面的介绍，我们可以看到，用 make 命令来编译自己写的程序确实是很方便。然而， Makefile 除了可以做到程序的编译和连接，也可以用来生成文档（如 manual page, info 文件等），还可以把源码文件包装起来以供发布，所以程序源代码所存放的目录结构以及 Makefile 文件最好符合 GNU 的标准惯例。
- automake 和 autoconf 就是自动产生标准惯例 makefile 的常用工具 。 接下来向大家介绍利用它们生成 Makefile 的过程。

autoconf&&automake

- 假设我们已经创建好源文件hello.cpp,接下来我们需要做的是：

（1） 生成configure.in

执行命令：$autoscan

$mv configure.scan configure.in

$vi configure.in

首先生成configure.scan文件，将其重命名为configure.in,然后编辑、修改里面的参数设置内容，注意修改三个设置，具体如下：

AC_CONFIG_SRCDIR([src/hello.cpp])

AM_INIT_AUTOMAKE

AC_OUTPUT(Makefile)　　//设置指明了配置的目的为生成Makefile

（2）生成configure

执行命令：$aclocal

$autoconf

首先生成aclocal.m4文件,然后根据configure.in生成为configure

configure是一个可移植的shell脚本，它检查编译环境以决定哪些库可用，所用平台有什么特征，哪些库和头文件已经找到等等。基于这些信息，它修改编译标记，生成Makefile文件

autoconf&&automake

（3） 生成 Makefile.in

执行命令： $vi Makefile.am

$automake --add-missing

新建 Makefile.am 文件，它主要是定义宏和目标，指定 makefile 的对象文件 filename.cpp 需要用到的特性，这里，我们以编译执行工程功能为例：

AUTOMAKE_OPTIONS=foreign

bin_PROGRAMS=filename

filename_SOURCES=filename.cpp

然后根据宏和目标 ,automake 生成指定的 Makefile.in 代码

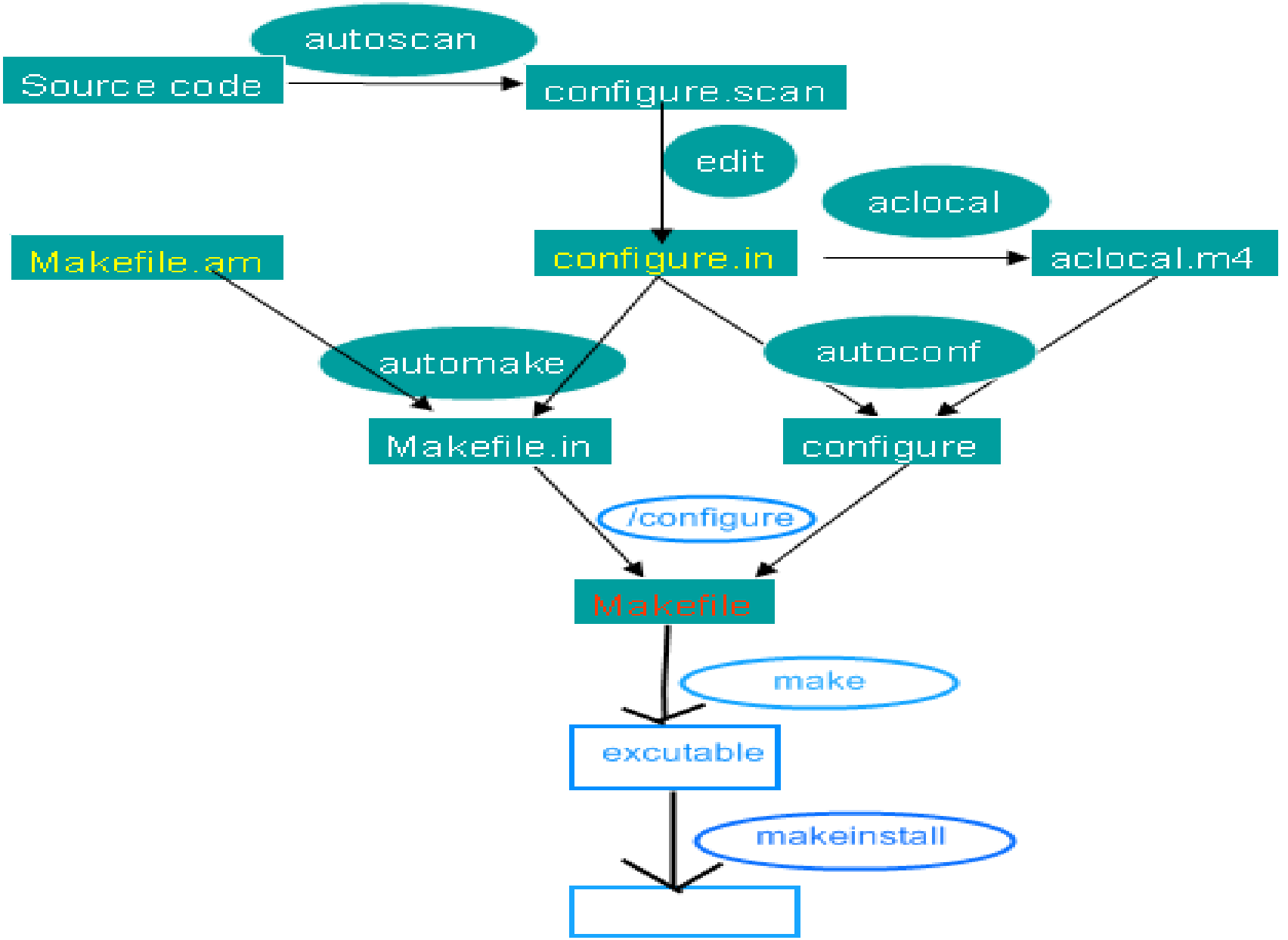
（4）生成 Makefile

执行命令： $./configure

根据生成的 configure 和 Makefile.in 生成为目标文件 Makefile

综上，我们可以看到 Makefile 生成的整个过程，列成图解如下所示：

autoconf&&automake



autoconf&&automake

现在已经生成Makefile文件，接下来我们就可以使用它了。

我们是由“配置文件+Makefile.in文件”生成的Makefile文件，所以只需"make","make install" 就可以把程序安装到系统中，其它操作及特性：

make clean 清除上次的make命令所产生的.o文件及可执行文件。

make install 将编译成功的可执行文件安装到系统目录/usr/local/bin中

make dist 产生发布软件包文件。它将会把可执行文件及相关文件打包成一个tar.gz压缩的文件用来作为发布软件的软件包。它会在当前目录下生成一个名字类似“PACKAGE-VERSION.tar.gz”的文件。PACKAGE和VERSION，是我们在configure.in中定义的AM_INIT_AUTOMAKE(PACKAGE, VERSION)。

make distcheck 生成发布软件包并对其进行测试检查，以确定发布包的正确性。这个操作将自动把压缩包文件解开，然后执行configure命令，并且执行make，来确认编译不出现错误，最后提示你软件包已经准备好，可以发布了。

make distclean 类似make clean，但同时也将configure生成的文件全部删除掉，包括Makefile

- 通过这些操作，我们还可以把在开发平台上编写的源程序和相关编译选项的配置打包，然后发布给用户

libtool

- 由于共享库在不同平台上可能有所不同，所以当我们把打包的配置文件解压后运行在其它平台上时，还需要 libtool 来确定共享库在该特定平台上的特性。 然后 configure 会检查该编译环境中哪些库可用，哪些库和头文件已找到，这样在 configure 和 libtool 的协同作用下完成不同用户平台对库文件的使用
- libtool 是一个通用库支持脚本，将使用动态库的复杂性隐藏在统一、可移植的接口中；使用 libtool 的标准方法，可以在不同平台上创建并调用动态库 (交叉编译)

libtool

- libtool包装了gcc或者其他的任何编译器，用户无需知道细节，只要告诉libtool需要编译哪些库即可，libtool将处理库的依赖等问题。

  接下来我将介绍libtool两个方面的内容：

  （1）使用libtools创建和使用安装动态库

  （2）库的依赖问题，合并依赖的静态库

- 首先介绍几种文件.a,.o,.lo,.la

  .a文件为静态库,是好多个.o合在一起,用于静态连接

  .o文件为目标文件,相当于windows中的.obj文件

  .so 为共享库,是shared object,用于动态连接的,和dll差不多

  .lo文件为libtool编译出来的目标文件也就是在.o文件中加一些信息

  .la文件为libtool编译出来的库文件,库中记录了其与其它库的依赖关系

- 静态连接库,在编译以后包含在可执行文件中,不会以单独文件的形式存在。
- 动态连接库是以单独文件的形式存在,被程序外部调用。

  可以多个进程访问一个动态连接库,并且共享一块内存；静态则包含在程序中,不能被外部调用

libtool 创建和安装动态库

- 假设已经存在源文件hello.c，那么接下来还需要以下操作：

  (1)由源文件编译生成lo文件（注意：不是.o文件，libtool只与.lo, .la文件打交道）

  执行命令：libtool --mode=compile gcc -g -O -c hello.c

  在这一步中，然后libtool作了3件事：

  1. 创建.libs
  2. 编译了一个与位置无关（-fPIC）的obj文件。 放在 .lib中。
  3. 编译了一个普通obj文件在本地。

  (2)由生成的lo文件，创建la文件

  执行命令：libtool --mode=link --tag=CC gcc -g -O -o libhello.la -rpath /usr/local/lib hello.lo

  这一步中，利用前面生成的hello.lo链接产生la文件，这里链接出静态和动态两个库

  (3)安装la文件

  执行命令：sudo libtool --mode=install cp libhello.la /usr/local/lib/libhello.la
  （将la文件复制到默认安装文件夹usr中，再安装）

libtool 创建和安装动态库

(4)编译生成可执行文件

执行指令：libtool --mode=link --tag=CC gcc hello.lo -o hello

(5)链接执行文件

执行命令： libtool --mode=link gcc -g -O -o hello hello.lo -rpath /usr/local/lib libhello.la

这一步中将当前目录下的执行文件和目标文件与usr文件夹中的la文件链接，可用指令ldd .libs/hello查看链接情况

(6)安装执行文件

执行命令：sudo libtool --mode=install cp hello /usr/local/bin/hello

(7)运行文件

执行命令：libtool --mode=execute hello 亦可直接输入 hello

- 通过上述操作，我们在/usr/local/lib中放入了libhello.la库文件，这样我们就可以在其它源文件中使用hello.h里定义的函数了，即在编译命令后加-lhello，例如假设在testhello.cpp文件下使用了hello中的功能函数，我们可以使用下面命令：

  g++ -g -o testhello testhello.cpp -lhello

libtool 创建和安装动态库

(4)编译生成可执行文件

执行指令：libtool --mode=link --tag=CC gcc hello.lo -o hello

(5)链接执行文件

执行命令： libtool --mode=link gcc -g -O -o hello hello.lo -rpath /usr/local/lib libhello.la

这一步中将当前目录下的执行文件和目标文件与usr文件夹中的la文件链接，可用指令ldd .libs/hello查看链接情况

(6)安装执行文件

执行命令：sudo libtool --mode=install cp hello /usr/local/bin/hello

(7)运行文件

执行命令：libtool --mode=execute hello 亦可直接输入 hello

- 通过上述操作，我们在/usr/local/lib中放入了libhello.la库文件，这样我们就可以在其它源文件中使用hello.h里定义的函数了，即在编译命令后加-lhello，例如假设在testhello.cpp文件下使用了hello中的功能函数，我们可以使用下面命令：

  g++ -g -o testhello testhello.cpp -lhello

libtool 库的依赖问题

- 存在这种情况：libtop.a依赖libsub1.a和libsub2.a，而libsub1.a和libsub2.a同时依赖libcom.a。有时我们希望使用libsub1.a和libsub2.a的用户不需要知道libcom.a，也就是在链接时不需要给出libcom.a；使用libtop.a的用户无需知道libsub1.a和libsub2.a，更无需知道libcom.a。那么怎样做，既能使用那个库又可隐藏细节呢？
- 针对这一问题，主要的解决思想是：解出需要隐藏的.a文件中的.o文件，然后将其加入到需要使用的.a文件中。
- 前面我们讲到，libtool中主要是与.lo和.la文件打交道的。那么在libtool中是怎么做到合并静态库的呢？下面以libcom.a为例：

  libtool先将libcom.a使用libtool编译成convenience library

  为了达到这一目的，需要将：

  lib_LTLIBRARIES = libcom.la　　noinst_LTLIBRARIES=libcom.la

  libcom_la_SOURCES = name.cpp　改成　libcom_la_SOURCES=name.cpp

  libsub1_la_LDFLAGS=libcom.la

  ‘noinst’说明生成的最终文件为静态库，libtool会到/usr/local/lib去寻找libname.la，然后从中读取实际的共享库的名字（libname.a）和路径，返回诸如/usr/local/lib/libname.a的参数给激发出 的gcc命令行

  这样libsub1.a里就包含了libcom.a里的.o文件；同理，需要在libtop.a中隐藏libsub1.a和libsub2.a也可以这么做。

  注意：使用_la_LDADD也可以达到上述目的，但是它会因不能避免.o文件的重复载入而出错。

libtool

- 通过上述操作，我们可以在需要使用 libtop.a 时，直接使用下面命令：　　g++ -g -o test test.cpp -ltop

  而不需要知道 libtop 文件中又依赖哪些库文件，然后进行链接：

  　　g++ -g -o test test.cpp -ltop -lsub1 -lsub2 -lcom

  这样在隐藏细节的同时，也给我们的书写带来方便

libtool

- 然而对于 libtool ，其实我们通常会将其与 autoconf 以及 automake 一起使用，像前面的那些命令行步骤也会自动生成在 makefile 里面，这样我们的工作就会方便很多。
- 那么，这三个工具是怎么结合起来使用的呢？

  答案在于 configure.in,Makefile.am 这几个文件的参数宏设置里面：

  (1) 在 configure.in 中添加库函数设置 AC_PROG_LIBTOOL;

  (2) 在 Makefile.am 中修改生成最终目标文件 lib_LTLIBRARIES=libname.la;

  添加增添库设置 name_LDADD=../lib/libname.la

  这样，我们设置好参数后，直接执行 ./configure, make, make install 即可安装好动态库文件和可执行文件。需要注意的是，由于默认情况下，编译器只会使用 /lib 和 /usr/lib 这两个目录下的库文件，而通过源码包进行安装时，一般将库安装在 /usr/local 目录下，所以我们需要手工载入库位置。
- Shell 搜索动态库路径的主要方法为：

  (1) 直接执行命令： export LD_LIBRARIES_PATH=dir(通常是 /usr/local/lib)

  (2) 在 /etc/ld.so.conf 文件中添加自动查找路径 /usr/local/lib /usr/lib /lib ，然后执行命令 ldconfig 更新设置。

结束语

- 很多内容讲解不充分，有些地方或许也有问题，欢迎大家积极指出，我们共同探讨互相学习。
- Thanks ！