第四章 UNIX下的C语言开发环境

冯 晶 E-mail: gfeng@whu.edu.cn

1. gcc 与 gdb

- 1.1 UNIX和C语言
  - C 是一种在 UNIX 操作系统的早期就被广泛使用的通用编程语言，它最早是由贝尔实验室的 Dennis Ritchie 为了 UNIX 的辅助开发而写的。
  - C 是所有版本的UNIX上的系统语言。
  - 几乎任何一种计算机上都有至少一种能用的 C 编译器；并且它的语法和函数库在不同的平台上都是统一的。
  - 80年代末期美国国家标准协会 (American National Standards Institute)发布了一个被称为 ANSI C 的 C 语言标准，这保证了在不同平台上的C的一致性。

1. gcc 与 gdb

- 1.2 GNU C编译器
  - GNU C 编译器(gcc)是一个全功能的 ANSI C 兼容编译器，它是所有UNIX系统可用的C编译器。
  - gcc是可以在多种硬体平台上编译出可执行程序的超级编译器，其执行效率与一般的编译器相比平均效率要高20%~30%。

1. gcc 与 gdb

- gcc编译过程：
  - 预处理，对源代码文件中的文件包含(include)、预编译语句(如宏定义define等)进行分析。
  - 编译，就是把C/C++代码“翻译”成汇编代码。
  - 汇编，将第二步输出的汇编代码翻译成符合一定格式的机器代码，生成以.o为后缀的目标文件。
  - 链接，将上步生成的目标文件和系统库的目标文件和库文件链接起来，最终生成了可以在特定平台运行的可执行文件。

1. UNIX下的C语言开发环境

- gcc遵循的文件类型规定
  - .c为后缀的文件，C语言源代码文件；
  - .a为后缀的文件，是由目标文件构成的档案库文件；
  - .C，.cc或.cxx 为后缀的文件，是C++源代码文件；
  - .h为后缀的文件，是程序所包含的头文件；
  - .i 为后缀的文件，是已经预处理过的C源代码文件；
  - .ii为后缀的文件，是已经预处理过的C++源代码文件；
  - .m为后缀的文件，是Objective-C源代码文件；
  - .o为后缀的文件，是编译后的目标文件；
  - .s为后缀的文件，是汇编语言源代码文件；
  - .S为后缀的文件，是经过预编译的汇编语言源代码文件。

1. gcc 与 gdb

- 1.3 gcc的使用
  - gcc最基本的用法：

    gcc [options] [filenames]

    options就是编译器所需要的参数；

    filenames给出相关的文件名称；

例：

```
$ gcc helloworld.c
$ ./a.out
$ Hello, World!
```

```
/* helloworld.c */
#include <stdio.h>

int main(int argc, char **argv){
    printf("Hello, World!\n");
    return 0 ;
}
```

1. gcc 与 gdb

- 1.4 gcc选项

  -Wall：允许所有有用的警告；

  -o：定义输出文件；

  -c：只编译生成.o目标文件，不链接；

  -I：设置头文件的搜索路径；

  -L：设置库文件的搜索路径；

  -O：O1，O2，O3，O4，O5优化级别；

  -g：gdb调试用，在可执行程序中包含标准调试信息；

  -w：关闭所有警告；

  -E：完成预处理/预编译停止；

1. gcc 与 gdb

-Wall：允许所有有用的警告（建议总是使用该选项）

```
/* bad.c */
#include <stdio.h>

int main(int argc, char **argv){
    printf("Two plus two is %f\n", 4);
    return 0 ;
}
```

例1：

$ gcc bad.c –o bad

```
[root@localhost jing]# ./bad
Two plus Two is -0.106536
[root@localhost jing]# ./bad
Two plus Two is -0.074180
```

例2：

$ gcc –Wall bad.c –o bad

```
[root@localhost jing]# gcc -Wall bad.c -o bad
bad.c: In function 'main':
bad.c:3: warning: format '%f' expects type 'double', but argument 2 has type 'in
t'
```

1. gcc 与 gdb

-o：定义输出文件

例：编译多个源文件

$ gcc -Wall main.c hello.c -o helloworld

```
/* hello.h */
void hello (const char * name);
```

```
/* hello.c */
#include <stdio.h>
#include "hello.h"
void hello (const char * name){
    printf ("Hello, %s!\n",  name);
}
```

```
/* main.c */
#include "hello.h"
int  main (int argc, char **argv){
    hello ("world");
    return 0;
}
```

1. gcc 与 gdb

-c：只编译生成.o目标文件，不链接

例：编译多个源文件

```
$ gcc -Wall -c main.c
$ gcc -Wall -c hello.c
$ gcc main.o hello.o -o helloworld
```

包含函数定义的对象文件应当出现在调用这些函数的任何文件之后

```
/* hello.h */
void hello (const char * name);
```

```
/* hello.c */
#include <stdio.h>
#include "hello.h"
void hello (const char * name){
    printf ("Hello, %s!\n",  name);
}
```

```
/* main.c */
#include "hello.h"
int  main (int argc, char **argv){
    hello ("world");
    return 0;
}
```

1. gcc 与 gdb

重新编译和重新链接

例：编译多个源文件

```
$ gcc -Wall -c main.c
$ gcc main.o hello.o -o helloworld
```

```
/* hello.h */
void hello (const char * name);
```

```
/* hello.c */
#include <stdio.h>
#include "hello.h"
void hello (const char * name){
    printf ("Hello, %s!\n", name);
}
```

World -> Everyone

```
/* main.c */
#include "hello.h"
int  main (int argc, char **argv){
    hello ("Everyone");
    return 0;
}
```

1. gcc 与 gdb

-l：链接外部库文件

- 库是已经编译好并能被链接入程序的对象文件的集合。库中提供一些最常用的系统函数，比如象C的数学库中求平方根函数sqrt。
- 库通常被存储在扩展名为“.a”或“.so”的特殊归档文件中。
- C标准库自身存放在“/usr/libc.a”中，包含ANSI/ISO C标准指定的各个函数，是默认自动加载的库。

例：

```
$ gcc -Wall sqrt.c -o sqrt
```

```
ccbR6Ojm.o: In function ‘main’:
ccbR6Ojm.o(.text+0x19): undefined
    reference to ‘sqrt’
```

```
/* sqrt.c */
#include <stdio.h>
# include <math.h>
int main(int argc, char **argv){
    double r = sqrt(3.0);
    printf ("("The square root of 3.0 is %f\n", r);
    return 0;
}
```

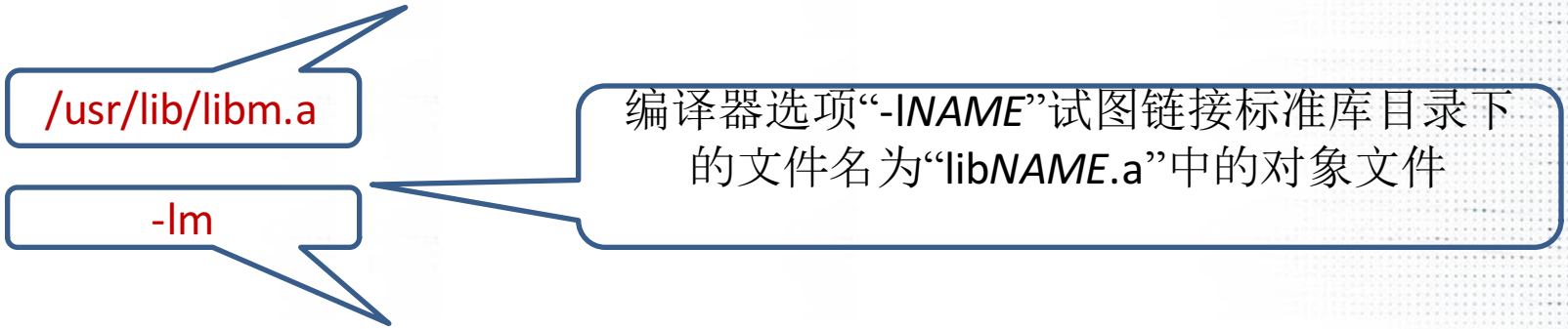
1. gcc 与 gdb

-l：链接外部库文件

- 函数sqrt()并不定义在源程序中或默认的C库“libc.a”中。
- 为了使得编译器能把sqrt()函数链接到主程序“sqrt.c”，需要提供“libm.a”库。

例：

```
$ gcc -Wall sqrt.c /usr/lib/libm.a -o sqrt
```




```
$ gcc -Wall sqrt.c -lm -o sqrt
```

1. gcc 与 gdb

-L：设置库文件的搜索路径

- 如果链接时用到的库不在gcc用到的标准库目录中，就会报这样的错。

```
[root@localhost UnixProgramming]# gcc -Wall sqrt.c -lm -o sqrt
/usr/bin/ld: cannot find -lm
collect2: ld returned 1 exit status
```

/usr/local/lib/
/usr/lib/

例：

$ gcc -Wall -L/tmp/lib sqrt.c -lm -o sqrt

-L/tmp/lib

1. gcc 与 gdb

-I：设置头文件的搜索路径

- 如果头文件不在gcc用到的标准include文件路径中，就会报这样的错。

```
[root@localhost UnixProgramming]# gcc -Wall sqrt.c -lm -o sqrt
sqrt.c:3:18: error: math.h: No such file or directory
sqrt.c: In function ‘main’:
sqrt.c:6: warning: implicit declaration of function ‘sqrt’
sqrt.c:6: warning: incompatible implicit declaration of built-in function ‘sqrt’
```

/usr/local/include/
/usr/include/

例：

$ gcc -Wall -I/tmp/include sqrt.c -lm -o sqrt

-I/tmp/include

1. gcc 与 gdb

库的链接次序

- **原则：**包含函数定义的库应该出现在任何使用到该函数的源文件和对象文件之后

例1：

```
$ gcc -Wall -lm sqrt.c -o sqrt          (incorrect)
$ gcc -Wall sqrt.c -lm -o sqrt          (correct)
```

- 程序“sqrt.c”用到了GNU Linear Programming库“libglpk.a”，而该库又依次用到数学库“libm.a”，那么应当这么编译：

例2：

```
$ gcc -Wall sqrt.c -lglpk -lm -o sqrt
```

1. gcc 与 gdb

-O：O0，O1，O2，O3，O4，O5，Os优化级别

- O1～O5选项，数字越大优化级别越高；
- Os选项选择缩减可执行文件大小的优化。它的目的是为内存和磁盘空间受限的系统生成尽可能小的可执行文件；
- 绝大部分目的而言，调试时用“-O0”，开发和部署时用“-O2”就足够了。

例：

```
$ gcc -Wall -O3 sqrt.c -lm -o sqrt
```

1. gcc 与 gdb

-g：存储调试信息

- 通常，可执行文件并不包含原来程序中源代码的任何引用信息，比如象变量名或行号----可执行文件只是编译器生成的作为机器码的指令序列。
- gcc提供了“-g”调试选项来在对象文件和可执行文件中存储另外的调试信息。这些调试信息可以使得在追踪错误时能从特定的机器码指令对应到源代码文件中的行。

1. gcc 与 gdb

- 1.5 gdb调试和分析选项
  - gdb 基本命令
    - file 装入想要调试的可执行文件；
    - kill 终止正在调试的程序；
    - list 列出产生执行文件的源代码的一部分；
    - next 执行一行源代码但不进入函数内部；
    - step 执行一行源代码而且进入函数内部；
    - run 执行当前被调试的程序；
    - quit 终止 gdb ；
    - watch 使你能监视一个变量的值而不管它何时被改变；
    - break 在代码里设置断点, 这将使程序执行到这里时被挂起；
    - make 使你能不退出 gdb 就可以重新产生可执行文件；
    - shell 使你能不离开 gdb 就执行 UNIX shell 命令.

1. gcc 与 gdb

- gdb调试举例

```
/* gdbtest.c */
#include <stdio.h>
int sum(int m);
int main(int argc, char **argv){
    int i, n = 0;
    sum(50);
    for(i=1; i<=50; i++){
            n += i;
    }
    printf("The sum of 1-50 is %d\n", n);
}
```

```
int sum(int m){
    int i, n = 0;
    for(i=1; i<=m; i++){
            n += i;
    }
    printf("The sum of 1-%d is %d\n", m, n);
}
```

例：

$ gcc -Wall -g gdbtest.c -o gdbtest

1. gcc 与 gdb

1. 启动gdb开始调试

例1：

$ gdb gdbtest

```
[root@localhost UnixProgramming]# gdb gdbtest
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) _
```

例2：

$ gdb

(gdb) file gdbtest

1. gcc 与 gdb

2. 在gdb中查看源代码

例：

(gdb) list

list也可以缩写为l

```
(gdb) list
1       /* gdbtest.c */
2       #include <stdio.h>
3
4       int sum(int m);
5
6       int main(int argc, char **argv){
7               int i, n = 0;
8               sum(50);
9               for(i=1; i<=50; i++){
10                      n += i;
(gdb) _
```

3. 在gdb中设置断点

例1：

(gdb) break 9

break也可以缩写为b

```
(gdb) break 9
Breakpoint 1 at 0x80483e8: file gdbtest.c, line 9.
```

- 注意：在gdb中利用行号设置断点，是指代码运行到对应行号之前暂停；
- gdb中可以设置多个断点。

3. 在gdb中设置断点

例2：

(gdb) break sum

设置函数断点，在函数体开始处

```
(gdb) break sum
Breakpoint 2 at 0x8048423: file gdbtest.c, line 16.
```

例3：

(gdb) break 10 if i==10

设置条件断点

```
(gdb) break 10 if i==10
Breakpoint 3 at 0x80483f1: file gdbtest.c, line 10.
```

1. gcc 与 gdb

4. 开看断点情况

例：

(gdb) info break

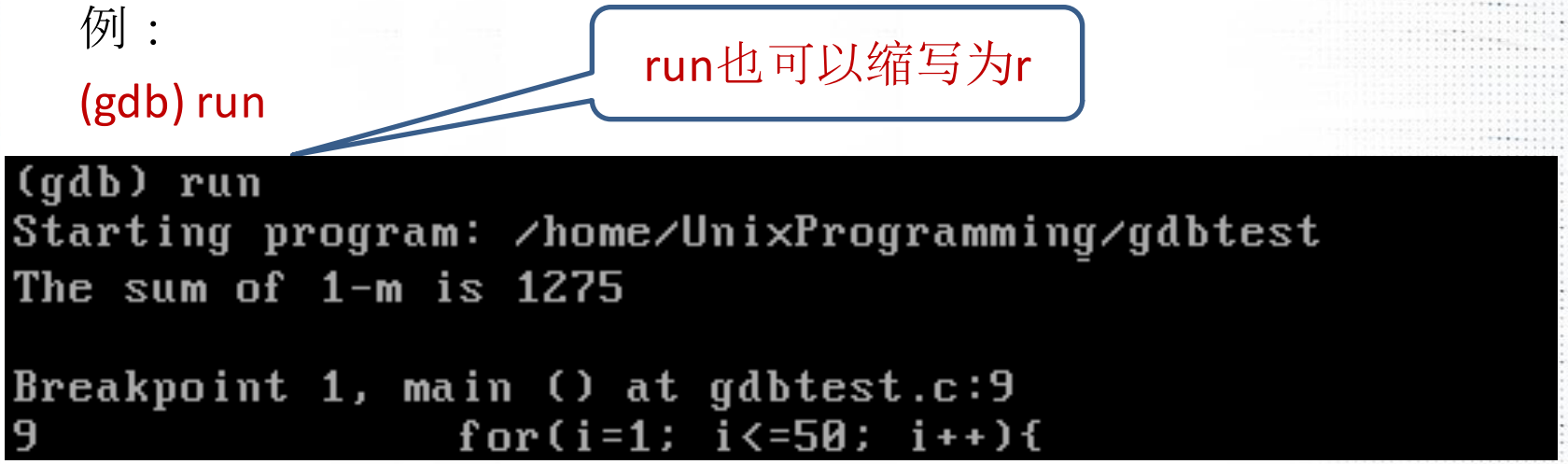
```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483e8 in main at gdbtest.c:9
2   breakpoint     keep y   0x08048423 in sum at gdbtest.c:16
3   breakpoint     keep y   0x080483f1 in main at gdbtest.c:10
        stop only if i == 10
(gdb) _
```

1. gcc 与 gdb

5. 运行代码

例：

(gdb) run



6. 查看变量

例：

(gdb) print i

print也可以缩写为p；i为变量名

```
(gdb) print i
$1 = 11358196
(gdb) print n
$2 = 0
(gdb)
```

“$N”是当前变量值的引用标记

7. 单步运行

- 单步运行可以使用命令“step”和“next”，它们之间的区别在于：若有函数调用的时候“step”会进入函数体，而“next”不会进入该函数；

8. 恢复程序运行

- 使用命令“continue”可以恢复程序正常运行，直到遇到下一个断点或者到程序结束；
- gdb中程序有“运行”、“暂停”和“停止”三种状态。“暂停”状态时，函数的地址、参数、局部变量都被压入“栈”中，故可以查看函数的各种属性；函数处于“停止”状态时，“栈”会自动撤销，也就无法查看函数的信息了。

2. make工程管理器

- 2.1 为什么要使用make？
  - 工作量问题：对于拥有多个（上百个）源文件的软件项目，只需编写一次编译过程，而不需要在每次源文件修改后重复输入众多的文件名和编译命令进行编译；
  - 效率问题：make能够根据文件的时间戳自动发现更新过的源文件，并通过读入Makefile文件来对更新的源文件进行编译而对其它文件只进行链接操作。

2. make命令

- 2.2 Makefile的格式
  - Makefile是make读入的唯一配置文件，Makefile文件通常包含如下内容：
    - 需要由make创建的目标体(target)，通常是目标文件或可执行文件；
    - 要创建的目标体所依赖的文件(dependency_file)；
    - 创建每个目标体时需要运行的命令(commmand)。

Makefile格式：

target: dependency_file

command

command之前必须有个“tab”

2. make命令

- 2.2 Makefile的格式

Makefile文件内容：

```
main: main.o hello.o
        gcc main.o hello.o -o main
main.o: main.c
        gcc -Wall -c main.c -o main.o
hello.o: hello.c hello.h
        gcc -Wall -c hello.c -o hello.o
```

```
[root@localhost UnixProgramming]# make main
gcc -Wall -c main.c -o main.o
gcc -Wall -c hello.c -o hello.o
gcc main.o hello.o -o main
```

```
/* hello.h */
void hello (const char * name);

/* hello.c */
#include <stdio.h>
#include "hello.h"
void hello (const char * name){
    printf ("Hello, %s!\n", name);
}

/* main.c */
#include "hello.h"
int main (int argc, char **argv){
    hello ("world");
    return 0;
}
```

2. make命令

- 2.3 Makefile中变量的使用

变量的定义：　　变量的使用：

OBJS = val　　$(OBJS)

Makefile文件内容：

```
main: main.o hello.o
        gcc main.o hello.o -o main
main.o: main.c
        gcc -Wall -c main.c -o main.o
hello.o: hello.c hello.h
        gcc -Wall -c hello.c -o hello.o
```

新Makefile文件内容：

```
OBJS = main.o hello.o
CC = gcc -Wall -c
main: $(OBJS)
        gcc $(OBJS) -o main
main.o: main.c
        $(CC) main.c -o main.o
hello.o: hello.c hello.h
        $(CC) hello.c -o hello.o
```

2. make命令

- 2.4 Makefile中常见的自动变量

$@	目标文件的完整名称；
$^	所有不重复的依赖文件，以空格分开；
$<	第一个依赖文件；
$*	不包含扩展名的目标文件名称；
$+	所有依赖文件，以出现的先后为序，包括重复文件；
$?	所有时间戳比目标文件旧的依赖文件；

2. make命令

- 2.4 Makefile中常见的自动变量

Makefile文件内容：

```
OBJS = main.o hello.o
CC = gcc -Wall -c
main: $(OBJS)
        gcc $(OBJS) -o main
main.o: main.c
        $(CC) main.c -o main.o
hello.o: hello.c hello.h
        $(CC) hello.c -o hello.o
```

新Makefile文件内容：

```
OBJS = main.o hello.o
CC = gcc -Wall -c
main: $(OBJS)
        gcc $^ -o $@
main.o: main.c
        $(CC) $^ -o $@
hello.o: hello.c hello.h
        $(CC) $< -o $@
```

2. make命令

- 2.5 make管理器的使用

  – 在make命令后面键入目标名即可建立指定的目标；如果不跟目标名则建立Makefile中定义的第一个目标。

  -C dir make管理器的使用；

  -f file 读入当前目录下的file文件作为Makefile；

2. make命令

- 2.5 使用autotools工具
  - autotools工具可以为源代码自动产生Makefile文件；
  - autotools工具主要包括一下命令：
    - autoscan
    - aclocal
    - autoconf
    - autoheader
    - automake

2. make命令

- 2.5 使用autotools工具

Step1. 在源代码目录下运行autoscan ；

```
[root@localhost main]# ls
hello.c  hello.h  main.c
[root@localhost main]# autoscan
[root@localhost main]# ls
autoscan.log  configure.scan  hello.c  hello.h  main.c
```

Step2. 重命名configure.scan为configure.in

mv configure.scan configure.in

2. make命令

- 2.5 使用autotools工具

Step3. 编辑configure.in文件；

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

AC_PREREQ(2.61)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

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

# Checks for library functions.

AC_OUTPUT
```

2. make命令

- 2.5 使用autotools工具

- AC_PREREQ宏声明本文件要求的autoconf版本；
- AC_INIT宏用来定义软件的名称和版本等信息，BUG_REPORT_ADDRESS可以省略；
- AC_CONFIG_SRCDIR宏用来侦测所指定的源代码文件是否存在，来确定源码目录的有效性；
- AC_CONFIG_HEADER宏用于生成config.h文件，以便autoheader使用；

- 修改AC_INIT(软件名称，版本信息)；
- 增加或修改AM_INIT_AUTOMAKE(软件名称，版本信息)；
- 增加或修改AC_CONFIG_FILES([Makefile])；

2. make命令

- 2.5 使用autotools工具

Step4. 运行aclocal命令；

Step5. 运行autoconf命令；

Step6. 运行autoheader命令；

```
[root@localhost main]# aclocal
[root@localhost main]# autoconf
[root@localhost main]# autoheader
[root@localhost main]# ls
aclocal.m4        autoscan.log  configure      hello.c  main.c
autom4te.cache    config.h.in   configure.in   hello.h
```

2. make命令

- 2.5 使用autotools工具

Step7. 新建Makefile.am文件；

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS=main
main_SOURCES=main.c hello.c hello.h
```

Step8. 运行automake --add-missing；

```
[root@localhost main]# automake --add-missing
configure.in:7: installing `./missing'
configure.in:7: installing `./install-sh'
Makefile.am: installing `./depcomp'
```

2. make命令

- 2.5 使用autotools工具

Step8. 运行./configure 命令；

```
[root@localhost main]# ls
aclocal.m4        config.h.in    configure.in  install-sh   Makefile.in
autom4te.cache    config.log     depcomp       main.c       missing
autoscan.log      config.status  hello.c       Makefile     stamp-h1
config.h          configure      hello.h       Makefile.am
```

Makefile文件已生成

2. make命令

- 2.6 autotools生成Makefile的编译、安装
  - make

```
[root@localhost main]# make
make  all-am
make[1]: Entering directory `/home/UnixProgramming/main'
gcc -DHAVE_CONFIG_H -I.     -g -O2 -MT main.o -MD -MP -MF .deps/main.Tpo -c -o m
ain.o main.c
mv -f .deps/main.Tpo .deps/main.Po
gcc -DHAVE_CONFIG_H -I.     -g -O2 -MT hello.o -MD -MP -MF .deps/hello.Tpo -c -o
 hello.o hello.c
mv -f .deps/hello.Tpo .deps/hello.Po
gcc  -g -O2   -o main main.o hello.o
make[1]: Leaving directory `/home/UnixProgramming/main'
[root@localhost main]# ls
aclocal.m4      config.h.in    configure.in  hello.o     main.o       missing
autom4te.cache  config.log     depcomp       install-sh  Makefile     stamp-h1
autoscan.log    config.status  hello.c       main        Makefile.am
config.h        configure      hello.h       main.c      Makefile.in
```

2. make命令

- 2.6 autotools生成Makefile的编译、安装
  - make install

```
[root@localhost main]# make install
make[1]: Entering directory `/home/UnixProgramming/main'
test -z "/usr/local/bin" || /bin/mkdir -p "/usr/local/bin"
  /usr/bin/install -c 'main' '/usr/local/bin/main'
make[1]: Nothing to be done for `install-data-am'.
make[1]: Leaving directory `/home/UnixProgramming/main'
[root@localhost main]# which main
/usr/local/bin/main
```

已经加入到系统目录

2. make命令

- 2.6 autotools生成Makefile的编译、安装
  - make clean

```
[root@localhost main]# make clean
test -z "main" || rm -f main
rm -f *.o
[root@localhost main]# ls
aclocal.m4      config.h.in    configure.in  install-sh   Makefile.in
autom4te.cache  config.log     depcomp       main.c       missing
autoscan.log    config.status  hello.c       Makefile     stamp-h1
config.h        configure      hello.h       Makefile.am
```

删除先前编译的所有.o目标文件和可执行文件

2. make命令

- 2.6 autotools生成Makefile的编译、安装
  - make dist

将程序和相关文档打包，以便发布

```
[root@localhost main]# make dist
{ test ! -d main-1.0 || { find main-1.0 -type d ! -perm -200 -exec chmod u+w {}
';' && rm -fr main-1.0; }; }
test -d main-1.0 || mkdir main-1.0
find main-1.0 -type d ! -perm -777 -exec chmod a+rwx {} \; -o \
          ! -type d ! -perm -444 -links 1 -exec chmod a+r {} \; -o \
          ! -type d ! -perm -400 -exec chmod a+r {} \; -o \
          ! -type d ! -perm -444 -exec /bin/sh /home/UnixProgramming/main/instal
l-sh -c -m a+r {} {} \; \
        || chmod -R a+r main-1.0
tardir=main-1.0 && /bin/sh /home/UnixProgramming/main/missing --run tar chof - "
$tardir" | GZIP=--best gzip -c >main-1.0.tar.gz
{ test ! -d main-1.0 || { find main-1.0 -type d ! -perm -200 -exec chmod u+w {}
';' && rm -fr main-1.0; }; }
[root@localhost main]# ls
aclocal.m4      config.h.in    configure.in  install-sh       Makefile.am
autom4te.cache  config.log     depcomp       main-1.0.tar.gz  Makefile.in
autoscan.log    config.status  hello.c       main.c           missing
config.h        configure      hello.h       Makefile         stamp-h1
```