第五章　GCC编译器

第五章：GCC编译器

目标:
本章使学员熟练掌握linux操作系统下GCC编译器的使用，通过本课的学习，学员应该掌握如下知识：

- ☑ 了解GCC编译选项
- ☑ 掌握如何利用GCC编译程序

学时：4 学时

教学方法：讲授ppt＋上机操作+实例演示

5.4 gcc使用第三方库

- 在Linux下开发软件时，完全不使用第三方函数库的情况是比较少见的，通常来讲都需要借助一个或多个函数库的支持才能够完成相应的功能。
- 从程序员的角度看，函数库实际上就是一些头文件（.h）和库文件（.so或者.a）的集合。虽然Linux下的大多数函数都默认将头文件放到/usr/include/目录下，而库文件则放到/usr/lib/目录下，但并不是所有的情况都是这样。正因如此，GCC在编译时必须有自己的办法来查找所需要的头文件和库文件。
- GCC采用搜索目录的办法来查找所需要的文件，-I选项可以向GCC的头文件搜索路径中添加新的目录。例如，如果在/home/hxy/upgrade/include/目录下有编译时所需要的头文件，为了让GCC能够顺利地找到它们，就可以使用-I选项：
  - gcc foo.c -I /home/xiaowp/include -o foo
- 在一个gcc命令中可以用多个 -I

5.4.1 两大类库形式

- C/C++可以使用两种库.一种是静态库,另外一种是动态库.
  - 静态库在链接时会把库目标代码与最终的可执行程序一起链接到一个文件,这样相对尺寸较大.但处理简单.
  - 而动态库是可执行程序在运行,动态加载到进程内存中去.动态库与可执行程序是分离的两部分文件.
  - 两者作用完全等效,主要是使用方法不同.由开发者根据项目情况自行评估使用哪种形式.
- Windows下的静态库是以 lib为后缀名的文件,而动态库是以DLL为后缀名的文件.
  Linux下的动态链接库是so为后缀,和静态链接库以.a为后缀名

5.4.2 Linux库的命名

Neuedu

- linux库的命名有一个特殊的要求,即要以lib打头,以.so或.a结尾
  - libc.so　　#标准C库,动态链接库
  - libpthread.a #线程库,的静态链接库版本.
  - 在一般使用时,为防止不同版本库互相覆盖,一般还在系统库名后加入版本号.
    - libm. 6. so #math库 ver 6.0版本
    - libc-2.3.2.so #标准C ver 2.3.2动态库
  - linux一般把系统库放在 /lib下.
  - 这是大部分库命名的习惯,也可以不遵守,如果强行做一个叫mystd.a 的库,但使用起来很不方便,如不能使用-l参数等,所以建议不要这样做.

5.4.3 gcc链接库

- gcc是在链接时,把库加入程序中.
- 一个特殊静态链接库方式.把库完整名字加入
  - gcc -o hello　hello.o libmy.a
    - 链接hello.o,和库libmy.a到程序hello中（静态链接）
  - gcc -o hello　hello.o libmy.so
    - 链接hello.o,和库libmy.so到程序hello中,注意这里没有直接把libmy.so代码加入hello中（动态链接）
  - 这一方法主要用于链接不标准库名称,或混和链接(即一部分库用于静态版本,一部分库用动态版本).但不是正规用法,强烈建议**不要**使用这一方法

5.4.3 gcc链接库

Neuedu

- gcc -l参数 用来链接库标准表达式方式.
- -l接的库名,是去掉lib和后缀名(.so,.a)剩下的部分,
  - gcc foo.c -lpthread -o foo
    - 构造foo,链接库pthread . -lpthread 表示 链接 libpthread.so
- 去掉后缀名,gcc –l是如何知道链接是动态库还是静态库的?,gcc有如下规则:
  - 如果gcc所能找到库目录同时有两种版本,优先链接动态链接库版本
  - 如果gcc所能找到库目录只有静态版本,则采用静态版本
  - 如果加上 -static 参数,gcc 则强制链接静态版本
    - gcc foo.c -static -lpthread -o foo
      - -lpthread 表示 链接 libpthread.a

5.4.3 gcc链接库

- gcc所编译的目标文件和库通常不是在同一个目录下.因此必须强制指明gcc要从哪一个目录加载库
- gcc 在链接时采用 -L参数来指明从哪一个目录加载库
  - 例如，如果在/home/hxy/lib/目录下有链接时所需要的库文件libfoo.so
    - gcc foo.c -L/home/hxy/lib -lfoo -o foo
  - 一个gcc语句可以包含多个-L参数
  - 在编译时目标文件时使用-L无效
  - 标准库,gcc能自行找到,无需使用-L参数
- 在一些应用中,链接多个库是有顺序的,大部分无所谓
  - 如在系统中 libpthread.a 使用liba.a中的函数,而可执行程序同时使用两个库,则使用者liba.a的链接语句放在被使用者libpthread.a的前面,
    - gcc foo.c -L/home/hxy/lib –la -lpthread -o foo

一个使用线程库的例子

Neuedu

```
hxy@TecherHost:~/gcc
[hxy@TecherHost gcc]$ gcc test_thread.c -o test_thread -lpthread
[hxy@TecherHost gcc]$ file test_thread
test_thread: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/L
inux 2.2.5, dynamically linked (uses shared libs), not stripped
[hxy@TecherHost gcc]$ ls -l test_thread
-rwxr-xr-x    1 hxy      users       12036 Jan 30 23:29 test_thread
[hxy@TecherHost gcc]$ gcc test_thread.c -o test_thread -lpthread -static
[hxy@TecherHost gcc]$ ls -l test_thread
-rwxr-xr-x    1 hxy      users      479369 Jan 30 23:29 test_thread
[hxy@TecherHost gcc]$ file test_thread
test_thread: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/L
inux 2.2.5, statically linked, not stripped
[hxy@TecherHost gcc]$ ./test_thread
I am thread
I am thread
I am thread
I am thread
I am thread
[hxy@TecherHost gcc]$
```

5.5 gcc创建库

关于库的演示代码

- 在随后的演示中,将采用如下演示代码.
  - mystrlen.c,myshow.c 分别实现了统计字符串长度的函数,以及打印一个字符串的函数。
  - 这两个函数的声明在my.h中
  - main.c分别用静态链接,隐式动态链接来测试使用mystrlen.c ,myshow.c的函数
- 在随后的例子里把mystrlen.c ,myshow.c 分别编译成静态库 libstr.a,隐式动态库 libstr.so库名中不要出现大写字母,gcc 是按小写字母来查找的。

5.5.1 静态链接库

- 在LINUX下，静态函数库是以.a作后缀的 ,类似于Windows 的 .lib
- 在链接后,静态库的函数都会链接到最终的可执行程序里.这样可执行程序的尺寸比动态链接要大.
- 静态链接的好处是不需要外部文件的支持,独立运行.在嵌入式环境下,如果尺寸影响不大,最好用静态编译.

5.5.2 创建静态链接库

Neuedu

- gcc不能直接创建静态库.必须要用归档命令ar来创建
- ar用于建立、修改、提取档案文件(archive)。archive是一个包含多个被包含文件的单一文件（也称之为库文件），其结构保证了可以从中检索并得到原始的被包含文件（称之为archive中的member）。
- ar可以把任何文件归在一起,但通常是用来把gcc编译的目标文件(.o),合在一个静态库中
- 静态库创建
  - $ gcc -Wall -c file1.c file2.c file3.c #一次性编译三个.o
  - $ ar rv libname.a file1.o file2.o file3.o #把三个o合在一起

5.5.2 创建静态链接库

Neuedu

创建一个静态库的脚本

```
gcc -c mystrlen.c
gcc -c myshow.c

#把两个目标文件装入静态库libstr.a中
ar rv libstr.a mystrlen.o myshow.o
gcc -c main.c -I.

#gcc main.o libstr.a -o main_a
#非标准用法,用全名
gcc main.o -L. -lstr -o main_a #标准用法
```

5.5.3 创建动态链接库

Neuedu

- 动态链接库的创建分为两步:
  - 1.编译目标文件,必须带上-fpic 标志,使输出的对象模块是按照可重定位地址方式生成的。
    - gcc -c mystrlen.c -fpic
    - gcc -c myshow.c -fpic
  - 2.将加入动态库的目标文件合并在一起,必须带上-shared ,明确表示是动态链接库
    - gcc -shared mystrlen.o myshow.o -o libstr.so
- 两步可以合并成一步,但一般不建议这样做
  - gcc -fpic -shared mystrlen.c myshow.c -o libstr.so
- so是Shared Object 的缩写

5.5.3 创建动态链接库

Neuedu

创建一个动态库的脚本

```
#!/bin/bash
#gcc -fpic -shared mystrlen.c myshow.c -o libstr.so
gcc -c -fpic mystrlen.c
gcc -c -fpic myshow.c
#生成动态链接库
gcc -shared mystrlen.o myshow.o -o libstr.so
#编译测试程序
gcc -c main.c -I.
#链接主程序和动态库
#gcc main.o libstr.so -o main_so #非标准链接方式
gcc main.o -L. -lstr -o main_so
```

5.5.4 运行中使用动态链接库

Neuedu

- 一个使用动态链接库的程序运行时,要做一下设置.否则应用程序会报找不到动态库的错误
- 隐式调用和显式调用两种调用方法：
  - 隐式调用是不采用特殊系统调用,只是在gcc链接时采用-l,-L链接。这样对代码影响不大。
  - 显式调用是在代码中加入一些特殊代码, 表示要调用哪个动态库程序。具有灵活的特点,缺点就是必须使用特定的,不可移植的系统调用来编写。过程比较复杂。
- 一个程序运行后,可以用命令ldd来检查它使用了哪一些动态库
  - ldd ./hello

5.5.5 隐式调用动态库的方法

- 如果让程序运行时能找动态链接库,Linux有如下几种方法.
  - 把库所在路径加入/etc/ld.so.conf,程序加载时首先到这里路径查找
  - 设置环境变量LD_LIBRARY_PATH,把库所在路径加入这个变量中,这是最常用的方法
    - 演示代码将采用这一方法运行,写一个脚本run_so.sh

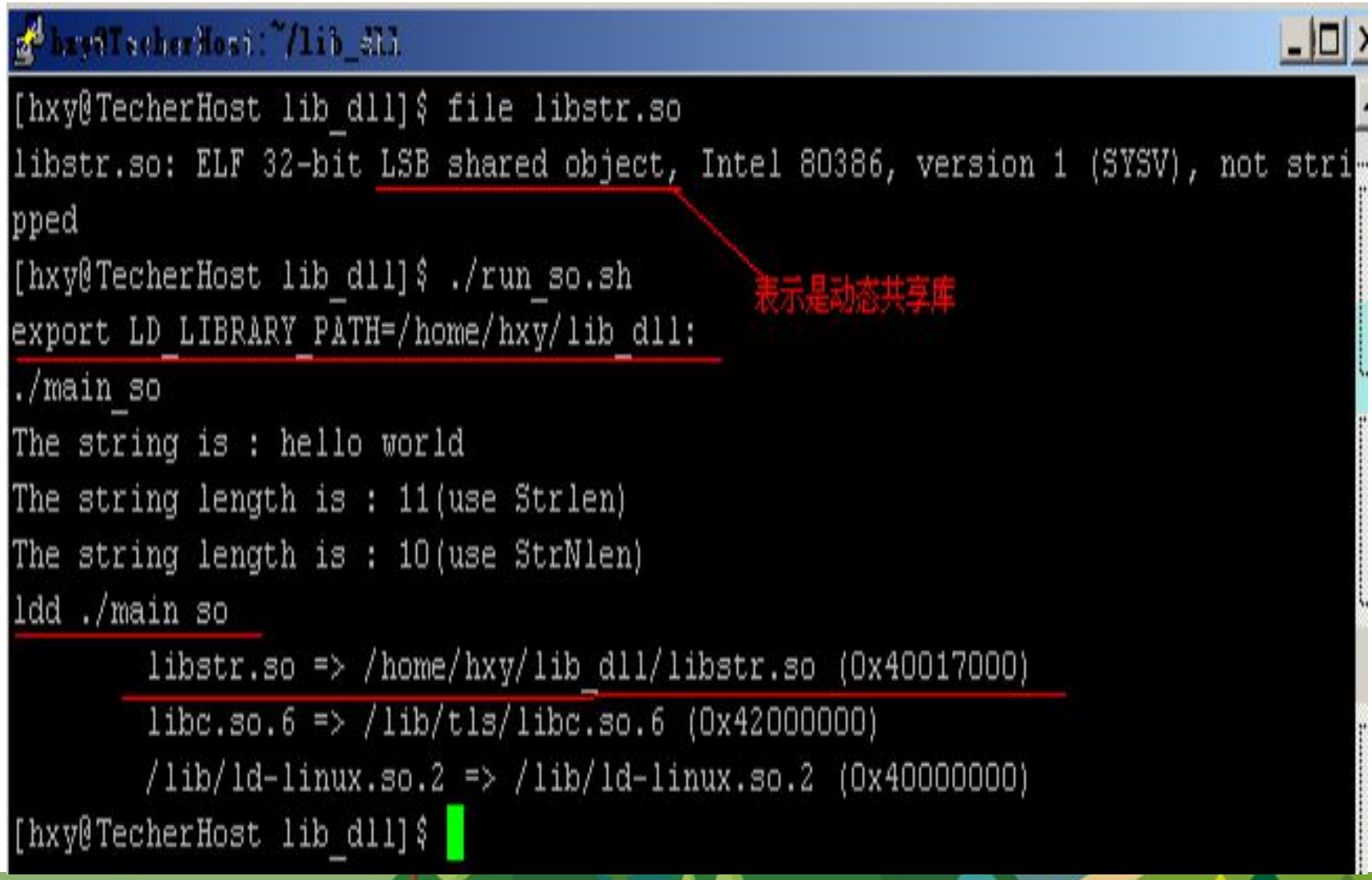
```
#!/bin/sh

export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
./main_so

ldd ./main_so #用ldd查看用了哪些动态库
exit
```

隐式动态库的执行结果

Neuedu



课堂练习

- 开发一个库函数,要求能对字符串的进行大写、小写转换.
- 要求把库函数,编译成静态和动态链接库各一个,取名libtu.a,libtu.so
- 编写一个测试程序,要求能测试libtu测试库函数

5.6 gcc代码优化

- 代码优化指的是编译器通过分析源代码，找出其中尚未达到最优的部分，然后对其重新进行组合，目的是改善程序的执行性能。GCC提供的代码优化功能非常强大，它通过编译选项-On来控制优化代码的生成，其中n是一个代表优化级别的整数。对于不同版本的GCC来讲，n的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从0变化到2或3。
- 编译时使用选项-O可以告诉GCC同时减小代码的长度和执行时间，其效果等价于-O1。通常来说，数字越大优化的等级越高，同时也就意味着程序的运行速度越快。许多Linux程序员都喜欢使用-O2选项，因为它在优化长度、编译时间和代码大小之间，取得了一个比较理想的平衡点。

5.6.1 gcc代码优化方法

- 不带优化
  - gcc -Wall optimize.c -o optimize
- 做了优化
  - gcc -Wall -O optimize.c -o optimize
- time
  - 借助Linux提供的time命令，可以大致统计出该程序在运行时所需要的时间,比较两次时间
  - # time ./optimize

代码优化实例

- 一个做了大量浮点数除法的程序.

```
 #include <stdio.h>
int main(void)
 {
     double counter;
     double result;
     double temp;
     for ( counter = 0; counter < 2000.0 * 2000.0 * 2000.0  /
20.0   + 2020;    counter += (5 - 1) / 4)
    {
          temp = counter / 1979;
          result  = counter;
     }
     printf("Result is %lf\n", result);
     return 0;
 }
```

代码优化的结果比较

Neuedu

```
[hxy@TecherHost gcc]$ gcc -Wall optimize.c -o optimize_0
[hxy@TecherHost gcc]$ gcc -Wall -O1 optimize.c -o optimize_1
[hxy@TecherHost gcc]$ gcc -Wall -O2 optimize.c -o optimize_2
[hxy@TecherHost gcc]$ gcc -Wall -O3 optimize.c -o optimize_3
[hxy@TecherHost gcc]$ time ./optimize_0
Result is 400002019.000000

real    0m6.149s
user    0m6.150s
sys     0m0.000s
[hxy@TecherHost gcc]$ time ./optimize_1
Result is 400002019.000000

real    0m1.925s
user    0m1.920s
sys     0m0.000s
[hxy@TecherHost gcc]$ time ./optimize_2
Result is 400002019.000000

real    0m1.927s
user    0m1.920s
sys     0m0.010s
[hxy@TecherHost gcc]$
[hxy@TecherHost gcc]$ time ./optimize_3
Result is 400002019.000000

real    0m1.946s
user    0m1.940s
sys     0m0.000s
```

5.6.2 gcc代码优化选项

- **-O1**
  - 多优化一些.除了涉及空间和速度交换的优化选项,执行几乎所有的优化工作.
- **-O2/O3**
  - 优化的更多.
- **-O0**
  - 基本不优化.
- 如果指定了多个**-O**选项,不管带不带数字,最后一个选项才是生效的选项.

5.6.3 避免gcc代码优化的场合

Neuedu

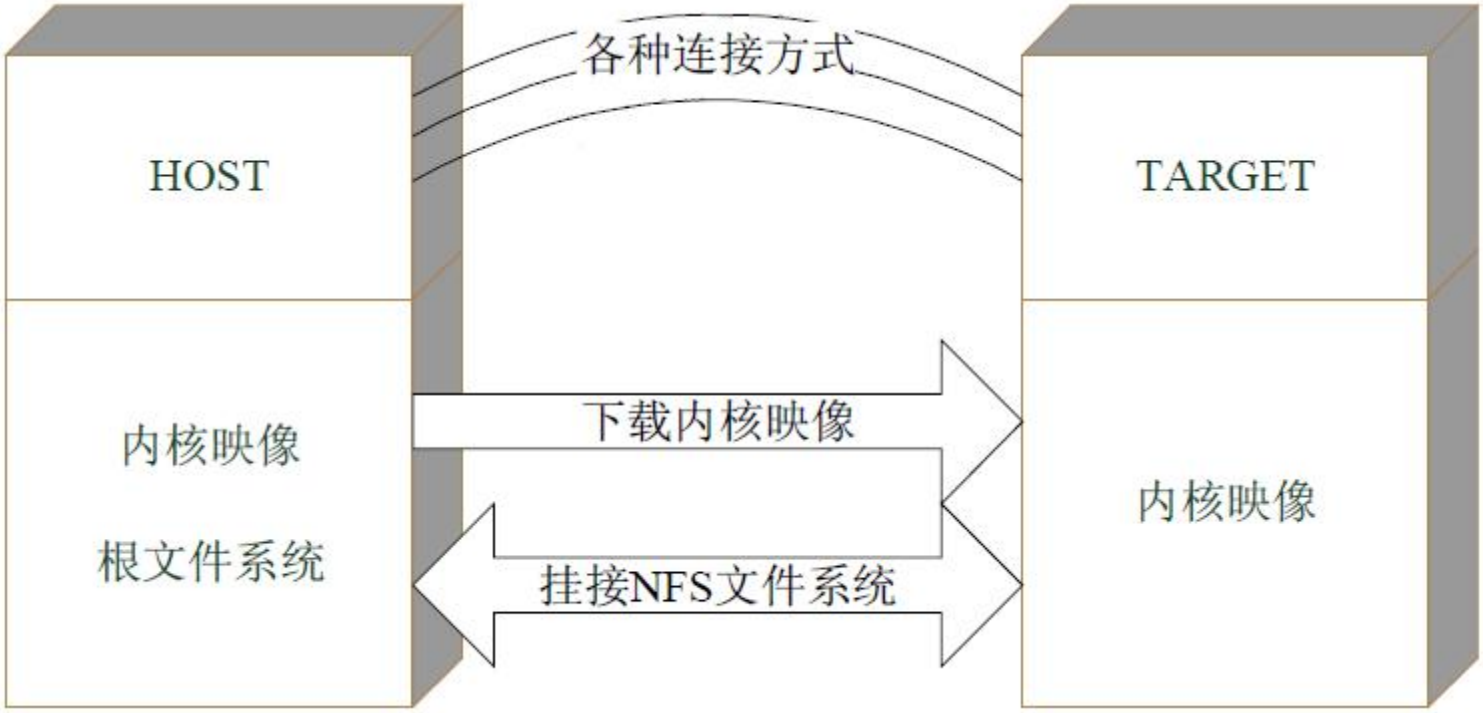
- 避免优化代码场合：
  - 程序开发的时候 优化等级越高，消耗在编译上的时间就越长，因此在开发的时候最好不要使用优化选项，只有到软件发行或开发结束的时候，才考虑对最终生成的代码进行优化。
  - 资源受限的时候 一些优化选项会增加可执行代码的体积，如果程序在运行时能够申请到的内存资源非常紧张（如一些实时嵌入式设备），那就不要对代码进行优化，因为由这带来的负面影响可能会产生非常严重的后果。
  - 跟踪调试的时候 在对代码进行优化的时候，某些代码可能会被删除或改写，或者为了取得更佳的性能而进行重组，从而使跟踪和调试变得异常困难。

5.7 常见错误或警告

- dl_list.h:42:7: warning: no newline at end of file
  - 这个通常是由于源码里最后一行不是UNIX文本字符要求的空行,造成,这不影响最终编译结果,如果想去掉,直接在源码中加入一个空行即可
- test_link.o(.text+0x308): In function ‘test2’:
  undefined reference to ‘dl_save_to_file’
  - 表示在最终链接时,没有找到dl_save_to_file这个函数的目标代码,原因有多种,如没有把这个源码包含进来,系统函数的笔误.找不到对应链接库
- main.c:8:47: String.h: No such file or directory
  - 某个头文件找不到,很大可能性是没有用-I参数把String.h所有在目录加入进来.
- ./main: error while loading shared libraries: libstr.so: cannot open shared object file: No such file or directory
  - 应用程序找不到自己链接的动态库libstr.so,需要把libstr.so加入到环境变量LD_LIBRARY_PATH中

5.8 嵌入式gcc简介

- 交叉开发模型



图中，TARGET就是目标板，HOST是开发主机。在开发主机上，可以安装开发工具，编辑、编译目标板的Linux引导程序、内核和文件系统，然后在目标板上运行。通常这种在主机环境下开发，在目标板上运行的开发模式叫作交叉开发。

5.8 嵌入式gcc简介

- 交叉工具链
  交叉开发模型中，在主机上编译目标板上运行的程序，需要用到交叉工具链。
  交叉工具链在文件名字上加了一个前缀，用来区别本地的工具链。例如：arm-linux-gcc，除了体系结构相关的编译选项以外，它的使用方法与Linux主机上的gcc相同。

- 获得交叉编译工具链一般有3种方法：
  1. Linux 社区的开发者已经编译出了常用体系结构的工具链，从因特网上可以直接下载来使用。
  2. 使用crosstool等自动化工具来构建交叉工具链。
  3. 手动编译各个工具源码包来构建交叉工具链。

5.8 嵌入式gcc简介

Neuedu

- 交叉编译器安装

  以直接获取交叉工具链为例。

  1. 下载交叉工具包cross-3.2.tar.bz2。
     http://ftp.arm.linux.org.uk/pub/armlinux/toolchain/
  2. 将压缩包拷贝至根目录，并解压。
  3. 设置环境变量。

```
root@ubuntu: /
File  Edit  View  Terminal  Help
root@ubuntu:/# cp /mnt/hgfs/VmShare/cross-3.2.tar.bz2 /
root@ubuntu:/# tar -xvjf cross-3.2.tar.bz2
root@ubuntu:/# export PATH=$PATH:/usr/local/arm/bin
```

- 交叉编译器使用

```
hegf@ubuntu: ~/wt_source
File  Edit  View  Terminal  Help
hegf@ubuntu:~/wt_source$ arm-linux-gcc hello.c -o hello
hegf@ubuntu:~/wt_source$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses share
d libs), for GNU/Linux 2.0.0, not stripped
```

编译生成的可执行文件只能在目标板上运行，不能在主机上运行。