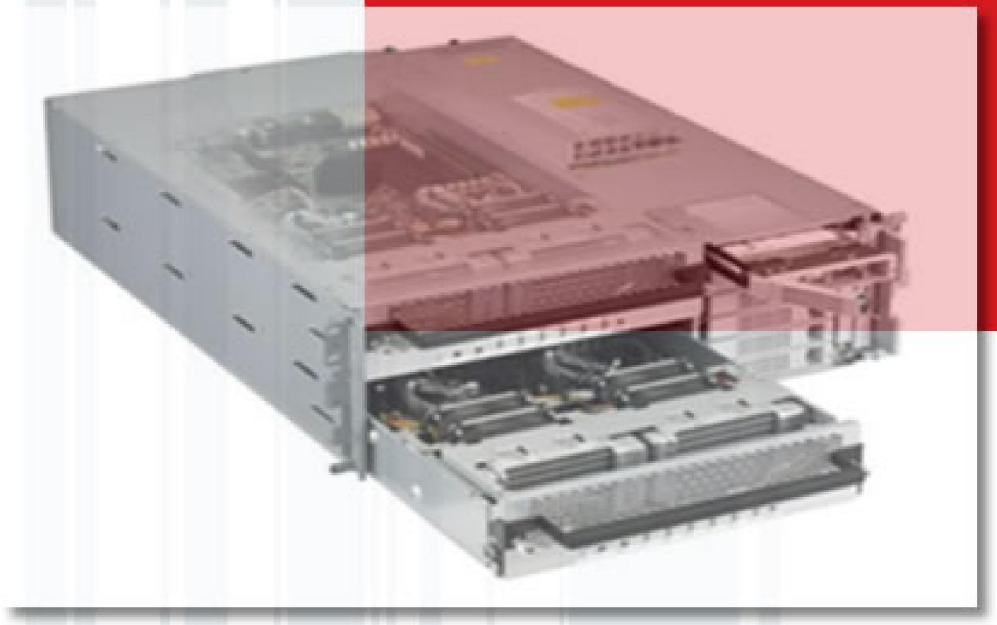
LINUX 环境下程序开发基础

曙光用户培训课程系列



课程时间：1.5小时

更新日期：2008年3月

曙光 DAWNING

Linux环境下程序编译

Linux与C

- Linux与C是天然的结合，从它们的诞生开始就有密切的联系：Linux的前身Unix在用C语言改写之后才为世界所广泛接受；而C语言也是在Unix编写、传播的过程中发展和流行起来的
- Linux平台为C语言提供的编译工具是gcc。Gcc除了处理一般的C语言程序，还支持C++、Objective C等一些语言。曙光4000A提供了PGI C/C++ compiler— pgcc/pgCC
- Linux系统与其他系统类似，C语言程序的运行过程分为三步：
  - 编写源代码
  - 用编译工具编译连接，生成可执行文件
  - 运行该可执行文件

一个简单的例子— hello.c

- 用vi编写源文件:

```
#include <stdio.h>
void main()
{
    printf(“hello world.\n”);
}
```

- 用gcc编译

```
gcc hello.c
```

- 运行

```
a.out
```

GCC简介

- gcc（GNU C Compiler）是GNU推出的功能强大、性能优越的多平台编译器，是GNU的代表作品之一。
- gcc编译器能将C、C++语言源程序、汇程式化序和目标程序编译、连接成可执行文件，如果没有给出可执行文件的名字，gcc将生成一个名为a.out的文件。
- 在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。而gcc则通过后缀来区别输入文件的类别，下面我们来介绍gcc常用的一些后缀。
  - .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
  - .o为后缀的文件，是编译后的目标文件
- gcc最基本的用法是：gcc [options] [filenames]
  其中options就是编译器所需要的参数，filenames给出相关的文件名称

GCC常用编译参数

- -c：只编译，不连接成为可执行文件，编译器只是由输入的.c等源代码文件生成.o为后缀的目标文件，通常用于编译不包含主程序的子程序文件。
- -o output_filename：确定输出文件的名称为output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc就给出预设的可执行文件a.out。
- -g：产生符号调试工具(GNU的gdb)所必要的符号资讯，要想对源代码进行调试，我们就必须加入这个选项。
- -O：对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、连接的速度就相应地要慢一些。
- -O2：比-O更好的优化编译、连接，当然整个编译、连接过程会更慢。

GCC常用编译参数

- -Idirname：将dirname所指出的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。C程序中的头文件包含两种情况：

  A)#include <stdio.h>

  B)#include “myinc.h”

  其中，A类使用尖括号(< >)，B类使用双引号(“ ”)。对于A类，预处理程序cpp在系统预设包含文件目录(如/usr/include)中搜寻相应的文件，而对于B类，cpp在当前目录中搜寻头文件，这个选项的作用是告诉cpp，如果在当前目录中没有找到需要的文件，就到指定的dirname目录中去寻找。在程序设计中，如果我们需要的这种包含文件分别分布在不同的目录中，就需要逐个使用-I选项给出搜索路径。

GCC常用编译参数

- -Ldirname：将dirname所指出的目录加入到程序函数档案库文件的目录列表中，是在连接过程中使用的参数。在预设状态下，连接程序ld在系统的预设路径中(如/usr/lib)寻找所需要的档案库文件，这个选项告诉连接程序，首先到-L指定的目录中去寻找，然后到系统预设路径中寻找，如果函数库存放在多个目录下，就需要依次使用这个选项，给出相应的存放目录。
- -lname：在连接时，装载名字为“ libname.a”的函数库，该函数库位于系统预设的目录或者由-L选项确定的目录下。例如，-lm表示连接名为“ libm.a”的数学函数库。

上面我们简要介绍了gcc编译器最常用的功能和主要参数选项，更为详尽的资料可以参看Linux系统的联机帮助。

GCC应用举例

1. gcc hello.c　　生成a.out
2. gcc –o hello helo.c　　生成hello
3. gcc –O –o hello hello.c　　生成hello
4. gcc –O2 –o hello hello.c　　生成hello
5. gcc –c hello.c　　生成hello.o
   gcc –o hello hello.o　　生成hello
6. gcc –c hello1.c　　生成hello1.o
   gcc –c hello2.c　　生成hello2.o
   gcc –o hello hello1.o hello2.o 生成hello
7. gcc –o test test1.o –lm –I/home/czn/include

Make简介

- 在开发大系统时，经常要将程序划分为许多模块。各个模块之间存在着各种各样的依赖关系，在Linux中通常使用Makefile来管理。
  - 由于各个模块间不可避免存在关联，所以当一个模块改动后，其他模块也许会有所更新，当然对小系统来说，手工编译连接是没问题，但是如果是一个大系统，存在很多个模块，那么手工编译的方法就不适用了。
  - 为此，在Linux系统中，专门提供了一个make命令来自动维护目标文件。
  - 与手工编译和连接相比，make命令的优点在于他只更新修改过的文件，而对没修改的文件则置之不理，并且make命令不会漏掉一个需要更新的文件。

一个简单的例子

先举一个例子：a.c b.c两个程序

a.c

```
extern void p(char *);
main()
{
p("hello world");
}
```

b.c

```
void p(char *str)
{
  printf("%sn",str);
}
```

- Makefile

```
hello: a.c b.c
    gcc a.c b.c -o hello   ß 注意这里是一个Tab
```

- 执行make

```
gcc a.c b.c -o hello
```

产生一个叫hello的可执行程序

书写makefile文件

- Makefile时由规则来组成的,每一条规则都有三部分组成:目标(object),依赖(dependency)和命令(command).在上面的例子中, Makefile只有一条规则,其目标为hello,期依赖为a.c b.c,其命令为gcc a.c b.c -o hello.
- 依赖可以是另一条规则的目标,也可以是文件.每一条规则被这样处理.如目标是一个文件是:当它的依赖是文件时,如果依赖的时间比目标要新, 则运行规则所包含的命令来更新目标; 如果依赖是另一个目标则用同样的方法先来处理这个目标.如目标不是一个存在的文件时,则一定执行.

一个简单的makefile文件

- 例如: Makefile

```
hello: a.o b.o
    gcc a.o b.o -o hello
a.o: a.c
    gcc –c a.c
b.o: b.c
    gcc –c b.c
```

- 当运行make时，可以接一目标名(eg:make hello)作为参数，表示要处理改目标。如没有参数，则处理第一个目标。
- 对上述例子执行make，则是处理hello这个目标。
- hello依赖于文件目标a.o和b.o，则先去处理a.o,调用gcc –c a.c来更新a.o，之后更新b.o，最后调用gcc a.c b.o -o hello 来更新hello.

Make中的宏 (macro)

在make中是用宏，要先定义，然后在makefile中引用。宏的定义格式为：

宏名 = 宏的值　　(宏名一般习惯用大写字母)

例：

```
CC = gcc
hello: a.o b.o
	$(CC) a.o b.o -o hello
a.o: a.c
	$(CC) –c a.c
b.o: b.c
	$(CC) –c b.c
```

系统定义的宏

- 还有一些设定好的内部变量，它们根据每一个规则内容定义。
  - $@ 当前规则的目的文件名
  - $< 依靠列表中的第一个依靠文件
  - $^ 整个依靠的列表（除掉了里面所有重复的文件名）。
  - $? 依赖中所有新于目标的
- 以用变量做许多其它的事情，特别是当你把它们和函数混合 使用的时候。如果需要更进一步的了解，请参考 GNU Make 手册。 ('man make', 'man makefile')

修改原先的make file

```
CC      = gcc
CFLAGS  = -O2
OBJS    = a.o b.o
hello: $(OBJS)
        $(CC) $^ -o $@
a.o: a.c
        $(CC) $(CFLAGS) -c $<
b.o: b.c
        $(CC) $(CFLAGS) -c $<
clean:
        rm –f *.o hello
```

隐含规则

- 请注意在上面的例子里，几个产生.o文件的命令都是一样的，都是从.c文件和相关文件里产生.o文件，这是一个标准的步骤。
- 其实make已经知道怎么做—它有一些叫做隐含规则的内置的规则，这些规则告诉它当你没有给出某些命令的时候，应该怎么办。
- 如果你把生成a.o和b.o的命令从它们的规则中删除，make将会查找它的隐含规则，然后会找到一个适当的命令。

- 它的命令会使用一些变量，因此你可以按照你的想法来设定它：它使用变量CC做为编译器，并且传递变量CFLAGS,CPPFLAGS,TARGET_ARCH，然后它加入‘-c’，后面跟变量$<，然后是‘-o’跟变量$@。一个C编译的具体命令将会是：

  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@
- 当然你可以按照你自己的需要来定义这些变量。

用户环境变量的设定

LINUX中BASH环境变量的设定顺序

- 登录Linux后，BASH要读取几个文件，这些文件（启动脚本文件）用来定义BASH环境，如果希望建立标准的别名，或者希望设置各种shell变量，就应该在bash启动文件中进行设置。

- 和Bash的环境设定有关的文件有
  （1）/etc/profile (主要)
  /etc/profile.d/*.sh (主要)
  （2）$HOME/.bash_profile (主要)
  （3）$HOME/.bash_login
  （4）$HOME/.profile
  $HOME/.bash_logout (主要)
  （5）$HOME/.bashrc (主要)
  /etc/bashrc

LINUX中 BASH环境变量的设定顺序

- 登入（login）交互式时（从字符终端或X Window登录）

1. 先执行 /etc/profile(包括/etc/profile.d/*.sh)

2. 接着bash会检查使用者的自家目录中，是否有 .bash_profile 或者 .bash_login或者 .profile，若有，则会执行其中一个，执行顺序为：

.bash_profile 最优先 .bash_login其次 .profile 最后 （执行最先碰到的一个，前面的设定会被后面的覆盖）

3.启动后读取.bashrc

LINUX中BASH环境变量的设定顺序

- 非登录交互式（从其它shell或者bash启动一个新的 shell ）

  bash 会检查使用者的自家目录中是否有 .bashrc，若有则予以执行，这是唯一的启动文件.

- 非交互式（即运行SHELL脚本）

  上述所有脚本都不执行，如果定义了环境变量ENV，则ENV的值作为启动脚本文件名首先执行。在前两种情况下，如果定义了环境变量ENV，则首先读取ENV指定的文件，然后按顺序读取前述脚本文件。

- 登出(exit/logout)时

  bash会检查使用者自家目录中是否有 .bash_logout，若有，则bash会执行其中的指令

LINUX中BASH环境变量的设定顺序

- 各文件用途说明

1. /etc/profile 由 root 所控制, 用来设定适合全体使用者的shell环境

2. 若使用者自己觉得 /etc/profile 的设定, 并不合意, 可以修改自家目录中的 .bash_profile

3. .bash_login 及 .profile 是为了方便那些从 Bourne shell 移转过来的用户, 只要将 Bourne shell 主要的启动档 .profile 移到自家目录中, 放弃使用 .bash_profile 及 .bash_login 即可继续沿用以前的设定环境

LINUX中BASH环境变量的设定顺序

- 各文件用途说明

4. .bashrc 则是用来设定 subshell 的环境的, 之所以要有这个 .bashrc 是为避免 subshell产生时, 又重覆将 /etc/profile 执行一次. 我们发现 .bashrc 中已预先会去执行 /etc/bashrc的指令, 这表示, 或许 root 会将产生 subshell 时的环境设好了,使用者只要沿用 /etc/bashrc的内容, 应该不会有任何问题.

5. .bash_logout 是使用者登出主机之前, 会去执行的设定档, 如果使用者希望在他登出系统之后, 能帮他自动处理一些琐事, 比如: 清除临时文件, 清除屏幕等, 可以在这个档案中加以设定.

使用库

使用编程库

- 编程库两个主要优点
  - Ø 实现代码重用
  - Ø 提供数百行经过测试和调试的工具代码
- 命名和编号约定
  - Ø 以lib开头 （gcc会在-l选项所指定的文件名前自动插入lib）
  - Ø 文件名以.a（archive）结尾的库都是静态库
  - Ø 文件名以.so（shared object）结尾的库都是共享库
  - Ø 如libdl.a是一个静态库而libc.so是一个共享库

使用编程库

- 编号约定
  - 一般格式为library_name.major_num.minor_num.patch_num
  - 如libgdbm.so.2.0.0, library_name为libgdbm.so，major_num是2，minor_num是0，patch_num是0
  - 当库有新的变化和以前不能兼容时需要增加major_num
  - 当库有新的变化又能和以前版本兼容时只改变minor_num
  - 为修正库中错误进行的改动只会改变patch_num

- 两类特殊的库
  - 以_g结尾的库（调试库）
  - 以_p结尾的库（代码剖析库profiling）
  - 如libform_g.a和libform_p.a，他们是基本库libform.a的特殊版本

库操作工具

u nm

- Ø 命令nm列出编入目标文件或者二进制文件的所有符号
- Ø 可以查看程序调用了什么函数
- Ø 或者查看给定的库或者目标文件是否提供了所需的函数

- Ø nm [options] file
- Ø 常用options：
- Ø -c|--demangle 将符号名转换为用户级的名字，在让C++函数名可读方面特别有用
- Ø -s|--print-armap 当用于.a文件时，输出把符号名映射到定义该符号的模块或成员名的索引
- Ø -u |--undefined-only 只显示未定义的符号（在被检查的文件外部定义的符号）

库操作工具

u ar

- Ø 用来操纵高度结构化的库文件（静态库），最常用来创建静态库
- Ø 创建和维护符号名的交叉索引表，如函数和变量名到定义它们的成员之间的交叉索引表

- Ø ar {dmpqrtx} [member] archive files ....

- Ø 常用选项 –c -s -r –q
- Ø -r： 向存档文件插入files，替换已有的任何同名成员。新成员添加到存档文件的末尾。
- Ø -s：创建或升级从符号到定义他们的成员之间的交叉索引映射表
- Ø ranlib –v file等价于 ar –s file

库操作工具

- ldd
  - ldd命令列出为使程序正常运行所需的共享库
  - 用法 ldd [options] file
  - 常用选项：

    -d 执行重定位并报告所有丢失的函数

    -r 执行对函数和数据对象的重定位并报告丢失的任何函数或数据对象。

库操作工具

- ldconfig
  - 用法 ldconfig [options] libs
  - ldconfig决定位于目录/usr/lib(lib64)和/lib(lib64)下的共享库所需的运行的链接，这些链接在命令行上的库被保存在/etc/ld.so.conf中
  - 常用选项
  - -p 仅打印出文件/etc/ld.so.cache的内容，此文件是ld.so所知道的共享库的当前列表
  - -v 更新/etc/ld.so.cache的内容，列出每个库的版本号，扫描的目录和所有创建和更新的链接。

环境变量和配置文件

- 动态链接器/加载器ld.so使用的两个环境变量
  - $LD_LIBRARY_PATH一个由冒号分割的目录清单，在这些目录下可以搜索运行时的共享库
  - 可以用这个环境变量告诉ld.so在哪儿找到没有保存在标准位置的库；这点跟$PATH类似
  - 第二个环境变量是$LD_PRELOAD，一个由空格分割的、附加的、用户指定共享库，它需要在其它库加载前加载。这样可以在其它共享库中有选择性的重载函数。
  - ld.so还使用两个配置文件ld.so.conf和ld.so.preload，这两个文件跟上述环境变量是平行的，除了标准目录/usr/lib(64)和/lib(64)以外，/etc/ld.so.conf中列出了链接器/加载器搜索共享库时要查看的目录。

创建并使用静态库

u 创建静态库

u 例程：头文件 liberr.h 实现文件liberr.c

1. 把代码编译成目标文件 gcc –c liberr.c–o liberr.o
2. 使用ar工具创建静态库 ar rcs liberr.a liberr.o

u 使用静态库

1. 用户程序errtest.c 需要 #include “liberr.h”
2. 使用-static选项链接liberr.a静态库

   gcc errtest.c –o errtest –static –L. –lerr

   注：如果没有指定-static选项，gcc将自动动态连接创建errtest

   $file errtest 检查生成的文件

创建并使用共享库

- 创建共享库
- 例程：头文件 liberr.h 实现文件liberr.c

1. 把代码编译成目标文件 gcc –fPIC –c liberr.c–o liberr.o
2. 链接库： gcc –shared –Wl,-soname,liberr.so –o liberr.so.1.0.0 liberr.o –lc
3. 建立必要的符号链接 ln –s liberr.so.1.0.0 liberr.so.1

   ln –s liberr.so.1.0.0 liberr.so

- 使用共享库

1. gcc –g errtest.c –o errtest –L. Lerr
2. LD_LIBRARY_PATH=$(pwd) ./errtest

谢谢！