作为Linux程序开发员，最好对开发工具和资源的位置有个初步了解。下面简要介绍一下主要的文件夹和应用程序。

**应用程序(Applications)**

应用程序通常都有固定的文件夹，系统通用程序放在/usr/bin，日后系统管理员在本地计算机安装的程序通常放在/usr/local/bin或者/opt文件夹下。除了系统程序外，大部分个人用到的程序都放在/usr/local下，所以保持/usr的整洁十分重要。当升级或者重装系统的时候，只要把/usr/local的程序备份一下就可以了。

一些其他的程序有自己特定的文件夹，比如X Window系统，通常安装在/usr/X11中，或者/usr/X11R6。GNU的编译器GCC，通常放置在/usr/bin或者/usr/local/bin中，不同的Linux版本可能位置稍有不同。

**头文件(Head Files)**

在C语言和其他语言中，头文件声明了系统函数和库函数，并且定义了一些常量。对于C语言，头文件基本上散落于/usr/include和它的子文件夹下。其他的编程语言的库函数分布在编译器定义的地方，比如在一些Linux版本中，X Window系统库函数分布在/usr/include/X11，GNU C++的库函数分布在/usr/include/g++。这些系统库函数的位置对于编译器来说都是“标准位置”，即编译器能够自动搜寻这些位置。

如果想引用位于标准位置之外的头文件，我们需要在调用编译器的时候加上-I标志，来显式的说明头文件所在文件夹。比如，

$ gcc -I/usr/openwin/include hello.c

会告诉编译器除了标准位置外，还要去/usr/openwin/include看看有没有所需的头文件。详细情况见编译器的使用手册(man gcc)。

**库函数(Library Files)**

库函数就是函数的仓库，它们都经过编译，重用性不错。通常，库函数相互合作，来完成特定的任务。比如操控屏幕的库函数(cursers和ncursers库函数)，[**数据库**](http://lib.csdn.net/base/14)读取库函数(dbm库函数)等。

系统调用的标准库函数一般位于/lib以及/usr/lib。C编译器(精确点说，连接器)需要知道库函数的位置。默认情况下，它只搜索标准C库函数。

库函数文件通常开头字母是lib。后面的部分标示库函数的用途(比如C库函数用c标识， 数学库函数用m标示)，小数点后的后缀表明库函数的类型：

* .a 指静态链接库
* .so 指动态链接库

去/usr/lib看一下，你会发现，库函数都有动态和静态两个版本。

与头文件一样，库函数通常放在标准位置，但我们也可以通过-L标识符，来添加新的搜索文件夹，-l指定特定的库函数文件。比如

        $ gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11

上述命令就会在编译期间，链接位于/usr/openwin/lib文件夹下的libX11函数库，编译生成x11fred。

静态链接库(Static Libraries)

最简单的函数库就是一些函数的简单集合。调用库函数中的函数时，需要在调用函数中include定义库函数的头文件。我们用-l选项添加标准函数库之外的函数库。

静态函数库，也称为*archives*，通常以后缀.a结尾。

我们也可以创建维护自己的静态链接库函数。下面就介绍一下：

我们创建的库函数包括两个函数，然后在后面的实例中调用其中之一。两个库函数名字分别是fred和bill，仅仅是输出字符串。

1. 首先，我们分别编写两个源文件(fred.c和bill.c)，源文件如下：

**[cpp]** [view plain](http://blog.csdn.net/liuzhuan_1986/article/details/4323274) [copy](http://blog.csdn.net/liuzhuan_1986/article/details/4323274)

1. /\* fred.c \*/
2. #include <stdio.h>
3. **void** fred(**int** arg)
4. {
5. printf(“fred: you passed %d/n”, arg);
6. }
7. /\* bill.c \*/
8. #include <stdio.h>
9. **void** bill(**char** \*arg)
10. {
11. printf(“bill: you passed %s/n”, arg);
12. }

    2. 接下来，我们将这两个源文件编译为两个独立的目标文件。这里要用到GCC的-c选项。命令如下所示：

        $ gcc -c fred.c bill.c

        $ ls \*.o

        bill.o fred.o

    3. 然后，写一个调用bill的测试函数，在此之前，最好为库函数建立一个头文件。头文件中有对库函数的声明。如果其他函数要调用库函数，必须在其代码中包含头文件。也可以在fred.c 和bill.c中包含该头文件，有利于编译器发现错误。头文件lib.h的内容如下所示：

**[c-sharp]** [view plain](http://blog.csdn.net/liuzhuan_1986/article/details/4323274) [copy](http://blog.csdn.net/liuzhuan_1986/article/details/4323274)

1. /\*
2. This is lib.h. It declares the functions fred and bill for users
3. \*/
4. **void** bill(**char** \*);
5. **void** fred(**int**);

    4. 测试函数program.c比较简单，代码如下：

**[c-sharp]** [view plain](http://blog.csdn.net/liuzhuan_1986/article/details/4323274) [copy](http://blog.csdn.net/liuzhuan_1986/article/details/4323274)

1. #include “lib.h”
2. **int** main()
3. {
4. bill(“Hello World”);
5. exit(0);
6. }

    5. 现在我们可以编译测试一下程序了：

        $ gcc -c program.c

        $ gcc -o program program.o bill.o

        $ ./program

        bill: we passed Hello World

        $

    6. 接下来，我们要创建一个函数库。利用ar函数建立归档文件(archive)，然后将目标文件加入其中。

        $ ar crv libfoo.a bill.o fred.o

        a - bill.o

        a - fred.o

    7. 现在可以使用函数库中的函数了。我们用-l指定函数库的名字。因为该函数库没有在标准文件夹中，我们还需要用-L将当前文件夹"."添加到搜索路径中。编译命令如下所示：

        $ gcc -o program program.o -L. -lfoo

**共享链接库(Shared Libraries)**

静态链接库的一个缺点是，如果我们同时运行了许多程序，并且它们使用了同一个库函数，这样，在内存中会大量拷贝同一库函数。这样，就会浪费很多珍贵的内存和存储空间。使用了共享链接库的Linux就可以避免这个问题。

共享函数库和静态函数在同一个地方，只是后缀有所不同。比如，在一个典型的Linux系统，标准的共享数序函数库是/usr/lib/libm.so。

当一个程序使用共享函数库时，在连接阶段并不把函数代码连接进来，而只是链接函数的一个引用。当最终的函数导入内存开始真正执行时，函数引用被解析，共享函数库的代码才真正导入到内存中。这样，共享链接库的函数就可以被许多程序同时共享，并且只需存储一次就可以了。共享函数库的另一个优点是，它可以独立更新，与调用它的函数毫不影响。

下面是程序运行时寻找，加载动态库的过程：

然后查到可能是因为共享库设置的问题：

下面的几段是转载的，看了之后明白linux怎么找到共享库的

===========================================================================================

Linux 运行的时候，是如何管理共享库(\*.so)的？在 Linux 下面，共享库的寻找和加载是由 /lib/ld.so 实现的。 ld.so 在标准路经(/lib, /usr/lib) 中寻找应用程序用到的共享库。

但是，如果需要用到的共享库在非标准路经，ld.so 怎么找到它呢？

目前，Linux 通用的做法是将非标准路经加入 /etc/ld.so.conf，然后运行 ldconfig 生成 /etc/ld.so.cache。 ld.so 加载共享库的时候，会从 ld.so.cache 查找。

传统上，Linux 的先辈 Unix 还有一个环境变量：LD\_LIBRARY\_PATH 来处理非标准路经的共享库。ld.so 加载共享库的时候，也会查找这个变量所设置的路经。

LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:./lib

export LD\_LIBRARY\_PATH

但是，有不少声音主张要避免使用 LD\_LIBRARY\_PATH 变量，尤其是作为全局变量。这些声音是：

\* LD\_LIBRARY\_PATH is not the answer - http://prefetch.net/articles/linkers.badldlibrary.html

\* Why LD\_LIBRARY\_PATH is bad - http://xahlee.org/UnixResource\_dir/\_/ldpath.html

\* LD\_LIBRARY\_PATH - just say no - http://blogs.sun.com/rie/date/20040710

解决这一问题的另一方法是在编译的时候通过 -R<path> 选项指定 run-time path。

1. 往/lib和/usr/lib里面加东西，是不用修改/etc/ld.so.conf的，但是完了之后要调一下ldconfig，不然这个library 会找不到

2. 想往上面两个目录以外加东西的时候，一定要修改/etc/ld.so.conf，然后再调用ldconfig，不然也会找不到。

比如安装了一个mysql到/usr/local/mysql，mysql有一大堆library在/usr/local/mysql/lib下 面，这时就需要在/etc/ld.so.conf下面加一行/usr/local/mysql/lib，保存过后ldconfig一下，新的 library才能在程序运行时被找到。

3. 如果想在这两个目录以外放lib，但是又不想在/etc/ld.so.conf中加东西（或者是没有权限加东西）。那也可以，就是export一个全局变 量LD\_LIBRARY\_PATH，然后运行程序的时候就会去这个目录中找library。一般来讲这只是一种临时的解决方案，在没有权限或临时需要的时 候使用。

4. ldconfig做的这些东西都与运行程序时有关，跟编译时一点关系都没有。编译的时候还是该加-L就得加，不要混淆了。

5. 总之，就是不管做了什么关于library的变动后，最好都ldconfig一下，不然会出现一些意想不到的结果。不会花太多的时间，但是会省很多的事。

LD\_LIBRARY\_PATH 这个环境变量是大家最为熟悉的，它告诉loader：在哪些目录中可以找到共享库。可以设置多个搜索目录，这些目录之间用冒号分隔开。在linux下，还 提供了另外一种方式来完成同样的功能，你可以把这些目录加到/etc/ld.so.conf中，然后调用ldconfig。当然，这是系统范围内全局有效 的，而环境变量只对当前shell有效。按照惯例，除非你用上述方式指明，loader是不会在当前目录下去找共享库的，正如shell不会在当前目前找 可执行文件一样。

================================================================================================

在shell下尝试设置LD\_LIBRARY\_PATH，以下面这种形式设置，老是报错bash: LD\_LIBRARY\_PATH: command not found，

LD\_LIBRARY\_PATH=/usr/local/lib

LD\_LIBRARY\_PATH = $ LD\_LIBRARY\_PATH:/usr/local/lib

可能是因为系统之前没有设置过LD\_LIBRARY\_PATH，于是改成这样：

export LD\_LIBRARY\_PATH=/usr/local/lib

然后用 echo $LD\_LIBRARY\_PATH检查一下是否真的设置成功，发现可以。

接着在该shell下运行eclipse生成的可执行文件，没有错误。

另外，如果不想每次新启一个shell都设置LD\_LIBRARY\_PATH，可以编辑~/.bash\_profile文件：

$ vi ~/.bash\_profile

添加：

LD\_LIBRARY\_PATH=/usr/local/lib

export LD\_LIBRARY\_PATH

这两行，完成之后.bash\_profile如下所示：

# .bash\_profile

# Get the aliases and functions

if [ -f ~/.bashrc ]; then

        . ~/.bashrc

fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

LD\_LIBRARY\_PATH=/usr/local/lib

export PATH

export LD\_LIBRARY\_PATH

然后运行

$ source ~/.bash\_profile 就行了。

但是这种方法只能用在shell下，想在eclipse里面运行，还是不行：

尝试了eclipse项目properties里面的各种设置都不起作用。

用“eclipse LD\_LIBRARY\_PATH”作为关键字(可见关键字多么重要)才搜到这么篇文章 《eclipse+cdt+gcc编译选项控制》 http://hi.baidu.com/zsffei/blog/item/7b17c043ceb51e1772f05de1.html

才知道应该在eclipse的项目属性-->C/C++ Build-->Settings-->Tool settings-->GCC C++ Linker-->Miscellaneous的Other options (-Xlinker [option])添加 -R/usr/local/lib

运行，一切正常，折腾了一下午，太感动了。

转载自：<http://skatings.blogbus.com/logs/50437681.html>

编译源码后，一般在config时，选择prefix=/usr/local/hstor/ceph\_dir后，在make及make install后，要进行以下环境变量的设置：

Ceph中有/etc/profile /etc/ld.so.conf 文件需要修改，我没有更新SVN

export LD\_LIBRARY\_PATH=/usr/local/hstor/sms\_install/usr/lib/:/usr/lib

export CEPH\_PATH=/usr/local/hstor/ceph\_dir

export LD\_LIBRARY\_PATH=/usr/local/hstor/ceph\_dir/lib:/usr/local/hstor/ceph\_dir/lib64:/usr/local/hstor/ceph\_dir/libexec:$LD\_LIBRARY\_PATH

export PYTHONPATH=/usr/local/hstor/ceph\_dir/lib/python2.7/site-packages:/usr/local/hstor/ceph\_dir/lib64/python2.7/site-packages

#export CEPH\_CONF=/etc/ceph/ceph.conf

export CEPH\_CONF=/usr/local/hstor/ceph\_dir/etc/ceph/ceph.conf

而我的在191.168.50.180上上config时没有选择prefix时，直接make及make install后，

只需要

vi .bash\_profile

export PYTONPATH=/usr/local/lib/python2.7/site-packages

source .bash\_profile

之后ceph –s就正常了。但还是要记住LD\_LIBRARY\_PATH是shell的全局环境变量这个意思，

使用env显示所有的环境变量。使用set显示本地定义的shell环境变量；

export d=hh定义的环境变量，可以用unset d 来删除；

但是用readonly d或者d=”New”，设置后就不能删除了；

环境变量的设置方法：

1,配置文件

/etc/profile 对所有用户都生效，设置后执行source /etc/profile才能生效

家目录下的 .bash\_profile 永久生效但只对当前用户有效

/etc/bashrc 同/etc/profile

家目录的 .bashrc 同.bash\_profile

/etc/profile、/etc/bashrc、~/.bash\_profile、~/.bashrc很容易混淆，他们之间有什么区别？它们的作用到底是什么？  
/etc/profile: 用来设置系统环境参数，比如$PATH. 这里面的环境变量是对系统内所有用户生效的。  
/etc/bashrc:  这个文件设置系统bash shell相关的东西，对系统内所有用户生效。只要用户运行bash命令，那么这里面的东西就在起作用。  
~/.bash\_profile: 用来设置一些环境变量，功能和/etc/profile 类似，但是这个是针对用户来设定的，也就是说，你在/home/user1/.bash\_profile 中设定了环境变量，那么这个环境变量只针对 user1 这个用户生效.  
~/.bashrc: 作用类似于/etc/bashrc, 只是针对用户自己而言，不对其他用户生效。  
另外/etc/profile中设定的变量(全局)的可以作用于任何用户,而~/.bashrc等中设定的变量(局部)只能继承/etc/profile中的变量,他们是"父子"关系.  
~/.bash\_profile 是交互式、login 方式进入 bash 运行的，意思是只有用户登录时才会生效。  
~/.bashrc 是交互式 non-login 方式进入 bash 运行的，用户不一定登录，只要以该用户身份运行命令行就会读取该文件。

2,直接运行export s=dd 只对当前的shell临时生效

一些常用的环境变量

PATH

HOME

HISTSIZE

LOGNAME

SHELL

LANGUAGE

MAIL

PS1 基本提示符，对于root为#

PYTONPATH python模块的程序查找

LD\_LIBRARY\_PATH 程序运行时路径查找