

# Unix 环境与编译工具

讲稿

Version 1.0

Unix Programming Environment & Tools

达内 IT 培训集团

# 修订历史

摘要	日期	修改原因	版本
文档创建	2009-4-18	新建	1.0



## 目录

1. GCC的使用 .....	5
1.1.编译C程序 .....	5
1.1.1.编译执行文件 .....	5
1.1.1.1.C程序中的文件后缀名 .....	5
1.1.1.2.编译单源程序 .....	6
1.1.1.3.编译多源程序 .....	7
1.1.2.编译目标文件 .....	8
1.1.2.1.编译成目标文件 .....	8
1.1.2.2.使用目标文件编译 .....	8
1.1.3.预处理 .....	8
1.1.3.1.预处理编译 .....	8
1.1.3.2.编译预处理文件 .....	8
1.1.3.3.预处理指令介绍 .....	9
1.1.3.4.预定义宏介绍 .....	13
1.1.3.5.预处理与make选项 .....	14
1.1.3.6.编译环境变量 .....	14
1.1.4.生成汇编 .....	14
1.1.4.1.编译成汇编 .....	14
1.1.4.2.编译汇编 .....	14
1.1.5.创建静态库 .....	15
1.1.5.1.编译静态库 .....	15
1.1.5.2.ar指令 .....	15
1.1.5.3.使用静态库 .....	15
1.1.6.创建共享库 .....	16
1.1.6.1.编译共享库 .....	16
1.1.6.2.定位共享库 .....	16
1.1.6.3.使用共享库 .....	17
1.1.6.4.库工具程序介绍 .....	18
1.1.6.5.其他编译选项 .....	22
1.1.7.C语言扩展 .....	22
1.1.7.1.控制C语言版本 .....	22
1.2.编译C++程序(基本上同C一样) .....	23
2.GDB的使用 .....	24
2.1.GDB基础 .....	24
2.1.1.生成调试信息 .....	24
2.1.2.启动调试 .....	24
2.1.3.调试模式设置 .....	25
2.1.4.退出调试 .....	25
2.1.5.  查看帮助 .....	25
2.2.使用GDB控制调试过程 .....	26

# 1.GCC的使用

## 1.1.编译C程序

### 1.1.1.编译执行文件

Linux下最常用的编译器是gcc.(GNU Compiler Collection) 她通过不同的前端模块来支持对各种不同语言的。

编译,如C、C++、Object C、Java、Fortran、Pascal、Ada等语言。GCC是可以在多种硬件平台上编译出可执行。

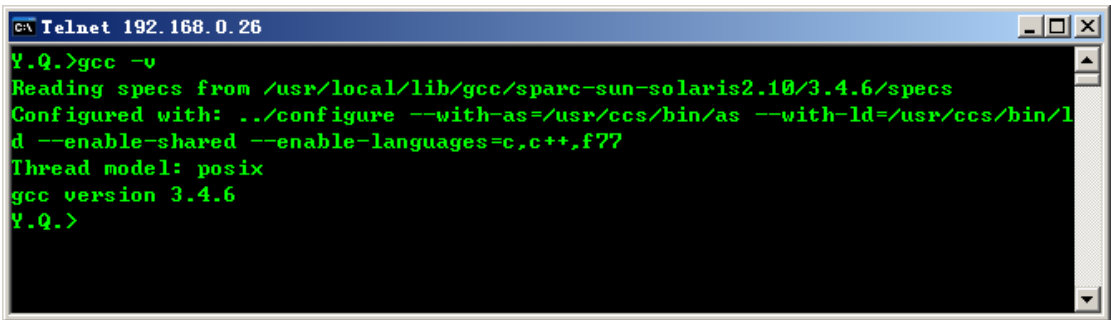
程序的超级编译器.其执行效率与一般的编译器相比，平均效率要高 20%--30%.

在使用GCC编译程序时,编译过程可以细分为 4 个阶段：

- a.预处理。
- b.编译。
- c.汇编。
- d.链接。

程序员可以对编译过程进行控制,同时GCC提供了强大的代码优化功能。

查看 gcc 的版本：gcc -v



```

c:\ Telnet 192.168.0.26
Y.Q.>gcc -v
Reading specs from /usr/local/lib/gcc/sparc-sun-solaris2.10/3.4.6/specs
Configured with: ../configure --with-as=/usr/ccs/bin/as --with-ld=/usr/ccs/bin/ld --enable-shared --enable-languages=c,c++,f77
Thread model: posix
gcc version 3.4.6
Y.Q.>
```

#### 1.1.1.1.C程序中的文件后缀名

扩展名	说明
.a	静态对象库
.c	需要预处理的 C 语言源代码
.h	C 语言源代码头文件
.i	不需要预处理的 C 语言源代码
.o	目标文件
.s	汇编语言代码
.so	共享对象库

### 1.1.1.2.编译单源程序

语法: gcc [选项参数] c 文件

例子: gcc ch01.c

通用选项参数说明如下:

1、指定输出文件名

-o 指定输出文件名

例子: gcc -o main ch01.c

2、警告与提示.

-pedantic 检测不符合ANSI/ISO C语言标准的源代码,使用扩展语法的地方将产生警告信息。

-Wall 生成尽可能多的警告信息。

-Werror 要求编译器将警告当做错误进行处理。

例子: gcc -Wall -o main ch01.c

3、指定编译文件类型

-x 指定编译代码类型, c、c++、assembler, none。None 根据扩展名自动确认。

例子: gcc -x c -Wall -o main ch01.c

4、生成调试信息与优化

-g 生成调试信息

-O 优化

5、建议: 在编译任何程序的时候都带上-Wall 选项。

示例:

```
#!/bin/bash
#GCC 使用
echo "编译....."
gcc -x c -o main -Wall ch01.c

echo "执行....."
./main
```

调试与优化:

```
CA Telnet 192.168.0.26
bash-3.00$ ls
a.out      ch01_1.c  ch02.c    ch02_2.c  ch02_4.c  ch02_6.c  cr.sh     libmy.so
ch01.c     ch01_1.o  ch02_1.c  ch02_3.c  ch02_5.c  ch03.c    libmy.a    main
bash-3.00$ gcc -o ch03.c
bash-3.00$ gcc -o1 ch03.c
bash-3.00$ gcc -o2 ch03.c
bash-3.00$ gcc -o3 ch03.c
bash-3.00$ gcc -g ch03.c
bash-3.00$
```

### 1.1.1.3.编译多源程序

1、语法：gcc [选项] C 源代码 1 C 源代码 2 C 源代码 3

2、示例：

代码 ch01.c

```
#include <stdio.h>
/* 演示编译器 gcc */
//int add(int ,int);
int main()
{
    printf("%d+%d=%d\n",34,68,add(34,68));
    return 0;
}
```

代码 ch01\_1.c

```
/* 函数实现 */
int add (int a,int b)
{
    return a+b;
}
```

编译脚本

```
gcc -x c -o main -Wall ch01.c ch01_1.c
```

注意：在调用处，最好加上显示 add 函数声明，否则会报一个警告(去掉-Wall 不会警告)。Add 函数的声明可以单独存放一个文件，就是头文件。

思考：头文件的作用是什么？

## 1.1.2.编译目标文件

### 1.1.2.1.编译成目标文件

语法: `gcc -c C 源代码文件`

示例:

方式一: 每个 C 文件都生成一个目标文件

```
gcc -c ch01.c ch01_1.c
```

方式二: 多个 C 文件生成一个目标文件

```
gcc -o main.o -c ch01.c ch01_1.c
```

### 1.1.2.2.使用目标文件编译

语法: `gcc -o 输出文件名 目标文件 1 目标文件 1`

示例:

方式一: 编译多个目标文件

```
gcc -o main ch01.o ch01_1.o
```

方式二: 编译一个目标文件

```
gcc -o main main.o
```

## 1.1.3.预处理

### 1.1.3.1.预处理编译

语法: `gcc -E C 源代码文件`

示例:

```
gcc -E -o ch01.i ch01.c
gcc -E -o ch01_1.i ch01_1.c
```

注意:

预处理每次只能处理一个文件。不能处理多个文件, 就是每个 .c 文件对应一个 .i 文件。  
不指定 `-o` 选项, 预处理的结果输出到标准输出设备。

### 1.1.3.2.编译预处理文件

```
gcc -o main ch01.i ch01_1.i
```



### 1.1.3.3.预处理指令介绍

在 C 语法中引入很多预处理指令，这些指令影响到 gcc 的预编译处理结果。

预编译指示符号	说明
#define	定义宏
#elif	else if 多选分支
#else	与#if、#ifndef、#ifdef 结合使用
#error	产生错误,挂起预处理程序
#if	判定
#endif	结束判定
#ifdef	判定宏是否定义
#ifndef	判定宏是否定义
#include	将指定的文件插入#include 的位置
#include_next	与#include 一样,但从当前目录之后的目录查找
#line	指定行号
#pragma	提供额外信息的方法,可用来指定平台
#undef	删除宏
#warning	创建一个警告
##	连接操作符号,用于宏内连接两个字符串

示例:

1、#define 、#undef

```
#include <stdio.h>
/* 演示编译器 gcc */
#define YQ
#define LOUIS "ENGLISH NAME"
int main()
{
    printf("%s\n",LOUIS);
    /*
    #undef LOUIS
    */
    printf("%s\n",LOUIS);
    return 0;
}
```

2、#error、#warning

```
#include <stdio.h>
/* 演示编译器 gcc */

int main()
{
    int a=20;
```

```

    if(a==2)
    {
        // #error "错误很多"
        #warning "警告一下!"
    }
    return 0;
}

```

### 3、#if、#elif、#else、#endif

```

#include <stdio.h>
#define VERSION 3
/* 演示编译器 gcc */
#if (VERSION < 2)
    #error "版本低"
#else
    #warning "版本高"
#endif

int main()
{
    printf("Hello gcc 使用!\n");
    return 0;
}

```

### 4、#ifdef、#else、#endif

```

#include <stdio.h>
#define DEBUG
/* 演示编译器 gcc */
#ifdef DEBUG
    #warning "调试"
#else
    #warning "非调试"
#endif
//这个指示符号的使用还是比较广泛的
int main()
{
    printf("Hello gcc 使用!\n");
    return 0;
}

```

使用**#ifdef**、**#define** 可以防止头文件二次引入。

### 5、#include、#include\_next

说明：

1. 系统头文件使用**#include <...>**

## 2. 用户头文件使用#include “...”

规则：

1. 系统头文件会在 I 参数指定得目录中查找。
2. 用户头文件会在当前目录查找。
3. Unix 标准系统目录
  - /usr/local/include
  - /usr/lib/gcc-lib/.../版本/include
  - /usr/.../include
  - /usr/include
4. 编译 C++ 优先查找/usr/include/g++v3
5. #include <sys/time.h>会在所有标准目录的子目录 sys 中查找 time.h
6. #include 的文件名不不含扩展，\*、?无意义。除非文件名中包含\*。

## 6. #line

```
#include <stdio.h>

int main()
{
    int re=0;
    printf("Hello gcc 使用!\n");
    for(int i=0;i<200)
    {
        re+=i;
    }
    printf("out:%d\n",re);
    //代码行数被修改
    #line 200
    //另外得用法
    // #line 200  "ch01_c.c"
    printf("out:%d\n",re,a);//人为错误
    printf("out:%d\n",re);

    return 0;
}
```

## 7、#pragma

所有 GCC 的 pragma 都定义两个词 GCC +其他

1. #pragma GCC dependency 文件名 提示符号

测试文件的时间戳，当指定文件比当前文件新的时候产生警告。

```
#include <stdio.h>
#pragma GCC dependency "ch02.c"
int main()
{
    int re=0;
```

```
printf("Hello gcc 使用!\n");
int i;
for(i=0;i<200;i++)
{
    re+=i;
}
printf("out:%d\n",re);
return 0;
}
```

## 2. #pragma GCC poison

每次使用指定名字就会产生警告。

```
#include <stdio.h>
#pragma GCC dependency "ch02.c"
#pragma GCC poison printf add
int main()
{
    int re=0;
    printf("Hello gcc 使用!\n");
    int i;
    for(i=0;i<200;i++)
    {
        re+=i;
    }
    printf("out:%d\n",re);
    return 0;
}
```

## 3. #pragma GCC system\_header

该指示符后的代码都做为系统头文件的一部分。

提示：#pragma 有一个等价的宏 \_Pragma

```
#include <stdio.h>
#pragma GCC dependency "ch02.c"
// #pragma GCC poison printf add

int main()
{
    int re=0;
    printf("Hello gcc 使用!\n");
    int i;
    _Pragma("GCC poison printf add")
    for(i=0;i<200;i++)
    {
        re+=i;
    }
}
```

```
}
printf("out:%d\n",re);
return 0;
}
```

```
8、##
#include <stdio.h>
#pragma GCC dependency "ch02.c"
//#pragma GCC poison printf add
#define HELLO(a) a##200
int main()
{
    int re=0;
    printf("Hello gcc 使用:%s!\n",HELLO(99));
    return 0;
}
```

1.1.3.4.预定义宏介绍

预定义的宏很多，下面列出常用的。

宏	说明
__BASE_LINE__	源代码的完整路径
__cplusplus	C++有效，程序不符合标准为 1, 否则是标准的年月
__DATE__	日期
__FILE__	源代码文件名
__func__	当前函数名
__FUNCTION__	同上
__INCLUDE_LEVEL__	包含层数,基本的为 0
__LINE__	行数
__TIME__	时间

示例：

```
#include <stdio.h>
int main()
{
    printf("Hello gcc 使用:%d!\n",__LINE__);
    printf("Hello gcc 使用:%s!\n",__DATE__);
    printf("Hello gcc 使用:%s!\n",__BASE_FILE__);
    return 0;
}
```

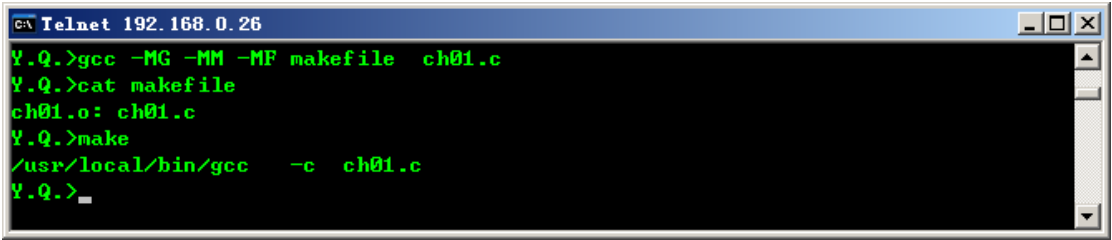
1.1.3.5.预处理与make选项

选项	说明
-M	输出适合 makefile 的规则
-MD	同上，需要显示打开-E
-MMD	与-M 同，但不列出头文件
-MF	指定输出文件名，与 MMD，-MM，-M 结合使用
-MG	假设头文件存在并不包含其他文件
-MM	与-M 同，但不列出系统头文件
-MP	与-M 或-MM 使用，为每个包含文件生成哑目标
-MT	与-M 或-MM 使用，指定 makefile 规则的目标文件名

示例：

```
gcc -MG -MM -MF makefile  ch01.c
```

效果



1.1.3.6.编译环境变量

- C\_INCLUDE\_PATH : 查找头文件的目录。C。
- CPATH : 查找头文件，相当于-I 选项。
- CPLUS\_INCLUDE\_PATH : 查找头文件的目录。C++。
- LD\_LIBRARY\_PATH : 编译没有影响，主要影响运行。指定目录便于定位共享库。
- LIBRARY\_PATH : 查找连接文件，相当于-l 选项。

1.1.4.生成汇编

1.1.4.1.编译成汇编

```
gcc -S ch01.c ch01_1.c
```

注意：汇编编译也要依赖检验的。

1.1.4.2.编译汇编

```
gcc ch01.s ch01_1.s -o main
```

```

C:\ Telnet 192.168.0.26
Y.Q.>gcc -S ch01.c ch01_1.c
Y.Q.>ls
ch01.c      ch01_1.c  ch02.c      ch02_2.c  ch02_4.c  ch02_6.c  cr.sh
ch01.s      ch01_1.s  ch02_1.c    ch02_3.c  ch02_5.c  ch03.c
Y.Q.>gcc ch01.s ch01_1.s -o main
Y.Q.>main
34+68=102
Y.Q.>
```

## 1.1.5.创建静态库

### 1.1.5.1.编译静态库

```
gcc -c -static ch01_1.c
```

其中-static 可选。

### 1.1.5.2.ar指令

```
ar -r libmy.a ch01_1.o
```

语法：ar [选项] 归档文件名 目标文件列表

指令 ar 的常用选项

选项	说明
-d	从归档文件删除指定目标文件列表。
-q	将指定目标文件快速附加到归档文件末尾。
-r	将指定目标文件插入文档，如果存在则更新。
-t	显示目标文件列表
-x	把归档文件展开为目标文件

### 1.1.5.3.使用静态库

```
gcc -o main ch01.c libmy.a
```

如果 libmy.a 在 LIBRARY\_PATH 的指定目录中，还可以采用如下方式编译。

```
gcc ch01.c -o main -lmy
```

注意：上面的先后位置错误可能导致编译错误。

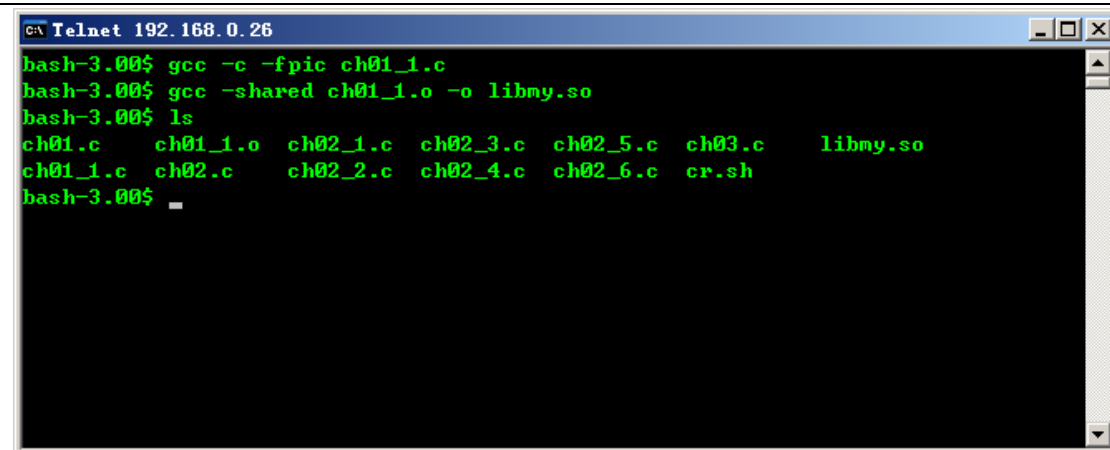
## 1.1.6.创建共享库

### 1.1.6.1. 编译共享库

编译共享库分成两个部分：

1. 编译成位置独立代码的目标文件，选项-fpic
2. 编译成共享库，选项-shared

```
gcc -c -fpic ch01_1.c
gcc -shared ch01_1.o -o libmy.so
```



A terminal window titled 'c:\ Telnet 192.168.0.26' showing the following commands and output:

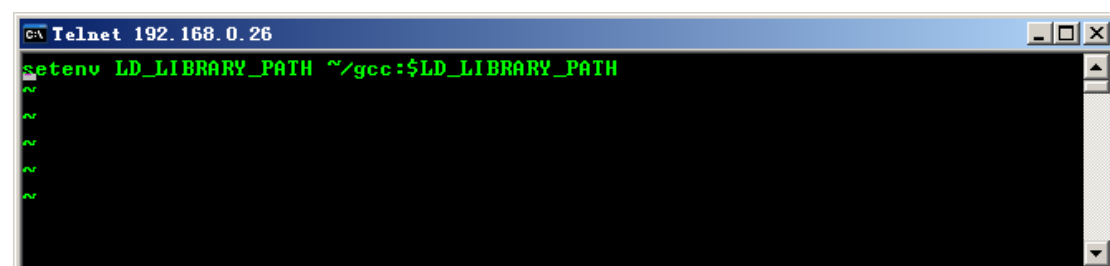
```
bash-3.00$ gcc -c -fpic ch01_1.c
bash-3.00$ gcc -shared ch01_1.o -o libmy.so
bash-3.00$ ls
ch01.c      ch01_1.o  ch02_1.c  ch02_3.c  ch02_5.c  ch03.c    libmy.so
ch01_1.c   ch02.c   ch02_2.c  ch02_4.c  ch02_6.c  cr.sh
```

使用一条指令的效果一样

```
gcc -fpic -shared ch01_1.c -o libmy.so
```

### 1.1.6.2.定位共享库

共享库编译的时候与静态库一样依赖 LIBRARY\_PATH，运行的时候依赖 LD\_LIBRARY\_PATH。  
下面是配置的 LD\_LIBRARY\_PATH



A terminal window titled 'c:\ Telnet 192.168.0.26' showing the following command:

```
setenv LD_LIBRARY_PATH ~/gcc:$LD_LIBRARY_PATH
```

规则：

1. 查找 LD\_LIBRARY\_PATH，目录使用冒号分隔。
2. /etc/ld.so.cache 中找到的列表。工具 ldconfig 维护。
3. 目录/lib
4. 目录/usr/lib



### 1.1.6.3.使用共享库

```
gcc ch01.c libmy.so -o main
```

在代码中动态加载共享库：

共享库代码

```
int add(int a, int b)
{
    int c=a+b;
    c=c/2;
    return c;
}
```

调用共享库代码

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main()
{
    /* 动态库 */
    void *handle;
    /* 加载中的错误描述 */
    char *error;
    /* 调用的函数指针 */
    int (*myadd)(int,int);

    /* 加载共享库 */
    handle=dlopen("./libadd.so",RTLD_LAZY);
    error=dlerror();
    if(error)
    {
        printf("lib load error:%s\n",error);
        exit(1);
    }
    myadd=dlsym(handle,"add");

    error=dlerror();
    if(error)
    {
        printf("lib load error:%s\n",error);
        exit(1);
    }

    int a=myadd(23,89);
```

```
printf("out:%d\n",a);
dlclose(handle);
return 0;
}
```

说明：共享库的四个函数

```
SYNOPSIS

#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
char *dlerror(void);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

其中 dlopen 的参数 flag 的含义如下：

- RTLD\_LAZY：符号查找时候才加载。
- RTLD\_NOW：马上加载。

### 1.1.6.4.库工具程序介绍

#### 1、ldconfig

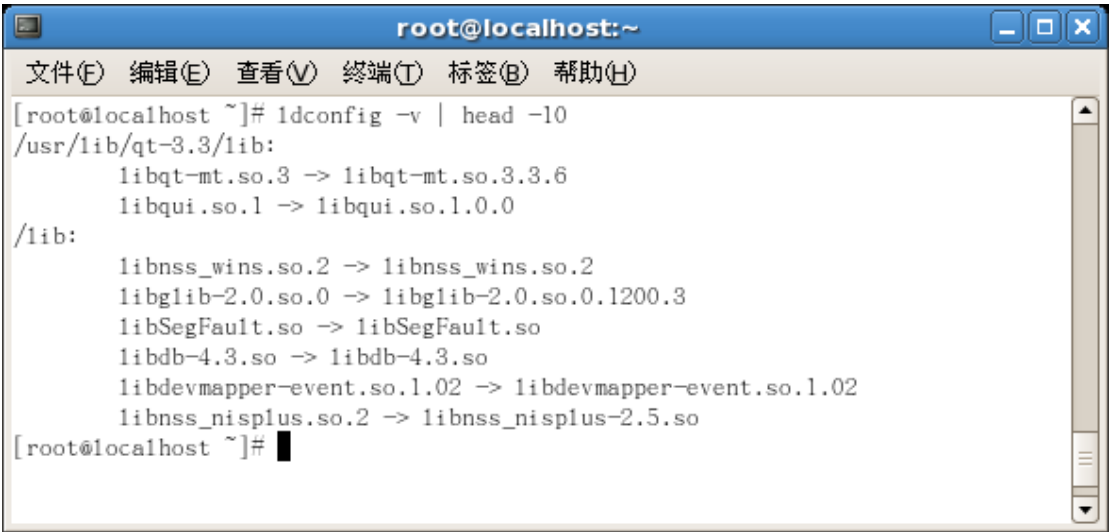
生成文件/etc/ld.so.cache2 新版本：ldconfig -v

其他选项：

选项	说明
-n	不产生缓冲文件, 列出指定目录下的库
-p	显示缓存文件中所有库按字母排序的列表, 包括连接库的完整路径
-v	

示例：

```
ldconfig -v
```



```
ldconfig -vn /lib
```

```
root@localhost:~  
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)  
[root@localhost ~]# ldconfig -vn /lib | head -10  
/lib:  
    libnss_wins.so.2 -> libnss_wins.so.2  
    libglib-2.0.so.0 -> libglib-2.0.so.0.1200.3  
    libSegFault.so -> libSegFault.so  
    libdb-4.3.so -> libdb-4.3.so  
    libdevmapper-event.so.1.02 -> libdevmapper-event.so.1.02  
    libnss_nisplus.so.2 -> libnss_nisplus-2.5.so  
    libnss_compat.so.2 -> libnss_compat-2.5.so  
    libresolv.so.2 -> libresolv-2.5.so  
    libsepol.so.1 -> libsepol.so.1  
[root@localhost ~]#
```

ldconfig -vnp /lib

```
root@localhost:~  
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)  
[root@localhost ~]# ldconfig -vnp /lib | head -10  
742 libs found in cache `/etc/ld.so.cache'  
    libz.so.1 (libc6) => /usr/lib/libz.so.1  
    libz.so (libc6) => /usr/lib/libz.so  
    libxslt.so.1 (libc6) => /usr/lib/libxslt.so.1  
    libxslt.so (libc6) => /usr/lib/libxslt.so  
    libxml2.so.2 (libc6) => /usr/lib/libxml2.so.2  
    libxml2.so (libc6) => /usr/lib/libxml2.so  
    libxmlsec1.so.1 (libc6) => /usr/lib/libxmlsec1.so.1  
    libxmlsec1.so (libc6) => /usr/lib/libxmlsec1.so  
    libxklavier.so.1.1 (libc6) => /usr/lib/libxklavier.so.1.1  
[root@localhost ~]#
```

## 2、nm

列出目标文件中的符号名:nm libmy.a

```
C:\ Telnet 192.168.0.26  
bash-3.00$ nm libmy.a  
  
libmy.a[ch01_1.o]:  
  
[Index]  Value      Size      Type      Bind      Other Shndx  Name  
[2]      :           0!        0!SECT    !LOCL    !0        !2        !  
[3]      :           0!        36!FUNC   !GLOB    !0        !2        !add  
[1]      :           0!        0!FILE    !LOCL    !0        !ABS      !ch01_1.c  
bash-3.00$
```

其他常用参数

选项	说明
-p	不排序显示
-r	逆序排列显示
--size-sort	大小排列显示
-o	用源文件标识成员符号
-s	包含模块的索引信息
-D	对共享库显示动态符号, 而不是静态符号
-g	显示定义为外部的符号

示例:

```

c:\ Telnet 192.168.0.26
bash-3.00$ nm -s libmy.a

libmy.a[ch01_1.o]:

[Index]  Value      Size      Type      Bind      Other Shname      Name
[2]      !           0!        0!SECT    !LOCL     !0        !.text          !
[3]      !           0!        36!FUNC   !GLOB     !0        !.text          !add
[1]      !           0!        0!FILE    !LOCL     !0        !ABS            !ch01_1.c
bash-3.00$

```

### 3、strip

去除指定目标文件与静态库中的调试信息 : strip ch01\_1.o libmy.a

```

c:\ Telnet 192.168.0.26
bash-3.00$ strip ch01_1.o libmy.so libmy.a
strip: WARNING: libmy.a: symbol table deleted from archive
execute 'ar -ts libmy.a' to restore symbol table.
bash-3.00$

```

### 4、ldd

列出共享库的依赖关系: ldd libmy.so

```
C:\ Telnet 192.168.0.26
bash-3.00$ ls
ch01.c  ch01_1.o  ch02_1.c  ch02_3.c  ch02_5.c  ch03.c  libmy.a  main
ch01_1.c  ch02.c  ch02_2.c  ch02_4.c  ch02_6.c  cr.sh  libmy.so
bash-3.00$ ldd libmy.so
        libgcc_s.so.1 =>          /usr/local/lib/libgcc_s.so.1
        libc.so.1 =>            /usr/lib/libc.so.1
        libm.so.2 =>            /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-U210/lib/libc_psr.so.1
bash-3.00$ ldd main
        libmy.so =>             <文件没有发现>
        libc.so.1 =>            /usr/lib/libc.so.1
        libm.so.2 =>            /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-U210/lib/libc_psr.so.1
bash-3.00$
```

## 5、objdump

显示目标文件的内部结构: `objdump -f -h -EB hello.o`

```
root@localhost:/home/louis/qt/examples/chap01/hello
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[root@localhost hello]# ls
hello  hello.cpp  hello.o  hello.pro  Makefile
[root@localhost hello]# objdump -f -h -EB hello.o

hello.o:      file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA           LMA           File off  Align
 0 .text          0000010a  00000000  00000000  00000040  2**4
               CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 1 .data           00000000  00000000  00000000  0000014c  2**2
               CONTENTS, ALLOC, LOAD, DATA
 2 .bss            00000000  00000000  00000000  0000014c  2**2
               ALLOC
 3 .rodata.strl.1  0000000a  00000000  00000000  0000014c  2**0
               CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .gcc_except_table 00000027  00000000  00000000  00000156  2**0
               CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .eh_frame       00000050  00000000  00000000  00000180  2**2
               CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
 6 .comment        0000002e  00000000  00000000  000001d0  2**0
               CONTENTS, READONLY
 7 .note.GNU-stack 00000000  00000000  00000000  000001fe  2**0
               CONTENTS, READONLY
[root@localhost hello]#
```

1.1.6.5.其他编译选项

- I 添加编译器搜索头文件的目录。
- L 添加编译器 搜索库文件的目录。
- l 指示编译器链接指定的库文件。

总结常用编译选项

- c 取消链接,生成目标文件
- Dmacro 定义指定的宏,在代码中用#ifdef检测
- E 预处理输出到标准输出设备
- g3 获得调试的详细信息
- I 头文件的搜索目录
- L 库文件的搜索目录
- l 库文件
- O -O2 -O3 编译优化
- S 汇编输出
- v 启动所有警报
- Wall 发生警报时取消编译
- w 禁止所有警报

1.1.7.C语言扩展

1.1.7.1.控制C语言版本

版本	说明
-ansi	编译标准程序，与 GNU 兼容
-pedantic	严格按照标准，提示警告信息
-std=c89	ISO C89 标准
-std=C99	ISO C99 标准
-std=gnu89	具有 GNU 扩展功能的 ISO C89 标准,某些 C99 标准
-traditional	兼容最原始的 C 语法检测

1.1.7.2.对齐

1.1.7.3.数组扩展

1.1.7.4.属性

1.1.7.5.使用复合语句的返回值

1.1.7.6.枚举不完全类型

1.1.7.7.函数原型

1.1.7.8.整数

1.1.7.9.更换关键字

1.1.7.10.标识地址

1.1.7.11.局部标签

1.1.7.12.switch 语句

1.1.7.13.typedef

1.1.7.14.typeof

1.1.7.15.联合类型强制转换

## 1.2.编译C++程序(基本上同C一样)

1.2.1.编译执行文件

1.2.1.1.C++后缀名

1.2.1.2.编译单源程序

1.2.1.3.编译多源程序

1.2.2.编译目标文件

1.2.2.1.编译成目标文件

1.2.2.2.使用目标文件编译

1.2.3.预处理

1.2.3.1.预处理编译

1.2.3.2.使用预处理文件编译

1.2.4.生成汇编

1.2.4.1.编译成汇编文件

1.2.4.2.使用汇编文件编译

### 1.2.5.创建静态库

#### 1.2.5.1.编译静态库

#### 1.2.5.2.使用 ar 打包静态库

#### 1.2.5.3.使用静态库

### 1.2.6.创建共享库

#### 1.2.6.1.编译动态库

#### 1.2.6.2.使用动态库

### 1.2.7.C++语言的扩展

#### 1.2.7.1.头文件引入

#### 1.2.7.2.函数名

#### 1.2.7.3.接口与实现

## 2.GDB的使用

### 2.1.GDB基础

#### 2.1.1.生成调试信息

在编译程序的时候使用-g 选项生成调试信息。

#### 2.1.2.启动调试

语法： gdb <使用-g 选项编译的可执行文件名>

也可以直接启动 gdb

其他启动方式

gdb <程序名>

gdb <程序名> core 用 gdb 同时调试一个运行程序和 core 文件.

core 是程序非法执行后 core dump 产生的文件.

gdb <程序名> <pid> 指定进程号

gdb 的常用启动开关.



-symbols <file>  
-s 从指定文件读取符号表.  
-se 从指定文件读取符号表, 并用在可执行文件中.  
-core <file>  
-c <file> 调试的时候 core dump 的 core 文件  
-directory <dir>  
-d <dir> 加入源文件的搜索路径. 默认搜索路径是 PATH

### 2.1.3. 调试模式设置

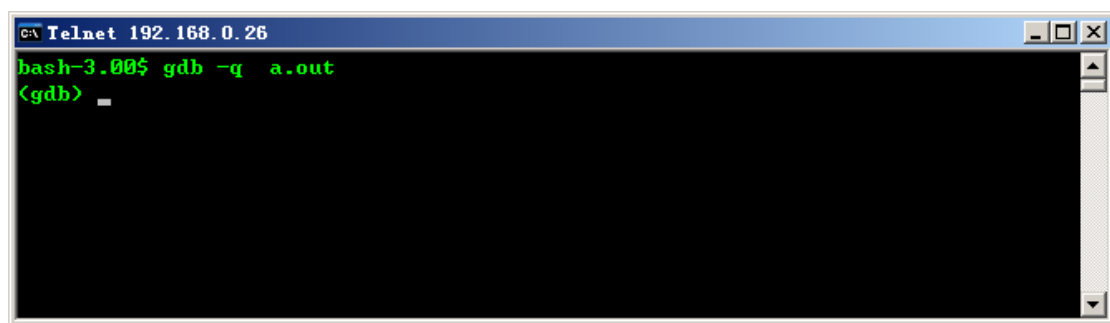
调试模式两种: 批模式与安静模式

安静模式: `gdb -quiet` 或者 `gdb -q`.

若不想看见 logo 请加 `-q` 或 `-quiet` 参数。

批模式 : `gdb -batch` 一般做为过滤器运行使用。

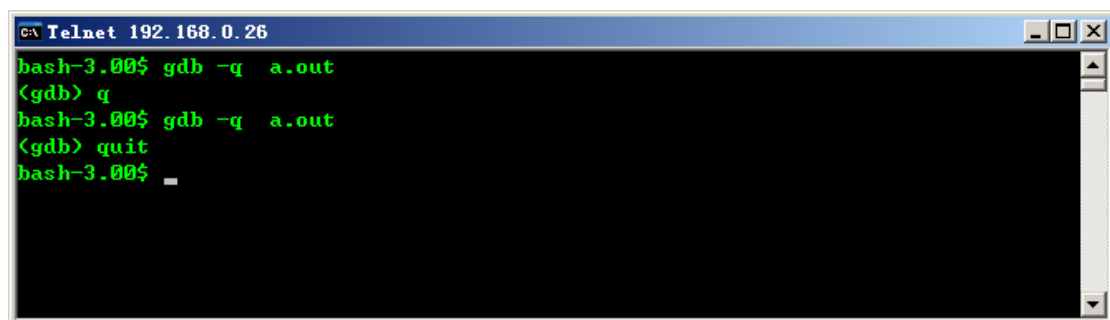
示例:



```
CA Telnet 192.168.0.26
bash-3.00$ gdb -q a.out
(gdb) _
```

### 2.1.4. 退出调试

在 `gdb` 下使用 `quit` 与 `q` 退出



```
CA Telnet 192.168.0.26
bash-3.00$ gdb -q a.out
(gdb) q
bash-3.00$ gdb -q a.out
(gdb) quit
bash-3.00$ _
```

### 2.1.5. 查看帮助

在 `gdb` 提示符号下: 输入 `info` 查看所有帮助。

技巧提示: `info` 首字母<tab> 可以获取快速输入。

```
CA Telnet 192.168.0.26
bash-3.00$ gdb -q a.out
(gdb) info
"info" must be followed by the name of an info command.
List of info subcommands:

info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info args -- Argument variables of current stack frame
info auxv -- Display the inferior's auxiliary vector
info breakpoints -- Status of user-settable breakpoints
info catch -- Exceptions that can be caught in the current stack frame
info classes -- All Objective-C classes
info common -- Print out the values contained in a Fortran COMMON block
info copying -- Conditions for redistributing copies of GDB
info dcache -- Print information on the dcache performance
```

快速帮助

```
CA Telnet 192.168.0.26
info variables -- All global and static variable names
info vector -- Print the status of the vector unit
info warranty -- Various kinds of warranty you do not have
info watchpoints -- Synonym for ``info breakpoints``
info win -- List of all displayed windows

Type "help info" followed by info subcommand name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
(gdb)
(gdb) info c
catch    classes    common    copying
(gdb) info c
```

## 2.2.使用GDB控制调试过程

### 2.2.1. 环境变量设置

path 目录 :添加一个新的 PATH 环境变量  
show paths :显示 PATH 环境变量  
show environment 环境变量名 :显示某个环境变量  
set environment 环境变量名=值 :设置某个环境变量的值  
set env 环境变量名=值  
unset environment 环境变量 :删除环境变量  
unset env 环境变量  
cd 目录 :改变目录  
pwd

### 2.2.2. 运行程序

run :执行程序  
file 执行文件名 :指定调试程序

### 2.2.3. 调试一个已经运行的程序

attach 进程 ID

### 2.2.4. 显示附加的信息

info proc all : 显示所有进程信息

#### 2.2.5. 查看代码

list[m,n] m,n 指开始与结束行号!

#### 2.2.6. 设置断点(break b)

break 行号

break 文件名:行号 if 条件

break +n :当前行数后 n 行

break 函数名

break 文件名:函数名 if 条件

break -n

#### 2.2.7. 检查数据(print p)

print 变量名

print \$! 打印历史记录。

print 数组名@n 打印从数组地址开始的 n 格内存区域。

print /格式 变量

x-16 进制

d-10 进制

u-无符号整数

o-8 进制

t-二进制

a-地址格式

c-字符

f-小数

#### 2.2.8. 检查数据类型(whatis)

whatis 变量名

ptype 结构名 //得到结构定义。

#### 2.2.9. 删除断点

clear 函数

clear 文件名:函数

clear 行数

clear 文件名:行数

#### 2.2.10. 显示断点信息。

info breakpoints

#### 2.2.11. 改变变量的值(set variable)。

set variable 变量名=值。

#### 2.2.12. 断点操作

start 开始逐行调试。

next 下一个, 不进入函数 n。

step 下一个, 进入函数 s

finish 完成回到上一个程序环境

continue 继续 c

until 执行到最后循环 u

#### 2.2.13. 关闭启动断点

Enable 断点行数

```
disable
enabled delete
disable once
```

#### 2.2.14. 观察点

```
watch 表达式 :当值发生改变的时候, 程序停止。
info watchpoints :显示观察点
```

#### 2.2.15. 调用 shell 指令

```
shell 命令
```

#### 2.2.16. 查看内存

```
x /fmt 地址
```

#### 2.2.17. 设置命令行参数

```
set args 参数1 参数2 .....
```

#### 2.2.18. 显示命令行参数

```
show args
```

#### 2.2.19. 各种信息查看

```
info locals 所有局部变量
info args
info frame <用 up down 展开栈>
```

#### 2.2.20. 终止正在调试的程序

```
kill 终止正在调试的程序.
```

#### 2.2.21. 设置自动变量

```
display 变量
display /i $pc : 显示指令地址, $pc 是一个特殊的环境变量, 表示指令存放地址。/i 表示输出格式是机器码。
```

#### 2.2.22. 删除自动显示变量与表达式

```
del display
```

#### 2.2.23. 查看汇编

```
disassemble : 当前过程汇编
disassemble 函数名 : 指定过程汇编
disassemble 地址1 地址2 : 指定地址范围内的汇编
```