# 嵌入式系统工程师

# Linux下编程工具(gcc、gdb)

# 大纲

- linux环境开发概述
- linux文件及目录结构
- linux常用命令
- linux文本编辑器vi+gedit
- linuxshell脚本编程
- linux编译器gcc
- linux调试器gdb
- linux工程管理软件—make

# 大纲

- linux环境开发概述
- linux文件及目录结构
- linux常用命令
- linux文本编辑器vi+gedit
- linuxshell脚本编程
- linux编译器gcc
- linux调试器gdb
- linux工程管理软件—make

# 大纲

- GCC概述
- gcc命令
- 动态库与静态库

# 大纲

- GCC概述
- gcc命令
- 动态库与静态库

# GCC概述

- 编译器是将易于编写、阅读和维护的高级计算机语言翻译为计算机能解读、运行的低级机器语言的程序。
- GUN项目中的一个子项目GCC（GNU Compiler Collection，GNU编译器套装）是一个编译器套装，是GNU计划的关键部分，也是GUN最优秀的软件之一。

  GCC最初用于编译C语言，随着项目的发展GCC已经成为了能够编译C、C++、Java、Ada、fortran、Object C、Object C++、Go语言的编译器大家族。

# GCC概述

- GCC的组成

  GCC由cpp(预处理器)、gcc(C编译器)、g++(C++编译器)、binutils（Binary Utilities 二进制工具）等工具组成。

  binutils是辅助GCC的主要软件，常用的工具有：as（汇编器）、ld（链接器）、ar（ar工具）等等。

# 大纲

- GCC概述
- gcc命令
- 动态库与静态库

# gcc

- gcc仅仅作为真实的编译器和链接器的入口。
  - 它会在需要的时候调用其它组件（预处理器、汇编器、连接器），并且会传一些额外的参数给编译器和连接器。
  - 输入文件的类型和传给gcc的参数决定了gcc调用哪些组件。

# gcc

- gcc识别的文件扩展名如下：

| | |
|---|---|
| .c | C语言文件 |
| .i | 预处理后的C语言文件 |
| .C、.cc、.cp、.cpp、.c++、.cxx | C++语言文件 |
| .ii | 预处理后的C++语言文件 |
| .S | 汇编文件 |
| .s | 预处理后的汇编文件 |
| .o | 编译后的目标文件 |
| .a | 目标文件的静态链接库（链接时使用） |
| .so | 目标文件的动态链接库（链接、运行时使用） |

# gcc

- 编译命令格式

  gcc [-option1] ... <filename>

  g++ [-option1] ... <filename>

  1. 命令、选项和源文件之间使用空格分隔
  2. 一行命令中可以有零个、一个或多个选项
  3. 文件名可以包含文件的绝对路径，也可以使用相对路径。
  4. 如果命令中不包含输出可执行文件的文件名，可执行文件的文件名默认为a.out。

- gcc、g++编译选项

| | |
|---|---|
| -o file | 指定生成的输出文件名为file |
| -E | 只进行预处理 |
| -S | 只进行预处理和编译 |
| -c | 只进行预处理、编译和汇编 |
| -Wall | 生成所有级别的警告信息 |
| -w | 关闭所有警告，建议不使用此选项 |

# gcc

| | |
|---|---|
| -O[0-3] | 编译器优化级别<br>数值越大级别越高，0表示不优化 |
| -include file | 插入一个文件<br>等同于源代码中的#include |
| -Dmacro[=def] | 将名为marco的宏定义为def<br>等同于#define macro [def]<br>若[def]忽略不写，则macro等于1 |
| -Umacro | 取消宏的定义<br>等同于源代码中的#undef macro |
| -v | 显示制作GCC工具时的配置命令<br>显示预处理器、编译器的版本号 |

# gcc

当前找 ->库里找 -> -I指定找

| | |
|---|---|
| -Idir | 将dir目录加入头文件搜索目录列表 /user/include<br>优先在dir目录中查找包含的头文件 |
| -Ldir | 将dir目录加入库文件目录列表 额外的库的头文件<br>优先在dir目录中查找库文件 |
| -lname | 链接库为name的库 |
| -static | 链接时使用静态库 |
| -shared | 编译动态库 |
| -g | 在可执行文件中加入标准调试信息 |

- gcc、g++的编译过程

  gcc和g++编译器的编译过程：

  1、预处理

  2、编译

  3、汇编

  4、链接

  编译过程示意图：

  gcc步骤.bmp

- gcc常用编译应用实例:
- 例1:

```
gcc -E hello.c -o hello.i
gcc -S hello.i -o hello.s
gcc -c hello.s -o hello.o
gcc    hello.o -o hello_elf
```

例2:

```
gcc    hello.c -o hello_elf
```

# 大纲

- GCC概述
- gcc
- 动态库与静态库
  - 链接方式
  - 静态库
  - 共享库

# 静态库与动态库

- 链接过程生成可执行代码
- 链接分为两种：**静态链接、动态链接**
- 静态链接的含义：

  由链接器在链接时将库的内容加入到可执行程序中
- 静态链接的特点是：
  - **优点：**

    对运行环境的依赖性较小，具有较好的兼容性
  - **缺点：**

    生成的程序比较大，需要更多的系统资源，在装入内存时会消耗更多的时间

    库函数有了更新，必须重新编译应用程序

# 静态库与动态库

- 动态链接的含义：（运行时拷贝）

  连接器在链接时仅仅建立与所需库函数的之间的链接关系，在程序运行时才将所需资源调入可执行程序

- 动态链接的特点：
  - **优点：**

    在需要的时候才会调入对应的资源函数

    简化程序的升级；有着较小的程序体积

    实现进程之间的资源共享（避免重复拷贝）
  - **缺点：**

    依赖动态库，不能独立运行

    动态库依赖版本问题严重

# 静态库与动态库

- 前面我们编写的应用程序大量用到了标准库函数
- 使用gcc hello.c -o hello时，系统默认采用动态链接的方式进行编译程序，若想采用静态编译，加入-static参数
- 以下是分别采用动态编译、静态编译时文件对比：

gcc hello.c -o hello_share

gcc hello.c -static -o hello_static

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("hello world\n");
6     return 0;
7 }
```

```
-rwxrwxr-x 1 edu edu 7.0K 11月 27 11:50 hello_share
-rwxrwxr-x 1 edu edu 726K 11月 27 11:50 hello_static
```

# 静态库与动态库

- 动态链接库与静态链接库的制作与使用
- 下面我们以把自己编写的函数分别制作为动态库与静态库为例讲解如何制作、使用两种库函数

**mylib.c:**

```c
int max(int x, int y)
{
    return x > y ? x : y;
}
int min(int x, int y)
{
    return x < y ? x : y;
}
```

**mylib.h**

```c
extern int max(int x, int y);
extern int min(int x, int y);
```

**mytest.c**

```c
#include <stdio.h>
#include "mylib.h"

int main(int argc, char *argv[])
{
    int a = 10, b = 20, max_num, min_num;
    max_num = max(a, b);
    min_num = min(a, b);
    printf("max = %d\n", max_num);
    printf("min = %d\n", min_num);
    return 0;
}
```

# 静态库与动态库

- 制作静态链接库:

  静态链链接库在linux中后缀为.a，以lib开头

  如: libtestlib.a

**1.制作:**

静态库多个文件，必须先做成.o

```
#gcc -c mylib.c -o mylib.o //编译目标文件
#ar rc libtestlib.a mylib.o   //制作静态库
```

**2.静态库使用:**

1)库函数、头文件均在当前目录下

```
#gcc -o my_test mytest.c libtestlib.a
```

库名放最后

2)库函数、头文件假设在/opt目录

```
#gcc -o mytest mytest.c -L/opt -ltestlib -I/opt
```

不写lib....a

# 静态库与动态库

### 3) 编译程序时

编译程序时,编译器默认会到/lib/、/usr/lib下查找库函数，到/usr/include下查找头文件

将libmylib.a移到/lib或/usr/lib下

```
#mv   libtestlib.a   /lib
```

将mylib.h移到/usr/include下

```
#mv   mylib.h   /usr/include
```

编译:

```
#gcc mytest.c -o mytest -ltestlib
```

//编译器会自动到/lib下查找库文件,到/usr/include 下查找头文件

# 静态库与动态库

**制作动态链接库:**

```
#gcc -shared mylib.c -o libtestlib.so
//使用gcc编译、制作动态链接库
```

**动态链接库的使用1:**

1) 库函数、头文件均在当前目录下

```
#gcc -o my_test mytest.c libtestlib.so
```

2) 库函数、头文件假设在/opt目录

```
#gcc -o mytest mytest.c -L/opt -ltestlib -I/opt
```

**编译通过**，运行时出错，编译时找到了库函数，但链接时找不到库，执行以下操作，把当前目录加入搜索路径

```
#export LD_LIBRARY_PATH=./:/opt:$LD_LIBRARY_PATH
#./mytest 可找到动态链接库
```

# 静态库与动态库

动态链接库的使用2:

1.库函数、头文件均在系统路径下

```
#cp libtestlib.so  /lib
#gcc mytest.c -o mytest  -ltestlib
#./mytest
```

编译运行都不会出错

# 静态库与动态库

问题：有个问题出现了？

我们前面的静态库也是放在/lib下，那么连接的到底是动态库还是静态库呢？

当静态库与动态库重名时，系统会优先连接动态库，或者我们可以加入-static指定使用静态库

# 大纲

- linux环境开发概述
- linux文件及目录结构
- linux常用命令
- linux文本编辑器vi+gedit
- linuxshell脚本编程
- linux编译器gcc
- **linux调试器gdb**
- linux工程管理软件—make

# gdb简介

- GNU工具集中的调试器是gdb，该程序是一个交互式工具，工作在字符模式。
- 除gdb外，linux下比较有名的调试器还有xxgdb，ddd，kgdb，ups。

# gdb的使用

- gdb是功能强大的调试器，可完成如下调试任务：
  1、设置断点
  2、监视程序变量的值
  3、程序的单步执行
  4、显示/修改变量的值
  5、显示/修改寄存器
  6、查看程序的堆栈情况
  7、远程调试

# gdb的使用

```
#include <stdio.h>

int sum(int m);
int main()
{
        int i,n=0;
        sum(50);
        for(i=1;i<=50;i++)
        {
                n += i;
        }
        printf("The sum of 1-50 is %d \n",n);
        return 0;
}

int sum(int m)
{
        int i,n=0;
        for(i=1;i<=m;i++)
                n += i;
        printf("The sum of 1-m is %d\n",n);
}
~
"test.c" 22L, 257C                                          22,1
```

# gdb的使用

这样才可以被调试

```
[root@localhost gdb]# gcc -g test.c -o test ①
[root@localhost gdb]# gdb test ②
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 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"...
③ (gdb) l
1       #include <stdio.h>
2
3       int sum(int m);
4       int main()
5       {
6               int i,n=0;
7               sum(50);
8               for(i=1;i<=50;i++)
9               {
10                      n += i;
(gdb) list
11              }
12              printf("The sum of 1-50 is %d \n",n);
13              return 0;
14      }
```

# gdb的使用

```
19              for(i=1;i<=m;i++)
20                      n += i;
(gdb) break 7  设置断点，必须有代码的行    ④
Breakpoint 1 at 0x804833f: file test.c, line 7.
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804833f in main at test.c:7
(gdb) run 运行    ⑤
Starting program: /home/lhaoyu/course/experiment/gdb/test

Breakpoint 1, main () at test.c:7          ⑥单步执行：next：n 不进函数
7               sum(50);                                step：s 进函数
(gdb) print n
$1 = 0
(gdb) p i print n 显示变量    ⑦
$2 = 1073828704
(gdb) step
sum (m=50) at test.c:18
18              int i,n=0;
(gdb) continue 全速到下一个断点    ⑧
Continuing.
The sum of 1-m is 1275
The sum of 1-50 is 1275

Program exited normally.
(gdb)
```

# gdb的使用

1. l（list） 列出程序清单
2. r（run）运行程序
3. b(breakpoint) 设置断点，格式：b [行号/函数名]
4. info b 查看断点信息
5. clear [行号]， 清除断点
6. c（continue）继续运行程序
7. s（step） 单步运行，step into
8. n（next）单步运行，step over
9. finish 跳出函数，step out
10. print 变量/表达式，显示变量或表达式的值
11. display 变量/表达式，每次程序停止运行是都显示变量或表达式的值

undisplay