

Linux应用程序开发

整理: Jims of [肥肥世家](#)

<jims.yang@gmail.com>

Copyright © 2006 本文遵从GNU 的自由文档许可证(Free Documentation License)的条款, 欢迎转载、修改、散布。

发布时间: 2006年11月01日

更新时间: 2007年11月14日, 增加网络编程内容。

Abstract

我的Linux应用程序开发笔记, 从这里开始我的Linux开发之旅。

Table of Contents

[1. C语言基础](#)

- [1.1. 数据类型](#)
- [1.2. 关键字](#)
- [1.3. 变量等级](#)
- [1.4. 特殊字符的表示方法:](#)
- [1.5. 格式化字符串](#)
- [1.6. 指针与数组](#)
- [1.7. 结构体](#)
- [1.8. typedef--自定义类型名](#)
- [1.9. ANSI标准头文件](#)

[2. 使用GCC编译程序](#)

[3. 使用gdb调试程序](#)

[4. Linux程序开发基础](#)

- [4.1. 路径](#)
- [4.2. 库文件](#)
- [4.3. 预处理](#)
- [4.4. 系统调用 \(system call\)](#)

[5. 文件处理](#)

[6. Linux环境编程](#)

- [6.1. 参数选项](#)
- [6.2. 环境变量](#)
- [6.3. 时间](#)
- [6.4. 临时文件](#)
- [6.5. 用户信息](#)
- [6.6. 日志信息](#)

[7. 进程](#)

- [7.1. 进程状态](#)

[8. 串口编程](#)

- [8.1. 常用函数](#)
- [8.2. 设置串口属性](#)

- [8.3. c iflag输入标志说明](#)
- [8.4. c oflag输出标志说明](#)
- [8.5. c cflag控制模式标志说明](#)
- [8.6. c cc\[\]控制字符说明](#)
- [8.7. c lflag本地模式标志说明](#)
- [8.8. 下面介绍一些常用串口属性的设置方法。](#)

9. 安全

- [9.1. 内核漏洞介绍](#)

10. 数据结构(Data Structure)

- [10.1. 基础概念](#)
- [10.2. 线性数据结构](#)

11. 网络编程

- [11.1. TCP/IP协议分析](#)
- [11.2. 入门示例程序](#)

List of Tables

- [1.1. 特殊字符的表示方法](#)

Chapter 1. C语言基础

Table of Contents

- [1.1. 数据类型](#)
- [1.2. 关键字](#)
- [1.3. 变量等级](#)
- [1.4. 特殊字符的表示方法：](#)
- [1.5. 格式化字符串](#)
- [1.6. 指针与数组](#)
- [1.7. 结构体](#)
- [1.8. typedef--自定义类型名](#)
- [1.9. ANSI标准头文件](#)

Linux是使用C语言开发的，基于Linux平台的应用程序开发，C语言是首选的开发语言。本章记录C语言的基本概念和基础知识。

1.1. 数据类型

整数类型（int），

各种整数数制表示法：

- ddd，十进制表示法，d为0--9的整数，但不能以0开头。如：123，345。
- 0ooo，八进制表示法，以0（数字0）开头，o为0--7的整数。如：010(八进制)=8(十进制)，014(八进制)=12(十进制)。
- 0xhhh，十六进制表示法，以0x或0X开头，h为0--9、A、B、C、D、E、F。
如：0x10(十六进制)=16(十进制)，0xA(十六进制)=10(十进制)。
- 以L或l结尾的数表示长整数(long int)，编译器会以32位空间存放此数字，但

GCC默认是以32位存放整数，所以此表示法在Linux下没什么作用。

1.2. 关键字

关键字是C语言本身保留使用的，不能用于变量和函数名。

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

1.3. 变量等级

- **auto**，内部变量，在函数内部声明。只能在函数内部使用，它的生命周期从调用函数开始，到函数执行完时消失。内部变量以堆栈存放，必须在函数执行时才会存在，这种方式称为声明。**auto**可省略。如：

```
auto int i = 0;
/* 可写成int i = 0; */
```

内部变量的优缺点：

- 内部变量只在函数内有效，能提高函数的安全。
 - 内部变量在函数结束时消失，不会长期占用内存空间，能提高内存的利用率。
 - 内部变量的缺点是生命周期短，函数运行结束后不能保留。
- **static auto**，内部静态变量，在函数内部定义，**auto**也可省略。内部静态变量以固定地址存放，编译时就已配置内在空间，这种方式称为定义。由于有固定地址，函数静态变量不会随函数的结束而消失。**static**变量会一直保存在内存空间中，当函数再次执行时，上次保留的使用静态变量可以继续使用。如：

```
static int i = 0;
```

- **extern**，外部变量，是在函数外定义的变量，可被多个函数存取。在外部变量定义覆盖范围内的函数内可以自由使用外部变量。不在外部变量定义覆盖范围内的函数要使用外部变量就要先使用**extern**关键字来声明外部变量。

```
int i;          /* 外部变量定义，在main函数外 */

int main(void)
{
    i = 1;          /* main()函数位于外部变量i定义的下面，不用声明可直接使用 */
    printf("%d\n", i);
}
```

不在外部变量定义覆盖范围内的函数要使用外部变量就要先使用**extern**关键字来声明外部变量。

```
int main(void)
{
extern int i;          /* 外部变量i在main()函数之后定义,需用extern关键字声明后才可
                           使用 */
    i = 1;
    printf("%d\n",i);
}
int i;
...
```

在另外的程序文件中我们也可以通过扩展声明使用其它程序文件中的外部变量。

程序1 hello.c
#include <stdio.h>

```
int main(void)
{
    extern int i;  //扩展声明外部变量
    i = 333;
    printf("%d\n", i);

    extern des(void); //扩展声明外部函数
    des();
}
int i;          //外部变量定义
```

程序2 hello1.c
#include <stdio.h>

```
extern int i;          //扩展声明其它程序文件中的外部变量

void des()
{
    i++;
    printf("%d\n",i);
}
```

编译

```
debian:~/c# gcc hello.c hello1.c
debian:~/c# ./a.out
333
334
```

外部变量有效范围总结:

- 由外部变量定义的位置开始,至文件结尾。
- 不在有效范围内的函数,也可通过**extern**扩展声明使用定义的外部变量,且可在多个函数中使用。注\\意:在各函数中使用的外部变量是一样的,对该变量的修改会影响到其它函数内的同一变量。
- 可用**extern**扩展声明使用另外一个程序文件中的外部变量。

外部变量的优点是生命周期长,可在函数间共享数据和传输数据。缺点是变量安全性较低,但可通过合理设置外部变量的有效范围提高安全性。

- **static extern**, 外部静态变量,在函数外部定义,只供单一程序文件使用,即使

其它程序文件定义了同样名称的变量，编译器也把它当成另外一个变量处理。外部静态变量能有效隔离变量在一个程序文件中。

```
static int i;
```

- **register**，**register**变量是以寄存器（**register**）来存放变量，而不是一般内存。只有内部变量才能使用**register**类型变量。使用这种变量能加快变量的处理速度。但缺点是要占用CPU寄存器。如：

```
register int i;  
register int j;
```

变量等级的概念也同样适用于函数。若想调用不在有效范围内的函数，则要用**extern**扩展声明函数的有效范围。

内部变量是以堆栈方式存放的，必须在函数执行时才会存在，所以称为声明（**Declaration**）。其它如**static auto**、**extern**和**static extern**等级的变量，都是以固定的地址来存放的，而不是以堆栈方式存放的，在程序编译时就已分配了空间，所以称之为定义（**Definition**）。

1.4. 特殊字符的表示方法：

Table 1.1. 特殊字符的表示方法

符号	ASCII字符（十六进制）	句柄符号	作用
\a	07	BEL	响铃
\b	08	BS	回格
\f	0C	FF	换页
\n	0A	LF	换行
\r	0D	CR	回车键
\t	09	HT	[tab]键
\v	0B	VT	空行
\0	00	NUL	空字符
\\	5C	\	反斜杠
\'	2C	'	单引号
\"	22	"	双引号
\?	3F	?	问号

1.5. 格式化字符串

- **%c**，表示字符变量。
- **%s**，表示字符串变量。
- **%f**，表示浮点数变量。
- **%d**，表示整数变量。
- **%x**，表示十六进制变量。

- %o, 表示八进制变量。

1.6. 指针与数组

- C语言中专门用来存放内存地址的变量叫指针（pointer）变量，简称指针。
- &运算符用来取得变量地址，
- "*"运算符用来取得指针变量的值。
- 数组名就是地址变量，指向内存中存放第一个数组元素的地址。数组元素编号从0开始，如a[0]表示数组a的第一个元素。

数组是内存中的连续区间，可根据声明类型存放多种数值类型。如：

```
int a[10];           声明一个有10个int元素的数组
char b[20];          声明一个有20个char元素的数组
```

指针示例：

```
int *p;              /* p是一个指针，p的内容是内存的地址，在这个地址中将存放一个整数。
```

数组名和指针都是用来存放内存地址的，不过数组名具有固定长度，不可变。而指针与一般变量一样，其值是可变的。

1.7. 结构体

结构体是用户定义的由基本数据类型组成的复合式数据类型。数组也是复合式数据类型，但二者是不同的，数组是相同类型数据的集合，而结构体是不同类型数据的集合。如我们可以把一个人的姓名、性别，年龄组成一个单一结构体。这样在程序处理时就把它当成一个独立对象进行处理。

结构体声明方法有两种，一种是分离式声明，一种是结合式声明。分离式声明是先声明结构体，在程序中再声明结构体变量。结合式声明是把结构体声明和变量声明同时完成。

分离式声明示例

```
struct person{
    char name;
    char sex;
    int age;
};
main(void){
    struct person worker;
    ...
}
```

结合式声明示例

```
struct person{
    char name;
    char sex;
    int age;
}worker;
```

每个结构体可以表示一个工人的信息，如果要表示多个工人的信息，则可以用结构

体数组。

```
struct person{
    char name;
    char sex;
    int age;
};
main(void){
    struct person worker[20];    //表示20个工人
    ...
}
```

结构体初始设置。

```
struct person{
    char name;
    char sex;
    int age;
}worker={"jims","male",30};
```

用"."和"->"运算符存取结构体中的数据。"."是直接存取法，"->"为间接存取法，用于结构体指针。如果p是一个指向person结构体的指针，则p->name和(*p).name的结果是一样的。

1.8. typedef-- 自定义类型名

结构体可以自定义数据类型，而typedef可以自定义新的类型名。如：

```
#include <stdio.h>

typedef char *STRING;    //定义一个新的字符指针类型名STRING

main(void){
    STRING a;

    a = "abc";
    printf("the a value is %s.\n",a);
}
```

a为字符指针类型，自定义类型名通常以大写方式表示，以示区别。

#define与typedef的区别是：#define只是单纯地进行变量替换，而typedef是创建新的类型名。typedef的一个主要作用是简化声明，提高程序的可读性。如：

```
typedef struct person{
    char name;
    char sex;
    int age;
} p
```

这样我们就定义一个新的结构体类型名p，在程序中我们可以使用它来声明变量。如

```
main(void){
    p worker;

    worker = {"jims","male",30};
}
```

1.9. ANSI标准头文件

Linux系统头文件位于/usr/include中。默认情况下编译器只在该目录下搜索头文件。

- assert.h, 定义assert宏, 可用来检查程序错误。
- ctype.h,
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h

Chapter 2. 使用GCC编译程序

直接生成a.out可执行文件

```
debian:~/c# gcc hello.c
```

编译hello.c程序, 生成hello可执行文件:

```
debian:~/c# gcc -o hello hello.c
```

生成.s的汇编代码文件。

```
debian:~/c# gcc -S hello.c
```

Chapter 3. 使用gdb调试程序

如果想利用gdb工具来调试程序, 在编译程序时要使用-g选项。如:


```
debian:~/c# gcc -g serial.c -o serial
```

调试serial程序。

```
debian:~/c# gdb serial
GNU gdb 6.5-debian
Copyright (C) 2006 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 "i486-linux-gnu"...Using host libthread_db library "

(gdb) list
8      #include      <errno.h>          /*错误号定义*/
9
10     int main(void)
11     {
12         int fd,n,status,buffsize;
13         struct termios a;
14         struct termios *oldtio;
15         char m[255],*comm;
16
17         fd = open("/dev/ttyS0",O_RDWR|O_NOCTTY|O_NDELAY);
(gdb)
```

gdb的list命令是列出程序源码。下面介绍**gdb**下的各种操作。

- **list**, 列出程序源代码, 一次只列出10行的内容。**list**命令可以指定范围。如: **list 5,10**可列出第5行到第10行的内容。
- **run**, 执行程序。按Ctrl+c可中断程序的执行。
- **shell**, 暂时退出**gdb**回到shell环境。在shell环境用**exit**命令可以返回**gdb**。
- **break**, 设置断点, 后跟行号则把断点设置在指定的行号, 后跟函数名则把断点设置在函数。如**break 6**, **break function**。还可根据条件设置断点, 如: **break 9 if result > 50**。这条命令的意思是, 当运行到第9行时, 如果**result**变量的值大于50, 则中断程序。

```
(gdb) break 6
Breakpoint 1 at 0x8048634: file serial.c, line 6.
```

- **watch**, 指定条件, 如果成立则中断。如: **watch result > 50**。当**result**的变量大于50时, 马上中断程序。
- **print**, 打印变量值, 如: **print result**。
- **whatis**, 查看变量类型, 如: **whatis result**。
- **continue**, 从中断点继续运行程序。
- **step**, 从中断点开始单步运行, 如果遇到函数, 则进入函数单步运行。
- **next**, 从中断点开始单步运行, 如果遇到函数, 则运行函数, 该命令不会进入函数单步运行, 而是运行整个函数。

- `info breakpoints`, 查看程序中所设置的所有中断点信息。

```
(gdb) info breakpoints
Num Type             Disp Enb Address      What
1  breakpoint        keep y    0x08048634 in main at serial.c:6
```

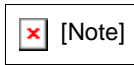
Enb字段是"y", 表示断点1现正生效。

- `disable/enable`, 控制中断点失效和启用。如: `disable 1`。如果`disable/enable`命令后没有指定断点号, 则该命令作用于所有已设置的断点。

```
(gdb) disable 1
(gdb) info breakpoints
Num Type             Disp Enb Address      What
1  breakpoint        keep n    0x08048634 in main at serial.c:6
```

Enb字段由"y"变成"n", 断点1暂时被禁止。

- `enable once`, 使断点生效一次。
- `delete`, 删除断点。如: `delete 1`。`delete`要指定断点号。
- `clear`, 删除断点。如: `clear 6`。`clear`要指定设置断点的行号或函数名。
- `help all`, 显示所有gdb环境的命令。



在gdb环境下, 按tab键可自动补全命令。直接按回车键可重复执行上一个操作。按上下光标键可显示历史命令。

Chapter 4. Linux程序开发基础

Table of Contents

- [4.1. 路径](#)
- [4.2. 库文件](#)
- [4.3. 预处理](#)
- [4.4. 系统调用 \(system call\)](#)

4.1. 路径

在设置Linux的系统路径时, 使用冒号分隔每个路径名。如:

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X"
```

在Linux中的程序有两种, 一种是可执行程序, 与Windows下的.exe文件类似, 一种是脚本, 与Windows下的.bat文件类似。

Linux中常用的程序存放路径有以下几个:

- `/bin`, 该路径存放系统启动时需要使用的程序。
- `/usr/bin`, 该路径存放用户需使用的标准程序。

- `/usr/local/bin`, 该路径存放本地安装的程序。
- Linux使用斜杠"/"分隔路径名, 而不是Windows的反斜杠"\"。
- Linux下的C编译器使用GCC, 由于历史的原因, 在POSIX兼容的操作系统中, C编译器都叫cc, 所以Linux下也有一个cc命令, 它是一个到gcc的软链接。

开发工具, 多数位于`/usr/bin`或`/usr/local/bin`目录下。

头文件, 位于`/usr/include`目录。头文件包含有常量定义、系统调用和库函数调用的声明。这是系统默认的头文件存放路径, 在编译程序时, 编译器会自动查找该目录。gcc编译器在编译程序时也可用`-I`参数指定另外的头文件路径。如:

```
gcc -I/usr/local/myinclude test.c。
```

4.2. 库文件

库文件, 库是一组已编译的函数集合, 可方便我们重用代码。默认存放在`/lib`和`/usr/lib`目录。库文件可分为静态和共享两类。

- `.a`, 静态库文件。使用静态库将会把所有的库代码引入程序, 占用更多的磁盘空间和内存空间, 所以一般建议使用共享库。
- `.so`, 共享库文件。使用共享库的程序不包含库代码, 只在程序运行才调用共享库中的代码。

在编译时可用包含路径的库文件名或用`-l`参数指定使用的库文件, `/usr/lib/libm.a`等价于`-lm`。如:

```
gcc -o hello hello.c /usr/lib/libm.a  
或用-l参数写成  
gcc -o hello hello.c -lm
```

如果我们要使用的库文件不在默认位置, 在编译程序时可用`-L`参数指定库文件的路径。下面例子使用了`/usr/hello/lib`目录下的`libhello`库文件:

```
gcc -o hello -L/usr/hello/lib hello.c -lhello
```

创建和使用静态库。

- 分别创建两个函数, 函数a的内容如下:

```
#include <stdio.h>  
  
void a(char *arg)  
{  
    printf("function a,hello world %s\n",arg);  
}
```

函数b的内容如下:

```
#include <stdio.h>  
  
void b(int arg)
```

```
{
    printf("function b,hello world %d\n",arg);
}
```

- 接着, 生成两个对象文件。

```
debian:~/c# gcc -c a.c b.c
debian:~/c# ls *.o
a.o  b.o
```

- 最后, 用ar归档命令把生成的对象文件打包成一个静态库libhello.a。

```
debian:~/c# ar crv libhello.a a.o b.o
r - a.o
r - b.o
```

- 为我们的静态库定义一个头文件lib.h, 包含这两个函数的定义。

```
/*
 * this is a header file.
 */
void a(char *arg);
void b(int arg);
}}}
* 创建jims.c程序, 内容如下。{{{#!cplusplus
#include "lib.h"

int main()
{
    a("jims.yang");
    b(3);
    exit(0);
}
```

- 利用静态链接库编译程序。

```
debian:~/c# gcc -c jims.c
debian:~/c# gcc -o jims jims.o libhello.a
debian:~/c# ./jims
function a,hello world jims.yang
function b,hello world 3
debian:~/c#
```

 [Note] gcc -o jims jims.o libhello.a也可以写成gcc -o jims jims.o -L. -lhello。

共享库比静态库具有以下优点:

- 当多个进程使用同一共享库时, Linux会把共享库中存放可执行代码的内存进行共享。所以共享库可节省内存, 提高系统性能。
- 程序可共享代码, 减少磁盘空间占用。
- 共享库出错, 只要重新编译共享库即可, 不用重新编译应用程序。

ldconfig程序用来安装一个共享库, 。

只有在为系统库安装一个库的时候,才需要在/etc/ld.so.conf中创建记录,并运行ldconfig更新共享库的缓存。

LD_LIBRARY_PATH环境变量用来指定附加的库文件路径。系统默认的库文件路径位于/usr/lib和/lib目录下。

LD_PRELOAD环境变量指定提前载入的库,用于替代系统库。

4.3. 预处理

预处理,在程序开头以“#”开头的命令就是预处理命令,它在语法扫描和分析法时被预处理程序处理。预处理有以下几类:

- 宏定义,用#define指令定义。如: #define BUFFER 1024。取消宏定义用#undef指令。宏还可带参数,如:

```
#define BUF(x) x*3
```

- 包含头文件,用#include指令,可把包含的文件代码插入当前位置。如:

```
<#include <stdio.h>。
```

包含的文件可以用尖括号,也可用双引号,如:

```
#include "stdio.h"。
```

不同之处是,使用尖括号表示在系统的包含目录(/usr/include)下查找该文件,而双引号表示在当前目录下查找包含文件。每行只能包含一个包含文件,要包含多个文件要用多个#include指令。

- 条件编译,格式如下:

格式一,如果定义了标识符,则编译程序段1,否则编译程序段2:

```
#ifdef 标识符  
程序段1  
#else  
程序段2  
#endif
```

格式二,如果定义了标识符,则编译程序段2,否则编译程序段1,与格式一相反:

```
#ifndef 标识符  
程序段1  
#else  
程序段2  
#endif
```

格式三,常量表达式为真则编译程序段1,否则编译程序段2:

```
#if 常量表达式  
程序段1  
#else  
程序段2  
#endif
```

使用gcc编译程序时,要经过四个步骤。

- 预处理（Pre-Processing），用-E参数可以生成预处理后的文件。

```
debian:~/c# gcc -E hello.c -o hello.i
```

- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

GCC默认将.i文件看成是预处理后的C语言源代码，所以我们可以这样把.i文件编译成目标文件。

```
debian:~# gcc -c hello.i -o hello.o}}
```

在GCC中使用-pedantic选项能够帮助程序员发现一些不符合ANSI/ISO C标准的代码，但不是全部。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。

4.4. 系统调用（system call）

要理解系统调用就要先理解程序代码运行的两种模式，一种是用户模式，一种是内核模式。我们编写的应用程序运行在用户模式下，而设备驱动程序和文件系统运行在内核模式。在用户模式下运行的程序受到严格的管理，不会破坏系统级应用。而在内核模式下运行的程序可以对电脑有完全的访问权。系统调用就是运行在内核模式下的代码为运行在用户模式下的代码提供服务。

系统调用的错误返回码是负数，定义在<errno.h>文件中。在系统调用中发生错误，C函数库就会用错误码填充全局变量errno。用perror()和strerror()函数可以输出错误信息。

系统调用多数在<unistd.h>中定义。

Chapter 5. 文件处理

在Linux系统内所有东西都是以文件的形式来表示的，除一般的磁盘文件外，还有设备文件，如硬盘、声卡、串口、打印机等。设备文件又可分为字符设备文件（character devices）和块设备文件（block devices）。使用man hier命令可以查看Linux文件系统的分层结构。文件的处理方法一般有五种，分别是：

- open，打开一个文件或设备。
- close，关闭一个打开的文件或设备。
- read，从一个打开的文件或者设备中读取信息。
- write，写入一个文件或设备。
- ioctl，把控制信息传递给设备驱动程序。

open，close，read，write和ioctl都是低级的，没有缓冲的文件操作函数，在实际程序开

发中较少使用，一般我们使用标准I/O函数库来处理文件操作。如：fopen, fclose, fread, fwrite, fflush等。在使用标准I/O库时，需用到stdio.h头文件。

- fopen()这个标准I/O库函数用于打开文件，在Linux中文件要先打开后才能进行读写操作。

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);

*mode选项:
"r" 或 "rb" 为读打开文件
"w" 或 "wb" b为写打开文件，如果文件不存在则创建，如果存在则覆盖
"a" 或 "ab" b为追加内容而打开文件
"r+" 或 "rb+" 或 "r+b"r 为更新打开文件，不会覆盖旧文件
"w+" 或 "wb+"b或 "w+b"w 为更新打开文件，会覆盖旧文件
"a+"a或 "ab+" 或 "a+b" 为更新打开文件，更新内容追加到文件末尾
```

一些常用的文件和目录维护函数：chmod、chown、unlink、link、symlink、mkdir、rmdir、chdir、getcwd、opendir, closedir、readdir、telldir、seekdir等。

fcntl用于维护文件描述符，mmap用于分享内存。

创建文档并输入信息的示例代码：

```
#include <stdio.h>

main(void)
{
    FILE *fp1;
    char c;

    fp1 = fopen("text.txt", "w");
    while ((c = getchar()) != '\n')
        putc(c, fp1);
    fclose(fp1);
}
```

显示路径的示例代码

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *topdir = ".";
    if (argc >= 2)
        topdir = argv[1];

    printf("Directory scan of %s\n", topdir);
    printdir(topdir, 0);
    printf("done.\n");

    exit(0);
}

printdir(char *dir, int depth)
```

```

{
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    if((dp = opendir(dir)) == NULL)
    {
        fprintf(stderr, "cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    while((entry = readdir(dp)) != NULL)
    {
        lstat(entry->d_name, &statbuf);
        if(S_ISDIR(statbuf.st_mode))
        {
            if(strcmp(".", entry->d_name) == 0 || strcmp("..", entry->d_name) == 0)
                continue;
            printf("%s%s/\n", depth, "", entry->d_name);
            printdir(entry->d_name, depth+4);
        }
        else printf("%s%s\n", depth, "", entry->d_name);
    }
    chdir("..");
    closedir(dp);
}

```

Chapter 6. Linux环境编程

Table of Contents

[6.1. 参数选项](#)

[6.2. 环境变量](#)

[6.3. 时间](#)

[6.4. 临时文件](#)

[6.5. 用户信息](#)

[6.6. 日志信息](#)

6.1. 参数选项

`void main()`表示程序没有参数, `int main(int argc, char *argv[])`表示程序要带参数, `argc`保存着参数的个数, `argv[]`数组保存着参数列表。如:

```

debian:~# mytest a b c
argc: 4
argv: [ "mytest", "a", "b", "c" ]

```

`getopt()`函数和`getopt_long()`用来处理程序选项。`getopt_long()`函数可以处理以"--"开头的选项。Gnu官方手册页:

http://www.gnu.org/software/libc/manual/html_node/Getopt.html

获取命令行参数的示例代码:

```

#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])

```



```

{
    int opt;

    while((opt = getopt(argc,argv,"if:lr")) != -1)        /* 返回"-1"表示已没选项需
    {
        switch(opt){
            case 'i':
            case 'l':
            case 'r':
                printf("option: %c\n", opt);
                break;
            case 'f':
                printf("filename: %s\n", optarg);          /*如果选项需要一
                break;
            case ':':
                printf("option needs a value \n");          /*": "表示选项需
                break;
            case '?':
                printf("unknown option: %c\n", optopt);      /*返回"?"表示无效
                break;
        }
    }
    for(; optind < argc; optind++)                        /*外部变量optind指向下一个
        printf("argument: %s\n", argv[optind]);
}

```

6.2. 环境变量

在bash shell中使用set命令可以列出Linux系统的环境变量，在C程序中我们也可以使用putenv()和getenv()函数来获取Linux系统的环境变量。这两个函数的声明如下：

```

char *getenv(const char *name);
int putenv(const char *string);

```

系统有一个environ变量记录了所有的系统变量。下面的示例代码可把environ的值显示出来。

```

#include <stdlib.h>
#include <stdio.h>

extern char **environ;

int main()
{
    char **env = environ;

    while(*env)
    {
        printf("%s\n", *env);
        env++;
    }
}

```

6.3. 时间

linux和其它unix一样，使用GMT1970年1月1日子夜作为系统时间的开始，也叫UNIX纪元的开始。现在的时间表示为UNIX纪元至今经过的秒数。

```

#include <time.h>

```

```
time_t time(time_t *t);
```

显示系统时间的示例代码:

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;
    time_t the_time;

    for(i = 1; i <= 10; i++){
        the_time = time((time_t *)0);
        printf("%d the time is %ld\n", i, the_time);
        sleep(2);
    }
}
```

用ctime()函数以友好方式返回当前时间, 它的函数声明格式:

```
#include <time.h>
char *ctime(const time_t *timeval);
```

示例:

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t time1;

    (void)time(&time1);
    printf("The date is: %s\n", ctime(&time1));
}
```

程序输出:

```
The date is: Thu Dec 7 09:58:23 2006
```

用localtime()函数可以返回本地时间, 它是一个tm结构, tm结构体的内容如下:

```
struct tm
{
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};

int tm_sec    Seconds, 0-61
int tm_min    Minutes, 0-59
int tm_hour    Hours, 0-23
int tm_mday    Day in the month, 1-31
int tm_mon    Month in the year, 0-11(January= 0)
int tm_year    Years since 1900
int tm_wday    Day in the week, 0-6. (Sunday = 0)
int tm_yday    Day in the year, 0-365
int tm_isdst    Daylight savings in effect
```

localtime()函数的使用方法如下:

函数声明:

```
#include <time.h>
struct tm *localtime(const time_t *timeval);
```

示例代码:

```
#include <time.h>
#include <stdio.h>
```

```
int main(void)
{
    time_t time1;
    struct tm *p;

    time1 = time(NULL);
    printf("The ctime is: %s\n", ctime(&time1));

    p = localtime(&time1);
    printf("The localtime is:\n tm_year+1900 = %d年\n tm_mon = %d月\n tm_m
}
运行结果:
```

```
The ctime is: Thu Dec  7 10:31:36 2006
```

```
The localtime is:
tm_year+1900 = 2006年
tm_mon = 11月
tm_mday = 7日
wday = 4
hour = 10 时
min = 31分
sec = 36秒
```

6.4. 临时文件

用mkstemp()函数创建临时文件。

```
#include<stdlib.h>
int mkstemp(char * template);
```

示例:

```
#include <stdio.h>
```

```
int main(void)
{
    char template[] = "template-XXXXXX";
    int fp;
    fp = mkstemp(template);
    printf("template = %s\n", template);
    close(fp);
}
```

6.5. 用户信息

获取用户信息。

声明:

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid(uid_t uid); /* 根据uid返回用户信息 */
```

```
struct passwd *getpwnam(const char *name);    /* 根据用户名返回用户信息 */
```

passwd结构体说明:

passwd Member Description

char *pw_name	The user's login name
uid_t pw_uid	The UID number
gid_t pw_gid	The GID number
char *pw_dir	The user's home directory
char *pw_gecos	The user's full name
char *pw_shell	The user's default shell

示例代码:

```
#include <stdio.h>
#include <sys/types.h>
#include <stdio.h>
#include <pwd.h>

int main(void)
{
    uid_t uid;
    gid_t gid;
    struct passwd *pw;

    uid = getuid();
    gid = getgid();
    pw = getpwuid(uid);

    printf("User is %s\n", getlogin());
    printf("The uid is:%d\n", uid);
    printf("The gid is:%d\n",gid);
    printf("The pw struct:\n name=%s, uid=%d, gid=%d, home=%s,shell=%s\n"
}

```

用gethostname()函数获取主机名。

函数声明:

```
#include <unistd.h>
int gethostname(char *name, size_t namelen);    /* 主机名返回给name变量 */
```

示例代码:

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    char computer[100];
    int status;

    status = gethostname(computer, 100);
    printf("The status is %d\n", status);
    printf("The hostname is: %s\n", computer);
}

```

用uname()函数获取主机详细信息, 就像shell的uname命令返回的信息一样。

函数声明:

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

utsname结构体说明:

utsname Member	Description
char sysname[]	The operating system name
char nodename[]	The host name

```
char release[]           The release level of the system
char version[]           The version number of the system
char machine[]           The hardware type
```

示例代码:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/utsname.h>

int main(void)
{
    char computer[100];
    int status;
    struct utsname uts;

    status = gethostname(computer,100);
    printf("The computer's size is %d\n",sizeof(computer));
    printf("The status is %d\n", status);
    printf("The hostname is: %s\n", computer);

    uname(&uts);
    printf("The uname's information.\n uts.sysname=%s\n uts.machine=%s\n uts.release=%s\n",
        uts.sysname, uts.machine, uts.release);
}
```

6.6. 日志信息

使用syslog()函数处理日志信息。

函数声明:

```
#include <syslog.h>
void syslog(int priority, const char *message, arguments...);
```

priority参数的格式 (severity level|facility code)

示例:

```
LOG_ERR|LOG_USER
```

severity level:		Description
Priority Level		
LOG_EMERG		An emergency situation
LOG_ALERT		High-priority problem, such as database corruption
LOG_CRIT		Critical error, such as hardware failure
LOG_ERR		Errors
LOG_WARNING		Warning
LOG_NOTICE		Special conditions requiring attention
LOG_INFO		Informational messages
LOG_DEBUG		Debug messages

facility value (转自syslog.h头文件):

```
/* facility codes */
#define LOG_KERN      (0<<3) /* kernel messages */
#define LOG_USER      (1<<3) /* random user-level messages */
#define LOG_MAIL      (2<<3) /* mail system */
#define LOG_DAEMON    (3<<3) /* system daemons */
#define LOG_AUTH      (4<<3) /* security/authorization messages */
#define LOG_SYSLOG    (5<<3) /* messages generated internally by syslogd */
#define LOG_LPR       (6<<3) /* line printer subsystem */
#define LOG_NEWS      (7<<3) /* network news subsystem */
#define LOG_UUCP      (8<<3) /* UUCP subsystem */
#define LOG_CRON      (9<<3) /* clock daemon */
#define LOG_AUTHPRIV  (10<<3) /* security/authorization messages (private) */
#define LOG_FTP       (11<<3) /* ftp daemon */
```

示例代码:

```
#include <syslog.h>
#include <stdio.h>

int main(void)
{
    FILE *f;

    f = fopen("abc","r");
    if(!f)
        syslog(LOG_ERR|LOG_USER,"test - %m\n");
}
```

上面的日志信息由系统自动给出, 我们也可过滤日志信息。用到以下函数:

```
#include <syslog.h>
void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
```

logopt参数的选项:

logopt Parameter	Description
LOG_PID	Includes the process identifier, a unique number allocated
LOG_CONS	Sends messages to the console if they can't be logged.
LOG_ODELAY	Opens the log facility at first call to .
LOG_NDELAY	Opens the log facility immediately, rather than at first :

示例代码:

```
#include <syslog.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int logmask;

    openlog("logmask", LOG_PID|LOG_CONS, LOG_USER); /*日志信息会包含进程id。*/
    syslog(LOG_INFO, "informative message, pid=%d", getpid());
    syslog(LOG_DEBUG, "debug message, should appear"); /*记录该日志信息。*/
    logmask = setlogmask(LOG_UPTO(LOG_NOTICE)); /*设置屏蔽低于NOTICE级别*/
    syslog(LOG_DEBUG, "debug message, should not appear"); /*该日志信息被屏*/
}
```

不同安全级别的日志信息存放在/var/log目录下的哪个文件中是由/etc/syslog.conf文件控制的, 下面是我系统中syslog.conf文件的内容:

```
# /etc/syslog.conf      Configuration file for syslogd.
#
#                       For more information see syslog.conf(5)
#                       manpage.
#
# First some standard logfiles.  Log by facility.
#
auth,authpriv.*        /var/log/auth.log
*.*;auth,authpriv.none -/var/log/syslog
#cron.*                /var/log/cron.log
daemon.*               -/var/log/daemon.log
kern.*                 -/var/log/kern.log
lpr.*                  -/var/log/lpr.log
mail.*                 -/var/log/mail.log
```

```

user.*                                -/var/log/user.log
uucp.*                                /var/log/uucp.log

#
# Logging for the mail system.  Split it up so that
# it is easy to write scripts to parse these files.
#
mail.info                            -/var/log/mail.info
mail.warn                            -/var/log/mail.warn
mail.err                             /var/log/mail.err

# Logging for INN news system
#
news.crit                            /var/log/news/news.crit
news.err                             /var/log/news/news.err
news.notice                          -/var/log/news/news.notice

#
# Some `catch-all' logfiles.
#
*.=debug;\
    auth,authpriv.none;\
    news.none;mail.none            -/var/log/debug
*.=info;*.=notice;*.=warn;\
    auth,authpriv.none;\
    cron,daemon.none;\
    mail,news.none                 -/var/log/messages

#
# Emergencies are sent to everybody logged in.
#
*.emerg                             *

#
# I like to have messages displayed on the console, but only on a virtual
# console I usually leave idle.
#
#daemon,mail.*;\
#    news.=crit;news.=err;news.=notice;\
#    *.=debug;*.=info;\
#    *.=notice;*.=warn             /dev/tty8

# The named pipe /dev/xconsole is for the `xconsole' utility.  To use it,
# you must invoke `xconsole' with the `-file' option:
#
#    $ xconsole -file /dev/xconsole [...]
#
# NOTE: adjust the list below, or you'll go crazy if you have a reasonably
#       busy site..
#
daemon.*;mail.*;\
    news.crit;news.err;news.notice;\
    *.=debug;*.=info;\
    *.=notice;*.=warn              | /dev/xconsole

```

Chapter 7. 进程

Table of Contents

[7.1. 进程状态](#)

进程是任何正在运行的程序代码，它是操作系统的基本调度单位，只有它能在CPU上运

行。对于一个进程，内核记录以下信息：

- 进程运行的当前位置。
- 进程正在访问的文件。
- 进程的所属的用户和组。
- 进程的当前目录。
- 进程访问的内存空间状况。

7.1. 进程状态

pid是进程的标识符，存放在pid_t结构的变量中。在一个进程中创建另一个进程时，这个新进程就是子进程，原来的进程就是父进程。子进程结束时会通知父进程。如果父进程结束而子进程没有结束，则子进程会成为孤儿进程。所有孤儿进程都会变成init进程的子进程。init进程是系统启动的第一个进程，它其中一个主要功能就是收集孤儿进程，以便内核将子进程从进程表中删除。通过getpid()和getppid()函数可以获得进程的pid。

Chapter 8. 串口编程

Table of Contents

- [8.1. 常用函数](#)
- [8.2. 设置串口属性](#)
- [8.3. c_iflag输入标志说明](#)
- [8.4. c_oflag输出标志说明](#)
- [8.5. c_cflag控制模式标志说明](#)
- [8.6. c_cc\[\]控制字符说明](#)
- [8.7. c_lflag本地模式标志说明](#)
- [8.8. 下面介绍一些常用串口属性的设置方法。](#)

8.1. 常用函数

使用open()函数打开串口，open()函数有两个参数，第一个是要打开的设备名（如：/dev/ttyS0）。第二个是打开的方式。打开方式有以下三种：

- O_RDWR，表示以读写方式打开串口。
- O_NOCTTY，表示不成为端口的控制终端，如果没有这个选项，则任何输入（键盘按键）都会中断程序的执行。
- O_NDELAY，表示程序不会关注DCD信号线所处的状态，即不管对端设备是运行或挂起。如果没有该选项，则程序会被设置成睡眠状态，直到DCD信号为低为止。

成功打开串口则会返回文件描述符，打开失败则返回-1。下面是一个打开串口的示例：


```
fd = open("/dev/ttyS0",O_RDWR|O_NDELAY|O_NDELAY);
```

使用close()关闭打开的串口, 唯一的参数是打开串口的文件描述符。下面是一个关闭串口的示例:

```
close(fd); //fd是打开串口返回的文件描述符
```

用write()函数向串口写数据。下面是一个向串口写数据的示例:

```
n = write(fd,buff,len);
/* n表示成功写到串口的字节数, 如果写入失败则返回-1
   fd是打开串口返回的文件描述符
   buff表示写入的内容
   len表示写入信息的长度。
*/
```

用read()函数从串口读取数据。下面是一个从串口读数据的示例:

```
n = read(fd,buff,len);
/* n表示从串口读到字节数
   fd是文件描述符
   buff是读入字节存放的缓冲区
   len表示读入的字节数
*/
```

通过fcntl()函数可以操作文件描述符, 用以控制读取数据的状态。fcntl(fd,F_SETFL,0)表示没有数据则阻塞, 处于等待状态, 直到有数据到来; fcntl(fd,F_SETFL,FNDELAY)表示当端口没有数据时马上返回0。

8.2. 设置串口属性

所有的串口属性都在一个名为termios的结构体中, 要使用该结构体要包含termios.h头文件。在该头文件中还定义两个重要的函数tcgetattr()和tcsetattr(), 分别用以获取和设置串口的属性。如: tcgetattr(fd,&old_termios), tcsetattr(fd,TCSANOW,&new_termios)。old_termios是旧的串口属性, new_termios是重新设置的新串口属性。tcsetattr()函数中常量的意义是:

- TCSANOW表示新设置的串口属性马上生效。
- TCSADRAIN表示等所有数据传送完成后才生效。
- TCSAFLUSH表示马上清空输入和输出缓存, 然后应用新的串口设置。

termios结构体内容:

成员	描述
c_cflag	控制模式标志
c_lflag	本地模式标志
c_iflag	输入模式标志
c_oflag	输出模式标志
c_line	line discipline
c_cc[NCCS]	控制字符
c_ispeed	输入波特率
c_ospeed	输出波特率

在termios结构中的四个标志控制了输入输出的四个不同部份。输入模式标志c_iflag决定如何解释和处理接收的字符。输出模式标志c_oflag决定如何解释和处理发送到tty设备的字符。控制模式标志决定设备的一系列协议特征, 这一标志只对物理设备有效。本地模式标志c_lflag决定字符在输出前如何收集和处理。

在串口传输中, 用波特率来表示传输的速度, 1波特表示在1秒钟内可以传输1个码元。波特率设置可以使用cfsetispeed(&new_termios,B19200)和cfsetospeed(&new_termios,B19200)这两个函数来完成, 默认的波特率为9600baud。cfsetispeed()函数用来设置输入的波特率, cfsetospeed()函数用来设置输出的波特率。B19200是termios.h头文件里定义的一个宏, 表示19200的波特率。

CLOCAL和CREAD是c_cflag成员中与速率相关的标志, 在串口编程中, 这两个标志一定要有效, 以确保程序在突发的作业控制或挂起时, 不会成为端口的占有都, 同时串口的接收驱动会自动读入数据。设置方法如下:

```
termios_new.c_cflag |= CLOCAL;           //保证程序不会成为端的占有者
termios_new.c_cflag |= CREAD;             //使端口能读取输入的数据
```

设置串口属性不能直接赋值, 要通过对termios不同成员进行"与"和"或"操作来实现。在termios.h文件, 定义了各种常量, 如上面介绍的CLOCAL, CREAD。这些常量的值是掩码, 通过把这些常量与termios结构成员进行逻辑操作就可实现串口属性的设置。在编程时用"|="来启用属性, 用"&=~"来取消属性。

8.3. c_iflag输入标志说明

- BRKINT和IGNBRK

如果设置了IGNBRK, 中断条件被忽略。如果没有设置IGNBRK而设置了BRKINT, 中断条件清空输入输出队列中所有的数据并且向tty的前台进程组中所有进程发送一个SIGINT信号。如果这两个都没有设置, 中断条件会被看作一个0字符。这时, 如果设置了PARMRK, 当检测到一个帧误差时将会向应用程序发送三个字节'\377'\0'\0', 而不是只发送一个'\0'。

- PARMRK和IGNPAR

如果设定了IGNPAR, 则忽略接收到的数据的奇偶检验错误或帧错误(除了前面提到的中断条件)。如果没有设置IGNPAR而设置了PARMRK, 当接收到的字节存在奇偶检验错误或帧错误的时候。将向应用程序发送一个三字节的'\377'\0'\n'错误报告。其中n表示所接收到的字节。如果两者都没有设置, 除了接收到的字节存在奇偶检验错误或帧误差之外的中止条件都会向应用程序发送一个单字节('\0')的报告。

- INPCK

如果设置, 则进行奇偶校验。如果不进行奇偶检验, PARMRK和IGNPAR将对存在的奇偶校验错误不产生任何的影响。

- ISTRIP

如果设置, 所接收到的所有字节的高位将会被去除, 保证它们是一个7位的字符。

- INLCR

如果设置，所接收到的换行字符（'\n'）将会被转换成回车符（'\r'）。

- IGNCR

如果设置，则会忽略所有接收的回车符（'\r'）。

- ICRNL

如果设置，但IGNCR没有设置，接收到的回车符向应用程序发送时会变换成换行符。

- IUCLC

如果IUCLC和IEXTEN都设置，接收到的所有大写字母发送给应用程序时都被转换成小写字母。POSIX中没有定义该标记。

- IXOFF

如果设置，为避免tty设备的输入缓冲区溢出，tty设备可以向终端发送停止符^S和开始符^Q，要求终端停止或重新开始向计算机发送数据。通过停止符和开始符来控制数据流的方式叫软件流控制，软件流控制方式较少用，我们主要还是用硬件流控制方式。硬件流控制在c_cflag标志中设置。

- IXON

如果设置，接收到^S后会停止向这个tty设备输出，接收到^Q后会恢复输出。

- IXANY

如果设置，则接到任何字符都会重新开始输出，而不仅仅是^Q字符。

- IMAXBEL

如果设置，当输入缓冲区空间满时，再接收到的任何字符就会发出警报符'\a'。POSIX中没有定义该标记。

8.4. c_oflag输出标志说明

OPOST是POSIX定义的唯一一个标志，只有设置了该标志后，其它非POSIX的输出标记才会生效。

- OPOST

开启该标记，后面的输出标记才会生效。否则，不会对输出数据进行处理。

- OLCUC

如果设置，大写字母被转换成小写字母输出。

- ONLCR

如果设置, 在发送换行符 ('`\n`') 前先发送回车符 ('`\r`')。

- **ONOCR**

如果设置, 当`current column`为0时, 回车符不会被发送也不会被处理。

- **OCRNL**

如果设置, 回车符会被转换成换行符。另外, 如果设置了`ONLRET`, 则`current column`会被设为0。

- **ONLRET**

如果设置, 当一个换行符或回车符被发送的时候, `current column`会被设置为0。

- **OXTABS**

如果设置, 制表符会被转换成空格符。

8.5. `c_cflag`控制模式标志说明

- **CLOCAL**

如果设置, `modem`的控制线将会被忽略。如果没有设置, 则`open()`函数会阻塞直到载波检测线宣告`modem`处于摘机状态为止。

- **CREAD**

只有设置了才能接收字符, 该标记是一定要设置的。

- **CSIZE**

设置传输字符的位数。`CS5`表示每个字符5位, `CS6`表示每个字符6位, `CS7`表示每个字符7位, `CS8`表示每个字符8位。

- **CSTOPB**

设置停止位的位数, 如果设置, 则会在每帧后产生两个停止位, 如果没有设置, 则产生一个停止位。一般都是使用一位停止位。需要两位停止位的设备已过时了。

- **HUPCL**

如果设置, 当设备最后打开的文件描述符关闭时, 串口上的`DTR`和`RTS`线会减弱信号, 通知`Modem`挂断。也就是说, 当一个用户通过`Modem`拨号登录系统, 然后注销, 这时`Modem`会自动挂断。

- **PARENB**和`PARODD`

如果设置`PARENB`, 会产生一个奇偶检验位。如果没有设置`PARODD`, 则产生

偶校验位, 如果设置了PARODD, 则产生奇校验位。如果没有设置PARENB, 则PARODD的设置会被忽略。

- CRTSCTS

使用硬件流控制。在高速(19200bps或更高)传输时, 使用软件流控制会使效率降低, 这个时候必须使用硬件流控制。

8.6. c_cc[]控制字符说明

只有在本地模式标志c_lflag中设置了IEXITEN时, POSIX没有定义的控制字符才能在Linux中使用。每个控制字符都对应一个按键组合(^C、^H等), 但VMIN和VTIME这两个控制字符除外, 它们不对应控制符。这两个控制字符只在原始模式下才有效。

- c_cc[VINTR]

默认对应的控制符是^C, 作用是清空输入和输出队列的数据并且向tty设备的前台进程组中的每一个程序发送一个SIGINT信号, 对SIGINT信号没有定义处理程序的进程会马上退出。

- c_cc[VQUIT]

默认对应的控制符是^\\, 作用是清空输入和输出队列的数据并向tty设备的前台进程组中的每一个程序发送一个SIGQUIT信号, 对SIGQUIT信号没有定义处理程序的进程会马上退出。

- c_cc[verase]

默认对应的控制符是^H或^?, 作用是在标准模式下, 删除本行前一个字符, 该字符在原始模式下没有作用。

- c_cc[VKILL]

默认对应的控制符是^U, 在标准模式下, 删除整行字符, 该字符在原始模式下没有作用。

- c_cc[VEOF]

默认对应的控制符是^D, 在标准模式下, 使用read()返回0, 标志一个文件结束。

- c_cc[VSTOP]

默认对应的控制字符是^S, 作用是使用tty设备暂停输出直到接收到VSTART控制字符。或者, 如果设备了IXANY, 则等收到任何字符就开始输出。

- c_cc[VSTART]

默认对应的控制字符是^Q, 作用是重新开始被暂停的tty设备的输出。

- c_cc[VSUSP]

默认对应的控制字符是`^Z`，使当前的前台进程接收到一个`SIGTSTP`信号。

- `c_cc[VEOL]`和`c_cc[VEOL2]`

在标准模式下，这两个下标在行的末尾加上一个换行符（`\n`），标志一个行的结束，从而使用缓冲区中的数据被发送，并开始新的一行。`POSIX`中没有定义`VEOL2`。

- `c_cc[VREPRINT]`

默认对应的控制符是`^R`，在标准模式下，如果设置了本地模式标志`ECHO`，使用`VERPRINT`对应的控制符和换行符在本地显示，并且重新打印当前缓冲区中的字符。`POSIX`中没有定义`VERPRINT`。

- `c_cc[VWERASE]`

默认对应的控制字符是`^W`，在标准模式下，删除缓冲区末端的所有空格符，然后删除与之相邻的非空格符，从而起到在一行中删除前一个单词的效果。`POSIX`中没有定义`VWERASE`。

- `c_cc[VLNEXT]`

默认对应的控制符是`^V`，作用是让下一个字符原封不动地进入缓冲区。如果要让`^V`字符进入缓冲区，需要按两下`^V`。`POSIX`中没有定义`VLNEXT`。

要禁用某个控制字符，只需把它设置为`_POSIX_VDISABLE`即可。但该常量只在Linux中有效，所以如果程序要考虑移植性的问题，请不要使用该常量。

8.7. `c_lflag`本地模式标志说明

- `ICANON`

如果设置，则启动标准模式，如果没有设置，则启动原始模式。

- `ECHO`

如果设置，则启动本地回显。如果没有设置，则除了`ECHONL`之外，其他以`ECHO`开头的标记都会失效。

- `ECHOCTL`

如果设置，则以`^C`的形式打印控制字符，如：按`Ctrl+C`显示`^C`，按`Ctrl+?`显示`^?`。

- `ECHOE`

如果在标准模式下设定了`ECHOE`标志，则当收到一个`ERASE`控制符时将删除前一个显示字符。

- `ECHOK`和`ECHOKE`

在标准模式下, 当接收到一个KILL控制符, 则在缓冲区中删除当前行。如果ECHOK、ECHOKE和ECHOE都没有设置, 则用ECHOCTL表示的KILL字符(^U)将会在输出终端上显示, 表示当前行已经被删除。

如果已经设置了ECHOE和ECHOK, 但没有设置ECHOKE, 将会在输出终端显示ECHOCTL表示的KILL字符, 紧接着是换行, 如果设置了OPOST, 将会通过OPOST处理程序进行适当的处理。

如果ECHOK、ECHOKE和ECHOE都有设置, 则会删除当前行。

在POSIX中没有定义ECHOKE标记, 在没有定义ECHOKE标记的系统中, 设置ECHOK则表示同时设置了ECHOKE标志。

- ECHONL

如果在标准模式下设置了该标志, 即使没有设置ECHO标志, 换行符还是会被显示出来。

- ECHOPRT

如果设置, 则字符会被简单地打印出来, 包括各种控制字符。在POSIX中没有定义该标志。

- ISIG

如果设置, 与INTR、QUIT和SUSP相对应的信号SIGINT、SIGQUIT和SIGTSTP会发送到tty设备的前台进程组中的所有进程。

- NOFLSH

一般情况下, 当接收到INTR或QUIT控制符的时候会清空输入输出队列, 当接收到SUSP控制符时会清空输入队列。但是如果设置了NOFLUSH标志, 则所有队列都不会被清空。

- TOSTOP

如果设置, 则当一个非前台进程组的进程试图向它的控制终端写入数据时, 信号SIGTTOU会被发送到这个进程所在的进程组。默认情况下, 这个信号会使进程停止, 就像收到SUSP控制符一样。

- IEXIEN

默认已设置, 我们不应修改它。在Linux中IUCLC和几个与删除字符相关的标记都要求在设置了IEXIEN才能正常工作。

8.8. 下面介绍一些常用串口属性的设置方法。

- 设置流控制

```
termios_new.c_cflag &= ~CRTSCTS;           //不使用流控制
termios_new.c_cflag |= CRTSCTS;             //使用硬件流控制
termios_new.c_iflag |= IXON|IXOFF|IXANY;    //使用软件流控制
```

- 屏蔽字符大小位

```
termios_new.c_cflag &= ~CSIZE;
```

- 设置数据位大小

```
termios_new.c_cflag |= CS8;           //使用8位数据位
termios_new.c_cflag |= CS7;           //使用7位数据位
termios_new.c_cflag |= CS6;           //使用6位数据位
termios_new.c_cflag |= CS5;           //使用5位数据位
```

- 设置奇偶校验方式

```
termios_new.c_cflag &= ~PARENB;       //无奇偶校验

termios_new.c_cflag |= PARENB;         //奇校验
termios_new.c_cflag &= ~PARODD;

termios_new.c_cflag |= PARENB;         //偶校验
termios_new.c_cflag &= ~PARODD;
```

- 停止位

```
termios_new.c_cflag |= CSTOPB;         //2位停止位
termios_new.c_cflag &= ~CSTOPB;        //1位停止位
```

- 输出模式

```
termios_new.c_cflag &= ~OPOST;         //原始数据 (RAW) 输出
```

- 控制字符

```
termios_new.c_cc[VMIN] = 1;            //读取字符的最小数量
termios_new.c_cc[VTIME] = 1;           //读取第一个字符的等待时间
```

- 关闭终端回显, 键盘输入的字符不会在终端窗口显示。

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

int main(void)
{
    struct termios ts,ots;
    char passbuf[1024];

    tcgetattr(STDIN_FILENO,&ts); /* STDIN_FILENO的值是1,表示标准输入的 */
    ots = ts;

    ts.c_lflag &= ~ECHO;          /* 关闭回终端回显功能 */
    ts.c_lflag |= ECHONL;
    tcsetattr(STDIN_FILENO,TCSAFLUSH,&ts); /* 应用新终端设置 */

    fgets(passbuf,1024,stdin);    /* 输入字符不会在终端显示 */
    printf("you input character = %s\n",passbuf);

    tcsetattr(STDIN_FILENO,TCSANOW,&ots); /* 恢复旧的终端设备 */
}
```


Chapter 9. 安全

Table of Contents

[9.1. 内核漏洞介绍](#)

Linux内核以稳定和安全著称，但随着Linux使用范围的不断扩展，各种漏洞也慢慢被内核开发人员或黑客发现。这里介绍有关Linux内核和基于Linux的开源软件的安全问题。

9.1. 内核漏洞介绍

- 权限提升类
- 拒绝服务类
- 溢出类
- IP地址欺骗类

Chapter 10. 数据结构(Data Structure)

Table of Contents

[10.1. 基础概念](#)

[10.2. 线性数据结构](#)

10.1. 基础概念

在实际解决问题的时候，各种数据都不是孤立的，数据之间总是存在关系，这种数据之间的关系叫做数据结构。我们可以把数据结构的形式归并为四种：

- 集合：数据之间没有对应关系，但同属于一个集合。如汽车是一个集合，编程语言也是一个集合。
- 线性结构：各数据有一一对应的关系，有前驱也有后续。
- 树形结构：各数据间存在一对多的关系，有一个前驱但有多个后续。
- 图：各数据间有多对多的关系，对前驱和后续没有限制。

数据类型是一个值的集合和定义在这个值集上的一组操作的总称。

数据类型可分两类，一类是每个对象仅由单值组成，称为原子类型，如整型、字符型等。另一类是由某种结构组成的类型，叫结构类型，如数组、字符串等。

抽象数据结构（Abstract Data Type, ADT）是一种数据类型及在这个类型上定义的一组合法的操作。

算法（Algorithm）是一个有穷规则（或语句、指令）的有序集合。通俗地说，就是计算机解决问题的过程。算法应具备以下几个重要的特性：

- 输入：一个算法有零个或多个输入。
- 输出：一个算法至少有一个输出，这种输出是同输入有着某些特定关系的量。没有输出的算法是没有意义的。
- 有穷性：一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。
- 确定性：算法中每条指令的含义都必须明确，无二义性。对相同的输入，必须有相同的结果。
- 可行性：算法中的每条指令的执行时间都是有限的。

描述算法的工具：自然语言、流程图、形式化语言和程序设计语言。

由瑞士科学家Niklaus Wirth提出的计算机界公认的公式：算法 + 数据结构 = 程序


算法设计的要求：正确、可读、健壮、快速、节省存储空间。

10.2. 线性数据结构

线性结构中的数据元素之间是一种线性关系，数据元素一个接一个地排列。如排除的队列、表格中一行行的记录等。数据元素可以包含多个数据项（字段），包含多个数据项的数据元素叫做记录。由大量记录组成的线性表又称为文件。

线性表的数学表示模型： $a_0, a_1, a_2, \dots, a_{(n-1)}$ 。

顺序连续存放的线性表是最简单的，称为顺序存储结构线性表。它在内存开辟一片连续的存储空间，让线性表的第一个元素存放在内存空间的第一个位置，第二个元素存放在第二个位置，其它元素以此类推。数据元素间的前驱和后继关系表现在存放位置的前后关系上。顺序存储结构线性表算法在插入或删除操作时的效率不高。平均起来，每插入或删除一个元素需要移动一半的元素，最坏的情况更要移动全部的元素。另外，顺序表不利于存储空间的分配。在经常需要进入插入或删除操作的线性表中，使用顺序存储结构线性表是不合适的。所以我们有了链式存储结构线性表。

 [Note] 数组就是顺序存储结构的程序实现。

链式存储结构线性表由结点组成，每个结点由一个数据元素和一个指向下个结点的指针组成。每个结点中如果只有一个指向后续指针的链表，叫单链表。由于链表通过指针指向下一个结点，所以数据元素可以分散存储。

单链表的建立是一种动态内存管理操作，表中的每个节点占用的存储空间无需预先指定，而是在运行时动态申请。

单链表一旦创建就可对链表进行操作。

- 查找值为x的节点，并返回该节点地址。算法分析：从单链表的第一个节点开始，判断当前节点的数据域的值是否为x，若是，则返回该节点的指针域，否则，依据指针域内的指针查找下一节点，直至表结束。若找不到，则返回空。

- 查找第*i*个节点，返回指针。算法分析：从单链表的第一个节点开始，依次判断当前节点是否为第*i*个节点，若是则返回其指针，否则，依据指针域内的指针查找下一节点，直至表结束。若找不到，则返回空。

Chapter 11. 网络编程

Table of Contents

[11.1. TCP/IP协议分析](#)

[11.2. 入门示例程序](#)

11.1. TCP/IP协议分析

EthernetII帧的结构（DMAC+SMAC+Type+Data+CRC），EthernetII帧的大小是有限制的，最小不能小于64字节，最大不能超过1518字节，否则帧会被丢弃。一个EthernetII帧包括的内容有：

- DMAC，目的MAC地址，占48个bit，共6个字节。
- SMAC，源MAC地址，占48个bit，共6个字节。
- Type，帧类型，如ip,arp等。占16个bit，共2个字节。
- Data，帧数据，容量是变化的，但最大不能越过1500个字节，最小不能小于46个字节。
- CRC，校验码，占32个bit，共4个字节。

IP包结构：



11.2. 入门示例程序

下面我们开发一个模拟Echo服务功能的tcp程序。通过这个简单的程序我们可以学习tcp/ip网络编程的基础结构。

tcpserver.c是服务端程序，运行后会监听一个端口。

```
debian:~/c/kernelmodule# cat tcpserver.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/wait.h>
```

```
#include <errno.h>

int main(int argc, char *argv[]){
    int iswork, data, fd1, fd2;
    pid_t pidchild;
    socklen_t clientlen;
    struct sockaddr_in clientaddr;
    struct sockaddr_in serveraddr;
    char buffer[1000];

    if(argc != 2){
        printf("Usage: tcpserver [port number]\n");
        exit(1);
    }

    if((fd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        printf("socket error!\n");
        exit(1);
    }

    memset(&serveraddr, 0, sizeof(serveraddr));

    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(atoi(argv[1]));

    if(bind(fd1, (struct sockaddr*)&serveraddr, sizeof(serveraddr)) < 0){
        printf("bind error!\n");
        exit(1);
    }

    if(listen(fd1, 3) < 0){
        printf("listen error!\n");
        exit(1);
    }

    iswork = 1;
    while(iswork){
        clientlen = sizeof(clientaddr);
        if((fd2 = accept(fd1, (struct sockaddr*)&clientaddr, &clientlen)) < 0){
            printf("accept error!\n");
            exit(1);
        }

        if((pidchild = fork()) == -1){
            printf("fork error!\n");
            exit(1);
        }

        if(pidchild == 0){
            if(close(fd1) == -1){
                printf("close error!\n");
                exit(1);
            }

            printf("Connect from %s\n", inet_ntoa(clientaddr.sin_addr));

            while(1){
                memset(buffer, 0, 1000);

                if(data = read(fd2, buffer, sizeof(buffer)) > 0){
                    printf("%s", buffer);
                    if(write(fd2, buffer, sizeof(buffer)) < 0){
                        printf("send error!\n");
                    }
                }
            }
        }
    }
}
```

```
                                exit(1);
                                }
                                }
                                }
                                exit(0);
                                }

                                if(close(fd2) == -1){
                                    printf("close error!\n");
                                    exit(1);
                                }
                                }
                                }
```

几个主要函数说明:

- socket()函数, 创建套接口, 返回套接口句柄。
- bind()函数,
- listen()函数,
- htonl()和htons()
- accept()
-
-

数据包在应用层称为data, 在TCP层称为segment, 在IP层称为packet, 在数据链路层称为frame.