

潘

C高级：C高级 shell指令 shell脚本

难点：C高级部分，指令比较多，难记。

1. 分文件

【1】分文件

目的：归纳整理功能函数，让函数的调用更加方便，并且能够快速的复用代码。

1) 格式

1. 将主函数（main函数）放在一个.c文件中（1.c）
2. 将功能函数放在其他的.c文件中。例如叫2.c
3. 书写头文件（.h为后缀的一个文件），将函数声明放在头文件中
4. 联合编译，将1.c和2.c一起编译

```
gcc 1.c 2.c -o main
```

2) 头文件

```
gcc -E src -o .iso
```

头文件会在**预处理步骤**展开

i. 书写格式

头文件名字：xxx.h

```
#ifndef 宏名 //如果没有定义过该宏，则该条件为真，防止头文件重复包含
#define 宏名 //定义该宏名
```

头文件

宏定义

结构体定义

不允许定义全局变量；

函数声明；

```
#endif
```

ii. 头文件的调用

```
#include <xxx.h>
#include "路径/xxx.h"           路径是./可以省略不写
```

<> : 到标准库目录下查找头文件: /usr/include
"": 到指定目录下查找头文件, 如果没有找到, 则进入标准库目录下查找

【2】变量的作用域

(1) 局部变量

作用域: 在定义该变量的{}内部

生存周期: 从变量被定义到最近的}位置;

(2) 全局变量

作用域: 整个程序, 可以用extern关键字引用到联合编译的.c文件中

生存周期: 从程序开始到程序结束

extern关键字:

extern int b: 该b变量, 在其他文件定义过; 注意extern只能引用全局变量

```
1 #include <stdio.h>
2 #include "3.h"
3 int b = 10;
4
5 int main(int argc, const char *argv[])
6 {
7     int a = 10;
8     {
9         int a = 20;
10    }
11    print();
12    return 0;
13 }
14
```

```
1 #ifndef __3_H__
2 #define __3_H__
3
4 void print();
5
6 #endif
```

```
1 #include <stdio.h>
2
3 extern int b; //extern:extern后面跟的变量在其他文件定过
4 extern int a; //错误的, a是局部变量
5
6 void print()
7 {
8     printf("b=%d\n", b);
9     printf("a=%d\n", a);
10 }
```

2.c 14,1 全部 3.h 7,6 全部 3.c 9,20-23

```
ubuntu@ubuntu:1_分文件 $ ./a.out
b=10
ubuntu@ubuntu:1_分文件 $ gcc 2.c 3.c
/tmp/ccpKhEdF.o: 在函数'print'中:
3.c:(.text+0x1f): 对'a'未定义的引用
collect2: error: ld returned 1 exit status
ubuntu@ubuntu:1_分文件 $
```

(3) 静态局部变量

static int a;

static 关键字: 可以延长生命周期, 同时限制变量或者函数的作用域;

作用域: 在定义该变量的{}内部, 如果不初始化, 则默认被初始化为0;

生存周期: 程序开始到程序结束

(4) 静态全局变量

作用域：将全局变量的作用域限制在当前的.c文件中

生存周期：程序开始到程序结束

(5) 静态函数

用static修饰的函数，将函数限制在当前.c文件中，不允许其他.c文件调用该函数。

```
1 #include <stdio.h>
2 #include "3.h"
3 int b = 10;
4
5 int func()
6 {
7     int i = 10;
8     i++;
9     printf("i = %d\n", i);
10    return 0;
11 }
12
13 int main(int argc, const char *argv[])
14 {
15     static int a;
16     printf("a = %d\n", a);
17     print();
18     show_b();
19
20
21     return 0;
22 }
```

```
1 #ifndef __3_H__
2 #define __3_H__
3
4 void print();
5 static void show_b();
6
7 #endif
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
1 #include <stdio.h>
2
3 extern int b; //extern:extern后面跟的变量在其他->
4
5 void print()
6 {
7     printf("b=%d\n", b);
8 }
9
10 static void show_b()
11 {
12     printf("b=%d\n", b);
13 }
```

ubuntu@ubuntu:1 分文件 \$ gcc 2.c 3.c
In file included from 2.c:2:0:
3.h:6:13: warning: 'show_b' used but never defined
static void show_b();
~~~~~  
/tmp/cc8Bw0C.o: 在函数'main'中:  
2.c:(.text+0x68): 对'show\_b'未定义的引用  
collect2: error: ld returned 1 exit status  
ubuntu@ubuntu:1 分文件 \$

## 2. 动态分配内存

由于很多情况下，不能直接知道数组容量的大小，该情况下，只能用变量标识数组的容量。

但是在老版本的C中，数组容量只能是常量，而不能是变量。所以需要动态分配内存，要多少申请多少。

### 1) malloc

功能：在堆空间动态分配内存，动态申请内存；

头文件：

```
#include <stdlib.h>
```

原型：

```
void *malloc(size_t size);
```

参数：

**size\_t size:** 指定要申请多少个字节的堆空间；

返回值：

成功，返回堆空间的首地址；

失败，返回NULL；

```
void free(void *ptr);
```

注意：

1. 堆空间手动申请, 手动释放;

2. 如果不释放，会造成内存溢出，内存泄漏等情况。
3. 由于堆空间申请成功会，会返回首地址，我们通过首地址维护堆空间。  
所以堆空间的首地址一定要保存好，如果首地址丢失，则会永远找不到这块堆空间，会造成内存泄漏。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, const char *argv[])
5 {
6     void* ptr = malloc(10);
7     if(NULL == ptr)
8     {
9         printf("malloc failed\n");
10        return -1;
11    }
12
13    char* cptr = (char*)ptr;
14    *cptr = 'a';           //从给定地址开始访问，往后访问char类型个字节
15
16    int* iptr = (int*)ptr;
17    *iptr = 1;           //从给定地址开始访问，往后访问int类型个字节
18
19    int* ptr1 = (int*)malloc(10);
20
21    //指针解引用的本质
22    //告诉指针首地址，解引用，就是从首地址位置开始访问数据类型的大小;
23
24
25    说点什么... | return 0;
26 }
```

t1\_malloc.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char *argv[])
4 {
5     void* ptr = malloc(32);
6     if(NULL == ptr)
7     {
8         printf("malloc failed\n");
9         return -1;
10    }
11
12    int* pint = (int*)ptr;
13    *pint = 10;
14
15    char* pstr = (char*)(pint+1);
16
17    strcpy(pstr, "hello world");
18
19
20
21    return 0;
22 }
```

说点什么... | <

t1\_malloc.c 18,0-

## 2) free

功能：手动释放堆空间；

头文件：

```
#include <stdlib.h>
```

原型：

```
void free(void *ptr);
```

参数：

**void \*ptr**: 填要释放的堆空间的首地址；

## 思考题

1. 有如下代码，由于在定义数组时长度不能用变量，因此上述代码无法编译通过，在不改变程序段功能的情况下，请修改上述代码，改正其错误：

```
unsigned int count = getNum(); //调用函数获得所需缓存个数
char* buf[count];
int i;
for(i=0; i<count; i++)
    buf[i] = (char*)malloc(100); //分配100字节缓存
```

2. 请分析下列代码有什么问题，并修改

```
void GetMemory(char* p)    //line1
{
    p = (char*)malloc(10)   //line2
}

int main(void)
{
    char* str = NULL;       //line3
    GetMemory(str);         //line4
    strcpy(str, "hello world"); //line5
    printf(str);            //line6
    free(str);              //line7
}
```

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

void GetMemory(char** p)    //char** p = &str;
{
    //p其实指向的是str的内存空间
    *p = (char*)malloc(12); //line2
    //上一行运行完毕后，*p从指向NULL，修改成指向堆空间首地址

    printf("%p %d\n", *p, __LINE__);
}
```

```

int main(void)
{
    char* str = NULL;    //line3
    GetMemory(&str);    //line4

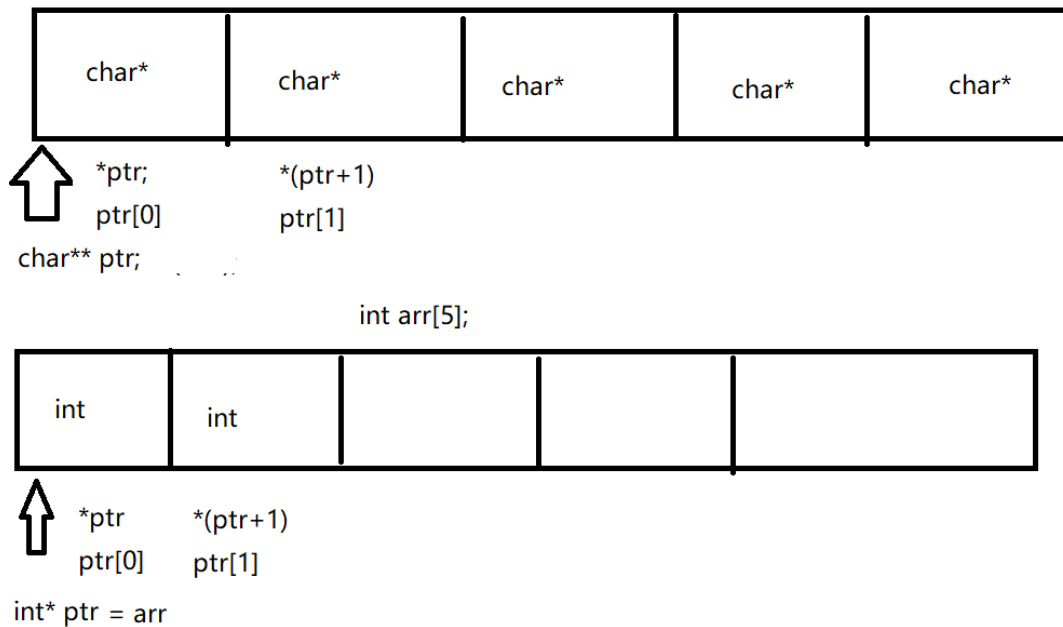
    printf("%d\n", __LINE__);    //打印行号

    printf("%p %d\n", str, __LINE__);
    strcpy(str, "hello world");    //line5

    printf("%s\n", str);    //line6
    free(str);    //line7
}

```

## 第1题



```

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

int main(int argc, const char *argv[])
{
    unsigned int count = 5;

    char** buf = (char**)malloc(count*sizeof(char*));

    int i = 0;
    for(i=0; i<count; i++)
    {
        buf[i] = (char*)malloc(100);
    }

    return 0;
}

```

## 3. GDB调试工具

### 1) 格式

gdb工具使用之前，需要对代码进行编译。gcc编译语句最后加上-g

```
gcc xxx.c -g      生成一个a.out可以调试的二进制可执行程序
```

```
gdb ./二进制可执行程序
```

```
ubuntu@ubuntu:2_malloc $ gcc t3_malloc.c -g
ubuntu@ubuntu:2_malloc $ gdb ./a.out
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...done.
(gdb)
```

### 2) 基本调试命令

#### i. 运行

```
(gdb) r          运行代码
```

#### ii. 打断点

```
breakpoint
(gdb) b 行号      在某一行开始位置打一个断点，运行到该位置会暂停程序
```

#### iii. 查看断点信息

```
(gdb) info b
disable 断点的编号 禁用断点
enable 断点编号 启用断点
delete 断点编号 删除断点
```

#### iv. 打印变量的值

```
(gdb) p 变量名
```

#### v. 单步执行

```
next
(gdb) n
```

## vi.继续执行

```
continue  
(gdb) c
```

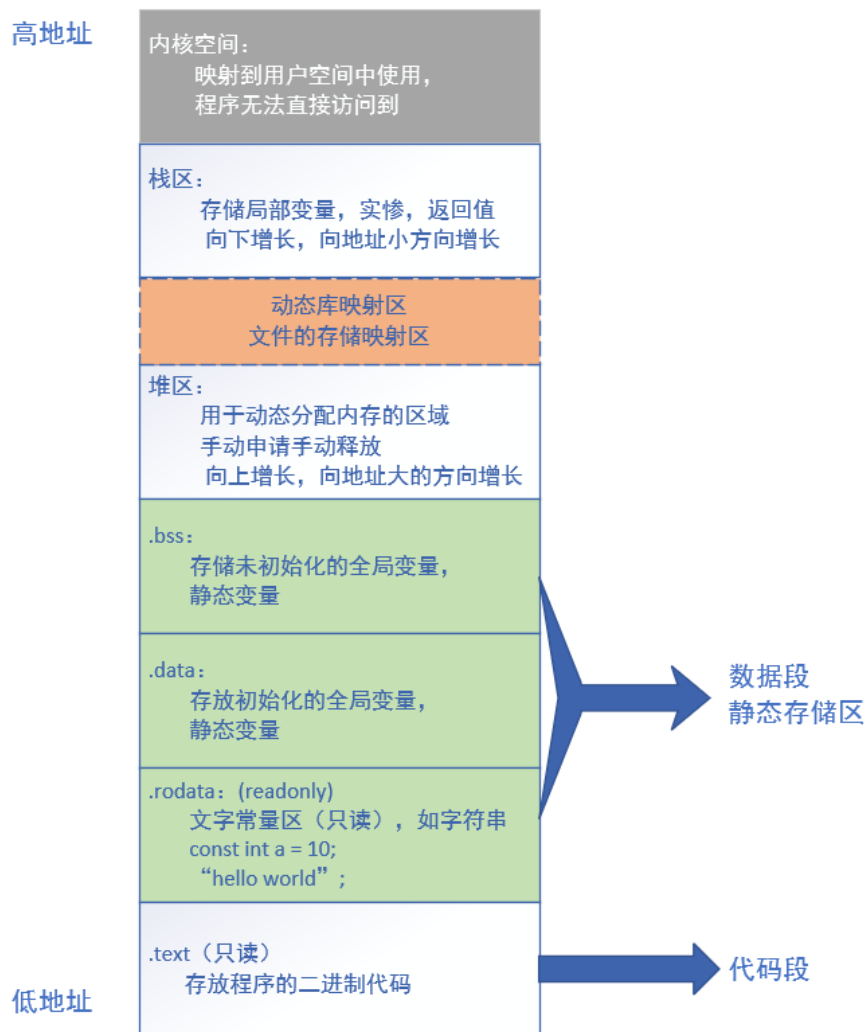
继续执行直到遇到下一个断点，或者程序结束

## vii.退出

```
quit  
(gdb) q
```

# 4. 内存分布图

内存：用户空间+内核空间



堆区和栈区有什么区别？

局部变量，全局变量，静态变量分别存储在哪块空间？

# 5. 结构体

## 【1】结构体定义



## 1) 概念

1. 当现有数据类型不足以满足实际需求的时候，可以自行定义新的数据类型
2. 构造类型，自定义类型。是由一批 **不同类型的数据** 组合而成的一种新的数据类型
3. 组成构造类型的每个数据称之为：成员

## 2) 定义格式以及变量申请

### i. 无名结构体

```
struct
{
    数据类型 成员1;
    数据类型 成员2;
    数据类型 成员n;
};      <---分号结尾
```

由于无名结构体没有名字，所以一般会在定义结构体类型的同时申请变量；

```
//无名结构体
struct
{
    int id;
    char name[20];
    float score;
    char sex;
} a, b, *pa, *pb;           //int a; int b,c, *pa;
```

要求用无名结构体定义一个数组，定义一个指针；

### ii. 有名结构体

```
struct 结构体标识
{
    数据类型 成员1;
    数据类型 成员2;
    数据类型 成员n;
};      <---分号结尾

//有名结构体
struct node      //结构体类型名字: struct node -->int char
{
    int id;
    char name[20];
    float score;
    char sex;
    char* ptr;

} e, *pe ;

struct node c, *pc;           //定义了一个结构体类型变量，变量名c, int c
struct node arr[10];
```

### iii. 重命名

#### (1) typedef关键字

功能：给数据类型重命名

```
typedef 旧的数据类型名 新的数据类型名;          <-----分号结尾

typedef int INT;
int f, *pf;
INT fi, *pfi;    //等价于 int fi

-----

typedef struct node1    //结构体类型名字: struct node1-->int char
{
    int id;
    char name[20];
    float score;
    char sex;
    char* ptr;

} Node1, No, Nod;    //结构名: struct node1 , Node1 ,No , Nod

struct node1 j;
Node1 j1

-----

typedef struct
{
    int id;
    char name[20];
    float score;
    char sex;
    char* ptr;

} Noname;
Noname s;
```

### 3) 结构体的访问

#### i. 普通变量访问成员：.

```
变量名.成员名;
#include <stdio.h>
#include <string.h>

//有名结构体
struct node    //结构体类型名字: struct node -->int char
{
    int id;
    char name[20];
    float score;
    char sex;
    char* ptr;
} ;
```

```

int main(int argc, const char *argv[])
{
    struct node d;
    d.id = 1;
    //d.name = "hello";    //name是数组首地址，是常量不能作为左值
    strcpy(d.name, "hello");
    d.score = 3.14;
    d.sex = 'a';
    d.ptr = NULL;

    printf("%d %s\n", d.id, d.name);

    return 0;
}

```

## ii. 指针变量访问成员: ->

指针变量名->成员变量名;

```

//有名结构体
struct node    //结构体类型名字: struct node --> int char
{
    int id;
    char name[20];
    float score;
    char sex;
    char* ptr;
} ;

struct node* pd = &d;
printf("%d %s %g %c %p\n", pd->id, pd->name, pd->score, pd->sex, pd->ptr);

```

## 思考题

1. 有如下结构体类型定义

```

typedef struct
{
    short with;
    short lenth;
}base;

typedef struct
{
    base baseinfo;
    short height;
}WinInfo;

typedef struct

```

```
{
    base* baseinfo;
    short height;
}BoxInfo;
```

系统中有如下三个全局变量，且已经为这些变量分配内存，类型声明为

```
extern WinInfo tmpInfo1;    extern BoxInfo tmpInfo2;    extern BoxInfo* tmpInfo3;
```

请根据需求写出赋值语句：

- (1) 为tmpInfo1的域height赋值为1: tmpInfo1.height=1;
- (2) 为tmpInfo1的域with赋值为2: tmpInfo1.baseinfo.with=2;
- (3) 为tmpInfo2的域height赋值为1: tmpInfo2.height=1;
- (4) 为tmpInfo2的域with赋值为2: tmpInfo2.baseinfo->with=2;
- (5) 为tmpInfo3的域height赋值为1: tmpInfo3->height=1;
- (6) 为tmpInfo3的域with赋值为1: tmpInfo3->baseinfo->with=2;

## 4) 初始化

定义结构体变量的同时赋值，称之为初始化

```
struct node
{
    int id;
    char name[20];
    int score[3];
    char sex;
};
```

初始化的时候用{}将所有成员的初始化数据包含起来，并且数据的顺序需要与成员类型一一匹配

```
struct node stu = {1, "zhangsan", {90,30,20}, 'F'};
struct node stu = {1, {'z', 'h'}, {90,30,20}, 'F'};
struct node stu = {1, "zhangsan", {0}, 'F'}; //全部初始化，每个成员都赋初始值
```

```
struct node stu3 = {2}; //部分初始化，初始化前面部分，后面默认初始化为0
```

```
struct node stu4 = { "wangwu"}; //错误的，“wangwu”是char*类型，而第一个成员是int类型
struct node stu4 = { , "wangwu"}; //错误的，部分初始化，不能省略前面部分
```

结构体数组的初始化

```
struct node stu5[3] = { {1,"zs",{10,11,12}, 'F'} , {2, "ls", {20,21,22}, 'M'} , {3, "ww", {30,31,32}, 'F'} };
struct node stu6[3] = {
    {1,"zs",{10,11,12}, 'F'} ,
    {2, "ls", {20,21,22}, 'M'} ,
```

```

    {3, "ww", {30,31,32}, 'F'}
};
结构体数组遍历访问成员
for(i=0; i<3; i++)
{
    printf("%d %s\n", stu5[i].id, stu5[i].name);
    printf("%d %s\n", (stu5+i)->id, (stu5+i)->name );
    printf("%d %s\n", (*(stu5+i)).id, (*(stu5+i)).name);
}

////////////////////////////////////
用其他结构体变量给当前结构体变量初始化;
int a = 10;
int b = a;

struct node stu7 = {2, "zhaoliu"};
struct node stu8 = stu7;

```

## 【2】结构体变量的大小

### 1) 结构体变量大小计算

```

sizeof(结构体变量名);
sizeof(结构体类型名);

#include <stdio.h>
struct node
{
    int a;
    char b;
    short c;
};

int main(int argc, const char *argv[])
{
    int a = 10;
    printf("%ld\n", sizeof(a));           //4
    printf("%ld\n", sizeof(int));         //4

    struct node s;
    printf("%ld\n", sizeof(s));           //8
    printf("%ld\n", sizeof(struct node)); //8

    return 0;
}

```

## 2) 结构体对齐

1. 结构体的大小并不是简单的成员大小相加。
2. 为了加速cpu的取值周期，会将结构体进行对齐，一次取n个字节。  
一次读取1个字节，会导致cpu取值周期拉长，但是如果cpu一次取4个字节，会让效率大大提高。
3. 不同的操作系统有自己默认的对齐系数;  
64OS默认：8个字节，8个字节取一次  
32OS默认：4个字节，4个字节取一次

### i. 默认对齐值

1. 结构体第一个成员偏移量是0，其他成员首地址在 **自身对齐值** 的整数倍位置上;  
自身对齐值：每个成员都有自身对齐值；首成员不偏移
2. 成员自身对齐值 = 操作系统默认对齐值 与 成员所占的字节数 对比，**取小的那个**作为自身对齐值;  
两个对齐值中选择小的那个作为成员的自身对齐值。  
例如64OS:8bytes, 成员是int类型，则是4bytes，由于4<8, 所以成员的自身对齐值为4。  
即该成员的首地址距离结构体首地址为：4的倍数

```
struct node
{
    int a;        //4byte, 4<8, 所以a的自身对齐值为4
    char b;       //1byte, 1<8, 所以b的自身对齐值为1
                //1byte 空出一个字节, 使c变量的首地址在2的倍数
    short c;      //2byte, 2<8, 所以c的自身对齐值为2
};
```

3. 结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员**最大对齐值** 比较后，取较小值的整数倍;

| 操作系统对齐系数 | 结构体所有成员中最大对齐系数 | 结构体大小 |
|----------|----------------|-------|
| 8        | 4              | 4的倍数  |
| 4        | 8              | 4的倍数  |
| 4        | 1              | 1的倍数  |

**注意：字符数组的对齐值为1，因为字符数组是由单个字节组成的。**

```
struct i
{
    char name[18]; //因为char数组是由单个char类型组成, 所以对齐值为1,
};
//结构体大小为1的倍数, 所以是18
```

```

1 #include <stdio.h>
2
3 //64位操作系统默认对齐值是:8
4
5 struct node
6 {
7     int a;      //4byte, 4<8,所以a的自身对齐值为4
8     char b;     //1byte, 1<8, 所以b的自身对齐值为1
9                //1byte 空出一个字节, 使c变量的首地址在2的倍数上
10    short c;    //2byte, 2<8, 所以c的自身对齐值为2
11 };
12 //结构体变量的总大小 = 8byte 与 4 比较后, 取较小值 4 的整数倍
13 //结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;
14
15
16 struct m
17 {
18     char b;     //1byte, 首成员不需要偏移
19                //3byte 空出3byte, 让a首地址在4的倍数上
20     int a;      //4byte, 4<8, 所以对齐值为4, 首地址在4的倍数上
21     short c;    //2byte, 2<8, 所以c的自身对齐值为2
22                //2byte, 结构体大小是4的整数倍
23 };
24
25 //结构体变量的总大小 = 8byte 与 4 比较后, 取较小值 4 的整数倍
26 //结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;
27

```

```

#include <stdio.h>

//64位操作系统默认对齐值是:8

struct node
{
    int a;      //4byte, 4<8,所以a的自身对齐值为4
    char b;     //1byte, 1<8, 所以b的自身对齐值为1
                //1byte 空出一个字节, 使c变量的首地址在2的倍数上
    short c;    //2byte, 2<8, 所以c的自身对齐值为2
};

//结构体变量的总大小 = 8byte 与 4 比较后, 取较小值 4 的整数倍
//结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;

struct m
{
    char b;     //1byte, 首成员不需要偏移
                //3byte 空出3byte, 让a首地址在4的倍数上
    int a;      //4byte, 4<8, 所以对齐值为4, 首地址在4的倍数上
    short c;    //2byte, 2<8, 所以c的自身对齐值为2
                //2byte, 结构体大小是4的整数倍
};

//结构体变量的总大小 = 8byte 与 4 比较后, 取较小值 4 的整数倍
//结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;

int main(int argc, const char *argv[])
{
    int a = 10;
    printf("%ld\n", sizeof(a));          //4
    printf("%ld\n", sizeof(int));        //4

    struct node s;
    printf("%ld\n", sizeof(s));           //8
    printf("%ld\n", sizeof(struct node)); //8
}

```

```

printf("%ld\n", sizeof(struct m)); //12
struct m text;
printf("%p %p %p\n", &text.b, &text.a, &text.c); //0x7ffd4615857c
0x7ffd46158580 0x7ffd46158584

return 0;
}

```

## ii. 指定对齐值

去修改编译的对齐值;

```

#pragma pack(value) //修改编译器的对齐值 64:8byte 32:4byte

//该范围是修改后的编译器对齐值的有效范围

#pragma pack() //重置编译器的对齐值为默认对齐值

```

**value : 2^n (n=0,1,2,3,4,5....) , 即value = 1, 2, 4, 8, 16,32 .....**

```

5 //修改编译器的对齐值
6 #pragma pack(2) //修改操作系统对齐值为2 , 若改成4, 结构体的大小分别为?
7
8 struct node
9 {
10     int a; //4byte, 4>2,所以a的自身对齐值为2
11     char b; //1byte, 1<2, 所以b的自身对齐值为1
12     //1byte 空出一个字节, 使c变量的首地址在2的倍数上
13     short c; //2byte, 2==2, 所以c的自身对齐值为2
14 };
15 //结构体变量的总大小 = 2byte 与 4 比较后, 取较小值 2 的整数倍 = 8
16 //结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;
17
18
19 struct m
20 {
21     char b; //1byte, 首成员不需要偏移, 1<2所以对齐值为1
22     //1byte
23     int a; //4byte, 4>2, 所以对齐值为2, 首地址在2的倍数上
24     short c; //2byte, 2==2, 所以c的自身对齐值为2
25 };
26
27 //结构体变量的总大小 = 2byte 与 4 比较后, 取较小值 2 的整数倍 = 8
28 //结构体变量的总大小 = 操作系统对齐值 与 结构体中所有成员最大对齐值 比较后, 取较小值的整数倍;
29
30
31 struct u
32 {
33     int a; //4
34     double b; //8, 8>2, 所以对齐值为2
35     char c; //1, 1<2, 所以对齐值为1,
36     //1
37     short d; //2, 2==2, 所以对齐值为2, d的首地址要在2的倍数上
38 };
39 //结构体大小为2的倍数, 所以是16

```

## iii. 自身对齐值

数据类型本身的对齐值, 例如char类型自身对齐值默认1, short类型默认就是2

```

//修改编译器的对齐值
#pragma pack(1) //修改操作系统对齐值为1
struct node
{
    int a;
    char b;
}

```



```

    short c;
};
#pragma pack() //重置为默认的对齐值:8bytes

struct n
{
    int a;
    char b;

    short c;
} __attribute__((packed));
//取消结构体对齐, 在编译过程中按照实际占用的字节数进行申请内存, 并按照实际字节数进行对齐;
//GCC编译器语法特有的。

```

### 3) 嵌套结构体的大小

内层结构体类型的对齐值为: 该内层结构体所有成员中最大对齐值;

```

#include <stdio.h>

typedef struct
{
    int a;        //4
    short b;      //2, 2<8, 所以2个字节一对齐
    char c;       //1, 1<8, 所以1个字节一对齐
                //1
}__t;

//总大小: 8

typedef struct msg
{
    char a;       //1, 1<8所以自身对齐值是1,
                //3
    __t b;        //8, 由于b是结构体, b的对齐值为所有成员中最大的对齐值4, 4<8(64位OS), 所以该
                //位置的自身对齐值为4
    short c;      //2, 2<8, 所以自身对齐值为2
                //2
}__y;
//总大小为16

typedef struct
{
    short b;      //2
                //6
    double c;     //8
    int a;        //4
                //4
}__i;
//总大小为: 24

typedef struct
{
    char a;       //1

```

```

        //7
__i b;    //24, 对齐值为8

short c;  //2
          //6
}__o;
//总大小为: 40

int main(int argc, const char *argv[])
{
    printf("%ld\n", sizeof(__t));
    printf("%ld\n", sizeof(__y));

    printf("%ld\n", sizeof(__i));
    printf("%ld\n", sizeof(__o));
    return 0;
}

```

## 6. 联合体（共用体）

功能：

1. 节省内存空间，当两个比较大的数据不需要同时使用的时候，可以让这两个数据共用一块内存空间。
2. 方便数据维护；

联合体：

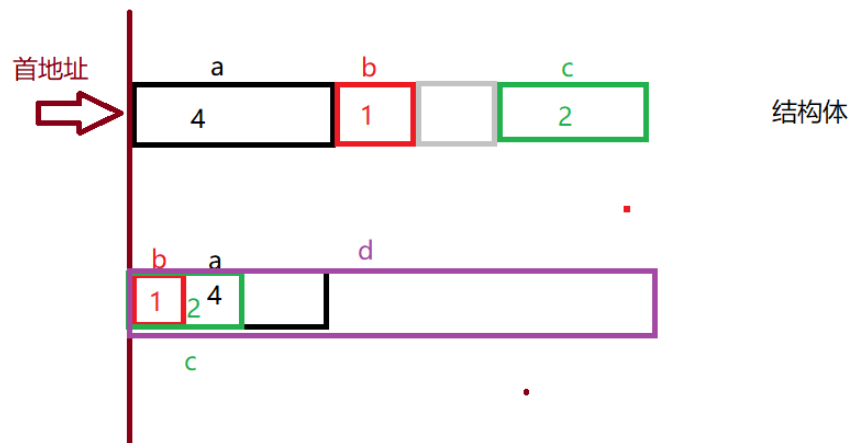
1. 联合体的所有成员**共用一块内存空间**，且成员的**首地址都相同**
2. 由于共用体，共用一块内存空间，所以所有成员变量的内容会相互覆盖。同一时间只有一个数据是有效的。

结构体：

```

int a;
char b;
short c;

```



### 1) 格式

1. 格式与结构体相似，只需要将struct关键字，修改成union关键字即可。同样也有有名共用体，无名共用体，重命名共用体。
2. 访问方式也与结构体类似，**普通联合体变量，用 . ； 指针类型联合体变量，用 ->**

**union** 联合体标识

```
{
    数据类型 成员1;
    数据类型 成员2;
    数据类型 成员n;
};      <---分号结尾
```

```
1 #include <stdio.h>
2
3 //有名联合体
4 union msg      //union msg是联合体的类型
5 {
6     int a;
7     char b;
8     short d;
9     char c[17];
10 };
11
12 int main(int argc, const char *argv[])
13 {
14     union msg u;
15
16     printf("%p, %p, %p\n", &u.a, &u.b, u.c);    //0x7ffeb0f43110, 0x7ffeb0f431
17
18     u.a = 0x12345678;    //0x12存储在一个字节中 0x34存储一个字节 0x56 0x78
19
20     //由于共用体公用一块内存空间,所以我想直到0x78存在低地址还是高地址
21     //只需要通过u.b去访问首地址即可
22     printf("u.b= %x\n", u.b);    //0x78
23     printf("u.d = %x\n", u.d);    //0x5678
24
25     union msg* pu = &u;
26     printf("a=%x %x\n", pu->a, u.a);
27     printf("b=%x %x\n", pu->b, u.b);
28     printf("d=%x %x\n", pu->d, u.d);
29
30     int i = 0;
31     for(i=0; i<17; i++)
32     {
33         printf("[%d] = %x\n", i, (pu->c)[i]);
34     }
35
8_union.c                                     21,37-30 顶端
```

```
b=0x78 0x78
ubuntu@ubuntu:3_struct $ gcc 8_union.c
ubuntu@ubuntu:3_struct $ ./a.out
0x7ffc7daae7f0, 0x7ffc7daae7f0, 0x7ffc7daae7f0
u.b= 0x78
u.d = 0x5678
a=0x12345678 0x12345678
b=0x78 0x78
d=0x5678 0x5678
ubuntu@ubuntu:3_struct $ gcc 8_union.c
ubuntu@ubuntu:3_struct $ ./a.out
0x7ffc4d5e8a0, 0x7ffc4d5e8a0, 0x7ffc4d5e8a0
u.b= 0x78
u.d = 0x5678
a=0x12345678 0x12345678
b=0x78 0x78
d=0x5678 0x5678
[0] = 0x78
[1] = 0x56
[2] = 0x34
[3] = 0x12
[4] = 0x7a
[5] = 0x55
[6] = 0
[7] = 0
[8] = 0xffffffffa0
[9] = 0xffffffff95
[10] = 0x7
[11] = 0xfffffffff9
[12] = 0x7a
[13] = 0x55
[14] = 0
[15] = 0
[16] = 0xffffffffa0
ubuntu@ubuntu:3_struct $
```

## 2) 大小

sizeof(联合体变量名);

sizeof(联合体类型名);

1. 先找到联合体中占用最大字节数的成员;
2. 找到联合体的对齐系数: 所有成员中最大的对齐系数 与 操作系统默认对齐系数中, **较小值**的整数倍

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

```
typedef union
```

```
{
    int a;           //4, 4<8  所以对齐系数是4
    char b;          //1, 1<8, 所以对齐系数是1
    char c[18];      //1, 1<8, 所以对齐系数是1
}__t;
```

//联合体成员中最大的对齐系数是4 < 8(640s), 所以大小是4的倍数: 20

```
typedef union
```

```
{
    int a;           //4, 4<8  所以对齐系数是4
    double b;        //8, 8<8, 所以对齐系数是8
    char c[18];      //1, 1<8, 所以对齐系数是1
}__t;
```

//联合体成员中最大的对齐系数是8 < 8(640s), 所以大小是8的倍数: 24

完成学生教师管理系统的增，查功能，用数组存储数据即可；数组容量定义为50；

查：1.只查看学生的信息，2.只查看老师的信息，3.查看所有人信息；

```
1 #include <stdio.h>
2 union info
3 {
4     float score;
5     char jd[20];
6 };
7
8 struct msg
9 {
10     int id;
11     char name[20];
12     char flag;
13     union info ino;
14 };
15
16
17 int main(int argc, const char *argv[])
18 {
19     struct msg message[3];
20     message[0].id = 1;
21     strcpy(message[0].name, "zhangsan");
22     message[0].flag = 's';
23     message[0].ino.score = 90;
24
25     message[1].id = 1;
26     strcpy(message[1].name, "lisi");
27     message[1].flag = 't';
28     strcpy(message[1].ino.jd, "数学老师");
29
30     return 0;
31 }
```

9\_union.c

28,38

## 1. main.c

```
#include "../func.h"

int main(int argc, const char *argv[])
{
    struct msg message[50];
    int pos = -1;           //最后一个有效数据的下标

    char choose = 0;
    while(1)
    {
        printf("-----\n");
        printf("-----1.增加-----\n");
        printf("-----2.学生信息-----\n");
        printf("-----3.教师信息-----\n");
        printf("-----4.所有信息-----\n");
        printf("-----5.退出-----\n");
```

```

printf("-----\n");

printf("请输入>>>");
choose = getchar();
while(getchar() != '\n');    //循环吸收垃圾字符

switch(choose)
{
case '1':
    do_addMsg(message, &pos);
    break;
case '2':
    //do_showStu();
    break;
case '3':
    //do_showTeach();
    break;
case '4':
    //do_showAll();
    break;
case '5':
    goto END;
    break;
default:
    printf("输入错误, 请重新输入\n");
}

}

END:
    return 0;
}

```

## 2. func.c

```

#include "./func.h"

//添加一个学生或者老师的信息
int do_addMsg(struct msg* pm, int* ppos)
{
    //数组是否满了
    if(49 == *ppos)
    {
        printf("数组满了, 添加失败\n");
        return -1;
    }

    printf("请问要输入的是学生, 还是老师\n");
    printf("-----1. 学生-----\n");
    printf("-----2. 教师-----\n");
    printf("请输入>>>");
    char choose = 0;
    choose = getchar();
    while(getchar() != '\n');    //循环吸收垃圾字符

```

```

(*ppos)+=1;    //将下标偏移，有效数据存储在pos位置

//输入id
printf("请输入id>>>");
scanf("%d", &(pm[*ppos].id));
while(getchar() != '\n');    //循环吸收垃圾字符

//输入名字
printf("请输入name>>>");
scanf("%s", pm[*ppos].name);
while(getchar() != '\n');    //循环吸收垃圾字符

if('1' == choose)
{
    //是学生flag == 's'
    pm[*ppos].flag = 's';

    printf("请输入学生成绩>>>");
    scanf("%f", &(pm[*ppos].ino.score));
    while(getchar() != '\n');    //循环吸收垃圾字符
}
else if('2' == choose)
{
    //是老师flag == 't'
    pm[*ppos].flag = 't';
    printf("请输入教师岗位>>>");
    scanf("%s", pm[*ppos].ino.jd);
    while(getchar() != '\n');    //循环吸收垃圾字符
}
else
{
    printf("输入错误\n");
    return -1;
}

return 0;
}

```

### 3. func.h

```

#ifndef __FUNC_H__
#define __FUNC_H__
#include <stdio.h>

union info
{
    float score;
    char jd[20];
};

struct msg
{
    int id;
    char name[20];
    char flag;
}

```

```
union info ino;
};

int do_addMsg(struct msg* pm, int* ppos);

#endif
```

## 7. shell指令

```
man shell指令
shell命令 --help
```

### 【1】软件安装命令

#### a. 离线安装

##### 1) 离线安装包介绍

| ubuntu  | xxx.deb |
|---------|---------|
| windows | xxx.exe |
| readhat | xxx.rmp |

ubuntu离线软件包的格式：

|      |      |      |              |      |
|------|------|------|--------------|------|
| sl _ | 3.03 | - 17 | build2_amd64 | .deb |
|      |      |      |              |      |
| 软件名  | 版本号  | 修订号  | 架构 (arm64)   |      |

##### 2) 安装离线包 (dpkg)

dpkg：管理系统中的deb离线包，可以对其进行安装，卸载，deb打包，deb解压等操作。

如果软件有依赖别的程序，使用该指令**不会安装其他依赖程序**：

```
dpkg: 处理软件包 sl (--install)时出错：
依赖关系问题 - 仍未被配置
正在处理用于 man-db (2.10.2-1) 的触发器 ...
在处理时有错误发生：
sl
```

```
sudo dpkg [参数] [软件包名/软件名]
```

[参数]

| -i 软件包的名字 | 安装软件包 install          |
|-----------|------------------------|
| -l 软件名    | 查看软件版本 list            |
| -L 软件名    | 显示软件关联的文件              |
| -r 软件名    | 卸载软件，配置文件依然存在; remove, |
| -P 软件名    | 卸载软件，同时删除其配置文件         |

想要彻底卸载软件，可以先使用 -r 卸载，用 -P 删除配置文件

#### 安装步骤：

1. 如果离线包在 windows 中，需要将包移动到 ubuntu 中。（可以选择用拖动方式，用共享文件夹方式）
2. 注意安装不要放在共享文件夹安装；
3. 执行离线安装指令

```
sudo dpkg -i sl_3.03-17build2_amd64.deb
```

安装完毕后，终端输入软件名，按下回车运行；

## b. 在线安装

### 1) 在线安装 apt-get

注意：中间没有空格

apt-get：

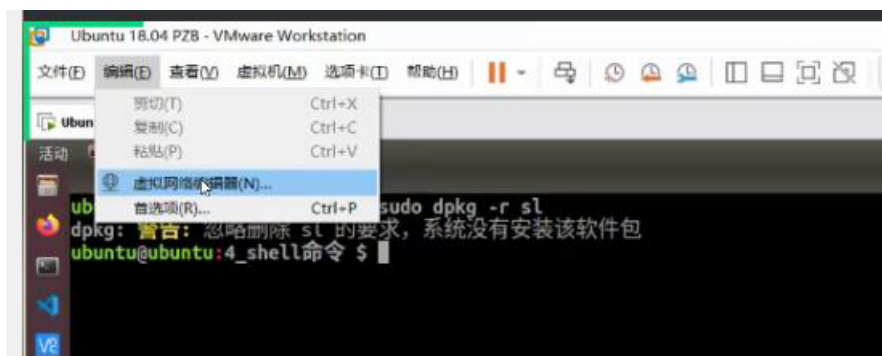
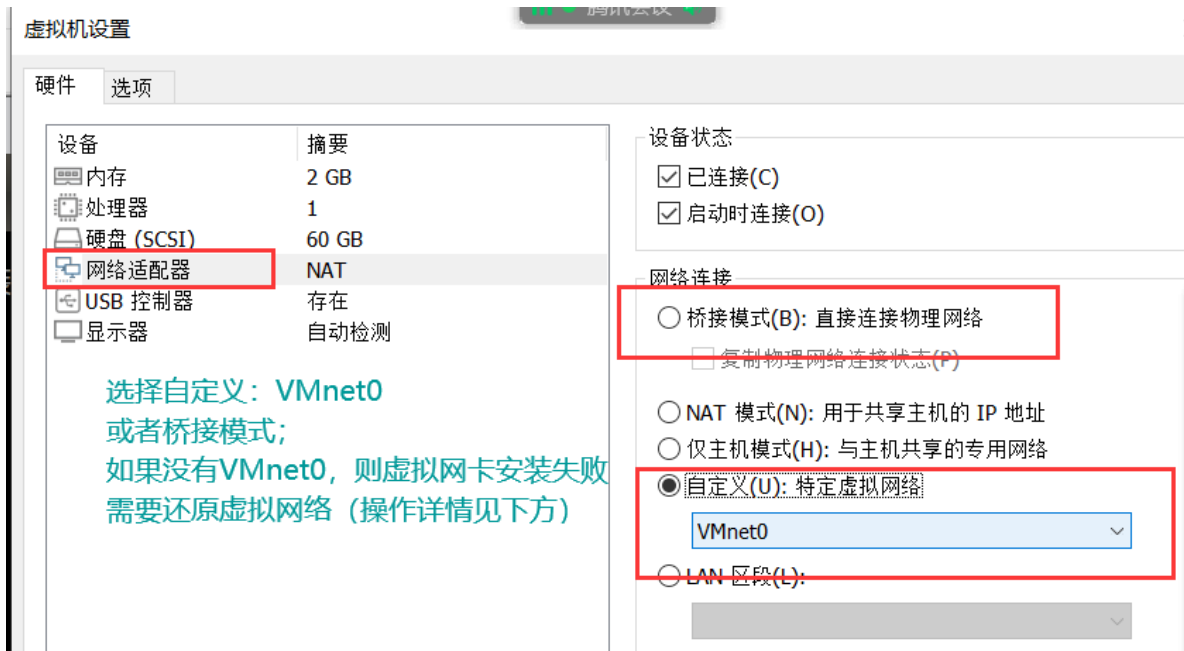
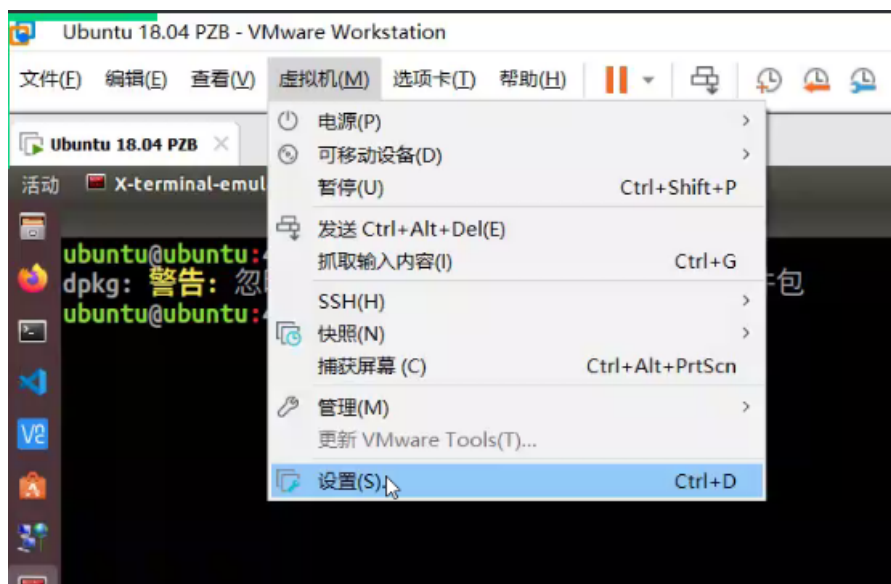
1. 适用于软件包以 .deb 结尾的操作系统；
2. 从互联网仓库中搜索、安装、升级、卸载软件，所以需要**保证操作系统联网**。
3. 如果安装的软件有依赖其他程序，则会将这些依赖程序一起安装了；

### 2) 联网操作

#### i. 桥接

会给虚拟机也分配一个 ip 地址



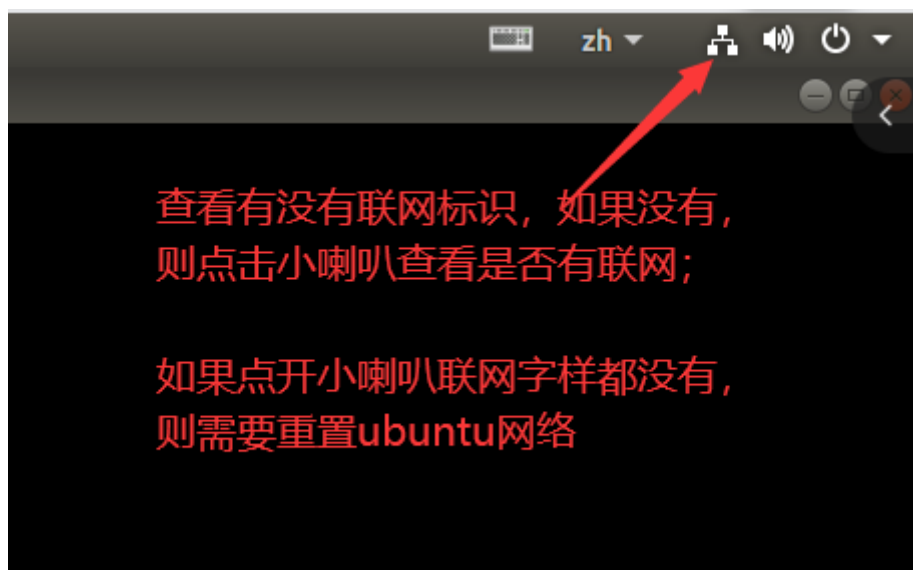




查看网卡名字：找到网络和Internet设置---->更改网络适配器---->找到电脑连上的网络对应的网卡名

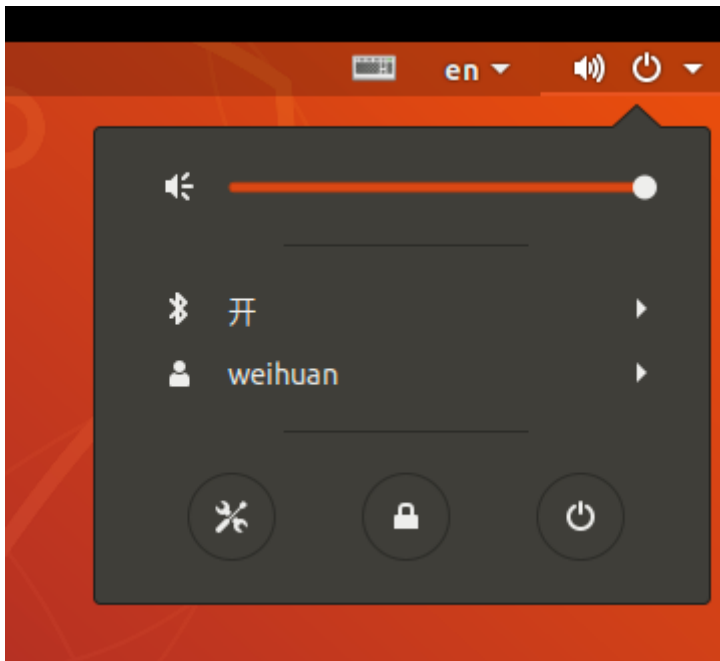


如果没VMnet0, 则需要还原默认设置:



例如下方这个现象, 就需要重置ubuntu的网络

```
sudo service network-manager stop
sudo rm /var/lib/NetworkManager/NetworkManager.state
sudo service network-manager start
```



配置完毕后，终端输入

```
ping baidu.com
```

出现以下现象则联网成功；

```
ubuntu@ubuntu:4_shell命令 $ ping baidu.com
```

```
PING baidu.com (220.181.38.251) 56(84) bytes of data.
```

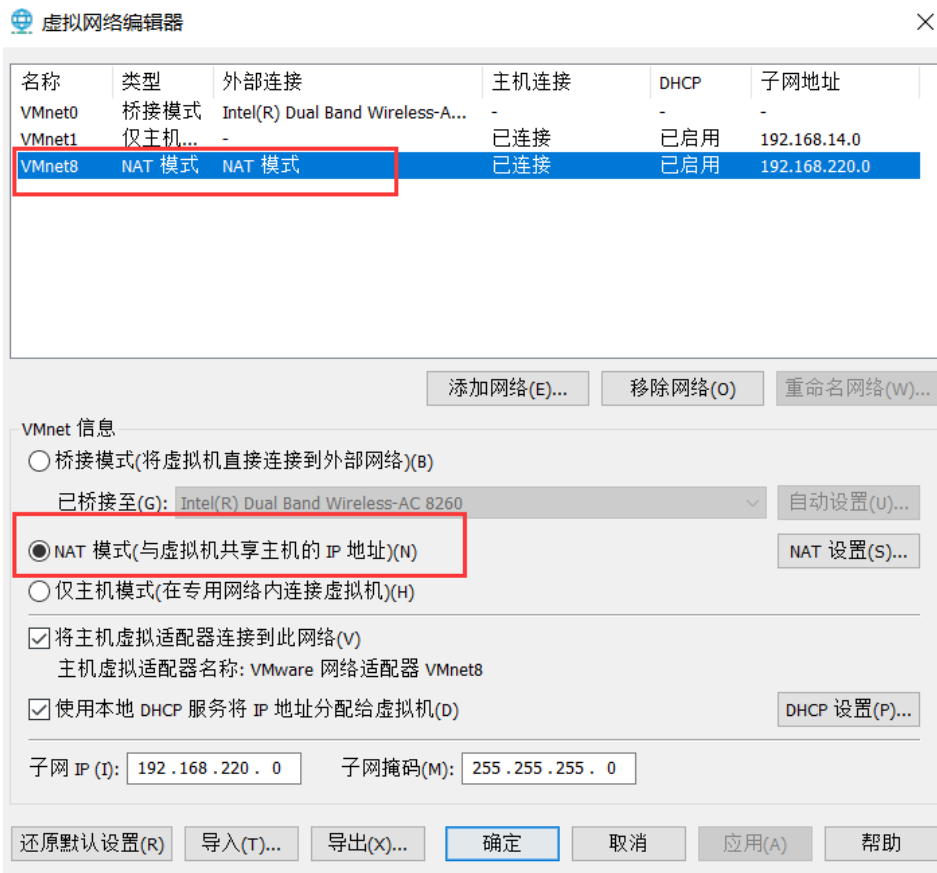
```
64 bytes from 220.181.38.251 (220.181.38.251): icmp_seq=1 ttl=128 time=30.1 ms
```

```
64 bytes from 220.181.38.251 (220.181.38.251): icmp_seq=2 ttl=128 time=29.7 ms
```

```
64 bytes from 220.181.38.251 (220.181.38.251): icmp_seq=3 ttl=128 time=30.2 ms
```

ctrl+c按键，退出

**ii.net模式**



### 3) 更新源

#### 更新成阿里源

<https://www.csdn.net/tags/OtDaMg4sNTkyOTktYmxvZw0000000000.html>

更新源步骤

```
sudo cp /etc/apt/sources.list /etc/apt/sources.list.save  
sudo vim /etc/apt/sources.list
```

删除文件中原有内容：将光标停在首行，按下dG

将原本的源文件进行备份  
打开源列表文件，进行修改

将搜索到的ubuntu对应版本的阿里源连接粘贴到/etc/apt/sources.list文件中

```
deb http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe multiverse  
deb http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted universe  
multiverse  
deb http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted universe  
multiverse  
deb http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted universe  
multiverse  
deb http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted universe  
multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe  
multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted  
universe multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted  
universe multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted  
universe multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted  
universe multiverse
```

保存退出source.list文件，执行

```
sudo apt-get update
```

刷新软件索引；

这个命令，会访问 源列表 里面的每个网址，并读取软件列表，然后将软件索引保存在本地电脑；**不会对本机软件做任何升级以及安装**

索引会保存在： `/var/lib/apt/lists`

```
sudo apt-get upgrade
```

升级本机所有软件

这个命令会升级本机的所有软件，不建议大家使用，耗时比较久

## 4) apt-get指令

```
sudo apt-get [参数] [软件名]
```

**[参数]**

|                                 |                                    |
|---------------------------------|------------------------------------|
| sudo apt-get install 软件名        | 安装软件，默认存储在/var/cache/apt/archives/ |
| sudo apt-get remove 软件名         | 卸载软件，配置文件依然存在                      |
| sudo apt-get remove 软件名 --purge | 卸载软件，同时删除配置文件                      |

|                         |                                         |
|-------------------------|-----------------------------------------|
| sudo apt-get clean      | 删除软件安装包，删除/var/cache/apt/archives/下的安装包 |
| sudo apt download 软件名   | 下载安装包，但是不安装                             |
| sudo apt-get source 软件名 | 下载软件的源码                                 |

i. sudo apt-get install

```
sudo apt-get install sl          在线安装跑火车软件
sudo apt-get install cmatrix    安装代码雨 cmatrix
sudo apt-get install bastet     俄罗斯方块
sudo apt-get install oneko      猫          运行 oneko & 运行在后端
jobs指令，查看后端所有运行的程序 --> [1]+ 运行中                oneko &
fg 编号                          --> fg 1          将1号后台任务运行到前台

sudo apt-get install lolcat     渐变色
ls -l ~ | lolcat

sudo apt-get install cowsay
ls ~ | cowsay | lolcat
cowsay -l    查看所有动物
ls ~ | cowsay -f turkey | lolcat

sudo apt-get install terminator  安装终结者终端
ctrl + alt + t    打开
ctrl + d          关闭
ctrl + shift + q  关闭所有
ctrl + shift + e  左右分屏
ctrl + shift + o  上下分屏
```

安装完毕后，终端输入软件名，按下回车运行;

下载后的软件包，默认存储在/var/cache/apt/archives/

5) 管道符: |

| : 连接两个指令，将前一个的结果当做后一个参数;

【2】 压缩和打包命令

## 1) 压缩、解压 ---对文件操作

注意：是对文件操作，**不能对目录操作**

### 压缩指令

| 指令        | 压缩后格式   |
|-----------|---------|
| gzip 文件名  | xxx.gz  |
| bzip2 文件名 | xxx.bz2 |
| xz 文件名    | xxx.xz  |

终端输入：

```
for i in {1..1000000}; do echo "aaaaaaaaa$i">>file1; done  
创建一个16M文件： 1s -lh可以查看
```

1. 上述指令从上往下，压缩率越来越高，但是压缩速度会越来越慢
2. 压缩指令默认删除源文件，只留下压缩后的文件

```
-rw-rw-r-- 1 ubuntu ubuntu 16M 七月 12 14:44 file  
-rw-rw-r-- 1 ubuntu ubuntu 2.3M 七月 12 14:44 file1.gz  
-rw-rw-r-- 1 ubuntu ubuntu 1.1M 七月 12 14:43 file2.bz2  
-rw-rw-r-- 1 ubuntu ubuntu 165K 七月 12 14:44 file3.xz
```

### 解压指令

|                      |
|----------------------|
| <b>gunzip xxx.gz</b> |
| bunzip2 xxx.bz2      |
| unxz xxx.xz          |

## 2) 打包、拆包 --对目录操作

打包：将目录归档成一个文件，（将目录变成文件）

拆包：将归档的文件重新释放成一个目录；

注意：

1. 打包拆包 与 压缩解压不同
2. 打包后目录会归档成文件，就可以使用压缩指令将打包后的文件进行压缩;

### i) 归档

#### 打包指令的格式

```
tar [参数] [要生成的包名] [要打包的目录]
```

#### 拆包指定的格式



tar [参数] [需要拆包的文件]

| [参数] |                                                                |
|------|----------------------------------------------------------------|
| -c   | 打包, 不压缩, 生成的文件名格式: xxx.tar=                                    |
| -t   | 查看打包文件中的内容 list                                                |
| -v   | 显示打包拆包信息; 该选项可以不加                                              |
| -f   | 指定要打包拆包的文件, 必须要加的, 并且必须放在参数结尾 file -cvf -vcf -cf -xf -xvf -vxf |
| -x   | 拆包 extrcat                                                     |

打包:

```
tar -cvf file4.tar file4
```

```
-----  
drwxrwxr-x 2 ubuntu ubuntu 4096 七月 12 15:00 file4  
-rw-rw-r-- 1 ubuntu ubuntu 31784960 七月 12 15:01 file4.tar
```

拆包

```
tar -xvf file4.tar
```

为什么文件夹的大小都是4096byte?

在Linux中读取磁盘是一块一块读取的, 而不是一个字节一个字节读取;

磁盘最小的划分叫做扇区, 一个扇区是512个bytes, 计算机一次读取8个扇区: 4096个bytes

目录文件的大小, 始终是4096的倍数

## ii) 归档同时解压缩

|    |                                                |
|----|------------------------------------------------|
| -z | 将归档后的文件压缩成gzip格式, 生成的包 xxx.tar.gz, 也可以用该选项解压   |
| -j | 将归档后的文件压缩成bzip2格式, 生成的包 xxx.tar.bz2, 也可以用该选项解压 |
| -J | 将归档后的文件压缩成xz格式, 生成的包 xxx.tar.xz, 也可以用该选项解压     |

上述参数, 当与打包指令一起使用的时候, 是压缩指令

当与拆包指令使用的时候, 是解压指令;

打包并压缩:

```
tar -cvzf file.tar.gz file
```

拆包并解压

```
tar -xvzf file.tar.gz
```

打包并压缩:

```
tar -cvjf file.tar.bz2 file
```

拆包并解压

```
tar -xvjf file.tar.bz2
```

打包并压缩:

```
tar -cvJf file.tar.xz file
```

拆包并解压

```
tar -xvjf file.tar.xz
```

### iii) 万能解压指令

```
tar -xvf file.tar.xxx
```

### iv) 指定解压路径

```
tar -xvf file4.2.tar.xxx -C 指定路径
```

## 【3】文件操作指令

---

### 1) Linux操作系统文件架构

linux操作系统：万物皆文件，且是一个倒插树的形象。

每一个文件夹中有特定的功能

# Linux根目录

- bin — 二进制目录文件，存储了常用的linux用户命令，ls cd ... 普通用户和root用户均可操作
- sbin — 超级用户的二进制目录，存放系统管理员使用的管理程序和守护进程，只有root用户可以操作
- boot — 系统启动目录，包含可引导linux内核和引导装载 (boot loader) 的配置文件，删除后虚拟机无法启动
- dev — 设备文件，任何设备与接口设备都是以文件形式储存在与这个目录的，(驱动文件)。包括终端设备 (tty\*) ,软盘 (fd\*) ,硬盘 (hd\*) , RAM (ram\*)和CD-ROM (cd\*)
- etc — 特定主机系统范围内的配置文件
- home — 普通用户的主目录，包含保存的文件、个人设置等等，一般为单独分区
- lib — 系统的函数库/bin /sbin 中二进制文件必要的库文件，几乎所有的应用程序都会使用到该目录下的共享库
- media — 媒体目录，提供挂载和自动挂载设备标准位置
- mnt — 临时挂载的文件系统，比如U盘，直接插入光驱无法使用，要先挂载后使用
- opt — 可选目录，存放第三方软件包和数据文件
- proc — 虚拟文件系统，将内核与进程状态归档成文本文件。该目录下文件只能看不能修改 (包括root)
- root — 超级用户的主目录
- run — 系统运行时候所需的文件
- srv — 服务目录，service的缩写，只要用来存储本机或本服务器提供的服务或者数据
- sys — 与/proc非常类似，也是一个虚拟的文件系统，主要记录与内核相关的信息
- usr — 默认软件都会存于该目录里下。用于存储只读用户数据的第二层次，包含绝大多数的用户工具 and 应用程序
- var — 可变目录，可以用于存储经常变化的文件，如日志文件

/usr    /etc    /dev    /boot    /lib    /media 重点需要详细看的目录

简述linux目录结构中 /usr 、/etc、/dev目录的作用

## 2) 查看文件

(1) **ls**: -l -a -i -h -R

(2) **cat**: 查看文件的内容, 默认显示到终端上;

```
cat 文件名
cat 1_malloc.c
cat /etc/issue      查看当前ubuntu的版本号
cat /etc/passwd     查看计算机中的所有用户

cat -n 文件名      带行号显示文件内容
```

(3) **vim**: 文本编辑器

(4) **head**: 显示文件的前几行

```
head 文件名          ==>默认显示文件的前10行
head -n line 文件名  ==>显示文件前line行
head -c bytes 文件名 ==>显示文件的前bytes个字节
```

(5) **tail**: 显示文件的后几行

```
tail 文件名          ==>默认显示文件的后10行
tail -n line 文件名  ==>显示文件后line行
```

## 练习

1. 查看文件的前6行, 带行号显示

```
head -n 6 /etc/passwd | cat -n
```

2. 查看文件的后6行, 带行号显示

```
cat -n /etc/passwd | tail -n 6
```

3. 查看文件的第5到第10行, 带行号显示

```
cat -n /etc/passwd | head -n 10 | tail -n 6
```

(6) **file**: 查看文件类型

```
file 文件名
file 1_malloc.c ==> C source
file a.out      ==> ELF 二进制可执行文件
```

### 3) 操作文件

(1) wc: 统计文件内容 word count

```
wc 文件名
wc 1_malloc.c
27      49      563      1_malloc.c
|        |        |
行数     单词个数   字符个数

wc -c 文件名      查看字节个数
wc -m 文件名      查看字符个数
wc -l 文件名      查看行数
```

man wc

#### (2) grep: 检索文件内容

```
grep "string" 文件名 --》显示文件中包含了string的那一行
grep "include" 1_malloc.c
```

grep -i "string" 文件名 --> ignore, 忽略string大小, 显示文件中包含string的那一行

```
ubuntu@ubuntu:4_shell命令 $ grep -i "include" 1_malloc.c
#include <stdio.h>
#include <stdlib.h>
INclude aaaaa
```

```
grep -n "string" 文件名 --> 带行号显示文件中包含string的那一行
grep -v "string" 文件名 --> 显示文件中 不包含 string的那一行
grep -w "string" 文件名 --> 精确查找, string单词, 显示文件中包含string单词的那一行 word
grep -R "string" * --> 递归查找, 显示当前目录下, 以及其子目录下所有包含string的那一行
```

```
grep "^string" 文件名 --> 显示文件中, 以string开头那一行
grep "string$" 文件名 --> 显示文件中, 以string结尾那一行
grep "^string$" 文件名 --> 显示文件中, 以string为单独成行的行
```

```
ubuntu@ubuntu:4_shell命令 $ grep "^#" 1_malloc.c
#include <stdio.h>
#include <stdlib.h>
ubuntu@ubuntu:4_shell命令 $ grep ";$" 1_malloc.c
void* ptr = malloc(10);
printf("malloc failed\n");
return -1;
int* iptr = (int*)ptr;
int* ptr1 = (int*)malloc(10);
return 0;
```

#### (3) find: 查找文件

`find` [路径] [选项]

根据名字查找文件: `-name`

`find ./ -name file` --> 查找指定路径 以及其所有子路径下, 符合文件名的文件  
`$find ./ -name file`

根据文件类型查找文件: `-type`      `bsp-lcd`: 普通文件在这里不是 `'-'` 而是 `'f'`

`find ./ -type f` --> 查找指定路径 以及其所有子路径下, 所有普通文件

根据文件权限查找文件: `-perm`

`find ./ -perm` 八进制权限

`find ./ -perm 0755` --> 查找指定路径 以及其所有子路径下, 所有权限是0755的文件

## 练习

将当前路径下所有.c结尾的普通文件查找出来;

```
find ./ -type f -name *.c
find ./ -type f | grep "\.c$"
```

### (4) cut: 截取文件

`cut -d "分割符号" -f 内容编号 文件名`

```
1 1 zs 90
2 2 ls 80
3 3 ww 78

ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 1 3.c
1
2
3
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 2 3.c
zs
ls
ww
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 3 3.c
90
80
78
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 1,3 3.c
1 90
2 80
3 78
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 1,2,3 3.c
1 zs 90
2 ls 80
3 ww 78
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 1-3 3.c
1 zs 90
2 ls 80
3 ww 78
ubuntu@ubuntu:4_shell命令 $ cut -d ' ' -f 1-2 3.c
1 zs
2 ls
3 ww
ubuntu@ubuntu:4_shell命令 $
```

### (5) >: 重定向

将指定内容打印到文件中, 并且前提是会将文件中原有内容删除后, 打印进去

```
cat 1.c > 2.c
ls -l > 2.c
```

## (6) >>: 追加

将指定内容打印到文件中，不会删除原有内容

```
cat 1.c >> 2.c
```

## 【4】通配符

\*: 通配任意长度，任意内容

```
ls *.c    rm *
```

?: 通配任意一个字符

```
rm ?.*    rm *.*?
```

[字符1字符2字符n]: 通配方括号中的任意一个字符

```
ls [123578].ac
```

[起始字符-结束字符]:

```
ls [1-3].c : 1.c 2.c 3.c
```

```
ls [a-c].c : a.c b.c c.c
```

[^字符1字符2字符n]

```
ls [^123].c    只要包含了123的字符都不打印
```

如果起始字符和结束字符是字母，涉及**本地语序**

本地语序默认的顺序是: aAbBcC..zZ

可以修改环境变量 LC\_ALL 清空本地语序 `export LC_ALL=C`

[a-z] 通配 a-z 中的任意一个字符

[A-Z] 通配 A-Z 中的任意一个字符

恢复本地语序 使用 `unset LC_ALL`

当前目录下创建 10.c 11.c 14.c 15.c 100.c 101.c

要求使用通配符查询到 10.c 11.c 14.c 15.c

```
ls ???.c
ls [1-9][1-9].c
```

## 【5】文件权限管理

```
-rw-rw-r-- 1 ubuntu ubuntu
```

```
0 七月 13 10:33 1.c
```

## 1) 修改权限

```
chmod [指定用户][运算符][权限] 文件名
[指定用户]: 当前用户(u) 组用户(g) 其他用户(o)
[运算符]: + 赋予权限 - 取出权限
[权限]: r w x
```

```
chmod g-r 1.c
chmod ug+rw 1.c
```

```
-----
chmod 八进制权限 文件名
chmod 0664 1.c
```

1. 将当前用户的读写执行权限全部删除

```
chmod u-rw 1.c
```

2. 将其他用户加上读写执行权限

```
chmod o+rw 1.c
```

3. 将文件权限修改为rw-rw----

```
chmod 0660 1.c
```

## 2) 修改文件所属用户

所有用户: /etc/passwd 可以查看

```
sudo chown 新用户名 文件名
sudo chown gdm 100.c

-rw-rw-r-- 1 gdm ubuntu 0 七月 13 11:22 100.c
```

## 3) 修改文件所属用户组

用户组信息所在的目录: /etc/group

```
sudo chown :新的组名 文件名
sudo chown :gdm 100.c
-rw-rw-r-- 1 gdm gdm 0 七月 13 11:22 100.c

-----
sudo chown 新的用户名:新的组名 文件
```

## 【6】链接命令



## 1) 软连接

相当于windows中的**快捷键**，可以通过软连接文件访问源文件，bsp-lcd中的l类型文件

```
ln -s 绝对路径/源文件 生成的软连接文件
ln -s 5.a s5.a
```

### 验证

1. 验证删除源文件，软连接文件是否还能使用
2. 移动软连接文件到上级目录，还能否通过软件连接文件访问源文件。
3. 验证软连接能够连接目录文件

注意：

1. **源文件被删除后，软连接文件会失效；**
2. 源文件最好使用绝对路径，防止软连接文件移动到其他目录后，链接关系不存在。
3. 软连接可以链接普通文件，也可以链接目录文件

## 2) 硬链接

硬链接相当于给文件取个别名，对文件硬链接一次，硬链接数就会+1；

```
ln 绝对路径/源文件 生成的硬链接文件
ln 7.b h17.b
-rw-rw-r-- 2 ubuntu ubuntu      16 七月 13 11:50 7.b
-rw-rw-r-- 2 ubuntu ubuntu      16 七月 13 11:50 h17.b
```

### 验证

1. 验证删除源文件，硬连接文件是否还能使用
2. 移动硬连接文件到上级目录，还能否通过硬件连接文件访问源文件。
3. 验证硬连接能够连接目录文件

注意：

1. 删除源文件，或者硬链接文件，硬链接数会减1，硬链接文件还是能够使用；
2. 当硬链接数被减为0，文件才会真正的被系统删除。
3. 硬链接可以随意移动位置以及目录，不会影响连接关系；
4. 硬链接只能连接文件，不能连接目录；

## 【7】管理用户的命令

用户信息存放：/etc/passwd

```
cat /etc/passwd
ubuntu:x:1000:1000:ubuntu,,,:/home/ubuntu:/bin/bash
ubuntu:用户名
x: 该用户有设置密码，真正的密码存储在/etc/shadow文件中，加密显示；
1000: UID，用户ID号
1000: GID，用户组ID
ubuntu,,,: 描述该用户，没什么用，
/home/ubuntu/: 该ubuntu用户的用户家目录；
/bin/bash: 该用户使用的命令解释器，是用户与Linux内核沟通的桥梁
```

## 1) 添加用户

```
sudo useradd 新用户名 --->快速创建用户名，不会给新用户创建家目录，也不会指定命令解析器，所以该指令基本不使用
sudo adduser 新用户名 --->详细创建用户；
```

## 2) 切换用户

```
sudo su 用户名
exit 返回上一个用户
```

用户名不在sudoer文件中，此事将被报告：



```
wjm@ubuntu:/home/ubuntu$ sudo su ubuntu
[sudo] wjm 的密码：
wjm 不在 sudoers 文件中。此事将被报告。
wjm@ubuntu:/home/ubuntu$ su ubuntu
```

1. 切换到root .

```
sudo vim /etc/sudoers
```

2. 在文件中找到

```
# User privilege specification
root    ALL=(ALL:ALL) ALL
```

3. 将需要sudo权限的账户，添加到root后面；

```
# User privilege specification
root    ALL=(ALL:ALL) ALL
psy     ALL=(ALL)      ALL
```

4. 保存退出 :wq!

## 3) 删除用户

```
sudo userdel -r 用户名      删除用户，并把家目录以及家目录下的一并删除
```

## 4) 修改密码

```
sudo passwd 用户名
```

如果要修改当前用户的密码，用户名可以省略

## 【8】管理用户组的命令

```
/etc/group
```

```
ubuntu:x:1000:组内成员列表
```

```
ubuntu: 组名
```

x: 该用户有设置密码，真正的密码存储在/etc/shadow文件中，加密显示；

```
1000: GID
```

组内成员列表：如果是空字段，则代表该用户中只有一个用户，就是自己

### 1) 新建组

```
sudo groupadd 组名
```

### 2) 删除组

```
sudo groupdel 组名
```

### 3) 组内添加新的成员

```
sudo addgroup 用户名 组名
```

### 4) 删除组内成员

```
sudo delgroup 用户名 组名
```

## 【9】磁盘相关的命令

### 1. 查看磁盘分区

```
sudo fdisk -l
```

```
/dev/sda 自己电脑的磁盘
```

```
/dev/loop 伪设备，仿真设备，这个设备是的文件向块设备一样可以一块一块的被访问；
```

```
/dev/sdb 外接的磁盘，是移动硬盘
```

### 2. 查看磁盘的使用率

```
sudo df -h
```

| 文件系统 | 容量 | 已用 | 可用 | 已用% | 挂载点 |
|------|----|----|----|-----|-----|
|------|----|----|----|-----|-----|

### 3. 挂载磁盘

如果不挂载，则无法访问磁盘，插入的外接磁盘的时候，有些操作系统会自动挂载磁盘

```
sudo df -h. 可以看到默认挂载到/media/ubuntu/My Passport
```

```
sudo mount /dev/sdb1 ./222/
```

#### 4. 取消挂载

```
sudo umount 挂载目录
```

##### 1) 取消默认挂载

```
sudo df -h  
sudo umount /media/ubuntu/My\ Passport
```

##### 2) sudo fdisk -l 找到要挂载的外借磁盘名字：/dev/sdb1

##### 3) 挂载到指定目录下，例如：./222/

```
sudo mount /dev/sdb1 ./222/
```

##### 4) 访问./222/其实就是访问我的外接磁盘;

## 【10】关机命令

### 1) 关机

```
sudo shutdown -h 时间  
sudo shutdown -h now      立即关机  
sudo shutdown -h 0  
  
sudo shutdown -h 60        60分钟后关机  
sudo shutdown -h 18:00     18:00后关机  
sudo shutdown -c           取消
```

### 2) 重启

```
sudo shutdown -r 时间  
  
sudo reboot      立即重启
```

## 【11】创建索引文件

### 1.ctags工具

```
sudo apt-get install ctags
```

## 2.为源码创建一个索引文件

```
如果想查看内核源码
cd /lib/modules/3.13.0-32-generic/build
sudo ctags -R
在vim ~/.vimrc中添加 set tags+=/usr/include/tags
-----
如果想查看库代码
cd /usr/include
sudo ctags -R
在vim ~/.vimrc中添加 set tags+=/usr/include/tags

应用层建议大家用下面这个
```

## 3.追代码

```
终端输入： vi -t 想要查看的变量名 宏名 数据类型          函数：用
           vi -t ssize_t

选择其中一个选项，按下回车

追代码
    将光标停留在要追的类型行：按下 ctrl + ]

回退：
    ctrl + t
```

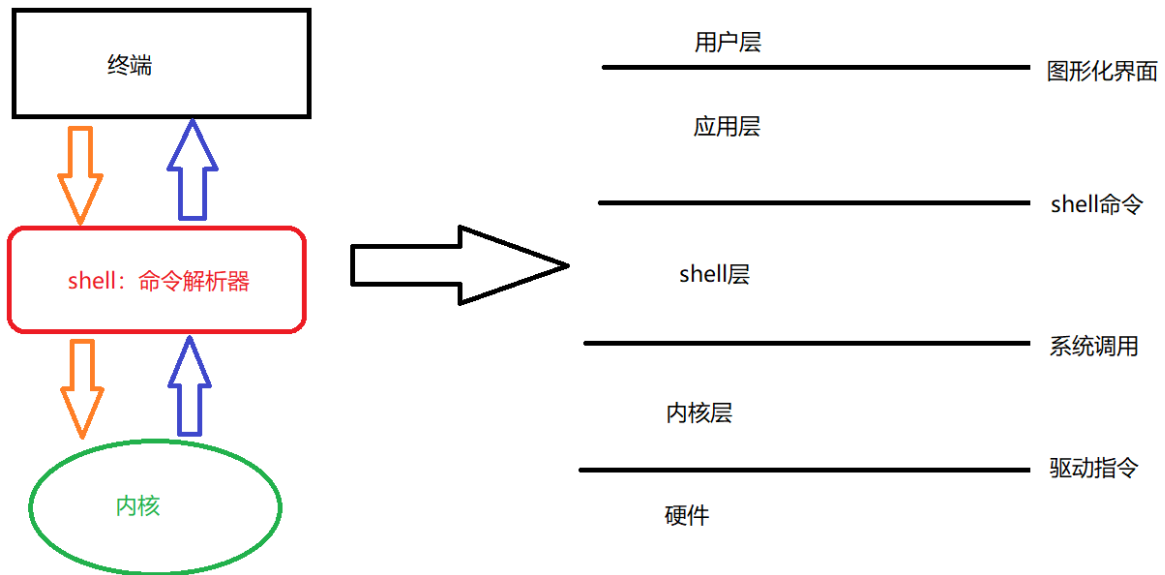
# 8. shell脚本

## 【1】shell的概念

### 1) 命令解析器

shell：命令解析器，解析可执行二进制程序；

用户在终端输入shell命令，由shell命令解析器对命令进行解析，解析成内核能够识别的指令，然后有内核去执行命令，最后由终端显示执行命令的结果给用户；



## 2) shell命令解析器的分类

### 查看本机的解析器

```
cat /etc/shells
```

```
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
```

```
ubuntu@ubuntu:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 五月 26 2020 /bin/sh -> dash
ubuntu@ubuntu:~$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1113504 六月 7 2019 /bin/bash
ubuntu@ubuntu:~$ ls -l /bin/rbash
lrwxrwxrwx 1 root root 4 五月 26 2020 /bin/rbash -> bash
ubuntu@ubuntu:~$ ls -l /bin/dash
-rwxr-xr-x 1 root root 121432 一月 25 2018 /bin/dash
ubuntu@ubuntu:~$
```

1. sh 最早期的shell，但是与终端的交互效果非常差
2. csh 在sh上修改，语法更接近C语言，交互效果差
3. ksh 兼容了sh csh
4. bash 目前在ubuntu中最常用的shell，语言风格更加接近C，增强了交互性;

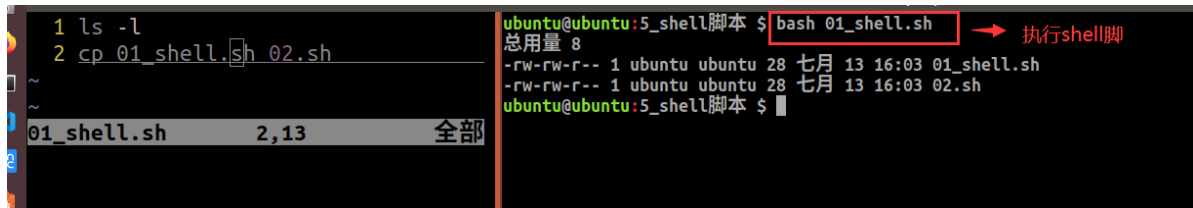
### 查看本机使用了什么shell

```
echo $SHELL
```

## 【2】shell脚本的基础

## 1) 概念

1. shell脚本是一个**以.sh为后缀**的文件
2. shell脚本中编写的是有序的，能够实现特定功能的 **shell命令集合**  
所以不需要像C语言一样编译;
3. 除了命令之外还有特定的语法：分支结构 循环结构体等等

A terminal window with a dark background. On the left, a file explorer shows a directory with files '01\_shell.sh' and '02.sh'. The main terminal area shows the command 'bash 01\_shell.sh' being executed, followed by a red arrow pointing to the text '执行shell脚本'. Below this, the output of the script is shown, including file permissions and timestamps for '01\_shell.sh' and '02.sh'.

任务：在当前目录下创建一个文件夹，将/etc/passwd文件拷贝到该文件夹下

```
mkdir ./test/
```

```
cp /etc/passwd ./test/
```

```
#!/bin/bash

mkdir ./test/
cp /etc/passwd ./test
```

## 2) shell脚本的基本格式

```
#!/bin/bash
```

 --->指定脚本的shell 的路径以及名字，必须放在第一行

shell命令集合

## 3) shell脚本的注释

### 单行注释

```
# shell脚本的单行注释
```

### 多行注释

```
:<<标识符
```

 --->表示反复自己取

要注释的内容

标识符

### 可视块注释

1. 将光标移动到要注释的第一行行首位置，按下`ctrl + v`，进入可视块模式（左下角可以看到该模式提示）
2. 按下方向键 或者 `hjkl`选中要注释的行
3. 按下`shift+i`，进入插入行模式
4. 输入`#`，注释第一行
5. 按下两下`esc`按键

-----

删除多行注释：

1. 将光标移动到要注释的第一行行首位置，按下`ctrl + v`，进入可视块模式（左下角可以看到该模式提示）
2. 按下方向键 或者 `hjkl`选中要删除的行
3. 按下`d`，删除

## 4) 执行

三种方式：

- `./xxx.sh`  
`xxx.sh` 必须有可执行权限。`chmod 0777 xxx.sh`
- `bash xxx.sh`
- `source xxx.sh`

区别：

1. `./`和`bash`执行的时候，是再开一个子终端，在子终端运行脚本后，将结果返回到当前终端；
2. `source` 方式是直接在当前终端执行shell脚本；
3. `./`方式需要由可执行权限。

## 【3】shell脚本中的变量

### 1) 变量定义

1. shell脚本中的变量默认都是字符串类型，变量不需要定义，可以直接使用
2. 变量后 **不需要加分号**
3. 默认情况下，变量的作用域都是全局的。

### 2) 变量的赋值

1. 变量赋值的时候，用`=`号赋值，注意**`=`号两边不能有空格**；
2. shell脚本的变量都是字符串，没有整型，浮点类型，字符串类型的区别；
3. `=`的右值，可以加双引号，单引号，或者不加符号。但是**建议都加上"** "
4. 如果字符串中间有空格，必须使用**单引号或者双引号将字符串括起来**  
区别：'单引号中无法 \$其他变量，" "中可以引用别的变量；



```

1 #!/bin/bash
2
3 a=10
4 #b = 10      #b: 未找到命令
5
6 c=abc
7 d='abc'
8 e="abc"
9
10 f=hello
11 #g=hello world #错误, 有空格需要加'或者'"
12 g='hello world'
13 g="hello world"
14
15 h="hhahaha$a"
16 echo $h      #打印hhahaha10
17
18 h='hhahaha$a' #hhahaha$a
19 echo $h
20
21

```

### 3) \$

#### i. 引用变量

获取变量中存储的值

```

$变量名
${变量名}
-----
var1=3.14
var2=$var1
echo $var2

var3=${var1}
echo $var3

```

#### ii. 其他

```

$?  判断上一条语句是否执行成功, 成功返回0, 失败返回非0: [1~255]
$$  查看当前脚本的进程号

```

### 4) echo: 输出到终端

```

echo var
var=1
echo var    #var

echo $var
echo ${var}

echo var=$var
echo "var=$var"
echo 'var=$var'    #单引号无法使用$

```

```
echo -n var=$var    #不会自动换行

var='\t'
echo -e "var=$var aaa"  #识别转义字符
echo "var=$var aaa"
```

```
1 #!/bin/bash
2
3 var=1
4 echo var    #var
5
6 echo $var
7 echo ${var}
8
9 echo var=$var
10 echo "var=$var"
11 echo 'var=$var'    #单引号无法使用$
12
13 echo -n var=$var    #不会自动换行
14
15 var='\t'
16 echo -e "var=$var aaa"  #识别转义字符
17 echo "var=$var aaa"
18
19 unset var
20 echo var=$var    #var=
21
```

```
ubuntu@ubuntu:5_shell脚本 $ bash 05_
var
1
1
var=1
var=1
var=$var
var=1var=    aaa
var=\t aaa
var=
ubuntu@ubuntu:5_shell脚本 $
```

## 5) unset: 清空变量的值

```
unset var
```

### 练习

1. 将两个变量中的值进行交换
2. 在当前目录下创建sub文件夹，将/etc/group拷贝到sub中，将sub重命名成sub1，将group的内容打印到中端上

## 6) 位置变量

什么是位置变量：脚本执行后的命令行位置传参，或者叫做外部传参

```
ubuntu@ubuntu:5_shell脚本 $ bash 01_shell.sh    aaa    bbb    ccc
-----
0号位置变量    1号    2    3
```

### 如何获取位置变量

- \$0: 获取到0号位置变量的内容
- 不同的执行方式，\$0位置参数结果不同。

```
ubuntu@ubuntu:5_shell脚本 $ bash 06.sh aaa bbb
06.sh
aaa
bbb
ubuntu@ubuntu:5_shell脚本 $ chmod u+x 06.sh
ubuntu@ubuntu:5_shell脚本 $ ./06.sh aaa bbb
./06.sh
aaa
bbb
ubuntu@ubuntu:5_shell脚本 $ source 06.sh aaa bbb
/bin/bash
aaa
bbb
ubuntu@ubuntu:5_shell脚本 $
```

- \$1: 获取到1号位置变量的内容
- \$2: 获取到2号位置变量的内容
- \$n: 获取到n号位置变量的内容
- \$@ 或者 \$\*: 获取除了0号位置变量外的所有位置变量内容
- \$#: 获取位置变量的个数, 除了0号

|                                                                                                                  |                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>1 #!/bin/bash 2 3 echo \$0 4 echo \$1 5 echo \$2 6 7 echo "@: \$@" 8 echo "*: \$*" 9 10 echo "#: \$#"</pre> | <pre>ubuntu@ubuntu:5_shell脚本 \$ bash 06.sh aaa bbb 111 222 33 06.sh aaa bbb @: aaa bbb 111 222 33 *: aaa bbb 111 222 33 #: 5 ubuntu@ubuntu:5_shell脚本 \$</pre> |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 7) \$() 或者 ``: 命令置换符

``: 注意该按键是esc下面的那个按键, 与~同一个按键;

执行一条命令语句, 将结果赋值给另外一个变量;

|                                                                                        |                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>1 #!/bin/bash 2 3 4 var=\$(ls) 5 echo \$var 6 7 var=`ls ../` 8 echo \$var 9</pre> | <pre>ubuntu@ubuntu:5_shell脚本 \$ bash 07.sh 01_shell.sh 02.sh 03_cd.sh 04_var.sh 05_echo.sh 06.sh 07.sh t1_text.sh text 1_分文件 2_malloc 3_struct 4_shell命令 5_shell脚本 ubuntu@ubuntu:5_shell脚本 \$</pre> |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 【4】shell脚本中的字符串

### 1) 字符串变量申请

shell中的变量存储的都是字符串, 可以加单引号, 双引号, 或者不加, 建议选择用双引号

## 2) 计算字符串的长度

`${#变量的名字}`

```
1 #!/bin/bash
2
3 var="aaa bbb"
4
5 var2=${#var}
6 echo var2=$var2      #var2=0var
7
8 var3=${#var}
9 echo var3=$var3      #7
10
```

任务：将var1和var2的内容引用到var3中，计算var3的长度

## 3) 字符串的拼接

```
var4="aaa"
var5="bbb"
var6="$var4$var5"    #"aaabbb"
```

## 4) 获取子串

|                            |                                           |
|----------------------------|-------------------------------------------|
| <code>\${var:n}</code>     | 获取var变量的子串，从第n个位置开始获取到结尾，字符串编号从第0开始编      |
| <code>\${var:n:m}</code>   | 获取var变量的子串，从第n个位置开始获取m个，字符串编号从第0开始编       |
| <code>\${var:0-n}</code>   | 获取var变量的子串，从倒数第n个位置开始获取到结尾，字符串编号从倒数第1位开始编 |
| <code>\${var:0-n:m}</code> | 获取var变量的子串，从倒数第n个位置开始获取m个，字符串编号从倒数第1位开始编  |

## 练习

```
var="abcdefg"
```

1. 想要获取defg字符串，
2. 想要获取abc子串
3. 想要获取cde子串
4. 想要获取fg子串

```
var="abcdefg"
var1=${var:3}
echo $var1

echo ${var:0:3}

echo ${var:2:3}
echo ${var:0-5:3}

echo ${var:0-2}
```

## 【5】shell脚本中的数组

### 1) 格式

1. shell脚本中只有一维数组的概念，且数组没有数据类型，所有的元素都是字符串
2. shell脚本中( )标识该变量是一个数组，每个元素之间用空格隔开

#### 格式1:

```
var=(111 222 33 444 555)
var=('111' '222' '33' '444' '555')
var=("111" "222" "33" "444" "555")
下标默认从0开始编；
${var[0]} ${var[1]}
```

#### 格式2:

shell脚本支持稀疏数组，下标整数可以不连续

```
var=([0]="000" [4]="444" [2]="222" [10]="101010")

var=([0]="000" [4]="444" [2]="222" [10]="101010")
echo "@: ${var[@]}"      #@: 000 222 444 101010
echo "*: ${var[*]}"
```

### 练习

将位置变量的内容存储到数组中，打印数组中的内容

```
#!/bin/bash

arr=("$1" "$2" "$3")

echo "0:${arr[0]}"
echo "1:${arr[1]}"
echo "2:${arr[2]}"

echo "@: ${arr[@]}"
echo "*: ${arr[*]}"
```

## 2) 访问

### i. 访问单个元素

`${数组名[下标]}`

下标从0开始标

### ii. 访问所有元素

`${数组名[@]}` 或者 `${数组名[*]}` :

获取到数组中的所有元素，中间用空格隔开，**打印顺序：从小下标到大下标**

```
1 #!/bin/bash
2
3 arr=("$1" "$2" "$3")
4
5 echo "0:${arr[0]}"
6 echo "1:${arr[1]}"
7 echo "2:${arr[2]}"
8
9
10 echo "@: ${arr[@]}"
11 echo "*: ${arr[*]}"
12
```

```
ubuntu@ubuntu:5_shell脚本 $
ubuntu@ubuntu:5_shell脚本 $
ubuntu@ubuntu:5_shell脚本 $
ubuntu@ubuntu:5_shell脚本 $ bash 09_array.sh aa bb cc
0:aa
1:bb
2:cc
@: aa bb cc
*: aa bb cc
ubuntu@ubuntu:5_shell脚本 $
```

## 3) 追加元素

```
var=("111" "222" "333")
var=(${var[@]} "字符串1" "字符串2" "字符串n")

var=("字符串1" "字符串2" "字符串n" ${var[@]} )

var[5]="aaa"

var=("11" "22" "33")
var[10]="101010"
echo "var=${var[@]}"
```

## 4) 元素个数

`${#var[@]}` 或者 `${#var[*]}`

```
echo "${#var[*]}" #4
```

## 5) 每个元素的字符个数

`${#数组名[下标]}`

```
echo ${#var[0]}      打印下标为0的元素的字符个数
echo ${#var[10]}     打印下标为10的元素的字符个数
```

## 练习

1. 将ls的内容存储到数组中，显示数组中有多少个元素，并把数组的内容打印出来;

```
var=( $(ls) )
echo ${#var[@]}
echo ${var[@]}
```

# 【6】 shell脚本的输入

## 1) read：从终端获取数据

### i. read 变量名

功能：从终端获取数据，存储到变量中

1. 阻塞等待数据输入
2. read可以获取到空格字符，但是无法获取到'\n'字符

```
read var
echo "var=$var"
```

```
1 #!/bin/bash
2
3 read var
4 echo -e "var=$var"
5
ubuntu@ubuntu:5_shell脚本 $ bash 10_read.sh
aaa bbb
var=aaa bbb
ubuntu@ubuntu:5_shell脚本 $
```

## ii. read 变量名1 变量名2

功能：同时获取多个数据，输入的时候数据之间用空格隔开；  
如果想要获取的数据中间有空格，则需要转换成单个元素输入；

```
read var1 var2                #111 222 333 444

echo "var1=$var1 var2=$var2"  #var1=111 var2=222 333 444
```

## iii. read -p "提示用的字符串" 变量名

```
read -p "请输入>>>" var
echo "var=$var"
```

## iv. read -a 数组的名字

功能：获取数据存储到数组中，输入的时候元素数据之间用空格隔开  
如果想要获取的数据中有空格，则可以转换成单个元素输入：i

```
#!/bin/bash

read -p "请输入数组>>>" -a arr
echo "arr=${arr[*]}"

read -p "请输入第10个元素的数据" arr[10]
echo "arr=${arr[*]}"
```

## v. read -n 字符个数 变量名

功能：指定个字符的个数，当时输入的个数满足指定字符个数时，直接解除阻塞，结束输入；

## vi. read -t 秒数 变量名

功能：阻塞等待指定秒数后，如果还是没有数据输入，则直接解除阻塞；

# 【7】运算符和运算指令

shell脚本中变量都是字符串，如果要进行算数运行（2+3 2\*3），必须使用算数运算指令；



## 1) shell脚本中支持的运算符

```
+ - * / %  
**      幂运算  2**5    2的5次方  
= += -= *= /= %=  
++ --  
&& || !  
< <= >= > != ==  
<< >>  
& | ^ ~
```

## 2) 算数运算指令

1. (( )) 高效，用法灵活 ---- 熟练并掌握
2. let 与(( ))功能相近
3. \$[] 效率低，用法不灵活
4. expr 语法要求非常严格

### i. (( ))

注意：两个圆括号中间不能有空格：((

```
(( 运算表达式 ))
```

1. 写法灵活，支持C的所有运算符语法
2. (( 运算表达式 ))可以单独存在，可以单独占用一行；

```
((1+2))  
  
var=10  
((var+=1))  
echo "var=$var"
```

3. 如果需要使用到运算结果，则在开始加上\$符号，引用结果: **\$(( 算数表达式 ))**

```
var2=11  
var2=$((var2+2))  
echo "var2=$var2"
```

4. (( 运算表达式 )) 表达式中的变量可以加\$ 或者 \${}，也可以不加

```
var2=11  
var2=$((var2+2))  
echo "var2=$var2"    #13  
  
var2=$(( $var2+2 ))  
echo "var2=$var2"    #15
```

5. ++、-- 运算符，(( 运算表达式 )) 运算符表达式中，**不要使用\$符号去引用变量**.

且运算逻辑与C语言完全一致，也分前置，后置的区别；

```
var3=1
((++$var3))

echo "var3=$var3" #1 # $取出var3中的值，将值做++运算，没有修改var3本身的内容

var3=1
((++var3))
echo "var3=$var3" #2
#####
var4=10
(($var4++))
echo "var4=$var4" #10 # 语法错误：需要操作数（错误符号是 "+"

((var4++))
echo "var4=$var4" #11

var5=$((var4++))
echo "var4=$var4" #12
echo "var5=$var5" #11

var6=$((++var4))
echo "var4=$var4" #13
echo "var6=$var6" #13
```

## 6. (( ))能不能使用幂运算符

```
var7=2
var8=$((var7 ** 2))
echo "var8=$var8"
```

## ii. let

```
let "运算符表达式"
```

### 1. 双引号可以省略不写，但是不能改成单引号

```
var=1
let "var=var+1"
echo var="$var"

let "var=$var+1"
echo var="$var"

let var=var+1
echo var="$var"

let var=$var+1
echo var="$var"
```

### 2. let同样也能够使用++ -- 幂运算等等

```
var=20
let var++
let ++var

let var1=var**2
```

### iii. \$[ ]

变量名=\$[算数表达式]

1. \$[ ] 不能单独使用，必须有变量接收或者打印出来

```
$[1+2]      #3: 未找到命令

var=$[1+2]
echo $[1+2]
```

2. \$[ ] 可以引用别的变量进行计算，

```
var1=19
var2=20
echo $[var1+var2]
echo $[${var1+$var2}]
```

3. 能使用++ -- 幂运算

++、-- 运算符，[ ] 中 **不要使用\$符号去引用变量**。

### iv. expr

#### (1) 算数运算

expr \$变量1 算数运算符 \$变量2

1. 变量之前 **必须加\$引用变量**，且变量与运算符之间 **必须有空格**
2. expr 会直接打印结果，并换行

```
1 #!/bin/bash
2
3 expr 1+ 2    #语法错误
4 expr 1 + 2   #运行后，直接打印3
5
6 var1=4
7 var2=5
8 expr var1 + $var2    #expr: 非整数参数
9 expr $var1 + $var2
10
11
```

```
ubuntu@ubuntu:5_shell脚本 $
ubuntu@ubuntu:5_shell脚本 $ bash 15.sh
expr: 语法错误
3
expr: 非整数参数
9
ubuntu@ubuntu:5_shell脚本 $
```

3. 无法做幂运算 ++ -- 运算
4. 如果想要做乘法运算，需要将称号写成 \\*

```
var3=3
expr $var3 * 2      #expr: 语法错误
expr $var3 \* 2     #6
```

5. 如果想要获取expr的结果，赋值给另外一个变量，需要使用命令置换符号 `$( )``

```
var4=$(expr $var3 \* 2)    # $( ) ``命令置换符号
echo var4=$var4
```

## (2) 字符串运算

```
expr $字符串变量1 : $字符串变量2
expr match $字符串变量1 $字符串变量2
```

功能：判断字符串1和字符串2是否相同，从第一个字节的位置开始判断。直到比较到不同，或者其中一个字符串结束。

返回值：

当比较到有不同字符的时候，返回0；

如果一直都一致，当其中一个字符串结束，返回相同的字节数；

```
-----
str1="www.baidu.com"
expr $str1 : "www" #3
expr $str1 : "wwwb" #0, 比较到不同, 返回0;
expr $str1 : "baidu" #0, 比较到不同, 返回0;
```

```
expr substr $源字符串 下标 长度
```

功能：从源字符串指定下标位置，或者指定长度的子串，字符串下标从1开始标；

```
str1="www.baidu.com"
str2=`expr substr $str1 5 5`
echo str2=$str2      # baidu
```

```
expr index $源字符串 字符
```

功能：从源字符串中查找指定字符所在的下标

返回值：

如果字符不在源字符串中，返回0；

如果找到了，返回第一个查找到字符的下标，下标从1开始编号

```
str1="www.baidu.com"
expr index $str1 w #1
expr index $str1 b #5
```

```
str1="www.baidu.com"
expr index $str1 "ba" #5--->b的下标
expr index $str1 "pa" #6--->a的下标
expr index $str1 "aw" #1--->w的下标，返回找到的最小的下标值
```

```
expr length $字符串
```

功能：计算字符串长度

```
str1="www.baidu.com"
```

```
expr index $str1 "ba"    #5--->b的下标
```

```
expr index $str1 "pa"    #6--->a的下标
```

```
expr length $str1        #13
```

## 练习

从终端获取一个文件名，例如 aaa.c 要求分离出aaa 以及后缀c

```
aaa.txt ==>  aaa  txt
```

```
bbb.c ==>   bbb  c
```

```
#!/bin/bash
```

```
read -p "请输入一个带后缀的文件名>>" str
```

```
pos=$(expr index $str ".")
```

```
#####
```

```
name=`expr substr $str 1 $((pos-1))`
```

```
echo name=$name
```

```
len=`expr length $str`  #计算这个字符串的长度
```

```
secname=`expr substr $str $((pos+1)) $((len-pos))`
```

```
echo secname=$secname
```

```
#####
```

```
name=${str:0:$((pos-1))}
```

```
echo name = $name
```

```
secname=${str:$pos}
```

```
echo secname = $secname
```

## 【8】shell中的分支结构

### 1) if-else语句

#### i. 格式

1. 相当于C语言中的 **if{}语句**

```
if [ 判断语句 ]      注意：if与[]之间有空格 判断语句与左右方括号[ ]，之间也有空格隔开
then
```

```
    语句块
```

```
fi
```

```
-----
```

```
#!/bin/bash
```

```
var1=22
var2=20

if [ $var1 -gt $var2 ]
then
    echo "var=$var1比较大"
fi
```

## 2. 相当于C语言中的 **if-else** 语句

```
if [ 判断语句 ]      注意: if与[]之间有空格 判断语句与左右方括号[ ], 之间也有空格隔开
then
    语句块
else                  注意: else下面不带then
    语句块
fi
-----
var1=19
var2=20

if [ $var1 -gt $var2 ]
then
    echo "var=$var1比较大"
else
    echo "var=$var1不大于$var2"
fi
```

## 3. 相当于C语言中的 **if-else if-else if-else**

```
if [ 判断语句 ]      注意: if与[]之间有空格 判断语句与左右方括号[ ], 之间也有空格隔开
then
    语句块

elif [ 判断语句 ]
then
    语句块

elif [ 判断语句 ]
then
    语句块

else                  注意: else下面不带then
    语句块
fi
-----
#!/bin/bash

var1=19
var2=20

if [ $var1 -gt $var2 ]
then
    echo "var=$var1比较大"
elif [ $var1 -eq $var2 ]
```

```
then
    echo "var=$var1等于$var2"
elif [ $var1 -lt $var2 ]
then
    echo "var=$var1小于$var2"
fi
```

## 2) shell脚本中的text命令

shell脚本中的test命令用于检查某个条件是否成立，它可以进行数值、字符和文件的判断

### i. 整数大小比较

#### 格式

`$变量 指令 $变量`

**注意：指令前后有空格**

|     |                 |
|-----|-----------------|
| -gt | 大于 greater than |
| -lt | 小于 less than    |
| -eq | 等于 equal        |
| -ge | 大于等于            |
| -le | 小于等于            |
| -ne | 不等于             |

#### 练习

0. 终端输入一个整形数据，判断整形数据是否为偶数。

如果是，打印出：%d是偶数，否则，打印出：%d是奇数。

1. 学生成绩管理：

从外部输入一个学习成绩，范围为0 - 100，  
成绩大于等于90分，则输出A，  
成绩大于等于80，则输出B，  
成绩大于等于60，则输出C，  
小于60，则输出D

2. 输入三个数：按照从小到大的顺序进行排列

int a = 10, b = 20, c = 5;

...

a = 5, b = 10, c = 20

3: 判定给定的年份是否为闰年（闰年：能被4整除，但是不能被100整除 或者能被400整除） // 选做

## 第2题

```
#!/bin/bash

read -p "请输入成绩" score

if [ $score -ge 90 ]
then
    echo "A"
elif [ $score -ge 80 ]
then
    echo "B"
elif [ $score -ge 60 ]
then
    echo "C"
else
    echo "D"
fi
```

## 第3题

```
#!/bin/bash

vara=10
varb=20
varc=5

if [ $vara -gt $varb ]
then
    temp=$vara
    vara=$varb
    varb=$temp
fi

if [ $vara -gt $varc ]
then
    temp=$vara
    vara=$varc
    varc=$temp
fi

if [ $varb -gt $varc ]
then
    temp=$varb
    varb=$varc
    varc=$temp
fi

echo "a=$vara b=$varb c=$varc"
```



## ii. 字符串比较

|           |                         |
|-----------|-------------------------|
| ==        | 判断两个字符串是否相等             |
| !=        | 判断两个字符串是否不相等            |
| \< \>     | 比较字符串的大小                |
| -z \$字符串名 | 判断字符串是否为空，如果为空则返回真 zero |
| -n \$字符串名 | 判断字符串是否不为空，如果不为空返回真     |

### 注意:

1. 对于单对的[ ]，字符串比较时候必须要加上双引号，防止字符串中有空格

```
#!/bin/bash

str1="hello world"

if [ -n "$str1" ]
then
    echo "str1=$str1"
fi
#####
#!/bin/bash

str1="hello world"

if [ -n $str1 ]          #第 5 行: [: hello: 需要二元表达式
then
    echo "str1=$str1"
fi
```

2. 用双对的[[ ]], 字符串变量中不需要加双引号

```
if [[ -n $str1 ]]
then
    echo "str1=$str1"
fi
```

### 只有bash解析器才能使用 [[ ]]

```
1 #!/bin/bash
2
3 read -p "请输入一个字符串>>>" str1
4 read -p "请输入另一个字符串>>>" str2
5
6 if test "$str1" \> "$str2"
7 then
8     echo "$str1 > $str2"
9 elif test "$str1" \< "$str2"
10 then
11     echo "$str1 < $str2"
12 else
13     echo "$str1 = $str2"
14 fi
```

```
linux@LINUX-Ubuntu: ~/HqYJ/22061班/AdvancedC/day6$ bash shell.sh
请输入一个字符串>>>hello
请输入另一个字符串>>>hallo
hello > hallo
linux@LINUX-Ubuntu: ~/HqYJ/22061班/AdvancedC/day6$ bash shell.sh
请输入一个字符串>>>helloworld
请输入另一个字符串>>>helloworlz
helloworld < helloworlz
linux@LINUX-Ubuntu: ~/HqYJ/22061班/AdvancedC/day6$
```

### iii. 逻辑运算符

| [ ] | [[ ]] |          |
|-----|-------|----------|
| -a  | &&    | 逻辑与      |
| -o  |       | 逻辑或      |
|     | !     | 计算结果总是为假 |
|     | ()    | 计算结果总是为真 |

#### 1. [ ]中只能使用 -a -o形式

```
#!/bin/bash

read -p "请输入一个整数" var

if [ $var -gt 10 -a $var -lt 20 ]
then
    echo "$var在(10-20)范围内"
else
    echo "$var不在(10-20)范围内"
fi
```

#### 2. if后面可以跟多个 [ ],中间用 && || 连接

```
if [ $var -gt 10 ] && [ $var -lt 20 ]
then
    echo "$var在(10-20)范围内"
else
    echo "$var不在(10-20)范围内"
fi
```

#### 3. 双对的[[ ]]中只能使用 && || () !的形式

```
if [[ $var -gt 10 && $var -lt 20 ]]
then
    echo "$var在(10-20)范围内"
else
    echo "$var不在(10-20)范围内"
fi
```

### iv. 判断文件属性

指令 \$文件名

|    |                                      |
|----|--------------------------------------|
| -f | 判断这个文件是否是普通文件                        |
| -d | 目录文件                                 |
| -b | 块设备驱动文件                              |
| -S | socket套接字文件                          |
| -p | 管道文件                                 |
| -L | 是否是软链接文件，链接文件同时也是普通文件                |
| -c | 字符设备驱动文件                             |
| -e | 判断文件是否存在，存在为真                        |
| -s | 判断文件中是否有数据，如果有数据则为真。如果文件不存在或者为空，则返回假 |

#### v. 判断文件权限

|    |            |
|----|------------|
| -r | 判断文件是否有读权限 |
| -w |            |
| -x |            |

#### 练习

输入一个文件名，判断文件是否存在，如果存在判断是否是普通文件，如果是普通文件，判断是否有可写权限

如果有，则将hello world追加到文件中

```
#!/bin/bash

read -p "请输入文件名>>>" file

if [ -e "$file" ]
then
    echo "$file文件存在"

    if [ -f "$file" ]
    then

        if [ -w "$file" ]
        then
            echo "hello world" >> "$file"

            if [ $? -eq 0 ]      #判断上一条语句是否执行成功
            then
                echo "追加成功"
            else
                echo "追加失败"
            fi
        fi
    fi
fi
```

```

else
    echo "$file没有写权限"

fi

else
    echo "$file不是普通文件"
fi

else
    echo "$file不存在"
fi

```

### 3) case-in语句

相当于：C语言中的switch-case语句

#### i. 格式

```

case $变量 in
    选项1)                #这一行相当于switch
                        #相当于c语言的case
        语句1
        ;;
    选项2)
        语句2
        ;;
    选项n)
        语句n
        ;;
    *)                    #相当于是default，也可以省略该选项
        默认执行语句
        ;;
esac                      //line1
                        #代表case结构结束      //line2

```

注意：line1和line2必须紧挨着，不能有空行

1. \$变量：可以是变量，可以是数字常量，可以是字符串常量，也可以是数学计算表达式

```

1 #!/bin/bash
2
3 read -p "请输入1/2/3/4/5/6/7 >>>" week
4
5 case $week in
6     "1")
7         echo "星期一"
8         ;;
9     "2")
10        echo "星期二"
11        ;;
12    *)
13        echo "星期34567"
14        ;;
15 esac
16
17 case "1" in
18     "1")
19         echo "星期一"
20         ;;
21     "2")
22         echo "星期二"
23         ;;
24    *)
25        echo "星期34567"
26        ;;
27 esac
28
29
30 case "hahaha" in
31     "1")
32         echo "星期一"
33         ;;
34     "2")
35         echo "星期二"
36         ;;
37    *)
38        echo "星期34567"
39        ;;
40 esac
41
42 case ${1+1}) in
43     "1")
44         echo "星期一"
45         ;;
46     "2")
47         echo "星期二"
48         ;;
49    *)
50        echo "星期34567"
51        ;;
52 esac
53

```

2. 若多个有关系的选项想要走同一条语句：可以使用正则表达式

[123] 代表123中任意一个数字

```
[1-3]    代表123中任意一个数字
[abc]
[a-zA-Z]
```

```
read -p "请输入>>>" var
case $var in
    [0-9])
        echo "$var"是数字
        ;;
    [a-zA-Z])
        echo "$var"是字符
        ;;
esac
```

3. 如果多个没有关系的选项，想要执行同一条语句，可以通过 **按位或** 连接

```
#!/bin/bash

read -p "请输入>>>" var

case $var in
    "wuwuwu"|"yingyingying")
        echo "哭唧唧"
        ;;
    "hahaha"|"hehehe")
        echo "笑嘻嘻"
        ;;
esac
```

## 练习

- 1.实现四则运算。如输入 4+5 输出9。
- 2.输入一个字符，判断该字符是否为元音（a o e i u, A O E I U） [aeiouAEIOU] a|e|i|o)
- 3.写一个脚本，要求提示输入软件名。然后提示是否确认下载该软件  
如果确认下载，输入 y 或者 yes 或者Yes 开始下载  
如果输入其他则不下载

## 第1题

```
#!/bin/bash

read -p "请输入运算式子" var1 char var2

case $char in
    "+")
        echo "$var1 $char $var2 = $((var1+var2))"
        ;;
    "-")
        echo "$var1 $char $var2 = $((var1-var2))"
        ;;
    \*)
        #或者写成 "*"
        echo "$var1 $char $var2 = $((var1*var2))"
        ;;
```

```
*)
    echo "$char输入错误"
    ;;
esac
```

### 第3题

```
#!/bin/bash

read -p "请输入要下载的软件名>>" app
read -p "请确认要下载$app软件: y|Y|yes>>" choose

case $choose in
    [yY]|"yes"|"Yes")
        sudo apt-get install $app

        if [ $? -eq 0 ]
        then
            echo "$app安装成功"
        else
            echo "$app安装失败"
        fi
        ;;
    *)
        echo "取消安装"
        ;;
esac
```

## 【9】 shell中的循环语句

### 1) while

#### i. 格式

```
while [ 判断条件 ]
do
    循环语句
done
```

```
1 #!/bin/bash
2
3 i=0
4 sum=0
5
6 while [ $i -le 100 ]
7 do
8     ((sum+=i))
9     i=$((i+1))
10 done
11
12 echo "sum=$sum"
13
22_while.sh [+] 8,12-15 顶端

ubuntu@ubuntu:5_shell脚本 $ bash 22_while.sh
sum=5050
ubuntu@ubuntu:5_shell脚本 $

1 #include <stdio.h>
2
3 int main(int argc, const char *argv[])
4 {
5     int i = 0;
6     int sum = 0;
7     while(i<=100 )
8     {
9         sum+=i;
10        i++;
11    }
12    printf("sum=%d\n", sum);
13
14    return 0;
15 }
16 }

1.c

ubuntu@ubuntu:5_shell脚本 $ gcc 1.c
ubuntu@ubuntu:5_shell脚本 $ ./a.out
sum=5050
ubuntu@ubuntu:5_shell脚本 $
```

#####

## ii.死循环

```
while [ 1 ]
do
    循环语句
done
-----
while true
do
    循环语句
done
-----
while ((1))
do
    循环语句
done
```

## 练习

1. 外部输入金字塔层数，打印对应层数的金字塔

```

*      spc 3  *  1
***      2    3
*****    1    5
*****    0    7
```

2. 打印99乘法表

```

1 x 1 = 1
2 x 1 = 2  2 x 2 = 4
3 x 1 = 3  3 x 2 = 6  3 x 3 = 9
```

```
#!/bin/bash

read -p "请输入层数>>>" var

i=0
while [ $i -lt $var ]
```

```

do
    j=0
    while [ $j -lt $(($var-1-$i)) ]
    do
        echo -n " "          #输出一个空格是不会换行的，
        ((j++))
    done

    k=0
    while [ $k -lt $(($i*2+1)) ]
    do
        echo -n "*"
        ((k++))
    done

    echo -ne "\n"

    ((i++))
done

```

## 2) for

### i. C语言风格

```

for ((i=0;i<10;i++))
do
    循环语句
done
-----
#!/bin/bash

sum=0
for ((i=0;i<=100;i++))
do
    ((sum+=i))
done

echo "sum=$sum"

-----
死循环格式
for ((;;))
do
    循环语句
done

```

### ii. shell风格

```

for 循环变量名 in 选项列表
do
    循环语句
done

```



```

11 i=0
12 for var in "hahaha" "hehehe" "heiheihei"
13 do
14     echo "var=$var i=$i"
15     ((i++))
16 done
17
18
var=hehehe i=1
var=heiheihei i=2
ubuntu@ubuntu:5_shell脚本 $ bash 23_for.sh
var=hahaha i=0
var=hehehe i=1
var=heiheihei i=2
ubuntu@ubuntu:5_shell脚本 $

```

运行顺序：

1. 先将选项列表中的第一个数据赋值给循环变量，在执行循环语句
2. 接着选项列表中的第二个数据赋值给循环变量，在执行循环语句
3. 直到列表中的所有数据遍历完毕，退出循环。

### (1) 连续序列表

```

$(seq 1 100)    --->1~100的连续整数序列
{1..100}        --->1~100的连续整数序列

```

```

sum=0
for var in $(seq 1 100)
do
    ((sum+=var))
done
echo "sum=$sum"

```

### (2) 跟其他命令的结果

```

for name in `ls`
do
    echo $name
done

echo `ls`

```

### (3) 省略in和选项列表

```

for 循环变量名
do
    循环语句
done

```

当省略了in和选项列表后，**列表默认从位置变量获取，从1号位置开始获取**；

```
for var
do
    echo "var=$var"
done
-----
bash 1.sh aaa bbb
var=aaa
var=bbb
```

## 练习

从终端输入一串字符串，求出空格个数。

```
#!/bin/bash

read -p "请输入带空格的字符串>>>" str

len=${#str}

spc=0

for ((i=0;i<len;i++))
do
    char=${str:$i:1}

    if [ "$char" == " " ]
    then
        ((spc++))
    fi

done

echo spc=$spc
```

## 3) select-in

```
select 变量名 in 选项列表
do
    循环语句
done
-----#!/bin/bash


echo "你最喜欢什么颜色>>>"

select color in red green yellow blue black
do
    echo color=$color
done

echo "end of shell"
```

```
1 #!/bin/bash
2
3 echo "你最喜欢什么颜色>>>"
4
5 select color in red green yellow blue black
6 do
7     echo color=$color
8 done
9
10 echo "end of shell"
11
```

```
ubuntu@ubuntu:5_shell脚本 $ bash 24_select.sh
你最喜欢什么颜色>>>
1) red
2) green
3) yellow
4) blue
5) black
#? 1
color=red
#? 6
color=
#?
end of shell
ubuntu@ubuntu:5_shell脚本 $
```



1. 会将选项列表生成界面，供于选择
2. 按下ctrl + d按键，会退出select..in循环，继续往后执行
3. 输入的选项不在选项列表中，会给变量赋空值
4. 如果输入的选项是空选项，会将界面再重新打印一遍

## 练习

练习：用select...in删除当前文件夹下的文件。如果文件是.c文件则删除，如果不是.c文件就打印文件名

```
select file in $(ls)    # `ls`
do
    if [ "${file:0-2}" == ".c" ]
    then
        rm -rf $file
    else
        echo "$file文件不是.c结尾的文件"
    fi
done
```

## 练习

1. 无符号整数，不考虑翻转后超出范围的情况，你需要将这个整数中每位上的数字进行反转。

输入：123 输出321

输入：45678 输出87654

```
read inter
len=${#inter}

for ((i=0;i<len;i++))
do
    str="${inter:i:1}$str"
done

echo "str = $str"
```

2. 从终端输入字符串，求出小写字母的个数，大写字母的个数，数字的个数。

3. 要求输入年月日，输出这是今年的第几天 ---->选做

如果起始字符和结束字符是字母，涉及**本地语序**

本地语序默认的顺序是：aAbBcC..zZ

可以修改环境变量 LC\_ALL 清空本地语序 `export LC_ALL=C`

`[a-z]` 通配 a-z 中的任意一个字符

`[A-Z]` 通配 A-Z 中的任意一个字符

恢复本地语序 使用 `unset LC_ALL`

## 【10】break 和 continue

### 1) break

```
break          跳出当前循环
break n        指定向外跳n层
               break 2
```

当n大于循环嵌套的层数，则直接跳出最外层循环；

```
#!/bin/bash

while true
do
    echo "外层循环-----"

    i=0
    while true
    do
        ((i++))

        if [ $i -gt 10 ]
        then
            break 10
        fi

        echo "i=$i"

    done

    echo "外层循环"
    sleep 1
done

echo "结束循环"
```

## 2) continue

**continue** 强制结束当前层的本次循环，进入下一次循环

**continue n** 强制结束n层的本次循环，进入下一次循环

当n大于循环嵌套的层数，则直接结束最外层的本次循环，进入下一次循环

```
#!/bin/bash

for ((j=0;j<10;j++))
do
    echo "外层的循环j=$j-----"
    sleep 1
    for ((i=0;i<10;i++))
    do
        if [ $i -gt 3 ] && [ $i -lt 7 ]
        then
            continue 2
        fi

        echo "i=$i"

    done
    echo "外层的循环j=$j"
done

echo 结束循环
```

## 【11】shell中的函数

### 1) 格式

**function** 函数的名字() 函数定义的时候，不需要写参数列表

```
{
    函数的功能实现
}

-----

function show()
{
    echo "this is function"
}
```

## 2) 函数的调用

函数定义必须在函数调用之前，因为shell脚本是从上往下运行的。

### i. 不传参的格式

直接写函数的名字，即可完成函数的调用，

```
1 #!/bin/bash
2
3 function show()
4 {
5     echo "this is function"
6 }
7
8 echo "start of shell file"
9
10 show
11
12 echo "end of shell file"
13
```

```
ubuntu@ubuntu:5_shell脚本 $ bash 27_function.sh
start of shell file
this is function
end of shell file
ubuntu@ubuntu:5_shell脚本 $
```

### ii. 需要传参的格式

调用： 函数名 参数1 参数2 参数n

函数体内部接收参数：\$1 \$2 \$3 \$n @\$\* \$#等位置变量来接收参数

```
1 #!/bin/bash
2
3 function show()
4 {
5     #函数体内部的位置变量，用来接收函数调用的时候传递过来的参数
6     echo "1:$1"
7     echo "2:$2"
8     echo "3:$3"
9
10    echo "this is function"
11 }
12
13 function func()
14 {
15     echo "1:$1"
16     echo "2:$2"
17     echo "3:$3"
18     echo "@:$@"
19     echo "*:~*"
20     echo "#:~#"
21 }
22
23
24 echo "start of shell file"
25 echo -ne "\n"
26
27 show $1 $2 $3 #函数体外部的变量，用来接收shell脚本运行时候的外部传参
28 echo -ne "\n"
29
30 func 'a' "hello" "world"
31 echo -ne "\n"
32
33 echo "end of shell file"
~
27_function.sh 22,0-1 全部
```

```
ubuntu@ubuntu:5_shell脚本 $ bash 27_function.sh aa bb cc
start of shell file
1:aa
2:bb
3:cc
this is function
1:a
2:hello
3:world
@a: a hello world
~: a hello world
#: 3
end of shell file
ubuntu@ubuntu:5_shell脚本 $
```

### iii. @\$\* "\$@" "\$\*"

`@$* "$@"`：接收所有参数，将传入的每个参数，都当做**独立的个体**

`"$*"`：接收所有参数，并将**所有参数当做一个整体**，

```
function func1()
{
    select var in $*    #@$* "$@"
    do
        echo "$var"
    done
}
```

```
func1 'a' "hello" "world"
```

结果:

```
1) a
2) hello
3) world
#?
```

---

```
function func1()
{
    select var in "$*"
    do
        echo "$var"
    done
}

func1 'a' "hello" "world"
-----
```

结果:

```
1) a hello world
```

```
#!/bin/bash
```

```
function athmatic()
{
    case "$var2" in
        +)
            echo "$var1 $var2 $var3 = $((var1+var3))"
            ;;
        -)
            echo "$var1 $var2 $var3 = $((var1-var3))"
            ;;
        \*)
            echo "$var1 $var2 $var3 = $((var1*var3))"
            ;;
        /)
            echo "$var1 $var2 $var3 = $((var1/var3))"
            ;;
        *)
            echo "输入错误"
            ;;
    esac
}
```

```
read -p "请输入式子>>>" var1 var2 var3
```

```
athmatic
```

### 3) 返回值

1. shell脚本中所有变量默认都是全局的，函数体内部可以直接修改，直接打印
2. 如果一定想要一个返回值，可以使用 return 返回，可以返回返回值，或者结束函数  
如果没有return ,或者有return , 但是后面不写数据，则返回最后一个指令的运行状态，成功为0 ;\$?  
return 后面可以指定参数，但是这个**参数的范围是：0-255**，一般默认，返回0代表函数运行成功，其余是函数运行失败。
3. 接收返回值用：\$?

```
#!/bin/bash

function func()
{
    rm sub -r
}

func
echo "func函数运行结果 $?"

function func1()
{
    rm sub
    return 256
}
func1
echo "func1函数运行结果 $?"
```

## 9. makefile

### 【1】什么是makefile

1. Makefile是一个工程管理工具，本质上是一个文件，文件中存放的是**代码编译规则**
2. Makefile会根据文件的 **时间戳** 来决定工程内的文件是否需要编译。  
时间戳：文件修改的时间

### 【2】makefile书写格式

```
gcc -E xxx.c -o xxx.i    预处理
gcc -S xxx.i -o xxx.s    编译
gcc -c xxx.s -o xxx.o    汇编      gcc xxx.c -c -o xxx.o
-----
gcc xxx.o xxx1.o -o a.out 链接
```



## 1) 格式

在指定目录下创建一个名字为：Makefile 或者 makefile 的文件

makefile文件格式：

|             |                  |
|-------------|------------------|
| 目标文件1:依赖文件1 | 最终要生成的文件放在顶行     |
| <Tab>命令1    | 注意，前面是tab按键，不是空格 |
| <Tab>命令2    |                  |
|             |                  |
| 目标文件2:依赖文件2 |                  |
| <Tab>命令1    | 注意，前面是tab按键，不是空格 |
| <Tab>命令2    |                  |
|             |                  |
| 目标文件n:依赖文件n |                  |
| <Tab>命令1    | 注意，前面是tab按键，不是空格 |
| <Tab>命令2    |                  |
|             |                  |
| 指令1:        |                  |
| <Tab>命令1    |                  |
|             |                  |
| 指令2:        |                  |
| <Tab>命令2    |                  |

执行makefile文件：在终端输入make

执行makefile文件中的指令：在终端输入make 指令名

```
1 #ifndef MYSTRING_H
2 #define MYSTRING_H
3
4 void mystringcpy(char*, char*);
5
6 #endif
mystring.h 3,0-1 全部
1 #include <stdio.h>
2
3 void mystringcpy(char*dst, char*src)
4 {
5     char* ptr = dst;
6     while((*dst++=*src++) != '\0');
7     return;
8 }
9
10
mystring.c 3,29 全部
1_main.c 9,0-1
1 #include <stdio.h>
2 #include "../mystring.h"
3
4 int main(int argc, const char *argv[])
5 {
6     char s1[20] = "";
7     char s2[20] = "hello world";
8
9     mystringcpy(s1, s2);
10    printf("s1=%s\n", s1);
11
12    return 0;
13 }
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
251
```

执行时候使用 `make -f makefile1`    `make clean -f makefile1`

### 3) @: 取消回显

如果不想把命令显示在终端上, 可以在命令前面加上@

```
1 main:1_main.o mystring.o
2   gcc 1_main.o mystring.o -o main
3
4 1_main.o:1_main.c
5   @gcc 1_main.c -c -o 1_main.o
6
7 mystring.o:mystring.c
8   @gcc mystring.c -c -o mystring.o
9
10 clean:
11   @rm *.o main
12
makefile                                11,2-5      全部
"makefile" 12L, 189C 已写入
```

```
ubuntu@ubuntu:6_makefile $ make
gcc 1_main.o mystring.o -o main
ubuntu@ubuntu:6_makefile $ ls
1_main.c 1_main.o  main  makefile  mystring.c  mystring.h  mystring.o
ubuntu@ubuntu:6_makefile $ make clean
ubuntu@ubuntu:6_makefile $ ls
1_main.c  makefile  mystring.c  mystring.h
ubuntu@ubuntu:6_makefile $
```

## 【4】makefile中的变量

### 1) 变量的引用

```
shell:
    $变量名
    ${变量名}
makefile:
    $(变量名)
```

```
1 buf="wangwu"
2 name=$(buf)
3
4 print:
5   @echo name=$(name)
```

```
ubuntu@ubuntu:6_makefile $ make print -f makefile1
echo name="wangwu"
name=wangwu
ubuntu@ubuntu:6_makefile $ make print -f makefile1
name=wangwu
ubuntu@ubuntu:6_makefile $
```

### 2) "=" 最终赋值

会将变量在makefile中的所有赋值都找到, 然后将 **最后一次** 的赋值当做结果赋值给变量

```
1 buf="wangwu"
2 name=$(buf)
3
4 print:
5   @echo name=$(name)
6
7 buf="lisi"
8 buf="zhaoliu"
```

```
ubuntu@ubuntu:6_makefile $ make print -f makefile1
name=zhaoliu
ubuntu@ubuntu:6_makefile $
```

### 3) ":=" 立即赋值

```

1 buf:="wangwu"
2 name:=$(buf)
3
4 print:
5     @echo name=${name}
6
7 buf:="lisi"
8 buf:="zhaoliu"

```

```

ubuntu@ubuntu:6_makefile $ make print -f makefile1
name=zhaoliu
ubuntu@ubuntu:6_makefile $ make print -f makefile1
name=wangwu
ubuntu@ubuntu:6_makefile $

```

#### 4) "+" 追加赋值

```

1 name:="wangwu"
2 name+="lisi"
3
4 print:
5     @echo name=${name}
6

```

```

ubuntu@ubuntu:6_makefile $ make print -f makefile1
name=wangwu lisi
ubuntu@ubuntu:6_makefile $

```

#### 5) "?" 询问赋值

如果变量为空，则赋值，如果变量中有数据，则不赋值

## 【4】简化makefile流程

### 1) 通过变量赋值进行简化

```

1 obj:=1_main.o mystring.o
2 Target:=text
3
4 CC:=gcc
5 CAN:= -c -o
6
7 $(Target):$(obj)
8     $(CC) $(obj) -o $(Target)
9
10 1_main.o:1_main.c
11     $(CC) 1_main.c $(CAN) 1_main.o
12
13 mystring.o:mystring.c
14     $(CC) mystring.c $(CAN) mystring.o
15
16 clean:
17     rm -rf $(obj) $(Target)
18

```

### 2) 用特殊符号简化

\$@ 目标文件  
 \$^ 所有依赖文件  
 \$< 第一个依赖文件

```

1 obj:=1_main.o mystring.o
2 Target:=text
3
4 CC:=gcc
5 CAN:= -c -o
6
7 $(Target):$(obj)
8     $(CC) $^ -o $@
9
10 1_main.o:1_main.c
11     $(CC) $^ $(CAN) $@
12
13 mystring.o:mystring.c
14     $(CC) $^ $(CAN) $@
15
16 clean:
17     rm -rf $(obj) $(Target)
18

```

### 3) 用makefile通配符

```

1 obj:=1_main.o mystring.o
2 Target:=text
3
4 CC:=gcc
5 CAN:= -c -o
6
7 $(Target):$(obj)
8     $(CC) $^ -o $@
9
10 %.o:%.c
11     $(CC) $^ $(CAN) $@
12
13 clean:
14     rm -rf $(obj) $(Target)
15

```

### 4) 将变量集成到一个xxx.cfg的文件中

1. 创建一个xxx.cfg文件,
2. 将变量剪切到xxx.cfg文件中
3. 在makefile文件顶行包含一下这个.cfg文件

```
-include 路径/xxx.cfg
```

```
1 obj:=1_main.o mystring.o
2 Target:=text
3
4 CC:=gcc
5 CAN:= -c -o
6
~
~
~
~
~
~
~
~
~
~
makefile.cfg 6,0-1 全部
1 -include ./makefile.cfg
2
3 $(Target):$(obj)
4     $(CC) $^ -o $@
5
6 %.o:%.c
7     $(CC) $^ $(CAN) $@
8
9 clean:
10     rm -rf $(obj) $(Target)
11
~
~
~
~
~
~
makefile2 11,0-1 全部
```

## 5) 伪目标 .PHONY

伪目标主要是为了避免makefile中定义的 **执行指令** 和 **工作目录** 下的实际文件名出现名字冲突

当前目录下如果有一个名为“clean”的文件，执行make clean指令，因为clean是一个文件，并且没有依赖文件，所以后序的rm指令不会被执行。

解决方法：将makefile中将指令声明为伪目标即可，即不要将clean当做目标

```
.PHONY:clean
```

## 6) 最终版本

```
1 obj:=1_main.o mystring.o
2 Target:=text
3
4 CC:=gcc
5 CAN:= -c -o
6
~
~
makefile.cfg 3,0-1
1 -include ./makefile.cfg
2
3 $(Target):$(obj)
4     $(CC) $^ -o $@
5
6 %.o:%.c
7     $(CC) $^ $(CAN) $@
8
9 .PHONY:clean    #将clean做伪目标
10
11 clean:
12     rm -rf $(obj) $(Target)
13
~
~
~
makefile2 9,24-25
```