

# Weekly Research Progress Report

Student: Xing Cong

Date: 7/12/2023 - 7/25/2023

## List of accomplishments this week: (工作成果列表)

- 重新回顾并总结 Linux 相关知识
- 对 Shell 编程进行深入学习，整理相关笔记
- 全面学习 Fortran 编程的基本用法，基本掌握 Fortran 语言的语法和运行机制，整理相关笔记

## Paper summary: (文献总结)

无

## Work summary (工作总结)

整理相关笔记，已放到附录中

## Next (下一步)

- 集中精力研究 HYCOM 模型中的并行计算方法
- 找到 HYCOM 中多次重复计算的函数单元
- 结合源码和文献，进行更进一步的学习

# **Linux 基础知识回顾总结 与 Shell 编程基本用法总结**

# 0-虚拟机部分

## 虚拟机克隆

如果想要构建服务器集群，没有必要一台一台的去进行安装，只要通过克隆就可以。

快速获得多台服务器主要有两种方式，分别为：**直接拷贝操作**和**vmware的克隆操作**

### 直接拷贝

将之前安装虚拟机的所有文件进行拷贝，复制到另一个文件夹下。

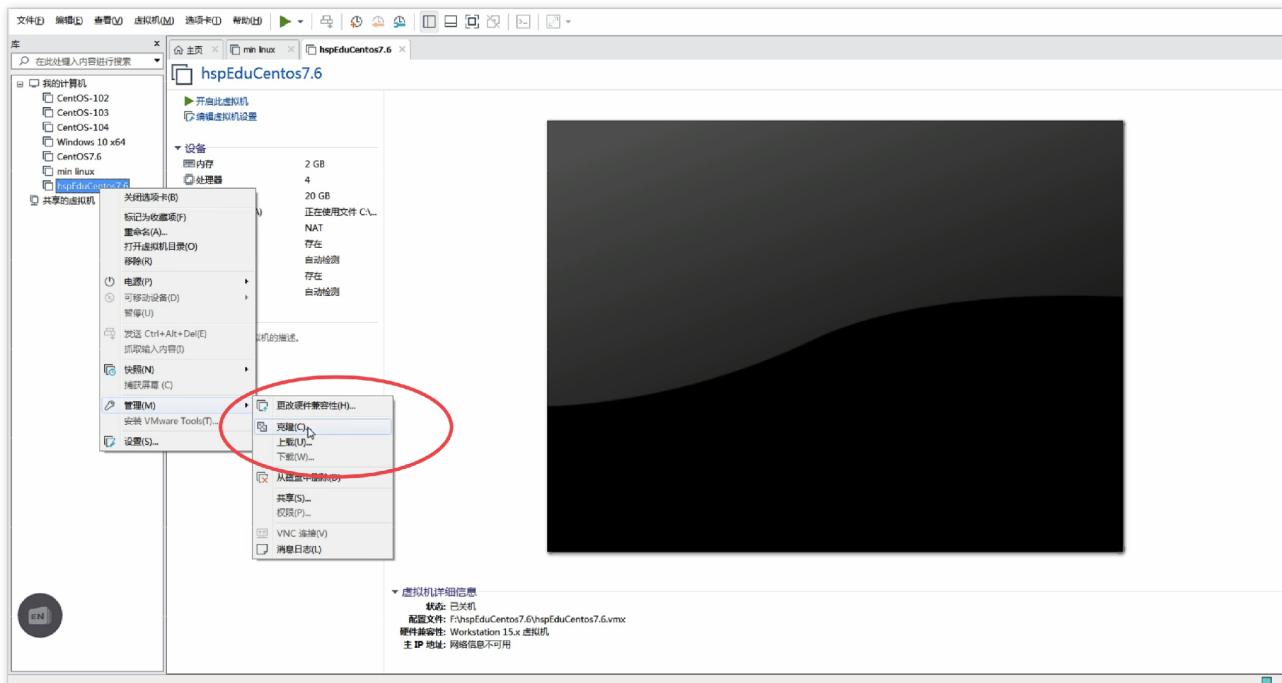
然后，通过vmware软件进行打开，即可。

### vmware克隆操作

注意：克隆时要关闭Linux虚拟机。

一般都是选择创建完整克隆。

克隆选项如下图所示。



## 虚拟机快照

虚拟机快照：相当于游戏中的存档和读档。

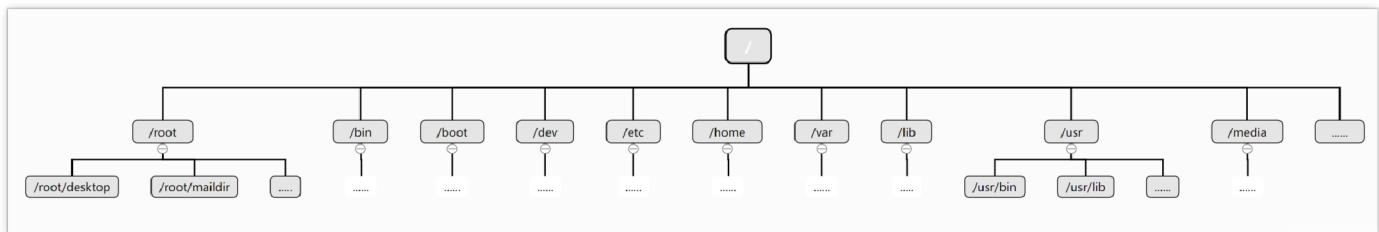
各个状态都可以进行转移。可以快速对出现故障的状态进行恢复，恢复到之前正常的状态。



# 1-Linux目录结构

Linux的目录结构是规定好的，不可以随意进行更改！

Linux的文件系统是采用级层式的树状目录结构，最上层是根目录--**/**，然后再在根目录下创建其它的目录。



各个目录中主要负责的功能和作用如下：（**主体的结构一定要知道！！！**）

- **/bin** ★
  - 是Binary的缩写，这个目录里存放着最经常使用命令。
- **/sbin**
  - 这里的s指的是Super User的意思，在这个文件夹下存放着系统管理员使用的系统管理程序。
- **/home** ★
  - 存放普通用户的主目录
  - 在Linux中每个用户都有一个自己的目录，一般该目录名是以用户的账号名进行命名的。
- **/root** ★
  - 该目录是系统管理员的目录
  - 区别与普通用户的是，该目录相当于是超级权限者的用户主目录。
- **/lib**
  - 这个是系统开机所需要的最基本的动态连接共享库。
  - 作用类似于win中的dll文件。
  - 几乎所有的程序都需要用到这些共享库。
- **/lost + found**
  - 该目录下一般没有内容
  - 当系统出现非法关机时，这里将会存放一些文件。

- **/etc** ★

- 里面存放所有系统管理所需要的配置文件和子目录
- 比如： config 文件
- 它是 "etcetera" 的缩写，意思是 "其他"。这些文件包括 **网络配置、用户账户信息、服务配置、软件包管理** 等等。

- **/usr** ★

- 非常重要的目录
- 用户的很多应用程序和文件都会放在该目录下面，类似于 win 中的 program files 目录
- Linux 中的 /usr 约等于 win 中的 program files 目录，也就是说如果安装应用程序，会默认安装到该文件下面。

- **/boot** ★

- 存放启动 Linux 时使用的一些核心文件，包括一些核心文件以及镜像文件。

- **/proc** !

- 它是一个动态生成的目录，其中的文件和子目录都是虚拟的，并不占用实际的内存空间。
- 其中的内容主要为，系统运行时的进程和内核相关的信息。
- 比如：在该目录下，每个运行的进程都有一个以数字命名的子目录，代表该进程的 ID（也就是操作系统中所学习的 **PID**）

- **/srv** !

- service
- 存放一些服务启动之后，需要提取的数据
- 存储特定服务的数据文件、配置文件、日志文件等。

- **/sys** !

- 是一个虚拟文件系统
- 提供对系统硬件和内核参数的访问
- 在 sys 目录下，可以找到与系统硬件设备相关的信息，比如 CPU、内存、磁盘等。可以通过读取这些文件来获取硬件设备的详细信息。比如设备型号、驱动程序、状态等。

- **/tmp**

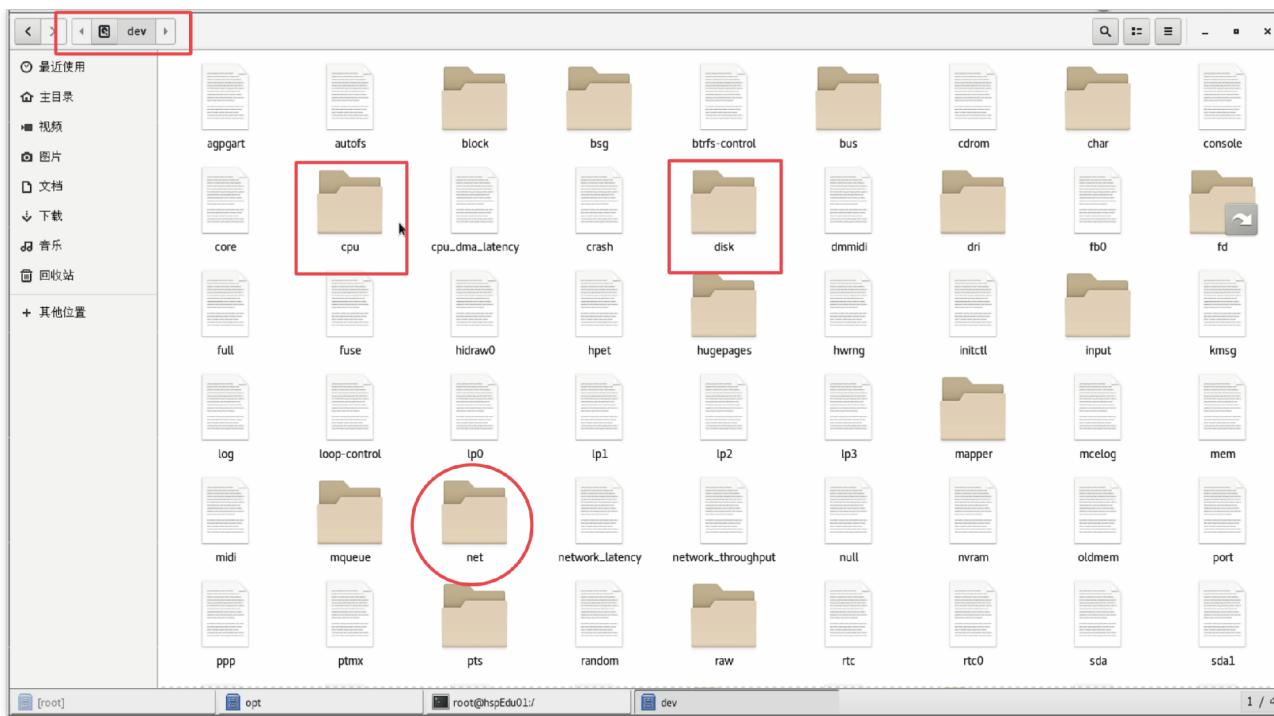
- 存放一些临时文件

- **/dev**

- 虚拟文件系统，用于表示的访问系统中的设备
  - 类似于win中的设备管理器， 把所有的硬件用文件的形式进行存储
  - 通过/dev目录中的设备文件， 用户和应用程序可以与硬件设备进行交互， 读取和写入数据， 进行输入和输出操作。
- **/media** 
    - 用于挂载可移动介质的目录
    - linux系统会自动识别一些设备， 例如U盘等， 可以通过该目录访问和操作U盘中的文件
    - 当识别后， 将会把该设备挂载在该文件下。
  - **/mnt** 
    - 用于挂载临时文件系统的目录
    - 比如外部存储设备、 网络共享等。
    - 比如， 当需要访问到这些文件系统时， 可以将其挂载到该目录上， 方便在文件系统中进行访问和操作
  - **/opt**
    - 这是给主机进行额外安装软件的软件包所存放的目录
  - **/usr/local** 
    - 给主机额外安装软件所安装的目标目录。
    - 就是将软件安装到该目录下。 否则可能将会默认安装到/usr下。
    - 一般是通过编译源码方式安装的程序
  - **/var** 
    - 用于存放系统运行过程中产生的可变数据， 包括日志文件、 缓存文件、 临时文件、 数据库文件等。
    - 也就是说， 这些文件可能会随着系统的运行而不断变换（可变数据）
  - **/selinux**
    - 安全子系统
    - 控制程序只能访问特定文件

在Linux的世界里面，一切皆文件！！！

因为， Linux不单单将我们熟悉的文件看作是文件， 而且， 它会将硬件当作一个文件去处理， 就放在**/dev**目录下面。



1 / 4

## 2-vi和vim的使用

### vi和vim的区别

- vi 是linux系统中内置的文本编辑器
- vim具有程序编辑能力

### vi和vim常用的三种模式

#### • 正常模式

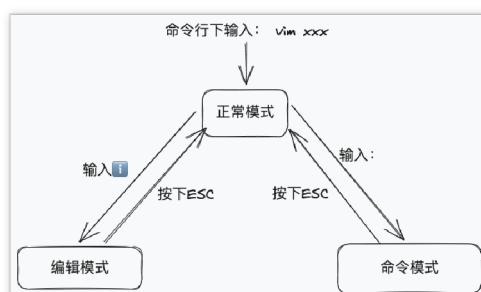
- 使用vim打开一个文件，就默认进入正常模式
- 可以使用方向键【上下左右】来移动光标
- 可以使用【删除字符/删除整行】来处理文件内容
- 也可以使用【复制/粘贴】快捷键

#### • 插入模式

- 一般来说，在正常模式下，按下*i*或者*I*，也就是insert的首字母，即可进入插入模式。
- 插入模式=编辑模式，可以进行输入操作

#### • 命令行模式

- 进入命令行模式步骤：首先按下*esc*键，再输入：`:`即可进入。
- 可以提供相关指令，完成读取、存盘、替换、离开vim编辑器、显示行号等操作
- 输入含义
  - `:wq` 保存并退出
  - `:q` 退出
  - `:q!` 强制退出，不保存



## 3-Linux 实操

# 开关机、重启、用户登陆注销

## 关机&重启

- 关机&重启之前的操作
  - `sync` 把内存的数据同步到磁盘上
- 关机指令
  - `shutdown -h now` 立刻关机
  - `shutdown -h 1` 1分钟后关机
  - `halt` 关机
- 重启指令
  - `shutdown -r now` 立刻重新启动计算机
  - `reboot` 重启

## 用户登陆和注销

注销只能在shell环境下进行使用

- 登陆
  - 登陆普通用户 `su - 用户名`
  - 登陆root用户 `sudo su` 或者 `su -root`
- 注销
  - 退出当前用户 `logout`

## 用户管理

Linux系统是一个多用户多任务的操作系统，任何一个要使用系统资源的用户，都必须首先想系统管理员申请一个账号，然后通过这个账号再进入系统。

## 添加用户

SHELL

**useradd** 用户名

添加一个系统操作用户，当用户创建成功后，会自动在home目录下创建和用户同名的目录

SHELL

**useradd -d 指定目录 新的用户名**

给新创建的用户指定存储路径，而不是存储在 /home 目录下(-d d就是directory 目录的意思)

```
[root@cvm-3jzbcc25i225 home]# userdel -r congxing
[root@cvm-3jzbcc25i225 home]# ls
[root@cvm-3jzbcc25i225 home]# useradd cx
[root@cvm-3jzbcc25i225 home]# ls
cx
[root@cvm-3jzbcc25i225 home]#
```

## 修改用户密码

SHELL

**passwd** 用户名

# 如果不加用户名，则默认是修改当前用户的密码

```
[root@cvm-3jzbcc25i225 /]# passwd test1
Changing password for user test1.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
[root@cvm-3jzbcc25i225 /]#
```

## 删除用户

现在的系统用户列表如下所示：

```
[root@cvm-3jzbcc25i225 home]# ls
cc  cx  test1
```

- 删除用户但要保留用户文件

```
userdel 用户名
```

- 删除用户同时删除用户文件

```
userdel -r 用户名
```

## 查询用户信息

```
id 用户名
```

```
[root@cvm-3jzbcc25i225 ~]# id root  
uid=0(root) gid=0(root) groups=0(root)  
[root@cvm-3jzbcc25i225 ~]# █
```

## 切换用户

如果当前用户的权限不够，可以通过 `su - 用户名`，切换到高权限用户，比如root用户

```
su - 切换用户名
```

！！注意：

- 从权限高的用户切换到权限低的用户，不需要输入密码，如果从权限低的切换到权限高的用户，则需要输入切换的用户密码
- 当需要返回到原来的用户时，可以使用 `exit/logout` 指令，进行退出

## 查看当前用户

```
whoami 或者 who am I
```

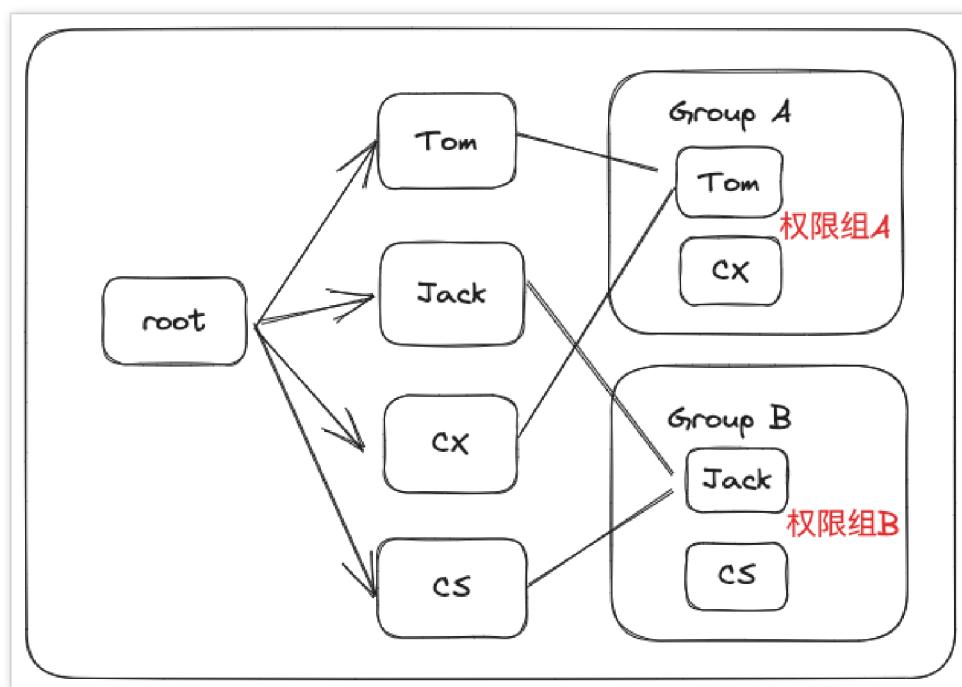
```
[root@cvm-3jzbcc25i225 ~]# whoami  
root  
[root@cvm-3jzbcc25i225 ~]# who am I  
root pts/2 2023-07-09 16:15 (218.12.19.143)  
[root@cvm-3jzbcc25i225 ~]# █
```

- **who am i** 这个指令，显示的是第一次登陆系统的用户，如果通过**su**指令进行了用户的切换，则仍然会是第一次登陆系统的用户。
- **whoami** 这个指令则会显示目前正在进行操作的用户

```
[root@cvm-3jzbcc25i225 ~]# su - cx
[cx@cvm-3jzbcc25i225 ~]$ who am i
root      pts/2        2023-07-09 16:15 (218.12.19.143)
[cx@cvm-3jzbcc25i225 ~]$ whoami
cx
[cx@cvm-3jzbcc25i225 ~]$
```

## 用户组的添加和删除

用户组的作用在于：系统可以对有共性【权限】的多用户进行统一的管理



- 新增组
  - **groupadd 组名**
- 删除组
  - **groupdel 组名**
- 增加用户时直接加上组
  - **useradd -g 用户组 用户名**

**！ ! 注意：** 如果在增加用户时，没有指定组，则会在创建用户时，同时创建一个名为用户名的组。

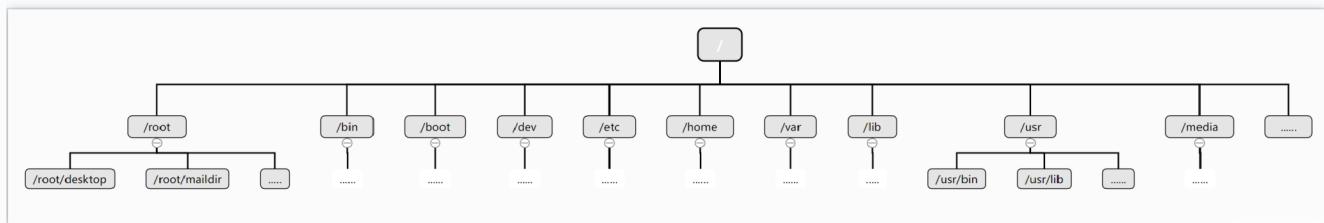
```
[root@cvm-3jzbcc25i225 ~]# id cx  
uid=1005(cx) gid=1005(cx) groups=1005(cx)  
[root@cvm-3jzbcc25i225 ~]#
```

- 修改用户的组
  - **usermod -g 用户组 用户名**

## 用户和组相关文件

Linux的目录结构是规定好的，不可以随意进行更改！

Linux的文件系统是采用级层式的树状目录结构，最上层是根目录--**/**，然后再在根目录下创建其它的目录。



各个目录中主要负责的功能和作用如下：（**主体的结构一定要知道！！！**）

- **/bin**

- 是Binary的缩写，这个目录里存放着最经常使用命令。

- **/sbin**

- 这里的s指的是Super User的意思，在这个文件夹下存放着系统管理员使用的系统管理程序。

- **/home**

- 存放普通用户的主目录
- 在Linux中每个用户都有一个自己的目录，一般该目录名是以用户的账号名进行命名的。

- **/root**

- 该目录是系统管理员的目录
- 区别与普通用户的是，该目录相当于是超级权限者的用户主目录。

- **/lib**

- 这个是系统开机所需要的最基本的动态连接共享库。
- 作用类似于win中的dll文件。

- 几乎所有的程序都需要用到这些共享库。
- /lost + found
  - 该目录下一般没有内容
  - 当系统出现非法关机时，这里将会存放一些文件。
- **/etc** ★
  - 里面存放所有系统管理所需要的配置文件和子目录
  - 比如： config文件
  - 它是"etcetera"的缩写，意思是"其他"。这些文件包括**网络配置、用户账户信息、服务配置、软件包管理**等等。
- **/usr** ★
  - 非常重要的目录
  - 用户的很多应用程序和文件都会放在该目录下面，类似于win中的program files 目录
  - Linux中的/usr 约等于 win中的program files 目录，也就是说如果安装应用程序，会默认安装到该文件下面。
- **/boot** ★
  - 存放启动Linux时使用的一些核心文件，包括一些核心文件以及镜像文件。
- **/proc** !
  - 它是一个动态生成的目录，其中的文件和子目录都是虚拟的，并不占用实际的内存空间。
  - 其中的内容主要为，系统运行时的进程和内核相关的信息。
  - 比如：在该目录下，每个运行的进程都有一个以数字命名的子目录，代表该进程的ID（也就是操作系统中所学习的**PID**）
- **/srv** !
  - service
  - 存放一些服务启动之后，需要提取的数据
  - 存储特定服务的数据文件、配置文件、日志文件等。
- **/sys** !
  - 是一个虚拟文件系统
  - 提供对系统硬件和内核参数的访问
  - 在sys目录下，可以找到与系统硬件设备相关的信息，比如CPU、内存、磁盘等。可以通过读取这些文件来获取硬件设备的详细信息。比如设备型号、驱动

程序、状态等。

- /tmp

- 存放一些临时文件

- /dev

- 虚拟文件系统，用于表示的访问系统中的设备
  - 类似于win中的设备管理器，把所有的硬件用文件的形式进行存储
  - 通过/dev目录中的设备文件，用户和应用程序可以与硬件设备进行交互，读取和写入数据，进行输入和输出操作。

- **/media** 

- 用于挂载可移动介质的目录
  - linux系统会自动识别一些设备，例如U盘等，可以通过该目录访问和操作U盘中的文件
  - 当识别后，将会把该设备挂载在该文件下。

- **/mnt** 

- 用于挂载临时文件系统的目录
  - 比如外部存储设备、网络共享等。
  - 比如，当需要访问到这些文件系统时，可以将其挂载到该目录上，方便在文件系统中进行访问和操作

- /opt

- 这是给主机进行额外安装软件的软件包所存放的目录

- **/usr/local** 

- 给主机额外安装软件所安装的目标目录。
  - 就是将软件安装到该目录下。否则可能将会默认安装到/usr下。
  - 一般是通过编译源码方式安装的程序

- **/var** 

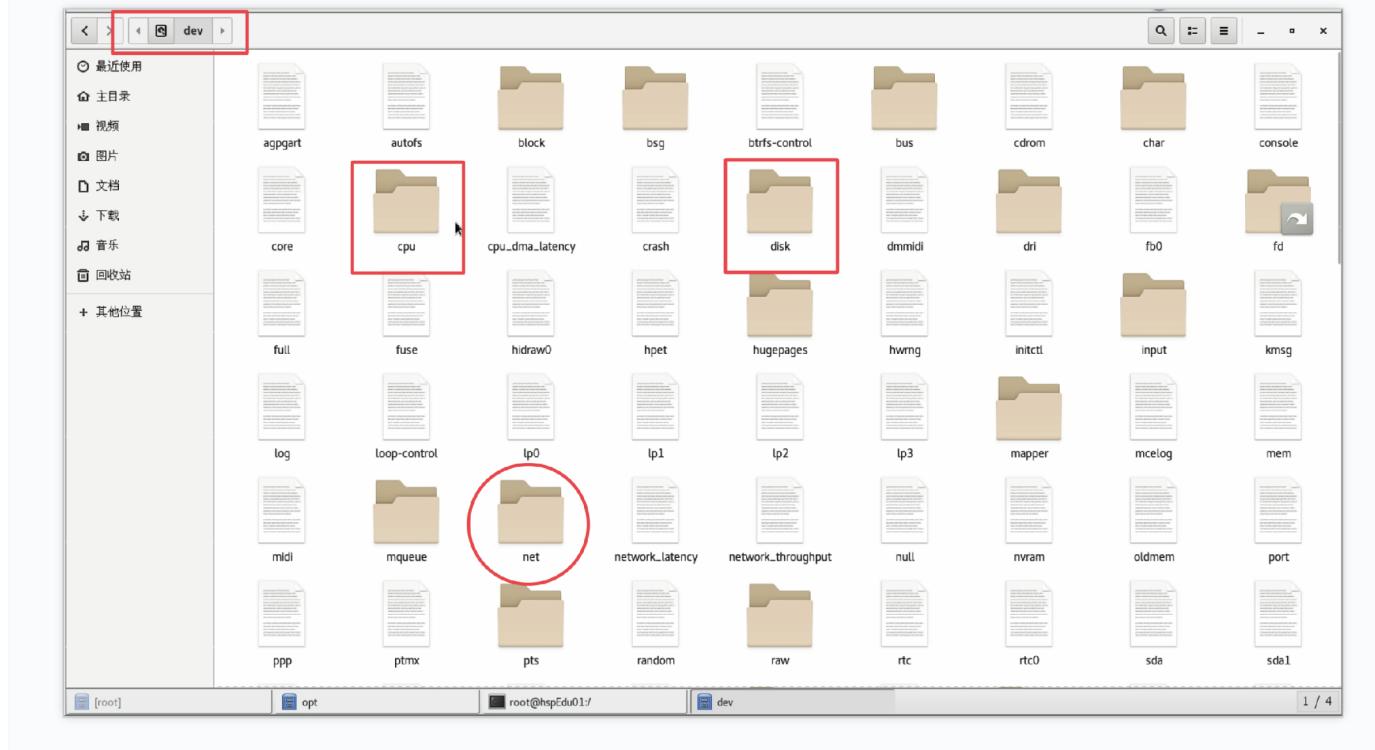
- 用于存放系统运行过程中产生的可变数据，包括日志文件、缓存文件、临时文件、数据库文件等。
  - 也就是说，这些文件可能会随着系统的运行而不断变换（可变数据）

- /selinux

- 安全子系统
  - 控制程序只能访问特定文件

在Linux的世界里面，一切皆文件！！！

因为，Linux不单单将我们熟悉的文件看作是文件，而且，它会将硬件当作一个文件去处理，就放在`/dev`目录下面。



### • /etc/passwd 文件

- 用户user的配置文件，记录用户的各种信息
- 每行信息的含义
- 用户名:口令:用户标识号:组标识号:注释性描述:主目录:登陆shell [【shell介绍】](#)

◦

```
[root@cvm-3jzbcc25i225 etc]# cat passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

### • /etc/shadow文件

- 口令的配置文件
- 每行的含义
- 登录名:加密口令:最后一次修改时间:最短时间间隔:最长时间间隔:警告时间:不活动时间:失效时间:标志

### • /etc/group文件

- 组的配置文件，记录Linux包含组的信息
- 每行的含义

- 组名:口令:组标识号:组内用户列表

！！注意：口令一般是不可见的，表示形式为x或为空。

# 实用指令

## 指定运行级别

运行级别	级别含义
0	关机
1	单用户状态 (找回丢失的密码)
2	多用户状态没有网络服务
3	多用户状态有网络服务
4	系统未使用保留给用户
5	图形界面
6	系统重启

## init 命令

通过init命令来切换不同的运行级别

比如： init [0123456] 然后关机，再启动即可进行切换

比如 init 0 表示关机； init 6 表示系统重启

## 帮助指令

- **man** 获得帮助信息
  - **man [命令或配置文件]**
  - 比如：查看 ls 命令的帮助信息：**man ls**
  - 选项可以进行组合使用，比如组合 ls -a 和 ls -l 为：ls -al 或 ls -la 都可以

```
[root@cvm-3jzbcc25i225 ~]# ls -la
total 44
dr-xr-x--. 4 root root 203 May 29 20:57 .
dr-xr-xr-x. 17 root root 224 Apr 22 2020 ..
-rw-----. 1 root root 6870 Apr 22 2020 anaconda-ks.cfg
-rw-----. 1 root root 669 Jul 9 17:34 .bash_history
-rw-r--r--. 1 root root 18 Dec 29 2013 .bash_logout
-rw-r--r--. 1 root root 176 Dec 29 2013 .bash_profile
-rw-r--r--. 1 root root 176 Dec 29 2013 .bashrc
-rw-r--r--. 1 root root 100 Dec 29 2013 .cshrc
drwxr-xr-x. 5 root root 89 May 29 21:04 ds-al-vue
-rw-----. 1 root root 41 Aug 18 2020 .lessht
-rw-----. 1 root root 6518 Apr 22 2020 original-ks.cfg
drwx-----. 2 root root 29 Aug 14 2020 .ssh
-rw-r--r--. 1 root root 129 Dec 29 2013 .tcshrc
[root@cvm-3jzbcc25i225 ~]# ls -l
total 16
-rw-----. 1 root root 6870 Apr 22 2020 anaconda-ks.cfg
drwxr-xr-x. 5 root root 89 May 29 21:04 ds-al-vue
-rw-----. 1 root root 6518 Apr 22 2020 original-ks.cfg
[root@cvm-3jzbcc25i225 ~]# ls -a
. anaconda-ks.cfg .bash_logout .bashrc ds-al-vue original-ks.cfg .tcshrc
.. .bash_history .bash_profile .cshrc .lessht .ssh
```

- **help 指令**

- 语法: **help 命令**
- 获得shell内置命令的帮助信息
- 查看cd命令的帮助信息

- [root@cvm-3jzbcc25i225 ~]# help cd  
cd: cd [-L|[-P [-e]]] [dir]  
Change the shell working directory.  
  
Change the current directory to DIR. The default DIR is the value of the  
HOME shell variable.

## 文件目录类

- **pwd 指令**

- 显示当前工作目录的绝对路径

- [root@cvm-3jzbcc25i225 ~]# pwd  
/root  
[root@cvm-3jzbcc25i225 ~]#

- **ls指令**

- **ls [选项] [目录或者文件]**
- 常用选项
  - a: 显示当前目录所有的文件和目录，包括隐藏的
  - l: 以列表的方式进行显示信息

- **cd 指令**

- 切换到指定目录
  - **cd [参数]**
    - cd ~ : 回到自己的家目录中
    - cd .. : 回到上一级目录

- **mkdir 指令**

- 创建目录
  - **mkdir [选项] 要创建的目录**
  - 常用选项
    - -p : 创建多级目录
  - ```
[root@hspEdu01 home]# mkdir /home/animal/tiger
mkdir: 无法创建目录"/home/animal/tiger": 没有那个文件或目录
[root@hspEdu01 home]# mkdir -p /home/animal/tiger
[root@hspEdu01 home]# ls
animal  dog  jack  milan  test  tom  zwj
[root@hspEdu01 home]# cd animal/
[root@hspEdu01 animal]# ls
tiger
[root@hspEdu01 animal]#
```

- **rmdir 指令**

- 删除空目录
  - **rmdir [选项] 要删除的空目录**
  - ! 删除的是空目录，如果目录下有内容时，则无法进行删除
- 删除非空目录

- **rm -rf 要删除的目录**
  - -r ( -R, --recursive): 递归地删除目录及其内容
  - -f ( --force ): 强制删除，忽略是否有文件和相关参数

- **touch 指令**

- 创建空文件
  - **touch 文件名**

- **cp 指令**

- cp 指令拷贝文件到指定目录
  - **cp [选项] source dest**
    - source 拷贝的源文件名

- dest 拷贝到的目的位置
- 常用选项
  - -r : 递归复制整个文件夹
- 注意 ! :
  - 强制覆盖不提示的方法, 在cp前面加上 \
  - 例如: \cp 源文件 目的文件
- rm 指令
  - 移除文件或目录
  - rm [选项] 要删除的文件或目录
  - 常用选项
    - -r ( -R, --recursive): 递归地删除目录及其内容
    - -f ( --force ): 强制删除, 忽略是否有文件和相关参数
  - 举例: 删除非空目录
    - rm -rf 要删除的目录
    - 强制删除不提示的方法: 带上 -f 参数即可
- mv 指令
  - 移动文件或目录 或 重命名
  - 重命名: mv oldname newname (在同一个目录下, 才可以进行重命名)
  - 移动文件: mv 移动文件 目标目录 (不在同一个目录下, 进行文件的移动操作)
- cat 指令
  - 查看文件的内容
  - cat [选项] 要查看的文件
  - 常用选项
    - -n : 显示行号
  - ! 注意:
    - 为了浏览方便, 一般会带上 管道命令 | 其他指令
    - 例如: cat -n /etc/profile | [其他命令]
- more 指令
  - 基于VI编辑器的文本过滤器, 可以以全屏幕的方式, 按页显示文本文件的内容。
  - more 要查看的文件
  - 使用more以后, 可以使用的交互指令:

| 操作          | 功能说明                     |
|-------------|--------------------------|
| 空白键 (space) | 代表向下翻一页；                 |
| Enter       | 代表向下翻『一行』；               |
| q           | 代表立刻离开 more , 不再显示该文件内容。 |
| Ctrl+F      | 向下滚动一屏                   |
| Ctrl+B      | 返回上一屏                    |
| =           | 输出当前行的行号                 |
| :f          | 输出文件名和当前行的行号             |

### • less 指令

- 分屏查看文件内容，less指令在显示文件内容时，并不是将整个文件加载之后才显示，而是根据显示需要，加载的内容，**对于显示大型文件具有较高的效率！**
- less 需要显示的文件**
- 使用less以后，可以使用的交互指令：

| 操作         | 功能说明                       |
|------------|----------------------------|
| 空白键        | 向下翻动一页；                    |
| [pagedown] | 向下翻动一页                     |
| [pageup]   | 向上翻动一页；                    |
| /字串        | 向下搜寻『字串』的功能；n：向下查找；N：向上查找； |
| ?字串        | 向上搜寻『字串』的功能；n：向上查找；N：向下查找； |
| q          | 离开 less 这个程序；              |

- 当输入**/要查找的字符串**时，查找到以后，输入n，可以继续向下查找匹配的字符串，输入N，可以继续向上查找匹配的字符串
- 当输入**?要查找的字符串**时，查找到以后，输入n，可以继续向上查找匹配的字符串，输入N，可以继续向下查找匹配的字符串

### • echo 指令

- 输入内容到控制台
- echo [选项] [输出内容]**

### • head 指令

- 用于显示文件的开头部分内容，默认情况下head指令**显示文件的前10行**
- 基本语法

- **head** 文件名
  - **head -n 5 文件名** 查看文件头5行内容
- **tail** 指令
    - 用于输出文件尾部内容， 默认情况下， tail指令显示文件的前10行内容
    - 基本语法
      - **tail** 文件 查看尾部后10行的内容
      - **tail -n 5 文件** 查看尾部后5行的内容
      - **tail -f 文件** 实时追踪该文件的更新
- > 指令
    - 输出重定向指令
    - 基础语法
      - **ls -l > 文件** 将列表的内容写入文件
      - **cat file1 > file2** 将文件1的内容覆盖到文件2中
- >> 指令
    - 追加指令
    - 基础语法
      - **ls -al >> 文件** 将列表内容追加到文件的末尾
      - **echo 内容 >> 文件** 在文件尾部追加内容
- **ln** 指令
    - 软链接， 也称为符号链接， 类似于windows中的快捷方式。
    - **ln -s [原文件或目录] [软链接名]**
      - 给一个原文件创建一个软链接
    - [root@cvm-3jzbcc25i225 home]# **ln -s /root/ /home/myroot**  
[root@cvm-3jzbcc25i225 home]# **ls**  
**cx myroot**  
[root@cvm-3jzbcc25i225 home]# **ls -l**  
**total 0**  
**drwx----- . 2 cx cx 83 Jul 9 17:34 cx**  
**lrwxrwxrwx . 1 root root 6 Jul 15 20:24 myroot -> /root/**
- **history** 指令
    - 查看已经执行过的历史命令， 也可以执行历史命令
    - **history** 显示所有的历史命令
    - **history 10** 显示最近的十条指令
    - **!5** 执行历史编号为5的指令

# 时间日期类

写shell脚本输出日志时可能会用到

- **date** 指令

- 显示当前日期 ◦ 基本语法
  - `date` 显示当前日期
  - `date "+%Y"` 显示当前年份
  - `date "+%m"` 显示当前月份
  - `date "+%d"` 显示当前的天
  - `date "+%Y-%m-%d %H:%M:%S"` 具体到今天的每一分，每一秒。
- 可选选项（设置日期）
  - `date -s 字符串时间`

- **cal** 指令

- 查看日历 

# 搜索查找类

- **find** 指令

- `find` 指令将从指定目录向下递归地遍历各个子目录，将满足条件的文件或目录显示在终端
- 基本语法：`find [搜索范围] [选项]`
- 选项说明
  - `-name 文件名` 按照指定的文件名查找文件
  - `-user 用户名` 查找属于指定用户名的所有文件
  - `-size 文件大小` 按照指定的文件大小查找文件
    - 注意
      - 大于 - 小于 = 等于
      - 单位有： k M G
- 比如：查找/home目录下的hello.txt文件
  - `find /home -name hello.txt`

- **locate** 指令

- locate指令可以快速定位文件路径， locate指令利用事先建立的系统中所有文件名称以及路径的locate数据库实现快速定位到给定的文件。locate指令无需便利整个文件系统，查询速度较快。
- 注意：为了保证查询结果的准确度，管理员必须定期更新locate时刻。
  - 基本语法：**locate 文件名称**
  - ：由于locate指令基于数据库进行查询，所以**第一次运行前，必须使用updatedb指令创建locate数据库**

### • **which**指令

- 可以查看某个指令在哪个目录下
- 例如：**which ls**

### • **find** vs **locate**

- find是在硬盘上查找
- locate是在数据库中查找

### • **grep**指令和管道符号 |

- grep 过滤查找
- 管道符号：用于将前一个命令的处理结果输出给后面的命令进行处理。
- 基本语法：**grep [选项] 查找内容 源文件**
- 常用选项
  - -n： 显示匹配行以及行号
  - -i： 忽略字母大小写
- 比如：在hello.txt 中，查找yes 所在行，并显示行号
  - 1 **cat hello.txt | grep "yes"**
  - 2 **grep -n "yes" hello.txt**

```
[root@cvm-3jzbcc25i225 home]# vim hello.txt
[root@cvm-3jzbcc25i225 home]# cat hello.txt | grep "yes"
yes
[root@cvm-3jzbcc25i225 home]# cat hello.txt | grep -n "yes"
5:yes
[root@cvm-3jzbcc25i225 home]# grep -n "yes" hello.txt
5:yes
[root@cvm-3jzbcc25i225 home]#
```

## 压缩和解压类

### • **gzip** 和 **gunzip**

- gzip：压缩文件， gunzip：解压文件

- 仅仅是对文件进行操作，不包含目录
- 基本语法：
  - **gzip 文件**：压缩文件，只能将文件压缩为\*.gz文件
  - **gunzip 文件.gz**：解压.gz文件

## • zip 和 unzip

- 解压和压缩文件
- 基本语法
  - **zip [选项] xxx.zip 要压缩的内容** 压缩文件和目录的命令
  - **unzip [选项] xxx.zip** 解压缩文件
- 常用选项
  - zip : -r (recursive) : 递归压缩 (**压缩目录**)
  - unzip: -d 目录：指定解压后文件的存放目录

## • tar 指令 ★★

- tar是打包指令，最后打包后的文件是.tar.gz的文件
- 基本语法
  - **tar [选项] xxx.tar.gz 打包的内容**
- 选项说明
  - -c 产生.tar 的打包文件(--creat)
  - -v 显示详细信息
  - -f 指定压缩后的文件名
  - -z 打包的时候同时压缩
  - - z, --gunzip, --gzip**  
(c mode only) Compress the resulting archive with gzip(1). In extract or list modes, this option is ignored. Note that this tar implementation recognizes gzip compression automatically when reading archives.
  - -x 解包.tar 文件(--extract)
- 压缩文件:**-z(gzip)c(产生打包文件)v(显示详细信息)f(指定压缩后的文件夹)**
- 解压文件:**-z(gunzip)x(解包.tar文件)v(显示详细信息)f(文件夹)**
- 指定压缩目录和解压目录 **-C**

**-C directory, --cd directory, --directory directory**  
In c and r mode, this changes the directory before adding the following files. In x mode, change directories after opening the archive but before extracting entries from the archive.

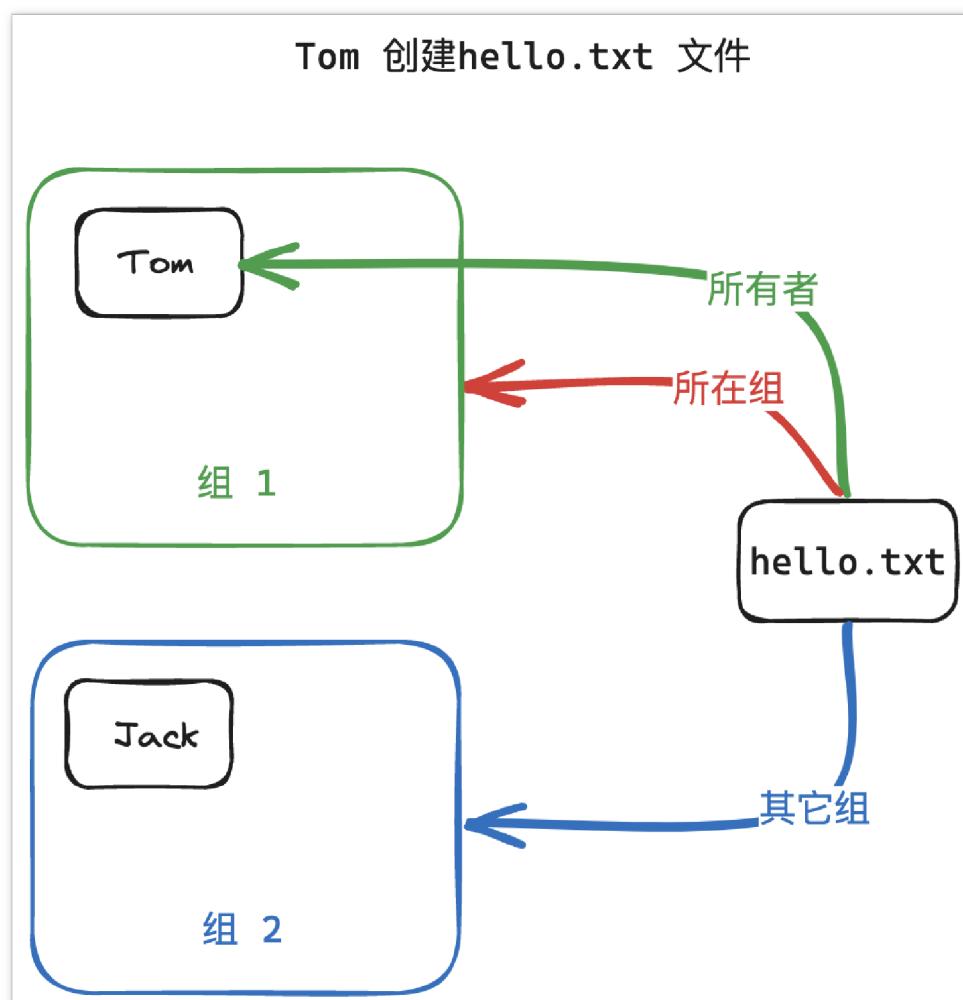
# 4-Linux 组管理和权限管理

## Linux组的基本介绍

在linux中，每个用户必须属于同一个组，不能独立于组外。

在linux中，每个文件有所有者、所在组、其它组的概念。

- 所有者
  - 一般为文件的创建者，谁创建了文件，就自然成为了该文件的所有者
- 所在组
- 其它组



## 文件/目录的所有者

- 查看文件的所有者
  - `ls -ahl`

- 修改文件所有者
  - `chown 用户名 文件名`

## 组的创建

- 创建组
  - `groupadd 组名`
- 创建一个用户，并放入monster组中 [用户组的相关操作](#)
  - `useradd -g monster fox`

## 文件/目录所在的组

当某个用户创建了一个文件，那么这个文件的所在组就是该用户所在的组。

- 查看文件/目录所在的组
  - 通过指令 `ls -ahl` 可以查看文件/目录所在的组
- 修改文件/目录所在的组
  - `chgrp 组名 文件名`

## 其它组

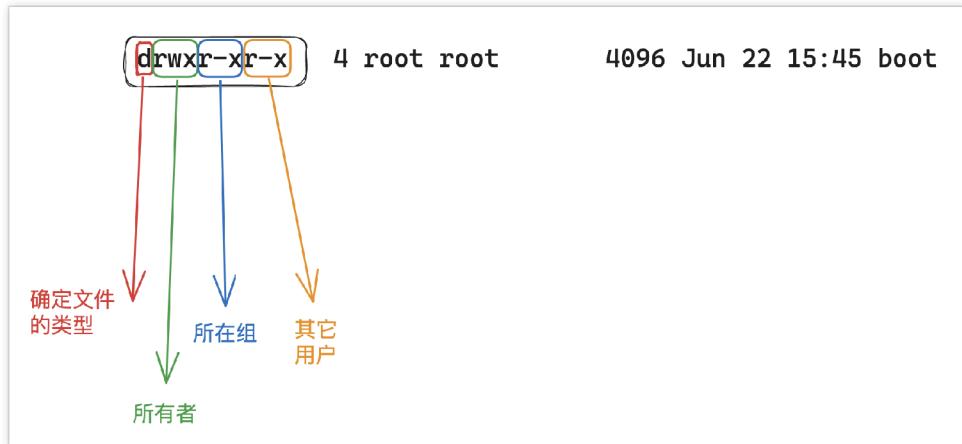
除了文件的所有者和所在组的用户外，系统的其它用户都是文件的其它组

## 改变用户所在的组

用root的管理权限，可以改变某个用户的所在组。

- 改变用户所在的组
  - `usermod -g 新组名 用户名`
  - `usermod -d 目录名 用户名 改变该用户登陆的初始目录`
    - ! 用户需要有进入到新目录的权限

# 权限的基本介绍



## 第0-9位说明

- **第0位**: 确定文件的类型
  - **l** 是链接，相当于windows的快捷方式
  - **d** 是目录，相当于windows的文件夹
  - **c** 是字符设备文件，比如鼠标、键盘等
  - **b** 是块设备，比如硬盘
  - **-** 是普通文件，比如\*.txt文件
- **第1-3位**: 确定文件所有者对该文件的权限 --**User**
- **第4-6位**: 确定所属组拥有该文件的权限 -- **Group**
- **第7-9位**: 确定其它用户（其它组）拥有该文件的权限 -- **Other**

## rwx权限详解

### rwx修饰文件时

- **【r】** : 代表可读 (read) : 可以对文件进行读取，查看
- **【w】** : 代表可写 (write) : 可以进行修改，但是不是代表可以删除该文件，删除一个文件的前提条件是对该文件所在的目录有写权限，才能删除该文件。
- **【x】** : 代表可执行 (execute) : 可以被执行

### rwx修饰目录时

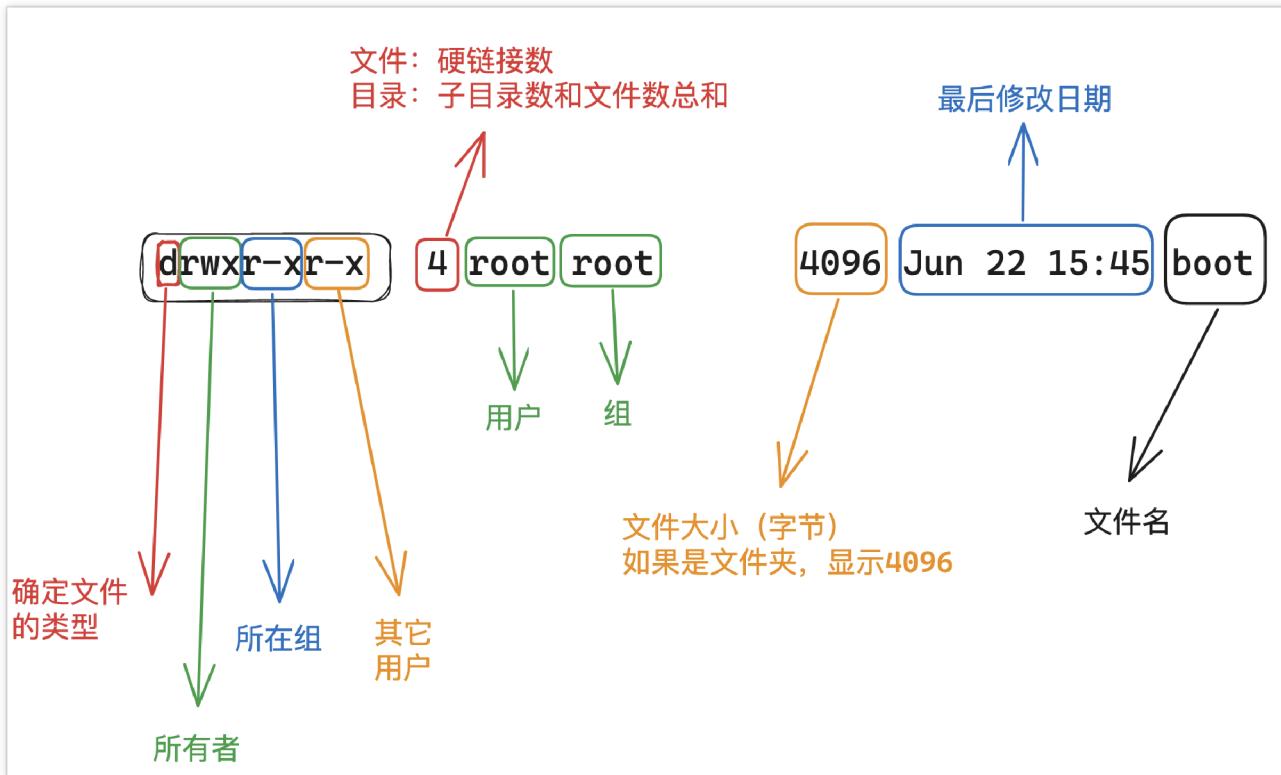
- **【r】** : 代表可读 (read) : 可以对文件进行读取，ls查看目录内容
- **【w】** : 代表可写 (write) : 可以进行修改，对目录内的内容进行创建+删除或重命名目

## 录

- 【x】：代表可执行 (execute) : 可以进入该目录

可以用数字进行表示: **r=4**、**w=2**、**x=1**, 因此 $rwx=4+2+1=7$ 。

其它说明:



## 修改权限

基本说明: 通过 **chmod** 指令, 可以修改文件或目录的权限

### 第一种方式: +、-、= 变更权限

+、-、=的含义:

- . 表示增加权限
- . 表示取消权限
- = 表示唯一设定权限

u、g、o、a的含义:

- u : 所有者
- g : 所有组 (所在组)
- o : 其它人

- a : 所有人

例如：

`chmod u=rwx,g=rx,o=r 文件名/目录` `chmod o+w 文件名/目录` : 给其它用户增加写的权限  
`chmod a-x 文件名/目录` : 移除所有用户的执行权限

## 第二种方式：通过数字变更权限

r=4、w=2、x=1

`chmod u=rwx,g=rw,o=r filename ==> chmod 751 filename`

## 修改文件所有者

`chown newowner filename` : 改变所有者

`chown newowner:newgroup filename` : 改变所有者和所在组

**-R** : 如果是目录，则应该使其下所有子文件或目录递归生效

# 5-定时任务调度

## crond 任务调度

crontab 进行 定时任务调度

### 概述

任务调度：是指系统在某个时间执行的特定的命令或程序

任务调度分类：

- 系统工作：有些重要的工作必须周而复始地执行
- 个别用户工作：希望定时执行某些程序

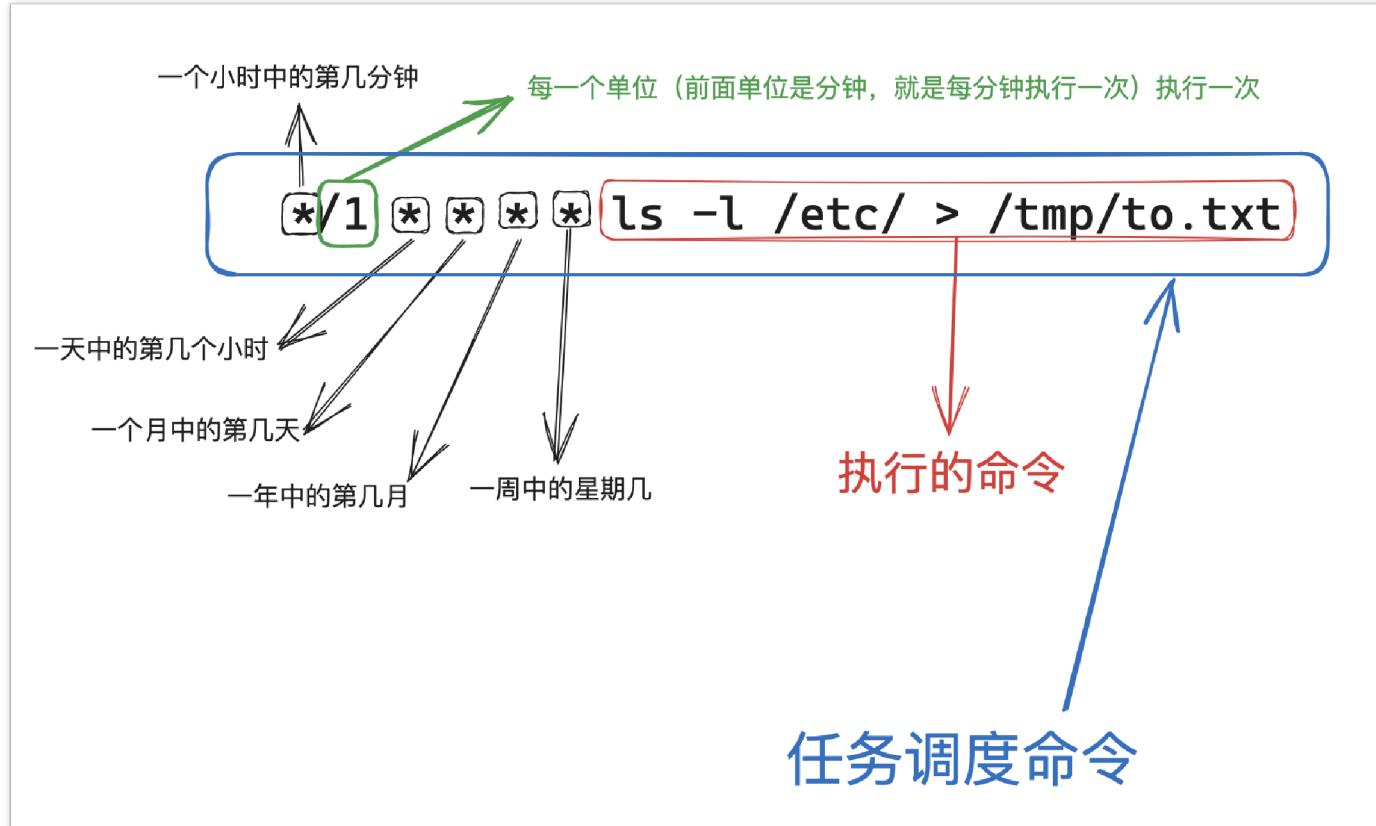
### 基本语法

**crontab [选项]**

### 常用选项

- **-e** : 编辑crontab定时任务
- **-l** : 查询crontab任务，列出任务列表
- **-r** : 删除当前用户所有的crontab任务

# 快速入门



## 特殊符号的说明

| 特殊符<br>号 | 含义                                                              |
|----------|-----------------------------------------------------------------|
| *        | 表示任何时间。比如，当第一占位符为*时，就表示一小时中每分钟的意思                               |
| ,        | 表示不连续的时间。比如命令： <b>0,8,12,16 * * *</b> , 表示在每天的8点、12点、16点都执行一次命令 |
| -        | 表示连续的时间。比如命令： <b>0 5 * * 1-6</b> 表示在每周一到周六的凌晨5点0分执行命令           |
| */n      | 代表每隔多久执行一次。比如命令： <b>*/10 * * * *</b> , 代表每隔10分钟就执行一遍命令          |

## 应用实例

- 每隔一分钟就将当前的日期信息，追加到/tmp/mydate文件中
  - \*/1 \* \* \* \* date >> /tmp/mydate**
- 每隔一分钟，将当前日期和日历都追加到/home/mycal文件中
  - 步骤
  - vim /home/my.sh** 写入内容，`date >> /home/mycal` 和 `cal >> /home/mycal`

- 给my.sh增加执行权限, `chmod u+x /home/my.sh`
- `crontab -e` 增加命令: `*/1 * * * * /home/my.sh`

## crond 相关指令

- `crontab -r` : 终止任务调度
- `crontab -l` : 列出当前有哪些任务调度
- `service crond restart` : 重启任务调度

## at 定时任务

### 基本介绍

- at 命令是一次性定时计划任务, at的守护进程会以后台模式运行, 检查作业队列来运行。
- 默认情况下, atd守护进程每60s检查作业队列, 有作业时, 会检查作业运行时间, 如果时间与当前时间相匹配, 则运行此作业。
- **at命令是一次性定时计划任务, 执行完一个任务后不再执行此任务了**
- 在使用at命令时, 一定要保证atd进程的启动, 可以通过指令 `ps -ef | grep atd` 来进行检查。

### at 命令格式

`at [选项] [时间]`

当输入完成时, 输入 `ctrl + D` 表示输入命令的结束 (输入两次)

当输入at命令时, 想要进行删除的话, 需要按`ctrl+del`

# at 命令选项

| 选项        | 含义                           |
|-----------|------------------------------|
| -m        | 当指定的任务被完成后，将给用户发送邮件，即使没有标准输出 |
| -I        | atq的别名                       |
| -d        | atrm的别名                      |
| -v        | 显示任务将被执行的时间                  |
| -c        | 打印任务的内容到标准输出                 |
| -V        | 显示版本信息                       |
| -q <队列>   | 使用指定的队列                      |
| -f <文件>   | 从指定文件读入任务而不是从标准输入读入          |
| -t <时间参数> | 以时间参数的形式提交要运行的任务             |

## at 时间的定义

- 接受在当天的hh:mm式的时间指定。如果时间已经过去，将会放到第二天执行。
- 使用比较模糊的词语来指定时间 比如：midnight、noon等
- 采用12小时制，在时间后面加上am或pm来说明是上午还是下午。例如：6pm
- 指定命令执行的具体日期
  - mm/dd/yy 或 dd.mm.yy
  - 指定的日期必须跟在指定时间的后面：4:00 2021-03-1
- 使用相对计时法
  - 指定格式：**now + 时间大小 时间单位**
  - 时间单位：minutes、hours、days、weeks
  - 比如：now + 5 minutes 含义：5分钟后执行指令
- 直接使用today、tomorrow 来指定完成的时间

## 其他指令

- atq**：显示系统中没用执行的工作任务
- atrm 编号**：删除已经设置的任务

# 6-Linux磁盘分区和挂载

## Linux分区

linux采用了一种叫‘载入’的处理方法，他的整个文件系统中包含了一整套的文件和目录，且将一个分区和一个目录联系起来。

这时，要载入的一个分区将使它的存储空间在一个目录下进行获得。

### 查看所有设备的挂载情况

`lsblk` 或者 `lsblk -f`

```
[root@cvm-3jzbcc25i225 ~]# lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda    253:0    0  20G  0 disk
└─vda1 253:1    0  20G  0 part /
vdb    253:16   0  20G  0 disk
```

## 将磁盘进行挂载的案例

### 增加一块磁盘的总体步骤

以虚拟机为例

1. 为虚拟机添加硬盘
2. 分区
3. 格式化
4. 挂载
5. 设置可以进行自动挂载

### 1-在虚拟机中增加磁盘

添加完硬盘后，需要重启系统，服务器才能识别新的硬盘。

## 2- 分区

查看挂载的新的硬盘名称，为**sdb**，由于**/dev** 文件目录是进行硬件管理的文件目录，所以，如果要对新的硬盘进行操作，就要对**/dev/sdb** 文件进行操作。

- **/dev**
  - 虚拟文件系统，用于表示的访问系统中的设备
  - 类似于win中的设备管理器，把所有的硬件用文件的形式进行存储
  - 通过**/dev**目录中的设备文件，用户和应用程序可以与硬件设备进行交互，读取和写入数据，进行输入和输出操作。

分区命令：**fdisk /dev/sdb** (**fdisk /dev/新加入的硬盘名称**)

开始对**/sdb**进行分区

- m 显示命令列表
- p 显示磁盘分区 等同于 **fdisk -l**
- n 新增分区
- d 删除分区
- w 写入并退出

## 3-格式化分区

命令：**mkfs -t ext4 /dev/sdb1**

将新分好的区进行格式化操作

其中，ext4 是文件类型

## 4-挂载分区

分区必须挂载上才能进行使用

挂载：将一个分区与一个目录联系起来

**mount 设备名称 挂载目录**

**umount 设备名称或挂载目录**

！用命令行进行挂载的分区，服务器重启后，会失效！！！

## 5-进行永久挂载

永久挂载：通过修改 **/etc/fstab** 实现永久挂载

添加完成后，执行 `mount -a` 即刻生效

```
#  
# /etc/fstab  
# Created by anaconda on Fri Oct 30 15:01:14 2020  
#  
# Accessible filesystems, by reference, are maintained under '/dev/disk'  
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info  
#  
/dev/sdb1          /newdisk      ext4  defaults    0 0  
UUID=12ae1cc1-09e0-42ce-b659-063df3e7c941 /      ext4  defaults    1 1  
UUID=df4d64bd-981e-41eb-8f73-16ac20f0371e /boot  ext4  defaults    1 2  
UUID=494c7f94-5656-45a3-b990-008c2b0e39b7 swap   swap  defaults    0 0  
-
```

## 磁盘情况查询

### 查询系统整体磁盘使用情况

基本语法：**df -h**

```
[root@cvm-3jzbcc25i225 ~]# df -h  
Filesystem      Size  Used Avail Use% Mounted on  
devtmpfs        473M    0  473M   0% /dev  
tmpfs          496M    0  496M   0% /dev/shm  
tmpfs          496M   57M  440M  12% /run  
tmpfs          496M    0  496M   0% /sys/fs/cgroup  
/dev/vda1       20G  1.5G   19G   8% /  
tmpfs         100M    0  100M   0% /run/user/0
```

### 查询指定目录的磁盘占用情况

• 基本语法：**du [选项] /目录**

- 查询指定目录的磁盘占用情况，默认为当前目录

- 选项

- -s 指定目录占用大小汇总
- -h 带计量单位，方便human进行阅读
- -a 含文件
- --max-depth=1 子目录深度
- -c 列出明细的同时，增加汇总值

# 磁盘情况-工作实用指令

前提指令：wc

wc - print newline, word, and byte counts for each file

**wc -l** 输出文件的行数

统计文件夹下文件的个数

```
ls -l | grep "^-" | wc -l
```

统计文件夹下目录的个数

```
ls -l | grep "^d" | wc -l
```

统计文件夹下文件的个数包括子文件夹里的

```
ls -lR | grep "^-" | wc -l
```

统计文件夹下目录的个数包括子文件夹里的

```
ls -lR | grep "^d" | wc -l
```

以树状显示目录结构

：如果没有tree，可以通过 yum install tree 进行安装

## 7-Linux的网络配置

### win下的ipconfig

### linux下的ifconfig

### ping 测试主机之间的连通性

### 设置主机名和hosts映射

#### 设置主机名

- 查看主机名: `hostname`
- 修改主机名: `vim /etc/hostname` , 重启后生效

- **/etc** ★

- 里面存放所有系统管理所需要的配置文件和子目录
- 比如: config文件
- 它是"etcetera"的缩写, 意思是"其他"。这些文件包括**网络配置、用户账户信息、服务配置、软件包管理**等等。

#### 设置hosts映射

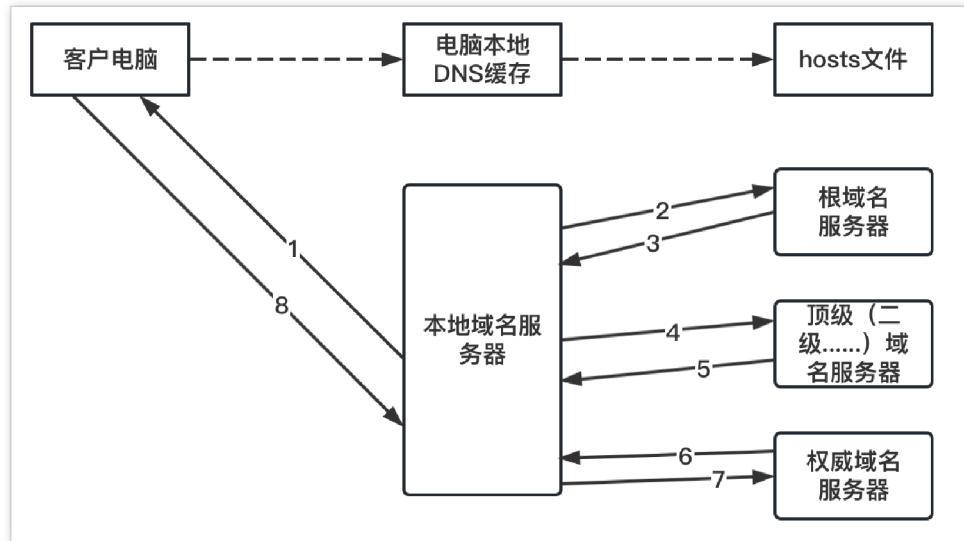
在win中:

C:\Windows\System32\drivers\etc\hosts

在linux中:

修改/etc/hosts文件, 进行指定

# 主机名解析分析过程



## hosts

"hosts" 是一个计算机中的文件，用于映射主机名和IP地址。

它通常用于在本地计算机上设置自定义的主机名解析规则。通过编辑hosts文件，你可以将特定的主机名映射到特定的IP地址，以便在浏览器或其他网络应用程序中访问这些主机名时，可以直接使用指定的IP地址进行连接。这对于测试网站、阻止恶意网站或在本地环境中进行开发和调试非常有用。

## DNS

DNS代表域名系统（Domain Name System）。

它是互联网中的一种网络协议，用于将域名（如example.com）转换为IP地址（如192.0.2.1），以便计算机能够识别和访问特定的网络资源。

DNS起到了类似电话簿的作用，帮助用户在互联网上定位和访问网站、电子邮件服务器和其他网络服务。

# 8-linux进程管理

## 基本介绍

在linux中，每个执行的程序都称为一个进程，每个进程都分配一个ID号（PID）

每个进程都可能以两种方式存在，**前台与后台**

一般系统的服务都是以后台进程的方式存在的，而且都会常驻在系统中，直到关机才结束。

## 显示系统执行的进程

### ps 指令基本介绍

ps (process show) 是用来显示目前系统中，有哪些正在执行，以及它们的执行情况。

#### ps [选项]

可选选项

1. ps -a 显示当前终端的所有进程信息
2. ps -u 以用户的格式显示进程信息
3. ps -x 显示后台进程运行的参数

一般来说，进行结合使用，即为： **ps -aux**

| 进程执行用户 | 进程号 | cpu占用比 | 物理内存占用比 | 进程状态  | 进程启动时间 | 命令  |      |       |      |                          |
|--------|-----|--------|---------|-------|--------|-----|------|-------|------|--------------------------|
| USER   | PID | %CPU   | %MEM    | VSZ   | RSS    | TTY | STAT | START | TIME | COMMAND                  |
| root   | 1   | 0.0    | 0.6     | 54428 | 6638   | ?   | Ss   | Jul09 | 1:10 | /usr/lib/systemd/systemd |
| root   | 2   | 0.0    | 0.0     | 0     | 0      | ?   | S    | Jul09 | 0:00 | [kthreadd]               |
| root   | 4   | 0.0    | 0.0     | 0     | 0      | ?   | S<   | Jul09 | 0:00 | [kworker/0:0H]           |
| root   | 6   | 0.0    | 0.0     | 0     | 0      | ?   | S    | Jul09 | 0:00 | [ksoftirqd/0]            |
| root   | 7   | 0.0    | 0.0     | 0     | 0      | ?   | S    | Jul09 | 0:00 | [migration/0]            |
| root   | 8   | 0.0    | 0.0     | 0     | 0      | ?   | S    | Jul09 | 0:00 | [rcu_bh]                 |

4. ps -e: 显示当前所有进程

## 5. ps -f: 全格式

| UID  | PID | PPID | C | S     | TIME | TTY | TIME     | CMD                      |
|------|-----|------|---|-------|------|-----|----------|--------------------------|
| root | 1   | 0    | 0 | Jul09 | ?    |     | 00:01:10 | /usr/lib/systemd/systemd |
| root | 2   | 0    | 0 | Jul09 | ?    |     | 00:00:00 | [kthreadd]               |
| root | 4   | 2    | 0 | Jul09 | ?    |     | 00:00:00 | [kworker/0:0H]           |
| root | 6   | 2    | 0 | Jul09 | ?    |     | 00:00:07 | [ksoftirqd/0]            |
| root | 7   | 2    | 0 | Jul09 | ?    |     | 00:00:00 | [migration/0]            |

父进程ID

CPU用于计算执行优先级的因子  
数值越大，优先级会降低，表示是CPU密集型进程  
数值越小，优先级会提高，表示是IO密集型进程

## 终止进程kill和killall

### 基本语法

- **kill [选项] 进程号**
  - 通过进程号来杀死/终结进程
  - 选项
    - **-9** 表示强迫进程立即停止
- **killall 进程名称**
  - 通过进程名称来杀死进程

## 查看进程树 pstree

! :如果显示没有此命令，在centos中可以通过**yum install psmisc**来进行安装此命令。

基本语法: **pstree [选项]**

可以更加直观的来看进程信息

### 常用选项

- **-p** 显示进程的pid
- **-u** 显示进程所属的用户

```
[root@cvm-3jzbcc25i225 ~]# pstree -p
systemd(1)─agetty(1005)
              ├─agetty(1010)
              ├─auditd(465)─{auditd}(466)
              ├─chronyd(564)
              ├─crond(1008)
              ├─dbus-daemon(559)─{dbus-daemon}(576)
              ├─dhclient(807)
              ├─gssproxy(570)─{gssproxy}(571)
                            ├─{gssproxy}(572)
                            ├─{gssproxy}(573)
                            ├─{gssproxy}(574)
                            └─{gssproxy}(575)
              ├─master(970)─pickup(11855)
                            └─qmgr(978)
              ├─polkitd(554)─{polkitd}(578)
                            ├─{polkitd}(581)
                            ├─{polkitd}(582)
                            ├─{polkitd}(591)
                            ├─{polkitd}(592)
                            └─{polkitd}(593)
              ├─rpcbind(565)
              ├─rsyslogd(1002)─{rsyslogd}(1021)
                            └─{rsyslogd}(1022)
              ├─sshd(1000)─sshd(9335)─bash(9341)
                            └─sftp-server(9358)
                            └─sshd(14451)─bash(14456)─pstree(16104)
              ├─systemd-journal(405)
              ├─systemd-logind(577)
              ├─systemd-udevd(443)
              └─tuned(865)─{tuned}(1035)
                            ├─{tuned}(1036)
                            ├─{tuned}(1039)
                            └─{tuned}(1040)
```

# 服务 service 管理

## service介绍

服务 (service) 的本质就是进程，但是该进程是运行在后台的。通常会监听某个端口，等待其它程序的请求。

因此，后台程序又被称为守护程序。

## service管理指令

- **service 服务名 [start|stop|restart|status]**

- 在centos7以后，很多服务不再使用service去管理，而是使用systemctl
- ★service 指令管理的服务在 **/etc/init.d** 中进行查看 ([linux中的/etc目录](#))

```
[root@cvm-3jzbcc25i225 init.d]# ls -l
total 40
-rw-r--r--. 1 root root 18281 Aug 19 2019 functions
-rwxr-xr-x. 1 root root 4569 Aug 19 2019 netconsole
-rwxr-xr-x. 1 root root 7928 Aug 19 2019 network
-rw-r--r--. 1 root root 1160 Apr  7 2020 README
```

## chkconfig 指令

通过chkconfig命令，可以给服务的各个运行级别设置自启动/关闭

也就是说，服务的自启动和关闭是针对不同的机器运行级别的

显示chkconfig指令支持的服务：**chkconfig --list**

```
[root@cvm-3jzbcc25i225 init.d]# chkconfig --list
Note: This output shows SysV services only and does not include native
      systemd services. SysV configuration data might be overridden by native
      systemd configuration.

      If you want to list systemd services use 'systemctl list-unit-files'.
      To see services enabled on particular target use
      'systemctl list-dependencies [target]'.

netconsole      0:off    1:off    2:off    3:off    4:off    5:off    6:off
network        0:off    1:off    2:on     3:on     4:on     5:on     6:off
```

使用chkconfig指令

格式：**chkconfig --level 级别 服务名 on/off**

## 指定运行级别

| 运行级别 | 级别含义 |
|------|------|
|------|------|

|   |    |
|---|----|
| 0 | 关机 |
|---|----|

|   |                |
|---|----------------|
| 1 | 单用户状态（找回丢失的密码） |
|---|----------------|

|   |             |
|---|-------------|
| 2 | 多用户状态没有网络服务 |
|---|-------------|

|   |            |
|---|------------|
| 3 | 多用户状态有网络服务 |
|---|------------|

|   |            |
|---|------------|
| 4 | 系统未使用保留给用户 |
|---|------------|

| 运行级别 | 级别含义 |
|------|------|
| 5    | 图形界面 |
| 6    | 系统重启 |

!：当使用chkconfig重新设置服务在不同级别下的自启动和关闭时，需要重启机器才会生效。

## systemctl 管理指令

### 基本指令

`systemctl [start|stop|restart|status] 服务名`

★：**systemctl** 指令管理的服务在 `/usr/lib/systemd/system` 中查看

- **/usr** ★

- 非常重要的目录
- 用户的很多应用程序和文件都会放在该目录下面，类似于win中的program files 目录
- Linux中的/usr 约等于 win中的program files 目录，也就是说如果安装应用程序，会默认安装到该文件下面。

## systemctl 设置服务的自启动状态

- 查看服务开机启动状态
  - `systemctl list-unit-files`
- 设置服务开机启动
  - `systemctl enable 服务名`
- 关闭服务开机启动
  - `systemctl disable 服务名`
- 查询某个服务是否是自启动的
  - `systemctl is-enabled 服务名`

### 示例

查看systemctl 支持的firewalld服务

```
[root@cvm-3jzmbcc25i225 ~]# ls -l /usr/lib/systemd/system | grep firewalld  
-rw-r--r--. 1 root root 657 Apr 28 2021 firewalld.service  
[root@cvm-3jzmbcc25i225 ~]#
```

发现存在firewalld.service

则可以通过命令进行控制并查看

- `systemctl status firewalld`
- `systemctl stop firewalld`
- `systemctl start firewalld`

```
[root@cvm-3jzmbcc25i225 ~]# systemctl status firewalld  
● firewalld.service - firewalld - dynamic firewall daemon  
  Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled; vendor preset: enabled)  
  Active: inactive (dead)  
    Docs: man:firewalld(1)  
[root@cvm-3jzmbcc25i225 ~]#
```

## 打开或者关闭指定端口

- 打开端口
  - `firewall-cmd --permanent --add-port=端口号/协议`
- 关闭端口
  - `firewall-cmd --permanent --add-port=端口号/协议`
- 重新载入，才能生效
  - `firewall-cmd --reload`
- 查询端口是否开放
  - `firewall-cmd --query-port=端口号/协议`

在win中，可以通过`telnet IP地址 端口`来测试某个主机的端口是否打开

## 动态监控进程

### 基本用法

`top [选项]`

"top"和"ps"都是在Unix和类Unix系统中使用的命令，用于查看系统中运行的进程信息。它们之间的区别：

## 1. 功能

- top是一个动态的进程监视工具，它实时显示系统中运行的进程信息，包括CPU使用率、内存使用情况等。
- ps是一个静态的进程查看工具，它一次性显示当前系统中的进程信息，不会实时更新。

## 2. 显示方式

- top以交互式的方式显示进程信息，可以动态排序和过滤进程，还可以实时更新显示。
- ps以命令行的方式显示进程信息，需要使用不同的选项来指定要显示的信息。

## 选项说明

- **-d 秒数**：指定top命令每间隔几秒更新，默认为3s
- **-i**：使用top不显示任何闲置或者僵死进程
- **-p**：通过指定监控进程ID来仅仅监控某个进程的状态

## 在top中交互操作

| 操作 | 功能                  |
|----|---------------------|
| P  | 以CPU使用率来进行排序，默认是此选项 |
| M  | 以内存的使用率排序           |
| N  | 以PID来进行排序           |
| u  | 输入用户名，监视特定用户        |
| k  | 输入要结束的进程ID号，终结指定的进程 |
| q  | 退出                  |

## 查看网络状态

### 查看系统网络状态 netstat

- 基本语法 **netstat [选项]**
  - 查看系统网络状态
  - 选项说明
    - **-an** 按照一定顺序排列输出
    - **-p** 显示哪个进程在调用

```
[root@cvm-3jzmbcc25i225 ~]# netstat -anp | grep sshd
tcp        0      0 0.0.0.0:22              0.0.0.0:*                  LISTEN      1000/sshd
tcp        0      64 172.16.2.52:22          221.192.180.105:7989    ESTABLISHED 32339/sshd: root@pt
tcp6       0      0 ::*:22                  ::*:*                   LISTEN      1000/sshd
unix  2      [ ]            DGRAM
unix  3      [ ]            STREAM     CONNECTED
   2536735  32339/sshd: root@pt
   17179   1000/sshd
```

## 检测主机连接命令 ping

检测连接远程主机是否正常

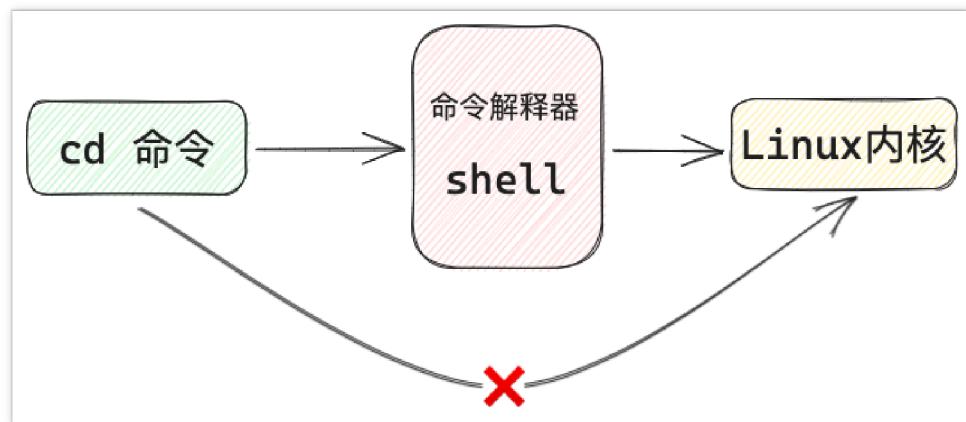
# shell编程

## 1-学习shell的意义

1. 在进行服务器集群管理时，需要编写shell程序来进行服务器管理
2. 编写一些shell脚本进行运行程序或者是服务器的维护
3. 编写shell程序来管理集群

## 2-shell 的概述

Shell是一种命令行解释器，它是操作系统和用户之间进行交互的界面。在计算机领域中，Shell是一种用于执行命令、管理文件和程序的软件。



Shell提供了一个命令行界面，用户可以通过输入命令来与操作系统进行交互。用户可以使用Shell来执行各种操作，如运行程序、管理文件和目录、设置环境变量、执行脚本等。

Shell还提供了一些特殊的功能，如命令历史记录、命令补全、管道操作等，以方便用户进行操作和管理。

在Unix和类Unix系统中，常见的Shell包括Bourne Shell (sh)、Bash (Bourne Again Shell)、C Shell (csh)、Korn Shell (ksh)等。每种Shell都有自己的特点和语法，但它们都提供了类似的基本功能。

Shell是一种命令行解释器，它提供了用户与操作系统交互的界面，使用户能够执行命令、管理文件和程序，并进行各种操作和管理。

# 3-Shell 脚本的执行方式

## 脚本格式要求

1. 脚本要以`#!/bin/bash`开头

- 当在一个脚本文件的开头添加`#!/bin/bash`时，它告诉操作系统在执行该脚本时使用**Bash**来解释和执行脚本中的命令。
- 这样就可以直接运行脚本文件，而不需要在命令行中显式地指定解释器。

2. 脚本需要有可执行权限!!!

## 编写第一个Shell脚本

创建一个shell脚本，用来输出hello world!

SHELL

```
#!/bin/bash
echo "hello world!"
```

```
[root@cvm-3jzmbcc25i225 shell-test]# cat hello.sh
#!/bin/bash
echo "hello world!"
[root@cvm-3jzmbcc25i225 shell-test]# 
```

执行结果

给hello.sh增加可执行权限

[root@cvm-3jzmbcc25i225 shell-test]# vim hello.sh
[root@cvm-3jzmbcc25i225 shell-test]# ./hello.sh
-bash: ./hello.sh: Permission denied
[root@cvm-3jzmbcc25i225 shell-test]# chmod u+x hello.sh
[root@cvm-3jzmbcc25i225 shell-test]# ls -l
total 4
-rwxr--r--. 1 root root 32 Jul 20 17:55 hello.sh
[root@cvm-3jzmbcc25i225 shell-test]# ./hello.sh
hello world!

## 脚本的常用执行方式

- 方式一：输入脚本的绝对路径或相对路径
  - 首先要赋予新建脚本执行的权限`(+x)`，然后再去执行脚本
  - 比如：

- 相对路径: `./hello.sh`
- 绝对路径: `/home/shell-test/hello.sh`

- 方式二: **sh + 脚本** 进行运行

- 说明: 不用赋予脚本执行权限, 直接执行即可
- 比如: `sh hello.sh` 也可以使用绝对路径

```
[root@cvm-3jzbcc25i225 shell-test]# ./hello.sh
hello world!
you are successful!
[root@cvm-3jzbcc25i225 shell-test]# sh hello.sh
hello world!
you are successful!
```

## 4-shell 的变量

### 变量介绍

Linux shell 中的变量分为 : 系统变量和用户自定义变量

### 系统变量

系统变量的例子: `$HOME`、`$PWD`、`$SHELL`、`$USER` 等等

显示所有的系统变量: `set`

```
[root@cvm-3jzbcc25i225 ~]# set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:histappend:hostcomplete:interactive_comments:login_shell
:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="2" [2]="46" [3]="2" [4]="release" [5]="x86_64-redhat-linux-gnu")
BASH_VERSION='4.2.46(2)-release'
COLUMNS=124
DIRSTACK=()
EUID=0
GROUPS=()
HISTCONTROL=ignoredups
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
```

### 自定义变量

#### 基本语法

- 定义变量: `变量名 = 值`
- 撤销变量: `unset 变量`

- 声明静态变量：**readonly 变量**
  - 注意：静态变量不能进行 unset

定义变量的规则：

- 和其它语言一样
- 等号两侧不能有空格
- 变量名称一般习惯为大写

变量的基本操作

SHELL

```
#!/bin/bash
#定义变量
A=100
#输出变量需要加上$
echo A=$A
echo "A=$A"
#撤销变量A
unset A
echo $A
#声明静态的变量B，不能unset
readonly B=2
echo "B=$B"
#将指令返回的结果赋值给变量
DATE=`date`
CAL=$(cal)
echo "$DATE and $CAL"

:<<!
多行注释
!
```

► 在echo输出的两种方式中，采用字符串输出可以直接输出空格，而如果不加双引号进行输出，则源文件中的空格将会被忽略

将Linux命令的返回值赋值给变量 (1):通过反引号运行里面的命令，并把结果返回给变量A

```
A=`date`
```

(2):通过\$()运算，运行里面的命令，并把结果返回给变量A

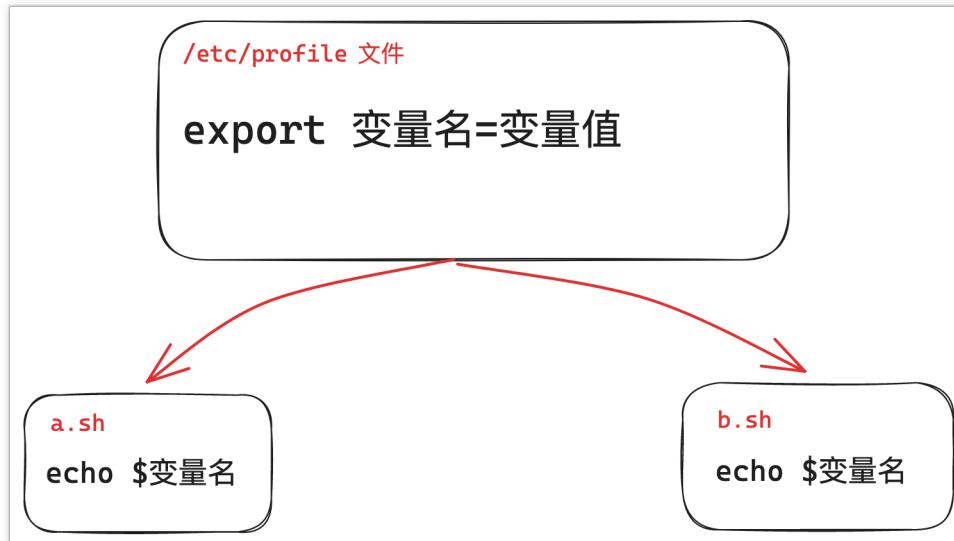
```
A=$(date) #这种写法更好，即使管道操作符，也能正确执行
```

```
[root@cvm-3jzmbcc25i225 shell-test]# vim cmd.sh
[root@cvm-3jzmbcc25i225 shell-test]# sh cmd.sh
Thu Jul 20 19:54:12 CST 2023
July 2023 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[root@cvm-3jzmbcc25i225 shell-test]# cat cmd.sh
DATE=`date`
CAL=$(cal)
echo $DATE
echo $CAL
```

## 5-环境变量（全局变量）

### 概述

这里的环境变量可以近似的看做为全局变量



### 基本语法

- **export 变量名=变量值**
  - 将shell变量输出为环境变量（全局变量）
- **source 配置文件**
  - 让修改后的信息立即生效

- `echo $变量名` 查询环境变量的值

```
export PATH=$JAVA_HOME/bin:$PATH  
#定义一个环境变量  
export TOMCAT_HOME=/opt/tomcat
```

## 6-位置参数变量

### 概要

位置参数变量：让脚本可以通过命令行获取到各个参数的信息

在Shell脚本中，位置参数变量是一组特殊的变量，用于存储通过命令行传递给脚本的参数值。

### 基本语法

位置参数变量以数字作为前缀，从1开始递增，表示参数的位置顺序。

常用的位置参数变量：

1. `$0`：表示脚本本身的名字。
2. `$1`、`$2`、`$3`，以此类推：表示命令行中传递给脚本的位置参数的值。例如，`$1`表示第一个参数，`$2`表示第二个参数，以此类推。十以上的参数，需要用到大括号进行包含。比如  `${10}`
3. `$*`：表示所有位置参数的值，作为一个单独的字符串。参数之间以空格分隔。
4. `$@`：表示所有位置参数的值，作为一个数组。每个参数都可以单独访问。
5. `$#`：表示传递给脚本的位置参数的个数。

通过使用这些位置参数变量，可以在Shell脚本中获取和处理命令行传递的参数值。

例如，`$1`可以用于获取第一个参数的值，`$#`可以用于获取参数的个数。

### 位置参数示例

以下是一个示例脚本，演示如何使用位置参数变量：

```
#!/bin/bash

echo "脚本名称: $0"
echo "第一个参数: $1"
echo "第二个参数: $2"
echo "所有参数: $*"
echo "参数个数: $"
```

当执行这个脚本并传递参数时，脚本将打印出相应的位置参数值和参数个数。

```
[root@cvm-3jzbcc25i225 shell-test]# sh hello.sh C X
name:hello.sh
first:C
second:X
count:2
all:C X
[root@cvm-3jzbcc25i225 shell-test]#
```

## 7-预定义变量

### 概述

预定义变量：就是shell设计者事先已经定义好的变量，可以直接在shell脚本中进行使用

### 基本用法

1. **\$\$**：表示当前脚本的进程ID。
2. **\$!**：表示最后一个在后台运行的进程的进程ID。
3. **\$?**：表示上一个命令的退出状态码。如果命令执行成功，该值为0；如果命令执行失败，该值为非零。

## 8-运算符

### 概要

在shell中如何进行各种运算操作

# 基本语法

计算表达式，在shell中表达式的计算，需要通过特殊的格式进行

三种方式

- `$((运算式))`
- `$[运算式]` ★ 推荐用法
- `expr m + n`
  - expr 运算符之间要有空格，如果希望将expr的结果赋值给某个变量，使用“”
  - ```
[root@cvm-3jzbcc25i225 ~]# expr 4 + 6
10
[root@cvm-3jzbcc25i225 ~]#
```
  - 运算符号
    - \* 乘号前面需要有个转移符号
      - ◦   ◦ / %

## 示例

求出命令行两个参数的和

SHELL

```
#!/bin/bash

echo "first: $1"
echo "second: $2"
echo "$1+$2=$[$1+$2]"
```

```
[root@cvm-3jzbcc25i225 shell-test]# cat hello.sh
#!/bin/bash
echo "first: $1"
echo "second: $2"
echo "ans:     $1+$2=$[$1+$2]"
```

```
[root@cvm-3jzbcc25i225 shell-test]# sh hello.sh 10 20
first: 10
second: 20
ans:    10+20=30
[root@cvm-3jzbcc25i225 shell-test]#
```

## 9-条件判断

### 基本语法

[ condition ] (注意: condition前后要有空格)

如果非空，返回true。可使用 \$? 进行验证（0为true, >1为false）

### 判断语句

#### 字符串比较

- ==：字符串比较，两个字符串是否相等

#### 两个整数进行比较



#### 按照文件权限进行判断

- r：有读的权限

- `-w` : 有写的权限
- `-x` : 有执行的权限

## 按照文件类型进行判断

- `-f` : 文件存在，并且是一个常规文件
- `-e` : 文件存在
- `-d` : 文件存在，并且是一个目录

```
#!/bin/bash

# 判断字符串是否相等
if [ "ok" = "ok" ]
then
    echo "ok"
fi

# 判断两个数字的大小
if [ 1 -gt 2 ]
then
    echo "1 > 2"
elif [ 1 -lt 2 ]
then
    echo "1 < 2"
fi

# 判断文件是否存在
if [ -f "test.sh" ]
then
    echo "test.sh is exist"
fi

# 判断权限
if [ -r "test.sh" ]
then
    echo "test.sh is readable"
fi

if [ $0 = "test.sh" ]
then
    echo "test.sh"
fi
```

注意💡：如果要将if和then写在同一行，需要加上⋮

比如：if [ "ok" = "ok" ]; then

# 流程控制

## 第一种：基础语法

SHELL

```
if [ 条件 ]
then
代码
fi
```

## 第二种：多分支

SHELL

```
if [ 条件 ]
then
代码
elif [ 条件 ]
then
代码
fi
```

# 10-case语句

## 基本语法

SHELL

```
case $变量名 in
"值1")
程序1
;;
"值2")
程序2
;;
*)
程序3
;;
esac
```

## 示例

```
#!/bin/bash
case $1 in
"1")
    echo "monday"
;;
"2")
    echo "tuesday"
;;
*)
    echo "other"
;;
esac
```

## 11-for循环和while循环

### for 基本语法

#### 方式一

SHELL

```
for 变量 in 值1 值2 值3 .....
do
    程序
done
```

#### 方式二

SHELL

```
for((初始值;循环控制条件;变量变换))
do
    程序
done
```

# for循环示例

## 方式一

区分\$@与\$\*的区别 (\$\* 和 \$@)

```
[root@cvm-3jzbcc25i225 shell-test]# sh test.sh 1 2 3 4 5
1 2 3 4 5
-----
1
2
3
4
5
[root@cvm-3jzbcc25i225 shell-test]# cat test.sh
#!/bin/bash
for i in "$*"
do
    echo $i
done
echo "-----"
for j in "$@"
do
    echo $j
done
```

## 方式二

```
[root@cvm-3jzbcc25i225 shell-test]# cat test.sh
#!/bin/bash
SUM=0
for(( i=1; i<=$1; i++ ))
do
    SUM=$[ $SUM+$i ]
done
echo "总和 SUM=$SUM"

[root@cvm-3jzbcc25i225 shell-test]# sh test.sh 50
总和 SUM=1275
[root@cvm-3jzbcc25i225 shell-test]#
```

# while循环基本语法

SHELL

```
while [ 条件判断式 ]
do
    程序代码
done
```

# 12-read 读取控制台输入

## 基本语法

- **read [选项] [参数]**
- 选项
  - -p : 指定读取值时的提示符
  - -t : 指定读取值时等待的时间 (s) , 如果没有在指定的时间内输入, 就不再等待了
- 参数
  - 变量: 指定读取值的变量名

## 示例

SHELL

```
#!/bin/bash
read -p "请输入NUM1: " NUM1
echo "你输入的NUM1的值为: $NUM1"
```

```
[root@cvm-3jzbcc25i225 shell-test]# cat test.sh
#!/bin/bash
read -p "请输入 NUM1:" NUM1
echo "你输入的 NUM1 的值为 $NUM1"

[root@cvm-3jzbcc25i225 shell-test]# sh test.sh
请输入 NUM1:50
你输入的 NUM1 的值为 50
[root@cvm-3jzbcc25i225 shell-test]#
```

# 12-函数介绍

## 概述

和其它语言一样, shell编程语言也有函数, 其中分为系统函数和自定义函数。

## 系统函数 (语言自带函数)

### basename

- 基本语法: **basename [pathname] [suffix]** 或 **basename [string] [suffix]**

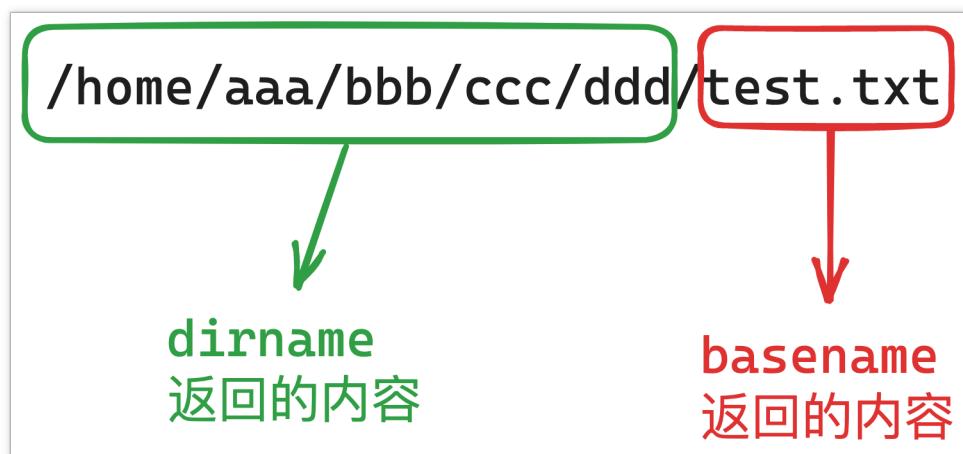
- 功能：返回完整路径最后的/部分，通常用于获取文件名
- 选项
  - suffix为后缀，如果suffix被制定了， basename会将pathname或string中的suffix去掉
- 应用示例

- ```
congxing@AirM1 ~ % basename
usage: basename string [suffix]
        basename [-a] [-s suffix] string [...]
congxing@AirM1 ~ % basename /home/aaa/test.txt
test.txt
congxing@AirM1 ~ %
```

## dirname

- 基本语法： dirname 文件绝对路径
- 功能： 返回完整路径最后的前面部分，通常用于返回路径部分
- 应用示例

- ```
congxing@AirM1 ~ % dirname /home/aaa/test.txt
/home/aaa
congxing@AirM1 ~ %
```



# 自定义函数

## 基本语法

SHELL

```
# 函数定义
function 函数名()
{
    代码
    [return int;]
}
```

```
# 函数调用
```

```
函数名 [值]
```

## 应用示例

编写一个名为getsum的函数，用于计算两个参数的和

SHELL

```
#!/bin/bash
# 定义函数
function getsum()
{
    SUM=$[n1+n2]
    echo $SUM # echo 后面的是函数的返回值
}

read -p "first: " n1
read -p "second: " n2

ANS=$(getsum $n1 $n2)

echo "SUM=$ANS"
```

```
[root@cvm-3jzmbcc25i225 ~]# sh test.sh
first: 1
second: 2
SUM=3
[root@cvm-3jzmbcc25i225 ~]# cat test.sh

#!/bin/bash
# 定义函数
function getsum()
{
    SUM=$[ $n1+$n2]
    echo $SUM;
}

read -p "first: " n1
read -p "second: " n2

ANS=$(getsum $n1 $n2)
```

# Fortran 语 言

# 基 本 用 法 总 结

# 1-fortran 简介

## 功能和用途

Fortran (Formula Translation) 是一种编程语言，最初于1957年开发。也可以称其为数学公式翻译器，即：把数学公式翻译成计算机机器语言，快捷地进行数值计算和科学数据的处理。

它是一种面向科学和工程计算的高级编程语言，用于数值计算和科学计算。

Fortran具有强大的数学和科学计算功能，并且在高性能计算领域被广泛使用。

## fortran 语言的发展

- 57年，fortran开始使用
- 66年，fortran的语言标准得到了统一 (Fortran 66)
- 77年，fortran引入结构化设计 (Fortran 77)
- 92年，fortran 加入面向对象、指针改良语法编写格式 (**Fortran 90**)
- 97年，fortran 支持平行运算，(Fortran 95)

## fortran 的后缀格式

Fortran语言的源代码文件通常使用以下后缀格式：

1. .f/.F/.for：这是Fortran 77的常见后缀格式，表示源代码文件是Fortran语言的程序。
2. **.f90**：这是Fortran 90及其后续版本的后缀格式，表示源代码文件是Fortran 90或更高版本的程序。
3. .f95：这是Fortran 95的后缀格式，表示源代码文件是Fortran 95的程序。
4. .f03：这是Fortran 2003的后缀格式，表示源代码文件是Fortran 2003的程序。
5. .f08：这是Fortran 2008的后缀格式，表示源代码文件是Fortran 2008的程序。

这些后缀格式有助于标识Fortran语言的源代码文件类型，并提供了一致的命名约定。

## fortran是编译语言

fortran语言的运行与C和C++类似，都需要经过编译链接形成最后的可执行文件才能进行运行。



由于fortran语言针对的是高性能计算，需要大量的数据输入和输出，因此，输入和输出通常以文件的形式存在。

而且，fortran语法可以与其他语言进行联合使用，比如，先使用fortran语言进行数据的处理，并对结果进行输出，而后，通过python对数据结果进行可视化显示。

## 编译器

和C++一样 ([C++常见的编译器](#)有：GCC、Clang、MVC++等)，作为编译语言，Fortran语言也有许多编译器用来编译fortran语言的源代码。

比如：

1. GNU Fortran (**gfortran**)：这是[GNU](#)编译器套件（GCC）中的Fortran编译器。它是一个免费的开源编译器，广泛用于Linux和其他操作系统。
2. Intel Fortran Compiler：这是英特尔公司提供的Fortran编译器，专注于优化和性能。它支持多平台，并提供了一些特定于英特尔处理器的优化功能。
3. IBM XL Fortran：这是IBM公司提供的Fortran编译器，适用于IBM Power和IBM Z系列的平台。它具有高度的优化能力和对并行计算的支持。
4. NAG Fortran Compiler：这是由数值算法集团（NAG）提供的商业Fortran编译器。它专注于数值计算和科学计算，并提供了广泛的数值算法库。
5. PGI Fortran Compiler：这是由NVIDIA公司提供的Fortran编译器，主要用于GPU加速计算。它支持Fortran和CUDA混合编程，可以实现高性能的并行计算。

这些编译器在不同的平台和应用场景中具有各自的特点和优势。选择合适的Fortran编译器取决于你的需求、平台支持和个人偏好。

## 2-fortran语言程序设计基础

在fortran语言中，不区分英文字母的大小写

### fortran程序的主要结构

fortran程序通常以 **program** + 程序名 进行开头，以 **end** 进行结尾

```
program main

! This is a comment line; it is ignored by the compiler
print *, 'Hello, World!'

end program main
! end program
! end
```

### fortran的两种编写格式

Fortran语言有两种主要的编写格式，分别是**固定格式** (Fixed Form) 和**自由格式** (Free Form)。

固定格式 (Fixed Form) 和自由格式 (Free Form) 的对比可以通过以下表格来展示：

特点	固定格式	自由格式
列限制	有，前6列用于行号和标签字段	无列限制，代码可以从任意列开始
缩进	有，以6个空格为一个缩进级别	无缩进要求，根据个人偏好进行
行长度	有限制，不超过72列，超出部分需要使用连字符	无行长度限制，可以编写较长的代码
注释	以"!"开头，从第7列开始	可以从任意列开始，使用"!"
标识符	有限制，通常在第7列到第72列之间	无限制，可以出现在任意列
扩展名	往往以.for或.f为扩展名	往往以.f90为扩展名

通过以上对比，可以看出自由格式相对于固定格式更加灵活和现代化。

自由格式在大多数现代Fortran编译器中得到广泛支持，并成为**Fortran 90**及其后续版本中的推荐编写格式。

固定格式仍然保留是为了向后兼容和支持旧有的代码库。

### fortran的数据类型

Fortran中基本的数据类型及其描述如下：

数据类型	描述
Integer	整数类型，用于存储整数值。
Real	浮点数类型，用于存储实数值。（小数和整数）
Complex	复数类型，包含实部和虚部，用于存储复数值。

数据类型	描述
Character	字符类型，用于存储字符和字符串值。
Logical	逻辑类型，用于存储逻辑值（True或False）。

这些数据类型在Fortran中用于声明变量，以便存储不同类型的数据。

根据需要，可以选择适当的数据类型来满足特定的计算和存储需求。

## fortran的数学符号

Fortran中常用的数学运算符号如下：

运算符	描述
+	加法
-	减法
*	乘法
/	除法
**	幂运算
mod()	取模运算
sqrt()	开平方
exp()	指数函数
log()	自然对数
abs()	绝对值
max()	最大值
min()	最小值
real()	转换为实数类型
int()	转换为整数类型
cmplx()	转换为复数类型

这些运算符和函数可用于进行数学计算和操作，根据需要选择适当的运算符和函数来实现所需的数学运算。

## 3-输入输出及声明

### fortran 程序的主要结构

fortran程序可以包括：主程序、子程序、函数、、、、

下面是一个使用Fortran编写的示例程序，其中包含一个子程序和一个函数调用的主程序：

```
program main_program

    !为了编写更加安全和可靠的代码，可以在程序的开头使用`implicit none`指令来禁止使用隐式类型声明
    !规则。
    implicit none

    ! 声明变量
    integer :: num1, num2, sum
    real :: radius, area

    ! 调用子程序
    call add_numbers(3, 5, sum)

    ! 调用函数
    radius = 2.5
    area = calculate_area(radius)

    ! 输出结果
    print *, "Sum of numbers: ", sum
    print *, "Area of circle: ", area

contains

    ! 子程序：将两个数相加
    subroutine add_numbers(a, b, result)
        implicit none
        integer, intent(in) :: a, b
        integer, intent(out) :: result
        result = a + b
    end subroutine add_numbers

    ! 函数：计算圆的面积
    function calculate_area(radius) result(area)
        implicit none
        real, intent(in) :: radius
        real :: area

        area = 3.14159 * radius**2
    end function calculate_area

end program main_program
```

主程序声明了变量 `num1`、`num2`、`sum` 和 `radius`、`area`，然后分别调用了一个子程序 `add_numbers` 和一个函数 `calculate_area`。

子程序 `add_numbers` 接受两个整数作为输入参数，并将它们相加，将结果存储在输出参数 `result` 中。

函数 `calculate_area` 接受一个实数作为输入参数，计算圆的面积，并将结果返回给调用者。

在主程序中，首先调用了子程序 `add_numbers` 来计算 3 和 5 的和，并将结果存储在变量 `sum` 中。

然后，调用了函数 `calculate_area` 来计算半径为 2.5 的圆的面积，并将结果存储在变量 `area` 中。

最后，使用 `print` 语句输出结果。

！！！在 Fortran 中，子程序使用 `subroutine` 关键字定义，函数使用 `function` 关键字定义。

！！！子程序可以通过 `call` 语句进行调用，而函数可以通过赋值语句将返回值赋给变量。

注意：fortran 中的 `contains` 用法和作用

### fortran 中的 `contains`

`contains` 是 Fortran 中的一个关键字，用于标识程序的主体部分中包含子程序或函数定义的位置。

在 Fortran 程序中，`contains` 关键字通常用于将子程序或函数的定义放置在主程序内部。

通过使用 `contains` 关键字，可以将相关的子程序或函数与主程序组织在一起，提高代码的可读性和可维护性。

`contains` 关键字通常位于主程序的末尾，在它之后可以定义一个或多个子程序或函数，

这些子程序或函数可以在主程序中进行调用，以实现代码的模块化和重用。

以下是一个示例程序，展示了 `contains` 关键字的使用：

```
program main_program
    implicit none

    ! 主程序变量声明

    ! 主程序执行语句

    contains
        ! 子程序或函数定义

    end program main_program
```

## 输出命令

### 1-write

在 Fortran 中，`write` 语句用于将数据写入输出文件或标准输出（通常是屏幕）。`write` 语句的一般语法为：

```
write(unit, format) [output_list]
```

其中：

- **unit** 是一个整数表达式，表示输出的目标单元。
  - 通常，使用 \* 表示标准输出（屏幕），或使用一个整数值表示一个打开的输出文件的逻辑单元号
  - 数字6不可用，因为6默认为电脑屏幕。
- **format** 是一个格式控制字符串，指定输出的格式。
  - 格式控制字符串可以包含格式编辑描述符和其他字符，用于控制输出的布局和格式。
  - \* 表示不限定格式
- **output\_list** 是一个由逗号分隔的输出项列表，用于指定要写入的数据。

注意 :

- 每执行一次 write，都会自动换到下一行
  - 不换行输出的话，使用 **write(\*,'\$(\*)')** 输出内容
- 可同时输出多个数据，中间使用逗号进行隔开

以下是一些常用的 **write** 语句示例：

1. 将一个字符串写入标准输出（屏幕）：

```
write(*,*) "Hello, World!"
```

2. 将一个整数和一个实数写入标准输出（屏幕）：

```
integer :: num = 42
real :: pi = 3.14159
write(*,*) num, pi
```

3. 将数据写入文件：

```
integer :: unit_number
unit_number = 10

! 打开输出文件
open(unit=unit_number, file="output.txt", status='replace')

! 将数据写入文件
write(unit_number, *) "Hello, File!"
write(unit_number, *) 42

! 关闭文件
close(unit_number)
```

注意，格式控制字符串中的格式编辑描述符和其他字符的使用方式和语法会有所不同，具体取决于所使用的Fortran版本和编译器。因此，建议参考Fortran编译器的文档以了解更多关于格式控制字符串的详细信息。

## 2-print

print的用法和write的用法大致相同，但是其后面不适用括号，而且只有一个\*号，表示不限定输出格式

print缺少指定设备输出的能力，仅仅只能针对屏幕输出来使用

例如：

```
print *, "hello world!"
print *, 1, 2.0, "string"
```

## 输入命令

在Fortran中，**read**语句用于从输入文件或标准输入（通常是键盘）读取数据。**read**语句的一般语法如下：

```
read(unit, format) [input_list]
```

其中：

- **unit**是一个整数表达式，表示输入的来源单元。通常，使用**\***或者**5**表示标准输入（键盘），或使用一个整数值表示一个打开的输入文件的逻辑单元号。
- **format**是一个格式控制字符串，指定输入的格式。格式控制字符串可以包含格式编辑描述符和其他字符，用于指定输入的布局和格式。
- **input\_list**是一个由逗号分隔的输入项列表，用于指定要读取的数据。

常用的**read**语句示例：

1. 从标准输入（键盘）读取一个整数：

```
integer :: num  
read(*, *) num
```

2. 从标准输入（键盘）读取一个实数和一个字符串：

```
real :: value  
character(len=20) :: text  
read(*, *) value, text
```

3. 从文件读取数据：

```
integer :: unit_number  
unit_number = 20  
  
! 打开输入文件  
open(unit=unit_number, file="input.txt", status='old')  
  
! 从文件读取数据  
read(unit_number, *) num  
read(unit_number, *) value, text  
  
! 关闭文件  
close(unit_number)
```

注意

- 在输入时，一定要注意对应的变量类型！！！
- 若输入内容出现空格，输入的字符串会被截断，只能得到第一个字符串的内容。
  - 用引号封装字符串
  - 使用格式化进行输入和输出

例如：

```
program main  
    character(len=100) :: str !需要指定字符串的最大长度  
    read(*,*) str  
    write(*,*) str  
end
```

执行结果：

```

program main
    character(len=100) :: str !需要指定字符串的最大长度
    read(*,*) str
    write(*,*) str
end

```

命令行参数:

标准输入:  
hello world

运行结果:

标准输出:  
hello

```

program main
    character(len=100) :: str !需要指定字符串的最大长度
    read(*,*) str
    write(*,*) str
end

```

命令行参数:

标准输入:  
"hello world"

运行结果:

标准输出:  
hello world

## 声明 (变量)

在程序代码中，向编译器要求预留一些存放数据的内存空间。

### 声明变量

声明变量的原则：

1. 变量必须在使用之前进行声明。在程序的开头或子程序的开头，需要使用 **INTEGER**、**REAL**、**CHARACTER** 等关键字声明变量的类型。
2. 变量名必须以字母开头，并且可以包含字母、数字和下划线。变量名长度不能超过31个字符。
3. 在fortran中，变量是不区分大小写的。
4. 变量可以使用 **IMPLICIT NONE** 语句来禁用隐式声明。这样可以确保所有变量都必须显式声明，防止可能的错误。
5. 变量可以在声明时进行初始化，例如 **INTEGER :: myVariable = 10**。如果未初始化，则变量的值是未定义的。

## 整数类型 integer

分类

- 长整型（默认）
  - interger(kind=4) a
- 短整型
  - interger(kind=2) b

注意：64位系统还支持kind=8，也就是64bit

这里的2，4指的是该变量所占的字节数 ( $2^2$ 、 $2^4$ ) 。

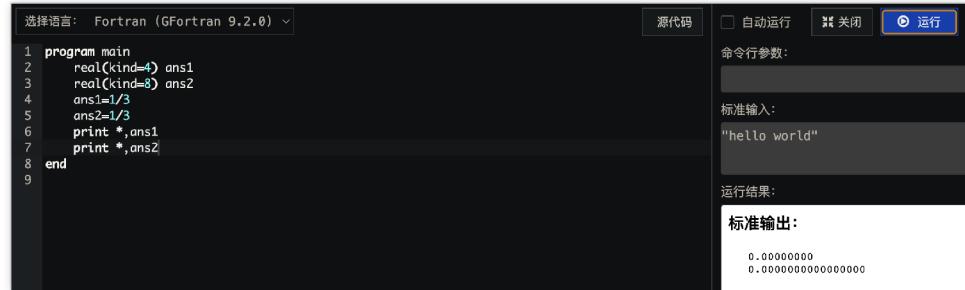
## 实数类型 real

实数和整数最大的区别在于：实数可以保存小数

## 分类

- 单精度（默认）
  - `real(kind=4) a`
- 双精度
  - `real(kind=8) b`

```
program main
    real(kind=4) ans1
    real(kind=8) ans2
    ans1=1/3
    ans2=1/3
    print *,ans1
    print *,ans2
end
```



💡 注意：在给fortran的数学库函数传入参数时，需要使用real类型

## kind的使用

```
integer(kind=1) -27~27-1
integer(kind=2) -215~215-1
integer(kind=4) -231~231-1
real(kind=4) ±1.18*10-38~±3.4*1038
real(kind=8) ±2.23*10-308~±1.79*10308
```

## 复数 complex

**COMPLEX** 是Fortran中的关键字，用于声明复数类型的变量。复数是由实部和虚部组成的数值类型。

在Fortran中，复数类型的声明方式如下：

```
COMPLEX :: z
```

在上述示例中，**z**是一个复数类型的变量。Fortran中的复数类型由两个浮点数组成，一个表示实部，另一个表示虚部。实部和虚部可以是单精度或双精度浮点数。

复数类型的变量可以进行各种复数运算，例如加法、减法、乘法和除法。Fortran提供了一些内置的复数运算函数和操作符，用于处理复数类型的变量。

以下是一些常见的复数运算函数和操作符：

- **cmplx(real, imag)**: 该函数用于创建一个复数，其中 **real** 表示实部，**imag** 表示虚部。
  - 例如，**z = CMPLX(1.0, 2.0)** 将创建一个复数 **z**，其实部为 1.0，虚部为 2.0。
  - 或者也可以在声明完之后，直接通过括号进行赋值。
- **REAL(z)**: 该函数用于获取复数 **z** 的实部。
- **AIMAG(z)**: 该函数用于获取复数 **z** 的虚部。
- **CONJG(z)**: 该函数用于获取复数 **z** 的共轭复数。
- **z1 + z2**: 该操作符用于将两个复数 **z1** 和 **z2** 相加。
- **z1 - z2**: 该操作符用于将复数 **z2** 从复数 **z1** 中减去。
- **z1 \* z2**: 该操作符用于将两个复数 **z1** 和 **z2** 相乘。
- **z1 / z2**: 该操作符用于将复数 **z1** 除以复数 **z2**。

Fortran还提供了其他一些复数相关的函数和操作符，用于处理复数类型的变量。通过使用这些函数和操作符，可以进行复数运算和处理复数类型的数据。

以下是一个使用复数类型进行计算的简单Fortran示例：

```
program complex_example
    implicit none

    complex :: z1, z2, z_sum, z_product

    ! 初始化复数
    z1 = cmplx(1.0, 2.0)
    z2 = cmplx(3.0, -1.0)

    ! 复数运算
    z_sum = z1 + z2
    z_product = z1 * z2

    ! 输出结果
    print *, 'z1 = ', real(z1), ' + ', aimag(z1), 'i'
    write(*, *) 'z2 = ', real(z2), ' + ', aimag(z2), 'i'
    write(*, *) 'z1 + z2 = ', real(z_sum), ' + ', aimag(z_sum), 'i'
    write(*, *) 'z1 * z2 = ', real(z_product), ' + ', aimag(z_product), 'i'

end program complex_example
```

```

program complex_example
implicit none

complex :: z1, z2, z_sum, z_product

! 初始化复数
z1 = cmplx(1.0, 2.0)
z2 = cmplx(3.0, -1.0)

! 复数运算
z_sum = z1 + z2
z_product = z1 * z2

! 输出结果
print *, 'z1 = ', real(z1), ' + ', aimag(z1), 'i'
write(*, *) 'z2 = ', real(z2), ' + ', aimag(z2), 'i'
write(*, *) 'z1 + z2 = ', real(z_sum), ' + ', aimag(z_sum), 'i'
write(*, *) 'z1 * z2 = ', real(z_product), ' + ', aimag(z_product), 'i'

end program complex_example

```

## 字符以及字符串类型（character）

### 概述

字符类型是用来保存一个字符或一长串字符所组成的字符串时，所使用的类型

注意：★★★，在fortran中需要声明长度 `character(len=100) str`

默认字符长度为1，也就是一个字符长度。

使用 `character` 类型的Fortran示例：

```

program character_example
implicit none

character :: letter
character(5) :: word
character(10) :: sentence = "Hello World"

! 单个字符赋值
letter = 'A'

! 字符串赋值
word = 'Hello'

! 输出结果
write(*, *) letter
write(*, *) word
write(*, *) sentence

end program character_example

```

在上述示例中，我们使用小写字母声明了三个 `character` 类型的变量：`letter`、`word` 和 `sentence`。`letter` 用于在fortran中，可以改变字符串的某一部分，比如：

要改变字符串的某一部分，可以使用字符串切片和字符串连接的方式。

```

program modify_string_example
    implicit none

    character(20) :: sentence = "Hello, World!"
    character(20) :: modified_sentence
    !通过双闭区间，来实现对单个元素的访问和修改
    print *, sentence(1:1)

    sentence(1:1) = 'h'

    ! 使用字符串切片和连接来修改字符串
    modified_sentence = sentence(1:6) // "Everyone" // sentence(13:)
    ! 输出结果
    write(*, '(a)') modified_sentence

end program modify_string_example

```

注意💡：

- 切片操作和python基本上是类似的
- 但是，计数是从1开始进行计数的，也就是说，1就是代表第一个字符
- 而且，区间的范围是，前闭后闭，由此可以实现对单个元素进行访问和修改
- 连接字符是通过//来进行连接的，而不是+



```

选择语言: Fortran (GFortran 9.2.0) ▾ 源代码
1 program modify_string_example
2     implicit none
3
4     character(20) :: sentence = "Hello, World!"
5     character(20) :: modified_sentence
6
7     print *, sentence(1:1) !通过双闭区间，来实现对单个元素的访问和修改
8
9     sentence(1:1) = 'h'
10
11    ! 使用字符串切片和连接来修改字符串
12    modified_sentence = sentence(1:6) // "Everyone" // sentence(13:)
13
14    ! 输出结果
15    write(*, '(a)') modified_sentence
16
17 end program modify_string_example

```

## 字符串函数

常用的字符串函数：

1. **len(string)**：返回字符串的长度。
2. **trim(string)**：删除字符串末尾的空格。
3. **adjustl(string)**：将字符串左对齐，删除开头的空格。
4. **adjustr(string)**：将字符串右对齐，删除末尾的空格。
5. **index(string, substring)**：返回子字符串在字符串中的第一次出现的位置。
6. **scan(string, set)**：返回字符串中第一个与给定字符集中的字符匹配的位置。
7. **verify(string, set)**：返回字符串中第一个与给定字符集中的字符不匹配的位置。
8. **repeat(string, count)**：将字符串重复指定次数。
9. **trim(leading/trailing/both, string, trim\_chars)**：删除字符串开头/末尾/两端的指定字符集。

- **trim(string)** : 表示去除尾端空格后的字符串
10. **transfer(source, dest)** : 将一个字符串转换为另一个字符串，可以改变字符串的长度。

## 逻辑变量 logical

在fortran中，**logical**是用于表示逻辑值的类型。**logical**类型的变量只能取两个值之一：**.true.**（真）或**.false.**（假）。

★★★：在将true或false赋值给logical变量之前，需要在两个单词前面和后面分别加上`.`。

使用**logical**类型的示例：

```
program logical_example
implicit none

logical :: is_true
logical :: is_false = .false.

! 逻辑值赋值
is_true = .true.

! 输出结果
write(*, '(l)') is_true
write(*, '(l)') is_false

end program logical_example
```

## 常数 parameter

常数只能在声明时通过parameter来设置数值，而且只能设置一次

```
real pi
parameter(pi=3.1415926)
```

```
real,parameter :: pi = 3.1415926
```

## 自定义数据类型 type

自定义数据类型：fortran 能够自由组合一些基本数据类型，创造出一个更复杂类型组合。

类似于c中的struct，结构体

例如：

```
module personmodule
    type :: person
        character(50) :: name
        integer :: age
        character(20) :: occupation
    end type person
end module personmodule

program personexample
    use personmodule
    type(person) :: john
    john%name = "John Smith" !这里的%，相当于c中的.
    john%age = 30
    john%occupation = "Engineer"
    write(*, *) "Name:", john%name
    write(*, *) "Age:", john%age
    write(*, *) "Occupation:", john%occupation
end program personexample
```

```
module personmodule
    type :: person
        character(50) :: name
        integer :: age
        character(20) :: occupation
    end type person
end module personmodule

program personexample
    use personmodule
    type(person) :: john
    john = person("John Smith", 30, "Engineer")
    write(*, *) "Name:", john%name
    write(*, *) "Age:", john%age
    write(*, *) "Occupation:", john%occupation
end program personexample
```

## 等价声明 equivalence

等价声明：把两个以上的变量，声明使用同一内存地址

使用同一内存位置的变量，只要改变其中一个变量，就会同时改变其他变量的数值

例如：

```
program example
    integer a,b
    equivalence(a,b)
        a=3
    print *,a,b
        b=4
    print *,a,b
end program example
```

## 标准输出：

3  
4

3  
4

## 声明在程序结构中的位置

- 声明的位置应该放在程序代码的可执行命令之前（这点和c++不一样）
- 在程序代码开始出现数值计算和输入输出命令时，就不能再声明变量了

主要结构如下：

```
program main
    implicit none

    声明变量

    可执行命令（赋值、计算、输入输出等）
end program main
```

## 格式化输入输出

格式化输出的目的，就是要把数据经过有计划的版面设计显示出来

在某些情况下，要读取数据时，要设置恰当的输入格式

### 格式化概述

格式化就是写在write和read命令中第二个参数下的东西。

比如

```
write(*,"(1x,i5)") a
```

- **(1X, I5)** 是格式控制字符串，它指定了输出的格式。在这个例子中 - **1X** 表示在输出前添加一个空格，**1** 表示一个字段的宽度，**X** 表示添加空格。 - **I5** 表示输出一个宽度为 5 的整数。

汇总表格如下：

<b>Aw</b>	以w个字符宽来输出字符串
<b>BN</b>	定义文本框中的空位为没有东西，在输入时才需要使用
<b>BZ</b>	定义文本框中的空位代表0，在输入时才需要使用
<b>Dw. d</b>	以w个字符宽来输出指数类型的浮点数，小数部分占d个字符宽
<b>Ew. d[Ee]</b>	以w个字符宽来输出指数类型的浮点数，小数部分占d个字符宽，指数部分占e个字符
<b>EW. d[Fe]</b>	以指数类型来输出浮点数
<b>ESw. d[Fe]</b>	以指数类型来输出浮点数
<b>Fw. d</b>	以w个字符宽来输出浮点数，小数部分占d个字符宽
<b>Gw. d[Fe]</b>	以w个字符宽来输出整数，最少输出m个数字
<b>Iw[, m]</b>	以w个字符宽来输出整数，最少输出m个数字
<b>Lw</b>	以w个字符宽来输出T或F的真假值
<b>nX</b>	把输出的位置向右跳过n个位置
<b>/</b>	代表换行
<b>:</b>	在没有更多数据时结束输出
<b>KP</b>	K值控制输入输出的SCALE
<b>Tn</b>	输出的位置移动到本行第n列
<b>TLn</b>	输出的位置向左相对移动n列
<b>TRn</b>	输出的位置向右相对移动n列
<b>SP</b>	在数值为正时加上“正号”
<b>SS</b>	取消SP
<b>Bw[, m]</b>	把整数转换成二进制来输出、输出会占w个字符宽，固定输出m个数字。m值可以不给定
<b>Ow[, m]</b>	把整数转换成八进制来输出，输出会占w个字符宽，固定输出m个数字。m值可以不给定
<b>Zw[, m]</b>	把整数转换成十六进制来输出，输出会占w个字符宽，固定输出m个数字。m值可以不给定

## 格式化输出详细介绍

if e a x 是最常用的几个格式

- i (integer) : 整型
- f (float) : 小数
- e : 科学计数法小数
- a : 字符
- x : 空格

## Iw 规定字符长来输出整数

格式： **Iw [ .m ]** 表示：以w个字符长度来输出整数，至少输出m个数字

`write(*,"(I5)") 100`

以5个字符的长度来输出一个整数



输出结果前面会补2个空格

`write(*,"(I3)") 1000`



输出格式设置位数不足时，会输出\*

`write(*,"(I5.3)") 10`

以5个字符的长度来输出一个整数，  
至少输出3个数字，位数不足补0



输出结果前补2个空格，1个0

## Fw.d 规定字符长来输出实数

格式： **Fw.d** 表示：以w个字符长来输出实数，小数部分占d个字符

`write(*,"(F9.3)") 123.45`

以9个字符长度来输出实数，小数部分占3个位数



总长度不足9位，前面补空格，  
小数部分不足3位，后面补0

`write(*,"(F6.3)") 123.45`



总长度减小数部分与小数点长度后，  
整数部分长度不足会输出\*

`write(*,"(F9.1)") 123.45`

以9个字符长度来输出实数，小数  
部分占1个位数

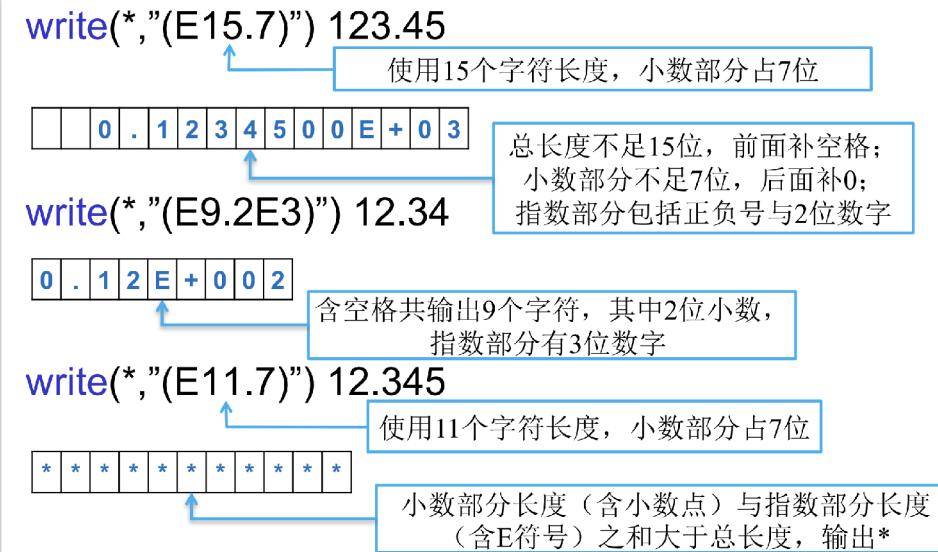


小数部分超过1位，舍入

## Ew.d 规定字符长度来输出实数

形式：**Ew.d[Ee]** 表示：用科学计数法，以w个字符长来输出实数，小数部分占d个字符长，指数部分最少输出e个字符。

： 指数部分默认为**2**位数字



## Aw 规定字符长度来输出字符串

形式: Aw 表示: 以w个字符长来输出字符串



## nX 输出位置移动

形式: nX 表示: 输出位置向右移动n位

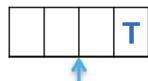


## Lw 规定输出布尔值的字符长度

形式: Lw 表示: 以w个字符长来输出T或F的真假值

`write(*,"(L4)") .true.`

使用4个字符长度输出逻辑变量



输出3个空格和1个T

## Gw.d 规定字符长输出所有类型的数据

以w个字符长来输出所有类型的数据，d不一定会使用，但是不能省略

- 用来输入/输出 **字符/整数/逻辑数** 时，Gw.d=Aw/Iw/Lw，其中，d必须随便给一个数字，不能省略
- 用来输入或输出 **实数** 时，Gw.d=Fw.d或Ew.d
  - 由于在以科学计数法输出时，具有 **0 . + E** 四个字符的占位而且指数部分的位数为2位，因此如果能表示为科学计数法，则优先表示为科学计数法，如果不能，则表示为浮点数，浮点数的形式为：(Fa.b,4X)，其中，该4x将不会显示出来，因为nx格式必须在fa.b指令前，才能进行显示

例如：

```
program example

  write(*,"(G9.4)") 123.0    !等同于F5.1

  write(*,"(G9.2)") 123.0    !等同于E9.2

  write(*,"(4X,F5.1)") 123.0

  write(*,"(F5.1,4X)") 123.0    !等同于F5.1

end program example

!123.0 = 0.12E+03 (2位小数)
!123.0 = 0.1230E+03(4位小数)
```

```
program example

  write(*,"(G9.4)") 123.0    !等同于F5.1,4X
  write(*,"(G9.2)") 123.0    !等同于E9.2
  write(*,"(4X,F5.1)") 123.0    !等同于F5.1,4X
  write(*,"(F5.1,4X)") 123.0    !等同于F5.1

end program example
```

命令行参数：

标准输入：

运行结果：

标准输出：

123.0  
0.12E+03  
123.0  
123.0

## / 换行符

在格式化中，添加 /，表示换行输出

```
write(*,"(I3//I3)") 10, 10 ! / 代表换行  
write(*,"(I3, /, /, I3)") 10, 10 ! 可以中间加逗号
```

```
program example  
print *,"-----"  
write(*,"(I3//I3)") 10, 10 ! 代表换行  
print *,"-----"  
write(*,"(I3, /, /, I3)") 10, 10 ! 可以中间加逗号  
print *,"-----"  
end program example
```

命令行参数:  
标准输入:  
运行结果:  
标准输出:  
-----  
10  
10  
10  
10  
-----

## Tn 移动输出位置

把输出的位置移动到本行的第n个字符

```
program example  
write(*,"(T3,I3)") 100 ! 把输出位置移动到第3个字符  
end program example
```

## 重复地以相同的格式输出数据

```
program example  
write(*,"(3(1x,f5.2))") 1.0,2.0,3.0  
write(*,"(3f6.2)") 1.0,2.0,3.0  
end program example
```

```
program example  
write(*,"(3(1x,f5.2))") 1.0,2.0,3.0  
write(*,"(3f6.2)") 1.0,2.0,3.0  
end program example
```

命令行参数:  
标准输入:  
运行结果:  
标准输出:  
1.00 2.00 3.00  
1.00 2.00 3.00

## 在格式化中输出字符串

```
program example  
write(*,"('3 + 4 =' ,1x,I1)") 3+4 ! 用单引号封装字符  
write(*,'("3 + 4 =" ,1x,I1)') 3+4 ! 用双引号封装字符  
end program example
```

```
program example
write(*,"('3 + 4 =' ,1x,I1)") 3+4 ! 用单引号封装字符
write(*,'("3 + 4 =" ,1x,I1)") 3+4 ! 用双引号封装字符
end program example
```

命令行参数:  
标准输入:  
运行结果:  
标准输出:  
3 + 4 = 7  
3 + 4 = 7

## 将输出格式变为字符串

可以把输出格式放在字符串变量中

```
program example
character(len=100) :: fmt = "('3 + 4 =' ,1x,I1)"
write(*,"('3 + 4 =' ,1x,I1)") 3+4 ! 用单引号封装字符
write(*,'("3 + 4 =" ,1x,I1)") 3+4 ! 用双引号封装字符
print fmt,3+4 ! 使用输出格式字符串进行格式化输出, 方便对输出格式进行控制
end program example
```

```
program example
character(len=100) :: fmt = "('3 + 4 =' ,1x,I1)"
write(*,"('3 + 4 =' ,1x,I1)") 3+4 ! 用单引号封装字符
write(*,'("3 + 4 =" ,1x,I1)") 3+4 ! 用双引号封装字符
print fmt,3+4
end program example
```

命令行参数:  
标准输入:  
运行结果:  
标准输出:  
3 + 4 = 7  
3 + 4 = 7  
3 + 4 = 7

## implicit命令

### implicit 设置默认类型

在Fortran中，**implicit**语句用于指定变量的隐式类型。它控制了在没有显式声明的情况下，变量名的首字母所隐含的数据类型。

默认情况下，Fortran使用以下隐式规则来确定变量的类型：

- 变量名以 **I**, **J**, **K**, **L**, **M**, 或 **N** 开头，默認為整数类型。
- 变量名以其他字母开头，默認為实数类型。

示例：

```
program implicit_example
    implicit integer(i-n)
    implicit real*8(x-z)

    i = 42
    x = 3.14

    write(*, *) i
    write(*, *) x

end program implicit_example
```

- **implicit integer(i-n)** 表示以字母 I 到 N 开头的变量默认为整数类型。
- **implicit real\*8(x-z)** 表示以字母 X 到 Z 开头的变量默认为双精度实数类型。

由于使用了 **IMPLICIT** 语句，我们无需显式声明变量的类型。

在赋值语句中，我们将整数值 42 赋给 **i**，将双精度实数值 3.14 赋给 **x**。

最后，我们使用 **WRITE** 语句将变量 **i** 和 **x** 的值输出到标准输出。

## implicit none 关闭默认类型

当关闭默认类型功能时，所有变量都需要进行声明

implicit命令必须放在program命令的下一行

以下是一个示例：

```
program implicit_example
    implicit none

    integer :: i
    real :: x

    i = 42
    x = 3.14

    write(*, *) i
    write(*, *) x

end program implicit_example
```

在上述示例中，我们使用了 **implicit none** 语句，它告诉Fortran禁止隐式声明变量的类型。这意味着所有的变量必须显式地声明其类型。

然后，我们声明了一个整数变量 **i** 和一个实数变量 **x**。由于 **implicit none** 的存在，我们必须显式地指定这些变量的类型。

我们将整数值 42 赋给 `i`，将实数值 3.14 赋给 `x`。

最后，我们使用 `write` 语句将变量 `i` 和 `x` 的值输出到标准输出。

通过使用 `implicit none`，我们可以确保所有的变量都被显式声明，增加了代码的可读性和可维护性，并减少了由于隐式声明引起的潜在错误。

# 4-流程控制与逻辑运算

## IF 语句

概述：

- 一个IF模块
- 一个IF-ELSE模块
- 多个IF模块
- 多重判断IF-ELSE IF模块

### 基本用法

```
if(逻辑判断式) 执行代码
```

```
if(逻辑判断式) then  
    执行代码  
else  
    执行代码  
endif
```

```
if(条件1) then  
    . . .  
else if (条件2) then  
    . . .  
else  
    . . .  
endif
```

```

if(...) then
    if(...) then
        if(...) then
            ...
        else
            ...
        endif
    else
    ...
endif
else
...
endif

```

## 逻辑运算

### 1-两个数字比较大小（使用逻辑运算符）

：比较大小也可以用在字符串上面

F90以上	F77	说明
==	.EQ.	判断是否“相等”
/=	.NE.	判断是否“不相等”
>	.GT.	判断是否“大于”
>=	.GE.	判断是否“大于或等于”
<	.LT.	判断是否“小于”
<=	.LE.	判断是否“小于或等于”

用法：

```

if(a .gt. 100) then
    ...
endif

```

```

if(a > 100) then
    ...
endif

```

## 2-由两个或多个小逻辑表达式组成（集合运算符）

.AND.	交集，如果两边的表达式都成立，整个表达式就成立
.OR.	并集，两边的表达式只要有一个成立，整个表达式就成立
.NOT.	如果后面的表达式不成立，整个表达式就成立
.EQV.	两边表达式的逻辑运算结果相同时，整个表达式就成立
.NEQV.	两边表达式的逻辑运算结果不同时，整个表达式就成立

例如：

```
if(a>=80 .and. a<90) then
  ...
endif

if((a>0 .and. b>0) .or. (a<0 .and. b<0)) then
  ...
endif
```

## select case 语句

示例：

```
select case(变量)
case(值1)
  ...
case(值2)
  ...
case(值n)
  ...
case default
  ...
end select
```

注意：在case里的冒号前后放入两个数值时，代表这两个数字范围中的所有数值（闭区间），还可以用逗号表示放入多个变量

```
case(1) ! 当变量为1时，会执行这个case中的程序模块
case(1:5) ! 1<=变量<=5时，会执行这个case中的程序模块
case(1:) ! 1<=变量    会执行这个case中的程序模块
case(:5) ! 变量<=5    会执行这个case中的程序模块
case(1,3,5) ! 变量=1   3   5时    会执行这个case中的程序模块
```

★★：在使用select-case时的限制：

- 只能使用整数、字符和逻辑变量，不能使用浮点数和复数

- 每个case中所使用的数值必须是固定的常量，不能使用变量

## **pause、 continue、 stop**

### **pause**

暂停执行，按下enter键，才会继续执行

### **continue**

继续向下执行程序

### **stop**

结束程序的执行

## 5-循环

### 固定次数的do循环结构

```
do counter=min,max,stride  
    ...  
end do
```

- counter: 循环的次数根据其数值而定
- min: counter的起始数值
- max: counter<=max时, 循环执行
- stride: 增量, 每次循环后, counter增加的数值, 默认为1, 可以省略
  - 增量如果为正数, 则为从小往大增加
  - 增量如果为负数, 则为从大往小减少

```
for(int counter=min; counter<=max; counter=counter+stride){  
    ...  
}
```

与c不同的是, 用来作为计数器的变量, 在循环中不能再使用命令去改变它的数值, 否则编译不通过。

do循环结构也可以进行多层循环进行嵌套

### do while 循环

循环不一定要由计数器的增、减来决定是否结束循环, 它可以由条件来做决定

```
do while(逻辑式)  
    ...  
    ...  
end do
```

### 循环的流程控制

#### cycle

作用: 略过当前循环, 直接跳回循环开头, 进行下一次的循环, 通常与if判断组合使用

例如:

```

program example
    implicit none
    integer :: dest = 9
    integer floor
    do floor=1, dest
        if ( floor==4 ) cycle
        write(*,"(I2)") floor
    end do
    stop
end

```

The screenshot shows a software interface for running Fortran code. On the left is the source code editor with the following text:

```

1 program example
2     implicit none
3     integer :: dest = 9
4     integer floor
5     do floor=1, dest
6         if ( floor==4 ) cycle
7         write(*,"(I2)") floor
8     end do
9     stop
10 end

```

The line "10 end" is highlighted with a yellow background. On the right side, there are several panels:

- 源代码**: A checkbox labeled "自动运行" (Automatic Run) is checked.
- 命令行参数:** An empty input field.
- 标准输入:** An empty input field.
- 运行结果:** An empty output field.
- 标准输出:** A panel containing the numbers 1, 2, 3, 5, 6, 7, 8, and 9, representing the output of the program.

类似于c中的continue语句

## exit

作用：直接强制跳出当前运行的循环

类似与c中的break语句

例如：

```

program example
    implicit none
    integer :: dest = 9
    integer floor
    do floor=1, dest
        if ( floor==4 ) then
            print "(1X,A)","程序结束, exit"
            exit
        end if
        write(*,"(I2)") floor
    end do
    stop
end

```

```

1 program example
2     implicit none
3     integer :: dest = 9
4     integer floor
5     do floor=1, dest
6         if ( floor==4 ) then
7             print "(1X,A)", "程序结束, exit"
8             exit
9         end if
10        write(*,"(I2)") floor
11    end do
12    stop
13 end

```

源代码  自动运行  
命令行参数:  
标准输入:  
运行结果:  
**标准输出:**  
1  
2  
3  
程序结束, exit

## 署名的循环

循环可以取名字

- 可以在编写循环时，知道end do 这个描述的位置是否正确
- 可以配合cycle、exit来进行使用

例如：

```

program main
    implicit none
    integer :: i, j
    loop1: do i=1,3
        if ( i==3 ) exit loop1 ! 跳离loop1循环
        loop2: do j=1,3
            if ( j==2 ) cycle loop2 ! 重做loop2循环
            write(*, "('(',i2,',',i2,')')") i, j
        end do loop2
    end do loop1
    stop
end

```

```

program main
    implicit none
    integer :: i, j
    loop1: do i=1,3
        if ( i==3 ) exit loop1 ! 跳离loop1循环
        loop2: do j=1,3
            if ( j==2 ) cycle loop2 ! 重做loop2循环
            write(*, "('(',i2,',',i2,')')") i, j
        end do loop2
    end do loop1
    stop
end

```

命令行参数:  
标准输入:  
运行结果:  
**标准输出:**  
( 1, 1)  
( 1, 3)  
( 2, 1)  
( 2, 3)

# 6-数组 array

## 基本使用

### 一维数组

数组可以一次声明处一长串相同数据类型的变量

声明数组的语法（以非自定义类型integer为例）

- `integer a(10)`
- `integer,dimension(10) :: a` : 先声明a是整型，再声明a是大小为10的数组

自定义类型的数组声明

- `type(person)::a(10)` : 用person新类型来声明数组

自定义类型数组的访问

- `a(2)%name` : 访问数组的第2个自定义元素的name属性

### 二维数组

声明二维数组（以非自定义类型integer为例）

- `integer a(3,3)`
- `integer,dimension(3,3) :: a`

访问：`a(1,2)`

### 多维数组

fortran 最多可以声明高达7维的数组

每一个维度都有对应的大小

```
integer a(D1,D2,...,Dn) ! n维数组  
a(I1,I2,...,In) ! 使用n维数组时，要给n个坐标值
```

### 特殊的数组声明

在没有特别赋值的情况下，数组的索引值都是从1开始的（即数组的下界默认为1）

`integer a(5) ==> a(1)、a(2)、a(3)、a(4)、a(5)`

可以特别赋值数组的坐标值的使用范围：() 指的是闭区间

- `integer a(0:5) ==> a(0)、a(1)、a(2)、a(3)、a(4)、a(5)`

- `integer a(-3:3)` ==> a(-3)、a(-2)、a(-1)、a(0)、a(1)、a(2)、a(3)
- `integer a(5, 0:5)` ==> a(1~5, 0~5)
- `integer b(2:3, -1:3)` ==> b(2~3, -1~3)

## 数组内容的设置

### 赋初值

#### data 语句

在Fortran中，**DATA**语句用于在程序中为变量赋初始值。**DATA**语句的语法如下：

```
DATA variable1 / value1 /, variable2 / value2 /, ...
```

其中，`variable1`, `variable2`, ... 是要赋值的变量，`value1`, `value2`, ... 是对应的初始值。

```
program dataexample
  integer :: x, y, z
  real :: a, b, c

  data x / 10 /, y / 20 /, z / 30 /
  data a / 1.5 /, b / 2.5 /, c / 3.5 /

  write(*, *) "x =", x
  write(*, *) "y =", y
  write(*, *) "z =", z

  write(*, *) "a =", a
  write(*, *) "b =", b
  write(*, *) "c =", c
end program dataexample
```

### 数组内容通过**data**来初始化

例如

```

program main
    implicit none
    integer a(5)
    integer b(5)
    integer c(5)
    data a /1,2,3,4,5/
    ! 不做任何赋初值操作
    data c/5*0/
    print *,a
    print *,b
    print *,c
end

```

```

program main
    implicit none
    integer a(5)
    integer b(5)
    integer c(5)
    data a /1,2,3,4,5/
    ! 不做任何赋初值操作
    data c/5*0/
    print *,a
    print *,b
    print *,c
end

```

命令行参数:

标准输入:

运行结果:

**标准输出:**

1 7 0	2 0 0	3 0 0	4 -791169654 0	5 0 0
-------------	-------------	-------------	----------------------	-------------

## 通过隐含式循环来赋值

例如:

```

integer a(5)
integer i
data (a(i),i=2,4) /2,3,4/

```

初值的设置结果为: 2, 3, 4 位置上的数组值分别为 2, 3, 4

```

program main
    integer a(8)
    integer i
    data (a(i),i=2,4) /2,3,4/
    print *,a
    print *,(a(i),i=2,4)
end

```

选择语言: Fortran (GFortran 9.2.0)

```

1 program main
2     integer a(8)
3     integer i
4     data (a(i),i=2,4) /2,3,4/
5     print *,a
6     print *,(a(i),i=2,4)
7 end
8

```

源代码

自动运行

全屏

运行

命令行参数:

标准输入:

运行结果:

**标准输出:**

0 2	2 3	3 4	4 0	0 0	0 0	0 0
--------	--------	--------	--------	--------	--------	--------

注意：

- 隐含式 `(a(i), i=min, max, stride)`
- 也可以进行多层嵌套
  - `((a(i, j), j=1, 2), i=1, 2)`

## 在Fortran90中赋初始值

可以省略 data

```
integer a(5) = (/ 1,2,3,4,5 /) ! 括号和除号之间不能有空格
```

直接把初值写在声明后面时，每个元素都要给定初始值

而且，直接赋值的 `(/ ... /)` 的 `...` 中也支持隐含式循环

```
program main
    integer :: i
    integer :: a(5) = (/ 1,2,3,4,5 /)
    integer :: b(5) = (/ 1,(0,i=2,4),5 /)
    integer :: c(5) = 100 ! 一次把整个数组内容设置为同一个数值
    print *,a
    print *,b
    print *,c
end
```

```
program main
    integer :: i
    integer :: a(5) = (/ 1,2,3,4,5 /)
    integer :: b(5) = (/ 1,(0,i=2,4),5 /)
    integer :: c(5) = 100
    print *,a
    print *,b
    print *,c
end
```

命令行参数:

标准输入:

运行结果:

标准输出:

1	2	3	4	5
1	0	0	0	5
100	100	100	100	100

1

2

3

4

5

100

100

100

100

## 对整个数组的操作

fortran 90 可以通过简单的命令来操作数组

- `a=5`
  - a是一个任意维数及大小的数组。
  - 这个命令是把数组a的每个元素的值都设置为5。
  - 以一维的情况来说，即 $a(i) = 5$
- `a=(/ 1,2,3 /)`
  - $a(1)=1, a(2)=2, a(3)=3$ 。
  - 等号右边所提供的数字个数，必须跟数组a的大小一样
- `a=b`
  - a和b是同样维数及大小的数组。
  - 这个命令会把数组a同样位置元素的内容设置成和数组b一样。

- 以一维的情况来说，即 $a(i) = b(i)$
- **$a=b+c$** 
  - a,b, c是三个同样维数及大小的数组
  - 这个命令会把数组b及c中同样位置的数值相加，得到的数值再放回数组a同样的位置中。
  - 以二维的情况来说，即 $a(i,j) = b(i,j) + c(i,j)$
- **$a=b-c$** 
  - a, b, c是三个同样维数及大小的数组
  - 这个命令会把数组b及c中同样位置的数值相减，得到的数值再放回数组a
  - 同样的位置中。以二维的情况来说，即 $a(i,j) = b(i,j) - c(i,j)$
- **$a=b*c$** 
  - a, b, c是三个同样维数及大小的数组
  - 执行后数组a的每一个元素值为相同位置的数组b元素乘以数组c元素。
  - 以二维的情况来说，即 $a(i,j) = b(i,j) \times c(i,j)$
- **$a=b/c$** 
  - a, b, c是三个同样维数及大小的数组
  - 执行后数组a的每一个元素值为相同位置的数组b元素除以数组c元素。
  - 以二维的情况来说，即 $a(i,j) = b(i,j) \div c(i,j)$
- **$a=\sin(b)$** 
  - 矩阵a的每一个元素为矩阵b元素的sin值
  - 数组b必须是实型数组，才能使用sin函数。
  - 以一维的情况来说，即 $a(i) = \sin(b(i))$
- **$a=b>c$** 
  - a,b,c是三个同样维数及大小的数组
  - 其中数组a是逻辑型数组，数组b、c则为元素间可以进行比较的变量类型。
  - 以一维情况来说，即为

```

if (b(i)>c(i)) then
    a(i)=.true.
else
    a(i)=.false.
endif

```

## 对部分数组的操作

和python中的类似，fortran中使用的前闭后闭区间

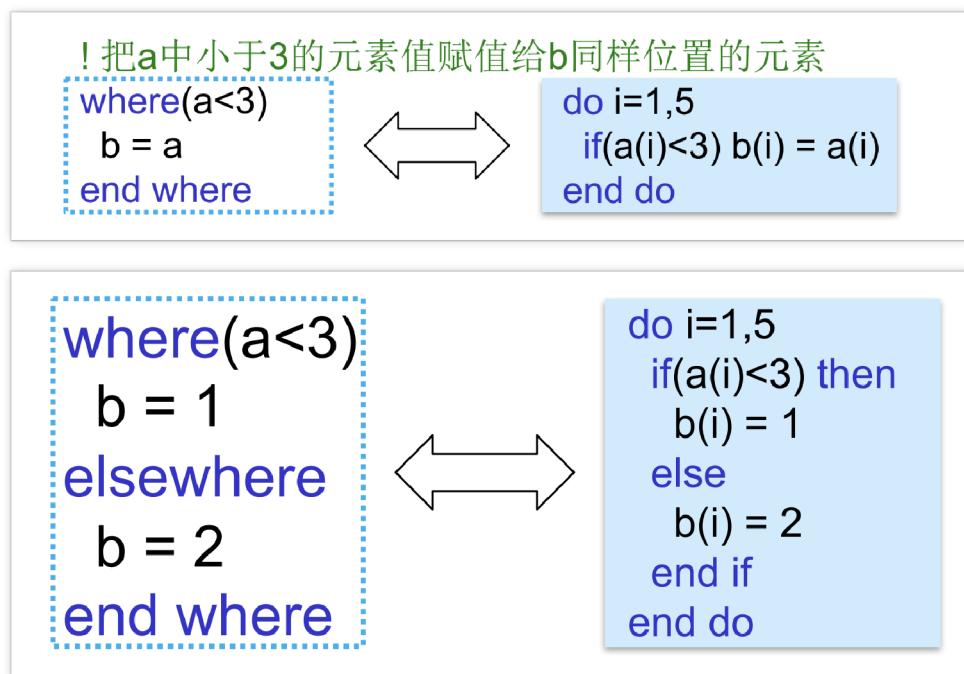
- **$a(3:3)$** 
  - 与 $a(3)$  相同
- **$a(3:5)=5$** 
  - 把 $a(3) \sim a(5)$ 的内容设置成5，其他值不变
- **$a(3:)=5$** 
  - 把 $a(3)$ 之后元素的内容设置成5，其他值不变

- **a(3:5)=(/3,4,5/)**
  - 设置a(3)=3, a(4)=4, a(5)=5, 其他值不变。
  - 等号左边所赋值的数组元素数目必须跟等号右边提供的数字个数相同
- **a(1:5:2)=3**
  - 设置a(1)=3, a(3)=3, a(5)=3。类似隐含式循环
- **a(1:10)=a(10:1:-1)**
  - 使用类似隐含式循环的方法把a(1~10)的内容给翻转
- **a(:)=b(:,2)**
  - 假设a声明为a(5), b声明为b(5,5)。
  - 等号右边b(:,2)是取出b(1~5,2)这5个元素。
  - 而a是一维数组, 所以a(:)和直接使用a是一样的, 都是指a(1~5)这5个元素。
  - 只要等号两边的元素数目一样多就成立。
  - 执行结果为a(i)=b(i,2)
- **a(:,:,1)=b(:,:,1)**
  - 假设a声明为a(5,5), b声明为b(5,5,5)。
  - 等号右边b(:,:,1)是取出b(1~5, 1~5, 1)这25个元素。
  - 而a是二维数组, 所以a(:,:,1)和直接使用a是一样的, 都是指a(1~5,1~5)这25个元素。
  - 执行结果为a(i,j)=b(i,j,1)

## where 语句

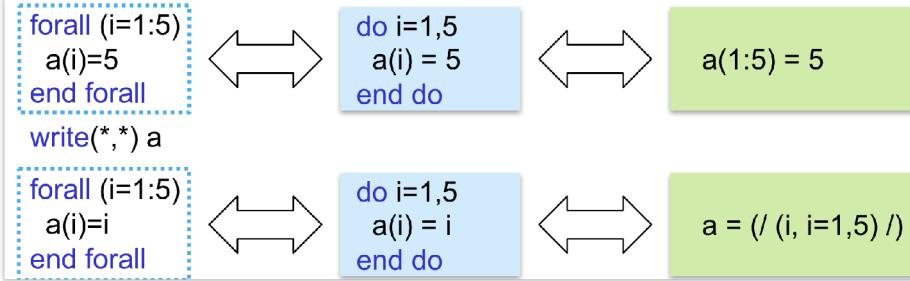
fortran 95新添加的内容

对比mysql的语句进行理解



## forall 语句

fortran 95 使用隐含式循环来使用数组的算法



## 数组的保存规则

### 一维数组

连续存放

A(5)的元素在内存中的排列情况为

$$A(1) \Rightarrow A(2) \Rightarrow A(3) \Rightarrow A(4) \Rightarrow A(5)$$

A(-1:3)的元素在内存中的排列情况为

$$A(-1) \Rightarrow A(0) \Rightarrow A(1) \Rightarrow A(2) \Rightarrow A(3)$$

### 二维数组

按照列的大小，从小往大开始排序

- A(3,3)的元素在内存中的排列情况为

$A(1,1) \Rightarrow A(2,1) \Rightarrow A(3,1)$ $\Rightarrow A(1,2) \Rightarrow A(2,2) \Rightarrow A(3,2)$ $\Rightarrow A(1,3) \Rightarrow A(2,3) \Rightarrow A(3,3)$	先放第1列中的元素 再放第2列中的元素 最后放第3列中的元素
--	--------------------------------------

所以，以后循环的内层循环应该从i开始，然后外层循环是j，这样可以提高效率

```

do i=1,4
  do j=1,4
    ...
    a(j,i)
  end do
end do

```

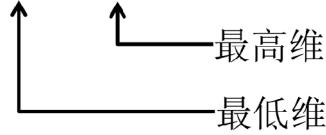
### 多维数组

先放入较低维的元素，再放入较高维的元素

最先变化的是低维的元素，最高维的元素（下标）最后变化。

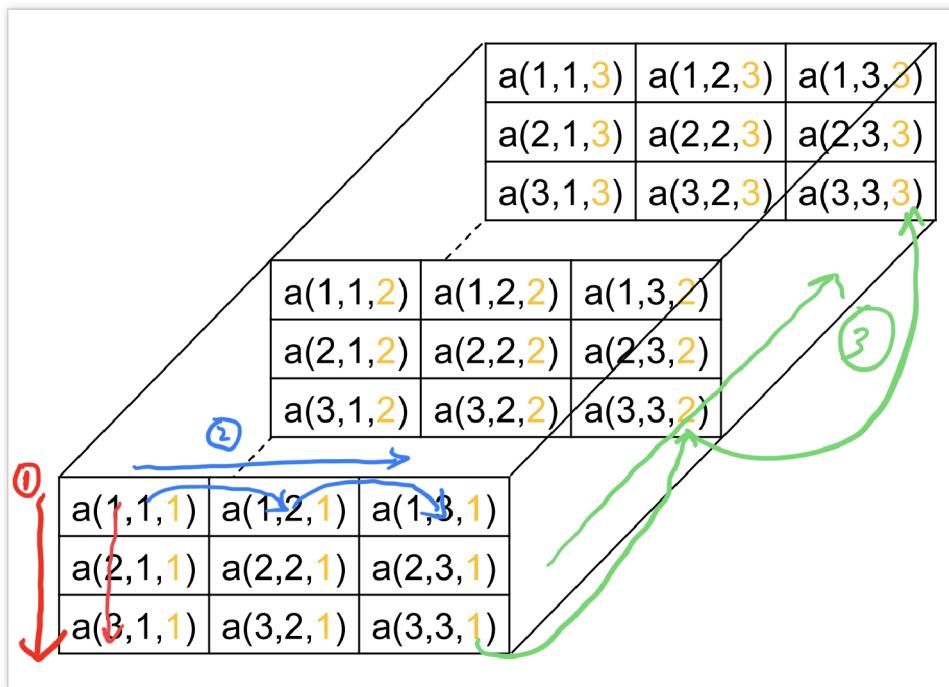
## 三维数组：

A(I, J, K)



A(1,1,1)=>A(2,1,1) 先放入第1维  
=>A(1,2,1)=>A(2,2,1) 接着放入第2维  
=>A(1,1,2)=>A(2,1,2) 接着放入第3维  
=>A(1,2,2)=>A(2,2,2)

```
do i=1,2
do j=1,2
do k=1,2
! 越小的维使用越内层的循环
write(*,*) a(k,j,i)
end do
end do
end do
```



## 数据的存储方式

根据内存的排列顺序来设置数值

```
integer :: a(2,2) = (/ 1,2,3,4 /)
! a(1,1)=1,a(2,1)=2,a(1,2)=3,a(2,2)=4
```

(1, 1)	(1, 2)
1	3
(2, 1)	(2, 2)
2	4

## 动态数组

在Fortran中，可以使用 **allocate** 关键字来动态分配数组的内存空间。

示例：

```
program dynamic_array_example
    integer, allocatable :: dynamic_array(:)
    integer :: size, i

    ! 获取数组大小
    write(*, *) "Enter the size of the array:"
    read(*, *) size

    ! 动态分配数组内存空间
    allocate(dynamic_array(size))

    ! 初始化数组
    do i = 1, size
        dynamic_array(i) = i
    end do

    ! 输出数组元素
    write(*, *) "Array elements:"
    do i = 1, size
        write(*, *) dynamic_array(i)
    end do

    ! 释放数组内存空间
    deallocate(dynamic_array)

end program dynamic_array_example
```

在上述示例中，声明了一个可分配的整数数组 **dynamic\_array**，并没有为其分配内存空间。

然后，通过使用 **allocate** 关键字动态地为数组分配内存空间，根据用户输入的大小来确定数组的大小。

接下来，使用循环结构初始化数组的每个元素，将其设置为数组索引值。

然后，我们使用循环结构输出数组的每个元素。

最后，我们使用 **dealloc** 关键字释放数组的内存空间，以便在程序执行完毕后释放内存。

通过动态数组，可以在运行时根据需要动态地分配和管理数组的内存空间，使程序更加灵活和可扩展。

## 7-函数

### 子程序 subroutine

#### 概述

主程序在程序开始就自动会执行，而子程序则不会自动执行，它需要被 **call** 命令调用才会执行

子程序中包含可执行的程序代码，就类似于主程序的整体框架。

#### 声明在程序结构中的位置

- 声明的位置应该放在程序代码的可执行命令之前（这点和c++不一样）
- 在程序代码开始出现数值计算和输入输出命令时，就不能再声明变量了

主要结构如下：

```
program main
    implicit none

    声明变量

    可执行命令（赋值、计算、输入输出等）
end program main
```

#### 子程序框架

含有子程序的代码框架为：

```
program main
    ...
    call sub()
    ...
end program main

subroutine sub()
    ...
    return
end subroutine sub
```

或者将子程序嵌套在主程序中

```

program main
    ...
    call sub()
    ...
contains
subroutine sub()
    ...
    return
end subroutine sub
end program main

```

## 子程序特点

子程序可以在主程序中的任意位置被调用，在fortran90中，支持自己调用自己（递归）

子程序独立拥有属于自己的变量声明和行代码

在调用子程序时，可以同时传递一些变量数据过去，让它处理 ---- 传递参数

► 注意：fortran在传递参数时，使用的是**传地址调用**，也就是说，调用时传递出去的参数和子程序中接受的参数，会使用相同的内存地址来记录数据。

选择语言: Fortran (GFortran 9.2.0) ▾ 源代码  自动运行

命令行参数:

标准输入:

运行结果:

标准输出:

```

1 program ex0804
2     implicit none
3     integer :: a = 1
4     integer :: b = 2
5     integer :: ans = 0
6     call add(a, b, ans)
7     print "('ans=' , i1)", ans
8     stop
9 end
10
11 subroutine add(first, second, res)
12     implicit none
13     integer :: first, second, res
14     write(*, "(4x, i1)") first + second
15     res = first + second
16     return
17 end

```

会进行改变，因为传递的是地址

## 自定义函数 function

- 自定义函数和子程序很类似
  - 经过调用才能执行
  - 可以独立声明变量
  - 参数传递
- 两者不同：
  - 调用自定义函数前要先声明
  - 自定义函数执行后会返回一个数值
  - 声明时建议使用 **external**，表明其是一个可调用的函数。

示例：

```
program main
    implicit none
    real :: a = 1
    real :: b = 2
    real,external :: add_first !声明外部函数，必须说明类型，可以省略external
    real,external :: add_second

    print *,add_first(a,b) !调用外部函数，不需要使用call命令
    print *,add_second(a,b)
    stop
end program main
```

!第一种实现函数的方式：直接把函数名作为返回结果

```
function add_first(a,b)
    implicit none
    real :: a,b
    real :: add_first !必须在此处再次声明类型，充当返回值
    add_first = a + b
    return
end
```

!第二种实现函数的方式：再定义一个返回值

```
function add_second(a,b) result(c)
    implicit none
    real :: a,b
    real :: c
    c = a + b
    return
end
```

SHELL

```
3.00000000
3.00000000
```

注意：函数的返回值类型的声明可以写在函数的最开头，与function写在一起比如**real function add(a,b)**

```

!第一种实现函数的方式：直接把函数名作为返回结果
real function add_first(a,b)
    implicit none
    real :: a,b
    !real :: add_first !必须在此处再次声明类型，充当返回值
    add_first = a + b
    return
end

```

```

!第二种实现函数的方式：再定义一个返回值
real function add_second(a,b) result(c)
    implicit none
    real :: a,b
    c = a + b
    return
end

```

★★★：使用函数时，传递给函数的参数只读取它的值，不要去改变它的数据，因为函数的参数传递也是通过地址进行的，如果改变数据，会使的主程序中的参数数据发生变化。

如果函数很短，并且只会在主程序或者某一函数中使用，可以直接把函数定义写在主程序里

```

program main
    implicit none
    real :: a = 1
    real :: b
    real add
    add(a,b) = a+b
    write(*,*) add(a,3.0)
    stop
end

```

## 子程序 vs. 函数

相同点

- 经过调用才能执行
- 可以独立声明变量
- 需要传递参数

不同点

### 子程序

- 没有返回值
- 通过变量将结果传回。
- 输入输出更灵活，可以输出多个变量，方便实现数组的输出。

### 函数

- 有且只有一个返回值，通常为一个数值，也可以实现数组的输出，但不能输出多变量。
- 写法上格式更灵活。可以直接参与计算。

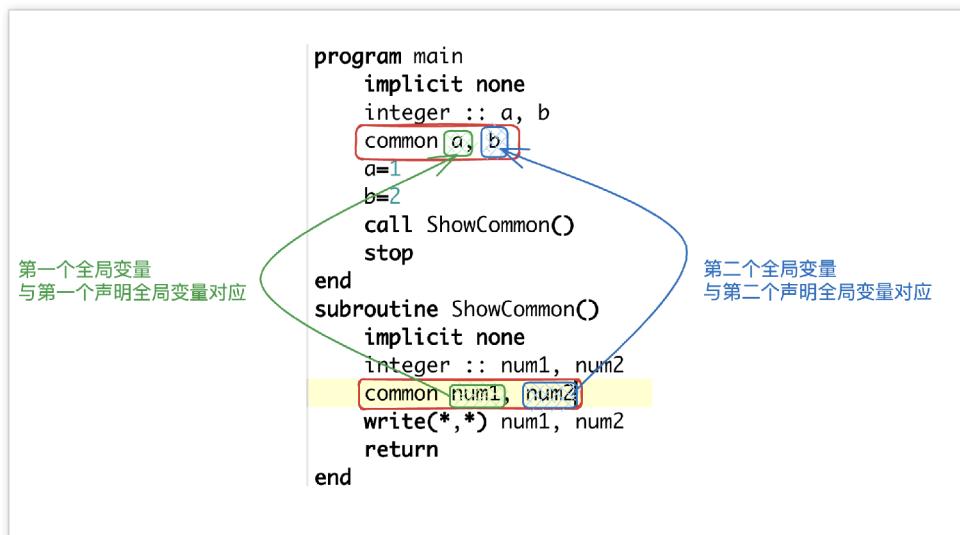
# 全局变量 common

## common的使用

common: 定义一块共享的内存空间，实现不同程序和函数之间传递参数

全局变量可以在程序中的任何一个部分被使用，采用地址对应的方法来实现数据的共享。

示例：



类似的，当全局变量较多时，可以把变量进行分类管理，将其放在彼此独立的common区间中

```
program ex0811
    implicit none
    integer :: a, b
    common /group1/ a
    common /group2/ b
    a=1
    b=2
    call ShowGroup1()
    call ShowGroup2()
    stop
end
subroutine ShowGroup1()
    implicit none
    integer :: num1
    common /group1/ num1
    write(*,*) num1
    return
end
subroutine ShowGroup2()
    implicit none
    integer :: num1
    common /group2/ num1
    write(*,*) num1
    return
end
```

## block data

由于common变量不能直接在子程序或主程序中使用data命令来设置其初始值，为了解决这一问题，规定可以将commom变量放到block data模块中进行data赋值。

block data：

- 类似于子程序，是一段独立的程序模块，拥有自己的变量声明（不能声明常量）
- 只能设置全局变量的值，不能有其它执行命令的出现。
- 不需要调用就可以自己执行，在主程序执行前就会生效

例如：

```
program main
    implicit none
    integer :: a,b
    common a,b
    integer :: c,d
    common /group1/ c,d
    integer :: e,f
    common /group2/ e,f
    write(*,"(6I4)") a,b,c,d,e,f
    stop
end
block data
    implicit none
    integer a,b
    common a,b
    data a,b /1,2/
    integer c,d
    common /group1/ c,d
    data c,d /3,4/
    integer e,f
    common /group2/ e,f
    data e,f /5,6/
end block data
```

执行结果如下：

```

program main
    implicit none
    integer :: a,b
    common a,b
    integer :: c,d
    common /group1/ c,d
    integer :: e,f
    common /group2/ e,f
    write(*,"(6I4)") a,b,c,d,e,f
    stop
end
block data
    implicit none
    integer a,b
    common a,b
    data a,b /1,2/
    integer c,d
    common /group1/ c,d
    data c,d /3,4/
    integer e,f
    common /group2/ e,f
    data e,f /5,6/
end block data

```

命令行参数:

标准输入:

运行结果:

**标准输出:**

1 2 3 4 5 6

## 函数中的变量

### 传递参数的注意事项

注意: fortran在传递参数时, 使用的是传地址调用, 也就是说, 调用时传递出去的参数和子程序中接受的参数, 会使用相同的内存地址来记录数据。

★★★: 使用函数时, 传递给函数的参数只读取它的值, 不要去改变它的数据, 因为函数的参数传递也是通过地址进行的, 如果改变数据, 会使的主程序中的参数数据发生变化。

## 数组参数

数组参数 == 字符串参数

在传递数组参数时, 实际上是传递数组元素当中的某个内存地址

类似于c++中的&引用

## 变量的生存周期

在声明中使用 **save** 可以增加变量的生存周期, 保留住所保存的数据

```

program main
    implicit none
    call sub()
    call sub()
    call sub()
    stop
end program
subroutine sub()
    implicit none
    integer :: count = 1
    save count ! 在子程序执行完后, count的值不会消失, 而是一直存在
    write(*,*) count
    count = count+1
    return
end

```

SHELL

```

1
2
3

```

## 传递参数

- 传递参数可以是数字, 字符等数据, 可以是函数名称或者子程序

### 传入函数

<pre> program main     implicit none     real, external :: func ! 声明func是一个用户自定义函数     real, intrinsic :: sin ! 声明sin是库函数     call ExecFunc(func)     call ExecFunc(sin)     stop end program  subroutine ExecFunc(f)     implicit none     real, external :: f     write(*,*) f(3.1415926/2)     return end  real function func(num)     implicit none     real :: num     func = num*2     return end function </pre>	<p>命令行参数:</p> <input type="text"/> <p>标准输入:</p> <input type="text"/> <p>运行结果:</p> <p><b>标准输出:</b></p> <p>3.14159250 1.00000000</p>
---	--

## 传入子程序

```
program main
    implicit none
    external sub1, sub2 ! 声明sub1和sub2是子程序的名称
    call sub(sub1) ! 把子程序sub1当参数传出去
    call sub(sub2) ! 把子程序sub2当参数传出去
    stop
end program

subroutine sub(sub_name)
    implicit none
    external sub_name ! 声明传入的是子程序
    call sub_name() ! 调用子程序
    return
end subroutine

subroutine sub1()
    implicit none
    write(*,*) "sub1"
end subroutine

subroutine sub2()
    implicit none
    write(*,*) "sub2"
end subroutine
```

调用子程序

根据传入的不同子程序名称，调用不同的子程序

命令行参数：  
标准输入：  
运行结果：  
**标准输出：**  
sub1  
sub2

注意：上面的声明external，不可以省略，因为这是要声明子程序的名称，从而当做参数去传递的。

## 特殊参数的使用方法

### 设置参数属性 (**intent**)

- **intent(in)** : 输入变量，不可以改变其值
- **intent(out)** : 输出变量
- **intent(inout)** : 既可以输入也可以输出

不指定参数属性不会影响程序执行的结果，但是可以避免编写程序的错误

### 函数的使用接口 (**interface**)

interface 是一段程序模块，用来描述调用函数的参数类型及返回值类型等的 使用接口

类似于使用手册，这样就不用直接定义了

```

interface
    function func_name
        ! 里面只能说明参数或返回值类型
        implicit none
        real....
        integer....
    end [function [func_name]]

    subroutine sub_name
    implicit none
    integer....
    end [subroutine [sub_name]] ←μ(-ŔÈTě
end interface

```

例如：

<pre> program main     implicit none     interface         function random10(lbnd, ubnd)             implicit none             real :: lbnd, ubnd             real :: random10(10)         end function     end interface     real :: a(10)     call random_seed() ! 使用随机函数前调用     a = random10(1.0, 10.0)     write(*, "(10F6.2)") a end  function random10(lbnd, ubnd)     implicit none     real :: lbnd, ubnd     real :: db     real :: random10(10)     real t     integer i     db = ubnd - lbnd     do i=1,10         call random_number(t)         random10(i) = lbnd + db * t     end do     return end </pre>	命令行参数： <input type="text"/>  标准输入： <input type="text"/>  运行结果： <b>标准输出：</b> 6.64 7.05 4.51 1.82 9.52 2.92 6.88 9.18 3.52 1.57
--	---

## 不定个数的参数传递

可以用 **optional** 命令来表示某些参数是可以省略的

要调用这类不定数目参数的函数时，一定要先声明出函数的 **interface**

函数 **present** 可以检查一个参数是否传递过来，返回值是逻辑型变量

```

program main
    implicit none
    interface
        subroutine sub(a,b) ! 定义子程序的接口，从而使用optional
            implicit none
            integer :: a
            integer, optional :: b
        end subroutine sub
    end interface
    call sub(1)
    call sub(2,3)
stop
end program main

subroutine sub(a,b)
    implicit none
    integer :: a
    integer, optional :: b
    write(*,"('是否传入b参数: ',I1)") present(b)
    if ( present(b) ) then
        write(*,"('a=',I1,' b=',I1)") a, b
    else
        write(*,"('a=',I1,' b=unknown')") a
    end if
    return
end subroutine sub

```

命令行参数:

标准输入:

运行结果:

**标准输出:**

是否传入b参数: F  
a=1 b=unknown  
是否传入b参数: T  
a=2 b=3

## 改变参数传递位置的方法

前提：一定要声明 **interface**

可以根据变量名称来传递参数

例如：

```

call sub(b=2,c=3,a=1)

subroutine sub(a,b,c)
    ...
end

```

## 封装函数和参数

### **contains**

**contains**所包含的函数只能被使用**contains**的主程序或者函数调用，之外的函数无法调用

```

program main
    implicit none
    call sub1()
    call sub2()
contains
    subroutine sub2()
        implicit none
        print*, 'This is sub2'
        call sub1()
    end subroutine sub2
end program

subroutine sub1()
    implicit none
    print*, 'This is sub1'
end subroutine sub1

```

可以被使用**contains**的主程序进行调用

命令行参数:

标准输入:

运行结果:

**标准输出:**

```

This is sub1
This is sub2
This is sub1

```

```

program main
    implicit none
    call sub1()
    call sub2()
contains
    subroutine sub2()
        implicit none
        print*, 'This is sub2'
    end subroutine sub2
end program

subroutine sub1()
    implicit none
    print*, 'This is sub1'
    call sub2()
end subroutine sub1

```

不可以被之外的函数进行调用

命令行参数:

标准输入:

运行结果:

**编译错误:**

```

/usr/bin/ld: /tmp/ccQFr7Fp.o: in function `sub1__':
main.f90:(.text+0x71): undefined reference to `sub2_'
collect2: error: ld returned 1 exit status

```

## module

### 概述

module 用来封装变量和函数，要配合contains来封装函数

module 可以被任何主程序或函数进行调用，命令为: **use 模块名**

在use声明后，其中的函数就可以被使用。

```

module test
    contains
        subroutine sub1()
            implicit none
            print*, 'This is sub1'
        end subroutine sub1
end module test

```

```

program main
    use test
    implicit none
    call sub1()
    call sub2()
contains
    subroutine sub2()
        implicit none
        print*, 'This is sub2'
    end subroutine sub2
end program

```

注意：：如果是在同一个.f90文件中，定义主程序和module，一定要把**module**定义在**main**之前，程序才能够正常去运行。

## public和private

module里面的数据和函数，可以通过public或private命令，设置成公开或私密

没有特殊说明时，默认函数或数据都是public的

```
module bank
    implicit none
    private money ! 变量声明为私有
    public LoadMoney, SaveMoney, Report ! 函数声明为公有
    integer :: money = 1000000
contains
    subroutine LoadMoney(num)
        implicit none
        integer :: num
        money=money-num
        return
    end subroutine
    subroutine SaveMoney(num)
        implicit none
        integer :: num
        money=money+num
        return
    end subroutine
    subroutine Report()
        implicit none
        write (*,"('银行目前库存为：',I9,'元')") money
    end subroutine
end module

program main
    use bank
    implicit none
    call LoadMoney(100)
    call SaveMoney(1000)
    call Report()
    stop
end
```

命令行参数：  
  
  
标准输入：  
  
  
运行结果：  
**标准输出：**  
银行目前库存为： 1000900元

## 注意事项

- Module不是必须写在最前面，但必须写在use这个module的程序单元的前面
- 如果所有代码写在同一个源代码文件中，则module应该写在前面
- 如果代码写在多个源代码文件中，确保module要先于use这个module的程序单元进行编译
- Module中不能直接书写执行语句（比如read、write），所有执行语句都要写在contains下面的子程序或自定义函数中

## 递归函数

递归函数每次被调用执行时，函数中所声明的局部变量（不是传递的参数，没有save的变量）都会使用不同的内存地址，也就是说，每次调用时都是独立存在的。

```
recursive integer function fact(n) result(ans)
```

- 开头以recursive来表示可以递归，被自己调用
- 要使用result来返回参数，传递结果
- 递归调用时，要明确递归结束的条件

- 不使用recursive还可以使用间接递归，来实现递归
  - 即在函数中先去调用别的函数，再经过那个函数来调用自己

比如：通过递归来计算n的阶乘

```

program main
  implicit none
  integer :: n
  integer, external :: fact
  read(*,*) n
  write(*, "(I2,'! = ',I2)") n, fact(n)
  stop
end
recursive integer function fact(n) result(ans) ! recursive 表示可以递归
  implicit none
  integer, intent(in) :: n
  if ( n < 0 ) then
    ans = -1
    return
  else if ( n <= 1 ) then
    ans = 1
    return
  end if
  ans = n * fact(n-1)
  return
end

```

## 使用多个文件

将程序代码分散到不同文件中

优点如下：

1. 独立文件中的函数，方便多人协同工作，方便移植，可以把部分文件给其它程序使用
2. 可以加快编译速度，修改其中一个文件时，编译器只需要重新编译这个文件后再链接即可，不需要编译全部文件

## include

include 用来在程序代码中插入另一个文件的内容

include 命令可以写在任何地方，通常应用在声明变量处

```
! file : main.f90
program main
    implicit none
    include 'test.inc' !插入test.inc的内容
    a=1
    b=2
    call sub()
    stop
end
subroutine sub()
    implicit none
    include 'test.inc' !插入test.inc的内容
    write(*,*) a,b
    return
end
```

```
! file : test.inc
integer a,b
common a,b
```

## 8-文件

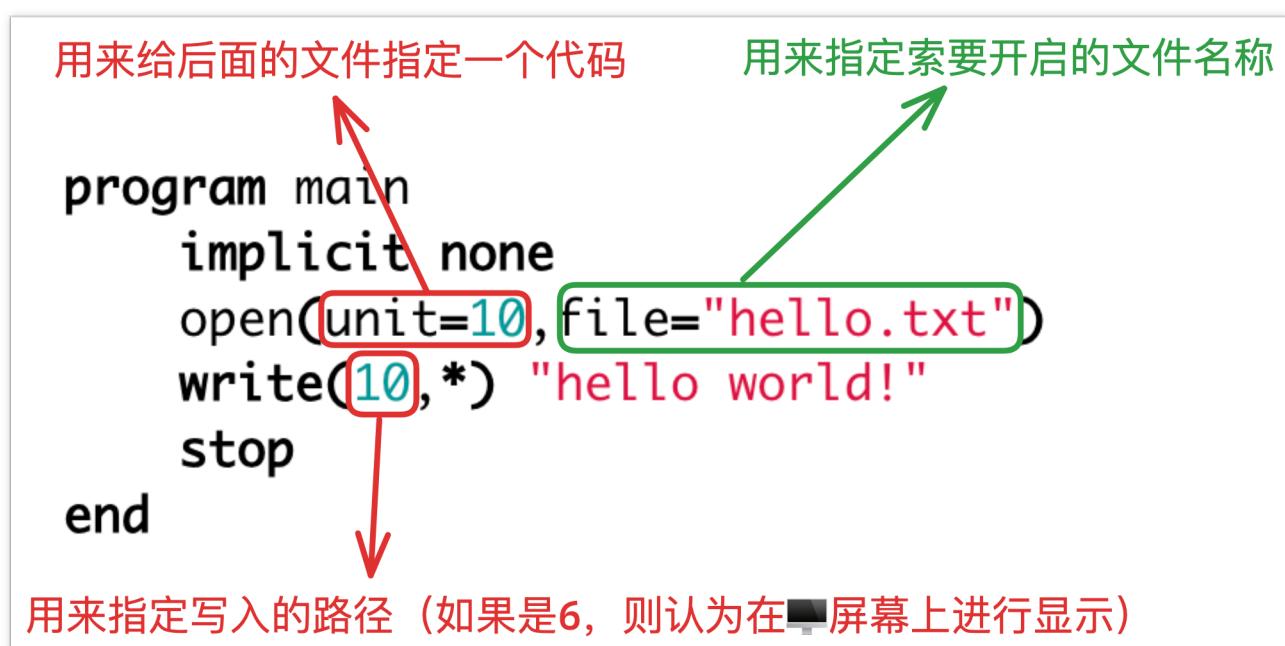
### 文件的操作

#### 概述

- 打开文件 open
- 读写文件 read write
- 关闭文件 close
- 查询文件 inquire
- 其它命令

#### open的使用

open : 打开文件 (关联一个文件与通道)



open的参数

```
open( [unit=]number [,file='filename'] [,form='...'] [,status='...']
      [,access='...']
      [,recl=length] [,err=label] [,iostat=iostat] [,blank='...'] [,position='...']
      [,action=action] [,pad='...'] [,delim='...'])
```

- `[unit=]number`: 指定文件的逻辑单元号 (unit number)。
  - 其中, `unit=` 可以进行省略
  - `number` 必须为正整数
  - `number` 可以使用变量或常量进行赋值
  - `number` 应该尽量避免使用5和6 (5默认是键盘输入, 6默认是屏幕输出)

- `[,file='filename']`: 指定文件的名称。
  - 文件名要符合操作系统的规定
  - 避免使用中文字符
- `[,form='...']`: 指定文件的格式。
  - `formatted` 表示文件使用文本文件格式来保存
  - `unformatted` 表示文件使用二进制文件格式来保存
- `[,status='...']`: 指定文件的状态。
  - `new`: 新建文件
  - `old`: 打开已有文件
  - `replace`: 替换已有文件或新建文件
  - `scratch`: 打开一个暂存盘, 不需要指定文件名, 程序结束后自动删除
  - `unknown`: 由编译器自定义, 通常为新建文件或打开已有文件 (如果有写入, 会覆盖原有的内容)
- `[,access='...']`: 指定文件的访问方式
  - `'sequential'` 表示顺序访问 **默认值**
  - `'direct'` 表示直接访问 (可指定任意位置)
- `[,recl=length]`: 顺序读取文件时, 设置一次可以读写多大容量的数据
  - 直接读取文件时, 设置文件中每个模块单元的分区长度
  - 单位
    - 在文本格式下, 为一个字符 (1B)
    - 在二进制格式下, 由编译器决定 (1B或4B)
- `[,err=label]`: 设置当文件打开发生错误时, 程序会跳到代码为label的行继续执行。
  - 1-99999范围内的整数
- `[,iostat=var]`: 返回文件操作的状态码。
  - 返回一个值给变量var, 用来说明文件打卡的状态
  - `var>0`: 表示文件读取操作发生错误
  - `var=0`: 表示读取操作正常
  - `var<0`: 表示文件终了
- `[,blank='...']`: 设置文件输入数字时, 当所设置的格式字段中有空格存在时, 所代表的意义
  - `null` : 表示空格代表没有东西 (**默认**)
  - `zero` : 空格自动以0代替
- `[,position='...']`: 指定文件打开时的读写初始位置
  - `rewind` 表示将文件指针重置到开头
  - `as is` 表示不特别指定, 通常在文件开头 (**默认**)
  - `append` 表示移到文件结尾 (多用于续写文件)
- `[,action='...']`: 设置打开文件的读写权限, 避免误写
  - `readwrite` : 表示可读取和写入 (**默认**)
  - `read`: 表示只能读取
  - `write`: 表示只能写入
- `[,pad='...']`: 确定格式化输入时, 前面不足的字段是否要自动以空格进行填充
  - `yes`
  - `no`

- `[,delim='...']`: 指定字符串之间的分隔符。
  - none : 只输出字符串的内容 (默认)
  - quote: 输出字符串会在前后加上双引号
  - apostrophe: 输出字符串时会在前后加上单引号

## read 和 write的使用

```
read/write ([unit=]number [,fmt=format] [,rec=record] [,iostat=stat]
           [,err=errlabel] [,end=endlabel] [,advance=advance]
           [,size=size])
```

- `([unit=]number)`: 指定文件的逻辑单元号 (unit number)。
- `[,fmt=format]`: 指定格式控制。
- `[,rec=record]`: 在直接读取文件中，设置读写的文件模块位置。
- `[,iostat=stat]`: 返回读取或写入操作的状态码。
  - `[,iostat=var]`: 返回文件操作的状态码。
    - 返回一个值给变量var, 用来说明文件打卡的状态
    - var>0: 表示文件读取操作发生错误
    - var=0: 表示读取操作正常
    - var<0: 表示文件终了
- `[,err=errlabel]`: 表示在读写过程中发生错误时，会跳到代码为errlabel的行继续执行。
- `[,end=endlabel]`: 表示读写到文件末尾时，跳到代码为endlable的行继续执行。
- `[,advance=advance]`
  - 设置在文本格式下的顺序文件中，每一次的读写命令完成后，读写位置是否自动向下移动一行。
    - yes (默认)
    - no
- `[,size=count]`:
  - 当advance='no'时，返回此次输入/输出的字符数目给整型变量count

## close的使用

```
close ([unit=] number [,status=string] [,err=errlabel] [,iostat=iostat])
```

- status : 决定文件关闭后的操作
  - keep 或 save : 表示在文件关闭后，保留该文件 (默认)
  - delete : 表示该文件关闭后，删除该文件

## 查询文件的状态

注意 📺：可以采用通道号或文件名来查询文件，但是，以上两种方式只能选择一种进行使用

```
inquire ([unit=number] [, file='filename'] [, iostat=stat] [, err=label]
[, exist=exist]
[, opened=opened] [, sequential=sequential] [, direct=direct] [, form=form]
[, formatted=formatted]
[, unformatted=unformatted]      [, recl=recl])
```

- **exist**: 检查文件是否存在
  - 返回一个布尔值给逻辑变量 **exist**
  - 真值, 表示文件存在
  - 假值, 表示文件不存在
- **opened**: 检查文件是否已经使用open命令打开
- **number**: 由文件名来查询该文件的通道号
- **named**: 查询文件是否取了名字
- **access**: 检查文件的读取格式, 返回一个字符串给字符型变量access
  - '**sequential**' 表示顺序访问 默认值
  - '**direct**' 表示直接访问 (可指定任意位置)
  - '**undefined**' 表示没有定义
    - **[,access='...']**: 指定文件的访问方式
      - '**sequential**' 表示顺序访问 默认值
      - '**direct**' 表示直接访问 (可指定任意位置)
- **sequential**: 查看文件是否使用顺序格式
- **direct**: 查看文件是否使用直接格式
- **form** 查看文件的保存方法
- **unformatted**: 查看文件是否为二进制文件
- **formatted** : 查看文件是否为文本文件

## 其它文件运行命令

- **backspace(unit=number,err=label,iostat=stat)**
  - 把文件的读写位置退回一步
- **endfile(unit=number,err=errlabel,iostat=stat)**
  - 把文件的读写位置设为文件末尾
- **rewind(unit=number,err=label,iostat=stat)**
  - 把文件的读写位置倒回文件开头

## 顺序文件的操作

顺序文件在读写时, 不能任意赋值到文件的某个位置读写数据, 只能从头开始一步步向下进行。

改变文件读写位置时, 只能一步步地进退, 或是直接移回文件开头。

例如: 文本文件

```

program main
    implicit none
    character (len=79)      ::      filename
    character (len=79)      ::      buffer
    integer,parameter :: fileid = 10
    integer ::      status = 0
    logical alive

    write (*,*)      "Filename:"
    read (*,"(A79)")filename
    inquire (file=filename, exist=alive)
    if (alive)      then
        open (unit=fileid,      file=filename, access="sequential",
status="old")
        do while (.true.)
            read (unit=fileid,fmt="(A79)",iostat=status) buffer
            if (status/=0)  exit !没有数据就跳出循环
            write (*,"(A79)")buffer
        end do
    else
        write (*,*)      trim(filename),"doesn't exist."
    end if

    stop
end

```

```

program main
    implicit none
    character (len=79)  ::  filename
    character (len=79)  ::  buffer
    integer,parameter :: fileid = 10
    integer ::  status = 0
    logical alive
    write (*,*) "Filename:"
    read (*,"(A79)")filename
    inquire (file=filename, exist=alive)
    if (alive)  then
        open (unit=fileid,file=filename,access="sequential",status="old")
        do while (.true.)
            read (unit=fileid,fmt="(A79)",iostat=status) buffer
            if (status/=0)  exit !没有数据就跳出循环
            write (*,"(A79)")buffer
        end do
    else
        write (*,*)  trim(filename),"doesn't exist."
    end if
    stop
end

```

打开文件后，一行一行的读取文件内容  
 循环一次，读入一行，并把这一行文本写到屏幕  
**status=0**表示读写正常  
**status>0**表示读写失败  
**status<0**表示读完毕

命令行参数:

标准输入:

 main.f90  

运行结果:

**标准输出:**

```

Filename:
program main
    implicit none
    character (len=79)      ::      filename
    character (len=79)      ::      buffer
    integer,parameter :: fileid = 10
    integer ::      status = 0
    logical alive
    write (*,*)      "Filename:"
    read (*,"(A79)")filename
    inquire (file=filename, exist=alive)
    if (alive)      then
        open (unit=fileid,file=filename,access="sequential",status="old")
        do while (.true.)
            read (unit=fileid,fmt="(A79)",iostat=status) buffer
            if (status/=0)  exit !没有数据就跳出循环
            write (*,"(A79)")buffer
        end do
    else
        write (*,*)      trim(filename),"doesn't exist."
    end if
end

```

## 直接访问文件的操作

直接访问文件，是把文件的空间、内容事先分区成好几个同样大小的模块，这些模块会自动按顺序编号。

读写文件是，要先赋值文件读写位置在第几个模块，再来进行读写的工作。

直接访问文件可以任意到文件的任一个模块来读写。

```

program main
    implicit none
    integer,parameter :: fileid = 10
    character (len=20) :: filename = "list.txt"
    integer player
    real hit
    integer error
    logical alive
    inquire (file=filename, exist=alive)
    if (.not.alive) then
        write(*,*) trim (filename), "doesn't exist."
        stop
    end if
    open
    (unit=fileid, file=filename, access="direct", form="formatted", recl=6, status="old")
    do while (.true.)
        write(*, "('查询第几棒?')")
        read (*,*) player
        read (fileid, fmt="(F4.2)", rec=player, IOSTAT=error)
hit
        if (error/=0) exit
        write(*, "('打击率: F4.2)") hit
    end do
    close (fileid)
    stop
end program

```

## 二进制文件的读取

```

program main
    implicit none
    integer,parameter ::fileid = 10
    character (len=20) :: filename = "list.bin"
    integer player
    real :: hit(9) = (/3.2,2.8,3.3,3.2,2.9,2.7,2.2,2.3,1.9/)
    open (unit=fileid,file=filename,access="direct",recl=4, status="replace")
do player=1,9
    write (fileid,rec=player) hit(player)
end do
close (fileid)
stop
end program main

```