

# 并行计算上机实践课讲义

刘浩洋\* 罗昊†

2022 年 3 月 22 日

## 1 概述

### 1.1 集群和个人电脑的区别

集群，即我们平时所说的“服务器”，将会是我们进行科学计算的第一劳动力。它有着和个人计算机 (PC) 相似的用法，但在某些关键方面和个人计算机有本质的不同。这些不同主要体现在：

- 使用服务器 Linux 系统，且很少用 ubuntu，系统各类软件版本较低，更新较慢，追求运行的稳定。
- 默认情况下图形界面为关闭状态，使用命令行跟系统进行交互，需要用户对命令行模式有一定的熟练度。用户虽然可以使用图形界面方式连接，例如 X11 转发或者 VNC 等，但由于其延迟较高，使用体验不好，因此还是推荐各位用户在命令行模式下操作。
- 多用户同时在线，用户之间会相互影响，会共享部分的计算资源。用户之间也可以相互通信。
- 严格的权限管理，普通用户无法使用 `sudo` 进行权限提升，因此不能执行只有管理员才能执行的操作，例如挂载，安装驱动，使用包管理器安装软件等。但并不意味着用户无法安装自己需要的软件。
- 用户不能在集群操作系统上开设自己的服务。既然服务器可以对外访问，那么用户可以利用这个特点在集群上开设自己的服务吗？**想得美**。集群防火墙会使得你的服务无法从外界直接访问，因此就算服务跑起来外面人也用不了。
- 登录和计算分离。即登录所在的机器通常不是实际运行计算程序的机器。这是集群初学者最容易产生的误区，认为超算就是一整个资源超级多的服务器，这是不对的。实际上，和用户交互的机器被我们称为**登录节点**，它只是众多机器中很少的一类。实际运行程序的机器用户通常情况下是看不见的。

---

\*北京国际数学研究中心 (BICMR)

†北京大学数学科学学院

## 1.2 相关概念

- 节点 (node): 指一台完整的主机, 即通常我们说的“服务器”。根据它们的作用可分为如下几类。
  - 登录节点: 供用户进行登录的机器, 可以从外界访问, 用户可在上面进行一些常规操作, 例如复制文件, 编译小型软件等。
  - 计算节点: 主要负责服务器的所有计算任务, 原则上只有计算节点才能运行用户的计算程序, 其他节点不可以。用户可以获得暂时的计算节点访问权限。根据计算的架构还可细分为 CPU 节点, GPU 节点, 胖节点等。
  - 管理节点: 主要负责集群各种系统的管理, 例如用户管理, 调度系统的控制器, 集群全局配置。只有管理员才有权限访问。
  - 存储节点: 主要负责集群的全局 IO, 通常情况下集群所有节点的存储设备都放在 IO 节点上, 然后由 IO 节点通过网络共享的方式挂载到其它节点。在大型超算集群中, 磁盘设备吞吐量特别大, 因此需要专门用 IO 节点进行负载均衡, 提高读写效率。
- 集群 (cluster): 一组节点经过局域网互联形成的计算机群。节点之间相互通信具有低延迟高速率的特点。一般的集群都至少有两种以上的网络互联: 一种是管理网, 速度较慢 (1000Mbps), 用于系统管理 (查询网络用户, 同步配置文件等); 另一种是计算网, 速度较快 (万兆或 Infiniband), 主要用户磁盘 IO 和分布式并程序序的通信 (MPI)。我们这节课学习的内容实际就是在编写支持**分布式计算**的程序, 这也就意味着分布式并行实际上并不是一个黑箱子, 不是说我们想让它并行它就会自己并行了, 需要我们一定的编程基础。
- 主机名 (hostname): 节点的名字, 用于区分是哪一台主机。一般在命令提示符 (prompt) 中用户名的后面。在集群中由于有很多个节点, **用户操作的时候一定要看清自己**在哪。

目前我们上课用的数院集群都有哪些节点呢? 这可以通过各个节点的 `/etc/hosts` 文件进行查询。

- 管理节点 & 登录节点: `mu01`;
- 计算节点 (CPU): `cu[01-10]`;
- 计算节点 (GPU): `gpu[01-04]`<sup>1</sup>;
- 存储节点: `iml,io[01-02]`。

## 2 登录

一般情况下, 集群登录对外的入口是 SSH 服务。虽然某些集群也可用网页进行操作, 但是网页提供的功能很有限, 且操作效率不高。用户需要安装 **SSH 客户端**来连接到集群的登录节点, 登录成功后将会获得远程操作的大部分权限。以下就讲述如何连接集群。

---

<sup>1</sup>其中 `gpu01` 目前为不对外开放

## 2.1 Linux/MacOS

使用 Linux 和 MacOS 的用户可以直接使用系统的 `ssh` 命令进行登录。需要注意的是 `ssh` 命令实际上是这两个系统 SSH 客户端的一部分。由于集群的操作模式主要为命令行，因此在自己的笔记本上也必须要有能执行命令行的工具。这两个系统的命令行工具均为**终端 (Terminal)**，相应的打开方式为：

- Linux(Ubuntu)，打开方式为 `Ctrl+Alt+T`
- MacOS，打开方式为访达 (Finder)→ 应用程序 (Applications)→ 实用工具 (Utilities)→ 终端 (Terminal)

成功打开终端之后，我们就可以输入命令了。以后有关集群的操作都要在这里进行。

使用 `ssh` 命令的基本语法为 (在自己的笔记本上执行)

```
1 ssh username@server
```

其中 `username` 是自己在集群中的用户名，`server` 是集群的 IP 地址或 集群的域名。例如在我们自己的集群，登录用户 `fdpc_1234567890` 可以使用命令：

```
1 ssh fdpc_1234567890@mu2.davidandjack.cn
```

第一次登录系统会警告你是否将远程主机添加到 `known hosts` 列表，输入 `yes` 确认。

之后就是输入密码的区域了，输入自己的密码，然后按 `Enter` 键继续。**注意：输入密码的时候屏幕上不会提示任何东西，好像卡住了一样。这实际是集群的安全保护措施，即输入密码时看不到反馈。用户只需要正常输入即可。如果输入错误，建议长时间按住 `Backspace` 删掉再重新输入。**

如果是第一次登录，服务器会要求你改掉初始的密码，此时按照提示进行操作即可。注意输入密码的顺序为：当前密码-> 新密码-> 重复新密码。即第一个输入的密码是当前的旧密码。

```
1 # 这里要求强制更改初始密码
2 You are required to change your password immediately (root enforced
   )
3 Changing password for user.
4 (current) UNIX password:          # 输入当前密码
5 New password:                    # 输入新密码
6 Retype new password:             # 重复新密码
7
8 (登录欢迎信息...)
9
10 [user@mu01 ~]$                  # 成功登录
```

到这里就已经成功登录了，我们会看到提示符变成了 `[user@mu01 ~]$` 字样，提示符的含义我们稍后再解释。

## 2.2 Windows

### 2.2.1 使用 PowerShell 或 Windows Terminal 登录

Windows 10 1903 以上版本自带 OpenSSH 客户端, 可直接打开 PowerShell 或者 Windows Terminal (Windows 11 自带 Windows Terminal, Windows 10 用户可通过微软商店或 github 下载安装), 然后输入 ssh 命令连接服务器。可参考 2.1 中的操作说明。

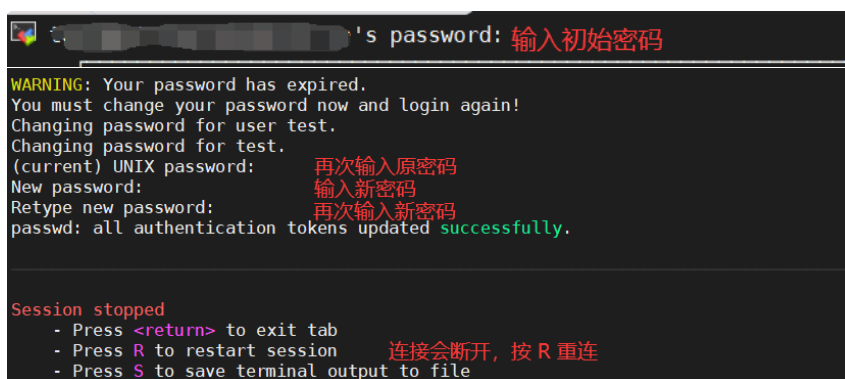
### 2.2.2 使用 MobaXterm 登录

Windows 用户推荐使用 MobaXterm<sup>2</sup> 登录服务器。本指南使用的版本为 MobaXterm Pro 20.1, 社区版的操作类似。

下载完成后, 点击左上角 Session -> SSH, 输入管理员提供的服务器地址, 用户名。端口号选择默认的 22。



第一次登录按照系统提示输入初始密码, 并即刻修改新密码。同样的, 输入密码时终端会没有反应, 这是服务器的安全保护机制, 详见 Linux/macOS 登录部分。



新密码设置后, MobaXterm 会询问是否保存密码, 选择是或否均可。登录成功后, 会看到和 Linux 等相似的命令行提示符。

<sup>2</sup>下载地址为: <https://mobaxterm.mobatek.net/download-home-edition.html>

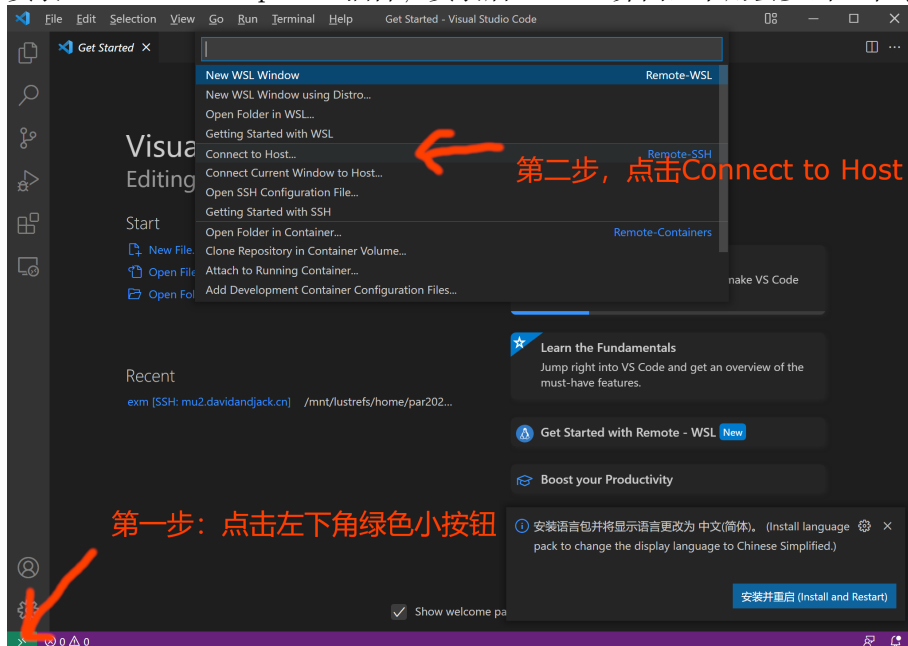
## 2.3 使用 vscode 登录

Visual Studio Code 是微软开发的代码编辑器，支持 Windows, Linux, MacOS 系统。vscode 提供了 Remote Development 插件，可以直接连接到服务器，编辑服务器上的文件。

由于服务器首次登录要求强制修改密码，因此首次登录建议使用 2.1 或 2.2.1 或 2.2.2 中的方式，完成修改密码后再使用 vscode 登录。

使用 vscode 配置远程服务器的步骤如下：

1. 安装 Remote Development 插件，安装后 vscode 界面左下角会多出一个绿色图标。



2. 点击左下角的绿色图标,然后在弹出的菜单中点击”Connect to Host...”,然后再点击”+Add New SSH Host”,
  3. 在弹出的输入框中输入命令
- ```
1 ssh username@mu2.davidandjack.cn # username 为你的用户名
```
4. 然后选择要使用的 ssh config 文件，一般选择第一个即可；
  5. 如此便完成了服务器的添加。

配置完成后，可以使用如下步骤连接服务器：

1. 点击左下角的绿色图标，然后在弹出的菜单中点击”Connect to Host...” 然后再点击已经添加的服务器名称，例如”mu2.davidandjack.cn”，会弹出新窗口连接服务器
2. 然后按照 vscode 的要求选择服务器操作系统的类型 (Linux)，然后按照提示输入密码
3. 如果是第一次登录或 vscode 更新后第一次登录，vscode 会在服务器上下载安装 vscode-server。此时如果服务器没有联网则会下载失败，可以在服务器上用 connect 命令连接北大网关。

4. 等待 vscode-server 安装完成后便连接成功，此时可以通过菜单栏打开文件或文件夹，如果打开文件夹，则 vscode 窗口会重新加载，此时可能需要重新输入密码
5. 也可以通过菜单栏新建终端，打开服务器上的终端，直接运行命令或提交 slurm 任务。

## 2.4 WPN

WPN (Web VPN) 是北京大学计算中心于 2019 年 11 月左右上线的校内资源访问平台，主要解决的是研究生从校外连接校内端口 (SSH 服务, RDP 服务) 的需求和对外信息服务需要申请备案的矛盾。此事在 BBS 上也曾经引起了大量讨论，详见[这个帖子](#)。使用 WPN 的好处是不再需要 VPN 更改全局网络设置，因此系统的其他网络访问使用的仍然是本地网络。这一方式的坏处是：**客户端内嵌在网页上，这是一个很糟糕的实现。**

要使用 WPN 方式访问集群可遵从如下步骤：

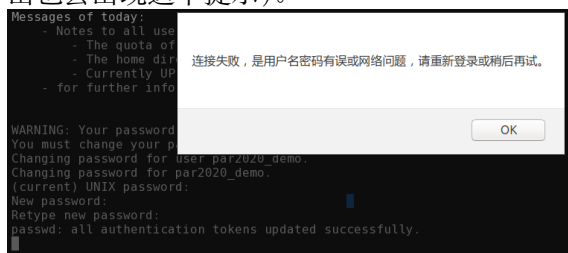
1. 打开网页 <https://wpn.pku.edu.cn>，使用 IAAA 账号登录。
2. 在上方的跳转栏选择 ssh 协议，输入服务器地址 162.105.98.22，然后点击**立即跳转**。注意：1) 浏览器可能会拦截弹出窗口，请选择放行；2) 目前 WPN 方式使用域名 mu2.davidandjack.cn 好像有问题，直接使用 IP 地址是可以的。



3. 在弹出的页面窗口中输入自己的用户名和密码。



4. 首次登录会需要修改密码。请按照提示输入：原密码，新密码，确认新密码。修改成功后会提示 “all authentication tokens updated successfully”，WPN 给出提示 “连接失败，是用户名密码有误或网络有问题”（可不予理会，实际含义是服务器断开了连接，正常登出也会出现这个提示）。



5. 重复上面的步骤，使用新密码登录。

## 3 Linux 基础

### 3.1 Shell 与提示符

能够成功登录集群后，我们想问的第一个问题就是：如何使用它？如何和 Linux 系统进行交互？建立用户和 Linux 系统的桥梁就是 Shell。用户通过 Shell 向系统发送指令来控制系统的硬件和软件如何运行。集群使用的 Shell 是命令行类型的，用户需要通过键盘向系统发送命令，系统将命令的结果以字符的形式返回到屏幕上。

学会如何使用命令行 Shell（以下简称**命令行**）对于 Linux 用户来说是重要的。其原因主要有：

1. 由于带宽，延迟等硬件因素，在远程服务器使用图形界面通常来说体验极差。同时图形界面比较消耗资源，当大量用户同时登录时对服务器也是不小的压力。
2. 命令行通常来说更加高效，可以很轻松完成大量重复的任务。
3. 使用命令行只需要键盘，熟练之后你可以完全不需要鼠标就能操作机器。在写代码——调试的时候不用频繁地在键盘和鼠标之间切换。

学习命令行需要有个出发点，而这个出发点就是我们的提示符 (Prompt)，就是窗口中最下方闪着光标等待我们输入的区域。

提示符只有短短一行，十几个字符，但是它却提供给我们非常有用的信息。简单来说，它把我们当前所处的状态简要描述清楚了。

```
1 [user @ mu01 ~]$ ls -al
2 ^~~~~ ^~~~~~ ^~~~~~
3 我是谁 我在哪 我在干什么
```

提示符主要包含三个区域：

- 用户名区域：处于符号 @ 之前，表示你的身份。
- 主机和路径区域，处于符号 @ 后，符号 \$ 前，表示你所处的主机名和主机中的路径。例如在上面例子中主机名为 mu01，所处路径为 “~” 即个人的家目录 (HOME)。
- 命令区域，你要执行的命令，出现在 \$ 符号的后面。这也正是用户不断要给计算机发出指令的地方，只不过我们从前习惯了用 Windows，在图形界面移动鼠标点点按钮，而在集群上变成了用键盘来敲命令。

命令行的信息之所以关键，是因为我们从前只有一台计算机，“我是谁，我在哪”这样的问题的答案是清楚的：我就是我笔记本上的用户。但是现在，我们需要远程连到集群的各个节点上，一个集群可能有上千个节点，这个时候“我是谁，我在哪”便成了一个相当重要的问题，如果搞不清这一点，很多操作都无法顺利进行。因此集群的初学者需要额外注意这方面的问题。

在对命令行操作时，我们大多数情况会使用到键盘。除了常规字符的输入以外，很多组合键也非常常用。强烈建议初学者学习并记住。



- **TAB**: 命令补全, Shell 会根据可用的命令或文件路径自动将其补全。如果候选不唯一时, 再按一下 TAB 会显示所有的可能补全方式。
- **Ctrl+C**: 向当前正在程序发送中断信号, 迫使程序停止 (即所说的“杀程序”)。但杀程序这个说法并不严谨, 因为发送的并不是杀死程序 (SIGKILL) 或终止程序 (SIGTERM) 的信号, 而是中断 (SIGINT)。
- **Ctrl+Z**: 挂起当前程序到后台, 类似于 Windows 中的“最小化”?
- **Ctrl+W**: 删除光标前一个单词。还在不停按 Backspace 吗? 试试 Ctrl+W 吧。
- **Ctrl+U**: 删除整行。实际上是删除行首到光标处的所有内容。
- **Ctrl+S**: 冻结窗口。注意: 这不是保存的命令。当屏幕输出过快时, 用户可冻结窗口来查看瞬时的输出。
- **Ctrl+Q**: 取消冻结窗口。
- **Ctrl+D**: 向命令行输入终止符 EOF, 有时候我们也用 ^D 来表示终止符。
- **Ctrl+A**: 将光标移动到行首。
- **Ctrl+E**: 将光标移动到行尾。
- **Alt+F**: 将光标向右移动一个单词。
- **Alt+B**: 将光标向左移动一个单词。
- 方向键的上和下: 查看命令历史的前一个/后一个指令。
- **Ctrl+R**: 逆向查找历史执行过的命令。非常常用。**思考: 如何进行顺向查找?**

## 3.2 基本命令

在这一节我们介绍 Linux 的一些最基本的操作命令。这些命令都可以直接输入到命令行提示符后面。我们可能注意到, Linux 的设计者似乎总是在试图设计一种通用的规则, 因此在它上面运行的命令基本上也分为以下几个部分

```
1 command [OPTIONS] [ARGUMENT]
```

即一个命令可以包含“命令名字 (command)”, 命令选项 (OPTIONS), 命令参数 (ARGUMENT)。命令的名字很好理解, 参数可以理解为命令操作的直接对象, 而选项可以理解为决定了命令执行的方式。不同选项之间可以相互组合, 而参数基本上是固定的。

Linux 可以执行的命令非常之多, 在我们进入到 Shell 中时, 就有一些命令已经预先写到 Shell 里了。不需要任何其他外界的辅助就可以使用。这样的命令也被称为“built-in 命令”。当然, Shell 本身提供的命令是有限的, 如果想要执行其他比较复杂的命令就需要**脚本**或者**可执行程序**了。当用户在命令行输入一个命令时, Shell 会在一定地方查找和这个命令或程序, 或者看看是不是它的内建命令或函数, 如果均没有找到就会给出“command not found”的错误。



### 3.2.1 目录相关

在 Linux 中，目录的和路径概念和 Windows 有所不同。Linux 的文件系统采用的是树形结构，树的根就是根目录 (root directory)，用一个正斜杠 / 表示，树的叶子是文件，中间结点则是目录。要注意的是，Linux 的路径名中用正斜杠 / 分隔目录名，而 Windows 中用的则是反斜杠 \。

切合目录的命令是 `cd`，它的基本用法如下

```
1 cd [DIR]
```

其中 `dir` 为路径名，可以为相对路径或绝对路径。下面是一些例子

```
1 cd /tmp      # 切换到绝对路径 /tmp 下
2 cd bin       # 切换到当前目录下的 bin 目录下，如果不存在则报错、
3 cd ../bin    # 切换到上层目录下的 bin 目录下，这里 .. 表示当前目录的
               上层
4 cd -         # 切换到刚才所在的目录，这个用法非常方便
```

使用命令 `cd` 后，提示符显示的路径也会发生变化。

查看当前的路径位置可以使用 `pwd` 命令。

```
1 pwd
```

`pwd` 命令无需给任何参数，即可显示当前的位置。

我们可以用 `cd` 命令来切换目录，那么如何查看当前目录下的内容呢？这就需要命令 `ls`，它的基本用法如下

```
1 ls [OPTION]... [FILE]...
```

下面给出一些常见用法。

```
1 ls          # 列出当前目录的文件
2 ls -a       # 列出当前目录的所有文件，包括隐藏文件
3 ls -l       # 列出当前目录的文件并显示详细信息
4 ls /tmp     # 列出 /tmp 文件夹的内容
5 ls -R       # 递归地列出当前目录下的文件
```

值得说明的是，这里的选项可以进行组合，例如 `ls -al` 表示列出所有文件，包括隐藏文件，并显示详细信息。

### 3.2.2 文件基本操作

光有列出文件还不够，我们还需要对文件做各种操作，例如查看内容，修改内容，复制，移动，删除等。在这一小节我们将列出这些操作的具体用法。

查看内容可用的命令有 `cat`, `more`, `less` 等，它们的用法为

```
1 cat FILE
2 more FILE
3 less FILE
```

`cat` 用于将文件内容直接显示在屏幕上，而 `more` 和 `less` 更像是一种交互式的阅读器，可提供全屏滚动，查找，快速跳转等功能。一般来讲 `less` 命令功能较为丰富。这其实是一个典故“less is more”。

要修改一个文件的内容时，需要使用**编辑器**。在命令行的模式下，我们需要使用命令行模式的编辑器来进行文件的修改。常用的命令行编辑器有 ViM 和 Emacs，它们都是很强大的编辑器，初学者可以挑选一个来学习。不过似乎 Emacs 更容易产生劝退效果，因为初学者大概率不知道它是怎么退出的。另一个重要的编辑命令是 `sed`，它本质上和 ViM/Emacs 编辑器不同的地方在于它是非交互式的，这在不方便打开文件或进行批处理编辑时非常有用。它的语法为

```
1 sed [OPTIONS] ACTIONS [FILE]
```

例如在某文件的最末尾添加一行 Hello world，可以使用

```
1 sed '$a Hello world' a.txt
```

注意，`sed` 将编辑好的内容输出到屏幕上，因此原文件的内容不会变。如果想要进行“原地”的更改需要增加 `-i` 选项。例如

```
1 sed -i 's/foo/bar/g' a.txt
```

这条命令会把 `a.txt` 中所有的 `foo` 都替换成 `bar`。

复制文件使用的命令为 `cp`。它的用法为

```
1 cp [OPTION]... SOURCE DEST
2 cp [OPTION]... SOURCE... DIRECTORY
```

在这里细心的读者已经注意到了我们写出了两种通用的用法。前者表示将文件 `SOURCE` 复制到文件 `DEST`，即复制后的文件取名为 `DEST`。后者表示将多个文件 `SOURCE` 复制到一个指定的目录 `DIRECTORY` 下，注意用法里的 `SOURCE` 后面出现了三个点，这表示 `SOURCE` 参数可重复任意多次，而 `DIRECTORY` 由于是目标文件夹，因此只能有一个，且必须存在。不加任何选项的 `cp` 命令只能复制普通文件，**不能复制目录**。若要复制目录，需要加上 `-r` 参数，这是递归复制的意思。以下是几个例子。

```
1 cp file1.txt file2.txt           # 将文件复制到文件
2 cp file1.txt file2.txt folder/   # 将多个文件复制到目录 folder
3 cp -r folder1 folder2           # 将文件夹复制到文件夹
```

在复制目录时有一个细节，使用 `cp -r folder1 folder2` 时，若目录 `folder2` 存在，则 `folder1` 会被复制到 `folder2` 内部；若不存在，则直接创建一个新的目录，名为 `folder2`，其内容和 `folder1` 相同。

除了复制外，我们还可对文件或文件夹进行移动和重命名操作。这两个操作本质相同，使用的命令均为 `mv`。它的用法为

```
1 mv [OPTION]... SOURCE DEST
2 mv [OPTION]... SOURCE... DIRECTORY
```

和 `cp` 命令基本一样，只不过做的是移动操作而非复制。以下是几个例子。

```
1 mv file1.txt file2.txt           # 重命名
2 mv file1.txt file2.txt folder/   # 将多个文件移动到目录 folder
3 mv folder1 folder2              # 重命名目录
```

使用 `mv` 时不需要递归处理，这点和 `cp` 不同。使用时间长了之后大家可能会发现，移动操作 `mv` 在很多情况下都要比 `cp` 要快。其背后的原因是在**同一个文件系统中**对文件进行移动或重命名时，只需要更改文件引用，不需要实际对数据进行真正的移动；而复制操作则是要对数据本身进行拷贝，因此数据越大时间越长。当在**不同文件系统时**对文件的移动需要真正往另一系统上写数据，因此 `mv` 命令会明显变慢。

我们还可对文件进行删除操作，删除操作的命令是 `rm`。**Linux 和 Windows 的最大不同之处在于，删除了的文件就彻底不见了，没有类似于回收站的东西。因此执行删除命令时需要思考再三。**它的基本用法是

```
1 rm [OPTION].. FILE...
```

注意的是不加选项的 `rm` 只能删除普通文件，若要彻底删除目录和里面的内容需要加上递归删除的选项 `-r`。

```
1 rm file1.txt           # 删除一个文件
2 rm file1.txt file2.txt # 删除多个文件
3 rm -r folder           # 删除目录
```

有些比较业余的网络博客会说千万不要试 `rm -rf /`，这样会把整个系统删掉。但在比较新版的 `rm` 中，这个操作通常是会失效的，其原因是 `rm` 默认情况下会保留根目录。想要去掉这个保护就必须使用

```
1 rm -rf --no-preserve-root /
```

此时这个命令才真正变成了一个危险的命令。感兴趣的同学可以试一下它。

Linux 中目录的地位比较特殊，因此在文件系统中专门创建和删除目录的命令。`mkdir` 用于创建，`rmdir` 用于删除。注意，`rmdir` 只能删除空目录，当目录里存在内容时还是需要使用 `rm -r` 进行删除。以下是一些基本用法：

```
1 mkdir folder           # 创建目录
2 mkdir -p folder/sub/   # 逐级创建目录
3 rmdir folder           # 删除目录
```

### 3.2.3 文件打包

在 Windows 中我们经常需要对文件进行打包，例如压缩成 `zip` 或者 `rar` 文件，在 Linux 如何去做呢？Linux 自带的打包命令为 `tar`，它的基本用法为

```
1 tar -cf ARCHIVE [FILE...] # 打包
2 tar -xf ARCHIVE [MEMBER...] # 解包
```

例如将当前目录下的 `folder` 目录打包成一个压缩包，可以用

```
1 tar -cf archive.tar folder
```

这里 `archive.tar` 表示**档名**，它需要紧跟在选项 `-f` 的后面，`folder` 为要打包的目录。对刚打包的文件进行解包，可以用

```
1 tar -xf archive.tar
```

`tar` 命令只有打包的功能，没有压缩的功能。如果要在打包的时候压缩，需要额外加入参数。

```
1 tar -czf archive.tar.gz folder    # 使用 gzip 方式压缩
2 tar -cjf archive.tar.bz2 folder   # 使用 bzip2 方式压缩
3 tar -cJf archive.tar.xz folder     # 使用 xz 方式压缩
```

相应地，在解包的时候同样需要加对应的参数，否则无法解包。

```
1 tar -xzf archive.tar.gz folder    # 使用 gzip 方式解压缩
2 tar -xjf archive.tar.bz2 folder   # 使用 bzip2 方式解压缩
3 tar -xJf archive.tar.xz folder     # 使用 xz 方式解压缩
```

在这里注意，我们在打包的时候故意将文件名后面增加了压缩方式，例如 `.tar.gz`。其道理是显然的：为了让收压缩包的人知道我们是用什么工具进行压缩的，从而使用相应的解压方式。另外，不同压缩方式消耗的时间和产生的文件大小不同，例如上面的例子中，使用 `xz` 方式进行压缩得到的文件一般最小，但其耗时是比较久的。

在 Linux 中，我们如果收到了来自别的系统用户发过来的压缩包，应该如何处理呢？例如在 Windows 中常见 `.zip`，`.rar`，`.7z`，在 MacOS 中比较常见 `.zip`，Linux 的 `tar` 命令不能处理这样的压缩格式。但 Linux 本身也可以安装解压这些格式的软件，且安装过程非常简单。顺带一提，由于是命令行环境，Linux 下的 RAR 软件是没有广告的（用 Windows 这么多年让我从 WinRAR 转 7zip 的重大原因就是广告）！

### 3.2.4 查看帮助文档

当不知道一个命令具体用法时，除了上网去搜，问其他人，一个最直接也是最有效的方法就是查看本地文档。查看文档的命令为

```
1 man <command>
```

例如 `man ls` 可以查看命令 `ls` 的帮助文档。初学者读文档时很可能会不喜欢里面的符号，但了解手册里的编写规则之后阅读文档就能准确查到自己想要的功能。一般来讲，一个文档有若干小节 (Section)：

- NAME：一句简短介绍该命令名字和主要功能。
- SYNOPSIS：命令的语法/用法，每一种用法都对应一条 SYNOPSIS。
- DESCRIPTION：命令的细致描述，例如 ARGUMENT 如何填写。
- OPTIONS：介绍每个选项的作用。

- AUTHOR/BUG REPORT/COPYRIGHT: 作者/BUG 报告/版权。

手册中的符号都有固定的含义，在这里我们举出一些常见规则：

| 规则               | 含义                              |
|------------------|---------------------------------|
| <b>bold</b>      | 黑体，使用的时候需要原封不动照打下来。             |
| <i>italic</i>    | 斜体 <sup>3</sup> ，使用的时候要替换成相应参数。 |
| [ <b>-abc</b> ]  | 任何或所有方括号 [] 中的内容都可以不出现。         |
| <b>-a -b</b>     | 选项 -a 和 -b 只能出现一个（不能同时使用）。      |
| <b>arg ...</b>   | 参数 arg 可重复出现任意多次。               |
| <b>[exp] ...</b> | 方括号 [] 内部整体可以重复出现任意多次。          |

3.3 Linux 目录结构

Linux 系统的目录结构和 Windows 有很大区别。在 Windows 中我们经常说“C 盘，D 盘”，但在 Linux 中没有相关的概念。所有的目录都是从根目录 / 衍生出来的。在 Linux 下不同硬盘分区以挂载点的形式存在于目录结构中。在本节我们需要了解 Linux 的目录结构是如何组织的。

- /bin: 供所有用户使用的完成基本任务（在单人维护模式下还能够被操作的指令）的可执行文件，如 bash, mv, cp, touch, more, chmod, ls, which。注意，在某些系统（如 CentOS 7）中已经不提供单独的 /bin；
- /boot: 启动 Linux 时使用的一些核心文件。如操作系统内核、引导程序 Grub 等；
- /dev: 所有系统设备文件，从此目录可以访问各种系统设备；
- /etc: 系统和应用软件的全局配置文件，如 passwd, shadow, hosts, timezone；
- /home: 普通用户的家目录所在的位置；
- /lib: 系统必需动态链接共享库文件和内核模块，注意，在某些系统（如 CentOS 7）中已经不提供单独的 lib；
- /lib64: 64 位系统动态链接共享库文件；
- /lost+found: 几乎每个分区都有，储存发生意外后丢失的文件，只有 root 用户才能打开；
- /media: 可移动设备的挂载点，如光盘，u 盘；
- /mnt: 空，可以作为外挂设备的挂接点；
- /opt: 一些第三方软件的安装位置；
- /proc: 存在于内存中的虚拟文件系统，保存内核和进程以及处理器等状态信息；
- /root: 超级用户的家目录（思考：为什么超级用户的家目录要和普通用户分开？）；
- /run: 系统开机后所产生的各项信息放置到 /var/run 目录下，新版的 FHS 则规范到 /run 下面；

- `/sbin`: 供超级用户用的可执行程序, 如 `reboot`, `shutdown`, `ifconfig`;
- `/tmp`: 创建临时文件或目录 (系统一般会定期清理);
- `/usr`: UNIX System Resources, 不是 USER 的缩写。静态的用户级应用程序数据;
  - `/usr/bin/`: 供一般用户用的非必需的可执行程序;
  - `/usr/games/`: 与游戏比较相关的数据放置处;
  - `/usr/include/`: 存放编写程序需要的头文件的目录;
  - `/usr/lib/`: 系统的非必需的库文件;
  - `/usr/local/`: 空, 安装本地程序的默认路径
  - `/usr/sbin/`: 供超级用户用的非必需的可执行程序;
  - `/usr/share/`: 放置只读架构的数据文件, 包括共享文件, 程序帮助文档, 模板, 程序图标等;
  - `/usr/src/`: 内核和软件的源代码的位置;
- `/var`: 动态的应用程序数据;
  - `/var/cache`: 应用程序的缓存文件;
  - `/var/lib`: 应用程序的信息、数据, 如数据库的数据等都存放在此文件夹;
  - `/var/local`: `/usr/local` 中程序的信息、数据;
  - `/var/lock`: 锁定文件, 许多程序遵循在 `/var/lock` 中产生一个锁定文件的约定, 以用来支持他们正在使用某个特定的设备或文件。其他程序注意到这个锁定文件时, 就不会再使用这个设备或文件;
  - `/var/log`: 系统日志文件, 如登录文件, 一般来讲有些日志只能管理员查询, 另一些日志是公开的;
  - `/var/opt`: `/opt` 中程序的信息、数据;
  - `/var/run`: 正在执行着的程序的信息, 如 PID 文件应存放于此;
  - `/var/tmp`: 临时文件 (不会被系统定期清理);

### 3.4 用户管理与文件权限

在 Linux 中, 用户管理和权限是很重要的概念。这是因为 Linux 的应用场合大多都是多用户环境, 所以在同一个系统中, 登录到该系统的人是谁, 他可以干什么事情不能干什么事情都需要严格进行控制。之前我们使用自己笔记本电脑的时候基本不会考虑这一问题: 因为电脑是我自己的, 只有我一个人使用, 因此想干什么就干什么。但在集群中这样的操作就不行了: 试想如果能让一个用户随便查看其他人的文件甚至修改或删除系统文件, **那还了得**? 鉴于上述的实际需求, Linux 设计了非常完善的机制解决权限问题。

### 3.4.1 Linux 用户

和多数系统一样，Linux 系统上可以设置很多用户，且这些用户可以同一时间登录到系统进行操作，甚至可以相互交互。例如下面的语句可以让你和其他登录集群的用户进行对话：

```
1 write <user>
2 Hi, nice to meet you.
3 ^D
```

这样对方的命令行就会立即显示 Hi, nice to meet you. 这句话。如果你想拒收这样的消息，可以用下面的命令：

```
1 mesg n
```

Linux 的每一个用户都有唯一的用户名，并且每个用户都有一个默认用户组 (Default group)，但一个用户可以属于多个用户组 (Group)。存储系统用户和用户组信息的文件大致有 /etc/passwd, /etc/shadow, /etc/group, /etc/gshadow。其中 /etc/passwd 存储了用户的常规信息，例如用户名，UID，默认组，默认 Shell 等，而 /etc/shadow 则存储一些敏感信息，如用户密码（以加密方式储存）等。

粗略来说，Linux 用户可以分为以下几种：

1. 超级用户 root：即系统管理员。root 用户拥有在这个系统的最高权限，几乎可以做任何事情。因此 root 用户也是黑客攻击服务器的首要目标（没有之一）。
2. 普通用户：即日常登录系统进行操作的用户。一般来说每一个普通用户都会有一个家目录 (Home directory)，用于存放他们自己的文件。在多数集群系统环境下，只有用户自己能访问家目录。
3. 系统用户：为了某些必要系统服务而开设的特殊账户。这些账户一般不直接由人来操作，多数情况下也不可以直接登录系统。系统管理员在开设服务的时候，由于安全因素等考虑一般不会以 root 身份运行，于是系统用户来减少权限泄露的风险。一般情况下系统用户的家目录可以不存在。

### 3.4.2 基本文件权限

Linux 系统中的每个文件/目录都有属主 (owner)，即拥有该文件的用户。大部分系统文件的属主是超级用户，普通用户无权修改它们。每个文件/目录除了有属主以外，还有一个用户组 (group) 的属性，表示哪一个用户组拥有该文件。默认情况下，文件的组就是属主的默认组。这样，对于一个文件来说，它的基本权限被分成三类：第一类是该文件的属主 (owner) 的权限，第二类是该文件的用户组 (group) 中的用户的权限，第三类是所有其他用户 (others) 的权限。

Linux 文件系统的文件有三种基本权限：可读 (r)，可写 (w)，可执行 (x)。它们的具体解释如下：

- 可读 (r)：可以查看里面的内容。目录的 r 权限是指可以读取其中的文件列表。
- 可写 (w)：可以更改其中的内容。目录的 w 权限是指可以在目录中创建或删除内容（删除时无视文件本身是否有 w 权限）。



- 可执行 (x): 可以将其作为可执行程序执行 (注意其与可读的区别)。例如我写一个 Shell 脚本, 它具有可读权限是指我可以看到代码, 它具有可执行权限是指我可以执行里面的内容。目录的 x 权限是指是否可以切换 (cd) 到这个目录下。

在这里注意, 可执行权限并不是所有文件系统都有的, 例如 Windows 系统常用的 NTFS 文件系统就没有可执行权限。因此这会产生一个重要的问题: 若在 NTFS 上放置可执行文件, 那么这个可执行文件永远没办法“执行”。因此在使用 Linux 的时候不要把软件安装在 NTFS 分区中, 那样会导致你的软件无法运行 (这是我刚接触 Linux 一个月的时候犯下的错误)。另外, 普通用户的文件权限对 root 用户是无效的。root 用户可以无视权限来修改普通用户的文件或者进入普通用户的目录。

查看一个文件的基本权限可以使用

```
1 ls -l FILE
```

例如使用 `ls -l /etc/passwd` 就可以查看该文件的权限。

```
1 ls -l /etc/passwd
2 -rw-r--r-- 1 root root 2469 Jun 17 2018 /etc/passwd
```

这里前面的十个字符中就含有权限信息。这十个字符中, 第一个字符实际上表示的是文件的类型, 并不是权限。从第二个字符起, 每三个一组来分别表示属主 (owner) 的权限, 用户组 (group) 的权限, 以及其它人 (other) 的权限。之后中间的两个 root 分别表示该文件的属主和用户组。因此该文件的属主可以进行读写操作, 用户组内的用户和其它用户仅仅可以进行读操作。由于这个文件不是可执行文件, 执行它没有意义, 因此所有人都没有执行权限。

文件的属主可以更改文件的权限。所使用的指令是

```
1 chmod MODE FILE
```

这里 MODE 可以有两种写法:

```
1 chmod u+w FILE      # 给予 owner 写权限
2 chmod 755 FILE      # 给予 owner 读, 写, 执行权限, 给予 group 和
   other 读和执行的权限
```

第二种写法是权限的八进制表示, 每一组权限用一个 0-7 的数字表示, 读是 4, 写是 2, 执行是 1。不同权限的组合就是将数字简单相加, 因此读 + 执行的表示是 5。

既然文件的权限可以更改, 那么文件的 owner 是否也可更改呢? 更改文件属主的命令是

```
1 chown [owner][:[group]] FILE
```

表示把 FILE 的属主和用户组进行更改, 但更改文件属主的命令只有 root 用户可以执行。即不能将不是自己的文件的属主改成自己, 就算是文件的原属主也不能把自己的文件“送给”别人。不能把别人文件改成自己的很好理解, 在现实生活中你也不能“明抢”别人的东西; 不能把自己的文件改成别人实际上也有一定道理, 在集群中因为资源有限, 管理员通常都会对每个用户设置磁盘配额, 即每个用户的文件总大小最多不能超过一个定值, 如果可以将自己的文件属主改成别人, 那么可以用这种方式“明抢”他人的磁盘配额, 这当然也是不允许的。

### 3.4.3 特殊文件权限

除了基本文件权限（`rwX`）之外，我们还有其它三种特殊权限：

- **SUID(SetUID)**：出现在文件属主的 `x` 权限上，仅对二进制程序有效，程序的执行者在执行该程序过程中具有程序属主的权限。例如：普通用户不能更改 `/etc/shadow` 文件，那么他是如何更改密码的呢？
- **SGID(SetGID)**：出现在文件用户组的 `x` 权限上。仅对二进制程序和目录有效。当作用在二进制文件上时，程序的执行者在执行该程序的过程中具有程序用户组的权限。作用在目录上时，用户在此目录下的用户组将会变成目录的用户组（即使该用户本身不是该组的成员）。SGID 可以用来制作共享目录。
- **SBIT(Sticky BIT)**：出现在其他用户的 `x` 权限上。仅对目录有效。当用户在该目录下创建新文件或目录时，仅有自己和 `root` 才有权力删除。SBIT 通常用于制作临时目录，也可辅助制作共享目录。

### 3.4.4 拓展：更精细的权限控制

上面我们提到了一个文件可以对 `owner`，`group`，`other` 设置权限，这样的权限控制还是有点太粗略了。例如，在集群上，我要和另一指定用户共享一个文件的读写操作，但不允许其他普通用户对这个文件读写。如果利用之前介绍的权限控制方式，必须先找系统管理员给我们创建一个单独的分组，然后将这个文件的 `group` 权限修改为可读写。显然管理员不可能每天光因为共享文件的事情而无休止地创建用户组，因此这并不是一个好的做法。为了实现这种更精细的权限控制，Linux 的 `ext4` 文件系统引入了 ACL(Access Control List，访问控制列表) 的特点。

具体来说，我们可以使用如下两个命令来获取，设置访问控制列表

```
1 getfacl FILE # 获取 ACL
2 setfacl -m|-x acl_spec FILE # 设置 ACL
```

这两个命令看起来也比较复杂，我们给出几个例子：

```
1 setfacl -m u:jack:rwX FILE # 在文件 FILE 中给用户 jack 赋予读，
    写，执行的权限
2 setfacl -x u:jack FILE # 在文件 FILE 中去掉用户 jack 相关的权
    限
```

## 3.5 特殊设备

Linux 有一个著名的说法就是 “Everything is a file(一切皆文件，虽然这么说有些夸张的成分)”，但在 Linux 系统中确实大多数和系统相关的东西都可以用一个文件来表示，哪怕它本身不是一个文件。例如许多硬件设备在 Linux 系统中看也是一个“文件”，借助于其文件表示我们可以使用操作 Linux 文件系统的方式直接对该设备进行操作。

多数设备文件都放在 `/dev` 目录，例如：

| 名称           | 说明             |
|--------------|----------------|
| /dev/sda     | 系统里第一块硬盘       |
| /dev/sda1    | 系统里第一块硬盘的第一个分区 |
| /dev/fd      | 软驱             |
| /dev/nvidia0 | 第一块 NVIDIA GPU |

但 /dev 中还有很多比较特殊的设备，例如

| 名称           | 说明                                     |
|--------------|----------------------------------------|
| /dev/stdin   | 标准输入                                   |
| /dev/stdout  | 标准输出，一般指命令行窗口                          |
| /dev/stderr  | 标准错误，一般也指命令行窗口                         |
| /dev/null    | 黑洞设备，会“吞掉”输入进去的一切数据，从其读取总会读到 EOF(文件终止) |
| /dev/zero    | 读取时无限提供空字符 0x0                         |
| /dev/random  | 随机设备，会根据系统目前的熵返回随机的字节（可以看成真随机数发生器）     |
| /dev/urandom | 非阻塞式随机设备，当系统的熵不足时，会采用非阻塞式随机数生成算法       |

设备和特殊设备的使用非常广泛，下面给几个例子

```
1 # 复制硬盘，完全复制两个硬盘的分区情况和数据
2 dd if=/dev/sda of=/dev/sdb
3
4 # 丢弃标准输出
5 firefox > /dev/null
6
7 # 创建指定大小的文件
8 dd if=/dev/zero of=test.bin bs=1M count=100
9
10 # 生成 1000 以内的随机数（这里面用了较多大家不认识的命令）
11 echo "$(cat /dev/urandom | head -c 10 | cksum | awk -F ' ' '{print
    $1}') % 1000" | bc
```

## 4 编译程序

### 4.1 gcc 基本用法

C 语言程序需要先通过**编译**生成可执行程序才能运行。根据这一特点，C 语言又被归类为**编译型语言**。与之相对的是**解释型语言**，即不存在将代码变成独立的二进制可执行程序这个环节，而是直接将代码输入给解释器（Interpreter）执行。典型的解释型语言包含 python，ruby，perl 等。

gcc 是 Linux 系统中常用的编译器，它的基本语法为：

```
1 gcc [OPTION] infile
```

表示从源码 `infile` 编译为可执行文件，方括号 `[]` 里是可选的选项，当不指定 `-o` 选项时，默认输出文件为 `a.out`。例如，对于如下的 `hello-world` 程序，只需要使用 `gcc hello.c` 就可以完成编译。

```
1 #include <stdio.h>
2
3 int main(){
4     printf("Hello world!\n");
5     return 0;
6 }
```

看到这里你可能会觉得似乎没什么可以讲的，但在实际场景中，编译一个 C 语言项目是比较复杂的。我们通常都需要指定一些选项完成编译。常用的选项为

- `-g`: 指示编译器在编译时产生调试信息，打开调试选项；
- `-c`: 仅编译，而不进行链接，此时输出为 `.o` 文件；
- `-L`: 指定链接时，搜索链接库的路径（和运行时搜索路径无关!）；
- `-I`: 指定编译时头文件的路径；
- `-l`: 指定链接库的简称，如数学库 `-lm`；
- `-D`: 编译时给定宏定义；
- `-f`: 启用某种特性，例如 `-fPIC`（生成位置独立代码），`-fopenmp`（启用 OpenMP 支持）等；
- `-m`: 设置和机器架构相关的选项，例如 `-m64`（生成 64 位代码）等；
- `-Wall`: 会打开一些很有用的警告选项，建议编译时加此选项；
- `-std`: 指定 C 标准，如 `-std=c99` 使用 c99 标准；
- `-Wl,:`: 传给链接器的选项，`gcc` 本身不处理它们，这些选项是链接器 `ld` 使用的。

在这里注意到编译器的选项和我们之前命令选项有所不同。选项后如果有选项参数，有些选项和参数之间是**没有空格分隔的**。例如 `-I`，`-L`，`-l`，`-D` 等。

## 4.2 编译 OpenMP 程序并运行

为了让编译器能识别 `#pragma omp parallel` 这样的语句，在编译时必须使用 `-fopenmp` 选项来启动 OpenMP 支持。

```
1 gcc -o omp_hello omp_hello.c -fopenmp
```

这个命令会生成多线程程序 `omp_hello`。需要注意 `-fopenmp` 这个选项实际上做了很多复杂的操作，包括但不限于：将众多 `#pragma omp` 语句编译成 OpenMP 的底层实现；自动添加

`-lgomp` 等编译/链接参数。如果去掉这个选项（并手动添加 `-lgomp` 库），程序也可以编译通过，但就不是并程序了。现阶段可以不用深入了解这个选项。

运行 OpenMP 程序和运行普通程序实际无区别。默认情况下，OpenMP 程序会使用系统所有可用的核心（线程数 = 核心数）。但用户可以在并行块上指定运行线程数，或者使用环境变量 `OMP_NUM_THREADS` 来控制。例如：

```
1 export OMP_NUM_THREADS=4
2 ./omp_hello # 此时会使用 4 线程运行
```

### 4.3 编译 MPI 程序并运行

课程中写的 MPI 程序需要进行**编译**才能够运行，而编译 MPI 程序所用的 C 编译器就是 `mpicc`。它是编译器的一个包装，实际上是调用了真正的 C 编译器 `gcc` 并增加了许多 MPI 的支持。

使用这个命令之前，需要添加集群安装的 `mpich` 或 `openmpi` 到当前环境模块中。

```
1 module add mpich # 或 module add openmpi
```

它的基本用法为

```
1 mpicc [OPTION]... [-o OUTFILE] SOURCE [-lLIBRARY] [-IHEADER]
```

例如可用以下命令编译

```
1 mpicc -o hello mpi_hello.c
2 mpicc -o cpi mpi_cpi.c -lm
3
4 # 仅编译，生成目标文件
5 mpicc -c mpi_hello.c
```

编译好的程序可使用 `mpiexec` 运行。运行的时候需要给出需要启动多少个进程。

```
1 mpiexec -n 4 ./hello
```

注意 `hello` 前面的 `./` 不能省略，这是因为可执行程序 `hello` 不在系统可执行程序的搜索路径 `PATH` 里。

### 4.4 使用运行库

在 Linux 下开发软件时，完全不使用第三方函数库的情况是比较少见的，通常来讲都需要借助一个或多个函数库的支持才能够完成相应的功能。从程序员的角度看，函数库实际上就是一些头文件和库文件的集合。以下介绍在编译过程中如何指定运行库，这里以编译器 `gcc` 为例，`mpicc` 等其他情形类似。

Linux 下的大多数函数都默认将头文件放到 `/usr/include/` 目录下，而库文件则放到 `/usr/lib/` 目录下，但并不是所有的情况都是这样。下面一个例子展示了在链接时使用非默认位置的外部库：

```
1 gcc -o axpby.exe -I../include -L../lib main.c -laxpby
```

值得好好解释一下的例子中的 `-laxpby`, 它指示 gcc 去链接库文件 `libaxpby.a` 或 `libaxpby.so` (如果都存在默认优先使用后者, 稍后解释二者的差异)。Linux 下的库文件在命名时有一个约定, 那就是应该以 `lib` 三个字母开头。由于所有的库文件都遵循了同样的规范, 因此在用 `-l` 选项指定链接的库文件名时可以省去 `lib` 三个字母。也就是说 gcc 在对 `-laxpby` 进行处理时, 会自动去链接名为 `libaxpby.a` 或 `libaxpby.so` 的库。当然, 我们这里还用 `-L` 选项指定了库文件的搜索路径。

Linux 下的库文件分为两大类分别是动态链接库 (通常以 `.so` 结尾) 和静态链接库 (通常以 `.a` 结尾), 两者的差别仅在程序执行时所需的代码是在运行时动态加载的, 还是在编译时静态加载 (直接打包到可执行文件中) 的。默认情况下, gcc 在链接时优先使用动态链接库, 只有当动态链接库不存在时才考虑使用静态链接库, 如果需要的话可以在编译时加上 `-static` 选项, 强制使用静态链接库。

静态库之所以称为【静态库】, 是因为在链接阶段, 会将汇编生成的目标文件 `.o` 与引用到的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接。其实一个静态库可以简单看成是一组目标文件的集合, 即很多目标文件经过压包后形成的一个文件。静态库特点总结:

- 静态库对函数库的链接是放在编译时期完成的。
- 程序在运行时与函数库再无瓜葛, 移植方便。
- 浪费空间和资源, 因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。
- 静态库对程序的更新、部署和发布会带来麻烦。如果静态库更新了, 所以使用它的应用程序都需要重新编译、发布给用户。

生成静态库需要使用 `ar` 命令。

```
1 ar rcs ../lib/libaxpby.a axpby.o
```

动态库在程序编译时并不会被链接到目标代码中, 而是在程序运行是才被载入。不同的应用程序如果调用相同的库, 那么在内存里只需要有一份该共享库的实例, 规避了空间浪费问题。动态库在程序运行时才被载入, 也解决了静态库对程序的更新、部署和发布带来的麻烦。用户只需要更新动态库即可, 而原有依赖此库的程序无需再进行编译。动态库特点总结:

- 动态库把对一些库函数的链接载入推迟到程序运行的时期。
- 可以实现进程之间的资源共享, 因此动态库也称为共享库。
- 将一些程序升级变得简单。
- 甚至可以真正做到链接载入完全由程序员在程序代码中控制 (显式调用)。

生成动态库需要在 gcc 中指定 `-shared` 和 `-fPIC` 参数。

```
1 gcc -o ../lib/libaxpby.so -shared -fPIC axpby.c
```



如果编译时涉及到库的链接，需要额外注意库文件和其它输入文件（.o 与 .c 等）书写顺序。基本原则是：若 A 依赖库 B，则需要把 A 写在库 B 之前（这里 A 可能是目标文件，源文件，或者其他库文件；若是目标文件与源文件之间的相互依赖则不需要考虑顺序）。假如 main.c 中的函数依赖 libaxpby.a 这个库（假设其在 ../lib 目录中），那么：

```
1 gcc -L../lib main.c -laxpby      # 正确
2 gcc -L../lib -laxpby main.c      # 错误，会报错有未定义的引用
```

出现这种现象的原因是进行链接时，链接器会从左到右来解析符号，若遇到已经定义但目前还没有使用的符号将会跳过。此例子中，main.c 使用了 libaxpby.a 中的函数，但若将 -laxpby 放在前面，则会因为没有使用对应函数而丢掉相应符号，当再次遇到 main.c 时，刚刚丢掉的内容已经无处可寻，最终引发报错。

如果上述例子链接时使用的是动态库 libaxpby.so，在有些链接器上可能不会引发报错。例如使用 CentOS 7 默认的链接器 ld，以下的链接都是 OK 的：

```
1 gcc -L../lib main.c -laxpby      # 正确
2 gcc -L../lib -laxpby main.c      # 同样 OK
```

其原因是一些链接器处理动态库有不同的机制，主要和 --no-as-needed 和 --as-needed 这两个参数相关。虽然在某些情况也可以工作，但我们依然建议严格按照依赖顺序进行书写，以防踩坑。

## 4.5 拓展：编译的四个阶段

这部分我们拓展一下，编译器 gcc 是如何产生可执行文件的呢？实际上它经过了四个阶段。

### 预处理 (Preprocessing)

使用预处理器 cpp，将所有的 #include 头文件以及宏定义替换成其真正的内容，预处理之后得到的仍然是文本文件，但文件体积会大很多。

```
1 cpp example.c -o example.i
```

这里也可以用 gcc -E 控制编译进度，预处理后可以对比文件大小和代码行数。打开后可以发现所有的注释都被删除了，宏的名字也已经找不到了，使用到宏的地方会把宏全部替换为具体的数值。

### 编译 (Compilation)

使用编译器 gcc，将经过预处理之后的程序转换成特定汇编代码 (assembly code) 的过程。

```
1 gcc -S example.c -o example.s
```

上述命令中 -S 让编译器在编译之后停止，不进行后续过程。编译过程完成后，将生成程序的汇编代码 example.s，这也是文本文件。打开后可以发现汇编代码中都是分段的，所有的 C 语句都已经转换成了汇编语句等。值得一提的是 OpenMP 中的 #pragma omp 系列指令在这个阶段会被转换成汇编语句，如果仔细看代码的话会看到很多我们没有写到的函数名。



### 汇编 (Assemble)

使用汇编器 `as`，将上一步的汇编代码转换成机器码 (machine code)，这一步产生的文件叫做目标文件，是二进制格式。

```
1 as example.s -o example.o
```

也可以用 `gcc -c` 来控制，

```
1 gcc -c example.c -o example.o
```

### 链接 (Linking)

使用链接器 `ld`，将多个目标文以及所需的库文件链接成最终的可执行文件 (executable file)。

```
1 ld -o run.exe example.o -lc
```

上面这个命令可以生成可执行文件，但运行时时报错。这是因为没有定义程序入口。利用这种方式链接的程序仍然不完整的，不过使用 `gcc` 进行链接就会把它变成一个真正的可执行程序。

```
1 gcc -o run.exe example.o
```

实际上，可以使用 `-v` 选项来检查 `gcc` 到底给 `ld` 传了什么参数进去。

```
1 gcc -v -o run.exe example.o
```

会看到很长一段输出，看来实际 `gcc` 做的操作要比我们想象得更复杂一些。

## 5 Linux 上程序的运行

### 5.1 PATH 变量

在 Linux 中输入一个程序名字时，Linux 会如何找到这个程序呢？答案就是环境变量 `PATH`。使用

```
1 echo $PATH
```

可以查看当前此变量的值。这个变量中不同的路径以 `:` 分隔，且处于前面的路径优先级较高。当系统遍历完该变量中的路径后，若还找不到该程序，就会提示 “Command not found” 了。

多数情况下，我们编译的程序都在当前目录下，而这些目录一般都不在环境变量 `PATH` 中。对于这种情况，我们还是可以使用**绝对路径**或**相对路径**来指定程序名，这样的话系统就直接按照指定的路径去寻找你的程序了。例如执行当前目录下的程序 `hello` 可以输入

```
1 ./hello
```

一个自然的问题就是：能不能在 `PATH` 中添加 `.` 来避免每次都要输入路径呢？是可以这样做的，但**非常不建议**。如果是服务器的管理者更应该注意这个问题。这其中的原因是什

呢？服务器管理员每天的日常就是要进行到不同目录下进行操作，有时候甚至会进入到普通用户的家目录或者是公用目录中。试想，如果有一个用户使坏在公用目录中放置了一个名为 `ls` 的程序，管理员再进入到该目录……后面会发生什么不用我再多说了吧。

## 5.2 动态库加载器/ELF 解释器

如果一个程序依赖动态库，那么在运行时应该如何找到这些库呢？这就要涉及到 Linux 下执行二进制可执行文件的机制。

一般来说，Linux 上的可执行文件或动态库文件大多是 ELF 格式 (Executable and Linkable Format)，当执行一个 ELF 文件时，Linux 系统内核会进行一系列比较复杂的准备工作。大致上说分为如下几个步骤：

1. Linux 内核读取 ELF 文件的头部信息 (header)，进行一些必要的检查。
2. 如果该可执行程序是动态链接的，则会取读取其 ELF 解释器 (ELF Interpreter，有时候也称为“动态库加载器”)，并将程序控制权交给 ELF 解释器。
3. ELF 解释器对程序的依赖库进行查找，定位符号入口，并将库加载到内存中。
4. 程序控制权交给原始可执行程序，程序正式运行。

当程序不是动态链接时，也就无需进行读取 ELF 解释器的操作。但在上一小节我们提到，几乎所有程序都需要外部库的依赖，因此上述步骤 2 和 3 基本都会出现。

值得好好一提的是上述的步骤 3，ELF 解释器是根据什么原则在磁盘中寻找动态库的呢？这就要涉及到动态库的查找顺序的问题。默认的 GNU 加载器 `ld.so`，按以下顺序搜索库文件：

1. 首先搜索程序中 `DT_RPATH` 区域 (不建议使用)，除非还有 `DT_RUNPATH` 区域。
2. 其次搜索环境变量 `LD_LIBRARY_PATH` 中的路径。如果程序的权限有 `SetUID/SetGID`，出于安全考虑会跳过这步。试想一下一个带有 `SetUID` 权限的程序依赖某外部库，现在我们找到这个库的源码并向其中增加一些恶意代码，重新编译一遍这个库，然后通过修改 `LD_LIBRARY_PATH` 变量让其指向魔改之后的库，如果 `LD_LIBRARY_PATH` 可以起作用的话这样做是不是对系统危害非常大呢？
3. 然后搜索程序中 `DT_RUNPATH` 区域，`DT_RUNPATH` 的优先级虽然不高，但是它的存在会直接短路 `DT_RPATH` 区域。这是由于 `DT_RPATH` 当初在设计的时候出现了一个“失误”，它拥有最高的优先级，因此程序编译好之后再想改动就非常困难。为了使得程序运行更具灵活性，人们由设计出了 `DT_RUNPATH`。
4. 搜索缓存文件 `/etc/ld.so.cache` (停用该步请使用 `-z nodeflib` 链接器参数)。
5. 搜索默认目录 `/lib`，然后 `/usr/lib` (停用该步请使用 `-z nodeflib` 链接器参数)。

注意，上面的五个顺序中的 `DT_RPATH` 和 `DT_RUNPATH` 是由链接器直接写在二进制程序中的，要想改动必须有专门的编辑 ELF 文件的工具。而 `LD_LIBRARY_PATH` 属于系统环境变量，用户可以比较轻松地修改这个值。

上面的顺序可通过 `man ld.so` 命令查看。如果是想让链接库的配置对所有用户可用，可以直接修改 `/etc/ld.so.conf` 然后使用 `ldconfig -v` 更新缓存文件（需要 `sudo` 权限）。或者也可以把库文件直接移入 `/lib` 或 `/usr/lib`，但不建议这样做。下面是一些使用示例，用 `-Wl` 来表示将后面的参数传给链接器，注意中间不要加空格。

```
1 # 使用 LD_LIBRARY_PATH
2 gcc -o axpby.exe main.c -I../include -L../lib -laxpby
3 echo $LD_LIBRARY_PATH
4 export LD_LIBRARY_PATH=${YOUR_PATH}:$LD_LIBRARY_PATH
5 ./a.out
6
7 # 使用 DT_RPATH
8 gcc -o axpby.exe main.c -I../include -L../lib -laxpby -Wl,-rpath=\
    $ORIGIN/../lib
9 # 使用 DT_RUNPATH
10 gcc -o axpby.exe main.c -I../include -L../lib -laxpby -Wl,-rpath=\
    $ORIGIN/../lib,--enable-new-dtags
```

上面的 `DT_RPATH` 和 `DT_RUNPATH` 中，我们使用了 `$ORIGIN` 来表示可执行程序的位置。这样可以保证在任何位置执行此程序都可以找到对应的库。在 `RPATH/RUNPATH` 中一般不建议使用相对路径，这是因为如果切换到其他目录来执行这个程序，依然会提示找不到动态库。`LD_LIBRARY_PATH` 则一般用于在开发测试时临时使用，设置较为灵活。也方便在软件更新较快的系统上迅速切换库的版本。

## 6 作业调度系统的使用

### 6.1 为什么要用调度系统

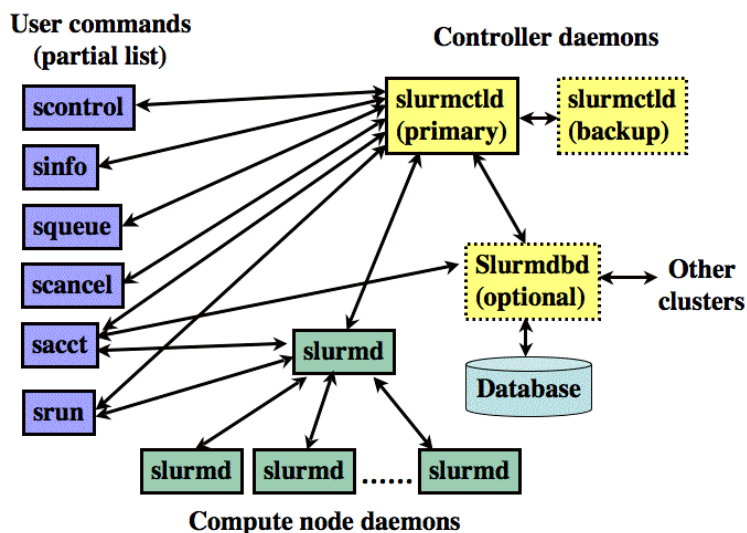
为什么要用调度系统，上来就开始算不好吗？**不好**。作业调度系统的使用可以极大减小用户程序之间的冲突，同时可追踪用户程序的运行情况。不过在某些执行效率方面会略有降低。我们说过了集群上其实是有很多的用户，如果两个用户在同一时间运行程序，且一个用户要测试自己的程序运行时间，两个人同时运行程序的时候势必要相互影响，这样的计时还能准确吗？当然不能。

- 在用户程序互不影响的前提下让各个程序按照指定的规则运行，注重资源的公平与合理的分配。
- 用户需要做的是提交任务，告诉系统该做什么，运行什么样的程序，需要多少资源，运行多长时间即可。不需要实时盯着屏幕看，非常适合**批处理 (batch)** 或**离线 (offline)** 任务。
- 实现任务和资源绑定，有利于资源的规范使用，也可以减少用户查询空余资源的痛苦。
- 可记录用户消耗的资源数，有利于用户审计。所有大型计算集群有一套计费标准，如果资源数量申请不合理，论文没写出来，经费却很快花光了！在收费集群上各位用户一定

要注意自己申请的资源大小，每一个 core 每一分钟都是需要花钱的。当然，在免费的小集群上，这并不意味着用户可以为所欲为。

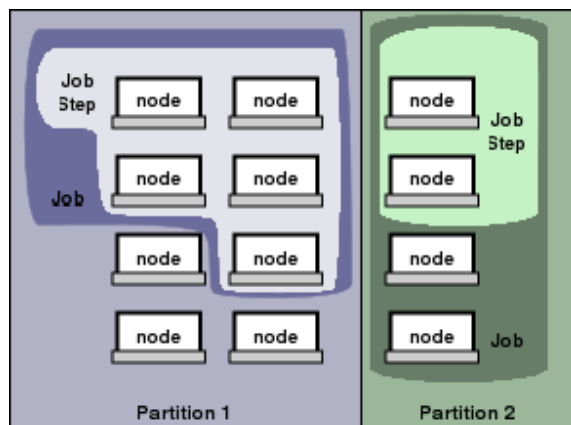
## 6.2 SLURM 调度系统的结构

集群用的是 SLURM 调度系统，由它负责集群的一切作业调度。实际上 SLURM 调度系统在每个集群节点上都会运行至少一个服务，用户通过 SLURM 命令和这些服务进行交互，而具体程序是如何运行则是交给这些服务完成。



- **slurmctld** 为主要的控制进程，负责处理提交的任务，任务的调度，任务控制等。只需要安装在控制节点上（例如管理节点），在某些系统里还会有备用的 SLURM 控制节点。
- **slurmdbd** 主要负责用户审计，例如账号的管理，qos 的管理等。可能需要与数据库进行交互 (mysql, mariadb)。只需要安装在控制节点上。如果不需要审计功能可以不安装这个模块。
- **slurmd** 计算节点控制进程，每个计算节点至少会安装一个。需要和其他节点的 **slurmd** 以及控制进程 **slurmctld** 进行通讯。主要功能是接收 **slurmctld** 分发的任务，创建资源池并执行任务。

在 SLURM 中有分区和 QoS 的概念，这两个概念对初学者而言是陌生的。



什么是分区 (Partition)? 为什么要指定? 不同的节点的特性和硬件属性不同, 设置分区可以帮助用户更好确定节点的特点, 进而选择最适合自己的节点进行运算。此外, 如果集群中部分机器是私有的, 那么设置分区可以使得只有部分用户能在这个分区提交 Job。总的来说, 分区可看做一系列节点的集合。需要注意的是, 分区的节点是可以重叠的: 即同一个节点可能同时属于不同分区, 这在 SLURM 中是允许的。试想一个场合: 某集群中有 40 台 CPU 机器, 一般用户只能在前 36 台机器上运行程序, 而 VIP 用户可以在所有机器上运行程序。在这种情况下设计 Partition 就可能出现重叠的情况。

QoS, 即 Quality of Service (服务质量) 指的是用户作业程序的附加属性。顾名思义, 不同的服务质量决定了用户在集群上运行程序的体验。相信用过未名一号的同学已经有体会: 此集群比较拥挤, 交上去的作业基本上都要等待很久才能运行。如果导师给你三天时间来更新结果, 结果你花了两天的时间在排队上, 到头来肯定**被骂**。那么如何改善这种糟糕的体验呢? 答案就是更改 QoS。当然, 这需要付出一定的代价——在高优先级的 QoS 上运行程序一般要花费正常的两倍以上价钱! 但这在土豪面前根本不是问题, 因为**有钱是真的可以为所欲为的**。

### 6.3 使用 SLURM 调度系统运行程序

概括来说, 使用 SLURM 调度系统运行程序主要分为四步。

1. 准备程序和数据。将程序所需要的数据和程序上传到集群上。
2. 准备作业脚本。作业脚本的作用是告诉调度系统你的作业属性, 例如申请的资源数量, 运行时间, 任务名字等。
3. 提交作业。使用 SLURM 的 `sbatch` 命令把任务提交。
4. 查看作业状态, 验收结果。提交作业后可用 `squeue` 查看作业运行情况。作业结束后可查看标准输出或者作业程序生成的文件。

以下分别介绍这四个步骤。

#### 6.3.1 准备程序和数据

程序和数据在笔记本上, 如何上传到集群中呢? 使用 Linux/MacOS 的用户可以直接使用远程文件拷贝命令 `scp`。这个是基于 SSH 的命令, 随 SSH 自动安装。这个命令的基本格式为

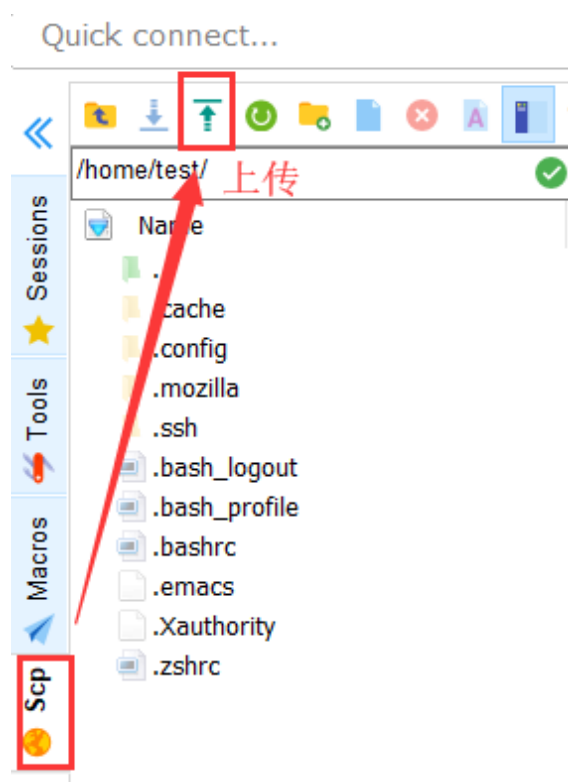
```
1 scp [OPTION]... [[user@]host1:]file1 ... [[user@]host2:]file2
```

在这里注意到 scp 的 SOURCE 和 DEST 均可以为远程服务器或本地，即这个命令的拷贝是双向的，上传数据和从服务器下载数据都是 scp。例如

```
1 scp file1 user@mu2.davidandjack.cn:           # 将 file1 上传到远程目
   录 HOME 下
2 scp -r folder1 user@mu2.davidandjack.cn:       # 将 folder1 目录上传到
   远程目录 HOME 下
3 scp user@mu2.davidandjack.cn:file1 .           # 将远程服务器 HOME 下
   的 file1 下载到本地当前目录
4 scp -r user@mu2.davidandjack.cn:folder1 .      # 将远程服务器 HOME 下
   的目录 folder1 下载到本地当前目录
```

需要注意的是，远程服务器的地址和文件之间用冒号：分隔，不要漏写其中的冒号。

使用 Windows 的用户可以直接利用 MobaXterm 的上传功能进行拖拽上传。



### 6.3.2 准备作业脚本

作业脚本其实是一个文本文件，里面叙述了作业的属性，例如作业申请的资源，占用时间，作业内容。以下是 SLURM 脚本的一个示例。

```
1 #!/bin/bash
2 #SBATCH -J name           # 任务名
3 #SBATCH -p gpu            # 分区名，可为 cpu 或 gpu
```

```

4 #SBATCH -N 1                # 节点数
5 #SBATCH --ntasks-per-node=1 # 单节点 tasks (进程)
6 #SBATCH --cpus-per-task=1   # 单进程 CPU 核心数 (也可写成 -c 1)
7 #SBATCH -o output.log       # 标准输出文件
8 #SBATCH -t 1-00:00:00       # 运行最长时间
9 #SBATCH --gres=gpu:1         # 单节点申请的 GPU 卡数
10                             # 只能在 -p 为 gpu 时填写
11
12 # 使用 environment modules 添加必要的模块
13 module add anaconda
14
15 python hello.py

```

这个文件中，以 `#SBATCH` 开头的是 SLURM 的配置区域，上面的注释中叙述了如何填写各个字段。之后的部分是告诉 SLURM 调度器要运行什么程序。

下面这个实例演示了如何提交一个 MPI/OpenMP 混合程序。

```

1 #!/bin/bash
2 #SBATCH -J hybrid
3 #SBATCH -p cpu
4 #SBATCH -t 5:00
5 #SBATCH -N 2
6 #SBATCH --ntasks-per-node=1
7 #SBATCH --cpus-per-task=2
8 #SBATCH -o hybrid.log
9
10
11 module add mpich/3.2.1
12
13 export OMP_NUM_THREADS=2
14 mpiexec ./hybrid

```

在这里注意，我们特地没有给 `mpiexec` 程序写上启动参数 `-n 2`，程序运行的结果如下

```

1 This is process 1 of 2 on host cu02, I have 2 threads.
2 This is process 0 of 2 on host cu01, I have 2 threads.

```

这好像挺符合我们的预期。实际上，集群中的 MPICH 和 SLURM 已经经过一定的整合，在 `sbatch` 模式下 `mpiexec` 可以正确识别 SLURM 脚本中的节点数和每个节点的进程数，根据这些信息自动分配资源并启动 MPI 进程。而 OpenMP 的线程数则不由 `mpiexec` 控制，它是由变量 `OMP_NUM_THREADS` 进行指定的。

严格来讲，用户并没有直接填写申请的资源数，而是通过指定 task 数（进程数）等信息来获取相应资源。并且这些参数和任务实际开的进程/线程数没有什么必然联系。就算只填写了 `--cpus-per-task=1`，用户仍然可以开启很多 MPI 进程或者 OpenMP 线程，只不过运行时



的效率通常比较低。因此，写在 SLURM 脚本上的这些参数（尤其是 `--ntasks-per-node` 和 `--cpus-per-task`）在某种程度上来讲只是一个**参考值**，程序实际开多少个进程是由用户决定的。详细的讨论我们放到第 6.6 节进行。

### 6.3.3 提交作业

写好脚本并检查无误后就可以提交作业了。提交命令为 `sbatch`。

```
1 sbatch run.slurm
```

以上命令是提交 `run.slurm` 的脚本。提交成功后将会显示该任务的 ID。

注意，作业提交成功后，初始的运行时目录为**提交时所在的目录**而并非 **SLURM 脚本所在的目录**。例如在 `/home/liuhy/src` 目录下有名为 `run.slurm` 的脚本，而用户在 `/home/liuhy` 目录下执行

```
1 sbatch src/run.slurm
```

程序实际运行时的所在初始目录为 `/home/liuhy`。

### 6.3.4 查看任务和验收结果

使用 `squeue` 可查看当前正在运行的作业。其实也可以查看排队中的作业，所有人的作业都能看到。程序运行完毕后，屏幕上的输出会被重定向到 `#SBATCH -o` 所指定的文件中。注意，程序中内存中的变量需要**手动保存**，如果不保存这些变量程序结束后将会丢失。程序画的图也要保存起来。

## 6.4 提交交互式任务

上面的小节实际介绍了提交批处理 (Batch) 任务的情况，而有的时候我们需要在集群上进行快速调试，在这个时候使用批处理任务就不是很合适。为了满足这种需求，SLURM 为我们提供了另一方便的运行程序模式：交互式。

使用交互式模式运行程序需要用到 `salloc` 命令。这个命令的实际作用是：在集群申请资源，而后借助申请到的资源运行程序，在程序结束时自动释放资源。一个具体的例子如下：

```
1 [liuhy@mu01 mpi-ex1]$ salloc -N 2 --ntasks-per-node=1 mpiexec ./
   hello
2 salloc: Granted job allocation 9724
3 salloc: Waiting for resource configuration
4 salloc: Nodes cu[01-02] are ready for job
5 From process 0 out of 2, Hello World!
6 That's all, folks! Time: 0.000047 sec
7 From process 1 out of 2, Hello World!
8 salloc: Relinquishing job allocation 9724
```

从这个例子可以看出，申请，运行，释放都是在一个命令中连续完成的，且命令确实是在计算节点中运行成功了。

当不给任何参数时, `salloc` 将会启动一个 `bash` 作为运行的程序, 由于 `bash` 是要手动退出的, 因此用户可以更灵活地运行自己的程序, 甚至还可以直接使用 `SSH` 登录到计算节点。当自己程序全部执行完毕后退出现这个 `bash`, 退出的一瞬间资源会被释放。**注意: 很多同学喜欢这种用法, 但他们忘性比较大, 经常以为自己程序运行完了之后资源就会自动释放了, 这是不对的。释放资源的时间点在退出 `bash` 的那个瞬间。今后大家不要因为这个坏习惯无端占用资源甚至是浪费机时(钱)。**

值得注意的是, 在 `salloc` 分配的 `bash` 中运行 `MPI` 程序, 申请的资源也会被自动识别。

```
1 [liuhy@mu01 mpi-ex1]$ salloc -N 2
2 salloc: Granted job allocation 9724
3 salloc: Waiting for resource configuration
4 salloc: Nodes cu[01-02] are ready for job
5
6 [liuhy@mu01 mpi-ex1]$ mpiexec ./hello
7 From process 0 out of 2, Hello World!
8 That's all, folks! Time: 0.000047 sec
9 From process 1 out of 2, Hello World!
```

但在 `salloc` 中运行非 `MPI` 程序(例如系统命令, 普通二进制程序, 或 `python` 等), 则依然会在主节点运行, 不会自动跳转到申请的节点。

```
1 [liuhy@mu01 ~]$ salloc hostname
2 salloc: .....
3 salloc: Nodes cu01 are ready for job
4 mu01 # <- 没有到 cu01 中
5
6 [liuhy@mu01 ~]$ salloc
7 salloc: .....
8 salloc: Nodes cu01 are ready for job
9 [liuhy@mu01 ~]$ hostname
10 mu01 # <- 同样没啥用
```

此时必须使用 `srun` 命令来运行程序, 这样才会使用分配到的节点。

```
1 [liuhy@mu01 ~]$ salloc
2 salloc: .....
3 salloc: Nodes cu01 are ready for job
4 [liuhy@mu01 ~]$ srun hostname
5 cu01
```

或者更直接的方法: 直接使用 `srun` 命令(不经过 `salloc`)。

```
1 [liuhy@mu01 ~]$ srun hostname
2 cu01
```

这里可以认为 `srun` 是在计算节点上运行任务（必要的时候会先申请资源再执行，此时会隐式地生成一个 SLURM 任务）。同样用户也可给 `srun` 提供申请资源相关的参数，具体请参考 `man srun`。

## 6.5 常用 SLURM 命令

除运行程序外，我们还经常需要使用一些命令来在 SLURM 调度系统中进行查询或修改。例如：查看集群目前节点运行情况，查看作业已经运行时间/剩余时间，亦或是计算一半的时候突然不想算了。常用的 SLURM 命令如下：

- `sinfo` 查看目前所有节点分区情况。其中 STATE 标为 `alloc` 表示已经没有剩余资源；标为 `idle` 的表示空闲；标为 `mix` 的表示有人在占用但是有剩余资源；标为 `down` 表示节点无响应（可能是故障）。此命令还支持长列表逐个节点的显示方式，即 `sinfo -lN`。
- `sbatch` 提交任务。后面跟着 SLURM 脚本作为参数。如果没有参数则直接从标准输入读参数。此命令后面也可以跟选项来指定申请的资源数，这些选项的含义和 SLURM 脚本中的完全一致。当命令行选项和 SLURM 脚本 **同时指定**了某资源时，命令行选项的优先级更高。
- `squeue` 检查队列情况。只能列出正在排队或正在运行的程序。可以看到作业的拥有者，运行状态，运行时间，占用节点数。处于运行状态的程序可查看所在计算节点，处于排队的程序可查看没被运行的原因。
- `scancel` 取消作业。用法 `scancel JOBID` 或 `scancel -u USERNAME`，可取消某特定 ID 的作业或某个用户的全部作业。当然，非管理员不能取消别人的作业。
- `sacctmgr` SLURM 账号管理。这条命令可以查看 SLURM 账号信息，QoS 限制情况。例如 `sacctmgr show qos`。
- `scontrol` 查看 SLURM 配置和状态信息。这条命令使用方法非常多，就算是普通用户也可以使用其中的查询和部分修改功能。例如 `scontrol show job JOBID` 可以查询运行中作业的详细信息；`scontrol show config` 可以查看全局 SLURM 配置信息。当然不小心将任务属性写错时，如果任务还没有运行，甚至可以通过 `scontrol update jobid=JOBID ...` 来修改作业的属性。在这里我们只列出部分的功能，对详细用法不作讲解。

## 6.6 SLURM 确定分配资源的方式

第 6.3 小节中提到，控制申请资源的方式是在 SLURM 脚本或命令行中使用 `--ntasks-per-node`，`--cpus-per-task` 等参数。在我们的印象中，集群运行的硬件资源应该是 CPU，内存，GPU 这些东西，在前面提到的选项中也是有这些名词。但这些选项中还多次出现“tasks”这个词，将其理解成一种资源显然有点牵强。那么这个 tasks 究竟是何作用，它跟申请的资源有什么关系？

如果仔细查看 SLURM 帮助文档的话，和资源申请的参数至少包含：

- `--nodes/-N`：节点数

- `--ntasks/-n`: 任务总数
- `--ntasks-per-node`: 每个节点的任务数 (可能是精确值, 也可能是最大值)
- `--ntasks-per-socket`: 每个 NUMA 节点的任务数
- `--ntasks-per-core`: 每个核的任务数
- `--cpus-per-task/-c`: 每个任务占用的 CPU 数
- `--gres`: 每个节点的一般资源数 (Gres, 例如 GPU)  
(SLURM 19.05+ 增加了以下选项)
- `--gpus`: GPU 总数
- `--gpus-per-node`: 每个节点的 GPU 数 (和 `--gres` 基本等价)
- `--gpus-per-task`: 每个任务的 GPU 数
- `--gpus-per-socket`: 每个 NUMA 节点的 GPU 数
- `--cpus-per-gpu`: 每个 GPU 占用的 CPU 数
- ...

这时候的你可能会大喊卧槽了, 这么多选项看着更晕了。难道就没办法愉快地申请资源了么? 实际上, 从中作怪的就是 `task` 这个概念。如果把它理解清楚了, 那么别的问题自然迎刃而解。

“task” 的翻译是“任务”, 但在这里更多的是指系统中的进程 (可以理解为一个程序)。在不使用 MPI 的情况下, 运行一个程序就等于有一个 task。资源的申请就是围绕 task 展开的, 简单来说, SLURM 会根据你的 task 数量计算出应该分配多少资源给你, 并设置好这些资源的分布。如果你的程序是 MPI 程序, 那么它还会帮你绑定每个进程和资源的位置关系。

为了简单现在先不考虑 GPU 的情形。我们的问题是: SLURM 如何将 CPU 和任务数对应起来的? 前面的脚本使用了 `-N`, `--ntasks-per-node`, `--cpus-per-task` 三个参数, 之后 SLURM 调度器就可以判断出申请的资源。这三个参数的含义分别为“节点数”、“每个节点任务数”, “每个任务的 CPU 数”, 那么结论很显然: SLURM 会分配指定的节点数, 且每个节点上划出来“任务数  $\times$  单任务 CPU 数”个 CPU 用于完成计算。这个方式跟我们的直觉比较符合。但我们完全可以换另外一种方式来指定: 例如

```
1 sbatch --ntasks=32 --ntasks-per-node=16 ...
```

注意这个例子中没有使用 `-N` 参数, 但依然能成功执行。根据这两个参数的含义, SLURM 会计算得知: 总共要开 32 个任务, 每个节点是 16 个, 因此要申请两个节点。而每个节点上是 16 个任务, 所以要划出来 16 个 CPU 完成计算 (默认不写 `-c` 时缺省是 1)。或者, 我们甚至可以这样指定:

```
1 sbatch --ntasks=32 ...
```

如果这样给命令, SLURM 会得知: 总共要开 32 个任务。由于没有额外信息, 这 32 个任务可以被放到任何节点上执行, 那么 SLURM 只需要在所有节点中凑出来 32 个 CPU 就行了 (缺

省 `-c` 的值是 1)。这种行为在某些场合下完全不符合我们的预期。例如如果我们的程序不是分布式并行的（只能放在一个节点上运行），那么其它节点的 CPU 就没法利用。当然，如果程序本身就是分布式并行的，这样做没有任何问题。

读到这里你可能会问另一个问题了，如果我同时指定 `-N`, `--ntasks`, 以及 `--ntasks-per-node` 会发生什么？还真是会找事啊。如果全部指定，则 SLURM 会优先遵从 `--ntasks` 和 `-N` 的设置，并将 `--ntasks-per-node` 视为每个节点任务数**最大值**。当然，如果这样还是不合法的话，SLURM 只好告诉你这样提交任务不行，建议你检查参数重新提交。

有了前面的解读，我们现在讨论一下 `--ntasks-per-socket` 这个参数。第一个问题：什么是 socket？一般来讲，常见的服务器可能有多颗 CPU（注意：“颗”和“核”不同，一颗 CPU 可能有很多个核），那么每一颗 CPU 就是一个 socket（也可以说是一个 NUMA 节点）。例如，数院集群的每一个刀片就有两颗 CPU，每个拥有 12 个核心，大家看到每个节点有 24 个核其实是一个总和。需要注意的是，虽然从外面看这两颗 CPU 是一个整体，但是运行程序的时候特别要注意 NUMA 问题。首先，不同 CPU 和内存的亲 and 性不同，访问另一个 CPU 附近的内存要比访问自身附近内存要慢一些；其次，如果程序的线程开在不同 CPU 上，线程之间的协同也会产生影响。我们运行并行程序时，尽量应该选择在同一个 NUMA 节点上开线程，这也就是 `--ntasks-per-socket` 的作用。例如

```
1 sbatch --ntasks-per-socket=12 ...
```

这个命令指定每个 socket 上的任务数是 12，SLURM 在分配资源时就会注意到这个设定，从而会分配同一个 NUMA 节点的 12 个 CPU 核心。如果换成 `--ntasks-per-node`，那么 SLURM 只能保证这些核心处于同一节点，而无法保证它们来自同一颗 CPU。

当然，`--ntasks-per-node` 和 `--ntasks-per-socket` 不能混用，因为后者是以 socket 这个层面上设置任务的分布的。但是 `--ntasks-per-socket` 可以和 `-N` 一起用，用于指定 NUMA 节点和物理节点的关系。例如

```
1 # 每个 socket 有一个任务，总共有 4 个任务，申请两个节点
2 # 如果假设一个物理节点有两个 NUMA 节点
3 # 那么结果就是 SLURM 会使用两个节点/四个 NUMA 节点运行任务
4 sbatch --ntasks=4 --ntasks-per-socket=1 -N 2 ...
5
6 # 不指定 -N 时，SLURM 只能保证最后会用到 4 个 socket
7 # 但不能保证这四个 socket 的来源，有可能其中两个 socket 来自
8 # 同一节点，另外两个 socket 分别来自两个不同节点
9 sbatch --ntasks=4 --ntasks-per-socket=1
```

最后我们考虑使用 GPU 的情形。经测试，SLURM 20.11.5 对部分 GPU 资源的处理和分配仍有一些问题，但处理的原则和上述 CPU 情形相同。例如

```
1 sbatch --gpus=2 --gpus-per-node=1 --ntasks-per-node=4 ...
```

该例子中由于设置了 `--gpus=2`, `--gpus-per-node=1`, SLURM 会推断出任务需要分配两个有 GPU 的节点。而 `--ntasks-per-node=4` 表示每个节点要开四个任务，根据默认分配策略，每个 task 会分配 1 个 CPU 核心。因此最终 SLURM 会分配两个节点，每个节点分配一个

GPU 和 4 个 CPU 核心。这个例子的核心在于，使用 GPU 相关参数隐式设置了总共需要的节点数。因此所需的 CPU 核心数也可推断出来。

前文提到，使用 `--ntasks`，`--ntasks-per-node` 也可以隐式指定节点。因此设置参数时不能出现冲突的选项，否则 SLURM 会报错。和 `--ntasks-per-node` 不同，`--gpus-per-node` 不能表示最大值，因此它必须能够整除 `--gpus`。

GPU 和 CPU 也有亲和性的问题，从一个 NUMA 节点访问另一个 NUMA 节点的 GPU 会显著地慢。然而上面的选项并没有考虑这个问题，例如使用 `--gpus-per-task` 和 `--ntasks` 时，SLURM 只确保数量正确（同一节点上 tasks 和 gpus 的比例正确），不会保证亲和性。要实现亲和性必须使用 `--gres-flags=enforce-binding` 参数（需要服务端支持），这里不做讨论。

## 7 思考题

### 7.1 理解动态链接

考虑如下代码片段：

```
1  /* 以下为 foo.c 的内容 */
2  #include <stdio.h>
3  void foo(){
4      printf("Hello world.\n");
5  }
6
7  /* 以下为 bar.c 的内容 */
8  #include <stdio.h>
9  void foo(int s){
10     printf("Hello world, %d\n", s);
11 }
12
13 /* 以下为 main.c 的内容 */
14 void foo();
15
16 int main(){
17     foo();
18     return 0;
19 }
```

1. 使用 `gcc` 将 `foo.c` 和 `bar.c` 分别编译成共享库 `libfoo.so` 和 `libbar.so`。
2. 编译 `main.c`，在指定链接选项时分别指定上述两个库，两种情况下编译结果如何？运行结果如何？
3. 若在编译 `main.c` 时使用 `libbar.so` 进行链接，但运行之前使用 `libfoo.so` 覆盖 `libbar.so`（覆盖后的文件名仍然是 `libbar.so`），那么运行情况又如何？

4. 试着解释你观察到的现象，并思考为什么在链接的时候仍然需要指定动态库。