常见 Linux 配置/使用问题

—— Some Miscellaneous Linux Configuration/Usage Issues

吴擎宇

2026-03-21

中国科学技术大学

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

1.1 端口转发

该部分主要参考 ustclug 201 文档[1]。

利用 ssh 进行端口转发是一个很常见并且非常实用的功能，比如：

- 你可以把远程主机开在 `localhost` 的服务（e.g. nginx/apache）重定向到本地，这样你无需向外界暴露端口即可测试自己的服务
- 你可以把本地的代理服务，比如开在 `localhost:7890` 的某某软件，重定向到远程主机上，然后再在 `.bashrc` 之类的文件里设置环境变量 `http_proxy` 和 `https_proxy`，这样就可以在远程主机上使用本机的网络代理了
- 你单纯只是想把某台机器作为一个网络流量代理服务器使用（e.g. Vlab），作为跳板用来访问一些内网服务（e.g. 不能对外公开的组内的服务器）

[1] https://201.ustclug.org/dev/ssh/#port-forwarding

1.1 端口转发

为方便讲解，我将使用具体的例子来说明 ssh 端口转发的三种使用方法：

1.1.1 本地端口转发（Local port forwarding）

1.1.1.1 网页预览

我有一个基于 nginx 的静态网页开放端口在 `localhost:80`，因为是测试，所以不想暴露在公网，但是我需要本地预览看看效果：

```
1 ssh -L 8080:localhost:80 user@remote-host
```

这条指令的含义是，在本机开一个 8080 端口，所有发往本机 8080 端口的请求，都会被**转发**到远程服务器，通过远程服务器去访问 `localhost:80`，也就是 nginx 的服务。此时，你只需要在本地浏览器里访问 `http://localhost:8080` 就可以访问 nginx 的服务了。

1.1 端口转发

1.1.1.2 网络通登录

有时候,你可能需要为校内的某台服务器临时登录一下网络通以便访问外网,这时候你也可以用到本地端口转发:

```
1 ssh -L 8081:wlt.ustc.edu.cn:80 user@remote-host
```

这条指令的含义是，在本机开一个 8081 端口，所有发往本机 8081 端口的请求，都会被**转发**到远程服务器，通过远程服务器去访问 `wlt.ustc.edu.cn:80`，也就是网络通的服务。

此时，你只需要访问 `http://localhost:8081` 就可以访问到网络通的页面，给远程的服务器登录网络通了。

1.1 端口转发

1.1.2 远程端口转发（Remote port forwarding）

1.1.2.1 本地代理透传

很多时候，远程的服务器不会像本地一样安装代理软件，这就导致在远程服务器上很难访问 github、huggingface 之类的网站，这时候你可以使用远程端口转发把远程的流量转发到本地的代理软件上：

```
ssh -R 7890:localhost:7890 user@remote-host
```

这条指令的含义是，在远程服务器上开一个 7890 端口，所有发往远程服务器 7890 端口的请求，都会被**转发**到本机，通过本机去访问 `localhost:7890`，也就是代理软件的端口。

此时，你只需要设置 `http_proxy` 和 `https_proxy` 就可以临时使用本机代理访问网站了。

1.1 端口转发

1.1.3 动态端口转发（Dynamic port forwarding）

1.1.3.1 作为跳板

有时候，我们身在校外，没法连接到校内网络，但是组内的服务器又被限制在校内访问，这时候我们就可以利用 Vlab 作为一个跳板，将流量先转发到 Vlab 上，然后再由 Vlab 转发到组内服务器上：

```
1 ssh -D 1080 user@vlab.ustc.edu.cn
```

这条指令的含义是，在本机开一个 1080 端口，用作 SOCKS5 代理，所有发往本机 1080 端口的代理请求，都会被**转发**到 Vlab 上，由 Vlab 去访问目标服务器。

与本地端口转发不同的是，你无需指定目标服务器的地址和端口，即所谓**动态**端口转发。

1.1 端口转发

如果你觉得每次写转发的命令很麻烦，你可以在 `~/.ssh/config` 文件中指定转发的配置，比如：

```
Host example
  LocalForward 8080 localhost:80
  LocalForward 8081 wlt.ustc.edu.cn:80
  RemoteForward 8080 localhost:80
  DynamicForward 1080
```

以上是之前的例子写成 ssh config 的样子，这样你每次只需要执行 `ssh example` 就可以自动应用这些转发配置了。

1.2 代理

该部分主要参考 ustclug 201 文档[1]。

有时候，我们在 git clone 一些仓库的时候，可能会遇到不能使用 https 协议访问仓库，只能通过 ssh 协议访问仓库的问题，但是国内对于 github.com 的访问往往不太稳定，这时候我们就可以给 ssh 配置代理来解决这个问题。在 `~/.ssh/config` 文件里添加以下内容：

```
Host github.com
    HostName github.com
    User git
    ProxyCommand nc -X 5 -x 127.0.0.1:7891 %h %p
```

注意，请安装 OpenBSD 版本的 netcat 工具，比如在 Debian 系统中包名为 `netcat-openbsd`。

[1] https://201.ustclug.org/dev/ssh/#proxy

1.3 跳板

该部分主要参考 ustclug 201 文档[1]。

有时候，我们身在校外，没法连接到校内网络，但是组内的服务器又被限制在校内访问，这时候我们就可以利用 Vlab 作为一个跳板，先连接到 Vlab 上，然后再由 Vlab 转发到组内服务器上。

首先，我们可以给 Vlab 先写好配置：

```
Host vlab
  HostName vlab.ustc.edu.cn
  User vlab
  IdentityFile ~/.ssh/vlab-vm12345.pem
```

[1] https://201.ustclug.org/dev/ssh/#jump-host

1.3 跳板

此时，你可以通过一条简单的指令就能连接上 Vlab 了：

```
ssh vlab
```

然后，再在 `~/.ssh/config` 文件里添加需要通过 Vlab 跳板访问的服务器配置：

```
Host lab-server
  HostName <your-lab-server-ip>
  User <your-username>
  ProxyJump vlab
```

此时，你就可以通过一条简单的指令连接上组内的校内服务器了：

```
ssh lab-server
```

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

2.1 安装

一提到安装 Docker，大部分人可能首先想到的是去 Docker 官网找教程下载 Docker Engine (Docker CE)，但是官网上的教程比较繁琐，其实 USTC Mirror Help[1] 里就包含了简单的安装 Docker CE 的指令，直接复制粘贴运行就可以了：

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo DOWNLOAD_URL=https://mirrors.ustc.edu.cn/docker-ce sh get-docker.sh
```

安装完 Docker 后如果想要某个用户能够直接使用 docker 命令而不需要 sudo，请使用 `usermod` 命令将该用户添加到 `docker` 组，这样就不用每次运行 docker 都得 sudo 了：

```
sudo usermod -aG docker $USER
```

[1] https://mirrors.ustc.edu.cn/help/docker-ce.html

2.2 拉取镜像

一个使用 Docker 最为常见的问题就是，刚安装完 Docker 尝试运行一下 hello-world:

```
docker run --rm hello-world
```

结果却提示：

```
Unable to find image 'hello-world:latest' locally
docker: Error response from daemon: Get "https://registry-1.docker.io/v2/": net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

Run 'docker run --help' for more information
```

~~What's wrong?~~

2.2 拉取镜像

解决这个问题需要分情况讨论：

2.2.1 admin/sudoer

如果你是系统管理员或者具有 sudo 权限的用户：

2.2.1.1 配置镜像站

创建并修改 `/etc/docker/daemon.json` 文件，添加以下内容：

```
{
  "registry-mirrors": ["https://<mirror-for-docker-hub>"]
}
```

需要注意的是，由于不可抗力，国内各大高校的 docker hub 镜像站已停止服务，你只能通过搜索引擎在网络上找到一些第三方的镜像站地址，使用时请务必注意安全问题。

2.2 拉取镜像

除了阿里云、华为云这类国内大厂，目前~~活得比较好的~~一个第三方镜像站是 DaoCloud[1]。该镜像源采取白名单机制，只有列入白名单的镜像才能被加速访问。使用方法很简单，直接在你要拉取的仓库前面加上 `m.daocloud.io` 前缀就行了，比如：

```
docker pull m.daocloud.io/docker.io/library/hello-world:latest
docker pull m.daocloud.io/code.forgejo.org/forgejo/forgejo:latest
```

除此之外，你也可以考虑自己搭建一个本地镜像缓存，仅用于内部使用，具体配置请见 https://mirrors.ustc.edu.cn/help/dockerhub.html#self-host

[1] https://m.daocloud.io

2.2 拉取镜像

2.2.1.2 配置代理

如果你的服务器上运行了网络代理软件，常见的比如说，位于 7890 端口的某某软件，那么你同样可以修改 /etc/docker/daemon.json 文件，添加以下内容：

```
{
  "proxies": {
    "http-proxy": "http://127.0.0.1:7890",
    "https-proxy": "http://127.0.0.1:7890"
  }
}
```

这需要你学会网络代理软件的正确配置方式，比如不要忘记添加 allow-lan: false 选项，或者配置到内网地址，否则你的代理端口会被暴露在校园网内，导致安全问题。

2.2 拉取镜像

2.2.1.3 WSL 代理

如果你是 WSL 用户，想要在 WSL 内使用宿主机的网络代理，那么最简单的方法就是打开宿主机的网络代理软件的 tun mode（虚拟网卡），它会全局代理宿主机的网络流量，包括 WSL 内的流量，这样就不需要在 WSL 内进行任何配置了。

2.2.1.4 应用配置

修改完 `/etc/docker/daemon.json` 文件后，需要执行以下命令让 Docker 应用新的配置：

```
1 sudo systemctl restart docker
```

2.2 拉取镜像

2.2.2 normal user

如果你是普通用户，无法修改 `daemon.json` 文件，或者觉得以上方法过于麻烦：

2.2.2.1 本地拉取上传

Docker 镜像是可以导出成一个 tar 包的，你可以首先在一台能够正常访问 Docker Hub 的机器上拉取你需要的镜像，然后使用 `docker save` 命令将其导出成一个 tar 包：

```
1 docker pull <image-name>
2 docker save <image-name> -o <image-name>.tar
```

然后将这个 tar 包传输到你的机器上：

```
1 scp <image-name>.tar user@your-machine:/path/to/save
```

2.2 拉取镜像

或者使用更快一点的 rsync，如果服务器上有安装：

```
rsync -zvhP <image-name>.tar user@your-machine:/path/to/save
```

最后使用 `docker load` 命令将其导入到 Docker 中：

```
docker load -i <image-name>.tar
```

本地拉取就很简单了，比如

- Windows 用户可以安装 Docker Desktop，然后在 Settings - Resources - Proxies 里设置网络代理，之后就可以直接在 Windows 上拉取镜像了。
- Linux/macOS 用户肯定有 root 权限，直接按照之前的方法配置镜像或者代理就可以拉取镜像了。

2.2 拉取镜像

2.2.2.2 Docker Rootless

如果你的镜像不需要使用 root 权限，比如使用 `--privileged` 之类的选项，那么你也可以考虑使用 rootless 模式来运行 Docker。

通常来说，你可以安装好 Docker 的机器上直接执行以下指令来安装 Docker Rootless：

```
1 dockerd-rootless-setuptool.sh install
```

如果报错，可能需要添加 `--force` 选项强制安装，但是这不会影响全局的 Docker，同时还需按照输出提示配置 `$DOCKER_HOST` 环境变量，请仔细阅读。

如要卸载 Docker Rootless 也很方便：

```
1 dockerd-rootless-setuptool.sh uninstall
```

2.2 拉取镜像

不同于 Docker Rootful，Docker Rootless 默认无法直接访问主机 localhost 上的服务，因此我们需要给装好的 Docker Rootless 添加一个环境变量去除这个限制：

```
mkdir -p ~/.config/systemd/user/docker.service.d
cat > ~/.config/systemd/user/docker.service.d/override.conf << 'EOF'
[Service]
Environment=DOCKERD_ROOTLESS_ROOTLESSKIT_DISABLE_HOST_LOOPBACK=false
EOF
systemctl --user daemon-reload
systemctl --user restart docker
```

也就是添加一个 `DOCKERD_ROOTLESS_ROOTLESSKIT_DISABLE_HOST_LOOPBACK=false` 的环境变量来让 Docker Rootless 禁止阻止访问主机 localhost 上的服务。

2.2 拉取镜像

接下来，我们可以通过修改 `~/.config/docker/daemon.json` 文件来配置镜像或者添加代理，配置方法和之前的 Docker Rootful 基本上是一样的，但是额外注意的是，你需要将所有的 `127.0.0.1` 替换成 `10.0.2.2`，比如：

```
{
  "proxies": {
    "http-proxy": "http://10.0.2.2:7890",
    "https-proxy": "http://10.0.2.2:7890",
  }
}
```

这是因为在 Docker Rootless 模式下，Docker 守护进程运行在一个非特权的 namespace 里，这个 namespace 和主机的 namespace 是隔离的，因此需要修改 localhost 的地址才能让 Docker Rootless 访问到主机上的服务。

2.2 拉取镜像

最后，修改完 `~/.config/docker/daemon.json` 文件后，需要执行以下命令让 Docker Rootless 应用新的配置：

```
systemctl --user restart docker
```

2.2.3 Podman

如果你觉得 Docker Rootless 配置代理怎么这么麻烦，那就对了，是时候使用 Podman 了，默认 Rootless 模式，只需要设置 `http_proxy` 和 `https_proxy` 环境变量就可以走代理，而且 Podman CLI 兼容 Docker CLI，你可以直接像这样直接把 docker 命令替换成 podman：

```
alias docker=podman
```

不过安装 Podman 需要 sudo 权限，需要和系统管理员沟通一下。

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

3.1 安装驱动

该部分主要参考 ustclug 201 文档[1]。

- 如果你是 wsl 用户，想要在 wsl 内安装 Nvidia 驱动，那么你不需要在 wsl 里做任何事情，而是应该在 Windows 上安装 Nvidia 的官方驱动，一旦安装完成，wsl 内的 Nvidia 驱动也就安装好了。
- 如果是普通的 Linux 用户，那么你应该尽可能通过**包管理器**安装 Nvidia 驱动，而不是通过 Nvidia 官网下载的 .run 文件来安装，~~否则的你的系统很可能会当场/延迟爆炸~~。
  - 通过包管理器也有两种方式，一种就是官方自行打包的驱动包，不需要添加源；一种是 Nvidia 自己打包的驱动包，位于 CUDA-Downloads[2]，请通过添加源的方式安装，**不要下载运行 .run 文件**。

[1] https://201.ustclug.org/advanced/cuda/#nvidia-driver
[2] https://developer.nvidia.com/CUDA-downloads

3.1 安装驱动

比如，在 Debian/Ubuntu 系统中，你可以直接安装 `nvidia-driver` 包：

```
sudo apt update
sudo apt install nvidia-driver
```

如果你已经通过 .run 文件安装了 Nvidia 驱动，还没有或者已经造成了驱动问题，那么强烈建议你通过之前的 .run 文件卸载掉之前安装的驱动：

```
sudo ./NVIDIA-Linux-x86_64-<version>.run --uninstall
```

然后通过包管理器重新安装 Nvidia 驱动。

3.2 安装 CUDA

该部分主要参考 ustclug 201 文档[1]。

Nvidia 驱动是内核模块（i.e. nvidia.ko）的安装，而 CUDA 是用户层库（e.g. libcuda.so）的安装，二者是相互独立的，因此你可以在安装了 Nvidia 驱动的系统上安装多个版本的 CUDA，或者在没有安装 Nvidia 驱动的系统上安装 CUDA（虽然没什么意义）。

CUDA 的安装同样推荐通过**包管理器**来安装，不需要添加源；注意的是，如果你之前通过添加 Nvidia 源安装了 Nvidia 驱动，那么你可能已经顺便安装好 CUDA 了。

除此之外，你也可以使用 conda 安装 CUDA，就像在系统里安装了一样，但是**单环境有效**；或者你可以通过 pip 安装 CUDA，但是这个 CUDA **只能在 Python 里使用**，无法被其他程序使用。

[1] https://201.ustclug.org/advanced/cuda/#install-cuda

3.2 安装 CUDA

比如在 Debian/Ubuntu 系统中，你可以直接安装官方打包的 CUDA 包：

```
sudo apt update
sudo apt install nvidia-cuda-toolkit nvidia-cuda-dev nvidia-visual-profiler
```

如果你需要在 Docker 里使用 CUDA，那么你需要添加源并安装 Nvidia Container Toolkit，具体可以参考 Help - USTC Mirror[1]。

[1] https://mirrors.ustc.edu.cn/help/libnvidia-container.html

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

4.1 虚拟环境

说到 Python，最让人印象深刻的可能就是 Python 地狱般的依赖管理了。如果所有 Python 包都放在同一个环境里，那么不同项目之间的依赖冲突就会变成一个大问题，venv 就是为了解决这个问题而诞生的。

- venv 是 Python 标准库自带的一个轻量级的虚拟环境管理工具，它可以让你在同一台机器上创建多个独立的 Python 环境，每个环境都有自己的 Python 解释器和包安装目录，这样就可以避免不同项目之间的依赖冲突了。
- 最近，uv 也成为了一个非常受欢迎的 Python 包管理工具，它的虚拟环境是基于 venv 的，通过 pyproject.toml 管理依赖，并且提供了更为简洁方便的命令行界面和更快的安装速度。
- 除了 venv，conda 也是一个非常流行的包和环境管理工具，但是 conda 不仅能用来管理 Python 包，还能管理其他语言的依赖。

4.2 venv

venv 的使用非常简单，首先你需要在项目目录下创建一个虚拟环境：

```
python -m venv .venv
```

这条命令会在当前目录下创建一个名为 `.venv` 的虚拟环境，接下来你需要激活这个虚拟环境：

```
source .venv/bin/activate
```

激活后，你就可以在这个虚拟环境里安装你需要的包了：

```
pip install <package-name>
```

4.3 uv

虽然 venv 已经足够满足大多数人的需求了，但是 uv 提供了更为现代化的使用体验，首先你需要安装 uv：

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

安装完成后，添加镜像，打开 `~/.config/uv/uv.toml` 写入：

```
[[index]]
url = "https://mirrors.ustc.edu.cn/pypi/simple"
default = true
```

4.3 uv

配置完成后，你可以使用 uv 来初始化项目：

```
1 uv init
```

这条指令会在当前目录初始化 git 仓库，创建 venv 虚拟环境，同时创建一个 pyproject.toml 文件来管理依赖，如果你只是想要通过 uv 创建一个虚拟环境，你可以使用以下指令：

```
1 uv venv
```

但是光创建 venv 还不够，你还需要给一个 pyproject.toml 文件让 uv 管理依赖，因此你需要运行一下指令：

```
1 uv init --bare
```

4.3 uv

此时，你可以通过 uv 来安装和管理依赖了，比如安装 PyTorch：

```
uv add torch torchvision
```

通过 uv 安装 Python 包的速度通常会比 pip 快很多，而且依赖可复现性更好，即使你把 .venv 目录直接删掉，只要 pyproject.toml 文件还在，你就可以通过 `uv sync` 重建出一样的虚拟环境。

最后如果你想要卸载某个包：

```
uv remove torch torchvision
```

4.4 conda

严格来说，conda 并不是专门为 Python 设计的包和环境管理工具，反而更像是 Docker，但是没有用到 cgroups 和 namespace 之类的内核特性，而是通过环境变量劫持（e.g. PATH）和动态链接库重定向（e.g. patchelf）等技术来实现环境隔离的效果的。

因此，conda 不仅能解决 Python 的依赖问题，还能解决一下更加底层的 C/C++ 库的依赖问题，比如一个常见的问题就是主机 glibc 版本过低，conda 能够通过劫持 glibc 的链接路径来让用户使用一个从 conda 源下载的更新版本的 glibc 来解决这个问题。

如果要安装 conda，推荐使用 Miniconda，它是 Anaconda 的精简版本，体积更小，但是保留核心的 conda 包管理器：

```
1 wget https://mirrors.ustc.edu.cn/anaconda/miniconda/Miniconda3-latest-
  Linux-x86_64.sh
2 bash Miniconda3-latest-Linux-x86_64.sh
```

4.4 conda

安装完成后，根据我校 Miniconda 对于 conda-forge 和 bioconda 源的镜像说明[1]配置镜像。

如果你需要修改 conda 默认创建环境和安装包的路径，可以使用以下命令：

```
conda config --add envs_dirs /path/to/conda/envs
conda config --add pkgs_dirs /path/to/conda/pkgs
```

配置完成后，你就可以使用 conda 来创建环境和安装/卸载包了：

```
conda create -n test python=3.13
conda activate test
conda install numpy scipy
conda deactivate
```

[1] https://mirrors.ustc.edu.cn/help/anaconda.html#miniconda

4.5 目录结构

使用 uv 创建的基于 venv 的虚拟环境的项目目录结构如下所示：

```
project/
├── .venv/
│   ├── bin/
│   │   └── python3
│   └── lib
│       └── python3.13/
│           └── site-packages/
│               └── (all python packages)
├── src/
└── pyproject.toml
```

其中：

- .venv/ 是 venv 创建的虚拟环境目录，里面包含了独立的 Python 解释器和包安装目录
- src/ 是项目的源代码目录
- pyproject.toml 是 uv 管理依赖的配置文件，里面记录了项目的依赖信息

特点：只能安装和管理 Python 包

4.5 目录结构

使用 conda 创建的虚拟环境的目录结构如下所示：

```
~/.conda (can be redirected)
├─ envs/
│   └─ <environment-name>
│       ├─ bin/
│       │   ├─ (all binary executables)
│       │   └─ (e.g. python, gcc)
│       └─ lib/
│           ├─ (all libraries, static or dynamic)
│           └─ (e.g. libssl.so, libcc1.so)
└─ pkgs/
    └─ (all conda packages ended with .conda)
```

其中：

- envs/ 是 conda 创建的所有虚拟环境的目录，每个环境都有一个独立的子目录，里面包含了该环境的所有二进制可执行文件和库文件
- pkgs/ 是 conda 下载的所有包的目录，每个包都是一个以 .conda 结尾的文件，里面包含了该包的所有文件和元数据

特点：可以管理任何类型的包

4.6 依赖问题

Python 的依赖问题向来非常复杂和困扰，当你在某个 repo 下配置环境出现依赖问题时，通常我推荐采用以下方法依次尝试解决：

1. 删除某个冲突的包，如果你发现实际上不需要
2. 重开一个新的环境，俗话说重装解决 99% 的问题
3. 重看 README 严格按照要求重配环境
4. 询问在同一个 repo 上 work 的同学怎么配出来的环境
5. 开个 issue 询问作者怎么配出来的环境
6. 清晰表述你所 work 的 repo，所使用的系统，配置的环境的步骤，给出尽可能详细的信息，在技术群里提问，也许有人有幸配成功了，皆大欢喜
7. ~~没招了，还是选择等死吧~~

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

5.1 进程

- top: 最**基本**的实时进程信息查看工具，支持按照 CPU、内存、时间等多种方式排序和过滤进程，但是界面比较简陋，功能也比较有限。
- htop: 最**常用**的实时进程信息查看工具，支持交互式操作，可以按照 CPU、内存、时间等多种方式排序和过滤进程，还可以显示进程树、线程信息等。
  - 常驻集内存：$\mathrm{RES} = \mathrm{USS} + \mathrm{SHR}$
  - 比例内存：$\mathrm{PSS} = \mathrm{USS} + \sum_{\mathrm{lib}} \frac{\mathrm{SHR}_{\mathrm{lib}}}{n_{\mathrm{lib}}}$（更合理的内存计算）
- btop: htop 的**增强**版，提供了更为美观和丰富的界面，支持更多的系统资源监控，比如网络、磁盘等。
- ps: 任一时刻下查看所有进程信息的工具，可以通过各种选项来显示进程的各种信息，比如 `ps aux` 和 `ps -ef` 都可以显示所有进程的详细信息，搭配 grep、awk 等工具可以过滤出你想要的进程信息。

5.2 网络

- ss/netstat: 查看当前系统的**网络连接和端口占用情况**，支持显示 TCP、UDP、UNIX 等多种协议的连接信息，可以通过各种选项来过滤和排序连接，常用的有 `ss -tlnup` 和 `netstat -tlnup`，列举出当前系统所有监听的 TCP 和 UDP 端口以及对应的进程信息。
- nload: 实时监控**各网卡流量**的工具，直观展示每个网卡进出口流量的折线图，可以帮助你快速了解系统的网络流量情况。
- iftop: 实时监控**各网络连接的流量**的工具，可以显示每个连接的 IP、流量大小、流量速率（2s、10s、40s）等信息，支持按照流量大小排序连接。
- nethogs: 实时监控**各进程的网络流量**的工具，可以显示每个进程的 PID、名称、流量大小、流量速率等信息，支持按照流量大小排序进程。
- tcpdump: 基于命令行的**网络数据包嗅探**工具，可以捕获和分析网络上的数据包，支持各种协议的解析和过滤，配合 Wireshark 等工具可以进行更为深入的网络分析。

5.3 I/O

- iostat: 实时监控**各磁盘 I/O 情况**的工具，可以显示每个磁盘的读写占用率、读写速率、平均等待时间等信息，帮助你了解系统的磁盘 I/O 性能，常用的有 `iostat -dx 1`，每秒刷新一次磁盘的详细 I/O 信息。
- iotop: 实时监控**各进程的 I/O 情况**，可以显示每个进程的 PID、名称、读写速率等信息，支持按照 I/O 占用率排序进程，常用的有 `iotop -o`，只显示正在进行 I/O 操作的进程。

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

6.1 Background

这一章主要讲一些 Linux 相关的基本知识，大部分取自于 ustclug 101 文档，但进行了大量删减，因此内容可能会比较杂乱，~~可能会想到哪讲到哪~~。

接下来我们将讲述以下内容：

- Linux 系统的基本概念和组成
  - Linux Kernel
  - C Standard Library (e.g. glibc, musl libc)
  - Core Utilities (e.g. GNU Coreutils, BusyBox)
  - …
- Kernel/User Space 的基本概念和关系
- Package Manager 的基本概念和作用
- 一个典型的 x86_64 平台下的 Linux 系统从开机上电到进入 tty 登录界面的全启动流程
  - Firmware → Bootloader → Kernel → Init System (Systemd)

6.2 Linux

Linux 不是操作系统（Operating System），而是系统的内核（Kernel）。

Linux 发行版（Distribution）才是完整的操作系统，一个完整的基于 Linux 内核的操作系统通常包含：

- Linux Kernel：系统内核，所有 Linux distro 的核心，负责底层硬件资源管理
- C Standard Library：C 标准库实现，常见的有 glibc（GNU/Linux）、musl libc（Alpine Linux）、Bionic libc（Android）等等，提供了 C 语言的标准库函数实现，供用户空间的程序调用
- Core Utilities：核心工具集，常见的有 GNU Coreutils、BusyBox 等等，提供了各种基本的命令行工具和实用程序，比如 ls、cp、mv、rm、cat、echo 等等
- Package Manager：包管理器，负责软件包的安装、更新、卸载和管理，比如 apt（Debian/Ubuntu）、yum/dnf（CentOS/Fedora）、pacman（Arch Linux）等等
- Other Applications/Libraries：其他发行版自带的应用程序和库，比如桌面环境等等

6.2 Linux

日常使用的 Ubuntu、Debian、CentOS、Fedora、Arch Linux 等等这些都属于 GNU/Linux 发行版，因为这些发行版主要采用的是 GNU 项目提供的 C 标准库（glibc）和核心工具集（GNU Coreutils）。

Android 属于 Android/Linux 发行版，因为它采用了 Google 自己开发的 Bionic libc 作为 C 标准库实现，使用 Toybox 作为核心工具集，以便绕过开源许可证的限制。

以上是对 ustclug 101 Chapter 1[1] 的一个非常简略的总结，更多细节请参阅原文档。

[1] https://101.lug.ustc.edu.cn/Ch01/

6.3 Kernel/User Space

在 Linux 系统中，Linux Kernel 运行的环境和其他软件运行的环境基本上是相对独立的，这在虚拟内存中表现为两个不同的地址空间，分别被称为 Kernel Space 和 User Space。

- Kernel Space 唯一运行着 Linux Kernel 及其内核模块，在系统启动前由 Bootloader 加载进内存，通常 Linux Kernel 存放在 `/boot` 目录下，文件名通常以 `vmlinuz` 开头，比如 `vmlinuz-6.12.74+deb13+1-amd64`，内核的生命周期和系统的生命周期是一样的，除非系统重启，否则内核一直在运行着。
- User Space 运行着各种各样的应用程序和库，这些程序和库通过 System Call（Syscall）来向 Kernel Space 发出请求，让内核帮它们完成一些底层的操作，比如文件读写、网络通信、进程管理等等。

简要来说，Linux Kernel 和其他所有应用程序和库之间的**主动联系**只有通过 System Call 来实现的，大部分时间里它们是相互独立运行着的。

6.4 System Call

System Call 在 x86_64 架构的机器上本质上就是一条名为 `syscall` 的机器指令，这条指令会触发 CPU 从 User Space 切换到 Kernel Space 来执行内核代码，完成系统调用的功能。

我们可以使用 strace 来监控任何一个应用的 System Call 调用情况，比如我们最先学到的程序，Hello World：

```c
#include <stdio.h>
int main() {
  printf("Hello, World!\n");
}
```

编译，运行和输出：

```
gcc test.c -o test && ./test
```

6.4 System Call

接下来我们可以使用以下命令来监控它的 System Call 调用情况：

```
strace ./test
```

输出：

```
execve("./test", ["./test"], 0x7ffe46b982a0 /* 95 vars */) = 0
...
write(1, "Hello, world!\n", 14Hello, world!
)          = 14
...
```

输出的每一行代表一个实际执行的 System Call，括号前面的是这个 System Call 的名字，括号里面的是这个 System Call 的参数，等号后面的是这个 System Call 的返回值。

6.4 System Call

我们可以使用 `unistd.h` 头文件里定义的 `syscall` 函数来直接发出 System Call 来让 Kernel 执行一些底层的操作，比如我们可以直接发出一个 `write` 的 System Call 来输出 Hello World：

```c
#include <unistd.h>
#include <sys/syscall.h>

int main() {
  syscall(SYS_write, 1, "Hello, World!\n", 14);
  return 0;
}
```

这和之前使用 `printf` 输出的效果是一样的，但是不再依赖于 C 标准库的 `stdio.h` 了。

6.5 Package Manager

开源社区是一个巨大的生态系统，成千上万的开源项目都依赖着其他项目，也被其他项目依赖着，形成一个巨大的依赖网络，而包管理器（Package Manager）就是用来管理这个依赖网络的工具，它能够帮助我们安装、更新、卸载这些软件包，并且自动处理它们之间的依赖关系，让我们能够更方便地使用 Linux 发行版。

以下是常见的 Linux 发行版和它们的包管理器：

- Debian/Ubuntu：使用 apt 作为包管理器，每个 package 使用 .deb 格式包装
- Red Hat/CentOS/Fedora：使用 dnf/yum 作为包管理器，每个 package 使用 .rpm 格式包装
- Arch Linux：使用 pacman 作为包管理器，每个 package 使用 .pkg.tar.zst 格式包装
- Alpine Linux：使用 apk 作为包管理器，每个 package 使用 .apk 格式包装

以 Debian/Ubuntu 为例，我们将简要介绍 apt 包管理器的基本概念。

6.5 Package Manager

一个包是以 .deb 结尾的文件，里面包含了这个包的所有文件和元数据，而包通常存放在一个叫做软件源（Source）的地方，软件源可以是官方的，比如 deb.debian.org，也可以是第三方的，比如 mirrors.ustc.edu.cn/debian，甚至可以是一个文件。

通常在安装/更新一个包前，我们需要先更新一下软件源的索引信息：

```
1 sudo apt update
```

这是因为远程的软件源可能会有新的包或者包的版本更新了，更新索引信息可以让我们获取到**最新**的软件包信息。

通常，安装一个包的命令是：

```
1 sudo apt install <package-name>
```

6.5 Package Manager

除此之外，更新一个或所有包的命令是：

```
sudo apt upgrade [package-name]
```

删除一个包的命令是：

```
sudo apt remove <package-name>
```

以上便是 apt 包管理器的基本使用方法了，更多更详细的用法可以参考 ustclug 101 Chapter 3[1]。

[1] https://101.lug.ustc.edu.cn/Ch03/

6.6 Boot Process

通常来说，一个操作系统从开机上电到进入登录界面的启动流程可以分为以下几个部分：

- Firmware (e.g. BIOS/UEFI)：负责基本的设备自检和硬件初始化，从存储设备中读取 Bootloader 程序的代码并将其加载到内存中，最后将控制权交给 Bootloader。
- Bootloader (e.g. GRUB/bootmgfw)：负责选择需要启动的系统，加载 Kernel 的代码到内存中，并将控制权交给 Kernel。
- Kernel (e.g. Linux Kernel)：负责硬件初始化和 Kernel Space 的初始化工作，比如设置内存管理、设备驱动、文件系统等等，最后启动 Init System 程序。
- Init System (e.g. Systemd)：负责 User Space 的初始化工作，比如启动各种服务、守护进程、登录界面等等，最后进入登录界面等待用户输入。

6.6 Boot Process

6.6.1 Firmware

在 x86_64 架构的机器上，Firmware 阶段通常指的是 BIOS 和 UEFI 这两种固件方案，其中 BIOS 是比较老旧的方案，UEFI 是比较新的方案，现代的计算机基本上都采用了 UEFI 方案。它们的执行流程如下所示：

- BIOS：系统自检 + 硬件初始化 → 从磁盘的主引导扇区（MBR，磁盘的第一个扇区，512 Bytes）加载 Bootloader 的第一阶段代码（446 bytes）到内存中，并将控制权交给 Bootloader。
- UEFI：系统自检 + 硬件初始化 → 从 EFI System Partition（ESP）中加载 Bootloader（通常是一个 .efi 文件）到内存中，并将控制权交给 Bootloader。

在 Linux 中，Bootloader 都存放在 `/boot` 目录下，在 BIOS 中 `/boot` 目录只是根目录下一个普通的目录，而在 UEFI 中 `/boot` 目录是一个挂载点，挂载了 ESP 分区。

6.6 Boot Process

6.6.2 Bootloader

常见的 Bootloader 有 GNU GRUB 2 和 Microsoft Windows Boot Manager（bootmgfw），分别是主流的 Linux 发行版和 Windows 系统的默认 Bootloader。

以 GRUB 为例，Bootloader 的主要功能就是提供一个引导菜单，让用户选择需要启动的系统，比如在装有双系统的计算机上，GRUB 会在开机时显示一个菜单，让用户选择是启动 Linux 还是 Windows，或者通过命令行操作使用 PXE 从网络启动等等。

当用户选择了需要启动的系统后，GRUB 会加载对应的 Kernel 的代码到内存中，并将控制权交给 Kernel 来完成后续的启动流程。

一个~~常见的~~定制化的 GRUB 引导菜单的示例：

6.6 Boot Process



6.6.3 Kernel

当 Bootloader 将 Linux Kernel 加载进内存后，Kernel 首先运行自解压代码将自己完全在内存中展开，然后进行硬件初始化和 Kernel Space 的初始化工作，比如设置内存管理、设备驱动、文件系统等等，最后启动 Init System 程序来完成 User Space 的初始化工作。

6.6.4 Init System

`/sbin/init` 是 Linux Kernel 加载完成后启动的第一个 User Space 的程序，它的主要功能就是完成 User Space 的初始化工作，比如启动各种服务、守护进程、登录界面等等，最后进入登录界面等待用户输入。

比如现代 Linux 发行版通常采用 Systemd 作为 Init System，以 Unit 为基本单位来管理系统服务之间的依赖关系，更古早的 Linux 发行版可能采用 SysVinit 作为 Init System，以 Runlevel 和优先级排序的方式来管理系统服务之间的依赖关系。

目录

1. ssh 1
   1.1 端口转发 2
   1.2 代理 8
   1.3 跳板 9
2. Docker 11
   2.1 安装 12
   2.2 拉取镜像 13
3. Nvidia 24
   3.1 安装驱动 25
   3.2 安装 CUDA 27
4. Python 29
   4.1 虚拟环境 30
   4.2 venv 31
   4.3 uv 32
   4.4 conda 35
   4.5 目录结构 37
   4.6 依赖问题 39
5. Monitor 40
   5.1 进程 41
   5.2 网络 42
   5.3 I/O 43
6. Basis 44
   6.1 Background 45
   6.2 Linux 46
   6.3 Kernel/User Space . 48
   6.4 System Call 49
   6.5 Package Manager . 52
   6.6 Boot Process 55
7. End 60
   7.1 References 61

7.1 References

- Linux 101[1]：ustclug 的 Linux 基础教程，内容非常全面，适合初学者入门，了解 Linux 的基本概念和使用方法。
- Linux 201[2]：ustclug 的 Linux 进阶教程，内容更加深入，适合有一定基础的用户继续学习，或者遇到一些疑难杂症可能会有帮助。
- USTC Mirror Help[3]：USTC Mirror 的帮助文档，包含了各种软件的安装和配置镜像的方法。

「常见 Linux 配置/使用问题」活动反馈



[1] https://101.lug.ustc.edu.cn/
[2] https://201.ustclug.org/
[3] https://mirrors.ustc.edu.cn/help/index.html