面向目标：**拉取现成镜像 + 少量开发**，用于跑 **VLA（Vision-Language-Action）模型**（训练/微调/推理/服务均可）。

> 适用前提：你已经在宿主机安装好 NVIDIA 驱动（你的是 580.95.05），并且 GPU 在宿主机上 `nvidia-smi` 正常。

目录

1. Docker 安装（官方仓库推荐）

1.1（可选）卸载旧版本/发行版自带 Docker

```
sudo apt remove -y docker docker-engine docker.io containerd runc
```

逐项解释：

- `sudo`：以 root 权限执行（apt 管理软件需要）
- `apt remove`：卸载软件包（通常保留配置文件）
- `-y`：自动确认 yes
- 后面的包名：Ubuntu 可能自带旧版本（`docker.io` 等），容易和官方 `docker-ce` 冲突

检查是否还有残留包：

```
dpkg -l | grep -E 'docker|containerd|runc'
```

- `dpkg -l`：列出已安装包
- `grep -E`：用正则筛选相关包

1.2 更新索引 + 安装依赖

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg
```

逐项解释：

- `apt update`：更新软件源索引（不安装，只刷新列表）
- `ca-certificates`：HTTPS 证书链，防止拉取 key/源失败
- `curl`：下载工具
- `gnupg`：导入 GPG key 用

1.3 添加 Docker 官方 GPG Key（apt 验证包签名）

```
sudo install -m 0755 -d /etc/apt/keyrings
```

解释：

- `install -d`：创建目录
- `-m 0755`：权限（owner 可读写执行，其他可读执行）
- `/etc/apt/keyrings`：推荐存放第三方源 key 的目录

导入 key：

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyring
```

逐参数说明：

- `curl -f`：HTTP 错误则返回非 0

- `-s`：静默（不显示进度条）
- `-S`：失败时显示错误
- `-L`：跟随重定向
- `|`：管道，把 key 内容传给 `gpg`
- `gpg --dearmor`：把 ASCII key 转二进制 keyring（apt 更推荐）
- `-o ...`：输出文件路径

设置 key 可读：

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

- `chmod a+r`：所有用户可读（apt 读取需要）

1.4 添加 Docker 官方 apt 源

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://downlc
  $(. /etc/os-release && echo ${UBUNTU_CODENAME}) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

逐段解释：

- `arch=$(dpkg --print-architecture)`：限制只使用本机架构（如 amd64）
- `signed-by=...`：指定该源使用哪个 key 验证
- `https://download.docker.com/linux/ubuntu`：Docker 官方源
- `$(. /etc/os-release && echo ${UBUNTU_CODENAME})`：
  - `. /etc/os-release`：加载系统版本变量
  - `${UBUNTU_CODENAME}`：24.04 为 `noble`
- `sudo tee ...`：以 root 权限写文件（避免 `sudo echo > file` 无法写）
- `> /dev/null`：丢弃 tee 的回显

检查源文件：

```
cat /etc/apt/sources.list.d/docker.list
```

1.5 安装 Docker Engine + 常用插件

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-pl
```

解释：

- `docker-ce`： Docker Engine（dockerd）
- `docker-ce-cli`： 命令行工具 `docker`
- `containerd.io`： 底层容器运行时
- `docker-buildx-plugin`： buildx（BuildKit、跨平台构建等）
- `docker-compose-plugin`： `docker compose`（新式 Compose）

1.6 检查服务状态、启动与开机自启

```
systemctl status docker --no-pager
```

- `systemctl status`： 查看服务状态
- `--no-pager`： 不启用分页器，便于复制

（通常已自动启用，仍可确认）

```
sudo systemctl enable docker
sudo systemctl start docker
```

- `enable`： 开机自启动
- `start`： 立刻启动

1.7 运行测试容器

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

逐项解释：

- `docker run`： 创建并启动一个新容器

- `hello-world`：镜像名（本地没有则自动 pull）
- `--rm`：容器退出后自动删除（避免留下停止容器）

2. 免 sudo 使用 Docker（docker 组）

把当前用户加入 docker 组：

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

逐参数解释：

- `usermod`：修改用户
- `-G docker`：设置附加组为 docker
- `-a`：append（追加）。**没有 `-a` 会覆盖你其他组，非常危险**
- `$USER`：当前用户名

让权限生效（二选一）：

- 注销/重登
- 或开启新 shell：

```
newgrp docker
```

验证无需 sudo：

```
docker ps
```

3. 让 Docker 容器使用 GPU（NVIDIA Container Toolkit）

3.1 先验证宿主机 GPU 正常

```
nvidia-smi
```

你应关注：

- GPU 是否识别为 RTX 5090
- Driver Version 是否显示 580.95.05

3.2 安装 NVIDIA Container Toolkit

```
sudo apt update
sudo apt install -y nvidia-container-toolkit
```

解释：

- 提供让容器识别 GPU 所需的 runtime/钩子与配置工具

3.3 配置 Docker runtime

```
sudo nvidia-ctk runtime configure --runtime=docker
```

解释：

- `nvidia-ctk`： NVIDIA 配置工具
- `runtime configure`： 生成/修改运行时配置
- `--runtime=docker`： 目标为 Docker（不是 k8s/containerd 单独配置）

这通常会修改/生成：

- `/etc/docker/daemon.json`

重启 Docker 生效：

```
sudo systemctl restart docker
```

3.4 GPU 容器验证（关键）

```
docker run --rm --gpus all nvidia/cuda:12.4.1-base-ubuntu22.04 nvidia-smi
```

逐项解释：

- `--rm`：用完删除容器
- `--gpus all`：把所有 GPU 暴露给容器（最常用）
- `nvidia/cuda:...`：带 CUDA runtime 的官方镜像
- `nvidia-smi`：在容器内执行验证工具

4. 必懂概念（镜像/容器/卷/网络/缓存）

- **镜像（Image）**：只读模板（多层 layer），用于创建容器
- **容器（Container）**：镜像的运行实例（加一层可写层），生命周期可控
- **Registry**：镜像仓库（Docker Hub/私有仓库）
- **Bind Mount**：把宿主机目录映射进容器（你的“少量开发”核心）
- **Volume**：Docker 管理的数据卷（更适合数据库/长期数据）
- **端口映射**：`-p host:container` 把容器服务暴露到宿主
- **缓存持久化**：把 HF/pip/torch 缓存目录挂载到宿主，避免反复下载

5. 日常工作流命令：镜像

5.1 搜索镜像（了解用途，生产不一定用）

```
docker search pytorch
```

- 在 Docker Hub 搜索（结果仅作参考）

5.2 拉取镜像

```
docker pull pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime
```

- `repo:tag`：指定版本，避免 `latest` 漂移

5.3 查看本地镜像

```
docker images
# 或更标准
docker image ls
```

5.4 查看镜像元信息（排错/确认入口命令）

```
docker inspect pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime
```

常用格式化提取：

```
docker inspect -f '{{.Id}}' pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime
```

- `-f`：Go template，避免看巨大 JSON

5.5 删除镜像

```
docker rmi pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime
```

如被容器引用会失败（需先删容器）。强制不建议日常使用：

```
docker rmi -f <image>
```

6. 日常工作流命令：容器（逐参数）

6.1 创建并进入交互式容器（临时、干净）

```
docker run -it --rm ubuntu:24.04 bash
```

逐参数解释：

- `run`：创建并启动新容器
- `-i`：保持 STDIN 打开（你可输入）

- `-t`：分配伪终端（交互体验正常）
- `--rm`：退出自动删除容器（避免垃圾）
- `bash`：容器启动后执行 bash（作为主进程）

退出：

```
exit
```

- 主进程 `bash` 结束 => 容器停止 => 因为 `--rm` 被删除

6.2 后台运行（做服务/长跑）

```
docker run -d --name my-ubuntu ubuntu:24.04 sleep infinity
```

逐参数解释：

- `-d`：detached，后台运行
- `--name my-ubuntu`：命名容器，便于管理
- `sleep infinity`：保持主进程存在，容器不退出（示例用途）

查看运行中容器：

```
docker ps
```

查看所有容器：

```
docker ps -a
```

6.3 进入运行中的容器（不创建新容器）

```
docker exec -it my-ubuntu bash
```

解释：

- `exec`：在“已运行容器”里启动新进程

- `-it`：交互终端
- 退出 exec 的 bash 不会停止容器（除非容器主进程本身退出）

6.4 容器生命周期控制

停止：

```
docker stop my-ubuntu
```

- 先发 SIGTERM，等待一段时间后 SIGKILL

指定等待时间：

```
docker stop -t 3 my-ubuntu
```

- `-t 3`：等待 3 秒

启动：

```
docker start my-ubuntu
```

重启：

```
docker restart my-ubuntu
```

删除（只能删停止的）：

```
docker rm my-ubuntu
```

强制删除（会 kill）：

```
docker rm -f my-ubuntu
```

6.5 看日志（服务化必备）

```
docker logs my-ubuntu
```

实时跟随：

```
docker logs -f my-ubuntu
```

- `-f`：follow（持续输出）

限制最后 N 行：

```
docker logs --tail 200 my-ubuntu
```

带时间戳：

```
docker logs -t my-ubuntu
```

6.6 inspect（排错必备）

```
docker inspect my-ubuntu
```

常用提取：

```
docker inspect -f '{{.State.Status}} {{.State.ExitCode}}' my-ubuntu
```

- 典型用于判断为何退出：状态/退出码

7. 存储：挂载目录 bind mount 与 Volume（逐参数）

7.1 Bind Mount（少量开发首选）

```
docker run -it --rm \
  -v $PWD:/workspace \
  -w /workspace \
  ubuntu:24.04 bash
```

逐项解释：

- `-v 宿主路径:容器路径`：把宿主目录映射进容器目录
  - `$PWD`：当前目录绝对路径
  - `/workspace`：容器中统一工作目录（习惯做法）
- `-w /workspace`：启动后默认在该目录（不用再 `cd`）

只读挂载（防止容器写宿主文件）：

```
docker run -it --rm -v $PWD:/workspace:ro ubuntu:24.04 bash
```

- `:ro`：read-only

更显式的 `--mount` 写法（推荐在复杂场景使用）：

```
docker run -it --rm \
  --mount type=bind,source="$PWD",target=/workspace \
  ubuntu:24.04 bash
```

解释：

- `type=bind`：绑定宿主路径
- `source=`：宿主目录
- `target=`：容器目录

7.2 Volume（长期数据更稳，适合数据库/缓存也可）

创建：

```
docker volume create mydata
```

使用：

```
docker run -d --name mysql \
  -e MYSQL_ROOT_PASSWORD=123456 \
  -v mydata:/var/lib/mysql \
  mysql:8
```

逐项解释：

- `-e KEY=VALUE`：环境变量（镜像用它做初始化）
- `-v mydata:/var/lib/mysql`：把数据目录放到 volume（容器删了数据还在）

查看：

```
docker volume ls
docker volume inspect mydata
```

删除（会删数据）：

```
docker volume rm mydata
```

8. 网络与端口映射（逐参数）

8.1 端口映射（跑 Gradio/Jupyter/API 必备）

```
docker run -d --name web -p 8080:80 nginx:alpine
```

解释：

- `-p 宿主端口:容器端口`：将容器 80 映射到宿主 8080
- 访问：`http://localhost:8080`

指定只允许本机访问（更安全）：

```
docker run -d -p 127.0.0.1:8080:80 nginx:alpine
```

查看容器映射：

```
docker port web
```

8.2 自定义网络（多容器互联）

创建：

```
docker network create mynet
```

加入网络启动：

```
docker run -d --name redis --network mynet redis:7
docker run -it --rm --network mynet redis:7 redis-cli -h redis ping
```

解释：

- `--network mynet`： 加入同一网络
- `-h redis`： 容器名可作为 DNS 名解析

查看：

```
docker network ls
docker network inspect mynet
```

9. VLA “黄金启动模板”（GPU+挂载+缓存+性能参数，逐参数）

9.1 为什么 VLA 必须重视缓存与 IPC

VLA 常见特点：

- 模型体积大（HF 模型频繁下载）
- 数据集大（HF datasets / 自定义数据）
- DataLoader 多进程 + 大 batch 容易触发共享内存不足（shm）

因此推荐：

- 挂载 **代码目录**
- 挂载 **缓存目录**
- 用 `--ipc=host` 或 `--shm-size` 增大共享内存

9.2 黄金启动命令（推荐直接使用）

```
docker run -it --rm \
  --name vla-dev \
  --gpus all \
  --ipc=host \
  --ulimit memlock=-1 --ulimit stack=67108864 \
  -e HF_HOME=/cache/hf \
  -e TRANSFORMERS_CACHE=/cache/hf/transformers \
  -e HF_DATASETS_CACHE=/cache/hf/datasets \
  -e TORCH_HOME=/cache/torch \
  -e PIP_CACHE_DIR=/cache/pip \
  -v $PWD:/workspace \
  -v $HOME/.cache/vla:/cache \
  -w /workspace \
  -p 7860:7860 \
  pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime \
  bash
```

逐参数精讲（按类别）：

A) 容器交互与清理

- `-it`：交互式终端组合（见上）
- `--rm`：退出即删（适合“少量开发、干净环境”）
- `--name vla-dev`：给容器命名，方便 `docker exec/logs/inspect`

B) GPU

- `--gpus all`：把所有 GPU 暴露给容器
  可替换为只给 0 号卡：

```
--gpus '"device=0"'
```

C) 共享内存与资源限制

- `--ipc=host`：容器共享宿主机 IPC（/dev/shm 通常更充足）
  常见收益：PyTorch DataLoader 不易报 `bus error` / shm 不足。
- `--ulimit memlock=-1`：解除内存锁定上限（少数库可能需要）
- `--ulimit stack=67108864`：增大栈（极少数情况保险）

> 替代方案：不用 `--ipc=host`，改用 `--shm-size=16g`（手动给容器 shm）：
>
> ```
> docker run ... --shm-size=16g ...
> ```

D) 缓存环境变量（避免反复下载）

- `-e KEY=VALUE`：设置环境变量
- `HF_HOME=/cache/hf`：HF 总缓存目录
- `TRANSFORMERS_CACHE`：transformers 模型缓存
- `HF_DATASETS_CACHE`：datasets 缓存
- `TORCH_HOME`：torch hub 等缓存
- `PIP_CACHE_DIR`：pip 下载缓存

E) 挂载（少量开发核心）

- `-v $PWD:/workspace`：挂载当前项目到容器
- `-v $HOME/.cache/vla:/cache`：把缓存持久化到宿主机
  建议把 `$HOME/.cache/vla` 放到大磁盘/数据盘

F) 工作目录与端口

- `-w /workspace`：容器默认工作目录
- `-p 7860:7860`：常用于 Gradio/服务端口
  若你跑 Jupyter，常用：
  - `-p 8888:8888`

G) 镜像与启动命令

- `pytorch/pytorch:...runtime`： 省心镜像（适合少量开发）
- `bash`： 进入 shell 由你手动启动训练/推理/服务

10. 容器内少量开发：pip/缓存/可重复安装

10.1 用正确方式调用 pip（避免指错 python）

容器内：

```
python -m pip install -U pip
python -m pip install -r requirements.txt
```

解释：

- `python -m pip`： 确保使用当前 python 的 pip
- `-U`： 升级

10.2 建议把“少量依赖”固化成 setup 脚本

原因：

- 你用 `--rm` 时容器会消失，但缓存不会消失（我们挂载了 pip cache）
- 每次容器起来跑 `setup.sh` 很快、且步骤一致

示例（在项目里建 `scripts/setup.sh`）：

```
#!/usr/bin/env bash
set -e

python -m pip install -U pip
python -m pip install -r requirements.txt
```

11. GPU 运行常用模式：选卡/验证/常见坑

11.1 验证 GPU（容器内）

```
nvidia-smi
```

若容器镜像里没有 `nvidia-smi`，并不代表 GPU 不可用；但最稳妥是用 `nvidia/cuda` 镜像验证（见 3.4）。

11.2 只用某张卡

```
docker run --rm --gpus '"device=0"' nvidia/cuda:12.4.1-base-ubuntu22.04 nvidia-smi
```

11.3 常见坑

1. **`--gpus` 报错**：多为 toolkit 未配置/没重启 docker
2. **端口映射后访问不到**：服务要监听 `0.0.0.0`，不能只监听 `127.0.0.1`
3. **DataLoader 报 shm/bus error**：用 `--ipc=host` 或 `--shm-size=16g`
4. **反复下载模型**：没挂载 HF cache 或没设置 `HF_HOME/TRANSFORMERS_CACHE`

12. 日志、排错与清理（逐参数）

12.1 Docker 服务日志（daemon 层排错）

```
sudo journalctl -u docker -n 200 --no-pager
```

解释：

- `journalctl -u docker`： 查看 docker 服务日志
- `-n 200`： 最近 200 行
- `--no-pager`： 不分页

12.2 容器日志（应用层）

```
docker logs -f --tail 200 <container>
```

- `-f`：跟随
- `--tail 200`：最后 200 行

12.3 查看磁盘占用（非常重要）

```
docker system df
```

12.4 清理停止容器、无用网络、悬空镜像

```
docker system prune
```

解释：

- 会交互确认
- 删除：停止容器、未使用网络、dangling 镜像、构建缓存

更彻底（删除所有未被引用镜像，慎用）：

```
docker system prune -a
```

- `-a`：all，可能删掉你之后还想用的镜像（重新 pull 会耗时）

清理未使用的卷（慎用，可能删数据）：

```
docker volume prune
```

13. 一键脚本：run_docker.sh（推荐）

用途：把一长串参数固化，避免每次手敲出错；也方便你在团队里共享一致环境。

在项目根目录执行：

```
cat > run_docker.sh <<'EOF'
#!/usr/bin/env bash
set -e

IMAGE="pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime"
NAME="vla-dev"

mkdir -p "$HOME/.cache/vla"

docker run -it --rm \
  --name "$NAME" \
  --gpus all \
  --ipc=host \
  --ulimit memlock=-1 --ulimit stack=67108864 \
  -e HF_HOME=/cache/hf \
  -e TRANSFORMERS_CACHE=/cache/hf/transformers \
  -e HF_DATASETS_CACHE=/cache/hf/datasets \
  -e TORCH_HOME=/cache/torch \
  -e PIP_CACHE_DIR=/cache/pip \
  -v "$PWD":/workspace \
  -v "$HOME/.cache/vla":/cache \
  -w /workspace \
  -p 7860:7860 \
  "$IMAGE" \
  bash
EOF

chmod +x run_docker.sh
```

逐项解释：

- `cat > run_docker.sh <<'EOF' ... EOF`：把多行内容写入文件
  `<<'EOF'`：**禁止变量展开**（避免意外替换）
- `chmod +x`：赋执行权限
- `set -e`：任一命令失败就退出（避免“失败还继续跑”）

运行：

```
./run_docker.sh
```

附：你下一步通常会做的两件事（建议）

1. **把模型/数据再单独挂载**（避免项目目录过大、权限混乱）
   例如额外挂载：
   - `-v /data/models:/models`
   - `-v /data/datasets:/datasets`
   - `-v /data/logs:/logs`
2. **把端口、镜像 tag、选卡策略写进脚本参数**（让脚本可复用）