Missing Semester : Development Environment

Presented by comonad

Intro

How to start a local WebUI like ChatSJTU? Or load a local model with CUDA?

What is a Development Environment?

A collection of components working together to build and run software including:

- Core System Components
- Language-Specific Toolchain & Libraries
- Development tools & Configuration
- External Resources & Services
- Miscellaneous

What is a Development Environment?

Core System Components

- C Runtime Library (e.g. glibc, musl-libc)
- System Libraries (e.g. POSIX APIs, Windows API)
- System Dependencies (e.g. OpenSSL, zlib, ncurses)

What is a Development Environment?

Language-Specific Toolchain & Libraries

- Compiler/Interpreter (e.g. GCC, Clang, Python, Node.js)
- Build Tools/Package Managers (e.g. GNU Make, CMake, npm, cargo)
- Standard & Third-Party Libraries (e.g. torch, numpy)

What is a Development Environment?

Development tools & Configuration

- Code Editors/IDEs (e.g. Vim, Emacs, VSCode, JetBrains IDE)
- Language Servers, Linters, Formatters (e.g. Clangd, Pylance)
- Debuggers, Testers (e.g. GDB, LLDB)
- Version Control (e.g. Git, Mercury(Hg))

What is a Development Environment?

Miscellaneous

- Documentation: Local or online references (e.g. man pages).
- Environment Variables: Configuration for path / secret.

Environment variables are key-value pairs that configure system or application behavior. They are inherited by child processes and used by tools, compilers, and IDEs.

Common Variables

- `PWD` : Current directory
- `PATH` : Directories where the system searches for executables (e.g. compilers, tools).
- `SHLVL` : Indicates nested shell levels, starting at 1
- Compiler / Linker Variables:
  - `CC` / `CXX` : Specify C/C++ compilers (e.g. gcc, clang).
  - `LD` : Linker executable.
  - `LIBRARY_PATH` : Directories to search for shared libraries at *comptime*.
  - `LD_PRELOAD` / `LD_LIBRARY_PATH` : Directories to search for shared libraries at *runtime*.
  - `CUDA_HOME`
- Build System Variables:
  - `CMAKE_INCLUDE_PATH` / `CMAKE_LIBRARY_PATH` : Additional include/library directories for CMake.

See what's in `PATH`?

How does environment variable interact with your program?

1. Program executable not found? (not in `PATH`)
2. Library not installed for this language (Scripting language has its library that is a frontend of another binary library)
3. `.so` not found? (binary library not found. related to package manager in `/lib`)
4. `ld` undefined symbol? (API version in Library)

How does environment variable interact with your program?

```
`PATH` →
Scripting Language Library →
Linker Loading Binary Library →
Finding the symbol/furction inside the library (e.g. `.gpu()` `.cpu()` in Pytorch) →
interact with OS/Driver using that specific function can be invoked by the Language
```

If one component goes wrong, the entire process is incorrect leading to fault. However program like Python only shows `CUDA not available` :(

You need to diagnose them one by one or take the purity form (see Docker/Nix/...).

! In case of any problem, add `--verbose` `--debug` `LOG_LEVEL=debug` or else (according to manual) to enable full diagnose of a program.

How does environment variable interact with your program?

CUDA initialization failed. Possible reasons:

1. GPU driver not found. (`nvidia-smi` Detected driver version: None)
2. CUDA toolkit mismatch. (PyTorch requires CUDA 12.1, found CUDA 11.8 in `/usr/local/cuda`)
3. No GPU detected. (Check if NVIDIA GPU is available and drivers are loaded. This may require a `Linus-NVIDIA.gif` .)
4. Missing library: `libcudart.so.12.1` (searched in `/lib` , `/usr/lib` , `LD_LIBRARY_PATH` , ...)

Anti-patterns & Suggestion

- **DO NOT** put everything in shellInit
- **DO NOT** install all toolchains globally

Anti-patterns & Suggestion

DO NOT put everything in shellInit

`/etc/bashrc` `/etc/profile` `/etc/environment` (system-level) (login shell, interactive shell)
`~/.bashrc` `~/.zshrc` (user-level)

(burden to switch back & forth, extra impurity).

Suggestion:

- Set ENVs *ONLY* after you fully understand its usage and effects. (otherwise RTFM)
- Use a reliable secret management tool, e.g. PAM, `git-agecrypt`, `sops-nix`, etc.

Anti-patterns & Suggestion

DO NOT install all toolchains globally

Most Linux distros provide an out-of-box system-level package manager.

Some basic toolchain can be used in most situations without doubt:

- (Deb) `build-essential`
- (RedHat) `"Development Tools"`
- (Arch) `base-devel`

However, some may notice that by default most distros have disabled **system-level** `pip install` command.

PyPA 规范

Anti-patterns & Suggestion

DO NOT install all toolchains globally

Suggestion:

- Carefully deal with multiple versions of the same tool.

e.g. `update-alternatives`, `venv`

- Put required buildInputs in project's subdirectory (or load from `.env`).

e.g. `node_modules` (despite of fragmentation issue), `direnv`

- Improve build system & IDE integration.

e.g. `compile_commands.json`, `pyproject.toml`, `toolchain.toml`, `devenv`, `direnv`

- If the required toolchain is deprecated or too complicated, use container (docker/podman)

Isolate for sanity (purity)

Docker acts like a virtual machine but is not virtual machine.

Namely namespaces, cgroups & chroots with OverlayFS basically gives a MATRIX-like AR headset for each process for them to believe they are in a separated UNIX socket, though they are indeed on a physical machine.

In this way, executables insider container can only change in a restricted environment.

Isolate for sanity (purity)

Use Dockerfile, docker-compose, arion, etc. to declaratively build containers.

`foo.Dockerfile`

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

`docker-compose.yaml`

```yaml
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./app:/app
  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: example
```

Nix

Nix provides reproducible development environments through:

- Declarative Configuration: Precise dependency specification
- Immutable Store: All dependencies stored in /nix/store with unique hashes
- Flakes: Experimental feature for dependency pinning

`flake.nix`

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
    treefmt-nix = {
      url = "github:numtide/treefmt-nix";
      inputs.nixpkgs.follows = "nixpkgs";
    };
    git-hooks-nix = {
      url = "github:cachix/git-hooks.nix";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };
```

Network issues

- Use mirror repositories when direct connection is poor

SJTUG: https://mirror.sjtu.edu.cn https://mirrors.sjtug.sjtu.edu.cn

MirrorZ (CERNET): https://mirrors.cernet.edu.cn

- Network inside container (`Netavark`)

host/bridge mode, communication between containers

Further reading

- Missing Semester - LCPU
- My blog
- Fast, Declarative, Reproducible and Composable Developer Environments using Nix
- direnv – unclutter your .profile
- CMAKE_EXPORT_COMPILE_COMMANDS
- Writing your pyproject.toml
- The pyproject.toml file - Poetry
- Networking overview | Docker Docs
- Podman Configuring Networking