Development Environments on NixOS

NixOS's reproducibility makes it ideal for building development environments. However, if you're used to other distros, you may encounter problems because NixOS has its own logic. We'll briefly explain this below.

In the following sections, we'll introduce how the development environment works in NixOS.

Creating a Custom Shell Environment with `nix shell`

The simplest way to create a development environment is to use `nix shell .` `nix shell` will create a shell environment with the specified Nix package installed.

Here's an example:

```
# hello is not available
› hello
hello: command not found

# Enter an environment with the 'hello' and `cowsay` package
› nix shell nixpkgs#hello nixpkgs#cowsay

# hello is now available
› hello
Hello, world!

# ponysay is also available
› cowsay "Hello, world!"
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

`nix shell` is very useful when you just want to try out some packages or quickly create a clean environment.

Creating a Development Environment

`nix shell` is simple and easy to use, but it's not very flexible, for a more complex development environment, we need to use `pkgs.mkShell` and `nix develop`.

We can create a development environment using `pkgs.mkShell { ... }` and open an interactive Bash shell of this development environment using `nix develop`.

To see how `pkgs.mkShell` works, let's take a look at its source code.

nix

```nix
{ lib, stdenv, buildEnv }:

# A special kind of derivation that is only meant to be consumed by the
# nix-shell.
{ name ? "nix-shell"
, # a list of packages to add to the shell environment
  packages ? [ ]
, # propagate all the inputs from the given derivations
  inputsFrom ? [ ]
, buildInputs ? [ ]
, nativeBuildInputs ? [ ]
, propagatedBuildInputs ? [ ]
, propagatedNativeBuildInputs ? [ ]
, ...
}@attrs:
let
  mergeInputs = name:
    (attrs.${name} or [ ]) ++
    (lib.subtractLists inputsFrom (lib.flatten (lib.catAttrs name inputsFrom)));

  rest = builtins.removeAttrs attrs [
    "name"
    "packages"
    "inputsFrom"
    "buildInputs"
    "nativeBuildInputs"
    "propagatedBuildInputs"
```

```
      "propagatedNativeBuildInputs"
      "shellHook"
    ];
  in

  stdenv.mkDerivation ({
    inherit name;

    buildInputs = mergeInputs "buildInputs";
    nativeBuildInputs = packages ++ (mergeInputs "nativeBuildInputs");
    propagatedBuildInputs = mergeInputs "propagatedBuildInputs";
    propagatedNativeBuildInputs = mergeInputs "propagatedNativeBuildInputs";

    shellHook = lib.concatStringsSep "\n" (lib.catAttrs "shellHook"
      (lib.reverseList inputsFrom ++ [ attrs ]));

    phases = [ "buildPhase" ];

    # ......

    # when distributed building is enabled, prefer to build locally
    preferLocalBuild = true;
  } // rest)
```

`pkgs.mkShell { ... }` is a special derivation (Nix package). Its `name`, `buildInputs`, and other parameters are customizable, and `shellHook` is a special parameter that will be executed when `nix develop` enters the environment.

Here is a `flake.nix` that defines a development environment with Node.js 18 installed:

nix

```nix
{
  description = "A Nix-flake-based Node.js development environment";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = { self , nixpkgs ,... }: let
    # system should match the system you are running on
    # system = "x86_64-linux";
    system = "x86_64-darwin";
```

```
  in {
    devShells."${system}".default = let
      pkgs = import nixpkgs {
        inherit system;
      };
    in pkgs.mkShell {
      # create an environment with nodejs_18, pnpm, and yarn
      packages = with pkgs; [
        nodejs_18
        nodePackages.pnpm
        (yarn.override { nodejs = nodejs_18; })
      ];

      shellHook = ''
        echo "node `${pkgs.nodejs}/bin/node --version`"
      '';
    };
  };
}
```

Create an empty folder, save the above configuration as `flake.nix`, and then execute `nix develop` (or more precisely, you can use `nix develop .#default`), the current version of nodejs will be outputted, and now you can use `node` `pnpm` `yarn` seamlessly.

Using zsh/fish/... instead of bash

`pkgs.mkShell` uses `bash` by default, but you can also use `zsh` or `fish` by add `exec <your-shell>` into `shellHook`.

Here is an example:

nix

```
{
  description = "A Nix-flake-based Node.js development environment";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = { self , nixpkgs ,... }: let

```

```
      # system should match the system you are running on
      # system = "x86_64-linux";
      system = "x86_64-darwin";
    in {
      devShells."${system}".default = let
        pkgs = import nixpkgs {
          inherit system;
        };
      in pkgs.mkShell {
        # create an environment with nodejs_18, pnpm, and yarn
        packages = with pkgs; [
          nodejs_18
          nodePackages.pnpm
          (yarn.override { nodejs = nodejs_18; })
          nushell
        ];

        shellHook = ''
          echo "node `${pkgs.nodejs}/bin/node --version`"
          exec nu
        '';
      };
    };
}
```

With the above configuration, `nix develop` will enter the REPL environment of nushell.

Creating a Development Environment with `pkgs.runCommand`

The derivation created by `pkgs.mkShell` cannot be used directly, but must be accessed via `nix develop`.

It is actually possible to create a shell wrapper containing the required packages via `pkgs.stdenv.mkDerivation`, which can then be run directly into the environment by executing the wrapper.

Using `mkDerivation` directly is a bit cumbersome, and Nixpkgs provides some simpler functions to help us create such wrappers, such as `pkgs.runCommand`.

Example:

nix

```nix
{
  description = "A Nix-flake-based Node.js development environment";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = { self , nixpkgs ,... }: let
    # system should match the system you are running on
    # system = "x86_64-linux";
    system = "x86_64-darwin";
  in {
    packages."${system}".dev = let
      pkgs = import nixpkgs {
        inherit system;
      };
      packages = with pkgs; [
          nodejs_20
          nodePackages.pnpm
          nushell
      ];
    in pkgs.runCommand "dev-shell" {
      # Dependencies that should exist in the runtime environment
      buildInputs = packages;
      # Dependencies that should only exist in the build environment
      nativeBuildInputs = [ pkgs.makeWrapper ];
    } ''
      mkdir -p $out/bin/
      ln -s ${pkgs.nushell}/bin/nu $out/bin/dev-shell
      wrapProgram $out/bin/dev-shell --prefix PATH : ${pkgs.lib.makeBinPath pack
    '';
  };
}
```

Then execute `nix run .#dev` or `nix shell .#dev --command 'dev-shell'` , you will enter a nushell session, where you can use the `node` `pnpm` command normally, and the node version is 20.

The wrapper generated in this way is an executable file, which does not actually depend on the `nix run` or `nix shell` command.

For example, we can directly install this wrapper through NixOS's `environment.systemPackages`, and then execute it directly:

```nix
{pkgs, lib, ...}:{

  environment.systemPackages = [
    # Install the wrapper into the system
    (let
      packages = with pkgs; [
          nodejs_20
          nodePackages.pnpm
          nushell
      ];
    in pkgs.runCommand "dev-shell" {
      # Dependencies that should exist in the runtime environment
      buildInputs = packages;
      # Dependencies that should only exist in the build environment
      nativeBuildInputs = [ pkgs.makeWrapper ];
    } ''
      mkdir -p $out/bin/
      ln -s ${pkgs.nushell}/bin/nu $out/bin/dev-shell
      wrapProgram $out/bin/dev-shell --prefix PATH : ${pkgs.lib.makeBinPath pack
    '')
  ];
}
```

Add the above configuration to any NixOS Module, then deploy it with `sudo nixos-rebuild switch`, and you can enter the development environment directly with the `dev-shell` command, which is the special feature of `pkgs.runCommand` compared to `pkgs.mkShell`.

Related source code:

- pkgs/build-support/trivial-builders/default.nix - runCommand
- pkgs/build-support/setup-hooks/make-wrapper.sh

Enter the build environment of any Nix package

Now let's take a look at `nix develop`, first read the help document output by `nix develop --help`:

```
Name
    nix develop - run a bash shell that provides the build environment of a deri

Synopsis
    nix develop [option...] installable
# ......
```

It tells us that `nix develop` accepts a parameter `installable`, which means that we can enter the development environment of any installable Nix package through it, not just the environment created by `pkgs.mkShell`.

By default, `nix develop` will try to use the following attributes in the flake outputs:

- `devShells.<system>.default`
- `packages.<system>.default`

If we use `nix develop /path/to/flake#<name>` to specify the flake package address and flake output name, then `nix develop` will try the following attributes in the flake outputs:

- `devShells.<system>.<name>`
- `packages.<system>.<name>`
- `legacyPackages.<system>.<name>`

Now let's try it out. First, test it to confirm that We don't have `c++` `g++` and other compilation-related commands in the current environment:

shell

```
ryan in 🌐 aquamarine in ~
› c++
c++: command not found

ryan in 🌐 aquamarine in ~
› g++
g++: command not found
```

Then use `nix develop` to enter the build environment of the `hello` package in `nixpkgs`:

```shell
# login to the build environment of the package `hello`
ryan in 🌐 aquamarine in ~
› nix develop nixpkgs#hello

ryan in 🌐 aquamarine in ~ via ❄️  impure (hello-2.12.1-env)
› env | grep CXX
CXX=g++

ryan in 🌐 aquamarine in ~ via ❄️  impure (hello-2.12.1-env)
› c++ --version
g++ (GCC) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

ryan in 🌐 aquamarine in ~ via ❄️  impure (hello-2.12.1-env)
› g++ --version
g++ (GCC) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

We can see that the `CXX` environment variable have been set, and the `c++` `g++` and other commands can be used normally now.

In addition, we can also call every build phase of the `hello` package normally:

> The default execution order of all build phases of a Nix package is: `$prePhases unpackPhase patchPhase $preConfigurePhases configurePhase $preBuildPhases buildPhase checkPhase $preInstallPhases installPhase fixupPhase installCheckPhase $preDistPhases distPhase $postPhases`

```shell
# unpack source code
ryan in 🌐 aquamarine in /tmp/xxx via ❄️  impure (hello-2.12.1-env)
› unpackPhase
unpacking source archive /nix/store/pa10z4ngm0g83kx9mssrqzz30s84vq7k-hello-2.12.
source root is hello-2.12.1
setting SOURCE_DATE_EPOCH to timestamp 1653865426 of file hello-2.12.1/ChangeLog

ryan in 🌐 aquamarine in /tmp/xxx via ❄️  impure (hello-2.12.1-env)
› ls
```

```
hello-2.12.1

ryan in 🌐 aquamarine in /tmp/xxx via ❄  impure (hello-2.12.1-env)
› cd hello-2.12.1/

# generate Makefile
ryan in 🌐 aquamarine in /tmp/xxx/hello-2.12.1 via ❄  impure (hello-2.12.1-env
› configurePhase
configure flags: --prefix=/tmp/xxx/outputs/out --prefix=/tmp/xxx/outputs/out
checking for a BSD-compatible install... /nix/store/02dr9ymdqpkb75vf0v1z2l91z2q3
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /nix/store/02dr9ymdqpkb75vf0v1z2l91z2q3iz
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
# ......
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating po/Makefile.in
config.status: creating config.h
config.status: config.h is unchanged
config.status: executing depfiles commands
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile

# build the package
ryan in 🌐 aquamarine in /tmp/xxx/hello-2.12.1 via C v12.3.0-gcc via ❄  impure
› buildPhase
build flags: SHELL=/run/current-system/sw/bin/bash
make  all-recursive
make[1]: Entering directory '/tmp/xxx/hello-2.12.1'
# ......
ranlib lib/libhello.a
gcc  -g -O2   -o hello src/hello.o  ./lib/libhello.a
make[2]: Leaving directory '/tmp/xxx/hello-2.12.1'
make[1]: Leaving directory '/tmp/xxx/hello-2.12.1'

# run the built program
ryan in 🌐 aquamarine in /tmp/xxx/hello-2.12.1 via C v12.3.0-gcc via ❄  impure
› ./hello
Hello, world!
```

This usage is mainly used to debug the build process of a Nix package, or to execute some commands in the build environment of a Nix package.

---

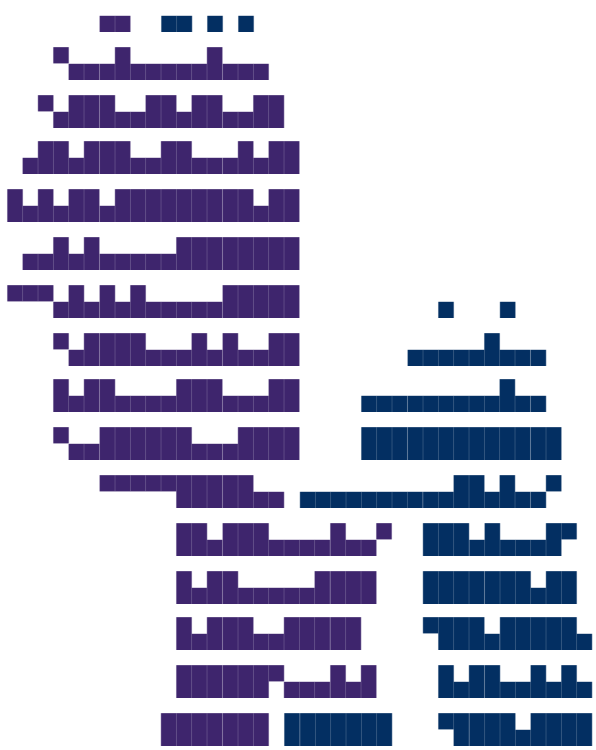
nix build

The `nix build` command is used to build a software package and creates a symbolic link named `result` in the current directory, which points to the build result.

Here's an example:

bash

```
# Build the package 'ponysay' from the 'nixpkgs' flake
nix build "nixpkgs#ponysay"
# Use the built 'ponysay' command
› ./result/bin/ponysay 'hey buddy!'
 _____________
< hey buddy! >
 -------------
     \
      \
       \
```



Using `nix profile` to manage development environments and entertainment environments

`nix develop` is a tool for creating and managing multiple user environments, and switch to different environments when needed.

Unlike `nix develop`, `nix profile` manages the user's system environment, instead of creating a temporary shell environment. So it's more compatible with Jetbrains IDE / VSCode and other IDEs, and won't have the problem of not being able to use the configured development environment in the IDE.

TODO

Other Commands

There are other commands like `nix flake init`, which you can explore in New Nix Commands. For more detailed information, please refer to the documentation.

References

- pkgs.mkShell - nixpkgs manual
- A minimal nix-shell
- Wrapping packages - NixOS Cookbook
- One too many shell, Clearing up with nix' shells nix shell and nix-shell - Yannik Sander
- Shell Scripts - NixOS Wiki

Loading comments...