The combination ability of Flakes and Nixpkgs module system

Nixpkgs Module Structure Explained

> The detailed workings of this module system will be introduced in the following Modularizing NixOS Configuration section. Here, we'll just cover some basic knowledge.

You might be wondering why the `/etc/nixos/configuration.nix` configuration file adheres to the Nixpkgs Module definition and can be referenced directly within the `flake.nix` .

To understand this, we need to first learn about the origin of the Nixpkgs module system and its purpose.

All the implementation code of NixOS is stored in the Nixpkgs/nixos directory, and most of these source codes are written in the Nix language. To write and maintain such a large amount of Nix code, and to allow users to flexibly customize various functions of their NixOS system, a modular system for Nix code is essential.

This modular system for Nix code is also implemented within the Nixpkgs repository and is primarily used for modularizing NixOS system configurations. However, it is also widely used in other contexts, such as nix-darwin and home-manager. Since NixOS is built on this modular system, it is only natural that its configuration files, including `/etc/nixos/configuration.nix` , are Nixpkgs Modules.

Before delving into the subsequent content, it's essential to have a basic understanding of how this module system operates.

Here's a simplified structure of a Nixpkgs Module:

nix

```nix
{lib, config, options, pkgs, ...}:
{
  # Importing other Modules
  imports = [

```

```
      # ...
      ./xxx.nix
    ];
    for.bar.enable = true;
    # Other option declarations
    # ...
  }
```

The definition is actually a Nix function, and it has five **automatically generated, automatically injected, and declaration-free parameters** provided by the module system:

1. `lib` : A built-in function library included with nixpkgs, offering many practical functions for operating Nix expressions.
   - For more information, see https://nixos.org/manual/nixpkgs/stable/#id-1.4.
2. `config` : A set of all options' values in the current environment, which will be used extensively in the subsequent section on the module system.
3. `options` : A set of all options defined in all Modules in the current environment.
4. `pkgs` : A collection containing all nixpkgs packages, along with several related utility functions.
   - At the beginner stage, you can consider its default value to be `nixpkgs.legacyPackages."${system}"` , and the value of `pkgs` can be customized through the `nixpkgs.pkgs` option.
5. `modulesPath` : A parameter available only in NixOS, which is a path pointing to nixpkgs/nixos/modules.
   - It is defined in nixpkgs/nixos/lib/eval-config-minimal.nix#L43.
   - It is typically used to import additional NixOS modules and can be found in most NixOS auto-generated `hardware-configuration.nix` files.

Passing Non-default Parameters to Submodules

If you need to pass other non-default parameters to submodules, you will need to use some special methods to manually specify these non-default parameters.

The Nixpkgs module system provides two ways to pass non-default parameters:

1. The `specialArgs` parameter of the `nixpkgs.lib.nixosSystem` function

2. Using the `_module.args` option in any module to pass parameters

The official documentation for these two parameters is buried deep and is vague and hard to understand. If readers are interested, I will include the links here:

1. `specialArgs`：There are scattered mentions related to it in the NixOS Manual and the Nixpkgs Manual.
   - Nixpkgs Manual: Module System - Nixpkgs
   - NixOS Manual: nixpkgs/nixos-24.11/nixos/doc/manual/development/option-types.section.md#L237-L244
2. `_module.args`：
   - NixOS Manual: Appendix A. Configuration Options
   - Source Code: nixpkgs/nixos-24.11/lib/modules.nix - _module.args

In short, `specialArgs` and `_module.args` both require an attribute set as their value, and they serve the same purpose, passing all parameters in the attribute set to all submodules. The difference between them is:

1. The `_module.args` option can be used in any module to pass parameters to each other, which is more flexible than `specialArgs`, which can only be used in the `nixpkgs.lib.nixosSystem` function.
2. `_module.args` is declared within a module, so it must be evaluated after all modules have been evaluated before it can be used. This means that **if you use the parameters passed through `_module.args` in `imports = [ ... ];`, it will result in an `infinite recursion` error**. In this case, you must use `specialArgs` instead.

I personally prefer `specialArgs` because it is more straightforward and easier to use, and the naming style of `_xxx` makes it feel like an internal thing that is not suitable for use in user configuration files.

Suppose you want to pass a certain dependency to a submodule for use. You can use the `specialArgs` parameter to pass the `inputs` to all submodules:

nix

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";
    another-input.url = "github:username/repo-name/branch-name";
  };

  outputs = inputs@{ self, nixpkgs, another-input, ... }: {
```

```
    nixosConfigurations.my-nixos = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";

      # Set all inputs parameters as special arguments for all submodules,
      # so you can directly use all dependencies in inputs in submodules
      specialArgs = { inherit inputs; };
      modules = [
        ./configuration.nix
      ];
    };
  };
}
```

Or you can achieve the same effect using the `_module.args` option:

nix

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";
    another-input.url = "github:username/repo-name/branch-name";
  };
  outputs = inputs@{ self, nixpkgs, another-input, ... }: {
    nixosConfigurations.my-nixos = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ./configuration.nix
        {
          # Set all inputs parameters as special arguments for all submodules,
          # so you can directly use all dependencies in inputs in submodules
          _module.args = { inherit inputs; };
        }
      ];
    };
  };
}
```

Choose one of the two methods above to modify your configuration, and then you can use the `inputs` parameter in `/etc/nixos/configuration.nix`. The module system will automatically match the `inputs` defined in `specialArgs` and inject it into all submodules that require this parameter:

```nix
# Nix will match by name and automatically inject the inputs
# from specialArgs/_module.args into the third parameter of this function
{ config, pkgs, inputs, ... }:
{
  # ...
}
```

The next section will demonstrate how to use `specialArgs` / `_module.args` to install system software from other flake sources.

Installing System Software from Other Flake Sources

The most common requirement for managing a system is to install software, and we have already seen in the previous section how to install packages from the official nixpkgs repository using `environment.systemPackages` . These packages all come from the official nixpkgs repository.

Now, we will learn how to install software packages from other flake sources, which is much more flexible than installing directly from nixpkgs. The main use case is to install the latest version of a software that is not yet added or updated in Nixpkgs.

Taking the Helix editor as an example, here's how to compile and install the master branch of Helix directly.

First, add the helix input data source to `flake.nix` :

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";

    # helix editor, use the master branch
    helix.url = "github:helix-editor/helix/master";
  };

  outputs = inputs@{ self, nixpkgs, ... }: {
    nixosConfigurations.my-nixos = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      specialArgs = { inherit inputs; };

```

```
        modules = [
          ./configuration.nix

          # This module works the same as the `specialArgs` parameter we used abov
          # choose one of the two methods to use
          # { _module.args = { inherit inputs; };}
        ];
      };
    };
  }
```

Next, you can reference this flake input data source in `configuration.nix` :

nix

```nix
{ config, pkgs, inputs, ... }:
{
  # ...
  environment.systemPackages = with pkgs; [
    git
    vim
    wget
    # Here, the helix package is installed from the helix input data source
    inputs.helix.packages."${pkgs.system}".helix
  ];
  # ...
}
```

Make the necessary changes and deploy with `sudo nixos-rebuild switch` . The deployment will take much longer this time because Nix will compile the entire Helix program from source.

After deployment, you can directly test and verify the installation using the `hx` command in the terminal.

Additionally, if you just want to try out the latest version of Helix and decide whether to install it on your system later, there is a simpler way to do it in one command (but as mentioned earlier, compiling from source will take a long time):

bash

```bash
nix run github:helix-editor/helix/master
```

We will go into more detail on the usage of `nix run` in the following section Usage of the New CLI.

Leveraging Features from Other Flakes Packages

In fact, this is the primary functionality of Flakes — a flake can depend on other flakes, allowing it to utilize the features they provide. It's akin to how we incorporate functionalities from other libraries when writing programs in TypeScript, Go, Rust, and other programming languages.

The example above, using the latest version from the official Helix Flake, illustrates this functionality. More use cases will be discussed later, and here are a few examples referenced for future mention:

- Getting Started with Home Manager: This introduces the community's Home-Manager as a dependency, enabling direct utilization of the features provided by this Flake.
- Downgrading or Upgrading Packages: Here, different versions of Nixpkgs are introduced as dependencies, allowing for flexible selection of packages from various versions of Nixpkgs.

More Flakes Tutorials

Up to this point, we have learned how to use Flakes to configure NixOS systems. If you have more questions about Flakes or want to learn more in-depth, please refer directly to the following official/semi-official documents:

- Nix Flakes's official documentation:
  - Nix flakes - Nix Manual
  - Flakes - nix.dev
- A series of tutorials by Eelco Dolstra(The creator of Nix) about Flakes:
  - Nix Flakes, Part 1: An introduction and tutorial (Eelco Dolstra, 2020)
  - Nix Flakes, Part 2: Evaluation caching (Eelco Dolstra, 2020)
  - Nix Flakes, Part 3: Managing NixOS systems (Eelco Dolstra, 2020)
- Other useful documents:
  - Practical Nix Flakes

Loading comments...