Module System and Custom Options

In our previous NixOS configurations, we set various values for `options` to configure NixOS or Home Manager. These `options` are actually defined in two locations:

- NixOS: nixpkgs/nixos/modules, where all NixOS options visible on https://search.nixos.org/options are defined.
- Home Manager: home-manager/modules, where you can find all its options at https://nix-community.github.io/home-manager/options.xhtml.

> If you are using nix-darwin too, its configuration is similar, and its module system is implemented in nix-darwin/modules.

The foundation of the aforementioned NixOS Modules and Home Manager Modules is a universal module system implemented in Nixpkgs, found in lib/modules.nix. The official documentation for this module system is provided below (even for experienced NixOS users, understanding this can be a challenging task):

- Module System - Nixpkgs

Because the documentation for Nixpkgs' module system is lacking, it directly recommends reading another writing guide specifically for NixOS module system, which is clearer but might still be challenging for newcomers:

- Writing NixOS Modules - Nixpkgs

In summary, the module system is implemented by Nixpkgs and is not part of the Nix package manager. Therefore, its documentation is not included in the Nix package manager's documentation. Additionally, both NixOS and Home Manager are based on Nixpkgs' module system implementation.

What is the Purpose of the Module System?

As ordinary users, using various options implemented by NixOS and Home Manager based on the module system is sufficient to meet most of our needs. So, what are the benefits of delving into the module system for us?

In the earlier discussion on modular configuration, the core idea was to split the configuration into multiple modules and then import these modules using `imports = [ ... ];`. This is the most basic usage of the module system. However, using only `imports = [ ... ];` allows us to import configurations defined in the module as they are without any customization, which limits flexibility. In simple configurations, this method is sufficient, but if the configuration is more complex, it becomes inadequate.

To illustrate the drawback, let's consider an example. Suppose I manage four NixOS hosts, A, B, C, and D. I want to achieve the following goals while minimizing configuration repetition:

- All hosts (A, B, C, and D) need to enable the Docker service and set it to start at boot.
- Host A should change the Docker storage driver to `btrfs` while keeping other settings the same.
- Hosts B and C, located in China, need to set a domestic mirror in Docker configuration.
- Host C, located in the United States, has no special requirements.
- Host D, a desktop machine, needs to set an HTTP proxy to accelerate Docker downloads.

If we purely use `imports`, we might have to split the configuration into several modules like this and then import different modules for each host:

bash

```
› tree
.
├── docker-default.nix  # Basic Docker configuration, including starting at boot
├── docker-btrfs.nix    # Imports docker-default.nix and changes the storage dri
├── docker-china.nix    # Imports docker-default.nix and sets a domestic mirror
└── docker-proxy.nix    # Imports docker-default.nix and sets an HTTP proxy
```

Doesn't this configuration seem redundant? This is still a simple example; if we have more machines with greater configuration differences, the redundancy becomes even more apparent.

Clearly, we need other means to address this redundant configuration issue, and customizing some of our own `options` is an excellent choice.

Before delving into the study of the module system, I emphasize once again that the following content is not necessary to learn and use. Many NixOS users have not

customized any `options` and are satisfied with simply using `imports` to meet their needs. If you are a newcomer, consider learning this part when you encounter problems that `imports` cannot solve. That's completely okay.

Basic Structure and Usage

The basic structure of modules defined in Nixpkgs is as follows:

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

{
  imports =
    [ # import other modules here
    ];

  options = {
    # ...
  };

  config = {
    # ...
  };
}
```

Among these, we are already familiar with `imports = [ ... ];` , but the other two parts are yet to be explored. Let's have a brief introduction here:

- `options = { ... };` : Similar to variable declarations in programming languages, it is used to declare configurable options.
- `config = { ... };` : Similar to variable assignments in programming languages, it is used to assign values to the options declared in `options` .

The most typical usage is to, within the same Nixpkgs module, set values for other `options` in `config = { .. };` based on the current values declared in `options = { ... };` . This achieves the functionality of parameterized configuration.

It's easier to understand with a direct example:

nix

```nix
# ./foo.nix
{ config, lib, pkgs, ... }:

with lib;

let
  cfg = config.programs.foo;
in {
  options.programs.foo = {
    enable = mkEnableOption "the foo program";

    package = mkOption {
      type = types.package;
      default = pkgs.hello;
      defaultText = literalExpression "pkgs.hello";
      description = "foo package to use.";
    };

    extraConfig = mkOption {
      default = "";
      example = ''
        foo bar
      '';
      type = types.lines;
      description = ''
        Extra settings for foo.
      '';
    };
  };

  config = mkIf cfg.enable {
    home.packages = [ cfg.package ];
    xdg.configFile."foo/foorc" = mkIf (cfg.extraConfig != "") {
      text = ''
        # Generated by Home Manager.

        ${cfg.extraConfig}
      '';
    };
  };
}
```

The module defined above introduces three `options`:

- `programs.foo.enable`: Used to control whether to enable this module.
- `programs.foo.package`: Allows customization of the `foo` package, such as using different versions, setting different compilation parameters, and so on.
- `programs.foo.extraConfig`: Used for customizing the configuration file of `foo`.

Then, in the `config` section, based on the values declared in these three variables in `options`, different settings are applied:

- If `programs.foo.enable` is `false` or undefined, no settings are applied.
  - This is achieved using `lib.mkIf`.
- Otherwise,
  - Add `programs.foo.package` to `home.packages` to install it in the user environment.
  - Write the value of `programs.foo.extraConfig` to `~/.config/foo/foorc`.

This way, we can import this module in another Nix file and achieve custom configuration for `foo` by setting the `options` defined here. For example:

nix

```nix
# ./bar.nix
{ config, lib, pkgs, ... }:

{
  imports = [
    ./foo.nix
  ];

  programs.foo ={
    enable = true;
    package = pkgs.hello;
    extraConfig = ''
      foo baz
    '';
  };
}
```

In the example above, the way we assign values to `options` is actually a kind of **abbreviation**. When a module only contains `config` without any other declaration (like `option` and other special parameters of the module system), we can omit the `config`

wrapping , just directly write the content of `config` to assign value to `option` section declared in other modules!

Assignment and Lazy Evaluation in the Module System

The module system takes full advantage of Nix's lazy evaluation feature, which is crucial for achieving parameterized configuration.

Let's start with a simple example:

nix

```nix
# ./flake.nix
{
  description = "NixOS Flake for Test";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";

  outputs = {nixpkgs, ...}: {
    nixosConfigurations = {
      "test" = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ({config, lib, ...}: {
            options = {
              foo = lib.mkOption {
                default = false;
                type = lib.types.bool;
              };
            };

            # Scenario 1 (works fine)
            config.warnings = if config.foo then ["foo"] else [];

            # Scenario 2 (error: infinite recursion encountered)
            # config = if config.foo then { warnings = ["foo"];} else {};

            # Scenario 3 (works fine)
            # config = lib.mkIf config.foo {warnings = ["foo"];};
          })
        ];
      };
    };

```

```
32      };
    }
```

In the examples 1, 2, and 3 of the above configuration, the value of `config.warnings` depends on the value of `config.foo`, but their implementation methods are different. Save the above configuration as `flake.nix`, and then use the command `nix eval .#nixosConfigurations.test.config.warnings` to test examples 1, 2, and 3 separately. You will find that examples 1 and 3 work correctly, while example 2 results in an error: `error: infinite recursion encountered`.

Let's explain each case:

1. Example 1 evaluation flow: `config.warnings` ⇒ `config.foo` ⇒ `config`
    1. First, Nix attempts to compute the value of `config.warnings` but finds that it depends on `config.foo`.
    2. Next, Nix tries to compute the value of `config.foo`, which depends on its outer `config`.
    3. Nix attempts to compute the value of `config`, and since the contents not genuinely used by `config.foo` are lazily evaluated by Nix, there is no recursive dependency on `config.warnings` at this point.
    4. The evaluation of `config.foo` is completed, followed by the assignment of `config.warnings`, and the computation ends.
2. Example 2: `config` ⇒ `config.foo` ⇒ `config`
    1. Initially, Nix tries to compute the value of `config` but finds that it depends on `config.foo`.
    2. Next, Nix attempts to compute the value of `config.foo`, which depends on its outer `config`.
    3. Nix tries to compute the value of `config`, and this loops back to step 1, leading to an infinite recursion and eventually an error.
3. Example 3: The only difference from example 2 is the use of `lib.mkIf` to address the infinite recursion issue.

The key lies in the function `lib.mkIf`. When using `lib.mkIf` to define `config`, it will be lazily evaluated by Nix. This means that the calculation of `config = lib.mkIf ...` will only occur after the evaluation of `config.foo` is completed.

The Nixpkgs module system provides a series of functions similar to `lib.mkIf` for parameterized configuration and intelligent module merging:

1. `lib.mkIf`：Already introduced.
2. `lib.mkOverride` / `lib.mkDefault` / `lib.mkForce`：Previously discussed in Modularizing NixOS Configuration.
3. `lib.mkOrder`, `lib.mkBefore`, and `lib.mkAfter`：As mentioned above.
4. Check Option Definitions - NixOS for more functions related to option assignment (definition).

Option Declaration and Type Checking

While assignment is the most commonly used feature of the module system, if you need to customize some `options`, you also need to delve into option declaration and type checking. I find this part relatively straightforward; it's much simpler than assignment, and you can understand the basics by directly referring to the official documentation. I won't go into detail here.

- Option Declarations - NixOS
- Options Types - NixOS

Passing Non-default Parameters to the Module System

We have already introduced how to use `specialArgs` and `_module.args` to pass additional parameters to other Modules functions in Managing Your NixOS with Flakes. No further elaboration is needed here.

How to Selectively Import Modules

In the examples above, we have introduced how to enable or disable certain features through custom options. However, our code implementations are all within the same Nix file. If our modules are scattered across different files, how can we achieve selective import?

Let's first look at some common incorrect usage patterns, and then introduce the correct way to do it.

Incorrect Usage #1 - Using `imports` in `config = { ... };`

The first thought might be to directly use `imports` in `config = { ... };`, like this:

nix

```nix
# ./flake.nix
{
  description = "NixOS Flake for Test";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";

  outputs = {nixpkgs, ...}: {
    nixosConfigurations = {
      "test" = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ({config, lib, ...}: {
            options = {
              foo = lib.mkOption {
                default = false;
                type = lib.types.bool;
              };
            };
            config = lib.mkIf config.foo {
              # Using imports in config will cause an error
              imports = [
                {warnings = ["foo"];}
                # ...omit other module or file paths
              ];
            };
          })
        ];
      };
    };
  };
}
```

But this won't work. You can try save the above `flake.nix` in a new directory, and then run `nix eval .#nixosConfigurations.test.config.warnings` in it, some error like `error: The option 'imports' does not exist.` will be encountered.

This is because `config` is a regular attribute set, while `imports` is a special parameter of the module system. There is no such definition as `config.imports`.

Correct Usage #1 - Define Individual `options` for All Modules That Require Conditional Import

This is the most recommended method. Modules in NixOS systems are implemented in this way, and searching for `enable` in https://search.nixos.org/options will show a large number of system modules that can be enabled or disabled through the `enable` option.

The specific writing method has been introduced in the previous Basic Structure and Usage section and will not be repeated here.

The disadvantage of this method is that all Nix modules that require conditional import need to be modified, moving all configuration declarations in the module to the `config = { ... };` code block, increasing code complexity and being less friendly to beginners.

Correct Usage #2 - Use `lib.optionals` in `imports = [];`

The main advantage of this method is that it is much simpler than the methods previously introduced, requiring no modification to the module content, just using `lib.optionals` in `imports` to decide whether to import a module or not.

> Details about how `lib.optionals` works: https://noogle.dev/f/lib/optionals

Let's look at an example directly:

nix

```nix
# ./flake.nix
{
  description = "NixOS Flake for Test";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";

  outputs = {nixpkgs, ...}: {
    nixosConfigurations = {
      "test" = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        specialArgs = { enableFoo = true; };
        modules = [
          ({config, lib, enableFoo ? false, ...}: {
            imports =

```

```
                [
                   # Other Modules
                ]
                # Use lib.optionals to decide whether to import foo.nix
                ++ (lib.optionals (enableFoo) [./foo.nix]);
            })
          ];
        };
      };
    };
  }
```

nix

```
# ./foo.nix
{ warnings = ["foo"];}
```

Save the two Nix files above in a folder, and then run `nix eval .#nixosConfigurations.test.config.warnings` in the folder, and the operation is normal:

bash

```
› nix eval .#nixosConfigurations.test.config.warnings
[ "foo" ]
```

One thing to note here is that **you cannot use parameters passed by `_module.args` in `imports =[ ... ];`** . We have already provided a detailed explanation in the previous section Passing Non-default Parameters to Submodules.

References

- Best resources for learning about the NixOS module system? - Discourse
- NixOS modules - NixOS Wiki
- NixOS: config argument - NixOS Wiki
- Module System - Nixpkgs
- Writing NixOS Modules - Nixpkgs

Loading comments...