Modularize Your NixOS Configuration

At this point, the skeleton of the entire system is configured. The current configuration structure in `/etc/nixos` should be as follows:

```
$ tree
.
├── flake.lock
├── flake.nix
├── home.nix
└── configuration.nix
```

The functions of these four files are:

- `flake.lock` : An automatically generated version-lock file that records all input sources, hash values, and version numbers of the entire flake to ensure reproducibility.
- `flake.nix` : The entry file that will be recognized and deployed when executing `sudo nixos-rebuild switch` . See Flakes - NixOS Wiki for all options of flake.nix.
- `configuration.nix` : Imported as a Nix module in flake.nix, all system-level configuration is currently written here. See Configuration - NixOS Manual for all options of configuration.nix.
- `home.nix` : Imported by Home-Manager as the configuration of the user `ryan` in flake.nix, containing all of `ryan` 's configuration and managing `ryan` 's home folder. See Appendix A. Configuration Options - Home-Manager for all options of home.nix.

By modifying these files, you can declaratively change the system and home directory status.

However, as the configuration grows, relying solely on `configuration.nix` and `home.nix` can lead to bloated and difficult-to-maintain files. A better solution is to use the Nix module system to split the configuration into multiple Nix modules and write them in a classified manner.

The Nix language provides an import function with a special rule:

> If the parameter of `import` is a folder path, it will return the execution result of the `default.nix` file in that folder.

The Nixpkgs module system provides a similar parameter, `imports`, which accepts a list of `.nix` files and **merge** all the configuration defined in these files into the current Nix module.

Note that `imports` will not simply overwrite duplicate configuration but handle it more reasonably. For example, if `program.packages = [...]` is defined in multiple modules, then `imports` will merge all `program.packages` defined in all Nix modules into one list. Attribute sets can also be merged correctly. The specific behavior can be explored by yourself.

> I only found a description of `imports` in Nixpkgs-Unstable Official Manual - evalModules Parameters: `A list of modules. These are merged together to form the final configuration.` It's a bit ambiguous...

With the help of `imports`, we can split `home.nix` and `configuration.nix` into multiple Nix modules defined in different `.nix` files. Lets look at an example module `packages.nix`:

```nix
{
  config,
  pkgs,
  ...
}: {
  imports = [
    (import ./special-fonts-1.nix {inherit config pkgs;}) # (1)
    ./special-fonts-2.nix # (2)
  ];

  fontconfig.enable = true;
}
```

This module loads two other modules in the imports section, namely `special-fonts-1.nix` and `special-fonts-2.nix`. Both files are modules themselves and look similar to this.

```nix
{ config, pkgs, ...}: {
    # Configuration stuff ...
}
```

Both import statements above are equivalent in the parameters they receive:

- Statement `(1)` imports the function in `special-fonts-1.nix` and calls it by passing `{config = config; pkgs = pkgs}` . Basically using the return value of the call (another partial configuration [attritbute set]) inside the `imports` list.
- Statement `(2)` defines a path to a module, whose function Nix will load *automatically* when assembling the configuration `config` . It will pass all matching arguments from the function in `packages.nix` to the loaded function in `special-fonts-2.nix` which results in `import ./special-fonts-2.nix {config = config; pkgs = pkgs}` .

Here is a nice starter example of modularizing the configuration, Highly recommended:

- Misterio77/nix-starter-configs

A more complicated example, ryan4yin/nix-config/i3-kickstarter is the configuration of my previous NixOS system with the i3 window manager. Its structure is as follows:

```shell
├── flake.lock
├── flake.nix
├── home
│   ├── default.nix         # here we import all submodules by imports = [...]
│   ├── fcitx5              # fcitx5 input method's configuration
│   │   ├── default.nix
│   │   └── rime-data-flypy
│   ├── i3                  # i3 window manager's configuration
│   │   ├── config
│   │   ├── default.nix
│   │   ├── i3blocks.conf
│   │   ├── keybindings
│   │   └── scripts
│   ├── programs
│   │   ├── browsers.nix
│   │   ├── common.nix
│   │   ├── default.nix   # here we import all modules in programs folder by imp
│   │   ├── git.nix
│   │   ├── media.nix
│   │   ├── vscode.nix
│   │   └── xdg.nix
│   ├── rofi              #  rofi launcher's configuration
│   │   ├── configs
│   │   │   ├── arc_dark_colors.rasi
│   │   │   ├── arc_dark_transparent_colors.rasi
│   │   │   ├── power-profiles.rasi
│   │   │   ├── powermenu.rasi
│   │   │   ├── rofidmenu.rasi
│   │   │   └── rofikeyhint.rasi
│   │   └── default.nix
│   └── shell             # shell/terminal related configuration
│       ├── common.nix
│       ├── default.nix
│       ├── nushell
│       │   ├── config.nu
│       │   ├── default.nix
│       │   └── env.nu
│       ├── starship.nix
│       └── terminals.nix
├── hosts
│   ├── msi-rtx4090      # My main machine's configuration
│   │   ├── default.nix  # This is the old configuration.nix, but most of the co
│   │   └── hardware-configuration.nix  # hardware & disk related configuration,
│   └── my-nixos       # my test machine's configuration
│       ├── default.nix
│       └── hardware-configuration.nix
├── modules          # some common NixOS modules that can be reused
│   ├── i3.nix
│   └── system.nix
└── wallpaper.jpg    # wallpaper
```

There is no need to follow the above structure, you can organize your configuration in any way you like. The key is to use `imports` to import all the submodules into the main module.

`lib.mkOverride`, `lib.mkDefault`, and `lib.mkForce`

In Nix, some people use `lib.mkDefault` and `lib.mkForce` to define values. These functions are designed to set default values or force values of options.

You can explore the source code of `lib.mkDefault` and `lib.mkForce` by running `nix repl -f '<nixpkgs>'` and then entering `:e lib.mkDefault`. To learn more about `nix repl`, type `:?` for the help information.

Here's the source code:

```nix
  # ......

  mkOverride = priority: content:
    { _type = "override";
      inherit priority content;
    };

  mkOptionDefault = mkOverride 1500; # priority of option defaults
  mkDefault = mkOverride 1000; # used in config sections of non-user modules to
  mkImageMediaOverride = mkOverride 60; # image media profiles can be derived by
  mkForce = mkOverride 50;
  mkVMOverride = mkOverride 10; # used by ‘nixos-rebuild build-vm’

  # ......
```

In summary, `lib.mkDefault` is used to set default values of options with a priority of 1000 internally, and `lib.mkForce` is used to force values of options with a priority of 50 internally. If you set a value of an option directly, it will be set with a default priority of 1000, the same as `lib.mkDefault` .

The lower the `priority` value, the higher the actual priority. As a result, `lib.mkForce` has a higher priority than `lib.mkDefault` . If you define multiple values with the same priority, Nix will throw an error.

Using these functions can be very helpful for modularizing the configuration. You can set default values in a low-level module (base module) and force values in a high-level module.

For example, in my configuration at ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-server.nix, I define default values like this:

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

{
  # ......

  nixpkgs.config.allowUnfree = lib.mkDefault false;

  # ......
}
```

Then, for my desktop machine, I override the value in ryan4yin/nix-config/blob/c515ea9/modules/nixos/core-desktop.nix like this:

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

{
  # import the base module
  imports = [
    ./core-server.nix
  ];

  # override the default value defined in the base module
  nixpkgs.config.allowUnfree = lib.mkForce true;

  # ......
}
```

`lib.mkOrder` , `lib.mkBefore` , and `lib.mkAfter`

In addition to `lib.mkDefault` and `lib.mkForce` , there are also `lib.mkBefore` and `lib.mkAfter` , which are used to set the merge order of **list-type options**. These functions further contribute to the modularization of the configuration.

> I haven't found the official documentation for list-type options, but I simply understand that they are types whose merge results are related to the order of merging. According to this understanding, both `list` and `string` types are list-type options, and these functions can indeed be used on these two types in practice.

As mentioned earlier, when you define multiple values with the same **override priority**, Nix will throw an error. However, by using `lib.mkOrder` , `lib.mkBefore` , or `lib.mkAfter` , you can define multiple values with the same override priority, and they will be merged in the order you specify.

To examine the source code of `lib.mkBefore`, you can run `nix repl -f '<nixpkgs>'` and then enter `:e lib.mkBefore`. To learn more about `nix repl`, type `:?` for the help information:

```nix
  # ......

  mkOrder = priority: content:
    { _type = "order";
      inherit priority content;
    };

  mkBefore = mkOrder 500;
  defaultOrderPriority = 1000;
  mkAfter = mkOrder 1500;

  # ......
```

Therefore, `lib.mkBefore` is a shorthand for `lib.mkOrder 500`, and `lib.mkAfter` is a shorthand for `lib.mkOrder 1500`.

To test the usage of `lib.mkBefore` and `lib.mkAfter`, let's create a simple Flake project:

```nix
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-24.11";
  outputs = {nixpkgs, ...}: {
    nixosConfigurations = {
      "my-nixos" = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";

        modules = [
          ({lib, ...}: {
            programs.bash.shellInit = lib.mkBefore ''
              echo 'insert before default'
            '';
            programs.zsh.shellInit = lib.mkBefore "echo 'insert before default';
            nix.settings.substituters = lib.mkBefore [
              "https://nix-community.cachix.org"
            ];
          })

          ({lib, ...}: {
            programs.bash.shellInit = lib.mkAfter ''
              echo 'insert after default'
            '';
            programs.zsh.shellInit = lib.mkAfter "echo 'insert after default';";
            nix.settings.substituters = lib.mkAfter [
              "https://ryan4yin.cachix.org"
            ];
          })

          ({lib, ...}: {
            programs.bash.shellInit = ''
              echo 'this is default'
            '';
            programs.zsh.shellInit = "echo 'this is default';";
            nix.settings.substituters = [
              "https://nix-community.cachix.org"
            ];
          })
        ];
      };
    };
  };
}
```

The flake above contains the usage of `lib.mkBefore` and `lib.mkAfter` on multiline strings, single-line strings, and lists. Let's test the results:

```bash
# Example 1: multiline string merging
› echo $(nix eval .#nixosConfigurations.my-nixos.config.programs.bash.shellInit)
trace: warning: system.stateVersion is not set, defaulting to 24.11. Read why th
n.
"echo 'insert before default'

echo 'this is default'

if [ -z \"$__NIXOS_SET_ENVIRONMENT_DONE\" ]; then
 . /nix/store/60882lm9znqdmbssxqsd5bgnb7gybaf2-set-environment
fi



```

```

echo 'insert after default'
"

# example 2: single-line string merging
› echo $(nix eval .#nixosConfigurations.my-nixos.config.programs.zsh.shellInit)
"echo 'insert before default';
echo 'this is default';
echo 'insert after default';"

# Example 3: list merging
› nix eval .#nixosConfigurations.my-nixos.config.nix.settings.substituters
[ "https://nix-community.cachix.org" "https://nix-community.cachix.org" "https:/
```

As you can see, `lib.mkBefore` and `lib.mkAfter` can define the order of merging of multiline strings, single-line strings, and lists. The order of merging is the same as the order of definition.

> For a deeper introduction to the module system, see Module System & Custom Options.

References

- Nix modules: Improving Nix's discoverability and usability
- Module System - Nixpkgs

Loading comments...