Remote Deployment

Nix's inherent design is well-suited for remote deployment, and the Nix community offers several tools tailored for this purpose, such as NixOps and colmena. Additionally, the official tool we've used extensively, `nixos-rebuild`, possesses some remote deployment capabilities too.

In addition, within multi-architecture scenarios, remote deployment can optimally leverage Nix's multi-architecture support. For example, you can cross-compile an aarch64/riscv64 NixOS system on an x86_64 host, followed by remote deployment onto the corresponding hosts via SSH.

Recently, I encountered a situation where I cross-compiled a NixOS system image for a RISCV64 SBC on my local machine. Consequently, I already possessed all the compilation caches for building this system locally. However, due to the lack of official binary caches for RISCV64 architecture, executing any uninstalled program directly on the development board (e.g., `nix run nixpkgs#cowsay hello`) triggered extensive compilations. This process consumed hours, which was quite unacceptable.

By adopting remote deployment, I could fully harness the computational power of my local high-performance CPU and the extensive compilation caches. This switch vastly improved my experience and significantly mitigated the previously time-consuming compilation issue.

Let me briefly guide you through using colmena or `nixos-rebuild` for remote deployment.

Prerequisites

Before embarking on remote deployment, a few preparatory steps are necessary:

1. To prevent remote host's sudo password verification failure, choose one of the following methods:
    1. Deploy as the remote host's `root` user.
    2. Add `security.sudo.wheelNeedsPassword = false;` to the remote host's configuration and manually deploy once in advance to grant the user passwordless sudo permissions..

    1. **This will allow user-level programs to silently obtain sudo permissions, posing a security risk**! Therefore, if you choose this method, it's advisable to create a dedicated user for remote deployment, rather than using your regular user account!
2. Configure SSH public key authentication for the remote hosts.
    1. Use the `users.users.<name>.openssh.authorizedKeys.keys` option to complete this task.
3. Add the remote host's Known Hosts record to your local machine. Otherwise, colmena/nixos-rebuild will fail to deploy due to the inability to verify the remote host's identity.
    1. Use the `programs.ssh.knownHosts` option to add the remote host's public key to the Known Hosts record.
4. Manually use the `ssh root@<you-host>` command to verify that you can login to the remote host.
    1. If you encounter any issues, resolve them before proceeding.

It's advisable to use the `root` user for deployment as it's more convenient and avoids the complexities of sudo permissions.

Assuming we intend to deploy remotely using the root user, the initial step involves configuring SSH public key authentication for the root user on the remote host. To accomplish this, simply add the following content to any NixOS Module in the remote host's Nix configuration (e.g., `configuration.nix`), then rebuild the system:

nix

```nix
# configuration.nix
{

  # ...

  users.users.root.openssh.authorizedKeys.keys = [
    # TODO Replace with your own SSH public key.
    "ssh-ed25519 AAAAC3Nxxxxx ryan@xxx"
  ];

  # ...
}
```

Furthermore, you'll need to add the SSH private key to the SSH agent on your local machine for authentication during remote deployment:

```bash
ssh-add ~/.ssh/your-private-key
```

Deploy through `colmena`

`colmena` doesn't directly use the familiar `nixosConfigurations.xxx` for remote deployment. Instead, it customizes a flake outputs named `colmena`. Although its structure is similar to `nixosConfigurations.xxx`, it's not identical.

In your system's `flake.nix`, add a new outputs named `colmena`. A simple example is shown below:

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

    # ...
  };
  outputs = { self, nixpkgs }: {
    # ...

    # Add this output, colmena will read its contents for remote deployment
    colmena = {
      meta = {
        nixpkgs = import nixpkgs { system = "x86_64-linux"; };

        # This parameter functions similarly to `specialArgs` in `nixosConfigura
        # used for passing custom arguments to all submodules.
        specialArgs = {
          inherit nixpkgs;
        };
      };

      # Host name = "my-nixos"
      "my-nixos" = { name, nodes, ... }: {
        # Parameters related to remote deployment
        deployment.targetHost = "192.168.5.42"; # Remote host's IP address
        deployment.targetUser = "root";  # Remote host's username

```

```
          # This parameter functions similarly to `modules` in `nixosConfiguration
          # used for importing all submodules.
          imports = [
            ./configuration.nix
          ];
        };
      };
    };
}
```

Now you can deploy your configuration to the device:

```bash
nix run nixpkgs#colmena apply
```

For more advanced usage, refer to colmena's official documentation at https://colmena.cli.rs/unstable/introduction.html

---

Deploy through `nixos-rebuild`

Using `nixos-rebuild` for remote deployment has the advantage of being similar to deploying to a local host. It only requires a few additional parameters to specify the remote host's IP address, username, and other details.

For instance, to deploy the configuration defined in the `nixosConfigurations.my-nixos` of your flake to a remote host, use the following command:

```bash
nixos-rebuild switch --flake .#my-nixos \
  --target-host root@192.168.4.1 --build-host localhost --verbose
```

The above command will build and deploy the configuration of `my-nixos` to a server with IP `192.168.4.1`. The system build process will occur locally.

If you prefer to build the configuration on the remote host, replace `--build-host localhost` with `--build-host root@192.168.4.1`.

To avoid repeatedly using IP addresses, you can define host aliases in your local machine's `~/.ssh/config` or `/etc/ssh/ssh_config`. For example:

> Generating the SSH configuration entirely through Nix configuration is possible, and this task is left to you.

```bash
› cat ~/.ssh/config

# ......

Host aquamarine
  HostName 192.168.4.1
  Port 22

# ......
```

With this setup, you can use host aliases for deployment:

```bash
nixos-rebuild switch --flake .#my-nixos --target-host root@aquamarine --build-ho
```

This offers a more convenient way to deploy using the defined host aliases.

Loading comments...