Nix Store and Binary Cache

Here we provide a brief introduction to the Nix Store, Nix binary cache, and related concepts, without delving into specific configurations and usage methods, which will be covered in detail in subsequent chapters.

Nix Store

The Nix Store is one of the core concepts of the Nix package manager. It is a read-only file system used to store all files that require immutability, including the build results of software packages, metadata of software packages, and all build inputs of software packages.

The Nix package manager uses the Nix functional language to describe software packages and their dependencies. Each software package is treated as the output of a pure function, and the build results of the software package are stored in the Nix Store.

Data in the Nix Store has a fixed path format:

```
/nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1
|--------| |------------------------------| |----------|
store directory         digest                  name
```

As seen, paths in the Nix Store start with a hash value (digest), followed by the name and version number of the software package. This hash value is calculated based on all input information of the software package (build parameters, dependencies, dependency versions, etc.), and any changes in build parameters or dependencies will result in a change in the hash value, thus ensuring the uniqueness of each software package path. Additionally, since the Nix Store is a read-only file system, it ensures the immutability of software packages - once a software package is built, it will not change.

Because the storage path of the build result is calculated based on all input information of the build process, **the same input information will yield the same storage path**. This design is also known as the *Input-addressed Model*.

How NixOS Uses the Nix Store

NixOS's declarative configuration calculates which software packages need to be installed and then soft-links the storage paths of these packages in the Nix Store to `/run/current-system`, and by modifying environment variables like `PATH` to point to the corresponding folder in `/run/current-system`, the installation of software packages is achieved. Each time a deployment is made, NixOS calculates the new system configuration, cleans up old symbolic links, and re-creates new symbolic links to ensure that the system environment matches the declarative configuration.

home-manager works similarly, soft-linking the software packages configured by the user to `/etc/profiles/per-user/your-username` and modifying environment variables like `PATH` to point to this path, thus installing user software packages.

bash

```
# Check where bash in the environment comes from (installed using NixOS)
› which bash
┌───┬─────────┬──────────────────────────────┬──────────┐
│ # │ command │             path             │   type   │
├───┼─────────┼──────────────────────────────┼──────────┤
│ 0 │ bash    │ /run/current-system/sw/bin/bash │ external │
└───┴─────────┴──────────────────────────────┴──────────┘

› ls -al /run/current-system/sw/bin/bash
lrwxrwxrwx 15 root root 76 1970年 1月 1日 /run/current-system/sw/bin/bash -> /ni

# Check where cowsay in the environment comes from (installed using home-manager
› which cowsay
┌───┬─────────┬──────────────────────────────────────┬──────────┐
│ # │ command │                 path                 │   type   │
├───┼─────────┼──────────────────────────────────────┼──────────┤
│ 0 │ cowsay  │ /etc/profiles/per-user/ryan/bin/cowsay │ external │
└───┴─────────┴──────────────────────────────────────┴──────────┘

› ls -al /etc/profiles/per-user/ryan/bin/cowsay
lrwxrwxrwx 2 root root 72 1970年 1月 1日 /etc/profiles/per-user/ryan/bin/cowsay -
```

The `nix develop` command, on the other hand, directly adds the storage paths of software packages to environment variables like `PATH` and `LD_LIBRARY_PATH`, enabling the newly created shell environment to directly use these software packages or libraries.

For example, in the source code repository for this book, [ryan4yin/nixos-and-flakes-book](), after executing the `nix develop` command, we can examine the contents of the `PATH` environment variable:

```bash
› nix develop
node v20.9.0
› env | egrep '^PATH'
PATH=/nix/store/h13fnmpm8m28qypsba2xysi8a90crphj-pre-commit-3.6.0/bin:/nix/store
```

Clearly, `nix develop` has added the storage paths of many software packages directly to the `PATH` environment variable.

Nix Store Garbage Collection

The Nix Store is a centralized storage system where all software package build inputs and outputs are stored. As the system is used, the number of software packages in the Nix Store will increase, and the disk space occupied will grow larger.

To prevent the Nix Store from growing indefinitely, the Nix package manager provides a garbage collection mechanism for the local Nix Store, to clean up old data and reclaim storage space.

According to Chapter 11. The Garbage Collector - nix pills, the `nix-store --gc` command performs garbage collection by recursively traversing all symbolic links in the `/nix/var/nix/gcroots/` directory to find all referenced packages and delete those that are no longer referenced. The `nix-collect-garbage --delete-old` command goes a step further by first deleting all old profiles and then running the `nix-store --gc` command to clean up packages that are no longer referenced.

It's important to note that build results from commands like `nix build` and `nix develop` are not automatically added to `/nix/var/nix/gcroots/` , so these build results may be cleaned up by the garbage collection mechanism. You can use `nix-instantiate` with `keep-outputs = true` and other means to avoid this, but I currently prefer setting up your own binary cache server and configuring a longer cache time (e.g., one year), then pushing data to the cache server. This way, you can share build results across machines

and avoid having local build results cleaned up by the local garbage collection mechanism, achieving two goals in one.

Binary Cache

The design of Nix and the Nix Store ensures the immutability of software packages, allowing build results to be shared directly between multiple machines. As long as these machines use the same input information to build a package, they will get the same output path, and Nix can reuse the build results from other machines instead of rebuilding the package, thus speeding up the installation of software packages.

The Nix binary cache is designed based on this feature; it is an implementation of the Nix Store that stores data on a remote server instead of locally. When needed, the Nix package manager downloads the corresponding build results from the remote server to the local `/nix/store`, avoiding the time-consuming local build process.

Nix provides an official binary cache server at https://cache.nixos.org, which caches build results for most packages in nixpkgs for common CPU architectures. When you execute a Nix build command on your local machine, Nix first attempts to find the corresponding binary cache on the cache server. If found, it will directly download the cache file, bypassing the time-consuming local compilation and greatly accelerating the build process.

Nix Binary Cache Trust Model

The **Input-addressed Model** only guarantees that the same input will produce the same output path, but it does not ensure the uniqueness of the output content. This means that even with the same input information, multiple builds of the same software package may produce different output content.

While Nix has taken measures such as disabling network access in the build environment and using fixed timestamps to minimize uncertainty, there are still some uncontrollable factors that can influence the build process and produce different output content. These differences in output content typically do not affect the functionality of the software package but do pose a challenge for the secure sharing of binary cache - the uncertainty in output content makes it difficult to determine whether the binary cache downloaded

from the cache server was indeed built with the declared input information, and whether it contains malicious content.

To address this, the Nix package manager uses a public-private key signing mechanism to verify the source and integrity of the binary cache. This places the responsibility of security on the user. If you wish to use a non-official cache server to speed up the build process, you must add the public key of that server to `trusted-public-keys` and assume the associated security risks - the cache server might provide cached data that includes malicious content.

Content-addressed Model

RFC062 - content-addressed store paths is an attempt by the community to improve build result consistency. It proposes a new way to calculate storage paths based on the build results (outputs) rather than the input information (inputs). This design ensures consistency in build results - if the build results are different, the storage paths will also be different, thus avoiding the uncertainty in output content inherent in the input-addressed model.

However, this approach is still in an experimental stage and has not been widely adopted.

References

- Nix Store - Nix Manual
