Nix Package Manager CLI (Nix From First Principles: Flake Edition #3)

Oct 05
2022

#Nix

This is part 3 of the Nix from First Principles: Flake Edition series.

The foundational part of Nix is the Nix package manager. This is a tool that allows you to install packages. The "first layer" of Nix usage is to use this tool to install packages. The Nix package manager, similar to projects like os-rpmtree (used in Fedora Silverblue) or snapper (used in snap based Linux distros), will allow you to easily revert package installs or updates if something goes wrong.

This post covers the basics of using the nix package manager at the command line - while most users use nix through more advanced workflows, the basics explained here will be useful in understanding those workflows, as well as for users who want to dip their toes into Nix by treating it like a more traditional package manager to start out.

To install your first package, run the `nix profile install` command. The package used in this demonstration is `ripgrep`, which is a rust based grep alternative that has a nicer out of the box experince.

```
nix profile install nixpkgs#ripgrep
```

This command tells `nix` to install the `ripgrep` package from the `nixpkgs` registry into the default profile. If you followed the installer you should now be able to run the `rg` binary from the `ripgrep` package.

Let's install a few more packages, like `jq` the CLI tool for JSON and `fzf` a fuzzy finder that lets you search a list of CLI arguments and select one.

```
nix profile install nixpkgs#fzf nixpkgs#jq
```

Now you can observe what it set up. By default, `nix profile` will set itself up in your `~/.nix-profile` directory.

```
> ls -l ~/.nix-profile/
total 4
dr-xr-xr-x 1 root root  44 Jan  1  1970 bin/
-r--r--r-- 1 root root 500 Jan  1  1970 manifest.json
dr-xr-xr-x 1 root root  78 Jan  1  1970 share/

> ls -l ~/.nix-profile/bin/
total 20
lrwxrwxrwx 1 root root 62 Jan  1  1970 fzf -> /nix/store/bds5wjz9j
lrwxrwxrwx 1 root root 68 Jan  1  1970 fzf-share → /nix/store/bds5
lrwxrwxrwx 1 root root 67 Jan  1  1970 fzf-tmux → /nix/store/bds5w
lrwxrwxrwx 1 root root 61 Jan  1  1970 jq → /nix/store/348rj7g1gg2
lrwxrwxrwx 1 root root 65 Jan  1  1970 rg → /nix/store/cmzib5qfpqs

> ls -l ~/.nix-profile/share/
total 20
lrwxrwxrwx 1 root root  80 Jan  1  1970 bash-completion → /nix/sto
dr-xr-xr-x 1 root root 102 Jan  1  1970 fish/
lrwxrwxrwx 1 root root  64 Jan  1  1970 fzf → /nix/store/bds5wjz9
lrwxrwxrwx 1 root root  68 Jan  1  1970 man → /nix/store/cmzib5qfp
lrwxrwxrwx 1 root root  72 Jan  1  1970 vim-plugins → /nix/store/b
lrwxrwxrwx 1 root root  68 Jan  1  1970 zsh → /nix/store/cmzib5qfp
```

First, note that `~/.nix-profile` is the location of your user's default profile. Every user can have their own independent profile with their own set of installed packages. This doesn't involve duplicating packages if multiple users have them installed, as the multi-user installation will have these all symlink back to your global `/nix` directory.

If you have performed the default installation and you're using a supported shell, then `~/.nix-profile/bin` should already be in your path. If not, you can add it if you want to have your commands from your nix profile available at all times.

What's with 1970?

The timestamps in the files created by Nix are all 1970. This is because one of the goals of the Nix ecosystem is to have reproducible packages. One major area of non-reproducability is when the build time gets added to an artifact, as this will be different each time a package is built. To ensure reproducability, Nix sets all the timestamps to the start of the Unix epoch so they are the same regardless of when you build a package.

Once that's done, let's try and use some of those commands.

```
jq -r '.elements[].attrPath' ~/.nix-profile/manifest.json \
| fzf --preview 'jq --arg attr {} ".elements[] | select(.attrPath =
```

You should get a window which lists the three installed packages according to the manifest json file and if you pick an item with arrow keys or typing, it should show you the object from your nix profile that describes the installed package.

Of course, while rooting around in the `manifest.json` files works nicely as a demonstration of the newly installed `jq` package, you don't need to do this to get a package listing. Instead you can use the `nix profile list` command, which will produce output like the following:

```
> nix profile list
0 flake:nixpkgs#legacyPackages.x86_64-linux.ripgrep github:NixOS/ni
1 flake:nixpkgs#legacyPackages.x86_64-linux.fzf github:NixOS/nixpkg
```

```
flake:nixpkgs#legacyPackages.x86_64-linux.fzf github:NixOS/nixpkg
flake:nixpkgs#legacyPackages.x86_64-linux.jq github:NixOS/nixpkgs
```

The fields here are:

1. A number referencing the installation order. This can be used as shorthand when installing or uninstalling the package, similar to a short hash in git.
2. The expanded version of the reference used to install the package. This is used by Nix when it needs to upgrade the package, or by users if they want to have something that's independent of installation order. This is also known as the *mutable flake reference*.
3. The full version of the reference when it is resolved. This means that not only will it refer to conceptually the same package, but also the same version of the same package. As a result, it's known as the *immutable flake reference*, as it will always get the same result, while the *mutable flake reference* can change as you upgrade your `nixpkgs` version.
4. The location on disk in the nix store the package was stored in.

This `list` output may seem complicated, but it's actually one of the big improvements made to the `nix profile` command over its predecessor, `nix-env`. `nix-env` was stuck doing a text based name comparison to find the same package to perform an upgrade, which very often did the wrong thing, along with being pretty slow.

Finally, you can update a specific package or all packages.

To update a specific package, you can use `nix profile upgrade <pkg reference>`. The package reference can be one of the numbers from `nix profile list`, or the *mutable flake reference* in either the short version used when installing it or the expanded version in `nix profile list`. To upgrade all packages, you can use `nix profile upgrade '.*'`.

Generations

So far, I've just covered how to do the things with nix that you can do with *any* package manager, and they seem a little more convoluted too, so now it's time to get into what you can do with Nix that you can't with `apt`, `brew` or `pacman`.

The first is that Nix keeps multiple versions of installed packages, and a record of the changes made. This allows you to easily rollback package management commands. Each version of your profile after you perform a package manager operation is known as a generation.

You can observe this log of history with the `nix profile history` command:

```
> nix profile history
Version 1 (2022-10-05):
  flake:nixpkgs#legacyPackages.x86_64-linux.ripgrep: ∅ → 13.0.0

Version 2 (2022-10-05) ← 1:
  flake:nixpkgs#legacyPackages.x86_64-linux.fzf: ∅ → 0.34.0, 0.34
  flake:nixpkgs#legacyPackages.x86_64-linux.fzf: ∅ → 0.34.0, 0.34.
  flake:nixpkgs#legacyPackages.x86_64-linux.jq: ∅ → 1.6-bin, 1.6-r
```

When you run this in your terminal, you will likely see the number 2 highlighted in green to indicate it is the current version.

Now, let's say you didn't want to keep around `jq` and `fzf` as you're done with them now after the demo. In this case, you can run the `nix profile rollback` command.

```
> nix profile rollback
switching profile from version 2 to 1
```

Run `nix profile history` again and you'll now see version 1 highlighed as the active version.

Change your mind and want to go back to version 2? The `--to` parameter allows you to set a specific version as the target, letting you go back multiple versions, or, as in this case, forward versions after rollbacks.

```
> nix profile rollback --to 2
switching profile from version 1 to 2
```

Profiles

The second uncommon feature in Nix is the ability to manage multiple sets of installed packages. While in part 7 I'll get onto flakes and a neater way to manage them on a per-project basis, the `nix profile` command is capable of doing it on its own by passing the `--profile` argumemt to a command.

```
> nix profile install --profile ~/py39-profile nixpkgs#python39
> nix profile install --profile ~/py310-profile nixpkgs#python310
> ~/py39-profile/bin/python --version
Python 3.9.14
> ~/py310-profile/bin/python --version
Python 3.10.7
```

Each of these profiles has the full suite of nix commands available to it, including upgrades, rollbacks, etc.

Garbage Collection

Some of you might have heard the mention of Nix keeping all versions of packages around and worried about disk space. There are two commands to keep in mind to resolve this.

The first of these is the `nix store gc` command. This will delete packages that are only used by profiles or generations that have been deleted. This will detect if you just delete a used by profiles or generations that have been deleted. This will detect if you just delete a profile with `rm /path/to/profile`, allowing you to delete and free up a profile entirely.

The second is the `nix profile wipe-history` command. This will remove past versions of a profile. By default it will delete all non-current versions of a profile. But if you want to keep some history, you can also use the `nix profile wipe-history --older-than 30d` to delete generations older than 30 days, for example.

With the usage of the package manager now covered, next time I'm going to discuss Nix's programming language.