What about flakes then? (Nix From First Principles: Flake Edition #7)

Nov 06
2022

#Nix

This is part 7 of the Nix from First Principles: Flake Edition series.

One of the big new concepts in Nix is the `flake`, which is a new standard format for nix projects to declare all their outputs. At the beginning of this series, I mentioned them as being one of the major components of modern Nix, and now this series has introduced enough of a foundation it's time to explain flakes themselves.

What is a flake?

A `flake` is a standard format for describing a collection of Nix resources. These resources can be packages, of the type described in previous posts, intended to be used to install some software to your system. There's also a number of other types of resources that can be exposed by the flake which we'll cover in later posts. Some examples of these are:

- developer environment descriptions
- modules for configuring NixOS systems
- runnable commands

Why the name 'flake'?

Nix's logo is a snow flake. Nix itself means snow in Latin. So Nix flakes are a pun on snow flakes.

By standardising a format to list all these outputs (compared to the ad-hoc files this series has been using until now), Nix allows a number of higher level tools to operate on these resources, like the `nix profile install` and `nix run` commands that have been shown in earlier parts.

Finally, the development of the flake based tooling allowed the Nix team to resolve some pain points with the older generation of tooling, such as making the process of recording versions for reproducibility more ergonomic with lock files, or making better ways to identify newer versions of the same package to allow tools to manage updates.

An example flake

For an example flake, here is a version of the rust derivation from part 6 defined as part of a `flake`. Save this file as `flake.nix`. Unlike the previous filenames, which have been basically arbitrary, this name is expected by the nix tooling as the standard name for defining flakes.

`flake.nix`

```
{
```

```
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-22.05";

  outputs = {
    self,
    nixpkgs,
  }: let
    system = "x86_64-linux";
    pkgs = import nixpkgs {inherit system;};
  in {
    packages.${system}.default =
      pkgs.stdenv.mkDerivation
      {
        src = ./rust-hello;
        name = "rust-hello-1.0";
        inherit system;
        nativeBuildInputs = [pkgs.cargo];
        buildPhase = ''
          cargo build --release
        '';
        installPhase = ''
          mkdir -p $out/bin
          cp target/release/rust-hello $out/bin/rust-hello
          chmod +x $out
        '';
      };
  };
}
```

The `flake.nix` file defines a Nix attribute set, with two top level attributes. The first of these is `inputs`, which defines all the dependencies this flake will import from. For this example, the only dependency is `nixpkgs`, and the flake points to its location on github.

The second top level attribute is `outputs`, which is defined as a function returning an attribute set. It receives two arguments, `self`, which is the flake iself so you can refer to other outputs without resorting to `rec`, and `nixpkgs`, which is the input the flake defined in the `inputs` attribute set. If you had defined extra inputs, they would be provided as extra arguments to this function.

When the function is called, it gets the `pkgs` object by importing the `nixpkgs` expression. Inside flakes you also need to tell it which system to build nixpkgs for, so the flake passes it the `x86_64-linux` system field. You may change this if you're on a different system type, see part 5 if you need a reminder of what options are available.

This example flake defines a single output: `packages.x86_64-linux.default`. A variable is used for the `system` part so that it can be updated in a single place at the top of the file. The value of `packages.x86_64-linux.default` is the package expression that we've previously used alone in files - this part is unchanged from chapter 6.

Now run the `nix build` command. This will lookup the default package for your current system from the flake, then build it. You'll notice like previously a `result` symlink is generated which will contain the built package. It will also generate a `flake.lock` file. This lockfile is the key to not having to manage hashes manually like was needed in the part 6 version - the lock file will automatically record the hashes at the time of building, and then you can include that lock file so that other users of your flake can be sure of getting the same versions, while in your script file you just handle a nice identifier (in this case the flake is asking for the version of nixpkgs used in NixOS 22.05, which is the current stable release).

You can also build my copy of the flake on gitlab, for example with the following command.

```
nix build git+https://gitlab.com/tonyfinn/nix-guide?dir=7-flakes
```

The URL-like parameter `git+https://gitlab.com/tonyfinn/nix-guide?dir=7-flakes/simple` provided to `nix build` to build my hosted copy of the flake is called a *flake reference*. This is provided in the format

`<flake location>[#flake output]`.

This `git+https` URL is just one type of flake location, other examples are given in the sidebar.

A flake can have multiple packages defined in the one flake. These are defined with a different name than the `default` that has been used so far. For example, lets add a `debug` package. This will be added with a new attribute name called `packages.x86_64-linux.debug` on my Linux system. Two things need to be changed for a debug build. The cargo build command command should not have the `--release` flag, and the compiled binary needs to be copied from the `target/debug` folder instead of `target/release`.

`flake.nix`

```
{
  inputs.nixpkgs.url = "github:NixOS/ni

  outputs = {
    self,
    nixpkgs,
  }: let
    system = "x86_64-linux";
    pkgs = import nixpkgs {inherit syst
  in {
```

Types of flake location

Some of the types of flake location supported by Nix include:

Directories, as in:

`nix build ./your-local-nix-guide-checkout`

URLs to git repositories e.g.

`git+https://gitlab.com/tonyfinn/nix-guide`.

You can add various parameters for git here as query parameters, e.g.

`git+https://gitlab.com/tonyfinn/nix-guide?ref=f9a13ed1dc0ed506932e215355a850bfd443691d`

or

`git+https://gitlab.com/tonyfinn/nix-guide?rev=main`

Shorthands for various hosting providers (e.g. `gitlab:tonyfinn/nix-guide`, `github:tonyfinn/nix-guide`). You

```
    in {
      # The default package is the same a
      packages.${system} = {
        default = pkgs.stdenv.mkDerivatio
          src = ./rust-hello;
          name = "rust-hello-1.0";
          inherit system;
          nativeBuildInputs = [pkgs.cargo
          buildPhase = ''
            cargo build --release
          '';
          installPhase = ''
            mkdir -p $out/bin
            cp target/release/rust-hello
            chmod +x $out
          '';
        };
        # Now there's a new package
        debug = pkgs.stdenv.mkDerivation
          src = ./rust-hello;
          name = "rust-hello-debug-1.0";
          inherit system;
          nativeBuildInputs = [pkgs.cargo
          buildPhase = ''
            cargo build
          '';
          installPhase = ''
            mkdir -p $out/bin
            cp target/debug/rust-hello $o
            chmod +x $out
          '';
        };
      };
    };
  }
```

can add an extra path component here to identify a specific git commit such as:

`gitlab:tonyfinn/nix-guide/f9a13ed1dc0ed506932e215355a850bfd443691d`

or

`gitlab:tonyfinn/nix-guide/main`

Finally, if the flake you want is in a subdirectory of the repository, with *any* of the above types, you can add the `dir=xyz` parameter, as was done in the example above.

The default value if unspecified is `.`, for the current directory.

As before, it's still possible to build the release package with a `nix build`. The newly added debug package can be built with `nix build .#debug`. This is using a flake location of `.` (i.e. the flake in the current directory), and a flake output of `debug`. This will look for an output of the right type for the current command and the current system, with the name `debug` and use that.

Since `nix build` is looking for packages to build, and I'm running this on a `x86_64-linux` system, it will find `packages.x86_64-linux.debug` and use this. If you wanted to be explicit you could write out `nix build .#packages.x86_64-linux.debug`, but that's a lot to type so Nix accepts the short version.

As before, this flake output can be combined with an external flake location, so if you wanted to build my copy of the debug package, you could run the following command:

```
nix build git+https://gitlab.com/tonyfinn/nix-guide?dir=7-flakes/mu
```

And that's the end of this introduction to the basics of flakes. The next article will cover other types of outputs that you can put into flakes.