Derivations (Nix From First Principles: Flake Edition #5)

Oct 05
2022

#Nix

This is part 5 of the Nix from First Principles: Flake Edition series.

So, time to explain the nix derivation. A Nix `derivation` is a special type of `set` created from the `derivation` function which describes how to obtain and build a package. The Nix package manger will then evaluate this derivation, and use the result to copy the built package into the Nix store, which is effectively a folder with all locally Nix built or installed packages in it. The subfolders are hashes derived from the inputs to the expression that built the stored packages.

`derivation` itself takes a set as its argument, which it will expand out into the full derivation when evaluted. There are three mandatory arguments:

- `system` defines what architecture and OS this package is for, e.g. `x86_64-linux` or `aarch64-darwin`. Note there is no `any` architecture, if you want a portable package you'll need to define it for every architecture it's portable to.
- `name` is a string which gives the package a name. Conventionally the version is appended, e.g. python-3.10.0.
- `builder` is a program that is run to build the derivation.

You can define extra variables, and they're passed to the `builder` as environment variables. If a variable is a path, there's a special behaviour - the object at the path is copied into the nix store itself, and the path to the copy in the nix store is provided as the environment variable.

There's one other variable of note, which is `outputs`. This is a list of strings which name specific outputs which this derivation builds. By default, a derivation builds a single output, which is `"out"` but you could for example include debug symbols or high res assets for a game as other outputs. The build script is expected to build all outputs, but someone installing the package in future could select only a subset of outputs. Each output results in a target directory for that output being created.

Your First Nix Derivation

With the over covered, it's time to write your first derivation. To write a Nix derivation, you need the following:

1. Something to install. Let's use this basic shell script:

`hello.sh`

```
#!/bin/sh
echo "Hello World"
```

2. A build script which produces the build output. Since a shell script is already executable, your build script should just copy it into place.

`builder.sh`

```
#!/bin/sh
cp $src $out
```

3. A nix expression which defines the derivation as using the src and builder just created.

`expression.nix`

```
derivation {
    builder = ./builder.sh;
    src = ./hello.sh;
    name = "hello-1.0";
    # Replace with x86_64-darwin, aarch64-darwin or aarch64-linux i
    system = "x86_64-linux";
}
```

So how do you run this derivation?

You run the `nix build` command. Since you only have a standalone nix expression in a file for now, you can use the `-f` option to tell it which file to build. So let's try it.

`nix build -f expression.nix`

This doesn't quite work...

```
error: builder for '/nix/store/inhcxhma450491shf69xmf2960lfcr2a-hel
       last 1 log lines:
       > /nix/store/j67w54ch35bca45cbvzx2qdg9p7557fc-builder.sh: li
       For full logs, run 'nix log /nix/store/inhcxhma450491shf69xr
```

The issue here is with the `line 2: cp: not found`. The derivation is built in an isolated environment, and since you haven't included anything in that environment, you get a bourne compatible shell and not much else. Since `cp` is not a shell builtin [1], you'll need to provide the command.

Bootstrapping

So let's bootstrap a bit. First grab a copy of busybox's cp and put it next to your expression.nix. Next, update your derivation to copy the command into the build environment of your derivation:

environment of your derivation.

```
derivation {
    builder = ./builder.sh;
    src = ./hello.sh;
    name = "hello-1.0";
    system = "x86_64-linux";
    # busybox_CP will be copied to the nix store, and the path
    # will be put in the $cp env variable.
    cp = ./busybox_CP;
}
```

Then you can update the builder script to use this version of cp

```
#!/bin/sh
$cp $src $out
```

Finally, make the busybox_CP binary executable with a `chmod +x busybox_CP` - this executable flag will be copied into your build environment.

Now let's try run `nix build -f expression.nix` again:

```
> nix build -f expression.nix
>
```

No errors are outputted, and you will notice a new file called `result` which is a symlink to the location in the nix store where your derivation output is stored

```
> ls -l result
lrwxrwxrwx 1 tony users 53 Sep 29 01:28 result → /nix/store/5nhksv
```

And if you run `./result`:

```
> ./result
Hello World
```

Congratulations, you've built your first nix package. Don't worry, in real world scenarios (next post will cover `stdenv`), you won't need to bring your own `cp` and other fundamental tools.

Cleaning up

Still, even before you get onto the whole stdenv and all it's tools it's a bit ugly to copy busybox's `cp` into your project. Not to mention having to manually `chmod +x` to build the

derivation, when ideally for reproducibility, you want all the steps to be listed. So let's make a few improvements to this first derivation.

First, the new `expression-v2.nix`:

```
let
  fetchurl = import <nix/fetchurl.nix>;
in derivation {
    builder = ./builder.sh;
    src = ./hello.sh;
    name = "hello-1.0";
    system = "x86_64-linux";
    cp = fetchurl {
      # Download from the busybox server
      url = "https://www.busybox.net/downloads/binaries/1.35.0-x86_
      # Check the downloaded file against the hash so Nix knows if
      # Download the file once and use `nix hash to-base32 $(nix ha
      # to find the value to put here
      sha256 = "0mq1487×7aaz89211wrc810k9d51nsfi7jwfy56lg3p20m54r2
      # Have Nix make the file executable on download
      executable = true;
    };
    chmod = fetchurl {
      url = "https://www.busybox.net/downloads/binaries/1.35.0-x86_
      sha256 = "06fp9hqf0cxjqvs9hjg5n81lm5yhkp6iwiaa74j4cfg0wbf7d8
      executable = true;
    };
}
```

And next, the new `build.sh`, which uses that `chmod` command to set the permission flag, saving users from having to run it themselves.

```
$cp $src $out
$chmod +x $out
```

From here you could go on to define an ever increasing pile of tools, working your way up to GNU coreutils for a more "normal" build environment, but luckily the Nix team have already done this work for you in the `Nixpkgs` environment, which the next post will cover.

---

1. Now that I've said that, I'm sure I will be shown a shell where it is a builtin ↩