Nixpkgs - How Not to Reinvent the Wheel (Nix From First Principles: Flake Edition #6)

Oct 12
2022

#Nix

This is part 6 of the Nix from First Principles: Flake Edition series.

In the last section I discussed creating your first derivation, which allows you to make a first nix package. As you might have noticed, the default execution environment is incredibly barebones, to the point that you needed to include such fundamental tools as `chmod` and `cp`. If that process had to be repeated by every Nix user, it would be very inconvenient. Luckily, there is nixpkgs, which provides a number of packages that the community has already built, along with the standard environment which includes a number of tools to use in building your own. You may remember installing some of these packages in part 3.

mkDerivation

The first item from the standard environment I'm going to discuss is `stdenv.mkDerivation`. This is a function that is an enhanced version of the built in `derivation`. It provides a number of advantages over using `derivation` directly:

1. It does the bootstrapping for you ensuring you have a decent minimal environment to build on. The full list of packages can be seen in your current nixpkgs manual, but it includes things like GNU coreutils, bash, tar, gzip, etc. The bash shell is used as the default interpreter for shell scripts, compared to `derivation` which only promises a bourne compatible shell.
2. It provides an easy way to add extra dependencies to your specific derivation and includes them in the path so you don't have to use the `$cp`, `$chmod` variables in the scripts like was required last time.
3. It provides a default builder which runs in bash and does a `./configure && make` installation, with some variables that lets you override parts of it without having to write a whole new script.

Here's a modified version of the "hello world" derivation from the last part.

`derivation-stdenv.nix`

```
let pkgs = import (fetchTarball "https://github.com/NixOS/nixpkgs/a
in pkgs.stdenv.mkDerivation {
    src = ./hello.sh;
    name = "hello-1.0";
    system = "x86_64-linux";
    dontUnpack = true;
    installPhase = ''
    cp $src $out
    chmod +x $out
```

```
      chmod +x $out
      '';
  }
```

Here `fetchTarball` is a function from nix's builtins that downloads a tarball from the internet and stores it in the nix store. The code then calls `import` on the returned result. `import` evaluates the downloaded value as nix code and then assign the result object to the `pkgs` name. The variable name `pkgs` is customary for nixpkgs, so you may see it used in docs without explaining where it comes from.

In the call to mkDerivation, there are the following differences to the builtin `derivation` example from last time:

1. There's no `builder` provided - in this example the standard builder provided by the `stdenv` is used, with one phase that is overridden from the `.nix` file.
2. The `installPhase` option is what overrides this one phase. The script is included inline using Nix's multi-line strings, which are signified with the doubled up single quotes.
3. The `dontUnpack` attribute also tells the builder not to try unpack the source. Since most software sources have more than one file, the standard builder defaults to treating the src as an archive and trying to unpack it. Here the `src` is just a shell script, so that unpacking is not required.
4. Inside the standard environment, items like `cp` and `chmod` are just available on the path, so they don't need to be explicitly passed - the script just uses them as in regular command line use.

Dependencies

Now that there's no longer a need to rebuild the world inside each derivation, it's time to start a more challenging package, one that needs some dependencies. Let's take an example of a Rust version of the hello world program from before. Even if you've never written Rust before, and don't have any Rust toolchain you can use Nix to get everything needed.

Do you have to use that URL?

The next section will explain flakes which is the modern solution to dependency management, but for now in the interest of not putting the cart before the horse I'm just hardcoding the URL in the nix script itself.

Classic nix also has a mechanism called channels which you may have heard of, but this series will likely not cover them unless in an appendix at the end.

Isn't downloading arbitrary code off the internet and executing it bad?

The glib answer is that downloading arbitrary code off the internet is the purpose of a package manager, but when the next post gets onto *flakes*, it will detail some ways to make this process safer and more reproducible.

As a less glib alternative, `fetchTarball` has an alternative version which lets you include the expected hash as an argument, so that you can verify the output. To do this, you can call `fetchTarball` with a set instead of a string, like so:

```
fetchTarball {
  url = "your url here";
  sha256 = "your hash here";
}
```

The `nix-prefetch-url --unpack $YOUR_URL` command can be used to find the expected sha256 hash on first use.

If you're worried about github refusing to serve any content in the future, you can also always host the tarball on your own local mirror.

First get Cargo, Rust's build tool/package manager from Nix and have it scaffold a Rust project for this example.

```
nix run nixpkgs#cargo init rust-hello
```

You will see a download progress bar as it downloads cargo from the nixpkgs cache, then it runs that `cargo` command with the rest of the arguments, as if you had run `cargo init rust-hello` with a locally installed version. Unlike `nix profile install`, the downloaded version of `cargo` isn't kept around permanently - it will be deleted at the next GC.

Cargo will then generate two files. Conveniently, when you generate a new rust project, it prefills in a hello world program to get you started.

`rust-hello/Cargo.toml`

```
[package]
name = "rust-hello"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/

[dependencies]
```

`rust-hello/src/main.rs`

```
fn main() {
    println!("Hello, world!");
}
```

Now it's time to write a derivation to produce a build output from this.

The first new feature needed is a new argument to `stdenv.mkDerivation`. This argument is `nativeBuildInputs`. This is used to download dependencies that need to run on the system building the package, and produce output suitable for the system intended to run the package.

```
nativeBuildInputs = [pkgs.cargo]
```

The second new feature used is that the script now overrides a second phase of the standard builder, the `buildPhase`. As the name suggests, this is where you should run commands to build your software. In this case `cargo build --release` is the command to build an optimised version of a rust program. Since `pkgs.cargo` was added

to the `nativeBuildInputs` section, `cargo` is available on the `PATH` of the build script.

```
  buildPhase = ''
      cargo build --release
  '';
```

Finally, this time the script places the output in `$out/bin` rather than copying it direct to `$out`. This is because `stdenv` puts the `$out/bin` directory onto the path of anything declaring this package as a dependency, so by placing the binary here it will be on the path of anything that uses this as a dependency.

```
  installPhase = ''
      mkdir -p $out/bin
      cp target/release/rust-hello $out/bin/rust-hello
      chmod +x $out
  ''
```

Putting all of these together, the final derivation is below

`rust-derivation.nix`

```
let pkgs = import (fetchTarball "https://github.com/NixOS/nixpkgs/a
in pkgs.stdenv.mkDerivation {
    src = ./rust-hello;
    name = "rust-hello-1.0";
    system = "x86_64-linux";
    nativeBuildInputs = [ pkgs.cargo ];
    buildPhase = ''
      cargo build --release
    '';
    installPhase = ''
      mkdir -p $out/bin
      cp target/release/rust-hello $out/bin/rust-hello
      chmod +x $out
    '';
}
```

Other builders

`stdenv.mkDerivation` is the foundational tool for working with derivations in Nixpkgs, but there's also a library of other builders for common languages and use cases. For example, rather than build the derivation for the first shell example manually, the nix expression could have used the `pkgs.buildShellApplication` builder.

In this case the updated shell derivation would be as follows:

shell-app.nix

```
let pkgs = import (fetchTarball "https://github.com/NixOS/nixpkgs/a
in pkgs.writeShellApplication {
  name = "hello";

  text = ''
    echo Hello World
  '';
}
```

You can also see the Nixpkgs manual on languages and frameworks to see if there's any builders or utilities for your preferred programming language.

Next time, I will cover *flakes*, one of the biggest changes to Nix packaging ever, and the foundation of a lot of the newer methods of interacting with Nix.

© Tony Finn 2011-2024 | Privacy | RSS