pkgs.callPackage

`pkgs.callPackage` is used to parameterize the construction of Nix Derivation. To understand its purpose, let's first consider how we would define a Nix package (also known as a Derivation) without using `pkgs.callPackage`.

---

1. Without `pkgs.callPackage`

We can define a Nix package using code like this:

```nix
pkgs.writeShellScriptBin "hello" ''echo "hello, ryan!"''
```

To verify this, you can use `nix repl`, and you'll see that the result is indeed a Derivation:

```
› nix repl -f '<nixpkgs>'
Welcome to Nix 2.13.5. Type :? for help.

Loading installable ''...
Added 19203 variables.

nix-repl> pkgs.writeShellScriptBin "hello" '' echo "hello, xxx!" ''
«derivation /nix/store/zhgar12vfhbajbchj36vbbl3mg6762s8-hello.drv»
```

While the definition of this Derivation is quite concise, most Derivations in nixpkgs are much more complex. In previous sections, we introduced and extensively used the `import xxx.nix` method to import Nix expressions from other Nix files, which can enhance code maintainability.

1. To enhance maintainability, you can store the definition of the Derivation in a separate file, e.g., `hello.nix`.
    1. However, the context within `hello.nix` itself doesn't include the `pkgs` variable, so you'll need to modify its content to pass `pkgs` as a parameter to `hello.nix`.
2. In places where you need to use this Derivation, you can use `import ./hello.nix pkgs` to import `hello.nix` and use `pkgs` as a parameter to execute the function defined within.

Let's continue to verify this using `nix repl`, and you'll see that the result is still a Derivation:

```shell
› cat hello.nix
pkgs:
  pkgs.writeShellScriptBin "hello" '' echo "hello, xxx!" ''

› nix repl -f '<nixpkgs>'
Welcome to Nix 2.13.5. Type :? for help.

warning: Nix search path entry '/nix/var/nix/profiles/per-user/root/channels' do
Loading installable ''...
Added 19203 variables.

nix-repl> import ./hello.nix pkgs
«derivation /nix/store/zhgar12vfhbajbchj36vbbl3mg6762s8-hello.drv»
```

2. Using `pkgs.callPackage`

In the previous example without `pkgs.callPackage`, we directly passed `pkgs` as a parameter to `hello.nix`. However, this approach has some drawbacks:

1. All other dependencies of the `hello` Derivation are tightly coupled with `pkgs`.
    1. If we need custom dependencies, we have to modify either `pkgs` or the content of `hello.nix`, which can be cumbersome.
2. In cases where `hello.nix` becomes complex, it's challenging to determine which Derivations from `pkgs` it relies on, making it difficult to analyze the dependencies between Derivations.

`pkgs.callPackage`, as a tool for parameterizing the construction of Derivations, addresses these issues. Let's take a look at its source code and comments nixpkgs/lib/customisation.nix#L101-L121:

```nix
  /* Call the package function in the file `fn` with the required
    arguments automatically.  The function is called with the
    arguments `args`, but any missing arguments are obtained from
    `autoArgs`.  This function is intended to be partially
```

```
    parameterised, e.g.,

      callPackage = callPackageWith pkgs;
      pkgs = {
        libfoo = callPackage ./foo.nix { };
        libbar = callPackage ./bar.nix { };
      };

    If the `libbar` function expects an argument named `libfoo`, it is
    automatically passed as an argument.  Overrides or missing
    arguments can be supplied in `args`, e.g.

      libbar = callPackage ./bar.nix {
        libfoo = null;
        enableX11 = true;
      };
  */
  callPackageWith = autoArgs: fn: args:
    let
      f = if lib.isFunction fn then fn else import fn;
      fargs = lib.functionArgs f;

      # All arguments that will be passed to the function
      # This includes automatic ones and ones passed explicitly
      allArgs = builtins.intersectAttrs fargs autoArgs // args;

    # ......
```

In essence, `pkgs.callPackage` is used as `pkgs.callPackage fn args`, where the place holder `fn` is a Nix file or function, and `args` is an attribute set. Here's how it works:

1. `pkgs.callPackage fn args` first checks if `fn` is a function or a file. If it's a file, it imports the function defined within.
    1. After this step, you have a function, typically with parameters like `lib`, `stdenv`, `fetchurl`, and possibly some custom parameters.
2. Next, `pkgs.callPackage fn args` merges `args` with the `pkgs` attribute set. If there are conflicts, the parameters in `args` will override those in `pkgs`.
3. Then, `pkgs.callPackage fn args` extracts the parameters of the `fn` function from the merged attribute set and uses them to execute the function.
4. The result of the function execution is a Derivation, which is a Nix package.

What can a Nix file or function, used as an argument to `pkgs.callPackage`, look like? You can examine examples we've used before in Nixpkgs's Advanced Usage - Introduction: `hello.nix`, `fcitx5-rime.nix`, `vscode/with-extensions.nix`, and `firefox/common.nix`. All of them can be imported using `pkgs.callPackage`.

For instance, if you've defined a custom NixOS kernel configuration in `kernel.nix` and made the development branch name and kernel source code configurable:

```nix
{
  lib,
  stdenv,
  linuxManualConfig,

  src,
  boardName,
  ...
}:
(linuxManualConfig {
  version = "5.10.113-thead-1520";
  modDirVersion = "5.10.113";

  inherit src lib stdenv;

  # file path to the generated kernel config file(the `.config` generated by mak
  #
  # here is a special usage to generate a file path from a string
  configfile = ./. + "${boardName}_config";

  allowImportFromDerivation = true;
})
```

You can use `pkgs.callPackage ./hello.nix {}` in any Nix module to import and use it, replacing any of its parameters as needed:

```nix
{ lib, pkgs, pkgsKernel, kernel-src, ... }:

{
  # ......

  boot = {
```

```
    # ......
    kernelPackages = pkgs.linuxPackagesFor (pkgs.callPackage ./pkgs/kernel {
        src = kernel-src;  # kernel source is passed as a `specialArgs` and inje
        boardName = "licheepi4a";  # the board name, used to generate the kernel
    });

  # ......
}
```

As shown above, by using `pkgs.callPackage`, you can pass different `src` and `boardName` to the function defined in `kernel.nix`, to generate different kernel packages. This allows you to adapt the same `kernel.nix` to different kernel source code and development boards.

The advantages of `pkgs.callPackage` are:

1. Derivation definitions are parameterized, and all dependencies of the Derivation are the function parameters in its definition. This makes it easy to analyze dependencies between Derivations.
2. All dependencies and other custom parameters of the Derivation can be easily replaced by using the second parameter of `pkgs.callPackage`, greatly enhancing Derivation reusability.
3. While achieving the above two functionalities, it does not increase code complexity, as all dependencies in `pkgs` can be automatically injected.

So it's always recommended to use `pkgs.callPackage` to define Derivations.

References

- Chapter 13. Callpackage Design Pattern - Nix Pills
- callPackage, a tool for the lazy - The Summer of Nix
- Document what callPackage does and its preconditions - Nixpkgs Issues

Loading comments...