The Ingenious Uses of Multiple nixpkgs Instances

In the section Downgrade or Upgrade Packages, we have seen how to instantiate multiple distinct nixpkgs instances using the method `import nixpkgs {...}` , and use them at any submodules via `specialArgs` . There are numerous applications for this technique, some common ones include:

1. Instantiate nixpkgs instances with different commit IDs to install various versions of software packages. This approach was used in the previous section Downgrade or Upgrade Packages.
2. If you wish to utilize overlays without affecting the default nixpkgs instance, you can instantiate a new nixpkgs instance and apply overlays to it.
    - The `nixpkgs.overlays = [...];` mentioned in the previous section on Overlays directly modifies the global nixpkgs instance. If your overlays make changes to some low-level packages, it might impact other modules. One downside is an increase in local compilation (due to cache invalidation), and there might also be functionality issues with the affected packages.
3. In cross-system architecture compilation, you can instantiate multiple nixpkgs instances to selectively use QEMU simulation for compilation and cross-compilation in different locations, or to add various GCC compilation parameters.

In conclusion, instantiating multiple nixpkgs instances is highly advantageous.

Instantiating `nixpkgs`

Let's first understand how to instantiate a non-global nixpkgs instance. The most common syntax is as follows:

```nix
{
  # a simple example
  pkgs-xxx = import nixpkgs {
    system = "x86_64-linux";
  };

  # nixpkgs with custom overlays
  pkgs-yyy = import nixpkgs {
    system = "x86_64-linux";

    overlays = [
      (self: super: {
        google-chrome = super.google-chrome.override {
          commandLineArgs =
            "--proxy-server='https=127.0.0.1:3128;http=127.0.0.1:3128'";
        };
        # ... other overlays
      })
    ];
  };

  # a more complex example (cross-compiling)
  pkgs-zzz = import nixpkgs {
    localSystem = "x86_64-linux";
    crossSystem = {
      config = "riscv64-unknown-linux-gnu";

      # https://wiki.nixos.org/wiki/Build_flags
      # this option equals to adding `-march=rv64gc` to CFLAGS.
      # CFLAGS will be used as the command line arguments for gcc/clang.
      gcc.arch = "rv64gc";
      # equivalent to `-mabi=lp64d` in CFLAGS.
      gcc.abi = "lp64d";
    };

    overlays = [
      (self: super: {
        google-chrome = super.google-chrome.override {
          commandLineArgs =
            "--proxy-server='https=127.0.0.1:3128;http=127.0.0.1:3128'";
        };
        # ... other overlays
      })
    ];
  };
}
```

We have learned in our study of Nix syntax:

> The `import` expression takes a path to another Nix file as an argument and returns the execution result of that Nix file. If the argument to `import` is a folder path, it returns the execution result of the `default.nix` file within that folder.

`nixpkgs` is a flake with a `default.nix` file in its root directory. So, `import nixpkgs` essentially returns the execution result of [nixpkgs/default.nix](). Starting from this file, you can find that the implementation of `import nixpkgs` is in [pkgs/top-level/impure.nix](), as excerpted below:

```nix
# ... skipping some lines

{ # We put legacy `system` into `localSystem` if `localSystem` was not passed.
  # If neither is passed, assume we are building packages on the current
  # (build, in GNU Autotools parlance) platform.
  localSystem ? { system = args.system or builtins.currentSystem; }

# These are needed only because nix's `--arg` command-line logic doesn't work
# with unnamed parameters allowed by ...
, system ? localSystem.system
, crossSystem ? localSystem

, # Fallback: The contents of the configuration file found at $NIXPKGS_CONFIG or
  # $HOME/.config/nixpkgs/config.nix.
  config ? let
  # ... skipping some lines

, # Overlays are used to extend Nixpkgs collection with additional
  # collections of packages.  These collection of packages are part of the
  # fix-point made by Nixpkgs.
  overlays ? let
  # ... skipping some lines

, crossOverlays ? []

, ...
} @ args:

# If `localSystem` was explicitly passed, legacy `system` should
# not be passed, and vice versa.
assert args ? localSystem -> !(args ? system);
assert args ? system -> !(args ? localSystem);

```

```
import ./. (builtins.removeAttrs args [ "system" ] // {
  inherit config overlays localSystem;
})
```

Therefore, `import nixpkgs {...}` effectively calls this function, and the subsequent attribute set becomes the arguments for this function.

Considerations

When creating multiple nixpkgs instances, there are some details to keep in mind. Here are some common issues to consider:

1. According to the article 1000 instances of nixpkgs shared by @fbewivpjsbsby, it's not a good practice to use `import` to customize `nixpkgs` in submodules or sub-flakes. This is because each `import` evaluates separately, creating a new nixpkgs instance each time. As the number of configurations increases, this can lead to longer build times and higher memory usage. Therefore, it's recommended to create all nixpkgs instances in the `flake.nix` file.
2. When mixing QEMU simulation and cross-compilation, care should be taken to avoid unnecessary duplication of package compilations.

Loading comments...