Cross-platform Compilation

On any Linux platform, there are two ways to do cross-platform compilation. For example, to build an `aarch64-linux` program on an `x86_64-linux` host, you can use the following methods:

1. Use the cross-compilation toolchain to compile the `aarch64` program.
   - The disadvantage is that you cannot use the NixOS binary cache, and you need to compile everything yourself (cross-compilation also has a cache, but there is basically nothing in it).
   - The advantages are that you don't need to emulate the instruction set, and the performance is high.
2. Use QEMU to emulate the `aarch64` architecture and then compile the program in the emulator.
   - The disadvantage is that the instruction set is emulated, and the performance is poor.
   - The advantage is that you can use the NixOS binary cache, and you don't need to compile everything yourself.

If you use method one, you don't need to enable `binfmt_misc`, but you need to execute the compilation through the cross-compilation toolchain.

If you use method two, you need to enable the `binfmt_misc` of the `aarch64` architecture in the NixOS configuration of the building machine.

Cross Compilation

`nixpkgs` provides a set of predefined host platforms for cross-compilation called `pkgsCross`. You can explore them in `nix repl`.

shell

```shell
› nix repl '<nixpkgs>'
warning: future versions of Nix will require using `--file` to load a file
Welcome to Nix 2.13.3. Type :? for help.

Loading installable ''...
Added 19273 variables.

```

```
nix-repl> pkgsCross.<TAB>
pkgsCross.aarch64-android                 pkgsCross.msp430
pkgsCross.aarch64-android-prebuilt        pkgsCross.musl-power
pkgsCross.aarch64-darwin                  pkgsCross.musl32
pkgsCross.aarch64-embedded                pkgsCross.musl64
pkgsCross.aarch64-multiplatform           pkgsCross.muslpi
pkgsCross.aarch64-multiplatform-musl      pkgsCross.or1k
pkgsCross.aarch64be-embedded              pkgsCross.pogoplug4
pkgsCross.arm-embedded                    pkgsCross.powernv
pkgsCross.armhf-embedded                  pkgsCross.ppc-embedded
pkgsCross.armv7a-android-prebuilt         pkgsCross.ppc64
pkgsCross.armv7l-hf-multiplatform         pkgsCross.ppc64-musl
pkgsCross.avr                             pkgsCross.ppcle-embedded
pkgsCross.ben-nanonote                    pkgsCross.raspberryPi
pkgsCross.fuloongminipc                   pkgsCross.remarkable1
pkgsCross.ghcjs                           pkgsCross.remarkable2
pkgsCross.gnu32                           pkgsCross.riscv32
pkgsCross.gnu64                           pkgsCross.riscv32-embedded
pkgsCross.i686-embedded                   pkgsCross.riscv64
pkgsCross.iphone32                        pkgsCross.riscv64-embedded
pkgsCross.iphone32-simulator              pkgsCross.rx-embedded
pkgsCross.iphone64                        pkgsCross.s390
pkgsCross.iphone64-simulator              pkgsCross.s390x
pkgsCross.loongarch64-linux               pkgsCross.sheevaplug
pkgsCross.m68k                            pkgsCross.vc4
pkgsCross.mingw32                         pkgsCross.wasi32
pkgsCross.mingwW64                        pkgsCross.x86_64-darwin
pkgsCross.mips-linux-gnu                  pkgsCross.x86_64-embedded
pkgsCross.mips64-linux-gnuabi64           pkgsCross.x86_64-freebsd
pkgsCross.mips64-linux-gnuabin32          pkgsCross.x86_64-netbsd
pkgsCross.mips64el-linux-gnuabi64         pkgsCross.x86_64-netbsd-llvm
pkgsCross.mips64el-linux-gnuabin32        pkgsCross.x86_64-unknown-redox
pkgsCross.mipsel-linux-gnu
pkgsCross.mmix
```

If you want to set `pkgs` to a cross-compilation toolchain globally in a flake, you only need to add a Module in `flake.nix`, as shown below:

```nix
{
  description = "NixOS running on LicheePi 4A";

  inputs = {
```

```nix
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = inputs@{ self, nixpkgs, ... }: {
    nixosConfigurations.lp4a = nixpkgs.lib.nixosSystem {
      # native platform
      system = "x86_64-linux";
      modules = [

        # add this module, to enable cross-compilation.
        {
          nixpkgs.crossSystem = {
            # target platform
            system = "riscv64-linux";
          };
        }

        # ...... other modules
      ];
    };
  };
}
```

The `nixpkgs.crossSystem` option is used to set `pkgs` to a cross-compilation toolchain, so that all the contents built will be `riscv64-linux` architecture.

Compile through emulated system

The second method is to cross-compile through the emulated system. This method does not require a cross-compilation toolchain.

To use this method, first your building machine needs to enable the binfmt_misc module in the configuration. If your building machine is NixOS, add the following configuration to your NixOS Module to enable the simulated build system of `aarch64-linux` and `riscv64-linux` architectures:

```nix
{ ... }:
{
  # ......
```

```

  # Enable binfmt emulation.
  boot.binfmt.emulatedSystems = [ "aarch64-linux" "riscv64-linux" ];

  # ......
}
```

As for `flake.nix`, its setting method is very simple, even simpler than the setting of cross-compilation, as shown below:

```nix
{
  description = "NixOS running on LicheePi 4A";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = inputs@{ self, nixpkgs, ... }: {
    nixosConfigurations.lp4a = nixpkgs.lib.nixosSystem {
      # native platform
      system = "riscv64-linux";
      modules = [
        # ...... other modules
      ];
    };
  };
}
```

You do not need to add any additional modules, just specify `system` as `riscv64-linux`. Nix will automatically detect whether the current system is `riscv64-linux` during the build. If not, it will automatically build through the emulated system(QEMU). For users, these underlying operations are completely transparent.

Linux binfmt_misc

The previous section only provided an introduction on how to use Nix's emulated system, but if you want to understand the underlying details, here's a brief introduction.

`binfmt_misc` is a feature of the Linux kernel, which stands for Kernel Support for miscellaneous Binary Formats. It enables Linux to run programs for almost any CPU architecture, including X86_64, ARM64, RISCV64, and more.

To enable `binfmt_misc` to run programs in various formats, two things are required: a specific identification method for the binary format and the location of the corresponding interpreter. Although `binfmt_misc` sounds powerful, its implementation is surprisingly easy to understand. It works similarly to how the Bash interpreter determines the interpreter to use by reading the first line of a script file (e.g., `#!/usr/bin/env python3`). `binfmt_misc` defines a set of rules, such as reading the magic number at a specific location in the binary file or determining the executable file format based on the file extension (e.g., .exe, .py). It then invokes the corresponding interpreter to execute the program. The default executable file format in Linux is ELF, but `binfmt_misc` expands the execution possibilities by allowing a wide range of binary files to be executed using their respective interpreters.

To register a binary program format, you need to write a line in the format `:name:type:offset:magic:mask:interpreter:flags` to the `/proc/sys/fs/binfmt_misc/register` file. The detailed explanation of the format is beyond the scope of this discussion.

Since manually writing the registration information for `binfmt_misc` can be cumbersome, the community provides a container to assist with automatic registration. This container is called `binfmt` and running it will install various `binfmt_misc` emulators. Here's an example:

```shell
# Register all architectures
podman run --privileged --rm tonistiigi/binfmt:latest --install all

# Register only common arm/riscv architectures
docker run --privileged --rm tonistiigi/binfmt --install arm64,riscv64,arm
```

The `binfmt_misc` module was introduced in Linux version 2.6.12-rc2 and has undergone several minor changes in functionality since then. In Linux 4.8, the "F" (fix binary) flag was added, allowing the interpreter to be invoked correctly in mount namespaces and chroot environments. To work properly in containers where multiple architectures need to be built, the "F" flag is necessary. Therefore, the kernel version needs to be 4.8 or above.

In summary, `binfmt_misc` provides transparency compared to explicitly calling an interpreter to execute non-native architecture programs. With `binfmt_misc`, users no longer need to worry about which interpreter to use when running a program. It allows programs of any architecture to be executed directly. The configurable "F" flag is an added benefit, as it loads the interpreter program into memory during installation and remains unaffected by subsequent environment changes.

Custom build toolchain

Sometimes we may need to use a custom toolchain for building, such as using our own gcc, or using our own musl libc, etc. This modification can be achieved through overlays.

For example, let's try to use a different version of gcc, and test it through `nix repl`:

```shell

```shell
› nix repl -f '<nixpkgs>'
Welcome to Nix 2.13.3. Type :? for help.

Loading installable ''...
Added 17755 variables.

# replace gcc through overlays, this will create a new instance of nixpkgs
nix-repl> a = import <nixpkgs> { crossSystem = { config = "riscv64-unknown-linux

# check the gcc version, it is indeed changed to 12.2
nix-repl> a.pkgsCross.riscv64.stdenv.cc
«derivation /nix/store/jjvvwnf3hzk71p65x1n8bah3hrs08bpf-riscv64-unknown-linux-gn

# take a look at the default pkgs, it is still 11.3
nix-repl> pkgs.pkgsCross.riscv64.stdenv.cc
«derivation /nix/store/pq3g0wq3yfc4hqrikr03ixmhqxbh35q7-riscv64-unknown-linux-gn
```

So how to use this method in Flakes? The example `flake.nix` is as follows:

```nix
{
  description = "NixOS running on LicheePi 4A";

```
```

```nix

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11-small";
  };

  outputs = { self, nixpkgs, ... }:
  {
    nixosConfigurations.lp4a = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        {
          nixpkgs.crossSystem = {
            config = "riscv64-unknown-linux-gnu";
          };

          # replace gcc with gcc12 through overlays
          nixpkgs.overlays = [ (self: super: { gcc = self.gcc12; }) ];
        }

        # other modules ......
      ];
    };
  };
}
```

`nixpkgs.overlays` is used to modify the `pkgs` instance globally, and the modified `pkgs` instance will take effect to the whole flake. It will likely cause a large number of cache missing, and thus require building a large number of Nix packages locally.

To avoid this problem, a better way is to create a new `pkgs` instance, and only use this instance when building the packages we want to modify. The example `flake.nix` is as follows:

```nix
{
  description = "NixOS running on LicheePi 4A";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11-small";
  };

  outputs = { self, nixpkgs, ... }: let
    # create a new pkgs instance with overlays
```

```
      pkgs-gcc12 = import nixpkgs {
        localSystem = "x86_64-linux";
        crossSystem = {
          config = "riscv64-unknown-linux-gnu";
        };

        overlays = [
          (self: super: { gcc = self.gcc12; })
        ];
      };
    in {
      nixosConfigurations.lp4a = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        specialArgs = {
          # pass the new pkgs instance to the module
          inherit pkgs-gcc12;
        };
        modules = [
          {
            nixpkgs.crossSystem = {
              config = "riscv64-unknown-linux-gnu";
            };
          }

          ({pkgs-gcc12, ...}: {
            # use the custom pkgs instance to build the package hello
            environment.systemPackages = [ pkgs-gcc12.hello ];
          })

          # other modules ......
        ];
      };
    };
}
```

Through the above method, we can easily customize the build toolchain of some packages without affecting the build of other packages.

References

- [Cross compilation - nix.dev](https://nix.dev)

Loading comments...