Nix

Чистый функциональный пакетный менеджер и система развёртывания для POSIX-совместимых ОС.

Список используемых источников

- Nix pillls.
- Nix pills(официальный перевод)
- Nix: Что это и с чем это употреблять?

Циклы статей

Nix в пилюлях

1. Nix в пилюлях: Почему вам стоит попробовать Nix.
2. Nix в пилюлях: Установка в вашей системе.
3. Nix в пилюлях: Погружаемся в среду.
4. Nix в пилюлях: Основы языка.
5. Nix в пилюлях: Функции и импорт.
6. Nix в пилюлях: Наша первая деривация.
7. Nix в пилюлях: Работающая деривация.
8. Nix в пилюлях: Универсальные скрипты сборки.
9. Nix в пилюлях: Автоматические зависимости времени выполнения.
10. Nix в пилюлях: Разработка с помощью nix-shell.

Nix. Что это и с чем это использовать

Где это всё взять?

Помимо NixOS, где ничего делать не нужно, Nix можно установить на любой (или почти любой) дистрибутив Linux. Для этого достаточно запустить следующую команду:

```
$ sh <(curl -L https://nixos.org/nix/install)
```

Дальше установочный скрипт сам всё сделает. После недавних изменений в MacOS, установка в ней немного осложнилась, раньше было достаточно команды выше. Про установку в последних версиях MacOS можно прочесть здесь.

Язык Nix

Когда речь идёт о Nix, часто имеют в виду две разные сущности: Nix как язык и nixpkgs как репозиторий пакетов, в том числе составляющий основу NixOS. Начнём с первого.

Nix — функциональный ленивый язык с динамической типизацией. Синтаксис во многом похож на языки семейства ML (SML, OCaml, Haskell), поэтому у тех, кто с ними знаком, особых проблем возникнуть не должно.

Начать знакомство с языком можно просто запустив интерпретатор.

```
$ nix repl
Welcome to Nix version 2.3.10. Type :? for help.


nix-repl>
```

Отдельного синтаксиса для объявления функций в Nix нет. Функции задаются через присваивание, так же как и другие значения.

```
nix-repl> "Hello " + "World!"
"Hello World!"


nix-repl> add = a: b: a + b


nix-repl> add 1 2
3
```

Как и в языках, повлиявших на Nix, все функции каррированы.

```
nix-repl> addOne = add 1


nix-repl> addOne 3
4
```

Помимо примитивных типов, таких как числа и строки, Nix поддерживает списки и словари (attribute sets в терминологии Nix).

```
nix-repl> list = [ 1 2 3 ]


nix-repl> set = { a = 1; b = list; }


nix-repl> set
{ a = 1; b = [ ... ]; }


nix-repl> set.b
[ 1 2 3 ]
```

Значения в локальной области видимости можно задать через выражение `let...in`. Для примера, простая функция, реализующая факториал, как это принято делать в других статьях по функциональному программированию.

`fac.nix`:

```
let
  fac = n:
    if n == 0
    then 1
    else n * fac (n - 1);
in { inherit fac; }
```

Директива `inherit` вносит или "наследует" термин из текущей области видимости и даёт ему такое же имя. Пример выше эквивалентен записи `let fac = ... in { fac = fac; }`.

```
$ nix repl fac.nix
Welcome to Nix version 2.3.10. Type :? for help.

Loading 'fac.nix'...
Added 1 variables.

nix-repl> fac 3
6
```

При загрузке файлов или модулей в REPL, Nix ожидает, что результатом вычисления модуля будет множество, элементы которого будут импортированы в текущую область видимости.

Для загрузки кода из других файлов в Nix есть функция `import`, принимающая путь к файлу с кодом и возвращающая результат выполнения этого кода.

`mul.nix`:

```
let
  mul = a: b: a * b;
in { inherit mul; }
```

Новый `fac.nix`:

```
let
  multMod = import ./mul.nix;
  fac = n:
    if n == 0
    then 1
    else multMod.mul n (fac (n - 1));
in { inherit fac; }
```

Хотя присваивание модуля в отдельную переменную — довольно частая практика, в данном случае это выглядит несколько нелепо, правда? В Nix есть директива `with`, добавляющая в текущую область видимости все имена из множества, переданного в качестве параметра.

`fac.nix` с использованием `with`:

```
with import ./mul.nix;
let
  fac = n:
    if n == 0
    then 1
    else mul n (fac (n - 1));
in { inherit fac; }
```

Сборка программ

Сборка программ и отдельных компонентов — это основная функция языка Nix.

В случае работы с пакетами, основным инструментом, про который нужно знать, является `Derivation`. Сам по себе `Derivation` — это специальный файл, содержащий рецепт для сборки в машинно-читаемом виде. Для компиляции программы на C, выводящей "Hello World!", derivation выглядит примерно следующим образом:

```
Derive([("out","/nix/store/1nq46fyv3629slgxnagqn2c01skp7xrq-hello-
world","","")],[("/nix/store/60xqp516mkfhf31n6ycyvxppcknb2dwr-build-
```

```
hello.drv",["out"])],["/nix/store/wiviq2xyz0ylhl0qcgfgl9221nkvvxfj-hello.c"],"x86_64-
linux","/nix/store/r5lh8zg768swlm9hxxfrf9j8gwyadi72-build-
hello",[],[("builder","/nix/store/r5lh8zg768swlm9hxxfrf9j8gwyadi72-build-hello"),("name","hello-
world"),("out","/nix/store/1nq46fyv3629slgxnagqn2c01skp7xrq-hello-
world"),("src","/nix/store/wiviq2xyz0ylhl0qcgfgl9221nkvvxfj-hello.c"),("system","x86_64-linux")])
```

Как видно, в этом выражении содержится путь к результату сборки, который получится в итоге, а также пути к исходным файлам, скрипту сборки, и метаданные: имя проекта и платформа. Стоит так же заметить, что пути к исходникам начинаются с `/nix/store`. При сборке, Nix копирует всё нужное в эту директорию, после чего сборка происходит в изолированном окружении (sandbox). Таким образом достигается воспроизводимость сборки всех пакетов.

Разумеется, никто в здравом уме руками писать такое не станет! Для простых случаев, в Nix есть встроенная функция `derivation`, принимающая описание сборки.

`simple-derivation/default.nix`:

```
{ pkgs ? import <nixpkgs> {} }:

derivation {
  name = "hello-world";
  builder = pkgs.writeShellScript "build-hello" ''
    ${pkgs.coreutils}/bin/mkdir -p $out/bin
    ${pkgs.gcc}/bin/gcc $src -o $out/bin/hello -O2
  '';
  src = ./hello.c;
  system = builtins.currentSystem;
}
```

Давайте попробуем разобрать этот пример. Весь файл представляет собой определение функции, которая берёт один параметр — словарь, содержащий поле `pkgs`. Если оно не было передано при вызове этой функции, используется значение по умолчанию: `import <nixpkgs> {}`.

`derivation` — функция, так же принимающая словарь с параметрами сборки: `name` — имя пакета, `builder` — сборочный скрипт, `src` — исходный код, `system`

— система или список систем, под который возможна сборка данного пакета.

`writeShellScript` — функция из `nixpkgs`, принимающая имя для скрипта и код и возвращающая путь к исполняемому файлу. Для многострочного текста в Nix есть альтернативный синтаксис с двумя парами одинарных кавычек.

С помощью команды `nix build`, этот рецепт для сборки можно запустить и получить работающий бинарник.

```
$ nix build -f ./simple-derivation/default.nix
[1 built]

$ ./result/bin/hello
Hello World!
```

При запуске `nix build`, в текущей директории создаётся символическая ссылка `result`, указывающая на созданный в `/nix/store` пакет.

```
$ ls -l result
lrwxrwxrwx 1 user users 50 Mar 29 17:53 result -> /nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple

$ find /nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple/bin
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple/bin/hello
```

Сборка программ, продвинутая версия

`derivation` — достаточно низкоуровневая функция, на базе которой в Nix построены куда более мощные примитивы. Для примера, можно рассмотреть сборку широко известной утилиты `cowsay`.

```
{ lib, stdenv, fetchurl, perl }:

stdenv.mkDerivation rec {
  version = "3.03+dfsg2";
  pname = "cowsay";

  src = fetchurl {
    url = "http://http.debian.net/debian/pool/main/c/cowsay/cowsay_${version}.orig.tar.gz";
    sha256 = "0ghqnkp8njc3wyqx4mlg0qv0v0pc996x2nbyhqhz66bbgmf9d29v";
  };

  buildInputs = [ perl ];

  postBuild = ''
    substituteInPlace cowsay --replace "%BANGPERL%" "!${perl}/bin/perl" \
      --replace "%PREFIX%" "$out"
  '';

  installPhase = ''
    mkdir -p $out/{bin,man/man1,share/cows}
    install -m755 cowsay $out/bin/cowsay
    ln -s cowsay $out/bin/cowthink
    install -m644 cowsay.1 $out/man/man1/cowsay.1
    ln -s cowsay.1 $out/man/man1/cowthink.1
    install -m644 cows/* -t $out/share/cows/
  '';

  meta = with lib; {
    description = "A program which generates ASCII pictures of a cow with a message";
    homepage = "https://en.wikipedia.org/wiki/Cowsay";
    license = licenses.gpl1;
    platforms = platforms.all;
    maintainers = [ maintainers.rob ];
  };
}
```

Оригинал скрипта находится здесь.

`stdenv` — специальный `derivation`, содержащий правила сборки для текущей системы: нужный компилятор, флаги и прочие параметры. Основное содержимое — гигантских размеров скрипт на баше под названием `setup`, который и выступает в роле скрипта `builder` из нашего простого примера выше.

```
 $ nix build nixpkgs.stdenv


 $ find result/
result/
result/setup
result/nix-support

$ wc -l result/setup
1330 result/setup
```

`mkDerivation` — функция, создающая `derivation` с этим скриптом и заодно заполняющая другие поля.

Те читатели, кто раньше писал скрипты для сборки пакетов в Arch Linux или Gentoo, могут увидеть здесь крайне знакомую структуру. Как и в других дистрибутивах, сборка разбита на фазы, присутствует перечисление зависимостей (`buildInputs`) и так далее.

Терминология

Каррирование (от англ. currying, иногда — карринг) — преобразование функции от многих аргументов в набор вложенных функций, каждая из которых является функцией от одного аргумента.

Определения

Набор атрибутов - ассоциативный массив строковых ключей с их значениями.

```
{ foo = "bar"; a-b = "baz"; "123" = "num"; }
```

Набор аргументов функции - набор ключей атрибутов использующийся в качестве аргументов вызова функции.

```
{ a, b }: a*b
```