Nix езикът

Универсални конфигурации с Nix

Павел Атанасов Камен Младенов

08.04.2025

Преговор

- Разгледахме повърхностно Nix езикът
- Написахме няколко много прости пакети
- Употребявахме `nix-build`, `nix-shell`, `nix-instantiate`, `nix-store`

- Миналия път `nix-collect-garbage` не проработи. Защо?
- `nix-build` добавя *символична връзка* `result` в текущата директория, сочеща към резултата в `/nix/store`
- Тази връзка **се добавя в roots**, произволна връзка не работи.
- И затова `nix-build` не казва “the result might be removed by the garbage collector”, както `nix-instantiate`

```
nix-build ./text.nix
nix-collect-garbage -d # Не бърника /nix/store/...-something.txt

rm result
nix-collect-garbage -d # Вече премахва /nix/store/...-something.txt
```

Пример text.nix

```
with import <nixpkgs> { };
writeTextFile { name = "something.txt"; text = "Hello!"; }
```

Раздел 1

Защо ни трябва нов език?

Скриптиращи езици - bash/powershell/...

Плюсове

- Nix програми се свеждат до скрипт, ще пропуснем тази фаза
- Позволяват на разработчика да прави всичко със системата

Минуси

- Позволяват на разработчика да прави всичко със системата
- Трудно е да се пишат големи и сигурни скриптове

Езици за данни - JSON/YAML/...

Плюсове

- Пишат се лесно
- Не е нужно да си програмист

Минуси

- Създаването на среда може да е сложно (условно избиране на компоненти, обхождане на настройки, променяне на файлове/променливи/...), **нужно ни е програмиране**

Императивни (процедурни) езици - C++/Java/Python/...

Плюсове

- Често срещани езици за програмиране
- Много налични ресурси, библиотеки, работна ръка

Минуси

- Операции извън програмата (обработка на файлове, shell променливи, ...) са сложни
- Лесно се пише код, който не дава възпроизводими резултати

Декларативни (функционални) езици - Haskell/Racket/Scheme/...

Плюсове

- Добре познати; ресурсите и библиотеките не са толкова много, но са достатъчно
- Не-възпроизводим код е много труден за писане

Минуси

- Големи езици, което означава че имат големи интерпретатори и са трудни за пренасяне към различни архитектури

От какво се нуждаем

- “Обектни” типове данни, заради йерархиите от данни
- Възможност за писане на изпълним код, за да поддържаме сложна логика
- Функционален, за да сме сигурни във възпроизводимостта си
- Малък, за да има бърз и портативен интерпретатор

Раздел 2

Езикът Nix

Езикът Nix

- Чисто-функционален, лениво-оценен, динамично-типизиран
- Интерпретаторът е ~13k реда C++ код
- *Приблизително* 7 примитивни типа, 3 композитни типа, 6 конструкции
- Едноредови коментари чрез `#` и многоредови чрез `/* */`

Лесния начин да експериментираме

- Във Версия 3 имаме `nix repl`, която предоставя интерактивна среда, в която директно пишем код на Nix и той се изпълнява
- Допълнително можем да създадем *глобални* променливи с `name = expression`

Раздел 3

Примитивни типове данни

Примитивни типове данни

null

```
null
```

Булеви

```
true false
```

Числа (цели и дробни)

```
2874 -300 182.384 1.5e7
```

Низове (едноредови и многоредови)

```
"Hello World!"
''
Низ
 на
  много
   редове
''
```

Поддържат интерполация

```
"Sum: ${2 + 5}!"
```

Път във файловата система

```
./directory /usr/share/bin ../test.txt
```

Раздел 4

Композитни типове данни

Композитни типове: списък

- Хетерогенен, константна дължина, лениво оценен
- Нямаме индексиране, трябва да използваме функциите `head`, `tail`, ...

Синтаксис

```
[ null true 287 -3.1e4 "Hi" ./main.txt ]
```

Композитни типове: атрибутно множество

- Редица изрази `име = стойност;`
- Еквивалента на `struct` в други езици
- **Не е обект!** Само контейнер за данни.

```
{
  name = "John";
  age = 22;
  children = null;
  identifiers = [ 841 "AXZH" ];
  vehicle = { brand = "Dacia"; };
  vehicle.model = "Sandero";
}
```

- Достъпваме стойности с `.` операторът

```
{ a = 5; b = 6; }.a
{ a = 8; b = { x = 2; y = 3; }; }.b.x
```

В `nix repl`:

```
var = { a = { x = 61; y = 82; }; b = 100; }
var.a.y
```

- **Атрибутите са произволни низове**
- Когато се съставят от латински букви, цифри, - и _, можем да **пропуснем кавичките**

```
{ item = "Apple"; } /* е същото като */ { "item" = "Apple"; }
```

- И когато правим **само** интерполация можем да ги изпуснем

```
{ ${6 * 7} = "Живот"; } /* е същото като */ { 42 = "Живот"; }
```

- За индексирането важат същите правила

```
var = { "Команда" = "echo"; _status = 1; 1 = "Успех"; }
var."Команда"
var.${var._status}
```

Композитни типове: безименна функция

- Функциите са “first-class citizen” (третират се като обикновени стойности)
- Приемат само **един** аргумент

```
input_argument_name: expression
```

- Можем да върнем функция и подаването на аргумент ще емулира много аргументи

```
x: y: (x + 1) * y
```

- Извикване става чрез изреждане на аргументите след функцията

```
func = x: y: (x + 1) * y
func 5 7   # Връща 42
```

- За да разграничим кои шпации са част от “тялото” на функцията и кои извън, ограждаме с кръгли скоби

```
(x: y: (x + 1) * y) 5 7   # Връща 42
```

Това важи и когато функцията е елемент на списък

```
[ 1 (x: y: (x + 1) * y) "Hi" ]
```

- Друг метод за много аргументи е да се използва атрибутно множество

```
x: x.a + x.b
```

- Много често се прави, затова имаме “деструктуриране”, където атрибутите могат да се достъпят директно

```
{ a, b }: a + b
```

- Понякога е полезно да работим u с цялото множество

```
{ a, b }@x: x.a + b
x@{ a, b }: x.a + b
```

- Ще получим грешка, ако атрибутнотно множество има повече аргументи от зададените. Можем да ги игнорираме чрез триточие в края на деструктирането.

```
({ a, b, ... }: a + b) { a = 5; b = 6; c = 7; }    # Връща 11
```

- Аналогично получаваме грешка ако аргументи липсват. Можем да зададем стойности по подразбиране чрез ?.

```
({ a ? 5, b ? 6 }: a + b) {}    # Връща 11
```

Раздел 5

Конструкции

Конструкции

- Всичко това бяха стойности
- Имаме и *конструкции*, които връщат стойности, но не са стойност сами по себе си

rec

Прави атрибутни множества рекурсивни, т.е. всеки атрибут е наличен на атрибутите след него. Пише се префиксно.

```
rec { a = 5; b = a + 1; c = a * b; }
```

let-in

Дефинираме променливи за даден израз. Истинският еквивалент на `name = value` в `nix repl`.

Името съдържа латински букви, цифри, `-`, `_`, `'`. Започва с латинска буква или `_`.

```
let x = 6; y = 1; in x + y * y    # Връща 7
```

Като при `rec`, всяка променлива е налична (в израза) на всяка следваща.

if-else

Условен израз. Винаги връща стойност, затова `else` не може да се изпусне.

```
if x < y then 2 * x else 3 * y
```

with

Работи като `let`, обаче му подаваш атрибутно множество. Така можем да зададем променливи по време на изпълнение.

```
with { a = 81; b = 36; }; a + b
/* е същото като */
let a = 81; b = 36; in a + b
```

inherit *name*

Създава `name = name` клаузи по подадена променлива. Полезно при дефиниране на атрибутни множества.

```
let num = 10; x = { inherit num; }; in x.num
```

inherit (*attrset*) *name*;

Създава `name = attrset.name` клаузи по подадена променлива. Полезно при предефиниране на атрибутни множества, използвайки "подмножество" от ключовете на друго атрибутно множество.

```
let numAttrset = { num = 10; }; x = { inherit (numAttrset) num; }; in x.num
```

or

При индексиране в атрибутно множество можем да използваме `or` след него за стойност по подразбиране

```
{ a = 7; }.b or 9    # Връща 9
```

Работи и при вложени множества

```
{ a = { b = { c = 7; }; }; }.a.b.n.m or 3    # Връща 3
```

Раздел 6

Оператори

Оператори

- Аритметически: `+`, `-`, `*`, `/`, unary `-`
- Логически: `<`, `<=`, `>`, `>=`, `==`, `!=`, `&&`, `||`, `!`
  - Импликация: `->`
  - Атрибутно множество съдържа ли атрибут: `?`

```
{ a = 3.14; } ? a    # Връща true
{ a = 3.14; } ? b    # Връща false
```

- Конкатенация: `+`, работи при низове и пътища (смесено също!)

```
"Hello" + "World" + /bin + /lib    # Връща "HelloWorld/bin/lib"
```

- Конкатенация на списъци: ++

```
[ 1 2 ] ++ [ 3 4 ]   # Връща [ 1 2 3 4 ]
```

- Обновяване на атрибутни списъци: //

Елементите на десния се добавят в левия или заменят стойностите от левия

```
{ a = 1; b = 2; } // { b = 3; c = 4; }
# Връща { a = 1; b = 3; c = 4; }
```

Раздел 7

Вградени неща

Вградени неща

- Имаме редица вградени функции в езика (главно свързани с функционално програмиране)
- Те се намират във вградената променлива `builtins`
- Няма да разглеждаме абсолютно всичко, за това си има документация
- Някои от тях не е нужно да се достъпват чрез индексиране в `builtins`.

Глобални `builtins`

derivation

Създава деривация, това е най-основната функция в изграждане на пакети. За нея ще говорим в следващата лекция.

abort

Приема низ, спира изпълнение и връща аргументът си като съобщение за грешка.

```
abort "Error message"
```

throw

Същото като `abort`, обаче позволява да бъде игнорирано в определени ситуации.

import

Приема път към Nix файл, оценява стойността му и я връща. Ако имаме `example.nix`:

```
1 + 2
```

Тогава

```
import ./example.nix    # Връща 3
```

Относно `with import <nixpkgs> { };`

Вече знаем какво прави онзи първи ред в нашия пакет!
Всички пакети се намират в едно атрибутно множество. `<nixpkgs>` е специална конструкция за път към изтеглените https://github.com/NixOS/nixpkgs/. `with` прави тези пакети свободно достъпни в последвалия израз.

Операции с низове

stringLength

Връща дължината на подадения низ.

substring

Приема начален индекс, дължина и низ. Връща съответния под-низ.

Операции с атрибутни множества

attrNames

Приема атрибутно множество, връща списък от всички атрибути в него.

attrValues

Приема атрибутно множество, връща списък от всички стойности в него.

removeAttrs

Приема атрибутно множество и списък с низове - имена на атрибути. Връща атрибутното множество без тези атрибути.

Операции със списъци

elem

Приема стойност и списък. Връща дали стойността се намира в списъка.

Операции с файлове

readFile

Подаваме път към файл и връща неговото съдържание като един дълъг низ

readDir

Връща имената на всички файлове/директории в подадения път и връща атрибутно множество, където всеки атрибут е името на файл/директория и стойността е `"regular"`, `"directory"`, `"symlink"` и `"unknown"`

Раздел 8

Въпроси?