Функцията `derivation`

Универсални конфигурации с Nix

Павел Атанасов Камен Младенов

15.04.2025

Преговор

- Писахме програми на Nix езика
- Написахме няколко много прости пакета на 03.04.2025

Раздел 1

Как от Nix езика стигаме до програми?

Как от Nix езика стигаме до програми?

- Последно създавахме пакети чрез `writeTextFile`, `writeShellApplication` и `mkShell`
- Нарекохме ги “trivial builders”
- Можем да ги мислим за опростения върху една по-основна функция

builtins.derivation

- `derivation` функцията (от `builtins`) е вградена функция, която приема описание как се създава нещо в `/nix/store`
- Тя е една от единици (вградени) функции, които директно влияят върху системата: създават директории, изпълняват команди, пренасят файлове, ...

Аргументи

- Един аргумент - атрибутно множество, задължително съдържащо
  - архитектура на системата
  - име на пакета
  - shell скрипт, който ще създаде резултатът на пакета от нулата

```
derivation {
  # Задължителни
  system = "x86_64-linux";
  name = "hello";
  builder = ./build-script.sh;
  # Опционално
  args = [ "a" "b" ];
  # и още много, рядко използвани опционални
}
```

- В нашия скрипт е достъпна променливата `out`
- Тя съдържа пътя в `/nix/store`, който се очаква да бъде **създаден от скрипта**

Раздел 2

Употреба

Пренаписваме `writeTextFile`

- Този израз:

```
myWriteTextFile {
  name = "something.txt";
  text = ''
    First line
      second line
    third line
  '';
}
```

- Трябва да създаде `/nix/store/HASH-something.txt` със съдържание:

```
First line
  second line
third line
```

Nix функцията myWriteTextFile.nix

```
myWriteTextFile = { name, text }:
  derivation {
    system = "x86_64-linux";
    inherit name;
    builder = ./myWriteTextFile.sh;
    args = [ text ];
  };
```

build скрипта myWriteTextFile.sh

```
#!/bin/sh
printf "$1" > "$out"
```

Пренаписваме `writeShellApplication`

- Този израз:

```
myWriteShellApplication {
  name = "page_lines";
  runtimeInputs = [ curl cloc ];
  text = "curl -s \"$1\" | cloc --force-lang=html -";
}
```

- Трябва да създаде изпълним `/nix/store/HASH-page_lines/bin/page_lines` с примерно съдържание:

```
#!/nix/store/HASH-bash-VERSION/bin/bash
export PATH="/nix/store/HASH-curl-VERSION/bin:..."
curl -s "$1" | cloc --force-lang=html -
```

- Нуждаем се от `/nix/store` пътищата на на 3 програми - `bash`, `curl` и `cloc`
- Програми в `nixpkgs` могат имплицитно да се конвертират към низове (защото са атрибутни множества с *метод* `__toString`)
- Тези низове сочат към `/nix/store/HASH-PROGRAM-VERSION`. Вътре, под `bin` директорията се намира изпълнимия файл.

Nix функцията myWriteShellApplication.nix

```
myWriteShellApplication = { name, text, runtimeInputs ? [] }:
  derivation {
    system = "x86_64-linux";
    inherit name;

    builder = ./myWriteShellApplication.sh;
    args = [
      toybox bash name text
      (builtins.concatStringsSep ":" (map (i: i + "/bin") runtimeInputs))
    ];
  };
```

build скрипта myWriteShellApplication.sh

```
mkdir -p "$out/bin"
cat <<EOF > "$out/bin/$3"
#!$2/bin/bash
export PATH="$5"
$4
EOF
chmod +x "$out/bin/$3"
```

Пренаписваме `mkShell`

- Този израз:

```
myMkShell {
  packages = [ curl cloc ];
  shellHook = "echo Hi";
}
```

- Трябва при `nix-shell ./myMkShell.nix` да отвори shell с програмите `curl` и `cloc` налични, и при отваряне да изпише “Hi”

- Преди казахме, че `nix-shell` “Позволява [...] да влезем в shell с всички зависимости на програмата”
- В `derivation` функцията не определяме зависимости
- На ниско ниво `nix-shell` **единствено изпълнява файла** `$stdenv/setup`

- Тук трябва да спомним един важен факт: всички атрибути, които се подават на `derivation` и не са сред предефинираните, **се зачитат за environment променливи**
- Тоест, трябва да дефинираме environment променлива на име `stdenv`, която е директория и която съдържа изпълним `setup` файл
- Допълнително, единствения начин да подадем данни на `setup` (за него нямаме `args` атрибут) е чрез задаване на други environment променливи

Nix функцията myMkShell.nix

```
myMkShell = { packages, shellHook ? "" }:
  derivation {
    system = "x86_64-linux"; name = "shell";
    builder = ./myMkShell/builder.sh;

    stdenv = ./myMkShell;

    PACKAGES = builtins.concatStringsSep
        ":"
        (map (i: i + "/bin") packages);
    HOOK = shellHook;
  };
```

setup скрипта myMkShell/setup

```
#!/bin/sh

export PATH="$PACKAGES"

$HOOK
```

Раздел 3

mkDerivation

mkDerivation

- На практика, никой не ползва `derivation` функцията директно (и затова казваме, че `nix-shell` ни слага в shell с всички зависимости на пакета)
- В `nixpkgs` имаме една често ползвана функция над `derivation` - `mkDerivation`, която идва с един голям предефиниран `builder` скрипт
- Използваме я по аналогичен начин: подаваме атрибутно множество с определени данни, описващи пакета. Обаче тук атрибутите са много повече.

Атрибути в аргументът на `mkDerivation`

- `name` за име на пакета
- `version` за версия, използваме `name-version` за да обозначим пакета в `/nix/store` (заедно с хешът)
- `src` за **път** в който се намира изходния архив
  - Предоставят ни се механизми да изтегляме и менажираме програми/файлове. За това по-късно.

Фази

- Най-съществената система при `builder` скрипта на `mkDerivation` е идеята за фази
- “Компилационния” процес се разделя на редица големи универсални стъпки: разархивираме сорс-кода, компилираме го, ...

0. **Unpack**: разархивиране на входните файлове
1. **Patch**: правят се промени по изтеглените сорс-файлове
2. **Configure**: конфигурира се средата за компилиране, примерно генериране на конфигурационни файлове които ще са полезни по време на компилация
3. **Build**: пуска се нужната програма за компилиране
4. **Check**: пускат се всички тестове, с целта да се провери дали компилацията е създала коректен резултат
5. **Install**: резултатните файлове се слагат под `/nix/store`
6. **Fixup**: финални (nix-specific) промени, примерно премахване на дебъг данни от резултата

Атрибути свързани с фази

- Всяка фаза се определя от атрибутът `${name}Phase` (където `${name}` варира през `unpack`, `patch`, `configure`, ...)
- Стойността му е низ, съдържащ shell скрипт с нужните действия (вместо да имаме файлове, имаме низове)
- `pre${name}` и `post${name}` съответно се изпълняват преди или след дадената фаза (примерно `postBuild`, `preCheck`). Особено полезно за функции над `mkDerivation`.

- `dont${name}` е булева стойност, отразяваща дали дадената фаза да се изпълни.
  - Валидно е само за фазите, които се изпълняват по подразбиране
  - За другите (`check`), имаме `do${name}`
- Някои фази имат свои си атрибути:
  - Patch Phase: `patches`, списък от patch файлове които ще се сложат върху сорса
  - Configure Phase: `configureScript`, име на конфигурационен скрипт

Раздел 4

Употреба

Употреба

- На практика `writeTextFile`, `writeShellApplication` и `mkShell` са дефинирани върху `mkDerivation`, не `derivation`
- Нека да ги напишем ръчно още един път. Сега ще е по-лесно.

writeTextFile

Функцията new_myWriteTextFile.nix

```
myWriteTextFile = { name, text }:
  stdenv.mkDerivation {
    inherit name;
    src = ./.;

    dontBuild = true;
    installPhase = ''
      printf "${text}" > "$out"
    '';
  };
```

writeShellApplication

Функцията new_myWriteShellApplication.nix

```
myWriteShellApplication = { name, text, runtimeInputs ? [] }:
  stdenv.mkDerivation {
    inherit name;
    src = ./.;

    buildInputs = [ toybox ];
    nativeBuildInputs = [ bash ] ++ runtimeInputs;

...
```

Функцията new_myWriteShellApplication.nix

```
...
    buildPhase = let
      runtimePath = builtins.concatStringsSep
        ":"
        (map (i: i + "/bin") runtimeInputs);
    in ''
      cat <<EOF > ./myScript
      #!$(which bash)
      export PATH="${runtimePath}:\$PATH"
      ${text}
      EOF
      chmod +x ./myScript
    '';
...
```

Функцията `new_myWriteShellApplication.nix`

```
...
    installPhase = ''
      mkdir -p "$out/bin"
      mv ./myScript "$out/bin/${name}"
    '';

    dontFixup = true;
  };
```

mkShell

Функцията new_myMkShell.nix

```
myMkShell = { packages, shellHook ? "" }:
  stdenv.mkDerivation {
    name = "shell";
    src = ./.;

    inherit shellHook;
    nativeBuildInputs = packages;

    phases = [ "installPhase" ];
    installPhase = ''
      mkdir -p "$out"
    '';
  };
```

Раздел 5

Fetcher функции

Fetcher функции

- В момента `src` го слагаме на `./.`
- В реалния свят, почти няма програми, които да идват с готов, конфигуриран Nix
- Искаме `src` да бъде нещо от интернет: файл, GitHub хранилище, ...
- За тази цел са създадени fetcher функциите

  Няма да ги разглеждаме чак толкова подборно колкото `derivation`

Възпроизводимост

- Обаче, тегленето от интернет носи проблем със себе си - как запазвам възпроизводимостта си?
- Не можем да сме сигурни, че ресурсът на даден линк ще остане същия вовеки веков
- Понеже този ресурс е част от входът, то той определя хеш. Ако ресурса на дадения линк се промени, то и хешът ще се спрямо това кога пускаме `nix-build`.
- Решение няма!
- Има заобиколка обаче: ще съхраняваме хеша на ресурса и ще го проверяваме. Така поне ще знаем ако се е променил.

- Как ще намерим хеша?
- Можем ръчно през `nix-hash`, но това означава допълнително да си теглим ресурса, да пускаме командата, и т.н.
- На практика, всички оставят хеша празен. Nix гръмва с грешка “очаквах този хеш”, и така направо копираме това, което Nix иска.
- Тази идея се разглежда по-подробно от така-наречените Fixed-Output Derivations, които няма да разглеждаме изкъсо сега.

fetchurl

- Тегли файл от даден линк. Приема атрибутно множество с
  - `url` - линк към дадения файл
  - `hash` - хеш за файла

Примерно извикване

```
fetchurl {
    url = "https://example.com/file.txt";
    hash = "sha256-lTeyxzJNQeMdu1IVdovNMtgn77jRIhSybLdMbTkf2Ww=";
}
```

Пример

- Нека направим деривация, която изтегля една уеб страница, конвертира я към нормален текст, който го записва в резултатния файл
- Конвертирането може да стане с командата `html2text`
- Нека страницата да бъде `https://danluu.com/web-bloat/index.html` и да кръстим пакета “web-bloat”
- Нека приемем, че името на резултатния файл няма кавички в себе си, т.е. можем директно да го интерполираме в шел команда (нормално е хубаво да следим за тези неща и да ползваме функции от `nixpkgs`, като `escapeShellArg`, че да се предпазваме от injection атаки)

Пакетът web-bloat.nix

```
stdenv.mkDerivation {
  name = "web-bloat";

  src = fetchurl {
    url = "https://danluu.com/web-bloat/index.html";
    hash = "sha256-4j9kUV4xyYKhjFjV0nC5SbdyDmSFCSEXyd0MYXvDF2s=";
  };

  unpackPhase  = "cp \"$src\" ./index.html";

  buildInputs  = [ html2text ];
  buildPhase   = "html2text index.html > page.txt";
  installPhase = "mv page.txt \"$out\"";
}
```

`fetchzip`

- Почти винаги сорс код може да се предостави във формата на (zip, tar, 7zip, ...) архив
- `fetchzip` работи почти като `fetchurl`, обаче също очаква подадения URL да е към архив, който той да разархивира
- Автоматично се усеща за типа архив. Въпреки името, поддържа всякакви архивни формати.

Пример

- Нека да направим деривация за `mawk` интерпретатора (версия `1.3.4`).
- Ще изтеглим сорс кода от `https://invisible-mirror.net/archives/mawk/mawk-1.3.4-20240819.tgz`
- Трябва ни `gcc` компилатора. Идва със скрипт, който се прилага от `make` командата.

Пакетът mawk.nix

```
stdenv.mkDerivation {
  name = "mawk";
  version = "1.3.4";
  src = fetchzip {
    url = "https://invisible-mirror.net/archives/mawk/mawk-1.3.4-20240819.tgz"
    hash = "sha256-zlJN4oOFBTOM4/mjYAgk5PifnKn61dFQoxFBpyuVz4w=";
  };

  buildDependencies = [ gnumake gcc ];
  buildPhase = "make";
  installPhase = "
    mkdir -p "$out/bin"
    mv mawk "$out/bin"
  '';
}
```

- Всъщност, много програми се компилират само с `gcc` и определят нужните скриптове за `make`
- Заради това, `mkDerivation` по подразбиране е готов да компилира такива проекти (и програмите са винаги налични)
- Реално, нашия пакет може да се сведе до:

```
stdenv.mkDerivation {
  name = "mawk";
  version = "1.3.4";

  src = fetchzip {
    url = "https://invisible-mirror.net/archives/mawk/mawk-1.3.4-20240819.
    hash = "sha256-zlJN4oOFBTOM4/mjYAgk5PifnKn6ldFQoxFBpyuVz4w=";
  };
}
```

`fetchFromGitHub`

- В модерно време все повече и повече програми се намират в хранилища като GitHub
- Целият сорс-код на Nix за всичките му компоненти се намира в GitHub
- Има си специална функция за GitHub.
- Вместо `url`, тя приема `owner`, `repo` и `rev`, съответно за потребителското име, името на хранилището и git commit-а или tag

Пример

- Нека да направим деривация за този проект:
  https://github.com/meta-rust/rust-hello-world
- За да се компилира изисква `rust` компилатора и `cargo` пакетния мениджър (но няма зависимости)

Пакетът rust-hello-world.nix

```
stdenv.mkDerivation {
  name = "rust-hello-world";
  version = "0.0.1";

  src = fetchFromGitHub {
    owner = "meta-rust";
    repo = "rust-hello-world";
    rev = "e0fa23f1a3cb1eb1407165bd2fc36d2f6e6ad728";
    hash = "sha256-xUyi1sgKhxvysMTS6tiSape+H4yHHLXjplAH4cD2Yb8=";
  };

  buildInputs = [ cargo ];
  buildPhase = "cargo build --release";
...
```

Раздел 6

Въпроси?