# Функции над `mkDerivation`

## Универсални конфигурации с Nix

Павел Атанасов  Камен Младенов

22.04.2025

# Преговор

- Разглеждахме `builtins.derivation`, как се ползва, какво прави
- Разглеждахме `stdenv.mkDerivation`
- Разглеждахме fetcher функции
- Писахме не малко пакети, използвайки `builtins.dervivation` и `stdenv.mkDerivation`

# Раздел 1

## Още функции

# Още функции

- Оказва се, че дори `mkDerivation` е на твърде ниско ниво
- Повечето пакети използват функции, писани над `mkDerivation`
- Познаваме някои от тях: trivial builders като `writeTextFile` или `writeShellApplication`
- Останалите са специализирани за конкретни програмни езици и техните компилационни системи

# Системи за менажиране на зависимости

- Почти всички модерни програмни езици идват със свои системи за менажиране на зависимости
- Това са NuGet в езика C#, cargo в Rust, setuptools в Python и так. нат.
- В чистата среда на Nix няма интернет.
  1. Трябва Nix предварително да осигури зависимостите
  2. Трябва тези системи да се конфигурират да използват предоставените зависимости

- Това е една от най-големите трудности при създаване на Nix пакети
- Още по-сложно става когато има зависимости, писани на други езици, и за една програма се пускат няколко различни менажиращи системи
- Ситуацията се влошава, защото тези системи само се разрастват

# Една деривация става две

- Все пак се разработват лесни за използване функции над `mkDerivation`
- Повечето разделят процеса на две деривации:
  1. Първата изтегля и конфигурира всички зависимости
  2. Втората компилира самата програма

# Раздел 2

## Go

# Go

- Проекти писани на езика Go се разделят на “Go модули”
- Всеки модул описва зависимостите си в `go.mod` файл
- Използваме `buildGoModule`, приема всички атрибути които и `mkDerivation` би, заедно с:
  - `vendorHash` - хеш на зависимостите на модула
  - `modRoot` - директорията в `src`, където Go модулът се намира. `"./"` по подразбиране.
  - `modPostBuild` - фаза след като зависимостите са изтеглени и преди `vendorHash` е изчислено/проверено. Позволява ни да изменим зависимости.

# fzf

- `fzf` е програма за размито търсене
- Намира се на https://github.com/junegunn/fzf
- Да направим пакет за версия 0.60.3

## Пакет fzf.nix

```
buildGoModule rec {
  name = "fzf"; version = "0.60.3";

  src = fetchFromGitHub { ... };

  vendorHash = "sha256-...=";
}
```

# ts

- `ts` е програма за командния ред, която конвертира между различни формати дата и час
- Намира се на https://github.com/liujianping/ts
- Да направим пакет за версия 0.0.7

- Go модули може да си идват със зависимостите (под директорията `vendor`). Тогава `vendorHash` **трябва** да е `null`.
- "Check" фазата е включена по подразбиране, може да ни донесе проблеми със собствени зависимости. Изключваме я с `doCheck`.
- Този проект има проблем: за една зависимост, във `vendor` пише една версия, но в `go.mod` друга. Трябва да закърпим файла.

## Пакет ts.nix

```
buildGoModule rec {
  name = "ts"; version = "0.0.7"; src = fetchFromGitHub { ... };

  postPatch = ''
    sed -i '/github.com\/x-mod\/errors/s/5/6/' ./vendor/modules.txt
  '';

  vendorHash = null;

  doCheck = false;
}
```

# Раздел 3

## Rust

# Rust

- Организацията е подобна като тази при Go
- Rust проекти се разделят на “crates”
- Използваме buildRustPackage, като може да приеме допълнителните атрибути:
  - `cargoHash` - хеш на зависимостите на crate-а
  - `cargoLock` - атрибут, описващ cargo lockfile от който да се вземат зависимости (ако такъв не съществува)

# cloak

- Програма за OTP автентикация на командния ред
- Намира се на https://github.com/evansmurithi/cloak
- Нека да направим пакет за версия 0.3.0

## Пакет cloak.nix

```
rustPlatform.buildRustPackage rec {
  name = "cloak"; version = "0.3.0";

  src = fetchFromGitHub { ... };

  cargoHash = "sha256-...=";
}
```

# Lockfiles

- “Lockfile” е файл, съдържащ всички зависимости, зависимости на зависимости и т.н., с точните им версии и места от където са взети
- Правят работата на уредите за менажиране на зависимости (включително Nix) много по-лесна
- В Go можем да зачетем, че този файл е `go.sum`
- По принцип този файл се включва като част от сорс-кода

# Cargo.lock

- В Rust този файл е `Cargo.lock`
- При по-стари версии на Rust се е казвало да не се включва в сорс-кода
  - Можем да го включим през `cargoLock.lockFile`
  - Понеже Nix ще знае всички зависимости, `cargoHash` става излишно. Неговата стойност не се ползва и можем да го махнем.
  - Обаче по време на компилация ще се нуждаем от `Cargo.lock`, трябва да го копираме вътре

# Добавяне на наш Cargo.lock

1. Изтегляме сорс-кода
2. Влизаме в shell с `cargo`

   ```
   nix-shell -p cargo
   ```

3. Генерираме `Cargo.lock`

   ```
   cargo generate-lockfile
   ```

4. Подаваме този файл като атрибут `cargoLock.lockFile`

   ```
   cargoLock = { lockFile = ./Cargo.lock; };
   ```

5. Вмъкваме файла в сорс кода

   ```
   postPatch = "cp ${./Cargo.lock} Cargo.lock";
   ```

# lorsrf

- lorsrf е уред за мрежово тестване за проникване
- Намира се на https://github.com/MindPatch/lorsrf
- Нека да направим пакет за най-новата версия, по commit 5c6945

- Някои програми се нуждаят от системни библиотеки, като `openssl`
- Такива случаи трябва да добавим `openssl.dev` като `buildInput`. Само `openssl` не е достатъчно, защото предоставя само изпълнимия файл, без библиотечните файлове.

- Проблем при системните библиотеки е, че трябва да се търсят в системата и след това да се подаде коректен аргумент на компилатора
- На ръка, това е досадно и не е универсално
- Повечето системи ползват `pkg-config`, което върши тази работа
- За да го ползваме, **трябва** да го добавим в `nativeBuildInputs`
- `nativeBuildInputs` е за зависимости само в средата за компилиране. `buildInputs` хем определя зависимост за средата, хем зависимостта ще бъде налична по време на изпълнение.

## Пакет lorsrf.nix

```
rustPlatform.buildRustPackage rec {
  name = "lorsrf"; version = "latest"; src = fetchFromGitHub { ... };

  postPatch = ''
    cp ${./lorsrf-Cargo.lock} Cargo.lock
  '';

  cargoLock.lockFile = ./lorsrf-Cargo.lock;

  buildInputs = [ openssl.dev ];
  nativeBuildInputs = [ pkg-config ];
}
```

# Раздел 4

## C#

# C#

- Подобно на Go, проекти писани на C# се разделят на модули
- Зависимостите са определени от `.sln` и `.csproj` файлове. Пакетния мениджър се нарича `NuGet`.
- Използваме `buildDotnetModule`, приема всички атрибути които и `mkDerivation` би, заедно с:
  - `projectFile` - име на `.csproj` или `.sln` файл. По подразбиране се взема този в `"./"` директорията.
  - `nugetDeps` - JSON файл, съдържащ всички зависимости

# NuGet и lockfiles

- В C# има нещо подобно на lockfile, обаче то не *трябва* да се включва в сорса
- И вместо да е един файл, всъщност е директория с директории за всяка зависимост, всяка от която има голямо множество файлове
- Трябва ръчно да създадем (и менажираме) наш lockfile

# Процедурата за генериране на lockfile

1. Изтегляме сорс-кода
2. Влизаме в shell с `dotnet-sdk` и `nuget-to-json`

   ```
   nix-shell -p dotnet-sdk nuget-to-json
   ```

3. Пускаме генерацията на онази директория (и нека тя се казва `out`)

   ```
   dotnet restore --packages out
   ```

4. Конвертираме към нашия lockfile чрез `nuget-to-json`

   ```
   nuget-to-json out > deps.json
   ```

5. Подаваме този файл като атрибут `nugetDeps`

   ```
   nugetDeps = ./deps.json;
   ```

# WaveFunctionCollapse

- Програма за генериране на определен вид изображения
- Намира се на https://github.com/mxgmn/WaveFunctionCollapse
- Нека да направим пакет за най-новата версия, по commit a08888

## Пакет WaveFunctionCollapse.nix

```
buildDotnetModule {
  name = "WaveFunctionCollapse"; version = "2024.12.14";

  src = fetchFromGitHub { ... };

  nugetDeps = ./WaveFunctionCollapse-deps.json;
}
```

# Раздел 5

## Nim

# Nim

- На практика, положението е същото като при C#
- Пакетния мениджър се казва `nimble`, зависимости се определят в `.nimble` файл
- Използваме buildNimPackage с допълнителен атрибут:
  - `lockFile` - път към JSON lockfile
- `nim_lk` директно генерира JSON lockfile от `.nimble` файл. Това е единствената команда, която трябва да пуснем.

# lester

- Програма за генериране на HTML файлове от Markdown
- Намира се на https://github.com/madprops/lester
- Нека да направим пакет за най-новата версия, по commit f6a0d9

## Пакетът lester.nix

```
buildNimPackage {
  name = "lester"; version = "latest";

  src = fetchFromGitHub { ... };

  lockFile = ./lester-lock.json;
}
```

# Раздел 6

## Python

# Python

- За разлика от останалите, Python няма lockfile-ове
- Също пакетните мениджъри не са унифицирани.
  - Има някои по-базови: setuptools, distlib, ...
  - И някои по-мощни: pip, PyPI, ...
- Допълнително има десетина различни версии на Python, които се поддържат паралелно

- В повечето случаи, има два варианта:
  - съдържа `setup.py`
  - съдържа `pyproject.toml`
- Използваме buildPythonPackage с допълнителни атрибути:
  - `pyproject` - булево, дали да използва `pyproject.toml` ако такъв съществува
  - `build-system` - списък от поне `setuptools`, когато се използва `setup.py`
  - `dependencies` - Python зависимости, намират се под `pythonXPackages`. Тук Nix менажира зависимостите.

# rpi-backlight

- Програма за контролиране на “Raspberry Pi 7” екран
- Намира се на https://github.com/linusg/rpi-backlight
- Да направим пакет за версия 2.7.0

## Пакет rpi-backlight.nix

```
with python3Packages;
buildPythonApplication rec {
  name = "rpi-backlight"; version = "2.7.0";

  src = fetchFromGitHub { ... };

  build-system = [ setuptools ];
}
```

# english_text_normalization

- Програма за нормализиране на формат на текст
- Намира се на https://github.com/jasminsternkopf/english_text_normalization
- Да направим пакет за версия 0.0.3

## Пакет english_text_normalization.nix

```
with python3Packages;
buildPythonApplication rec {
  name = "english_text_normalization"; version = "0.0.3";
  src = fetchFromGitHub { ... };

  pyproject = true;

  dependencies = [ setuptools pyenchant nltk inflect unidecode ];
}
```

# Раздел 7

## Въпроси?