Théorie

Comment fonctionne Nix. Je trouve ça intéressant d’abord parler de la théorie avant de passer à la pratique, car ça aide à mieux apprécier et comprendre Nix.

- Le langage
- Les dérivations
- Profiles
- Garbage Collector
- Channels et nixpkgs
- Le cache (ou "nix store partagé")
- Flakes

Le langage

Tout d'abord il est intéressant de comprendre comment fonctionne le langage Nix car c'est vraiment la base de tout le reste. On va donc passer en revue les bases nécessaires de la programmation fonctionnelle et la syntaxe de Nix.

On va d'abord passer en revue toutes les bases du langage, puis on va voir la fonction `derivation`, c'est là que ça va devenir vraiment intéressant.

> **Tip** : Utilisez la commande `nix repl` pour tester la syntaxe de Nix et les exemples donnés ici.

Types de données

Voici la liste des types primitifs et comment ils sont représentés en Nix :

Type	Description	Exemple
String	Chaine de caractère	`''hello world''` ou `"hello world"`
Nombre	Nombre entier ou non	`123` ou `12.4`
Chemin	Chemin vers un fichier ou dossier (doit contenir un `/`)	`./builder.sh` ou `/bin/bash`
Booléen	Vrai ou faux	`true` ou `false`
Null	Rien du tout	`null`

Ensuite voici les autres types :

Type	Description	Exemple
Liste	Liste d'expressions Nix (notez qu'il n'y a pas de `,` mais seulement un espace)	`[ 123 ./foo.bar "abc" ]`
Ensemble d'attributs	Association de clé-valeur	`{ x = 123; y = 56; file = ./builder.sh; name = "something"; }`, `{ x = 123; y = 56; }.x` retourne 123

Les différentes expressions

- Les commentaires se font soit sur une ligne avec `#`, soit avec `/* */`

```
nix-repl> # hello world
        /* hello world */
        1 + 1
2
```

- Ensemble d'attributs récursifs qui permettent de référencer d'autres attributs du même set depuis le set lui-même

```
# x référence y, et y référence 123
rec {
  x = y;
  y = 123;
}.x # retourne 123
```

- Expressions "let-in" qui permet de définir des variables ou fonctions locales qui sont ensuite utilisées dans le "in"

```
# Le let défini les variables locales x et y
let
  x = "foo";
  y = "bar";

# Qui sont ensuite utilisée dans le in, ce qui donne donc "foobar"
in x + y
```

- L'héritage des attributs avec `inherit` permet d'éviter de se répèter

```
#  Variable locale x et y
let
  x = 123;
  y = 456;

# Dans un set
in {
  inherit x y; # équivaut à dire "x = x; y = y;"
  # soit définir l'attribut "x" du nouveau set comme étant la valeur de la variable locale "x" définie dans let
```

```
} # ceci retournera donc { x = 123; y = 456; }
```

```
# Donc
let
  x = 123;
  y = 456;
in { inherit x y; }

# est la même chose que
let
  x = 123;
  y = 456;
in {
  x = x;
  y = y;
}
```

- On peut aussi créer des fonctions que l'on peut mettre dans des variables

```
# concat est le nom de la variable
# x et y sont les paramètres de la fonction
# x + y est le corps de la fonction
nix-repl> concat = x: y: x + y

# On peut appeller la fonction en donnant le nom de la fonction, puis ses attributs séparés par des espaces
nix-repl> concat "hello " "world"
"hello world"

# Mais on peut aussi répondre à moitié à la fonction (pour plus d'info, renseignez-vous sur la programmation fonctionnelle
nix-repl> half-concat = concat "hello"
nix-repl> half-concat "world"
"hello world"

# Mais on peut aussi utiliser des sets d'arguments lorsque l'on veut tous les attributs d'un coup
nix-repl> concat = { a, b }: a + b
nix-repl> concat { a = "hello "; b = "world"; }
"hello world"
```

- Les conditions ne se font que par des genre de ternaires `if then else`

```
nix-repl> if true then "hello" else "foo"
"hello"
```

- Pour empécher encore plus la redondance de code, on peut utiliser `with` qui va en quelque sorte "transformer" les attributs d'un set en variables locales

```
# Ceci va retourner { a = "foo"; b = "bar"; }
let
  my-set = { x = "foo"; y = "bar"; };
in
  with my-set; {
    a = x;
    b = y;
  }

# Ceci va aussi retourner { a = "foo"; b = "bar"; }
let
  my-set = { x = "foo"; y = "bar"; };
in
  {
    a = my-set.x;
    b = my-set.y;
  }
```

- Pour simplifier les choses il est aussi possible d'utiliser l'interpolation de strings

```
nix-repl> foo = "hello"
nix-repl> bar = "world"

# ...pour les chemins de fichiers ou dossier
nix-repl> ./${foo}-${bar}.nix
/home/snowcode/hello-world.nix

# ...pour les strings
nix-repl> "${foo}"
"hello"
nix-repl> ''${bar}''
"world"

# ...pour les clé des attributs
```

```
nix-repl> { ${foo} = "foo"; ${bar} = "bar"; }
{ hello = "foo"; world = "bar"; }
```

Quelques opérations

Par convention ici les set d'attributs seront appellé `attrset`, les noms des clés seront appellé `attrpath`.

Opération	Syntaxe	Retourne
Sélection d'un attribut	`attrset.attrpath`	La valeur
Test si a un certain attribut	`attrset ? attrpath`	Booléen
Concaténation de string	`string + string`	String
Concaténation de path	`path + path`	Path
Concaténation de path et string	`path + string`	Path
Concaténation de string et path	`string + path`	String
Combiner un set avec un autre (le deuxième à priorité si conflit)	`attrset1 // attrset2`	Set d'attributs
Comparaisons	`< == > <= >= !=`	Booléen
Opérateurs booléens	`|| && !`	Booléen

Les dérivations

Les dérivations dans Nix sont la base des builds des applications. Elle se basent sur la fonction `derivation`, cependant vous n'utiliserez probablement jamais cette fonction directement car elle est bien trop basique. C'est pour cela que des fonctions tel que `mkDerivation` du package `stdenv` sont là, ce sont des fonctions basée sur `derivation` mais qui rendent les choses plus simple et plus sûres pour les mainteneurs de paquets.

Nous allons voir ici comment la fonction de base `derivation` fonctionne, ce qu'elle fait et comment les builds fonctionnent dans Nix.

La syntaxe de `derivation`

Cette fonction prends comme argument un set d'attribut avec les attributs suivant :

- `system`, qui précise l'architecture du paquet
- `name`, qui précise le nom du paquet
- `builder` qui est le programme utilisé pour construire le paquet (cela peut être une autre dérivation, ou un chemin de fichier)
- (optionnel) `args` qui est une liste des arguments à passer au *builder*
- (optionnel) `outputs` qui spécifie la liste symbolique des outputs (par exemple `outputs = [ "lib" "doc" ]`. Par défault, Nix crée une variable d'environnement `$out` qui spécifie la localisation du dossier dans le Nix store, mais on peut très bien en ajouter d'autres tel que `$lib` ou `$doc`

Ce qui se passe lors d'un build d'une dérivation

Dans cet exemple regardons ce qu'il se passe si on build la dérivation suivante :

```
nix-repl> :l <nixpkgs>
nix-repl> my-derivation = derivation { name = "foo"; builder = "${bash}/bin/bash"; args = [ ./builder.sh ];
system = builtins.currentSystem; }
nix-repl> :b my-derivation
```

1. Déplacement de tous les fichiers nécessaires dans le store

Tout d'abord Nix va détecter que `./builder.sh` est un chemin de fichier dans les arguments et va le passer dans le Nix Store. Pour cela il va le hash d'une certaine manière et y ajouter le nom du fichier à la fin.

Il en va de même si on précise un dossier de source, etc.

2. Création d'un `.drv`

Ensuite Nix va créer un fichier `.drv` dans le nix store.

Si on utilise la commande `nix show-derivation` sur ce fichier, on peut le lire, ce qui nous donne ceci :


```
[snowcode@snowcode ~]$ nix show-derivation /nix/store/yg5ps2gvx39l38w90iyxss0xalcqka0y-foo.drv
{
  "/nix/store/yg5ps2gvx39l38w90iyxss0xalcqka0y-foo.drv": {
    "args": [
      "/nix/store/qfs6b6gcq9xi6gpf11vribfir8xn2nln-builder.sh"
    ],
    "builder": "/nix/store/561wgc73s0x1250hrgp7jm22hhv7yfln-bash-5.2-p15/bin/bash",
    "env": {
      "builder": "/nix/store/561wgc73s0x1250hrgp7jm22hhv7yfln-bash-5.2-p15/bin/bash",
      "name": "foo",
      "out": "/nix/store/vk7mk50mqswblpi88l86lf40bmrlffs1-foo",
      "system": "x86_64-linux"
    },
    "inputDrvs": {
      "/nix/store/0hnjp6s8k71xm62157v37zg3qzwvl8lx-bash-5.2-p15.drv": [
        "out"
      ]
    },
    "inputSrcs": [
      "/nix/store/qfs6b6gcq9xi6gpf11vribfir8xn2nln-builder.sh"
    ],
    "outputs": {
      "out": {
        "path": "/nix/store/vk7mk50mqswblpi88l86lf40bmrlffs1-foo"
```

```
      }
    },
    "system": "x86_64-linux"
  }
}
```

Ce fichier contient toutes les informations pour build un programme, sans tout les machins de Nix. Il contient les choses suivantes :

- Les "out paths", comme on en a parlé avant par défault il n'y en a qu'un c'est le `$out`. Le hash est calculé en faisant un hash spécifique du .drv ayant un out path vide.
- Les dérivations en input et les fichiers de source, ici nous avons bash (car nous y avons fait référence dans la dérivation) et le builder.sh (car nous avons mis son chemin). Nix va donc récupérer leur chemin dans le Nix store
- Le système, l'exécutable (builder) et ses arguments
- Les variables d'environnement qui sont passées au builder

Tous les chemins fonctionnant avec des hash des différentes dérivations et fichiers, cela assure donc qu'une même dérivation donnera toujours le même résultat quoi qu'il arrive.

3. Le build de la dérivation

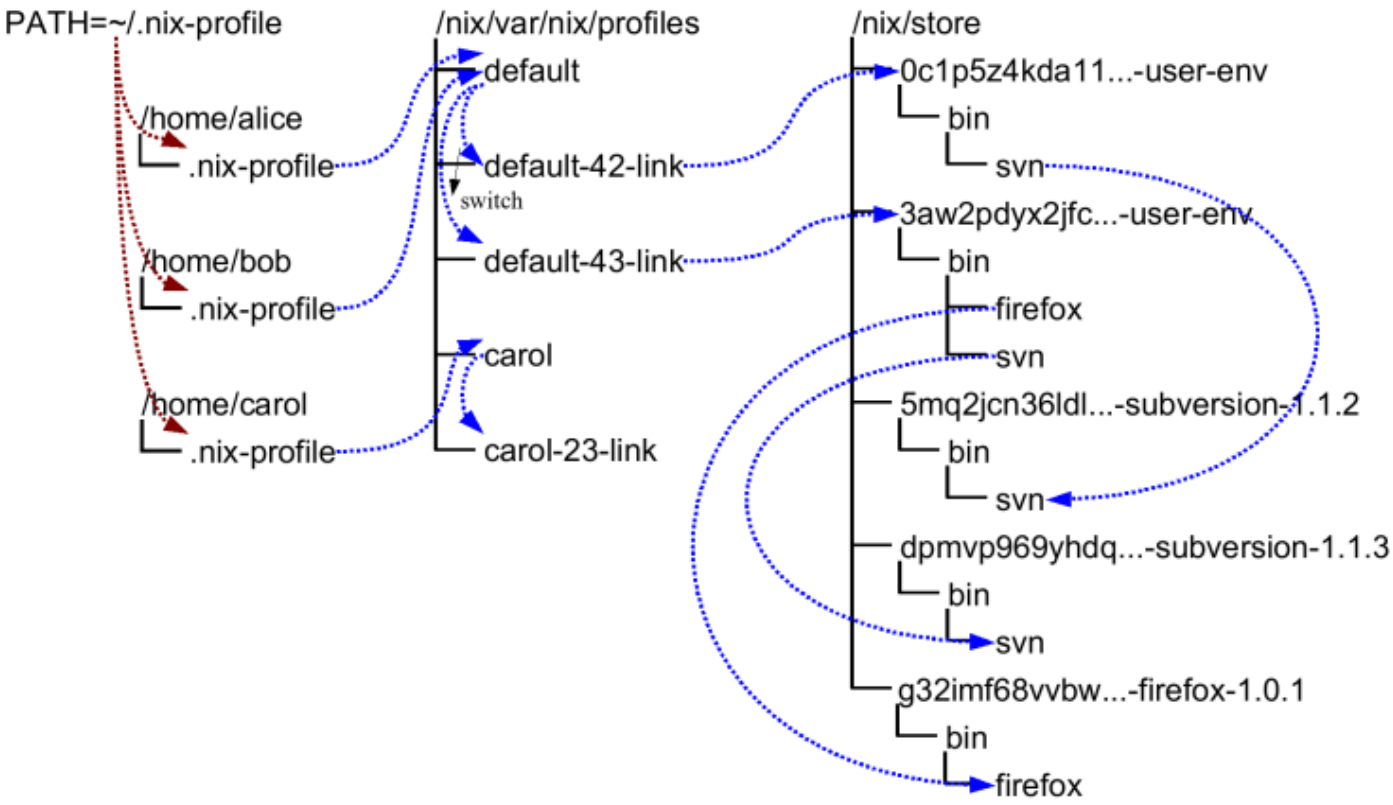
Ensuite grâce au fichier intermédiaire `.drv`, Nix n'a plus qu'a ajouter les variables d'environnement et exécuter le builder avec les bons arguments (et uniquement celles là).

Une fois cela fait, un raccourcis vers le dossier de la dérivation dans le Nix store est créé sous le nom de "result" dans le dossier courrant.

Profiles

Les profiles sont là pour isoler la configuration de chaque utilisateur·ice et permettre de revenir en arrière à tout moment. Chaque profile (et chaque *génération* de ce profile est un raccourcis vers `/nix/var/nix/profiles/`.

A chaque changement dans l'environement de l'utilisateur·ice (avec `nix-env` par exemple), une nouvelle génération est crée depuis la précédente.



Pour illustrer tout ceci, voici ce qu'il se passe lorsque Carol installe Svn :

1. La dérivation est installée dans le Nix store (soit en la compilant soit en la téléchargeant depuis le cache)
2. Une nouvelle dérivation est crée depuis la génération précédente de Carol, dans cette nouvelle dérivation `svn` est lié dans un dossier `bin` (pour imiter la structure des fichiers de Linux)
3. Cette nouvelle dérivation est liée dans le dossier `/nix/var/nix/profiles` comme une nouvelle dérivation de Carol (par exemple `carol-24-link`)
4. Cette nouvelle dérivation est liée dans le profile de carol (`carol`)
5. Enfin le profile de carol est liée dans son dossier `~/.nix-profile`
6. Enfin tous les fichiers dans `~/.nix-profile/bin` sont ajouté à son `$PATH` pour être accessible à Carol. Carol peut maintenant utiliser Svn.

Maintenant lorsque Carol veut rollback vers sa génération précédente, il suffit simplement de remplacer le lien de `carol` vers `carol-23-link` à la place de `carol-24-link`.

Ainsi faire un rollback se fait en une seule opération. Par contre ce système à le désavantage de prendre beaucoup de place car rien n'est supprimé. Dans la page suivante, nous allons voir comment le "garbage collector" fonctionne pour pouvoir faire de la place.

Garbage Collector

Maintenant je pense qu'il est intéressant de se demander ce qu'il se passe lorsque l'on veut faire de l'espace sur le système.

Ce qu'il se passe pour faire de l'espace c'est que lorsque un élément du Nix store est lié quelque part, un lien se fait dans le dossier `/nix/var/nix/gcroots/`. Ainsi lorsque l'on lance le garbage collector, il va supprimer tous les éléments qui ne sont pas référencé dans le dossier `gcroots`.

Pour toujours permettre un rollback, tant qu'un profile a des générations comportant certains paquets, ces paquets sont aussi des `gcroots` et ne peuvent donc pas être supprimé avant que leur génération associée soit supprimée.

Lorsque l'on lance `nix-collect-garbage` c'est qu'il va regarder récursivement tous les raccourcis vers le Nix Store qui sont dans gcroots, pour chaque fichier n'y étant pas il va le déplacer dans le dossier `/nix/store/trash`. Une fois tous transféré, les éléments sont supprimés.

Channels et nixpkgs

Les *channels* sont les repos de Nix. Un channel c'est simplement une URL qui pointe vers un endroit ayant plein d'expressions Nix et un manifeste. Les channels les plus populaires (et probablement les seuls à vraiment être utilisé) sont ceux de *nixpkgs* (*nixpkgs unstable* et *nixpkgs 23.05* par exemple)

Globalement Nixpkgs est un repo git dont chaque branche est un channel. On "s'abonne" à un channel et à chaque fois que l'on veut télécharger un paquet, on va d'abord essayer d'aller chercher sa dérivation dans ses channels.

Plus tard on va parler des *flakes* qui sont un moyen alternatif de proposer des paquets Nix de manière décentralisée (contrairement aux channels).

Overlays

Il est possible d'appliquer des modifications à Nixpkgs via des *overlays* . Cela peut permettre d'ajouter de nouveau paquets ou de modifier des existants.

Le cache (ou "nix store partagé")

Pour éviter de build plusieurs fois le même fichier et de perdre du temps à ça, il est possible de mettre en place un "cache", le nix store partagé. C'est à dire que tous les fichiers compilés, identifiés par leur hash de dérivation est stoqué et qu'avant d'essayer de build quelque chose, Nix va d'abord aller voir si le fichier déjà compilé n'existe pas déjà.

Ce système existe déjà pour tous les paquets de *nixpkgs*, c'est le `cache.nixos.org`. Ce cache fait plus de 400TiB et devient compliqué à maintenir (les couts de serveur s'élèvent à 108 000€ par an). Mais c'est grâce à ce cache que Nix peut permettre d'a la fois build les paquet depuis leur source, et d'à la fois télécharger directement leur fichiers binaires.

Certains membres de la communauté pensent peut'être à essayer de décentraliser le cache mais cela est toujours en discussion et rien n'est encore concret.

Pour le moment les 3 seuls moyen officiel de partager un nix store c'est par SSH, HTTP et S3.

Flakes

Les Nix Flakes sont des dossiers (généralement des repo git) ayant à leur racine un fichier `flake.nix`. Ce fichier indique des "inputs" et des "outputs". Ainsi il permet de prendre des expressions Nix en entrée et donner toutes sortes de sorties.

Par exemple, on peut construire un flake qui:

- Setup un environnement de dev
- Build le paquet
- Crée une image docker minimales
- Build et effectue des tests sur le résultat
- Crée des configurations NixOS
- Crée des overlays de nixpkgs

et bien d'autres.

Un flake a également l'avantage par rapport aux channels, d'être vraiment reproductible. Lorsqu'un flake est créé et lancé pour la première fois, un fichier `flake.lock` est automatiquement créé. Ce fichier va spécifier chaque dépendence au commit prêt. Contrairement aux channels de Nixpkgs qui sont spécifiés par "release" et non par commit.

Les flakes bénéficient aussi de très bons outils (flake-utils et les commandes nix). Les flake-utils donnent des outils puissant pour créer des flakes et les commandes nix sont des outils très puissant et simple pour *utiliser* ces flakes.

Ainsi par exemple :

- `nix build` va build la dérivation se trouvant dans l'output nomé `defaultPackage.<system>` ou `packages.<system>.default`
- `nix shell` va entrer dans un shell où le package/derivation dit "default" est accessible dans le `$PATH`
- `nix run` va lancer le binaire du package "default" qui correspond au nom de la dérivation
- `nix develop` qui va donner tous les outils de développement pour build le paquet et développer dans le `$PATH` (va d'abord chercher après `devShells.<system>.default` ou `devShell.<system>` avant de chercher pour le package default.
- `nix flake show` donne la liste de tous les outputs d'un flake
- `nix flake info` donne des métadonnées générales sur le flake
- `nix flake new` crée un nouveau flake
- `nix flake update` met à jour le fichier `nix.lock`
- `nix flake check` qui effectue les checks sur le flake
- `nix profile install` pour installer le defaultPackage du flake dans son profile
- `nix profile list` pour voir la liste des flakes installés

- `nix profile remove <id>` pour supprimer un flake de son profile

Je vais expliquer plus en détail comment ça fonctionne dans la partie pratique.

Il est aussi possible de créer des alias et d'utiliser `nixpkgs` via les flakes. Les alias peuvent être défini grâce au *flake registery*. Par exemple le flake `nixpkgs` est un alias pour `github:NixOS/nixpkgs/nixpkgs-unstable`. Et imaginons que je veut installer cowsay temporairement via un flake je peux simplement faire.

```
nix shell nixpkgs#cowsay
```