GNU LINUX MAGAZINE / FRANCE

DÉVELOPPEMENT SUR SYSTÈMES OPEN SOURCE

N°245

FÉVRIER 2021

FRANCE MÉTRO. : 8,90 €
BELUX : 9.90 € - CH : 15.10 CHF
ESP/IT/PORT-CONT : 9,90 €
DOM/S : 9,90 €
TUN : 25 TND - MAR : 100 MAD
CAN : 16,50 $CAD

L 19275 - 245 - F: 8,90 € - RD

Développement / Langage

NE RATEZ PAS LA RÉVOLUTION RUST !

p.44

SGBD / Actus

TIREZ PARTI DES NOUVEAUTÉS DE POSTGRESQL 13 p.06

Java / Conteneur

REDÉCOUVREZ QUARKUS, LE FRAMEWORK POUR LE CLOUD ET LES MICROSERVICES EN JAVA p.16

JavaScript / Syntax Highlighting

AMÉLIOREZ LA LISIBILITÉ DU CODE SUR UNE PAGE WEB AVEC TAMPERMONKEY p.74

Obfuscation / Acme::Buffy

ENCODAGE D'UN SCRIPT ET EXÉCUTION SANS DÉCODAGE EN PYTHON p.36

Kernel / C

COMPRENEZ LE MÉCANISME DE TRAITEMENT DES APPELS SYSTÈME DANS LE NOYAU p.22

MINISTÈRE
DES ARMÉES
Liberté
Égalité
Fraternité

Attrape-moi si tu peux !

À Bruz, on recrute dans les métiers de la détection d'intrusion et de la furtivité numérique.

crédit photo : Jean-Claude Moschetti

Bruz, c'est à 1h de Saint-Malo ! #CyberVallée

DGA Maîtrise de l'information
POSTULEZ en cybersécurité !

CONTACT Envoyez votre candidature

dga-mi-bruz.recrutement.fct@intradef.gouv.fr

DGA
DIRECTION GÉNÉRALE
DE L'ARMEMENT

GNU/Linux Magazine France
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND

10, Place de la Cathédrale - 68000 Colmar - France
Tél. : 03 67 10 00 20 – **Fax :** 03 67 10 00 21
E-mail : lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Résponsable service infographie : Kathrin Scali
Réalisation graphique : Thomas Pichon
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution, **N° ISSN :** 1291-78 34
Commission paritaire : K78 976

Périodicité : Mensuel
Prix de vente : 8,90 €

Membre April www.april.org

https://www.gnulinuxmag.com

RETROUVEZ-NOUS SUR :

@gnulinuxmagazine

@gnulinuxmag

p.29 DÉCOUVREZ TOUS NOS ABONNEMENTS PAPIER !

Codes sources sur :
https://github.com/glmf

ÉDITO

Comme chaque année, le classement **TIOBE [1]** a été publié et comme depuis plusieurs années c'est le langage **C** qui est le plus populaire. Néanmoins, **Python** a été élu langage de l'année 2020, car il s'agit du langage ayant connu la plus grosse progression au niveau de la popularité (+2,01 % cette année) ! Cela ne représente guère une surprise et je vous propose donc de soulever plutôt les autres faits marquants de ce classement annuel.

Pour rappel, TIOBE est un indicateur de la popularité des langages de programmation basé sur les requêtes effectuées sur les principaux moteurs de recherche (**Google**, **Bing**, **YouTube**, etc.). Les 100 langages les plus recherchés se voient attribuer ensuite des notes permettant de les classer et de dégager des tendances quant à leur popularité et leur utilisation.

Dans le classement de 2020, le **C** reste le langage le plus populaire.

Le langage **Java** est à -4,93% soit la plus grosse chute de l'année qui le relègue à la deuxième position du classement, derrière le C, mais encore devant Python.

Globalement, nous constatons donc logiquement peu de mouvement dans les langages les plus populaires. Ce qu'il est intéressant de noter ce sont les progressions, les langages qui prennent de l'importance pour différentes raisons comme **Groovy** (+1,23%, 10ème place) et le langage **R** (+1,10%, 9ème place). Mais dans ce classement, le **JavaScript** n'apparaît qu'en 7ème position, et l'**Assembleur** est en 11ème position, quant à **Rust**, bien que proche du top 20, il ne se retrouve qu'à la 26ème place… un peu étrange (il faut toujours se méfier de la manière dont sont effectués les calculs).

Si l'on observe un autre classement **[2]**, cette fois basé sur l'activité dans des dépôts **GitHub**, JavaScript occupe la première place et Rust se retrouve en 14ème position (devant R), premier des langages émergents. Il est donc difficile de se fier à un classement en particulier, mais si l'on recoupe plusieurs d'entre eux, la tendance générale est à un plébiscite de Python, une utilisation toujours importante de C/C++, Java, JavaScript et une utilisation croissante de Rust, R, Groovy, et Kotlin.

Ce mois-ci, nous vous proposons justement un article sur l'un de ces langages prometteurs : Rust ! Laissez-lui donc une chance et testez-le, il y a quelques années Python était à la même place...

Tristan Colombo

[1] Classement TIOBE 2021 : https://www.tiobe.com/tiobe-index/

[2] Un classement des langages sur GitHub : https://madnight.github.io/githut

SOMMAIRE

GNU/LINUX MAGAZINE FRANCE

N°245

ACTUS & HUMEUR

06 TIREZ PARTI DES NOUVEAUTÉS DE POSTGRESQL 13

Le 24 septembre 2020 est sortie la version 13 de PostgreSQL. Elle comprend de nombreuses nouvelles fonctionnalités. Certaines ont pour cible les utilisateurs et développeurs, d'autres sont pour les administrateurs...

IA, ROBOTIQUE & SCIENCE

16 QUARKUS DANS LES NUAGES

Lancé il y a moins de deux ans, le projet Open Source Quarkus vient déjà d'être consacré « produit » par la compagnie qui l'a initié, Red Hat. Qu'est-ce que cela signifie ?...

KERNEL & BAS NIVEAU

22 LE FONCTIONNEMENT DES NAMESPACES DANS LE NOYAU

Après la présentation des structures de données supportant les namespaces, ce nouvel opus se consacre à la partie immergée dans le noyau des appels système...

HACK & BIDOUILLE

36 ENCODAGE D'UN SCRIPT PYTHON ET EXÉCUTION DU SCRIPT ENCODÉ (ACME::BUFFY STYLE)

Il est parfois intéressant de considérer des problèmes anodins, complètement inutiles, mais qui permettent de mettre en œuvre des éléments de programmation encore jamais employés...

29 ABONNEMENTS PAPIER

LIBS & MODULES

44 RUST, LE LANGAGE INOXYDABLE !

Rust a fêté il y a quelques mois ses dix ans. Huit ans après sa première version alpha, cinq ans après sa première version stable, il était temps de voir ce qu'il était advenu de ce langage qui avait démarré quasiment dans un garage et qui était rapidement devenu l'un des langages les plus intéressants de la décennie...

62 GESTION DE PROJETS AVEC ERLANG/OTP

Un langage de programmation se doit d'être facile d'accès, que ce soit pour son apprentissage, la réalisation de concepts ou de produits finaux. La création de projets en Erlang se fait via les notions d'application et de release...

MOBILE & WEB

74 RENDRE UNE PAGE PRÉSENTANT DU CODE PLUS ERGONOMIQUE AVEC TAMPERMONKEY/GREASEMONKEY

Lire des articles contenant du code informatique sur le Web n'est pas nécessairement simple, ne serait-ce que de par la complexité inhérente au sujet traité...

Chez votre marchand de journaux

et sur **www.ed-diamond.com**

PAPIER
en kiosque

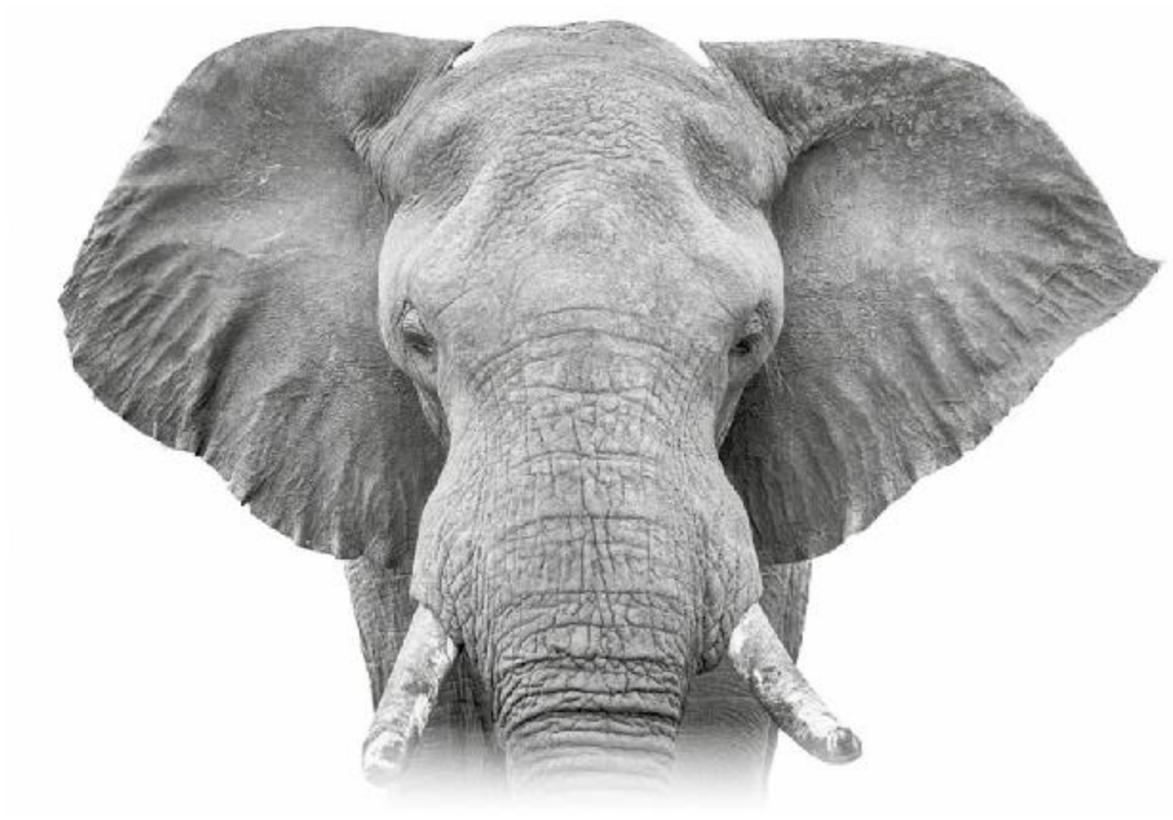
FLIPBOOK HTML5
sur www.ed-diamond.com

CONNECT
LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT
sur connect.ed-diamond.com

TIREZ PARTI DES NOUVEAUTÉS DE POSTGRESQL 13

GUILLAUME LELARGE
[Contributeur majeur de PostgreSQL, consultant Dalibo,
auteur du livre « PostgreSQL – Architecture et notions avancées »]

MOTS-CLÉS : POSTGRESQL, BACKUP, SUPERVISION, RÉPLICATION, MAINTENANCE



Le 24 septembre 2020 est sortie la version 13 de PostgreSQL. Elle comprend de nombreuses nouvelles fonctionnalités. Certaines ont pour cible les utilisateurs et développeurs, d'autres sont pour les administrateurs. La première version corrective de cette branche, la 13.1, est sortie le 12 novembre et cela nous semble une bonne occasion de regarder en profondeur certaines de ces nouveautés.

Clairement, cet article ne pourra pas être une liste exhaustive de toutes les nouvelles fonctionnalités. Ce qui suit compose une liste purement subjective, délibérément tournée vers l'administration du serveur (et non pas le développement d'applications avec PostgreSQL).

De tout ceci découle qu'il est évident que d'autres personnes pourraient à raison mettre en avant d'autres nouveautés. Donc, nous parlerons ici principalement d'administration, en découpant les fonctionnalités sur cinq thématiques : la maintenance, la sauvegarde, les performances, la supervision et la réplication.

1. MAINTENANCE

Parmi les opérations de maintenance, le `VACUUM` a une place prépondérante. Il dispose depuis plusieurs années d'un processus dédié d'automatisation. Ce processus appelé `autovacuum` est un processus en tâche de fond ayant pour but d'exécuter automatiquement des `VACUUM` et des `ANALYZE` sur les tables qui en auraient besoin. Pour le `VACUUM`, l'`autovacuum` fait attention au nombre de lignes mortes dans la table. Pour l'`ANALYZE`, il s'inquiète du nombre de lignes insérées, modifiées et supprimées. Tout ceci fonctionnait bien depuis l'intégration de l'`autovacuum` lors du développement de la version 8.1. Cependant, depuis la version 9.2, le `VACUUM` est aussi en charge de mettre à jour deux fichiers de métadonnées : les fichiers `FSM` (pour Free Space Map) et `VM` (pour Visibility Map) d'une table. Ces fichiers sont à mettre à jour après des insertions, des modifications et des suppressions de lignes. Or, l'`autovacuum` ne déclenche un `VACUUM` que si un certain taux de lignes mortes est dépassé, donc après un certain nombre de modifications et

de suppressions. Cela ne prend pas du tout en compte le nombre d'insertions. De ce fait, l'**autovacuum** ne lance que très peu de **VACUUM** sur les tables principalement en écriture (une table de log, par exemple). Cela veut dire que leurs fichiers de métadonnées ne sont pas mis à jour après des insertions, ce qui a des conséquences fortes, notamment sur l'optimiseur de requêtes qui ne peut plus utiliser correctement le fichier **VM** et sélectionner une opération comme **Index Only Scan**.

La version 13 corrige cela en forçant l'exécution d'un **VACUUM** si un certain nombre d'insertions a eu lieu. Ce nombre d'insertions dépend des paramètres **autovacuum_vacuum_insert_threshold** (nombre minimal de lignes insérées) et **autovacuum_vacuum_insert_scale_factor** (pourcentage du nombre de lignes insérées par rapport au nombre total de lignes dans la table). La formule de déclenchement est identique à ce qui se fait pour les deux autres modes de déclenchement, à savoir :

```
autovacuum_vacuum_
insert_threshold +
autovacuum_vacuum_
insert_scale_factor *
nombre lignes
```

Le nombre d'insertions depuis le dernier **VACUUM** sur la table est disponible dans la colonne **n_ins_since_vacuum** de la vue **pg_stat_all_tables**. Pour voir quand un **VACUUM** est sur le point d'être exécuté, il suffit d'utiliser la requête suivante :

```
SELECT c.relname,
       n_ins_since_
vacuum,
       current_
setting('autovacuum_
vacuum_insert_
threshold')::numeric
       +
       current_setting('autovacuum_vacuum_insert_scale_
factor')::numeric*reltuples
       AS declenchement,
      last_autovacuum,
       autovacuum_count
FROM   pg_stat_all_tables s
JOIN   pg_class c ON c.oid=s.relid
WHERE  c.relname='nom_table';
```

Avec l'exemple suivant :

```
CREATE TABLE t1 (c1 integer, c2 text);
INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1,
1001) i;
```

Voici le résultat de la requête de test après l'insertion :

```
                    Sun 15 Nov 2020 11:38:17 AM CET (every 1s)

relname  | n_ins_since_vacuum | declenchement | last_autovacuum | autovacuum_count
---------+--------------------+---------------+-----------------+-----------------
 t1      |               1001 |          1000 |                 |               0
(1 row)
[...]
                         Sun 15 Nov 2020 11:39:03 AM CET (every 1s)

relname  | n_ins_since_vacuum | declenchement |   last_autovacuum    | autovacuum_count
---------+--------------------+---------------+----------------------+-----------------
 t1      |                  0 |        1200.2 | 2020-11-15 11:39:03 |               1
(1 row)
```

À 11h38, le nombre d'insertions depuis le dernier **VACUUM** (colonne **n_ins_since_vacuum**) dépassait le seuil de déclenchement (colonne **declenchement**). Un peu après (11h39), lors de la prochaine vérification réalisée par l'**autovacuum**, ce dernier s'aperçoit du dépassement et exécute un **VACUUM**. La colonne **n_ins_since_vacuum** passe à **0**.

Pour en revenir à l'opération **VACUUM**, cette dernière a pour but de consigner l'ensemble des lignes mortes d'une table dans un fichier de métadonnées appelé la **FSM** (Free Space Map). Il en profite pour supprimer les enregistrements d'index pointant vers des lignes mortes. L'opération entière se fait en trois étapes : lecture de la table pour récupérer toutes les lignes mortes à nettoyer, parcours de chaque index pour supprimer les enregistrements d'index devenus inutiles, et enfin nettoyage de la table. Ces trois étapes sont réalisées plusieurs fois si le nombre de lignes mortes à nettoyer est très important. Les différentes étapes ne peuvent pas s'exécuter en parallèle les unes des autres. Par contre, lorsque la table dispose de plusieurs index, il peut être intéressant de traiter les index en parallèle. Ce n'était pas le cas auparavant, mais par défaut, la version 13 utilisera plusieurs processus pour traiter les index. Il faut néanmoins qu'il y ait au moins deux index et que les index soient plus volumineux que la configuration indiquée par le paramètre **min_parallel_index_scan_size**. Le niveau de parallélisation dépend du paramètre **max_parallel_maintenance_workers**,

mais ce dernier peut être surchargé avec la clause **PARALLEL** de la commande **VACUUM**. De plus, un index ne pourra être traité que par un seul processus, donc inutile d'indiquer un niveau de parallélisation supérieur au nombre d'index de la table à traiter (ou du nombre maximal d'index sur les tables de cette base pour un **VACUUM** global). Enfin, il est à noter que la parallélisation de traitement des index n'est disponible que pour le **VACUUM** simple, et donc pas pour le **VACUUM FULL**. L'utilitaire **vacuumdb** a lui aussi été modifié pour gérer la clause **PARALLEL** et dispose donc maintenant d'une option **--parallel**.

```
INSERT INTO t1 SELECT i, 'Ligne '||i FROM
generate_series(1002, 10000000) i;
CREATE INDEX ON t1(c1);
CREATE INDEX ON t1(c2);
DELETE FROM t1 WHERE c1<10000;
```

Après avoir fait la suppression de quelques lignes, il est possible d'exécuter un **VACUUM** de la table dans une session et d'exécuter plusieurs fois la requête suivante dans une autre session :

```
SELECT pid, backend_type, query
FROM pg_stat_activity
WHERE query LIKE '%VACUUM%'
  AND pid<>pg_backend_pid();
```

Cette requête récupère tous les processus exécutant une commande contenant le texte « VACUUM ». Il sera peut-être nécessaire de l'exécuter plusieurs fois (ce qui se fait automatiquement sous **psql** avec la métacommande **\watch 1** après une première exécution). Vous devriez récupérer un résultat identique à celui-ci :

```
   pid    |  backend_type    |   query
---------+------------------+-----------
 1246892 | client backend   | VACUUM t1;
 1249729 | parallel worker | VACUUM t1;
(2 rows)
```

Il y a deux index éligibles au traitement parallélisé, donc deux processus traitent les index : le leader (PID 1246892) et le parallel worker (PID 1249729).

Dans la même idée, l'utilitaire **reindexdb** se voit ajouter une option de parallélisation. La nouvelle option **--jobs** permet de créer plusieurs connexions au serveur, chaque connexion étant chargée d'exécuter la réindexation d'un index. Cependant, il convient de faire attention à la charge supplémentaire que cela imposera aux CPU du serveur.

Maintenant, voici une nouveauté qui va faciliter la vie à certains DBA. Lors de la suppression d'une base, il était fréquent d'avoir une erreur indiquant que la base étant utilisée, elle ne pouvait pas être supprimée. Ceci est toujours vrai, il n'est toujours pas possible de supprimer une base si des utilisateurs sont connectés dessus. Le contournement généralement utilisé dans ce cas revient à déconnecter de force les utilisateurs, par exemple avec la requête ci-dessous :

```
SELECT pg_terminate_backend(pid)
FROM   pg_stat_activity
WHERE  datname='base_à_supprimer';
```

Avec la version 13, une alternative existe permettant d'oublier cette requête. La clause **FORCE** a été ajoutée à la commande SQL **DROP DATABASE**. En utilisant cette clause, les utilisateurs seront automatiquement déconnectés pour que la suppression puisse se faire. L'utilitaire **dropdb** a été modifié pour bénéficier de cette nouvelle possibilité en passant par l'option en ligne de commande **--force**.

Depuis leur apparition en version 9.1, les extensions ont peu progressé. Il y a néanmoins un souci qui devient de plus en plus problématique, surtout avec les bases hébergées en mode cloud. Une extension ne peut être installée que par un utilisateur ayant l'attribut **SUPERUSER**. Or, dans certaines architectures (par exemple les bases dans le cloud), le client n'a pas accès à un utilisateur avec ce niveau de confiance. Il a généralement un simple utilisateur, propriétaire de sa base, mais cela ne suffit pas pour ajouter une extension à cette base. Arrive en version 13 la notion d'extension de confiance. Une extension de confiance peut être installée par tout utilisateur qui a le droit **CREATE** sur la base (pour ajouter l'extension) et qui a le droit **CREATE** sur le schéma d'installation de l'extension (pour que l'extension puisse y installer ses objets).

Le caractère de confiance d'une extension est disponible dans la vue **pg_available_extension_versions**, avec la colonne **trusted**.

Voici un exemple où l'utilisateur **u1** va chercher à créer deux extensions, une de confiance, l'autre non.

```
# CREATE USER u1;
# GRANT CREATE ON DATABASE articlev13 TO u1;
# \c articlev13 u1
$ SELECT name, version, trusted
  FROM pg_available_extension_versions
```

```
 WHERE name='pgcrypto'
 ORDER BY version DESC
 LIMIT 1;
  name    | version | trusted
----------+---------+---------
 pgcrypto | 1.3     | t
(1 row)
-- l'extension pgcrypto est de confiance
(colonne trusted à true)
-- un utilisateur simple doit pouvoir
l'ajouter
$ CREATE EXTENSION pgcrypto;
$ SELECT name, version, trusted
  FROM pg_available_extension_versions
  WHERE name='pg_buffercache'
  ORDER BY version DESC
  LIMIT 1;
      name       | version | trusted
-----------------+---------+---------
 pg_buffercache | 1.3     | f
(1 row)
-- l'extension pg_buffercache n'est pas de
confiance (colonne trusted à false)
-- un utilisateur simple ne doit pas pouvoir
l'ajouter
$ CREATE EXTENSION pg_buffercache;
ERROR:  permission denied to create
extension "pg_buffercache"
HINT:  Must be superuser to create this
extension.
```

2. SAUVEGARDE

La sauvegarde de base effectuée notamment par **pg_basebackup** apporte une fonctionnalité très intéressante : la création d'un manifeste de sauvegarde. Ce manifeste est un fichier contenant la liste des fichiers sauvegardés et quelques métadonnées, dont notamment une somme de contrôle. Le but de ce fichier est de permettre de vérifier l'état d'une sauvegarde. Cela ne protège pas contre tout, mais apporte une sécurité supplémentaire. La création de ce fichier est activée par défaut, donc aucun besoin d'ajouter une option pour en profiter.

Faisons une sauvegarde du serveur avec **pg_basebackup** :

```
$ mkdir sauvegarde
$ pg_basebackup --pgdata sauvegarde
--checkpoint fast
```

Une fois la sauvegarde terminée, regardons le contenu du répertoire de sauvegarde :

```
$ ll sauvegarde/
total 296
-rw-------. 1 guillaume guillaume    226 Nov 15 14:22
backup_label
-rw-------. 1 guillaume guillaume 178333 Nov 15 14:22
backup_manifest
drwx------. 6 guillaume guillaume   4096 Nov 15 14:22
base
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
global
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_commit_ts
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_dynshmem
-rw-------. 1 guillaume guillaume   4760 Nov 15 14:22
pg_hba.conf
-rw-------. 1 guillaume guillaume   1636 Nov 15 14:22
pg_ident.conf
drwx------. 4 guillaume guillaume   4096 Nov 15 14:22
pg_logical
drwx------. 4 guillaume guillaume   4096 Nov 15 14:22
pg_multixact
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_notify
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_replslot
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_serial
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_snapshots
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_stat
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_stat_tmp
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_subtrans
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_tblspc
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_twophase
-rw-------. 1 guillaume guillaume      3 Nov 15 14:22
PG_VERSION
drwx------. 3 guillaume guillaume   4096 Nov 15 14:22
pg_wal
drwx------. 2 guillaume guillaume   4096 Nov 15 14:22
pg_xact
-rw-------. 1 guillaume guillaume     88 Nov 15 14:22
postgresql.auto.conf
-rw-------. 1 guillaume guillaume  27930 Nov 15 14:22
postgresql.conf
```

Nous trouvons ici tous les fichiers habituels, plus un nouveau fichier, appelé **backup_manifest**. Ce fichier contient donc la liste des fichiers sauvegardés. Il est au format JSON. Voici un extrait de son contenu :

```
$ head sauvegarde/backup_manifest
{ "PostgreSQL-Backup-Manifest-Version": 1,
"Files": [
{ "Path": "backup_label", "Size": 226, "Last-Modified":
"2020-11-15 13:22:35 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "dd4a1d8f" },
{ "Path": "postgresql.conf", "Size": 27930, "Last-
Modified": "2020-11-15 10:33:39 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "7f7c44fd" },
{ "Path": "PG_VERSION", "Size": 3, "Last-Modified":
"2020-11-15 10:33:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "b825884b" },
{ "Path": "pg_ident.conf", "Size": 1636, "Last-
Modified": "2020-11-15 10:33:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "4a7b7b6f" },
{ "Path": "pg_logical/replorigin_checkpoint", "Size": 8,
"Last-Modified": "2020-11-15 13:22:35 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "c74b6748" },
{ "Path": "postgresql.auto.conf", "Size": 88, "Last-
Modified": "2020-11-15 10:33:12 GMT", "Checksum-
Algorithm": "CRC32C", "Checksum": "536f950b" },
{ "Path": "pg_xact/0000", "Size": 8192, "Last-Modified":
"2020-11-15 10:58:16 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "e0adee9a" },
{ "Path": "base/1/4153", "Size": 0, "Last-Modified":
"2020-11-15 10:33:12 GMT", "Checksum-Algorithm":
"CRC32C", "Checksum": "00000000" },
```

Donc, pour un fichier, on dispose du chemin vers le fichier (clé **Path**), sa taille (**Size**), sa date de dernière modification (**Last-Modified**), l'algorithme utilisé pour calculer la somme de contrôle (**Checksum-Algorithm**, modifiable avec l'option –**manifest-checksums** de **pg_basebackup**), et enfin sa somme de contrôle (**Checksum**).

Il est possible de vérifier la sauvegarde avec un nouvel outil appelé **pg_verifybackup** :

```
$ pg_verifybackup sauvegarde/
backup successfully verified
```

Prenons la table **t1**. Récupérer le nom du fichier de cette table est aussi simple qu'exécuter la requête suivante :

```
SELECT pg_relation_filepath('t1');
```

Ce qui nous donne le fichier **base/16384/16385**.

Supposons maintenant que le contenu du fichier de la sauvegarde est modifié (**hexedit** est parfait pour ça). Il ne nous reste plus qu'à voir la réaction de **pg_verifybackup** :

```
$ pg_verifybackup sauvegarde/
pg_verifybackup: error: checksum
mismatch for file "base/16384/16385"
```

Dans les autres changements concernant **pg_basebackup**, ce dernier estime par défaut la taille totale de la sauvegarde. Ceci est important pour suivre l'évolution de la sauvegarde au travers de la nouvelle vue de progression, **pg_stat_progress_basebackup**, dont nous parlerons plus tard. Il est à noter que cela constitue un changement de comportement qui peut se révéler néfaste, si ce calcul de taille prend beaucoup de temps. Il est de ce fait possible de revenir à l'ancien comportement en utilisant l'option **--no-estimate-size**.

Pour en terminer avec les changements sur la sauvegarde et ses outils, il est à noter l'apparition d'une nouvelle option de **pg_dump** : **--include-foreign-data**. Cette option a pour but de sauvegarder les données des tables externes. À la restauration, les données seront renvoyées vers le serveur distant, en supposant que le pilote de ce serveur accepte les écritures distantes. L'utilisation de cette option devra donc être mûrement réfléchie.

3. PERFORMANCE

PostgreSQL sait utiliser un index pour trier les données. Cependant, dans certains cas, il ne sait pas utiliser l'index alors qu'il pourrait le faire. Prenons un exemple.

```
CREATE TABLE t2 (c1 integer, c2
integer, c3 boolean);
INSERT INTO t2
  SELECT i,
         i%3,
          i<5000000
  FROM generate_series(1, 10000000) i;
CREATE INDEX ON t2(c1);
ANALYZE t2;
```

PostgreSQL sait utiliser un index pour trier les données. Voici le plan d'exécution pour un tri sur la colonne **c1** :

```
EXPLAIN (ANALYZE, COSTS OFF) SELECT * FROM
t2 ORDER BY c1;
                QUERY PLAN
-----------------------------------------
 Index Scan using t2_c1_idx on t2 (actual
time=0.144..1490.351 rows=10000000 loops=1)
 Planning Time: 1.131 ms
 Execution Time: 1766.678 ms
(3 rows)
```

En revanche, si le tri concerne les colonnes **c1** et **c2**, les versions 12 et antérieures ne savent pas utiliser l'index, comme le montre ce plan d'exécution :

```
EXPLAIN (ANALYZE, COSTS OFF) SELECT *
FROM t2 ORDER BY c1, c2;
                QUERY PLAN
-----------------------------------------
 Gather Merge (actual
time=1390.361..3563.469 rows=10000000
loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Sort (actual
time=1376.274..1855.708 rows=3333333
loops=3)
         Sort Key: c1, c2
         Sort Method: external merge
Disk: 54696kB
         Worker 0:  Sort Method:
external merge  Disk: 77584kB
         Worker 1:  Sort Method:
external merge  Disk: 53928kB
         ->  Parallel Seq Scan on
t2 (actual time=0.051..391.157
rows=3333333 loops=3)
 Planning Time: 0.062 ms
 Execution Time: 3982.706 ms
(11 rows)
```

Comme PostgreSQL ne sait pas utiliser un index pour réaliser ce tri, il passe par un parcours de table (parallélisé dans le cas présent), puis effectue le tri, ce qui prend beaucoup de temps. La requête a plus que doublé en durée d'exécution.

La version 13 est beaucoup plus maline à cet égard. Elle est capable d'utiliser l'index pour faire un premier tri des données (sur la colonne **c1** d'après notre exemple), puis elle complète le tri par rapport à la colonne **c2** :

```
          QUERY PLAN
-----------------------------
 Incremental Sort (actual
time=0.205..2581.413 rows=10000000 loops=1)
   Sort Key: c1, c2
   Presorted Key: c1
   Full-sort Groups: 312500  Sort Method:
quicksort
     Average Memory: 26kB  Peak Memory:
26kB
   ->  Index Scan using t2_c1_idx on t2
       (actual time=0.133..1519.481
rows=10000000 loops=1)
 Planning Time: 0.406 ms
 Execution Time: 2860.087 ms
(7 rows)
```

La requête en version 12 prenait 3,9 secondes en parallélisant sur trois processus. La version 13 ne prend que 2,8 secondes, sans parallélisation. On remarque un nouveau type de nœud, le **Incremental Sort**, qui s'occupe de retrier les données après un renvoi de données partiellement triées, grâce au parcours d'index.

L'apport en performance est déjà très intéressant, d'autant qu'il réduit à la fois le temps d'exécution de la requête, mais aussi la charge induite sur l'ensemble du système. Cet apport en performance devient remarquable si on utilise une clause **LIMIT**. Voici le résultat en version 12 :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM t1 ORDER BY c2, c3 LIMIT 10;
                QUERY PLAN
-------------------------------------------
 Limit (actual time=720.030..723.946 rows=10 loops=1)
   ->  Gather Merge (actual time=720.028..723.943 rows=10 loops=1)
         Workers Planned: 2
         Workers Launched: 2
        ->  Sort (actual time=705.577..705.578 rows=10 loops=3)
              Sort Key: c1, c2
              Sort Method: top-N heapsort  Memory: 25kB
              Worker 0:  Sort Method: top-N heapsort  Memory: 25kB
              Worker 1:  Sort Method: top-N heapsort  Memory: 25kB
              ->  Parallel Seq Scan on t2
                  (actual time=0.082..386.972 rows=3333333 loops=3)
 Planning Time: 0.196 ms
 Execution Time: 724.134 ms
(12 rows)
```

Et celui en version 13 :

```
                     QUERY PLAN
-----------------------------------------------------
 Limit (actual time=0.088..0.092 rows=10 loops=1)
   ->  Incremental Sort (actual time=0.087..0.089 rows=10 loops=1)
         Sort Key: c1, c2
         Presorted Key: c1
         Full-sort Groups: 1  Sort Method: quicksort
           Average Memory: 25kB  Peak Memory: 25kB
         ->  Index Scan using t2_c1_idx on t2
             (actual time=0.074..0.077 rows=11 loops=1)
 Planning Time: 0.120 ms
 Execution Time: 0.108 ms
(8 rows)
```

La requête passe donc de 724 ms avec parallélisation, à 0,108 ms sans parallélisation.

Les index ont bénéficié eux aussi d'améliorations. En voici un exemple très édifiant. Un index Btree contient pour chaque valeur de la table la valeur et un pointeur vers la ligne de la table contenant cette valeur. Donc si une table contient 10 millions de lignes, l'index contiendra pour la colonne indexée les 10 millions de valeurs ainsi que les 10 millions de pointeurs. C'est simple et efficace. Cependant, ça prend aussi beaucoup de place. Si une valeur est fréquente, si elle apparaît de nombreuses fois dans la table, conserver des duplicatas de cette valeur n'a pas vraiment de sens. Il serait bien de ne garder qu'une seule fois la même valeur et la liste des pointeurs pour cette valeur. C'est déjà ce que font entre autres les index GIN, mais les index Btree n'ont jamais bénéficié de cette optimisation. Enfin, nous devrions plutôt dire « pas encore », vu que la version 13 implémente cela. C'est ce que les développeurs ont appelé de la dé-duplication. Prenons un exemple avec la table **t2** en lui ajoutant deux nouveaux index :

```
CREATE INDEX ON t2(c2);
CREATE INDEX ON t2(c3);
```

Pour rappel, la colonne **c1** contient uniquement des valeurs distinctes, la colonne **c2** contient trois valeurs (**0**, **1** et **2**), également réparties, et la colonne c3 est un booléen contenant en gros 50 % de valeurs **true** et 50 % de valeurs **false**. Avec la requête suivante, il est possible d'avoir la taille de la table et des trois index :

```
SELECT 't2' AS relname,
       pg_size_pretty(pg_
table_size('t2')) AS relsize
UNION
SELECT ci.relname,
       pg_size_pretty(pg_
table_size(indexrelid))
FROM pg_index i
JOIN pg_class ct ON
ct.oid=i.indrelid
JOIN pg_class ci ON
ci.oid=i.indexrelid
WHERE ct.relname='t2'
ORDER BY 1;
```

Et voici le résultant, pour une version 12 et pour une version 13 :

Table ou Index	Version 12	Version 13
t2	422 Mio	422 Mio
t2_c1_idx	214 Mio	214 Mio
t2_c2_idx	215 Mio	66 Mio
t2_c3_idx	215 Mio	66 Mio

Nous voyons bien que la taille du premier index n'a pas bougé. En effet, il indexe une colonne ne contenant que des valeurs distinctes, l'optimisation ne peut pas fonctionner ici. Le deuxième et le troisième index concernent des colonnes avec beaucoup de valeurs communes. La dé-duplication entre ici en œuvre et permet un gain en espace disque très important. Un gain en espace disque, pour une base de données, se traduit immédiatement en un gain en performances. C'est donc une optimisation très valable pour PostgreSQL. Cette déduplication ne fonctionne que pour certains types de données. Elle ne fonctionne pas notamment pour les valeurs à virgule flottante (type **float4** et **float8**) ainsi que pour les valeurs numériques de précision (type **numeric**).

Enfin, il est à noter que l'utilisation de **pg_upgrade** ne permet pas de bénéficier immédiatement de cette optimisation, les index n'étant pas réécrits pendant la phase de réindexation. Il conviendra donc de réindexer les index intéressants.

Les index Btree n'ont pas été les seuls à bénéficier d'optimisations en tout genre. Les index GiST, SP-GiST et GIN ont tous eu droit à des améliorations lors du développement de la version 13.

Dans la gestion des performances d'une requête, il y a une partie importante, mais pour laquelle on a peu d'informations : l'utilisation des journaux de transactions. Par exemple, lors de l'exécution d'un **INSERT**, combien d'octets ont été écrits dans les journaux de transactions ? Ce type d'information n'est réellement disponible nulle part. Cette nouvelle version apporte un début de réponse.

La commande **EXPLAIN** fournit cette information si la clause **WAL** est ajoutée :

```
Insert on t1  (cost=0.00..17500.00
rows=1000000 width=36)
        (actual time=10176.211..10176.212
rows=0 loops=1)
  WAL: records=3056552 fpi=2181
bytes=237613978
  ->  Function Scan on generate_series i
      (cost=0.00..17500.00 rows=1000000
width=36)
      (actual time=134.410..488.872
rows=1000000 loops=1)
Planning Time: 0.190 ms
Execution Time: 10179.684 ms
```

Trois informations sont fournies : le nombre d'enregistrements avec le champ **records**, le nombre de blocs complets de table avec le champ **fpi** (pour Full Page Image) et le nombre d'octets avec le champ **bytes**.

L'extension **auto_explain** a été mise à jour pour tirer profit de cette option supplémentaire. Il faut pour cela configurer le paramètre **auto_explain.log_wal** à **true**.

L'**autovacuum** trace aussi ce type d'informations quand le paramètre **log_autovacuum_min_duration** est activé :

```
2020-11-14 16:50:41.193 CET [1135327] LOG:
automatic vacuum of table "articlev13.public.
t2": index scans: 0
    pages: 0 removed, 108109 remain, 0
skipped due to pins, 0 skipped frozen
   tuples: 0 removed, 20000000 remain, 0 are
dead but not yet removable, oldest xmin: 528
    buffer usage: 70366 hits, 125392 misses,
51246 dirtied
    avg read rate: 22.493 MB/s, avg write
rate: 9.192 MB/s
    system usage: CPU: user: 3.54 s, system:
1.65 s, elapsed: 43.55 s
    WAL usage: 105290 records, 51232 full
page images, 423954613 bytes
```

Enfin, l'extension **pg_stat_statements** a été mise à jour pour tracer l'activité des journaux de transactions :

```
SELECT query, calls, rows, wal_records, wal_
fpi, wal_bytes
FROM pg_stat_statements
WHERE query like 'insert into t2%';
-[ RECORD 1 ]-----------------------
query         | insert into t2 [...]
calls         | 1
rows          | 10000000
wal_records   | 40293339
wal_fpi       | 27503
wal_bytes     | 2902798552
```

La clause **BUFFERS** de la commande **EXPLAIN** se voit ajouter une sortie supplémentaire : elle indique le nombre de blocs utilisés pour la planification de la requête. Auparavant, elle ne le faisait que pour l'exécution.

4. SUPERVISION

PostgreSQL a intégré le parallélisme d'exécution d'une requête avec la version 9.6. Depuis, de nombreuses améliorations ont lieu pour que de plus en plus de nœuds d'exécution soient parallélisables. Du côté de la supervision de la parallélisation, il y avait encore peu de choses. Il était possible de trouver les processus d'aide dans la vue **pg_stat_activity**, mais il était assez complexe de savoir quel processus ces processus aidaient. La version 13 améliore cela en ajoutant la colonne **leader_pid** à la vue **pg_stat_activity**. Cette colonne indique le PID du processus leader.

Pour la supervision automatique, cela permet de différencier les processus standards (**leader**) des processus d'aide (**worker**). On peut très bien imaginer aussi une requête de comptage des processus workers et de comparaison avec le nombre maximum de processus workers :

```
SELECT current_setting('max_parallel_
workers') AS max_parallel_workers,
       count(*) AS current_parallel_
workers
FROM pg_stat_activity
WHERE leader_pid IS NOT NULL;

 max_parallel_workers | current_parallel_
workers
----------------------+-----------------
 8                    |                 2
(1 row)
```

Si jamais la valeur de la colonne **current_parallel_workers** reste proche de, voire pire, égale à la valeur de la colonne **max_parallel_workers**, il sera certainement bon de revoir la configuration pour augmenter la valeur du paramètre **max_parallel_workers**.

On peut aussi concevoir une requête qui retourne tous les processus qui sont en train d'exécuter une requête, et de placer les processus d'aide sous les processus leaders. Prenons l'exemple de cette requête :

```
WITH RECURSIVE backends(level, leader_pid, pid, backend_
type, query) AS
(
        SELECT 0, pid, pid, backend_type, substr(query,
1, 20)
         FROM pg_stat_activity
         WHERE leader_pid IS NULL
           AND state = 'active'
       UNION ALL
         SELECT 1, s.leader_pid, s.pid,
                concat('↳ ', s.backend_type),
                ''
         FROM pg_stat_activity s, backends b
         WHERE s.leader_pid=b.pid
)
SELECT leader_pid, pid, backend_type, query FROM
backends ORDER BY 2, 1;
```

Elle pourrait renvoyer un résultat de ce type :

```
 leader_pid |   pid   |   backend_type    |     query
------------+---------+-------------------+----------------
    1043875 | 1043875 | client backend    | explain(analyze) sel
    1043875 | 1107811 | ↳ parallel worker |
    1043875 | 1107810 | ↳ parallel worker |
    1097485 | 1097485 | client backend    | WITH RECURSIVE backe
    1105623 | 1105623 | client backend    | VACUUM t1;
    1105623 | 1107812 | ↳ parallel worker |
```

Le paramètre **log_min_duration_statement** permet de tracer toutes les requêtes dont la durée d'exécution dépasse la valeur de ce paramètre. Lors d'un audit, ce paramètre est souvent mis à **0**, mais cette valeur n'est pas conservée longtemps, car sur les systèmes très utilisés, cela implique de fortes écritures dans les traces, pouvant ralentir le système et qui vont à coup sûr prendre beaucoup de place sur le disque. Dans le fonctionnement normal, ce paramètre est configuré à une valeur assez haute pour éviter les ralentissements et suffisamment basse pour tracer quand même quelques requêtes. Généralement, cette configuration ne permet pas d'avoir une bonne idée de ce qui est exécuté sur le serveur. Arrivent la version 13 et son nouveau système de traçage des requêtes exécutées. Il est maintenant possible de tracer juste un échantillon des

requêtes qui dépassent une certaine durée d'exécution. La durée limite se configure avec le paramètre **log_min_duration_sample** et le taux de requêtes tracées dépend de deux paramètres : **log_statement_sample_rate** pour un taux par requêtes et **log_transaction_sample_rate** pour un taux par transactions. Ainsi, on peut avoir une meilleure idée des différents types de requêtes exécutées sans écraser son système disque avec l'écriture des traces.

L'extension **pg_stat_statements** tracera en plus le temps de planification des requêtes avec cinq nouvelles colonnes : **total_plan_time**, **min_plan_time**, **max_plan_time**, **mean_plan_time** et **stddev_plan_time**. Il est ainsi possible de récupérer les requêtes avec leur durée totale de planification et d'exécution :

```
SELECT query, total_plan_time, total_exec_time, calls
FROM pg_stat_statements
WHERE query like '%t1%';

           query              | total_plan_time |   total_exec_time    | calls
------------------------------+-----------------+----------------------+-------
select * from t1 where c1 >$1 |               0 |             965.9638 |     1
select * from t1 where c1 =$1 |               0 | 0.12426000000000001 |     2
```

Les versions précédentes avaient intégré quelques vues de progression de commandes. C'est aussi le cas de la version 13, qui ajoute deux vues :

- **pg_stat_progress_basebackup**, pour suivre la progression d'une sauvegarde passant par le protocole de réplication (donc **pg_basebackup**, mais aussi **pgbackrest**, par exemple) ;
- **pg_stat_progress_analyze**, pour suivre la progression de la commande de maintenance **ANALYZE**.

Voici un exemple de lecture de la première vue lors de la sauvegarde avec **pg_basebackup** :

```
-[ RECORD 1 ]--------+-------------------------
pid                 | 1290288
phase               | streaming database files
backup_total        | 1095626240
backup_streamed     | 89233408
tablespaces_total   | 1
tablespaces_streamed | 0
```

Nous voyons ainsi à quelle phase se trouve l'opération de sauvegarde, et combien d'octets ont été envoyés et doivent être envoyés.

5. RÉPLICATION

La réplication a eu droit aussi à son lot de changements, mais ils restent assez limités. Néanmoins, PostgreSQL peut enfin recharger la configuration spécifique aux serveurs secondaires, sans avoir à couper toutes les connexions.

Parmi les autres modifications, il faut quand même noter la possibilité de limiter la volumétrie maximale de journaux de transactions à conserver pour un slot de réplication. Cette volumétrie se configure avec le paramètre **max_slot_wal_keep_size** et permettra de choisir exactement la place disque que peuvent prendre les journaux à conserver pour les slots de réplication. Cela a évidemment un inconvénient : en cas de dépassement de cette volumétrie, les anciens journaux, toujours indispensables aux slots de réplication, sont recyclés. Autrement dit, le serveur secondaire utilisant un slot de réplication risque de ne plus pouvoir être synchronisé avec les derniers changements survenus sur le serveur primaire. Il va donc falloir trouver la bonne balance entre éviter un manque d'espace disque (qui ferait tomber le serveur primaire) et éviter de désynchroniser un secondaire.

Il est à noter que le paramètre **wal_keep_segments** a été renommé en **wal_keep_size**. Il s'agit donc maintenant d'une taille et non d'un nombre de segments. Si ce paramètre était configuré sur une version antérieure, il convient de multiplier sa valeur par 16 Mio, la taille (par défaut) d'un segment de journal de transactions.

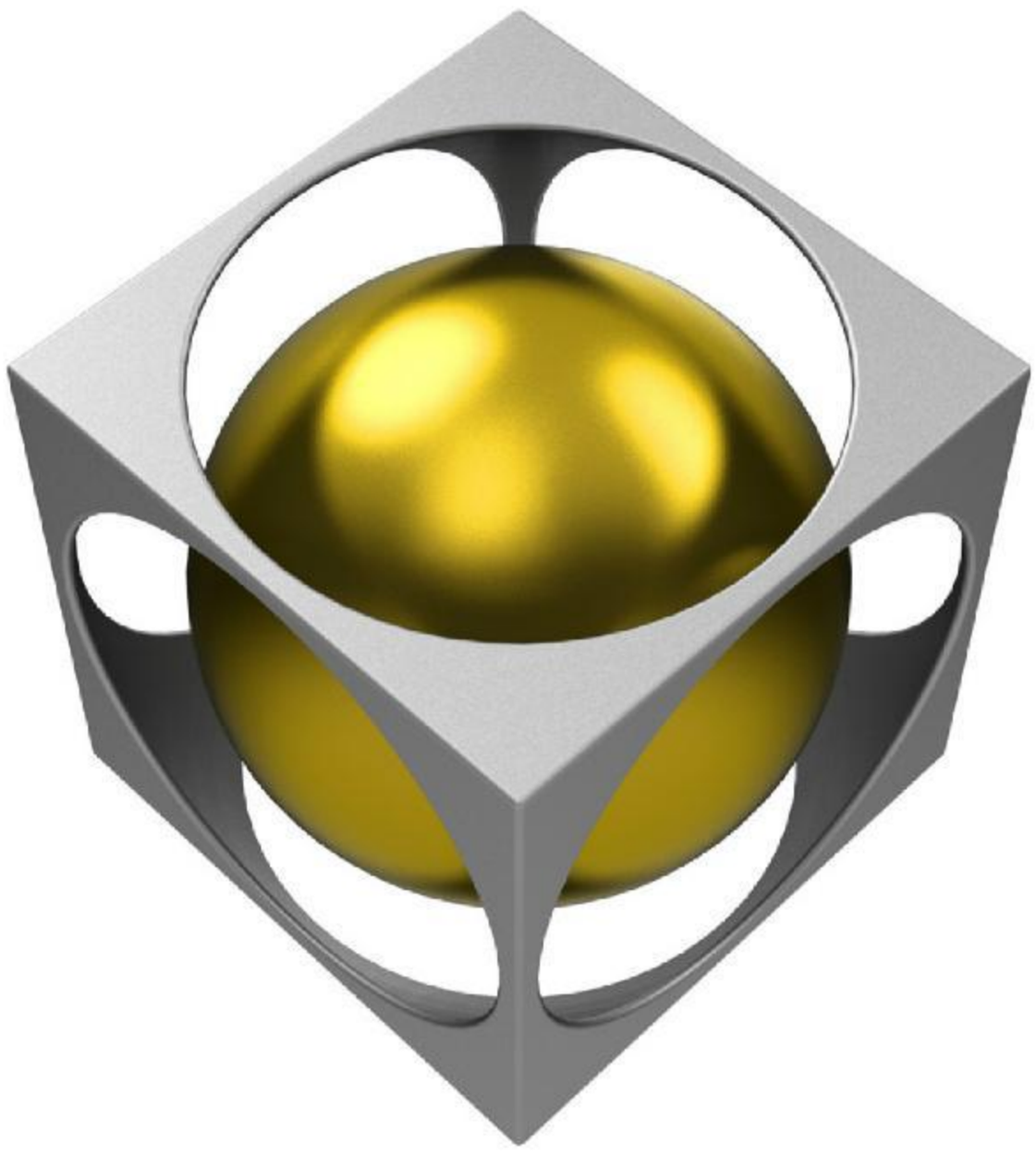
CONCLUSION

Comme dit au tout début de cet article, ce ne sont que quelques-unes des nouveautés proposées par la version 13 de PostgreSQL. Pour les curieux, une liste complète se trouve dans les Release Notes de la version. ■

QUARKUS
DANS LES NUAGES

ROMAIN PELISSE
[Sustain Developer chez Red Hat]

MOTS-CLÉS : JAVA, JEE, QUARKUS, CODE NATIF, DOCKER, REST



Lancé il y a moins de deux ans, le projet Open Source Quarkus vient déjà d'être consacré « produit » par la compagnie qui l'a initié, Red Hat. Qu'est-ce que cela signifie ? Simplement que, désormais, en plus d'être Open Source, le projet dispose d'une version entièrement certifiée et supportée par le chapeau rouge. Mais c'est aussi un indicateur que Quarkus a le vent en poupe. Bref, tout ceci forme une excellente opportunité de revenir sur cet excitant cadre de développement Java !

Le projet **Quarkus** est un nouveau cadre de développement Java conçu et pensé avant tout pour le Cloud et les microservices. Particularité notable, le framework propose les mêmes API (ou des API très similaires) que le monde **Jakarta EE** (soit les API disponibles au sein d'un serveur d'application JEE tel que **WildFly**). L'objectif avoué étant de permettre aux développeurs habitués à utiliser ces API d'être rapidement et facilement productifs sur ce nouvel environnement, tout en bénéficiant de tout le confort et la légèreté d'un framework dédié à la conception de microservices.

Pour atteindre tous ces objectifs, mais aussi produire des applications Java ultralégères, les développeurs du projet Quarkus ont fait des choix de conception très réfléchis. Ainsi, Quarkus **[1]** encadre beaucoup plus le développement que ne le fait un serveur d'applications. Si ce type de serveur offre une grande liberté sur la nature même de l'application déployée, avec Quarkus, on ne parle que de microservice REST **[2]** (ou éventuellement d'outil en ligne de commande, mais ce n'est pas un cas d'utilisation central à la proposition du framework).

Le projet se concentre sur les briques nécessaires à la réalisation de tels services et il est donc loin, fort heureusement, de supporter toutes les fonctionnalités d'un serveur conforme à la spécification Jakarta EE.

Une autre différence cruciale est qu'une fois l'application Quarkus réalisée, elle se déploie sous forme d'application « sans serveur ». Il n'y a donc besoin d'aucun serveur à faire fonctionner. La construction achevée, l'application est disponible sous la forme d'une simple archive **Java** (JAR) qui embarque l'ensemble des dépendances nécessaires à son exécution. Seule une machine virtuelle Java est requise pour démarrer le service !

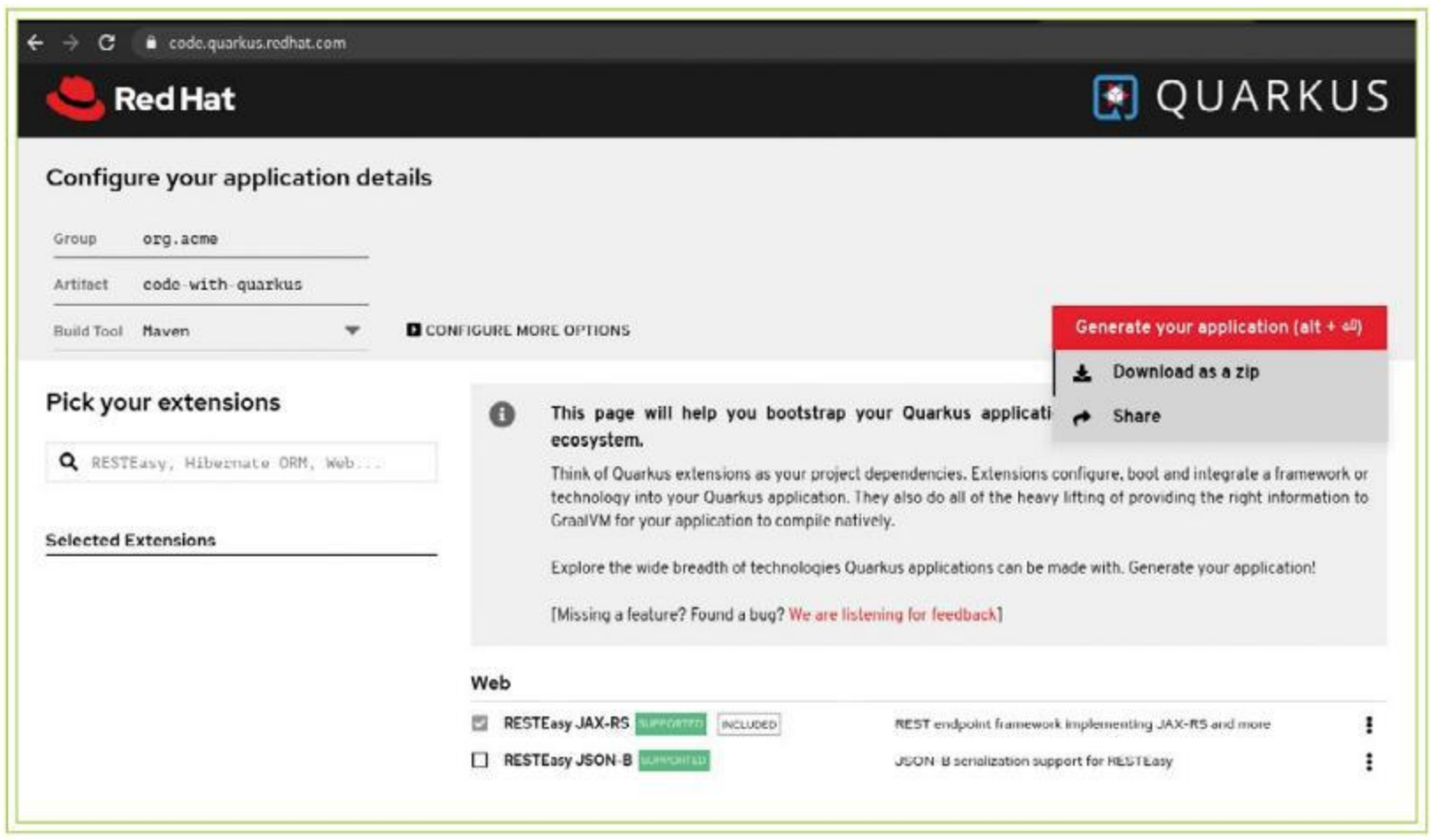


Fig. 1 : Génération du projet à partir du site de Red Hat.

Le projet a aussi pris le soin de réduire ses dépendances au strict minimum afin d'assurer que le JAR généré soit le plus léger possible. C'est un point essentiel pour la stratégie Cloud de Quarkus. En effet, la taille des applications Java, souvent très gourmandes en espace disque avec l'accumulation de dépendances en tout genre, pose beaucoup de difficultés et freine leur déploiement sous forme de conteneur du type de Docker, sur lesquels l'infrastructure de la plupart des Clouds repose.

1. UN MICROSERVICE QUARKUS EN JUSTE QUELQUES SECONDES

Sans revenir dans le détail sur la mise en place d'un projet Quarkus, déjà évoqué dans plusieurs précédentes publications **[5, 6, 7, 8]**, il est important de souligner, encore une fois, l'incroyable facilité de mise en place d'un microservice Java en seulement quelques instants à l'aide du framework. Et l'on ne parle pas ici seulement de la vitesse de génération du projet, mais aussi de la vitesse de démarrage du microservice en lui-même! Celle-ci étant historiquement rarement le point fort des solutions Java, c'est un point important à retenir.

Si dans une précédente publication **[5]**, nous avions illustré la mise en place d'un projet Quarkus directement à l'aide de l'outil de build **Maven [3]**, ici, comme le montre la figure 1, nous allons utiliser le générateur d'application en ligne mise à disposition par **Red Hat** (là encore, afin d'utiliser la version supportée par l'éditeur logiciel).

Une fois le projet téléchargé et décompressé, il suffit d'exécuter la commande suivante, comme indiqué sur le site (voir figure 2).

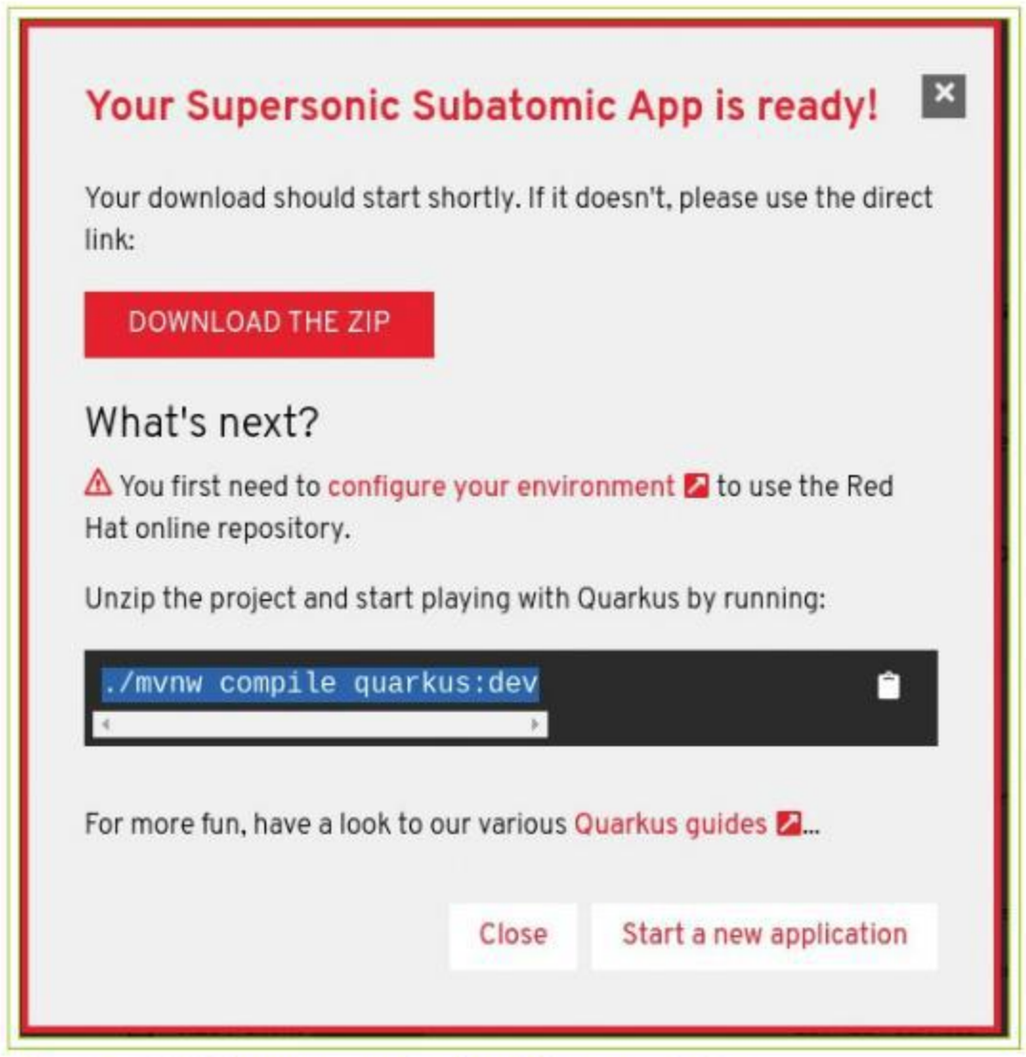


Fig. 2 : Téléchargement de l'archive produite.

Attention, Quarkus requiert une version récente du langage Java (au moins égale ou supérieur à la version 11). Il est donc important, au préalable, de ne pas oublier de correctement configurer la machine virtuelle Java du système :

```
$ export JAVA_HOME=${JAVA_HOME:-'/usr/lib/jvm/
java-11-openjdk'}
$ export PATH=${JAVA_HOME}/bin:${PATH}
```

Ceci fait, on peut exécuter la commande indiquée par le générateur de projet :

```
$./mvnw compile quarkus:dev
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources
(default-resources) @ code-with-quarkus ---
[INFO] Using 'UTF-8' encoding to copy filtered
resources.
[INFO] Copying 2 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile
(default-compile) @ code-with-quarkus ---
[INFO] Nothing to compile - all classes are up
to date
[INFO]
[INFO] --- quarkus-maven-plugin:1.7.5.Final-
redhat-00007:dev (default-cli) @ code-with-
quarkus ---
Listening for transport dt_socket at address:
5005
...

__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
Powered by Quarkus 1.7.5.Final-redhat-00007
2020-11-20 12:37:50,067 INFO  [io.quarkus]
(Quarkus Main Thread) code-with-quarkus
1.0.0-SNAPSHOT on JVM (powered by Quarkus
1.7.5.Final-redhat-00007) started in 1.085s.
Listening on: http://0.0.0.0:8080
2020-11-20 12:37:50,082 INFO  [io.quarkus]
(Quarkus Main Thread) Profile dev activated.
Live Coding activated.
2020-11-20 12:37:50,082 INFO  [io.quarkus]
(Quarkus Main Thread) Installed features: [cdi,
resteasy]
```

Notez bien dans l'extrait de console précédent que la version de Quarkus utilisée contient, dans son numéro, le suffixe **-redhat**. Ceci veut dire que la compagnie a généré (depuis les fichiers sources) l'ensemble des binaires du projet. Elle peut donc ainsi se porter garante de son contenu et fournir des correctifs sur l'ensemble du code du projet. Et ce, y compris sur d'éventuels composants communautaires que la compagnie a pris soin de générer de la même manière depuis leurs sources disponibles. Cette recompilation se fait à partir des fichiers publiés et disponibles pour l'ensemble de la communauté. Seule exception, bien sûr : les éventuels correctifs de sécurité qui sont toujours sous embargo et qui ne doivent pas être publiés.

VERSION SUPPORTÉE ET ACCÈS AU SUPPORT

Un point important avant de passer à la suite : l'utilisation d'archives JAR fournies par Red Hat ne signifie pas que le projet sera automatiquement supporté par la compagnie. Bien évidemment, il faut être un client Red Hat détenteur de la souscription nécessaire pour bénéficier de support sur Quarkus.

En termes de fonctionnalité, il n'y a donc pas de différence entre la version Open Source et celle supportée par Red Hat. C'est seulement à des fins de certification et de conformité que Red Hat est obligé de fournir ses propres archives binaires et ne peut simplement fournir du support sur celles publiées par les pendants communautaires de ses produits.

GRADLE

Le très populaire concurrent de Maven, Gradle [4] est lui aussi supporté par Quarkus et peut être utilisé à la place.

Une fois généré, le projet vient avec un squelette de microservice prêt à l'emploi sous forme d'une simple classe Java :

```
package org.acme;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
```

```
@Path("/hello")
public class ExampleResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

Non seulement ce code contient tout ce dont nous avons besoin pour mettre en place un service REST, mais il est appréciable de noter aussi que les dépendances sont toutes issues des standards Jakarta EE (`javax.ws...`) et non du projet Quarkus et encore moins d'une technologie propriétaire à Red Hat (pas de dépendance sous le nom de paquet `com.redhat...`).

À titre de comparaison, on constate que l'utilisation de **Spring**, à l'inverse, impose l'import de classes issues de paquets propriétaires au framework :

```
package com.example.demo;

import org.springframework.boot.
SpringApplication;
import org.springframework.boot.
autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

 public static void main(String[] args) {
  SpringApplication.run(DemoApplication.
class, args);
 }
}
```

Attention cependant, il ne faut pas en conclure que Quarkus est « meilleur » que **Springboot** pour cette unique raison. Il faut simplement retenir ici que Quarkus vient avec une adhérence à une API standardisée et que Springboot vient avec une adhérence à sa bibliothèque Open Source.

Avant d'aller plus loin, testons que le service fonctionne correctement :

```
$ curl -v -X GET http://localhost:8080/
hello  ; echo ""
Note: Unnecessary use of -X or --request,
GET is already inferred.
*   Trying ::1:8080...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.66.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Length: 5
< Content-Type: text/plain;charset=UTF-8
<
* Connection #0 to host localhost left intact
hello
```

Dans notre première publication au sujet de Quarkus **[5]**, nous avons fait état des performances du cadre d'exécution, mais aussi du confort de développement. Avec Quarkus, on dispose en effet du déploiement à chaud et d'une grande facilité de mise en place de tests unitaires ou du débogage.

Nous avons évoqué en détail les formats de packaging proposés par Quarkus, soit le déploiement sous forme de simple archive JAR complète (surnommé le mode Java native) et évoqué aussi la possibilité, sous des contraintes techniques très fortes, de générer un exécutable natif (libérant le système de déploiement du besoin d'installer une machine virtuelle Java).

Tous ces éléments sont essentiels à la bonne compréhension de la valeur de la proposition de Quarkus en tant que cadre de développement et d'exécution Java. C'est pour cette raison que nous les rappelons ici, mais à l'inverse des précédentes publications, nous n'allons pas rentrer dans le détail de ces fonctionnalités.

Le lecteur est invité à se référer aux différents articles publiés sur Quarkus, s'il souhaite plus d'information à leurs sujets **[5, 6, 7, 8]**.

2. QUARKUS PAR RED HAT

Comme nous venons de l'expliciter, la version supportée par Red Hat est construite à partir des mêmes sources que la version communautaire associée. Néanmoins, celle-ci bénéficie d'un travail conséquent et appréciable de la part de l'éditeur, qui n'est pas forcément évident. Nous allons prendre le temps de l'évoquer un peu ici, pas tellement afin de vendre des souscriptions (le lecteur de cet article n'est probablement pas la bonne cible pour une telle démarche commerciale), mais sur un angle technique, de manière à bien comprendre ce que l'utilisateur gagne à utiliser la version fournie par Red Hat (même sans souscription).

Tout d'abord, il faut réaliser que la version supportée a été testée dans de nombreux environnements (différents types et versions de systèmes d'exploitation et de machines virtuelles Java) et de manière plus intensive que ne le permet l'infrastructure de tests du projet Open Source. En outre, les équipes de qualification ont pris soin d'aller plus loin que les cas couverts par la suite de tests du projet. En résumé, il y a une réelle plus-value en termes de qualification à la version supportée par Red Hat.

En outre, comme tous les produits de l'éditeur, cette version est surveillée par les équipes de sécurité et toute vulnérabilité découverte en son sein sera étudiée et corrigée, aussi rapidement que possible.

Voyons maintenant quelques caractéristiques supplémentaires du produit proposé par le chapeau rouge.

2.1 Certification pour OpenShift 4.5

Si Quarkus a toujours été compatible avec **OpenShift**, les versions communautaires n'ont jusqu'à maintenant pas été certifiées sur la plateforme. Pour la version proposée par Red Hat, cet important travail de qualification a été effectué. C'est un point essentiel, car cette plateforme connaît une très large adoption et est devenue, pour beaucoup d'entreprises et de larges organisations, le cadre de déploiement par défaut.

Cet élément est aussi important pour l'utilisateur communautaire de Quarkus. En effet, toutes versions postérieures à celle proposée par Red Hat ont, par construction, bénéficié de ce travail de la qualification. Il est donc recommandé d'utiliser celles-ci afin de limiter les risques de difficultés de déploiement sur OpenShift.

Toujours dans le contexte d'OpenShift, il est important de noter que cette version de Quarkus supporte aussi l'intégration avec sa fonctionnalité Serverless. Là encore, l'utilisateur de la version communautaire de Quarkus bénéficie tout autant de cette intégration.

2.2 Intégration avec Red Hat Data Grid et Red Hat SSO

Si le projet communautaire Quarkus a toujours permis de s'intégrer avec **Infinispan** et **Keycloak**, le fait de disposer avec cette version Red Hat d'une intégration avec leur pendant commercial (soit **Red Hat Data Grid** et **Red Hat Single Sign On**) est un apport important, car celui-ci facilite grandement le déploiement d'applications Quarkus au sein de systèmes d'information ayant déjà mis en place ces produits.

Si cet aspect n'a que peu d'importance pour un déploiement communautaire, il est à l'inverse critique dans le contexte de systèmes d'information d'entreprises, où l'utilisation de produits certifiés est souvent un prérequis à tout nouveau déploiement.

2.3 Compatibilité avec Spring

Le framework de développement et d'exécution Java Spring est très populaire, et ce depuis des années. On le voit souvent comme un concurrent au standard décrit par la spécification Jakarta EE, mais ses librairies et fonctionnalités sont aussi utilisables de manière conjointe à celui-ci. C'est donc très appréciable que Quarkus ait pris le soin, dès son démarrage, d'assurer une compatibilité avec Spring.

Et cette compatibilité est aujourd'hui mise en avant comme l'une des qualités de la version proposée par Red Hat.

2.4 Support pour les appels distants avec gRPC

Même dans la version proposée par Red Hat, Quarkus supporte l'utilisation de **gRPC [9]** afin de réaliser des appels à procédures distantes. Ces appels fonctionnent aussi si l'on déploie le serveur distant au sein d'OpenShift. Et bien sûr, Quarkus permet d'utiliser les tout derniers protocoles liés à la sécurité (TLS) et à l'authentification, lors de la réalisation de ces appels.

2.5 Programmation réactive avec Mutiny

L'écosystème Java d'entreprise, spécialement s'il repose sur les précédentes versions de la spécification JEE, n'est pas propice à l'utilisation de la programmation réactive. C'est regrettable, car les qualités de ce modèle de programmation sont pourtant de plus en plus appréciées des développeurs depuis maintenant plusieurs années.

L'arrivée du framework de programmation réactive **Mutiny** par le truchement de Quarkus est une excellente nouvelle pour de nombreuses équipes de développement qui ne disposaient pas, jusqu'à maintenant, d'une offre similaire (tout du moins dans le catalogue de Red Hat) qu'elles puissent utiliser au sein de leur organisation.

CONCLUSION

Dans cet article, nous avons fait un tour des raisons des principales caractéristiques techniques spécifiques à la version supportée par Red Hat de Quarkus, ce qui nous a donné l'occasion de revenir une nouvelle fois sur ce novateur et dynamique projet, qui offre un cadre de développement et d'exécution Java absolument phénoménal. Si les précédentes publications n'ont pas déjà incité le lecteur à explorer ce nouveau framework, nous espérons donc que ce nouvel article l'y amènera ! ■

RÉFÉRENCES

[1] Quarkus : https://quarkus.io/

[2] REST : https://fr.wikipedia.org/wiki/Representational_state_transfer/

[3] Maven : http://maven.apache.org/

[4] Gradle : https://gradle.org/

[5] R. PELISSE, « Quarkus : sécurisez votre code Java avec des conteneurs », GNU/Linux Magazine n°227, juin 2019 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-227/Quarkus-securisez-votre-code-Java-avec-des-conteneurs/

[6] R. PELISSE, « Service REST ultrarapide avec Quarkus », GNU/Linux Magazine n°229, septembre 2019 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-229/Service-ReST-ultrarapide-avec-Quarkus

[7] R. PELISSE, « Utiliser Quarkus avec Panache », GNU/Linux Magazine n°231, novembre 2019 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-231/Utiliser-Quarkus-avec-Panache

[8] R. PELISSE, « Programmation réactive avec Quarkus », GNU/Linux Magazine n°231, novembre 2019 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-237/Programmation-reactive-avec-Quarkus

[9] gRPC : https://grpc.io/

LE FONCTIONNEMENT DES NAMESPACES DANS LE NOYAU

RACHID KOUCHA
[Ingénieur développement logiciel]

MOTS-CLÉS : NAMESPACES, ISOLATION, NOYAU LINUX, VFS, NSFS, PROCFS, INODE

Après la présentation des structures de données supportant les namespaces, ce nouvel opus se consacre à la partie immergée dans le noyau des appels système.

Nous continuons notre balade hors des sentiers battus dans le code source de **Linux 5.3.0** afin de nous pencher sur le fonctionnement des appels système dans le noyau.

NOTE

Le code de cet article est disponible sur http://www.rkoucha.fr/tech_corner/linux_namespaces/linux_namespaces.tgz.

1. NSFS

Le **V**irtual **F**ile system **S**witch (**VFS**) **[1]** de Linux est une couche d'abstraction qui présente à l'utilisateur un ensemble d'opérations génériques (par exemple `open()`, `close()`, `ioctl()`, etc.) en masquant les spécificités des différents types de systèmes de fichiers **[2]**. Le **N**ame**S**pace **F**ile **S**ystem (**NSFS**) est l'un d'eux et il est dédié aux namespaces **[3]**.

1.1 Interactions avec PROCFS

NSFS gère plus précisément les cibles des liens symboliques du répertoire `/proc/<pid>/ns`. Son code source se trouve dans `fs/nsfs.c`. Il n'est pas monté explicitement par l'utilisateur, mais de manière interne lors de l'initialisation du noyau (fonction `nsfs_init()` étiquetée avec `__init`) :

```
static struct file_
system_type nsfs = {
 .name = "nsfs",
 .init_fs_context =
nsfs_init_fs_context,
```

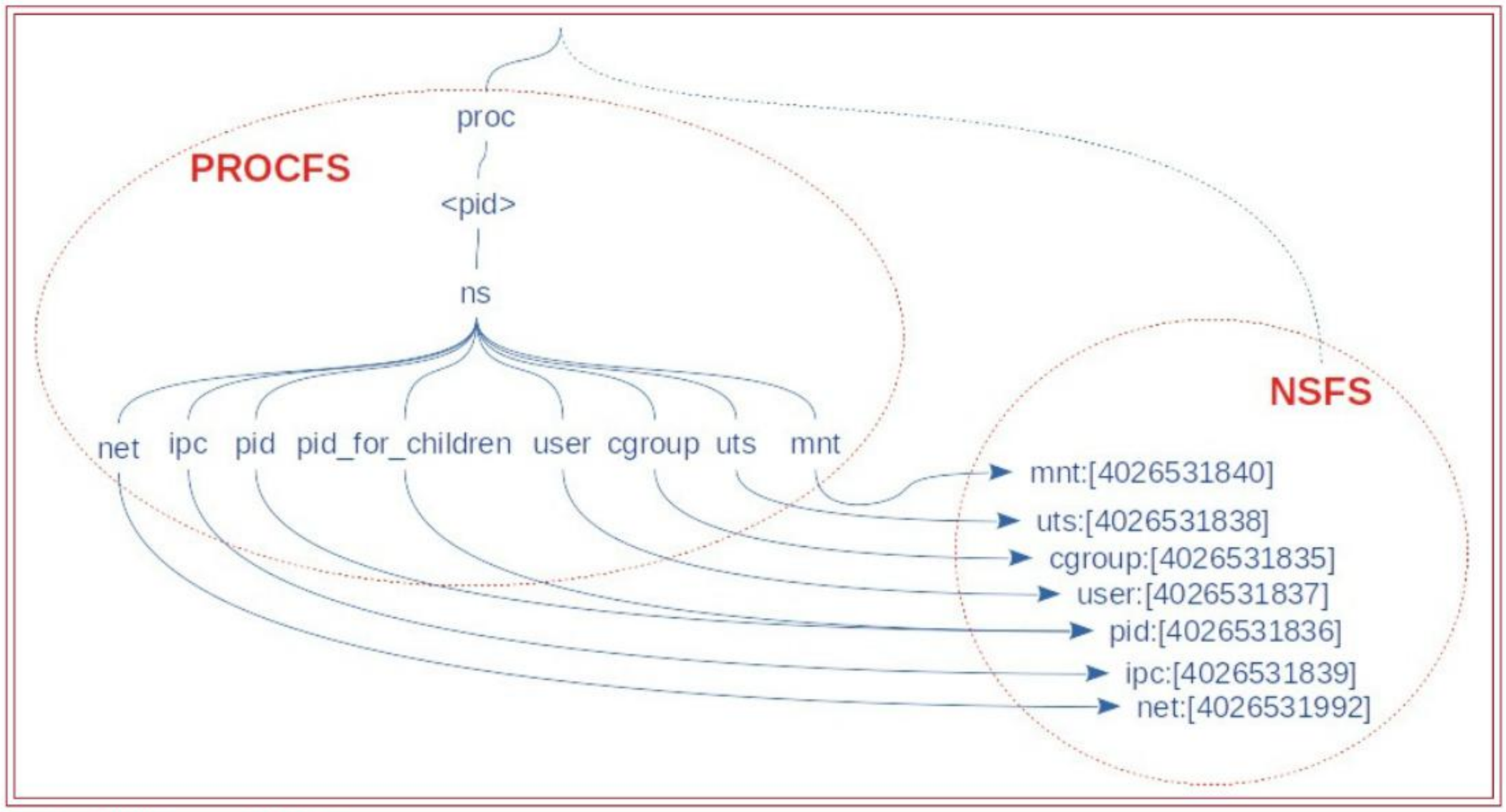


Fig. 1 : NSFS versus PROCFS.

```
 .kill_sb = kill_anon_super,
};

void __init nsfs_init(void)
{
 nsfs_mnt = kern_mount(&nsfs);
[...]
 nsfs_mnt->mnt_sb->s_flags &= ~SB_NOUSER;
}
```

La figure 1 schématise la gestion des fichiers relatifs aux namespaces à travers **PROCFS** d'une part (les liens symboliques) et **NSFS** d'autre part (cibles des liens symboliques).

Notre programme **linfo** prend en paramètre le chemin d'un lien symbolique et affiche des informations sur le lien et sa cible, en s'appuyant respectivement sur les appels système **lstat()** et **stat()** :

```
int main(int ac, char *av[])
{
[...]
  // Get information on the symbolic link
  rc = lstat(av[1], &stl);
[...]
  // Get the information on the target
  rc = stat(av[1], &stt);
[...]
```

```
  // Get the name of the target
  memset(lname, 0, sizeof(lname));
  ssz = readlink(av[1], lname, sizeof(lname));
[...]
  printf("Symbolic link:\n"
         "\tName: %s\n"
         "\tRights: 0%o\n"
         "\tDevice (major/minor): 0x%x/0x%x\n"
         "\tInode: 0x%x (%u)\n"
         "Target:\n"
         "\tName: %s\n"
         "\tRights: 0%o\n"
         "\tDevice (major/minor): 0x%x/0x%x\n"
         "\tInode: 0x%x (%u)\n"
         ,
         av[1],
         stl.st_mode & 0777,
         major(stl.st_dev), minor(stl.st_dev),
         (unsigned int)(stl.st_ino), (unsigned
int)(stl.st_ino),
         lname,
         stt.st_mode & 0777,
         major(stt.st_dev), minor(stt.st_dev),
         (unsigned int)(stt.st_ino), (unsigned
int)(stt.st_ino)
        );
```

Lancé avec le lien sur le net_ns associé au shell courant, **linfo** affiche d'abord les informations retournées par

procfs (le lien symbolique) et les informations retournées par **NSFS** (la cible du lien) :

```
$ ./linfo /proc/self/ns/net
Symbolic link:
 Name: /proc/self/ns/net
 Rights: 0777
 Device (major/minor): 0x0/0x5
 Inode: 0x14a61 (84577)
Target:
 Name: net:[4026531992]
 Rights: 0444
 Device (major/minor): 0x0/0x4
 Inode: 0xf0000098 (4026531992)
```

Pour la cible, on retrouve bien le numéro d'inode affiché dans le nom du fichier par la commande **ls** dans le répertoire :

```
$ ls -l /proc/self/ns/net
lrwxrwxrwx 1 rachid rachid 0 avril   6 11:43
/proc/self/ns/net -> 'net:[4026531992]'
```

Notre programme **fsinfo** reçoit en paramètre le nom d'un fichier et s'appuie sur l'appel système **statfs()** pour afficher des informations à propos de son système de fichiers :

```
int main(int ac, char *av[])
{
[...]
  // Get information on the file system
  rc = statfs(av[1], &st);
[...]
  printf("Type              : 0x%lx (%s)\n"
         "Block size        : %lu\n"
         "Total blocks      : %lu\n"
         "Total free blocks: %lu\n"
         "Total files       : %lu\n"
         "Max name length   : %lu\n"
         "Mount flags       : 0x%lx (%s)\n"
         ,
         (unsigned long)(st.f_type), get_
fstype((unsigned long)(st.f_type)),
         (unsigned long)(st.f_bsize),
         (unsigned long)(st.f_blocks),
         (unsigned long)(st.f_bfree),
         (unsigned long)(st.f_files),
         (unsigned long)(st.f_namelen),
         (unsigned long)(st.f_flags), get_
mntflags((unsigned long)(st.f_flags))
        );
[...]
```

Pour la cible d'un lien symbolique sur un namespace, il retourne bien le type **NSFS** :

```
$ ./fsinfo /proc/self/ns/pid
Type              : 0x6e736673 (NSFS)
Block size        : 4096
Total blocks      : 0
Total free blocks: 0
Total files       : 0
Max name length   : 255
Mount flags       : 0x20 (REMOUNT)
```

Pour le répertoire où se trouve ce même lien, c'est le type **PROCFS** :

```
$ ./fsinfo /proc/self/ns
Type              : 0x9fa0 (PROCFS)
Block size        : 4096
Total blocks      : 0
Total free blocks: 0
Total files       : 0
Max name length   : 255
Mount flags       : 0x102e (NODEV NOEXEC
NOSUID REMOUNT BIND)
```

Sur l'appel système **readlink()** (utilisé dans le programme **linfo** pour récupérer la cible du lien symbolique) dans **/proc/<pid>/ns**, le système de fichiers **PROCFS** appelle le service interne **ns_get_name()** de **NSFS** pour retourner le nom des cibles (cf. fonction **proc_ns_readlink()** dans **fs/proc/namespaces.c**) :

```
static int proc_ns_readlink(struct
dentry *dentry, char __user *buffer,
int buflen)
{
 struct inode *inode = d_inode(dentry);
 const struct proc_ns_operations *ns_
ops = PROC_I(inode)->ns_ops;
 struct task_struct *task;
 char name[50];
 int res = -EACCES;

 task = get_proc_task(inode);
[...]
 if (ptrace_may_access(task, PTRACE_
MODE_READ_FSCREDS)) {
  res = ns_get_name(name, sizeof(name),
task, ns_ops);
  if (res >= 0)
   res = readlink_copy(buffer, buflen,
name);
 }
 put_task_struct(task);
 return res;
}
```

Définie dans le fichier **fs/nsfs.c**, la fonction **ns_get_name()** construit le nom de la cible avec les informations du descripteur de namespace (c'est-à-dire les champs **name**, **real_name** et **inum** de la structure **ns_common** vue dans l'article précédent dans GNU/Linux Magazine n°243) :

```
int ns_get_name(char *buf, size_t size, struct task_struct *task,
    const struct proc_ns_operations *ns_ops)
{
 struct ns_common *ns;
 int res = -ENOENT;
 const char *name;
 ns = ns_ops->get(task);
 if (ns) {
  name = ns_ops->real_ns_name ? : ns_ops->name;
  res = snprintf(buf, size, "%s:[%u]", name, ns->inum);
  ns_ops->put(ns);
 }
 return res;
}
```

1.2 Les inodes

Lorsque la cible d'un lien symbolique dans **/proc/<pid>/ns** est demandée, l'opération **proc_ns_get_link()** dans **fs/proc/namespace.c** est déclenchée. Cette dernière appelle la fonction **ns_get_path()** dans **fs/nsfs.c** qui finit par appeler **__ns_get_path()** pour allouer l'inode et le dentry associé :

```
static void *__ns_get_path(struct path *path, struct ns_common *ns)
{
 struct vfsmount *mnt = nsfs_mnt;
 struct dentry *dentry;
 struct inode *inode;
 unsigned long d;

 rcu_read_lock();
 d = atomic_long_read(&ns->stashed);
 if (!d)
  goto slow;
 dentry = (struct dentry *)d;
 if (!lockref_get_not_dead(&dentry->d_lockref))
  goto slow;
 rcu_read_unlock();
 ns->ops->put(ns);
got_it:
 path->mnt = mntget(mnt);
 path->dentry = dentry;
 return NULL;
slow:
 rcu_read_unlock();
 inode = new_inode_pseudo(mnt->mnt_sb);
[...]
 inode->i_ino = ns->inum;
 inode->i_mtime = inode->i_atime = inode->i_ctime = current_time(inode);
 inode->i_flags |= S_IMMUTABLE;
 inode->i_mode = S_IFREG | S_IRUGO;
 inode->i_fop = &ns_file_operations;
 inode->i_private = ns;

 dentry = d_alloc_anon(mnt->mnt_sb);
[...]
 d_instantiate(dentry, inode);
 dentry->d_fsdata = (void *)ns->ops;
 d = atomic_long_cmpxchg(&ns->stashed, 0, (unsigned long)dentry);
[...]
 goto got_it;
}
```

L'inode ainsi alloué est présenté en figure 2.

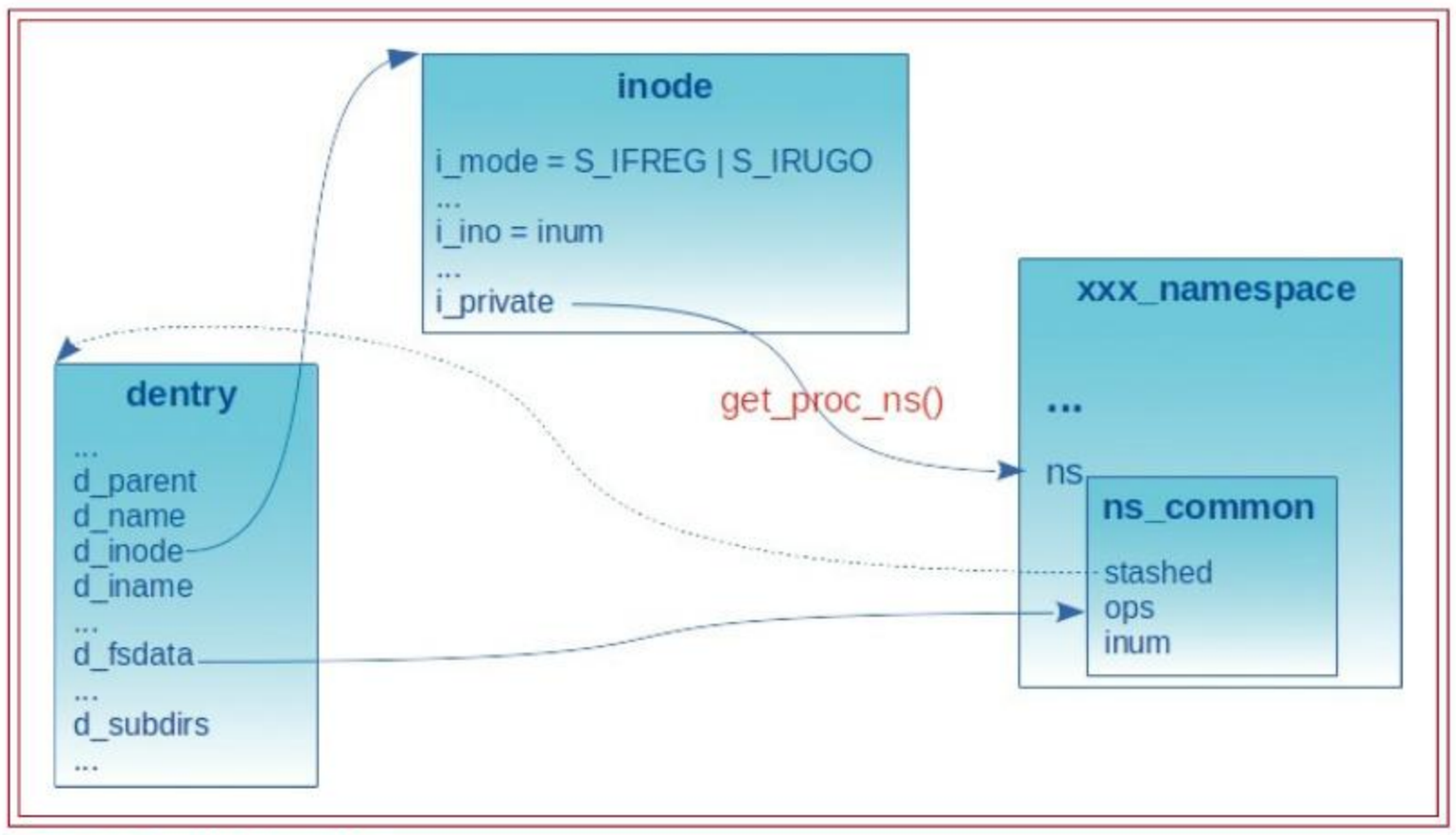


Fig. 2 : Inode dans nsfs.

Le champ **ns** de type **ns_common** dans le descripteur de namespace est pointé par le champ **i_private** de la structure **inode**. Le numéro d'inode stocké dans le champ **inum** du namespace lors de son allocation est assigné au champ **i_ino**. Le champ **stashed** de **ns_common** pointe sur la structure **dentry** liée à l'inode. Ce champ permet de savoir si l'inode est associé à un dentry ou pas, car dans certaines situations où les ressources mémoire se raréfient, le **VFS** peut faire de « l'élagage » (c.-à-d. « prune » en anglais) dans le cache dentry, de sorte à libérer des ressources. Dans ce cas, cela aboutit à l'appel à la fonction **ns_prune_dentry()** dans **fs/nsfs.c** qui marque le champ à **0** :

```
static void ns_prune_dentry(struct dentry
*dentry)
{
 struct inode *inode = d_inode(dentry);
 if (inode) {
  struct ns_common *ns = inode->i_private;
  atomic_long_set(&ns->stashed, 0);
 }
}
```

1.3 Du descripteur de fichier au namespace

L'appel système **open()** sur un lien symbolique de namespace aboutit via les fonctions précédentes à l'allocation d'une structure **dentry**, de l'inode et la structure **file**. Un descripteur de fichier est retourné côté espace utilisateur.

Ensuite, avec les différentes fonctions de translation du **VFS**, le descripteur de fichier passé aux appels système, tels que **ioctl()** ou **setns()**, permet de retrouver la structure **file** avec le service **proc_ns_fget()** défini dans **fs/nsfs.c** :

```
struct file *proc_ns_fget(int fd)
{
 struct file *file;

 file = fget(fd);
[...]
 if (file->f_op != &ns_file_operations)
  goto out_invalid;

 return file;
[...]
}
```

Le service **file_inode()** du fichier **include/linux/fs.h** retrouve l'inode à partir de la structure **file** :

```
static inline struct inode *file_
inode(const struct file *f)
{
 return f->f_inode;
}
```

Le service **get_proc_ns()** du fichier **include/linux/proc_ns.h** retrouve un namespace à partir d'un pointeur sur une structure **inode** :

```
#define get_proc_ns(inode) ((struct ns_
common *)(inode)->i_private)
```

Le champ **i_private** de l'inode pointe sur la structure du namespace et les opérations associées comme indiqué en figure 2.

1.4 Les montages

Un namespace est désalloué par le noyau à partir du moment où il n'est plus référencé (valeur **0** pour le compteur **kref** ou **count** vu dans l'article précédent). En général, il n'est plus référencé lorsqu'il n'y a plus aucune tâche associée. Cependant, comme on l'a vu avec la commande **ip** par exemple, il peut s'avérer nécessaire de garder un namespace actif, même s'il n'a plus de tâche associée. Garder une tâche active liée au namespace est une solution, mais ce n'est pas très économique en termes de ressources de mémoire et CPU. L'astuce consiste à monter le fichier du namespace sur un autre fichier dans l'arborescence. Ainsi, tant qu'il est monté, le namespace reste référencé **[4]**. Reprenons l'exemple de création d'un net_ns avec la commande **ip** :

```
# ls -l /run/netns
ls: cannot access '/run/netns': No such file
or directory
# ip netns add newnet
# ls -l /run/netns
total 0
-r--r--r-- 1 root root 0 avril  5 12:17 newnet
# cat /proc/$$/mountinfo
[...]
634 26 0:23 /netns /run/netns
rw,nosuid,noexec,relatime shared:5 - tmpfs
tmpfs rw,size=1635172k[...]
655 634 0:4 net:[4026532881] /run/netns/newnet
rw shared:359 - nsfs nsfs rw
656 26 0:4 net:[4026532881] /run/netns/newnet
rw shared:359 - nsfs nsfs rw
```

La commande a appelé **unshare(CLONE_NEWNET)** pour créer un nouveau net_ns. Une fois créée, la seule tâche associée au net_ns est la commande **ip** elle-même. Pour que ce net_ns ne disparaisse pas à la fin de la commande (c.-à-d. lorsqu'elle rend la main à l'opérateur), l'astuce consiste à monter **/proc/self/ns/net** (c.-à-d. la cible de ce lien symbolique !) sur **/run/netns/newnet** (c.-à-d. le nom passé en paramètre) afin de laisser une référence sur le net_ns. Le fichier **mountinfo** montre le type de système de fichiers **NSFS** et le nom du fichier cible.

Ainsi, la commande **ip** pourra accéder de nouveau au net_ns pour y exécuter des commandes ou y migrer des interfaces réseau. Listons les net_ns créés :

```
# ip netns list
newnet
```

Listons les interfaces réseau dans le net_ns à l'aide de la commande **ip link list** associée à ce net_ns. On y trouve uniquement l'interface **loopback**, car c'est la seule interface disponible dans tout nouveau net_ns. On remarquera qu'elle a l'indice **1** comme nous l'avions mentionné dans l'article précédent :

```
# ip netns exec newnet ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state
DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
```

2. LES APPELS SYSTÈME

2.1 Les identifiants

Un certain nombre de fonctions et macros « helpers » ont été définies en interne :

- **pid_nr()** : identifiant de processus (global) vu du pid_ns initial ;
- **pid_vnr()** : identifiant de processus (virtuel) vu du pid_ns courant ;
- **pid_nr_ns()** : identifiant de processus (virtuel) vu du pid_ns passé en paramètre ;
- **pid_<xid>_nr()** : identifiant (global) vu du namespace initial ;
- **pid_<xid>_vnr()** : identifiant (virtuel) vu du namespace courant ;
- **pid_<xid>_nr_ns()** : identifiant (virtuel) vu du namespace passé en paramètre.

Ces dernières sont largement mises à contribution dans les appels système tels que **getpid()**, **gettid()**, **getgid()**, **getuid()** et autres afin de les **virtualiser** (c.-à-d. leur faire retourner un résultat correspondant aux namespaces auxquels le processus appelant est associé).

2.2 ioctl

Le point d'entrée **ioctl** du système de fichiers **NSFS** est la fonction **ns_ioctl()** dans le fichier **fs/nsfs.c**. Le début de la fonction retrouve l'inode à partir de la structure **file** passée en paramètre. Puis la structure **ns_common** du namespace cible est retrouvée (variable locale **ns**) depuis l'inode, comme on l'a vu précédemment (cf. figure 2) :

```
static long ns_ioctl(struct file *filp,
unsigned int ioctl,
    unsigned long arg)
{
 struct user_namespace *user_ns;
 struct ns_common *ns = get_proc_ns(file_
inode(filp));
 uid_t __user *argp;
 uid_t uid;
```

Dans l'article précédent, nous avons vu que le champ **ops** de la structure **ns_common** contient des informations et opérations communes à tous les namespaces. Elles vont être exploitées dans la suite.

Comme souvent avec les fonctions **ioctl()**, l'algorithme consiste en un « switch/case » pour chaque opération supportée :

```
 switch (ioctl) {
```

L'opération **NS_GET_USERNS** déclenche le point d'entrée **owner** afin de retourner le descripteur de fichiers sur le user_ns propriétaire :

```
case NS_GET_USERNS:
 return open_related_ns(ns, ns_get_owner);
```

L'opération **NS_GET_PARENT** déclenche le point d'entrée **get_parent** afin de retourner le descripteur de fichier sur le namespace parent. Le retour erreur **EINVAL** est pour les namespaces non hiérarchiques (c.-à-d. autres que user_ns et pid_ns). Leur champ **get_parent** est **NULL** :

```
 case NS_GET_PARENT:
  if (!ns->ops->get_parent)
   return -EINVAL;
  return open_related_ns(ns, ns->ops->get_
parent);
```

L'opération **NS_GET_TYPE** retourne la valeur du champ **type** (c.-à-d. un drapeau **CLONE_NEWXXX**) :

```
 case NS_GET_NSTYPE:
  return ns->ops->type;
```

L'opération **NS_GET_OWNER_UID** ne s'applique qu'aux user_ns (sous peine de retour **EINVAL**) et retourne l'identifiant d'utilisateur d'un user_ns :

```
 case NS_GET_OWNER_UID:
  if (ns->ops->type != CLONE_NEWUSER)
   return -EINVAL;
  user_ns = container_of(ns, struct user_
namespace, ns);
  argp = (uid_t __user *) arg;
  uid = from_kuid_munged(current_user_ns(),
user_ns->owner);
  return put_user(uid, argp);
```

Un mauvais identifiant d'opération tombe sous le coup du cas « default » pour retourner **ENOTTY**. Ce code d'erreur peut prêter à confusion, car dans le cas où les paramètres n'ont pas les valeurs attendues, on utilise généralement **EINVAL**. Mais c'est devenu une règle dans les ioctls des drivers et systèmes de fichiers afin de signifier que la commande passée n'est pas connue **[5]** :

```
 default:
  return -ENOTTY;
 }
}
```

On comprend maintenant la signification du nom du fichier d'en-tête **<linux/nsfs.h>** nécessaire à l'utilisation de **ioctl()** côté espace utilisateur :

```
[...]
/* Returns a file descriptor that refers to
an owning user namespace */
#define NS_GET_USERNS   _IO(NSIO, 0x1)
/* Returns a file descriptor that refers to
a parent namespace */
#define NS_GET_PARENT   _IO(NSIO, 0x2)
/* Returns the type of namespace (CLONE_NEW*
value) referred to by
   file descriptor */
#define NS_GET_NSTYPE   _IO(NSIO, 0x3)
/* Get owner UID (in the caller's user
namespace) for a user namespace */
#define NS_GET_OWNER_UID _IO(NSIO, 0x4)
[...]
```

L'opération non documentée **SIOCGSKNS** **[6]** qui retourne un descripteur de fichier sur le net_ns auquel une socket est associée est gérée dans **net/socket.c** :

```
static long sock_ioctl(struct file *file,
unsigned cmd, unsigned long arg)
{
 struct socket *sock;
 struct sock *sk;
 void __user *argp = (void __user *)arg;
 int pid, err;
 struct net *net;

 sock = file->private_data;
 sk = sock->sk;
 net = sock_net(sk);
[...]
  case SIOCGSKNS:
   err = -EPERM;
   if (!ns_capable(net->user_ns, CAP_NET_
ADMIN))
    break;

   err = open_related_ns(&net->ns, get_net_ns);
   break;
[...]
```

De la structure **file** est déduit le descripteur de socket duquel on récupère une référence sur le net_ns associé. Ensuite, la fonction vérifie que l'appelant a bien la capacité **CAP_NET_ADMIN** dans le user_ns propriétaire du net_ns avec le service interne **ns_capable()** **[7]**. Enfin, la fonction **open_relelated_ns()** de **fs/nsfs.c** est invoquée pour retourner un descripteur de fichier sur le net_ns.

2.3 clone

Le fichier **kernel/fork.c** contient l'implémentation de l'appel système **clone()**.

```
SYSCALL_DEFINE5(clone, unsigned long,
clone_flags, unsigned long, newsp,
   int __user *, parent_tidptr,
   int __user *, child_tidptr,
   unsigned long, tls)
{
 struct kernel_clone_args args = {
  .flags = (clone_flags & ~CSIGNAL),
```

ABONNEZ-VOUS !

11 MAGAZINES/AN

69 €*

au lieu de 97,90 €

DISPONIBLE EN VERSION PAPIER OU FLIPBOOK

FRAIS DE PORT OFFERTS

*Prix TTC en Euros / France Métropolitaine

DISPONIBLE EN VERSION PAPIER OU FLIPBOOK

Offre Papier : LM1

Offre Flipbook : LM4

11X

69 €*

Prix kiosque : 97,90 €

Économie : 28,90 €

*Prix TTC en Euros / France Métropolitaine

DISPONIBLE EN VERSION PAPIER OU FLIPBOOK

Offre Papier : LM+1

Offre Flipbook : LM+4

129 €*

Prix kiosque : 187,30 €

Économie : 58,30 €

*Prix TTC en Euros / France Métropolitaine

FLIPBOOK HTML5

DÉCOUVREZ LE FLIPBOOK ! sur : www.ed-diamond.com

BULLETIN D'ABONNEMENT

JE M'ABONNE À
GNU/LINUX MAGAZINE

- ☐ Offre **LM1** - 11 numéros pour 69 €* (papier)
- ☐ Offre **LM4** - 11 numéros pour 69 €* (Flipbook)

JE M'ABONNE À **GNU/LINUX MAGAZINE ET SES HORS-SÉRIES**

- ☐ Offre **LM+1** - 11 numéros et 6 hors-séries pour 129 €* (papier)
- ☐ Offre **LM+4** - 11 numéros et 6 hors-séries pour 129 €* (Flipbook)

*Prix TTC en Euros / France Métropolitaine - Les tarifs hors France Métropolitaine, Europe, Asie, etc. sont disponibles en ligne !

À découper ou recopier et à renvoyer avec votre réglement à :

Les Éditions Diamond Service des Abonnements
10 Place de la Cathédrale – 68000 Colmar – France

JE RÈGLE ..... €

- ☐ par **chèque bancaire** ou **postal** à l'ordre des Éditions Diamond (uniquement France et DOM TOM)
- ☐ Pour les règlements par **virement**, veuillez nous contacter **par e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20**

COORDONNÉES DE L'ABONNÉ

Société : .....
Nom : .....
Prénom : .....
Adresse : .....
.....
Code Postal : .....
Ville : .....
Pays : .....
Téléphone : .....
E-mail : .....

☐ J'autorise GNU/Linux Magazine à me contacter par e-mail ou par téléphone

RETROUVEZ TOUTES NOS OFFRES SUR : www.ed-diamond.com !

Chez votre marchand de journaux

et sur **www.ed-diamond.com**

PAPIER
en kiosque

FLIPBOOK HTML5
sur www.ed-diamond.com

CONNECT
LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT
sur connect.ed-diamond.com

```
  .pidfd  = parent_tidptr,
  .child_tid = child_tidptr,
  .parent_tid = parent_tidptr,
  .exit_signal= (clone_flags & CSIGNAL),
  .stack  = newsp,
  .tls  = tls,
 };
[...]
 return _do_fork(&args);
}
```

La fonction **clone()** appelle **copy_process()** à partir de **_do_fork()** pour dupliquer la tâche appelante :

```
/*
 * This creates a new process as a copy
of the old one,
 * but does not actually start it yet.
 *
 * It copies the registers, and all the
appropriate
 * parts of the process environment (as
per the clone
 * flags). The actual kick-off is left
to the caller.
 */
static __latent_entropy struct task_
struct *copy_process(
     struct pid *pid,
     int trace,
     int node,
     struct kernel_clone_args *args)
{
[...]
```

La fonction interdit la combinaison des drapeaux **CLONE_NEWNS** et **CLONE_FS**, car deux processus évoluant dans des mount_ns différents ne peuvent pas partager le répertoire racine ou le répertoire courant. Elle interdit aussi la combinaison des drapeaux **CLONE_NEWUSER** et **CLONE_FS**, car deux processus évoluant dans des user_ns différents ne peuvent pas partager les mêmes informations dans le système de fichiers pour raisons de sécurité :

```
 if ((clone_flags & (CLONE_NEWNS|CLONE_FS))
== (CLONE_NEWNS|CLONE_FS))
  return ERR_PTR(-EINVAL);

 if ((clone_flags & (CLONE_NEWUSER|CLONE_
FS)) == (CLONE_NEWUSER|CLONE_FS))
  return ERR_PTR(-EINVAL);
```

À moins que le drapeau CLONE_NEWUSER ne soit passé, la fonction ne peut être exécutée qu'à la condition où l'utilisateur a la capacité **CAP_SYS_ADMIN**. Nous verrons la raison de cette exception dans l'article dédié au user_ns :

```
 if ((clone_flags & CLONE_NEWUSER) &&
!unprivileged_userns_clone)
  if (!capable(CAP_SYS_ADMIN))
   return ERR_PTR(-EPERM);
```

Pour des raisons de sécurité, tous les threads d'un processus doivent résider dans le même user_ns et pid_ns. Par conséquent, la fonction interdit la création d'un thread (drapeau **CLONE_THREAD** passé par **pthread_create()** de la librairie C) dans un nouveau user_ns ou pid_ns. Cela comprend aussi le cas où un processus fait un premier appel à **clone(CLONE_NEWPID)** pour créer un premier fils dans un nouveau pid_ns, puis appelle **pthread_create()** pour créer un thread (sans ce contrôle, le nouveau thread s'exécuterait aussi dans le nouveau pid_ns, bien que **CLONE_NEWPID** n'est pas passé).

```
 if (clone_flags & CLONE_THREAD) {
  if ((clone_flags & (CLONE_NEWUSER | CLONE_
NEWPID)) ||
      (task_active_pid_ns(current) !=
    current->nsproxy->pid_ns_for_children))
   return ERR_PTR(-EINVAL);
 }
```

Ensuite, parmi les nombreuses autres actions, il y a la copie (héritage) des informations de sécurité (« credentials ») via **copy_creds()** et des namespaces du processus appelant via **copy_namespaces()**. La copie des informations de sécurité **donne toutes les capacités au nouveau processus si le drapeau CLONE_NEWUSER est passé** (le nouvel user_ns associé est d'ailleurs créé à ce moment-là). De plus, cela nous permet de souligner que **dès lors que le drapeau CLONE_NEWUSER est passé, il est traité en premier** ! Nous verrons l'importance de ces points dans l'article dédié aux user_ns :

```
[...]
 retval = copy_creds(p, clone_flags);
[...]
 retval = copy_namespaces(clone_flags, p);
```

Enfin, la fonction **alloc_pid()** est appelée pour allouer une structure **pid** afin de créer et mémoriser l'identifiant de la nouvelle tâche dans tous les pid_ns en remontant jusqu'à la racine. On notera que c'est ici que le pointeur **pid_ns_for_children** du **nsproxy** est utilisé pour référencer le pid_ns. Ce dernier pointe sur le pid_ns de la tâche appelante (le père) sauf si le drapeau **CLONE_NEWPID** a été passé, auquel cas il pointe sur le pid_ns nouvellement alloué :

```
 pid = alloc_pid(p->nsproxy->pid_ns_for_children);
```

Détaillons la fonction **copy_namespace()** définie dans **kernel/nsproxy.c**. Si aucun des drapeaux **CLONE_NEWXXX** n'est passé, cela signifie que le processus créé hérite des namespaces de l'appelant (la variable locale **old_ns** pointe sur le **nsproxy** de la tâche appelante). **Dans ce cas, on n'alloue pas un nouveau nsproxy, mais la nouvelle tâche va pointer sur le nsproxy de son père** en incrémentant son compteur de références (on a vu cela dans l'article précédent) :

```
 if (likely(!(flags & (CLONE_NEWNS | CLONE_
NEWUTS | CLONE_NEWIPC |
          CLONE_NEWPID | CLONE_NEWNET |
          CLONE_NEWCGROUP)))) {
  get_nsproxy(old_ns);
  return 0;
 }
```

Si au moins un des drapeaux CLONE_NEWXXX est passé, alors on va obligatoirement allouer une nouvelle structure nsproxy. Les capacités du processus appelant sont de nouveau vérifiées (Rappel : si le drapeau **CLONE_NEWUSER** a été passé, toutes les capacités sont activées) :

```
 if (!ns_capable(user_ns, CAP_SYS_ADMIN))
  return -EPERM;
```

La combinaison des drapeaux **CLONE_NEWIPC** et **CLONE_SYSVSEM** n'est pas autorisée, car dans un nouvel ipc_ns, les sémaphores de l'appelant ne sont plus accessibles et par conséquent, il n'est pas possible de partager la liste des **semadj** (mécanisme qui sera détaillé dans un prochain article dédié aux ipc_ns) :

```
if ((flags & (CLONE_NEWIPC | CLONE_SYSVSEM)) ==
 (CLONE_NEWIPC | CLONE_SYSVSEM))
 return -EINVAL;
```

Ensuite est appelée la fonction **create_new_namespaces()** située dans le même fichier. Cette fonction alloue une nouvelle structure **nsproxy**. Pour chaque drapeau **CLONE_NEWXXX** passé en argument à **clone()**, le namespace associé est alloué et le pointeur associé dans le **nsproxy** pointe dessus. Pour les autres namespaces dont les drapeaux ne sont pas spécifiés, il n'y a pas d'allocation de nouveaux namespaces, mais seulement une référence au namespace de l'appelant (avec incrémentation du compteur de références) et le pointeur correspondant du **nsproxy** pointe dessus :

```
 new_ns = create_new_namespaces(flags,
tsk, user_ns, tsk->fs);
```

Enfin, le **nsproxy** ainsi alloué est assigné au champ **nsproxy** de la nouvelle tâche :

```
 tsk->nsproxy = new_ns;
 return 0;
}
```

La figure 3 schématise l'opération **clone(CLONE_NEWUTS)**. Les actions et structures mises en œuvre pour la nouvelle tâche sont en orange. Les pointeurs du **nsproxy** nouvellement alloués sont identiques à ceux du processus père, sauf pour l'uts_ns qui pointe sur le descripteur d'uts_ns nouvellement alloué. Pour des raisons de lisibilité, nous n'avons pas représenté la structure **pid**.

2.4 unshare

Le fichier **kernel/fork.c** contient l'implémentation de l'appel système **unshare()** :

```
SYSCALL_DEFINE1(unshare, unsigned long,
unshare_flags)
{
 return ksys_unshare(unshare_flags);
}
```

Tout le travail est fait dans la fonction **ksys_unshare()** située dans le même fichier. **unshare()** accepte bon nombre de drapeaux en paramètres. Nous nous concentrerons uniquement sur ceux concernant les namespaces. La fonction commence par positionner implicitement les drapeaux **CLONE_THREAD** et **CLONE_FS** si **CLONE_NEWUSER** est passé :

```
int ksys_unshare(unsigned long unshare_flags)
{
 struct fs_struct *fs, *new_fs = NULL;
 struct files_struct *fd, *new_fd = NULL;
 struct cred *new_cred = NULL;
 struct nsproxy *new_nsproxy = NULL;
 int do_sysvsem = 0;
 int err;
 /*
  * If unsharing a user namespace must also
unshare the thread group
  * and unshare the filesystem root and
working directories.
  */
 if (unshare_flags & CLONE_NEWUSER)
  unshare_flags |= CLONE_THREAD | CLONE_FS;
```

Le drapeau **CLONE_FS** est aussi implicitement positionné s'il y a création d'un nouveau mount_ns :

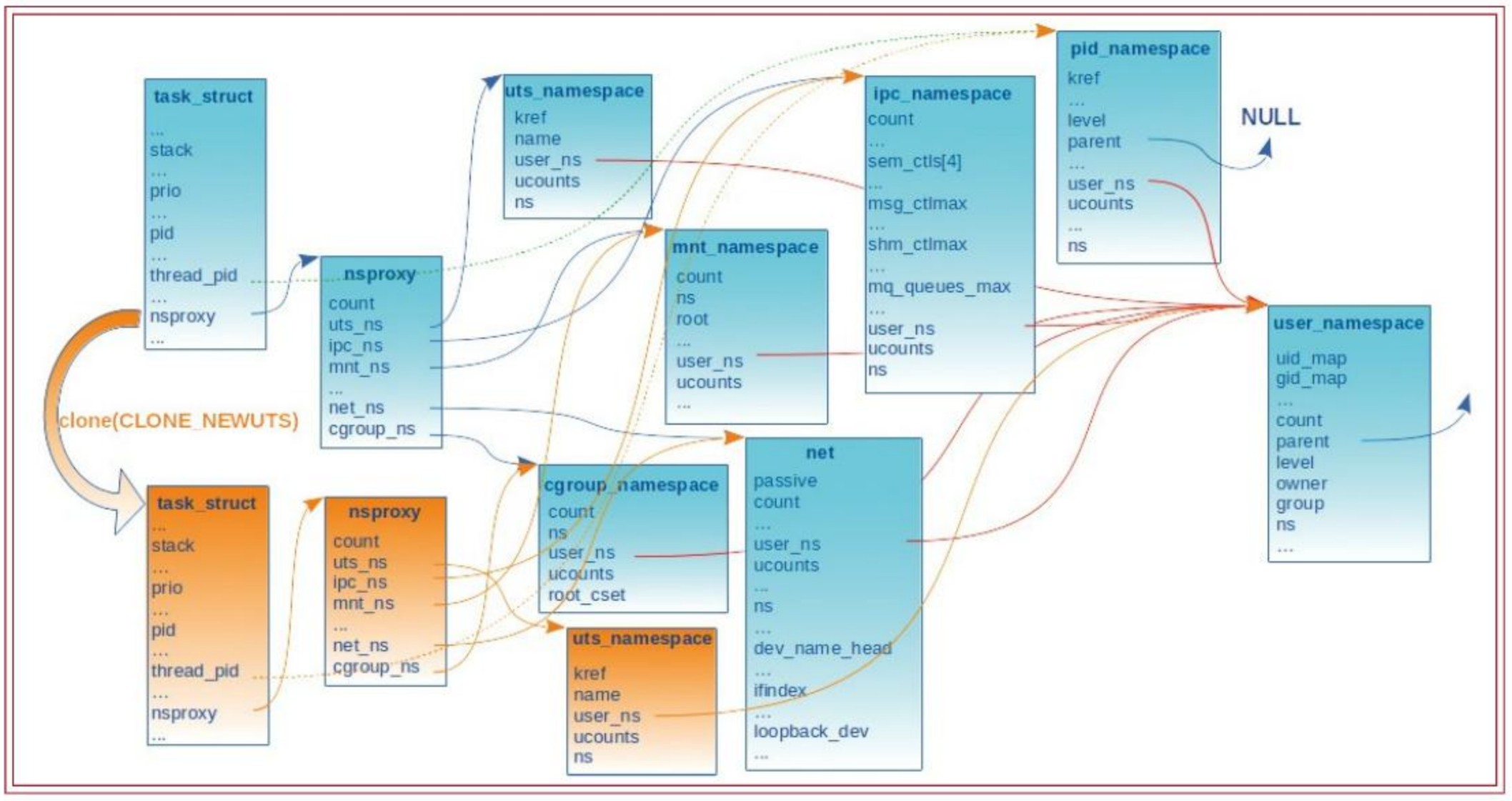


Fig. 3 : Appel système clone() avec CLONE_NEWUTS.

```
 /*
  * If unsharing namespace, must also
unshare filesystem information.
  */
 if (unshare_flags & CLONE_NEWNS)
  unshare_flags |= CLONE_FS;
```

Comme pour **clone()** vu précédemment, si **CLONE_NEWUSER** n'est pas passé, alors la fonction retourne en erreur (**EPERM**) si l'appelant n'a pas la capacité **CAP_SYS_ADMIN**. Lors de l'étude détaillée des user_ns, nous verrons pourquoi il est important d'autoriser la fonction lorsque la création d'un user_ns est demandée :

```
 if ((unshare_flags & CLONE_NEWUSER) &&
!unprivileged_userns_clone) {
  err = -EPERM;
  if (!capable(CAP_SYS_ADMIN))
   goto bad_unshare_out;
 }
```

Un certain nombre de contrôles de cohérence et de validité sur les drapeaux est effectué :

```
 err = check_unshare_flags(unshare_flags);
```

Si le drapeau **CLONE_NEWUSER** est passé, comme précédemment souligné pour **clone()**, **il est traité en premier** en allouant un nouvel user_ns (appel à **unshare_userns()**), puis les autres namespaces sont créés conformément aux autres drapeaux passés en paramètres (appel à **unshare_nsproxy_namespaces()** qui appelle **create_new_namespaces()** vue avec **clone()**). On notera que si **CLONE_NEWIPC** est passé, **un nouveau système de fichiers mqueue** (généralement monté sur **/dev/mqueue** côté espace utilisateur) **est monté en interne pour le nouvel ipc_ns**. L'ancien est « oublié ». En d'autres termes, les noms des queues de message POSIX créés par la tâche courante ne seront plus accessibles pour une destruction dans le système de fichiers, car la tâche ne les verra plus à partir de son nouvel ipc_ns. Par contre, les descripteurs de fichiers actuellement ouverts sur les queues de message permettent toujours d'y accéder pour émettre et recevoir des messages ! On reverra cela dans un exemple lors de l'étude détaillée des ipc_ns.

```
 err = unshare_userns(unshare_flags, &new_cred);
[...]
 err = unshare_nsproxy_namespaces(unshare_flags,
&new_nsproxy, new_cred, new_fs);
```

Si le drapeau **CLONE_NEWIPC** a été passé (la variable locale **do_sysvsem** vaut **1**), la fonction **exit_sem()** est appelée pour mettre en œuvre le mécanisme **semadj** afin d'éviter les interblocages en annulant les opérations en cours sur les sémaphores. Nous reviendrons sur ce sujet dans le paragraphe dédié à l'ipc_ns.

```
 if (new_fs || new_fd || do_sysvsem || new_cred ||
new_nsproxy) {
  if (do_sysvsem) {
```

```
 /*
  * CLONE_SYSVSEM is equivalent to sys_exit().
  */
 exit_sem(current);
}
```

Toujours pour le drapeau **CLONE_NEWIPC**, **exit_shm()** est appelée pour les identifiants de segments de mémoire partagée. Les segments créés par la tâche courante sont marqués orphelins (c.-à-d. l'identifiant de tâche créatrice est effacé afin de pouvoir les détruire plus tard dans l'ipc_ns que l'on quitte). Puis la liste des segments créés est mise à **0** pour la tâche courante. De plus, si le fichier **/proc/sys/kernel/shm_rmid_forced** est différent de **0**, alors les segments créés par la tâche et non encore attachés (c.-à-d. non mappés par au moins une tâche) sont détruits.

```
if (unshare_flags & CLONE_NEWIPC) {
 /* Orphan segments in old ns (see sem above). */
 exit_shm(current);
 shm_init_task(current);
}
```

Ensuite, on met à jour le champ **nsproxy** de la tâche appelante avec le nouveau **nsproxy** alloué, si au moins un des drapeaux **CLONE_NEWXXX** autre que **CLONE_NEWUSER** est passé en paramètre à l'aide de **switch_task_namespaces()**. Cette dernière décrémente le compteur de référence sur le **nsproxy** courant et sa désallocation si le compteur atteint **0** :

```
if (new_nsproxy)
 switch_task_namespaces(current, new_nsproxy);
```

Enfin, si **CLONE_NEWUSER** a été passé, le nouveau user_ns est « attaché » à la tâche courante :

```
  if (new_cred) {
   /* Install the new user namespace */
   commit_creds(new_cred);
   new_cred = NULL;
  }
 }
```

La figure 4 schématise l'opération **unshare(CLONE_NEWUTS)**. Les actions et structures mises en œuvre sont en orange.

On notera pour conclure que dans le cas de la création d'un nouveau pid_ns (drapeau **CLONE_NEWPID**), l'appelant n'est pas associé au nouveau pid_ns. Le champ **pid_ns_for_children** de son **nsproxy** est modifié pour le référencer afin qu'un prochain appel à **clone()** (sans le drapeau **CLONE_NEWPID**) s'appuie dessus pour y associer la nouvelle tâche. Dans un prochain article, nous expliquerons la raison de ce fonctionnement particulier pour le drapeau **CLONE_NEWPID**.

2.5 setns

L'appel système **setns()** est défini dans le fichier **kernel/nsproxy.c**. Le descripteur de fichier **fd** passé en paramètre permet de retrouver la référence sur le descripteur du namespace cible, comme on l'a déjà vu (variable locale **ns**). Si le paramètre **nstype** est différent de **0**, on vérifie qu'il correspond bien au type du namespace lié au descripteur de fichier (sinon, l'erreur **EINVAL** est retournée) :

```
SYSCALL_DEFINE2(setns, int, fd, int, nstype)
{
 struct task_struct *tsk = current;
 struct nsproxy *new_nsproxy;
 struct file *file;
 struct ns_common *ns;
 int err;

 file = proc_ns_fget(fd);
...]
 err = -EINVAL;
 ns = get_proc_ns(file_inode(file));
 if (nstype && (ns->ops->type != nstype))
  goto out;
```

La fonction **create_new_namespaces()** déjà vue lors de l'étude de **clone()** est alors appelée pour allouer une nouvelle structure **nsproxy**. Le premier paramètre « flags » étant à **0**, ses pointeurs vont référencer les mêmes structures que le **nsproxy** de la tâche appelante, mais leur compteur de références est incrémenté.

```
 new_nsproxy = create_new_namespaces(0,
tsk, current_user_ns(), tsk→fs);
```

Puis la fonction **install()** du namespace cible (c.-à-d. correspondant au descripteur de fichier passé en paramètre) est appelée. Cette fonction vérifie la présence de la capacité **CAP_SYS_ADMIN** sous peine de retourner l'erreur **EPERM**, décrémente le compteur de références sur le namespace source (celui dans lequel l'appelant se trouve), effectue quelques actions d'intégrité dans le namespace source, incrémente le compteur de références sur le namespace cible et fait pointer le nouveau **nsproxy** dessus :

```
 err = ns->ops->install(new_nsproxy, ns);
```

Parmi les opérations d'intégrité, on peut citer l'exemple où le namespace cible serait un ipc_ns : le mécanisme **semadj** est déclenché pour l'ipc_ns source avec un appel à **exit_sem()**. Dans le cas où le namespace cible est un user_ns, on retourne en erreur (**EINVAL**) si l'appelant est une tâche d'un processus multithreadé (car tous les threads d'un processus doivent être dans le même user_ns).

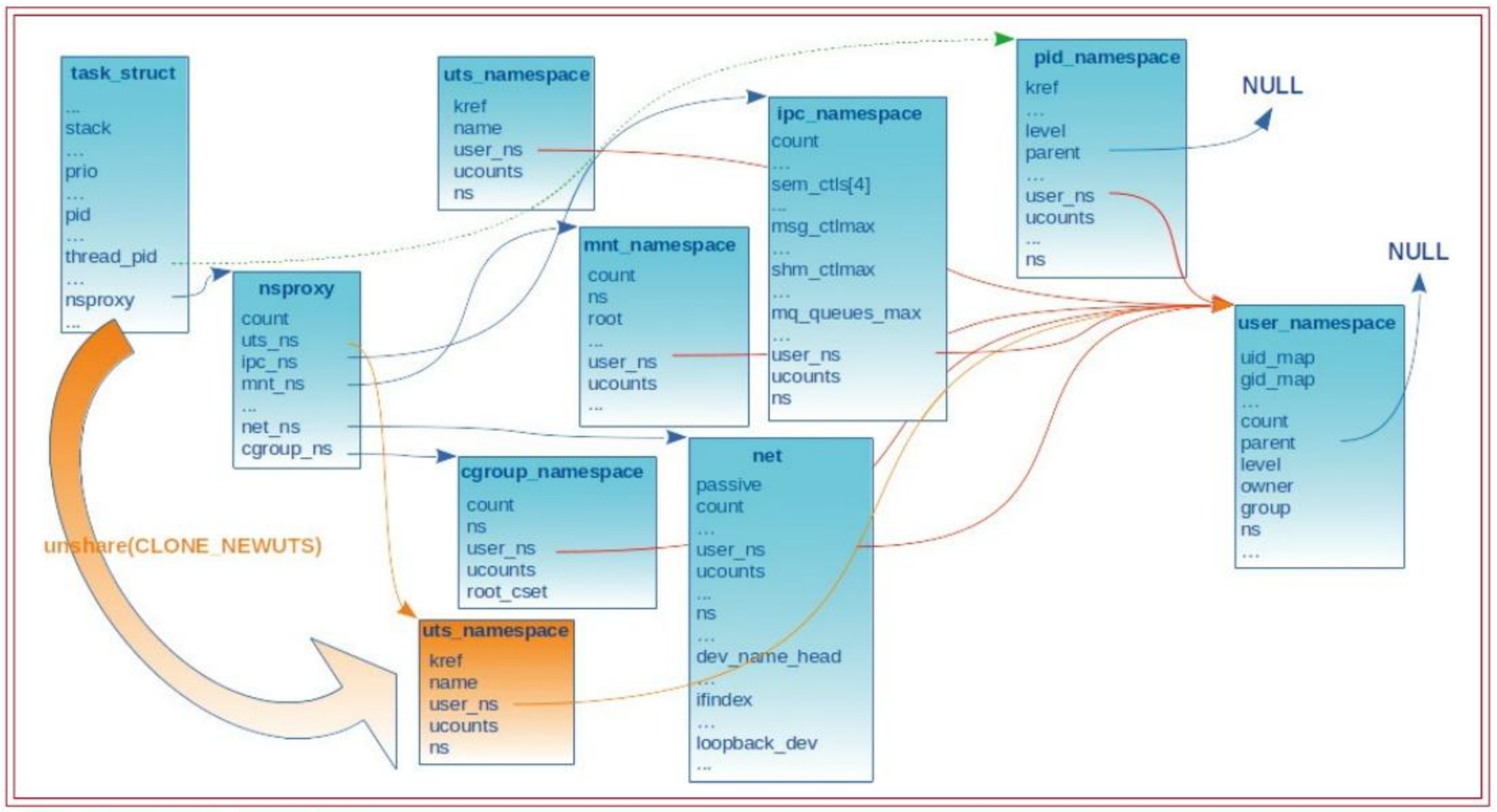


Fig. 4 : Appel système unshare().

Enfin, le champ **nsproxy** de la tâche courante est remplacé pour laisser place à celui que l'on vient de créer avec l'un de ses pointeurs référençant le namespace cible. L'ancien **nsproxy** est libéré si la décrémentation de son compteur de références aboutit à la valeur **0** :

```
switch_task_namespaces(tsk, new_nsproxy);
```

En conclusion, la tâche appelante se retrouve avec un nouveau **nsproxy** identique au précédent, sauf pour le pointeur associé au namespace cible. Cette opération est identique à l'opération **unshare()**, car elle alloue un nouveau **nsproxy**. Mais **unshare()** crée un nouveau namespace, alors que **setns()** fait pointer son **nsproxy** sur un namespace existant.

CONCLUSION

Ainsi s'achève notre plongée dans le noyau, pour observer de près l'implémentation des namespaces. Le but n'était pas de comprendre tous les détails, mais plutôt de donner des clés à qui voudrait approfondir le sujet ou comprendre certaines limitations que nous évoquerons par la suite.

À partir du prochain article, nous repasserons en espace utilisateur pour commencer notre passage en revue de chaque namespace de manière pratique, à l'aide de programmes d'exemples. ■

RÉFÉRENCES

[1] Overview of the Linux Virtual File System : https://www.kernel.org/doc/Documentation/filesystems/vfs.txt

[2] Anatomy of the Linux Virtual File System switch : https://developer.ibm.com/technologies/linux/tutorials/l-virtual-filesystem-switch/

[3] The namespace file system (NSFS) : https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e149ed2b805fefdccf7ccdfc19eca22fdd4514ac

[4] Namespace file descriptors : https://lwn.net/Articles/407495/

[5] J. CORBET, A. RUBINI, G. KROAH-HARTMAN, « Linux Device Drivers (3rd Edition) », O'Reilly, February 2005.

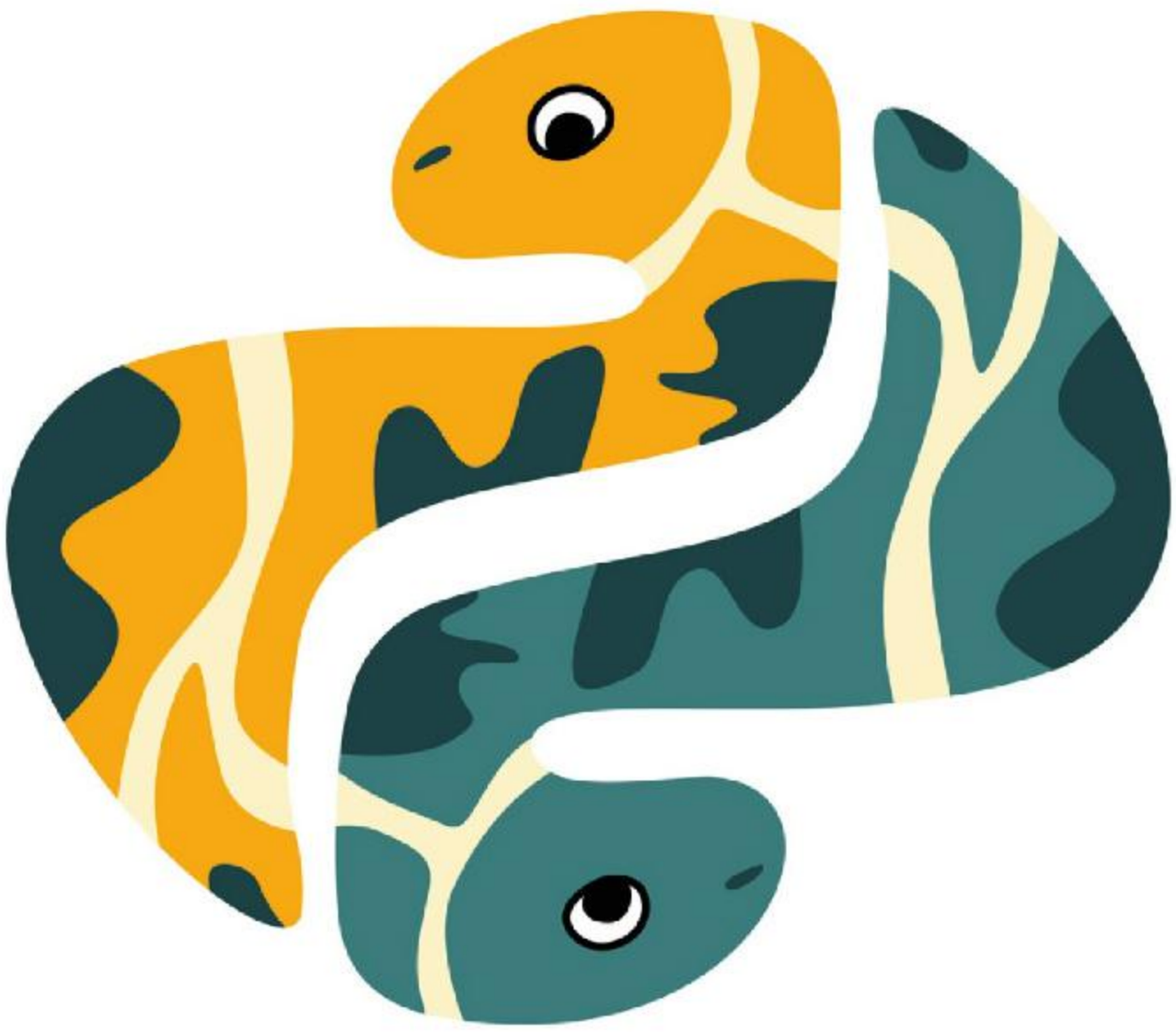
[6] Add an ioctl to get a socket network namespace : https://lore.kernel.org/patchwork/patch/728774/

[7] Linux capabilities support for user namespaces : https://lwn.net/Articles/420624/

ENCODAGE D'UN SCRIPT PYTHON ET EXÉCUTION DU SCRIPT ENCODÉ (ACME::BUFFY STYLE)

TRISTAN COLOMBO

MOTS-CLÉS : ENCODAGE, DÉCODAGE, PYTHON



Il est parfois intéressant de considérer des problèmes anodins, complètement inutiles, mais qui permettent de mettre en œuvre des éléments de programmation encore jamais employés. Je vous propose ici de réécrire en Python un module bien connu des développeurs Perl : Acme::Buffy.

Acme::Buffy est un module **Perl** amusant : il suffit de le charger au début d'un code Perl pour que celui-ci soit encodé sous la forme d'une suite de termes « Buffy », où la casse sera modifiée de manière à offrir la diversité nécessaire pour couvrir l'ensemble des caractères. Un petit test sera sans doute plus parlant, si vous avez déjà installé Perl :

```
$ cpan -i Acme::Buffy
```

Cette ligne permet d'installer le module Acme::Buffy. Nous pouvons ensuite créer un petit programme de test dans **essai.pl** :

```
use Acme::Buffy;

print "Hello world";
```

Au lancement du script, nous obtiendrons l'affichage attendu de « Hello world » :

```
$ perl essai.pl
Hello world
```

Mais une petite surprise vous attend en ouvrant à nouveau le fichier **essai.pl** :

```
use Acme::Buffy;
BUffY bUFFY BUffY bUFFY bUfFy buffy BUFfy        buFFY
BufFy      BufFY   bUFfy BuFFY buffy        bufFy bUffy
bUffY BuFfy BuffY   bUFfy BUfFY BUFFY         Buffy
bUffY      BuFFY BUFFy      BufFy
BUFfy BUfFY buFfy        BufFy bUffY      bUFFy bUfFy bu
```

On peut se dire que le code a été modifié, mais qu'il n'est plus fonctionnel... eh bien non :

```
$ perl essai.pl
Hello world
```

Je vous propose dans cet article d'écrire un module Python ayant le même comportement, à la différence prêt que le fichier encodé ne viendra pas écraser le code source et sera stocké dans un autre fichier, par sécurité.

1. PRINCIPE DE L'ENCODAGE

Je me suis inspiré du module Acme::Buffy, mais je n'ai pas voulu regarder le code source, pour ne pas appliquer nécessairement le même processus.

Supposons donc que nous ayons le mot « hello » à encoder sous la forme d'une suite de termes « Buffy ». **buffy** peut être considéré comme la clé d'encodage qui donnera naissance à des variants **Buffy**, **bUffy**, **buFFy**, etc. Notre clé comportant 5 caractères, il y a 25 combinaisons possibles, ce qui ne sera pas suffisant pour encoder des caractères ASCII. La solution consiste donc à utiliser deux variants pour chaque caractère, ce qui porte le nombre de combinaisons à 625.

Techniquement, on récupérera le code ASCII de chaque caractère, on le convertira en base 25 (ce qui se fera nécessairement sur deux digits, car nous considérerons que les caractères ASCII employés commencent avec l'espace de code **32** (soit **17** en base 25, c.-à-d. $1 \times 25^1 + 7 \times 25^0$), puis les deux digits seront convertis en variants de **buffy**.

NOTE

Seul le caractère de tabulation **\t** de code ASCII **9** pourrait poser problème. Comme tout éditeur de code qui se respecte convertit automatiquement les tabulations en espaces, je n'ai pas traité ce cas, mais si vous le souhaitez, il suffira d'ajouter un test dans le code pour ajouter un digit à **0** de manière à obtenir **09**.

Prenons l'exemple du mot « hello ». La première lettre est un « h » dont le code ASCII, donné par **ord()**, est **104**. Pour convertir ce nombre en base 25, nous devons savoir combien de « dizaines » de 25 il contient et combien d'« unités ». Pour cela, le plus simple est de construire le nombre en le « remontant » depuis les « unités » : le reste de la division entière par 25 nous donne le nombre d'« unités », puis en prenant le résultat de la division entière, nous passons aux « dizaines » et ainsi de suite. En reprenant notre lettre « h » de code **104**, nous obtenons donc :

- $104 \% 25 = 4$: il y a 4 « unités », notre nombre se terminera par **4** (4×25^0) ;
- $104 // 25 = 4$: il y a 4 « dizaines », notre nombre sera 44 ($4 \times 25^1 + 4 \times 25^0$) ;
- $4 \% 4 = 0$: il n'y a pas de « centaines » et le calcul s'arrête.

Nous pouvons écrire une petite fonction pour nous détailler ces calculs automatiquement :

```
def toBase(value : int, base : int) ->
str:
    result = ''
    while value > 0:
        result = str(value % base) +
result
        value = value // base
        print(f'{result=} et {value=}')
    return result
```

Deux choses à noter dans ce code :

- l'utilisation de **str()** pour convertir les digits en chaîne de caractères, de manière à pouvoir les concaténer les uns à la suite des autres sans calcul ;
- l'affichage des valeurs de **result** et **value** à l'aide d'une f-string contenant le formatage **nom_variable=** qui affiche le nom de la variable suivi du caractère égal, puis du contenu de la variable.

Nous pouvons appliquer cette fonction sur le code ASCII des lettres qui composent notre mot dans un Shell Python :

```
>>> toBase(ord('h'), 25)
result='4' et value=4
result='44' et value=0
'44'
>>> toBase(ord('o'), 25)
result='11' et value=4
result='411' et value=0
'411'
```

Pour la lettre « h », nous obtenons le bon résultat. Par contre, pour la lettre « o », nous avons un petit problème : nous obtenons **411**, or ce nombre correspond en base **10** à $4 \times 25^2 + 1 \times 25^1 + 1 \times 25^0$ soit **2526**. L'erreur provient du fait que **11** n'existe pas en base 25 : il faudrait écrire **b**. Le module **string** nous fournit une chaîne de caractères **printable** contenant les caractères affichables dans l'ordre. Ainsi **string.printable[9]** donne « 9 », **string.printable[10]** donne « a », etc.

Corrigeons notre code :

```
import string

def toBase(value : int, base : int) -> str:
    result = ''
    while value > 0:
        result = string.printable[value % base] +
result
        value = value // base
        print(f'{result=} et {value=}')
    return result
```

Cette fois, le résultat est correct :

```
>>> toBase(ord('o'), 25)
result='b' et value=4
result='4b' et value=0
'4b'
```

Une fois que nous avons converti notre lettre en base 25, il faut l'encoder à l'aide de variants de **buffy**. Pour cela, il faut disposer desdits variants...

De manière à être évolutifs, nous allons considérer une variable **TAG** à partir de laquelle nous souhaitons obtenir les variants. C'est à partir de celle-ci que nous pourrons calculer le nombre de combinaisons possibles et donc, la base dans laquelle convertir le code ASCII des lettres :

```
TAG = 'buffy'
base = len(buffy)**2
```

Il nous faut maintenant générer toutes les combinaisons possibles. Le module **itertools** fournit une fonction **product()** qui effectue un produit cartésien des données transmises en paramètre :

```
>>> import itertools
>>> list(itertools.product('ab', 'cd'))
[('a', 'c'), ('a', 'd'), ('b', 'c'),
('b', 'd')]
```

Comme nous voulons les variants de **TAG** comportant des caractères minuscules et majuscules, nous allons calculer la liste de tous les couples de caractères minuscules/majuscules :

```
>>> list(((car.lower(), car.upper())
for car in TAG))
[('b', 'B'), ('u', 'U'), ('f', 'F'),
('f', 'F'), ('y', 'Y')]
```

En transmettant cette liste à **itertools.product()**, nous obtiendrons les combinaisons :

```
>>> list(itertools.product(*((car.
lower(), car.upper()) for car in TAG) ))
[('b', 'u', 'f', 'f', 'y'), ('b', 'u',
'f', 'f', 'Y'), ('b', 'u', 'f', 'F',
'y'), ('b', 'u', 'f', 'F', 'Y'), ('b',
'u', 'F', 'f', 'y'), ('b', 'u', 'F', 'f',
'Y'), ('b', 'u', 'F', 'F', 'y'),
('b', 'u', 'F', 'F', 'Y'), ('b', 'U',
'f', 'f', 'y'), ('b', 'U', 'f', 'f',
'Y'), ('b', 'U', 'f', 'F', 'y'), ('b',
'U', 'f', 'F', 'Y'), ('b', 'U', 'F', 'f',
'y'), ('b', 'U', 'F', 'f', 'Y'), (
'b', 'U', 'F', 'F', 'y'), ('b', 'U', 'F',
'F', 'Y'), ('B', 'u', 'f', 'f', 'y'),
('B', 'u', 'f', 'f', 'Y'), ('B', 'u',
'f', 'F', 'y'), ('B', 'u', 'f', 'F',
'Y'), ('B', 'u', 'F', 'f', 'y'), ('
B', 'u', 'F', 'f', 'Y'), ('B', 'u', 'F',
'F', 'y'), ('B', 'u', 'F', 'F', 'Y'),
('B', 'U', 'f', 'f', 'y'), ('B', 'U',
'f', 'f', 'Y'), ('B', 'U', 'f', 'F',
'y'), ('B', 'U', 'f', 'F', 'Y'), ('B
', 'U', 'F', 'f', 'y'), ('B', 'U', 'F',
'f', 'Y'), ('B', 'U', 'F', 'F', 'y'),
('B', 'U', 'F', 'F', 'Y')]
```

L'utilisation de * s'impose pour que notre expression précédente soit considérée comme **('b', 'B'), ('u', 'U'), ('f', 'F'), ('f', 'F'), ('y', 'Y')**.

Nous voyons toutefois que nous nous approchons du résultat attendu, sans l'atteindre : nous avons une liste de

tuples de caractères et non une liste de chaînes de caractères. Pour corriger cela, **map()** permettra d'appliquer à chaque tuple **''.join()** pour « coller » les caractères entre eux :

```
>>> list(map(''.join, itertools.
product(*((car.lower(), car.upper())
for car in TAG)))
['buffy', 'buffY', 'bufFy', 'bufFY',
'buFfy', 'buFfY', 'buFFy', 'buFFY',
'bUffy', 'bUffY', 'bUfFy', 'bUfFY',
'bUFfy', 'bUFfY', 'bUFFy', 'bUFFY',
'Buffy', 'BuffY', 'BufFy', 'BufFY',
'BuFfy',
'BuFfY', 'BuFFy', 'BuFFY', 'BUffy',
'BUffY', 'BUfFy', 'BUfFY', 'BUFfy',
'BUFfY', 'BUFFy', 'BUFFY']
```

Comme nous allons associer ces termes les uns à la suite des autres, autant leur ajouter un espace à la fin :

```
>>> list(map(lambda s: s + ' ', map(''.
join, itertools.product(*((car.lower(),
car.upper()) for car in TAG)))))
['buffy ', 'buffY ', 'bufFy ', 'bufFY
', 'buFfy ', 'buFfY ', 'buFFy ', 'buFFY
', 'bUffy ', 'bUffY ', 'bUfFy ', 'bUfFY
', 'bUFfy ', 'bUFfY ', 'bUFFy ', 'bUFFY
', 'Buffy ', 'BuffY ', 'BufFy ',
 'BufFY ', 'BuFfy ', 'BuFfY ', 'BuFFy
', 'BuFFY ', 'BUffy ', 'BUffY ', 'BUfFy
', 'BUfFY ', 'BUFfy ', 'BUFfY ', 'BUFFy
', 'BUFFY ']
```

Puis, comme nous voulons obtenir un variant à partir d'un digit, nous allons continuer à travailler la liste obtenue pour la transformer en dictionnaire. La fonction **zip()** va nous donner les associations clé/valeur nécessaires entre **range(base)** et la liste obtenue :

```
>>> dictTAG = dict(zip(range(base),
list(map(lambda s: s + ' ', map(''.
join, itertools.product(*((car.lower(),
car.upper()) for car in TAG)))))))
>>> dictTAG
{0: 'buffy ', 1: 'buffY ', 2: 'bufFy ',
3: 'bufFY ', 4: 'buFfy ', 5: 'buFfY ',
6: 'buFFy ', 7: 'buFFY ', 8: 'bUffy ',
9: 'bUffY ', 10: 'bUfFy ', 11: 'bUfFY
', 12: 'bUFfy ', 13: 'bUFfY ', 14:
 'bUFFy ', 15: 'bUFFY ', 16: 'Buffy ',
17: 'BuffY ', 18: 'BufFy ', 19: 'BufFY
', 20: 'BuFfy ', 21: 'BuFfY ', 22:
'BuFFy ', 23: 'BuFFY ', 24: 'BUffy '}
```

On constate toutefois que les clés ne sont pas correctes : il faut les transformer en chaînes de caractères et à partir de **10**, ce devrait être des lettres. Nous créons donc un nouvel ensemble de clés :

```
newKeys = [str(i) for i in range(10)]
newKeys += [string.printable[i] for i
in range(10, base)]
```

Et nous remplaçons les clés du dictionnaire **dictTAG** :

```
>>> dictTAG = dict(zip(newKeys, dictTAG.values()))
>>> dictTAG
{'0': 'buffy ', '1': 'buffY ', '2': 'bufFy ',
'3': 'bufFY ', '4': 'buFfy ', '5': 'buFfY ', '6':
'buFFy ', '7': 'buFFY ', '8': 'bUffy ', '9':
'bUffY ', 'a': 'bUfFy ', 'b': 'bUfFY ', 'c': 'bUF
fy ', 'd': 'bUFfY ', 'e': 'bUFFy ', 'f': 'bUFFY
', 'g': 'Buffy ', 'h': 'BuffY ', 'i': 'BufFy ',
'j': 'BufFY ', 'k': 'BuFfy ', 'l': 'BuFfY ', 'm':
'BuFFy ', 'n': 'BuFFY ', 'o': 'BUffy '}
```

Nous pouvons maintenant simplement encoder par exemple la lettre « o » dont le code ASCII en base 25 vaut **4b** : il s'agit de **dictTAG['4'] + dictTAG['b']** soit **'buFfy bUfFY '**.

Nous avons décortiqué le principe de l'encodage, nous pouvons maintenant nous lancer dans le code à proprement dit.

2. LE CODE

Pour que notre programme ait un nom semblable au module Perl, il faut le placer dans un répertoire **Acme** et le nommer **Buffy.py**. L'import s'écrira alors :

```
import Acme.Buffy
```

Ensuite, il faut que lors de l'exécution du programme contenant cette ligne, le module soit capable de retrouver le nom du fichier appelant. Cela se fait grâce au module **__main__** :

```
import __main__

print(f'Chemin absolu du fichier appelant :
{__main__.__file__}')
```

Nous avons maintenant toutes les informations nécessaires pour nous lancer dans l'écriture de **Buffy.py** :

```
import __main__
import itertools
import string
import os
import tempfile
from typing import List, Tuple, Dict
```

Les trois premiers modules importés ont été évoqués précédemment pour récupérer le nom du fichier appelant et pour encoder. Le module **os** permettra d'exécuter une commande système, de supprimer un fichier ou encore de manipuler le nom d'un fichier. Le module **tempfile** nous aidera à manipuler un fichier temporaire et les imports du module **typing** sont présents afin de pouvoir correctement annoter les types des paramètres des méthodes, ainsi que les types des valeurs de retour.

```
class Slayer:
    # Tag used for obfuscation
    TAG : str = 'perl'
```

J'ai décidé d'utiliser une architecture orientée objet avec la création d'une classe **Slayer** (puisque l'on parle de Buffy...). L'attribut de classe **TAG** représente la clé d'encodage. Par hommage au module d'origine, je lui ai donné la valeur **'perl'**, mais vous pouvez tout à fait changer cette valeur en **'buffy'** ou autre (nous verrons plus loin pour les restrictions).

```
    def __init__(self):
        self._base = len(Slayer.TAG)**2

        self._dictTAG = dict(zip(range(self._base),
map(lambda s: s + ' ', map(''.join, itertools.product(*((car.
lower(), car.upper()) for car in Slayer.TAG))))))
        newKeys = [str(i) for i in range(10)]
        newKeys += [string.printable[i] for i in range(10,
self._base)]
        self._dictTAG = dict(zip(newKeys, self._dictTAG.
values()))

        self._reverseDictTAG = {}
        for key, value in self._dictTAG.items():
            self._reverseDictTAG[value.rstrip()] = key

        self._filename = __main__.__file__
        with open(self._filename, 'r') as fic:
            self._data = fic.readlines()

        self._mode = None
        self._modeDetection()
```

Le constructeur définit de nombreux attributs. **_base**, **_dictTAG** et **_filename** ont été expliqués précédemment. **_reverseDictTAG** est simplement un dictionnaire où les clés et les valeurs de **_dictTAG** ont été inversées. Ce dictionnaire sera utilisé pour le décodage. **_data** est une liste de chaîne de caractères correspondant aux lignes qui sont lues dans le fichier **_filename**. **_mode** permettra de savoir si le fichier doit être encodé (**'w'** pour write) ou décodé (**'r'** pour read) et exécuté. Cela est déterminé par la méthode **_modeDetection()** qui est appelée en fin de constructeur.

```
    def _modeDetection(self)
-> None:
        for line in self.
_data[1:]:
            if line.lower().
startswith(f"'{Slayer.TAG}"):
                self._mode = 'r'
            elif not line[0].
isspace():
                self._mode = 'w'
```

La première ligne du fichier doit nécessairement contenir l'import d'**Acme.Buffy**, donc nous n'en tenons pas compte (**_data[1:]**). Nous recherchons ensuite une ligne débutant par **'** suivi d'un variant de **Slayer.TAG** (pour rappel, un attribut de classe est préfixé par le nom de la classe lors de l'appel). Si c'est le cas, alors le fichier est déjà encodé et nous sommes en mode **'r'**. Sinon, si la ligne n'est pas vide, alors le fichier n'est pas encore encodé et nous sommes en mode **'w'**.

```
    def _toBase(self, value :
int) -> str:
        result = ''
        while value > 0:
            result = string.
printable[value % self._base]
+ result
            value = value //
self._base
        return result
```

Nous retrouvons dans la méthode **_toBase()** une adaptation de la fonction **toBase()** permettant de convertir un entier dans une base donnée. Ce code a été analysé précédemment.

```
    def _fromBase(self,
value : str) -> int:
        return int(value,
self._base)
```

La méthode **_fromBase()** permet de convertir une valeur donnée dans une base quelconque en entier.

ATTENTION !

La conversion en utilisant **int()** impose une contrainte particulière : la base ne peut pas être supérieure **à 32**. Ainsi, **TAG** ne pourra pas être un mot de plus de 5 caractères.

Si cette limite vous paraît bloquante, il suffit de recoder **_fromBase()** :

```
    def _fromBase(self, value : str)
-> int:
        n = 0
        result = 0
        for car in value[::-1]:
            result += string.
printable.index(car) * self._base**n
            n += 1
        return result
```

Mais attention : cette modification fait qu'en fonction de la longueur de la clé, les caractères pourront être codés sur un seul variant et donc, il en découlera d'autres modifications !

```
    def _obfuscate(self, car : str) ->
str:
        result = ''
        for n in self._toBase(ord(car)):
            result += self._dictTAG[n]
        return result
```

La méthode **_obfuscate()**, comme son nom le laisse deviner, encode le caractère **car** à l'aide de la méthode **_toBase()** et du dictionnaire **_dictTAG** pour obtenir les variants.

```
    def _deobfuscate(self, tag :
Tuple[str]) -> str:
        return chr(self.
_fromBase(self._reverseDictTAG[tag[0]])
* self._base + self._fromBase(self.
_reverseDictTAG[tag[1]]))
```

La méthode **deobfuscate()** décode un couple de variants transmis via le paramètre **tag** et renvoie le caractère correspondant.

```
    def _write(self) -> None:
        output = os.path.splitext(self.
_filename)[0] + f'_{Slayer.TAG}.py'

        with open(output, 'w') as fic:
            for n, line in enumerate(self.
_data):
                if n == 0:
                    fic.write(line)
                else:
                    newline = ''
                    for car in line:
                        if car == '\n':
                            newline += car
                        else:
                            newline += self.
_obfuscate(car)
                    if newline.isspace():
                        fic.write(newline)
                    else:
                        newline = newline.
rstrip()
                        fic.
write(f"'{newline} '\n")
```

La méthode **_write()** est appelée pour encoder un fichier. Elle crée un nouveau fichier en ajoutant un **_** suivi de la clé entre le nom de base du fichier et l'extension (par exemple pour **essai.py** avec **TAG='perl'**, le nouveau fichier sera **essai_perl.py**). Ensuite, la méthode parcourt les données de **_data** et encode chaque caractère, avant de les écrire dans le fichier de sortie. Si la ligne de sortie ne contient que des caractères d'espacement, alors on la copie telle quelle, sinon on l'encadre par des apostrophes (sinon, une erreur de syntaxe sera détectée dans le code généré et il ne sera pas exécutable ; les chaînes de caractères sont considérées comme des commentaires, ce qui résout le problème).

```
    def _read(self) -> None:

        def pairwise(iterable):
            elt = iter(iterable)
            return zip(elt, elt)

        output = ''
        with tempfile.NamedTemporaryFile(mode='w',
delete=False) as fic:
```

```
            output = fic.name
            for n, line in enumerate(self._data):
                if n == 0:
                    next
                else:
                    newline = ''
                    line = line[1:-1].split(' ')
                    for tag in pairwise(line):
                        newline += self.
_deobfuscate(tag)
                    fic.write(newline + '\n')

        os.system(f'python3 {output}')

        os.unlink(output)
```

`_read()` va lire un fichier encodé, le décoder et le stocker dans un fichier temporaire créé à l'aide de `tempfile.NamedTemporaryFile()`. Le fichier temporaire sera ensuite lu pour être exécuté (`os.system()`) et effacé (c'est le paramètre `delete=False` qui rend le fichier persistant).

Au niveau du décodage, il faut préciser que le premier (`'`) et le dernier (`\n`) caractère de chaque ligne non vide sont omis (`line[1:-1]`) et que les tags sont lus deux par deux, grâce à la fonction `pairwise()` qui crée des tuples de deux éléments.

```
    def action(self) -> None:
        if self._mode == 'r':
            self._read()
        else:
            self._write()
```

La méthode `action()` sert seulement d'aiguillage pour appeler `_read()` ou `_write()` en fonction de la valeur de l'attribut `_method`.

```
if __name__ == 'Acme.Buffy':
    s = Slayer()
    s.action()
```

Le code du programme principal est très succinct, puisqu'il suffit de créer une instance de `Slayer` et d'appeler la méthode `action()`.

NOTE

Voici quelques explications sur le fonctionnement de `pairwise()`.

La fonction `iter()` renvoie un itérateur sur l'élément transmis. Voyons sur un exemple ce qui se passe :

```
>>> l = ['A', 'B', 'C', 'D', 'E', 'F',
'G', 'H']
>>> l
['A', 'B', 'C', 'D', 'E', 'F', 'G',
'H']
>>> next(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an
iterator
>>> elt = iter(l)
>>> elt
<list_iterator object at
0x7f36022b3d30>
>>> next(elt)
'A'
>>> next(elt)
'B'
```

Ainsi, une fois qu'un élément de la liste est utilisé, il est « consommé » : on a accès au suivant. Ceci est intéressant pour pouvoir créer des couples, car si nous utilisons `zip()` directement sur la liste `l`, le résultat est loin d'être satisfaisant :

```
>>> list(zip(l, l))
[('A', 'A'), ('B', 'B'), ('C', 'C'),
('D', 'D'), ('E', 'E'), ('F', 'F'),
('G', 'G'), ('H', 'H')]
```

Comme nous utilisons la même liste, l'association se fait entre chaque élément `i` de `l` et nous n'obtenons que des couples `(l[i], l[i])`. Alors qu'en utilisant `iter(l)`, lors de la création d'un couple le premier élément étant « consommé », on passe au suivant et on obtient `(l[i], l[i+1])` :

```
>>> elt = iter(l)
>>> list(zip(elt, elt))
[('A', 'B'), ('C', 'D'), ('E', 'F'),
('G', 'H')]
```

Attention : si vous avez déjà « consommé » des éléments de `l`, la liste de couples ne sera pas complète ! C'est la raison pour laquelle j'ai réinitialisé `elt` (nous avions utilisé `'A'` et `'B'` et le couple `('A', 'B')` aurait donc été manquant).

Pour tester le code, créons un programme simple dans **essai.py** :

```
import Acme.Buffy

a = 2
if a == 2:
    print('Ceci est un test !')
```

Exécutons-le :

```
$ python3 essai.py
Ceci est un test !
```

Nous constatons qu'un nouveau fichier **essai_perl.py** est apparu. Ce dernier contient :

```
import Acme.Buffy

'pERl perL peRl perl peRL PErL peRl
perl peRL peRl '
'pERl PerL pERl pERl peRl perl pERl
perL peRl perl peRL PErL peRL PErL peRl
perl peRL peRl peRL PeRl '
'peRl perl peRl perl peRl perl peRl
perl pERL perl pERL peRl pERl PerL pERl
PERl pERL pErl peRl Perl peRl pERL pErl
peRL pERl pErL pERl peRL pERl PerL peRl
perl pERl pErL pERL peRL pERL pErl peRl
perl pERL pErL pERl PERl peRl perl pERL
pErl pERl pErL pERL peRL pERL pErl peRl
perl peRl perL peRl pERL peRl PerL '
```

Pour autant, il reste exécutable :

```
$ python3 essai_perl.py
Ceci est un test !
```

CONCLUSION

Ainsi s'achève notre petit exercice d'encodage/décodage. Il est bien entendu possible d'améliorer encore les choses avec notamment, comme nous l'avons vu, la possibilité d'augmenter la taille de la clé, mais également la possibilité de modifier l'écriture du fichier de sortie, pour n'utiliser qu'une docstring au lieu de plusieurs chaînes de caractères. Bien d'autres cas sont envisageables et si vous voulez vous amuser avec le code, celui-ci est disponible sur https://github.com/tcolombo/Acme_Buffy. ■

RUST, LE LANGAGE INOXYDABLE !

JEAN-MICHEL ARMAND
[Cofondateur Hybird, Développeur Crème CRM et Djangonaute]

MOTS-CLÉS : RUST, MOZILLA, SYNTAXE

Rust a fêté il y a quelques mois ses dix ans. Huit ans après sa première version alpha, cinq ans après sa première version stable, il était temps de voir ce qu'il était advenu de ce langage qui avait démarré quasiment dans un garage et qui était rapidement devenu l'un des langages les plus intéressants de la décennie.

Débuté comme un projet personnel de Graydon Hoare, **Rust** fut assez rapidement soutenu par la fondation **Mozilla** qui en fit son langage du futur, celui qui devait être utilisé pour le prochain cœur de son navigateur. C'est un langage qui se veut « fiable, concurrent et pratique ». On y retrouve des influences d'**OCaml** (le premier compilateur Rust était d'ailleurs écrit en OCaml) ou de langages orientés concurrence comme le **Erlang**. Être fiable pour tout ce qui concerne la gestion de la mémoire, garantir une absence d'erreurs de concurrence, rester pratique tout en étant aussi rapide que l'ancêtre **C**, voilà le défi que Rust a décidé de relever. Avec une telle promesse, comment ne pas avoir envie de le tester ?

1. PRÉSENTATION ET INSTALLATION

1.1 Présentation

Rust **[1]** n'était donc au démarrage qu'un side project. Très rapidement, il fut intégré dans les projets Mozilla et le développement de **Servo [2]**, le

futur moteur de rendu de **Firefox** fut lancé. Il fallut ensuite attendre cinq ans, en mai 2015, pour voir sortir la première version stable de Rust, la 1.0. Cinq ans plus tard, nous voilà en 2020 et la 1.48 vient tout juste de sortir (cela dépend bien entendu de quand vous allez lire ces lignes, pour être exact, la 1.48 est sortie le 19 novembre 2020). Et un long chemin fut parcouru, Rust n'est plus seulement « le nouveau langage de Mozilla avec lequel Servo est développé ». On trouve du Rust un petit peu partout, dans la programmation système, chez **Dropbox [3]** ou **Sentry**. On peut faire du front web en Rust en utilisant **WebAssembly**, on fait de l'embarqué en Rust ou du jeu vidéo. Et c'est tant mieux, sinon on aurait pu craindre pour le futur du langage, du fait de la restructuration de Mozilla et du licenciement de quasiment toutes les personnes travaillant sur Servo en août 2020 **[4]**.

Mais qu'est-ce qui fait que tellement de gens se sont mis à utiliser Rust ? À réécrire des applications en Rust, des applications déjà en production ? À lancer de nouveaux projets en Rust, à prendre le risque de construire une stack complète avec un langage somme toute relativement jeune ?

C'est tout simple : « rapide, fiable, concurrent et pratique ». On pourrait croire qu'on lit une liste au père Noël. Après tout, un langage à fois rapide, fiable, concurrent et pratique, ce n'est pas loin d'être un langage parfait. Et voilà la réponse du succès de Rust. On pourrait dire qu'il n'est pas loin d'être un langage parfait. Un langage qui garantit l'absence d'erreurs de gestion mémoire ou de concurrence, et cela dès la compilation, avec une rapidité égale à celle du **C** ou du **C++** et sans avoir besoin de mettre en place de garbage collector. Et cerise sur le gâteau, comme il se veut pratique, il propose aussi des abstractions fortes de haut niveau. Parce qu'en 2020, on n'a plus envie de développer comme il y a quarante ans et on aimerait bien pouvoir enfin utiliser des concepts nous facilitant la vie. Vous n'y croyez pas ? Continuez à lire, on ne sait jamais. Mais avant de rentrer dans le vif du sujet, on va tout de même voir comment installer Rust.

1.2 Installation

Rust étant multiplateforme, quel que soit votre système d'exploitation préféré, vous allez pouvoir l'installer. Sous **GNU/Linux**, il vous suffit de lancer la commande suivante :

```
$ curl --proto '=https' --tlsv1.2 -sSf
https://sh.rustup.rs | sh
```

Sous les autres systèmes d'exploitation, il vous faudra aller voir la page dédiée disponible sur https://forge.rust-lang.org/infra/other-installation-methods.html et trouver la version qui vous convient. Rust est mis à jour assez régulièrement : pour ne pas rater les mises à jour, il vous faudra lancer de temps en temps la commande **`rustup update`**. Si malheureusement après quelques tests, vous n'êtes pas conquis par Rust et que vous vouliez le désinstaller, il vous suffira de lancer un **`rustup self uninstall`**.

Mais peut-être que vous n'avez pas envie de tout de suite installer Rust et ses outils. Dans ce cas là, vous pouvez vous amuser à écrire quelques lignes de code en ligne, sur le playground disponible sur https://play.rust-lang.org/.

1.3 Cargo et autres outils

Avant d'écrire notre premier programme, un Hello world! bien entendu, il me faut vous présenter encore deux ou trois outils. Tout d'abord **Cargo**. Cargo est le couteau suisse du développeur Rust. C'est à la fois un outil de build (avec **`cargo build`**), de lancement des tests (**`cargo test`**), de construction de documentation (**`cargo doc`**), de lancement de programme (**`cargo run`**) et de gestion de paquets. Il peut même publier vos bibliothèques sur **Crates.io** (**`cargo publish`**). Et vous l'avez installé en installant Rust, avec la ligne de commande de la section précédente. Autant vous dire qu'un certain nombre de développeurs Python sont plus que jaloux de ce magnifique outil.

Rustfmt [5] et **Clippy [6]** sont deux outils qui sont maintenant inclus dans Rust. Ils seront donc directement utilisables en lançant respectivement :

```
$ cargo fmt
```

Et :

```
$ cargo clippy
```

Si par hasard ils n'étaient pas installés, vous pouvez les installer en lançant les commandes suivantes :

```
$ rustup component add rustfmt
```

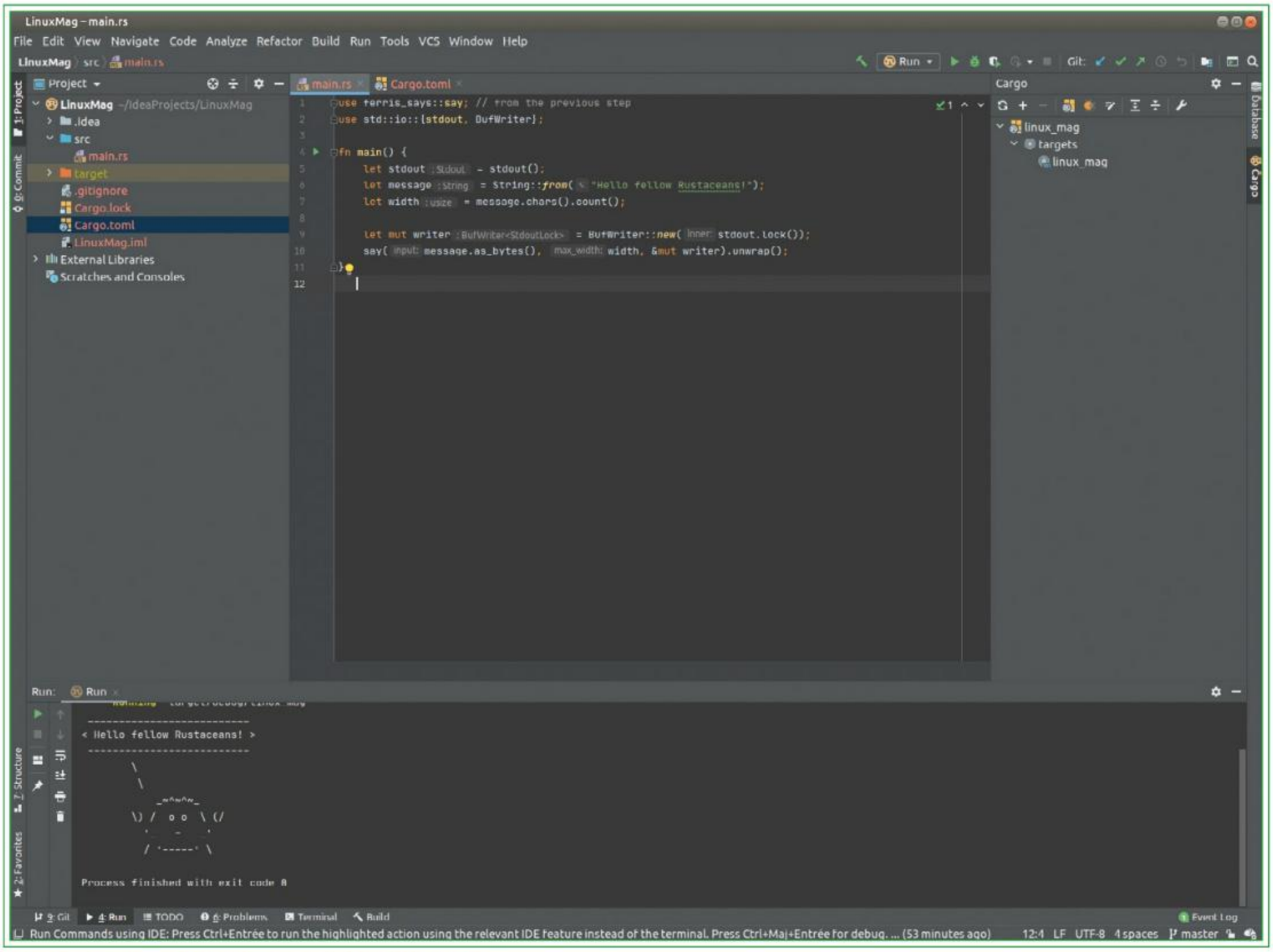


Fig. 1 : Capture d'un IDE Rust, ici IntelliJ IDEA.

Et :

```
$ rustup component add clippy
```

rustmt vous permettra de formater votre code Rust, en suivant les standards de code officiel. Quant à **clippy**, c'est un outil de conseil. Il analysera votre code source pour vous prévenir des erreurs qu'il contient, sans avoir à lancer une compilation ou vous proposer des manières plus idiomatiques d'écrire les choses.

Enfin, un point important, l'éditeur de code. Quel outil utiliser pour écrire votre code Rust ? Vous avez sûrement déjà un éditeur de code préféré, avec un peu de chance, il aura un plugin permettant de prendre en charge le Rust. Si ce n'est pas le cas, **Sublime Text**, **Atom**, **Eclipse**, **IntelliJ IDEA** et **VScode** proposent des plugins pour Rust. IntelliJ vous affichera automatiquement les types de data (voir capture d'écran en figure 1) ou vous indiquera dans la mesure du possible le code qui ne passera pas à la compilation.

1.4 Premier programme, Hello word et crustacé

Comme le veut la tradition, nous allons commencer par écrire un « Hello world! ». Enfin, « écrire », c'est un bien grand mot, nous allons le générer. Et nous allons utiliser **cargo**. En effet **cargo** va nous générer un programme d'exemple avec la bonne structure de fichier. Pour cela, lançons la commande :

```
$ cargo new hello-linux-mag
```

Cette commande va vous créer l'arborescence suivante :

```
hello-linux-mag
|- Cargo.toml
|- src
  |- main.rs
```

Cargo.toml est le fichier manifeste de votre projet Rust. Il vous permet de configurer les métadonnées de votre

projet (version, nom, auteurs, etc.) ainsi que vos dépendances. Le fichier **main.rs** quant à lui et le fichier principal et pour l'instant unique de votre projet.

Son contenu est très simple :

```
fn main() {
    println!("Hello, world!");
}
```

Il nous faut maintenant le compiler et le lancer. Pour cela, encore une fois, utilisons **cargo** :

```
$ cargo run
```

Vous devriez alors voir la sortie suivante :

```
$ cargo run
   Compiling hello-linux-mag v0.1.0 (/home/
jmad/IdeaProjects/hello-linux-mag)
    Finished dev [unoptimized + debuginfo]
target(s) in 0.29s
     Running `target/debug/hello-linux-mag`
Hello, world!
```

Cargo a compilé votre premier programme, puis en a lancé l'exécution.

Un deuxième exemple, tiré directement du get-started Rust **[7]**. Tout d'abord, nous allons ajouter une dépendance dans notre projet. Pour cela, ouvrez le fichier **Cargo.toml** et sous l'en-tête de section des dépendances, ajoutez la ligne suivante :

```
ferris-says = "0.2"
```

Ensuite, remplacez le contenu de votre fichier **main.rs** par celui-ci :

```
use ferris_says::say; // from the previous
step
use std::io::{stdout, BufWriter};

fn main() {
    let stdout = stdout();
    let message = String::from("Hello
fellow Rustaceans!");
    let width = message.chars().count();

    let mut writer =
BufWriter::new(stdout.lock());
    say(message.as_bytes(), width, &mut
writer).unwrap();
}
```

Nous avons ajouté une dépendance, commençons donc par lancer **cargo build** pour l'installer et ensuite lançons notre programme :

```
$ cargo run
    Finished dev [unoptimized + debuginfo]
target(s) in 0.01s
     Running `target/debug/linux_mag`
 _____________________________
< Hello fellow Rustaceans! >
 -----------------------------
        \
         \
            ~^~^~
        \) /  o o  \ (/
          '      _     '
           \  '-----'  /
```

Et voilà, **Ferris** vous dit bonjour ! Le code de ce petit programme d'exemple peut vous sembler, pour l'instant, un peu obscur. Mais au fur et à mesure de la lecture de cet article, les choses deviendront limpides.

Ferris le crabe est la mascotte non officielle de la communauté Rust (figure 2). Un certain nombre de développeurs Rust se nomment eux-mêmes Rustaceans, un jeu de mots avec crustacean. Pour le nom du crabe, c'est un jeu de mots avec ferrous (ferreux) qui provient de l'idée que le fer rouille (rust en anglais)...

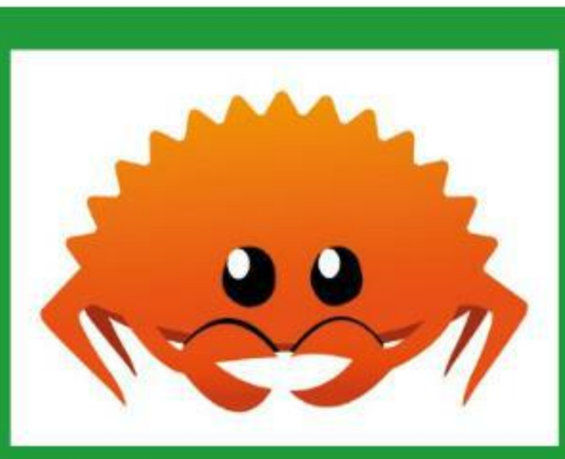


Fig. 2 : Ferris le crabe.

2. CONCEPTS DE BASE

2.1 Variables, immutabilité et types

Prenons un premier exemple de code dans **exemple.rs** :

```
01: fn main() {
02:     let x = 12;
03:     println!("La valeur de X est: {}", x);
04:     x = 33;
05:     println!("La valeur de X est: {}", x);
06: }
```

Rien de bien sorcier, cela ressemble même sûrement à du code que vous avez déjà écrit. Ligne 2, on déclare une variable à l'aide du mot clé **let**, on affiche sa valeur avec **println!**, puis on modifie la valeur de la variable et on affiche à nouveau la valeur. Et pourtant, cela ne compile pas avec **rustc exemple.rs**. Le compilateur va vous générer un long message d'erreur et finira par la conclusion suivante :

```
error[E0384]: cannot assign twice to
immutable variable `x`
```

LE COMPILATEUR RUSTC, UN GENTIL BOYSCOUT

Rust vous certifie que si votre programme compile, alors il n'y aura pas de segmentation fault, de fuite mémoire ou de problème de concurrence. C'est une sacrée responsabilité pour le compilateur de vous garantir cela. Alors, il ne prend aucun risque. Et il ne vous laissera rien passer. On dit souvent que le plus long quand on écrit un programme Rust, ce n'est pas de l'écrire, c'est d'arriver à ce qu'il compile. Mais bien que sévère, le compilateur rustc n'est pas cruel. Il fera tout pour vous aider. Les messages sont très détaillés, ils expliquent bien le pourquoi de l'erreur, proposent parfois des solutions ou vous donnent des pistes vers de la documentation à lire pour comprendre et trouver une correction.

Eh oui, en Rust par défaut, les variables sont immutables. Une fois déclarées, il n'est plus possible d'en modifier la valeur. Enfin, sauf si vous utilisez le mot clé **mut** pour définir une variable comme mutable. Modifions donc très légèrement le code précédent pour le faire fonctionner et remplaçons la ligne 2 par :

```
    let mut x = 12;
```

Même si les variables sont immutables, on peut les redéclarer plusieurs fois, c'est ce qu'on appelle le **shadowing**, les premières déclarations étant cachées par les suivantes. Le morceau de code suivant va donc bien compiler et afficher **12**, puis **33**.

```
fn main() {
    let x = 12;
    println!("La valeur de X est: {}", x);
    let x = 33;
    println!("La valeur de X est: {}", x);
}
```

Enfin, Rust est un langage statiquement typé. On peut en redéclarant une variable lui changer son type. Le code suivant fonctionnera donc bien :

```
let spaces = "   ";
let spaces = spaces.len();
```

Par contre, une variable, même mutable, ne pourra pas changer de type, et le code suivant échouera à la compilation :

```
let mut spaces = "   ";
spaces = spaces.len();
```

Penchons-nous un peu plus sur cette notion de type. En Rust, chaque valeur a un type. Jusqu'à présent, il n'y a pas dans les exemples de types explicitement définis, parce que Rust va tenter autant que possible d'inférer les types. Il est toutefois possible et parfois absolument nécessaire d'ajouter des annotations de type à notre code pour indiquer clairement les choses au compilateur. Les annotations de type s'écrivent en collant un **:** au nom de notre variable, puis en laissant un espace et en écrivant le nom du type. Par exemple :

```
let x: i32 = 42;
```

Ou encore :

```
let x: i32 = "I am not a number!";
```

Cette seconde déclaration ne passera pas à la compilation. On indique en effet au compilateur que l'on veut une variable de type entier et on lui assigne une chaîne de caractères.

De même, la ligne suivante ne compilera pas :

```
let guess= "42".parse().expect("Not a
number!");
```

En effet le compilateur ici n'est pas capable d'inférer la valeur que doit avoir **guess**. Il vous indiquera donc qu'il manque une annotation de type. Si on rajoute l'annotation de type comme ci-après, alors le compilateur ne dira plus rien :

```
let guess : u32 = "42".parse().
expect("Not a number!");
```

Avant de continuer, une dernière remarque. Vous avez sûrement remarqué l'utilisation de la méthode **expect**. Un grand nombre de fonctions de la librairie standard renvoie un objet de type **Result**. Un objet **Result** est un enum pouvant prendre deux valeurs **Ok** ou **Err**. La méthode **expect** permet de gérer ce retour. Si le retour est **Ok**, elle va simplement renvoyer la valeur retour réelle contenue dans l'objet **Result**, sinon elle affichera le message d'erreur que vous lui avez donné en paramètre. Ne pas utiliser **expect** n'empêchera pas la compilation d'aboutir, mais cela donnera lieu à un warning.

Concernant les types de bases, Rust gère les types classiques tels que les entiers, les flottants, les booléens ou les caractères. Concernant les types composés, il gère les tuples qui peuvent être composés d'éléments de types différents et les tableaux (**Array**) qui sont composés d'éléments de même type. On accède à un élément d'un tuple en utilisant un **.** directement suivi de l'index de la valeur que l'on veut récupérer. Pour récupérer une valeur dans un tableau, on utilise les index de l'élément entre crochets. Voici quelques exemples concernant **tuples** et **Array**.

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);
    let first_element = x.0;
    let last_element = x.2;

    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

2.2 Définition de fonction

En Rust, les fonctions se définissent par le mot clé **fn** que vous avez déjà vu dans nos précédents exemples lorsque nous définissions la fonction **main**. Une fonction doit avoir un nom, un ensemble de paramètres qui peuvent être vides, un corps et, mais ce n'est pas obligatoire, un type de retour. Commençons par définir une fonction simple, sans retour :

```
01: fn main() {
02:     simple_function(12, 22);
03: }
04:
05: fn simple_function(x: i32, y: i32) {
06:     println!("The value of x is: {}", x);
07:     println!("The value of y is: {}", y);
08: }
```

Ici, on déclare notre fonction de la ligne 5 à la ligne 8. On voit tout de suite que le compilateur ne tient pas compte de l'ordre de définition des fonctions. Notre fonction est définie après la fonction **main** et on peut tout de même l'utiliser à l'intérieur de celle-ci. On voit que l'on a ici déclaré le type des arguments. C'est une obligation. Si vous ne le faites pas, la compilation ne passera pas.

NOTE

En Rust, une instruction ne peut être qu'une expression ou une déclaration (statement). Rust est essentiellement composé d'expressions. Une expression renvoie une valeur alors qu'une déclaration n'en renvoie pas. Pour être tout à fait précis, une déclaration renvoie **()** qui correspond à un tuple vide, une valeur vide, un peu comme le **void** du C. Pour transformer une expression en déclaration, il suffit de la terminer avec un **;**.

Pour pouvoir permettre à une fonction de retourner une valeur, il faut déclarer le type de la valeur à l'aide de l'opérateur **->**. La valeur pourra être retournée par le mot clé **return**, mais la plupart du temps, on se contente de faire terminer la fonction par une expression. Et c'est la valeur évaluée de cette dernière expression qui sera implicitement utilisée comme valeur de retour.

```
01: fn main() {
02:     println!("result: {}", addition_with_
return(12, 22));
03:     println!("result: {}", addition(11, 11));
04: }
05:
06: fn addition_with_return(x: i32, y: i32) -> i32{
07:     return x + y
08: }
09:
10: fn addition(x: i32, y: i32) -> i32{
11:     x + y
12: }
```

Lignes 6 et 7, on définit une fonction retournant explicitement un résultat et lignes 10 et 11, on définit la même fonction, mais en utilisant simplement une expression. Si on avait fini la ligne 11 par un **;** on aurait alors transformé notre expression en déclaration et le compilateur nous aurait indiqué une erreur, en expliquant qu'il attendait une valeur entière et qu'il recevait la valeur vide **()**.

2.3 Mécanisme de contrôle de flux

Rust propose du branchement conditionnel classique ainsi que plusieurs mécanismes de boucle.

2.3.1 Branchements conditionnels

Rust met à votre disposition un mécanisme de branchement conditionnel utilisant les mots clés **if** et **else**. Son utilisation est très similaire à ce que l'on peut trouver dans d'autres langages. En voici deux exemples simples :

```
01: if answer == 42 {
02:     println!(" Thanks for all the fish");
03: } else {
04:     println!(" Doctor !");
05: }
06:
07: let x = 1;
08: let y = if x == 1 { 3 } else { 5 };
```

De la ligne 1 à la ligne 5, on a une utilisation habituelle. Ligne 8, on utilise une notation monoligne directement au sein d'une affectation.

À noter qu'il est aussi possible d'utiliser une structure de type **if else** avec de multiple **else if** avant de finir par un **else**.

2.3.2 Mécanismes de boucle.

Rust propose trois mécanismes de boucle. On va tout d'abord trouver le **while** classique qui boucle jusqu'à ce qu'une condition se vérifie. On va avoir ensuite le **for** qui va lui parcourir un **iterator**. La forme la plus simple d'un **for** étant celle qui parcourt un intervalle entre deux nombres, mais on n'est pas limité à cela. Enfin, on va avoir une version simplifiée du **while**, c'est la boucle **loop** qui tourne de manière infinie. Pour pouvoir en sortir, il faudra donc utiliser l'instruction **break**.

Voyons deux premiers exemples :

```
01: fn test_finish(answer: i32) -> bool {
02:     if answer >= 3 {
03:         true
04:     }
05:     else{
06:         false
07:     }
08: }
09: fn main() {
10:     let mut x = 0; // mut x: i32
11:     let mut done = false; // mut done:
bool
12:
13:     while !done {
14:         x += 1;
15:         println!("{}", x);
16:         if test_finish(x) {
17:             done = true;
18:         }
19:     }
20:     x = 0 ;
21:     loop {
22:         x += 1;
23:         println!("{}", x);
24:         if test_finish(x) {
25:             break;
26:         }
27:     }
28: }
```

Entre la ligne 13 et 19, on voit l'utilisation du **while <condition>**, la condition étant ici **tant que la valeur de done est fausse**. Ligne 20 à 27, on met en place la même boucle, mais cette fois avec un **loop** et un test dans le corps de la boucle pour déclencher si nécessaire la sortie grâce à l'appel à **break**.

Il nous manque encore à voir la mise en place d'une boucle avec **for**, la voici donc :

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}",
element);
    }
}
```

On commence par définir un tableau (un type **Array** en Rust), puis on en parcourt simplement toutes les valeurs dans la boucle **for** en utilisant la méthode **iter()**.

2.3.3 Pattern Matching

Le pattern matching est un concept aussi plaisant que puissant à utiliser. C'est une des pierres angulaires du OCaml, il n'est donc pas inattendu de retrouver cette notion dans Rust. Le pattern matching de Rust et d'OCaml est d'ailleurs très similaire. Le concept est simple. Il va

consister à prendre la valeur d'une variable et à tester si cette valeur valide différents patterns. Dès qu'on trouvera un patron qui correspond à notre valeur, on exécutera l'action qui est reliée au patron sélectionné. Étudions un exemple de code pour mieux comprendre :

```
01: fn main() {
02:     let i = 1;
03:     match i {
04:         x @ 10 ..= 100 => println!("
{} est entre 10 et 101", x),
05:         11 | 12  => println!("{} est
égal à 11 ou 12", i),
06:          _ => println!("{} ne respecte
pas les conditions précédentes", i),
07:     };
08: }
```

Ligne 3, on démarre le pattern matching avec le mot clé **match** suivi de la variable que l'on va vouloir utiliser (ici **i**). Ensuite, on déclare les conditions. Ligne 4, on teste sur un intervalle de valeurs (on n'est pas limité à un intervalle numérique, on pourrait aussi utiliser un intervalle de lettres). On peut assigner la valeur que l'on teste en utilisant un **@**. On le fait d'ailleurs ligne 4 en assignant notre valeur **i** à la variable **x**. Ce qui ne sert pas à grand-chose, à part à expliciter l'utilisation du **@**. Mais dans le cas où vous testez une sous-partie d'une structure complexe, cela peut avoir un intérêt. Ligne 5, on fait un test sur deux valeurs avec un **|** qui signifie « ou ». Enfin ligne 6, on utilise le caractère **_** qui permet de dire « pour tous les autres choix possibles ».

Là encore, le compilateur va vérifier les choses. Si par exemple, je remplace la ligne 4 par :

```
        x @ 50 ..= 100 => println!(" {}
est entre 50 et 101", x),
```

Mon programme ne compilera plus. En effet, le compilateur va détecter que la deuxième condition, celle de la ligne 5 n'est jamais atteignable et va donc me demander de corriger les choses.

2.4 Ownership et borrowing

2.4.1 Ownership

La notion d'**ownership** est, associée à la notion de **borrowing**, la notion fondamentale du Rust. Pour bien comprendre ce que cela implique, il faut avoir en tête les trois règles suivantes :

- toute valeur dans Rust est possédée par une variable. La variable est alors appelée **owner** ;
- à chaque instant, il ne peut y avoir qu'un seul owner par variable ;
- si un owner sort du **scope**, la valeur qui lui était rattachée disparaît.

La notion de scope ici est relativement similaire à celle des autres langages, cela peut définir un corps de fonction ou un bloc de code délimité par des accolades.

Pour mettre en lumière cette notion d'ownership, nous allons devoir utiliser des types de données plus complexes que ceux que nous avons vus jusque là et utiliser des **String**. Regardons l'exemple suivant :

```
01: fn main() {
02:     let x = 5;
03:     let y = x;
04:     println!("{}", y);
05:
06:     let s1 = String::from("Découvrons
l'ownership");
07:     let s2 = s1;
08:     println!("{} !", s1);
09: }
```

NOTE

Un langage a deux façons de stocker des données. Il peut décider de les stocker sur la **Stack** (pile) ou sur le **Heap** (tas). La Stack (pile ou pour être exact, pile d'exécution) est la structure de données utilisée pour stocker les adresses de retour des fonctions appelées, mais aussi leurs arguments et leurs variables locales. Les données stockées dans la Stack ne sont donc accessibles que localement et la taille globale de celle-ci est limitée. Enfin, la taille d'une donnée ne peut varier sur la pile. On ne peut donc y stocker des données de tailles variables, comme des chaînes de caractères non fixes ou des listes. Par contre, son utilisation est très rapide. Le tas n'est pas soumis à ces limitations. En effet, le Heap est d'une taille quasi illimitée et accessible de manière globale par tout le programme. On alloue de la mémoire, on utilise un mécanisme de pointeur pour retrouver nos informations et tout se passe sans problème. Mais les opérations sur le tas sont plus longues. Elles peuvent même devenir relativement lourdes si par exemple, on veut agrandir une zone mémoire allouée et que cela nécessite une nouvelle allocation complète, plus une recopie.

DÉCOUVREZ LA NOUVELLE VERSION !

CONNECT

LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT

LES 6 NOUVEAUTÉS :

1. **Nouvelle ergonomie**
2. **Articles premiums jamais publiés**
3. **Parcours thématiques de la rédaction**
4. **Listes de lecture personnalisées**
5. **Création d'alertes de publication**
6. **Moteur de recherche amélioré**

ET TOUJOURS...

- ✓ **Plus de 2580 articles disponibles**
- ✓ **Plus de 130 nouveaux articles chaque année**
- ✓ **Magazines et hors-séries d'hier et d'aujourd'hui**
- ✓ **Accès illimité 24h/24 - 7j/7**
- ✓ **Abonnements multi-lecteurs**

CONTACTEZ-NOUS :

par téléphone : **03 67 10 00 28**

par e-mail : **connect@ed-diamond.com**

connect.ed-diamond.com

LINUX MAGAZINE / FRANCE

Lecture en avant-première des magazines en ligne !

LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT

OUI, JE M'ABONNE

Abonnement à retourner avec votre règlement à :
Les Éditions Diamond Service des Abonnements 10 Place de la Cathédrale, 68000 Colmar, France

Réf. : LM+3	Réf. : LM+3/5
☐ **1 LECTEUR : 369 € HT* / AN** soit : 30,75 € HT / MOIS / LECTEUR	☐ **5 LECTEURS : 479 € HT* / AN** soit : 7,98 € HT / MOIS / LECTEUR

(France) TVA 20% = ...

Total = ...

* Tarifs France Métro.

Merci de consulter les tarifs hors France Métro. sur : www.ed-diamond.com

Société : ..... Nom : ..... Prénom : .....

Adresse : .....

Code Postal : ..... Ville : .....

Pays : ..... Téléphone : .....

Veuillez indiquer svp l'adresse e-mail du référent : .....

Veuillez adresser les règlements par **chèque bancaire** ou **postal** à l'ordre des Éditions Diamond (uniquement France et DOM TOM). Pour les règlements par **virement**, merci de nous contacter **par e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20**

Ici, la compilation va échouer à la ligne 8. Le compilateur nous informera que :

```
error[E0382]: borrow of moved value: `s1`
```

Tout d'abord, commençons par regarder les premières lignes du programme. Ligne 2, on crée un entier **x** et ligne 3, on copie la valeur de l'entier **x** pour avoir un deuxième entier **y**. Les entiers étant des valeurs simples, de taille fixe, Rust fait automatiquement une copie.

Par contre, ce n'est pas la même chose pour les **String**. Ligne 6, on crée une première chaîne de caractères et ligne 7, on ne crée pas une copie complète de la chaîne (ce qui pourrait être une opération lourde) où on ne crée pas un deuxième pointeur vers la chaîne de caractères, ce qui pourrait engendrer des problèmes de mémoire (ce que le compilateur s'engage à ne pas avoir). Non, on déplace (ce qu'on appelle un move en Rust) le ownership de la chaîne de caractères de **s1** vers **s2**. Et on « détruit » donc **s1**, mais sans détruire la valeur (donc la **String**) qui est maintenant possédée par **s2**.

Prenons un autre exemple :

```
01: fn str_owner(s: String){
02:     println!("{} !!!", s);
03: }
04:
05: fn main() {
06:     let s1 = String::from("Découvrons
l'ownership");
07:     str_owner(s1);
08:     println!("{} !", s1);
09: }
```

Là encore, le compilateur renvoie une erreur. La fonction **str_owner** a pris la propriété de la chaîne de caractères quand celle-ci a été passée en argument à la ligne 7 et donc, à la ligne 8, **s1** ne possède plus la valeur qui a été déplacée. Pour avoir un code fonctionnel, il faudrait écrire quelque chose comme cela :

```
fn str_owner(s: String) -> String {
    println!("{} !!!", s);
    s
}

fn main() {
    let s1 = String::from("Découvrons
l'ownership");
    let s1 = str_owner(s1);
    println!("{} !", s1);
}
```

Il est clair qu'à cet instant, on peut se demander s'il est vraiment viable de développer des programmes en Rust, s'il faut à chaque appel de fonction renvoyer les arguments en résultat pour récupérer leur ownership au niveau de l'appelant. Heureusement, ce n'est pas le cas. Parce que c'est ici qu'intervient la notion de borrowing et de references.

2.4.2 Borrowing et References

Borrowing peut se traduire en français comme emprunt ou prêt. Et c'est exactement cela : on va pouvoir prêter une valeur à une variable sans lui en transférer la propriété, le ownership. Et on va faire cela en utilisant une référence à un objet. Reprenons notre exemple avec notre fonction et son argument string et utilisons une référence. Le code devient le suivant :

```
01: fn str_owner(s: &String){
02:     println!("{} !!!", s);
03: }
04:
05: fn main() {
06:     let s1 = String::from("Découvrons
l'ownership");
07:     str_owner(&s1);
08:     println!("{} !", s1);
09: }
```

Ligne 1, on change la signature de la fonction pour déclarer que notre paramètre sera une référence en utilisant le caractère **&**. Ligne 7, lors de l'appel de la fonction, on ajoute également le **&** pour indiquer au compilateur qu'on veut prêter l'ownership. Cet exemple montre une utilisation des références en lecture seule. Si on veut pouvoir utiliser une référence en écriture, il faut ajouter le mot clé **mut**. En voilà un exemple :

```
01: fn main() {
02:     let mut s1 =
String::from("Découvrons l'ownership");
03:     str_change(&mut s1);
04:     println!("{} !", s1);
05: }
06:
07: fn str_change(s: &mut String){
08:     s.push_str(", reference en
écriture !");
09: }
```

Tout d'abord, pour pouvoir modifier une variable, il faut la déclarer comme étant mutable ; on le fait ligne 2 en ajoutant le mot clé **mut**. On va devoir ajouter le même mot

clé dans la définition de la fonction ligne 7 et dans l'appel à celle-ci ligne 3. Une petite chose supplémentaire, le code suivant ne fonctionnera pas :

```
fn str_change(s: &mut
String) -> String {
    s.push_str(", reference
en écriture !");
    s
}
```

En effet, `s` n'est pas de type `String`, mais de type `&mut String`. L'utilisation des références permet d'assouplir suffisamment les notions d'ownership pour pouvoir imaginer créer de vrais programmes en Rust. Toutefois, pour que le compilateur puisse continuer à garantir qu'il n'y aura pas de problèmes de mémoire une fois la compilation réussie, il y a quelques règles à respecter concernant le borrowing. Tout d'abord concernant le nombre de références : on peut avoir, sur une même variable, autant de références simultanées en lecture que l'on veut. Mais concernant les références mutable, on ne peut en avoir qu'une par variable. Et enfin, pour une même variable, on ne peut à la fois avoir des références en lecture et en écriture. Pourquoi avoir défini ces règles ? Un problème mémoire, une data race peut arriver quand trois conditions sont réunies :

- deux pointeurs ou plus ont accès à la même donnée au même moment ;
- au moins un de ces pointeurs à un accès en écriture ;
- il n'y a pas de mécanismes de synchronisation d'accès mis en place.

En mettant en place les règles que l'on vient de voir sur les références, Rust s'assure que les trois conditions nécessaires pour risquer des data races ne peuvent être présentes au même moment et donc peut s'engager sur l'absence de data race après une compilation réussie. Pour finir, deux exemples de code mettant en place des références qui ne passent pas la compilation. Premier exemple :

```
01: let mut s = String::from("hello");
02: let r1 = &s;
03: let r2 = &s;
04: let r3 = &mut s; // Référence lecture et écriture en même
temps ERREUR !
05: println!("{}, {}, and {}", r1, r2, r3););
```

Deuxième exemple :

```
01: let mut s = String::from("hello");
02: let r1 = &mut s;
03: let r2 = &mut s; // Deux références en écriture en même
temps ERREUR !
04: println!("{}, {}", r1, r2);
```

Et deux exemples de morceaux de code qui eux vont passer la compilation. Premier exemple :

```
01: let mut s = String::from("hello");
02: {
03:     let r1 = &mut s;
04: } // r1 sort du scope courant, elle est donc détruite.
on peut créer une nouvelle référence
05: let r2 = &mut s;
```

Deuxième exemple :

```
01: let mut s = String::from("hello");
02:
03: let r1 = &s;
04: let r2 = &s;
05: println!("{} and {}", r1, r2);
06: // r1 et r2 sortent du scope au retour de println!
et sont donc détruites
07:
08: let r3 = &mut s; // no problem
09: println!("{}", r3);
```

3. TYPES COMPLEXES OU COMPOSÉS

3.1 Les Collections

Rust propose diverses collections pour stocker des données, dont les **vecteurs** qui permettent de stocker des listes de valeurs, les `Strings` qui servent à gérer les chaînes de caractères UTF-8 et les **Hash-maps**.

Un vecteur est une liste d'objets de même type. Son type Rust est `Vect<T>`. Pour en créer un, on peut utiliser son constructeur `Vec::new` et utiliser la ligne de code suivante :

```
let mut v: Vec<i32> = Vec::new();
```

Comme on ne peut stocker que des valeurs de même type dans un vecteur et que le compilateur doit connaître le type, on a donc mis une annotation de type. Une fois notre vecteur créé, on pourrait lui ajouter des valeurs avec la méthode **push**. Mais il y a une méthode plus « rustique » pour créer un vecteur, c'est d'utiliser une macro. Les macros, vous en avez déjà vu durant votre lecture, ce sont les appels qui se terminent par un **!**. **println!** que l'on a utilisé pour afficher des choses est donc une macro. Et pour créer un vecteur avec une macro, il suffit de taper :

```
let mut v: vec![1, 2, 3];
```

Ici, plus besoin de l'annotation de type, vu que l'on commence à remplir directement notre vecteur, le compilateur va pouvoir inférer les choses. Nous savons créer des vecteurs et y ajouter des valeurs, il ne nous manque plus que de savoir lire les valeurs qu'ils contiennent. On peut le faire de deux façons, les voici :

```
01: fn main() {
02:
03:     let v = vec![1, 2, 3];
04:     let two = &v[2];
05:     let third = v.get(3);
06:     match third{
07:         Some(real_value) =>
print!("{}", real_value),
08:         None => print!("No
value"),
09:     }
10: }
```

Ligne 4, on va utiliser la même notation qu'avec les **Array**, à savoir la notation utilisant **[]** et l'index. Cette notation a un désavantage : si l'index est plus grand que la taille du vecteur, elle va renvoyer une erreur. Utiliser la méthode **get()** est l'autre manière pour récupérer un élément dans un vecteur. Toutefois, on ne récupère pas directement un élément, mais un objet de type **Option<&T>**. **Option** est un enum qui contient soit un attribut **Some(T)**, ici **Some(i32)**, soit **None**. Cela permet ensuite de gérer facilement les erreurs avec par exemple du pattern matching, comme entre les lignes 6 et 9.

En ce qui concerne les Strings, on en a déjà vu un certain nombre dans les pages précédentes. Elles sont encodées en UTF-8 et peuvent être créées à partir de chaînes littérales de différentes façons :

```
fn main() {

    let data = "initial contents";
    let s = data.to_string();

    let s = "bla bla bla".to_string();

    let hello = String::from("Hello");
}
```

On peut ajouter une sous-chaîne à une chaîne avec la méthode **push_str**. Quant à la méthode **push**, elle permettra d'ajouter un caractère à une chaîne. Comme vous le verrez un peu plus tard, on peut également utiliser la macro **format!** pour concaténer plusieurs chaînes avec des séparateurs différents.

La dernière des collections classiques fournies par Rust, c'est la Hash Map. Son type **HashMap<K, V>** stocke donc des valeurs de type **K** en les rattachant à des clés **K**. Regardons quelques opérations que l'on peut faire avec une telle collection :

```
01: fn main() {
02:
03:     use std::collections::HashMap;
04:
05:     let mut scores = HashMap::new();
06:
07:     scores.insert(String::from("Blue"), 10);
08:     scores.insert(String::from("Blue"), 2);
09:
10:     scores.entry(String::from("Blue")).or_insert(50);
11:     scores.entry(String::from("Green")).or_insert(50);
12:
13:     scores.insert(String::from("Yellow"), 50);
14:
15:     let team_name = String::from("Blue");
16:     let score = scores.get(&team_name);
17:     for (key, value) in &scores {
18:         println!("{}: {}", key, value);
19:     }
20: }
```

Ligne 3, on déclare vouloir utiliser les **HashMap**. Ensuite ligne 5, on déclare une instance de **HashMap**. Lignes 7 et 8, on va insérer des données dans notre collection. Un insert sur une clé déjà existante (comme la ligne 8) permet de

mettre à jour la valeur. Les lignes 10 et 11 ne permettent d'ajouter une valeur que si la clé n'existe pas déjà ; dans notre exemple, la ligne 10 ne sert donc à rien et la ligne 11 ajoute une clé `Green` ayant pour valeur `50`. En ligne 16, on récupère la valeur stockée dans la map pour une clé précise. Et enfin, lignes 17 à 18, on utiliser un `for` pour parcourir toutes les clés et les valeurs.

3.2 Structure de données composées.

Jusqu'à présent, nous n'avons manipulé que des données « simples » : des entiers, des tableaux statiques, ou même des `String`. Rust permet bien entendu de créer ses propres structures de données se composant de plusieurs données. On utilise pour cela le mot clé `struct`. On pourra également ajouter des méthodes à nos structures de données. En fait, les `struct` sont ce qu'il y a de plus proche en Rust des classes dans les langages objets classique. Prenons un exemple :

```
01: fn main() {
02:     struct Character {
03:         first_name: String,
04:         last_name: String,
05:         class: String,
06:         life: i32,
07:         attack: i32,
08:     }
09:
10:     let mut conan = Character {
11:         first_name:
String::from("Conan"),
12:         last_name: String::from("The
Barbarian"),
13:         class:
String::from("Barbarian"),
14:         life: 45,
15:         attack: 53,
16:     };
17:
18:     conan.attack = 56;
19:     println!(
20:         "{} {}, life: {}, Attack: {}",
21:         conan.first_name, conan.last_
name, conan.life, conan.attack
22:     )
23: }
```

Lignes 2 à 7, on commence par déclarer notre `struct`. Ensuite, on crée une instance dans les lignes 10 à 11. On la définit comme étant mutable, ce qui nous permet, ligne 18, de modifier la valeur d'un des attributs de notre instance. L'accès en lecture et en modification aux attributs d'une instance se fait en utilisant le `.`.

Définir des structures de données composées, c'est bien, mais leur ajouter des méthodes, c'est tout de même mieux. Rust propose un bloc de code spécifique, le bloc `impl` pour définir les méthodes d'une structure. Le premier argument d'une méthode doit être `self`, `&self` ou `&mut self` suivant les besoins de nos méthodes. Reprenons notre `struct` modélisant un personnage et ajoutons-lui quelques méthodes.

```
01: fn main() {
02:     struct Character {
03:         first_name: String,
04:         last_name: String,
05:         class: String,
06:         life: i32,
07:         armor: i32,
08:         attack: i32,
09:     }
10:
11:     impl Character{
12:         fn get_name(&self) -> String{
13:             format!("{} {}", self.first_
name, self.last_name)
14:         }
15:         fn take_damages(&mut self,
damage: i32){
16:             self.life -= damage - self.
armor;
17:         }
18:
19:         fn attacked(&mut self, other:
&Character){
20:             self.take_damages(other.
attack);
21:         }
22:
23:     }
24:
25:     let mut conan = Character {
26:         first_name:
String::from("Conan"),
27:         last_name: String::from("The
Barbarian"),
28:         class: String::from("Barbarian"),
29:         life: 45,
30:         armor: 2,
31:         attack: 25,
32:     };
33:
34:     let mut naughty = Character {
```

```
35:         first_name: String::from("Bob"),
36:         last_name: String::from("The Naughty"),
37:         class: String::from("thief"),
38:         life: 45,
39:         armor: 1,
40:         attack: 8,
41:     };
42:
43:      conan.take_damages(10);
44:      conan.attacked(&naughty);
45:      println!(
46:         "{}, life: {}",
47:         conan.get_name(), conan.life
48:     )
49: }
```

On commence par rajouter quelques attributs. Ensuite, le bloc implémentation commence ligne 11. On indique que l'on veut commencer une implémentation pour la **struct Character**. Ensuite lignes 12 à 14, on définit notre première méthode qui se contente de concaténer le nom et prénom pour renvoyer une **String**. On utilise pour cela la macro **format!**. Ensuite lignes 15 à 17, on définit une méthode qui va pouvoir muter l'instance qui la lance et qui va simplement faire une soustraction sur la vie du personnage. Enfin, on met en place une dernière méthode, lignes 19 à 21, qui, en plus de prendre l'instance en premier paramètre, prend un deuxième objet personnage comme paramètre non mutable cette fois.

Ensuite, on déclare nos deux personnages et on utilise nos trois méthodes. Et si vous voulez définir une méthode de classe et pas une méthode d'instance ? C'est assez similaire aux autres langages utilisant un premier argument **self** : il suffit de ne pas le mettre. Les méthodes de classes sont souvent utilisées pour définir des constructeurs d'objets. On pourrait ainsi définir une telle méthode pour nos personnages de la manière suivante.

```
01: fn create_character(first_name: String,
last_name: String,
02:     class: String, life: i32, armor: i32,
attack: i32,
03: ) -> Character {
04:     Character {
05:         first_name,
06:         last_name,
07:         class: class,
08:         life: life,
09:         armor,
10:         attack,
11:     }
12: }
```

Vous remarquerez que pour construire l'objet, je n'ai majoritairement pas utilisé la notation **nom_attribut : valeur**, mais plutôt uniquement le nom de la variable. C'est une optimisation appelée le Field Init Shorthand que permet Rust, lorsque la variable contenant la future valeur et l'attribut ont le même nom. Pour appeler notre constructeur, on devra utiliser l'opérateur **::**, celui-là même que l'on utilise depuis le quasi-début de cet article pour construire une **String** à partir d'une chaîne littérale, grâce à la méthode de classe **from**.

3.3 Les traits

Les traits permettent en Rust de partager des comportements. Un **trait** est constitué de plusieurs signatures de méthodes qui peuvent avoir ou pas des implémentations par défaut. Ensuite, on pourra implémenter différents traits pour des structures. Reprenons notre classe de personnage et ajoutons-lui un **trait** permettant de décrire les différentes caractéristiques :

```
01: trait Describe {
02:     fn describe(&self) -> String ;
03:
04:     fn describe_with_default(&self) ->
String{
05:         String::from("On ne sait pas
quoi afficher ici !")
06:     }
07: }
08:
09: impl Describe for Character {
10:     fn describe(&self) -> String {
11:         format!("{} life: {}, armor:
{}, attack: {} , class: {}",
12:                 self.get_name(), self.
life, self.armor, self.attack, self.class)
13:     }
14: }
15: . . .
16: println!("{}", conan.describe())
17: println!("{}", conan.describe_with_
default());
```

On commence ligne 1 à 7 par déclarer notre **trait**. On lui déclare tout d'abord une signature de méthode ligne 2 et ensuite, on déclare une méthode dont on donne une implémentation par défaut. Ensuite lignes 9 à 14, on implémente le **trait** pour la **struct Character**. Pour finir,

lignes 16 et 17, on utilise les deux méthodes que le **trait** nous fournit, avec notre instance **Conan**. Les traits vont aussi pouvoir être utilisés pour définir un retour de fonction.

On pourra par exemple définir une fonction ainsi :

```
fn returns_describable()  -> Describe {
```

Attention toutefois, ce n'est pas parce qu'on définit uniquement le type de retour avec un **trait** que la fonction va pouvoir retourner des instances de types différents. Ici, **returns_describable** devra retourner des objets d'un seul type, pas précisé dans la signature. Mais on sait que cet objet implémentera le **trait Describe**.

4. NOTIONS AVANCÉES

4.1 Écrire des tests

Lors de la rapide présentation de Cargo, **test** a été listé comme l'une des commandes possibles. Eh oui, c'est Cargo qui va également s'occuper de lancer nos tests. Basiquement, un test en Rust, c'est simplement une fonction à laquelle on a rajouté une annotation la désignant comme test. L'exemple le plus simple étant :

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}
```

Les fonctions de tests peuvent se trouver à peu près partout dans votre code ; si vous voulez écrire les tests dans le même fichier que votre code, vous aurez alors quelque chose ressemblant plus à cela :

```
01: #[cfg(test)]
02: mod tests {
03:     #[test]
04:     fn it_works() {
05:         assert_eq!(2 + 2, 4);
06:     }
07: }
```

Ligne 1, l'annotation **cfg** permet d'indiquer à **cargo** de ne compiler le module **test** défini juste en dessous que lorsqu'on lance la commande **test** et pas lorsqu'on lance la commande **build**, par exemple. On définit ensuite un module **test** qui va regrouper nos tests (ligne 2), puis on retrouve notre test.

Si vous préférez regrouper vos tests dans des répertoires tests, vous n'aurez alors plus besoin de l'annotation **cfg**. Rust traite en effet les dossiers tests d'une manière spéciale et ne les compile que lorsqu'on lance la commande **test**.

Pour coder vos tests, vous pourrez utiliser les assertions suivantes :

- **assert_eq!** qui teste l'égalité des deux arguments ;
- **assert_ne!** qui teste l'inégalité des deux arguments ;
- ou **assert!** qui ne prend qu'un argument et vérifie si celui-ci est vrai.

4.2 Les closures

En Rust, les closures sont des fonctions anonymes qui vont pouvoir être sauvegardées dans des variables ou passées en argument à d'autres fonctions. On utilise **||** pour les définir. Une closure peut avoir ou ne pas avoir d'argument. Quelques exemples :

```
let identity_closure = |x: i32| x;
let add_closure = |x: i32| { x + 1 };
let add_closure_v2 = |x: i32|  x + 1 ;
let multi_closure = |x: i32| {
    println!("on affiche un truc");
    x + 1
};
let closure_one = || 1 ;
```

On le voit, on peut avoir des closures avec ou sans argument, avec des corps très simples ou au contraire prenant la forme de blocs d'instructions.

Un dernier point important sur les closures : elles sont capables de capturer les variables qui sont dans l'environnement où elles ont été définies. Cela va permettre de faire la chose suivante :

```
fn main() {
    let x = 4;
    let equal_to_x = |z| z == x;
    let y = 4;
    assert!(equal_to_x(y));
}
```

Ici, la closure capture la valeur de **x** au moment de sa définition. C'est un comportement que l'on ne peut pas avoir avec une fonction.

4.3 Programmation concurrente

L'un des points sur lequel insiste Rust, c'est d'être un langage proposant une concurrence efficace et sûre. Tout est donc prévu pour pouvoir mettre en place des threads. Commençons par le « Hello world! » de la programmation concurrente, à savoir plusieurs threads qui se lancent, puis que le programme principal attend sagement avant de quitter :

```
01: use std::thread;
02: use std::time::Duration;
03:
04: fn main() {
05:     let handle = thread::spawn(|| {
06:         for i in 1..10 {
07:             println!("Hello World Concurrent venant du
Thread {}  ! ", i);
08:             thread::sleep(Duration::from_millis(1));
09:         }
10:     });
11:
12:     handle.join().unwrap();
13: }
```

On commence par déclarer les modules dont on va avoir besoin, puis on appelle la fonction **spawn** du module **thread** qui prend en paramètre une closure qui contient le code que l'on veut lancer dans notre nouveau thread (ligne 5 à 10).

Ensuite, on va tout simplement attendre la fin de tous les threads en appelant la méthode **join()** du handle en ligne 12. Tout cela est très bien, mais ce qui est bien avec les threads, c'est que l'on peut les faire communiquer. Pour y arriver, il nous faut résoudre deux problèmes : avoir un moyen de communication entre deux threads et arriver à respecter les règles d'ownership de Rust. Le moyen de communication, Rust nous le propose avec son type **channel** qui permet de faire du passage de message. Quant au respect des règles d'ownership de Rust, on va y arriver grâce à la propriété de capture des closures et du mot clé **move**.

```
01: use std::sync::mpsc;
02: use std::thread;
03:
04: fn main() {
05:     let (tx, rx) = mpsc::channel();
06:
07:     thread::spawn(move || {
08:         let val = String::from("Ping !");
09:         tx.send(val).unwrap();
10:     });
11:
12:     let received = rx.recv().unwrap();
13:     println!("Reçu: {}", received);
14: }
```

Ligne 5, on commence par créer notre **channel**. Un **channel** est constitué de deux parties : un récepteur (ici **rx**) et un émetteur (ici **tx**).

Ensuite, on crée notre thread ligne 7 à 10 et la closure que l'on utilise capture la partie transmetteur du **channel** pour pouvoir envoyer « Ping ! » à la ligne 9. Remarquez bien le mot clé **move** à la ligne 7. C'est grâce à lui que tout fonctionne. Il permet en effet de forcer le fait que l'ownership sur **tx** passe à la closure. Sans cela, le thread ne posséderait pas le transmetteur et ne pourrait pas écrire à l'intérieur.

Du côté du thread principal, on se contente d'attendre une valeur du côté récepteur, puis on l'affiche. D'ailleurs, avec cette valeur envoyée d'un thread à l'autre, qu'est-ce qui se passe au niveau de son ownership ? Au tout début, elle est crée ligne 8. À ce moment-là, c'est le thread émetteur qui la possède. Mais ensuite quand on l'envoie à la partie récepteur à la ligne 9, cela transfère également l'ownership au thread récepteur. Si bien que si juste après avoir fait le **tx.send()**, on essayait de faire un **println!("{}", val)**, on aurait une erreur de compilation.

CONCLUSION

Rust est un phénomène à lui tout seul. Et pas uniquement pour les opinions très tranchées qu'il met en place. Il est très étonnant qu'un langage aussi jeune, à peine dix ans, et dont le but était au départ très spécialisé, se soit développé aussi vite dans autant de domaines différents. Que vous vouliez faire du développement web front sans utiliser du **JavaScript**, du développement web backend **[8]**

ou même du jeu vidéo **[9]**, il y a des solutions en Rust. Même la communauté scientifique s'y met et c'est la très sérieuse revue **Nature** qui le dit **[10]**. Et pourtant, écrire ses premiers programmes peut être très rageant, se battre avec les règles d'ownership peut parfois donner envie de baisser les bras. La barrière à l'entrée est loin d'être évidente, bien au contraire. C'est même une des plus hautes que je connaisse pour un langage de programmation moderne. Mais rien ne se gagne sans effort. Et la certitude qu'au final cela va marcher, qu'il n'y aura pas d'erreurs de mémoire, que cela sera rapide, que la concurrence ne se prendra pas les pieds dans les fils du tapis, cela vaut bien quelques nœuds au cerveau... Pour ma part, j'espère que ce long article vous a donné envie de vous lancer dans la découverte de ce langage, qui est à mon avis l'un des plus innovants et motivants depuis de très longues années. Et je finirai en citant Ferris : « Hello, fellow Rustaceans ! ». ■

RÉFÉRENCES

[1] Rust : http://rust-lang.org

[2] Servo : https://github.com/servo/servo

[3] Rust chez Dropbox : https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[4] Article ZDNet : https://www.zdnet.com/article/mozilla-lays-off-250-employees-while-it-refocuses-on-commercial-products/

[5] Rustfmt : https://github.com/rust-lang/rustfmt

[6] Clippy : https://github.com/rust-lang/rust-clippy

[7] Commencer avec Rust : https://www.rust-lang.org/fr/learn/get-started

[8] Rust pour le dev web : https://github.com/flosse/rust-web-framework-comparison

[9] Rust pour le développement de jeux : https://arewegameyet.rs/

[10] « Why scientists are turning to Rust » : https://www.nature.com/articles/d41586-020-03382-2

GESTION DE PROJETS AVEC ERLANG/OTP

MATHIEU KERJOUAN
[Dealer de nœuds distribués en Erlang]

MOTS-CLÉS : ERLANG, OTP, BEAM, GESTION DE PROJETS

Un langage de programmation se doit d'être facile d'accès, que ce soit pour son apprentissage, la réalisation de concepts ou de produits finaux. La création de projets en Erlang se fait via les notions d'application et de release. Couplés à différents outils internes ou fournis par la communauté, ces principes permettent de créer un environnement de production flexible et maintenable sur le long terme, tout en facilitant la diffusion et le partage des modules conçus par les créateurs.

Après avoir présenté la syntaxe et la logique du langage **Erlang** dans un premier article **[1]**, puis après avoir fait une introduction de la suite logicielle et des concepts liés à Erlang/OTP dans un second **[2]**, ce nouvel article s'attaque à la lourde tâche de présenter les concepts d'application et de release au sein d'Erlang. Toujours dans la continuité des précédents écrits, le projet d'exemple utilisant un système de cache sera une fois de plus utilisé. Ce projet s'adapte parfaitement au modèle acteur conçu par Erlang et offre ainsi l'opportunité d'être mis à jour étape par étape, comme un projet réalisé en entreprise ou dans le vaste monde de l'open-source.

Avant de rentrer dans le vif du sujet, un récapitulatif historique s'impose tout de même. Erlang **[3]**, langage créé à la fin des années 90 par Joe Armstrong au sein de la société **Ericsson**, s'était donné pour objectif de créer une suite d'outils permettant de régler les nombreux problèmes rencontrés dans le monde de l'industrie

et de la télécommunication. Ce langage, après de nombreuses années de développement et de test, a permis de révolutionner l'utilisation de la programmation fonctionnelle et distribuée. La mise en avant d'une philosophie innovante de gestion d'erreurs au travers de l'exploitation de microprocessus isolés a permis de redéfinir des concepts que l'on pensaient alors avoir été totalement acquis par la communauté des développeurs, comme la notion d'objet qui, par définition, aurait dû être totalement isolée, mais qui, en pratique, ne l'est que trop rarement.

Qui parle de révolution sous-entend des changements profonds dans les principes fondamentaux. Ce renversement de paradigme donna la possibilité de modeler un nouvel univers sur des principes novateurs ou encore trop peu utilisés dans le monde du logiciel. Des concepts tels que les processus, les superviseurs, les travailleurs, les arbres de supervision ou encore les behaviours sont autant de nouvelles notions qui ont émergé et qui ont généralement bouleversé les habitudes des personnes ayant pour tâche de concevoir, de créer ou encore de déployer les produits issus de cet environnement. C'est pour cela qu'il est nécessaire de mettre à disposition les outils ainsi que les ressources **[4]** facilitant la transition, tout en permettant de répondre aux besoins de cohérence et de flexibilité. Les modèles d'applications, de releases et de gestionnaires de projets présentés ici en font partie.

1. INTRODUCTION

Le système de cache, conçu lors des articles précédents, est normalement fonctionnel. Certes, il est loin d'être parfait et est extrêmement limité en fonctionnalités, mais répond au besoin originel : stocker des valeurs associées à des clés, dans une structure de données isolée au sein d'un processus nommé. Ce comportement est globalement similaire à celui d'une base de données **Redis** ou **MongoDB**. Bien que l'outil soit fonctionnel, le lecteur a dû gérer manuellement le projet sans aide et uniquement via la ligne de commandes. La compilation des différents fichiers s'est faite sans outils spécifiques. Pour les lecteurs habitués à concevoir des logiciels, leur premier réflexe fut sans doute celui de créer un `Makefile`, permettant ainsi d'automatiser les différentes tâches récurrentes et rébarbatives, telles que la compilation, les tests ou encore celle du lancement de la machine virtuelle, voire pour les plus téméraires, du déploiement.

Erlang étant un langage moderne, les projets conçus sur cette plateforme ont la possibilité d'être maintenus par un gestionnaire de projet. Effectivement, de tels outils existent d'ores et déjà pour d'autres langages, tels que **Maven** ou **Ant** avec **Java**, **SBT** avec **Scala**, **Cargo** avec **Rust** ou encore **Leiningen** pour **Clojure**. Contrairement aux deux derniers langages cités, qui fournissent nativement ces outils pour la gestion de projets, Erlang laisse le choix aux développeurs de prendre le ou les outils qui leur semble être le plus adapté. Deux grands noms reviennent fréquemment, à savoir **Erlang.mk** et **Rebar3**, qui deviendront prochainement vos points d'entrée pour la conception de logiciels écrits en Erlang.

Avant de parler de cesdits gestionnaires de projets qui feront la joie des développeurs pressés, il est nécessaire, voire primordial de comprendre ainsi que de connaître la structure d'un projet conçu avec et pour l'environnement Erlang/OTP.

2. APPLICATION

Le terme « application » peut parfois porter à confusion pour les nouveaux arrivants dans l'écosystème Erlang. La définition peut être trompeuse et regroupe plusieurs concepts. Tout d'abord, une application est un regroupement de code implémentant une ou plusieurs fonctionnalités, et agissant comme un composant unique au sein du système. Par extension, une application facilite la modularité ainsi que le partage avec d'autres systèmes ayant besoin de ces fonctionnalités pour différents contextes.

Concrètement, une application **[5]** est généralement constituée d'un arbre de supervision composé lui-même de processus superviseurs et de processus travailleurs. Une grande partie de ces éléments utilisent une ou plusieurs bibliothèques fournissant les implémentations nécessaires au bon fonctionnement de l'application. Pour lier toutes ces briques entre elles, Erlang/OTP fournit un behaviour appelé `application`, facilitant le regroupement des sous-processus entre eux et standardisant par la même occasion les méthodes de démarrage puis d'arrêt, au travers de fonctions exportées par ce même module.

La création d'une application en Erlang reste classique et se base sur un module créé par le développeur utilisant le behaviour `application`. Comme tout logiciel créé en Erlang, le code nécessite la création d'un en-tête correspondant

aux différents paramètres utilisés par le compilateur pour lui faciliter son travail. Cet en-tête permet tout d'abord de définir le namespace, correspondant au nom du module, ainsi que les fonctions exportées par celui-ci et terminées par le type de behaviour utilisé.

```
-module(mon_application).
-export([start/2, stop/1]).
-behaviour(application).
```

Le premier callback à définir est **start/2**. Ce callback reçoit au travers du premier argument le type de démarrage de l'application. Effectivement, Erlang permet en grande partie de faciliter le déploiement de systèmes distribués, et une application peut naturellement démarrer dans un tel environnement. Le type de démarrage correspond alors à définir le comportement qu'une application pourra adopter si cette dernière est démarrée sur un nœud clusterisé.

Il existe globalement trois types de démarrages. Le type **normal** est celui par défaut, et se contentera de démarrer l'application localement, cloisonné sur son nœud. Le second type de démarrage se nomme **takeover**, et deviendra alors l'application principale sur le cluster Erlang, dans le cas où la même application fonctionnerait sur un des autres nœuds. Le dernier mode se nomme **failover** et fait office de comportement de roue de secours. Si la même application présente sur le cluster crashe ou s'arrête d'une façon non désirée, alors l'application en failover reprend automatiquement la main dessus.

Le deuxième argument de ce callback correspond aux options passées à l'application lors de son démarrage. Ces arguments sont libres et restent généralement définis par le développeur qui voudrait que son application réagisse différemment en fonction d'un contexte précis. Finalement, ce callback doit retourner un tuple semblable à celui retourné par un processus démarré avec les fonctions **start/3** ou **start_link/3**, présentent dans les modules **gen_server**, **gen_statem** ou **supervisor**. Le premier élément de ce tuple contient généralement un tag sous forme d'atome (**ok** ou **error**) suivi de l'identifiant du processus faisant référence au superviseur ou au processus démarré.

```
start(_Type, _Arguments) ->
  mon_application_sup:start_link().
```

Le deuxième callback implémenté est **stop/1**. Son premier et seul argument correspond à l'état de l'application, généralement le contenu du retour du callback **start/2**. Cette fonction ne retourne pas de donnée particulière et agit comme un appel asynchrone.

```
stop(_Etat) ->
  ok.
```

Ces deux callbacks ne sont pas les seuls disponibles, mais sont les deux les plus utilisés et indispensables pour le behaviour **application**. Les autres callbacks correspondent à des fonctionnalités rencontrées généralement sur OTP, mais appliquées au monde de l'application. Le callback **config_change/3** permet de faire une transition de configuration, correspondant à un changement d'état. Le callback **prep_stop/1** permet d'effectuer des actions avant l'arrêt de l'application, comme arrêter un service externe ou sauvegarder un état. Finalement, le callback **start_phase/3** permet de définir un ordre de démarrage, permettant de synchroniser le démarrage de plusieurs applications.

Mais ce n'est pas tout. Le module **application**, correspondant au behaviour du même nom, fournit un nombre important de fonctions pour configurer ce système durant sa période d'activité. Effectivement, une application doit pouvoir être paramétrable facilement, et ces différentes configurations doivent pouvoir se répercuter tout aussi aisément sur les sous-processus gérés par le superviseur applicatif. Un exemple est souvent plus compréhensible qu'un long discours. Lors du démarrage d'un nœud Erlang, plusieurs applications sont d'ores et déjà démarrées sur le système, telles que **kernel** ou **stdlib**, qui permettent de gérer l'état global de la machine virtuelle.

Ces deux applications ne dérogent pas à la règle précédente, et peuvent être paramétrées ou possèdent déjà des paramètres de configuration. Pour récupérer la liste de ces variables, la fonction **application:get_all_env/1** peut-être utilisée. Son premier argument correspond au nom de l'application visée et retournera les éléments sous forme d'une liste de tuples, ou comme montrée dans l'exemple suivant, d'une liste vide.

```
1> application:get_all_env(stdlib).
[]
```

Évidemment, une liste vide ne saurait rester vide très longtemps. Pour rajouter une valeur dans les variables d'environnement de l'application, la fonction **application:set_env/3**

peut-être utilisée. Le premier argument reçoit le nom de l'application suivi de la clé, puis de la valeur associée.

```
2> application:set_env(stdlib, ma_variable,
test).
ok
```

Comme le lecteur doit s'en douter, il est aussi possible de récupérer un élément de la liste, tout comme le gestionnaire de cache créé en exemple durant les deux derniers articles, les variables d'environnement d'une application fonctionnent sur le même principe. La fonction **application:get_env/2** est alors utilisée. Son premier argument correspond au nom de l'application, et le second à la clé. Cette fonction retourne la valeur associée à la clé.

```
3> application:get_env(stdlib, ma_variable).
{ok,test}
```

En revenant sur la fonction **application:get_all_env/1**, son utilisation permet de retourner une liste de tuples, comme déclarée précédemment. Cette structure de données est appelée **proplist** et se retrouve à pratiquement tous les niveaux des applications et des releases en Erlang, même si cette structure de données est maintenant mise au second plan, au profit des **maps**.

```
4> application:get_all_env(stdlib).
[{ma_variable,test}]
```

Pour démarrer ou mettre fin à une application, les fonctions **application:start/1** et **application:stop/1** sont exportées. Par exemple, pour démarrer l'application **crypto**, permettant de gérer une grande partie des fonctionnalités de chiffrement sur un nœud Erlang, la commande suivante sera utilisée :

```
5> application:start(crypto).
ok
```

Pour récupérer la liste des applications démarrées actuellement sur le système, la fonction **application:which_applications/0** pourra alors être utilisée. Cette fonction retourne une liste contenant le nom des applications, la description de ces dernières, ainsi que la version associée en fonctionnement sur le nœud.

```
5> application:which_applications().
[{crypto,"CRYPTO","4.4"},
 {stdlib,"ERTS  CXC 138 10","3.7"},
 {kernel,"ERTS  CXC 138 10","6.2"}]
```

Évidemment, une application fonctionne rarement seule isolée dans son coin, et elle peut avoir besoin de dépendances indispensables pour lui permettre de se mouvoir correctement. Si ces différentes dépendances n'ont pas été démarrées manuellement par le développeur avant le lancement de l'application, cette dernière ne devrait pas être en capacité de démarrer et devrait retourner une erreur indiquant la raison de l'échec.

Pour éviter ce genre de situation pouvant être dramatique lors d'une mise en production, la fonction **application:ensure_all_started/1** a été conçue. Cette fonction permet de démarrer automatiquement la liste des dépendances. Par exemple, l'application **ssl** permet d'utiliser l'implémentation SSL/TLS au sein d'un nœud Erlang et dépend des applications **crypto**, **asn1**, et **public_key**. Dans le cadre d'un démarrage manuel de l'application avec la fonction **application:start/1**, cette dernière retourne une erreur, car une ou plusieurs dépendances ne sont pas en cours de fonctionnement.

```
6> application:start(ssl).
{error,{not_started,crypto}}
```

Le démarrage de l'application **ssl** via la fonction **application:ensure_all_started/1** se passe sans accrocs, car cette fonction va regarder la liste des dépendances et démarrer les démarrer une par une.

```
7> application:ensure_all_started(ssl).
{ok,[crypto,asn1,public_key,ssl]}
```

Le lecteur attentif se posera alors une question fondamentale : comment une application peut-elle connaître ses dépendances ? Pour le savoir, le système applicatif se tourne vers un fichier de ressources **[6]**, étroitement lié au module utilisant le behaviour **application**. Un tel fichier est livré aux côtés de l'application elle-même et contient une structure de données en terme Erlang. Ce fichier a pour nom celui de l'application, suivi de l'extension **.app**. Son contenu est standardisé et correspond à un tuple ayant alors trois champs. Le premier contient l'atome **application**, le second contient le nom de l'application sous la forme d'un atome et le troisième contient les différents paramètres liés à l'application. Voici l'exemple d'un tel fichier, nommé **mon_application.app**, qui permet de spécifier une application nommée **mon_application**.

```
{ application, mon_application,
   [{description, "une application de test"},
```

```
    {id, "MON APPLICATION"},
    {vsn, "0.0.1"},
    {modules, [mon_application_sup]},
    {registered, [mon_application_sup]}
    {applications, []}]
}.
```

Ce fichier contient des informations essentielles, telles que le nom de l'application, sa description, son identifiant, sa version, la liste des modules chargés par l'application, la liste des applications qui seront exposées (visible des autres nœuds), la liste de dépendances des applications et encore bien d'autres champs disponibles qui sont définis dans la documentation officielle **[7]**, permettant ainsi de garantir le bon fonctionnement de l'application. Après enregistrement du fichier, et si les sous-modules ont été correctement créés, l'application pourra alors être démarrée et utilisée sur le nœud Erlang.

```
8> application:start(mon_application).
ok
```

Toutes les étapes présentées ici correspondent à la création d'une application Erlang artisanalement conçue, et font généralement partie de la culture générale à avoir pour comprendre la logique derrière les gestionnaires de projets. Ces derniers permettent d'ailleurs la création et la génération automatique des fichiers et autres configurations qui ont été vues dans cette partie, mais laisse une fois de plus la possibilité au développeur de les optimiser ou customiser en fonction des besoins du projet en cours de conception.

3. RELEASE

Une release **[8]** est ce qu'est une application aux modules. Une application unique n'est généralement pas suffisante pour de nombreux projets et particulièrement ceux complexes nécessitant l'utilisation de nombreuses fonctionnalités. Les releases permettent donc de créer un système ayant la possibilité de contrôler plusieurs sous-applications. Comme le lecteur peut s'en douter, la gestion d'une release n'est pas forcement des plus simples. Cette partie de l'article est uniquement ici présente pour survoler ce concept et ainsi voir les possibilités offertes par Erlang pour la livraison de produits finis.

Tout comme une application est livrée avec ses spécifications sous forme d'un fichier de ressources, la release suit le même principe, à l'exception du fait que ce fichier utilise une extension en **.rel**. Pour donner un exemple, les commandes suivantes permettent de créer un répertoire **ma_release** qui contiendra le fichier **ma_release.rel**.

```
$ mkdir ma_release
$ cd ma_release
$ touch ma_release.rel
```

Encore une fois, tout comme le modèle de l'application, le fichier de ressources concernant une release contient les informations nécessaires au bon fonctionnement de cette dernière. Sous la forme d'un tuple de dimension quatre, le premier élément est un atome release, le second élément contient le nom de la release suivi de sa version, le troisième élément contient la version d'**erts** (Erlang Run-Time System) correspondant grossièrement à la version de l'interpréteur Erlang, puis de la liste des différentes applications à livrer avec la release, telles que le **kernel**, **stdlib** et **sasl**.

```
{ release,
  {"ma_release","VersionRelease"},
  {erts, "VersionERTS"},
  [{kernel, "VersionKernel"},
   {stdlib, "VersionStdLib"},
   {sasl, "VersionSASL"}
  ]
}.
```

Le lecteur attentif remarquera que les versions ne sont pas configurées dans le fichier de la release. Ces versions dépendent de votre système, et sont à trouver au niveau du chemin où les bibliothèques Erlang sont stockées. Éventuellement, plusieurs fonctions peuvent retourner des numéros de version, comme **application:which_applications/0** ou encore **release_handler:which_releases/0**.

La machine virtuelle Erlang, en plus de ce fichier décrivant les applications, a besoin d'un script pour permettre le démarrage des applications. Le module **systools** exporte la fonction **systools:make_script/1** qui prend en argument le nom de la release et génère automatiquement le script de boot.

```
1> systools:make_script("ma_release").
ok
```

À la suite de cette action, deux fichiers sont normalement générés. Le premier se nomme **ma_release.script**, contenant une structure de données Erlang sous forme d'un

tuple et décrivant textuellement les différentes étapes du boot. Le deuxième fichier se nomme **ma_release.boot** et est l'équivalent du précédent fichier, mais compilé en format binaire ETF (Erlang Term Format). Pour vérifier le bon fonctionnement de ce script, l'utilisation de la commande **erl** suivie du nom de la release suffit à démarrer le système et donne accès à un shell Erlang.

```
$ erl ma_release
Erlang/OTP 21 [erts-10.2] [source] [64-bit]
[smp:2:2] [ds:2:2:10] [async-threads:1]

Eshell V10.2  (abort with ^G)
1> q().
ok
```

Maintenant que le script fonctionne et que le système semble utilisable, il doit alors pouvoir être livré. Le module **systools** offre alors la fonction **systools:make_tar/1** qui permet de créer un tarball contenant tous les fichiers compilés et prêts à l'usage, donc prêts à être déployés dans un environnement de production ou livrés au client final.

```
2> systools:make_tar("ma_release").
ok
```

Le fichier généré se nomme **ma_release.tar.gz** et contient, comme convenu, la liste des applications, compilées et prêtes à l'usage. Il est à noter qu'une release ne contient pas que l'application conçue, mais aussi le système dans son intégralité, comme le kernel, les bibliothèques standard et tout ce qui est nécessaire au bon fonctionnement du futur nœud. Pour le déploiement, ce fichier peut-être décompressé sur le serveur qui contiendra le système. Tout comme les applications, il est aussi possible de gérer la mise à jour intelligemment via différents handlers et principes... qui seront probablement détaillés dans un article dédié à la mise en production.

Finalement, pour les personnes pour lesquelles la ligne de commande est répulsive ou pour les personnes ayant peur de faire des erreurs, Erlang est livré avec le module **reltool** qui permet de générer les fichiers de ressources, mais aussi de fournir un nombre important d'outils pour voir et gérer les dépendances entre les versions. Le bonus : tout est réalisé au travers d'une GUI (qui nécessite le module **wx**, généralement fourni séparément sous **GNU/Linux** / **UNIX**, pour ceux qui rencontreraient des problèmes pour le démarrer).

4. REBAR3

Rebar3 **[9]** est probablement à ce jour l'outil le plus utilisé dans la communauté Erlang pour la gestion des projets. Développé il y a maintenant plusieurs années, il permet de fonctionner comme un gestionnaire de projets classique, offrant toutes les fonctionnalités d'un Maven, **SBT** ou encore **Leiningen**. Il est à l'origine issu d'un fork de **rebar2**, basé sur une réécriture complète du code par Fred Hebert, Tristan Sloughter et Tuncer Ayaz. Ce projet fait d'ailleurs partie des outils officiellement maintenus par l'équipe Erlang. Par ailleurs, et comme indiqué au début de l'article, Rebar3 n'est pas disponible nativement avec les releases d'Erlang et nécessite donc son installation soit via un paquet, soit via les sources. Évidemment, dans le cadre de cet article, l'auteur choisit ici de vous montrer l'installation via les sources. Rebar3 est intégralement écrit en Erlang et nécessite d'avoir d'ores et déjà l'environnement installé sur votre système via les sources ou votre gestionnaire de paquets favoris.

```
$ git clone https://github.com/erlang/rebar3
$ cd rebar3
$ git checkout 3.9.1
  ...
$ ./bootstrap
  ...
$ PATH=${PATH}:$(pwd)
$ rebar3
Rebar3 is a tool for working with Erlang projects.

Usage: rebar3 [-h] [-v] [<task>]
  -h, --help     Print this help.
  -v, --version  Show version information.
  <task>         Task to run.
```

Par souci de clarté, la sortie de l'écran a été tronquée, mais chacune des tâches les plus utilisées va être développée au cas par cas dans cette partie de l'article, ainsi qu'au travers d'un exemple concret. Maintenant que Rebar3 est disponible dans le **PATH**, son exécution pourra se faire n'importe où sur le système.

Il n'est généralement pas bon de produire ce qui a d'ores et déjà été produit dans le passé, mais pour pouvoir avancer sereinement, le lecteur est invité à recréer le module cache. Pour se faire, la sous-commande **new** est utilisable et permet de générer automatiquement une arborescence pour un type de projet défini. Cette arborescence est alors produite à partir d'un template, généralement fourni avec Rebar3, mais pouvant être aussi produit par le développeur pour ses besoins.

Ces templates permettent de créer facilement une application (**app**), des facilités pour la compilation de programme en **C** et **C++** (**cmake**), un script Erlang (**escript**), une bibliothèque (**lib**), un plugin Rebar3 (**plugin**), une release (**release**) ou encore un projet basé sur **Umbrella** (généralement utilisé avec **Elixir**). Ces templates correspondent au premier argument de la sous-commande **new** et possèdent des attributs modifiables sous forme de variable à définir à la suite, permettant ainsi de modifier, par exemple, le nom du projet.

```
$ rebar3 new app name=cache
===> Writing cache/src/cache_app.erl
===> Writing cache/src/cache_sup.erl
===> Writing cache/src/cache.app.src
===> Writing cache/rebar.config
===> Writing cache/.gitignore
===> Writing cache/LICENSE
===> Writing cache/README.md
```

L'arborescence créée est compatible avec celle standardisée dans Erlang/OTP et offre les mêmes avantages. La compilation du projet se fait via la sous-commande **compile**, qui va alors compiler tous les fichiers Erlang présents dans le sous-répertoire **src**.

```
$ rebar3 compile
===> Verifying dependencies...
===> Compiling cache
```

Évidemment, dans le cadre d'un projet un peu plus complexe, un développeur aura probablement besoin d'utiliser des bibliothèques ou des applications externes. Pour ce faire, Rebar3 fournit une sous-commande nommée **get-deps** permettant de télécharger les différentes dépendances configurées dans le fichier **rebar.config**. À noter qu'Erlang a pour vocation de créer un « univers » cohérent, et générera alors un fichier **rebar.lock**, contenant les versions des dépendances téléchargées et garantissant ainsi le fonctionnement du système pour ces versions seulement.

```
$ rebar3 get-deps
===> Verifying dependencies...
```

Pour faciliter l'interaction avec le code créé, Rebar3 permet de démarrer, à la demande du développeur, un shell préchargé avec le code. Cette sous-commande est nommée **shell** et sera d'une grande utilité lors d'un développement. Outre le lancement d'un REPL classique, elle permet aussi d'offrir des outils pour la compilation du projet via les différents modules livrés avec Rebar3.

```
$ rebar3 shell
===> Verifying dependencies...
===> Compiling cache
===> The rebar3 shell is a development tool; to
deploy applications in production, consider using
releases (http://www.rebar3.org/docs/releases)
===> Booted cache
Erlang/OTP 21 [erts-10.2] [source] [64-bit]
[smp:2:2] [ds:2:2:10] [async-threads:1]

1> r3:do(compile).
===> This feature is experimental and may be
modified or removed at any time.
Verifying dependencies...
Compiling cache
ok
```

Rebar3 offre aussi la possibilité de tester plus facilement le code en intégrant les tests unitaires et autres modules de tests fournis avec Erlang/OTP. Effectivement, le module **eunit** permet de gérer les tests unitaires et **ct** permet de gérer les « tests communs » ou « common test » en anglais. Pour les utiliser avec Rebar3, les sous-commandes **eunit** et **ct** ont été créées, exécutant alors le code présent dans le sous-répertoire **test**.

```
$ rebar3 eunit
===> Verifying dependencies...
===> Compiling cache
===> Performing EUnit tests...

Finished in 0.092 seconds
0 tests

$ mkdir test
$ rebar3 ct
===> Verifying dependencies...
===> Compiling cache
===> Running Common Test suites...
All 0 tests passed.
```

Les différents supports de tests statiques et dynamiques sont aussi supportés via les sous-commandes **cover** et **dialyzer**. Des outils qui seront encore une fois développés en profondeur dans de futurs articles. Pour donner tout de même une indication, **cover** et **dialyzer** sont des outils permettant de faire une analyse de code basée sur les spécifications et les types, en s'assurant que, par exemple, les fonctions sont correctement appelées ou que les différentes variables contiennent les bons types de données lors d'une exécution.

```
$ rebar3 cover
===> Verifying dependencies...
===> Compiling cache
===> Performing cover analysis...
===> No coverdata found
```

```
$ rebar3 dialyzer
===> Verifying dependencies...
===> Compiling cache
===> Dialyzer starting, this may take a while...
===> Updating plt...
===> Resolving files...
===> Updating base plt...
===> Resolving files...
===> Building with 191 files in "/home/user/.cache/
rebar3/rebar3_21.2_plt"...
```

Rebar3 permet aussi de créer des releases et facilite le déploiement du projet en cours de création via les sous-commandes **release** et **relup**.

```
$ rebar3 release
$ rebar3 relup
```

Finalement, le fichier **rebar.config** permet de maintenir la cohérence au niveau des besoins du projet. Ce fichier est écrit en purs termes Erlang et donne la possibilité de rajouter les différentes dépendances ou fonctionnalités nécessaires au bon fonctionnement du projet en cours de réalisation. Chaque section est définie par un tuple, dont le premier élément est un atome désignant la catégorie de configuration, suivi des arguments et paramètres de cette dernière.

```
{erl_opts, [debug_info]}.
{deps, []}.

{shell, [
  % {config, "config/sys.config"},
    {apps, [cache]}
]}.
```

Rebar3 est un outil à avoir près de soi. Il permet d'accomplir un nombre incroyable de tâches habituellement laborieuses. Largement supporté par la communauté Erlang, il a aussi la particularité d'être modulable et donc, d'être extensible pratiquement à l'infini.

5. ERLANG.MK

Erlang.mk **[10]** est une solution alternative développée et maintenue par Loïc Hoguin. Ce gestionnaire de projets est intégralement écrit à l'aide de **GNU Makefile**, ce qui lui garantit une très grande portabilité. Effectivement, la commande **make** se retrouve globalement sur tous les systèmes. Erlang.mk s'appuie aussi sur les outils standard, tel que **Git**, pour pouvoir fonctionner correctement. Le **Makefile** en question, simple fichier texte, est récupérable sur le site officiel du projet.

```
$ mkdir my_application
$ cd my_application
$ curl -O https://erlang.mk/erlang.mk
```

Tout comme Rebar3, Erlang.mk utilise un système de template pour générer automatiquement une arborescence et ainsi permettre la création d'un projet en quelques secondes. La création d'une application OTP classique se fait en appelant la cible **bootstrap**. La création d'une bibliothèque OTP se fait en appelant la cible **bootstrap-lib**. Finalement, la création d'une release se fait en appelant la cible **bootstrap-rel**. Le rôle de ces cibles, en plus de générer les différents fichiers, est aussi de créer le fichier **Makefile** qui contiendra les informations relatives au projet.

```
$ make -f erlang.mk bootstrap
$ find .
./erlang.mk
./Makefile
./src
./src/my_application_app.erl
./src/my_application_sup.erl
./.erlang.mk
./.erlang.mk/last-makefile-change
```

Ce **Makefile** nouvellement créé contient alors le nom du projet, la description du projet ainsi que sa version. Le fichier **erlang.mk** devient alors une bibliothèque, directement incluse à la fin du **Makefile**, et permettant d'utiliser toutes les fonctionnalités sans avoir à l'appeler manuellement via le flag **-f** de la commande **make**.

```
PROJECT = my_application
PROJECT_DESCRIPTION = New project
PROJECT_VERSION = 0.1.0

include erlang.mk
```

La compilation du projet en exécutant la commande **make** dans le répertoire du projet ou en appelant explicitement les cibles **all** ou **app**. Les binaires qui en résultent sont alors stockés dans le répertoire **ebin**.

```
$ gmake all
 DEPEND my_application.d
ERLC   my_application_app.erl my_application_
sup.erl
 APP    my_application
$ find ebin
ebin/
ebin/my_application.app
ebin/my_application_app.beam
ebin/my_application_sup.beam
```

Erlang.mk, comme Rebar3, donne accès au REPL Erlang, incluant par défaut, lors de son exécution, les chemins nécessaires au bon fonctionnement de l'application en cours de développement. Pour y avoir accès, la commande **make** doit être exécutée avec la cible **shell**.

```
$ gmake shell
 GEN    shell
Erlang/OTP 21 [erts-10.2] [source] [64-bit]
[smp:2:2] [ds:2:2:10] [async-threads:1]

Eshell V10.2  (abort with ^G)
1>
```

Enfin, Erlang.mk supporte aussi les différentes bibliothèques et frameworks de tests fournis avec Erlang. Les tests unitaires sont exécutables via les cibles **eunit** et **ct**. L'analyse statique est exécutée via la cible **dialyze**, qui fait appel alors à **dialyzer**, générant automatiquement les fichiers nécessaires à cet outil. Finalement, la cible **check** permet de se simplifier la vie en exécutant tous les types de tests à la suite.

```
$ gmake check
 GEN    test-build
 GEN    eunit
  There were no tests to run.
 GEN    clean-app
 GEN    coverdata-clean
 DEPEND my_application.d
 ERLC   my_application_app.erl my_application_sup.erl
 APP    my_application
  Creating PLT my_application/.my_application.plt ...
Unknown functions:
  compile:file/2
  compile:forms/2
  compile:noenv_forms/2
  compile:output_generated/1
  crypto:block_decrypt/4
  crypto:start/0
Unknown types:
  compile:option/0
done in 2m32.76s
done (passed successfully)
 Checking whether the PLT my_application/.my_
application.plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.22s
done (passed successfully)
```

Erlang.mk est un gestionnaire de projets agréable à utiliser, offrant les mêmes avantages que Rebar3. Sa flexibilité et sa grande portabilité en font un outil parfait pour les projets multiplateformes et/ou multilangages. Il appartient désormais au lecteur de choisir lequel de ces deux outils, Rebar3 ou Erlang.mk, deviendra son préféré. Il ne faudra pas oublier une chose : ces projets permettent de faire bien d'autres choses et la documentation reste une très bonne source d'information pour les personnes qui seraient tentées par ces solutions.

6. HEX

Un article parlant des applications, des releases et des gestionnaires de projets en Erlang ne pouvait pas se faire sans une petite note sur **Hex.pm [11]**. Comme de nombreux langages présents sur le marché, la communauté a mis en place il y a de ça quelques années un site permettant de recenser les différents outils disponibles. Ce site permet donc de diffuser les modules créés par la communauté. Il permet aussi de créer des dépôts publics ou privés pour les créateurs de contenu, ainsi qu'un système de recherche assez pratique. Pour les plus curieux, il est bon de savoir que Hex.pm n'est pas réservé exclusivement à Erlang, mais est aussi disponible pour les utilisateurs du langage Elixir.

7. GESTIONNAIRE DE CACHE ET APPLICATION

Maintenant que le lecteur est au courant des différentes suites d'outils offertes par la communauté, il est temps de migrer et d'adapter le projet de cache développé depuis le début des articles. Pour que l'exemple soit réellement probant, une arborescence complètement compatible avec Rebar3 sera créée manuellement et chacune des étapes sera expliquée pour étayer les différentes informations qui ont été vues précédemment.

La création d'une arborescence commence par la création des répertoires. Pour ce faire, trois répertoires sont essentiels, le répertoire « racine » qui sera nommé d'après le nom du projet. Dans le cadre de cet exemple, il aura pour nom **cache**. Le sous-répertoire **src** contiendra les sources de l'application en cours de développement. Pour finir, le sous-répertoire **test** contiendra les tests basés sur les frameworks de test **eunit** ou **common_test**.

```
$ mkdir -p cache cache/test cache/src
```

Les fichiers **cache_server.erl** ainsi que **cache_sup.erl** font référence aux fichiers créés dans le précédent article **[2]** et peuvent-être simplement copiés et/ou déplacés dans le répertoire **cache/src**. Ils ne sont pas à modifier et permettront d'être le « cœur » de l'application.

```
$ mv ${origine}/cache.erl cache/src/cache.erl
$ mv ${origine}/cache_sup.erl cache/src/
cache_sup.erl
```

Le fichier **cache_app.erl** va utiliser le behaviour **application** et ainsi pouvoir créer l'application **cache**.

```
$ touch cache/src/cache_app.erl
```

Le contenu de **cache_app.erl** contiendra alors le nom de l'application, **cache_app**. Le callback **start/2** démarre le module **cache_sup** en utilisant la fonction **supervisor:start_link/2** et retourne alors l'identifiant de processus en cas de succès. Le callback **stop/1** ne réalise aucune action particulière, et se contente simplement d'arrêter implicitement le service.

```
-module(cache_app).
-behaviour(application).
-export([start/2, stop/1]).

start(_Type, _Options) ->
  supervisor:start_link(cache_sup, []).

stop(_Etat) ->
  ok.
```

Pour que l'application soit complète et fonctionnelle, le fichier **cache.app.src** doit alors être créé et va contenir les besoins de l'application, tels que les dépendances applicatives, de bibliothèques ou encore définir l'ordre de démarrage de certains services.

```
$ touch cache/src/cache.app.src
```

Le fichier en question devrait contenir les informations essentielles permettant de démarrer l'application cache. Étant donné que l'application en question est rudimentaire, seuls les outils essentiels sont à rajouter. Le code en question devrait ressembler à ça :

```
{application, cache,
 [{description, "A simple cache application"},
  {vsn, "0.1.0"},
  {registered, [cache_sup, cache]},
  {mod, {cache_app, []}},
  {applications,
   [kernel,
    stdlib
   ]},
 {env,[]},
 {modules, []},
 {licenses, ["Apache 2.0"]},
 {links, []}
]}.
```

Finalement, pour que toute cette arborescence soit compatible avec Rebar3, le fichier **rebar.conf** doit être créé et contiendra la configuration relative au gestionnaire de projets.

```
$ touch cache/rebar.config
```

Comme vu précédemment, voici un exemple du contenu :

```
{erl_opts, [debug_info]}.
{deps, []}.
{shell, [
    {apps, [cache]}
]}.
```

Pour les personnes qui souhaiteraient diffuser ce code ailleurs, comme sur **GitHub** ou **Gitlab**, un fichier de **LICENCE** ainsi qu'un fichier de **README** peuvent être créés.

```
$ touch cache/LICENCE
$ touch cache/README.md
```

L'application est maintenant prête à être compilée avec Rebar3. Pour ce faire, la sous-commande **compile** est à utiliser :

```
$ cd cache
$ rebar3 compile
===> Verifying dependencies...
===> Compiling cache
```

Le résultat de la compilation se trouve alors dans le répertoire **_build**, dynamiquement créé lors de l'exécution de la commande précédente. Dans tous les cas, il est possible d'utiliser la sous-commande **shell** pour contrôler si tout est correctement fonctionnel.

```
$ rebar3 shell
===> Verifying dependencies...
===> Compiling cache
===> The rebar3 shell is a development tool; to
deploy applications in production, consider using
releases (http://www.rebar3.org/docs/releases)
===> Booted cache
Erlang/OTP 21 [erts-10.2] [source] [64-bit]
[smp:2:2] [ds:2:2:10] [async-threads:1]

Eshell V10.2  (abort with ^G)
```

Il semblerait que l'application soit correctement démarrée, pour le vérifier, la commande **application:which_application/0** devrait offrir la réponse recherchée. Effectivement, l'application **cache** est présente sur le nœud, elle contient bien la description configurée dans le fichier de ressources, tout comme la version.

```
1> application:which_applications().
[{cache,"A simple cache application","0.1.0"},
 {inets,"INETS  CXC 138 49","7.0.3"},
 {ssl,"Erlang/OTP SSL application","9.1"},
 {public_key,"Public key
infrastructure","1.6.4"},
 {asn1,"The Erlang ASN1 compiler version
5.0.8","5.0.8"},
 {crypto,"CRYPTO","4.4"},
 {stdlib,"ERTS  CXC 138 10","3.7"},
 {kernel,"ERTS  CXC 138 10","6.2"}]
```

Qu'en est-il des différents processus démarrés sur le nœud ? Est-ce que le gestionnaire de cache fonctionne ? Il suffit pour cela d'utiliser la fonction **cache:add/3**, puis **cache:get/2** pour le savoir...

```
2> cache:add(cache, test, 1).
ok
3> cache:get(cache, test).
1
```

L'application **cache** est maintenant compatible avec Rebar3 et peut utiliser toutes les fonctionnalités de ce gestionnaire de projets. Il est désormais possible de déployer l'application, de l'exporter proprement vers un dépôt ou encore de l'envoyer sur Hex.pm.

CONCLUSION

Une étape supplémentaire vient d'être franchie. La gestion d'un projet est une épreuve, mais permet sur le long terme d'avoir une structure compréhensible par un grand nombre. Cet article n'a montré que peu de code en Erlang, mais était nécessaire pour les articles qui viendront par la suite. Effectivement, la création des tests, mais aussi la conception de produits plus complexes, passe obligatoirement par une façon de ranger les différentes données créées par le développeur.

Le lecteur a maintenant une vision plus précise du concept d'application, de release et de projet sous Erlang. Il est évident que ceci n'est qu'une introduction, et que la compréhension de l'ensemble nécessite de la pratique. Comme dans tous les autres langages, Erlang ne faisant pas exception, la création de contenu est un processus long et nécessitant de la patience ainsi que de la rigueur, condition sine qua non pour atteindre un niveau d'excellence. ■

RÉFÉRENCES

[1] M. KERJOUAN, « Erlang, programmation distribuée et modèle acteur », GNU/Linux Magazine n°237, mai 2020 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-237/Erlang-programmation-distribuee-et-modele-acteur

[2] M. KERJOUAN, « Système extensible et hautement disponible avec Erlang/OTP », GNU/Linux Magazine n°241, octobre 2020 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-241/Systeme-extensible-et-hautement-disponible-avec-Erlang-OTP

[3] Site officiel d'Erlang : https://erlang.org

[4] Documentation officielle d'Erlang : https://erlang.org/doc

[5] Documentation du behaviour Application : https://erlang.org/doc/man/application.html

[6] Documentation du fichier de ressources applicatif : https://erlang.org/doc/man/app.html

[7] Design des applications en Erlang : https://erlang.org/doc/design_principles/applications.html

[8] Design des releases en Erlang : https://erlang.org/doc/design_principles/release_structure.html

[9] Site officiel de Rebar3 : https://www.rebar3.org

[10] Site officiel d'Erlang.mk : https://erlang.mk

[11] Site officiel d'Hex.pm : https://hex.pm

[12] Code source des exemples : https://github.com/niamtokik/linux-mag

POUR ALLER PLUS LOIN

Les personnes désirant en savoir plus sur la gestion de projets en Erlang peuvent se tourner vers les différents liens se trouvant dans la partie Références. La documentation officielle est une source d'exemples précieux qui permettront de donner une plus grande profondeur à cet article. Le livre « Designing for Scalability with Erlang/OTP » écrit par Francesco Cesarini et Steve Vinoski fait partie de la littérature à posséder pour avoir une vue d'ensemble de la situation, tout en profitant des meilleures pratiques. Un dernier mot pour la fin, le code source est mis à disposition du lecteur via le compte GitHub [12] de l'auteur.

LISEZ DÈS À PRÉSENT TOUS NOS MAGAZINES EN LIGNE !

Simple. Rapide. Pratique.

FLIPBOOK HTML5

Tous nos magazines ont leur version Flipbook :

LINUX MAGAZINE / FRANCE · MISC · LINUX PRATIQUE · HACKABLE MAGAZINE

Rendez-vous sur www.ed-diamond.com

RENDRE UNE PAGE PRÉSENTANT DU CODE PLUS ERGONOMIQUE AVEC TAMPERMONKEY/ GREASEMONKEY

TRISTAN COLOMBO

MOTS-CLÉS : TAMPERMONKEY, GREASEMONKEY, JAVASCRIPT

Lire des articles contenant du code informatique sur le Web n'est pas nécessairement simple, ne serait-ce que de par la complexité inhérente au sujet traité. Pourquoi alors se compliquer la tâche avec une présentation des informations qui n'est pas nécessairement adaptée au contenu, alors qu'on peut l'améliorer ?

Les « ergonomes » des agences web se donnent souvent des titres pompeux, sans avoir suivi le moindre début de formation. N'ayant moi-même que quelques notions d'ergonomie, je n'ai pas la prétention d'affirmer que nous pouvons partir d'une page web mal conçue pour la rendre parfaite, mais nous allons tenter de pallier certains défauts de conception vendus par des « spécialistes » de l'ergonomie et du design web.

Pour cela, nous allons nous intéresser à des pages présentant du code en nous basant sur un modèle imaginaire (figure 1) regroupant des défauts présents sur de nombreux sites. Cet exemple aura l'avantage de nous permettre d'avoir un aperçu de nombreuses modifications qui seront transposables par la suite à toutes sortes de sites.

```
# Programme principal
if __name__ == '__main__':
    # Récupération du nom de fichier dans les paramètres
    filename = getFilename()
    # Lecture et affichage du fichier
    readFile(filename)
```

Ce programme est très simple. On peut noter :

- une fonction `getFilename()` qui ...
- une fonction `readFile()` qui ...
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

II Deuxième partie

*Lorem ipsum*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

II.I Sous-partie

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Fig. 1 : Exemple imaginaire d'un site présentant des articles contenant du code informatique.

Nous allons utiliser **Tampermonkey** (ou **Greasemonkey** en fonction de votre navigateur) pour retraiter les pages localement. Voici les différents éléments que nous pouvons améliorer (certains des points cités ne sautent pas forcément aux yeux sur la figure, du fait du changement d'échelle) :

- la police du texte est trop petite et le gris n'est pas la couleur la plus lisible ;
- le texte en gras et en italique ne saute pas aux yeux ;
- les couleurs de fond du code et les couleurs des titres ne sont pas du meilleur goût ;
- les puces de l'énumération pourraient être plus jolies ;
- une coloration syntaxique du code serait utile ;
- un bouton permettant de copier le contenu d'un bloc de code d'un simple clic serait vraiment pratique, de même qu'un bouton permettant de récupérer les références de l'article pour pouvoir le citer ;
- certains éléments ne sont pas cliquables (par exemple, une référence notée **[1]** pourrait renvoyer vers la référence en question en bas d'article et un clic sur la référence ramener vers la première citation de celle-ci).

NOTE

Je ne reviendrai pas dans cet article sur les bases de l'utilisation de Tampermonkey/Greasemonkey que vous pourrez retrouver dans **[1]**.

1. LE CODE DE DÉPART

Pour commencer, il nous faut ajouter un nouveau script en nous rendant sur la page cible et en cliquant sur l'icône de Tampermonkey dans la barre d'outils, puis sur ***Ajouter un nouveau script...***. Nous obtenons alors le code suivant qu'il faut légèrement modifier pour lui attribuer un nom, une description et un auteur :

```
// ==UserScript==
// @name         Modernize Code Pages
// @namespace    http://tampermonkey.net/
// @version      0.1
// @description  Change design of web pages containing code
// @author       Tristan Colombo <tristan@gnulinuxmag.com>
// @match        https://page-cible
// @grant        none
// ==/UserScript==

(function() {
    'use strict';

    // Your code here...
})();
```

Si la page que l'on modifie ne charge pas déjà un framework JavaScript permettant de simplifier l'écriture de code, alors il va falloir s'en charger. Nous

pouvons par exemple jeter notre dévolu sur **JQuery [2]** en utilisant un CDN (Content Delivery Network) :

```
...
// @match        https://page-cible
// @require      https://code.jquery.com/jquery-3.5.1.min.js#sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=
// @grant        none
...
```

Ceci fait, nous pouvons commencer à modifier la page.

2. MODIFICATION DU STYLE DU TEXTE

La police du texte est trop petite, mais celle du code n'est pas nécessairement plus adaptée. Pour le code, la police **Cascadia Code [3]** aurait été une bonne solution, mais il n'y a malheureusement pas de fichier WOFF ou OTF disponible en ligne. Deux solutions seraient possibles : un chargement local et une mise à disposition sur un serveur pour un téléchargement distant. La première solution serait bloquante pour une distribution facile de notre script et la seconde solution est un peu lourde à mettre en place pour une simple police... nous pouvons en choisir une autre parmi les **Google Fonts [4]**. Mon choix s'est porté sur la police **Source Code Pro** et pour le texte, j'aime bien la police **Lora**. Le chargement de ressources externes implique de retirer la ligne `// grant none` pour ajouter `// @grant GM_addStyle` et `// @grant GM_getResourceText` permettant l'accès aux fonctions de chargement des CSS :

```
// ==UserScript==
// @name         Modernize Code Pages
...
// @resource     SourceCodeFont    https://fonts.googleapis.com/css2?family=Source+Code+Pro
// @resource     TextFont          https://fonts.googleapis.com/css2?family=Nora
// @grant        GM_addStyle
// @grant        GM_getResourceText
// ==/UserScript==

var sourceCodeFontCSS = GM_getResourceText("SourceCodeFont");
GM_addStyle(sourceCodeFontCSS);


var textFontCSS = GM_getResourceText("TextFont");
GM_addStyle(textFontCSS);
...
```

Ces lignes nous permettent de charger les polices, mais nous ne les utilisons pas encore.

ATTENTION !

Si lors de l'appel à `GM_getResourceText()`, vous passez en paramètre une chaîne ne correspondant pas au nom de la ressource définie précédemment, vous n'obtiendrez aucun message d'erreur dans la console des Outils de Développement de votre navigateur ! Typiquement, si vous tapez `sourceCodeFont` au lieu de `SourceCodeFont` (`s` au lieu de `S`), la police Source Code Pro ne sera pas chargée.

Il faut repérer dans le code les classes ou tags qui permettront de propager la modification. Ici, il s'agira simplement de `<p>` pour le texte et de la classe `.code` pour le code. Nous en profiterons pour changer la couleur du texte dans un gris très foncé, bien plus lisible, et la couleur de fond du code par un gris clair moins agressif que le vert de la figure 1 :

```
...
(function() {
    'use strict';

    var $ = window.$;

    $('.code').css('font-family', 'Source Code Pro');
    $('.code').css('font-size', '1.2em');

    $('p').css('font-family', 'Lora');
    $('p').css('font-size', '1.2em');
    $('p').css('color', '#373832');
})();
```

Les modifications portant sur les titres (`<h1>`, `<h2>`, etc.) ou encore le style de la mise en gras ou en italique passeront par des lignes de code semblables aux précédentes :

```
...
    $('h1').css('font-family', 'Lora');
    $('h1').css('color', '#d4210c');

...
```

3. MODIFICATION DES PUCES DANS LES LISTES

Pour modifier les puces, on peut passer par la solution simple consistant à modifier la propriété CSS **list-style-type** en indiquant un style prédéfini (**disc**, **circle**, etc.) ou un caractère (avec l'UTF-8, on peut avoir des glyphes). Le problème de cette solution est qu'un changement de couleur de la puce entraîne un changement de couleur du texte. Pour pouvoir effectuer cette opération, nous allons devoir utiliser la pseudoclasse **:before** sur **li**, ce qui va nous permettre de voir une nouvelle syntaxe JQuery, car les pseudoclasses ne sont pas accessibles via la fonction **css()** :

```
...
    $('li').css('list-style-type', 'none');
    $('head').append('<style>li:before{content:
"\u21e2 "; color: #d4210c;}</style>');
...
```

Nous devons désactiver les puces pour pouvoir ensuite les redéfinir dans le header du document entre des balises de style.

À ce point, notre page a déjà bien changé, comme le montre la figure 2.

NOTE

Si vous souhaitez exécuter un script sur une page locale, vous devrez vous rendre dans les paramètres de l'extension dans votre navigateur. Dans Google Chrome, il faut cliquer sur ***Plus d'outils > Extension*** et sur la page de Tampermonkey et activer ***Autoriser l'accès aux URL de fichiers***.

4. COLORATION SYNTAXIQUE DU CODE

Il existe de nombreux scripts JavaScript de coloration syntaxique du code tels que **Code-Prettify [5]** (qui n'est plus maintenu), **highlight.js [6]** ou encore **Prism.js [7]**. Ces

```
# Programme principal
if __name__ == '__main__':
    # Récupération du nom de fichier dans les paramètres
    filename = getFilename()
    # Lecture et affichage du fichier
    readFile(filename)
```

Ce programme est très simple. On peut noter :

- une fonction **getFilename()** qui ...
- une fonction **readFile()** qui ...
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

II Deuxième partie

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Fig. 2 : La page en cours de modification.

scripts ont en commun de pouvoir être appliqué sur un document HTML correctement formé, c'est-à-dire où le code source est présenté sous la forme :

```
<pre>
    <code>
        ... code source ...
    </code>
</pre>
```

Or, ceci n'est pas forcément respecté par tous les sites. Prenons l'exemple où les blocs de code ont la forme suivante :

```
<p>Texte précédent le code</p>
<div class="code"># Programme principal</div>
<div class="code">if __name__ == '__main__':</div>
...
<p>Fin du bloc de code et nouveau bloc de texte</p>
```

Il faut récupérer le contenu des **div** de classe **code** et les regrouper dans des tags **<pre><code>...</pre></code>** :

```
...

function prettifyCode() {
    $(':not(div.code) + div.code, * > div.
code:first-of-type').
    each(function() {
        $(this).
            nextUntil(':not(div.code)').
            addBack().
```

```
        wrapAll('<pre />').
        wrapAll('<code />');
});
$('div.code').each(function() {
    $(this).text($(this).text() + '\n');
    $(this).contents().unwrap();
});
$('pre').css('white-space', 'pre-wrap');
$('pre').addClass('code');

    $('.code').css('font-family', 'Source Code Pro');
    $('.code').css('font-size', '1.2em');
    $('.code').css('background', '#f0f0f0');
}


(function() {
    'use strict';

    $(document).ready(function() { prettifyCode(); });
    ...
})();
```

Vous noterez que le jeu des feuilles de style en cascade fait qu'il faut déplacer les 3 lignes de modification des CSS de la classe **.code** dans la fonction **prettifyCode()**.

Il faut maintenant choisir un script de coloration syntaxique. Prism.js est de loin le plus beau... mais il ne dispose pas d'une fonctionnalité de détection automatique du langage (il faut ajouter une classe **language-xxx** où xxx correspond au nom du langage employé dans le bloc). Highlight.js, bien que plus rudimentaire, dispose de cette fonctionnalité et c'est donc celui-ci que j'ai choisi. L'utilisation d'un CDN permet une écriture rapide du code en sélectionnant l'un des thèmes proposés sur **[8]** (j'ai choisi le thème **GoogleCode**) :

```
...
// @require      https://cdnjs.cloudflare.com/ajax/libs/highlight.
js/10.3.1/highlight.min.js#sha512-U12+KlhI3X2EY7U4NJZ+O0wujKcaMQZHAB
taiZtE8UrPiK1O3Y4cjBe0mMFyyBptdaf+eh45hqNdsayeLQcneg==
// @resource     SourceCodeFont    https://fonts.googleapis.com/
css2?family=Source+Code+Pro
// @resource     HljsCss           https://cdnjs.cloudflare.com/
ajax/libs/highlight.js/10.3.1/styles/googlecode.min.css#sha512-6eDOw
D0VUB7xNznn4o4iXE2macfjgv1buz6o8ITvrf45pTBVmvbRctkKl2OmNX0vKbzt45DS0
xbzHNL+ZZwVNA==
...
// ==/UserScript==


...

var hljsCSS = GM_getResourceText("HljsCss");
GM_addStyle(hljsCSS);
```

```
...

function prettifyCode() {
    ...

    hljs.initHighlighting();
}
...
```

La page commence à devenir lisible (figure 3) !

5. AJOUT D'UN BOUTON POUR COPIER UN BLOC DE CODE

Maintenant que nous avons modifié le code des blocs de code de manière à ce qu'il ait une structure plus rigoureuse, l'ajout d'un bouton ne va pas poser beaucoup de problèmes. Il suffit d'ajouter un nœud **button** avant les tags **pre** de classe **code**. Chaque tag **pre** se verra ajouter un nom de classe permettant d'identifier de manière unique un bloc de code et qui sera transmis à une fonction **copyToClipBoard()** en charge d'effectuer la copie en mémoire. Cette fonction ne pourra pas être écrite de manière conventionnelle, car comme elle est appelée depuis la page, elle doit être accessible depuis celle-ci et donc chargée à l'intérieur de son code :

```
...

function prettifyCode() {
    var n = 1;


    ...


    $('pre').each(function() {
        $(this).
addClass('code_' + n);
        n += 1;
    });
```

```
# Programme principal
if __name__ == '__main__':
    # Récupération du nom de fichier dans les paramètres
    filename = getFilename()
    # Lecture et affichage du fichier
    readFile(filename)
```

Ce programme est très simple. On peut noter :

- une fonction **getFilename()** qui ...
- une fonction **readFile()** qui ...
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Fig. 3 : Modification de la page après coloration syntaxique.

```
    ...

    hljs.initHighlighting();

    n = 1;
    $('pre.code').each(function() {
        $(this).before($('<button class="rawCode" onclick="copyToClip
Board(\'code_' + n + '\')">Raw copy</button>'));
        n += 1;
    });
}


(function() {
    'use strict';

    ...

    function main() {
        window.copyToClipBoard = function(idCode) {
            var clipboard = document.createElement('textarea');
            clipboard.value = document.getElementsByClassName(idCode)
[0].innerText;
            clipboard.setAttribute('readonly', '');
            clipboard.style = {position: 'absolute', left: '-9999px'};
            document.body.appendChild(clipboard);
            clipboard.select();
            document.execCommand('copy');
            document.body.removeChild(clipboard);
        }
    };

    var script = document.createElement('script');
    script.appendChild(document.createTextNode('('+ main +')();'));
    $('head').append(script);
})();
```

Vous pouvez voir dans le code précédent que la fonction **copyToClipBoard()** est déclarée globalement (**window.copyToClipBoard**) et qu'elle est enfouie dans une autre fonction (**main()**). Un nouveau nœud **script** est créé, puis inséré dans le header. Ce script exécute **main()** de manière à déclarer **copyToClipBoard()** qui sera ainsi accessible par **onclick** sur les boutons.

6. COPIE DE LA RÉFÉRENCE D'UN ARTICLE POUR CITATION

Ce cas sera traité exactement comme le cas précédent : nous ajoutons un bouton et nous exécutons une fonction JavaScript qui va parcourir le DOM à la recherche des informations à copier dans le presse-papier. Pour pouvoir réutiliser la fonction que vous avons écrite précédemment, il va falloir la modifier afin que le paramètre corresponde soit au nom de la classe de l'élément dont on souhaite récupérer le contenu, soit simplement à la chaîne de caractères que l'on souhaite stocker dans le presse-papier. La solution retenue est de faire commencer une chaîne par le caractère **:** pour signaler qu'il ne s'agit pas d'un nom de classe :

```
...

(function() {
    'use strict';

    ...

    $('h1').after($('<div
class="ref"><button class="getRef" on
click="getReference(copyToClipBoard)"
>Ref</button></div>'));
    $('.ref').css('float', 'right');

    function main() {
        window.copyToClipBoard =
function(idCode) {
            var clipboard = document.
createElement('textarea');
            if (idCode[0] == ':') {
                clipboard.value =
idCode.slice(1);
```

```
            } else {
                clipboard.value = document.
getElementsByClassName(idCode)[0].innerText;
            }
            clipboard.setAttribute('readonly', '');
            clipboard.style = {position: 'absolute', left:
'-9999px'};
            document.body.appendChild(clipboard);
            clipboard.select();
            document.execCommand('copy');
            document.body.removeChild(clipboard);
        };

        window.getReference = function(cpClip) {
            let reference = "";
            try {
                let title = document.
getElementsByClassName("block_content")[0].children[0].innerHTML;
                let author = document.
getElementsByClassName("field--item")[7].children[0].innerHTML;
                author = author.split(' ')
                let magazine = document.
getElementsByClassName("field--item")[4].innerHTML;
                let number = document.
getElementsByClassName("field--item")[5].innerHTML;
                let date = document.
getElementsByClassName("field--item")[6].innerHTML;
                reference = author[1][0] + ". " + author[0].
toUpperCase() +', \u00ab\u00a0' + title + '\u00a0\u00bb, ' +
magazine + ' n°' + number + ', ' + date + '\u00a0: ' + window.
location.href;
                reference = reference.replace(/\n/g, "");
            }
            catch (error) {
                return "";
            }

            cpClip(':' + reference);
        }
    };

    ...
})();
```

Vous remarquerez que l'appel à **copyToClipBoard()** dans **getReference()** n'est pas un appel direct : un pointeur est passé en paramètre à la fonction (**cpClip**), car un appel direct depuis l'action **onclick** du bouton, appelant **getReference()** qui elle-même devait appeler **copyToClipBoard()**, est impossible.

Ici, nous récupérons les données qui nous intéressent et nous les stockons simplement dans le presse-papier. Si vous utilisez des logiciels de gestion de références bibliographiques tels que **Mendeley [9]**, **Zotero [10]** ou encore **EndNote**, il peut être intéressant d'ajouter ces données sous forme de meta-tags de manière à ce que la détection se fasse automatiquement. Mendeley propose ainsi une page **[11]** décrivant les données à ajouter aux pages pour une détection automatique des articles.

Étrangement, il n'est pas possible de s'appuyer sur du contenu détecté automatiquement pour l'enrichir des données manquantes. Dans mon cas par exemple, le titre de l'article et l'URL étaient correctement détectés. L'ajout du nom de l'auteur seulement n'est pas prise en compte :

```
...
    var author = document.
createElement('meta');
        author.
setAttribute('name',
'citation_authors');
        author.
setAttribute('content',
'John Doe');
        $('head').
append(author);
...
```

Par contre, si l'on ajoute également un titre, Mendeley détectera deux références : la référence « automatique » et celle que nous avons renseignée (figure 4).

```
...
   var title = document.
createElement('meta');
   title.setAttribute('name',
'citation_title');
   title.setAttribute('content',
'Title');
   $('head').append(title);
...
```

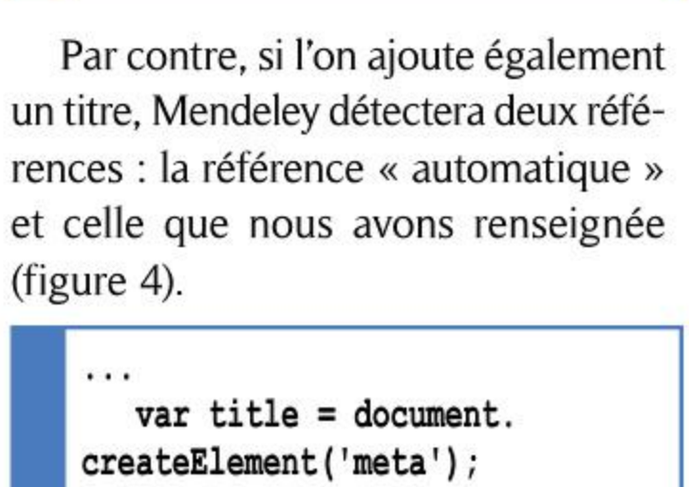


Fig. 4 : Détection d'une référence par Mendeley grâce à l'ajout de tags meta.

7. NAVIGATION PARMI LES RÉFÉRENCES

J'ai conservé le point le plus problématique pour la fin. Les références apparaissant sous la forme **[n]** où **n** est un nombre, elles peuvent être confondues avec un accès à un tableau dans un code. Comme nous avons prétraité le code HTML pour que le code soit intégré dans des balises **<pre /><code />**, il devrait suffire de parcourir les tags **p**, d'y détecter les références pour les transformer en liens internes, puis d'effectuer un traitement équivalent en bas de page, avec renvoi vers la première apparition de la référence sur la page. Cela va faire un petit peu plus de code que précédemment :

```
...

function linksToRefs() {
    const regex = /\[(\d+)\]/g;
    $('p').each(function() {
        var content = $(this).text();
        var text_content = $(this).text();
        var refs = null;
        var result = '';
        while ((refs = regex.exec(content)) !=
null) {
            var n = refs[1];
            var i = refs.index + n.length + 1;

            if (text_content.charAt(0) == '[') {
                result = '<a class="reference"
name="ref_' + n + '" href="#cite_' + n + '">['
+ n + ']</a>' + content.substr(i + 1, content.
length);
            } else {
                result = content.substr(0, refs.
index) + '<a class="reference" name="cite_' +
n + '" href="#ref_' + n + '">[' + n + ']</a>' +
content.substr(i + 1, content.length);
            }
            $(this).html(result);
        };
    });
}


(function() {
    'use strict';
    ...
```

```
    linksToRefs();
    $('html').css('scroll-
behavior', 'smooth');
    $('.reference').css('font-
weight', 'bold');
})();
```

C'est la fonction **linksToRefs()** qui est chargée de détecter les références et d'y ajouter des liens. Pour cela, une simple expression régulière **/\([\d+\])/** suffit. On parcourt chaque paragraphe **p** (**$('p').each()**), on récupère le contenu HTML (**content**) et textuel (**text_content**). **content** permet de rechercher les positions des références dans le code HTML et **text_content** permet de retrouver simplement les références de bas de page qui commencent par le caractère **[**. En appliquant l'expression régulière à **content**, on obtient la position du **[** (**refs.index**) et l'élément capturé par les parenthèses de l'expression régulière (**refs[1]**).

NOTE

Si une expression régulière définit plusieurs motifs de capture, l'indice du tableau permettra de récupérer la énième capture. Par exemple, avec **const regex = /\[(\d+)\]>(\w+)</g;** un appel à **refs = regex.exec(content)** permettra de rechercher dans la chaîne de caractères **regex** et **refs[1]** fera référence à **(\d+)**, alors que **refs[2]** fera référence à **(\w+)**.

Une fois les références identifiées, il n'y a plus qu'à remplacer dans le code HTML leur apparition par un lien interne au document (la méthode **substr()** est utilisée pour isoler les parties à conserver en indiquant en paramètre la position du premier caractère et celle du dernier). Pour rappel, un lien interne se fait en nommant des ancres auxquelles on peut ensuite faire référence. Par exemple, on peut scroller vers **[1]** à partir d'un lien **Aller à [1]**.

Pour finir, après l'appel à **linksToRef()**, la règle **scroll-behavior** est fixée à **smooth** pour que le scrolling se fasse de manière progressive.

CONCLUSION

En quelques lignes de code, nous avons amélioré l'ergonomie et la lisibilité d'un site. Si le « designer » est connu, il n'y a plus qu'à lui transmettre la preuve de concept pour qu'il l'adapte au site et que le script Tampermonkey puisse être désactivé. Il pourra ainsi se faire rémunérer pour votre travail et ne pas manquer de critiquer allègrement votre vision des choses, qui détruit totalement une présentation savamment réfléchie... ou pas. Quoi qu'il en soit, voici une nouvelle preuve de la liberté offerte par cette formidable extension qu'est Tampermonkey ! ■

RÉFÉRENCES

[1] T. COLOMBO, « Modifiez l'ergonomie d'une page web avec Tampermonkey », GNU/Linux Magazine n°213, mars 2018 : https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-213/Modifiez-l-ergonomie-d-une-page-web-avec-Tampermonkey

[2] Framework JavaScript JQuery : https://jquery.com/

[3] Police Cascadia Code : https://github.com/microsoft/cascadia-code

[4] Polices Google Fonts : https://fonts.google.com/

[5] Code-Prettify : https://github.com/googlearchive/code-prettify

[6] Highlight.js : https://highlightjs.org/

[7] Prism.js : https://prismjs.com/

[8] Liste des CDN highligh.tjs : https://cdnjs.com/libraries/highlight.js

[9] Mendeley : https://www.mendeley.com

[10] Zotero : https://www.zotero.org/

[11] Format des pages HTML pour une détection automatique des références par Mendeley : https://www.mendeley.com/guides/information-for-publishers

BON DE SOUTIEN RÉGULIER

À RENVOYER À L'ADRESSE :
HANDICAP INTERNATIONAL LIBRE RÉPONSE N° 45134 69129 LYON 08

☐ **OUI,** je souhaite soutenir dans la durée les actions de Handicap International

Je choisis le montant de mon soutien
☐ **10€/mois** soit 2,5€ après réduction fiscale
☐ **20€/mois** soit 5€ après réduction fiscale
☐ **30€/mois** soit 7,5€ après réduction fiscale

Le premier pélèvement devra avoir lieu le 10 du mois de **2021**
Je pourrai faire suspendre l'exécution de ce prélèvement par simple demande à l'association, signifiée au plus tard le 20 du mois précédent.
Je recevrai un reçu fiscal annuel pour l'ensemble de mes dons.

MANDAT DE PRÉLÈVEMENT SEPA Désignation du compte à débiter **(merci de joindre un RIB)** :

IBAN |_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|

BIC |_|_|_|_|_|_|_|_|_|_|_|

Signature :
(obligatoire)

Fait à .....
le |_|_| / |_|_| / |_|_|_|_|

Merci de compléter vos informations ci-dessous :

○ M. ○ Mme ○ Mlle

Nom : .....

Prénom : .....

E-mail : .....@.....
votre adresse e-mail sera utilisée exclusivement par Handicap International

Adresse : .....

.....

Code postal : |_|_|_|_|_| **Ville :** .....

En signant ce formulaire de mandat, vous autorisez Handicap International à envoyer des instructions à votre banque pour qu'elle débite votre compte et votre banque à débiter votre compte conformément aux instructions de Handicap International. Vous bénéficiez du droit d'être remboursé(e) par votre banque selon les conditions décrites dans la convention que vous avez passée avec elle. Toute demande de remboursement doit être présentée dans les 8 semaines suivant la date de débit de votre compte pour un prélèvement autorisé et au plus tard dans les 13 mois en cas de prélèvement non autorisé. Vos droits concernant le présent mandat sont expliqués dans un document que vous pouvez obtenir auprès de votre banque. Les informations demandées sont enregistrées dans un fichier informatisé par Handicap International qui dispose d'un délégué à la protection des données (dpo). Elles sont nécessaires pour répondre à vos demandes ou faire appel à votre générosité. Elles sont conservées pendant la durée strictement nécessaire à la réalisation des finalités précitées. Conformément à la loi Informatique et Libertés et à la réglementation européenne, en vous adressant à donateurs@france.hi.org, vous bénéficiez d'un droit d'accès, rectification, limitation, portabilité, effacement et opposition à l'utilisation de vos données à caractère personnel. En cas de difficulté, vous pouvez introduire une réclamation auprès de la CNIL.

COUJ

EUROPEAN CYBER CUP

LA 1ÈRE COMPÉTITION DE ESPORT DÉDIÉE AU HACKING ÉTHIQUE

LE 7 & 8
AVRIL 2021
À LILLE
GRAND PALAIS

TO BE [PWNED]
OR NOT TO BE [PWNED]
THAT IS THE {CYBER}QUESTION

WWW.EUROPEAN-CYBERCUP.COM

Organisé par :

FIC 2021

ZOSD

En partenariat avec :

CYBERSÉCURITÉ OFFENSIVE & DÉFENSIVE
MISC
PAPIER + BASE DOCUMENTAIRE ONLINE

Avec le soutien de :

MEL
MÉTROPOLE
EUROPÉENNE DE LILLE