Découvrir la programmation avec Rust

Rust est un langage de programmation qui a gagné en popularité ces dernières années. En 2006, un employé de Mozilla a commencé un projet personnel, qui a ensuite reçu le soutien de l'entreprise et a été publié en 2010. Ce projet s'appelle Rust.

Ensuite, lançons le premier programme en Rust. Ouvrez le site officiel et voyons comment exécuter un programme.

Le site officiel fournit un script :

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

Sur Mac, vous pouvez également utiliser Homebrew, le gestionnaire de paquets du système Mac, pour l'installation. Vous pouvez exécuter la commande suivante :

```
brew install rust
```

Je vais utiliser Homebrew pour installer Rust. Pendant l'installation, continuons à consulter le site officiel.

Ensuite, nous voyons que le site officiel mentionne Cargo, un outil de construction et de gestion de paquets pour Rust.

Le site officiel indique :

- construisez votre projet avec `cargo build`
- exécutez votre projet avec `cargo run`
- testez votre projet avec `cargo test`

Nous explique comment construire, exécuter et tester un programme Cargo.

Exécuter :

```
brew install rust
```

Sortie :

```
==> Téléchargement de https://homebrew.bintray.com/bottles/rust-1.49.0_1.big_sur.bottle.tar.gz
==> Téléchargement depuis https://d29vzk4ow07wi7.cloudfront.net/5a238d58c3fa775fed4e12ad74109deff54a82a
######################################################################## 100.0%
```

```
==> Extraction de rust-1.49.0_1.big_sur.bottle.tar.gz
==> Avertissements
La complétion Bash a été installée dans :
  /usr/local/etc/bash_completion.d
==> Résumé
   /usr/local/Cellar/rust/1.49.0_1: 15 736 fichiers, 606,2 Mo
```

Cela signifie que l'installation a réussi.

Lorsque vous exécutez `cargo` dans le terminal, la sortie est la suivante :

```
Le gestionnaire de paquets de Rust
```

UTILISATION : cargo [OPTIONS] [SOUS-COMMANDE]

OPTIONS : -V, –version Affiche les informations de version et quitte –list Liste les commandes installées –explain Exécute `rustc --explain CODE` -v, –verbose Utilise une sortie détaillée (-vv très détaillée/sortie de build.rs) -q, –quiet Aucune sortie imprimée sur stdout –color Coloration : auto, toujours, jamais –frozen Exige que Cargo.lock et le cache soient à jour –locked Exige que Cargo.lock soit à jour –offline Exécute sans accéder au réseau -Z ... Options instables (uniquement pour les versions nightly) pour Cargo, voir 'cargo -Z help' pour plus de détails -h, –help Affiche les informations d'aide

Voici quelques commandes courantes de Cargo (vous pouvez voir toutes les commandes avec `--list`) : build, b Compile le package actuel check, c Analyse le package actuel et signale les erreurs, mais ne génère pas les fichiers objets clean Supprime le répertoire target doc Génère la documentation du package actuel et de ses dépendances new Crée un nouveau package Cargo init Crée un nouveau package Cargo dans un répertoire existant run, r Exécute un binaire ou un exemple du package local test, t Exécute les tests bench Exécute les benchmarks update Met à jour les dépendances listées dans Cargo.lock search Recherche des crates dans le registre publish Empaquette et téléverse ce package vers le registre install Installe un binaire Rust. L'emplacement par défaut est $HOME/.cargo/bin uninstall Désinstalle un binaire Rust

Voir 'cargo help ' pour plus d'informations sur une commande spécifique.

```
Il n'est pas nécessaire de comprendre toutes les commandes. Il suffit de connaître les commandes couramm
```

```
Continuons à consulter la documentation officielle :
```

```c
```

Écrivons une petite application avec notre nouvel environnement de développement Rust. Pour commencer, r

```bash
cargo new hello-rust
```

Cela générera un nouveau répertoire appelé `hello-rust` avec les fichiers suivants :

hello-rust |- Cargo.toml |- src |- main.rs Cargo.toml est le fichier manifeste pour Rust. C'est là que vous conserverez les métadonnées de votre projet, ainsi que les dépendances.

`src/main.rs` est l'endroit où nous écrirons le code de notre application.

Cela explique comment créer un projet. Ensuite, nous allons le créer.

```
$ cargo new hello-rust
```

Création du paquet binaire (application) `hello-rust`

Nous ouvrons le projet avec VSCode.

main.rs :

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

Ensuite, il est tout naturel de penser à **build** et **run** le programme.

```
$ cargo build
```

erreur : impossible de trouver `Cargo.toml` dans `/Users/lzw/ideas/curious-courses/program/run/rust` ou dans aucun répertoire parent

Une erreur s'est produite. Pourquoi ? Cela indique que `cargo` ne peut être exécuté que dans le réperto

À ce moment-là, je me suis demandé ce qui se passerait si je l'exécutais directement.

```shell
```

```
$ cargo run

   Compilation de hello-rust v0.1.0 (/Users/lzw/ideas/curious-courses/program/run/rust/hello-rust)
    Terminé en mode dev [non optimisé + debuginfo] target(s) en 4.43s
     Exécution de `target/debug/hello-rust`
Bonjour, le monde !
```

Parfait, ça a fonctionné. La chaîne de caractères a été affichée, le programme commence à fonctionner.

Essayez de modifier le programme.

```rust
fn main() {
    println!(2+3);
}
```

Après avoir exécuté `cargo run`, le message suivant est apparu :

```
   Compilation de hello-rust v0.1.0 (/Users/lzw/ideas/curious-courses/program/run/rust/hello-rust)
erreur: l'argument de format doit être un littéral de chaîne
 --> src/main.rs:2:14
  |
2 |     println!(2+3);
  |              ^^^
  |
aide: il se peut qu'il vous manque un littéral de chaîne pour formater
  |
2 |     println!("{}", 2+3);
  |              ^^^^^
```

erreur : abandon en raison d'une erreur précédente

erreur : impossible de compiler `hello-rust`

Pour en savoir plus, exécutez à nouveau la commande avec l'option `--verbose`.

```
Je n'ai encore appris aucune syntaxe Rust. En essayant de modifier le code en suivant mon intuition, j'a

```rust
```
```

```
fn main() {
    println!("{}", 2+3);
}
```

Cette fois-ci, c'est correct, et effectivement, cela a bien affiché 5.

Et pour la commande `build`, qu'en est-il ?

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
```

Pourquoi avons-nous besoin de `build` ? Parce qu'il est possible que nous souhaitions simplement générer un programme exécutable sans vouloir l'exécuter immédiatement. Pour certains programmes volumineux, l'exécution peut être chronophage. Il se peut également que nous voulions générer le programme localement, puis le transférer vers un serveur distant pour l'exécution.

Nous avons déjà réussi à exécuter un programme en Rust. La prochaine étape consiste à se familiariser davantage avec la syntaxe du langage Rust, afin de retrouver les concepts que nous avons abordés dans « Démystifier l'informatique », tels que les variables, les fonctions, les appels de fonctions et les expressions, et de comprendre comment ils sont représentés symboliquement en Rust.

Petit exercice

- Essayez, comme ci-dessus, de programmer en Rust sur votre propre ordinateur en tant qu'étudiant.
- Après avoir terminé les exercices, vous pouvez soumettre un résumé de cent mots maximum ou des compléments à cet article.
