Zeste de savoir

La version stable de Rust 1.28 est désormais disponible !

3 septembre 2018

Table des matières

1. Introduction

Rust est un langage de programmation système axé sur la *sécurité*, la *rapidité* et la *concurrence*.

Pour mettre à jour votre version stable, il suffit d'exécuter la commande habituelle.

```
$ rustup update stable
```

Si vous ne disposez pas de rustup, vous pouvez en obtenir une copie sur la page de téléchargement du site officiel. N'hésitez pas également à consulter la release note de la `1.28` sur GitHub !

2. Quoi de neuf ?

Les événements passés, d'une contre-attaque (!) matée, laissent désormais place à une nouvelle version majeure de Rust s'axant principalement sur la *mémoire*, autant sur sa gestion directe (optimisation de certains comportements que nous verrons plus bas) qu'indirecte (manipulation d'outils/services proposés permettant d'adapter cette gestion en fonction des besoins).

Allons-y !

3. Personnalisation de l'allocateur global

Si vous n'avez qu'une vague idée de ce à quoi, structurellement, un allocateur mémoire pourrait ressembler, en voici un résumé :

Il existe au moins deux "types" d'allocateurs :

1. Les allocateurs système[1], qui disposent d'une implémentation native de malloc[2], vous permettant ainsi d'effectuer les appels système désirés. C'est, en quelque sorte, l'interaction classique entre un programme et le reste de son environnement (concernant la gestion de la mémoire, tout du moins) ;
2. Les allocateurs personnalisés, qui peuvent représenter tout ce que vous pourriez imaginer pour organiser le contenu de votre mémoire, son allocation, sa libération et sa (dé)fragmentation. Les allocateurs personnalisés peuvent alors tout aussi bien proposer une implémentation de malloc avec des spécifications différentes, adaptés à un besoin très spécifique, tout comme créer un ensemble d'outils génériques offrant une alternative à l'implémentation système. Pour ne citer que l'un d'entre-eux : je vous présente jemalloc !

3.0.1. `jemalloc`

Actuellement utilisé par le projet Rust , il propose des fonctionnalités facilitant l'analyse des performances, la protection contre la fragmentation, le support avancé de la concurrence et l'interfaçage de l'allocateur lui-même avec des composants externes (tels qu'un projet C/C++), faisant de ce dernier un outil fondamentalement modulaire et adaptable au besoin.

3.1. Intégration à Rust

Avec la venue de la `1.28`, il est désormais possible de *choisir* l'allocateur à la compilation. Notez que, pour la plupart des OS , l'allocateur choisi par rust est *jemalloc* et peut être remplacé par l'allocateur du système grâce à la structure `std::alloc::System` ainsi qu'à l'attribut `#[global_allocator]`.

```rust
use std::alloc::System;

#[global_allocator]
static GLOBAL: System = System;

fn main() {
    let mut v = Vec::new();
    // Cette allocation sera effectuée
    // par l'allocateur du système d'exploitation
    // et non jemalloc.
    v.push(1);
}
```

Pour diverses raisons, vous pourriez également avoir besoin d'utiliser un allocateur fait-maison pour combler les besoins d'une situation délicate (notamment dans le cas où votre programme est très gourmand en RAM sur le long terme, effectuant de très nombreuses petites allocations et ralentissant alors le fonctionnement de l'implémentation classique de malloc). La bibliothèque standard propose le trait `GlobalAlloc` pour fournir les services nécessaires à la bonne utilisation de l'allocateur et permettre son enregistrement en tant qu'allocateur global lors l'exécution.

4. Amélioration de la formulation des erreurs de formatage

Cas très spécifique pour cette révision, puisqu'elle consiste à améliorer l'intelligibilité du diagnostic fourni par le compilateur lorsque l'utilisateur soumet un nom d'argument invalide dans son modèle de formatage.

i

A titre informatif, le formatage d'une chaîne de caractères en Rust par le biais d'arguments positionnels nommés (et non indexés) se déroule comme suit :

```
println!("{bar}{foo}", foo = ", world!", bar = "Hello")
```

`foo` et `bar` ne sont pas initialisés au préalable ; l'ordre d'initialisation n'est pas important.

Cependant, les identificateurs utilisés pour le formatage ne peuvent pas être préfixés par un underscore (`_`) et le compilateur nous renvoyait jusqu'ici un message assez peu... compréhensible.

```
format!("{_foo}", _foo = 6usize);
```

```
error: invalid format string: expected `'}'`, found `'_'`
  |
2 |     format!("{_foo}", _foo = 6usize);
  |              ^^^^^^^^
```

Si nous analysons le message "tel quel", on remarque que le compilateur reste plutôt évasif. Le problème d'exhaustivité tend à se régler dans cette nouvelle version grâce à un message d'erreur spécifique à ce cas.

1. Et donc spécifique à une plateforme, un système d'exploitation bien précis.
2. Ou équivalent pour les systèmes d'exploitation non-Unix.
3. Pour ceux que ça intéresse, vous pourrez retrouver un autre exemple d'allocateur dans l'annexe.

```
error: invalid format string: invalid argument name `_foo`
  |
2 |     let _ = format!("{_foo}", _foo = 6usize);
  |                         ^^^^ invalid argument name in format
    string
  |
  = note: argument names cannot start with an underscore
```

Mieux !

5. Optimisation des types numériques non-signés

Une nouvelle collection de wrappers, censés représenter des entiers non-signés et non-nuls, pointe le bout de son nez pour amorcer une optimisation ciblant la mémoire consommée par les `enum`s. Le cas le plus pertinent reste le stockage d'un entier non-signé dans un conteneur `Option<u8>` (ou tout autre type d'entier non-signé).

```rust
assert_eq!(std::mem::size_of::<Option<u8>>(), 2);
```

Ici, 1 octet est réservé pour le type `Option` et le second l'est pour le primitif `u8`. Grâce aux nouveaux types `NonZero` , le compilateur est capable d'épargner la machine de la consommation de la structure.

? Comment ?

Je ne suis, personnellement, pas parvenu à trouver d'explications officielles. Toutefois, m'est avis que `rustc` doit se permettre de ne pas effectuer d'alignement mémoire pour économiser le moindre octet. J'en suis arrivé à cette conclusion car :

- un entier `u16` est codé sur 2 octets ;
- une énumération (ne contenant que des constantes simples) n'est codée que sur un seul octet ;
- `size_of` devrait renvoyer `3`.

```rust
assert_eq!(std::mem::size_of::<Option<u16>>(), 3);
```

Et... ce n'est pas le cas.

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `3`', src/main.rs:4:5
```

Maintenant, utilisons la structure `NonZeroU16`.

```
assert_eq!(std::mem::size_of::<Option<NonZeroU16>>(), 3);
```

... ça ne tient toujours pas.

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `2`,
 right: `3`', src/main.rs:4:5
```

Ma théorie tombe donc à l'eau, j'ignore où est passé le dernier octet sur lequel est censée être codée l'instance de l'énumération.

En attendant d'avoir la réponse, les performances sont tout de même au rendez-vous !

6. Mise à jour de Cargo

Enfin, nous terminerons sur une révision plutôt courte apportée à cargo.

Il n'est désormais plus possible de publier une crate dont le manifest `build.rs` modifie le répertoire `src` lors du processus de compilation. `src` sera dorénavant considéré comme une ressource immuable lorsque le code est distribué.

7. Annexe

Comme précisé plus haut, je tenais à laisser en annexe un dépôt github présentant des architectures différentes d'allocateurs. L'auteur fournit des explications et des illustrations qui méritent clairement de s'y intéresser ne serait-ce que par curiosité.

Note : L'allocateur est écrit en C++ mais la documentation se suffit à elle-même.

Bonne lecture !

8. Conclusion

9. Remerciements

Merci à @unidan et @backmachine pour leur relecture orthographique ainsi que leur retour en général !

10. Source(s)

— Le blog de l'équipe Rust
— Release note de la version

Liste des abréviations

OS Operating System (Système d'exploitation). 2