Zeste de savoir

La version stable de Rust 1.27.0 est désormais disponible !

22 mars 2019

Table des matières

% LA VERSION STABLE DE RUST 1.27.0 EST DÉSORMAIS DISPONIBLE ! % Songbird % 24 juin 2018

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.

```
1 $ 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 `1.27.0` sur GitHub !

> **!** Peu de temps avant la publication de la version `1.27.0`, un nouveau bug , relatif au patch d'ergonomie appliqué sur l'expression du `match`, a été détecté. Rappelez-vous, cette modification avait déjà causé des problèmes lors de la publication de la `1.26.0` et l'équipe Rust suppose que ce bug a également été introduit à cet instant. Le cycle de publication n'a, malgré tout, pas été interrompu et un correctif (numéroté `1.27.1` ou `1.26.3`, selon les vœux de la communauté) devrait être rapidement disponible pour pallier au souci.

2. Quoi de neuf ?

Deux fonctionnalités majeures, relatives au langage et dont nous reparlerons plus bas, débarquent avec cette nouvelle version. Avant de nous atteler à celles-ci, intéressons-nous à la mise à jour de mdbook qui permet désormais aux lecteurs, des livres présents dans la Rust Bookshelf , de rechercher des termes à travers toutes les pages de leur ressource en temps réel. Essayez, par exemple, d'effectuer une recherche dans le livre officiel avec le terme `ownership` et vous remarquerez que le livre vous souligne toutes les occurrences du terme à travers les pages.

Enfin, toujours en rapport avec la documentation, un nouveau livre, dédié à `rustc`, vient s'ajouter à la bibliothèque. Il apprendra à ses lecteurs à se servir directement du compilateur, sans passer par cargo et pourra toujours être utile en tant que pense-bête pour les arguments , par exemple.

3. Single Instruction, Multiple Data (SIMD)

Penchons-nous maintenant sur l'une des deux fonctionnalités majeures, promises par la `1.27.0` : l'exploitation du paradigme SIMD[1].

Pour cette partie, j'ai décidé de remanier l'exemple proposé car il me semblait un peu confus et peu explicite.

```rust
use std::fmt::Write;

pub fn foo(a: &[u8], b: &[u8], c: &mut [u8]) {
    let mut buf: String = String::new();

    // `d` représente la valeur courante lorsque nous itérons sur la slice `a`.
    // `e` représente la valeur courante lorsque nous itérons sur la slice `b`.
    // `f` représente la valeur courante lorsque nous itérons sur la slice `c`.

    for ((d, e), f) in a.iter().zip(b).zip(c) {
        *f = *d + *e;
        write!(buf, "\n`d` value: {0}\n`e` value: {1}\n`f` value: {2}\n----------", d, e, f).expect("Oops");
    }

    println!("{}", buf.as_str());
}

// Voir ci-dessous pour imaginer la provenance des slices

fn main() {
```

```
    let a: Vec<u8> = vec![1, 2, 3];
    let b: Vec<u8> = vec![4, 5, 6];
    let mut c: Vec<u8> = vec![0, 0, 0];
    foo(&a, &b, &mut c);
    println!("{:?}", &c);
}
```

Résultat :

```
`d` value: 1
`e` value: 4
`f` value: 5
----------
`d` value: 2
`e` value: 5
`f` value: 7
----------
`d` value: 3
`e` value: 6
`f` value: 9
----------
[5, 7, 9]
```

Ici, nous traitons deux slices et ajoutons leurs nombres respectifs dans une troisième slice. La plus simple des manières d'effectuer ces traitements est déjà illustrée par le code ci-dessus. Cependant, le compilateur peut généralement faire mieux grâce à LLVM et sa capacité à auto-vectoriser des codes de ce type.

Imaginons que nos deux slices (`a` et `b`) disposent d'une taille égale à 16 éléments. Si chaque élément représente un octet (`u8`), cela signifierait que chaque slice contient 128 bits de données. En utilisant le SIMD nous pourrions placer `a` et `b` dans des registres de 128 bits, fusionner les contenus en *une seule instruction*, puis les copier dans `c`, ce qui devrait être bien plus rapide.

Bien que les versions stables de Rust ont toujours eu recours à l'auto-vectorisation, le compilateur n'est, parfois, pas capable de déceler les cas où l'optimisation peut être effectuée. Ajoutons à cela que tous les CPU ne supportent pas le paradigme SIMD, imposant à LLVM de ne pas utiliser ce dernier pour ne pas compromettre la portabilité de votre programme.

En réponse à ces problèmes, en `1.27.0`, le module `std::arch` donnera accès à des outils permettant de déclencher *manuellement* l'optimisation, pour couvrir les cas où le compilateur n'est pas capable de le faire lui-même. Il permettra également d'inclure des fonctionnalités suivant l'implémentation choisie. Par exemple :

```
#[cfg(all(any(target_arch = "x86", target_arch = "x86_64"),
      target_feature = "avx2"))]
fn foo() {
```

```
    #[cfg(target_arch = "x86")]
    use std::arch::x86::_mm256_add_epi64;
    #[cfg(target_arch = "x86_64")]
    use std::arch::x86_64::_mm256_add_epi64;

    unsafe {
        _mm256_add_epi64(...);
    }
}
```

`_mm256_add_epi64` est l'une des fonctions nécessitant l'inclusion d'une fonctionnalité particulière : AVX2. Pour plus d'information, rendez-vous sur la partie de la documentation relative à la détection du CPU hôte .

Ici, donc, les appels de fonction sont ajoutés à la compilation suivant l'architecture (`x86` ou `x86_64`, en l'occurrence). Ces vérifications peuvent, toutefois, être effectuées à l'exécution grâce à la macro `is_x86_feature_detected!`.

```
fn foo() {
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    {
        if is_x86_feature_detected!("avx2") {
            return unsafe { foo_avx2() };
        }
    }

    foo_fallback();
}
```

`is_x86_feature_detected!` se charge de générer le code qui vous permettra de détecter si le CPU hôte supporte AVX2 et, si c'est le cas, d'exécuter la fonction `foo_avx2`. Sinon, le test est court-circuité et le programme retourne à une implémentation se passant des services de la fonctionnalité. Dans un cas l'exécution bénéficiera d'une vélocité accrue, dans l'autre vous n'aurez, au moins, pas à vous soucier de la portabilité de votre code.

Dû à l'aspect plutôt primitif du module, il se peut que l'équipe Rust prévoit d'en stabiliser un autre, modestement nommé `std::simd`, pour permettre l'encapsulation de toutes ces notions et ainsi fournir des outils plus abstraits. En attendant, vous pouvez toujours vous tourner vers la crate `faster` . Cette dernière fournit le nécessaire pour écrire du code portable tout en exploitant le paradigme SIMD avec, somme toute, un maximum d'abstraction. Voici un exemple (sans `faster`) :

```rust
let lots_of_3s = (&[-123.456f32; 128][..]).iter()
    .map(|v| {
        9.0 * v.abs().sqrt().sqrt().recip().ceil().sqrt() - 4.0 -
    2.0
    })
    .collect::<Vec<f32>>();
```

Avec `faster` :

```rust
let lots_of_3s = (&[-123.456f32; 128][..]).simd_iter()
    .simd_map(f32s(0.0), |v| {
        f32s(9.0) * v.abs().sqrt().rsqrt().ceil().sqrt() -
    f32s(4.0) - f32s(2.0)
    })
    .scalar_collect();
```

Les services proposés sont sensiblement les mêmes que ceux de la std, ce qui devrait faciliter l'adoption du paradigme.

4. Déclaration d'un trait dans une signature (`dyn Trait`)

Avant la `1.27.0`, il pouvait être difficile de faire la différence entre une signature déclarant l'utilisation d'un trait, d'une autre déclarant l'utilisation d'un objet.

```rust
Box<Foo>
```

Ici, `Foo` pourrait tout à fait être un trait comme une structure sans que nous puissions clairement faire la distinction. Désormais, lorsque nous aurons recours à la généricité, nous pourrons utiliser le mot-clé `dyn`. Ce dernier précèdera chaque type représentant un trait.

```rust
// old => new
Box<Foo> => Box<dyn Foo>
&Foo => &dyn Foo
&mut Foo => &mut dyn Foo
```

Voici un exemple d'utilisation :

1. Pour plus d'informations au sujet du concept même, vous pouvez obtenir un élément de réponse ici .

```rust
use std::borrow::Borrow;

trait Foo{
    fn bar(&self);
}

struct Baz;

impl Foo for Baz {
    fn bar(&self) {
        println!("Hello there!");
    }
}


fn bang() -> Box<dyn Foo> {
    Box::new(Baz)
}

fn main() {
    let a: Box<dyn Foo> = bang();
    let b: &dyn Foo = a.borrow();
    b.bar();
}
```

Pour autant, l'ancienne syntaxe n'a pas été retirée et peut donc toujours être utilisée, l'équipe Rust informant que cela casserait la compatibilité descendante. Vous pouvez cependant mettre à niveau votre code grâce à `rustfix` qui a été conçu spécialement dans ce but.

A l'avenir, si vous ne souhaitez plus adopter l'ancienne syntaxe, vous pouvez activer une lint pour vous signaler des patterns obsolètes : `bare-trait-object` . Elle n'est pas activée par défaut pour éviter le flood de warning.

5. Utilisation de l'attribut `#[must_use]` sur les fonctions

Précédemment utilisé sur les types (tels que `Result<T, E>`), l'attribut `#[must_use]` peut désormais être utilisé sur les signatures des fonctions et ainsi vous prévenir, lors de la compilation, lorsque la valeur renvoyée n'est pas utilisée.

```rust
#[must_use]
fn double(x: i32) -> i32 {
    2 * x
```

```
}

fn main() {
    double(4); // warning: unused return value of `double` which
    must be used

    let _ = double(4); // (no warning)
}
```

Les services `Clone::clone`, `Iterator::collect` ainsi que `ToOwned::to_owned` font d'ores et déjà usage de l'attribut pour vous prévenir d'éventuelles opérations inutiles.

6. Stabilisations apportées à la bilbiothèque standard

21 nouveaux services ont été stabilisés et publiés dans cette nouvelle version. Je ne vous ferai pas l'affront de tous vous les citer, je vous invite à vous rendre directement sur la release note si vous souhaitez en savoir plus à ce sujet.

7. Fonctionnalités ajoutées à Cargo

Dans cette version, deux fonctionnalités ont été intégrées au gestionnaire de projet.

Premièrement, il est désormais possible de redéfinir, à la volée, le répertoire de sortie (*target directory*) pour la prochaine compilation grâce au flag `--target-dir` .

Enfin, la dernière, mais pas des moindres : un patch de "prévention" a été appliqué à Cargo pour prévenir les utilisateurs d'un défaut connu mais qui pourrait ne pas être corrigé dans le futur (correction entravée par la compatibilité descendante). Lorsque vous créez de nouveaux exemples (ou de nouveaux tests), Cargo se chargera de chercher les nouvelles entités à chaque recompilation. Seulement, ce comportement est court-circuité dès que vous rédigez une configuration spécifique à un exemple. Dans ce cas, Cargo traitera ce dernier mais ne lancera plus la détection automatique pour les autres.

En réponse à cela, une Pull Request a été soumise pour prévenir l'utilisateur lorsqu'il déclare une configuration dans son manifeste. Le message d'information ressemble à ceci :

```
warning: An explicit [[bin]] section is specified in Cargo.toml
    which currently
disables Cargo from automatically inferring other binary targets.
This inference behavior will change in the Rust 2018 edition and
    the following
files will be included as a binary target:

* /Users/eric/Proj/rust/cargo/src/bin/cli.rs
* /Users/eric/Proj/rust/cargo/src/bin/command_prelude.rs
```

De nouvelles clés ont également été ajoutées au manifest pour permettre de configurer chaque entité (e.g. `[[example]]`, `[[bin]]`) explicitement et ainsi forcer Cargo à lancer la détection. Comme pour les stabilisation, je vous invite vivement à consulter la section de la release note relatives aux clés .

8. Conclusion

9. Source

Le blog de l'équipe Rust

10. Voir aussi

— La version stable de Rust 1.26.2 est désormais disponible !

Liste des abréviations

SIMD Single Instruction, Multiple Data. 1–4