Zeste de savoir

La version 1.29 de Rust est désormais disponible !

12 août 2019

Table des matières

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 `1.29` sur GitHub !

1. Quoi de neuf ?

Ce 13 septembre, certainement l'une des plus petites versions majeures de Rust a été publiée ! En effet, la version `1.29` vient apporter les bases de ce qui sera amélioré dans les deux prochaines versions (i.e. `1.30` et `1.31`), ce qui rend la note un peu moins copieuse. En contrepartie, bon nombre de stabilisations ont été appliquées, du côté de la bibliothèque standard comme de Cargo.

Allons voir ça de plus près !

2. Un correcteur pas comme les autres !

Un nouvel outil, cette fois-ci dédié à la bonne tenue du code, a été intégré pour cette version. Ce dernier est bien entendu rattaché à Cargo et peut être appelé par le biais de la commande `cargo fix`.

Lancée à la racine de votre projet, `cargo fix` corrigera toutes les erreurs que `rustc` pourrait vous remonter lors de la phase d'analyse. Dans une plus grande mesure, cette commande étant,

en quelque sorte, un frontend pour rustfix , elle vous permettra d'adapter du vieux code aux conventions d'écriture de 2018 (ce qui comprend également l'indentation et la disposition des tokens).

> *i* Pour bien faire la distinction entre *rustfmt* et *rustfix*, il faut prendre conscience de ce qu'ils proposent réellement, respectivement :
> — L'un est *dédié* au formatage ;
> — L'autre débarrasse le code d'erreurs mineures. Le formatage n'est que l'une de ses fonctionnalités parmi d'autres.

(Exemple officiel un peu modifié)

```rust
fn main() {
    let foo = 117;
    for i in 0..100 {
        do_something();
    }
}
```

Ici, les ressources `foo` et `i` ne sont jamais utilisées et le compilateur vous affichera un avertissement vous invitant à vous en servir ou à préfixer l'identificateur d'un underscore (`_`).

```
warning: unused variable: `foo`
 --> src\main.rs:4:6
  |
4 |     let foo = 117;
  |         ^^^ help: consider using `_foo` instead
  |
  = note: #[warn(unused_variables)] on by default

warning: unused variable: `i`
 --> src\main.rs:5:9
  |
5 |     for i in 0..100 {
  |         ^ help: consider using `_i` instead

warning: crate `Foo` should have a snake case name such as `foo`
  |
  = note: #[warn(non_snake_case)] on by default
```

En exécutant la commande, chaque occurrence de l'avertissement sera traitée et la suggestion appliquée.

```rust
fn do_something() {}

fn main() {
        let _foo = 117;
    for _i in 0..100 {
        do_something();
    }
}
```

Bien que, en théorie, rustfix puisse avoir un grand potentiel, le nombre de lints actuellement corrigées automatiquement est encore très restreint, l'équipe ne souhaitant pas corriger d'autres avertissements que ceux concernant les conventions que elle-même a instauré pour le moment.

3. Préversion de clippy disponible

Juste après le compilateur lui-même, Clippy est sans doute le meilleur ami du développeur lors de ses phases d'analyses et de recherche de bug. Il a d'abord été conçu pour compléter ce que `rustc` ne peut se permettre de relever, dans un souci d'objectivité, et est donc l'extension (malgré tout optionnelle) de ce dernier.

Bien qu'il puisse supporter des lints moins objectives (entendez ici : qui vont au-de-là du bon sens et proposent des solutions parmi d'autres), leur ajout n'en est pas moins réglementé et peuvent être ignorées par l'utilisateur en choisissant le niveau de sévérité auquel il souhaite avoir affaire.

3.0.1. Champs d'action

L'une des forces de Clippy est également son support plutôt important de lints, toutes abordant des aspects différents d'un programme.

- Les conventions de nommage ;
- Les conventions d'écriture (des expressions, entre autres choses) ;
- Les performances ;
- La pertinence des comparaisons ;
- La sémantique des opérations ;
- ...

... et j'en passe.

Par exemple, il lui est possible d'analyser la gestion de notre mémoire et de nous prévenir lorsque nous n'en faisons pas un bon usage.

(Exemple provenant du billet officiel)

```
$ cargo clippy
error: calls to `std::mem::drop` with a reference instead of an
    owned value. Dropping a reference does nothing.
 --> src\main.rs:5:5
  |
5 |     std::mem::drop(&lock_guard);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[deny(drop_ref)] on by default
note: argument has type
    &std::result::Result<std::sync::MutexGuard<'_, i32>,
    std::sync::PoisonError<std::sync::MutexGuard<'_, i32>>>
 --> src\main.rs:5:20
  |
5 |     std::mem::drop(&lock_guard);
  |                    ^^^^^^^^^^^
  = help: for further information visit
    https://rust-lang-nursery.github.io/rust-clippy/v0.0.212/index.html#drop_re
```

Le message est plutôt explicite mais ici Clippy nous explique que nous tentons de libérer une *référence* de la ressource plutôt que la ressource elle-même, ce qui n'est logiquement pas correct (et surtout inutile). Comme vous pouvez le remarquer, les messages d'erreur sont généralement plus explicites que ceux fournis par `rustc`.

Actuellement, le dépôt affiche 257 lints intégrées au compteur.

3.0.2. Disponibilité

Seulement, jusqu'ici, l'outil n'était compilé qu'avec la version la plus récente de Rust (Nightly) et donc la plus instable. Rustup accueillant désormais la préversion de Clippy, il est tout à fait possible d'analyser du code écrit et compilé dans une version se trouvant dans le canal stable.

Comme d'habitude l'ajout de ce nouveau composant à rustup est extrêmement simple.

```
rustup component add clippy-preview
```

4. Support de Clippy par `cargo fix`

L'intégration de `cargo fix` étant toute fraîche, les retours faits par Clippy lors de son exécution ne sont malheureusement pas pris en compte, l'outil pouvant proposer des modifications qui ne font pas l'unanimité (comme précisé en début de billet). La cohabitation entre les deux composants sera malgré tout améliorée au fil des versions.

À l'instar de Clippy, nous pourrions supposer que, à terme, différents degrés de correction devraient être disponibles pour permettre aux développeurs d'adapter l'analyse à leurs besoins.

5. Du nouveau pour Cargo

5.0.1. Corruption de `Cargo.lock` lors d'une merge

Un problème, posant des difficultés à Cargo pour parser le manifest `Cargo.lock`, a été remonté il y a déjà quelques mois par un développeur. Ce dernier avait ajouté une nouvelle dépendance à son projet sur une branche qu'il s'apprêtait à fusionner. Après avoir résolu les conflits entre la première version du manifest et la seconde, il a remarqué que le fichier `Cargo.lock` final était corrompu et inintelligible pour le gestionnaire.

Dans l'attente d'une solution moins rébarbative, le premier palliatif à ce souci était d'écraser la première version du manifest et laisser la seconde prendre sa place, ayant été récemment mis à jour sur la branche temporaire. À partir de la `1.29`, Cargo se charge de régler ce genre de problèmes lui-même en régénérant le manifest à chaque fois que ces erreurs de parsing surviennent, plutôt que de faire planter le programme. Ce comportement est bien entendu totalement transparent pour l'utilisateur final et peut tout à fait être désactivé en passant le flag `--locked` lors de la compilation ; ce dernier empêche la modification de `Cargo.lock` lorsque ça n'est pas approprié (e.g. durant le processus d'intégration continue).

5.0.2. Documentation des éléments privés

Petite précision pour ceux qui ne sont pas familiers avec l'encapsulation des composants en Rust : malgré lui, le mot-clé `pub` rend un élément public *uniquement* aux yeux du module *parent*. L'état public d'un élément doit être propagé jusqu'au plus haut niveau pour être visible et utilisé en dehors de la crate. Il est donc tout à fait approprié de préfixer par `pub` un élément qui est censé être interne à la crate (pour que ce dernier soit visible pour les éléments voisins).

Imaginons une structure de projet toute simple (les principales informations se trouvent dans les commentaires).

```
1 // lib.rs
2 /// Ceci est le message d'introduction d'un module
3 /// public qui verra sa documentation générée sur le site.
4 pub mod my_public_mod;
```

```
1 // my_public_mod/mod.rs
2
```

```rust
/// Documentation d'introduction du module privé.
/// Ce texte ne devrait pas non plus apparaître sur le site.
mod my_private_mod;
use self::my_private_mod::Bar;

/// Cette structure est publique.
/// Sa documentation figurera dans la rustdoc.
pub struct Foo;

impl Foo {

        /// Créé un objet `Foo`.
        pub fn new() -> Self {
                Foo
        }

        /// Affiche un message de bienvenue.
        pub fn public_greetings(&self) {
                println!("Hello there! I'm a `Foo` object!");
        }

        /// Affiche un message de bienvenue
        /// en faisant appel à une fonction privée...
        pub fn private_greetings(&self) {
                let my_private_object = Bar::new();
                my_private_object.say_hello();
        }
}
```

Enfin :

```rust
// my_private_mod.rs
/// `Bar` est une structure privée.
/// Un appel à `cargo doc` n'affichera pas
/// cette documentation sur le site.
pub struct Bar;

impl Bar {

        /// Créé un objet `Bar`.
        pub fn new() -> Self {
                Bar
        }

    /// Fonction privée qui dit des choses
    /// mais qui ne verra pas sa documentation générée.
        pub fn say_hello(&self) {
                println!("I AM the night, I'm... Batman.");
```

```
18          }
19  }
```

Jusqu'à présent, lorsque vous généreriez la documentation grâce à `cargo doc --open` et que vous vous rendiez sur le site, vous obteniez quelque chose dans ce style :

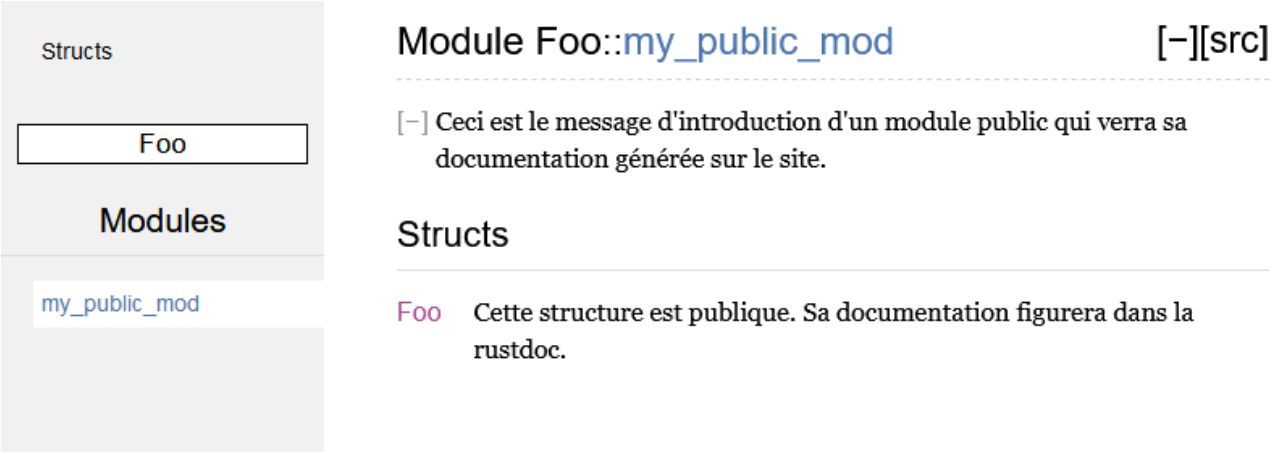

FIGURE 5. – Documentation publique

Les composants publics (et donc accessibles aux utilisateurs finaux) sont bien visibles mais pour ce qui est des composants privés, il fallait se contenter des commentaires directement rédigés dans le code. Ces derniers peuvent toujours se montrer utiles mais deviennent beaucoup moins pratiques lorsqu'il faut également disposer d'une documentation interne lisible et accessible (pour une architecture micro-service, par exemple).

En ce sens, le flag `--document-private-items` devient le porteur de cette nouvelle fonctionnalité et nous permet de consulter l'ensemble de nos briques ; ce qui, dans notre cas, nous donne accès à un nouveau bouton pour accéder à `my_private_mod`.

FIGURE 5. – Documentation privée

C'est tout pour aujourd'hui ! Comme prévu, les fonctionnalités croustillantes se sont gardées de montrer le bout de leur nez mais espérons que ça ne soit que partie remise ! Il ne nous reste plus qu'à surveiller l'avancement de Clippy et `cargo fix` jusqu'à la prochaine mise à jour.

6. Remerciements

Un grand merci à @backmachine pour ses questions et sa relecture ainsi qu'à @imperio pour l'éclaircissement apporté.

7. Source

— Billet officiel du blog Rust

8. Voir aussi

— (Article précédent) La version stable de Rust 1.28 est désormais disponible !
— La version stable de Rust 1.27.2 est désormais disponible !