eBook Gratuit

APPRENEZ Rust

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#rust

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rust

It is an unofficial and free Rust ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Rust.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Rust

Remarques

Rust est un langage de programmation conçu pour la sécurité, la vitesse et la concurrence. Rust dispose de nombreuses fonctionnalités de compilation et de contrôles de sécurité pour éviter les courses de données et les bogues courants, le tout avec un minimum de temps d'exécution.

Versions

Stable

Version	Date de sortie
1,17,0	2017-04-27
1,16,0	2017-03-16
1.15.1	2017-02-09
1,15.0	2017-02-02
1.14.0	2016-12-22
1.13.0	2016-11-10
1.12.1	2016-10-20
1.12.0	2016-09-30
1.11.0	2016-08-18
1.10.0	2016-07-07
1.9.0	2016-05-26
1.8.0	2016-04-14
1.7.0	2016-03-03
1.6.0	2016-01-21
1.5.0	2015-12-10
1.4.0	2015-10-29
1.3.0	2015-09-17

Version	Date de sortie
1.2.0	2015-08-07
1.1.0	2015-06-25
1.0.0	2015-05-15

Bêta

Version	Date de publication prévue
1,18.0	2017-06-08

Examples

Utilisation avancée de println!

`println!` (et ses frères et sœurs, `print!`) fournit un mécanisme pratique pour produire et imprimer du texte contenant des données dynamiques, similaire à la famille de fonctions `printf` dans de nombreux autres langages. Son premier argument est une *chaîne de format* qui dicte comment les autres arguments doivent être imprimés en tant que texte. La chaîne de format peut contenir des espaces réservés (entourés de `{}`) pour indiquer qu'une substitution doit avoir lieu:

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

À ce stade, vous demandez peut-être: comment est-ce que `println!` savoir imprimer la valeur booléenne `true` comme la chaîne "true"? `{}` est vraiment une instruction au formateur que la valeur doit être convertie en texte en utilisant le trait d' `Display` . Ce trait est implémenté pour la plupart des types de rouille primitifs (chaînes, nombres, booléens, etc.) et est destiné à la "sortie de l'utilisateur". Par conséquent, le nombre 42 sera imprimé en décimal sous la forme 42, et non, par exemple, en binaire, ce qui correspond à la manière dont il est stocké en interne.

Comment pouvons-nous imprimer les types qui n'implémentent *pas* `Display` , par exemple les Slices (`[i32]`), les vecteurs (`Vec<i32>`) ou les options (`Option<&str>`)? Il n'y a pas de représentation textuelle claire de celles-ci (c.-à-d. Que vous pourriez insérer dans une phrase). Pour faciliter l'impression de telles valeurs, Rust possède également le trait `Debug` et l'espace réservé `{:?}` Correspondant. Dans la documentation: "Le `Debug` doit formater la sortie dans un contexte de débogage orienté programmeur." Voyons quelques exemples:

```
println!("{:?}", vec!["a", "b", "c"]);
// Output: ["a", "b", "c"]

println!("{:?}", Some("fantastic"));
// Output: Some("fantastic")

println!("{:?}", "Hello");
// Output: "Hello"
// Notice the quotation marks around "Hello" that indicate
// that a string was printed.
```

`Debug` possède également un mécanisme de jolie impression intégré, que vous pouvez activer en utilisant le modificateur `#` après les deux points:

```
println!("{:#?}", vec![Some("Hello"), None, Some("World")]);
// Output: [
//     Some(
//         "Hello"
//     ),
//     None,
//     Some(
//         "World"
//     )
// ]
```

Les chaînes de format permettent d'exprimer des substitutions assez complexes :

```
// You can specify the position of arguments using numerical indexes.
println!("{1} {0}", "World", "Hello");
// Output: Hello World

// You can use named arguments with format
println!("{greeting} {who}!", greeting="Hello", who="World");
// Output: Hello World

// You can mix Debug and Display prints:
println!("{greeting} {1:?}, {0}", "and welcome", Some(42), greeting="Hello");
// Output: Hello Some(42), and welcome
```

`println!` et vos amis vous avertiront également si vous essayez de faire quelque chose qui ne fonctionnera pas, plutôt que de vous écraser à l'exécution:

```
// This does not compile, since we don't use the second argument.
println!("{}", "Hello World", "ignored");

// This does not compile, since we don't give the second argument.
println!("{} {}", "Hello");

// This does not compile, since Option type does not implement Display
println!("{}", Some(42));
```

À la base, les macros d'impression Rust sont simplement des wrappers autour du `format!` macro, qui permet de construire une chaîne en assemblant des représentations textuelles de différentes valeurs de données. Ainsi, pour tous les exemples ci-dessus, vous pouvez remplacer `println!` pour le `format!` stocker la chaîne formatée au lieu de l'imprimer:

```
let x: String = format!("{} {}", "Hello", 42);
assert_eq!(x, "Hello 42");
```

Sortie de console sans macros

```
// use Write trait that contains write() function
use std::io::Write;

fn main() {
    std::io::stdout().write(b"Hello, world!\n").unwrap();
}
```

- Le trait `std::io::Write` est conçu pour les objets qui acceptent les flux d'octets. Dans ce cas, un handle vers la sortie standard est acquis avec `std::io::stdout()` .
- `Write::write()` accepte une tranche d'octets (`&[u8]`), qui est créée avec un littéral d'octet (`b"<string>"`). `Write::write()` renvoie un `Result<usize, IoError>` , qui contient soit le nombre d'octets écrits (en cas de succès), soit une valeur d'erreur (en cas d'échec).
- L'appel à `Result::unwrap()` indique que l'appel est censé réussir (`Result<usize, IoError> -> usize`) et que la valeur est ignorée.

Exemple minimal

Pour écrire le programme Hello World traditionnel dans Rust, créez un fichier texte appelé `hello.rs` contenant le code source suivant:

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

Ceci définit une nouvelle fonction appelée `main` , qui ne prend aucun paramètre et ne renvoie aucune donnée. C'est là que votre programme démarre l'exécution lorsqu'il est exécuté. A l'intérieur, vous avez une `println!` , qui est une macro qui imprime du texte dans la console.

Pour générer une application binaire, appelez le compilateur Rust en lui transmettant le nom du fichier source:

```
$ rustc hello.rs
```

L'exécutable résultant aura le même nom que le module source principal, donc pour exécuter le programme sur un système Linux ou MacOS, exécutez:

```
$ ./hello
Hello World!
```

Sur un système Windows, exécutez:

```
C:\Rust> hello.exe
```

```
Hello World!
```

Commencer

Installation

Avant de pouvoir faire quoi que ce soit en utilisant le langage de programmation Rust, vous devrez l'acquérir - soit pour Windows, soit en utilisant votre terminal sur des *systèmes de type* Unix, où `$` symbolise l'entrée dans le terminal:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Cela permettra de récupérer les fichiers requis et de configurer la dernière version de Rust pour vous, quel que soit le système sur lequel vous vous trouvez. Pour plus d'informations, voir la page du projet .

Note: Certaines distributions Linux (par exemple Arch Linux) fournissent un `rustup` tant que paquet pouvant être installé à la place. Et bien que de nombreux systèmes de type Unix fournissent du `rustc` et du `cargo` comme paquets séparés, il est toujours recommandé d'utiliser plutôt `rustup` car il est beaucoup plus facile de gérer plusieurs canaux de version et de faire de la compilation croisée.

Compilateur de rouille

Nous pouvons maintenant vérifier si *Rust* a bien été installé sur nos ordinateurs en exécutant la commande suivante dans notre terminal - sous UNIX - ou à l'invite de commande - sous Windows:

```
$ rustc --version
```

Si cette commande réussit, la version du compilateur *Rust* installé sur nos ordinateurs sera affichée sous nos yeux.

Cargaison

Avec Rust vient *Cargo* , qui est un outil de construction utilisé pour gérer vos packages et vos projets *Rust* . Pour vous assurer que ceci est également présent sur votre ordinateur, exécutez la commande suivante dans la console: console faisant référence à un terminal ou à une invite de commande selon le système sur lequel vous vous trouvez:

```
$ cargo --version
```

Tout comme la commande équivalente pour le compilateur *Rust* , cela renverra et affichera la version actuelle de *Cargo* .

Pour créer votre premier projet Cargo, vous pouvez vous rendre à Cargo .

Alternativement, vous pouvez compiler des programmes directement en utilisant `rustc` comme indiqué dans Exemple minimal .

Lire Démarrer avec Rust en ligne: https://riptutorial.com/fr/rust/topic/362/demarrer-avec-rust

Chapitre 2: Applications GUI

Introduction

Rust n'a pas de cadre propre pour le développement de l'interface graphique. Pourtant, il existe de nombreuses liaisons avec les cadres existants. La liaison de bibliothèque la plus avancée est rust-gtk . Une liste complète des liaisons peut être trouvée ici

Examples

Simple Gtk + Fenêtre avec texte

Ajoutez la dépendance de Gtk à votre `Cargo.toml` :

```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```

Créez une fenêtre simple avec les éléments suivants:

```
extern crate gtk;

use gtk::prelude::*; // Import all the basic things
use gtk::{Window, WindowType, Label};

fn main() {
    if gtk::init().is_err() { //Initialize Gtk before doing anything with it
        panic!("Can't init GTK");
    }

    let window = Window::new(WindowType::Toplevel);

    //Destroy window on exit
    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));
    window.add(&label);
    window.show_all();
    gtk::main();
}
```

Fenêtre Gtk + avec entrée et étiquette dans GtkBox, connexion de signal GtkEntry

```
extern crate gtk;

use gtk::prelude::*;
use gtk::{Window, WindowType, Label, Entry, Box as GtkBox, Orientation};
```

```
fn main() {
    if gtk::init().is_err() {
        println!("Failed to initialize GTK.");
        return;
    }

    let window = Window::new(WindowType::Toplevel);

    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));

    // Create a VBox with 10px spacing
    let bx = GtkBox::new(Orientation::Vertical, 10);
    let entry = Entry::new();

    // Connect "activate" signal to anonymous function
    // that takes GtkEntry as an argument and prints it's text
    entry.connect_activate(|x| println!("{}",x.get_text().unwrap()));

    // Add our label and entry to the box
    // Do not expand or fill, zero padding
    bx.pack_start(&label, false, false, 0);
    bx.pack_start(&entry, false, false, 0);
    window.add(&bx);
    window.show_all();
    gtk::main();
}
```

Lire Applications GUI en ligne: https://riptutorial.com/fr/rust/topic/7169/applications-gui

Chapitre 3: Arguments de ligne de commande

Introduction

La bibliothèque standard de Rust ne contient pas un analyseur d'argument correct (contrairement à `argparse` dans Python), préférant laisser cela à des caisses tierces. Ces exemples montreront l'utilisation de la bibliothèque standard (pour former un gestionnaire d'argument brut) et de la bibliothèque `clap` qui peut analyser les arguments de ligne de commande plus efficacement.

Syntaxe

- utilisez std :: env; // Importe le module env
- let args = env :: args (); // Stocker un itérateur Args dans la variable args.

Examples

Utiliser std :: env :: args ()

Vous pouvez accéder aux arguments de ligne de commande transmis à votre programme à l'aide de la fonction `std::env::args()` . Cela retourne un itérateur `Args` que vous pouvez `Args` boucle ou collecter dans un `Vec` .

Itérer à travers les arguments

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

Collecter dans un `Vec`

```
use std::env;

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("{} arguments passed", arguments.len());
}
```

Vous pourriez obtenir plus d'arguments que prévu si vous appelez votre programme comme ceci:

```
./example
```

Bien qu'il semble que aucun argument n'ait été transmis, le premier argument est (**généralement**) le nom de l'exécutable. Ce n'est pas une garantie, donc vous devez toujours valider et filtrer les arguments que vous obtenez.

En utilisant clap

Pour les programmes en ligne de commande plus importants, l'utilisation de `std::env::args()` est assez fastidieuse et difficile à gérer. Vous pouvez utiliser `clap` pour gérer votre interface de ligne de commande, qui analysera les arguments, générera des affichages d'aide et évitera les bogues.

Il existe plusieurs *modèles* que vous pouvez utiliser avec `clap` , et chacun fournit une quantité de flexibilité différente.

Motif de constructeur

C'est la méthode la plus prolixe (et la plus flexible), elle est donc utile lorsque vous avez besoin d'un contrôle précis de votre interface de ligne de commande.

`clap` distingue les *sous - commandes* et les *arguments* . Les sous-commandes agissent comme des sous-programmes indépendants dans votre programme principal, tout comme les opérations de `cargo run` et de `git push` . Ils peuvent avoir leurs propres options et entrées de ligne de commande. Les arguments sont des indicateurs simples tels que `--verbose` , et ils peuvent prendre des entrées (par exemple `--message "Hello, world"`)

```
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let app = App::new("Foo Server")
        .about("Serves foos to the world!")
        .version("v0.1.0")
        .author("Foo (@Example on GitHub)")
        .subcommand(SubCommand::with_name("run")
            .about("Runs the Foo Server")
            .arg(Arg::with_name("debug")
                .short("D")
                .about("Sends debug foos instead of normal foos.")))

    // This parses the command-line arguments for use.
    let matches = app.get_matches();

    // We can get the subcommand used with matches.subcommand(), which
    // returns a tuple of (&str, Option<ArgMatches>) where the &str
    // is the name of the subcommand, and the ArgMatches is an
    // ArgMatches struct:
    // https://docs.rs/clap/2.13.0/clap/struct.ArgMatches.html

    if let ("run", Some(run_matches)) = app.subcommand() {
        println!("Run was used!");
    }
}
```

Lire Arguments de ligne de commande en ligne:
https://riptutorial.com/fr/rust/topic/7015/arguments-de-ligne-de-commande

Chapitre 4: Assemblage en ligne

Syntaxe

- #! [feature (asm)] // Activer l'asm! porte caractéristique macro
- asm! (<template>: <output>: <input>: <clobbers>: <options>) // Emet le modèle d'assemblage fourni (par exemple "NOP", "ADD% eax, 4") avec les options données.

Examples

L'asm! macro

L'assemblage en ligne ne sera pris en charge que dans les versions nocturnes de Rust jusqu'à ce qu'il soit stabilisé . Pour permettre l'utilisation de l' `asm!` macro, utilisez l'attribut de fonctionnalité suivant en haut du fichier principal (une *porte de fonctionnalité*):

```
#![feature(asm)]
```

Ensuite, utilisez l' `asm!` macro dans tout bloc `unsafe` :

```
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }

    // asm!("NOP");
    // That would be invalid here, because we are no longer in an
    // unsafe block.
}
```

Compiler sous condition l'assemblage en ligne

Utilisez la compilation conditionnelle pour vous assurer que le code ne compile que pour le jeu d'instructions voulu (tel que `x86`). Sinon, le code pourrait devenir invalide si le programme est compilé pour une autre architecture, telle que les processeurs ARM.

```
#![feature(asm)]

// Any valid x86 code is valid for x86_64 as well. Be careful
// not to write x86_64 only code while including x86 in the
// compilation targets!
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }
}

#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
```

```
fn do_nothing() {
    // This is an alternative implementation that doesn't use any asm!
    // calls. Therefore, it should be safe to use as a fallback.
}
```

Entrées et sorties

```
#![feature(asm)]

#[cfg(any(target_arch="x86", target_arch="x86_64"))]
fn subtract(first: i32, second: i32) {
   unsafe {
        // Output values must either be unassigned (let result;) or mutable.
        let result: i32;
        // Each value that you pass in will be in a certain register, which
        // can be accessed with $0, $1, $2...
        //
        // The registers are assigned from left to right, so $0 is the
        // register containing 'result', $1 is the register containing
        // 'first' and $2 is the register containing 'second'.
        //
        // Rust uses AT&T syntax by default, so the format is:
        // SUB source, destination
        // which is equivalent to:
        // destination -= source;
        //
        // Because we want to subtract the first from the second,
        // we use the 0 constraint on 'first' to use the same
        // register as the output.
        // Therefore, we're doing:
        // SUB second, first
        // and getting the value of 'first'

        asm!("SUB $2, $0" : "=r"(result) : "0"(first), "r"(second));
        println!("{}", result);
    }
}
```

Les codes de contraintes de LLVM peuvent être trouvés ici , mais cela peut varier en fonction de la version de LLVM utilisée par votre compilateur `rustc` .

Lire Assemblage en ligne en ligne: https://riptutorial.com/fr/rust/topic/6998/assemblage-en-ligne

Chapitre 5: Boucles

Syntaxe

- loop { *block* } // boucle infinie
- tandis que *condition* { *bloc* }
- while let *pattern* = *expr* { *bloc* }
- pour *pattern* dans *expr* { *block* } // expr doit implémenter IntoIterator
- continue // saut à la fin du corps de la boucle, commence une nouvelle itération si nécessaire
- pause // arrête la boucle
- ' *label* : loop { *block* }
- ' *label* : while *condition* { *block* }
- ' *label* : while let *pattern* = *expr* { *bloc* }
- ' *label* : pour le *pattern* dans *expr* { *block* }
- continuer *label* // saut à la fin du corps de la boucle *étiquette* marquée, le démarrage d' une nouvelle itération si nécessaire
- break ' *label* // arrête l' *étiquette* libellée de la boucle

Examples

Les bases

Il y a 4 constructions en boucle dans Rust. Tous les exemples ci-dessous produisent la même sortie.

Boucles infinies

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

Pendant que les boucles

```
let mut x = 0;
while x <= 3 {
    println!("{}", x);
    x += 1;
}
```

Voir aussi: Quelle est la différence entre `loop` et `while true` ?

Boucles de motifs assorties

Celles-ci sont parfois connues sous le nom de « `while let` , pour des raisons de brièveté.

```
let mut x = Some(0);
while let Some(v) = x {
    println!("{}", v);
    x = if v < 3 { Some(v + 1) }
        else     { None };
}
```

Ceci est équivalent à une `match` dans un bloc de `loop` :

```
let mut x = Some(0);
loop {
    match x {
        Some(v) => {
            println!("{}", v);
            x = if v < 3 { Some(v + 1) }
                else     { None };
        }
        _       => break,
    }
}
```

Pour les boucles

Dans Rust, `for` loop ne peut être utilisé qu'avec un objet "itérable" (c'est-à-dire qu'il doit implémenter `IntoIterator`).

```
for x in 0..4 {
    println!("{}", x);
}
```

Cela équivaut à l'extrait de code suivant impliquant `while let` :

```
let mut iter = (0..4).into_iter();
while let Some(v) = iter.next() {
```

```
    println!("{}", v);
}
```

Remarque: `0..4` renvoie un objet `Range` qui implémente déjà le trait `Iterator` . Par conséquent, `into_iter()` est inutile, mais est conservé pour illustrer ce `for` fait. Pour un regard en profondeur, voir les documents officiels sur `for` boucles et `IntoIterator` .

Voir aussi: Itérateurs

En savoir plus sur les boucles

Comme mentionné dans Basics, nous pouvons utiliser tout ce qui implémente `IntoIterator` avec la boucle `for` :

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

Production attendue:

```
foo
bar
baz
```

Notez que l'itération sur `vector` de cette manière la consomme (après la boucle `for` , le `vector` ne peut plus être utilisé). En effet , `IntoIterator::into_iter` se déplace `self` .

`IntoIterator` est également implémenté par `&Vec<T>` et `&mut Vec<T>` (produisant des valeurs avec les types `&T` et `&mut T` respectivement), vous pouvez donc empêcher le déplacement du `vector` en le faisant simplement passer par référence:

```
let vector = vec!["foo", "bar", "baz"];
for val in &vector {
    println!("{}", val);
}
println!("{:?}", vector);
```

Notez que `val` est de type `&&str` , puisque le `vector` est de type `Vec<&str>` .

Contrôle de boucle

Toutes les constructions en boucle permettent l'utilisation d'instructions `break` et `continue` . Ils affectent la boucle immédiatement environnante (la plus interne).

Contrôle de boucle de base

`break` termine la boucle:

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

Sortie

```
0
1
2
```

`continue` finit l'itération en cours tôt

```
for x in 0..5 {
    if x < 2 { continue; }
    println!("{}", x);
}
```

Sortie

```
2
3
4
```

Contrôle de boucle avancé

Maintenant, supposons que nous avons des boucles imbriquées et que vous voulez `break` vers la boucle extérieure. Ensuite, nous pouvons utiliser des étiquettes de boucle pour spécifier à quelle boucle une `break` ou une `continue` s'applique. Dans l'exemple suivant, `'outer` est l'étiquette donnée à la boucle externe.

```
'outer: for i in 0..4 {
    for j in i..i+2 {
        println!("{} {}", i, j);
        if i > 1 {
            continue 'outer;
        }
    }
    println!("--");
}
```

Sortie

```
0 0
0 1
--
1 1
1 2
--
2 2
3 3
```

Pour `i > 1` , la boucle interne n'a été itérée qu'une seule fois et `--` n'a pas été imprimée.

Remarque: Ne confondez pas une étiquette de boucle avec une variable de durée de vie. Les variables de durée de vie apparaissent uniquement à côté d'un paramètre `&` ou en tant que paramètre générique dans `<>` .

Lire Boucles en ligne: https://riptutorial.com/fr/rust/topic/955/boucles

Chapitre 6: Cadre Web Iron

Introduction

Iron est un framework Web populaire pour Rust (basé sur la bibliothèque Hyper de niveau inférieur) qui promeut l'extensibilité par le biais du *middleware* . La plupart des fonctionnalités nécessaires à la création d'un site Web utile se trouvent dans le middleware d'Iron plutôt que dans la bibliothèque elle-même.

Examples

Simple 'Hello' Server

Cet exemple envoie une réponse codée en dur à l'utilisateur lorsqu'il envoie une demande de serveur.

```
extern crate iron;

use iron::prelude::*;
use iron::status;

// You can pass the handler as a function or a closure. In this
// case, we've chosen a function for clarity.
// Since we don't care about the request, we bind it to _.
fn handler(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello, Stack Overflow")))
}

fn main() {
    Iron::new(handler).http("localhost:1337").expect("Server failed!")
}
```

Lors de la création d'un nouveau serveur `Iron` dans cet exemple, `expect` - `expect` à détecter les erreurs avec un message d'erreur plus descriptif. Dans les applications de production, *gérez l'erreur* générée (voir la documentation de `http()`).

Installation du fer

Ajoutez cette dépendance au fichier `Cargo.toml` :

```
[dependencies]
iron = "0.4.0"
```

Exécuter la `cargo build` et Cargo téléchargera et installera la version spécifiée d'Iron.

Routage simple avec fer

Cet exemple fournira un routage Web de base avec Iron.

Pour commencer, vous devrez ajouter la dépendance Iron à votre fichier `Cargo.toml` .

```
[dependencies]
iron = "0.4.*"
```

Nous utiliserons la bibliothèque du routeur d'Iron. Pour plus de simplicité, le projet Iron fournit cette bibliothèque dans la bibliothèque principale d'Iron, éliminant ainsi le besoin de l'ajouter en tant que dépendance distincte. Ensuite, nous référençons à la fois la bibliothèque Iron et la bibliothèque Router.

```
extern crate iron;
extern crate router;
```

Nous importons ensuite les objets requis pour nous permettre de gérer le routage et retourner une réponse à l'utilisateur.

```
use iron::{Iron, Request, Response, IronResult};
use iron::status;
use router::{Router};
```

Dans cet exemple, nous allons rester simple en écrivant la logique de routage dans notre fonction `main()` . Bien entendu, à mesure que votre application se développe, vous souhaiterez séparer le routage, la journalisation, les problèmes de sécurité et d'autres domaines de votre application Web. Pour l'instant, c'est un bon point de départ.

```
fn main() {
    let mut router = Router::new();
    router.get("/", handler, "handler");
    router.get("/:query", query_handler, "query_handler");
```

Passons en revue ce que nous avons accompli jusqu'à présent. Notre programme instancie actuellement un nouvel objet Iron `Router` , et attache deux "gestionnaires" à deux types de requête d'URL: le premier (`"/"`) est la racine de notre domaine, et le second (`"/:query"`) sous root.

En utilisant un point-virgule avant le mot "query", nous disons à Iron de prendre cette partie du chemin de l'URL en tant que variable et de la transmettre à notre gestionnaire.

La prochaine ligne de code est la façon dont nous instancions Iron, en désignant notre propre objet `router` pour gérer nos requêtes URL. Le domaine et le port sont codés en dur dans cet exemple pour plus de simplicité.

```
    Iron::new(router).http("localhost:3000").unwrap();
```

Ensuite, nous déclarons deux fonctions en ligne qui sont nos gestionnaires, `handler` et `query_handler` . Celles-ci sont toutes deux utilisées pour démontrer les URL fixes et les URL variables.

Dans la deuxième fonction, nous prenons la variable `"query"` de l'URL détenue par l'objet de requête, et nous la renvoyons à l'utilisateur en réponse.

```
    fn handler(_: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "OK")))
    }

    fn query_handler(req: &mut Request) -> IronResult<Response> {
        let ref query = req.extensions.get::<Router>()
            .unwrap().find("query").unwrap_or("/");
        Ok(Response::with((status::Ok, *query)))
    }
}
```

Si nous exécutons cet exemple, nous pourrons afficher le résultat dans le navigateur Web à l' `localhost:3000` . La racine du domaine devrait répondre par `"OK"` , et tout ce qui se trouve sous la racine devrait répéter le chemin.

L'étape suivante de cet exemple pourrait être la séparation du routage et de la diffusion des pages statiques.

Lire Cadre Web Iron en ligne: https://riptutorial.com/fr/rust/topic/8060/cadre-web-iron

Chapitre 7: Cargaison

Introduction

Cargo est le gestionnaire de paquets de Rust, utilisé pour gérer les *caisses* (terme de Rust pour les bibliothèques / packages). Cargo récupère principalement des paquets à partir de crates.io et peut gérer des arbres de dépendance complexes avec des exigences de version spécifiques (en utilisant le contrôle de version sémantique). Cargo peut également vous aider à construire, exécuter et gérer des projets Rust avec la `cargo build` `cargo run` `cargo test` et `cargo test` (entre autres commandes utiles).

Syntaxe

- cargo new crate_name [--bin]
- cargo init [--bin]
- construction de la cargaison [--release]
- course de cargaison [--release]
- chèque de cargaison
- test de cargaison
- banc de fret
- mise à jour de la cargaison
- paquet de fret
- publication de la cargaison
- cargo [un] installe binary_crate_name
- recherche de fret crate_name
- version cargo
- identifiant de cargaison api_key

Remarques

- Pour le moment, la sous-commande `cargo bench` nécessite la version nocturne du compilateur pour fonctionner efficacement.

Examples

Créer un nouveau projet

Bibliothèque

```
cargo new my-library
```

Cela crée un nouveau répertoire appelé `my-library` contenant le fichier de configuration cargo et

un répertoire source contenant un seul fichier source Rust:

```
my-library/Cargo.toml
my-library/src/lib.rs
```

Ces deux fichiers contiendront déjà le squelette de base d'une bibliothèque, de sorte que vous pouvez effectuer un `cargo test` (à partir du répertoire de `my-library`) immédiatement pour vérifier si tout fonctionne.

Binaire

```
cargo new my-binary --bin
```

Cela crée un nouveau répertoire appelé `my-binary` avec une structure similaire à celle d'une bibliothèque:

```
my-binary/Cargo.toml
my-binary/src/main.rs
```

Cette fois-ci, `cargo` aura mis en place un simple binaire Hello World que nous pourrons utiliser immédiatement avec `cargo run` .

Vous pouvez également créer le nouveau projet dans le répertoire en cours avec la sous-commande `init` :

```
cargo init --bin
```

Tout comme ci-dessus, supprimez l'indicateur `--bin` pour créer un nouveau projet de bibliothèque. Le nom du dossier actuel est utilisé comme nom de caisse automatiquement.

Construire un projet

Déboguer

```
cargo build
```

Libération

La construction avec l'indicateur `--release` permet certaines optimisations du compilateur qui ne sont pas effectuées lors de la construction d'une version de débogage. Cela rend le code plus rapide, mais rend également la compilation un peu plus longue. Pour des performances optimales, cette commande doit être utilisée une fois la version terminée.

```
cargo build --release
```

Tests en cours

Utilisation de base

```
cargo test
```

Afficher la sortie du programme

```
cargo test -- --nocapture
```

Exécuter un exemple spécifique

```
cargo test test_name
```

Bonjour tout le monde

Ceci est une session shell montrant comment créer un programme "Hello world" et l'exécuter avec Cargo:

```
$ cargo new hello --bin
$ cd hello
$ cargo run
   Compiling hello v0.1.0 (file:///home/rust/hello)
     Running `target/debug/hello`
Hello, world!
```

Après cela, vous pouvez éditer le programme en ouvrant `src/main.rs` dans un éditeur de texte.

Publier une caisse

Pour publier une caisse sur crates.io , vous devez vous connecter avec Cargo (voir « *Connexion de la cargaison à un compte Crates.io* »).

Vous pouvez conditionner et publier votre caisse avec les commandes suivantes:

```
cargo package
cargo publish
```

Toute erreur dans votre fichier `Cargo.toml` sera mise en évidence pendant ce processus. Vous devez vous assurer de **mettre** à **jour votre version** et de vous assurer que votre fichier `.gitignore` ou `Cargo.toml` exclut les fichiers indésirables.

Connecter Cargo à un compte Crates.io

Les comptes sur crates.io sont créés en se connectant avec GitHub; vous ne pouvez pas vous inscrire avec une autre méthode.

Pour connecter votre compte GitHub à crates.io, cliquez sur le bouton " *Connexion avec GitHub* " dans la barre de menu supérieure et autorisez crates.io à accéder à votre compte. Cela vous connectera alors à crates.io, en supposant que tout s'est bien passé.

Vous devez ensuite trouver votre **clé API** , que vous pouvez trouver en cliquant sur votre avatar, en sélectionnant " *Paramètres du compte* " et en copiant la ligne qui ressemble à ceci:

```
cargo login abcdefghijklmnopqrstuvwxyz1234567890rust
```

Cela devrait être collé dans votre terminal / ligne de commande et devrait vous authentifier avec votre installation de `cargo` locale.

Soyez prudent avec votre clé API - elle **doit** rester secrète, comme un mot de passe, sinon vos caisses pourraient être détournées!

Lire Cargaison en ligne: https://riptutorial.com/fr/rust/topic/1084/cargaison

Chapitre 8: Constantes Associées

Syntaxe

- #! [feature (related_consts)]
- ID de const: i32;

Remarques

Cette fonctionnalité est actuellement disponible uniquement dans le compilateur de nuit. Problème de suivi # 29646

Examples

Utilisation de constantes associées

```
// Must enable the feature to use associated constants
#![feature(associated_consts)]

use std::mem;

// Associated constants can be used to add constant attributes to types
trait Foo {
    const ID: i32;
}

// All implementations of Foo must define associated constants
// unless a default value is supplied in the definition.
impl Foo for i32 {
    const ID: i32 = 1;
}

struct Bar;

// Associated constants don't have to be bound to a trait to be defined
impl Bar {
    const BAZ: u32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);

    // The defined constant value is only stored once, so the size of
    // instances of the defined types doesn't include the constants.
    assert_eq!(4, mem::size_of::<i32>());
    assert_eq!(0, mem::size_of::<Bar>());
}
```

Lire Constantes Associées en ligne: https://riptutorial.com/fr/rust/topic/7042/constantes-associees

Chapitre 9: Cordes

Introduction

Contrairement à beaucoup d'autres langages, Rust possède **deux** types de chaînes principaux: `String` (un type de chaîne alloué au tas) et `&str` (une chaîne **empruntée** , qui n'utilise pas de mémoire supplémentaire). Connaître la différence et savoir quand utiliser chacun est essentiel pour comprendre le fonctionnement de Rust.

Examples

Manipulation de base des cordes

```
fn main() {
    // Statically allocated string slice
    let hello = "Hello world";

    // This is equivalent to the previous one
    let hello_again: &'static str = "Hello world";

    // An empty String
    let mut string = String::new();

    // An empty String with a pre-allocated initial buffer
    let mut capacity = String::with_capacity(10);

    // Add a string slice to a String
    string.push_str("foo");

    // From a string slice to a String
    // Note: Prior to Rust 1.9.0 the to_owned method was faster
    // than to_string. Nowadays, they are equivalent.
    let bar = "foo".to_owned();
    let qux = "foo".to_string();

    // The String::from method is another way to convert a
    // string slice to an owned String.
    let baz = String::from("foo");

    // Coerce a String into &str with &
    let baz: &str = &bar;
}
```

Remarque: les méthodes `String::new` et `String::with_capacity` créeront des chaînes vides. Cependant, ce dernier alloue un tampon initial, le rendant initialement plus lent, mais aidant à réduire les allocations ultérieures. Si la taille finale de la chaîne est connue, `String::with_capacity` doit être préféré.

Tranchage

```
fn main() {
```

```
    let english = "Hello, World!";

    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

Notez que nous devons utiliser l'opérateur `&` ici. Il prend une référence et donne ainsi au compilateur des informations sur la taille du type de tranche, dont il a besoin pour l'imprimer. Sans la référence, les deux `println!` les appels seraient une erreur de compilation.

Avertissement: Le découpage fonctionne par **décalage d'octet** , et non par décalage de caractère, et panique lorsque les limites ne se trouvent pas à la limite d'un caractère:

```
fn main() {
    let icelandic = "Halló, heimur!"; // note that “ó” is two-byte long in UTF-8

    println!("{}", &icelandic[0..6]); // Prints "Halló", “ó” lies on two bytes 5 and 6
    println!("{}", &icelandic[8..]);  // Prints "heimur!", the “h” is the 8th byte, but the
7th char
    println!("{}", &icelandic[0..5]); // Panics!
}
```

C'est aussi la raison pour laquelle les chaînes ne supportent pas l'indexation simple (par exemple, `icelandic[5]`).

Diviser une chaîne

```
let strings = "bananas,apples,pear".split(",");
```

`split` renvoie un itérateur.

```
for s in strings {
  println!("{}", s)
}
```

Et peut être "collecté" dans un `Vec` avec la méthode `Iterator::collect` .

```
let strings: Vec<&str> = "bananas,apples,pear".split(",").collect(); // ["bananas", "apples",
"pear"]
```

De l'emprunt à la propriété

```
// all variables `s` have the type `String`
let s = "hi".to_string();  // Generic way to convert into `String`. This works
                           // for all types that implement `Display`.

let s = "hi".to_owned();   // Clearly states the intend of obtaining an owned object

let s: String = "hi".into();       // Generic conversion, type annotation required
let s: String = From::from("hi");  // in both cases!
```

```
let s = String::from("hi");  // Calling the `from` impl explicitly -- the `From`
                             // trait has to be in scope!

let s = format!("hi");       // Using the formatting functionality (this has some
                             // overhead)
```

Outre le `format!()` , Toutes les méthodes ci-dessus sont également rapides.

Briser les littéraux de chaîne longue

Briser les littéraux de chaîne réguliers avec le caractère `\`

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

Casser les littéraux de chaîne brute pour séparer les chaînes et les joindre au `concat!` macro

```
let c = r"foo\bar";
let d = concat!(r"foo\", r"bar");

// `c` and `d` are equal.
assert_eq!(c, d);
```

Lire Cordes en ligne: https://riptutorial.com/fr/rust/topic/998/cordes

Chapitre 10: Correspondance de motif

Syntaxe

- _ // motif de caractères génériques, correspond à tout[1]
- *ident* // modèle de liaison, correspond à tout et le lie à *ident*[1]
- *ident* @ *pat* // idem ci-dessus, mais permet de mieux faire correspondre ce qui est lié
- ref *ident* // modèle de liaison, correspond à tout et le lie à un *identifiant de* référence [1]
- ref mut *ident* // modèle de liaison, correspond à tout et le lie à un *identifiant de* référence mutable [1]
- & *pat* // correspond à une référence (*pat* n'est donc pas une référence mais l'arbitre) [1]
- & mut *pat* // idem ci-dessus avec une référence mutable[1]
- *CONST* // correspond à une constante nommée
- *Struct* { *field1* , *field2* } // correspond à une valeur de structure et la déconstruit, voir ci-dessous la note sur les champs[1]
- *EnumVariant* // correspond à une variante d'énumération
- *EnumVariant* (*pat1* , *pat2*) // correspond à une variante d'énumération et aux paramètres correspondants
- *EnumVariant* (*pat1* , *pat2* , .., *patn*) // identique à ci-dessus mais ignore tous les paramètres sauf le premier, le deuxième et le dernier
- (*pat1* , *pat2*) // correspond à un tuple et aux éléments correspondants[1]
- (*pat1* , *pat2* , .., *patn*) // même chose que ci-dessus mais ignore tout sauf les premier, deuxième et dernier éléments[1]
- *allumé* // correspond à une constante littérale (char, types numériques, booléen et chaîne)
- *pat1* ... *pat2* // correspond à une valeur dans cette plage (incluse) (types char et numérique)

Remarques

Lors de la déconstruction d'une valeur de structure, le champ doit être de la forme `field_name` ou `field_name : pattern` . Si aucun modèle n'est spécifié, une liaison implicite est effectuée:

```
let Point { x, y } = p;
// equivalent to
let Point { x: x, y: y } = p;

let Point { ref x, ref y } = p;
// equivalent to
let Point { x: ref x, y: ref y } = p;
```

1: modèle irréfutable

Examples

Correspondance de motif avec les liaisons

Il est possible de lier des valeurs à des noms en utilisant `@` :

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

Cela va imprimer:

```
John is 8 years old, he eats honey most of the time
```

Correspondance de base

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

Si nous ne couvrons pas tous les cas, nous aurons une erreur de compilation:

```
match a {
  true => println!("most important case")
}
// error: non-exhaustive patterns: `false` not covered [E0004]
```

Nous pouvons utiliser _ comme cas par défaut / générique, cela correspond à tout:

```
// Create an 32-bit unsigned integer
let b: u32 = 13;

match b {
  0 => println!("b is 0"),
  1 => println!("b is 1"),
  _ => println!("b is something other than 0 or 1")
```

```
}
```

Cet exemple va imprimer:

```
a is true
b is something else than 0 or 1
```

Faire correspondre plusieurs modèles

Il est possible de traiter plusieurs valeurs distinctes de la même manière, en utilisant `|` :

```
enum Colour {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
    Black
}

enum ColourModel {
    RGB,
    CMYK
}

// let's take an example colour
let colour = Colour::Red;

let model = match colour {
    // check if colour is any of the RGB colours
    Colour::Red | Colour::Green | Colour::Blue => ColourModel::RGB,
    // otherwise select CMYK
    _ => ColourModel::CMYK,
};
```

Correspondance conditionnelle avec des gardes

Les modèles peuvent être adaptés en fonction des valeurs indépendantes à la valeur étant en correspondance utilisant `if` les gardes:

```
// Let's imagine a simplistic web app with the following pages:
enum Page {
  Login,
  Logout,
  About,
  Admin
}

// We are authenticated
let is_authenticated = true;

// But we aren't admins
let is_admin = false;
```

```
let accessed_page = Page::Admin;

match accessed_page {
    // Login is available for not yet authenticated users
    Page::Login if !is_authenticated => println!("Please provide a username and a password"),

    // Logout is available for authenticated users
    Page::Logout if is_authenticated => println!("Good bye"),

    // About is a public page, anyone can access it
    Page::About => println!("About us"),

    // But the Admin page is restricted to administators
    Page::Admin if is_admin => println!("Welcome, dear administrator"),

    // For every other request, we display an error message
    _ => println!("Not available")
}
```

Cela affichera *"Non disponible"* .

si let / while laisser

---

if let

Combine une `match` modèle et une instruction `if` et permet d'effectuer des correspondances brèves et non exhaustives.

```
if let Some(x) = option {
    do_something(x);
}
```

Ceci est équivalent à:

```
match option {
    Some(x) => do_something(x),
    _ => {},
}
```

Ces blocs peuvent également avoir d' `else` instructions.

```
if let Some(x) = option {
    do_something(x);
} else {
    panic!("option was None");
}
```

Ce bloc équivaut à:

```
match option {
    Some(x) => do_something(x),
    None => panic!("option was None"),
```

```
}
```

---

while let

Combine une correspondance de motif et une boucle while.

```
let mut cs = "Hello, world!".chars();
while let Some(x) = cs.next() {
    print("{}+", x);
}
println!("");
```

Ceci imprime `H+e+l+l+o+,+ +w+o+r+l+d+!+` .

Cela équivaut à utiliser une `loop {}` et une déclaration de `match` :

```
let mut cs = "Hello, world!".chars();
loop {
    match cs.next() {
        Some(x) => print("{}+", x),
        _ => break,
    }
}
println!("");
```

Extraire des références de motifs

Il est parfois nécessaire de pouvoir extraire des valeurs à partir d'un objet en utilisant uniquement des références (c'est-à-dire sans transférer la propriété).

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };

    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }

    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
```

```
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }

    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

Il imprimera ceci:

```
User token exists? true
Token value: 3
New token value: 5
```

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/rust/topic/1188/correspondance-de-motif

Chapitre 11: Déréférencement automatique

Examples

L'opérateur de points

Le `.` opérateur dans Rust est livré avec beaucoup de magie! Lorsque vous utilisez `.` , le compilateur insérera autant de `*` s (opérations de déréférencement) nécessaires pour trouver la méthode dans l'arborescence "deref". Comme cela se produit au moment de la compilation, il n'y a pas de coût d'exécution pour trouver la méthode.

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

Le déréférencement automatique fonctionne également pour tout type implémentant le trait `std::ops::Deref` .

```
let vec = vec![1, 2, 3];
let iterator = vec.iter();
```

Ici, `iter` n'est pas une méthode de `Vec<T>` , mais une méthode de `[T]` . Cela fonctionne parce que `Vec<T>` implémente `Deref` avec `Target=[T]` ce qui permet à `Vec<T>` transformer en `[T]` quand il est déréférencé par l'opérateur `*` (que le compilateur peut insérer pendant un `.`).

Deref coercions

Avec deux types `T` et `U` , `&T` contraindra (convertit implicitement) à `&U` si et seulement si `T` implémente `Deref<Target=U>`

Cela nous permet de faire des choses comme ceci:

```
fn foo(a: &[i32]) {
    // code
}

fn bar(s: &str) {
    // code
```

```
}

let v = vec![1, 2, 3];
foo(&v); // &Vec<i32> coerces into &[i32] because Vec<T> impls Deref<Target=[T]>

let s = "Hello world".to_string();
let rc = Rc::new(s);
// This works because Rc<T> impls Deref<Target=T> ∴ &Rc<String> coerces into
// &String which coerces into &str. This happens as much as needed at compile time.
bar(&rc);
```

Utiliser Deref et AsRef pour les arguments de fonction

Pour les fonctions nécessitant une collection d'objets, les tranches sont généralement un bon choix:

```
fn work_on_bytes(slice: &[u8]) {}
```

Parce que `Vec<T>` et les tableaux `[T; N]` implémenter `Deref<Target=[T]>` , ils peuvent être facilement forcés à une tranche:

```
let vec = Vec::new();
work_on_bytes(&vec);

let arr = [0; 10];
work_on_bytes(&arr);

let slice = &[1,2,3];
work_on_bytes(slice); // Note lack of &, since it doesn't need coercing
```

Cependant, au lieu d'exiger explicitement une tranche, la fonction peut accepter n'importe quel type *pouvant être* utilisé comme une tranche:

```
fn work_on_bytes<T: AsRef<[u8]>>(input: T) {
    let slice = input.as_ref();
}
```

Dans cet exemple, la fonction `work_on_bytes` prendra tout type `T` qui implémente `as_ref()` , qui renvoie une référence à `[u8]` .

```
work_on_bytes(vec);
work_on_bytes(arr);
work_on_bytes(slice);
work_on_bytes("strings work too!");
```

Implémentation Deref pour Option et structure de wrapper

```
use std::ops::Deref;
use std::fmt::Debug;

#[derive(Debug)]
struct RichOption<T>(Option<T>); // wrapper struct
```

```
impl<T> Deref for RichOption<T> {
    type Target = Option<T>; // Our wrapper struct will coerce into Option
    fn deref(&self) -> &Option<T> {
        &self.0 // We just extract the inner element
    }
}

impl<T: Debug> RichOption<T> {
    fn print_inner(&self) {
        println!("{:?}", self.0)
    }
}

fn main() {
    let x = RichOption(Some(1));
    println!("{:?}",x.map(|x| x + 1)); // Now we can use Option's methods...
    fn_that_takes_option(&x); // pass it to functions that take Option...
    x.print_inner() // and use it's own methods to extend Option
}

fn fn_that_takes_option<T : std::fmt::Debug>(x: &Option<T>) {
    println!("{:?}", x)
}
```

Exemple simple de Deref

`Deref` a une règle simple: si vous avez un type `T` et qu'il implémente `Deref<Target=F>` , alors `&T` faufile vers `&F` , le compilateur répètera cela autant de fois que nécessaire pour obtenir F, par exemple:

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

La contrainte de Deref est particulièrement utile lorsque vous travaillez avec des types de pointeurs, tels que `Box` ou `Arc` , par exemple:

```
fn main() {
    let val = Box::new(vec![1,2,3]);
    // Now, thanks to Deref, we still
    // can use our vector method as if there wasn't any Box
    val.iter().fold(0, |acc, &x| acc + x ); // 6
    // We pass our Box to the function that takes Vec,
    // Box<Vec> coerces to Vec
    f(&val)
}

fn f(x: &Vec<i32>) {
    println!("{:?}", x) // [1,2,3]
}
```

Lire Déréférencement automatique en ligne:

https://riptutorial.com/fr/rust/topic/2574/dereferencement-automatique

Chapitre 12: Dérivé personnalisé: "Macros 1.1"

Introduction

Rust 1.15 a ajouté (stabilisé) une nouvelle fonctionnalité: Dériver de manière personnalisée les macros aka 1.1.

En dehors de `PartialEq` ou `Debug` habituel, vous pouvez avoir `#[deriving(MyOwnDerive)]` . Les deux principaux utilisateurs de cette fonctionnalité sont le serde et le diesel .

Lien Rust Book: https://doc.rust-lang.org/stable/book/procedural-macros.html

Examples

Verbose dumpy helloworld

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(Hello)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    println!("Normalized source code: {}", source);
    let ast = syn::parse_derive_input(&source).unwrap();
    println!("Syn's AST: {:?}", ast); // {:#?} - pretty print
    let struct_name = &ast.ident;
    let quoted_code = quote!{
        fn hello() {
            println!("Hello, {}!", stringify!(#struct_name));
```

```
        }
    };
    println!("Quoted code: {:?}", quoted_code);
    quoted_code.parse().unwrap()
}
```

examples / hello.rs:

```
#[macro_use]
extern crate customderive;

#[derive(Hello)]
struct Qqq;

fn main(){
    hello();
}
```

sortie:

```
$ cargo run --example hello
   Compiling customderive v0.1.1 (file:///tmp/cd)
Normalized source code: struct Qqq;
Syn's AST: DeriveInput { ident: Ident("Qqq"), vis: Inherited, attrs: [], generics: Generics {
lifetimes: [], ty_params: [], where_clause: WhereClause { predicates: [] } }, body:
Struct(Unit) }
Quoted code: Tokens("fn hello ( ) { println ! ( \"Hello, {}!\" , stringify ! ( Qqq ) ) ; }")
warning: struct is never used: <snip>
    Finished dev [unoptimized + debuginfo] target(s) in 3.79 secs
     Running `target/x86_64-unknown-linux-gnu/debug/examples/hello`
Hello, Qqq!
```

Dummy Minimal personnalisé dériver

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(Dummy)]
pub fn qqq(input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

examples / hello.rs

```
#[macro_use]
extern crate customderive;

#[derive(Dummy)]
struct Qqq;

fn main(){}
```

Getters et Setters

Cargo.toml:

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(GetSet)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    let ast = syn::parse_derive_input(&source).unwrap();

    let struct_name = &ast.ident;
    if let syn::Body::Struct(s) = ast.body {
        let field_names : Vec<_> = s.fields().iter().map(|ref x|
                x.ident.clone().unwrap()).collect();

        let field_getter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("get_{}", x).as_str()));
        let field_setter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("set_{}", x).as_str()));
        let field_types : Vec<_> = s.fields().iter().map(|ref x|
                x.ty.clone()).collect();
        let field_names2 = field_names.clone();
        let field_names3 = field_names.clone();
        let field_types2 = field_types.clone();

        let quoted_code = quote!{
            #[allow(dead_code)]
            impl #struct_name {
                #(
                    fn #field_getter_names(&self) -> &#field_types {
                        &self.#field_names2
                    }
```

```
                    fn #field_setter_names(&mut self, x : #field_types2) {
                        self.#field_names3 = x;
                    }
                )*
            }
        };
        return quoted_code.parse().unwrap();
    }
    // not a struct
    "".parse().unwrap()
}
```

examples / hello.rs:

```
#[macro_use]
extern crate gettersetter;

#[derive(GetSet)]
struct Qqq {
    x : i32,
    y : String,
}

fn main(){
    let mut a = Qqq { x: 3, y: "zzaaqq".to_string() };
    println!("{}", a.get_x());
    a.set_y("123213".to_string());
    println!("{}", a.get_y());
}
```

Voir aussi: https://github.com/emk/accessors

Lire Dérivé personnalisé: "Macros 1.1" en ligne: https://riptutorial.com/fr/rust/topic/9104/derive-personnalise---macros-1-1-

Chapitre 13: Des tests

Examples

Tester une fonction

```
fn to_test(output: bool) -> bool {
    output
}

#[cfg(test)] // The module is only compiled when testing.
mod test {
    use super::to_test;

    // This function is a test function. It will be executed and
    // the test will succeed if the function exits cleanly.
    #[test]
    fn test_to_test_ok() {
        assert_eq!(to_test(true), true);
    }

    // That test on the other hand will only succeed when the function
    // panics.
    #[test]
    #[should_panic]
    fn test_to_test_fail() {
        assert_eq!(to_test(true), false);
    }
}
```

(Lien Playground)

Exécuter avec `cargo test` .

Tests d'intégration

`lib.rs` :

```
pub fn to_test(output: bool) -> bool {
    output
}
```

Chaque fichier dans le dossier `tests/` est compilé en tant que caisse unique.
`tests/integration_test.rs`

```
extern crate test_lib;
use test_lib::to_test;

#[test]
fn test_to_test(){
    assert_eq!(to_test(true), true);
}
```

Tests de benchmark

Avec les tests de performances, vous pouvez tester et mesurer la vitesse du code, mais les tests de performances sont toujours instables. Pour activer les tests de performances dans votre projet cargo, vous avez besoin de rouille nocturne, placez vos tests d'intégration sur les `benches/` dossier à la racine de votre projet Cargo et exécutez `cargo bench` .

Exemples de llogiq.github.io

```
extern crate test;
extern crate rand;

use test::Bencher;
use rand::Rng;
use std::mem::replace;

#[bench]
fn empty(b: &mut Bencher) {
    b.iter(|| 1)
}

#[bench]
fn setup_random_hashmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::HashMap::new();

    b.iter(|| { map.insert(rng.gen::<u8>() as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::new();

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap_cap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::with_capacity(256);

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}
```

Lire Des tests en ligne: https://riptutorial.com/fr/rust/topic/961/des-tests

Chapitre 14: Des vies

Syntaxe

- Fonction fn <'a> (x: &' a Type)
- struct Struct <'a> {x: &' a Type}
- enum Enum <'a> {Variante (&' un Type)}
- impl <'a> Struct <' a> {fn x <'a> (& self) -> &' a Type {self.x}}
- impliquer <'a> Trait <' a> pour le type
- impliquer <'a> Trait pour le type <' a>
- `fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)`
- `struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }`
- `enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }`
- `impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }`

Remarques

- Toutes les références dans Rust ont une durée de vie, même si elles ne sont pas explicitement annotées. Le compilateur est capable d'affecter implicitement des durées de vie.
- La `'static` durée de vie `'static` est assignée aux références stockées dans le programme binaire et sera valide pendant toute son exécution. Cette durée de vie est plus particulièrement affectée aux littéraux de chaîne, qui ont le type `&'static str` .

Examples

Paramètres de fonction (durée de vie des entrées)

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

Cela spécifie que `foo` a la durée `'a` vie `'a` et que le paramètre `x` doit avoir une durée de vie d'au moins `'a` . Les durées de vie des fonctions sont généralement omises lors de l' *élision à vie* :

```
fn foo(x: &u32) {
    // ...
}
```

Dans le cas où une fonction prend plusieurs références en tant que paramètres et renvoie une référence, le compilateur ne peut pas déduire la durée de vie du résultat au cours de la durée de *vie* .

```
error[E0106]: missing lifetime specifier
1 | fn foo(bar: &str, baz: &str) -> &i32 {
  |                                 ^ expected lifetime parameter
```

Au lieu de cela, les paramètres de durée de vie doivent être explicitement spécifiés.

```
// Return value of `foo` is valid as long as `bar` and `baz` are alive.
fn foo<'a>(bar: &'a str, baz: &'a str) -> &'a i32 {
```

Les fonctions peuvent également prendre plusieurs paramètres de durée de vie.

```
// Return value is valid for the scope of `bar`
fn foo<'a, 'b>(bar: &'a str, baz: &'b str) -> &'a i32 {
```

Champs Struct

```
struct Struct<'a> {
    x: &'a u32,
}
```

Cela spécifie que toute instance donnée de `Struct` a la durée `'a` vie `'a` et que le `&u32` stocké dans `x` doit avoir une durée de vie d'au moins `'a` .

Blocs d'implants

```
impl<'a> Type<'a> {
    fn my_function(&self) -> &'a u32 {
        self.x
    }
}
```

Cela spécifie que `Type` a la durée `'a` vie `'a` et que la référence renvoyée par `my_function()` peut ne plus être valide après `'a` ends car le `Type` n'existe plus pour contenir `self.x`

Limites de trait de rang supérieur

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

Cela spécifie que la référence sur i32 dans le trait `Fn` lié peut avoir n'importe quelle durée de vie.

Les éléments suivants ne fonctionnent pas:

```
fn wrong_copy_if<'a, F>(slice: &[i32], pred: F) -> Vec<i32>
    where F: Fn(&'a i32) -> bool
{                                   // <-----------------+
```

```
    let mut result = vec![];        //       'a scope   |
    for &element in slice {         // <--------+       |
        if pred(&element) {         //          |       |
            result.push(element);   // element's|       |
        }                           //   scope  |       |
    }                               // <--------+       |
    result                          //                  |
}                                   // <-----------------+
```

Le compilateur donne l'erreur suivante:

```
error: `element` does not live long enough
if pred(&element) {         //          |       |
         ^~~~~~~
```

parce que l' `element` variable locale ne vit pas aussi longtemps que la `'a` vie (comme on peut le voir dans les commentaires du code).

La durée de vie ne peut pas être déclarée au niveau de la fonction, car nous avons besoin d'une autre durée de vie. C'est pourquoi nous avons utilisé `for<'a>` : pour spécifier que la référence peut être valide pour n'importe quelle durée de vie (une durée de vie plus courte peut donc être utilisée).

Les limites de trait de rang supérieur peuvent également être utilisées sur des structures:

```
struct Window<F>
    where for<'a> F: FnOnce(&'a Window<F>)
{
    on_close: F,
}
```

ainsi que sur d'autres articles.

Les limites de trait de rang supérieur sont les plus couramment utilisées avec les traits `Fn*` .

Pour ces exemples, l'élision à vie fonctionne bien, nous n'avons donc pas à spécifier les durées de vie.

Lire Des vies en ligne: https://riptutorial.com/fr/rust/topic/2074/des-vies

Chapitre 15: Directives dangereuses

Introduction

Expliquez pourquoi certaines choses sont marquées comme `unsafe` dans Rust et pourquoi nous pourrions avoir besoin d'utiliser cette trappe d'évacuation dans certaines situations (rares).

Examples

Courses de données

Les courses de données se produisent quand une partie de la mémoire est mise à jour par une partie, tandis qu'une autre tente de la lire ou de la mettre à jour simultanément (sans synchronisation entre les deux). Regardons l'exemple classique d'une course de données en utilisant un compteur partagé.

```
use std::cell::UnsafeCell;
use std::sync::Arc;
use std::thread;

// `UnsafeCell` is a zero-cost wrapper which informs the compiler that "what it
// contains might be shared mutably." This is used only for static analysis, and
// gets optimized away in release builds.
struct RacyUsize(UnsafeCell<usize>);

// Since UnsafeCell is not thread-safe, the compiler will not auto-impl Sync for
// any type containig it. And manually impl-ing Sync is "unsafe".
unsafe impl Sync for RacyUsize {}

impl RacyUsize {
    fn new(v: usize) -> RacyUsize {
        RacyUsize(UnsafeCell::new(v))
    }

    fn get(&self) -> usize {
        // UnsafeCell::get() returns a raw pointer to the value it contains
        // Dereferencing a raw pointer is also "unsafe"
        unsafe { *self.0.get() }
    }

    fn set(&self, v: usize) { // note: `&self` and not `&mut self`
        unsafe { *self.0.get() = v }
    }
}

fn main() {
    let racy_num = Arc::new(RacyUsize::new(0));

    let mut handlers = vec![];
    for _ in 0..10 {
        let racy_num = racy_num.clone();
        handlers.push(thread::spawn(move || {
            for i in 0..1000 {
```

```
                if i % 200 == 0 {
                    // give up the time slice to scheduler
                    thread::yield_now();
                    // this is needed to interleave the threads so as to observe
                    // data race, otherwise the threads will most likely be
                    // scheduled one after another.
                }

                // increment by one
                racy_num.set(racy_num.get() + 1);
            }
        }));
    }

    for th in handlers {
        th.join().unwrap();
    }

    println!("{}", racy_num.get());
}
```

La sortie sera presque toujours inférieure à `10000` (10 threads × 1000) lorsqu'elle est exécutée sur un processeur multicœur.

Dans cet exemple, une course de données a produit une valeur logiquement incorrecte mais toujours significative. C'est parce que seul un mot a été impliqué dans la course et donc une mise à jour n'a pas pu la modifier partiellement. Mais les courses de données en général peuvent produire des valeurs de corruption non valides pour un type (type non sécurisé) lorsque l'objet en cours de course couvre plusieurs mots et / ou produire des valeurs pointant vers des emplacements mémoire invalides (mémoire non sécurisée).

Toutefois, une utilisation prudente des primitives atomiques peut permettre la construction de structures de données très efficaces pouvant nécessiter certaines de ces opérations "peu sûres" pour effectuer des actions qui ne sont pas vérifiables statiquement par le système de type Rust, abstraction).

Lire Directives dangereuses en ligne: https://riptutorial.com/fr/rust/topic/6018/directives-dangereuses

Chapitre 16: Documentation

Introduction

Le compilateur de Rust a plusieurs fonctionnalités pratiques pour documenter votre projet rapidement et facilement. Vous pouvez utiliser les analyses du compilateur pour appliquer la documentation pour chaque fonction et avoir des tests intégrés à vos exemples.

Syntaxe

- /// Commentaire sur la documentation externe (s'applique à l'article ci-dessous)
- //! Commentaire sur la documentation interne (s'applique à l'élément englobant)
- cargo doc # Génère de la documentation pour cette caisse de bibliothèque.
- cargo doc --open # Génère de la documentation pour cette bibliothèque et ce navigateur ouvert.
- cargo doc -p CRATE # Génère de la documentation pour la caisse spécifiée uniquement.
- cargo doc --no-deps # Génère de la documentation pour cette bibliothèque et aucune dépendance.
- test de la cargaison # Exécute des tests unitaires et des tests de documentation.

Remarques

Cette section du «Rust Book» peut contenir des informations utiles sur les tests de documentation et de documentation.

Les commentaires sur la documentation peuvent être appliqués à:

- Modules
- Structs
- Enums
- Les méthodes
- Les fonctions
- Traits et Méthodes de Trait

Examples

Lint de documentation

Pour vous assurer que tous les éléments possibles sont documentés, vous pouvez utiliser le lien `missing_docs`

pour recevoir des avertissements / erreurs du compilateur. Pour recevoir des avertissements à l'échelle de la bibliothèque, placez cet attribut dans votre fichier `lib.rs` :

```
#![warn(missing_docs)]
```

Vous pouvez également recevoir des erreurs pour la documentation manquante avec cette fibre:

```
#![deny(missing_docs)]
```

Par défaut, `missing_docs` est autorisé, mais vous pouvez les autoriser explicitement avec cet attribut:

```
#![allow(missing_docs)]
```

Cela peut être utile de placer dans un module pour permettre la documentation manquante pour un module, mais le refuser dans tous les autres fichiers.

Commentaires sur la documentation

Rust fournit deux types de commentaires sur la documentation: les commentaires de la documentation interne et les commentaires de la documentation externe. Des exemples de chacun sont fournis ci-dessous.

Commentaires sur la documentation interne

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

Commentaires sur la documentation externe

```
/// Outer documentation comments go *outside* the item that they refer to.
/// They use the syntax /// to distinguish them from inner comments.
pub enum Test {
    Success,
    Fail(Error)
}
```

Conventions

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
```

```
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```

```
/// It is considered good practice to have examples in your documentation
/// under an "Examples" header, like this:
/// # Examples
/// Code can be added in "fences" of 3 backticks.
///
/// ```
/// let bar = Bar::new();
/// ```
///
/// Examples also function as tests to ensure the examples actually compile.
/// The compiler will automatically generate a main() function and run the
/// example code as a test when cargo test is run.
struct Bar {
    ...
}
```

Tests de documentation

Le code dans les commentaires de documentation sera automatiquement exécuté par `cargo test` . Celles-ci sont appelées "tests de documentation" et permettent de s'assurer que vos exemples fonctionnent et n'induisent pas les utilisateurs en erreur.

Vous pouvez importer par rapport à la racine de la caisse (comme s'il y avait une `extern crate mycrate;` masquée `extern crate mycrate;` en haut de l'exemple)

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

Si votre code risque de ne pas s'exécuter correctement dans un test de documentation, vous pouvez utiliser l'attribut `no_run` , comme ceci:

```
/// ```no_run
/// use mycrate::NetworkClient;
/// NetworkClient::login("foo", "bar");
/// ```
```

Vous pouvez également indiquer que votre code *devrait* paniquer, comme ceci:

```
/// ```should_panic
/// unreachable!();
/// ```
```

Lire Documentation en ligne: https://riptutorial.com/fr/rust/topic/4865/documentation

Chapitre 17: Fermetures et expressions lambda

Examples

Expressions lambda simples

```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```

Cela affiche:

```
8
15
```

Fermetures simples

Contrairement aux fonctions régulières, les expressions lambda peuvent capturer leurs environnements. Ces lambda sont appelées fermetures.

```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;

// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```

Cela va imprimer:

```
663
```

Lambdas avec des types de retour explicites

```
// lambda expressions can have explicitly annotated return types
let floor_func = |x: f64| -> i64 { x.floor() as i64 };
```

Passer des lambdas autour

Comme les fonctions lambda sont des valeurs elles-mêmes, vous les stockez dans des collections, vous les transmettez à des fonctions, etc., comme vous le feriez avec d'autres valeurs.

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

Cela va imprimer:

```
3 + 6 = 9
-4 * 9 = -36
7 - (-3) = 10
```

Retour de lambda de fonctions

Renvoyer des lambdas (ou des fermetures) à partir de fonctions peut être délicat car ils implémentent des traits et leur taille exacte est donc rarement connue.

```
// Box in the return type moves the function from the stack to the heap
fn curried_adder(a: i32) -> Box<Fn(i32) -> i32> {
    // 'move' applies move semantics to a, so it can outlive this function call
    Box::new(move |b| a + b)
}

println!("3 + 4 = {}", curried_adder(3)(4));
```

Cela affiche: `3 + 4 = 7`

Lire Fermetures et expressions lambda en ligne:
https://riptutorial.com/fr/rust/topic/1815/fermetures-et-expressions-lambda

Chapitre 18: Fichier I / O

Examples

Lire un fichier dans son ensemble en tant que chaîne

```
use std::fs::File;
use std::io::Read;

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode.
    match File::open(filename) {
        // The file is open (no error).
        Ok(mut file) => {
            let mut content = String::new();

            // Read all the file content into a variable (ignoring the result of the
operation).
            file.read_to_string(&mut content).unwrap();

            println!("{}", content);

            // The file is automatically closed when is goes out of scope.
        },
        // Error handling.
        Err(error) => {
            println!("Error opening file {}: {}", filename, error);
        },
    }
}
```

Lire un fichier ligne par ligne

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode (ignoring errors).
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);

    // Read the file line by line using the lines() iterator from std::io::BufRead.
    for (index, line) in reader.lines().enumerate() {
        let line = line.unwrap(); // Ignore errors.
        // Show the line and its number.
        println!("{}. {}", index + 1, line);
    }
}
```

Ecrire dans un fichier

```
use std::env;
use std::fs::File;
use std::io::Write;

fn main() {
    // Create a temporary file.
    let temp_directory = env::temp_dir();
    let temp_file = temp_directory.join("file");

    // Open a file in write-only (ignoring errors).
    // This creates the file if it does not exist (and empty the file if it exists).
    let mut file = File::create(temp_file).unwrap();

    // Write a &str in the file (ignoring the result).
    writeln!(&mut file, "Hello World!").unwrap();

    // Write a byte string.
    file.write(b"Bytes\n").unwrap();
}
```

Lire un fichier en tant que Vec

```
use std::fs::File;
use std::io::Read;

fn read_a_file() -> std::io::Result<Vec<u8>> {
    let mut file = try!(File::open("example.data"));

    let mut data = Vec::new();
    try!(file.read_to_end(&mut data));

    return Ok(data);
}
```

`std::io::Result<T>` est un alias pour le `Result<T, std::io::Error>` .

La macro `try!()` Renvoie la fonction en cas d'erreur.

`read_to_end()` est une méthode de `std::io::Read` trait, qui doit être `use` explicitement d.

`read_to_end()` ne renvoie pas les données lues. Au lieu de cela, il place des données dans le conteneur qu'il a fourni.

Lire Fichier I / O en ligne: https://riptutorial.com/fr/rust/topic/1307/fichier-i---o

Chapitre 19: Futures et Async IO

Introduction

futures-rs est une bibliothèque qui implémente des contrats à terme et des flux à coût zéro dans Rust.

Les concepts de base de la caisse à terme sont Future et Stream .

Examples

Créer un avenir avec la fonction onehot

Il y a quelques implémentations générales de traits d' `Future` dans la caisse à terme . L'une d'entre elles est implémentée dans le module `futures::sync::oneshot` et est disponible via les `futures::oneshot` fonction `futures::oneshot` :

```
extern crate futures;

use std::thread;
use futures::Future;

fn expensive_computation() -> u32 {
    // ...
    200
}

fn main() {
    // The oneshot function returns a tuple of a Sender and a Receiver.
    let (tx, rx) = futures::oneshot();

    thread::spawn(move || {
        // The complete method resolves a values.
        tx.complete(expensive_computation());
    });

    // The map method applies a function to a value, when it is resolved.
    let rx = rx.map(|x| {
        println!("{}", x);
    });

    // The wait method blocks current thread until the value is resolved.
    rx.wait().unwrap();
}
```

Lire Futures et Async IO en ligne: https://riptutorial.com/fr/rust/topic/8595/futures-et-async-io

Chapitre 20: Génération de nombres aléatoires

Introduction

Rust a une capacité intégrée pour fournir une génération de nombres aléatoires à travers la caisse de *rand* . Une fois partie de la bibliothèque standard Rust, la fonctionnalité du *rand* crate a été séparée pour permettre à son développement de se stabiliser séparément du reste du projet Rust. Ce sujet couvrira comment ajouter simplement le rand crate, puis générer et sortir un nombre aléatoire dans Rust.

Remarques

Il existe un support intégré pour un RNG associé à chaque thread stocké dans un stockage local. Ce RNG est accessible via `thread_rng` , ou implicitement via le `random` . Ce RNG est normalement extrait au hasard d'une source aléatoire du système d'exploitation, par exemple `/dev/urandom` sur les systèmes Unix, et se redimensionnera automatiquement à partir de cette source après avoir généré 32 KiB de données aléatoires.

Une application nécessitant une source d'entropie à des fins de cryptographie doit utiliser `OsRng` , qui lit le caractère aléatoire de la source fournie par le système d'exploitation (par exemple, `/dev/urandom` sur Unix ou `CryptGenRandom()` sous Windows). Les autres générateurs de nombres aléatoires fournis par ce module ne sont pas adaptés à de telles fins.

Examples

Générer deux nombres aléatoires avec Rand

Tout d'abord, vous devrez ajouter la caisse dans votre fichier Cargo.toml en tant que dépendance.

```
[dependencies]
rand = "0.3"
```

Cela permettra de récupérer le `rand` crate.io . Ensuite, ajoutez ceci à votre racine de caisse.

```
extern crate rand;
```

Comme cet exemple va fournir une sortie simple via le terminal, nous allons créer une fonction principale et imprimer deux nombres générés aléatoirement dans la console. Le générateur de nombres aléatoires local de thread sera mis en cache dans cet exemple. Lorsque vous générez plusieurs valeurs, cela peut souvent s'avérer plus efficace.

```
use rand::Rng;
```

```
fn main() {

    let mut rng = rand::thread_rng();

    if rng.gen() { // random bool
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>())
    }
}
```

Lorsque vous exécutez cet exemple, vous devriez voir la réponse suivante dans la console.

```
$ cargo run
     Running `target/debug/so`
i32: 1568599182, u32: 2222135793
```

Générer des caractères avec Rand

Pour générer des caractères, vous pouvez utiliser la fonction de générateur de nombres aléatoires local au fil, `random` .

```
fn main() {
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

Pour les demandes ponctuelles ou singulières, comme celle ci-dessus, il s'agit d'une méthode efficace et raisonnable. Cependant, si vous avez l'intention de générer plus d'une poignée de chiffres, vous constaterez que la mise en cache du générateur sera plus efficace.

Vous devriez vous attendre à voir la sortie suivante dans ce cas.

```
$ cargo run
     Running `target/debug/so`
(0.906881, '\u{9edc}')
```

Lire Génération de nombres aléatoires en ligne:
https://riptutorial.com/fr/rust/topic/8864/generation-de-nombres-aleatoires

Chapitre 21: Génériques

Examples

Déclaration

```
// Generic types are declared using the <T> annotation

struct GenericType<T> {
    pub item: T
}

enum QualityChecked<T> {
    Excellent(T),
    Good(T),
    // enum fields can be generics too
    Mediocre { product: T }
}
```

Instanciation

```
// explicit type declaration
let some_value: Option<u32> = Some(13);

// implicit type declaration
let some_other_value = Some(66);
```

Paramètres de type multiple

Les types génériques peuvent avoir plusieurs paramètres de type, par exemple. `Result` est défini comme ceci:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Types génériques liés

```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

Les bornes doivent couvrir toutes les utilisations du type. L'ajout est fait par le trait `std::ops::Add` ,

qui a lui-même des paramètres d'entrée et de sortie. `where T: std::ops::Add<u32,Output=U>` indique qu'il est possible d' `Add T` à `u32` , et que cette addition doit produire le type `U`

```
fn try_add_one<T, U>(input_value: T) -> Result<U, String>
    where T: std::ops::Add<u32,Output=U>
{
    return Ok(input_value + 1);
}
```

`Sized` limite est implicite par défaut. `?Sized` bound permet également les types non personnalisés.

Fonctions génériques

Les fonctions génériques permettent de paramétrer tout ou partie de leurs arguments.

```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```

Si le compilateur ne peut pas déduire le paramètre de type, il peut être fourni manuellement lors de l'appel:

```
let result: Result<u32, String> = convert_value::<f64, u32>(13.5);
```

Lire Génériques en ligne: https://riptutorial.com/fr/rust/topic/1801/generiques

Chapitre 22: Globals

Syntaxe

- const IDENTIFIER: type = constexpr;
- static [mut] IDENTIFICATEUR: type = expr;
- lazy_static! {static ref IDENTIFIER: type = expr; }

Remarques

- `const` valeurs `const` sont toujours en ligne et n'ont pas d'adresse en mémoire.
- `static` valeurs `static` ne sont jamais intégrées et comportent une instance avec une adresse fixe.
- `static mut` valeurs de `static mut` ne sont pas sûres pour la mémoire et ne peuvent donc être accédées que dans un bloc `unsafe` .
- Parfois, l'utilisation de variables mutables statiques globales dans du code multithread peut être dangereuse, alors envisagez d'utiliser std :: sync :: Mutex ou d'autres alternatives
- `lazy_static` objets `lazy_static` sont immuables, ne sont initialisés qu'une seule fois, sont partagés entre tous les threads et peuvent être directement accessibles (il n'y a pas de type de wrapper). En revanche, les objets `thread_local` sont censés être mutables, sont initialisés une fois pour chaque thread et les accès sont indirects (impliquant le type de wrapper `LocalKey<T>`)

Examples

Const

Le mot-clé `const` déclare une liaison de constante globale.

```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```

Cette sorties

```
DEADBEEF
```

Statique

Le mot-clé `static` déclare une liaison statique globale, qui peut être mutable.

```
static HELLO_WORLD: &'static str = "Hello, world!";
```

```
fn main() {
    println("{}", HELLO_WORLD);
}
```

Cette sorties

```
Hello, world!
```

lazy_static!

Utilisez la caisse `lazy_static` pour créer des variables immuables globales qui sont initialisées à l'exécution. Nous utilisons `HashMap` comme démonstration.

Dans `Cargo.toml` :

```
[dependencies]
lazy_static = "0.1.*"
```

En `main.rs` :

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "hello");
        m.insert(1, ",");
        m.insert(2, " ");
        m.insert(3, "world");
        m
    };
    static ref COUNT: usize = HASHMAP.len();
}

fn main() {
    // We dereference COUNT because it's type is &usize
    println!("The map has {} entries.", *COUNT);

    // Here we don't dereference with * because of Deref coercions
    println!("The entry for `0` is \"{}\".", HASHMAP.get(&0).unwrap());
}
```

Thread-local Objects

Un objet thread-local est initialisé lors de sa première utilisation dans un thread. Et comme son nom l'indique, chaque thread aura une nouvelle copie indépendante des autres threads.

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
```

```
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

Les sorties:

```
inner: 1
main: 3
```

Mut statique sûr avec mut_static

Les éléments globaux mutables (appelés `static mut` , mettant en évidence la contradiction inhérente à leur utilisation) sont dangereux car il est difficile pour le compilateur de s’assurer qu’ils sont utilisés correctement.

Cependant, l'introduction de verrous mutuellement exclusifs autour des données permet des globaux mutables sécurisés par la mémoire. Cela ne signifie pas qu'ils sont logiquement sûrs, cependant!

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;

use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
```

```
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}

fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
         // Using it mutably is too...
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

Ce code produit la sortie:

```
Before mut: 0
Changed value to 3.
After mut: 3
After foo: 4
```

Ce n'est pas quelque chose qui devrait arriver à Rust normalement. `foo()` n'a pas pris de

référence mutable, alors ça n'aurait pas dû faire de mutations, et pourtant il l'a fait. Cela peut rendre très difficile le débogage des erreurs de logique.

D'un autre côté, c'est parfois exactement ce que vous voulez. Par exemple, de nombreux moteurs de jeu nécessitent un cache global d’images et d’autres ressources qui est chargé paresseusement (ou utilise une autre stratégie de chargement complexe) - MutStatic est parfait pour cela.

Lire Globals en ligne: https://riptutorial.com/fr/rust/topic/1244/globals

Chapitre 23: Guide de style rouille

Introduction

Bien qu'il n'y ait pas de guide officiel de style Rust, les exemples suivants montrent les conventions adoptées par la plupart des projets Rust. Suivre ces conventions alignera le style de votre projet sur celui de la bibliothèque standard, ce qui facilitera la lecture de la logique dans votre code.

Remarques

Les directives officielles de style Rust étaient disponibles dans le dépôt Rust `rust-lang/rust` Rust sur GitHub, mais elles ont été supprimées récemment, en attendant la migration vers le dépôt `rust-lang-nursery/fmt-rfcs` . Tant que de nouvelles directives ne sont pas publiées, vous devriez essayer de suivre les instructions du référentiel `rust-lang` .

Vous pouvez utiliser rustfmt et clippy pour vérifier automatiquement votre code pour les problèmes de style et le formater correctement. Ces outils peuvent être installés avec Cargo, comme ceci:

```
cargo install clippy
cargo install rustfmt
```

Pour les exécuter, vous utilisez:

```
cargo clippy
cargo fmt
```

Examples

Espace blanc

Longueur de la ligne

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

Échancrure

```
// You should always use 4 spaces for indentation.
// Tabs are discouraged - if you can, set your editor to convert
// a tab into 4 spaces.
let x = vec![1, 3, 5, 6, 7, 9];
for item in x {
    if x / 2 == 3 {
```

```
        println!("{}", x);
    }
}
```

Trailing Whitespace

Les espaces à la fin des fichiers ou des lignes doivent être supprimés.

Opérateurs binaires

```
// For clarity, always add a space when using binary operators, e.g.
// +, -, =, *
let bad=3+4;
let good = 3 + 4;
```

Cela s'applique également aux attributs, par exemple:

```
// Good:
#[deprecated = "Don't use my class - use Bar instead!"]

// Bad:
#[deprecated="This is broken"]
```

Les points-virgules

```
// There is no space between the end of a statement
// and a semicolon.

let bad = Some("don't do this!") ;
let good: Option<&str> = None;
```

Alignement des champs de structure

```
// Struct fields should **not** be aligned using spaces, like this:
pub struct Wrong {
    pub x  : i32,
    pub foo: i64
}

// Instead, just leave 1 space after the colon and write the type, like this:
pub struct Right {
    pub x: i32,
    pub foo: i64
}
```

Signatures de fonction

```
// Long function signatures should be wrapped and aligned so that
// the starting parameter of each line is aligned
fn foo(example_item: Bar, another_long_example: Baz,
       yet_another_parameter: Quux)
       -> ReallyLongReturnItem {
    // Be careful to indent the inside block correctly!
}
```

Bretelles

```
// The starting brace should always be on the same line as its parent.
// The ending brace should be on its own line.
fn bad()
{
    println!("This is incorrect.");
}

struct Good {
    example: i32
}

struct AlsoBad {
    example: i32 }
```

Créer des caisses

Préludes et réexportations

```
// To reduce the amount of imports that users need, you should
// re-export important structs and traits.
pub use foo::Client;
pub use bar::Server;
```

Parfois, les caisses utilisent un module de `prelude` pour contenir des structures importantes, tout comme `std::io::prelude` . Habituellement, ceux-ci sont importés avec `use std::io::prelude::*;`

Importations

Vous devez commander vos importations et vos déclarations comme suit:

- déclarations de `extern crate`
- `use` importations
    - Les importations externes provenant d'autres caisses devraient être prioritaires
- Re-exportations (`pub use`)

Appellation

Structs

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {
```

```
    }

    // Good:
    pub struct Ordered {

    }
}
```

Traits

```
// Traits use the same naming principles as
// structs (UpperCamelCase).
trait Read {
    fn read_to_snafucator(&self) -> Result<(), Error>;
}
```

Caisses et Modules

```
// Modules and crates should both use snake_case.
// Crates should try to use single words if possible.
extern crate foo;
mod bar_baz {
    mod quux {

    }
}
```

Variables statiques et constantes

```
// Statics and constants use SCREAMING_SNAKE_CASE.
const NAME: &'static str = "SCREAMING_SNAKE_CASE";
```

Enums

```
// Enum types and their variants **both** use UpperCamelCase.
pub enum Option<T> {
   Some(T),
   None
}
```

Fonctions et méthodes

```
// Functions and methods use snake_case
fn snake_cased_function() {

}
```

Liaisons variables

```
// Regular variables also use snake_case
let foo_bar = "snafu";
```

Des vies

```
// Lifetimes should consist of a single lower case letter. By
// convention, you should start at 'a, then 'b, etc.

// Good:
struct Foobar<'a> {
    x: &'a str
}

// Bad:
struct Bazquux<'stringlife> {
    my_str: &'stringlife str
}
```

Acronymes

Les noms de variables contenant des acronymes, tels que `TCP` doivent être libellés comme suit:

- Pour les noms `UpperCamelCase` , la **première lettre** doit être en majuscule (par exemple, `TcpClient`)
- Pour les noms `snake_case` , il ne devrait pas y avoir de majuscule (par exemple, `tcp_client`)
- Pour les noms `SCREAMING_SNAKE_CASE` , l'acronyme doit être complètement mis en majuscule (par exemple, `TCP_CLIENT`)

Les types

Type Annotations

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```

Les références

```
// The ampersand (&) of a reference should be 'touching'
// the type it refers to.

// GOOD:
let x: &str = "Hello, world.";
// BAD:
fn fooify(x: & str) {
    println!("{}", x);
}
```

```
// Mutable references should be formatted like so:
fn bar(buf: &mut String) {

}
```

Lire Guide de style rouille en ligne: https://riptutorial.com/fr/rust/topic/4620/guide-de-style-rouille

Chapitre 24: Interface de fonction étrangère (FFI)

Syntaxe

- # [link (name = "snappy")] // la bibliothèque étrangère à associer (facultatif)

  extern {...} // liste des signatures de fonctions dans la bibliothèque étrangère

Examples

Appeler la fonction libc à partir de la rouille nocturne

La caisse de `libc` est « gated » et n'est accessible que sur les versions de Rust jusqu'à ce qu'elle soit considérée comme stable.

```
#![feature(libc)]
extern crate libc;
use libc::pid_t;

#[link(name = "c")]
extern {
    fn getpid() -> pid_t;
}

fn main() {
    let x = unsafe { getpid() };
    println!("Process PID is {}", x);
}
```

Lire Interface de fonction étrangère (FFI) en ligne:
https://riptutorial.com/fr/rust/topic/6140/interface-de-fonction-etrangere--ffi-

Chapitre 25: La gestion des erreurs

Introduction

Rust utilise les valeurs `Result<T, E>` pour indiquer les erreurs récupérables lors de l'exécution. Les erreurs irrécupérables provoquent des paniques qui sont un sujet à part.

Remarques

Les détails de la gestion des erreurs sont décrits dans *The Rust Programming Language* (aka The Book)

Examples

Méthodes de résultat communes

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
```

```
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

Si les fichiers de configuration n'existent pas, cela génère:

```
Error: Failed to read config file: No such file or directory (os error 2)
```

Si l'analyse échoue, cela génère:

```
Error: Failed to parse the config string!
```

Remarque: Au fur et à mesure de la croissance du projet, il sera difficile de gérer les erreurs avec ces méthodes de base (docs) sans perdre d'informations sur l'origine et le chemin de propagation des erreurs. En outre, convertir les erreurs en chaînes prématurément afin de gérer plusieurs types d'erreur, comme indiqué ci-dessus, est une mauvaise pratique. Une méthode bien meilleure consiste à utiliser la `error-chain` caisses.

Types d'erreur personnalisés

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)] // Allow the use of "{:?}" format specifier
enum CustomError {
    Io(IoError),
    Utf8(Utf8Error),
    Other,
}
```

```
// Allow the use of "{}" format specifier
impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CustomError::Io(ref cause) => write!(f, "I/O Error: {}", cause),
            CustomError::Utf8(ref cause) => write!(f, "UTF-8 Error: {}", cause),
            CustomError::Other => write!(f, "Unknown error!"),
        }
    }
}

// Allow this type to be treated like an error
impl Error for CustomError {
    fn description(&self) -> &str {
        match *self {
            CustomError::Io(ref cause) => cause.description(),
            CustomError::Utf8(ref cause) => cause.description(),
            CustomError::Other => "Unknown error!",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CustomError::Io(ref cause) => Some(cause),
            CustomError::Utf8(ref cause) => Some(cause),
            CustomError::Other => None,
        }
    }
}

// Support converting system errors into our custom error.
// This trait is used in `try!`.
impl From<IoError> for CustomError {
    fn from(cause: IoError) -> CustomError {
        CustomError::Io(cause)
    }
}
impl From<Utf8Error> for CustomError {
    fn from(cause: Utf8Error) -> CustomError {
        CustomError::Utf8(cause)
    }
}
```

Itérer à travers les causes

Il est souvent utile, à des fins de débogage, de trouver la cause première d'une erreur. Afin d'examiner une valeur d'erreur qui implémente `std::error::Error` :

```
use std::error::Error;

let orig_error = call_returning_error();

// Use an Option<&Error>. This is the return type of Error.cause().
let mut err = Some(&orig_error as &Error);

// Print each error's cause until the cause is None.
while let Some(e) = err {
    println!("{}", e);
```

```
    err = e.cause();
}
```

Rapport d'erreur et gestion de base

`Result<T, E>` est un type `enum` qui a deux variantes: `Ok(T)` indiquant une exécution réussie avec un résultat significatif de type `T` , et `Err(E)` indiquant une erreur inattendue lors de l'exécution, décrite par une valeur de type `E` .

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

Voir aussi docs pour plus de détails sur `?` opérateur.

La bibliothèque standard contient un trait d' `Error` que tous les types d'erreur sont recommandés pour implémenter. Un exemple de mise en œuvre est donné ci-dessous.

```
use std::error::Error;
use std::fmt;

#[derive(Debug)]
enum DateError {
    InvalidDay(u8),
    InvalidMonth(u8),
}

impl fmt::Display for DateError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            &DateError::InvalidDay(day) => write!(f, "Day {} is outside range!", day),
            &DateError::InvalidMonth(month) => write!(f, "Month {} is outside range!", month),
        }
```

```
    }
}

impl Error for DateError {
    fn description(&self) -> &str {
        match self {
            &DateError::InvalidDay(_) => "Day is outside range!",
            &DateError::InvalidMonth(_) => "Month is outside range!",
        }
    }

    // cause method returns None by default
}
```

Remarque: En règle générale, l' `Option<T>` ne doit pas être utilisée pour signaler des erreurs. `Option<T>` indique une possibilité attendue de non-existence d'une valeur et une seule raison simple. En revanche, le `Result<T, E>` est utilisé pour signaler des erreurs inattendues lors de l'exécution, en particulier lorsqu'il existe plusieurs modes de défaillance pour les distinguer. De plus, le `Result<T, E>` est utilisé uniquement comme valeur de retour. (Une vieille discussion.)

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/rust/topic/1762/la-gestion-des-erreurs

Chapitre 26: La possession

Introduction

La propriété est l'un des concepts les plus importants de Rust, et elle n'est pas présente dans la plupart des autres langues. L'idée qu'une valeur puisse *appartenir* à une variable particulière est souvent assez difficile à comprendre, en particulier dans les langues où la copie est implicite, mais cette section passera en revue les différentes idées concernant la propriété.

Syntaxe

- Soit x: & T = ... // x est une référence immuable
- Soit x: & mut T = ... // x est une référence exclusive, mutable
- let _ = & mut foo; // emprunter de façon mutuelle (c'est-à-dire exclusivement)
- let _ = & foo; // emprunter immanquablement
- let _ = foo; // déplace foo (nécessite la propriété)

Remarques

- Dans les versions beaucoup plus anciennes de Rust (avant 1.0; mai 2015), une variable possédée avait un type commençant par `~` . Vous pouvez voir cela dans des exemples très anciens.

Examples

Propriété et emprunt

Toutes les valeurs dans Rust ont exactement un propriétaire. Le propriétaire est responsable de la suppression de cette valeur lorsqu'il est hors de portée et est le seul à pouvoir *transférer* la propriété de la valeur. Le propriétaire d'une valeur peut lui donner des *références* en laissant d'autres parties de code *emprunter* cette valeur. À tout moment, il peut y avoir un certain nombre de références immuables à une valeur:

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

ou une seule référence mutable (`ERROR` indique une `ERROR` compilation):

```
// for us to borrow a value mutably, it must be mutable
let mut owned = String::from("hello");
// we can borrow owned mutably
let mutable_borrow = &mut owned;
// but note that we cannot borrow owned *again*
let mutable_borrow2 = &mut owned; // ERROR, already borrowed
// nor can we cannot borrow owned immutably
// since a mutable borrow is exclusive.
let immutable_borrow = &owned; // ERROR, already borrowed
```

S'il existe des références en attente (mutables ou immuables) à une valeur, cette valeur ne peut pas être déplacée (c.-à-d. Que sa propriété est donnée). Nous devrions nous assurer que toutes les références ont été supprimées en premier pour pouvoir déplacer une valeur:

```
let foo = owned; // ERROR, outstanding references to owned
let owned = String::from("hello");
{
    let borrow = &owned;
    // ...
} // the scope ends the borrow
let foo = owned; // OK, owned and not borrowed
```

Emprunts et vies

Toutes les valeurs de Rust ont une *durée de vie* . La durée de vie d'une valeur couvre le segment de code de la valeur introduit à l'endroit où il est déplacé ou la fin de l'étendue contenant

```
{
    let x = String::from("hello"); //             +
    // ...                                       :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

Chaque fois que vous empruntez une valeur, la référence résultante a une *durée de vie* liée à la durée de vie de la valeur empruntée:

```
{
    let x = String::from("hello");
    let y = String::from("world");
    // when we borrow y here, the lifetime of the reference
    // stored in foo is equal to the lifetime of y
    // (i.e., between let y = above, to the end of the scope below)
    let foo = &y;
    // similarly, this reference to x is bound to the lifetime
    // of x --- bar cannot, for example, spawn a thread that uses
    // the reference beyond where x is moved below.
    bar(&x);
}
```

Appels de propriété et fonction

La plupart des questions relatives à la propriété apparaissent lors de l'écriture de fonctions. Lorsque vous spécifiez les types d'arguments d'une fonction, vous pouvez choisir la *manière dont* cette valeur est transmise. Si vous avez uniquement besoin d'un accès en lecture seule, vous pouvez prendre une référence immuable:

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

Si `foo` besoin de modifier l'argument, il devrait prendre une référence exclusive, mutable:

```
fn foo(x: &mut String) {
    // foo is still not responsible for dropping x before returning,
    // nor is it allowed to. however, foo may modify the String.
    let x2 = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // mutating OK
    drop(*x); // ERROR, cannot drop value when not owned
    println!("{}", x.len()); // reading OK
}
```

Si vous ne spécifiez ni `&` ni `&mut` , vous dites que la fonction prendra possession d'un argument. Cela signifie que `foo` est maintenant aussi responsable de laisser tomber `x` .

```
fn foo(x: String) {
    // foo may do whatever it wishes with x, since no-one else has
    // access to it. once the function terminates, x will be dropped,
    // unless it is moved away when calling another function.
    let mut x2 = x; // moving OK
    x2.push_str("foo"); // mutating OK
    let _ = &mut x2; // mutable borrow OK
    let _ = &x2; // immutable borrow OK (note that &mut above is dropped)
    println!("{}", x2.len()); // reading OK
    drop(x2); // dropping OK
}
```

Propriété et copie

Certains types de rouille implémentent le trait de `Copy` . Les types qui sont `Copy` peuvent être déplacés sans posséder la valeur en question. En effet, le contenu de la valeur peut simplement être copié octet par octet dans la mémoire pour produire une nouvelle valeur identique. La plupart des primitives de Rust (`bool` , `usize` , `f64` , etc.) sont des `Copy` .

```
let x: isize = 42;
let xr = &x;
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

Notamment, `Vec` et `String` *ne* sont *pas des* `Copy` :

```
let x = Vec::new();
let xr = &x;
let y = *xr; // ERROR, cannot move out of borrowed content
```

Lire La possession en ligne: https://riptutorial.com/fr/rust/topic/4395/la-possession

Chapitre 27: Les itérateurs

Introduction

Les itérateurs sont une caractéristique de langage puissante de Rust, décrite par le trait `Iterator` . Les itérateurs vous permettent d'effectuer de nombreuses opérations sur des types de type collection, par exemple `Vec<T>` , et sont facilement composables.

Examples

Adaptateurs et consommateurs

Les méthodes d'itérateur peuvent être divisées en deux groupes distincts:

Adaptateurs

Les adaptateurs prennent un itérateur et renvoient un autre itérateur

```
//          Iterator  Adapter
//              |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

Sortie

```
Map { iter: 1..6 }
```

Notez que les valeurs n'ont pas été énumérées, ce qui indique que les itérateurs ne sont pas évalués avec impatience - les itérateurs sont "paresseux".

Les consommateurs

Les consommateurs prennent un itérateur et retournent autre chose qu'un itérateur, consommant l'itérateur dans le processus.

```
//                    Iterator  Adapter       Consumer
//                        |       |              |
let my_squares: Vec<_> = (1..6).map(|x| x * x).collect();
println!("{:?}", my_squares);
```

Sortie

```
[1, 4, 9, 16, 25]
```

Parmi les autres exemples de consommateurs, citons `find` , `fold` et `sum` .

```
let my_squared_sum: u32 = (1..6).map(|x| x * x).sum();
println!("{:?}", my_squared_sum);
```

Sortie

```
55
```

Un test de primalité court

```
fn is_prime(n: u64) -> bool {
    (2..n).all(|divisor| n % divisor != 0)
}
```

Bien sûr, ce n'est pas un test rapide. Nous pouvons arrêter de tester à la racine carrée de `n` :

```
(2..n)
    .take_while(|divisor| divisor * divisor <= n)
    .all(|divisor| n % divisor != 0)
```

Itérateur personnalisé

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

Exemple d'utilisation:

```
// the iterator method `take()` is an adapter which limits the number of items
// generated by the original iterator
for i in Fibonacci(0, 1).take(10) {
    println!("{}", i);
}
```

Lire Les itérateurs en ligne: https://riptutorial.com/fr/rust/topic/4657/les-iterateurs

Chapitre 28: Les structures

Syntaxe

- struct Foo {field1: Type1, field2: Type2}
- let foo = Foo {field1: Type1 :: new (), field2: Type2 :: new ()};
- struct Bar (Type1, Type2); // type de tuple
- let _ = Bar (Type1 :: new (), Type2 :: new ());
- struct Baz; // type d'unité
- let _ = Baz;
- let Foo {field1, ..} = foo; // extrait le champ1 par correspondance de modèle
- let Foo {field1: x, ..} = foo; // extraire field1 comme x
- laisser foo2 = Foo {field1: Type1 :: new (), .. foo}; // construit à partir de l'existant
- implémenter Foo {fn fiddle (& self) {}} // déclarer la méthode d'instance pour Foo
- impliquer Foo {fn tweak (& mut self) {}} // déclarer la méthode d'instance mutable pour Foo
- impliquer Foo {fn double (self) {}} // déclare la méthode d'instance propriétaire pour Foo
- impliquer Foo {fn new () {}} // déclare la méthode associée pour Foo

Examples

Définir des structures

Les structures de Rust sont définies à l'aide du mot `struct` clé `struct` . La forme de structure la plus courante consiste en un ensemble de champs nommés:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

Le ci-dessus déclare une `struct` avec trois champs: `my_bool` , `my_num` et `my_string` , respectivement des types `bool` , `isize` et `String` .

Une autre façon de créer des `struct` dans Rust consiste à créer une *structure de tuple* :

```
struct Bar (bool, isize, String);
```

Ceci définit un nouveau type, `Bar` , qui comporte trois champs sans nom, de type `bool` , `isize` et `String` , dans cet ordre. Ceci est connu comme le modèle *newtype* , car il introduit effectivement un nouveau "nom" pour un type particulier. Cependant, il le fait d'une manière plus puissante que les alias créés à l'aide du mot-clé `type` ; `Bar` est ici un type entièrement fonctionnel, ce qui signifie que vous pouvez écrire vos propres méthodes (ci-dessous).

Enfin, déclarez une `struct` *sans* champs, appelée *structure unitaire* :

```
struct Baz;
```

Cela peut être utile pour se moquer ou tester (lorsque vous souhaitez implémenter un trait de manière triviale) ou comme type de marqueur. En général, cependant, il est peu probable que vous rencontriez de nombreuses structures semblables à des unités.

Notez que les champs de `struct` dans Rust sont tous privés par défaut - c'est-à-dire qu'ils ne sont pas accessibles depuis le code en dehors du module qui définit le type. Vous pouvez préfixer un champ avec le mot-clé `pub` pour rendre ce champ accessible au public. De plus, le type de `struct` *lui - même* est privé. Pour que le type soit disponible pour les autres modules, la définition de la `struct` doit également être précédée de `pub` :

```
pub struct X {
    my_field: bool,
    pub our_field: bool,
}
```

Créer et utiliser des valeurs de structure

Considérons les définitions de `struct` suivantes:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

La construction de nouvelles valeurs de structure pour ces types est simple:

```
let foo = Foo { my_bool: true, my_num: 42, my_string: String::from("hello") };
let bar = Bar(true, 42, String::from("hello"));
let baz = Baz;
```

Champs d'accès d'une structure utilisant `.` :

```
assert_eq!(foo.my_bool, true);
assert_eq!(bar.0, true); // tuple structs act like tuples
```

Une liaison mutable à une structure peut avoir ses champs mutés:

```
let mut foo = foo;
foo.my_bool = false;
let mut bar = bar;
bar.0 = false;
```

Les fonctionnalités de corrélation de Rust peuvent également être utilisées pour visualiser une `struct` :

```
// creates bindings mb, mn, ms with values of corresponding fields in foo
let Foo { my_bool: mb, my_num: mn, my_string: ms } = foo;
assert_eq!(mn, 42);
// .. allows you to skip fields you do not care about
let Foo { my_num: mn, .. } = foo;
assert_eq!(mn, 42);
// leave out `: variable` to bind a variable by its field name
let Foo { my_num, .. } = foo;
assert_eq!(my_num, 42);
```

Ou créez une structure en utilisant une deuxième structure en tant que "modèle" avec la *syntaxe de mise à jour* de Rust:

```
let foo2 = Foo { my_string: String::from("world"), .. foo };
assert_eq!(foo2.my_num, 42);
```

Méthodes de structure

Pour déclarer des méthodes sur une structure (c'est-à-dire des fonctions pouvant être appelées "sur" la `struct` , ou des valeurs de ce type de `struct`), créez un bloc `impl` :

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

`&self` indique ici une référence immuable à une instance de `struct Foo` est nécessaire pour invoquer la méthode `fiddle` . Si nous voulions modifier l'instance (comme changer l'un de ses champs), nous prendrions plutôt un `&mut self` (c'est-à-dire une référence mutable):

```
impl Foo {
    fn tweak(&mut self, n: isize) {
        self.my_num = n;
    }
}

// ...
foo.tweak(43);
assert_eq!(foo.my_num, 43);
```

Enfin, nous pourrions également utiliser `self` (notez l'absence d'un `&`) comme récepteur. Cela nécessite que l'instance appartienne à l'appelant et que l'instance soit déplacée lors de l'appel de la méthode. Cela peut être utile si vous souhaitez consommer, détruire ou transformer complètement une instance existante. Un exemple d'un tel cas d'utilisation est de fournir des méthodes de "chaînage":

```
impl Foo {
```

```
    fn double(mut self) -> Self {
        self.my_num *= 2;
        self
    }
}

// ...
foo.my_num = 1;
assert_eq!(foo.double().double().my_num, 4);
```

Notez que nous avons également préfixés `self` avec `mut` afin que nous puissions muter soi avant de le retourner à nouveau. Le type de retour de la méthode `double` mérite également une explication. `Self` à `Self` intérieur d'un bloc d' `impl` fait référence au type auquel l' `impl` s'applique (dans ce cas, `Foo`). Ici, c'est surtout un raccourci utile pour éviter de retaper la signature du type, mais dans les traits, il peut être utilisé pour faire référence au type sous-jacent qui implémente un trait particulier.

Déclarer une *méthode associée* (communément appelée "méthode de classe" dans d'autres langages) pour une `struct` tout simplement `self` argument de `self` . De telles méthodes sont appelées sur le type de `struct` lui-même, et non sur une instance de celui-ci:

```
impl Foo {
    fn new(b: bool, n: isize, s: String) -> Foo {
        Foo { my_bool: b, my_num: n, my_string: s }
    }
}

// ...
// :: is used to access associated members of the type
let x = Foo::new(false, 0, String::from("nil"));
assert_eq!(x.my_num, 0);
```

Notez que les méthodes de structure ne peuvent être définies que pour les types déclarés dans le module actuel. De plus, comme pour les champs, toutes les méthodes de structure sont privées par défaut et ne peuvent donc être appelées que par code dans le même module. Vous pouvez préfixer les définitions avec le mot-clé `pub` pour les rendre appelables ailleurs.

Structures génériques

Les structures peuvent être génériques sur un ou plusieurs paramètres de type. Ces types sont indiqués entre `<>` en référence au type:

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

Plusieurs types peuvent être donnés en utilisant des virgules:

```
struct Gen2<T, U> {
    x: T,
    y: U,
}

// ...
let _: Gen2<bool, isize> = Gen2{x: true, y: 42};
```

Les paramètres de type font partie du type, donc deux variables du même type de base, mais avec des paramètres différents, ne sont pas interchangeables:

```
let mut a: Gen<bool> = Gen{x: true, z: 1};
let b: Gen<isize> = Gen{x: 42, z: 2};
a = b; // this will not work, types are not the same
a.x = 42; // this will not work, the type of .x in a is bool
```

Si vous voulez écrire une fonction qui accepte une `struct` indépendamment de son affectation de paramètre de type, cette fonction devra également être générique:

```
fn hello<T>(g: Gen<T>) {
    println!("{}", g.z); // valid, since g.z is always an isize
}
```

Mais si nous voulions écrire une fonction qui pourrait toujours imprimer `gx` ? Il faudrait restreindre `T` à un type pouvant être affiché. Nous pouvons le faire avec des limites de type:

```
use std::fmt;
fn hello<T: fmt::Display>(g: Gen<T>) {
    println!("{} {}", g.x, g.z);
}
```

La fonction `hello` est désormais définie *uniquement* pour les instances `Gen` dont le type `T` implémente `fmt::Display` . Si on essayait de passer un `Gen<(bool, isize)>` par exemple, le compilateur se plaindrait que `hello` n'est pas défini pour ce type.

Nous pouvons également utiliser des limites de type directement sur les paramètres de type de la `struct` pour indiquer que vous ne pouvez construire cette `struct` que pour certains types:

```
use std::hash::Hash;
struct GenB<T: Hash> {
    x: T,
}
```

Toute fonction ayant accès à un `GenB` sait maintenant que le type de `x` implémente `Hash` , et donc qu'elle peut appeler `.x.hash()` . Des limites de type multiples pour le même paramètre peuvent être données en les séparant par un `+` .

Comme pour les fonctions, les limites de type peuvent être placées après le `<>` utilisant le mot-clé `where` :

```
struct GenB<T> where T: Hash {
```

```
    x: T,
}
```

Cela a la même signification sémantique, mais peut rendre la signature plus facile à lire et à formater lorsque vous avez des limites complexes.

Les paramètres de type sont également disponibles pour les méthodes d'instance et les méthodes associées de la `struct` :

```
// note the <T> parameter for the impl as well
// this is necessary to say that all the following methods only
// exist within the context of those type parameter assignments
impl<T> Gen<T> {
    fn inner(self) -> T {
        self.x
    }
    fn new(x: T) -> Gen<T> {
        Gen{x: x}
    }
}
```

Si vous avez des limites de type sur le `T Gen` , celles-ci devraient également être reflétées dans les limites de type de l' `impl` . Vous pouvez également rendre les limites d' `impl` plus strictes pour indiquer qu'une méthode donnée n'existe que si le type satisfait une propriété particulière:

```
impl<T: Hash + fmt::Display> Gen<T> {
    fn show(&self) {
        println!("{}", self.x);
    }
}

// ...
Gen{x: 42}.show(); // works fine
let a = Gen{x: (42, true)}; // ok, because (isize, bool): Hash
a.show(); // error: (isize, bool) does not implement fmt::Display
```

Lire Les structures en ligne: https://riptutorial.com/fr/rust/topic/4583/les-structures

Chapitre 29: Macros

Remarques

Une revue des macros peut être trouvée dans *le langage de programmation Rust* (aka The Book)
.

Examples

Didacticiel

Les macros nous permettent d'abstraire des schémas syntaxiques répétés plusieurs fois. Par exemple:

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

Nous remarquons que les deux déclarations de `match` sont très similaires: les deux ont le même motif

```
match expression {
    Some(x) => x,
    None => return None,
}
```

Imaginez que nous représentions le modèle ci-dessus en tant que `try_opt!(expression)` , nous pourrions alors réécrire la fonction en 3 lignes seulement:

```
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = try_opt!(b.checked_mul(c));
    let sum = try_opt!(a.checked_add(product));
    Some(sum)
}
```

`try_opt!` ne peut pas écrire une fonction car une fonction ne prend pas en charge le retour anticipé. Mais nous pourrions le faire avec une macro - chaque fois que nous avons ces modèles syntaxiques qui ne peuvent pas être représentés en utilisant une fonction, nous pouvons essayer d'utiliser une macro.

Nous définissons une macro en utilisant la `macro_rules!` syntaxe:

```
macro_rules! try_opt {
//                  ^ note: no `!` after the macro name
    ($e:expr) => {
//   ^~~~~~~ The macro accepts an "expression" argument, which we call `$e`.
//           All macro parameters must be named like `$xxxxx`, to distinguish from
//           normal tokens.
        match $e {
//            ^~ The input is used here.
            Some(x) => x,
            None => return None,
        }
    }
}
```

C'est tout! Nous avons créé notre première macro.

(Essayez-le dans Rust Playground)

Créer une macro HashSet

```
// This example creates a macro `set!` that functions similarly to the built-in
// macro vec!

use std::collections::HashSet;

macro_rules! set {
    ( $( $x:expr ),* ) => {  // Match zero or more comma delimited items
        {
            let mut temp_set = HashSet::new();  // Create a mutable HashSet
            $(
                temp_set.insert($x); // Insert each item matched into the HashSet
            )*
            temp_set // Return the populated HashSet
        }
    };
}

// Usage
let my_set = set![1, 2, 3, 4];
```

Récursivité

Une macro peut s'appeler, comme une récursion de fonction:

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

Allons-y l'expansion de la `sum!(1, 2, 3)` :

```
   sum!(1, 2, 3)
//      ^  ^~~~
```

```
//     $a  $rest
=> 1 + sum!(2, 3)
//           ^  ^
//         $a  $rest
=> 1 + (2 + sum!(3))
//               ^
//               $base
=> 1 + (2 + (3))
```

Limite de récursivité

Lorsque le compilateur étend trop les macros, il va abandonner. Par défaut, le compilateur échouera après avoir étendu les macros à 64 niveaux, de sorte que l'extension suivante provoquera une défaillance:

```
sum!(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62)

// error: recursion limit reached while expanding the macro `sum`
//  --> <anon>:3:46
// 3 |>         ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
//   |>                                              ^^^^^^^^^^^^^^^^
```

Lorsqu'une limite de récursivité est atteinte, vous devriez envisager de refactoriser votre macro, par exemple

- Peut-être la récursivité pourrait-elle être remplacée par la répétition?
- Peut-être que le format d'entrée pourrait être changé en quelque chose de moins sophistiqué, donc nous n'avons pas besoin de récursivité pour le faire correspondre?

S'il y a une raison légitime, 64 niveaux ne suffisent pas, vous pouvez toujours augmenter la limite de la caisse en appelant la macro avec l'attribut:

```
#![recursion_limit="128"]
//                  ^~~ set the recursion limit to 128 levels deep.
```

Plusieurs modèles

Une macro peut produire différentes sorties par rapport à différents modèles d'entrée:

```
/// The `sum` macro may be invoked in two ways:
///
///     sum!(iterator)
///     sum!(1234, iterator)
///
macro_rules! sum {
    ($iter:expr) => {    // This branch handles the `sum!(iterator)` case
        $iter.fold(0, |a, b| a + *b)
    };
//   ^ use `;` to separate each branch
```

```
    ($start:expr, $iter:expr) => {  // This branch handles the `sum!(1234, iter)` case
        $iter.fold($start, |a, b| a + *b)
    };
}

fn main() {
    assert_eq!(10, sum!([1, 2, 3, 4].iter()));
    assert_eq!(23, sum!(6, [2, 5, 9, 1].iter()));
}
```

Spécificateurs de fragment - Type de modèle

Dans `$e:expr` , l' `expr` est appelé le *spécificateur de fragment* . Il indique à l'analyseur quel type de jeton le paramètre `$e` attend. Rust fournit une variété de spécificateurs de fragment, permettant à l'entrée d'être très flexible.

Spécificateur	La description	Exemples
`ident`	Identifiant	`x` , `foo`
`path`	Nom qualifié	`std::collection::HashSet` , `Vec::new`
`ty`	Type	`i32` , `&T` , `Vec<(char, String)>`
`expr`	Expression	`2+2` , `f(42)` , `if true { 1 } else { 2 }`
`pat`	Modèle	`_` , `c @ 'a' ... 'z'` , `(true, &x)` , `Badger { age, .. }`
`stmt`	Déclaration	`let x = 3` , `return 42`
`block`	Bloc délimité par des accolades	`{ foo(); bar(); }` , `{ x(); y(); z() }`
`item`	Article	`fn foo() {}` , `struct Bar;` , `use std::io;`
`meta`	Intérieur de l'attribut	`cfg!(windows)` , `doc="comment"`
`tt`	Arbre jeton	`+` , `foo` , `5` , `[?!(???)]`

Notez qu'un commentaire de commentaire `/// comment` est traité de la même manière que `#[doc="comment"]` dans une macro.

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}
```

```
declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
}

// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

Suivre ensemble

Certains spécificateurs de fragment nécessitent que le jeton suivant soit un ensemble restreint, appelé "ensemble de suivi". Cela permet une certaine souplesse pour que la syntaxe de Rust évolue sans casser les macros existantes.

Spécificateur	Suivre ensemble
`expr` , `stmt`	`=>` , `;`
`ty` , `path`	`=>` , `=` | `;` `:` `>` `[` `{` `as` `where`
`pat`	`=>` , `=` | `if` `in`
`ident` , `block` , `item` , `meta` , `tt`	*n'importe quel jeton*

```
macro_rules! invalid_macro {
    ($e:expr + $f:expr) => { $e + $f };
//           ^
//           `+` is not in the follow set of `expr`,
//           and thus the compiler will not accept this macro definition.
    ($($e:expr)/+) => { $($e)/+ };
//              ^
//              The separator `/` is not in the follow set of `expr`
//              and thus the compiler will not accept this macro definition.
}
```

Exportation et importation de macros

Exportation d'une macro pour permettre à d'autres modules de l'utiliser:

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

Utiliser des macros d'autres caisses ou modules:

```
   #[macro_use] extern crate lazy_static;
// ^~~~~~~~~~~~ Must add this in order to use macros from other crates
```

```
   #[macro_use] mod macros;
// ^~~~~~~~~~~~ The same for child modules.
```

Débogage des macros

(Tous ces éléments sont instables et ne peuvent donc être utilisés que par un compilateur nocturne.)

log_syntax! ()

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

Lors de la compilation, il affichera ce qui suit dans stdout:

a = 1, reste = 2, 3
a = 2, reste = 3
base = 3

--pretty élargi

Exécutez le compilateur avec:

```
rustc -Z unstable-options --pretty expanded filename.rs
```

Cela va développer toutes les macros et ensuite imprimer le résultat étendu à stdout, par exemple, le résultat ci-dessus va probablement sortir:

```
#![feature(prelude_import)]
#![no_std]
#![feature(log_syntax)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;


const V: u32 = { false; 1 + { false; 2 + { false; 3 } } };
```

(Ceci est similaire à la `-E` drapeau dans les compilateurs C `gcc` et `clang`).

Lire Macros en ligne: https://riptutorial.com/fr/rust/topic/1031/macros

Chapitre 30: Mise en réseau TCP

Examples

Une application client et serveur TCP simple: echo

Le code suivant est basé sur les exemples fournis par la documentation sur std :: net :: TcpListener . Cette application serveur écoutera les demandes entrantes et renverra toutes les données entrantes, agissant ainsi comme un serveur "écho". L'application client envoie un petit message et attend une réponse avec le même contenu.

serveur:

```
use std::thread;
use std::net::{TcpListener, TcpStream, Shutdown};
use std::io::{Read, Write};

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
    drop(listener);
```

```
}
```

client:

```
use std::net::{TcpStream};
use std::io::{Read, Write};
use std::str::from_utf8;

fn main() {
    match TcpStream::connect("localhost:3333") {
        Ok(mut stream) => {
            println!("Successfully connected to server in port 3333");

            let msg = b"Hello!";

            stream.write(msg).unwrap();
            println!("Sent Hello, awaiting reply...");

            let mut data = [0 as u8; 6]; // using 6 byte buffer
            match stream.read_exact(&mut data) {
                Ok(_) => {
                    if &data == msg {
                        println!("Reply is ok!");
                    } else {
                        let text = from_utf8(&data).unwrap();
                        println!("Unexpected reply: {}", text);
                    }
                },
                Err(e) => {
                    println!("Failed to receive data: {}", e);
                }
            }
        },
        Err(e) => {
            println!("Failed to connect: {}", e);
        }
    }
    println!("Terminated.");
}
```

Lire Mise en réseau TCP en ligne: https://riptutorial.com/fr/rust/topic/1350/mise-en-reseau-tcp

Chapitre 31: Modules

Syntaxe

- mod *modname* ; // Recherche le module dans *modname* .rs ou *modname* /mod.rs dans le même répertoire
- mod *modname* { *block* }

Examples

Arbre des modules

Des dossiers:

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

Modules:

```
- example         -> example
  - first         -> example::first
  - second        -> example::second
    - sub         -> example::second::sub
  - third         -> example::third
```

`example.rs`

```
pub mod first;
pub mod second;
pub mod third {
    ...
}
```

Le module `second` doit être déclaré dans le fichier `example.rs` comme son parent est un `example` et non, par exemple, d' `first` et ne peut donc pas être déclaré dans le `first.rs` fichier ou un autre fichier au même niveau de répertoire

`second/mod.rs`

```
pub mod sub;
```

L'attribut # [chemin]

L'attribut `#[path]` Rust peut être utilisé pour spécifier le chemin pour rechercher un module particulier s'il ne se trouve pas dans l'emplacement standard. Cela est généralement déconseillé ,

car cela rend la hiérarchie des modules fragile et facilite la décomposition de la construction en déplaçant un fichier dans un répertoire complètement différent.

```
#[path="../path/to/module.rs"]
mod module;
```

Noms dans le code vs noms dans `use`

La syntaxe à deux points des noms dans l'instruction `use` ressemble aux noms utilisés ailleurs dans le code, mais la signification de ces chemins est différente.

Les noms dans l'instruction `use` par défaut sont interprétés comme absolus, en commençant à la racine de la caisse. Les noms ailleurs dans le code sont relatifs au module actuel.

La déclaration:

```
use std::fs::File;
```

a la même signification dans le fichier principal de la caisse ainsi que dans les modules. D'un autre côté, un nom de fonction tel que `std::fs::File::open()` fera référence à la bibliothèque standard de Rust uniquement dans le fichier principal de la caisse, car les noms du code sont interprétés par rapport au module actuel.

```
fn main() {
    std::fs::File::open("example"); // OK
}

mod my_module {
   fn my_fn() {
       // Error! It means my_module::std::fs::File::open()
       std::fs::File::open("example");

       // OK. `::` prefix makes it absolute
       ::std::fs::File::open("example");

       // OK. `super::` reaches out to the parent module, where `std` is present
       super::std::fs::File::open("example");
   }
}
```

Pour que les noms `std::…` se comportent partout comme dans la racine du coffre, vous pouvez ajouter:

```
use std;
```

Inversement, vous pouvez `use` chemins relatifs en les préfixant avec `self` mots `super` clés `self` ou `super` :

```
 use self::my_module::my_fn;
```

Accéder au module parent

Parfois, il peut être utile d'importer relativement des fonctions et des structures sans avoir à `use` quelque chose avec son chemin absolu dans votre projet. Pour ce faire, vous pouvez utiliser le module `super` , comme ceci:

```
fn x() -> u8 {
    5
}

mod example {
    use super::x;

    fn foo() {
        println!("{}", x());
    }
}
```

Vous pouvez utiliser plusieurs fois `super` pour atteindre le «grand-parent» de votre module actuel, mais vous devriez vous méfier des problèmes de lisibilité si vous utilisez `super` trop de fois dans une importation.

Exportations et Visibilité

Structure du répertoire:

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

main.rs

```
// This is import from writer.rs
mod writer;

fn main() {
    // Call of imported write() function.
    writer::write()

    // BAD
    writer::open_file()
}
```

writer.rs

```
// This function WILL be exported.
pub fn write() {}

// This will NOT be exported.
fn open_file() {}
```

Organisation du code de base

Voyons comment nous pouvons organiser le code lorsque le code devient plus grand.

01. Fonctions

```
fn main() {
  greet();
}

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

02. Modules - Dans le même fichier

```
fn main() {
  greet::hello();
}

mod greet {
  // By default, everything inside a module is private
  pub fn hello() { // So function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

03. Modules - Dans un fichier différent du même répertoire

Lorsque vous déplacez du code dans un nouveau fichier, vous n'avez pas besoin d'encapsuler le code dans une déclaration de `mod` . Le fichier lui-même agit comme un module.

```
// ↳ main.rs
mod greet; // import greet module

fn main() {
  greet::hello();
}
```

```
// ↳ greet.rs
pub fn hello() { // function has to be public to access from outside
  println!("Hello, world!");
}
```

> Lorsque vous déplacez du code dans un nouveau fichier, si ce code a été encapsulé dans une déclaration de `mod` , ce sera un sous-module du fichier.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello::greet();
}
```

```
// ↳ greet.rs
pub mod hello { // module has to be public to access from outside
  pub fn greet() { // function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

04. Modules - Dans un fichier différent dans un répertoire différent

Lorsque vous déplacez du code dans un nouveau fichier dans un répertoire différent, le répertoire lui-même agit comme un module. Et `mod.rs` dans la racine du module est le point d'entrée du module de répertoire. Tous les autres fichiers de ce répertoire agissent comme un sous-module de ce répertoire.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello();
}
```

```
// ↳ greet/mod.rs
pub fn hello() {
  println!("Hello, world!");
}
```

Lorsque vous avez plusieurs fichiers dans la racine du module,

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello_greet()
}
```

```
// ↳ greet/mod.rs
mod hello;

pub fn hello_greet() {
  hello::greet()
}
```

```
// ↳ greet/hello.rs
pub fn greet() {
  println!("Hello, world!");
}
```

05. Modules - Avec `self`

```
fn main() {
  greet::call_hello();
}

mod greet {
```

```
  pub fn call_hello() {
    self::hello();
  }

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

06. Modules - Avec `super`

1. Lorsque vous souhaitez accéder à une fonction racine depuis un module,

```
fn main() {
  dash::call_hello();
}

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

mod dash {
  pub fn call_hello() {
    super::hello();
  }
}
```

2. Lorsque vous souhaitez accéder à une fonction dans le module externe / parent depuis un module imbriqué,

```
fn main() {
  outer::inner::call_hello();
}

mod outer {

  pub fn hello() {
    println!("Hello, world!");
  }

  mod inner {
    pub fn call_hello() {
      super::hello();
    }
  }

}
```

07. Modules - À l' `use`

1. Lorsque vous souhaitez lier le chemin d'accès complet à un nouveau nom,

```
use greet::hello::greet as greet_hello;

fn main() {
  greet_hello();
```

```
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}
```

2. Lorsque vous souhaitez utiliser le contenu de niveau de portée de la caisse

```
fn main() {
  user::hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}

mod user {
  use greet::hello::greet as call_hello;

  pub fn hello() {
    call_hello();
  }
}
```

Lire Modules en ligne: https://riptutorial.com/fr/rust/topic/2528/modules

Chapitre 32: Opérateurs et surcharge

Introduction

La plupart des opérateurs de Rust peuvent être définis ("surchargés") pour les types définis par l'utilisateur. Cela peut être réalisé en implémentant le trait respectif dans le module `std::ops` .

Examples

Surcharger l'opérateur d'addition (+)

La surcharge de l'opérateur d'addition (+) nécessite l'implémentation du trait `std::ops::Add` .

À partir de la documentation, la définition complète du trait est la suivante:

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Comment ça marche?

- le trait est implémenté pour le type côté gauche
- le trait est implémenté pour *un* argument du côté droit, sauf s'il est spécifié qu'il a le même type que le côté gauche
- le type du résultat de l'addition est spécifié dans le type associé de `Output`

Ainsi, avoir 3 types différents est possible.

Remarque: le trait consomme des arguments de gauche et de droite, vous préférerez peut-être l'implémenter pour des références à votre type plutôt qu'aux types dénudés.

Implémenter `+` pour un type personnalisé:

```
use std::ops::Add;

#[derive(Clone)]
struct List<T> {
    data: Vec<T>,
}
//  Implementation which consumes both LHS and RHS
impl<T> Add for List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.data.extend(rhs.data.drain(..));
        self
    }
}
```

```
//  Implementation which only consumes RHS (and thus where LHS != RHS)
impl<'a, T: Clone> Add<List<T>> for &'a List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.clone() + rhs
    }
}
```

Lire Opérateurs et surcharge en ligne: https://riptutorial.com/fr/rust/topic/7271/operateurs-et-surcharge

Chapitre 33: Option

Introduction

Le type `Option<T>` est l'équivalent de Rust des types nullables, sans tous les problèmes qui l'accompagnent. La majorité des langages de type C permettent à toute variable d'être `null` s'il n'y a pas de données présentes, mais le type `Option` est inspiré par les langages fonctionnels qui favorisent les «optionnels» (par exemple, la monade Haskell's `Maybe`). L'utilisation des types `Option` vous permettra d'exprimer l'idée que les données peuvent ou non être présentes (puisque Rust n'a pas de types nullables).

Examples

Création d'une valeur d'option et d'une correspondance de modèle

```
// The Option type can either contain Some value or None.
fn find(value: i32, slice: &[i32]) -> Option<usize> {
    for (index, &element) in slice.iter().enumerate() {
        if element == value {
            // Return a value (wrapped in Some).
            return Some(index);
        }
    }
    // Return no value.
    None
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    // Pattern match against the Option value.
    if let Some(index) = find(2, &array) {
        // Here, there is a value.
        println!("The element 2 is at index {}.", index);
    }

    // Check if the result is None (no value).
    if let None = find(12, &array) {
        // Here, there is no value.
        println!("The element 12 is not in the array.");
    }

    // You can also use `is_some` and `is_none` helpers
    if find(12, &array).is_none() {
        println!("The element 12 is not in the array.");
    }
}
```

Destructurer une option

```
fn main() {
    let maybe_cake = Some("Chocolate cake");
```

```
    let not_cake = None;

    // The unwrap method retrieves the value from the Option
    // and panics if the value is None
    println!("{}", maybe_cake.unwrap());

    // The expect method works much like the unwrap method,
    // but panics with a custom, user provided message.
    println!("{}", not_cake.expect("The cake is a lie."));

    // The unwrap_or method can be used to provide a default value in case
    // the value contained within the option is None. This example would
    // print "Cheesecake".
    println!("{}", not_cake.unwrap_or("Cheesecake"));

    // The unwrap_or_else method works like the unwrap_or method,
    // but allows us to provide a function which will return the
    // fallback value. This example would print "Pumpkin Cake".
    println!("{}", not_cake.unwrap_or_else(|| { "Pumpkin Cake" }));

    // A match statement can be used to safely handle the possibility of none.
    match maybe_cake {
        Some(cake) => println!("{} was consumed.", cake),
        None       => println!("There was no cake.")
    }

    // The if let statement can also be used to destructure an Option.
    if let Some(cake) = maybe_cake {
        println!("{} was consumed.", cake);
    }
}
```

Déballer une référence à une option possédant son contenu

Une référence à une option `&Option<T>` ne peut pas être dépliée si le type `T` n'est pas copiable. La solution consiste à modifier l'option `&Option<&T>` utilisant `as_ref()` .

Rust interdit le transfert de propriété des objets pendant que les objets sont empruntés. Lorsque l'option elle-même est empruntée (`&Option<T>`), son contenu est également - indirectement - emprunté.

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

ne peut pas sortir du contenu emprunté [--expliquer E0507]

Cependant, il est possible de créer une référence au contenu de l' `Option<T>` . La méthode `as_ref()` de Option renvoie une option pour `&T` , qui peut être déballée sans transfert de propriété:

```
println!("{:?}", wrapped_ref.as_ref().unwrap());
```

Utiliser Option avec map et and_then

L'opération de `map` est un outil utile pour travailler avec des tableaux et des vecteurs, mais elle peut également être utilisée pour gérer les valeurs des `Option` de manière fonctionnelle.

```
fn main() {

    // We start with an Option value (Option<i32> in this case).
    let some_number = Some(9);

    // Let's do some consecutive calculations with our number.
    // The crucial point here is that we don't have to unwrap
    // the content of our Option type - instead, we're just
    // transforming its content. The result of the whole operation
    // will still be an Option<i32>. If the initial value of
    // 'some_number' was 'None' instead of 9, then the result
    //  would also be 'None'.
    let another_number = some_number
        .map(|n| n - 1) // => Some(8)
        .map(|n| n * n) // => Some(64)
        .and_then(|n| divide(n, 4)); // => Some(16)

    // In the last line above, we're doing a division using a helper
    // function (definition: see bottom).
    // 'and_then' is very similar to 'map', but allows us to pass a
    // function which returns an Option type itself. To ensure that we
    // don't end up with Option<Option<i32>>, 'and_then' flattens the
    // result (in other languages, 'and_then' is also known as 'flatmap').

    println!("{}", to_message(another_number));
    // => "16 is definitely a number!"

    // For the sake of completeness, let's check the result when
    // dividing by zero.
    let final_number = another_number
        .and_then(|n| divide(n, 0)); // => None

    println!("{}", to_message(final_number));
    // => "None!"
}

// Just a helper function for integer division. In case
// the divisor is zero, we'll get 'None' as result.
fn divide(number: i32, divisor: i32) -> Option<i32> {
    if divisor != 0  { Some(number/divisor) } else { None }
}

// Creates a message that tells us whether our
// Option<i32> contains a number or not. There are other
// ways to achieve the same result, but let's just use
// map again!
fn to_message(number: Option<i32>) -> String {
    number
        .map(|n| format!("{} is definitely a number!", n)) // => Some("...")
        .unwrap_or("None!".to_string()) // => "..."
}
```

Lire Option en ligne: https://riptutorial.com/fr/rust/topic/1125/option

Chapitre 34: Panique et Déroulement

Introduction

Lorsque les programmes Rust atteignent un état où une erreur critique est survenue, la `panic!` la macro peut être appelée pour sortir rapidement (souvent comparée, mais subtilement différente, à une exception dans d'autres langues). Une gestion correcte des erreurs devrait impliquer les types de `Result` , bien que cette section ne traite que de la `panic!` et ses concepts.

Remarques

Les paniques ne provoquent pas toujours des fuites de mémoire ou d'autres fuites de ressources. En fait, les paniques conservent généralement les invariants RAII, exécutant les destructeurs (implémentations Drop) des structures à mesure que la pile se déroule. Cependant, s'il y a une seconde panique au cours de ce processus, le programme abandonne simplement; à ce stade, les garanties invariantes RAII sont nulles.

Examples

Essayez de ne pas paniquer

Dans Rust, il existe deux méthodes principales pour indiquer que quelque chose a mal tourné dans un programme: Une fonction renvoyant un `Err(E)` (potentiellement personnalisé `Err(E)` , du type `Result<T, E>` et une `panic!` .

La panique n'est **pas** une alternative pour les exceptions, que l'on trouve couramment dans d'autres langues. Dans Rust, la panique est d'indiquer que quelque chose a vraiment mal tourné et que cela ne peut pas continuer. Voici un exemple de la source de `Vec` pour `push` :

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

Si nous manquons de mémoire, il n'y a pas grand-chose d'autre que Rust puisse faire, donc il va paniquer (comportement par défaut) ou abandonner (ce qui doit être défini avec un indicateur de compilation).

La panique va dérouler la pile, exécuter des destructeurs et s'assurer que la mémoire est nettoyée. Abandonner ne le fait pas, et repose sur le système d'exploitation pour le nettoyer correctement.

Essayez d'exécuter le programme suivant normalement et avec

```
[profile.dev]
panic = "abort"
```

dans votre `Cargo.toml` .

```
// main.rs
struct Foo(i32);
impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping {:?}!", self.0);
    }
}
fn main() {
    let foo = Foo(1);
    panic!("Aaaaaaahhhhh!");
}
```

Lire Panique et Déroulement en ligne: https://riptutorial.com/fr/rust/topic/6895/panique-et-deroulement

Chapitre 35: Parallélisme

Introduction

Le parallélisme est bien supporté par la bibliothèque standard de Rust à travers diverses classes telles que le module `std::thread` , les canaux et les atomiques. Cette section vous guidera à travers l'utilisation de ces types.

Examples

Commencer un nouveau fil

Pour démarrer un nouveau sujet:

```
use std::thread;

fn main() {
    thread::spawn(move || {
        // The main thread will not wait for this thread to finish. That
        // might mean that the next println isn't even executed before the
        // program exits.
        println!("Hello from spawned thread");
    });

    let join_handle = thread::spawn(move || {
        println!("Hello from second spawned thread");
        // To ensure that the program waits for a thread to finish, we must
        // call `join()` on its join handle. It is even possible to send a
        // value to a different thread through the join handle, like the
        // integer 17 in this case:
        17
    });

    println!("Hello from the main thread");

    // The above three printlns can be observed in any order.

    // Block until the second spawned thread has finished.
    match join_handle.join() {
        Ok(x) => println!("Second spawned thread returned {}", x),
        Err(_) => println!("Second spawned thread panicked")
    }
}
```

Communication croisée avec les canaux

Les canaux peuvent être utilisés pour envoyer des données d'un thread à un autre. Vous trouverez ci-dessous un exemple de système producteur-consommateur simple, dans lequel le thread principal produit les valeurs 0, 1, ..., 9 et le thread généré les imprime:

```
use std::thread;
```

```
use std::sync::mpsc::channel;

fn main() {
    // Create a channel with a sending end (tx) and a receiving end (rx).
    let (tx, rx) = channel();

    // Spawn a new thread, and move the receiving end into the thread.
    let join_handle = thread::spawn(move || {
        // Keep receiving in a loop, until tx is dropped!
        while let Ok(n) = rx.recv() { // Note: `recv()` always blocks
            println!("Received {}", n);
        }
    });

    // Note: using `rx` here would be a compile error, as it has been
    // moved into the spawned thread.

    // Send some values to the spawned thread. `unwrap()` crashes only if the
    // receiving end was dropped before it could be buffered.
    for i in 0..10 {
        tx.send(i).unwrap(); // Note: `send()` never blocks
    }

    // Drop `tx` so that `rx.recv()` returns an `Err(_)`.
    drop(tx);

    // Wait for the spawned thread to finish.
    join_handle.join().unwrap();
}
```

Communication croisée avec les types de session

Les types de session permettent au compilateur de connaître le protocole que vous souhaitez utiliser pour communiquer entre les threads, et non le protocole comme HTTP ou FTP, mais le modèle de flux d'informations entre les threads. Ceci est utile car le compilateur va maintenant vous empêcher de violer accidentellement votre protocole et de provoquer des blocages ou des "vivelocks" entre les threads - certains des problèmes les plus difficiles à déboguer et une source majeure de Heisenbugs. Les types de session fonctionnent de manière similaire aux canaux décrits ci-dessus, mais peuvent être plus intimidants à utiliser. Voici une communication simple à deux fils:

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;
```

```
// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

Vous devriez noter que la méthode principale ressemble beaucoup à la méthode principale de communication inter-thread définie ci-dessus, si le serveur a été déplacé vers sa propre fonction. Si vous deviez lancer ceci, vous obtiendriez la sortie:

```
The client just sent the number 42!
The server received some data: 42
```

dans cet ordre.

Pourquoi ne pas avoir à définir les types de client et de serveur? Et pourquoi redéfinir le canal dans le client et le serveur? Ces questions ont la même réponse: le compilateur nous empêchera de briser le protocole! Si le client essayait de recevoir des données au lieu de les envoyer (ce qui entraînerait un blocage dans le code ordinaire), le programme ne compilerait pas, car l'objet channel du client *n'a pas de méthode* `recv` *.* En outre, si nous essayions de définir le protocole de manière à provoquer un blocage (par exemple, si le client et le serveur essayaient de recevoir une valeur), la compilation échouerait lorsque nous créerions les canaux. En effet, `Send` et `Recv` sont des "Dual Types", ce qui signifie que si le serveur en fait un, le client doit faire l'autre - si les deux essaient de `Recv` , vous allez avoir des problèmes. `Eps` est son propre type, car le client et le serveur acceptent de fermer le canal.

Bien sûr, lorsque nous effectuons des opérations sur le canal, nous passons à un nouvel état

dans le protocole, et les fonctions disponibles peuvent changer - nous devons donc redéfinir la liaison du canal. Heureusement, `session_types` s'en charge pour nous et renvoie toujours le nouveau canal (sauf `close` , auquel cas il n'y a pas de nouveau canal). Cela signifie également que toutes les méthodes sur un canal prennent également en charge le canal - donc si vous oubliez de redéfinir le canal, le compilateur vous donnera également une erreur à ce sujet. Si vous laissez tomber un canal sans le fermer, c'est une erreur d'exécution (malheureusement, il est impossible de vérifier au moment de la compilation).

Il y a beaucoup plus de types de communication que `Send` et `Recv` - par exemple, `Offer` offre à l'autre côté du canal la possibilité de choisir entre deux branches possibles du protocole, et `Rec` et `Var` fonctionnent ensemble pour autoriser les boucles et la récursivité dans le protocole. . De nombreux autres exemples de types de session et d'autres types sont disponibles dans le référentiel GitHub `session_types` . La documentation de la bibliothèque peut être trouvée ici.

Commande atomique et mémoire

Les types atomiques sont les blocs de construction des structures de données sans verrouillage et des autres types concurrents. Un ordre de mémoire, représentant la force de la barrière de mémoire, devrait être spécifié lors de l'accès / modification d'un type atomique. Rust fournit 5 primitives d'ordonnancement de mémoire: **assouplies** (les plus faibles), **Acquire** (pour les lectures des charges alias), **Release** (pour les écritures alias les magasins), **AcqRel** (équivalent à «Acquire-for-load et Release-for-store») sont impliqués dans une seule opération telle que compare-and-swap), et **SeqCst** (le plus fort). Dans l'exemple ci-dessous, nous montrerons en quoi l'ordre "Détendu" diffère des ordres "Acquérir" et "Libérer".

```
use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::{Arc, Barrier};
use std::thread;

struct UsizePair {
    atom: AtomicUsize,
    norm: UnsafeCell<usize>,
}

// UnsafeCell is not thread-safe. So manually mark our UsizePair to be Sync.
// (Effectively telling the compiler "I'll take care of it!")
unsafe impl Sync for UsizePair {}

static NTHREADS: usize = 8;
static NITERS: usize = 1000000;

fn main() {
    let upair = Arc::new(UsizePair::new(0));

    // Barrier is a counter-like synchronization structure (not to be confused
    // with a memory barrier). It blocks on a `wait` call until a fixed number
    // of `wait` calls are made from various threads (like waiting for all
    // players to get to the starting line before firing the starter pistol).
    let barrier = Arc::new(Barrier::new(NTHREADS + 1));

    let mut children = vec![];
```

```
    for _ in 0..NTHREADS {
        let upair = upair.clone();
        let barrier = barrier.clone();
        children.push(thread::spawn(move || {
            barrier.wait();

            let mut v = 0;
            while v < NITERS - 1 {
                // Read both members `atom` and `norm`, and check whether `atom`
                // contains a newer value than `norm`. See `UsizePair` impl for
                // details.
                let (atom, norm) = upair.get();
                if atom > norm {
                    // If `Acquire`-`Release` ordering is used in `get` and
                    // `set`, then this statement will never be reached.
                    println!("Reordered! {} > {}", atom, norm);
                }
                v = atom;
            }
        }));
    }

    barrier.wait();

    for v in 1..NITERS {
        // Update both members `atom` and `norm` to value `v`. See the impl for
        // details.
        upair.set(v);
    }

    for child in children {
        let _ = child.join();
    }
}

impl UsizePair {
    pub fn new(v: usize) -> UsizePair {
        UsizePair {
            atom: AtomicUsize::new(v),
            norm: UnsafeCell::new(v),
        }
    }

    pub fn get(&self) -> (usize, usize) {
        let atom = self.atom.load(Ordering::Relaxed); //Ordering::Acquire

        // If the above load operation is performed with `Acquire` ordering,
        // then all writes before the corresponding `Release` store is
        // guaranteed to be visible below.

        let norm = unsafe { *self.norm.get() };
        (atom, norm)
    }

    pub fn set(&self, v: usize) {
        unsafe { *self.norm.get() = v };

        // If the below store operation is performed with `Release` ordering,
        // then the write to `norm` above is guaranteed to be visible to all
        // threads that "loads `atom` with `Acquire` ordering and sees the same
        // value that was stored below". However, no guarantees are provided as
```

```
        // to when other readers will witness the below store, and consequently
        // the above write. On the other hand, there is also no guarantee that
        // these two values will be in sync for readers. Even if another thread
        // sees the same value that was stored below, it may actually see a
        // "later" value in `norm` than what was written above. That is, there
        // is no restriction on visibility into the future.

        self.atom.store(v, Ordering::Relaxed); //Ordering::Release
    }
}
```

Remarque: les architectures x86 ont un modèle de mémoire robuste. Cet article l' explique en détail. Jetez également un coup d'oeil à la page Wikipedia pour la comparaison des architectures.

Verrous de lecture-écriture

RwLocks permet à un seul producteur de fournir à un nombre quelconque de lecteurs des données tout en empêchant les lecteurs de voir des données invalides ou incohérentes.

L'exemple suivant utilise RwLock pour montrer comment un thread producteur unique peut augmenter périodiquement une valeur tandis que deux threads consommateurs lisent la valeur.

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };

fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
```

```
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }

            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

Exemple de sortie:

```
Updated value: 1
Read value: 1
Read value squared: 1
Read value: 1
Read value squared: 1
Updated value: 2
Read value: 2
Read value: 2
Read value squared: 4
Updated value: 3
Read value: 3
Read value squared: 9
Read value: 3
Updated value: 4
Read value: 4
Read value squared: 16
Read value: 4
Read value squared: 16
Updated value: 5
```

```
Read value: 5
Read value: 5
Read value squared: 25
...(Interrupted)...
```

Lire Parallélisme en ligne: https://riptutorial.com/fr/rust/topic/1222/parallelisme

Chapitre 36: PhantomData

Examples

Utilisation de PhantomData comme marqueur de type

L'utilisation du type `PhantomData` comme ceci vous permet d'utiliser un type spécifique sans avoir besoin de faire partie de Struct.

```
use std::marker::PhantomData;

struct Authenticator<T: GetInstance> {
    _marker: PhantomData<*const T>, // Using `*const T` indicates that we do not own a T
}

impl<T: GetInstance> Authenticator<T> {
    fn new() -> Authenticator<T> {
        Authenticator {
            _marker: PhantomData,
        }
    }

    fn auth(&self, id: i64) -> bool {
        T::get_instance(id).is_some()
    }
}

trait GetInstance {
    type Output; // Using nightly this could be defaulted to `Self`
    fn get_instance(id: i64) -> Option<Self::Output>;
}

struct Foo;

impl GetInstance for Foo {
    type Output = Self;
    fn get_instance(id: i64) -> Option<Foo> {
        // Here you could do something like a Database lookup or similarly
        if id == 1 {
            Some(Foo)
        } else {
            None
        }
    }
}

struct User;

impl GetInstance for User {
    type Output = Self;
    fn get_instance(id: i64) -> Option<User> {
        // Here you could do something like a Database lookup or similarly
        if id == 2 {
            Some(User)
        } else {
            None
```

```
        }
    }
}

fn main() {
    let user_auth = Authenticator::<User>::new();
    let other_auth = Authenticator::<Foo>::new();

    assert!(user_auth.auth(2));
    assert!(!user_auth.auth(1));

    assert!(other_auth.auth(1));
    assert!(!other_auth.auth(2));

}
```

Lire PhantomData en ligne: https://riptutorial.com/fr/rust/topic/7226/phantomdata

Chapitre 37: Pointeurs bruts

Syntaxe

- laisser raw_ptr = & pointee comme * const *type* // créer un pointeur brut constant sur certaines données
- laisser raw_mut_ptr = & mut pointee comme * mut *type* // créer un pointeur brut mutable sur des données mutables
- laisser deref = * raw_ptr // déréférencer un pointeur brut (nécessite un blocage non sécurisé)

Remarques

- Les pointeurs bruts ne sont pas sûrs de pointer vers une adresse de mémoire valide et, par conséquent, une utilisation imprudente peut entraîner des erreurs inattendues (et probablement fatales).
- Toute référence Rust normale (par exemple, `&my_object` où le type de `my_object` est T) contraindra à `*const T` De même, les références mutables contraignent `*mut T`
- Les pointeurs bruts ne déplacent pas la propriété (contrairement aux valeurs Box qui)

Examples

Créer et utiliser des pointeurs bruts constants

```
// Let's take an arbitrary piece of data, a 4-byte integer in this case
let some_data: u32 = 14;

// Create a constant raw pointer pointing to the data above
let data_ptr: *const u32 = &some_data as *const u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    let deref_data: u32 = *data_ptr;
    println!("Dereferenced data: {}", deref_data);
}
```

Le code ci-dessus va sortir: `Dereferenced data: 14`

Créer et utiliser des pointeurs bruts mutables

```
// Let's take a mutable piece of data, a 4-byte integer in this case
let mut some_data: u32 = 14;

// Create a mutable raw pointer pointing to the data above
let data_ptr: *mut u32 = &mut some_data as *mut u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
```

```
unsafe {
    *data_ptr = 20;
    println!("Dereferenced data: {}", some_data);
}
```

Le code ci-dessus va sortir: `Dereferenced data: 20`

Initialiser un pointeur brut sur null

Contrairement aux références Rust normales, les pointeurs bruts sont autorisés à prendre des valeurs nulles.

```
use std::ptr;

// Create a const NULL pointer
let null_ptr: *const u16 = ptr::null();

// Create a mutable NULL pointer
let mut_null_ptr: *mut u16 = ptr::null_mut();
```

Déréférencement de chaîne

Tout comme en C, les pointeurs bruts de Rust peuvent pointer vers d'autres pointeurs bruts (qui à leur tour peuvent indiquer d'autres pointeurs bruts).

```
// Take a regular string slice
let planet: &str = "Earth";

// Create a constant pointer pointing to our string slice
let planet_ptr: *const &str = &planet as *const &str;

// Create a constant pointer pointing to the pointer
let planet_ptr_ptr: *const *const &str = &planet_ptr as *const *const &str;

// This can go on...
let planet_ptr_ptr_ptr = &planet_ptr_ptr as *const *const *const &str;

unsafe {
    // Direct usage
    println!("The name of our planet is: {}", planet);
    // Single dereference
    println!("The name of our planet is: {}", *planet_ptr);
    // Double dereference
    println!("The name of our planet is: {}", **planet_ptr_ptr);
    // Triple dereference
    println!("The name of our planet is: {}", ***planet_ptr_ptr_ptr);
}
```

Cela va produire: `The name of our planet is: Earth` quatre fois.

Affichage des pointeurs bruts

Rust a un formateur par défaut pour les types de pointeurs qui peuvent être utilisés pour afficher des pointeurs.

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

Cela va générer quelque chose comme ceci:

```
Data address: 0x7fff59f6bcc0
Raw pointer address: 0x7fff59f6bcc0
Null pointer address: 0x0
```

Lire Pointeurs bruts en ligne: https://riptutorial.com/fr/rust/topic/7270/pointeurs-bruts

Chapitre 38: Regex

Introduction

La bibliothèque standard de Rust ne contient aucun analyseur / analyseur de regex, mais le `regex` (qui se trouve dans la rust-lang-nursery et donc semi-officiel) fournit un analyseur regex. Cette section de la documentation fournira un aperçu de la manière d'utiliser la caisse `regex` dans des situations courantes, ainsi que des instructions d'installation et toute autre remarque utile nécessaire lors de l'utilisation de la caisse.

Examples

Match simple et recherche

Le support des expressions régulières pour tust est fourni par le `regex` crate, ajoutez-le à votre `Cargo.toml` :

```
[dependencies]
regex = "0.1"
```

L'interface principale du `regex` crate est `regex::Regex` :

```
extern crate regex;
use regex::Regex;

fn main() {
    //"r" stands for "raw" strings, you probably
    // need them because rustc checks escape sequences,
    // although you can always use "\\" withour "r"
    let num_regex = Regex::new(r"\d+").unwrap();
    // is_match checks if string matches the pattern
    assert!(num_regex.is_match("some string with number 1"));

    let example_string = "some 123 numbers";
    // Regex::find searches for pattern and returns Option<(usize,usize)>,
    // which is either indexes of first and last bytes of match
    // or "None" if nothing matched
    match num_regex.find(example_string) {
        // Get the match slice from string, prints "123"
        Some(x) => println!("{}", &example_string[x.0 .. x.1]),
        None    => unreachable!()
    }
}
```

Groupes de capture

```
extern crate regex;
use regex::Regex;

fn main() {
```

```
    let rg = Regex::new(r"was (\d+)").unwrap();
    // Regex::captures returns Option<Captures>,
    // first element is the full match and others
    // are capture groups
    match rg.captures("The year was 2016") {
        // Access captures groups via Captures::at
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.at(1)),
        None    => unreachable!()
    }

    // Regex::captures also supports named capture groups
    let rg_w_named = Regex::new(r"was (?P<year>\d+)").unwrap();
    match rg_w_named.captures("The year was 2016") {
        // Named capures groups are accessed via Captures::name
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.name("year")),
        None    => unreachable!()
    }

}
```

Remplacer

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"(\d+)").unwrap();

    // Regex::replace replaces first match
    // from it's first argument with the second argument
    // => Some string with numbers (not really)
    rg.replace("Some string with numbers 123", "(not really)");

    // Capture groups can be accesed via $number
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $1)");

    let rg = Regex::new(r"(?P<num>\d+)").unwrap();

    // Named capture groups can be accesed via $name
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $num)");

    // Regex::replace_all replaces all the matches, not only the first
    // => Some string with numbers (not really) (not really)
    rg.replace_all("Some string with numbers 123 321", "(not really)");
}
```

Lire Regex en ligne: https://riptutorial.com/fr/rust/topic/7184/regex

Chapitre 39: Rouille nue

Introduction

La bibliothèque Rust Standard (`std`) est compilée uniquement avec une poignée d'architectures. Ainsi, pour compiler dans d'autres architectures (supportées par LLVM), les programmes Rust peuvent choisir de ne pas utiliser l'intégralité du `std` , mais d'utiliser uniquement le sous-ensemble portable, appelé Core Library (`core`).

Examples

#! [no_std] Bonjour, Monde!

```
#![feature(start, libc, lang_items)]
#![no_std]
#![no_main]

// The libc crate allows importing functions from C.
extern crate libc;

// A list of C functions that are being imported
extern {
    pub fn printf(format: *const u8, ...) -> i32;
}

#[no_mangle]
// The main function, with its input arguments ignored, and an exit status is returned
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
    // Print "Hello, World" to stdout using printf
    unsafe {
        printf(b"Hello, World!\n" as *const u8);
    }

    // Exit with a return status of 0.
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { panic!() }
```

Lire Rouille nue en ligne: https://riptutorial.com/fr/rust/topic/8344/rouille-nue

Chapitre 40: Rouille orientée objet

Introduction

Rust est orienté objet en ce sens que ses types de données algébriques peuvent avoir des méthodes associées, ce qui les rend des objets au sens de données stockées avec du code qui sait les utiliser.

La rouille ne supporte cependant pas l'héritage, favorisant la composition avec des traits. Cela signifie que beaucoup de modèles OO ne fonctionnent pas tel quel et doivent être modifiés. Certains sont totalement hors de propos.

Examples

Héritage avec des traits

Dans Rust, il n'y a pas de concept d'héritage des propriétés d'une structure. Au lieu de cela, lorsque vous concevez la relation entre les objets, faites-la de manière à ce que sa fonctionnalité soit définie par une interface (une **caractéristique** de Rust). Cela favorise la composition par rapport à l'héritage , ce qui est considéré comme plus utile et plus facile à étendre à des projets plus importants.

Voici un exemple d'utilisation d'un exemple d'héritage dans Python:

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

Pour traduire cela en rouille, nous devons éliminer ce qui constitue un animal et le transformer en traits.

```
trait Speaks {
     fn speak(&self);

     fn noise(&self) -> &str;
}

trait Animal {
    fn animal_type(&self) -> &str;
}

struct Dog {}

impl Animal for Dog {
```

```
    fn animal_type(&self) -> &str {
        "dog"
    }
}

impl Speaks for Dog {
    fn speak(&self) {
        println!("The dog said {}", self.noise());
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

fn main() {
    let dog = Dog {};
    dog.speak();
}
```

Notez comment nous avons divisé cette classe parente en deux composants distincts: la partie qui définit la structure en tant qu'animal et la partie qui lui permet de parler.

Les lecteurs avertis remarqueront que ce n'est pas du tout un à un, car chaque implémenteur doit réimplémenter la logique pour imprimer une chaîne sous la forme "{animal} a dit {noise}". Vous pouvez le faire avec une légère refonte de l'interface où nous implémentons `Speak` for `Animal` :

```
trait Speaks {
     fn speak(&self);
}

trait Animal {
    fn animal_type(&self) -> &str;
    fn noise(&self) -> &str;
}

impl<T> Speaks for T where T: Animal {
    fn speak(&self) {
        println!("The {} said {}", self.animal_type(), self.noise());
    }
}

struct Dog {}
struct Cat {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

impl Animal for Cat {
    fn animal_type(&self) -> &str {
        "cat"
    }
```

```
    fn noise(&self) -> &str {
        "meow"
    }
}

fn main() {
    let dog = Dog {};
    let cat = Cat {};
    dog.speak();
    cat.speak();
}
```

Remarquez maintenant que l'animal fait du bruit et parle simplement maintenant, il a une implémentation pour tout ce qui est un animal. C'est beaucoup plus flexible que la méthode précédente et l'héritage Python. Par exemple, si vous voulez ajouter un `Human` avec un son différent, nous pouvons simplement avoir une autre implémentation de `speak` pour quelque chose d' `Human` :

```
trait Human {
    fn name(&self) -> &str;
    fn sentence(&self) -> &str;
}

struct Person {}

impl<T> Speaks for T where T: Human {
    fn speak(&self) {
        println!("{} said {}", self.name(), self.sentence());
    }
}
```

Motif de visiteur

L'exemple typique de visiteur en Java serait:

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }
```

```
    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

Cela peut être facilement traduit par Rust, de deux manières.

La première méthode utilise le polymorphisme d'exécution:

```
trait ShapeVisitor {
    fn visit_circle(&mut self, c: &Circle);
    fn visit_rectangle(&mut self, r: &Rectangle);
}

trait Shape {
    fn accept(&self, sv: &mut ShapeVisitor);
}

struct Circle {
    center: Point,
```

```
    radius: f64,
}

struct Rectangle {
    lowerLeftCorner: Point,
    upperRightCorner: Point,
}

impl Shape for Circle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_circle(self);
    }
}

impl Rectangle {
    fn length() -> double { ... }
    fn width() -> double { ... }
}

impl Shape for Rectangle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_rectangle(self);
    }
}

fn computeArea(s: &Shape) -> f64 {
    struct AreaCalculator {
        area: f64,
    }

    impl ShapeVisitor for AreaCalculator {
        fn visit_circle(&mut self, c: &Circle) {
            self.area = std::f64::consts::PI * c.radius * c.radius;
        }
        fn visit_rectangle(&mut self, r: &Rectangle) {
            self.area = r.length() * r.width();
        }
    }

    let mut ac = AreaCalculator { area: 0.0 };
    s.accept(&mut ac);
    ac.area
}
```

La seconde méthode utilise à la place le polymorphisme à la compilation, seules les différences sont affichées ici:

```
trait Shape {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V);
}

impl Shape for Circle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

impl Shape for Rectangle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
```

```
    }
}

fn computeArea<S: Shape>(s: &S) -> f64 {
    // same body
}
```

Lire Rouille orientée objet en ligne: https://riptutorial.com/fr/rust/topic/6737/rouille-orientee-objet

Chapitre 41: rouiller

Introduction

`rustup` gère votre installation de rouille et vous permet d'installer différentes versions, qui peuvent être configurées et échangées facilement.

Examples

Mise en place

Installez la chaîne d'outils avec

```
curl https://sh.rustup.rs -sSf | sh
```

Vous devriez avoir la dernière version stable de la rouille déjà. Vous pouvez vérifier cela en tapant

```
rustc --version
```

Si vous souhaitez mettre à jour, exécutez simplement

```
rustup update
```

Lire rouiller en ligne: https://riptutorial.com/fr/rust/topic/8942/rouiller

Chapitre 42: Serde

Introduction

Serde est un ialization **ser** populaire et cadre **de** sérialisation pour Rust, utilisé pour convertir des *données* (par exemple un *numéro* de *série* JSON et XML) aux structures Rouille et vice versa. Serde prend en charge de nombreux formats, notamment: JSON, YAML, TOML, BSON, Pickle et XML.

Examples

Struct ↔ JSON

main.rs

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Cargo.toml

```
[package]
name = "serde-example"
```

```
version = "0.1.0"
build = "build.rs"

[dependencies]
serde = "0.9"
serde_json = "0.9"
serde_derive = "0.9"
```

Sérialiser enum comme chaîne

```
extern crate serde;
extern crate serde_json;

macro_rules! enum_str {
    ($name:ident { $($variant:ident($str:expr), )* }) => {
        #[derive(Clone, Copy, Debug, Eq, PartialEq)]
        pub enum $name {
            $($variant,)*
        }

        impl ::serde::Serialize for $name {
            fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
                where S: ::serde::Serializer,
            {
                // Serialize the enum as a string.
                serializer.serialize_str(match *self {
                    $( $name::$variant => $str, )*
                })
            }
        }

        impl ::serde::Deserialize for $name {
            fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
                where D: ::serde::Deserializer,
            {
                struct Visitor;

                impl ::serde::de::Visitor for Visitor {
                    type Value = $name;

                    fn expecting(&self, formatter: &mut ::std::fmt::Formatter) ->
::std::fmt::Result {
                        write!(formatter, "a string for {}", stringify!($name))
                    }

                    fn visit_str<E>(self, value: &str) -> Result<$name, E>
                        where E: ::serde::de::Error,
                    {
                        match value {
                            $( $str => Ok($name::$variant), )*
                            _ => Err(E::invalid_value(::serde::de::Unexpected::Other(
                                &format!("unknown {} variant: {}", stringify!($name), value)
                            ), &self)),
                        }
                    }
                }

                // Deserialize the enum from a string.
                deserializer.deserialize_str(Visitor)
```

```
            }
        }
    }
}

enum_str!(LanguageCode {
    English("en"),
    Spanish("es"),
    Italian("it"),
    Japanese("ja"),
    Chinese("zh"),
});

fn main() {
    use LanguageCode::*;
    let languages = vec![English, Spanish, Italian, Japanese, Chinese];

    // Prints ["en","es","it","ja","zh"]
    println!("{}", serde_json::to_string(&languages).unwrap());

    let input = r#" "ja" "#;
    assert_eq!(Japanese, serde_json::from_str(input).unwrap());
}
```

Sérialiser les champs en tant que camelCase

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Serialize)]
struct Person {
    #[serde(rename="firstName")]
    first_name: String,
    #[serde(rename="lastName")]
    last_name: String,
}

fn main() {
    let person = Person {
        first_name: "Joel".to_string(),
        last_name: "Spolsky".to_string(),
    };

    let json = serde_json::to_string_pretty(&person).unwrap();

    // Prints:
    //
    //    {
    //      "firstName": "Joel",
    //      "lastName": "Spolsky"
    //    }
    println!("{}", json);
}
```

Valeur par défaut pour le champ

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is
    // not included in the input.
    #[serde(default="default_resource")]
    resource: String,

    // Use the type's implementation of std::default::Default if
    // "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not
    // included in the input. This may also be a trait method.
    #[serde(default="Priority::lowest")]
    priority: Priority,
}

fn default_resource() -> String {
    "/".to_string()
}

/// Timeout in seconds.
#[derive(Deserialize, Debug)]
struct Timeout(u32);
impl Default for Timeout {
    fn default() -> Self {
        Timeout(30)
    }
}

#[derive(Deserialize, Debug)]
enum Priority { ExtraHigh, High, Normal, Low, ExtraLow }
impl Priority {
    fn lowest() -> Self { Priority::ExtraLow }
}

fn main() {
    let json = r#"
        [
          {
            "resource": "/users"
          },
          {
            "timeout": 5,
            "priority": "High"
          }
        ]
    "#;

    let requests: Vec<Request> = serde_json::from_str(json).unwrap();

    // The first request has resource="/users", timeout=30, priority=ExtraLow
    println!("{:?}", requests[0]);

    // The second request has resource="/", timeout=5, priority=High
    println!("{:?}", requests[1]);
```

```
}
```

Ignorer le champ de sérialisation

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be skipped.
    #[serde(skip_serializing_if="Map::is_empty")]
    metadata: Map<String, String>,
}

fn main() {
    let resources = vec![
        Resource {
            name: "Stack Overflow".to_string(),
            hash: "b6469c3f31653d281bbbfa6f94d60fea130abe38".to_string(),
            metadata: Map::new(),
        },
        Resource {
            name: "GitHub".to_string(),
            hash: "5cb7a0c47e53854cd00e1a968de5abce1c124601".to_string(),
            metadata: {
                let mut metadata = Map::new();
                metadata.insert("headquarters".to_string(),
                                "San Francisco".to_string());
                metadata
            },
        },
    ];

    let json = serde_json::to_string_pretty(&resources).unwrap();

    // Prints:
    //
    //    [
    //      {
    //        "name": "Stack Overflow"
    //      },
    //      {
    //        "name": "GitHub",
    //        "metadata": {
    //          "headquarters": "San Francisco"
    //        }
    //      }
    //    ]
    println!("{}", json);
```

```
}
```

Implémenter la sérialisation pour un type de carte personnalisé

```
impl<K, V> Serialize for MyMap<K, V>
    where K: Serialize,
          V: Serialize
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            state.serialize_entry(k, v)?;
        }
        state.end()
    }
}
```

Implémenter Deserialize pour un type de carte personnalisé

```
// A Visitor is a type that holds methods that a Deserializer can drive
// depending on what is contained in the input data.
//
// In the case of a map we need generic type parameters K and V to be
// able to set the output type correctly, but don't require any state.
// This is an example of a "zero sized type" in Rust. The PhantomData
// keeps the compiler from complaining about unused generic type
// parameters.
struct MyMapVisitor<K, V> {
    marker: PhantomData<MyMap<K, V>>
}

impl<K, V> MyMapVisitor<K, V> {
    fn new() -> Self {
        MyMapVisitor {
            marker: PhantomData
        }
    }
}

// This is the trait that Deserializers are going to be driving. There
// is one method for each type of data that our type knows how to
// deserialize from. There are many other methods that are not
// implemented here, for example deserializing from integers or strings.
// By default those methods will return an error, which makes sense
// because we cannot deserialize a MyMap from an integer or string.
impl<K, V> de::Visitor for MyMapVisitor<K, V>
    where K: Deserialize,
          V: Deserialize
{
    // The type that our Visitor is going to produce.
    type Value = MyMap<K, V>;

    // Deserialize MyMap from an abstract "map" provided by the
    // Deserializer. The MapVisitor input is a callback provided by
    // the Deserializer to let us see each entry in the map.
```

```
    fn visit_map<M>(self, mut visitor: M) -> Result<Self::Value, M::Error>
        where M: de::MapVisitor
    {
        let mut values = MyMap::with_capacity(visitor.size_hint().0);

        // While there are entries remaining in the input, add them
        // into our map.
        while let Some((key, value)) = visitor.visit()? {
            values.insert(key, value);
        }

        Ok(values)
    }

    // As a convenience, provide a way to deserialize MyMap from
    // the abstract "unit" type. This corresponds to `null` in JSON.
    // If your JSON contains `null` for a field that is supposed to
    // be a MyMap, we interpret that as an empty map.
    fn visit_unit<E>(self) -> Result<Self::Value, E>
        where E: de::Error
    {
        Ok(MyMap::new())
    }

    // When an unexpected data type is encountered, this method will
    // be invoked to inform the user what is actually expected.
    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a map or `null`")
    }
}

// This is the trait that informs Serde how to deserialize MyMap.
impl<K, V> Deserialize for MyMap<K, V>
    where K: Deserialize,
          V: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Instantiate our Visitor and ask the Deserializer to drive
        // it over the input data, resulting in an instance of MyMap.
        deserializer.deserialize_map(MyMapVisitor::new())
    }
}
```

Traiter un tableau de valeurs sans les mettre en mémoire tampon dans un Vec

Supposons que nous ayons un tableau d'entiers et que nous voulons déterminer la valeur maximale sans conserver tout le tableau en mémoire en une seule fois. Cette approche peut être adaptée pour gérer une variété d'autres situations dans lesquelles les données doivent être traitées tout en étant désérialisées au lieu d'être après.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use serde::{de, Deserialize, Deserializer};

use std::cmp;
```

```
use std::fmt;
use std::marker::PhantomData;

#[derive(Deserialize)]
struct Outer {
    id: String,

    // Deserialize this field by computing the maximum value of a sequence
    // (JSON array) of values.
    #[serde(deserialize_with = "deserialize_max")]
    // Despite the struct field being named `max_value`, it is going to come
    // from a JSON field called `values`.
    #[serde(rename(deserialize = "values"))]
    max_value: u64,
}

/// Deserialize the maximum of a sequence of values. The entire sequence
/// is not buffered into memory as it would be if we deserialize to Vec<T>
/// and then compute the maximum later.
///
/// This function is generic over T which can be any type that implements
/// Ord. Above, it is used with T=u64.
fn deserialize_max<T, D>(deserializer: D) -> Result<T, D::Error>
    where T: Deserialize + Ord,
          D: Deserializer
{
    struct MaxVisitor<T>(PhantomData<T>);

    impl<T> de::Visitor for MaxVisitor<T>
        where T: Deserialize + Ord
    {
        /// Return type of this visitor. This visitor computes the max of a
        /// sequence of values of type T, so the type of the maximum is T.
        type Value = T;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            write!(formatter, "a sequence of numbers")
        }

        fn visit_seq<V>(self, mut visitor: V) -> Result<T, V::Error>
            where V: de::SeqVisitor
        {
            // Start with max equal to the first value in the seq.
            let mut max = match visitor.visit()? {
                Some(value) => value,
                None => {
                    // Cannot take the maximum of an empty seq.
                    let msg = "no values in seq when looking for maximum";
                    return Err(de::Error::custom(msg));
                }
            };

            // Update the max while there are additional values.
            while let Some(value) = visitor.visit()? {
                max = cmp::max(max, value);
            }

            Ok(max)
        }
    }
```

```
    // Create the visitor and ask the deserializer to drive it. The
    // deserializer will call visitor.visit_seq if a seq is present in
    // the input data.
    let visitor = MaxVisitor(PhantomData);
    deserializer.deserialize_seq(visitor)
}

fn main() {
    let j = r#"
        {
          "id": "demo-deserialize-max",
          "values": [
            256,
            100,
            384,
            314,
            271
          ]
        }
    "#;

    let out: Outer = serde_json::from_str(j).unwrap();

    // Prints "max value: 384"
    println!("max value: {}", out.max_value);
}
```

Limites de type générique manuscrit

Lorsque vous `Deserialize` implémentations `Serialize` et `Deserialize` pour des structures avec des paramètres de type génériques, la plupart du temps, Serde est capable de déduire les limites de trait correctes sans l'aide du programmeur. Il utilise plusieurs heuristiques pour deviner la limite droite, mais surtout, il place une limite de `T: Serialize` sur chaque paramètre de type `T` qui fait partie d'un champ sérialisé et une limite de `T: Deserialize` sur chaque paramètre de type `T` qui fait partie d'un champ désérialisé. Comme avec la plupart des heuristiques, ce n'est pas toujours correct et Serde fournit un hachage d'échappement pour remplacer la liaison générée automatiquement par une écriture écrite par le programmeur.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use serde::de::{self, Deserialize, Deserializer};

use std::fmt::Display;
use std::str::FromStr;

#[derive(Deserialize, Debug)]
struct Outer<'a, S, T: 'a + ?Sized> {
    // When deriving the Deserialize impl, Serde would want to generate a bound
    // `S: Deserialize` on the type of this field. But we are going to use the
    // type's `FromStr` impl instead of its `Deserialize` impl by going through
    // `deserialize_from_str`, so we override the automatically generated bound
    // by the one required for `deserialize_from_str`.
    #[serde(deserialize_with = "deserialize_from_str")]
    #[serde(bound(deserialize = "S: FromStr, S::Err: Display"))]
    s: S,
```

```
    // Here Serde would want to generate a bound `T: Deserialize`. That is a
    // stricter condition than is necessary. In fact, the `main` function below
    // uses T=str which does not implement Deserialize. We override the
    // automatically generated bound by a looser one.
    #[serde(bound(deserialize = "Ptr<'a, T>: Deserialize"))]
    ptr: Ptr<'a, T>,
}

/// Deserialize a type `S` by deserializing a string, then using the `FromStr`
/// impl of `S` to create the result. The generic type `S` is not required to
/// implement `Deserialize`.
fn deserialize_from_str<S, D>(deserializer: D) -> Result<S, D::Error>
    where S: FromStr,
          S::Err: Display,
          D: Deserializer
{
    let s: String = try!(Deserialize::deserialize(deserializer));
    S::from_str(&s).map_err(|e| de::Error::custom(e.to_string()))
}

/// A pointer to `T` which may or may not own the data. When deserializing we
/// always want to produce owned data.
#[derive(Debug)]
enum Ptr<'a, T: 'a + ?Sized> {
    Ref(&'a T),
    Owned(Box<T>),
}

impl<'a, T: 'a + ?Sized> Deserialize for Ptr<'a, T>
    where Box<T>: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        let box_t = try!(Deserialize::deserialize(deserializer));
        Ok(Ptr::Owned(box_t))
    }
}

fn main() {
    let j = r#"
        {
          "s": "1234567890",
          "ptr": "owned"
        }
    "#;

    let result: Outer<u64, str> = serde_json::from_str(j).unwrap();

    // result = Outer { s: 1234567890, ptr: Owned("owned") }
    println!("result = {:?}", result);
}
```

Implémenter Serialize et Deserialize pour un type dans un autre caisse

La règle de cohérence de Rust exige que soit le trait, soit le type pour lequel vous implémentez le trait, soit défini dans le même dossier que l'impl. Le modèle newtype et la contrainte `Deref` permettent d'implémenter Serialize et Deserialize pour un type qui se comporte de la même

manière que celui souhaité.

```
use serde::{Serialize, Serializer, Deserialize, Deserializer};
use std::ops::Deref;

// Pretend this module is from some other crate.
mod not_my_crate {
    pub struct Data { /* ... */ }
}

// This single-element tuple struct is called a newtype struct.
struct Data(not_my_crate::Data);

impl Serialize for Data {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        // Any implementation of Serialize.
    }
}

impl Deserialize for Data {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Any implementation of Deserialize.
    }
}

// Enable `Deref` coercion.
impl Deref for Data {
    type Target = not_my_crate::Data;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now `Data` can be used in ways that require it to implement
// Serialize and Deserialize.
#[derive(Serialize, Deserialize)]
struct Outer {
    id: u64,
    name: String,
    data: Data,
}
```

Lire Serde en ligne: https://riptutorial.com/fr/rust/topic/1170/serde

Chapitre 43: Tableaux, vecteurs et tranches

Examples

Tableaux

Un tableau est une liste d'objets de type unique, allouée de manière statique et de taille statique.

Les tableaux sont généralement créés en incluant une liste d'éléments d'un type donné entre crochets. Le type d'un tableau est désigné par la syntaxe spéciale: `[T; N]` où `T` est le type de ses éléments et `N` leur nombre, qui doivent tous deux être connus au moment de la compilation.

Par exemple, `[4u64, 5, 6]` est un tableau de 3 éléments de type `[u64; 3]` . Remarque: `5` et `6` sont supposés être de type `u64` .

Exemple

```
fn main() {
    // Arrays have a fixed size.
    // All elements are of the same type.
    let array = [1, 2, 3, 4, 5];

    // Create an array of 20 elements where all elements are the same.
    // The size should be a compile-time constant.
    let ones = [1; 20];

    // Get the length of an array.
    println!("Length of ones: {}", ones.len());

    // Access an element of an array.
    // Indexing starts at 0.
    println!("Second element of array: {}", array[1]);

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", array[5]);
}
```

Limites

La correspondance des motifs sur les tableaux (ou les tranches) n'est pas prise en charge dans Rust stable (voir les modèles de tranches et les numéros 23121).

La rouille ne supporte pas la généricité des chiffres au niveau du type (voir RFC # 1657). Par conséquent, il n'est pas possible d'implémenter simplement un trait pour tous les tableaux (de toutes tailles). Par conséquent, les caractéristiques standard ne sont implémentées que pour les

tableaux d'un nombre limité d'éléments (dernier contrôle, jusqu'à 32 inclus). Les tableaux avec plus d'éléments sont supportés, mais n'implémentent pas les traits standard (voir docs).

Nous espérons que ces restrictions seront levées à l'avenir.

Vecteurs

Un vecteur est essentiellement un pointeur sur une liste d'objets de type unique, de taille dynamique et allouée au tas.

Exemple

```
fn main() {
    // Create a mutable empty vector
    let mut vector = Vec::new();

    vector.push(20);
    vector.insert(0, 10); // insert at the beginning

    println!("Second element of vector: {}", vector[1]); // 20

    // Create a vector using the `vec!` macro
    let till_five = vec![1, 2, 3, 4, 5];

    // Create a vector of 20 elements where all elements are the same.
    let ones = vec![1; 20];

    // Get the length of a vector.
    println!("Length of ones: {}", ones.len());

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", till_five[5]);
}
```

Tranches

Les tranches sont des vues dans une liste d'objets et ont le type `[T]` , indiquant une tranche d'objets de type `T`

Une tranche est un type non formaté et ne peut donc être utilisé qu'avec un pointeur. *(Analogie du monde des chaînes: `str` , appelée string slice, est également non formatée.)*

Les tableaux sont forcés dans des tranches et les vecteurs peuvent être déréférencés en tranches. Par conséquent, les méthodes de tranche peuvent être appliquées aux deux. *(Analogie du monde de la chaîne: `str` est à `String` , ce que `[T]` à `Vec<T>` .)*

```
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8];
    let slice = &vector[3..6];
    println!("length of slice: {}", slice.len()); // 3
```

```
    println!("slice: {:?}", slice); // [4, 5, 6]
}
```

Lire Tableaux, vecteurs et tranches en ligne: https://riptutorial.com/fr/rust/topic/5004/tableaux--vecteurs-et-tranches

Chapitre 44: The Drop Trait - Destructeurs à Rust

Remarques

Utiliser le Drop Trait ne signifie pas qu'il sera exécuté à chaque fois. Bien qu'il s'exécute lorsqu'il est hors de portée ou qu'il se déroule, ce n'est peut-être pas toujours le cas, par exemple lorsque `mem::forget` est appelé.

Cela est dû au fait qu'une panique lors du déroulement du programme entraîne une interruption du programme. Il aurait également pu être compilé avec `Abort on Panic` activé.

Pour plus d'informations, consultez le livre: https://doc.rust-lang.org/book/drop.html

Examples

Implémentation simple de la goutte

```
use std::ops::Drop;

struct Foo(usize);

impl Drop for Foo {
    fn drop(&mut self) {
        println!("I had a {}", self.0);
    }
}
```

Drop for Cleanup

```
use std::ops::Drop;

#[derive(Debug)]
struct Bar(i32);

impl Bar {
    fn get<'a>(&'a mut self) -> Foo<'a> {
        let temp = self.0; // Since we will also capture `self` we..
                           // ..will have to copy the value out first
        Foo(self, temp) // Let's take the i32
    }
}

struct Foo<'a>(&'a mut Bar, i32); // We specify that we want a mutable borrow..
                                  // ..so we can put it back later on

impl<'a> Drop for Foo<'a> {
    fn drop(&mut self) {
        if self.1 < 10 { // This is just an example, you could also just put..
                         // ..it back as is
```

```
            (self.0).0 = self.1;
        }
    }
}

fn main() {
    let mut b = Bar(0);
    println!("{:?}", b);
    {
        let mut a : Foo = b.get(); // `a` now holds a reference to `b`..
        a.1 = 2;                   // .. and will hold it until end of scope
    }                              // .. here

    println!("{:?}", b);
    {
        let mut a : Foo = b.get();
        a.1 = 20;
    }
    println!("{:?}", b);
}
```

Drop vous permet de créer des conceptions simples et sûres.

Suppression de la journalisation pour le débogage de la gestion de la mémoire d'exécution

La gestion de la mémoire d'exécution avec Rc peut être très utile, mais elle peut également être difficile à comprendre, en particulier si votre code est très complexe et qu'une instance unique est référencée par des dizaines, voire des centaines d'autres types.

Ecrire un trait Drop qui inclut `println!("Dropping StructName: {:?}", self);` peut être extrêmement utile pour le débogage, car il vous permet de voir précisément quand les fortes références à une instance de structure sont épuisées.

Lire The Drop Trait - Destructeurs à Rust en ligne: https://riptutorial.com/fr/rust/topic/7233/the-drop-trait---destructeurs-a-rust

Chapitre 45: Traitement du signal

Remarques

Rust ne dispose pas d'un moyen approprié et idiomatique pour communiquer avec les signaux du système d'exploitation, mais il y a des caisses qui fournissent un traitement du signal, mais elles sont très *expérimentales* et *peu sûres* .

Cependant, il y a une discussion dans le dépôt rust-lang / rfcs sur l'implémentation de la gestion du signal natif pour la rouille.

Discussion RFC: https://github.com/rust-lang/rfcs/issues/1368

Examples

Traitement du signal avec caisse de signal de chan

La caisse de signal de chan fournit une solution pour gérer le signal de système d'exploitation en utilisant des canaux, bien que cette caisse soit **expérimentale** et devrait être utilisée **avec soin** .

Exemple tiré de BurntSushi / chan-signal .

```
#[macro_use]
extern crate chan;
extern crate chan_signal;

use chan_signal::Signal;

fn main() {
    // Signal gets a value when the OS sent a INT or TERM signal.
    let signal = chan_signal::notify(&[Signal::INT, Signal::TERM]);
    // When our work is complete, send a sentinel value on `sdone`.
    let (sdone, rdone) = chan::sync(0);
    // Run work.
    ::std::thread::spawn(move || run(sdone));

    // Wait for a signal or for work to be done.
    chan_select! {
        signal.recv() -> signal => {
            println!("received signal: {:?}", signal)
        },
        rdone.recv() => {
            println!("Program completed normally.");
        }
    }
}

fn run(_sdone: chan::Sender<()>) {
    println!("Running work for 5 seconds.");
    println!("Can you send a signal quickly enough?");
    // Do some work.
    ::std::thread::sleep_ms(5000);
```

```
    // _sdone gets dropped which closes the channel and causes `rdone`
    // to unblock.
}
```

Manipulation des signaux avec une caisse nix.

Le nix crate fournit une API UNIX Rust pour gérer les signaux, mais il faut utiliser **une** rouille **non sécurisée** pour être **prudent** .

```
use nix::sys::signal;

extern fn handle_sigint(_:i32) {
    // Be careful here...
}

fn main() {
    let sig_action = signal::SigAction::new(handle_sigint,
                                            signal::SockFlag::empty(),
                                            signal::SigSet::empty());
    signal::sigaction(signal::SIGINT, &sig_action);
}
```

Exemple Tokio

La caisse de tokio-signal fournit une solution basée sur tokio pour gérer les signaux. Il est encore dans ses premiers stades cependant.

```
extern crate futures;
extern crate tokio_core;
extern crate tokio_signal;

use futures::{Future, Stream};
use tokio_core::reactor::Core
use tokio_signal::unix::{self as unix_signal, Signal};
use std::thread::{self, sleep};
use std::time::Duration;
use std::sync::mpsc::{channel, Receiver};

fn run(signals: Receiver<i32>) {
    loop {
        if let Some(signal) = signals.try_recv() {
            eprintln!("received {} signal");
        }
        sleep(Duration::from_millis(1));
    }
}

fn main() {
    // Create channels for sending and receiving signals
    let (signals_tx, signals_rx) = channel();

    // Execute the program with the receiving end of the channel
    // for listening to any signals that are sent to it.
    thread::spawn(move || run(signals_rx));
```

```
    // Create a stream that will select over SIGINT, SIGTERM, and SIGHUP signals.
    let signals = Signal::new(unix_signal::SIGINT, &handle).flatten_stream()
        .select(Signal::new(unix_signal::SIGTERM, &handle).flatten_stream())
        .select(Signal::new(unix_signal::SIGHUP, &handle).flatten_stream());

    // Execute the event loop that will listen for and transmit received
    // signals to the shell.
    core.run(signals.for_each(|signal| {
        let _ = signals_tx.send(signal);
        Ok(())
    })).unwrap();
}
```

Lire Traitement du signal en ligne: https://riptutorial.com/fr/rust/topic/3995/traitement-du-signal

Chapitre 46: Traits

Introduction

Les traits sont un moyen de décrire un «contrat» qu'une `struct` doit implémenter. Les caractères définissent généralement des signatures de méthode mais peuvent également fournir des implémentations basées sur d'autres méthodes du trait, à condition que les *limites de trait le* permettent.

Pour ceux qui sont familiers avec la programmation orientée objet, les traits peuvent être considérés comme des interfaces avec quelques différences subtiles.

Syntaxe

- trait Trait {méthode fn (...) -> ReturnType; ...}
- trait Trait: Bound {fn method (...) -> ReturnType; ...}
- impliquer Trait for Type {fn method (...) -> ReturnType {...} ...}
- implicite <T> Trait pour T où T: Bounds {méthode fn (...) -> ReturnType {...} ...}

Remarques

- Les traits sont généralement assimilés à des interfaces, mais il est important de faire la distinction entre les deux. Dans les langages OO tels que Java, les interfaces font partie intégrante des classes qui les étendent. Dans Rust, le compilateur ne sait rien des traits d'une structure à moins que ces traits ne soient utilisés.

Examples

Les bases

Créer un trait

```
trait Speak {
    fn speak(&self) -> String;
}
```

Implémenter un trait

```
struct Person;
struct Dog;

impl Speak for Person {
    fn speak(&self) -> String {
        String::from("Hello.")
```

```
    }
}

impl Speak for Dog {
    fn speak(&self) -> String {
        String::from("Woof.")
    }
}

fn main() {
    let person = Person {};
    let dog = Dog {};
    println!("The person says {}", person.speak());
    println!("The dog says {}", dog.speak());
}
```

Envoi statique et dynamique

Il est possible de créer une fonction qui accepte des objets qui implémentent un trait spécifique.

Envoi statique

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
}
```

La répartition statique est utilisée ici, ce qui signifie que le compilateur Rust générera des versions spécialisées de la fonction `generic_speak` pour les types `Dog` et `Person` . Cette génération de versions spécialisées d'une fonction polymorphe (ou de toute entité polymorphe) au cours de la compilation s'appelle **Monomorphization** .

Envoi dynamique

```
fn generic_speak(speaker: &Speak) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person as &Speak);
    generic_speak(&dog); // gets automatically coerced to &Speak
}
```

Ici, une seule version de `generic_speak` existe dans le fichier binaire compilé, et l'appel `speak()` est effectué à l'aide d'une recherche vtable à l'exécution. Ainsi, l'utilisation de la distribution dynamique accélère la compilation et réduit la taille des fichiers binaires compilés, tout en étant légèrement plus lente à l'exécution.

Les objets de type `&Speak` ou `Box<Speak>` sont appelés **objets de trait** .

Types associés

- Utilisez le type associé lorsqu'il existe une relation biunivoque entre le type implémentant le trait et le type associé.
- Il est parfois également appelé *type de sortie* , car il s'agit d'un élément donné à un type lorsque nous lui appliquons un trait.

Création

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                          ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

Mise en œuvre

```
impl<T, U: ?Sized> GetItems for (T, U) {
    type First = T;
    type Last = U;
//              ^~~ assign the associated types
    fn first_item(&self) -> &Self::First { &self.0 }
    fn last_item(&self) -> &Self::Last { &self.1 }
    fn set_first_item(&mut self, item: Self::First) { self.0 = item; }
}

impl<T> GetItems for [T; 3] {
    type First = T;
    type Last = T;
    fn first_item(&self) -> &T { &self[0] }
//                            ^ you could refer to the actual type instead of `Self::First`
    fn last_item(&self) -> &T { &self[2] }
    fn set_first_item(&mut self, item: T) { self[0] = item; }
}
```

Référencement aux types associés

Si nous sommes sûrs qu'un type `T` implémente `GetItems` par exemple dans les génériques, nous pourrions simplement utiliser `T::First` pour obtenir le type associé.

```
fn get_first_and_last<T: GetItems>(obj: &T) -> (&T::First, &T::Last) {
//                                              ^~~~~~~~ refer to an associated type
    (obj.first_item(), obj.last_item())
}
```

Sinon, vous devez indiquer explicitement au compilateur la caractéristique implémentée par le type

```
let array: [u32; 3] = [1, 2, 3];
let first: &<[u32; 3] as GetItems>::First = array.first_item();
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~ [u32; 3] may implement multiple traits which many
//                                        of them provide the `First` associated type.
//                                        thus the explicit "cast" is necessary here.
assert_eq!(*first, 1);
```

Contrainte avec les types associés

```
fn clone_first_and_last<T: GetItems>(obj: &T) -> (T::First, T::Last)
    where T::First: Clone, T::Last: Clone
//  ^~~~ use the `where` clause to constraint associated types by traits
{
    (obj.first_item().clone(), obj.last_item().clone())
}

fn get_first_u32<T: GetItems<First=u32>>(obj: &T) -> u32 {
//                          ^~~~~~~~~~~ constraint associated types by equality
    *obj.first_item()
}
```

Méthodes par défaut

```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```

La méthode sera appelée par défaut sauf si elle est écrasée dans le bloc `impl` .

```
struct Human;
struct Cat;

impl Speak for Human {}

impl Speak for Cat {
```

```
    fn speak(&self) -> String {
        String::from("Meow.")
    }
}

fn main() {
    let human = Human {};
    let cat = Cat {};
    println!("The human says {}", human.speak());
    println!("The cat says {}", cat.speak());
}
```

Sortie:

> L'humain dit bonjour.
>
> Le chat dit Miaou.

Placer une borne sur un trait

Lors de la définition d'un nouveau trait, il est possible d'imposer aux types souhaitant implémenter ce trait de vérifier un certain nombre de contraintes ou de limites.

Prenant un exemple de la bibliothèque standard, le trait `DerefMut` exige qu'un type implémente d'abord son trait de caractère `Deref` :

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Cela permet à `DerefMut` d'utiliser le type de `Target` associé défini par `Deref` .

Bien que la syntaxe puisse rappeler l'héritage:

- il apporte tous les éléments associés (constantes, types, fonctions, ...) du trait lié
- il permet le polymorphisme de `&DerefMut` à `&Deref`

C'est différent dans la nature:

- il est possible d'utiliser une durée de vie (telle que `'static`)
- il n'est pas possible de remplacer les éléments de trait liés (même pas les fonctions)

Il est donc préférable de le considérer comme un concept distinct.

Plusieurs types d'objet lié

Il est également possible d'ajouter plusieurs types d'objet à une fonction de distribution statique .

```
fn mammal_speak<T: Person + Dog>(mammal: &T) {
    println!("{0}", mammal.speak());
}
```

```
fn main() {
    let person = Person {};
    let dog = Dog {};

    mammal_speak(&person);
    mammal_speak(&dog);
}
```

Lire Traits en ligne: https://riptutorial.com/fr/rust/topic/1313/traits

Chapitre 47: Traits de conversion

Remarques

- `AsRef` et `Borrow` sont similaires mais servent des objectifs distincts. `Borrow` est utilisé pour traiter plusieurs méthodes d'emprunt de la même manière, ou pour traiter les valeurs empruntées comme leurs homologues possédés, alors que `AsRef` est utilisé pour la généralisation de références.
- `From<A> for B` implique `Into<B> for A` , mais pas l'inverse.
- `From<A> for A` est implicitement implémenté.

Examples

De

Le trait Rust's `From` est un trait général pour la conversion entre les types. Pour deux types `TypeA` et `TypeB` ,

```
impl From<TypeA> for TypeB
```

indique qu'une instance de `TypeB` est *garantie* d'être constructible à partir d'une instance de `TypeA` . Une implémentation de `From` ressemble à ceci:

```
struct TypeA {
    a: u32,
}

struct TypeB {
    b: u32,
}

impl From<TypeA> for TypeB {
    fn from(src: TypeA) -> Self {
        TypeB {
            b: src.a,
        }
    }
}
```

AsRef & AsMut

`std::convert::AsRef` et `std::convert::AsMut` sont utilisés pour convertir les types en références à moindre coût. Pour les types `A` et `B` ,

```
impl AsRef<B> for A
```

indique que a `&A` peut être converti en a `&B` et,

```
impl AsMut<B> for A
```

indique que a `&mut A` peut être converti en `&mut B`

Ceci est utile pour effectuer des conversions de type sans copier ou déplacer des valeurs. Un exemple dans la bibliothèque standard est `std::fs::File.open()` :

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

Cela permet à `File.open()` d'accepter non seulement le `Path` , mais aussi `OsStr` , `OsString` , `str` , `String` et `PathBuf` avec une conversion implicite car ces types implémentent tous `AsRef<Path>` .

Emprunter, emprunterMut et ToOweded

Les traits `std::borrow::BorrowMut` `std::borrow::Borrow` et `std::borrow::BorrowMut` sont utilisés pour traiter les types empruntés comme les types possédés. Pour les types `A` et `B` ,

```
impl Borrow<B> for A
```

indique qu'un `A` emprunté peut être utilisé lorsqu'un `B` est souhaité. Par exemple, `std::collections::HashMap.get()` utilise `Borrow` pour sa méthode `get()` , permettant à `HashMap` avec les clés de `A` d'être indexées avec a `&B`

En revanche, `std::borrow::ToOwned` implémente la relation inverse.

Ainsi, avec les types susmentionnés `A` et `B` on peut implémenter:

```
impl ToOwned for B
```

Remarque: alors que `A` peut implémenter `Borrow<T>` pour plusieurs types distincts `T` , `B` ne peut implémenter que `ToOwned` une fois.

Deref & DerefMut

Les traits `std::ops::Deref` et `std::ops::DerefMut` sont utilisés pour surcharger l'opérateur de déréférencement, `*x` . Pour les types `A` et `B` ,

```
impl Deref<Target=B> for A
```

indique que le déréférencement d'une liaison de `&A` donnera un `&B` et,

```
impl DerefMut for A
```

indique que le déréférencement d'une liaison de `&mut A` donnera un `&mut B`

`Deref` (resp. `DerefMut`) fournit également une fonction de langage utile appelée *deref coercion* , qui

permet à a `&A` (resp. `&mut A`) de forcer automatiquement a `&B` (resp. `&mut B`). Ceci est couramment utilisé lors de la conversion de `String` en `&str` , as `&String` étant implicitement contraint à `&str` si nécessaire.

Remarque: `DerefMut` ne prend pas en charge la spécification du type résultant, il utilise le même type que `Deref` .

Remarque: En raison de l'utilisation d'un type associé (contrairement à `AsRef`), un type donné ne peut implémenter que `Deref` et `DerefMut` au plus une fois.

Lire Traits de conversion en ligne: https://riptutorial.com/fr/rust/topic/2661/traits-de-conversion

Chapitre 48: Tuples

Introduction

La structure de données la plus triviale, après une valeur singulière, est le tuple.

Syntaxe

- (A, B, C) // un triplet (un tuple à trois éléments), dont le premier élément a le type A, le deuxième type B et le troisième type C
- (A, B) // un deux tuple, dont les deux éléments ont respectivement les types A et B
- (A,) // un tuple (notez la fin `,`), qui ne contient qu'un seul élément de type A
- () // le tuple vide, qui est à la fois un type et le seul élément de ce type

Examples

Types de tuple et valeurs de tuple

Les tuples de rouille, comme dans la plupart des autres langues, sont des listes de taille fixe dont les éléments peuvent tous être de types différents.

```
// Tuples in Rust are comma-separated values or types enclosed in parentheses.
let _ = ("hello", 42, true);
// The type of a tuple value is a type tuple with the same number of elements.
// Each element in the type tuple is the type of the corresponding value element.
let _: (i32, bool) = (42, true);
// Tuples can have any number of elements, including one ..
let _: (bool,) = (true,);
// .. or even zero!
let _: () = ();
// this last type has only one possible value, the empty tuple ()
// this is also the type (and value) of the return value of functions
// that do not return a value ..
let _: () = println!("hello");
// .. or of expressions with no value.
let mut a = 0;
let _: () = if true { a += 1; };
```

Correspondance des valeurs de tuple

Programmes de rouille utilisent modèle correspondant largement à déconstruire les valeurs, que ce soit en utilisant `match` , `if let` , ou déconstruisant `let` des motifs. Les tuples peuvent être déconstruits comme vous vous en doutez avec `match`

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
```

```
        _ => println!("it's something else"),
    }
}
```

ou avec `if let`

```
fn foo(x: (&str, isize, bool)) {
    if let (_, 42, _) = x {
        println!("it's 42");
    } else {
        println!("it's something else");
    }
}
```

vous pouvez également lier à l'intérieur du tuple en utilisant `let` -deconstruction

```
fn foo(x: (&str, isize, bool)) {
    let (_, n, _) = x;
    println!("the number is {}", n);
}
```

Regarder à l'intérieur des tuples

Pour accéder aux éléments d'un tuple directement, vous pouvez utiliser le format `.n` pour accéder au `n` élément -ème

```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```

Vous pouvez également sortir partiellement d'un tuple

```
let x = (String::from("hello"), 42);
let (s, _) = x;
let (_, n) = x;
println!("{}, {}", s, n);
// the following would fail however, since x.0 has already been moved
// let foo = x.0;
```

Les bases

Un tuple est simplement une concaténation de plusieurs valeurs:

- de types éventuellement différents
- dont le nombre et les types sont connus statiquement

Par exemple, `(1, "Hello")` est un tuple à 2 éléments composé d'un `i32` et d'un `&str` , et son type est noté `(i32, &'static str)` de manière similaire à sa valeur.

Pour accéder à un élément d'un tuple, on utilise simplement son index:

```
let tuple =  (1, "Hello");
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```

Comme le tuple est intégré, il est également possible d'utiliser une correspondance de motif sur les n-uplets:

```
match (1, "Hello") {
    (i, _) if i < 0 => println!("Negative integer: {}", i),
    (_, s) => println!("{} World", s),
}
```

Cas spéciaux

L'élément 0 tuple: `()` est également appelé *unité* , *type d'unité* ou *type singleton* et sert à indiquer l'absence de valeurs significatives. C'est le type de retour par défaut des fonctions (lorsque `->` n'est pas spécifié). *Voir aussi: Quel type est le "type ()" dans Rust?* .

Le tuple à 1 élément: `(a,)` , avec la virgule finale, dénote un tuple à 1 élément. La forme sans virgule `(a)` est interprétée comme une expression entre parenthèses et est évaluée à seulement `a` .

Et pendant que nous y sommes, les virgules sont toujours acceptées: `(1, "Hello",)` .

Limites

Le langage Rust ne supporte pas les *variads* , à part les tuples. Par conséquent, il n'est pas possible d'implémenter simplement un trait pour tous les tuples et, par conséquent, les traits standard ne sont implémentés que pour un nombre limité d'éléments (aujourd'hui, jusqu'à 12 inclus). Les tuples avec plus d'éléments sont supportés, mais n'implémentent pas les traits standard (bien que vous puissiez implémenter vos propres traits).

Nous espérons que cette restriction sera levée à l'avenir.

Déballage des tuples

```
// It's possible to unpack tuples to assign their inner values to variables
let tup = (0, 1, 2);
// Unpack the tuple into variables a, b, and c
let (a, b, c) = tup;

assert_eq!(a, 0);
assert_eq!(b, 1);

// This works for nested data structures and other complex data types
let complex = ((1, 2), 3, Some(0));

let (a, b, c) = complex;
let (aa, ab) = a;

assert_eq!(aa, 1);
assert_eq!(ab, 2);
```

Lire Tuples en ligne: https://riptutorial.com/fr/rust/topic/3941/tuples

Chapitre 49: Types de données primitifs

Examples

Types scalaires

Entiers

Signé: `i8` , `i16` , `i32` , `i64` , `isize`

Non signé : `u8` , `u16` , `u32` , `u64` , `usize`

Le type d'un littéral entier, disons `45` , sera automatiquement déduit du contexte. Mais pour le forcer, nous ajoutons un suffixe: `45u8` (sans espace) sera tapé `u8` .

Note: La taille de l' `isize` et de l' `usize` dépendent de l'architecture. Sur 32-bit arch, c'est 32 bits, et sur 64 bits, vous l'avez deviné!

Points flottants

`f32` et `f64` .

Si vous écrivez simplement `2.0` , il s'agit de `f64` par défaut, à moins que l'inférence de type n'en décide autrement!

Pour forcer `f32` , définissez la variable avec le type `f32` , ou le suffixe le littéral: `2.0f32` .

Booléens

`bool` , ayant des valeurs `true` et `false` .

Personnages

`char` , avec les valeurs écrites comme `'x'` . Dans les guillemets simples, contient une seule valeur scalaire Unicode, ce qui signifie qu'il est valide d'avoir un emoji! Voici 3 autres exemples: `'□'` , `'\u{3f}'` , `'\u{1d160}'` .

Lire Types de données primitifs en ligne: https://riptutorial.com/fr/rust/topic/8705/types-de-donnees-primitifs

Chapitre 50: Valeurs en boîte

Introduction

Les boîtes sont une partie très importante de Rust, et chaque paysan doit savoir ce qu'il est et comment l'utiliser.

Examples

Créer une boîte

Dans Rust stable, vous créez une Box en utilisant la fonction `Box::new` .

```
let boxed_int: Box<i32> = Box::new(1);
```

Utiliser les valeurs en boîte

Comme les boîtes implémentent le `Deref<Target=T>` , vous pouvez utiliser des valeurs encadrées comme la valeur qu'elles contiennent.

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

Si vous souhaitez créer un motif sur une valeur encadrée, vous devrez peut-être déréférencer la boîte manuellement.

```
struct Point {
    x: i32,
    y: i32,
}

let boxed_point = Box::new(Point { x: 0, y: 0});
// Notice the *. That dereferences the boxed value into just the value
match *boxed_point {
    Point {x, y} => println!("Point is at ({}, {})", x, y),
}
```

Utiliser des boîtes pour créer des énumérations et des structures récursives

Si vous essayez de créer une énumération récursive dans Rust sans utiliser Box, vous obtiendrez une erreur de compilation indiquant que l'énumération ne peut pas être dimensionnée.

```
// This gives an error!
enum List {
    Nil,
    Cons(i32, List)
}
```

Pour que l'énum ait une taille définie, la valeur contenue dans la récursivité doit être dans une boîte.

```
// This works!
enum List {
    Nil,
    Cons(i32, Box<List>)
}
```

Cela fonctionne parce que Box a toujours la même taille quel que soit T, ce qui permet à Rust de donner une taille à List.

Lire Valeurs en boîte en ligne: https://riptutorial.com/fr/rust/topic/9341/valeurs-en-boite

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Rust	Andy Hayden, ar-ms, Aurora0001, Community, Cormac O'Brien, D. Ataro, David Grayson, Eric Platon, gavinb, IceyEC, John, Jon Gjengset, Kellen, kennytm, Kevin Montrose, Lukabot, mmstick, Neikos, Pavel Strakhov, Shepmaster, Timidger, Tot Zam, Tshepang, Wolf, xfix, Yohaï Berreby
2	Applications GUI	eddy, vaartis
3	Arguments de ligne de commande	Aurora0001
4	Assemblage en ligne	4444, Aurora0001
5	Boucles	Andy Hayden, apopiak, Arrem, Aurora0001, JDemler, John, kennytm, Mario Carneiro, Matt Smith, Matthieu M., mcarton, Sanpi, Shepmaster, Timidger, Winger Sendon, YOU
6	Cadre Web Iron	4444, Aurora0001, Phil J. Laszkowicz
7	Cargaison	Arrem, Aurora0001, Bo Lu, Charlie Egan, Cormac O'Brien, David Grayson, Enigma, John, Lukas Kalbertodt
8	Constantes Associées	Ameo, Aurora0001, Hauleth
9	Cordes	Arrem, Aurora0001, David Grayson, KokaKiwi, Lukas Kalbertodt , mcarton, rap-2-h, tmr232, Yos Riady
10	Correspondance de motif	adelarsq, Andy Hayden, aSpex, Aurora0001, Cormac O'Brien, Lukas Kalbertodt, mcarton, mnoronha, xea
11	Déréférencement automatique	Aurora0001, John, kennytm, Kornel, Timidger, vaartis, Winger Sendon
12	Dérivé personnalisé: "Macros 1.1"	Vi.
13	Des tests	Aurora0001, Cormac O'Brien, IceyEC, JDemler, Jean Pierre Dudey, kennytm, Lu.nemec, mcarton
14	Des vies	antoyo, Cormac O'Brien, Jean Pierre Dudey, letmutx, xetra11
15	Directives	Aurora0001, John

	dangereuses	
16	Documentation	Aurora0001, Cormac O'Brien, Hauleth
17	Fermetures et expressions lambda	xea
18	Fichier I / O	antoyo, Kornel
19	Futures et Async IO	KolesnichenkoDS
20	Génération de nombres aléatoires	Phil J. Laszkowicz
21	Génériques	Kornel, xea
22	Globals	Cormac O'Brien, Jean Pierre Dudey, John, kennytm, Leo Tindall , mcarton
23	Guide de style rouille	Aurora0001, Cldfire, James Gilles, tversteeg
24	Interface de fonction étrangère (FFI)	Aurora0001, John, Konstantin V. Salikhov
25	La gestion des erreurs	Cormac O'Brien, John, kennytm, Winger Sendon, xea
26	La possession	Aurora0001, Jon Gjengset
27	Les itérateurs	Aurora0001, Chris Emerson, Hauleth, John, Lukas Kalbertodt, Matt Smith, Shepmaster
28	Les structures	4444, Jon Gjengset, letmutx
29	Macros	Aurora0001, kennytm, Matt Smith
30	Mise en réseau TCP	E_net4
31	Modules	Aurora0001, Cormac O'Brien, Dumindu Madunuwan, John, KokaKiwi, Kornel, Lu.nemec, xetra11
32	Opérateurs et surcharge	Aurora0001, John, Matthieu M.
33	Option	antoyo, Arrem, Aurora0001, fxlae, Kornel, letmutx, mcarton, Shepmaster
34	Panique et Déroulement	Aurora0001, Leo Tindall, Timidger
35	Parallélisme	Aurora0001, John, Ruud, xea, zrneely

36	PhantomData	Neikos
37	Pointeurs bruts	xea
38	Regex	Aurora0001, vaartis
39	Rouille nue	John, mmstick, SplittyDev
40	Rouille orientée objet	adelarsq, Aurora0001, Leo Tindall, Marco Alka, Matthieu M., s3rvac, Sorona, Timidger
41	rouiller	torkleyy
42	Serde	Aurora0001, dtolnay, kennytm
43	Tableaux, vecteurs et tranches	antoyo, Aurora0001, John, Matthieu M.
44	The Drop Trait - Destructeurs à Rust	Leo Tindall, Neikos
45	Traitement du signal	Aurora0001, Jean Pierre Dudey, mmstick
46	Traits	a10y, adelarsq, Arrem, Aurora0001, Cormac O'Brien, Hauleth, John, kennytm, Leo Tindall, Matt Smith, Matthieu M., Mylainos, RamenChef, SplittyDev, tversteeg, xea
47	Traits de conversion	Cormac O'Brien, Matthieu M.
48	Tuples	adelarsq, Ameo, Aurora0001, John, Jon Gjengset, Matthieu M.
49	Types de données primitifs	John
50	Valeurs en boîte	BookOwl