SLH - Sécurité logicielle haut niveau

Rust

24 octobre 2025

1 Traitement d'Erreurs

1.1 Result< T,E >

Le type `Result` représente une opération qui peut échouer:

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

Utilisation basique avec `match`:

```
fn do_that(&self) -> Result<Foo,
FooException> {
  if self.broken {
    return
Err(FooException::EverythingIsBroken)
  }
  Ok(foo)
}
```

1.2 Opérateur ?

Sucre syntaxique pour chaîner les opérations qui peuvent échouer:

```
let x = expr_that_can_fail?;
```

Équivalent à:

```
let x = match expr_that_can_fail {
  Err(e) => return Err(e.into()),
  Ok(r) => r,
};
```

Permet le chaînage fluide:

```
fn do_something(&self) -> Result<(),
FooException> {
self.do_that()?.and_this()?.but_also_that();
  Ok(())
}
```

1.3 Option

Remplace `null` pour marquer l'absence de valeur:

```
enum Option<T> {
  None,
  Some(T),
}
```

Méthodes utiles:

- `o.is_some()` / `o.is_none()`
- `o.map(|x| { ... })`
- `o.unwrap_or(default)`
- `o.expect("message")`

1.4 If-let et Let-else

Déstructuration partielle:

```
if let Some(x) = maybe_x {
  // using x
}

let Some(x) = maybe_x else {
  eprintln!("error");
  return 42
}
```

1.5 Librairies d'Erreurs

Pour les applications: `anyhow` ou `color-eyre`

- Type d'erreur unifié
- Conversion automatique

Pour les librairies: `thiserror`

- Automatise la création d'enums d'erreurs

2 Types de Données

2.1 Structures

Trois styles de structures:

```
// Named-field
struct Circle {
  x: f64,
  y: f64,
  radius: f64,
}

// Tuple-struct
struct Circle(f64, f64, f64);

// Tuple (anonyme)
let c = (0.5, 0.0, 1.0);
```

Accès: `c.x`, `c.0`

2.2 Représentation Mémoire

Le compilateur réordonne et aligne les champs pour optimiser. Par défaut, la représentation n'est pas garantie.

Newtypes: Une struct avec un seul champ a la même représentation que son champ - excellent pour imposer des invariants:

```
struct String { vec: Vec<u8> }
```

2.3 Enums

Collection de variants:

```
enum Message {
  Quit,
  Move { x: i32, y: i32 },
  Print(String),
}
```

2.4 Pattern Matching

Déconstruction exhaustive:

```
match msg {
  Message::Quit => { ... },
  Message::Move { x, y } => { ... },
  Message::Print(text) => { ... },
}
```

Le compilateur exige l'exhaustivité. Captures par référence:

```
match &msg {
  Message::Print(text) => { ... } // text:
&String
}
```

3 Méthodes

Syntaxe spéciale pour dispatch statique sur une variable:

```
impl Rectangle {
  fn area(&self) -> u32 {
    self.height * self.width
  }
}

let r = Rectangle { height: 20, width: 30 };
r.area()
```

Plusieurs blocs `impl` peuvent exister pour le même type. Le type `Self` alias le type implémenté.

Paramètres: `self` (prise de propriété), `&self` (emprunt), `&mut self` (emprunt mutable).

4 Allocation & Sûreté Mémoire

4.1 Invariants Fondamentaux

Propriété: Il existe exactement un chemin propriétaire de chaque ressource.

- Une valeur est copiée ou déplacée selon son type
- La ressource est désallouée quand le propriétaire disparaît

Exclusivité: S'il existe un chemin mutable, il n'existe aucun autre chemin actif.

- Élimine les effets de bord inattendus

4.2 Copie vs Déplacement

```
let x: i32 = 42;
let y = foo(x); // i32 est copié; x reste
actif

let x: Box<i32> = Box::new(42);
let y = foo(x); // Box est déplacé; x est
inutilisable
```

4.3 Borrows (Références)

```
fn foo(x: &mut i32) { ... }
let x: Box<i32> = Box::new(42);
foo(&mut *x); // Conversion explicite
foo(&mut x);  // Autoderef appliqué
automatiquement
```

Mode facile Rust:

- Pas de références en valeur de retour
- Pas de références dans des structs
- Clone si nécessaire

5 Lifetimes

5.1 Problème des Pointeurs

Le type d'une valeur ne change pas avec le temps. Mais avec les pointeurs, c'est faux:

```
let r;
{
  let x: usize = ...;
  r = &x; // x disparaît à la fin du bloc
}
println!("{}", *r); // r pointe sur une
donnée libérée!
```

5.2 Notation des Lifetimes

Intègre au système de types la notion de temps pour les références:

```
// C
size_t * const ptr;

// Rust
ptr: &'p mut usize
```

Fonctionne comme un générique: `&'p usize ≈ Ref<'p, usize>`

5.3 Règles Clés

- Une référence ne peut pas sortir du bloc contenant la variable référée
- Pour `&x` : `x` doit survivre à la lifetime, `x` perd `mut`
- Pour `&mut x` : `x` doit être `mut` , `x` est inutilisable pendant la lifetime

5.4 Exemple: strip_prefix()

```
fn strip_prefix<'s>(s: &'s str, prefix:
&str) -> Option<&'s str> { ... }
```

Le type de retour est lié à la durée de vie du premier argument.

5.5 Structures Génériques

```
enum Json<'s> {
  Int(i64),
  String(&'s str),
  Array(Vec<Json<'s>>),
}
```

5.6 Élision des Lifetimes

Peuvent être omis quand:

- Sur une variable (toujours inféré)
- Type de retour avec un seul lifetime générique
- Méthode prenant `&self` ou `&mut self`

6 Closures

6.1 Traits Fn

Hiérarchie basée sur comment on utilise l'environnement:

```
fn : Fn : FnMut : FnOnce
```

- `Fn(&self)`: Pas d'utilisation mut de l'environnement
- `FnMut(&mut self)`: Pas de consommation de l'environnement
- `FnOnce(self)`: Peut être appelée qu'une seule fois

6.2 Closures en Arguments

Chaque closure est un type unique:

```
fn map<B, F>(self, f: F) -> impl
Iterator<Item=B>
where F: FnMut(Self::Item) -> B
```

6.3 Closures move

`move` transfère la propriété de l'environnement vers la closure (pendant sa construction):

```
let env: Environment = { ... };
let tx = tx.clone();
thread::spawn(move || { do_some_work(env,
tx) })
```

7 Concurrence

7.1 Threads

```
use std::thread;
let handle = thread::spawn(|| {
  for i in (1..100) {
    println!("thread 1 = {i}");
  }
});
handle.join()?;
```

Signature:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
  F: FnOnce() -> T + Send + 'static,
  T: Send + 'static,
```

- `F` est `FnOnce` car le thread démarre une seule fois
- `F` et `T` doivent être `'static` (pas de références non-"static)

7.2 Send & Sync

- **Send:** Peut être déplacé sans danger d'un thread à un autre
- **Sync:** Peut être accédé sans danger depuis plusieurs threads simultanément
  - `T: Sync` ⟺ `&T: Send`

7.3 Arc & Mutex

`std::sync::Arc`: Atomic Reference Counter

```
pub fn new(data: T) -> Arc<T>;
pub fn clone(&self) -> Self; // Clone l'Arc,
pas les données
pub fn deref(&self) -> &T;
```

`std::sync::Mutex`: Mutual Exclusion

```
pub const fn new(t: T) -> Mutex<T>;
pub fn lock(&self) ->
LockResult<MutexGuard<'_,T>>;
```

7.4 Message Passing

`std::sync::mpsc`: Multiple-Producer Single-Consumer

```
pub fn channel<T>() -> (Sender<T>,
Receiver<T>);
pub fn Sender::send(&self, t:T) ->
Result<(),SendError<T>>;
pub fn Receiver::recv(&self) -> Result<T,
RecvError>;
```

- `Sender` est `Clone` (Arc en interne)
- `Receiver` n'est pas `Sync`

8 Modules & Crates

8.1 Crates vs Modules

- **Crate:** Unité de compilation

- **Module:** Organisation du code, crée des namespaces

8.2 Versionage Sémantique

`MAJOR.MINOR.PATCH` :

- **Majeur:** Changements incompatibles
- **Mineur:** Nouvelles fonctionnalités (compatibles)
- **Patch:** Corrections sans changement d'API

Dépendance `X.Y` = `(MAJOR = X) ∧ (MINOR ≥ Y)`

8.3 Modules

En ligne:

```
mod tests {
  fn foo() { ... }
}
```

Ou dans un fichier:

```
mod tests; // tests.rs ou tests/mod.rs
```

8.4 Visibilité

- **Privé par défaut:** Accessible depuis le module et ses enfants
- `pub` : Accessible de partout
- `pub(crate)` : Accessible dans toute la crate

`use` pour importer dans le scope:

```
use std::io::Result;
use std::{io, cmp::Ordering};
use std::io::Result as IOR;
```

9 Traits

9.1 Définition

Collection de méthodes abstraites et concrètes:

```
trait Vehicle {
  fn drive(&mut self);
  fn honk(&self) { println!("Bip bip"); }
}
```

9.2 Implémentation

```
struct Car { position: u8 }
impl Vehicle for Car {
  fn drive(&mut self) { ... }
}
```

9.3 Traits comme Marqueurs

Certains traits n'ont pas de méthodes, seulement un rôle de marqueur:

- `Copy` : Type peut être copié (shallow)
- `Sized` : Taille connue à la compilation
- `Send` : Thread-safe pour être envoyé entre threads
- `Sync` : Thread-safe pour être partagé entre threads

10 Génériques

10.1 Syntaxe

```
fn hello<V: Vehicle>(v: V) { ... }
struct Container<T> { data: T }
trait Motor<D: Driver> { ... }
```

Contrainte par un ou plusieurs traits.

10.2 Traits Génériques

```
trait From<T>: Sized {
  fn from(value: T) -> Self;
}

impl From<&str> for String { ... }
impl<T> From<T> for Box<T> { ... }
```

10.3 Types Unitaires pour Statique

Une struct sans champs existe seulement à la compilation:

```
struct Public;
struct Secret;

struct Actor<S> {
  access: S,
}

impl Actor<Secret> {
  fn write_confidential(self) ->
Actor<Public> { ... }
}
```

10.4 Types Fantômes

Un générique non utilisé par la structure:

```
struct Key<T: RedisType>(String,
PhantomData<T>);
```

10.5 Iterator

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  fn count(self) -> usize { ... }
  fn zip<U: Iterator>(self, other: U) ->
Zip<Self,U> { ... }
}
```

10.6 Types Associés

Types choisis en fonction de l'implémentation:

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
}
```

vs générique où type est choisi à l'appel.

10.7 Dispatch Dynamique

`dyn Vehicle` représente une implémentation dynamique du trait:

```
let v: &dyn Vehicle = &car;
let v: Box<dyn Vehicle> = Box::new(car);
```

Rarement utilisé directement, fonctionne via référence/pointeur.

10.8 Résolution de Conflits

Deux traits avec la même méthode? Solution Rust avec namespaces:

```
<Car as Vehicle>::drive(&car)
```

10.9 Règle des Orphelins

Une implémentation de trait `T for Type` doit avoir:

- `T` défini dans cette crate, OU
- `Type` défini dans cette crate

Évite l'« action à distance ».

10.10 impl dans les Signatures

Sucre pour générique implicite en argument:

```
fn f(&self, drivable: &impl Vehicle) { ... }
// Équivalent à:
fn f<V: Vehicle>(&self, vehicle: &V) { ... }
```

En retour, rend le type opaque (existe vraiment un type unique):

```
fn f(&self) -> impl Vehicle { ... }
```

11 Tableaux, Vecteurs & Slices

11.1 Tableaux

Taille constante, éléments contigus:

```
let foo: [usize; 4] = [1, 3, 2, 4];
let bar: [i32; 20] = [-1; 20];
```

Accès: `foo[1]`, `foo.get(5)` retourne `Option`

11.2 Vecteurs

Taille variable, allouée sur le heap:

```
let foo: Vec<usize> = vec![1, 3, 2, 4];
foo.push(42);
let x = foo.pop(); // Option
```

Représentation: 3 mots (pointeur, longueur, capacité)

11.3 Slices

Suite contigue d'éléments, longueur connue en runtime:

```
&[T]     // Fat pointer (ptr + len)
&mut [T] // Fat pointer mutable
```

Construction:

```
let xs = [0,1,2,3,4];
let slice = &xs[1..3];    // &[1, 2]
let slice = &xs[2..];     // &[2,3,4]
```

11.4 Bonnes Pratiques

- Jamais `&Vec<T>` en argument; utiliser `&[T]`
- `&mut Vec<T>` est ok, différent de `&mut [T]`

12 Strings

12.1 Types String

- `str` : Équivalent à `[u8]`, doit être UTF-8 valide
- `&str` : Référence, immutable
- `String` : Équivalent à `Vec<u8>`
- `char` : Point de code UTF-8 (32 bits)

Pas de terminateur NUL.

12.2 Littéraux

Stockés dans `.data`, immutables, type `&str` :

```
const HELLO: &str = "Hello World!";
const QUOTE: &str = r#"we "quote" hehe"#;
const BYTES: &[u8] = b"\xE9";
```

12.3 Allocation

Toujours sur la heap:

```
let x = String::from(HELLO);
let x = HELLO.to_string();
let x = HELLO.to_owned();
let x: String = HELLO.into();
```

12.4 Concaténation

```
let mut buf: String = "abcd".into();
buf += "efgh";
buf = format!("{}{}", buf, "efgh");
```

12.5 Méthodes Utiles

- `.len()` : Longueur EN BYTES
- `.chars().count()` : Nombre de points de code (O(n))
- `.as_bytes()` : &str → &[u8]
- `.trim()`, `.split()`, ...

Pas d'indexation directe (à cause d'UTF-8).

13 Tests & Documentation

13.1 Tests Unitaires

Attribut `#[test]` pour les fonctions de test:

```
#[test]
fn my_test() {
  assert_eq!(42, answer());
}
```

Exécution: `cargo test`

`#[cfg(test)]` pour code exclusif aux tests.

Tests d'intégration dans le répertoire `tests/`.

13.2 Doc-comments

Commentaires de documentation:

```
/// Une matrice n-by-n
///
/// Elle utilise un [`Vec`] en interne.
struct Matrix { ... }
```

Génération avec `cargo doc`.

13.3 Doctests

Exemples dans la documentation qui sont aussi testés:

```
///
/// let a = identity(2);
/// assert_eq!(matsum(a), Ok(...))
///
pub fn matsum(a: &Matrix) { ... }
```