ULB UNIVERSITÉ LIBRE DE BRUXELLES

MLG MACHINE LEARNING GROUP

En quoi apprendre Rust fait de vous un·e meilleur·e dev dans d'autres languages

Smart Mondays

Yannick Molinghen

ULB – DI

14 Octobre 2024

ULB Faculté des Sciences

Parlons de Rust

- Débuté à Mozzilla entre 2006 et 2009
- Compilé
- Typage fort et statique
- Axé sur
  - la sécurité
  - la rapidité

Rust est un langage *sécurisé*

- Pas de `malloc` ni de `free`
- Gestion obligatoire et explicite des erreurs
- Pattern matching exhaustif des énumerations
- Ownership et borrowing
- `Option<T>`

Rust n'est pas null

La `NullPointerException` est probablement l'exception la plus connue en Java.

```
User user = null;   // Valid Java
user.greet();       // NullPointerException
```

En Rust, on utilise le type générique `Option<T>` pour représenter un type qui peut être null.

```
let user1 = None;
user1.greet(); // Ne compile pas
let user2 = Some(User{name:"Yannick"});
if let Some(u) = user2 {
  u.greet();
}
```

Conséquences dans d'autres langages

Certains langages utilisent une syntaxe similaire à Rust. Java a introduit le type `Optional<T>` qui nécessite aussi d'être vérifié. Exemple en Python et utilité du type hinting.

Des objets complets


```
class User:
        age: int
        name: str

        def __init__(self, name):
                self.name = name

u = User("Yannick")
print(u.age)
```

Des objets complets


```
class User:
        age: int
        name: str

        def __init__(self, name):
                self.name = name

u = User("Yannick")
print(u.age)
```

Est-ce possible en Rust ?

Des objets complets


```
class User:
        age: int
        name: str

        def __init__(self, name):
                self.name = name

u = User("Yannick")
print(u.age)
```

Est-ce possible en Rust ?

Non, grace à son typage fort, Rust garantit que chaque attribut d'une structure correspond bel et bien au type indiqué.

Ownership et mutabilité (1)

En regardant ce code Java, que peut-on dire de la variable address donnée en paramètre ?

```
void setAddress(Address address) {
  this.address = address;
}
```

Ownership et mutabilité (1)

En regardant ce code Java, que peut-on dire de la variable address donnée en paramètre ?

```
void setAddress(Address address) {
  this.address = address;
}
```

- La méthode ne précise pas si la variable address est exclusivement utilisée par this ou si d'autres objets y ont accès.
- Si d'autres objets y ont accès, a-t-on le droit de modifier cette variable ?

Valeur ou référence ?

En Python, Java ou JavaScript, (presque) tout se passe par référence.

```python
# Python
def change_name(name: str):
    self.name = name
```

```java
// Java
void setAddress(Address addr) {
    this.address = addr;
}
```

Et rien ne montre de l'extérieur que ces fonctions modifient le contenu de l'utilisateur.

Ownership et mutabilité (2)

En Rust, c'est explicite et vérifié à la compilation.

```
change_address1(&mut self, address: &Address) {
    self.address = address.clone();
}

change_address2(&mut self, address: Address) {
    self.address = address;
}

change_address3(&mut self, address: &mut Address) {
    address.postcode = 1050;
    self.address = address.clone();
}
```

Parallélisme

La modification concurrente d'une variable partagée est un problème récurrent dans le cadre du multi-threading (et du parallélisme plus généralement) ?

Parallélisme

La modification concurrente d'une variable partagée est un problème récurrent dans le cadre du multi-threading (et du parallélisme plus généralement) ?

En quoi ce problème est-il lié à l'ownership ?

Parallélisme

La modification concurrente d'une variable partagée est un problème récurrent dans le cadre du multi-threading (et du parallélisme plus généralement) ?

En quoi ce problème est-il lié à l'ownership ?

Si on donne l'ownership de plusieurs variables à plusieurs threads, alors chacun sait que lui seul possède la variable, et il ne peut pas y avoir de *race condition*.

Parallélisme (2)

Et si on veut quand-même modifier une variable partagée ?
Alors on utilise le pattern d'*interior mutability* avec une `Mutex<T>`.

Interior mutability

C'est un principe qui va permettre de modifier une variable interne sans montrer au compilateur qu'une modification a lieu. `Mutex<T>` utilise ce pattern.

Itérateurs parallèles

Rust propose beaucoup de fonctions sur les itérateurs.

Itérateurs parallèles

Comme on sait ce qui appartient à qui, on sait aussi ce qu'on peut paralléliser !

Itérateurs parallèles

Rust propose beaucoup de fonctions sur les itérateurs.

Itérateurs parallèles

Comme on sait ce qui appartient à qui, on sait aussi ce qu'on peut paralléliser !

Ce dont on n'a pas encore parlé...

- La puissance des `enum` et le pattern matching
- L'absence d'héritage
- La gestion des erreurs
- interopérabilité avec d'autres langages
- `cargo`
- Les autres projets (deno2, ruff, uv, ...) qui exploitent les capacités de Rust
- La courbe d'apprentissage (oups !)

Conclusions

Rust vous contraint à respecter de nombreuses règles qui sont condiérées comme des *bonnes pratiques* ou des *edge cases* dans d'autres langages

- typage en Python
- clone ou pas clone ?
- muable ou immuable ?
- `free` ou pas `free` ?