Rust : Gestion des valeurs optionnelles et des erreurs

Qualité de Développement — R5.A.08

C. Raïevsky

2023-2024

IUT A Valence
Université Grenoble Alpes

Département Informatique

BUT Informatique 3ème année

Gestion explicite

Un code de qualité gère explicitement les valeurs optionnelles et les erreurs

Une fonction peut :

- Ne pas être applicable → retour optionnel
- Échouer en cas d'erreur → propagation de l'erreur

Valeurs optionnelles

Une fonction doit pouvoir ne rien retourner :

- Recherche d'un élément absent
- Paramètres invalides (division par 0)

Dans de nombreux langages : `null`, `nullptr`, `NULL`

$\rightarrow$ gestion implicite, risquée

- null pointer exception...

En Rust :

Le type `Option`

Le type `Option`

Une variable de type `Option<T>` peut prendre deux valeurs :

- `Some(T)`
- `None`

```
1 let mut maybe_bool : Option<bool> = Some(true);
2 maybe_bool = None;
3
4 let mut maybe_int : Option<i32> = Some(42);
5 maybe_int = None;
```

Utilisation d'une Option

Le type `Option` s'utilise entre autres pour :

- le type de retour d'une fonction
- un champ optionnel d'un struct

```rust
fn max(list: Vec<i32>) -> Option<i32> { ... }

struct Person {
    age: u32,
    address: String,
    tik_tok: Option<String>,
}
```

Utilisation d'une Option - Destructuration

Comment accéder à la valeur "emballée" (*wrapped*) dans une Option ?

Destructuration d'une Option

- `unwrap`
- `match`
- `if let`
- `while let`
- opérateur "?"

Option Destructuration - *unwrap*

`une_option.unwrap()`

- Retourne la valeur emballée si `une_option` est `Some`
- Panique (plante) sinon
- À éviter

Option Destructuration - *match*

match

- Permet de tester l'existence d'une valeur
- Permet de faire un test sur l'éventuelle valeur
- Garantit le traitement de tous les cas

```
1 let string_option = Some("Maybe");
2 match string_option {
3     Some("Sure") => println!("You're sure!"),
4     Some("Maybe") => println!("Not so sure..."),
5     Some(_) => println!("You don't know."),
6     None => println!("Nothing to know."),
7 }
```

Option Destructuration - *if let*

Exécution conditionnelle à une destructuration particulière

- Test du type, `Some` ou `None`
- Test sur la valeur
- Possibilité de mettre un `else`

```
1  let option_string = Some("Maybe");
2  if let Some(s) = option_string {
3      println!("Successfully destructured {s}");
4  }
5  if let Some("KO") = option_string {
6      println!("Successfully destructured KO");
7  }
8  if let Some("Maybe") = option_string {
9      println!("Successfully destructured Maybe");
10 }
```

Option Destructuration - *while let*

Répétition conditionnée par la destructuration

- Particulièrement utile pour les itérateurs

```
1 let mut commands = "NAME=Bastien#GunTrig=1.0#ORIENT#".split("#");
2 while let Some(cmd) = commands.next() {
3     println!("Command: {cmd}");
4 }
```

Pour propager une Option : " ?"

```rust
fn add_last_two(stack: &mut Vec<i32>) -> Option<i32> {
    let a = stack.pop();
    let b = stack.pop();

    match (a, b) {
        (Some(x), Some(y)) => Some(x + y),
        _ => None,
    }
}
```

Devient :

```rust
fn add_last_two(stack: &mut Vec<i32>) -> Option<i32> {
    Some(stack.pop()? + stack.pop()?)
}
```

Option - autres fonctions

Autre fonctions utiles de Option :

- `is_some()`
- `is_none()`
- `unwrap_or(default value)`
- `unwrap_or_default()`
- `expect("Error message")`

Pour le reste, voir la documentation

Gestion des erreurs - Le type `Result`

`Result` peut prendre deux valeurs :

- `Ok(valeur)` en cas de succès
- `Err(erreur)` en cas d'échec

Un `Result` est donc paramétré par deux types :

- Celui de la valeur
- Celui de l'erreur
- Par exemple : `Result<TcpStream, std::io::Error>`

Gestion des erreurs - Exemple

```rust
struct Client {
    stream: TcpStream,
}

impl Client {
    pub fn new(address: String) -> Result<Self, std::io::Error> {
        match TcpStream::connect(address) {
            Ok(s) => Ok(Self { stream: s }),
            Err(e) => Err(e),
        }
    }
}
```

Gestion des erreurs - Exemple - Propagation

```rust
struct Client {
    stream: TcpStream,
}

impl Client {
    pub fn new(address: String) -> Result<Self, std::io::Error> {
        let s = TcpStream::connect(address)?;
        Ok(Self {stream: s})
    }
}
```

L'éventuelle erreur de connexion est propagée à l'aide de l'opérateur "?"

Gestion des erreurs - Exemple - `Option` ou `Result` ?

Est-ce que l'utilisateur de `Client` s'intéresse aux `std::io::Error` ?

- Si oui → on laisse comme ça.
- Si non → on retourne plutôt une `Option`

Retourner une `Option` permet de :

- Gérer l'erreur en interne,
- Conserver le sens du résultat : `None` en cas d'erreur,
- Isoler l'appelant des détails de l'erreur

Gestion des erreurs - Exemple

```
1  struct Client {
2      stream: TcpStream,
3  }
4
5  impl Client {
6      pub fn new(address: String) -> Option<Self> {
7          match TcpStream::connect(address) {
8              Ok(s) => Some(Self { stream: s }),
9              Err(_e) => None,
10         }
11     }
12 }
```

Gestion des erreurs - Conversions - `Result` → `Option`

Pour convertir un `Result` en `Option` : `a_result.ok()`

```
impl Client {
    pub fn new(address: String) -> Option<Self> {
        let s = TcpStream::connect(address).ok()?;
        Some(Self {stream: s})
    }
}

match Client::new("invalid".to_string()) {
    Some(c) => println!("Successfully created client."),
    None => println!("Failed to create client"),
}
```

Gestion des erreurs - Conversions - `Option` → `Result`

Pour convertir une `Option` en `Result` : `an_option.ok_or(err)`

- `Some(s)` → `Ok(s)`
- `None` → `Err(err)`

Questions ?