6_Enums_fr

May 2, 2023

1 Enumérations

Les énumérations (enum) permettent de définir un type par toutes ses valeurs possibles.

De nos jours, une adresse IP peut être soit une adresse IPv4, soit une adresse IPv6, mais pas les deux en même temps. Cette propriété des adresses IP les rend très appropriés pour les représenter sous forme d'énumération, car une valeur énuméré ne peut que une des variantes à la fois. Toutes les adresses, qu'elles soient IPv4 ou IPv6 sont fondamentalement des adresses et il est logique de les traiter comme un seul type par du code qui doit fonctionner avec n'importe quel type d'adresse IP.

[ ]:
```
enum IpAddrKind {
    V4, // ceci est une variante
    V6, // une autre variante
}
```

2 Valeurs énumérées

`IpAddrKind` est maintenant un type utilisateur que nous pouvons réutiliser dans notre code. Nous pouvons créer de nouvelles instances de ce type, avec l'une des deux variantes. Chacune des valeurs `IpAddrKind::V4` et `IpAddrKind::V6` sont de type `IpAddrKind`.

[ ]:
```
enum IpAddrKind {
    V4, // ceci est une variante
    V6, // une autre variante
}
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

[ ]:
```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind, // <-- utilisation de IpAddrKind
    address: String,
}
```

```
let home = IpAddr {
    kind: IpAddrKind::V4, // <-- instanciation avec la variante V4
    address: String::from("127.0.0.1"),
}

let loopback = IpAddr {
    kind: IpAddrKind::V6, // <-- instanciation avec la variante V6
    address: String::from("::1"),
}
```

Nous pouvons représenter la même chose d'une manière un peut plus concise avec simplement un `enum`, au lieu d'un membre `enum` dans un `struct`. Rust nous permet de stocker directement des données dans les différentes variantes de l'énumération :

[ ]:
```
enum IpAddr {
    V4(String), // <-- la variante porte directement la chaîne pour l'adresse
    V6(String), // <-- idem
}

let home = IpAddr::V4(String::from("127.0.0.1")); // <-- instanciation de la␣
 ↪variante

let loopback = IpAddr::V6(String::from("::1"));
```

Un autre avantage de cette manière de représenter l'information : chaque variante peut porter des données de types différents.

Vouloir stocker une adresse IP est une préoccupation si commune que la librairie standard de Rust possède déjà les définitions pour cela. Regardons un peu comment la librairie standard définie `IpAddr`. Cette énumération a les même variantes que celles proposées ci-dessus, mais les données portées par chacune des variantes sont de types différents :

[ ]:
```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Les variantes des énumérations peuvent porter n'importer quel type de données, String, types primitifs, structures, et même d'autres énumérations.

2.1 Le type *Option* et son avantage sur `null`

Option est une énumération définie dans la librairie standard de Rust. Elle est très couramment utilisée car elle représente un scénario très courant où une valeur peut être absente. Exprimer ceci directement dans le système de type de Rust permet au compilateur de vérifier que le code traite correctement tous les cas. Cette fonctionnalité permet d'éviter des bugs très courants dans d'autres langages de programmation.

En Rust, le concept de `null`, exprimant une valeur absente n'existe simplement pas. Dans les langages avec `null`, les variables (en particuliers les pointeurs et références) peuvent être dans deux états : `null` ou avec une valeur. Le problème avec `null` se pose si vous essayez de l'utiliser à la place d'une valeur non-`null`, auquel cas vous devriez avoir une erreur (ou pire, pas d'erreur …). Rust ne fonctionnne pas comme cela, à la place il convient d'utiliser `Option`. Voici comment cette énumération est définie :

[ ]:
```
enum Option<T> {
    Some(T), // une valeur est présente
    None,  // pas de valeur
}
```

Cette énumération est si utile qu'elle est inclue dans le prélude de la librairie standard, vous n'avez pas besoin de l'importer explicitement pour l'utiliser, de même que ses variantes `Some` et `None` qui peuvent être utilisées sans le préfixe `Option::`.

[ ]:
```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

Lorsque la variante `Some` est utilisée, nous sommes sûrs qu'une valeur est bien présente dans la variante. A l'inverse, lorsque la variante `None` nous savons que nous n'avons pas de valeur. Parce que `Option` est sont propre type (paramétré par le type `T`), le compilateur fait la différente entre `Option<T>` et `T`, et nous informe si l'on essaye d'utiliser l'un à la place de l'autre.

[ ]:
```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

Il est nécessaire de convertir `Option< T >` en `T` afin d'effectuer des opérations sur ce dernier type. En général, ce mécanisme permet d'éviter beaucoup d'erreurs courantes où une valeur `null` est utilisée par erreur à cause de fausses hypothèses.

2.2 L'opérateur *match*

Pour récupérer la valeur `T` à l'intérieur d'une variante `Some` de `Option< T >`, il est possible d'utiliser une expression `match`. Dans ce cas, `match` est utilisé pour spécifier quelle branche de code exécuter en fonction de la variante réellement utilisée. De manière générale, `match` permet de comparer une valeur en regard de différents patterns et de choisir la branche associée.

Dans le code ci-dessous, la fonction `value_in_cents` prend en paramètre une valeur `Coin`, sans en connaitre la variante exacte. La fonction détermine la variante avec `match`.

[ ]:
```
fn main() {
    enum Coin {
        Penny,  // toutes les variantes
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => 1,  // ceci est une branche du match
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
}
main();
```

Il est possible d'écrire du code arbitraire à l'intérieur des branches de `match`:

[ ]:
```
fn main() {
    enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => {
                println!("Lucky penny!");
                1  // it is the last value of the block that is returned
            },
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
    value_in_cents(Coin::Penny);
}
main();
```

`match` permet de récupérer les valeurs à l'intérieur des variantes de l'énumération :

[ ]:
```rust
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),  // <-- cette variante contient des données
}

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    //            ^^^^^ <-- binding de la variable state au contenu de la variante
    _ => count += 1,
}
```

2.3 Matching avec Option< T >

Considérons le cas où l'on veut écrire une fonction qui prend un paramètre de type `Option<i32>` et, s'il y a effectivement une valeur, lui ajoute 1. Si il n'y a pas de valeur, `None` doit être retourné et aucune autre opération de doit être faite.

[ ]:
```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,           // pas de valeur
        Some(i) => Some(i + 1), // ajoutons 1 à la valeur
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Considérons maintenant cette version de la fonction :

[ ]:
```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

Dans cette deuxième version, nous n'avons pas couvert le cas `None`. Rust sait que nous n'avons pas couvert ce cas et produit une erreur à la compilation. En Rust, le `match` doit être exhaustif, toutes les possibilités doivent être couvertes pour que le code soit valide. Dans le cas présent, Rust nous empêche d'oublier de traiter le cas `None`.

2.4 Le remplaçant _ (placeholder)

Rust nous permet d'écrire un pattern remplaçant (placeholder) pour couvrir tous les autres cas non couverts par les autres branches de `match`. Par exemple, le type `u8`a des valeurs valides de 0 à 255. Cependant, si l'on ne s'intéresse que aux valeurs 1, 3, 5 et 7, nous ne voulons pas avoir à lister tous les autres cas. En rust, ce pattern s'exprime avec `_` :

[ ]:
```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

2.5 Syntaxe raccourcie avec *if let*

La syntaxe `if let` permet de combiner une condition avec le match d'un pattern, d'une façon similaire à `match`, mais sans avoir à traiter tous les cas.

Considérons le programme suivant qui match une valeur de type `Option<u8>`, mais ne veut effectuer quelque chose que dans le cas où la valeur est 3 :

[ ]:
```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Il est possible d'utiliser `if let` à la place pour raccourcir la syntaxe. Voici le code équivalent.

[ ]:
```
if let Some(3) = some_u8_value {
    println!("three");
}
```

3 Exercises

Exercise 1: Ré-écrivez ce code avec la syntaxe `if let`

[ ]:
```
#[derive(Debug)]
enum UsState {
    Alabama,
```

```
    Alaska,
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

// TO DO Re-write the code below using if let
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

Solution:

[ ]:
```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

Exercise 2: Créer une énumération pour classer des évènements web : * Chargement d'une page * Déchargement d'une page * Appui sur une touche * Un texte est collé dans la page * Un click survient à la position (x, y)

Implémentez une fonction (ou une méthode) pour afficher un message pour chacun de ces évène-

ments : * page chargée * page décharchée * la touche x a été appuyée * le texte sss a été collé * click survenu à la position x, y

[ ]:
```
// TODO (1) Define an enum called WebEvent holding the different types of events

// TODO (2) Implement a function that takes a web event as parameter and␣
 ↪displays a custom message for each event type

fn main() {
    // TODO (3) Test your implementation for different event types
}
main();
```

Solution:

[ ]:
```
enum WebEvent {
    PageLoad,
    PageUnload,
    KeyPress(char),
    Paste(String),
    Click { x: i64, y: i64 },
}

impl WebEvent {
    fn inspect(self) {
        match self {
            WebEvent::PageLoad => println!("page loaded"),
            WebEvent::PageUnload => println!("page unloaded"),
            WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
            WebEvent::Paste(s) => println!("pasted \"{}\".", s),
            WebEvent::Click { x, y } => {
                println!("clicked at x={}, y={}.", x, y);
            },
        }
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    let pasted  = WebEvent::Paste("my text".to_string());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    pressed.inspect();
    pasted.inspect();
    click.inspect();
    load.inspect();
```

```
    unload.inspect();
}
main();
```