Rust asynchrone et Embassy

4SE02

Guillaume Duc Samuel Tardieu

Mars 2025

Programmation asynchrone en Rust

Mono-tâche vs. multi-tâches

- Un système passe la majorité de son temps à attendre (l'utilisateur, le stockage, le réseau)
- On souhaite faire plusieurs choses en même temps (système multi-tâches)
- Créer des threads prend beaucoup de ressources :
  - Il faut réserver un espace pour la pile
  - Il faut réserver un espace pour sauvegarder le contexte

Programmation asynchrone

La programmation asynchrone permet de multiplexer plusieurs flux de traitement sur le même thread :

- Un exécuteur[1] exécute tour à tour les flux dont il a la charge
- Chaque traitement se suspend de lui-même lorsqu'il ne peut plus avancer
- Chaque traitement met en place, avant d'indiquer qu'il est bloqué, un mécanisme permettant d'indiquer à l'exécuteur qu'il est de nouveau prêt à avancer

Exemple : lecture de données sur une carte SD

Quand l'exécuteur nous donne la main :

- Si les données sont immédiatement disponibles, nous les renvoyons. Notre travail est terminé.
- Si les données ne sont pas disponibles :
  1. Nous demandons au système de lancer la lecture sur la carte SD
  2. Nous mettons en place un handler d'interruption à la réception de ces données pour prévenir (à ce moment là) l'exécuteur que nous avons progressé
  3. Nous indiquons à l'exécuteur que nous ne sommes pas encore prêt

1. ou exécuti

Les types de la programmation asynchrone en Rust

Chaque flux de traitement est représenté par un objet implémentant le trait `Future`[2]. Ce trait est implémenté par les objets qui savent lancer un calcul en fond et prévenir l'exécuteur quand il est terminé.

```rust
pub trait Future {
    /// Type of the data returned by the future when it is ready
    type Output;

    /// Ask the future if it is ready (Poll::Ready) or not (Poll::Pending)
    fn poll(self: …, …) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    /// Result is available
    Ready<T>,
    /// Result is not yet ready
    Pending,
}
```

2. `core::future::Future`

Le couple `async`/`.await`

En Rust, des facilités permettent de manipuler les `Future` sans les mentionner directement :

- `async` permet de déclarer qu'une fonction ou un bloc est asynchrone et renvoie une `Future`
- À l'intérieur d'un contexte `async`, `.await` sur une `Future` permet de demander à l'exécuteur d'interroger cette `Future` et de nous suspendre si le résultat n'est pas disponible

```
1  async fn add_two_values() -> u32 {
2      let a = first_value().await;   // Wait for the `Future` returned by `first_value()`
3      let b = second_value().await;  // Idem
4      a + b
5  }
```

Ici, `add_two_values()` est en fait une fonction qui renvoie un résultat de type `Future<Output = u32>`. Quand on demandera la valeur de cette `Future` (avec `.await` par exemple), on pourra alors être suspendu si les `Future` renvoyées par `first_value()` et `second_value()` ne sont pas immédiatement disponibles.

To `poll()` or not to `poll()`

En pratique, on n'appelle quasiment jamais `poll()` soi-même sur une `Future`. C'est le travail de l'exécuteur asynchrone :

- Il appelle `poll()` jusqu'à ce qu'il renvoie `Poll::Ready(…)`
- Il fournit à chaque `Future` le moyen de signaler à l'exécuteur qu'elle a progressé

Plusieurs exécuteurs sont disponibles, par exemple :

- Tokio est le plus complet et offre plein de services supplémentaires (connexions réseau asynchrones, etc.)
- Embassy apporte un exécuteur adapté à l'embarqué ainsi qu'un HAL fonctionnant de manière asynchrone

Future et blocage

Que se passe-t-il si une `Future` bloque au lieu d'indiquer qu'elle n'est pas prête ?

⚠ Cela bloque l'exécuteur et toutes les `Future` que cet exécuteur doit interroger. Il ne faut pas bloquer dans une `Future` ! Cela vaut pour tous les contextes asynchrones, car ils ne sont qu'une facilité d'écriture :

```
async fn toggle_led_when_button_pressed() {
  loop {
    while !button_pressed() {}   // Do NOT do this
    led.toggle();
  }
}
```

Si on dispose d'une fonction asynchrone pour attendre que le bouton soit appuyé, on écrira :

```
async fn toggle_led_when_button_pressed() {
  loop {
    wait_for_button_press().await;  // Will suspend current `Future` when not ready
    led.toggle();
  }
}
```

Transformer des `Future` : combinateurs

En important le trait `FutureExt` du crate `futures`, on dispose de combinateurs pour tous les types implémentant `Future`[3] :

```
1 use futures::FutureExt as _;
2
3 let input = some_function_returning_future_with_string();
4 let derived = input.map(|s| s.len()).map(|n| n * 2);
```

Ici, `derived` est un objet implémentant `Future<Output = usize>`. Quand `derived` sera `poll()` par l'exécuteur, cela appellera `poll()` sur le deuxième `map()` qui *pollera* le premier `map()` qui *pollera* `input`.

C'est l'occasion d'indiquer que comme pour les itérateurs, une `Future` ne fait rien tant qu'elle n'est pas *pollée*. Elle ne démarre son exécution éventuelle en tâche de fond qu'au premier `poll()`.

3. Cela ressemble aux combinateurs du trait `Iterator`

Combiner des `Future` : exécution “simultanée”

Des fonctions et macros utiles sont disponibles dans le crate `futures` pour transformer deux `Future` en une seule. Si

- `fut1` est de type `Fut1` qui implémente `Future<Output = T1>`
- `fut2` de type `Fut2` qui implémente `Future<Output = T2>`

Alors par exemple, on a :

- `join(fut1, fut2)` construit un objet qui implémente `Future<Output = (T1, T2)>` (exécution “simultanée” de `fut1` et `fut2` quand on *polle* cette `Future`)
- `select(fut1, fut2)` renvoie un `Either<(T1, F2), (T2, F1)>` :
  - `Either::Left((r1, fut2))` si `fut1` termine en premier
  - `Either::Right((r2, fut1))` si `fut2` termine en premier

  On peut continuer à interroger la `Future` qui n’a pas terminé car elle nous est retournée

Un exécuteur simple (pseudo-code)

L'exécuteur le plus simple est celui qui prend une `Future` en paramètre, l'appelle jusqu'à ce qu'elle ait terminé et renvoie son résultat :

```
fn block_on<T>(fut: impl Future<Output = T>) -> T {
    loop {
        match fut.poll(…) {
            Poll::Ready(v) => { return v; }
            Poll::Pending => {}
        }
    }
}
```

Malgré son aspect simple, cet exécuteur peut pourtant exécuter un nombre illimité de `Future` "simultanément". Comment ?

En utilisant par exemple les fonction `join()` et `select()` du crate `futures`, la `Future` passée à `block_on()` peut créer des sous-futures. Elles seront *pollées* par le `poll()` de la `Future` du niveau supérieur.

Exemple de “sous-future” (pseudo-code)

```rust
pub struct FirstResult<Fut1, Fut2>
{
    fut1: Fut1,
    fut2: Fut2,
}

impl Future<T, Fut1, Fut2> for FirstResult<Fut1, Fut2>
where Fut1: Future<Output = T>, Fut2: Future<Output = T>
{
    type Output = T;
    fn poll(self: …, …) -> Poll<T> {
        if let Poll::Ready(r1) = self.fut1.poll(…) {
            Poll::Ready(r1)  // `fut1` has terminated, return its result
        } else if let Poll::Ready(r2) = self.fut2.poll(…) {
            Poll::Ready(r2)  // `fut2` has terminated, return its result
        } else {
            Poll::Pending    // `fut1` or `fut2` will tell the executor in case of progress
        }
    }
}
```

Note sur `poll()`

Les types sont volontairement non détaillés ici, mais le deuxième argument de la méthode `poll()` contient des informations, dédiée à notre `Future`, permettant de signaler à l'exécuteur que nous avons progressé et qu'il faut nous rappeler.

```
impl Future for … {
    fn poll(self: …, …) -> Poll<T> { … }
}
```

Nous avons transmis à nos sous-`Future` en les *pollant* ces informations. Si elles les utilisent, c'est nous qui seront *pollés* à nouveau par l'exécuteur et qui re-*pollerons* nos sous-`Future`.

Instancier l'exécuteur

Il y a plusieurs manières d'instancier un exécuteur :

- On peut le créer explicitement et lui passer une `Future` :

```rust
fn main() {
    let rt = tokio::Runtime::new().unwrap();     // Create an executor
    rt.block_on(some_asynchronous_function());  // Execute the future returned by the function
}
```

- On peut utiliser un attribut fourni par l'exécuteur pour que `main()` soit définie comme une fonction asynchrone et la `Future` de plus haut niveau qui lancera les autres :

```rust
#[tokio::main]
async fn main() {
    loop {
        some_fonction().await;
        join(func1(), func2()).await;
    }
}
```

La plupart des exécuteurs populaires fournissent un attribut de ce type.

Embassy

Qu'est-ce qu'Embassy ?

Embassy est un cadre logiciel (*framework*) permettant de créer des applications embarquées en Rust. Il est composé de briques indépendantes mais se complétant bien :

- Plusieurs HAL (*hardware abstraction layer*) pour les microcontrôleurs STM32, Nordic, Raspberry Pi RP2040 et ESP32
- Fonctionne en mode synchrone ou asynchrone (les HAL sont compatibles avec les deux)
- Fournit des exécuteurs asynchrone
- Les exécuteurs asynchrones peuvent être associés à des niveaux de priorité différents pour les applications temps-réel
- Gère le temps, intègre des protocoles réseaux (Ethernet, IP, TCP, UDP, LoRa)
- Etc...

On peut par exemple utiliser l'exécuteur de Embassy mais un HAL provenant d'une autre source.

C'est un des *frameworks*[4] les plus pratiques à utiliser pour développer des applications embarquées en Rust.

4. Il existe aussi également RTIC, plus académique

Illustration : délais et répétitions

Embassy fournit des services de `Timer` et de `Ticker` :

- Un `Timer` permet d'attendre qu'un certain délai soit passé :

```rust
Timer::after_millis(50).await;   // Suspend current future for 50 milliseconds
```

- Un `Ticker` permet d'implémenter un service périodique qui ne dérive pas avec le temps :

```rust
let mut ticker = Ticker::every(Duration::from_hz(100));  // 100Hz ticker, starting now
loop {
    do_something();
    ticker.next().await;  // Wait until 1/100th of second has passed since last time
}
```

Même si `do_something()` prend un temps différent à chaque tour de boucle, le ticker attendra qu'il se soit écoulé $k/100$ secondes depuis le début en incrémentant k à chaque passage.

Qu'utiliserait-on pour faire clignoter une led à 1Hz ? Et pour allumer une ligne d'une matrice ?

Embassy : programme principal et tâches

En utilisant le crate `embassy_executor`, on bénéficie d'un attribut permettant de démarrer automatiquement un exécuteur et d'appeler notre fonction `main()` dessus de manière asynchrone :

```
1  #[embassy_executor::main]
2  async fn main(spawner: Spawner) {
3     let config = Config::default();   // Here configure the clocks, etc.
4     let p = embassy_stm32::init(config);
5     …
6     let matrix = Matrix::new(p.PA2, …).await;  // Because we use `Timer` in `new()`
7     …
8     spawner.spawn(display(matrix)).unwrap();  // Use the spawner to start independant tasks
9  }
10
11 #[embassy_executor::task]
12 async fn display(matrix: Matrix<'static>) {
13     let ticker = Ticker::new(…);
14     …
15     loop {
16         ticker.next().await;
17         …     // Display one line
18     }
```

Embassy : utiliser le HAL

Le HAL permet d'utiliser les fonctionalités du micro-contrôleur, ici un port série :

```rust
// Redirect USART1 interrupt to the HAL so that it can do what it needed
bind_interrupts!(struct Irqs {
    USART1 => usart::InterruptHandler<USART1>;
});

// The UART will use DMA channels for efficient transmission
let mut serial = Uart::new(
    p.USART1, p.PB7, p.PB6, Irqs, p.DMA1_CH4, p.DMA1_CH5, usart::Config::default()
);

// Receive 5 bytes asynchronously
let mut buffer = [0u8; 5];
serial.read(&mut buffer).await.unwrap();
```

Le cœur du processeur pourra être mis en veille s'il n'a rien d'autre à faire tandis que le module matériel DMA (*direct memory access*), configuré par le HAL, transfèrera les 5 caractères reçus en mémoire. L'interruption DMA à la fin du transfert réveillera le processeur et préviendra l'exécuteur.

Embassy : une utilisation intuitive des périphériques

Cet exemple est tiré de la documentation d'Embassy :

```rust
#[embassy::task]
async fn my_task(mut button: ExtiInput<'_>) {
    loop {
        button.wait_for_falling_edge().await;
        info!("Pressed!");
        button.wait_for_rising_edge().await;
        info!("Released!");
    }
}
```

Il illustre bien l'interêt d'utiliser ce genre de *framework* pour faciliter le développement d'applications. Le code applicatif reste localisé et n'est pas dispersé à plusieurs endroits : gestionnaire d'interruptions, configuration des `EXTI`, démultiplexage des `EXTI` de 10 à 15 qui partagent la même interruption, configuration des fronts à attendre, utilisation d'objets de synchronisation.

Embassy : gestion des ressources

L'utilisation du système de types et d'emprunt de Rust permet de garantir une utilisation correcte des ressources matérielles :

```rust
#[embassy::main]
async fn main(spawner: Spawner) {
    …
    // This takes ownership of p.PC13 and p.EXTI13, so p.EXTI13 cannot be
    // associated to another pin. The typing system ensures that EXTI13
    // can be used to watch PC13, e.g., that wouldn't have worked with PC12.
    let button = ExtiInput::new(p.PC13, p.EXTI13, Pull::Up);
    spawner.spawn(my_task(button));
}

#[embassy::task]
async fn my_task(mut button: ExtiInput<'_>) {
    loop {
        button.wait_for_falling_edge().await;
        info!("Pressed!");
        button.wait_for_rising_edge().await;
        info!("Released!");
    }
}
```

Embassy : synchronisation

Embassy offre des outils de plus bas niveau pour pouvoir gérer l'échange de données entre tâches, par exemple :

- Des canaux de communication FIFO multi-écrivains multi-lecteurs
- Des canaux avec possibilité de doubler les éléments moins prioritaires
- Des mécanismes de *publish*/*subscribe*
- La possibilité de ne retenir que la dernière valeur d'un canal
- Des verrous

Embassy : exécuteurs multiples

Lorsque le contexte le nécessite, plusieurs exécuteurs peuvent être lancés :

- L'exécuteur par défaut exécute le code en mode "normal"
- Des exécuteurs supplémentaires peuvent exécuter leurs tâches en mode "interruption" à différents niveaux de priorité
- Des interruptions libres peuvent être utilisées par ces exécuteurs : elles seront par exemple sollicitées lorsqu'une tâche de fond voudra signaler que sa `Future` a progressé.

Par exemple, un affichage avec persistence rétinienne pourra préférer, afin d'être sans défaut, l'utilisation d'un exécuteur dédié plus prioritaire que les autres fonctionalités du système.

Embassy : documentation

La documentation d'Embassy se trouve sur `https://embassy.dev`.

⚠ Il y a un sélecteur lorsqu'on regarde la documentation du HAL `embassy-stm32` permettant de choisir le bon micro-contrôleur. Attention à sélectionner le bon modèle pour avoir la documentation la plus adaptée.