Async

Programmation Générique en Rust

12 janvier 2026

1 Le problème

Dans ce TP on va écrire un afficheur pour openstreetmap. Plus exactement on se propose d'écrire un bout de code qui lit un fichier xml renvoyé par une requête overpass et le convertit en fichier svg.

Le fichier xml en entrée contient des nœuds (nodes), situés en début de fichier et constitués d'un identifiant unique (id), d'une latitude (coordonnée y) et d'une longitude (coordonnée x). Dans une première étape on se contente de parser les nœudes et de les stocker dans une table de hachage. Après les nœuds viennent les ways qui codent les routes et les chemins. Chaque way contient des références vers des nœuds (grâce à leurs identifiants) et cette séquence de nœuds correspondant à un ensemble de segments à dessiner dans le fichier de sortie.

Une approche naive du problème consisterait à lire l'intégralité du fichier en mémoire (dans un *String*) puis à calculer tous les segments à dessiner (dans un vecteur par exemple) puis à calculer les dimensions de l'image et à écrire tous les segments dans un fichier svg.

Cette approche pose problème pour les gros fichiers de part sa consommation mémoire. Il est alors classique de remplacer la lecture de l'intégralité du fichier par un itérateur afin de traiter les ways une par une. On réduit ainsi la consommation mémoire à la taille de la table de hachage.

On se propose ici d'aller encore plus loin en utilisant la programmation asynchrone. Une application pratique serait pour un serveur générant des images à la volée.

2 Détails d'implémentation

Afin de réaliser notre programme on utilise la bibliothèque *quick-xml* qui fournit un parseur xml asynchrone.

La documentation donne l'exemple d'utilisation suivant :

```
use quick_xml::events::Event;
use quick_xml::reader::Reader;
```

```rust
// This explicitly uses `from_reader("...".as_bytes())` to use a buffered
// reader instead of relying on the zero-copy optimizations for reading
// from byte slices, which provides the sync interface anyway.
let mut reader = Reader::from_reader(r#"
    <tag1 att1 = "test">
       <tag2><!--Test comment-->Test</tag2>
       <tag2>Test 2</tag2>
    </tag1>
"#.as_bytes());
reader.config_mut().trim_text(true);

let mut count = 0;
let mut buf = Vec::new();
let mut txt = Vec::new();
loop {
    match reader.read_event_into_async(&mut buf).await {
        Ok(Event::Start(_)) => count += 1,
        Ok(Event::Text(e)) => txt.push(e.unescape().unwrap().into_owned()),
        Err(e) => panic!("Error at position {}: {:?}", reader.error_position(), e),
        Ok(Event::Eof) => break,
        _ => (),
    }
    buf.clear();
}
assert_eq!(count, 3);
assert_eq!(txt, vec!["Test".to_string(), "Test 2".to_string()]);
```

Dans notre cas on boucle d'une manière similaire sur tous les *Event* mais seuls les suivants nous intéressent :

- *Eof* pour la fin de fichier afin de générer le pied de fichier svg
- *Start* pour deux types de balises
  - "way" : on démarre une nouvelle way
  - "node" : un nœud avec toutes ses données dans ses attributs
- *Empty* (balise fermée directement) pour deux types de balises
  - "node" : un nœud avec toutes ses données dans ses attributs
  - "nd" : une référence vers un nœud dans une way

On vous fournit différents fichiers xml de tests ainsi qu'un fichier svg d'exemple.

Afin d'éviter de finir avec une fonction tros grosse et difficile à lire on vous conseille de ne pas hésiter à découper votre fonction en sous-fonctions ce qui vous permettra également de programmer quelques fonctions asynchrones supplémentaires.