さうすの Rust 勉強会

lesson01

- 目標：Rust の基礎文法をある程度理解する
  - 今日全部理解しなくてももちろん大丈夫 (これからまた同じ概念が出てくるとき聞いても ◎)
- 困ったら The Book (日本語 / English) を読みましょう
  - これ読んだら今回の講義はやらなくていい
- 質問はいつでも Slack でどうぞ (授業中でも授業中じゃなくても)
  - 答えられなくても調べる

前提

- Rust は結構複雑な言語なのですべてを理解しなくてよい
- とりあえず最低限必要なものを習得して、作りたいものを作れるようになろう

環境構築

- `rustup` をインストールする
- たぶんデフォルト `rustup install stable` やってくれるので大丈夫
- VSCode 使うなら Rust Analyzer 入れとこう
    - VSCode 使わなくても入れる方法あれば入れとこう

基礎文法

もう例を示して終わったことにする。

```
// bin crate だと main 関数が使える
fn main() {
    println!("Hello, world!");
}

// unit / (): 他の言語の void と大体同じonaji
// unit の場合書かなくてもよい
fn hello_world() -> () {
    // println! ← マクロである
    let world = "world";
    println!("Hello, {}!", world);
}
```

```rust
fn add(a: i32, b: i32) -> i32 {
    // こういうブロックを使って計算することもできる
    let result = {
        let sum = a + b;
        sum
    };
    // 最後の一文は return 書かなくていい
    result
}

// 一番普通の書き方
fn add2(a: i32, b: i32) -> i32 {
    a + b
}
```

Struct, Enum

```
struct S1 {
    a: i32,
    b: i32
}

// フィールドがない struct
struct S2;

struct S3(i32);

fn main() {
    let s1 = S1 { a: 5, b: 6 };
    let s2 = S2;
    let s3 = S3(7);
}
```

```
enum E1 {
    A,
    B
}

enum E2 {
    X(i32),
    Y {
        z: i32
    }
}

// ケースがないので作れない
enum E3 {}

fn main() {
    let e1_1 = E1::A;
    let e1_2 = E1::B;
    let e2_1 = E2::X(32);
    let e2_2 = E2::Y { z: 64 };
}
```

所有権

一番大事な部分：

> - Rust の各値は、所有者と呼ばれる変数と対応している。
> - いかなる時も所有者は一つである。
> - 所有者がスコープから外れたら、値は破棄される。
> - 任意のタイミングで、一つの可変参照か不変な参照いくつでものどちらかを行える。
> - 参照は常に有効でなければならない。

https://doc.rust-jp.rs/book-ja/ch04-00-understanding-ownership.html

```rust
#[derive(Debug)] struct T { inner: i32 }
fn main() {
    let t1 = T { inner: 5 };
    let t2 = t1; // OK: move
    // println!("{:?}", t1); // NG: moved

    let t3 = T { inner: 10 };
    let t4 = &t3; // OK: borrow
    let t5 = &t3; // OK: borrow
    println!("{:?}", *t4); // OK: readonly dereference
    // let t6 = t3; // NG: cannot move

    let mut t7 = T { inner: 15 };
    let t8 = &mut t7; // OK: mutably borrow
    t8.inner = 20; // OK: mutate using mutable reference
    println!("{:?}", &t7);
}
```

```
T { inner: 10 }
T { inner: 20 }
```

match 式

https://doc.rust-jp.rs/book-ja/ch06-02-match.html

```
enum T {
    A(i32),
    B(i64),
}

fn main() {
    let t: T = T::A(5);
    match t {
        T::A(a) => { println!("A: {}", a); },
        T::B(b) => { println!("B: {}", b); },
    }
}
```

impl

https://doc.rust-jp.rs/book-ja/ch05-03-method-syntax.html

```
struct S { a: i32 }

impl S {
    fn double(self) -> S {
        S { a: self.a * 2 }
    }
    fn read(&self) -> &i32 {
        &self.a
    }
    fn add1(&mut self) {
        self.a += 1;
    }
}
```

trait（トレイト）

今回は無視しても大丈夫。https://doc.rust-jp.rs/book-ja/ch10-00-generics.html 読んでね。

```
trait T {
    fn t() -> i32;
}

struct S;

impl T for S {
    fn t() -> i32 {
        42
    }
}
```

ジェネリック

https://doc.rust-jp.rs/book-ja/ch10-01-syntax.html

```
struct S1<T> {
    s: T,
}

struct S2<T: Copy> {
    t: T,
}

fn main() {
    let s11 = S1 { s: "test".to_string() };
    let s12 = S1 { s: 1 };
    let s21 = S2 { t: 2 };
    // let s22 = S2 { t: "test".to_string() };
}
```

Option, Result

めちゃくちゃ使われる enum である。

https://doc.rust-jp.rs/book-ja/ch06-01-defining-an-enum.html#option-enumとnull値に勝る利点

https://doc.rust-jp.rs/book-ja/ch09-02-recoverable-errors-with-result.html

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Vec

可変長配列。

```
fn main() {
    // ここで型を書かないとなんの Vec なのか推論できない
    let empty: Vec<i32> = vec![];

    let numbers = vec![1, 2, 3];
    println!("{}", numbers[1]);

    let mut words = vec!["happy", "birthday"];
    words.push("to");
    words.push("you");
    println!("{:?}", &words);
}
```

Iterator

`.iter()`, `.iter_mut()` または `.into_iter()` で作るもの。

```
fn main() {
    let numbers = [1, 2, 3];
    numbers
        .iter()
        .map(|number| number * 2)
        .filter(|number| number >= 4)
        .for_each(|number| {
            println!("{}", number);
        });
}
```

derive

proc-macro で実装されている。

```
#[derive(Copy, Clone, Debug)]
struct D { a: i32, b: i32 }

struct E { a: i32, b: i32 }

fn main() {
    let d = D { a: 5, b: 6 };
    let e = E { a: 5, b: 6 };
    let d2 = d; // OK: copy
    let d3 = d.clone(); // OK: clone
    println!("{:?}", d); // OK
    let e2 = e; // OK: move
    // let e3 = e.clone(); // NG: Clone trait がない
    // println!("{:?}", e); // NG: Debug trait がない
}
```

pub / pub(crate) / pub(super)

https://doc.rust-jp.rs/rust-by-example-ja/mod/visibility.html

モジュールの話だけど、今はあんまり気にしなくてもよい

```
/// crate 外からアクセスできる
pub fn fn1() {}

/// crate 内からしかアクセスできない
pub(crate) fn fn2() {}

/// 親モジュールからしかアクセスできない
pub(super) fn fn3() {}

/// 今のモジュールからしかアクセスできない
fn fn4() {}
```

serde

serde は serialize-deserialize の略で、ライブラリの名前でもある。オブジェクト ↔️ いろんな形式 (JSON, YAML など) の変換のためのライブラリである。

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, PartialEq)]
struct S { a: i32, b: i32 }

fn main() {
    let s = S { a: 5, b: 6 };
    println!("{}", serde_json::to_string(&s).unwrap());
    // {"a":5,"b":6}
    let s2: S = serde_json::from_str(r#"{"a":7,"b":8}"#).unwrap();
    println!("{}", &s2);
    assert!(s2 == S { a: 7, b: 8 });
}
```

async-await, futures (tokio)

（ async-await 使うからスレッドあんまり気にしなくていいはず…？ ）

https://rust-lang.github.io/async-book/ を呼んでもよいが、今は `async` 関数はそのあと `.await` と書かないといけなくて、あと `async` 関数からしか呼べないぐらいの理解で大丈夫。

```
async fn async_fn(a: i32, b: i32) -> i32 {
    // async 関数を呼び出すために await を使う
    tokio::time::sleep(tokio::time::Duration::from_millis(500)).await;
    a + b
}

#[tokio::main] // main 関数を async にするための
async fn main() {
    let sum = async_fn(250, 250).await;
    println!("{}", sum);
}
```

課題

> **ポイント**
>
> - 協力してもよい！
> - もう完全に理解してたら他の人に教えよう！
> - わからないときは自由に Slack のチャンネルで聞こう！

- まず `project/` を埋めて、テストが通るようにしよう
  - きれいな解を書くように心がけよう。（時間があれば）書いたあと他の人のコードを比べてみよう。
  - あとで `cargo clippy` 実行するといい。

- 次に Rust で以下の問題を解こう
  - コードは汚くてもいいので、Rust 使える〜という肌感覚をつけてほしい (すぐには無理だが)
  - https://atcoder.jp/contests/language-test-202301/tasks/abc086_a
  - https://atcoder.jp/contests/language-test-202301/tasks/abc081_a
  - https://atcoder.jp/contests/language-test-202301/tasks/abc081_b
  - https://atcoder.jp/contests/language-test-202301/tasks/abc083_b

さうすの Rust 勉強会

lesson02

内容

- `axum` でサーバーを作る
- テストを書く

HTTP の基本

- GET
  - Safe (安全) + Idempotent (冪等)
    - Safe：0 回のとき n 回と同じ
    - Idempotent；1 回とき n 回と同じ
    - Safe であれば Idempotent
  - Body がない (* と思ったほうがいい)
- POST
  - Idempotent でない
  - Body がある

- DELETE
  - Idempotent
  - Body がない (* と思ったほうがいい)
- PUT
  - Idempotent
  - Body がある
- OPTIONS
  - Safe + Idempotent
  - 普通は HTTP サーバーが処理してくれる
- 他のメソッドもあるが、主に使われるのはこのへん

axum とは？

- Rust の HTTP サーバー用のライブラリ
- tokio（async runtime）が開発したライブラリ
- とりあえず使いやすい
  - けど最近の actix-web もほとんど同じ書き方で行けるらしい
- ドキュメント：https://docs.rs/axum/latest/axum/index.html
  - 全部読んだら今日のスライド見なくていい

Hello World

```rust
use axum::{response::IntoResponse, routing::get, Router};

async fn get_hello_world() -> impl IntoResponse {
    "Hello, World!"
}

#[tokio::main]
async fn main() {
    let app = Router::new().route("/", get(get_hello_world));

    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

trait（Rust の一般的な概念）

他の言語の interface に近い概念。

```
trait Edible {
  fn eat(&self);
}

struct Apple;

impl Edible for Apple {
  fn eat(&self) {
    println!("Apple is eaten!");
  }
}

fn main() {
  let apple = Apple;
  apple.eat();
}
```

trait extension（よくあるパターン）

他のライブラリで定義された trait を他のライブラリで定義された struct の上に impl することはできないので、他のライブラリで定義された struct を拡張したければ、自前の trait を定義する必要がある。

```
trait StringExt { fn double(&self) -> String; }

// String は std で定義されているので、普通に impl String できない
impl StringExt for String {
  fn double(&self) -> String {
    format!("{self}{self}")
  }
}

fn main() {
  println!("{}", "test".to_string().double());
}
```

handler

HTTP リクエストを処理するもの。ただの関数である。

https://docs.rs/axum/latest/axum/handler/index.html

- Handler では extractor を使ってリクエストから情報を抽出することができる（後述）
- 普通の関数だが、引数は FromResponsePart や FromResponse を impl している必要があり、戻り値は IntoResponse を実装している必要がある

```
// Router::new().route("/:id", get_id) だと仮定する
async fn get_id(Path(id): Path<String>) -> impl IntoResponse {
  return format!("HTTP request tried to get {id}");
}
```

extractor

https://docs.rs/axum/latest/axum/extract/index.html

HTTP リクエストのヘッダー / ボディから何かを「抽出」するもの。

二種類あって、

- `FromRequestParts` はヘッダーしか読めないので、一つの handler で何個も使える
- `FromRequest` はボディも読めるので、一つの handler で一個しか使えない

注：Executor を自作するときは `#[async_trait]` を使うこと！

先と同じ例で説明する

```
// Router::new().route("/:id", get_id) だと仮定する
async fn get_id(
  // ここでは Path という extractor が使われている
  Path(id): Path<String>
) -> impl IntoResponse {
  return format!("HTTP request tried to get {id}");
}
```

state

https://docs.rs/axum/latest/axum/#using-the-state-extractor
https://docs.rs/axum/latest/axum/extract/struct.State.html

extension layer

https://docs.rs/axum/latest/axum/#using-request-extensions
https://docs.rs/axum/latest/axum/struct.Extension.html

State よりは自由に使えるもの。

debug_handler

https://docs.rs/axum-macros/latest/axum_macros/attr.debug_handler.html

もし axum がよくわからないエラーが出たとき handler のほうに付けるとエラーメッセージが読みやすくなるらしい。

```
#[debug_handler]
fn route() -> String {
  "Hello, world".to_string()
}
```

```
error: handlers must be async functions
  --> main.rs:xx:1
   |
xx | fn route() -> String {
   | ^^
```

middleware

https://docs.rs/axum/latest/axum/middleware/index.html

HTTP リクエスト・HTTP レスポンスを編集しうるもの。

実装すると自分で Future を実装する必要があるので、ほとんどの場合は `axum::middleware::from_fn` を使って実装する。

一般的なテストの書き方

少なくとも二つの書き方：

- 普通のファイル (の最後) に

```
#[cfg(test)]
mod tests {
  #[test]
  fn test_1_plus_1_is_2() {
    assert_eq!(1 + 1, 2);
  }
}
```

  と書く

- crate の `tests` ディレクトリに `.rs` ファイルを作る

axum のテストの書き方 (公式 example)

`project01` から取ってきた例である。

```
#[tokio::test]
async fn test_nonexistent_key() -> anyhow::Result<()> {
    let router = init_test_router();
    let response = router
        .oneshot(Request::builder().uri("/key/1").body(Body::empty())?)
        .await?;
    assert_eq!(response.status(), StatusCode::NOT_FOUND);
    let body: Bytes = hyper::body::to_bytes(response.into_body()).await.unwrap();
    assert_eq!(&body[..], b"");
    Ok(())
}
```

axum のテストの書き方（axum-test-helper）

`axum-test-helper` というライブラリを使うとより簡潔にテストを書けるはず。しかし正直ある程度複雑なプロジェクトなら自分でヘルパー関数を用意してもいい気がする...

```
#[tokio::test]
async fn test_nonexistent_key() -> anyhow::Result<()> {
    let router = init_test_router();
    let client = axum_test_helper::TestClient::new(router);
    let response = client.get("/uri/1").send().await;
    assert_eq!(response.status(), StatusCode::NOT_FOUND);
    assert_eq!(response.text().await, "");
    Ok(())
}
```

課題

- `project01` を埋めて、テストが通ることを確認しよう
- `project02` を埋めて、テストが通ることを確認しよう
- `project01` と `project02` のテストのコードを理解して、なぜ仕様を確認できていることを理解してみよう
- `project03` を完成する。テストも書こう。

さうすの Rust 勉強会

lesson03

内容

- データベースも使ったサーバーを作る

sqlx

- マクロでデータベースを叩く
- コンパイル時型チェックが入っている
- いろんなデータベースをサポートしているが、今回は PostgreSQL を使う

環境構築

- `direnv` を入れよう

direnv

cargo-expand

`sqlx::query!`

コンパイル時型チェック付きのクエリ用マクロ。

```rust
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let pool =
        sqlx::PgPool::connect(
            &std::env::var("DATABASE_URL").expect("DATABASE_URL must be set"))
            .await?;

    let items = sqlx::query!("SELECT * FROM items").fetch_all(&pool).await?;
    println!("{:?}", &items);

    Ok(())
}
```

`sqlx::query!` で生成されたコード

```
let items = {
    {
        #[allow(clippy::all)]
        {
            use ::sqlx::Arguments as _;
            let query_args = <sqlx::postgres::Postgres as ::sqlx::database::HasArguments>::Arguments::default();
            struct Record { name: String, value: String }
            #[automatically_derived]
            impl ::core::fmt::Debug for Record {
                fn fmt(&self, f: &mut ::core::fmt::Formatter) -> ::core::fmt::Result {
                    ::core::fmt::Formatter::debug_struct_field2_finish(f, "Record", "name", &self.name, "value", &&self.value)
                }
            }
            ::sqlx::query_with::<sqlx::postgres::Postgres, _>("SELECT * FROM items", query_args)
                .try_map(|row: sqlx::postgres::PgRow| {
                    use ::sqlx::Row as _;
                    let sqlx_query_as_name = row.try_get_unchecked::<String, _>(0usize)?;
                    let sqlx_query_as_value = row.try_get_unchecked::<String, _>(1usize)?;
                    Ok(Record { name: sqlx_query_as_name, value: sqlx_query_as_value, })
                })
        }
    }
}
    .fetch_all(&pool)
    .await?;
```

sqlx::query_as!

Row ではなく、クエリの結果を構造体に格納する

```rust
#[derive(Debug)]
struct Item { name: String, value: String }

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let pool =
        sqlx::PgPool::connect(
            &std::env::var("DATABASE_URL").expect("DATABASE_URL must be set"))
            .await?;

    let items = sqlx::query_as!(Item, "SELECT * FROM items")
        .fetch_all(&pool)
        .await?;
    println!("{:?}", &items);

    Ok(())
}
```

`sqlx::query_as!` で生成されたコード

```
let items = {
    {
        #[allow(clippy::all)]
        {
            use ::sqlx::Arguments as _;
            let query_args = <sqlx::postgres::Postgres as ::sqlx::database::HasArguments>::Arguments::default();
            ::sqlx::query_with::<sqlx::postgres::Postgres, _>("SELECT * FROM items", query_args)
                .try_map(|row: sqlx::postgres::PgRow| {
                    use ::sqlx::Row as _;
                    let sqlx_query_as_name = row.try_get_unchecked::<String, _>(0usize)?;
                    let sqlx_query_as_value = row.try_get_unchecked::<String, _>(1usize)?;
                    Ok(Item { name: sqlx_query_as_name, value: sqlx_query_as_value })
                })
        }
    }
}
    .fetch_all(&pool)
    .await?;
```

`sqlx::query`

たまに `sqlx::query!` で書けないものがあって、そのときは `sqlx::query`（`!` がない）を使う。例えば

```
enum OrderBy { DESC, ASC }
fn get_all_entries(conn: &mut PgConnection, order_by: OrderBy) -> Vec<Entry> {
    let order_by_str = if order_by == DESC { "DESC" } else { "ASC" };
    sqlx::query(&format!("
        SELECT * FROM entries
        ORDER BY {order_by_str}
    "))
    .fetch_all(&mut *conn)
    .await
    .unwrap()
}
```

axum と sqlx の使い方

テストの書き方

https://docs.rs/sqlx/latest/sqlx/attr.test.html

`#[sqlx::test]` を使うと、`sqlx::PgPool` をテスト関数の引数から取れる。しかもテストごとに独立なので、並列にテストを実行することができる。`sqlx` がデータベースの作成・マイグレーション・削除までしてくれて、必要なテーブルが作成した状態でテストを作成することができる。

さうすの Rust 勉強会

lesson04

内容

- GraphQL サーバーを作る
  - めんどくさいので今回はおそらくデータベースなしで

GraphQL とは

GraphQL Query

GraphQL Mutation

開発ツール

さうすの Rust 勉強会

lesson05

内容

- 応用プログラムを作る
  - ログイン付きのファイルアップロード・共有ソフトウェア
    - プライベート・制限付き共有・公開
  - 今回に限ってフロントエンドも作る

フロントエンド

今回は Vite を使います。

```
npm create vite@latest frontend --template react-ts
```

Session

Session というのは、

- サーバー側に保存された、ブラウザに紐付けられた情報
  - よってユーザーはその内容を読めないし、書けない (「削除」することはできる)
- ブラウザに紐付けるため、Cookies に Session ID が入っている
  - ユーザーがそれを削除すると、紐付きができなくなる
- 認証と関係ある情報を Session に保存することはよくある

https://crates.io/crates/axum-sessions を用いて実装する。

`async_graphql::Guard`

Authentication（認証）v.s. Authorization（認可）

- Authentication
  - 本人確認
  - 今回は session を用いて実装する
- Authorization
  - 権限
  - 今回では guard を用いて実装する

S3 / Minio + Presigned URL

Containerize（Container 化）

アプリを Docker 化すると、簡単にいろんなサーバーにデプロイできる。Kubernetes などの環境にも動かせることができる。

デプロイ

課題

- `fileshare` のソースコードを読んで、理解せよ
  - わからないことあったら Slack で聞こう

さうすの Rust 勉強会

lesson06

内容

- アプリ、全部 Rust で作ったらどうなるか？
- フロントエンドも実は Rust で書けるぞ！というのを伝えたい