Rust で Web 開発コソコソ噂話

目次

- 自己紹介
- 『Rust による Web アプリケーション開発』について
- 超主観による、バックエンド開発の嬉しさと辛さ
- 書籍で紹介した Rust の実装テクニック

自己紹介

- Yuki (@helloyuki_)
- Rust.Tokyo オーガナイザー
- 普段は某 an 株式会社の、グローバルチーム側に勤務している。
- 共著『実践 Rust プログラミング入門』『Rust による Web アプリケーション開発』
- など。

『Rust による Web アプリケーション開発』について

『Rust による Web アプリケーション開発』について

- 2024 年 9 月 26 日刊行。
- 「現場に Rust を導入し、バックエンド開発をするならどう作るか？」を主眼に置いている。
- すでに Rust への入門は済ませた方向けの一冊。

私の担当章

- 1 章: はじめに
- 3 章: 最小構成アプリケーションの実装
- 4 章: 蔵書管理サーバーアプリケーションの設計
- 7 章: アプリケーションの運用
- 8 章: エコシステムの紹介

ならびに、各章に追加した「コラム」等のうちいくつかが含まれる。

書くきっかけ

- 講談社さん側から当時「TypeScript の入門書を書いてくれないか」と言われていた。
- TypeScript は無理だけど、Rust でこういう企画の本を持ってて…
- 単著しようと半分くらい思っていたけど、👶 が誕生する予定があった。
- 一人ではモチベーションが持たない！→ 共著にしよう！🚀

「リアルワールドな」実装を目指した

- よくあるこの手の本への課題意識
  - 「サンプルで作り方はなんとなくわかった。で、私たちのアプリケーションはどう作ればいいの？」
- 実装の元ネタは、「Rust の新しい HTTP サーバーのクレート Axum をフルに活用してサーバーサイドアプリケーション開発をしてみる」という記事。
  - https://blog-dry.com/entry/2021/12/26/002649
- 『Zero To Production In Rust』から影響を受けた。
  - ただありものを実装して終わりの本ではなく、考え方を伝える本にすることを目指した。
- 十分とは言えないが、保守運用に関する Tips を入れた。

VP 王 E に誉めていただいた 🙌

名誉市民になれたかもしれない。

https://x.com/vaaaaanquish/status/1844230317378503054

Rust でバックエンド開発ってどうなんだ？

下記は書籍に書いた意見ですが...

- バックエンド開発はどのプログラミング言語を使おうとも、欲しいリターンはそれなりに得られる傾向にある。
- Ruby on Rails とか PHP とか Java、TypeScript、Python などがメインストリームと思われるけれど、
- 中には Haskell で作っている会社もある。
- そういうわけでいいとは思うんだけど、「システムプログラミング言語を利用するのはオーバーキルでは」という疑義は拭えないと思われる。

ここまでは「公的」な意見

でも、公的な意見（タテマエ）と私的な意見（ホンネ）は違うのが常！

ここからは私の意見！

超主観による、バックエンド開発の嬉しさと辛さ

嬉しさ

- リソース効率や速度に対する圧倒的な安心感がある。
- 可変の箇所がコード上に確実に落ちている。
- 意外に書き方に迷うことがない。
- エルゴノミックなツール周り。
- 運用は楽だったかも？

リソース効率や速度に対する圧倒的な安心感がある。

- Scala や Kotlin でもあまり感じたことはないが、「遅くはないだろう」という安心感はある。
- もちろん、計算量の多いコードや不注意による「遅さ」がなくなるわけではない。
- ただ、JVM 系の言語との比較になってしまうが、それらより圧倒的に小さなリソースで動く印象がある。

可変の箇所がコード上に確実に落ちている

- コードは書く時間より、読まれる時間の方が圧倒的に長い。
- とくに関数の引数に `&mut` が入ってくる嬉しさ。
- どれだけ呼び出し階層の多い関数であっても、不変だった参照が突然可変に変わることはない。
- 明示的に書かれているので、コードを読んでいて関数内で行われるであろう処理の予測を立てやすい。
- 型として情報が落ちている = 処理にまつわる多くの情報がコード上に落ちている。

意外に書き方に迷うことがない

- やはり Kotlin との比較になってしまうが、書き方に迷うことが意外になく、意外に統一感が出る。
- Kotlin の場合、
  - データの表現に class、data class、value class、enum、sealed interface など、本当にさまざまな表現方法がある。
  - どう違うん？となりがち。
  - 他だと、例外と Result 型が混ざったり。
  - 手段を複数取れると、人によって解決策にばらつきが出がちになる。
- Rust の場合、意外に 1 か 2 パターンくらいに実装のバリアンスが収まる印象。

エルゴノミックなツール周り

- cargo が圧倒的に使いやすい。
- 1 つ入れるだけで、linter や formatter まで全部揃うのがよい。
    - 他の言語だと、「どのリンターを入れるか」みたいな議論が起こるものもある。
- cargo は起動も速くていいと思う。

運用は楽だったかも？

あくまで、「広告配信サーバー」が前提です。

- 普通に実装しても広告配信サーバーに必要な QPS に耐えられるくらいのパフォーマンスを楽に出せる。
  - JVM 系言語で広告配信サーバーを実装していたことはあるが、結構いろんなプロファイルをして、緩和策を敷いてという頑張りを行なっていた印象がある。
  - Rust 側はほとんどハックした記憶がない。JVM 系言語だったときは結構がんばった記憶がある。
- メモリの動きの予測が立ちやすいような気がする。
  - ここはスタック、ここはヒープ、みたいなのが明示的。
  - 単に GC がなく、オブジェクトが残り続けている…みたいなことが起こらない。StW もない。
  - VM に対する変なパラメータチューニングはもちろん不要。

辛さ

- 「難しいんでしょ？」と常に言われること。
- 「Web に使うのは非合理的だ、意味ない」と言われること。
- コンパイルが遅くてコーヒータイム ☕ が発生しがち。
- サードパーティライブラリへの依存が多くなりがち。
- 1 箇所書き換えると芋づる式に修正が発生する。

「難しいんでしょ？」と常に言われること

- バックエンド開発に限っていえば、単にリクエストを受け取ってデータベースとのやりとりをし、レスポンスを返すだけなので…と言いたいところだが。
- async/await や futures、tokio をしっかり使おうとすると難しい場面が出てくる。
  - ライフタイムと非同期処理が絡むと難しさは確かに増幅される。
  - が、futures と組み合わせると解消できる場面も多くあるような？（パッと例は出てきませんが…）
  - ただ Scala の経験から行くと、Akka Streams や cats-effect みたいなライブラリを入れる時に同様の「難しさ」を感じたことがある。
  - 非同期処理というか Future というかストリームというかそういう概念自体がそもそも難しいのかも。
- なので、両手をあげて「難しくなんかないですよ」とは言えない。
- ただ、手に負えないほど難しいわけではない、とも言いたい。

「Web に使うのは非合理的だ、意味ない」と言われること

- 「用途違い」という主旨の主張の場合、たとえば Kotlin を入れるのと同じでは？と思っている。
  - システムプログラミング用の Rust みたいな言語をバックエンド開発にいれるのは、
  - ほとんど Android 用みたいな言語をバックエンド開発に入れるのと同じ構図。
  - Kotlin 自体、言語自体のデザインの尖り度合いのバランスがよくて使っている（vs Java、Scala）
- 「オーバーキルだ」という主旨の主張の場合
  - 要するに他の言語なら簡単に達成できる目標に、わざわざ余計な難しさを投入していないか？という話。だいたいのケースがそうでしょうね。
  - Rust の言語デザインあるいはシンタックスが好きで入れているケースが多そう。そしてそれは必ずしも悪いことではない。
  - Swift や Scala ではダメなのか…？→ 難しい理由がいくつかある。

コンパイルが遅くてコーヒータイム ☕ が発生しがち

- ローカルマシンでのビルドも、CI も。
- 快適性のためにはマシンパワーが必要になるが、業務上強いマシンを渡せない環境（たとえばオフショア開発とか）での採用は本当におすすめできないかも。
- とくに tokio と AWS SDK を入れたあたりから急に重くなる印象を持っている。
- ある程度緩和する方法は書籍に書きました！

サードパーティライブラリへの依存が多くなりがち

- Rust は標準ライブラリが薄く作られている。標準ライブラリを増やすと標準ライブラリの管理が増えるため。
- 個人的にはこれが一番辛かった。
- サードパーティライブラリを入れるということは、いくつかの追加のリスクを抱えるということである。
  - そのライブラリの使い方を覚えなければならない。
  - ライブラリのアップデートをしなければならない。
  - なぜかマイナーバージョンの変更でも API に破壊的変更が入ることがある（e.g. Axum）。1.x じゃないからというのはそうなのだが...
  - 最近だと、サードパーティライブラリ周りを狙った攻撃も気になるところ。
  - 作者が突然開発を停止する可能性がある。

1 箇所書き換えると芋づる式に修正が発生する

- 型情報やコード上の表現として、やる行為のすべてが落ちているということは、1 箇所変えるといろんな箇所もそれに伴って変更する必要があるということである。
- たとえば元々普通の同期関数だったものを `async fn` に置き換えた時、芋づる式に修正が発生して面倒くさいことがある。
    - What Color is Your Function?: https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
- 他には、ライフタイム `<'a>`、型パラメータ `<T>`、可変制御 `mut`、 `&mut` など。
- しかも結構発生する…コード書く時って探索的になるから…

書籍で紹介した Rust の実装テクニック

書籍で紹介した Rust の実装テクニック

- クリーンアーキテクチャ（風）
- DI の実装
- `AppRegistryImpl` と動的ディスパッチ
- エラーハンドリング
- モデル変換バケツリレーには `From`
- Newtype Pattern
- 関連し合う型同士の情報を型に落としておく
- フィーチャーフラグを使った実装のオン・オフの管理
- cargo workspace
- 「こうすればよかった」と思う箇所

クリーンアーキテクチャ（風）

- 「レイヤードアーキテクチャ」「オニオンアーキテクチャ」
- `api`、`kernel`、`adapter`の 3 層に分けた。
  - `kernel`と`adapter`はいわゆる DIP を施している。
- レイヤードアーキテクチャを採用し、DIP させた例としては、単によく見る実装だから。説明用であって、推奨しているわけではない。

DI の実装

- 軽量な DI コンテナを実装した。
  - `AppRegistryImpl` という構造体の中に必要な依存情報を持たせておいた。
  - トレイトと具象実装である構造体の紐付けは、動的ディスパッチを利用した。
  - 依存は、トレイトを経由して取り出せるようにしている。
- `AppRegistryImpl` を Axum の `State` に持たせて、各ハンドラに配れるようにした。
- ちなみにだが、DI コンテナの中身は `HashMap` とかでもいいと思います。

`AppRegistryImpl` と動的ディスパッチ

```rust
#[derive(Clone)]
pub struct AppRegistryImpl {
    health_check_repository: Arc<dyn HealthCheckRepository>,
    book_repository: Arc<dyn BookRepository>,
    auth_repository: Arc<dyn AuthRepository>,
    checkout_repository: Arc<dyn CheckoutRepository>,
    user_repository: Arc<dyn UserRepository>,
}
```

`AppRegistryExt`

DI コンテナをテストでモックできるようにトレイトに切り出している。

```
#[automock]
pub trait AppRegistryExt {
    fn health_check_repository(&self) -> Arc<dyn HealthCheckRepository>;
    fn book_repository(&self) -> Arc<dyn BookRepository>;
    fn auth_repository(&self) -> Arc<dyn AuthRepository>;
    fn checkout_repository(&self) -> Arc<dyn CheckoutRepository>;
    fn user_repository(&self) -> Arc<dyn UserRepository>;
}
```

`State`

- Axum の `State` は、Axum のサーバー全体で保持する状態を管理するための機構。
- 書籍では `State` しか利用していないが、 `FromRef` トレイトを実装させることで、子の State を作り、それをハンドラ側で取り出させる、みたいな実装も可能。

`State`の利用例

`registry`から`BookRepository`を呼び出している。ここには、依存関係が解決された状態の`BookRepository`が来ている。

```
pub async fn register_book(
    user: AuthorizedUser,
    State(registry): State<AppRegistry>,
    Json(req): Json<CreateBookRequest>,
) -> AppResult<StatusCode> {
    req.validate(&())?;

    registry
        .book_repository()
        .create(req.into(), user.id())
        .await
        .map(|_| StatusCode::CREATED)
}
```

エラーハンドリング

- エラー型の実装
- アプリケーション内でのエラーハンドリング
- エラーごとに返す HTTP ステータスを決めるハンドリング

エラー型の実装

`thiserror` というクレートを使用し、エラーメッセージの生成や他のクレートからのエラー型を変換する作業を簡略化した。Rust では、エラー型は `enum` のヴァリアントとして表現させておき、あとでパターンマッチに回すのが比較的一般的かと思われる。

```
#[derive(Error, Debug)]
pub enum AppError {
    #[error("{0}")]
    UnprocessableEntity(String),
    // 略
    #[error("{0}")]
    ValidationError(#[from] garde::Report),
    #[error("トランザクションを実行できませんでした。")]
    TransactionError(#[source] sqlx::Error),
    // 略
}
```

アプリケーション内でのエラーハンドリング

Rust には `?` があるので、基本的にその伝播をハンドラーまで回している。最後、ハンドラーがさらにエラーごとに HTTP ステータスコードを切り替える。エラーハンドリングの方針はプロジェクトによりけりだと思うので、参考程度に。

```
pub async fn login(
    State(registry): State<AppRegistry>,
    Json(req): Json<LoginRequest>,
) -> AppResult<Json<AccessTokenResponse>> {
    // 略
    let user_id = registry
        .auth_repository()
        .verify_user(&req.email, &req.password)
        .await?;
    let access_token = registry
        .auth_repository()
        .create_token(CreateToken::new(user_id))
        .await?;
    // 略
}
```

エラーごとに返す HTTP ステータスを決めるハンドリング

Axum の機能で、`IntoReponse` トレイトをエラー型に実装させると、エラーの伝播時に、ハンドラ内でステータスコードのハンドリングを明示的に書くことなく、裏でハンドリングさせることができるようになる。ミス防止やボイラープレート防止に使える。

続き: エラーごとに返す HTTP ステータスを決めるハンドリング

```
impl IntoResponse for AppError {
    fn into_response(self) -> axum::response::Response {
        let status_code =
            match self {
                AppError::UnprocessableEntity(_) => {
                    StatusCode::UNPROCESSABLE_ENTITY
                }
                // 略
                e @ (AppError::TransactionError(_)
                // 略
                | AppError::ConversionEntityError(_)) => {
                    tracing::error!(
                        error.cause_chain = ?e,
                        error.message = %e,
                        "Unexpected error happened"
                    );
                    StatusCode::INTERNAL_SERVER_ERROR
                }
            };
      // 略
    }
}
```

モデル変換バケツリレーには`From`

- レイヤードアーキテクチャを採用すると発生するモデル変換のバケツリレーだが、Rustでは`From`トレイトで済ませられる。`.into`も生えて便利。
- `kernel`には変換機構を持たせておらず、`api`で`From`を使って変換させ、レイヤー間の依存のルールを守らせている。

モデル変換バケツリレー

```
impl From<CreateBookRequest> for CreateBook {
    fn from(value: CreateBookRequest) -> Self {
        let CreateBookRequest {
            title,
            author,
            isbn,
            description,
        } = value;
        CreateBook {
            title,
            author,
            isbn,
            description,
        }
    }
}
```

Newtype Pattern

ID 型など、通常は UUID 型などで全統一しておいてもよいものの、使う ID の取り違えを防げると有益なケースがある。Newtype Pattern を使うと、中身は同じだが実質別物として判定され、取り違えるとコンパイルエラーを発生させることができる。オーバーヘッドはない。

書籍内ではマクロを使って関連実装を自動生成させている。

```
pub struct BookId(uuid::Uuid);
```

関連し合う型同士の情報を型に落としておく

- Redis への値の保存時、基本的には `String` で済ませられはするのだが、可能なら個別に型付けしておきたい。
- 最終的に行き着く Redis 用クライアントに代表されるような、実装の抽象度の高い箇所におけるキーと値の渡し間違えを防ぎたい。
- キーと値の型でどのような割り付けが行われているのかを情報としてコードに落としておきたい。

`RedisKey`、`RedisValue`

まずは二つのトレイトを用意し、Key 側にどの Value の型を持ちうるかの情報を持たせられるよう、関連型を利用する。

```
pub trait RedisKey {
    type Value: RedisValue + TryFrom<String, Error = AppError>;
    fn inner(&self) -> String;
}

pub trait RedisValue {
    fn inner(&self) -> String;
}
```

実際に適用する

```
pub struct AuthorizationKey(String);
pub struct AuthorizedUserId(UserId);

impl RedisKey for AuthorizationKey {
    type Value = AuthorizedUserId;

    fn inner(&self) -> String {
        self.0.clone()
    }
}

impl RedisValue for AuthorizedUserId {
    fn inner(&self) -> String {
        self.0.to_string()
    }
}
```

フィーチャーフラグを使った実装のオン・オフの管理

- utoipa で OpenAPI 向け実装を生やす際に使用している。
- 今回は、下記のような要件とした。
  - ローカル環境では OpenAPI が欲しい。
  - 本番リリース時には OpenAPI は不要。
- utoipa はマクロでいろいろ実装を生やすが、本番モジュール（リリースビルド）にはそうした実装を含めたくない。
- こういったケースでフィーチャーフラグが使える。

サーバーの起動時

`#[cfg(debug_assertions)]` とすると、デバッグビルドでのみコンパイル対象とできる。

```
async fn bootstrap() -> Result<()> {
    // 略

    let router = Router::new().merge(v1::routes()).merge(auth::routes());
    #[cfg(debug_assertions)]
    let router = router.merge(Redoc::with_url("/docs", ApiDoc::openapi()));
    // 略
```

各ハンドラにおける制御

`#[cfg_attr(debug_assertions, ...)]` で制御することができる。

```
#[cfg_attr(
    debug_assertions,
    utoipa::path(post, path="/api/v1/books",
        request_body = CreateBookRequest,
        responses(
            (status = 201, description = "蔵書の登録に成功した場合。"),
            (status = 400, description = "リクエストのパラメータに不備があった場合。"),
            (status = 401, description = "認証されていないユーザーがアクセスした場合。"),
            (status = 422, description = "リクエストした蔵書の登録に失敗した場合。")
        )
    )
)]
// 略
pub async fn register_book(/* 略 */) {}
```

cargo workspace

- 一つのワークスペース下で、パッケージごとに`api`、`kernel`、`adapter`を切るのも悪くないとは思うが、年月を経るとだんだん肥大化する。
- ワークスペースひとつだと、たとえば`api`に変更を入れただけでも`kernel`と`adapter`のコンパイルないしはチェックが走る。
- これにより、コンパイル時間が増大する可能性がある。

cargo workspace

- 上記のような問題の解決策として、cargo workspace という機能が利用できる。レイヤードアーキテクチャとは相性がいいように思う。
- 複数プロジェクトを管理できるようになる機能。本書では、`api`、`kernel`などの単位をワークスペースにしている。
- 他のプログラミング言語用のビルドツールとかで見る。Gradle とか、sbt とかでは割と普通の機能ではある。
- workspace 機能を利用すると、変更しておらず再コンパイル不要なプロジェクトをコンパイル対象外に置いてくれるケースが増える。

本書で「こうすればよかった」と思う箇所

- アプリケーションサービスを導入すればよかった。
  - `Repository`を見てもらうとわかるが、意外にエンティティを跨いでクエリをかけている箇所がある。
  - 集約という観点から見ると、一応整合性がとれてはいるものの、考え方によってはやりすぎ。
  - アプリケーションサービスを導入して、`Repository`の呼び出しをまとめ上げるとよいかもしれない。
  - トランザクションもアプリケーションサービスで発行するとよいかも。
    - `sqlx::Acquire`というトレイトを実装し、それをアプリケーションサービスが持つことで綺麗に実現できそう。
    - https://qiita.com/FuJino/items/08b4c3298918191eab65

本書で「こうすればよかった」と思う箇所

- Always Valid Domain Model に従えばよかった。
  - ドメインモデルは、常に valid な状態でしか生成されないということ。
  - つまりバリデーションチェックを `kernel` のデータを生成する瞬間に行わせればよかった。
  - 単にコードが無意味なバケツリレーをしているように見えるため。

本書で「こうすればよかった」と思う箇所

- ↑ を実装して誰かブログに書いてください！💛

おわりに

- Rust による Web アプリケーション開発にもやはりトレードオフはある。
- 本書の実装はあくまで一例。新しい手法を開発して欲しい。
- 業務委託（副業）として呼んでもらえればアドバイスなどできます。ぜひ。