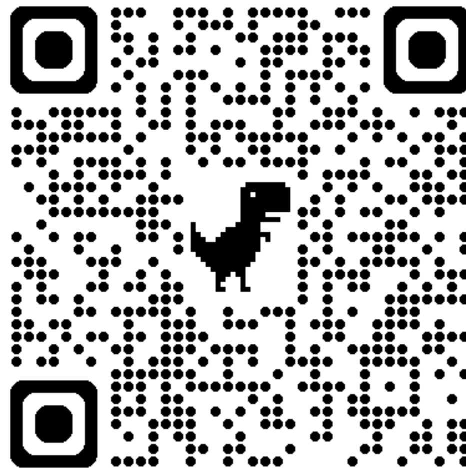
開発効率と性能の両立を目指すRust連携プログラミング言語Kaede

平本一桐
言語処理系開発コース

コメントアプリのデモ

Kaede で書いた Web アプリ

https://bit.ly/40PWr4p



QRコードを読み取ってコメントしてください

自己紹介

平本 一桐 ヒラモト イットウ
放送大学 教養学部 教養学科 情報コース 3年

普段はアルバイトとして、以下の開発をしています。

- Web フロントエンド と バックエンド
- Elasticsearch を使った検索基盤の構築
- ソースコードからコールグラフを生成する静的解析ツールの開発

Web サーバーを書くのが好きで、その中で感じた課題を解決するために Kaede を作りました。

今日話すこと

まず全体像、その後に内部実装を見る

1. 背景と発想
2. 言語機能
3. 内部実装
4. まとめ

1. 背景と発想

問題意識

Rust で Web サーバーを書いていて感じたこと

Rust は性能と安全性に優れるが、Web サーバーを書くときに
「もう少し軽く書きたい」と感じる場面が何度もあった

- 共有状態や、データのまたぎが増えるほど
  所有権・借用・ライフタイムを意識する量が増える
- 並行処理のたびに、データの持ち方を意識する必要が出やすい
- 結果として、**本筋より言語・型システム側の都合に時間が取られやすい**

Kaede の発想

Web サーバーを軽く書きたい、でも性能は捨てたくない

1. サーバーの大部分は Kaede で軽く書く
2. 性能が必要な部分だけ Rust を使う
3. その連携の手間を最小化する

Kaede の中核

この発想に沿った 3 つの柱

- **軽く書く** — GC で所有権を気にせず書ける。式指向・列挙型・パターンマッチなど、Rust に近い表現力
- **速さは捨てない** — 性能が必要な部分だけ Rust をそのまま使える
- **並行処理** — 軽量スレッド・チャネルと non-blocking I/O でリクエストを捌く

2. 言語機能

標準 HTTP / WebSocket ライブラリ

「軽く書く」— 標準ライブラリだけでサーバーが書ける

```
mut app := std.http.App::new()

app.static_file("/", "index.html")

app.get("/api/hello", |req, res| {
    res.send_text("hello")
})

app.ws("/ws/echo", |req, ws| {
    ws.send_text("connected")
})

app.listen(port=8080)
```

軽量スレッド

「並行処理」— 少数の OS スレッドで多数のタスクを捌く

```
fn worker(id: u64) {
    println(id)
}

spawn worker(1)
spawn worker(2)
spawn worker(3)
```

- OS スレッドより軽量なので、接続ごとに1タスク割り当てても問題ない
- タスクの切り替えはユーザ空間のコンテキストスイッチで、OS スレッドの切り替えよりコストが軽い
- I/O 待ちの間はスレッドを塞がないので、少数のスレッドで多数のタスクを捌ける

CSPモデル

「並行処理」— チャネルでタスク間の通信ができる

```
ch := Channel<String>::new()

fn sender(ch: Channel<String>) {
    ch <- "hello" // 送信
}

spawn sender(ch)

match <-ch { // 受信
    Option::Some(msg) => println(msg),
    Option::None => {},
}
```

- 受信側はデータが届くまで待機し、他のタスクにスレッドを譲る

Rust 連携

「速さは捨てない」— 必要な部分だけ Rust を直接使う

```
pub fn comment_json(id: u64, author: &str, message: &str) -> String {
    json!({ "id": id, "author": author, "message": message }).to_string()
}
```

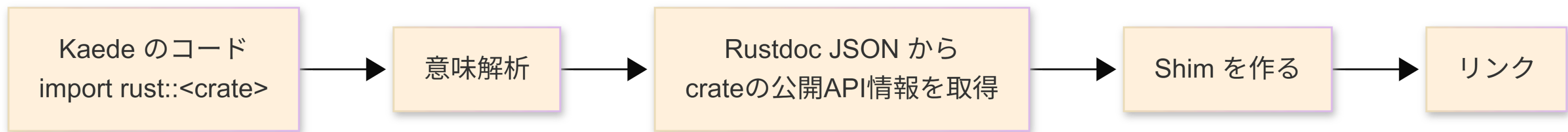
```
import rust::comment_app

json := rust::comment_app::comment_json(123, "Taro", "hello, world!")
```

- サーバーの流れは Kaede、CPU を使う処理は Rust — GC のオーバーヘッドを避けられる
- 今回のデモでは、JSON 生成を Rust（`serde`）に任せている

3. 内部実装

Rust 連携の仕組み

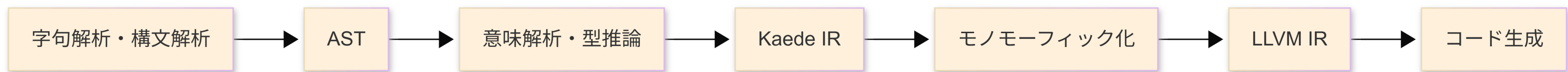
追加の記述なく Rust を使える裏側



- `Shim` は Rust と Kaede の間をつなぐ薄い互換層
- 最終的には Kaede の実行ファイルにまとめてリンクされる

コンパイルパイプライン

Kaede は LLVM バックエンドの静的型付けコンパイル型言語



- `Kaede IR` は型と名前解決が済んだ Kaede 専用の中間表現
- 標準ライブラリ・ランタイム・GC も合わせてリンクされる

並行処理ランタイムの考え方

書き味は同期的、実行は non-blocking

- コードは上から順に書くだけ — async / await やコールバックは不要
- I/O 待ちは epoll / kqueue で検知し、ユーザ空間のコンテキストスイッチでタスクを切り替える
- Channel 待ちも同じスケジューラで扱う

イベントループや状態機械を意識せずに、サーバー処理を書ける

まとめ

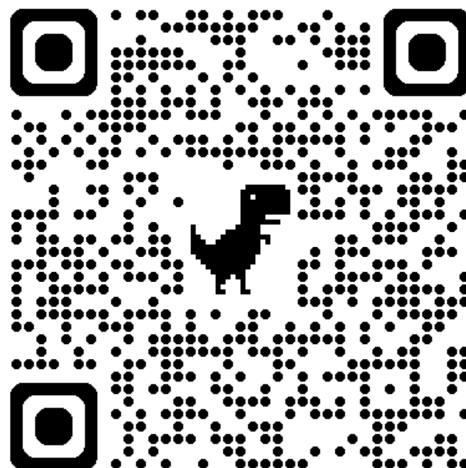
Web サーバー開発の課題から生まれた言語

- Rust で Web サーバーを書く中で
  所有権や並行まわりの記述の重さを感じた
- Kaedeは、サーバー処理を素直に書ける言語機能を標準で備える

サーバーの大部分を Kaede で軽く書き、要所だけ Rust に任せる設計

GitHubリポジトリ

https://github.com/itto-hiramoto/kaede



今回使ったコメントアプリもリポジトリ内にあります