ハイパフォーマンスプログラミング言語Rust

TU 坂田誠也

お品書き

- 自己紹介
- Rustとは
- 実行速度
- 開発パフォーマンス
- Rustのユースケース

お品書き

- **自己紹介**
- Rustとは
- 実行速度
- 開発パフォーマンス
- Rustのユースケース

自己紹介

坂田誠也(28歳エンジニア歴3年ちょっと)

技術的なこと

- 一応サーバサイドエンジニアだけどだいたいなんでもやってる
- 最近はTypeScript+GCP

技術的じゃないこと

この発表の対象

プログラミング経験n年のエンジニアonly

Rust自体の学習コストが高め&普段意識しづらい内容が多めなので初学者だと難しいかもしれません。。。
（それでも得られるものはあります）

フロントエンド向けの話題あります(wasmの話)

お品書き

- 自己紹介
- **Rustとは**
- 実行速度
- 開発パフォーマンス
- Rustのユースケース

Rustとは

- Rustの紹介
- 機能・特徴
- Rust製ソフトウェア

Rustとは

- **Rustの紹介**
- 機能・特徴
- Rust製ソフトウェア

Rustとは

Rustの紹介

- Mozillaが支援しているオープンソースのシステムプログラミング言語
- 速度、並行性、安全性を保証するC言語とC++に替わるプログラミング言語を目指している
- 強い型付けでマルチパラダイム

Rustとは

- Rustの紹介
- **機能・特徴**
- Rust製ソフトウェア

機能・特徴

- 所有権・借用・lifetimeによるGCなしの自動メモリ管理
- null safe
- 総称性(generics)
- パターンマッチ
- 言語レベルでのスレッド安全性の保証
- マクロ
- 言語レベルで組み込まれているテスト機能
- 優秀なパケージマネージャー
- 環境構築がとても簡単(コマンドコピペして実行で終わり)

Rust とは

- Rustの紹介
- 機能・特徴
- **Rust製ソフトウェア**

Rustとは

Rust製ソフトウェア

- Firefox
  - Servo(HTML レンダリングエンジン)
  - Quantum(ウェブエンジン)
- Redox(マイクロカーネル OS)
- Deno
- Read States(Discord のメッセージ基盤サービス)
- Nucleus(Dropbox の同期エンジン)
- Piston(ゲームエンジン)

お品書き

- 自己紹介
- Rustとは
- **実行速度**
- 開発パフォーマンス
- Rustのユースケース

実行速度

Debianのプロジェクトが様々なプログラミング言語の実行速度を比較検証している。

The Computer Language Benchmarks Game

C++やCに次ぐ実行速度を誇る。

実行速度

Q. なんで速い？
A1. 機械語に直接コンパイルされるため
A2. ガベージコレクションを持たないため
A3. ゼロコスト抽象化によりランタイムシステムのオーバーヘッドの削減を追求しているため

機械語に直接コンパイル

PythonやJavaは独自のVMを介してVM用の言語に変換される

➡️ 速度面で不利になる。

機械語に直接コンパイル

C、C++やRustはコンパイル後の最終結果は機械語になり、VMを介さずに実行できるのでVMによる速度低下が起きない。

➡️ 多くのプログラミング言語との速度の違い

GCC(GNU Compiler Collection)やLLVM(Low Level Virtual Machime)などによってプロセッサに応じた機械語を生成する。（RustはLLVMを使用）

ガベージコレクションがない

多くのプログラミング言語では、不要なメモリ領域を開放する処理を裏で逐一行う。

CやC++は手動でメモリを確保・開放する必要があるのでガベージコレクションは便利

ガベージコレクションがない

ガベージコレクションの処理はいつ実行されるか分からず、不要なメモリを開放するのに処理の停止時間が発生する。

➡️ これがGoとの速度の違い

じゃあどうやってメモリ管理を行っているのか

➡️ 所有権・借用とライフタイムによって第3の道を選択

ガベージコレクションがない

Q. このコードはコンパイル可能でしょうか？

```
fn main() {
    let s = String::from("無職転生はいいぞ");
    let s2 = s;
    println!("{}", s);
    println!("{}", s2);
}
```

ガベージコレクションがない

A. コンパイルできません

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:4:20
  |
2 |     let s = String::from("無職転生はいいぞ");
  |         - move occurs because `s` has type `std::string::String`, which does not implement the `Copy` trait
3 |     let s2 = s;
  |              - value moved here
4 |     println!("{}", s);
  |                    ^ value borrowed here after move
```

所有権(ownership)は

1. 変数は値に束縛され所有権を持つ
2. 値に束縛される変数は必ず一つ

ガベージコレクションがない

```
fn main() {
    let a = String::from("無職転生はいいぞ"); // aのライフタイムここから
    {
        let b = String::from("ひぐらしもいいぞ"); // bのライフタイムここから
        println!("{}", a);
        println!("{}", b);
    } // bのライフタイムここまで
    println!("{}", a);

    let c = a; // aのライフタイムここまで cのライフタイムここから

    // ERROR!
    // println!("{}", a);
    println!("{}", c);
} // cのライフタイムここまで
```

ガベージコレクションがない

```
fn main() {
    let mut s = String::from("コードギアス 反逆のルルーシュ"); // mutで再代入可能になる(javascriptのlet)
    s = print_favorite_anime(s); // 値の所有権を返す
    println!("{}", s);
}
// 所有権を返す必要があるので戻り値がString型になってしまう
fn print_favorite_anime(anime: String) -> String {
    println!("my favorite anime is {}.", anime);
    anime
}
```

さらに詳しい話（ヒープやポインタなどの話）は公式ドキュメント「4.所有権を理解する」を読もう

ガベージコレクションがない

所有権は渡さないで値の参照だけ渡す（借用）ことができる。
ただし制約あり

- 参照は元の所有者のライフタイムよりも長く生存できない
- 不変参照はいくつでも参照を渡すことができるが可変参照は一度に一つ

```rust
fn main() {
    let s = String::from("コードギアス 反逆のルルーシュ");
    // 値の所有権ではなく値の参照を渡す（借用）
    print_favorite_anime(&s);
    println!("{}", &s);
}
fn print_favorite_anime(anime: &String) {
    println!("my favorite anime is {}.", anime);
}
```

ゼロコスト抽象化

端的に言えば**抽象化した処理を実行時の追加コストなしに動作すること**（元はC++）

trait と dyn

クラスは内部に状態を持ち隠蔽することで外から見たときに内部の細かい処理に気を使わなくても良いようにする（カプセル化）

➡️ 実行時に負荷が高くなる

Rust は trait で様々な型に共通のメソッドを実装できるようにすることでポリモーフィズムを実現。

trait と dyn

Rustはデフォルトでコンパイル時に実行する処理を解決（静的ディスパッチ）しますが、実行時までインスタンスが確定しない処理では動的ディスパッチも使うことができる

```rust
trait UmaMusume {
    fn victory_voice(&self);
}
// トウカイテイオー
struct TokaiTeio;
// ダイワスカーレット
struct DaiwaScalet;
impl UmaMusume for TokaiTeio {
    fn victory_voice(&self) {
        println!("よゆーよゆー、次も1着取るもんね！")
    }
}
impl UmaMusume for DaiwaScalet {
    fn victory_voice(&self) {
        println!("やっぱりアタシが一番なんだから！")
    }
}
fn main() {
    let teio = TokaiTeio {};
    let dasuka = DaiwaScalet {};
    let uma_musume_vec: Vec<Box<dyn UmaMusume>> = vec![Box::new(teio), Box::new(dasuka)];
    for uma_musume in uma_musume_vec {
        // どのインスタンスかコンパイル時に確定しない
        uma_musume.victory_voice();
    }
}
```

マーカートレイト

メソッドを持たないトレイトで

- 値の所有権を渡す代わりに値のコピーを行う `Copy` トレイト
- スレッド間で安全に参照を共有できる `Sync` トレイト

などがあります。

コンパイラが安全性の検査や最適化をするのに使用します。

お品書き

- 自己紹介
- Rustとは
- 実行速度
- **開発パフォーマンス**
- Rustのユースケース

開発パフォーマンス

ツール群

- rustup(ツールチェイン)
- rustc(コンパイラ)
- cargo(ビルドシステム&パッケージマネージャー)
- rustfmt(標準formatter)
- clippy(標準linter)

エディタ

VSCode or CLion

マクロ

標準マクロが提供されています。自分で`macro_rules!`で宣言的マクロ、`#[derive]`アトリビュートで手続き的マクロを定義することもできます。
（マクロとはメタプログラミングの文脈で、他のコードを記述するコードを書く術を指します）

```
format!   // フォーマット文字列
cancat!   // リテラル結合
println!  // 標準出力
vec!      // ベクター型の宣言
```

公式ドキュメント「19.5.マクロ」

お品書き

- 自己紹介
- Rustとは
- 実行速度
- 開発パフォーマンス
- **Rustのユースケース**

Rustのユースケース

メモリ安全性・スレッド安全性と実行速度を生かして

- CLI
- ネットワーク
- 組み込み

で利用されることが多い

プロダクション利用事例

web server side

最速のwebフレームワークであるC++のDragonに次ぐ速度のactix-webをはじめとしていくつかのフレームワークがある。

webフレームワーク比較

AtCoder Problems

WebAssembly

WebAssemblyに対応しているのでブラウザ上でも動作します。

GCが無いのでファイルサイズが小さくなります。(ベンチマーク参考)

WebAssembly - Rust プログラミング言語

Rust から WebAssembly にコンパイルする - WebAssembly | MDN

FEフレームワークもある

yewstack/yew

参考資料

公式ドキュメント

実践Rustプログラミング入門