rfmt Ruby Layer

fujitani sora

fujitani sora

2001（24）

toridori inc engineer

@_fs0414

github.com/fs0414

sorafujitani.me

rfmt

- 高速なRuby code formatter
- RubyGems Rust Extentionを使用し、Code ModuleをRustで実装
  - https://bundler.io/blog/2023/01/31/rust-gem-skeleton.html
- GitHub: https://github.com/fs0414/rfmt
  - いまStrt数55くらい
- RubyGems: https://rubygems.org/gems/rfmt
  - 資料書いてる時で9937 installs
- Zenn: https://zenn.dev/soramarjr/articles/0b2464bc09b643

Architecture

- **Ruby Layer**
  - CLI, LSP Integration
  - Config
  - Cache
  - PrismBridge Module によるAST Parse
- **FFI Boundary**
  - Magnus + rb_sys で Ruby ↔ Rust をJSON経由で接続
- **Rust Layer**
  - Emitterによる具体的なformat処理

今日はこのRuby layerの全体的な話

データフロー

1. **Ruby source code** を受け取る
2. Prism でパースして **AST** を取得
3. PrismBridge で AST を走査し、Rust が処理できる **JSON** に変換
4. JSON を FFI 経由で Rust に渡す
5. Rust から返ってきた結果をファイル書き込み or 標準出力

PrismBridge

AST parseとドメインモデル変換

PrismBridge とは

`lib/rfmt/prism_bridge.rb` / `lib/rfmt/prism_node_extractor.rb`

- source codeを入力にPrism parserを呼び出し、ASTを受け取る
- 各ノードからRfmt内部が利用したいドメインモデルに合わせたメタデータを抽出
  - クラス名、メソッド名、パラメータ数、etc.
  - ロケーション情報 (行番号、カラム、オフセット)
- コメント情報の収集とシリアライズ

PrismBridgeの中間層を入れることで、Prismとの依存を吸収し、Parserを差し替えられる設計

PrismとのIntegration

- PrismはRuby標準のパーサーGem
- Ruby側でPrismを呼ぶのが最も自然
- RustからPrismを呼ぶにはRuby VMを経由するFFIが必要で複雑になる
- Ruby側でPrism ASTを走査し、Rust側が処理しやすい形式への事前処理を行う役割

`lib/rfmt/prism_bridge.rb`

```
def self.parse(source)
  result = Prism.parse(source)
  handle_parse_errors(result) if result.failure?
  serialize_ast_with_comments(result)
end
```

AST変換 — 具体例

このRubyコードを入力すると…

```ruby
def greet(name)
  puts "Hello, #{name}"
end
```

PrismBridge がこのような JSON に変換する

```json
{
  "node_type": "def_node",
  "metadata": {
    "name": "greet",
    "parameters_count": "1"
  },
  "children": [
    { "node_type": "required_parameter_node" },
    { "node_type": "statements_node" }
  ]
}
```

Rust 側はこの JSON を受け取り、AST として再構築してフォーマットを行う

Foreign Function Interface

Magnus — Ruby bindings for Rust

- Rust で Ruby の拡張 gem を書くためのライブラリ
- Rust の関数を Ruby のメソッドとして公開できる
- Ruby ↔ Rust 間の型変換を自動で処理
- 引数のバリデーションやエラーハンドリングも Ruby の慣習に沿って動作
- rfmt では Magnus 経由で Rust のフォーマッタを Ruby から呼び出している

Ruby-Rust 間のFFI境界

```ruby
# Ruby側 (lib/rfmt.rb)
def self.format(source)
  prism_json = PrismBridge.parse(source)
  format_code(source, prism_json)
end
```

```rust
// Rust側 (ext/rfmt/src/lib.rs)
#[magnus::init]
fn init(ruby: &Ruby) -> Result<(), Error> {
    let module =
      ruby.define_module("Rfmt")?;

    module.define_singleton_method(
      "format_code",
      function!(format_ruby_code, 2))?;
    module.define_singleton_method(
      "parse_to_json",
      function!(parse_to_json, 1))?;
    module.define_singleton_method(
      "rust_version",
      function!(rust_version, 0))?;
    Ok(())
}
```

- Magnus crate による Ruby-Rust FFI
- Ruby から呼べるフォーマット・パース・バージョン取得の3つの関数を公開
- Ruby と Rust 間のデータ型変換は Magnus が自動で処理

Command Line Interface

CLI の概要

`lib/rfmt/cli.rb`

- Thor ベースのCLI — Thor gem で宣言的なコマンド定義
- コマンド: format / check / version / config / cache / init
- 並列処理の自動判定 (ファイル数・サイズに基づくヒューリスティクス)
  - 余談で、10fileほどであれば並列化しない方が速い
- diff表示: diffy / diff-lcs gem (unified / side_by_side / color)
- プログレス表示

CLI の処理フロー

1. `rfmt [FILES]` コマンド実行
2. YAML設定ファイルを探索・読込 → 対象ファイルを展開 → キャッシュで変更有無を判定
    - ファイル数やサイズに応じて並列/逐次を自動判定
3. フォーマット実行 — 逐次 or 並列処理
    - 各ファイルを読み込み → フォーマット → 結果を比較
4. 結果処理 — 書き込み / diff表示 / check結果の出力 → キャッシュ更新

Configuration & Cache

Configuration & Cache

Configuration - 設定管理

`lib/rfmt/configuration.rb`

- YAML設定ファイルの探索・読込・バリデーション
- ファイルglobパターンによるinclude/exclude
- デフォルト設定とのマージ

Ruby の得意分野: YAML パース、Dir.glob によるファイル探索

Cache - キャッシュシステム

`lib/rfmt/cache.rb`

- mtime (ファイル更新日時) ベースの変更検知
- ~/.cache/rfmt/cache.json にJSONで永続化
- clear / prune / stats 操作

低頻度・軽量処理のためRubyで十分な速度

ネイティブ拡張 & エディタ連携

Gem配布 & ネイティブ拡張ロード

`lib/rfmt/native_extension_loader.rb`

- rfmt.gemspec + ext/rfmt/extconf.rb による標準的なnative extension gem構造
- gem install rfmt で Rust 拡張含めてビルド&インストール
- rb_sys / magnus による Ruby-Rust FFI の標準的なパターン
- Ruby 3.0〜3.3+ のバージョン別パス対応

ビルド: extconf.rb → Rust の Makefile を生成 → Cargo でリリースビルド → 共有ライブラリ (.bundle / .so) を出力

ロード: Ruby のバージョンに応じたパスから共有ライブラリを自動検出して読み込み

Ruby LSP Integration

`lib/ruby_lsp/rfmt/addon.rb` / `lib/ruby_lsp/rfmt/formatter_runner.rb`

- Ruby LSP の Addon として登録
- format-on-save で rfmt を呼び出し
- FormatterRunner インターフェースに準拠

Ruby LSP の Addon は Ruby で書く必要があるため、Ruby Layer に実装

E2E テスト

テスト構成

- **フォーマットテスト** — Rubyコードを入力し、期待する整形結果と比較
  - 条件分岐、ループ、ブロック、rescue、lambda、パターンマッチなど構文ごとに網羅
- **設定テスト** — YAML設定の探索・読込・親ディレクトリからの継承を検証
- **CLIテスト** — コマンド実行の正常系・異常系
- **LSP連携テスト** — Ruby LSP Addon の登録と format-on-save の動作
- **ネイティブ拡張テスト** — Rubyバージョン別のロードパス切り替え

Ruby テストと Rust テスト (`cargo test`) を `rake dev:test_all` で一括実行

テストの具体例

```
it 'formats if with elsif and else' do
  source = "if x > 0\nputs \"positive\"\nelsif x < 0\n…"

  result = Rfmt.format(source)

  # フォーマット後、正しくインデントされていることを検証
  expect(result).to eq(expected)
end
```

入力のRubyコードをフォーマットし、期待するインデントや構造と一致するかを検証

まとめ: Ruby Layer の設計思想

- **境界の明確さ**: Ruby = パース + I/O + ユーザーインターフェース、Rust = AST処理 + コード生成
- **Prism活用**: Rubyの公式パーサーをRuby側で呼び、JSONでRustに渡す
- **Gemエコシステム**: Thor, diffy, parallel, ruby_lsp などのGemを活用
- **実用性重視**: 実際のformatなどの計算負荷の高い処理をRustで、それ以外のエコシステム連携や開発者とのInterfaceをRubyで実装

see you later 👋

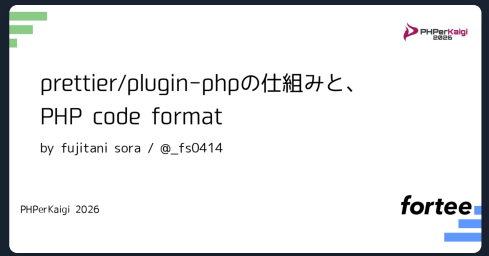
Rails Girls Tokyo #18

コーチで参加するよ

https://railsgirls.com/tokyo-2026-02-13

PHPerKaigi 2026

Day1に登壇するよ



rfmt

GitHub Starしてね

https://github.com/fs0414/rfmt