@IT
atmarkIT

Web エンジニアからみた Rust、Rust で始める Web アプリケーション

藤田直己 , paiza [著]

なぜわざわざ学習コストを払ってまで Rust を採用するのか？ Web エンジニア目線で Rust を考察

Web 開発者としての興味、関心に基づき Rust を端的に紹介し、その強みや弱みについて理解を深める本連載。第 1 回では、Rust を採用するモチベーションとは何かを整理、考察します。

（2021 年 09 月 30 日）

Rust への関心の高まり

近年、プログラミング言語「Rust」に関する重要なニュースを多く見るようになりました。例えば以下のような記事です。

Android の開発へ「Rust」を導入、なぜなのか

Microsoft、「Rust for Windows v0.9」を公開

実装言語を「Go」から「Rust」に変更、ゲーマー向けチャットアプリ「Discord」の課題とは

これらの記事が指し示すことは、いわゆる「GAFAM」（Google、Amazon.com、Facebook、Apple、Microsoft）と呼ばれる米国主要 IT 企業および先進的な Web ベンチャー企業がプロダクションユースとして Rust を採用し始めており、さらに継続的投資意欲があることを示していると考えられます。他にも Rust の勢いを示すものとして以下のような記事があります。

「Rust」はなぜ人気があるのか、Stack Overflow がユーザーのコメントを紹介

Stack Overflow における最も愛される言語の 1 位に Rust が君臨しています。2016 年から 2020 年まで 5 年連続で Rust が 1 位をキープし続け、さらには 2020 年の結果では非常に勢いのある言語である 2 位の TypeScript に 19 ポイントという大差をつけて 1 位を獲得しており、エンジニアコミュニティーにおいても極めて勢いがある言語であるように感じられます。

これらは単なる一過性のブームなのでしょうか。手慣れた既存言語ではなく、新しい言語である Rust を利用、学習するのはなぜでしょうか。第 1 回では Rust を採用するモチベーションとは何かを整理、考察します。

Rust と C/C++

Rust はその開発のいきさつからして、C/C++ からの移行が意識されています。Rust が出現するまでは、C/C++ はハイパフォーマンス（最速、最高効率）なアプリケーションを開発するならば避けて通れない言語だとされてきました。一方で C/C++ では、言語規格に動作の定義がない命令を記述することができてしまいます。その命令の結果生じる動作のことを、未定義動作と呼びます。未定義動作を引き起こす命令の典型例としては以下のようなものがあります。

1. すでに解放されているメモリ領域を参照する（use after free）
2. すでに解放されているメモリ領域をさらにメモリ解放する（double free）
3. 無効なメモリ領域を指しているポインタ（ダングリングポインタ）を参照解決する
4. 配列の範囲外アクセス
5. 複数スレッドによるあるメモリの同時参照・更新（データ競合） etc.

未定義動作の典型例は（厳密には予測不能ですが例えば）以下のようなものがあります。

1. 本来アクセスできないデータを参照・更新できてしまう（不正アクセス）
2. OS のメモリ保護機構によりメモリの不正アクセスを検出しプログラムが停止する（segmentation fault）

etc.

```c
// deref_null_pointer.c
#include <stddef.h>
int main(void)
{
  int *null_pointer = NULL;
  int value = *null_pointer; // 未定義動作発生
}
```

【未定義動作を起こす極めて単純な C のコード】このコードの問題は明白ですが、現実の問題はここまで簡単ではないはずです

```
~/p/rust_the_book ❯❯❯ gcc deref_null_pointer.c
~/p/rust_the_book ❯❯❯ ./a.out
[1]    3745754 segmentation fault (core dumped)  ./a.out
~/p/rust_the_book ❯❯❯
```

【C のコードの実行例】この実行では OS のメモリ保護機構により segmentation fault となりましたが、いつでもこの結果になるとは限りません

```rust
// deref_null_pointer.rs
fn main() {
    let p: *const i32 = std::ptr::null(); // null pointerそのものは安全である
    unsafe { // unsafeで囲まないとコンパイルエラー
        println!("{}", *p); // 未定義動作発生
    }
}
```

【未定義動作を起こす Rust のコード】unsafe を使うと Rust コンパイラの安全性保証から外れてしまう一方、C 同等の自由度を獲得できます

未定義動作を引き起こすプログラムは安全とは言えません。例えば、セキュリティ問題や障害を引き起こす可能性があります。未定義動作が生じないことを C/C++ コンパイラは（警告は出せますが）保証しないので、プログラマーがコードを確認し保証する必要があります。未定義動作は、コンパイラによる最適化などにより予期せぬ振る舞いや非決定論的な振る舞いを生じることも多くデバッグも困難なものになりがちです。言語仕様に精通したプログラマーの尽力による問題解決が必要になります。

Rust は、unsafe という言語機能を用いない限り未定義動作を引き起こし得る命令を記述しようとしてもコンパイル時または実行時エラー（パニック）になります。もしパニックとなったときでも決定論的に振る舞うのでデバッグがしやすく、なおかつ C/C++ に比肩するパフォーマンスを同時に有しています。

※この記事では上述の「未定義動作を引き起こす命令」のうち 1、2、3、4 のような命令から保護されている、あるいは適切な動作が規定されていることをメモリ安全、5 のような命令から保護されていることをスレッド安全と呼ぶことにします。

Web 開発者の視点から見た Rust

Rust はその開発のいきさつから C/C++ との比較を通じて語られることが多いですが、Web 開発において C/C++ を採用することはまれで、メリットと呼ばれていることがそんなに大したことではないように思えるかもしれません。

実際のところ、Ruby、Python、JavaScript といった典型的な Web 開発言語を用いる限り、メモリ管理を言語処理系に委譲できるため use after free を気にする機会はありません。また、配列の範囲外アクセスをすると実行時エラーにはなりますが言語に備わっている例外機構により守られます。動作は言語の定義する範囲内に収まり、決定論的な振る舞いを示すことが多いので、未定義動作を相手にするのに比べるとデバッグするのも比較的容易です。

Rust と典型的な Web 開発言語の安全性や開発生産性に関するさらなる考察は次回以降改めて検討することにして、ここではパフォーマンスに着目することにします。

伝統的な Web アプリケーションは計算時間よりもネットワーク I/O がボトルネックになることが多く、複雑なデータクエリ処理はデータベースに処理を委譲するよう設計することが基本とされています。そのため、Web アプリケーション自体の計算パフォーマンスはそれほど重視されない傾向にあります。そしてそれが比較的低速な言語で Web 開発してもよい理由であり、C/C++ が採用されてこなかった理由だと考えられます。

一方で Web 開発の文脈において計算パフォーマンスが改善されるとどのようなメリットがあるのかを思い付く範囲で列挙してみます。

1. メモリ使用量の低減→アプリケーション安定性の向上
2. バッチ計算時間の短縮→アプリケーション安定性、計算スループットの向上
3. 必要となるクラウドリソースが少なくできる→運用コスト削減
4. 計算時間改善→事前計算テーブルが不要に→技術的負債の削減
5. 計算時間改善→サーバレスポンスタイム改善→ユーザーエクスペリエンスの改善

安全性とパフォーマンスを両立させる Rust を使用することによって、「サービスの提供価値」「サイトリライアビリティ」「ユーザーエクスペリエンス」「運用コスト」など、今日のサービス開発における重要なファクターを安全性を確保しながら改善していけることが期待できます。一方で開発生産性への影響などは客観的に語ることがかなり難しいですが、次回以降検討を深めます。

Rust の卓越性、あるいは特異性

この世にはすでにたくさんのプログラミング言語があります。その中で Rust の卓越性、あるいは特異性とも呼ぶべき要素について主観的な説明を交えて紹介します。

Rust 以前の代表的な言語はメモリ安全性と GC（ガベージコレクタ）が不可分なものとして扱われており、メモリ安全性を得るためには GC を使わなければならないという関係にありました。でなければ、人間がメモリ安全であることを保証する必要があります。

一方で Rust は「所有権システム」と呼ばれるモデルを採用し、コンパイル段階でメモリ安全性を検証することで、GC がオプトアウトできることを示しました。さらにその所有権モデルはコンパイル段階でのスレッド安全性を検証することにも利用できて、データ競合も合わせて検出します（これは実に驚くべきことで、私が Rust を学習し始めた重要なモチベーションの一つになります）。

所有権システムが Rust の卓越性の根幹となるものと筆者は考えていますが、同時に Rust は現代的な言語であり、さまざまな点において妥協のない言語設計になっていると強く感じます。ドキュメンテーション、依存性管理、テスト、エラーハンドリング、合理的な構文――これらのトピックに関する明確な答えを Rust は提示していて、それらもまた Rust の卓越性を構成しています。

まとめ

第 1 回では、Rust へのモチベーションと題して、Rust の魅力や Web 開発者目線での Rust のメリットについて提示しました。第 2 回では、Python と Rust の構文比較をしながら、パフォーマンスについて簡易なベンチマークを取ることを予定しています。そのままパフォーマンス比較をすると自明で退屈な結果になるため、Python 側も少し工夫をしてもう一歩踏み込んだ比較をしていきます。

Rust は本当に動作が高速なのか？
Python との比較で分かる、Rust のパフォーマンス特性

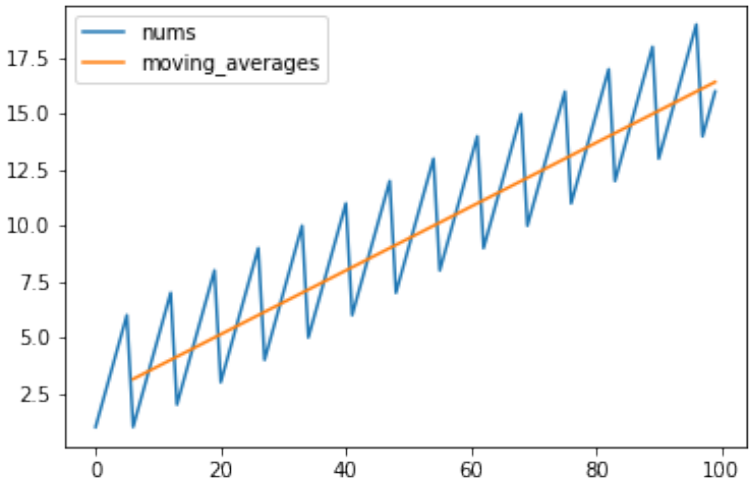
Web 開発者としての興味、関心に基づき Rust を端的に紹介し、その強みや弱みについて理解を深める本連載。第 2 回は、Python との比較を通じて Rust のパフォーマンス特性を整理、考察します。

（2021 年 12 月 01 日）

今回は、Rust のパフォーマンス特性を理解し、Python との比較を通じて Rust の構文、記述性を簡潔に紹介します。そのために構文、パフォーマンスを比較するための課題（要件）を設け、それぞれの言語でどのようなプログラムになるのかを確認していきます。いろいろと高速化させたり要件を変化させたりすることで、改めて Rust の強み（あるいは Python の強み）を浮き彫りにしていきます。この記事を作成するに当たり関連コードを収録した Github リポジトリも用意しましたので、検証したい方はぜひご利用ください。

課題設定

昨今のコロナ禍の情勢において、感染者数の 7 日間移動平均というデータをニュースでよく見掛けます。N 日間移動平均とは、ある日次（時系列）データに対して、直近 N 日間の平均を計算して得られるデータのことです。例えば [10,20,30,40] という日次データに対して、2 日間移動平均を計算すると、[（10+20）/2,（20+30）/2,（30+40）/2]=[15,25,35] というデータが得られます。



周期 7 の摂動（変動）を持つ時系列データと 7 日間移動平均：7 日間移動平均を取ることで曜日ごとの摂動を時系列データから取り除くことができます

大した計算ではないですが、考察を深める上で良い題材なのでこれを課題にします。今回のプログラムの要件をひとまず以下のように設定します（後で少し変更します）。

1. 64bit 浮動小数点で表現できる 1000 万個のデータ点がある時系列データの csv を用意
2. この 7 日間移動平均を計算し、メモリに保持する

パフォーマンス比較の形式

Python や Rust には「line_profiler」や「criterion-rs」など、それぞれ優れたプロファイリングのためのライブラリやツールがあります。基本的にはこれらのツールを使うべきですが、計測自体が計測対象に影響を与えてしまいますし（特に line_profiler）、異なる言語間で比較する必要があるため、経過時間を print 出力する方式で進めます。環境によって結果は異なるのですが、参考までに筆者のプログラムの実行環境を以下に記載しました。

- OS：ArchLinux（kernel 5.7.10-arch1-1）
- CPU：AMD Ryzen 9 3950X 16-Core Processor
- RAM：G.Skill F4-3200C16-32GVK×4（DDR4-3200 32GB×4）
- SSD：Crucial CT1000MX500SSD1（1000GB Serial ATA 6Gb/s）

最適化無しの Python で実装する

まずは最適化一切なしの純粋な Python で要件を満たすためのコードを記述しました。Python を利用したことがない人でも雰囲気がつかめるよう、コードにコメントを付記しています。

```python
import csv
import math
import sys
import psutil
from datetime import datetime
from typing import List
import pandas as pd
def process_memory_usage_mb():
    """
    実行プロセスのメモリ使用量を取得する(単位はMB)
    """
    return psutil.Process().memory_info().rss/1e6
def read_csv(relative_path):
    res = []
    with open(relative_path) as f:
        reader = csv.reader(f)
        next(reader) # ヘッダをskipする
        for row in reader:
            res.append(float(row[0])) # csvの全行をfloat型に変換して読み込む
    return res

def calc_batch_list(calc_strategy, average_length) -> List:
    """
    csvからPythonのlistを読み込み、バッチ的に移動平均を計算させる
    """
    before_read = datetime.utcnow()  # データ読み込み前の時刻記録
    nums = read_csv("../data/time_series.csv") # データ一括読み込み
    after_read = datetime.utcnow()  # データ読み込み後の時刻記録
    moving_averages = calc_strategy(nums, average_length)  # 関数を用いて移動平均計算
    after_calc = datetime.utcnow()  # 移動平均計算後の時刻記録
    print(f"移動平均計算に使用した関数：{calc_strategy}")
    print(f"移動平均の長さ：{average_length}")
    print(f"移動平均の最後の要素：{moving_averages[-1]}")
    print(f"csvロードにかかった時間：{after_read - before_read }秒")
    print(f"移動平均計算にかかった時間：{after_calc - after_read}秒")
    print(f"リストのメモリ使用量(参考)：{sys.getsizeof(moving_averages)/1e6}MB")
    print(f"プロセスメモリ使用量(参考)：{process_memory_usage_mb()}MB")
    return moving_averages
def moving_average_batch_python(nums: List, average_length: int) -> List:
    """
    Pythonのlistを使う、移動平均を素直に計算する
    """
    assert len(nums) - average_length + 1 > 0  # データが不足する場合は例外を送出する
    # 直近N日間の総計を計算しそれをNで割る
    res = [sum(nums[i-average_length+1:i+1]) / average_length for i in range(average_length-1, len(nums))]
    return res
if __name__ == "__main__":
    ma1 = calc_batch_list(calc_strategy=moving_average_batch_python, average_length=7)
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python（Naive,7）	3.3秒	2.6秒	**5.9秒**	89MB	870MB

1000 万行のデータを処理したことを考えれば決して悪くはない結果です。6 秒以内で処理を完結させることができます。

Rust で実装する

次に Rust で要件を満たすためのコードを記述しました。エラーを扱う上で便利な「anyhow」など幾つか外部のクレート（ライブラリ）を使用しています。

```rust
// リポジトリのルートディレクトリを起点とした絶対パスを取得する（Github参照）
fn get_csv_path(relative_path: &str) -> std::path::PathBuf {
    let project_path = env!("CARGO_MANIFEST_DIR"); // RustのプロジェクトファイルCargo.tomlがあるディレクトリ
    std::path::Path::new(project_path)
        .parent() // project_pathの1つ上のディレクトリ(＝リポジトリのルート)
        .unwrap()
        .join(relative_path)
}
fn read_csv(relative_path: &str) -> anyhow::Result<Vec<f64>> {
    let csv_path = get_csv_path(relative_path); // csvデータの絶対パスを取得する
    let mut csv_reader = csv::Reader::from_path(csv_path)?;
    let nums = csv_reader
        .deserialize::<f64>() // 何もしないと行データは文字列として読み込まれるので、f64に変換する
        .filter_map(|row_result| row_result.ok()) // f64として読み込めなかった行を無視する
        .collect::<Vec<_>>(); // 可変長配列に格納する
    Ok(nums)
}
fn moving_average_batch_naive(nums: &[f64], average_length: usize) -> anyhow::Result<Vec<f64>> {
    let size = nums.len() as i64 - average_length as i64 + 1; // 出力される移動平均の数列のサイズ
    if size <= 0 {
        // サイズが0以下ならばエラー値を関数の戻り値として返す
        return Err(anyhow::anyhow!(
            "average length must be less than nums array length"
        ));
    }
    let averages = nums
        .windows(average_length) // 直近N個のデータを記憶しながらループを回す
        .map(|window| window.iter().sum::<f64>() / (window.len() as f64)) // 直近N個のデータの総和をとり、Nで割る
        .collect::<Vec<_>>(); // 結果を可変長配列に格納する
    Ok(averages) // 可変長配列を関数の戻り値として返す、returnは省略している
}
```

```rust
fn calc_batch<F: FnOnce(&[f64], usize) -> anyhow::Result<Vec<f64>>>(
    strategy: F,
    average_length: usize,
) -> anyhow::Result<Vec<f64>> {
    let before_read = chrono::Utc::now(); //  データ読み込み前の時刻記録
    let nums = read_csv("data/time_series.csv")?; // 指定したcsvデータをf64の可変長配列として読み取る
    let after_read = chrono::Utc::now(); // データ読み込み後の時刻記録
    let moving_averages = strategy(&nums, average_length)?; // 関数を用いて移動平均計算
    let after_calc = chrono::Utc::now(); // 移動平均計算後の時刻記録
    println!(
        "移動平均計算に使用した関数：{:?}",
        std::any::type_name::<F>()
    );
    println!("移動平均の長さ：{}", average_length);
    println!(
        "移動平均の最後の要素：{:?}",
        moving_averages[moving_averages.len() - 1]
    );
    let load_time = after_read - before_read;
    let calc_time = after_calc - after_read;
    println!(
        "csvロードにかかった時間：{:?}秒",
        load_time.num_nanoseconds().unwrap() as f64 / 1e9
    );
    println!(
        "移動平均計算にかかった時間：{:?}秒",
        calc_time.num_nanoseconds().unwrap() as f64 / 1e9
    );
    println!(
        "Vecの使用メモリ量(参考)：{:?}MB",
        std::mem::size_of_val(&*moving_averages) as f64 / 1e6
    );
    println!(
        "プロセスの使用メモリ量(参考)：{:?}MB",
        psutil::process::Process::new(std::process::id())
            .unwrap()
            .memory_info()
            .unwrap()
            .rss() as f64
            / 1e6
    );
    Ok(moving_averages)
}
fn main() -> anyhow::Result<()> {
    let ma1 = calc_batch(moving_average_batch_naive, 7)?;
    Ok(())
}
```

```rust
fn read_csv(relative_path: &str) -> anyhow::Result<Vec<f64>> {
    let csv_path: PathBuf = get_csv_path(relative_path);
    let mut csv_reader: Reader<File> = csv::Reader::from_path(csv_path)?;
    let nums: Vec<f64> = csv_reader: Reader<File>
        .deserialize::<f64>(): DeserializeRecordsIter<File, ···>
        .filter_map(|row_result: Result<f64, Error>| row_result.ok()): impl Iterator<Item = f64>
        .collect::<Vec<_>>();
    Ok(nums)
}

fn calc_batch<F: FnOnce(&[f64], usize) -> anyhow::Result<Vec<f64>>>(
    strategy: F,
    average_length: usize,
) -> anyhow::Result<Vec<f64>> {
    let before_read: DateTime<Utc> = chrono::Utc::now();
    let nums: Vec<f64> = read_csv(relative_path: "data/time_series.csv")?;
    let after_read: DateTime<Utc> = chrono::Utc::now();
    let moving_averages: Vec<f64> = strategy(&nums, average_length)?;
    let after_calc: DateTime<Utc> = chrono::Utc::now();
```

Rust の優れた開発体験：Rust はほとんど型推論される。型推論の結果は rust-analyzer という開発ツールを用いると非常に簡単に確認できる

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Rust（Naive,7）	0.65秒	0.047秒	**0.7秒**	80MB	162MB

純粋な Python 実装と比べて処理が高速化されています。また、メモリ使用量についてもかなりの部分が説明可能であることも分かります。f64（8 バイト）のデータが 1000 万行あるので、1000 万 ×8 バイト =80MB で、それが nums 変数と moving_averages 変数に保持されているため 160MB が必要最小限のメモリと計算できますが、それに非常に近い値となっています。

ソースコードの面で比較してみると、Rust のコードは Python に比べてやや冗長です。冗長となっている理由を簡単にまとめると、以下のようになります。

- 型を明示する必要がある（関数の入出力、計算時の細かな型変換など）
- エラー値の処理を細かく記述している（読み込めない行があった場合どうするか、移動平均を計算できない場合どうするかなど）
- （抽象化の余地があり、大した差ではないが）相対パスの解決のために get_csv_path 関数を作っている

これらを除外して見比べてみると、Rust は Python と非常に似通ったプログラム構造で記述可能であることが分かります。

Numpy を利用した Python で実装する

ある程度データ分析に習熟した Python 使用者にとって、上記の比較はフェアではないと思うことでしょう。Python が機械学習分野という多くの計算量を必要とする分野で利用され続けている理由は、それを支えるエコシステムがあるからです。その代表格といえる「Numpy」を利用して、要件を満たすコードを記述します（一度書いた関数は再利用します）。

```
1.  def calc_batch_ndarray(calc_strategy, average_length) -> np.ndarray:
2.      before_read = datetime.utcnow()  # データ読み込み前の時刻記録
3.      nums = pd.read_csv("../data/time_series.csv")['value'].values  # データ一括読
    み込み
4.      after_read = datetime.utcnow()  # データ読み込み後の時刻記録
5.      moving_averages = calc_strategy(nums, average_length)  # 関数を用いて移動平均
    計算
6.      after_calc = datetime.utcnow()  # 移動平均計算後の時刻記録
7.      print(f"移動平均計算に使用した関数:{calc_strategy}")
8.      print(f"移動平均の長さ:{average_length}")
9.      print(f"移動平均の最後の要素:{moving_averages[-1]}")
10.     print(f"csvロードにかかった時間:{after_read - before_read }秒")
11.     print(f"移動平均計算にかかった時間:{after_calc - after_read}秒")
12.     print(f"配列のメモリ使用量(参考):{moving_averages.nbytes/1e6}MB")
13.     print(f"プロセスメモリ使用量(参考):{process_memory_usage_mb()}MB")
14.     return moving_averages
15. def moving_average_batch_numpy(nums: np.ndarray, average_length: int) ->
    np.ndarray:  # numpy arrayを使う、convolve APIを使う
16.     assert len(nums) - average_length + 1 > 0
17.     return np.convolve(nums, np.ones(average_length), 'valid') /
    average_length
18. if __name__ == "__main__":
19.     ma1 = calc_batch_ndarray(calc_strategy=moving_average_batch_numpy,
    average_length=7)
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python+Numpy（Naive,7）	0.55秒	0.029秒	**0.58秒**	80MB	227MB

プロセスメモリ使用量を除いて、Rust 実装と比べても良い結果が得られてしまいました。難しいテーマですが、このパフォーマンスについて考察してみます。Numpy ではベクター計算を高度に最適化してくれますが、筆者が用意した Rust コードではそのような最適化を含んでいないことが理由の一つだと考えられます。Rust コードに最適化の余地があるわけですが、本記事では計算のユースケースや汎用（はんよう）性に着目して議論を進めたいと思います。

移動平均の長さを変えて比較する

上記までの計算では移動平均の長さを 7 と固定していました。この要件を変更して 5000 に変更してみるとどうなるでしょうか？

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python+Numpy（Naive,5000）	0.53秒	5.77秒	**6.3秒**	80MB	227MB

Numpy による最適化の恩恵を受けても 6 秒近くの計算時間がかかりました。というのも上記までの計算手法では、移動平均の長さが大きくなるほど、必要となる足し算の回数が増えるからです（1 ループ当たり 6 回→ 4999 回）。

移動平均を計算するに当たって、もし 1 日前の和を参照できれば、今日の和 = 前日の和 + 今日のデータ -N 日前のデータという関係性が成り立つので計算量を落とすことができます。この考え方に基づき Rust コードを書き換えてみます。

```rust
fn moving_average_batch_online(nums: &[f64], average_length: usize) -> anyhow::Result<Vec<f64>> {
    let size = nums.len() as i64 - average_length as i64 + 1; // 出力される移動平均の数列のサイズ
    if size <= 0 {
        // サイズが0以下ならばエラー値を関数の戻り値として返す
        return Err(anyhow::anyhow!(
            "average length must be less than nums array length"
        ));
    }
    let mut res = Vec::with_capacity(nums.len());
    // 直近N個のデータの総和を計算する、最初のデータは素直に総和をとる
    res.push(nums[0..average_length].iter().sum::<f64>());
    // 最初のデータ以外は前の総和から新しいデータを足して、古いデータを引くことで計算し、計算量を減らす
    for i in average_length..nums.len() {
        res.push(nums[i] as f64 - nums[i - average_length] as f64 + res[i - average_length])
    }
    // 出力するデータをNで割る、後で割るのは計算誤差を小さくするため
    for i in 0..(nums.len() - average_length + 1) {
        res[i] /= average_length as f64;
    }
    Ok(res)
}
fn main() -> anyhow::Result<()> {
    let _ma = calc_batch(moving_average_batch_online, 5000)?;
    Ok(())
}
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Rust（Online,5000）	0.58秒	0.079秒	**0.66秒**	80MB	162MB

7 日移動平均を計算していたときとそれほど変わらない計算時間で計算を完了させることができました。

一方で Numpy の方はもともと Numpy の convolve 関数を使っていたので、そのまま書き換えることができません。for ループを使うことになりますが、Numpy でそのまま for ループを使ってしまうと基本的に遅くなってしまうので、ここでは「Numba」というライブラリを使って最適化します。

```
1.  @numba.jit(nopython=True)
2.  # numpy arrayを使う、オンラインアルゴリズムを使用、Numbaを使う
3.  def moving_average_batch_numpy_numba_online(nums: np.ndarray, average_length:
    int) -> np.ndarray:
4.      assert len(nums) - average_length + 1 > 0
5.      N_i = nums.shape[0]
6.      res = np.empty_like(nums, dtype=np.float64)
7.      res[average_length - 1] = np.sum(nums[:average_length])
8.      for i in range(average_length, N_i):
9.          res[i] = nums[i] - nums[i - average_length] + res[i - 1]
10.     for i in range(average_length-1, N_i):
11.         res[i] = res[i] / average_length
12.     return res[average_length-1:]
13.
14. if __name__ == "__main__":
15.     ma8 =
    calc_batch_ndarray(calc_strategy=moving_average_batch_numpy_numba_online,
    average_length=5000)
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python+Numpy+Numba（Online,5000）	0.55秒	0.31秒	**0.86秒**	80MB	304MB

Python を利用してもそこまで速度を落とすことなく計算が終わるようになりました。

ストリーム処理で比較する

これまでは 1000 万行のデータを全てメモリに配置しバッチ処理をさせるという要件（インタフェース）で考えていました。この場合では 1 秒当たり何行処理できるかのスループットの観点が重要であり、いずれの言語でもスループットは引き出せることが分かりました。一方で、現状のプログラムは全部の値を読み込んでから計算することを前提にしているため、処理遅延（レイテンシ）とメモリー括確保が避けられません。Web アプリケーションにおいては一括でデータを読み込まなければならないという前提はかなり不自然です。そこで以下の制約を要件に追加します。

- 中間変数（nums）を用いてデータを一括で読み込むことを禁止する（ストリーム処理で計算する）

この前提に基づき Rust コードを書き換えてみます。

```rust
#[derive(Debug, Clone)]
pub struct MovingAverage {
    period: usize,
    sum: f64,
    deque: std::collections::VecDeque<f64>,
}
impl MovingAverage {
    pub fn new(period: usize) -> Self {
        Self {
            period,
            sum: 0.0,
            deque: std::collections::VecDeque::new(),
        }
    }
    pub fn latest(&mut self, new_val: f64) -> Option<f64> {
        self.deque.push_back(new_val);
        let old_val = match self.deque.len() > self.period {
            true => self.deque.pop_front().unwrap(),
            false => 0.0,
        };
        self.sum += new_val - old_val;
        match self.deque.len() == self.period {
            true => Some(self.sum / self.period as f64),
            false => None,
        }
    }
}
fn calc_stream(average_length: usize) -> anyhow::Result<Vec<f64>> {
    let before_read = chrono::Utc::now();
    let csv_path = get_csv_path("data/time_series.csv");
    let mut csv_reader = csv::Reader::from_path(csv_path)?;
    let mut ma = MovingAverage::new(average_length);
    let moving_averages = csv_reader
```

```
            .deserialize::<f64>()
            .filter_map(|row_result| row_result.ok())
            .filter_map(|new_val| ma.latest(new_val))
            .collect::<Vec<_>>();
    let after_calc = chrono::Utc::now();
    println!("移動平均の長さ:{}", average_length);
    println!(
        "移動平均の最後の要素:{:?}",
        moving_averages[moving_averages.len() - 1]
    );
    let total = after_calc - before_read;
    println!(
        "計算にかかった時間:{:?}秒",
        total.num_nanoseconds().unwrap() as f64 / 1e9
    );
    println!(
        "Vecの使用メモリ量(参考):{:?}MB",
        std::mem::size_of_val(&*moving_averages) as f64 / 1e6
    );
    println!(
        "プロセスの使用メモリ量(参考):{:?}MB",
        psutil::process::Process::new(std::process::id())
            .unwrap()
            .memory_info()
            .unwrap()
            .rss() as f64
            / 1e6
    );
    Ok(moving_averages)
}
fn main() -> anyhow::Result<()> {
    let ma4 = calc_stream(5000)?;
    Ok(())
}
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Rust（Stream,5000）	---	---	**0.59秒**	80MB	83MB

csv の読み込み結果を一時的に受け取る中間変数 nums がなくなりました。計算時間はバッチ計算のものとほぼ同じです。また MovingAverage という構造体を用意し、移動平均の計算の本質部分を抽出しました。これはストリーム・バッチ処理を問わず利用可能な汎用的な抽象であり、つまりパフォーマンスを損ねることなくアプリケーションコードとシミュレーション（データ分析）コードを同一にできる可能性を示しています。

次に Python でストリーム処理を行うコードを記述します。

```python
class MovingAveragePython(object):
    def __init__(self, period):
        self.sum = 0
        self.period = period
        self.deque = collections.deque()
    def latest(self, new_val):
        self.deque.append(new_val)
        old_val = self.deque.popleft() if len(self.deque) > self.period else 0.0
        self.sum += new_val - old_val
        return self.sum / self.period if len(self.deque) == self.period else None
def calc_stream(constructor, average_length) -> np.ndarray:
    before_read = datetime.utcnow()  # データ読み込み前の時刻記録
    moving_averages = []
    ma = constructor(average_length)
    with open("../data/time_series.csv") as f:
        reader = csv.reader(f)
        next(reader) # ヘッダをskipする
        for row in reader:
            num = float(row[0])
            moving_averages.append(ma.latest(num))
    after_calc = datetime.utcnow()  # 移動平均計算後の時刻記録
    print(f"移動平均の長さ:{average_length}")
    print(f"移動平均の最後の要素:{moving_averages[-1]}")
    print(f"計算にかかった時間:{after_calc - before_read}秒")
    print(f"リストのメモリ使用量(参考):{sys.getsizeof(moving_averages)/1e6}MB")
    print(f"プロセスメモリ使用量(参考):{process_memory_usage_mb()}MB")
    return moving_averages
if __name__ == "__main__":
    ma3 = calc_stream(constructor=MovingAveragePython, average_length=5000)
```

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python（Stream,5000）	---	---	**6.9秒**	84MB	529MB

ストリーム処理をするということは、Python のコンテキストでループを回さざるを得なくなってしまうため、速度はどうしても落ちてしまいます。また、Numba を使ってもストリーム処理の計算時間は改善できず、また抽象化にも限界があります。バッチ処理のコードとストリーム処理のコードを抽象化することは難しく、どちらかのパフォーマンスに妥協した結果になると思われます。

それでも 1000 万行のデータの処理時間であるので、レイテンシという観点で見れば 1 つの入力当たり 69 ナノ秒で処理できているともいえます。これを高いと見るか、低いと見るかはアプリケーションの性質や計算の重さによるところが大きいでしょう。また、Github のリポジトリにチャネルという言語機能を用いたマルチスレッドの検証結果と async/await を用いた結果を試験的に追加していますが、上述のシングルスレッドの方が速いことが分かっています。これは移動平均の計算コストよりもスレッド間の通信コストの方が高価であるためだと考察できます。アプリケーション特性に応じて自在に計算モデルを選択できるのも Rust の魅力だといえるでしょう。

まとめ

今回、Python と Rust の構文比較をしながら、パフォーマンス特性について考察を深めました。結論としては以下のようになります。

- バッチ計算では Python と Rust はそこまで大きな差はない
- ただし、Python で高速化するためには Numpy、Numba のコンテキストでループを回すという制約がつく
- ストリーム計算では、Rust に優位性がある
- Rust ではパフォーマンスを損ねることなくアプリケーションコードとシミュレーション（データ分析）コードを抽象化できる

言語（計算手法,移動平均長）	csvロード時間	計算時間	総計算時間	変数のメモリ使用量	プロセスのメモリ使用量
Python（Naive,7）	3.3秒	2.6秒	**5.9秒**	89MB	870MB
Rust（Naive,7）	0.65秒	0.047秒	**0.7秒**	80MB	162MB
Python+Numpy（Naive,7）	0.55秒	0.029秒	**0.58秒**	80MB	227MB
Python+Numpy（Naive,5000）	0.53秒	5.77秒	**6.3秒**	80MB	227MB
Rust（Online,5000）	0.58秒	0.079秒	**0.66秒**	80MB	162MB
Python+Numpy+Numba（Online,5000）	0.55秒	0.31秒	**0.86秒**	80MB	304MB
Rust（Stream,5000）	---	---	**0.59秒**	80MB	83MB
Python（Stream,5000）	---	---	**6.9秒**	89MB	529MB

次回は、ここまであまり触れることがなかった Rust の難しさと開発生産性について考えてみたいと思います。

なぜ「Rust は難しい言語」とされるのか ――習得の難しさとその対策を Web エンジニアが考察

Web 開発者としての興味、関心に基づき Rust を端的に紹介し、その強みや弱みについて理解を深める本連載。第 3 回は、Rust の開発生産性を支える言語機能と難しさについて。

（2022 年 02 月 01 日）

最終回となる今回は、Rust の開発生産性を支える言語機能および難しさにフォーカスを当てて簡潔に紹介します。

開発生産性とはいうものの、この言葉は定義付けをすること自体が難しいです。下記の Rust 公式が提供するツール群は開発生産性を間違いなく向上させますが、実際に使ってみた方が理解がはかどるのでここでは紹介にとどめます。

- Rust コンパイラによるコンパイルエラーメッセージの丁寧さ
- Docs.rs のドキュメンテーション
- Cargo によるパッケージ管理（≒ Ruby の bundler、JavaScript の npm）
- rust-analyzer による強力な開発支援（≒インテリセンス）
- 言語標準のユニットテスト

端的にいえばモダン開発のプラクティスが Rust のプロジェクトでもシームレスに利用でき、簡単に開発環境を整えることができます。

下記はプロジェクトファイル（Cargo.toml）の例です。外部ライブラリなどを簡単に管理できます。

```
# Cargo.toml
# JavaのpomやC++のCMakeLists.txtを書くより圧倒的に簡単
[package]
authors = ["Naoki Fujita"]
edition = "2021"
license = "MIT"
name = "web_engineer_in_rust"
version = "0.1.0"
[dependencies]
# 利用するライブラリ(クレート)名とバージョン
# RubyのGemfileと考え方は似ている
anyhow = "1.0.52"
chrono = "0.4.19"
itertools = "0.10.3"
num-traits = "0.2.14"
[[bin]]
# エントリーポイントを作成
# cargo run --bin thread_safe_queueで実行可能
name = "thread_safe_queue"
path = "src/thread_safe_queue.rs"
[[bin]]
name = "moving_average_f64"
path = "src/moving_average_f64.rs"
[[bin]]
name = "moving_average_trait"
path = "src/moving_average_trait.rs"
[[bin]]
name = "use_trait_extension"
path = "src/use_trait_extension.rs"
```

```
~/p/r/server ❯❯❯ cargo build
             ✗ 101 +[main]
   Compiling rt v0.1.0 (/home/lb/pj/rt/server)
error[E0382]: borrow of moved value: `execs`
  --> src/exchange/bf/ws.rs:45:35
   |
27 |     let execs = stream::iter(values)
   |         ----- move occurs because `execs` has type `Vec<exec::Exec>`, whi
ch does not implement the `Copy` trait
...
43 |     dbg!(execs);
   |     ----------- value moved here
44 |     if cfg!(feature = "save_exec") {
45 |         let _ = Exec::bulk_insert(&execs, &state.pool).await?;
   |                                   ^^^^^^ value borrowed here after move
```

Rust のエラーメッセージ 1：所有権（借用規則）ルールに違反しても、コンパイラが分かりやすく指摘してくれる

この記事では開発生産性に関連する Rust の言語機能の紹介、抜粋と、Rust の難しさについての考察を主眼とします。この記事を作成するに当たり関連コードを収録した GitHub リポジトリも用意したので、自分でも検証したい方はぜひご利用ください。

スレッド安全性を型で表現する

Web 開発でよく用いられている Ruby や Python（の C 言語実装）には、グローバルインタープリタロック（GIL）という機構があり、スレッドを複数動作させていても同時に 1 つのスレッドのみがプログラムを実行できるという制約下にあります。

同時に 1 つのスレッドしか動かないのであれば、これらの言語で書かれたプログラムは常にスレッドセーフであるといえればよいのですが、スレッドセーフといえるのは単純な処理のみです。複雑な（非アトミックな）処理については、ロックを取らないと正しく動作しません。例えば、あるスレッドでオブジェクトを変更し、別のスレッドでそのオブジェクトを参照すると、例外が起こり得ます。

ドキュメントなどを見ていて「このコレクションはスレッドセーフである」や「この操作はスレッドセーフである」というような記述があるとロックを取らなくてよいと分かりますが、いつでもそのような記述があるとは限りません。スレッドセーフか否かを識別するのは大変です。

Rust はスレッド安全性をドキュメントではなく型として表現します。それによりスレッド安全でないコードをコンパイルエラーとしてはじくことができます。

この振る舞いを確認するために、少し複雑な課題を用意しました。2 つのスレッド（record_thread1,2）が観測データを記録していき、一方で 1 つのスレッド（observe_thread）がキューを定期的に観測し、その最新値を取得するというものです。

上記の要件を満たすようなサンプルコードを用意しました。

```
1.  use chrono::NaiveDateTime; // タイムゾーンなしの日時
2.  use std::sync::{Arc, Mutex}; // スレッドセーフな共有参照とMutexロックを使用
3.  #[derive(Copy, Clone, Debug)]
4.  // 観測データの構造体(データの組)を定義
5.  struct Measurement {
6.      // 日時(タイムゾーン無し)
7.      time: NaiveDateTime,
8.      // 観測値
9.      value: f64,
10.     // データを観測したスレッドID
11.     thread_id: usize,
```

```rust
}
impl Measurement {
    // Measurement構造体の関連関数(メソッド)を定義
    fn new(value: f64, thread_id: usize) -> Self {
        Measurement {
            // 生成時の現在時刻を記録
            time: chrono::Utc::now().naive_utc(),
            // 観測値を登録, キーと変数名と同じなら省略記法が使える
            value,
            // データを生成したスレッドを記録
            thread_id,
        }
    }
}
fn main() -> anyhow::Result<()> {
    // キューの生成、キューの所有権はメインスレッドが持つ
    let queue = Vec::new();
    // (1):キューのロック付(Mutex)の共有参照(Arc)を定義する
    let arc_queue = Arc::new(Mutex::new(queue));
    // キューの共有参照をコピー(そうしないと複数のスレッドからアクセスできない)
    let arc_queue1 = arc_queue.clone();
    // データ記録スレッド1を作成
    // moveで共有参照を別スレッドに移動
    // 共有参照を移動=キューの参照権限を別スレッドにも渡したと考えると分かりやすい
    let record_thread1 = std::thread::spawn(move || {
        for i in 1..=10000 {
            let m = Measurement::new(i as f64, 1);
            // (2):キューのロックを取りキューに観測値を記録
            arc_queue1.lock().unwrap().push(m);
        }
    });
    let arc_queue2 = arc_queue.clone();
    // データ記録スレッド2を作成
    let record_thread2 = std::thread::spawn(move || {
        for i in 1..=10000 {
            let m = Measurement::new(i as f64, 2);
            arc_queue2.lock().unwrap().push(m);
        }
    });
    // データ観測スレッドを作成
    // 1ミリ秒のスリープを挟みつつ、キューの最新値を出力
    let observe_thread = std::thread::spawn(move || loop {
        // ここで{}と書いてスコープを作ることには意味がある
        {
            // ロックを取得
            let queue_lock = arc_queue.lock().unwrap();
            let latest = queue_lock.last();
            println!("{:?}", latest);
            // スコープによりここでqueue_lock変数が無効になる→ロックが解放される
        }
        std::thread::sleep(std::time::Duration::from_millis(1));
    });
```


→目次に戻る

```
64.     // 2つのデータ記録スレッドの終了を待つ
65.     for thread in [record_thread1, record_thread2] {
66.         let _ = thread.join();
67.     }
68.     Ok(())
69. }
```

今回特に説明したいのは、サンプルコード内（1）と（2）で表現されている部分です。

（1）は ArcMutex パターンと呼ばれているもので、これによりスレッド安全性を手軽に実現できます。今回は Vec（可変長配列）を用いましたが、さまざまなデータ構造を組み合わせられるので極めて汎用（はんよう）的です。また、（2）を見ると lock() というメソッド呼び出しがありますがこれは Mutex 型が持つメソッドであり、lock() を呼んでロックを取得しない限り内部のデータ構造にアクセスできないことを型としてうまく表現し、スレッド安全でないデータアクセスを排除します。非同期プログラムの難しさの一つを型システムが解決するというアイデアに引かれて筆者は Rust に入門しました。

Rust のトレイト 1（型要件の抽象化）

Rust のトレイトはある要件を満たす（実装している）型の総称、集合と捉えることができます。トレイトにより高度な型の抽象化が実現できます。例えば、関数の引数にトレイトを指定することで、そのトレイトを実装している具体型とその集合を引数として捉えることができます。

ここでは第 2 回のプログラムを簡略化し、トレイトを活用したプログラムに書き直してみようと思います。

```rust
#[derive(Debug, Clone)]
pub struct MovingAverage {
    period: usize,
    sum: f64,
    deque: std::collections::VecDeque<f64>,
}
impl MovingAverage {
    pub fn new(period: usize) -> Self {
        Self {
            period,
            sum: 0.0,
            deque: std::collections::VecDeque::new(),
        }
    }
    pub fn latest(&mut self, new_val: f64) -> Option<f64> {
        self.deque.push_back(new_val);
        let old_val = match self.deque.len() > self.period {
            true => self.deque.pop_front().unwrap(),
            false => 0.0,
        };
        self.sum += new_val - old_val;
        match self.deque.len() == self.period {
            true => Some(self.sum / self.period as f64),
            false => None,
        }
    }
}
fn calc_stream(average_length: usize) -> Vec<f64> {
    let input_data = 1..=10;
    let mut ma = MovingAverage::new(average_length);
    let moving_averages = input_data
        // .map(|n| n as f64) // このf64への変換を入れれば型が合うが……
        .filter_map(|new_val| ma.latest(new_val))
        .collect::<Vec<_>>();
    moving_averages
}
pub fn main() -> () {
    let ma = calc_stream(2);
    println!("{:?}", ma);
}
```

第 2 回では取り上げませんでしたがこのコードの不満点は、MovingAverage が 64bit の浮動小数点数しか受け付けないという前提があることです。上記コメントのように型変換を含めることで問題なく利用できますが、32bit 整数などのもう少し汎用的な型を受け付けるようにしたいです。こうしたユースケースにトレイトが活用できます。トレイトを利用したコードが下記になります。num_traits という便利な外部ライブラリを利用しています。

```rust
#[derive(Debug, Clone)]
// 型パラメーターTを用いる
pub struct MovingAverage<T>
where
    // TはNumトレイトを満たす型、かつf64型にキャストできる型
    // 具体的にはu8, u32, u64, f32, f64など
    T: num_traits::Num + num_traits::cast::AsPrimitive<f64>,
{
    period: usize,
    sum: T,
    deque: std::collections::VecDeque<T>,
}
impl<T> MovingAverage<T>
where
    T: num_traits::Num + num_traits::cast::AsPrimitive<f64>,
{
    pub fn new(period: usize) -> Self {
        Self {
            period,
            // Numトレイトを満たす型はzeroという関数を持つ
            // i32なら0, f64なら0.0を返すだろう
            sum: T::zero(),
            deque: std::collections::VecDeque::new(),
        }
    }
    pub fn latest(&mut self, new_val: T) -> Option<f64> {
        self.deque.push_back(new_val);
        let old_val = match self.deque.len() > self.period {
            true => self.deque.pop_front().unwrap(),
            false => T::zero(),
        };
        // self.sumは加算・減算ができなければならない
        self.sum = self.sum + new_val - old_val;
        match self.deque.len() == self.period {
            // self.sumはT型、割り算するので出力型はf64とした
            true => Some(self.sum.as_() / self.period as f64),
            false => None,
        }
    }
}
fn calc_stream(average_length: usize) -> Vec<f64> {
    // i32型の数列
    let input_data = 1..=10;
    let mut ma = MovingAverage::new(average_length);
    let moving_averages = input_data
        .filter_map(|new_val| ma.latest(new_val))
        .collect::<Vec<_>>();
    moving_averages
}
pub fn main() -> () {
    let ma = calc_stream(2);
    println!("{:?}", ma);
}
```

型パラメーター T を利用することで、具体的な型（f64）を指定しない抽象化されたプログラムを記述することが可能になります。一方で今回の移動平均の計算の実装を見ると T は何でもよいわけではなく、0 が定義されており、加算、減算ができて、f64 にキャストできる型に限定されます。トレイトにより関数の引数や構造体に求められる制約をより抽象的、包括的に記述できるのが利点です。

Rust のトレイト 2（安全な機能拡張）

他のトレイトの重要な利用方法として、既存の型にメソッドを後から安全に追加できるという特徴があります。ここでは便利な外部ライブラリである itertools を用いて、説明します。

```
1.  // (1):Rust標準ではないItertoolsトレイトをuse
2.  use itertools::Itertools;
3.  fn main() {
4.      let iter1 = vec![1, 2, 3].into_iter();
5.      let product1 = iter1
6.          // (2):Rust標準の型にメソッドを後付けで追加できる
7.          .cartesian_product(vec!['a', 'b'])
8.          .collect::<Vec<(_, _)>>();
9.      println!("{:?}", product1);
10.     // => [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
11.     let iter2 = vec![1, 2, 3].into_iter();
12.     // メソッド名が重複時はコンパイルエラーになるので安全
13.     // 下記のように明示的に呼び出す関数を指定して衝突回避可能
14.     let product2 = Itertools::cartesian_product(iter2, vec!['a',
    'b']).collect::<Vec<(_, _)>>();
15.     println!("{:?}", product2);
16.     // => [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
17. }
```

このコードの iter1 の型は IntoIter 型でこれは Rust 標準（std）で提供されている型です。この型自体は cartesian_product というメソッドを持ちません。これに対して、（1）のように Itertools というトレイトを use する（スコープに含める）ことで、Itertools トレイトが IntoIter 型に cartesian_product メソッドなどを供給し、呼び出せるようになります。

既存の構造体、あるいはクラスに後付けでメソッドを追加することは利便性が高く、「オープンクラス」と呼ばれています。一方で、そのメソッドがどこで追加されたのかを追跡することが困難であったり、同名のメソッドを複数箇所で上書きして既存挙動を容易に破壊できたりしてしまうなど非常にリスクが高く、開発のスケーラビリティを低下させてしまう手法だと考えられます。

一方で Rust のトレイトは、use を用いてメソッドの出自を明示し、同名のメソッドが重複定義された場合はコンパイルエラーとなるため、非常に安全性の高い後付けでの機能拡張を実現できます。

Rust のトレイト 3（継承を持たない）

Rust は C++ に影響を受けている言語ではありますが、Rust は構造体の継承という概念を C++ から継承しませんでした。改めて継承とは何かを考えると「ある構造体間同士の関係性を木構造（多重継承を認めるなら有向非巡回グラフ）で表現する」という制約の下にプログラムを抽象化することだと捉えられます。

しかし、木構造は一度その構造（序列）を決めてしまうとその順序関係を変更することはできないため、事前の要件定義や設計などウオーターフォールプロセスを余儀なくされると考えています。「Bird クラスは fly メソッドを持ち、ハトクラスは Bird クラスを継承する」といった序列を構築してしまうと、要件定義を進めていく中でペンギンが出てきたときにモデリングが崩壊してしまいます。

Rust でこのモデリングを考えたとしたら、例えば Flyable というトレイトを用意し、ハトは Flyable トレイトを実装するというやり方で端的に表現できます。要件定義が進んでニワトリが出てきたとしても悩むこともなく漸進的かつスケーラブルに判断し開発を進めることができます。Web エンジニアが慣れ親しんだリレーショナルデータベースと同じように集合志向でプログラムを整理できると言い換えられるかもしれません。

Rust の難しさとその対策

世間一般では、Rust は「難しい言語」という評価のようです。Rust が難しい理由は主に 4 つほど思い付きます。

型解決が難しい

静的型付け言語を習得する上でいつも思うことは、言語設計者やライブラリ設計者が考えた型とユーザーのメンタルモデルが完全に一致することはないので整合させるために学習が必要になるということです。

その点、Rust コンパイラは非常に優秀です。エラーメッセージに型解決の答えが書いてあることが多いですが、抽象化された型（つまりトレイト）を解決するのが難しく体系的な言語理解や Rust 特有のイディオムの習得が必要になると感じます。所有権（借用規則）は Rust 特有の概念で難しいとされていますが、筆者の体感では型を利用するだけならば、慣れの問題ではないかと思っています。

```
error[E0277]: the trait bound `[exec::Exec; 1048576]: arraydeque::Array` is not satisfied
  --> src/shared_state.rs:15:26
   |
15 |     pub execs: Vec<Mutex<ArrayDeque<[Exec; 1048576], Wrapping>>>,
   |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `arraydeque::Array
` is not implemented for `[exec::Exec; 1048576]`
   |
   = help: the following implementations were found:
             <[T; 0] as arraydeque::Array>
             <[T; 100] as arraydeque::Array>
             <[T; 1024] as arraydeque::Array>
             <[T; 10] as arraydeque::Array>
           and 53 others
note: required by a bound in `ArrayDeque`
  --> /home/lb/.cargo/registry/src/github.com-1ecc6299db9ec823/arraydeque-0.4.5/src/lib.rs:97:26
   |
97 | pub struct ArrayDeque<A: Array, B: Behavior = Saturating> {
   |                          ^^^^^ required by this bound in `ArrayDeque`
```

Rust のエラーメッセージ 2：入力がトレイト（境界）を満たさない場合のメッセージ例、習熟しないと解決が難しい

幅広いユースケース（組み込みシステムから Web サービスまで）

Rust は低レイヤーから高レイヤーのドメイン（関心領域）を記述できます。それ故に記述できる解像度や自由度が極めて高いです。実装上の選択肢が多く、例えば本記事で紹介したスレッド安全なキューの代わりにチャネル、async/await、ロックフリーなデータ構造を活用した実装が考えられ、選択肢がない言語に比べて悩む余地があります。

マルチパラダイム

Rust は、C++ のようにメモリやその所有権など物理的な実体を意識しつつも、関数型言語由来の高度な抽象を織り込んだ言語であり、マルチパラダイムな言語です。その奇跡的な融合こそが Rust の魅力だと思いますが、難しいと感じるところかもしれません。

本質的困難性に挑む

簡単な問題を解決するならばスクリプト言語で解決するのが楽です。あえて Rust を使うのは、難しいプログラミングを進める場合で、「人月の神話」で有名なブルックスの言葉を援用すると「本質的複雑性」が高いプログラムを書くために使われる傾向にあると推察されます。

その点で行くと Rust は「偶発的複雑性」あるいは「技術的負債」を増加させないための非常に慎重な言語設計がされています。今回の例でもマルチスレッドが出てきており、難しいプログラムの部類に入るはずですが、Rust の力により端的に記述することができています。

難しさへの対策としては、自分の実力に応じてプログラムのレベルを決めるということが重要だと思います。例えば第 2 回で紹介したようなプログラムを書く分には、Python とほぼ同等の記述で書けます。またデータコピーを許せば、所有権に関するコンパイルエラーからある程度逃げることもできると思います。高速化も抽象化も突き詰めると、設計行為が必要であり難しくなるのは必然なので、少しずつ理解を深めるスタンスが良いと考えます。

まとめ

今回は Rust の特に重要な言語機能を抜粋して紹介しました。Rust には多くの優れた機能がありますが、ここでは全てを紹介しません。

Rust はどちらかといえば安全性（≒不変性）を志向しており、自由度（≒可変性）がやや犠牲になっている言語です。雑に書きにくいけれども読みやすく、振る舞いがよく定義されている言語です。まずは手慣れた言語で要件定義をして、Rust に移植して効率的に稼働させるというアプローチも一つの手です。また、公開されているコードを読み解き知識を獲得するにも有益な言語であり、成長につながる言語だと筆者は考えています。

【サンプルコードあり】Rust で作る Web アプリケーション——データベース利用と自動テストの基本事項を押さえる

Rust で Web アプリケーションを開発する際に基礎となる要素技術から Rust の応用まで、Rust に関するあれこれを解説する本連載。第 1 回では Rust を使った Web アプリケーション開発におけるデータベースと自動テストの位置付けとコード例を紹介する。

（2022 年 05 月 26 日）

paiza で Web エンジニアをやっています藤田と申します。今回の連載では、Rust で Web アプリケーションを開発する上での基礎となり得る要素技術や Rust の応用にフォーカスを当てて簡潔に紹介します。

Rust を採用するモチベーションや Rust の有益な言語機能について知りたい方は、前回の連載（全 3 回）にて端的に要約しているのでご参照ください。

今回のプロジェクトも GitHub のサンプルリポジトリを用意していますので、解説コードを実行する際はご利用ください。第 1 回では、Web アプリケーションを構築する上で基礎となるデータベースの利用と自動テストについて基本的な事項を押さえつつ、Rust でのソースコード例を示します。

Web アプリケーションとデータベース

Web アプリケーション（web サービス）を構築するに当たり、仕様、要件を達成できるアーキテクチャであればどのように設計してもよいわけですが、データを記録するコンポーネントとして（リレーショナル）データベースを採用することが多いです。

システムを Web アプリケーションサーバとデータベースサーバに分離して、共有、記録すべき状態（ステート）の管理をデータベースサーバに委譲します。このことにより、Web アプリケーションサーバが状態に依存しない、ステートレスな処理のみを担当することになり、ステートレス処理のスケールアウトやアプリケーションのデプロイプロセスが簡略化されるとともにデータの管理が容易になります。

この方式のデメリットとしては、データベースサーバが単一障害点になること、スケールアップが必要になること、通信遅延やクエリの処理遅延がボトルネックになり得ることなどが挙げられます。しかしながら、厳しい性能要求がない限りは第一選択になる構成だと考えられます。

アプリケーション開発と自動テスト

Web 開発に限定せず継続的に開発を進める必要があるアプリケーション開発全般において自動テストは有用です。特にアジャイルな開発プロセスを採用する場合、自動テストの整備はほぼ必須に近い位置付けとなりつつあります。

改めて自動テストの主目的を説明すると「システムに期待する振る舞い（≒仕様）をテストコードとして事前に記述し、ソースコードの変更が行われたときなどにテストコードを実行することで、ソースコードの変更後にも期待する振る舞いが破壊されていないことを自動的に検証すること」だといえます。

Ruby on Rails での開発によく採用されている RSpec などの分類を参考にすると、Web 開発では以下のようなテストをよく行います。

1. ロジックテスト（純粋なロジックの検証）
2. ミドルウェアテスト（データベースなどのミドルウェアと連携し、レスポンスや副作用を検証する）
3. エンドポイントテスト（HTTP エンドポイントにリクエストし、レスポンスや副作用を検証する）
4. ブラウザテスト（ブラウザをプログラムで制御し、レスポンスや副作用を検証する）
5. etc……（パフォーマンステストなど）

Rust ではテストは言語標準機能として存在しますが、2022 年 5 月現在では RSpec のようなデファクトスタンダードなテストフレームワークがあるわけではなく、複雑な自動テストの構成には自由度があります。自動テストの記述を簡単にするライブラリも存在するようですが、今回の記事ではテスト目的を達成する単純で拡張可能なテストコードを示します。

Rust の主要なデータベースクライアントライブラリ

Rust からデータベースを利用するためによく用いられるライブラリとしては「Diesel」と「SQLx」があります。この 2 つのライブラリはかなりコンセプトが異なります。

Diesel は Ruby on Rails の ActiveRecord に似た ORM（オブジェクトリレーショナルマッパー）で、Rust の構文を用いて SQL を生成、発行できます。一方 SQLx は ORM 機能を持たないライブラリであり、素の SQL をそのまま書いてデータベースにクエリを発行します。また 2022 年 5 月現在では、SQLx の方は async/await 構文に対応している一方で Diesel は未対応であるという違いもあります。

この記事では非同期処理に対応していて Rust の Web フレームワークと相性が良く、ライブラリとして比較的軽量な SQLx を用いて MySQL に接続します。

Rust+SQLx で MySQL に接続する

ここでは、Rust から MySQL を制御する方法を例示します。

プログラムの実行には Rust 開発ツールのインストールの他、「MySQL」のインストールが必要です。サンプルリポジトリには Docker を用いたサンプルもあるので、ホスト環境へのインストールを避けたい場合はご利用ください。

今回は、データ分析や機械学習によく用いられるアヤメのデータセットを Rust の構造体として定義し、MySQL に読み書きするプログラムを用意しました。 データセットのテーブル生成 SQL は以下のようになります。

```
CREATE TABLE IF NOT EXISTS iris_measurements (
  id SERIAL,
  sepal_length DOUBLE NOT NULL, -- がくの長さ
  sepal_width DOUBLE NOT NULL, -- がくの幅
  petal_length DOUBLE NOT NULL, -- 花弁の長さ
  petal_width DOUBLE NOT NULL, -- 花弁の幅
  class VARCHAR(16) NOT NULL -- 分類名
);
```

SQL があればこれをそのまま実行してテーブルを作成することもできますが、Rust を用いて SQL を発行するための基本的なプログラムを用意しました。

今回はテーブルの作成プログラム（src/init_db.rs）とデータの読み書きプログラム（src/main.rs）の 2 つを作る関係上、共通化したいコードを src/lib.rs に記述します。また今回のプロジェクトファイル（Cargo.toml）は以下のように記述しています。

```
# Cargo.toml
[package]
authors = ["Naoki Fujita"]
edition = "2021"
name = "web_engineer_in_rust"
version = "0.1.0"
[dependencies]
anyhow = "1.0.56"
csv = "1.1.6"
futures = "0.3.21"
serde = "1.0.136"
sqlx = {version = "0.5.11", features = ["runtime-tokio-native-tls", "mysql", "chrono", "json"]}
tokio = {version = "1.17.0", features = ["full"]}
[[bin]]
name = "init_db"
path = "src/init_db.rs"
```

```rust
// src/lib.rs
use sqlx::{
    mysql::{MySqlPoolOptions, MySqlQueryResult},
    Executor as _, MySql, Pool,
};
// (1)本番DB(想定)のデータベース接続文字列
pub const DB_STRING_PRODUCTION: &'static str =
"mysql://user:pass@localhost:53306/production";
// テストDB(想定)のデータベース接続文字列
pub const DB_STRING_TEST: &'static str =
"mysql://user:pass@localhost:53306/test";
// (2)非同期処理を実行するランタイムを作成
pub fn create_tokio_runtime() -> tokio::runtime::Runtime {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
}
// MySQL接続のためのクライアント
// コネクションプーリングにより動的なクライアント生成を回避
pub async fn create_pool(url: &str) -> Result<Pool<MySql>, sqlx::Error> {
    MySqlPoolOptions::new().connect(url).await
}
// DBに記録するデータとして、アヤメの測定データを定義
#[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize,
sqlx::FromRow)]
pub struct IrisMeasurement {
    pub id: Option<u64>,
    pub sepal_length: f64, // がくの長さ
    pub sepal_width: f64, // がくの幅
    pub petal_length: f64, // 花弁の長さ
    pub petal_width: f64, // 花弁の幅
    pub class: String, // 分類名
}
impl IrisMeasurement {
    const TABLE_NAME: &'static str = "iris_measurements";
    // (3)include_str!マクロの利用
    pub async fn create_table(pool: &Pool<MySql>) -> Result<MySqlQueryResult,
sqlx::Error> {
        pool.execute(include_str!("../sql/ddl/iris_measurements_create.sql"))
            .await
    }
    // (4)プリペアドステートメントの利用
    pub async fn insert(self, pool: &Pool<MySql>) -> Result<MySqlQueryResult,
sqlx::Error> {
        let sql = format!(
            r#"INSERT INTO {} (sepal_length, sepal_width, petal_length,
petal_width, class) VALUES (?, ?, ?, ?, ?)"#,
            Self::TABLE_NAME
        );
        let result = sqlx::query(&sql)
```

```
46.             .bind(self.sepal_length)
47.             .bind(self.sepal_width)
48.             .bind(self.petal_length)
49.             .bind(self.petal_width)
50.             .bind(self.class)
51.             .execute(pool)
52.             .await;
53.         result
54.     }
55.     // アヤメの種類を指定しリストを取得
56.     pub async fn find_by_class(
57.         pool: &Pool<MySql>,
58.         class: &str,
59.     ) -> Result<Vec<IrisMeasurement>, sqlx::Error> {
60.         let sql = format!(r#"SELECT * FROM {} WHERE class = ?"#,
    Self::TABLE_NAME);
61.         let rows = sqlx::query_as::<_, IrisMeasurement>(&sql)
62.             .bind(class)
63.             .fetch_all(pool)
64.             .await?;
65.         Ok(rows)
66.     }
67. }
68. // 詳細はWebエンジニアからみたRust第2章参照
69. fn get_csv_path(relative_path: &str) -> std::path::PathBuf {
70.     std::path::Path::new(env!("CARGO_MANIFEST_DIR")).join(relative_path)
71. }
72. // 詳細はWebエンジニアからみたRust第2回参照
73. pub fn read_csv(relative_path: &str) -> anyhow::Result<Vec<IrisMeasurement>> {
74.     let csv_path = get_csv_path(relative_path);
75.     let mut csv_reader = csv::Reader::from_path(csv_path)?;
76.     let nums = csv_reader
77.         .deserialize::<IrisMeasurement>()
78.         .filter_map(|row_result| row_result.ok())
79.         .collect::<Vec<_>>();
80.     Ok(nums)
81. }
```

共通部分である lib.rs について特記すべきところについて解説します。

（1）のデータベース接続文字列については、サンプルリポジトリの Docker コンテナを動かした場合の想定で、localhost の 53306 ポートに MySQL が待ち受けている前提になっています。MySQL 標準である 3306 ポートの使用を回避していますが、それでもポートが衝突してしまう場合などは適宜設定変更が必要です。

（2）については、Rust では使用する非同期処理ランタイム（Tokio など）を指定し、そのランタイムの実行コンテキスト内でのみ async/await 構文を用いることができます。今回のコード例では非同期処理を用いるメリットはほぼありませんが、複数のタスクを同時並行的に処理する状況において計算リソースを効率的に使用できるため、Rust の Web フレームワークや SQLx は非同期処理が前提となっています。

（3）の include_str! マクロは外部ファイルをコンパイル時に文字列として読み込むことができるマクロで、ファイルが存在しないとコンパイル段階でエラーとなるので非常に便利です。ただし実行ファイルは事前に読み込む分大きくなるため、大きいファイルに対してはあまり良い手段ではありません。

（4）は SQLx が ORM 機能を持たないことを象徴するような書き方で、SQL をプリペアドステートメントにより構築しています。

ActiveRecord や Diesel などの ORM を利用するのに比べると単純なクエリを発行させるのは煩雑ですが、例えば IrisMeasurement.class を String（文字列）ではなく Enum（列挙型）で表現するなどして、より適切な型で読み書きすることでデータモデルの表現能力を改善しやすいなどの利点があると考えられます。

```
1.  // src/init_db.rs
2.  use web_engineer_in_rust::{create_pool, create_tokio_runtime, IrisMeasurement,
    DB_STRING_PRODUCTION};
3.  fn main() -> anyhow::Result<()> {
4.      let tokio_rt = create_tokio_runtime();
5.      tokio_rt.block_on(run())
6.  }
7.  async fn run() -> anyhow::Result<()> {
8.      // 本番データベースに接続するクライアントプールを作成
9.      let pool = create_pool(DB_STRING_PRODUCTION).await?;
10.     // 本番データベースにiris_measurementsテーブルを作成
11.     let query_result = IrisMeasurement::create_table(&pool).await?;
12.     println!("{:?}", query_result);
13.     Ok(())
14. }
```

次は init_db.rs ですが、特に難解な処理は含まれていないので解説を割愛します。

下記シェルコマンドで実行できます（Cargo.toml に init_db というエントリーポイントを定義しており、src/init_db.rs の main 関数が呼ばれます）。

```
1.  cargo run --bin init_db
```

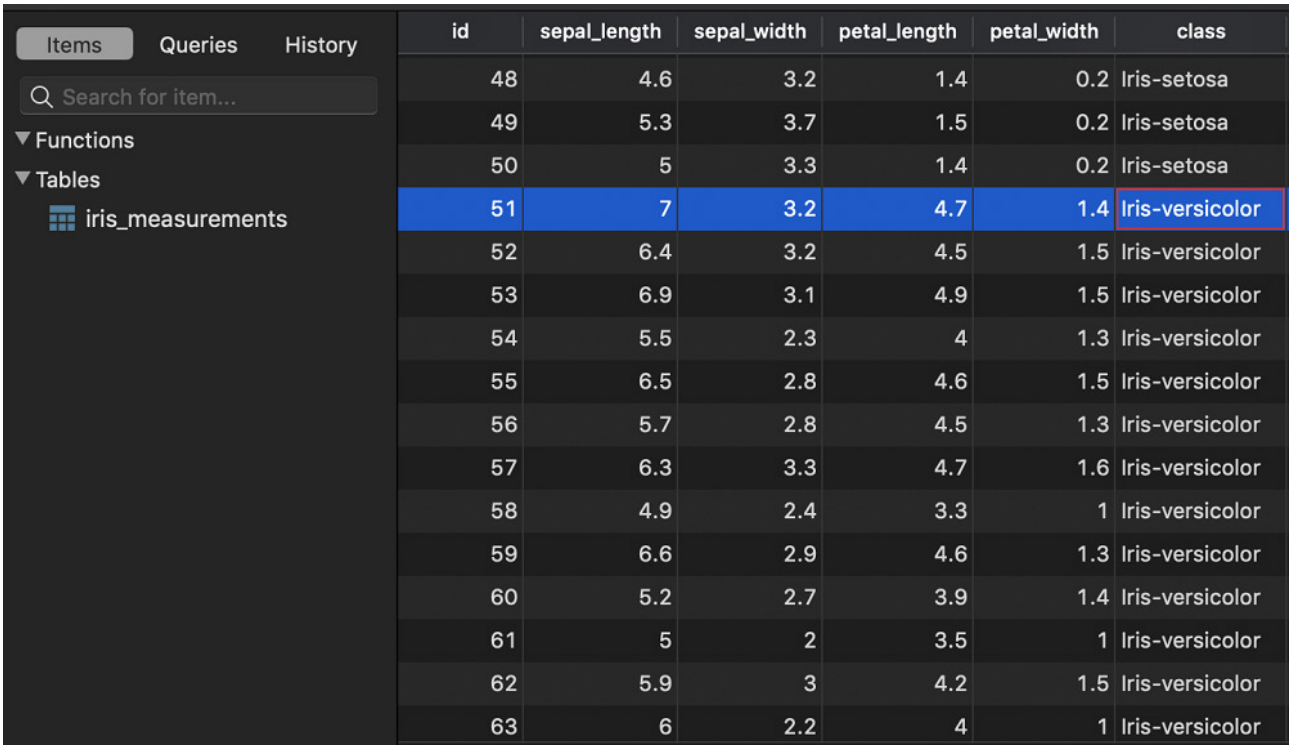
今回は単一テーブルのみの生成の例ですが、外部キー制約などを利用すると適切な順序でテーブル生成する必要が出てきます。

```rust
// src/main.rs
use web_engineer_in_rust::{
    create_pool, create_tokio_runtime, read_csv, IrisMeasurement,
DB_STRING_PRODUCTION,
};
fn main() -> anyhow::Result<()> {
    // 非同期ランタイムを生成
    let tokio_rt = create_tokio_runtime();
    tokio_rt.block_on(run())
}
async fn run() -> anyhow::Result<()> {
    // csvからデータセットをメモリにロード
    let measurements = read_csv("data/iris.csv")?;
    let pool = create_pool(DB_STRING_PRODUCTION).await?;
    // 1件ずつデータベースにINSERT
    for m in measurements {
        m.insert(&pool).await?;
    }
    // Iris-versicolorのデータを取得する
    let rows = IrisMeasurement::find_by_class(&pool, "Iris-
versicolor").await?;
    println!("{:?}", rows);
    Ok(())
}
```

最後に main.rs ですが、こちらも特に難解な処理は含まれていないので解説を割愛します。

下記シェルコマンドで実行できます（エントリーポイントが複数ある場合はクレート名を指定すると main.rs の main 関数を呼び出せます）。

```
cargo run --bin web_engineer_in_rust
```



id	sepal_length	sepal_width	petal_length	petal_width	class
48	4.6	3.2	1.4	0.2	Iris-setosa
49	5.3	3.7	1.5	0.2	Iris-setosa
50	5	3.3	1.4	0.2	Iris-setosa
51	7	3.2	4.7	1.4	Iris-versicolor
52	6.4	3.2	4.5	1.5	Iris-versicolor
53	6.9	3.1	4.9	1.5	Iris-versicolor
54	5.5	2.3	4	1.3	Iris-versicolor
55	6.5	2.8	4.6	1.5	Iris-versicolor
56	5.7	2.8	4.5	1.3	Iris-versicolor
57	6.3	3.3	4.7	1.6	Iris-versicolor
58	4.9	2.4	3.3	1	Iris-versicolor
59	6.6	2.9	4.6	1.3	Iris-versicolor
60	5.2	2.7	3.9	1.4	Iris-versicolor
61	5	2	3.5	1	Iris-versicolor
62	5.9	3	4.2	1.5	Iris-versicolor
63	6	2.2	4	1	Iris-versicolor

GUI ツールにてデータベース内にデータが記録されていることを確認

Rust で自動テストとカバレッジ計測

Rust は言語機能としてテストがサポートされています。今回はデータベースの連携を紹介したので、データベースを期待通りに制御できているかを検証するテストコードを書きます。

```rust
#[cfg(test)]
mod tests {
    use super::*;
    pub async fn truncate_table(
        pool: &Pool<MySql>,
        name: &str,
    ) -> Result<MySqlQueryResult, sqlx::Error> {
        let sql = format!("TRUNCATE TABLE {}", name);
        pool.execute(sql.as_str()).await
    }
    // テスト用データ生成
    // RailsのFactoryBotをリスペクト
    fn create_fake() -> IrisMeasurement {
        IrisMeasurement {
            id: None,
            sepal_length: 3.0,
            sepal_width: 4.0,
            petal_length: 5.0,
            petal_width: 6.0,
            class: "Iris-virginica".to_string(),
        }
    }
```

```
    // テーブルの生成と初期化
    pub async fn setup_database(pool: &Pool<MySql>) {
        let _ = IrisMeasurement::create_table(pool).await.unwrap();
        let _ = truncate_table(pool, IrisMeasurement::TABLE_NAME)
            .await
            .unwrap();
    }
    #[tokio::test]
    async fn create_and_select_ok() {
        // テスト用のデータベースに接続
        let pool = create_pool(DB_STRING_TEST).await.unwrap();
        // 前回のテスト実行による副作用を初期化
        let _ = setup_database(&pool).await;
        let measurement = create_fake();
        let insert_result = measurement.insert(&pool).await.unwrap();
        // INSERT文によりデータが記録されたか検証する
        assert_eq!(
            "MySqlQueryResult { rows_affected: 1, last_insert_id: 1 }",
            format!("{:?}", insert_result)
        );
        let actual1 = IrisMeasurement::find_by_class(&pool, "Iris-virginica")
            .await
            .unwrap();
        // 条件を満たす登録データが取得できたか検証する
        assert_eq!(actual1.len(), 1);
        let actual2 = IrisMeasurement::find_by_class(&pool,
"abc").await.unwrap();
        // 条件を満たす登録データは取得できないことを検証する
        assert_eq!(actual2.len(), 0);
    }
}
```

データベーステストは通常副作用（データの保存や更新）を伴います。他の言語で利用可能なテストフレームワークでは、テスト初期化処理（セットアップ）とテスト終了時処理（ティアダウン）が枠組みとして用意されることが多いのですが、今回のケースでは初期化処理時に完全に初期化することで終了時処理を省く構成としました。

また、Rust のテストは指定がない限り全てのテストが並行的に動作するので、同一テーブルに対するテストが複数あると、競合して失敗することがあります。Mutex などを活用することで、テストの並行性を可能な限り維持しつつ失敗しないテストプログラムを書くこともできますが、実行スレッド数を 1 に指定する方法が一番簡単です。

```
cargo test -- --test-threads=1 # 実行スレッドを1に制限して自動テスト
```

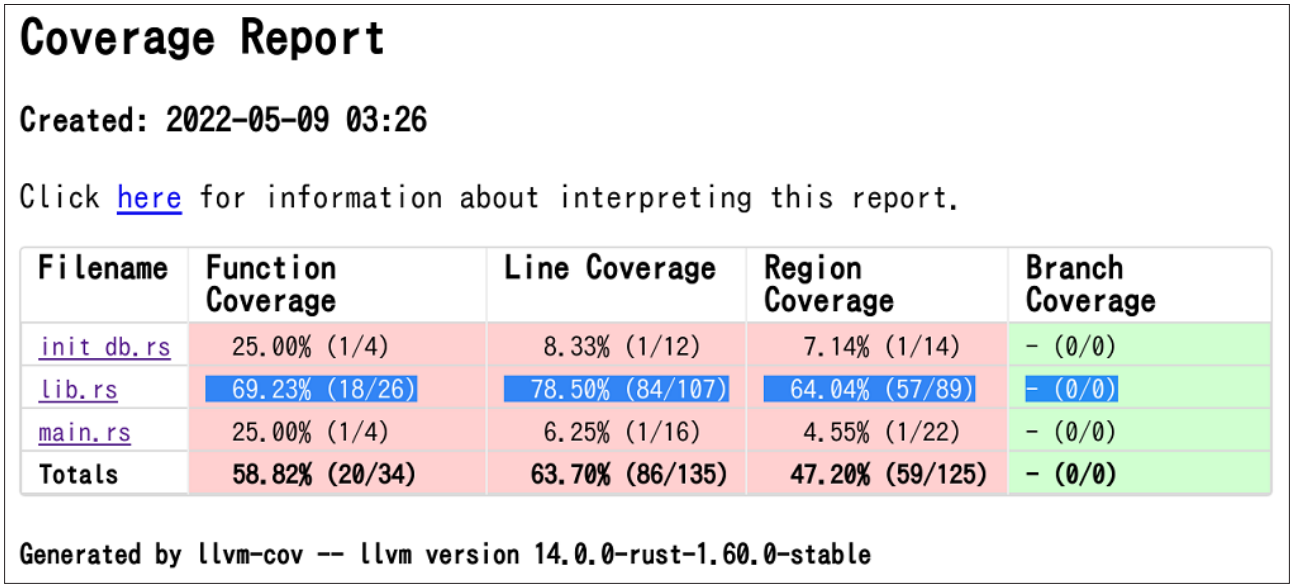
また自動テストによりソースコードが実行されどの程度検証されているかの割合をカバレッジと言います。Rust のカバレッジ計測ツールはいろいろ選択肢がありましたが、2022 年 5 月時点で直近のリリースである Rust1.60 で利用できる cargo-llvm-cov がより詳細な情報を取得できるので、こちらの利用例を紹介します。

インストールは下記の cargo と rustup コマンドで簡単に行うことができます。

```
cargo install cargo-llvm-cov
rustup component add llvm-tools-preview
```

カバレッジの計測は下記のシェルコマンドで実行することができ、カバレッジ情報が分かりやすく表示された html ファイルが出力されます。

```
cargo llvm-cov --html # カバレッジをhtmlファイルに出力
```



Coverage Report

Created: 2022-05-09 03:26

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
init_db.rs	25.00% (1/4)	8.33% (1/12)	7.14% (1/14)	- (0/0)
lib.rs	69.23% (18/26)	78.50% (84/107)	64.04% (57/89)	- (0/0)
main.rs	25.00% (1/4)	6.25% (1/16)	4.55% (1/22)	- (0/0)
Totals	**58.82% (20/34)**	**63.70% (86/135)**	**47.20% (59/125)**	**- (0/0)**

Generated by llvm-cov -- llvm version 14.0.0-rust-1.60.0-stable

ファイルごとの実行カバレッジの可視化

```
16          // 非同期処理を実行するランタイムを作成
17      0   pub fn create_tokio_runtime() -> tokio::runtime::Runtime {
18      0       tokio::runtime::Builder::new_multi_thread()
19      0           .enable_all()
20      0           .build()
21      0           .unwrap()
22      0   }
23
24          // MySQL接続のためのクライアント
25          // コネクションプーリングによりクライアント生成コストを削減
26      1   pub async fn create_pool(url: &str) -> Result<Pool<MySql>, sqlx::Error> {
27      7       MySqlPoolOptions::new().connect(url).await
28      1   }
29
30          // DBに格納するデータとして、アヤメの測定データを定義
31      1   #[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize, sqlx::FromRow)]
            Unexecuted instantiation: <web_engineer_in_rust::IrisMeasurement as core::cmp::PartialEq>::eq
```

lib.rs ファイルのカバレッジ状況の可視化

こうしたカバレッジ情報を見ながらテストを拡充することで、継続的な開発を進めることができます。

まとめ

今回は、アプリケーション開発におけるデータベースと自動テストの位置付けおよび Rust でのコード例について紹介しました。

筆者は Ruby on Rails の ActiveRecord や RSpec、Python における pytest を知っており、それと比べると Rust ではデータベース連携やテスト構築方法に自由度がありプラクティスが定まっていない感覚はあります。

一方で言語機能や開発ツールが極めて強力であることと、リソース効率の高さは魅力であり、開発規模の拡大に応じて開発がスローダウンする要因（テストを含めたパフォーマンス低下、複雑な状態管理）を削減できる選択を取れるのは、Rust を扱っていてとても良いと感じます。Diesel を利用すればまた別の意見が出てくるかもしれません。

次回では、今回の基礎的事項を踏まえつつ、Web フレームワークを利用しネットワークアプリケーションを作成します。

Rust の Web フレームワーク「axum」で SNS アプリの API サーバを作る ――Web 開発での記述性、要素技術を解説

Rust で Web アプリケーションを開発する際に基礎となる要素技術から Rust の応用まで、Rust に関するあれこれを解説する本連載。第 2 回では API サーバを構築し、SNS アプリを簡易実装することで Rust を使った Web 開発での記述性や要素技術を解説する。

（2022 年 08 月 09 日）

paiza で Web エンジニアをやっている藤田と申します。

前回は、Web アプリケーションにおける RDB（リレーショナルデータベース）の立ち位置と、Rust から RDB を制御する実装および自動テストについて記述しました。今回は、Rust で Web フレームワークである「axum」を用いて（REST）API サーバを構築し、SNS アプリを簡易実装することで、Rust での Web 開発での記述性や要素技術を解説します。

今回のプロジェクトも GitHub のサンプルリポジトリを用意していますので、コードを実行する際はご利用ください。

今回の記事のスコープ設定

現代の Web アプリケーションは、ネットワーク、データベース、ブラウザ、暗号などの技術を核として、データモデリング、デザイン、UI/UX（ユーザーインタフェース／ユーザーエクスペリエンス）、セキュリティ、ログ、トレーシング、テレメトリー、データ分析、自動テスト、監視などの非常に多種多様な関心事に取り囲まれています。

その全てを詳述し、一度にその解決策を実装するのも難しいため、この記事ではサーバサイドで重要な基礎となるデータモデリングと RDB+ クッキーによるセッション管理のサンプル実装を提示します。HTTPS、認証機能、ロギングなども実装されていないので、本番環境で利用する際はアプリに要求されるセキュリティ基準にのっとり機能を追加してください。

Rust で API 開発を進める利点の再整理

Rust で API 開発を進めた所感として、Ruby on Rails などの成熟した Web フレームワークに比べると、モノリシックに提供されるソリューションが少なく、硬めの型システムを持つ静的型付け言語で API を記述するのはそれなりに難しいということです。一方で、以下のような利点があると考えています。

- 実行性能が高い
  - ・限られた計算量を UX 改善に使ったり、システムの複雑性を下げたり、インスタンスサイズを下げるのに使える
- OSS（オープンソースソフトウェア）フレームワークの開発ポリシーの影響を受けにくく、最新のソリューションへの段階的移行を進めやすい
- 高度な静的型システム
  - ・システムがよく定義された（well-defined）状態になる
  - ・エラーハンドリングを精緻に行いやすい
  - ・技術的負債が積まれにくい

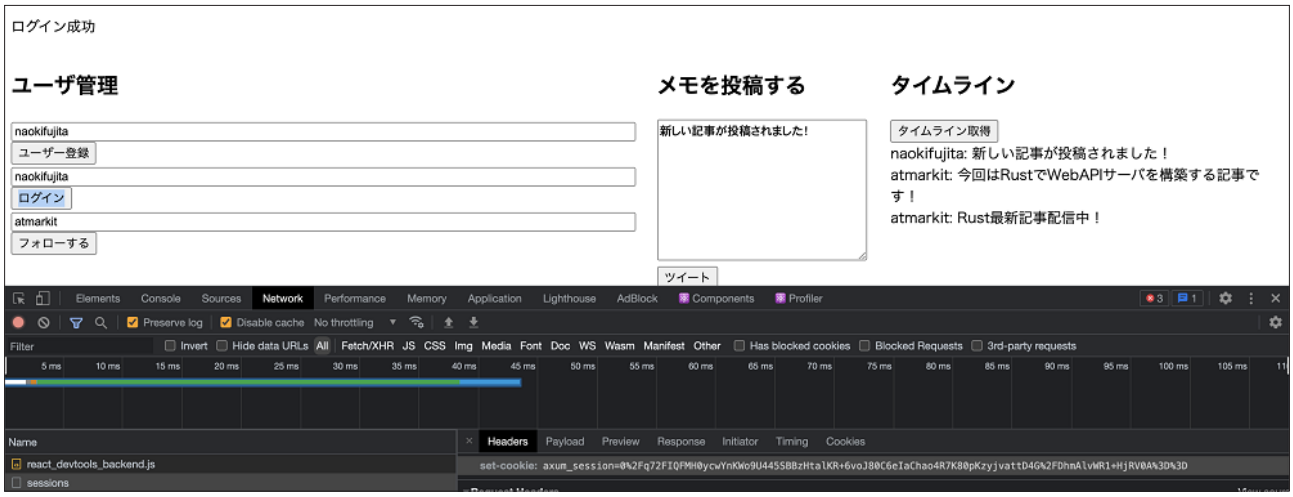
Rust での Web 開発は、初期段階でさまざまな難しさ（言語の難しさ、実装選択の難しさ）に直面するため短期、小規模での開発には向かないと思われますが、長期、継続的に規模が拡大する開発において有利に作用するファクターが多いように感じます。

今回の記事で作る SNS アプリの API 要件

今回は Rust で API サーバのサンプルを実装するに当たり某 SNS サービスの機能をまねたアプリ「Ruitter」を作成しようと思います。作成する API とその基本的要件は以下のようになります。

1. ユーザー新規作成 API（ユーザー名を POST すると、ユーザー名が重複していなければユーザーが作成できる）
2. ログイン API（ユーザー名を POST すると、ユーザー名が RDB に登録されていればログインでき、クッキーと RDB にセッションキーが記録される）
3. フォロー API（ログイン状態でフォローしたいユーザー名を POST すると、フォローできる）
4. メモ作成 API（ログイン状態で 140 文字以内のテキストを POST すると、テキストを記録できる）
5. タイムライン API（ログイン状態で GET すると、自分とフォローしているユーザーのメモが最新順で見られる）

あくまで今回の記事の主体は Rust による API サーバ実装ですが、サンプルコードにはデバッグの Web UI プログラムも用意しています。



API サーバ動作確認用のデバッグ UI。左の「ユーザー管理」で機能 1 ～ 3 を、中央の「メモを投稿する」で機能 4 を、右の「タイムライン」で機能 5 を検証できます。また Chrome DevTools を活用し、Set-Cookie レスポンスヘッダや Cookie リクエストヘッダを確認すると検証がはかどります

機能 2 に至ってはユーザー名さえ知っていれば誰でもなりすましができるなど本番運用するには課題がありますが、今回の内容を理解すれば機能拡張を進めていくことができると思います。

今回の利用ライブラリ

今回は Web フレームワークライブラリとして axum を利用します。このライブラリは非同期ランタイム Tokio の開発チームが開発を進めており、扱いやすいため採用しています。その他利用ライブラリはプロジェクトファイル（Cargo.toml）に記述しています。

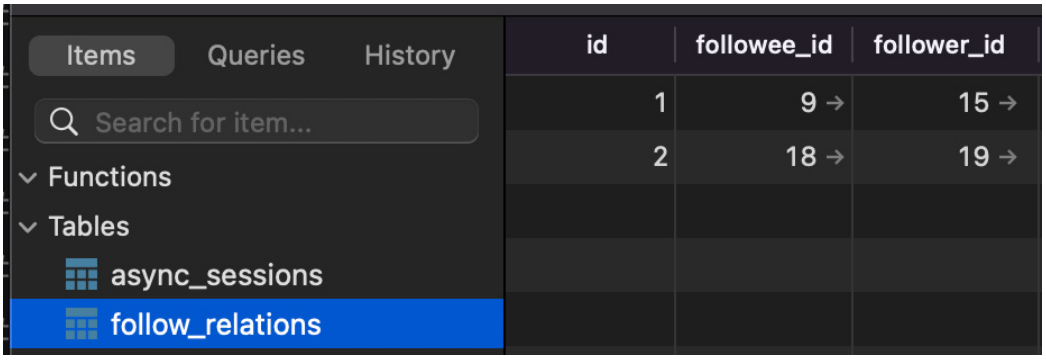
```
[package]
authors = ["Naoki Fujita"]
edition = "2021"
name = "ruitter"
version = "0.1.0"
[dependencies]
# 便利なエラーハンドリングライブラリ
anyhow = "1.0.58"
# セッションライブラリ
async-session = "3.0.0"
# セッションデータをRDBに格納するためのライブラリ
async-sqlx-session = {version = "0.4.0", features = ["mysql"]}
# Webフレームワーク
axum = {version = "0.5.13", features = ["headers", "http2", "ws", "tower-log"]}
# Cookie管理に便利なユーティリティーがあるので使用
axum-extra = {version = "0.3.6", features = ["cookie"]}
# 非同期処理の基本ライブラリ
futures = "0.3.21"
# シリアライズ・デシリアライズのライブラリ
serde = "1.0.140"
```

```
# JSONとRust構造体間をシリアライズ、デシリアライズするためのライブラリ
serde_json = "1.0.82"
# RustからRDBを扱うためのライブラリ
sqlx = {version = "0.6.0", features = ["runtime-tokio-native-tls", "mysql", "chrono", "json"]}
# クッキーの基本ライブラリ
cookie = "0.16.0"
# 非同期ランタイムライブラリ
tokio = {version = "1.17.0", features = ["full"]}
[[bin]]
name = "init_db"
path = "src/init_db.rs"
```

データモデルレイヤーの実装

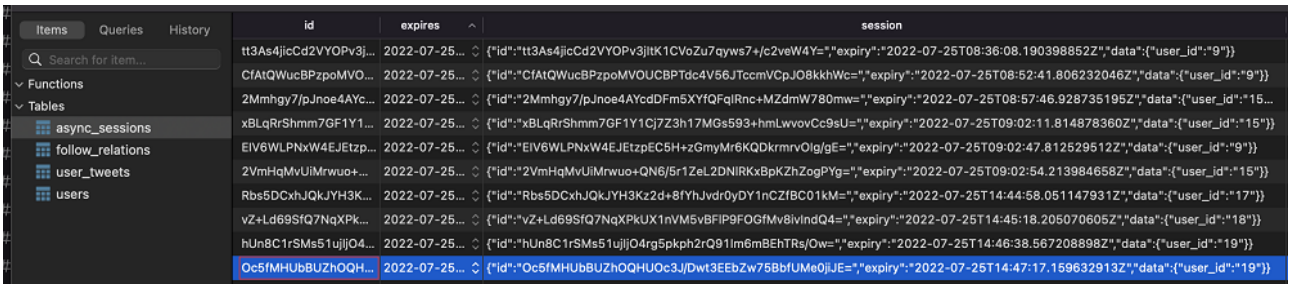
5 つの機能実現に当たり、データテーブルが幾つか必要になります。

- ユーザーテーブル（user）
  - ・機能 1 の実現に必須
- セッションデータテーブル（async_sessions）
  - ・機能 2 の実現に必須
  - ・async-sqlx-session ライブラリに管理させる
- フォロー関連テーブル（follow_relations）
  - ・機能 3 の実現に必須
  - ・フォロワー（フォローするユーザー）とフォロイー（フォローされるユーザー）を多対多でひも付け
- メモテーブル（user_tweets）
  - ・機能 4,5 の実現に必須
  - ・メモを記録し、ユーザーとメモを 1 対多でひも付け



follow_relations テーブル。ID15 のユーザーは ID9 のユーザーをフォローし、ID19 のユーザーは ID18 のユーザーをフォローしています

→目次に戻る



async_sessions テーブル。クッキーに保存されたセッションキーをデコードし、該当するセッション ID がこのテーブルに存在する場合、セッションデータを参照できます。この記事ではデータ構造例示のために公開していますが本来は公開してはいけません

モデルの実装を下記に示します。

```rust
// src/models.rs
use sqlx::{
    mysql::{MySqlPoolOptions, MySqlQueryResult},
    Executor as _, MySql, Pool,
};
use std::collections::HashSet;
// 本番DB(想定)のデータベース接続文字列
pub const DB_STRING_PRODUCTION: &'static str =
"mysql://user:pass@localhost:53306/production";
// テストDB(想定)のデータベース接続文字列
pub const DB_STRING_TEST: &'static str =
"mysql://user:pass@localhost:53306/test";
// 非同期処理を実行するランタイムを作成
pub fn create_tokio_runtime() -> tokio::runtime::Runtime {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
}
// MySQL接続のためのクライアント
// コネクションプーリングによりクライアント生成コストを削減
pub async fn create_pool(url: &str) -> Result<Pool<MySql>, sqlx::Error> {
    MySqlPoolOptions::new().connect(url).await
}
#[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize,
sqlx::FromRow)]
pub struct User {
    pub id: Option<u64>,
    pub name: String, // ユーザー名
}
impl User {
    pub const TABLE_NAME: &'static str = "users";
    pub async fn create_table(pool: &Pool<MySql>) -> Result<MySqlQueryResult,
sqlx::Error> {
        pool.execute(include_str!("../sql/ddl/users_create.sql"))
            .await
    }
    // 指定ユーザー名からUser構造体を取得
```

```
35.     pub async fn find_by_name(name: &str, pool: &Pool<MySql>) ->
    Result<Option<User>, sqlx::Error> {
36.         let sql = format!(r#"SELECT * FROM {} WHERE name = ?;"#,
    Self::TABLE_NAME);
37.         let result = sqlx::query_as::<_, User>(&sql)
38.             .bind(name)
39.             .fetch_optional(pool)
40.             .await;
41.         result
42.     }
43.     // UserデータをRDBに永続化する
44.     pub async fn insert(&self, pool: &Pool<MySql>) -> Result<MySqlQueryResult,
    sqlx::Error> {
45.         let sql = format!(r#"INSERT INTO {} (name) VALUES (?);"#,
    Self::TABLE_NAME);
46.         let result = sqlx::query(&sql).bind(&self.name).execute(pool).await;
47.         result
48.     }
49. }
50. #[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize,
    sqlx::FromRow)]
51. pub struct UserTweet {
52.     pub id: Option<u64>,
53.     pub user_id: u64,
54.     pub content: String,
55. }
56. impl UserTweet {
57.     pub const TABLE_NAME: &'static str = "user_tweets";
58.     pub async fn create_table(pool: &Pool<MySql>) -> Result<MySqlQueryResult,
    sqlx::Error> {
59.         pool.execute(include_str!("../sql/ddl/user_tweets_create.sql"))
60.             .await
61.     }
62.     pub async fn insert(&self, pool: &Pool<MySql>) -> Result<MySqlQueryResult,
    sqlx::Error> {
63.         let sql = format!(
64.             r#"INSERT INTO {} (user_id, content) VALUES (?, ?);"#,
65.             Self::TABLE_NAME
66.         );
67.         let result = sqlx::query(&sql)
68.             .bind(&self.user_id)
69.             .bind(&self.content)
70.             .execute(pool)
71.             .await;
72.         result
73.     }
74. }
75. #[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize,
    sqlx::FromRow)]
76. pub struct FollowRelation {
77.     pub id: Option<u64>,
78.     pub followee_id: u64, // フォローされる側のユーザーID
```

```
    pub follower_id: u64, // フォローする側のユーザーID
}
impl FollowRelation {
    pub const TABLE_NAME: &'static str = "follow_relations";
    pub async fn create_table(pool: &Pool<MySql>) -> Result<MySqlQueryResult, sqlx::Error> {
        pool.execute(include_str!("../sql/ddl/follow_relations_create.sql"))
            .await
    }
    pub async fn insert(&self, pool: &Pool<MySql>) -> Result<MySqlQueryResult, sqlx::Error> {
        let sql = format!(
            r#"INSERT INTO {} (followee_id, follower_id) VALUES (?, ?);"#,
            Self::TABLE_NAME
        );
        let result = sqlx::query(&sql)
            .bind(&self.followee_id)
            .bind(&self.follower_id)
            .execute(pool)
            .await;
        result
    }
    pub async fn find_by_follower_id(
        follower_id: u64,
        pool: &Pool<MySql>,
    ) -> Result<Vec<Self>, sqlx::Error> {
        let sql = format!(
            r#"SELECT * FROM {} WHERE follower_id = ?;"#,
            Self::TABLE_NAME
        );
        let result = sqlx::query_as::<_, Self>(&sql)
            .bind(follower_id)
            .fetch_all(pool)
            .await;
        result
    }
}
#[derive(Debug, PartialEq, serde::Serialize, serde::Deserialize, sqlx::FromRow)]
pub struct TimelineItem {
    name: String,
    content: String,
}
// タイムラインデータを返す
// 本当はページネーションなどが必要
pub async fn timeline(
    follower_id: u64,
    pool: &Pool<MySql>,
) -> Result<Vec<TimelineItem>, sqlx::Error> {
    // フォローしているユーザーIDを列挙
    let mut ids = FollowRelation::find_by_follower_id(follower_id, &pool)
        .await?
```

```
            .into_iter()
            .map(|r| r.followee_id)
            .collect::<HashSet<_>>();
        // タイムラインには自分自身の投稿も含める
        ids.insert(follower_id);
        // 現在のsqlxではIN句に配列を直接bindできないのでハックする
        // idの個数分パラメータをbindする
        let placeholders = format!("?{}", ",?".repeat(ids.len() - 1));
        let sql = format!(
            r#"
              SELECT users.name as name, user_tweets.content as content
              FROM user_tweets
              INNER JOIN users
              ON user_tweets.user_id = users.id
              WHERE user_id IN ({})
              ORDER BY user_tweets.id DESC;
            "#,
            placeholders
        );
        let mut query = sqlx::query_as::<_, TimelineItem>(&sql);
        for id in ids {
            query = query.bind(id);
        }
        let result = query.fetch_all(pool).await;
        result
    }
    // MySQLではINDEXにIF NOT EXISTSを宣言できないのでエラーハンドリングする
    pub fn panic_except_duplicate_key(result: Result<MySqlQueryResult,
    sqlx::Error>) {
        if let Err(e) = result {
            let is_duplicate_index_error = e
                .as_database_error()
                .unwrap()
                .message()
                .starts_with("Duplicate key name");
            if !is_duplicate_index_error {
                panic!("{}", e);
            }
        };
    }
    // テーブルを生成する
    // structに対するループはマクロなどを使うことを実現できるが省略
    pub async fn setup_tables(pool: &Pool<MySql>) {
        panic_except_duplicate_key(User::create_table(&pool).await);
        panic_except_duplicate_key(UserTweet::create_table(&pool).await);
        panic_except_duplicate_key(FollowRelation::create_table(&pool).await);
    }
```

※本当はモデルごとにファイル分割したほうがよいです

大枠は第 1 回で説明した話の延長上にすぎないので、詳述は割愛します。timeline 関数については、SQL の複雑度が上がっていたり、SQLx が現状配列の bind をサポートしていなかったりするので、プレースホルダを id の個数分用意して bind するなどのテクニカルな実装を行っています。しかしながら、SQL で必要なデータを取得するということが基本になります。

モデルメソッドが自動実装される「ActiveRecord」に比べると実装が増えますが、必要以上のメソッドが自動実装されないので、モデルの状態管理がしやすいというメリットもあります。Rust のトレイトとマクロを活用すれば、例えば insert メソッドを自動実装することなども可能ですが、難しいので可能性だけを示唆します。

API レイヤーの実装

次は HTTP リクエストを受け付ける API エンドポイントを実装します。API 要件については既に示しており、対応する実装例を下記に示します。

```rust
// src/endpoints.rs
// モデルレイヤーで定義した構造体や関数を読み込み
use crate::models::{timeline, FollowRelation, User, UserTweet};
use async_session::{Session, SessionStore as _};
// セッション情報をMySQLに保存するライブラリ
use async_sqlx_session::MySqlSessionStore;
use axum::{
    extract::{Extension, FromRequest, Json, RequestParts},
    http::StatusCode,
    response::IntoResponse,
    routing::{get, post},
    Router,
};
// クライアントクッキーを制御する便利なライブラリ
use axum_extra::extract::cookie::{Cookie, CookieJar};
use sqlx::{MySql, Pool};
use std::sync::Arc;
// ユーザー新規作成APIのリクエストJSONのスキーマ
#[derive(serde::Deserialize)]
pub struct CreateUserParams {
    pub name: String,
}
// ユーザー新規作成API
pub(crate) async fn create_user(
    Json(payload): Json<CreateUserParams>,
    arc_pool: Extension<Arc<Pool<MySql>>>,
) -> impl IntoResponse {
    let user = User {
        id: None,
        name: payload.name,
    };
    // ユーザー登録を試みる
```

```
33.     match user.insert(&arc_pool).await {
34.         // 成功したらHTTPステータスコード201を返す
35.         Ok(_res) => StatusCode::CREATED,
36.         // 失敗したらHTTPステータスコード400を返す
37.         // ユーザー名重複やサーバ接続エラーなど
38.         // より精緻にステータスコードを分けることもできる
39.         Err(_e) => StatusCode::BAD_REQUEST,
40.     }
41. }
42. // ログインAPI
43. #[derive(serde::Deserialize)]
44. pub struct CreateSessionParams {
45.     pub name: String,
46. }
47. pub(crate) async fn create_session(
48.     Json(payload): Json<CreateSessionParams>,
49.     arc_pool: Extension<Arc<Pool<MySql>>>,
50.     session_store: Extension<MySqlSessionStore>,
51.     cookie_jar: CookieJar,
52. ) -> impl IntoResponse {
53.     // リクエストされた名前が存在するか調べる
54.     match User::find_by_name(&payload.name, &arc_pool).await {
55.         Ok(user) => match user {
56.             // ユーザー名が存在するならログイン処理
57.             Some(user) => {
58.                 let mut session = Session::new();
59.                 let expire_seconds = 86400;
60.
session.expire_in(std::time::Duration::from_secs(expire_seconds));
61.                 session.insert("user_id", user.id).unwrap();
62.                 // RDBにセッション保存を試みる
63.                 match session_store.store_session(session).await {
64.                     Ok(cookie_value) => Ok((
65.                         StatusCode::CREATED,
66.                         // 成功したらSet-Cookieレスポンスヘッダを通じてクッキーを更新
67.                         cookie_jar.add(
68.                             Cookie::build(AXUM_SESSION_COOKIE_KEY,
cookie_value.unwrap())
69.                                 // HTTPS(TLS)非対応なのでfalseとした
70.                                 .secure(false)
71.                                 .http_only(true)
72.                                 .same_site(cookie::SameSite::Lax)
73.
.max_age(cookie::time::Duration::new(expire_seconds as i64, 0))
74.                                 .finish(),
75.                         ),
76.                     )),
77.                     )),
78.                     Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
79.                 }
80.             }
81.             // ユーザー名が存在しない場合
```

```
            None => Err(StatusCode::BAD_REQUEST),
        },
        Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
    }
}
#[derive(serde::Deserialize)]
pub struct CreateUserTweetParams {
    pub content: String,
}
// ツイート作成API
pub(crate) async fn create_user_tweet(
    Json(payload): Json<CreateUserTweetParams>,
    arc_pool: Extension<Arc<Pool<MySql>>>,
    session: CurrentSession,
) -> impl IntoResponse {
    // セッションからuser_idを取得する
    match session.0.get::<u64>("user_id") {
        Some(user_id) => {
            let tweet = UserTweet {
                id: None,
                user_id,
                content: payload.content,
            };
            match tweet.insert(&arc_pool).await {
                Ok(_) => Ok(StatusCode::CREATED),
                Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
            }
        }
        // セッションからuser_idを復元できない場合
        None => Err(StatusCode::UNAUTHORIZED),
    }
}
#[derive(serde::Deserialize)]
pub struct CreateFollowRelationParams {
    pub name: String,
}
// フォローAPI
pub(crate) async fn create_follow_relation(
    Json(payload): Json<CreateFollowRelationParams>,
    arc_pool: Extension<Arc<Pool<MySql>>>,
    session: CurrentSession,
) -> impl IntoResponse {
    match session.0.get::<u64>("user_id") {
        Some(user_id) => {
            // 指定した名前のユーザーが存在するか確認する
            let result = User::find_by_name(&payload.name, &arc_pool).await;
            match result {
                Ok(followee) => match followee {
                    // 存在するならばフォローする
                    Some(followee) => {
                        let follow_relation = FollowRelation {
                            id: None,
```

```
                                followee_id: followee.id.unwrap(),
                                follower_id: user_id,
                            };
                            match follow_relation.insert(&arc_pool).await {
                                Ok(_) => Ok(StatusCode::CREATED),
                                Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
                            }
                        }
                        None => Err(StatusCode::BAD_REQUEST),
                    },
                    Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
                }
            }
            None => Err(StatusCode::UNAUTHORIZED),
        }
    }
    pub(crate) async fn get_timeline(
        arc_pool: Extension<Arc<Pool<MySql>>>,
        session: CurrentSession,
    ) -> impl IntoResponse {
        match session.0.get::<u64>("user_id") {
            Some(user_id) => match timeline(user_id, &arc_pool).await {
                Ok(tweets) => Ok(axum::Json(tweets)),
                Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
            },
            None => Err(StatusCode::UNAUTHORIZED),
        }
    }
    pub async fn run_server(
        arc_pool: Arc<Pool<MySql>>,
        session_store: MySqlSessionStore,
    ) -> anyhow::Result<()> {
        // 8888番ポートで全てのIPアドレスから待ち受ける
        let addr = std::net::SocketAddr::from(([0, 0, 0, 0], 8888));
        // ルーティングを定義する
        // postはHTTP POSTエンドポイント
        // getはHTTP GETエンドポイント
        let app = Router::new()
            .route("/api/users", post(create_user))
            .route("/api/sessions", post(create_session))
            .route("/api/user_tweets", post(create_user_tweet))
            .route("/api/follow_relations", post(create_follow_relation))
            .route("/api/pages/timeline", get(get_timeline))
            // RDBクライアントをアクション関数から呼び出せるようにする
            .layer(Extension(arc_pool))
            // セッションストアをアクション関数から呼び出せるようにする
            .layer(Extension(session_store));
        axum::Server::bind(&addr)
            .serve(app.into_make_service())
            .await?;
        Ok(())
    }
```

```rust
pub struct CurrentSession(Session);
const AXUM_SESSION_COOKIE_KEY: &str = "axum_session";
// https://github.com/tokio-rs/axum/blob/main/examples/sessions/src/main.rsを改変
// axumのカスタムextractorを定義
// クッキーに格納されたセッションキーからセッションデータを復元する
#[axum::async_trait]
impl<B> FromRequest<B> for CurrentSession
where
    B: Send,
{
    type Rejection = StatusCode;
    async fn from_request(req: &mut RequestParts<B>) -> Result<Self, Self::Rejection> {
        // MySQLセッションストアを参照する
        let Extension(store) = Extension::<MySqlSessionStore>::from_request(req)
            .await
            .unwrap();
        // ブラウザから送信されたクッキーを参照する
        let cookie = CookieJar::from_request(req).await.unwrap();
        // クッキーからセッションキーを取得
        let session_id = cookie
            .get(AXUM_SESSION_COOKIE_KEY)
            .map(|cookie| cookie.value())
            .unwrap_or("")
            .to_string();
        // セッションキーからセッションデータを復元する
        let session_data = store.load_session(session_id).await;
        match session_data {
            Ok(session_data) => match session_data {
                // セッションデータが存在=セッションデータを返す
                Some(session_data) => Ok(CurrentSession(session_data)),
                // セッションデータが存在しない=ログインできていない
                None => Err(StatusCode::UNAUTHORIZED),
            },
            // RDBとの接続が切れている可能性がある、500を返す
            Err(_) => Err(StatusCode::SERVICE_UNAVAILABLE),
        }
    }
}
```

※本当はエンドポイントごとにファイル分割した方がよいです

まず見るべきは run_server 関数に含まれている URL パスと関数（リクエストハンドラ）の対応表（ルーティング）です。例えば /api/pages/timeline に GET リクエストを送信すると、get_timeline リクエストハンドラが呼び出されます。またその下にある .layer メソッドを通じて、リクエストハンドラ内から RDB やセッションストアを参照できるよう拡張（Extension）しています。

次にユーザー新規作成 API の create_user リクエストハンドラですが、Json(payload) と arc_pool という 2 つの引数を持っています。arc_pool は前述した .layer メソッドによる拡張の結果であり、MySQL クライアントプールがハンドラ内で使用できることを示しています。

拡張なしでも axum はリクエストに付随する情報（リクエストヘッダ、パス・クエリパラメーター、JSON データなど）を自由に引数として記述でき、この機能は「extractor」と呼ばれています。必要なパラメータを関数の引数として自由に引き出せるのは魔法のように思えますが、Rust のトレイトによって実現されています。

CreateUserParams は JSON データのスキーマを規定しており、このスキーマに適合しないと axum がステータスコード 422 を返してくれるようです。これにより型定義による入力保護と型駆動での開発を推し進めることができます。create_user はユーザー作成の成否に応じて適切なステータスコードを返しています。より緻密にステータスコードを分けたり、エラーメッセージを返却したりすることもできます。

動的型付け言語での実装に比べるとやや煩雑ですが、Rust の null 安全な型システムにより、動作を精緻に記述しやすいです。axum に処理を委譲したり、Option や Result のメソッドを活用したりすると冗長な記述も削れます。

axum の extractor は FromRequest トレイトを実装することで作成できます。create_session 関数では CookieJar という extractor を使用していますが、これは axum_extra というライブラリから持ってきたものです。

また今回は CurrentSession というカスタム extractor を定義し、ログインしていなければステータスコード 401 を返すような実装を作ってみました。このようなカスタム実装をしなくても、毎回セッションストアとクッキーを付き合わせるという実装をすれば要件は実現できますが、少し大変です。

このカスタム extractor を利用してログイン状態を要求するツイート作成 API やフォロー API、タイムライン API を実装しました。

プログラムの実行

プログラムを実行する場合は先にデータベーステーブルの作成が必要です。テーブル初期化するプログラムを示します。

```
// src/init_db.rs
use ruitter::models::{create_pool, create_tokio_runtime, setup_tables, DB_STRING_PRODUCTION};
fn main() -> anyhow::Result<()> {
    let tokio_rt = create_tokio_runtime();
    tokio_rt.block_on(run())
}
async fn run() -> anyhow::Result<()> {
    // 本番DBにセッションテーブルを作成
    let session_store = async_sqlx_session::MySqlSessionStore::new(DB_STRING_PRODUCTION).await?;
    session_store.migrate().await?;
    // 本番DBに接続するクライアントプールを作成
    let pool = create_pool(DB_STRING_PRODUCTION).await?;
    // 本番DBにその他テーブルを作成
    setup_tables(&pool).await;
    Ok(())
}
```

セッションストアを async_sqlx_session に作らせている他は第 1 回での説明と重複するので割愛します。MySQL が 53306 ポートで待ち受けている状態で下記コマンドを実行します。

```
cargo run --bin init_db
```

サーバ起動プログラムを下記に示します。

```
// src/main.rs
use ruitter::endpoints::run_server;
use ruitter::models::{create_pool, create_tokio_runtime, DB_STRING_PRODUCTION};
use std::sync::Arc;
fn main() -> anyhow::Result<()> {
    // 非同期ランタイムを生成
    let tokio_rt = create_tokio_runtime();
    tokio_rt.block_on(run())
}
async fn run() -> anyhow::Result<()> {
    let arc_pool = Arc::new(create_pool(DB_STRING_PRODUCTION).await?);
    let session_store = async_sqlx_session::MySqlSessionStore::new(DB_STRING_PRODUCTION).await?;
    // APIサーバの起動
    run_server(arc_pool, session_store).await
}
```

下記コマンドでコンパイルして実行できます。

```
cargo run --bin ruitter --release
```

API の動作検証としてはリポジトリの /front にある Web UI を使う他、curl などの HTTP クライアントを用いることができます。

まとめ

今回はSNSアプリをサンプル実装するという題材でRustにおけるWeb APIサーバ開発事例を示しました。その過程で

1. Web フレームワーク axum の活用方法
2. サーバサイドセッション実装の例示
3. Rust + SQLx によるデータモデリングの良い点、悪い点
4. Rust + axum による Web API 実装の良い点、悪い点

などを部分的にでも示せたのではないかと思います。特に axum の抽象化は洗練されており、ソースを読むといろいろ参考になりそうです。

Python から利用できる Rust 製超高速データ分析ライブラリ Polars の実力

Rust で Web アプリケーションを開発する際に基礎となる要素技術から Rust の応用まで、Rust に関するあれこれを解説する本連載。第 3 回は、Rust 製の高速データ分析ライブラリである Polars の速度を簡易的に検証し、考察する。

（2022 年 10 月 31 日）

paiza で Web エンジニアをやっています藤田と申します。前回の連載では、Rust で Web アプリの基礎となるセッション管理と、SNS の API サーバを構築するための実装概略、Rust の強力な型システムによるサーバサイドアプリケーションの記述性について示しました。

今回は、趣向を変えて Rust 製の高速データ分析ライブラリである「Polars」を利用し、その速度を簡易的に検証、考察します。今回のプロジェクトも GitHub のサンプルリポジトリを用意していますので、コードを実行する際はご利用ください。

Polars と pandas

Polars は Python の小規模データ分析文脈でよく用いられる pandas を強く意識したライブラリです。pandas がパンダならば、それに対して Polars はホッキョクグマというわけです。

どちらも「データフレーム」とよばれる抽象データ型が使いやすいインタフェースを形成しており「高速にデータ処理できる Excel のような分析ライブラリ」あるいは「インメモリリレーショナルデータベース」という感じで、気軽に扱えるツールとなっています。

利用可能なデータソースとして、リレーショナルデータベースはもちろん、csv や「Microsoft Excel」「Apache Parquet」などのファイル、また「Amazon S3」などにあるデータを扱うことができる点も共通しています。

相違点として、pandas が NumPy 配列という C 言語の構造体的なシンプルなデータ構造をバックエンドとして持つ一方、Polars は Apache Arrow memory model という列指向データ構造を有しており、分析クエリに適しています。また Polars の公式サイトでは pandas 含む他のデータ分析ライブラリよりも卓越した処理速度を持っていることが示されています。

さらに、Polars には pandas にない遅延評価や並列処理などクエリ最適化の機能が織り込まれており、計算資源を有効活用できます。一方でクエリ最適化を行えるようにする都合上 pandas とは API が異なる部分があります（どちらかといえば大規模データ分析ライブラリの「Apache Spark」にやや API が似ています）。そして pandas は Python の API しか提供されていませんが、Polars は Python および Rust の API が提供されています。

探索的データ分析（Exploratory Data Analysis、EDA）をあえて Rust API で実施するといったユースケースは考えにくいため、Python で作成した分析、学習結果を Rust 製アプリケーションにシームレスに組み込むなどのユースケースが考えられるでしょう。今回は Rust からの利用についても試してみます。

今回の試験の実行環境

パフォーマンスは実行環境に依存しますが、参考までに筆者のプログラムの実行環境を下記に示します。物理コアが 16 個あるので、並列処理可能だと大きく高速化できます。

- OS：Debian 11（Linux 5.10.0-17-amd64）
- CPU：AMD Ryzen 9 3950X 16-Core Processor
- RAM：G.Skill F4-3200C16-32GVK×4（DDR4-3200 32GB×4）
- SSD：Crucial CT1000MX500SSD1（1000GB Serial ATA 6Gb/s）

Hello Polars

まずは以前の記事でも取り上げたアヤメのデータセットについて典型的な集計クエリを実行します。

アヤメの種類（class）ごとに、がく片の長さ（sepal length）、がく片の幅（sepal width）、花弁の長さ（petal length）、花弁の幅（petal witdth）の平均および標準偏差を計算します。がく片の長さの平均値の降順でデータを並び替えることにします。また Python の実行時間計測ツール「timeit」を用いて pandas と Polars の実行速度を比較します。

アヤメのデータセットはもともと 150 行の非常に小さなデータセットであるので、実践ではより巨大なデータを処理することをイメージし、1 万回データを繰り返したもの（つまり 150 万行のデータ）を対象として処理時間を計測します。Python および今回利用するライブラリのバージョンは以下のようになります。表示には、バージョン管理ツール Poetry を利用しました。

```toml
# pyproject.toml
[tool.poetry]
authors = ["NaokiFujita"]
description = "test script for performance comparison between polars and pandas"
name = "web_engineer_in_rust"
version = "0.1.0"
[tool.poetry.dependencies]
pandas = "^1.5.0"
python = "^3.9"
polars = "^0.14.18"
[tool.poetry.group.dev.dependencies]
autopep8 = "^1.7.0"
pylint = "^2.15.4"
ipython = "^8.5.0"
```

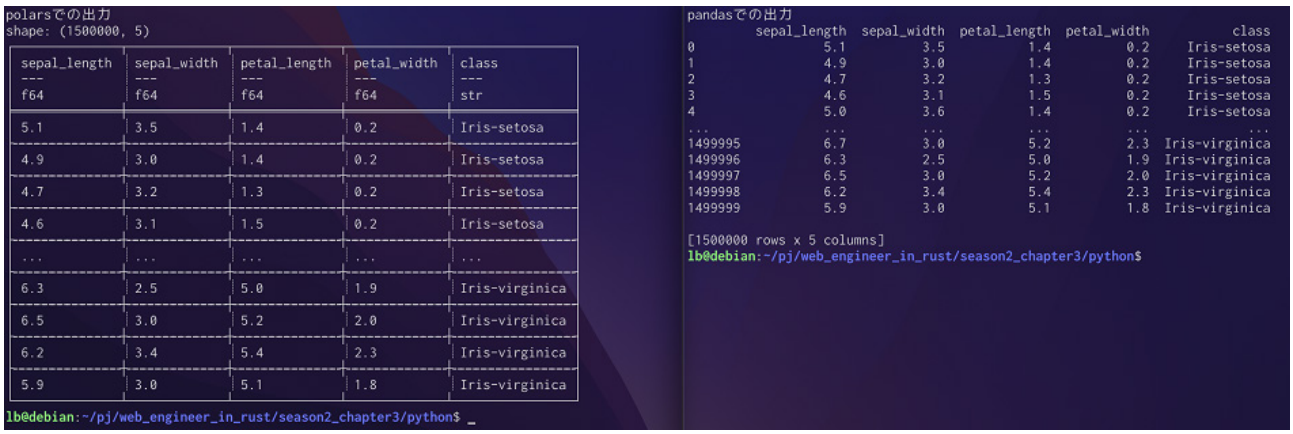
早速ですが pandas と Polars それぞれについて、今回の要件を実現するプログラムを記述します。

```python
# aggregate_query.py
from typing import Union
import numpy as np
import pandas as pd
import polars as pl
def by_pandas(df: pd.DataFrame):
    """pandasを用いてアヤメの種類ごとの特徴量を抽出する"""
    features = (
        df
        .groupby('class')  # アヤメの種類で集約する
        # pandasではマルチインデックスがサポートされているのでこう書ける
        .agg({
            'sepal_length': [np.mean, np.std],
            'sepal_width': [np.mean, np.std],
            'petal_length': [np.mean, np.std],
            'petal_width': [np.mean, np.std],
        })
        # がく片の長さの平均値で表を並び替える(降順)
        .sort_values(by=[('sepal_length', 'mean')], ascending=False)
    )
    return features
def by_polars(df: Union[pl.DataFrame, pl.LazyFrame]):
    """polarsを用いてアヤメの種類ごとの特徴量を抽出する"""
    features = (
        df
        .groupby('class')  # アヤメの種類で集約する
        .agg(
            [
                # polarsはマルチインデックスを現在サポートしていない
                # リスト内包表記などを使うことで抽象化は可能
                pl.col('sepal_length').mean().alias('sepal_length_mean'),
                pl.col('sepal_length').std().alias('sepal_length_std'),
```


→目次に戻る

```
                pl.col('sepal_width').mean().alias('sepal_width_mean'),
                pl.col('sepal_width').std().alias('sepal_width_std'),
                pl.col('petal_length').mean().alias('petal_length_mean'),
                pl.col('petal_length').std().alias('petal_length_std'),
                pl.col('petal_width').mean().alias('petal_width_mean'),
                pl.col('petal_width').std().alias('petal_width_std'),
            ]
        )
        # がく片の長さの平均値で表を並び替える(降順)
        .sort('sepal_length_mean', reverse=True)
    )
    return features
if __name__ == '__main__':
    # pandasでの特徴量計算
    features_by_pandas = by_pandas(pd.read_csv('../data/irisx10000.csv'))
    print(features_by_pandas)
    # polarsでの特徴量計算(遅延評価しない)
    features_by_eager_polars =
by_polars(pl.read_csv('../data/irisx10000.csv'))
    print(features_by_eager_polars)
    # polarsでの特徴量計算(遅延評価する)
    features_by_lazy_polars =
by_polars(pl.scan_csv('../data/irisx10000.csv')).collect()
    print(features_by_lazy_polars)
```

pandas も Polars も DataFrame というクラスを持っており、これが 2 次元表を表すためのデータ構造になります。



データフレームを出力した結果です。単なる 2 次元表として取り扱うことができます。pandas の表示では左に連番が振られた列（インデックス列）がありますが、Polars の出力にはありません

by_pandas 関数では pandas の API を用いてクエリを、by_polars 関数では polars の API を用いてクエリを記述しています。提供されている API による違いはありますが、どちらも class（アヤメの種類）列で groupby（集約）をして、集約関数として mean（平均）と std（標準偏差）を適用し、その出力列でソートをするという概略を素直に書くことができています。慣れる必要はありますが、SQL を書くことができる人であれば習得するのはそこまで難しくはないでしょう。

Polars には pandas にはない LazyFrame というクラスが存在しており、この利用が推奨されています。これは計算の実行を極力遅延させ、collect メソッドを呼び出したときに計算を実行し DataFrame を返却するという機能を有しています。上記のプログラムにおいては、read_csv メソッドを scan_csv メソッドに置き換えるだけで、LazyFrame クラスを用いた遅延実行が可能になります。この遅延実行により、中間オブジェクトの生成を抑えるなど処理最適化を行える余地が大きくなると考えられます。

上記の 3 条件（pandas の DataFrame を用いる、Polars の DataFrame を用いる、Polars の LazyFrame を用いる）について Python の実行時間計測ツール timeit を用いて、簡易計測を行った結果が以下となります。

```
In [5]: %timeit -n 10 by_pandas(pd.read_csv('../data/irisx10000.csv'))
419 ms ±1.14 ms per loop (mean ±std. dev. of 7 runs, 10 loops each)

In [6]: %timeit -n 10 by_polars(pl.read_csv('../data/irisx10000.csv'))
75.8 ms ±1.19 ms per loop (mean ±std. dev. of 7 runs, 10 loops each)

In [7]: %timeit -n 10 by_polars(pl.scan_csv('../data/irisx10000.csv')).collect()
72.8 ms ±1.48 ms per loop (mean ±std. dev. of 7 runs, 10 loops each)
```

3 条件についてそれぞれ時間計測を行った時の出力

結果として Polars は pandas よりもおよそ 6 倍速く実行できました。また大きな差はないですが LazyFrame を用いた方がわずかに速いという結果が得られました。集約クエリは一度集約キーで分割した後はそれぞれ独立な表として扱えるので、並列処理可能な余地が大きいと考えられます。また Polars のバックエンドは Rust で実装されており、物理コアの性能が引き出しやすいと考えることができそうです。

時系列クエリを試してみる

集約クエリは並列処理させやすいので、次は時系列に対するクエリを試してみたいところです。以前の Rust パフォーマンス検証記事で作った時系列データを用いて再び移動平均を計算してみようと思います。

```python
# time_series_query.py
from typing import Union
import pandas as pd
import polars as pl
def by_pandas(df: pd.DataFrame, window_size: int):
    """pandasを用いて移動平均を計算する"""
    features = df.rolling(window_size).mean()
    return features
def by_polars(df: Union[pl.DataFrame, pl.LazyFrame], window_size: int):
    """polarsを用いて移動平均を計算する"""
    features = df.select(
        [
            pl.col('value').rolling_mean(window_size),
        ]
    )
    return features
if __name__ == '__main__':
    # pandasでの特徴量計算
    series_by_pandas = by_pandas(pd.read_csv('../data/time_series.csv'), 50)
    print(series_by_pandas)
    # polarsでの特徴量計算(遅延評価しない)
    series_by_eager_polars = by_polars(pl.read_csv('../data/time_series.csv'),
50)
    print(series_by_eager_polars)
    # polarsでの特徴量計算(遅延評価する)
    series_by_lazy_polars = by_polars(pl.scan_csv('../data/time_series.csv'),
50).collect()
    print(series_by_lazy_polars)
```

pandas で移動平均を計算する場合、文字通り 1 行で処理が記述できてしまいます。この手軽さはやはり pandas の魅力を非常によく表しています。一方で Polars で移動平均を計算する場合、DataFrame、LazyFrame（表データ）自体には移動窓計算を行う API がなく、Series（列データ）にその機能があるので少し複雑な表現になります。

集約クエリの時と同様、3 条件（pandas を DataFrame を用いる、Polars の DataFrame を用いる、Polars の LazyFrame を用いる）について時間計測を行った結果が以下となります。

```
In [6]: %timeit by_pandas(pd.read_csv('../data/time_series.csv'), 50)
686 ms ±24 ms per loop (mean ±std. dev. of 7 runs, 1 loop each)

In [7]: %timeit by_polars(pl.read_csv('../data/time_series.csv'), 50)
83.3 ms ±1.47 ms per loop (mean ±std. dev. of 7 runs, 10 loops each)

In [8]: %timeit by_polars(pl.scan_csv('../data/time_series.csv'), 50).collect()
80.7 ms ±725 µs per loop (mean ±std. dev. of 7 runs, 10 loops each)
```

3 条件についてそれぞれ時間計測を行った時の出力

結果として polars は pandas よりもおよそ 8 倍実行できました。また集計クエリの時と同じく LazyFrame を用いた方がわずかに速いという結果が得られました。前回の記事で作ったプログラムの中では 590ms が最高という結果であったので、それと比較してもおよそ 7 倍速いです。

Rust API を試す

Polars では Rust API も提供されているとのことなので、時系列クエリを Rust でも記述してみます。Rust および今回利用するライブラリのバージョンは以下のようになります。こちらはバージョン管理ツール Cargo を利用して表示しています。

```toml
[package]
authors = ["Naoki Fujita"]
edition = "2021"
license = "MIT"
name = "web_engineer_in_rust"
rust-version = "1.64.0"
version = "0.1.0"
[dependencies]
anyhow = "1.0.65"
polars = {version = "0.24.3", features = ["lazy", "csv-file",
"rolling_window"]}
serde_json = "1.0.68"
[[bin]]
name = "time_series_query"
path = "src/time_series_query.rs"
```

Python の時系列クエリプログラムとほぼ等価な Rust プログラムは以下のようになります。

```rust
// time_series_query.rs
use polars::prelude::{col, DataFrame, Duration, LazyCsvReader, LazyFrame, RollingOptions};
// プロジェクトディレクトリからの相対パスを絶対パスにするユーティリティー
fn get_csv_path(relative_path: &str) -> std::path::PathBuf {
    let project_path = env!("CARGO_MANIFEST_DIR");
    std::path::Path::new(project_path)
        .parent()
        .unwrap()
        .join(relative_path)
}
// polarsで移動平均を計算する
fn by_polars(df: LazyFrame, window_size: i64) -> anyhow::Result<DataFrame> {
    let duration = Duration::new(window_size);
    let rolling_options = RollingOptions {
        window_size: duration,
        min_periods: window_size as usize,
        ..RollingOptions::default()
    };
    let features = df
        .select([col("value").rolling_mean(rolling_options)])
        .collect()?;
    Ok(features)
}
fn main() -> anyhow::Result<()> {
    let csv_path = get_csv_path("data/time_series.csv");
    // csvを遅延読み込みする
    let df = LazyCsvReader::new(csv_path).has_header(true).finish()?;
    // polarsでの特徴量計算(遅延評価する)
    let features = by_polars(df, 50)?;
    println!("{:?}", features);
    Ok(())
}
```

Python 版と API の対比が取りやすいようにプログラムを構成しました。Rust と Python の大きな違いの一つとして、Python はキーワード引数という非常に利便性の高い関数呼び出し方法がありますが、Rust にはパフォーマンスを最大化するためにそのような機能が提供されていないことが挙げられます。結果として上記の LazyCsvReader のように一つ一つのオプションを逐次的に選択するような API（Builder パターンとも呼ばれる）が提供されていることが多く、そのような前提が API 設計の差異を形成しています。

そのような差異があることと API の型に合わせる必要性があることを除けば、Python とおおむね同じプログラム構成で Rust からも Polars を利用できます。一方で Rust API はドキュメントが Python API ほど充実してはおらず、基本的には Python からの利用を想定したライブラリであると思われます。

まとめ

今回は超高速データ分析ライブラリ Polars を用いた集約クエリ、時系列移動平均計算クエリの記述方法を示しました。 さらに pandas API との差異や速度検証を行い、また Rust API を用いたクエリの記述方法も示しました。

Polars はベースが Rust で設計されていることもあり、pandas と比べて API が「かたい」印象を受けますが、それにより卓越した高速化を実現しているため、より大きなデータセットをスムーズに分析したいというニーズに適したライブラリであると考えられます。

筆者紹介

藤田直己

1988 年生まれ、大阪府枚方市出身

京都大学工学部電気電子工学科卒、同大学エネルギー科学研究科修了

応用情報技術者・ネットワークスペシャリスト・情報処理安全確保支援士試験合格者

YKK AP にて超高層建築物の外装設計に従事し、型・モジュール設計・ウオーターフォールプロセスに精通する。その後 IT エンジニアに転向。paiza にて、Ruby on Rails や React を用いた Web サービスのスクラム開発に従事、現在に至る。

最も得意な言語は Python、最も影響を受けた言語は Clojure であり、シンプルな関数型（的書き方ができる）言語を好む。関数型的記法を持ちながら、実行性能が高い Rust に興味を持ち研さんを続けている。

編集：@IT 編集部
発行：アイティメディア株式会社