Toyoda Yuki

Software Engineer

```rust
use actix_web::{get, App, HttpResponse, HttpServer

#[get("/health")]
async fn hc() -> impl Responder {
    HttpResponse::Ok().body("Can we build web appl
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().service(hc))
        .bind("127.0.0.1:8080")?
        ()
        it
```

Session

Rustで Webアプリケーションは どこまで開発できるのか

Yuki Toyoda (a.k.a. yuki)

is a Software Engineer @ Dynalyst
& Next Experts in Rust

furthermore,

Coauthor of 『実践Rustプログラミング入門』

Co-organizer of Rust.Tokyo & RustFest Global

@helloyuki_

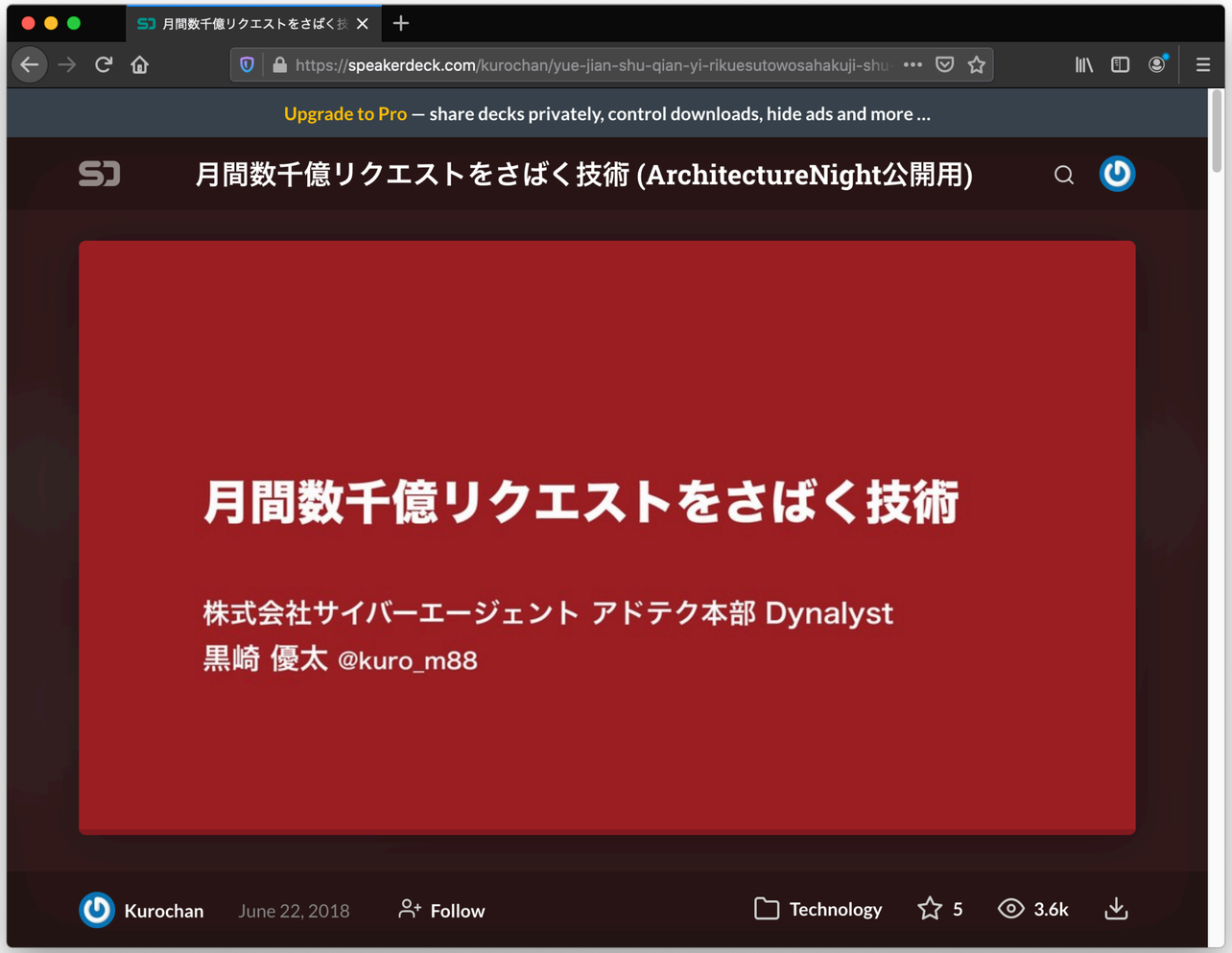
Next Experts

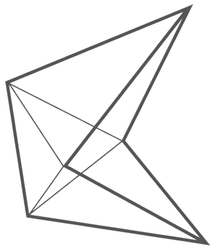
Dynalyst

Dynalyst

詳しくは弊社
黒崎 @kuro_m88 の
スライドをご覧ください。

https://speakerdeck.com/kurochan/yue-jian-shu-qian-yi-rikuesutowosahakuji-shu-architecturenightgong-kai-yong





Next
Experts

Next Expertsは特定領域への貢献意欲や
一定の実績を有し、将来的なDeveloper Expertsを
目指すエンジニアのための活動支援制度です。

サイバーエージェントにはDeveloper Expertsと
呼ばれる制度が存在します。

自身の技術的な専門領域によって、
サイバーエージェント全体への貢献を目指します。

1. Rust とは

2. Rust で Webアプリケーションを
   作ると何が嬉しいのか？

3. Rust を実際のプロダクトに入れてみた

1. Rust とは

2. Rust で Webアプリケーションを
   作ると何が嬉しいのか？

3. Rust を実際のプロダクトに入れてみた

Rustとは

What’s Rust?

システムプログラミング言語

Rustはシステムプログラミング言語

低レイヤーを扱う

RustはOSやコンパイラなどの低レイヤーのソフトウェアを扱うために設計された。

C/C++ 並の速度

GCがない、機械語を直接扱えるなどの理由から、C/C++並の速度を実現できている。

メモリ安全である

言語独自の機能により、メモリ安全にコードを書くことができる。セキュリティ向上に貢献できる。

Rustはシステムプログラミング言語

低レイヤーを扱う

RustはOSやコンパイラなどの低レイヤーのソフトウェアを扱うために設計された。

C/C++ 並の速度

GCがない、機械語を直接扱えるなどの理由から、C/C++並の速度を実現できている。

メモリ安全である

言語独自の機能により、メモリ安全にコードを書くことができる。セキュリティ向上に貢献できる。

Rustはシステムプログラミング言語

低レイヤーを扱う

RustはOSやコンパイラなどの低レイヤーのソフトウェアを扱うために設計された。

C/C++ 並の速度

GCがない、機械語を直接扱えるなどの理由から、C/C++並の速度を実現できている。

メモリ安全である

言語独自の機能により、メモリ安全にコードを書くことができる。セキュリティ向上に貢献できる。

Rustはシステムプログラミング言語

低レイヤーを扱う

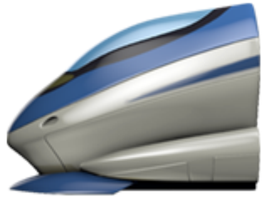
RustはOSやコンパイラなどの低レイヤーのソフトウェアを扱うために設計された。

C/C++ 並の速度

GCがない、機械語を直接扱えるなどの理由から、C/C++並の速度を実現できている。

メモリ安全である

言語独自の機能により、メモリ安全にコードを書くことができる。セキュリティ向上に貢献できる。



次の40年のための
プログラミング言語

A Language for the Next 40 years

Rust: A Language for the Next 40 Years - Carol Nichols
https://www.youtube.com/watch?v=A3AdN7U24iU

1. Rust とは

2. Rust で Webアプリケーションを
作ると何が嬉しいのか？

3. Rust を実際のプロダクトに入れてみた

Rustで
Webアプリケーション
を作ると何が嬉しいのか？

What’s the benefit of using Rust in web development?

低レイヤー向けの言語で Webアプリケーションは作れるの？

C/C++ にそのイメージがないんだよね。

A.

作れます。

しかも、驚くほど快適に ✨

RustでWebアプリケーションを作る良さ

1. パフォーマンスの心配をしなくていい
2. 表現力豊かな言語機能
3. 強く型付けする = 型がドキュメントに

パフォーマンスの心配をしなくてよい

基本的に最速

時と場合によるが、基本的に最速の部類の言語なので、余計なパフォーマンスの心配をしなくて良くなる。

起動も速い

VMやランタイムがないので、アプリケーションの起動時間も気ににになくてよくなる。
コンテナとの相性もいい。

安定的

GC がないので、その分パフォーマンスのブレが少なくなり、安定化する。

パフォーマンスの心配をしなくてよい

基本的に最速

時と場合によるが、基本的に最速の部類の言語なので、余計なパフォーマンスの心配をしなくて良くなる。

起動も速い

VMやランタイムがないので、アプリケーションの起動時間も気ににになくてよくなる。
コンテナとの相性もいい。

安定的

GC がないので、その分パフォーマンスのブレが少なくなり、安定化する。

パフォーマンスの心配をしなくてよい

基本的に最速

時と場合によるが、基本的に最速の部類の言語なので、余計なパフォーマンスの心配をしなくて良くなる。

起動も速い

VMやランタイムがないので、アプリケーションの起動時間も気ににになくてよくなる。
コンテナとの相性もいい。

安定的

GC がないので、その分パフォーマンスのブレが少なくなり、安定化する。

パフォーマンスの心配をしなくてよい

基本的に最速

時と場合によるが、基本的に最速の部類の言語なので、余計なパフォーマンスの心配をしなくて良くなる。

起動も速い

VMやランタイムがないので、アプリケーションの起動時間も気ににになくてよくなる。
コンテナとの相性もいい。

安定的

GC がないので、その分パフォーマンスのブレが少なくなり、安定化する。

表現力豊かな言語機能

- トレイトやジェネリクスによる抽象と実装の切り離しが可能。
- 列挙型を駆使した見通しのよいコード。
- マクロにより自分のアプリケーションに合うように
  言語をカスタマイズ。

マクロによるDSLの構築

```
use actix_web::{get, web, App, HttpServer, Responder};

#[get("/{id}/{name}/index.html")]
async fn index(web::Path((id, name)): web::Path<(u32, String)>) ->
impl Responder {
    format!("Hello {}! id:{}", name, id)
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().service(index))
        .bind("127.0.0.1:8080")?
        .run()
        .await
}
```

マクロによるDSLの構築

```
macro_rules! mdo {
    ($i:ident <- $e:expr; $($t:tt)*) => {
        $e.bind(move |$i| mdo!($($t)*))
    };
    ($e:expr; $($t:tt)*) => {
        $e.bind(move |()| mdo!($($t)*))
    };
    (ret $e:expr) => {
        $e
    };
}
```

マクロによるDSLの構築

```
let actual = mdo! {
    x <- Option::pure(1);
    y <- Option::pure(2);
    ret Option::pure(x + y)
};

assert_eq!(actual, Some(3));
```

強く型付けする = 型がドキュメントに

- New Type パターンを使うと値クラスを
  オーバーヘッドなしで表現できる。
- たとえばバリデーションチェックの情報を
  型に落とすことができる。

New Type パターン

```
struct UserId(u64);

struct User {
    id: UserId,
}
```

New Type パターン

```
trait Repository<T, I> {
    fn find(&self, key: I) -> Option<T>;
}

impl Repository<User, UserId> for UserRepository {
    fn find(&self, key: UserId) -> Option<User> {
        todo!()
    }
}
```

バリデーション情報を型に落とす

```
use std::marker::PhantomData;

struct LowerEqualThan500Kb;
struct LowerEqualThan1000Kb;

struct LimitedSizeFile<T>(File, PhantomData<T>);

impl LimitedSizeFile<LowerEqualThan500Kb> for CsvData {
    // 500 KB 以下に収まっていて欲しいファイル
}

impl LimitedSizeFile<LowerEqualThan1000Kb> for Thumbnail {
    // 1000KB (1MB) 以下に収まっていて欲しいファイル
}
```

でも、

フレームワークがないんでしょう？

代表的なフレームワーク

- actix-web
- Rocket
- Warp
- tide (こちらは現在開発中)

代表的なフレームワーク

- actix-web
- Rocket
- Warp

tokioベース

- tide (こちらは現在開発中)

async-stdベース

非同期処理ランタイムがいくつかある

- Rust は用途が広範な言語
- それぞれの用途に適したランタイムが必要になる
- tokio と aysnc-std が現在注目されている

tokio と async-std

tokio

歴史がある。エコシステムが充実している。

async-std

標準ライブラリと同じ API で非同期処理 API を提供するのが目標。

充実しはじめたエコシステム

サーバーサイドフレームワーク以外にもライブラリが充実し始めている。

- serde：JSON パーサー
- diesel：O/R マッパー
- tonic：gRPC を扱う
- juniper：GraphQL を扱う
- rusoto：AWS SDK（ちなみに先日、公式が SDK をアルファリリースした）

🌟 必要なものは揃っている。あとは使うだけ。

1. Rust とは

2. Rust で Webアプリケーションを
   作ると何が嬉しいのか？

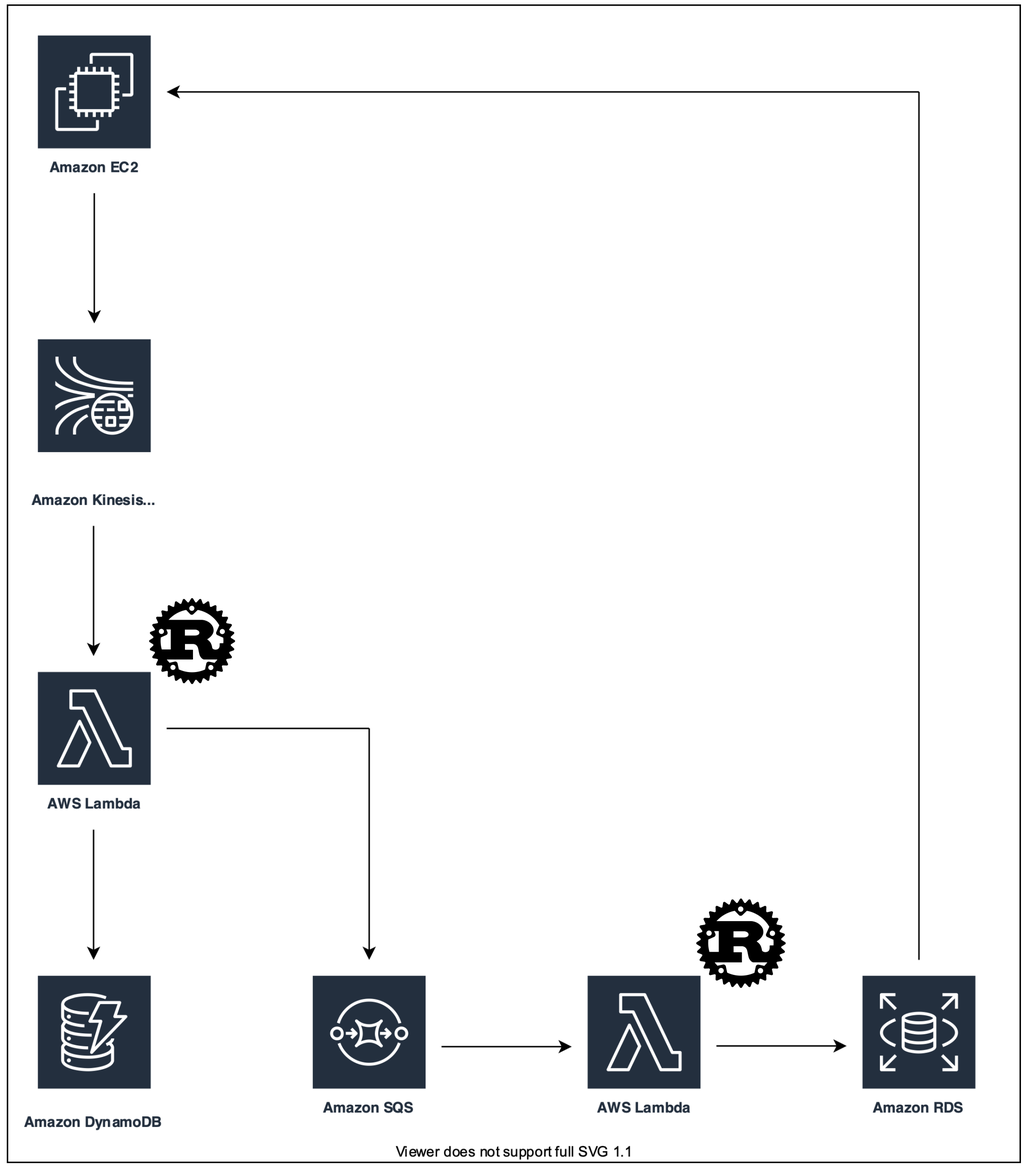
3. Rust を実際のプロダクトに入れてみた

Rustを
実際のプロダクトに
入れてみた

A brief story of our “Rust in Production”

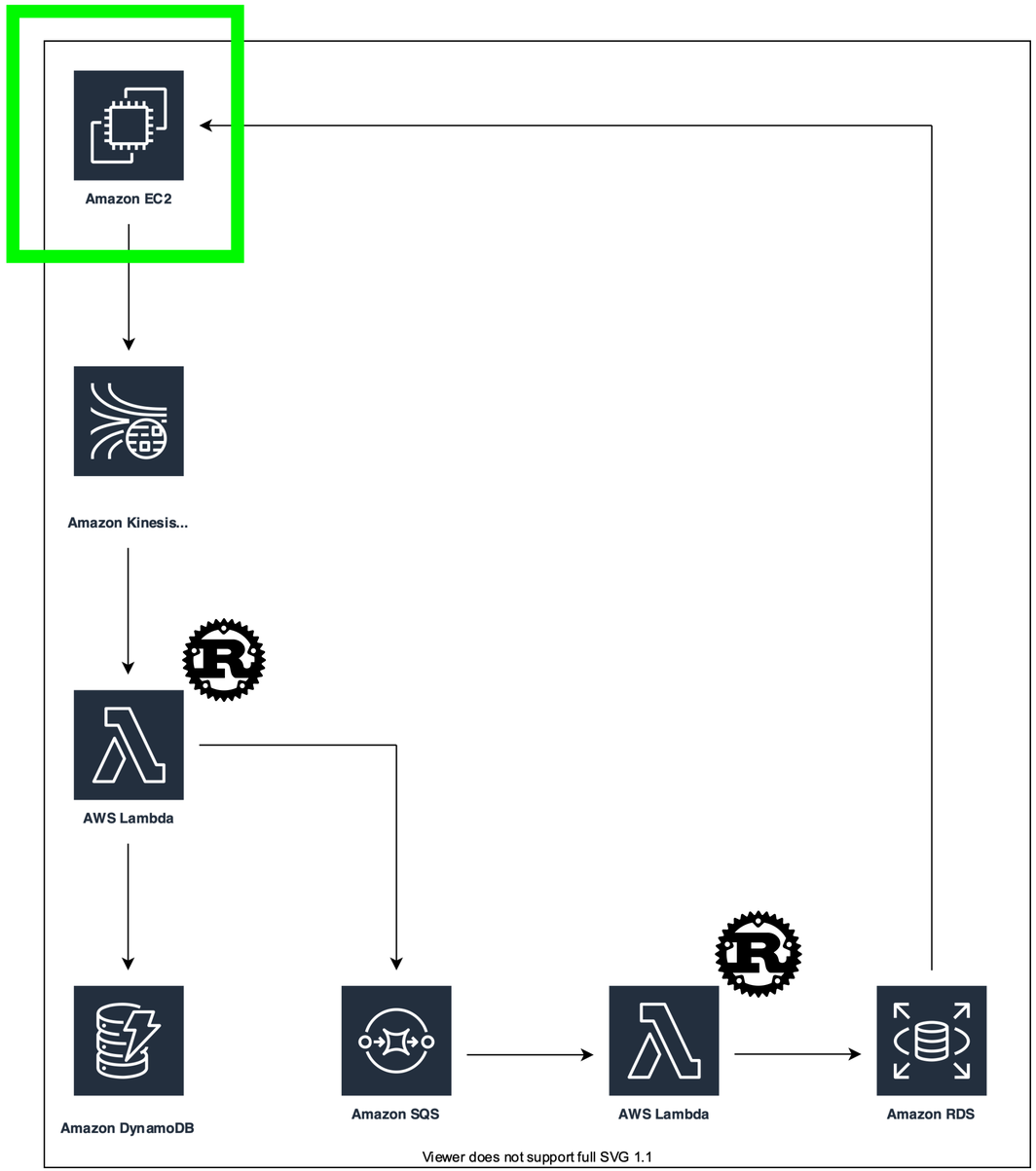
事例紹介: AirTrack

- 位置情報を使った広告配信を行うプロダクト。
- セグメント情報の保存部分に Rust を使用した。
- レコードは最大で十数億件。



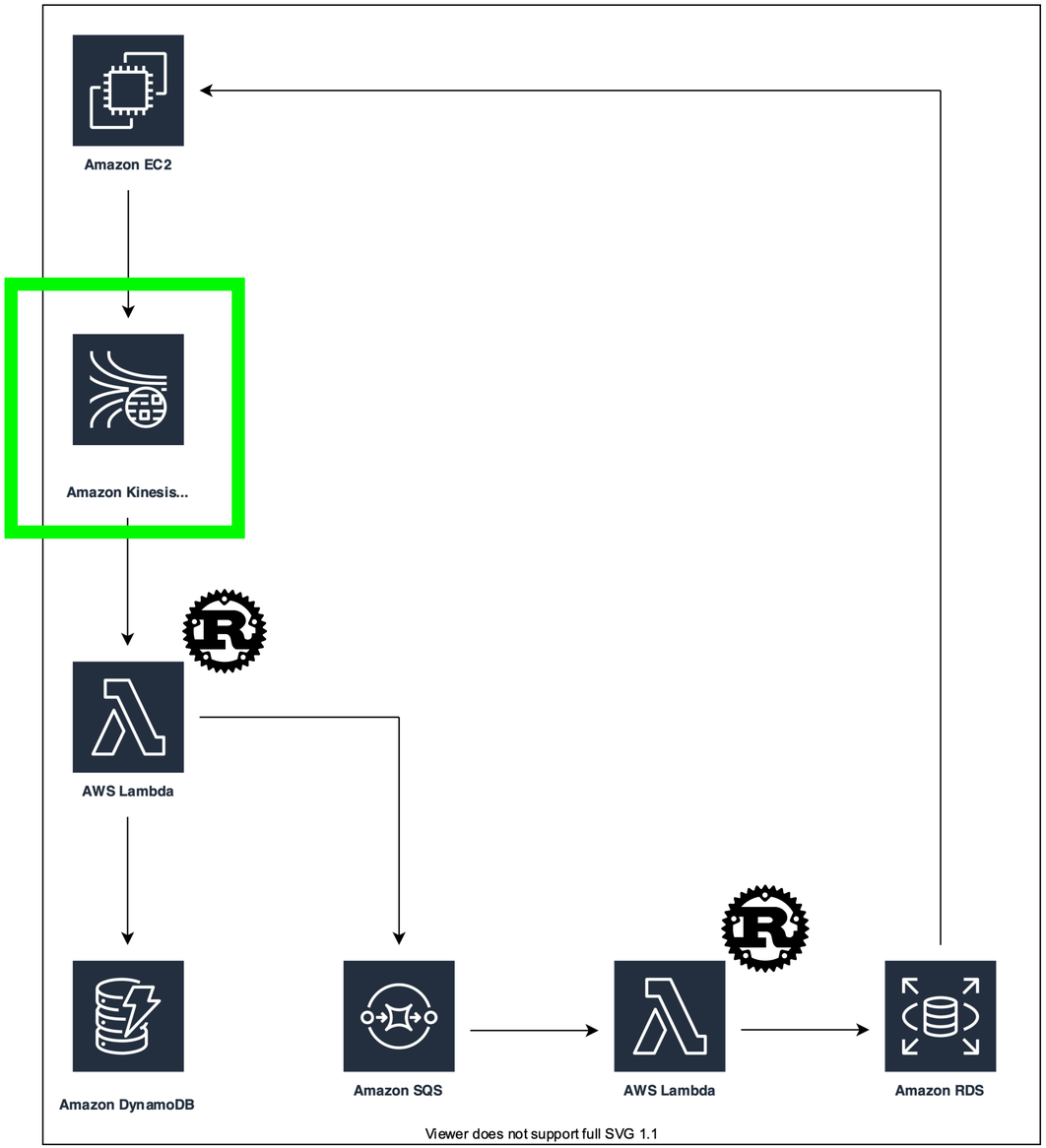
アーキテクチャ

- EC2 上にアプリーケーションは存在する。
- シャードの数を調整して、DynamoDB への書き込みアクセスを調整する Kinesis Data Streams。
- データを DynamoDB へ入れる。
- 保存時に起きた失敗や、保存後の件数を RDS に保存する。



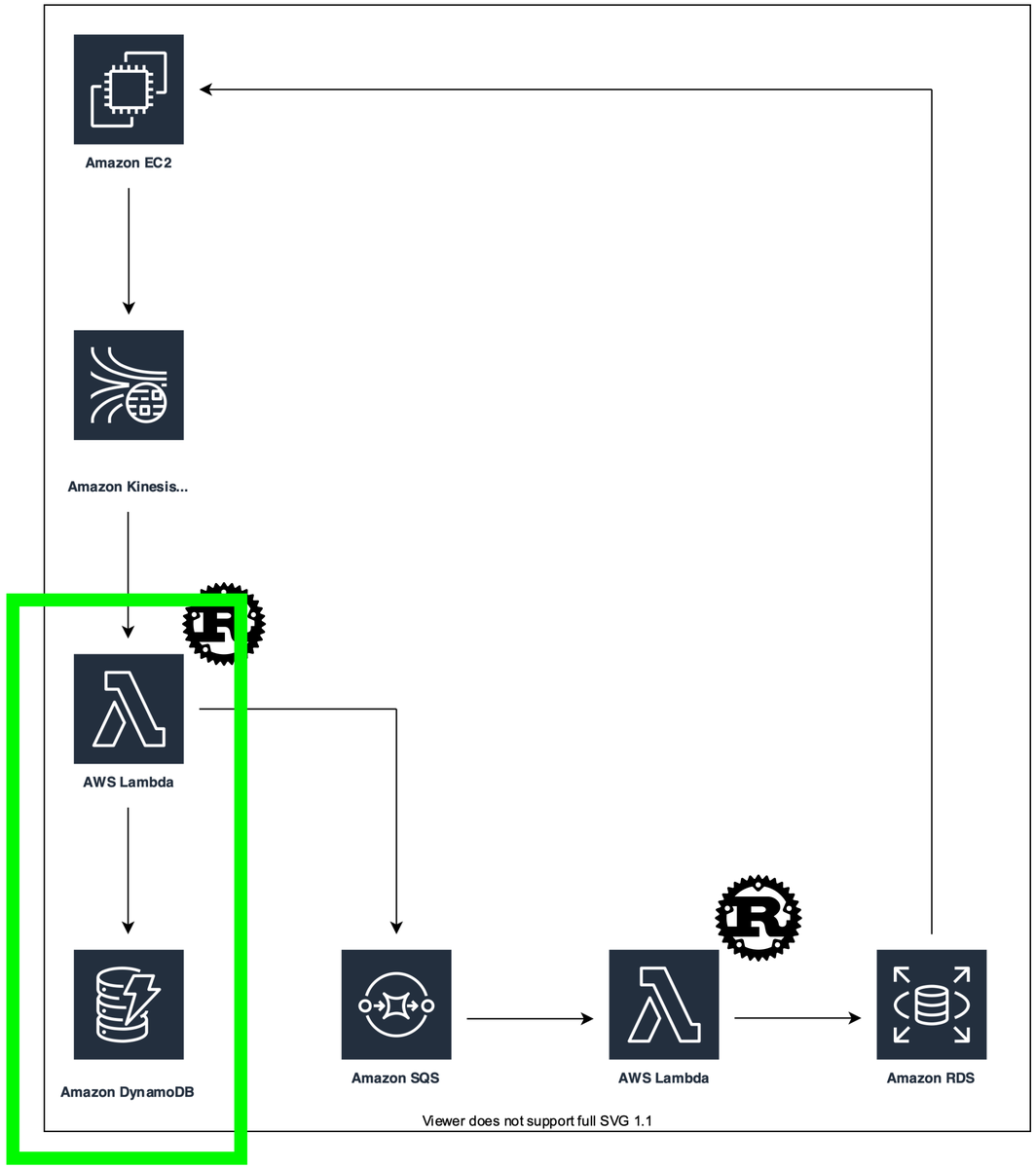
アーキテクチャ

- EC2 上にアプリーケーションは存在する。
- シャードの数を調整して、DynamoDB への書き込みアクセスを調整する Kinesis Data Streams。
- データを DynamoDB へ入れる。
- 保存時に起きた失敗や、保存後の件数を RDS に保存する。



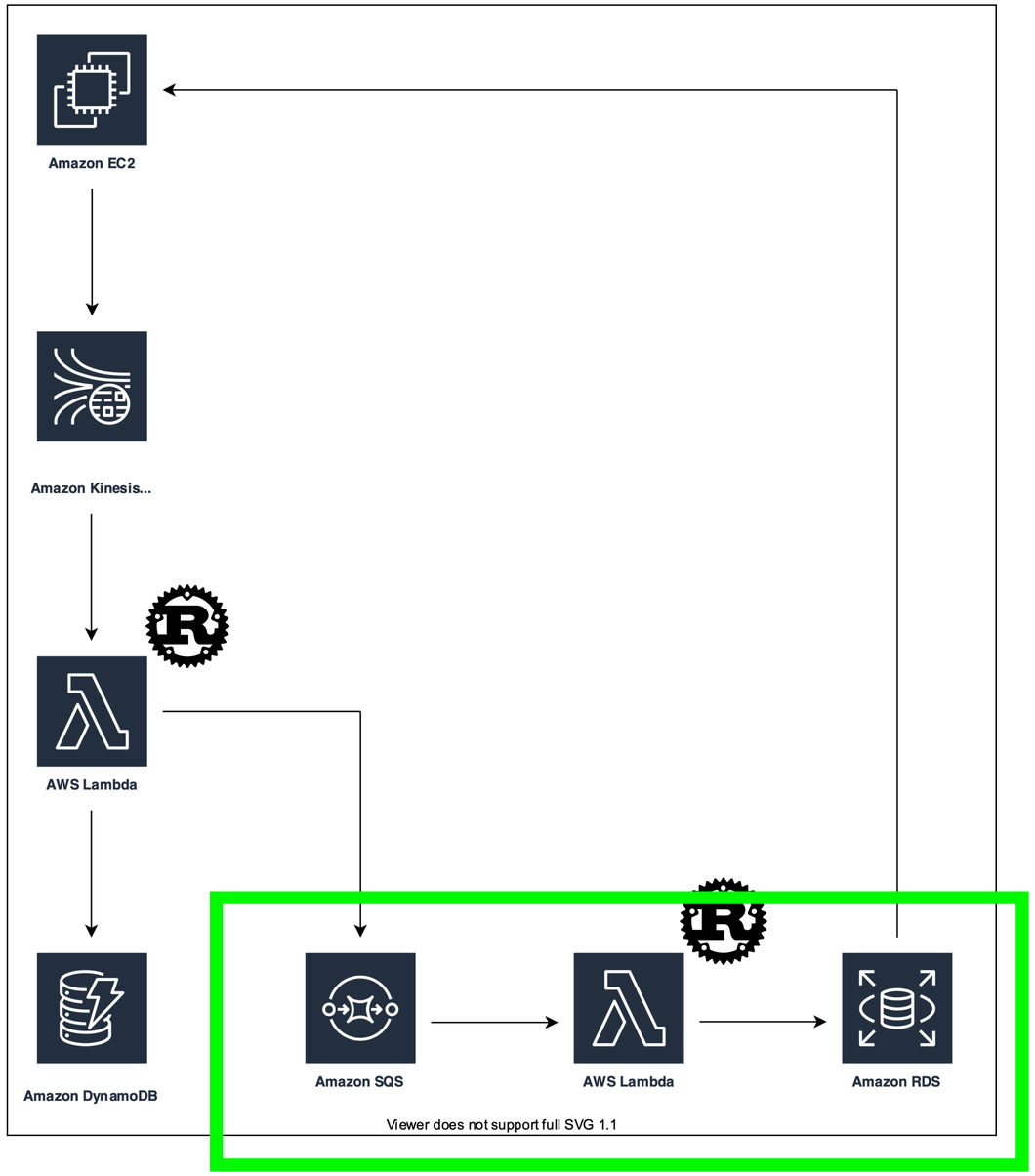
アーキテクチャ

- EC2 上にアプリーケーションは存在する。
- シャードの数を調整して、DynamoDB への書き込みアクセスを調整する Kinesis Data Streams。
- データを DynamoDB へ入れる。
- 保存時に起きた失敗や、保存後の件数を RDS に保存する。



アーキテクチャ

- EC2 上にアプリーケーションは存在する。
- シャードの数を調整して、DynamoDB への書き込みアクセスを調整する Kinesis Data Streams。
- データを DynamoDB へ入れる。
- 保存時に起きた失敗や、保存後の件数を RDS に保存する。



アーキテクチャ

- EC2 上にアプリーケーションは存在する。
- シャードの数を調整して、DynamoDB への書き込みアクセスを調整する Kinesis Data Streams。
- データを DynamoDB へ入れる。
- 保存時に起きた失敗や、保存後の件数を RDS に保存する。

Rust を AWS Lambda で使う

AWS Lambda と Rust の相性

カスタムランタイムがあり、連携が楽。

メモリのフットプリントが小さい傾向にある。使用メモリも小さくできる。

処理速度が速い
= Lambdaでの課金が少ない

AWS Lambda と Rust の相性

カスタムランタイムがあり、連携が楽。

メモリのフットプリントが小さい傾向にある。使用メモリも小さくできる。

処理速度が速い
= Lambdaでの課金が少ない

AWS Lambda と Rust の相性

カスタムランタイムがあり、連携が楽。

メモリのフットプリントが小さい傾向にある。使用メモリも小さくできる。

処理速度が速い
= Lambdaでの課金が少ない

AWS Lambda と Rust の相性

カスタムランタイムがあり、連携が楽。

メモリのフットプリントが小さい傾向にある。使用メモリも小さくできる。

処理速度が速い
= Lambdaでの課金が少ない

AWS Lambda と Rust の相性

カスタムランタイムがあり、連携が楽。

メモリのフットプリントが小さい傾向にある。使用メモリも小さくできる。

処理速度が速い = Lambdaでの課金が少ない

つまり相性がいいってこと

運用してみた結果

- 思惑通り、速度面はまったく気にするポイントがなかった。
- Kinesis Data Streams 側のがんばりもあるが、
  メモリのフットプリントも最小構成で済んだ。
- 運用時の安心感に Rust 導入は貢献したと思う。

Rust を Scala チームで使う

Scala チームで Rust を使う

- プロダクトの大部分は Scala で記述されている。
- Scala と Rust は書き方が似ている部分が多い。

⚠ うちのチームではこうだった、という話をこれからします。

Scala にある文法は Rust にもある

いろいろあるリスト↓

- トレイトはお互いの言語に存在する。
- パラメトリック多相は Rust にも搭載されている。
- implicit を用いて実現するアドホック多相は Rust ではトレイトで実現できる。
- パターンマッチは Rust にもある。
- for-yield やモナドを用いた実装は Rust ではできない。
- が、map や flatmap などの基本的なコンビネータは Rust でも使える。

コードを書く際の考え方の類似

似ている点、たとえば…

- 扱いたいデータを代数的データ型に落としてパターンマッチする。
- デフォルトで変数宣言は不変。可変にしたい際は変数のスコープを絞る。
- 手堅く型付けをしながらプログラムを記述していく。
- map, filter, flatMap, etc など。

💛 Scala で書くのと似たロジックで Rust のコードを書けた

広告配信は比較的単純なので…

- 参照を構造体に持たせてぐるぐる使い回す、という
  ロジックが 登場しにくい。

- そこまでリソースがシビアではないので、
  いざというときにはcloneするのも手。

(もちろんサイズの小さいデータに限る; また、そういう部分は後で私が直す、という運用をした)

🐣 手軽に Rust を書くというスタンスを採り入れた

苦労していたポイント

- Rust は標準ライブラリが薄い
- async エコシステムが発展途上
- Rust 特有のワークアラウンド

Rust は標準ライブラリが薄い

- 正確には「薄く作られている」。
  互換性の担保のしやすさのため。
- なので、ちょっとしたことでもサードパーティーの
  ライブラリ（クレート）を使う必要がある。
- 「使えるクレート」の知識が別途必要になる側面がある。

async エコシステムが発展途上

- そもそも 2018 年までまともな非同期処理基盤が存在しなかった。
- 2018年ごろに async/.await という構文が導入された。
- 先日ようやく tokio が 1.0 に到達した。
- それに伴うライブラリ側の対応のアップデートが多々あった。
- …という話を知っていないとわかりにくいコンパイルエラーが発生した。

Rust のワークアラウンド

- ある型からある型へ変換する際には From トレイトが使用可能、といったような定番のパターンは最初は気づきにくい。
- ペアプロなどを実施し、その中で覚えていってもらうことで対応した。
- Rust Design Patterns のような資料もあるので、それを読んでもらうのも手。

※ Rust Design Patterns: https://rust-unofficial.github.io/patterns/

知識の伝播の必要性

- Rust で一部書いたものの、メンテできる人が退職 or 異動などでメンテできなくなってしまったという事例をよく聞く。
- 自身も異動を控えていたこともあり、Rust を使える人をどう増やすか悩んでいた。
- Next Experts に就任したタイミングで Rust ハンズオンを実施し、参加してもらうことで対応した。

入れたあとは覚悟を持って
1から Rust を教える気概で

まとめと展望

Wrap up & future plan

まとめと展望

- Scala エンジニアは Rust に入りやすいかもしれない。
- Rust をやるなら AWS Lambda からはじめるのは
  おすすめできる。
- Rust の活用はこれからも期待される。

CA BASE NEXT

CyberAgent Developer Conference by Next Generations