Ruby on Rails on AWS Lambda

@joker1007

銀座Rails #35

Self-intro

- @joker1007
- Repro inc. ex-CTO -> Chief Architect
  - Ruby/Rails
  - fluentd/embulk
  - RDB
  - Docker/ECS
  - Kafka
  - Bigquery/EMR/Hive/Presto/Cassandra

Agenda

- RailsをLambdaで動作動作させる方法とユースケース
- これからの非同期処理のあり方とマイクロサービス

AWS Lambda Container Image Support

2020年の12月辺りにAWS Lambdaにコンテナイメージをデプロイし任意の実行環境のコードを呼べる様になりました。
今迄も頑張ればRailsをLambda上で動かすことは可能だったのですが、ネイティブライブラリのコンパイル環境を考えると非常に面倒だった。

コンテナイメージがサポートされたことで、通常のデプロイフローの中に混ぜ込むことが容易になり活用が現実的になった。

コンテナイメージを利用する時、AWS Lambdaはどの様に動作するのか

Lambdaの作成時

作成する時にコンテナイメージのURLを指定することで、コンテナを利用したLambdaを作成できます。
RubyによるAPI呼び出しだと以下の様になります。

```
client.create_function(
  function_name: "function_name",
  role: role_name,
  package_type: "Image", # Imageを指定する
  code: {
    image_uri: image_registry_uri, # 通常はECRを利用
  },
  image_config: { # コンテナ設定の上書き
    entry_point: ["aws_lambda_ric"],
    command: ["main.LambdaEntry::RakeHandler.process"],
    working_directory: "/app/lambdas/rake_handler",
  },
  # 省略
)
```

Lambdaの挙動 (1)

関数が作成されるとコンテナイメージが内部でキャッシュされる。
それによって起動にかかる時間はイメージサイズに余り影響されない。
FargateやECSによる起動の場合イメージのダウンロードを挟むのでイメージサイズが大きいと起動まで数分かかるケースがあるが、Lambdaの場合はコールドスタートでも30秒程で起動する。
一度起動した後ならほぼ待ち時間無しで起動できる。

Lambdaの挙動 (2)

内部で動作するコンテナイメージはLambda Runtime APIを実装し、規約に則った動作をする必要がある。
Python, Node, Java, .NET, Go, Rubyは公式のクライアントが存在するのでそれを利用できる。

RubyのLambda Runtime Interface Clientについて

1. コマンドラインからハンドラーを引数にして起動する
2. 起動すると環境変数からLambda Runtime APIの接続先を取得する
3. 起動後にrun loopが開始され一定期間プロセスが生存し続ける
4. run loopの中で実行待ち状態になっている関数呼び出しをAPIを利用してpollingする
5. pollingによって取得した情報を元にハンドラーのメソッドを呼び出す
6. 戻り値をJSONシリアライズしてAPIにPOSTし関数の完了とする

上記の処理は基本的にシングルスレッド・シングルプロセスで動作する。
現状では言語に関わらずそういう実装が推奨されている。
(中の人に確認済みだがドキュメントに明記されている訳ではないので変わる可能性がある)

注意点

プロセスが一定期間生存するため、グローバルに状態を変更する処理を実行してしまうと、次の呼び出しに影響が出る可能性がある。
例えばアプリケーションコード内でENVを設定すると、プロセスが生きてる限りはそれが引き継がれる。

一方でRubyの公式の実装ではシングルプロセス・シングルスレッドで実行され、並列呼び出しの制御はLambdaが面倒を見てくれる。
処理の本体自体がシングルスレッドである限りはレースコンディションは気にしなくて良い。

Railsで動作させるには

普通のRailsアプリケーションのコンテナイメージに`aws_lambda_ric`というgemを追加する。

```
WORKDIR /app
RUN mkdir -p vendor
COPY Gemfile Gemfile.lock /app/
RUN bundle install && \
  bundle clean
RUN gem install aws_lambda_ric # 追加
COPY . /app
```

エントリポイント/コマンドの設定

Railsのコンテナイメージで、aws_lambda_ricコマンドを呼び出す様に設定を上書きしておく必要がある。
先に紹介したcreate_functionの例を参照。

ファイルシステムに関する注意

Lambdaの実行環境ではコンテナ内部のファイルシステムが基本的にfreezeされる。
/tmpなどは書き込み可能だが、WORKDIR以下のファイルは一切変更ができなくなる。

Railsにおいては以下の様なケースで問題になる。

- 起動時に環境に合わせてdatabase.ymlを生成している
- assets precompileを行う必要がある
- bootsnapが有効になっていてcacheが無い
- /tmp以外にファイル出力する様なコードを実行しようとしている

弊社ではdatabase.ymlをerbから生成するコードとbootsnapで問題になったので一部調整が必要になった。
昨今ならParameterStore等を使って環境変数を整えるなどをした方が良い。

Rakeを呼び出したい場合の注意

Rakeは一度実行した処理を再度実行しない様にinvokeを記録している。
Lambdaの挙動として起動後のプロセスが一定期間生存して再利用されるので、
Lambdaの中でRakeを呼び出すと次回以降の呼び出しの時に実行されなくなる。
解決策としては以下の様な方法になる。

- そもそもRakeを使わない様にする
- RakeのInvoke情報を都度クリアする
- 内部でforkして子プロセスを立ち上げる (実現できた)

ユースケース

現状で一番フィットするユースケースは以下の様なものだと思う。

- 15分以下で確実に完了する小規模バッチ
- ActiveJobの代替
- SQSやStep Functionと組み合わせる

ALBやAPI Gatewayから起動することでWebリクエストにも対応可能だとは思うが、コントローラーを通すのが大変なので、通常のWebリクエストであれば普通にECSを使う方が良いだろう。

弊社での活用

Embulkを利用したimport処理の後実行ステータスの更新やメール配信ジョブの実行等、sidekiqでやる様な処理を実行している。
StepFunctionを利用してワークフローを構築し、複数のステップに分けて実行状況を可視化している。
元々はステータス更新にFargateを利用していたが、起動時間が遅いためLambdaに置き換えた。
常に3分かかっていたのが、ホットスタンバイ状態であれば2秒で処理が終わる様になった。

デプロイについて

CapistranoでECSを更新しているが、デプロイが完了したらafter hookでLambdaのAPIを叩いてコンテナイメージのバージョンを更新する様にしている。
イメージ自体はWebリクエストを受け付けるRailsアプリと同じものを利用している。
これにより通常のアプリケーションデプロイフローに完全に統合できた。

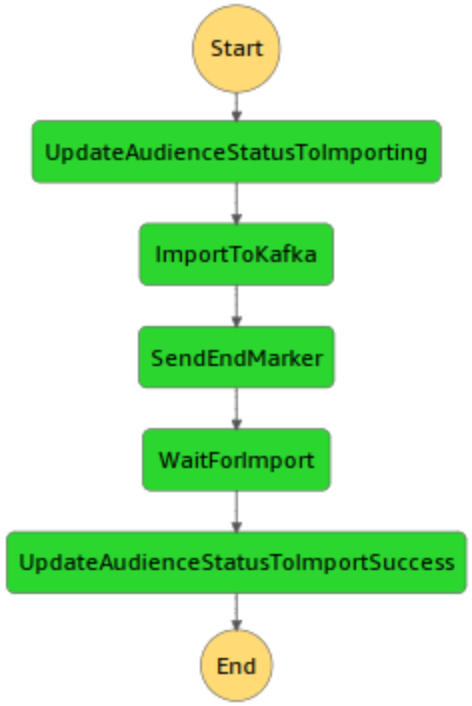
Step Functionと組み合わせる利点

StepFunctionであればユーザーから見た非同期処理を多段に繋げて実行状況を簡単に可視化できるし、ボタン一発でリトライ出来て運用が簡単になる。
実行ログ等もコンソールから簡単にCloudWatch Logsを参照できる。

しばしばsidekiqのジョブからsidekiqのジョブを呼ぶという非常に処理の流れが分かりにくい実装が世の中には存在するが、それを避けてこういったワークフローツールを活用する方が圧倒的に見通しが良い。

Step Functionのグラフインスペクタの例

こんな感じで実行状況が可視化される。



Rails on Lambdaの所感

AWSを使う限りではActiveJob/sidekiqはもう要らないんじゃないかという気がする。
非同期処理は呼び出しプロトコルだけ決めてSQSにペイロードを投げる様にすれば良い。
後はそこからstep functionを起動すれば、サーバーリソース無しで同時実行が数千ぐらいまでは余裕でスケールでき、マルチスレッドにおけるメモリ消費量も気にしなくて良い実行環境が手に入る。
CloudWatch Eventsを利用すればスケジュール起動も可能。
更にsidekiqと違ってSQSを活用すれば安全なat_least_onceを実現するのが非常に簡単になる。

マイクロサービスへの応用

Railsのコードベースを利用したまま単機能の小さな処理を独立させることが簡単になってきた。
Step Functionを組み合わせることで複雑な処理をサーバーレスでリトライアブルに実現できる。
必要になったら単機能だけを別言語の実装に置き換えることも容易。

こういった性質から、Webリクエストに依存しない処理を小さくLambda関数にして切り出し、マイクロサービスとして独立させることに応用可能だと考えている。
この場合、Step Functionが一つのサービスの単位になるだろう。

gem化していない理由

Lambdaの起動方法にはかなりバリエーションがあって、用途に合わせて呼び出し方が異なる。
呼び出し方の規約は各々の環境で決める方が良い。
実際のところペイロードとハンドラさえ書けば後はクラウドサービス側の設定なのでGemを作る程でもない。
加えて、自分はActiveJobの様な最大公約数的なインターフェースは後々困ることが多いので余り積極的に使わない方が良いと思っていて、そのインターフェースを整えるモチベーションも余り無かった。

We are hiring!!

Repro inc.