

An Introduction to AWS Lambda and Event Driven Programming

impress
top gear



AWS Lambda 実践ガイド

アーキテクチャと
イベント駆動型プログラミング

大澤 文孝 = 著

実践的なプログラミング手法を解説

サーバーレスシステムとは／Lambdaの仕組み

Lambdaと連携するAWSサービス／S3のイベント処理

API Gateway・DynamoDB・SESとの連携／並列処理とSQS

インプレス

An Introduction to AWS Lambda and Event Driven Programming

impress
top gear



AWS Lambda 実践ガイド

アーキテクチャと
イベント駆動型プログラミング

大澤 文孝 = 著

実践的なプログラミング手法を解説

サーバーレスシステムとは／Lambdaの仕組み
Lambdaと連携するAWSサービス／S3のイベント処理
API Gateway・DynamoDB・SESとの連携／並列処理とSQS

インプレス

●本書の利用について

- ◆本書の内容に基づく実施・運用において発生したいかなる損害も、株式会社インプレスと著者は一切の責任を負いません。
- ◆本書の内容は、2017年8月の執筆時点のものです。本書で紹介した製品／サービスなどの名称や内容は変更される可能性があります。あらかじめご注意ください。
- ◆Webサイトの画面、URLなどは、予告なく変更される場合があります。あらかじめご了承ください。
- ◆本書に掲載した操作手順は、実行するハードウェア環境や事前のセットアップ状況によって、本書に掲載した通りにならない場合もあります。あらかじめご了承ください。

●商 標

- ◆Amazon Web Services、Amazon Web Services ロゴは、Amazon Web Services 社の米国および他の国における商標です。
- ◆その他、本書に登場する会社名、製品名、サービス名は、各社の登録商標または商標です。
- ◆本文中では、®、©、TM は表記しておりません。

はじめに

AWS で展開されている現行システムのほとんどは、開発したプログラムを動かすために、仮想サーバーとして EC2 インスタンスを利用しています。仮想サーバーという違いこそあれ、構成そのものは、オンプレミスのときと大きく変わっていません。

この構成を大きく変えるのが、AWS Lambda です。Lambda は、サーバーを必要としないプログラムの実行環境です。開発者が処理したい内容を小さな関数として実装すると、必要に応じて、それが実行される仕組みです。

サーバーを必要としないということは、運用の手間やコストを削減できるということです。この利点はとても大きいので、新規の開発案件では、Lambda が採用されるケースが増えました。しかし、Lambda に魅力があるからといって、すぐに移行できるほど話は簡単ではありません。従来の EC2 インスタンスを使った開発と Lambda を使った開発とでは、プログラミングの方法はもちろん、設計の考え方も大きく異なるからです。Lambda を使い始めるには、Lambda の仕組みの理解と、その特性を活かした設計の考え方を習得することが不可欠です。

ひと言で言うと、Lambda を使った開発では、AWS サービスの操作に重きを置きます。Lambda を活かせるかどうかは、AWS の各種サービス——ストレージの S3、プッシュサービスの SNS、メール送信の SES など——を、いかに使いこなせるかに左右されます。そこで本書では、実例を通じて、こうした AWS サービスの実際の使い方にも言及するよう努めました。

筆者が本書を通じて言いたいのは、「勘所さえつかめば、Lambda はとても簡単」ということです。とても短いコードで実用的な処理を書けるため、一度、Lambda を使ったら、もう EC2 インスタンスを使った開発には戻れなくなっていても不思議ではありません。

読者の皆様が本書を通じて Lambda に触れあい、その結果、開発効率やシステムの堅牢性の向上につながれば、幸いです。

2017 年 9 月吉日

大澤文孝

●本書の表記

- ・注目すべき要素は、太字で表記しています。
- ・コマンドラインのプロンプトは、“>” や “\$” で示します。
- ・プログラムリストが右端で折り返す場合は、“⇒” で示されます。
- ・プログラムに関する説明は、言語仕様のコメント（“#” や “//”）で記しています。
- ・プログラムや実行結果の出力を省略している部分は、「...」で表記します。
- ・GUI の “→” は、マウスでクリックする部分を、入力部分や参照など注目すべき部分は、四角枠で示します。

例：



●実行環境

◆サービス

- ・AWS Lambda
- ・AWS IAM (Identity and Access Management)
- ・Amazon S3
- ・Amazon EC2
- ・Amazon DynamoDB
- ・Amazon SNS (Simple Notification Service)
- ・Amazon SQS (Simple Queue Service)
- ・Amazon SES (Simple Email Service)

◆ソフトウェア

- ・Windows PowerShell
- ・Python3
- ・virtualenv

●本書での操作上の注意

◆AWS アカウントの作成

本書で解説する AWS の操作では、すでに AWS のアカウントは作成済みであり、AWS にログインして、マネジメントコンソールにアクセスできる状態を前提にしています。まだ AWS のアカウントを作成していない方は、以下に示す URL の掲載内容を参考にして、あらかじめ AWS アカウントを作成しておいてください。

<https://aws.amazon.com/jp/register-flow/>

◆セキュリティおよびコストの管理

本書では、AWS の運用上必要となる、サービスのコスト管理や多要素認証（MFA）仮想デバイスの有効化などについては解説していません。必要に応じて各自で設定してください。また、AWS の利用前には、意図しない課金を避けるため、無料枠のサービスの範囲（条件）、各サービスの料金体系、課金レポートおよび予算設定など、十分理解したうえで、AWS のサービスを利用してください。

AWS が提供するユーザーガイドでは、以下の URL などが参考になります。

- Identity and Access Management ドキュメント

http://docs.aws.amazon.com/ja_jp/IAM/latest/UserGuide/introduction.html

- MFA 仮想デバイスの有効化

http://docs.aws.amazon.com/ja_jp/IAM/latest/UserGuide/id_credentials_mfa_enable_virtual.html

- AWS 請求情報とコスト管理

- AWS Billing and Cost Management とは何か

http://docs.aws.amazon.com/ja_jp/awsaccountbilling/latest/aboutv2/billing-what-is.html

- 無料利用枠の使用

<http://aws.amazon.com/jp/free/>

http://docs.aws.amazon.com/ja_jp/awsaccountbilling/latest/aboutv2/billing-free-tier.html

本書の構成

本書は、Lambda プログラミングが始めての人に、その基本的な使い方から、いくつかの Lambda 関数を組み合わせた少し実用的な使い方までを解説するものです。

最初の 3 つの章で、Lambda の基本を説明します。

第1章 Lambda で実現するサーバーレスシステム

AWS における EC2 インスタンスを利用したシステムと Lambda によるシステム構築の違いについて、その概要と Lambda によるシステム構築のメリットを説明します。

第2章 Lambda 事始め

Lambda は、関数として実装します。この章では、Lambda 関数を、どのような書式で作成する必要があり、どのように登録するのかを説明します。また Lambda 関数の実行には、どのような権限が必要なのかも説明します。そして簡単な Lambda 関数を作り、AWS マネジメントコンソールからテストを実行します。

第3章 Lambda の仕組み

Lambda の実行環境について説明します。そもそも Lambda が実行されるときの OS は、どのようなものなのか、そして、割り当てられるメモリや一時ディスク、ネットワーク通信の可否などについて説明します。また、実行に失敗したときの再処理の仕組み、そして原理的に、ときには 2 回以上呼び出されてしまうこともあるなど、プログラミングするうえで欠かせない知識を説明します。この章の最後では、指定したスケジュール通りに Lambda 関数を定期的に実行してみます。

第4章以降は、実践編です。

第4章 S3 のイベント処理

S3 にファイルが配置されたときに、それを暗号化 ZIP に変換する Lambda 関数を作ります。

この章でポイントになるのは、S3 に関するイベント、そして S3 に置かれたファイルの読み書きの方法です。また、暗号化 ZIP に変換するには、ライブラリが必要です。必要なライブラリをどのようにして準備する必要があるのかについても説明します。

第5章 API Gateway、DynamoDB、SES との連携

Lambda を使って Web アプリケーションを作ります。具体的には、CGI や PHP などで従来実装

されていたサーバーサイドプログラミングを Lambda 関数に置き換える方法を説明します。

この章で作るサンプルは、「ユーザー登録すると、一定時間だけコンテンツをダウンロードできるリンクをメールで送信する」という、少し実用的なサンプルです。

そのための仕組みとして、AWS における NoSQL データベースである「DynamoDB」の使い方、そして一定期間だけ有効な URL を作る「署名付き URL」の扱い方や、メールを送信するための Amazon SES の使い方も説明します。

第6章 SQSとSNSトピックを使った連携

最後の章では、複数の Lambda 関数で呼び出し合うための方法として、SQS や SNS トピックを使う並列処理について解説します。

SQS はキューイングのシステム、SNS は通知を送信する仕組みです。これらをうまく利用することで、Lambda 関数間を疎結合でき、溜まっている処理をひとつずつ処理したり、並列していくつも同時に処理したりすることができます。

この章のサンプルは、メーリングリストを実現する例です。あらかじめデータベースに登録しておいたユーザー宛に、並列してメールを送信します。

これから Lambda を使う人は、まず、第2章と第3章をしっかりと読んで、その仕組みを理解してください。Lambda は、従来のプログラミング環境と違うので、理解せずにプログラムを作ると、思わず罠にはまることがあります。

そして第4章以降は、自分が作りたいシステムに近い構成のものを挿い摘まんで参考にしていくとよいでしょう。

なお、Lambda 関数を作ったり実行したり、本書のサンプルで使っている S3 や DynamoDB、SES や SQS、SNS トピックなどを扱うには、適切な権限を持つユーザー や ロールが必要です。アクションやリソースへのアクセス権限については、Appendix A～D にまとめました。実際にサンプルを動かす際には、Appendix を参照して、ユーザーやロールを作成してください。

目 次

はじめに	3
本書の構成	6
第 1 章 Lambda で実現するサーバーレスシステム	11
1-1 管理の手間を軽減しコスト削減を実現する Lambda	14
1-2 イベントドリブン（駆動）の糊付けプログラミング	19
1-3 まとめ	20
第 2 章 Lambda 事始め	23
2-1 サンプル用 Lambda 関数の仕様	25
2-2 Lambda 関数の構造と設計	26
2-3 Lambda の利用に必要なアクセス権	30
2-4 Lambda 関数の作成	35
2-5 Lambda 関数の実行	43
2-6 まとめ	49
第 3 章 AWS Lambda の仕組み	51
3-1 イベントの発生と Lambda 関数	53

3-2	Lambda コンテナ	53
3-3	Lambda 関数の実行	62
3-4	Lambda 関数を呼び出すイベントソース	68
3-5	定期的に Lambda 関数を実行する例	70
3-6	まとめ	83
第4章 S3 のイベント処理		85
4-1	S3 のイベント事例	88
4-2	S3 バケットの作成	89
4-3	S3 バケットに対するイベント	95
4-4	ライブラリ込みの Lambda 関数の作成	107
4-5	まとめ	123
第5章 API Gateway、DynamoDB、SES との連携		125
5-1	API Gateway のイベント事例	127
5-2	API Gateway と Lambda 関数を組み合わせる	131
5-3	API Gateway から実行される Lambda 関数を作る	133
5-4	DynamoDB の基本	153
5-5	Lambda 関数で DynamoDB にアクセスする	167

5-6	署名付き URL を発行する	179
5-7	メールの送信	187
5-8	クロスオリジンの場合の注意点	199
5-9	まとめ	204
第 6 章 SQS と SNS トピックを使った連携		205
6-1	SQS と SNS トピックのイベント事例	207
6-2	DynamoDB テーブルによるメールアドレス管理	211
6-3	S3 バケットと SQS を構成する	221
6-4	SQS からメッセージを取り出してメールを送信する	242
6-5	バウンスマールを処理する	270
6-6	まとめ	279
Appendix A Lambda 開発者アカウントの作成		281
Appendix B Lambda 実行ロールの作成		291
Appendix C Lambda 開発マシンの準備		297
Appendix D AWS CLI の準備		312
Appendix E バージョニングとエイリアス		315
索引		322

第1章

Lambdaで実現する

サーバーレスシステム

Lambdaの登場で、AWS界隈のシステム開発の現場は大きく変わりました。今までのEC2を使った開発から、Lambdaを使った開発に大きくシフトしているのです。

Lambdaを採用する大きな理由は、マネージドサービスであるため保守・運用の手間がかからないこと。そしてEC2に比べて、堅牢で低コストなシステムを作ることができるからです。本章では、Lambdaはシステム開発に、どのような影響をもたらすのか、そして、利用すると、どのようなメリットが得られるのかなど、Lambda開発を勧める理由を説明します。

- 1-1 管理の手間を軽減しコスト削減を実現するLambda [p.14]
- 1-2 イベントドリブン（駆動）の糊付けプログラミング [p.19]
- 1-3 まとめ [p.20]

本章のポイント

● Lambdaのメリット

Lambdaは、サーバーレスのプログラム実行環境です。従来のEC2インスタンスでプログラムを実行するのに比べて、次のメリットがあります。

(1) 保守・運用に手間がかからない

Lambdaはマネージドサービスです。実行環境がAWSによって用意されるため、OSやフレームワークなどの保守を必要としません。

(2) 高負荷に耐えられる

Lambdaによるプログラムの実行は、必要に応じてスケーリングします。そのため、高負荷に耐えられます。

(3) コストを削減できる

Lambdaは、実行時間に対する課金です。稼働中はずっとコストが生じるEC2インスタンスでの運用に比べて、コストを削減できます。

● Lambdaの制限

Lambdaには、次の制限があります。

(1) 前回の状況を保持しない

Lambdaは、実行が終わったときに環境が破棄される、ステートレスな実行環境のもとで実行されます。そのため、前回の状況（ステート）を保持することはできません。

(2) 最大稼働時間は5分

Lambdaを構成するときは、最大稼働時間の設定が必要です。その時間を越えると、自動的に終了します。設定できる最大の稼働時間は5分です。ずっと実行し続けなければならないプログラムを、Lambdaで構成することはできません。

● イベントドリブン型のプログラミング

Lambdaはイベントの発生によって、実行が開始されます。イベントとなりうるのは、AWSのさまざまなサービスです。たとえば「アラームによって一定時間が経過した」「S3にファイルがアップロードされた」などです。

また「API Gateway」というサービスと組み合わせると、HTMLフォームやAjax通信などに

よるリクエストを Lambda で処理することができます。つまり、Web システムや Web サービスの構築にも活用できます。

● 小さな関数を組み合わせて全体を作る

Lambda プログラミングは、わずかな時間で処理が完了する小さな関数を、たくさん組み合わせてシステム全体を作るのが基本です。

組み合わせるときは、キューイングサービスである SQS や通知サービスである SNS トピックを使うと、Lambda 関数同士を疎結合にでき、拡張性・保守性に優れたシステムを構築できるようになります。

1-1 管理の手間を軽減しコスト削減を実現するLambda

Amazon Web Services（以下 AWS）環境でアプリケーションを構築する場合、プログラムの実行環境として仮想サーバーの EC2 インスタンスを用意し、その環境でプログラムを動かすのが一般的です。しかし、その運用には手間もコストもかかります。

1-1-1 EC2 インスタンス運用上の課題

EC2 インスタンスは、単純な仮想サーバーです。OS やプログラムの実行環境、フレームワーク、ライブラリなどをインストールして、プログラムの実行環境を整える必要があります。こうした EC2 インスタンスを使ったアプリケーションの実行環境の構築は、手間がかかる作業となります。

また、運用や管理も大変です。たとえば、OS やフレームワーク、ライブラリなどに脆弱性が発見されたなら、その脆弱性を塞ぐためのアップデートをするのはユーザーの責任です（図 1-1）。

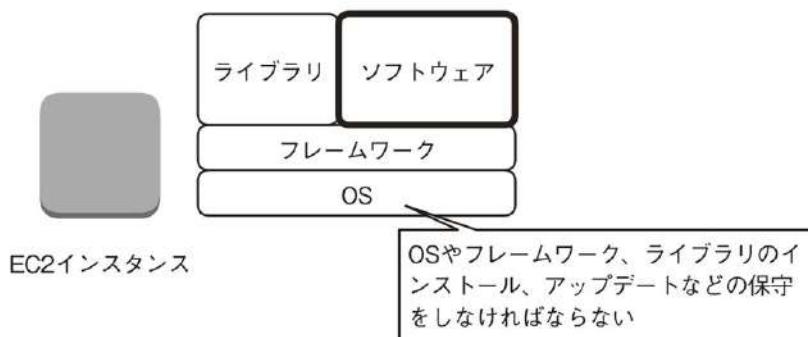


図 1-1 EC2 インスタンスを管理する

また、EC2 には性能の問題もあります。EC2 インスタンスは、起動したときに選んだ「インスタンスタイプ（CPU、メモリ、ディスク IO、ネットワークなどのスペック）」によって性能が決まります。EC2 インスタンスを一時停止すれば、インスタンスタイプを変えることはできますが、稼働中は固定です。そのため急激な負荷増に耐えられないことがあります。負荷を予測して適切なインスタンスタイプで起動しておくとか、負荷分散機能である ELB（Elastic Load Balancing）と、負荷に応じてインスタンス数を自動的に変更できる Auto Scaling という仕組みを組み合わせて、動的に EC2 インスタンスの数を増減するような構成をとるなどの工夫が必要です（図 1-2）。

こうした EC2 インスタンスで運用する場合の手間を軽減することはできないのでしょうか？ その答えのひとつが、本書の主題である AWS Lambda（以下 Lambda）です。

● 1-1 管理の手間を軽減しコスト削減を実現する Lambda

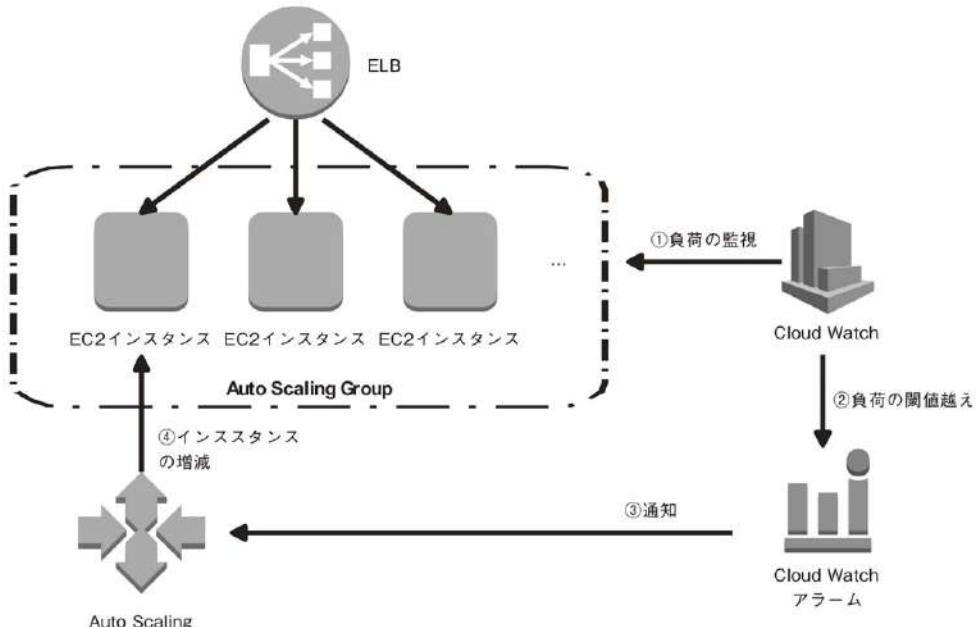


図 1-2 EC2 インスタンスをスケールする

1-1-2 アップロードした関数を実行してくれる Lambda

Lambda は、サーバーレスアーキテクチャと呼ばれ、実行環境として EC2 インスタンスのような仮想サーバーを必要としません。実行環境となるサーバーシステムは、AWS によって実行の都度、用意されます。そのため、サーバーシステムの保守・運用をする必要がありません。また、負荷が高まれば自動的にスケールするため、負荷増のときの対応も考えずに済みます（実際には、スケールできる同時実行数などに制限はあります）。

Lambda でプログラムを実行するために必要なのは、実行したいプログラムを関数として実装し、Lambda にアップロードするだけです。実行の段になると、それらの実行環境（コンテナ）が自動的に作られ、実行してくれます（図 1-3）。

本書の執筆時点^{*1}では、Lambda の実行環境として「JavaScript (Node.js)」「Java」「C#」「Python」の 4 種類の環境が提供されています。

* 1 2017 年 8 月

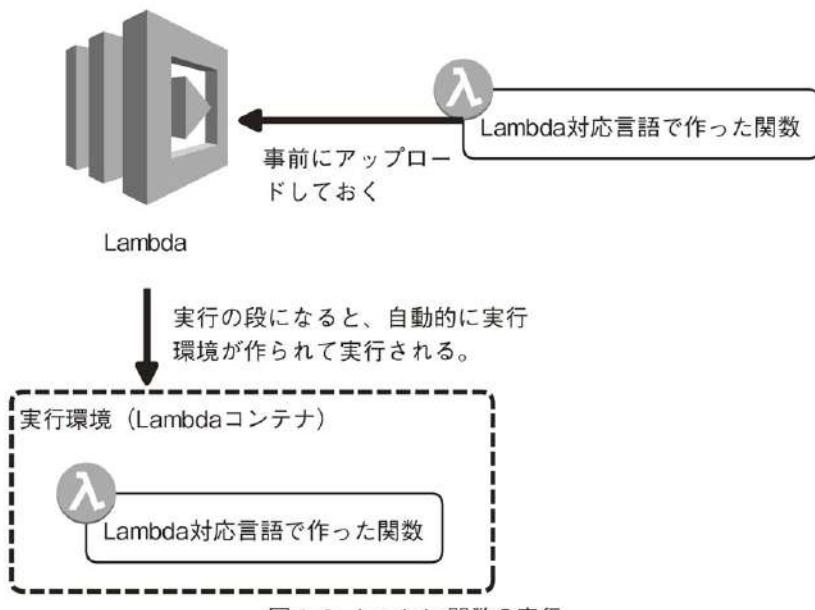


図1-3 Lambda関数の実行

1-1-3 Lambdaの制限

ただし、Lambdaは完全にEC2インスタンスを置き換えるものではなく、いくつかの制限があります。

●ステートレスである

アップロードしたLambda関数は、都度、実行され、実行が終わると破棄されるステートレスな構成です。前回実行したときの状態などを保持することはできません。

●最大稼働時間は5分

Lambda関数には、最大稼働時間があります。設定できる最大時間は5分です。それ以上、時間がかかる処理を実行することはできません。

これらの制限から考えると、Lambdaに向いているのは、継続的に稼働する処理ではなく、「必要に応じて、少しだけ動く処理」です。

継続的に稼働する処理の場合は、従来通り、EC2インスタンスのほうが適しています。

1-1-4 開発者に優しい Lambda

Lambda は、開発しやすいのも特徴です。実行環境が AWS によって用意されるので、開発者は、実行したい機能を関数として作り、それを AWS システムにアップロードするだけで済むからです。

本書を通じて見ていきますが、Lambda を使ってシステムを作る場合、「たくさんの Lambda 関数を作り、それを組み合わせて使う」という運用になります（図 1-4）。ひとつひとつのプログラムの単位が小さく、それぞれが独立するので、テストも容易になります。

また、Lambda 関数同士の結合部分にキュー（Amazon SQS）や通知（Amazon SNS トピック）を使うと、関数を互いに疎結合にすることもできます（その例は、第 6 章で実際に解説していきます）。

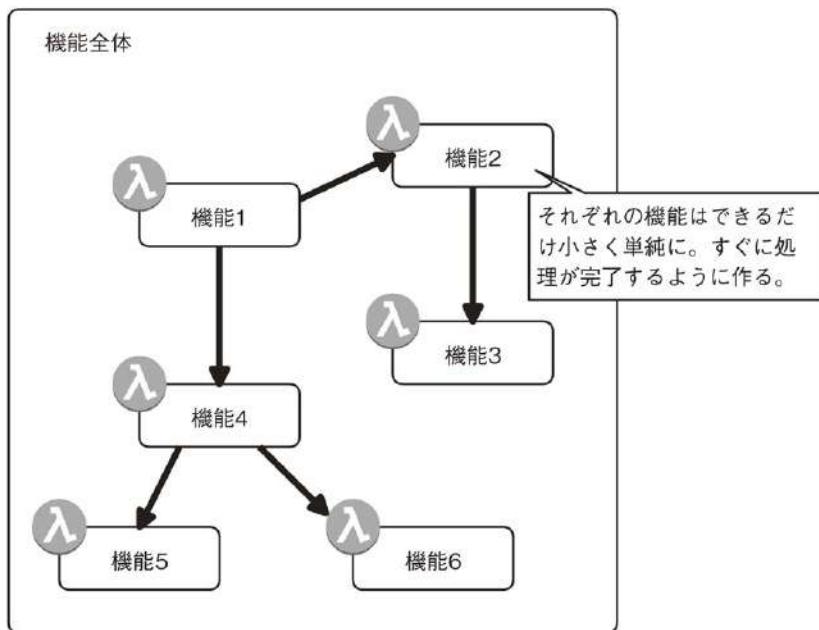


図 1-4 たくさんの Lambda 関数を組み合わせてシステム全体を構築する

1-1-5 Lambda はコスト削減にも貢献する

Lambda は、運用コストも軽減します。Lambda の課金単位は、「実行時間」です。実行されていないときは、まったく費用がかかりません。

たとえば、「毎日 0 時に夜間バッチを 3 分ほど実行したい」としましょう。こうした運用を EC2 インスタンスで構築する場合には、EC2 インスタンスをずっと起動したままにしておき、「0 時に

なったらプログラムを実行する」というように cronなどを設定しておくことになります。つまり、EC2インスタンスは24時間実行し続けます。

それに対してLambdaの場合、Lambda関数をアップロードして、「毎時0時に動かす」というようにスケジューリングの設定をしておくだけです。Lambda関数の実行時間が仮に3分だとすれば、3分間の実行費用しかかかりません。そのため、うまくLambdaで構成すれば、大幅なコスト削減につながります（図1-5）。

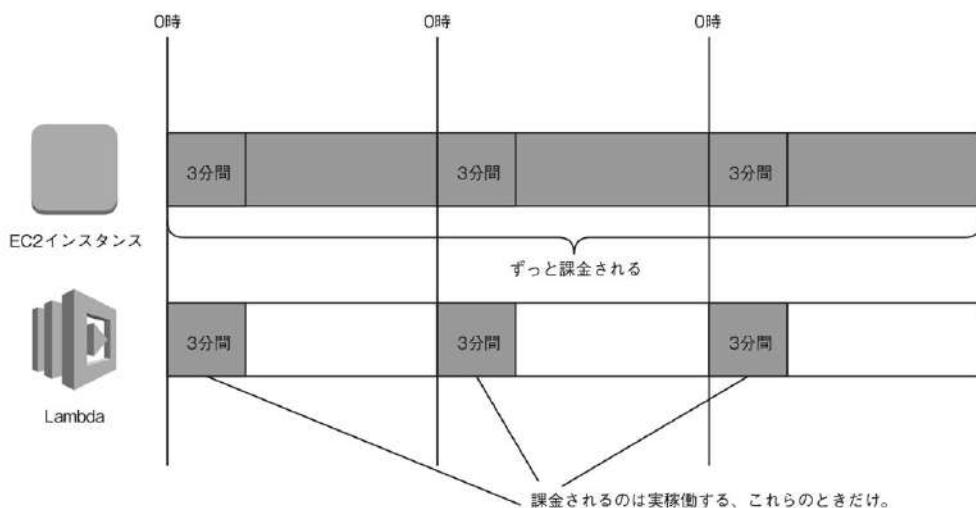


図1-5 Lambdaは実行している間だけ課金される

1-1-6 Lambdaが利用できるリージョン

Lambdaは、主要なほとんどのリージョンで対応しています。どのリージョンで作成しても、ほぼ同じですが、本書では、とくに断りのない限り、「東京リージョン」に作成していきます。

ただし、Lambdaとともに利用するサービスのなかには、現時点では東京リージョンではサポートされていないものもあります。本書の第5章で扱うAmazon SESもそのひとつです。こうした東京リージョンでサポートされていないサービスを利用する場合は、リージョンを変更する必要があります。

1-2 イベントドリブン（駆動）の糊付けプログラミング

Lambda 関数は、「毎時間」や「毎日指定した時間」など、決まったスケジュールで動かすこともできますが、それ以外にも、AWS のさまざまなサービスと連携したタイミングで実行できます。

1-2-1 Lambda 関数を稼働させるイベントソース

Lambda 関数は、AWS 上のさまざまなサービスと連携して実行できます。実行のトリガー（引き金）になるものをイベントソースと呼びます（すべてのイベントソースについては、「3-4 Lambda 関数を呼び出すイベントソース」で説明します）。

たとえば、AWS のストレージサービスである「Amazon S3」は、「ファイルが置かれたとき」などに、Lambda 関数を呼び出すことができます。この機能を使えば、たとえば、「画像ファイルがアップロードされたときにサムネイルを作る」とか「ファイルがアップロードされたときに、パスワード付きの暗号化 ZIP を作る」という処理が実現できます（図 1-6：この例は、本書の第 4 章で、実際に作成します）。

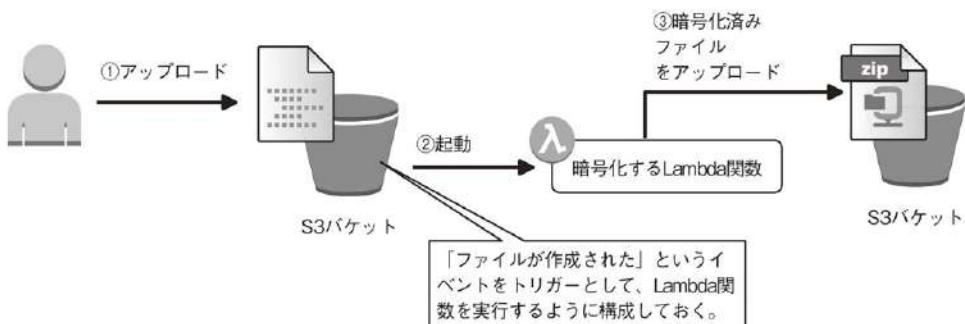


図 1-6 ファイルが置かれたときに Lambda 関数を実行して暗号化 ZIP を作る

また、API Gateway という機能と組み合わせると、Lambda 関数を REST 形式の Web API として呼び出すことができるようになります。たとえば、HTML の入力フォームで POST したり、JavaScript の Ajax で通信したりした結果を Lambda 関数で処理して、データベースに保存するという処理もできます（図 1-7：この例は、本書の第 5 章で、実際に作成します）。

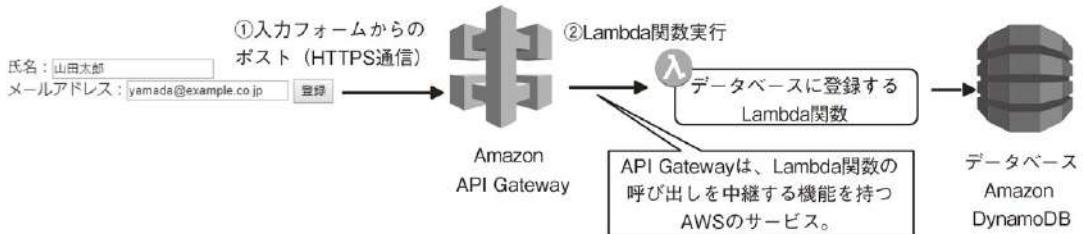


図 1-7 API Gateway と Lambda 関数を組み合わせて REST 形式の Web API を作る

1-2-2 Lambda は AWS のサービスをつなぎ合わせる

図 1-6 や図 1-7 の例を見るとわかるように、Lambda 関数は、「AWS のサービスから呼び出され、何か処理して、別の AWS サービスを実行する」というような処理の構成になることが多いです。

本書を読んでいくとわかりますが、実際、Lambda プログラミングのほとんどは、さまざまな AWS サービスを糊付けするために使われます。ですから、Lambda でプログラミングする場合は、Lambda プログラミングの方法はもちろんですが、どのような AWS サービスがあり、どのようにして利用すればよいのかという知識も不可欠です。たとえば、図 1-6 の構成では、S3へのアクセス方法を知らないければなりませんし、図 1-7 の構成では、データベースの操作方法を知らないければなりません。

1-3 まとめ

本章では、システム開発における Lambda を利用するメリット、そして、Lambda では実現できないことについて説明してきました。

① Lambda はステートレスな実行環境

関数を作りアップロードすれば、イベントが発生したときに実行してくれます。マネージドサービスであるため、OS やフレームワークなどの保守の手間がありません。

② EC2 インスタンスよりも堅牢で低コスト

負荷に応じたスケーリングにも対応するため、EC2 インスタンスで構成するよりも堅牢です。また実行時間に対する課金であるため、コストも低く抑えられます。

③ずっと実行させ続けることはできない

Lambda の最大稼働時間は、5 分です。ずっと実行させ続けることはできないので、もし、そうした機能が必要なら、その部分は従来通り、EC2 インスタンスで構成することになるでしょう。

魅力的な Lambda ですが、利用するのには、最大の難関があります。それは、

Lambda におけるプログラミングを習得しなければならない
という点です。

EC2 インスタンス上で動くプログラムは、オンプレミス環境の延長上であり、さまざまな従来の開発技術が、そのまま使えました。

それに対して Lambda は、AWS が用意した環境の上で実行されるプログラムであるため、さまざまな作法を Lambda に合わせなければなりません。つまり、プログラミング技法は AWS に固有のものとなり、そのためには習得コストがかかります。

しかし、心配することはありません。

Lambda でのプログラミングは、難しくありません。そもそも、Lambda の思想が「小さいプログラムを組み合わせてシステム全体を作る」という考え方なので、プログラムの規模は小さく、100 行に満たない関数を作るだけで、実用十分な処理ができます。

では、Lambda におけるプログラミングとは、いったいどのようなものか、次章からじっくりと見ていきましょう。

λ Column ログをきちんと残そう

Lambda関数はステートレスな実行環境なので、実行したときの状態が残りません。そのため、もし何か実行の際に不具合があったとしても、そのときの状況を知ることができません。

不具合が生じたときの原因を突き止めるために、Lambda関数では、処理したときのログをきちんと残すことを心がけましょう。たとえば、CloudWatch Logsに処理内容や状態を書き出すなどです。

このとき、Lambda関数の処理で何か異常が発生したら、いつも決まった文字列を書き出すように実装しておくとよいでしょう。そうすれば、CloudWatch Logsでその文字列を監視するように構成することで、開発者や管理者に自動的に通知するような仕組みを作れるからです。

第2章

Lambda 事始め



Lambda を使うには、まず、イベントを処理するプログラムについて、いくつかの取り決めを理解しなければなりません。その取り決めとは、「関数の引数や戻り値の書式、エラーの返し方」などです。また、Lambda 関数を作成したり呼び出したりするために必要なアクセス権についても、あらかじめ設定しておく必要があります。本章では、こうした Lambda を利用するうえで必要となる基本的な事柄を、構造の簡単なサンプルプログラムを例に説明します。

- 2-1 サンプル用 Lambda 関数の仕様 [p.25]
- 2-2 Lambda 関数の構造と設計 [p.26]
- 2-3 Lambda の利用に必要なアクセス権 [p.30]
- 2-4 Lambda 関数の作成 [p.35]
- 2-5 Lambda 関数の実行 [p.43]

本章のポイント

● Lambda 関数の構造と設計

Lambda で実行したい処理は、規定の書式に則った関数として記述しなければなりません。呼び出すときに渡された入力値は、関数の引数として取得できます。そして関数からの戻り値が、Lambda 関数の出力となります。

● Lambda の利用に必要なアクセス権

Lambda 関数を作ったり実行したりする操作には、適切なアクセス権が必要です。そこで開発者の IAM ユーザーと、Lambda 関数を実行するときに使う IAM ロールを、あらかじめ作成しておく必要があります。

● Lambda 関数の作成

Lambda 関数を作るときには、関数のコードだけでなく、「割り当てるメモリ」や「最大実行可能な時間」などの実行環境も設定します。

● Lambda 関数の実行

AWS マネジメントコンソールでは、テストイベントというものを作成することで、Lambda 関数を手動で実行できます。

Lambda 関数が実行されると、そのログが CloudWatch Logs に書き込まれます。標準出力に出力したデータも、CloudWatch Logs に書き込まれます。

2-1 サンプル用 Lambda 関数の仕様

本章では、AWS マネジメントコンソールを使って簡単な Lambda 関数を作りながら、Lambda の基本を説明していきます。Lambda 関数の仕様を説明するためのサンプル例なので、動作がシンプルならどのような Lambda 関数でもよいのですが、ここでは、「x」と「y」の引数を渡すと、その値をログに出力し、「 $x \div y$ 」の計算結果を戻り値として返す Lambda 関数を作成します。この関数の仕様を図で示すと、図 2-1 のようになります。

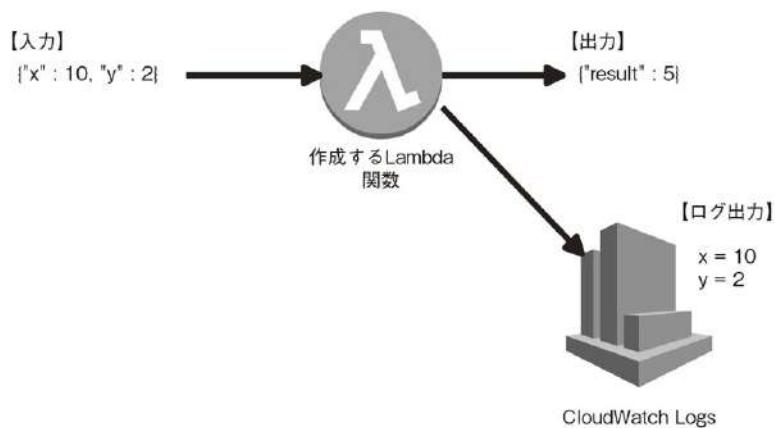


図 2-1 本章で作成する Lambda 関数

引数は、次のように JSON^{*1}形式で渡すものとします。

```
{"x" : 10, "y" : 2}
```

そして、戻り値も同じく、次のように JSON 形式で返すものとします。

```
{"result" : 5}
```

JSON 形式では、

- ・オブジェクト（ここでは引数や戻り値を表す名前と値の組み合わせ）は、「{}」で囲みます。
- ・各名前の後ろには「:」が付き、その後ろに値を書きます。名前／値のペアは、「,」で区切れます。
- ・文字列は「""」で囲みます。

* 1 JSON (Java Script Object Notation) の文法については、以下の URL を参照
<http://www.json.org/json-ja.html>

また、このとき受け取った引数 `x` と `y` の値を、CloudWatch Logs^{*2}にログ出力するようにします。CloudWatch Logs は、ログデータを使用してアプリケーションとシステムをモニタリングする AWS のサービスのひとつです。

2-2 Lambda 関数の構造と設計

先に述べたように、Lambda 関数を作るには、Lambda 関数の構造と決まり事の理解が必要です。具体的に言えば、関数の引数、戻り値の書式、エラーの返し方といった Lambda 関数の基本的な決まり事です。以下、Python3 で作成したプログラムを例に説明します。

2-2-1 Lambda 関数の書式

第1章で説明したように、Lambda 関数は、S3（Simple Storage Service）や SES（Simple Email Services）などで発生するイベントをトリガーとして呼び出されます。イベントが発生すると、Lambda 関数は、適当な実行環境のなかで実行されます（実行環境の詳細は第3章で説明します）。これをコンテキストと言います。

Lambda 関数が実行されるときには、イベントの情報とコンテキストの情報が渡されます。

利用するプログラミング言語によっても異なりますが、Python3 の場合、Lambda 関数の書式は、次のように、「`event`」と「`context`」の2つの引数をとる書式で定義します（`event` と `context` の引数名は任意の名称です）。

```
def 関数名 (event, context):
    …関数の処理…
    return 戻り値
```

Lambda 関数はハンドラとも呼ばれます。そこで慣例的に関数には、「機能名_handler」という名称を付けます。たとえば、次の例では `myfunc_handler` が関数の名称です。

```
def myfunc_handler (event, context):
    …関数の処理…
    return 戻り値
```

Lambda 関数の動作を図示すると、図2-2 のようになります。図2-2において入力は、S3 や SES などのイベントです。このイベントの情報が、第1引数の `event` に渡されます。具体的には、イ

* 2 CloudWatch Logs

http://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/logs/WhatIsCloudWatch_Logs.html

イベントが発生した時刻やファイルが置かれた場所、メールアドレスなど、イベントに付随する情報です。

点線で示したのが実行環境です。実行環境の状態——割り当てられているメモリ容量や実行時間などの環境——が、第2引数の `context` に渡されます。

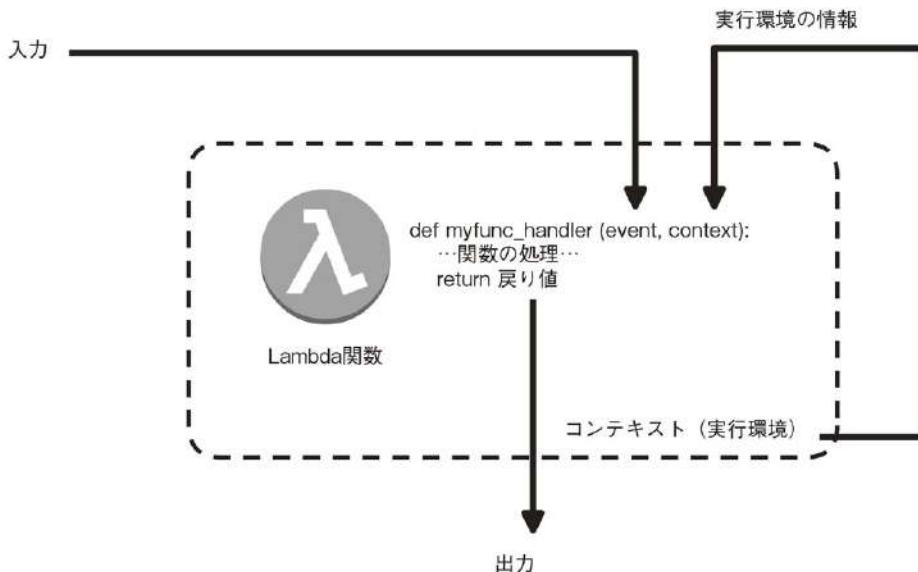


図 2-2 Lambda 関数には 2 つの引数が渡される。

■イベント引数 (event)

イベント引数は、イベントから引き渡される任意の入力値です。入力値がそのまま渡されるので、どのような書式であるのか、どのようなデータが含まれているのかについては、イベント次第です。

イベントのデータは JSON 形式の文字列です。この文字列は、Lambda 関数の引数として渡されるときは、パース済みです。たとえばイベントのデータが、次の JSON 文字列であるとします。

```
{"x" : 10, "y" : 2}
```

このとき、以下の書式の Lambda 関数で受け取る場合、

```
def myfunc_handler(event, context):
```

`event` はもう JSON 文字列ではなく、パースされて Python のオブジェクトに変換されているので、`x` と `y` の値は、それぞれ、

`event["x"]` は「10」
`event["y"]` は「2」

として取得できます。

■コンテキスト引数 (context)

コンテキスト引数は、コンテキスト（実行環境）の情報が含まれる Context オブジェクトです。Lambda 関数名、割り当てられたメモリや実行可能な残時間など、さまざまな環境情報を取得できます（表 2-1 (1)、表 2-1 (2)）。

表 2-1 (1) Context オブジェクト（メソッド）

メソッド	意味
<code>get_remaining_time_in_millis()</code>	実行可能な残時間（ミリ秒）を取得する

表 2-1 (2) Context オブジェクト（プロパティ）

プロパティ	意味
<code>function_name</code>	この Lambda 関数名
<code>function_version</code>	この Lambda 関数のバージョン (Appendix E を参照)
<code>invoked_function_arn</code>	この Lambda 関数の呼び出しに使われた ARN (Amazon Resource Name : AWS リソースを一意に識別する ID)
<code>memory_limit_in_mb</code>	このコンテキストに割り当てられているメモリ容量 (MB)
<code>aws_request_id</code>	リクエストに関連付けられたリクエスト ID
<code>log_group_name</code>	CloudWatch Logs のロググループ名
<code>log_stream_name</code>	CloudWatch Logs のログストリーム名
<code>identity</code>	AWS Mobile SDK を通じて呼び出された場合は、Amazon Cognito 認証プロバイダに関する情報。そうでないときは <code>None</code>
<code>client_context</code>	AWS Mobile SDK を通じて呼び出された場合は、クライアントアプリケーションやデバイス情報。そうでないときは <code>None</code>

■戻り値と出力

Lambda 関数の出力は、Python の `return` 文を使って設定します。戻り値の書式を、どのように使うかは任意であり、イベント次第です。イベントによっては、無視されることもあります。

■標準出力と CloudWatch Logs

Lambda 関数から、`print` 関数などを使って標準出力に出力した内容は、すべて、CloudWatch Logs に書き出されます。

2-2-2 Lambda 関数のサンプル

ここまで説明を踏まえて、本章の冒頭で説明したプログラムは、リスト 2-1 のように記述できます。ここでは Lambda 関数名を「`myfunc01_handler`」としました。

● プログラムの仕様

イベント引数として JSON 形式の「`{"x":x の値, "y":y の値}`」を与えると、その `x` と `y` を `print` で出力して、「`x ÷ y`」の結果を JSON 形式の「`{"result": 答え}`」の形で返す。

リスト 2-1 割り算の結果を返す Lambda 関数

```
import json ← import 文で json モジュールにアクセスする

def myfunc01_handler(event, context): ← 関数の宣言。def は関数を定義する予約語
    x = int(event['x'])
    y = int(event['y'])
    print("x = " + str(x)) ← print 関数で文字列を出力
    print("y = " + str(y))
    retval = {'result': x / y}
    return json.dumps(retval) ← JSON 形式で結果を返す
```

リスト 2-1 では、まず、渡された `event` 引数から `x` と `y` の値を得ます。イベントは JSON 形式の「`{"x": x の値, "y": y の値}`」として渡されることを想定しています。この JSON 文字列はパースされて、この Lambda 関数に渡されるので、次のように、`event['x']`、`event['y']` として

取得できます。ここでは、のちに「 $x \div y$ 」という除算を行うので、int 関数を使って文字列から整数に変換しています。

```
x = int(event['x'])
y = int(event['y'])
```

そして、その値を print 関数を使って、標準出力へと書き出しています。この出力は、CloudWatch Logs に書き込まれます。

```
print("x = " + str(x))
print("y = " + str(y))
```

そして最後に、戻り値を設定します。ここでは、json.dumps メソッドで JSON 文字列に変換しました。

```
retval = {'result' : x / y}
return json.dumps(retval)
```

2-3 Lambda の利用に必要なアクセス権

早速 AWS マネジメントコンソールを使って、リスト 2-1 に示した Lambda 関数を作っていくたいところですが、その前に、必要な IAM (Identity and Access Management) ユーザーと IAM ロールを作成しておく必要があります。

2-3-1 IAM ユーザーと IAM ロール

AWS のアカウント作成時に指定したメールアドレスとパスワードは、ルートアカウント（管理者アカウント）の認証情報です。この認証情報で AWS にサインインした場合は、AWS のすべてのリソースにアクセスできるため、通常時の利用は推奨されていません。通常は、IAM (Identity and Access Management) ユーザーという、AWS リソースに対するユーザー認証と用途ごとに細かいリソースへのアクセス権限を適用したアカウントを利用します。ここでは、Lambda 関数の作成・実行・操作を行うために、IAM ユーザーを用意します。

IAM ロールとは、AWS のリソースへのアクセス権限です。管理者がロールにアクセス権限を付与し、このロールを IAM ユーザーに適用することで、IAM ユーザーは AWS のリソースにアクセスしたり、Lambda 関数を実行したりできるようになります。また、Lambda 関数にロールを適用することで、Lambda 関数から AWS サービスやリソースにアクセスできるようになります。

● 2-3 Lambda の利用に必要なアクセス権



Memo アクセス権

もし、管理者アカウント（Administrators 権限を持つアカウントや root アカウント）を使って操作するのであれば、前もって IAM ロールを作らなくても、Lambda 関数を作るときに、IAM ロールも一緒に作成できます（後述の Column 「その場で IAM ロールを作成する」を参照）。

2-3-2 Lambda の利用に必要なアクセス権

Lambda を利用するときには、次の 3 つのアクセス権が関与します（図 2-3）。

- Lambda 関数の作成権限（開発者用の IAM ユーザー）
- Lambda 関数の実行権限（開発者用の IAM ユーザーやイベント元の権限）
- Lambda 関数からの操作権限

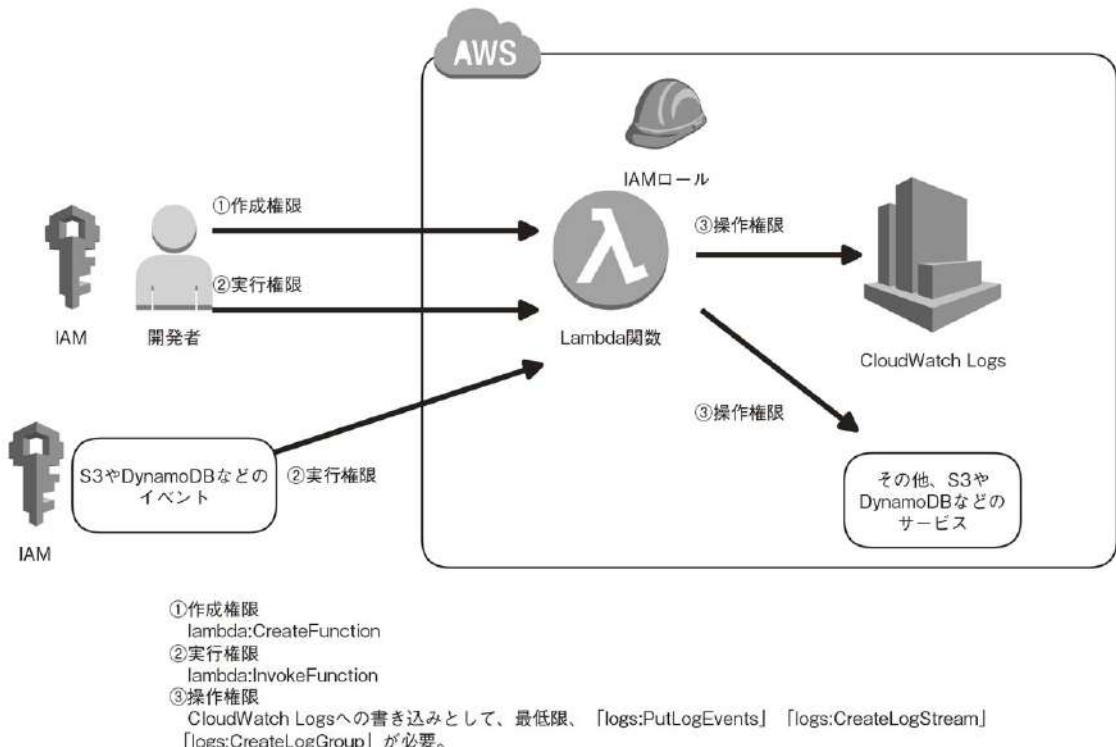


図 2-3 Lambda に関するアクセス権

■ Lambda 関数の作成権限（開発者用の IAM ユーザー）

まず考慮すべきは、開発者の IAM ユーザーの権限です。開発者は、AWS マネジメントコンソールや AWS CLI などを用いて、AWS 上に Lambda 関数を作成する権限が必要です。この権限は、`lambda:CreateFunction` です。この権限がないと Lambda 関数を作れないので、開発者にはこの権限を与えておく必要があります。

■ Lambda 関数の実行権限（開発者用の IAM ユーザーやイベント元の権限）

次に考慮しなければならないのが、Lambda 関数を実行する権限です。この権限は、`lambda:InvokeFunction` です。本章では、開発者が手作業で Lambda 関数を実行します。つまり開発者の IAM ユーザーに、`lambda:InvokeFunction` の権限がないと、実行に失敗します。



Memo `lambda:InvokeFunction` の権限

手動で実行しないなら、開発者の IAM ユーザーには `lambda:InvokeFunction` の権限は必要ありません。しかしテストやデバッグなどの際に、手動で実行する場面は少なくありません。手動で実行しない場合でも、開発者には `lambda:InvokeFunction` の権限をあらかじめ与えておいたほうがよいでしょう。

■ Lambda 関数からの操作権限

Lambda 関数は、作成するときに指定する IAM ロールのもとで実行されます（後述）。Lambda 関数は、ログを CloudWatch Logs へ書き込むので、この IAM ロールには、最低限、CloudWatch Logs へ書き込む権限が必要です。具体的には、`logs:PutLogEvents`、`logs>CreateLogStream`、`logs>CreateLogGroup` の 3 つの権限が必要です。

これらの権限以外に、たとえば、Lambda から S3 を操作したり、DynamoDB を操作したりするなど、AWS のサービスを操作したいときには、Lambda 関数を実行するときに使う IAM ロールに対して、それらのリソースへのアクセス権が必要です。

2-3-3 既存の管理ポリシーを適用する

こうしたアクセス権を、ひとつずつ設定するのは煩雑なため、通常アクセス権の設定には、既存のアクセスポリシーを適用します。既存の管理ポリシーを適用するのは、大雑把な印象がありますが、適切なアクセス権を持った IAM ユーザーや IAM ロールを簡単に構成できます。

● 2-3 Lambda の利用に必要なアクセス権

開発者が手動で実行する Lambda 関数を作る場合は、開発者に割り当てる IAM ユーザーと、Lambda 関数に割り当てる IAM ロールを、それぞれ次のように構成します。

■開発者に割り当てる IAM ユーザー

開発者の IAM ユーザーには、表 2-2 に示すいずれかのポリシーを適用します。いくつかのポリシーがありますが、Lambda に関する全アクセス権が構成されている AWSLambdaFullAccess ポリシーを適用するのが簡単です。

このポリシーを適用すると、Lambda 関数を作ったり、実行したりできるようになります。また、AWSLambdaFullAccess ポリシーは、Lambda 関数と一緒に使うことが多い S3 や DynamoDB などへの書き込み権限も設定されています。ですから、このポリシーが適用された開発者は、Lambda 開発に必要な権限が、ほとんど付与されるので、操作で不自由を感じることはないはずです。

表 2-2 Lambda 開発する開発者向けの管理ポリシー

ポリシー	意味
AWSLambdaReadOnlyAccess	Lambda および DynamoDB、S3 など、Lambda 関連の各種リソースへの読み取り専用アクセス権。このポリシーには lambda:InvokeFunction の権限がないため、Lambda 関数を実行することはできない
AWSLambdaRole	Lambda 関数の実行権限 (lambda:InvokeFunction) のみを設定したもの。任意の Lambda 関数を実行できるが、Lambda 関数を作成したり、一覧を取得したりすることはできない
AWSLambdaFullAccess	Lambda および DynamoDB、S3 など、Lambda 関連の各種リソースへのフルアクセス権。Lambda 関数を実行するほか、Lambda 関数を作成したり、変更したり、削除したりできる

■ Lambda 関数を実行する IAM ロールのためのポリシー

Lambda 関数を実行する IAM ロールには、表 2-3 に示すいずれかのポリシーを適用します。

表 2-3 Lambda 関数を実行する IAM ロールのためポリシー

ポリシー	意味
AWSLambdaBasicExecutionRole	CloudWatch Logs への書き込みアクセス権として 3 つの権限 (「logs:PutLogEvents」「logs>CreateLogStream」「logs>CreateLogGroup」) を付与する

AWSLambdaKinesisExecutionRole	AWSLambdaBasicExecutionRole に加えて、Kinesis Streams アクションへのアクセス権限を付与する
AWSLambdaDynamoDBExecutionRole	AWSLambdaBasicExecutionRole に加えて、DynamoDB アクションへのアクセス権限を付与する
AWSLambdaVPCAccessExecutionRole	AWSLambdaBasicExecutionRole に加えて、ENI (Elastic Network Interface) を管理する EC2 アクションへのアクセス権限を付与する。Lambda 関数から、VPC にアクセスしたいときに用いる

ほとんどの場合、ポリシーとして AWSLambdaBasicExecutionRole を適用すれば十分です。このポリシーは、CloudWatch Logs への書き込みアクセス権(logs:PutLogEvents、logs>CreateLogStream、logs>CreateLogGroup) を与えます。AWSLambdaKinesisExecutionRole と AWSLambdaDynamoDB ExecutionRole は、それぞれ Kinesis や DynamoDB における処理として、データの変化(ストリーミングデータ)を Lambda 側で拾うときに使うものです。そして AWSLambdaVPCAccessExecutionRole は、Lambda 関数から VPC にアクセスするときに必要となる権限を与えます(「3-2-3 VPC へのアクセス」参照)。

2-3-4 IAM ユーザーと IAM ロールを作成する

ここまで説明してきたように、Lambda を利用するには、最低限、開発者の IAM ユーザーと Lambda の実行ロールを、Lambda 関数の作成前に用意しておく必要があります。

■開発者の IAM ユーザー

本書では、ポリシーとして AWSLambdaFullAccess を適用した IAM ユーザーを作り、以降、AWS マネジメントコンソールでの操作はこのユーザーを使うものとします。AWSLambdaFullAccess が適用されたユーザーの作り方については、Appendix A を参照してください。

■Lambda の実行ロール

本書では、AWSLambdaBasicExecutionRole の権限を付与した IAM ロールをあらかじめ作成しておくものとします。ロール名は、何でもかまいませんが、ここでは「role-lambdaexec」という名前にします。次の節に進む前に、Appendix B を参考にして、このロールを作成しておいてください。

λ Column Lambda 関数と Lambda の実行ロールと一緒に作成する

Lambda の実行ロールは、事前に作成せずに、Lambda 関数を作成するときに、一緒に作成することもできます。ただし、その場合は、Lambda 関数を作成する IAM ユーザー（開発者）に、`iam:CreateRole` の権限が必要になります。

2-4 Lambda 関数の作成

それでは実際に、AWS マネジメントコンソールで操作し、Lambda 関数を作成していきます。

「2-3 Lambda の利用に必要なアクセス権」で説明したように、本書では、開発者には「AWSLambdaFullAccess」のポリシーを適用した IAM アカウントを使うことにします。以下の操作は、このポリシーを適用した IAM アカウントで操作してください。

2-4-1 Lambda コンソールのオープン

Lambda 関数は、AWS マネジメントコンソールの Lambda コンソールから操作します。マネジメントコンソールで Lambda を検索し（図 2-4）、Lambda コンソールを開いてください。



図 2-4 Lambda コンソールを開く

まだ Lambda 関数を作成していないときは、スタート画面が表示されます。【関数の作成】ボタンをクリックすると、Lambda 関数の作成を開始できます（図 2-5）。



図 2-5 Lambda 関数を作り始める

2-4-2 設計図の選択

Lambda には、さまざまなシナリオを前提に、多数の設計図 (Blueprint、青写真) があらかじめ用意されています。自分の目的に近い設計図を選ぶと、用途に応じた各種設定と、標準的なサンプルコードがデフォルトで入力されるので、プログラミング作業が楽になります。ただし、本章の場合はすでにリスト 2-1 として、作成する Lambda 関数が決まっているので、[一から作成] をクリックし、設計図を使わず真っさらな状態から始めることにします (図 2-6)。



図 2-6 「一から作成」を選ぶ

2-4-3 トリガーの選択

次に、作成する Lambda 関数がどのような AWS サービスによって、どのような状況のときに呼び出されるのかを、トリガーとして定めます (トリガーの詳細については、第 3 章の事例で解説します)。

第 1 章で説明したように、Lambda 関数は、S3 や SNS、DynamoDB など、さまざまなイベント

● 2-4 Lambda 関数の作成

ソースから呼び出すことができます。本来なら、これらのイベントソースをトリガーとして設定しますが、本章では話を簡単にするため、こうした AWS サービスから呼び出されるように構成するのではなく、まずは「手作業で呼び出す」ことから始めます。そのため、この段階ではトリガーを未設定のまま、【次へ】をクリックしてください（図 2-7）。



図 2-7 トリガーは未設定のままとする

2-4-4 関数の設定

トリガーの設定の次は、Lambda 関数を設定します。この画面では、Lambda 関数の名前やランタイムのほか、コードも入力していきます。いくつか項目があるので順に説明します。

■関数の諸設定

関数の「名前」と「説明」、「ランタイム」を設定します（図 2-8）。



図 2-8 関数の設定

名前

「名前」は、関数に付ける名前です。他の Lambda 関数と重複しない任意の名前を付けられます。ここでは、「HelloLambda」という名前にします。

説明

「説明」は、任意の説明文です。ここでは、「はじめての Lambda 関数」としておきます。

ランタイム

「ランタイム」には、実行環境を選択します。本書では、Python3でプログラミングするので、[Python 3.6]を選択します。

■ Lambda 関数のコード

Lambda 関数のコードを入力します。[一から作成]をクリックして作成した場合、次のひな形コードが自動的に入力されています。

```
def lambda_handler(event, context):
    # TODO implement
    return 'Hello from Lambda'
```

図2-9のこの部分を、リスト2-1に示したプログラムで上書きしてください。なお、プログラムはここに直書きする以外に、ZIP形式でアーカイブしたものもアップロードすることもできます。その詳細は、「4-4 ライブドリ込みの Lambda 関数の作成」で説明します。



図2-9 プログラムのコードを記述する

■環境変数

環境変数は、設定した値を Lambda 関数から参照できるようにする機能です。ここでは利用しないので空欄のままにしておきます（図 2-10）。



図 2-10 環境変数

λ Column 環境変数

環境設定の値を、図 2-10 の画面で設定すると、Lambda 関数から参照できます。この値は、ランタイム環境（プログラミング環境）における環境変数にマッピングされます。たとえば、図 2-10 で環境変数として「foo」を定義したら、Python3 の場合 Lambda 関数からは「os.environ['foo']」として参照できます。

■ Lambda 関数ハンドラおよびロール

この Lambda 関数のハンドラや、実行するときのロールを設定します（図 2-11）。



図 2-11 Lambda ハンドラとロールを設定する

ハンドラ

ハンドラは、Lambda 関数が呼び出されたとき、それを処理する関数のことです。Lambda 関数を Python で記述する場合、ハンドラは次のように「ファイル名」と「関数名」をピリオドでつなげた書式で記述します。

ファイル名. 関数名

リスト 2-1 では Lambda 関数を「myfunc01_handler」として作成したので、関数名は「myfunc01_handler」です。

「ファイル名」は、Lambda コンソール上に直接入力した場合は、「lambda_function」という名前が暗黙的に設定されます。そこで、ハンドラには次に示す値を設定してください。

```
lambda_function.myfunc01_handler
```

間違えて、「myfunc01_handler」のように関数名だけ記述すると動かないで注意してください。

ロール

「ロール」は、前掲の図 2-3 に示した、Lambda 関数を実行する IAM ロールのことです。

ここでは、Appendix B に示す手順で AWSLambdaBasicExecutionRole ポリシーが適用された「role-lambdaexec」というロールを作成済みであることを前提に、【既存のロールを選択】を選択し、【既存のロール】から、それを選択します。（【既存のロールを選択】の選択肢は、Lambda の実行に必要な権限を持つロールが存在しないときは、表示されません。選択肢が表示されていないときは、Appendix B の操作手順に従いロールを作成してください）。

Λ Column その場で IAM ロールを作成する

図 2-11 で「ロール」の選択肢として「テンプレートから新しいロールを作成」や「カスタム ロールを作成」を選択すると、この場で IAM ロールを作成できます。ただし、iam:CreateRole 権限を持たないユーザー（Appendix A で作成した開発者用の IAM ユーザーには、この権限がありません）が操作すると、失敗します。

■タグ

Lambda 関数にタグ付けできます。タグ付けすると、グループ化したり検索したりできますが、ここでは利用しないので空欄にしておきます（図 2-12）。



図 2-12 タグを設定する

■ 詳細設定

Lambda を実行する環境（コンテキスト）について設定します。最も重要な設定は、「メモリ」と「タイムアウト」です（図 2-13）。



図 2-13 メモリとタイムアウトを設定する

メモリ

メモリのデフォルトは「128MB」です。ここではデフォルトのままとしますが、もし、もっとメモリを必要とする場合や、実行時間を短くしたい場合は、スライダーを動かして大きな値を設定してください。ただし、メモリ容量を大きくすると費用がかかるので注意してください。なお、Lambda 関数は実行時間当たりの課金です。

タイムアウト

タイムアウトのデフォルトは「3秒」です。タイムアウト値を大きく設定するだけでは追加の課金はありませんが、実行時間が長くなれば課金は大きくなる点に注意してください（たとえば、タイムアウトを「30秒」にしても、実際は「3秒」で終わるなら 3 秒間の課金しかありません。しかし「30秒」にすると最大 30 秒実行可能なので、Lambda 関数のプログラミングミスなどで無限ループに陥るような場合に、最大 30 秒の課金が発生する可能性があるということです）。

残りの設定は、設定してもしなくても多くの場合、支障がありません。

DLQ リソース

非同期呼び出しが失敗した場合に、SNS や SQS で通知するかどうかを決めます。「3-3-2 プッシュモデルの実行」を参照してください。

VPC

Lambda 関数から VPC に配置されたリソースにアクセスしたい場合に指定します。必要ないのであれば「非 VPC」とします。「3-2-3 VPC へのアクセス」を参照してください。

アクティブラーティス

AWS X-Ray という機能を使って、Lambda 関数の呼び出しを追跡します。

KMS キー

環境変数を暗号化する場合、その暗号化に用いる鍵を選択します。

ここではデフォルトのままにし、「次へ」をクリックして次のページに進んでください（図 2-14）。



図 2-14 その他の設定



Column メモリを多く割り当てる CPU 性能が向上し、実行時間が短くなる

Lambda 関数は、割り当てたメモリの容量が多いほど、実行に費用がかかりますが、メモリの割り当て量に応じて、CPU 性能も高くなる性質があります。そうすると、処理にかかる時間が短くなる可能性があります。

Lambda の料金は、「割り当てたメモリの量に応じたリソース単価」と「実行時間」によって決まります。

Lambda の料金 = 「割り当てたメモリの量に応じたリソース単価」 × 「実行時間」

割り当てるメモリの量を増やすとリソース単価は増えますが、CPU性能が上がる所以実行時間が減り、トータルの料金が下がる可能性があります。

そのため、メモリの量を増やすことが、一概にコスト増につながるとは限りません。処理によっては、メモリの量を増やしたほうが実行時間が短くなり、コスト減になることもあるので、いくつか試してみる価値があります。

最後に確認画面が表示されます。【関数の作成】をクリックすると、関数が作成されます（図 2-15）。



図 2-15 関数の作成

2-5 Lambda 関数の実行

前節で作成した関数を手動で実行し、結果を確認してみましょう。Lambda コンソールでは、「テストイベント」というテスト用のイベントを定義することで、Lambda 関数を手動で実行できます。

2-5-1 テストイベントの定義と手動実行

Lambda コンソールの【関数】の項目を開くと、作成した Lambda 関数が表示されます。ここまでの操作では、「HelloLambda」という Lambda 関数があるはずです。

この Lambda 関数を操作するため、この「HelloLambda」をクリックしてください（図 2-16）。



図 2-16 操作したい Lambda 関数をクリックして開く

■テストイベントを定義する

Lambda コンソール画面には、先ほど作成した Lambda 関数が表示されます。図 2-17 では、Lambda 関数の設定を変更したり、コードを編集したり、実行したりできます。

実行するには、テストイベントを作成します。【アクション】メニューから【テストイベントの設定】を選択してください。



図 2-17 テストイベントの設定を選ぶ

すると、イベントに渡す引数の設定画面が表示されます。今回の Lambda 関数は、「x」と「y」

● 2-5 Lambda 関数の実行

に値を設定すると、「 $x \div y$ 」の値が戻り値として返されるというものです。そこで、次の JSON データを設定してください。ここでは「 x は 10、 y は 2」としました。

```
{  
  "x" : 10,  
  "y" : 2  
}
```

設定したら、[保存してテスト] をクリックしてください（図 2-18）。



図 2-18 テストイベントを設定する

■ 実行結果を確認する

図 2-18 の [保存してテスト] をクリックすると、図 2-19 (1) のように実行結果が成功か失敗が表示されます。



図 2-19 (1) 実行結果の成功の確認

「詳細」をクリックすると実行結果が表示されます。この Lambda 関数では、「{"result" : x を y で割った答え }」を戻り値として返しています。先の例では、x に 10、y に 2 を指定したので、 $10 \div 2$ の結果となる「{"result" : 5.0}」が出力されていることがわかります。

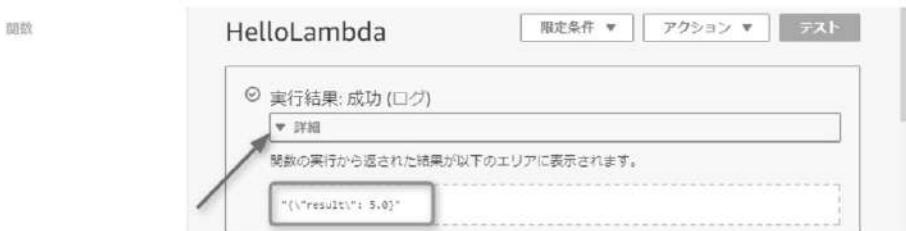


図 2-19 (2) 実行結果の表示する

2-5-2 CloudWatch Logs を確認する

このときの Lambda 関数のログを確認してみましょう。図 2-19 (2) の「(ログ)」の部分をクリックすると、図 2-20 のようにログが表示されます。



図 2-20 CloudWatch Logs を確認する

図 2-20 のログの行をクリックすると、詳細を表示できます。詳細には、CloudWatch のログが表示されます（図 2-21）。



図 2-21 ログの詳細出力

x と y の値は、それぞれ次のようになっていることがわかります。

● 2-5 Lambda 関数の実行

```
x = 10  
y = 2
```

これは、リスト 2-1 における Lambda 関数の以下の `print` 関数の実行によって表示されたものです。

```
print("x = " + str(x))  
print("y = " + str(y))
```

このように、Lambda 関数で `print` 関数などを用いて標準出力に出力した内容は、CloudWatch Logs に出力されることがわかります。

λ Column CloudWatch コンソールからたどる

図 2-19 (1) の画面で、「ログ」をクリックするのではなく、CloudWatch コンソールからたどる場合は、図 2-22 左側メニューの【ログ】を確認してください。

Lambda のログはグループ化されており、【ログ】をクリックすると、「aws/lambda/関数名」というグループがあるはずです。ここをクリックすると、前掲の図 2-20 の画面が表示されます。



図 2-22 CloudWatch からログをたどる

■例外が発生したときの動作

ところで、Lambda 関数の実行時に何かしらのエラーや例外が発生したときは、関数はどのような動きをするのでしょうか？ その動作を確認してみましょう。

リスト 2-1 では、「 $x \div y$ 」の計算をしています。もし、「 y が 0」なら、「0 で割るエラー」が発生するはずです。そこでもう一度、【テストイベントの設定】を選び、以下のように、 y の値を「0」

第2章 Lambda 事始め

に変更して、[保存してテスト] をクリックし、Lambda 関数を実行してみましょう（図 2-23）。

```
{  
    "x" : 10,  
    "y" : 0,  
}
```



図 2-23 y に 0 を指定して実行する

すると実行結果として、「"errorMessage": "division by zero"」というメッセージを含む JSON データが戻り、呼び出しに失敗することがわかります（図 2-24）。



図 2-24 実行に失敗した場合

このように Python で例外が発生した場合には errorMessage、errorType、stackTrace の 3 つの

JSON 値が返されます。なお、この情報は、CloudWatch Logs にも記載されます（図 2-25）。

```

イベントのフィルター
すべて 30秒 5分 1時間 6時間 1日 1週 カスタム ▾

時間 (UTC +00:00) メッセージ
2017-06-08
▶ 14:05:48 START RequestId: 9257bf13-4c53-11e7-ac7b-dd6accbc2438 Version: $LATEST
▶ 14:05:48 x = 10
▶ 14:05:48 y = 2
▶ 14:05:48 END RequestId: 9257bf13-4c53-11e7-ac7b-dd6accbc2438
▶ 14:05:48 REPORT RequestId: 9257bf13-4c53-11e7-ac7b-dd6accbc2438 Duration: 0.28 ms Billed Duration: 100 ms Memory Size: 128 MB
▶ 14:24:55 START RequestId: 3e27e6a2-4c56-11e7-9cbe-97a3c1645e2a Version: $LATEST
▶ 14:24:55 x = 10
▶ 14:24:55 y = 0
▶ 14:24:55 division by zero: ZeroDivisionError Traceback (most recent call last): File "/var/task/lambda_function.py", line 8, in my
division by zero: ZeroDivisionError
Traceback (most recent call last):
File "/var/task/lambda_function.py", line 8, in myFunc01_handler
retval = {'result': x / y}
ZeroDivisionError: division by zero

▶ 14:24:55 END RequestId: 3e27e6a2-4c56-11e7-9cbe-97a3c1645e2a
▶ 14:24:55 REPORT RequestId: 3e27e6a2-4c56-11e7-9cbe-97a3c1645e2a Duration: 15.92 ms Billed Duration: 100 ms Memory Size: 128 MB

```

いまのところ新しいイベントはありません。再試行してください。

■ フィードバック ■ 日本語 © 2009–2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. プライバシーポリシー 利用規約

図 2-25 CloudWatch Logs 上で、実行の失敗を確認したところ

2-6 まとめ

本章では、Lambda 関数の基本を説明しました。Lambda 関数を実行するために、アクセス権の設定といった特殊なところもありますが、結局のところ、Lambda 関数とは、次の書式の関数にすぎません。

```
def ハンドラ(event, context):
    ...処理...
    return 戻り値
```

ここでは、手作業で Lambda 関数を実行しましたが、AWS の各種サービスをトリガーとして実行される場合には、event の引数に、そのときのさまざまな情報が格納されて呼び出されます。

次章では、こうした「イベントから呼び出されるときの仕組み」について、見ていきます。

Λ Column どのプログラミング言語で書くのがよいのか

本書では、プログラミング言語として Python3 系（Python 3.6）を使っています。どのプログラミング言語を使っても、Lambda の仕組み自体は、もちろん同じですが、その書き方は大きく異なります。

たとえば、JavaScript（Node.js）を使う場合、ハンドラは次のように3つの引数をとり、3つ目の引数である「callback」は、戻り値を設定するのに使われます。また Lambda 関数から S3 や DynamoDB などの AWS リソースにアクセスするときに使うライブラリも、まったく違います。

```
exports.handler = (event, context, callback) => {
    // ここに処理を書く
    callback(null, 'Hello');
};
```

単純な文法の違いではなく、関数の書式や戻り値の設定方法、そして利用するライブラリの違いもあるため、他のプログラミング言語で書かれたサンプルプログラムを、自分が使いたいプログラミング言語に翻訳して使うのは、少し困難です。

ですから Lambda プログラミングは、できるだけ実例の多いプログラミング言語を使うのがよいでしょう。

第3章

AWS Lambda の仕組み

Lambda で実行されるプログラムは、EC2 インスタンス上で実行されるプログラムと、少し性質が異なります。動作の仕組みを理解してプログラミングしないと、思うようなパフォーマンスが出なかったり、同時実行時に期待した動作にならなかったりするなどの問題が起こりがちです。本章では、Lambda 関数の実行環境となる Lambda コンテナ環境、イベントソースの種類と特徴、Lambda 関数の具体的な事例などを解説します。

- 3-1 イベントの発生と Lambda 関数 [p.53]
- 3-2 Lambda コンテナ [p.53]
- 3-3 Lambda 関数の実行 [p.62]
- 3-4 Lambda 関数を呼び出すイベントソース [p.68]
- 3-5 定期的に Lambda 関数を実行する例 [p.70]

本章のポイント

● Lambda コンテナ

Lambda 関数は、Lambda コンテナと呼ばれる Linux ベースの環境で実行されます。

コンテナは、実行後に破棄されるステートレスな環境です。2回目以降の実行ではコンテナが再利用されるため、前回実行したときの状態が、一部、残っていることがあります。

● 同期呼び出しと非同期呼び出し

Lambda 関数は、同期呼び出しと非同期呼び出しの2種類の呼び出し方があります。非同期の場合、エラーが発生したときには、2回（初回の1回を含めれば計3回）、リトライ処理されます。それでも失敗したときには、DLQ（Dead Letter Queue）という場所に、通知するように構成できます。

● 2回以上の呼び出しの可能性

非同期で実行される Lambda 関数は、ときには2回以上、実行される可能性があります。

● VPC へのアクセス

デフォルトでは、Lambda 関数はインターネットと接続可能なパブリックなネットワーク上で実行されます。しかし設定を変更することで、プライベートな VPC 上で実行するようにもできます。ただしその場合、パフォーマンスは低下します。

● イベントソース

Lambda 関数の呼び出し元となるのが、イベントソースです。S3 や SES など、さまざまな AWS リソースが、イベントソースとして機能します。

● 一定時間ごとに Lambda 関数を実行する

UNIX 環境の cron のように、一定時間ごとに Lambda 関数を実行したいときは、CloudWatch イベントを使います。

イベントソースに基づいて実行される Lambda 関数を作るときは、設計図（blueprint）と呼ばれる雛形から作ると簡単です。

3-1 イベントの発生と Lambda 関数

イベントの発生や手作業での実行操作などによって Lambda 関数が実行される場合、まず、Lambda コンテナと呼ばれるコンテナが作られ、そのなかで実行されます。このコンテナは、Linux ベースの実行環境です。

イベントには、プッシュモデルとストリームベースの 2 つのタイプがあります。さらに、プッシュモデルは、同期呼び出しと非同期呼び出しに細分されます。どの方式（タイプ）が使われるのかは、イベントの種類によって異なります（図 3-1）。

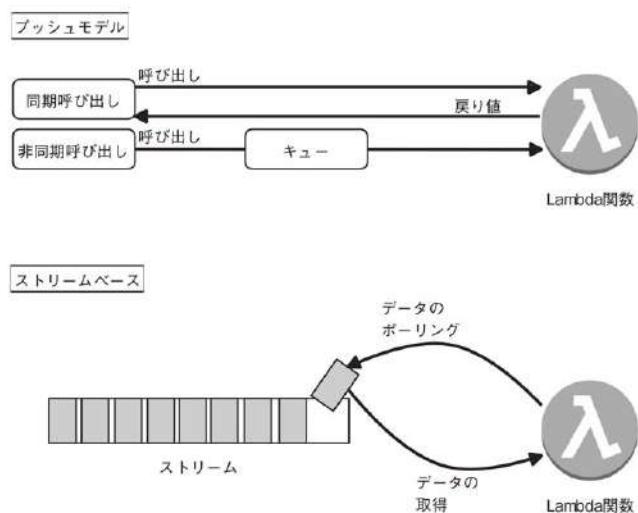


図 3-1 Lambda の仕組み

イベントが呼び出されるときは、そのイベントの種類に応じた引数が渡されます。本章では、「5 分ごとに非同期実行される Lambda 関数」を実際に作ることで、イベントが発生したときに、どのような引数が渡されるのかも見ていきます。

3-2 Lambda コンテナ

第 2 章では、Lambda コンソール上で Lambda 関数を作り、手動で Lambda 関数を呼び出し、実行結果を確認しました。この過程で、メモリ容量やタイムアウトは設定しましたが、仮想マシンなどの設定は行っていません。Lambda 関数は、いったいどのような環境で実行されているのでしょうか？ まずは、その仕組みから見ていきましょう。

3-2-1 Lambda コンテナは Linux 環境

結論から言うと、Lambda 関数は Lambda コンテナと呼ばれるコンテナ環境で実行されます。Lambda コンテナは、Amazon Linux が稼働する Linux コンテナ環境です。本書の執筆時点では、次の AMI をベースとしています。

パブリック Amazon Linux AMI バージョン
(AMI 名: amzn-ami-hvm-2016.03.3.x86_64-gp2)

λ Column Linux コンテナ

Lambda 関数が呼び出されると、Lambda は指定された構成設定に基づいて、Lambda 関数の実行環境となるコンテナを起動します。コンテナとは、単一の OS 環境において、リソースや複数の分離された名前空間によって構築された、個々に独立したアプリケーションの実行環境を意味します。

コンテナ環境は、ハイパーバイザ型仮想化とは異なり、ホスト OS カーネルを各コンテナが共有します。そのため、Windows と Linux といった異なる OS は共存できません。また、同じ AMI OS であっても、カーネルのバージョンが異なれば、コンテナが動かないこともあります。

ハイパーバイザ型仮想化に比べてコンテナ型の仮想化のメリットとしては、オーバーヘッドが小さく、より効率的で性能の高い基盤が構築できるという点も評価されますが、テストや検証環境を迅速に構築できるため、アプリケーションの開発においてもメリットがあります。

Linux 環境なので、「/tmp」などの一時ディスクの領域や、環境変数なども利用できます。また、動的に割り当てられるグローバル IP アドレスを利用して、インターネットと通信も可能です。

■ /tmp 領域

一時的なデータ格納場所として、「/tmp」ディレクトリを利用できます。ただし最大容量は、512MB までです。

■ 定義済みの環境変数

Linux の定義済み環境変数として、表 3-1 に示すものが設定されています。自分の Lambda 関数が配置されているディレクトリを確認したいときなどには、これらの環境変数を参照するとよいでしょう。

表 3-1 定義済みの環境変数

環境変数	意味
LAMBDA_TASK_ROOT	Lambda 関数コードが配置されたパス
AWS_EXECUTION_ENV	実行環境のランタイムの種類。「AWS_Lambda_nodejs」「AWS_Lambda_python3.6」など
LAMBDA_RUNTIME_DIR	Lambda ランタイム関連ライブラリの場所。たとえば、Node.js なら aws-sdk が、Python なら boto3 が、この場所にインストールされている
AWS_REGION	実行されているリージョン名
AWS_DEFAULT_REGION	同上
AWS_LAMBDA_LOG_GROUP_NAME	CloudWatch Logs のグループ名
AWS_LAMBDA_LOG_STREAM_NAME	CloudWatch Logs のストリーム名
AWS_LAMBDA_FUNCTION_NAME	Lambda 関数名
AWS_LAMBDA_FUNCTION_MEMORY	このコンテナで利用できるメモリサイズ
AWS_LAMBDA_FUNCTION_VERSION	Lambda 関数のバージョン
AWS_ACCESS_KEY	実行されている IAM ロールに基づく、アクセスキー ID、シークレットキー ID
AWS_ACCESS_KEY_ID	
AWS_SECRET_KEY	
AWS_SECRET_ACCESS_KEY	
AWS_SESSION_TOKEN	
AWS_SECURITY_TOKEN	
PATH	実行ファイルのパス。「/usr/local/bin」「/usr/bin」「/bin」が含まれる
LANG	ロケール。デフォルトは「en.UTF8」
LD_LIBRARY_PATH	ライブラリのパス。「/lib64」「/usr/lib64」「LAMBDA_TASK_ROOT」「LAMBDA_TASK_ROOT/lib」
NODE_PATH	Node.js 環境のパス
PYTHON_PATH	Python 環境のパス

■ Linux 環境でのプログラミング技法がそのまま使える

実行環境は Linux ベースのシステムなので、Lambda 関数の作り方自体は、EC2 インスタンスの Linux 環境上でプログラミングする場合と、まったく同じです。たとえば、Lambda 関数から外部のライブラリを使いたいときには、Lambda 関数とともにライブラリも一緒に ZIP 形式としてアーカイブしておくと、それをコンテナに展開して実行できます（詳しくは「4-4 ライブラリ込みの Lambda 関数の作成」で説明します）。

ライブラリによっては、バイナリでビルドしなければならないものもありますが、そのような場合は「Amazon Linux AMI」用にあらかじめビルドしたものを同梱するように構成します。

λ Column Amazon Linux AMI で開発・デバッグする

Lambdaの開発・デバッグをする場合、都度、AWSにアップロードして動作確認するのは煩雑であり、開発効率が落ちます。そこで実際の開発では、Amazon Linux AMIをインストールしたEC2上で開発・デバッグを行い、ある程度動いたところで、AWSにアップロードして確認するという手法が、よくとられます。

3-2-2 コンテナの再利用

Lambdaコンテナは、EC2インスタンスと比べると起動時間も短く、必要なメモリやCPUリソースも多くありません。しかし、負荷がまったくないわけではなく、Lambda関数の実行時にLambdaコンテナを作成するには、多少の時間がかかります。そのため、一度作成したLambdaコンテナは、しばらく破棄せずにとておき、次にLambda関数が実行されたときに再利用することで、2回目以降のLambda関数の実行を高速化する仕組みが採用されています（図3-2）。

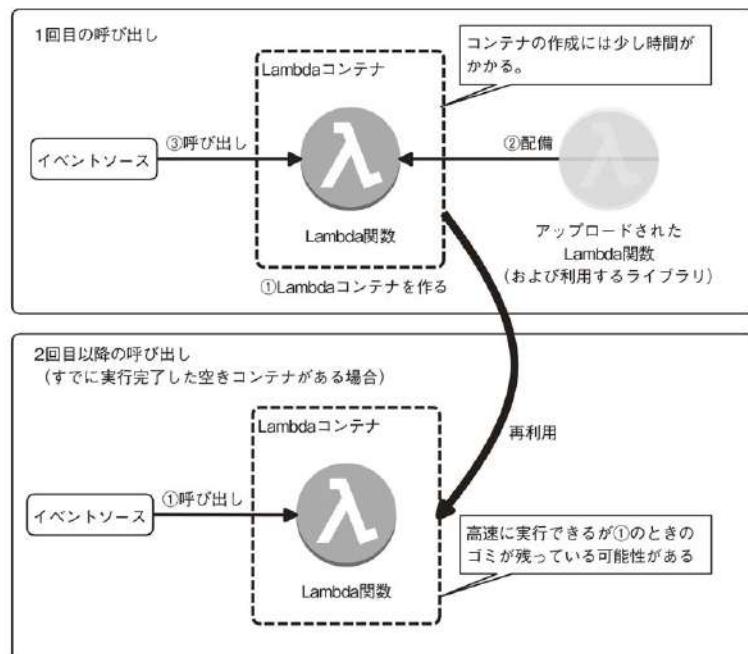


図3-2 Lambdaコンテナを再利用する

コンテナの再利用を行うため、既存の Lambda コンテナの有無によって、処理を開始するまでの時間にはばらつきが生じます。Lambda コンテナがすでにあるときは、それを再利用するので高速です。ないときには Lambda コンテナを作るところから始めるので、処理開始までの時間が長くなります。

■再利用メカニズムの注意

Lambda コンテナは、このような再利用メカニズムがあるため、プログラミングの際にいくつかの注意が必要です。

①/tmp などに前回のデータが残っていることがある

Lambda 関数では、/tmp などの領域に一時データを書き込むことができますが、/tmp には、前回実行したときの状態が残っている可能性があります。

この状態は保証されているわけではなく、「たまたまそうなっているだけ」です。

/tmp の状態が前回と同じであることを想定してはいけませんし、空であることも想定してはいけません。

Lambda 関数はステートレスです。(1) 前回の状態が失われること、(2) 前回実行したときのゴミが残っているかもしれないことを想定して作らないと、思わぬ不具合につながります。

②実行ユーザーが異なることがある

Linux 環境なので、Lambda 関数は Linux システム上のいずれかのユーザーで実行されることになりますが、このユーザーについての規定はありません。そして、いつも同じユーザーで実行されるとは限らず、1 回目に実行するときと、2 回目に実行するときとで、実行ユーザーが異なる可能性があります。

①の動作を考えると、/tmp ディレクトリを使う場合には、前の状態は当てにならないので、「最初に/tmp ディレクトリを削除してから、処理を開始しよう」と思うことでしょう。しかし前回の実行と今回の実行とでユーザーが異なる場合は、前回の実行で /tmp に作成されたファイルは、今回の実行ではパーティションがないために削除できない可能性があります。

この問題を避けるには、/tmp ディレクトリに書き込むファイル名は、都度、ランダムなものを指定するとか、(関数の最初で /tmp ディレクトリを消すのではなく、次に備えるという意味で) 関数の処理の最後で /tmp ディレクトリの中身を消す、という動作にするのがよいでしょう。

3-2-3 VPCへのアクセス

デフォルトでは、Lambda コンテナはインターネットと接続可能なパブリックな場所にあるため、プライベートなネットワークである VPC と接続できません。もし、Lambda コンテナから、VPC に接続された、EC2 インスタンスや RDS インスタンスと通信する必要がある場合には、Lambda 関数の VPC オプションを設定し、VPC 上の任意のサブネットに Lambda コンテナを配置するように構成します。ただし、VPC を利用する場合は、その利便性と引き換えに、次に示すデメリットをあらかじめ理解しておく必要があります。

①起動に時間がかかる

すでに Lambda コンテナが起動している状態であれば、コンテナが再利用されるので起動時の遅延はありませんが、はじめて Lambda コンテナを起動する場合、ネットワークの初期化や DHCP による IP アドレスの割り当てなどのフェーズにより、稼働までに数十秒の時間を必要とします。

②同時実行性が落ちる

VPC と接続した Lambda コンテナには、サブネットから動的に IP アドレスが割り当てられます。そのためサブネットで利用できる IP アドレス数の空きが十分にないと、もうそれ以上 Lambda コンテナを実行できず、実行時エラーが発生します。

このような状況を避けるため、Lambda コンテナを配置するサブネットには、十分な IP アドレスの空きがあることが必要です。

とくに①はパフォーマンスに大きく影響が出ます。もし VPC と通信する必要がないのなら、できるだけ VPC を使わない構成にしてください。

■ Lambda コンテナをサブネットに配置する

Lambda 関数から VPC にアクセスしたい場合は、Lambda コンソールで [関数] をクリックし、目的の関数を選択したのちに、[設定] タブを選び、[詳細設定] オプションの [VPC] で設定します。図 3-3 の設定画面では、「VPC」「サブネット」「セキュリティグループ」を指定します。サブネットは、どこか一箇所を選択すれば動作するには十分ですが、あるサブネットに障害が生じても問題なく動作するよう、複数のサブネットを選択することが推奨されています。図 3-3 のように設定すると、Lambda コンテナに ENI (Elastic Network Interface) と呼ばれるネットワークインターフェイスが構成され、指定したサブネットに配置されて動的に IP アドレスが設定されます。

● 3-2 Lambda コンテナ

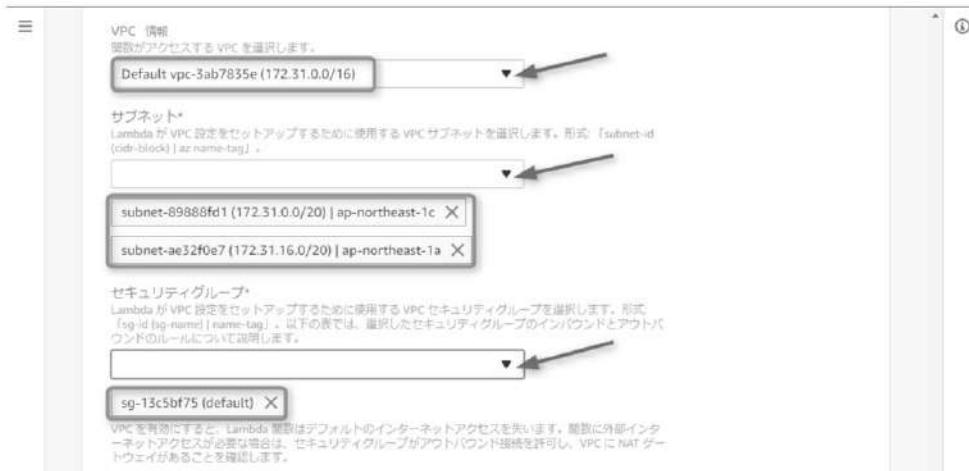


図 3-3 Lambda コンテナをサブネットに配置する

その結果、図 3-4 に示したように同じサブネットに配置した EC2 インスタンスや RDS インスタンスと通信できるようになります。

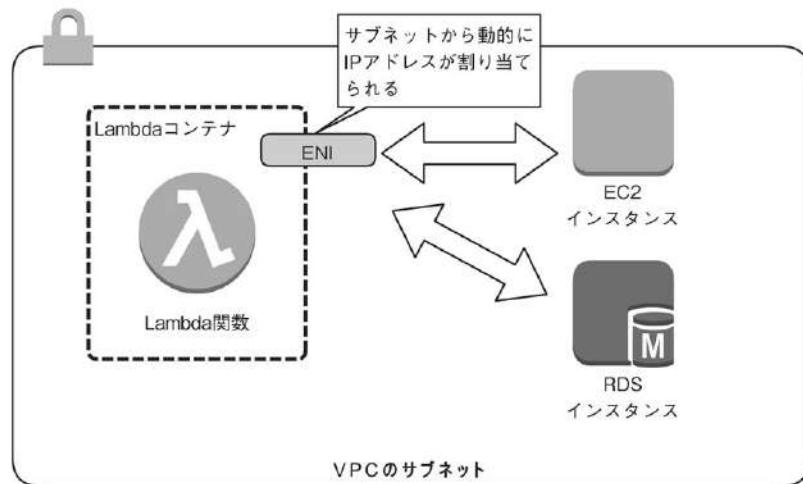


図 3-4 ENI を構成するとサブネットに接続できる

λ Column NATゲートウェイでIPアドレスを固定化する

Lambdaコンテナをサブネットに構成した場合、そのルーティングのふるまいはEC2と同じです。つまり、サブネット間でルーティングが構成されていれば、それらのサブネットにも通信できますし、VPNで社内と通信するように構成されていれば、そうした通信も可能です。そしてNATゲートウェイを構成していれば、さらにインターネットに出て行くこともできます。また、NATゲートウェイにはElastic IPアドレスを設定してIPアドレスを固定化する機能があるため、Lambda関数からの送信元IPアドレスを固定にすることができます（図3-5）。

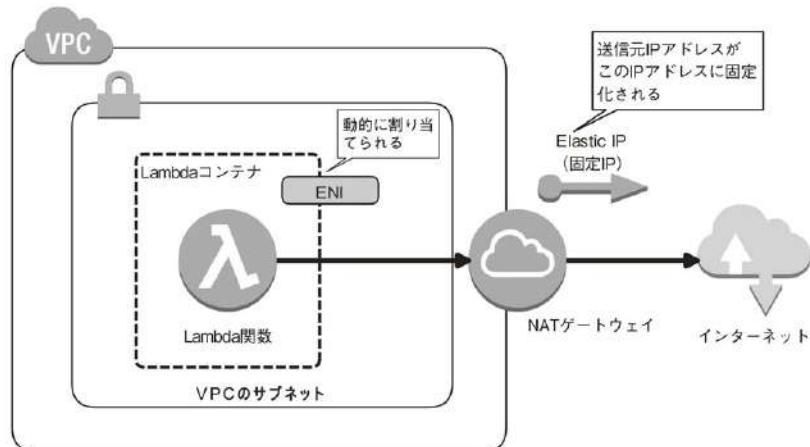


図3-5 Lambda関数からの送信元IPアドレスを固定にする

λ Column 課金にはLambdaコンテナを作成するまでの時間は含まれない

Lambda関数は、時間当たりの課金ですが、Lambdaコンテナを作成する時間（スタートアップ）は課金には含まれません。関数が実行された時間のみ課金されます（図3-6）。

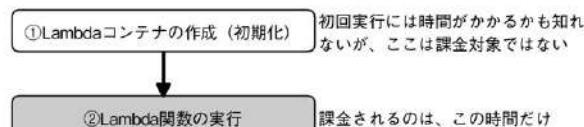


図3-6 Lambdaコンテナの課金

■必要なセキュリティポリシー

Lambda コンテナを VPC に接続する場合、Lambda 関数を実行する IAM ロールに対して、「AWS LambdaVPCAccessExecutionRole ポリシー」のアタッチが必要です。この権限がないと、実行に失敗します。



Column RDS ではなく DynamoDB

Lambda 関数の処理では、多くの場合データベースとして RDS (Relational Database Service) ではなく、DynamoDB が使われます。RDS が使われない大きな理由は、VPC への接続が必要であるためですが、それ以外にも、Lambda 関数は「1回処理したら終了する」という性質のものであるため、RDS とのデータベースコネクションを保持して使い回すことができず、都度、データベースコネクションを張ることになり、パフォーマンス低下の原因となるからです。

もし、RDS への格納がリアルタイムでなくてもよいなら、図 3-7 に示したように、保存すべきデータをいったん DynamoDB に保存しておき、そこからさらに別の Lambda 関数を使って、RDS に徐々に転送するように構成するとよいでしょう。そうすれば直接、RDS に書き込むよりもパフォーマンスが向上します。

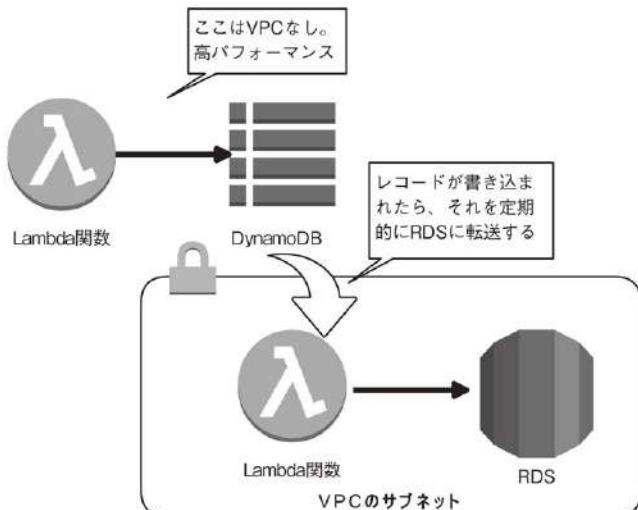


図 3-7 いったん DynamoDB に保存して、徐々に RDS に転送する

3-3 Lambda 関数の実行

Lambda 関数が実行されるタイミングは、さまざまです。第2章で行ったように、Lambda コンソールから手動で呼び出すこともできますし、AWS リソースのさまざまなイベント——「S3 にファイルが書き込まれた」「SES にメールが届いた」など——から呼び出されることもあります。目的とする処理内容によって呼び出し方法が異なり、処理結果にも影響があるため、ここでは、Lambda 関数が実行されるときの仕組みを確認しておきましょう。

3-3-1 プッシュモデルとストリームベース

Lambda 関数の呼び出しは、大きく2つのタイプに分かれます。プッシュモデルとストリームベースです（図3-8）。「Amazon Kinesis データストリーム」と「DynamoDB データストリーム」の2つ以外は、プッシュモデルです。

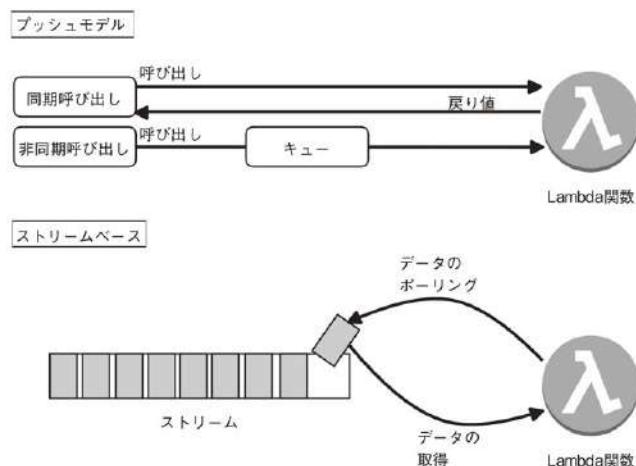


図3-8 プッシュモデルとストリームベース

● プッシュモデル

イベントソースが Lambda 関数を呼び出すモデルです。

● ストリームベース

Lambda 関数側が、イベントソースを流れるデータ（ストリーム）をポーリング（監視）して、データを拾ってくるモデルです。

3-3-2 プッシュモデルの実行

プッシュモデルには、同期呼び出しと非同期呼び出しの2種類があります。第2章で試した「Lambda コンソールでの実行（テスト）」は、同期呼び出しですが、第5章で説明する API Gateway を除き、ほとんどの場合、同期呼び出しへなく非同期呼び出しが使われます。

● 同期呼び出し

Lambda 関数の実行が終わるまで、呼び出し元に戻りません。実行が完了すると、Lambda 関数の戻り値（`return` した値）が、呼び出し元に渡されます。

● 非同期呼び出し

イベントはキューに送信され、呼び出し元にすぐに戻ります。Lambda 関数の戻り値（`return` した値）は、破棄されます。

■ 再試行と DLQ (Dead Letter Queue)

非同期呼び出しへは、イベントがキューに送信されるので、呼び出しから実際の実行までには、少しタイムラグが生じことがあります。なお、キューからの取り出しあはひとつずつではあります。ほとんどの場合、同時に複数取り出されて、Lambda 関数は並列に実行されます（図 3-9）。

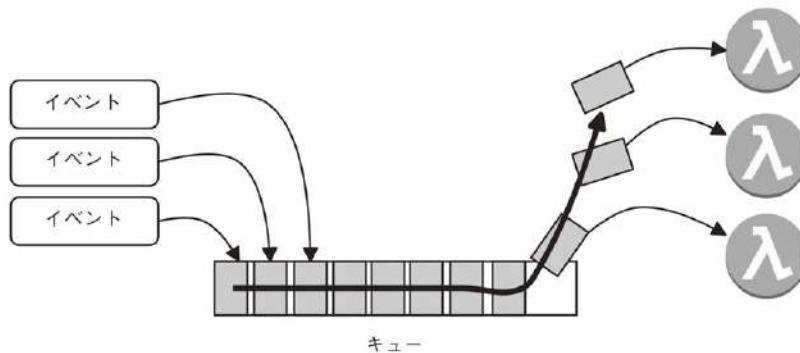


図 3-9 キューイングされて並列実行される

■ 2回の再試行

さて、イベントをキューから取り出して Lambda 関数を実行したとき、その実行がエラーになったときは、どのように処理されるのでしょうか？ エラーが発生したときは、その記録が CloudWatch Logs に記録されます。そしてしばらく待ったのち、最大「2回」試行します。初回1回の実行のち、再試行「2回」なので、最大3回呼び出されるチャンスがあるということです。

■ DLQ を構成する

3回とも呼び出しに失敗したときには、デフォルトでは、そのイベントは消失します。アプリケーションでは、失敗したときに何かエラー処理したり、通知を受けたりしたいこともあるでしょう。そのようなときは、「DLQ (Dead Letter Queue)」という仕組みを使います。DLQ を構成するには、図 3-10 のように DLQ リソースを設定します。



図 3-10 DLQ を構成する設定

DLQ を構成しておけば、実行に3回とも失敗したときには、そのイベントを、あらかじめ作成しておいた「SQS のキュー」に溜めたり、「SNS トピック」として通知したりできます（図 3-11）。

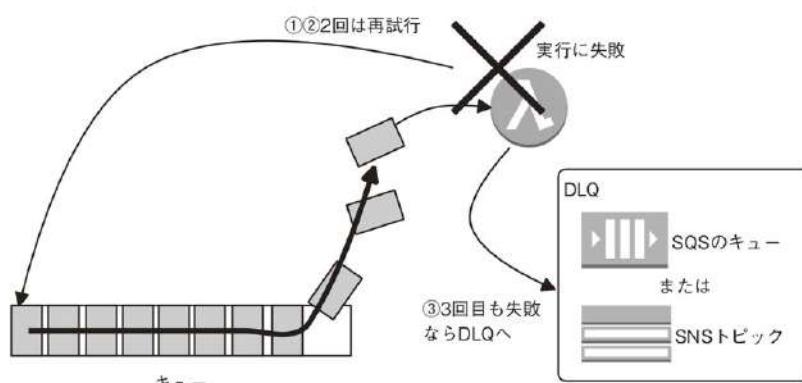


図 3-11 DLQ を設定したときの動作

■再試行しても失敗することが明らかなときは例外を返さない

この仕組み（DLQ）で注意したいのは、非同期呼び出しの場合、Lambda 関数の呼び出しに失敗すると、最大 3 回試行される可能性があるという点です。

ここで言う「失敗」とは、AWS が Lambda 関数の呼び出しになんらかの理由で失敗するときだけではありません。Lambda 関数が例外を発生した場合も「失敗」としてみなされます。そのため、再試行しても失敗することが明らかな場合は（たとえば、Lambda 関数に渡されるパラメータが期待した書式ではないなど）、例外を発生すべきではありません。Lambda 関数では、エラーと例外は区別すべきです。例外を発生してしまうと、二度三度と再実行されてしまいます。

■実行順序は保障されない

非同期呼び出しの場合、キューから呼び出されて処理することになりますが、その実行順序の保障はありません。先に到着したイベントよりも、後から到着したイベントが先に実行される可能性もあります。

■正常なときも 2 回以上、呼び出されることがある

そしてこれはとても大事なことなのですが、非同期呼び出しの Lambda 関数は、正常に実行が完了したとしても、タイミングによっては 2 回以上実行されることがあります。そのため非同期呼び出しでは、何度も実行しても同じ結果となる実装にしておくのが安全です。これを幂等性（べきとうせい）と言います。たとえば、もし 1 回しか実行してはいけない処理なら、前回実行したかどうかのフラグをどこかに保存しておき、フラグが立っていて実行済みであれば、それ以上実行しないように（二度の実行がされないように）、Lambda 関数側で工夫が必要です。

■同時実行の問題

Lambda 関数は、同時に複数実行される可能性があります。たとえば、Amazon S3 に画像ファイルをアップロードすると、そのサムネイル画像が自動生成される Lambda 関数を作るとします。この場合、2 つのファイルを書き込めば Lambda 関数が同時に 2 つ、3 つのファイルを書き込めば同時に 3 つ実行されます。

■同時実行可能な最大数

同時実行可能な最大数は、デフォルトでは 1000 までです（増やさなければならない事情があるなら、AWS に申請することで増やすこともできます）。デフォルト値を超えそうな場合は、SQS を使ってキューイングしながら、少しづつ処理するなどの工夫が必要です。

■同時実行可能な最大数を超えたとき

同時実行可能な最大数を超えたときは実行できません。これは、CloudWatch の「スロットル」として記録されます。このとき、さらにどのような動きとなるのかは、実行方式やイベントのタイプによって異なります。

●同期呼び出しの場合

再試行されず、そのままエラーとして戻ります。

●非同期呼び出しの場合

少しづつ実行を遅らせながら、最大 6 時間の間、再試行されます。

ただしどちらの場合も、イベントの種類によって（Lambda の仕組みではなく）、それぞれのイベントソースに実装された独自処理によって、さらに再試行されることがあります。たとえば、CloudWatch Logs の場合、呼び出しが失敗したときは、最大 5 回、再試行されます。また、Amazon S3 の場合、6 時間経っても成功しなければ、さらに 24 時間、再試行されます。

3-3-3 ストリームベースの実行

プッシュモデルに対してストリームベースとは、流れるデータを Lambda 側で監視する方式です。本書の執筆時点では、Amazon Kinesis と DynamoDB が対応しています。

ストリームベースのイベントは、Lambda 関数を構成すると、イベントソースのデータの流れをポーリングし、その状態の変化によって、Lambda 関数が実行されます。

たとえば、DynamoDB ストリームに対して Lambda 関数を設定すると、ストリームが監視され、レコードの「追加」「更新」「削除」の際に、Lambda 関数が実行されるようになります（図 3-12）。つまり、DynamoDB テーブルに対して、Lambda を使ったトリガー関数を実装できます。

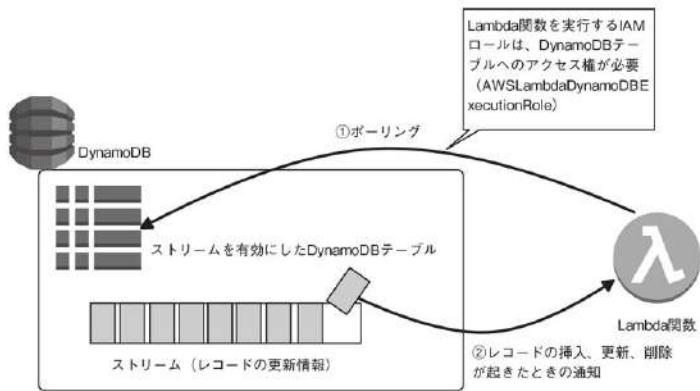


図 3-12 ストリームベースのイベントの動作

■ Lambda 関数の IAM ロールにリソースへのアクセス権が必要

ストリームベースで動作する Lambda 関数において、プッシュモデルと大きく違うのが、アクセス権の設定です。プッシュモデルでは、イベントソースが Lambda 関数を実行するので、Lambda 関数の実行権限がイベントソース側に必要です。それに対して、ストリームモデルでは、Lambda 関数が、DynamoDB や Kinesis をポーリングする流れになるので、Lambda 関数側（Lambda 関数を実行する IAM ロール）に対して、DynamoDB や Kinesis へのアクセス権が必要になります。具体的には、Lambda 関数を実行する IAM ロールに対して、`AWSLambdaDynamoDBExecutionRole` や `AWSLambdaKinesisExecutionRole` のポリシーを適用しておくようにします。

■ エラー時の処理

ストリームベースでは、Lambda 関数の処理が失敗した場合、ストリームの有効期間の間、ずっと、再試行します。DynamoDB の場合、ストリームの有効期間は 24 時間です。エラーが解決されるまでストリーム処理はブロックされ、ストリームから新たなデータを受信しません。そうすることで、ストリームに届いたデータを「順に処理する」という確実性を確保します。

■ 同時実行数のカウント

ストリームベースの Lambda 関数の場合、同時実行単位は、ストリームごとのシャード数です。シャード数とは、「同時に並列実行される数」です。たとえば、ストリームに 100 個の実行中のシャードがある場合、Lambda 関数が 100 個、同時に実行されているとカウントされます。

3-4 Lambda 関数を呼び出すイベントソース

ここまで、Lambda 関数の実行環境である Lambda コンテナや Lambda 関数が実行される仕組みについて見てきました。Lambda 関数の機能概要については、おおむね理解できたと思います。それでは次に、Lambda 関数を実行する引き金となるイベントソースについて見ていきましょう。

3-4-1 イベントソースの種類

Lambda 関数を呼び出す引き金となるものが「イベントソース」です。イベントソースの種類によって「非同期」「同期」「ストリーム」の、どのタイプで動作するのかが異なります（表 3-2）。

表 3-2 イベントソース一覧

サービス	イベントのタイプ	概要
S3	非同期	S3 バケットへの書き込みや削除が発生したとき
DynamoDB	ストリーム	DynamoDB のデータに対して操作したとき。データベースのトリガー関数のように Lambda 関数を利用できる
Kinesis	ストリーム	Kinesis に新しいレコードが到着したとき。トリガー関数のように Lambda 関数を利用できる
SNS	非同期	SNS トピックに対してメッセージが到來したとき
SES	非同期	メールを受信したとき
Cognito	同期	Cognito イベントが発生したとき
CloudFormation	非同期	CloudFormation のスタックを作成、更新、削除したとき
CloudWatch Logs	非同期	CloudWatch Logs にログが記録されたとき
CloudWatch Events	非同期	CloudWatch イベントが発生したとき。EC2 インスタンスの状態や AutoScaling サービスの状態などが変化したときや、スケジュールされた時間になったときなど
CodeCommit	非同期	ブランチまたはタグが作成されたときや既存のブランチに対してプッシュされたとき
Config	非同期	リソースの作成、削除、変更されたとき
Echo	同期	音声アシストサービス Echo から実行される処理関数として利用できる
Lex	同期	Lex ポットでコードをフックできる
API Gateway	同期	HTTPS 経由で Lambda 関数を呼び出したいとき。Web システムの構築に利用できる

3-4-2 間接的な Lambda 関数の実行

表 3-2 は、イベントソースとして利用できる AWS サービスのすべてですが、それ以外のサービスも、S3 や SNS を使って、間接的にイベントソースとして利用できるものがあります。

■ S3 を使用した事例

AWS では、ログや証跡などの結果は、S3 バケットに書き込まれる動作となっています。そこで「S3 に対して何かファイルが書き込まれたときのイベント」を利用すれば、「ログや証跡に関するファイルが更新されたとき」に、Lambda 関数を実行できます。たとえば、CloudTrail という証跡サービスは、表 3-2 に含まれていないので、Lambda のイベントソースにはなりません。しかし証跡を記録する S3 バケットに Lambda 関数を設定しておけば、証跡状態の変化を Lambda 関数で処理できます（図 3-13）。

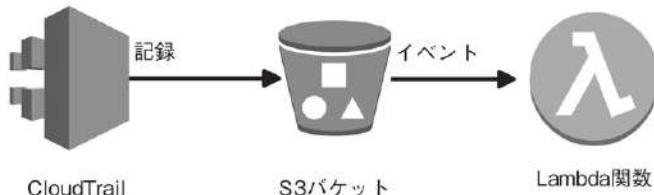


図 3-13 S3 バケットをイベントソースにすることで間接的に通知を受ける

■ SNS を使用した事例

S3 と同じく、通知を受けるのによく使われるのが SNS です。AWS では、各種エラーや状態の変化が生じたときに、「SNS トピック」として、それを通知するサービスが、いくつかあります。たとえば、CloudWatch でリソースを監視しているとき、もしアラートが発生したときは、SNS トピックスに書き込まれます。このとき、SNS トピックスをイベントソースとして利用すれば、アラートが発生したときの処理を Lambda 関数で処理できます（図 3-14）。

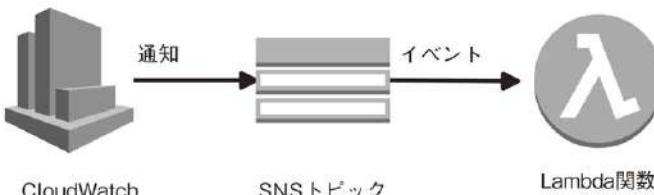


図 3-14 SNS トピックをイベントソースにすることで間接的に通知を受ける

3-4-3 同期イベントと非同期イベント、ストリームベース

「3-3 Lambda 関数の実行」で説明したように、Lambda 関数の呼び出しには、同期と非同期、ストリームベースがあります。

表 3-2 に示したように、ほとんどのイベントソースは非同期です。つまり、処理はキューイングして実行されます。そして失敗したときには、2 回試行（すなわち最大 3 回実行）され、それでも失敗すれば実行が取りやめられます。

2 回の試行をもってしても実行に失敗したときの再処理は、イベントソースによって異なります。

3-4-4 イベント引数の内容

第2章では、Lambda 関数の書式は、次のようにあると説明しました。

```
def 関数名(event, context):
    …関数の処理…
    return 戻り値
```

この第1引数の `event` には、イベントソースによって設定された、さまざまな状態が格納されて Lambda 関数が実行されます。では実際に、`event` にはどのような値が、どのような構造で格納されているのでしょうか？ 残念ながら、イベントソースに依存する任意のデータであるため、「このような形式である」と言うことはできません。たとえば、S3 なら「書き込まれたバケットの場所」や「ファイル名」などが渡されますし、SES なら「受信したメールアドレス」や「メールタイトル」、「メール本文」などが渡されますが、そのデータ構造はさまざまです。ですから、利用したいイベントソースのリファレンスを参照して、確認しなければなりません。

イベントソースに関するリファレンスは、Lambda のドキュメントではなく、イベントソース側のドキュメントに記載されています。たとえば、「S3 上でイベントが発生したとき、どのようなイベント引数の形式になるのか」を調べたいときは、(Lambda のドキュメントではなく)、S3 のドキュメントを探すと見つかります^{*1}。

3-5 定期的に Lambda 関数を実行する例

本章の最後の例として、非同期実行の実例を見てみましょう。ここでは、5 分ごとに実行する Lambda 関数を考えます。いわゆる EC2 インスタンスで言うところの「cron による定期実行」の機能を Lambda で構成してみます。

* 1 "Amazon Simple Storage Services Developer Guide" API Version 2006-03-01

● 3-5 定期的に Lambda 関数を実行する例

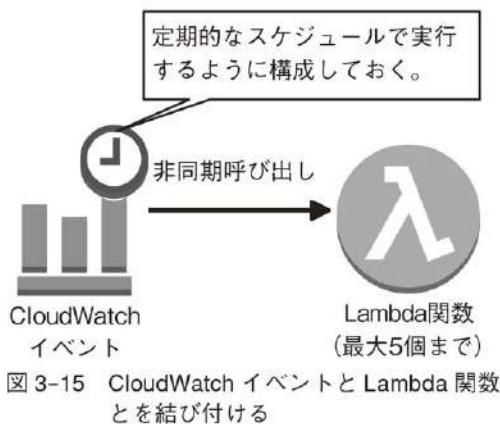
話を簡単にするため、ここで作成する Lambda 関数は、実行されたときに以下の処理を行うものとします。

「Hello」という文字列(①)とイベント引数として渡された値(②)を、CloudWatch Logs にそれぞれ書き込む

3-5-1 Lambda 関数を一定時間ごとに実行する

一定時間ごとに Lambda 関数を実行したいときは、イベントソースとして、「CloudWatch イベント」を使います。

CloudWatch イベントには、「スケジュール」と呼ばれるルールタイプが用意されていて、「毎△分」「毎○時」「毎月 XX 日」など UNIX システムの cron と同じ形式や、「何分ごと」「何時間ごと」「何日ごと」など、周期的なスケジュールを構成できます(図 3-15)。ひとつの CloudWatch イベントには、最大 5 個の Lambda 関数を結び付けられます。



3-5-2 lambda-canary 設計図から作成する

CloudWatch イベントから実行される Lambda 関数を作るには、「lambda-canary」という設計図から構成するのが簡単です。本書では Python で作るので、「lambda-canary-python3」を選択してください(図 3-16)。



図 3-16 lambda-canary-python3 を選択する

■スケジュールを作成する

「lambda-canary-python3」を選択すると、[トリガーの設定]の画面が表示されます。lambda-canary-python3 (lambda-canary でも同様) を選択したときには、イベントソースとして「CloudWatch Event」が設定されています。

そこで CloudWatch Event の、どのルールが発動したときに、Lambda 関数を呼び出すかを決めます。あらかじめ作成しておいたルールを指定することができますが、ここでは、その場で作成しましょう。「ルール」で [新規ルールの作成] を選択してください（図 3-17）。



図 3-17 新規ルールの作成を選択する

すると、ルール名やルールタイプ、スケジュール方式などを選択する画面になるので、次のように入力します（図 3-18）。

● 3-5 定期的に Lambda 関数を実行する例



図 3-18 新規ルールの作成

ルール名

ルールに付ける名前を入力します。どのような名前でもかまいません。ここでは、「MyScheduleRule」と名付けることにします。

ルールの説明

任意のルールの説明です。未入力でもかまいません。

ルールタイプ

一定時間ごとに実行するイベントを設定するため、【スケジュール式】を選択します。ちなみに、【イベントパターン】を選択すると、AWS のイベント（たとえば、CloudTrail で監視された API の呼び出しやリソースの状態変更など）についてのルールを設定することもできます。

スケジュール式

どのような時間単位でイベントを発生するのかを設定します。

cron 式 (UNIX の cron コマンドの書式) または rate 式で指定できます。時刻で指定したいときは前者を、周期として指定したいときは後者を指定してください（「Column cron 式と rate 式」参照）。なお時刻は、UTC（世界標準時）で指定するので、注意してください。

ここでは、「0分」「5分」「10分」…と、5分刻みで実行したいので、「cron(0/5 * * * ? *)」とい

う cron 式を入力してください。なお、5分周期で実行するという意味では、rate式として「rate(5 minutes)」と入力してもかまいません。

トリガーの有効化

このトリガーを、有効にするかどうかを設定します。ほとんどの場合、動作テストしてからトリガーを有効にしたほうがよいので、ここではチェックを付けないでおきます。

Column cron式とrate式

cron式は、左から順に、下記のフィールドを空白で区切った式です。

フィールド	値	指定できるワイルドカード
分	0-59	, - */
時	0-23	, - */
日	1-31。日を指定したときは曜日は「?」を指定 する必要がある	, - * ? / L W
月	1-12 または JAN-DEC	, - */
曜日	1-7 または SUN-SAT。曜日を指定したとき は日は「?」を指定する必要がある	, - * ? / L
年	1970-2199	, - */

ワイルドカードの意味は、次の通りです。

ワイルドカード	意味
,	値を列挙するときに使う。たとえば分に対して「0,15,30,45」を指定して「0分」「15分」「30分」「45分」に実行したいとき
-	指定した範囲を示すときに使う。たとえば月に対して「1-3」で「1日」「2日」「3日」に実行したいとき
*	すべての範囲を示したいとき
/	増分を示したいとき。たとえば分に対して「0/15」で15分ごとに実行したいとき
?	任意の値を示したいとき
W	平日

rate式は、指定した周期での実行を指定する式です。「minute/minutes」「hour/hours」「day/days」を指定でき、たとえば、「15 minutes」は15分ごとに実行するという意味です。

トリガーの設定が終わったら、[次へ] をクリックします。



Memo rate 式での指定

rate 式で指定する場合、起点は、ここで説明する一連の操作によって、このイベントが作成された（保存された）日時になります。たとえば、仮に 12 時 3 分にこのイベントが作成された（ここでの一連の操作が完了した）とすると、rate (5 minutes) は、「12 時 8 分」「12 時 12 分」…というように実行されます。

3-5-3 Lambda 関数を作る

次の画面（図 3-19）では、Lambda 関数を作成します。この操作は、第 2 章で説明した操作とほとんど同じです。

■ 関数の設定

関数の「名前」と「説明」、「ランタイム」を設定します。

Lambda 関数名を入力します。任意の名前でかまいません。ここでは「MyScheduleFunction」という名前にします（図 3-19）。

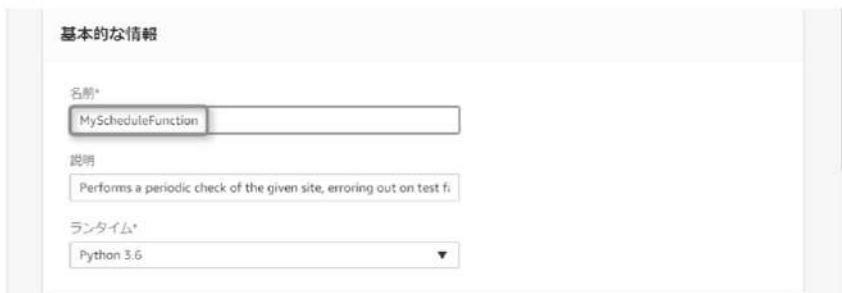


図 3-19 関数の設定

■ Lambda 関数のコード

Lambda 関数のコードを入力します。「lambda-canary-python3」から作成したひな形では、「環境変数で指定した Web ページにアクセスし、その Web ページに特定の文字列が含まれているかどうかを確認する」というサンプルプログラムが、あらかじめ入力された状態になっています。

ここで作成する関数では、そのような処理ではなく、「Hello という文字列」と「渡されたイベント引数」を CloudWatch Logs に書き込みたいだけなので、リスト 3-1 のように修正します。

イベント引数は、オブジェクトなので、そのまま print できません。そこで JSON 形式で出力するため、次のように指定しました。

```
print(json.dumps(event))
```

リスト 3-1 「Hello」とイベント引数を CloudWatch Logs に書き込むプログラム

```
import json

def lambda_handler(event, context):
    print('Hello')
    print(json.dumps(event))
```



図 3-20 Lambda 関数のプログラム

■環境変数

デフォルトでは、サンプルプログラムに合わせた環境変数が設定されています。必要ないので削除] をクリックして削除してください（削除しなくても支障はありません）（図 3-21）。



図 3-21 環境変数の設定を削除する

■関数ハンドラとロール

関数ハンドラとロールを設定します。

● 3-5 定期的に Lambda 関数を実行する例

関数ハンドラ

関数ハンドラは、デフォルトの「lambda_function.lambda_handler」のままでします。

ロール

ロールは「既存のロール」を選びます。

既存のロール

本書の Appendix A で作成している「role-lambdaexec」を選択してください（図 3-22）。



図 3-22 関数ハンドラとロールを設定する

残る「タグ」と「詳細設定」は、とくに変更せず、デフォルトのままでします。[次へ] をクリックしてください。すると確認画面が表示されるので、[関数の作成] をクリックして Lambda 関数を作成してください（図 3-23）。



図 3-23 関数の作成の確認

3-5-4 動作テスト

では、関数の動作テストしてみましょう。

■テストイベントの設定

[アクション] メニューから [テストイベントの設定] をクリックしてください（図 3-24）。



図 3-24 テストイベントを設定する

するとデフォルトで、スケジュールイベント用の設定が開かれるので、そのまま [保存してテスト] をクリックしてください（図 3-25）。



図 3-25 テストイベントを構成する

● 3-5 定期的に Lambda 関数を実行する例

■実行結果とログの確認

[保存してテスト] をクリックすると、実行結果が表示されます。「詳細」をクリックし、[ここをクリックし] をクリックして確認します。CloudWatch Logs のログの行をクリックすると、「Hello」というメッセージと、event 引数に渡された値が出力されていることがわかります。ここでは、テストとして実行しているので、event 引数に渡される値は、図 3-25 のテストイベントで設定した値と同一です（図 3-26、図 3-27）。



図 3-26 実行結果



図 3-27 ログを確認する

3-5-5 スケジュールを有効化する

動作テストが済んだら、CloudWatch イベントを有効にしましょう。[トリガー] タブで [有効化] を選択すると、有効になります（図 3-28、図 3-29）。



図3-28 有効化する



図3-29 有効化の確認

■引数に渡される値を確認する

有効化したら、5分ほど待ちましょう。cron式として設定した「cron(0/5 * * * ? *)」のタイミング——「00分」「05分」「10分」…というタイミング——で実行されるはずです。そしてCloudWatch Logsを確認してみてください。するとログに書き込まれていることがわかるはずです（図3-30）。

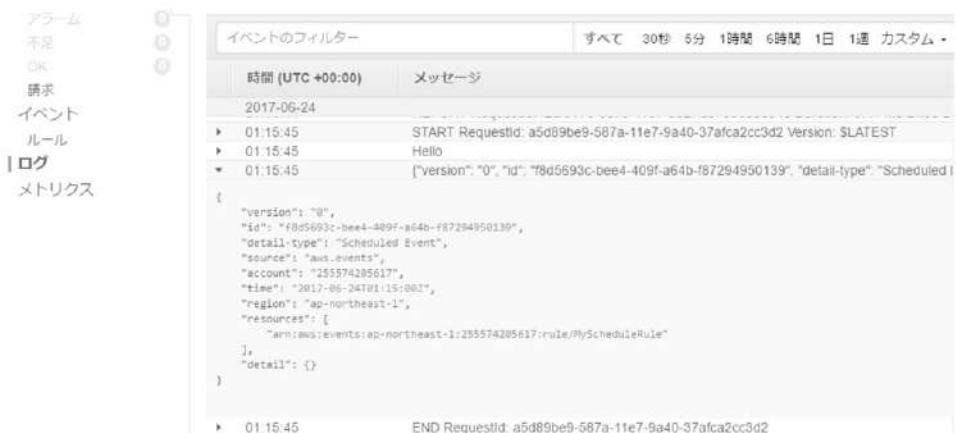


図3-30 しばらくしてから CloudWatch Logs を確認する

● 3-5 定期的に Lambda 関数を実行する例

このログは、CloudWatch イベントから実行されたときに記録されたものなので、出力されている引数は、CloudWatch イベントから渡されたものです。

ログに記録された内容から、次の書式であることがわかります。

```
{  
    "version": "0",  
    "id": "f8d5693c-bee4-409f-a64b-f87294950139",  
    "detail-type": "Scheduled Event",  
    "source": "aws.events",  
    "account": "XXXXXXXXXXXXXX",  
    "time": "2017-06-24T01:15:00Z",  
    "region": "ap-northeast-1",  
    "resources": [  
        "arn:aws:events:ap-northeast-1:XXXXXXXXXXXXX:rule/MyScheduleRule"  
    ],  
    "detail": {}  
}
```

CloudWatch イベントで渡される引数の書式は、以下の CloudWatch のドキュメントに記載されています。

【CloudWatch イベントに渡される書式の解説】

<http://docs.aws.amazon.com/AmazonCloudWatch/latest/events/EventTypes.html>

この資料によると、スケジュールされたイベントの場合の書式は、次の通りです。実際の出力結果と合致していることがわかります。

```
{  
    "version": バージョン番号  
    "id": イベントのID,  
    "detail-type": "Scheduled Event",  
    "source": "aws.events",  
    "account": ユーザーアカウントID,  
    "time": 発生日時,  
    "region": 発生対象のリージョン,  
    "resources": [ 引き金となったルール],  
    "detail": {}  
}
```

■イベント引数を任意に指定する

CloudWatch イベントでは、Lambda 関数に引き渡す引数は、任意に変更できます。変更したいときは、(Lambda コンソールではなく) CloudWatch コンソール画面から操作します。たとえば、イベントの引数として「定数 (JSON テキスト)」を指定すると、任意の JSON テキストを `event` 引数にできます。たとえば、図 3-31 のように、「`{"action": "0001", "eventname": "foobar"}`」を指定するとします (詳細は、章末の Column を参照)。



図 3-31 イベントの引数をカスタマイズする

この状態でしばらく待って、CloudWatch Logs の出力を再確認すると、指定した通りの値が、`event` 引数に設定されていることがわかります (図 3-32)。



図 3-32 `event` 引数に渡される値が変わった

このように `event` 引数は、カスタマイズすることで、いくらでも変更できます。Lambda 関数の呼び出しにおいては、引数の書式や値は本当に自由であり、「このような書式でなければならぬ」というルールは存在しないのです。

■スケジュールされたイベントを無効にする

動作テストが終わったら、スケジュールされたイベントを無効にしておきましょう。そうしないと、5分単位でずっと Lambda 関数が実行されてしまい、無駄な課金が発生してしまうからです。

スケジュールされたイベントを無効化するには、【無効化】を選択します（図 3-33）。



図 3-33 スケジュールされたイベントを無効化する

3-6 まとめ

本章では、Lambda 関数を実行するための仕組みやイベントの種類について説明してきました。つまるところ、Lambda で肝要なポイントは次の 3 点です。

- ①Linux 環境で実行される。
- ②その環境には、前回に実行された状態が残っているかもしれない。
- ③VPC に接続するには、追加の設定が必要である（そしてパフォーマンスが低下する）。

そして、非同期イベントの場合には、次のことも理解しておくとよいでしょう。

- ④エラーがあると最大 3 回まで再試行される。
- ⑤正常であっても 2 回以上実行されてしまうことがある。

これらをきちんと理解しておけば、Lambda 関数は難しくありません。あとは、「さまざまなイベントを、どのように組み合わせて、システムを作っていくのか」を考えるだけです。次章からは、Lambda 関数を使って、少し実用的なサンプルを作ていきます。

Column CloudWatch コンソールから設定を確認する

あとからスケジュールの間隔を変更したいときや、イベント引数を変更したいとき、あるいはひとつのスケジュールに、複数のLambda 関数（最大5個まで）を設定したいときなどは、CloudWatch コンソールから設定を変更できます。左メニューから【ルール】を選択し、個々のルール（ここでは MyScheduleRule）をクリックして（図3-34）、右上の【アクション】メニューから【編集】を選ぶと（図3-35）、ルールの作成画面になります（図3-36）。



図 3-34 ルールを選択する



図 3-35 ルールの編集画面を開く



図 3-36 スケジュールの時間や実行する Lambda 関数の割り当てなどを変更できる

第4章

S3 のイベント処理



前章までに Lambda サービスに関する基本的な内容の解説は終わりましたので、本章からは、Lambda を使った実例を紹介します。最初に紹介する実例は、S3 に対するアクセスが発生したときの、Lambda 関数での処理方法についてです。具体的には、ある S3 バケットにファイルがアップロードされたとき、それを別の S3 バケットに暗号化 ZIP で保存するという処理を実装してみます。

- 4-1 S3 のイベント事例 [p.88]
- 4-2 S3 バケットの作成 [p.89]
- 4-3 S3 バケットに対するイベント [p.95]
- 4-4 ライブドリ込みの Lambda 関数の作成 [p.107]
- 4-5 まとめ [p.123]

本章のポイント

● S3 にファイルがアップロードされたときに Lambda 関数を呼び出す仕組み

まずは、S3 にファイルがアップロードされたときに Lambda 関数を呼び出す仕組みについて理解する必要があります。

設定の方法はもちろん、ファイルがアップロードされたときには、イベント引数にどのような値が渡されるのかも知らなければなりません。

● S3 の操作方法

S3 バケットにアップロードされたファイルを読み取り、暗号化 ZIP に変換し、それをもうひとつの S3 バケットに書き込む方法です。

これは、次の 3 手順に分かれます。

(1) S3 バケットからファイルの読み込み

Lambda 関数が呼び出されるときには、配置された S3 バケット上でファイルを特定するキー名（パス名を含むファイル名）が渡されます。そこで、そのキー名に相当するデータを、一度、Lambda コンテナにダウンロードします。

AWS では、Python から AWS のさまざまなサービスを操作するための「Boto3」と呼ばれるライブラリを提供しています。Lambda 関数では、このライブラリを使って、S3 バケットから Lambda コンテナにファイルをダウンロードします。

(2) 暗号化 ZIP ファイルの作成

暗号化 ZIP を作るには、いくつかの方法がありますが、本書では、`pyminizip` というライブラリを使います。このライブラリは、「zlib ライブラリ」を使って、暗号化 ZIP を作ります。

【pyminizip】

<https://pypi.python.org/pypi/pyminizip>

(3) S3 へのファイルの書き込み

(1) と同様に Boto3 ライブラリを使って、(2) で作成した暗号化 ZIP を目的の S3 バケットへとアップロードします。

● ライブラリを含む Lambda 関数を登録する方法

「S3 の操作方法」で説明したように、本章で作る Lambda 関数の実行には、「Boto3」と「pyminizip」の 2 つのライブラリが必要です。前者は Lambda コンテナに標準で含まれますが、後者は含まれません。そのため AWS Lambda には、Lambda 関数とともに、pyminizip ライブラリも、一緒に登録する必要があります。

詳しくは、「4-4 ライブラリ込みの Lambda 関数の作成」で説明しますが、ライブラリを必要とする場合、そのライブラリを Lambda 関数とともに ZIP 形式としてまとめます。このファイルを「Lambda デプロイパッケージ」と言います。

具体的には、開発用の別マシン（本書では EC2 インスタンスを使います）でライブラリをビルドし、そのマシンで作成したビルト後のライブラリファイルをコピーして、ZIP 形式にまとめます。そして、それを AWS Lambda に登録します（図 4-1）。

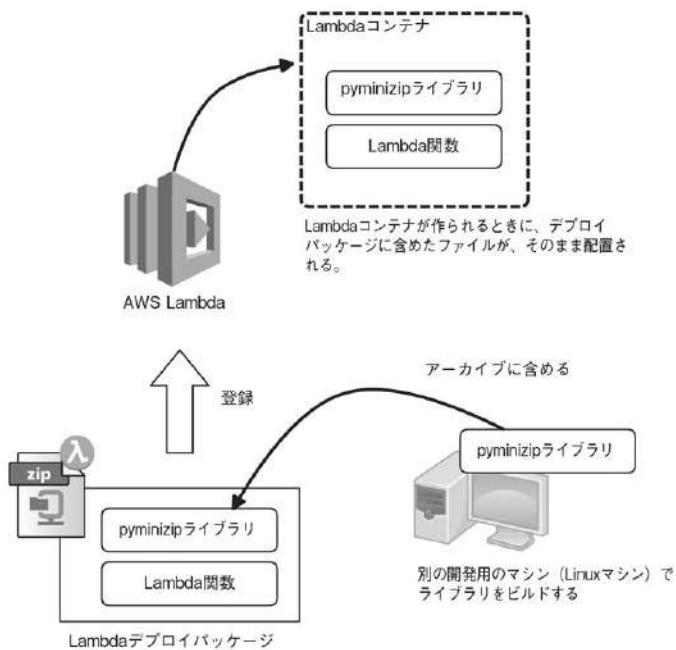


図 4-1 ライブラリを使う場合は、Lambda デプロイパッケージとしてまとめる

4-1 S3 のイベント事例

本章では、S3 のイベントを扱う例として、「S3 バケットにファイルがアップロードされたとき、それを暗号化 ZIP 形式ファイルとして、別の S3 バケットに保存する」という実例を取り上げます。

そのためには、「S3 バケットにファイルがアップロードされたとき」のイベントで Lambda 関数を呼び出すように設定します。呼び出される Lambda 関数は、そのファイルを読み込んで暗号化 ZIP に変換し、もうひとつの S3 バケットに保存するように実装しておきます（図 4-2）。

図 4-2 の構成からわかるように、この Lambda 関数は、S3 バケット上のファイルをダウンロードして処理し、別の S3 バケットへとアップロードしています。そのため、Lambda 関数の IAM ロールは、前者の S3 バケットには読み取り権限が、後者の S3 バケットには書き込み権限がないと失敗します。

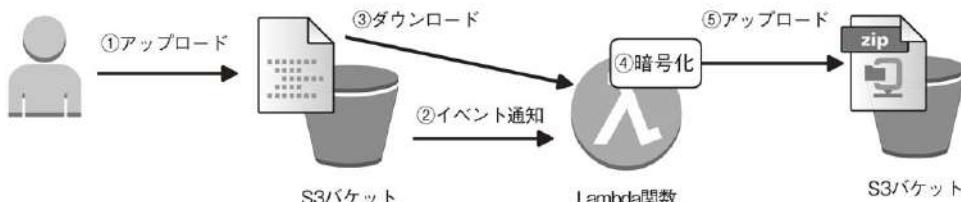


図 4-2 S3 バケットにファイルがアップロードされたとき、別の S3 バケットに暗号化 ZIP ファイルとして保存する例

Column イベントのループに注意

図 4-2 では、暗号化 ZIP ファイルを保存する S3 バケットが、元とは違う S3 バケットである点に注目してください。もし同じ S3 バケットに暗号化 ZIP ファイルを書き込むと、それに対して、またイベントが発動して Lambda 関数が実行されるので、暗号化 ZIP ファイルをさらに暗号化 ZIP にするというように、永遠にループしてしまいます。

もし、同じ S3 バケットに対して書き戻すときは、たとえば、「書き込み先を特定のディレクトリとして、そのディレクトリに書き込まれたファイルはイベント処理しない」「拡張子が.zip のファイルはイベント処理しない」のように構成し、ループが発生しないように注意してください。

4-2 S3 バケットの作成

これから、元ファイルを置いたり、暗号化したZIPファイルを配置したりするためのS3バケットを作っていきます。S3コンソールからバケットの各種設定を行い、アクセス権を設定します。なお、ここでは、元ファイルを置くバケットを「examplerread000」、暗号化したZIPファイルを置くバケットを「examplewrite000」として作成します。



Memo S3 バケットの名前

AWSシステム内において、S3バケットの名前はワールドワイドでユニークに設定されなければなりません。本書の実例において、バケット名として「examplerread000」「examplewrite000」という名前を使っているため、読者の方は同名のバケットを作成できません。各自で他の適当なバケット名を付け、今後の操作事例では各自で作成したバケット名に読み替えてください。

4-2-1 S3 バケットの作成

◎ 操作手順 ◎ S3 バケットを作成する

[1] S3 コンソールを開く

- S3バケットは、AWSマネジメントコンソールの「S3」から操作します。まずは、S3コンソールを開いてください（図4-3）。



図 4-3 S3 コンソールを開く

[2] バケットを作成する

- ・[バケットを作成する] をクリックして、S3 バケットを作成します（図 4-4）。



図 4-4 バケットを作成する

[3] バケット名やリージョンを設定する

- ・バケット名やリージョンを設定します。まずは、「examplerread000」という名前のバケットを作成します（図 4-5）。



図 4-5 バケット名やリージョンを設定する

- ・リージョンは、これから Lambda 関数を配置する予定のリージョンと同じでなければなりません（異なるリージョンの Lambda 関数には、イベントを発行できません）。ここでは、「アジアパシフィック（東京）」を選び、【次へ】をクリックします。

[4] プロパティの設定

- バージョニングやログの記録、タグ付けなどを設定します。ここでは、とくに何も指定する必要はないので、[次へ] をクリックして次の画面に進んでください（図 4-6）。



図 4-6 プロパティの設定

[5] アクセス許可の設定

- アクセス許可を設定します。この画面では、ユーザーに対する設定しかできません。あとの操作で、ロールに対するセキュリティを設定するので、[次へ] をクリックします（図 4-7）。



図 4-7 アクセス許可の設定

[6] 確認

- 確認画面が表示されます。[バケットを作成] をクリックして、バケットを作成してください(図 4-8)。



図 4-8 確認

[7] もうひとつ S3 バケットを作る

- 同じように操作して、もうひとつ「examplewrite000」という名前のバケットを作成してください(図 4-9)。[次へ] をクリックしてください。以降は図 4-6 から図 4-8 と同手順です。



図 4-9 「examplewrite000」というバケットを作る

4-2-2 S3 バケットに Lambda 関数からアクセスできるようにする

次に、作成した S3 バケットに対して、Lambda 関数からアクセスできるように設定します。そのためには、①S3 バケット側で設定する方法、②アクセスする IAM ロール側で設定する方法の 2 通りがあります。ここでは後者の設定方法を使います。

本書では、Appendix B で作成方法を示している「role-lambdaexec」というロールのもとで Lambda 関数を実行するものとします。そこで role-lambdaexec ロールに対して、あらかじめ S3 バケットへのアクセス権を設定しておきます。

IAM ロールに対して S3 バケットへのアクセス権を設定する方法は、いくつかありますが、ここでは話を簡単にするため、role-lambdaexec ロールに対して、S3 へのフルアクセス権である AmazonS3FullAccess という管理ポリシーを適用しておくものとします。詳しい設定方法は、Appendix B を参照してください。

λ Column 細かいポリシーをひとつずつ追加する

AmazonS3FullAccess ポリシーはフルアクセス権を与えるので、どの S3 バケットにもアクセスできてしまい、好ましくありません。実運用では、必要な S3 バケットだけにアクセス権を与えるべきです。そうするにはいくつかの方法がありますが、ここでは【インラインポリシー】を追加する方法を紹介します。

IAM ロールのアクセス許可を設定する画面において、インラインポリシーの【ここをクリックしてください。】をクリックすると、インラインポリシーを追加できます（図 4-10）



図 4-10 インラインポリシーを追加する

ポリシーを簡易に追加するため、【Policy Generator】を選択します（図 4-11）。



図 4-11 Policy Generator を選択する

次に S3 へのアクセス権限を設定します（図 4-12）。[アクション] は、許す操作です。読み込み専用なら [GetObject] を設定します。[Amazon リソースネーム (ARN)] は、リソースの場所です。「arn:aws:s3:::バケット名/*」を指定して [ステートメントを追加] ボタンをクリックします。



図 4-12 アクセス権を設定する

すると設定行が追加されるので、[次のステップ] のボタンをクリックします（図 4-13）。



図 4-13 次に進む

最後に設定確認画面が表示されるので、[ポリシーの適用] ボタンをクリックすると、適用されます（図 4-14）。



図 4-14 ポリシーの適用

ここでは、読み込みを許可するため「GetObject」をアクセス権として設定しましたが、書き込みを許可する場合は「PutObject」を設定してください。

4-3 S3 バケットに対するイベント

S3 バケットの準備ができたので、S3 のイベントを処理する Lambda 関数のサンプルプログラムを作成します。ここで理解しておく必要があるのは、「S3 バケットではどのようなときにイベントが発生し」「そのイベント引数には、どのような値が渡されるか」という点です。

4-3-1 S3 バケットのイベント

S3 は、「①新しいオブジェクトの作成」「②オブジェクトの削除」「③オブジェクトの消失」という、3 つのカテゴリのイベントを発生することができます。本章のサンプルでは、「ファイルがアップロードされたときに暗号化 ZIP に変換する」ので、ファイルが作成されたときのイベントである、①の「s3:ObjectCreated」のイベントを用います。

s3:ObjectCreated イベントには、いくつかの種類がありますが、「Put」「Post」など、どのようなメソッドで新しいオブジェクト（S3 における「オブジェクト」とは「ファイル」や「ディレクトリ」のことです）が作成されたときでもイベントが発生するよう、メソッドの種類を問わない「s3:ObjectCreated:*」を用います。

①新しいオブジェクトの作成 (s3:ObjectCreated) (表 4-1)

表 4-1 新しいオブジェクトが作成されたときのイベント (s3:ObjectCreated)

イベントタイプ	意味
s3:ObjectCreated:*	下記のすべての場合
s3:ObjectCreated:Put	Put メソッドでオブジェクトが配置された
s3:ObjectCreated:Post	Post メソッドでオブジェクトが配置された
s3:ObjectCreated:Copy	Copy メソッドでオブジェクトが配置された
s3:ObjectCreated:CompleteMultipartUpload	Multipart Upload API でオブジェクトが配置された

②オブジェクトが削除された (s3:ObjectRemoved) (表 4-2)

表 4-2 オブジェクトが削除されたときのイベント (s3:ObjectRemoved)

イベントタイプ	意味
s3:ObjectRemoved:*	下記のすべての場合
s3:ObjectRemoved:Delete	一般的な削除
s3:ObjectRemoved:DeleteMarkerCreated	バージョニングによって削除マーカーが作成されたとき

③消失イベント (s3:ReducedRedundancyLostObject) (表 4-3)

低冗長化ストレージ (RRS) を用いているときに、オブジェクトが消失した場合に発生します。
「s3:ReducedRedundancyLostObject」という一種類のイベントしかありません。

表 4-3 オブジェクトが消失した場合のイベント (s3:ReducedRedundancyLostObject)

イベントタイプ	意味
s3:ReducedRedundancyLostObject	オブジェクトが消失した

4-3-2 イベント引数の書式

S3 のイベントが発生したときのイベント引数の例を JSON 形式で示すと、リスト 4-1 のようになります。これは、Lambda 関数のテストイベントのひな形（後掲の図 4-22）を掲載したものです。

リスト 4-1 S3 のイベント引数の例

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "s3": {
        "configurationId": "testConfigRule",
        "object": {
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901",
          "key": "HappyFace.jpg",
          "size": 1024
        },
        "bucket": {
          "arn": "arn:aws:s3:::mybucket",
          "name": "sourcebucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          }
        },
        "s3SchemaVersion": "1.0"
      },
      "responseElements": {
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklmabaisawesome/mnopqrstuvwxyz⇒
xyzABCDEFGH",
        "x-amz-request-id": "EXAMPLE123456789"
      },
      "awsRegion": "us-east-1",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "eventSource": "aws:s3"
    }
  ]
}
```

イベントの対象となったファイルの情報は、「Records」キーに配列として記録されています。配列のそれぞれのフィールドの意味は、表 4-4 の通りです。

表 4-4 S3 のイベント引数のフィールド

フィールド名	意味
eventVersion	イベントのバージョン。本書執筆時点（※）では「2.0」
eventSource	イベントのソース。「aws:s3」に固定
awsRegion	リージョンコード
eventTime	イベントの発生日時。ISO 8601 形式
eventName	発生したイベント名
userIdentify	操作したユーザー情報
principalId	プリンシパル ID
requestParameters	リクエストしたホストの情報
sourceIPAddress	操作したホストの IP アドレス
responseElements	S3 からのレスポンス
x-amz-request-id	AWS で自動生成されたリクエスト ID
x-amz-id-2	処理した Amazon S3 のホスト ID
s3	S3 バケットの情報
s3SchemaVersion	この項目のバージョン番号。本書執筆時点（※）では「1.0」
configurationId	通知の設定で構成された ID
bucket	バケットに関する情報
name	バケット名
ownerIdentify	所有者の ID
arn	この S3 バケットの ARN
object	対象のオブジェクト（ファイル・ディレクトリ）の情報
key	ファイルやディレクトリ名
size	ファイルサイズ
eTag	Etag の値

※本書執筆時点（2017年7月）

本章で作成する Lambda 関数は、S3 にアップロードされたファイルをダウンロードして暗号化します。表 4-4 からわかるように、そのファイル名は「`s3.object.key`」で取得できます。また、バケット名は「`s3.bucket.name`」で取得できます。

λ Column イベント引数の実際の値を確認する

イベント引数に、どのような値が渡されているのかを確認するときは、CloudWatch Logs に出力してみるとよいでしょう。

たとえば、次のような Lambda 関数を作ります。

```
def lambda_handler(event, context):
    print(json.dumps(event, indent=2))
```

4-3-3 S3 のイベント処理の例

では、S3 のイベントから Lambda 関数を実行できるように構成したときに、本当に「`s3.object.key`」に、配置したファイル名が渡されるのかを確認してみましょう。ここでは、`s3.object.key` の値を `print` するだけの簡単な Lambda 関数を作って実行し、CloudWatch Logs に出力される結果を確認します。

◎ 操作手順 ◎ S3 バケットにファイルがアップロードされたときに Lambda 関数を実行するように構成する

【1】 設計図を選択する

Lambda コンソールを開き、Lambda 関数の作成を始めてください。

- ・S3 バケットに何か配置されたときに操作するには、「`s3-get-object`」の設計図から操作するのが簡単です。その Python3 版である `s3-get-object-python3` を選択します（図 4-15）。



図 4-15 「`s3-get-object-python3`」の設計図を選択する

[2] トリガーを設定する

- どのS3バケットに対して、どのようなイベントが発生したときに、Lambda関数を実行するのかを設定します（図4-16）。

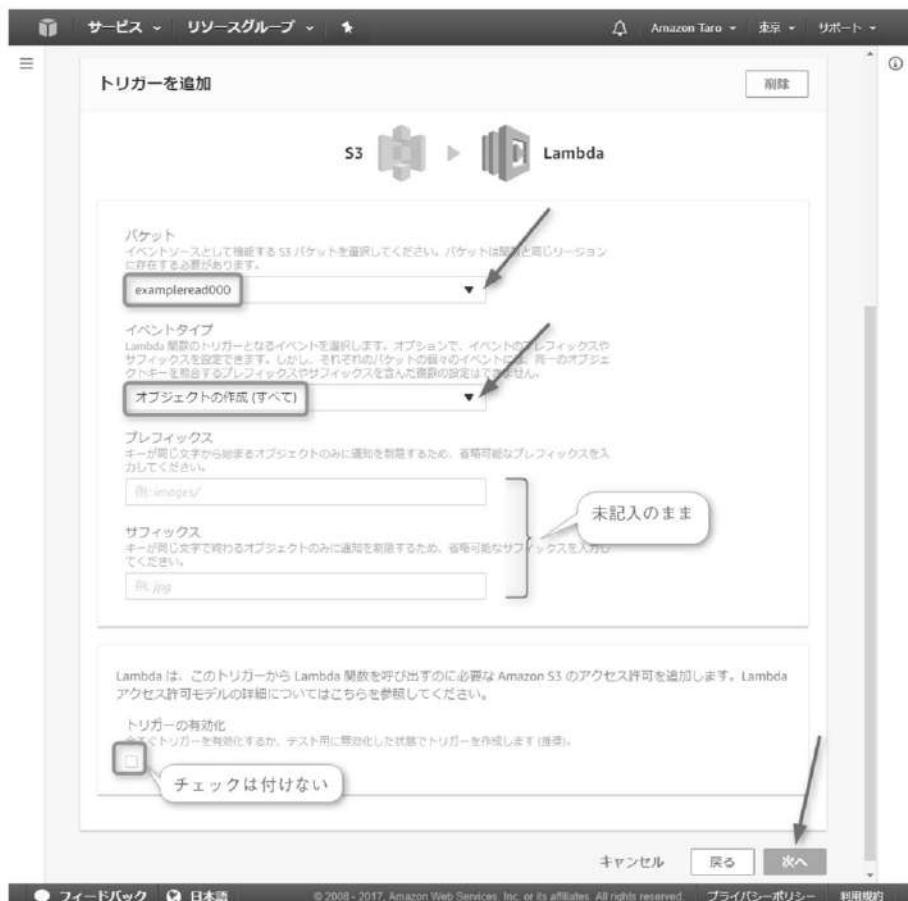


図4-16 トリガーの設定

バケット

対象のバケット名を選択します。ここまで手順の場合、ファイルが置かれることになる「examplered000」です。

イベントタイプ

イベントの種類です。ファイルが置かれたときに処理するため、「オブジェクトの作成（すべて）」を選択します。これは、先に説明した「s3:ObjectCreated:*」に相当します。

プレフィックス

特定のパス名やファイル名から始まるものだけを処理するときは、そのプレフィックスを記述します。ここでは、置かれたすべてのファイルを処理するので、未記入とします。

サフィックス

特定の末尾のもの（主に拡張子だが、拡張子を越えた範囲でもかまわない）のときだけ処理するときは、サフィックスを設定します。ここでは、置かれたすべてのファイルを処理するので、未記入とします。

一番下の【トリガーの有効化】は、作成した直後にイベントを有効にするかどうかです。動作テストしてから有効化したほうがよいので、チェックは付けないでおきましょう。設定が終わったら【次へ】をクリックします。

[3] Lambda 関数の設定

・Lambda 関数を作ります。「名前」のフィールドに適当な関数名を入力します。ここでは「S3ExampleFunction」としました（図 4-17）。



図 4-17 関数名を設定する

・そして Lambda 関数を書きます。先に説明したように、ここでは、表 4-4 に示したように、「`s3.object.key`」の内容を CloudWatch Logs に出力するものとします。そのプログラムは、リスト 4-2 のようになります。設計図に書かれているプログラムをすべて消して、リスト 4-2 の内容を記述してください（図 4-18）。

リスト 4-2 s3.object.key の内容を CloudWatch Logs に出力するプログラム

```
def lambda_handler(event, context):
    for rec in event['Records']:
        print (rec['s3']['object']['key'])
```



図 4-18 Lambda 関数を入力する

[4] ハンドラ名とロールの選択

- ハンドラ名とロールを選択します。リスト 4-2 では「lambda_handler」という関数名を付けたので、ハンドラ名はデフォルトの「lambda_function_lambda_handler」のままとします。
- ロールは Appendix B で設定している「role-lambdaexec」ロールを選択します（図 4-19）。設定が済んだら [次へ] をクリックします。



図 4-19 ハンドラ名とロールの選択

● 4-3 S3 バケットに対するイベント

[5] 確認と作成

- 確認画面が表示され（図 4-20）、[関数の作成] をクリックすると、Lambda 関数が作成されます。



図 4-20 Lambda 関数の確認と作成

4-3-4 動作テストと実行

では、ここまでに作成した Lambda 関数（lambda_handler）の動作をテストしましょう。

■ テストイベントでテストする

[アクション] メニューから [テストイベントの設定] を選択します（図 4-21）。



図 4-21 テストイベントを作成する

イベントソースが S3 である場合、デフォルトのテストとして、S3 にファイルがアップロード

第4章 S3 のイベント処理

されたときのイベントデータをエミュレートする「S3 Put」というテストデータが設定されています。そのまま [保存してテスト] をクリックし、このテストを実行してください（図 4-22）。



図 4-22 「S3 Put」のエミュレートデータでイベントをテストする

そして、動作の結果に問題がないかを確認してください。図 4-22 は「HappyFace.jpg」というファイルをアップロードしたときのエミュレートデータです。このファイル名が、`s3.object.key` という値として取得でき、CloudWatch Logs に書き込まれます（図 4-23）。



図 4-23 結果を確認する

■イベントを有効にしてテストする

では次に、イベントを有効にして、実際に S3 にファイルを配置したときに実行されることを確認しましょう。【トリガー】タブで【有効化】をクリックしてください（図 4-24、図 4-25）。



図 4-24 イベントを有効化する



図 4-25 有効化の確認

S3 コンソールを開いて、ファイルをアップロードしてみましょう（図 4-26、図 4-27）。



図 4-26 S3 コンソールでファイルをアップロードする

第4章 S3 のイベント処理

トリガーを有効化しているので、この操作によって Lambda 関数が実行されるはずです。



図 4-27 アップロードが完了したところ

ここでは、「myface.png」というファイルをアップロードしてみました。アップロードが完了したら、CloudWatch Logs を確認しましょう。「myface.png」という値が出力されたことがわかります（図 4-28、図 4-29）。



図 4-28 ログを確認する①



図 4-29 ログを確認する②

4-4 ライブラリ込みの Lambda 関数の作成

ここまで実験で、S3においてファイルが作成されたときには、そのファイル名が `s3.object.key` に渡されることがわかりました。あとは、このプログラムを改良し、該当のファイルを読み込んで暗号化 ZIP ファイルにして、もう一方の S3 バケットにアップロードする処理を実装すれば、目的を達成できます。

4-4-1 ライブラリを必要とする Lambda 関数の作り方

本書では、暗号化 ZIP を作成するのに「pyminizip」というライブラリを使います。このライブラリは「zlib ライブラリ」を使って暗号化 ZIP ファイルを作ります。こうしたライブラリを利用した Lambda 関数を作成する場合は、「Lambda 関数」と「利用したいライブラリ」を、ひとつの ZIP 形式ファイルにまとめてアップロードします。このように ZIP 形式でまとめたものを Lambda デプロイパッケージと言います（図 4-30）。



図 4-30 Lambda デプロイパッケージ



Column 画像ファイルやデータなども含められる

Lambda デプロイパッケージは、その内容が、そのまま Lambda コンテナにコピーされます。そのため、ライブラリ以外に、Lambda 関数で利用したい画像ファイルや各種データなども同梱できます。なお、第 3 章の表 3-1 に示したように、配置されたディレクトリのルートパスは、`LAMBDA_TASK_ROOT` という環境変数で取得できます。

4-4-2 EC2 で開発する

では、ライブラリは、どのように作ればよいのでしょうか？ とくに今回利用する pyminizip ライブラリは、C 言語で書かれたライブラリなので、OS や CPU に依存します。gcc を使ったコンパイルが必要です。コンパイルのやり方は、いくつかありますが、最も簡単なのは、「EC2 インスタンスで開発して、それを Lambda デプロイパッケージにする」というものです。

EC2 インスタンスは、AWS における仮想サーバーです。第2章で説明したように、Lambda コンテナは、Amazon Linux AMI をベースとした実行環境です。この Amazon Linux AMI を使った EC2 インスタンスを作り、その EC2 インスタンス上で、ライブラリのビルドやコンパイルします。それらの成果物を ZIP ファイルとして固め、Lambda デプロイパッケージとして、AWS マネジメントコンソールからアップロードするのです（図 4-31）。

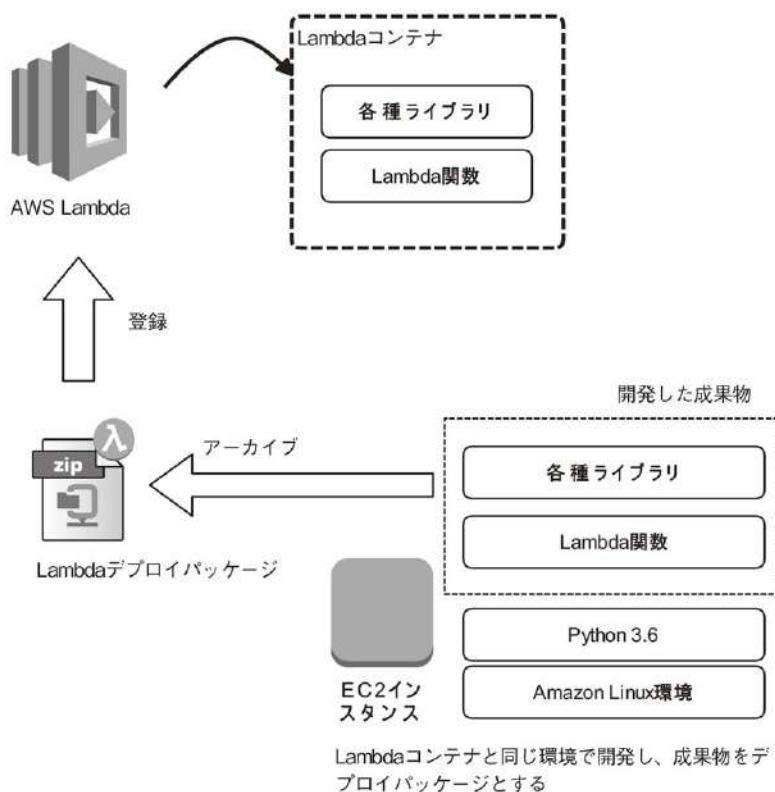


図 4-31 EC2 環境で Lambda 開発する

■ EC2 インスタンスで開発するときのポイント

Lambda 開発で使う EC2 インスタンスの作り方は、Appendix C に記載しました。Appendix C に記載した通りに操作して、EC2 インスタンスを作成してください。

Lambda 開発に使う EC2 インスタンスは、次の点がポイントとなります。

①あとでライブラリをひとまとめにしやすくする

図 4-31 に示したように、EC2 インスタンス上で開発したプログラムやライブラリの一式は、あとでアーカイブすることになります。アーカイブしやすくするために、開発したプログラムやライブラリは、ひとつのディレクトリにまとめておくようにします。

すぐあとに説明しますが、Python で開発する場合、その手法として、「virtualenv」というツールを利用できます。virtualenv は、仮想的に Python の実行環境を作るツールで、ディレクトリごとに独立した Python の実行環境を作ることができます。

通常、ライブラリをインストールすると、システムのディレクトリにインストールされますが、virtualenv を使うことで、ライブラリも、そのディレクトリ以下に格納できます。



Column 開発に EC2 インスタンスは必須なのか？

本章では、Lambda 開発に EC2 インスタンスを用いていますが、EC2 インスタンスは必須なのでしょうか？たとえば、Windows や Mac などの Python 環境で開発できないのでしょうか？答えは、「どのようなライブラリを利用するか」と「S3 などの AWS リソースにアクセスするかどうか」によります。もし、バイナリのライブラリを利用するのであれば、ライブラリをビルドするのに、Amazon Linux をベースとした EC2 インスタンスが必須です（厳密に言うと、Amazon Linux は Red Hat ベースなので、Red Hat システムでも可能です）。

S3 などの AWS リソースにアクセスする場合、開発機（Windows や Mac）で動作テストしたいなら、アクセス権が問題となります。これらのクライアントには、EC2 と違って実行ロールを設定できないからです。代わりに、「アクセスキー」と「シークレットアクセスキー」という鍵ペアを使って認証する方式をとるため、Lambda とは少し違った処理になります。

短いプログラムで、しかも、Python のみで完結するような Lambda 関数であれば、実際、Windows や Mac でも開発できるでしょう。しかし AWS では、EC2 インスタンスを構築するのは難しいことではありません。しかも、使わないときは、停止しておけば、(EBS のディスク容量以外は) 費用がかかりません。環境の違いなどで動かないなどのトラブルを考えると、あえて Windows や Mac 上で Lambda 開発をする利点はないと思います。

②EC2インスタンスの実行ロールをLambda関数と合わせる

EC2インスタンスでLambdaを開発する場合、都度、Lambdaデプロイパッケージを作つてアップロードしてテストするのは煩雑です。ある程度、EC2インスタンス上で動作テストできると、開発効率が格段と上がります。そのときに注意したいのが、EC2インスタンスのIAMロールです。EC2インスタンスのIAMロールを、Lambda関数を実行するIAMロールと同じセキュリティ構成にしておくと、EC2インスタンスからS3などのAWSリソースにアクセスすることができます(図4-32)。

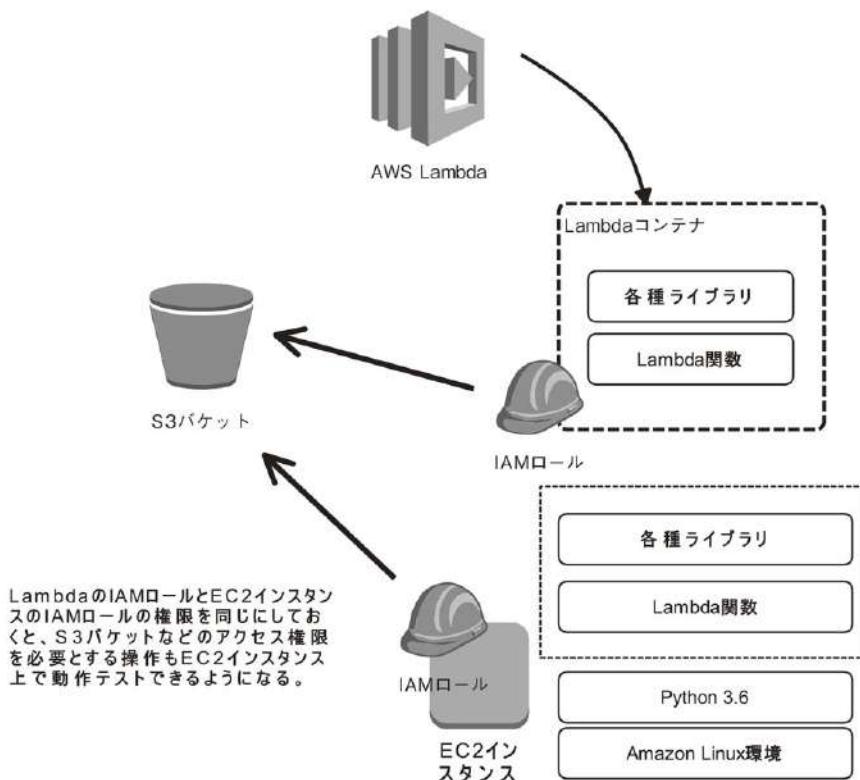


図4-32 EC2インスタンスはLambda関数と同じIAMロールに設定しておく

4-4-3 ライブラリのインストールと環境整備

まずは、Lambda関数を開発するための環境整備を進めていきましょう。

以下の手順は、Appendix Cに記載した通りに、EC2インスタンスを実行しており、「Python3.6」と「pip」「virtualenv」のソフトウェアがインストールされていることを前提とします。

■ virtualenv 環境を作る

本書では、ディレクトリ単位で Python 実行環境を分けることができる「virtualenv」というツールを開発に使います。

開発用の EC2 インスタンスに SSH で接続したら、適当な開発用ディレクトリを作成してください。たとえば、次のようにすると、カレントディレクトリに「encryptfile」というディレクトリができ、そのなかに、仮想的な Python の開発環境が作られます。なお、指定している「-p python 3.6」というのは、Python 3.6 の環境を作るという意味です。



Memo Python3 仮想環境の切り替え

Appendix C の手順では、Python3.6 環境を /usr/local/bin にインストールしています。 /usr/bin には Python2.7 がインストールされているので、下記のようにフルパスで入力しないと、Python2.7 環境の pip が起動してしまうので注意してください。

```
$ /usr/local/bin/virtualenv -p python3.6 encryptfile
```

この開発環境に切り替えるには、カレントディレクトリを変更してから、`activate` コマンドを実行します。

```
$ cd encryptfile/
$ source bin/activate
```

すると、コマンドプロンプトが次のように変わり、「(encryptfile)」というように、仮想環境のなかにいることがわかるようになります。

```
(encryptfile)[ec2-user@... encryptfile]$
```

この状態で、Python コマンドを実行したり、ライブラリをインストールしたりすると、この `encryptfile` ディレクトリ以下に対して実行されるようになります。

仮想環境での作業を終えて、元に戻るには、

```
$ deactivate
```

と入力します（いまは、そのまま作業を続けるので、入力しないでください）。

■ Boto3 ライブラリをインストールする

開発用のディレクトリを作ったら、開発を進めていきましょう。Python から S3 などの AWS サービスにアクセスするには、「Boto3」というライブラリを使います。まずは、このライブラリをインストールしましょう。virtualenv をアクティベートした状態で、次のコマンドを入力して、Boto3 をインストールします。

```
$ pip install boto3
```

ここでは、「`sudo pip install boto3`」ではなく「`pip install boto3`」であることに注意してください。`sudo` を付けなければ、virtualenv 環境の下にインストールされます。もし `sudo` を付けてしまうと、仮想環境外のメインの Python（これは Python2.7）に対してインストールされてしまします。

■ pyminizip をインストールする

次に、暗号化 ZIP ファイルを作成するライブラリである `pyminizip` をインストールします。このライブラリには、「`gcc`」と「`zlib`」が必要です。そこでまず、これらをインストールしておきます（これらは `sudo` を指定して root ユーザーでインストールします）。

```
$ sudo yum install -y gcc zlib-devel
```

そして次に、`pyminizip` をインストールします（こちらは `sudo` を付けません）。

```
$ pip install pyminizip
```



Memo 暗号化 ZIP を作る

`gcc`、`zlib-devel`、`pyminizip` のインストールは、「暗号化 ZIP を作る」という本章のサンプルのためにだけ必要な作業です。汎用的な Lambda 開発に必要な作業ではありません。

■ EC2 インスタンスの IAM ロールを調整する

本章では、S3 バケットを読み書きする Lambda 関数の開発やデバッグに、EC2 インスタンスを利用するため、EC2 インスタンスからも S3 バケットにアクセスできるように、EC2 インスタンスの IAM ロールを調整します。本章では、Appendix C で示しているように、開発用の EC2 インス

● 4-4 ライブリ込みの Lambda 関数の作成

タンスは、「role-ec2-lambdaexec」というロールで実行されているものとします（図 4-33）。



図 4-33 EC2 インスタンスの実行ロール

この role-ec2-lambdaexec というロールに対して、S3 へのフルアクセス権 (AmazonS3FullAccess) を与えておきます。詳細な手順は、Appendix C を参考にしてください。



図 4-34 EC2 インスタンスの IAM ロールにも S3 へのフルアクセス権を設定する

λ Column IAM Policy Simulator

AWS では、さまざまなものでアクセス権を設定できるので、あるユーザーやグループ、ロールに対して、アクセス権があるかどうか調べる作業は複雑になります。

そうしたときには、IAM Policy Simulator を使いましょう。IAM Policy Simulator では、ユーザーやグループ、ロールが、S3 などの AWS リソースへのアクセス権を持つかどうかを調べることができます。

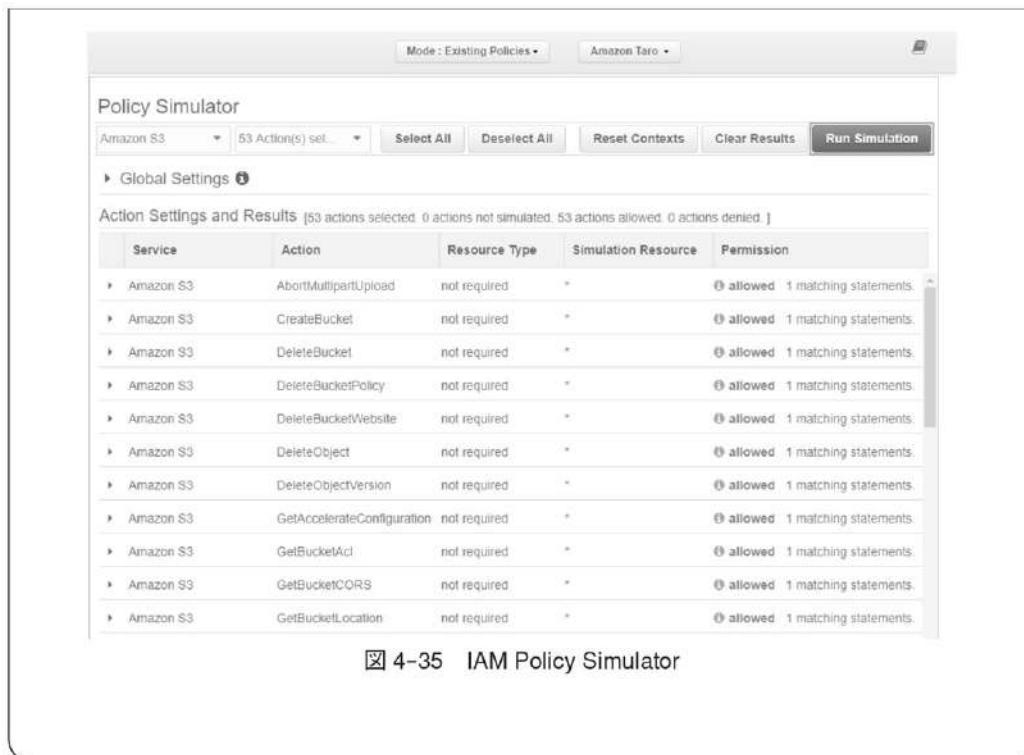


図 4-35 IAM Policy Simulator

4-4-4 暗号化 ZIP を作成する Lambda 関数

EC2 インスタンスの作成、Python 3.6 の開発環境のインストール、EC2 インスタンスの IAM ロールの設定が終わりましたので、ライブラリ込みの Lambda 関数を作成する準備が整いました。以下では S3 バケットのイベントを処理する Lambda 関数について手順を追って作成していきます。

■ S3 バケット間の移動と暗号化

いきなり Lambda 関数として考え始めると難しいので、まずは、Lambda は関係なく、S3 バケットからファイルをダウンロードして、それを暗号化し、別の S3 バケットにアップロードするというプログラムを作ってみましょう（リスト 4-3）。

S3 バケットには、すでに図 4-27 の手順で「myface.png」というファイルを置いているので、これを使います。すなわち、以下の 3 つの処理を行います。

- ① 「examplerread000」バケットに保存されている「myface.png」を読み込む
- ② 暗号化する

③ 「examplewrite000」 パケットに書き込む

リスト 4-3 S3 を操作するプログラムの例（Lambda とは関係ない。s3example.py）

```

import boto3
import pyminizip
import tempfile
import os

filename = 'myface.png'
s3 = boto3.resource('s3')

# ファイルの読み込み
obj = s3.Object('examplerread000', filename)
response = obj.get()
tmpdir = tempfile.TemporaryDirectory()
fp = open(tmpdir.name + '/' + filename, 'wb')
fp.write(response['Body'].read())
fp.close();

# 暗号化
zipname = tempfile.mkstemp(suffix='.zip')[1]
os.chdir(tmpdir.name)
pyminizip.compress(filename, zipname, 'mypassword', 0)

# S3にアップロード
obj = s3.Object('examplewrite000', filename + '.zip')
response = obj.put(
    Body=open(zipname, 'rb')
)

tmpdir.cleanup()
os.unlink(zipname)

```

リスト 4-3 の処理は、次のようにになります。

①S3 オブジェクトの取得

S3 を操作するには、S3 オブジェクトを取得します。

```
s3 = boto3.resource('s3')
```

②S3 パケットからの読み取り

S3 パケットから読み取るには、次のようにしてオブジェクトを得ます。「examplerread000」はバ

ケツト名、filename はファイル名（S3 オブジェクトのキー名）です。

```
obj = s3.Object("exampleread000", filename)
```

このオブジェクトに対して get メソッドを実行すると、その詳細情報が得られます。

```
response = obj.get()
```

このようにして取得した response の ['Body'] の read メソッドを呼び出すと、ファイルのデータをダウンロードできます。リスト 4-3 では、次のように、一時ディレクトリを作って、そこに書き込んでいます。

```
tmpdir = tempfile.TemporaryDirectory()
fp = open(tmpdir.name + '/' + filename, 'wb')
fp.write(response['Body'].read())
fp.close();
```

③暗号化する

pyminizip.compress メソッドを使って、暗号化 ZIP ファイルを作ります。ここでは、「mypassword」というパスワードにしました。最後に指定している「0」は、圧縮率です。ここではデフォルトの「0」を指定しています。

```
zipname = tempfile.mkstemp(suffix='.zip')[1]
os.chdir(tmpdir.name)
pyminizip.compress(filename, zipname, 'mypassword', 0)
```

④S3 にアップロード

S3 にアップロードします。S3 にアップロードするには put メソッドを実行し、「Body」に、そのデータを指定します。ZIP ファイル名は、元のファイル名に「.zip」を付けたもの (myface.png.zip) としました。

```
obj = s3.Object('examplerewrite000', filename + '.zip')
response = obj.put(
    Body=open(zipname, 'rb')
)
```

⑤一時ファイルの削除

すべての操作が終わったら、一時ファイルを削除します。

```
tmpdir.cleanup()
os.unlink(zipname)
```

■実行して確認する

```
$ python s3example.py
```

実行したあと、「example000write」バケットを確認すると、暗号化 ZIP となったファイルがアップロードされていることがわかります（図 4-36）。



図 4-36 暗号化 ZIP ファイルが作成されたことの確認

4-4-5 Lambda 関数として作る

これで S3 バケットへのアクセスができるようになりました。あとは、Lambda 関数にするだけです。すでにリスト 4-2 で説明したように、次のように記述すれば、アップロードされたファイルがわかります。

```
def lambda_handler(event, context):
    for rec in event['Records']:
        filename = rec['s3']['object']['key']
```

そこで、このプログラムとリスト 4-3 を組み合わせて、リスト 4-4 のようにすれば、Lambda 関数化できます。

リスト 4-4 は、のちに実際に Lambda にアップロードして Lambda 関数として使います。このファイル名は、「lambda_function.py」としておきます。

ファイル名は、Lambda コンソールで「ハンドラ名」として指定する値の一部になります（前掲の図 4-19）。Lambda コンソールから直接コード入力する場合は、「lambda_function.py」という名前が暗黙的にデフォルトとなるため、Lambda 関数を作る場合は、このファイル名にしておくのが無難です（そうでない場合は、Lambda コンソールで登録するとき、ハンドラ名を変更する必要

があります)。

リスト 4-4 では、対象とするファイル名とバケット名を、次のように、イベント引数の `rec['s3']['object']['key']`、`rec['s3']['bucket']['name']` から取得しているという点だけが違い、残りの処理は、リスト 4-3 とほぼ同じです。

```
filename = rec['s3']['object']['key']
obj = s3.Object(rec['s3']['bucket']['name'], filename)
```

リスト 4-4 Lambda 関数にしたもの (lambda_function.py)

```
import boto3
import pyminizip
import tempfile
import os

def lambda_handler(event, context):
    s3 = boto3.resource('s3')
    for rec in event['Records']:
        filename = rec['s3']['object']['key']
        obj = s3.Object(rec['s3']['bucket']['name'], filename)
        response = obj.get()
        tmpdir = tempfile.TemporaryDirectory()
        fp = open(tmpdir.name + '/' + filename, 'wb')
        fp.write(response['Body'].read())
        fp.close();

        # 暗号化
        zipname = tempfile.mkstemp(suffix='.zip')[1]
        os.chdir(tmpdir.name)
        pyminizip.compress(filename, zipname, 'mypassword', 0)

        # S3にアップロード
        obj = s3.Object('examplewrite000', filename + '.zip')
        response = obj.put(
            Body=open(zipname, 'rb')
        )

        tmpdir.cleanup()
        os.unlink(zipname)
```

■ EC2 インスタンス上で、Lambda 関数をデバッグする

すぐあとに実際に試しますが、リスト 4-4 は Lambda デプロイパッケージを作つて Lambda に登録すれば、きっと動くはずです。しかし、もし動かなければ、また EC2 インスタンス上でプログラムを修正して、デプロイパッケージを作成し直す手順が発生してしまいます。ですから、この時点では、ある程度、動くかどうかの確認をとりたいと思います。

実は、その方法は、難しくありません。`event` 引数に、それらしい値を渡して、呼び出してしまえばよいのです。それには、テストに使つた JSON 文字列（前掲の図 4-22）を使います。リスト 4-4 では、渡された `event` 引数のうち、`['s3']['object']['key']`、`['s3']['bucket']['name']` のキーしか使っていません。そこで、リスト 4-4 の後ろにこうした定義を加え、リスト 4-5 のようにします。

リスト 4-5 テスト用のコードも含めたもの

```

import boto3
import pyminizip
import tempfile
import os

def lambda_handler(event, context):
    ... 略（リスト 4-4 と同じ） ...

# 以下、テスト用のプログラム
import json
if __name__ == "__main__":
    data = '''
{
    "Records": [
        {
            "s3": {
                "object": {
                    "eTag": "0123456789abcdef0123456789abcdef",
                    "sequencer": "0A1B2C3D4E5F678901",
                    "key": "myface.png",
                    "size": 1024
                },
                "bucket": {
                    "arn": "arn:aws:s3:::exampleread000",
                    "name": "exampleread000",
                    "ownerIdentity": {
                        "principalId": "EXAMPLE"
                    }
                }
            }
        }
    ]
}
    '''
    event = json.loads(data)

lambda_handler(event, None)

```

```
        }
    }
}
]
...
event = json.loads(data)
context = None
lambda_handler(event, context)
```

リスト 4-5 は、次のような定義をしておき、

```
data = JSON文字列で示したイベントテスト用データ
```

以下のようにして、lambda_handler 関数を呼び出しています。

```
event = json.loads(data)
context = None
lambda_handler(event, context)
```

つまり、テスト用のデータが渡され lambda_handler 関数が実行されます。

実際に実行してみましょう。

```
$ python lambda_function.py
```

そうすると、JSON 形式データが渡された状態で lambda_handler 関数が呼び出され、動作テストすることができます。なお、この動作テスト用のコードは、Lambda デプロイパッケージを作るときでも、消す必要はなく、そのまままでかまいません（もっとも、あまりにサイズが大きいようなら削除すべきですが）。なぜなら、Lambda は、登録したときには「イベントハンドラ」として登録した関数だけが実行されるので、それ以外の余計なところは、あっても害がないからです。

4-4-6 デプロイパッケージを作成して登録する

以上で、Lambda 関数の開発は完了です。最後に、デプロイパッケージを作って、Lambda コンソールからアップロードして登録しましょう。

■デプロイパッケージを作る

何度か説明しているように、デプロイパッケージは、①Lambda 関数、②ライブラリなど、を ZIP 形式ファイルとしてまとめたものです。次のようにして作ります。ここでは、作成する ZIP 形式ファイル名を「lambda_function.zip」という名前とします。

①Lambda 関数本体を ZIP ファイルにまとめる

まずは、Lambda 関数本体を ZIP ファイルにまとめます。今回の場合、リスト 4-5 に示した `lambda_function.py` です。

```
$ zip /tmp/lambda_function.zip lambda_function.py
```

②ライブラリを ZIP 形式ファイルにまとめる

`virtualenv` を用いている場合、そのディレクトリ以下の「`lib/python3.6/site-packages` 以下」(または「`lib64/python3.6/site-packages` 以下」) にライブラリがあります。このなかから必要なライブラリを探して次のようにして ZIP ファイルにまとめます。

```
$ cd lib/python3.6/site-packages
$ zip -r /tmp/lambda_function.zip *
```

まとめ方としては、ライブラリは「そのディレクトリ」ではなくて、ZIP ファイルのルート階層に配置するように作ります。今回の場合、`pyminizip` ライブラリを使っていますが、これはサブディレクトリに置くのではなく、`lambda_function.py` と同じルートの階層に存在することが必要です(図 4-37)。



図 4-37 ライブラリの階層例

ちなみに図4-37は、話をわかりやすくするためとライブラリの依存関係などで正しく動かないという問題を避けるため、site-packages以下をすべてアーカイブしていますが、実際に必要なのは「pyminizip.cpython-36m-x86_64-linux-gnu.so」というライブラリだけです。その他のPython3.6の基本ライブラリは、本来は含める必要はありません。また、Boto3ライブラリも含める必要はありません。

■デプロイパッケージをアップロードする

こうして作成したデプロイパッケージ（ZIPファイル）をAWS Lambdaにアップロードします。Lambdaコンソールを開いて、先ほどリスト4-2として作成したLambda関数を、このコードで差し替えます。

[コード]タブで [ZIPファイルをアップロード] にします。そして、このように作成したlambda_function.zipファイルをアップロードし、[保存]ボタンをクリックします（[保存]ボタンをクリックしないと適用されないので注意してください（図4-38）。



図4-38 デプロイパッケージをアップロードする

以上で設定は完了です。example000readのS3バケットにファイルを置くと、それが暗号化ZIPとなり、example000writeに置かれることを確認しましょう。

4-5 まとめ

本章では、S3 バケットのイベントを扱う方法と S3 を操作する方法、実行時にライブラリを必要とする Lambda 関数の作り方を説明しました。

①S3 バケットのイベントと操作

S3 バケットにファイルが置かれたときのイベントでは、イベント引数に、対象となったバケット名やファイル名などの情報が渡されます。

ファイル自体は渡されません。Boto3 ライブラリを使ってダウンロードする必要があります。このとき、適切な権限がないと、ダウンロードに失敗します。

②ライブラリを必要とする Lambda 関数

Amazon Linux 上で、Python 3.6 環境を作り、ライブラリなどをビルドします。そして、Lambda 関数本体とライブラリをデプロイパッケージとしてまとめて、AWS Lambda に登録します。

少し、長いプログラムになってくると、Lambda コンソールでソースコードを書いて実行するという開発スタイルは、現実的ではありません。

ですから、本章で説明したように、EC2 インスタンスに Python3 + Boto3 ライブラリの環境を作って、そこで開発・デバッグして、最終成果物をデプロイパッケージとしてアップロードするというやり方になるでしょう。

λ Column 環境変数を使う

リスト 4-4 のプログラムは、配置先のバケット名である example000write をソースコードに記述しています。そのため、配置先のバケット名を変更したいときは、デプロイパッケージの作り直しになってしまいます。

```
#S3にアップロード
obj = s3.Object('examplewrite000', filename + '.zip')
```

ビルト直さずに、保存先を自由に変更できるようにするには、「環境変数」を使うのが適切です。たとえば、Lambda コンソールにて、図 4-39 のように「outbucketname」という環境変数を作成しておきます。

このとき、プログラムからは、以下のようにすると、それを参照でき、出力先のバケット名を、デプロイし直すことなく、環境変数の設定変更だけでできるようになります。

```
obj = s3.Object(os.environ['outbucketname'], filename + '.zip')
```



図 4-39 環境変数を設定する

第5章

API Gateway、DynamoDB、 SESとの連携

本章では、Lambda を使って Web アプリケーションを実装します。そのためには、Lambda と API Gateway とを組み合わせます。そうすることで HTTPS 経由で Lambda 関数を実行できるようになり、入力フォームなどのデータを処理できるようになります。

本章で作成する Web アプリケーションでは、API Gateway との連携だけでなく、入力されたデータを DynamoDB という NoSQL データベースに書き込んだり、Amazon SES というメールサービスを使って、自動返信メールを送信したりします。また、最後に一定期間だけアクセスできる、コンテンツをダウンロードするための限定的な URL を作る方法も紹介します。

- 5-1 API Gateway のイベント事例 [p.127]
- 5-2 API Gateway と Lambda 関数を組み合わせる [p.131]
- 5-3 API Gateway から実行される Lambda 関数を作る [p.133]
- 5-4 DynamoDB の基本 [p.153]
- 5-5 Lambda 関数で DynamoDB にアクセスする [p.167]
- 5-6 署名付き URL を発行する [p.179]
- 5-7 メールの送信 [p.187]
- 5-8 クロスオリジンの場合の注意点 [p.199]
- 5-9 まとめ [p.204]

本章のポイント

● API Gateway を経由して Lambda 関数を呼び出す

API Gateway は、HTTPS 経由で Lambda 関数を呼び出すための中継機能です。API Gateway を構成すると、エンドポイントと呼ばれる HTTPS の URL が作成され、その URL に対してリクエストを送信することで、Lambda 関数が実行されるようになります。

このとき Lambda 関数には、HTTPS の GET メソッドや POST メソッドで送信されたデータや Web ブラウザから送信された User-Agent などのヘッダ情報などが、イベント引数として渡されます。

● DynamoDB にアクセスする

Lambda 関数において、データを永続的に保存したいときによく使うのが、DynamoDB です。DynamoDB は完全マネージドな NoSQL データベースサービスです。RDS と違って VPC 環境でなくてもアクセスできます。Python からは Boto3 ライブラリを使って、DynamoDB にアクセスします。

● 署名付き URL

S3 には、署名付き URL という機能があります。この機能を使うと、S3 バケットの特定のファイル（オブジェクト）に対して、一定期間だけアクセスできる URL を作成できます。

● Amazon SES を使ったメール送信

「Amazon SES (Simple Email Service)」というマネージドサービスを使うと、Lambda 関数からメールを送信できます。ただしデフォルトでは迷惑メール防止のため、送信できる宛先が、あらかじめ登録したものに限られています。

5-1 API Gateway のイベント事例

本章では、Web フォームから送信してきたデータを処理する Lambda 関数を作ります。そうした Lambda 関数は、API Gateway と組み合わせて作ります。API Gateway は、HTTPS プロトコルと Lambda 関数とを中継する機能です。

本章で取り上げる Web アプリケーションの事例では、Web フォームでユーザー情報を登録すると、特別なコンテンツをダウンロードできる URL がメールで通知される仕組みを作ります。このシステム構成を図示すると、図 5-1 のようになります。

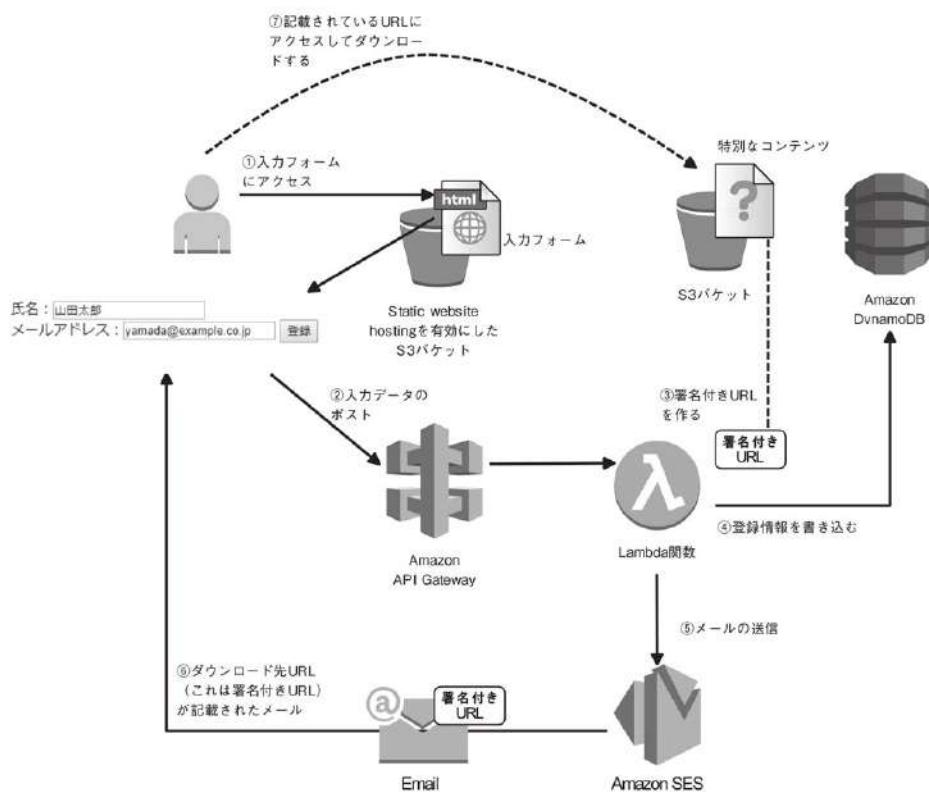


図 5-1 登録ユーザーが特別なコンテンツをダウンロードできるようにする仕組み

図 5-1 に示されているいくつかの要素について、簡単に説明します。

● 提供するコンテンツ

ユーザーがダウンロードできるコンテンツは、適当な S3 バケットにあらかじめ保存しておきます。配置した時点では、そのコンテンツへのアクセス権を、誰にも与えません。

●入力フォーム

「氏名」と「メールアドレス」を入力するWebフォームを用意します。このWebフォームは、静的なHTMLファイルです。ここでの事例では、この静的なHTMLファイルは、Amazon S3の「Static website hosting（静的ウェブホスティング）」の機能を使って配信します。



Memo 静的なHTMLファイルの配信

静的なHTMLファイルの配信に、Amazon S3を利用する必然性はありません。EC2にApacheなどをインストールすることで構成したWebサーバーや、AWS以外のレンタルサーバーなどに配置してもかまいません。

●API GatewayとLambda関数

入力フォームでPOSTしたときに実行されるサーバー側のプログラムは、API GatewayとLambda関数で構成します。

- ・Lambda関数では、コンテンツへアクセス可能なダウンロード用URL（後述の署名付きURL）を作成します。
- ・そして入力された「氏名」「メールアドレス」と「アクセス日時」「ホスト名」そして、作成した「URL」をデータベースとなるDynamoDBへ登録します。
- ・入力されたメールアドレス宛には、作成したURLを含む文面を送信することで、ユーザーにダウンロード先のURLを伝えます。

●ダウンロード用URL

ユーザーに通知するダウンロード用のURLは、以下のような仕様上の制限を付けることにより、掲示板などでURLが公開されてしまったときの被害を小さくするように対策を施します。

- ・ユーザーごとに別のものとする。
- ・有効期限を付けて48時間以内しかダウンロードできないようにする。

これらの条件を満たすURLを作成するには、S3の署名付きURLという機能を使います。また、メールを送信するには、Amazon SESを用います。

5-1-1 サービスの使用権限

本章のシステムでは、「API Gateway」「Lambda関数」「DynamoDB」「署名付きURL」「Amazon SESを使ったメール送信」を使いますが、これらを使うには、適切な権限が必要です（図5-2）。

● 5-1 API Gateway のイベント事例

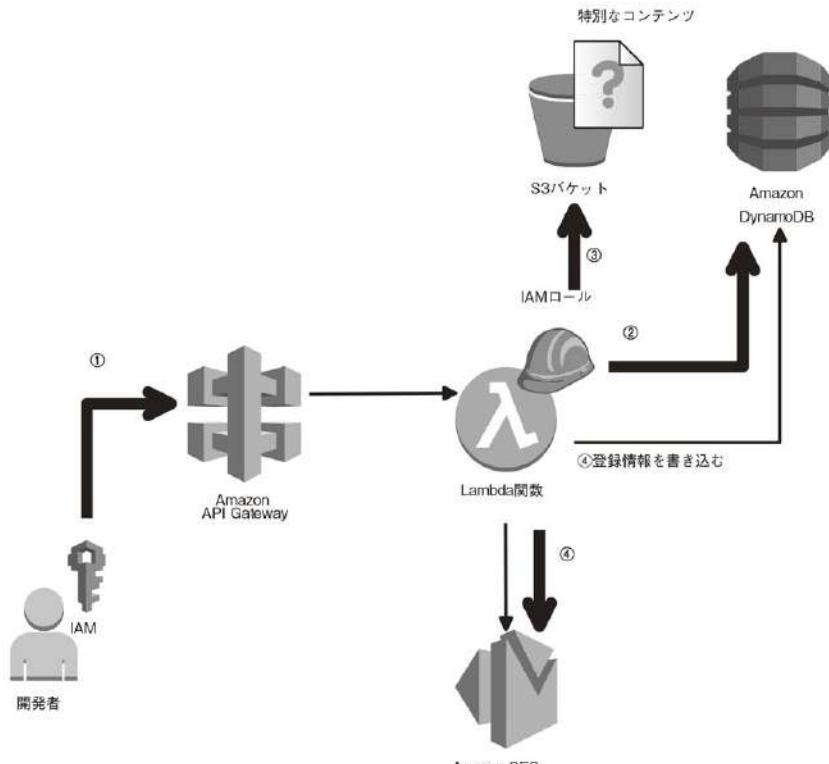


図 5-2 必要な権限

各種サービスの操作に必要な権限は、都度必要に応じて設定していきますが、はじめに、どのような権限が必要となるのか、その概要を説明しておきます。

● API Gateway の作成

開発者が API Gateway を作成するには適切な権限が必要です（図 5-2①）。作成やデプロイなどで権限が異なるため、ひとつずつ権限を設定するよりも、`AmazonAPIGatewayAdministrator` という管理ポリシーを適用するのが簡単です。

● Lambda 関数から DynamoDB へのアクセス権限

Lambda 関数から DynamoDB にアクセスするには、読み込み、書き込みなど操作に応じた適切な権限が必要です（図 5-2②）。

ひとつずつ設定するのが望ましい方法ですが、本章では話を簡単にするために、Lambda の実行ロールに `AmazonDynamoDBFullAccess` という管理ポリシーを適用するものとします。このポリシーを適用すると、DynamoDB にフルアクセス可能になります。



Memo DynamoDBへのアクセス権限

AmazonDynamoDBFullAccess は、DynamoDBに関するすべての操作を許可するので、本番環境での使用は避けたほうがよいでしょう。本番環境では、少し面倒になりますが、テーブル単位で読み書きを細かく設定するのが、セキュリティ上、望ましい方法です。

● Lambda 関数から署名付き URL を作成する

Lambda 関数から S3 の署名付き URL を作成するには、S3 バケットに対して、適切な権限が必要です（図 5-2③）。S3 に対するフルアクセスを許可する AmazonS3FullAccess 管理ポリシーを適用していれば、この要件を満たせます。

● Lambda 関数から Amazon SES を利用するための権限

Amazon SES を利用してメールを送信するには、適切な権限が必要です（図 5-2④）。本章では、Lambda の実行ロールに AmazonSESFullAccess という管理ポリシーを適用することで、この要件を満たすようにします。

5-1-2 Web アプリケーションの作成手順

本章は、「API Gateway」「Lambda 関数」「DynamoDB」「署名付き URL」「Amazon SES を使ったメール送信」と、広範囲にわたる説明を行うため、次の 4 段階に分けて Web アプリケーションの作成手順を追いながら説明していきます。

① API Gateway と Lambda の基本

まずは、「5-2 API Gateway と Lambda 関数を組み合わせる」と「5-3 API Gateway から実行される Lambda 関数を作る」の 2 つの節にて、API Gateway と Lambda を組み合わせて、HTTPS プロトコルから Lambda 関数を呼び出せるように構成します。

実際の手順としては、Web フォームを作成し、氏名やメールアドレスを入力して POST し、それを読み取って、CloudWatch Logs に書き込むというところまでを確認します。

② DynamoDB の操作

次に、「5-4 DynamoDB の基本」「5-5 Lambda 関数で DynamoDB にアクセスする」の 2 つの節にて、①で受け取ったデータを DynamoDB に書き込む方法を説明します。

● 5-2 API Gateway と Lambda 関数を組み合わせる

③署名付き URL

そして、「5-6 署名付き URL を発行する」では、署名付き URL を発行して、それも合わせて②の DynamoDB に保存するようにプログラムを改良します。

④メールの送信

最後に、「5-7 メールの送信」では、Amazon SES を使って③の URL を含む文面をメール送信する仕組みを作ることで、完成とします。

5-2 API Gateway と Lambda 関数を組み合わせる

Lambda 関数を HTTPS プロトコル経由で呼び出せるようにする仕組みが、API Gateway です^{*1}。この仕組みにより、クライアントサイドの Web フォームや JavaScript からの呼び出し、他システムと HTTPS プロトコルを使った連携などが可能となります。ひと言で言えば、これまで EC2 インスタンスなどの上で Perl や PHP、Java などで作成していた「サーバーサイドプログラム」を、Lambda 関数を使って実装できるようにする仕組みです（図 5-3）。なお、API Gateway は、Lambda を「同期的」に実行します。

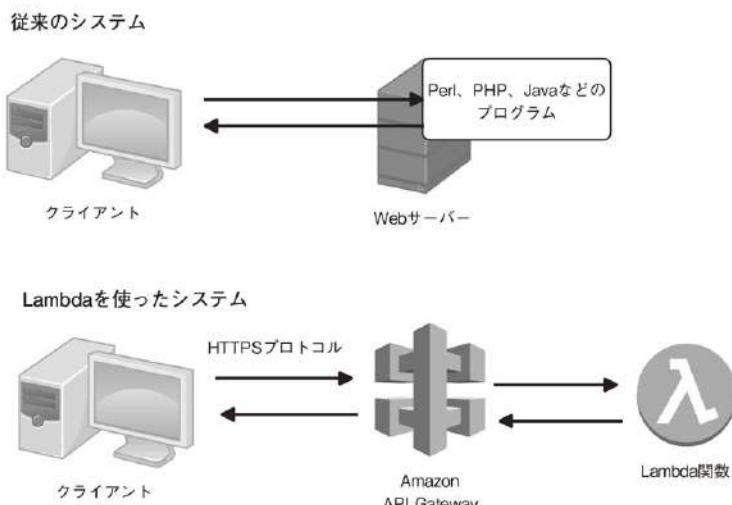


図 5-3 サーバーサイドプログラムを API Gateway + Lambda で実現する

* 1 API Gateway では、Lambda 関数以外にも、EC2 インスタンスで作成したワークロード、他の HTTP システムなどを呼び出すこともできます。

5-2-1 HTTPSプロトコルとLambda関数とのデータマッピング

図5-3の構成図からわかるように、API Gatewayは、一種のプロトコル変換器です。HTTPSプロトコルで流れてきたデータを受け取り、それをLambda関数に渡し、Lambda関数からの戻り値を、HTTPSプロトコルに変換してクライアントに返します。

そう考えると、HTTPSプロトコルのデータ項目とLambda関数のイベント引数や戻り値とを、どのようにマッピングするかを決めることが必要になることがわかります。

API Gatewayでは、次の2つの方法でマッピング方法を決めることができます。

①マッピングテンプレートを使う方法

ひとつめの方法は、API Gatewayの設定画面から、「HTTPSプロトコルのリクエストのどの項目をイベント引数のどの項目に設定するのか」、そして、「Lambda関数からの戻り値をHTTPSプロトコルのどのレスポンス項目に設定するのか」を、ひとつひとつマッピングテンプレートとして定義する方法です。

②Lambdaプロキシ統合を使う方法

もうひとつの方法は、Lambdaプロキシ統合という機能を使う方法です。この機能をオンにすると、ひとつひとつマッピングしなくとも、自動で設定されます。

多くの場合、②のほうが簡単です。そして、Lambda関数を設計図(blue-print)から作成する場合も②の方法がとられます。本書でも、API GatewayからLambda関数を呼び出すデータのマッピングは、②の方法をとることにします。

具体的に、どのようにマッピングされるのかについては、「5-3-4 イベント引数に渡される値(p.139)」で説明します。

Λ Column Lambdaプロキシ統合の制限

Lambdaプロキシ統合を使う場合には、「重複したクエリ文字列」「重複したヘッダ」がサポートされない、そして、「HTTPのHostヘッダが転送されない」という既知の問題があります。

重複したクエリ文字列とは、たとえば、「…/apiname?foo=bar&foo=bar2」のように、クエリ文字列「foo」が2回以上、登場することです。もし、こうしたクエリ文字列を使いたいのなら、Lambdaプロキシ統合の利用を諦めなければなりません。Lambdaの制限事項について

ては、以下に示す URL に記載されています。

[既知の問題]

http://docs.aws.amazon.com/ja_jp/apigateway/latest/developerguide/api-gateway-known-issues.html

5-3 API Gateway から実行される Lambda 関数を作る

では、API Gateway と組み合わせた Lambda 関数を作っていくましょう。

以下の操作では、次のことを行います。

① API Gateway と組み合わせた Lambda 関数を作る

設計図（blue-print）から作成する方法で、API Gateway と Lambda 関数をまとめて作ります。作成する Lambda 関数には、入力フォームから受け取ったデータを、そのまま print 関数で書き出す、ごく簡単なコードを記述します。そうすることで、受け取ったデータは、どのような構造で、どのような情報が含まれているのかを見ていきます。

② 入力フォームを用意する

①の Lambda 関数を呼び出すための入力フォームを作ります。入力フォームを構成する HTML ファイルは、Static website hosting（静的ウェブホスティング）を有効にした S3 バケットに配置することにします。

5-3-1 API Gateway を作成するための権限

API Gateway を作成するには、適切な権限が必要です。開発者に対して、「AWSLambdaFullAccess ポリシー」に加え「AmazonAPIGatewayAdministrator ポリシー」の適用が必要です。まずは、開発者アカウントに対して、AmazonAPIGatewayAdministrator ポリシーを適用します。この手順については、Appendix A を参照してください。ポリシーの適用が済んだら、Lambda 関数の作成に進んでください。

5-3-2 API Gatewayから実行されるLambda関数を作る

API Gatewayを作成する準備ができたら、実際に、API Gatewayから実行されるLambda関数を作っていくましょう。どのような機能を持つLambda関数を作る場合でも、Lambdaの「設計図(blue-print)」を参照し、これを改良していくのが簡単です。ここでは、API Gatewayから呼び出されることを想定とした設計図から、Lambda関数を作成します。

◎ 操作手順 ◎ API Gatewayから実行されるLambda関数を作る

[1] Lambda関数を作成する

- ・Lambdaコンソールを開き、[関数の作成]をクリックし、新しいLambda関数を作成します(図5-4)。



図5-4 新しいLambda関数を作成する

[2] microservice-http-endpointから作成する

- ・API Gatewayと組み合わせるときの設計図は、microservice-http-endpointです。本書ではPython3を使うので、microservice-http-endpoint-python3を設計図として選んで、次に進んでください(図5-5)。この設計図からLambda関数を作成すると、「API Gateway」と「Lambda関数」の2つと一緒に作成されます。そして、作られるAPI Gatewayには、Lambdaプロキシ統合が設定されます。

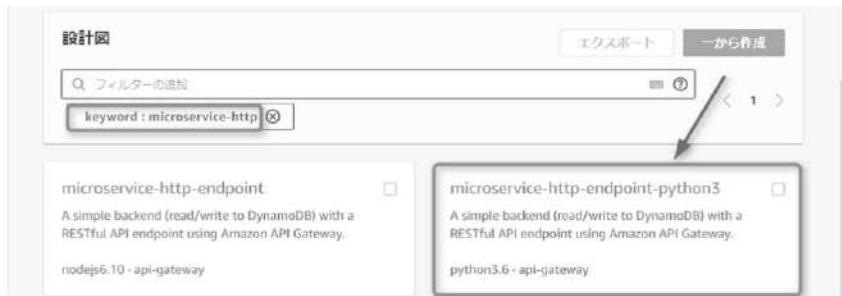


図5-5 microservice-http-endpoint-python3を選択する

[3] トリガーの設定

トリガーとして API Gateway が自動的に設定されます。

- この画面では、「API名」と「デプロイされるステージ」、「セキュリティ」の3つの項目を決め、[次へ] をクリックしてください（図5-6）。「API名」と「デプロイされるステージ」はリストボックスですが、[値の入力] ボタンをクリックすることで、自由入力できるようになります。



図 5-6 トリガーの設定

①API名

任意の API 名です。API を呼び出すときの URL の一部になります。ここでは仮に「UserRegistAPI」という名前にします。

②デプロイされるステージ

API Gateway には、公開する API を「ステージ」という機能を使って、切り替えることができます。たとえば本番用のものを「prod」(product の意味)、開発用のものを「dev」(development の意味)など、名前を付けて公開することで、これらを切り替えて使えます。本書では、そうした運用の話までは踏み込まないので、デフォルトの「prod」のまま進めてください。

(3) セキュリティ

このAPIを呼び出せる権限を選択します。次の3種類から選べます。ここで作ろうとしているWebシステムなど、不特定多数のユーザーから呼び出されるプログラムを作るときは認証する必要がないので、「オープン」を選択します。

(1) AWS IAM

IAMに基づいたセキュリティで認証します。

(2) オープン

認証しません。誰でも呼び出せます。

(3) アクセスキー使用でのオープン

IAMコンソールからアクセス用の「アクセスキー」を発行し、そのアクセスキーを使って認証します。

[4] Lambda関数の設定

Lambda関数の名前やコードを設定します(図5-7)。

- ・コードはのちに修正することにして、ここでは「関数の名前」を設定します。どのような名前でもよいのですが、ユーザー登録する機能を作っていくという意味で、registという名前にします。この名称は、呼び出すときのURL(後述するエンドポイント)の一部になります。
- ・「説明」には、適当な説明文を入力します。ここでは「ユーザー登録」としておきます。
- ・「ランタイム」には実行するランタイムを指定します。「Python 3.6」が設定されているはずなので、そのままにします。



図5-7 Lambda関数の設定

引き続き、Lambda関数のコードを記述します(図5-8)。このコードは、あとで変更するので、そのままにしておきます。

● 5-3 API Gateway から実行される Lambda 関数を作る

サービス リソースグループ ★

Amazon Taro 東京 サポート

Lambda 関数のコード

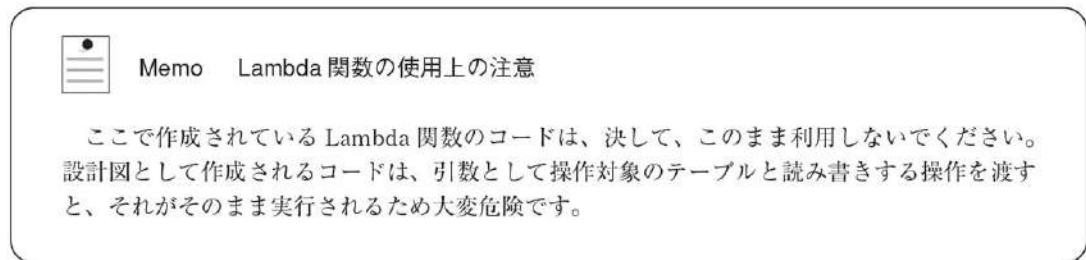
関数のコードを指定します。コードに(boto3以外の)カスタムライブラリが必要でない場合は、エディタを使用します。カスタムライブラリが必要な場合は、コードとライブラリを ZIP ファイルとしてアップロードすることができます。

コードエントリータイプ

コードをインラインで編集

```
1 import boto3
2 import json
3
4 print('Loading function')
5 dynamo = boto3.client('dynamodb')
6
7
8
9 def respond(err, res=None):
10     return {
11         'statusCode': '400' if err else '200',
12         'body': err.message if err else json.dumps(res),
13         'headers': {
14             'Content-Type': 'application/json',
15         },
16     }
17
18
19 def lambda_handler(event, context):
20     '''Demonstrates a simple HTTP endpoint using API Gateway. You have full
21     access to the request and response payload, including headers and
22     status code.'''
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
```

図 5-8 Lambda 関数は、デフォルトのままにしておく



(5) Lambda 関数ハンドラおよびロール

Lambda 関数ハンドラとロールを設定します（図 5-9）。

- ・ハンドラはデフォルトのままにしておきます（デフォルトは「lambda_function.lambda_handler」なので、以降の手順では、ソースコードに記載する Lambda 関数のエントリ関数名は「lambda_handler」という名前にするということです）。
 - ・ロールでは、【既存のロールを選択】を選択して、本書でずっと使ってきた「role-lambdaexec」（Appendix B を参照）を選択します。
 - ・タグと詳細はデフォルト設定のままにし、【次へ】ボタンをクリックして、次に進んでください。



図 5-9 Lambda 関数ハンドラおよびロールの設定



Memo メモリと実行時間の調整

【詳細設定】を展開するとわかりますが、microservice-http-endpoint-python3 の設計図から作成した場合、メモリが 512MB、実行最大時間が 10 秒に設定されます。ほとんどの場合、これで問題ないはずですが、必要に応じて調整してください。

[6] 確認

- 確認画面が表示されます（図 5-10）。【関数の作成】ボタンをクリックします。



図 5-10 確認画面

5-3-3 API を設計する

では、このひな形を修正して、目的のシステムを作成していきます。最初に、本章で作成するシステムを、どのような動きにするのかを決めます。

本章のサンプルでは、氏名およびメールアドレスを、次のような入力フォームで実装するものとします（図 5-11）。HTML で示すと、次の通りです。

```
<form method="POST" action="API Gateway の呼び出し先URL(後述)">
  氏名 : <input type="text" name="username"><br>
  メールアドレス : <input type="text" name="email">
  <input type="submit" value="登録">
</form>
```

図 5-11 想定する入力フォーム

5-3-4 イベント引数に渡される値

こうした Web フォームの Submit ボタン（この例では【登録】ボタン）がクリックされたときには、「氏名」や「メールアドレス」に入力したデータは、Lambda 関数からは、どのようにして取得できるのでしょうか？

■イベント引数の構造

これらの情報は、もちろん、Lambda 関数が呼び出されるときの「イベント引数」に含まれています。イベント引数に渡されるデータの内容は、API Gateway の「Lambda プロキシ統合」という機能が有効であるかどうかによって異なります。microservice-http-endpoint の設計図から作成した場合は、Lambda プロキシ統合が有効に設定されています。この場合、イベント引数は、次の書式をとります。各イベント引数の意味は、表 5-1 を参照してください。

```
{
  "resource": リソースのパス,
  "path": URL のパス,
  "httpMethod": HTTP メソッドの種類
  "headers": ヘッダ情報
```

```

    "queryStringParameters": クエリパラメータ情報
    "pathParameters": 拡張パス情報
    "stageVariables": ステージング名
    "requestContext": リクエストコンテキスト
    "body": 送信されたボディ部
    "isBase64Encoded": bodyがBase64エンコードされているかどうか
}

```

表5-1 API Gatewayから呼び出されたときに設定されるイベント引数の内容

項目	意味
resource	呼び出し元のリソース名。Lambdaプロキシ統合から呼び出される場合、「/{proxy+}」という文字列
path	URLのパス
httpMethod	HTTPのメソッド。「GET」「HEAD」「POST」など
headers	クライアントから送信されたHTTPヘッダのリスト
queryStringParameters	URLの末尾（「?」以降）に付けられたクエリパラメータのリスト
pathParameters	URLの末尾に付けられた拡張パス名
stageVariables	ステージに設定された変数の値群
requestContext	クライアントのリクエストに関するコンテキスト情報。identifyのなかにクライアントの詳細情報が格納されており、たとえば、送信元IP（sourceIP）やブラウザの種類（userAgent）などが含まれている（表5-2を参照）
body	クライアントから送信されたボディ部のデータ
isBase64Encoded	bodyがBase64エンコードされている場合はtrue、そうでなければfalse。通常はfalse

■フォームにポストされたデータ

WebフォームがPOSTメソッドで呼び出される、すなわち「method="POST"」としているときには、フォームのデータはボディ部として送信されます。このデータは、イベント引数の「body」を通じて読み取れます。

```
<form method="POST" action="API Gatewayの呼び出し先URL（後述）">
```

ちなみに、今回は使いませんが、GETメソッドで送信する場合には、queryStringParametersに、その情報が設定されます。

```
<form method="GET" action="API Gatewayの呼び出し先URL（後述）">
```

■クライアントに関する情報

クライアントの IP アドレスやユーザーエージェントなどの情報は、イベント引数の「requestContext」の「identify」という項目で取得できます（表 5-2）。

表 5-2 requestContext の identify

項目名	意味
cognitoIdentityPoolId	Cognito(※) から呼び出されたときのプール ID
accountId	アカウント ID
cognitoIdentityId	Cognito から呼び出されたときの ID
caller	呼び出し元の情報
apiKey	API キー情報
sourceIp	クライアントの IP アドレス
cognitoAuthenticationType	Cognito の認証タイプ
cognitoAuthenticationProvider	Cognito の認証プロバイダ
userArn	ユーザーの ARN
userAgent	ユーザーエージェント情報
user	ユーザー情報

※ モバイルアプリやウェブアプリに認証機能を提供するサービス。

5-3-5 戻り値の規定

Lambda プロキシ統合を有効にしている場合、Lambda 関数の戻り値は、次の書式のデータでなければなりません。この書式に沿わないときは、クライアントには 502 エラーが返されてしまいます。ただし、「isBase64Encoded」は省略可能です。

```
{
  "isBase64Encoded": bodyがBase64エンコードされているときはtrue、そうでなければfalse
  "statusCode": HTTPステータスコード,
  "headers": { クライアントに返したいヘッダ情報 ... },
  "body": クライアントに返したいボディ情報
}
```

たとえば、単純な HTML を返す場合には、次のような戻り値とします。

```
return {
  'statusCode' : 200,
  'headers' : {
    'content-type' : 'text/html'
```

```

},
'body' : '<html><body>OK</body></html>',
}

```

5-3-6 イベント引数の内容を確認するプログラムを作る

これだけの説明だと少しあわかりにくいので、実際にイベント引数に、どのような値が渡されるのかを表示するプログラムを作成してみます。具体的には、イベント引数の値を JSON 形式で print することで CloudWatch Logs に書き出し、その値を表示するプログラムを作ります。そのプログラムはリスト 5-1 のようになります。

リスト 5-1 イベント引数をログに書き出すプログラム

```

import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=4))
    # JSON形式の戻り値を設定する
    return {
        'statusCode' : 200,
        'headers' : [
            'content-type' : 'text/html'
        ],
        'body' : '<html><body>OK</body></html>'
    }

```



図 5-12 リスト 5-1 を入力したところ

正常に実行されれば、クライアントには、「<html><body>OK</body></html>」という HTML 文

● 5-3 API Gateway から実行される Lambda 関数を作る

字列が送られるので、画面には「OK」と表示されるはずです。

5-3-7 API Gateway 経由で実行する

API Gateway と結び付けた Lambda 関数は、[トリガー] タブで、呼び出し先の URL を確認できます（図 5-13）。呼び出し先の URL のことをエンドポイントと言います。



図 5-13 エンドポイントを確認する

実際に Web ブラウザなどでこのエンドポイントを呼び出すと、いま作成した Lambda 関数が呼び出されます。実際に実行すると、画面には「OK」と表示されることがわかります（図 5-14）。



図 5-14 ブラウザでエンドポイントを呼び出したところ

リスト 5-1 では、次のように、イベント引数を JSON 化して print しています。この結果は、CloudWatch Logs に書き込まれているはずです。

```
print(json.dumps(event, indent=4))
```

実際に CloudWatch Logs を確認すると、次の書式のデータであることがわかります。

```
{
  "resource": "/regist",
  "path": "/regist",
```

```
    "httpMethod": "GET",
    "headers": {
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/web⇒
p,image/apng,*/*;q=0.8",
        "Accept-Encoding": "gzip, deflate, br",
        … 略（ブラウザから送信されたヘッダの値）…
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.3⇒
6 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
    },
    "queryStringParameters": null,
    "pathParameters": null,
    "stageVariables": null,
    "requestContext": {
        "path": "/prod/regist",
        "accountId": "255574205617",
        "resourceId": "hvjxv4",
        "stage": "prod",
        "requestId": "a04f23fa-78a4-11e7-b977-b1d15124aff3",
        "identity": {
            "cognitoIdentityPoolId": null,
            "accountId": null,
            "cognitoIdentityId": null,
            "caller": null,
            "apiKey": "",
            "sourceIp": "116.0.244.66",
            "accessKey": null,
            "cognitoAuthenticationType": null,
            "cognitoAuthenticationProvider": null,
            "userArn": null,
            "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
            "user": null
        },
        "resourcePath": "/regist",
        "httpMethod": "GET",
        "apiId": "1up3pu8ctj"
    },
    "body": null,
    "isBase64Encoded": false
}
```

この結果からわかるように、クライアントのIPアドレスは、`event['requestContext']['identity']['sourceIp']`でわかります。

● 5-3 API Gateway から実行される Lambda 関数を作る



図 5-15 CloudWatch Logs で確認したところ

5-3-8 S3 に Web フォームを配置する

次に、Web フォームからこの Lambda 関数を呼び出した場合は、どのようになるのか、確かめてみましょう。

■ Web フォームを作る

まずは、Web フォームを `index.html` という名前で作ります（実際には、どのような名前でもかまいません）。

form 要素の POST 先となる action 属性を、図 5-13 で確認した API Gateway のエンドポイントになるように設定します。

```
<form method="POST" action="図5-13で確認したエンドポイント">
```

リスト 5-2 入力フォームとなる Web フォーム

```
<!DOCTYPE html>
<html lang="ja">
<head>
<meta charset="UTF-8">
</head>
<body>
<form method="POST" action="https://XXXXXXXXXX.execute-api.ap-northeast-1.amazonaws.com/prod/register">
    氏名 :<input type="text" name="username"><br>
    メールアドレス :<input type="text" name="email">

```

```
<input type="submit" value="登録">  
</form>  
</body>  
</html>
```

■ Static website hosting を有効にした S3 バケットを作成する

この Web フォームを適当な Web サーバーに配置して、アクセスできるようにします。どこでもよいのですが、ここでは、「Static website hosting（静的 Web サイトホスティング）」機能を有効にした S3 に配置することでアクセスできるようにします。まずは、適当な S3 バケットを作り、それから Static website hosting 機能を有効にします。

◎ 操作手順 ◎ S3 バケットを作り、静的 Web サーバー機能を有効にする

[1] S3 バケットを作り始める

- ・S3 コンソールを開き、[バケットを作成する] をクリックして、S3 バケットを作り始めます（図 5-16）。



図 5-16 S3 バケットを作り始める

[2] バケット名とリージョンの選択

- ・適当なバケット名を入力し、リージョンを選択します。ここでは、webexample000 というバケット名で東京リージョンに作成しました。バケット名は任意の名前でかまいませんが、他のユーザーと重複しない名前を付けてください（図 5-17）。

● 5-3 API Gateway から実行される Lambda 関数を作る



図 5-17 バケット名とリージョンの選択

【3】オプションの設定

- 各種オプションを設定します。ここではとくに必要ないので、そのまま [次へ] をクリックしてください(図 5-18)。



図 5-18 オプションの設定

なお、実運用する場合は、ログを有効にして、アクセスログを記録できるようにしておくといいでしょう。

[4] Static website hosting を有効にする

- 以上の手順で、S3 バケットが作成されます。作成されたら [プロパティ] タブを開き、[Static website hosting] の機能を有効にしてください（図 5-19）。

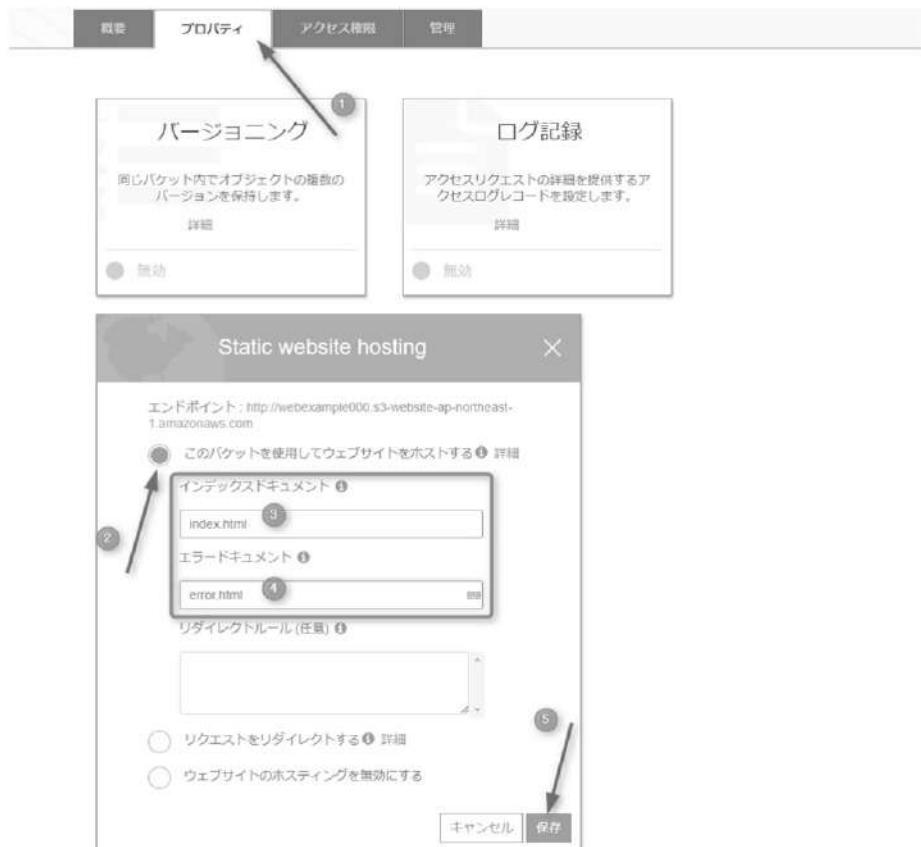


図 5-19 Static web hosting の機能を有効にする

このとき、インデックスドキュメントとエラードキュメントを入力する必要があります。どのような名称でもよいのですが、ここでは前者を「index.html」、後者を「error.html」としました。

このように設定すると、ユーザーがパス名までだけでアクセスした場合は index.html が自動的に参照されるようになります。またエラーが発生したときは、error.html が表示されるようになります。

● 5-3 API Gateway から実行される Lambda 関数を作る

■ index.html をアップロードする

S3 バケットの作成が完了したら、index.html をアップロードします。

◎ 操作手順 ◎ index.html をアップロードする

[1] アップロードを始める

- ・作成した S3 バケットを開き、[アップロード] ボタンをクリックします（図 5-20）。



図 5-20 アップロードを始める

[2] ファイルを追加する

- ・アップロード画面が表示されます。ここに index.html をドラッグ & ドロップして登録します。登録したら [次へ] をクリックします（図 5-21）。



図 5-21 index.html を追加する

[3] アクセス権を設定する

- ・誰でもアクセスできるようにしたいので、[このオブジェクトにパブリック読み取りアクセス権限を付与する]を選択し、[次へ]をクリックしてください（図5-22）。



図5-22 読み取り権限を与える

[4] ストレージクラスや暗号化、メタデータの設定

- ・ここでは、これらの値は設定しないので、そのまま[次へ]をクリックしてください（図5-23）。



図5-23 ストレージクラスや暗号化、メタデータの設定

● 5-3 API Gateway から実行される Lambda 関数を作る

[5] アップロードする

- 最後に [アップロード] をクリックするとアップロードされます（図 5-24）。



図 5-24 アップロードする

■ Web フォームから POST してみる

Static website hosting を有効にすると、「エンドポイント」と書かれている URL で、配置したコンテンツが表示されます。プロパティ画面からエンドポイントを確認しましょう（図 5-25）。

実際に Web ブラウザでアクセスすると、図 5-26 のような入力フォームが表示されるはずです。



図 5-25 エンドポイントを確認する

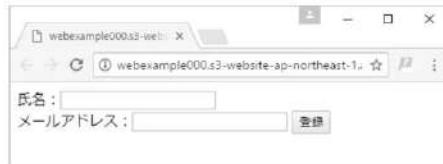


図 5-26 作成した入力フォーム

たとえばここで、氏名に「山田太郎」、メールアドレスに「yamada@example.co.jp」と入力して【登録】ボタンをクリックします（図 5-27）。すると Lambda 関数が実行され、リスト 5-1 の結果、前掲の図 5-14 と同様に、画面には「OK」と表示されます。



図 5-27 Web フォームから POST してみる

このとき、CloudWatch Logs の画面を表示して、Web フォームから入力したデータが、イベント引数では、どのように見えるのかを確認しましょう。body のところに URL エンコードされた形式で格納されていることがわかります（図 5-28）。

図 5-28 CloudWatch Logs で POST したデータの構造を確認する

```
"body": "username=%E5%B1%B1%E7%94%B0%E5%A4%AA%E9%83%8E&email=yamada%40example.co.jp"
```

つまりこの部分をパースすれば、入力フォームに入力された値を取り出せるということになります。

5-4 DynamoDB の基本

次に、データベースに書き込む仕組みを作ります。DynamoDB は NoSQL のデータベースということも相まって、その扱いが少し特殊です。とくに SQL データベースに慣れた人にとっては、考え方を改めなければならないところもあります。そこでまず、DynamoDB の特徴や扱う上での注意点を説明します。そのあと、本章のアプリケーションのデータを保存するためのテーブルを作成していきましょう。

5-4-1 DynamoDB の特徴

DynamoDB は、キーバリューストア型の NoSQL のデータベースです。AWS のデータベースサービスとして、次のような特徴があります。

①完全マネージド型のサービス

完全なマネージドサービスとして提供されており、運用保守の手間がありません。テーブルを作成するだけで、すぐに使い始められます。

②非 VPC 接続環境での利用

利用するのに VPC 環境を必要としません。Lambda 関数はデフォルトでは非 VPC 環境で動くので、Lambda とは親和性の高いデータベースであると言えます。

③データベースサイズの自動拡張

事前にデータベースのサイズを決める必要はありません。保存するデータ量が多くなれば、自動的にサイズが拡張されます。

④負荷に応じたオートスケーリング

のちに詳しく説明しますが、DynamoDB の性能は、「1 秒間に、読み込みや書き込みのリクエストをどれだけ処理できるか」の設定で決まります。処理能力を高く設定するほど、コストがかかりますが、DynamoDB はオートスケーリングに対応しています。あらかじめ指定した範囲で、負荷に応じて処理できる秒間リクエスト数を動的に変更すれば、コストも相応になります。

⑤高い信頼性

DynamoDB は、複数のアベイラビリティゾーンで運用されます。そのため、一部のアベイラビリティゾーンに障害が生じても処理し続けることができ、高い信頼性を誇ります。



Memo アベイラビリティゾーン

アベイラビリティゾーンとは、AWSにおけるデータセンターの拠点のことです。

⑤トランザクション機能はサポートされない

上記①～④はメリットですが、デメリットもあります。最大のデメリットは、トランザクション機能がサポートされないという点です。書き込み処理に失敗したときの操作を自前で作り込んだり、トランザクション処理用のライブラリを使ったりすれば、擬似的なトランザクション処理を実現することはできますが、機能として提供されているわけではありません。

■スキーマレスのデータベース

DynamoDBは、キーバリューストア型のNoSQLデータベースであるため、一般的なSQLデータベースと扱い方が少し異なります。

とくに大きく違うのは、事前にテーブルに対してスキーマを定義しなくてよいという点です。一般にSQLデータベースは、テーブルに対して、どのような型でどのような名前のカラム(列、フィールド)を使うのかをスキーマとして定義します。スキーマに合わない書式のデータは保存できません。それに対してDynamoDBでは、テーブルに対してプライマリーキーという、SQLデータベースにおける主キーに相当する名前とデータ型を定めるだけでよく、残りの列は、格納時に好きなものを格納できます。

図5-29に示すように、レコードごとにカラムの数や種類が変わってもかまわないので、表構造として考えると設定されていない値があり、表が歯抜けのような状態になります。このような特性は、表構造で考えるのではなく、プライマリーキーに対して任意のデータを結び付ける構造をとっていると考えたほうがわかりやすいでしょう。

このように表構造をとるわけではないため、DynamoDBでは、「列」や「レコード」という用語は使わず、それぞれ、属性(アトリビュート)や項目(アイテム)と呼びます。それぞれの属性の最大サイズは、400KBです。

■プライマリーキー

DynamoDBにおいて、プライマリーキーは、項目を特定する値となります。

单一のキーを指定する以外に、追加で「ソートキー」を指定することもできます。

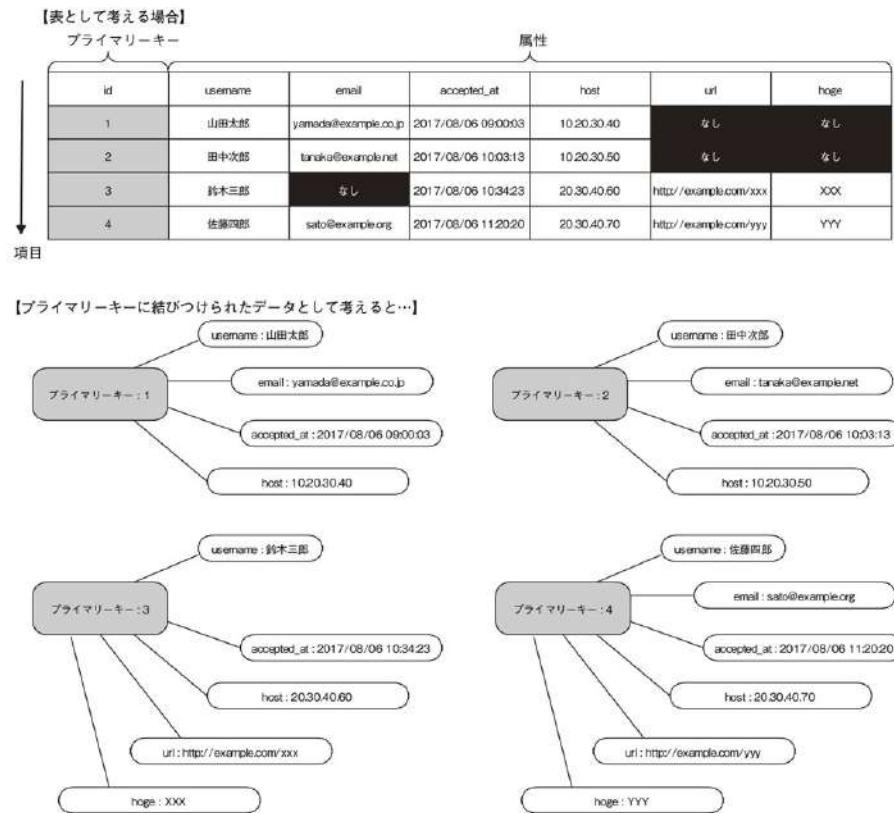


図 5-29 DynamoDB はスキーマレス

●パーティションキー

ひとつの単一の属性で項目を区別するとき、このキーをパーティションキーと言います。パーティションキーは、全項目でユニークな値である必要があります。

●パーティションとソートキー

パーティションキーに加え、もうひとつ、ソートキーを組み合わせて、項目を特定する方法です。ユニークであることが必要なのは、パーティションキーとソートキーの組み合わせです。パーティションキー自体、もしくは、ソートキー自体の重複は許されます。

■ DynamoDB のデータ型

DynamoDB では、表 5-3 に示すデータ型を利用できます。単一値か複数値かの違いがありますが、大雑把に言うと、「数値」「文字列」「バイナリ」の 3 種類しかありません。

数値型において、整数と浮動小数の違いはありません。なお、Python の Float 型は Decimal 型に

変換しないと DynamoDB に書き込めません。

また日付時刻型がないので、日付や時刻情報を扱う場合は、文字列として保存するか、タイムスタンプで表現して数値型として保存するなどします。大小比較するならタイムスタンプ値が処理しやすいと思いますが、もし、文字列として保存するときは、ISO 8601 形式にするのがお勧めです。ISO 8601 形式だと、タイムゾーンを UTC などに揃えておけば、文字列としての大小と日付時刻の大小が一致するので、並べ替えや範囲設定などが容易になるからです。

表 5-3 データ型

データ型	意味
Number	数値
String	文字列
Binary	バイナリ
NumberSet	数値の集合（順不同）
StringSet	文字列の集合（順不同）
BinarySet	バイナリの集合（順不同）
List	リスト型（配列。順序は保障される）
Map	マップ型（連想配列。キーと値のセット。順不同）
Boolean	true か false かの値
Null	Null 値

5-4-2 結果整合性、スループット、価格

DynamoDB は、あらかじめ、読み込みや書き込みの性能がどれだけなのかをテーブル単位で定めて運用します。もし、定めた性能を越えたアクセスがあると、それらのアクセスが失敗する可能性があります。

■キャパシティーユニットと価格

DynamoDB の性能は、「読み込みキャパシティーユニット」と「書き込みキャパシティーユニット」という 2 つの設定値で決まります。これらはテーブルに対する設定値であり、秒間の処理能力を示します。

DynamoDB では、「読み込みを 4KB ずつ」「書き込みを 1KB ずつ」処理します。この処理単位を「1」としたものが「キャパシティーユニット」です。言い換えると、「読み込みキャパシティーユニットが 1」とは「4KB 分読み込む能力」、「書き込みキャパシティーユニットが 1」とは「1KB

分書き込む能力」です。

仮に、テーブルに対して「読み込みキャパシティーユニットを 1」に設定すると、そのテーブルに対して、1 秒間に 4KB 以上の読み込みアクセスをすると、失敗する可能性があります。書き込みについても同様に、「書き込みキャパシティーユニットを 1」に設定したなら、1 秒間に 1KB 以上の書き込みアクセスをすると、失敗する可能性があります。そのため、DynamoDB を利用する場合には、あらかじめ、秒間にどれだけの読み書きをするのかを予想して、その設定をしておく必要があります。この設定値は、固定の値にすることも、AutoScaling を有効にして「最小値」と「最大値」とを設定して、その範囲内で負荷に応じて変動させることもできます（図 5-30）。



図 5-30 キャパシティーユニットの設定例

読み込みキャパシティーユニットや書き込みキャパシティーユニットの設定は、コストにも影響します。DynamoDB のテーブル作成画面からは、「キャパシティー計算ツール」を起動して、およそのコストを算出できます（図 5-31）。



図 5-31 キャパシティー計算ツール

■結果整合性と強力な整合性

Dynamo DBは、複数のアベイラビリティゾーンで分散して運用されています。項目の更新があると、その更新が、すべてのアベイラビリティゾーンに対して書き込まれます。書き込み処理は、ほとんどの場合1秒以内で完了しますが、若干のタイムラグがあります。そのため、データを書き込んだ直後にデータを読み込むと、タイミングによっては書き込みが終わっておらず、データを書き込む前の古い値が取得されることがあります。

DynamoDBでは、データを書き込んだあと、その値が確実に読み込めることを保障するかどうかによって、次の2種類の読み込み方法がサポートされています。

①結果整合性のある読み込み

読み込んだときには、最新の書き込みが反映されていない可能性がある読み込みです。②に比べてパフォーマンスが高いのが特徴です。

②強力な整合性のある読み込み

読み込んだとき、最新の書き込みが反映されていることを保障する読み込みです。この方式で読み込む場合、アベイラビリティゾーン間のネットワークの遅延や停止があったときには、一時的に読み込みに失敗する可能性があります。

また読み込みキャパシティーユニットが、①に比べて倍必要となります。

①②のどちらの読み込み方式を使うのかは、APIを呼び出すときの引数で決まります。デフォルトは①の結果整合性のある読み込みです。

②の強力な整合性のある読み込みを使うと、データの確実性は高まりますが、パフォーマンスは低下します。また、読み込みキャパシティーユニットが①に比べて倍必要となるので、必要なコストも高くなりがちです。

■運用時にはキャパシティーユニットの設定値に注意

DynamoDBを使うときの難しさは、適切な読み込みキャパシティーユニットと書き込みキャパシティーユニットを設定することにあります。

これらを低く見積もって運用すると、アクセスが多くなったときにはエラーが頻繁する恐れがあります。もし、制限を超えてエラーになった場合は、「400 Bad Request」ならびに「ProvisionedThroughputExceededException」という例外が発生し、CloudWatch Logsに記録されます。逆に高く見積もると、コストが高くなります。

本書は開発を主としたものなので、運用にまでは踏み込みませんが、実際に運用する際には、

エラーが発生していないかを監視したり、AutoScaling を適切に設定してスケーリングしたりするなどの工夫が不可欠です。

5-4-3 ユーザー情報を作成するテーブルを作る

ここでは DynamoDB テーブルの作成例として、user という名前のテーブルを作成し、表 5-4 に示す属性に、各種の値を保存するものとします。

表 5-4 user テーブル

属性名	型	意味
id	Number	連番。プライマリキーとして設定する
username	String	ユーザーが入力した氏名
email	String	ユーザーが入力したメールアドレス
accepted_at	Number	受付日時の UNIX タイムスタンプ
host	String	クライアントのホスト名 (IP アドレス)
url	String	コンテンツをダウンロードする URL

◎ 操作手順 ◎ user テーブルを作成する

[1] DynamoDB コンソールを開く

- AWS マネジメントコンソールから、DynamoDB コンソールを開きます（図 5-32）。

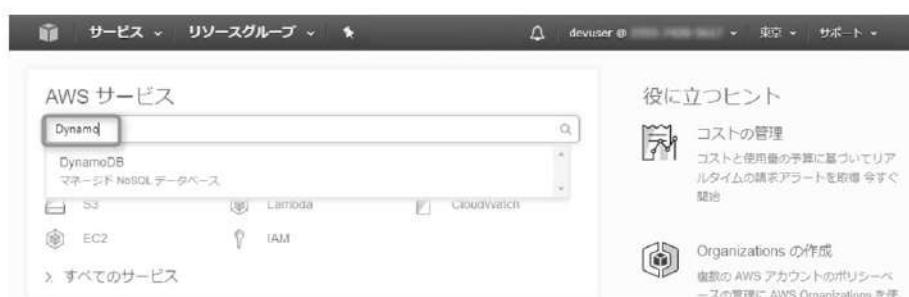


図 5-32 DynamoDB コンソールを開く

[2] テーブルを作成する

- 【テーブルの作成】をクリックして、テーブルの作成を始めます（図5-33）。



図5-33 テーブルを作成する

[3] user テーブルを作成する

- user テーブルを作成します。テーブル名には「user」と入力します。プライマリーキーは「id」という名前にして、数値型（これはNumber型のことです）を選択します。
- 本来なら、要求されるパフォーマンス性能とコストの兼ね合いで、読み込みキャパシティーユニットや書き込みキャパシティーユニットを調整すべきですが、ここでは【デフォルト設定の使用】にチェックを付け、デフォルトのままとします。
- 図5-34の【作成】ボタンをクリックすると、user テーブルが作成されます。すでに説明したように、DynamoDBはスキーマレスのデータベースです。そのため、表5-4に示したプライマリーキーとして設定した id 以外の属性は、この時点で定義する必要はありません。

The screenshot shows the 'Create Table' wizard for a 'user' table. It has two main sections: 'Table Creation' and 'Table Settings'.

Table Creation:

- Table Name: user
- Primary Key: id (Number type)
- Sort Key: None
- Default Setting Use: Selected (marked with a checkmark)

Table Settings:

- Default Setting Use: Selected (marked with a checkmark)
 - Scan indexing is disabled.
 - Auto Scaling capacity type is AnyScale (NEW!).
 - Estimated provisioned capacity: 70% target usage (minimum capacity is 5 reads and 5 writes).
- CloudWatch or Simple Notification Service notifications are disabled.

At the bottom are 'Cancel' and 'Create' buttons, with 'Create' being highlighted by an arrow.

図5-34 user テーブルを作成する

5-4-4 シーケンス番号を作るには

さて、user テーブルのプライマリーキーである id 属性には、追加されるたびに「1」「2」…のように連番を設定していきたいと思います。

■アトミックカウンタ

残念なことに DynamoDB には、「シーケンス番号」や「オートナンバー」などと呼ばれる自動的に連番を発生させる機能はありません。代わりの方法として、連番を管理する別のテーブルを作成しておき、「読み込むたびにカウントアップしていく」という手法をとります。

AWS の公式ドキュメントでも紹介されている方法が、「テーブル名」をキーにして、「シーケンス番号（連番）」を属性として設定するテーブルを用意する手法です。ここでもそれにならい、図 5-35 に示すように「sequence」というテーブルを作り、このテーブルで user テーブル用の連番を管理することにします。

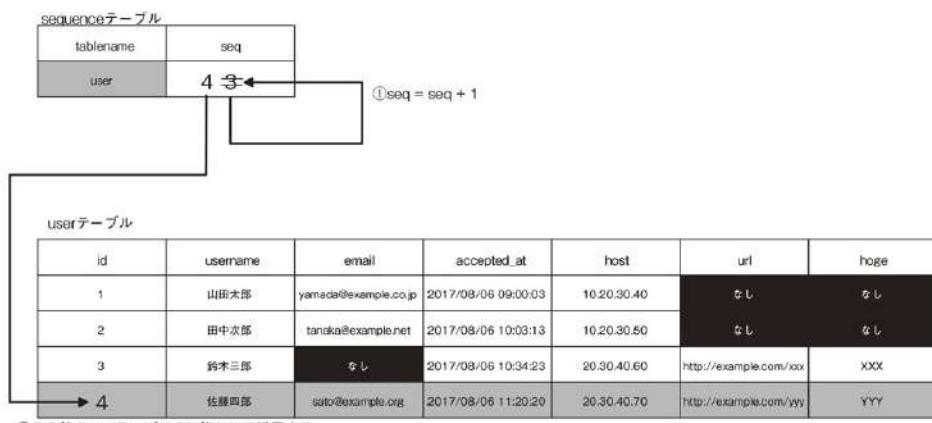


図 5-35 連番を管理するための sequence テーブル

このような手法で連番を作成する場合、同時に複数のアクセスがあっても、同じ連番値を返さないことが重要です。user テーブルの id 属性はプライマリーキーなので重複が許されないため、もし sequence テーブルからの連番で同じ値が返されてしまうと、user テーブルの登録に失敗してしまうからです。

すぐあとに説明しますが、DynamoDB には、計算式に基づいて Number 型の属性を更新し、更新した後の結果を返す手法があります。この「更新する→更新した結果を返す」の処理中に別の処理が割り込むことはないので、この機能を用いて連番を更新すれば、同じ連番が発生してしまうことはありません。この手法を使って、連番などを管理することを、AWS 公式ドキュメントで

はアトミックカウンタと呼んでいます。

Λ Column 連続性は保障されない

アトミックカウンタの手法を使うと、user テーブルにレコードを追加する流れは、以下の2つの連続した処理となります。

- ① sequence テーブルの id を更新しつつ読み込んで連番を得る
- ② user テーブルに①の連番とともに、他の属性値を設定して書き込む

DynamoDBにはトランザクション機能がないので、②に失敗すると、①の連番だけが更新される結果になります。つまり、連番の連続性が保障されません。そのため、「アクセスされた回数を記録したい。しかも誤差が寸分たりとも許されない」というときは、「条件付き更新」という命令を使って、①②を擬似的にトランザクション操作するようにする工夫が必要です。

今回の用途では、id の値として「重複しない値が欲しい」だけで、連続性が保障されなくても支障ないため、この対策をしません。

■シーケンス番号用のテーブルを作成する

では、図5-35に示したシーケンス管理用のsequence テーブルを作成します。

◎ 操作手順 ◎ シーケンス管理用のテーブルを作成する

【1】 テーブルを作成する

- ・DynamoDB コンソールの【テーブル】をクリックしてテーブル一覧を表示します。【テーブルの作成】ボタンをクリックして(図5-36)、テーブルを作成してください。



図5-36 テーブルを作成する

[2] sequence テーブルを作成する

- ・sequence テーブルを作成します。テーブル名に「sequence」と入力します。
- ・プライマリーキーは「tablename」として、文字列型を選択します。テーブル設定は、[デフォルトの設定を使用]にチェックを付け、デフォルトのままでします。
- ・[作成] ボタンをクリックします（図 5-37）。



図 5-37 sequence テーブルを作成する

■シーケンス番号の初期値を設定する

sequence テーブルを作成したら、そのテーブルを編集して、連番の初期値を入力しておきます。具体的には、AWS マネジメントコンソールから、いま作成した sequence テーブルに、新しい項目（レコード）を追加する操作をします。

◎ 操作手順 ◎ 初期値を入力する

[1] テーブルに項目を追加する

- ・作成した sequence テーブルをクリックして開きます。
- ・テーブルに項目（SQL データベースで言うところのレコードのことです）を追加するため、[項

【目の作成】ボタンをクリックしてください（図5-38）。



図5-38 テーブルに項目を追加する

[2] プライマリーキーの値を入力する

- 新しく項目を追加する画面が表示されます。まずは、プライマリーキーとなる tablename に値を入力します。プライマリーキー名は、何でもよいのですが、ここでは「user」と入力します（図5-39）。

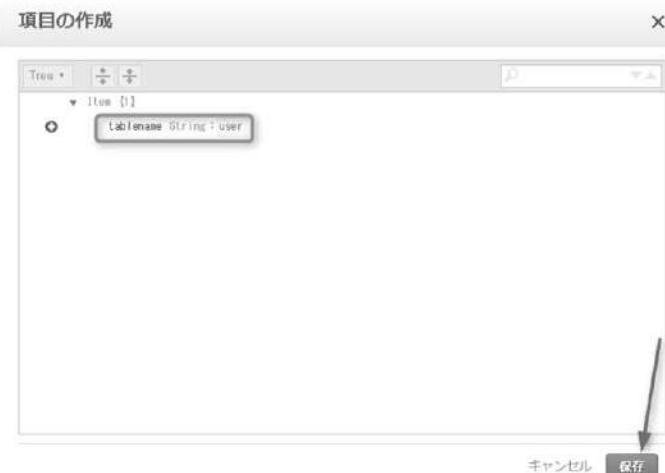


図5-39 プライマリーキーを入力する

[3] 属性を追加する

- ・属性を追加します。ここでは、Number 型（数値型）の属性を追加します。
- ・[+] ボタンをクリックし、[Append] → [Number] を選択してください（図 5-40）。



図 5-40 Number 型の属性を追加する

[4] seq 属性として設定する

- ・追加した属性を seq という名前で設定します。値は「0」とします。
- ・入力したら [保存] をクリックして保存します（図 5-41）。



図 5-41 seq 属性として設定する

すると、「tablenameがuser、seqが0」の項目が追加されます（図5-42）。



図5-42 追加された項目

のちに作るLambda関数では、この値をuserテーブルのidプライマリーキーの値として使い、userテーブルに項目が追加されるたびに、1、2、3…と増やしていくようにプログラミングします。

5-4-5 Lambda関数からDynamoDBにアクセスするための権限設定

ここまで操作で、本システムに必要となるテーブルの作成が終わりました。

次の節では、これらのテーブルにアクセスして、ユーザーが入力フォームに入力したデータなどを書き込むLambda側のプログラムを作っていくますが、そうしたプログラムの実行には、当然、これらのテーブルへのアクセス権限が必要です。

本書では、Lambda関数を「role-lambdaexec」というロールのもとで実行しています。そこでこのロールに対して、作成したsequenceテーブルとuserテーブルに対するアクセス権を設定します。

いくつかの方法がありますが、ここでは話を簡単にするため、role-lambdaexecロールに対して、すべてのDynamoDB操作ができるポリシーとなる「AmazonDynamoDBFullAccess」を適用することでアクセス権を与えることにします。その方法については、Appendix Bを参照してください。

λ Column 個別のテーブルに対してアクセス権を設定する

本書では、話を簡単にするため、AmazonDynamoDBFullAccessポリシーを適用していますが、実際には、sequenceテーブルとuserテーブルに対してだけ読み書きの権限を設定すれば十分です。

もし、個別の設定をしたいときは、ポリシージェネレータを使って、

● 5-5 Lambda 関数で DynamoDB にアクセスする

arn:aws:dynamodb:region:AWSのアカウントID:table/テーブル名

に対して、DynamoDB の適切なアクセス権を設定します。テーブルに対して、「読み込みだけ」や「書き込みだけ」など、特定の操作に対してだけ権限を設定することもできます（図 5-43）。



図 5-43 DynamoDB へのアクセス権をテーブル単位で設定する

5-5 Lambda 関数で DynamoDB にアクセスする

DynamoDB データベースのテーブルの準備が整ったところで、作成した Lambda 関数を改良し、入力フォームから POST されたデータを、これらのテーブルに書き込むようにしてみましょう。

具体的には、入力フォームに入力された「氏名」「メールアドレス」、そして、「クライアントの IP アドレス」を、テーブルに書き込む処理を加えます。すでに説明したように、DynamoDB には連番を作成するシーケンス番号のような機能がないので、どのようにして連番を作るのかというのも、実装のポイントとなります。

5-5-1 Lambda関数から DynamoDBに書き込むプログラムの例

すでに提示したリスト5-1の結果からわかるように、入力フォームに入力されたデータやクライアントのIPアドレスは、イベント引数eventの次のパラメータで取得できます。

入力フォームに入力された値：event['body']

クライアントのIPアドレス：event['requestContext']['identity']['sourceIp']

これらのデータをDynamoDBのuserテーブルに書き込む処理は、リスト5-3のようになります。

このプログラムを、Lambdaコンソールに入力して保存してください（図5-44）。そして、入力フォームで、適当な氏名とメールアドレスを入力して登録しましょう（図5-45）。登録すると、画面には「登録ありがとうございました。」と表示されます。

リスト5-3 DynamoDBにデータを書き込む例

```

import json
import boto3
import urllib.parse
import time
import decimal

# DynamoDBオブジェクト
dynamodb = boto3.resource('dynamodb')

# 連番を更新して返す関数
def next_seq(table, tablename):
    response = table.update_item(
        Key={
            'tablename' : tablename
        },
        UpdateExpression="set seq = seq + :val",
        ExpressionAttributeValues= {
            ':val' : 1
        },
        ReturnValues='UPDATED_NEW'
    )
    return response['Attributes']['seq']

def lambda_handler(event, context):
    try:
        # シーケンスデータを得る
        seqtable = dynamodb.Table('sequence')

```

```

nextseq = next_seq(seqtable, 'user')

# フォームに入力されたデータを得る
param = urllib.parse.parse_qs(event['body'])
username = param['username'][0]
email = param['email'][0]

# クライアントのIPを得る
host = event['requestContext']['identity']['sourceIp']

# 現在のUNIXタイムスタンプを得る
now = time.time()

# userテーブルに登録する
usertable = dynamodb.Table("user")
usertable.put_item(
    Item={
        'id' : nextseq,
        'username' : username,
        'email' : email,
        'accepted_at' : decimal.Decimal(str(now)),
        'host' : host
    }
)

# 結果を返す
return {
    'statusCode' : 200,
    'headers' : {
        'content-type' : 'text/html'
    },
    'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head><body>登録ありがとうございました。</body></html>'
}
except:
    import traceback
    traceback.print_exc()
    return {
        'statusCode' : 500,
        'headers' : {
            'content-type' : 'text/html'
        },
        'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head><body>内部エラーが発生しました。</body></html>'
    }

```



図5-44 リスト5-3をLambdaコンソールで入力する

氏名：山田太郎	<input type="button" value="登録"/>
メールアドレス：yamada@example.co.jp	<input type="button" value="登録"/>

図5-45 入力フォームからテストする

■ DynamoDBに書き込まれたことを確認する

入力フォームからPOSTしたら、その内容がDynamoDBに書き込まれているかどうかを確認します。

まずは、userテーブルを確認します。ここには、idが「1」の値として、各種の値が登録されていることがわかります（図5-46）。日付であるaccepted_atは、UNIXタイムスタンプとして保存されています。

The screenshot shows the AWS DynamoDB console. On the left, a sidebar lists tables: 'example', 'exampleA', 'sequence', and 'user'. The 'user' table is selected. The main area shows the table details for 'user'. The table has the following schema:

- Primary Key: 'user_id'
- Attributes:
 - 'id' (Number, Value: 1)
 - 'accepted_at' (Number, Value: 1501992057.0741503)
 - 'email' (String, Value: yamada@example.co.jp)
 - 'host' (String, Value: 116.0.244.66)
 - 'username' (String, Value: 山田太郎)

図5-46 userテーブルを確認する

● 5-5 Lambda 関数で DynamoDB にアクセスする

次に、sequence テーブルを確認してみましょう。こちらは、seq の値が 1 増えていることがわかります（図 5-47）。

テーブル名による	名前	seq
example		
exampleA		
sequence	user	1

図 5-47 sequence テーブルを確認する

5-5-2 DynamoDB にアクセスするためのライブラリとオブジェクト

プログラムの動作を確認したところで、リスト 5-3 の内容を順に説明していきます。

Python の場合、DynamoDB にアクセスするには、Boto3 ライブラリを使います。Boto3 ライブラリは、Lambda コンテナにデフォルトでインストールされているため、別途、インストールすることなく利用できます。

Boto3 ライブラリで DynamoDB にアクセスするには、「DynamoDB.Client」というオブジェクトを使います。このオブジェクトは、次のようにして生成します。

```
dynamodb = boto3.resource('dynamodb')
```

λ Column 接続先のリージョンを指定する

次のように `region_name` 引数を設定すると、接続先のリージョンを指定することもできます。省略した場合は、実行されているのと同じリージョンとなります。

```
dynamodb = boto3.resource('dynamodb', region_name='ap-northeast-1')
```

5-5-3 シーケンス番号の更新と取得

リスト5-3において、次の部分がシーケンス番号を取得する処理です。

```
# シーケンスデータを得る
seqtable = dynamodb.Table('sequence')
nextseq = next_seq(seqtable, 'user')
```

■テーブルの取得

テーブルを操作するには、対象のテーブルを `DynamoDB.Table` オブジェクトとして取得します。このオブジェクトは、`DynamoDB.Client` オブジェクトの `Table` メソッドで取得できます。たとえば、`sequence` テーブルを取得する場合、次のようにになります。

```
seqtable = dynamodb.Table('sequence')
```

`DynamoDB.Table` オブジェクトには、そのテーブルに含まれる項目を読み書きするメソッドが提供されており、それらのメソッドを使ってテーブルにアクセスします。

■シーケンス番号の更新と取得

`sequence` テーブルを操作して、シーケンス番号を増やし、その増えた値を取得する処理が、`next_seq` 関数です。

次のように定義しています。

```
def next_seq(table, tablename):
    response = table.update_item(
        Key={
            'tablename' : tablename
        },
        UpdateExpression="set seq = seq + :val",
        ExpressionAttributeValues= {
            ':val' : 1
        },
        ReturnValues='UPDATED_NEW'
    )
    return response['Attributes']['seq']
```

ここまで操作において、`sequence` テーブルには、「user」というプライマリーキーを持つ項目があり、その項目の「seq」という属性に「0」を設定しています(図5-48)。この値を増やして、

● 5-5 Lambda 関数で DynamoDB にアクセスする

その増えた結果を返すのが、この関数の処理です。



図 5-48 next_seq 関数では、この seq の値を増やして、増やした結果を返す

①属性の更新

属性の値を増やして、その増やした結果を得るには、`update_item` メソッドを使って計算式を使うことで更新します。

```
response = table.update_item(
    Key={
        'tablename' : tablename
    },
    UpdateExpression="set seq = seq + :val",
    ExpressionAttributeValues= {
        ':val' : 1
    },
    ReturnValues='UPDATED_NEW'
)
```

設定している引数の意味は、次の通りです。

(1) Key

対象となる項目を特定するキーです。リスト 5-3 の処理では、`tablename` 変数には、実際には「'user'」という文字列が渡されるので、「`tablename` 属性が'user' という値であるものを対象とする」という意味になります。

(2) UpdateExpression

更新する式です。次のように指定しています。

```
set seq = seq + :val
```

これは `seq` 属性を `seq` 属性の値に「`:val` というパラメータの値」を足したものを設定するという意味です。パラメータの名称は「`:`」で始まります。`:val` というパラメータは、次の(3)で指定します。

(3) ExpressionAttributeValues

(2) の式で利用するパラメータです。次の値を設定しています。

```
'val' : 1
```

これは「`:val`」というパラメータに「1」を設定するという意味です。すなわち、(2)(3)を総合すると、

```
set seq = seq + 1
```

ということになり、`seq` の属性値が「1増える」という挙動になります。

(4) ReturnValue

戻り値として、どのような値を返すかを指定します。表5-5のいずれかの値を指定できます。

ここでは「`'UPDATED_NEW'`」を指定しているので、「更新した後の新しい値」が戻り値として返されます。すなわち、(2)(3)で `seq` 属性を1増やしているので、戻り値として、「1だけ増えた後の `seq` 属性の値」が得られます。

表5-5 ReturnValueの取り得る値

値	意味
'NONE'	値を返さない
'ALL_OLD'	追加・更新・削除されたすべての「元の値」を返す
'UPDATED_OLD'	更新した「元の値」を返す
'ALL_NEW'	追加・更新したすべての「新しい値」を返す
'UPDATED_NEW'	更新したすべての「新しい値」を返す



Memo 戻り値の値

この戻り値は、`update_item` メソッド以外に `add_item` メソッド、`delete_item` メソッドでも使われる汎用的なものなので「追加」と「削除」の意味を含んでいますが、`update_item` メソッドでは、「更新」操作しかありません。

②更新後の属性値の取得

`updated_item` メソッドの戻り値は、次の書式のディクショナリ型となります。

```
{
    'Attributes': { 更新された項目の値 },
    'ConsumedCapacity': { 処理に要したキャパシティーユニット情報など },
    'ItemCollectionMetrics': { ReturnItemCollectionMetrics引数を指定したときに⇒
        限り、そのコレクション情報 }
}
```

結果が含まれるのは、`Attributes` です。たとえば次のようにすれば、`seq` 属性の値を取得できます。

```
return response['Attributes']['seq']
```

`ReturnValues` には '`UPDATED_NEW`' を指定しているため、ここで得られる値は、更新後の値——`seq` に 1 が加えられたとの値——です。

5-5-4 user テーブルに項目を追加する

シーケンス番号を取得したら、その値をプライマリーキーの値として、`user` テーブルに、「入力フォームに入力された値」や「クライアントの IP アドレス」などを格納していきます。

■ body 部分の URL デコード

入力フォームから入力されたデータは、`event['body']` として取得できます。この値は、次のように URL エンコードされたクエリ文字列となっています。

```
username=%E5%B1%B1%E7%94%B0%E5%A4%AA%E9%83%8E&email=yamada%40example.co.jp
```

そこでは URL デコードします。URL デコードするには、`urllib` ライブラリを使います。`parse.parse_qs` メソッドを使うと、URL デコードして、それぞれの値を取り出せます。

```
param = urllib.parse.parse_qs(event['body'])
username = param['username'][0]
email = param['email'][0]
```

■クライアントのIPアドレスの取得

クライアントのIPアドレスは、次のようにして取得できます。

```
host = event['requestContext']['identity']['sourceIp']
```

■UNIXタイムスタンプを得る

現在の日時は、`time` ライブラリの `time` 関数を使って取得できます。この値は、Float型の UNIX タイムスタンプとなります。

```
now = time.time()
```

■userテーブルに書き込む

書き込むべき値が揃つたら、`put_item` メソッドを使って `user` テーブルに書き込みます。`put_item` メソッドはプライマリーキーを確認し、「存在するときは更新」「存在しないときは追加」という処理をします。

以下に示すように、Item引数にディクショナリとして格納したい値を指定するだけで更新できます。

```
usertable = dynamodb.Table("user")
usertable.put_item(
    Item={
        'id' : nextseq,
        'username' : username,
        'email' : email,
        'accepted_at' : decimal.Decimal(str(now)),
        'host' : host
    }
)
```

このとき1点だけ注意があります。それはFloat型の扱いです。`time` 関数で取得した UNIX タイムスタンプは Float型なのですが、DynamoDBではFloat型では保存できません。代わりに、Decimal型を使います。具体的には、次のように変換して書き込みます。

```
'accepted_at' : decimal.Decimal(str(now)),
```

5-5-5 例外処理

以上で DynamoDB への書き込み処理が完了です。あとは、ユーザーに完了の旨のメッセージを戻り値として書き出すようにします。

リスト 5-3 では、全体を `try～catch` で括り、何かしらの例外が発生したときには「500」(Internal Server Error) のステータスコードを返すようにしています。

```
try:
    …DBなどの処理…
    # 正常に処理された場合
    return {
        'statusCode' : 200,
        'headers' : {
            'content-type' : 'text/html'
        },
        'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head>⇒
<body>登録ありがとうございました。</body></html>'
    }
except:
    import traceback
    traceback.print_exc()
    return {
        'statusCode' : 500,
        'headers' : {
            'content-type' : 'text/html'
        },
        'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head>⇒
<body>内部エラーが発生しました。</body></html>'
    }
```

こうすることで、ユーザーへ生のエラーメッセージが output されてしまうことを防ぎます。

なお、例外が発生したときには、以下のようにスタックトレースを標準出力に書き出している——これは CloudWatch Logs に書き出される——ので、例外の原因もつかみやすくなります。

```
import traceback
traceback.print_exc()
```

Λ Column DynamoDBから項目を読み取るには

本章では、DynamoDBの操作は、「更新」しかしませんが、実際にアプリケーションを作るときには、「読み取り」も必要になると思うので、補足しておきます。

DynamoDBにおいて、データの読み込みをするには、次のいずれかのメソッドを使います。

①get_item メソッド

パーティションキーとソートキーの組み合わせ（もしくはパーティションキーのみ）を指定して、どちらにも合致する項目をひとつ取得します。

②query メソッド

パーティションキーとソートキーを指定し、ソートキーが指定した条件の範囲内にあるものを複数取得します。

③scan メソッド

すべての属性に対して、条件を指定して検索します。条件を指定せず、全件取得することもできます。1項目ずつ読み込むため、多くの読み取りキャパシティーを消費します。

たとえば全件取得するには、③の scan メソッドを使って、リスト 5-4 のような Lambda 関数を作ります。この Lambda 関数を API Gateway 経由で呼び出すと、user テーブルに登録した、すべてのユーザーを取得して JSON 形式で出力できます。

リスト 5-4 にあるように、JSON で出力するには、

```
'body' : json.dumps(response['Items'], cls=DecimalEncoder)
```

のように変換クラスを利用します。これは、json.dumps メソッドが、標準では Decimal オブジェクトを変換できないためで、DynamoDB のデータを JSON 出力したいときの常套句です。

実行の結果、たとえば、次のような JSON データが得られます。

```
"[{"accepted_at": 1501992057.0741503, "username": "\u5c71\u7530\u592a\u90ce", "id": 1, "email": "yamada@example.co.jp", "host": "11.0.244.66"}]"
```

リスト 5-4 user テーブルに登録したすべてのユーザーを取得して JSON 形式として出力する例

```
import boto3
import json
import decimal

dynamodb = boto3.resource('dynamodb')

class DecimalEncoder(json.JSONEncoder):
```

```

def default(self, o):
    if isinstance(o, decimal.Decimal):
        if o % 1 > 0:
            return float(o)
        else:
            return int(o)
    return super(DecimalEncoder, self).default(o)

def lambda_handler(event, context):
    usertable = dynamodb.Table("user")
    response = usertable.scan()

    return {
        'statusCode' : 200,
        'body' : json.dumps(response['Items'], cls=DecimalEncoder),
        'headers' : {
            'Content-Type' : 'application/json'
        }
    }
}

```

5-6 署名付き URL を発行する

ここまで操作で、入力フォームに入力された氏名やメールアドレスを DynamoDB に保存する操作まで実現できました。次に、ユーザーごとに有効期限付きの URL を発行する機能を実装します。

5-6-1 署名付き URL とは

S3 には、有効期限付きの URL を発行する機能があります。この機能を署名付き URL と言います。署名付き URL には、「いまから 48 時間」のように有効期限が定められており、次に示すように、後ろに ID や有効期限、そして、偽装されないための署名が付けられた構造をしています。

<https://secretweb000.s3.amazonaws.com/special.pdf?AWSAccessKeyId=…略…&Signature=…略…&Expires=1501706972>

Expires パラメータが、無効になる日時（UNIX タイムスタンプ）です。それ以外のさまざまなデータは、この URL の偽装を防ぐためのパラメータです。

署名付き URL は、AWS SDK や aws-cli コマンドを使うことで生成できます。



Memo aws-cli コマンド

aws-cli は、AWS を操作するためのコマンドラインツールです。開発者のクライアント PC などにインストールして、AWS を操作するときに使います。

ここでは、この署名付き URL の機能を使って、ユーザーが Web フォームで登録してから 48 時間だけ有効なダウンロード専用 URL を発行する仕組みを作成します。

5-6-2 コンテンツを配信する S3 バケットを作る

署名付き URL は、S3 の機能です。そこで、S3 バケットを作り、そこにユーザーに配信するコンテンツを配置しておきます。

■ S3 バケットの作成

まずは、適当な S3 バケットを作成します。

◎ 操作手順 ◎ S3 バケットを作る

【1】 バケットを作成する

- ・S3 コンソールを開き、【バケットを作成する】ボタンをクリックして、S3 バケットを作成します（図 5-49）。

The screenshot shows the Amazon S3 console interface. At the top, there's a navigation bar with 'サービス' (Services), 'リソースグループ' (Resource Groups), and other account settings. Below the header, a message says 'Amazon S3へようこそ。新しいバケットを作成するか、既存のバケットを選択してプロパティを表示、設定します。' (Welcome to Amazon S3. Create a new bucket or select an existing one to view and manage properties). The main area has a search bar 'Q バケット検索' (Search buckets) and a button '+ バケットを作成する' (Create new bucket). There are also buttons for 'バケットを削除する' (Delete bucket) and 'バケットを空にする' (Empty bucket). Below these buttons, it shows '3 バケット' (3 Buckets) and '1 リージョン' (1 Region). A table lists the buckets: 'exampleread000' located in 'アジアパシフィック(東京)' with a creation date of '2017/07/15 16:53:10'. A red arrow points from the text '【1】 バケットを作成する' to the '+ バケットを作成する' button.

図 5-49 S3 バケットを作成する

[2] バケット名やリージョンを設定する

- ・バケット名やリージョンを設定します。ここでは「secretweb000」という名前のバケットとしました。リージョンは、アジアパシフィック（東京）にしておきます（図 5-50）。



図 5-50 バケット名やリージョンを設定する

なお、S3 バケットの名前は、世界でユニークです。本書の執筆において、著者が「secretweb000」という名前を使っているので、読者の皆様は、他の適当なバケット名で作成してください。

[3] プロパティの設定

- ・バージョニングやログの記録、タグ付けなどを設定します。ここでは、とくに何も指定する必要はないので、そのまま次の画面に進んでください（図 5-51）。



図 5-51 プロパティの設定

[4] アクセス許可の設定

- ・アクセス許可を設定します。このS3バケットへは、署名付きURLを使って、一定期間だけアクセスさせてるので、とくにセキュリティ設定は必要ありません。そのまま、次に進んでください(図5-52)。

ただし、S3コンソールを使って開発者がファイルをアップロードするのであれば、そのための権限は必要です。



図5-52 アクセス許可の設定

[5] 確認

- ・確認画面が表示されます。[バケットを作成]をクリックして、バケットを作成してください(図5-53)。



図5-53 確認

● 5-6 署名付き URL を発行する

■ユーザーに提供するコンテンツをアップロードする

S3 バケットを作成したら、ユーザーに提供するコンテンツをアップロードします。

ここでは、「special.pdf」というファイルを、この S3 バケットに配置します（図 5-54）。アップロードの手順は、「■ index.html をアップロードする（p.149）」を参照してください。



図 5-54 提供するコンテンツをアップロードする

ただし、アクセス権の設定では、【このオブジェクトにパブリック読み取りアクセスを付与しない（推奨）】を選択しておきます。こうすることで、アップロードしたファイルは、所有者（AWS マネジメントコンソールで操作しているユーザー）以外は、アクセスできなくなります（図 5-55）。



図 5-55 パブリック読み取りアクセス権限を付与しない

5-6-3 署名付き URL を作る

では、このS3バケットに置いたコンテンツに対して、署名付きURLを作る処理を加えましょう。このプログラムはリスト5-5のようになります。

リスト5-5 署名付きURLを作る（抜粋）

```
# 署名付きURLを作る
s3 = boto3.client('s3')
url = s3.generate_presigned_url(
    ClientMethod = 'get_object',
    Params = {'Bucket' : 'secretweb000', 'Key' : 'special.pdf'},
    ExpiresIn = 48 * 60 * 60,
    HttpMethod = 'GET')

# userテーブルに登録する
usertable = dynamodb.Table("user")
usertable.put_item(
    Item={
        'id' : nextseq,
        'username' : username,
        'email' : email,
        'accepted_at' : decimal.Decimal(str(now)),
        'host' : host,
        'url' : url
    }
)
```

まずは、Boto3ライブラリのS3オブジェクトを取得します。

```
s3 = boto3.client('s3')
```

そして、generate_presigned_urlメソッドを呼び出すと、署名付きURLを生成できます。

```
url = s3.generate_presigned_url(
    ClientMethod = 'get_object',
    Params = {'Bucket' : 'secretweb000', 'Key' : 'special.pdf'},
    ExpiresIn = 48 * 60 * 60,
    HttpMethod = 'GET')
```

①ClientMethod

操作可能なS3の操作です。読み取り権限を与えるときは「get_object」を指定します。

②Params

S3 のパラメータを指定します。Bucket キーには S3 バケット名を、Key キーにはオブジェクト名（ファイル名）を渡します。

ここでは「secretweb000」という名前の S3 バケットにある「special.pdf」というファイルを対象とした署名付き URL を発行しています。

③ExpiresIn

有効期間を指定します。単位は秒です。ここでは 48 時間を指定しています。

④HTTPMethod

許可する HTTP メソッドを指定します。ここでは GET メソッドでの操作を許可しました。

こうしてできた URL を、DynamoDB の url 属性に書き込むようにしました。

```
usertable.put_item(  
    Item={  
        'id' : nextseq,  
        'username' : username,  
        'email' : email,  
        'accepted_at' : decimal.Decimal(str(now)),  
        'host' : host,  
        'url' : url  
    }  
)
```

λ Column 署名付き URL をアップロード目的に利用する

本書では「一定期間だけダウンロードできる仕組み」として署名付き URL を利用しましたが、「一定期間だけアップロードする仕組み」にも、署名付き URL は利用できます。

たとえば、ユーザーに何か大きいファイルを送信してもらう場合に、一時的なアップロード先を提供する場面で活用できます。

■発行された署名付き URL でアクセスする

では、発行した署名付き URL でアクセスできるかを確かめてみましょう。用意した入力フォームから適当なユーザーを登録してみてください（図 5-56（1））。そして、DynamoDB コンソール

で確認すると、url属性に署名付きURLが書き込まれているはずです（図5-56（2））。

氏名：	田中次郎
メールアドレス：	tanaka@example.co.jp
<input type="button" value="登録"/>	

図5-56（1）ユーザーを登録する

The screenshot shows the AWS Lambda console interface. On the left, there's a sidebar with a search bar and a list of tables: 'example', 'exampleA', 'sequence', and 'user'. The 'user' table is selected. The main area shows a table with columns: 'createdAt', 'email', 'host', 'url', and 'username'. There are two rows of data: one for 'tanaka@example.co.jp' and another for 'myemail@example.com'. A modal dialog is overlaid on the page, displaying a URL: <https://secretweb000.s3.amazonaws.com/special.pdf?AWSAccessKeyId=ASIAJORN2HIMG62I4AA&Signature=%2FgppAnykmDxmtiqPN...>. The URL is highlighted with a red box.

図5-56（2）署名付きURLを確認する

署名付きURLを確認したら、この署名付きURLをWebブラウザに貼り付けてください（図5-57）。すると、`secret.pdf`をダウンロードできるはずです。

The screenshot shows a web browser window with a single tab labeled '新しいタブ'. The URL in the address bar is <https://secretweb000.s3.amazonaws.com/special.pdf?AWSAccessKeyId=ASIAJORN2HIMG62I4AA&Signature=%2FgppAnykmDxmtiqPN...>. Below the address bar, there's a message: 'こちらのブックマークバーにブックマークを追加すると簡単にページにアクセスできます。今すぐブックマークをインポート。' In the center of the page, there's a circular logo with a stylized 'OO' and a hat. Below the logo, the text 'シークレットモードです' is displayed. At the bottom of the page, there's a note: '現在、シークレットモードで閲覧しています。あなたのアクティビティは、この端末を利用する他のユーザーには表示されません。ただし、ダウンロードしたファイルとブックマークは通常どおり保存されます。詳しく見る'.

図5-57 署名付きURLでアクセスする

この URL は 48 時間後には、無効になります。無効になった状態でアクセスすると、図 5-58 のようなメッセージが表示されます。



図 5-58 期限切れでアクセスした場合

5-7 メールの送信

最後に、登録者に対して、この署名付き URL をメールで伝えるようにすれば、プログラムは完成です。

5-7-1 Amazon SES でメール送信する

AWS からメールを送信するには、Amazon SES を使います。ただし SES が提供されているリージョンは、本書の執筆時点では限定的で、東京リージョンもサポート対象外です。そのため本書では、「バージニア北部 (us-east-1)」の Amazon SES を使うことにします（図 5-59 (1)、(2)）。



図 5-59 (1) バージニア北部の Amazon SES を使う (1)

Region Unsupported

SES is not available in アジアパシフィック (東京). Please select another region.

Supported Regions

EU (アイルランド)

米国東部 (バージニア北部)

米国西部 (オレゴン)

図 5-59 (2) バージニア北部の Amazon SES を使う (2)



Memo リージョンの切り替え

一度バージニア北部に切り替えると、S3やLambdaなど他のサービスを操作するときもバージニア北部が対象になります。Amazon SES以外の操作をするときは、右上から「東京リージョン」に戻してください。

5-7-2 メールの送信制限を解除する

デフォルトではAmazon SESの機能が一部制限され、あらかじめ登録した宛先にしかメールを送信できない仕様になっています。この制限された動作のことをサンドボックスと言います。

サンドボックスには、次の制限があります。

- SESメールボックスシミュレーターと検証されたメールアドレス宛およびドメイン宛だけにメールを送信できる
- 送信できるメール数は24時間当たり200メッセージまで
- 受信する場合、1秒間に1メッセージしか受け取れない

サンドボックスを解除するには、SESコンソールの【Sending Statistics】メニューを開き、表示されている枠内の【Request a Sending Limit increase】ボタンをクリックします(図5-60)。

The screenshot shows the SES console interface. On the left, a sidebar menu is open with 'Sending Statistics' selected. A callout bubble points to this selection with the number '1'. In the main content area, there is a message about being in a 'sandbox' region and a 'Request a Sending Limit Increase' button. A callout bubble points to this button with the number '2'. To the right, there is an 'Additional Information' sidebar.

図5-60 SESコンソール画面

するとサポートセンターへの申請フォームが表示されるので、必要事項を入力して申請します(図5-61)。手続きには1営業日かかります。申請に通れば、これらの制限は解除されます。

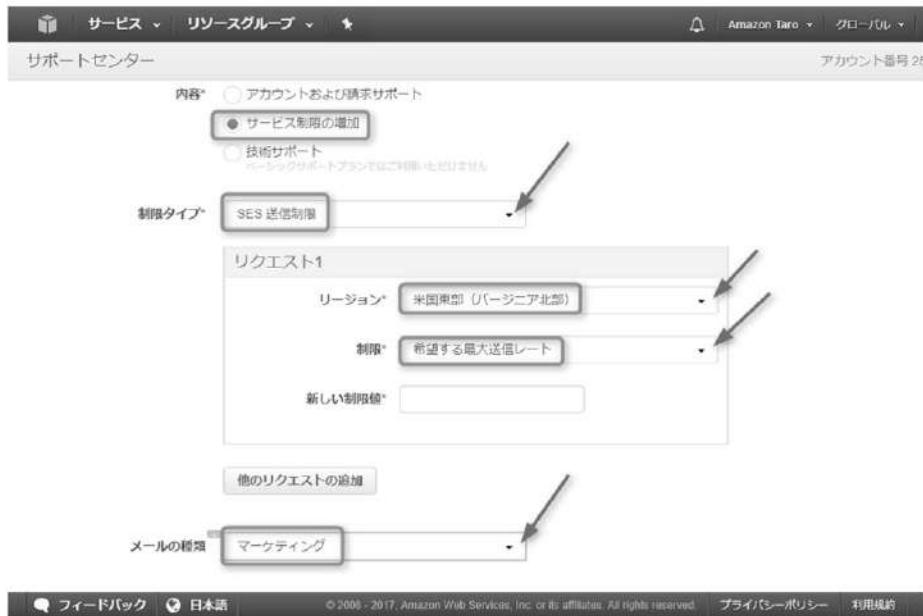


図 5-61 制限の緩和を申請する

5-7-3 ドメインやメールアドレスの検証

SES を使ってメールを送信する場合、「検証」と呼ばれる操作が必要です。

①サンドボックス環境でない場合

メールの差出人のメールアドレス、またはドメインの検証が必要。

②サンドボックス環境である場合

①に加えて、宛先のメールアドレス、またはドメインの検証が必要。

検証には「ドメインの検証」と「メールアドレスの検証」の2通りの方法があり、いずれかの方法で設定します。

①ドメインの検証

ドメイン単位で検証する方法です。ドメインに対して DKIM (Domain Key Identified Mail) 署名を設定することで、そのドメインに対するメールの送受信を許可します。

②メールアドレスの検証

メールアドレス単位で検証する方法です。SESコンソールから、送受信したいメールアドレスに対して確認用のメールを送信する操作をします。メールソフトなどで、そのメールを受信して、記載されているURLをクリックすると検証されたものとみなされ、以降、そのメールアドレスで送受信可能になります。

■メールアドレスを検証する

ドメインの検証をするには、複雑なDNSサーバーの設定が必要になるため、ここでは、メールアドレスを検証する方法を説明します。

メールアドレスを検証するには、次の操作を行います。

この操作は、差出人のメールアドレスに対しては必須です。そしてサンドボックス環境の場合は、宛先のメールアドレスに対しても必要です。

◎操作手順 ◎ **特定のメールアドレス宛、もしくはメールアドレスから送信できるよう
に検証する**

【1】メールアドレスの検証を始める

- ・SESコンソールで [Email Addresses] メニューを開き、[Verify a New Email Address] ボタンをクリックします（図5-62）。

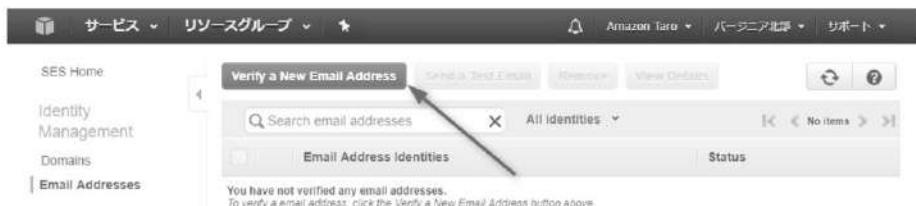


図5-62 [Verify a New Email Address] ボタンをクリックする

【2】検証用のメールを送信する

- ・検証するメールアドレスをテキストボックスに入力し、[Verify This Email Address] ボタンをクリックします（図5-63（1））。するとメールが送信されます。操作が終わったら、[Close] ボタンをクリックして閉じてください（図5-63（2））。
- ・メールが送信されると、[Email Addresses] メニューに、そのメールアドレスが pending verification というステータスで登録されます（図5-64）。



図 5-63 (1) 検証用のメールを送信する (1)

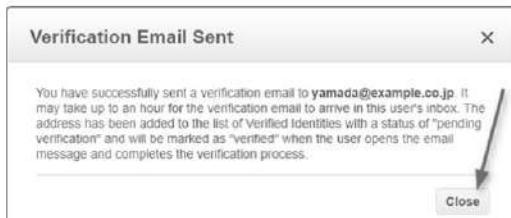


図 5-63 (2) 検証用のメールを送信する (2)

The screenshot shows the AWS SES Identity Management interface. Under the "Email Addresses" section, an email address "yamada@example.co.jp" is listed. To the right of the address, a "Status" field displays "pending verification (resend)". A red arrow points from the "Status" field towards the status text.

図 5-64 検証中のステータスとして登録される。

[3] メールに記載されたリンクをクリックします。

【2】の宛先には、次のようなメールが届きます。

【メールの例】

Dear Amazon Web Services Customer,

We have received a request to authorize this email address for use with Amazon SES and Amazon Pinpoint in region US East (N. Virginia). If you requested this verification, please go to the following URL to confirm that you are authorized to use this email address:

<https://email-verification.us-east-1.amazonaws.com/?AWSAccessKeyId=XXXXXX>

… 略 …

Your request will not be processed unless you confirm the address using this URL. This link expires 24 hours after your original verification request.

If you did NOT request to verify this email address, do not click on the link. Please note that many times, the situation isn't a phishing attempt, but either a misunderstanding of how to use our service, or someone setting up email-sending capabilities on your behalf as part of a legitimate service, but without having fully communicated the procedure first. If you are still concerned, please forward this notification to aws-email-domain-verification@amazon.com and let us know in the forward that you did not request the verification.

To learn more about sending email from Amazon Web Services, please refer to the Amazon SES Developer Guide at [http://docs.aws.amazon.com/pinpoint/latest/userguide/welcome.html](http://docs.aws.amazon.com/ses/latest/DeveloperGuide>Welcome.html and Amazon Pinpoint Developer Guide at <a href=).

Sincerely,

The Amazon Web Services Team.

ここに記載されているリンクをクリックすると、次の画面が表示されます。これで検証は完了です（図5-65）。



図5-65 検証の完了

SESコンソールの[Email Addresses]メニューに表示されるステータスは「verified」になり

ます（図 5-66）。このように登録されたメールアドレスは、「差出人」として利用したり、サンドボックス環境の「宛先」として利用したりできます。

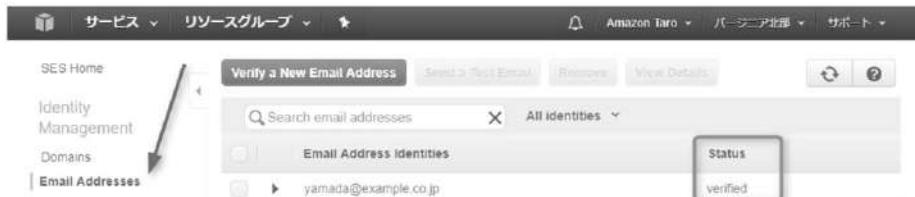


図 5-66 ステータスが verified になれば、そのメールアドレスを利用できる

5-7-4 Lambda 関数が SES を使ってメール送信するための権限設定

ここまで設定が、SES を使ってメール送信するための汎用的な設定です。

次に、Lambda 関数が利用するための設定をします。Lambda から SES を使ってメールを送信するには、Lambda 関数の実行ロールに対して、SES へのアクセス権が必要です。具体的には、本書で使っている Lambda 関数の実行ロールである「role-lambdaexec ロール」に対して、「AmazonSESFullAccess」 という管理ポリシーを適用します。その方法については、Appendix B を参照してください。

5-7-5 SES を使ってメールを送信する

Lambda 関数から SES を使ってメールを送信します。プログラムはリスト 5-6 のようになります。

リスト 5-6 Lambda 関数から SES を使ってメールを送信する例

```

import json
import boto3
import urllib.parse
import time
import decimal

# DynamoDBオブジェクト
dynamodb = boto3.resource('dynamodb')

# メール送信関数
MAILFROM = 'example@example.com'
def sendmail(to, subject, body):

```

```
client = boto3.client('ses', region_name='us-east-1')

response = client.send_email(
    Source=MAILFROM,
    ReplyToAddresses=[MAILFROM],
    Destination={
        'ToAddresses' : [
            to
        ]
    },
    Message={
        'Subject': {
            'Data' : subject,
            'Charset' : 'UTF-8'
        },
        'Body' : {
            'Text': {
                'Data' : body,
                'Charset' : 'UTF-8'
            }
        }
    }
)

# 連番を更新して返す関数
def next_seq(table, tablename):
    response = table.update_item(
        Key={
            'tablename' : tablename
        },
        UpdateExpression="set seq = seq + :val",
        ExpressionAttributeValues= {
            ':val' : 1
        },
        ReturnValues='UPDATED_NEW'
    )
    return response['Attributes']['seq']

def lambda_handler(event, context):
    try:
        # フォームに入力されたデータを得る
        param = urllib.parse_qs(event['body'])
        username = param['username'][0]
        email = param['email'][0]

        # クライアントのIPを得る
```

```

host = event['requestContext']['identity']['sourceIp']

# シーケンスデータを得る
seqtable = dynamodb.Table('sequence')
nextseq = next_seq(seqtable, 'user')

# 現在のUNIXタイムスタンプを得る
now = time.time()

# 署名付きURLを作る
s3 = boto3.client('s3')
url = s3.generate_presigned_url(
    ClientMethod = 'get_object',
    Params = {'Bucket' : 'secretweb000', 'Key' : 'special.pdf'},
    ExpiresIn = 10, # 48 * 60 * 60,
    HttpMethod = 'GET')

# userテーブルに登録する
usertable = dynamodb.Table("user")
usertable.put_item(
    Item={
        'id' : nextseq,
        'username' : username,
        'email' : email,
        'accepted_at' : decimal.Decimal(str(now)),
        'host' : host,
        'url' : url
    }
)

# メールを送信する
mailbody = """
{0}様

ご登録ありがとうございました。
下記のURLからダウンロードできます。

{1}
""".format(username, url)

sendmail(email, "登録ありがとうございました", mailbody)

# 結果を返す
return {
    'statusCode' : 200,
    'headers' : {

```

```

        'content-type' : 'text/html'
    },
    'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head>
d><body>登録ありがとうございました。</body></html>'
}
except:
    import traceback
    traceback.print_exc()
    return {
        'statusCode' : 200,
        'headers' : {
            'content-type' : 'text/html'
        },
        'body' : '<!DOCTYPE html><html><head><meta charset="UTF-8"></head>
d><body>内部エラーが発生しました。</body></html>'
}

```

リスト5-6において、メールを送信する関数を `sendmail` 関数として用意しました。次のようにしてメールを送信しています。

```

# メールを送信する
mailbody = """
{0}様

ご登録ありがとうございました。
下記のURLからダウンロードできます。

{1}
""".format(username, url)

sendmail(email, "登録ありがとうございました", mailbody)

```

またメールの差出人は、次のように定義しています。

```
MAILFROM = 'example@example.com'
```

このメールアドレスは、SESで検証済みのメールアドレスに変更する必要があります。
実際にから登録して動作テストすると、次のような文面のメールが送信されます。
このリンクをクリックすると、48時間以内なら、コンテンツをダウンロードできます。

山田太郎様

ご登録ありがとうございました。

下記のURLからダウンロードできます。

```
https://secretweb000.s3.amazonaws.com/special.pdf?AWSAccessKeyId=ASIAJ6XZHPT6
3N2660RQ&Signature=… 略 …
```

■ Boto3 ライブラリを使ってメールを送信する処理

Boto3 を使ってメールを送信するには、まず、`SES.Client` オブジェクトを取得します。

```
client = boto3.client('ses', region_name='us-east-1')
```

`region_name` は利用する SES のリージョン名です。省略すると Lambda 関数を実行しているのと同じリージョンになりますが、東京リージョンで Lambda 関数を利用している場合、東京リージョンでは SES は提供されていないので、リージョン名の指定は、本書の執筆時点では、必須です。

メールを送信するには、`send_email` メソッドを呼び出します。

```
response = client.send_email(
    Source=MAILFROM,
    ReplyToAddresses=[MAILFROM],
    Destination= {
        'ToAddresses' : [
            to
        ]
    },
    Message={
        'Subject': {
            'Data' : subject,
            'Charset' : 'UTF-8'
        },
        'Body' : {
            'Text': {
                'Data' : body,
                'Charset' : 'UTF-8'
            }
        }
    }
)
```

指定すべきパラメータは、以下の通りです。



Memo send_email メソッドのパラメータ

これは全パラメータではありません。ほかにも、宛先不明のときの戻り先を指定する ReturnPath や、SES のメールキーに設定する任意のタグなどを指定できます。

①Source

送信元のメールアドレスを指定します。検証済みのメールアドレスでなければなりません。

②ReplyToAddress

メールヘッダの Reply-To (返信先) として指定するメールアドレスをリストとして指定します。ほとんどの場合、①と同じ値のはずです。

③Destination

宛先を指定します。サンドボックス環境の場合、検証済みのメールアドレスでなければなりません。「ToAddresses」「CcAddresses」「BccAddresses」のいずれかで指定します。それぞれリストとして指定します。

④Messages

送信メッセージを指定します。次のパラメータを指定します。

(1) Subject

メールの件名 (タイトル) を、文字コードとともに指定します。

(2) Body

メールの本文を文字コードとともに指定します。テキストメールと HTML メールのいずれかを指定できます。テキストメールのときは「Text パラメータ」、HTML メールのときは「Html パラメータ」に指定します。

このプログラムでは利用していませんが、send_email からの戻り値は、次の書式で、SES 上のメッセージ ID が返されます。

```
{  
    'MessageId': 'string'  
}
```



Column メールの受信時に Lambda 関数を実行する

本書では、Lambda 関数からメールを送信する方法しか説明していませんが、SES ではメールの受信もでき、受信のタイミングで Lambda 関数を呼び出すようにも構成できます。

その仕組みを使うと、たとえば、「あるメールアドレスに空メールを送信したときに、その差出人のメールアドレスを自動的に DynamoDB に登録していく」といったメールでのユーザー登録システムなどを作成できます。

5-8 クロスオリジンの場合の注意点

以上でプログラムは完成となります。最後に、API Gateway 経由の Lambda 関数を、入力フォームではなく、JavaScript から実行するときの注意点について説明します。

5-8-1 Ajax 経由で API Gateway を呼び出す

ここまで掲載したプログラムでは、入力フォームの [Submit] ボタンをクリックすることで、API Gateway 経由で Lambda 関数を実行していましたが、プログラムを実行する環境によっては、クライアントサイドの JavaScript から Ajax 通信を用いて、呼び出したいこともあります（図 5-67）。

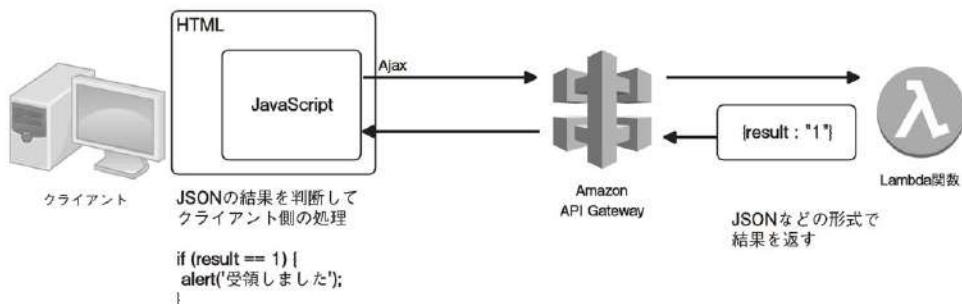


図 5-67 Ajax で API Gateway 経由の Lambda 関数を実行する

このような構成の場合、Lambda 関数側では、成功したか否かを JSON 形式のデータで返し、それをクライアント側の JavaScript で受け取って何か処理することがほとんどです。たとえば、次のように、成功したら「result に 1 を設定したものを返す」などに Lambda 関数の戻り値を変更します。

リスト5-7 Lambda関数側の戻り値をJSON形式に変更する

```

body = {'result' : 1}
result = {
    'statusCode' : 200,
    'headers' : {
        'content-type' : 'application/json'
    },
    'body' : json.dumps(body)
}

```

そしてWebフォームを次のように修正して、Ajax経由で呼び出すようにします。リスト5-8では、jQueryを利用して送信しています。

リスト5-8 Ajaxで呼び出すようにした入力フォーム

```

<!DOCTYPE html>
<html lang="ja">
<head>
<meta charset="UTF-8">
<script src="http://code.jquery.com/jquery-3.2.1.min.js"></script>
<script>
function regist() {
# XXXXXXXXXXXXは、API Gatewayのエンドポイント(図5-13)
    var apiurl = 'https://XXXXXXXXXX.execute-api.ap-northeast-1.amazonaws.com/⇒
prod/regist';
    var form = $('#myform');
    var formdata = form.serialize();

    $.ajax({
        type : 'POST',
        url : apiurl,
        data : formdata,
        dataType : 'json'
    }).done(function(response) {
        if (response.result == 1) {
            $('#msg').html('登録ありがとうございました。ダウンロード先を記載したメールをお送りしました;');
        } else {
            $('#msg').html('エラーが発生しました。' + response.message);
        }
    }).fail(function(request, status, err) {
        alert(status);
    });
}

```

```

});;

    return false;
}
</script>
</head>
<body>
<form id="myform">
氏名 :<input type="text" name="username"><br>
メールアドレス :<input type="text" name="email">
<input type="button" value="登録" onclick="regist()">
</form>
<p id="msg"></p>
</body>
</html>

```

5-8-2 ドメインをまたいだスクリプトを実行できないクロスオリジンの問題

このプログラムは、プログラムとして正しいのですが、正しく実行できません。一見して、何が原因で動かないのかわかりにくいためですが、Google の開発者ツール（F12 キーを押す）を開いて確認すると、確かにエラーが発生していることがわかります（図 5-68）。

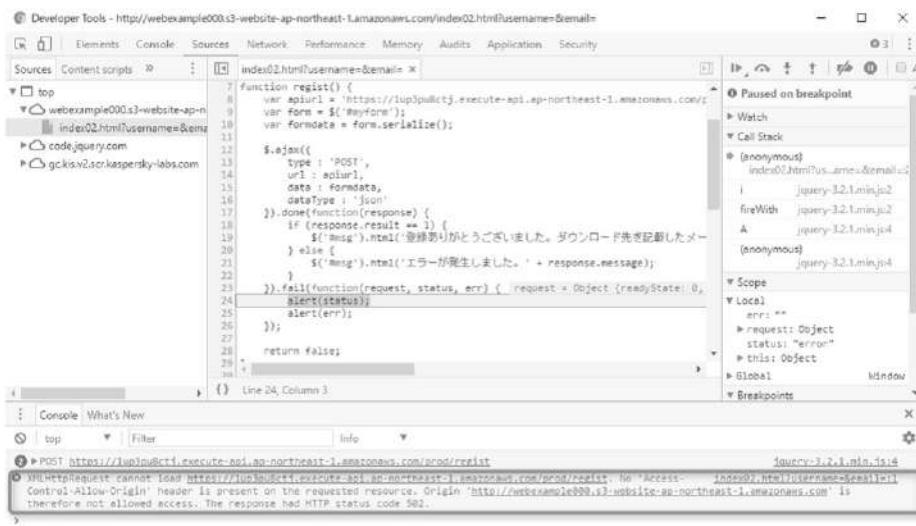


図 5-68 Ajax で呼び出すときに発生するエラー

このエラーが発生する原因是、同一生成元ポリシー（Same-Origin Policy）というセキュリティ

方針に違反するためです。JavaScriptでは、セキュリティを高めるため、デフォルトでは、「HTMLを送信したホストとしかAjax通信できない」という取り決めがあります。今回の構成の場合、HTMLを配信しているのはS3、Ajaxの呼び出し先はAPI Gatewayなので、この規約に違反します(図5-69)。

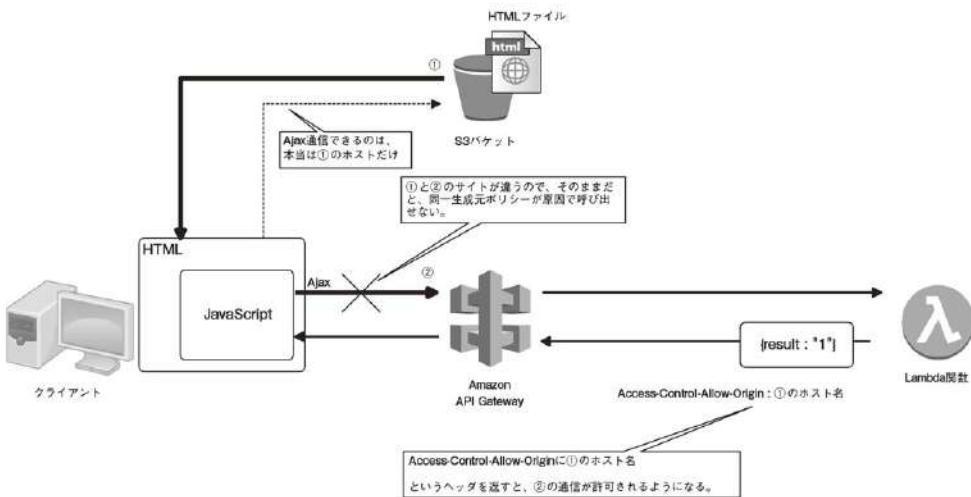


図5-69 同一生成元ポリシーに違反する

この違反を解決するには、Ajaxの呼び出し先——この構成の場合はAPI Gateway——において、`Access-Control-Allow-Origin`というヘッダに、HTMLを配信するホスト名——この場合はS3のエンドポイント——を含めた値を返すようにします。

仮に、S3のエンドポイントが以下の場合、

`http://webexample000.s3-website-ap-northeast-1.amazonaws.com`

次のHTTPヘッダをAPI Gatewayから返すようにします。

```
Access-Control-Allow-Origin : http://webexample000.s3-website-ap-northeast-1.amazonaws.com
```

そのためには、Lambda関数の戻り値を設定する`return`文を、次のように変更します。

```
result = {
    'statusCode' : 200,
    'headers' : {
        'access-control-allow-origin' : 'http://webexample000.s3-website-ap->northeast-1.amazonaws.com',
        'content-type' : 'application/json'
```

● 5-8 クロスオリジンの場合の注意点

```
},
'body' : json.dumps(body)
}
```

このように、Access-Control-Allow-Origin ヘッダを返すなどして、異なるドメイン（クロスドメイン）での呼び出せる構成にすることを CORS (Cross-Origin Resource Sharing) と言います。

CORS 構成にすることで、同一生成元ポリシーの問題が解消され、Ajax 経由でも実行できるようになります。



Column API Gateway での CORS 有効化は Lambda 統合では無効

本書では、AWS マネジメントコンソールの API Gateway の項目からの操作はしていませんが、実は、API Gateway の設定には、【CORS の有効化】というメニューがあり、CORS を許可する設定ができます。しかしここでの設定は、Lambda プロキシ統合を有効にしている場合は効きません（図 5-70）。



図 5-70 API Gateway の【CORS の有効化】をしても、クロスドメインで呼び出せるようにはならない

Lambda プロキシ統合を有効にしているときは、本文中に示しているように、Lambda 関数からの戻り値で、Access-Control-Allow-Origin ヘッダを指定してください。

5-9 まとめ

本章では、API GatewayとLambdaを組み合わせて利用する方法、DynamoDBへのデータの読み書き、S3の署名付きURLを発行する方法、そして、SESを使ってメールを送信する方法、さらに、クロスドメインで呼び出す場合の注意点までを説明しました。

① API GatewayとLambda関数の組み合わせ

- ・microservice-http-endpoint設計図から作成すると、API GatewayとLambda関数をまとめて作れます。
- ・HTTPS通信で送信されたフォームの内容やクライアントのIPアドレスなどは、イベント引数で取得できます。
- ・Lambda関数からの戻り値は、次の書式でなければなりません。

```
{  
    "isBase64Encoded": bodyがBase64エンコードされているときはtrue、そうでなければfalse  
    "statusCode": HTTPステータスコード,  
    "headers": { クライアントに返したいヘッダ情報 ... },  
    "body": クライアントに返したいボディ情報  
}
```

② DynamoDBへのアクセス

DynamoDBは、キーバリューストア型のNoSQLデータベースです。プライマリーキーを定義してテーブルを作ると、任意のデータを保存できます。

③ 署名付きURL

署名付きURLを使うと、一定期間だけ有効なURLを作成できます。

④ SESを使ったメール送信

SESを使うとメールを送信できます。デフォルトでは、検証されたメールアドレス宛にしか送信できません。

以上で、本章のプログラムは完成となります。

⑤ クロスドメインで呼び出す場合の注意点

Ajaxで呼び出す場合は、そのHTML配信するホスト名をLambda関数から、Access-Control-Allow-Originというヘッダに設定しなければなりません。

第6章

SQSとSNSトピックを使った連携

複雑な処理を構成するときは、複数の Lambda 関数を連携して実現することも珍しくありません。こうした連携の際によく使われるのが、SQS (Simple Queue Service) や SNS (Simple Notification Service) トピックです。SQS や SNS トピックをうまく利用すると、複数の処理を同時に実行する並列処理も実現できます。

- 6-1 SQSとSNSトピックのイベント事例 [p.207]
- 6-2 DynamoDB テーブルによるメールアドレス管理 [p.211]
- 6-3 S3 バケットとSQSを構成する [p.221]
- 6-4 SQSからメッセージを取り出してメールを送信する [p.242]
- 6-5 バウンスメールを処理する [p.270]
- 6-6 まとめ [p.279]

本章のポイント

● SQS を使った処理のキューイング

SQS は、メッセージをキューイングする機能です。これから処理しようとする内容を、一度、キューに溜めることで、順次、好きなときに取り出して処理できるようにします。複数のプロセスでキューを処理すれば、並列処理も実現できます。

● SNS トピックで Lambda 関数を起動

SNS トピックは、任意のメッセージを通知する機能です。SNS トピックは、Lambda 関数のイベントソースになることができ、Lambda 関数を実行できます。つまり、SNS トピックを、Lambda 関数を起動する際の引き金として、さまざまな場面で利用できます。

● バウンスメールの処理

SES でメールを送信する際、宛先不明などのエラーメールは、SNS トピックとして通知されます。SNS トピックから Lambda 関数を実行するように構成することで、バウンスマップ (bounce message) が戻ってきた宛先を、データベーステーブルから削除するといった、バウンスが発生したときの削除処理を自動化できます。

6-1 SQS と SNS トピックのイベント事例

本章では、複数の Lambda 関数を組み合わせてひとつのシステムを構成する事例として、メールの同報送信を取り上げます。

同報送信プログラムは、宛先の数だけループして送信処理を行うのが基本です。だからと言って、宛先リストの先頭から 1 通ずつ送信していくのは、あまり効率的ではありません。メール送信では API 呼び出しの待ち時間などがあるため、いくつかのメール送信ルーチンを並列動作させたほうが、全体の送信時間は短くなります。

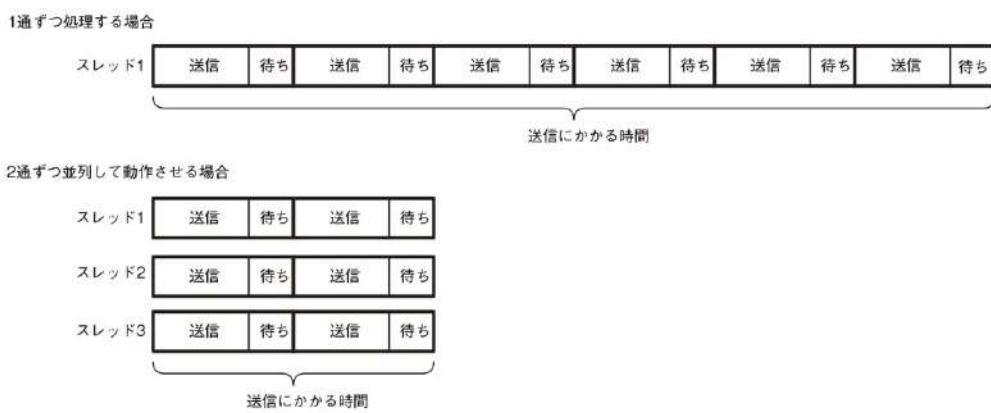


図 6-1 並列送信で高速化する

従来のプログラミング環境で図 6-1 のように並列処理でプログラミングするには、スレッドを使ったプログラミングが必要になり、構成が複雑になります。しかし Lambda なら、このようなプログラムを比較的簡単に実装できます。

6-1-1 Lambda を使ってメールを同報送信する

本章では、Lambda を使ったメールの同報送信システムを、図 6-2 のように構成します。

このシステムでは、宛先メールアドレスや氏名などを、DynamoDB にあらかじめ登録しておくものとします（図中の①）。

メールを送信したいときは、図中②の S3 バケットに、メールの「タイトル」と「本文」が記載されたテキストファイルを置くものとします。このファイルが置かれたとき、図中③の Lambda 関数が実行され、DynamoDB のテーブルから宛先となるべきメールアドレスを 1 件ずつ取得し、それを SQS というキューに格納するよう構成します。

図中④の Lambda 関数は、CloudWatch イベントを利用したスケジューリング機能を使って 5 分

第6章 SQSとSNSトピックを使った連携

ごとに起動するように構成しておきます。このLambda関数は、キューの中身に、どれだけメッセージが溜まっているかを確認し、溜まっていたときは、SNSトピックに通知を送信するものとします。

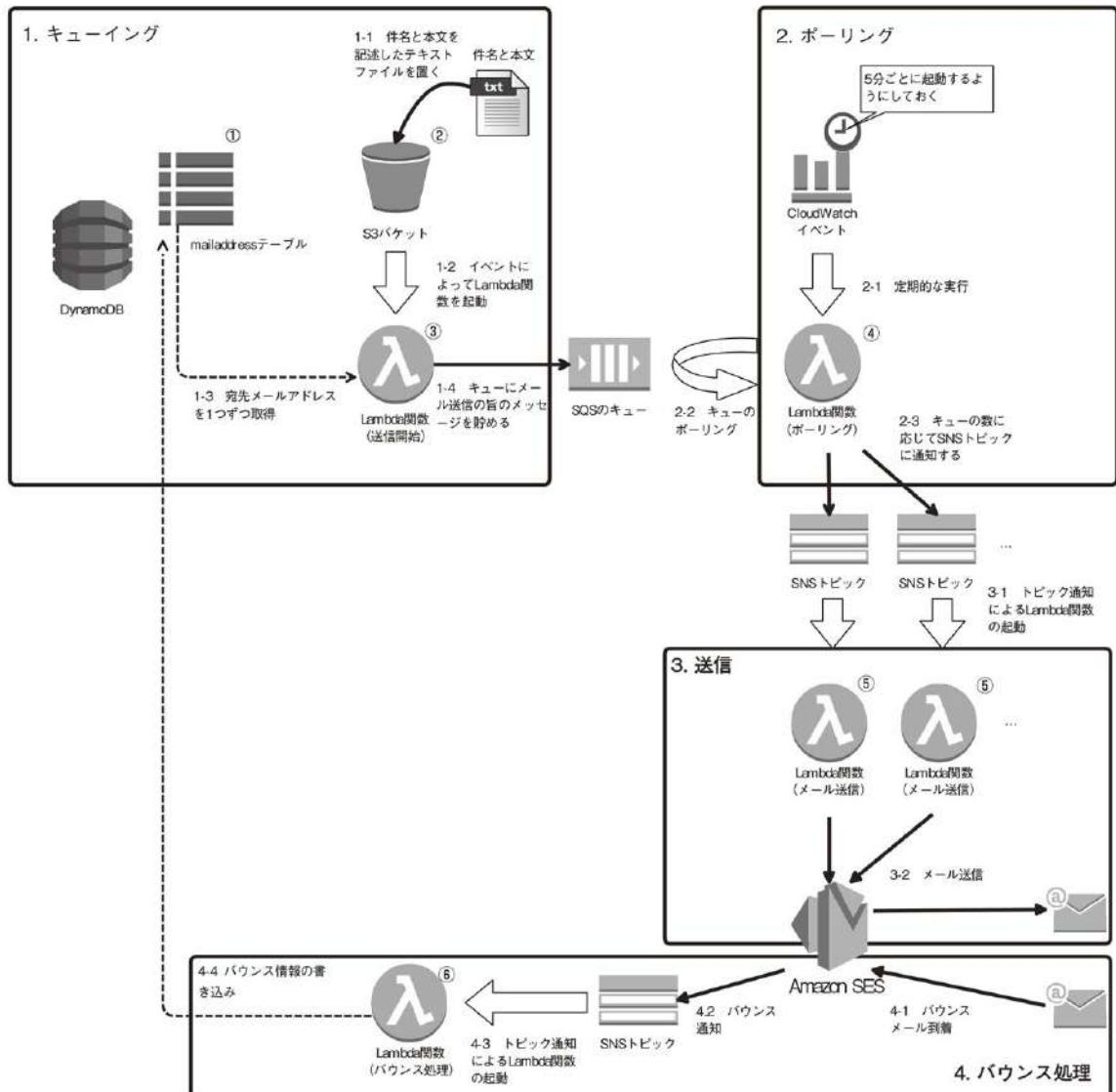


図 6-2 Lambda を使ったメールの同報送信システム

図中⑤のLambda関数はSNSトピックの通知によって実行されるように構成されたもので、このLambda関数では、キューに溜められたメッセージを取得して、実際にメールを送信します。ここでポイントとなるのが、並列実行性です。図中⑥のLambda関数では、最大10個のメ

● 6-1 SQS と SNS トピックのイベント事例

セージを処理するように構成し、図中④の Lambda 関数では、「溜まっているメッセージの数 ÷ 10」の数だけ、SNS トピックに通知を出すようにするものとします。たとえば、キューに 30 個のメッセージが溜まっていたときは、「 $30 \div 10 = 3$ 」なので、3 回、SNS トピックの通知を出します。このとき図中⑤の Lambda 関数は 3 つ起動します。つまり、3 つの Lambda 関数で並列にキューから取り出す処理をすることで、より短い時間で処理が終わるようにします（図 6-3）。

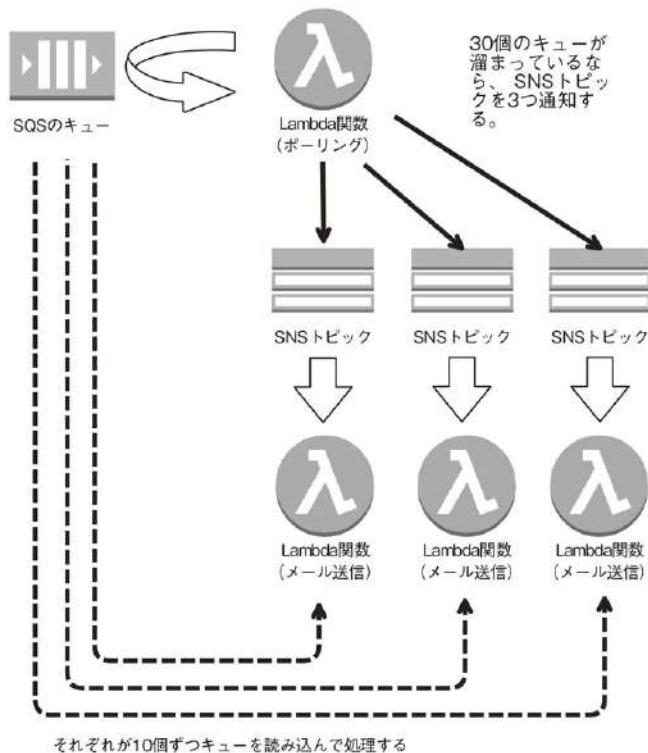


図 6-3 キューを複数の Lambda 関数で処理して並列実行性を高める

図中の⑥は、おまけの機能で、バウンスメールの処理です。バウンスメールを受け取ったときに SNS トピックに通知が送られるように構成し、その通知によって Lambda 関数を実行するように構成しておきます。Lambda 関数では、図中①のデータベースを変更することで、その宛先のメール送信に失敗したという記録を残します。

6-1-2 2回以上呼び出される可能性

SQS は、ベストエフォート型のキューイングシステムです。ほとんどの場合、登録したのと同じ順序で取り出せるのですが、希に、取り出し順序が変わったり、同じメッセージが 2 回以上、読

み出せてしまったりすることがあります。AWS のドキュメントでは「少なくとも 1 回、配信される」と表現されており、これは裏を返すと「2 回以上、配信されることがあるかもしれない」ということを意味します。

本システムでは、取り出し順序が変わることは問題ありませんが、2 回以上、読み出せてしまうと、その人に 2 通以上のメールが送信される恐れがあります。

そこで、「その宛先に、もう送信したか」などを記録しておいて、送信しているならば再度送信はしないというように、Lambda 関数のほうで、二重実行を避けるように実装します。

Λ Column FIFO キューを使う

SQS のキューには、「標準キュー」と「FIFO キュー」の 2 種類があります。もし、FIFO キューを使うなら、「順序が正しいことと、1 回しか取り出されないこと」を保障できます。

本当に確実性を求めるなら、FIFO キューを使うとよいでしょう。FIFO キューを使うには、SQS コンソールから設定するだけなので簡単です。しかしこの制限があります。

- 一部のリージョンしか対応していない

対応しているのは、本書の執筆時点において、「米国西部（オレゴン）」「米国東部（オハイオ）」「米国東部（バージニア北部）」「欧洲（アイルランド）」の 4 リージョンのみです。

- 動作が遅い

FIFO のアクセス速度は、API アクションごとに 1 秒当たり、300 件のトランザクションまでに制限されます。

6-1-3 メールの同報システム作成手順

本章では、図 6-2 に示したシステムを、次の順で作成していきます。

①DynamoDB テーブルを作る

「6-2 DynamoDB テーブルによるメールアドレス管理」では、宛先メールアドレスや氏名、そして、送信したかどうかやバウンスエラーがあったかどうかなどのフラグを管理する DynamoDB テーブルを作成します。

②S3 バケットと SQS を構成する

「6-3 S3 バケットと SQS を構成する」では、メールのタイトルと本文を記載したテキスト

● 6-2 DynamoDB テーブルによるメールアドレス管理

ファイルを配置する S3 バケットを作成します。

そして処理内容をキューイングするための SQS のキューを作成し、S3 バケットにファイルが置かれたときに Lambda 関数を実行するように構成して、そのキューにメッセージが溜まるところまでを作ります。

③キューから取り出してメールを送信する

「6-4 SQS からメッセージを取り出してメールを送信する」では、②でキューに溜められたメッセージを取り出し、SES を使ってメールを送信する処理を実装します。

④バウンスメールの処理

「6-5 バウンスメールを処理する」では、バウンスメールを SNS トピックの通知として受け取るように構成します。SNS トピックの通知を受けたときに、①DynamoDB テーブル上の該当メールアドレスに、バウンスエラーが戻ってきたというマークを付ける処理を実装します。

6-2 DynamoDB テーブルによるメールアドレス管理

まずは、メールアドレスを管理する DynamoDB テーブルを作成し、そこに宛先のメールアドレスを登録していきます。

6-2-1 メールアドレスを管理するテーブルの設計

「メールを送信する」という目的では、メールアドレスだけ管理すれば十分ですが、「氏名」の欄があると誰のメールアドレスなのか管理しやすくなりそうです。また、このシステムでは、システムの構成上「バウンスメールが戻ってきたか」と「(二重送信を防ぐための目的として) 送信済みかどうか」の 2 つの情報を管理する必要があります。

そこで本章で扱う事例では、表 6-1 に示すテーブル構造とします。テーブル名は、どのようなものでもかまいませんが、ここでは mailaddress テーブルと名付けます。

表 6-1 mailaddress テーブル

属性名	型	意味
email	String	プライマリーキー。宛先のメールアドレス
username	String	氏名

haserror	Number	配信の際にエラーがあったかどうかのフラグ。デフォルトは0。エラーメールが戻ってきたとき（後述）には1を設定する
issend	Number	送信済みかどうかのフラグ。送信前に0を設定し、送信完了したら1にする。同じユーザーに2回送信しないようにする目的で使う

6-2-2 セカンダリインデックスの設定

DynamoDBでは、プライマリーキーとして設定した項目以外で検索しようとすると、全件（全項目を対象とした）検索となるため、時間も費用もかかります。そこで検索の対象となりそうな属性は、セカンダリインデックスとして登録しておくのが望ましい方法です。表6-1で言うならば、`haserror`がこれに当たります。

メールを送信するときは、すでにバウンスされたことがわかっているメールアドレスに配信するのは無駄なので、バウンスされていないメールアドレスに限って——`haserror`が0であるものに限って——メールを送信するのが適切な対応です（SESはバウンス率が一定以上になると利用できなくなるため、そういう意味でも、到達不能なメールアドレス宛に送信すべきではありません）。

そこでここでは、`haserror`属性をセカンダリインデックスとして登録しておくものとします。

6-2-3 DynamoDB テーブルの作成

以上を踏まえて、DynamoDBコンソールを開き、表6-1に示したメールアドレスを管理する`mailaddress`テーブルを作成します。

◎ 操作手順 ◎ `mailaddress` テーブルを作成する

[1] テーブルを作成する

DynamoDBコンソールを開きます。そして【テーブルの作成】をクリックして、テーブルの作成を始めます（図6-4）。

● 6-2 DynamoDB テーブルによるメールアドレス管理



図 6-4 テーブルを作成する

[2] mailaddress テーブルを作成する

mailaddress テーブルを作成します。テーブル名には「mailaddress」と入力します(図 6-5)。

- ・プライマリーキーは「email」という名前にして、文字列型(String型)を選択します。
- ・セカンダリインデックスを設定したいので、【デフォルト設定の使用】のチェックを外します。



図 6-5 テーブル名とプライマリーキーを設定する

[3] セカンダリインデックスを設定する

図 6-6 の画面で、セカンダリインデックスの【インデックスの追加】ボタンをクリックしてインデックスを追加します。

- ・まずは、プライマリーキーの部分に、haserror 属性を数値型として登録します(図 6-7)。

- インデックス名は、プライマリーキーを入力すると、自動的に「プライマリーキー-index」という名前が付く（つまりこの例なら haserror-index）ので、この値にしておきます。
- 射影される属性は、このインデックスで検索したときに、取得したい属性です。ここでは「すべて」としますが、検索するときに一部の属性しか取得しないのでよいなら、いくつ制限することで、読み込み時のキャパシティーユニットを節約できます。
- [インデックスの追加] ボタンをクリックします。



図 6-6 セカンダリインデックスを追加する



図 6-7 haserror 属性をセカンダリインデックスとして追加する

[4] テーブルを作成する

読み込みキャパシティーユニットや書き込みキャパシティーユニットなど、その他の設定項目は、デフォルトのままにしておきます（図 6-8）。

- [作成] ボタンをクリックすると、テーブルが作成されます。

● 6-2 DynamoDB テーブルによるメールアドレス管理



図 6-8 テーブルを作成する

6-2-4 初期データを登録する

以上で mailaddress テーブルを作成できました。ここに、宛先のメールアドレスを登録していきます。

DynamoDB コンソールから、メールアドレスをひとつずつ追加していくこともできますが、操作性がよくないので、ひとつや二つの登録ならまだしも、数多くのメールアドレスを登録するには向いていません（図 6-9）。登録すべきデータが多い場合は、CSV 形式や Excel 形式などでデータを作っておき、それをまとめてインポートするのが現実的です。



図 6-9 ひとつずつ DynamoDB コンソールから登録するのは大変

■ DynamoDBにデータをインポートする方法

DynamoDBにデータをインポートするには、主に、次の3つの方法があります。

① DynamoDBにインポートするカスタムプログラムを作る

ひとつめの方法は、DynamoDBにインポートするプログラムをカスタムで作る方法です。たとえば、S3バケットにCSV形式ファイルやExcel形式ファイルを配置すると、それを読み取ってDynamoDBに1行ずつ登録するLambda関数を作るなどの方法が挙げられます。



Memo Excelファイルの読み込み用ライブラリ

`xlrd` (<http://www.pythont-excel.org/>) というPythonのライブラリを使うと、Excelファイルを読み込めます。

② S3との連携機能を使う

AWS Data Pipelineという機能を使うと、DynamoDBのデータをS3にエクスポートしたり、S3からインポートしたりできます。操作するには、DynamoDBコンソールのテーブル一覧で【アクション】メニューから選びます(図6-10)。

AWS Data Pipelineを使って初期データを登録できますが、インポートやエクスポートの機能は、どちらかと言うと、テーブルをバックアップしたり、リストアしたりするための機能です。データ構造はJSON形式であり、人が直接読み書きする形式としては、適切ではありません。



図6-10 AWS Data Pipelineを使ってインポートやエクスポート操作する

● 6-2 DynamoDB テーブルによるメールアドレス管理

③AWS CLI を使う

本書では、AWS を操作するのに、主に AWS マネジメントコンソールを使っていますが、「AWS CLI」というコマンドを使うと、コマンドラインから AWS を操作できます。

AWS CLI を使うには、インストールしたあと、アクセスキー／シークレットアクセスキーの設定が必要です。詳細は、Appendix D 「AWS CLI のインストール」を参照してください。

AWS CLI は「aws」というコマンド名で、「dynamodb」というパラメータを渡すと、DynamoDB を操作できます。たとえば Windows の場合、Windows PowerShell から次のように入力し、put-item という API を呼び出すと、mailaddress テーブルに対して項目（レコード）を追加できます（図 6-11）。

```
> aws dynamodb put-item --table-name mailaddress --item '{"email":{"S":"yamada@example.co.jp"}, "username":{"S":"山田太郎"}, "haserror": {"N":0}, "issend": {"N":0}}'
```



図 6-11 Windows PowerShell から aws コマンド (AWS CLI) で DynamoDB に項目を追加する

実際に上記のコマンドを入力してから、DynamoDB コンソールで確認してみると、確かに項目が登録されていることがわかります（図 6-12）。



email	haserror	issend	username
yamada@example.co.jp	0	0	山田太郎

図 6-12 AWS CLI の操作で追加された項目

Column シェルの種類によってエスケープ文字が異なる

シェルによって、文字列のエスケープ方法が異なります。本文では、Window PowerShellを使ってるので注意してください。

LinuxやMacの場合もWindows PowerShellの場合と同じですが、Windowsのコマンドプロンプトの場合は、次のように「""」ではなく「\"」のように表記します。

```
aws dynamodb put-item --table-name mailaddress --item "{\"email\": \"S\": \"yamada@example.co.jp\"}, \"username\": \"S\": \"山田太郎\"}, \"haserror\": \"N\": \"0\"}, \"issend\": \"N\": \"0\"}"}"
```

■ Excel表から実行するコマンドを作る

このようにawsコマンドを使ってDynamoDBに対してput-itemというAPIを呼び出すと、項目を追加できます。つまり、こうしたコマンドを登録したいメールアドレスの数だけ用意すれば、初期データの書き込みができます。

それにはいくつかの方法がありますが、比較的簡単なのが、Excelを使う方法です。

Excelシートに、次のように記載しておきます。

A列	メールアドレス
B列	氏名
C列	haserrorの値(0)
D列	issendの値(0)

このときE列以降に、図6-13に示すように数式を入れると、先に、Windows PowerShellで入力したawsコマンドの文字列をI列に得られます。

```
E列 = "" & A$1 & """:{"" & E$1 & """ : "" & A2 & """}"""
F列 = "" & B$1 & """:{"" & F$1 & """ : "" & B2 & """}"""
G列 = "" & C$1 & """:{"" & G$1 & """ : "" & C2 & """}"""
H列 = "" & D$1 & """:{"" & H$1 & """ : "" & D2 & """}"""
I列 ="aws dynamodb put-item --table-name mailaddress --item '{" & E2 & "," &
F2 & "," & G2 & "," & H2 & "}"
```

このI列の部分をコピーしてWindows PowerShellに貼り付ければ、それらをDynamoDBテーブルの項目として登録できます。件数が多いときには、データ登録用プログラムを作成して処理を

● 6-2 DynamoDB テーブルによるメールアドレス管理

行うべきですが、数百件程度のデータの登録であれば、こうした簡易な登録方法で、十分なはずです。

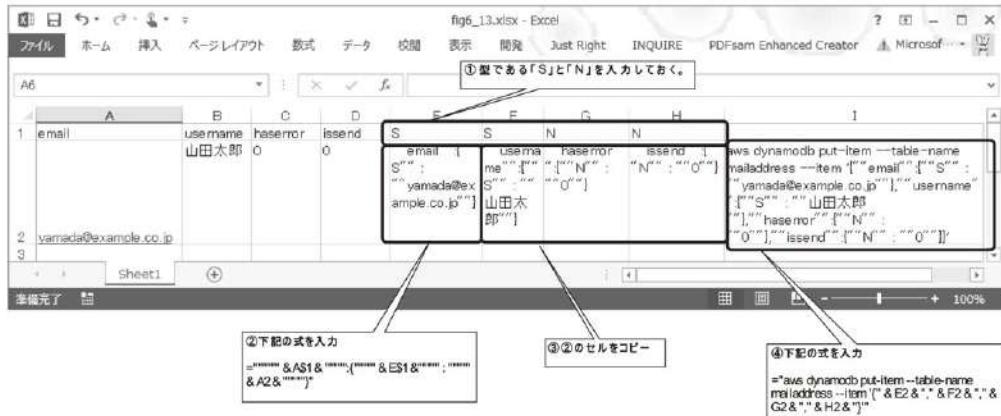


図 6-13 Excel シートを使って aws コマンドを作る

■ SES が提供しているテスト用メールアドレス

さて、これ以降では、実際にメールを送信する操作を行っていきます。SES を使ってメールを送信する場合、適当なメールアドレスに送信して動作テストするのは、望ましくありません。動作テストとはいえ、不特定多数に大量にメールを送信すると、迷惑メールを送信されていると判断され、SES を利用できなくなる恐れがあります。

とくに注意したいのが、バウンス率です。AWS のドキュメントには、バウンス率が一定以上になると、配信できなくなる旨が明記されているため、配信不能なメールアドレスの扱いは、慎重に行うべきです。

SES では、テスト用にメールボックスシミュレーターというメールアドレスが提供されています。これらのメールアドレスへの配信は、送信クォータ（1 日に送信できる総量）やバウンス率に影響を与えないで、テストの際にはこれらのメールアドレスを利用するようにします。用意されているのは、表 6-2 に示す 5 個のメールアドレスです。

表 6-2 メールボックスシミュレーターのアドレス

宛先	結果	意味
success@simulator.amazonaws.com	成功	正常な送信先。このメールアドレスに送信すると、正常に配信されたものとみなされる

bounce@simulator.amazonses.com	バウンス	このメールアドレスに送信すると、RFC3454に準拠するバウンスマールが返信される
ooto@simulator.amazonses.com	不在	このメールアドレスに送信すると、RFC3834に準拠する不在の際の自動応答メールが返信される
complaint@simulator.amazonses.com	苦情	このメールアドレスに送信すると苦情のメールが返信される
suppressionlist@simulator.amazonses.com	ハードバウンス	SESにおいてサプレッションリストという配信停止アドレスとして登録されており、このメールアドレス宛のメールは、(あとでバウンスされるのではなくて)、その場で配信に失敗する

■登録する初期データ

以上を踏まえて、DynamoDBに初期データを設定しましょう。ここでは、以下に示す計6個を登録することにします。

- ①自分のメールアドレス
- ②表6-2に示したテストアドレスすべて

Excelシートとして図6-14に示すを作り、このI列を、Windows PowerShellやMacやLinuxのシェルなどで実行してください。実行すると、図6-15のようにmailaddressテーブルに登録されるはずです。

	A	B	C	D	E	F	G	H	I
①	山田太郎	0	0	"email": "j.yamada@example.co.jp", "name": "山田太郎", "haserror": "N", "issend": "0"					aws dynamodb put-item --table-name mailaddress --item "[{"email": "j.yamada@example.co.jp", "useName": "1", "name": "山田太郎", "haserror": "N", "issend": "0"}]"
②	yamada@example.co.jp	0	0	"email": "j.yamada@example.co.jp", "name": "山田太郎", "haserror": "N", "issend": "0"					aws dynamodb put-item --table-name mailaddress --item "[{"email": "j.yamada@example.co.jp", "useName": "1", "name": "山田太郎", "haserror": "N", "issend": "0"}]"
③	success@simulator.amazonses.com	鈴木次郎	0	"email": "success@simulator.amazonses.com", "name": "鈴木次郎", "haserror": "N", "issend": "0"					aws dynamodb put-item --table-name mailaddress --item "[{"email": "success@simulator.amazonses.com", "useName": "1", "name": "鈴木次郎", "haserror": "N", "issend": "0"}]"
④	bounce@simulator.amazonses.com	田中三郎	0	"email": "bounce@simulator.amazonses.com", "name": "田中三郎", "haserror": "N", "issend": "0"					aws dynamodb put-item --table-name mailaddress --item "[{"email": "bounce@simulator.amazonses.com", "useName": "1", "name": "田中三郎", "haserror": "N", "issend": "0"}]"
									aws dynamodb put-item --table-name mailaddress --item "[{"email": "j.yamada@example.co.jp", "useName": "1", "name": "山田太郎", "haserror": "N", "issend": "0"}]"

図6-14 登録するメールアドレス

● 6-3 S3 バケットと SQS を構成する

email	haserror	issend	username
bounce@simulator.amazonaws.com	0	0	鈴木次郎
ooto@simulator.amazonaws.com	0	0	田中三郎
success@simulator.amazonaws.com	0	0	山田太郎
complaint@simulator.amazonaws.com	0	0	加藤四郎
suppressionlist@simulator.amazonaws.co	0	0	佐藤五郎

図 6-15 DynamoDB に登録されたところ

6-3 S3 バケットと SQS を構成する

S3 バケットにメール本文が置かれたとき、それを SQS にメッセージとして登録する処理を作っていきます（図 6-16）。まず、メールの本文を置くための S3 バケットを作ります。そして、その S3 バケットに対して、「ファイルが作成されたとき」に Lambda 関数を実行するように、トリガーを設定します。実行される Lambda 関数では、次の処理を行います。

①mailaddress テーブルのうち配信エラーを示す haserror 属性の値が 0 であるものを抽出し、それぞれの「メールアドレス」「ユーザー名」、そして「S3 バケット名」と「本文ファイルのパス名」を SQS にメッセージとして登録する。

②このとき、後続のメール送信処理のために、送信済みであることを示すフラグとなる issend を 0（未送信）に設定する

ここで登録されたメッセージは、「6-4 SQS からメッセージを取り出してメールを送信する」の処理で取り出して、メールを送信します。

メールアドレス、ユーザー名、本文が記されたS3バケットファイルのパス名

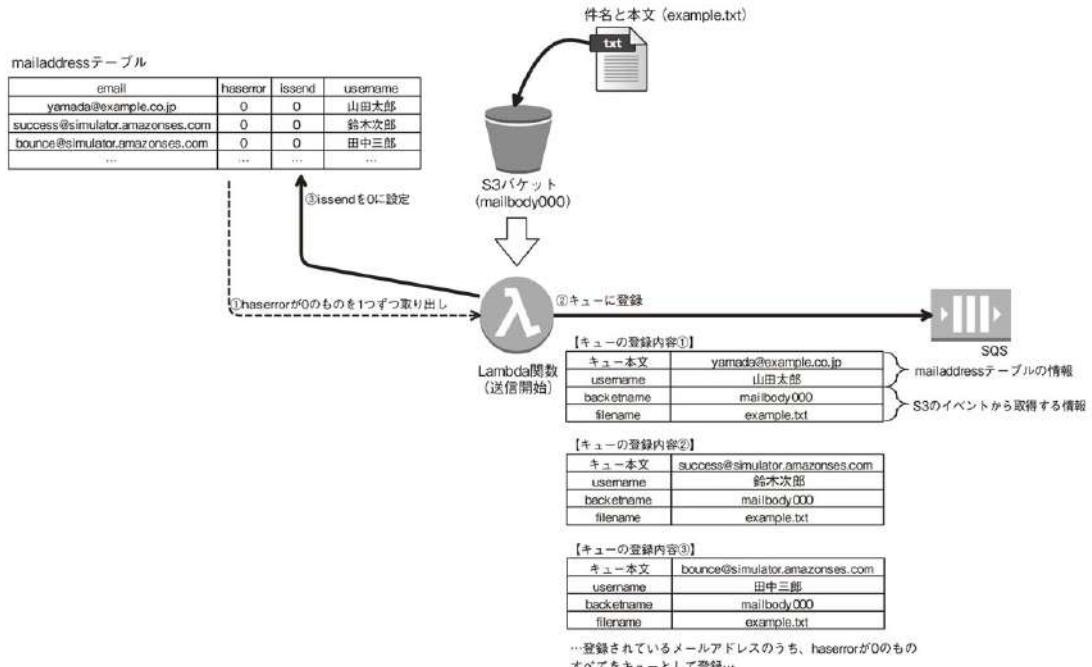


図 6-16 本節での処理の流れ

6-3-1 S3バケットの作成

まずは、S3バケットを作成しましょう。どのような名前でもかまいませんが、ここでは、「mailbody000」という名前のS3バケットを作ります（本書の執筆において、著者が「mailbody000」という名前を使っているので、読者の皆様は、同名のバケットを作れません。他の適当なバケット名で作成してください）。

◎ 操作手順 ◎ S3バケットを作成する

[1] S3コンソールを開く

S3コンソールを開き、[バケットを作成する]をクリックして、S3バケットを作成します。

● 6-3 S3 バケットと SQS を構成する



図 6-17 バケットを作成する

[2] バケット名やリージョンを設定する

バケット名やリージョンを設定します。まずは、「mailbody000」という名前のバケットを作成します。ここでは、「アジアパシフィック（東京）」に作成します（図 6-18）。



図 6-18 バケット名やリージョンを設定する

[3] プロパティの設定

バージョニングやログの記録、タグ付けなどを設定します。ここでは、とくに何も指定する必要はないので、そのまま次の画面に進みます（図 6-19）。



図 6-19 プロパティの設定

[4] アクセス許可の設定

アクセス許可を設定します。この S3 バケットには、Lambda 関数がアクセスします。

本書のこれまでの処理と同様に、以降の Lambda 関数を登録するときには、Lambda 関数の実行ロールを S3 バケットに対してフルアクセス権がある IAM ロールに設定します。つまり、ここで何も設定しなくても Lambda 関数からアクセスできる構成で利用します。この画面では、そのまま何も設定せず、次の画面に進みます（図 6-20）。



図 6-20 アクセス許可の設定

[5] 確認

確認画面が表示されます。[バケットの作成] をクリックして、バケットを作成します（図 6-21）。



図 6-21 確認

6-3-2 SQS のキューへのアクセス権の設定

次に、SQSに対する操作を行いますが、キューを新規に作成したり、読み書きしたりするためには、適切な権限が必要です。

開発者と Lambda 実行ロールの両方について、次のように設定しておきます。具体的な手順については、Appendix A ならびに Appendix B を参照してください。

①開発者（本書では devuser ユーザー）

開発者は SQS のキューを作成したり操作したりする権限が必要です。そこで、AmazonSQSFullAccess という管理ポリシーを設定します。

②Lambda 関数の実行ユーザー（本書では role-lambdaexec ロール）

以下で作る Lambda 関数を実行する場合、SQS への登録と取得できる権限が必要です。そこで開発者と同様に、AmazonSQSFullAccess 管理ポリシーを適用しておきます（図 6-22）。



図 6-22 開発者にも Lambda を実行する IAM ユーザーにも AmazonSQSFullAccess 管理ポリシーを適用しておく

6-3-3 SQS のキューを作成する

AmazonSQSFullAccess 管理ポリシーが適用されている開発者は、SQS にキューを作成できます。まずは、キューを作成しましょう。

キュー名は、どのようなものでもかまいませんが、ここでは「mailsendqueue000」という名前のキューを作成します。

◎ 操作手順 ◎ SQS のキューを作成する

[1] SQS コンソールを開く

AWS マネジメントコンソールで「SQS」を開き、SQS コンソールを開きます（図 6-23）。

● 6-3 S3 バケットと SQS を構成する

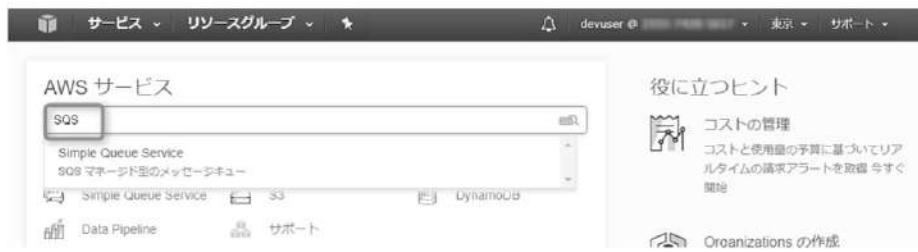


図 6-23 SQS コンソールを開く

[2] キューを作り始める

はじめてのときはウェルカム画面が表示されるので、[今すぐ始める] をクリックします（図 6-24）。



図 6-24 キューを作り始める

[3] キューの情報を入力する

キューの情報を入力します。ほとんどの値はデフォルト値でよいので、キュー名として「mailsend queue000」とだけ入力して [キーの作成] ボタンをクリックします（図 6-25）。

それぞれの設定の意味は、以下の通りです。

● キュー名

キューの名前です。どのような名前でもかまいませんが、ここでは「mailsend queue000」という名前とします。

● キューの属性

キューのタイムアウトやメッセージ保持期間などです。設定の意味の詳細は表 6-3 を参照してください。ここではデフォルト値のままとします。

表 6-3 キューの属性

設定項目	意味
デフォルトの可視性タイムアウト	キュー内メッセージの隠蔽時間
メッセージ保持期間	キューに保存する期間を設定する。メッセージがこの期間を超えても取り出されなかったときは、デッドレターキューに移動される
最大メッセージサイズ	メッセージの最大サイズ
配信遅延	キューに送信してから、取り出せるようになるまでの遅延時間を設定する
メッセージ受信待機時間	メッセージを受信するときの待機時間を設定する

新しいキューの作成

キューにどのような名前を付けますか。

キュー名

リージョン

詳細については、Amazon SQS FAQs よりも Amazon SQS 開発者ガイドを参照してください。
これらのデフォルトのパラメーターを変更できます。

キューの属性

デフォルトの可視性タイムアウト <input type="text" value="30"/> 秒	値は 0 秒 ~ 12 周期の間である必要があります。
メッセージ保持期間 <input type="text" value="4"/> 日	値は 1 分 ~ 14 日の間である必要があります。
最大メッセージサイズ <input type="text" value="256"/> KB	値は 1 ~ 256 KB の間である必要があります。
配信遅延 <input type="text" value="0"/> 秒	値は 0 秒 ~ 15 分の間である必要があります。
メッセージ受信待機時間 <input type="text" value="0"/> 秒	値は 0 秒 ~ 20 秒の間である必要があります。

デッドレターキュー設定

再処理ポリシーの使用

デッドレターキュー

最大受信数

キャンセル **キューの作成**

図 6-25 キューを作成する

● 6-3 S3 バケットと SQS を構成する

● デッドレターキュー設定

メッセージ保持期間をすぎても取り出されなかったキューを、別のキューに保存するための設定です。ここではデッドレターキューは設定しません。

これまでのひと通りの手順を終えると、キューが作成されます（図 6-26）。



図 6-26 キューが作成された

Λ Column デフォルトの可視性タイムアウト

SQSにおいては、キューからメッセージを取り出す操作をすると、そのメッセージを他のプログラムが重複して取り出してしまわないよう、一時的に隠されます。その隠される時間を設定します。この時間を持つると、取り出されたメッセージであっても、他のプログラムから見えるようになるので、たとえば、あるプログラムがメッセージを取り出した後にメッセージの削除をする前にハングアップした場合に、他のプログラムが、そのメッセージを再処理できます。この項目で設定する時間は、プログラムがメッセージの処理を完了する時間以上の、余裕を持った値を設定する必要があります。そうしないと、まだ処理中のメッセージを他のプログラムが参照してしまい、二重に処理してしまう恐れがあります。

6-3-4 SQSにメッセージを登録するLambda関数を作る

ここでは、メール本文のファイルがS3バケットに置かれたときに、送信すべきメールアドレスなどを記載したメッセージを、SQSのキューとして登録するプログラムを作ります。このプログラムは、「S3にファイルが作られたとき」をトリガーとして実行するため、第4章で作成したサンプルと同様に、「s3-get-object-python3」の設計図から作成します。

◎操作手順 ◎ SQSにキューを登録するLambda関数を作る

[1] Lambda関数を作り始める

Lambdaコンソールを開き、ダッシュボードの【関数の作成】ボタンをクリックして、Lambda関数を作り始めます（図6-27）。



図 6-27 Lambda関数を作り始める

[2] s3-get-object-python3を選択する

検索ボックスに「s3-get-object-python3」と入力し、設計図を選択します（図6-28）。



図 6-28 s3-get-object-python3 設計図から作り始める

[3] S3 バケットとイベントの種類を選択する

トリガーとして「S3」が選択された状態で表示されるので、[バケット] の部分で、対象のバケットを選択します（図 6-29）。

- ・先に作成しておいた「mailbody000」のバケットを選択します。そして、この S3 バケットにファイルが置かれたときに実行したいので、「オブジェクトの作成（すべて）」を選択します。
- ・ префиксとサフィックスは設定せず、どんな名前のファイルが置かれたときでも Lambda 関数を呼び出すようにするものとします。
- ・ この画面の下にはさらに、[トリガーの有効化] があります。最初は有効化せずに、あとで有効化するのがセオリーですが、ここでは話を簡単にするためと、S3 バケットから起動する Lambda 関数の作成は第 4 章ですでに行っていてある程度、挙動がわかっているという点から、チェックを付けて最初からトリガーを有効化してしまうことにします（図 6-30）。そうすれば、Lambda 関数を登録した後は、すぐに、このイベントが発生するようになります。
- ・ [次へ] ボタンをクリックします。



図 6-29 バケット名とイベントタイプの設定



図 6-30 トリガーを有効化する

【3】関数名やコードなどを入力する

- Lambda関数の名前や説明、ランタイムなどを入力します。どのような名前でもかまいませんが、ここでは「sendqueue」という名前とします（図6-31）。



図 6-31 関数名を付ける

- 次に、下方向にスクロールして、Lambda関数のコードを入力します（図6-32）。ここでは、リスト6-1に示す内容を入力します。このプログラムの動きについては「6-3-6 SQSにメッセージを登録するプログラムの仕組み」で説明します。
- その下には「暗号化ヘルパーを有効にする」と「環境変数」の入力欄がありますが、どちらも未記入とします。

● 6-3 S3 バケットと SQS を構成する



図 6-32 Lambda 関数を入力する

リスト 6-1 入力する Lambda 関数

```

import json
import urllib.parse
import boto3
from boto3.dynamodb.conditions import Key, Attr

def lambda_handler(event, context):
    # ①DynamoDBのmailaddressテーブルを操作するオブジェクト
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('mailaddress')

    # ②SQSのキューを操作するオブジェクト

```

```

sq = boto3.resource('sqs')
queue = sq.get_queue_by_name(QueueName="mailsendqueue000")

for rec in event['Records']:
    # ③S3に置かれたファイルパスを取得
    bucketname = rec['s3']['bucket']['name']
    filename = rec['s3']['object']['key']

    # ④haserrorが0のものをmailaddressテーブルから取得
    response = table.query(
        IndexName='haserror-index',
        KeyConditionExpression=Key('haserror').eq(0)
    )

    # ⑤上記の1件1件についてループ処理
    for item in response['Items']:
        # ⑥送信済みを示すissendを0にする
        table.update_item(
            Key={'email' : item['email']},
            UpdateExpression="set issend=:val",
            ExpressionAttributeValues= {
                ':val' : 0
            }
        )
        # ⑦SQSにメッセージとして登録する
        sqsresponse = queue.send_message(
            MessageBody=item['email'],
            MessageAttributes={
                'username' : {
                    'DataType' : 'String',
                    'StringValue' : item['username']
                },
                'bucketname' : {
                    'DataType' : 'String',
                    'StringValue' : bucketname
                },
                'filename' : {
                    'DataType' : 'String',
                    'StringValue' : filename
                }
            }
        )
        # 結果をログに出力しておく
        print(json.dumps(sqsresponse))
    
```

- ・最後に、ハンドラ名やロールなどを設定します（図 6-33）。

● 6-3 S3 バケットと SQS を構成する

● ハンドラ

デフォルトの「lambda_function.lambda_handler」のままとしてください。

● ロール

【既存のロールを選択】を選択します。すると、【既存のロール】でロールが選べるようになるので、「role-lambdaexec」を選択してください。

- ・ そのほか、「タグ」と「詳細設定」の項目がありますが、デフォルトのまま何も変更しないものとします。
- ・ [次へ] ボタンをクリックします。



図 6-33 ハンドラ名やロールの入力

【6】 確認

- ・ 最後に確認のメッセージが表示されます。【関数の作成】ボタンをクリックしてください（図 6-34）。



図 6-34 Lambda 関数の確認と完成

Λ Column まとめて登録してコストを削減する

リスト 6-1 はわかりやすさを優先したコードであり、パフォーマンス的に優れたものではありません。リスト 6-1 では 1 件ずつ SQS にメッセージを登録していますが、実は、`send_message` 関数の代わりに `send_message_batch` 関数を使えば、最大 10 件のメッセージをまとめて登録できます。SQS は API 呼び出し回数も課金対象であるため、関数の呼び出し数を少なくすると、コスト削減にも貢献します。

6-3-5 動作テスト

以上で Lambda 関数の作成が終わりました。以下では、動作テストを行います。

■メール本文ファイルを配置する

メールの送信テスト用に、S3 バケットにメールの本文となるファイルを配置します。たとえば、次のような内容の「example.txt」とします。ここでは、1 行目がタイトル、2 行目が空きで、3 行目以降が本文とします。また、文字コードは「UTF-8」とします。

【本文の内容テキストファイル（example.txt。文字コードは「UTF-8」）】

メール配信のテスト

このメールはテストです。
テスト配信をしております。

このファイルを先ほど作成した mailbody000 バケットに配置しましょう。これで Lambda 関数が動いたはずです（図 6-35）。

● 6-3 S3 バケットと SQS を構成する



図 6-35 mailbody000 バケットに example.txt をアップロードする

■ キューの内容を確認する

リスト 6-1 に示した Lambda 関数の処理が実行されたら、SQS にメッセージが登録されるはずです。SQS コンソールを開いて mailsendqueue000 を開くと、「利用可能なメッセージ」が「5」になっていることがわかります（図 6-36）。この「5」という値は、mailaddress テーブルに登録したメールアドレスのうち、haserror が 0 である項目の数です（前掲の図 6-15 を参照）。もし mailaddress テーブルに、本書で説明している以外の項目を追加しているなら、この数は異なったものになります。



図 6-36 キューの一覧を確認する

[キュー操作] メニューから [メッセージの表示／削除] をクリックすると、キューの中身を確認できます（図 6-37）。



図 6-37 キューの中身を確認する

図 6-38 のメッセージが表示されるので [メッセージのポーリングを開始] をクリックします。すると、メッセージの内容一覧が表示されます（図 6-39）。

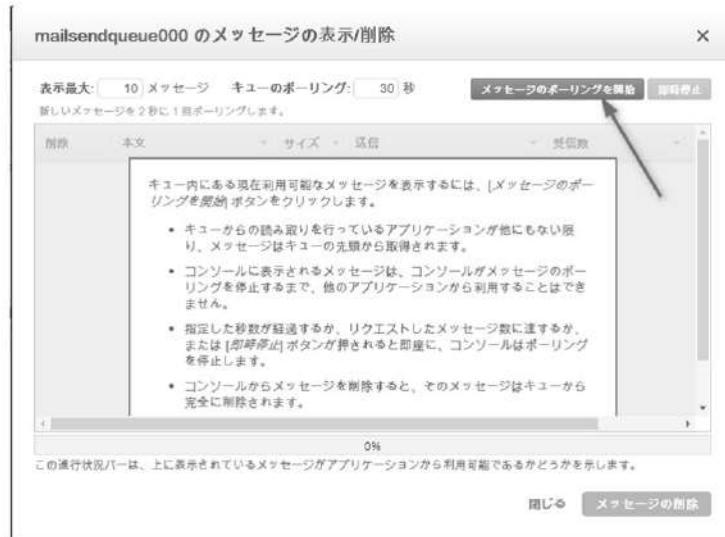


図 6-38 メッセージのポーリングを開始する

削除	本文	サイズ	送信	受信数	詳細
■	success@simulator.amazon.com	31 バイト	2017-08-15 21:20:41 GMT+09:00	1	詳細
■	coto@simulator.amazonaws.com	28 バイト	2017-08-15 21:20:40 GMT+09:00	1	詳細
■	bounce@simulator.amazon.com	30 バイト	2017-08-15 21:20:40 GMT+09:00	1	詳細
■	suppressionlist@simulator.amazon.com	99 バイト	2017-08-15 21:20:41 GMT+09:00	1	詳細
■	complaint@simulator.amazon.com	33 バイト	2017-08-15 21:20:41 GMT+09:00	1	詳細

図 6-39 メッセージのポーリング結果

● 6-3 S3 バケットと SQS を構成する

各メッセージの右側にある [詳細] をクリックするとメッセージの詳細情報を参照できます（図 6-40、図 6-41）



図 6-40 メッセージの詳細①（本文）



図 6-41 メッセージの詳細②（属性）

6-3-6 SQSにメッセージを登録するプログラムの仕組み

Lambda関数の動作を確認できたので、リスト6-1に提示したLambda関数の仕組みを、順を追って見ていきます。

①DynamoDBのテーブルを操作するオブジェクトの取得

まずは、このLambda関数から各種操作するためのオブジェクトを取得します。

ひとつは、DynamoDBのテーブルにアクセスするためのオブジェクトです。mailaddressテーブルにアクセスするため、次のようにしてTableオブジェクトを取得しています。これは第5章で説明した方法と同じです。

```
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('mailaddress')
```

②SQSのキューを操作するオブジェクト

同様にして、SQSのキューを操作するオブジェクトを取得します。まずSQSオブジェクトを取得し、それからget_queue_by_nameメソッドを使うと、指定した名前のキューを取得できます。ここでは「mailsendqueue000」という名前のキューを取得しています。

```
sqs = boto3.resource('sqs')
queue = sqs.get_queue_by_name(QueueName="mailsendqueue000")
```

③S3バケットに置かれたオブジェクトパスの取得

次に、S3バケットに置かれたテキストファイルを取得して処理していきます。第4章で説明したように、S3バケットに置かれたファイル群は、イベント引数のRecords要素に含まれているので、次のようにループ処理します。

```
for rec in event['Records']:
    ...置かれたファイルの処理...
```

置かれたファイルのバケット名とファイルのパス名（オブジェクト名）を、次のようにして取得します。

```
# S3に置かれたファイルパスを取得
bucketname = rec['s3']['bucket']['name']
filename = rec['s3']['object']['key']
```

④条件に合致するレコードの取り出し

次に、mailaddressテーブルからhaserror属性が0である項目を取り出します。前掲の図6-7に

● 6-3 S3 バケットと SQS を構成する

示したように、`haserror` 属性はセカンダリインデックスとして構成しており、「`haserror-index`」という名前を付けました。このようにしておくと、下記のようにして、`haserror` が 0 であるものだけに絞り込んで取り出せます。

```
response = table.query(  
    IndexName='haserror-index',  
    KeyConditionExpression=Key('haserror').eq(0)  
)
```

「`eq`」は等しいことを示すメソッドですが、それ以外に、表 6-4 に示す大小比較のメソッドを使えます。

表 6-4 評価に使える比較メソッド

比較メソッド	意味
<code>eq</code>	等しい
<code>le</code>	小さいか等しい
<code>lt</code>	小さい
<code>ge</code>	大きいか等しい
<code>gt</code>	大きい
<code>begins_with</code>	指定した文字列から始まる
<code>between</code>	ある値以上、ある値以下

⑤項目の取り出し

④で取得した項目は、次のようにループ処理すると、すべて取り出せます。

```
for item in response['Items']:  
    ...項目に関する処理…
```

⑥テーブルを更新する

この処理では、送信前に `issend` 属性を「0」に設定します。第 5 章でもシーケンス番号の更新のときに使った `update_item` メソッドを使って、`issend` 属性を 0 に設定します。

```
table.update_item(  
    Key={'email' : item['email']},  
    UpdateExpression="set issend=:val",  
    ExpressionAttributeValues={  
        ':val' : 0  
    }  
)
```

)

⑦SQSにメッセージとして登録する

そして、SQSにメッセージを登録します。SQSへのメッセージは、次の2種類のデータを指定できます。

- 本文（ボディ）
- 属性（アトリビュート）

ここでは本文として宛先のメールアドレスを、属性として「宛先の氏名」「S3バケット名」「S3に置かれたファイル名（オブジェクト名）」を設定しました。

```
sqsresponse = queue.send_message(  
    MessageBody=item['email'],  
    MessageAttributes={  
        'username' : {  
            'DataType' : 'String',  
            'StringValue' : item['username']  
        },  
        'bucketname' : {  
            'DataType' : 'String',  
            'StringValue' : bucketname  
        },  
        'filename' : {  
            'DataType' : 'String',  
            'StringValue' : filename  
        }  
    }  
)
```

この処理によって、先の図6-36などで見たように、キューにメッセージが登録されます。

6-4 SQSからメッセージを取り出してメールを送信する

前節までの操作でSQSのキューにメッセージとして登録されました。次に、SQSからこのメッセージを取り出してメールを送信するプログラムを作ります。

SQSのキューに登録されたメッセージを取得する場合、「届いたときにLambda関数が呼び出される」という構成にはできません。代わりに、Lambda関数側から、一定時間ごとにポーリングするのが基本です。いくつかの方法がありますが、代表的な方法は、第3章で説明したスケジュー

● 6-4 SQS からメッセージを取り出してメールを送信する

ルータスクを使う方法です。たとえば、5分ごとなどに Lambda 関数を実行し、キューにメッセージがあるかどうかを調べ、メッセージがあれば処理します（図 6-42）。

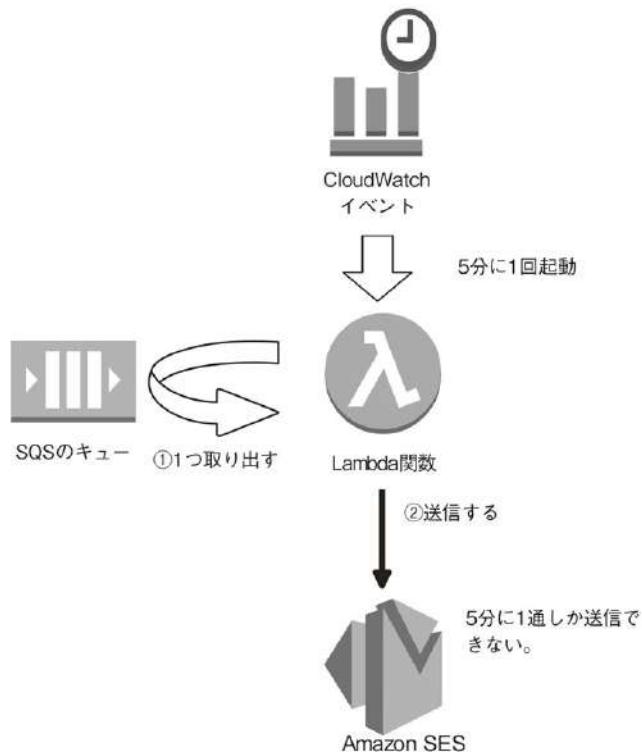


図 6-42 Lambda 関数でキューをポーリングする

ただし、この考え方方が基本であるものの、「メールを送信する」ということを考えた場合、ポーリングしたプログラムで1通ずつメールを送信していると、全部のメールを送信するのに、とても時間がかかってしまいます。そこでポーリングするプログラムとは別に、メールを送信する別のプログラムを用意し、メールの送信は、そちらのプログラムに任せるようにします。つまり、溜まっているメッセージの数だけ、送信用のプログラムを起動するのです。そうすれば、同時に複数のメールを送信できるので、素早くたくさんのメールを送信できます。

しかし、あまりにたくさんのプログラムを起動すると、今度は、費用が嵩みます。そこでメール送信用のプログラムは「10通単位」で処理することにします。たとえば、30個のメッセージが溜まっていたとすると、ポーリングするプログラムは、それを10個ずつの束に分けて、3個の送信用プログラムを起動するようにします（図 6-43）。

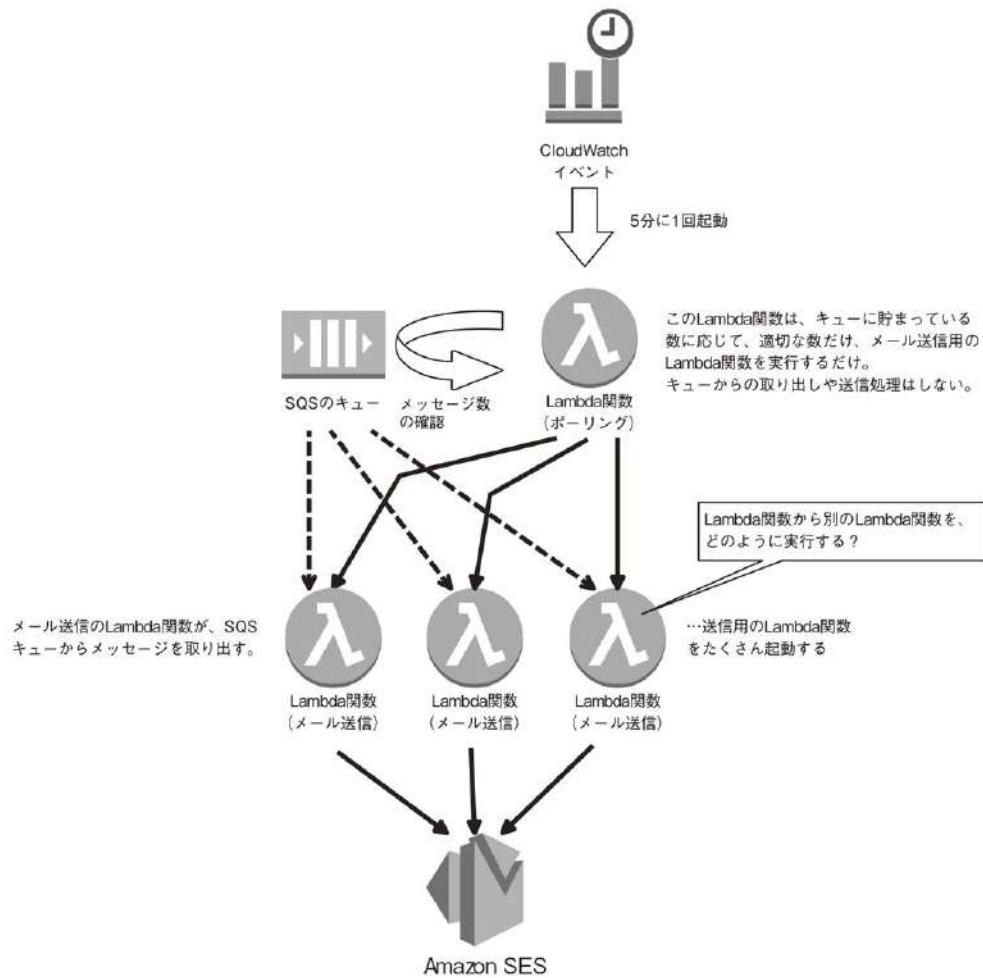


図 6-43 ポーリングする処理とメールを送信する処理に分ける

メールを送信するプログラムも、Lambdaで作ります。このとき考慮しなければならないのは、「Lambda関数から別のLambda関数を呼び出すには、どうすればよいのか」という点です。

その答えは、SNSトピックです。SNSトピックは、AWS上の「通知」をするサービスです。SNSトピックに通知が届いたときにLambda関数を実行できるので、そのLambda関数で実際にメールを送信する処理をします（図6-44）。

● 6-4 SQS からメッセージを取り出してメールを送信する

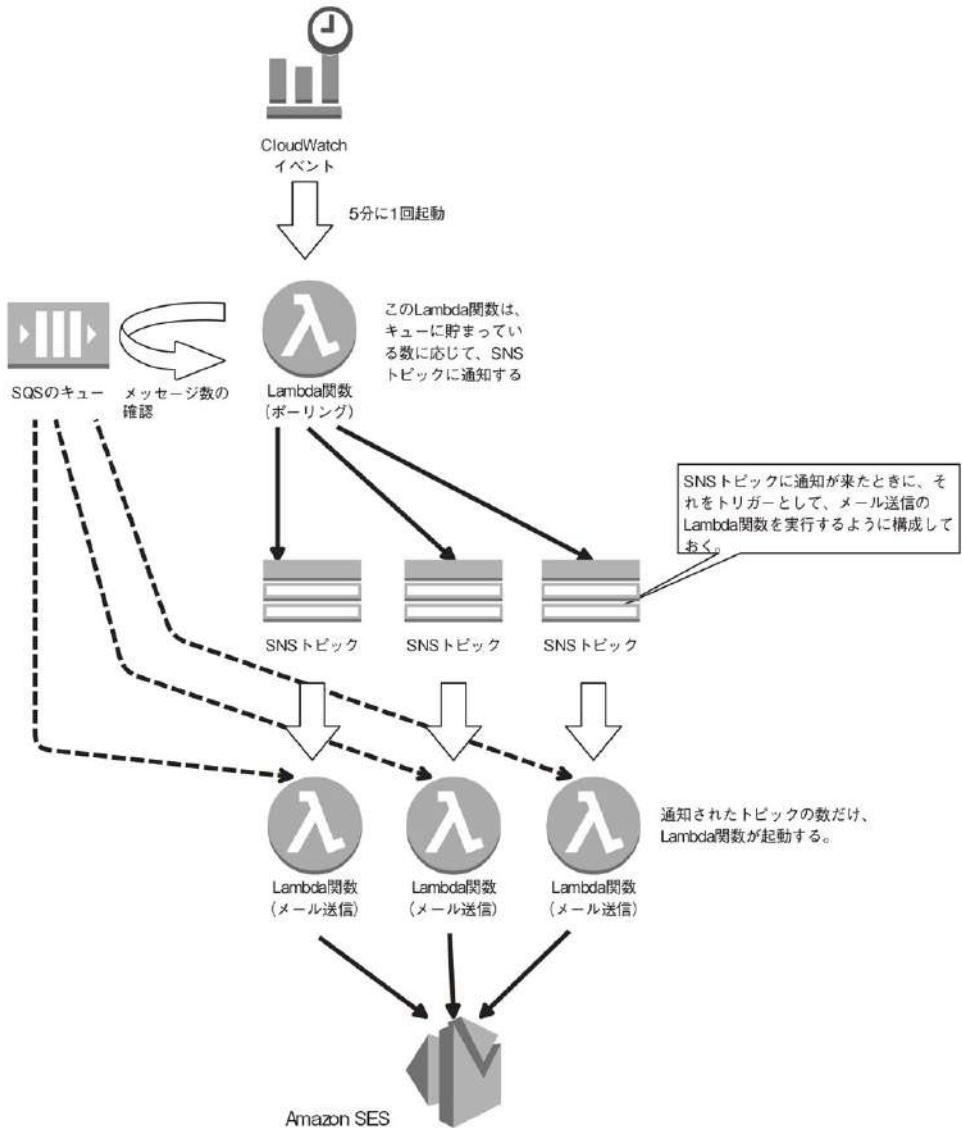


図 6-44 ポーリング処理から SNS トピックのメッセージを作り、その通知を受けた Lambda 関数がメールを送信する

図 6-44 からわかるように、本章では、2 つの Lambda 関数を作ります。ひとつは「ポーリングする Lambda 関数」、もうひとつは「メールを送信する Lambda 関数」です。また話を簡単にするため、SNS では「通知を出すだけ」として、実際のメッセージの取り出しは、メールを送信する側の Lambda 関数で実行するものとします。

■メッセージ取得の考え方

図6-44の構成では、ポーリングするLambda関数は、「溜まっているメッセージの数÷10（小数以下は切り上げ）の数だけ、メールを送信するLambda関数を起動する」という処理しかしていません。キューからメッセージを取り出すのは、メールを送信するLambda関数です。そのため、どのLambda関数が、キューのどのメッセージを取り出すのかは不定です。それに、全部取り出し切るとも限りません。たとえば、30通のメッセージが溜まっている場合、ポーリングするLambda関数は、3個のSNSトピックを通知します。すると、メールを送信するLambda関数が3つ並列して起動します。それぞれのLambda関数は、キューからメッセージを10個ずつ読み込んで処理しますが、タイミングによっては10個取り出せないかもしれません。つまり、1回の処理でメッセージすべてを取り出し切るとは限らないということです。その場合、メッセージはいくつかキューに残ることになりますが、5分後には、またポーリングのプログラムが実行されるため、取り残されたメッセージは、そのときには処理されるはずです。

このように、この節で説明する処理は、確実にメッセージを取得するわけではなく、取り残しても、次の処理で拾えればよいという考え方で作成しています。確実に、5分単位ですべてのメッセージを取り出し切らなければならないときは、また設計が違うので注意してください。

6-4-1 SNSトピックを操作する権限

SNSトピックを操作するには、適切な権限が必要です。ここでは、開発者とLambdaの実行ロールの両方に、AmazonSNSFullAccessの管理ポリシーを適用しておくものとします（図6-45）。具体的な手順については、Appendix AならびにAppendix Bを参照してください。



図6-45 AmazonSNSFullAccessを設定しておく

● 6-4 SQS からメッセージを取り出してメールを送信する

6-4-2 SNS トピックの作成

まずは、ポーリングする Lambda 関数から、メール送信する Lambda 関数を呼び出すために使う、SNS トピックを作成します。

◎ 操作手順 ◎ SNS トピックを作る

[1] SNS トピックを作る

SNS コンソールを開き、ダッシュボードから [Create topic] をクリックします（図 6-46）。

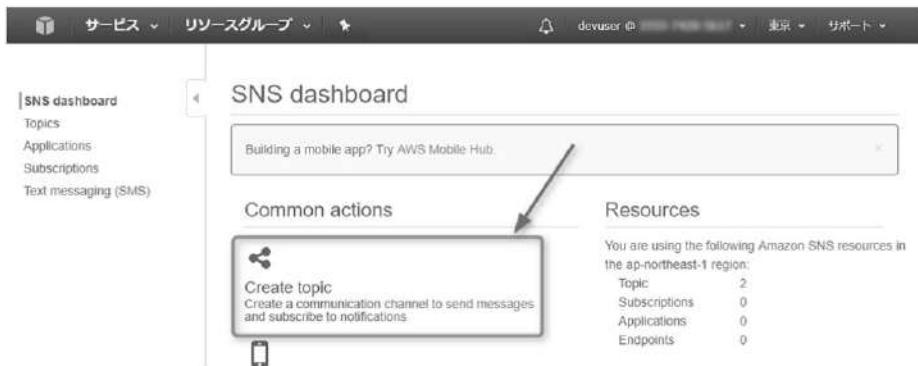


図 6-46 SNS トピックを作る

[2] トピック名を決める

トピック名 (Topic name) と表示名 (Display name) を決めます。ここでは仮に「sendmail-notifications」とします。表示名は必須ではないので、空欄にしておきます（図 6-47）。

The screenshot shows the 'Create new topic' dialog box. At the top, there's a 'Topic name' input field containing 'sendmail-notifications'. Below it is a 'Display name' input field with the placeholder 'Enter topic display name. Required for topics with SMS subscriptions.' At the bottom right of the dialog, there are 'Cancel' and 'Create topic' buttons, with the 'Create topic' button being highlighted with a red box and an arrow pointing to it from the top right.

図 6-47 トピック名を設定する

トピックの名前には、以下の書式の「トピック ARN」という名前が付けられます。

書式 arn:aws:sns:リージョン名:アカウントID:トピック名

トピック ARN は、通知を送信する際の宛先となるので、確認しておきましょう（図 6-48）。

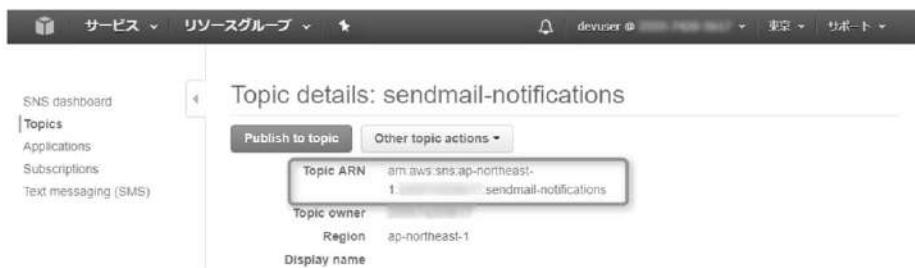


図 6-48 トピック名を確認しておく

6-4-3 SQSを処理する Lambda 関数を作る

SQS を 5 分ごとにポーリングするための Lambda 関数を作ります。第3章でも CloudWatch イベントを使って Lambda 関数を作りましたが、ここでもそれと同様にして作成します。ここでは、mailworker という名前の Lambda 関数を作ります。

◎ 操作手順 ◎ 5分間隔で SQS を処理する Lambda 関数を作る

[1] lambda-canary-python3 から作成する

Lambda コンソールを開き、lambda-canary-python3 を設計図として作成します（図 6-49）。



図 6-49 lambda-canary-python3 設計図から開く

● 6-4 SQS からメッセージを取り出してメールを送信する

[2] スケジュールを作成する

- [ルール] で、「新規ルールの作成」を選択して、新しくルールを作成します（図 6-50）。
- ・ルール名は何でもかまいませんが、ここでは「mailcron」とします。
 - ・5 分単位で実行したいので、スケジュール式を選択し、「cron(0/5 * * * ? *)」と入力してください。
 - ・すぐに実行したくはないので、トリガーの有効化には、チェックを付けないでください。

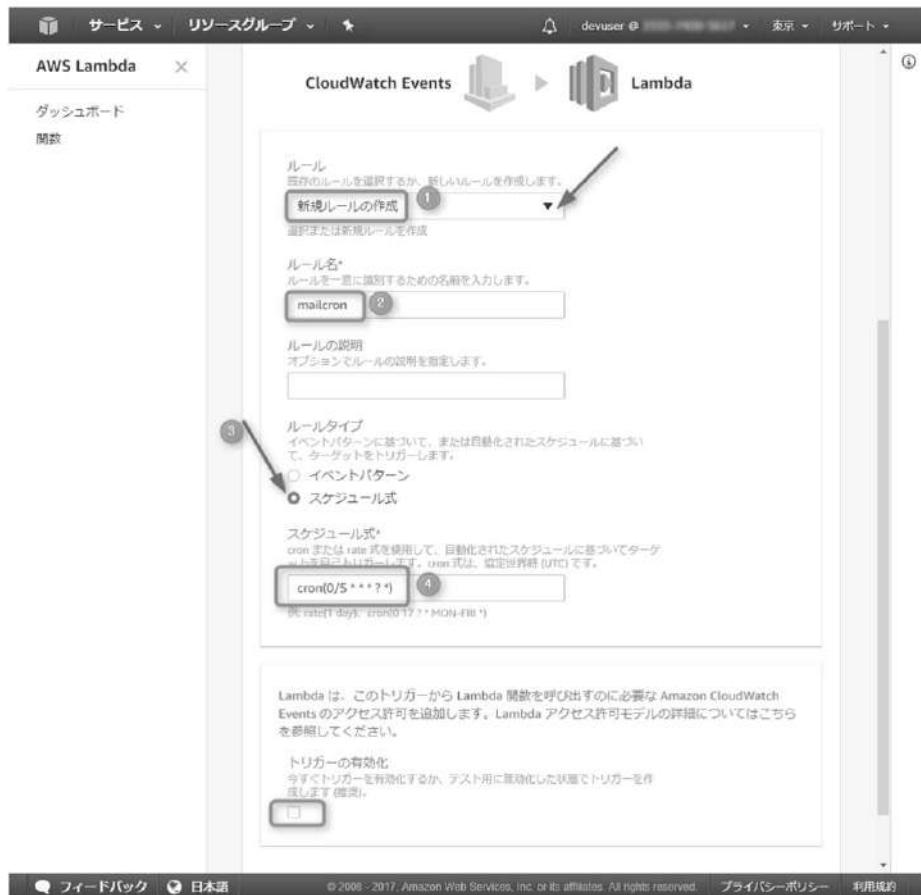


図 6-50 スケジュールを設定する

[3] 関数名の設定

関数名を設定します。ここでは mailworker という名前にします（図 6-51）。



図 6-51 関数名を設定する

関数のコードを入力します。ここでは、リスト 6-2 のように入力します。なお、10 行目の

```
sns.Topic('arn:aws:sns:ap-northeast-1:XXXXXXXXXXXX:sendmail-notifications')
```

の部分は、SNS トピックの宛先となります。図 6-48 で確認したトピック ARN に合わせてください。

このプログラムの動きについては、「6-4-5 SQS の内容を確認して SNS トピックに通知する仕組み」で説明します。

リスト 6-2 SNS トピックを送信する例

```
import json
import boto3

# SQSのキューを取得
sns = boto3.resource('sns')
queue = sns.get_queue_by_name(QueueName="mailsendqueue000")

# SNSトピックを取得
topic = sns.Topic('arn:aws:sns:ap-northeast-1:XXXXXXXXXXXX:sendmail-notifications')

def lambda_handler(event, context):
    # キューの待ち数を確認する
    n = queue.attributes['ApproximateNumberOfMessages']
```

● 6-4 SQS からメッセージを取り出してメールを送信する

```
# 10単位でSNSトピックに通知する
for i in range(int((int(n) + 9) / 10)):
    topic.publish(
        Message='mailsendqueue000',
    )
```



図 6-52 関数のコードを入力する

Lambda 関数ハンドラやロールを設定します。ハンドラはデフォルトのままとし、ロールは、role-lambdaexec を選択します。「タグ」や「詳細設定」はデフォルトのままとします（図 6-53）。



図 6-53 関数ハンドラやロールを設定する

最後に確認画面が表示されるので、[関数の作成]をクリックして関数を作成します(図6-54)。



図6-54 関数の確認

6-4-4 動作テスト

図6-54のLambda関数を実行すると、もし、SQSにメッセージが溜まっていたなら、SNSに通知が送信されます。SNSに通知が届いたときに「何をするのか」という規定のことをサブスクリプションと言います。

■サブスクリプションをメールアドレスとして仮登録する

のちの手順では、サブスクリプションとしてLambda関数を設定することで、「通知が届いたときにLambda関数を実行する」という設定をしていきますが、この段階では、まだ設定していないので、通知が届いても何も起きず、通知が届いたかどうかを確認できません。そこで、ここでの処理ではサブスクリプションとして「メールアドレス」を設定しておきます。こうすることで、SNSに通知が届いたときにメールとして受け取れます。

◎操作手順 ◎ 通知が届いたときにメールを受け取れるようにする

[1] サブスクリプションを登録する

トピックの設定画面に開き、[Topics]タブの[Create subscription]ボタンをクリックします。

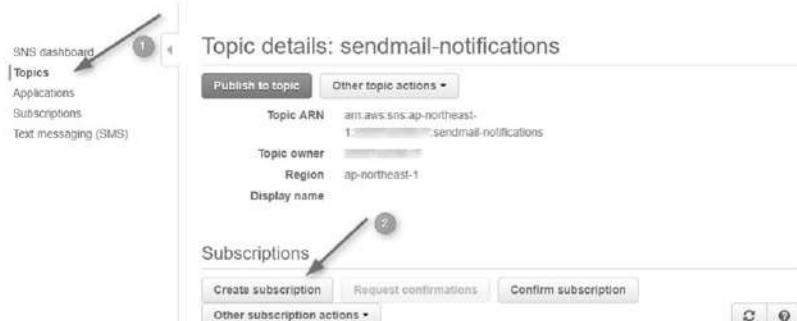


図6-55 サブスクリプションの登録を始める

● 6-4 SQS からメッセージを取り出してメールを送信する

[2] メールで配信するようにする

- ・ [Protocol] に [Email] を選択して、メールアドレスを登録します（図 6-56）。
- ・ [Create subscription] をクリックします。

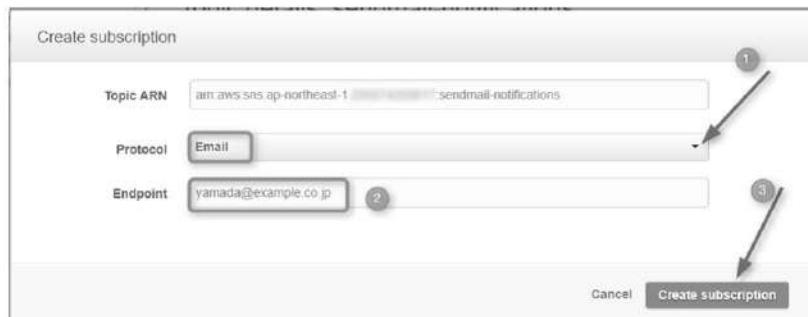


図 6-56 メールアドレスを登録する

[3] 登録する

すると Subscription ID が「PendingConfirmation」として登録されます（図 6-57（1））。

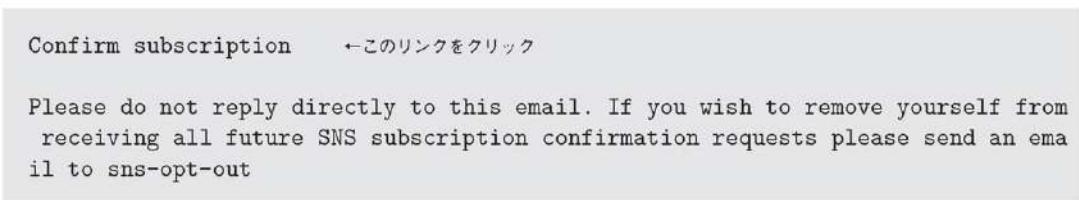
Subscriptions	
Create subscription Request confirmations Confirm subscription	
Other subscription actions ▾	
Filter	
Subscription ID	Protocol
PendingConfirmation	email

図 6-57（1） PendingConfirmation として登録されたところ（このステータスではまだメールは届かない）

このとき、入力したメールアドレス宛に、次のような確認メールが届くので、そのメールのリンクをクリックします。

【受け取るメールの例】

```
You have chosen to subscribe to the topic:  
arn:aws:sns:ap-northeast-1:XXXXXXX:sendmail-notifications  
  
To confirm this subscription, click or visit the link below (If this was in error or no action is necessary):
```



すると Subscription ID が割り当てられ（図 6-57（2））、以降、SNS トピックを受け取ったときに、このメールアドレス宛に受け取るようになります。

Subscription ID	Protocol
arn:aws:sns:ap-northeast-1:...:sendmail-notifications	email

図 6-57（2） Subscription ID が設定された

■キューに溜まった状態で Lambda 関数をテスト実行する

SNS トピックの通知を受け取る準備ができたところで、SQS のキューにメッセージが溜まっているときに実行すると、本当に SNS トピックとして通知されるのかを確認してみましょう。

ここでは、前掲の図 6-39 のようにキューにメッセージが溜まっているものとします。このとき、ポーリングする Lambda 関数の [テスト] をクリックしてテスト実行しましょう（図 6-58）。

Lambda > 関数 > mailworker
ARN - arn:aws:lambda:ap-northeast-1:...:function:mailworker

mailworker

①

おめでとうございます。Lambda 関数「mailworker」が正常に作成され、無効化された状態でトリガーとして設定されました。トリガーを有効化する前に、関数の動作をテストすることをお勧めします。

図 6-58 テスト実行する

すると、「キューに溜められた数 ÷ 10（小数以下切り上げ）」だけ、SNS トピックに通知が発行されるはずです。

先ほどの操作で、SNS トピックに通知が届くときにはメールとして送信されるはずなので、メー

● 6-4 SQS からメッセージを取り出してメールを送信する

ルを確認してみてください。次の内容のメールが届いていることがわかるはずです。

【SNS トピックの通知として届く内容】

```
mailsendqueue000

-- 
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.ap-northeast-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:ap-northeast-1:XXXXXXXXXXXX:sendmail-notifications:86724717-ac9b-44d8-9228-8db87250d4fb&Endpoint=yamada@example.co.jp

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support
```

確認できたら、SNS トピックに設定したメールアドレスへの通知は解除しておきましょう。SNS コンソールの [Subscriptions] から、該当のサブスクリプションにチェックを付け、[Actions] → [Delete subscriptions] をクリックすることで削除できます（図 6-59）。

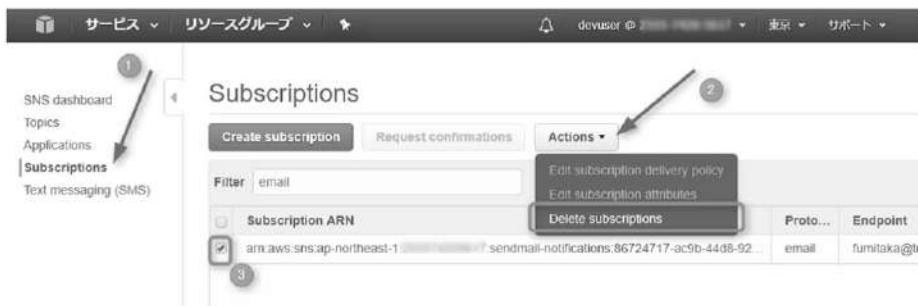


図 6-59 メールでの受け取りを解除する

そしてこのポーリングをする Lambda 関数のトリガーを有効に設定しておきましょう。以上で 5 分単位で、キューをポーリングする Lambda 関数が動くようになりました（図 6-60、図 6-61）。



図 6-60 トリガーを有効化する



図 6-61 トリガーの有効化の確認



Note キューのポーリング処理

このポーリングする Lambda 関数は、メッセージの取り出しをしていないのでメッセージが減りません。取り出しをするのは、こののちに作るメールを送信するプログラムのほうです。そのため、5分ごとに何度も同じメッセージをポーリングしようとしています。もし、すぐに「メールを送信するプログラム」を作らないのなら、メールを送信するプログラムを作ったあとに、トリガーを有効化するほうがよいでしょう。

● 6-4 SQS からメッセージを取り出してメールを送信する

6-4-5 SQS の内容を確認して SNS トピックに通知する仕組み

では、リスト 6-2 の動きを説明します。

■ キューに溜められたメッセージ数を確認する

まずは、SQS のキューに溜められたメッセージを確認します。そのためには、キューにアクセスするための Queue オブジェクトを取得します。

```
sqs = boto3.resource('sqs')
queue = sqs.get_queue_by_name(QueueName="mailsendqueue000")
```

キューに溜められているメッセージ数は、`ApproximateNumberOfMessages` 属性の値として取得できます。

```
n = queue.attributes['ApproximateNumberOfMessages']
```

■ SNS トピックに通知する

通知を送信するには、まず、SNS トピックを示す Topic オブジェクトを取得します。

```
sns = boto3.resource('sns')
topic = sns.Topic('arn:aws:sns:ap-northeast-1:XXXXXXXXXXXX:sendmail-notifications')
```

そして `publish` メソッドを送信すると通知を送信できます。必須なのはメッセージ本文を示す `Message` 引数だけです。ほかにも、属性を指定する `MessageAttributes` なども指定できます。この例では、本文として、「mailsendqueue000」というように、SQS のキュー名を指定することにしました。

```
# 10 単位で SNS トピックに通知する
for i in range(int((int(n) + 9) / 10)):
    topic.publish(
        Message='mailsendqueue000'
    )
```

6-4-6 SNSトピックの通知を受けたときにキュー処理するプログラム

これでSNSトピックに通知が送信されるところまでできました。最後に、この通知を受け取ったときに、キューから受信してメール送信する処理を作っていきます。

◎ 操作手順 ◎ SNSトピックの通知を受けたときにキューから受信してメールを送信するプログラムを作る

[1] sns-message-pythonの設計図から作成する

- SNSトピックを扱う設計図は、「sns-message-python」です。本書の執筆時点では、Python3用の設計図はありません。そこで、Python2.7用の設計図として作り、あとでPython3に変更することにします（図6-62）。



図6-62 sns-message-pythonの設計図から作成する

[2] SNSトピックを選択する

- トリガーとして、SNSトピックを選択します。ここでは、すでに作っておいたSNSトピック「sendmail-notifications」を選択します（図6-63）。
- トリガーの有効化には、チェックを付けないでおきます。

● 6-4 SQS からメッセージを取り出してメールを送信する

- ・ [次へ] ボタンをクリックします。



図 6-63 トリガーの設定

【3】関数名の設定

- ・ 関数名を設定します。どのような名前でもかまいませんが、ここでは、sendmail という関数名にします（図 6-64）。
- ・ ランタイムを「Python 3.6」に変更します。「sns-message-python」の設計図で作成したとき、デフォルトは Python 2.7 になるので、この点はとくに注意してください。



図 6-64 関数名を設定し、Python 3.6 に変更する

そして、関数のコードを記述します（図 6-65）。ここでは、リスト 6-3 のように記述します。

コードの意味は、「6-4-8 キューからメッセージを読み出してメールを送信する仕組み」で説明します。

なおリスト6-3の10行目にあるMAILFROMの設定は、送信元のメールアドレスです。適切な値に変更してください。

```
MAILFROM = 'yamada@example.com'
```

このメールアドレスは、SESで検証済みとしたメールアドレスでなければならぬので注意してください（第5章の「5-7-2 メールの送信制限を解除する」を参照）。



図6-65 関数のコードを入力する

リスト6-3 キューからメッセージを取り出してメールを送信する例

```
import json
import boto3

sns = boto3.resource('sns')
s3 = boto3.resource('s3')
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('mailaddress')

client = boto3.client('ses', region_name='us-east-1')
MAILFROM = 'yamada@example.com' ←メールアドレスの変更

def lambda_handler(event, context):
    #届いた通知を処理する
    for rec in event['Records']:
        snsmessage = rec['Sns']['Message']
        #SQSのキューを取得
```

● 6-4 SQS からメッセージを取り出してメールを送信する

```
queue = sqs.get_queue_by_name(QueueName=snsmessage)
# キューからメッセージを読み込む
messages = queue.receive_messages(MessageAttributeNames=['All'], MaxNumberOfMessages = 10)

# メッセージを処理してメールを送信する
for m in messages:
    # キューの内容
    email = m.body
    if m.message_attributes is not None:
        print("Sending...")
        username = m.message_attributes.get('username').get('StringValue')
        bucketname = m.message_attributes.get('bucketname').get('StringValue')
        filename = m.message_attributes.get('filename').get('StringValue')
        print(bucketname)
        print(filename)
        print(username)
        print(email)
        # S3バケットから本文を取得する
        obj = s3.Object(bucketname, filename)
        response = obj.get()
        maildata = response['Body'].read().decode('utf-8')
        datas = maildata.split("\n", 3)
        subject = datas[0]
        body = datas[2]

        print(subject)
        print(body)

    # 送信済みでないことを確認し、また、送信済みに設定する
    response = table.update_item(
        Key = {
            'email' : email
        },
        UpdateExpression = "set issend=:val",
        ExpressionAttributeValues = {
            ':val' : 1
        },
        ReturnValues = 'UPDATED_OLD'
    )

    if response['Attributes']['issend'] == 0:
        # メール送信
```

```
response = client.send_email(
    Source=MAILFROM,
    ReplyToAddresses=[MAILFROM],
    Destination={
        'ToAddresses' : [
            email
        ]
    },
    Message={
        'Subject': {
            'Data' : subject,
            'Charset' : 'UTF-8'
        },
        'Body' : {
            'Text': {
                'Data' : body,
                'Charset' : 'UTF-8'
            }
        }
    }
)
else:
    print("Resend Skip")

print("Send To" + email)
else:
    print("Message None")

# キューから取り除く
m.delete()
```

さらに、Lambda 関数ハンドラやロールを設定します。ハンドラはデフォルトのままでし、ロールは、`role-lambdaexec` を選択します。「タグ」や「詳細設定」はデフォルトのままでします。
[次へ] ボタンをクリックします（図 6-66）。

● 6-4 SQS からメッセージを取り出してメールを送信する



図 6-66 関数ハンドラやロールを設定する

最後に確認画面が表示されるので、[関数の作成] をクリックして関数を作成します（図 6-67）。



図 6-67 関数の確認

6-4-7 動作テスト

このように Lambda 関数を作ったら、動作テストをしてみましょう。この Lambda 関数は、SNS トピックが到着すると、「snsmessage」という名前が付けられたキューからメッセージを受信し、そこに記載されているメールアドレス宛に、メールを送信します。

そこでテストするため、[テスト] ボタンをクリックしてください（図 6-68）。はじめてテストするときにはテストイベントの設定画面が表示されますが、ここでは「SNS」を選択します（図

6-69)。

先ほどのリスト6-2の例では、次のようにメッセージ本文に「mailsendqueue000」というSQSのキュー名を渡しています。

```
topic.publish(
    Message='mailsendqueue000',
)
```

そこで、この画面でテストするときも、ボディ部に「mailsendqueue000」が設定されるように、テストメッセージの「Message」という部分を変更します。

```
"Message" : 'mailsendqueue000'
```

そして、図6-69の画面で【保存してテスト】ボタンをクリックすると、SNSトピックを受け取ったのと同じ状態をテストできます。



図6-68 テストを始める



図6-69 Messageを変更する

するとキューが処理されてメールが送信されるはずです。メールが送信されているかどうかを

● 6-4 SQS からメッセージを取り出してメールを送信する

確認してください。また、SQS のメッセージが空になることも確認しておきます（図 6-70）。



図 6-70 SQS のメッセージが空になることを確認する

以上で動作確認は終了です。トリガーを有効化しておいてください（図 6-71、図 6-72）。



図 6-71 トリガーを有効化する



図 6-72 有効化の確認

6-4-8 キューからメッセージを読み出してメールを送信する仕組み

ひと通りの動作テストをして、正しく動作することを確認したところで、リスト 6-3 の解説をしていきます。リスト 6-3 では、大きく、次の 3 つの処理をしています。

①通知メッセージを受け取る

通知メッセージは、イベント引数の `Records` 属性で取得できます。すなわち、次のようにループ処理することで、届いた通知を処理できます。

```
for rec in event['Records']:
    …通知を処理する…
```

通知の本文データは、次のように Sns と Message の要素の値として取得できます。

```
snsmessage = rec['Sns']['Message']
```

この値は、通知を送信するときに送信した値です。すなわち、リスト 6-2 における次の処理で送信された文字列に相当します。これは、SQS のキュー名を示す「mailsendqueue000」という文字列です。

【リスト 6-2 通知を送信する処理】

```
topic.publish(
    Message='mailsendqueue000'
)
```

②キューからメッセージを取り出す

①で対象のキュー名が得られるので、そのキューからメッセージを取り出します。まずは、Queue オブジェクトを作成します。

```
queue = sqs.get_queue_by_name(QueueName=snsmessage)
```

キューからメッセージを読み取るには、いくつかの方法がありますが、ここでは、最大 10 件までのメッセージをまとめて取り出せる receive_messages メソッドを使いました。

```
messages = queue.receive_messages(MessageAttributeNames=['All'], MaxNumberOfMessages = 10)
```

指定している 2 つの引数の意味は、次の通りです。

●MessageAttributeNames

取得する属性を指定します。「['ALL']」という文字列を渡すと、すべての属性という意味になります。

SQS は送受信するメッセージサイズが大きいほどコストがかかります。ですから、もしそれぞれの属性を処理する必要がないのなら、使う属性だけを列挙するほうが適切です。

●MaxNumberOfMessages

取得する最大メッセージ数を指定します。10 まで指定できます。

● 6-4 SQS からメッセージを取り出してメールを送信する

`receive_messages` メソッドを実行してメッセージを読み込むと、キューの設定における「デフォルトの可視性タイムアウト」の時間の間、他のプログラムからは隠され、メッセージが二重に処理されてしまうことを防ぎます（前掲の表 6-3 を参照）

`receive_messages` メソッドを実行しても、キューから取り出して、他から見えないように隠すだけで、キューからなくなりません。キューからなくすには、メッセージの `delete` メソッド（後述）を呼び出して、キューから削除する必要があります（もしくはキュー `delete_queue` メソッドや `delete_queue_batch` メソッドを使う方法もあります）。

③キューのメッセージや属性値を取り出す

②の戻り値のメッセージを、次のようにループ処理して、届いたメッセージをひとつずつ処理します。

```
for m in messages:
```

リスト 6-1 の処理では、キューに登録するとき、

- 本文には宛先メールアドレス
- `username` 属性に氏名
- `bucketname` に S3 バケット名
- `filename` に S3 のファイル名（オブジェクト名）

を登録しています。

そこでこれらの値を読み取ります。

●本文

`body` 属性で取得できます。

```
email = m.body
```

●属性

`message_attributes` 属性で取得できます。たとえば、次のようにして取得します。

```
username = m.message_attributes.get('username').get('StringValue')
bucketname = m.message_attributes.get('bucketname').get('StringValue')
```

```
filename = m.message_attributes.get('filename').get('StringValue')
```

取得したならば、S3 バケットに置かれたメール本文を示すファイルを取得します。

```
# S3 バケットから本文を取得する
obj = s3.Object(bucketname, filename)
response = obj.get()
maildata = response['Body'].read().decode('utf-8')
```

このファイルは「1行目にタイトル」「2行目が空」「3行目以降が本文」ということを想定しているので、次のように切り分けます。

```
datas = maildata.split("\n", 3)
subject = datas[0]
body = datas[2]
```

④送信済みでないかどうかを確認し、送信済みに設定する

冒頭で説明したように、今回の構成では、同じメールアドレスに対して、2回以上、送信の Lambda 関数が実行される恐れがあります。そこで送信済みかどうかを判定する `issend` 属性を用意して、送信済みであるとき（1のとき）は送信せず、そうでないときは送信して1に設定するという処理をします。そのためには、第5章でシーケンスの作成をするときに用いた `update_item` メソッドを利用できます。

```
response = table.update_item(
    Key = {
        'email' : email
    },
    UpdateExpression = "set issend=:val",
    ExpressionAttributeValues = {
        ':val' : 1
    },
    ReturnValues = 'UPDATED_OLD'
)
```

キーは「`email`」とし、`email` が合致する項目を探します。そして、`UpdateExpression` と `ExpressionAttributeValues` にあるように、`issend` 属性を1に設定します。

このとき第5章の例とは違って、`ReturnValues` には「`UPDATED_OLD`」を指定しています。そのため「更新後の値」ではなくて「更新前の値」を取得できます。

そこで、次のように `issend` 属性の現在の値を確認し、0であるときはメールを送信する処理を実現できます。

● 6-4 SQS からメッセージを取り出してメールを送信する

```
if response['Attributes']['isSend'] == 0:  
    ...isSend属性が0である。すなわちメールはまだ送信されていない…
```

⑤メールを送信する

ここまでできたら、SES を使ってメールを送信します。この処理は、第 5 章で説明したものと同じです。

```
response = client.send_email(  
    Source=MAILFROM,  
    ReplyToAddresses=[MAILFROM],  
    Destination={  
        'ToAddresses' : [  
            email  
        ]  
    },  
    Message={  
        'Subject': {  
            'Data' : subject,  
            'Charset' : 'UTF-8'  
        },  
        'Body' : {  
            'Text': {  
                'Data' : body,  
                'Charset' : 'UTF-8'  
            }  
        }  
    }  
)
```

⑥キューから排除する

以上でメッセージの処理が終わったので、キューから削除します。削除するには、`delete` メソッドを呼び出します。

```
m.delete()
```

以上で、基本的なメール送信処理まで実装できました。すなわち、あらかじめ宛先のメールアドレスを mailaddress テーブルに記述しておいたうえで、S3 バケットに「1 行目がタイトル」「2 行目が空行」「3 行目以降が本文」という構造のテストファイルを S3 バケットに置くことで、それが SQS を伝わり、最終的に SNS トピックとなってメールが送信されていくという一連の流れを作りました。

6-5 バウンスメールを処理する

インターネットのメール（SMTPプロトコル）では、宛先が不明な場合には、エラーメールが配信元に戻されます。これが、バウンスメールです。

SESはバウンスメールの率をカウントしており、バウンスメールであるにも関わらずメールを送信し続けると、配信できなくなる制限が課せられます。そのため、バウンスメールを確認して、次回以降は、送信しないようにするような処理が必要です。

SESでメールを送信している場合、バウンスメールが届いたとき、それをSNSトピックとして通知できます。SNSトピックからLambda関数を実行するように構成しておくことで、バウンスメールとして戻ってきたメールアドレスを取り出し、それをデータベースに「エラーメールである」という印を付けることができます。

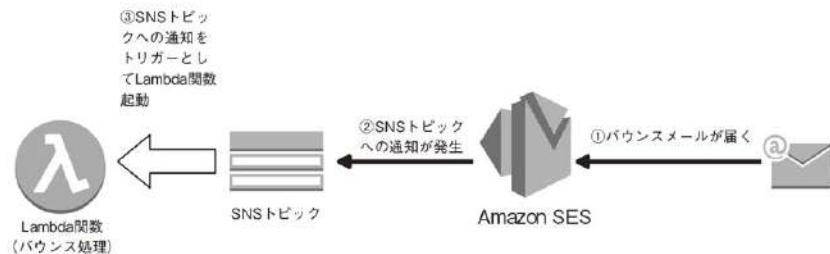


図 6-73 Lambda関数を使ったバウンスメールの処理

6-5-1 バウンスメールをSNSトピックに送信する

バウンスメールを処理するには、SNSトピックを作り、バウンスメールが届いたときに、そのSNSトピックに通知を送るように構成します。

■ SESと同じリージョンでSNSトピックを作成する

SNSトピックを作成する操作そのものは、今までの説明と同じなのですが、ひとつだけ注意点があります。それは、SESと同じリージョンで作成するという点です。

本書の構成では、バージニア北部のSESを使っています。ですからSNSトピックも、バージニア北部で作成してください。対象のリージョンは右上で変更できます（図6-74）。

● 6-5 バウンスメールを処理する

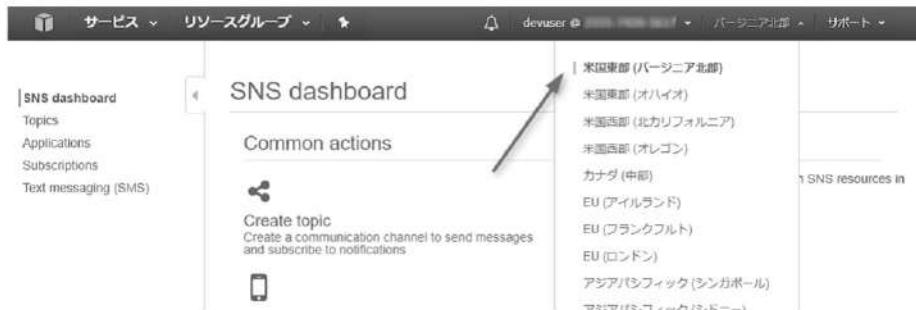


図 6-74 バージニア北部リージョンに切り替えて SNS を操作する

リージョンを切り替えたら、先ほどと同じ手順で SNS ダッシュボードの [Topics] メニューから [Create new topic] をクリックして、SNS トピックを作ります（図 6-75）。



図 6-75 [Create new topic] をクリックしてトピックを作成する

トピック名は何でもかまいませんが、ここでは、「bounce-notifications」という名前の SNS トピックを作ります（図 6-76）。

The screenshot shows the "Create new topic" dialog box. It has two input fields: "Topic name" containing "bounce-notifications" and "Display name" containing "Enter topic display name. Required for topics with SMS subscriptions.". A large arrow points from the "Create topic" button at the bottom right towards the "Create topic" button in the main SNS Topics page screenshot above.

図 6-76 SNS トピックを作成する

■ SES のバウンスメールの設定を変更する

次に、SES コンソールを開き、バウンスメールの設定を変更します。

◎ 操作手順 ◎ SESのバウンスメールの設定を変更する

[1] 差出人メールアドレスまたはドメインの設定を開く

- ・SESコンソールを開き、[Domains]（ドメイン単位で設定する場合）か[Email Addresses]（メールアドレス単位で設定する場合）かによって、いずれかを開きます。ここでは後者を選びます（図6-77）。
- ・設定したい差出人にチェックを付けて、[View Details]をクリックします。

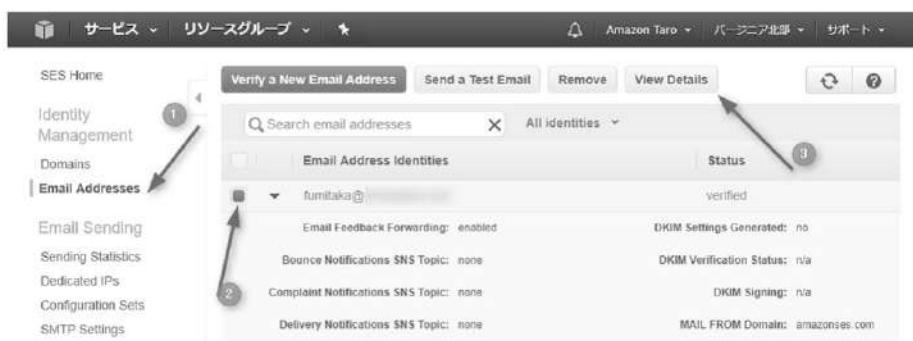


図6-77 詳細画面を開く

[2] 設定変更画面を開く

- ・[Notifications]のなかにある[Edit Configuration]をクリックして開きます（図6-78）。

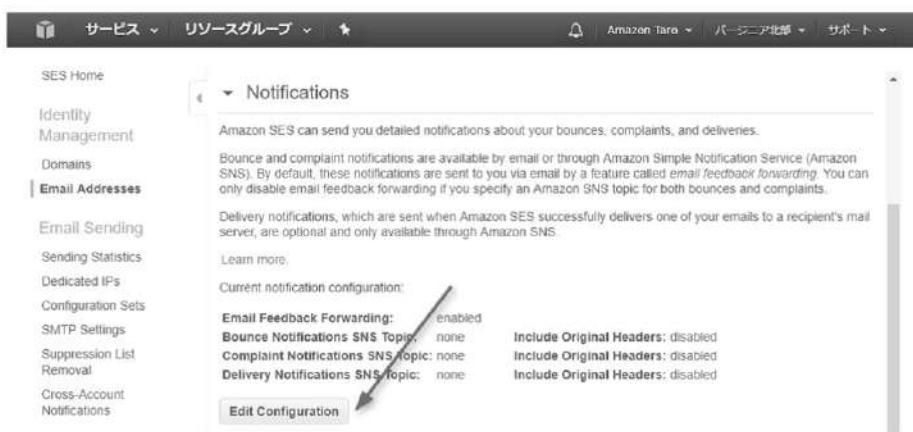


図6-78 Notificationsの設定画面を開く

[3] SNS トピックに通知するように構成する

- ・ [Bounces] の部分で先ほど作成した SNS トピックを選択し、[Save Config] ボタンをクリックすると、バウンスメールが届いたときに、通知が送信されるようになります（図 6-79）。

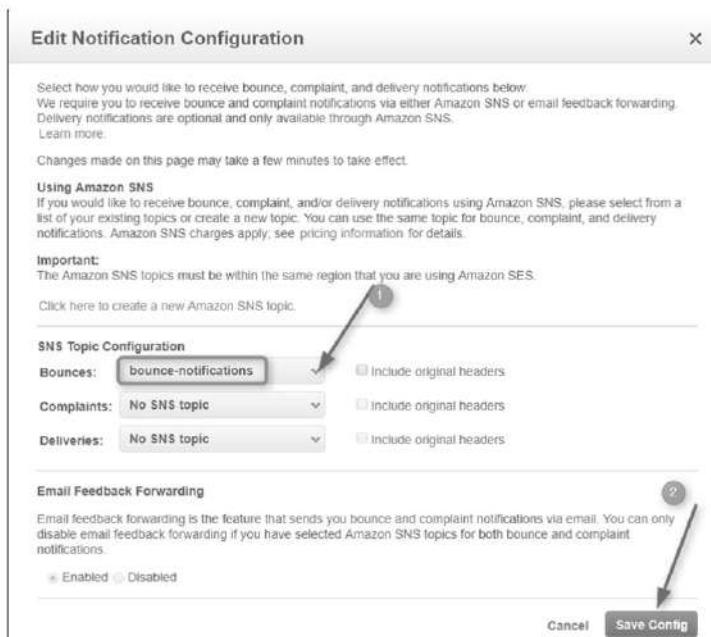


図 6-79 バウンスメールを SNS トピックに通知するように構成する

6-5-2 SNS トピックでバウンスメールを処理する

次に、この SNS トピックでバウンスを受け取って処理するように構成します。

そのためには、先ほど「6-4-6 SNS トピックの通知を受けたときにキュー処理するプログラム」と同じようにして、SNS トピックの通知を受けたときに Lambda 関数を起動できるように構成し、その Lambda 関数で、バウンスメールの対象となったメールアドレスを抽出し、DynamoDB の mailaddress テーブルの haserror 属性を 1 にする処理を実行します。

■ Lambda 関数を作成する

この Lambda 関数は、SNS トピックと同様に、バージニア北部リージョンに作成します。

◎ 操作手順 ◎ バウンスメールを処理する Lambda 関数を作る

[1] Lambda関数を作り始める

先ほどの例と同様に、「sns-message-python」の設計図から作成します（図6-80）。



図 6-80 バージニアリージョンで Lambda 関数を作る

[2] SNSトピックを選択する

- ・トリガーとなる SNS トピックを選択します。ここでは、先にバウンスメールが発生したときに通知するように構成しておいた bounce-notification という SNS トピックを選びます。ここでは話を簡単にするため、【トリガーの有効化】にはチェックを付けて、有効になるように構成しておくものとします。【次へ】ボタンをクリックします。



図 6-81 ソースとしてバウンスメールを受け取ったときに通知を受け取る SNS トピックを選択する

● 6-5 バウンスメールを処理する

【3】関数名の設定

- ・関数名を設定します。どのような名前でもかまいませんが、ここでは「bounce」という関数名にします。このとき、先に説明したのと同様、ランタイムを「Python 3.6」に変更するのを忘れないでください（図 6-82）。



図 6-82 関数名を設定し、Python 3.6 に変更する

- ・そして関数のコードを記述します（図 6-83）。ここでは、リスト 6-4 のように記述します。コードの意味は、「6-5-4 バウンスメールを処理する仕組み」で説明します。
以降の操作は、「6-4-6 SNS トピックの通知を受けたときにキュー処理するプログラム」とほぼ同じなので、説明は割愛します。



図 6-83 コードを入力する

リスト6-4 バウンスメールを処理するためのコード

```
import json
import boto3

# リージョンが違うため、明示的に指定している点に注意
dynamodb = boto3.resource('dynamodb', region_name='ap-northeast-1')
table = dynamodb.Table('mailaddress')

def lambda_handler(event, context):
    for rec in event['Records']:
        # バウンスしたメールアドレスの取得
        message = rec['Sns']['Message']
        data = json.loads(message)
        bounces = data['bounce']['bouncedRecipients']
        for b in bounces:
            email = b['emailAddress']
            # haserrorを1に設定する
            response = table.update_item(
                Key={'email' : email},
                UpdateExpression="set haserror=:val",
                ExpressionAttributeValues= {
                    ':val' : 1
                }
            )
```

6-5-3 動作テスト

プログラムを入力したら、動作テストを実行します。このLambda関数を設定した後、S3バケットにメール本文を置き直して、メールを送信してみます。

本章の手順では、配信先として配信エラーが発生するメールボックスシミュレーターのアドレスが含まれているため、いくつかの配信に失敗します。このときバウンスマールが戻ってきます。

そこでDynamoDBのmailaddressテーブルを確認してみると、バウンスマールが戻ってきたメールアドレスは、haserror属性が1に設定されたのがわかります。

● 6-5 バウンスメールを処理する

The screenshot shows the AWS Lambda function configuration interface. On the left, there's a sidebar with a tree view of resources, including 'mailaddress' which is selected. The main area shows the 'mailaddress' table details. A search bar at the top says 'mailaddress'. Below it are tabs for '概要', '項目', 'メトリックス', 'アラーム', '容量', 'インデックス', 'トリガー', and 'さらに'. The '項目' tab is selected. A search bar within this tab says 'スキャン: [テーブル] mailaddress: email'. Below that is a section titled 'スキャン' with a dropdown set to 'テーブル: mailaddress: email' and a radio button for 'フィルターの追加'. A '開始' button is present. The main table lists six items:

email	haserror	issend	username
bounce@simulator.amazonaws.com	1	1	鈴木次郎
oto@simulator.amazonaws.com	1	1	田中三郎
success@simulator.amazonaws.com	0	1	山田太郎
complain@simulator.amazonaws.com	0	1	加藤四郎
yamada@example.co.jp	0	1	大澤文彦
suppressionlist@simulator.amazonaws.com	1	1	佐藤五郎

図 6-84 バウンスしたメールの項目の haserror が「1」になることがわかる

6-5-4 バウンスメールを処理する仕組み

リスト 6-4 のプログラムの動作を説明します。

①リージョンが異なるリソースにアクセスする

このプログラムでは、mailaddress テーブルを操作しますが、このテーブルは、北バージニアリージョンではなく東京リージョンにあります。そのようなときには、region_name パラメータで、該当のリージョンを明示してリソースを操作するオブジェクトを取得する必要があります。

```
dynamodb = boto3.resource('dynamodb', region_name='ap-northeast-1')
table = dynamodb.Table('mailaddress')
```

②バウンスマールのデータを取得する

SNS トピックに通知されたメッセージを取得するには、すでに説明したように、次のように Records 属性を参照してループ処理します。

```
for rec in event['Records']:
    ...メッセージの処理…
```

このときのデータ構造は、実際に、JSON 形式で print して CloudWatch Logs に書き出された

データを見て確認するとわかりますが、次のようなデータです。

【Records 属性のデータの例】

```
{
  "EventSource": "aws:sns",
  "EventVersion": "1.0",
  "EventSubscriptionArn": "...略...",
  "Sns": {
    "Type": "Notification",
    "MessageId": "メッセージID",
    "TopicArn": "トピックSNSトピックのARN",
    "Subject": null,
    "Message": "{\"notificationType\":\"Bounce\", \"bounce\":{\"bounceType\":\"Permanent\", \"bounceSubType\":\"Suppressed\", \"bouncedRecipients\":[{\"emailAddress\":\"suppressionlist@simulator.amazonaws.com\", \"action\":\"failed\", \"status\":\"5.1.1\", \"diagnosticCode\":\"Amazon SES has suppressed sending to this address because it has a recent history of bouncing as an invalid address. For more information about how to remove an address from the suppression list, see the Amazon SES Developer Guide: http://docs.aws.amazon.com/ses/latest/DeveloperGuide/remove-from-suppressionlist.html\"}], \"timestamp\":\"2017-08-15T17:19:12.856Z\", \"feedbackId\":\"0100015de6e809a2-db7123f9-81dd-11e7-9f88-37a68432784a-000000\", \"reportingMTA\":\"dns; amazonses.com\"}, \"mail\":{\"timestamp\":\"2017-08-15T17:19:12.856Z\", \"source\":\"yamada@example.com\", \"sourceArn\":\"arn:aws:ses:us-east-1:255574205617:identity:yamada@example.com\", \"sourceIp\":\"13.114.47.53\", \"sendingAccountId\":\"アカウントID\", \"messageId\":\"0100015de6e8088c-66a214c8-c875-4e11-a9fd-c13d0ba139ce-000000\", \"destination\":[\"suppressionlist@simulator.amazonaws.com\"]}}",
    "Timestamp": "2017-08-15T17:19:12.905Z",
    "SignatureVersion": "1",
    "Signature": "...略...",
    "SigningCertUrl": "...略...",
    "UnsubscribeUrl": "...略...",
    "MessageAttributes": {}
  }
}
```

この文字列からわかるように、バウンスメールの各種データは、JSON化されて Message 属性に格納されています。そこで、次のようにして元に戻します。

```
message = rec['Sns']['Message']
data = json.loads(message)
```

バウンスに関する情報は、この bounce 属性にあり、bouncedRecipients 属性に、その情報がディクショナリとして格納されています。

```
bounces = data['bounce']['bouncedRecipients']
for b in bounces:
    ...#バウンスされた情報の処理…
```

メールアドレスは、次のように `emailAddress` 属性で取得できます。

```
email = b['emailAddress']
```

③DynamoDB のテーブルを更新する

②で見つかったメールアドレスに相当する項目の `haserror` 属性を 1 に設定します。そのためには、次のように `update_item` メソッドを実行します。

```
response = table.update_item(
    Key={'email' : email},
    UpdateExpression="set haserror=:val",
    ExpressionAttributeValues= {
        ':val' : 1
    }
)
```

6-6 まとめ

本章では、SQS と SNS トピックを組み合わせることで、処理を溜めたり、通知を使って別の Lambda 関数を実行したりする仕組みを説明しました。

①SQS で処理をキューイングする

SQS を利用すると、処理をいったんキューに溜めて、別のプログラムで処理できるようになります。

SQS からデータを取り出すには、スケジューリングされた Lambda 関数を使って定期的にポーリングします。

②他の Lambda 関数を実行するには SNS トピックを使う

SNS トピックは通知機能です。この通知機能を使って、Lambda 関数を実行できます。

Lambda 関数から別の Lambda 関数を実行したいときだけでなく、SES でバウンスマールを受け取ったときの処理をしたいなど、SNS トピックに通知できるあらゆる AWS のサービスと Lambda 関数を連携したいときには、SNS トピックを利用することになるでしょう。

第6章 SQSとSNSトピックを使った連携

SQSやSNSトピックを使うと、Lambdaにおけるプログラミングは、ひとつのLambda関数ですべてを処理するのではなく、小さなLambda関数をたくさん組み合わせてシステム全体を構築するという作り方になります。

Lambda関数は、時間単位での課金ですから長い時間稼働させるとコストが嵩みます。またそもそも最大5分間しか処理できません。

こうしたデメリットは、SQSを使って処理をキューイングして逐次処理する設計にすることで、解決できるはずです。

Appendix A

Lambda 開発者アカウントの作成

「2-3 Lambda の利用に必要なアクセス権」で説明したように、Lambda で開発するには、Lambda 関数を作成したり、実行したりできる権限が必要です。ここでは、そうした権限を付与したユーザーとして、AWSLambdaFullAccess ポリシーを適用した IAM ユーザーを作成します。

■ IAM ユーザーの作成

◎ 操作手順 ◎ AWSLambdaFullAccess ポリシーを適用した IAM ユーザーを作成する

[1] IAM コンソールを開く

- ・ IAM ユーザーは、AWS マネジメントコンソールの「IAM コンソール」から操作します。まずは、IAM コンソールを開いてください（図 A-1）。



図 A-1 IAM コンソールを開く

[2] ユーザーを追加する

- ・ [ユーザー] を選択して、ユーザー一覧を表示します。 [ユーザーを追加] ボタンをクリックして、ユーザーを追加します（図 A-2）。



図 A-2 ユーザーを追加する

[3] ユーザー名を決め、アクセスの種類を選択する

- ・ ユーザー名を決めます。ここでは仮に、「devuser」とします（図 A-3）。
- ・ [アクセスの種類] では、このユーザーが、どのようにアクセスするのかを決めます。 [プログラムによるアクセス] と [AWS マネジメントコンソールへのアクセス] の 2 つがあります。前者は AWS CLI などのコマンドを使ってログインするもの、後者は Web ブラウザで AWS マネジメントコンソールを使ってログインするためのものです。本書では、どちらの操作もするので、両方にチェックを付けておいてください。
- ・ [AWS マネジメントコンソールへのアクセス] にチェックを付けたときは、 [コンソールのパスワード] と [パスワードのリセットが必要] の選択肢が現れます。前者は、パスワードを自動生成するか、それともここで入力するかを決めるものです。どちらでもかまいませんが、ここでは [自動作成パスワード] を選択します。後者は、初回ログイン時にパスワードの変更が必要かを決めるものです。ここではチェックを付けておきます。
- ・ 上記すべてを入力したら、 [次のステップ：アクセス権限] をクリックします。



図 A-3 ユーザー名を決め、アクセスの種類を選択する

[4] アクセス権の設定

- ・アクセス権を設定します。ここでは、AWSLambdaFullAccess ポリシーを設定したいので、【既存のポリシーを直接アタッチ】を選択して、AWSLambdaFullAccess ポリシーにチェックを付けてください（検索の部分に「AWSLambda」など、頭から何文字か入力するとフィルタでき、見つけやすくなります）。
- ・チェックを付けたら、【次のステップ: 確認】をクリックしてください（図 A-4）。

Appendix A Lambda 開発者アカウントの作成

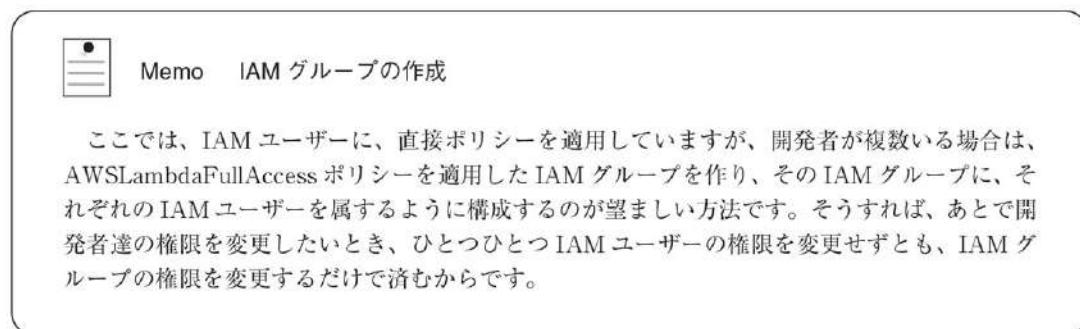


図 A-4 既存のポリシーを直接アタッチする

[5] 確認する

- 確認画面が表示されます。[ユーザーの作成] ボタンをクリックしてください。

確認

選択内容を確認します。ユーザーを作成した後で、自動生成パスワードとアクセスキーを確認してダウンロードできます。

ユーザー詳細

ユーザー名	devuser
AWS アクセスの種類	プログラムによるアクセスと AWS マネジメントコンソールへのアクセス
コンソールのパスワードの種類	自動生成
パスワードのリセットが必要	はい

アクセス権限の概要

次のポリシー例は、上記のユーザーにアタッチされます。

タイプ	名前
管理ポリシー	AWSLambdaFullAccess
管理ポリシー	IAMUserChangePassword

キャンセル 戻る **ユーザーの作成**

図 A-5 ユーザーを作成する

[6] アクセスキー ID、シークレットアクセスキーや、パスワードを確認する

- 作成したユーザーの「アクセスキー ID」「シークレットアクセスキーや」「パスワード」が表示されます（図 A-6）。隠されているところは [表示] をクリックすると、表示されます。[.csv のダウンロード] をクリックすると、CSV 形式でダウンロードすることもできます。
- 「シークレットアクセスキーや」と「パスワード」は、この画面から離れると、もう確認できないので注意してください（忘れた場合は、再発行するしかありません）。

成功

以下に示すユーザーを正常に作成しました。ユーザーのセキュリティ認証情報を確認してダウンロードできます。AWS マネジメントコンソールへのサインイン手順を E メールでユーザーに送信することもできます。今後が、これらの認証情報をダウンロードできる最後の機会です。ただし、新しい認証情報はいつでも作成できます。

AWS マネジメントコンソールへのアクセス権を持つユーザーは [https://\[REDACTED\].signin.aws.amazon.com/console](https://[REDACTED].signin.aws.amazon.com/console) でサインインできます

.csv のダウンロード

IAMユーザーのサインインページ

ユーザー	アクセスキー ID	シークレットアクセスキーアクション	パスワードアクション
devuser	AKIAJFUBAO3KHUVTKLQQ	表示	表示

閉じる

図 A-6 アクセスキー ID、シークレットアクセスキーや、パスワードを確認する

■作成した IAM ユーザーでログインする

AWS の root ユーザー（最初に AWS アカウントを作成したメールアドレスでのログイン）以外は、下記の「IAM ユーザーのサインインリンク」と呼ばれる場所からログインします。

【IAM ユーザーのサインインリンク】

<https://アカウント ID.signin.aws.amazon.com/console>

アカウント ID は、AWS の利用者ごとに割り当てられる番号です。この URL は、IAM コンソールのダッシュボードで確認できます（図 A-7）。



図 A-7 サインインリンクを確認する

サインインリンクにアクセスし、「User Name」と「Password」の入力欄に、作成した「ユーザー名」と「設定されたパスワード」を入力すると、サインインできます（図 A-8）。



図 A-8 IAM ユーザーでサインインする

Λ Column パスワードやシークレットアクセスキーを再設定するには

あとからパスワードやシークレットアクセスキーを変更したいときは、ユーザーの設定画面を表示し、【認証情報】から操作します（図 A-9）。

アクセスキー ID とシークレットアクセスキーはペアなので、シークレットキーだけを作り直すことはできません。シークレットアクセスキーを忘れてしまったときは、いまのアクセスキーを無効にし、新たにアクセスキー ID / シークレットアクセスキーを作り直してください。



図 A-9 パスワードやシークレットアクセスキーを再設定する

■ API Gateway、SES や SQS、SNS を操作する権限を加える

AWSLambdaFullAccess には、Lambda および DynamoDB、S3 など、Lambda 関連の各リソースへのフルアクセス権しかありません。

本書では、開発者が API Gateway や SES、SQS、SNS を操作します。そこでこれらに必要な権限を与えるため、下記の管理ポリシーを適用します。

なお、下記のポリシーはフルアクセス権を与えるものです。実運用では、必要な権限だけを与えるようにしてください。

- ・AmazonAPIGatewayAdministrator : API Gateway の作成やデプロイに必要なポリシーです（第4章）
- ・AmazonSESFullAccess : SES の操作に必要なポリシーです（第5章、第6章）
- ・AmazonSQSFullAccess : SQS の操作に必要なポリシーです（第6章）
- ・AmazonSNSFullAccess : SNS の操作に必要なポリシーです（第6章）

◎操作手順◎ 開発者に必要な管理ポリシーを適用する

[1] ユーザー設定を開く

- ・IAM コンソールを開いて、[ユーザー] をクリックしてユーザー一覧を表示します。開発者に相当するユーザー（本書の場合は「devuser」）をクリックして、ユーザーの設定画面を開いてください。



図 A-10 ユーザーの設定画面を開く

[2] アクセス権限を追加する

- ・[アクセス権限] タブをクリックして開き、[アクセス権限の追加] ボタンをクリックします。



図 A-11 アクセス権限を追加する

[3] 必要な管理ポリシーをアタッチする

- ・ [既存のポリシーを直接アタッチ] をクリックします（図 A-12）。
- ・ 「AmazonAPIGatewayAdministrator」「AmazonSESFullAccess」「AmazonSQSFullAccess」「AmazonSNSSFullAccess」の、それぞれの管理ポリシーにチェックを付けます。



図 A-12 既存のポリシーをアタッチする

まずは「AmazonAPIGatewayAdministrator」を検索してチェックを付けます。残りの「AmazonSESFullAccess」「AmazonSQSFullAccess」「AmazonSNSSFullAccess」も、同様に検索してチェックを付けます。

[4] 適用する

- 確認画面が表示されるので、[アクセス権限の追加] ボタンをクリックして、これらのポリシーを適用します（図 A-13）。適用すると、図 A-14 のように設定されます。



図 A-13 ポリシーを適用する

The screenshot shows the 'Attached Policies' section of the AWS IAM Policies page for user 'devuser'. It lists six policies that have been attached:

- AmazonSQSFullAccess
- AWSLambdaFullAccess
- AmazonSESFullAccess
- IAMUserChangePassword
- AmazonAPIGatewayAdministrator
- AmazonSNSSFullAccess

図 A-14 各種管理ポリシーが追加されたところ

Appendix B

Lambda 実行ロールの作成

「2-3 Lambda の利用に必要なアクセス権」で説明したように、Lambda 関数は、ある IAM ロールで実行されます。そのため、実行用の IAM ロールを作成しておく必要があります。

Lambda 関数の実行には、最低限 AWSLambdaBasicExecutionRole ポリシーを付与したロールが必要なため、ここでは「role-lambdaexec」という名前で作成する方法について説明します。また、AWS のリソースを操作する場合（たとえば、S3 や DynamoDB にアクセスする場合など）には、追加の権限が必要です。

■ IAM ロールの作成

◎ 操作手順 ◎ **AWSLambdaBasicExecutionRole ポリシーを適用した IAM ロールを作成する**

[1] ロールを追加する

- IAM コンソールを開き（IAM コンソールについては Appendix A を参照）、左側メニューの【ロール】を選択して、ロール一覧を表示します。【ロールの作成】ボタンをクリックして、ロールを追加します（図 B-1）。



図 B-1 ロールの追加

[2] ロールタイプを選択する

- ・ロールを使用するサービスの種類が表示されるので、[AWS サービス] をクリックし、[Lambda] を選択します（図 B-2）。次の画面で [次のステップ：アクセス権限] をクリックします（図 B-3）。



図 B-2 AWS Lambda を選択する

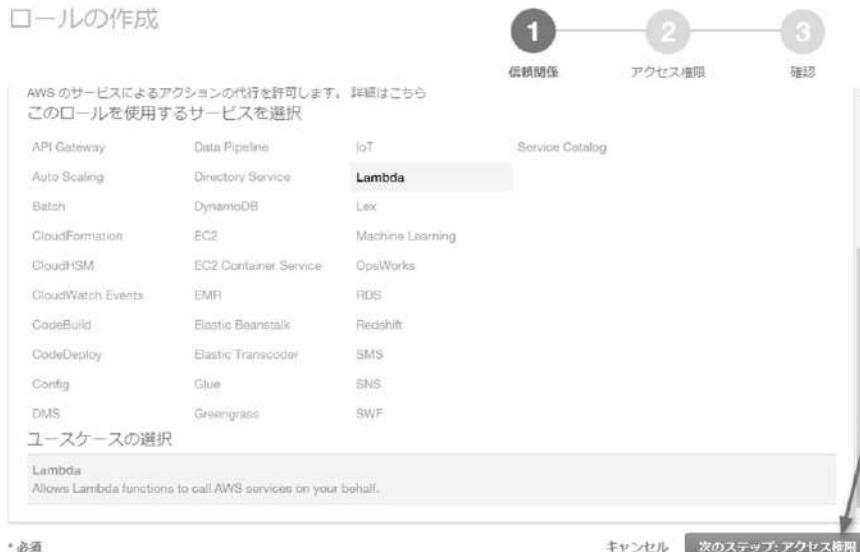


図 B-3 [次のステップ：アクセス権限] をクリック

[3] ポリシータイプを選択する

- 適用するポリシーを選択します。ここでは、AWSLambdaBasicExecutionRole ポリシーにチェックを付け、次に進んでください（図 B-4）。

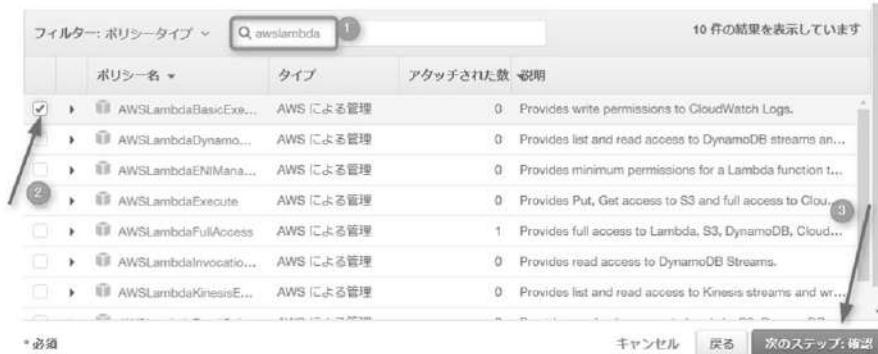


図 B-4 AWSLambdaBasicExecutionRole ポリシーを選択する

[4] ロール名を設定する

- ロール名を設定します。ここでは、「role-lambdaexec」というロール名とします。Role descriptionには、好きな説明文を記述できますが、ここでは空欄のままとします。

[ロールの作成] ボタンをクリックすると、ロールが作成されます（図 B-5）。

The screenshot shows a confirmation dialog for creating a role. It includes fields for 'Role Name' (set to 'role-lambdaexec'), 'Role Description' (empty), and a dropdown for 'Assumed by' (set to 'ID プロバイダ lambda.amazonaws.com'). At the bottom, there is a 'Role Name' summary and three buttons: 'キャンセル' (Cancel), '戻る' (Back), and a large 'Roleを作成' (Create Role) button which is highlighted with a thick arrow.

図 B-5 ロール名を設定する

■ S3 や DynamoDB、SES、SQS、SNS を操作する権限を加える

S3 や DynamoDB などの AWS リソースにアクセスするには、追加のアクセス権が必要です。本書のサンプルを実行するために必要となるアクセス権を与えるため、下記の管理ポリシーを適用します。

なお、下記のポリシーはフルアクセス権を与えるものです。実運用では、必要な権限だけを与えるようにしてください。

- ・ AmazonS3FullAccess : S3 にアクセスするのに必要なポリシーです（第 4 章）
- ・ AmazonDynamoDBFullAccess : DynamoDB にアクセスするのに必要なポリシーです（第 5 章）
- ・ AmazonSESFullAccess : SES の操作に必要なポリシーです（第 5 章、第 6 章）
- ・ AmazonSQSFullAccess : SQS の操作に必要なポリシーです（第 6 章）
- ・ AmazonSNSFullAccess : SNS の操作に必要なポリシーです（第 6 章）

◎操作手順◎ Lambda の実行ロールに必要な管理ポリシーを適用する

【1】 ロール設定を開く

- ・ IAM コンソールを開いて、「ロール」をクリックしてロール一覧を表示します。Lambda 実行ロールに相当するロール（本書の場合は「role-lambdaexec」をクリックして、ロールの設定画面を開いてください。



図 B-6 ロールの設定画面を開く

[2] アクセス権限を設定する

- ・ [アクセス権限] タブをクリックして開き、[ポリシーのアタッチ] ボタンをクリックします(図 B-7)。



図 B-7 アクセス権限を追加する

[3] 必要な管理ポリシーをアタッチする

- ・ポリシーの一覧が表示されるので、「AmazonS3FullAccess」「AmazonDynamoDBFullAccess」「AmazonSESFullAccess」「AmazonSQSFullAccess」「AmazonSNSFullAccess」の、それぞれの管理ポリシーにチェックを付け、[ポリシーのアタッチ] ボタンをクリックします(図 B-8)。



図 B-8 ポリシーをアタッチする

Appendix B Lambda 実行ロールの作成

まずは「AmazonS3FullAccess」を検索してチェックを付けます。そして残りも、同様に検索してチェックを付けます。

【4】 設定完了

- ・アタッチを完了すると、図 B-9 のように設定されます。

The screenshot shows the 'Attached Policies' section of the AWS Lambda role configuration. It lists six policies attached to the role:

Policy Name	Policy Type
AmazonSQSFullAccess	AWS Management Policy
AmazonS3FullAccess	AWS Management Policy
AmazonDynamoDBFullAccess	AWS Management Policy
AmazonSESFullAccess	AWS Management Policy
AWSLambdaBasicExecutionRole	AWS Management Policy
AmazonSNSSFullAccess	AWS Management Policy

At the bottom right of the table, there is a link labeled 'Create inline policy'.

図 B-9 各種管理ポリシーが追加されたところ

Appendix C

Lambda 開発マシンの準備

Lambda 関数でライブラリを用いる場合、ライブラリをビルドしてアーカイブに含める必要があるため、Lambda コンテナと同じ環境のマシンで開発するのが確実です。ここでは、インターネットから SSH ログインできる EC2 インスタンスを作り、Python での開発ができるようにする方法を説明します。

■ Lambda 開発マシンのポイント

Lambda 開発マシンを作るときのポイントは、次の 3 つです。

①インターネットから SSH を利用して EC2 を操作できるようにする

開発には EC2 インスタンスを利用するため、SSH を使って EC2 にログインします。そのためには、EC2 にグローバル IP アドレスを割り当てる必要があります。これにはいくつか方法がありますが、それぞれのリージョンに用意された「デフォルトの VPC／デフォルトのサブネット」に配置するのが簡単です。

②Amazon Linux AMI を用いる

EC2 インスタンスでは、さまざまな AMI (OS のイメージ) から起動できますが、第 2 章で説明しているように、Lambda の実行環境は、Amazon Linux AMI です。ですから、Lambda 開発マシンも、Amazon Linux AMI で構成するのが確実です。

③IAM ロールの権限を Lambda の実行ロールの権限に合わせる

第 3 章でも説明していますが、Lambda 開発マシンの IAM ロールを Lambda の実行ロールと同じ権限にしておくと、S3 などのリソースにアクセスする場合のアクセス権なども同じ構成でデバッグできます。

■ EC2 インスタンスの作成

まずは、インターネットから SSH ログインできる EC2 インスタンスを作成します。

◎ 操作手順 ◎ デフォルトのサブネットに EC2 インスタンスを配置する

[1] EC2 コンソールを開く

EC2 インスタンスは、AWS マネジメントコンソールの「EC2」から操作します。まずは、EC2 コンソールを開いてください（図 C-1）。

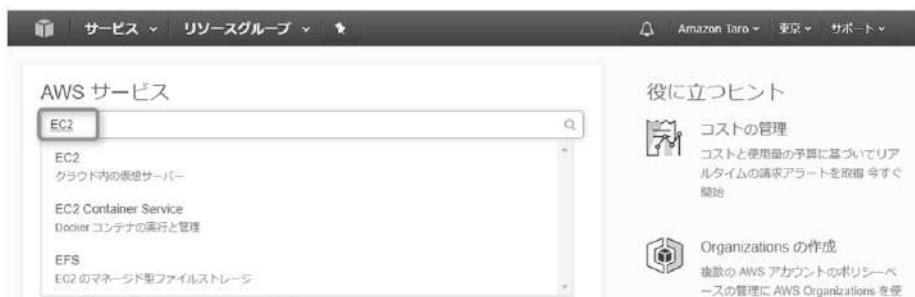


図 C-1 EC2 コンソールを開く

[2] EC2 インスタンスの作成を始める

【インスタンス】メニューをクリックして、インスタンス一覧を開きます。【インスタンスの作成】ボタンをクリックして、EC2 インスタンスの作成を始めてください（図 C-2）。



図 C-2 インスタンスの作成を始める

[3] AMI を選ぶ

AMI を選択します。「Amazon Linux AMI」を選択してください（図 C-3）。



図 C-3 Amazon Linux AMI を選択する

[4] インスタンスタイプを選択する

インスタンスタイプを選択します（図 C-4）。ここでは「t2.micro」を選択します。

設定したら、【次の手順：インスタンスの詳細】を選択します（【確認と作成】を選択すると、その後の設定すべてがスキップされるので注意してください）。



図 C-4 インスタンスタイプを選択する

[5] 配置するサブネットと IAM ロールを設定する

インスタンスの詳細な情報を設定します。ここでは、「配置するサブネット」と「IAM ロール」を主に設定します。

①ネットワークとサブネット

ネットワークには「(デフォルト)」と書かれたものを選択します(図 C-5)。これが「デフォルトの VPC」と呼ばれるもので、インターネットに接続し、グローバル IP アドレスを割り当てる構成されたネットワークです(「vpc-XXXX」の XXXX の部分の名称は環境によって異なります)。サブネットは、どこを選んでもかまいませんが、ここでは「優先順位なし」を選択します。すると、どれかが適当に選ばれます。

デフォルトの VPC に配置することで、グローバル IP アドレスが割り当てられ、インターネットと接続できる EC2 インスタンスとなります。



図 C-5 ネットワークとサブネットを設定する

②IAM ロール

Lambda 開発として使う EC2 インスタンスの実行ロールは、Lambda の実行ロールと同じ構成にしておくと、開発やデバッグがしやすくなります。本書では図 B-9 に示したように、Lambda 関数の実行には、「AWSLambdaBasicExecutionRole」「AmazonS3FullAccess」「AmazonDy

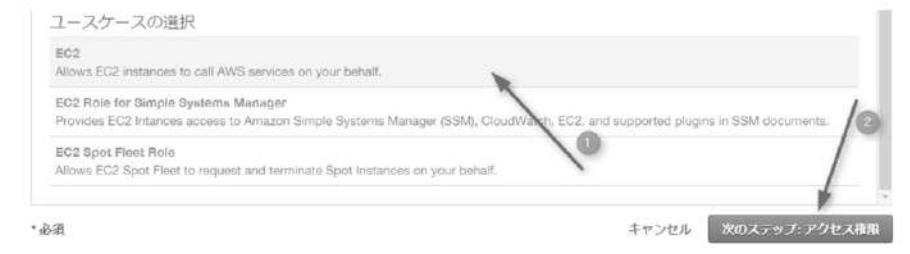
「AmazonDynamoDBFullAccess」「AmazonSESFullAccess」「AmazonSQSFullAccess」「AmazonSNSTFullAccess」の6つの管理ポリシーを適用したIAMロールを利用しています。そこで、これと同じ管理ポリシーを適用したIAMロールを作り、それをEC2インスタンスの実行ロールとして使うように構成しましょう。

図C-5で、「新しいIAMロールの作成」をクリックします。すると、図C-6の画面が表示されるので、「ロールの作成」ボタンをクリックします。



図C-6 ロールを作成する

ロールタイプの選択で「EC2」を選び、EC2に関するロールを作成したいので、「Amazon EC2」を選択します（図C-7）。



図C-7 Amazon EC2 を選択する

ポリシーを適用します。図B-9に示した「AWSLambdaBasicExecutionRole」「AmazonS3FullAccess」「AmazonDynamoDBFullAccess」「AmazonSESFullAccess」「AmazonSQSFullAccess」「AmazonSNSTFullAccess」の管理ポリシーを適用します。まずは、「AWSLambdaBasicExecutionRole」を検索してチェックを付けます（図C-8）。そして残りの項目も同様に検索してチェックを付け、最後に「次のステップ」ボタンをクリックします。

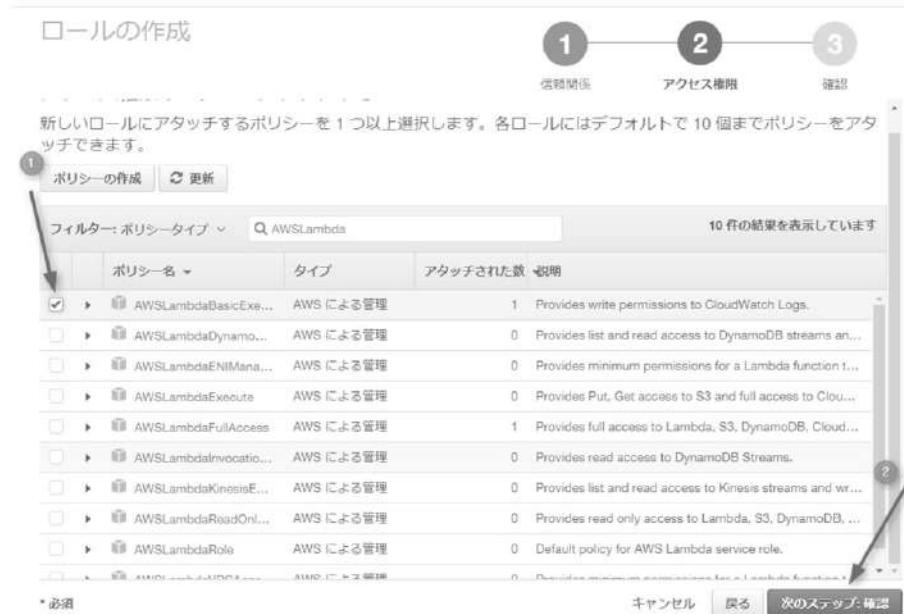


図 C-8 ポリシーを適用する



注意

本書において、Lambda 開発用の EC2 インスタンスを用いているのは第4章です。第4章では、S3 にアクセスする例を示しています。もしここで、設定する IAM ロールに AmazonS3FullAccess の管理ポリシーの適用をし忘れる、たとえばリスト 4-3 のような、EC2 インスタンスから S3 にアクセスするサンプルの一部が動きません。

最後にロール名を設定します。ここでは「role-ec2-lambdaexec」とします（図 C-9）。



図 C-9 ロール名を設定する

ロールを作成したら、図 C-5 の画面に戻り、いま作成した「role-ec2-lambdaexec」を選択してください（図 C-10）。選択したら、「次の手順：ストレージの追加」をクリックしてください。



図 C-10 IAM ロールを選択する

[6] ストレージの追加

ストレージを設定します。デフォルトでは「8GB」のストレージが構成されるので、そのまま [次の手順：タグの追加] を選択してください（図 C-11）。



図 C-11 ストレージの追加

[7] タグの追加

タグを追加します。あとで区別するため、インスタンスの名前だけは、追加しておくことを推奨します。

[タグの追加] ボタンをクリックして項目を追加し、Name というキーに、インスタンス名を設定しましょう（図 C-12）。インスタンス名は何でもかまいませんが、ここでは「LambdaDevelopServer」としました。タグを追加したら、[次の手順：セキュリティグループの設定] をクリックしてください。



図 C-12 タグの追加

[8] セキュリティグループの設定

セキュリティグループとは、パケットフィルタ型のファイアウォールのことです。デフォルトでは、ポート 22 番しか接続できない構成になっています。Lambda 開発マシンとして使うには、これで十分なので【確認と作成】ボタンをクリックしてください（図 C-13）。



図 C-13 ファイアグループを構成する

最後に最終確認画面が表示され、【作成】ボタンをクリックすると、EC2 インスタンスが作成されます（図 C-14）。



図 C-14 構築前の最終確認

[9] キーペアの選択または作成

作成した EC2 インスタンスへは、SSH でログインしますが、その際、「キーペア（公開鍵・秘密鍵）」を使ってアクセスします。

キーペアを新規に作成するときには、図 C-15 のように【新しいキーペアの作成】を選択して、キーペアを作成します。新しく作成したキーペアは、この画面からダウンロードできるので、ダウンロードしてなくさないようにしてください（再度、ダウンロードすることはできません）。

既存のキーペアを使う場合は、図 C-16 のように【既存のキーペアの選択】を選択して、利用するキーペアファイルを選択してください。この画面では、キーペアのダウンロードではなく、すでに図 C-15 の行程でダウンロード済みのキーペアを使うことになります。



図 C-15 キーペアを新規に作成する場合



図 C-16 既存のキーペアを使う場合

以上で設定完了です。EC2 インスタンスが起動し始めます（図 C-17）。

作成ステータス

❶ インスタンスは現在作成中です
次のインスタンスの作成が開始されました: i-01b853abddab42259 作成ログの表示

❷ 予想請求額の通知を受け取る
請求アラートの件数 AWS 請求書の予想請求額が設定した金額を超えた場合(つまり、無料利用枠を超えた場合)、メール通知を受け取ります。

インスタンスへの接続方法

インスタンスは作成中です。実行中状態になり、使用する準備ができるまでに数分かかることがあります。新しいインスタンスの使用時間は、すぐに始まり、インスタンスを停止または削除するまで継続します。

[インスタンスの表示]をクリックして、インスタンスのステータスを監視します。インスタンスが一度実行中状態になれば、[インスタンス]画面から接続できます。インスタンスへの接続方法の詳細は[こちら](#)。

▼ ここには、作業を始めるのに役立つリソースがあります

- Linux インスタンスへの接続方法
- AWS 無料利用枠の詳細
- Amazon EC2 ユーザーガイド
- Amazon EC2 ディスカッションフォーラム

インスタンスの作成中、次のことも行うことができます

図 C-17 EC2 インスタンス作成

■ EC2 インスタンスに SSH でログインする

EC2 インスタンスが起動したら、SSH でログインしてみましょう。まずは、EC2 コンソールの [インスタンス] で、EC2 インスタンスの IP アドレスを確認します。

「パブリック DNS」や「IPv4 / パブリック IP」と表示されているところです（図 C-18）

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with navigation links like Events, Tags, Reports, Reservations, Instances, Images, AMIs, Bandwidth Tasks, Elastic Block Store, Volumes, and Snapshots. A red arrow points to the 'Instances' link under the 'Instances' section. The main content area has tabs for 'Instances' (which is selected), 'Connect', and 'Actions'. A search bar at the top right says 'タグや属性によるフィルター、またはキーワードによる検索' (Filter by tag or attribute, or search by keyword). Below it, a table lists one instance:

Name	インスタンス ID	インスタンスタイプ	アベイラビリティゾーン
LambdaDevelopServer	i-01b853abddab42259	t2.micro	ap-northeast-1c

Below the table, there's a detailed view for the selected instance (LambdaDevelopServer):

説明	ステータスチェック	モニタリング	タグ
インスタンス ID: i-01b853abddab42259	インスタンスの状態: running	インスタンスタイプ: t2.micro	Elastic IP
アベイラビリティゾーン: ap-northeast-1c	セキュリティグループ: launch-wizard-3	インバウンドルールの表示	
パブリック DNS (IPv4): ec2-52-197-50-39.ap-northeast-1.compute.amazonaws.com	IPv4 / パブリック IP: 52.197.50.39	IPv6 IP: -	プライベート DNS: ip-172-31-26-140.ap-northeast-1.compute.internal
			プライベート IP: 172.31.26.140
			セカンダリプライベート IP: -

図 C-18 IP アドレスを確認する

確認したら、SSH で接続します。たとえば、Tera Term や Putty などのソフトを使えばよいでしょう。ここでは、Tera Term で接続してみます（図 C-19）。

接続するときは、ダウンロードしたキーペアが必要です。ユーザー名は「ec2-user」で固定です（図 C-20）。



図 C-19 接続先の入力

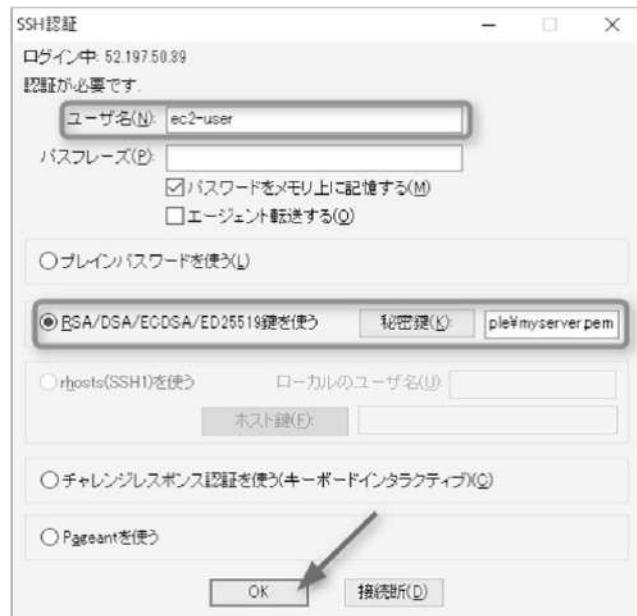
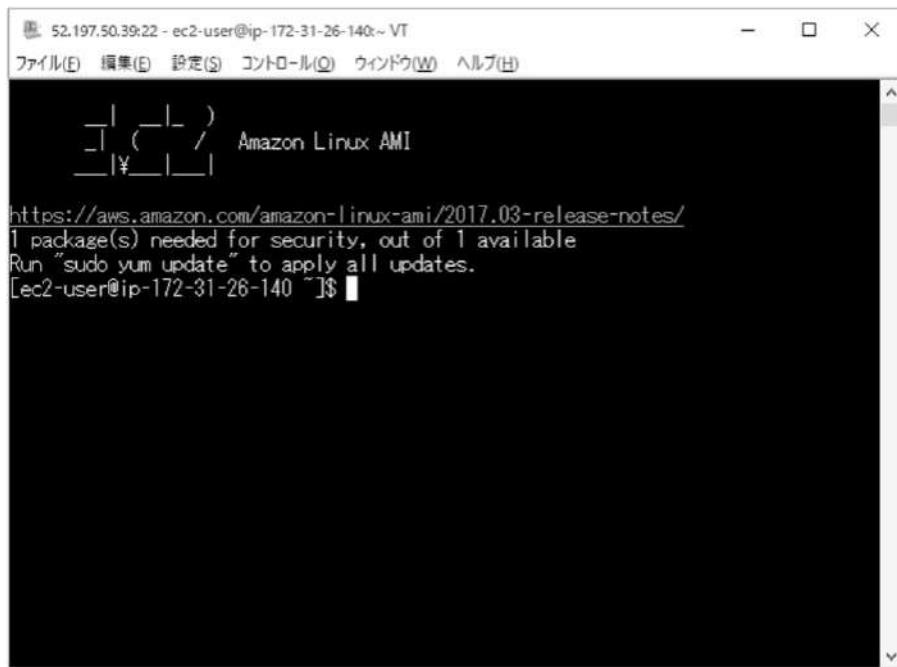


図 C-20 ユーザー名は「ec2-user」。秘密鍵も選択する

接続すると、図 C-21 のような画面になります。このシェルで、さまざまな操作ができます。

なお、ec2-user は、「sudo コマンド」を使って、root での操作ができます。



The screenshot shows a terminal window titled '52.197.50.39:22 - ec2-user@ip-172-31-26-140:~ VT'. The window has standard minimize, maximize, and close buttons at the top right. The menu bar includes 'ファイル(F)', '編集(E)', '設定(S)', 'コントロール(O)', 'ウインドウ(W)', and 'ヘルプ(H)'. The main area of the terminal displays the following text:

```
Amazon Linux AMI
https://aws.amazon.com/amazon-linux-ami/2017.03-release-notes/
1 package(s) needed for security, out of 1 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-26-140 ~]$
```

図 C-21 SSH でログインしたところ

■ Python3 のインストール

本書では、Python を使って Lambda 開発するので、Python をインストールします。

Lambda コンテナの環境は Python 3.6 ですが、本書の執筆時点（2017 年 8 月）において、Amazon Linux には、Python 3.6 がありません。そして、yum コマンドでインストールすることもできないので、ソースコードからインストールします。



Memo Python 3 のインストール

もし、近い将来 yum コマンドでインストールできるようになったなら、yum コマンドでインストールするほうがよいでしょう。その暁には、「`sudo yum install -y python36-devel python36-pip python36-virtualenv`」というコマンドだけでインストールできるはずです。



Column Python 3.4 や Python 3.5 ではいけないのか

Amazon Linux では、Python 3.4 や Python 3.5 は yum コマンドでインストールできます。こうした、少し古いバージョンではダメなのでしょうか？ 答えは、ライブラリ次第です。ライブラリが Python のみで書かれているなら、おそらく Python 3.4 や Python 3.5 で、問題ありません。しかし、第 4 章で説明している pyminizip のようなバイナリライブラリの場合、保存先のディレクトリパスなどが違うため（3.6 であるべきところが 3.4 や 3.5 になる）、Lambda デプロイパッケージを作つてアップロードしたときに、ライブラリが見つからないので、ディレクトリの調整などが必要となります。

■ Python 3.6.1 のインストール

Python 3.6.1 をソースコードからビルドしてインストールするには、次のようにします（もし 3.6.1 よりも新しいバージョンがあるなら、そのバージョンに読み替えてください）。

以下の操作手順では、Python 3.6.1 が「/usr/local」以下にインストールされます。「/usr」以下には、デフォルトでインストールされている Python 2.7 が残ります。

◎ 操作手順 ◎ Python 3.6.1 をソースからインストールする

【1】 ビルドに必要なライブラリをインストールする

ビルドに必要なライブラリをインストールします。

```
$ sudo yum -y groupinstall 'Development tools'
$ sudo yum -y install zlib-devel openssl-devel
```

【2】 作業用ディレクトリに、ソースコードダウンロードする

作業用ディレクトリを作り、そこにソースコードをダウンロードします。

```
$ mkdir python3.6
$ cd python3.6
$ wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tgz
```

[3] 展開してビルド、インストールする

展開してビルド、インストールします。

```
$ tar zxvf Python-3.6.1.tgz  
$ cd Python-3.6.1  
$ ./configure  
$ make  
$ sudo make install
```

■ virtualenv のインストール

次に、Python の実行環境を仮想的に切り替えられる `virtualenv` をインストールします。

次のコマンドを入力すると、インストールできます。

```
$ sudo /usr/local/bin/pip3.6 install virtualenv
```

ここでは、「`pip`」ではなく、「`/usr/local/bin/pip3.6`」を指定している点に注意してください。
「`pip`」と入力すると、デフォルトでインストールされている Python 2.7 の `pip` コマンドが実行されてしまうためです。

Appendix D AWS CLI の準備

AWS CLI を使うと、AWS をコマンド操作できるようになります。
ここでは、AWS CLI のインストールと設定方法を説明します。

■ AWS CLI のインストール

AWS CLI は Python で書かれたツールです。次の URL で配布されています。

【AWS CLI】

<https://aws.amazon.com/jp/cli/>

Windows の場合

AWS CLI のページの [Windows] にある [64 ビット] または [32 ビット] のリンクをクリックします（図 D-1）。すると、インストールファイルがダウンロードされます。



図 D-1 Windows 用の AWS CLI をダウンロードする

ダウンロードしたファイルを起動してインストールします（図 D-2）。すると、aws というコマンドがインストールされ、コマンドプロンプトや Windows PowerShell などから利用できるようになります。



図 D-2 AWS CLI のインストール

Mac や Linux の場合

Python 2.6.5 以降がインストールされている環境で、次のように pip コマンドを入力してインストールします。

```
$ sudo pip install awscli
```

すると、aws というコマンドがインストールされ、利用できるようになります。

なお、Amazon Linux AMI から起動した EC2 インスタンスには、デフォルトで aws コマンドがインストールされているため、改めてインストールする必要はありません。

■アクセスキーとシークレットアクセスキーを登録する

aws コマンドを使って AWS を操作するには、認証が必要です。認証には、「アクセスキー」と「シークレットアクセスキー」を使います。

まずは、Appendix A に示した手順で、ユーザーに結び付けられた「アクセスキー」と「シークレットアクセスキー」入手してください。そして、次のように aws コマンドを入力します。

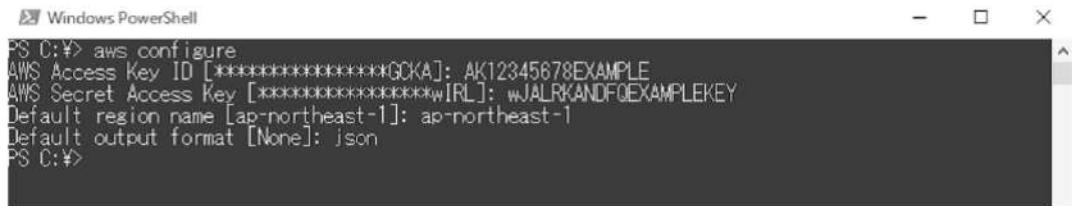
```
> aws configure
```

すると以下のように、「アクセスキー」「シークレットアクセスキー」そして「デフォルトのリージョン」「デフォルトの出力形式」の入力が求められます。

```
> aws configure
AWS Access Key ID [*****GCKA]: アクセスキーを入力
AWS Secret Access Key [*****WIRL]: シークレットアクセスキーを入力
Default region name [ap-northeast-1]: ap-northeast-1 ※東京リージョンなら「ap-northeast-1」
```

Default output format [None]: json ※「json」か「text」か「table」のいずれか

この操作によって、以降は認証情報を入力せずに済みます。



```
PS C:\$ aws configure
AWS Access Key ID [*****GCKA]: AK12345678EXAMPLE
AWS Secret Access Key [*****wIRL]: wJALRKANDFQEXAMPLEKEY
Default region name [ap-northeast-1]: ap-northeast-1
Default output format [None]: json
PS C:\$
```

図 D-3 aws コマンドの初期設定

初期設定の保存場所は、以下の通りです。

【Windows の場合】 "%UserProfile%\aws"

【Linux や Mac の場合】 ~/.aws

Λ Column 一時的なリージョンの変更

初期設定の値は、一時にオプション引数で変更できます。たとえば、別のリージョンを操作したいときは、aws コマンドで「--region」オプションを指定して、対象リージョンを別ものに一時に変更できます。

■ AWS CLI の基本書式

aws コマンドは、「AWS の API を実行する」ためのものです。そのため、コマンドは「API 名」となっています。基本的な実行書式は、次の通りです。

書式 aws サービス名 API 名 引数

たとえば、次のようにすると、webexample000 という名前の S3 バケットに example.txt というファイルをコピー（アップロード）できます。

```
> aws s3 cp example.txt s3://webexample000/
```

Appendix E

バージョニングとエイリアス

Lambda を使ってシステムを構築する場合でも、従来のシステム構築と同様に、開発やデバッグ、テストを経て、本番運用という流れになるはずです。そして本番運用を始めたあとも、その Lambda 関数に機能を加えたり、不具合を修正したりするなどの改修作業を進めていくことになるでしょう。

ひとたび運用を始めると、改修は運用とは別の環境で行うことになるはずです。つまり、「開発環境」「テスト環境」「本番環境」などで、実行環境を分ける必要があります。それを実現するのが、バージョニングとエイリアスの機能です。

■関数 ARN

Lambda 関数を作ると、その Lambda 関数には関数 ARN (Amazon Resource Name) と呼ばれる唯一無二の名前が定められます。関数 ARN は、次の書式です。

書式 arn:aws:lambda:リージョンコード:アカウント ID:function:関数名

関数 ARN は、Lambda コンソールで関数を表示したときに右上に表示されるので、そこで確認できます。たとえば、第 5 章で東京リージョンに作成した、`regist` という名前の Lambda 関数には、次の関数 ARN が割り当てられています（図 E-1）。



図 E-1 関数 ARN を確認する

Appendix E バージョニングとエイリアス

デフォルトでは、イベントが発生したときに呼び出す関数は、この関数 ARN によって定められます。

たとえば、第 5 章では、この Lambda 関数を API Gateway から呼び出されるように構成しました。このとき内部的に、arn:aws:lambda:ap-northeast-1:アカウント ID:function:regist という関数 ARN を持つ Lambda 関数を実行するように構成されます（図 E-2）。



図 E-2 イベントと関数 ARN との結び付け

■バージョニング

関数 ARN は、バージョニングに対応しており、バージョン付けして、切り替えて利用できます。たとえば、「開発版からテスト版にする」とか「テスト版から本番環境版にする」というときには、「新しいバージョンを発行」という操作をします。この操作は、右上の【アクション】メニューから実行できます（図 E-3）。



図 E-3 新しいバージョンを発行する

新しいバージョンを発行するときは、説明の入力を求められます。たとえば「テスト版」や「本番」、そして、よりわかりやすくするため「テスト版 20170801」や「本番 20170911 リリース」などのように年月日を入れた文字列を指定するとよいでしょう（図 E-4）。



図 E-4 バージョンに説明を付けて発行する

新しいバージョンを発行すると現在の環境のスナップショットが作られ、そのスナップショットに対して「バージョン 1」「バージョン 2」…のように、発行操作をするたびに、連番のバージョン番号が付けられます。

バージョン付けしたとき、それぞれのバージョンの関数 ARN は、後ろに「:バージョン番号」を付けたものとなります。開発中のバージョンには、「\$LASTEST」という特別な名前が付けられます（図 E-5）。



図 E-5 バージョニングしたときの構成

どのバージョンに対する操作なのかは、プルダウンで変更できます（図 E-6）。この画面では、右上の ARN の部分に、確かに末尾に「:1」のようにバージョン番号が付いていることも確認できます。

Appendix E バージョニングとエイリアス



図 E-6 バージョンを切り替えて操作する

バージョンのうち、編集可能なものは「\$LASTEST」の版だけです。それ以外の版は、新しいバージョンの発行という操作をしたときのスナップショットであり、構成を変更することはできません。たとえば、開発が完了して本番運用を始めるときは、新しいバージョンの発行操作をします。すると、その時点の環境のスナップショットが作られるため、その後、(\$LASTEST 環境の) ソースコードを修正するなど、改修作業をしても、本番環境に影響を与えることがありません。

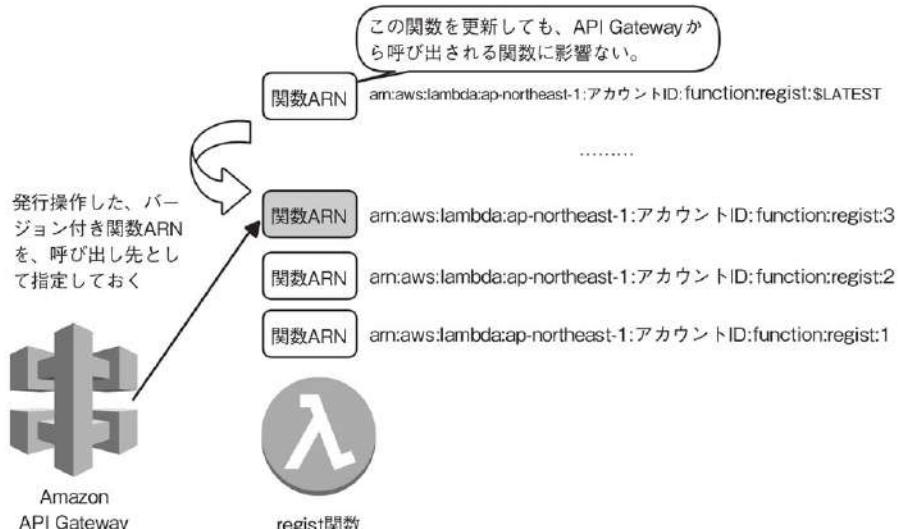


図 E-7 バージョン発行した版を本番環境に割り当てる（推奨されない）

■エイリアス ARN

では、本番運用するときには、図 E-7 のようにバージョン発行し、バージョン付きの関数 ARN をイベントの呼び出し先として設定すればよいのかというと、それは、あまりよくありません。なぜなら、バージョンアップしたときには、バージョン付きの関数 ARN が変わるので、イベントの設定し直しになるからです（図 E-8）。

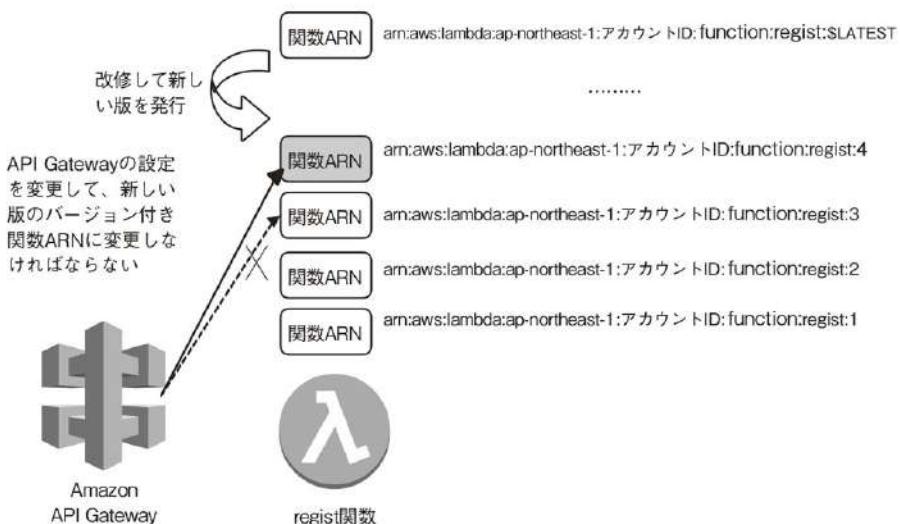


図 E-8 バージョン付きの関数 ARN をイベントの呼び出し先として直接設定するのはよくない

この問題を解決するのが、エイリアス ARN です。エイリアス ARN は、関数 ARN へのショートカットです。次の書式の名称です。

書式 arn:aws:lambda:リージョンコード:アカウント ID:function:関数名:エイリアス名

イベントソースに対して呼び出し先の関数を設定するときは、関数 ARN を直接指定するのではなく、呼び出したいバージョンと結び付けたエイリアスをあらかじめ作っておき、そのエイリアス ARN を指定するようにします。つまり、間にエイリアス ARN を挟みます（図 E-9）。

Appendix E バージョニングとエイリアス

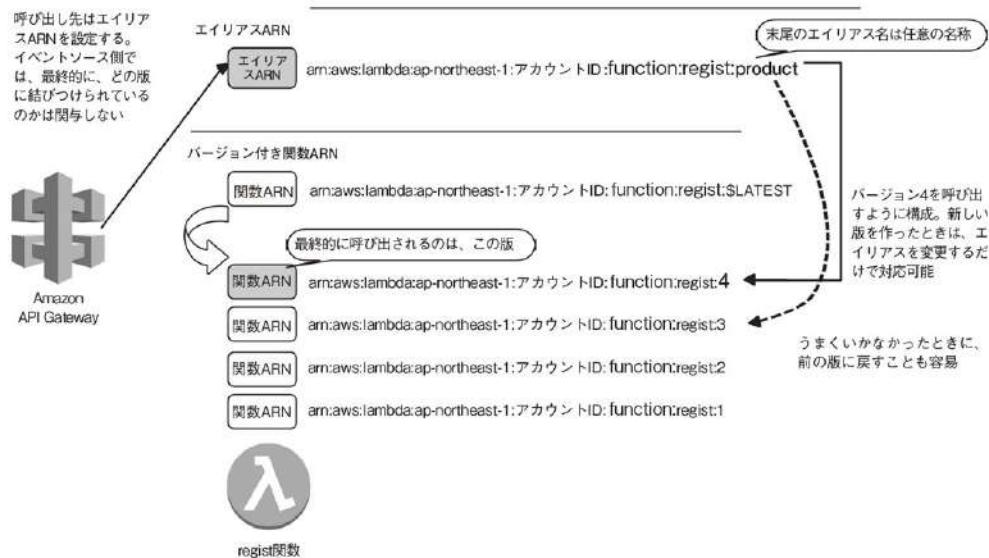


図 E-9 エイリアス ARN を使う

このようにすればバージョンが変わったときは、エイリアス ARN の指すものを変更するだけですみます。イベントソースから見えている呼び出し先は、エイリアス ARN のまま変わらないので設定変更する必要がありません。エイリアス ARN を使えば、バージョンの更新も簡単ですし、もし、うまく行かなかったときに、前のバージョンに戻すのも容易です。

エイリアス ARN を使うには、[アクション] メニューの [エイリアスの作成] から操作してエイリアスを作成します（図 E-10）。エイリアスの作成では、任意のエイリアス名と、どのバージョンを示すのかを指定します（図 E-11）。



図 E-10 エイリアスを作成する



図 E-11 エイリアス名と呼び出し先となるバージョンを指定する

エイリアスが実際に呼び出す版は、【設定】メニューから、あとで好きなとき切り替えられます（図 E-12）。



図 E-12 エイリアスの指すバージョンを変更する

本編では、話を簡単にするため、バージョニングやエイリアス ARN の機能は使っていませんが、運用まで考えるなら、積極的にエイリアス ARN を利用しましょう。

索引

Symbols

SLATEST	318
2回の再試行	63

A

Access-Control-Allow-Origin	202
add_item メソッド	174
Amazon Linux	54
Amazon Linux AMI	297
AmazonAPIGatewayAdministrator	289
AmazonDynamoDBFullAccess	295
AmazonS3FullAccess	295
AmazonSESFullAccess	289, 295
AmazonSNSSFullAccess	289, 295
AmazonSQSFullAccess	289, 295
API Gateway	19, 126
API Gateway のイベント	127
API Gateway の作成	129
API 名	135
AWS CLI のインストール	312
AWS Data Pipeline	216
aws-cli コマンド	180
AWSLambdaBasicExecutionRole	34, 291, 293
AWSLambdaDynamoDBExecutionRole	34, 67
AWSLambdaFullAccess	33, 281
AWSLambdaKinesisExecutionRole	34, 67
AWSLambdaVPCAccessExecutionRole	34

B

Boto3	86, 112
-------	---------

C

CloudWatch Logs	24, 26, 46
CloudWatch イベント	52, 71
context	28
Context オブジェクト	28
CORS	203

cron 式	73
Cross-Origin Resource Sharing	203

D

Dead Letter Queue	52
delete_queue_batch メソッド	267
delete_queue メソッド	267
delete メソッド	267, 269
delte_item メソッド	174
DLQ	52, 64
DLQ リソース	42
DynamoDB	61, 126, 153
DynamoDB テーブル	212
DynamoDB のデータ型	155
DynamoDB へのアクセス権限	130

E

EC2 インスタンス	14, 298
Elastic Load Balancing	14
Elastic Network Interface	58
ELB	14
ENI	58
event	27

F

FIFO キュー	210
----------	-----

G

generate_presigned_url メソッド	184
get_item メソッド	178
get_queue_by_name メソッド	240

I

IAM	30
iam : CreateRole	35
IAM グループ	284
IAM ユーザー	24, 30, 281
IAM ユーザーのサインインリンク	286
IAM ロール	24, 30, 291, 300

Identity and Access Management	30	Python3 のインストール	309
J			
json.dumps メソッド	30, 178	query メソッド	178
JSON 形式	25		
K			
KMS キー	42		
L			
Lambda	15	S3 Put	104
lambda : InvokeFunction	32	s3.bucket.name	98
lambda-canary-python3	71	s3.object.key	98
LAMBDA_TASK_ROOT	107	S3 : ObjectCreated	96
Lambda 開発者アカウント	281	S3 : ObjectRemoved	96
Lambda 開発マシン	297	S3 : ReducedRedundancyLostObject	96
Lambda 関数の構造と設計	24	S3 のイベント	88
Lambda 関数のコード	38, 75	S3 のイベント処理	85, 99
Lambda 関数の作成	24	S3 バケット	69
Lambda 関数の実行	43, 62	S3 バケット間の移動	114
Lambda 関数の書式	26	S3 バケットのイベント	95
Lambda 関数の動作	26	S3 バケットの作成	180, 222
Lambda コンテナ	52, 53	S3 バケットの名前	89
Lambda 実行ロール	291	scan メソッド	178
Lambda デプロイパッケージ	87, 107	send_email メソッドのパラメータ	198
Lambda プロキシ統合	132, 139	send_mail メソッド	197
Linux コンテナ	54	SES	126, 187
logs : CreateLogGroup	34	Simple Email Service	126
logs : CreateLogStream	32, 34	SNS トピック	17, 64, 69, 206, 247
logs : PutLogEvents	32, 34	SQS	17, 206, 209
M			
MaxNumberOfMessages	266	SQS のキュー	64, 226
MessageAttributeNames	266	SSH	297
microservice-http-endpoint-python3	134	Static website hosting	146
N			
NAT ゲートウェイ	60	T	
P			
parse.parse_qs メソッド	175	Table メソッド	172
publish メソッド	257		
put_item メソッド	176		
pyminizip	87, 107, 112	update_item メソッド	173, 174, 241, 268, 279

V

- virtualenv 111
 VPC 42, 52, 58
 VPCへのアクセス 58

W

- Web フォーム 145

Z

- zlib ライブラリ 107

あ

- アカウント ID 286
 アクセスキー 313
 アクセス権 31, 283, 294
 アクセス権限 288
 アクティブトレース 42
 アトミックカウンタ 161
 アベイラビリティゾーン 154
 暗号化 ZIP 86, 114

い

- イベント 26, 62
 イベントソース 19, 52, 68
 イベントタイプ 100
 イベントドリブン 19
 イベントの情報 26
 イベントの発生 53
 イベント引数 27
 イベント引数の構造 139
 インスタンスタイプ 14
 インラインポリシー 93

え

- エイリアス ARN 319
 エスケープ文字 218

か

- 課金 60
 環境変数 39
 関数 15
 関数 ARN 315
 関数の設定 37, 75
 関数ハンドラ 77

- 管理者アカウント 30
 管理ポリシー 288

き

- キーペアの選択 306
 キャパシティーユニット 156
 キュー 17
 キューの属性 227
 キューのポーリング処理 256
 キュー名 227

く

- クロスオリジン 199

け

- 結果整合性 156
 権限 281

こ

- 項目 154
 コスト削減 17
 コンテキスト 26
 コンテキストの情報 26
 コンテキスト引数 28
 コンテナ 15
 コンテナ環境 54
 コンテナの再利用 56

さ

- サーバーレス 12
 サーバーレスアーキテクチャ 15
 再試行と DLQ 63
 最大稼働時間 12
 最大稼働時間 16
 サフィックス 101
 サブネット 58, 300

し

- シークレットアクセスキー 287, 313
 実行環境 15
 実行順序の保障 65
 手動実行 44
 署名付き URL 126, 179

す

- スケジュール 71
 スケジュール式 73
 ステート 12
 ステートレス 12, 16
 ストリームベース 53, 62
 ストリームベースの実行 66
 スナップショット 318

せ

- セカンダリインデックス 212
 セキュリティ 136
 セキュリティグループ 58
 セキュリティグループの設定 305
 セキュリティポリシー 61
 設計図の選択 36

そ

- ソートキー 155
 属性 154

た

- タイムアウト 41
 タグ 40, 235

つ

- 通知 17

て

- 定義済みの環境変数 54
 テストイベント 24
 テストイベントの設定 78
 テストイベントの定義 44
 デッドレターキュー 229
 アプロイされるステージ 135
 アプロイパッケージ 120

と

- 同期呼び出し 52, 53, 63
 同時実行可能な最大数 66
 ドメインの検証 189
 トリガー 26, 36
 トリガーの選択 36

な

- 名前 38

ね

- ネットワーク 300

は

- バージョニング 316
 パーティションキー 155
 バウンスマール 206, 270
 バケット 100
 ハンドラ 26, 40, 235

ひ

- 非同期実行の実例 70
 非同期呼び出し 52, 53, 63
 標準キュー 210

ふ

- ブッシュモデル 53, 62
 プライマリーキー 154
 プレフィックス 101

へ

- 算等性 65

め

- メールアドレスの検証 190
 メールの送信 187
 メールの同報システム 210
 メールの同報送信 207
 メールボックスシミュレーター 219
 メモリ 41

よ

- 読み取りキャパシティー 178

ら

- ランタイム 38

り

- リージョンの切り替え 188
 リージョンの変更 314

索引

る

- ルートアカウント 30
- ルールタイプ 73
- ルール名 73

れ

- 例外 47
- 例外処理 177

ろ

- ロール 40, 77, 235
- ログの確認 79

●著者プロフィール

大澤文孝（おおさわふみたか）

テクニカルライター／プログラマー、情報処理資格としてセキュリティスペシャリスト、ネットワークスペシャリストを取得。Webシステムの設計・開発とともに、長年の執筆活動のなかで、電子工作、Webシステム、プログラミング、データベースシステム、パブリッククラウドに関する書籍を多数執筆している。近年のクラウド関連の書籍としては、『さわってわかる機械学習 Azure Machine Learning 実践ガイド』（共著：日経BP）、『Amazon Web Services クラウドデザインパターン実装ガイド』（共著：日経BP）『Amazon Web Services はじめる Web サーバ』（工学社）などがある。

●お断り

ITの環境は変化が激しく、Amazon Web Services の展開するパブリッククラウドの世界は、最も変化の激しい先端分野の一つです。本書に記載されている内容は、2017年8月時点のものですが、サービスの改善や新機能の追加は、日々行われているため、本書の内容と異なる場合があることは、ご了承ください。また、本書の実行手順や結果については、筆者の使用するハードウェアとソフトウェア環境において検証した結果ですが、ハードウェア環境やソフトウェアの事前のセットアップ状況によって、本書の内容と異なる場合があります。この点についても、ご了解いただきますよう、お願ひいたします。

●正誤表

インプレスの書籍紹介ページ「<https://book.impress.co.jp/books/1116101044>」からたどれる「正誤表」をご確認ください。これまでに判明した正誤があれば「お問い合わせ／正誤表」タブのページに正誤表が表示されます。

●スタッフ

AD／装丁：岡田 章志 + GY

本文デザイン／制作／編集：TSUC

図版制作：プロトコード/TSUC

本書のご感想をぜひお寄せください

<https://book.impress.co.jp/books/1116101044>



読者登録サービス アンケート回答者の中から、抽選で商品券(1万円分)や図書カード(1,000円分)などを毎月プレゼント。当選は賞品の発送をもって代えさせていただきます。

■ 商品に関する問い合わせ先

インプレスブックスのお問い合わせフォームより入力してください。

<https://book.impress.co.jp/info/>

上記フォームご利用頂けない場合のメールでの問い合わせ先
info@impress.co.jp

●本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。

●電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に問しましてはお答えできませんのでご了承ください。

●インプレスブックス (<https://book.impress.co.jp/>) では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。

■ 落丁・乱丁本などの問い合わせ先

TEL 03-6837-5016 FAX 03-6837-5023
service@impress.co.jp

(受付時間／10:00-12:00、13:00-17:30 土日、祝祭日を除く)

●古書店で購入されたものについてはお取り替えできません。

■ 書店／販売店の窓口

株式会社インプレス 受注センター
TEL 048-449-8040

FAX 048-449-8041

株式会社インプレス 出版営業部

TEL 03-6837-4635

エーダブリュウエスラムダジッセンガイド

AWS Lambda実践ガイド

2017年10月21日 初版発行

2020年04月21日 第1版第2刷発行

著者 大澤文孝
おおさわ ふみたか

発行人 小川亨

編集人 高橋隆志

発行所 株式会社インプレス
〒101-0051 東京都千代田区神田神保町一丁目105番地
ホームページ <https://book.impress.co.jp/>

本書は著作権法上の保護を受けています。本書の一部あるいは全部について（ソフトウェア及びプログラムを含む）、株式会社インプレスから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

Copyright © 2017 Fumitaka Osawa. All rights reserved.

印刷所 株式会社ウイル・コーポレーション

ISBN978-4-295-00252-9 C3055

Printed in Japan