サーバーレスでプログラムを動かす

Lambdaを使うと、EC2インスタンスを使うことなくプログラムを稼働できます。

常時実行している必要がないサービスは、Lambdaを使って構築すると、コストを削減できるばかりか、開発工数も抑えられます。

●ファイルが保存されたときに、サムネイルを作りたい

【導入】

N社では、Webブラウザーで参照できる商品カタログを作成しています。商品カタログの運用には、S3の静的ウェブホスティング機能を使っており、商品の写真をS3にアップロードしています。

オリジナルの商品写真はサイズが大きいので、サムネイルも一緒にアップロードしています。サムネイルは担当者がフォトレタッチソフトを用いて、手作業で作っているので、とても手間がかかっています。

この手間を軽減するため、画像をアップロードしたら、自動的にサムネイルが作れるといいと考えています。

○利用サービス

S3、Lambda

○解説

AWS以外のソリューションで、こうした機能を実現する場合、すぐ思いつく方法は、サムネイルを作成するWebアプリケーションを作り、Webフォームから画像をアップロードする手法です（図3-3-1）。

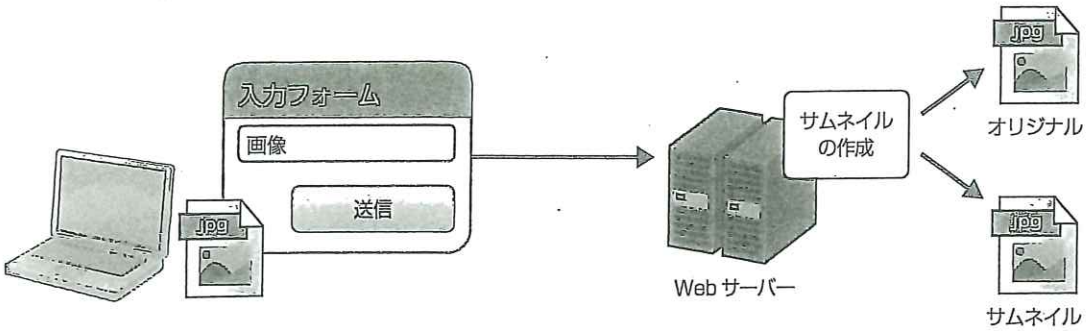

図3-3-1 Webアプリケーションでサムネイルを作る

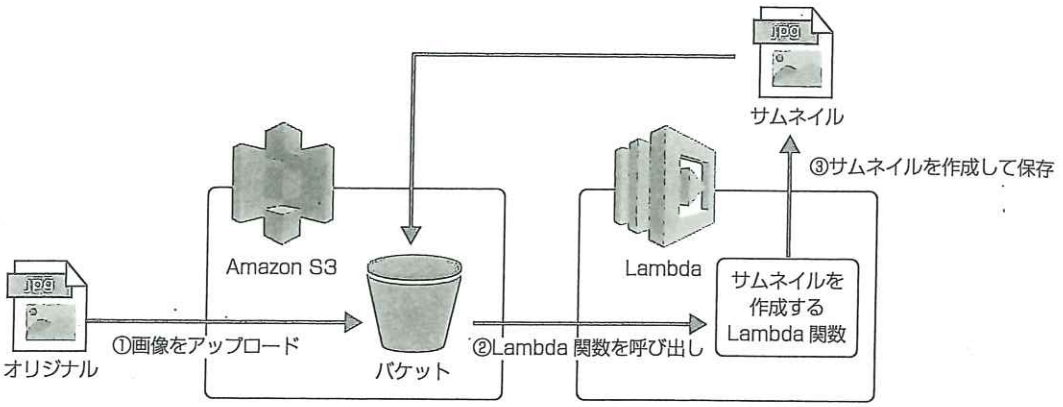

図 3-3-2 Lambda 関数を使ってサムネイルを作成する

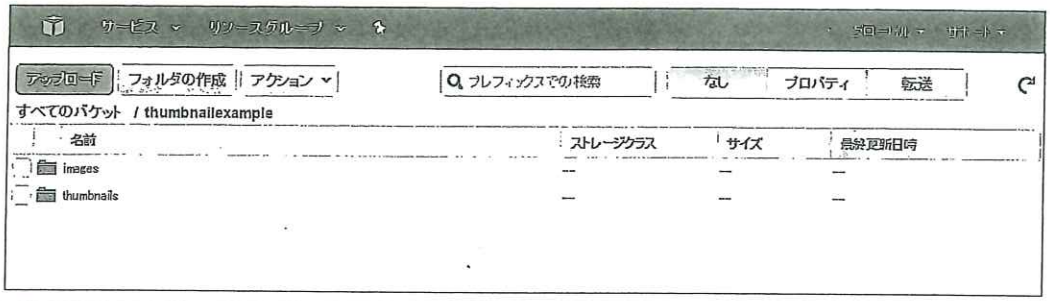

図 3-3-3 S3 バケットを作成し、images ディレクトリーと thumbnails ディレクトリーを作成しておく

しかし AWS なら、もっとシンプルな方法で作れます。それは、Lambda を使う方法です。

S3 にファイルをアップロードすると、それをイベントのトリガにして Lambda 関数を実行できます。つまり、サムネイルを作成して保存するプログラムを Lambda 関数として作っておけば、画像がアップロードされたタイミングでサムネイルを作れます（図 3-3-2）。

この方法だと、S3 に画像を保存する方法は問いません。AWS マネジメントコンソールでも、Cyberduck のようなツールでも、もしくは、CLI や何か開発したアプリケーションなどからでも、S3 に画像が保存されれば、サムネイルが作成されます。

○手順

【S3 バケットを準備する】

まずは、何かしらの S3 バケットを用意しておきます。

そして、この images ディレクトリーに JPEG 形式のファイルを保存したときに、thumbnails ディレクトリーに同名でサムネイル画像が作られるようにします。images ディレクトリーと thumbnails ディレクトリーをあらかじめ作成しておいてください（図 3-3-3）。

【Lambda 関数を実行するロールを作成する】

Lambda 関数は、何らかの IAM ロールの権限で実行されます。これから作る Lambda 関数を実行するには、次の権限が必要です。

①Lambda 関数自体を実行するための基本的な権限
→ AWSLambdaBasicExecutionRole ポリシー

②図 3-3-3 で作成した S3 バケットに対するフルアクセス権
→ AmazonS3FullAccess ポリシー

メモ AmazonS3FullAccess ポリシーは、すべての S3 バケットに対するアクセス権を与えるため、この権限を与えることは、あまり望ましい選択肢ではありません。本書では、話を簡単にするため、AmazonS3FullAccess ポリシーを使いますが、より安全に使うには、図 3-3-3 の S3 バケットにだけアクセスできるカスタムポリシーを作成して、そのポリシーを適用するのが望ましい運用です。

そこで、これらの権限を持つ IAM ロールを作成します。まずは、AWS マネジメントコンソールの IAM 画面から［ロール］メニューをクリックして開き、［新しいロールの作成］をクリックしてください（図 3-3-4）。

ロール名を入力します。ここでは「myLambdaS3Role」としておきます（図 3-3-5）。

ロールタイプを選びます。ここでは、Lambda 関数が AWS のさまざまなサービスを呼び出せるようにするロールを作りたいので［AWS Lambda］を選択します（図 3-3-6）。

次にポリシーをアタッチします。ここでは、二つのポリシーをアタッチします。

一つは、「AWSLambdaBasicExecutionRole」です。これは Lambda のログ出力先となる CloudWatch Logs へのアクセス権を付与します。フィルターのところに「lambda」と入力すると、lambda という文字列を含むポリシーが表示されるので、その中から選びます（図 3-3-7）。

そしてそのまま今度はフィルターのところに「S3」と入力して S3 関連のポリシーを表示し、「AmazonS3FullAccess」にチェックを付けます。これが S3 に対するフルアクセス権となります。

AWSLambdaBasicExecutionRole と AmazonS3FullAccess の二つにチェックを付けたなら、［次のステップ］をクリックして、次の画面に進んでください（図 3-3-8）。

すると確認画面が表示されます。［ロールの作成］ボタンをクリックして、ロールの作成を完了してください（図 3-3-9）。

【Lambda 関数を作る】

ロールができたら、Lambda 関数を作成していきます。AWS マネジメントコンソールで［コンピューティング］のなかの［Lambda］を開いてください（図 3-3-10）。

まだ Lambda 関数を一つも作っていないときは、図 3-3-11 の画面が表示されるので［Get Started Now］をクリックしてください。

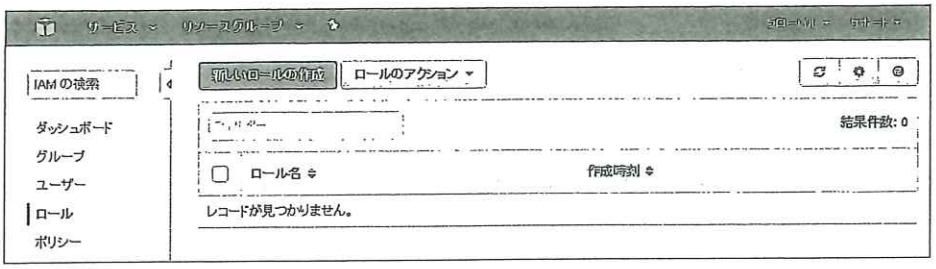

図 3-3-4 新しいロールを作成する

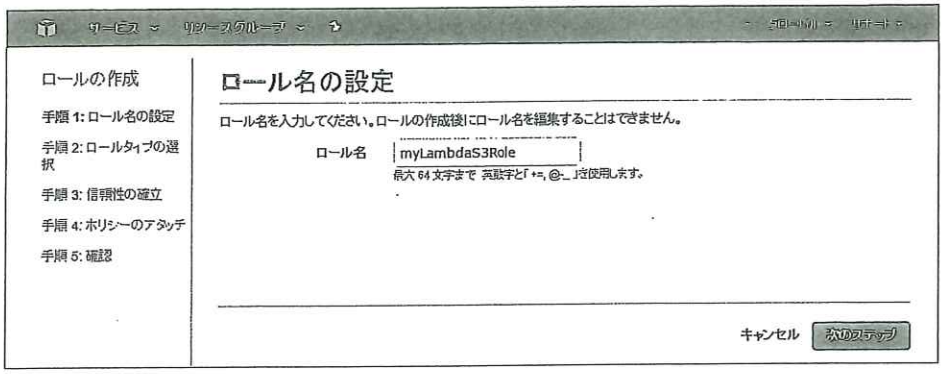

図 3-3-5 ロール名を付ける

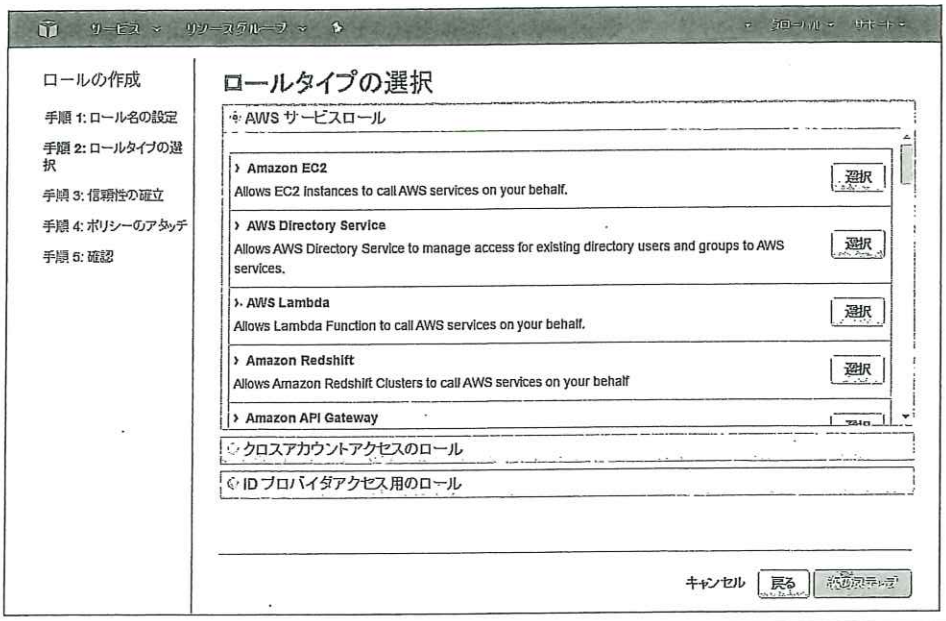

図 3-3-6 AWS Lambda ロールタイプを選択する

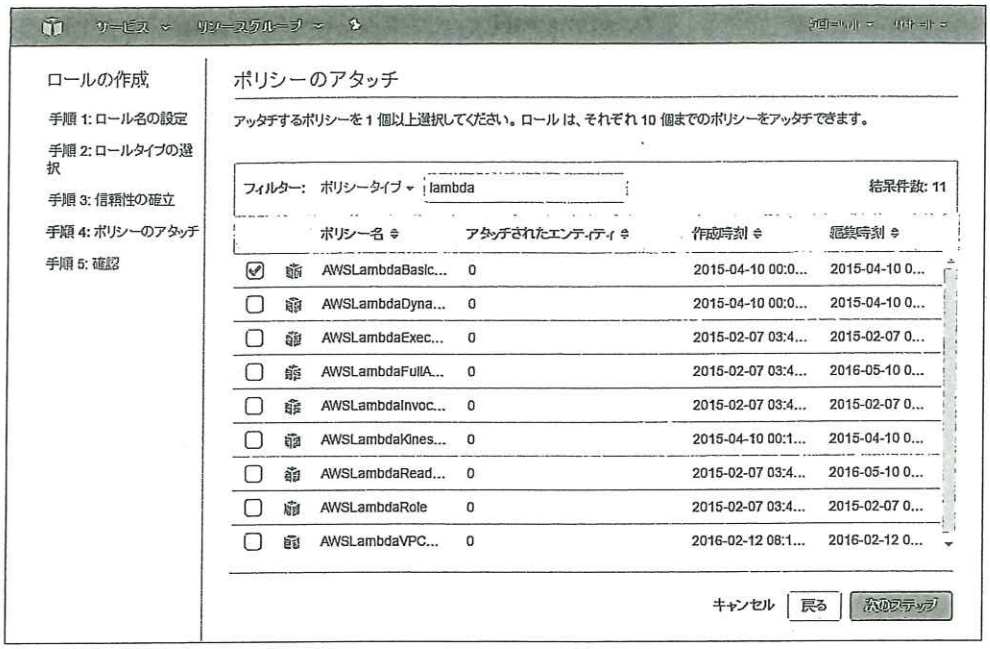

図 3-3-7 AWSLambdaBasicExecutionRole を選ぶ

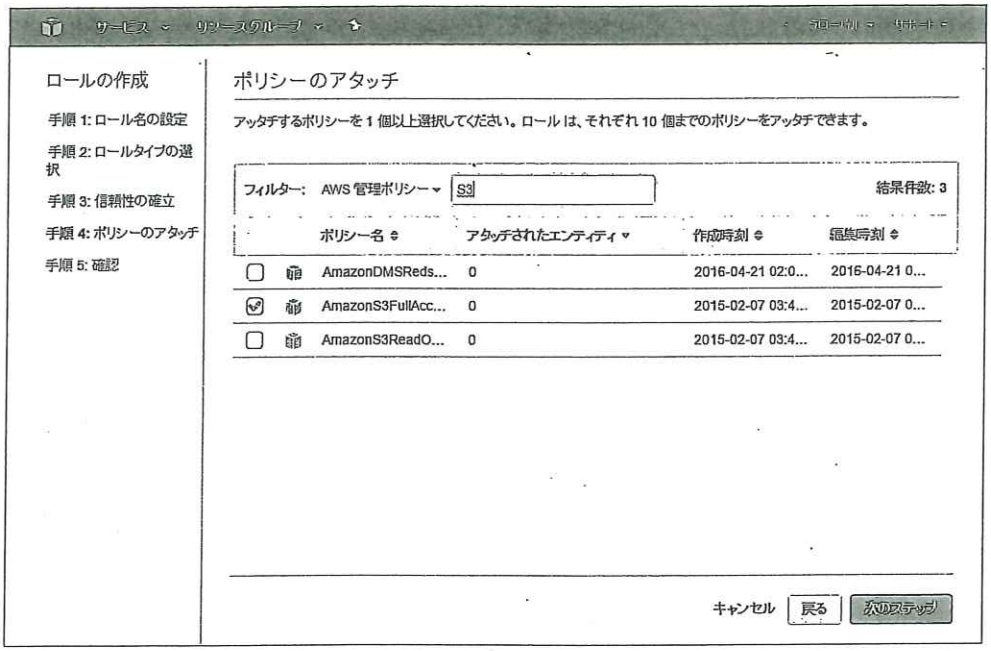

図 3-3-8 AmazonS3FullAccess を選ぶ

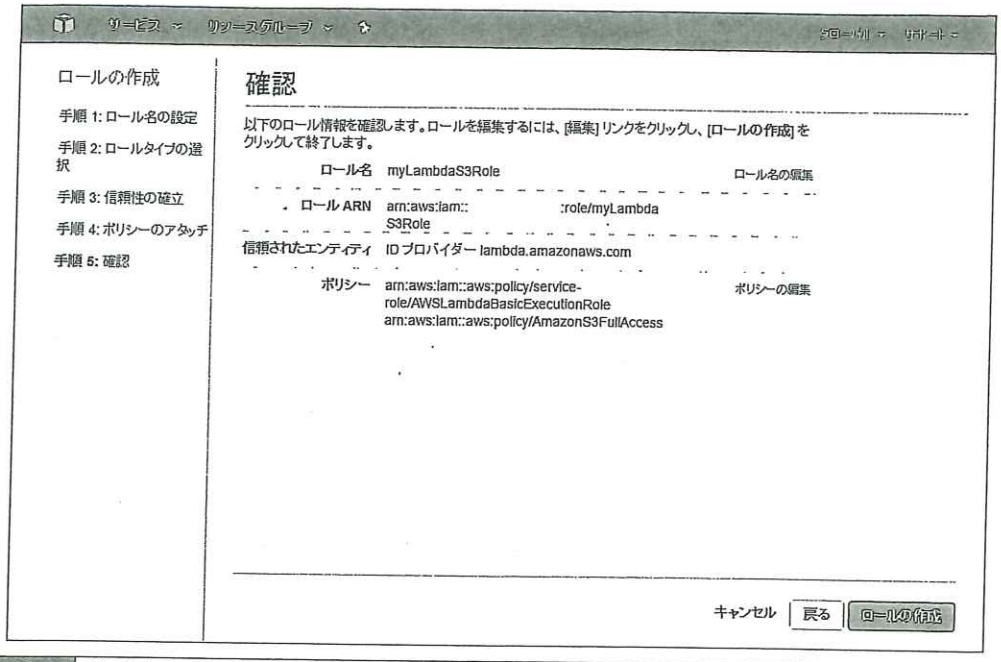

図 3-3-9 作成するロールの確認

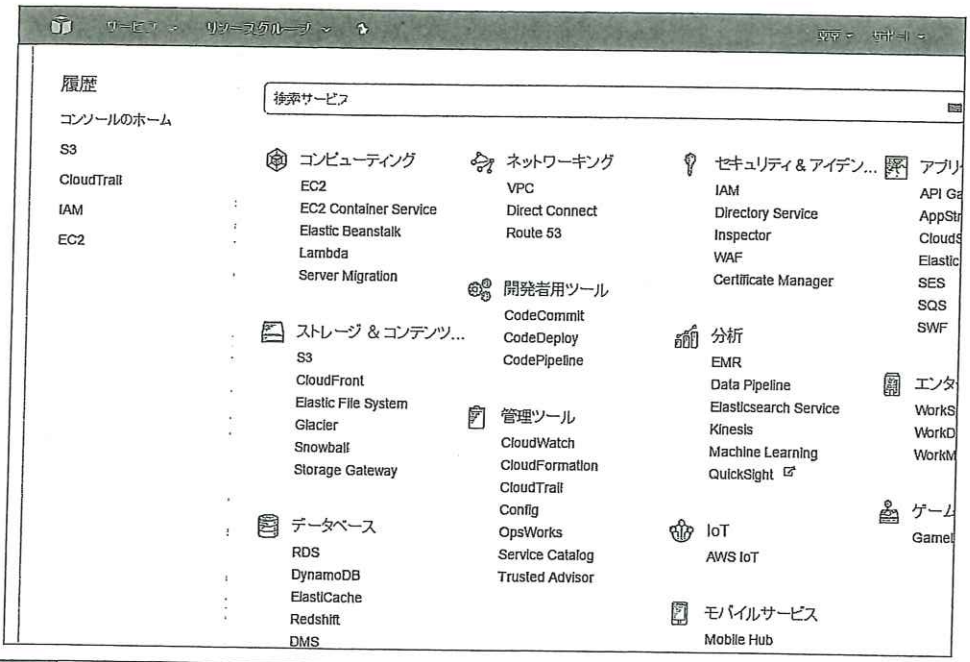

図 3-3-10 Lambda を開く

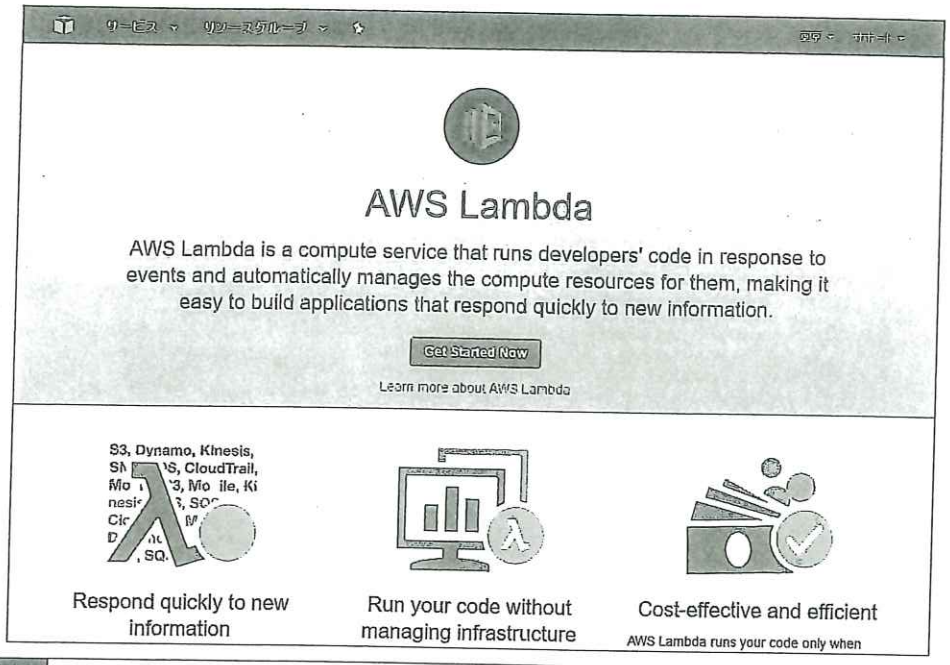

図 3-3-11 Get Started Now をクリックする

メモ すでに作っているときは［Functions］メニューから［Create a Lambda Function］をクリックすることで作成してください（後掲の図 3-3-33 を参照）。

すると、Lambda 関数の作成画面になります。Lambda 関数は「blueprint」と呼ばれるひな形（青写真）を基に作ることができます。Lambda 関数を作るときは、最も機能が近い blueprint を選ぶのが簡単です。

S3 に関する blueprint は、2016 年 11 月時点で二つあり、フィルターのところに「s3」と入力すると、S3-get-object と s3-get-object-python の二つに絞り込めます。ここでは、Node.js（JavaScript）を利用した「s3-get-object」を選択します（**図 3-3-12**）。

すると、トリガの選択画面（**図 3-3-13**）になります。次のように設定してください。

・Bucket

対象とするバケットを選択します。図 3-3-3 で事前に用意しておいたバケットを選択してください。

・Event type

トリガの種類を選択します。ここでは、ファイルが配置されたときにトリガを実行したいので「Object Created（ALL）」を選択してください。

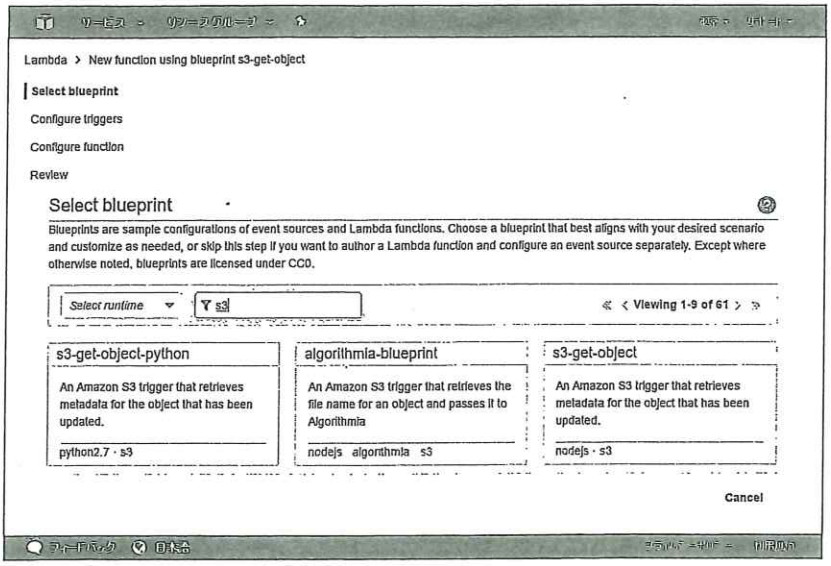

図 3-3-12 s3-get-object を選択する

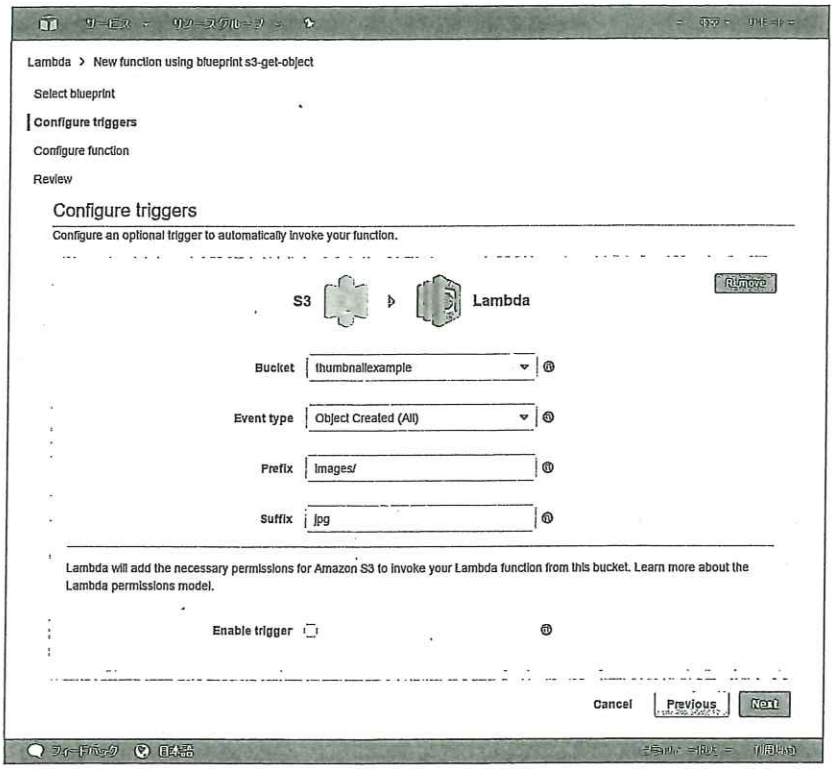

図 3-3-13 トリガを設定する

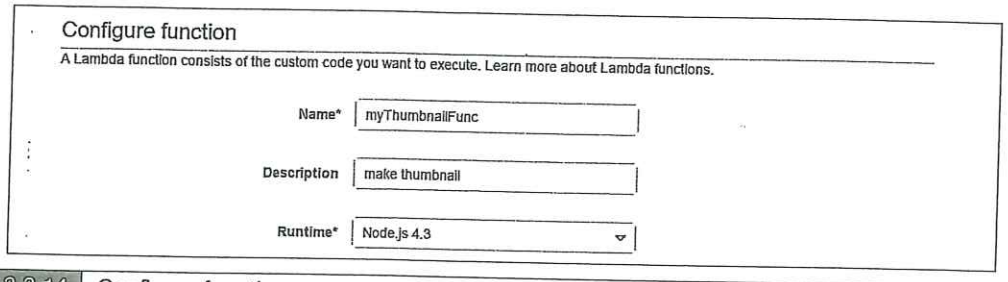

図 3-3-14 Configure function

・Prefix

トリガをかけるファイル名の接頭辞を設定します。ここでは「images/ ファイル名 .jpg」のように「images から始まるファイルが置かれたとき」に、トリガを発生したいので、「images/」と入力してください。

・Suffix

トリガをかけるファイル名の接尾辞を設定します。ここでは「ファイル名 .jpg」のように末尾が「jpg で終わるファイルが置かれたとき」にトリガを発生したいので「jpg」と入力してください。

・Enable trigger

作成したトリガをすぐに有効にするかどうかの設定です。チェックを付けると、すぐにこのトリガが有効になりますが、多くの場合、コードを記述して少しデバッグしてから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。

次の画面では、Lambda 関数を作り始めます。次の設定があります。

① Configure function

Lambda 関数名や実行環境を設定します。

Name には、任意の名前を入力します（図 3-3-14）。ここでは「myThumbnailFunc」という名前にしました。Description は解説文です。何でもかまいません。

Runtime は実行環境です。Lambda 関数は「Java」「Python」「Node.js（JavaScript）」のいずれかの言語を使って記述できます。ここでは「Node.js 4.3」を選択しました。

② Lambda function code

ここに Lambda 関数のコードを書きます。「Edit code inline」を選択すると、この画面のテキストボックスでコードを書けます（図 3-3-15）。

それ以外に「Upload a .ZIP file」を選択して ZIP ファイルをアップロードしたり、「Upload a file from Amazon S3」を選択して S3 にアップロードしたファイルを取り込んだりできます。

blueprint から選んだときは、ここにひな形のプログラムが書かれているので、これをあとで編集することにして、このままの状態にしておきます。

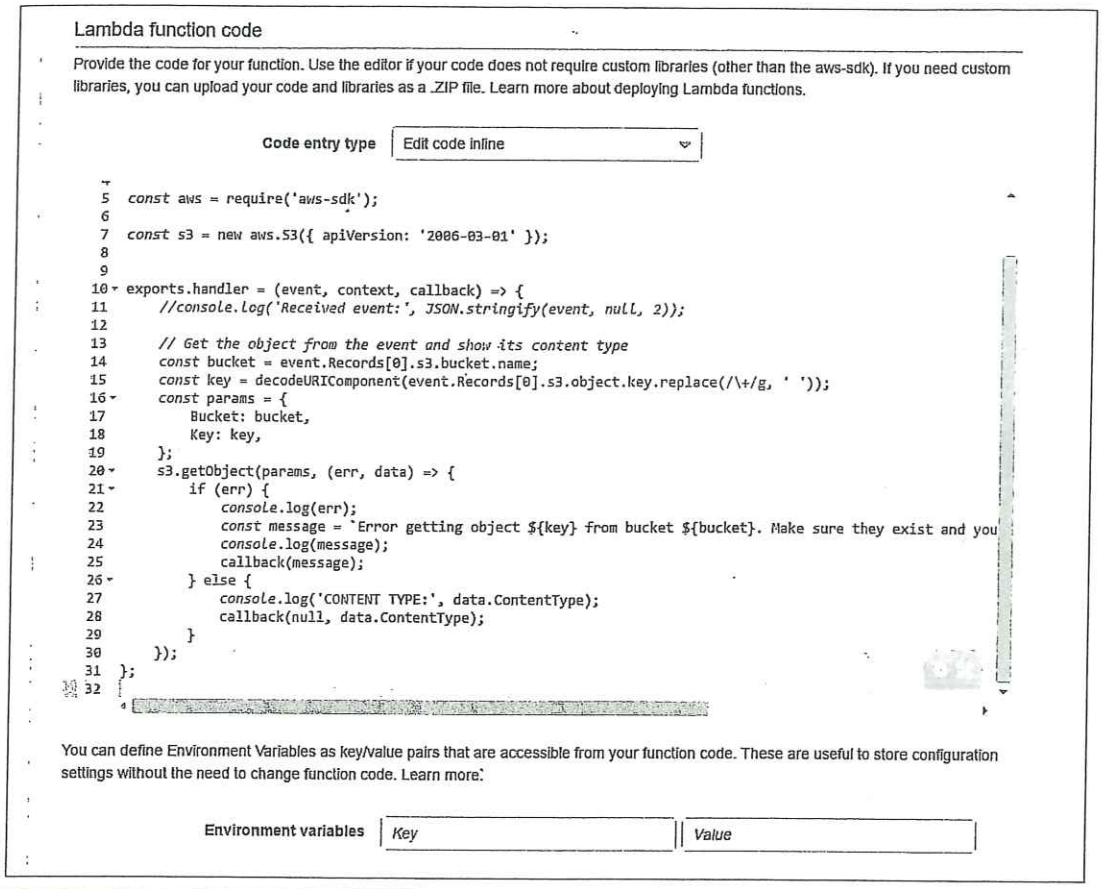

図 3-3-15 Lambda function code

メモ 》インラインで書けるのは、標準ライブラリしか利用しない場合です。何かライブラリを使いたいときは、そのライブラリを含んだ ZIP ファイルを作成し、ライブラリも一緒にアップロードする必要があります。

メモ 》その下にある「Environment variables」は、Lambda 関数に対して環境変数を設定できる機能です。外部から Lambda 関数に対して設定情報を与えたいときなどに利用できます。

③ Lambda function handler and role

Lambda 関数の実行ハンドラとロールを設定します。

(a) Handler

プログラムを実行するときに呼び出すメソッド名を設定します。デフォルトは「index.handler」となっており、これは「handler」というメソッドを実行するという意味です。

blueprint のコードは、メインルーチンが次のように定義されています。

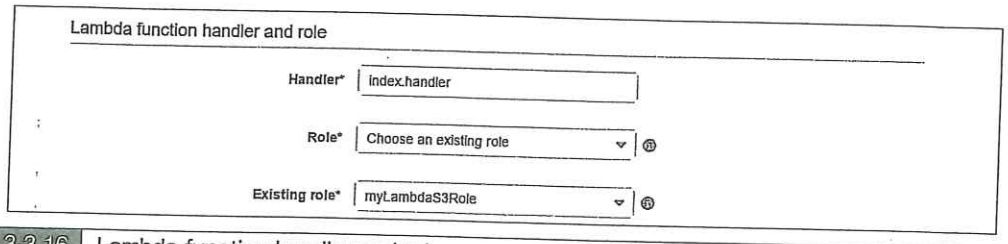

図 3-3-16 Lambda function handler and role

```
exports.handler = (event, context, callback){
…プログラム…
}
```

「index.handler」は、この「handler」を指しています。もし、プログラムのメソッド名を変更したときは、この設定をそれに合わせる必要があります。

(b) Role、Role name、Policy templates

この Lambda 関数を実行するロールを指定します。三つの選択肢があります。

・Choose an existing role

既存のロールから選択します。

・Create new role from template(s)

Policy tempaltes で選択したポリシーテンプレートをひも付けた新しいロールを作ります。作成するロール名は、「Role name」のところで決めます。

・Create a custom role

カスタムロールを指定して、新しいロールを作成します。このとき作成するロール名は「Role name」のところで決めます。

ここではすでにロールを作っているので、[Choose an existing role] を選び、先に作成しておいた myLambdaS3Role を選択してください（図 3-3-16）。

④ Advanced settings

Lambda の実行環境を設定します。

Memory の部分で、割り当てるメモリーを設定します。最小値は 128MB で、デフォルトはそれに設定されていますが、ここで作成する Lambda 関数は画像を扱うため、それではメモリーが足りないかもしれません。そこでここでは、256MB を設定することにします。メモリーを増やすと、利用可能な CPU パワーも容量に比例して増加します。

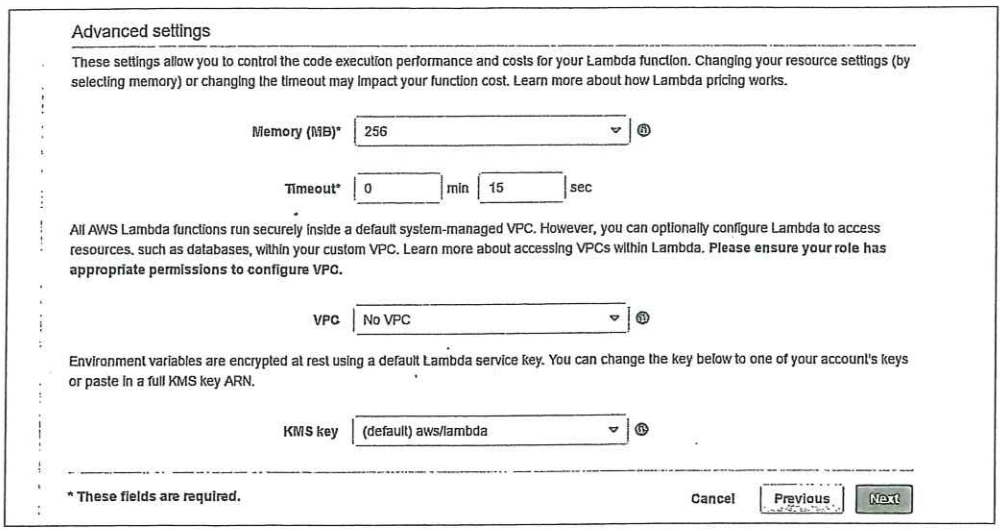

図 3-3-17 Advanced settings

メモ 》 Lambda 関数は、割り当てたメモリーと実行時間によって金額が異なります。大きなメモリーを割り当てると費用が高くなります。

「VPC」の項目は、この Lambda 関数を VPC の内部で実行するときに指定します。例えば、EC2 インスタンスや RDS などと通信したいときは、VPC を指定する必要があります。 ここで作る Lambda 関数は、VPC 上のリソースと通信する必要がないので、ここでは［No VPC］とします。

なお、Lambda から VPC 上のリソースにアクセスする場合は、特有の制約があります。設計上やむを得ない場合は https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/vpc.html を参照して、制約事項を確認しておくことをお勧めします。

［KMS key］の項目は、環境変数を、どのキーで暗号化するかという設定です。デフォルトの［aws/lambda］にしておきます（図 3-3-17）。

以上を設定すると、確認画面が表示されます。［Create function］をクリックして Lambda 関数を作成してください（図 3-3-18）。

すると、Lambda 関数が作られます。この時点では、まだトリガが［Disable］になっており、無効になっていることが分かります（図 3-3-19）。

【サムネイルを作成するコードを記述する】

作成した Lambda 関数は、［Code］タブで編集できます。

サムネイルを作成するコードを List 3-3-1 に示します（図 3-3-20）。これを入力して、［Save］ボタンをクリックしてください。

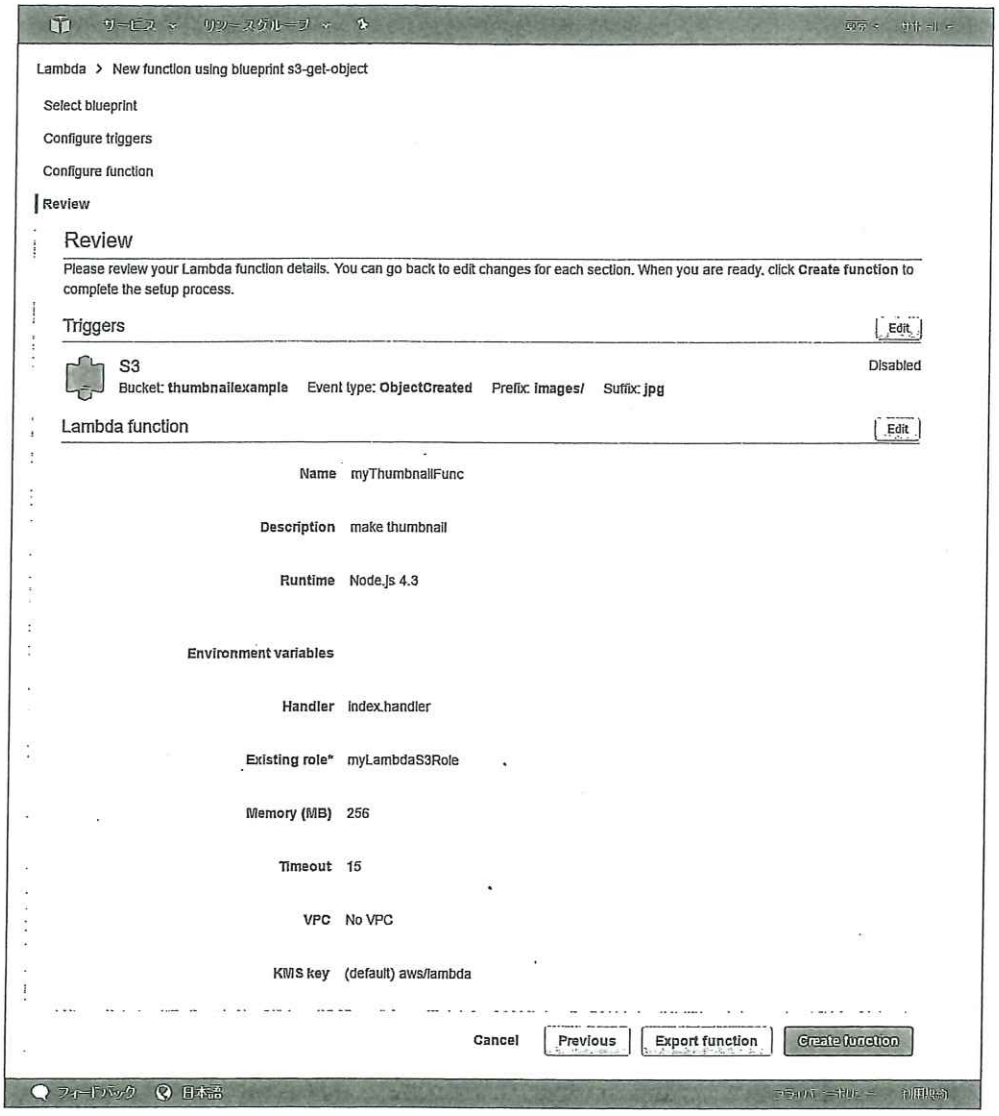

図 3-3-18 Lambda 関数を作成する

メモ 》何かコードを変更すると［Save］ボタンが表示されます。変更していないときは［Test］ボタンしかありません。

List 3-3-1 サムネイルを作成するための Lambda 関数

```
'use strict';
console.log('Loading function');
const aws = require('aws-sdk');
```

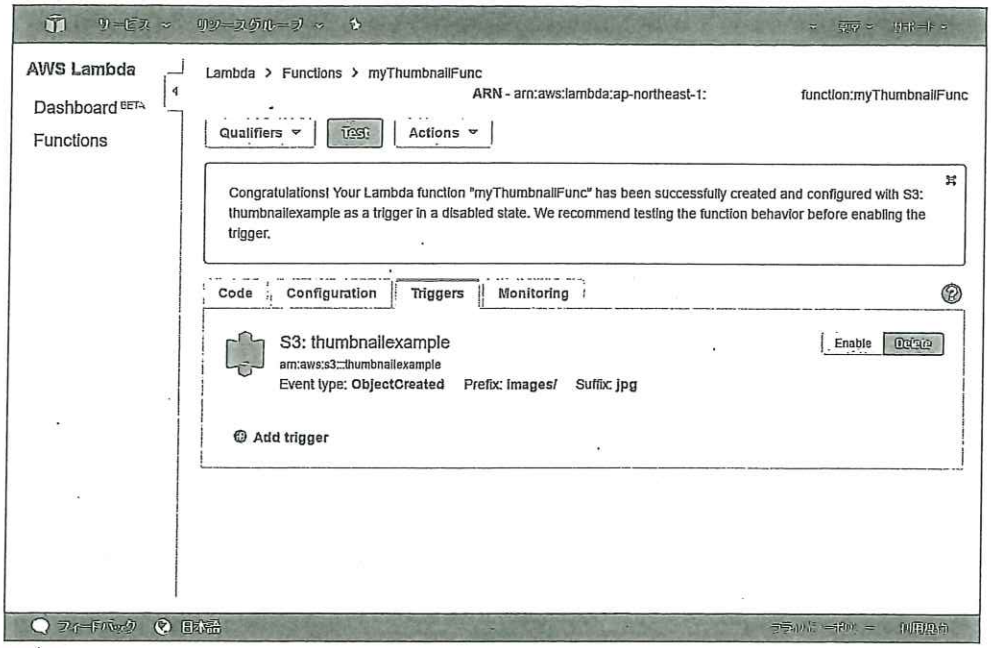

図 3-3-19 作成された Lambda 関数

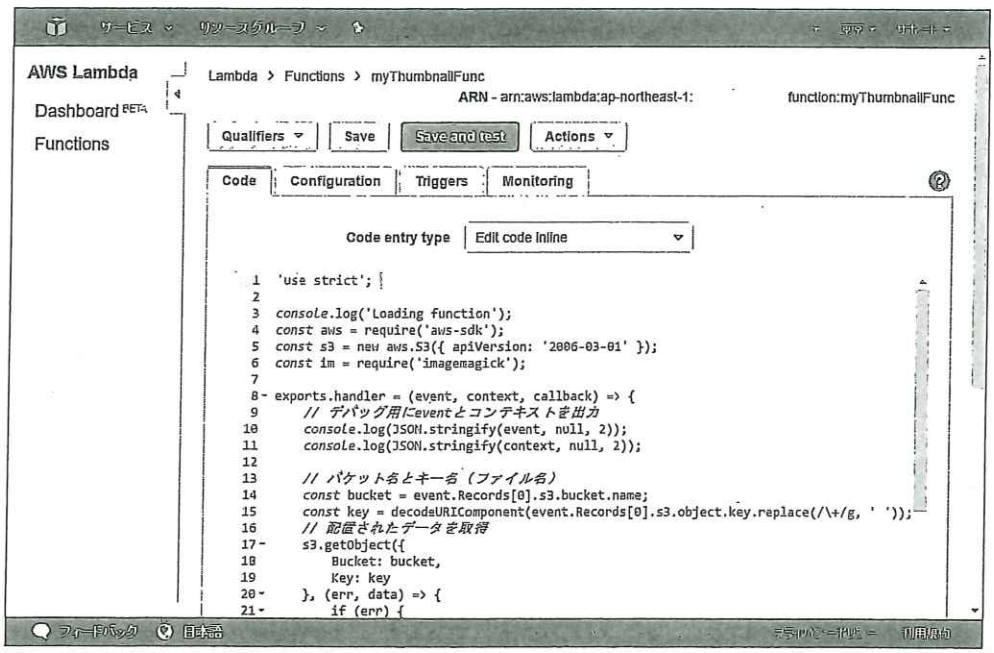

図 3-3-20 サムネイルを作るためのコードを記述する

```
const s3 = new aws.S3({ apiVersion: '2006-03-01' });
const im = require('imagemagick');

exports.handler = (event, context, callback) => {
    // デバッグ用にeventとコンテキストを出力
    console.log(JSON.stringify(event, null, 2));
    console.log(JSON.stringify(context, null, 2));

    // バケット名とキー名（ファイル名）
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/¥+/g, ' '));
    // 配置されたデータを取得
    s3.getObject({
        Bucket: bucket,
        Key: key
    }, (err, data) => {
        if (err) {
            console.log(err);
            callback(`Cannot read ${key} from bucket ${bucket}`);
        } else {
            // 幅320pxのサムネイルを作成
            im.resize({
                srcData: data.Body,
                format: "jpeg",
                width: 320
            }, function(err, stdout, stderr) {
                if (err) {
                    console.log(err);
                    callback('Cannot create thumbnail.');
                } else {
                    // 作成したサムネイルを保存
                    var thumbnailkey = key.replace("images/", "thumbnails/");
                    s3.putObject({
                        Bucket: bucket,
                        Key: thumbnailkey,
                        Body: new Buffer(stdout, "binary"),
                        ContentType : data.ContentType
                    }, function(err, res) {
                        if (err) {
                            callback(`Cannot write {$thumbnailkey}`);
                        } else {
                            // 成功
                            callback(null);
                        }
                    });
                }
            });
        }
    });
};
```

プログラムの冒頭では、利用するライブラリを読み込んでいます。ここではawsのライブラリ、S3のライブラリ、そして、画像を縮小するためのImageMagickライブラリを読み込みました。

メモ ImageMagickは、Lambda環境に標準で組み込まれているため、このようにして読み込めます。標準外のライブラリを使いたいときは、この画面でソースコードを記述する方法ではなく、ソースコードとライブラリをZIP形式でまとめて、それをアップロードするという方法でLambda関数を作る必要があります。特定のバージョンのImageMagickを利用したい場合は、あえて利用したいバージョンのImageMagickを自分でアップロードしても構いません。

```
const aws = require('aws-sdk');
const s3 = new aws.S3({ apiVersion: '2006-03-01' });
const im = require('imagemagick');
```

なお、ソースコードの先頭にある、

```
console.log('Loading function');
```

というのは、コンソールに出力したいメッセージです。console.logで出力した内容は、CloudWatch Logs（あとで説明します）で参照できるので、デバッグ時に役立ちます。

Lambda関数の本体は、次のように指定したhandlerメソッドです。イベントが発生したときには、このメソッドが呼び出されます。

```
exports.handler = (event, context, callback) => {
}
```

event引数はイベントに関するデータが格納されています。データの内容は、トリガとなったイベントの種類によって異なります。例えばS3ならバケット名などが含まれていますし、SESならメールアドレスなどが含まれているという具合です。

context引数はイベントのコンテキスト情報です。そしてcallback引数は結果を返すときに使うメソッドです。

S3バケットに対する書き込みイベントからLambda関数が呼び出された場合、書き込まれたデータのバケット名とキー名（ファイル名）は、event引数から、次のようにして取得できます。

```
const bucket = event.Records[0].s3.bucket.name;
const key =
  decodeURIComponent(
      event.Records[0].s3.object.key.replace(/¥+/g, ' '));
```

このバケットからデータを取得するには、getObjectメソッドを呼び出します。

```
s3.getObject({
    Bucket: bucket,
    Key: key
}, (err, data) => {
…処理…
}
```

失敗した場合は、err にエラーメッセージが設定されます。エラーが発生したら、callback メソッドに「失敗した旨のメッセージを返す」という処理をします。これで Lambda 関数は失敗したという意味になります。

```
if (err) {
    console.log(err);
    callback(`Cannot read ${key} from bucket ${bucket}`);
}
```

成功したら、サムネイルを作ります。ここでは ImageMagick ライブラリを使って生成しました。幅は 320px としました。

```
im.resize({
  srcData: data.Body,
  format: "jpeg",
  width: 320
}, function(err, stdout, stderr) {
…処理…
}
```

生成したサムネイルは、putObject メソッドを呼び出すと S3 バケットに書き込めます。

```
var thumbnailkey = key.replace("images/", "thumbnails/");
s3.putObject({
    Bucket: bucket,
    Key: thumbnailkey,
    Body: new Buffer(stdout, "binary"),
    ContentType : data.ContentType
}, function(err, res) {
…処理…
}
```

処理に成功した場合は、callback メソッドに null を渡してください。

```
callback(null);
```

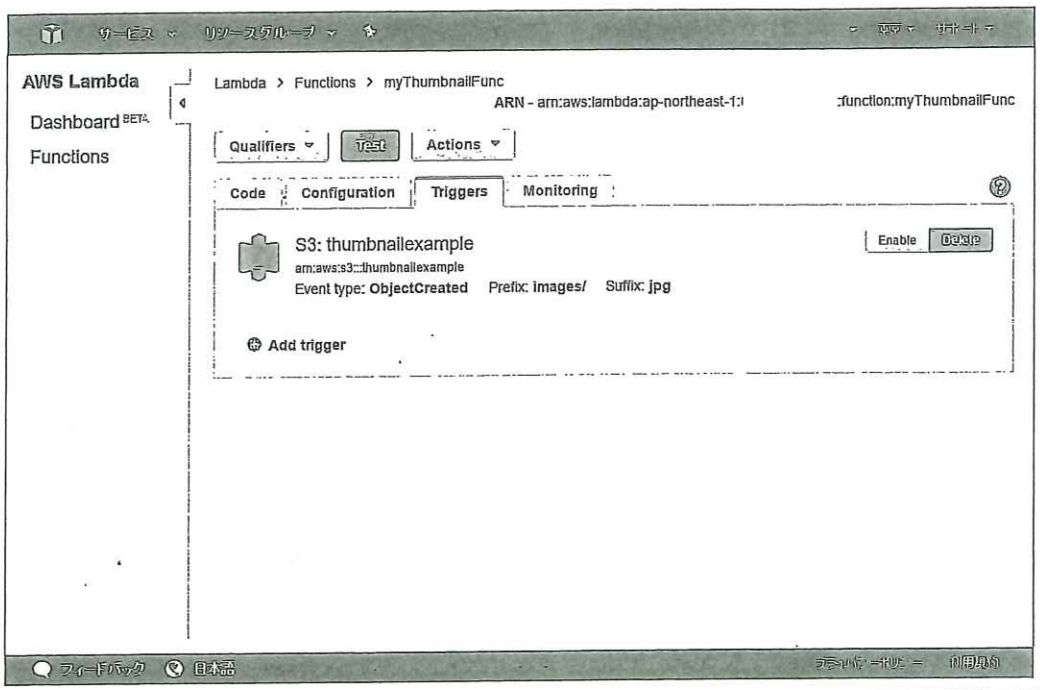

図 3-3-21 トリガを有効にする

【トリガを有効にしてテストする】

では、実際にテストしてみましょう。まずは、S3 のトリガを有効にします。[Triggers] タブを開き、[Enable] をクリックして、トリガを有効にしてください（図 3-3-21）。確認メッセージが表示されたら [Enable] をクリックしてください（図 3-3-22）。

メモ ここでは Lambda 関数のテストをせずに、いきなりトリガを有効にしていますが、本来なら、Lambda 関数のテスト機能を使って、一通りの動作テストをしてから、トリガを有効にすべきです。しかし画像のサムネイルを作る処理を Lambda 関数のテスト機能で実行するのは難しいので、ここではテストを省略しました。Lambda 関数のテストの方法については「●コンテンツが役立ったかどうかをアンケートを採りたい」で説明します。

この状態で、先の S3 バケットの images ディレクトリーに、適当な JPEG 画像をアップロードしてください（図 3-3-23）。すると、thumbnails ディレクトリーに、同名のサムネイル画像が保存されるはずです。

実際に Lambda 関数が実行されたかどうかは、[Monitoring] タブで確認できます（図 3-3-24）。

[View logs in CloudWatch] をクリックすると、CloudWatch Logs でログを確認できます（図 3-3-25）。ログストリームの各行をクリックすると、その詳細が表示されます。この詳細には、Console.log メソッドで書き出したメッセージも含まれています（図 3-3-26）。

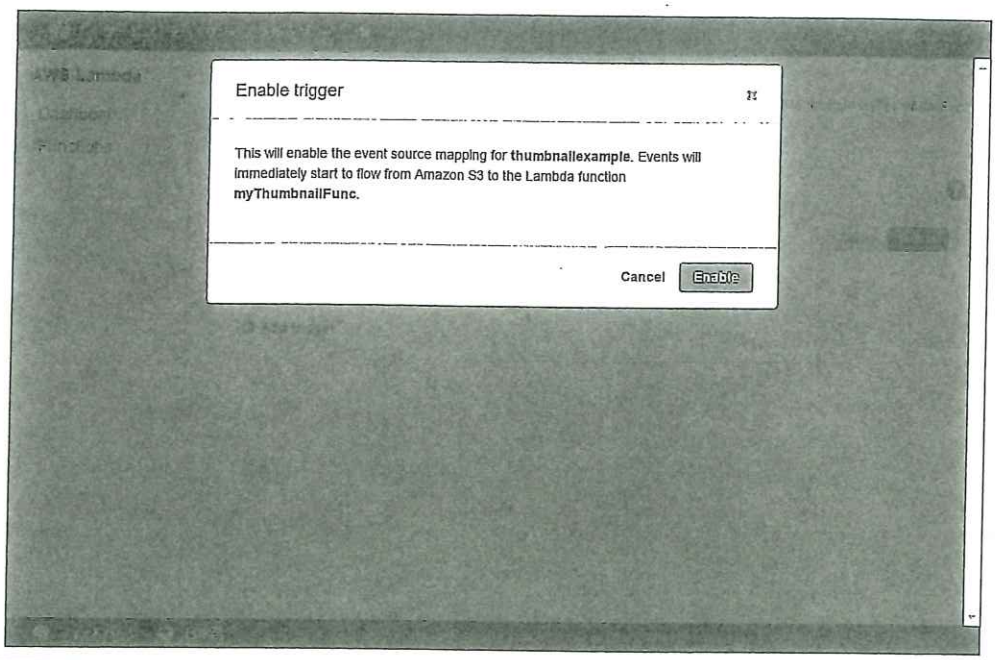

図 3-3-22 確認メッセージ

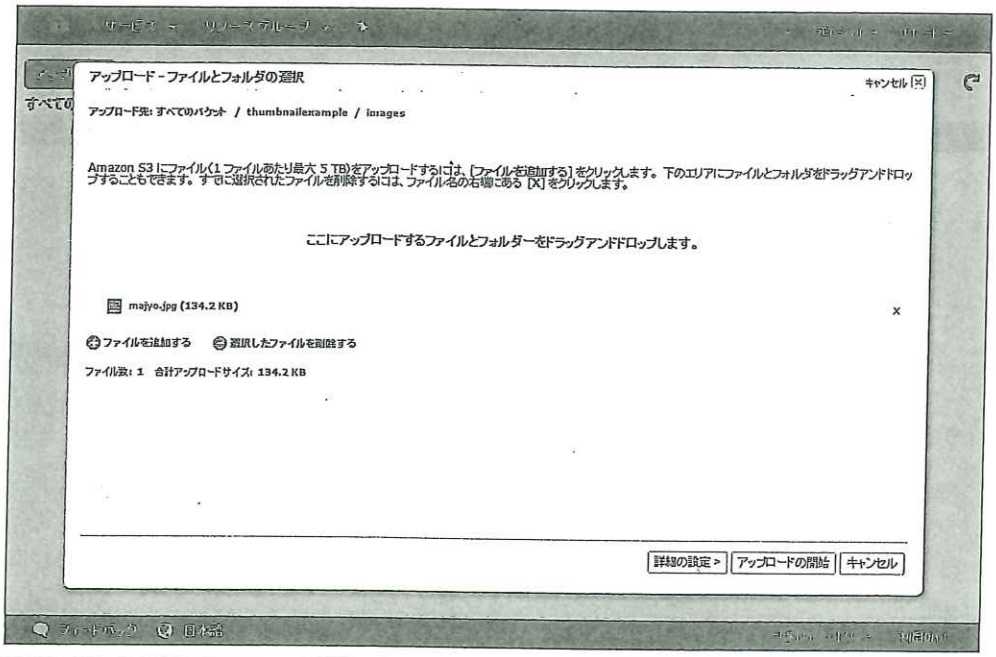

図 3-3-23 JPEG 画像をアップロードする

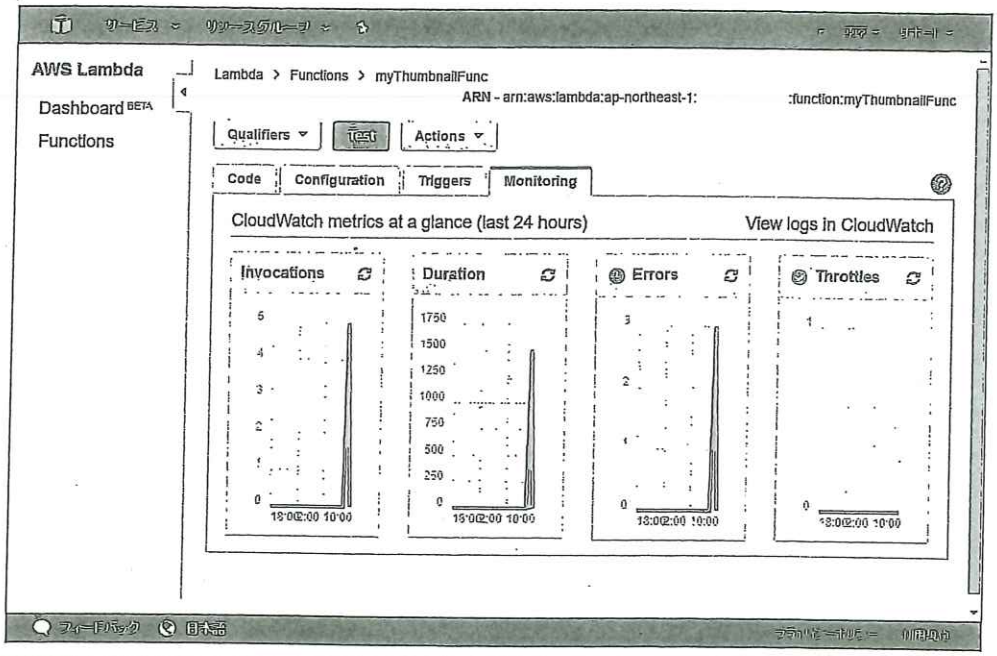

図 3-3-24 [Monitoring] タブで確認する

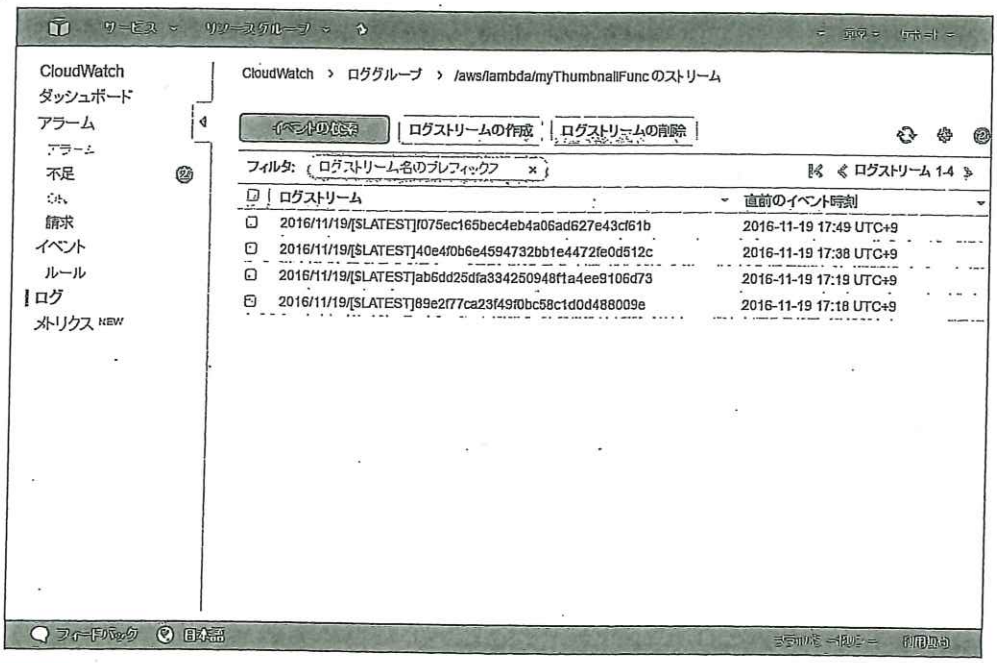

図 3-3-25 CloudWatch Logs で確認する