AWS CDK 入門 (3) CDK サンプルコード

AWS AWS/CDK

更新: 2021-10-21
作成: 2021-10-04

AWS CDK を使った TypeScript サンプルコードいろいろです。

リソースにタグを付ける

```
import * as cdk from '@aws-cdk/core'
import { MyappStack } from '../lib/myapp-stack'
```

```
const app = new cdk.App()
new MyappStack(app, 'MyappStack', {
  tags: {
    Owner: 'TeamA',
    Purpose: 'Project1',
  },
})
```

AWS リソース用のコンストラクトの props パラメーターで、`tags` プロパティを指定することで、そのリソースにタグを設定できます。

タグの設定方法は、どの AWS リソース用のコンストラクトでも同様です。 上記のように `Stack` コンストラクトに対してタグを設定すると、その中に配置した AWS リソースにもそのタグが設定されます。

S3 バケットや DynamoDB テーブルをスタック削除時に自動削除する

バケットが空のときだけ自動削除する

```
import * as cdk from '@aws-cdk/core'
import * as s3 from '@aws-cdk/aws-s3'

new s3.Bucket(this, 'MyBucket', {
  removalPolicy: cdk.RemovalPolicy.DESTROY,
})
```

CDK で作成した S3 バケットや DynamoDB テーブルは、デフォルトでは、（内容が空であっても）スタック削除時にそのまま残るようになっています。 つまり、スタックから独立したリソースとして S3 バケットだけが残ります。 これは、CloudFormation の DeletionPolicy のデフォルトの挙動とは逆なので注意してください。 スタックの削除 (`cdk destroy`) と同時に S3 バケットや DynamoDB テーブルを削除したいときは、上記のように `removalPolicy` を設定します。

バケットが空じゃなくても自動削除する

```
import * as cdk from '@aws-cdk/core'
import * as s3 from '@aws-cdk/aws-s3'

new s3.Bucket(this, 'MyBucket', {
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true,
})
```

`removalPolicy` を `RemovalPolicy.DESTROY` に設定しても、S3 バケットにオブジェクト含まれているときは、スタック削除時に連動して自動削除してくれません。 スタック削除時に、バケット内のオブジェクトを自動削除してバケットの削除までやってしまいたいときは、`removalPolicy` に加えて、**`autoDeleteObjects`** プロパティを設定します。 この設定を行うと、たとえ S3 バケットのバージョニングが有効 (`versioned: true`) になっていても、問答無用で削除されるので注意してください。 `autoDeleteObjects` の機能を実現するために、内部的に Lambda 関数が自動生成されるので、CDK のブートストラッピング (`cdk bootstrap`) をあらかじめ実行しておく必要があります。

S3 バケットの物理名 (Physical ID) を指定する

```
import * as s3 from '@aws-cdk/aws-s3'

const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'bucket-123456789012-user-data',
})
```

L2 コンストラクトの **`s3.Bucket`** は、物理バケット名（物理 ID）を自動生成してくれますが、何らかの理由で固定の物理名を指定したいときは、上記のように **`bucketName`** プロパティで明示的に指定することができます。 逆に、L1 コンストラクトの **`CfnBucket`** を使ってバケットを作成するときは、必ず `bucketName` の指定が必要です。

S3 バケットを作成して Lambda 関数から参照できるようにする

```
import * as lambda from '@aws-cdk/aws-lambda'
import * as s3 from '@aws-cdk/aws-s3'
```

```
const bucket = new s3.Bucket(this, 'my-bucket')
new lambda.Function(this, 'my-lambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
})

// バケットポリシーで Lambda 関数から読み書きできるようにする
bucket.grantReadWrite(lambda)
```

`Function` コンストラクトをインスタンス化するときに、上記のように **`environment`** props を指定することで、Lambda 関数の実装の中で、環境変数 `BUCKET_NAME` としてバケット名を参照できるようになります。 実際に Lambda 関数から S3 バケットにアクセスできるようにするには、`Bucket` コンストラクト側の `grantRead` / `grantWrite` / `grantReadWrite` 関数を呼び出して、Lambda 関数にアクセス権限を付けておく必要があります（バケットポリシーが生成されます）。 これを忘れると、Lambda 関数の実行時に Access Denied エラーになります。

既存の S3 バケットを参照する

```
import * as s3 from '@aws-cdk/aws-s3'

// (A) Construct a resource (bucket) just by its name (must be same acc
const myBucket = s3.Bucket.fromBucketName(
  this, 'MyBucket', 'bucket-123456789012-user-data'
)

// (B) Construct a resource (bucket) by its full ARN (can be cross accou
const myBucket = s3.Bucket.fromBucketArn(
  this, 'MyBucket', 'arn:aws:s3:::bucket-123456789012-user-data'
)

// あとは Lambda 関数などに読み書き権限を与える
myBucket.grantReadWrite(lambdaFunc)
```

すでに別の CloudFormation スタック内に作成済みの S3 バケットなどを参照する必要がある場合は、上記のようにバケット名や ARN をもとに `s3.Bucket` インスタンスを生成できます。

既存の DynamoDB テーブルを参照する

```
import * as dynamodb from '@aws-cdk/aws-dynamodb'

// (A) 既存の DynamoDB テーブルをテーブル名で参照する（同一アカウント内）
const configTable = dynamodb.Table.fromTableName(
  this, 'ConfigTable', 'myapp-dev-config'
)

// (B) 既存の DynamoDB テーブルを ARN で参照する（クロスアカウント）
const configTable = dynamodb.Table.fromTableArn(
  this,
  'ConfigTable',
  'arn:aws:dynamodb:<Region>:<Account>:table/myapp-dev-config'
)

// あとは Lambda 関数などに読み書き権限を与える
configTable.grantReadWriteData(lambdaFunc)
```

S3 バケットにオブジェクトを追加したときに SNS トピック通知を発行する

```
import * as s3 from '@aws-cdk/aws-s3'
import * as s3notify from '@aws-cdk/aws-s3-notifications'
import * as sns from '@aws-cdk/aws-sns'

const bucket = new s3.Bucket(this, 'bucket')
const topic = new sns.Topic(this, 'topic')
bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic))
```

上記のように S3 バケットと SNS トピックを生成して、`addObjectCreatedNotification` で関連付けると、S3 バケットへのオブジェクト追加（および更新）時に SNS トピックからの通知を発行できるようになります。 CloudFormation を直書きすると、S3 との連携用にポリシー設定（`AWS::SNS::TopicPolicy`）まで記述しないといけなくて非常に面倒ですが、CDK であれば、上記のように記述だけで済みます（トピックポリシーは内部で自動生成してくれます）。

SNS Topic に自動設定されるアクセスポリシーの例

応用として、`NotifyingBucket` をインスタンス化するときの props 引数で、`prefix: '/images'` と指定すれば、監視対象となるオブジェクトをフィルタすることができます。 次の

独自コンストラクト (`NotifyingBucket`) の実装例では、オプションで `prefix` を指定できるようにしています。

```
import * as cdk from '@aws-cdk/core'
import * as s3 from '@aws-cdk/aws-s3'
import * as s3notify from '@aws-cdk/aws-s3-notifications'
import * as sns from '@aws-cdk/aws-sns'

export interface NotifyingBucketProps {
  prefix?: string
}

export class NotifyingBucket extends cdk.Construct {
  constructor(scope: cdk.Construct, id: string, props: NotifyingBucketPr
    super(scope, id)
    const bucket = new s3.Bucket(this, 'bucket')
    const topic = new sns.Topic(this, 'topic')
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topi
      prefix: props.prefix
    })
  }
}
```

- 参考: Constructs - AWS Cloud Development Kit (CDK)

S3 バケットにオブジェクトが追加されたときに Lambda 関数を呼び出す

```
import * as lambda from '@aws-cdk/aws-lambda'
import * as s3 from '@aws-cdk/aws-s3'
import * as s3notify from '@aws-cdk/aws-s3-notifications'

const handler = new lambda.Function(this, 'Handler', {/* ... */})
const bucket = new s3.Bucket(this, 'Bucket')
bucket.addObjectCreatedNotification(new s3notify.LambdaDestination(handl
```

S3 バケットにオブジェクトが追加されたときに Lambda 関数を呼び出すには、`Bucket#addObjectCreatedNotification()` に `LambdaDestination` オブジェクトを渡します。 呼び出す関数は SNS トピック通知を発行する場合と同様ですが、引数で渡すオブジェクトが異なります。

SNS トピックの通知で Lambda 関数を呼び出す

```
import * as sns from '@aws-cdk/aws-sns'
import * as snsSub from '@aws-cdk/aws-sns-subscriptions'

// 既存の SNS Topic からの通知で Lambda 関数を起動する
const myTopic = sns.Topic.fromTopicArn(
  this,
  'MyTopic',
  'arn:aws:sns:<Region>:<Account>:myapp-dev-xxx-topic'
)
myTopic.addSubscription(new snsSub.LambdaSubscription(handler))
```

S3 バケットを Read 可能な IAM グループを作成する

```
import * as s3 from '@aws-cdk/aws-s3'
import * as iam from '@aws-cdk/aws-iam'

const rawData = new s3.Bucket(this, 'raw-data')
const dataScience = new iam.Group(this, 'data-science')
rawData.grantRead(dataScience)
```

S3 バケットへの CORS アクセスを許可する

```
import * as s3 from '@aws-cdk/aws-s3'

const myBucket = new s3.Bucket(this, 'MyBucket', {
  cors: [
    {
      allowedHeaders: ['*'],
      allowedMethods: [s3.HttpMethods.GET, s3.HttpMethods.POST],
      allowedOrigins: [
        'http://localhost:*',
        'https://example.com/',
        'https://*.example.com/',
      ],
    },
  ],
})
```

Web サイトのクライアントサイド JavaScript から、S3 バケット内のファイルを取得する場合は、S3 バケットの CORS 設定でクロスオリジンのアクセスを許可しておく必要があります。 Web ブラウザのコンソール出力で、次のようなエラーが出たら、この CORS 設定ができていない証拠です。

> *Access to fetch at '…' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.*

`allowedOrigins` の指定方法ですが、`http://localhost:3000` からアクセスするのであれば、`http://localhost:3000` や `http://localhost:*` と記述する必要があり、スキーマを省略したり (`localhost:3000`)、ポート番号を省略したり (`http://localhost`) するのは NG です。 ワイルドカードのアスタリスクは、アドレスの一か所でのみ使用可能です。

- 参考: **CORS configuration - Amazon S3**

SQS キューと Lambda 関数を結びつける

```
import * as lambda from '@aws-cdk/aws-lambda'
import * as sqs from '@aws-cdk/aws-sqs'

const jobsQueue = new sqs.Queue(this, 'jobs')
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_14_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
})
```

このように、`lambda.Function` の props で `environment` を設定しておくと、Lambda 関数の実装の中から、`QUEUE_URL` 環境変数の形で SQS キューの URL を参照できるようになります。

- 参考: **Constructs - AWS Cloud Development Kit (CDK)**

ECS クラスターを作って EC2 サービスから参照する

```
import * as ecs from '@aws-cdk/aws-ecs'

const cluster = new ecs.Cluster(this, 'Cluster', {/* ... */})
const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster }
```

別のスタック内の S3 バケットを参照する

```
const prod = { account: '123456789012', region: 'ap-northeast-1' }
const stack1 = new Stack1(app, 'Stack1', { env: prod })
const stack2 = new Stack2(app, 'Stack2', { env: prod, bucket: stack1.buc
```

`Stack2` のコンストラクタの `bucket` props として、`Stack1` オブジェクトの公開プロパティ `bukcket` を渡しています。同じアカウント＆リージョンのスタックであることが条件です。

EventBridge で定期的に Lambda 関数を実行する

```
import * as cdk from '@aws-cdk/core'
import * as events from '@aws-cdk/aws-events'
import * as eventsTargets from '@aws-cdk/aws-events-targets'
import * as lambdaNodejs from '@aws-cdk/aws-lambda-nodejs'

// Lambda 関数を作成する
const myLambda = new lambdaNodejs.NodejsFunction(this, 'MyLambda', {
  entry: 'lambda/index.ts',
})

// EventBridge ルールで 10 分おきに Lambda 関数呼び出し
new events.Rule(this, 'MyRule', {
  schedule: events.Schedule.rate(cdk.Duration.minutes(10)), // 10分おき
  // schedule: events.Schedule.cron({ ... }), // cron 形式で指定する場合
  targets: [
    new eventsTargets.LambdaFunction(myLambda, { retryAttempts: 3 }),
  ],
})
```

上記の例では、10 分ごとに Lambda 関数を呼び出すように EventBridge のスケジュール設定（ルール設定）を行っています。

- 参考: ルールのスケジュール式 - Amazon CloudWatch Events

CDK コードの中で SSM パラメーターストアのパラメーター値を取得する

```
import * as ssm from '@aws-cdk/aws-ssm'

const bucketName = ssm.StringParameter.valueForStringParameter(this,
  '/myapp/dev/ImageBucketName'
)
```

上記の例では、SSM パラメーターストアの `/myapp/dev/ImageBucketName` というパラメーターに格納された値を取得しています。 他のアプリが生成した S3 バケットの名前をこのパラメーターストアに格納してくれていれば、その値を介して連携させることができます。 パラメーターの種類が SecureString の場合は、次のように別の関数を使います（バージョン情報の指定が必要です）。

```
const gitHubToken = ssm.StringParameter.valueForSecureStringParameter(th
  '/myapp/dev/GitHubToken', 1
)
```

Lambda 関数から SSM パラメーターストアにアクセスできるようにする

```
import * as lambdaNodejs from '@aws-cdk/aws-lambda-nodejs'
import * as ssm from '@aws-cdk/aws-ssm'

// Lambda 関数を作成する
const myLambda = new lambdaNodejs.NodejsFunction(this, 'MyLambda', {
  runtime: lambda.Runtime.NODEJS_14_X,
  entry: 'lambda/index.ts',
  environment: {
    GITHUB_TOKEN_SSM_PARAM: '/myapp/dev/GitHubToken',
  },
})

// 既存の SSM パラメーターの読み取り権限を Lambda 関数に与える
const mySsmParam = ssm.StringParameter.fromStringParameterName(
```

```
  this, 'MySsmParam', '/myapp/dev/GitHubToken'
)
mySsmParam.grantRead(myLambda)

// エラーが出るときは、下記でやってみる
// const mySsmParam = ssm.StringParameter.fromSecureStringParameterAttr
//   this, 'MySsmParam', {
//     parameterName: '/myapp/dev/GitHubToken',
//     version: 1,
//   }
// )
```

ここでは、パラメーターストア上のパラメーター (`/myapp/dev/GitHubToken`) として GitHub のアクセストークンを格納して、Lambda 関数内からその値を取得することを想定しています。既存のパラメーターのコンストラクト参照を取得するには、上記のように `ssm.StringParameter.fromStringParameterName` 関数を使用します。 あとは、`grantRead` で Lambda 関数からの読み込みを許可してやると、次のようなアクションがまとめて許可されます。

- `ssm:DescribeParameters`
- `ssm:GetParameter`
- `ssm:GetParameterHistory`
- `ssm:GetParameters`

`StringParameter` の `grantRead` 関数を使わずに、次のように Lambda 関数に Policy ステートメントを直接追加してしまう方法もありますが、パラメーターの ARN 指定などが面倒ですね。`grantRead` を使った方がシンプルでよいと思います。

```
myLambda.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['ssm:GetParameter*'],
  resources: ['arn:aws:ssm:<Region>:<Account>:parameter/myapp/dev/GitHub
}))
```

ちなみに、Lambda 関数 (AWS SDK) の方では、次のような感じでパラメーターの値を取得できます。

Lambda 関数内で SSM パラメーター値を取得

```
import * as AWS from 'aws-sdk'
const ssm = new AWS.SSM({ region: 'ap-northeast-1' })

export async function getGitHubToken(): Promise<string | undefined> {
  const result = await ssm.getParameter({
    Name: process.env.GITHUB_TOKEN_SSM_PARAM as string,
    WithDecryption: true,
  }).promise()
  return result.Parameter?.Value
}
```

ApiGateway + Lambda 関数で REST API を作成する

lib/myapi-stack.ts

```
import * as cdk from '@aws-cdk/core'
import * as apigateway from '@aws-cdk/aws-apigateway'
import * as lambdaNodejs from '@aws-cdk/aws-lambda-nodejs'

export class MyapiStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps)
    super(scope, id, props)

    // Lambda 関数（GET books/ のハンドル用）
    const getBooksHandler = new lambdaNodejs.NodejsFunction(
      this, 'getBooksHandler', {
        entry: 'lambda/index.ts',
        handler: 'getBooksHandler',
      })

    // Lambda 関数（GET books/{id} のハンドル用）
    const getSingleBookHandler = new lambdaNodejs.NodejsFunction(
      this, 'getSingleBookHandler', {
        entry: 'lambda/index.ts',
        handler: 'getSingleBookHandler',
      })

    // ApiGateway (RestApi) の作成
    const api = new apigateway.RestApi(this, 'api')

    // Lambda 関数を結びつける (GET books/)
    const books = api.root.addResource('books')
    books.addMethod('GET',
      new apigateway.LambdaIntegration(getBooksHandler))

    // Lambda 関数を結びつける (GET books/{id})
    const singleBook = books.addResource('{id}')
    singleBook.addMethod('GET',
```

```
        new apigateway.LambdaIntegration(getSingleBookHandler))
    }
}
```

下記はバックエンドとして動く Lambda 関数の適当な実装です。 実際には、DynamoDB などから情報を取得して JSON データとして返すように実装します。

lambda/index.ts

```
import { Handler } from 'aws-lambda'

const BOOKS = [
  { id: '1', title: 'Title 1' },
  { id: '2', title: 'Title 2' },
  { id: '3', title: 'Title 3' },
]

/** 全ての本情報を取得する。 */
export const getBooksHandler: Handler = async () => {
  return {
    statusCode: 200,
    body: JSON.stringify(BOOKS),
  }
}

/** 指定された ID の本情報を取得する。 */
export const getSingleBookHandler: Handler = async (event: any = {}) =>
  const id = event.pathParameters.id
  return {
    statusCode: 200,
    body: JSON.stringify(BOOKS.find((b) => b.id === id)),
  }
}
```

関連記事

- AWS CDK でナゾの CDKMetadata を生成しないようにする
- AWS CDK 入門 (1) インストールから Hello World まで
- AWS CDK 入門 (2) コンストラクトの概念を理解する
- DynamoDB 用のポリシー設定例
- Amazon S3: 未整理・雑多メモ
- DynamoDB の未整理・雑多メモ

- Amazon Cognito: Amplify SDK が提供する UI コンポーネントあれこれ