AWS CDK レイヤーガイド

AWS 規範ガイダンス

AWS 規範ガイダンス: AWS CDK レイヤーガイド

Table of Contents

AWS CDK レイヤーガイド

Amazon Web Services (AWS)、Steven Guggenheimer

2023 年 12 月 (ドキュメント履歴)

AWS Cloud Development Kit (AWS CDK) を支える主な概念の 1 つは、寒い日に暖かく過ごすための考え方によく似ています。この考え方をレイヤリングといいます。寒い日には、シャツの上にジャケットを着、寒さによってはさらに厚めのジャケットを羽織ることもあるでしょう。その状態で室内に入ったときに、暖房が効いている場合は、暑くならないように、一方または両方のレイヤー、つまりジャケットを脱ぐことができます。AWS CDK では、レイヤリングによって、クラウドコンポーネントを使用するためのさまざまな抽象化レベルを実現しています。レイヤリングの仕組みがあれば、Infrastructure as Code (IAC) スタックのデプロイ時に、必要以上に多くのコードを記述したり、アクセス権が少なすぎてリソースプロパティにアクセスできなかったりすることがなくなります。

AWS CDK を使用しない場合は、手動で AWS CloudFormation テンプレートを記述する必要があります。つまり、1 つのレイヤーしか活用できず、通常必要な数よりもはるかに多くのコードを記述せざるを得なくなります。一方、CloudFormation では通常、記述が不要なものの、それらが AWS CDK によってすべて抽象化されると、エッジケースに対応できなくなります。

この問題に対処するために、AWS CDK では、リソースのプロビジョニングを 3 つの異なるレイヤーに分割します。

- レイヤー 1 – CloudFormation レイヤー: 最も基本的なレイヤーであり、CloudFormation と AWS CDK のリソースは、ほぼ同じです。
- レイヤー 2 – キュレートされたレイヤー: CloudFormation リソースがプログラムクラスに抽象化されるとともに、内部にある CloudFormation ボイラープレート構文の多くが合理化されるレイヤーであり、AWS CDK のほぼ全体を占めています。
- レイヤー 3 – パターンレイヤー: 最も抽象化されたレイヤーであり、ここでは、レイヤー 1 と 2 が備える構成要素を使用して、特定のユースケースに合うようにコードをカスタマイズできます。

各レイヤーの各項目は、特別な AWS CDK クラスのインスタンスであり、これを `Construct` と呼びます。AWS ドキュメント では、こう説明しています。「コンストラクトは、AWS CDK アプリケーションの基本的な構成要素で、クラウドコンポーネントを表すものです。これによって、AWS CloudFormation でのコンポーネント作成に必要なあらゆる要素をカプセル化します」。こうしたレイヤー内のコンストラクトを、それらが属するレイヤーに応じて L1、L2、L3 コンストラクトと呼び

ます。このガイドでは、各 AWS CDK レイヤーを解説しながら、その用途や、レイヤーが重要である理由を明らかにしていきます。

このガイドは、AWS CDK が動作するための主な概念を深く掘り下げたい技術マネージャー、リーダー、開発者を対象にしています。AWS CDK は、広く利用されているツールですが、それが持つ機能の大部分が見落とされがちです。このガイドが示す概念を理解し始めると、新たな可能性の世界を導き出し、チームのリソースプロビジョニングプロセスを最適化できるでしょう。

このガイドの内容

レイヤー 1 コンストラクト

L1 コンストラクトは AWS CDK の構成要素であり、Cfn というプレフィックスが付くため、他のコンストラクトと簡単に区別できます。例えば、AWS CDK の Amazon DynamoDB パッケージには、L2 コンストラクトである Table コンストラクトが含まれています。対応する L1 コンストラクトを CfnTable と呼び、これによって、CloudFormation DynamoDB Table を直接表します。通常、AWS CDK アプリケーションで L1 コンストラクトを直接使用することはありませんが、この最初のレイヤーにアクセスせずに AWS CDK を使用することは不可能です。ただし、ほとんどの場合、開発者が使い慣れている L2 および L3 コンストラクトは、L1 コンストラクトに大きく依存しているため、L1 コンストラクトは CloudFormation と AWS CDK 間の橋渡しをするものと考えると良いでしょう。

AWS CDK は、標準のコーディング言語を使用して CloudFormation テンプレートを生成することに特化しています。つまり、cdk synth CLI コマンドを実行し、CloudFormation テンプレートが生成されると、AWS CDK は仕事を終えます。利便性に配慮して cdk deploy コマンドが用意されていますが、これを実行すると、その処理は CloudFormation 内ですべて実行されます。CloudFormation で理解可能な形式に AWS CDK コードを変換する際に重要な役割を果たすのが、L1 コンストラクトなのです。

AWS CDK と CloudFormation 間における、L1 コンストラクトのライフサイクル

L1 コンストラクトを作成して使用するプロセスは、以下のステップで構成されます。

1. AWS CDK のビルドプロセスにより、CloudFormation の仕様が L1 コンストラクトの形式でプログラムコードに変換されます。
2. 開発者が、AWS CDK アプリケーションの一部として L1 コンストラクトを直接または間接的に参照するコードを記述します。
3. 開発者が、cdk synth コマンドを実行して、プログラムコードを CloudFormation 仕様 (テンプレート) で指定された形式に変換します。
4. 開発者が、cdk deploy コマンドを実行して、こうしたテンプレート内の CloudFormation スタックを AWS アカウント環境にデプロイします。

簡単な演習を行いましょう。GitHub の AWS CDK オープンソースリポジトリにアクセスして、AWS サービスの 1 つをランダムに選択し、そのサービスの AWS CDK パッケージに移動します (次のよ

うにフォルダをたどります: packages、aws-cdk-lib、aws-<servicename>、lib)。この例では Amazon S3 を選択しますが、どのサービスを使用しても動作します。そのパッケージのメイン index.ts ファイルを確認すると、次の行があります。

```
export * from './s3.generated';
```

ただし、対応するディレクトリのどこにも、s3.generated ファイルはありません。なぜなら、L1 コンストラクトは、AWS CDK ビルドプロセス中に CloudFormation リソース仕様に基づいて自動生成されるからです。そのため、対象のパッケージに AWS CDK ビルドコマンドを実行した後にのみ、パッケージ内に s3.generated が存在することになります。

AWS CloudFormation リソース仕様

AWS CloudFormation リソース仕様は、AWS の Infrastructure as Code (IAC) を定義するとともに、この仕様によって、CloudFormation テンプレート内のコードをどのように AWS アカウントのリソースに変換するかを決定するものです。この仕様には、リージョンごとのレベルで、JSON 形式の AWS リソースが定義されています。各リソースには、provider::service::resource の形式に従い、一意のリソースタイプ名が付与されます。例えば、Amazon S3 バケットのリソースタイプには AWS::S3::Bucket という名前付き、Amazon S3 アクセスポイントのリソースタイプには AWS::S3::AccessPoint という名前が付きます。こうしたリソースタイプは、AWS CloudFormation リソース仕様で定義されている構文を使用して CloudFormation テンプレート内にレンダリングすることもできます。AWS CDK ビルドプロセスを実行すると、各リソースタイプも L1 コンストラクトに変換されます。

したがって、各 L1 コンストラクトは、対応する CloudFormation リソースがプログラムによってそのまま反映されたものです。CloudFormation テンプレートに適用するどのプロパティも、L1 コンストラクトのインスタンス化に使用でき、対応する L1 コンストラクトをインスタンス化する際にも、必要なすべての CloudFormation プロパティを引数に指定しなければなりません。次の表は、CloudFormation テンプレートに表された S3 バケットと、AWS CDK L1 コンストラクトとして定義した同じ S3 バケットを比較したものです。

CloudFormation テンプレート

```
"amzns3demobucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": "amzn-s3-demo-
bucket",
```

L1 コンストラクト

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
```

```
      "BucketEncryption": {
        "ServerSideEncryptionConfig
uration": [
          {
            "ServerSideEncrypt
ionByDefault": {
              "SSEAlgorithm": "AES256"
            }
          }
        ]
      },
      "MetricsConfigurations": [
        {
          "Id": "myConfig"
        }
      ],
      "OwnershipControls": {
        "Rules": [
          {
            "ObjectOwnership":
 "BucketOwnerPreferred"
          }
        ]
      },
      "PublicAccessBlockConfigura
tion": {
        "BlockPublicAcls": true,
        "BlockPublicPolicy": true,
        "IgnorePublicAcls": true,
        "RestrictPublicBuckets": true
      },
      "VersioningConfiguration": {
        "Status": "Enabled"
      }
    }
  }
```

```
    serverSideEncryptionConfigu
ration: [
      {
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

ご覧のとおり、L1 コンストラクトは、CloudFormation リソースのコードを正確に示すマニフェストであり、ショートカットや単純化が行われないため、記述が必要なボイラープレートテキストの量はほぼ同じです。ただし、AWS CDK を使用する大きな利点の 1 つは、こうした CloudFormation 構文

ボイラープレートの多くを排除しやすくなることです。これを可能にするために、有用なのが、L2 コンストラクトです。

レイヤー 2 コンストラクト

AWS CDK オープンソースリポジトリ内のコードは、主に TypeScript プログラミング言語で記述され、多数のパッケージとモジュールで構成されています。メインパッケージライブラリは、aws-cdk-lib と呼ばれ、AWS サービスごとに 1 つのパッケージに大まかに分割されていますが、常にその状態であるとは限りません。前述したように、L1 コンストラクトは、ビルドプロセス中に自動生成されます。つまり、リポジトリ内に存在するあらゆるコードは、L2 コンストラクトであり、これらは、L1 コンストラクトが抽象化されたものです。

パッケージには、TypeScript タイプ、列挙型、インターフェイスのコレクションに加え、機能を追加するヘルパークラスも含まれていますが、こうした要素はいずれも、L2 コンストラクトに利用します。どの L2 コンストラクトも、インスタンス化された際に、自身のコンストラクタ内にある対応する L1 コンストラクトを呼び出します。作成された L1 コンストラクトには、レイヤー 2 から次のようにアクセスできます。

```
const role = new Bucket(this, "amzn-s3-demo-bucket", {/*...BucketProps*/});
const cfnBucket = role.node.defaultChild;
```

L2 コンストラクトでは、デフォルトのプロパティ、便利なメソッド、その他のシンタックスシュガーを取り込み、L1 コンストラクトに適用します。これにより、CloudFormation でリソースを直接プロビジョニングするために必要な反復と冗長性の大部分を取り除くことができます。

どの L2 コンストラクトでも、対応する L1 コンストラクトが内部に構築されます。しかし、L2 コンストラクトは、L1 コンストラクトを継承することはありません。L1 コンストラクトも L2 コンストラクトも、Construct という特別なクラスを継承します。AWS CDK バージョン 1 の Construct クラスは、開発キットに組み込まれていましたが、バージョン 2 では、独立したスタンドアロンパッケージとして提供されています。そうすることで、Cloud Development Kit for Terraform (CDKTF) などの他のパッケージ内に、依存関係として追加できるからです。Construct クラスを継承するクラスはいずれも、L1、L2、L3 コンストラクトのいずれかです。次の表に示すように、L2 コンストラクトは、このクラスを直接継承しますが、L1 コンストラクトは CfnResource というクラスを継承します。

L1 継承ツリー	L2 継承ツリー
L1 コンストラクト	L2 コンストラクト
→ クラス CfnResource	→ クラス Construct

→ → 抽象クラス CfnRefElement

→ → → 抽象クラス CfnElement

→ → → → クラス Construct

L1 コンストラクトも L2 コンストラクトも Construct クラスを継承しているのに、なぜ L2 コンストラクトは L1 を継承しないのでしょうか。Construct クラスとレイヤー 1 間にあるクラスでは、CloudFormation リソースをそのまま反映したものとして、L1 コンストラクトが特定の位置に固定されるからです。こうしたクラスには、ダウンストリームクラスに含まれる必要のあるメソッドが含まれています。_toCloudFormation がその例ですが、このメソッドは、CloudFormation 構文が直接出力されるよう強制するものです。L2 コンストラクトは、こうしたクラスをスキップし、Construct クラスを直接継承します。そのように、L2 コンストラクタ内で L1 を個別に構築することで、L1 コンストラクトに必要なコードの多くを柔軟に抽象化できるのです。

前のセクションでは、CloudFormation テンプレートの S3 バケットと、L1 コンストラクトとしてレンダリングした同じ S3 バケットを並べて比較しました。この比較では、プロパティと構文がほぼ同じであることがわかり、L1 コンストラクトのコードは、CloudFormation コンストラクトよりも、3 ~ 4 行だけ短くなっていました。今度は、同じ S3 バケットをそれぞれ記述した L1 コンストラクトと L2 コンストラクトを比較してみましょう。

S3 バケットの L1 コンストラクト

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
    serverSideEncryptionConfigu
ration: [
      {
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
```

S3 バケットの L2 コンストラクト

```
new Bucket(this, "amzns3demobucket",
 {
  bucketName: "amzn-s3-demo-bucket",
  encryption: BucketEncryption.S
3_MANAGED,
  metrics: [
    {
      id: "myConfig"
    },
  ],
  objectOwnership: ObjectOwnership.BU
CKET_OWNER_PREFERRED,
  blockPublicAccess: BlockPubl
icAccess.BLOCK_ALL,
  versioned: true
});
```

```
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

ご覧のとおり、L2 コンストラクトのサイズは、L1 コンストラクトの半分未満です。L2 コンストラクトでは、このような統合を実現するために多くの手法を使用します。そうした手法の一部は 1 つの L2 コンストラクトに適用しますが、それ以外は、複数のコンストラクトで再利用可能なため、クラスとして独立させ再利用します。L2 コンストラクトでは、複数の方法で CloudFormation 構文を統合します。これについては、以下のセクションで説明します。

デフォルトのプロパティ

リソースプロビジョニングのコードを最も簡単に統合する方法とは、最も一般的なプロパティ設定をデフォルトとして扱うことです。AWS CDK では、強力なプログラミング言語にアクセスできますが、CloudFormation ではそれが行えないため、多くの場合、こうしたデフォルト設定には、その性質上、条件が設定されます。これにより、AWS CDK コードから、数行を削除できる場合があります。CloudFormation の設定は、コンストラクトに渡される他のプロパティ値から推測できるからです。

構造体、型、インターフェイス

AWS CDK は複数のプログラミング言語で使用できますが、ネイティブ言語は TypeScript であるため、L2 コンストラクトを構成する型の定義には、言語の型システムを使用します。そうした型システムの掘り下げは、このガイドの範囲外です。詳細については、TypeScript ドキュメントを参照してください。要約すると、TypeScript の `type` とは、特定の変数が保持するデータの種類を表すもので、こうしたデータは、`string` などの基本的なものから、`object` などの複雑なものまで多岐にわたります。TypeScript オブジェクトの型は、TypeScript `interface` によっても表すことができ、`struct` は、インターフェイスの別名とも言えます。

TypeScript では、構造体という用語を使用しませんが、AWS CDK API リファレンスを見ると、構造体がコード内で TypeScript インターフェイスと同じ概念で使用されていることがわかります。同時に、API リファレンスでは、特定のインターフェイスをインターフェイスと呼んでいます。構造体とインターフェイスは同じ概念を持つにもかかわらず、AWS CDK ドキュメントでは、なぜそれらを区別しているのでしょうか。

AWS CDK では、L2 コンストラクトで使用するオブジェクトを表すインターフェイスを構造体と呼んでいます。これには、インスタンス化中に L2 コンストラクトに渡すプロパティ引数のオブジェクト型も含まれます。例えば、S3 バケットコンストラクトの `BucketProps` や DynamoDB Table コンストラクトの `TableProps`、AWS CDK 内で使用するその他の TypeScript インターフェイスなどです。つまり、それが AWS CDK 内の TypeScript インターフェイスであり、名前の先頭に `I` の文字が付いていないものを AWS CDK では構造体と呼びます。

逆に、AWS CDK でインターフェイスという用語が使用されるのは、プレーンオブジェクトによって特定のコンストラクトまたはヘルパークラスを適切に表現するのに必要な基本要素を表す場合です。つまり、インターフェイスによって、L2 コンストラクトのパブリックプロパティを記述します。AWS CDK のインターフェイス名はいずれも、既存のコンストラクトまたはヘルパークラスの名前にプレフィックス `I` を付けて表します。すべての L2 コンストラクトに、`Construct` クラスが継承され、対応するインターフェイスも実装されます。つまり、L2 コンストラクトの `Bucket` には、`IBucket` インターフェイスが実装されます。

静的メソッド

L2 コンストラクトのすべてのインスタンスは、対応するインターフェイスのインスタンスでもありますが、その逆が成り立つことはありません。この点は、構造体を調べ、必要なデータ型を確認する際に重要となります。構造体に `bucket` というプロパティがあり、そのプロパティにデータ型 `IBucket` が必要な場合、`IBucket` インターフェイスにリストしているプロパティが含まれるオブジェクト、または L2 `Bucket` のインスタンスのいずれかを渡すことができます。この場合は、どち

らでも機能しますが、その bucket プロパティに L2 Bucket が必要な場合、そのフィールドに渡すことができるのは Bucket インスタンスのみです。

こうした区別は、既存のリソースをスタックにインポートする際に非常に重要となります。そのスタックのネイティブリソースには L2 コンストラクトを作成できますが、スタック外で作成したリソースの参照が必要な場合は、その L2 コンストラクトのインターフェイスを使用しなければなりません。なぜなら、L2 コンストラクトの作成時にそのスタック内にリソースが存在しない場合、新しいリソースが作成されるからです。既存のリソースを参照するには、その L2 コンストラクトのインターフェイスに準拠するプレーンオブジェクトを使用する必要があります。

実践上、これを簡単にするために、ほとんどの L2 コンストラクトには、一連の静的メソッドが関連付けられており、こうしたメソッドによってインターフェイスが返ります。静的メソッドは通常、from という単語で始まります。これらのメソッドに渡す最初の 2 つの引数は、標準的な L2 コンストラクトに必要な scope および id の引数と同じものです。ただし、3 番目の引数には、props ではなく、インターフェイスを定義する小さいプロパティサブセット (場合によっては 1 つのプロパティのみ) を指定します。そのため、L2 コンストラクトを渡す際には、多くの場合、インターフェイスの要素のみが必要となります。これにより、可能な限りインポート済みのリソースを使用できるようになります。

```
// Example of referencing an external S3 bucket
const preExistingBucket = Bucket.fromBucketName(this, "external-bucket", "name-of-
bucket-that-already-exists");
```

ただし、インターフェイスに過度に依存してはなりません。リソースをインポートしてインターフェイスを直接使用するのは、本当に必要な場合のみにします。インターフェイスには、ヘルパーメソッドといった、L2 コンストラクトの効果を高める各種プロパティが用意されていないからです。

ヘルパーメソッド

L2 コンストラクトは単純なオブジェクトではなくプログラムクラスであるため、これによって、インスタンス化後にリソース設定を操作可能なクラスメソッドを公開できます。それがよくわかる例として、AWS Identity and Access Management (IAM) L2 Role コンストラクトが挙げられます。以下のスニペットに、2 つの方法を示します。ここでは、L2 Role コンストラクトを使用して、同じ IAM ロールを作成しています。

ヘルパーメソッドを使用しない場合:

```
const role = new Role(this, "my-iam-role", {
```

```
   assumedBy: new FederatedPrincipal('my-identity-provider.com'),
   managedPolicies: [
      ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess")
   ],
   inlinePolicies: {
      lambdaPolicy: new PolicyDocument({
         statements: [
            new PolicyStatement({
               effect: Effect.ALLOW,
               actions: [ 'lambda:UpdateFunctionCode' ],
               resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-
function' ]
            })
         ]
      })
   }
});
```

ヘルパーメソッドを使用する場合:

```
const role = new Role(this, "my-iam-role", {
   assumedBy: new FederatedPrincipal('my-identity-provider.com')
});

role.addManagedPolicy(ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess"));
role.attachInlinePolicy(new Policy(this, "lambda-policy", {
   policyName: "lambdaPolicy",
   statements: [
      new PolicyStatement({
         effect: Effect.ALLOW,
         actions: [ 'lambda:UpdateFunctionCode' ],
         resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-function' ]
      })
   ]
 }));
```

インスタンス化後にインスタンスメソッドを使用してリソース設定を操作できるため、L2 コンストラクトを使用すると、以前のレイヤーよりも柔軟性が大幅に向上します。L1 コンストラクトも一部のリソースメソッド (addPropertyOverride など) を継承しますが、そのリソースおよびプロパティ専用に設計されたメソッドは、レイヤー 2 で初めて利用できるようになりました。

列挙型

CloudFormation 構文では、多くの場合、リソースを適切にプロビジョニングするために、多数の詳細項目を指定しなければなりません。しかし、ほとんどのユースケースは、わずか数種類の設定を行うだけで済みます。こうした設定を一連の列挙値で表すと、必要なコード量を大幅に削減できます。

例えば、このセクションで前述した S3 バケットの L2 のコード例では、CloudFormation テンプレートの `bucketEncryption` プロパティを使用する必要があり、これによって、使用する暗号化アルゴリズムの名前などの詳細をすべて指定しました。AWS CDK では、そうした操作ではなく、`BucketEncryption` という列挙型を利用できます。これによって、バケット暗号化で最も一般的な 5 つの形式を受け取り、それぞれを単一の変数名で表現することが可能です。

次に、列挙型では対応できないエッジケースについて考えてみましょう。L2 コンストラクトの目標の 1 つは、レイヤー 1 リソースのプロビジョニングタスクを簡素化することです。そのため、あまり使用されない特定のエッジケースは、レイヤー 2 ではサポートされない可能性があります。これらのエッジケースに対応するために、AWS CDK では、addPropertyOverride メソッドを使用して、CloudFormation リソースの基盤となるプロパティを直接操作できるようになっています。プロパティのオーバーライドの詳細については、このガイドの「ベストプラクティス」セクションと、AWS CDK ドキュメントの「Abstractions and escape hatches」セクションを参照してください。

ヘルパークラス

列挙型では、特定のユースケースのリソース設定に必要なプログラムロジックを実現できないことがあります。こうした状況での AWS CDK では、多くの場合、列挙型ではなく、ヘルパークラスが利用できます。列挙型は単純なオブジェクトで、キーと値の一連のペアが使用可能な一方で、ヘルパークラスは TypeScript クラスの機能をすべて備えています。ヘルパークラスは、静的プロパティの公開によって列挙型のように動作しますが、こうしたプロパティ値は、ヘルパークラスのコンストラクタまたはヘルパーメソッド内の条件付きロジックを使用して内部的に設定できます。

前述のとおり、`BucketEncryption` 列挙型を使用すると、S3 バケットへの暗号化アルゴリズム設定に必要なコードの量を削減できますが、期間を設定する場合は、選択可能な値が多いという理由で、これと同じ戦略の効果は得られません。値ごとの列挙型の作成は、価値に見合わないかなりの労力を伴うからです。そのため、S3 バケットのデフォルト S3 Object Lock 設定には、ヘルパークラスを使用し、これを ObjectLockRetention クラスで表します。`ObjectLockRetention` は、2 つの静的メソッドで構成されます。1 つは、コンプライアンス維持を、もう 1 つはガバナンス維持を目的とするものです。どちらのメソッドも、ロック設定が必要な期間を表す Duration ヘルパークラスのインスタンスを引数として受け取ります。

もう 1 つの例として、AWS Lambda のヘルパークラスである Runtime が挙げられます。このクラスに関連付けられた静的プロパティは、一見すると、列挙型で処理できそうに思えます。しかし、内部的には、各プロパティ値は `Runtime` クラス自体のインスタンスを表しているため、クラスのコンストラクタで実行されるロジックは、列挙型では実現できません。

レイヤー 3 コンストラクト

L1 コンストラクトでは CloudFormation リソースをプログラムコードにそのまま変換し、L2 コンストラクトでは冗長な CloudFormation 構文の多くをヘルパーメソッドやカスタムロジックに置換しました。では、L3 コンストラクトではどのような処理を行うのでしょうか。それは、ユーザーの想像力によって決まります。レイヤー 3 は、どのような特定のユースケースにも合うよう作成できるのです。例えば、プロジェクトで特定のプロパティサブセットを持つリソースが必要な場合は、再利用可能な L3 コンストラクトを作成して、そうしたニーズを満たすことができます。

AWS CDK内では、L3 コンストラクトをパターンと呼びます。パターンは、 のConstructクラスを拡張する AWS CDK (またはクラスを拡張するConstruct クラスを拡張する) オブジェクトで、レイヤー 2 を超えて抽象化されたロジックを実行します。 AWS CDK CLI を使用して cdk init を実行して新しい AWS CDK プロジェクトを開始する場合は、app、、 libの 3 つの AWS CDK アプリケーションタイプから選択する必要がありますsample-app。

```
Available templates:
* app: Template for a CDK Application
   └ cdk init app --language=[csharp|fsharp|go|java|javascript|python|typescript]
* lib: Template for a CDK Construct Library
   └ cdk init lib --language=typescript
* sample-app: Example CDK Application with some constructs
   └ cdk init sample-app --language=[csharp|fsharp|go|java|javascript|python|typescript]
```

app と sample-appはどちらもCloudFormation スタックを構築して AWS 環境にデプロイする従来の AWS CDK アプリケーションを表します。を選択するとlib、まったく新しい L3 コンストラクトの構築が選択されます。 appと sample-appでは、 が AWS CDK サポートする任意の言語を選択できますが、 では TypeScript しか選択できませんlib。これは、 AWS CDK が TypeScript でネイティブに記述され、JSii というオープンソースシステムを使用して元のコードを他のサポートされている言語に変換するためです。 AWS CDKの拡張機能を構築する場合は、lib を選択してプロジェクトを開始します。

Construct クラスを継承するクラスはすべて L3 コンストラクトとして定義できますが、レイヤー 3 の最も一般的なユースケースは、リソースインタラクション、リソース拡張、カスタムリソースです。ほとんどの L3 コンストラクトでは、 AWS CDK の機能を拡張するために、これら 3 つのケースの 1 つ以上を使用します。

リソースインタラクション

ソリューションでは、通常、連携して動作する複数の AWS サービスを利用します。例えば、Amazon CloudFront ディストリビューションでは、S3 バケットをオリジンとして使用し、一般的なエクスプロイトから保護 AWS WAF することがよくあります。 AWS AppSync Amazon API Gateway では、APIs のデータソースとして Amazon DynamoDB テーブルがよく使用されます。のパイプラインでは、Amazon S3 をソースとして使用し、ビルドステージ AWS CodeBuild に使用する AWS CodePipeline ことがよくあります。こうした場合に有用なのは、単一の L3 コンストラクトを作成し、これによって、相互接続した 2 つ以上の L2 コンストラクトのプロビジョニングを処理することです。

CloudFront ディストリビューションを SL3 オリジン、その前に配置 AWS WAF する 、Amazon Route 53 レコード、転送中の暗号化でカスタムエンドポイントを追加する AWS Certificate Manager (ACM) 証明書とともにプロビジョニングする L3 コンストラクトの例を次に示します。これらはすべて 1 つの再利用可能なコンストラクトです。 S3

```
// Define the properties passed to the L3 construct
export interface CloudFrontWebsiteProps {
     distributionProps: DistributionProps
     bucketProps: BucketProps
     wafProps: CfnWebAclProps
     zone: IHostedZone
}

// Define the L3 construct
export class CloudFrontWebsite extends Construct {
     public distribution: Distribution

     constructor(
        scope: Construct,
        id: string,
        props: CloudFrontWebsiteProps
     ) {
         super(scope, id);

         const certificate = new Certificate(this, "Certificate", {
            domainName: props.zone.zoneName,
            validation: CertificateValidation.fromDns(props.zone)
         });
         const defaultBehavior = {
            origin: new S3Origin(new Bucket(this, "bucket", props.bucketProps))
```

```
        }
        const waf = new CfnWebACL(this, "waf", props.wafProps);
        this.distribution = new Distribution(this, id, {
           ...props.distributionProps,
           defaultBehavior,
           certificate,
           domainNames: [this.domainName],
           webAclId: waf.attrArn,
         });
    }
}
```

CloudFront、Amazon S3、Route 53、ACM ではいずれも、L2 コンストラクトを使用していますが、Web ACL (Web リクエストの処理ルールを定義する) では L1 コンストラクトを使用していることに注意してください。これは、 AWS CDK は完全には完了していない進化するオープンソースパッケージであり、`WebAcl`まだ L2 コンストラクトがないためです。ただし、新しい L2 コンストラクトを作成 AWS CDK することで、誰でも に貢献できます。したがって、 が の L2 コンストラクト AWS CDK を提供するまでは`WebAcl`、L1 コンストラクトを使用する必要があります。`CloudFrontWebsite` L3 コンストラクトを使用して新しいウェブサイトを作成するには、次のコードを使用します。

```
const siteADotCom = new CloudFrontWebsite(stack, "siteA", siteAProps);
const siteBDotCom = new CloudFrontWebsite(stack, "siteB", siteBProps);
const siteCDotCom = new CloudFrontWebsite(stack, "siteC", siteCProps);
```

この例では、CloudFront `Distribution` の L2 コンストラクトを L3 コンストラクトのパブリックプロパティとして公開しています。このように、場合によっては、L3 プロパティの公開が必要なケースも依然として存在します。実際に、以降の「カスタムリソース」セクションでも、`Distribution` を再度取り上げます。

AWS CDK には、このようなリソースインタラクションパターンの例がいくつか含まれています。例えば、Amazon Elastic Container Service (Amazon ECS) の L2 コンストラクトが含まれる `aws-ecs` パッケージのほか、 AWS CDK には、aws-ecs-patterns というパッケージもあります。このパッケージ内には、Amazon ECS を Application Load Balancer、Network Load Balancer、ターゲットグループと組み合わせる複数の L3 コンストラクトに加え、Amazon Elastic Compute Cloud (Amazon EC2) と AWS Fargate向けにプリセットされた異なるバージョンが用意されています。多くのサーバーレスアプリケーションでは Amazon ECS を Fargate とのみ組み合わせて使用するため、こうした L3 コンストラクトを使用すると、開発側の時間と顧客側のコストをともに節約できます。

リソース拡張

一部のユースケースのリソースには、L2 コンストラクトでネイティブではない特定のデフォルト設定が必要です。スタックレベルであれば、アスペクトを使用してこれに対応できますが、L2 コンストラクトに新しいデフォルトを設定するもう 1 つの便利な方法は、レイヤー 2 を継承することです。どのコンストラクトも Construct クラスを継承したコンストラクトであり、L2 コンストラクトはそのクラスを継承するため、L2 コンストラクトを直接継承することでも L3 コンストラクトを作成できます。

これが特に有用なのは、顧客の特定のニーズに対応したカスタムビジネスロジックを実行するときです。会社には、 という名前の単一のディレクトリにすべての AWS Lambda 関数コードを保存src/lambdaし、ほとんどの Lambda 関数が毎回同じランタイムとハンドラー名を再利用するリポジトリがあるとします。その場合は、Lambda 関数の新規設定のたびにコードパスを設定せず、新しい L3 コンストラクトを作成すると良いでしょう。

```
export class MyCompanyLambdaFunction extends Function {
      constructor(
        scope: Construct,
        id: string,
        props: Partial<FunctionProps> = {}
     ) {
         super(scope, id, {
            handler: 'index.handler',
            runtime: Runtime.NODEJS_LATEST,
            code: Code.fromAsset(`src/lambda/${props.functionName || id}`),
            ...props
     });
}
```

そうすることにより、リポジトリ内の任意の場所で L2 Function のコンストラクトを次のように置き換えることができます。

```
new MyCompanyLambdaFunction(this, "MyFunction");
new MyCompanyLambdaFunction(this, "MyOtherFunction");
new MyCompanyLambdaFunction(this, "MyThirdFunction", {
   runtime: Runtime.PYTHON_3_11
});
```

デフォルト設定を使用すると、新しい Lambda 関数を 1 行で作成でき、必要に応じてデフォルトのプロパティをオーバーライドできるように L3 コンストラクトを設定することも可能です。

既存の L2 コンストラクトに新しいデフォルト設定を追加するだけの場合に最良なのは、L2 コンストラクトを直接継承することです。他のカスタムロジックも必要な場合は、Construct クラスを継承すると良いでしょう。その理由は、コンストラクタ内で呼び出す super メソッドにあります。他のクラスを継承したクラスでは、super メソッドを使用して親クラスのコンストラクタを呼び出します。これは、コンストラクタ内で最初に実行する必要があります。つまり、渡す引数やその他のカスタムロジックの操作は、元の L2 コンストラクトが作成された後にのみ実行できます。L2 コンストラクトのインスタンス化前にこうしたカスタムロジックを実行する必要がある場合は、「リソースインタラクション」セクションで前述したパターンに従うことをお勧めします。

カスタムリソース

カスタムリソースは、CloudFormation の強力な機能であり、これによって、スタックのデプロイ中にアクティブ化した Lambda 関数からカスタムロジックを実行できます。CloudFormation で直接サポートされていないプロセスがデプロイ中に必要な場合は、カスタムリソースを使用して処理すると良いでしょう。 AWS CDK には、カスタムリソースをプログラムで作成可能なクラスも用意されています。L3 コンストラクタ内でカスタムリソースを使用すると、ほぼあらゆるリソースからコンストラクトを作成できます。

Amazon CloudFront を使用する利点の 1 つは、強力なグローバルキャッシュ機能です。キャッシュを手動でリセットしてオリジンへの新しい変更をウェブサイトにすぐに反映したい場合は、CloudFront の無効化を利用できます。ただし、無効化は、CloudFront ディストリビューション上で実行されるプロセスであり、CloudFront ディストリビューションのプロパティではありません。既存のディストリビューションにいつでも作成し適用できるため、プロビジョニングおよびデプロイプロセスに、ネイティブには組み込まれていません。

このシナリオでは、ディストリビューションのオリジンを更新するたびに無効化を作成し実行するという要件が考えられます。カスタムリソースを使用すると、次のような L3 コンストラクトを作成できます。

```
export interface CloudFrontInvalidationProps {
     distribution: Distribution
     region?: string
     paths?: string[]
}

export class CloudFrontInvalidation extends Construct {
      constructor(
        scope: Construct,
        id: string,
```

```
        props: CloudFrontInvalidationProps
    ) {
        super(scope, id);
        const policy = AwsCustomResourcePolicy.fromSdkCalls({
            resources:AwsCustomResourcePolicy.ANY_RESOURCE
        });
        new AwsCustomResource(scope, `${id}Invalidation`, {
           policy,
           onUpdate: {
                service: 'CloudFront',
                action: 'createInvalidation',
                region: props.region || 'us-east-1',
                physicalResourceId:
 PhysicalResourceId.fromResponse('Invalidation.Id'),
                parameters: {
                   DistributionId: props.distribution.distributionId,
                   InvalidationBatch: {
                       Paths: {
                          Quantity: props.paths?.length || 1,
                          Items: props.paths || ['/*']
                       },
                       CallerReference: crypto.randomBytes(5).toString('hex')
                   }
                }
           }
        }
    }
}
```

先ほど CloudFrontWebsite L3 コンストラクトで作成したディストリビューションを使用すると、これを非常に簡単に実行できます。

```
new CloudFrontInvalidation(this, 'MyInvalidation', {
       distribution: siteADotCom.distribution
});
```

この L3 コンストラクトは、AwsCustomResource と呼ばれる AWS CDK L3 コンストラクトを使用して、カスタムロジックを実行するカスタムリソースを作成します。AwsCustomResourceは、Lambda コードを記述しなくても 1 つの AWS SDK 呼び出しを正確に行う必要がある場合に非常に便利です。より複雑な要件があり、独自ロジックの実装が必要な場合は、基本的な CustomResource クラスを直接使用すると良いでしょう。

カスタムリソース L3 コンストラクト AWS CDK を使用する のもう 1 つの良い例は、S3 バケットのデプロイです。カスタムリソースによってこの L3 コンストラクトのコンストラクタ内に作成された Lambda 関数は、CloudFormation では処理できない機能、つまり S3 バケットにオブジェクトを追加し更新する機能を備えるようになります。S3 バケットのデプロイ機能がないと、スタックの一部として作成した S3 バケットにコンテンツを追加できないため、非常に不便です。

CloudFormation 構文のリームを書き出す必要がなく AWS CDK なるのに最適な例は、次の基本的なですS3BucketDeployment。

```
new BucketDeployment(this, 'BucketObjects', {
  sources: [Source.asset('./path/to/amzn-s3-demo-bucket')],
  destinationBucket: amzn-s3-demo-bucket
});
```

これと同じ目的を達成するために、CloudFormation ではどれくらいの量のコードが必要かを比較してみましょう。

```
"lambdapolicyA5E98E09": {
   "Type": "AWS::IAM::Policy",
   "Properties": {
    "PolicyDocument": {
     "Statement": [
      {
       "Action": "lambda:UpdateFunctionCode",
       "Effect": "Allow",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
      }
     ],
     "Version": "2012-10-17"
    },
    "PolicyName": "lambdaPolicy",
    "Roles": [
     {
      "Ref": "myiamroleF09C7974"
     }
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/lambda-policy/Resource"
   }
  },
  "BucketObjectsAwsCliLayer8C081206": {
```

```
    "Type": "AWS::Lambda::LayerVersion",
    "Properties": {
     "Content": {
      "S3Bucket": {
       "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
      },
      "S3Key": "e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip"
     },
     "Description": "/opt/awscli/aws"
    },
    "Metadata": {
     "aws:cdk:path": "CdkScratchStack/BucketObjects/AwsCliLayer/Resource",
     "aws:asset:path":
"asset.e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip",
     "aws:asset:is-bundled": false,
     "aws:asset:property": "Content"
    }
   },
   "BucketObjectsCustomResourceB12E6837": {
    "Type": "Custom::CDKBucketDeployment",
    "Properties": {
     "ServiceToken": {
      "Fn::GetAtt": [
       "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536",
       "Arn"
      ]
     },
     "SourceBucketNames": [
      {
       "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
      }
     ],
     "SourceObjectKeys": [
      "f888a9d977f0b5bdbc04a1f8f07520ede6e00d4051b9a6a250860a1700924f26.zip"
     ],
     "DestinationBucketName": {
      "Ref": "amzn-s3-demo-bucket77F80CC0"
     },
     "Prune": true
    },
    "UpdateReplacePolicy": "Delete",
    "DeletionPolicy": "Delete",
    "Metadata": {
     "aws:cdk:path": "CdkScratchStack/BucketObjects/CustomResource/Default"
```

```
    }
   },
   "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265": {
    "Type": "AWS::IAM::Role",
    "Properties": {
     "AssumeRolePolicyDocument": {
      "Statement": [
       {
        "Action": "sts:AssumeRole",
        "Effect": "Allow",
        "Principal": {
         "Service": "lambda.amazonaws.com"
        }
       }
      ],
      "Version": "2012-10-17"
     },
     "ManagedPolicyArns": [
      {
       "Fn::Join": [
        "",
        [
         "arn:",
         {
          "Ref": "AWS::Partition"
         },
         ":iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        ]
       ]
      }
     ]
    },
    "Metadata": {
     "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/Resource"
    }
   },

  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF":
  {
    "Type": "AWS::IAM::Policy",
    "Properties": {
     "PolicyDocument": {
      "Statement": [
```

```
      {
       "Action": [
        "s3:GetBucket*",
        "s3:GetObject*",
        "s3:List*"
       ],
       "Effect": "Allow",
       "Resource": [
        {
         "Fn::Join": [
          "",
          [
           "arn:",
           {
            "Ref": "AWS::Partition"
           },
           ":s3:::",
           {
            "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
           },
           "/*"
          ]
         ]
        },
        {
         "Fn::Join": [
          "",
          [
           "arn:",
           {
            "Ref": "AWS::Partition"
           },
           ":s3:::",
           {
            "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
           }
          ]
         ]
        }
       ]
      },
      {
       "Action": [
        "s3:Abort*",
```

```
        "s3:DeleteObject*",
        "s3:GetBucket*",
        "s3:GetObject*",
        "s3:List*",
        "s3:PutObject",
        "s3:PutObjectLegalHold",
        "s3:PutObjectRetention",
        "s3:PutObjectTagging",
        "s3:PutObjectVersionTagging"
       ],
       "Effect": "Allow",
       "Resource": [
        {
         "Fn::GetAtt": [
          "amzns3demobucket77F80CC0",
          "Arn"
         ]
        },
        {
         "Fn::Join": [
          "",
          [
           {
            "Fn::GetAtt": [
             "amzns3demobucket77F80CC0",
             "Arn"
            ]
           },
           "/*"
          ]
         ]
        }
       ]
      }
     ],
     "Version": "2012-10-17"
    },
    "PolicyName":
"CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
    "Roles": [
     {
      "Ref":
"CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
     }
```

```
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/DefaultPolicy/
Resource"
   }
  },
  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536": {
   "Type": "AWS::Lambda::Function",
   "Properties": {
    "Code": {
     "S3Bucket": {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     },
     "S3Key": "9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd.zip"
    },
    "Role": {
     "Fn::GetAtt": [
      "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265",
      "Arn"
     ]
    },
    "Environment": {
     "Variables": {
      "AWS_CA_BUNDLE": "/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem"
     }
    },
    "Handler": "index.handler",
    "Layers": [
     {
      "Ref": "BucketObjectsAwsCliLayer8C081206"
     }
    ],
    "Runtime": "python3.9",
    "Timeout": 900
   },
   "DependsOn": [

 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
    "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
   ],
   "Metadata": {
```

```
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/Resource",
    "aws:asset:path":
 "asset.9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd",
    "aws:asset:is-bundled": false,
    "aws:asset:property": "Code"
   }
  }
```

4 行に対し 241 行とは、かなりの差です。しかし、これは、レイヤー 3 を活用してスタックをカスタマイズする場合の可能性を示す一例にすぎません。

ベストプラクティス

L1 コンストラクト

- L1 コンストラクトを直接使用することを常に避けることはできないものの、可能な限り避けるようにすべきです。特定の L2 コンストラクトがエッジケースに対応していない場合は、L1 コンストラクトを直接使用する代わりに、以下の 2 つの選択肢を検討すると良いでしょう。
  - **`defaultChild`** にアクセスする: 必要な CloudFormation プロパティが L2 コンストラクトで利用できない場合は、`L2Construct.node.defaultChild` を使用して、基盤となる L1 コンストラクトにアクセスできます。このプロパティを介してアクセスすると、L1 コンストラクトの public プロパティを更新可能なため、L1 コンストラクトを手動で作成しなくても済みます。
  - プロパティオーバーライドを使用する: 更新対象のプロパティが public でない場合の対処を考えてみましょう。CloudFormation テンプレートで実行可能な操作を、AWS CDK でも実行できるようにする究極の方法は、addPropertyOverride を使用することです。このメソッドは、すべての L1 コンストラクトで利用できます。このメソッドに CloudFormation プロパティの名前と値を直接渡すと、CloudFormation テンプレートレベルでスタック操作が可能になります。

L2 コンストラクト

- ここで重要なのは、多くの場合 L2 コンストラクトに用意されているヘルパーメソッドの活用です。レイヤー 2 では、インスタンス化時にすべてのプロパティを渡す必要はありません。L2 ヘルパーメソッドを使用すると、リソースのプロビジョニングが飛躍的に容易になります。特に、条件付きロジックが必要な場合にそれが顕著です。Grant クラスから派生したヘルパーメソッドは、最も便利なヘルパーメソッドの 1 つと言えるでしょう。このクラスは、直接使用するものではありませんが、多くの L2 コンストラクトでは、このクラスを使用して、権限の実装を大幅に簡素化するヘルパーメソッドが提供されています。例えば、L2 Lambda 関数に L2 S3 バケットへのアクセス権限を付与する場合、`s3Bucket.grantReadWrite(lambdaFunction)` を呼び出すことができるため、新しいロールやポリシーを作成しなくても済みます。

L3 コンストラクト

- L3 コンストラクトは、スタックの再利用性とやカスタマイズ性を高めるときに非常に便利なものですが、慎重に使用することをお勧めします。どのような L3 コンストラクトのタイプが必要かや、そもそも L3 コンストラクトが必要かどうかを検討してください。
  - AWS リソースと直接やり取りしない場合は、一般的に、`Construct` クラスを継承するよりもヘルパークラスを作成する方が適切です。なぜなら、`Construct` クラスがデフォルトで実行す

るアクションの多くは、AWS リソースと直接やり取りする場合にのみ必要となるからです。こうしたアクションの実行が不要な場合は、それらを実行しない方が効率的です。

- L3 コンストラクトの新規作成が適切と判断したら、`Construct` クラスを直接継承してください。他の L2 コンストラクトの継承は、そのコンストラクトのデフォルトプロパティを更新する場合にのみ行います。他の L2 コンストラクトやカスタムロジックが必要な場合は、`Construct` を直接継承し、そのコンストラクタ内ですべてのリソースをインスタンス化します。

よくある質問

レイヤーを理解 AWS CDK せずに を使用することはできませんか?

もちろんです。ただし、最も強力なツールと同様に、 はより強力 AWS CDK になります。 AWS CDKレイヤーがどのように相互作用するかを理解することで、基本的な AWS CDK 知識だけでは不可能なスタックのデプロイを簡素化するのに役立つ新しいレベルの理解が解き放たれます。

L2 から L3 コンストラクトを作成する場合と同じ方法で L1 から L2 コンストラクトを作成することはできますか?

リソースに既に L2 コンストラクトがある場合は、そのコンストラクトを使用してレイヤー 3 でカスタマイズすることをお勧めします。なぜなら、特定のリソースに既存の L2 コンストラクトを設定する最善の方法を解き明かそうと、多くの研究が既に行われているからです。しかし、一部の L1 コンストラクトには、L2 コンストラクトがまだ存在しません。そのような場合は、 AWS CDK オープンソースライブラリのコントリビューターになることをお勧めします。ぜひ、独自の L2 コンストラクトを作成し、他のユーザーと共有してください。 AWS CDKのコントリビューターガイドラインには、それを始めるために必要なあらゆる情報が記載されています。

公式の L2 コンストラクトがまだない AWS リソースはどれですか?

L2 コンストラクトを持たない AWS リソースの数は日ごとに減少していますが、これらのリソースのいずれかの L2 コンストラクトの作成を支援したい場合は、 AWS CDK API リファレンスを参照してください。左側ペインのリソースリストで、名前の横に上付き文字「1」があるリソースには、公式の L2 コンストラクトがありません。

が AWS CDK サポートする任意の言語で L2 または L3 コンストラクトを作成できますか?

は、TypeScript、JavaScript、Python、Java、C#、Go など、いくつかのプログラミング言語 AWS CDK をサポートしています。関連する言語にコンパイルされた AWS CDK コードを使用して、個人

用 L3 コンストラクトを作成できます。ただし、 に貢献 AWS CDK したり、ネイティブ AWS CDK コンストラクトを作成したりする場合は、TypeScript を使用する必要があります。 AWS CDKでは、TypeScript が唯一のネイティブ言語だからです。他の言語 AWS CDK のバージョンは、JSii という AWS ライブラリを使用してネイティブ TypeScript コードから構築されます。

AWS CDK以外では、どこで、既存の L3 コンストラクトを入手できますか?

ここで共有する場所が多すぎますが、最も人気のあるコンストラクトの多くについては、AWS「ソリューションコンストラクト」のウェブサイトと「コンストラクトハブ」の AWS CDK 「」セクションを参照してください。 https://constructs.dev/search?q=&cdk=aws-cdk&cdkver=2&sort=downloadsDesc&offset=0

リソース

- AWS CDK API リファレンス
- AWS CloudFormation リソース仕様
- AWS CDK コンストラクトに関するドキュメント
- AWS CDK の抽象化およびエスケープハッチ
- Leverage L2 constructs to reduce the complexity of your AWS CDK application (AWS ブログ記事)
- AWS CloudFormation カスタムリソース
- AWS Solutions Constructs
- Construct Hub
- AWS CDK の例 (GitHub リポジトリ)

ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、RSS フィード をサブスクライブできます。

変更	説明	日付
初版発行	—	2023 年 12 月 4 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。