layer AWS CDK 指南

AWS 方案指引

AWS 方案指引: layer AWS CDK 指南

Table of Contents

layer AWS CDK 指南

Steven Guggenheimer，Amazon Web Services (AWS)

2023 年 12 月 (文件歷史記錄)

背後的主要概念之一 AWS Cloud Development Kit (AWS CDK) 與在冷天保持暖和的概念非常相似。該概念稱為分層。在冷天穿著襯衫、夾克，有時甚至更大型的夾克，具體取決於它有多冷。然後，如果您進入其中且加熱器正在閃電，您可以脫下一個或兩個夾克層，這樣您就不會太熱。 AWS CDK 使用分層為使用雲端元件提供不同層級的抽象。分層可確保在部署基礎設施做為程式碼 (IAC) 堆疊時，您永遠不需要撰寫太多程式碼或對資源屬性的存取過少。

如果您不使用 AWS CDK，則必須手動撰寫AWS CloudFormation範本；也就是說，您只會利用單一 layer 來強制您撰寫比平常更多的程式碼。另一方面，如果 AWS CDK 是要抽象化 CloudFormation 中通常不需要寫出的所有內容，您將無法處理任何邊緣案例。

為了解決此問題， 會將資源佈建 AWS CDK 分割成三個不同的層級：

- 第 1 層 – CloudFormation 層：CloudFormation 資源和資源 AWS CDK 幾乎完全相同的最基本層。
- 圖層 2 – 精選圖層：CloudFormation 資源抽象至程式設計類別的圖層，可簡化機庫下的大部分樣板 CloudFormation 語法。此 layer 組成大部分的 AWS CDK。
- 第 3 層 – 模式層：最抽象的層，您可以使用第 1 層和第 2 層提供的建置區塊來自訂特定使用案例的程式碼。

每個 layer 中的每個項目都是稱為 之特殊 AWS CDK 類別的執行個體`Construct`。根據AWS 文件，建構是「 AWS CDK 應用程式的基本建置區塊。建構代表「雲端元件」，並封裝建立元件 AWS CloudFormation 所需的一切。」 這些層內的建構稱為 L1、L2 和 L3 建構，取決於其所屬層。在本指南中，我們將瀏覽每一 AWS CDK 層，以了解它們的用途及其重要性。

本指南適用於技術經理、主管和開發人員，他們有興趣更深入探索讓 AWS CDK 工作成為工作的核心概念。 AWS CDK 是熱門的工具，但團隊通常會錯過其提供的大部分內容。當您開始了解本指南中所述的概念時，您可以釋放充滿可能性的全新世界，並最佳化團隊的資源佈建程序。

在本指南中：

- 最佳實務
- 常見問答集
- 資源

第 1 層建構

L1 建構模組是 的建置區塊， AWS CDK 且很容易與其他建構模組區別為字首 `Cfn`。例如， 中的 Amazon DynamoDB `Table` 套件 AWS CDK 包含一個建構，即 L2 建構。對應的 L1 建構稱為 `CfnTable`，直接代表 CloudFormation DynamoDB `Table`。雖然 AWS CDK 應用程式通常不會直接使用 L1 建構，但無法在 AWS CDK 未存取第一層的情況下使用 。不過，在大多數情況下，開發人員習慣使用的 L2 和 L3 建構會大量依賴 L1 建構。因此，您可以將 L1 建構視為 CloudFormation 與 之間的橋樑 AWS CDK。

的唯一目的是使用標準編碼語言 AWS CDK 產生 CloudFormation 範本。在您執行 cdk 合成 CLI 命令並產生產生的 CloudFormation 範本後， AWS CDK任務即完成。cdk 部署命令僅為方便起見，但執行該命令時所執行的動作完全發生在 CloudFormation 中。將 AWS CDK 程式碼轉譯為 CloudFormation 理解格式的拼圖片段是 L1 建構。

L1 建構的 AWS CDK–CloudFormation 生命週期

建立和使用 L1 建構的程序包含下列步驟：

1. AWS CDK 建置程序會將 CloudFormation 規格轉換為 L1 建構的程式設計程式碼。
2. 開發人員編寫程式碼，直接或間接參考 L1 建構做為 AWS CDK 應用程式的一部分。
3. 開發人員執行 cdk 合成命令，將程式設計程式碼轉換回 CloudFormation 規格 （範本） 指定的格式。
4. 開發人員會執行 cdk 部署命令，在這些範本中將 CloudFormation 堆疊部署到 AWS 帳戶環境。

讓我們做一些練習。前往 GitHub 上的AWS CDK 開放原始碼儲存庫，挑選隨機 AWS 服務，然後前往該服務的 AWS CDK 套件 （位於資料夾 `packages`、`aws-cdk-lib`、`aws-<servicename>`、)`lib`。在此範例中，讓我們挑選 Amazon S3，但這適用於任何服務。如果您查看該套件的主要 index.ts 檔案，您會看到一行顯示：

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

不過，您不會在對應目錄中的任何位置看到 `s3.generated` 檔案。這是因為 L1 建構會在 AWS CDK 建置程序期間從 CloudFormation 資源規格自動產生。因此，只有在您執行套件的 AWS CDK 建置命令之後，才會在套件`s3.generated`中看到 。

AWS CloudFormation 資源規格

AWS CloudFormation 資源規格將基礎設施定義為 的程式碼 (IAC)， AWS 並決定 CloudFormation 範本中的程式碼如何轉換為帳戶中 AWS 的資源。此規格會在每個區域層級上以 JSON 格式定義 AWS 資源。每個資源都會獲得遵循格式 的唯一資源類型名稱`provider::service::resource`。例如，Amazon S3 儲存貯體的資源類型名稱為 `AWS::S3::Bucket`，Amazon S3 存取點的資源類型名稱則為 `AWS::S3::AccessPoint`。您可以使用 資源規格中定義的語法，在 CloudFormation 範本中轉譯這些 AWS CloudFormation 資源類型。當 AWS CDK 建置程序執行時，每個資源類型也會成為 L1 建構。

因此，每個 L1 建構都是其對應 CloudFormation 資源的程式設計鏡像影像。當您執行個體化 L1 建構時，您可以在 CloudFormation 範本中套用的每個屬性都可用，而當您執行個體化對應的 L1 建構時，也需要每個必要的 CloudFormation 屬性做為引數。下表將 CloudFormation 範本中顯示的 S3 儲存貯體與定義為 an AWS CDK L1 建構的相同 S3 儲存貯體進行比較。

CloudFormation 範本

```
"amzns3demobucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": "amzn-s3-demo-
bucket",
      "BucketEncryption": {
        "ServerSideEncryptionConfig
uration": [
          {
            "ServerSideEncrypt
ionByDefault": {
              "SSEAlgorithm": "AES256"
            }
          }
        ]
      },
      "MetricsConfigurations": [
        {
          "Id": "myConfig"
        }
      ],
      "OwnershipControls": {
        "Rules": [
          {
```

L1 建構

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

```
            "ObjectOwnership":
 "BucketOwnerPreferred"
          }
        ]
      },
      "PublicAccessBlockConfigura
tion": {
        "BlockPublicAcls": true,
        "BlockPublicPolicy": true,
        "IgnorePublicAcls": true,
        "RestrictPublicBuckets": true
      },
      "VersioningConfiguration": {
        "Status": "Enabled"
      }
    }
  }
```

```
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

如您所見，L1 建構是 CloudFormation 資源程式碼中的確切資訊清單。沒有捷徑或簡化，因此必須寫入的樣板文字量大致相同。不過，使用 AWS CDK 的一大優點是，它有助於消除許多樣板 CloudFormation 語法。那麼，這種情況如何發生？ 這就是 L2 建構的來源。

第 2 層建構

AWS CDK 開放原始碼儲存庫主要是使用 TypeScript 程式設計語言撰寫，由許多套件和模組組成。主要套件程式庫稱為 aws-cdk-lib，大致分為每個 AWS 服務一個套件，但情況並非總是如此。如前所述，L1 建構會在建置程序期間自動產生，所以當您查看儲存庫時，看到的所有程式碼為何？ 這些是 L2 建構，這是 L1 建構的抽象。

這些套件也包含 TypeScript 類型、列舉和介面的集合，以及新增更多功能的協助程式類別，但這些項目都提供 L2 建構。所有 L2 建構函數會在執行個體化時在其建構函數中呼叫其對應的 L1 建構函數，而建立的 L1 建構函數可以從第 2 層存取，如下所示：

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

L2 建構會使用預設屬性、便利方法和其他語法含糖，並將其套用至 L1 建構。這消除了直接在 CloudFormation 中佈建資源所需的大部分重複性和詳細程度。

所有 L2 建構模組都會在幕後建置其對應的 L1 建構模組。不過，L2 建構不會實際擴展 L1 建構。L1 和 L2 建構都會繼承稱為 Construct 的特殊類別。在 AWS CDK Construct 類別的第 1 版中， 類別是內建在開發套件中，但在第 2 版中則是獨立的套件。因此，雲端開發套件 (CDKTF) 等其他套件可以將其納入做為相依性。繼承類別的任何Construct類別都是 L1, L2 或 L3 建構。L2 建構模組會直接擴展此類別，而 L1 建構模組則會擴展名為 的類別CfnResource，如下表所示。

L1 繼承樹	L2 繼承樹
L1 建構	L2 建構
→ CfnResource 類別 CfnResource	→ 類別建構
→→ 抽象類別 CfnRefElement	
→→→ 抽象類別 CfnElement	
→→→→ 類別建構	

如果 L1 和 L2 建構都繼承 Construct類別，為什麼不 L2 建構只延伸 L1？ 類別Construct與層 1 之間的類別會將 L1 建構鎖定到位，做為 CloudFormation 資源的鏡像影像。它們包含抽象方法 （ 下

游類別必須包含的方法），例如 _toCloudFormation，這會強制建構直接輸出 CloudFormation 語法。L2 建構會略過這些類別，並直接擴展該Construct類別。這可讓他們彈性地抽象 L1 建構所需的大部分程式碼，方法是在建構函數中分別建置這些程式碼。

上一節針對來自 CloudFormation 範本的 S3 儲存貯體，以及轉譯為 L1 建構的相同 S3 儲存貯體，side-by-side比較。該比較顯示屬性和語法幾乎相同，L1 建構與 CloudFormation 建構相比，只會儲存三行或四行。現在，讓我們比較 L1 建構與相同 S3 儲存貯體的 L2 建構：

S3 儲存貯體的 L1 建構

```
new CfnBucket(this, "amzns3demobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
    serverSideEncryptionConfiguration: [
      {
        serverSideEncryptionByDefault: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOwnerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
```

S3 儲存貯體的 L2 建構

```
new Bucket(this, "amzns3demobucket",
 {
  bucketName: "amzn-s3-demo-bucket",
  encryption: BucketEncryption.S3_MANAGED,
  metrics: [
    {
      id: "myConfig"
    },
  ],
  objectOwnership: ObjectOwnership.BUCKET_OWNER_PREFERRED,
  blockPublicAccess: BlockPublicAccess.BLOCK_ALL,
  versioned: true
});
```

```
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

如您所見，L2 建構體小於 L1 建構體大小的一半。L2 建構會使用多種技術來完成此整合。其中一些技術適用於單一 L2 建構模組，但其他技術可以在多個建構模組之間重複使用，以便將其分隔成自己的類別以重複使用。L2 建構會以多種方式合併 CloudFormation 語法，如以下章節所述。

預設屬性

合併用於佈建資源的程式碼最簡單的方法是將最常見的屬性設定轉換為預設值。 AWS CDK 可存取強大的程式設計語言，而 CloudFormation 則無法存取，因此這些預設值通常具有條件性。有時可以從 AWS CDK 程式碼中消除數行 CloudFormation 組態，因為這些設定可以從傳遞給建構模組的其他屬性值推斷。

結構、類型和界面

雖然 AWS CDK 提供多種程式設計語言，但會以 TypeScript 原生撰寫，因此語言的類型系統可用來定義組成 L2 建構的類型。深入探索該類型的系統超出本指南的範圍；如需詳細資訊，請參閱 TypeScript 文件。總而言之，TypeScript `type`會描述特定變數所保留的資料類型。這可以是基本資料，例如 `string`，或更複雜的資料，例如 `object`。TypeScript `interface`是表達 TypeScript 物件類型的另一種方式，而 `struct`是介面的另一種名稱。

TypeScript 不會使用 struct 一詞，但如果您在 AWS CDK API 參考中查看，您會看到 struct 實際上只是程式碼中的另一個 TypeScript 界面。API 參考也稱特定界面為界面。如果結構和界面是相同的，為什麼文件會 AWS CDK 區分它們？

AWS CDK 稱為 結構的界面代表 L2 建構所使用的任何物件。這包括在執行個體化期間傳遞給 L2 建構的屬性引數的物件類型，例如 `BucketProps` S3 儲存貯體建構`TableProps`和 DynamoDB Table 建構，以及 中使用的其他 TypeScript 介面 AWS CDK。簡而言之，如果它是 內的 TypeScript 介面 AWS CDK ，而且其名稱不是字母 的字首`I`，則 會將其 AWS CDK 呼叫為結構。

相反地， AWS CDK 使用 術語界面來表示純物件需要被視為特定建構函數或協助程式類別的適當表示的基本元素。也就是說，界面說明 L2 建構的公有屬性必須是什麼。所有 AWS CDK 界面名稱都是字母前綴的現有建構或協助程式類別的名稱`I`。所有 L2 建構模組都會擴展 `Construct`類別，但也會實作其對應的界面。因此 L2 建構會`Bucket`實作 `IBucket` 界面。

靜態方法

L2 建構的每個執行個體也是其對應界面的執行個體，但反向不是真的。這在查看結構時很重要，以查看需要哪些資料類型。如果結構具有稱為 的屬性bucket，其需要資料類型 IBucket，您可以傳遞包含IBucket介面中列出屬性的物件或 L2 的執行個體Bucket。其中一個都可以運作。不過，如果該bucket屬性針對 L2 呼叫 Bucket，則您只能在該欄位中傳遞Bucket執行個體。

當您將預先存在的資源匯入堆疊時，此區別變得非常重要。您可以為堆疊原生的任何資源建立 L2 建構，但如果您需要參考在堆疊外部建立的資源，則必須使用該 L2 建構的界面。這是因為如果 L2 建構模組不存在於該堆疊中，則建立該建構模組會建立新的資源。現有資源的參考必須是符合該 L2 建構的界面的純物件。

為了讓實務上更輕鬆，大多數 L2 建構都有一組與其相關聯的靜態方法，可傳回該 L2 建構的界面。這些靜態方法通常以 一詞開頭from。傳遞給這些方法的前兩個引數相同，scope且標準 L2 建構所需的id引數相同。不過，第三個引數不是props定義界面的一小部分屬性（有時只是一個屬性）。因此，當您傳遞 L2 建構時，在大多數情況下只需要界面的元素。這樣您就可以盡可能使用匯入的資源。

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

不過，您不應該高度依賴界面。您應該匯入資源並僅在絕對必要時直接使用界面，因為界面不提供許多屬性，例如協助程式方法，讓 L2 建構變得如此強大。

協助程式方法

L2 建構是程式設計類別，而不是簡單的物件，因此可以公開類別方法，允許您在執行個體化發生後操作資源組態。例如 AWS Identity and Access Management (IAM) L2 角色建構。下列程式碼片段顯示使用 L2 建構模組建立相同 IAM Role 角色的兩種方式。

沒有協助程式方法：

```
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' ]
            })
         ]
      })
   }
});
```

使用 協助程式方法：

```
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' ]
      })
   ]
 }));
```

在執行個體化後使用執行個體方法來操作資源組態的功能，可讓 L2 建構比上一層更多的彈性。L1 建構也繼承一些資源方法（例如 addPropertyOverride），但直到第二層取得專為該資源及其屬性設計的方法，才會發生這種情況。

列舉

CloudFormation 語法通常需要您指定許多詳細資訊，才能正確佈建資源。不過，大多數使用案例通常只涵蓋少數組態。使用一系列列舉值來代表這些組態，可以大幅減少所需的程式碼數量。

例如，在本節稍早的 S3 儲存貯體 L2 程式碼範例中，您必須使用 CloudFormation 範本的 bucketEncryption 屬性來提供所有詳細資訊，包括要使用的加密演算法名稱。反之， AWS CDK

會提供BucketEncryption列舉，採用五種最常見的儲存貯體加密形式，並可讓您使用單一變數名稱來表達每個形式。

列舉未涵蓋的邊緣案例呢？ L2 建構的其中一個目標是簡化佈建第 1 層資源的任務，因此第 2 層可能不支援較不常用的特定邊緣案例。為了支援這些邊緣案例， AWS CDK 可讓您使用 addPropertyOverride 方法直接操作基礎 CloudFormation 資源屬性。如需屬性覆寫的詳細資訊，請參閱本指南的最佳實務一節，以及 AWS CDK 文件中的抽象和逃生艙一節。

協助程式類別

有時列舉無法完成為指定使用案例設定資源所需的程式設計邏輯。在這些情況下， AWS CDK 通常會改為提供協助程式類別。列舉是提供一系列鍵值對的簡單物件，而協助程式類別則提供 TypeScript 類別的完整功能。協助程式類別仍然可以透過公開靜態屬性來充當列舉，但這些屬性可以在內部使用協助程式類別建構函數或協助程式方法中的條件邏輯來設定其值。

因此，雖然列舉可以減少在 S3 BucketEncryption 儲存貯體上設定加密演算法所需的程式碼數量，但相同的策略無法用於設定時間持續時間，因為只有太多可能的值可供選擇。為每個值建立列舉會比它更麻煩。因此，協助程式類別用於 S3 儲存貯體的預設 S3 物件鎖定組態設定，如 ObjectLockRetention 類別所示。 ObjectLockRetention包含兩種靜態方法：一種用於合規保留，另一種用於控管保留。這兩種方法都需要持續時間協助程式類別的執行個體做為引數，以表示應該設定鎖定的時間量。

另一個範例是 AWS Lambda 協助程式類別執行期。乍看之下，與此類別相關聯的靜態屬性可能會由列舉處理。不過，在幕後，每個屬性值代表Runtime類別本身的執行個體，因此在類別建構函數中執行的邏輯無法在列舉內實現。

第 3 層建構

如果 L1 建構模組將 CloudFormation 資源轉譯為程式設計程式碼，而 L2 建構模組將大部分詳細 CloudFormation 語法取代為協助程式方法和自訂邏輯，L3 建構模組會做什麼？ 的答案僅受限於您的想像。您可以建立第 3 層，以符合任何特定使用案例。如果您的專案需要具有特定屬性子集的資源，您可以建立可重複使用的 L3 建構以滿足該需求。

L3 建構稱為 內的模式 AWS CDK。模式是延伸 `Construct`類別的任何物件 AWS CDK （或延伸延伸 `Construct` 類別的類別），以執行第 2 層以外的任何抽象邏輯。當您使用 AWS CDK CLI 執行 cdk init 來啟動新 AWS CDK 專案時，您必須從三種 AWS CDK 應用程式類型中選擇：`app`、 `lib`和 `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` 和 `sample-app`都代表您在 AWS 環境中建置和部署 CloudFormation 堆疊的傳統 AWS CDK 應用程式。當您選擇 時`lib`，您選擇建置全新的 L3 建構。 `app` 並`sample-app`允許您選擇 AWS CDK 支援的任何語言，但您只能選擇 TypeScript 搭配 `lib`。這是因為 AWS CDK 以 TypeScript 原生寫入，並使用名為 JSii 的開放原始碼系統，將原始程式碼轉譯為其他支援的語言。當您選擇`lib`啟動專案時，您會選擇建置 的延伸 AWS CDK。

擴展類別的任何`Construct`類別都可以是 L3 建構，但第 3 層最常見的使用案例是資源互動、資源延伸和自訂資源。大多數 L3 建構會使用這三個案例的一個或多個來擴展 AWS CDK 功能。

資源互動

解決方案通常會採用數個可一起運作的 AWS 服務。例如，Amazon CloudFront 分佈通常會使用 S3 儲存貯體做為原始伺服器，並 AWS WAF 防止常見的入侵。 AWS AppSync 而 Amazon API Gateway 通常會使用 Amazon DynamoDB 資料表做為其 APIs的資料來源。中的管道 AWS CodePipeline 通常會使用 Amazon S3 做為其來源，並 AWS CodeBuild 用於其建置階段。在這些情況下，建立處理兩個或多個互連 L2 建構的單一 L3 建構通常很有用。 L2

以下是 L3 建構範例，其會佈建 CloudFront 分佈及其 S3 原始伺服器、 AWS WAF 要放在前面的 、Amazon Route 53 記錄和 AWS Certificate Manager (ACM) 憑證，以新增具有傳輸中加密的自訂端點，全都在一個可重複使用的建構中：

```
// 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,
          });
     }
}
```

請注意，CloudFront、Amazon S3、Route 53 和 ACM 都使用 L2 建構，但 Web ACL （定義處理 Web 請求的規則） 使用 L1 建構。這是因為 AWS CDK 是不斷發展的開放原始碼套件，尚未完全完成，而且`WebAcl`尚無 L2 建構。不過，任何人都可以 AWS CDK 透過建立新的 L2 建構來為 做出貢獻。因此，在 為 AWS CDK 提供 L2 建構之前`WebAcl`，您必須使用 L1 建構。若要使用 L3 建構 建立新的網站`CloudFrontWebsite`，請使用下列程式碼：

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

在此範例中，CloudFront `Distribution` L2 建構會公開為 L3 建構的公有屬性。在某些情況下，您仍然需要公開這類 L3 屬性。事實上，我們將在稍後的自訂資源區段中`Distribution`再次看到 。

AWS CDK 包含一些資源互動模式的範例，例如此模式。除了包含 Amazon Elastic Container Service (Amazon ECS) L2 建構的`aws-ecs`套件之外， AWS CDK 還具有名為 aws-ecs-patterns 的套件。此套件包含數個 L3 建構模組，結合 Amazon ECS 與 Application Load Balancer、Network Load Balancer 和目標群組，同時提供 Amazon Elastic Compute Cloud (Amazon EC2) 和 的預設不同版本 AWS Fargate。由於許多無伺服器應用程式只會搭配 Fargate 使用 Amazon ECS，因此這些 L3 建構提供便利性，可節省開發人員時間和客戶金錢。

資源擴充功能

有些使用案例需要資源具有非 L2 建構模組原生的特定預設設定。在堆疊層級，這可以透過使用層面來處理，但另一個提供 L2 建構新預設值的便利方法是擴展第 2 層。由於建構是繼承 類別的任何`Construct`類別，而 L2 建構會延伸該類別，因此您也可以直接延伸 L2 建構來建立 LL3 建構。

這對於支援客戶單一需求的自訂商業邏輯特別有用。假設公司有一個儲存庫，將所有 AWS Lambda 函數程式碼存放在名為 的單一目錄中，`src/lambda`而且大多數 Lambda 函數每次都會重複使用相同的執行時間和處理常式名稱。您可以建立新的 L3 建構，而不是在每次設定新的 Lambda 函數時設定程式碼路徑：

```
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
     });
}
```

然後，您可以在儲存庫中的任何位置取代 L2 `Function` 建構，如下所示：

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

預設值可讓您在單行上建立新的 Lambda 函數，並設定 L3 建構，如此您仍然可以視需要覆寫預設屬性。

當您只想將新的預設值新增至現有的 L2 建構時，擴充 L2 建構會直接運作最佳。如果您也需要其他自訂邏輯，最好擴展 Construct類別。原因來自 super方法，在建構函數中稱為 。在擴展其他類別的類別中， super方法用於呼叫父類別的建構函式，這必須是建構函式中發生的第一件事。這表示只有在建立原始 L2 建構模組之後，才能處理傳遞的引數或其他自訂邏輯。如果您需要在執行個體化 L2 建構之前執行任何此自訂邏輯，最好遵循先前在資源互動區段中概述的模式。

自訂資源

自訂資源是 CloudFormation 中的強大功能，可讓您從堆疊部署期間啟用的 Lambda 函數執行自訂邏輯。每當您在部署期間需要 CloudFormation 未直接支援的任何程序時，您可以使用自訂資源來實現。AWS CDK 提供的類別也可讓您以程式設計方式建立自訂資源。透過在 L3 建構函數中使用自訂資源，您可以讓建構幾乎沒有任何效果。

使用 Amazon CloudFront 的優點之一是其強大的全域快取功能。如果您想要手動重設該快取，讓您的網站立即反映對原始伺服器所做的新變更，您可以使用 CloudFront 無效。不過，失效是在 CloudFront 分佈上執行的程序，而不是 CloudFront 分佈的屬性。它們可以隨時建立並套用到現有的 分佈，因此它們本質上不是佈建和部署程序的一部分。

在此案例中，您可能想要在每次更新分佈的原始伺服器後建立並執行失效。由於自訂資源，您可以建立看起來像這樣的 L3 建構：

```
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')
                   }
                }
           }
        }
    }
}
```

使用先前在 `CloudFrontWebsite` L3 建構中建立的分佈，您可以非常輕鬆地執行此操作：

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

此 L3 建構模組使用稱為 AwsCustomResource 的 an AWS CDK L3 建構模組來建立執行自訂邏輯的自訂資源。當您需要只進行一次 AWS 開發套件呼叫時， `AwsCustomResource` 非常方便，因為它可讓您執行此操作，而無需撰寫任何 Lambda 程式碼。如果您有更複雜的需求，並想要實作自己的邏輯，您可以直接使用基本的 CustomResource 類別。

AWS CDK 另一個使用自訂資源 L3 建構的良好範例是 S3 儲存貯體部署。此 L3 建構的建構函數中的自訂資源所建立的 Lambda 函數會新增 CloudFormation 無法以其他方式處理的功能：它會新增和更新

S3 儲存貯體中的物件。如果沒有 S3 儲存貯體部署，您將無法將內容放入剛在堆疊中建立的 S3 儲存貯體，這會非常不方便。

AWS CDK 消除需要寫入 CloudFormation 語法範圍的最佳範例是此基本 `S3BucketDeployment`：

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

將 與您必須寫入才能完成相同操作的 CloudFormation 程式碼進行比較：

```
"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 行與 241 行是很大的差異！這只是您利用第 3 層自訂堆疊時可能的範例之一。

最佳實務

L1 建構

- 您不一定會直接避免使用 L1 建構，但您應該盡可能避免。如果特定 L2 建構不支援您的邊緣案例，您可以探索這兩個選項，而不是直接使用 L1 建構：
  - 存取 `defaultChild`： 如果 L2 建構模組中無法使用您需要的 CloudFormation 屬性，您可以使用 存取基礎 L1 建構模組`L2Construct.node.defaultChild`。您可以透過此屬性存取 L1 建構體的任何公有屬性，而不是自行建立 L1 建構體的麻煩。
  - 使用屬性覆寫：如果您要更新的屬性不是公開的？ 允許 AWS CDK 執行 CloudFormation 範本可執行的任何操作的最終逃生艙是使用每個 L1 建構中可用的方法：addPropertyOverride。您可以在 CloudFormation 範本層級操作堆疊，方法是直接將 CloudFormation 屬性名稱和值傳遞至此方法。

L2 建構

- 請記得利用 L2 建構常提供的協助程式方法。透過第 2 層，您不需要在執行個體化時傳遞每個屬性。L2 協助程式方法可以讓資源佈建更方便，特別是需要條件式邏輯時。最方便的協助程式方法之一衍生自授予類別。此類別不會直接使用，但許多 L2 建構會使用它來提供協助程式方法，讓許可更容易實作。例如，如果您想要授予 L2 Lambda 函數存取 L2 S3 儲存貯體的許可，您可以呼叫 ，`s3Bucket.grantReadWrite(lambdaFunction)`而不是建立新的角色和政策。

L3 建構

- 雖然當您想要讓堆疊更可重複使用且可自訂時，L3 建構非常方便，但建議您謹慎使用。考慮您需要哪種類型的 L3 建構，或是否需要 L3 建構：
  - 如果您不直接與 AWS 資源互動，通常更適合建立協助程式類別，而不是擴展`Construct`類別。這是因為 `Construct`類別預設會執行許多動作，只有在您直接與 AWS 資源互動時才需要這些動作。因此，如果您不需要執行這些動作，避免這些動作會更有效率。
  - 如果您判斷建立新的 L3 建構是適當的，在大多數情況下，您會想要直接擴展 `Construct`類別。只有當您想要更新建構的預設屬性時，才擴展其他 L2 建構。如果涉及其他 L2 建構模組或自訂邏輯，請`Construct`直接擴展並執行個體化建構模組中的所有資源。

常見問答集

如果 AWS CDK 不了解圖層，我無法使用 嗎？

您絕對可以。但是，與最強大的工具一樣， AWS CDK 變得越來越強大。了解 AWS CDK圖層互動如何釋放新的理解層級，這有助於簡化堆疊部署，遠遠超過您只需要基本 AWS CDK 知識就能完成的操作。

我可以使用與從 L2 建立 L3 建構相同的方式，從 L1 建立 L2 建構嗎？

如果資源已有 L2 建構，建議您使用該建構，並在第 3 層中進行自訂。這是因為許多研究已經開始找出為特定資源設定現有 L2 建構的最佳方法。不過，有數個 L1 建構的 L2 建構尚不存在。在這些情況下，我們鼓勵您建立自己的 L2 建構，並透過成為開放原始碼程式庫的 AWS CDK 貢獻者來與他人共用。您可以在 的貢獻準則中找到開始使用所需的一切 AWS CDK。

哪些 AWS 資源還沒有官方的 L2 建構？

沒有 L2 建構 AWS 的 資源數量依日減少，但如果您有興趣協助為其中一個資源建立 L2 建構，請造訪 AWS CDK API 參考。查看左側窗格中的資源清單。名稱旁具有上標 1 的資源沒有官方 L2 建構。

我可以使用 AWS CDK 支援的任何語言進行 L2 或 L3 建構嗎？

AWS CDK 支援多種程式設計語言，包括 TypeScript、JavaScript、Python、Java、C# 和 Go。您可以使用編譯為相關語言的 AWS CDK 程式碼來建立個人 L3 建構。不過，如果您想要對 做出貢獻 AWS CDK 或建立原生 AWS CDK 建構，則必須使用 TypeScript。這是因為 TypeScript 是唯一原生於 的語言 AWS CDK。其他語言的 AWS CDK 版本是使用名為 JSii 的 AWS 程式庫，從原生 TypeScript 程式碼建置而成。

哪裡可以找到 外部的現有 L3 建構 AWS CDK？

這裡有許多位置可以共用，但您可以在 AWS 解決方案建構網站和 Construct Hub 的 AWS CDK 區段中找到許多最熱門的建構。

資源

- AWS CDK API 參考
- AWS CloudFormation 資源規格
- AWS CDK 建構文件
- AWS CDK 抽象和逃生艙
- 利用 L2 建構來降低 AWS CDK 應用程式的複雜性 (AWS 部落格文章)
- AWS CloudFormation 自訂資源
- AWS 解決方案建構
- 建構中樞
- AWS CDK 範例 (GitHub 儲存庫)

文件歷史紀錄

下表描述了本指南的重大變更。如果您想收到有關未來更新的通知，可以訂閱 RSS 摘要。

變更	描述	日期
初次出版	—	2023 年 12 月 4 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。