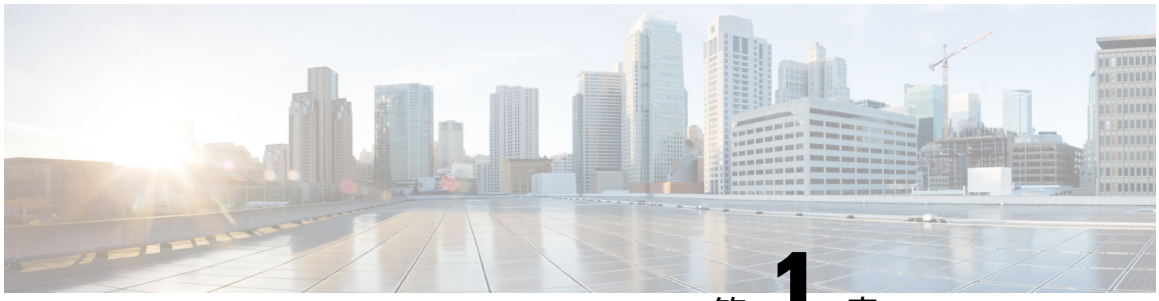
Cisco Crosswork Workflow Manager 1.0 アダプタ開発者ガイド

初版：2023 年 6 月 1 日

シスコシステムズ合同会社
〒107-6227 東京都港区赤坂9-7-1　ミッドタウン・タワー
http://www.cisco.com/jp
お問い合わせ先：シスコ コンタクトセンター
0120-092-255 （フリーコール、携帯・PHS含む）
電話受付時間：平日 10:00～12:00、13:00～17:00
http://www.cisco.com/jp/go/contactcenter/



第 1 章

概要

ここでは、次の内容について説明します。

概要

ワークフローアダプタは、ワークフローがCWM外のシステムと連携できるようにするツールです。CWM プラットフォームと任意の外部サービス間のエージェントや仲介者と考えられます。その役割は、ワークフローストリームの一部である外部システムでアクションを発生させること、またはワークフローが進行するために必要なデータを取得することです。

すべてのアダプタは、目的のターゲットサービスと通信するために開発されます。ターゲットサービスは、HTTP を介した REST API などの汎用的なものもあれば、ベンダー製品（例：シスコの Network Services Orchestrator）などの特定のものもあります。

ワークフローが 1 つ以上の外部サービスにアクセスする必要がある場合は、**アダプタ SDK** を使用して、各サービス用のカスタムアダプタを開発できます。また、CWM が提供する機能の一部として利用可能な2つの構築済みアダプタを使用することもできます。これらの既製のソリューションには、Network Services Orchestrator アダプタと汎用 REST API アダプタが含まれます。

アダプタの構成要素

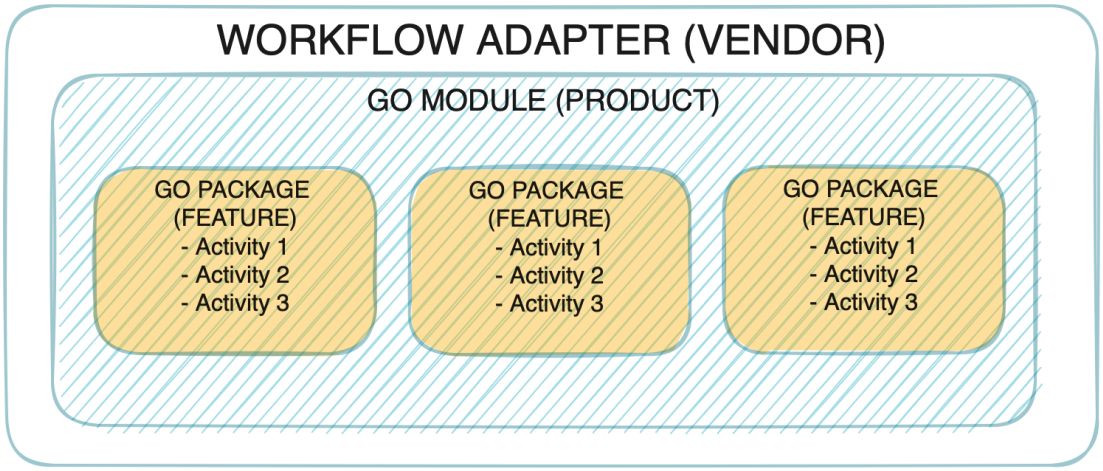
アダプタは、ワークフローアダプタ SDK を使用して開発されます。この SDK は、Golang を使用してアダプタロジックを定義し、プロトコルバッファを使用してアダプタインターフェイスを表します。

モジュール、パッケージ、アクティビティ

すべてのアダプタは、特定のベンダーによる製品を表す**go モジュール**です。**go モジュール**は、**go パッケージ**にまとめられた製品機能のコレクションです。パッケージ内で、アダプタのアク

ティビティを定義します。これは、特定の外部システム内でアダプタがトリガーできる特定のアクションです。関連するアクティビティを個別のパッケージにバンドルすることで、1 つのアダプタ内に複数の機能を含めることができます。

図 *1:* アダプタ構造



図に示すように、すべてのアダプタは、上記のモジュールとパッケージを含む標準の **go** プロジェクトレイアウトに対応するベンダー、製品、および機能の命名規則に従います。

インターフェイス

各製品機能は、`proto` フォルダにある protobuf ファイルによって表されます。アダプタ SDK には、アダプタの構造とファイルを作成するためのコマンド引数が用意されています。

前述のように、アダプタ機能の命名規則は `<vendor>.<product>.<feature>` です（例：`cisco.nso.restconf`）。

アダプタを作成すると、アダプタ SDK によって、指定したインターフェイス（機能）ごとに `.proto` ファイルが生成されます。

```
syntax = "proto3";

package <vendor>.<product>.<feature>;

option go_package = "<module>/<feature>";
```

インターフェイスは、「Activities」という名前のサービス内の RPC のリストとして定義されます。

```
service Activities {
    rpc <ActOne> (<ActOne>Request) returns (<ActOne>Response);
    rpc <ActTwo> (<ActTwo>Request) returns (<ActTwo>Response);
}
```

最後に、各アクティビティの入出力が protobuf メッセージとして定義されます。

```
message <ActOne>Request {
    ...
}
```

```
message <ActOne>Response {
    ...
}
...
```

common.adapter.proto

アダプタ インターフェイスを表す .proto ファイルの他に、<vendor>.<product>.common.adapter.proto ファイルがあります。

common .proto ファイルは、アダプタに必要な追加の構成を定義するため、また、アダプタがターゲットシステムに接続できるようにする情報を定義するために使用されます。ファイルはアダプタ SDK によって自動的に生成されますが、開発者は必要な更新を手動で行うことができます。

（注） *common* .proto ファイルでは、CWM Resource Manager がこのデータを正しく処理できるようにする、特定のメッセージを定義する必要があります。これは、ファイル内で直接実行するか（デフォルト）、別の .proto をインポートすることで実行できます。

```
// Can be defined anywhere and imported to common .proto file.
message Resource {
    ...
}
message Secret {
    ...
}

// Must be defined in common .proto file.
message Config {
    Resource resource = 1;
    Secret   secret   = 2;
}
```

アクティビティ

アダプタ SDK は、Golang に実装されるアクティビティを生成します。各アクティビティは、アダプタ構造体へのポインタであるレシーバを持つメソッドとして表されます。各メソッド定義は、proto で定義されたアクティビティ RPC に基づいています。

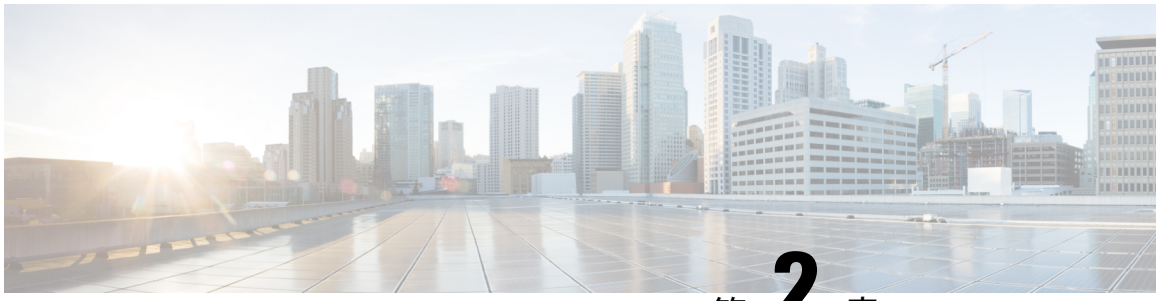
```
func (adp *Adapter) <ActivityName>(
        ctx context.Context,
        req *<ActivityName>Request,
        cfg *common.Config) (*<ActivityName>Response, error) {
    /* Activity implementation */
}
```

（注） アクティビティの実装方法に制限はありません。開発者は、使用可能な go パッケージを自由にインポートできます。アクティビティが意味のあるエラーコードを返して堅牢なエラー処理ができるようにして、パニックを回避することが推奨されます。

プロパティ

各アダプタには、アダプタに関する基本データのソースとして CWM に対して機能する `.properties` ファイルがあります。次に、必須プロパティについて例と合わせて説明します。

プロパティ	説明
author=cisco	アダプタ開発者の名前
vendor=cisco	ターゲットシステムベンダーの名前
product=nso	ターゲットシステムの名前
version=1.0.0	アダプタのバージョン
cwmsdk=1.0.0	アダプタの開発に使用される SDK のバージョン
cwmversion=1.0	互換性がある CWM バージョン
resourcetype=cisco.nso.resource.v1.0.0	CWM Resource Manager によって保存される互換性のあるリソースタイプ



第 2 章

アダプタ SDK の使用

ここでは、次の内容について説明します。

前提条件

ワークフローアダプタ SDK の使用を開始するには、**Golang** 環境、プロトコルバッファ、専用の **go** プラグインをインストールし、CWM ソフトウェアパッケージに含まれている **アダプタ SDK** をダウンロードする必要があります。

go のインストール

アダプタを開発してテストするには、**Golang** 環境をインストールする必要があります。お使いの OS 専用のインストール手順（https://grpc.io/docs/protoc-installation/）に従います。

プロトコルバッファのインストール

アダプタインターフェイスを定義して、入力パラメータと出力パラメータを生成するには、Protobufs コンパイラが必要です。お使いの OS 専用のインストール手順（https://grpc.io/docs/protoc-installation/）に従います。少なくともバージョン **3.15**（proto3）が必要であることに注意してください。

go プラグインのインストール

ステップ 1 **go** 用の追加のプロトコル コンパイラ プラグインをインストールします。

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2
```

ステップ 2 **JSON スキーマ**用のプロトコル コンパイラ プラグインをインストールします。

```
go install github.com/chrusty/protoc-gen-jsonschema/cmd/protoc-gen-jsonschema@latest
```

ステップ 3 システム PATH を更新して、protoc コンパイラがプラグインを検出できるようにします。

```
export PATH="$PATH:$(go env GOPATH)/bin
```

CWM アダプタ SDK の入手

シスコのソフトウェア ダウンロード ページに移動して、アダプタ SDK が含まれている CWM ソフトウェアパッケージをダウンロードします。

環境変数パスを設定して、cwm-sdk-binaries の場所を含めます。

```
export PATH=/path/to/cwm-sdk-binaries:$PATH
```

（注） /path/to/ を必ず実際のパスに置き換えてください。

コマンドの概要

アダプタ SDK アプリケーションには、アダプタを管理するための次の一連のコマンドが用意されています。

- `cwm-sdk create-adapter`：パッケージおよび対応する .proto ファイルを含む go モジュールを作成する場合に使用します。
- `cwm-sdk Extend-adapter`：既存のアダプタに新しい機能を追加する場合に使用します（go パッケージおよび .proto ファイル）。
- `make generate-model`：アクティビティ、入力および出力（go コード）を生成します。
- `make generate-code`：アクティビティ、入力および出力（go コード）を更新します。
- `cwm-sdk upgrade-adapter`：CWM と一致するようにアダプタをアップグレードします。
- `cwm-sdk create-installable`：CWM によってインストール可能なアーカイブを作成します。

新規アダプタの作成

アダプタを作成するには、ターミナルを開き、cwmsdk リポジトリディレクトリから次のコマンドを実行します。

```
cwm-sdk create-adapter [options] -product <product-name>
```

オプション（Options）

`create-adapter` コマンドに追加できるオプションは次のとおりです。

- -exclude-resource：テンプレートからの .resource.proto ファイルの作成をスキップします。
- -go-module *string*：go.mod ファイルに割り当てられたモジュールの名前を指定します（デフォルト："www.cisco.com/cwm/adapters/<vendor> /<adapter-name>"）。
- -feature *string*：アクティビティに割り当てられた go パッケージの名前を指定します（デフォルト："<adapter-name> "）。
- -location *string*：アダプタの場所を指します（デフォルト：現在のディレクトリ）。
- -os-architecture *string*：アダプタが開発されるアーキテクチャを定義します。有効なオプションは、「linux」、「mac-intel」、「mac-arm」、および「windows」です（デフォルト：「linux」）。
- -vendor *string*：アダプタを作成する会社の一意の名前を指定します（デフォルトは「cisco」）。
- -product string：アダプタを作成する対象の製品名に対応する go モジュールの名前を指定します（必須）。

出力

コマンドが実行されたら、新しいアダプタディレクトリ内で生成された出力を確認します。

- <adapter-name>/go/go.mod
- <adapter-name>/proto/<vendor\>.<module\>.<package\>.adapter.proto
- <adapter-name>/proto/<vendor\>.<module\>.<package\>.resource.proto（-exclude-resource オプションが使われなかった場合）
- <adapter-name>/Makefile

アダプタの機能を拡張する

アダプタの機能（**go パッケージ**）を追加するには、ターミナルを開き、cwmsdk リポジトリディレクトリから次のコマンドを実行します。

```
cwm-sdk extend-adapter [options] -feature <feature_name>
```

オプション（Options）

- -exclude-resource：テンプレートからの .resource.proto ファイルの作成をスキップします。
- -location *string*：新しいパッケージによって拡張されるアダプタの場所を指します（デフォルト：現在のディレクトリ）。

出力

コマンドが実行されたら、新しいアダプタディレクトリ内で生成された出力を確認します。

- `<adapter-name>/proto/<vendor>.<module>.<package>.adapter.proto`
- `<adapter-name>/proto/<vendor>.<module>.<package>.resource.proto`（`-exclude-resource` オプションが使用されていない場合）

入力と出力の生成

アダプタの入力ファイルと出力ファイルを生成するには、アダプタのルートディレクトリに移動して、次のコマンドを実行します。

```
make generate-model
```

出力

コマンドが実行されたら、アダプタディレクトリ内に生成された出力を確認します。

- `go/<feature\>/<vendor>.<product>.<feature>.adapter.pb.go`
- `go/common/<vendor>.<product>.common.adapter.pb.go`

`.pb.go` ファイルには、アダプタの入力パラメータと出力パラメータを定義する **go** 構造体が含まれています。手動で変更しないでください。

アクティビティの生成

以前に定義したアクティビティを生成するには、アダプタのルートディレクトリに移動して、`make generate-code` を実行します。

出力

コマンドが実行されたら、アダプタディレクトリ内に生成された出力を確認します。

- go/<package>/activities.go

`activity.go` ファイルには、`.adapter.proto` で定義された gRPC のスタブが含まれています。生成されたら、メッセージを定義してアクティビティに機能を追加できます。

アダプタのアップグレード

go モジュールをアップグレードして、go および必要なインポートに一致するバージョンを含めるには、ターミナルを開き、`cwmsdk` リポジトリディレクトリから次を実行します。

"Linux" `cwm-sdk upgrade-adapter [options]`

オプション（Options）

- `-cwm-version` *string* ：アップグレード先の CWM のバージョンを指定します（デフォルトは最新バージョンです）。
- `-location` *string*：アップグレードするアダプタの場所を指します（デフォルト：現在のディレクトリ）。

出力

- go/go.mod

`go.mod` ファイルモジュールが変更され、アダプタが正しくインストールされるようになります。

インストール可能アダプタのリリース

さまざまなオペレーティングシステム用のアダプタをインストールするためのアーカイブを作成するには、ターミナルを開き、`cwmsdk` リポジトリ ディレクトリから次のコマンドを実行します。

"Linux" `cwm-sdk create-installable [options]`

これにより、proto ファイルに基づいてコードが生成されます。

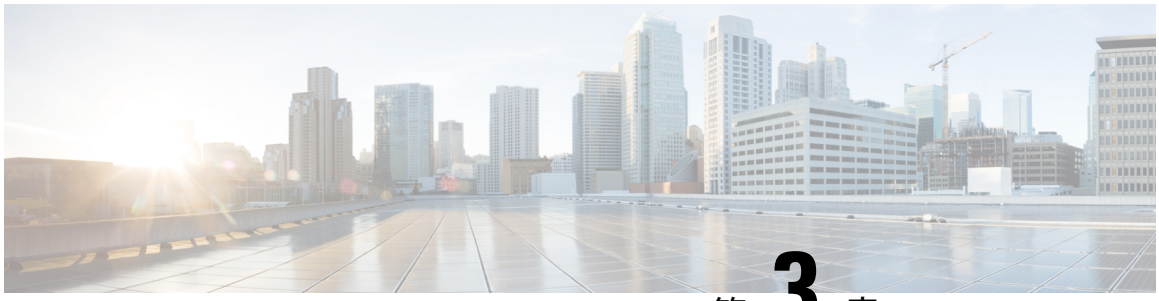
オプション（Options）

- `-location` *string*：アダプタのインストール可能ファイルの場所を指します（デフォルトは「.」）。

出力

- out/<vendor>-<product>-v<X.Y.Z>.tar.gz

生成されたアーカイブには、アダプタ go モジュールと proto ファイルが含まれています。go モジュールは、外部依存関係を持たないように、go vendor コマンドを使用して変更されます。



第 3 章

アダプタの例

ここでは、次の内容について説明します。

アダプタの例

このチュートリアルでは、ワークフローアダプタ SDK を使用してアダプタを構築する例を取り上げ、手順を説明します。アダプタ構造の概要と、ワークフローワーカーによって使用されるアダプタアクティビティを定義するための入力を提供する方法について説明します。開始する前に、「前提条件」セクション全体を確認して開発環境を設定する必要があります。

ステップ 1: 新規アダプタの作成

ターミナルウィンドウで `cwmsdk` リポジトリディレクトリを開き、次のコマンドを実行します。

```
cwm-sdk create-adapter -location ~/your_repo/adapters -vendor companyX -feature featureX
-product productX
```

これで、`adapters` に、次の内容を含む `companyX.productX` という名前のディレクトリが作成されました。

```
Makefile
adapter.properties
go
proto

 ./go:
  common
  go.mod
  featureX

  ./go/common:

  ./go/featureX:

 ./proto:
  cisco.cwm.sdk.resource.proto
  companyX.productX.common.adapter.proto
  companyX.productX.featureX.adapter.proto
```

ステップ 2: モックアクティビティの定義

アダプタ SDK は、.proto ファイルを生成しました。companyX.productX.featureX.adapter.proto ファイルで、アダプタのインターフェイスを定義します。

ステップ 1 テキストエディタまたはターミナルウィンドウ内で companyX.productX.featureX.adapter.proto ファイルを開きます。次のような内容が表示されます。

```
syntax = "proto3";

package productXfeatureX;

option go_package = "www.cisco.com/cwm/adapters/companyX/productX/featureX";

service Activities {
 // NOTE: Activity functions are defined as RPCs here e.g.

 /* Documentation for MyActivity */
 rpc MyActivity(MyRequest) returns (MyResponse);
}

// NOTE: Messages here e.g.

/* Documentation for MyRequest */
message MyRequest {
 string input = 1;
}

/* Documentation for MyResponse */
message MyResponse {
 string output = 1;
}
```

ステップ 2 アクティビティを定義するには、次に示すように、プレースホルダ「MyActivity」を、モックの「Hello」アクティビティに置き換えて、MyRequest と MyResponse のプレースホルダ名とメッセージパラメータを次のように置き換えます。

```
service Activities {
 /* Documentation for Hello Activity */
 rpc Hello(Request) returns (Response);
}

/* Documentation for Request */
message Request {
 string name = 1;
}

/* Documentation for Response */
message Response {
 string message = 1;
}
```

ステップ 3：アダプタ ソースコードの生成

ステップ 1 編集した adapter.proto ファイルと残りの .proto ファイルに基づいて、アダプタのソース **go** コードを生成し、ファイルを検査します。メインのアダプタ ディレクトリで、次のコマンドを実行します。

```
make generate-model && ls

 .go/
  common
  go.mod
  featureX

  go//common:
  companyX.productX.common.adapter.pb

  go//featureX:
  companyX.productX.featureX.adapter.pb

The `.adapter.pb.go` files generated using the **Protobufs compiler** define all the messages from
 the `adapter.proto` files.
!!! caution
 The `.adapter.pb.go` files should not be edited manually.
```

ステップ 2 次に、定義されたアクティビティの **Go** コードを生成します。メインのアダプタ ディレクトリで、次のコマンドを実行します。

```
 make generate-code && ls

 .go/
  common
  go.mod
  featureX
  main.go

  go//common:
  companyX.productX.common.adapter.pb.go

  go//featureX:
  activities.go
  adapter.go
  companyX.productX.featureX.adapter.pb.go
```

生成された activity.go ファイルには、.adapter.proto ファイルで定義したすべての RPC のスタブが含まれています。ファイルを開きます。

```
 package featureX

 import (
  "context"
  "errors"
  "go.temporal.io/sdk/activity"
 )

 func (adp *Adapter) Hello(ctx context.Context, req *Request, cfg *Config) (*Response, error) {

  activity.GetLogger(ctx).Info("Activity Hello called")

  var res *Response
  var err error

  if ctx == nil {
```

```
 return nil, errors.New("Invalid context")
}

if req == nil {
 return nil, errors.New("Invalid request")
}

if cfg == nil {
 return nil, errors.New("Invalid config")
}

cancel := ctx.Done()
done := make(chan any)

go func() {

 //
 // NOTE:
 //
 // Enter activity code to set response and error here...
 //
 // Perform step 1
 //
 // ...
 //
 // activity.activity.RecordHeartbeat(ctx, "Activity completed step 1")
 //
 // Perform step 2
 //
 // ...
 //
 // activity.activity.RecordHeartbeat(ctx, "Activity completed step 2")
 //
 // ...
 //
 // All logic steps are completed
 //

 done <- nil
}()

//
// NOTE
//
// For a long running call heartbeats can be recorded in a separate
//
// go func () {
//     for {
//         activity.RecordHeartbeat(ctx, "Activity is running")
//         // TODO sleep for some interval
//     }
// } ()
//

for {
 select {
 case <-cancel:

  //
  // NOTE
  //
  // Execute any cleanup required for a canceled activity here...
  //
```

```
    return nil, errors.New("Activity was canceled")
   case <-done:
    return res, err
   }
  }
 }
```

ステップ 3 メッセージを返すようにファイルを編集します。

```
go func() {

 res = &Response {Message: "Hello, " + req.GetName() + "!"}
 err = nil

 done <- nil
}()
```

別のアクティビティの定義

既存の機能セット（**go**パッケージ）に別のアクティビティを追加する場合は、次の手順を実行します。

ステップ 1 `adapter.proto`ファイルを開いて編集し、既存のアクティビティの下に別のアクティビティを定義します。

```
service Activities {
 rpc Hello(Request) returns (Response);
 rpc Fancy(Request) returns (Response);
}
```

ステップ 2 SDK を使用してアクティビティ go コードを更新します。

```
make generate-code
```

コードが生成されると、`activity.go`ファイルは新しい「Fancy」アクティビティスタブで更新されますが、「Hello」アクティビティのコードは残ります。

ステップ 4：別の機能の追加

アダプタの例に別の機能（**go**パッケージ）を追加する場合は、`extend-adapter`コマンドを使用します。ターミナルで`cwmsdk`リポジトリディレクトリを開き、次のコマンドを実行します。

```
cwm-sdk extend-adapter -feature featureY
```

ステップ 1 新しい`companyX.productX.featureY.adapter.proto`ファイルがアダプタに追加されました。

```
.proto/
  cisco.cwm.sdk.resource.proto
  companyX.productX.common.adapter.proto
  companyX.productX.featureY.adapter.proto
  companyX.productX.featureX.adapter.proto
```

ステップ 2 新しい機能のアクティビティを定義するには、`companyX.productX.featureY.adapter.proto` ファイルを開き、それに応じて内容を変更します。

```
syntax = "proto3";

package companyXproductX;

option go_package = "www.cisco.com/cwm/adapters/companyX/productX/featureY";

service Activities {
 /* Documentation for Goodbye Activity */
 rpc Goodbye(Request) returns (Response);
}

/* Documentation for Request */
message Request {
 string name = 1;
}

/* Documentation for Response */
message Response {
 string message = 1;
}
```

ステップ 3 「featureY」パッケージとアクティビティのコードを生成します。

```
make generate-model && generate-code && ls

.go/goodbyes
 activities.go
 adapter.go
 companyX.productX.featureY.adapter.pb.go
```

ステップ 5: インストール可能アーカイブの作成

```
cwm-sdk create-installable
```

生成されたアーカイブには、アダプタに必要なすべてのファイルが含まれています。外部のすべての依存関係を排除するために、**go** vendor コマンドが実行されています。

翻訳について
このドキュメントは、米国シスコ発行ドキュメントの参考和訳です。リンク情報につきましては、日本語版掲載時点で、英語版にアップデートがあり、リンク先のページが移動/変更されている場合がありますことをご了承ください。あくまでも参考和訳となりますので、正式な内容については米国サイトのドキュメントを参照ください。