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。