タスクの管理

この章は、次の項で構成されています。

タスク

ワークフロー タスクは、Cisco UCS Director が保持するタスク ライブラリへの提供に必要なアーティファクトを提供します。タスクは、ワークフロー定義で使用できます。

タスクには少なくとも次のクラスが必要です。

- TaskConfigIf インターフェイスを実装するクラス。
- AbstractTask クラスのメソッドを拡張および実装するクラス。

TaskConfigIf

このインターフェイスを実装するクラスが、タスクの入力になります。つまり、タスクの実行のために入力を受け入れる必要のあるタスクは、**TaskConfigIf** を実装するクラスに依存することになります。このインターフェイスを実装するクラスには、ユーザにプロンプトを表示するための、適切に注釈を付けたすべての入力フィールド定義も含める必要があります。このクラスには、プラットフォーム ランタイムがデータベースでこのオブジェクトの持続性を維持できるようにする JDO 注釈も必要です。

サンプル コードに、サンプルの Config クラスを示しています。

AbstractTask

タスクの実装は **AbstractTask** 抽象クラスを拡張し、すべての抽象メソッドに対する実装を提供する必要があります。これは、タスクに関連するすべてのビジネスロジックを処理するメイン クラスです。ビジネス ロジックの実装が記述されるこのクラスで最も重要なメソッドは、**executeCustomAction( )** です。その他のメソッドは、プラットフォーム ランタイムでタスクをオーケストレーション デザイナ ツリーに表示したり、ワークフロー内でタスクのドラッグ アンド ドロップを可能にしたりするための十分なコンテキストを提供します。

TaskConfigIf の開発

タスクを開発するには、最初に **TaskConfigIf** を実装する必要があります。タスク設定インターフェイスの設定プロセス中、タスクの実行に必要なデータを決定します。

次の例では、**EnableSNMPConfig** が **TaskConfigIf** の開発プロセスの詳細情報を公開しています。**Enable SNMP** タスクは、Cisco Nexus デバイスで SNMP を有効にするように設計されています。

処理を進めるには、Nexus デバイスの IP アドレス、ログイン、パスワードが必要です。

EnableSNMPConfig の先頭に注釈が付いています。

```
@PersistenceCapable(detachable= "true", table = "foo_enable_snmp_config")
public class EnableSNMPConfig implements TaskConfigIf
{
```

`PersistenceCapable` の注釈には、モジュール ID のプレフィックスが付いたテーブル名を指定する必要があります。モジュール ID のプレフィックスが付いていないテーブル名を使用しようとすると、Cisco UCS Director によってタスクの登録が妨げられるため、この規則には必ず従う必要があります。

続いて、次のフィールドを見ていきます。

- **handler name**
- **configEntryId**
- **actionId**

```
public static finald String HANDLER_NAME = "Enable SNMP for Nexus";

//configEntryId and actionId are mandatory fields
@Persistent
private long   configEntryId
@Persistent
private long   actionId
```

handler name はタスクの名前です。この名前は必ず一意の文字列にします。複数のタスクで同じハンドラ名を使用すると、問題が生じることになります。

上の例に示しているとおり、各タスクには、**configEntryId** と **actionId** が必要です。これらの 2 つのフィールドに対応する getter と setter を定義する必要があります。これらの 2 つのフィールドは不可欠です。これらのフィールドは config オブジェクトに設定する必要があります。

次に、タスクを実行する上で実際に必要なデータを見ていきます。

```
//This is the ip address for the Nexus device on which you want to enable SNMP.
@FormField(label = "Host IP Address", help = "Host AP Address", mandatory = true,
     type = FormFieldDefinition.FIELD_TYPE_EMBEDDED_LOV,
     lovProvider = ModuleConstants.NEXUS_DEVICES_LOV_PROVIDER)
@USerInputField(type = ModuleConstants.NEXUS_DEVICE_LIST)
@Peristent
private String   ipAddress   = "";

@FormField(label = "Login", help = "Login", mandatory = true
@Persistent
private String   login;

@FormField(label = "Password", help = "Password", mandatory = true
@Persistent
private String   password;
```

上のコード サンプルを確認して、開発者が次の項目を含める必要があることに注意します。

- デバイスの IP アドレス。

  この例では LOV を使用して、この IP アドレスを取得しています。注釈および LOV の詳細については、注釈 を参照してください。

- ログインおよびパスワードは、ユーザが入力する必要があります。

  これらを取得するには、フォーム フィールドの注釈を使用して、ユーザが入力するデータとしてこれらのフィールドをマーキングします。

- これらの各フィールドの getter および setter。

config オブジェクトの定義が終了したら、Java Data Object（JDO）の拡張としてマーキングします。

始める前に

Cisco UCS Director Open Automation ソフトウェア開発キット（SDK）が必要になります。

ステップ 1 jdo.files ファイルを config オブジェクトと同じパッケージに含めます。

SDK のサンプルを jdo.files とパッケージ化の参考にしてください。jdo.files は、このとおりの名前にしてください。

ステップ 2 jdo.files に、JDO の拡張を介して処理する必要のあるすべてのクラスを指定します。
このステップを正しく実行していれば、SDK が提供するビルド スクリプトによって JDO の拡張がすべて処理されます。

次のタスク

ハンドラ オブジェクトは、カスタム コードを実際に実行する場所です。ハンドラ オブジェクトは **AbstractTask** を実装する必要があります。`executeCustomAction` メソッドを使用して、コードの実行のために以前に開発した、対応する config オブジェクトを取得できます。

抽象タスクの開発

config オブジェクトの準備ができたら、AbstractTask を拡張して、新しい config オブジェクトを実際に使用できるようにする必要があります。この例では、EnableSNMPTask を示しています。

ここで、次のメソッド executeCustomAction について見てみます。

```
public void executeCustomAction(CustomActionTriggerContext context, CustomActionLogger
actionLogger) throws Exception
{
   long configEntryId = context.getConfigEntry().getConfigEntryID();
   //retrieving the corresponding config object for this handler
   EnableSNMPConfig config = (EnableSNMPConfig) context.loadConfigObject();
```

executeCustomAction は、カスタム ロジックが処理される場所です。context.loadConfigObject() をコールする際、以前に定義した config オブジェクトにキャストできます。この処理により、タスクの実行に必要なすべての詳細情報を取得するできるようになります。この例では、config オブジェクトを取得した後、SSH API を使用して、enable SNMP コマンドを実行しています。

ワークフローをロールバックする場合、行った変更を元に戻す方法をタスクが指定する必要があります。この例では、変更トラッカーを使用しています。

```
//If the user decides to roll back a workflow containing this task,
//then using the change tracker, we can take care of rolling back this task (i.e.,
//disabling snmp)
context.getChangeTracker().undoableResourceAdded("assetType", "idString",
SNMP enabled", "SNMP enabled on " + config.getIpAddress(),
new DisableSNMPNexusTask().getTaskName(), new DisableSNMPNexusConfig(config));
```

ロールバック コードはシステムに、Enable SNMP タスクの undo タスクが Disable SNMP タスクであることを伝えます。undo config オブジェクトとその名前を指定してください。その他の引数はデータのロギングを行いますが、指定してもしなくても構いません。

DisableConfig DisableConfig は、実際は **EnableConfig** で実行されます。 この場合、enable config にデバイスの詳細情報が含まれているため、Disable SNMP タスクが呼び出されたとき、どのデバイスで disable SNMP を実行するのかを正確に把握できます。

また、getTaskConfigImplementation も実装する必要があります。この例では、config オブジェクトのインスタンスを返す際にインスタンス化しています。

```
@Override
public TaskConfigIf getTaskConfigImplementation() {
   return new EnbleSNMPConfig();
}
```

（注） このタスクで使用する config オブジェクトを必ず指定してください。

次の作業：このタスクをモジュールに含めて、Cisco UCS Director で使用できるようにします。

スケジュール タスクについて

消去タスクや集約タスク、あるいは繰り返し実行可能な何らかのタスクを開発する必要がある場合は、スケジュール タスク フレームワークを使用できます。これには、次のようなコンポーネントが含まれています。

- **AbstractScheduleTask**
- **AbstractCloupiaModule**

AbstractScheduleTask

タスク ロジックを、このクラスの `execute()` メソッドに含める必要があります。モジュール ID と、このタスクを説明する文字列を指定することから始めます。固有のモジュール ID を指定する必要があります。指定しないと、モジュールが正しく登録されません。

詳細については、foo モジュールの DummyScheduleTask クラスを参照してください。

```
public DummyScheduleTask(){
super("foo");
}
```

スケジュール タスクの追加/削除

AbstractCloupiaModule AbstractCloupiaModule には、スケジュール タスクを追加および削除する API が含まれています。一般に、**AbstractCloupiaModule** の `onStart()` 実装でタスクをインスタンス化し、次のようにモジュール クラス内で addScheduleTask メソッドを呼び出すことによって add メソッドでインスタンス化したタスクを登録します。

```
addScheduleTask(new DummyScheduleTask());
```

詳細については、FooModule.java クラスを参照してください。

カスタム ワークフローの入力の登録

独自の入力タイプを Cisco UCS Director で開発できます。詳細については、『*Cisco UCS Director Orchestration Guide, Release 4.1*』を参照してください。ただし、これらの入力タイプには、モジュール ID のプレフィックスを付ける必要があります。TaskConfigIf の開発 （2 ページ） を参照してください。ここでは、追加の注釈を使用してカスタム ワークフローの入力を指定しています。

```
public static final String NEXUS_DEVICE_LIST = "foo_nexus_device_list";
@UserInputField(type = ModuleConstants.NEXUS_DEVICE_LIST)
```

この例では、ModuleConstants.NEXUS_DEVICE_LIST が `foo_nexus_device_list` に解決されています。

始める前に

カスタム ワークフローに必要な TaskConfigIf および AbstractTask コンポーネントを開発します。

次のタスク

カスタム ワークフローの出力を登録します。カスタム タスクの出力の登録 （6 ページ）を参照してください。

カスタム タスクの出力の登録

出力を追加するタスクを有効にできます。

始める前に

カスタム タスクの出力を作成する方法の例については、EmailDatacentersTask を参照してください。

手順の概要

1. タスクの実装にメソッド `getTaskoutputDefinitions()` を実装し、タスクが返す出力定義を返します。
2. タスクの実装からの出力を設定します。

手順の詳細

ステップ 1 タスクの実装にメソッド `getTaskoutputDefinitions()` を実装し、タスクが返す出力定義を返します。

```
@Override
public TaskOutputDefinition[] getTaskOutputDefinitions() {
   TaskOutputDefinition[] ops = new TaskOututDefiniton[1];
   ops[0] = FooModule.OP_TEMP_EMAIL_ADDRESS;
   return ops;
]
```

ステップ 2 タスクの実装からの出力を設定します。

```
@Override
public void executeCustomAction(CustomActionTriggerContext context,
CustomerActionLogger action Logger) throws Exception
{

long configEntryId = context.getConfigEntry().getConfigEntryId();
//retrieving the corresponding config object for this handler
EmailDatacentersConfig config = (EmailDatacentersConfig context.loadConfigObject();

if (config == null)
{
    throw net Exception("No email configuration found for custom Action"
        + context.getAction().getName
        + "entryId" + configEntryId);
}
```

```
|.......
 .......

try
{
    context.saveOutputValue(OutPutConstants.OUTPUT_TEMP_EMAIL_ADDRESS, toAddresses);
```

他のタスクの入力として使用するカスタム出力

この項では、出力を別のタスクの入力として使用する方法について説明します。前の項で示した例の一部をここでも使用します。出力の定義は、次のように定義されます。

```
@Override
public TaskOutputDefinition[] get TaskOutputDefinitions() {
    TaskOutputDefinition[] ops = new TaskOutputDefinitions[1};
//NOTE: If you want to use the output of this task as input to another task. Then the
second argument
//of the output definition MUST MATCH the type of UserInputField in the config of the
task that will
//be recieving this output. Take a look at the HelloWorldConfig as an example.
ops[0] = new TaskOutputDefinition(
      FooConstants.EMAIL_TASK_OUTPUT_NAME,
      FooConstants.FOO_HELLO_WORLD_NAME,
      "EMAIL IDs");
return ops;
}
,
```

この例では、FooConstants.EMAIL_TASK_OUTPUT_NAME という名前と、FooConstants.FOO_HELLO_WORLD_NAME という型を使用して、出力を定義しています。出力を入力として使用できる別のタスクを設定するには、型を一致させる必要があります。

このため、FooConstants.FOO_HELLO_WORLD_NAME を入力として使用する新規タスクで、設定オブジェクトに次の記述を含める必要があります。

```
//This field is supposed to consume output from the EmailDatacentersTask.
//You'll see the type in user input field below matches the output type
//in EmailDatacentersTasks's output definition.
@FormField(label = "name", help = "Name passed in from a previous task", mandatory =
true)
@UserInputField(type = FooConstants.FOO_HELLO_WORLD_NAME)
@Persistent
private String     login;
```

`UserInputField`注釈内の型は、出力定義に登録されている型と一致しています。このように型が一致している場合に、ワークフローの開発中に Cisco UCS Director の**Workflow Designer**で新規タスクをドラッグアンドドロップすると、あるタスクの出力を別のタスクの入力としてマップできます。

入力として使用する既存のタスクの出力

この項では、組み込みワークフロー タスクの出力を、カスタム タスクの入力として使用する方法を示します。このプロセスは、1 つの重要な点で、カスタム出力を入力として使用できるように設定する場合と似ています。それは、タスクの設定オブジェクトに、出力と全く同じ型のフィールドが必要であるという点です。

手順の概要

1. [ポリシー（Policies）] > [オーケストレーション（Orchestration）] > [ワークフロー（Workflows）] の順に選択し、[タスクライブラリ（Task Library）] をクリックします。
2. 追加するタスクを見つけたら、それを選択して、[ユーザとグループのタスク：グループの追加（User and Group Tasks: Add Group）] という見出しの下に表示される情報を確認します。
3. [出力（Outputs）] テーブルから、該当する型の値を取得します。
4. UserInputField に型の値を指定します。
5. ワークフローの開発時、ワークフローにアクションを追加する際や、ワークフローの関連情報を編集する際に、[タスク入力属性へのユーザ入力のマッピング（User Input Mapping to Task Input Attributes）] ウィンドウを使用して、マッピングを設定します。

手順の詳細

ステップ 1 [ポリシー（Policies）] > [オーケストレーション（Orchestration）] > [ワークフロー（Workflows）] の順に選択し、[タスクライブラリ（Task Library）] をクリックします。

ヒント **Cntl–Find** を押して、表示される非常に長いリストでタスクを検索します。たとえば、「Group」と入力すると、すぐに [ユーザとグループのタスク（User and Group Tasks）] に移動できます。

ステップ 2 追加するタスクを見つけたら、それを選択して、[ユーザとグループのタスク：グループの追加（User and Group Tasks: Add Group）] という見出しの下に表示される情報を確認します。

ヒント **Cntl–Find** を押して、表示される非常に長いリストでタスクを検索します。たとえば、「Group」と入力すると、すぐに [ユーザとグループのタスク（User and Group Tasks）] に移動できます。

重要な型のデータが、見出しの下方の最後にある [出力（Outputs）] テーブルに示されています。

表 *1 : [グループの追加（Add Group）] - [出力（Outputs）] テーブル*

出力	説明	タイプ
OUTPUT_GROUP_NAM	管理者が作成したグループの名前	gen_text_input
OUTPUT_GROUP_ID	管理者が作成したグループの ID	gen_text_input

ステップ 3 [出力（Outputs）] テーブルから、該当する型の値を取得します。

目的は、タスクと一致する型の値を取得することです。この例では、タスクはグループ ID を使用するため、型は *gen_text_input* であることがわかります。

ステップ 4 UserInputField に型の値を指定します。

例：

```
@FormField(label = "Name", help = "Name passed in from previous task",
mandatory = true)
@UserInputField(type="gen_text_input")
@Persistent
private String       name;
```

（注） また、*@UserInputField(type = WorkflowInputFieldTypeDeclaration.GENERIC_TEXT)* を使用することもできます。これは、*@UserInputField(type="gen_text_input")* を使用することと同じです。SDK に定義されている定数を使用する *type = WorkflowInputFieldTypeDeclaration.GENERIC_TEXT* を使用する方が簡単なこともあります。

最後のステップは、ワークフローを開発する際に、正しくマッピングを設定することです。

ステップ 5 ワークフローの開発時、ワークフローにアクションを追加する際や、ワークフローの関連情報を編集する際に、[タスク入力属性へのユーザ入力のマッピング（User Input Mapping to Task Input Attributes）] ウィンドウを使用して、マッピングを設定します。

カスタム タスクが存在することの確認

モジュールが正常に動作している場合、Cisco UCS Director タスク ライブラリを開き、タスクが表示されていることを確認することで、カスタム タスクの存在を確認できます。

手順の概要

1. Cisco UCS Director で **[ポリシー（Policies）]** > **[オーケストレーション（Orchestration）]** を選択し、続いて [ワークフロー（Workflows）] タブを選択します。
2. [ワークフロー（Workflows）]**Workflows**ツリー ディレクトリで、タスクが表示されるワークフローに移動して、そのワークフローの行を選択します。
3. ワークフローを選択して、[ワークフローデザイナ（Workflow Designer）] をクリックします。
4. 使用可能なタスクのリスト、およびワークフロー内のタスクのグラフィック表示に、目的のタスクが表示されていることを確認します。

手順の詳細

ステップ 1 Cisco UCS Director で **[ポリシー（Policies）]** > **[オーケストレーション（Orchestration）]** を選択し、続いて [ワークフロー（Workflows）] タブを選択します。

[ワークフロー（Workflows）] タブには、使用可能なすべてのワークフローを一覧したテーブルが表示されます。

ステップ 2 [ワークフロー（Workflows）]**Workflows**ツリー ディレクトリで、タスクが表示されるワークフローに移動して、そのワークフローの行を選択します。

移動を容易にするために、テーブルの上部の右上隅にある [検索（Search）] オプションを使用して、ワークフローに移動します。

ワークフロー テーブルの上部に、ワークフローに関連するその他のコントロールが表示されます。

ステップ 3 ワークフローを選択して、[ワークフローデザイナ（Workflow Designer）] をクリックします。

[ワークフローデザイナ（Workflow Designer）] 画面が開き、[使用可能なタスク（Available Tasks）] リストおよびワークフロー設計グラフィック ビューが表示されます。

ステップ 4 使用可能なタスクのリスト、およびワークフロー内のタスクのグラフィック表示に、目的のタスクが表示されていることを確認します。