

VMware vRealize Orchestrator を使用した 開発

vRealize Orchestrator 7.4



vmware®

最新の技術ドキュメントは VMware の Web サイト (<https://docs.vmware.com/jp/>) にあります
このドキュメントに関するご意見および感想がある場合は、docfeedback@vmware.com までお送りください。

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

ヴィエムウェア株式会社
105-0013 東京都港区浜松町 1-30-5
浜松町スクエア 13F
www.vmware.com/jp

Copyright © 2008–2018 VMware, Inc. 無断転載を禁ず。 [著作権および商標情報](#)。

目次

VMware vRealize Orchestrator を使用した開発 10

1 ワークフローの開発 11

- ワークフローの主要な概念 13
 - ワークフローのパラメータ 13
 - ワークフローの属性 14
 - ワークフローのスキーマ 14
 - ワークフロー プレゼンテーション 14
 - ワークフロー トークン 14
- ワークフロー開発プロセスのフェーズ 15
- ワークフローの開発のベスト プラクティス 15
- Orchestrator クライアントのアクセス権限 16
- 開発時のワークフローのテスト 16
- ワークフローの作成および編集 16
 - ワークフローの作成 17
 - ワークフローの編集 17
 - 標準ライブラリのワークフローの編集 17
 - ワークフロー エディタのタブ 18
- ワークフローの全般情報の指定 19
- 属性とパラメータの定義 20
 - ワークフローのパラメータの定義 21
 - ワークフローの属性の定義 22
 - 属性名とパラメータ名に関する制約 22
- ワークフローのスキーマ 23
 - ワークフローのスキーマの表示 24
 - ワークフロー スキーマでのワークフローのビルド 25
 - スキーマ要素 28
 - スキーマ要素のプロパティ 31
 - リンクとバインド 34
 - 決定 40
 - 例外処理 43
 - エラー ハンドラの使用 44
 - Foreach 要素および複合タイプ 45
 - 切り替えアクティビティをワークフローに追加する 48
- プラグインの開発 49
 - プラグインの概要 49
 - プラグインの内容と構造 58
 - Orchestrator プラグイン API リファレンス 64

vso.xml プラグイン定義ファイルの要素	76
Orchestrator プラグインを開発する場合のベスト プラクティス	95
ワークフロー開始時のユーザーからの入力パラメータの取得	109
[プレゼンテーション] タブでの入力パラメータ ダイアログ ボックスの作成	109
パラメータのプロパティの設定	111
ワークフローの実行中にユーザー操作を要求する	113
ワークフローへのユーザー操作の追加	114
ユーザー操作の security.group 属性の設定	115
timeout.date 属性の絶対日時の設定	116
ユーザー操作の相対タイムアウトの計算	117
timeout.date 属性の相対日時の設定	118
ユーザー操作の外部入力の定義	119
ユーザー操作の例外の動作を定義する	120
ユーザー操作の入力パラメータ ダイアログ ボックスの作成	121
ユーザー操作の要求への応答	122
ワークフロー内でのワークフローの呼び出し	123
ワークフローを呼び出すワークフロー要素	124
ワークフローの同期呼び出し	126
ワークフローの非同期呼び出し	127
ワークフローのスケジュール設定	128
リモートワークフローを他のワークフロー内から呼び出すための前提条件	129
複数のワークフローの同時呼び出し	130
選択したオブジェクトに対するワークフローの実行	131
「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローの実装	132
長期実行ワークフローの開発	134
タイマーに基づいたワークフローでの相対日時の設定	134
タイマーベースの長期実行ワークフローの作成	136
トリガ オブジェクトの作成	137
トリガベースの長時間ワークフローの作成	139
構成要素	140
構成要素の作成	140
ワークフローのユーザー権限	141
ワークフローに対するユーザー権限の設定	142
ワークフローの検証	142
ワークフローの検証と検証エラーの修正	143
ワークフローのデバッグ	144
ワークフローのデバッグ	144
ワークフローのデバッグ例	145
実行中のワークフロー	146
ワークフロー エディタでのワークフローの実行	147
ワークフローの実行	147

失敗したワークフローの実行の再開	149
失敗したワークフローの実行を再開するための動作の設定	149
失敗したワークフローの実行を再開するためのカスタム プロパティの設定	149
失敗したワークフローの実行の再開	150
ワークフロー ドキュメントの生成	150
ワークフロー バージョン履歴の使用	151
削除済みワークフローのリストア	152
単純なサンプル ワークフローの開発	152
単純なワークフロー サンプルの作成	154
単純なワークフロー サンプルのスキーマの作成	155
単純なワークフロー サンプルのゾーンの作成	157
単純なワークフロー サンプルのパラメータの定義	159
単純なワークフロー サンプルの決定バインドの定義	160
単純なワークフローでのアクション要素のバインド例	160
単純なワークフローでのスクリプト化されたタスク要素のバインド例	164
単純なワークフロー サンプルの例外バインドの定義	172
単純なワークフロー サンプルの属性の読み書きプロパティの設定	173
単純なワークフロー サンプルのパラメータのプロパティの設定	174
単純なワークフロー サンプルの入力パラメータ ダイアログ ボックスのレイアウトの設定	175
単純なワークフロー サンプルの検証と実行	177
複合ワークフローの開発	178
複雑なワークフロー サンプルの作成	179
複雑なワークフローサンプル向けのカスタム アクションの作成	180
複雑なワークフロー サンプルのスキーマの作成	181
複雑なワークフロー サンプルのゾーンの作成	183
複雑なワークフロー サンプルのパラメータの定義	185
複雑なワークフロー サンプルのバインドの定義	186
複雑なワークフロー サンプルの属性プロパティの設定	196
複雑なワークフロー サンプルの入力パラメータのレイアウト作成	197
複雑なワークフロー サンプルの検証と実行	198

2 スクリプティング 200

スクリプティングを必要とする Orchestrator 要素	200
Orchestrator での Mozilla Rhino 実装の制限事項	201
Orchestrator Scripting API の使用	202
ワークフロー エディタからのスクリプト エンジンへのアクセス	203
アクション エディタまたはポリシー エディタからのスクリプト エンジンへのアクセス	203
Orchestrator API Explorer へのアクセス	204
Orchestrator API Explorer によるオブジェクトの検索	204
スクリプトの記述	205
スクリプトへのパラメータの追加	207
JavaScript とワークフローから Orchestrator サーバ ファイル システムにアクセスする	207

	JavaScript から Java クラスにアクセスする	208
	JavaScript からオペレーティングシステム コマンドにアクセスする	208
	vCenter Server プラグインを使用した XPath 式の使用	209
	vCenter Server プラグインを使用した XPath 式の使用	209
	例外処理のガイドライン	210
	Orchestrator の JavaScript サンプル	211
	スクリプティングの基本例	212
	E メール スクリプトの例	214
	ファイル システム スクリプトの例	215
	LDAP スクリプティング サンプル	216
	ログ スクリプティング サンプル	216
	ネットワーク スクリプティング サンプル	217
	ワークフローのスクリプティング サンプル	217
3	アクションの開発	219
	アクションの再利用	219
	アクション ビューへのアクセス	220
	アクション ビューのコンポーネント	220
	アクションの作成	220
	アクションの作成	221
	アクションを実装する要素の検出	222
	アクションのコーディング ガイドライン	222
	アクション バージョン履歴の使用	223
	削除済みアクションのリストア	224
4	リソース要素の作成	225
	リソース要素の表示	225
	リソース要素として使用する外部オブジェクトのインポート	226
	リソース要素の情報とアクセス権限の表示	226
	リソース要素のファイルへの保存	227
	リソース要素の更新	228
	リソース要素をワークフローに追加する	228
5	パッケージの作成	230
	パッケージの作成	230
	パッケージに対するユーザー権限の設定	231
6	プラグインの開発	233
	プラグインの概要	233
	Orchestrator プラグインの構造	234
	外部の API を Orchestrator に公開する	235
	プラグインのコンポーネント	236

vso.xml ファイルの役割	237
プラグイン アダプタのロール	238
プラグイン ファクトリのロール	239
ファインダ オブジェクトの役割	240
スクリプト オブジェクトの役割	241
イベント ハンドラの役割	241
プラグインの内容と構造	242
vso.xml ファイル内でのアプリケーション マッピングの定義	243
vso.xml プラグイン定義ファイルの形式	244
プラグイン オブジェクトの命名	245
プラグイン オブジェクトの命名規則	246
プラグインのファイル構造	247
Orchestrator プラグイン API リファレンス	248
IAop インターフェイス	248
IDynamicFinder インターフェイス	249
IPluginAdaptor インターフェイス	249
IPluginEventPublisher インターフェイス	250
IPluginFactory インターフェイス	251
IPluginNotificationHandler インターフェイス	252
IPluginPublisher インターフェイス	253
WebConfigurationAdaptor インターフェイス	253
PluginTrigger クラス	254
PluginWatcher クラス	255
QueryResult クラス	255
SDKFinderProperty クラス	256
PluginExecutionException クラス	258
PluginOperationException クラス	258
HasChildrenResult 列挙	258
ScriptingAttribute 注釈タイプ	259
ScriptingFunction 注釈タイプ	260
ScriptingParameter 注釈タイプ	260
vso.xml プラグイン定義ファイルの要素	260
module 要素	260
description 要素	261
廃止された要素	262
url 要素	262
installation 要素	262
action 要素	263
finder-datasource 要素	263
finder-datasource 要素	264
inventory 要素	265
finders 要素	265

finder 要素	266
properties 要素	267
property 要素	267
relations 要素	268
relation 要素	269
id 要素	269
inventory-children 要素	269
relation-link 要素	270
events 要素	270
トリガ要素	270
trigger-properties 要素	271
trigger-property 要素	271
gauge 要素	271
scripting-objects 要素	272
object 要素	272
constructors 要素	273
constructor 要素	273
コンストラクタの parameters 要素	273
コンストラクタの parameter 要素	274
attributes 要素	274
attribute 要素	274
methods 要素	275
method 要素	276
example 要素	276
code 要素	277
メソッドの parameters 要素	277
メソッドの parameter 要素	277
singleton 要素	278
enumerations 要素	278
enumeration 要素	278
entries 要素	279
entry 要素	279
Orchestrator プラグインを開発する場合のベスト プラクティス	280
Orchestrator プラグインのビルド方法	280
Orchestrator プラグインのタイプ	282
プラグインの実装	285
Orchestrator プラグイン開発の推奨事項	290
プラグインと API ドキュメントの ユーザー インターフェイス文字列	292

7 Maven を使用したプラグインの作成 295

Maven を使用してアーキタイプから Orchestrator プラグインを作成する	295
Maven のアーキタイプ	296

Maven ベースのプラグイン開発のベスト プラクティス 296

VMware vRealize Orchestrator を使用した開発

この「VMware vRealize Orchestrator を使用した開発」では、VMware[®] vRealize Orchestrator のカスタムのワークフローとアクションを開発する方法について説明します。

また、このドキュメントには、スクリプトを必要とする Orchestrator 要素に関する情報と、サンプルの JavaScript も記載されています。この「VMware vRealize Orchestrator を使用した開発」では、リソースとパッケージの作成方法についても説明します。

対象者

このドキュメントには、Orchestrator のカスタムのワークフロー、アクション、ビルディング ブロックを作成する開発者向けの情報が記載されています。

ワークフローの開発

ワークフローは Orchestrator クライアント インターフェイスで開発します。ワークフロー開発には、ワークフロー エディタ、組み込みの Mozilla Rhino JavaScript スクリプト エンジン、Orchestrator および vCenter Server API を使用します。

■ ワークフローの主要な概念

ワークフローは、スキーマ、属性、およびパラメータで構成されています。ワークフロー スキーマは、すべてのワークフロー要素および要素間の論理接続を定義する、ワークフローの主なコンポーネントです。ワークフローの属性およびパラメータは、ワークフローでデータを転送するために使用する変数です。Orchestrator では、ワークフローを実行するたびにワークフロー トークンが保存され、ワークフローの特定の実行の詳細が記録されます。

■ ワークフロー開発プロセスのフェーズ

ワークフローの開発のプロセスには、一連のフェーズが含まれます。開発しているワークフローのタイプに応じて、さまざまな順序のフェーズに従ったり、フェーズをスキップしたりできます。たとえば、カスタム スクリプトなしでワークフローを作成できます。

■ ワークフローの開発のベスト プラクティス

クラスタ化された環境で複数のユーザーが Orchestrator ワークフローを開発する場合、VMware ではいくつかのベスト プラクティスを推奨しています。

■ Orchestrator クライアントのアクセス権限

デフォルトでは、Orchestrator 管理者 LDAP グループのメンバーだけが Orchestrator クライアントにアクセスすることができます。

■ 開発時のワークフローのテスト

ワークフローが完了していない、または終了要素が含まれていない場合であっても、開発プロセスの任意の時点でワークフローをテストできます。

■ ワークフローの作成および編集

ワークフローは、Orchestrator クライアントで作成し、ワークフロー エディタで編集します。ワークフロー エディタは、ワークフローを開発するための Orchestrator クライアントの IDE です。

■ ワークフローの全般情報の指定

ワークフロー エディタの [全般] タブでは、ワークフロー名と説明の入力、ワークフロー動作の属性および特定の機能の定義、署名の確認、ユーザー権限の設定をします。

■ 属性とパラメータの定義

ワークフローの作成後、ワークフローのグローバル属性と入力および出力パラメータを定義する必要があります。

■ ワークフローのスキーマ

ワークフローのスキーマは、ワークフローを、相互接続されたワークフロー要素のフロー図として表したグラフィカル表示です。ワークフローのスキーマによって、ワークフローの論理フローが定義されます。

■ プラグインの開発

Orchestrator のオープンなプラグイン アーキテクチャにより、Orchestrator を各種の管理ソリューションと統合することができます。プラグイン ワークフローを作成して実行し、プラグイン API にアクセスするには、Orchestrator クライアントを使用します。

■ ワークフロー開始時のユーザーからの入力パラメータの取得

ワークフローで入力パラメータが必要な場合、ダイアログ ボックスが開かれ、ユーザーはここに実行の際に必要な入力パラメータ値を入力します。このダイアログ ボックスのコンテンツおよびレイアウト、またはプレゼンテーションは、ワークフロー エディタの [プレゼンテーション] タブで調整できます。

■ (オプション) ワークフローの実行中にユーザー操作を要求する

ワークフローの実行中に外部ソースの追加の入力パラメータが必要となることがあります。別のアプリケーションまたはワークフローの入力パラメータを指定することも、ユーザーが直接指定することもできます。

■ ワークフロー内でのワークフローの呼び出し

ワークフローは、実行中に他のワークフローを呼び出すことができます。ワークフローで他のワークフローを開始するケースには、呼び出し側ワークフローの実行のために、他のワークフローの結果が入力パラメータとして必要なケースと、ワークフローを開始して、そのワークフローに単独で実行を継続させるケースがあります。また、将来の所定の時間にワークフローを開始したり、複数のワークフローを同時に開始したりすることもできます。

■ 選択したオブジェクトに対するワークフローの実行

選択したオブジェクトに対してワークフローを実行することにより、繰り返しタスクを自動化することができます。たとえば、仮想マシン フォルダ内のすべての仮想マシンのスナップショットを作成するワークフローを作成したり、特定のホスト上に存在するすべての仮想マシンをパワーオフするワークフローを作成したりできます。

■ 長期実行ワークフローの開発

待機中のワークフローは、応答を必要とするオブジェクトを絶えずポーリングするためシステム リソースを消費します。ワークフローが必要な応答を受け取るまでの待機時間が長くなる可能性があることがわかっている場合、ワークフローに長期実行ワークフロー要素を追加できます。

■ 構成要素

構成要素は、展開した Orchestrator サーバ全体で、定数の構成に使用できる属性のリストです。

■ ワークフローのユーザー権限

Orchestrator では、グループに適用してワークフローへのアクセスを許可または拒否できる権限のレベルが定義されています。

■ ワークフローの検証

Orchestrator には、ワークフローの検証ツールが用意されています。ワークフローを検証することで、ワークフローのエラーを特定したり、要素間のデータの流れが正しいかどうか確認したりすることができます。

■ ワークフローのデバッグ

Orchestrator には、ワークフローのデバッグ ツールが用意されています。ワークフローをデバッグすることにより、特定の処理の開始時における入力パラメータと出力パラメータの内容を検証したり、編集モードでのワークフローの実行中にパラメータ値や属性値を置き換えたり、最後に失敗した処理からワークフローを再開したりすることができます。

■ 実行中のワークフロー

Orchestrator ワークフローは、イベントの論理的なフローに従って実行されます。

■ 失敗したワークフローの実行の再開

ワークフローに失敗した場合、Orchestrator では最後に失敗したアクティビティからワークフローを再開することができます。

■ ワークフロー ドキュメントの生成

ワークフローまたはワークフロー フォルダに関する文書は、いつでも選択して PDF 形式でエクスポートできます。

■ ワークフロー バージョン履歴の使用

バージョン履歴を使用すると、ワークフローを以前に保存したときの状態に戻すことができます。ワークフローの状態は、ワークフロー バージョンより前のバージョンに戻すことも、より新しいバージョンに戻すこともできます。現在の状態のワークフローと保存されているバージョンのワークフローの違いを比較することもできます。

■ 削除済みワークフローのリストア

ワークフロー ライブラリから削除したワークフローはリストアすることができます。

■ 単純なサンプル ワークフローの開発

単純なサンプル ワークフローの開発では、ワークフロー開発プロセスで最も一般的な手順を示します。

■ 複合ワークフローの開発

複合サンプル ワークフローの開発では、ワークフロー開発プロセスで最も一般的な手順と、カスタムの決定やループの作成などの高度なシナリオを示します。

ワークフローの主要な概念

ワークフローは、スキーマ、属性、およびパラメータで構成されています。ワークフロー スキーマは、すべてのワークフロー要素および要素間の論理接続を定義する、ワークフローの主なコンポーネントです。ワークフローの属性およびパラメータは、ワークフローでデータを転送するために使用する変数です。Orchestrator では、ワークフローを実行するたびにワークフロー トークンが保存され、ワークフローの特定の実行の詳細が記録されます。

ワークフローのパラメータ

ワークフローは、実行時に入力パラメータを受け取って出力パラメータを生成します。

入力パラメータ

ほとんどのワークフローで、特定の入力パラメータ セットを実行する必要があります。入力パラメータは、ワークフローが開始時に処理する引数です。ユーザー、アプリケーション、別のワークフロー、またはアクションが入力パラメータをワークフローに渡し、ワークフローが開始時に処理します。

たとえば、ワークフローで仮想マシンをリセットする場合、そのワークフローでは入力パラメータとして仮想マシンの名前が必要になります。

出力パラメータ

ワークフローの出力パラメータは、ワークフローの実行結果です。出力パラメータは、ワークフローまたはワークフロー要素が実行されると変わる場合があります。ワークフローは、実行時に、他のワークフローの出力パラメータを入力パラメータとして受け取ることができます。

たとえば、ワークフローで仮想マシンのスナップショットを作成する場合、そのワークフローの出力パラメータは、結果として生成されたスナップショットになります。

ワークフローの属性

ワークフロー要素は、入力パラメータとして受け取ったデータを処理し、結果のデータをワークフロー属性または出力パラメータとして設定します。

読み取り専用のワークフロー属性は、ワークフローのグローバル定数として機能します。書き込み可能な属性は、ワークフローのグローバル変数として機能します。

属性を使用して、ワークフローの要素間でデータを転送することができます。次の方法で属性を取得できます。

- ワークフローを作成するときに属性を定義する
- ワークフロー要素の出力パラメータをワークフロー属性として設定する
- 構成要素から属性を継承する

ワークフローのスキーマ

ワークフローのスキーマは、ワークフローを、相互接続されたワークフロー要素のフロー図として表したグラフィカル表示です。ワークフローのスキーマはワークフローの論理を決定するため、ワークフローで最も重要な要素です。

ワークフロー プレゼンテーション

ユーザーがワークフローを実行する際は、ユーザーがワークフローの入力パラメータ値をワークフロー プレゼンテーションに指定します。ワークフロー プレゼンテーションを編成する際は、ワークフローの入力パラメータのタイプと数について考慮します。

ワークフロー トークン

ワークフロー トークンは実行中または完了済みのワークフローを表します。

ワークフローは必要な入力パラメータの汎用手順と汎用セットを定義する抽象的な概念です。実際の入力パラメータセットを使用してワークフローを実行すると、この抽象的なワークフローのインスタンスを受け取ります。このインスタンスは指定した特定の入力パラメータに従って動作します。完了済みまたは実行中のワークフローで 사용되는この特定のインスタンスは、ワークフロー トークンと呼ばれます。

ワークフロー トークン属性

ワークフロー トークン属性はワークフロー トークンの実行に使用される特定のパラメータです。ワークフロー トークン属性はワークフローのグローバル属性の集約であり、ワークフロー トークンの実行に使用される特定の入力および出力パラメータです。

ワークフロー開発プロセスのフェーズ

ワークフローの開発のプロセスには、一連のフェーズが含まれます。開発しているワークフローのタイプに応じて、さまざまな順序のフェーズに従ったり、フェーズをスキップしたりできます。たとえば、カスタム スクリプトなしでワークフローを作成できます。

一般的には、次のフェーズでワークフローを開発します。

- 1 新しいワークフローを作成するか、標準ライブラリから既存のワークフローの複製を作成します。
- 2 ワークフローについての概説を指定します。
- 3 ワークフローの入力パラメータを定義します。
- 4 ワークフロー スキーマをレイアウトおよびリンクして、ワークフローの論理フローを定義します。
- 5 各スキーマ要素の入力および出力パラメータをワークフロー属性にバインドします。
- 6 スクリプト化可能タスク要素またはカスタム決定要素の必要なスクリプトを記述します。
- 7 ユーザーがワークフローを実行するときに見る、入力パラメータ ダイアログ ボックスのレイアウトを定義するワークフロー プレゼンテーションを作成します。
- 8 ワークフローを検証します。

ワークフローの開発のベスト プラクティス

クラスタ化された環境で複数のユーザーが Orchestrator ワークフローを開発する場合、VMware ではいくつかのベスト プラクティスを推奨しています。

- 各開発者は、ワークフローの作成および開発にあたり、スタンドアロン Orchestrator インスタンスの専用テストを受けます。
- ワークフローは、共有のソース コードのコントロール システム上に maven プロジェクトとして保存されます。
- Orchestrator の本番の展開で最適なパフォーマンスを確実に得るには、スケジュールされたウィンドウでワークフローをインポートすることをお勧めします。

- Orchestrator クラスタにワークフローをインポートする場合は、ロード バランサの仮想サーバのアドレスではなく、ローカル ホスト名または IP アドレスを使用して、ノードの 1 つに Orchestrator クライアントを接続します。

注意 ワークフローの変更は、次のワークフローの実行時に有効になります。

Orchestrator クライアントのアクセス権限

デフォルトでは、Orchestrator 管理者 LDAP グループのメンバーだけが Orchestrator クライアントにアクセスすることができます。

Orchestrator 管理者は、**表示権限**以上の権限を設定することにより、Orchestrator クライアントに対するアクセス権限を他のユーザー グループに付与することができます。

特定のユーザーによる Orchestrator クライアントへのアクセスを許可する場合、管理者は、そのユーザーを Orchestrator 管理者 LDAP グループに追加するか、**表示権限**、**確認権限**、**編集権限**、**実行権限**、**管理権限**のいずれかを、そのユーザーが属しているグループに付与する必要があります。

開発時のワークフローのテスト

ワークフローが完了していない、または終了要素が含まれていない場合であっても、開発プロセスの任意の時点でワークフローをテストできます。

Orchestrator はデフォルトで、ワークフローの実行前にそのワークフローが有効かどうかをチェックします。テストのためワークフローを部分的に実行できるように、ワークフローの開発時に自動検証を無効化することができます。

注意 ワークフローの開発が終了したら、忘れずに自動検証を再有効化しておいてください。

手順

- 1 Orchestrator クライアントのメニューで、[ツール] - [ユーザー環境設定] の順にクリックします。
- 2 [ワークフロー] タブをクリックします。
- 3 [ワークフローを実行する前に検証する] チェック ボックスをオフにします。

自動ワークフロー検証が無効化されました。

ワークフローの作成および編集

ワークフローは、Orchestrator クライアントで作成し、ワークフロー エディタで編集します。ワークフロー エディタは、ワークフローを開発するための Orchestrator クライアントの IDE です。

ワークフロー エディタを開くには、既存ワークフローを編集します。

- **ワークフローの作成**

Orchestrator クライアントのワークフローの階層リストで、ワークフローを作成できます。

- **ワークフローの編集**

ワークフローを編集して、既存のワークフローを変更したり、新しい空のワークフローを開発したりします。

■ 標準ライブラリのワークフローの編集

Orchestrator には、ワークフローの標準ライブラリが付属しています。このライブラリを使用して、仮想インフラストラクチャでの操作を自動化することができます。標準ライブラリ内のワークフローは、読み取り専用の状態でロックされています。

■ ワークフロー エディタのタブ

ワークフロー エディタはタブで構成されており、各タブでワークフローのコンポーネントの編集を行います。

ワークフローの作成

Orchestrator クライアントのワークフローの階層リストで、ワークフローを作成できます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 (オプション) ワークフロー フォルダの階層リストのルート、またはリストのフォルダを右クリックし、[フォルダの追加] を選択して新しいワークフロー フォルダを作成します。
- 4 (オプション) 新しいフォルダの名前を入力します。
- 5 新しいフォルダまたは既存のフォルダを右クリックし、[新しいワークフロー] を選択します。
- 6 新しいワークフローの名前を入力し、[OK] をクリックします。

選択したフォルダ内に新しい空のワークフローが作成されます。

次に進む前に

これで、ワークフローを編集できるようになりました。

ワークフローの編集

ワークフローを編集して、既存のワークフローを変更したり、新しい空のワークフローを開発したりします。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ワークフローの階層リストを展開し、編集するワークフローに移動します。
- 4 編集するワークフローを開くには、ワークフローを右クリックして [編集] を選択します。

ワークフロー エディタで編集するワークフローが開きます。

標準ライブラリのワークフローの編集

Orchestrator には、ワークフローの標準ライブラリが付属しています。このライブラリを使用して、仮想インフラストラクチャでの操作を自動化することができます。標準ライブラリ内のワークフローは、読み取り専用の状態でロックされています。

標準ライブラリ内のワークフローを編集するには、そのワークフローの複製を作成する必要があります。複製されたワークフローやカスタム ワークフローであれば、編集することができます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 (オプション) ワークフロー フォルダの階層リストの root を右クリックし、[新しいフォルダ] を選択して、編集対象のワークフローを保存するフォルダを作成します。
- 4 標準ワークフローの [ライブラリ] 階層リストを展開し、編集するワークフローに移動します。
- 5 編集するワークフローを右クリックします。

グレーアウトされた [編集] オプションが表示されます。これは、このワークフローが読み取り専用であることを示しています。

- 6 ワークフローを右クリックして [ワークフローの複製] を選択します。
- 7 複製ワークフローの名前を入力します。
デフォルトでは、複製ワークフローの名前は「<workflow_name> のコピー」になります。
- 8 [ワークフロー フォルダ] という値をクリックし、複製ワークフローを保存するフォルダを検索します。

[手順 3](#) で作成したフォルダを選択します。フォルダを作成しなかった場合は、標準ワークフローのライブラリ内に含まれていないフォルダを選択してください。

- 9 [はい] または [いいえ] をクリックして、ワークフローのバージョン履歴を複製ワークフローにコピーします。

オプション	説明
[はい]	元のワークフローのバージョン履歴が複製ワークフローにコピーされます。
[いいえ]	複製ワークフローのバージョンが「0.0.0」になります。

- 10 [複製] をクリックして、ワークフローを複製します。
- 11 複製されたワークフローを右クリックして [編集] を選択します。

ワークフロー エディタが開きます。これで、複製ワークフローを編集できるようになりました。

標準ライブラリのワークフローが複製されました。これで、複製ワークフローを編集できるようになりました。

ワークフロー エディタのタブ

ワークフロー エディタはタブで構成されており、各タブでワークフローのコンポーネントの編集を行います。

表 1-1. ワークフロー エディタのタブ

タブ	説明
[全般]	ワークフロー名の編集、ワークフローの機能の説明の入力、バージョン番号の設定、ユーザー権限の確認、Orchestrator サーバ再起動時のワークフローの動作の定義、ワークフローのグローバル属性の定義を行うことができます。
[入力]	ワークフローの実行時に必要となるパラメータを定義できます。この入力パラメータは、ワークフローで処理されるデータです。ワークフローの動作はこのパラメータに応じて変わります。
[出力]	ワークフローの実行完了時に生成される値を定義できます。他のワークフローやアクションを実行する際にこの値を使用できます。
[スキーマ]	ワークフローを構築できます。ワークフローを構築するには、[スキーマ] タブの左側にあるワークフロー パレットからワークフロー スキーマ要素をドラッグします。スキーマ図の要素をクリックすると、[スキーマ] タブの下部で要素の動作を定義および編集できるようになります。
[プレゼンテーション]	ユーザーがワークフローを実行するときに表示されるユーザー入力ダイアログ ボックスのレイアウトを定義できます。入力パラメータ ダイアログ ボックスでパラメータを識別しやすくなるように、パラメータと属性をプレゼンテーション ステップおよびグループにまとめることができます。また、パラメータのプロパティを設定することによって、ユーザーがプレゼンテーションで指定できる入力パラメータの制約を定義することもできます。
[パラメータ参照]	ワークフローの論理フローで特定の属性およびパラメータを使用しているワークフロー要素を確認できます。このタブには、[プレゼンテーション] タブで定義したパラメータおよび属性の制約も表示されます。
[ワークフロー トークン]	各ワークフロー実行に関する詳細を確認できます。この情報には、ワークフローのステータス、ワークフローを実行したユーザー、現在の要素のビジネス ステータス、ワークフローの開始日時と終了日時が含まれます。
[イベント]	ワークフロー実行時に発生した個々のイベントに関する情報を確認できます。この情報には、イベントの説明、イベントをトリガしたユーザー、イベントのタイプと発生元、イベント発生日時が含まれます。
[権限]	ユーザーまたはユーザー グループのワークフローと対話する権限を設定できます。

ワークフローの全般情報の指定

ワークフロー エディタの [全般] タブでは、ワークフロー名と説明の入力、ワークフロー動作の属性および特定の機能の定義、署名の確認、ユーザー権限の設定をします。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [全般] タブをクリックします。

- 2 [バージョン] の数字をクリックしてワークフローのバージョン番号を設定します。

[バージョンの説明] ダイアログ ボックスが開きます。

- 3 ワークフローのこのバージョンの説明を入力し、[OK] をクリックします。

たとえば、ワークフローを作成したばかりの場合、**初期作成** と入力します。

新しいバージョンのワークフローが作成されます。後でワークフローの状態をこのバージョンに戻すことができます。

- 4 [サーバ再起動時の動作] の値を設定して、Orchestrator サーバが再起動した場合のワークフローの動作を定義します。

- サーバが停止したときにワークフロー実行が中断されたポイントからワークフローを再開するには、[ワークフロー実行の再開] をデフォルト値のままにします。

- Orchestrator サーバが再起動した場合にワークフローが再開しないようにするには、[ワークフロー実行の再開] をクリックし、[ワークフロー実行を再開しない (失敗として設定)] を選択します。

ワークフローが実行している環境に依存している場合、ワークフローの再開はしないでください。たとえば、ワークフローが特定の vCenter Server を必要とした場合に、開発者が Orchestrator を異なった vCenter Server に接続するように再構成した場合、Orchestrator サーバの再起動後にワークフローを再開すると、ワークフローは失敗します。

- 5 [説明] テキスト ボックスに、ワークフローの詳しい説明を入力します。

- 6 ワークフロー エディタの下部にある [保存] をクリックします。

ワークフロー エディタの左下の緑色のメッセージに、変更を保存したことが示されます。

ワークフローの動作の機能の定義、バージョンの設定、ユーザーがワークフローに対して実行する操作の定義をしました。

次に進む前に

ワークフローの属性およびパラメータを定義する必要があります。

属性とパラメータの定義

ワークフローの作成後、ワークフローのグローバル属性と入力および出力パラメータを定義する必要があります。

ワークフローの属性とは、ワークフローで内部処理されるデータのことです。ワークフローの入力パラメータとは、ユーザーや別のワークフローなどの外部ソースから入力されるデータのことです。ワークフローの出力パラメータは、ワークフローが実行を終了するときに出力されるデータです。

■ ワークフローのパラメータの定義

入力および出力パラメータを使用して、ワークフローとの間でデータをやりとりできます。

■ ワークフローの属性の定義

ワークフローの属性とは、ワークフローで処理されるデータのことです。

■ 属性名とパラメータ名に関する制約

OGNL 式を使用すると、ワークフローの実行中に入力パラメータを動的に判断することができます。Orchestrator の OGNL パーサーは、OGNL 処理の実行中に、ワークフローの属性名やパラメータ名では使用できない特定のキーワードを使用します。

ワークフローのパラメータの定義

入力および出力パラメータを使用して、ワークフローとの間でデータをやりとりできます。

ワークフローのパラメータはワークフロー エディタで定義できます。入力パラメータはワークフローの実行に必要な初期データです。ユーザーはワークフローを実行するときに入力パラメータの値を指定します。出力パラメータは、ワークフローが実行を完了したときに返すデータです。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタで適切なタブをクリックします。

- 入力パラメータを作成するには [入力] をクリックします。
- 出力パラメータを作成するには [出力] をクリックします。

- 2 パラメータ タブの内側を右クリックして、[パラメータの追加] を選択します。

- 3 パラメータ名をクリックして変更します。

デフォルト名は、入力パラメータについては **arg_in_<X>**、出力パラメータについては **arg_out_<X>** です。ここで、<X> は数字です。

- 4 (オプション) パラメータ タイプの値を変更するには、値をクリックして、使用可能な値のリストから値を選択します。

パラメータ タイプの値はデフォルトで [文字列] です。

- 5 [説明] テキスト ボックスにパラメータの説明を追加します。

- 6 (オプション) パラメータを、パラメータでなく属性に変更するには、パラメータを右クリックして、[属性として移動] を選択します。

これで、ワークフローの入力または出力パラメータを定義できました。

次に進む前に

ワークフローのパラメータを定義した後、ワークフロー スキーマを構築します。

ワークフローの属性の定義

ワークフローの属性とは、ワークフローで処理されるデータのことです。

注意 ワークフロー属性は、ワークフロー スキーマを作成するときにワークフロー スキーマ要素に定義することもできます。多くの場合、属性の定義は属性を処理するワークフロー スキーマ要素を作成するときに行った方が簡単です。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [全般] タブをクリックします。

[全般] タブの下半分に [属性] ペインが表示されます。

- 2 [属性] ペインを右クリックし、[属性の追加] を選択します。

属性リストに新しい属性が表示されます。[文字列] がデフォルトのタイプになっています。

- 3 属性名をクリックして変更します。

デフォルト名は **att<X>** です。ここで、<X> は数字です。

注意 ワークフロー属性の名前は、ワークフローのパラメータの名前と同じであってはなりません。

- 4 属性タイプをクリックして、使用可能な値のリストから新しいタイプを選択します。

デフォルトの属性タイプは [文字列] です。

- 5 設定する属性値をクリックするか、属性タイプに応じて値を選択します。

- 6 [説明] テキスト ボックスに、属性の説明を追加します。

- 7 属性が変数ではなく定数の場合、属性名の左側にあるチェック ボックスをオンにして値を読み取り専用にします。

ロック アイコンは、読み取り専用のチェック ボックスの列を示します。

- 8 (オプション) 属性を、属性ではなく入力または出力パラメータに変更する場合、属性を右クリックして、[[入力または出力パラメータとして移動]] を選択します。

これで、ワークフローの属性を定義できました。

次に進む前に

ワークフローの入力および出力パラメータを定義できます。

属性名とパラメータ名に関する制約

OGNL 式を使用すると、ワークフローの実行中に入力パラメータを動的に判断することができます。Orchestrator の OGNL パーサーは、OGNL 処理の実行中に、ワークフローの属性名やパラメータ名では使用できない特定のキーワードを使用します。

OGNL 予約キーワードを属性名のプリフィックスとして使用しても、OGNL 処理が中断されることはありません。たとえば、パラメータに **trueParameter** という名前を付けることができます。予約キーワードでは、大文字と小文字は区別されません。

以下のキーワードを、ワークフローの属性名とパラメータ名で使用することはできません。

表 1-2. 属性名とパラメータ名で使用できないキーワード

使用できないキーワード	使用できないキーワード	使用できないキーワード
<ul style="list-style-type: none"> abstract back_char_esc back_char_literal boolean byte char char_literal class _classResolver const context debugger dec_digits dec_flt default delete digit double dynamic_subscript enum 	<ul style="list-style-type: none"> eof esc exponent export extends false final flt_literal flt_suff ident implements import in int int_literal interface _keepLastEvaluation _lastEvaluation letter long 	<ul style="list-style-type: none"> _memberAccess native package private public root short static string_esc string_literal synchronized this _traceEvaluations true _typeConverter volatil with WithinBackCharLiteral WithinCharLiteral WithinStringLiteral

ワークフローのスキーマ

ワークフローのスキーマは、ワークフローを、相互接続されたワークフロー要素のフロー図として表したグラフィカル表示です。ワークフローのスキーマによって、ワークフローの論理フローが定義されます。

ワークフローのスキーマの表示

ワークフローのスキーマを表示するには、Orchestrator クライアントのワークフローの [スキーマ] タブを使用します。

ワークフロー スキーマでのワークフローのビルド

ワークフロー スキーマは、一連のスキーマ要素で構成されます。ワークフロー スキーマ要素は、ワークフローのビルディング ブロックであり、決定、スクリプト化されたタスク、アクション、例外ハンドラのほか、その他のワークフローを表すこともできます。

スキーマ要素

ワークフロー エディタでは、ワークフロー スキーマ要素が [スキーマ] タブの各メニューに表示されます。[スキーマ] タブで使用可能なスキーマ要素を使用して、ワークフローを構築できます。

■ スキーマ要素のプロパティ

スキーマ要素には、ワークフロー パレットの [スキーマ] タブでユーザーが定義したり編集したりできるプロパティがあります。

■ リンクとバインド

要素間のリンクは、ワークフローの論理フローを決定します。バインドでは、入力および出力パラメータをワークフロー属性にバインドすることによって、要素に他の要素からのデータを取り込みます。

■ 決定

ワークフローでは、ブール値の **true** または **false** ステートメントに応じた異なる一連のアクションを定義する決定関数を実装できます。

■ 例外処理

例外処理ではスキーマ要素の実行中に発生したすべてのエラーをキャッチします。例外処理ではエラーが発生したときにスキーマ要素がどのように動作するかを定義します。

■ エラー ハンドラの使用

標準のエラー ハンドラを使用して、特定のワークフローのスキーマ要素でエラーが発生した場合の動作を定義できます。グローバル エラー ハンドラを使用して、標準のエラー ハンドラではキャッチできないエラーが発生した場合の動作を定義できます。

■ Foreach 要素および複合タイプ

作成したワークフローに Foreach 要素を挿入し、パラメータまたは属性の配列に対して繰り返し実行されるサブワークフローを実行できます。ワークフローをより理解して読みやすくするには、論理的に接続されているがタイプが異なる複数のワークフロー パラメータを、複合タイプと呼ばれる単一のタイプとしてグループ化します。

■ 切り替えアクティビティをワークフローに追加する

基本的な切り替えアクティビティを、ワークフローの属性やパラメータに基づいて切り替えケースを定義するワークフロー スキーマに追加することができます。

ワークフローのスキーマの表示

ワークフローのスキーマを表示するには、Orchestrator クライアントのワークフローの [スキーマ] タブを使用します。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 ワークフローの階層リストのワークフローに移動します。
- 3 ワークフローをクリックします。

右側のペインに、そのワークフローに関する情報が表示されます。

- 4 右側のペインにある [スキーマ] タブを選択します。

ワークフローのグラフィカル表示を確認できます。

ワークフロー スキーマでのワークフローのビルド

ワークフロー スキーマは、一連のスキーマ要素で構成されます。ワークフロー スキーマ要素は、ワークフローのビルディングブロックであり、決定、スクリプト化されたタスク、アクション、例外ハンドラのほか、その他のワークフローを表すこともできます。

ワークフローのビルドは、ワークフロー エディタで行います。ワークフロー エディタの左側にあるワークフロー パレットから、スキーマ要素をワークフロー スキーマ図にドラッグします。

ワークフローのスキーマの編集

ワークフローの論理フローを定義する一連のスキーマ要素を作成することによって、ワークフローを構築します。

デフォルトでは、ワークフロー スキーマ内のすべての要素はリンクされています。要素間のリンクは矢印で示されます。ワークフロー スキーマに新しい要素を追加するときは、要素を矢印にドラッグするか、次の要素にリンクされていない既存のワークフロー要素にドラッグする必要があります。ワークフロー要素をスキーマに追加した後、既存のリンクを削除して新しいリンクを作成し、ワークフローの論理フローを定義できます。

既存のワークフローのスキーマの 1 つ以上の要素を、編集中的ワークフローのスキーマにコピーすることができます。[「ワークフローのスキーマ要素のコピー」](#)を参照してください。

ワークフロー スキーマは [ワークフローの終了] 要素を少なくとも 1 つ持つ必要がありますが、複数持つこともできます。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 左側のペインの [汎用] メニューにあるスキーマ要素をワークフロー スキーマにドラッグします。
- 3 ワークフロー スキーマにドラッグした要素をダブルクリックし、適切な名前を入力して Enter キーを押します。

要素には、ワークフローのコンテキストで一意的な名前を指定する必要があります。

[待機中のタイマー]、[待機中のイベント]、[ワークフローの終了]、または [例外のスロー] 要素の名前は変更できません。

- 4 (オプション) スキーマ内の要素を右クリックして、[コピー] を選択します。
- 5 (オプション) スキーマの適切な位置を右クリックして、[貼り付け] を選択します。

既存のスキーマ要素をコピーして貼り付けると、類似の要素をスキーマにすばやく追加できます。コピー元の要素のビジネスの状態を除くすべての設定が、貼り付けられた要素に表示されます。貼り付けられた要素の設定を適宜調整してください。

- 6 [基本]、[ログ]、または [ネットワーク] メニューのスキーマ要素をワークフロー スキーマにドラッグします。

[基本]、[ログ]、または [ネットワーク] メニューの要素の名前は編集できます。それらのスクリプトは編集できません。

7 [汎用] メニューのスキーマ要素をワークフロー スキーマにドラッグします。

アクションまたはワークフローをワークフロー スキーマにドラッグすると、アクションまたは挿入するワークフローを検索できるダイアログ ボックスが表示されます。

8 [フィルタ] テキスト ボックスに、ワークフローまたはワークフローに挿入するアクションの名前または名前の一部を入力します。

検索に一致したワークフローまたはアクションがダイアログ ボックスに表示されます。

9 ワークフローまたはアクションをダブルクリックして選択します。

ワークフローまたはアクションがワークフロー スキーマに挿入されました。

10 必要なすべてのスキーマ要素をワークフロー スキーマに追加するまでこの手順を繰り返します。

次に進む前に

ワークフロー スキーマに追加した要素のプロパティを定義し、プロパティをすべてリンクしてバインドします。

ワークフローのスキーマ要素のコピー

既存のワークフローのスキーマの 1 つ以上の要素を、編集中のワークフローのスキーマにコピーすることができます。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 左側のペインで以下のいずれかの操作を実行して、スキーマ要素のコピー元となるワークフローを選択します。
 - [すべてのワークフロー] をクリックし、ワークフローの階層リストで目的のワークフローを選択します。
 - 検索テキスト ボックスで目的のワークフローの名前を入力して Enter キーを押します。
- 3 選択したワークフローを右クリックして [オープン] を選択します。
ワークフローのプロパティを表示するウィンドウが開きます。
- 4 ワークフローのウィンドウで [スキーマ] タブをクリックします。
- 5 1 つ以上のスキーマ要素を選択して右クリックし、[コピー] を選択します。
- 6 編集中のワークフローの [スキーマ] タブ内で右クリックして [貼り付け] を選択します。

これで、コピー元のワークフローからコピー先のワークフローにスキーマ要素がコピーされました。

次に進む前に

コピーしたスキーマ要素は、既存のワークフローのスキーマにリンクしてバインドする必要があります。

入力および出力パラメータの昇格

入力および出力パラメータを子要素から親ワークフローに昇格させることができます。

ワークフロー エディタの [全般] タブで定義したカスタムの属性を昇格させることができます。事前定義した属性は、入力パラメータを一致タイプの属性と交換することでのみ昇格させることができます。

注意 事前定義した属性を昇格させてカスタムの値を割り当てた場合、重複した属性が作成されて、元の属性の値の上書きを防止します。重複した属性は元の属性の名前のままになり、属性の名前の末尾に付いている数字が増加します。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 ワークフロー スキーマにワークフローまたはアクション要素を追加します。

スキーマ ペインの上部に以下の通知が表示されます。

現在のワークフローの入力/出力としてアクティビティのパラメータを追加しますか？

- 3 通知上で、[セットアップ] をクリックします。
使用可能なオプションのポップアップ ウィンドウが表示されます。
- 4 各入力パラメータのマッピング タイプを選択します。

オプション	説明
Input	引数は入力ワークフロー パラメータにマッピングされます。
Skip	引数は NULL 値にマッピングされます。
Value	引数は Value の列に設定される値で属性にマッピングされます。

- 5 各出力パラメータのマッピング タイプを選択します。

オプション	説明
Output	引数は出力ワークフローパラメータにマッピングされます。
Skip	引数は NULL 値にマッピングされます。
Local variable	引数は属性にマッピングされます。

- 6 [昇格] をクリックします。

パラメータが親ワークフローに昇格されます。

検索結果の変更

[検索] テキスト ボックスを使用して、ワークフローやアクションなどの要素を検索します。検索で返される結果が部分的な場合、検索で返される結果の数を変更できます。

要素に検索を使用するときに、緑色のメッセージ ボックスに、検索ですべての結果を一覧表示することが示されます。黄色のメッセージ ボックスには、検索が部分的な結果のみを一覧表示することが示されます。

手順

- 1 (オプション) ワークフロー エディタでワークフローを編集している場合、[保存して閉じる] をクリックしてエディタを終了します。
- 2 Orchestrator クライアントのメニューから、[ツール] - [ユーザー環境設定] の順に選択します。
- 3 [全般] タブをクリックします。
- 4 [ファインダの最大サイズ] テキスト ボックスに検索結果を返す数を入力します。
- 5 [ユーザー環境設定] ダイアログ ボックスで [保存して閉じる] をクリックします。

これで、検索で返される結果の数が変更されました。

スキーマ要素

ワークフロー エディタでは、ワークフロー スキーマ要素が [スキーマ] タブの各メニューに表示されます。[スキーマ] タブで使用可能なスキーマ要素を使用して、ワークフローを構築できます。

表 1-3. スキーマ要素およびアイコン

スキーマ要素の名前	説明	アイコン	ワークフロー エディタ内の場所
[ワークフローの開始]	ワークフローの開始点。すべてのワークフローがこの要素を含んでいます。ワークフローに設定できる開始要素は 1 つのみです。開始要素には 1 つの出力があり、入力はありません。また、ワークフローのスキーマから開始要素を削除することはできません。		常に [スキーマ] タブに表示される
[スクリプト化可能タスク]	ユーザーが定義する汎用のタスク。この要素で JavaScript 関数を記述します。		[汎用] ワークフロー パレット
[決定]	ブール値関数。決定要素は 1 つの入力パラメータを取り、 true または false のどちらかを返します。要素が行う決定のタイプは、入力パラメータのタイプによって異なります。決定要素を使用すると、ワークフローは、その決定要素が受信する入力パラメータに応じて異なる方向に分岐できます。受信された入力パラメータが期待される値に対応している場合、ワークフローは特定のルートに沿って続行されます。入力が期待される値でない場合、ワークフローは代替パスで続行されます。		[汎用] ワークフロー パレット
[カスタム決定]	ブール値関数。カスタム決定は複数の入力パラメータを取り、カスタム スクリプトに従ってそれら进行处理できます。 true または false のどちらかを返します。		[汎用] ワークフロー パレット
[決定アクティビティ]	ブール値関数。決定アクティビティはワークフローを実行し、その出力パラメータを true または false パスにバインドします。		[汎用] ワークフロー パレット

表 1-3. スキーマ要素およびアイコン (続き)

スキーマ要素の名前	説明	アイコン	ワークフロー エディタ内の場所
[ユーザー操作]	ユーザーがワークフローに新しい入力パラメータを渡すことができますようにします。ユーザー操作要素が入力パラメータへの要求を提示したり、ユーザーが指定できるパラメータに制約を設定したりする方法を設計できます。どのユーザーが入力パラメータを指定できるかを決定する権限を設定できます。実行中のワークフローは、ユーザー操作要素に達するとパッシブ状態になり、ユーザーに入力を要求します。ユーザーが入力を指定する必要があるタイムアウト期間を設定できます。ワークフローはユーザーから渡されたデータに従って再開するか、またはタイムアウト期間が経過した場合は例外を返します。ユーザーからの応答を待機している間、ワークフロー トークンは waiting 状態にあります。		[汎用] ワークフロー パレット
[待機中のタイマー]	長時間実行されるワークフローによって使用されます。実行中のワークフローは、待機中のタイマー要素に達するとパッシブ状態になります。ワークフローが実行を再開する絶対的な日付を設定します。この日付を待機している間、ワークフロー トークンは waiting-signal 状態にあります。		[汎用] ワークフロー パレット
[待機中のイベント]	長時間実行されるワークフローで使用されます。実行中のワークフローは、待機中のイベント要素に達するとパッシブ状態になります。ワークフローが実行を再開する前に待機するトリガー イベントを定義します。このイベントを待機している間、ワークフロー トークンは waiting-signal 状態にあります。		[汎用] ワークフロー パレット
[ワークフローの終了]	ワークフローのエンド ポイント。スキーマ内に、ワークフローで考えられるさまざまな結果を表す複数の終了要素を設定できます。終了要素には 1 つの入力があり、出力はありません。ワークフローがワークフローの終了要素に達すると、ワークフロー トークンは completed 状態になります。		[汎用] ワークフロー パレット
[例外のスロー]	例外を作成し、ワークフローを停止します。この要素は、ワークフロー スキーマ内に複数回現れる場合があります。例外要素には、文字列タイプのみが可能な 1 つの入力パラメータがあり、出力パラメータはありません。ワークフローが例外要素に達すると、ワークフロー トークンは failed 状態になります。		[汎用] ワークフロー パレット
[ワークフロー メモ]	ユーザーがワークフローの各セクションに注釈を付けることができますようにします。ワークフローの各セクションを詳細に説明するようにメモを拡張できます。メモの背景色を変更することによって、ワークフローのゾーンを区別できます。ワークフロー メモは、スキーマの理解に役立つ視覚情報のみを提供します。		[汎用] ワークフロー パレット
[アクション要素]	アクションの Orchestrator ライブラリからアクションを呼び出します。ワークフローは、アクション要素に達すると、そのアクションを呼び出して実行します。		[汎用] ワークフロー パレット

表 1-3. スキーマ要素およびアイコン (続き)

スキーマ要素の名前	説明	アイコン	ワークフロー エディタ内の場所
[ワークフロー要素]	別のワークフローを同期的に開始します。ワークフローは、そのスキーマ内のワークフロー要素に達すると、そのワークフローを独自のプロセスの一部として実行します。元のワークフローは、呼び出されたワークフローがその実行を完了した後にのみ続行されます。		[汎用] ワークフロー パレット
[Foreach 要素]	アレイのすべての要素でワークフローを実行します。たとえば、あるフォルダからすべての仮想マシンで [仮想マシン名の変更] ワークフローを実行できます。		[汎用] ワークフロー パレット
[非同期ワークフロー]	ワークフローを非同期的に開始します。ワークフローは、非同期ワークフロー要素に達すると、そのワークフローを開始して独自の実行を続行します。元のワークフローは、呼び出されたワークフローの完了を待ちません。		[汎用] ワークフロー パレット
[ワークフローのスケジュール設定]	設定された時間にワークフローを実行するためのタスクを作成した後、ワークフローはその実行を続行します。		[汎用] ワークフロー パレット
[ネストされたワークフロー]	複数のワークフローを同時に開始します。別の Orchestrator サーバに存在するローカルワークフローとリモートワークフローをネストすることを選択できます。また、異なる認証情報を使用してワークフローを実行することもできます。ワークフローは、ネストされたすべてのワークフローが完了するのを待ってから、その実行を続行します。		[汎用] ワークフロー パレット
[エラーの処理]	特定のワークフロー要素のエラーを処理します。ワークフローは、例外を作成するか、別のワークフローを呼び出すか、またはカスタム スクリプトを実行することによってエラーを処理できます。		[汎用] ワークフロー パレット
[デフォルトのエラー ハンドラ]	標準のエラー ハンドラではキャッチできないワークフロー エラーを処理します。使用可能な任意のスキーマ要素を使用してエラーを処理できます。		[汎用] ワークフロー パレット

表 1-3. スキーマ要素およびアイコン (続き)

スキーマ要素の名前	説明	アイコン	ワークフロー エディタ内の場所
[切り替え]	ワークフロー属性またはパラメータに基づいて、代替ワークフローパスに切り替えます。		[汎用] ワークフロー パレット
[事前定義済みタスク]	ワークフローが一般に使用する標準タスクを実行する、編集不可能なスクリプト化された要素。次のタスクが事前定義済みです。 [基本] <ul style="list-style-type: none"> ■ スリープ ■ 認証情報の変更 ■ 日付までの待機 ■ カスタム イベントの待機 ■ カウンタの増加 ■ カウンタの減少 [ログ] <ul style="list-style-type: none"> ■ システム ログ ■ システム警告 ■ システム エラー ■ サーバ ログ ■ サーバ警告 ■ サーバエラー ■ システム + サーバ ログ ■ システム + サーバ警告 ■ システム + サーバエラー [ネットワーク] <ul style="list-style-type: none"> ■ HTTP post ■ HTTP get 		[基本]、[ログ]、および [ネットワーク] ワークフロー パレット

スキーマ要素のプロパティ

スキーマ要素には、ワークフロー パレットの [スキーマ] タブでユーザーが定義したり編集したりできるプロパティがあります。

スキーマ要素のグローバル プロパティの編集

スキーマ要素のグローバル プロパティは要素の [情報] タブで定義します。

開始する前に

ワークフロー エディタの [スキーマ] タブに要素が含まれていることを確認します。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 [編集] アイコン (✎) をクリックして編集する要素を選択します。
要素のプロパティをリストするダイアログ ボックスが表示されます。

3 [情報] タブをクリックします。

4 [名前] テキスト ボックスに、スキーマ要素の名前を入力します。

この名前が、ワークフロー スキーマ図のスキーマ要素に表示されます。

5 [相互作用] ドロップダウン メニューで、説明を選択します。

[相互作用] プロパティによって、この要素がワークフロー外部のオブジェクトとどのように相互作用するかについての基本的な説明を選択できます。このプロパティは情報の提供のみを目的としています。

6 (オプション) [ビジネス ステータス] テキスト ボックスに、ビジネス ステータスの説明を入力します。

[ビジネス ステータス] プロパティは、この要素が何をするかについての簡単な説明です。ワークフローの実行中に、ワークフロー トークンは各要素が実行するときのビジネス ステータスを表示します。この機能はワークフロー ステータスのトラッキングに便利です。

7 (オプション) [説明] テキスト ボックスに、スキーマ要素の説明を入力します。

スキーマ要素のプロパティ タブ

スキーマ要素のプロパティには、ワークフローのスキーマにドラッグした要素をクリックすることによってアクセスします。その要素のプロパティが、ワークフロー エディタの下部にあるタブに表示されます。

プロパティ タブは、スキーマ要素ごとに異なります。

表 1-4. スキーマ要素ごとのプロパティ タブ

スキーマ要素のプロパティ タブ	説明	適用対象のスキーマ要素タイプ
[属性]	外部ソース（ユーザー、イベント、タイマーなど）から要素に対して必要な属性。属性には、タイムアウト制限、日時、トリガー、ユーザー認証情報などがあります。	<ul style="list-style-type: none"> ■ ユーザー操作 ■ 待機中のイベント ■ 待機中のタイマー
[決定]	決定ステートメントを定義します。決定要素が受信する入力パラメータは、決定ステートメントに一致するか一致しないかのどちらかであり、それによって2つのアクションが考えられます。	決定
[ワークフローの終了]	ワークフローが正常に完了したか、またはエラーが発生して例外が返されたかのどちらかのために、ワークフローを停止します。	<ul style="list-style-type: none"> ■ 終了 ■ 例外

表 1-4. スキーマ要素ごとのプロパティ タブ (続き)

スキーマ要素のプロパティ タブ	説明	適用対象のスキーマ要素タイプ
[例外]	例外が発生した場合のこのスキーマ要素の動作。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ 例外 ■ ネストされたワークフロー ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク ■ ユーザー操作 ■ 待機中のイベント ■ 待機中のタイマー ■ ワークフロー
[外部入力]	ワークフロー実行中の特定の時点でユーザーが指定する必要がある入力パラメータ。	<ul style="list-style-type: none"> ■ ユーザー操作
[IN]	この要素に対する IN バインド。IN バインドは、スキーマ要素が、ワークフロー内でその前にある要素からの入力を受信する方法を定義します。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ カスタム決定 ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク ■ ワークフロー
[情報]	スキーマ要素の全般プロパティおよび説明。[情報] タブに表示される情報は、スキーマ要素のタイプによって異なります。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ カスタム決定 ■ 決定 ■ ネストされたワークフロー ■ メモ ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク ■ ユーザー操作 ■ 待機中のイベント ■ 待機中のタイマー ■ ワークフロー
[OUT]	この要素に対する OUT バインド。OUT バインドは、スキーマ要素が出力パラメータをワークフロー属性またはワークフロー出力パラメータにバインドする方法を定義します。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク ■ ワークフロー

表 1-4. スキーマ要素ごとのプロパティ タブ (続き)

スキーマ要素のプロパティ タブ	説明	適用対象のスキーマ要素タイプ
[プレゼンテーション]	ワークフローの実行中にユーザー入力が必要になった場合にユーザーに表示される入力パラメータ ダイアログ ボックスのレイアウトを定義します。	ユーザー操作
[スクリプティング]	このスキーマ要素の動作を定義する JavaScript 関数を示します。非同期ワークフロー、ワークフローのスケジュール設定、およびアクション要素の場合、このスクリプティングは読み取り専用です。スクリプト化可能タスクおよびカスタム決定要素の場合は、このタブで JavaScript を編集します。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ カスタム決定 ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク
[視覚的なバインド]	このスキーマ要素のパラメータと属性が、ワークフロー内でその前後にある要素のパラメータと属性にどのようにバインドされるかをグラフィカルに表示します。これは、その要素の IN および OUT バインドの別の表現です。	<ul style="list-style-type: none"> ■ アクション ■ 非同期ワークフロー ■ 事前定義済みタスク ■ ワークフローのスケジュール設定 ■ スクリプト化可能タスク ■ ワークフロー
[ワークフロー]	ネストするワークフローを選択します。	ネストされたワークフロー

リンクとバインド

要素間のリンクは、ワークフローの論理フローを決定します。バインドでは、入力および出力パラメータをワークフロー属性にバインドすることによって、要素に他の要素からのデータを取り込みます。

リンクおよびバインドを理解するには、ワークフローの論理フローとワークフローのデータ フローの違いを理解する必要があります。

ワークフローの論理フロー

ワークフローの論理フローとは、ワークフローが実行するときのスキーマでの 1 つの要素から次の要素へのワークフローの進行のことです。ワークフローの論理フローは、スキーマの要素をリンクして定義します。

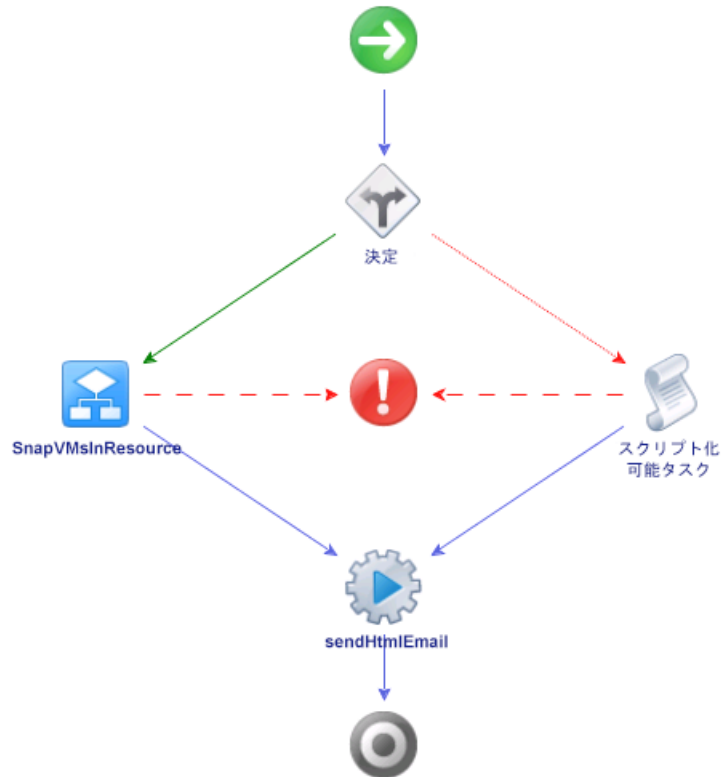
標準パスとは、すべての要素が期待どおりに実行した場合にワークフローが論理フローを通るパスです。例外パスとは、すべての要素が期待どおりに実行しない場合にワークフローが論理フローを通るパスです。

ワークフロー スキーマにあるさまざまなスタイルの矢印は、ワークフローが論理フローを通ることができるさまざまなパスを示しています。

- 青色の矢印は、ワークフローが 1 つの要素から次の要素へ移動する標準パスを示しています。
- 緑色の矢印は、ブール決定要素が **true** を返した場合にワークフローが通るパスを示しています。
- 赤色の点線矢印は、ブール決定要素が **false** を返した場合にワークフローが通るパスを示しています。
- 赤色の破線矢印は、ワークフロー要素が正常に実行しない場合にワークフローが通る例外パスを示しています。

次の図に、ワークフローが通ることができるさまざまなパスを表したワークフロー スキーマのサンプルを示します。

図 1-1. ワークフローの論理フローを通るさまざまなワークフロー パス



このワークフロー サンプルは、論理フローを介して次のパスを通ります。

- 標準パス、**true** の決定結果、例外なし。
 - a 決定要素が **true** を返します。
 - b **SnapVMsInResourcePool** ワークフローが正常に実行します。
 - c **sendHtmlEmail** アクションが正常に実行します。
 - d ワークフローは、**completed** の状態で正常に終了します。
- 標準パス、**false** の決定結果、例外なし。
 - a 決定要素が **false** を返します。
 - b スクリプト化可能タスク要素が定義する動作が正常に実行します。
 - c **sendHtmlEmail** アクションが正常に実行します。
 - d ワークフローは、**completed** の状態で正常に終了します。
- **true** の決定結果、例外。
 - a 決定要素が **true** を返します。
 - b **SnapVMsInResourcePool** ワークフローでエラーが発生します。
 - c ワークフローは例外を返し、**failed** の状態で停止します。

- **false** の決定結果、例外。
 - a 決定要素が **false** を返します。
 - b スクリプト化可能タスク要素が定義する動作でエラーが発生します。
 - c ワークフローは例外を返し、**failed** の状態で停止します。

要素リンク

リンクはスキーマ要素を接続し、1 つの要素から別の要素へのワークフローの論理フローを定義します。

要素は通常、ワークフロー内の別の要素への 1 つのみの発信リンクと、要素の例外動作を定義する要素への 1 つの例外リンクを設定できます。発信リンクはワークフローの標準パスを定義します。例外リンクはワークフローの例外パスを定義します。たいていの場合、単一スキーマ要素は複数の要素からの受信標準パス リンクを受け取ることができます。

次の要素は前の内容の例外です。

- [ワークフローの開始] 要素は受信リンクを受け取ることができず、例外リンクもありません。
- 例外要素は複数の受信例外リンクを受け取ることができ、発信リンクも例外リンクもありません。
- 決定要素は決定の **true** または **false** 結果に応じてワークフローが選出するパスを定義する 2 つの発信リンクを持ちます。決定には例外リンクがありません。
- [ワークフローの終了] 要素は発信リンクおよび例外リンクを持つことができません。

標準パス リンクの作成

標準パス リンクは、ワークフローの通常実行を決定します。

要素を別の要素にリンクするときは、ワークフローで実行する順序で要素をリンクします。2 つの要素間でリンクを作成するには、先に実行する要素から開始します。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- ワークフロー エディタの [スキーマ] タブに要素が含まれていることを確認します。

手順

- 1 別の要素に接続する要素にポインタを置きます。
青色および赤色の矢印が要素の右に表示されます。
- 2 青色の矢印にポインタを置きます。
青色の矢印が大きくなります。
- 3 青色の矢印を左クリックし、マウスの左ボタンを押したままターゲット要素にポインタを移動します。
2 つの要素の間に青色の矢印が表示され、ターゲット要素の周囲に緑色の長方形が表示されます。
- 4 マウスの左ボタンを離します。
青色の矢印は 2 つの要素の間に残ります。

これで、標準パスが要素をリンクしました。

次に進む前に

要素は結合されましたが、データ フローは定義していません。入力および出力のバインドを定義して、受信および送信データをワークフロー属性にバインドする必要があります。

ワークフローのデータ フロー

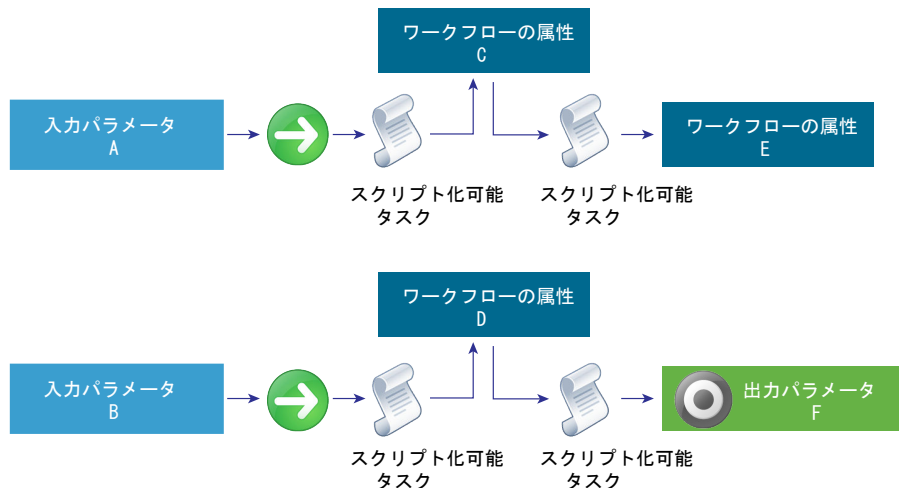
ワークフローのデータ フローとは、ワークフローの各要素が実行されるときにワークフロー要素の入力および出力パラメータがワークフロー属性にバインドされるための手段です。ワークフローのデータ フローを定義するには、スキーマ要素バインドを使用します。

ワークフロー スキーマ内の要素が実行されるときは、入力パラメータの形式でデータが要求されます。入力パラメータのデータを取得するには、ワークフローを作成するときに設定したワークフロー属性にバインドするか、またはワークフローで先行要素が実行されたときに設定された属性にバインドします。

要素はデータを処理し、必要に応じて変換し、実行の結果を出力パラメータの形式で生成します。要素は、生成された出力パラメータを作成する新しいワークフロー属性にバインドします。スキーマ内のその他の要素は、入力パラメータとしてこのような新しいワークフロー属性にバインドできます。ワークフローは、実行の最後に出力パラメータとして属性を生成します。

次の図に、非常に単純なワークフローを示します。青色の矢印は、要素のリンク、およびワークフローの論理フローを表します。赤色の線は、ワークフローのデータ フローを示しています。

図 1-2. ワークフローのデータ フローのサンプル



このワークフローにおけるデータ フローは次のようになります。

- 1 ワークフローは、入力パラメータ A および B で開始します。
- 2 1 つ目の要素は、パラメータ A を処理し、処理の結果をワークフロー属性 C にバインドします。
- 3 1 つ目の要素は、パラメータ B を処理し、処理の結果をワークフロー属性 D にバインドします。
- 4 2 つ目の要素は、ワークフロー属性 C を入力パラメータとして使用して処理し、得られた出力パラメータをワークフロー属性 E にバインドします。

- 5 2つ目の要素は、ワークフロー属性 D を入力パラメータとして使用して処理し、出力パラメータ F を生成します。
- 6 ワークフローは終了し、実行の結果としてワークフロー属性 F を出力パラメータとして生成します。

要素のバインド

ワークフロー要素のすべての入力および出力パラメータをワークフロー属性にバインドする必要があります。バインドにより要素内にデータが設定され、要素の出力および例外の動作が定義されます。リンクはワークフローの論理フローを定義し、バインドはデータフローを定義します。

要素内にデータを設定し、処理後に要素から出力パラメータを生成し、要素の実行中に発生する可能性があるすべてのエラーを処理するには、要素バインドを設定する必要があります。

IN バインド

スキーマ要素の受信データを設定します。要素のローカル入力パラメータをソースワークフロー属性にバインドします。[入力] タブは要素の入力パラメータを [ローカル パラメータ] 列にリストします。[入力] タブは、ローカル パラメータがバインドするワークフロー属性を [ソース パラメータ] 列にリストします。タブにはパラメータタイプおよびパラメータの説明も表示されます。

OUT バインド

要素が実行を終了したときにワークフロー属性を変更して出力パラメータを生成します。[出力] タブは要素の出力パラメータを [ローカル パラメータ] 列にリストします。[出力] タブは、ローカル パラメータがバインドするワークフロー属性を [ソース パラメータ] 列にリストします。タブにはパラメータタイプおよびパラメータの説明も表示されます。

例外バインド

要素の実行中に例外が発生した場合に例外ハンドラにリンクします。

IN バインドはバインドされているソース パラメータから値を読み取ります。**OUT** バインドはバインドされているソース パラメータに値を書き込みます。

スキーマ要素内で使用するすべての属性または入力パラメータをワークフロー属性にバインドするには、**IN** バインドを使用する必要があります。要素が実行中に受信する入力パラメータの値が要素によって変更される場合は、**OUT** バインドを使用して、入力パラメータをワークフロー属性にバインドする必要があります。要素の出力パラメータをワークフロー要素にバインドすると、ワークフロースキーマ内でその要素の後に続く他の要素が、それらの出力パラメータを入力パラメータとして取得することができます。

ワークフローを作成するときによくある間違いは、要素がワークフロー属性に対して加えた変更を反映するように出力パラメータ値をバインドすることを怠ることです。

重要 ワークフロー内ですでに定義したタイプの入力および出力パラメータを必要とする要素を追加した場合、それらのパラメータへのバインドは Orchestrator によって設定されます。Orchestrator によってバインドされたパラメータが正しいことを確認する必要があります。これは、その要素がバインドできる同じタイプの異なるパラメータがワークフローで定義されている場合があるためです。

要素バインドの定義

要素をリンクしてワークフローの論理フローを作成したら、要素バインドを定義して、各要素が受信および生成するデータを処理する方法を定義します。

開始する前に

ワークフロー エディタの [スキーマ] タブにワークフロー スキーマがあること、および要素の間にリンクを作成してあることを確認します。

手順

- 1 バインドを設定する要素の [編集] アイコン (✎) をクリックします。
要素のプロパティをリストしたダイアログ ボックスが表示されます。
- 2 [入力] タブをクリックします。
[入力] タブの内容は、選択した要素のタイプによって異なります。
 - 事前定義されているタスク、ワークフロー、またはアクション要素を選択した場合、[入力] タブにはその要素タイプで使用可能なローカル入力パラメータがリストされますが、バインドは設定されていません。
 - 別のタイプの要素を選択した場合、[入力] タブを右クリックして [ワークフローのパラメータ/属性にバインド] を選択することにより、ワークフローですでに定義した入力パラメータおよび属性のリストから選択できます。
 - 必要な属性がまだ存在しない場合は、[入力] タブを右クリックして [ワークフローのパラメータ/属性にバインド] - [ワークフローでパラメータ/属性を作成] を選択することにより、必要な属性を作成できます。
- 3 適切なパラメータが存在する場合、バインドする入力パラメータを選択し、[ソース パラメータ] テキスト ボックスの [未設定] ボタンをクリックします。
バインドに使用できるソース パラメータおよび属性のリストが表示されます。
- 4 表示されたリストからローカル入力パラメータにバインドするソース パラメータを選択します。
- 5 (オプション) バインドするソース パラメータを定義していない場合は、パラメータ選択ダイアログ ボックスの [ワークフローでパラメータ/属性を作成] リンクをクリックすることにより、ソース パラメータを作成できます。
- 6 [出力] タブをクリックします。
[出力] タブの内容は、選択した要素のタイプによって異なります。
 - 事前定義されているタスク、ワークフロー、またはアクション要素を選択した場合、[出力] タブにはその要素タイプで使用可能なローカル出力パラメータがリストされますが、バインドは設定されていません。
 - 別のタイプの要素を選択した場合、[出力] タブを右クリックして [ワークフローのパラメータ/属性にバインド] を選択することにより、ワークフローで定義した出力パラメータおよび属性のリストから選択できます。
 - 必要な属性が存在しない場合は、[入力] タブを右クリックして [ワークフローのパラメータ/属性にバインド] - [ワークフローでパラメータ/属性を作成] を選択することにより、必要な属性を作成できます。
- 7 バインドするパラメータを選択します。
- 8 [ソース パラメータ] - [未設定] ボタンの順にクリックします。
- 9 入力パラメータにバインドするソース パラメータを選択します。
- 10 (オプション) バインドするパラメータを定義していない場合は、パラメータ選択ダイアログ ボックスの [ワークフローでパラメータ/属性を作成] ボタンをクリックすることにより、パラメータを作成できます。

要素が受信する入力パラメータおよび要素が生成する出力パラメータを定義し、ワークフローの属性およびパラメータにバインドしました。

次に進む前に

これで、決定を定義してワークフローのパスにフォークを作成できるようになりました。

決定

ワークフローでは、ブール値の **true** または **false** ステートメントに応じた異なる一連のアクションを定義する決定関数を実装できます。

決定とは、ワークフローにおけるフォークです。ワークフローの決定は、ユーザー、ほかのワークフロー、アプリケーション、またはワークフローの実行環境によって提供される入力に従って行われます。決定要素が受信する入力パラメータの値によって、ワークフローが取るフォークの分岐が決まります。たとえば、ワークフローの決定が仮想マシンの電源ステータスを入力として受信するとします。仮想マシンがパワーオン状態のとき、このワークフローは論理フローで特定のパスを取ります。仮想マシンがパワーオフ状態のとき、このワークフローは別のパスを取ります。

決定は、常にブール関数です。各決定で起こりうる結果は **true** または **false** のみです。

カスタム決定

カスタム決定が標準の決定と異なるのは、決定ステートメントをスクリプトで定義する点です。次のサンプルに示すように、カスタム決定は定義したステートメントに応じて **true** または **false** を返します。

```
if (<decision_statement>){  
    return true;  
}else{  
    return false;  
}
```

決定要素リンクの作成

決定要素は、ワークフローのその他の要素とは異なります。決定要素には **true** または **false** 出力パラメータのみがあります。決定要素に例外リンクはありません。

開始する前に

ワークフロー エディタの [スキーマ] タブに、他の要素にリンクされていない 1 つ以上の決定要素を含め、複数の要素が表示されていることを確認します。

手順

- 1 決定要素にマウス ポインタを置き、ワークフローで可能性のある 2 つの分岐を定義する他の 2 つの要素にリンクします。

青色の矢印および赤色の矢印が要素の右に表示されます。

- 2 青色の矢印にポインタを置き、マウスの左ボタンを押したままターゲット要素にポインタを移動します。

2つの要素の間に緑色の矢印が表示され、ターゲット要素が緑色になります。緑色の矢印は、決定要素が受信した入力パラメータまたは属性が決定ステートメントと一致する場合にワークフローが選択する **true** パスを表します。

- 3 マウスの左ボタンを離します。

緑色の矢印は2つの要素の間に残ります。これで、決定要素が予想される値を受信したときにワークフローが選択するパスを定義しました。

- 4 決定要素にポインタを置き、マウスの左ボタンを押したままターゲット要素にポインタを移動します。

2つの要素の間に赤色の点線矢印が表示され、ターゲット要素が緑色になります。赤色の矢印は、決定要素が受信した入力パラメータまたは属性が決定ステートメントと一致しない場合にワークフローが選択する **false** パスを表します。

- 5 マウスの左ボタンを離します。

赤色の点線矢印は2つの要素の間に残ります。これで、決定要素が予想しない入力を受信したときにワークフローが選択するパスを定義しました。

これで、決定要素が受信する入力パラメータまたは属性に応じてワークフローが選択する **true** または **false** パスを定義しました。

次に進む前に

決定ステートメントを定義します。[「決定を使用したワークフロー分岐の作成」](#) を参照してください。

リンクされている決定要素の削除

リンクされている決定要素をワークフロー スキーマから削除する場合は、削除するワークフローのパスを指定する必要があります。

開始する前に

ワークフロー エディタの [スキーマ] タブに、**true** パスと **false** パスを持つ 1 つ以上の決定要素を含め、複数の要素が表示されていることを確認します。

手順

- 1 決定要素を選択して [削除] を押します。

使用可能なオプションを示すダイアログ ボックスが表示されます。

- 2 削除する決定分岐を選択します。

オプション	説明
成功分岐	決定要素と、 true 決定パスに従うすべての要素が、ワークフロー スキーマから削除されます。
失敗分岐	決定要素と、 false 決定パスに従うすべての要素が、ワークフロー スキーマから削除されます。

オプション	説明
両方の分岐	決定要素と、両方の決定パスに従うすべての要素が、ワークフロー スキーマから削除されます。
なし	決定要素とそのリンクだけが、ワークフロー スキーマから削除されます。両方の決定パスに従うすべての要素は、ワークフロー スキーマ内に残ったままになります。

3 [OK] をクリックします。

決定を使用したワークフロー分岐の作成

決定要素は、ワークフロー内で分岐を作成するために使用する単純なブール関数です。決定要素は、受信した入力と設定した決定ステートメントとが一致するかどうかを判断します。この決定の機能として、ワークフローは取りうる 2 つのパスの一方に沿って処理を続行します。

開始する前に

決定を定義する前に、ワークフロー エディタで決定要素がスキーマのその他 2 つの要素にリンクしていることを確認します。

手順

- 決定要素の [編集] アイコン (✎) をクリックします。
決定要素のプロパティをリストしたダイアログ ボックスが表示されます。
- ダイアログ ボックスの [決定] タブをクリックします。
- [未設定 (NULL)] リンクをクリックし、この決定のソース入力パラメータを選択します。
このワークフローで定義されているすべての属性および入力パラメータをリストしたダイアログ ボックスが表示されます。
- リストから入力パラメータをダブルクリックして選択します。
- バインドするソース パラメータを定義していない場合は、パラメータ選択ダイアログ ボックスの [ワークフローでパラメータ/属性を作成] リンクをクリックすることにより、ソース パラメータを作成します。
- ドロップダウン メニューから決定ステートメントを選択します。
メニューに表示されるステートメントはコンテキスト依存であり、選択した入力パラメータのタイプによって異なります。
- 決定ステートメントと一致させる値を追加します。
選択した入力タイプおよびステートメントによっては、値テキスト ボックスに [未設定 (NULL)] リンクが表示されることがあります。このリンクをクリックすると、事前定義されている値の選択肢が表示されます。それ以外の場合、たとえば文字列では値を入力するテキスト ボックスになります。

決定要素のステートメントを定義しました。決定要素が入力パラメータを受信すると、入力パラメータの値とステートメント内の値とを比較し、ステートメントの真偽を判断します。

次に進む前に

ワークフローが例外を処理する方法を設定する必要があります。

例外処理

例外処理ではスキーマ要素の実行中に発生したすべてのエラーをキャッチします。例外処理ではエラーが発生したときにスキーマ要素がどのように動作するかを定義します。

決定要素および開始と終了要素を除くワークフロー内のすべての要素には、例外処理のためだけに機能する特定の出力パラメータ タイプが含まれています。要素の実行中にエラーが発生すると、要素は例外ハンドラにエラー信号を送信できます。例外ハンドラはエラーをキャッチし、受け取ったエラーに応じて応答します。定義した例外ハンドラが特定のエラーを処理できない場合、要素の例外出力パラメータを例外要素にバインドすることができ、例外要素がワークフロー実行を **failed** 状態で終了します。

例外は、ワークフロー要素内で **try** および **catch** のシーケンスとして機能します。特定の例外を要素内で処理する必要がない場合、要素の例外出力パラメータをバインドする必要はありません。

例外の出力パラメータ タイプは常に **errorCode** オブジェクトになります。

例外のバインドの作成


要素は、その要素でエラーが発生した場合のワークフローの動作を定義するバインドを設定できます。

開始する前に

ワークフロー エディタの [スキーマ] タブに要素が含まれていることを確認します。

手順

- 1 例外のバインドを定義する要素にポインタを置きます。
赤色の矢印が要素の右に表示されます。
- 2 赤色の矢印にポインタを置き、矢印が大きくなったらマウスの左ボタンを押したままターゲット要素にドラッグします。
赤色の破線矢印が2つの要素をリンクします。ターゲット要素は、リンク先要素でエラーが発生した場合のワークフローの動作を定義します。
- 3 例外処理要素にリンクしている要素の [編集] アイコン (✎) をクリックします。
- 4 スキーマ要素プロパティ タブの [例外] タブをクリックします。
- 5 [出力例外のバインド] 値を設定するには、[未設定] をクリックします。
 - 例外属性バインド ダイアログ ボックスで例外出力パラメータにバインドするパラメータを選択し、[選択] をクリックします。
 - [ワークフローでパラメータ/属性を作成] をクリックして、例外出力パラメータを作成します。
- 6 例外処理動作を定義するターゲット要素をクリックします。
- 7 スキーマ要素プロパティ タブの [入力] タブをクリックします。
- 8 [ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
入力パラメータを選択するためのダイアログ ボックスが表示されます。

- 9 例外出力パラメータを選択し、[選択] をクリックします。
- 10 スキーマ要素プロパティ タブで例外処理要素の [出力] タブをクリックします。
- 11 例外処理要素の動作を定義します。
 - [ワークフローのパラメータ/属性にバインド] アイコン () をクリックして、生成する例外処理要素の出力パラメータを選択します。
 - [スクリプティング] タブをクリックし、JavaScript を使用して例外処理要素の動作を定義します。

これで、要素が例外を処理する方法を定義しました。

次に進む前に

ユーザーがワークフローを実行するときにユーザーから入力パラメータを取得する方法を定義する必要があります。

エラー ハンドラの使用

標準のエラー ハンドラを使用して、特定のワークフローのスキーマ要素でエラーが発生した場合の動作を定義できます。グローバル エラー ハンドラを使用して、標準のエラー ハンドラではキャッチできないエラーが発生した場合の動作を定義できます。

エラー ハンドラをワークフローに追加する

エラー ハンドラをワークフロー要素に追加することにより、ワークフローの実行中に特定のワークフロー要素で発生したエラーの処理方法を定義することができます。エラー ハンドラは、エラー パスが指定されていないワークフロー要素に対してのみ追加することができます。

重要 バージョン 5.5.x 以前の Orchestrator では、[エラーの処理] 要素が含まれているワークフローを使用することはできません。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [エラーの処理] 要素を、ワークフロー スキーマ内の適切な要素にドラッグします。
ダイアログ ボックスが表示されます。
- 2 ダイアログ ボックスのドロップダウン メニューで、エラーの処理方法を選択します。

オプション	説明
例外のスロー	エラーが発生すると、例外がスローされます。例外のバインドを変更することができます。
ワークフローの呼び出し	エラーが発生すると、選択したワークフローが実行されます。
カスタム スクリプト	エラーが発生すると、カスタム スクリプトが実行されます。

3 [選択] をクリックします。

これで、エラー ハンドラがワークフローに追加されました。ワークフローがこの要素に到達すると、上記の手順で選択したアクションが実行されてから、ワークフローが終了します。

グローバル エラー ハンドラをワークフローに追加する

グローバル エラー ハンドラをワークフロー スキーマに追加することにより、ワークフローの実行中に標準のエラー ハンドラではキャッチできないエラーの処理方法を定義することができます。1 つのワークフロー スキーマに対して 1 つのグローバル エラー ハンドラを追加することができます。

重要 バージョン 5.5.x 以前の Orchestrator では、[デフォルトのエラー ハンドラ] 要素が含まれているワークフローを使用することはできません。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [デフォルトのエラー ハンドラ] 要素をワークフロー スキーマにドラッグします。
- 2 (オプション) [デフォルトのエラー ハンドラ] 要素と [例外のスロー] 要素の間にスキーマ要素を追加し、グローバルなワークフロー エラーの処理方法を指定します。

これで、グローバル エラー ハンドラがワークフローに追加されました。ワークフローの標準のエラー ハンドラではキャッチできないエラーが発生すると、グローバル エラー ハンドラにより、上記の手順で指定したアクションが実行されてから、ワークフローの実行が終了します。

Foreach 要素および複合タイプ

作成したワークフローに Foreach 要素を挿入し、パラメータまたは属性の配列に対して繰り返し実行されるサブワークフローを実行できます。ワークフローをより理解して読みやすくするには、論理的に接続されているがタイプが異なる複数のワークフロー パラメータを、複合タイプと呼ばれる単一のタイプとしてグループ化します。

Foreach 要素の使用

Foreach 要素は入力パラメータまたは属性の配列に対して繰り返し実行されるサブワークフローを実行します。サブワークフローの実行対象の配列を選択し、この配列の値をワークフローの実行時に渡すことができます。サブワークフローは配列に定義した要素の数だけ実行されます。

属性の配列が含まれている構成要素が存在する場合は、Foreach 要素内のこれらの属性に対して繰り返し実行されるワークフローを実行できます。

たとえば、フォルダ内に名前を変更したい仮想マシンが 10 台含まれているとします。その場合、ワークフロー内に Foreach 要素を挿入し、その要素内で「仮想マシンの名前変更」ワークフローをサブワークフローとして定義します。「仮想マシンの名前変更」ワークフローは、2 つの入力パラメータ、すなわち 1 台の仮想マシン、その仮想マシンの新しい名前を取得します。これらのパラメータは入力として現在のワークフローに昇格できます。その場合、これらのパラメータは「仮想マシンの名前変更」ワークフローが繰り返し実行される配列になります。ワークフローを実行する際、フォルダ内の 10 台の仮想マシンとそれらの新しい名前を指定できます。ワークフローが実行されるたびに仮想マシンの配列の要素と、新しい名前の配列の要素が仮想マシンに取得されます。

複合タイプの使用

複合タイプは論理的に接続しているがタイプが異なる複数の入力パラメータまたは属性のグループです。Foreach 要素ではパラメータのグループを複合値としてバインドできます。これにより、Foreach 要素はワークフローが実行されるたびにグループ化されたパラメータの値をすぐに取得します。

たとえば、1 台の仮想マシンの名前を変更するとします。その場合、仮想マシン オブジェクトとその新しい名前が必要になります。複数の仮想マシンの名前を変更する場合は、仮想マシン用と仮想マシンの名前用に 2 つの配列が必要になります。これらの 2 つの配列は明示的に接続されていません。複合タイプでは各要素に仮想マシンと仮想マシンの名前の両方が含まれている 1 つの配列を使用できます。これにより、複数の値を持つこれらの 2 つのパラメータ間の接続はワークフロー スキーマで明示的に指定され、暗黙的に指定されることはありません。

注意 複合タイプが含まれているワークフローを vSphere Web Client から実行することはできません。

Foreach 要素の定義

サブワークフローを繰り返し実行し、そのたびに異なるパラメータ値や属性値をサブワークフローに渡す場合は、親のワークフローに Foreach 要素を挿入します。

Foreach 要素を挿入する場合は、Foreach 要素の繰り返しの対象となる配列を 1 つ以上選択する必要があります。配列要素ごとに、親のワークフローに続いて実行される各サブワークフローに対して複数の異なる値を設定することができます。

サブワークフローに出力パラメータが指定されている場合、その出力パラメータに対してサブワークフローを繰り返し実行できるように、ワークフローの出力を蓄積する Foreach 要素の出力パラメータを選択する必要があります。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタで [スキーマ] タブを選択します。
- 2 [一般] メニューの Foreach 要素をワークフロー スキーマにドラッグします。
- 3 [選択] ダイアログ ボックスで任意のワークフローを選択します。

スキーマ ペインの上部に以下の通知が表示されます。

現在のワークフローの入力/出力としてアクティビティのパラメータを追加しますか？

4 通知上で、[セットアップ] をクリックします。

使用可能なオプションのポップアップ ウィンドウが表示されます。

5 各入力パラメータのマッピング タイプを選択します。

オプション	説明
Input	引数は入力ワークフロー パラメータにマッピングされます。
Skip	引数は NULL 値にマッピングされます。
Value	引数は Value の列に設定される値で属性にマッピングされます。

6 各出力パラメータのマッピング タイプを選択します。

オプション	説明
Output	引数は出力ワークフローパラメータにマッピングされます。
Skip	引数は NULL 値にマッピングされます。
Local variable	引数は属性にマッピングされます。

7 [昇格] をクリックします。

8 Foreach 要素を右クリックして [同期] - [プレゼンテーションの同期] を選択します。

確認用ダイアログ ボックスが表示されます。

9 [OK] をクリックして、Foreach 要素のプレゼンテーションを現在のワークフローに適用します。

操作の結果を通知するダイアログ ボックスが表示されます。

10 [入力] タブで、サブワークフローのパラメータが配列タイプの要素として追加されていることを確認します。

11 [出力] タブで、サブワークフローのパラメータが配列タイプの要素として追加されていることを確認します。

これで、ワークフロー内に Foreach 要素が定義されました。Foreach 要素は、パラメータまたは属性の定義済み配列の各要素からパラメータを取得するワークフローを実行します。

パラメータや属性が配列として定義されていない状態でワークフローを繰り返し実行すると、常に同じ値が使用されます。

例: Foreach 要素を使用した仮想マシン名の変更

Foreach 要素を使用して、複数の仮想マシンの名前を同時に変更することができます。そのためには、Foreach 要素をワークフロー内に挿入し、**vm** パラメータと **newName** パラメータを現在のワークフローに対する入力として昇格させる必要があります。この方法の場合、ワークフローの実行時に、名前を変更する仮想マシンと、その仮想マシンの新しい名前を指定します。名前を変更する仮想マシンは、**vm** パラメータに対して作成した配列内に要素として格納されます。これらの仮想マシンの新しい名前は、**newName** パラメータに対して作成した配列内に格納されます。

Foreach 要素で複合タイプを定義する

論理的に接続されている複数のワークフロー パラメータを、複合タイプと呼ばれる新しいタイプにグループ化することができます。Foreach 要素を使用すると、パラメータのグループを複合値としてバインドし、パラメータの複数の配列を 1 つの配列として接続することができます。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- ワークフローに Foreach 要素が含まれていることを確認します。

手順

- 1 Foreach 要素の [IN] タブまたは [OUT] タブを選択します。
- 2 他のローカル パラメータとともに 1 つの複合タイプとしてグループ化したいローカル パラメータを選択します。
- 3 [IN] タブまたは [OUT] タブの上部に表示されている [パラメータのグループを複合値としてバインド] をクリックします。
- 4 [バインド] ペインで、複合タイプとしてグループ化したいパラメータを選択します。
- 5 [イテレータとしてバインド] を選択します。

これで、複合タイプの配列に対して Foreach 要素が繰り返し適用されるようになります。

- 6 [承諾] をクリックします。

これで、複合タイプが定義されました。この複合タイプの配列に対して、ワークフローが繰り返し実行されるようになります。複合タイプとしてグループ化されたパラメータには、`<composite_type_name>.<parameter_name>` という名前が付けられます。たとえば、**snapshots** という複合タイプを作成した場合、この複合タイプとしてグループ化されるパラメータの名前は、**snapshots.vm[in-parameter]** や **snapshots.name[in-parameter]** などのようになります。複合タイプの配列のすべての要素には、その複合タイプとしてグループ化された各パラメータの 1 つのインスタンスが格納されます。

例: 仮想マシン名の変更

たとえば、10 台の仮想マシンの名前を同時に変更するとします。その場合、ワークフロー内に Foreach 要素を挿入し、その要素内で [仮想マシンのワークフロー名を変更] を選択します。次に、複合タイプを作成して、**vm** パラメータと **newName** パラメータを明示的に接続します。その後、作成した複合タイプをイテレータとしてバインドします。これにより、**vm** パラメータと **newName** パラメータの両方を格納する 1 つの配列が作成されます。

切り替えアクティビティをワークフローに追加する

基本的な切り替えアクティビティを、ワークフローの属性やパラメータに基づいて切り替えケースを定義するワークフロー スキーマに追加することができます。

1 つの切り替えアクティビティに対して複数の切り替えケースを設定することができます。すべての切り替えケースは、属性またはパラメータに関連する条件によって定義されます。該当する条件が満たされると、実行中のワークフローが、対応する定義済みワークフロー要素に切り替わります。指定されたいずれの条件も満たされない場合、実行中のワークフローは、デフォルトの定義済みワークフロー要素に切り替わります。

重要 バージョン 5.5.x 以前の Orchestrator では、[切り替え] 要素が含まれているワークフローを使用することはできません。

開始する前に

ワークフロー エディタの [スキーマ] タブに要素が含まれていることを確認します。

手順

- 1 [切り替え] 要素を、ワークフロー スキーマ内の適切な要素にドラッグします。
- 2 [切り替え] 要素の [編集] アイコン (✎) をクリックします。
- 3 [ケース] タブで、切り替えケースを追加または削除します。
その際、切り替えケースの優先順位を変更することができます。
- 4 各切り替えケースの条件を定義します。
- 5 各切り替えケースについて、対応するワークフロー要素を選択します。
- 6 切り替え先となるデフォルトのワークフロー要素を選択します。
- 7 [閉じる] をクリックします。
- 8 [保存] をクリックします。

これで、切り替えケースの条件とワークフローのパスが定義されました。

プラグインの開発

Orchestrator のオープンなプラグイン アーキテクチャにより、Orchestrator を各種の管理ソリューションと統合することができます。プラグイン ワークフローを作成して実行し、プラグイン API にアクセスするには、Orchestrator クライアントを使用します。

プラグインの概要

Orchestrator のプラグインには標準のコンポーネント セットが含まれている必要があり、プラグインは標準のアーキテクチャに従う必要があります。これらのプラクティスは、使用可能な幅広いさまざまな外部のテクノロジーに対してプラグインを作成する際に役立ちます。

■ [Orchestrator プラグインの構造](#)

Orchestrator プラグインは、特定の機能を実装する様々なタイプのレイヤで構成される共通の構造を持ちます。

■ [外部の API を Orchestrator に公開する](#)

Orchestrator プラグインを作成することにより、外部製品の API を Orchestrator プラットフォームに公開することができます。API を公開する任意のテクノロジーに対してプラグインを作成し、Orchestrator で使用可能な JavaScript オブジェクトにその API をマップすることができます。

■ [プラグインのコンポーネント](#)

プラグインは、プラグイン テクノロジーのオブジェクトを Orchestrator プラットフォームに公開する、標準のコンポーネント セットで構成されています。

■ vso.xml ファイルの役割

vso.xml ファイルを使用して、プラグイン テクノロジーのオブジェクト、クラス、メソッド、属性を、Orchestrator のインベントリ オブジェクト、スクリプト タイプ、スクリプト クラス、スクリプト メソッド、属性にマップします。**vso.xml** ファイルは、プラグインの構成動作と起動動作も定義します。

■ プラグイン アダプタのロール

プラグイン アダプタは、Orchestrator サーバへのプラグインのエントリ ポイントです。プラグイン アダプタは、Orchestrator サーバのプラグイン テクノロジー用のデータストアとして機能し、プラグイン ファクトリを作成し、プラグイン テクノロジーで発生するイベントを管理します。

■ プラグイン ファクトリのロール

プラグイン ファクトリは、Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらのオブジェクトに対して操作を実行する方法を定義します。

■ ファインダ オブジェクトの役割

ファインダ オブジェクトは、プラグイン テクノロジーを使用する管理対象オブジェクト タイプの特定のインスタンスを識別、特定します。Orchestrator はファインダ オブジェクトに対してワークフローを実行することにより、プラグイン テクノロジーのオブジェクトを変更、使用できます。

■ スクリプト オブジェクトの役割

スクリプト オブジェクトは、プラグイン テクノロジーのオブジェクトを表す JavaScript 表現です。プラグインのスクリプト オブジェクトは Orchestrator の Javascript API に表示され、ワークフローおよびアクションのスクリプト化された要素内で使用できます。

■ イベント ハンドラの役割

イベントは変更されたプラグイン テクノロジーのオブジェクトの状態または属性で、Orchestrator によって検出されます。Orchestrator はイベント ハンドラを実装してイベントを監視します。

Orchestrator プラグインの構造

Orchestrator プラグインは、特定の機能を実装する様々なタイプのレイヤで構成される共通の構造を持ちます。

Orchestrator プラグインの下から 3 つのレイヤは、インフラストラクチャ クラス、ラッパー クラス、スクリプト オブジェクトであり、プラグイン テクノロジーと Orchestrator 間の接続を実装します。

Orchestrator プラグインの上から 3 つのレイヤはユーザーから見える部分であり、アクション、ビルディング ブロック ワークフロー、ハイレベル ワークフローというレイヤがあります。

図 1-3. Orchestrator プラグインの構造



インフラストラクチャ クラス	プラグイン テクノロジーと Orchestrator 間を接続する一連のクラスです。インフラストラクチャ クラスには、プラグインの定義に従い、プラグイン ファクトリ、プラグイン アダプタなど、実装のためのクラスが含まれます。インフラストラクチャ クラスにはまた、ヘルパー、キャッシュ、インベントリなど、共通のタスクやオブジェクトに機能を備えるためのクラスも含まれています。
ラッパー クラス	プラグイン テクノロジーのオブジェクト モデルを、Orchestrator 内で公開するオブジェクト モデルに取り入れる一連のクラスです。
スクリプト オブジェクト	JavaScript オブジェクト タイプであり、プラグイン テクノロジーのラッパー クラス、メソッド、属性へのアクセスを可能にします。 vso.xml ファイルでは、プラグイン テクノロジーのどのラッパー クラス、属性およびメソッドが Orchestrator に公開されるか定義します。
アクション	ワークフローやスクリプティング タスクで直接使用できる一連の JavaScript の機能です。アクションでは複数の入力パラメータを使用でき、戻り値は1つになります。
ビルディング ブロック ワークフロー	プラグインで実装するすべての一般的な機能をカバーする一連のワークフローです。通常、ビルディング ブロック ワークフローは、統合されたテクノロジーのユーザー インターフェイスの操作を表します。ビルディング ブロック ワークフローは、直接使用する、またはハイレベル ワークフロー内に含めることができます。
ハイレベル ワークフロー	プラグインの特定の機能をカバーする一連のワークフローです。ハイレベル ワークフローにより、具体的な要件を満たしたり、プラグインの複雑な使用例を示したりすることができます。

外部の API を Orchestrator に公開する

Orchestrator プラグインを作成することにより、外部製品の API を Orchestrator プラットフォームに公開することができます。API を公開する任意のテクノロジーに対してプラグインを作成し、Orchestrator で使用可能な JavaScript オブジェクトにその API をマップすることができます。

プラグインにより、Java のオブジェクトとメソッドが JavaScript オブジェクトにマップされ、Orchestrator のスクリプト API に追加されます。外部のテクノロジーを使用して Java API を公開する場合は、その API を Orchestrator の JavaScript に直接マップし、ワークフローやアクションで使用することができます。

公開した API を、WSDL (Web Service Definition Language)、REST (Representational State Transfer)、またはメッセージング サービスを使用して Java オブジェクトに統合することにより、Java 以外の言語で API を公開するアプリケーションのプラグインを作成することができます。その後、統合された Java オブジェクトを Orchestrator の JavaScript にマップして使用することができます。

プラグインを使用するテクノロジーは、Orchestrator からは独立して機能します。ソース コードにアクセスできず、Java アーカイブ (JAR ファイル) などのバイナリ コードにしかアクセスできない場合であっても、外部製品用の Orchestrator プラグインを作成することができます。

プラグインのコンポーネント

プラグインは、プラグイン テクノロジーのオブジェクトを Orchestrator プラットフォームに公開する、標準のコンポーネント セットで構成されています。

プラグインの主要なコンポーネントは、プラグイン アダプタ、ファクトリ、およびイベント実装です。アダプタ、ファクトリ、およびイベント実装で定義されたオブジェクトおよび操作は、**vso.xml** という名前の XML 定義ファイル内の Orchestrator オブジェクトにマップします。**vso.xml** ファイルは、プラグイン テクノロジーのオブジェクトおよび機能を、Orchestrator JavaScript API に示される JavaScript スクリプト オブジェクトにマップします。また、**vso.xml** ファイルは、プラグイン テクノロジーのオブジェクト タイプを Orchestrator の [インベントリ] タブに表示されるファインダにマップします。

プラグインは次のコンポーネントで構成されています。

プラグイン モジュール	Java クラスのセットで定義されるプラグイン自体、 vso.xml ファイル、およびプラグインを介してアクセスするオブジェクトと通信するワークフローとアクションのパッケージ。プラグイン モジュールは必須になります。
プラグイン アダプタ	プラグイン テクノロジーと Orchestrator サーバ間のインターフェイスを定義します。アダプタは、Orchestrator プラットフォームへのプラグインのエントリ ポイントです。アダプタは、プラグイン ファクトリを作成し、プラグインのロードおよびアンロードを管理し、プラグイン テクノロジーのオブジェクトで発生するイベントを管理します。プラグイン アダプタは必須になります。
プラグイン ファクトリ	Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらに対して操作を実行する方法を定義します。アダプタにより、Orchestrator とプラグイン テクノロジーとの間で開かれるクライアント セッション用のファクトリが作成されます。ファクトリでは、すべてのクライアント接続間で 1 つのセッションを共有するか、クライアント接続ごとに 1 つのセッションを開くことができます。プラグイン ファクトリは必須になります。

構成	Orchestrator では、プラグインの構成を保存する標準的な方法は定義されていません。構成情報は、Windows レジストリや静的構成ファイルを使用する、またはデータベースや XML ファイルに情報を保存することによって保存できます。Orchestrator プラグインは、Orchestrator クライアントで構成ワークフローを実行して構成できます。
ファインダ	Orchestrator がプラグイン テクノロジーのオブジェクトを配置または表示する方法を定義する操作ルール。ファインダは、プラグイン テクノロジーが Orchestrator に公開しているオブジェクトのセットからオブジェクトを取得します。開発者は、オブジェクトのネットワークを経由して移動できるようにするため、オブジェクト間の関係を vso.xml ファイルに定義します。Orchestrator は、プラグイン テクノロジーのオブジェクト モデルを [インベントリ] タブに表示します。プラグイン テクノロジーのオブジェクトを Orchestrator に公開する場合には、ファインダは必須になります。
スクリプト オブジェクト	プラグイン テクノロジーのオブジェクト、操作、および属性へのアクセスを提供する JavaScript オブジェクト タイプ。スクリプト オブジェクトは、Orchestrator が JavaScript を介してプラグイン テクノロジーのオブジェクト モデルにアクセスする方法を定義します。開発者は、プラグイン テクノロジーのクラスおよびメソッドを vso.xml ファイルの JavaScript オブジェクトにマップします。Orchestrator のスクリプト API にある JavaScript オブジェクトにアクセスし、それらを Orchestrator のスクリプト化されたタスク、アクション、およびワークフローに統合できます。スクリプト オブジェクトは、スクリプト タイプ、クラス、およびメソッドを Orchestrator JavaScript API に追加する場合には、必須になります。
インベントリ	Orchestrator が Orchestrator クライアントの [インベントリ] ビューに表示されたファインダを使用して配置する、プラグイン テクノロジーにあるオブジェクトのインスタンス。インベントリのオブジェクトに対しては、それらに対してワークフローを実行することで、操作を実行できます。インベントリはオプションです。開発者は、Orchestrator JavaScript API にスクリプト タイプおよびクラスを追加するのみで、インベントリ内のオブジェクトのインスタンスを公開しないプラグインを作成できます。
イベント	プラグイン テクノロジーにあるオブジェクトの状態の変化。Orchestrator は、プラグイン テクノロジーで発生するイベントを受動的にリスニングできます。また、Orchestrator はプラグイン テクノロジーのイベントをアクティブにトリガすることもできます。イベントはオプションです。

vso.xml ファイルの役割

vso.xml ファイルを使用して、プラグイン テクノロジーのオブジェクト、クラス、メソッド、属性を、Orchestrator のインベントリ オブジェクト、スクリプト タイプ、スクリプト クラス、スクリプト メソッド、属性にマップします。**vso.xml** ファイルは、プラグインの構成動作と起動動作も定義します。

vso.xml ファイルは以下の主要な役割を実行します。

起動および構成の動作	プラグインを起動する方法と、プラグインが定義する構成の実装を特定する方法を定義します。プラグイン アダプタをロードします。
インベントリ オブジェクト	プラグイン テクノロジーを使用してプラグインがアクセスするオブジェクトのタイプを定義します。プラグイン ファクトリの実装のファインダ メソッドは、これらのオブジェクトのインスタンスを特定し、Orchestrator インベントリに表示します。
スクリプト タイプ	Orchestrator の JavaScript API にスクリプト タイプを追加して、インベントリ内のオブジェクトの異なるタイプを表します。これらのスクリプト タイプはワークフローで入力パラメータとして使用できます。
スクリプト クラス	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能なクラスを、Orchestrator の JavaScript API に追加します。
スクリプト メソッド	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能なメソッドを、Orchestrator の JavaScript API に追加します。
スクリプト属性	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能な、プラグイン テクノロジーを使用するオブジェクトの属性を、Orchestrator の JavaScript API に追加します。

プラグイン アダプタのロール

プラグイン アダプタは、Orchestrator サーバへのプラグインのエントリ ポイントです。プラグイン アダプタは、Orchestrator サーバのプラグイン テクノロジー用のデータストアとして機能し、プラグイン ファクトリを作成し、プラグイン テクノロジーで発生するイベントを管理します。

プラグイン アダプタを作成するには、**IPluginAdaptor** インターフェイスを実装する Java クラスを作成します。

作成するプラグイン アダプタ クラスは、プラグイン テクノロジーのプラグイン ファクトリ、イベント、およびトリガを管理します。**IPluginAdaptor** インターフェイスには、これらのタスクの実行に使用するメソッドが用意されています。

プラグイン アダプタは次の主要なロールを実行します。

ファクトリの作成	プラグイン アダプタの最も重要なロールは、各接続用の 1 つのプラグイン ファクトリ インスタンスを Orchestrator からプラグイン テクノロジーにロードまたはアンロードすることです。プラグイン アダプタ クラスは IPluginAdaptor.createPluginFactory() メソッドを呼び出して、 IPluginFactory インターフェイスを実装するクラスのインスタンスを作成します。
イベントの管理	プラグイン アダプタは、Orchestrator サーバとプラグイン テクノロジーとの間のインターフェイスです。プラグイン アダプタは、Orchestrator がプラグイン テクノロジーのオブジェクトに対して実行またはウォッチするイベントを管理します。このアダプタは、イベント パブリッシャを介してイベントを管理します。イベント パブリッシャは、アダプタが

IPluginAdaptor.registerEventPublisher() メソッドを呼び出すことによって作成する **IPluginEventPublisher** インターフェイスのインスタンスです。イベント パブリッシャは、プラグイン テクノロジーのオブジェクトに対してトリガおよびゲージを設定し、オブジェクトで特定のイベントが発生した場合、またはオブジェクトの値が特定のしきい値を超えた場合に、Orchestrator が定義済みアクションを起動できるようにします。同様に、長期実行ワークフローの待機イベント要素が待機するイベントを定義する **PluginTrigger** および **PluginWatcher** インスタンスを定義できます。

プラグイン名の設定

プラグインの名前は、**vso.xml** ファイル内で指定します。プラグイン アダプタはこの名前を **vso.xml** ファイルから取得し、Orchestrator クライアントの [インベントリ] ビューに公開します。

ライセンスのインストール

アダプタ実装においてプラグイン テクノロジーが必要とするライセンス ファイルをインストールするメソッドを呼び出すことができます。

IPluginAdaptor インターフェイス、そのすべてのメソッド、およびプラグイン API の他のすべてのクラスについての詳細は、[「Orchestrator プラグイン API リファレンス」](#) を参照してください。

プラグイン ファクトリのロール

プラグイン ファクトリは、Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらのオブジェクトに対して操作を実行する方法を定義します。

プラグイン ファクトリを作成するには、Orchestrator プラグイン API から **IPluginFactory** インターフェイスを実装および拡張する必要があります。作成したプラグイン ファクトリ クラスは、Orchestrator がプラグイン テクノロジーのオブジェクトにアクセスするために使用するファインダ機能を定義します。ファクトリを使用すると、Orchestrator サーバは ID によって、他のオブジェクトに対する関係によって、またはクエリ文字列を検索することによって、オブジェクトを検索できます。

プラグイン ファクトリは次の主要なロールを実行します。

オブジェクトの検索

オブジェクトの名前およびタイプに従ってオブジェクトを検索する機能を作成できます。**IPluginFactory.find()** メソッドを使用してオブジェクトを名前およびタイプで検索します。

他のオブジェクトに関連するオブジェクトの検索

指定されたオブジェクトに関連するオブジェクトを、指定された関係のタイプによって検索する機能を作成できます。関係は、**vso.xml** ファイルに定義します。また、すべての親に関連する依存子オブジェクトを、指定された関係のタイプによって検索するファインダを作成できます。**IPluginFactory.findRelation()** メソッドを実装して、指定された親オブジェクトに關係するオブジェクトを、指定さ

れた関係のタイプによって検索します。

IPluginFactory.hasChildrenInRelation() メソッドを実装して、親インスタンスに対して少なくとも 1 つの子オブジェクトが存在するかどうかを調べます。

独自の基準に基づいてオブジェクトを検索するクエリを定義

自分で定義したクエリ ルールを実装するオブジェクト ファインダを作成できます。**IPluginFactory.findAll()** メソッドを実装して、ファクトリがこのメソッドを呼び出したときに、自分で定義したクエリ ルールを満たすすべてのオブジェクトを検索します。自分で定義したクエリ ルールに一致するすべてのオブジェクトのリストを含む **QueryResult** オブジェクトの **findAll()** メソッドの結果を取得します。

IPluginFactory インターフェイス、そのすべてのメソッド、およびプラグイン API の他のすべてのクラスについての詳細は、[「Orchestrator プラグイン API リファレンス」](#) を参照してください。

ファインダ オブジェクトの役割

ファインダ オブジェクトは、プラグイン テクノロジーを使用する管理対象オブジェクト タイプの特定のインスタンスを識別、特定します。Orchestrator はファインダ オブジェクトに対してワークフローを実行することにより、プラグイン テクノロジーのオブジェクトを変更、使用できます。

プラグイン テクノロジーを使用する特定の管理対象オブジェクト タイプのすべてのインスタンスには、Orchestrator のファインダ オブジェクトで見つけられるように一意の識別子を付ける必要があります。プラグイン テクノロジーでは、オブジェクトインスタンスの一意の識別子が文字列として提供されます。ワークフローの実行中、Orchestrator は見つかったオブジェクトの一意の識別子をワークフローの属性値として設定します。特定のタイプのオブジェクトを入力パラメータとして使用するワークフローは、そのタイプのオブジェクトの特定のインスタンスに対して実行されます。

プラグインにより Orchestrator の JavaScript API に追加されるファインダ オブジェクトには、プラグイン名がプリフィックスとして付きます。たとえば、vCenter Server API の **VirtualMachine** 管理対象オブジェクト タイプは、Orchestrator で **VC:VirtualMachine** JavaScript タイプとして表示されます。

たとえば、Orchestrator は特定の **VC:VirtualMachine** インスタンスにアクセスする場合、vCenter Server プラグインを使用し、仮想マシンの **id** 属性を一意の識別子として使用するファインダ オブジェクトを実装します。このオブジェクト インスタンスは属性値としてワークフロー要素に渡すことができます。

Orchestrator プラグインは、プラグイン テクノロジーのオブジェクトを、**vso.xml** ファイル内の **<finder>** 要素にある同等の Orchestrator ファインダ オブジェクトにマップします。**<finder>** 要素は、プラグイン テクノロジーのメソッドまたは関数を特定します。プラグイン テクノロジーでは、オブジェクトの特定のインスタンスに使用する一意の識別子が取得されます。また、**<finder>** 要素はオブジェクト間の関係も定義し、他のオブジェクトとの関係に基づいてオブジェクトを見つけます。

Orchestrator の [インベントリ] タブには、ファインダ オブジェクトが含まれているプラグインの下にファインダ オブジェクトが表示されます。

スクリプト オブジェクトの役割

スクリプト オブジェクトは、プラグイン テクノロジーのオブジェクトを表す JavaScript 表現です。プラグインのスクリプト オブジェクトは Orchestrator の JavaScript API に表示され、ワークフローおよびアクションのスクリプト化された要素内で使用できます。

プラグインのスクリプト オブジェクトは JavaScript のモジュール、タイプ、クラスとして Orchestrator の JavaScript API に表示されます。ほとんどのファインダ オブジェクトにはスクリプト オブジェクト表現があります。JavaScript クラスはメソッドと属性を Orchestrator の JavaScript API に追加できます。Orchestrator の JavaScript API は、プラグイン テクノロジーの API にあるオブジェクトのメソッドと属性を表示します。プラグイン テクノロジーは、Orchestrator とは別にオブジェクト、タイプ、クラス、属性、メソッドの実装を提供します。たとえば、vCenter Server プラグインは vCenter Server API のすべてのオブジェクトを Orchestrator の JavaScript API の JavaScript オブジェクトとして表示します。vCenter Server API が定義するすべてのクラス、メソッド、属性は JavaScript 表現で表示されます。vCenter Server のスクリプト クラス、およびスクリプト クラスが Orchestrator のスクリプト関数で定義するメソッドと属性は使用することができます。

たとえば、**VC:VirtualMachine** ファインダによって vCenter Server API で **VirtualMachine** 管理対象オブジェクト タイプが見つかったと、Orchestrator の JavaScript API に **VcVirtualMachine** JavaScript クラスとして表示されます。Orchestrator の JavaScript API 内の **VcVirtualMachine** JavaScript クラスは、vCenter Server API 内のすべての同じメソッドと属性を **VirtualMachine** 管理対象オブジェクトとして定義します。

Orchestrator プラグインは、プラグイン テクノロジーのオブジェクト、タイプ、クラス、属性、メソッドを、**vso.xml** ファイル内の **<scripting-objects>** 要素にある同等の Orchestrator JavaScript のオブジェクト、タイプ、クラス、属性、メソッドにマップします。

イベント ハンドラの役割

イベントは変更されたプラグイン テクノロジーのオブジェクトの状態または属性で、Orchestrator によって検出されます。Orchestrator はイベント ハンドラを実装してイベントを監視します。

Orchestrator プラグインを使用すると、プラグイン テクノロジーのイベントを異なる方法で監視できます。Orchestrator プラグイン API を使用すると、以下のタイプのイベント ハンドラを作成して、プラグイン テクノロジーのイベントを監視できます。

リスナー

プラグイン テクノロジーのオブジェクトの状態に変更がないか受動的に監視します。プラグイン テクノロジーまたはプラグインの実装では、リスナーが監視するイベントが定義されます。リスナーはイベントを開始しませんが、イベントが発生すると Orchestrator に通知します。リスナーはプラグイン テクノロジーをポーリン

グするか、プラグイン テクノロジーから通知を受信することにより、イベントを検出します。イベントが発生すると、イベントを待機している Orchestrator のポリシーまたはワークフローは、Orchestrator サーバで操作を開始して応答することができます。リスナーコンポーネントの設定はオプションです。

ポリシー

プラグイン テクノロジーの特定のイベントを監視し、イベントが発生した場合に Orchestrator サーバで操作を開始します。ポリシーではポリシー トリガとポリシー ゲージを監視できます。ポリシー トリガはプラグイン テクノロジーのイベントを定義します。イベントが発生すると、実行中のポリシーによって Orchestrator サーバで操作（ワークフローの実行など）が開始されます。ポリシー ゲージはプラグイン テクノロジーのオブジェクトの属性値の範囲を定義します。この範囲を超過すると、Orchestrator は操作を開始します。ポリシーの設定はオプションです。

ワークフロー トリガ

実行中のワークフローに待機イベント要素が含まれていてその要素に達した場合、実行がサスペンドして、プラグイン テクノロジーのイベントが発生するまで待機状態になります。ワークフロー トリガでプラグイン テクノロジーのイベントを定義して、ワークフローの待機要素を待機させます。ワークフロー トリガはウォッチャーに登録して使用します。ワークフロー トリガの設定はオプションです。

ウォッチャー

「ウォッチ」ワークフローはワークフローの待機イベント要素の代わりに、プラグイン テクノロジーの特定のイベントをトリガします。ウォッチャーはイベントが発生するとそのイベントを待機しているワークフローに通知します。ウォッチャーの設定はオプションです。

プラグインの内容と構造

Orchestrator のプラグインには、標準のコンポーネント セットが含まれている必要があります。また、標準のファイル構造に準拠している必要があります。プラグインを標準のファイル構造に準拠させるには、特定のフォルダとファイルを含める必要があります。

Orchestrator プラグインを作成するには、そのプラグインを使用するテクノロジー内のオブジェクトに対するアクセス方法と対話方法を定義する必要があります。また、プラグインを使用するテクノロジーのすべてのオブジェクトと関数を、**vso.xml** ファイル内の対応する Orchestrator のオブジェクトと関数にマップする必要があります。

vso.xml ファイルには、Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照が含まれている必要があります。プラグインを使用するテクノロジー内でプラグインによって検出されるすべてのオブジェクトに対して、一意の ID を指定する必要があります。**vso.xml** ファイルの **finder** 要素とオブジェクト要素で、オブジェクト名を定義します。

プラグインは、標準の Java アーカイブ ファイル (JAR) として提供することも、ZIP ファイルとして提供することもできますが、いずれの場合も、ファイル名に **.dar** という拡張子を付加する必要があります。

注意 Orchestrator コントロール センターを使用して、DAR ファイルを Orchestrator サーバにインポートすることができます。

■ vso.xml ファイル内でのアプリケーション マッピングの定義

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

■ vso.xml プラグイン定義ファイルの形式

vso.xml ファイルは、Orchestrator サーバがプラグイン テクノロジーと連携する方法を定義します。Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照を **vso.xml** ファイルに含める必要があります。

■ プラグイン オブジェクトの命名

プラグインがプラグイン テクノロジーにおいて検出する各オブジェクトに対し、一意の識別子を与える必要があります。オブジェクト名は、**vso.xml** ファイルの **<finder>** 要素および **<object>** 要素に定義します。

■ プラグイン オブジェクトの命名規則

プラグイン内のすべてのオブジェクトに名前を付ける場合は、Java クラスの命名規則に従ってください。

■ プラグインのファイル構造

プラグインは、標準のファイル構造に準拠するようにし、特定のフォルダとファイルを含める必要があります。プラグインは、標準の Java アーカイブ ファイル (JAR) または ZIP ファイルとして提供します。ファイル名には **.dar** という拡張子を付加する必要があります。

vso.xml ファイル内でのアプリケーション マッピングの定義

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

vso.xml ファイルは以下の情報を Orchestrator サーバに提供します。

- プラグインのバージョン、名前、および説明
- プラグイン テクノロジーのクラスおよび関連するプラグイン アダプタへのリファレンス
- Orchestrator サーバが起動するときにプラグインを初期化します
- プラグイン テクノロジー内のオブジェクトのタイプを表すスクリプト タイプ
- オブジェクトが Orchestrator インベントリ内で表示される方法を定義するオブジェクト タイプ間の関係
- プラグイン テクノロジー内のオブジェクトおよび操作を Orchestrator JavaScript API の機能およびオブジェクト タイプにマップするスクリプト クラス
- 特定のタイプのすべてのオブジェクトに適用される定数値のリストを定義する列挙
- Orchestrator がプラグイン テクノロジー内で監視するイベント

vso.xml ファイルは Orchestrator プラグインの XML スキーマ定義に適合する必要があります。スキーマ定義には VMware サポート サイトからアクセスできます。

```
http://www.vmware.com/support/orchestrator/plugin-4-1.xsd
```

vso.xml ファイルのすべての要素についての説明は、[\[vso.xml プラグイン定義ファイルの要素\]](#) を参照してください。

vso.xml プラグイン定義ファイルの形式

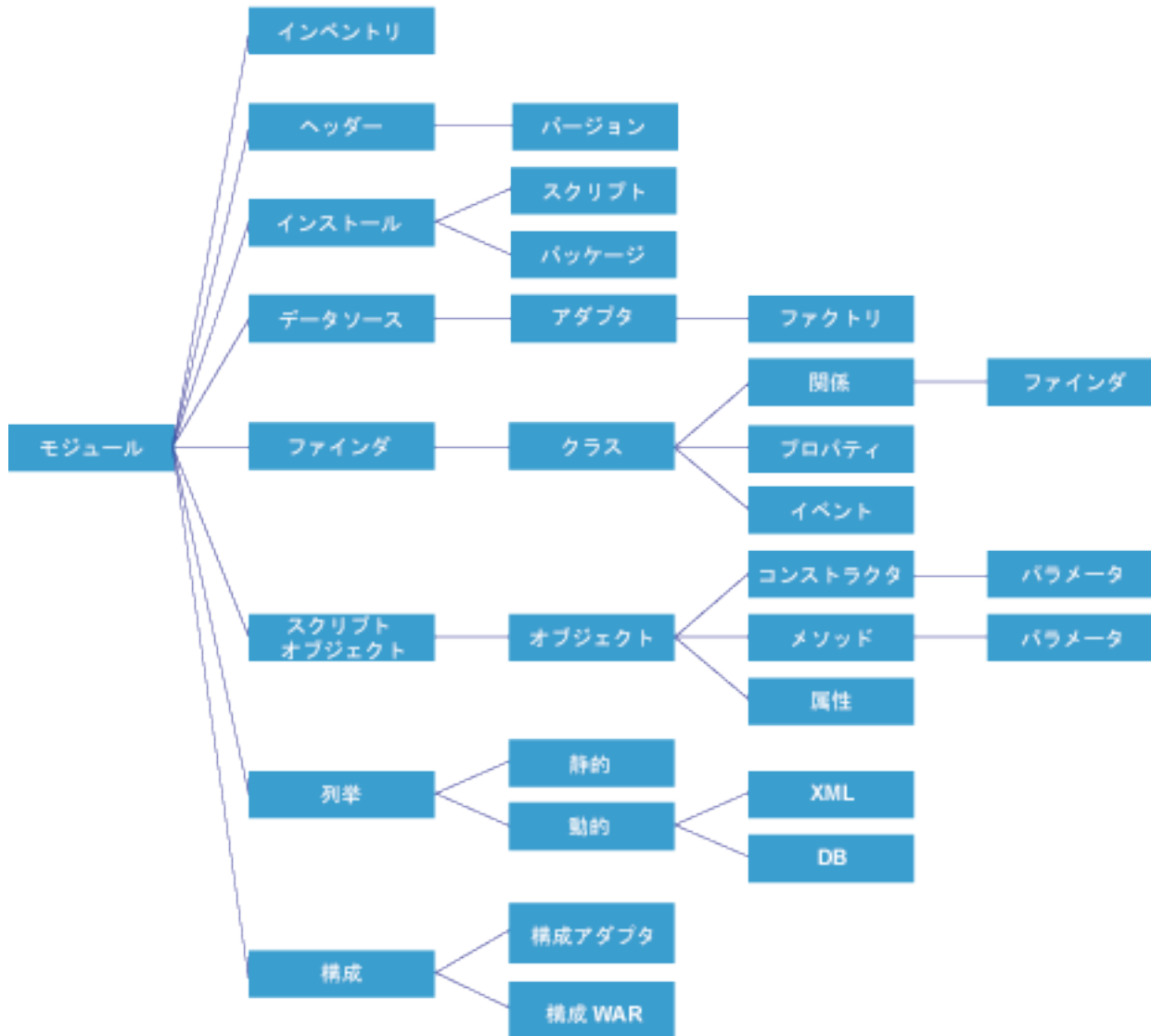
vso.xml ファイルは、Orchestrator サーバがプラグイン テクノロジーと連携する方法を定義します。Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照を **vso.xml** ファイルに含める必要があります。

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

プラグインのオープン アーキテクチャおよび標準化された実装の一部として、**vso.xml** ファイルは標準形式に従う必要があります。

次の図は、**vso.xml** プラグイン定義ファイルの形式と、要素が互いにどのようにネストされているかを示しています。

図 1-4. vso.xml プラグイン定義ファイルの形式



プラグイン オブジェクトの命名

プラグインがプラグインテクノロジーにおいて検出する各オブジェクトに対し、一意の識別子を与える必要があります。オブジェクト名は、**vso.xml** ファイルの **<finder>** 要素および **<object>** 要素に定義します。

ファクトリ実装で定義するファインダ操作では、プラグインテクノロジーのオブジェクトを検索します。プラグインがオブジェクトを検索するとき、Orchestrator のワークフローでそれらを使用して、それらを 1 つのワークフロー要素から別の要素へ渡すことができます。オブジェクトに一意の識別子を与えると、ワークフローにおいて要素間でのそれらの引き渡しが可能になります。

Orchestrator サーバは、サーバが処理する各オブジェクトのタイプおよび識別子のみを保存し、Orchestrator がオブジェクトを取得した場所または方法についての情報は保存しません。プラグインから取得したオブジェクトを追跡できるように、プラグインの実装においてオブジェクトを一貫性を持って命名する必要があります。

ワークフローの実行中に Orchestrator サーバが停止した場合、サーバの再起動時、ワークフローはサーバが停止したときに実行していたワークフロー要素から再開します。ワークフローは、サーバが停止したときに要素が処理していたオブジェクトを取得するために、識別子を使用します。

プラグイン オブジェクトの命名規則

プラグイン内のすべてのオブジェクトに名前を付ける場合は、Java クラスの命名規則に従ってください。

重要 ワークフロー エンジンによるデータのシリアル化の実行手段に基づいて、オブジェクト名には以下の文字列を使用しないでください。これらの文字列をオブジェクト識別子に使用すると、ワークフロー エンジンによるワークフローの解析が不正確になり、ワークフローを実行したときに予期せぬ動作が発生する可能性があります。

- `#;`
- `#,`
- `#=`

オブジェクトのプラグインに名前を付ける場合は、これらのガイドラインを使用してください。

- 名前の各単語には大文字の頭文字を使用してください。
- 単語の区切りにスペースを使用しないでください。
- 文字は、**A** から **Z** および **a** から **z** の標準文字のみを使用してください。
- 特殊文字（アクセントなど）を使用しないでください。
- 名前の最初の文字に数字を使用しないでください。
- できるだけ 10 文字未満にしてください。

オブジェクト タイプごとの規則は、[表 1-5](#) に記載されています。

表 1-5. プラグイン オブジェクトの命名規則

オブジェクト タイプ	命名規則
プラグイン	<ul style="list-style-type: none"> ■ vso.xml ファイルの <module> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ 一意のものにしてください。Orchestrator サーバで同じ名前の 2 つのプラグインを実行することはできません。
ファインダ オブジェクト	<ul style="list-style-type: none"> ■ vso.xml ファイルの <finder> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ プラグイン内で一意のものにしてください。 <p>Orchestrator は、Orchestrator のスクリプト API のファインダ オブジェクト タイプのオブジェクト名にプラグイン名とコロンを追加します。たとえば、vCenter Server プラグインの VirtualMachine オブジェクトタイプは、Orchestrator のスクリプト API では VC:VirtualMachine と表示されます。</p>
オブジェクトのスクリプティング	<ul style="list-style-type: none"> ■ vso.xml ファイルの <scripting-object> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ Orchestrator サーバ内で一意のものにしてください。 ■ スクリプト オブジェクトには、同じ名前のファインダ オブジェクトやその他のオブジェクトのスクリプト オブジェクトとの混乱を避けるために、スクリプト オブジェクト名の前に常にプラグインの名前が付きますが、コロンは付きません。たとえば、vCenter Server プラグインの VirtualMachine クラスには、Orchestrator のスクリプト API が VcVirtualMachine クラスとして表示されます。

プラグインのファイル構造

プラグインは、標準のファイル構造に準拠するようにし、特定のフォルダとファイルを含める必要があります。プラグインは、標準の Java アーカイブ ファイル (JAR) または ZIP ファイルとして提供します。ファイル名には **.dar** という拡張子を付加する必要があります。

DAR アーカイブのコンテンツでは、次のフォルダ構造と命名規則を使用する必要があります。

表 1-6. DAR アーカイブの構造

フォルダ	説明
<plug-in_name>\VSO-INF\	プラグイン テクノロジーのオブジェクトの Orchestrator オブジェクトへのマッピングを定義する vso.xml ファイルを格納します。 VSO-INF フォルダと vso.xml ファイルは必須です。
<plug-in_name>\lib\	プラグイン テクノロジーのバイナリが含まれる JAR ファイルを格納します。また、アダプタ、ファクトリ、通知ハンドラ、およびその他のインターフェイスの実装がプラグインに含まれる JAR ファイルも格納します。 lib フォルダと JAR ファイルは必須です。
<plug-in_name>\resources\	プラグインに必要なリソース ファイルを格納します。 resources フォルダには次の要素タイプを格納できます。 <ul style="list-style-type: none"> ■ イメージファイル。Orchestrator の [インベントリ] タブでプラグインのオブジェクトを表します。 ■ スクリプト。プラグイン起動時の初期化動作を定義します。 ■ Orchestrator パッケージ。プラグインを使用してアクセスするオブジェクトと対話するカスタム ワークフロー、アクション、および他のリソースを含めることができます。 リソースは、サブフォルダを使用して整理できます（例： resources\images\ 、 resources\scripts\ 、 resources\packages\ ）。 resources フォルダの使用はオプションです。

Orchestrator コントロール センターを使用して、DAR ファイルを Orchestrator サーバにインポートします。

Orchestrator プラグイン API リファレンス

Orchestrator プラグイン API は、プラグインを作成するために **IPluginAdaptor** および **IPluginFactory** の実装を開発する際に、実装および拡張する Java インターフェイスおよびクラスを定義します。

すべてのクラスは、記載がないかぎり **ch.dunes.vso.sdk.api** パッケージに含まれます。

IAop インターフェイス

IAop インターフェイスは、プラグインを使用するテクノロジーでオブジェクトのプロパティを取得および設定する方法を提供します。

```
public interface IAop
```

IAop インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)</code>	<code>java.lang.Object</code>	プラグインで指定のオブジェクトからプロパティを取得します。
<code>set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)</code>	<code>Void</code>	プラグインで指定のオブジェクトにプロパティを設定します。

IDynamicFinder インターフェイス

IDynamicFinder インターフェイスは、ファインダの ID とプロパティを **vso.xml** ファイルに定義する代わりに、ID とプロパティをプログラムで戻します。

IDynamicFinder インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getIdAccessor(java.lang.String type)</code>	<code>java.lang.String</code>	オブジェクト ID をプログラムで取得するための OGNL 式を提供します。
<code>getProperties(java.lang.String type)</code>	<code>java.util.List<SDKFinderProperty></code>	オブジェクト プロパティのリストをプログラムで提供します。

IPluginAdaptor インターフェイス

IPluginAdaptor インターフェイスは、プラグイン ファクトリ、イベント、およびウォッチャーを管理するために実装します。**IPluginAdaptor** インターフェイスはプラグインと Orchestrator サーバの間のアダプタを定義します。

IPluginAdaptor インスタンスはセッション管理の役割を担っています。**IPluginAdaptor** インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>addWatcher(PluginWatcher watcher)</code>	Void	特定のイベントを監視するウォッチャーを追加します
<code>createPluginFactory(java.lang.String sessionId, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)</code>	IPluginFactory	<p>IPluginFactory インスタンスを作成します。Orchestrator サーバはファクトリを使用して、オブジェクト ID や他のオブジェクトとの関係などを基準にしてプラグイン テクノロジーからオブジェクトを取得します。</p> <p>セッション ID によって実行中のセッションを特定できます。たとえば、ユーザーは 2 つの異なる Orchestrator クライアントにログインして 2 つのセッションを同時に実行できます。</p> <p>同様に、ワークフローを開始すると、ワークフローの開始に使用したクライアントとは無関係のセッションが作成されます。Orchestrator クライアントを閉じた場合でもワークフローは実行し続けます。</p>
<code>installLicenses(PluginLicense[] licenses)</code>	Void	VMware が提供する標準プラグインのライセンス情報をインストールします
<code>registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	インベントリ内の要素にトリガとゲージを設定します
<code>removeWatcher(java.lang.String watcherId)</code>	Void	ウォッチャーを削除します
<code>setPluginName(java.lang.String pluginName)</code>	Void	vso.xml ファイルからプラグイン名を取得します
<code>setPluginPublisher(IPluginPublisher pluginPublisher)</code>	Void	プラグインの公開者を設定します
<code>uninstallPluginFactory(IPluginFactory plugin)</code>	Void	プラグインファクトリをアンインストールします。
<code>unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	インベントリ内の要素からトリガとゲージを除去します

IPluginEventPublisher インターフェイス

IPluginEventPublisher インターフェイスは、Orchestrator ポリシーを監視するためのイベント通知バスにゲージとトリガを公開します。

IPluginEventPublisher インスタンスはプラグイン アダプタ実装に直接作成したり、別々のイベント ジェネレータ クラスに作成したりできます。

IPluginEventPublisher インターフェイスを実装して、Orchestrator ポリシー エンジンにプラグイン テクノロジーのイベントを公開することができます。プラグイン テクノロジーのオブジェクト内のポリシー トリガとゲージを設定する方法や、それらのオブジェクトのイベントをリッスンするイベント リスナーを設定する方法を作成できます。

ポリシーは、プラグイン テクノロジー内のオブジェクトを監視するためのゲージまたはトリガのいずれかに適用することができます。ポリシー ゲージ は、オブジェクトの属性を監視し、オブジェクトの値が指定の値を超えた場合に Orchestrator サーバのイベントをプッシュします。ポリシーは監視オブジェクトをトリガし、定義されているイベントがオブジェクトで発生した場合に、Orchestrator サーバのイベントをプッシュします。ポリシー ゲージおよびトリガを **IPluginEventPublisher** インスタンスに登録することで、Orchestrator ポリシーはこれらを監視できるようになります。

IPluginEventPublisher インターフェイスは次のメソッドを定義します。

タイプ	戻り値	説明
<code>pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)</code>	Void	監視するポリシーのためのゲージを公開します。 次のパラメータを取ります。 <ul style="list-style-type: none"> ■ type : 監視するオブジェクトのタイプ。 ■ id : 監視するオブジェクトの ID。 ■ gaugeName : このゲージの名前。 ■ deviceName : ゲージが監視する属性のタイプの名前。 ■ gaugeValue : ゲージがオブジェクトを監視する対象の値。
<code>pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)</code>	Void	監視するポリシーのためのトリガを公開します。 次のパラメータを取ります。 <ul style="list-style-type: none"> ■ type : 監視するオブジェクトのタイプ。 ■ id : 監視するオブジェクトの ID。 ■ triggerName : このトリガの名前。 ■ additionalProperties : 監視するトリガの追加プロパティ。

IPluginFactory インターフェイス

IPluginAdaptor は **IPluginFactory** インスタンスを戻します。**IPluginFactory** インスタンスはプラグイン アプリケーション内でコマンドを実行し、Orchestrator の操作を実行する対象のオブジェクトを検出します。

IPluginFactory インターフェイスは次のフィールドを定義します。

`static final java.lang.String RELATION_CHILDREN`

IPluginFactory インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>executePluginCommand(java.lang.String cmd)</code>	<code>Void</code>	プラグインを使用してコマンドを実行します。このメソッドを使用しないことをお勧めします。
<code>find(java.lang.String type, java.lang.String id)</code>	<code>java.lang.Object</code>	プラグインを使用してオブジェクトを検出します。オブジェクトを ID およびタイプによって特定します。
<code>findAll(java.lang.String type, java.lang.String query)</code>	<code>QueryResult</code>	プラグインを使用して、クエリ文字列に一致する特定のタイプのオブジェクトを検出します。クエリの構文はプラグインの IPluginFactory 実装内に定義します。クエリ構文を定義しない場合は、 findAll() によって、指定されたタイプのすべてのオブジェクトが戻されます。
<code>findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>java.util.List</code>	オブジェクトが子を持つかどうかを判別します。
<code>hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>HasChildrenResult</code>	特定の親と特定の関係をもつすべての子を検出します。
<code>invalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	タイプと ID によってオブジェクトを無効化します。
<code>void invalidateAll()</code>	<code>Void</code>	キャッシュ内のすべてのオブジェクトを無効化します。

IPluginNotificationHandler インターフェイス

IPluginNotificationHandler は、Orchestrator がプラグインを介してアクセスするオブジェクト上で発生するさまざまなタイプのイベントを Orchestrator に通知するためのメソッドを定義します。

IPluginNotificationHandler インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getSessionID()</code>	<code>java.lang.String</code>	現在のセッション ID を戻します。
<code>notifyElementDeleted(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	特定のタイプおよび ID のオブジェクトが削除されたことをシステムに通知します
<code>notifyElementInvalidate(java.lang.String type, java.lang.String id)</code>	<code>Void</code>	オブジェクトの関係が変更されたことをシステムに通知します。 notifyElementInvalidate() メソッドを使用すれば、オブジェクトを無効化した関係の変更だけでなく、オブジェクト間のすべての関係の変更を Orchestrator に通知できます。たとえば、子オブジェクトを親に追加すると、2 つのオブジェクト間の関係が変更されたことになります。

メソッド	戻り値	説明
<code>notifyElementUpdated(java.lang.String type, java.lang.String id)</code>	Void	オブジェクトの属性が変更されたことをシステムに通知します
<code>notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)</code>	Void	現在のモジュールに関するエラー メッセージを公開します

IPluginPublisher インターフェイス

IPluginPublisher インターフェイスは長期実行ワークフローの待機イベント要素を監視するために、イベント通知バスにウォッチャー イベントを公開します。

ワークフローのトリガがプラグイン テクノロジーのイベントを開始すると、トリガを監視し **IPluginPublisher** インスタンスで登録されたプラグイン ウォッチャーは、イベントが発生している待機中のワークフローに通知を行います。

IPluginPublisher インターフェイスは次のメソッドを定義します。

タイプ	値	説明
<code>pushWatcherEvent(java.lang.String id, java.util.Properties properties)</code>	Void	イベント通知バスにウォッチャー イベントを公開します

WebConfigurationAdaptor インターフェイス

WebConfigurationAdaptor インターフェイスは **IConfigurationAdaptor** を実装し、プラグインの構成タブにある Web アプリケーションを見つけてインストールするメソッドを定義します。

注意 Orchestrator 4.1 以降、**WebConfigurationAdaptor** インターフェイスは廃止されました。Web アプリケーションを構成に追加するには、**IConfigurationAdaptor** を実装し、**vso.xml** ファイルの **configuration-war** 属性を使用して Web アプリケーションを特定します。

WebConfigurationAdaptor インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getWebAppContext()</code>	文字列	構成タブにある Web アプリケーションの WAR ファイルを見つけます。 /webapps ディレクトリから WAR ファイルへのパスと名前を文字列として WAR ファイルに返します。
<code>setWebConfiguration(boolean webConfiguration)</code>	ブール値	構成タブの内容に Web アプリケーションが定義されているかどうかを特定します。

PluginTrigger クラス

PluginTrigger クラスは、ワークフローの待機イベント要素の代わりに、プラグイン テクノロジーにおいて監視するオブジェクトおよびイベントについての情報を取得する、トリガ モジュールを作成します。

PluginTrigger クラスは、監視するオブジェクトの名前やタイプ、イベントの特性およびタイムアウト期間を取得または設定します。

ワークフローの待機イベント要素が使用するために実装する専用の **PluginTrigger** クラスを作成します。

Orchestrator ポリシーのポリシー トリガを、イベントを定義し **IPluginEventPublisher.pushTrigger()** メソッドを実装するクラスに定義します。

```
public class PluginTrigger
    extends java.lang.Object
    implements java.io.Serializable
```

PluginTrigger クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getModuleName()</code>	<code>java.lang.String</code>	トリガ モジュールの名前を取得します。
<code>getProperties()</code>	<code>java.util.Properties</code>	トリガのプロパティのリストを取得します。
<code>getSdkId()</code>	<code>java.lang.String</code>	プラグイン テクノロジーで監視するオブジェクトの ID を取得します。
<code>getSdkType()</code>	<code>java.lang.String</code>	プラグイン テクノロジーで監視するオブジェクトのタイプを取得します。
<code>getTimeout()</code>	<code>Long</code>	トリガのタイムアウト期間を取得します。
<code>setModuleName(java.lang.String moduleName)</code>	<code>Void</code>	トリガ モジュールの名前を設定します。
<code>setProperties(java.util.Properties properties)</code>	<code>Void</code>	トリガのプロパティのリストを設定します。
<code>setSdkId(java.lang.String sdkId)</code>	<code>Void</code>	プラグイン テクノロジーで監視するオブジェクトの ID を設定します。
<code>setSdkType(java.lang.String sdkType)</code>	<code>Void</code>	プラグイン テクノロジーで監視するオブジェクトのタイプを設定します。
<code>setTimeout(long timeout)</code>	<code>Void</code>	タイムアウト期間を秒で設定します。負の値は、タイムアウトを無効にします。

コンストラクタ

- `PluginTrigger()`
- `PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)`

PluginWatcher クラス

PluginWatcher クラスは、長時間実行ワークフローの待機イベント要素の代わりに、プラグイン テクノロジーの定義済みイベント用のトリガ モジュールをウォッチします。

PluginWatcher クラスは、プラグイン ウォッチャー インスタンスの作成に使用するコンストラクタを定義します。**PluginWatcher** クラスは、監視するワークフロー トリガの名前およびタイムアウト期間を取得および設定する方法を定義します。

```
public class PluginWatcher
extends java.lang.Object
implements java.io.Serializable
```

PluginWatcher クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getId()</code>	<code>java.lang.String</code>	トリガの ID を取得します
<code>getModuleName()</code>	<code>java.lang.String</code>	トリガ モジュール名を取得します
<code>getTimeoutDate()</code>	<code>Long</code>	トリガのタイムアウト日を取得します
<code>getTrigger()</code>	<code>Void</code>	トリガを取得します
<code>setId(java.lang.String id)</code>	<code>Void</code>	トリガの ID を設定します
<code>setTimeoutDate()</code>	<code>Void</code>	トリガのタイムアウト日を設定します

コンストラクタ

`PluginWatcher(PluginTrigger trigger)`

QueryResult クラス

QueryResult クラスには、Orchestrator がプラグインを介してアクセスするオブジェクトに対する、**find** クエリの結果が含まれます。

```
public class QueryResult
extends java.lang.Object
implements java.io.Serializable
```

結果の合計数がクエリが返す結果の数を超えた場合、**totalCount** 値は **QueryResult** が返す数字よりを大きくなる可能性があります。クエリが返す結果の数は、**vso.xml** ファイルのクエリの構文で定義されます。

QueryResult クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>addElement(java.lang.Object element)</code>	Void	QueryResult に要素を追加します
<code>addElements(java.util.List elements)</code>	Void	QueryResult に要素のリストを追加します
<code>getElements()</code>	java.util.List	プラグイン アプリケーションから要素を取得します
<code>getTotalCount()</code>	Long	プラグイン テクノロジーで使用可能なすべての要素の合計数を取得します
<code>isPartialResult()</code>	Boolean	取得した結果が完全であるかを判断します
<code>removeElement(java.lang.Object element)</code>	Void	プラグイン テクノロジーから要素を削除します
<code>setElements(java.util.List elements)</code>	Void	プラグイン テクノロジーに要素を設定します
<code>setTotalCount(long totalCount)</code>	Void	プラグイン テクノロジーで使用可能な要素の合計数を設定します

コンストラクタ

- `QueryResult()`
- `QueryResult(java.util.List ret)`
- `QueryResult(java.util.List elements, long totalCount)`

SDKFinderProperty クラス

SDKFinderProperty クラスは、プラグインされたテクノロジー内で Orchestrator ファインダ オブジェクトによって検出されたオブジェクトでプロパティを取得および設定するためのメソッドを定義します。

IDynamicFinder.getProperties メソッドは **SDKFinderProperty** オブジェクトを返します。

```
public class SDKFinderProperty
extends java.lang.Object
```

SDKFinderProperty クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getAttributeName()</code>	java.lang.String	オブジェクトの属性名を取得する
<code>getBeanProperty()</code>	java.lang.String	Java bean からプロパティを取得する
<code>getDescription()</code>	java.lang.String	オブジェクトの説明を取得する
<code>getDisplayName()</code>	java.lang.String	オブジェクトの表示名を取得する
<code>getPossibleResultType()</code>	java.lang.String	ファインダが返す結果の可能性のあるタイプを取得する
<code>getPropertyAccessor()</code>	java.lang.String	オブジェクトのプロパティ アクセサを取得する

メソッド	戻り値	説明
<code>getPropertyAccessorTree()</code>	<code>java.lang.Object</code>	オブジェクトのプロパティ アクセサ ツリーを取得する
<code>isHidden()</code>	ブール値	オブジェクトの表示/非表示を切り替える
<code>isShowInColumn()</code>	ブール値	データベース カラム内のオブジェクトの表示/非表示を切り替える
<code>isShowInDescription()</code>	ブール値	オブジェクトの説明の表示/非表示を切り替える
<code>setAttributeName(java.lang.String attributeName)</code>	Void	オブジェクトの属性名を設定する
<code>setBeanProperty(java.lang.String beanProperty)</code>	Void	Java bean にプロパティを設定する
<code>setDescription(java.lang.String description)</code>	Void	オブジェクトの説明を設定する
<code>setDisplayName(java.lang.String displayName)</code>	Void	オブジェクトの表示名を設定する
<code>setHidden(boolean hidden)</code>	Void	オブジェクトの表示/非表示を切り替える
<code>setPossibleResultType(java.lang.String possibleResultType)</code>	Void	ファインダが返す結果の可能性のあるタイプを設定する
<code>setPropertyAccessor(java.lang.String propertyAccessor)</code>	Void	オブジェクトのプロパティ アクセサを設定する
<code>setPropertyAccessorTree(java.lang.Object propertyAccessorTree)</code>	Void	オブジェクトのプロパティ アクセサ ツリーを設定する
<code>setShowInColumn(boolean showInTable)</code>	Void	データベース カラム内のオブジェクトの表示/非表示を切り替える
<code>setShowInDescription(boolean showInDescription)</code>	Void	オブジェクトの説明の表示/非表示を切り替える

コンストラクタ

`SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)`

PluginExecutionException クラス

PluginExecutionException クラスは、プラグインが操作を実行する際に例外が発生した場合に、エラー メッセージを返します。

```
public class PluginExecutionException
extends java.lang.Exception
implements java.io.Serializable
```

`PluginExecutionException` クラスは、`class java.lang.Throwable` から次のメソッドを継承します。

`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`、`printStackTrace`、`printStackTrace`、`setStackTrace`、`toString`
`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`

コンストラクタ

`PluginExecutionException(java.lang.String message)`

PluginOperationException クラス

`PluginOperationException` クラスは、プラグイン操作中に発生したエラーを処理します。

```
public class PluginOperationException
extends java.lang.RuntimeException
implements java.io.Serializable
```

`PluginOperationException` クラスは、`class java.lang.Throwable` から次のメソッドを継承します。

`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`、`printStackTrace`、`printStackTrace`、`setStackTrace`、`toString`

コンストラクタ

`PluginOperationException(java.lang.String message)`

HasChildrenResult 列挙

`HasChildrenResult` 列挙は、特定の親が子を持つかどうかを宣言します。

`IPluginFactory.hasChildrenInRelation` メソッドは `HasChildrenResult` オブジェクトを返します。

```
public enum HasChildrenResult
extends java.lang.Enum<HasChildrenResult>
implements java.io.Serializable
```

`HasChildrenResult` 列挙は次の定数を定義します。

- `public static final HasChildrenResult Yes`
- `public static final HasChildrenResult No`
- `public static final HasChildrenResult Unknown`

`HasChildrenResult` 列挙は次のメソッドを定義します。

メソッド	戻り値	説明
<code>getValue()</code>	整数	次のいずれかの値を返します。 1 親が子を持ちます -1 親が子を持ちません 0 不明または無効なパラメータ
<code>valueOf(java.lang.String name)</code>	<code>static HasChildrenResult</code>	指定された名前を持つこのタイプの列挙定数を返します。String は、このタイプの列挙定数を宣言するために使用された ID と正確に一致する必要があります。列挙名にスペース文字を使用しないでください。
<code>values()</code>	<code>static HasChildrenResult[]</code>	この列挙型の定数を含む配列を、宣言される順に返します。このメソッドは、次のようにすべての定数にわたって繰り返すことができます。 <pre>for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);</pre>

`HasChildrenResult` 列挙は `class java.lang.Enum` から次のメソッドを継承します。

`clone`、`compareTo`、`equals`、`finalize`、`getDeclaringClass`、`hashCode`、`name`、`ordinal`、`toString`、`valueOf`

ScriptingAttribute 注釈タイプ

`ScriptingAttribute` 注釈タイプは、スクリプティングでプロパティとして使用される、プラグインされたテクノロジー内のオブジェクトの属性に注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value={METHOD, FIELD})
public @interface ScriptingAttribute
```

`ScriptingAttribute` 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

ScriptingFunction 注釈タイプ

`ScriptingFunction` 注釈タイプは、スクリプティングでプロパティとして使用されるメソッドに注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value={METHOD, CONSTRUCTOR})
public @interface ScriptingFunction
```

ScriptingFunction 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

ScriptingParameter 注釈タイプ

ScriptingParameter 注釈タイプは、スクリプティングでプロパティとして使用されるパラメータに注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

ScriptingParameter 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

vso.xml プラグイン定義ファイルの要素

vso.xml ファイルには標準要素のセットが含まれています。一部の要素は必須ですが、オプションの要素もあります。それぞれの要素には、Orchestrator のオブジェクトおよび操作にマップする、オブジェクトおよび操作の値を定義する属性があります。

また、要素は 0 個以上の子要素を持つことができます。さらに子要素は親要素を定義します。同じ子要素が複数の親要素に存在してもかまいません。たとえば、**description** 要素は子要素を持ちませんが、多くの親要素 **module**、**example**、**trigger**、**gauge**、**finder**、**constructor**、**method**、**object**、および **enumeration** の子要素として表示されます。

次に示す各要素の定義では、要素の属性、親要素、および子要素を示します。

module 要素

module は、プラグイン オブジェクトのセットを Orchestrator で使用できるように記述します。

モジュールには、プラグイン テクノロジーからのデータを Java クラスにマップする方法、バージョンing、モジュールの展開方法、および Orchestrator インベントリでのプラグインの表示方法などの情報が含まれます。

<module> 要素はオプションです。**<module>** 要素には、以下の属性が含まれています。

属性	値	説明
name	文字列	プラグインのすべての <finder> 要素のタイプを定義します。これは必須属性です。
version	数値	プラグインの新しいバージョンにパッケージを再ロードするときに使用する、プラグイン バージョン番号。これは必須属性です。
build-number	数値	プラグインの新しいバージョンにパッケージを再ロードするときに使用する、プラグイン ビルド番号。これは必須属性です。

属性	値	説明
image	イメージ ファイル	Orchestrator インベントリに表示するアイコン。これは必須属性です。
display-name	文字列	Orchestrator インベントリに表示される名前。これはオプション属性です。
interface-mapping-allowed	true または false	インターフェイス マッピングは推奨されていません。これはオプション属性です。

表 1-7. 要素の階層

親要素	子要素
なし	<ul style="list-style-type: none"> ■ <description> ■ <installation> ■ <configuration> ■ <finder-datasources> ■ <inventory> ■ <finders> ■ <scripting-objects> ■ <enumerations>

description 要素

<description> 要素により、API Explorer ドキュメントに表示されるプラグインの要素の説明を指定することができます。

API Explorer ドキュメントに表示されるテキストを、<description> タグと </description> タグの間に追加します。

<description> 要素はオプションです。<description> 要素の属性はありません。

表 1-8. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ■ <module> ■ <example> ■ <trigger> ■ <gauge> ■ <finder> ■ <constructor> ■ <method> ■ <object> ■ <enumeration> 	なし

廃止された要素

<deprecated> 要素により、API Explorer ドキュメントで廃止されたオブジェクトとメソッドにマークを付けます。

API Explorer ドキュメントに表示されるテキストを、`<deprecated>` タグと `</deprecated>` タグの間に追加します。

`<deprecated>` 要素はオプションです。`<deprecated>` 要素の属性はありません。

表 1-9. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ■ <code><method></code> ■ <code><object></code> 	なし

url 要素

`<url>` 要素では、オブジェクトまたは列挙に関する外部ドキュメントの URL を指定できます。

URL は、`<url>` タグと `</url>` タグの間に指定します。

`<url>` 要素はオプションです。`<url>` 要素の属性はありません。

表 1-10. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ■ <code><enumeration></code> ■ <code><object></code> 	なし

installation 要素

`<installation>` 要素によって、サーバが起動したときにパッケージをインストールするかスクリプトを実行することができます。

`<installation>` 要素はオプションです。`<installation>` 要素には、次の属性が含まれています。

属性	値	説明
<code>mode</code>	<code>always</code> 、 <code>never</code> 、または <code>version</code>	<p><code>mode</code> 値を設定すると、Orchestrator サーバの起動時の動作は次のいずれかになります。</p> <ul style="list-style-type: none"> ■ アクションが常に実行される (<code>always</code>) ■ アクションがまったく実行されない (<code>never</code>) ■ サーバが新しいバージョンのプラグインを検出したときにアクションが実行される <p>これは必須属性です。</p>

表 1-11. 要素の階層

親要素	子要素
<code><module></code>	<code><action></code>

action 要素

<action> 要素は、Orchestrator サーバの起動時に実行されるアクションを指定します。

<action> 要素の属性は、プラグインが起動されたときの動作を定義する Orchestrator パッケージまたはスクリプトへのパスを指定します。

<action> 要素はオプションです。1 つのプラグインに対して設定できる **<action>** 要素の数に制限はありません。**<action>** 要素には、以下の属性が含まれています。

属性	値	説明
resource	文字列	dar ファイルのルートから Java パッケージまたはスクリプトへのパスです。これは必須属性です。
type	install-package または execute-script	指定された Orchestrator パッケージを Orchestrator サーバにインストールするか、または指定されたスクリプトを実行します。これは必須属性です。

表 1-12. 要素の階層

親要素	子要素
<installation>	なし

finder-datasource 要素

<finder-datasources> 要素は **<finder-datasource>** 要素のコンテナです。

<finder-datasources> 要素はオプションです。**<finder-datasources>** 要素の属性はありません。

表 1-13. 要素の階層

親要素	子要素
<module>	<finder-datasource>

finder-datasource 要素

<finder-datasource> 要素は、プラグイン用に作成された **IPluginAdaptor** 実装の Java クラス ファイルを参照します。

<finder-datasource> 要素を使用して、プラグインを使用するテクノロジーのオブジェクトに対する Orchestrator のアクセス方法を設定することができます。<finder-datasource> 要素により、ユーザーが作成するプラグイン アダプタの Java クラスが特定されます。プラグイン アダプタ クラスは、ユーザーが作成するプラグイン ファクトリをインスタンス化します。プラグイン ファクトリは、プラグインを使用するテクノロジー内のオブジェクトを検索する方法を定義します。<finder-datasource> 要素で、ファクトリが実行するファインディングメソッド呼び出しのタイムアウトを設定することができます。IPluginFactory インターフェイスの異なるファインディングメソッドに対して、異なるタイムアウトの値が適用されます。

<finder-datasource> 要素はオプションです。1 つのプラグインに対して設定できる <finder-datasources> 要素の数に制限はありません。<finder-datasource> 要素には、以下の属性が含まれています。

属性	値	説明
name	文字列	<finder> 要素の datasource 属性のデータソースを識別します。XML の id 属性に相当します。これは必須属性です。
adaptor-class	Java クラス	com.vmware.plugins.sample.Adaptor などのプラグイン アダプタを作成するために定義された IPluginAdaptor 実装を参照します。これは必須属性です。
concurrent-call	true (デフォルト値) または false	複数のユーザーが同じアダプタに同時にアクセスすることを許可します。プラグインで同時呼び出しがサポートされていない場合は、 concurrent-call を false に設定する必要があります。これはオプション属性です。
invoker-mode	direct (デフォルト値) または timeout	ファインディング関数のタイムアウトを設定します。値を direct に設定すると、ファインディング関数の呼び出しでタイムアウトが発生しなくなります。値を timeout に設定すると、ファインディングメソッドに対応するタイムアウト期間が Orchestrator サーバによって適用されます。これはオプション属性です。
anonymous-login-mode	never (デフォルト値) または always	ユーザーの名前とパスワードをプラグインに渡すかどうかを指定します。これはオプション属性です。
timeout-fetch-relation	数値 (デフォルトは 30 秒)	findRelation() からの呼び出しに適用されます。これはオプション属性です。
timeout-find-all	数値 (デフォルトは 60 秒)	findAll() からの呼び出しに適用されます。これはオプション属性です。
timeout-find	数値 (デフォルトは 60 秒)	find() からの呼び出しに適用されます。これはオプション属性です。

属性	値	説明
timeout-has-children-in-relation	数値 (デフォルトは 2 秒)	<code>findChildrenInRelation()</code> からの呼び出しに適用されます。これはオプション属性です。
timeout-execute-plugin-command	数値 (デフォルトは 30 秒)	<code>executePluginCommand()</code> からの呼び出しに適用されます。これはオプション属性です。

表 1-14. 要素の階層

親要素	子要素
<finder-datasources>	なし

inventory 要素

<inventory> 要素は Orchestrator クライアントの [インベントリ] ビューおよびオブジェクト選択ダイアログ ボックスに表示されるプラグインのための階層リストのルートを定義します。

<inventory> 要素はプラグイン アプリケーション内のオブジェクトを表すのではなく、Orchestrator スクリプト API のオブジェクトとしてのプラグインそのものを表します。

<inventory> 要素はオプションです。<inventory> 要素には、以下の属性が含まれています。

属性	値	説明
type	Orchestrator オブジェクト タイプ	オブジェクトの階層のルートを表す <finder> 要素のタイプ。これは必須属性です。

表 1-15. 要素の階層

親要素	子要素
<module>	なし

finders 要素

<finders> 要素はすべての <finder> 要素のコンテナです。

<finders> 要素はオプションです。<finders> 要素の属性はありません。

表 1-16. 要素の階層

親要素	子要素
<module>	<finder>

finder 要素

<finder> 要素は Orchestrator クライアント内でプラグインを介して検出されるオブジェクトのタイプを表します。

<finder> 要素はオブジェクト ファインダが示すオブジェクトを定義する Java クラスを特定します。**<finder>** 要素は、オブジェクトが Orchestrator クライアント インターフェイス内で表示される方法を定義します。また、このオブジェクトを示すために Orchestrator スクリプト API が定義するスクリプト オブジェクトを特定します。

ファインダは、異なるタイプのプラグイン テクノロジーによって使用されるオブジェクト形式どうしの間のインターフェイスの役割を果たします。

<finder> 要素はオプションです。1 つのプラグインに対して設定できる **<finder>** 要素の数に制限はありません。**<finder>** 要素は次の属性を定義します。

属性	値	説明
type	Orchestrator オブジェクト タイプ	ファインダによって示されるオブジェクトのタイプ。これは必須属性です。
datasource	<finder-datasource name> 属性	データソース refid を使用してオブジェクトを定義する Java クラスを特定します。これは必須属性です。
dynamic-finder	Java メソッド	ファインダの ID とプロパティを vso.xml ファイルに定義する代わりにプログラムで戻すために、 IDynamicFinder インスタンスに実装するカスタム ファインダ メソッドを定義します。これはオプション属性です。
hidden	true または false (デフォルト)	true の場合、Orchestrator クライアント内でファインダを非表示にします。これはオプション属性です。
image	グラフィック ファイルへのパス	Orchestrator クライアント内の階層リストにファインダを表示するための 16x16 のアイコン。これはオプション属性です。
java-class	Java クラスの名前	ファインダによって検出されてスクリプト オブジェクトにマップされるオブジェクトを定義する Java クラス。これはオプション属性です。
script-object	<scripting-object type> 属性	このファインダをマップする <scripting-object> タイプ (ある場合)。これはオプション属性です。

表 1-17. 要素の階層

親要素	子要素
<finders>	<ul style="list-style-type: none"> ■ <id> ■ <description> ■ <properties> ■ <default-sorting> ■ <inventory-children> ■ <relations> ■ <inventory-tabs> ■ <events>

properties 要素

<properties> 要素は、**<finder>****<property>** 要素のコンテナです。

<properties> 要素はオプションです。**<properties>** 要素の属性はありません。

表 1-18. 要素の階層

親要素	子要素
<finder>	<property>

property 要素

<property> 要素は、見つかったオブジェクトのプロパティを Java プロパティまたはメソッド呼び出しにマップします。

プラグイン ファクトリを実装して、処理するプラグイン ファクトリ実装のプロパティを取得するときに、**SDKFinderProperty** クラスのメソッドを呼び出すことができます。

オブジェクトのプロパティは、Orchestrator クライアントのビューに表示したり、非表示にしたりできます。また、列挙を使用してオブジェクトのプロパティを定義できます。

<property> 要素はオプションです。1 つのプラグインに対して設定できる **<property>** 要素の数に制限はありません。**<property>** 要素には、以下の属性が含まれています。

属性	値	説明
name	ファインダ名	FinderResult が要素の保存に使用する名前。これは必須属性です。
display-name	ファインダ名	表示されたプロパティ名。これはオプション属性です。
bean-property	プロパティ名	bean-property 属性を使用して、 get および set 操作を使用して取得するプロパティを特定します。 MyProperty という名前のプロパティを特定した場合、プラグインは getMyProperty および setMyProperty 操作を定義します。 bean-property または property-accessor のどちらか一方を設定できますが、両方を設定することはできません。これはオプション属性です。
property-accessor	オブジェクトからプロパティ値を取得するメソッド。	property-accessor 属性を使用すると、OGNL 式がオブジェクトのプロパティを検証するように定義できます。 bean-property または property-accessor のどちらか一方を設定できますが、両方を設定することはできません。これはオプション属性です。

属性	値	説明
show-in-column	true (デフォルト値) または false	true の場合、このプロパティは Orchestrator クライアントの結果テーブルが示します。これはオプション属性です。
show-in-description	true (デフォルト値) または false	true の場合、このプロパティは、オブジェクトの説明を示します。これはオプション属性です。
hidden	true または false (デフォルト)	true の場合、このプロパティはすべてのケースで非表示になります。これはオプション属性です。
linked-enumeration	列挙名	ファインダのプロパティを列挙にリンクします。これはオプション属性です。

表 1-19. 要素の階層

親要素	子要素
<properties>	子要素

relations 要素

<relations> 要素は、<finder><relation> 要素のコンテナです。

<relations> 要素はオプションです。<relations> 要素の属性はありません。

表 1-20. 要素の階層

親要素	子要素
<finder>	<relation>

relation 要素

<relation> 要素は、オブジェクトが他のオブジェクトに関係する方法を定義します。

<relation> 要素には関係名を定義します。

<relation> 要素はオプションです。1 つのプラグインに対して設定できる <relation> 要素の数に制限はありません。<relation> 要素には、以下の属性が含まれています。

属性	値	説明
name	関係名	この関係の名前。これは必須属性です。
type	Orchestrator オブジェクト タイプ	この関係によって別のオブジェクトに関係するオブジェクトのタイプ。これは必須属性です。
cardinality	to-one または to-many	1 対 1 または 1 対多数として、オブジェクト間の関係を定義します。これはオプション属性です。

表 1-21. 要素の階層

親要素	子要素
<relations>	なし

id 要素

<id> 要素は、ファインダが特定するオブジェクトの固有 ID を取得するためのメソッドを定義します。

<id> 要素はオプションです。<id> 要素には、以下の属性が含まれています。

属性	値	説明
accessor	メソッド名	accessor 属性によって、オブジェクトのプロパティを検証するための OGNL 式を定義できます。これは必須属性です。

表 1-22. 要素の階層

親要素	子要素
<finder>	なし

inventory-children 要素

<inventory-children> 要素は Orchestrator クライアントの [インベントリ] ビューおよびオブジェクト選択ボックス内のオブジェクトを表示するリストの階層を定義します。

<inventory-children> 要素はオプションです。<inventory-children> 要素の属性はありません。

表 1-23. 要素の階層

親要素	子要素
<finder>	<relation-link>

relation-link 要素

<relation-link> 要素は、[インベントリ] タブの親オブジェクトと子オブジェクト間の階層を定義します。

<relation-link> 要素はオプションです。1 つのプラグインに対して設定できる <relation-link> 要素の数の制限はありません。<relation-link> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	関係名	関係名への refid。これは必須属性です。

表 1-24. 要素の階層

親要素	子要素
<inventory-children>	なし

events 要素

`<events>` 要素は `<trigger>` および `<gauge>` 要素のコンテナです。

`<events>` 要素に含めることができるトリガまたはゲージの数に制限はありません。

`<events>` 要素はオプションです。`<events>` 要素の属性はありません。

表 1-25. 要素の階層

親要素	子要素
<code><finder></code>	<ul style="list-style-type: none"> ▪ <code><trigger></code> ▪ <code><gauge></code>

トリガ要素

`<trigger>` 要素はこのファインダで使用可能なトリガを宣言します。トリガを設定するには、`IPluginAdaptor` の `registerEventPublisher()` および `unregisterEventPublisher()` メソッドを実装する必要があります。

`<trigger>` 要素はオプションです。`<trigger>` 要素には、以下の属性が含まれています。

タイプ	値	説明
<code>name</code>	トリガ名	このトリガの名前。これは必須属性です。

表 1-26. 要素の階層

親要素	子要素
<code><events></code>	<ul style="list-style-type: none"> ▪ <code><description></code> ▪ <code><trigger-properties></code>

trigger-properties 要素

`<trigger-properties>` 要素は `<trigger-property>` 要素のコンテナです。

`<trigger-properties>` 要素はオプションです。`<trigger-properties>` 要素の属性はありません。

表 1-27. 要素の階層

親要素	子要素
<code><trigger></code>	<code><trigger-property></code>

trigger-property 要素

`<trigger-property>` 要素は、トリガ オブジェクトを特定するプロパティを定義します。

`<trigger-property>` 要素はオプションです。1 つのプラグインに対して設定できる `<trigger-property>` 要素の数に制限はありません。`<trigger-property>` 要素には、以下の属性が含まれています。

タイプ	値	説明
name	トリガ名	このトリガの名前。これはオプション属性です。
display-name	トリガ名	Orchestrator クライアントに表示される名前。これはオプション属性です。
type	トリガタイプ	このトリガを定義するオブジェクトタイプ。これは必須属性です。

表 1-28. 要素の階層

親要素	子要素
<trigger-properties>	なし

gauge 要素

<gauge> 要素はこのファインダで使用可能なゲージを定義します。ゲージを設定するには、**IPluginAdaptor** の **registerEventPublisher()** および **unregisterEventPublisher()** メソッドを実装する必要があります。

<gauge> 要素はオプションです。1 つのプラグインに対して設定できる <gauge> 要素の数に制限はありません。<gauge> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	ゲージ名	ゲージの名前。これは必須属性です。
min-value	数値	最小しきい値。これはオプション属性です。
max-value	数値	最大しきい値。これはオプション属性です。
unit	オブジェクトタイプ	ゲージを定義するオブジェクトタイプ。これは必須属性です。
format	文字列	監視される値の形式。これはオプション属性です。

表 1-29. 要素の階層

親要素	子要素
<events>	<description>

scripting-objects 要素

<scripting-objects> 要素は <object> 要素のコンテナです。

<scripting-objects> 要素はオプションです。<scripting-objects> 要素の属性はありません。

表 1-30. 要素の階層

親要素	子要素
<module>	<object>

object 要素

<object> 要素は、プラグイン テクノロジーのコンストラクタ、属性、およびメソッドを、Orchestrator のスクリプティング API が公開する JavaScript オブジェクトのタイプにマップします。

オブジェクトの命名規則については、「[プラグイン オブジェクトの命名](#)」を参照してください。

<object> 要素はオプションです。1 つのプラグインに対して設定できる **<object>** 要素の数に制限はありません。**<object>** 要素には、以下の属性が含まれています。

タイプ	値	説明
script-name	JavaScript 名	クラスのスクリプティング名。グローバルで一意である必要があります。これは必須属性です。
java-class	Java クラス	この JavaScript クラスにラップされた Java クラス。これは必須属性です。
create	true (デフォルト値) または false	true の場合、このクラスの新しいインスタンスを作成できます。これはオプション属性です。
strict	true または false (デフォルト)	true の場合、vso.xml ファイルで注釈を付けるか、宣言したメソッドのみを呼び出すことができます。これはオプション属性です。
is-deprecated	true または false (デフォルト)	true の場合、オブジェクトは非推奨 Java クラスをマップします。これはオプション属性です。
since-version	文字列	Java クラスが非推奨になってからのバージョン。これはオプション属性です。

表 1-31. 要素の階層

親要素	子要素
<scripting-objects>	<ul style="list-style-type: none"> ■ <description> ■ <deprecated> ■ <url> ■ <constructors> ■ <attributes> ■ <methods> ■ <singleton>

constructors 要素

<constructors> 要素は、**<object><constructor>** 要素のコンテナです。

<constructors> 要素はオプションです。**<constructors>** 要素の属性はありません。

表 1-32. 要素の階層

親要素	子要素
<object>	<constructor>

constructor 要素

<constructor> 要素は、コンストラクタのメソッドを定義します。**<constructor>** メソッドは、API Explorer でドキュメントを作成します。

<constructor> 要素はオプションです。1 つのプラグインに対して設定できる **<constructor>** 要素の数に制限はありません。**<constructor>** 要素の属性はありません。

表 1-33. 要素の階層

親要素	子要素
<constructors>	<ul style="list-style-type: none"> ▪ <description> ▪ <parameters>

コンストラクタの parameters 要素

<parameters> 要素は、**<constructor>****<parameter>** 要素のコンテナです。

<parameters> 要素はオプションです。**<parameters>** 要素の属性はありません。

表 1-34. 要素の階層

親要素	子要素
<constructor>	<parameter>

コンストラクタの parameter 要素

<parameter> 要素は、コンストラクタのパラメータを定義します。

<parameter> 要素はオプションです。1 つのプラグインに対して設定できる **<parameter>** 要素の数に制限はありません。**<parameter>** 要素には、以下の属性が含まれています。

タイプ	値	説明
name	文字列	API ドキュメントで使用するパラメータ名。これは必須属性です。
type	Orchestrator のパラメータ タイプ	API ドキュメントで使用するパラメータ タイプ。これは必須属性です。
is-optional	true または false	true の場合、値は NULL にすることができません。これはオプション属性です。
since-version	文字列	メソッドのバージョン。これはオプション属性です。

表 1-35. 要素の階層

親要素	子要素
<parameters>	なし

attributes 要素

`<attributes>` 要素は、`<object>``<attribute>` 要素のコンテナです。

`<attributes>` 要素はオプションです。`<attributes>` 要素の属性はありません。

表 1-36. 要素の階層

親要素	子要素
<code><object></code>	<code><attribute></code>

attribute 要素

`<attribute>` 要素は、プラグイン テクノロジーの Java クラスの属性を、Orchestrator の JavaScript エンジンが使用できる JavaScript 属性にマップします。

`<attribute>` 要素はオプションです。1 つのプラグインに対して設定できる `<attribute>` 要素の数に制限はありません。`<attribute>` 要素には、以下の属性が含まれています。

タイプ	値	説明
<code>java-name</code>	Java 属性	Java 属性の名前。これは必須属性です。
<code>script-name</code>	JavaScript オブジェクト	対応する JavaScript オブジェクトの名前。これは必須属性です。
<code>return-type</code>	文字列	この属性が返すオブジェクトのタイプ。API Explorer のドキュメントに記述されています。これはオプション属性です。 注意 JavaScript の戻り値のタイプが Properties の場合、 java.util.HashMap と java.util.Hashtable が基盤となる Java の実装としてサポートされます。
<code>read-only</code>	true または false	true の場合、この属性を変更することはできません。これはオプション属性です。
<code>is-optional</code>	true または false	true の場合、このフィールドは NULL にすることができます。これはオプション属性です。
<code>show-in-api</code>	true または false	false の場合、この属性は API ドキュメントに記述されていません。これはオプション属性です。
<code>is-deprecated</code>	true または false	true の場合、オブジェクトは非推奨属性をマップします。これはオプション属性です。
<code>since-version</code>	数値	この属性が非推奨となったバージョン。これはオプション属性です。

表 1-37. 要素の階層

親要素	子要素
<attributes>	なし

methods 要素

<methods> 要素は、<object><method> 要素のコンテナです。

<methods> 要素はオプションです。<methods> 要素の属性はありません。

表 1-38. 要素の階層

親要素	子要素
<object>	<method>

method 要素

<method> 要素は、プラグイン テクノロジーの Java メソッドを、Orchestrator の JavaScript エンジンが公開する JavaScript メソッドにマップします。

<method> 要素はオプションです。1 つのプラグインに対して設定できる <method> 要素の数に制限はありません。<method> 要素には、以下の属性が含まれています。

タイプ	値	説明
java-name	Java メソッド	引数のタイプを括弧内に示した Java メソッド署名の名前。例: <code>getVms(DataStore)</code> 。これは必須属性です。
script-name	JavaScript メソッド	対応する JavaScript メソッドの名前。これは必須属性です。
return-type	Java オブジェクトタイプ	このメソッドが取得するタイプ。これはオプション属性です。 注意 JavaScript の戻り値のタイプが Properties の場合、 <code>java.util.HashMap</code> と <code>java.util.Hashtable</code> が基盤となる Java の実装としてサポートされます。
static	true または false	true の場合、このメソッドは静的になります。これはオプション属性です。
show-in-api	true または false	false の場合、このメソッドは API ドキュメントに表示されません。これはオプション属性です。
is-deprecated	true または false	true の場合、オブジェクトは非推奨のメソッドをマップします。これはオプション属性です。
since-version	数値	このメソッドが非推奨となったバージョン。これはオプション属性です。

表 1-39. 要素の階層

親要素	子要素
<methods>	<ul style="list-style-type: none"> ■ <deprecated> ■ <description> ■ <example> ■ <parameters>

example 要素

<example> 要素を使用すると、API Explorer ドキュメントに表示される Javascript メソッドにサンプルコードを追加することができます。

<example> 要素はオプションです。<example> 要素の属性はありません。

表 1-40. 要素の階層

親要素	子要素
<method>	<ul style="list-style-type: none"> ■ <code> ■ <description>

code 要素

<code> 要素により、API Explorer ドキュメントに表示されるサンプルコードを指定することができます。

サンプルコードは、<code> タグと </code> タグの間に指定します。<code> 要素はオプションです。<code> 要素の属性はありません。

表 1-41. 要素の階層

親要素	子要素
<example>	なし

メソッドの parameters 要素

<parameters> 要素は、<method><parameter> 要素のコンテナです。

<parameters> 要素はオプションです。<parameters> 要素の属性はありません。

表 1-42.

親要素	子要素
<method>	<parameter>

メソッドの parameter 要素

<parameter> 要素はメソッドの入力パラメータを定義します。

<parameter> 要素はオプションです。1 つのプラグインに対して設定できる <parameter> 要素の数に制限はありません。<parameter> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	文字列	パラメータ名。これは必須属性です。
type	Orchestrator のパラメータ タイプ	パラメータ タイプ。これは必須属性です。
is-optional	true または false	true の場合、値は null にすることができます。これはオプション属性です。
since-version	文字列	メソッドのバージョン。これはオプション属性です。

表 1-43. 要素の階層

親要素	子要素
<parameters>	なし

singleton 要素

<singleton> 要素は JavaScript スクリプト オブジェクトを singleton インスタンスとして作成します。

singleton オブジェクトは静的 Java クラスと同様に動作します。singleton オブジェクトは、Orchestrator がプラグイン テクノロジーを使用してアクセスするオブジェクトの特定のインスタンスを定義するのではなく、使用するプラグインの汎用オブジェクトを定義します。たとえば、singleton オブジェクトを使用してプラグイン テクノロジーへの接続を確立できます。

<singleton> 要素はオプションです。<singleton> 要素には、以下の属性が含まれています。

タイプ	値	説明
script-name	JavaScript オブジェクト	対応する JavaScript オブジェクトの名前。これは必須属性です。
datasource	Java オブジェクト	この JavaScript オブジェクトの Java ソース オブジェクトです。これは必須属性です。

表 1-44. 要素の階層

親要素	子要素
<object>	なし

enumerations 要素

<enumerations> 要素は <enumeration> 要素のコンテナです。

<enumerations> 要素はオプションです。<enumerations> 要素の属性はありません。

表 1-45. 要素の階層

親要素	子要素
<module>	<enumeration>

enumeration 要素

<enumeration> 要素は特定のタイプのすべてのオブジェクトに適用される共通の値を定義します。

特定のタイプのすべてのオブジェクトで 1 つの特定の属性が必要で、その属性の値の範囲が限られている場合、異なる値を列挙エントリとして定義できます。たとえば、あるタイプのオブジェクトで **color** 属性が必要で、使用可能な色が、赤、青、および緑のみの場合、これらの 3 つの色の値を定義するための 3 つの列挙エントリを定義できます。エントリは enumeration 要素の子要素として定義します。

<enumeration> 要素はオプションです。1 つのプラグインに対して設定できる **<enumeration>** 要素の数に制限はありません。**<enumeration>** 要素には、以下の属性が含まれています。

タイプ	値	説明
type	Orchestrator オブジェクト タイプ	列挙タイプ。これは必須属性です。

表 1-46. 要素の階層

親要素	子要素
<enumerations>	<ul style="list-style-type: none"> ■ <url> ■ <description> ■ <entries>

entries 要素

<entries> 要素は **<enumeration>****<entry>** 要素のコンテナです。

<entries> 要素はオプションです。**<entries>** 要素の属性はありません。

表 1-47. 要素の階層

親要素	子要素
<enumeration>	<entry>

entry 要素

<entry> 要素は列挙属性の値を提供します。

<entry> 要素はオプションです。1 つのプラグインに対して設定できる **<entry>** 要素の数に制限はありません。

<entry> 要素には、以下の属性が含まれています。

タイプ	値	説明
id	テキスト	列挙エントリを属性として設定するためにオブジェクトが使用する ID。これは必須属性です。
name	テキスト	エントリ名。これは必須属性です。

表 1-48. 要素の階層

親要素	子要素
<entries>	なし

Orchestrator プラグインを開発する場合のベスト プラクティス

Orchestrator プラグインの構造と内容について理解し、特定の問題を回避する方法を知ることにより、プラグインを効率的に開発することができます。

■ Orchestrator プラグインのビルド方法

Orchestrator プラグインはいくつかの方法でビルドできます。プラグインをレイヤ単位でビルドすることも、プラグインのすべてのレイヤを同時にビルドすることもできます。

■ Orchestrator プラグインのタイプ

プラグインを使用すると、汎用ライブラリ、XML や SSH のようなユーティリティ、および vCloud Director などのシステム全体を Orchestrator と統合できます。Orchestrator と統合するテクノロジーに応じて、プラグインはサービス用のプラグイン、汎用プラグイン、およびシステム用のプラグインに分類できます。

■ プラグインの実装

ワークフローのプレゼンテーションの提供と同様に、プラグインの構成、必要な Java クラスおよび JavaScript オブジェクトの実装、プラグイン ワークフローの開発において、便利なプラクティスやテクニックを使用することができます。

■ Orchestrator プラグイン開発の推奨事項

複数の Orchestrator プラグインのコンポーネントを開発する際に、特定のプラクティスを順守することでプラグインの品質を向上させることができます。

■ プラグインと API ドキュメントのユーザー インターフェイス文字列

Orchestrator のプラグインとそれに関連する API ドキュメントでユーザー インターフェイス (UI) の文字列を指定する場合は、スタイルと形式に関するルールに従う必要があります。

Orchestrator プラグインのビルド方法

Orchestrator プラグインはいくつかの方法でビルドできます。プラグインをレイヤ単位でビルドすることも、プラグインのすべてのレイヤを同時にビルドすることもできます。

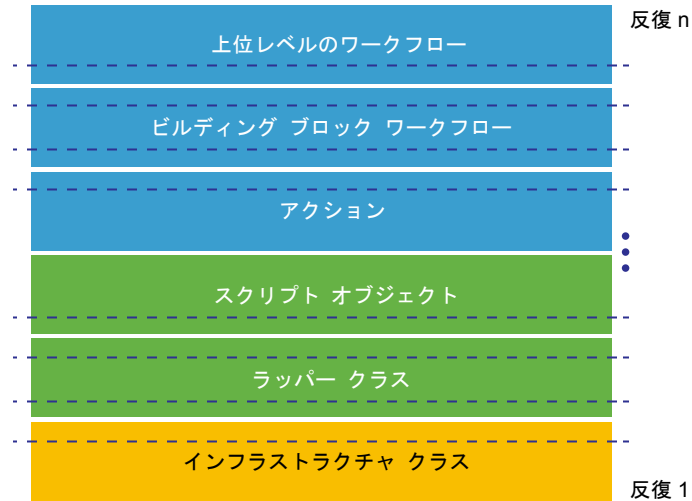
プラグインのレイヤについては、「[Orchestrator プラグインの構造](#)」を参照してください。

ボトムアップ方式によるプラグイン開発

ボトムアップの開発アプローチを使用すると、レイヤ単位でプラグインを開発することができます。

ボトムアップの開発アプローチでは、下位レベルのレイヤから開発を開始し、上位レベルのレイヤに向かって開発を進めていくことにより、レイヤ単位でプラグインを開発することができます。このアプローチを、対話式的開発アプローチと反復開発アプローチと組み合わせて使用すると、それぞれの反復でレイヤの一部だけを作成することも、レイヤ全体を作成することもできます。その場合、N 回目の反復が完了した時点で、プラグインの開発も完了することになります。

図 1-5. ボトムアップ方式によるプラグイン開発



ボトムアップによるプラグイン開発アプローチの利点は、一度に 1 つのレイヤだけに焦点を当てて開発を行うことができるということです。

ただし、ボトムアップによるプラグイン開発アプローチには、以下の欠点もあることに注意する必要があります。

- ある程度まで作業が進まないと、プラグイン開発の進捗状況を把握できない。
- アジャイル開発手法にはあまり適していない。

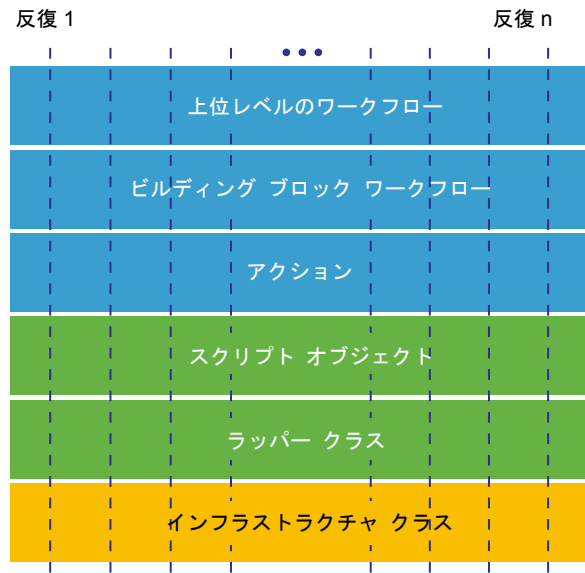
ラッパー クラス、スクリプト オブジェクト、アクション、ワークフローをあまり使用しない（あるいはまったく使用しない）小規模なプラグインの場合は、ボトムアップ方式の開発プロセスでも問題はありません。

トップダウン方式によるプラグイン開発

トップダウン方式の開発アプローチを使用すると、プラグインをトップダウン機能にスライスしてビルドできます。

トップダウン方式のアプローチをアジャイル開発プロセスと組み合わせて使用すると、それぞれの反復で新しい機能を作成できます。その場合、N 回目の反復が完了した時点で、プラグインも実装されることになります。

図 1-6. トップダウン方式によるプラグイン開発



トップダウン方式によるプラグイン開発には、以下のような利点があります。

- プラグイン開発の進行状況が初回の反復からわかりやすい。これは新しいプロセスが反復ごとに完了するため、プラグインが解放されて次の反復に使用できることによります。
- 縦にスライスされたプロセスを完了させることにより、成功基準と完了済みプロセスの定義を明確にできるほか、開発者、製品管理担当者、品質保証 (QA) 技術者間のコミュニケーションが向上する。
- QA 技術者が開発プロセスの初期段階からテストと自動化を開始できる。このようなアプローチでは有益なフィードバックを入手して、プロジェクトの納入期間全体を短縮できます。

トップダウン方式によるプラグイン開発アプローチの欠点は、さまざまなレイヤで開発作業が同時に進められる点です。

トップダウン方式によるプラグイン開発プロセスはほとんどのプラグインに適していますが、特に動的な要件を持つプラグインに適しています。

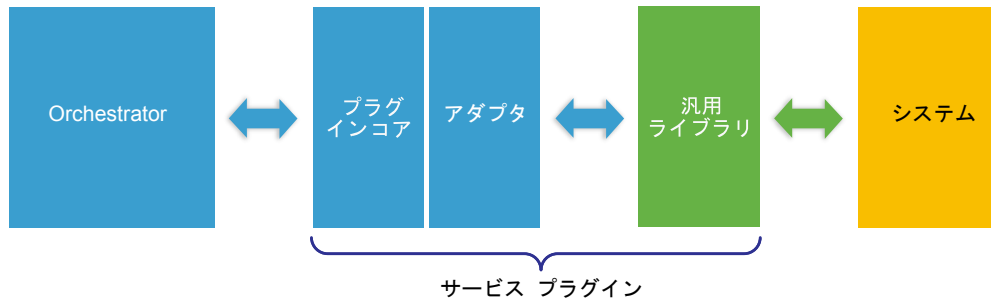
Orchestrator プラグインのタイプ

プラグインを使用すると、汎用ライブラリ、XML や SSH のようなユーティリティ、および vCloud Director などのシステム全体を Orchestrator と統合できます。Orchestrator と統合するテクノロジーに応じて、プラグインはサービス用のプラグイン、汎用プラグイン、およびシステム用のプラグインに分類できます。

サービス用プラグイン

サービス用プラグインまたは汎用のプラグインは、Orchestrator 内のサービスとみなされる機能を提供します。

図 1-7. サービス用プラグインのアーキテクチャ



サービス用プラグインは、一般ライブラリやユーティリティを XML、SSH、SOAP といった Orchestrator に公開します。たとえば、Orchestrator で使用可能な以下のプラグインは、サービス用プラグインです。

JDBC プラグイン	ワークフロー内でデータベースの使用を可能にします。
メール プラグイン	ワークフロー内で電子メールの使用を可能にします。
SSH プラグイン	SSH 接続を開き、ワークフロー内でコマンドを実行します。
XML プラグイン	ワークフロー内の XML ドキュメントを管理します。

サービス用プラグインには、以下のような特徴があります。

複雑性	サービス用プラグインの複雑性は、低から中レベルです。サービス用プラグインは、具体的な機能が分かるように、Orchestrator 内でライブラリの詳細またはライブラリの一部を公開します。たとえば、XML プラグインは、Orchestrator JavaScript API にドキュメントオブジェクトモデル (DOM) XML パーサーの実装を追加します。
サイズ	サービス用プラグインは比較的小さいサイズです。すべてのプラグインに同じ基本的な一連のクラスが必要であり、また新機能追加のための新しいスクリプトオブジェクトを提供するその他のクラスが必要です。
インベントリ	サービス用プラグインでは作業オブジェクトについて、小規模なインベントリを必要とするか、もしくはインベントリを全く必要としません。サービス用プラグインのオブジェクトモデルは一般的で小規模なため、Orchestrator インベントリ内でこのモデルを表示する必要はありません。

システム用プラグイン

システム用プラグインは、Orchestrator ワークフロー エンジンと外部システムに接続することで、外部のシステムを統合することができます。

システム用プラグインの例を以下に示します。

vCenter Server プラグイン	ワークフローを使用して vCenter Server インスタンスの管理を行います。
vCloud Director プラグイン	ワークフロー内で vCloud Director のインストールの操作を行います。
Cisco UCSM プラグイン	ワークフロー内で Cisco エンティティの操作を行います。

システム用プラグインの主な特徴は、以下のとおりです。

複雑性

システム用プラグインは、公開するテクノロジーが比較的複雑であるため、汎用のプラグインより複雑性が高レベルになっています。システム用プラグインは、Orchestrator で外部システムの操作を行いその機能を使用するための、Orchestrator 内にあるすべての外部システムの要素を表します。外部システムが統合メカニズムを持っている場合、それを使用してより簡単に Orchestrator 内でシステムの機能を公開することができます。ただし、外部システムの要素を表す以外に、システム用プラグインには高いスケーラビリティとキャッシュ メカニズムの提供、イベントや通知の処理なども求められる可能性があります。

サイズ

システムのプラグインのサイズは、中規模から大規模です。システム用プラグインは、通常スクリプト オブジェクトを多数持つため、基本的な一連のクラスとは別に多くのクラスを必要とします。また、それらと相互に作用するその他のヘルパーや補助クラスを必要とする場合があります。

インベントリ

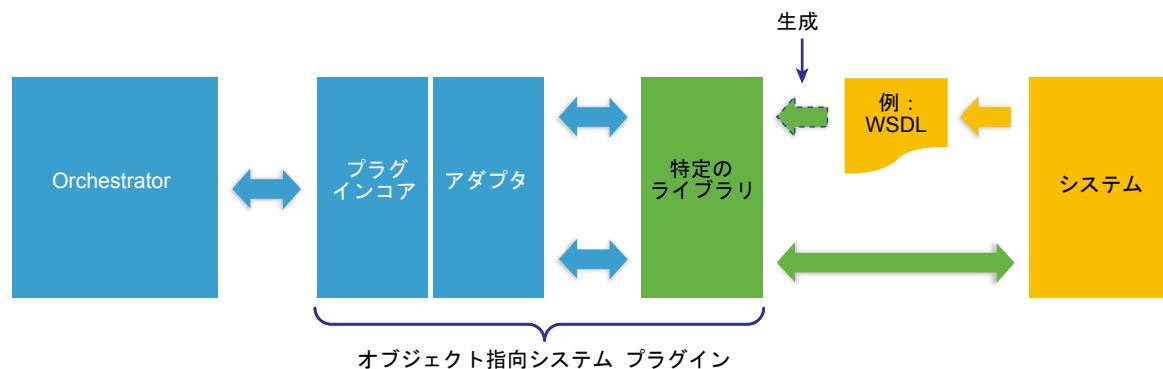
通常、システム用プラグインには多数のオブジェクトがあるため、これらのオブジェクトをインベントリに適切に公開して、Orchestrator 内で容易に場所が分り使用できるようにする必要があります。システム用プラグインが公開する必要のあるオブジェクトの数が多いため、プラグイン用にできるだけ多くのコードを自動生成する補助ツールまたはプロセスを構築することが推奨されます。たとえば、vCenter Server プラグインは、そのようなツールを提供します。

オブジェクト指向システム用プラグイン

オブジェクト指向のシステムでは、オブジェクトおよび RPC に基づいた操作メカニズムを提供します。

最も一般的に使用されているオブジェクト指向システムのモデルは、SOAP を使用する Web サービスモデルです。このモデル内のオブジェクトは、オブジェクトの状態に関する属性のセットを持ち、ターゲットシステム側で起動されるリモート メソッドのセットを提供します。

図 1-8. オブジェクト指向システム用プラグイン



オブジェクト指向システム用のプラグインの実行時には、以下を考慮します。

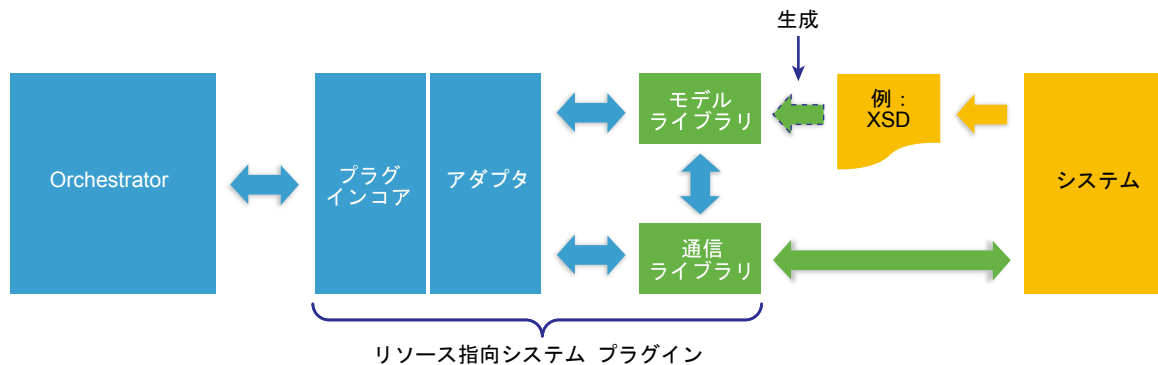
- SOAP を使用する場合は、WSDL ファイルを使用して、オブジェクトモデルと通信メカニズムを組み合わせるクラスのセットを生成することができます。
- このオブジェクトモデルは、Orchestrator 内で公開するほぼすべてのものになります。

リソース指向システム用プラグイン

リソース指向システムでは、リソースに基づいた操作メカニズムおよび HTTP 手法を用いた簡単な操作を提供しています。

リソース指向システム モデルの最も代表的なものは、XML などと組み合わせた REST モデルです。モデル内のオブジェクトは、状態に関連した属性を持ちます。ターゲットシステム（通信メカニズム）でメソッドを開始するには、GET、POST、PUT といった標準的な HTTP 手法を使用し、いくつかの規則に従う必要があります。

図 1-9. リソース指向システム用プラグイン



リソース指向システム用のプラグインの実行時には、以下を考慮します。

- REST を使用する、または XML で HTTP のみを使用する場合、メッセージの読み書き可能な XML スキーマ ファイルを 1 つ以上取得します。このスキーマから、オブジェクト モデルを定義する一連のクラスを生成することができます。この一連のクラスでは、たとえば vCloud Director プラグインの定義のように HTTP 手法で暗示的に操作を定義したり、また Cisco UCSM プラグインのように、XML メッセージで明示的に定義したりします。
- 別のクラスでは、通信メカニズムを実装する必要があります。この一連のクラスでは、元のオブジェクト モデルを操作する新しいオブジェクトモデルを定義します。通信メカニズム用のオブジェクト モデルは、オブジェクトとメソッドのみで構成されます。
- Orchestrator 内では、元のオブジェクト モデルと通信メカニズム用のオブジェクト モデルの両方を公開することができます。両方のオブジェクトモデルの公開方法や、関連オブジェクトを両側から統合するか（オブジェクト指向システムのシミュレーションのため）または別々にするかどうかにより、複雑性が増す可能性があります。

プラグインの実装

ワークフローのプレゼンテーションの提供と同様に、プラグインの構成、必要な Java クラスおよび JavaScript オブジェクトの実装、プラグイン ワークフローの開発において、便利なプラクティスやテクニックを使用することができます。

プロジェクト構造

Orchestrator プラグインのプロジェクトに対して、標準の構造を適用することができます。

内部プロジェクト

プラグインの実装時に、オブジェクトのキャッシュや、オブジェクトのバックグラウンドへの移動、オブジェクトのクローンの作成など、特定のアプローチを適用することができます。それらの手法に従うことで、プラグインのパフォーマンスを改善、並行処理の問題を回避、および Orchestrator クライアントの応答性を改善することができます。

■ 内部ワークフロー

ワークフローを実装することで、Orchestrator プラグインが実行する長時間の処理を監視することができます。

■ ワークフローとアクション

ワークフロー開発と使用を容易にするために、特定のプラクティスを使用することができます。

■ ワークフロー プレゼンテーション

ワークフローのプレゼンテーションを作成するには、特定の構造と規則を適用する必要があります。

プロジェクト構造

Orchestrator プラグインのプロジェクトに対して、標準の構造を適用することができます。

プラグイン プロジェクト用のモジュールを持つ標準の Maven 構造を使用して、各種の機能が存在する場所を明確に確認することができます。

表 1-49. プラグイン プロジェクトの構造

モジュール	説明
/myAwesomePlugin-plugin	プラグイン プロジェクトの root モジュール。
/o11nplugin-myAwesomePlugin	最終プラグインの DAR ファイルを構成するモジュール。
/o11nplugin-myAwesomePlugin-config	プラグイン構成 Web アプリケーションを格納するモジュール。このモジュールにより、標準の WAR ファイルが生成されます。
/o11nplugin-myAwesomePlugin-core	Orchestrator 標準プラグインの任意のインターフェイスを実装するすべてのクラスと、そのクラスで使用される他の補助クラスを格納するモジュール。このモジュールにより、標準の JAR ファイルが生成されます。
/o11nplugin-myAwesomePlugin-model	プラグイン経由でサードパーティ テクノロジーを Orchestrator に統合するためのすべてのクラスを格納するモジュール。これらのクラスに、API の標準 Orchestrator プラグインに対する直接参照を含めないでください。
/o11nplugin-myAwesomePlugin-package	アクションとワークフローを使用して Orchestrator の外部パッケージ ファイルをインポートし、最終プラグインの DAR ファイル内のそのファイルを含めるモジュール。このモジュールはオプションです。

内部プロジェクト

プラグインの実装時に、オブジェクトのキャッシュや、オブジェクトのバックグラウンドへの移動、オブジェクトのクローンの作成など、特定のアプローチを適用することができます。それらの手法に従うことで、プラグインのパフォーマンスを改善、並行処理の問題を回避、および Orchestrator クライアントの応答性を改善することができます。

オブジェクトのキャッシュ

プラグインはリモート サービスと通信する場合があります、この通信はサービス側のリモート オブジェクトを代表するローカル オブジェクトによって行われます。ローカル オブジェクトをリモートサービスから毎回取得するのではなくキャッシュに格納することで、プラグインのパフォーマンスを Orchestrator のユーザー インターフェイスの応答性と同程度に向上させることができます。キャッシュの範囲は、たとえばすべてのプラグイン クライアントにつき 1 キャッシュ、プラグインのユーザーごとに 1 キャッシュ、および第三者サービスのユーザーごとに 1 キャッシュなどにすることができます。実装時には、キャッシュ メカニズムは、オブジェクトの検索および無効化のプラグイン インターフェイスと統合されます。

オブジェクトのバックグラウンドへの移動

プラグイン インベントリのオブジェクトのリストが大きく、オブジェクトをすばやく検索する方法がない場合は、オブジェクトをバックグラウンドに移動させることができます。オブジェクトをバックグラウンドへ移動させるには、たとえばオブジェクトを **fake** と **loaded** の 2 つの状態にします。**fake** オブジェクトの作成が非常に容易で、インベントリに表示される情報が名前や ID など最小限であると仮定します。その場合、常に **fake** オブジェクトを返すことが可能であり、またすべての情報（実際のオブジェクト）が必要になった場合には、使用しているエンティティまたはプラグインによって、自動的に **load** メソッドが開始され、実際のオブジェクトが取得されます。エンティティを使用するアクションを予測して、**fake** オブジェクトが返された後にオブジェクトの読み込みが自動的に開始するように構成することもできます。

並行処理の問題回避のためクローンを作成する

プラグインにキャッシュを使用する場合、オブジェクトのクローンを作成する必要があります。要求したすべてのエンティティに対して常に同じオブジェクトのインスタンスを返すキャッシュを使用すると、望ましくない影響が生じる可能性があります。たとえば、エンティティ A はオブジェクト O を要求し、エンティティの表示では、インベントリ内のオブジェクトのすべての属性が表示されるとします。同時に、エンティティ B もオブジェクト O を要求し、エンティティ A はオブジェクト O の属性を変更するワークフローを実行します。実行の最後に、ワークフローはオブジェクトの **update** メソッドを起動して、サーバ側のオブジェクトの更新を行います。エンティティ A とエンティティ B がオブジェクト O と同じインスタンスを取得する場合、エンティティ A は、変更がサーバ側でコミットされる前でも、インベントリ内のエンティティ B が実行したすべての変更を表示します。実行が成功した場合は問題ありませんが、実行が失敗した場合は、エンティティ A のオブジェクト O の属性は戻らなくなります。このような場合は、キャッシュ（プラグインの **find** 操作）が常に同一のインスタンスの代わりにオブジェクトのクローンを返すことで、それぞれがエンティティの表示を使用してコピーを変更し、少なくとも Orchestrator 内では並行処理の問題を回避することができます。

変更を他の人に通知する

キャッシュやクローン オブジェクトを同時に使用する場合、問題が発生する可能性があります。最大の問題は、エンティティのビューを使用しているオブジェクトが、そのオブジェクトで使用可能な最新バージョンではなくなることです。たとえば、エンティティがインベントリを表示する場合、オブジェクトは一度読み込まれますが、同時に他のエンティティがオブジェクトを変更している場合は、最初のエンティティにはその変更が表示されません。この問題を回避するには、Orchestrator のプラグイン API から **PluginWatcher** メソッドおよび **IPluginPublisher** メソッドを使用して、変更について通知し、Orchestrator クライアントの他のインスタンスに変更の確認を許可します。また、これは Orchestrator クライアント独自のインスタンスでも、インベントリのオブジェクト 1 つの変更がその他のインベントリに影響するため通知が必要な場合に適用されます。通知が使用されることの多い操作は、オブジェクトそのものまたはプロパティがインベントリに表示されているオブジェクトの追加、更新、削除です。

オブジェクトをいつでも検索可能にする

オブジェクトをタイプと ID のみで検索可能にするには、**IPluginFactory** インターフェイスの **find** メソッドを実装します。**find** メソッドは、Orchestrator を再起動し任意のワークフローを再開した後に直接開始することができます。

保有していないクエリ サービスをシミュレーションする

Orchestrator クライアントは、特定のケースのオブジェクトについてクエリを作成したり、クエリをツリーではなくリストまたは表などとして表示する必要がある場合があります。これは、プラグインが一連のオブジェクトについていつでもクエリを作成できる必要があることを意味します。サードパーティのテクノロジーによりクエリ サービスが提供されている場合は、このサービスを適用して使用する必要があります。クエリ サービスが提供されていない場合は、非常に複雑であったりパフォーマンスが低い場合であっても、クエリ サービスのシミュレーションをする必要があります。

検索メソッドがランタイム例外を返さないようにする

プラグイン内の検索を実装するための **IPluginFactory** インターフェイスは、コントロールまたは非コントロールされたランタイム例外をスローしません。これは、ワークフロー実行中の **validation error** 不具合の原因となる場合があります。たとえば、ワークフローの 2 つのノード間で、最初のノードの出力が 2 番目のノードの入力である場合に **find** メソッドが開始されます。その時オブジェクトがランタイム例外のため見つからない場合は、Orchestrator クライアント内では **validation error** 以上の情報を得ることはできません。その後、プラグインが例外をどのようにログに記録するかによって、ログ ファイル内の情報の量が変わります。

内部ワークフロー

ワークフローを実装することで、Orchestrator プラグインが実行する長時間の処理を監視することができます。

タスクの監視など、長時間実行される処理の監視のためにワークフローを実装することができます。このワークフローは、Orchestrator トリガおよび待機中のイベントに基づいて作成できます。ただし、タスク待機中でブロックされているワークフローは、Orchestrator サーバの起動後すぐに再開されることを考慮しておく必要があります。プラグインは、監視プロセスを正しく再開するために必要なすべての情報を取得できなければなりません。

内部的に使用される監視ワークフローまたはタスクは、ポーリング レートおよび有効なタイムアウト期間を指定するメカニズムを持っている必要があります。

特にコードが Java コードを開始するものではない場合、ワークフロー内のスクリプティング コードの一部をデバッグするプロセスは簡単ではありません。このため、デフォルトの Orchestrator スクリプト オブジェクトで提供されているログの記録方法を使用することが唯一の選択肢となる場合があります。

ワークフローとアクション

ワークフロー開発と使用を容易にするために、特定のプラクティスを使用することができます。

開発ワークフローをビルディング ブロックとして開始する

ビルディング ブロックは数個のパラメータ入力で単純な出力を返す、簡易的なワークフローです。十分なビルディング ブロックがある場合は、より高レベルなワークフローを簡単に作成でき、よりよいツールのセットを提供することで複雑なワークフローを構成することができます。

小規模なコンポーネントに基づいて高レベルのワークフローを作成する

複雑なワークフローをいくつかの入力と内部手順で開発する場合、そのワークフローをより小さく単純なビルディング ブロックとアクションに分割することができます。

アクションの作成がいつでも可能

ワークフローの開発時、柔軟にアクションを作成できます。

- スクリプティング方法に合わせて複雑なオブジェクトやパラメータを簡単に作成
- 共通のコードの繰り返しを常に回避
- ユーザー インターフェイスの検証を実施

ワークフローでアクションの呼び出しがいつでも可能

ワークフローのスキーマ内のノードとして、アクションを直接呼び出すことができます。これにより、1 つのアクションを呼び出すためにスクリプティング コードを追加する必要がなくなり、ワークフローのスキーマがより単純になります。

予測情報の入力

ワークフローまたはアクションの要素ごとに情報を提供します。

- ワークフローまたはアクションの説明を提供
- 入力パラメータの説明を提供
- 出力の説明を提供
- ワークフローの属性の説明を提供

バージョン情報を最新に維持

プラグインのバージョン更新時、プラグインのメジャー アップデート、実装の重要な詳細などの情報とともに、重要なコメントを追加します。

ワークフロー プレゼンテーション

ワークフローのプレゼンテーションを作成するには、特定の構造と規則を適用する必要があります。

ワークフロー プレゼンテーションのワークフローの入力には、次のプロパティを使用します。

表 1-50. ワークフロー入力のプロパティ

プロパティ	用途
Show in Inventory	このプロパティにより、ユーザーはワークフローをインベントリ ビューから実行できます。
Specify a root object to be shown in the chooser	このプロパティは、ユーザーが入力を選択できるようにします。root オブジェクトが、プレゼンテーション内で更新可能な場合、属性の場合、またはオブジェクト メソッドで取得された場合、プレゼンテーション内でオブジェクトを更新する適切なアクションを作成または設定する必要があります。
Maximum string length	このプロパティは、名前、説明、ファイル パスなど、長い文字列に使用します。

表 1-50. ワークフロー入力のプロパティ (続き)

プロパティ	用途
Minimum string length	このプロパティは、テスト ツールの空の文字列を防止するために使用します。
Custom validation	アクションを伴う詳細な検証を実装します。

入力をステップと表示グループで整理します。このように整理することで、ユーザーはワークフローのすべての入力パラメータを認識および識別することができます。

Orchestrator プラグイン開発の推奨事項

複数の Orchestrator プラグインのコンポーネントを開発する際に、特定のプラクティスを順守することでプラグインの品質を向上させることができます。

表 1-51. プラグインの実装における有益なプラクティス

コンポーネント	アイテム	説明
一般	サードパーティ API へのアクセス	プラグインでは、可能な限り簡素化したサードパーティ API へのアクセスメソッドを使用します。
	インターフェイス	プラグインでは、API が複雑であっても、ユーザーに分かりやすく標準的なインターフェイスを使用します。
アクション	スクリプト オブジェクト	オブジェクトの作成、変更、削除、ならびにオブジェクトのスクリプティングに使用するその他すべてのメソッドにアクションを作成します。
	説明	アクションの説明には、そのアクションがどのように機能するかではなく、そのアクションが何をするかを記述します。
	スクリプティング	オブジェクトのプロパティまたはメソッドの取得にスクリプティングを使用する場合、オブジェクトの値が null または undefined と異なることを確認してください。
	廃止	あるアクションが廃止された場合、 comment または throw ステートメントで代替のアクションを示すか、そのアクションが代替のアクションを呼び出して、廃止されたバージョンのアクションでビルドされたソリューションが失敗しないようにします。
ワークフロー	統合されたテクノロジーのユーザー インターフェイス操作	統合されたテクノロジーのユーザー インターフェイスで利用できる各操作に対して、ワークフローを作成します。
	説明	ワークフローの説明には、そのワークフローがどのように機能するかではなく、そのワークフローが何をするかを記述します。
	プレゼンテーション プロパティ mandatory input	必須のワークフロー入力にはすべて、 mandatory input プロパティを設定する必要があります。
	プレゼンテーション プロパティ default value	任意のエンティティを構成するワークフローを開発する場合、ワークフローのプレゼンテーションは、そのエンティティ用にデフォルトの設定値を読み込む必要があります。たとえば、「ホスト構成」という名前のワークフローを開発する場合、ワークフローのプレゼンテーションはホスト構成のデフォルト値を読み込む必要があります。
	プレゼンテーション プロパティ Show in inventory	インベントリのオブジェクトにコンテキスト ワークフローを作成するため、 Show in inventory プロパティを設定する必要があります。

表 1-51. プラグインの実装における有益なプラクティス (続き)

コンポーネント	アイテム	説明
	プレゼンテーション プロパティ specify a root parameter	このプロパティは、ツリー ルートからインベントリを閲覧する必要がない時に、ワークフローで使します。
	ワークフローの検証	ワークフローの検証を行いすべてのエラーを修正する必要があります。
	オブジェクトの作成	新規オブジェクトを作成するすべてのワークフローは、出力パラメータとして新規プロジェクトを返します。
	廃止	ワークフローが廃止された場合、 comment または throw ステートメントで代替のワークフローを示すか、廃止されたワークフローが代替のワークフローを呼び出して、以前のバージョンのワークフローでビルドされたソリューションが失敗しないようにします。
インベントリ	ホストの切断	インベントリにホストへの接続が含まれていて、そのホストが使用できなくなった場合は、ホストが切断されたことを示す必要があります。 root オブジェクトに disconnected を付けて名前を変更する、または vCloud Director プラグインと同様にそのオブジェクトの下にあるツリー オブジェクトを削除することで、ホストが切断されたことを示すことができます。
	Select value as list プロパティ	インベントリ オブジェクトは treeview または list 形式で選択可能である必要があります。
	ホスト マネージャ	プラグインが host オブジェクトをターゲットシステムに実装する場合、親である hostmanager root オブジェクトには、追加、削除、またはホスト プロパティの編集のプロパティが存在する必要があります。
	オブジェクトの取得または更新	クエリ サービスが統合されたテクノロジー上で実行されている場合、複数のオブジェクトの取得にクエリ サービスを使用します。
	子の検出	子オブジェクトを個別に取得する必要がある場合、取得プロセスにはマルチスレッド機能を用いて、1 つのエラーでブロックされないようにします。
	Orchestrator オブジェクトの変更	インベントリの要素の状態を変更する可能性のあるすべてのワークフローでは、インベントリの更新を行い、オブジェクトが非同期状態にならないようにします。
	外部オブジェクトの変更	通知メカニズムを使用して、Orchestrator の外部で行われた操作の結果生じた統合テクノロジーの変更を通知することができます。そのような操作により統合テクノロジーからオブジェクトが削除される場合、インベントリを更新してデータのエラーや消失を防ぐ必要があります。たとえば、仮想マシンが vCenter Server から削除された場合、vCenter Server プラグインはインベントリを更新して削除された仮想マシンのオブジェクトを削除します。
	ファインダ オブジェクト	ファインダ オブジェクトには、オブジェクトの識別ができるプロパティを設定します。これらは通常、ユーザー インターフェイスに表示されるプロパティです。
オブジェクトのスク립ティング	実装	equals メソッドを実装して、オブジェクトに 2 つのインスタンスが含まれている場合と同じオブジェクトに == の操作が行われるようにします。
	プラグイン オブジェクトのプロパティ	親オブジェクトを持つオブジェクトには、 parent プロパティを実装します。
	プラグイン オブジェクトのプロパティ	子オブジェクトを持つオブジェクトには、子オブジェクトの配列を返す GET メソッドを実装します。
	インベントリ オブジェクト	インベントリ オブジェクトは Server.find で検索できるようにします。

表 1-51. プラグインの実装における有益なプラクティス (続き)

コンポーネント	アイテム	説明
		ワークフロー内で入力属性または出力属性として使用できるように、すべてのインベントリ オブジェクトを直列化します。
	コンストラクタとメソッド	多くの場合、スクリプト可能なオブジェクトは、コンストラクタを持っている、もしくは他のオブジェクト属性またはメソッドにより返されたものである必要があります。
	オブジェクト ID	外部システムから発行された ID を持つオブジェクトには、1 つ以上のサーバを統合した場合に ID の重複が発生しないように、内部 ID を使用します。
	オブジェクトの検索	search メソッドまたは find メソッドでは、すべてのオブジェクトではなく指定した名前または ID が検索されるように、フィルタを実装します。たとえば、Orchestrator サーバに、プラグイン オブジェクトを ID で検索できる Server.FindForId メソッドがあるとします。実行するには、このメソッドをプラグイン内の検索可能な各オブジェクトに実装する必要があります。
	トリガ	可能な場合は、Orchestrator での様々なイベントにおけるポリシーのトリガを可能にするため、オブジェクトが変更されてもトリガを使用できるようにします。たとえば、Orchestrator は新しい仮想マシンの追加、電源オン、電源オフなどがいつ行われたかなどの確認のため、 Datacenter オブジェクトの vCenter プラグインでトリガまたはイベントを監視することができます。
	オブジェクトのプロパティ	他のプラグインにあるオブジェクトには、あるプラグインから別のプラグインへ容易に変換できるプロパティを設定します。たとえば、仮想マシンのオブジェクトは、 moref (managed object reference ID) プロパティを持つなどです。
	セッション マネージャ	異なるセッションを持つ可能性のあるリモート サーバに接続する場合、共有セッションおよびユーザーごとのセッションをプラグインに実装します。
トリガ	トリガ	すべての長時間の処理およびブロック メソッドは、返されたタスクを非同期で開始でき、完了時にはトリガ イベントを生成できることが推奨されます。
列挙	列挙	特定のタイプの列挙は、列挙内の異なる値から選択できるインベントリ オブジェクトを持つことが推奨されます。
ログ	ログ	メソッドは様々なログ レベルを実装する必要があります。
バージョンニング	プラグインのバージョン	プラグインのバージョンは、基準に従い、プラグインの更新に合わせて更新されるようにします。
API ドキュメント	メソッド	API ドキュメントに記述されているメソッドは、オブジェクトに no xyz method / property 例外をスローしないようにします。代わりにメソッドは、使用できるプロパティがない場合は null を返し、プロパティが使用できない場合は詳細についてドキュメントを作成します。
	vso.xml	すべてのオブジェクト、メソッド、ならびにプロパティは vso.xml 内に記録される必要があります。

プラグインと API ドキュメントの ユーザー インターフェイス文字列

Orchestrator のプラグインとそれに関連する API ドキュメントでユーザー インターフェイス (UI) の文字列を指定する場合は、スタイルと形式に関するルールに従う必要があります。

一般的な推奨事項

- プラグインで使用される VMware 製品については、正式な名称を使用してください。たとえば、以下の製品と VMware 用語については、正式な名称を使用してください。

正しい名称	間違った名称
vCenter Server	VC、vCenter
vCloud Director	vCloud

- ワークフローの説明の最後には、必ずピリオド (.) を指定してください。たとえば、「**Creates a new Organization.**」のように指定してください。
- スペル チェック機能付きのテキスト エディタを使用して説明を入力してから、その説明をプラグインに移動してください。
- プラグインの名前は、そのプラグインが関連付けられている承認済みサードパーティ製品の名前と正確に一致している必要があります。

ワークフローとアクション

- わかりやすい説明を入力してください。多くのアクションとワークフローの場合、説明としては 1 文または 2 文で十分です。
- 上位レベルのワークフローの場合、より詳しい説明やコメントが必要になることがあります。
- 説明は、「**Creates...**」などの動詞で始めてください。「**This workflow creates**」などのような文章を入力しないでください。
- 説明文の最後には、必ずピリオド (.) を指定してください。
- ワークフローやアクションがどのように実行されるかということではなく、ワークフローやアクションによって何が実行されるのかということを入力してください。
- 通常、ワークフローとアクションは、フォルダやパッケージ内に保存されます。そのため、これらのフォルダとパッケージについても、簡単な説明を入力してください。たとえば、ワークフロー フォルダの場合は、「**Set of workflows related to vApp Template management**」などのような説明を入力します。

ワークフローとアクションのパラメータ

- ワークフローとアクションのパラメータの説明は、「**Name of**」などの名詞で始めてください。「**It's the name of**」などのような文章を入力しないでください。
- パラメータとアクションの説明の最後にピリオド (.) を指定しないでください。この場合の説明は、完全な文章ではないためです。
- ワークフローの入力パラメータの場合、適切な名前のラベルをプレゼンテーション ビューで指定する必要があります。多くの場合、関連する入力情報を 1 つの表示グループとして組み合わせることができます。たとえば、「組織名」と「組織の正式名称」というラベルを持つ 2 つの入力情報を個別に使用する代わりに、「組織」というラベルを持つ表示グループを 1 つだけ作成し、「組織名」と「組織の正式名称」というラベルを持つ 2 つの入力情報を「組織」という表示グループに含めることができます。
- ステップと表示グループについては、ワークフローのプレゼンテーションに表示される説明やコメントも追加してください。

プラグイン API

- API のドキュメントは、**vso.xml** ファイルと Java ソース ファイル内のすべてのドキュメントを参照します。
- **vso.xml** ファイルについては、ワークフローとアクションの説明を入力する場合と同じルールで、ファインダオブジェクトとスクリプト オブジェクトの説明を入力してください。オブジェクトの属性とメソッドのパラメータの説明については、ワークフローとアクションのパラメータを入力する場合と同じルールで説明を入力してください。
- **vso.xml** ファイル内で特殊文字を使用しないでください。また、`<![CDATA[insert your description here!]]>` タグ内に説明を入力してください。
- Java ソース ファイルの場合は、標準の Javadoc スタイルを使用してください。

ワークフロー開始時のユーザーからの入力パラメータの取得

ワークフローで入力パラメータが必要な場合、ダイアログ ボックスが開かれ、ユーザーはここに実行の際に必要な入力パラメータ値を入力します。このダイアログ ボックスのコンテンツおよびレイアウト、またはプレゼンテーションは、ワークフロー エディタの [プレゼンテーション] タブで調整できます。

[プレゼンテーション] タブでパラメータを調整した方法は、ワークフローが実行するときに入力パラメータ ダイアログ ボックスに変換されます。

[プレゼンテーション] タブでは、ユーザーが入力パラメータを指定するときに役立つ入力パラメータの説明を追加することもできます。また、[プレゼンテーション] タブでは、パラメータのプロパティおよび制約を設定して、ユーザーが指定するパラメータを制限することができます。ユーザーが指定するパラメータが [プレゼンテーション] タブに指定した制約を満たしていない場合、ワークフローは実行しません。

■ [プレゼンテーション] タブでの入力パラメータ ダイアログ ボックスの作成

ユーザーがワークフローを実行するときに入力パラメータを指定するダイアログ ボックスのレイアウトは、ワークフロー エディタの [プレゼンテーション] タブで定義します。

■ パラメータのプロパティの設定

Orchestrator では、ユーザーがワークフローを実行するとき指定する入力パラメータ値を制限するためのプロパティを定義できます。定義したパラメータのプロパティは、ユーザーが指定する入力パラメータのタイプや値に制限を課します。

[プレゼンテーション] タブでの入力パラメータ ダイアログ ボックスの作成

ユーザーがワークフローを実行するときに入力パラメータを指定するダイアログ ボックスのレイアウトは、ワークフロー エディタの [プレゼンテーション] タブで定義します。

[プレゼンテーション] タブでは、入力パラメータをカテゴリへとグループ化したり、入力パラメータ ダイアログ ボックスに表示するカテゴリの順序を定義したりできます。

プレゼンテーションの説明

パラメータごとまたはパラメータのグループごとに関連する説明を追加できます。この説明は、入力パラメータ ダイアログ ボックスに表示されます。説明では、ユーザーが適切な入力パラメータを選択できるように補足情報を提供します。HTML フォーマットを使用すると、説明テキストのレイアウトを拡張できます。

プレゼンテーションの入力ステップの定義

デフォルトでは、入力パラメータ ダイアログ ボックスにはすべての必須入力パラメータが 1 つのリストに表示されます。ユーザーによる入力パラメータへの入力をサポートするため、入力ステップと呼ばれるノードを [プレゼンテーション] タブで定義できます。入力ステップでは、類似している入力パラメータをグループ化します。入力ステップに含まれる入力パラメータは、ワークフローを実行するときに入力パラメータ ダイアログ ボックス内の異なるセクションに表示されます。

プレゼンテーション表示グループの定義

各入力ステップには、表示グループと呼ばれる専用のノードを設定できます。表示グループは、入力パラメータ ダイアログ ボックスのセクション内でパラメータ入力テキスト ボックスが表示される順序を定義します。入力ステップとは別に表示グループを定義できます。

入力パラメータ ダイアログ ボックスのプレゼンテーションの作成

ユーザーがワークフローを実行するときに入力パラメータを指定するダイアログ ボックスのプレゼンテーションは、ワークフロー エディタの [プレゼンテーション] タブで作成します。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 入力パラメータの定義済みリストがワークフローに存在することを確認します。

手順

- 1 ワークフロー エディタで [プレゼンテーション] タブをクリックします。
デフォルトでは、ワークフローのすべてのパラメータは作成順でメインの [プレゼンテーション] ノードに表示されます。
- 2 [プレゼンテーション] ノードを右クリックして [新しいステップの作成] を選択します。
[新規ステップ] ノードが [プレゼンテーション] ノードの下に表示されます。
- 3 ステップに適切な名前を付け、Enter キーを押します。
この名前は、ワークフローを実行するときに入力パラメータ ダイアログ ボックスのセクション ヘッダーとして表示されます。
- 4 入力ステップをクリックし、[プレゼンテーション] タブの下半分にある [全般] タブで説明を追加します。
この説明は、ユーザーが適切な入力パラメータを指定できるように補足の情報を提供するために入力パラメータ ダイアログ ボックスに表示されます。HTML フォーマットを使用すると、説明テキストのレイアウトを拡張できます。
- 5 作成した入力ステップを右クリックし、[表示グループの作成] を選択します。
[新規グループ] ノードが入力ステップ ノードの下に表示されます。

- 表示グループに適切な名前を付け、Enter キーを押します。

この名前は、ワークフローを実行するときに入力パラメータ ダイアログ ボックスのサブセクション ヘッダーとして表示されます。

- 表示グループをクリックし、[プレゼンテーション] タブの下半分にある [全般] タブで説明を追加します。

この説明は、入力パラメータ ダイアログ ボックスに表示されます。HTML フォーマットを使用すると、説明テキストのレイアウトを拡張できます。`${#param}` などの OGNL ステートメントを使用して、パラメータ値をグループの説明に追加できます。

- ワークフローを実行するときに入力パラメータ ダイアログ ボックスに表示されるすべての入力ステップおよび表示グループを作成するまで、前述の手順をくりかえします。

- [プレゼンテーション] ノードの下のパラメータを選択したステップおよびグループにドラッグします。

これで、ワークフローを実行するときユーザーが入力パラメータ値を指定する入力パラメータ ダイアログ ボックスのレイアウトを作成しました。

次に進む前に

パラメータのプロパティを設定する必要があります。

パラメータのプロパティの設定

Orchestrator では、ユーザーがワークフローを実行するとき指定する入力パラメータ値を制限するためのプロパティを定義できます。定義したパラメータのプロパティは、ユーザーが指定する入力パラメータのタイプや値に制限を課します。

どのパラメータも複数のプロパティを持つことができます。入力パラメータのプロパティは、[プレゼンテーション] タブの特定のパラメータのための [プロパティ] タブで定義します。

パラメータのプロパティは入力パラメータを検証し、入力パラメータ ダイアログ ボックスにテキスト ボックスが表示される方法を変更します。パラメータのプロパティには、パラメータ間の依存関係を作成できるものもあります。

静的および動的なパラメータのプロパティ値

パラメータのプロパティ値は、静的または動的のどちらかです。静的なプロパティ値は一定のままです。プロパティ値を静的として設定する場合は、パラメータ タイプに応じて、ワークフロー エディタが生成するリストからプロパティの値を設定または選択します。

動的なプロパティ値は、別のパラメータまたは属性の値に依存します。動的なプロパティが OGNL (Object Graph Navigation Language) 式を使用して値を取得するための関数を定義します。動的なパラメータのプロパティ値が別のパラメータのプロパティ値の値に依存しており、他のパラメータのプロパティ値が変化した場合、OGNL 式は動的なプロパティ値を再計算して変更します。

パラメータのプロパティの設定

ワークフローは開始時に、ユーザーからの入力パラメータの値を、設定されているすべてのパラメータのプロパティに対して検証します。



開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 入力パラメータの定義済みリストがワークフローに存在することを確認します。

手順

- 1 ワークフロー エディタで [プレゼンテーション] タブをクリックします。
- 2 [プレゼンテーション] タブでパラメータをクリックします。
そのパラメータの [全般] および [プロパティ] タブが [プレゼンテーション] タブの下部に表示されます。
- 3 パラメータの [プロパティ] タブをクリックします。
- 4 [プロパティ] タブ内で右クリックし、[プロパティの追加] を選択します。
ダイアログ ボックスが開き、選択されたタイプのパラメータに指定できるプロパティのリストが表示されます。
- 5 ダイアログ ボックスに表示されているリストからプロパティを選択し、[OK] をクリックします。
そのプロパティが [プロパティ] タブに表示されます。
- 6 [値] の下で、ドロップダウン メニューから対応する記号を選択することによって、そのプロパティ値を静的または動的のどちらかにします。

オプション	説明
	静的なプロパティ
	動的なプロパティ

- 7 プロパティ値を静的として設定する場合は、プロパティを設定しているパラメータのタイプに応じてプロパティ値を選択します。
- 8 プロパティ値を動的として設定する場合は、OGNL 式を使用してパラメータのプロパティ値を取得するための関数を定義します。
ワークフロー エディタでは、OGNL 式の記述に役立つサポートが提供されます。
 - a この式が呼び出すことのできる、ワークフローによって定義されたすべての属性およびパラメータのリストを取得するには、 アイコンをクリックします。
 - b プロパティを定義しているタイプの出力パラメータを返す Orchestrator API 内のすべてのアクションのリストを取得するには、 アイコンをクリックします。
提示されたパラメータやアクションのリスト内の項目をクリックすると、それらが OGNL 式に追加されます。
- 9 ワークフロー エディタの下部にある [保存] をクリックします。

ワークフローの入力パラメータのプロパティを定義しました。

次に進む前に

ワークフローを検証およびデバッグします。

Object Missing

This object is not available in the repository.

OGNL 式の事前定義済み定数値

OGNL 式を作成して動的パラメータのプロパティ値を取得する際、事前定義済み定数値を使用できます。

Orchestrator では OGNL 式で使用する次の定数を定義しています。

表 1-52. 事前定義済み OGNL 定数値

定数値	説明
<code>\${#__current}</code>	カスタム検証プロパティまたは式の一致プロパティの現在の値
<code>\${#__username}</code>	ワークフローを開始したユーザーのユーザー名前
<code>\${#__userdisplayname}</code>	ワークフローを開始したユーザーの表示名
<code>\${#__serverurl}</code>	ユーザーがワークフローを開始するサーバの IP アドレスを含む URL。この URL は、サーバの IP アドレスとルックアップ ポートで構成されます。 <code><{ServerIP}:{lookupPort}></code>
<code>\${#__datetime}</code>	現在の日時
<code>\${#__date}</code>	時間が 00:00:00 に設定された現在の日にち
<code>\${#__timezone}</code>	現在のタイムゾーン

(オプション) ワークフローの実行中にユーザー操作を要求する

ワークフローの実行中に外部ソースの追加の入力パラメータが必要となることがあります。別のアプリケーションまたはワークフローの入力パラメータを指定することも、ユーザーが直接指定することもできます。

たとえば、ワークフローの実行中に特定のイベントが発生した場合、実行する一連のアクションを決定するため、ワークフローはユーザー操作を要求することがあります。ユーザーが情報の要求に応答するか、待機時間がタイムアウト期間を超えるまで、ワークフローは続行を待機します。待機時間がタイムアウト期間を超えた場合、ワークフローは例外を返します。

ユーザー操作のデフォルトの属性は **security.group** および **timeout.date** です。**security.group** 属性を特定の LDAP ユーザー グループに設定すると、ユーザー操作の要求に応答する権限は、そのユーザー グループのメンバーに制限されます。

timeout.date 属性を設定すると、ワークフローがユーザーからの情報を待機する日時を設定できます。絶対日時を設定することも、スクリプト化したワークフロー要素を作成して現在の時刻に基づいて相対的に時刻を計算することもできます。

手順

1 ワークフローへのユーザー操作の追加

ワークフローに[ユーザー操作]スキーマ要素を追加すると、ワークフローの実行中にユーザーに対してパラメータの入力を求めることができます。[ユーザー操作]要素が見つかり、ワークフローは実行をサスペンドし、必要なデータをユーザーが入力するのを待機します。

2 ユーザー操作の `security.group` 属性の設定

ユーザー操作要素の `security.group` 属性は、ユーザー操作に応答する権限を持つユーザーまたはユーザーグループを設定します。

3 `timeout.date` 属性の絶対日時の設定

ユーザー操作の `timeout.date` 属性を設定し、ユーザーが操作に応答するのをワークフローが待機する期間を設定します。

4 ユーザー操作の相対タイムアウトの計算

`Date` オブジェクトを使用して、ユーザー操作がタイムアウトする相対日時を計算することができます。

5 `timeout.date` 属性の相対日時の設定

[ユーザー操作] 要素の `timeout.date` 属性は、`Date` オブジェクトにバインドすることにより相対日時を設定できます。オブジェクトはスクリプト関数で定義することができます。

6 ユーザー操作の外部入力の変数

ここでは、ワークフローの実行中にユーザー操作の入力パラメータとして指定する必要がある情報の設定方法について説明します。

7 ユーザー操作の例外の動作を定義する

指定されているタイムアウトの期間内にユーザーが入力パラメータを指定しなかった場合、ユーザー操作から例外が返されます。例外の動作は、スクリプト関数で定義することができます。

8 ユーザー操作の入力パラメータ ダイアログ ボックスの作成

ワークフローの実行中に入力パラメータのダイアログ ボックスで入力パラメータを指定する場合は、ワークフローの初回起動時に入力パラメータを指定する場合と同じ方法で指定します。

9 ユーザー操作の要求への応答

実行時にユーザー操作が必要となるワークフローは、必要な情報をユーザーが提供するまで、またはワークフローがタイムアウトするまでサスペンドします。

ワークフローへのユーザー操作の追加

ワークフローに[ユーザー操作]スキーマ要素を追加すると、ワークフローの実行中にユーザーに対してパラメータの入力を求めることができます。[ユーザー操作]要素が見つかり、ワークフローは実行をサスペンドし、必要なデータをユーザーが入力するのを待機します。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 ワークフロー スキーマ内の適切な場所へ [ユーザー操作] 要素をドラッグします。
- 2 [ユーザー操作] 要素の [編集] アイコン (✎) をクリックします。
- 3 [情報] タブにユーザー操作の名前と説明を入力し、[閉じる] をクリックします。
- 4 [保存] をクリックします。

これで、ワークフローにユーザー操作要素が追加されました。ワークフローがこの要素に到達すると、ユーザーが情報を入力するのを待機してから、実行を続行します。

次に進む前に

ユーザー操作の **security.group** 属性を設定して、ユーザー操作に応答する権限を特定のユーザーまたはユーザー グループに限定します。[「ユーザー操作の security.group 属性の設定」](#) を参照してください。

ユーザー操作の security.group 属性の設定

ユーザー操作要素の **security.group** 属性は、ユーザー操作に応答する権限を持つユーザーまたはユーザー グループを設定します。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素とユーザー操作をワークフロー スキーマに追加します。
- ユーザー操作の要求に応答する LDAP ユーザー グループを特定します。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 [ユーザー操作] の [属性] タブをクリックします。
- 3 [未設定] をクリックし、ユーザー操作に応答できるユーザーを **security.group** ソース パラメータで設定します。
- 4 (オプション) [NULL] を選択すると、すべてのユーザーがユーザー操作の要求に応答できます。
- 5 特定のユーザーまたはユーザー グループの応答する権限を制限するには、[ワークフローでパラメータ/属性を作成] をクリックします。
[パラメータ情報] ダイアログ ボックスが表示されます。
- 6 パラメータに名前を付けます。
- 7 [同じ名前でワークフロー属性を作成] を選択し、ワークフローに **LdapGroup** 属性を作成します。
- 8 パラメータの値で [未設定] をクリックし、[LdapGroup] 選択ボックスを表示します。

9 [フィルタ] テキスト ボックスで、LDAP ユーザー グループの名前を入力します。

10 リストから LDAP ユーザー グループを選択し、[選択] をクリックします。

たとえば、[管理者] グループを選択すると、このグループのメンバーのみがユーザー操作の要求に応答できるようになります。

ユーザー操作の要求に応答する権限は制限されます。

11 [OK] をクリックして [パラメータ情報] ダイアログ ボックスを閉じます。

ユーザー操作用に **security.group** 属性を設定します。

次に進む前に

timer.date 属性を設定して、ユーザー操作のタイムアウト期間を設定します。

- タイムアウト期間を絶対日時に設定するには、[「timeout.date 属性の絶対日時の設定」](#) を参照してください。
- 現在の日時に基づいて相対的にタイムアウト期間を計算するには、[「ユーザー操作の相対タイムアウトの計算」](#) を参照してください。

timeout.date 属性の絶対日時の設定

ユーザー操作の **timeout.date** 属性を設定し、ユーザーが操作に応答するのをワークフローが待機する期間を設定します。

Date オブジェクトを使用して、絶対日時に設定します。指定した日時になると、ユーザー操作を待機していたワークフローがタイムアウトになり、**Failed** 状態で終了します。たとえば、2 月 12 日の正午にユーザー操作がタイムアウトするように設定できます。現在の日時に基づいて相対的にタイムアウト期間を計算するには、[「ユーザー操作の相対タイムアウトの計算」](#) を参照してください。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- ユーザー操作要素をワークフロー スキーマに追加します。
- ユーザー操作用に **security.group** 属性を設定します。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 [ユーザー操作] の [属性] タブをクリックします。
- 3 [未設定] をクリックし、**timeout.date** ソース パラメータにタイムアウト パラメータ値を設定します。
- 4 (オプション) [NULL] を選択すると、ユーザーが操作に応答するまでワークフローは無期限に待機します。
- 5 [ワークフローでパラメータ/属性を作成] をクリックし、タイムアウト期間が過ぎるとワークフローが失敗するように設定します。

[パラメータ情報] ダイアログ ボックスが表示されます。

- 6 パラメータに名前を付けます。

- 7 [同じ名前で作成] を選択し、ワークフローに **Date** 属性を作成します。
- 8 パラメータの [値] で [未設定] をクリックします。
- 9 カレンダーを使用して、ユーザーが応答するまでワークフローが待機する絶対日時を選択します。
- 10 [OK] をクリックしてカレンダーを閉じます。
- 11 [OK] をクリックして [パラメータ情報] ダイアログ ボックスを閉じます。

timeout.date 属性に絶対日時を設定します。この日時前にユーザーが操作に応答しない場合、ワークフローはタイムアウトします。

次に進む前に

ユーザー操作でユーザーが指定する必要がある外部入力パラメータを定義します。[「ユーザー操作の外部入力の定義」](#)を参照してください。

ユーザー操作の相対タイムアウトの計算

Date オブジェクトを使用して、ユーザー操作がタイムアウトする相対日時を計算することができます。

Date オブジェクトを使用して、絶対日時を設定することができます。指定した日時になると、ユーザー操作の要求がタイムアウトします。別の方法として、ユーザーが定義した関数に従って日時を計算して相対的な **Date** オブジェクトを生成するワークフロー要素を作成することもできます。たとえば、現在の時刻に 24 時間を加算する相対的な **Date** オブジェクトを作成することができます。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- ユーザー操作要素をワークフロー スキーマに追加します。
- ユーザー操作に **security.group** 属性を設定します。

手順

- 1 [スクリプト化可能タスク] 要素を [汎用] メニューからワークフローのスキーマの、その **timeout.date** 属性に相対的な **Date** オブジェクトを必要とする要素の前にドラッグします。
- 2 ワークフロー スキーマで [スクリプト化可能タスク] 要素の [編集] アイコン (✎) をクリックします。
- 3 [情報] プロパティ タブで、スクリプト化されたワークフロー要素の名前と説明を指定します。
- 4 [出力] プロパティ タブをクリックし、[ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
- 5 [ワークフローでパラメータ/属性を作成] をクリックして、ワークフロー属性を作成します。
 - a 属性 **timerDate** に名前を付けます。
 - b 属性タイプのリストから **Date** を選択します。
 - c [同じ名前で作成] を選択します。

d この属性値はスクリプト関数によって提供されるため、[未設定] に設定されたままにします。

e [OK] をクリックします。

6 スクリプト化されたワークフロー要素の [スクリプティング] タブをクリックします。

7 [スクリプティング] タブのスクリプティング パッドで、**timerDate** という名前の **Date** オブジェクトを計算して生成するための関数を定義します。

たとえば、タイムアウト期間がミリ秒単位の相対的な遅延である次の JavaScript 関数を実装することによって **Date** オブジェクトを作成できます。

```
timerDate = new Date();
System.log( "Current date : '" + timerDate + "'" );
timerDate.setTime( timerDate.getTime() + (86400 * 1000) );
System.log( "Timer will expire at '" + timerDate + "'" );
```

前の JavaScript 関数の例では、**getTime** メソッドを使用して現在の日時を取得し、86,400,000 ミリ秒（つまり 24 時間）を追加する **Date** オブジェクトを定義します。[スクリプト化可能タスク] 要素は、この値を出力パラメータとして生成します。

8 [閉じる] をクリックします。

9 [保存] をクリックします。

これで、現在の日時に対する相対的な日時を計算して **Date** オブジェクトを生成する関数が作成されました。[ユーザー操作] 要素を使用して、この **Date** オブジェクトを入力パラメータとして取得し、ユーザーからの入力を待機する時間を設定することができます。ワークフローが [ユーザー操作] 要素に到達すると、ワークフローの実行が停止し、ユーザーが必要な情報を入力するまで、ワークフローが待機状態になります。ユーザーが何も情報を入力しないまま 24 時間が経過すると、ワークフローがタイムアウトします。

次に進む前に

Date オブジェクトを [ユーザー操作] 要素の **timeout.date** パラメータにバインドする必要があります。

[\[timeout.date 属性の相対日時の設定\]](#) を参照してください。

timeout.date 属性の相対日時の設定

[ユーザー操作] 要素の **timeout.date** 属性は、**Date** オブジェクトにバインドすることにより相対日時を設定できます。オブジェクトはスクリプト関数で定義することができます。

スクリプト関数に相対的な **Date** オブジェクトを作成すると、ユーザー操作の **timeout.date** 属性をこの **Date** オブジェクトにバインドできます。たとえば、現在の時刻に 24 時間を加算する **Date** オブジェクトに **timeout.date** 属性をバインドする場合、ユーザー操作は 24 時間待機した後にタイムアウトします。

開始する前に

- Add a user interaction element to the workflow schema.
- Set the **security.group** attribute for the user interaction.

- 相対日時を計算するスクリプト関数を作成し、ワークフローの **Date** オブジェクトでカプセル化します。[「ユーザー操作の相対タイムアウトの計算」](#) を参照してください。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 [ユーザー操作] の [属性] タブをクリックします。
- 3 [未設定] をクリックし、**timeout.date** ソース パラメータにタイムアウト パラメータ値を設定します。
- 4 スクリプト関数に定義した相対日時をカプセル化する **Date** オブジェクトを選択し、[選択] をクリックします。

timeout.date 属性にスクリプト関数で計算される相対日時を設定します。

次に進む前に

ユーザー操作でユーザーが指定する必要がある外部入力パラメータを定義します。[「ユーザー操作の外部入力の定義」](#) を参照してください。

ユーザー操作の外部入力の定義

ここでは、ワークフローの実行中にユーザー操作の入力パラメータとして指定する必要がある情報の設定方法について説明します。

ワークフローがユーザー操作要素に到達すると、そのユーザー操作で必要な情報をユーザーが入力するまで、ワークフローが停止状態になります。

開始する前に

- ユーザー操作要素をワークフロー スキーマに追加します。
- Set the **security.group** attribute for the user interaction.
- ユーザー操作用に **timer.date** 属性を設定します。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 [外部入力] タブをクリックします。
- 3 [ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックして、ユーザー操作で指定する必要があるパラメータを定義します。
- 4 (オプション) ワークフロー内に入力パラメータがすでに定義されている場合は、表示されたリストで適切なパラメータを選択します。
- 5 [ワークフローでパラメータ/属性を作成] をクリックして、ユーザーが指定する入力パラメータにバインドされるワークフロー属性を作成します。
- 6 パラメータに適切な名前を付けます。

- 7 [フィルタ] ボックスでオブジェクト タイプを検索し、タイプ リストで入力パラメータ タイプを選択します。
たとえば、入力パラメータとして仮想マシンを指定する必要があるユーザー操作の場合は **VC:VirtualMachine** を選択します。
- 8 [同じ名前でワークフロー属性を作成] を選択します。これにより、ユーザーが指定する入力パラメータが、ワークフロー内の新しい属性にバインドされます。
- 9 入力パラメータの値は、[未設定] のままにしておきます。
この値は、ワークフローの実行中にユーザー操作に対して応答するときに指定します。
- 10 [OK] をクリックして [パラメータ情報] ダイアログ ボックスを閉じます。

これで、ユーザー操作の実行時に指定する入力パラメータが定義されました。

次に進む前に

ユーザー操作でエラーが発生した場合の例外の動作を定義します。[「ユーザー操作の例外の動作を定義する」](#) を参照してください。

ユーザー操作の例外の動作を定義する

指定されているタイムアウトの期間内にユーザーが入力パラメータを指定しなかった場合、ユーザー操作から例外が返されます。例外の動作は、スクリプト関数で定義することができます。

ユーザー操作がタイムアウトになった場合にワークフローによって実行される処理が定義されていない場合、**Failed** 状態でワークフローが終了します。そのため、ワークフローを開発する場合は、例外の動作を定義することをお勧めします。

開始する前に

- ユーザー操作要素をワークフロー スキーマに追加します。
- ユーザー操作用に **security.group** 属性と **timer.date** 属性を設定します。
- ユーザー操作の外部入力パラメータを定義します。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 [例外] タブをクリックします。
- 3 出力例外のバインドで [未設定] をクリックします。
- 4 [ワークフローでパラメータ/属性を作成] をクリックして、ユーザー操作のバインド先となる例外属性を作成します。
[パラメータ情報] ダイアログ ボックスが表示されます。
- 5 **errorCode** 属性を作成します。

errorCode 属性には、以下に示すデフォルトのプロパティがあります。

- 名前: [errorCode]

- タイプ: 文字列
- 作成: [同じ名前でワークフロー属性を作成]
- 値: 適切なエラー メッセージを入力します。

6 [OK] をクリックして [パラメータ情報] ダイアログ ボックスを閉じます。

7 スクリプト化可能タスク要素を、ワークフロー スキーマ内のユーザー操作の上にドラッグします。

2つの要素の間に、例外リンクを示す赤い破線の矢印が表示されます。ユーザー操作のスクリプト化可能タスク要素が、**errorCode** 属性に自動的にバインドされます。

8 スクリプト化可能タスク要素をダブルクリックして、適切な名前を入力します。

たとえば、**ログのタイムアウト** などを入力します。

9 スクリプト化可能タスク要素の [スクリプト] タブで、例外を処理する JavaScript 関数を入力します。

たとえば、Orchestrator ログのタイムアウトを記録する場合は、以下の関数を入力します。

```
System.log("No response from user. Timed out.");
```

10 例外を処理するスクリプト化可能タスク要素を、ワークフロー内でその要素を追跡する要素にリンクしてバインドします。

たとえば、ワークフローをエラーで終了させる場合は、スクリプト化可能タスク要素を [例外のスロー] 要素にリンクしてバインドします。

これで、ユーザー操作がタイムアウトになった場合の例外の動作が定義されました。

次に進む前に

入力パラメータを指定するためのダイアログ ボックスを作成します。[「ユーザー操作の入力パラメータ ダイアログボックスの作成」](#)を参照してください。

ユーザー操作の入力パラメータ ダイアログ ボックスの作成

ワークフローの実行中に入力パラメータのダイアログ ボックスで入力パラメータを指定する場合は、ワークフローの初回起動時に入力パラメータを指定する場合と同じ方法で指定します。

ダイアログ ボックスのレイアウトは、ワークフロー全体の [プレゼンテーション] タブではなく、ユーザー操作要素の [プレゼンテーション] タブで作成します。ワークフロー全体の [プレゼンテーション] タブを使用すると、ワークフローの起動時に表示される入力パラメータ ダイアログ ボックスのレイアウトが作成されます。ユーザー操作要素の [プレゼンテーション] タブを使用すると、実行中のワークフローがユーザー操作要素に到達したときに表示される [入力パラメータ] ダイアログ ボックスのレイアウトが作成されます。

開始する前に

- ユーザー操作要素をワークフロー スキーマに追加します。
- ユーザー操作に **security.group** 属性と **timer.date** 属性を設定します。
- ユーザー操作の外部入力パラメータを定義します。

- 例外の動作を定義します。

手順

- 1 ワークフロー スキーマで [ユーザー操作] の [編集] アイコン (✎) をクリックします。
- 2 ユーザー操作要素の [プレゼンテーション] タブをクリックします。
[プレゼンテーション] タブには、ユーザー操作用に作成された外部の入力パラメータが表示されます。
- 3 (オプション) [プレゼンテーション] タブの [プレゼンテーション] ノードを右クリックして [新しいステップの作成] を選択します。
ステップを使用すると、ダイアログ ボックス内に複数のセクションを作成し、これらのセクションに説明やヘッダーを追加して、入力パラメータを見やすく配置することができます。
- 4 (オプション) [プレゼンテーション] タブの [プレゼンテーション] ノードを右クリックして [表示グループの作成] を選択します。
表示グループを使用すると、ステップ内に表示される入力パラメータの順序を変更したり、ダイアログ ボックスにサブヘッダーや説明を追加したりすることができます。
- 5 リスト内の入力パラメータをクリックし、そのパラメータの [全般] タブで、入力パラメータの説明を追加します。
ここで入力した説明テキストは、入力パラメータ ダイアログ ボックスにラベルとして表示されます。このラベルにより、ユーザー操作に応答する際に指定する必要がある情報を確認することができます。
- 6 入力パラメータのプロパティを定義します。
入力パラメータのプロパティにより、ユーザーが指定できる入力パラメータの値を定義したり、OGNL 式を使用してパラメータの値を動的に決定したりすることができます。
- 7 [保存して閉じる] をクリックして、ワークフロー エディタを終了します。

これで、入力パラメータのダイアログ ボックスが作成されました。ワークフローの実行中にユーザー操作に対して応答する場合は、このダイアログ ボックスを使用して入力パラメータを指定することになります。

次に進む前に

プレゼンテーションのステップとグループを作成する方法と、入力パラメータを設定する方法については、[「\[プレゼンテーション\] タブでの入力パラメータ ダイアログ ボックスの作成」](#)を参照してください。

ユーザー操作の要求への応答


実行時にユーザー操作が必要となるワークフローは、必要な情報をユーザーが提供するまで、またはワークフローがタイムアウトするまでサスペンドします。

ユーザー操作を必要とするワークフローでは、必要な情報を入力するユーザーを定義して操作の要求を指示します。

開始する前に

少なくとも 1 つのワークフローがユーザー操作の待機状態にあることを確認します。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[実行] を選択します。
- 2 Orchestrator クライアントの [My Orchestrator] ビューをクリックします。
- 3 [入力待機中] タブをクリックします。
[入力待機中] タブは、権限を持つユーザーまたはユーザー グループのメンバーからの入力を待機中のワークフローを一覧表示します。
- 4 入力待機中のワークフローをダブルクリックします。
入力待機中のワークフロー トークンは [ワークフロー] の階層リストに以下の記号とともに表示されます。 
- 5 ワークフロー トークンを右クリックして [応答] を選択します。
- 6 [入力パラメータ] ダイアログ ボックスの指示に従って、ワークフローに必要な情報を入力します。

実行中にユーザー入力を待機していたワークフローに情報が提供されました。

ワークフロー内でのワークフローの呼び出し

ワークフローは、実行中に他のワークフローを呼び出すことができます。ワークフローで他のワークフローを開始するケースには、呼び出し側ワークフローの実行のために、他のワークフローの結果が入力パラメータとして必要なケースと、ワークフローを開始して、そのワークフローに単独で実行を継続させるケースがあります。また、将来の所定の時間にワークフローを開始したり、複数のワークフローを同時に開始したりすることもできます。

■ ワークフローを呼び出すワークフロー要素

ワークフロー内から他のワークフローを呼び出す方法は 4 つあります。ワークフローを呼び出す方法はそれぞれ異なるワークフロー スキーマ要素で表されます。

■ ワークフローの同期呼び出し

ワークフローの同期呼び出しは、呼び出されたワークフローを呼び出し側ワークフローの実行の一部として実行します。呼び出し側ワークフローは、呼び出されたワークフローが後続のスキーマ要素を実行する場合、呼び出されたワークフローの出力パラメータを入力パラメータとして使用できます。

■ ワークフローの非同期呼び出し

ワークフローの非同期呼び出しは、呼び出し側ワークフローによって呼び出されたワークフローを個別に実行します。呼び出し側ワークフローは、呼び出されたワークフローの完了を待機せずに、実行を続けます。

■ ワークフローのスケジュール設定

ワークフローを別のワークフローから呼び出し、後の日時に開始するようにスケジュール設定できます。

■ リモート ワークフローを他のワークフロー内から呼び出すための前提条件

開発中のワークフローがリモートの Orchestrator サーバにある別のワークフローを呼び出す場合、リモートのワークフローが正しく実行されるように、特定の前提条件を満たす必要があります。

■ 複数のワークフローの同時呼び出し

複数のワークフローの同時呼び出しは、呼び出されたワークフローを呼び出し側ワークフローの実行の一部として同時に実行します。呼び出し側ワークフローは、呼び出されたすべてのワークフローが完了するのを待機してから、実行を続行します。呼び出し側ワークフローは、後続のスキーマ要素を実行するときに、呼び出されたワークフローの結果を入力パラメータとして使用できます。

ワークフローを呼び出すワークフロー要素

ワークフロー内から他のワークフローを呼び出す方法は 4 つあります。ワークフローを呼び出す方法はそれぞれ異なるワークフロー スキーマ要素で表されます。

同期ワークフロー

ワークフローは別のワークフローを同期的に開始できます。呼び出し先ワークフローは、呼び出し元ワークフローの実行の一部として実行され、呼び出し元ワークフローと同じメモリ領域で実行されます。呼び出し元ワークフローは、別のワークフローを開始し、呼び出し先ワークフローの実行が終了するのを待ってから、スキーマの次の要素の実行を開始します。通常、ワークフローは同期的に呼び出します。これは、呼び出し元ワークフローにおいて、後続のスキーマ要素のための入力パラメータとして呼び出し先ワークフローの出力が必要になるためです。たとえば、ワークフローで「仮想マシンの起動と待機」ワークフローを呼び出して仮想マシンを起動し、その仮想マシンの IP アドレスを取得して別の要素に渡したり、E メールでユーザーに送信したりすることができます。

非同期ワークフロー

ワークフローは別のワークフローを非同期的に開始できます。呼び出し元ワークフローは、別のワークフローを開始しますが、呼び出し先ワークフローの結果を待つことなく直ちにスキーマの次の要素を実行します。呼び出し先ワークフローは、呼び出し元ワークフローが定義した入力パラメータを使用して実行されますが、呼び出し先ワークフローのライフサイクルは呼び出し元ワークフローのライフサイクルから独立しています。非同期ワークフローでは、あるワークフローから次のワークフローに入力パラメータを渡すワークフローのチェーンを作成できます。たとえば、実行時にさまざまなオブジェクトが作成されるワークフローがあります。そのワークフローは、それらのオブジェクトが実行の入力パラメータとして使用される非同期ワークフローを開始することができます。元のワークフローは、必要なワークフローをすべて開始して残りの要素も実行したときに終了となります。ただし、元のワークフローが開始した非同期ワークフローは、元のワークフローとは関係なく実行が続行されます。

呼び出し元ワークフローが呼び出し先ワークフローの結果を待つようにするには、ネストされたワークフローを使用するか、スクリプト化可能タスク（呼び出し先ワークフローのワークフロー トークンの状態を取得して、完了したらワークフローの結果を取得する）を作成します。

スケジュール設定されたワークフロー

ワークフローは別のワークフローを呼び出すことができますが、そのワークフローの開始は後の日時まで先送りされます。呼び出し元ワークフローは、終了するまで実行が続行されます。スケジュール設定されたワークフローを呼び出すと、そのワークフローを特定の日に開始するタスクが作成されます。呼び出し元ワークフローが実行されたら、Orchestrator クライアントの [スケジュール] ビューおよび [My Orchestrator] ビューで、スケジュール設定されたワークフローを確認できます。

スケジュール設定されたワークフローの実行は 1 回だけです。ワークフローが周期的に実行されるようにスケジュール設定するには、同期ワークフローのスクリプト化可能タスク要素で **Workflow.scheduleRecurrently** メソッドを呼び出します。

ネストされたワークフロー

ワークフローは、単一のスキーマ要素に複数のワークフローをネストすることによって、複数のワークフローを同時に開始することができます。呼び出し元ワークフローがスキーマ内のネストされたワークフロー要素に到達すると、ネストされたワークフロー要素にリストされているすべてのワークフローが同時に開始されます。重要なのは、ネストされた各ワークフローは呼び出し元ワークフローのメモリ領域とは異なるメモリ領域で開始されるという点です。呼び出し元ワークフローは、ネストされたすべてのワークフローの実行が完了するのを待ってから、スキーマの次の要素の実行を開始します。したがって、呼び出し元ワークフローは、残りの要素を実行するときに、ネストされたワークフローの結果を入力パラメータとして使用できます。

他のワークフローへのワークフロー変更の伝達

ワークフローを別のワークフローから呼び出した場合、ワークフロー要素をスキーマに追加した瞬間に、Orchestrator が子ワークフローの入力パラメータを親ワークフローにインポートします。

子ワークフローを別のワークフローに追加した後に子ワークフローを変更した場合、親ワークフローは新しいバージョンの子ワークフローを呼び出しますが、新しい入力パラメータはインポートしません。ワークフローに対する変更が、それらを呼び出す他のワークフローの動作に影響しないようにするため、Orchestrator は呼び出し元のワークフローに新しい入力パラメータを自動的に伝達しません。

1 つのワークフローから、それを呼び出す他のワークフローにパラメータを伝達するには、ワークフローを呼び出しているワークフローを見つけて、ワークフローを手動で同期させる必要があります。

開始する前に

他のワークフローが呼び出すワークフローがあることを確認します。

手順

- 1 他のワークフローが呼び出すワークフローを変更して保存します。
- 2 ワークフロー エディタを閉じます。
- 3 Orchestrator クライアントの [ワークフロー] ビューの階層リストで、変更したワークフローに移動します。
- 4 ワークフローを右クリックし、[リファレンス] - [この要素を使用している要素の検索] の順に選択します。

このワークフローを呼び出すワークフローのリストが表示されます。

- 5 リストのワークフローをダブルクリックして、Orchestrator クライアントの [ワークフロー] ビューでこれをハイライトします。
- 6 ワークフローを右クリックして [編集] を選択します。
ワークフロー エディタが開きます。
- 7 ワークフロー エディタの [スキーマ] タブをクリックします。
- 8 ワークフロー スキーマから変更されたワークフローのワークフロー要素を右クリックし、[同期] - [パラメータの同期]の順に選択します。
- 9 確認ダイアログ ボックスで [続行] を選択します。
- 10 保存してワークフロー エディタを終了します。
- 11 変更されたワークフローを使用するすべてのワークフローについて、[手順 5](#) から[手順 10](#) を繰り返します。

変更されたワークフローを、それを呼び出す他のワークフローに伝達しました。

子ワークフローの入力パラメータおよびプレゼンテーションを親ワークフローに伝達する

他のワークフローの呼び出しを行うワークフローを開発している場合、子ワークフローの入力パラメータおよびプレゼンテーションを親ワークフローに伝達することができます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[実行] を選択します。
- 2 変更したいワークフローを右クリックして [編集] を選択します。
ワークフロー エディタが開きます。
- 3 [スキーマ] タブを選択します。
- 4 親ワークフローに伝達したい入力パラメータおよびプレゼンテーションを持つ子ワークフローの要素を右クリックして、[同期] - [プレゼンテーションの同期] を選択します。
- 5 確認ダイアログで [OK] を選択します。
- 6 (オプション) 親ワークフローに伝達したい入力パラメータおよびプレゼンテーションを持つすべての子ワークフローについて、[手順 4](#) と [手順 5](#) を繰り返します。

子ワークフローの入力パラメータが親ワークフローの入力パラメータに追加されます。親ワークフローのプレゼンテーションが子ワークフローのプレゼンテーションで拡張されます。

ワークフローの同期呼び出し

ワークフローの同期呼び出しは、呼び出されたワークフローを呼び出し側ワークフローの実行の一部として実行します。呼び出し側ワークフローは、呼び出されたワークフローが後続のスキーマ要素を実行する場合、呼び出されたワークフローの出力パラメータを入力パラメータとして使用できます。

[ワークフロー]要素を使用すると、別のワークフローからワークフローを同期的に呼び出せます。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [汎用] メニューの[ワークフロー]要素を、ワークフロー スキーマ内の適切な場所へドラッグします。
[ワークフローの選択] 選択ダイアログ ボックスが表示されます。
- 2 目的のワークフローを検索して選択し、[OK] をクリックします。
検索によって部分的な結果しか返されない場合は、検索条件を絞り込むか、またはクライアントの[ツール] - [ユーザー環境設定] メニューで、検索結果の数を増やします。
- 3 [ワークフロー]要素をクリックすると、[スキーマ] タブの下半分にプロパティ タブが表示されます。
- 4 ワークフロー スキーマで[ワークフロー]要素の [編集] アイコン (✎) をクリックします。
- 5 ワークフロー スキーマ要素の [入力] タブで、必要な入力パラメータをワークフローにバインドします。
- 6 ワークフロー スキーマ要素の [出力] タブで、必要な出力パラメータをワークフローにバインドします。
- 7 [例外] タブで、ワークフローの例外動作を定義します。
- 8 [閉じる] をクリックします。
- 9 ワークフロー エディタの下部にある [保存] をクリックします。

別のワークフローから、ワークフローを同期的に呼び出しました。ワークフローがその実行中に同期ワークフローに到達すると、その同期ワークフローが開始され、最初のワークフローは、同期ワークフローの完了を待機してから、その実行を続けます。

次に進む前に

ワークフローから、ワークフローを非同期で呼び出すことができます。

ワークフローの非同期呼び出し

ワークフローの非同期呼び出しは、呼び出し側ワークフローによって呼び出されたワークフローを個別に実行します。呼び出し側ワークフローは、呼び出されたワークフローの完了を待機せずに、実行を続けます。

[非同期ワークフロー]要素を使用すると、別のワークフローからワークフローを非同期で呼び出せます。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [汎用] メニューの[非同期ワークフロー]要素を、ワークフロー スキーマ内の適切な場所へドラッグします。
[ワークフローの選択] 選択ダイアログ ボックスが表示されます。

- 2 リストを検索して、目的のワークフローを選択し、[OK] をクリックします。
- 3 ワークフロー スキーマで[非同期ワークフロー]要素の [編集] アイコン (✎) をクリックします。
- 4 非同期ワークフロー要素の [入力] タブで、必要な入力パラメータをワークフローにバインドします。
- 5 非同期ワークフロー要素の [出力] タブで、必要な出力パラメータをバインドします。

出力パラメータは、呼び出されたワークフローにバインドすることもできれば、そのワークフローの結果にバインドすることもできます。

- 呼び出されたワークフローを出力パラメータとして返すには、呼び出されたワークフローにバインドします。
- 呼び出されたワークフローの実行結果を返すには、呼び出されたワークフローのワークフロー トークンにバインドします。

- 6 [例外] タブで、非同期ワークフロー要素の例外動作を定義します。
- 7 [閉じる] をクリックします。
- 8 ワークフロー エディタの下部にある [保存] をクリックします。

別のワークフローから、ワークフローを非同期的に呼び出しました。ワークフローがその実行中に非同期ワークフローに到達すると、その非同期ワークフローが開始され、最初のワークフローは、非同期ワークフローの完了を待機せずに実行を続けます。

次に進む前に

ワークフローを別の日時に開始するようにスケジュール設定できます。

ワークフローのスケジュール設定

ワークフローを別のワークフローから呼び出し、後の日時に開始するようにスケジュール設定できます。

ワークフローは、[ワークフローのスケジュール設定] 要素を使用して別のワークフローでスケジュール設定します。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [ワークフローのスケジュール設定] 要素を [汎用] メニューからワークフロー スキーマ内の適切な位置にドラッグします。
- 2 呼び出すワークフローを、その名前の一部をテキスト ボックスに入力することによって検索します。
- 3 リストからワークフローを選択し、[OK] をクリックします。
- 4 ワークフロー スキーマで [ワークフローのスケジュール設定] 要素の [編集] アイコン (✎) をクリックします。

5 [入力] プロパティ タブをクリックします。

workflowScheduleDate という名前のパラメータが、定義するプロパティのリストに、呼び出し側ワークフローの入力パラメータと共に表示されます。

6 **workflowScheduleDate** パラメータの [未設定] をクリックして、パラメータを設定します。

7 [ワークフローでパラメータ/属性を作成] をクリックして、パラメータを作成し、パラメータ値を設定します。

8 [値] の [未設定] をクリックして、パラメータ値を設定します。

9 表示されるカレンダーを使用して、スケジュール設定されたワークフローを開始する日時を設定し、[OK] をクリックします。

10 スケジュール設定されたワークフロー要素の [入力] タブで、残りの入力パラメータをスケジュール設定されたワークフローにバインドします。

11 スケジュール設定されたワークフロー要素の [出力] タブで、必要な出力パラメータを **Task** オブジェクトにバインドします。

12 [例外] タブで、スケジュール設定されたワークフロー要素の例外の動作を定義します。

13 [閉じる] をクリックします。

14 ワークフロー エディタの下部にある [保存] をクリックします。

別のワークフローから、特定の日時に開始するようにワークフローをスケジュール設定しました。

次に進む前に

ワークフローから複数のワークフローを同時に呼び出すことができます。

リモート ワークフローを他のワークフロー内から呼び出すための前提条件

開発中のワークフローがリモートの Orchestrator サーバにある別のワークフローを呼び出す場合、リモートのワークフローが正しく実行されるように、特定の前提条件を満たす必要があります。

- リモート ワークフローのすべての入力パラメータは、リモートの Orchestrator サーバ上で解決可能である必要があります。
- リモート ワークフローのすべての出力パラメータは、ローカルの Orchestrator サーバ上で解決可能である必要があります。

リモート ワークフローのパラメータを確実に解決可能にするには、ワークフローが使用するインベントリ オブジェクトをリモートとローカルの Orchestrator サーバの両方で使用可能にする必要があります。リモートのワークフローでプラグインからのオブジェクトを使用する場合、プラグインは両方の Orchestrator サーバで使用できる必要があります。リモートのプラグインとローカルのプラグインのインベントリは同一である必要があります。リモートのワークフローが、ワークフローやアクションなど Orchestrator のシステム オブジェクトを使用する場合、同じワークフローおよびアクションがリモートとローカルの Orchestrator サーバに存在する必要があります。

たとえば、ネストされたワークフロー要素内で「仮想マシンの名前の変更」ワークフローを選択する場合を想定します。リモートの Orchestrator サーバで「仮想マシンの名前の変更」ワークフローを実行するとします。テストワークフローを実行する場合、「仮想マシンの名前の変更」ワークフローがテストワークフローの実行中に呼び出しされます。ローカルの Orchestrator サーバのインベントリから、名前を変更する仮想マシンを指定します。仮想マシンの

名前の変更ワークフローがリモートの Orchestrator サーバで実行されているため、同じ仮想マシンがそのサーバのインベントリで使用できる必要があります。それ以外の場合、仮想マシンの名前の変更ワークフローは **vm** 入力パラメータを解決することはできません。したがって、ローカルとリモートの Orchestrator サーバの vCenter Server プラグインは同じ vCenter Server インスタンスに接続されている必要があります。

複数のワークフローの同時呼び出し

複数のワークフローの同時呼び出しは、呼び出されたワークフローを呼び出し側ワークフローの実行の一部として同時に実行します。呼び出し側ワークフローは、呼び出されたすべてのワークフローが完了するのを待機してから、実行を続行します。呼び出し側ワークフローは、後続のスキーマ要素を実行するときに、呼び出されたワークフローの結果を入力パラメータとして使用できます。

[ネストされたワークフロー] 要素を使用すると、別のワークフローから複数のワークフローを同時に呼び出せます。また、ネストされたワークフローは、呼び出し側ワークフローのユーザーの認証情報とは異なるユーザー認証情報が設定されたワークフローを実行するために使用することもできます。

開始する前に

- ワークフロー エディタで編集対象のワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [アクションとワークフロー] メニューの [ネストされたワークフロー] 要素を、ワークフロー スキーマ内の適切な位置へドラッグします。

[ワークフローの選択] 選択ダイアログ ボックスが表示されます。

- 2 開始するワークフローを検索して選択し、[OK] をクリックします。
- 3 ワークフロー スキーマで [ネストされたワークフロー] 要素の [編集] アイコン (✎) をクリックします。
- 4 [ワークフロー] タブをクリックします。

手順 2 で選択したワークフローがタブに表示されます。

- 5 [ワークフロー] スキーマ要素のプロパティ タブの右側パネル内にある [入力] タブと [出力] タブで、このワークフローの IN バインドと OUT バインドを設定します。
- 6 [ワークフロー] スキーマ要素のプロパティ タブの右側パネル内にある [接続情報] をクリックします。
[接続情報] タブでは、適切な認証情報を使用して、ローカル サーバとは異なるサーバに保存されているワークフローにアクセスすることができます。
- 7 リモート サーバ上のワークフローにアクセスするには、[リモート] を選択して、[未設定] をクリックし、リモート サーバのホスト名または IP アドレスを入力します。

注意 vRealize Orchestrator マルチノード プラグインを使用して、リモート サーバ上のワークフローを呼び出すことができます。

- 8 リモート サーバへのアクセスに使用する認証情報を設定します。
 - 呼び出し側ワークフローを実行するユーザーの認証情報を使用するには、[継承] を選択します。
 - 一連の動的な認証情報を選択するには、[動的] を選択し、[未設定] をクリックします。一連の動的な認証情報は、ワークフロー内の別の箇所で、**credentials** タイプのパラメータを使用して設定します。
 - 認証情報を直接入力するには、[固定] を選択し、[未設定] をクリックします。
- 9 [ワークフロー] タブの [ワークフローの追加] ボタンをクリックして、ネストされたワークフロー要素にさらに追加するワークフローを選択します。
- 10 手順 2 から手順 8 までは繰り返して、追加するワークフローごとに設定を行います。
- 11 ワークフロー スキーマでネストされたワークフロー要素をクリックします。

要素内でネストされたワークフローの数が、ネストされたワークフロー要素上に数値として表示されます。

ワークフローから複数のワークフローを同時に呼び出しました。

次に進む前に

実行時間の長いワークフローを定義できます。

選択したオブジェクトに対するワークフローの実行

選択したオブジェクトに対してワークフローを実行することにより、繰り返しタスクを自動化することができます。たとえば、仮想マシン フォルダ内のすべての仮想マシンのスナップショットを作成するワークフローを作成したり、特定のホスト上に存在するすべての仮想マシンをパワーオフするワークフローを作成したりできます。

以下のいずれかの方法を使用し、選択したオブジェクトに対してワークフローを実行できます。

- [ライブラリ] - [vCenter] - [バッチ] - [選択したオブジェクトに対するワークフローの実行] ワークフローを実行する。
- [ライブラリ] - [Orchestrator] - [ワークフローの順次開始] ワークフローまたは [ワークフローの同時開始] ワークフローを呼び出すワークフローを作成する。
- オブジェクトの配列を取得するワークフローを作成し、ワークフロー要素のループ内の配列にある各オブジェクトに対してワークフローを実行する。
- ワークフロー内のスクリプト化した要素にある **For** ループ内の **Workflow.execute()** メソッドを呼び出して、JavaScript からワークフローを実行する。

選択したオブジェクトに対してワークフローを実行する方法は、実行するワークフローによって異なります。また、方法によってはワークフローのパフォーマンスに影響が及ぶ可能性があります。たとえば、複数のオブジェクトに対してワークフローを実行する場合、「選択したオブジェクトに対するワークフローの実行」ワークフローを実行することが最も簡単な方法です。この場合、ワークフローを開発する必要はありませんが、1 つの入力パラメータを取得するワークフローしか実行できません。

「ワークフローの順次開始」ワークフローまたは「ワークフローの同時開始」ワークフローを呼び出すワークフローを作成すると、複数の入力パラメータを取得する複数のオブジェクトワークフローを実行できます。呼び出すワークフローは、プロパティ配列を作成し、入力パラメータを「ワークフローの順次開始」ワークフローまたは「ワークフローの同時開始」ワークフローに渡す必要があります。これらのワークフローは他のワークフローでのみ使用します。直接実行しないようにしてください。

スクリプト化した要素の **For** ループ内にあるワークフローの実行は、ワークフロー要素のループ内にあるワークフローを実行する場合よりも高速に行われますが、柔軟性が低く再利用の可能性が制限されます。最も重要なことは、スクリプト化したループ内にあるワークフローを実行すると、ワークフローの実行中に Orchestrator が各要素を起動する際に実行するチェックポイント処理が行われなくなることです。このため、スクリプト化したループの実行中に Orchestrator サーバが停止した場合、サーバが再起動されると、スクリプト化した要素の初めからワークフローが再開されるため、ループ全体が繰り返されることになります。ワークフロー要素のループが含まれているワークフローの実行中に Orchestrator サーバが停止した場合は、サーバが停止したときに実行されていたループ内の特定の要素からワークフローが再開されます。

バッチ ワークフローの詳細については、『VMware vRealize Orchestrator プラグインの使用』を参照してください。

ワークフロー要素のループ内にあるオブジェクトの配列に対してワークフローを実行するワークフローを作成する方法については、『[複合ワークフローの開発](#)』を参照してください。

スクリプト化した **For** ループ内のワークフローを実行する方法については、『[ワークフローのスクリプティング サンプル](#)』に記載されています。

「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローの実装

「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローを使用すると、選択したオブジェクトに対してワークフローを実行できます。

「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローを直接実行することはできません。これらのワークフローは、ワークフローをもう 1 つ作成して、その中に含める必要があります。「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローを使用して、選択したオブジェクトに対してワークフローを実行するには、ワークフローの実行対象のオブジェクトを取得する必要があります。これらのオブジェクト、およびワークフローに必要なその他の入力パラメータを、プロパティの配列としてワークフローに渡します。「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローは、選択されたオブジェクトに対してワークフローを実行した結果を、**WorkflowToken** オブジェクトの配列として出力します。

「ワークフローの順次開始」ワークフローと「ワークフローの同時開始」ワークフローは、同じ方法で実装します。「ワークフローの順次開始」ワークフローは、ワークフローを各オブジェクトに対して順次実行します。「ワークフローの同時開始」ワークフローは、ワークフローを各オブジェクトに対して同時実行します。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフローのスキーマで、スクリプト化可能なタスク要素またはアクションを追加して、ワークフローを実行するオブジェクトのリストを取得します。

たとえば、仮想マシン フォルダ内のすべての仮想マシンに対してワークフローを実行するには、ワークフローに **getAllVirtualMachinesByFolder** アクションを追加します。

- 2 スクリプト化した要素またはアクションをリンクし、スクリプト化した要素またはアクションの入出力をワークフローの入力または属性にバインドします。

たとえば、**getAllVirtualMachinesByFolder** アクションの **vmFolder** 入力をワークフロー入力パラメータにバインドし、**actionResult** 出力を呼び出し側ワークフローのワークフロー属性にバインドします。

- 3 スクリプト化可能なタスク要素を追加して、オブジェクトのリストをプロパティの配列にキャストします。

たとえば、ワークフローを実行する対象のオブジェクトが、**getAllVirtualMachinesByFolder** アクションの **actionResult** 出力によって返される仮想マシンの配列 **allVMs** である場合は、次のスクリプトを記述して、オブジェクトをプロパティの配列にキャストします。

```
propsArray = new Array();

for each (var vm in allVMs) {
    var prop = new Properties();
    prop.put("vm", vm);
    propsArray.push(prop);
}
```

- 4 スクリプト化可能なタスク要素の入出力をワークフロー属性にバインドします。

[手順 3](#) のスクリプト化可能なタスク要素の例では、入力を仮想マシンの **allVMs** 配列にバインドし、**propsArray** 出力属性を **Properties** オブジェクトの配列として作成します。

- 5 ワークフロー要素をワークフロー スキーマに追加します。

- 6 「ワークフローの順次開始」 ワークフローまたは「ワークフローの同時開始」 ワークフローを選択し、ワークフロー要素をその他の要素にリンクします。

- 7 「ワークフローの順次開始」 ワークフローまたは「ワークフローの同時開始」 ワークフローの **wf** 入力を、オブジェクトに対して実行するワークフローにバインドします。

たとえば、**getAllVirtualMachinesByFolder** アクションによって返されるすべての仮想マシンのあらゆるスナップショットを削除するには、「すべてのスナップショットを削除」 ワークフローを選択します。

- 8 「ワークフローの順次開始」 ワークフローまたは「ワークフローの同時開始」 ワークフローの **parameters** 入力を、ワークフローを実行する対象のオブジェクトが含まれている **Properties** オブジェクトの配列にバインドします。

たとえば、**parameters** 入力を、[手順 4](#) で定義した **propsArray** 属性にバインドします。

- 9 (オプション) 「ワークフローの順次開始」 ワークフローまたは「ワークフローの同時開始」 ワークフローの **workflowTokens** 出力を、ワークフロー内の属性にバインドします。

10 (オプション) 引き続き、「ワークフローの順次開始」ワークフローまたは「ワークフローの同時開始」ワークフローの実行結果を使用する要素をさらに追加します。

これで、「ワークフローの順次開始」ワークフローまたは「ワークフローの同時開始」ワークフローを使用するワークフローが作成され、選択したオブジェクトに対してワークフローが実行されるようになりました。

長期実行ワークフローの開発

待機中のワークフローは、応答を必要とするオブジェクトを絶えずポーリングするためシステム リソースを消費します。ワークフローが必要な応答を受け取るまでの待機時間が長くなる可能性があることがわかっている場合、ワークフローに長期実行ワークフロー要素を追加できます。

実行中のすべてのワークフローはシステム スレッドを消費します。ワークフローが長期実行ワークフロー要素に到達すると、長期実行ワークフロー要素はワークフローをパッシブ状態に設定します。次に長期実行ワークフローはワークフロー情報を単一スレッドに渡します。この単一スレッドが、サーバ上で実行中のすべての長期実行ワークフロー要素を調べるためにシステムをポーリングします。個々の長期実行ワークフロー要素がシステムからの情報の取得を絶えず行う代わりに、長期実行ワークフロー要素を指定された期間だけパッシブ状態にし、その代理として長期実行ワークフロー スレッドがシステムをポーリングします。

待機する期間は次のいずれかの方法で設定できます。

- 特定の日時までワークフローを一時停止する、**Date** オブジェクトでカプセル化したタイマーを設定します。スキーマに [待機中のタイマー] 要素を含めることによって、タイマーに基づく長期実行ワークフロー要素を実装します。
- トリガー イベントの発生後にワークフローを再開する、**Trigger** オブジェクトでカプセル化したトリガー イベントを定義します。スキーマに [待機中のイベント] 要素または [ユーザー操作] 要素を追加することによって、トリガーに基づく長期実行ワークフロー要素を実装します。

タイマーに基づいたワークフローでの相対日時の設定

待機中のタイマー要素の **timer.date** 属性は、**Date** オブジェクトにバインドすることによって相対日時に設定できます。**Date** オブジェクトは、スクリプト関数で定義します。

特定の日付の時刻に達すると、タイマーに基づいた長時間実行されるワークフローは再アクティブ化され、その実行を続行します。たとえば、ワークフローを 2 月 12 日の正午に再アクティブ化されるように設定できます。別の方法として、ユーザーが定義した関数に従って日時を計算して相対的な **Date** オブジェクトを生成するワークフロー要素を作成することもできます。たとえば、現在の時刻に 24 時間を加算する相対的な **Date** オブジェクトを作成することができます。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [スクリプト化可能タスク] 要素を [汎用] メニューからワークフローのスキーマの、その **timeout.date** 属性に相対的な **Date** オブジェクトを必要とする要素の前にドラッグします。
- 2 ワークフロー スキーマで [スクリプト化可能タスク] 要素の [編集] アイコン (✎) をクリックします。
- 3 [情報] プロパティ タブで、スクリプト化されたワークフロー要素の名前と説明を指定します。
- 4 [出力] プロパティ タブをクリックし、[ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
- 5 [ワークフローでパラメータ/属性を作成] をクリックして、ワークフロー属性を作成します。
 - a 属性 **timerDate** に名前を付けます。
 - b 属性タイプのリストから **Date** を選択します。
 - c [同じ名前でワークフロー属性を作成] を選択します。
 - d この属性値はスクリプト関数によって提供されるため、[未設定] に設定されたままにします。
 - e [OK] をクリックします。
- 6 スクリプト化されたワークフロー要素の [スクリプティング] タブをクリックします。
- 7 [スクリプティング] タブのスクリプティング パッドで、**timerDate** という名前の **Date** オブジェクトを計算して生成するための関数を定義します。

たとえば、タイムアウト期間がミリ秒単位の相対的な遅延である次の JavaScript 関数を実装することによって **Date** オブジェクトを作成できます。

```
timerDate = new Date();
System.log( "Current date : " + timerDate + " " );
timerDate.setTime( timerDate.getTime() + (86400 * 1000) );
System.log( "Timer will expire at " + timerDate + " " );
```

前の JavaScript 関数の例では、**getTime** メソッドを使用して現在の日時を取得し、86,400,000 ミリ秒（つまり 24 時間）を追加する **Date** オブジェクトを定義します。[スクリプト化可能タスク] 要素は、この値を出力パラメータとして生成します。

- 8 [閉じる] をクリックします。
- 9 [保存] をクリックします。

Date オブジェクトを計算して生成するための関数を作成しました。[待機中のタイマー] 要素は、長時間実行されるワークフローをこの **Date** オブジェクト内にカプセル化された日付までサスペンドするために、このオブジェクトを入力パラメータとして受信できます。ワークフローは、[待機中のタイマー] 要素に達すると、その実行をサスペンドし、24 時間待ってから続行します。

次に進む前に

タイマーに基づいた長時間実行されるワークフローを実装するには、ワークフローに [待機中のタイマー] 要素を追加する必要があります。

タイマーベースの長期実行ワークフローの作成

ワークフローで予測できる期間、外部ソースからの応答を待機する必要があることがわかっている場合は、ワークフローをタイマーベースの長期実行ワークフローとして実装できます。タイマーベースの長期実行ワークフローは、指定された日時まで待機してから、実行を再開します。

タイマーベースの長期実行ワークフローとしてワークフローを実装するには、[タイマーの待機] 要素を使用します。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 ワークフロー スキーマ内でワークフローの実行を中断する位置に、[汎用] メニューから [タイマーの待機] 要素をドラッグします。

スクリプト化可能タスクを実装して日時を計算する場合は、この要素を [タイマーの待機] 要素の前に置く必要があります。
- 2 ワークフロー スキーマ内で [タイマーの待機] 要素の [編集] アイコン (✎) をクリックします。
- 3 [情報] プロパティ タブにタイマーを実装する理由の説明を入力します。
- 4 [属性] プロパティ タブをクリックします。

属性のリストに **timer.date** パラメータが表示されます。
- 5 **timer.date** パラメータの [未設定] ボタンをクリックして、パラメータを適切な **Date** オブジェクトにバインドします。

[タイマーの待機] 選択ダイアログ ボックスが開き、選択できるバインドのリストが表示されます。
 - 表示されたリストから、あらかじめ定義した **Date** オブジェクトを選択します。たとえば、[スクリプト化可能タスク] 要素を使用してワークフローの別の場所で定義したオブジェクトを選択します。
 - または、ワークフローが待機する特定の日時を設定する **Date** オブジェクトを作成します。
- 6 (オプション) ワークフローが待機する特定の日時を設定する **Date** オブジェクトを作成します。
 - a [タイマーの待機] 選択ダイアログ ボックスで、[ワークフローでパラメータ/属性を作成] をクリックします。
[パラメータ情報] ダイアログ ボックスが表示されます。
 - b パラメータに適切な名前を付けます。
 - c タイプは **Date** に設定されたままにします。
 - d [同じ名前でワークフロー属性を作成] をクリックします。

e [値] プロパティの [未設定] ボタンをクリックして、パラメータの値を設定します。
カレンダーが表示されます。

f カレンダーを使用して、ワークフローを再開する日時を設定します。

g [OK] をクリックします。

7 [閉じる] をクリックします。

8 ワークフロー エディタの下部にある [保存] をクリックします。

タイマーベースの長期実行ワークフローを、設定した日時まで中断するタイマーを定義しました。

次に進む前に

トリガ イベントを待機してから続行する長期実行ワークフローを作成することができます。

トリガ オブジェクトの作成

トリガ オブジェクトは、プラグインで定義するイベントトリガを監視します。たとえば、vCenter Server プラグインは、トリガ イベントを **Task** オブジェクトとして定義します。タスクが終了すると、トリガは、待機しているトリガベースの長期実行ワークフロー要素にメッセージを送信し、ワークフローが再開されます。

トリガベースの長期実行ワークフローが待機する、時間のかかるイベントは、**VC:Task** オブジェクトを返す必要があります。たとえば、仮想マシンを開始する **startVM** アクションが **VC:Task** オブジェクトを返すことによって、ワークフロー内の後続の要素がその進行状況を監視することができます。トリガベースの長期実行ワークフローのトリガ イベントは、この **VC:Task** オブジェクトを入力パラメータとして必要とします。

Trigger オブジェクトは、[スクリプト化可能タスク] 要素の JavaScript 関数内に作成します。この [スクリプト化可能タスク] 要素は、トリガ イベントを待機する、トリガベースの長期実行ワークフローの一部にすることができます。または、トリガベースの長期実行ワークフローに入力パラメータを渡す別のワークフローの一部にすることもできます。トリガ関数は、Orchestrator API の **createEndOfTaskTrigger()** メソッドを実装する必要があります。

重要 すべてのトリガにはタイムアウト期間を定義する必要があります。定義しないと、ワークフローが無期限に待機することがあります。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。
- ワークフローで、**VC:Task** オブジェクトを属性または入力パラメータとして宣言します。たとえば、仮想マシンの開始やクローン作成を行うワークフローまたはワークフロー要素の **VC:Task** オブジェクトとして宣言します。

手順

- 1 ワークフローのスキーマに、[汎用] メニューから [スクリプト化可能タスク] 要素をドラッグします。
[スクリプト化可能タスク] に先行する要素のいずれかで、**VC:Task** オブジェクトを出力パラメータとして生成する必要があります。
- 2 ワークフロー スキーマで [スクリプト化可能タスク] 要素の [編集] アイコン (✎) をクリックします。
- 3 [情報] プロパティ タブに、トリガの名前と説明を入力します。
- 4 [入力] プロパティ タブをクリックします。
- 5 [ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
入力パラメータを選択するダイアログ ボックスが開きます。
- 6 タイプ [VC:Task] の入力パラメータを選択または作成します。
この **VC:Task** オブジェクトは、別のワークフローまたは要素が開始する、時間のかかるイベントを表します。
- 7 (オプション) タイムアウト期間を秒単位で定義する、数値タイプの入力パラメータを選択または作成します。
- 8 [出力] プロパティ タブをクリックします。
- 9 [ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
出力パラメータを選択するダイアログ ボックスが開きます。
- 10 次のプロパティを持つ出力パラメータを作成します。
 - a 値 **trigger** を持つ名前プロパティを作成します。
 - b 値 **Trigger** を持つタイプ プロパティを作成します。
 - c [同じ名前で作成] をクリックし、属性を作成します。
 - d 値を [未設定] のままにします。
- 11 例外の動作があれば、[例外] プロパティ タブで定義します。
- 12 [スクリプティング] タブで、**Trigger** オブジェクトを生成する関数を定義します。

たとえば、次の JavaScript 関数を実装して **Trigger** オブジェクトを作成することができます。

```
trigger = task.createEndOfTaskTrigger(timeout);
```

createEndOfTaskTrigger() メソッドは、**VC:Task** オブジェクトで指定した **task** を監視する **Trigger** オブジェクトを返します。

- 13 [閉じる] をクリックします。
- 14 ワークフロー エディタの下部にある [保存] をクリックします。

トリガベースの長期実行ワークフロー向けのトリガ イベントを作成するワークフロー要素を定義しました。トリガ要素は、**Trigger** オブジェクトを出力パラメータとして生成します。このオブジェクトに [待機イベント] 要素をバインドできます。

次に進む前に

このトリガ イベントを、トリガベースの長期実行ワークフロー内の [待機イベント] 要素にバインドする必要があります。

トリガベースの長時間ワークフローの作成

ワークフローの実行中に外部ソースの応答を待機する必要があることがわかっているものの、その待機する時間がわからない場合は、ワークフローをトリガベースの長時間ワークフローとして実装できます。トリガベースの長時間ワークフローは、定義したトリガ イベントが発生するのを待機してから再開します。

トリガベースの長時間ワークフローとしてワークフローを実装するには、[待機イベント] 要素を使用します。トリガベースの長時間ワークフローが [待機中のイベント] 要素に到達すると、実行がサスペンドして、トリガからメッセージを受信するまでパッシブ状態で待機します。待機時間中はパッシブワークフローによってスレッドが消費されませんが、長時間ワークフロー要素は、サーバのすべての長時間ワークフローを監視している単ースレッドに対してワークフロー情報を渡します。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。
- **Trigger** オブジェクトにカプセル化されるトリガ イベントを定義します。

手順

- 1 [汎用] メニューから [待機イベント] 要素をワークフロー スキーマ内でワークフローの実行をサスペンドする位置にドラッグします。

トリガを宣言するスクリプト化可能タスクは、[待機イベント] 要素の直前に配置する必要があります。

- 2 ワークフロー スキーマ内で [待機イベント] 要素の [編集] アイコン (✎) をクリックします。

- 3 [情報] プロパティ タブに待機する理由の説明を入力します。

- 4 [属性] プロパティ タブをクリックします。

trigger.ref パラメータが属性のリストに表示されます。

- 5 **trigger.ref** パラメータの [未設定] リンクをクリックして、パラメータを適切な **Trigger** オブジェクトにバインドします。

[待機イベント] 選択ダイアログ ボックスが開き、バインド可能なパラメータのリストが表示されます。

- 6 表示されたリストで事前定義されている **Trigger** オブジェクトを選択します。

この **Trigger** オブジェクトは、別のワークフローまたはワークフロー要素で定義されているトリガ イベントを示します。

- 7 [例外] プロパティ タブで例外の動作を定義します。

- 8 [閉じる] をクリックします。

9 ワークフロー エディタの下部にある [保存] をクリックします。

トリガベースの長時間ワークフローをサスペンドするワークフロー要素を定義しました。このワークフロー要素は、特定のトリガ イベントを待機してから再開します。

次に進む前に

これで、ワークフローを実行できるようになりました。

構成要素

構成要素は、展開した Orchestrator サーバ全体で、定数の構成に使用できる属性のリストです。

特定の Orchestrator サーバで実行中のすべてのワークフロー、アクション、およびポリシーは、構成要素で設定した属性を使用することができます。構成要素で属性を設定すると、Orchestrator サーバで実行中のすべてのワークフロー、アクション、ポリシーで同じ属性値を使用できるようになります。

構成要素の属性を使用するワークフロー、アクション、またはポリシーを含むパッケージを作成すると、Orchestrator は、パッケージ内にその構成要素を自動的に含めます。構成要素を含むパッケージを別の Orchestrator サーバにインポートすると、構成要素の属性値もインポートすることができます。たとえば、実行元の Orchestrator サーバによって異なる属性値が必要なワークフローを作成する場合に、構成要素内でこれらの属性を設定すると、そのワークフローをエクスポートして、別の Orchestrator サーバで使用することができます。このため、構成要素を使用すると、サーバ間でワークフロー、アクション、ポリシーをより簡単に交換できます。

注意 5.1 以前の Orchestrator からエクスポートされた構成要素から、構成要素の属性値をインポートすることはできません。

構成要素の作成

構成要素を使用すると、Orchestrator サーバ全体に共通の属性を設定することができます。サーバで実行中のすべての要素は、構成要素で設定した属性を呼び出すことができます。構成要素を作成すると、共通の属性をサーバ内に一度で定義でき、それぞれの要素内に個別に定義する必要がありません。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [構成] ビューをクリックします。
- 3 フォルダの階層リストでフォルダを右クリックし、[新規フォルダ] を選択して、新しいフォルダを作成します。
- 4 フォルダの名前を入力し、[OK] をクリックします。
- 5 作成したフォルダを右クリックし、[新規要素] を選択します。
- 6 構成要素の名前を入力し、[OK] をクリックします。
構成要素エディタが表示されます。
- 7 [全般] タブのバージョンの数字をクリックして、バージョン番号を増分し、バージョンに関するコメントを入力します。
- 8 [全般] タブの [説明] テキスト ボックスに構成要素の説明を入力します。

- 9 [属性] タブをクリックします。
- 10 [属性の追加] アイコン (A+) をクリックして、新しい属性を作成します。
- 11 [名前]、[タイプ]、[値]、[説明] の下の属性値をクリックし、属性の名前、タイプ、値、説明を設定します。
- 12 [権限] タブをクリックします。
- 13 [アクセス権限の追加] アイコン (👤) をクリックして、この構成要素にアクセスする権限をユーザー グループに付与します。
- 14 [フィルタ] テキスト ボックスでユーザー グループを検索し、表示されたリストから関連するユーザー グループを選択します。
- 15 該当するチェック ボックスをオンにして、選択したユーザー グループにアクセス権を設定します。

構成要素には次の権限を設定できます。

権限	説明
表示	ユーザーは、構成要素を表示できますが、スキーマやスクリプトは表示できません。
確認	ユーザーは、構成要素を、スキーマやスクリプトも含めて表示することができます。
管理	ユーザーは、構成要素内の要素に権限を設定でき、その他すべての権限を持ちます。
実行	ユーザーは、構成要素内の要素を実行できます。
編集	ユーザーは、構成要素内の要素を編集できます。

- 16 [選択] をクリックします。
- 17 [保存して閉じる] をクリックして構成要素エディタを終了します。

これで、Orchestrator サーバ全体に共通の属性を設定する構成要素を定義しました。

次に進む前に

構成要素を使用して、ワークフローやアクションに属性を提供することができます。

ワークフローのユーザー権限

Orchestrator では、グループに適用してワークフローへのアクセスを許可または拒否できる権限のレベルが定義されています。

表示	ユーザーはワークフローの要素を表示できますが、スキーマやスクリプトは表示できません。
確認	ユーザーは、スキーマおよびスクリプトを含む、ワークフローの要素を表示できます。
実行	ユーザーはワークフローを実行できます。
編集	ユーザーはワークフローを編集できます。
管理	ユーザーはワークフローに権限を設定でき、他のすべての権限も有します。

管理権限には、表示権限、確認権限、編集権限、実行権限が含まれます。どの権限にも表示権限が必要です。

ワークフローに権限を設定しなかった場合は、ワークフローが含まれているフォルダから権限が継承されます。ワークフローに権限を設定すると、ワークフローが含まれているフォルダの権限の方が制約が厳しい場合でも、設定された権限によってフォルダの権限がオーバーライドされます。

ワークフローに対するユーザー権限の設定


あるワークフローに対してユーザー グループが持つことができるアクセス権を制限するために、そのワークフローに対して権限のレベルを設定します。

Orchestrator LDAP サーバから、権限を設定するユーザーおよびユーザー グループを選択できます。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- 一部の要素をワークフロー スキーマに追加します。

手順

- 1 [権限] タブをクリックします。
- 2 [アクセス権限の追加] アイコン () をクリックして、新しいユーザー グループの権限を定義します。
- 3 ユーザー グループを検索します。

検索結果には、検索に一致する Orchestrator LDAP サーバのすべてのユーザー グループが含まれます。

- 4 ユーザー グループを選択し、適切なチェック ボックスを選択して、このユーザー グループの権限のレベルを設定します。

このユーザー グループのユーザーがワークフローを表示したり、スキーマやスクリプティングを検査したり、ワークフローを実行および編集したり、権限を変更したりできるようにするには、すべてのチェック ボックスを選択する必要があります。

- 5 [選択] をクリックします。

ユーザー グループが権限リストに表示されます。

- 6 [保存して閉じる] をクリックしてエディタを終了します。

ワークフローの検証

Orchestrator には、ワークフローの検証ツールが用意されています。ワークフローを検証することで、ワークフローのエラーを特定したり、要素間のデータの流が正しいかどうか確認したりすることができます。

ワークフローの検証を実行すると、検証ツールによってエラーや警告のリストが作成されます。このリストのエラーをクリックすると、エラーが含まれるワークフロー要素が強調表示されます。

ワークフロー エディタで検証ツールを実行すると、検出されたエラーを修正するためのクイック修正アクションが提示されます。クイック修正アクションによっては、追加情報や入力パラメータが必要になるものもあります。また、そのままエラーを解決できるクイック修正アクションもあります。

ワークフローの検証では、要素間のデータのバインドと接続がチェックされます。ただし、ワークフローの各要素が実行するデータ処理はチェックされません。したがって、スキーマ要素の関数が間違っている場合は、有効なワークフローが不適切に実行され、誤った結果をもたらす可能性があります。

Orchestrator ではデフォルトで、ワークフローの実行時に必ずワークフロー検証が実行されます。Orchestrator クライアントのデフォルトの検証動作は変更可能です。[「開発時のワークフローのテスト」](#)を参照してください。たとえば、ワークフローの開発時には、テストのために、無効とわかっているワークフローを実行する必要がある場合もあります。

ワークフローの検証と検証エラーの修正

ワークフローを実行するには、事前にワークフローを検証しておく必要があります。ワークフローの検証は、Orchestrator クライアントまたはワークフロー エディタで行います。ただし、編集のためにワークフロー エディタでワークフローを開いている場合は、検証エラーの修正のみ可能です。

開始する前に

スキーマ要素がリンクされ、バインドが定義された、検証すべき完成したワークフローがあることを確認します。

手順

- 1 [ワークフロー] ビューをクリックします。
- 2 [ワークフロー] 階層リストのワークフローに移動します。
- 3 (オプション) ワークフローを右クリックし、[ワークフローの検証] を選択します。

ワークフローが有効な場合は、確認メッセージが表示されます。ワークフローが無効な場合は、エラー リストが表示されます。

- 4 (オプション) [ワークフローの検証] ダイアログ ボックスを閉じます。
- 5 ワークフローを右クリックし、[編集] を選択してワークフロー エディタを開きます。
- 6 [スキーマ] タブをクリックします。
- 7 [スキーマ] タブのツールバーにある [検証] ボタンをクリックします。

ワークフローが有効な場合は、確認メッセージが表示されます。ワークフローが無効な場合は、エラー リストが表示されます。

8 無効なワークフローのエラー メッセージをクリックします。

検証ツールによって、エラーが発生しているスキーマ要素が強調表示されます（赤色のアイコンが追加されます）。クイック修正アクションが表示される場合もあります。

- 提示されたクイック修正アクションに同意する場合は、そのアクションをクリックして実行します。
- 提示されたクイック修正アクションに同意しない場合は、[ワークフローの検証] ダイアログ ボックスを閉じ、手でスキーマ要素を修正します。

重要 提示された修正方法が適切かどうか必ず確認してください。

たとえば、未使用の属性を削除するように提示されることがありますが、実は属性が正しくバインドされていないことが原因の場合もあります。

9 検証エラーがすべてなくなるまで前述の手順を繰り返します。

ワークフローが検証され、検証エラーが修正されました。

次に進む前に

ワークフローを実行できます。

ワークフローのデバッグ

Orchestrator には、ワークフローのデバッグ ツールが用意されています。ワークフローをデバッグすることにより、特定の処理の開始時における入力パラメータと出力パラメータの内容を検証したり、編集モードでのワークフローの実行中にパラメータ値や属性値を置き換えたり、最後に失敗した処理からワークフローを再開したりすることができます。

ワークフローのデバッグは、標準のワークフロー ライブラリから実行することができます。また、カスタム ワークフローをデバッグすることもできます。ワークフロー エディタでカスタム ワークフローを開発しながら、そのワークフローをデバッグすることができます。

ワークフローのデバッグ

ワークフロー スキーマ内にブレイクポイントを追加することにより、ワークフローの各種要素をデバッグすることができます。


ブレイクポイントに達した場合は、いくつかの方法でデバッグ プロセスを継続することができます。ワークフロー スキーマから要素をデバッグする場合は、ワークフローの実行に関する一般的な情報の表示、ワークフローの変数の変更、ログ メッセージの確認を行うことができます。





開始する前に

ワークフローの実行権限を持つユーザーとして Orchestrator クライアントにログインします。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。

- 3 ワークフロー ライブラリでワークフローを選択し、[スキーマ] タブを選択します。
- 4 デバッグしたいスキーマ要素にブレークポイントを追加するには、ワークフロー要素を右クリックして [ブレークポイントの切り替え] を選択します。
ブレークポイントの有効と無効を切り替えることができます。
- 5 [ワークフローのデバッグ] アイコン () をクリックします。
ワークフローで入力パラメータが必要な場合は、そのパラメータを指定します。
- 6 ブレークポイントに到達してワークフローの実行が停止したら、以下に示すいずれかのオプションを選択します。

オプション	説明
 再開	ワークフローを再開し、次のブレークポイントに達するまでワークフローを実行します。
 ステップイン	ワークフロー要素にステップインします。 <small>注意 ワークフロー エディタでデバッグしている場合は、ネストされたワークフロー要素にステップインすることはできません。</small>
 ステップ オーバー	スキーマ内の現在の要素をステップ オーバーし、次の要素でワークフローの実行を停止します。
 ステップ リターン	ステップインしているワークフロー要素を終了します。

- 7 (オプション) [ブレークポイント] タブで、ブレークポイントを変更します。
既存のブレークポイントの有効と無効を切り替えたり、削除したりすることができます。
- 8 (オプション) [変数] タブで、変数を確認します。
デバッグ プロセスの実行中に、一部の変数の値を変更することができます。

ワークフローのデバッグ例


標準ワークフロー ライブラリ内のワークフローをデバッグすることができます。



たとえば、正しくない受信者のアドレスを指定した場合は、「メールでの対話例」ワークフローのデバッグを行うことにより、正しくない値を修正することができます。

開始する前に

「メール」ワークフローを実行できるユーザーとして Orchestrator クライアントにログインします。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ワークフローの階層リストで、[ライブラリ] - [メール] を開きます。
- 4 「メールでの対話例」ワークフローを選択して、[スキーマ] タブをクリックします。
- 5 [電子メールの送信 (対話式)] ワークフロー要素を右クリックして [ブレークポイントの切り替え] を選択します。
- 6 [ワークフローのデバッグ] アイコン () をクリックします。

- 7 必要な情報を入力します。
 - a [宛先アドレス] テキスト ボックスに、正しくない受信者のアドレスを入力します。
たとえば、**<name>@<company>.c** などのように入力します。
 - b クエリに対する応答権限を持つユーザーが属している LDAP グループを選択します。
 - c [送信] をクリックします。
 - 8 ワークフローがブレイクポイントに到達したら、[ステップイン] アイコン () をクリックします。
 - 9 [変数] タブで、変数の値を確認します。
 - 10 [toAddress] テキスト ボックスで、正しい受信者のアドレスを入力します。
たとえば、**<name>@<company>.com** などのように入力します。
 - 11 [再開] アイコン () をクリックして、ワークフローの実行を続けます。
- デバッグ プロセスでは、ユーザーが入力した値を使用してワークフローが実行されます。

実行中のワークフロー

Orchestrator ワークフローは、イベントの論理的なフローに従って実行されます。

ワークフローを実行すると、ワークフロー内の各スキーマ要素が次のシーケンスに従って実行されます。

- 1 ワークフローは、ワークフロー トークン属性と入力パラメータをスキーマ要素の入力パラメータにバインドします。
- 2 スキーマ要素が実行されます。
- 3 スキーマ要素の出力パラメータが、ワークフロー トークン属性とワークフロー出力パラメータにコピーされます。
- 4 ワークフロー トークン属性と出力パラメータがデータベース内に格納されます。
- 5 次のスキーマ要素が実行を開始します。

ワークフローの終了まで、このシーケンスがスキーマ要素ごとに繰り返されます。

ワークフロー トークンのチェック ポイント

ワークフローが実行される場合は、各スキーマ要素がチェック ポイントになります。各スキーマ要素が実行された後、Orchestrator がワークフロー トークン属性をデータベース内に格納し、次のスキーマ要素が実行を開始します。ワークフローが予期せずに停止した場合は、Orchestrator サーバの次の再起動時に、現在アクティブなスキーマ要素が再度実行され、ワークフローは中断が発生したときに実行されていたスキーマ要素の最初から続行されます。ただし、Orchestrator はトランザクション管理やロールバック機能を実装していません。

ワークフローの終了

ワークフローは、現在のアクティブなスキーマ要素が終了要素である場合に終了します。ワークフローが終了要素に達した後、他のワークフローまたはアプリケーションはそのワークフローの出力パラメータを使用できます。

ワークフロー エディタでのワークフローの実行

ワークフローは作成中に実行することができます。

ワークフロー エディタでワークフローを実行すると、作成プロセスを中断しないでワークフローが正しく実行していることを検証できます。ワークフローの実行に関する情報を示すログ メッセージを表示できます。ワークフローの実行が予期しない結果を返す場合は、ワークフローを変更し、ワークフロー エディタを閉じずに再実行できます。

開始する前に

- ワークフローを作成します。
- ワークフロー エディタで編集の対象となるワークフローを開きます。
- ワークフローを検証します。

手順

- 1 [スキーマ] タブをクリックします。
- 2 [実行] をクリックします。
- 3 (オプション) [ログ] タブでメッセージを確認します。

ワークフローの実行

標準ライブラリのワークフローまたはユーザーが作成したワークフローを実行することによって、vCenter Server で自動操作を実行できます。

たとえば、[単純な仮想マシンの作成] ワークフローを実行することによって仮想マシンを作成できます。

開始する前に

vCenter Server プラグインを構成していることを確認してください。詳細については、『VMware vCenter Orchestrator のインストールおよび構成』を参照してください。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[実行] または [設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ワークフローの階層リストで、[ライブラリ] - [vCenter] - [仮想マシンの管理] - [基本] の順に開いて [単純な仮想マシンの作成] ワークフローに移動します。
- 4 [単純な仮想マシンの作成] ワークフローを右クリックし、[ワークフローの開始] を選択します。

- 5 [ワークフローの開始] 入力パラメータ ダイアログ ボックスに次の情報を入力して、Orchestrator に接続された vCenter Server に仮想マシンを作成します。

オプション	アクション
[仮想マシン名]	仮想マシンの名前を orchestrator-test に変更します。
[仮想マシンのフォルダ]	<p>a [仮想マシンのフォルダ] 値の [未設定] をクリックします。</p> <p>b インベントリから仮想マシンのフォルダを選択します。</p> <p>[選択] ボタンは、正しいタイプのオブジェクト（この場合は VC:VmFolder）を選択するまでは無効です。</p>
[新しいディスクのサイズ (GB)]	適切な数値を入力します。
[メモリ サイズ (MB)]	適切な数値を入力します。
[仮想 CPU の数]	[仮想 CPU の数] ドロップダウン メニューから、適切な CPU の数を選択します。
[仮想マシンのゲスト OS]	[未設定] リンクをクリックし、リストからゲスト OS を選択します。
[仮想マシンを作成するホスト]	[仮想マシンを作成するホスト] 値の [未設定] をクリックし、vCenter Server インフラストラクチャ階層内をホスト マシンまで移動します。
[リソース プール]	[リソース プール] 値の [未設定] をクリックし、vCenter Server インフラストラクチャ階層内をリソース プールまで移動します。
[接続先のネットワーク]	<p>[接続先のネットワーク] 値の [未設定] をクリックし、ネットワークを選択します。</p> <p>使用可能なすべてのネットワークを表示するには、[フィルタ] テキスト ボックスで Enter キーを押します。</p>
[仮想マシン ファイルを格納するデータストア]	[仮想マシン ファイルを格納するデータストア] 値の [未設定] をクリックし、vCenter Server インフラストラクチャ階層内をデータストアまで移動します。

- 6 [送信] をクリックして、ワークフローを実行します。

[単純な仮想マシンの作成] ワークフローの下に、ワークフロー実行中アイコンを示すワークフロー トークンが表示されます。

- 7 そのワークフロー トークンをクリックして、実行中のワークフローのステータスを表示します。

- 8 ワークフロー トークン ビューで [イベント] タブをクリックして、ワークフローが完了するまでワークフロー トークンの進行状況を観察します。

- 9 [インベントリ] ビューをクリックします。

- 10 定義したリソース プールまで vCenter Server インフラストラクチャ階層内を移動します。

リストに仮想マシンが表示されない場合は、更新ボタンをクリックしてインベントリを再ロードします。

orchestrator-test 仮想マシンがリソース プール内に存在します。

- 11 (オプション)[インベントリ] ビューで **orchestrator-test** 仮想マシンを右クリックして、**orchestrator-test** 仮想マシン上で実行できるワークフローのコンテキスト リストを表示します。

[単純な仮想マシンの作成] ワークフローが正常に実行されました。

次に進む前に

vSphere Client にログインし、新しい仮想マシンを管理できます。

失敗したワークフローの実行の再開

ワークフローに失敗した場合、Orchestrator では最後に失敗したアクティビティからワークフローを再開することができます。

ワークフローのパラメータを変更して実行の再開を試みるか、パラメータを保持してワークフローの実行に影響を及ぼしている外部コンポーネントに変更を加えることができます。たとえば、サードパーティ システム内の問題が原因でワークフローの実行が失敗する場合は、そのシステムに変更を加えて、失敗したアクティビティからワークフローを再開できます。ワークフローのパラメータを変更したり、成功したアクティビティを繰り返し実行したりする必要はありません。

失敗したワークフローの実行を再開するための動作の設定

失敗した実行を再開する動作をカスタム ワークフローごとに設定することができます。ライブラリのデフォルトのワークフローは、デフォルトのシステム設定を使用して、失敗したワークフローの実行を再開します。

デフォルトのシステム動作は、構成ファイルを変更することにより変更できます。[「失敗したワークフローの実行を再開するためのカスタム プロパティの設定」](#)を参照してください。

開始する前に

ワークフローを編集するための権限があることを確認します。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ワークフローの階層リストを展開し、動作を設定するワークフローに移動します。
- 4 ワークフローを右クリックして [編集] を選択します。
ワークフロー エディタが開きます。
- 5 [全般] タブで、[失敗した動作の再開] ドロップダウン メニューからオプションを選択します。

オプション	説明
システムのデフォルト	デフォルトの動作に従います。
有効	ワークフローの実行が失敗した場合、ワークフローの実行を再開するためのオプションがポップアップウィンドウに表示されます。
無効化	ワークフローの実行が失敗した場合、再開することはできません。

- 6 [保存して閉じる] をクリックします。

失敗したワークフローの実行を再開するためのカスタム プロパティの設定

デフォルトでは、Orchestrator は失敗したワークフローの実行を再開するように設定されていません。失敗したワークフローの実行の再開を Orchestrator で有効にできます。また、カスタムのタイムアウト期間を設定し、失敗したワークフローの実行がこのタイムアウト期間後に再開できないようにすることができます。

手順

- 1 Orchestrator サーバシステムで、`/etc/vco/app-server/` に移動します。
- 2 `Vmo.properties` 構成ファイルをテキスト エディタで開きます。
- 3 `vmo.properties` ファイルの以下の行を編集し、失敗したワークフローの実行が Orchestrator で再開されるように設定します。

```
com.vmware.vco.engine.execute.resume-from-failed=true
```

- 4 `vmo.properties` ファイルの以下の行を編集し、失敗したワークフローの実行を再開するまでのカスタムのタイムアウト期間を設定します。

```
com.vmware.vco.engine.execute.resume-from-failed.timeout-sec=<<seconds>>
```

設定した値により、デフォルトのタイムアウト設定である 86,400 秒はオーバーライドされます。

- 5 `vmo.properties` ファイルを保存します。
- 6 Orchestrator サーバを再起動します。

失敗したワークフローの実行の再開

失敗したワークフローの実行の再開が有効になっている場合、最後に失敗したアクティビティからワークフローの実行を再開することができます。

失敗したワークフローの実行を再開するオプションが有効になっている場合、ワークフローのパラメータを変更し、ワークフローが失敗した後に表示されるポップアップ ウィンドウのオプションを使用して、実行の再開を試みることができます。また、パラメータを保持してワークフローの実行に影響を及ぼしている外部コンポーネントに変更を加えることもできます。オプションを選択しない場合、ワークフローの実行はタイムアウトして再開できなくなります。タイムアウト期間の変更については、[「失敗したワークフローの実行を再開するためのカスタム プロパティの設定」](#)を参照してください。

手順

- 1 ポップアップ ウィンドウのドロップダウン メニューから、[再開] を選択し、[次へ] をクリックします。
[キャンセル] を選択すると、ワークフローの実行を後で再開できなくなります。
- 2 (オプション) ワークフローのパラメータを変更します。
- 3 [送信] をクリックします。

ワークフロー ドキュメントの生成

ワークフローまたはワークフロー フォルダに関する文書は、いつでも選択して PDF 形式でエクスポートできます。

エクスポートしたドキュメントには、選択したワークフローまたはフォルダ内のワークフローに関する詳細情報が記載されています。各ワークフローに関する情報には、ワークフローの名前、バージョン履歴、属性、パラメータ表示、ワークフローのスキーマ、ワークフローのアクションなどが含まれます。ドキュメントにはさらに、使用されたアクションのソース コードも含まれています。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[実行] または [設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ドキュメントを生成するワークフローまたはワークフロー フォルダのある場所に移動し、右クリックします。
- 4 [ドキュメントの生成] を選択します。
- 5 PDF ファイルを保存するフォルダを参照して指定し、ファイル名を入力し、[保存] をクリックします。

選択したワークフローまたはフォルダ内のワークフローに関する情報を含む PDF ファイルが、システムに保存されます。

ワークフロー バージョン履歴の使用

バージョン履歴を使用すると、ワークフローを以前に保存したときの状態に戻すことができます。ワークフローの状態は、ワークフロー バージョンより前のバージョンに戻すことも、より新しいバージョンに戻すこともできます。現在の状態のワークフローと保存されているバージョンのワークフローの違いを比較することもできます。

ワークフロー バージョンを増やして保存すると、Orchestrator で各ワークフローの新しいバージョン履歴項目が作成されます。これ以降にワークフローに変更を加えても、現在保存されているバージョンは変更されません。たとえば、ワークフロー バージョン 1.0.0 を作成して保存すると、ワークフローの状態がバージョン履歴に保存されます。ワークフローに変更を加えると、ワークフローの状態を Orchestrator クライアントに保存することはできませんが、変更をワークフロー バージョン 1.0.0 に適用することはできません。バージョン履歴に変更を保存するには、ワークフロー バージョンを増やして保存する必要があります。バージョン履歴はワークフロー自体とともにデータベース内に保持されます。

ワークフローを削除すると、Orchestrator でデータベース内の要素が削除済みとしてマークされますが、要素のバージョン履歴はデータベースから削除されません。この方法により、削除したワークフローをリストアすることができます。[「削除済みワークフローのリストア」](#)を参照してください。

開始する前に

ワークフロー エディタで編集対象のワークフローを開きます。

手順

- 1 ワークフロー エディタの [全般] タブをクリックし、[バージョン履歴の表示] をクリックします。
- 2 ワークフロー バージョンを選択し、[現在のバージョンとの差分] をクリックして違いを比較します。
現在のワークフロー バージョンと選択したワークフロー バージョンの違いがウィンドウに表示されます。
- 3 ワークフロー バージョンを選択し、[元に戻す] をクリックして、ワークフローの状態をリストアします。

注意 現在のワークフロー バージョンを保存しなかった場合はバージョン履歴から削除されるため、現在のバージョンに戻せなくなります。

ワークフローの状態は選択したバージョンの状態に戻ります。

削除済みワークフローのリストア

ワークフロー ライブラリから削除したワークフローはリストアすることができます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[実行] または [設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 削除済みワークフローをリストアする先のワークフロー フォルダに移動します。
- 4 フォルダを右クリックし、[削除済みワークフローのリストア] を選択します。
- 5 リストアするワークフローを 1 つ以上選択し、[リストア] をクリックします。

リストアされたワークフローが選択したフォルダに表示されます。

単純なサンプル ワークフローの開発

単純なサンプル ワークフローの開発では、ワークフロー開発プロセスで最も一般的な手順を示します。

作成するサンプル ワークフローでは、vCenter Server 内の既存の仮想マシンが開始され、仮想マシンが開始されたことを確認するための E メールが管理者に送信されます。

このサンプル ワークフローは次のタスクを実行します。

- 1 開始する仮想マシンを選択するようユーザーに求めます。
- 2 通知の送信先 E メール アドレスを入力するようユーザーに求めます。
- 3 選択された仮想マシンがすでにパワーオン状態になっているかどうかを確認します。
- 4 仮想マシンを開始する要求を vCenter Server インスタンスに送信します。
- 5 vCenter Server が仮想マシンを開始するのを待機し、仮想マシンが開始しなかったり仮想マシンの開始に時間がかかりすぎている場合はエラーを返します。
- 6 vCenter Server が仮想マシン上の VMware Tools を開始するのを待機し、仮想マシンが開始しなかったり VMware Tools の開始に時間がかかりすぎている場合はエラーを返します。
- 7 仮想マシンが実行中であることを確認します。
- 8 マシンが開始されたかエラーが発生したことを知らせる通知を、指定された E メール アドレスに送信します。

Orchestrator ドキュメントの Web ページからダウンロードできる Orchestrator のサンプルの ZIP ファイルには、「仮想マシンの開始と Eメールの送信」ワークフローの完全なバージョンが含まれています。

サンプル ワークフローを開発するプロセスは、いくつかのタスクで構成されています。

開始する前に

単純なサンプル ワークフローを開発する前に、「[ワークフローの主要な概念](#)」をお読みください。

手順

1 単純なワークフロー サンプルの作成

ワークフロー開発プロセスは、クライアントでワークフローを作成することから始める必要があります。

2 単純なワークフロー サンプルのスキーマの作成

ワークフロー エディタでワークフローのスキーマを作成できます。ワークフローのスキーマにはワークフローで実行する要素が含まれ、ワークフローの論理フローを決定します。

3 (オプション) 単純なワークフロー サンプルのゾーンの作成

さまざまな色のワークフロー メモを追加することによって、ワークフローのさまざまなゾーンを強調できます。さまざまなワークフロー ゾーンを作成することで、込み入ったワークフローのスキーマを読解しやすくなります。

4 単純なワークフロー サンプルのパラメータの定義

ワークフロー開発のこの段階では、ワークフローの実行に必要な入力パラメータを定義します。ワークフロー サンプルでは、仮想マシンをパワーオン状態にするための入力パラメータ、操作結果を通知する相手のメールアドレスのパラメータが必要です。ユーザーがワークフローを実行するときは、パワーオン状態にする仮想マシンおよびメール アドレスが要求されます。

5 単純なワークフロー サンプルの決定バインドの定義

ワークフロー エディタの [スキーマ] タブで、ワークフローの要素をバインドすることができます。決定バインドは、決定要素が受信した入力パラメータと決定ステートメントとを比較し、入力パラメータが決定ステートメントに一致するかどうかに応じて出力パラメータを生成します。

6 単純なワークフローでのアクション要素のバインド例

ワークフロー エディタでワークフローの要素をバインドすることができます。バインドは、アクション要素による入力パラメータの処理方法と、出力パラメータの生成方法を定義します。

7 単純なワークフローでのスクリプト化されたタスク要素のバインド例

ワークフロー エディタの [スキーマ] タブで、ワークフローの要素をバインドすることができます。バインドは、スクリプト化されたタスク要素による入力パラメータの処理方法と、出力パラメータの生成方法を定義します。スクリプト化可能タスク要素をその JavaScript 関数にバインドすることもできます。

8 単純なワークフロー サンプルの例外バインドの定義

ワークフロー エディタの [スキーマ] タブで例外バインドを定義します。例外バインドは、要素がエラーを処理する方法を定義します。

9 単純なワークフロー サンプルの属性の読み書きプロパティの設定

パラメータや属性が読み取り専用の定数または書き込み可能な変数のどちらであるかを定義できます。また、ユーザーが入力パラメータに指定できる値に制限を設定することもできます。

10 単純なワークフロー サンプルのパラメータのプロパティの設定

ワークフロー エディタでパラメータのプロパティを設定できます。パラメータのプロパティを設定すると、そのパラメータの動作に影響を与え、またそのパラメータに指定できる値に制約が設定されます。

11 単純なワークフロー サンプルの入力パラメータ ダイアログ ボックスのレイアウトの設定

ワークフロー エディタで、入力パラメータ ダイアログ ボックスのレイアウトまたはプレゼンテーションを作成します。この入力パラメータ ダイアログ ボックスは、ユーザーが、実行のために入力パラメータが必要なワークフローを実行するときに開きます。

12 単純なワークフロー サンプルの検証と実行

ワークフローを作成したら、検証を行って、予想されるエラーを検出することができます。エラーがなければ、そのワークフローを実行できます。

単純なワークフロー サンプルの作成

ワークフロー開発プロセスは、クライアントでワークフローを作成することから始める必要があります。

開始する前に

次のコンポーネントがシステムにインストールおよび設定されていることを確認します。

- 少なくとも 1 つがパワーオフされている仮想マシンを制御する vCenter Server
- SMTP サーバへのアクセス
- 有効なメール アドレス

vCenter Server をインストールおよび設定する方法については、『vSphere のインストールとセットアップ』を参照してください。SMTP サーバを使用するように Orchestrator を設定する方法については、『VMware vRealize Orchestrator のインストールと構成』を参照してください。

ワークフローを記述するには、サーバまたは操作しているワークフロー フォルダに少なくとも**表示権限**、**実行権限**、**確認権限**、**編集権限**、およびできれば**管理権限**を持つ Orchestrator ユーザー アカウントが必要です。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 ワークフロー リストのルートを右クリックして、[フォルダの追加] を選択します。
- 4 新しいフォルダに **ワークフロー サンプル** という名前を付け、[OK] をクリックします。
- 5 [ワークフロー サンプル] フォルダを右クリックし、[新しいワークフロー] を選択します。
- 6 新しいワークフローに **仮想マシンの開始と E メール送信** という名前を付け、[OK] をクリックします。

ワークフロー エディタが開きます。

- 7 [全般] タブで、バージョン番号の数値をクリックしてバージョン番号を増やします。
これはワークフローの初期作成状態であるため、バージョンを [0.0.1] に設定します。
- 8 [全般] タブの [サーバ再起動時の動作] 値をクリックし、サーバが再起動した後でワークフローを再開するかどうかを設定します。
- 9 [全般] タブの [説明] テキスト ボックスで、ワークフローの処理の説明を入力します。
たとえば、次の説明を追加できます。

このワークフローは仮想マシンを開始し、確認メールを Orchestrator 管理者に送信します。

- 10 [全般] タブの一番下にある [保存] をクリックします。

「仮想マシンの開始とメール送信」というワークフローを作成しましたが、その機能は定義していません。

次に進む前に

ワークフローのスキーマを作成します。

単純なワークフロー サンプルのスキーマの作成

ワークフロー エディタでワークフローのスキーマを作成できます。ワークフローのスキーマにはワークフローで実行する要素が含まれ、ワークフローの論理フローを決定します。

開始する前に

次の操作を行います。

- [「単純なワークフロー サンプルの作成」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 [汎用] メニューから、スキーマ内の開始要素と **End** 要素とをリンクする矢印に対して決定要素をドラッグします。
- 3 決定要素をダブルクリックし、名前を **仮想マシンをパワーオンするか**に変更します。
決定要素は、仮想マシンがすでにパワーオン状態であるかどうかを確認するブール関数に対応します。
- 4 [汎用] メニューから、決定要素と **End** 要素とをリンクする赤色の矢印に対してアクション要素をドラッグします。
アクションを選択するためのダイアログ ボックスが表示されます。
- 5 [フィルタ] テキスト ボックスに **start** と入力し、フィルタされたアクションのリストから **[startVM]** アクションを選択し、[選択] をクリックします。
- 6 **startVM** アクション要素と **End** 要素とをリンクする青色の矢印に対して次のアクション要素を 1 つずつドラッグします。

vim3WaitTaskEnd

ワークフローの実行をサスペンドし、実行中の vCenter Server タスクが完了するまで定期的に ping を実行します。**startVM** アクションは仮想マシンを開始し、**vim3WaitTaskEnd** アクションは仮想マシンが開始中の間ワークフローを待機させます。仮想マシンが開始すると、**vim3WaitTaskEnd** によってワークフローが再開します。

vim3WaitToolsStarted

ワークフローの実行をサスペンドし、VMware Tools がターゲット仮想マシンで開始するまで待機します。

- 7 [汎用] メニューから、**vim3WaitToolsStarted** アクション要素と **End** 要素とをリンクする青色の矢印に対してスクリプト化可能タスク要素をドラッグします。
- 8 スクリプト化可能タスク要素をダブルクリックし、名前を **OK** に変更します。
- 9 **VM powered on?** 決定要素と **End** 要素とをリンクする緑色の矢印に対して別のスクリプト化可能タスク要素をドラッグし、このスクリプト化可能タスク要素に**すでに開始済み**という名前を付けます。

10 **Already started** スクリプト化可能タスク要素のリンクを変更します。

- a **Already started** スクリプト化可能タスク要素を **startVM** アクション要素の左にドラッグします。
- b **Already started** スクリプト化可能タスク要素と **End** 要素を接続する青色の矢印を削除します。
- c **Already started** スクリプト化可能タスク要素と **vim3WaitToolsStarted** アクション要素とを青色の矢印でリンクします。

11 [汎用] メニューから、次のスクリプト化可能タスク要素をスキーマにドラッグします。

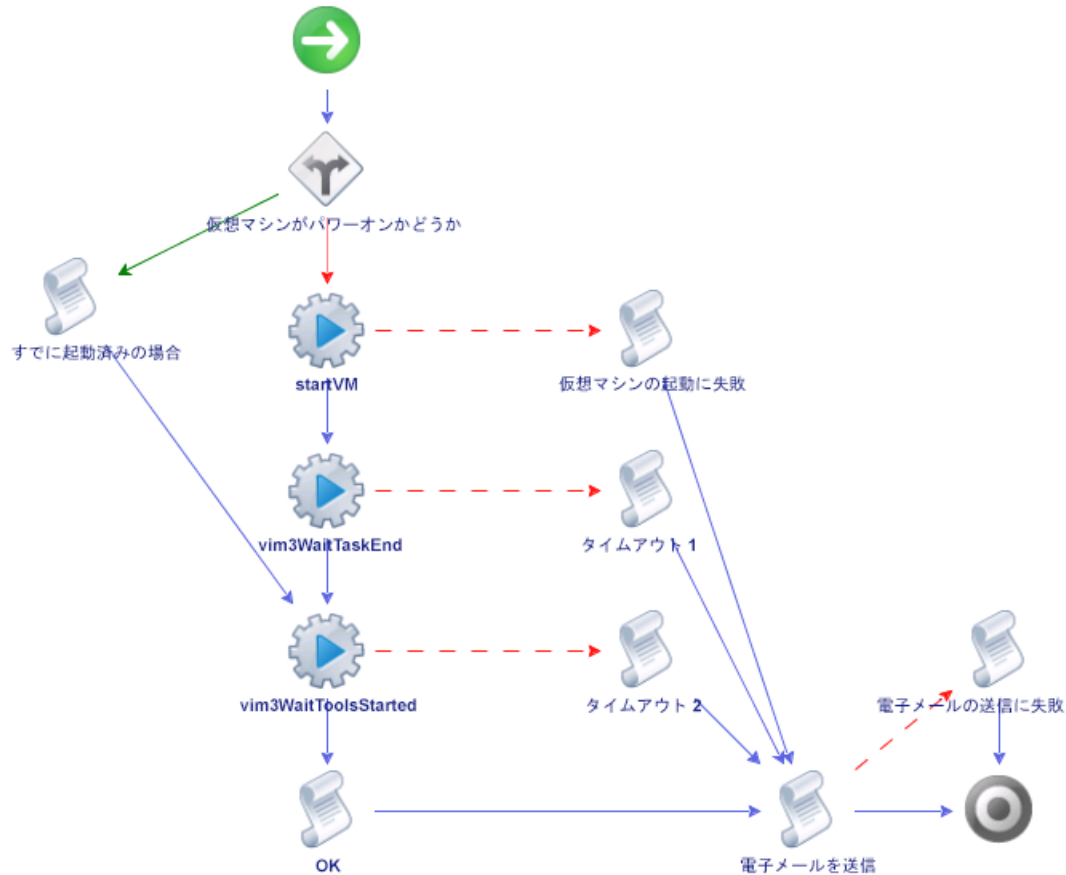
- スクリプト化可能タスク要素を **startVM** アクション要素にドラッグし、スクリプト化可能タスク要素に **仮想マシンの開始失敗** という名前を付けます。
- スクリプト化可能タスク要素を **vim3WaitTaskEnd** アクション要素にドラッグし、スクリプト化可能タスク要素に **タイムアウト 1** という名前を付けます。
- スクリプト化可能タスク要素を **vim3WaitToolsStarted** アクション要素にドラッグし、スクリプト化可能タスク要素に **タイムアウト 2** という名前を付けます。
- **OK** スクリプト化可能タスク要素と **End** 要素とをリンクする青色の矢印にスクリプト化可能タスク要素をドラッグします。新しいスクリプト化可能タスク要素に **E メール送信** という名前を付け、**OK** スクリプト化可能タスク要素の右にドラッグします。
- **Start VM Failed**、**Timeout 1**、および **Timeout 2** スクリプト化可能タスク要素と **Send Email** スクリプト化可能タスク要素とを青色の矢印でリンクします。
- スクリプト化可能タスク要素を **Send Email** スクリプト化可能タスク要素にドラッグします。新しいスクリプト化可能タスク要素に **E メール送信失敗** という名前を付け、**Timeout 2** スクリプト化可能タスク要素の右にドラッグし、**End** 要素と青色の矢印でリンクします。

12 **End** 要素を **Send Email** スクリプト化可能タスク要素の右にドラッグします。

13 [スキーマ] タブの一番下にある [保存] をクリックします。

次の図に、「仮想マシンの開始と E メール送信」ワークフロー スキーマ要素のレイアウトを示します。

図 1-10. 「仮想マシンの開始と E メール送信」 ワークフロー サンプルの要素のリンク



次に進む前に

ワークフローのさまざまなゾーンをハイライトできます。

(オプション) 単純なワークフロー サンプルのゾーンの作成

さまざまな色のワークフロー メモを追加することによって、ワークフローのさまざまなゾーンを強調できます。さまざまなワークフロー ゾーンを作成することで、込み入ったワークフローのスキーマを読解しやすくなります。

開始する前に

次の操作を行います。

- 「単純なワークフロー サンプルの作成」。
- 「単純なワークフロー サンプルのスキーマの作成」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

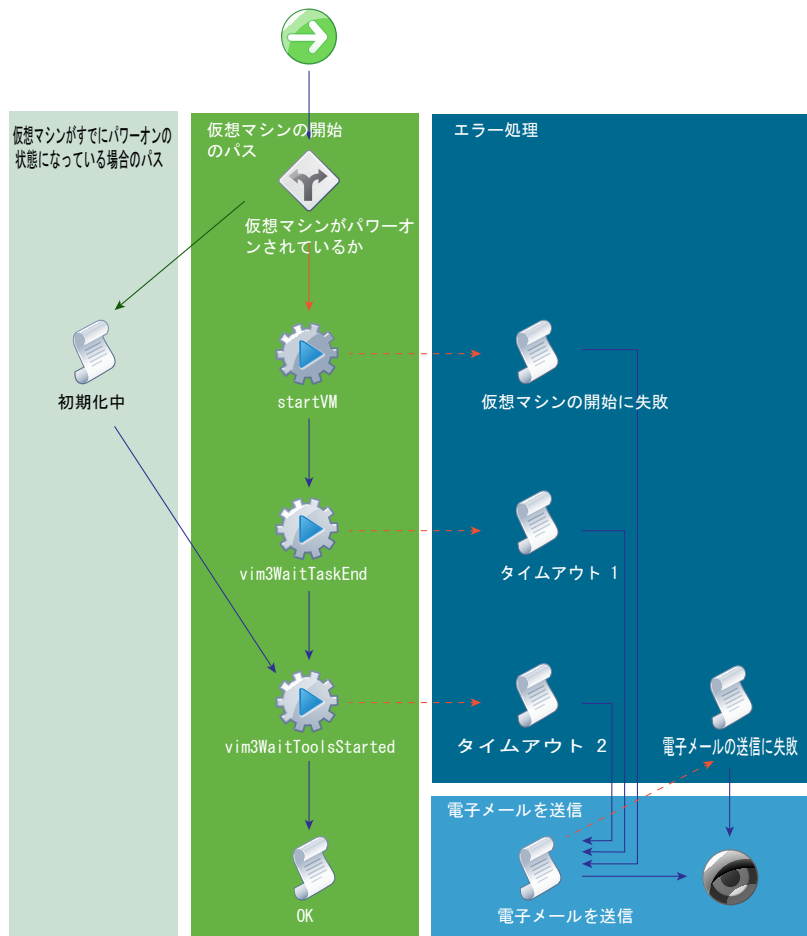
手順

- 1 ワークフロー メモ要素を [汎用] メニューからワークフローエディタにドラッグします。
- 2 ワークフロー メモを **Already started** スクリプト化可能タスク要素の上に配置します。

- 3 ワークフロー メモの端をドラッグして、**Already started** スクリプト化可能タスク要素を囲むようにサイズを変更します。
- 4 テキストをダブルクリックし、説明を追加します。
たとえば、**仮想マシンがすでにパワーオン状態になっている場合のパス** にします。
- 5 Ctrl + E キーを押して背景色を選択します。
- 6 前述の手順を繰り返して、ワークフローのその他のゾーンをハイライトします。
 - **VM powered on?** 決定要素から **OK** 要素まで縦に並んだ一連の要素の上にメモを配置します。**仮想マシンの開始パス** という説明を追加します。
 - **startVM failed**、両方の **Timeout** スクリプト化可能タスク要素、および **Send Email Failed** スクリプト化可能タスク要素の上にメモを配置します。**エラー処理** という説明を追加します。
 - **Send Email** スクリプト化可能タスク要素の上にメモを配置します。**メールの送信** という説明を追加します。

次の図に、ワークフロー サンプルのゾーンの様子を示します。

図 1-11. 「仮想マシンの開始とメールの送信」ワークフロー サンプルのゾーン



次に進む前に

ワークフローの属性と、入力および出力パラメータを定義する必要があります。

単純なワークフロー サンプルのパラメータの定義

ワークフロー開発のこの段階では、ワークフローの実行に必要な入力パラメータを定義します。ワークフロー サンプルでは、仮想マシンをパワーオン状態にするための入力パラメータ、操作結果を通知する相手のメール アドレスのパラメータが必要です。ユーザーがワークフローを実行するときは、パワーオン状態にする仮想マシンおよびメール アドレスが要求されます。

開始する前に

次の操作を行います。

- 「[単純なワークフロー サンプルの作成](#)」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [入力] タブをクリックします。
- 2 [入力] タブの内側を右クリックして、[パラメータの追加] を選択します。
`arg_in_0` という名前のパラメータが [入力] タブに表示されます。
- 3 [`arg_in_0`] をクリックします。
- 4 [属性名の選択] ダイアログ ボックスに名前 **vm** を入力し、[OK] をクリックします。
- 5 [タイプ] テキスト ボックスをクリックし、パラメータ タイプ ダイアログ ボックスの検索テキスト ボックスに **vc:virtualm** と入力します。
- 6 表示されたパラメータ タイプのリストから [**VC:VirtualMachine**] を選択し、[受け入れる] をクリックします。
- 7 [説明] テキスト ボックスにパラメータの説明を追加します。
たとえば、**パワーオン状態にする仮想マシン** と入力します。
- 8 [手順 2](#) から [手順 7](#) を繰り返して、2 つ目の入力パラメータを次の値で作成します。
 - 名前: **toAddress**
 - タイプ: 文字列
 - 説明: **このワークフローの結果を送信するメール アドレス**
- 9 [入力] タブの一番下にある [保存] をクリックします。

ワークフローの入力パラメータを定義しました。

次に進む前に

要素パラメータ間のバインドを定義します。

単純なワークフロー サンプルの決定バインドの定義

ワークフロー エディタの [スキーマ] タブで、ワークフローの要素をバインドすることができます。決定バインドは、決定要素が受信した入力パラメータと決定ステートメントとを比較し、入力パラメータが決定ステートメントに一致するかどうかに応じて出力パラメータを生成します。

開始する前に

次の操作を行います。

- [「単純なワークフロー サンプルの作成」](#)。
- [「単純なワークフロー サンプルのスキーマの作成」](#)。
- [「単純なワークフロー サンプルのパラメータの定義」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 [スキーマ] タブで、[VM Powered On?] 決定要素の [編集] アイコン (✎) をクリックします。
- 2 [決定] タブで、[未設定 (NULL)] ボタンをクリックし、表示されたパラメータのリストから **vm** を決定要素の入力パラメータとして選択します。
- 3 ドロップダウン メニューに表示された決定ステートメントのリストから [電源状態が次に等しい] ステートメントを選択します。

[未設定] ボタンが値テキスト ボックスに表示されます。このボタンを使用すると、使用可能な値のみが選択肢として表示されます。
- 4 **[poweredOn]** を選択します。
- 5 ワークフロー エディタの [スキーマ] タブの一番下にある [保存] をクリックします。

決定要素が受信した入力パラメータの値と比較する **true** または **false** ステートメントを定義しました。

次に進む前に

ワークフローのその他の要素についてバインドを定義する必要があります。

単純なワークフローでのアクション要素のバインド例

ワークフロー エディタでワークフローの要素をバインドすることができます。バインドは、アクション要素による入力パラメータの処理方法と、出力パラメータの生成方法を定義します。

開始する前に

次の操作を行います。

- [「単純なワークフロー サンプルの作成」](#)。
- [「単純なワークフロー サンプルのスキーマの作成」](#)。

- 「単純なワークフロー サンプルのパラメータの定義」。
- 「単純なワークフロー サンプルの決定バインドの定義」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 [スキーマ] タブで、**startVM** アクション要素の [編集] アイコン (✎) をクリックします。
- 2 [情報] タブで、次の一般的な情報を設定します。

オプション	アクション
相互作用	[外部操作なし] を選択します。
ビジネス ステータス	チェックボックスをオンにして、「 start VM の送信 」というテキストを追加します。
説明	「仮想マシンの起動/再開。起動タスクを返す」というテキストを残します。

- 3 [入力] タブをクリックします。

[入力] タブには、**startVM** アクションで利用できる 2 つの入力パラメータである **vm** と **host** が表示されます。

Orchestrator は、**vm** パラメータを **vm[in-parameter]** に自動的にバインドします。**startVM** アクションは、**VC:VirtualMachine** のみを入力パラメータとして取得することができるからです。Orchestrator は、ワークフローの入力パラメータを設定したときに定義した **vm** パラメータを検出し、アクションに自動的にバインドします。

- 4 **host** を [NULL] に設定します。

これはオプションのパラメータであるため、NULL に設定することができます。ただし、[未設定] に設定されたままにすると、ワークフローを検証することができません。

- 5 [出力] タブをクリックします。

すべてのアクションが生成するデフォルトの出力パラメータ、**actionResult** が表示されます。

- 6 **actionResult** パラメータで、[未設定] をクリックします。

- 7 [ワークフローでパラメータ/属性を作成] をクリックします。

[パラメータ情報] ダイアログ ボックスに、この出力パラメータに設定できる値が表示されます。**startVM** アクションの出力パラメータのタイプは、**VC:Task** オブジェクトです。

- 8 パラメータに「**powerOnTask**」という名前を付け、説明を入力します。

たとえば、「**仮想マシンの電源投入結果を含む**」と入力します。

- 9 [同じ名前で作成ワークフロー属性を作成] をクリックし、[OK] をクリックして、[パラメータ情報] ダイアログ ボックスを終了します。

- 10 前述の手順を繰り返して、入力パラメータと出力パラメータを **vim3WaitTaskEnd** および **vim3WaitToolsStarted** アクション要素にバインドします。

「[単純なワークフロー サンプルのアクション要素のバインド](#)」には、**vim3WaitTaskEnd** および **vim3WaitToolsStarted** アクション要素のバインドを一覧表示しています。

- 11 ワークフロー エディタの [スキーマ] タブの一番下にある [保存] をクリックします。

アクション要素の入力および出力パラメータが、適切なパラメータのタイプと値にバインドされました。

次に進む前に

スクリプト化可能なタスク要素をバインドし、その機能を定義します。

単純なワークフロー サンプルのアクション要素のバインド

バインドは、単純なワークフロー サンプルのアクション要素が入力および出力パラメータを処理する方法を定義します。

バインドを定義する場合、Orchestrator には、ワークフローですでに定義されているパラメータがバインドの候補として表示されます。ワークフローに必要なパラメータがまだ定義されていない場合、唯一のパラメータ選択は **NULL** です。[ワークフローでパラメータ/属性を作成] をクリックして、新しいパラメータを作成します。

vim3WaitTaskEnd アクション

vim3WaitTaskEnd アクション要素は、タスクの進行状況を追跡するための定数とポーリング レートを宣言します。次の表は、**vim3WaitTaskEnd** アクションに必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-53. vim3WaitTaskEnd アクションのバインド値

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
task	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: powerOnTask ソース パラメータ: task[attribute] タイプ: VC:Task 説明: 仮想マシンのパワーオンの結果を含む。
progress	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: progress ソース パラメータ: progress[attribute] タイプ: ブール値 値: いいえ (false) 説明: vCenter Server タスクの完了を待機している間の進行状況のログ記録。

表 1-53. vim3WaitTaskEnd アクションのバインド値 (続き)

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
pollRate	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ : pollRate ソース パラメータ : pollRate[attribute] タイプ : 数値 値: 2 説明: vim3WaitTaskEnd が vCenter Server タスクの進捗をチェックするポーリング レート(秒単位)。
actionResult	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ : actionResult[attribute] ソース パラメータ : returnedManagedObject[attribute] タイプ : 任意 説明: waitTaskEnd アクションから返された管理対象オブジェクト。

vim3WaitToolsStarted アクション

vim3WaitToolsStarted アクション要素は、VMware Tools が仮想マシンにインストールされるまで待機し、ポーリング レートとタイムアウト期間を定義します。次の表は、**vim3WaitToolsStarted** アクションに必要な入力パラメータのバインドを示しています。

vim3WaitToolsStarted アクション要素には出力がないため、出力のバインドは必要ありません。

表 1-54. vim3WaitToolsStarted アクションのバインド値

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	自動バインド	<ul style="list-style-type: none"> ローカル パラメータ : vm ソース パラメータ : vm[in-parameter] タイプ : VC:VirtualMachine 値: 編集できず、変数はワークフロー属性ではありません。 説明: 起動する仮想マシン。
pollingRate	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ : pollRate ソース パラメータ : pollRate[attribute] タイプ : 数値 説明: vim3WaitTaskEnd が vCenter Server タスクの進捗をチェックするポーリング レート(秒単位)。
timeout	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ : timeout ソース パラメータ : timeout[attribute] タイプ : 数値 値: 10 説明: vim3WaitToolsStarted が例外をスローする前に待つタイムアウト制限。

単純なワークフローでのスクリプト化されたタスク要素のバインド例

ワークフロー エディタの [スキーマ] タブで、ワークフローの要素をバインドすることができます。バインドは、スクリプト化されたタスク要素による入力パラメータの処理方法と、出力パラメータの生成方法を定義します。スクリプト化可能タスク要素をその JavaScript 関数にバインドすることもできます。

開始する前に

次の操作を行います。

- 「[単純なワークフロー サンプルの作成](#)」。
- 「[単純なワークフロー サンプルのスキーマの作成](#)」。
- 「[単純なワークフロー サンプルのパラメータの定義](#)」。
- 「[単純なワークフロー サンプルの決定バインドの定義](#)」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 [スキーマ] タブで、**Already Started** スクリプト化可能タスク要素の [編集] アイコン (✎) をクリックします。
- 2 [情報] タブに、次の一般的な情報を設定します。

オプション	アクション
相互作用	[外部操作なし] を選択します。
ビジネス ステータス	チェックボックスをオンにして、「 電源投入済みの仮想マシン 」というテキストを追加します。
説明	「仮想マシンは電源投入済み。startVM と waitTaskEnd を迂回。仮想マシンの各ツールが起動し、実行されていることを確認」というテキストを残します。

- 3 [入力] タブをクリックします。
これは、カスタムのスクリプト化可能タスク要素であるため、あらかじめ定義されているプロパティはありません。
- 4 [ワークフローのパラメータ/属性にバインド] アイコン (🔗) をクリックします。
- 5 提示されたパラメータのリストから **vm** を選択します。
- 6 [出力] タブと [例外] タブを空欄のままにします。
この要素は出力パラメータや例外を生成しません。
- 7 [スクリプティング] タブをクリックします。
- 8 次の JavaScript 関数を追加します。

```
//Writes the following event in the Orchestrator database
Server.log("VM '"+ vm.name +"' already started");
```

- 9 前述の手順を繰り返して、残りの入力パラメータをその他のスクリプト化可能タスク要素にバインドします。
「[単純なワークフロー サンプルのスクリプト化可能タスク要素のバインド](#)」には、**Start VM failed** (Timeout と Error の両方)、**Send Email Failed**、および **OK** スクリプト化可能タスク要素のバインドを一覧表示しています。
- 10 ワークフロー エディタの [スキーマ] タブの一番下にある [保存] をクリックします。

これで、スクリプト化可能タスク要素をその入力パラメータと出力パラメータにバインドし、その機能を定義するスクリプトを入力しました。

次に進む前に

例外処理を定義する必要があります。

単純なワークフロー サンプルのスクリプト化可能タスク要素のバインド

バインドは、単純なワークフロー サンプルのスクリプト化可能タスク要素が入力パラメータを処理する方法を定義します。スクリプト化可能タスク要素をその JavaScript 関数にバインドすることもできます。

バインドを定義する場合、Orchestrator には、ワークフローですでに定義されているパラメータがバインドの候補として表示されます。ワークフローで必要なパラメータがまだ定義されていない場合、唯一のパラメータ選択は **NULL** です。[ワークフローでパラメータ/属性を作成] をクリックして、新しいパラメータを作成します。

「仮想マシンの開始に失敗」スクリプト化可能タスク

「仮想マシンの開始に失敗」スクリプト化可能タスクは、仮想マシンの開始の失敗に関する電子メール通知の内容を設定し、そのイベントを Orchestrator ログに書き込むことによって、**startVM** アクションがスローする例外をすべて処理します。

次の表は、「仮想マシンの開始に失敗」スクリプト化可能タスク要素に必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-55. 「仮想マシンの開始に失敗」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[in-parameter] タイプ: VC:VirtualMachine 説明: パワーオンする仮想マシン。
errorCode	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: errorCode ソース パラメータ: errorCode[attribute] タイプ: 文字列 説明: 仮想マシンのパワーオン中のすべての例外のキャッチ。
body	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: body ソース パラメータ: body[attribute] タイプ: 文字列 説明: メールの本文

「仮想マシンの開始に失敗」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
body = "Unable to execute powerOnVM_Task() on VM '"+vm.name+"', exception found: "+errorCode;
//Writes the following event in the Orchestrator database
Server.error("Unable to execute powerOnVM_Task() on VM '"+vm.name+"', exception found:
"+errorCode);
```

「タイムアウト 1」スクリプト化可能タスク要素

「タイムアウト 1」スクリプト化可能タスクは、タスクの失敗に関する電子メール通知の内容を設定し、そのイベントを Orchestrator ログに書き込むことによって、**vim3WaitTaskEnd** アクションがスローする例外をすべて処理します。

次の表は、「タイムアウト 1」スクリプト化可能タスク要素に必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-56. 「タイムアウト 1」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[in-parameter] タイプ: VC:VirtualMachine 説明: 起動する仮想マシン。
errorCode	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: errorCode ソース パラメータ: errorCode[attribute] タイプ: 文字列 説明: 仮想マシンのパワーオン中のすべての例外のキャッチ。
body	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ: body ソース パラメータ: body[attribute] タイプ: 文字列 説明: メールの本文

「タイムアウト 1」スクリプト化可能タスク要素には、次のスクリプト関数が必要です。

```
body = "Error while waiting for poweredOnVM_Task() to complete on VM '"+vm.name+"', exception
found: "+errorCode;
//Writes the following event in the Orchestrator database
Server.error("Error while waiting for poweredOnVM_Task() to complete on VM '"+vm.name+"',
exception found: "+errorCode);
```

「タイムアウト 2」スクリプト化可能タスク要素

「タイムアウト 2」スクリプト化可能タスク要素は、タスクの失敗に関する電子メール通知の内容を設定し、そのイベントを Orchestrator ログに書き込むことによって、**vim3WaitToolsStarted** アクションがスローする例外をすべて処理します。

次の表は、「タイムアウト 2」スクリプト化可能タスク要素に必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-57. 「タイムアウト 2」 スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[in-parameter] タイプ: VC:VirtualMachine 説明: パワーオンする仮想マシン。
errorCode	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: errorCode ソース パラメータ: errorCode[attribute] タイプ: 文字列 説明: 仮想マシンのパワーオン中のすべての例外のキャッチ。
body	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ: body ソース パラメータ: body[attribute] タイプ: 文字列 説明: メールの本文

「タイムアウト 2」 スクリプト化可能タスク要素には、次のスクリプト関数が必要です。

```
body = "Error while waiting for VMware tools to be up on VM '"+vm.name+"', exception found: "+errorCode;
//Writes the following event in the Orchestrator database
Server.error("Error while waiting for VMware tools to be up on VM '"+vm.name+"', exception found: "+errorCode);
```

「OK」 スクリプト化可能タスク要素

「OK」 スクリプト化可能タスク要素は、仮想マシンが正常に開始されたという通知を受信し、仮想マシンの正常な開始に関する電子メール通知の内容を設定して、そのイベントを Orchestrator ログに書き込みます。

次の表は、「OK」 スクリプト化可能タスク要素に必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-58. 「OK」 スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ : vm ソース パラメータ : vm[in-parameter] タイプ : VC:VirtualMachine 説明: パワーオンする仮想マシン。
body	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ : body ソース パラメータ : body[attribute] タイプ : 文字列 説明: メールの本文

「OK」 スクリプト化可能タスク要素には、次のスクリプト関数が必要です。

```
body = "The VM '"+vm.name+"' has started successfully and is ready for use";
//Writes the following event in the Orchestrator database
Server.log(body);
```

「電子メールの送信に失敗」 スクリプト化可能タスク要素

「電子メールの送信に失敗」 スクリプト化可能タスク要素は、電子メールの送信に失敗したという通知を受信し、そのイベントを Orchestrator ログに書き込みます。

次の表は、「電子メールの送信に失敗」 スクリプト化可能タスク要素に必要な入力パラメータのバインドを示しています。

表 1-59. 「電子メールの送信に失敗」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[in-parameter] タイプ: VC:VirtualMachine 説明: パワーオンする仮想マシン。
toAddress	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: toAddress ソース パラメータ: toAddress[in-parameter] タイプ: 文字列 説明: このワークフローの結果を通知するユーザーのメール アドレス
emailErrorCode	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: emailErrorCode ソース パラメータ: emailErrorCode[attribute] タイプ: 文字列 説明: 電子メールの送信中のすべての例外のキャッチ

「電子メールの送信に失敗」スクリプト化可能タスク要素には、次のスクリプト関数が必要です。

```
//Writes the following event in the Orchestrator database
Server.error("Couldn't send result email to '"+toAddress+"' for VM '"+vm.name+"', exception
found: "+emailErrorCode);
```

「電子メールを送信」スクリプト化可能タスク要素

「仮想マシンの開始と電子メールの送信」ワークフローの目的は、そのワークフローが仮想マシンを開始したときに管理者に通知することです。それを行うには、電子メールを送信するスクリプト化可能タスクを定義する必要があります。電子メールを送信するために、「電子メールを送信」スクリプト化可能タスク要素には SMTP サーバ、メールの送信者と受信者のアドレス、メールの件名、およびメールの内容が必要です。

次の表は、「電子メールを送信」スクリプト化可能タスク要素に必要な入力パラメータと出力パラメータのバインドを示しています。

表 1-60. 「電子メールを送信」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、 またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ : vm ソース パラメータ : vm[in-parameter] タイプ : VC:VirtualMachine 説明: パワーオンする仮想マシン。
toAddress	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ : toAddress ソース パラメータ : toAddress[in-parameter] タイプ : 文字列 説明: このワークフローの結果を通知するユーザーのメール アドレス
body	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ : body ソース パラメータ : body[attribute] タイプ : 文字列 説明: メールの本文
smtpHost	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ : smtpHost ソース パラメータ : smtpHost[attribute] タイプ : 文字列 説明: メールの SMTP サーバ
fromAddress	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ : fromAddress ソース パラメータ : fromAddress[attribute] タイプ : 文字列 説明: 送信者のメール アドレス
subject	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ : subject ソース パラメータ : subject[attribute] タイプ : 文字列 説明: メールの件名

「電子メールを送信」スクリプト化可能タスク要素には、次のスクリプト関数が必要です。

```
//Create an instance of EmailMessage
var myEmailMessage = new EmailMessage() ;

//Apply methods on this instance that populate the email message
myEmailMessage.smtpHost = smtpHost;
myEmailMessage.fromAddress = fromAddress;
myEmailMessage.toAddress = toAddress;
myEmailMessage.subject = subject;
```

```
myEmailMessage.addMimePart(body , "text/html");

//Apply the method that sends the email message
myEmailMessage.sendMessage();
System.log("Sent email to '"+toAddress+"'");
```

単純なワークフロー サンプルの例外バインドの定義

ワークフロー エディタの[スキーマ]タブで例外バインドを定義します。例外バインドは、要素がエラーを処理する方法を定義します。

ワークフローの **startVM**、**vim3WaitTaskEnd**、**Send Email**、および **vim3WaitToolsStarted** の各要素が例外を返します。

開始する前に

次の操作を行います。

- 「[単純なワークフロー サンプルの作成](#)」。
- 「[単純なワークフロー サンプルのスキーマの作成](#)」。
- 「[単純なワークフロー サンプルのパラメータの定義](#)」。
- 「[単純なワークフロー サンプルの決定バインドの定義](#)」。
- 「[単純なワークフローでのアクション要素のバインド例](#)」。
- 「[単純なワークフローでのスクリプト化されたタスク要素のバインド例](#)」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 [スキーマ] タブで、**[startVM]** アクション要素の [編集] アイコン (✎) をクリックします。
- 2 [例外] タブをクリックします。
- 3 [未設定] ボタンをクリックします。
- 4 表示されたリストから **[errorCode]** を選択します。
- 5 前述の手順を繰り返して、**vim3WaitTaskEnd** と **vim3WaitToolsStarted** の両方について **errorCode** への例外バインドを設定します。
- 6 **[Send Email]** スクリプト化可能タスク要素の [編集] アイコン (✎) をクリックします。
- 7 [例外] タブをクリックします。
- 8 [未設定] ボタンをクリックします。
- 9 表示されたリストから **[emailErrorCode]** を選択します。
- 10 ワークフロー エディタの [スキーマ] タブの一番下にある [保存] をクリックします。

例外を返す要素について例外バインドを定義しました。

次に進む前に

属性およびパラメータで読み書きプロパティを設定する必要があります。

単純なワークフロー サンプルの属性の読み書きプロパティの設定

パラメータや属性が読み取り専用の定数または書き込み可能な変数のどちらであるかを定義できます。また、ユーザーが入力パラメータに指定できる値に制限を設定することもできます。

特定のパラメータを読み取り専用に設定すると、ワークフローのコア機能を中断させることなく、他の開発者がそのワークフローを適用したり変更したりできるようになります。

開始する前に

次の操作を行います。

- [「単純なワークフロー サンプルの作成」](#)。
- [「単純なワークフロー サンプルのスキーマの作成」](#)。
- [「単純なワークフロー サンプルのパラメータの定義」](#)。
- [「単純なワークフロー サンプルの決定バインドの定義」](#)。
- [「単純なワークフローでのアクション要素のバインド例」](#)。
- [「単純なワークフローでのスクリプト化されたタスク要素のバインド例」](#)。
- [「単純なワークフロー サンプルの例外バインドの定義」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの上部にある [全般] タブをクリックします。
 [属性] の下に、各属性の横にチェック ボックスが付いた、定義されているすべての属性のリストが表示されます。これらのチェック ボックスを選択すると、それらの属性が読み取り専用として設定されます。
- 2 チェック ボックスを選択して、次の属性を読み取り専用の定数にします。
 - progress
 - pollRate
 - timeout
 - smtpHost
 - fromAddress
 - subject

ワークフローの属性のどれが定数で、どれが変数であるかを定義しました。

次に進む前に

パラメータのプロパティを設定し、そのパラメータに指定できる値に制約を設定します。

単純なワークフロー サンプルのパラメータのプロパティの設定



ワークフロー エディタでパラメータのプロパティを設定できます。パラメータのプロパティを設定すると、そのパラメータの動作に影響を与え、またそのパラメータに指定できる値に制約が設定されます。



開始する前に

次の操作を行います。

- [「単純なワークフロー サンプルの作成」](#)。
- [「単純なワークフロー サンプルのスキーマの作成」](#)。
- [「単純なワークフロー サンプルのパラメータの定義」](#)。
- [「単純なワークフロー サンプルの決定バインドの定義」](#)。
- [「単純なワークフローでのアクション要素のバインド例」](#)。
- [「単純なワークフローでのスクリプト化されたタスク要素のバインド例」](#)。
- [「単純なワークフロー サンプルの例外バインドの定義」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [プレゼンテーション] タブをクリックします。
このワークフローに対して定義した 2 つの入力パラメータが一覧表示されます。
- 2 [(VC:VirtualMachine)vm] パラメータをクリックします。
- 3 画面の下半分にある [全般] タブで説明を追加します。
たとえば、「**起動する仮想マシン**」と入力します。
- 4 画面の下半分にある [プロパティ] タブをクリックします。
このタブで、(VC:VirtualMachine)vm パラメータのプロパティを設定できます。
- 5 [プロパティの追加] アイコン () をクリックします。
- 6 提示されたプロパティのリストから [必須入力] プロパティを選択し、[OK] をクリックして、その値を [はい] に設定します。
このプロパティを有効にした場合、ユーザーは、起動する仮想マシンを指定しなければ [仮想マシンの開始と電子メールの送信] ワークフローを実行できません。
- 7 [プロパティの追加] アイコン () をクリックします。
- 8 提示されたプロパティのリストから [値の選択方法] を選択し、[OK] をクリックして、指定できる値のリストから [リスト] を選択します。
このプロパティを設定するときは、ユーザーが (VC:VirtualMachine)vm 入力パラメータの値を選択する方法を設定します。

- 9 [プレゼンテーション] タブの上半分にある [(string)toAddress] パラメータをクリックします。
- 10 画面の下半分にある [説明] タブで説明を追加します。
たとえば、「**通知するユーザーのメール アドレス**」と入力します。
- 11 (string)toAddress の [プロパティ] タブをクリックし、[プロパティの追加] アイコン () をクリックします。
- 12 提示されたプロパティのリストから [必須入力] プロパティを選択し、[OK] をクリックして、その値を [はい] に設定します。
- 13 [プロパティの追加] アイコン () をクリックします。
- 14 提示されたプロパティのリストから、[一致する正規表現] を選択し、[OK] をクリックします。
このプロパティを使用すると、ユーザーが入力として指定できる内容に制約を設定できます。
- 15 [一致する正規表現] の [値] テキスト ボックスをクリックし、制約を **[a-zA-Z0-9_%-+]+@[a-zA-Z0-9-.\]+\.[a-zA-Z]{2,4}** に設定します。
これらの制約を設定すると、ユーザー入力がメール アドレスに適した文字に制限されます。ユーザーがワークフローを開始するときに受信者のメール アドレスとしてその他のいずれかの文字を入力しようとすると、ワークフローは開始されません。

両方のパラメータを必須にして、起動する仮想マシンをユーザーが選択する方法を定義し、さらに受信者のメール アドレスとして入力できる文字を制限しました。

次に進む前に

ユーザーがワークフローを実行するときにその入力パラメータ値を指定する入力パラメータ ダイアログ ボックスのレイアウト（またはプレゼンテーション）を作成する必要があります。

単純なワークフロー サンプルの入力パラメータ ダイアログ ボックスのレイアウトの設定

ワークフロー エディタで、入力パラメータ ダイアログ ボックスのレイアウトまたはプレゼンテーションを作成します。この入力パラメータ ダイアログ ボックスは、ユーザーが、実行のために入力パラメータが必要なワークフローを実行するときに開きます。

開始する前に

次の操作を行います。

- 「[単純なワークフロー サンプルの作成](#)」。
- 「[単純なワークフロー サンプルのスキーマの作成](#)」。
- 「[単純なワークフロー サンプルのパラメータの定義](#)」。
- 「[単純なワークフロー サンプルの決定バインドの定義](#)」。
- 「[単純なワークフローでのアクション要素のバインド例](#)」。
- 「[単純なワークフローでのスクリプト化されたタスク要素のバインド例](#)」。

- 「単純なワークフロー サンプルの例外バインドの定義」。
- 「単純なワークフロー サンプルの属性の読み書きプロパティの設定」。
- 「単純なワークフロー サンプルのパラメータのプロパティの設定」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [プレゼンテーション] タブをクリックします。
- 2 プレゼンテーション階層リストの [プレゼンテーション] ノードを右クリックし、[表示グループの作成] を選択します。

[プレゼンテーション] ノードの下に [新規ステップ] ノードと [新規グループ] サブノードが表示されます。
- 3 [新規ステップ] を右クリックし、[削除] を選択します。

このワークフローには 2 つのパラメータしかないため、入力パラメータ ダイアログ ボックス内の表示セクションの複数のレイヤは必要ありません。
- 4 [新規グループ] をダブルクリックしてグループ名を編集し、Enter キーを押します。

たとえば、表示グループに「**仮想マシン**」という名前を付けます。

ここで入力したテキストは、ユーザーがワークフローを開始するときに入力パラメータ ダイアログ ボックスの見出しとして表示されます。
- 5 [プレゼンテーション] タブの下部にある [全般] タブの [説明] テキスト ボックスで、新しい表示グループの説明を指定します。

たとえば、「**起動する仮想マシンの選択**」と入力します。

ここで入力したテキストは、ユーザーがワークフローを開始するときに入力パラメータ ダイアログ ボックスのプロンプトとして表示されます。
- 6 [(VC:VirtualMachine)vm] パラメータを [仮想マシン] 表示グループの下にドラッグします。

入力パラメータ ダイアログ ボックスでは、「仮想マシン」の見出しの下に、ユーザーが仮想マシン名を入力するテキスト ボックスが表示されます。
- 7 前の手順を繰り返して、**toAddress** パラメータのための表示グループを作成します。次のプロパティを設定します。
 - a 表示グループを作成し、それに「**受信者のメール アドレス**」という名前を付けます。
 - b 表示グループの説明を追加します。たとえば、「**この仮想マシンがパワーオンされたときに通知するユーザーのメール アドレスの入力**」とします。
 - c [toAddress] パラメータを [受信者のメール アドレス] 表示グループの下にドラッグします。

ユーザーがワークフローを実行するときに表示される入力パラメータ ダイアログ ボックスのレイアウトを設定しました。

次に進む前に

単純なワークフロー サンプルの開発を完了しました。これで、このワークフローを検証して実行できます。

単純なワークフロー サンプルの検証と実行

ワークフローを作成したら、検証を行って、予想されるエラーを検出することができます。エラーがなければ、そのワークフローを実行できます。

開始する前に

次の操作を行います。

- 「[単純なワークフロー サンプルの作成](#)」。
- 「[単純なワークフロー サンプルのスキーマの作成](#)」。
- 「[単純なワークフロー サンプルのパラメータの定義](#)」。
- 「[単純なワークフロー サンプルの決定バインドの定義](#)」。
- 「[単純なワークフローでのアクション要素のバインド例](#)」。
- 「[単純なワークフローでのスクリプト化されたタスク要素のバインド例](#)」。
- 「[単純なワークフロー サンプルの例外バインドの定義](#)」。
- 「[単純なワークフロー サンプルの属性の読み書きプロパティの設定](#)」。
- 「[単純なワークフロー サンプルのパラメータのプロパティの設定](#)」。
- 「[単純なワークフロー サンプルの入力パラメータ ダイアログ ボックスのレイアウトの設定](#)」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブにある [検証] をクリックします。
検証ツールによって、ワークフローの定義のエラーが検出されます。
- 2 エラーがなくなったら、ワークフロー エディタの下部にある [保存して閉じる] をクリックします。
Orchestrator クライアントに戻ります。
- 3 [ワークフロー] ビューをクリックします。
- 4 ワークフローの階層リストで、[ワークフロー サンプル] - [仮想マシンの開始と E メール送信] の順に選択します。
- 5 [仮想マシンの開始と E メール送信] ワークフローを右クリックし、[ワークフローの開始] を選択します。
入力パラメータ ダイアログ ボックスが開き、起動する仮想マシンと通知を送信するメール アドレスを指定するように求められます。
- 6 vCenter Server インベントリから、起動する仮想マシンを選択します。
- 7 E メール通知を送信するメール アドレスを入力します。

8 [送信] をクリックして、ワークフローを開始します。

[仮想マシンの開始と E メール送信] ワークフローの下にワークフロー トークンが表示されます。

9 ワークフロー トークンをクリックし、実行中のワークフローの進行状況を確認します。

ワークフローが正常に実行された場合は、選択した仮想マシンがパワーオン状態になり、指定した E メール受信者が確認 E メールを受信します。

次に進む前に

ワークフローに関する情報を確認するためのドキュメントを生成できます。[「ワークフロー ドキュメントの生成」](#)を参照してください。

複合ワークフローの開発

複合サンプル ワークフローの開発では、ワークフロー開発プロセスで最も一般的な手順と、カスタムの決定やループの作成などの高度なシナリオを示します。

複合ワークフローの演習では、特定のリソース プールに含まれているすべての仮想マシンのスナップショットを作成するワークフローを開発します。作成したワークフローは次のタスクを実行します。

- 1 スナップショットの作成対象となる仮想マシンを含むリソース プールを指定するようユーザーに求めます。
- 2 リソース プールに実行中の仮想マシンが含まれているかどうかを調べます。
- 3 実行中の仮想マシンがリソースにいくつ含まれているか調べます。
- 4 プール内で実行中の個々の仮想マシンが、作成するスナップショットのための特定の条件を満たしているかどうかを確認します。
- 5 仮想マシンのスナップショットを作成します。
- 6 スナップショットの作成対象となる仮想マシンがほかにもプールに存在するかどうかを調べます。
- 7 リソース プール内の該当するすべての仮想マシンのスナップショットがワークフローによって作成されるまで、検証およびスナップショット プロセスを繰り返します。

Orchestrator ドキュメントの Web ページからダウンロードできる Orchestrator のサンプルの ZIP ファイルには、「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローの完全なバージョンが含まれています。

開始する前に

この複合ワークフローの開発を行う前に、[「単純なサンプル ワークフローの開発」](#)の演習を順に行ってください。複合ワークフローを開発するための手順には開発プロセスの幅広い手順が含まれていますが、単純なワークフローの演習ほど細かくありません。

手順

1 複雑なワークフロー サンプルの作成

ワークフロー開発プロセスは、クライアントでワークフローを作成することから始める必要があります。

2 複雑なワークフローサンプル向けのカスタム アクションの作成

Check VM スクリプト化可能要素は、Orchestrator API に存在しないアクションを呼び出します。
getVMDiskModes アクションを作成する必要があります。

3 複雑なワークフロー サンプルのスキーマの作成

ワークフロー エディタでワークフローのスキーマを作成できます。ワークフローのスキーマにはワークフローで実行する要素が含まれ、ワークフローの論理フローを決定します。

4 (オプション) 複雑なワークフロー サンプルのゾーンの作成

必要に応じて、ワークフロー メモを追加してワークフローのさまざまなゾーンをハイライトできます。さまざまなワークフロー ゾーンを作成することで、込み入ったワークフローのスキーマを読解しやすくできます。

5 複雑なワークフロー サンプルのパラメータの定義

ワークフロー パラメータはワークフロー エディタで定義します。入力パラメータは、ワークフローが処理するデータを提供します。出力パラメータは、ワークフローが実行を完了したときに返すデータです。

6 複雑なワークフロー サンプルのバインドの定義

ワークフロー エディタでワークフローの要素をバインドすることができます。バインドは、ワークフローのデータフローを定義します。スクリプト化可能タスク要素をその JavaScript 関数にバインドすることもできます。

7 複雑なワークフロー サンプルの属性プロパティの設定

ワークフロー エディタの [全般] タブで属性プロパティを設定します。

8 複雑なワークフロー サンプルの入力パラメータのレイアウト作成

ワークフロー エディタの [プレゼンテーション] タブで、入力パラメータ ダイアログ ボックスのレイアウト (プレゼンテーション) を作成します。入力パラメータ ダイアログ ボックスはユーザーがワークフローを実行すると開きます。このダイアログ ボックスを使用して、ワークフローの実行で使用する入力パラメータを入力します。

9 複雑なワークフロー サンプルの検証と実行

ワークフローを作成したら、検証を行って、予想されるエラーを検出することができます。エラーがなければ、そのワークフローを実行できます。

複雑なワークフロー サンプルの作成

ワークフロー開発プロセスは、クライアントでワークフローを作成することから始める必要があります。

vCenter Server をインストールおよび設定する方法については、『vSphere のインストールとセットアップ』を参照してください。Orchestrator を設定する方法については、『VMware vRealize Orchestrator のインストールと構成』を参照してください。

開始する前に

次のコンポーネントがシステムにインストールおよび設定されていることを確認します。

- vCenter Server。いくつかの仮想マシンが含まれるリソース プールを制御します。
- ワークフローの階層リスト内の **ワークフロー サンプル** フォルダ (『[単純なワークフロー サンプルの作成](#)』で作成)。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー]-[ワークフロー サンプル] の順に選択します。
- 3 [ワークフロー サンプル] フォルダを右クリックし、[新しいワークフロー] を選択します。
- 4 新しいワークフローに**リソース プール内のすべての仮想マシンのスナップショットの作成**という名前を付け、[OK] をクリックします。

ワークフロー エディタが開きます。

- 5 ワークフロー エディタの [全般] タブで、バージョン番号の数値をクリックしてバージョン番号を増やします。
ワークフローの初期作成状態では、バージョンを [0.0.1] に設定します。
- 6 [サーバ再起動時の動作] 値をクリックし、サーバが再起動した後でワークフローを再開するかどうかを設定します。
- 7 [説明] テキスト ボックスで、ワークフローの処理の説明を入力します。
- 8 [全般] タブの一番下にある [保存] をクリックします。

これで、「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローを作成しました。

次に進む前に

カスタム アクションを作成する必要があります。

複雑なワークフローサンプル向けのカスタム アクションの作成

Check VM スクリプト化可能要素は、Orchestrator API に存在しないアクションを呼び出します。
getVMDiskModes アクションを作成する必要があります。

アクションの作成の詳細については、[第 3 章「アクションの開発」](#)を参照してください。

開始する前に

「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローを作成します。[「複雑なワークフロー サンプルの作成」](#)を参照してください。

手順

- 1 [保存して閉じる] をクリックしてワークフロー エディタを閉じます。
- 2 Orchestrator クライアントで [アクション] ビューをクリックします。
- 3 アクション階層リストのルートを右クリックして、[新規モジュール] を選択します。
- 4 新しいモジュールに **com.vmware.example** という名前を付けます。
- 5 [com.vmware.example] モジュールを右クリックして、[アクションの追加] を選択します。
- 6 **getVMDiskModes** という名前のアクションを作成します。
- 7 アクション エディタの [全般] タブで、バージョンの数字をクリックして、バージョン番号を増分します。

- 8 [全般] タブで、アクションについての次の説明を追加します。

```
This action returns an array containing the disk modes of all disks on a VM.
The elements in the array each have one of the following string values:
- persistent
- independent-persistent
- nonpersistent
- independent-nonpersistent
Legacy values:
- undoable
- append
```

- 9 [スクリプティング] タブをクリックします。

- 10 [スクリプティング] タブの上部のペインを右クリックし、[パラメータの追加] を選択して、次の入力パラメータを作成します。

- 名前: **vm**
- タイプ: **VC:VirtualMachine**
- 説明: **ディスク モードを返す仮想マシン**

- 11 次のスクリプトを [スクリプティング] タブの下部に追加します。

次のコードは、仮想マシンのディスクのディスク モードからなる配列を返します。

```
var devicesArray = vm.config.hardware.device;
var retArray = new Array();
if (devicesArray!=null && devicesArray.length!=0) {
    for (i in devicesArray) {
        if (devicesArray[i] instanceof VcVirtualDisk) {
            retArray.push(devicesArray[i].backing.diskMode);
        }
    }
}
return retArray;
```

- 12 [保存して閉じる] をクリックして、[アクション] パレットを終了します。

これで、「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローに必要なカスタム アクションを定義しました。

次に進む前に

ワークフローのスキーマを作成します。

複雑なワークフロー サンプルのスキーマの作成

ワークフロー エディタでワークフローのスキーマを作成できます。ワークフローのスキーマにはワークフローで実行する要素が含まれ、ワークフローの論理フローを決定します。

開始する前に

次の操作を行います。

- 「複雑なワークフロー サンプルの作成」。
- 「複雑なワークフローサンプル向けのカスタム アクションの作成」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 次のスキーマ要素をワークフローのスキーマに追加します。

要素タイプ	要素名	スキーマ内の位置
スクリプト化可能タスク	Initializing	Start 要素の下
決定	VMs to Process?	Initializing スクリプト化可能タスク要素の下
スクリプト化可能タスク	Pool Has No VMs	VMs to Process? カスタム決定要素の下で、赤色の矢印でリンク
カスタム決定	Remaining VMs?	VMs to Process? カスタム決定要素の右で、緑色の矢印でリンク
アクション	getVMDiskModes	Remaining VMs? カスタム決定要素の右で、緑色の矢印でリンク
カスタム決定	Create Snapshot?	getVMDiskModes アクション要素の右で、青色の矢印でリンク
ワークフロー	Create a snapshot	Create Snapshot? カスタム決定要素の上で、緑色の矢印でリンク
スクリプト化可能タスク	VM Snapshots	Create a snapshot ワークフローの左で、青色の矢印でリンク
スクリプト化可能タスク	Increment	VM Snapshots スクリプト化可能タスク要素の左で、青色の矢印でリンク
スクリプト化可能タスク	Set Output	Pool Has No VMs スクリプト化可能タスク要素の右で、青色の矢印でリンク

- 3 **Log Exception** スクリプト化可能タスク要素を追加します。
 - a スナップショット作成ワークフローから **End** 要素に例外処理リンクを作成します。
 - b スナップショット作成ワークフローから **End** 要素にリンクする赤色の破線矢印に、スクリプト化可能タスク要素をドラッグします。
 - c スクリプト化可能タスク要素をダブルクリックし、名前を**例外をログ**に変更します。
 - d **Log Exception** スクリプト化可能タスク要素を **VM Snapshots** スクリプト化可能タスク要素の上に移動します。
- 4 すべての **End** 要素 (**Set Output** スクリプト化可能タスク要素の右にある **End** 要素を除く) をリンク解除します。

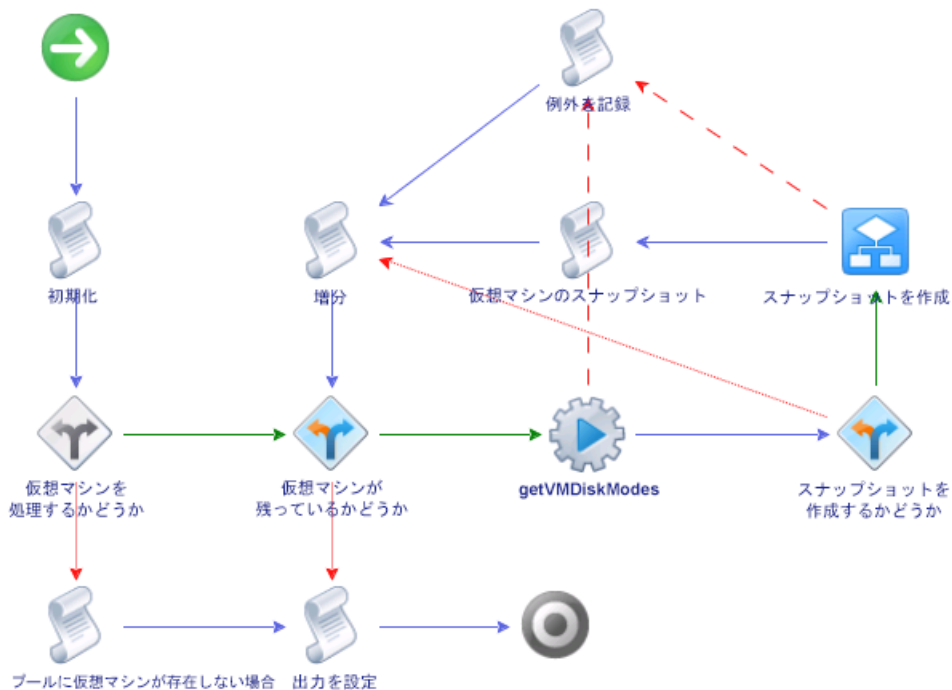
5 次の表の説明に従って残りの要素をリンクします。

要素	リンク先	矢印のタイプ	説明
getVMDiskModes アクション要素	Log Exception スクリプト化可能タスク要素	赤色の破線	例外処理
Create Snapshot? カスタム決定要素	Increment スクリプト化可能タスク要素	赤色	False 結果
Log Exception スクリプト化可能タスク要素	Increment スクリプト化可能タスク要素	青色	通常のワークフロー進行
Increment スクリプト化可能タスク要素	Remaining VMs? カスタム決定要素	青色	通常のワークフロー進行
Remaining VMs? カスタム決定要素	Set Output スクリプト化可能タスク要素	赤色	False 結果

6 [スキーマ] タブの一番下にある [保存] をクリックします。

次の図に、「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローのリンクされた要素の様子を示します。

図 1-12. 「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフロー サンプルのリンク



次に進む前に

必要に応じて、ワークフロー メモを使用してワークフロー ゾーンを定義できます。

(オプション) 複雑なワークフロー サンプルのゾーンの作成

必要に応じて、ワークフロー メモを追加してワークフローのさまざまなゾーンをハイライトできます。さまざまなワークフロー ゾーンを作成することで、込み入ったワークフローのスキーマを読解しやすくなります。

開始する前に

次の操作を行います。

- 「複雑なワークフロー サンプルの作成」。
- 「複雑なワークフロー サンプルのスキーマの作成」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

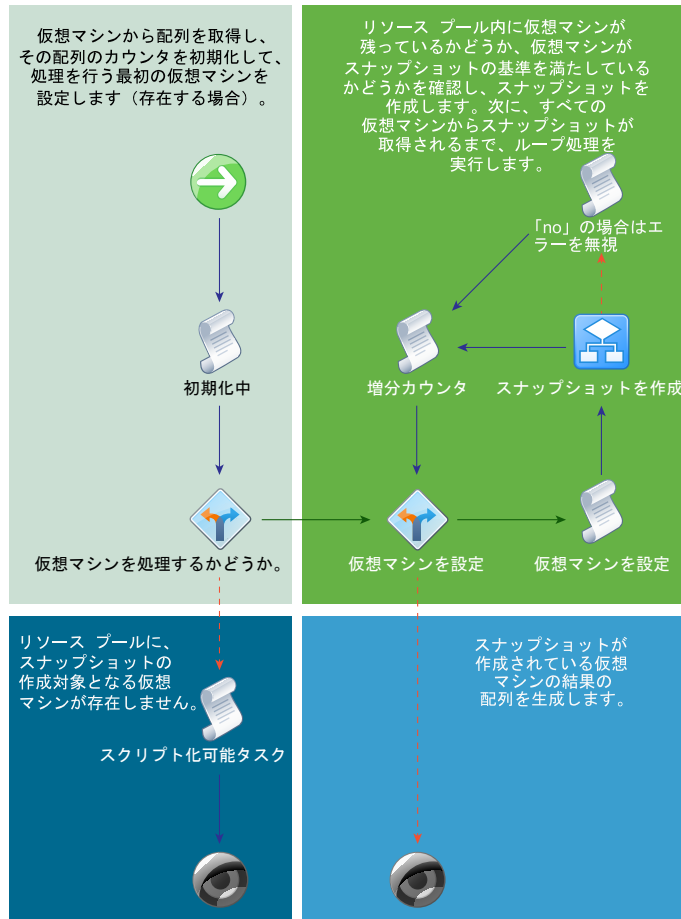
- 1 ワークフロー メモを使用して次のワークフロー ゾーンを作成します。

ゾーン内の要素	説明
開始要素、「初期化」スクリプト化可能タスク、処理を行う仮想マシン。カスタム決定	リソース プールから仮想マシンの配列を取得し、その配列のカウンタを初期化して、処理を行う最初の仮想マシンを設定します(存在する場合)。
「プールに対象仮想マシンなし」スクリプト化可能タスク	リソース プールに、スナップショットの作成対象となる仮想マシンが存在しません。
仮想マシンが残っているか。カスタム決定、 getVMDisksModes アクション、スナップショットを作成するか。決定、「スナップショット作成」ワークフロー、「仮想マシンのスナップショット」スクリプト化可能タスク、「増分」スクリプト化可能タスク、「例外をログ」スクリプト化可能タスク	リソース プール内に仮想マシンが残っているかどうか、仮想マシンがスナップショットの基準を満たしているかどうかを確認し、スナップショットを作成します。次に、すべての仮想マシンからスナップショットが取得されるまで、ループ処理を実行します。
「出力設定」スクリプト化可能タスク、終了要素	スナップショットが取得された仮想マシンの結果の配列を生成します。

- 2 ワークフロー メモを選択し、Ctrl + E キーを押して背景色を選択します。
- 3 ワークフロー エディタの [スキーマ] タブの一番下にある [保存] をクリックします。

ワークフロー ゾーンは次の図のようになります。

図 1-13. 「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフロー サンプルのスキーマ図



次に進む前に

ワークフローの入力および出力パラメータを定義する必要があります。

複雑なワークフロー サンプルのパラメータの定義

ワークフロー パラメータはワークフロー エディタで定義します。入力パラメータは、ワークフローが処理するデータを提供します。出力パラメータは、ワークフローが実行を完了したときに返すデータです。

開始する前に

次の操作を行います。

- 「複雑なワークフロー サンプルの作成」。
- 「複雑なワークフロー サンプルのスキーマの作成」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [入力] タブをクリックします。

2 次の入力パラメータを定義します。

- 名前: **resourcePool**
- タイプ: **VC:ResourcePool**
- 説明: **スナップショットの作成対象となる仮想マシンが含まれるリソース プール。**

3 ワークフロー エディタの [出力] タブをクリックします。

4 次の出力パラメータを定義します。

- 名前: **snapshotVmArrayOut**
- タイプ: **Array/VC:VirtualMachine**
- 説明: **スナップショットの作成が終了した仮想マシンの配列。**

ワークフローの入力および出力パラメータを定義しました。

次に進む前に

要素パラメータ間のバインドを定義する必要があります。

複雑なワークフロー サンプルのバインドの定義

ワークフロー エディタでワークフローの要素をバインドすることができます。バインドは、ワークフローのデータ フローを定義します。スクリプト化可能タスク要素をその JavaScript 関数にバインドすることもできます。

開始する前に

次の操作を行います。

- [「複雑なワークフロー サンプルの作成」](#)。
- [「複雑なワークフロー サンプルのスキーマの作成」](#)
- [「複雑なワークフロー サンプルのパラメータの定義」](#)
- 定義する必要があるバインドを確認します。[「複雑なワークフローサンプルのバインド」](#)を参照してください。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [スキーマ] タブをクリックします。
- 2 バインドを定義します。
- 3 [スキーマ] タブの一番下にある [保存] をクリックします。

要素のすべての入力および出力パラメータが、適切なパラメータのタイプと値にバインドされました。

次に進む前に

属性プロパティを設定します。

複雑なワークフローサンプルのバインド

バインドは、単純なワークフローサンプルのアクション要素が入力および出力パラメータを処理する方法を定義します。

リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローでは、次の入力および出力パラメータをバインドする必要があります。また、スクリプト化可能タスク要素に JavaScript 関数を定義します。

既存のパラメータにバインドする場合は、バインドすることによって、元のパラメータのタイプと説明の値が継承されます。

スクリプト化可能タスクの初期化

スクリプト化可能タスクの初期化要素は、ワークフローの属性を初期化します。次の表に、スクリプト化可能タスクの初期化要素で必要な入力および出力パラメータのバインドを示します。

表 1-61. スクリプト化可能タスクの初期化要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
resourcePool	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: resourcePool ソース パラメータ: resourcePool[in-parameter] タイプ: VC:ResourcePool 説明: スナップショットの作成対象となる仮想マシンを含むリソース プール
allVMs	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: allVMs ソース パラメータ: allVMs[attribute] タイプ: Array/VC:VirtualMachine 説明: リソース プール内の仮想マシン。
numberOfVMs	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: numberOfVMs ソース パラメータ: numberOfVMs[attribute] タイプ: 数値 説明: resourcePool 内に見つかった仮想マシンの数
vmCounter	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: vmCounter ソース パラメータ: vmCounter[attribute] タイプ: 数値 説明: 配列内の仮想マシンのカウンタ

表 1-61. スクリプト化可能タスクの初期化要素のバインド (続き)

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vm	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[attribute] タイプ: VC:VirtualMachine 説明: スナップショットを作成中の現在の仮想マシン
snapshotVmArray	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: snapshotVmArray ソース パラメータ: snapshotVmArray[attribute] タイプ: Array/VC:VirtualMachine 説明: スナップショットの作成が終了した仮想マシンの配列

スクリプト化可能タスク要素の初期化は、次のスクリプト関数を実行します。

```
//Retrieve an array of virtual machines contained in the specified Resource Pool
allVMs = resourcePool.vm;
//Initialize the size of the Array and the first VM to snapshot
if (allVMs!=null && allVMs.length!=0) {
    numberOfVms = allVMs.length;
    vm = allVMs[0];
} else {
    numberOfVms = 0;
}
//Initialize the VM counter
vmCounter = 0;
//Initializing the array of VM snapshots
snapshotVmArray = new Array();
```

「処理を行う仮想マシン」決定要素

「処理を行う仮想マシン」決定要素は、リソース プール内にスナップショットの作成対象となる仮想マシンがあるかどうかを判断します。次の表に、「処理を行う仮想マシン」決定要素に必要なバインドを示します。

表 1-62. 「処理を行う仮想マシン」決定要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
numberOfVMs	決定	バインド	<ul style="list-style-type: none"> ■ ソース パラメータ： numberOfVMs[attribute] ■ 決定ステートメント：より大きい ■ 値：0.0 ■ 説明： resourcePool 内に見つかった仮想マシンの数

「プールに対象仮想マシンなし」スクリプト化可能タスク要素

「プールに対象仮想マシンなし」スクリプト化可能タスク要素は、リソース プールに対象となる仮想マシンが含まれないことを Orchestrator データベースのログに記録します。次の表に、「プールに対象仮想マシンなし」スクリプト化可能タスク要素に必要なバインドを示します。

表 1-63. 「プールに対象仮想マシンなし」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
resourcePool	IN	バインド	<ul style="list-style-type: none"> ■ ローカル パラメータ：resourcePool ■ ソース パラメータ： resourcePool[in-parameter] ■ タイプ：VC:ResourcePool ■ 説明： スナップショットの作成対象となる仮想マシンが含まれるリソース プール。

「プールに対象仮想マシンなし」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
//Writes the following event in the Orchestrator database
Server.warn("The specified ResourcePool "+resourcePool.name+" does not contain any VMs.");
```

「残りの仮想マシン」カスタム決定要素

「残りの仮想マシン」カスタム決定要素は、リソース プール内にスナップショットの作成対象となる仮想マシンが残っているかどうかを判断します。次の表に、「残りの仮想マシン」カスタム決定要素に必要なバインドを示します。

表 1-64. 「残りの仮想マシン」カスタム決定要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
numberOfVms	IN	バインド	<ul style="list-style-type: none"> ■ ソース パラメータ： numberOfVms[attribute] ■ 決定ステートメント：より大きい ■ 値：0.0 ■ 説明： resourcePool 内に見つかった仮想マシンの数
vmCounter	IN	バインド	<ul style="list-style-type: none"> ■ ローカル パラメータ：vmCounter ■ ソース パラメータ： vmCounter[attribute] ■ タイプ：数値 ■ 説明：配列内の仮想マシンのカウンタ

「残りの仮想マシン」カスタム決定要素は、次のスクリプト関数を実行します。

```
//Checks if the workflow has reached the end of the array of VMs
if (vmCounter < numberOfVms) {
    return true;
} else {
    return false;
}
```

getVMDisksModes アクション要素

getVMDisksModes アクション要素は、仮想マシン内で実行中のディスクのモードを取得します。次の表に、**getVMDisksModes** アクション要素に必要なバインドを示します。

表 1-65. getVMDisksModes アクション要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ■ ローカル パラメータ：vm ■ ソース パラメータ：vm[attribute] ■ タイプ：VC:VirtualMachine ■ 説明： スナップショットを作成中の現在の仮想マシン
actionResult	OUT	作成	<ul style="list-style-type: none"> ■ ローカル パラメータ：actionResult ■ ソース パラメータ： vmDisksModes[attribute] ■ タイプ：配列/文字列 ■ 説明：仮想マシンの現在のディスク モード
errorCode	例外	作成	ローカル パラメータ： errorCode

「スナップショットの作成対象」カスタム決定要素

「スナップショットの作成対象」カスタム決定要素は、仮想マシンのディスク モードに応じて、仮想マシンのスナップショットを作成するかどうかを決定します。次の表に、「スナップショットの作成対象」カスタム決定要素に必要なバインドを示します。

表 1-66. 「スナップショットの作成対象」決定要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vmDisksMode	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: <code>vmDisksMode</code> ソース パラメータ: <code>vmDisksMode[attribute]</code> タイプ: 配列/文字列 説明: 仮想マシンの現在のディスク モード
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: <code>vm</code> ソース パラメータ: <code>vm[attribute]</code> タイプ: <code>VC:VirtualMachine</code> 説明: スナップショットを作成中の現在の仮想マシン

「スナップショットの作成対象」カスタム決定要素は、次のスクリプト関数を実行します。

```
//A snapshot cannot be taken if one of its disks is in independent mode
// (independent-persistent or independent-nonpersistent)
var containsIndependentDisks = false;
if (vmDisksModes!=null && vmDisksModes.length>0) {
    for (i in vmDisksModes) {
        if (vmDisksModes[i].charAt(0)=="i") {
            containsIndependentDisks = true;
        }
    }
} else {
    //if no disk found no need to try to snapshot the VM
    System.warn("Won't snapshot '"+vm.name+"', no disks found");
    return false;
}
if (containsIndependentDisks) {
    System.warn("Won't snapshot '"+vm.name+"', independent disk(s) found");
    return false;
} else {
    System.log("Snapshotting '"+vm.name+"'");
    return true;
}
```

「スナップショットの作成」ワークフロー要素

「スナップショットの作成」ワークフロー要素は、仮想マシンのスナップショットを作成します。次の表に、「スナップショットの作成」ワークフロー要素に必要なバインドを示します。

表 1-67. 「スナップショットの作成」ワークフロー要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[attribute] タイプ: VC:VirtualMachine 説明: スナップショットの作成対象となるアクティブな仮想マシン。
name	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: name ソース パラメータ: snapshotName[attribute] タイプ: 文字列 説明: このスナップショットの名前。スナップショットの名前は、この仮想マシンに一意である必要はありません
description	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: description ソース パラメータ: snapshotDescription[attribute] タイプ: 文字列 説明: このスナップショットの説明。
memory	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: memory ソース パラメータ: snapshotMemory[attribute] タイプ: ブール値 値: no 説明: TRUE の場合、仮想マシンの内部の状態のダンプ(メモリ ダンプ)がスナップショットに含まれます。
quiesce	IN	作成	<ul style="list-style-type: none"> ローカル パラメータ: quiesce ソース パラメータ: snapshotQuiesce[attribute] タイプ: ブール値 値: yes 説明: TRUE の場合で、スナップショットの作成時に仮想マシンに電源が入っている場合、VMware Tools を使用して、仮想マシン内のファイル システムが静止されます。

表 1-67. 「スナップショットの作成」ワークフロー要素のバインド (続き)

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
snapshot	OUT	作成	<ul style="list-style-type: none"> ローカル パラメータ: snapshot ソース パラメータ: NULL タイプ: VC:VirtualMachineSnapshot 説明: 作成されたスナップショット。
errorCode	例外	作成	ローカル パラメータ: errorCode

「仮想マシンのスナップショット」スクリプト化可能タスク要素

「仮想マシンのスナップショット」スクリプト化可能タスク要素は、スナップショットを配列に追加します。次の表に、「仮想マシンのスナップショット」スクリプト化可能タスク要素に必要なバインドを示します。

表 1-68. 「仮想マシンのスナップショット」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[attribute] タイプ: VC:VirtualMachine 説明: スナップショットの作成対象となるアクティブな仮想マシン。
snapshotVmArray	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: snapshotVmArray ソース パラメータ: snapshotVmArray[attribute] タイプ: Array/VC:VirtualMachine 説明: スナップショットの作成が終了した仮想マシンの配列
snapshotVmArray	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ: snapshotVmArray ソース パラメータ: snapshotVmArray[attribute] タイプ: Array/VC:VirtualMachine 説明: スナップショットの作成が終了した仮想マシンの配列

「仮想マシンのスナップショット」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
//Writes the following event in the Orchestrator database
Server.log("Successfully took snapshot of the VM '"+vm.name);
//Inserts the VM snapshot in an array
snapshotVmArray.push(vm);
```

「増分」スクリプト化可能タスク要素

「増分」スクリプト化可能タスク要素は、配列内の仮想マシンの数をカウントするカウンタを増分します。次の表に、「増分」スクリプト化可能タスク要素に必要なバインドを示します。

表 1-69. 「増分」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vmCounter	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vmCounter ソース パラメータ: vmCounter[attribute] タイプ: 数値 説明: 配列内の仮想マシンのカウンタ
allVMs	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: allVMs ソース パラメータ: allVMs[attribute] タイプ: Array/VC:VirtualMachine 説明: リソース プール内の仮想マシン。
vmCounter	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vmCounter ソース パラメータ: vmCounter[attribute] タイプ: 数値 説明: 配列内の仮想マシンのカウンタ
vm	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[attribute] タイプ: VC:VirtualMachine 説明: スナップショットを作成中の現在の仮想マシン

「増分」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
//Increases the array VM counter
vmCounter++;
//Sets the next VM to be snapshot in the attribute vm
vm = allVMs[vmCounter];
```

「例外ログ」スクリプト化可能タスク要素

「例外ログ」スクリプト化可能タスク要素は、ワークフロー要素およびアクション要素からの例外を処理します。次の表に、「例外ログ」スクリプト化可能タスク要素に必要なバインドを示します。

表 1-70. 「例外ログ」スクリプト化可能タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
vm	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: vm ソース パラメータ: vm[attribute] タイプ: VC:VirtualMachine 説明: スナップショットを作成中の現在の仮想マシン
errorCode	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ: errorCode ソース パラメータ: errorCode[attribute] タイプ: 文字列 説明: 仮想マシンのスナップショット作成中にキャッチされた例外

「例外ログ」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
//Writes the following event in the Orchestrator database
Server.error("Couldn't snapshot the VM '"+vm.name+"', exception: "+errorCode);
```

「出力設定」スクリプト化可能タスク要素

「出力設定」スクリプト化可能タスク要素は、ワークフローの出力を生成します。出力パラメータには、スナップショットが作成された仮想マシンの配列が含まれます。次の表に、「出力設定」スクリプト化可能タスク要素に必要なバインドを示します。

表 1-71. 「出力設定」タスク要素のバインド

パラメータ名	バインド タイプ	既存パラメータへのバインド、またはパラメータの作成	バインドする値
snapshotVmArray	IN	バインド	<ul style="list-style-type: none"> ローカル パラメータ： snapshotVmArray ソース パラメータ： snapshotVmArray[attribute] タイプ： Array/VC:VirtualMachine 説明： スナップショットの作成が終了した仮想マシンの配列
snapshotVmArrayOut	OUT	バインド	<ul style="list-style-type: none"> ローカル パラメータ： snapshotVmArrayOut ソース パラメータ： snapshotVmArrayOut[out-parameter] タイプ： Array/VC:VirtualMachine 説明： スナップショットの作成が終了した仮想マシンの配列

「出力設定」スクリプト化可能タスク要素は、次のスクリプト関数を実行します。

```
//Passes the value of the internal attribute to a workflow output parameter
snapshotVmArrayOut = snapshotVmArray;
```

複雑なワークフロー サンプルの属性プロパティの設定

ワークフロー エディタの [全般] タブで属性プロパティを設定します。

開始する前に

次の操作を行います。

- [「複雑なワークフロー サンプルの作成」](#)。
- [「複雑なワークフロー サンプルのスキーマの作成」](#)。
- [「複雑なワークフロー サンプルのバインドの定義」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 [全般] タブをクリックします。

2 次の属性の読み取り専用のチェック ボックスを選択して、それらを読み取り専用の定数にします。

- snapshotName
- snapshotDescription
- snapshotMemory
- snapshotQuiesce

ワークフローの属性のどれが定数で、どれが変数であるかを定義しました。

次に進む前に

ワークフローのプレゼンテーションを作成する必要があります。これにより、ユーザーがワークフローを実行するときその入力パラメータ値を指定する入力パラメータ ダイアログ ボックスのレイアウトが作成されます。

複雑なワークフロー サンプルの入力パラメータのレイアウト作成

ワークフロー エディタの [プレゼンテーション] タブで、入力パラメータ ダイアログ ボックスのレイアウト（プレゼンテーション）を作成します。入力パラメータ ダイアログ ボックスはユーザーがワークフローを実行すると開きます。このダイアログ ボックスを使用して、ワークフローの実行で使用される入力パラメータを入力します。

開始する前に

次の操作を行います。

- [「複雑なワークフロー サンプルの作成」](#)。
- [「複雑なワークフロー サンプルのスキーマの作成」](#)。
- [「複雑なワークフロー サンプルのパラメータの定義」](#)。
- [「複雑なワークフロー サンプルのバインドの定義」](#)。
- [「複雑なワークフロー サンプルの属性プロパティの設定」](#)。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

- 1 ワークフロー エディタの [プレゼンテーション] タブをクリックします。

「リソース プール内のすべての仮想マシンのスナップショットの作成」ワークフローには入力パラメータが 1 つだけであるため、プレゼンテーションの作成は簡単です。

- 2 プレゼンテーション階層リストの [プレゼンテーション] ノードを右クリックして [表示グループの作成] を選択します。
- 3 [新規グループ] 要素の上に表示されている [新規ステップ] 要素を削除します。
- 4 [新規グループ] 要素をダブルクリックし、グループ名を **リソース プール** に変更します。
- 5 [プレゼンテーション] タブの一番下にある [全般] タブの [説明] テキスト ボックスに、[リソース プール] 表示グループの説明を入力します。

たとえば、**スナップショットの作成対象となる仮想マシンを含むリソース プールの名前を入力します。**とします。

6 (VC:ResourcePool)resourcePool パラメータをクリックします。

7 (VC:ResourcePool)resourcePool の [プロパティ] タブをクリックします。

8 [プロパティ] タブ内を右クリックし、[プロパティの追加] - [必須入力] の順に選択します。

9 [プロパティ] タブ内を右クリックし、[プロパティの追加] - [値の選択方法] の順に選択します。

このプロパティを設定するときは、ユーザーが (VC:ResourcePool)resourcePool 入力パラメータの値を選択する方法を設定します。

10 (VC:ResourcePool)resourcePool パラメータを [リソース プール] 表示グループの下にドラッグします。

ユーザーがワークフローを実行するときに表示されるダイアログ ボックスのレイアウトを作成しました。

次に進む前に

複雑なワークフロー サンプルの開発が完了しました。これで、ワークフローを検証して実行できるようになりました。

複雑なワークフロー サンプルの検証と実行

ワークフローを作成したら、検証を行って、予想されるエラーを検出することができます。エラーがなければ、そのワークフローを実行できます。

開始する前に

ワークフローの作成、そのスキーマのレイアウト、リンクとバインドの定義、パラメータのプロパティの定義、入力パラメータ ダイアログ ボックスのプレゼンテーションの作成を行います。

次の操作を行います。

- 「複雑なワークフロー サンプルの作成」。
- 「複雑なワークフローサンプル向けのカスタム アクションの作成」。
- 「複雑なワークフロー サンプルのスキーマの作成」。
- 「複雑なワークフロー サンプルのパラメータの定義」。
- 「複雑なワークフロー サンプルのバインドの定義」。
- 「複雑なワークフロー サンプルの属性プロパティの設定」。
- 「複雑なワークフロー サンプルの入力パラメータのレイアウト作成」。
- ワークフロー エディタで編集の対象となるワークフローを開きます。

手順

1 ワークフロー エディタの [スキーマ] タブにある [検証] をクリックします。

検証ツールによって、ワークフローの定義のエラーが検出されます。

2 エラーがなくなったら、ワークフロー エディタの下部にある [保存して閉じる] をクリックします。

Orchestrator クライアントに戻ります。

3 [ワークフロー] ビューをクリックします。

- 4 ワークフローの階層リストで、[ワークフロー サンプル] - [リソース プール内のすべての仮想マシンのスナップショットの作成] の順に選択します。
- 5 [リソース プール内のすべての仮想マシンのスナップショットの作成] ワークフローを右クリックし、[ワークフローの開始] を選択します。

入力パラメータ ダイアログ ボックスが開き、スナップショットを作成する仮想マシンが含まれているリソース プールを指定するように求められます。

- 6 [送信] をクリックして、ワークフローを実行します。

[リソース プール内のすべての仮想マシンのスナップショットの作成] ワークフローの下にワークフロー トークンが表示されます。

- 7 ワークフロー トークンをクリックし、実行中のワークフローの進行状況を確認します。

ワークフローが正常に実行された場合は、選択したリソース プール内のすべての仮想マシンのスナップショットが作成されます。

次に進む前に

ワークフローに関する情報を確認するためのドキュメントを生成できます。[「ワークフロー ドキュメントの生成」](#)を参照してください。

スクリプティング

Orchestrator は、JavaScript を使用して、ユーザーが Orchestrator にプラグインするテクノロジーの API にアクセスするアクション、ワークフロー要素、およびポリシーの作成元となるビルディング ブロックを作成します。

Orchestrator は、そのスクリプティング エンジンとして Mozilla Rhino 1.7R4 JavaScript エンジンを使用します。このスクリプティング エン진은、変数タイプのチェック、名前空間の管理、オート コンプリート、および例外処理を提供します。

Orchestrator ワークフロー エンジンを使用すると、if、loop、array、string などの基本的な JavaScript 言語機能を使用できます。スクリプティングでは Orchestrator API が提供するオブジェクト、またはプラグインを通して Orchestrator にインポートし、JavaScript オブジェクトにマッピングするその他の任意の API のオブジェクトを使用できます。Rhino については、Mozilla Rhino の Web サイトを参照してください。

この章では次のトピックについて説明します。

- [スクリプティングを必要とする Orchestrator 要素](#)
- [Orchestrator での Mozilla Rhino 実装の制限事項](#)
- [Orchestrator Scripting API の使用](#)
- [vCenter Server プラグインを使用した XPath 式の使用](#)
- [例外処理のガイドライン](#)
- [Orchestrator の JavaScript サンプル](#)

スクリプティングを必要とする Orchestrator 要素

すべての Orchestrator 要素でスクリプトの記述が必要なわけではありません。アプリケーションに最大限の柔軟性を提供するため、JavaScript 機能を追加して特定の要素をカスタマイズすることができます。

次の Orchestrator 要素にスクリプトを追加できます。

アクション	アクションはスクリプト化された機能です。1 つのアクションに対して記述するスクリプティングを 1 つの操作に制限して、アクションが他のワークフローなどの他の要素に再利用される可能性を最大にできます。または、1 つのアクションに多くの操作を含めて、ワークフローの複雑さを制限できますが、これはアクションを再利用する能力が低下します。
ポリシー	トリガ イベントをウォッチするスクリプトを使用してポリシーを設定します。トリガ イベントが発生すると、ポリシーはスクリプトに定義したオーケストレーション操作を起動します。
ワークフロー	スクリプト化可能タスク ワークフロー要素では、カスタムのスクリプト化された操作、またはワークフローで使用できる操作のシーケンスを記述できます。また、スクリプトに、 true または false を返すカスタム決定要素の決定ステートメントを定義します。

Orchestrator での Mozilla Rhino 実装の制限事項

Orchestrator は Mozilla Rhino 1.7R4 JavaScript エンジンを使用します。ただし、Orchestrator での Rhino の実装にはいくつかの制限事項があります。

ワークフローのスクリプトを記述するときには、Orchestrator での Mozilla Rhino 実装に関する次の制限事項を考慮する必要があります。

- ワークフローを実行するとき、ワークフロー要素間で受け渡されるオブジェクトは、JavaScript オブジェクトではありません。ワークフロー要素間で受け渡されるのは、シリアル化された Java オブジェクトであり、JavaScript イメージが含まれています。そのため、JavaScript 言語全体を使用することはできません。使用できるのは、API Explorer に表示されるクラスのみです。ワークフロー要素間で機能オブジェクトを受け渡すことはできません。
- Orchestrator は、Rhino root コンテキスト以外のコンテキストでスクリプト化可能タスク要素に含まれるコードを実行できます。Orchestrator は、スクリプト化可能タスク要素とアクションを透過的にラップして JavaScript 機能を生成し、実行します。**System.log(this);** が含まれているスクリプト化可能タスク要素は、Rhino の標準実装とは異なる方法でグローバル オブジェクト **this** を表示します。
- ワークフローではなくスクリプトから、シリアル化不能オブジェクトを返すアクションを呼び出すことができます。シリアル化不能オブジェクトを返すアクションを呼び出すには、**System.getModule<ModuleName>.action()** メソッドを使用してアクションを呼び出すスクリプト化可能タスク要素を記述する必要があります。
- ワークフローの検証では、ワークフローの属性タイプがアクションまたはサブワークフローの入力タイプと異なるかどうかはチェックされません。ワークフロー入力パラメータのタイプを（たとえば、**VIM3:VirtualMachine** から **VC:VirtualMachine** に）変更するとき、元の入力タイプを使用するスクリプト化可能タスクまたはアクションを更新しない場合、ワークフローは検証されますが、実行されません。

Orchestrator Scripting API の使用

Orchestrator API では、Orchestrator がプラグインを使用してアクセスする、技術内のすべてのオブジェクトと機能が JavaScript オブジェクトおよびメソッドとして公開されています。

たとえば、Orchestrator API を介して vCenter Server API の JavaScript 実装にアクセスして、作成するスクリプト化された要素に vCenter 操作を含めることができます。また、Orchestrator サーバにインストールした他のすべてのプラグインから、オブジェクトの JavaScript 実装にアクセスすることもできます。サードパーティ アプリケーションへのカスタム プラグインを作成する場合は、その API のオブジェクトを、Orchestrator API で公開される JavaScript オブジェクトにマッピングします。

手順

1 ワークフロー エディタからのスクリプト エンジンへのアクセス

Orchestrator のスクリプト エンジンには、Mozilla Rhino 1.7R4 JavaScript エンジンを採用しているため、ワークフロー内のスクリプト化された要素のスクリプトを記述するのに役立ちます。スクリプト化されたワークフロー要素のスクリプト エンジンには、ワークフロー エディタの [スクリプティング] タブからアクセスします。

2 アクション エディタまたはポリシー エディタからのスクリプト エンジンへのアクセス

Orchestrator のスクリプト エンジンには、Mozilla Rhino JavaScript エンジンを採用しているため、アクションやポリシーのスクリプトを記述するのに役立ちます。アクションおよびポリシー用のスクリプト エンジンには、アクション エディタおよびポリシー エディタの [スクリプティング] タブからアクセスします。

3 Orchestrator API Explorer へのアクセス

Orchestrator に用意されている API Explorer を使用すると、Orchestrator API を検索したり、スクリプト化された要素で使用できる JavaScript オブジェクトのドキュメントを参照したりすることができます。

4 Orchestrator API Explorer によるオブジェクトの検索

Orchestrator API では、すべての vCenter Server API を含む、全部のプラグイン テクノロジーの API が公開されます。Orchestrator API Explorer では、スクリプトに追加する必要があるオブジェクトを検索することができます。

5 スクリプトの記述

Orchestrator のスクリプト エンジンには、スクリプトの記述に役立ちます。関数が自動で挿入され、スクリプトの行がオート コンプリートされるため、スクリプトの作成プロセスを短縮でき、スクリプト内にエラーを記述する可能性を最小限に抑えることができます。

6 スクリプトへのパラメータの追加

Orchestrator のスクリプト エンジンには、使用できるパラメータをスクリプトにインポートするのに役立ちます。

7 JavaScript とワークフローから Orchestrator サーバ ファイル システムにアクセスする

JavaScript とワークフローから Orchestrator サーバ ファイル システムにアクセスする場合、アクセスできるのは特定のディレクトリだけに制限されています。

8 JavaScript から Java クラスにアクセスする

デフォルトでは、Orchestrator の JavaScript からアクセスできる Java クラスのセットは制限されています。JavaScript からさまざまな Java クラスにアクセスするには、そのための Orchestrator システム プロパティを設定する必要があります。

9 JavaScript からオペレーティング システム コマンドにアクセスする

Orchestrator API には、Orchestrator サーバ ホストのオペレーティング システム上で各種のコマンドを実行するための **Command** というスクリプト クラスが用意されています。Orchestrator サーバ ホストに対する無許可のアクセスを防ぐため、デフォルトで、Orchestrator アプリケーションには **Command** クラスを実行するための権限が設定されていません。

ワークフロー エディタからのスクリプト エンジンへのアクセス

Orchestrator のスクリプト エン진은、Mozilla Rhino 1.7R4 JavaScript エンジンを採用しているため、ワークフロー内のスクリプト化された要素のスクリプトを記述するのに役立ちます。スクリプト化されたワークフロー要素のスクリプト エンジンには、ワークフロー エディタの [スクリプティング] タブからアクセスします。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 Orchestrator クライアントの [ワークフロー] ビューでワークフローを右クリックし、[編集] を選択します。
- 3 ワークフロー エディタで [スキーマ] タブをクリックします。
- 4 ワークフロー スキーマにスクリプト化可能タスク要素またはカスタム決定要素を追加します。
- 5 スクリプト化可能要素の [スクリプティング] タブをクリックします。

これでスクリプト エンジンにアクセスできたため、ワークフロー要素のスクリプト関数を定義できます。[スクリプティング] タブでは、API を参照したり、オブジェクトに関するドキュメントを参照できるほか、オブジェクトの検索、JavaScript の記述を行えます。

次に進む前に

API Explorer を使用して Orchestrator API を検索します。

アクション エディタまたはポリシー エディタからのスクリプト エンジンへのアクセス

Orchestrator のスクリプト エンジンには、Mozilla Rhino JavaScript エンジンを採用しているため、アクションやポリシーのスクリプトを記述するのに役立ちます。アクションおよびポリシー用のスクリプト エンジンには、アクション エディタおよびポリシー エディタの [スクリプティング] タブからアクセスします。

手順

- 1 Orchestrator クライアントで、スクリプトを編集する要素の種類に応じてドロップダウン メニューからオプションを選択します。

オプション	説明
設計	アクション要素のスクリプトを編集するには、このオプションを選択します。
実行	ポリシーのスクリプトを編集するには、このオプションを選択します。

- 2 [アクション] または [ポリシー] ビューでアクションまたはポリシーを右クリックし、[編集] を選択します。
- 3 アクション エディタまたはポリシー エディタで [スクリプティング] タブをクリックします。

これでスクリプト エンジンにアクセスできたため、アクションまたはポリシー要素のスクリプト関数を定義できます。[スクリプティング] タブでは、API を参照したり、オブジェクトに関するドキュメントを参照できるほか、オブジェクトの検索、JavaScript の記述を行えます。

次に進む前に

API Explorer を使用して Orchestrator API を検索します。

Orchestrator API Explorer へのアクセス

Orchestrator に用意されている API Explorer を使用すると、Orchestrator API を検索したり、スクリプト化された要素で使用できる JavaScript オブジェクトのドキュメントを参照したりすることができます。

Orchestrator ドキュメントのホーム ページでは、vCenter Server プラグインの Scripting API のオンライン バージョンを参照できます。

手順

- 1 Orchestrator クライアントにログインします。
- 2 [ツール] - [API Explorer] の順に選択します。

API Explorer が表示されます。API Explorer を使用して Orchestrator API のすべてのオブジェクトと関数を検索できます。

次に進む前に

API Explorer を使用してスクリプト化可能な要素のスクリプトを記述します。

Orchestrator API Explorer によるオブジェクトの検索

Orchestrator API では、すべての vCenter Server API を含む、全部のプラグイン テクノロジーの API が公開されます。Orchestrator API Explorer では、スクリプトに追加する必要があるオブジェクトを検索することができます。

開始する前に

API Explorer を開きます。

手順

- 1 API Explorer の [検索] テキスト ボックスにオブジェクトの名前または名前の一部を入力し、[検索] をクリックします。

特定のオブジェクト タイプに限定して検索するには、[スクリプト クラス]、[属性およびメソッド]、[タイプおよび列挙] の各チェック ボックスをオフまたはオンにします。
- 2 提示されたリスト内の要素をダブルクリックします。

左側の階層リストで、オブジェクトが強調表示されます。階層リストの下側のドキュメント ペインに、オブジェクトに関する情報が表示されます。

次に進む前に

見つかったオブジェクトをスクリプトで使用します。

API Explorer 内の JavaScript オブジェクト

Orchestrator API Explorer は、[スクリプティング] タブの左側の階層ツリー内または API Explorer ダイアログ ボックスで、さまざまな種類の JavaScript オブジェクトを特定してグループ化します。API Explorer ではアイコンを使用することで、さまざまな種類のオブジェクトが識別しやすくなっています。

次の表に、Orchestrator API のオブジェクトの説明とオブジェクトのアイコンを示します。

表 2-1. Orchestrator API の JavaScript オブジェクト

オブジェクト	階層リスト内のアイコン	説明
タイプ		タイプ
機能セット		静的メソッドのセットを含む内部タイプ
プリミティブ		プリミティブ タイプ
オブジェクト		標準の Orchestrator スクリプト オブジェクト
属性		JavaScript 属性
メソッド		JavaScript メソッド
コンストラクタ		JavaScript コンストラクタ
列挙		JavaScript 列挙
文字列セット		文字列セット、デフォルト値
モジュール		一連のアクション
プラグイン	プラグインが定義するイメージ	プラグインが Orchestrator に公開する API

スクリプトの記述

Orchestrator のスクリプト エンジンには、スクリプトの記述に役立ちます。関数が自動で挿入され、スクリプトの行がオート コンプリートされるため、スクリプトの作成プロセスを短縮でき、スクリプト内にエラーを記述する可能性を最小限に抑えることができます。

開始する前に

編集するスクリプト化された要素を開き、その [スクリプティング] タブをクリックします。

手順

- 1 [スクリプティング] タブの左側にあるオブジェクトの階層リストを参照するか、または API Explorer の検索機能を使用して、スクリプトに追加するタイプ、クラス、またはメソッドを選択します。
- 2 タイプ、クラス、またはメソッドを右クリックし、[コピー] を選択します。

スクリプト エンジンで選択した要素をコピーできない場合は、このオブジェクトをスクリプトのコンテキストで使用することはできません。

- 3 スクリプト パッドで右クリックし、コピーした要素をスクリプト内の適切な場所に貼り付けます。

スクリプト エンジンによって要素がスクリプト内に入力され、そのコンストラクタとインスタンス名が付加されます。

たとえば、**Date** オブジェクトをコピーすると、スクリプト エンジンによって次のコードがスクリプトに貼り付けられます。

```
var myDate = new Date();
```

- 4 スクリプトに追加するメソッドをコピーして貼り付けます。

スクリプト エンジンによってメソッド呼び出しが完了し、必要な属性が追加されます。

たとえば、**com.vmware.library.vc.vm** モジュールから **cloneVM()** メソッドをコピーした場合、スクリプト エンジンによって次のコードがスクリプトに貼り付けられます。

```
System.getModule("com.vmware.library.vc.vm").cloneVM(vm, folder, name, spec)
```

スクリプト エンジンは、要素内で定義済みのパラメータを強調表示します。未定義のパラメータは、すべて非強調表示のままになります。

- 5 スクリプトに貼り付けた要素の末尾にカーソルを置き、Ctrl + スペース キーを押して、オブジェクトが呼び出ることができるメソッドと属性のコンテキスト リストから選択を行います。

注意 オート コンプリート機能は、現在試験段階にあります。

これで、オブジェクトや関数をスクリプトに追加しました。

次に進む前に

スクリプトにパラメータを追加します。

スクリプト キーワードの色分け

スクリプト化されたワークフロー要素の [スクリプティング] タブでスクリプトを追加する際に、特定のタイプのキーワードが異なる色で表示されます。これは、コードを見やすくするためです。

特に記載がない場合、すべてのスクリプトが通常の黒いフォントで表示されます。

表 2-2. スクリプト キーワードの色分け

キーワードのタイプ	[スクリプティング] タブのテキストの色
JavaScript の標準的なキーワード (if 、 else 、 for 、 new など)	太字の黒
変数の宣言 (var など)	緑
ループ内の修飾語句 (in など)	赤
Null 変数の値	紫
Null 変数以外の変数の値	緑
コードのコメント	斜体のグレー

表 2-2. スクリプト キーワードの色分け (続き)

キーワードのタイプ	[スクリプティング] タブのテキストの色
Orchestrator のプラグイン オブジェクト タイプ (VC:VirtualMachine や VC:Host など)	緑
出力テキスト	緑
ワークフローの属性	ピンク
ワークフローの入力	ピンク
ワークフローの出力	ピンク

スクリプトへのパラメータの追加

Orchestrator のスクリプト エンジンには、使用できるパラメータをスクリプトにインポートするのに役立ちます。

編集中の要素のパラメータをすでに定義している場合は、[スクリプティング] タブのツールバーにリンクとして表示されます。

開始する前に

編集するスクリプト化された要素を開いて、その [スクリプティング] タブを開いている必要があります。

手順

- 1 [スクリプティング] タブのスクリプティング パッドで、カーソルをスクリプト内の適切な場所へ移動します。
- 2 [スクリプティング] タブのツールバーで、パラメータ リンクをクリックします。

Orchestrator によって、カーソルの位置にそのパラメータが挿入されます。

- 3 NULL 値を持つパラメータをスクリプトに挿入します。

整数、ブール値、文字列などのプリミティブ タイプに NULL 値を渡すと、Orchestrator のスクリプト API によって、この引数のデフォルト値が自動的に設定されます。

これで、スクリプトにパラメータを追加できました。

次に進む前に

スクリプト内の Java クラスへのアクセスを追加します。

JavaScript とワークフローから Orchestrator サーバ ファイル システムにアクセスする

JavaScript とワークフローから Orchestrator サーバ ファイル システムにアクセスする場合、アクセスできるのは特定のディレクトリだけに制限されています。

JavaScript 関数とワークフローには、永続ディレクトリ **c:\orchestrator** での読み取り権限、書き込み権限、実行権限だけが設定されています。

Orchestrator 管理者は、システム プロパティを設定することにより、JavaScript 関数とワークフローが読み取り権限、書き込み権限、実行権限を持つフォルダを変更することができます。システム プロパティの構成方法については、「VMware vRealize Orchestrator のインストールと構成」を参照してください。

JavaScript 関数とワークフローには、サーバシステムのデフォルトの一時入出力フォルダでの読み取り権限、書き込み権限、実行権限も設定されています。現在の構成内容にかかわらず、すべての権限が構成されているファイルシステムにアクセスする唯一の確実な方法は、デフォルトの一時入出力フォルダに書き込む方法です。ただし、サーバを再起動すると、一時入出力フォルダに書き込んだファイルが失われます。

デフォルトの一時入出力フォルダを取得するには、JavaScript 関数で **System.getTempDirectory** メソッドを呼び出します。

System.getTempDirectory メソッドによるサーバ ファイル システムへのアクセス

管理者に対して適切な権限が設定されている Orchestrator サーバ システムのフォルダに書き込む別の方法として、デフォルトの一時入出力フォルダに書き込むことができます。

Orchestrator には、デフォルトの一時入出力フォルダに関する完全な読み取り権限、書き込み権限、実行権限がデフォルトで設定されています。デフォルトの一時入出力フォルダにアクセスするには、JavaScript 関数の **System.getTempDirectory** メソッドを使用します。

手順

- ◆ **java.io.temp-dir** フォルダにアクセスするには、以下のコード行を JavaScript 関数に追加します。

```
var tempDir = System.getTempDirectory()
```

JavaScript から Java クラスにアクセスする

デフォルトでは、Orchestrator の JavaScript からアクセスできる Java クラスのセットは制限されています。

JavaScript からさまざまな Java クラスにアクセスするには、そのための Orchestrator システム プロパティを設定する必要があります。

デフォルトでは、Orchestrator の JavaScript エンジンからアクセスできるのは **java.util.*** パッケージ内のクラスだけです。

Orchestrator 管理者は、システム プロパティを設定することにより、JavaScript 関数から他の Java クラスへのアクセスを許可することができます。システム プロパティの構成方法については、「VMware vRealize Orchestrator のインストールと構成」を参照してください。

JavaScript からオペレーティング システム コマンドにアクセスする

Orchestrator API には、Orchestrator サーバ ホストのオペレーティング システム上で各種のコマンドを実行するための **Command** というスクリプト クラスが用意されています。Orchestrator サーバ ホストに対する無許可のアクセスを防ぐため、デフォルトで、Orchestrator アプリケーションには **Command** クラスを実行するための権限が設定されていません。

Orchestrator 管理者は、**com.vmware.js.allow-local-process=true** というシステム プロパティを設定することにより、**Command** スクリプト クラスへのアクセスを許可することができます。

システム プロパティの構成方法については、「VMware vCenter Orchestrator のインストールと構成」を参照してください。

システム プロパティの構成方法については、「VMware vCenter Orchestrator のインストールと構成」を参照してください。

vCenter Server プラグインを使用した XPath 式の使用

vCenter Server プラグインのファインダ メソッドを使用して vCenter Server インベントリ オブジェクトのクエリを実行できます。XPath 式を使用して検索パラメータを定義できます。

vCenter Server プラグインには `getAllDatastores()`、`getAllResourcePools()`、`findAllForType()` など、オブジェクトのファインダ メソッド セットが含まれています。これらのメソッドを使用して Orchestrator サーバに接続している vCenter Server インスタンスのインベントリにアクセスし、ID や名前などのプロパティを指定して、オブジェクトを検索できます。

パフォーマンス上の理由により、検索クエリで一連のプロパティを指定しない限り、ファインダ メソッドではクエリ対象のオブジェクトのプロパティを返しません。

Orchestrator ドキュメントのホーム ページでは、vCenter Server プラグインの Scripting API のオンライン バージョンを参照できます。

重要 XPath 式に基づくクエリは、Orchestrator のパフォーマンスに影響を及ぼすことがあります。ファインダ メソッドは vCenter Server 側に存在する指定したタイプのオブジェクトをすべて返すほか、vCenter Server プラグイン側にクエリ フィルタを適用するためです。

vCenter Server プラグインを使用した XPath 式の使用

ファインダ メソッドを呼び出す際は、XPath クエリ言語に基づいた式を使用できます。検索では XPath 式に一致するすべてのインベントリ オブジェクトが返されます。すべてのプロパティがクエリの対象になる場合は、これらのプロパティを文字列アレイ形式の検索スクリプトに含めます。

次の JavaScript の例は **VcPlugin** スクリプト オブジェクトと XPath 式を使用して、vCenter Server の管理対象オブジェクトに含まれていて、かつ名前に **ds** という文字列が使用されているすべてのデータベース オブジェクトの名前を返しています。

```
var datastores = VcPlugin.getAllDatastores(null, "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
    System.log(datastore.name);
}
```

Server スクリプト オブジェクトと `findAllForType` ファインダ メソッドを使用して、同じ XPath 式を呼び出すことができます。

```
var datastores = Server.findAllForType("VC:Datastore", "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
    System.log(datastore.name);
}
```

次のスクリプトの例は、ID が **1** の数字で始まるすべてのホスト システム オブジェクトの名前を返しています。

```
var hosts = VcPlugin.getAllHostSystems(null, "xpath:id[starts-with(.,'1')]");
for each (host in hosts){
    System.log(host.name);
}
```

次のスクリプトは、名前に **DC** という文字列が大文字または小文字で含まれているすべてのデータセンター オブジェクトの名前と ID を返しています。スクリプトには、**tag** プロパティも取得します。

```
var datacenters = VcPlugin.getAllDatacenters(['tag'], "xpath:name[contains(translate(.,
'DC', 'dc'), 'dc')]");
for each (datacenter in datacenters){
    System.log(datacenter.name + " " + datacenter.id);
}
```

例外処理のガイドライン

Mozilla Rhino JavaScript Engine の Orchestrator 実装では、エラーを処理できる例外処理がサポートされています。スクリプトで例外ハンドラを記述するときは、次のガイドラインに従う必要があります。

- 次に示す European Computer Manufacturers Association (ECMA) のエラー タイプを使用します。プラグイン関数が返す一般的な例外として **Error** を使用し、次の特定のエラー タイプを使用します。
 - **TypeError**
 - **RangeError**
 - **EvalError**
 - **ReferenceError**
 - **URIError**
 - **SyntaxError**

次の例は **URIError** の定義を示します。

```
try {
    ...
    throw new URIError("VirtualMachine with ID 'vm-0056'
                        not found on 'vcenter-test-1'");
    ...
} catch ( e if e instanceof URIError ) {
}
}
```

- スクリプトがキャッチしないすべての例外は、**<type>:SPACE<human readable message>** という形式の単純な文字列オブジェクトにする必要があります。次に例を示します。

```
throw "ValidationError: The input parameter 'myParam' of type 'string' is too short."
```

- 人間が判読可能なメッセージはできるだけわかりやすく記載します。
- 単純な文字列の例外タイプのチェックには次のパターンを使用する必要があります。

```
try {
    throw "VMwareNoSpaceLeftOnDatastore: Datastore 'myDatastore' has no space left" ;
} catch ( e if (typeof(e)=="string" && e.indexOf("VMwareNoSpaceLeftOnDatastore:") == 0) )
{
    System.log("No space left on device") ;
    // Do something useful here
}
```

- ワークフロー内のスクリプト化した要素では、単純な文字列の例外タイプのチェックには次のパターンを使用する必要があります。

```
if (typeof(errorCode)=="string"
    && errorCode.indexOf("VMwareNoSpaceLeftOnDatastore:")
    == 0) {
    // Do something useful here
}
```

Orchestrator の JavaScript サンプル

Orchestrator の JavaScript サンプルを切り取り、貼り付け、適用すると、一般的なオーケストレーション タスクの JavaScript の記述に役立ちます。

■ スクリプティングの基本例

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なタスクの基本的なスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ E メール スクリプトの例

ワークフローのスクリプト化された要素に、E メールに関連した一般的なタスクのスクリプトを組み込むことができます。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ ファイル システム スクリプトの例

ワークフローのスクリプト化された要素、アクション、およびポリシーには、一般的なファイル システム タスクのスクリプトが必要です。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ LDAP スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的な LDAP タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ ログ スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なログ タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ ネットワーク スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なネットワーク タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

■ ワークフローのスクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なワークフロー タスクのスクリプティング サンプルが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

スクリプティングの基本例

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なタスクの基本的なスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

XML ドキュメントへのアクセス

次の JavaScript の例を使用すると、Orchestrator JavaScript API 内の XML (E4X) 実装の ECMAScript を使用して、JavaScript から XML ドキュメントにアクセスできます。

注意 Orchestrator は、JavaScript API に E4X を実装しているほか、XML プラグインでドキュメント オブジェクト モデル (DOM) XML 実装も提供しています。XML プラグインとそのサンプル ワークフローの詳細については、vRealize Orchestrator プラグインの使用を参照してください。

```
var people = <people>
    <person id="1">
        <name>Moe</name>
    </person>
    <person id="2">
        <name>Larry</name>
    </person>
</people>;

System.log("'people' = " + people);

// built-in XML type
System.log("'people' is of type : " + typeof(people));

// list-like interface System.log("which contains a list of " +
people.person.length() + " persons");
System.log("whose first element is : " + people.person[0]);

// attribute 'id' is mapped to field '@id'
```

```

people.person[0].@id='47';
// change Moe's id to 47
// also supports search by constraints
System.log("Moe's id is now : " + people.person.(name=='Moe').@id);

// suppress Moe from the list
delete people.person[0];
System.log("Moe is now removed.");

// new (sub-)document can be built from a string
people.person[1] = new XML("<person id=\"3\"><name>James</name></person>");
System.log("Added James to the list, which is now :");
for each(var person in people..person)

for each(var person in people..person){
    System.log("- " + person.name + " (id=" + person.@id + ")");
}

```

ハッシュテーブルでのプロパティの設定と、ハッシュテーブルからのプロパティの取得

次の JavaScript の例は、ハッシュテーブルにプロパティを設定し、そのハッシュテーブルからプロパティを取得します。次の例では、キーは常に文字列であり、値はオブジェクト、数値、ブール値、または文字列です。

```

var table = new Properties() ;
table.put("myKey",new Date()) ;
// get the object back
var myDate= table.get("myKey") ;
System.log("Date is : "+myDate) ;

```

文字列の内容の置換

次の JavaScript の例は、文字列の内容を、新しい内容に置き換えます。

```

var str1 = "'hello'" ;
var reg = new RegExp("'", "g");
var str2 = str1.replace(reg,"\\'") ;
System.log(""+str2) ; // result : \'hello\'

```

タイプの比較

次の JavaScript の例は、あるオブジェクトが特定のオブジェクトタイプに一致するかどうかを確認します。

```

var path = 'myurl/test';
if(typeof(path, string)){
    throw("string");
} else {
    throw("other");
}

```

Orchestrator サーバでのコマンドの実行

次の JavaScript の例を使用すると、Orchestrator サーバ上でコマンドラインを実行できます。サーバの起動に使用する認証情報を使用します。

注意 ファイル システムへのアクセスはデフォルトで制限されています。

```
var cmd = new Command("ls -al") ;
cmd.execute(true) ;
System.log(cmd.output) ;
```

E メール スクリプトの例

ワークフローのスクリプト化した要素に、E メールに関連した一般的なタスクのスクリプトを組み込むことができます。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

メール ワークフローを実行する場合は、ワークフローでは、[メールの設定] ワークフローに設定したデフォルトのメール サーバ構成が使用されます。入力パラメータを使用するか、ワークフローのスクリプト化した要素にカスタム値を定義することによって、デフォルト値をオーバーライドできます。

E メール アドレスの取得

次の JavaScript 例は、実行中のスクリプトの現在の所有者の E メール アドレスを取得します。

```
var emailAddress = Server.getRunningUser().emailAddress ;
```

Eメールの送信

次の JavaScript 例は、定義された内容の E メールを定義された受信者に SMTP サーバ経由で送信します。

```
var message = new EmailMessage() ;
message.smtpHost = "smtpHost" ;
message.subject= "my subject" ;
message.toAddress = "receiver@vmware.com" ;
message.fromAddress = "sender@vmware.com" ;
message.addMimePart("This is a simple message","text/html") ;
message.sendMessage() ;
```

E メール メッセージの取得

次の JavaScript 例は、**MailClient** クラスによって提供されるスクリプト API を使用して、ある E メール アカウントのメッセージを削除することなく取得します。

```
var myMailClient = new MailClient();

myMailClient.setProtocol(mailProtocol);
if(useSSL){
    myMailClient.enableSSL();
}
```

```

myMailClient.connect( mailServer, mailPort, mailUsername, mailPassword);
System.log("Successfully login!");

try {
    myMailClient.openFolder("Inbox");

    var messages = myMailClient.getMessages();
    System.log("Reading messages...!");
    if ( messages != null && messages.length > 0 ) {
        System.log( "You have " + messages.length + " email(s) in your inbox" );
        for ( i = 0; i < messages.length; i++) {
            System.log("");
            System.log("-----MSG-----");
            System.log("Headers: ");
            var headerProp = messages[i].getHeaders();
            for each(key in headerProp.keys){
                System.log(key+": "+headerProp.get(key));
            }
            System.log("");

            System.log( "Message["+ i +"] with from: " + messages[i].from + " to: " +
messages[i].to);
            System.log( "Message["+ i +"] with subject: " + messages[i].subject);
            var content = messages[i].getContent();
            System.log("Msg content as string: " + content);
        }
    } else {
        System.warn( "No messages found" );
    }
} finally {
    myMailClient.closeFolder();
    myMailClient.close();
}

```

ファイル システム スクリプトの例

ワークフローのスクリプト化した要素、アクション、およびポリシーには、一般的なファイル システム タスクのスクリプトが必要です。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

単純なテキスト ファイルへのコンテンツの追加

次の JavaScript の例はテキスト ファイルにコンテンツを追加します。

```

var tempDir = System.getTempDirectory() ;
var fileWriter = new FileWriter(tempDir + "/readme.txt") ;
fileWriter.open() ;
fileWriter.writeLine("File written at : "+new Date()) ;
fileWriter.writeLine("Another line") ;
fileWriter.close() ;

```

ファイルのコンテンツの取得

次の JavaScript の例は、Orchestrator サーバ ホスト マシンからファイルのコンテンツを取得します。

```
var tempDir = System.getTempDirectory() ;
var fileReader = new FileReader(tempDir + "/readme.txt") ;
fileReader.open() ;
var fileContentAsString = fileReader.readAll();
fileReader.close() ;
```

LDAP スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的な LDAP タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

LDAP オブジェクトの Active Directory オブジェクトへの変換

次の JavaScript サンプルでは、LDAP グループ要素を Active Directory ユーザー グループ オブジェクトに、Active Directory ユーザー グループ オブジェクトを LDAP グループ要素に変換します。

```
var ldapGroup ;
// convert from ldap element to Microsoft:UserGroup object
var adGroup = ActiveDirectory.search("UserGroup",ldapGroup.commonName) ;
// convert back to LdapGroup element
var ldapElement = Server.getLdapElement(adGroup.distinguishedName) ;
```

ログ スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なログ タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

パーシステント ログ

次の JavaScript サンプルは、パーシステント ログ エントリを作成します。

```
Server.log("This is a persistant message", "enter a long description here");
Server.warn("This is a persistant warning", "enter a long description here");
Server.error("This is a persistant error", "enter a long description here");
```

非パーシステント ログ

次の JavaScript サンプルは、非パーシステント ログ エントリを作成します。

```
System.log("This is a non-persistant log message");
System.warn("This is a non-persistant log warning");
System.error("This is a non-persistant log error");
```


ネットワーク スクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なネットワーク タスクのスクリプティングが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

URL からのテキストの取得

次の JavaScript サンプルは、URL にアクセスし、テキストを取得し、それを文字列に変換します。

```
var url = new URL("http://www.vmware.com") ;
var htmlContentAsString = url.getContent() ;
```

ワークフローのスクリプティング サンプル

ワークフローのスクリプト化された要素、アクション、およびポリシーでは、一般的なワークフロー タスクのスクリプティング サンプルが必要になります。これらのサンプルは、切り取って、スクリプト化された要素に貼り付けたり、適用したりすることができます。

現在のユーザーによって実行されたワークフローをすべて返す

次の JavaScript サンプルでは、サーバからすべてのワークフロー実行を取得し、それらが現在のユーザーに属しているかをチェックしています。

```
var allTokens = Server.findAllForType('WorkflowToken');
var currentUser = Server.getCredential().username;
var res = [];
for(var i = 0; i<res.length; i++){
    if(allTokens[i].runningUserName == currentUser){
        res.push(allTokens[i]);
    }
}
return res;
```

現在のワークフロー トークンにアクセスする

現在のワークフロー トークンにアクセスするには、**workflow** 変数を使用します。これは、現在のワークフロー実行へのアクセスを可能にする **WorkflowToken** タイプのオブジェクトです。次の JavaScript サンプルでは、ワークフロー トークンの ID と開始日を取得しています。

```
System.log("Current workflow run ID: " + workflow.id);
System.log("Current workflow run start date: "+workflow.startDate);
```

ワークフローをスケジュール設定する

次の JavaScript サンプルでは、特定のプロパティ セットを使用してワークフローを開始し、ワークフローが 1 時間後に開始されるようにスケジュール設定しています。

```
var workflowToLaunch = myWorkflow ;
// create parameters
var workflowParameters = new Properties() ;
workflowParameters.put("name","John Doe") ;
// change the task name
workflowParameters.put("__taskName","Workflow for John Doe") ;

// create scheduling date one hour in the future
var workflowScheduleDate = new Date() ;
var time = workflowScheduleDate.getTime() + (60*60*1000) ;
workflowScheduleDate.setTime(time) ; var scheduledTask =
workflowToLaunch.schedule(workflowParameters,workflowScheduleDate);
```

選択したオブジェクトに対してループでワークフローを実行する

次の JavaScript サンプルでは、仮想マシンの配列を取得し、そのそれぞれに対して **For** ループでワークフローを実行しています。VMs と workflowToRun はワークフローの入力です。

```
var len=VMs.length;
for (var i=0; i < len; i++ )
{
    var VM = VMs[i];
    //var workflowToLaunch = Server.getWorkflowWithId("<workflowId>");
    var workflowToLaunch = workflowToRun;
    if (workflowToLaunch == null) {
        throw "Workflow not found";
    }
    var workflowParameters = new Properties();
    workflowParameters.put("vm",VM);
    var wfToken = workflowToLaunch.execute(workflowParameters);
    System.log ("Ran workflow on " +VM.name);
}
```

アクションの開発

Orchestrator では事前定義アクションのライブラリが提供されます。アクションは、ワークフローおよびスクリプトのビルディング ブロックとして使用する個々の機能を表します。

アクションは JavaScript 機能です。アクションでは複数の入力パラメータを使用でき、戻り値は 1 つになります。これらは Orchestrator API のすべてのオブジェクトのほか、プラグインを使用して Orchestrator にインポートしたすべての API のオブジェクトを呼び出すことができます。

ワークフローを実行すると、アクションはワークフローの属性から入力パラメータを取得します。これらの属性は、ワークフローの初期入力パラメータか、ワークフロー セット内の他の要素が実行時に設定した属性のいずれかになります。

この章では次のトピックについて説明します。

- [アクションの再利用](#)
- [アクション ビューへのアクセス](#)
- [アクション ビューのコンポーネント](#)
- [アクションの作成](#)
- [アクション バージョン履歴の使用](#)
- [削除済みアクションのリストア](#)

アクションの再利用

スクリプト可能タスク ワークフロー要素に個々の機能を直接コーディングする代わりに、アクションとして定義するときには、ライブラリに公開できます。ライブラリにアクションが表示されているときには、他のワークフローはそれを使用できます。

アクションを呼び出すワークフローから個別にアクションを定義するときに、アクションを簡単に更新または最適化できます。個々のアクションを定義すると、他のワークフローもアクションを再利用できます。ワークフローが実行するとき、Orchestrator はワークフローが各アクションを初めて実行するときのみアクションをキャッシュします。Orchestrator は、その後キャッシュされたアクションを再利用できます。アクションをキャッシュすると、ワークフローの再帰呼び出しまたは高速ループで役に立ちます。

アクションを複製し、それらを他のワークフローまたはパッケージにエクスポートするか、アクション階層リストの異なるモジュールに移動することができます。

アクション ビューへのアクセス

Orchestrator クライアント インターフェイスには、Orchestrator サーバのアクション ライブラリにアクセスできる [アクション] ビューがあります。

Orchestrator クライアント インターフェイスの [アクション] ビューには、Orchestrator サーバで使用できるすべてのアクションが階層リスト形式で表示されます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [アクション] ビューをクリックします。
- 3 アクションの階層リストのノードを展開して、アクションのライブラリを参照します。

[アクション] ビューを使用して、ライブラリ内のアクションに関する情報を参照し、アクションの作成や編集を行います。

アクション ビューのコンポーネント

アクション階層リストのアクションをクリックすると、アクションに関する情報が Orchestrator クライアントの右側のペインに表示されます。

[アクション] ビューには 4 つのタブがあります。

全般	アクションの名前、バージョン番号、権限、説明など、アクションについての全般的な情報が表示されます。
スクリプティング	アクションの戻り値のタイプ、入力パラメータ、アクションの機能を定義する JavaScript コードが表示されます。
イベント	このアクションで発生したイベント、またはこのアクションがトリガしたイベントがすべて表示されます。
権限	このアクションにアクセスする権限を持つユーザーおよびユーザー グループが表示されます。

アクションの作成

個々の機能をその他の要素（ワークフローなど）で使用するアクションとして定義できます。アクションは、入力および出力パラメータと権限が定義された JavaScript 関数です。

■ アクションの作成

個々の機能をアクションとして定義する場合は、スクリプト可能タスク ワークフロー要素に直接コーディングする代わりに、他のワークフローでできるようにライブラリとして公開できます。

■ アクションを実装する要素の検出

アクションを編集してその動作を変更すると、そのアクションを実行するワークフローまたはアプリケーションを誤って中断させてしまうことがあります。Orchestrator には、特定の要素を実装するすべてのアクション、ワークフロー、またはパッケージを検出する機能が備わっています。要素の変更が別の要素の操作に影響しないかどうかを検査できます。

■ アクションのコーディング ガイドライン

ワークフローのパフォーマンスを最適化し、アクションを最大限再利用できるようにするためには、アクションを作成するときに、コーディングに関するいくつかの基本的なガイドラインに従う必要があります。

アクションの作成

個々の機能をアクションとして定義する場合は、スクリプト可能タスク ワークフロー要素に直接コーディングする代わりに、他のワークフローで使用できるようにライブラリとして公開できます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [アクション] ビューをクリックします。
- 3 アクション階層リストのルートを展開し、アクションを作成するモジュールに移動します。
- 4 モジュールを右クリックし、[アクションの追加] を選択します。
- 5 テキスト ボックスにアクションの名前を入力し、[OK] をクリックします。
カスタム アクションがアクションのライブラリに追加されます。
- 6 アクションを右クリックして [編集] を選択します。
- 7 [スクリプティング] タブをクリックします。
- 8 デフォルトの戻り値のタイプを変更するには、[Void] リンクをクリックします。
- 9 矢印アイコンをクリックして、アクション入力パラメータを追加します。
- 10 アクション スクリプトを記述します。
- 11 アクション権限を設定します。
- 12 [保存して閉じる] をクリックします。

カスタム アクションを作成し、アクション入力パラメータを追加しました。

次に進む前に

これで、ワークフローで新しいカスタム アクションを使用できるようになりました。

アクションを実装する要素の検出

アクションを編集してその動作を変更すると、そのアクションを実行するワークフローまたはアプリケーションを誤って中断させてしまうことがあります。Orchestrator には、特定の要素を実装するすべてのアクション、ワークフロー、またはパッケージを検出する機能が備わっています。要素の変更が別の要素の操作に影響しないかどうかを検査できます。

重要 [この要素を使用している要素の検索] 機能はすべてのパッケージ、ワークフロー、およびポリシーを検査しますが、スクリプト内は検査しません。したがって、あるアクションを変更することで、[この要素を使用している要素の検索] 機能で識別されないスクリプト内でこのアクションを呼び出す要素に影響する可能性があります。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [アクション] ビューをクリックします。
- 3 アクション階層リストのノードを展開し、特定のアクションに移動します。
- 4 アクションを右クリックし、[この要素を使用している要素の検索] を選択します。
このアクションを実装するワークフローまたはパッケージなどのすべての要素がダイアログ ボックスに表示されます。
- 5 結果のリストから要素をダブルクリックして、その要素を Orchestrator クライアント内で表示します。

アクションを実装するすべての要素が見つかりました。

次に進む前に

この要素の変更が別の要素に影響しないかどうかを検査できます。

アクションのコーディング ガイドライン

ワークフローのパフォーマンスを最適化し、アクションを最大限再利用できるようにするためには、アクションを作成するときに、コーディングに関するいくつかの基本的なガイドラインに従う必要があります。

アクションの基本的なガイドライン

アクションを作成するときは、基本的なガイドラインに従ってください。

- すべてのアクションには、その役割と機能についての説明を含める必要があります。
- 短い、基本的なアクションを記述し、ワークフロー内でアクションを結合します。
- 複数の機能を実行するアクションは記述しないようにします。アクションを再利用できる可能性が限定されるからです。
- 実行時間の長いアクションは記述しないようにします。代わりに、ワークフロー内でループを作成し、アクション要素の後に待機中のイベント要素または待機中のタイマー要素を含めます。
- アクション内にチェック ポイントを記述しないようにします。ワークフローによって、各要素の実行の開始時と終了時にチェック ポイントが設定されます。

- アクション内にループを記述しないようにします。代わりに、ワークフロー内にループを作成します。サーバが再起動した場合、実行中のワークフローは、要素の開始にある、最後のチェック ポイントから再開されます。アクション内にループを記述し、ワークフローがそのアクションを実行しているときにサーバが再起動すると、ワークフローはそのアクションの最初にあるチェック ポイントから再開し、ループが再び最初から開始されます。

アクションの名前の指定に関するガイドライン

アクションに名前を付けるときは、基本的なガイドラインに従ってください。

- アクション名は英語で記述します。
- アクション名は小文字で始めます。名前の中に複数の単語を含める場合は、それぞれの単語を大文字で始めます。たとえば、**myAction** のように指定します。
- アクション名はできるかぎり明確な名前にして、アクションの機能がわかるようにします。たとえば、**backupAllVMsInPool** のように指定します。
- モジュール名はできるかぎり明確な名前にします。
- モジュール名は一意にします。
- モジュール名にはインターネット アドレスを逆にした形式を使用します。たとえば、**com.vmware.myactions.myAction** のように指定します。

アクション パラメータに関するガイドライン

アクション パラメータを定義するときは、基本的なガイドラインに従ってください。

- パラメータ名は英語で記述します。
- パラメータ名は小文字で始めます。
- パラメータ名はできるかぎり明確な名前にします。
- パラメータ名は、できれば 1 語にします。名前に複数の単語を含める必要がある場合は、名前を構成する各単語を大文字で始めます。たとえば、**myParameter** のように指定します。
- オブジェクトの配列を表すパラメータには、複数形を使用します。
- 変数名は明確な名前にします。たとえば、**displayName** のように指定します。
- 各パラメータには説明を含めて、その目的がわかるようにします。
- 1 つのアクション内で大量のパラメータを使用しないでください。

アクション バージョン履歴の使用

バージョン履歴を使用すると、アクションを以前のバージョンに戻すことができます。アクションの状態は、アクション バージョンより前のバージョンに戻すことも、より新しいバージョンに戻すこともできます。現在の状態のアクションと保存されているバージョンのアクションの違いを比較することもできます。

アクション バージョンを増やして保存すると、Orchestrator で各アクションの新しいバージョン履歴項目が作成されます。これ以降にアクションに変更を加えても、現在のバージョン項目は変更されません。たとえば、アクション バージョン 1.0.0 を作成して保存すると、アクションの状態がデータベースに保存されます。アクションに変更を加えると、アクションの状態を Orchestrator クライアントに保存することはできますが、変更をアクション バージョン 1.0.0 に適用することはできません。データベースに変更を保存するには、アクション バージョンを増やして保存する必要があります。バージョン履歴はアクション自体とともにデータベース内に保持されます。

アクションを削除すると、Orchestrator でデータベース内の要素が削除済みとしてマークされますが、要素のバージョン履歴はデータベースから削除されません。この方法により、削除したアクションをリストアすることができます。[「削除済みアクションのリストア」](#)を参照してください。

開始する前に

編集するアクションを開きます。

手順

- 1 アクション エディタの [全般] タブをクリックします。
- 2 [バージョン履歴の表示] をクリックします。
バージョン履歴のウィンドウが表示されます。
- 3 アクション バージョンを選択し、[現在のバージョンとの差分] をクリックして違いを比較します。
現在のアクション バージョンと選択したアクション バージョンの違いがウィンドウに表示されます。
- 4 アクション バージョンを選択し、[元に戻す] をクリックして、アクションの状態をリストアします。

注意 現在のアクション バージョンを保存しなかった場合はバージョン履歴から削除されるため、現在のバージョンに戻せなくなります。

アクションの状態は選択したバージョンの状態に戻ります。

削除済みアクションのリストア

ライブラリから削除したアクションはリストアすることができます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [アクション] ビューをクリックします。
- 3 1 つ以上の削除済みアクションをリストアする先のフォルダに移動します。
- 4 フォルダを右クリックし、[削除済みアクションのリストア] を選択します。
- 5 リストアするアクションを 1 つ以上選択し、[リストア] をクリックします。

1 つ以上のアクションが選択したフォルダに表示されます。

リソース要素の作成

ワークフローを実行する場合、Orchestrator とは別にオブジェクトを作成し、そのオブジェクトを属性として使用しなければならないことがあります。ワークフロー内で外部オブジェクトを属性として使用するには、そのオブジェクトをリソース要素として Orchestrator サーバにインポートする必要があります。

ワークフロー内でリソース要素として使用できるオブジェクトには、イメージ ファイル、スクリプト、XML テンプレート、HTML ファイルなどがあります。Orchestrator サーバ上で実行されるワークフローの場合、Orchestrator にインポートされたすべてのリソース要素を使用することができます。

オブジェクトをリソース要素として Orchestrator にインポートすることにより、そのオブジェクトを 1 か所を変更し、そのリソース要素を使用しているすべてのワークフローに対して変更内容を自動的に反映させることができます。

リソース要素は、フォルダ内で整理して保存することができます。1 つのリソース要素の最大サイズは 16MB です。

この章では次のトピックについて説明します。

- リソース要素の表示
- リソース要素として使用する外部オブジェクトのインポート
- リソース要素の情報とアクセス権限の表示
- リソース要素のファイルへの保存
- リソース要素の更新
- リソース要素をワークフローに追加する

リソース要素の表示

Orchestrator クライアントの既存のリソース要素を表示してコンテンツを確認し、このリソース要素を使用しているワークフローを特定できます。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [リソース] ビューをクリックします。
- 3 階層ツリー ビューを展開し、リソース要素に移動します。
- 4 リソース要素をクリックし、リソース要素に関する情報を右ペインに表示します。
- 5 [ビューア] タブをクリックし、リソース要素のコンテンツを表示します。

- リソース要素を右クリックし、[この要素を使用している要素の検索] を選択します。

このリソース要素を使用しているすべてのワークフローが Orchestrator で一覧表示されます。

次に進む前に

リソース要素をインポートして編集します。

リソース要素として使用する外部オブジェクトのインポート

ワークフローを実行する場合、Orchestrator とは別にオブジェクトを作成し、そのオブジェクトを属性として使用しなければならないことがあります。ワークフロー内で外部オブジェクトを属性として使用するには、そのオブジェクトをリソース要素として Orchestrator サーバにインポートする必要があります。

開始する前に

インポートするイメージ ファイル、スクリプト、XML テンプレート、HTML ファイル、またはその他のタイプのオブジェクトがあることを確認します。

手順

- Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- [リソース] ビューをクリックします。
- 階層リスト内のリソース フォルダまたは root を右クリックし、[新しいフォルダ] を選択して、リソース要素を保存するフォルダを作成します。
- リソース要素をインポートする先のリソース フォルダを右クリックし、[リソースのインポート] を選択します。
- インポートするリソースを選択し、[オープン] をクリックします。

選択したフォルダにリソース要素が追加されます。

これで、Orchestrator サーバにリソース要素がインポートされました。

次に進む前に

リソース要素の一般的な情報を編集し、ユーザーのアクセス権限を設定します。

リソース要素の情報とアクセス権限の表示


オブジェクトをリソース要素として Orchestrator サーバにインポートすると、そのリソース要素の詳細情報と権限を編集できるようになります。

開始する前に

イメージ、スクリプト、XML ファイル、HTML ファイル、またはその他の種類のオブジェクトがリソース要素として Orchestrator にインポートされていることを確認します。

手順

- Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- [リソース] ビューをクリックします。

- 3 リソース要素を右クリックして [編集] を選択します。
- 4 [全般] タブをクリックし、リソース要素の名前、バージョン、説明を入力します。
- 5 [権限] タブをクリックしてから [アクセス権限の追加] アイコン () をクリックして、ユーザー グループの権限を定義します。
- 6 [フィルタ] テキスト ボックスにユーザー グループの名前を入力します。
- 7 ユーザー グループを選択して [OK] をクリックします。
- 8 ユーザー グループを右クリックして [アクセス権限の追加] を選択します。
- 9 適切なチェック ボックスを選択し、このユーザー グループの権限レベルを設定して [OK] をクリックします。

権限は、個別に設定する必要があります。たとえば、特定のユーザーに対して、リソース要素の表示、ワークフロー内でのリソース要素の使用、権限の変更を許可する場合は、該当するチェック ボックスをすべて選択する必要があります。
- 10 [保存して閉じる] をクリックしてエディタを終了します。

これで、リソース要素に関する一般的な情報とユーザーのアクセス権限が設定されました。

次に進む前に

リソース要素をファイルに保存して更新するか、リソース要素をワークフローに追加します。

リソース要素のファイルへの保存

リソース要素はローカル システム上のファイルに保存できます。リソース要素をファイルとして保存すると、リソース要素を編集することができます。

Orchestrator クライアントのリソース要素を編集することはできません。たとえば、リソース要素が XML 構成ファイルやスクリプトである場合、編集するためにはまずローカルに保存する必要があります。

開始する前に

ファイルに保存可能なリソース要素が Orchestrator サーバに含まれていることを確認します。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [リソース] ビューをクリックします。
- 3 リソース要素を右クリックして [ファイルに保存] を選択します。
- 4 必要な変更をファイルに加えます。

リソース要素がファイルに保存されます。

次に進む前に

Orchestrator サーバ内のリソース要素が更新されます。

リソース要素の更新

リソース要素を更新する場合は、リソース要素をファイル システムにエクスポートし、エクスポートしたファイルを適切なツールを使用して編集し、編集したファイルをインポートする必要があります。

開始する前に

イメージ、スクリプト、XML ファイル、HTML ファイル、またはその他の種類のオブジェクトがリソース要素として Orchestrator にインポートされていることを確認します。

手順

- 1 ローカル システムでリソース要素のソース ファイルを変更します。
- 2 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 3 [リソース] ビューをクリックします。
- 4 階層リストで更新したリソース要素に移動します。
- 5 リソース要素を右クリックして [リソースの更新] を選択します。
- 6 (オプション) [ビューア] タブをクリックし、Orchestrator でリソース要素が更新されていることを確認します。

Orchestrator サーバに含まれているリソース要素が更新されます。

リソース要素をワークフローに追加する

リソース要素は、Orchestrator サーバにインポートできる外部オブジェクトです。ワークフローの実行時に、リソース要素を属性として使用することができます。たとえば、マップを定義するための XML ファイルをインポートしてワークフローで使用するにより、特定のタイプのデータを別のタイプのデータに変換したり、実行時に関数を定義するスクリプトに変換したりすることができます。

開始する前に

Orchestrator サーバに次のオブジェクトがあることを確認します。

- リソース要素として Orchestrator にインポートされているイメージ、スクリプト、XML ファイル、HTML ファイル、またはその他の種類のオブジェクト。
- リソース要素を属性として使用する必要があるワークフロー。

手順

- 1 Orchestrator クライアントのドロップダウン メニューから、[設計] を選択します。
- 2 [ワークフロー] ビューをクリックします。
- 3 階層ツリー ビューを展開して、リソース要素を属性として使用する必要があるワークフローに移動します。
- 4 ワークフローを右クリックして [編集] を選択します。
- 5 [属性] ペインの [全般] タブで [属性の追加] アイコン (A+) をクリックします。

- 6 属性名をクリックして、属性に新しい名前を入力します。
- 7 [タイプ] をクリックして属性タイプを設定します。
- 8 [タイプを選択] ダイアログ ボックスで、[フィルタ] ボックスに **resource** と入力してオブジェクト タイプを検索します。

オプション	アクション
単一のリソース要素を属性として定義します	リストから ResourceElement を選択します。
複数のリソース要素を含むフォルダを属性として定義します	リストから ResourceElementCategory を選択します。

- 9 [値] をクリックして、[フィルタ] テキスト ボックスにリソース要素の名前またはリソース要素のカテゴリを入力します。
- 10 リストが表示されたら、リソース要素（またはリソース要素が含まれているフォルダ）を選択して [選択] をクリックします。
- 11 [保存して閉じる] をクリックしてエディタを終了します。

これで、リソース要素（またはリソース要素が含まれているフォルダ）が属性としてワークフローに追加されました。

パッケージの作成

パッケージは、Orchestrator サーバ間でコンテンツを配布するために使用されます。パッケージには、ワークフロー、アクション、ポリシー テンプレート、構成、またはリソースを含めることができます。

パッケージに要素を追加すると、Orchestrator によって依存関係がチェックされ、従属要素がパッケージに追加されます。たとえば、アクションまたは他のワークフローが使用されるワークフローを追加した場合は、Orchestrator によってそのアクションと他のワークフローがパッケージに追加されます。

パッケージをインポートすると、サーバによって、コンテンツのさまざまな要素のバージョンと、対応するローカル要素のバージョンが比較されます。この比較により、ローカル要素とインポートされた要素のバージョンの違いが明確になります。管理者は、パッケージをインポートするか、特定の要素を選択してインポートするかを判断できます。

パッケージはデジタル著作権管理を使用して、受信側サーバでのパッケージのコンテンツの使用方法を制御できます。パッケージは Orchestrator によって署名され、データ保護のために暗号化されます。パッケージは、X509 証明書を使用して、度のユーザーが要素のエクスポートや再配布を行うかを追跡できます。

パッケージの使用の詳細については、『VMware vRealize Orchestrator クライアントの使用』を参照してください。

■ パッケージの作成

ワークフロー、ポリシー テンプレート、アクション、プラグイン リファレンス、リソース、および構成要素をパッケージとしてエクスポートすることができます。パッケージ内の要素が実装するすべての要素は、パッケージに自動的に追加され、バージョン間の互換性が保証されます。参照先の要素を追加しない場合は、パッケージ エディタで削除することができます。

■ パッケージに対するユーザー権限の設定

あるパッケージの内容に対してさまざまなユーザーまたはユーザー グループが持つことのできるアクセス権を制限するために、そのパッケージに対して異なる権限のレベルを設定します。

パッケージの作成

ワークフロー、ポリシー テンプレート、アクション、プラグイン リファレンス、リソース、および構成要素をパッケージとしてエクスポートすることができます。パッケージ内の要素が実装するすべての要素は、パッケージに自動的に追加され、バージョン間の互換性が保証されます。参照先の要素を追加しない場合は、パッケージ エディタで削除することができます。

開始する前に

パッケージに追加することができるワークフロー、アクション、ポリシー テンプレートなどの要素が Orchestrator サーバに含まれることを確認します。

手順

1 Orchestrator クライアントのドロップダウン メニューから、[管理者] を選択します。

2 [パッケージ] ビューをクリックします。

3 左側のペインを右クリックして、[パッケージの追加] を選択します。

4 新しいパッケージの名前を入力し、[OK] をクリックします。

パッケージ名の構文は `<domain.your_company>.<folder>.<package_name>` になります。

たとえば、`com.vmware.myfolder.mypackage` という名前を付けます。

5 パッケージを右クリックして [編集] を選択します。

パッケージ エディタが表示されます。

6 [全般] タブでパッケージの説明を追加します。

7 [ワークフロー] タブでワークフローをパッケージに追加します。

- 選択ダイアログ ボックスでワークフローを検索して選択するには、[ワークフローの挿入 (リスト検索)] をクリックします。
- 階層リストでワークフローのフォルダを参照して選択するには、[ワークフローの挿入 (ツリー参照)] をクリックします。

8 [ポリシー テンプレート]、[アクション]、[構成]、[リソース]、[使用プラグイン] の各タブで、ポリシー テンプレート、アクション、構成要素、リソース要素、プラグインをパッケージに追加します。

9 [保存して閉じる] をクリックしてエディタを終了します。

パッケージを作成し、要素を追加しました。

次に進む前に

このパッケージにユーザー権限を設定します。

パッケージに対するユーザー権限の設定

あるパッケージの内容に対してさまざまなユーザーまたはユーザー グループが持つことのできるアクセス権を制限するために、そのパッケージに対して異なる権限のレベルを設定します。


Orchestrator LDAP または vCenter Single Sign-On サーバ内のユーザーおよびユーザー グループから、権限を設定するさまざまなユーザーおよびユーザー グループを選択できます。Orchestrator では、ユーザーまたはグループに適用できる権限のレベルを定義します。

表示	ユーザーは、パッケージ内の要素を表示できますが、スキーマやスクリプティングは表示できません。
確認	ユーザーは、スキーマやスクリプティングを含むパッケージ内の要素を表示できます。
編集	ユーザーは、パッケージ内の要素を編集できます。
管理	ユーザーは、パッケージ内の要素に対する権限を設定できます。

開始する前に

パッケージを作成し、それをパッケージ エディタで編集のために開いて、そのパッケージに必要な要素を追加します。

手順

- 1 パッケージ エディタの [権限] タブをクリックします。
- 2 [アクセス権限の追加] アイコン () をクリックして、新しいユーザーまたはユーザー グループの権限を定義します。
- 3 ユーザーまたはユーザー グループを検索します。
検索結果には、検索に一致するすべてのユーザーおよびユーザー グループが表示されます。
- 4 ユーザーまたはユーザー グループを選択します。
- 5 適切なチェック ボックスを選択して、このユーザーの権限のレベルを設定し、[選択] をクリックします。
ユーザーが要素を表示したり、スキーマやスクリプティングを検査したり、要素を実行および編集したり、権限を変更したりできるようにするには、すべてのチェック ボックスを選択する必要があります。
- 6 [保存して閉じる] をクリックしてエディタを終了します。

パッケージを作成し、適切なユーザー権限を設定しました。

プラグインの開発

Orchestrator のオープンなプラグイン アーキテクチャにより、Orchestrator を各種の管理ソリューションと統合することができます。プラグイン ワークフローを作成して実行し、プラグイン API にアクセスするには、Orchestrator クライアントを使用します。

この章では次のトピックについて説明します。

- [プラグインの概要](#)
- [プラグインの内容と構造](#)
- [Orchestrator プラグイン API リファレンス](#)
- [vso.xml プラグイン定義ファイルの要素](#)
- [Orchestrator プラグインを開発する場合のベスト プラクティス](#)

プラグインの概要

Orchestrator のプラグインには標準のコンポーネント セットが含まれている必要があり、プラグインは標準のアーキテクチャに従う必要があります。これらのプラクティスは、使用可能な幅広いさまざまな外部のテクノロジーに対してプラグインを作成する際に役立ちます。

- [Orchestrator プラグインの構造](#)
Orchestrator プラグインは、特定の機能を実装する様々なタイプのレイヤで構成される共通の構造を持ちます。
- [外部の API を Orchestrator に公開する](#)
Orchestrator プラグインを作成することにより、外部製品の API を Orchestrator プラットフォームに公開することができます。API を公開する任意のテクノロジーに対してプラグインを作成し、Orchestrator で使用可能な JavaScript オブジェクトにその API をマップすることができます。
- [プラグインのコンポーネント](#)
プラグインは、プラグイン テクノロジーのオブジェクトを Orchestrator プラットフォームに公開する、標準のコンポーネント セットで構成されています。
- [vso.xml ファイルの役割](#)
vso.xml ファイルを使用して、プラグイン テクノロジーのオブジェクト、クラス、メソッド、属性を、Orchestrator のイベントリ オブジェクト、スクリプト タイプ、スクリプト クラス、スクリプト メソッド、属性にマップします。**vso.xml** ファイルは、プラグインの構成動作と起動動作も定義します。

■ プラグイン アダプタのロール

プラグイン アダプタは、Orchestrator サーバへのプラグインのエントリ ポイントです。プラグイン アダプタは、Orchestrator サーバのプラグイン テクノロジー用のデータストアとして機能し、プラグイン ファクトリを作成し、プラグイン テクノロジーで発生するイベントを管理します。

■ プラグイン ファクトリのロール

プラグイン ファクトリは、Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらのオブジェクトに対して操作を実行する方法を定義します。

■ ファインダ オブジェクトの役割

ファインダ オブジェクトは、プラグイン テクノロジーを使用する管理対象オブジェクト タイプの特定のインスタンスを識別、特定します。Orchestrator はファインダ オブジェクトに対してワークフローを実行することにより、プラグイン テクノロジーのオブジェクトを変更、使用できます。

■ スクリプト オブジェクトの役割

スクリプト オブジェクトは、プラグイン テクノロジーのオブジェクトを表す JavaScript 表現です。プラグインのスクリプト オブジェクトは Orchestrator の Javascript API に表示され、ワークフローおよびアクションのスクリプト化された要素内で使用できます。

■ イベント ハンドラの役割

イベントは変更されたプラグイン テクノロジーのオブジェクトの状態または属性で、Orchestrator によって検出されます。Orchestrator はイベント ハンドラを実装してイベントを監視します。

Orchestrator プラグインの構造

Orchestrator プラグインは、特定の機能を実装する様々なタイプのレイヤで構成される共通の構造を持ちます。

Orchestrator プラグインの下から 3 つのレイヤは、インフラストラクチャ クラス、ラッパー クラス、スクリプト オブジェクトであり、プラグイン テクノロジーと Orchestrator 間の接続を実装します。

Orchestrator プラグインの上から 3 つのレイヤはユーザーから見える部分であり、アクション、ビルディング ブロック ワークフロー、ハイレベル ワークフローというレイヤがあります。

図 6-1. Orchestrator プラグインの構造



インフラストラクチャ クラス	プラグイン テクノロジーと Orchestrator 間を接続する一連のクラスです。インフラストラクチャ クラスには、プラグインの定義に従い、プラグイン ファクトリ、プラグイン アダプタなど、実装のためのクラスが含まれます。インフラストラクチャ クラスにはまた、ヘルパー、キャッシュ、インベントリなど、共通のタスクやオブジェクトに機能を備えるためのクラスも含まれています。
ラッパー クラス	プラグイン テクノロジーのオブジェクト モデルを、Orchestrator 内で公開するオブジェクト モデルに取り入れる一連のクラスです。
スクリプト オブジェクト	JavaScript オブジェクト タイプであり、プラグイン テクノロジーのラッパー クラス、メソッド、属性へのアクセスを可能にします。 vso.xml ファイルでは、プラグイン テクノロジーのどのラッパー クラス、属性およびメソッドが Orchestrator に公開されるか定義します。
アクション	ワークフローやスクリプティング タスクで直接使用できる一連の JavaScript の機能です。アクションでは複数の入力パラメータを使用でき、戻り値は1つになります。
ビルディング ブロック ワークフロー	プラグインで実装するすべての一般的な機能をカバーする一連のワークフローです。通常、ビルディング ブロック ワークフローは、統合されたテクノロジーのユーザー インターフェイスの操作を表します。ビルディング ブロック ワークフローは、直接使用する、またはハイレベル ワークフロー内に含めることができます。
ハイレベル ワークフロー	プラグインの特定の機能をカバーする一連のワークフローです。ハイレベル ワークフローにより、具体的な要件を満たしたり、プラグインの複雑な使用例を示したりすることができます。

外部の API を Orchestrator に公開する

Orchestrator プラグインを作成することにより、外部製品の API を Orchestrator プラットフォームに公開することができます。API を公開する任意のテクノロジーに対してプラグインを作成し、Orchestrator で使用可能な JavaScript オブジェクトにその API をマップすることができます。

プラグインにより、Java のオブジェクトとメソッドが JavaScript オブジェクトにマップされ、Orchestrator のスクリプト API に追加されます。外部のテクノロジーを使用して Java API を公開する場合は、その API を Orchestrator の JavaScript に直接マップし、ワークフローやアクションで使用することができます。

公開した API を、WSDL (Web Service Definition Language)、REST (Representational State Transfer)、またはメッセージング サービスを使用して Java オブジェクトに統合することにより、Java 以外の言語で API を公開するアプリケーションのプラグインを作成することができます。その後、統合された Java オブジェクトを Orchestrator の JavaScript にマップして使用することができます。

プラグインを使用するテクノロジーは、Orchestrator からは独立して機能します。ソース コードにアクセスできず、Java アーカイブ (JAR ファイル) などのバイナリ コードにしかアクセスできない場合であっても、外部製品用の Orchestrator プラグインを作成することができます。

プラグインのコンポーネント

プラグインは、プラグイン テクノロジーのオブジェクトを Orchestrator プラットフォームに公開する、標準のコンポーネント セットで構成されています。

プラグインの主要なコンポーネントは、プラグイン アダプタ、ファクトリ、およびイベント実装です。アダプタ、ファクトリ、およびイベント実装で定義されたオブジェクトおよび操作は、**vso.xml** という名前の XML 定義ファイル内の Orchestrator オブジェクトにマップします。**vso.xml** ファイルは、プラグイン テクノロジーのオブジェクトおよび機能を、Orchestrator JavaScript API に示される JavaScript スクリプト オブジェクトにマップします。また、**vso.xml** ファイルは、プラグイン テクノロジーのオブジェクト タイプを Orchestrator の [インベントリ] タブに表示されるファインディングにマップします。

プラグインは次のコンポーネントで構成されています。

プラグイン モジュール	Java クラスのセットで定義されるプラグイン自体、 vso.xml ファイル、およびプラグインを介してアクセスするオブジェクトと通信するワークフローとアクションのパッケージ。プラグイン モジュールは必須になります。
プラグイン アダプタ	プラグイン テクノロジーと Orchestrator サーバ間のインターフェイスを定義します。アダプタは、Orchestrator プラットフォームへのプラグインのエントリ ポイントです。アダプタは、プラグイン ファクトリを作成し、プラグインのロードおよびアンロードを管理し、プラグイン テクノロジーのオブジェクトで発生するイベントを管理します。プラグイン アダプタは必須になります。
プラグイン ファクトリ	Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらに対して操作を実行する方法を定義します。アダプタにより、Orchestrator とプラグイン テクノロジーとの間で開かれるクライアント セッション用のファクトリが作成されます。ファクトリでは、すべてのクライアント接続間で 1 つのセッションを共有するか、クライアント接続ごとに 1 つのセッションを開くことができます。プラグイン ファクトリは必須になります。

構成	Orchestrator では、プラグインの構成を保存する標準的な方法は定義されていません。構成情報は、Windows レジストリや静的構成ファイルを使用する、またはデータベースや XML ファイルに情報を保存することによって保存できます。Orchestrator プラグインは、Orchestrator クライアントで構成ワークフローを実行して構成できます。
ファインダ	Orchestrator がプラグイン テクノロジーのオブジェクトを配置または表示する方法を定義する操作ルール。ファインダは、プラグイン テクノロジーが Orchestrator に公開しているオブジェクトのセットからオブジェクトを取得します。開発者は、オブジェクトのネットワークを経由して移動できるようにするため、オブジェクト間の関係を vso.xml ファイルに定義します。Orchestrator は、プラグイン テクノロジーのオブジェクト モデルを [インベントリ] タブに表示します。プラグイン テクノロジーのオブジェクトを Orchestrator に公開する場合には、ファインダは必須になります。
スクリプト オブジェクト	プラグイン テクノロジーのオブジェクト、操作、および属性へのアクセスを提供する JavaScript オブジェクト タイプ。スクリプト オブジェクトは、Orchestrator が JavaScript を介してプラグイン テクノロジーのオブジェクト モデルにアクセスする方法を定義します。開発者は、プラグイン テクノロジーのクラスおよびメソッドを vso.xml ファイルの JavaScript オブジェクトにマップします。Orchestrator のスクリプト API にある JavaScript オブジェクトにアクセスし、それらを Orchestrator のスクリプト化されたタスク、アクション、およびワークフローに統合できます。スクリプト オブジェクトは、スクリプト タイプ、クラス、およびメソッドを Orchestrator JavaScript API に追加する場合には、必須になります。
インベントリ	Orchestrator が Orchestrator クライアントの [インベントリ] ビューに表示されたファインダを使用して配置する、プラグイン テクノロジーにあるオブジェクトのインスタンス。インベントリのオブジェクトに対しては、それらに対してワークフローを実行することで、操作を実行できます。インベントリはオプションです。開発者は、Orchestrator JavaScript API にスクリプト タイプおよびクラスを追加するのみで、インベントリ内のオブジェクトのインスタンスを公開しないプラグインを作成できます。
イベント	プラグイン テクノロジーにあるオブジェクトの状態の変化。Orchestrator は、プラグイン テクノロジーで発生するイベントを受動的にリスニングできます。また、Orchestrator はプラグイン テクノロジーのイベントをアクティブにトリガすることもできます。イベントはオプションです。

vso.xml ファイルの役割

vso.xml ファイルを使用して、プラグイン テクノロジーのオブジェクト、クラス、メソッド、属性を、Orchestrator のインベントリ オブジェクト、スクリプト タイプ、スクリプト クラス、スクリプト メソッド、属性にマップします。**vso.xml** ファイルは、プラグインの構成動作と起動動作も定義します。

vso.xml ファイルは以下の主要な役割を実行します。

起動および構成の動作	プラグインを起動する方法と、プラグインが定義する構成の実装を特定する方法を定義します。プラグイン アダプタをロードします。
インベントリ オブジェクト	プラグイン テクノロジーを使用してプラグインがアクセスするオブジェクトのタイプを定義します。プラグイン ファクトリの実装のファインダ メソッドは、これらのオブジェクトのインスタンスを特定し、Orchestrator インベントリに表示します。
スクリプト タイプ	Orchestrator の JavaScript API にスクリプト タイプを追加して、インベントリ内のオブジェクトの異なるタイプを表します。これらのスクリプト タイプはワークフローで入力パラメータとして使用できます。
スクリプト クラス	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能なクラスを、Orchestrator の JavaScript API に追加します。
スクリプト メソッド	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能なメソッドを、Orchestrator の JavaScript API に追加します。
スクリプト属性	ワークフロー、アクション、ポリシーなどのスクリプト化された要素で使用可能な、プラグイン テクノロジーを使用するオブジェクトの属性を、Orchestrator の JavaScript API に追加します。

プラグイン アダプタのロール

プラグイン アダプタは、Orchestrator サーバへのプラグインのエントリ ポイントです。プラグイン アダプタは、Orchestrator サーバのプラグイン テクノロジー用のデータストアとして機能し、プラグイン ファクトリを作成し、プラグイン テクノロジーで発生するイベントを管理します。

プラグイン アダプタを作成するには、**IPluginAdaptor** インターフェイスを実装する Java クラスを作成します。

作成するプラグイン アダプタ クラスは、プラグイン テクノロジーのプラグイン ファクトリ、イベント、およびトリガを管理します。**IPluginAdaptor** インターフェイスには、これらのタスクの実行に使用するメソッドが用意されています。

プラグイン アダプタは次の主要なロールを実行します。

ファクトリの作成	プラグイン アダプタの最も重要なロールは、各接続用の 1 つのプラグイン ファクトリ インスタンスを Orchestrator からプラグイン テクノロジーにロードまたはアンロードすることです。プラグイン アダプタ クラスは IPluginAdaptor.createPluginFactory() メソッドを呼び出して、 IPluginFactory インターフェイスを実装するクラスのインスタンスを作成します。
イベントの管理	プラグイン アダプタは、Orchestrator サーバとプラグイン テクノロジーとの間のインターフェイスです。プラグイン アダプタは、Orchestrator がプラグイン テクノロジーのオブジェクトに対して実行またはウォッチするイベントを管理します。このアダプタは、イベント パブリッシャを介してイベントを管理します。イベント パブリッシャは、アダプタが

IPluginAdaptor.registerEventPublisher() メソッドを呼び出すことによって作成する **IPluginEventPublisher** インターフェイスのインスタンスです。イベント パブリッシャは、プラグイン テクノロジーのオブジェクトに対してトリガおよびゲージを設定し、オブジェクトで特定のイベントが発生した場合、またはオブジェクトの値が特定のしきい値を超えた場合に、Orchestrator が定義済みアクションを起動できるようにします。同様に、長期実行ワークフローの待機イベント要素が待機するイベントを定義する **PluginTrigger** および **PluginWatcher** インスタンスを定義できます。

プラグイン名の設定

プラグインの名前は、**vso.xml** ファイル内で指定します。プラグイン アダプタはこの名前を **vso.xml** ファイルから取得し、Orchestrator クライアントの [インベントリ] ビューに公開します。

ライセンスのインストール

アダプタ実装においてプラグイン テクノロジーが必要とするライセンス ファイルをインストールするメソッドを呼び出すことができます。

IPluginAdaptor インターフェイス、そのすべてのメソッド、およびプラグイン API の他のすべてのクラスについての詳細は、[「Orchestrator プラグイン API リファレンス」](#) を参照してください。

プラグイン ファクトリのロール

プラグイン ファクトリは、Orchestrator がプラグイン テクノロジーのオブジェクトを検索し、それらのオブジェクトに対して操作を実行する方法を定義します。

プラグイン ファクトリを作成するには、Orchestrator プラグイン API から **IPluginFactory** インターフェイスを実装および拡張する必要があります。作成したプラグイン ファクトリ クラスは、Orchestrator がプラグイン テクノロジーのオブジェクトにアクセスするために使用するファインダ機能を定義します。ファクトリを使用すると、Orchestrator サーバは ID によって、他のオブジェクトに対する関係によって、またはクエリ文字列を検索することによって、オブジェクトを検索できます。

プラグイン ファクトリは次の主要なロールを実行します。

オブジェクトの検索

オブジェクトの名前およびタイプに従ってオブジェクトを検索する機能を作成できます。**IPluginFactory.find()** メソッドを使用してオブジェクトを名前およびタイプで検索します。

他のオブジェクトに関連するオブジェクトの検索

指定されたオブジェクトに関連するオブジェクトを、指定された関係のタイプによって検索する機能を作成できます。関係は、**vso.xml** ファイルに定義します。また、すべての親に関連する依存子オブジェクトを、指定された関係のタイプによって検索するファインダを作成できます。**IPluginFactory.findRelation()** メソッドを実装して、指定された親オブジェクトに関係するオブジェクトを、指定さ

れた関係のタイプによって検索します。

IPluginFactory.hasChildrenInRelation() メソッドを実装して、親インスタンスに対して少なくとも 1 つの子オブジェクトが存在するかどうかを調べます。

独自の基準に基づいてオブジェクトを検索するクエリを定義

自分で定義したクエリ ルールを実装するオブジェクト ファインダを作成できます。**IPluginFactory.findAll()** メソッドを実装して、ファクトリがこのメソッドを呼び出したときに、自分で定義したクエリ ルールを満たすすべてのオブジェクトを検索します。自分で定義したクエリ ルールに一致するすべてのオブジェクトのリストを含む **QueryResult** オブジェクトの **findAll()** メソッドの結果を取得します。

IPluginFactory インターフェイス、そのすべてのメソッド、およびプラグイン API の他のすべてのクラスについての詳細は、[「Orchestrator プラグイン API リファレンス」](#) を参照してください。

ファインダ オブジェクトの役割

ファインダ オブジェクトは、プラグイン テクノロジーを使用する管理対象オブジェクト タイプの特定のインスタンスを識別、特定します。Orchestrator はファインダ オブジェクトに対してワークフローを実行することにより、プラグイン テクノロジーのオブジェクトを変更、使用できます。

プラグイン テクノロジーを使用する特定の管理対象オブジェクト タイプのすべてのインスタンスには、Orchestrator のファインダ オブジェクトで見つけられるように一意の識別子を付ける必要があります。プラグイン テクノロジーでは、オブジェクト インスタンスの一意の識別子が文字列として提供されます。ワークフローの実行中、Orchestrator は見つかったオブジェクトの一意の識別子をワークフローの属性値として設定します。特定のタイプのオブジェクトを入力パラメータとして使用するワークフローは、そのタイプのオブジェクトの特定のインスタンスに対して実行されます。

プラグインにより Orchestrator の JavaScript API に追加されるファインダ オブジェクトには、プラグイン名がプリフィックスとして付きます。たとえば、vCenter Server API の **VirtualMachine** 管理対象オブジェクト タイプは、Orchestrator で **VC:VirtualMachine** JavaScript タイプとして表示されます。

たとえば、Orchestrator は特定の **VC:VirtualMachine** インスタンスにアクセスする場合、vCenter Server プラグインを使用し、仮想マシンの **id** 属性を一意の識別子として使用するファインダ オブジェクトを実装します。このオブジェクト インスタンスは属性値としてワークフロー要素に渡すことができます。

Orchestrator プラグインは、プラグイン テクノロジーのオブジェクトを、**vso.xml** ファイル内の **<finder>** 要素にある同等の Orchestrator ファインダ オブジェクトにマップします。**<finder>** 要素は、プラグイン テクノロジーのメソッドまたは関数を特定します。プラグイン テクノロジーでは、オブジェクトの特定のインスタンスに使用する一意の識別子が取得されます。また、**<finder>** 要素はオブジェクト間の関係も定義し、他のオブジェクトとの関係に基づいてオブジェクトを見つけます。

Orchestrator の [インベントリ] タブには、ファインダ オブジェクトが含まれているプラグインの下にファインダ オブジェクトが表示されます。

スクリプト オブジェクトの役割

スクリプト オブジェクトは、プラグイン テクノロジーのオブジェクトを表す JavaScript 表現です。プラグインのスクリプト オブジェクトは Orchestrator の JavaScript API に表示され、ワークフローおよびアクションのスクリプト化された要素内で使用できます。

プラグインのスクリプト オブジェクトは JavaScript のモジュール、タイプ、クラスとして Orchestrator の JavaScript API に表示されます。ほとんどのファインダ オブジェクトにはスクリプト オブジェクト表現があります。JavaScript クラスはメソッドと属性を Orchestrator の JavaScript API に追加できます。Orchestrator の JavaScript API は、プラグイン テクノロジーの API にあるオブジェクトのメソッドと属性を表示します。プラグイン テクノロジーは、Orchestrator とは別にオブジェクト、タイプ、クラス、属性、メソッドの実装を提供します。たとえば、vCenter Server プラグインは vCenter Server API のすべてのオブジェクトを Orchestrator の JavaScript API の JavaScript オブジェクトとして表示します。vCenter Server API が定義するすべてのクラス、メソッド、属性は JavaScript 表現で表示されます。vCenter Server のスクリプト クラス、およびスクリプト クラスが Orchestrator のスクリプト 関数で定義するメソッドと属性は使用することができます。

たとえば、**VC:VirtualMachine** ファインダによって vCenter Server API で **VirtualMachine** 管理対象オブジェクト タイプが見つかったと、Orchestrator の JavaScript API に **VcVirtualMachine** JavaScript クラスとして表示されます。Orchestrator の JavaScript API 内の **VcVirtualMachine** JavaScript クラスは、vCenter Server API 内のすべての同じメソッドと属性を **VirtualMachine** 管理対象オブジェクトとして定義します。

Orchestrator プラグインは、プラグイン テクノロジーのオブジェクト、タイプ、クラス、属性、メソッドを、**vso.xml** ファイル内の **<scripting-objects>** 要素にある同等の Orchestrator JavaScript のオブジェクト、タイプ、クラス、属性、メソッドにマップします。

イベント ハンドラの役割

イベントは変更されたプラグイン テクノロジーのオブジェクトの状態または属性で、Orchestrator によって検出されます。Orchestrator はイベント ハンドラを実装してイベントを監視します。

Orchestrator プラグインを使用すると、プラグイン テクノロジーのイベントを異なる方法で監視できます。Orchestrator プラグイン API を使用すると、以下のタイプのイベント ハンドラを作成して、プラグイン テクノロジーのイベントを監視できます。

リスナー	プラグイン テクノロジーのオブジェクトの状態に変更がないか受動的に監視します。プラグイン テクノロジーまたはプラグインの実装では、リスナーが監視するイベントが定義されます。リスナーはイベントを開始しませんが、イベントが発生すると Orchestrator に通知します。リスナーはプラグイン テクノロジーをポーリン
-------------	---

グするか、プラグイン テクノロジーから通知を受信することにより、イベントを検出します。イベントが発生すると、イベントを待機している Orchestrator のポリシーまたはワークフローは、Orchestrator サーバで操作を開始して応答することができます。リスナーコンポーネントの設定はオプションです。

ポリシー

プラグイン テクノロジーの特定のイベントを監視し、イベントが発生した場合に Orchestrator サーバで操作を開始します。ポリシーではポリシー トリガとポリシー ゲージを監視できます。ポリシー トリガはプラグイン テクノロジーのイベントを定義します。イベントが発生すると、実行中のポリシーによって Orchestrator サーバで操作（ワークフローの実行など）が開始されます。ポリシー ゲージはプラグイン テクノロジーのオブジェクトの属性値の範囲を定義します。この範囲を超過すると、Orchestrator は操作を開始します。ポリシーの設定はオプションです。

ワークフロー トリガ

実行中のワークフローに待機イベント要素が含まれていてその要素に達した場合、実行がサスペンドして、プラグイン テクノロジーのイベントが発生するまで待機状態になります。ワークフロー トリガでプラグイン テクノロジーのイベントを定義して、ワークフローの待機要素を待機させます。ワークフロー トリガはウォッチャーに登録して使用します。ワークフロー トリガの設定はオプションです。

ウォッチャー

「ウォッチ」ワークフローはワークフローの待機イベント要素の代わりに、プラグイン テクノロジーの特定のイベントをトリガします。ウォッチャーはイベントが発生するとそのイベントを待機しているワークフローに通知します。ウォッチャーの設定はオプションです。

プラグインの内容と構造

Orchestrator のプラグインには、標準のコンポーネント セットが含まれている必要があります。また、標準のファイル構造に準拠している必要があります。プラグインを標準のファイル構造に準拠させるには、特定のフォルダとファイルを含める必要があります。

Orchestrator プラグインを作成するには、そのプラグインを使用するテクノロジー内のオブジェクトに対するアクセス方法と対話方法を定義する必要があります。また、プラグインを使用するテクノロジーのすべてのオブジェクトと関数を、**vso.xml** ファイル内の対応する Orchestrator のオブジェクトと関数にマップする必要があります。

vso.xml ファイルには、Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照が含まれている必要があります。プラグインを使用するテクノロジー内でプラグインによって検出されるすべてのオブジェクトに対して、一意の ID を指定する必要があります。**vso.xml** ファイルの **finder** 要素とオブジェクト要素で、オブジェクト名を定義します。

プラグインは、標準の Java アーカイブ ファイル (JAR) として提供することも、ZIP ファイルとして提供することもできますが、いずれの場合も、ファイル名に **.dar** という拡張子を付加する必要があります。

注意 Orchestrator コントロール センターを使用して、DAR ファイルを Orchestrator サーバにインポートすることができます。

■ vso.xml ファイル内でのアプリケーション マッピングの定義

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

■ vso.xml プラグイン定義ファイルの形式

vso.xml ファイルは、Orchestrator サーバがプラグイン テクノロジーと連携する方法を定義します。Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照を **vso.xml** ファイルに含める必要があります。

■ プラグイン オブジェクトの命名

プラグインがプラグイン テクノロジーにおいて検出する各オブジェクトに対し、一意の識別子を与える必要があります。オブジェクト名は、**vso.xml** ファイルの **<finder>** 要素および **<object>** 要素に定義します。

■ プラグイン オブジェクトの命名規則

プラグイン内のすべてのオブジェクトに名前を付ける場合は、Java クラスの命名規則に従ってください。

■ プラグインのファイル構造

プラグインは、標準のファイル構造に準拠するようにし、特定のフォルダとファイルを含める必要があります。プラグインは、標準の Java アーカイブ ファイル (JAR) または ZIP ファイルとして提供します。ファイル名には **.dar** という拡張子を付加する必要があります。

vso.xml ファイル内でのアプリケーション マッピングの定義

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

vso.xml ファイルは以下の情報を Orchestrator サーバに提供します。

- プラグインのバージョン、名前、および説明
- プラグイン テクノロジーのクラスおよび関連するプラグイン アダプタへのリファレンス
- Orchestrator サーバが起動するときにプラグインを初期化します
- プラグイン テクノロジー内のオブジェクトのタイプを表すスクリプト タイプ
- オブジェクトが Orchestrator インベントリ内で表示される方法を定義するオブジェクト タイプ間の関係
- プラグイン テクノロジー内のオブジェクトおよび操作を Orchestrator JavaScript API の機能およびオブジェクト タイプにマップするスクリプト クラス
- 特定のタイプのすべてのオブジェクトに適用される定数値のリストを定義する列挙
- Orchestrator がプラグイン テクノロジー内で監視するイベント

vso.xml ファイルは Orchestrator プラグインの XML スキーマ定義に適合する必要があります。スキーマ定義には VMware サポート サイトからアクセスできます。

```
http://www.vmware.com/support/orchestrator/plugin-4-1.xsd
```

vso.xml ファイルのすべての要素についての説明は、[\[vso.xml プラグイン定義ファイルの要素\]](#) を参照してください。

vso.xml プラグイン定義ファイルの形式

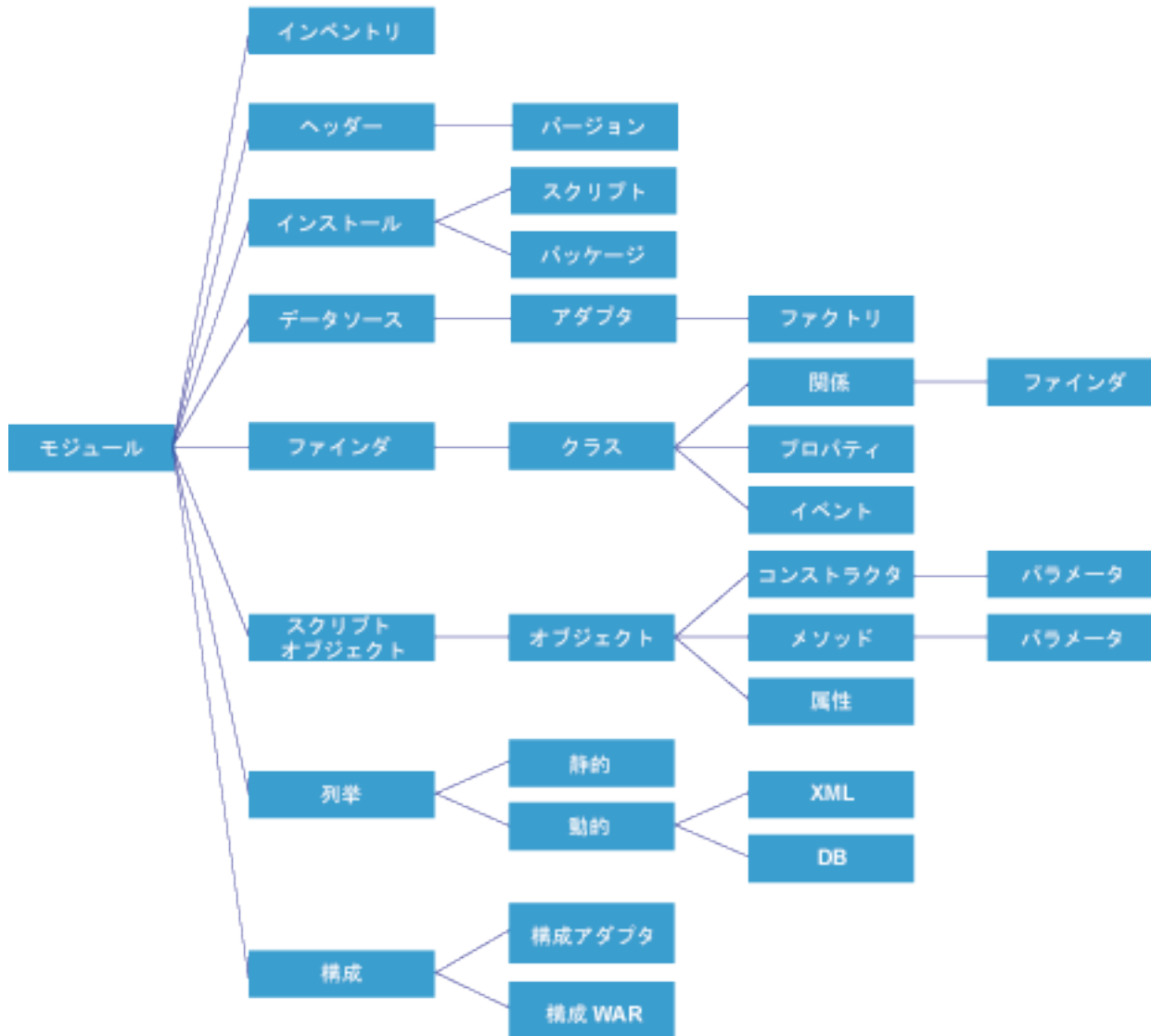
vso.xml ファイルは、Orchestrator サーバがプラグイン テクノロジーと連携する方法を定義します。Orchestrator に公開されるすべてのタイプのオブジェクトまたは操作に対する参照を **vso.xml** ファイルに含める必要があります。

vso.xml ファイルに含めたオブジェクトは、Orchestrator スクリプト API のスクリプト オブジェクトか、Orchestrator の [インベントリ] タブのファインダ オブジェクトとして表示されます。

プラグインのオープン アーキテクチャおよび標準化された実装の一部として、**vso.xml** ファイルは標準形式に従う必要があります。

次の図は、**vso.xml** プラグイン定義ファイルの形式と、要素が互いにどのようにネストされているかを示しています。

図 6-2. vso.xml プラグイン定義ファイルの形式



プラグイン オブジェクトの命名

プラグインがプラグイン テクノロジーにおいて検出する各オブジェクトに対し、一意の識別子を与える必要があります。オブジェクト名は、**vso.xml** ファイルの **<finder>** 要素および **<object>** 要素に定義します。

ファクトリ実装で定義するファインダ操作では、プラグインテクノロジーのオブジェクトを検索します。プラグインがオブジェクトを検索するとき、Orchestrator のワークフローでそれらを使用して、それらを 1 つのワークフロー要素から別の要素へ渡すことができます。オブジェクトに一意の識別子を与えると、ワークフローにおいて要素間でのそれらの引き渡しが可能になります。

Orchestrator サーバは、サーバが処理する各オブジェクトのタイプおよび識別子のみを保存し、Orchestrator がオブジェクトを取得した場所または方法についての情報は保存しません。プラグインから取得したオブジェクトを追跡できるように、プラグインの実装においてオブジェクトを一貫性を持って命名する必要があります。

ワークフローの実行中に Orchestrator サーバが停止した場合、サーバの再起動時、ワークフローはサーバが停止したときに実行していたワークフロー要素から再開します。ワークフローは、サーバが停止したときに要素が処理していたオブジェクトを取得するために、識別子を使用します。

プラグイン オブジェクトの命名規則

プラグイン内のすべてのオブジェクトに名前を付ける場合は、Java クラスの命名規則に従ってください。

重要 ワークフロー エンジンによるデータのシリアル化の実行手段に基づいて、オブジェクト名には以下の文字列を使用しないでください。これらの文字列をオブジェクト識別子に使用すると、ワークフロー エンジンによるワークフローの解析が不正確になり、ワークフローを実行したときに予期せぬ動作が発生する可能性があります。

- `#;`
- `#,`
- `#=`

オブジェクトのプラグインに名前を付ける場合は、これらのガイドラインを使用してください。

- 名前の各単語には大文字の頭文字を使用してください。
- 単語の区切りにスペースを使用しないでください。
- 文字は、**A** から **Z** および **a** から **z** の標準文字のみを使用してください。
- 特殊文字（アクセントなど）を使用しないでください。
- 名前の最初の文字に数字を使用しないでください。
- できるだけ 10 文字未満にしてください。

オブジェクト タイプごとの規則は、[表 6-1](#) に記載されています。

表 6-1. プラグイン オブジェクトの命名規則

オブジェクト タイプ	命名規則
プラグイン	<ul style="list-style-type: none"> ■ vso.xml ファイルの <module> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ 一意のものにしてください。Orchestrator サーバで同じ名前の 2 つのプラグインを実行することはできません。
ファインダ オブジェクト	<ul style="list-style-type: none"> ■ vso.xml ファイルの <finder> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ プラグイン内で一意のものにしてください。 <p>Orchestrator は、Orchestrator のスクリプト API のファインダ オブジェクト タイプのオブジェクト名にプラグイン名とコロンを追加します。たとえば、vCenter Server プラグインの VirtualMachine オブジェクトタイプは、Orchestrator のスクリプト API では VC:VirtualMachine と表示されます。</p>
オブジェクトのスクリプティング	<ul style="list-style-type: none"> ■ vso.xml ファイルの <scripting-object> 要素で定義されます。 ■ Java クラスの命名規則に従ってください。 ■ Orchestrator サーバ内で一意のものにしてください。 ■ スクリプト オブジェクトには、同じ名前のファインダ オブジェクトやその他のオブジェクトのスクリプト オブジェクトとの混乱を避けるために、スクリプト オブジェクト名の前に常にプラグインの名前が付きますが、コロンは付きません。たとえば、vCenter Server プラグインの VirtualMachine クラスには、Orchestrator のスクリプト API が VcVirtualMachine クラスとして表示されます。

プラグインのファイル構造

プラグインは、標準のファイル構造に準拠するようにし、特定のフォルダとファイルを含める必要があります。プラグインは、標準の Java アーカイブ ファイル (JAR) または ZIP ファイルとして提供します。ファイル名には **.dar** という拡張子を付加する必要があります。

DAR アーカイブのコンテンツでは、次のフォルダ構造と命名規則を使用する必要があります。

表 6-2. DAR アーカイブの構造

フォルダ	説明
<plug-in_name>\VSO-INF\	プラグイン テクノロジーのオブジェクトの Orchestrator オブジェクトへのマッピングを定義する vso.xml ファイルを格納します。 VSO-INF フォルダと vso.xml ファイルは必須です。
<plug-in_name>\lib\	プラグイン テクノロジーのバイナリが含まれる JAR ファイルを格納します。また、アダプタ、ファクトリ、通知ハンドラ、およびその他のインターフェイスの実装がプラグインに含まれる JAR ファイルも格納します。 lib フォルダと JAR ファイルは必須です。
<plug-in_name>\resources\	プラグインに必要なリソース ファイルを格納します。 resources フォルダには次の要素タイプを格納できます。 <ul style="list-style-type: none"> ■ イメージファイル。Orchestrator の [インベントリ] タブでプラグインのオブジェクトを表します。 ■ スクリプト。プラグイン起動時の初期化動作を定義します。 ■ Orchestrator パッケージ。プラグインを使用してアクセスするオブジェクトと対話するカスタム ワークフロー、アクション、および他のリソースを含めることができます。 リソースは、サブフォルダを使用して整理できます（例： resources\images\ 、 resources\scripts\ 、 resources\packages\ ）。 resources フォルダの使用はオプションです。

Orchestrator コントロール センターを使用して、DAR ファイルを Orchestrator サーバにインポートします。

Orchestrator プラグイン API リファレンス

Orchestrator プラグイン API は、プラグインを作成するために **IPluginAdaptor** および **IPluginFactory** の実装を開発する際に、実装および拡張する Java インターフェイスおよびクラスを定義します。

すべてのクラスは、記載がないかぎり **ch.dunes.vso.sdk.api** パッケージに含まれます。

IAop インターフェイス

IAop インターフェイスは、プラグインを使用するテクノロジーでオブジェクトのプロパティを取得および設定する方法を提供します。

```
public interface IAop
```

IAop インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)</code>	<code>java.lang.Object</code>	プラグインで指定のオブジェクトからプロパティを取得します。
<code>set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)</code>	<code>Void</code>	プラグインで指定のオブジェクトにプロパティを設定します。

IDynamicFinder インターフェイス

IDynamicFinder インターフェイスは、ファインダの ID とプロパティを **vso.xml** ファイルに定義する代わりに、ID とプロパティをプログラムで戻します。

IDynamicFinder インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getIdAccessor(java.lang.String type)</code>	<code>java.lang.String</code>	オブジェクト ID をプログラムで取得するための OGNL 式を提供します。
<code>getProperties(java.lang.String type)</code>	<code>java.util.List<SDKFinderProperty></code>	オブジェクト プロパティのリストをプログラムで提供します。

IPluginAdaptor インターフェイス

IPluginAdaptor インターフェイスは、プラグイン ファクトリ、イベント、およびウォッチャーを管理するために実装します。**IPluginAdaptor** インターフェイスはプラグインと Orchestrator サーバの間のアダプタを定義します。

IPluginAdaptor インスタンスはセッション管理の役割を担っています。**IPluginAdaptor** インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>addWatcher(PluginWatcher watcher)</code>	Void	特定のイベントを監視するウォッチャーを追加します
<code>createPluginFactory(java.lang.String sessionId, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)</code>	IPluginFactory	<p>IPluginFactory インスタンスを作成します。Orchestrator サーバはファクトリを使用して、オブジェクト ID や他のオブジェクトとの関係などを基準にしてプラグイン テクノロジーからオブジェクトを取得します。</p> <p>セッション ID によって実行中のセッションを特定できます。たとえば、ユーザーは 2 つの異なる Orchestrator クライアントにログインして 2 つのセッションを同時に実行できます。</p> <p>同様に、ワークフローを開始すると、ワークフローの開始に使用したクライアントとは無関係のセッションが作成されます。Orchestrator クライアントを閉じた場合でもワークフローは実行し続けます。</p>
<code>installLicenses(PluginLicense[] licenses)</code>	Void	VMware が提供する標準プラグインのライセンス情報をインストールします
<code>registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	インベントリ内の要素にトリガとゲージを設定します
<code>removeWatcher(java.lang.String watcherId)</code>	Void	ウォッチャーを削除します
<code>setPluginName(java.lang.String pluginName)</code>	Void	vso.xml ファイルからプラグイン名を取得します
<code>setPluginPublisher(IPluginPublisher pluginPublisher)</code>	Void	プラグインの公開者を設定します
<code>uninstallPluginFactory(IPluginFactory plugin)</code>	Void	プラグインファクトリをアンインストールします。
<code>unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)</code>	Void	インベントリ内の要素からトリガとゲージを除去します

IPluginEventPublisher インターフェイス

IPluginEventPublisher インターフェイスは、Orchestrator ポリシーを監視するためのイベント通知バスにゲージとトリガを公開します。

IPluginEventPublisher インスタンスはプラグイン アダプタ実装に直接作成したり、別々のイベント ジェネレータ クラスに作成したりできます。

IPluginEventPublisher インターフェイスを実装して、Orchestrator ポリシー エンジンにプラグイン テクノロジーのイベントを公開することができます。プラグイン テクノロジーのオブジェクト内のポリシー トリガとゲージを設定する方法や、それらのオブジェクトのイベントをリッスンするイベント リスナーを設定する方法を作成できます。

ポリシーは、プラグイン テクノロジー内のオブジェクトを監視するためのゲージまたはトリガのいずれかに適用することができます。ポリシー ゲージ は、オブジェクトの属性を監視し、オブジェクトの値が指定の値を超えた場合に Orchestrator サーバのイベントをプッシュします。ポリシーは監視オブジェクトをトリガし、定義されているイベントがオブジェクトで発生した場合に、Orchestrator サーバのイベントをプッシュします。ポリシー ゲージおよびトリガを **IPluginEventPublisher** インスタンスに登録することで、Orchestrator ポリシーはこれらを監視できるようになります。

IPluginEventPublisher インターフェイスは次のメソッドを定義します。

タイプ	戻り値	説明
<code>pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)</code>	Void	監視するポリシーのためのゲージを公開します。 次のパラメータを取ります。 <ul style="list-style-type: none"> ■ type : 監視するオブジェクトのタイプ。 ■ id : 監視するオブジェクトの ID。 ■ gaugeName : このゲージの名前。 ■ deviceName : ゲージが監視する属性のタイプの名前。 ■ gaugeValue : ゲージがオブジェクトを監視する対象の値。
<code>pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)</code>	Void	監視するポリシーのためのトリガを公開します。 次のパラメータを取ります。 <ul style="list-style-type: none"> ■ type : 監視するオブジェクトのタイプ。 ■ id : 監視するオブジェクトの ID。 ■ triggerName : このトリガの名前。 ■ additionalProperties : 監視するトリガの追加プロパティ。

IPluginFactory インターフェイス

IPluginAdaptor は **IPluginFactory** インスタンスを戻します。**IPluginFactory** インスタンスはプラグイン アプリケーション内でコマンドを実行し、Orchestrator の操作を実行する対象のオブジェクトを検出します。

IPluginFactory インターフェイスは次のフィールドを定義します。

```
static final java.lang.String RELATION_CHILDREN
```

IPluginFactory インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>executePluginCommand(java.lang.String cmd)</code>	Void	プラグインを使用してコマンドを実行します。このメソッドを使用しないことをお勧めします。
<code>find(java.lang.String type, java.lang.String id)</code>	<code>java.lang.Object</code>	プラグインを使用してオブジェクトを検出します。オブジェクトを ID およびタイプによって特定します。
<code>findAll(java.lang.String type, java.lang.String query)</code>	<code>QueryResult</code>	プラグインを使用して、クエリ文字列に一致する特定のタイプのオブジェクトを検出します。クエリの構文はプラグインの IPluginFactory 実装内に定義します。クエリ構文を定義しない場合は、 findAll() によって、指定されたタイプのすべてのオブジェクトが戻されます。
<code>findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>java.util.List</code>	オブジェクトが子を持つかどうかを判別します。
<code>hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)</code>	<code>HasChildrenResult</code>	特定の親と特定の関係をもつすべての子を検出します。
<code>invalidate(java.lang.String type, java.lang.String id)</code>	Void	タイプと ID によってオブジェクトを無効化します。
<code>void invalidateAll()</code>	Void	キャッシュ内のすべてのオブジェクトを無効化します。

IPluginNotificationHandler インターフェイス

IPluginNotificationHandler は、Orchestrator がプラグインを介してアクセスするオブジェクト上で発生するさまざまなタイプのイベントを Orchestrator に通知するためのメソッドを定義します。

IPluginNotificationHandler インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getSessionID()</code>	<code>java.lang.String</code>	現在のセッション ID を戻します。
<code>notifyElementDeleted(java.lang.String type, java.lang.String id)</code>	Void	特定のタイプおよび ID のオブジェクトが削除されたことをシステムに通知します
<code>notifyElementInvalidate(java.lang.String type, java.lang.String id)</code>	Void	オブジェクトの関係が変更されたことをシステムに通知します。 notifyElementInvalidate() メソッドを使用すれば、オブジェクトを無効化した関係の変更だけでなく、オブジェクト間のすべての関係の変更を Orchestrator に通知できます。たとえば、子オブジェクトを親に追加すると、2 つのオブジェクト間の関係が変更されたことになります。

メソッド	戻り値	説明
<code>notifyElementUpdated(java.lang.String type, java.lang.String id)</code>	Void	オブジェクトの属性が変更されたことをシステムに通知します
<code>notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)</code>	Void	現在のモジュールに関するエラー メッセージを公開します

IPluginPublisher インターフェイス

IPluginPublisher インターフェイスは長期実行ワークフローの待機イベント要素を監視するために、イベント通知バスにウォッチャー イベントを公開します。

ワークフローのトリガがプラグイン テクノロジーのイベントを開始すると、トリガを監視し **IPluginPublisher** インスタンスで登録されたプラグイン ウォッチャーは、イベントが発生している待機中のワークフローに通知を行います。

IPluginPublisher インターフェイスは次のメソッドを定義します。

タイプ	値	説明
<code>pushWatcherEvent(java.lang.String id, java.util.Properties properties)</code>	Void	イベント通知バスにウォッチャー イベントを公開します

WebConfigurationAdaptor インターフェイス

WebConfigurationAdaptor インターフェイスは **IConfigurationAdaptor** を実装し、プラグインの構成タブにある Web アプリケーションを見つけてインストールするメソッドを定義します。

注意 Orchestrator 4.1 以降、**WebConfigurationAdaptor** インターフェイスは廃止されました。Web アプリケーションを構成に追加するには、**IConfigurationAdaptor** を実装し、**vso.xml** ファイルの **configuration-war** 属性を使用して Web アプリケーションを特定します。

WebConfigurationAdaptor インターフェイスは次のメソッドを定義します。

メソッド	戻り値	説明
<code>getWebAppContext()</code>	文字列	構成タブにある Web アプリケーションの WAR ファイルを見つけます。 /webapps ディレクトリから WAR ファイルへのパスと名前を文字列として DAR ファイルに返します。
<code>setWebConfiguration(boolean webConfiguration)</code>	ブール値	構成タブの内容に Web アプリケーションが定義されているかどうかを特定します。

PluginTrigger クラス

PluginTrigger クラスは、ワークフローの待機イベント要素の代わりに、プラグイン テクノロジーにおいて監視するオブジェクトおよびイベントについての情報を取得する、トリガ モジュールを作成します。

PluginTrigger クラスは、監視するオブジェクトの名前やタイプ、イベントの特性およびタイムアウト期間を取得または設定します。

ワークフローの待機イベント要素が使用するために実装する専用の **PluginTrigger** クラスを作成します。

Orchestrator ポリシーのポリシー トリガを、イベントを定義し **IPluginEventPublisher.pushTrigger()** メソッドを実装するクラスに定義します。

```
public class PluginTrigger
extends java.lang.Object
implements java.io.Serializable
```

PluginTrigger クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getModuleName()</code>	<code>java.lang.String</code>	トリガ モジュールの名前を取得します。
<code>getProperties()</code>	<code>java.util.Properties</code>	トリガのプロパティのリストを取得します。
<code>getSdkId()</code>	<code>java.lang.String</code>	プラグイン テクノロジーで監視するオブジェクトの ID を取得します。
<code>getSdkType()</code>	<code>java.lang.String</code>	プラグイン テクノロジーで監視するオブジェクトのタイプを取得します。
<code>getTimeout()</code>	<code>Long</code>	トリガのタイムアウト期間を取得します。
<code>setModuleName(java.lang.String moduleName)</code>	<code>Void</code>	トリガ モジュールの名前を設定します。
<code>setProperties(java.util.Properties properties)</code>	<code>Void</code>	トリガのプロパティのリストを設定します。
<code>setSdkId(java.lang.String sdkId)</code>	<code>Void</code>	プラグイン テクノロジーで監視するオブジェクトの ID を設定します。
<code>setSdkType(java.lang.String sdkType)</code>	<code>Void</code>	プラグイン テクノロジーで監視するオブジェクトのタイプを設定します。
<code>setTimeout(long timeout)</code>	<code>Void</code>	タイムアウト期間を秒で設定します。負の値は、タイムアウトを無効にします。

コンストラクタ

- `PluginTrigger()`
- `PluginTrigger(java.lang.String moduleName, long timeout, java.lang.String sdkType, java.lang.String sdkId)`

PluginWatcher クラス

PluginWatcher クラスは、長時間実行ワークフローの待機イベント要素の代わりに、プラグイン テクノロジーの定義済みイベント用のトリガ モジュールをウォッチします。

PluginWatcher クラスは、プラグイン ウォッチャー インスタンスの作成に使用するコンストラクタを定義します。**PluginWatcher** クラスは、監視するワークフロー トリガの名前およびタイムアウト期間を取得および設定する方法を定義します。

```
public class PluginWatcher
extends java.lang.Object
implements java.io.Serializable
```

PluginWatcher クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getId()</code>	<code>java.lang.String</code>	トリガの ID を取得します
<code>getModuleName()</code>	<code>java.lang.String</code>	トリガ モジュール名を取得します
<code>getTimeoutDate()</code>	<code>Long</code>	トリガのタイムアウト日を取得します
<code>getTrigger()</code>	<code>Void</code>	トリガを取得します
<code>setId(java.lang.String id)</code>	<code>Void</code>	トリガの ID を設定します
<code>setTimeoutDate()</code>	<code>Void</code>	トリガのタイムアウト日を設定します

コンストラクタ

`PluginWatcher(PluginTrigger trigger)`

QueryResult クラス

QueryResult クラスには、Orchestrator がプラグインを介してアクセスするオブジェクトに対する、**find** クエリの結果が含まれます。

```
public class QueryResult
extends java.lang.Object
implements java.io.Serializable
```

結果の合計数がクエリが返す結果の数を超えた場合、**totalCount** 値は **QueryResult** が返す数字よりを大きくする可能性があります。クエリが返す結果の数は、**vso.xml** ファイルのクエリの構文で定義されます。

QueryResult クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>addElement(java.lang.Object element)</code>	Void	QueryResult に要素を追加します
<code>addElements(java.util.List elements)</code>	Void	QueryResult に要素のリストを追加します
<code>getElements()</code>	java.util.List	プラグイン アプリケーションから要素を取得します
<code>getTotalCount()</code>	Long	プラグイン テクノロジーで使用可能なすべての要素の合計数を取得します
<code>isPartialResult()</code>	Boolean	取得した結果が完全であるかを判断します
<code>removeElement(java.lang.Object element)</code>	Void	プラグイン テクノロジーから要素を削除します
<code>setElements(java.util.List elements)</code>	Void	プラグイン テクノロジーに要素を設定します
<code>setTotalCount(long totalCount)</code>	Void	プラグイン テクノロジーで使用可能な要素の合計数を設定します

コンストラクタ

- `QueryResult()`
- `QueryResult(java.util.List ret)`
- `QueryResult(java.util.List elements, long totalCount)`

SDKFinderProperty クラス

SDKFinderProperty クラスは、プラグインされたテクノロジー内で Orchestrator ファインダ オブジェクトによって検出されたオブジェクトでプロパティを取得および設定するためのメソッドを定義します。

IDynamicFinder.getProperties メソッドは **SDKFinderProperty** オブジェクトを返します。

```
public class SDKFinderProperty
extends java.lang.Object
```

SDKFinderProperty クラスは、次のメソッドを定義します。

メソッド	戻り値	説明
<code>getAttributeName()</code>	java.lang.String	オブジェクトの属性名を取得する
<code>getBeanProperty()</code>	java.lang.String	Java bean からプロパティを取得する
<code>getDescription()</code>	java.lang.String	オブジェクトの説明を取得する
<code>getDisplayName()</code>	java.lang.String	オブジェクトの表示名を取得する
<code>getPossibleResultType()</code>	java.lang.String	ファインダが返す結果の可能性のあるタイプを取得する

メソッド	戻り値	説明
<code>getPropertyAccessor()</code>	<code>java.lang.String</code>	オブジェクトのプロパティ アクセサを取得する
<code>getPropertyAccessorTree()</code>	<code>java.lang.Object</code>	オブジェクトのプロパティ アクセサ ツリーを取得する
<code>isHidden()</code>	ブール値	オブジェクトの表示/非表示を切り替える
<code>isShowInColumn()</code>	ブール値	データベース カラム内のオブジェクトの表示/非表示を切り替える
<code>isShowInDescription()</code>	ブール値	オブジェクトの説明の表示/非表示を切り替える
<code>setAttributeName(java.lang.String attributeName)</code>	Void	オブジェクトの属性名を設定する
<code>setBeanProperty(java.lang.String beanProperty)</code>	Void	Java bean にプロパティを設定する
<code>setDescription(java.lang.String description)</code>	Void	オブジェクトの説明を設定する
<code>setDisplayName(java.lang.String displayName)</code>	Void	オブジェクトの表示名を設定する
<code>setHidden(boolean hidden)</code>	Void	オブジェクトの表示/非表示を切り替える
<code>setPossibleResultType(java.lang.String possibleResultType)</code>	Void	ファインダが返す結果の可能性のあるタイプを設定する
<code>setPropertyAccessor(java.lang.String propertyAccessor)</code>	Void	オブジェクトのプロパティ アクセサを設定する
<code>setPropertyAccessorTree(java.lang.Object propertyAccessorTree)</code>	Void	オブジェクトのプロパティ アクセサ ツリーを設定する
<code>setShowInColumn(boolean showInTable)</code>	Void	データベース カラム内のオブジェクトの表示/非表示を切り替える
<code>setShowInDescription(boolean showInDescription)</code>	Void	オブジェクトの説明の表示/非表示を切り替える

コンストラクタ

`SDKFinderProperty(java.lang.String attributeName, java.lang.String displayName, java.lang.String beanProperty, java.lang.String propertyAccessor)`

PluginExecutionException クラス

PluginExecutionException クラスは、プラグインが操作を実行する際に例外が発生した場合に、エラー メッセージを返します。

```
public class PluginExecutionException
    extends java.lang.Exception
    implements java.io.Serializable
```

PluginExecutionException クラスは、`class java.lang.Throwable` から次のメソッドを継承します。

`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`、`printStackTrace`、`printStackTrace`、`setStackTrace`、`toString`
`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`

コンストラクタ

`PluginExecutionException(java.lang.String message)`

PluginOperationException クラス

PluginOperationException クラスは、プラグイン操作中に発生したエラーを処理します。

```
public class PluginOperationException
    extends java.lang.RuntimeException
    implements java.io.Serializable
```

PluginOperationException クラスは、`class java.lang.Throwable` から次のメソッドを継承します。

`fillInStackTrace`、`getCause`、`getLocalizedMessage`、`getMessage`、`getStackTrace`、`initCause`、`printStackTrace`、`printStackTrace`、`printStackTrace`、`setStackTrace`、`toString`

コンストラクタ

`PluginOperationException(java.lang.String message)`

HasChildrenResult 列挙

HasChildrenResult 列挙は、特定の親が子を持つかどうかを宣言します。

`IPluginFactory.hasChildrenInRelation` メソッドは **HasChildrenResult** オブジェクトを返します。

```
public enum HasChildrenResult
    extends java.lang.Enum<HasChildrenResult>
    implements java.io.Serializable
```

HasChildrenResult 列挙は次の定数を定義します。

- `public static final HasChildrenResult Yes`
- `public static final HasChildrenResult No`
- `public static final HasChildrenResult Unknown`

HasChildrenResult 列挙は次のメソッドを定義します。

メソッド	戻り値	説明
<code>getValue()</code>	整数	次のいずれかの値を返します。 1 親が子を持ちます -1 親が子を持ちません 0 不明または無効なパラメータ
<code>valueOf(java.lang.String name)</code>	<code>static HasChildrenResult</code>	指定された名前を持つこのタイプの列挙定数を返します。String は、このタイプの列挙定数を宣言するために使用された ID と正確に一致する必要があります。列挙名にスペース文字を使用しないでください。
<code>values()</code>	<code>static HasChildrenResult[]</code>	この列挙型の定数を含む配列を、宣言される順に返します。このメソッドは、次のようにすべての定数にわたって繰り返すことができます。 <pre>for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);</pre>

HasChildrenResult 列挙は `class java.lang.Enum` から次のメソッドを継承します。

`clone`、`compareTo`、`equals`、`finalize`、`getDeclaringClass`、`hashCode`、`name`、`ordinal`、`toString`、`valueOf`

ScriptingAttribute 注釈タイプ

ScriptingAttribute 注釈タイプは、スクリプティングでプロパティとして使用される、プラグインされたテクノロジー内のオブジェクトの属性に注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value={METHOD, FIELD})
public @interface ScriptingAttribute
```

ScriptingAttribute 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

ScriptingFunction 注釈タイプ

ScriptingFunction 注釈タイプは、スクリプティングでプロパティとして使用されるメソッドに注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value={METHOD, CONSTRUCTOR})
public @interface ScriptingFunction
```

ScriptingFunction 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

ScriptingParameter 注釈タイプ

ScriptingParameter 注釈タイプは、スクリプティングでプロパティとして使用されるパラメータに注釈をつけます。

```
@Retention(value=RUNTIME)
@Target(value=PARAMETER)
public @interface ScriptingParameter
```

ScriptingParameter 注釈タイプは、次の値を持っています。

```
public abstract java.lang.String value
```

vso.xml プラグイン定義ファイルの要素

vso.xml ファイルには標準要素のセットが含まれています。一部の要素は必須ですが、オプションの要素もあります。それぞれの要素には、Orchestrator のオブジェクトおよび操作にマップする、オブジェクトおよび操作の値を定義する属性があります。

また、要素は 0 個以上の子要素を持つことができます。さらに子要素は親要素を定義します。同じ子要素が複数の親要素に存在してもかまいません。たとえば、**description** 要素は子要素を持ちませんが、多くの親要素 **module**、**example**、**trigger**、**gauge**、**finder**、**constructor**、**method**、**object**、および **enumeration** の子要素として表示されます。

次に示す各要素の定義では、要素の属性、親要素、および子要素を示します。

module 要素

module は、プラグイン オブジェクトのセットを Orchestrator で使用できるように記述します。

モジュールには、プラグイン テクノロジーからのデータを Java クラスにマップする方法、バージョンング、モジュールの展開方法、および Orchestrator インベントリでのプラグインの表示方法などの情報が含まれます。

<module> 要素はオプションです。**<module>** 要素には、以下の属性が含まれています。

属性	値	説明
name	文字列	プラグインのすべての <finder> 要素のタイプを定義します。これは必須属性です。
version	数値	プラグインの新しいバージョンにパッケージを再ロードするときに使用する、プラグインバージョン番号。これは必須属性です。
build-number	数値	プラグインの新しいバージョンにパッケージを再ロードするときに使用する、プラグインビルド番号。これは必須属性です。
image	イメージファイル	Orchestrator インベントリに表示するアイコン。これは必須属性です。
display-name	文字列	Orchestrator インベントリに表示される名前。これはオプション属性です。
interface-mapping-allowed	true または false	インターフェイス マッピングは推奨されていません。これはオプション属性です。

表 6-3. 要素の階層

親要素	子要素
なし	<ul style="list-style-type: none"> ■ <description> ■ <installation> ■ <configuration> ■ <finder-datasources> ■ <inventory> ■ <finders> ■ <scripting-objects> ■ <enumerations>

description 要素

<description> 要素により、API Explorer ドキュメントに表示されるプラグインの要素の説明を指定することができます。

API Explorer ドキュメントに表示されるテキストを、**<description>** タグと **</description>** タグの間に追加します。

<description> 要素はオプションです。**<description>** 要素の属性はありません。

表 6-4. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ▪ <code><module></code> ▪ <code><example></code> ▪ <code><trigger></code> ▪ <code><gauge></code> ▪ <code><finder></code> ▪ <code><constructor></code> ▪ <code><method></code> ▪ <code><object></code> ▪ <code><enumeration></code> 	なし

廃止された要素

`<deprecated>` 要素により、API Explorer ドキュメントで廃止されたオブジェクトとメソッドにマークを付けます。

API Explorer ドキュメントに表示されるテキストを、`<deprecated>` タグと `</deprecated>` タグの間に追加します。

`<deprecated>` 要素はオプションです。`<deprecated>` 要素の属性はありません。

表 6-5. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ▪ <code><method></code> ▪ <code><object></code> 	なし

url 要素

`<url>` 要素では、オブジェクトまたは列挙に関する外部ドキュメントの URL を指定できます。

URL は、`<url>` タグと `</url>` タグの間に指定します。

`<url>` 要素はオプションです。`<url>` 要素の属性はありません。

表 6-6. 要素の階層

親要素	子要素
<ul style="list-style-type: none"> ▪ <code><enumeration></code> ▪ <code><object></code> 	なし

installation 要素

`<installation>` 要素によって、サーバが起動したときにパッケージをインストールするかスクリプトを実行することができます。

`<installation>` 要素はオプションです。`<installation>` 要素には、次の属性が含まれています。

属性	値	説明
mode	always、never、または version	<p>mode 値を設定すると、Orchestrator サーバの起動時の動作は次のいずれかになります。</p> <ul style="list-style-type: none"> ■ アクションが常に実行される (always) ■ アクションがまったく実行されない (never) ■ サーバが新しいバージョンのプラグインを検出したときにアクションが実行される <p>これは必須属性です。</p>

表 6-7. 要素の階層

親要素	子要素
<module>	<action>

action 要素

<action> 要素は、Orchestrator サーバの起動時に実行されるアクションを指定します。

<action> 要素の属性は、プラグインが起動されたときの動作を定義する Orchestrator パッケージまたはスクリプトへのパスを指定します。

<action> 要素はオプションです。1 つのプラグインに対して設定できる <action> 要素の数に制限はありません。<action> 要素には、以下の属性が含まれています。

属性	値	説明
resource	文字列	dar ファイルのルートから Java パッケージまたはスクリプトへのパスです。これは必須属性です。
type	install-package または execute-script	指定された Orchestrator パッケージを Orchestrator サーバにインストールするか、または指定されたスクリプトを実行します。これは必須属性です。

表 6-8. 要素の階層

親要素	子要素
<installation>	なし

finder-datasource 要素

<finder-datasources> 要素は <finder-datasource> 要素のコンテナです。

<finder-datasources> 要素はオプションです。<finder-datasources> 要素の属性はありません。

表 6-9. 要素の階層

親要素	子要素
<module>	<finder-datasource>

finder-datasource 要素

<finder-datasource> 要素は、プラグイン用に作成された **IPluginAdaptor** 実装の Java クラス ファイルを参照します。

<finder-datasource> 要素を使用して、プラグインを使用するテクノロジーのオブジェクトに対する Orchestrator のアクセス方法を設定することができます。<finder-datasource> 要素により、ユーザーが作成するプラグイン アダプタの Java クラスが特定されます。プラグイン アダプタ クラスは、ユーザーが作成するプラグイン ファクトリをインスタンス化します。プラグイン ファクトリは、プラグインを使用するテクノロジー内のオブジェクトを検索する方法を定義します。<finder-datasource> 要素で、ファクトリが実行するファインディングメソッド呼び出しのタイムアウトを設定することができます。**IPluginFactory** インターフェイスの異なるファインディングメソッドに対して、異なるタイムアウトの値が適用されます。

<finder-datasource> 要素はオプションです。1 つのプラグインに対して設定できる <finder-datasources> 要素の数に制限はありません。<finder-datasource> 要素には、以下の属性が含まれています。

属性	値	説明
name	文字列	<finder> 要素の datasource 属性のデータソースを識別します。XML の id 属性に相当します。これは必須属性です。
adaptor-class	Java クラス	com.vmware.plugins.sample.Adaptor などのプラグイン アダプタを作成するために定義された IPluginAdaptor 実装を参照します。これは必須属性です。
concurrent-call	true (デフォルト値) または false	複数のユーザーが同じアダプタに同時にアクセスすることを許可します。プラグインで同時呼び出しがサポートされていない場合は、 concurrent-call を false に設定する必要があります。これはオプション属性です。
invoker-mode	direct (デフォルト値) または timeout	ファインディング関数のタイムアウトを設定します。値を direct に設定すると、ファインディング関数の呼び出しでタイムアウトが発生しなくなります。値を timeout に設定すると、ファインディングメソッドに対応するタイムアウト期間が Orchestrator サーバによって適用されます。これはオプション属性です。
anonymous-login-mode	never (デフォルト値) または always	ユーザーの名前とパスワードをプラグインに渡すかどうかを指定します。これはオプション属性です。
timeout-fetch-relation	数値 (デフォルトは 30 秒)	findRelation() からの呼び出しに適用されます。これはオプション属性です。

属性	値	説明
timeout-find-all	数値 (デフォルトは 60 秒)	findAll() からの呼び出しに適用されます。これはオプション属性です。
timeout-find	数値 (デフォルトは 60 秒)	find() からの呼び出しに適用されます。これはオプション属性です。
timeout-has-children-in-relation	数値 (デフォルトは 2 秒)	findChildrenInRelation() からの呼び出しに適用されます。これはオプション属性です。
timeout-execute-plugin-command	数値 (デフォルトは 30 秒)	executePluginCommand() からの呼び出しに適用されます。これはオプション属性です。

表 6-10. 要素の階層

親要素	子要素
<finder-datasources>	なし

inventory 要素

<inventory> 要素は Orchestrator クライアントの [インベントリ] ビューおよびオブジェクト選択ダイアログ ボックスに表示されるプラグインのための階層リストのルートを定義します。

<inventory> 要素はプラグイン アプリケーション内のオブジェクトを表すのではなく、Orchestrator スクリプト API のオブジェクトとしてのプラグインそのものを表します。

<inventory> 要素はオプションです。<inventory> 要素には、以下の属性が含まれています。

属性	値	説明
type	Orchestrator オブジェクト タイプ	オブジェクトの階層のルートを表す <finder> 要素のタイプ。これは必須属性です。

表 6-11. 要素の階層

親要素	子要素
<module>	なし

finders 要素

<finders> 要素はすべての <finder> 要素のコンテナです。

<finders> 要素はオプションです。<finders> 要素の属性はありません。

表 6-12. 要素の階層

親要素	子要素
<module>	<finder>

finder 要素

<finder> 要素は Orchestrator クライアント内でプラグインを介して検出されるオブジェクトのタイプを表します。

<finder> 要素はオブジェクト ファインダが示すオブジェクトを定義する Java クラスを特定します。**<finder>** 要素は、オブジェクトが Orchestrator クライアント インターフェイス内で表示される方法を定義します。また、このオブジェクトを示すために Orchestrator スクリプト API が定義するスクリプト オブジェクトを特定します。

ファインダは、異なるタイプのプラグインテクノロジーによって使用されるオブジェクト形式どうしの間のインターフェイスの役割を果たします。

<finder> 要素はオプションです。1 つのプラグインに対して設定できる **<finder>** 要素の数に制限はありません。**<finder>** 要素は次の属性を定義します。

属性	値	説明
type	Orchestrator オブジェクト タイプ	ファインダによって示されるオブジェクトのタイプ。これは必須属性です。
datasource	<finder-datasource name> 属性	データソース refid を使用してオブジェクトを定義する Java クラスを特定します。これは必須属性です。
dynamic-finder	Java メソッド	ファインダの ID とプロパティを vso.xml ファイルに定義する代わりにプログラムで戻すために、 IDynamicFinder インスタンスに実装するカスタム ファインダ メソッドを定義します。これはオプション属性です。
hidden	true または false (デフォルト)	true の場合、Orchestrator クライアント内でファインダを非表示にします。これはオプション属性です。
image	グラフィック ファイルへのパス	Orchestrator クライアント内の階層リストにファインダを表示するための 16x16 のアイコン。これはオプション属性です。
java-class	Java クラスの名前	ファインダによって検出されてスクリプト オブジェクトにマップされるオブジェクトを定義する Java クラス。これはオプション属性です。
script-object	<scripting-object type> 属性	このファインダをマップする <scripting-object> タイプ (ある場合)。これはオプション属性です。

表 6-13. 要素の階層

親要素	子要素
<finders>	<ul style="list-style-type: none"> ■ <id> ■ <description> ■ <properties> ■ <default-sorting> ■ <inventory-children> ■ <relations> ■ <inventory-tabs> ■ <events>

properties 要素

<properties> 要素は、<finder><property> 要素のコンテナです。

<properties> 要素はオプションです。<properties> 要素の属性はありません。

表 6-14. 要素の階層

親要素	子要素
<finder>	<property>

property 要素

<property> 要素は、見つかったオブジェクトのプロパティを Java プロパティまたはメソッド呼び出しにマップします。

プラグイン ファクトリを実装して、処理するプラグイン ファクトリ実装のプロパティを取得するときに、**SDKFinderProperty** クラスのメソッドを呼び出すことができます。

オブジェクトのプロパティは、Orchestrator クライアントのビューに表示したり、非表示にしたりできます。また、列挙を使用してオブジェクトのプロパティを定義できます。

<property> 要素はオプションです。1 つのプラグインに対して設定できる <property> 要素の数に制限はありません。<property> 要素には、以下の属性が含まれています。

属性	値	説明
name	ファインダ名	FinderResult が要素の保存に使用する名前。これは必須属性です。
display-name	ファインダ名	表示されたプロパティ名。これはオプション属性です。

属性	値	説明
bean-property	プロパティ名	bean-property 属性を使用して、 get および set 操作を使用して取得するプロパティを特定します。 MyProperty という名前のプロパティを特定した場合、プラグインは getMyProperty および setMyProperty 操作を定義します。 bean-property または property-accessor のどちらか一方を設定できますが、両方を設定することはできません。これはオプション属性です。
property-accessor	オブジェクトからプロパティ値を取得するメソッド。	property-accessor 属性を使用すると、OGNL 式がオブジェクトのプロパティを検証するように定義できます。 bean-property または property-accessor のどちらか一方を設定できますが、両方を設定することはできません。これはオプション属性です。
show-in-column	true (デフォルト値) または false	true の場合、このプロパティは Orchestrator クライアントの結果テーブルが示します。これはオプション属性です。
show-in-description	true (デフォルト値) または false	true の場合、このプロパティは、オブジェクトの説明を示します。これはオプション属性です。
hidden	true または false (デフォルト)	true の場合、このプロパティはすべてのケースで非表示になります。これはオプション属性です。
linked-enumeration	列挙名	ファインダのプロパティを列挙にリンクします。これはオプション属性です。

表 6-15. 要素の階層

親要素	子要素
<properties>	子要素

relations 要素

<relations> 要素は、<finder><relation> 要素のコンテナです。

<relations> 要素はオプションです。<relations> 要素の属性はありません。

表 6-16. 要素の階層

親要素	子要素
<finder>	<relation>

relation 要素

<relation> 要素は、オブジェクトが他のオブジェクトに関係する方法を定義します。

<relation> 要素には関係名を定義します。

<relation> 要素はオプションです。1 つのプラグインに対して設定できる **<relation>** 要素の数に制限はありません。**<relation>** 要素には、以下の属性が含まれています。

属性	値	説明
name	関係名	この関係の名前。これは必須属性です。
type	Orchestrator オブジェクト タイプ	この関係によって別のオブジェクトに関係するオブジェクトのタイプ。これは必須属性です。
cardinality	to-one または to-many	1 対 1 または 1 対多数として、オブジェクト間の関係を定義します。これはオプション属性です。

表 6-17. 要素の階層

親要素	子要素
<relations>	なし

id 要素

<id> 要素は、ファインダが特定するオブジェクトの固有 ID を取得するためのメソッドを定義します。

<id> 要素はオプションです。**<id>** 要素には、以下の属性が含まれています。

属性	値	説明
accessor	メソッド名	accessor 属性によって、オブジェクトのプロパティを検証するための OGNL 式を定義できます。これは必須属性です。

表 6-18. 要素の階層

親要素	子要素
<finder>	なし

inventory-children 要素

<inventory-children> 要素は Orchestrator クライアントの [インベントリ] ビューおよびオブジェクト選択ボックス内のオブジェクトを表示するリストの階層を定義します。

<inventory-children> 要素はオプションです。**<inventory-children>** 要素の属性はありません。

表 6-19. 要素の階層

親要素	子要素
<finder>	<relation-link>

relation-link 要素

<relation-link> 要素は、[インベントリ] タブの親オブジェクトと子オブジェクト間の階層を定義します。

<relation-link> 要素はオプションです。1 つのプラグインに対して設定できる <relation-link> 要素の数に制限はありません。<relation-link> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	関係名	関係名への refid 。これは必須属性です。

表 6-20. 要素の階層

親要素	子要素
<inventory-children>	なし

events 要素

<events> 要素は <trigger> および <gauge> 要素のコンテナです。

<events> 要素に含めることができるトリガまたはゲージの数に制限はありません。

<events> 要素はオプションです。<events> 要素の属性はありません。

表 6-21. 要素の階層

親要素	子要素
<finder>	<ul style="list-style-type: none"> ■ <trigger> ■ <gauge>

トリガ要素

<trigger> 要素はこのファインダで使用可能なトリガを宣言します。トリガを設定するには、IPluginAdaptor の `registerEventPublisher()` および `unregisterEventPublisher()` メソッドを実装する必要があります。

<trigger> 要素はオプションです。<trigger> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	トリガ名	このトリガの名前。これは必須属性です。

表 6-22. 要素の階層

親要素	子要素
<events>	<ul style="list-style-type: none"> ■ <description> ■ <trigger-properties>

trigger-properties 要素

<trigger-properties> 要素は <trigger-property> 要素のコンテナです。

<trigger-properties> 要素はオプションです。<trigger-properties> 要素の属性はありません。

表 6-23. 要素の階層

親要素	子要素
<trigger>	<trigger-property>

trigger-property 要素

<trigger-property> 要素は、トリガ オブジェクトを特定するプロパティを定義します。

<trigger-property> 要素はオプションです。1 つのプラグインに対して設定できる <trigger-property> 要素の数に制限はありません。<trigger-property> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	トリガ名	このトリガの名前。これはオプション属性です。
display-name	トリガ名	Orchestrator クライアントに表示される名前。これはオプション属性です。
type	トリガ タイプ	このトリガを定義するオブジェクトタイプ。これは必須属性です。

表 6-24. 要素の階層

親要素	子要素
<trigger-properties>	なし

gauge 要素

<gauge> 要素はこのファインダで使用可能なゲージを定義します。ゲージを設定するには、IPluginAdaptor の registerEventPublisher() および unregisterEventPublisher() メソッドを実装する必要があります。

<gauge> 要素はオプションです。1 つのプラグインに対して設定できる <gauge> 要素の数に制限はありません。<gauge> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	ゲージ名	ゲージの名前。これは必須属性です。
min-value	数値	最小しきい値。これはオプション属性です。
max-value	数値	最大しきい値。これはオプション属性です。
unit	オブジェクト タイプ	ゲージを定義するオブジェクト タイプ。これは必須属性です。
format	文字列	監視される値の形式。これはオプション属性です。

表 6-25. 要素の階層

親要素	子要素
<events>	<description>

scripting-objects 要素

<scripting-objects> 要素は <object> 要素のコンテナです。

<scripting-objects> 要素はオプションです。<scripting-objects> 要素の属性はありません。

表 6-26. 要素の階層

親要素	子要素
<module>	<object>

object 要素

<object> 要素は、プラグイン テクノロジーのコンストラクタ、属性、およびメソッドを、Orchestrator のスクリプティング API が公開する JavaScript オブジェクトのタイプにマップします。

オブジェクトの命名規則については、「[プラグイン オブジェクトの命名](#)」を参照してください。

<object> 要素はオプションです。1 つのプラグインに対して設定できる <object> 要素の数に制限はありません。<object> 要素には、以下の属性が含まれています。

タイプ	値	説明
script-name	JavaScript 名	クラスのスクリプティング名。グローバルで一意である必要があります。これは必須属性です。
java-class	Java クラス	この JavaScript クラスにラップされた Java クラス。これは必須属性です。
create	true (デフォルト値) または false	true の場合、このクラスの新しいインスタンスを作成できます。これはオプション属性です。
strict	true または false (デフォルト)	true の場合、vso.xml ファイルで注釈を付けるか、宣言したメソッドのみを呼び出すことができます。これはオプション属性です。

タイプ	値	説明
is-deprecated	true または false (デフォルト)	true の場合、オブジェクトは非推奨 Java クラスをマップします。これはオプション属性です。
since-version	文字列	Java クラスが非推奨になってからのバージョン。これはオプション属性です。

表 6-27. 要素の階層

親要素	子要素
<scripting-objects>	<ul style="list-style-type: none"> ▪ <description> ▪ <deprecated> ▪ <url> ▪ <constructors> ▪ <attributes> ▪ <methods> ▪ <singleton>

constructors 要素

<constructors> 要素は、<object><constructor> 要素のコンテナです。

<constructors> 要素はオプションです。<constructors> 要素の属性はありません。

表 6-28. 要素の階層

親要素	子要素
<object>	<constructor>

constructor 要素

<constructor> 要素は、コンストラクタのメソッドを定義します。<constructor> メソッドは、API Explorer でドキュメントを作成します。

<constructor> 要素はオプションです。1 つのプラグインに対して設定できる <constructor> 要素の数に制限はありません。<constructor> 要素の属性はありません。

表 6-29. 要素の階層

親要素	子要素
<constructors>	<ul style="list-style-type: none"> ▪ <description> ▪ <parameters>

コンストラクタの parameters 要素

<parameters> 要素は、<constructor><parameter> 要素のコンテナです。

<parameters> 要素はオプションです。<parameters> 要素の属性はありません。

表 6-30. 要素の階層

親要素	子要素
<constructor>	<parameter>

コンストラクタの parameter 要素

<parameter> 要素は、コンストラクタのパラメータを定義します。

<parameter> 要素はオプションです。1 つのプラグインに対して設定できる <parameter> 要素の数に制限はありません。<parameter> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	文字列	API ドキュメントで使用するパラメータ名。これは必須属性です。
type	Orchestrator のパラメータ タイプ	API ドキュメントで使用するパラメータ タイプ。これは必須属性です。
is-optional	true または false	true の場合、値は NULL にすることができます。これはオプション属性です。
since-version	文字列	メソッドのバージョン。これはオプション属性です。

表 6-31. 要素の階層

親要素	子要素
<parameters>	なし

attributes 要素

<attributes> 要素は、<object><attribute> 要素のコンテナです。

<attributes> 要素はオプションです。<attributes> 要素の属性はありません。

表 6-32. 要素の階層

親要素	子要素
<object>	<attribute>

attribute 要素

<attribute> 要素は、プラグイン テクノロジーの Java クラスの属性を、Orchestrator の JavaScript エンジンが使用できる JavaScript 属性にマップします。

<attribute> 要素はオプションです。1 つのプラグインに対して設定できる <attribute> 要素の数に制限はありません。<attribute> 要素には、以下の属性が含まれています。

タイプ	値	説明
java-name	Java 属性	Java 属性の名前。これは必須属性です。
script-name	JavaScript オブジェクト	対応する JavaScript オブジェクトの名前。これは必須属性です。
return-type	文字列	この属性が返すオブジェクトのタイプ。API Explorer のドキュメントに記述されています。これはオプション属性です。 注意 JavaScript の戻り値のタイプが Properties の場合、 java.util.HashMap と java.util.Hashtable が基盤となる Java の実装としてサポートされます。
read-only	true または false	true の場合、この属性を変更することはできません。これはオプション属性です。
is-optional	true または false	true の場合、このフィールドは NULL にすることができます。これはオプション属性です。
show-in-api	true または false	false の場合、この属性は API ドキュメントに記述されていません。これはオプション属性です。
is-deprecated	true または false	true の場合、オブジェクトは非推奨属性をマップします。これはオプション属性です。
since-version	数値	この属性が非推奨となったバージョン。これはオプション属性です。

表 6-33. 要素の階層

親要素	子要素
<attributes>	なし

methods 要素

<methods> 要素は、<object><method> 要素のコンテナです。

<methods> 要素はオプションです。<methods> 要素の属性はありません。

表 6-34. 要素の階層

親要素	子要素
<object>	<method>

method 要素

<method> 要素は、プラグイン テクノロジーの Java メソッドを、Orchestrator の JavaScript エンジンが公開する JavaScript メソッドにマップします。

<method> 要素はオプションです。1 つのプラグインに対して設定できる **<method>** 要素の数に制限はありません。 **<method>** 要素には、以下の属性が含まれています。

タイプ	値	説明
java-name	Java メソッド	引数のタイプを括弧内に示した Java メソッド署名の名前。例: getVms(DataStore) 。これは必須属性です。
script-name	JavaScript メソッド	対応する JavaScript メソッドの名前。これは必須属性です。
return-type	Java オブジェクト タイプ	このメソッドが取得するタイプ。これはオプション属性です。 注意 JavaScript の戻り値のタイプが Properties の場合、 java.util.HashMap と java.util.Hashtable が基盤となる Java の実装としてサポートされます。
static	true または false	true の場合、このメソッドは静的になります。これはオプション属性です。
show-in-api	true または false	false の場合、このメソッドは API ドキュメントに表示されません。これはオプション属性です。
is-deprecated	true または false	true の場合、オブジェクトは非推奨のメソッドをマップします。これはオプション属性です。
since-version	数値	このメソッドが非推奨となったバージョン。これはオプション属性です。

表 6-35. 要素の階層

親要素	子要素
<methods>	<ul style="list-style-type: none"> ■ <deprecated> ■ <description> ■ <example> ■ <parameters>

example 要素

<example> 要素を使用すると、API Explorer ドキュメントに表示される Javascript メソッドにサンプル コードを追加することができます。

<example> 要素はオプションです。 **<example>** 要素の属性はありません。

表 6-36. 要素の階層

親要素	子要素
<method>	<ul style="list-style-type: none"> ■ <code> ■ <description>

code 要素

<code> 要素により、API Explorer ドキュメントに表示されるサンプル コードを指定することができます。

サンプル コードは、<code> タグと </code> タグの間に指定します。<code> 要素はオプションです。<code> 要素の属性はありません。

表 6-37. 要素の階層

親要素	子要素
<example>	なし

メソッドの parameters 要素

<parameters> 要素は、<method><parameter> 要素のコンテナです。

<parameters> 要素はオプションです。<parameters> 要素の属性はありません。

表 6-38.

親要素	子要素
<method>	<parameter>

メソッドの parameter 要素

<parameter> 要素はメソッドの入力パラメータを定義します。

<parameter> 要素はオプションです。1 つのプラグインに対して設定できる <parameter> 要素の数に制限はありません。<parameter> 要素には、以下の属性が含まれています。

タイプ	値	説明
name	文字列	パラメータ名。これは必須属性です。
type	Orchestrator のパラメータ タイプ	パラメータ タイプ。これは必須属性です。
is-optional	true または false	true の場合、値は null にすることができます。これはオプション属性です。
since-version	文字列	メソッドのバージョン。これはオプション属性です。

表 6-39. 要素の階層

親要素	子要素
<parameters>	なし

singleton 要素

<singleton> 要素は JavaScript スクリプト オブジェクトを singleton インスタンスとして作成します。

singleton オブジェクトは静的 Java クラスと同様に動作します。singleton オブジェクトは、Orchestrator がプラグイン テクノロジーを使用してアクセスするオブジェクトの特定のインスタンスを定義するのではなく、使用するプラグインの汎用オブジェクトを定義します。たとえば、singleton オブジェクトを使用してプラグイン テクノロジーへの接続を確立できます。

<singleton> 要素はオプションです。<singleton> 要素には、以下の属性が含まれています。

タイプ	値	説明
script-name	JavaScript オブジェクト	対応する JavaScript オブジェクトの名前。これは必須属性です。
datasource	Java オブジェクト	この JavaScript オブジェクトの Java ソース オブジェクトです。これは必須属性です。

表 6-40. 要素の階層

親要素	子要素
<object>	なし

enumerations 要素

<enumerations> 要素は <enumeration> 要素のコンテナです。

<enumerations> 要素はオプションです。<enumerations> 要素の属性はありません。

表 6-41. 要素の階層

親要素	子要素
<module>	<enumeration>

enumeration 要素

<enumeration> 要素は特定のタイプのすべてのオブジェクトに適用される共通の値を定義します。

特定のタイプのすべてのオブジェクトで 1 つの特定の属性が必要で、その属性の値の範囲が限られている場合、異なる値を列挙エントリとして定義できます。たとえば、あるタイプのオブジェクトで **color** 属性が必要で、使用可能な色が、赤、青、および緑のみの場合、これらの 3 つの色の値を定義するための 3 つの列挙エントリを定義できます。エントリは enumeration 要素の子要素として定義します。

<enumeration> 要素はオプションです。1 つのプラグインに対して設定できる **<enumeration>** 要素の数に制限はありません。**<enumeration>** 要素には、以下の属性が含まれています。

タイプ	値	説明
type	Orchestrator オブジェクト タイプ	列挙タイプ。これは必須属性です。

表 6-42. 要素の階層

親要素	子要素
<enumerations>	<ul style="list-style-type: none"> ▪ <url> ▪ <description> ▪ <entries>

entries 要素

<entries> 要素は **<enumeration>****<entry>** 要素のコンテナです。

<entries> 要素はオプションです。**<entries>** 要素の属性はありません。

表 6-43. 要素の階層

親要素	子要素
<enumeration>	<entry>

entry 要素

<entry> 要素は列挙属性の値を提供します。

<entry> 要素はオプションです。1 つのプラグインに対して設定できる **<entry>** 要素の数に制限はありません。

<entry> 要素には、以下の属性が含まれています。

タイプ	値	説明
id	テキスト	列挙エントリを属性として設定するためにオブジェクトが使用する ID。これは必須属性です。
name	テキスト	エントリ名。これは必須属性です。

表 6-44. 要素の階層

親要素	子要素
<entries>	なし

Orchestrator プラグインを開発する場合のベスト プラクティス

Orchestrator プラグインの構造と内容について理解し、特定の問題を回避する方法を知ることにより、プラグインを効率的に開発することができます。

■ Orchestrator プラグインのビルド方法

Orchestrator プラグインはいくつかの方法でビルドできます。プラグラインをレイヤ単位でビルドすること、プラグインのすべてのレイヤを同時にビルドすることもできます。

■ Orchestrator プラグインのタイプ

プラグインを使用すると、汎用ライブラリ、XML や SSH のようなユーティリティ、および vCloud Director などのシステム全体を Orchestrator と統合できます。Orchestrator と統合するテクノロジーに応じて、プラグインはサービス用のプラグイン、汎用プラグイン、およびシステム用のプラグインに分類できます。

■ プラグインの実装

ワークフローのプレゼンテーションの提供と同様に、プラグインの構成、必要な Java クラスおよび JavaScript オブジェクトの実装、プラグイン ワークフローの開発において、便利なプラクティスやテクニックを使用することができます。

■ Orchestrator プラグイン開発の推奨事項

複数の Orchestrator プラグインのコンポーネントを開発する際に、特定のプラクティスを順守することでプラグインの品質を向上させることができます。

■ プラグインと API ドキュメントの ユーザー インターフェイス文字列

Orchestrator のプラグインとそれに関連する API ドキュメントでユーザー インターフェイス (UI) の文字列を指定する場合は、スタイルと形式に関するルールに従う必要があります。

Orchestrator プラグインのビルド方法

Orchestrator プラグインはいくつかの方法でビルドできます。プラグラインをレイヤ単位でビルドすることも、プラグインのすべてのレイヤを同時にビルドすることもできます。

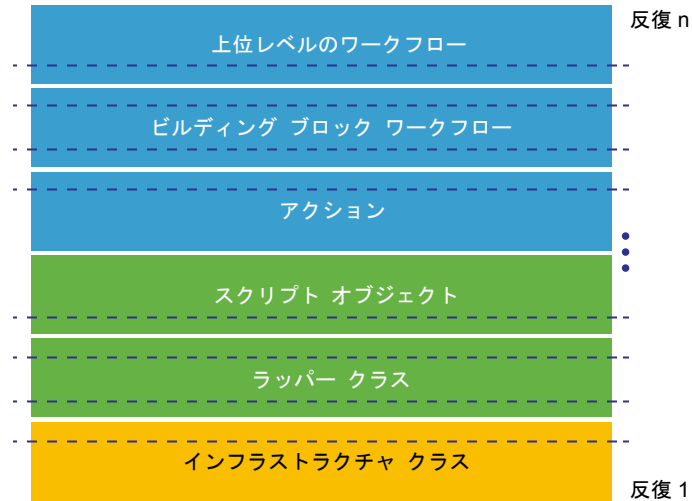
プラグインのレイヤについては、「[\[Orchestrator プラグインの構造\]](#)」を参照してください。

ボトムアップ方式によるプラグイン開発

ボトムアップの開発アプローチを使用すると、レイヤ単位でプラグインを開発することができます。

ボトムアップの開発アプローチでは、下位レベルのレイヤから開発を開始し、上位レベルのレイヤに向かって開発を進めていくことにより、レイヤ単位でプラグインを開発することができます。このアプローチを、対話式的開発アプローチと反復開発アプローチと組み合わせて使用すると、それぞれの反復でレイヤの一部だけを作成することも、レイヤ全体を作成することもできます。その場合、N 回目の反復が完了した時点で、プラグインの開発も完了することになります。

図 6-3. ボトムアップ方式によるプラグイン開発



ボトムアップによるプラグイン開発アプローチの利点は、一度に 1 つのレイヤだけに焦点を当てて開発を行うことができるということです。

ただし、ボトムアップによるプラグイン開発アプローチには、以下の欠点もあることに注意する必要があります。

- ある程度まで作業が進まないと、プラグイン開発の進捗状況を把握できない。
- アジャイル開発手法にはあまり適していない。

ラッパー クラス、スクリプト オブジェクト、アクション、ワークフローをあまり使用しない（あるいはまったく使用しない）小規模なプラグインの場合は、ボトムアップ方式の開発プロセスでも問題はありません。

トップダウン方式によるプラグイン開発

トップダウン方式の開発アプローチを使用すると、プラグインをトップダウン機能にスライスしてビルドできます。

トップダウン方式のアプローチをアジャイル開発プロセスと組み合わせて使用すると、それぞれの反復で新しい機能を作成できます。その場合、N 回目の反復が完了した時点で、プラグインも実装されることになります。

図 6-4. トップダウン方式によるプラグイン開発



トップダウン方式によるプラグイン開発には、以下のような利点があります。

- プラグイン開発の進行状況が初回の反復からわかりやすい。これは新しいプロセスが反復ごとに完了するため、プラグインが解放されて次の反復に使用できることによります。
- 縦にスライスされたプロセスを完了させることにより、成功基準と完了済みプロセスの定義を明確にできるほか、開発者、製品管理担当者、品質保証 (QA) 技術者間のコミュニケーションが向上する。
- QA 技術者が開発プロセスの初期段階からテストと自動化を開始できる。このようなアプローチでは有益なフィードバックを入手して、プロジェクトの納入期間全体を短縮できます。

トップダウン方式によるプラグイン開発アプローチの欠点は、さまざまなレイヤで開発作業が同時に進められる点です。

トップダウン方式によるプラグイン開発プロセスはほとんどのプラグインに適していますが、特に動的な要件を持つプラグインに適しています。

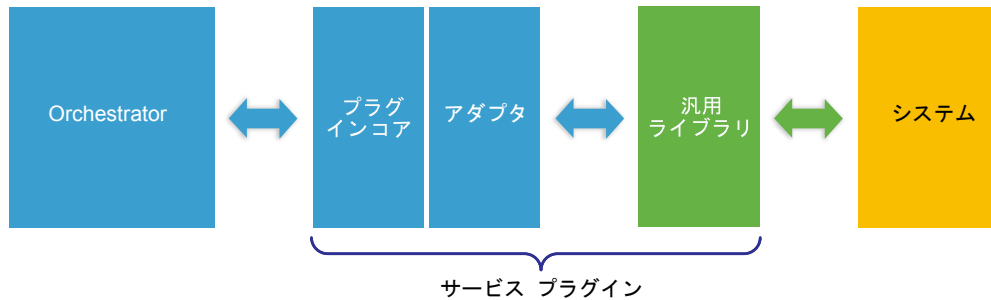
Orchestrator プラグインのタイプ

プラグインを使用すると、汎用ライブラリ、XML や SSH のようなユーティリティ、および vCloud Director などのシステム全体を Orchestrator と統合できます。Orchestrator と統合するテクノロジーに応じて、プラグインはサービス用のプラグイン、汎用プラグイン、およびシステム用のプラグインに分類できます。

サービス用プラグイン

サービス用プラグインまたは汎用のプラグインは、Orchestrator 内のサービスとみなされる機能を提供します。

図 6-5. サービス用プラグインのアーキテクチャ



サービス用プラグインは、一般ライブラリやユーティリティを XML、SSH、SOAP といった Orchestrator に公開します。たとえば、Orchestrator で使用可能な以下のプラグインは、サービス用プラグインです。

JDBC プラグイン	ワークフロー内でデータベースの使用を可能にします。
メール プラグイン	ワークフロー内で電子メールの使用を可能にします。
SSH プラグイン	SSH 接続を開き、ワークフロー内でコマンドを実行します。
XML プラグイン	ワークフロー内の XML ドキュメントを管理します。

サービス用プラグインには、以下のような特徴があります。

複雑性	サービス用プラグインの複雑性は、低から中レベルです。サービス用プラグインは、具体的な機能が分かるように、Orchestrator 内でライブラリの詳細またはライブラリの一部を公開します。たとえば、XML プラグインは、Orchestrator JavaScript API に ドキュメントオブジェクトモデル (DOM) XML パーサーの実装を追加します。
サイズ	サービス用プラグインは比較的小さいサイズです。すべてのプラグインに同じ基本的な一連のクラスが必要であり、また新機能追加のための新しいスクリプトオブジェクトを提供するその他のクラスが必要です。
インベントリ	サービス用プラグインでは作業オブジェクトについて、小規模なインベントリを必要とするか、もしくはインベントリを全く必要としません。サービス用プラグインのオブジェクトモデルは一般的で小規模なため、Orchestrator インベントリ内でこのモデルを表示する必要はありません。

システム用プラグイン

システム用プラグインは、Orchestrator ワークフロー エンジンと外部システムに接続することで、外部のシステムを統合することができます。

システム用プラグインの例を以下に示します。

vCenter Server プラグイン	ワークフローを使用して vCenter Server インスタンスの管理を行います。
vCloud Director プラグイン	ワークフロー内で vCloud Director のインストールの操作を行います。
Cisco UCSM プラグイン	ワークフロー内で Cisco エンティティの操作を行います。

システム用プラグインの主な特徴は、以下のとおりです。

複雑性

システム用プラグインは、公開するテクノロジーが比較的複雑であるため、汎用のプラグインより複雑性が高レベルになっています。システム用プラグインは、Orchestrator で外部システムの操作を行いその機能を使用するための、Orchestrator 内にあるすべての外部システムの要素を表します。外部システムが統合メカニズムを持っている場合、それを使用してより簡単に Orchestrator 内でシステムの機能を公開することができます。ただし、外部システムの要素を表す以外に、システム用プラグインには高いスケーラビリティとキャッシュ メカニズムの提供、イベントや通知の処理なども求められる可能性があります。

サイズ

システムのプラグインのサイズは、中規模から大規模です。システム用プラグインは、通常スクリプト オブジェクトを多数持つため、基本的な一連のクラスとは別に多くのクラスを必要とします。また、それらと相互に作用するその他のヘルパーや補助クラスを必要とする場合があります。

インベントリ

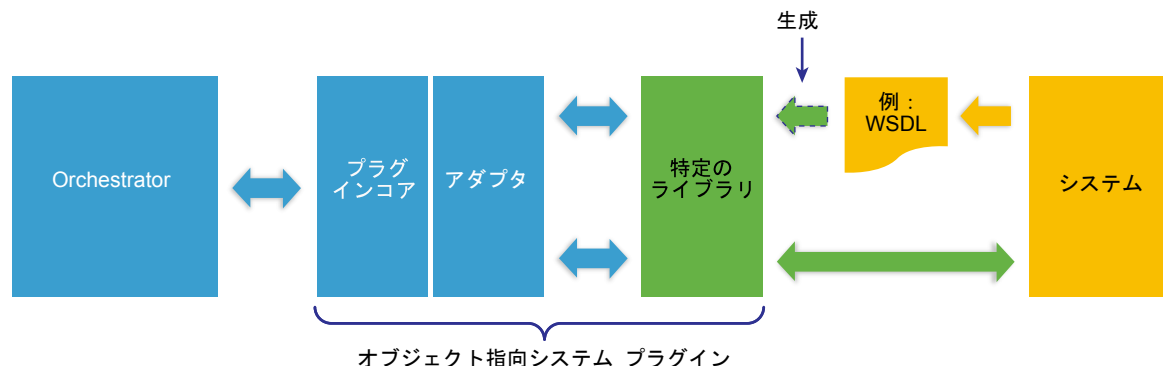
通常、システム用プラグインには多数のオブジェクトがあるため、これらのオブジェクトをインベントリに適切に公開して、Orchestrator 内で容易に場所が分かり使用できるようにする必要があります。システム用プラグインが公開する必要のあるオブジェクトの数が多いため、プラグイン用にできるだけ多くのコードを自動生成する補助ツールまたはプロセスを構築することが推奨されます。たとえば、vCenter Server プラグインは、そのようなツールを提供します。

オブジェクト指向システム用プラグイン

オブジェクト指向のシステムでは、オブジェクトおよび RPC に基づいた操作メカニズムを提供します。

最も一般的に使用されているオブジェクト指向システムのモデルは、SOAP を使用する Web サービスモデルです。このモデル内のオブジェクトは、オブジェクトの状態に関する属性のセットを持ち、ターゲットシステム側で起動されるリモート メソッドのセットを提供します。

図 6-6. オブジェクト指向システム用プラグイン



オブジェクト指向システム用のプラグインの実行時には、以下を考慮します。

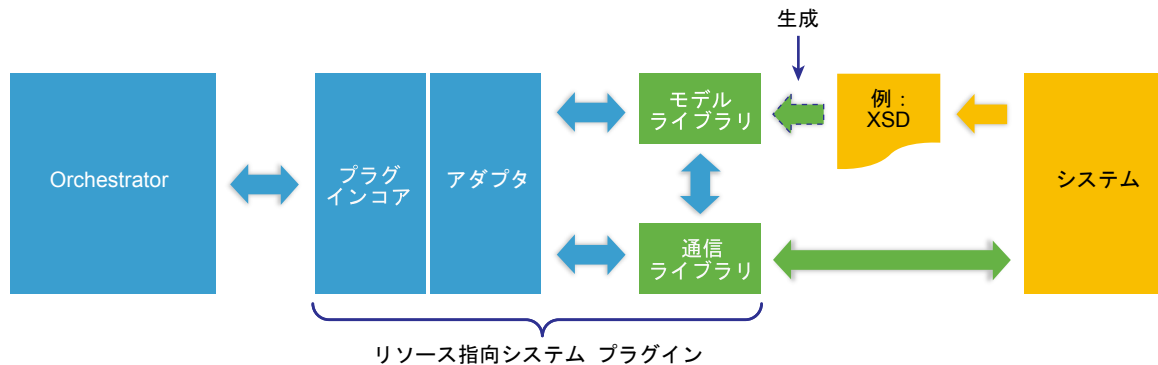
- SOAP を使用する場合は、WSDL ファイルを使用して、オブジェクトモデルと通信メカニズムを組み合わせるクラスのセットを生成することができます。
- このオブジェクトモデルは、Orchestrator 内で公開するほぼすべてのものになります。

リソース指向システム用プラグイン

リソース指向システムでは、リソースに基づいた操作メカニズムおよび HTTP 手法を用いた簡単な操作を提供しています。

リソース指向システム モデルの最も代表的なものは、XML などと組み合わせた REST モデルです。モデル内のオブジェクトは、状態に関連した属性を持ちます。ターゲットシステム（通信メカニズム）でメソッドを開始するには、GET、POST、PUT といった標準的な HTTP 手法を使用し、いくつかの規則に従う必要があります。

図 6-7. リソース指向システム用プラグイン



リソース指向システム用のプラグインの実行時には、以下を考慮します。

- REST を使用する、または XML で HTTP のみを使用する場合、メッセージの読み書き可能な XML スキーマ ファイルを 1 つ以上取得します。このスキーマから、オブジェクト モデルを定義する一連のクラスを生成することができます。この一連のクラスでは、たとえば vCloud Director プラグインの定義のように HTTP 手法で暗示的に操作を定義したり、また Cisco UCSM プラグインのように、XML メッセージで明示的に定義したりします。
- 別のクラスでは、通信メカニズムを実装する必要があります。この一連のクラスでは、元のオブジェクト モデルを操作する新しいオブジェクトモデルを定義します。通信メカニズム用のオブジェクト モデルは、オブジェクトとメソッドのみで構成されます。
- Orchestrator 内では、元のオブジェクト モデルと通信メカニズム用のオブジェクト モデルの両方を公開することができます。両方のオブジェクトモデルの公開方法や、関連オブジェクトを両側から統合するか（オブジェクト指向システムのシミュレーションのため）または別々にするかどうかにより、複雑性が増す可能性があります。

プラグインの実装

ワークフローのプレゼンテーションの提供と同様に、プラグインの構成、必要な Java クラスおよび JavaScript オブジェクトの実装、プラグイン ワークフローの開発において、便利なプラクティスやテクニックを使用することができます。

プロジェクト構造

Orchestrator プラグインのプロジェクトに対して、標準の構造を適用することができます。

■ 内部プロジェクト

プラグインの実装時に、オブジェクトのキャッシュや、オブジェクトのバックグラウンドへの移動、オブジェクトのクローンの作成など、特定のアプローチを適用することができます。それらの手法に従うことで、プラグインのパフォーマンスを改善、並行処理の問題を回避、および Orchestrator クライアントの応答性を改善することができます。

■ 内部ワークフロー

ワークフローを実装することで、Orchestrator プラグインが実行する長時間の処理を監視することができます。

■ ワークフローとアクション

ワークフロー開発と使用を容易にするために、特定のプラクティスを使用することができます。

■ ワークフロー プレゼンテーション

ワークフローのプレゼンテーションを作成するには、特定の構造と規則を適用する必要があります。

プロジェクト構造

Orchestrator プラグインのプロジェクトに対して、標準の構造を適用することができます。

プラグイン プロジェクト用のモジュールを持つ標準の Maven 構造を使用して、各種の機能が存在する場所を明確に確認することができます。

表 6-45. プラグイン プロジェクトの構造

モジュール	説明
/myAwesomePlugin-plugin	プラグイン プロジェクトの root モジュール。
/o11nplugin-myAwesomePlugin	最終プラグインの DAR ファイルを構成するモジュール。
/o11nplugin-myAwesomePlugin-config	プラグイン構成 Web アプリケーションを格納するモジュール。このモジュールにより、標準の WAR ファイルが生成されます。
/o11nplugin-myAwesomePlugin-core	Orchestrator 標準プラグインの任意のインターフェイスを実装するすべてのクラスと、そのクラスで使用される他の補助クラスを格納するモジュール。このモジュールにより、標準の JAR ファイルが生成されます。
/o11nplugin-myAwesomePlugin-model	プラグイン経由でサードパーティ テクノロジーを Orchestrator に統合するためのすべてのクラスを格納するモジュール。これらのクラスに、API の標準 Orchestrator プラグインに対する直接参照を含めないでください。
/o11nplugin-myAwesomePlugin-package	アクションとワークフローを使用して Orchestrator の外部パッケージ ファイルをインポートし、最終プラグインの DAR ファイル内のそのファイルを含めるモジュール。このモジュールはオプションです。

内部プロジェクト

プラグインの実装時に、オブジェクトのキャッシュや、オブジェクトのバックグラウンドへの移動、オブジェクトのクローンの作成など、特定のアプローチを適用することができます。それらの手法に従うことで、プラグインのパフォーマンスを改善、並行処理の問題を回避、および Orchestrator クライアントの応答性を改善することができます。

オブジェクトのキャッシュ

プラグインはリモート サービスと通信する場合があります、この通信はサービス側のリモート オブジェクトを代表するローカル オブジェクトによって行われます。ローカル オブジェクトをリモートサービスから毎回取得するのではなくキャッシュに格納することで、プラグインのパフォーマンスを Orchestrator のユーザー インターフェイスの応答性と同程度に向上させることができます。キャッシュの範囲は、たとえばすべてのプラグイン クライアントにつき 1 キャッシュ、プラグインのユーザーごとに 1 キャッシュ、および第三者サービスのユーザーごとに 1 キャッシュなどにすることができます。実装時には、キャッシュ メカニズムは、オブジェクトの検索および無効化のプラグイン インターフェイスと統合されます。

オブジェクトのバックグラウンドへの移動

プラグイン インベントリのオブジェクトのリストが大きく、オブジェクトをすばやく検索する方法がない場合は、オブジェクトをバックグラウンドに移動させることができます。オブジェクトをバックグラウンドへ移動させるには、たとえばオブジェクトを **fake** と **loaded** の 2 つの状態にします。**fake** オブジェクトの作成が非常に容易で、インベントリに表示される情報が名前や ID など最小限であると仮定します。その場合、常に **fake** オブジェクトを返すことが可能であり、またすべての情報（実際のオブジェクト）が必要になった場合には、使用しているエンティティまたはプラグインによって、自動的に **load** メソッドが開始され、実際のオブジェクトが取得されます。エンティティを使用するアクションを予測して、fake オブジェクトが返された後にオブジェクトの読み込みが自動的に開始するように構成することもできます。

並行処理の問題回避のためクローンを作成する

プラグインにキャッシュを使用する場合、オブジェクトのクローンを作成する必要があります。要求したすべてのエンティティに対して常に同じオブジェクトのインスタンスを返すキャッシュを使用すると、望ましくない影響が生じる可能性があります。たとえば、エンティティ A はオブジェクト O を要求し、エンティティの表示では、インベントリ内のオブジェクトのすべての属性が表示されるとします。同時に、エンティティ B もオブジェクト O を要求し、エンティティ A はオブジェクト O の属性を変更するワークフローを実行します。実行の最後に、ワークフローはオブジェクトの **update** メソッドを起動して、サーバ側のオブジェクトの更新を行います。エンティティ A とエンティティ B がオブジェクト O と同じインスタンスを取得する場合、エンティティ A は、変更がサーバ側でコミットされる前でも、インベントリ内のエンティティ B が実行したすべての変更を表示します。実行が成功した場合は問題ありませんが、実行が失敗した場合は、エンティティ A のオブジェクト O の属性は戻らなくなります。このような場合は、キャッシュ（プラグインの **find** 操作）が常に同一のインスタンスの代わりにオブジェクトのクローンを返すことで、それぞれがエンティティの表示を使用してコピーを変更し、少なくとも Orchestrator 内では並行処理の問題を回避することができます。

変更を他の人に通知する

キャッシュやクローン オブジェクトを同時に使用する場合、問題が発生する可能性があります。最大の問題は、エンティティのビューを使用しているオブジェクトが、そのオブジェクトで使用可能な最新バージョンではなくなることです。たとえば、エンティティがインベントリを表示する場合、オブジェクトは一度読み込まれますが、同時に他のエンティティがオブジェクトを変更している場合は、最初のエンティティにはその変更が表示されません。この問題を回避するには、Orchestrator のプラグイン API から **PluginWatcher** メソッドおよび **IPluginPublisher** メソッドを使用して、変更について通知し、Orchestrator クライアントの他のインスタンスに変更の確認を許可します。また、これは Orchestrator クライアント独自のインスタンスでも、インベントリのオブジェクト 1 つの変更がその他のインベントリに影響するため通知が必要な場合に適用されます。通知が使用されることの多い操作は、オブジェクトそのものまたはプロパティがインベントリに表示されているオブジェクトの追加、更新、削除です。

オブジェクトをいつでも検索可能にする

オブジェクトをタイプと ID のみで検索可能にするには、**IPluginFactory** インターフェイスの **find** メソッドを実装します。**find** メソッドは、Orchestrator を再起動し任意のワークフローを再開した後に直接開始することができます。

保有していないクエリ サービスをシミュレーションする

Orchestrator クライアントは、特定のケースのオブジェクトについてクエリを作成したり、クエリをツリーではなくリストまたは表などとして表示する必要がある場合があります。これは、プラグインが一連のオブジェクトについていつでもクエリを作成できる必要があることを意味します。サードパーティのテクノロジーによりクエリ サービスが提供されている場合は、このサービスを適用して使用する必要があります。クエリ サービスが提供されていない場合は、非常に複雑であったりパフォーマンスが低い場合であっても、クエリ サービスのシミュレーションをする必要があります。

検索メソッドがランタイム例外を返さないようにする

プラグイン内の検索を実装するための **IPluginFactory** インターフェイスは、コントロールまたは非コントロールされたランタイム例外をスローしません。これは、ワークフロー実行中の **validation error** 不具合の原因となる場合があります。たとえば、ワークフローの 2 つのノード間で、最初のノードの出力が 2 番目のノードの入力である場合に **find** メソッドが開始されます。その時オブジェクトがランタイム例外のため見つからない場合は、Orchestrator クライアント内では **validation error** 以上の情報を得ることはできません。その後、プラグインが例外をどのようにログに記録するかによって、ログ ファイル内の情報の量が変わります。

内部ワークフロー

ワークフローを実装することで、Orchestrator プラグインが実行する長時間の処理を監視することができます。

タスクの監視など、長時間実行される処理の監視のためにワークフローを実装することができます。このワークフローは、Orchestrator トリガおよび待機中のイベントに基づいて作成できます。ただし、タスク待機中でブロックされているワークフローは、Orchestrator サーバの起動後すぐに再開されることを考慮しておく必要があります。プラグインは、監視プロセスを正しく再開するために必要なすべての情報を取得できなければなりません。

内部的に使用される監視ワークフローまたはタスクは、ポーリング レートおよび有効なタイムアウト期間を指定するメカニズムを持っている必要があります。

特にコードが Java コードを開始するものではない場合、ワークフロー内のスクリプティング コードの一部をデバッグするプロセスは簡単ではありません。このため、デフォルトの Orchestrator スクリプト オブジェクトで提供されているログの記録方法を使用することが唯一の選択肢となる場合があります。

ワークフローとアクション

ワークフロー開発と使用を容易にするために、特定のプラクティスを使用することができます。

開発ワークフローをビルディング ブロックとして開始する

ビルディング ブロックは数個のパラメータ入力で単純な出力を返す、簡易的なワークフローです。十分なビルディング ブロックがある場合は、より高レベルなワークフローを簡単に作成でき、よりよいツールのセットを提供することで複雑なワークフローを構成することができます。

小規模なコンポーネントに基づいて高レベルのワークフローを作成する

複雑なワークフローをいくつかの入力と内部手順で開発する場合、そのワークフローをより小さく単純なビルディング ブロックとアクションに分割することができます。

アクションの作成がいつでも可能

ワークフローの開発時、柔軟にアクションを作成できます。

- スクリプティング方法に合わせて複雑なオブジェクトやパラメータを簡単に作成
- 共通のコードの繰り返しを常に回避
- ユーザー インターフェイスの検証を実施

ワークフローでアクションの呼び出しがいつでも可能

ワークフローのスキーマ内のノードとして、アクションを直接呼び出すことができます。これにより、1 つのアクションを呼び出すためにスクリプティング コードを追加する必要がなくなり、ワークフローのスキーマがより単純になります。

予測情報の入力

ワークフローまたはアクションの要素ごとに情報を提供します。

- ワークフローまたはアクションの説明を提供
- 入力パラメータの説明を提供
- 出力の説明を提供
- ワークフローの属性の説明を提供

バージョン情報を最新に維持

プラグインのバージョン更新時、プラグインのメジャー アップデート、実装の重要な詳細などの情報とともに、重要なコメントを追加します。

ワークフロー プレゼンテーション

ワークフローのプレゼンテーションを作成するには、特定の構造と規則を適用する必要があります。

ワークフロー プレゼンテーションのワークフローの入力には、次のプロパティを使用します。

表 6-46. ワークフロー入力のプロパティ

プロパティ	用途
Show in Inventory	このプロパティにより、ユーザーはワークフローをインベントリ ビューから実行できます。
Specify a root object to be shown in the chooser	このプロパティは、ユーザーが入力を選択できるようにします。root オブジェクトが、プレゼンテーション内で更新可能な場合、属性の場合、またはオブジェクト メソッドで取得された場合、プレゼンテーション内でオブジェクトを更新する適切なアクションを作成または設定する必要があります。

表 6-46. ワークフロー入力のプロパティ (続き)

プロパティ	用途
Maximum string length	このプロパティは、名前、説明、ファイルパスなど、長い文字列に使用します。
Minimum string length	このプロパティは、テスト ツールの空の文字列を防止するために使用します。
Custom validation	アクションを伴う詳細な検証を実装します。

入力をステップと表示グループで整理します。このように整理することで、ユーザーはワークフローのすべての入力パラメータを認識および識別することができます。

Orchestrator プラグイン開発の推奨事項

複数の Orchestrator プラグインのコンポーネントを開発する際に、特定のプラクティスを順守することでプラグインの品質を向上させることができます。

表 6-47. プラグインの実装における有益なプラクティス

コンポーネント	アイテム	説明
一般	サードパーティ API へのアクセス	プラグインでは、可能な限り簡素化したサードパーティ API へのアクセスメソッドを使用します。
	インターフェイス	プラグインでは、API が複雑であっても、ユーザーに分かりやすく標準的なインターフェイスを使用します。
アクション	スクリプト オブジェクト	オブジェクトの作成、変更、削除、ならびにオブジェクトのスクリプティングに使用するその他すべてのメソッドにアクションを作成します。
	説明	アクションの説明には、そのアクションがどのように機能するかではなく、そのアクションが何をするかを記述します。
	スクリプティング	オブジェクトのプロパティまたはメソッドの取得にスクリプティングを使用する場合、オブジェクトの値が null または undefined と異なることを確認してください。
	廃止	あるアクションが廃止された場合、 comment または throw ステートメントで代替のアクションを示すか、そのアクションが代替のアクションを呼び出して、廃止されたバージョンのアクションでビルドされたソリューションが失敗しないようにします。
ワークフロー	統合されたテクノロジーのユーザー インターフェイス操作	統合されたテクノロジーのユーザー インターフェイスで利用できる各操作に対して、ワークフローを作成します。
	説明	ワークフローの説明には、そのワークフローがどのように機能するかではなく、そのワークフローが何をするかを記述します。
	プレゼンテーション プロパティ mandatory input	必須のワークフロー入力にはすべて、 mandatory input プロパティを設定する必要があります。
	プレゼンテーション プロパティ default value	任意のエンティティを構成するワークフローを開発する場合、ワークフローのプレゼンテーションは、そのエンティティ用にデフォルトの設定値を読み込む必要があります。たとえば、「ホスト構成」という名前のワークフローを開発する場合、ワークフローのプレゼンテーションはホスト構成のデフォルト値を読み込む必要があります。

表 6-47. プラグインの実装における有益なプラクティス (続き)

コンポーネント	アイテム	説明
	プレゼンテーション プロパティ Show in inventory	インベントリのオブジェクトにコンテキスト ワークフローを作成するため、 Show in inventory プロパティを設定する必要があります。
	プレゼンテーション プロパティ specify a root parameter	このプロパティは、ツリー ルートからインベントリを閲覧する必要がない時に、ワークフローで使します。
	ワークフローの検証	ワークフローの検証を行いすべてのエラーを修正する必要があります。
	オブジェクトの作成	新規オブジェクトを作成するすべてのワークフローは、出力パラメータとして新規プロジェクトを返します。
	廃止	ワークフローが廃止された場合、 comment または throw ステートメントで代替のワークフローを示すか、廃止されたワークフローが代替のワークフローを呼び出して、以前のバージョンのワークフローでビルドされたソリューションが失敗しないようにします。
	インベントリ	インベントリにホストへの接続が含まれていて、そのホストが使用できなくなった場合は、ホストが切断されたことを示す必要があります。 root オブジェクトに disconnected を付けて名前を変更する、または vCloud Director プラグインと同様にそのオブジェクトの下にあるツリー オブジェクトを削除することで、ホストが切断されたことを示すことができます。
	Select value as list プロパティ	インベントリ オブジェクトは treeview または list 形式で選択可能である必要があります。
	ホスト マネージャ	プラグインが host オブジェクトをターゲット システムに実装する場合、親である hostmanager root オブジェクトには、追加、削除、またはホスト プロパティの編集のプロパティが存在する必要があります。
	オブジェクトの取得または更新	クエリ サービスが統合されたテクノロジー上で実行されている場合、複数のオブジェクトの取得にクエリ サービスを使用します。
	子の検出	子オブジェクトを個別に取得する必要がある場合、取得プロセスにはマルチスレッド機能を用いて、1 つのエラーでブロックされないようにします。
	Orchestrator オブジェクトの変更	インベントリの要素の状態を変更する可能性のあるすべてのワークフローでは、インベントリの更新を行い、オブジェクトが非同期状態にならないようにします。
	外部オブジェクトの変更	通知メカニズムを使用して、Orchestrator の外部で行われた操作の結果生じた統合テクノロジーの変更を通知することができます。そのような操作により統合テクノロジーからオブジェクトが削除される場合、インベントリを更新してデータのエラーや消失を防ぐ必要があります。たとえば、仮想マシンが vCenter Server から削除された場合、vCenter Server プラグインはインベントリを更新して削除された仮想マシンのオブジェクトを削除します。
	ファインダ オブジェクト	ファインダ オブジェクトには、オブジェクトの識別ができるプロパティを設定します。これらは通常、ユーザー インターフェイスに表示されるプロパティです。
	オブジェクトのスク립ティング	equals メソッドを実装して、オブジェクトに 2 つのインスタンスが含まれている場合と同じオブジェクトに == の操作が行われるようにします。
	プラグイン オブジェクトのプロパティ	親オブジェクトを持つオブジェクトには、 parent プロパティを実装します。
	プラグイン オブジェクトのプロパティ	子オブジェクトを持つオブジェクトには、子オブジェクトの配列を返す GET メソッドを実装します。

表 6-47. プラグインの実装における有益なプラクティス (続き)

コンポーネント	アイテム	説明
	インベントリ オブジェクト	インベントリ オブジェクトは Server.find で検索できるようにします。 ワークフロー内で入力属性または出力属性として使用できるように、すべてのインベントリ オブジェクトを直列化します。
	コンストラクタとメソッド	多くの場合、スクリプト可能なオブジェクトは、コンストラクタを持っている、もしくは他のオブジェクト属性またはメソッドにより返されたものである必要があります。
	オブジェクト ID	外部システムから発行された ID を持つオブジェクトには、1 つ以上のサーバを統合した場合に ID の重複が発生しないように、内部 ID を使用します。
	オブジェクトの検索	search メソッドまたは find メソッドでは、すべてのオブジェクトではなく指定した名前または ID が検索されるように、フィルタを実装します。たとえば、Orchestrator サーバに、プラグインオブジェクトを ID で検索できる Server.FindForId メソッドがあるとします。実行するには、このメソッドをプラグイン内の検索可能な各オブジェクトに実装する必要があります。
	トリガ	可能な場合は、Orchestrator での様々なイベントにおけるポリシーのトリガを可能にするため、オブジェクトが変更されてもトリガを使用できるようにします。たとえば、Orchestrator は新しい仮想マシンの追加、電源オン、電源オフなどがいつ行われたかなどの確認のため、 Datacenter オブジェクトの vCenter プラグインでトリガまたはイベントを監視することができます。
	オブジェクトのプロパティ	他のプラグインにあるオブジェクトには、あるプラグインから別のプラグインへ容易に変換できるプロパティを設定します。たとえば、仮想マシンのオブジェクトは、 moref (managed object reference ID) プロパティを持つなどです。
	セッション マネージャ	異なるセッションを持つ可能性のあるリモート サーバに接続する場合、共有セッションおよびユーザーごとのセッションをプラグインに実装します。
トリガ	トリガ	すべての長時間の処理およびブロック メソッドは、返されたタスクを非同期で開始でき、完了時にはトリガ イベントを生成できることが推奨されます。
列挙	列挙	特定のタイプの列挙は、列挙内の異なる値から選択できるインベントリ オブジェクトを持つことが推奨されます。
ログ	ログ	メソッドは様々なログ レベルを実装する必要があります。
バージョンング	プラグインのバージョン	プラグインのバージョンは、基準に従い、プラグインの更新に合わせて更新されるようにします。
API ドキュメント	メソッド	API ドキュメントに記述されているメソッドは、オブジェクトに no xyz method / property 例外をスローしないようにします。代わりにメソッドは、使用できるプロパティがない場合は null を返し、プロパティが使用できない場合は詳細についてドキュメントを作成します。
	vso.xml	すべてのオブジェクト、メソッド、ならびにプロパティは vso.xml 内に記録される必要があります。

プラグインと API ドキュメントの ユーザー インターフェイス文字列

Orchestrator のプラグインとそれに関連する API ドキュメントでユーザー インターフェイス (UI) の文字列を指定する場合は、スタイルと形式に関するルールに従う必要があります。

一般的な推奨事項

- プラグインで使用する VMware 製品については、正式な名称を使用してください。たとえば、以下の製品と VMware 用語については、正式な名称を使用してください。

正しい名称	間違った名称
vCenter Server	VC、vCenter
vCloud Director	vCloud

- ワークフローの説明の最後には、必ずピリオド (.) を指定してください。たとえば、「**Creates a new Organization.**」のように指定してください。
- スペルチェック機能付きのテキスト エディタを使用して説明を入力してから、その説明をプラグインに移動してください。
- プラグインの名前は、そのプラグインが関連付けられている承認済みサードパーティ製品の名前と正確に一致している必要があります。

ワークフローとアクション

- わかりやすい説明を入力してください。多くのアクションとワークフローの場合、説明としては 1 文または 2 文で十分です。
- 上位レベルのワークフローの場合、より詳しい説明やコメントが必要になることがあります。
- 説明は、「**Creates...**」などの動詞で始めてください。「**This workflow creates**」などのような文章を入力しないでください。
- 説明文の最後には、必ずピリオド (.) を指定してください。
- ワークフローやアクションがどのように実行されるかということではなく、ワークフローやアクションによって何が実行されるのかということを入力してください。
- 通常、ワークフローとアクションは、フォルダやパッケージ内に保存されます。そのため、これらのフォルダとパッケージについても、簡単な説明を入力してください。たとえば、ワークフロー フォルダの場合は、「**Set of workflows related to vApp Template management**」などのような説明を入力します。

ワークフローとアクションのパラメータ

- ワークフローとアクションのパラメータの説明は、「**Name of**」などの名詞で始めてください。「**It's the name of**」などのような文章を入力しないでください。
- パラメータとアクションの説明の最後にピリオド (.) を指定しないでください。この場合の説明は、完全な文章ではないためです。
- ワークフローの入力パラメータの場合、適切な名前のラベルをプレゼンテーション ビューで指定する必要があります。多くの場合、関連する入力情報を 1 つの表示グループとして組み合わせることができます。たとえば、「組織名」と「組織の正式名称」というラベルを持つ 2 つの入力情報を個別に使用する代わりに、「組織」というラベルを持つ表示グループを 1 つだけ作成し、「組織名」と「組織の正式名称」というラベルを持つ 2 つの入力情報を「組織」という表示グループに含めることができます。

- ステップと表示グループについては、ワークフローのプレゼンテーションに表示される説明やコメントも追加してください。

プラグイン API

- API のドキュメントは、**vso.xml** ファイルと Java ソース ファイル内のすべてのドキュメントを参照します。
- **vso.xml** ファイルについては、ワークフローとアクションの説明を入力する場合と同じルールで、ファインダオブジェクトとスクリプト オブジェクトの説明を入力してください。オブジェクトの属性とメソッドのパラメータの説明については、ワークフローとアクションのパラメータを入力する場合と同じルールで説明を入力してください。
- **vso.xml** ファイル内で特殊文字を使用しないでください。また、**<![CDATA[insert your description here!]]>** タグ内に説明を入力してください。
- Java ソース ファイルの場合は、標準の Javadoc スタイルを使用してください。

Maven を使用したプラグインの作成

Orchestrator Appliance には、Maven のアーティファクトを格納するためのリポジトリが用意されています。これらのアーティファクトを使用すると、アーキタイプからプラグイン プロジェクトを作成することができます。

このリポジトリの場所は、`https://<orchestrator_server>:8281/vco-repo/` です。使用している Maven バージョンが HTTPS をサポートしていない場合は、`http://<orchestrator_server>:8280/vco-repo/` にあります。この場所は、Orchestrator の標準 Maven プラグイン プロジェクトの `pom.xml` ファイルに組み込まれています。Orchestrator Appliance が展開されていない場合、このリポジトリにアクセスすることはできません。

この章では次のトピックについて説明します。

- [Maven を使用してアーキタイプから Orchestrator プラグインを作成する](#)
- [Maven のアーキタイプ](#)
- [Maven ベースのプラグイン開発のベスト プラクティス](#)

Maven を使用してアーキタイプから Orchestrator プラグインを作成する

コマンドライン インターフェイスで一連のコマンドを実行することにより、Orchestrator の標準 Maven プラグインをアーキタイプから作成することができます。

開始する前に

- Orchestrator Appliance 5.5.1 以降がインストールされていることを確認します。
- Apache Maven 3.0.4 または 3.0.5 がインストールされていることを確認します。

手順

- 1 任意のアーキタイプを選択し、インタラクティブ モードでプロジェクトを作成します。

```
mvn archetype:generate -DarchetypeCatalog=https://<orchestrator_server>:8281/vco-repo/archetype-catalog.xml -DrepoUrl=https://<orchestrator_server>:8281/vco-repo -Dmaven.repo.remote=https://<orchestrator_server>:8281/vco-repo -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

注意 Orchestrator Appliance を展開している場合は、Maven リポジトリにのみアクセスできます。

- 2 (オプション) HTTPS 経由でリポジトリにアクセスできない場合は、HTTP 経由でアクセスします。HTTP 経由でリポジトリにアクセスする場合や、有効な SSL 証明書が存在する場合は、-

`Dmaven.wagon.http.ssl.allowall=true` フラグを使用することなくプロジェクトを作成することができます。

```
mvn archetype:generate -DarchetypeCatalog=http://<orchestrator_server>:8280/vco-repo/archetype-catalog.xml -DrepoUrl=http://<orchestrator_server>:8280/vco-repo -Dmaven.repo.remote=http://<orchestrator_server>:8280/vco-repo -Dmaven.wagon.http.ssl.insecure=true
```

- 3 プロジェクト ディレクトリに移動してプラグインをビルドします。

```
cd <project_dir> && mvn clean install -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

ビルド処理が正常に実行されると、プラグインの `.dar` ファイルが DAR モジュールの `target/` ディレクトリ内に作成されます。

Maven のアーキタイプ

事前定義の Maven アーキタイプセットをテンプレートとして使用して、Orchestrator プラグインを開発することができます。

次の表で、Orchestrator で使用できるデフォルトの Maven アーキタイプについて説明します。

表 7-1. デフォルトの Maven アーキタイプ

アーキタイプ	説明
<code>com.vmware.o11n:011n-plugin-archetype-simple</code>	<code>com.vmware.o11n:011n-plugin-archetype-simple</code>
<code>com.vmware.o11n:011n-package-archetype</code>	コンテンツ専用の Maven プロジェクト。これを使用すると、パッケージをソース フォームで保持することができるため、RCS、差分、ポストプロセスなどとの対話の効率性を高めることができます。
<code>com.vmware.o11n:011n-client-archetype-rest</code>	Orchestrator REST API と通信してワークフローを呼び出す単純なコマンドライン ツール。
<code>com.vmware.o11n:011n-plugin-archetype-inventory</code>	インベントリの使用をデモンストレーションするプラグイン。このプラグインは、単一タイプのリポジトリ、アダプタ、およびファクトリを実装します。インベントリはディスク上のファイルに保存されます。
<code>com.vmware.o11n:011n-archetype-inventory-annotation</code>	注釈の上部に <code>vso.xml</code> 記述子が生成されているプラグイン。
<code>com.vmware.o11n:011n-archetype-spring</code>	Spring ベースの SDK を使用するプラグイン。DI 対応環境を提供し、標準のプラグイン API より高いレベルのサービスを追加します。
<code>com.vmware.o11n:011n-plugin-archetype-modeldriven</code>	ModelDriven でのプラグインのビルドに使用されるプラグイン スケルтонを生成するアーキタイプ。

Maven ベースのプラグイン開発のベスト プラクティス

Maven で作成した Orchestrator プラグインの提供プロセスを、一連のタスクを実行して改善することができます。

リポジトリマネージャの使用

より大きな組織でプラグインを作成する場合、エンタープライズ リポジトリ マネージャを使用して、プロキシ リポジトリに追加するデフォルトの Orchestrator Appliance リポジトリをセットアップします。中央リポジトリを使用することで、プロジェクト コラボレーションの管理およびプラグインが改善されます。新しいリポジトリの最初のビルドが完了したら、リポジトリ マネージャによって Orchestrator Appliance リポジトリからアーティファクトがキャッシュに格納され、デフォルトのリポジトリはオフにすることができます。

ロック ワークフロー

プラグイン作業においてすべてのワークフローが正しいことを検証した後、意図しない変更を防止するためワークフローをロックします。ワークフローをロックすることで、プラグインの基本機能が損なわれることがなくなります。特定の目的によりユーザーがデフォルトのワークフローを変更する必要がある場合は、元のワークフローのコピーを作成してから、そのコピーを編集します。

ロックされたワークフローでリリース ビルドを生成するには、2 つの方法があります。

- `-DallowedMask=vf` パラメータを Maven に渡します。
- `pom.xml` を編集し、`vf` に対する `allowedMask` パラメータの値を変更します。

```
<allowedMask>vf</allowedMask>
```

パッケージ署名証明書の使用

自己署名の証明書または認証局により署名された証明書を使用して、プラグインの統一性と信頼性を確実にすることができます。証明書を JDK のキーツールでインポートして、`_dunesrsa_alias_` エイリアスにあるキーストアに保存してください。

キーストア ファイルのパスとキーストアのパスワードを指定するには、2 つの方法があります。

- コマンドライン パラメータ `-DkeystoreLocation` および `-DkeystorePassword` を `<MAVEN_OPTS>` 変数に定義します。
- `pom.xml` ファイルを編集して、値を手動で入力します。たとえば、

```
<keystore><path to the keystore file></keystore>
<storepass><keystore password></storepass>
```

キーストアがインポートされていない場合、`.package` ファイルは `archetype.keystore` ファイルで署名されます。