使用 VMware vRealize Orchestrator 进行开发

vRealize Orchestrator 7.6

vmware®

您可以从 VMware 网站下载最新的技术文档：

https://docs.vmware.com/cn/。

如果您对本文档有任何意见或建议，请将反馈信息发送至：

docfeedback@vmware.com

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

威睿信息技术（中国）有限公司
北京办公室
北京市
朝阳区新源南路 8 号
启皓北京东塔 8 层 801
www.vmware.com/cn

上海办公室
上海市
淮海中路 333 号
瑞安大厦 804-809 室
www.vmware.com/cn

广州办公室
广州市
天河路 385 号
太古汇一座 3502 室
www.vmware.com/cn

目录

使用 VMware vRealize Orchestrator 开发

《使用 VMware vRealize Orchestrator 开发》提供了有关开发自定义 VMware® vRealize Orchestrator 工作流和操作的信息和说明。

此外，文档还包含需要运行脚本的 Orchestrator 元素的相关信息并提供了 JavaScript 示例。《使用 VMware vRealize Orchestrator 开发》还提供了有关如何创建资源和软件包的说明。

目标读者

本文档主要面向想要创建自定义 Orchestrator 工作流和操作以及自定义构建块的开发人员。

注　本指南中介绍的步骤基于 vRealize Orchestrator 旧版客户端的用户界面。

开发工作流

1

您可以在 Orchestrator 客户端界面开发工作流。工作流开发需要使用工作流编辑器、内置 Mozilla Rhino JavaScript 引擎以及 Orchestrator 和 vCenter Server API。

- 工作流的主要概念

  工作流由架构、属性和参数组成。工作流架构是工作流的主要组成部分，用于定义所有工作流元素及其相互间的逻辑关系。工作流属性和参数是工作流在传输数据使用的变量。Orchestrator 会在工作流每次运行时保存一个工作流令牌，用于记录工作流本次运行的详细信息。

- 工作流开发流程的各阶段

  工作流开发流程包含一系列阶段。您可以遵循不同的阶段顺序或跳过某个阶段，具体取决于正在开发的工作流类型。例如，可以创建不含自定义脚本的工作流：

- 开发工作流的最佳做法

  VMware 推荐了几种由多个用户在群集环境中开发 Orchestrator 工作流的最佳做法。

- Orchestrator 客户端的访问权限

  默认情况下，只有 Orchestrator 管理员 LDAP 组的成员才能访问 Orchestrator 客户端。

- 在开发过程中测试工作流

  您可以在开发过程的任何节点测试工作流，即使尚未完成工作流或包含结束元素也是如此。

- 创建和编辑工作流

  您可以在 Orchestrator 客户端中创建工作流，并在工作流编辑器中对其进行编辑。工作流编辑器是 Orchestrator 客户端用于开发工作流的 IDE。

- 提供常规工作流信息

  您可在工作流编辑器的**常规**选项卡中，提供工作流名称和说明，定义工作流行为的属性和相关方面，设置版本号，检查签名，设置用户权限等。

- 定义属性和参数

  在创建工作流后，您必须定义工作流的全局属性、输入参数和输出参数。

- 工作流架构

  工作流架构是工作流的一种图形表现形式，采用互相连接的工作流元素流程图显示工作流的情况。工作流架构用于定义工作流的逻辑流。

- 开发插件

  Orchestrator 可通过其开放插件架构与管理解决方案进行集成。您可以使用 Orchestrator 客户端运行并创建插件工作流并访问插件 API。

- 工作流启动时获取用户的输入参数

  如果某工作流需要输入参数，它在启动时会打开一个对话框，让用户输入必需的输入参数值。您可以在工作流编辑器的**展示**选项卡中设置该对话框的内容、布局或展示。

- （可选） 在工作流运行时请求用户交互

  工作流在运行时有时需要外部源中的其他输入参数。这些输入参数可来自其他应用程序或工作流，或者用户可直接提供。

- 在工作流中调用工作流

  工作流可以在运行期间调用其他工作流。一个工作流可以启动另一个工作流，可能因为它需要将其他工作流的运行结果作为自己的输入参数，或者因为它可以启动一个工作流并让其独立运行。工作流也可在未来给定时间启动另一工作流或同时启动多个工作流。

- 在选择的对象上运行工作流

  您可以在选择的对象上运行工作流来自动执行重复任务。例如，您可以创建一个工作流来为虚拟机文件夹中的所有虚拟机拍摄快照，或者创建一个工作流来关闭指定主机上所有虚拟机的电源。

- 开发长时间运行的工作流

  处于等待状态的工作流会消耗系统资源，因为该工作流会不断轮询能向其提供响应的对象。如果您知道某个工作流可能需要很长时间才能收到所需的响应，您可以向该工作流添加长时间运行的工作流元素。

- 配置元素

  配置元素是一个属性列表，您可用这些属性在整个 Orchestrator 服务器范围内配置各种常量。

- 工作流用户权限

  Orchestrator 会定义您对用户组可应用的权限级别，从而允许或拒绝其访问工作流。

- 验证工作流

  Orchestrator 提供了工作流验证工具。验证工作流有助于识别工作流中的错误，并检查相邻元素之间的数据流动是否正确。

- 调试工作流

  Orchestrator 提供了工作流调试工具。您可以调试工作流以在任何活动开始时检查输入和输出参数以及属性、以编辑模式在工作流运行时替换参数或属性值，以及从上次失败活动中恢复工作流。

- 运行工作流

  Orchestrator 工作流会根据事件的逻辑流来运行。

- 恢复失败的工作流运行

  如果工作流失败，Orchestrator 会提供一个选项，可从上次失败的活动中恢复工作流运行。

- 生成工作流文档

  您可以将在任意时间选择的工作流或工作流文件夹的相关文档以 PDF 格式导出。

- 使用工作流版本历史记录

  您可以使用版本历史记录将工作流恢复为先前保存的状态。您可以将工作流状态恢复为较早或较新的工作流版本。您还可以比较当前状态的工作流与已保存版本的工作流之间的差异。

- 还原已删除工作流

  您可以还原已从工作流库中删除的工作流。
- 开发简单示例工作流

  开发简单示例工作流可以展示工作流开发过程中最常见的步骤。
- 开发复杂工作流

  开发复杂示例工作流可以展示工作流开发过程中最常见的步骤和更高级的场景，例如创建自定义决策和循环。

工作流的主要概念

工作流由架构、属性和参数组成。工作流架构是工作流的主要组成部分，用于定义所有工作流元素及其相互间的逻辑关系。工作流属性和参数是工作流在传输数据使用的变量。Orchestrator 会在工作流每次运行时保存一个工作流令牌，用于记录工作流本次运行的详细信息。

工作流参数

工作流在运行时会接收输入参数并生成输出参数。

输入参数

大多数工作流需要一组特定输入参数才能运行。输入参数是工作流启动时要处理的参数。用户、应用程序、其他工作流或某项操作会将输入参数传递给工作流，以便工作流启动时对其进行处理。

例如：如果某工作流在重置虚拟机时，则会需要将该虚拟机的名称作为输入参数。

输出参数

工作流的输出参数表示工作流运行的结果。输出参数会在工作流或工作流元素运行时发生变化。工作流在运行时可接收其他工作流的输出参数作为输入参数。

例如，如果某个工作流在创建虚拟机快照时，则其输出参数即是所产生的快照。

工作流属性

工作流元素会处理作为输入参数收到的数据，并将结果数据设置为工作流属性或输出参数。

只读工作流属性会充当工作流的全局常量。可写属性充当工作流的全局变量。

您可以使用这些属性在工作流的元素之间传输数据。您可以通过以下方式获取属性：

- 在创建工作流时定义属性
- 将工作流元素的输出参数设置为工作流属性
- 从配置元素继承属性

工作流架构

工作流架构是一种图形表现形式，采用互相连接的工作流元素流程图显示工作流的情况。工作流架构是工作流最重要的元素，因为它确定了工作流的逻辑。

工作流展示

用户运行工作流时，会在工作流展示中提供工作流的输入参数值。整理工作流展示时，请考虑工作流的类型和输入参数数量。

工作流令牌

工作流令牌可显示工作流是正在运行还是已经运行。

工作流是一个进程的抽象描述，定义了一系列常规步骤以及一组所需常规输入参数。当您使用一组真实输入参数运行工作流时，您会收到此抽象工作流的实例，其行为依据您给定的特定输入参数而定。这一已完成或正在运行工作流的特定实例就称为工作流令牌。

工作流令牌属性

工作流令牌属性是工作流令牌运行时使用的指定参数。工作流令牌属性是工作流全局属性以及您用来运行工作流令牌的特定输入和输出参数的汇总。

工作流开发流程的各阶段

工作流开发流程包含一系列阶段。您可以遵循不同的阶段顺序或跳过某个阶段，具体取决于正在开发的工作流类型。例如，可以创建不含自定义脚本的工作流：

通常，开发工作流时需要经过以下阶段。

1 创建新工作流或从标准库中创建现有工作流的副本。

2 提供有关该工作流的一般信息。

3 定义该工作流的输入参数。

4 布局并链接工作流架构以定义工作流的逻辑流。

5 将每个架构元素的输入和输出参数与工作流属性绑定。

6 编写适用于可编辑脚本任务元素或自定义决策元素的脚本。

7 创建工作流展示，定义用户运行工作流时将看到的输入参数对话框的布局。

8 验证工作流。

开发工作流的最佳做法

VMware 推荐了几种由多个用户在群集环境中开发 Orchestrator 工作流的最佳做法。

- 每个开发人员都具有用于创建和开发工作流的专有测试独立 Orchestrator 实例。
- 工作流将在共享源代码管理系统上另存为 maven 项目。

- 为确保 Orchestrator 生产部署获得最佳性能，最好在已调度的窗口中导入工作流。
- 将工作流导入 Orchestrator 群集时，使用其本地主机名或 IP 地址而不是负载平衡器虚拟服务器的地址将 Orchestrator 客户端连接到其中一个节点。

  注 工作流的任何修改在下一次工作流运行时即可生效。

Orchestrator 客户端的访问权限

默认情况下，只有 Orchestrator 管理员 LDAP 组的成员才能访问 Orchestrator 客户端。

Orchestrator 管理员可将 Orchestrator 客户端的访问权限授予其他用户组，方法是至少设置**查看**权限。

为使您能访问 Orchestrator 客户端，管理员必须将您添加到 Orchestrator 管理员 LDAP 组，或为您所在的组设置**查看**、**检查**、**编辑**、**执行**或**管理**权限。

在开发过程中测试工作流

您可以在开发过程的任何节点测试工作流，即使尚未完成工作流或包含结束元素也是如此。

默认情况下，Orchestrator 会在运行工作流之前先检查工作流是否有效。您可以在工作流开发期间取消激活自动验证，从而对部分工作流进行测试运行。

注 在完成工作流开发后，请务必重新激活自动验证。

步骤

1 在 Orchestrator 客户端菜单中，单击**工具 > 用户首选项**。

2 单击**工作流**选项卡。

3 取消选中**在运行工作流前进行验证**复选框。

您即取消激活了工作流自动验证。

创建和编辑工作流

您可以在 Orchestrator 客户端中创建工作流，并在工作流编辑器中对其进行编辑。工作流编辑器是 Orchestrator 客户端用于开发工作流的 IDE。

您通过编辑现有工作流的方式打开工作流编辑器。

- 创建工作流

  您可以在 Orchestrator 客户端的工作流层次结构列表中创建工作流。
- 编辑工作流

  您可以对工作流进行编辑，更改现有工作流或开发新的空工作流。

- 编辑标准库中的工作流

  Orchestrator 提供了一个工作流标准库，供您在虚拟基础架构中用来自动执行操作。标准库中的工作流锁定为只读状态。

- 工作流编辑器选项卡

  工作流编辑器包含各种选项卡，您可在其上编辑工作流的组件。

创建工作流

您可以在 Orchestrator 客户端的工作流层次结构列表中创建工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 （可选） 右键单击工作流层次结构列表的根，或列表中的文件夹，然后选择**添加文件夹**以创建新工作流文件夹。

4 （可选） 输入新文件夹的名称。

5 右键单击新文件夹或现有文件夹，然后选择**新建工作流**。

6 为新工作流命名，然后单击**确定**。

一个新的空工作流即会创建在所选的文件夹中。

后续步骤

您可以编辑该工作流。

编辑工作流

您可以对工作流进行编辑，更改现有工作流或开发新的空工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 展开工作流层次结构列表以导航到要编辑的工作流。

4 右键单击要编辑的工作流并选择**编辑**以打开该工作流。

工作流编辑器会打开此工作流进行编辑。

编辑标准库中的工作流

Orchestrator 提供了一个工作流标准库，供您在虚拟基础架构中用来自动执行操作。标准库中的工作流锁定为只读状态。

若要编辑标准库中的工作流，您必须创建该工作流的副本。您可以编辑副本工作流或自定义工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 （可选） 右键单击工作流文件夹的层次结构列表的根目录，并选择**新建文件夹**来创建包含要编辑的工作流的文件夹。

4 展开标准工作流的**库**层次结构列表以导航到要编辑的工作流。

5 右键单击要编辑的工作流。

编辑选项显示为灰色。工作流为只读。

6 右键单击工作流并选择**复制工作流**。

7 提供副本工作流的名称。

默认情况下，Orchestrator 会将副本工作流命名为 *workflow_name* 的副本。

8 单击**工作流文件夹**值以搜索要在其中保存副本工作流的文件夹。

选择在步骤 3 中创建的文件夹。如果未创建文件夹，请选择不在标准工作流库中的文件夹。

9 单击**是**或**否**将工作流版本历史记录复制到副本。

选项	描述
是	原始工作流的版本历史记录复制到副本中。
否	副本的版本恢复到 0.0.0。

10 单击**复制**以复制工作流。

11 右键单击副本工作流并选择**编辑**。

此时会打开工作流编辑器。您可以编辑副本工作流。

您已复制了标准库中的工作流。您可以编辑副本工作流。

工作流编辑器选项卡

工作流编辑器包含各种选项卡，您可在其上编辑工作流的组件。

表 1-1. 工作流编辑器选项卡

选项卡	描述
常规	编辑工作流名称、提供工作流用途说明、设置版本号、查看用户权限、定义 Orchestrator 服务器重新启动时的工作流行为以及定义工作流的全局属性。
输入	定义工作流在运行时所需的参数。这些输入参数是工作流要处理的数据。工作流的行为会根据这些参数发生改变。
输出	定义工作流完成运行后生成的值。其他工作流或操作在运行时可以使用这些值。

表 1-1. 工作流编辑器选项卡 （续）

选项卡	描述
架构	构建工作流。您可以从**架构**选项卡左侧的工作流调色板中拖动相应的工作流架构元素来构建工作流。单击架构图中的元素可以在**架构**选项卡的下半部分定义和编辑元素的行为。
展示	定义用户运行工作流时显示的用户输入对话框的布局。您可以将参数和属性编排到展示步骤和组中，从而在输入参数对话框中简化参数的标识。您可以设置参数属性，以此定义用户在展示中提供输入参数时的限制。
参数参考	查看哪些工作流元素在工作流的逻辑流中使用了这些属性和参数。此选项卡还会显示您在**展示**选项卡上对这些参数和属性定义的限制。
工作流令牌	查看每个工作流运行的详细信息。此信息包含工作流的状态、工作流的运行用户、当前元素的业务状态以及工作流启动和结束的时间与日期。
事件	查看工作流运行时发生的各个事件的相关信息。此信息包含事件的说明、该事件的触发用户、事件的类型和来源以及发生的时间和日期。
权限	对用户或用户组设置与工作流进行交互的权限。

提供常规工作流信息

您可在工作流编辑器的**常规**选项卡中，提供工作流名称和说明，定义工作流行为的属性和相关方面，设置版本号，检查签名，设置用户权限等。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**常规**选项卡。

2 单击**版本**数字以设置该工作流的的版本号。

将打开**版本注释**对话框。

3 为此版本的工作流键入备注并单击**确定**。

例如，如果您刚刚创建工作流，键入**初始创建**。

新版本工作流即已创建成功。您可在稍后将工作流状态恢复到此版本。

4 设置**服务器重新启动行为**的值，定义 Orchestrator 服务器重新启动时工作流的行为。

- 保留**恢复工作流运行**默认值，则工作流会从服务器停止时的中断处恢复运行。
- 单击**恢复工作流运行**并选择**不恢复工作流运行（设置为失败）**，以在 Orchestrator 服务器重新启动时阻止工作流重新启动。

如果工作流严重依赖其运行环境，则应阻止工作流重新启动。例如，如果某个工作流运行时需要特定的 vCenter Server，而您对 Orchestrator 进行了重新配置将其连接到了其他 vCenter Server 上，则在重新启动服务器后重新启动工作流将导致工作流失败。

5 在**说明**文本框中输入工作流的详细说明。

6 单击工作流编辑器底部的**保存**。

工作流编辑器左下角会显示一条绿色消息，确认更改已成功保存。

您即定义了工作流行为的相关方面、设置了版本号并定义了用户可对该工作流执行的操作。

后续步骤

您必须定义工作流的属性和参数。

定义属性和参数

在创建工作流后，您必须定义工作流的全局属性、输入参数和输出参数。

工作流属性会存储工作流内部处理的数据。工作流输入参数是用户或其他工作流等外部源提供的数据。工作流输出参数是工作流完成运行时提供的数据。

- 定义工作流参数

  您可以使用输入和输出参数在工作流中传入和传出数据。

- 定义工作流属性

  工作流属性是工作流处理的数据。

- 属性和参数命名限制

  您可以在工作流运行时使用 OGNL 表达式来动态确定输入参数。Orchestrator OGNL 解析程序在 OGNL 处理过程中会使用您无法用于工作流属性或参数名称的特定关键字。

定义工作流参数

您可以使用输入和输出参数在工作流中传入和传出数据。

您可以在工作流编辑器中定义工作流参数。这些输入参数是工作流运行所需的初始数据。用户在运行工作流时会提供输入参数的值。输出参数是工作流运行完成后返回的数据。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的相应选项卡。

   - 单击**输入**创建输入参数。
   - 单击**输出**创建输出参数。

2 在参数选项卡中单击右键，然后选择**添加参数**。

3 单击参数名称进行更改。

输入参数默认名称为 `arg_in_X`，输出参数默认名称为 `arg_out_X`，*X* 表示数字。

4 （可选） 若要更改参数类型的值，请单击该值，然后从可用值列表中选择一个值。

参数类型的值默认为“字符串”。

5 在**说明**文本框中添加参数的说明。

6 （可选） 如果决定参数应该为属性而不是参数，请右键单击该参数，然后选择**作为属性移动**将参数更改为属性。

您即定义了工作流的输入或输出参数。

后续步骤

在定义工作流参数后，您可以构建工作流架构。

定义工作流属性

工作流属性是工作流处理的数据。

注 您也可以在创建工作流架构时在工作流架构元素中定义工作流属性。通常在创建会处理该属性的工作流架构元素时对属性进行定义会更为方便。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**常规**选项卡。

此时在**常规**选项卡下半部分会显示属性窗格。

2 在属性窗格中单击右键，然后选择**添加属性**。

属性列表中会显示一个新属性，默认类型为“字符串”。

3 单击该属性名称进行更改。

默认名称为 `attX`，*X* 表示数字。

注 工作流属性不得与工作流的任何参数同名。

4 单击属性类型，从可能值列表中选择新类型。

默认属性类型为“字符串”。

5 单击该属性值，根据属性类型设置或选择一个值。

6 在**说明**文本框中添加该属性的说明。

7 如果属性为常量而非变量，请单击属性名称左侧的复选框，将其值设置为只读。

锁定图标标识了只读复选框列。

8 （可选） 如果决定该属性应该为输入或输出参数，而不是属性，请右键单击属性，然后选择**作为输入/输出参数移动**将属性更改为参数。

您即定义了工作流的属性。

后续步骤

您可以定义工作流的输入和输出参数。

属性和参数命名限制

您可以在工作流运行时使用 OGNL 表达式来动态确定输入参数。Orchestrator OGNL 解析程序在 OGNL 处理过程中会使用您无法用于工作流属性或参数名称的特定关键字。

将预留的 OGNL 关键字用作属性名称前缀不会中断 OGNL 处理过程。例如，您可以命名参数 `trueParameter`。预留的关键字不区分大小写。

您无法在工作流属性和参数名称中使用以下关键字。

表 1-2. 属性和参数名称中的禁用关键字

禁用关键字	禁用关键字	禁用关键字
■ abstract	■ eof	■ _memberAccess
■ back_char_esc	■ esc	■ native
■ back_char_literal	■ exponent	■ package
■ boolean	■ export	■ private
■ byte	■ extends	■ public
■ char	■ false	■ root
■ char_literal	■ final	■ short
■ class	■ flt_literal	■ static
■ _classResolver	■ flt_suff	■ string_esc
■ const	■ ident	■ string_literal
■ context	■ implements	■ synchronized
■ debugger	■ import	■ this
■ dec_digits	■ in	■ _traceEvaluations
■ dec_flt	■ int	■ true
■ default	■ int_literal	■ _typeConverter
■ delete	■ interface	■ volatil
■ digit	■ _keepLastEvaluation	■ with
■ double	■ _lastEvaluation	■ WithinBackCharLiteral
■ dynamic_subscript	■ letter	■ WithinCharLiteral
■ enum	■ long	■ WithinStringLiteral

工作流架构

工作流架构是工作流的一种图形表现形式，采用互相连接的工作流元素流程图显示工作流的情况。工作流架构用于定义工作流的逻辑流。

- 查看工作流架构

  您可以在 Orchestrator 客户端中某个工作流的**架构**选项卡中查看该工作流的架构。

- 在工作流架构中构建工作流

  工作流架构由按序排列的架构元素组成。工作流架构元素是工作流的构建块，可用于表示决策、脚本任务、操作、异常处理程序、甚至其他工作流。

- 架构元素

  工作流编辑器会在**架构**选项卡上的菜单中提供这些工作流架构元素。您可以使用**架构**选项卡中的架构元素来构建工作流。

- 架构元素属性

  架构元素具有多种属性，您可在工作流调色板的**架构**选项卡中对这些属性进行定义和编辑。

- 链接和绑定

  元素之间的链接用于确定工作流的逻辑流。绑定会将输入和输出参数与工作流属性进行绑定，使用其他元素的数据来填充相关元素。

- 决策

  工作流可以实现相关决策函数，根据布尔 `true` 或 `false` 语句定义不同的操作运行方向。

- 异常处理

  异常处理会捕捉架构元素运行过程中产生的任何错误。异常处理将定义错误发生时架构元素的行为。

- 使用错误处理程序

  如果在特定工作流架构元素中发生错误，您可以使用标准错误处理程序来定义行为。如果发生了标准错误处理程序未发现的错误，您可以使用全局错误处理程序来定义行为。

- Foreach 元素和复合类型

  您可以在开发的工作流中插入 Foreach 元素以运行对参数或属性数组进行迭代的子工作流。为改进工作流的理解性和可读性，您可以将类型不同但在逻辑上相连的多个工作流参数编入单个类型（即复合类型）。

- 将切换活动添加到工作流

  您可以将基本切换活动添加到根据工作流属性或参数定义了切换案例的工作流架构。

查看工作流架构

您可以在 Orchestrator 客户端中某个工作流的**架构**选项卡中查看该工作流的架构。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 在工作流层次结构列表中导航到工作流。

3 单击工作流。

此时在右侧窗格会显示有关该工作流的信息。

4 在右侧窗格中选择**架构**选项卡。

您会看到该工作流的图形表现。

在工作流架构中构建工作流

工作流架构由按序排列的架构元素组成。工作流架构元素是工作流的构建块，可用于表示决策、脚本任务、操作、异常处理程序、甚至其他工作流。

您可在工作流编辑器中构建工作流，只需将工作流编辑器左侧工作流调色板中的架构元素拖放到工作流架构图中即可。

编辑工作流架构

您可以创建一系列架构元素来定义工作流的逻辑流，从而构建工作流。

默认情况下，工作流架构中的所有元素都相互链接。元素间的链接采用箭头表示。将新元素添加至工作流架构时，必须将其拖放到箭头上或未与下一元素链接的现有工作流元素上。将工作流元素添加到架构后，您可以删除现有链接并创建新链接，以定义工作流的逻辑流。

您可以将现有工作流架构中的某个元素或选择的多个元素复制到正在编辑的工作流架构中。请参见复制工作流架构元素。

工作流架构至少需要一个**结束工作流**元素，但可以有多个元素。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 将架构元素从左侧窗格的**通用**菜单中拖放到工作流架构中。

3 双击已拖放到工作流架构中的元素，键入合适的名称，然后按 Enter 键。

所键入的元素名称在工作流上下文中必须唯一。

您不能重命名**等待定时器**、**等待事件**、**结束工作流**或**出现异常**元素。

4 （可选） 右键单击架构中的元素，然后选择**复制**。

5 （可选） 在架构中的适当位置单击右键，选择**粘贴**。

复制并粘贴现有架构元素，可快速将相似元素添加至架构。所复制元素的所有设置（业务状态除外）会显示在粘贴的元素中。对粘贴的元素进行相应调整。

6 将架构元素从**基本**、**日志**或**网络**菜单中拖放至工作流架构中。

您可以在**基本**、**日志**或**网络**菜单中编辑元素名称。您不能编辑元素的脚本。

7 将架构元素从**基本**菜单中拖放至工作流架构中。

将操作或工作流拖放至工作流架构时，系统会显示一个对话框，您可以从中搜索要插入的操作或工作流。

8 在**筛选器**文本框中，输入要在工作流中插入的操作或工作流的名称或部分名称。

与搜索匹配的工作流或操作会显示在对话框中。

9 双击工作流或操作将其选中。

您即在工作流架构中插入了此工作流或操作。

10 重复此步骤，直到将全部所需的架构元素添加至工作流架构中。

后续步骤

定义已添加至工作流架构中的元素属性，将它们全部链接和绑定到一起。

复制工作流架构元素

您可以将现有工作流架构中的某个元素或选择的多个元素复制到正在编辑的工作流架构中。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 在左侧窗格中，选择要从中复制架构元素的工作流。

- 单击**所有工作流**并从工作流层次结构列表中选择工作流。
- 在搜索文本框中输入工作流名称并按 Enter 键。

3 右键单击选定的工作流并选择**打开**。

此时会显示包含工作流属性的窗口。

4 在工作流窗口中，单击**架构**选项卡。

5 选择一个或多个工作流架构元素，右键单击所选内容，然后选择**复制**。

6 在正在编辑的工作流的**架构**选项卡中，右键单击并选择**粘贴**。

这样，一个工作流中的工作流架构元素即复制到另一个工作流。

后续步骤

您必须将复制的架构元素链接并绑定至现有工作流架构。

升级输入和输出参数

您可以将某个子元素的输入和输出参数升级到父工作流。

您可以升级在工作流编辑器的**常规**选项卡中定义的自定义属性。您只有通过将输入参数替换为匹配类型的属性，才能升级预定义的属性。

注 如果升级预定义的属性并为其分配自定义值，则会创建副本属性以避免覆盖原始属性值。副本属性会保留原始属性的名称并增加属性名称结尾处的数字值。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 向工作流架构添加工作流或操作元素。

以下通知会显示在架构窗格的顶部。

是否要将活动参数作为输入/输出添加到当前工作流？(Do you want to add the activity's parameters as input/output to the current workflow?)

3 在通知上，单击**设置**。

此时会显示包含可用选项的弹出窗口。

4 选择每个输入参数的映射类型。

选项	描述
输入	参数映射到输入工作流参数。
跳过	参数映射到 NULL 值。
值	参数映射到属性，其中属性包含您可以在“值”列中进行设置的值。

5 选择每个输出参数的映射类型。

选项	描述
输出	参数映射到输出工作流参数。
跳过	参数映射到 NULL 值。
本地变量	参数映射到属性。

6 单击**升级**。

参数即升级到父工作流。

修改搜索结果

您可以使用**搜索**文本框来查找工作流或操作等元素。如果搜索只返回了部分结果，您可以修改搜索应返回的结果数量。

使用搜索查找元素时，绿色消息框表示该搜索已列出了所有结果。黄色消息框则表示仅列出了部分搜索结果。

步骤

1 （可选） 如果正在工作流编辑器中编辑工作流，单击**保存并关闭**以退出该编辑器。

2 在 Orchestrator 客户端菜单中，单击**工具 > 用户首选项**。

3 单击**常规**选项卡。

4 在**查找器最大大小**文本框中键入搜索应返回的结果数量。

5 在“用户首选项”对话框中单击**保存并关闭**。

您即修改了搜索应返回的结果数量。

架构元素

工作流编辑器会在**架构**选项卡上的菜单中提供这些工作流架构元素。您可以使用**架构**选项卡中的架构元素来构建工作流。

表 1-3. 架构元素和图标

架构元素名称	描述	图标	在工作流编辑器中的位置
起始工作流	工作流的起点。所有工作流都包含该元素。一个工作流只能包含一个起始元素。起始元素包含一个输出，不含输入，并且无法从工作流架构中移除。		始终显示在**架构**选项卡上
可编辑脚本任务	您定义的常规用途任务。您可在该元素中编写 JavaScript 函数。		**通用**工作流调色板
决策	一个布尔函数。决策元素获取一个输入参数，并返回 `true` 或 `false`。元素作出的决策的类型取决于输入参数类型。决策元素能将工作流分为不同的运行方向，具体取决于决策元素收到的输入参数。如果收到的输入参数对应预期的值，则工作流会按特定路径继续运行。如果输入不是预期的值，则工作流会按备选路径继续运行。		**通用**工作流调色板
自定义决策	一个布尔函数。自定义决策可获取多个参数并根据自定义脚本对其进行处理。返回 `true` 或 `false`。		**通用**工作流调色板
决策活动	一个布尔函数。决策活动会返回一个工作流，并将其输出参数绑定到 `true` 或 `false` 路径。		**通用**工作流调色板

表 1-3. 架构元素和图标 （续）

架构元素名称	描述	图标	在工作流编辑器中的位置
用户交互	能让用户将新输入参数传递到工作流。您可以设计用户交互元素对输入参数的请求方式，并对用户可提供的参数进行限制。您可以设置权限来确定用户可提供的输入参数。当正在运行的工作流到达用户交互元素时，会进入被动状态并提示用户进行输入。您可以设置一个超时时间段，用户必须在此时间段内提供输入。工作流会根据用户传回的数据恢复运行，但如果超时时段过期，则会返回异常。在等待用户响应时，工作流令牌会处于 `waiting` 状态。		**通用**工作流调色板
等待定时器	用于长时间运行的工作流。当正在运行的工作流到达等待定时器元素时，会进入被动状态。您需要设置一个绝对日期让工作流恢复运行。在等待该日期时，工作流令牌会处于 `waiting-signal` 状态。		**通用**工作流调色板
等待事件	用于长时间运行的工作流。当正在运行的工作流到达等待事件元素会，会进入被动状态。您需要定义工作流需要等待的触发器事件。在等待该事件时，工作流令牌会处于 `waiting-signal` 状态。		**通用**工作流调色板
结束工作流	工作流的终点。一个架构中可以包含多个结束元素，用来表示工作流的多个可能结果。结束元素包含一个输入，不含输出。当工作流到达“结束工作流”元素时，工作流令牌会进入 `completed` 状态。		**通用**工作流调色板
出现异常	创建异常并停止工作流。工作流架构中可以多次出现该元素。异常元素包含一个输入参数（其类型只能是字符串），不含输出参数。当工作流到达异常元素时，工作流令牌会进入 `failed` 状态。		**通用**工作流调色板
工作流备注	让您对工作流的各部分进行注释。您可以编写备注来说明工作流的各部分。您可以更改备注的背景颜色来区分不同工作流区域。工作流备注仅提供视觉信息，帮助您了解架构。		**通用**工作流调色板
操作元素	从 Orchestrator 操作库中调用操作。当工作流到达操作元素时，会调用和运行该操作。		**通用**工作流调色板
工作流元素	同时启动另一工作流。当工作流到达其架构中的某个工作流元素时，会运行该工作流，作为其进程的一部分。只有当被调用的工作流完成运行时，原始工作流才会继续运行。		**通用**工作流调色板
Foreach 元素	对数组中的每个元素运行工作流。例如，您可以在某个文件夹中的所有虚拟机上运行“重命名虚拟机”工作流。		**通用**工作流调色板

表 1-3. 架构元素和图标 （续）

架构元素名称	描述	图标	在工作流编辑器中的位置
异步工作流	异步启动工作流。当工作流到达某个异步工作流元素时，会启动该工作流同时继续运行。原始工作流无需等待被调用的工作流完成运行。		**通用**工作流调色板
调度工作流	创建任务以在设置的时间运行工作流，然后该工作流继续运行。		**通用**工作流调色板
嵌套工作流	同时启动多个工作流。您可以选择将本地工作流与位于不同 Orchestrator 服务器的远程工作流进行嵌套。您还可以运行具有不同凭据的工作流。工作流会等待所有嵌套的工作流完成运行，然后再继续运行。		**通用**工作流调色板
处理错误	处理某个特定工作流元素的错误。工作流可以通过创建异常、调用其他工作流或运行自定义脚本等方式来处理错误。		**通用**工作流调色板
默认错误处理程序	处理标准错误处理程序未发现的工作流错误。您可以使用任何可用的架构元素来处理这些错误。		**通用**工作流调色板
切换	根据工作流属性或参数，切换到备选工作流路径。		**通用**工作流调色板
预定义任务	不可编辑的脚本元素，用来执行工作流通常使用的标准任务。以下任务为预定义： **基本** ■ 睡眠 ■ 更改凭据 ■ 等待截止日期 ■ 等待自定义事件 ■ 发送自定义事件 ■ 增加计数器 ■ 减少计数器 **日志** ■ 系统日志 ■ 系统警告 ■ 系统错误 ■ 服务器日志 ■ 服务器警告 ■ 服务器错误 ■ 系统+服务器日志 ■ 系统+服务器警告 ■ 系统+服务器错误 **网络** ■ HTTP POST ■ HTTP GET		**基本**、**日志**和**网络**工作流调色板

架构元素属性

架构元素具有多种属性，您可在工作流调色板的**架构**选项卡中对这些属性进行定义和编辑。

编辑架构元素的全局属性

您可在元素的信息选项卡中定义架构元素的全局属性。

前提条件

验证工作流编辑器的**架构**选项卡是否包含元素。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 单击**编辑**图标 ( )，选择要编辑的元素。

此时系统会显示一个对话框，列出元素的各种属性。

3 单击**信息**选项卡。

4 在**名称**文本框内为架构元素输入一个名称。

这是将显示在工作流架构图中架构元素的名称。

5 在**交互**下拉菜单中，选择一个说明。

交互属性允许您在该元素与工作流以外对象的交互方式的标准说明之间进行选择。此属性仅供参考。

6 （可选） 在**业务状态**文本框内输入业务状态说明。

业务状态属性是对该元素功能的简要说明。工作流运行时，工作流令牌会显示每个元素运行时的业务状态。此功能用于跟踪工作流状态。

7 （可选） 在**说明**文本框中，输入架构元素的说明。

架构元素属性选项卡

在已拖放到工作流架构中的元素上单击，可以访问架构元素的属性。元素的属性会显示在工作流编辑器底部的选项卡中。

不同架构元素包含不同属性选项卡。

表 1-4. 每个架构元素的属性选项卡

架构元素属性选项卡	说明	应用到架构元素类型
属性	元素从外部资源获取的属性，例如用户、事件或定时器。这些属性可以是超时限制、时间和日期、触发器或用户凭据。	■ 用户交互 ■ 等待事件 ■ 等待定时器
决策	用于定义决策语句。决策元素收到的输入参数可能与决策语句匹配或不匹配，从而导致两种可能的操作。	决策

表 1-4. 每个架构元素的属性选项卡 （续）

架构元素属性选项卡	说明	应用到架构元素类型
结束工作流	停止工作流，原因可以是工作流已成功完成或是工作流遇到错误并返回异常。	■ 结束 ■ 异常
异常	此架构元素在遇到异常事件时应采取的行为。	■ 操作 ■ 异步工作流 ■ 异常 ■ 嵌套工作流 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务 ■ 用户交互 ■ 等待事件 ■ 等待定时器 ■ 工作流
外部输入	用户在工作流运行的特定时刻必须提供的输入参数。	用户交互
输入	该元素的“输入”绑定。“输入”绑定用于定义架构元素从工作流中其之前的元素接收输入的方式。	■ 操作 ■ 异步工作流 ■ 自定义决策 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务 ■ 工作流
信息	架构元素的常规属性和说明。**信息**选项卡显示的信息取决于架构元素的类型。	■ 操作 ■ 异步工作流 ■ 自定义决策 ■ 决策 ■ 嵌套工作流 ■ 注意 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务 ■ 用户交互 ■ 等待事件 ■ 等待定时器 ■ 工作流
输出	该元素的“输出”绑定。“输出”绑定用于定义架构元素将输出参数绑定到工作流属性或工作流输出参数的方式。	■ 操作 ■ 异步工作流 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务 ■ 工作流

表 1-4. 每个架构元素的属性选项卡 （续）

架构元素属性选项卡	说明	应用到架构元素类型
展示	用于定义工作流在运行中需要用户输入时用户所看到的输入参数对话框的布局。	用户交互
脚本	显示用于定义该架构元素行为的 JavaScript 函数。对于异步工作流、调度工作流和操作元素，该脚本为只读。对于可编辑脚本任务和自定义决策元素，您在该选项卡中编辑 JavaScript 即可。	■ 操作 ■ 异步工作流 ■ 自定义决策 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务
可视化绑定	采用图形方式显示此架构元素的参数和属性与工作流中该元素之前/之后元素的参数和属性的绑定方式。这是元素的“输入”和“输出”绑定的另一种表现形式。	■ 操作 ■ 异步工作流 ■ 预定义任务 ■ 调度工作流 ■ 可编辑脚本任务 ■ 工作流
工作流	选择要嵌套的工作流。	嵌套工作流

链接和绑定

元素之间的链接用于确定工作流的逻辑流。绑定会将输入和输出参数与工作流属性进行绑定，使用其他元素的数据来填充相关元素。

若要了解链接和绑定，必须了解工作流的逻辑流与数据流之间的区别。

工作流的逻辑流

工作流的逻辑流是工作流运行时从架构中的一个元素向下一个元素的运行进度。通过链接架构中的元素可以定义工作流的逻辑流。

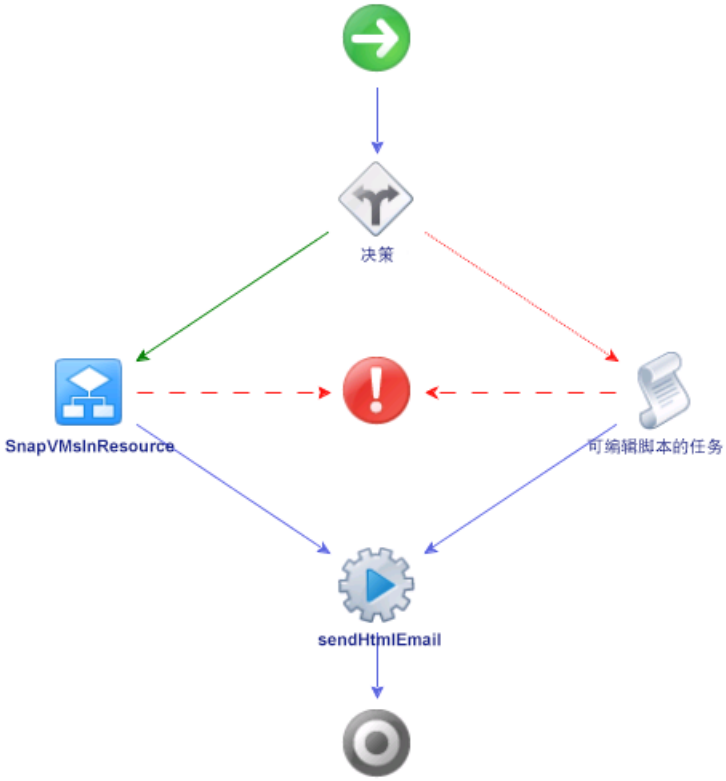
标准路径是工作流在所有元素正常运行时所采用的逻辑流路径。异常路径是某个元素未正常工作时工作流所采用的逻辑流路径。

工作流架构中的不同箭头表示此工作流所采用的不同逻辑流路径。

- 蓝色箭头表示工作流从一个元素进入下一个元素时所采用的标准路径。
- 绿色箭头表示布尔决策元素返回 `true` 时工作流所采用的路径。
- 红色点状虚线箭头表示布尔决策元素返回 `false` 时工作流所采用的路径。
- 红色虚线箭头表示工作流元素未正常工作时工作流所采用的异常路径。

下图显示了一个工作流架构示例，演示了工作流可采用的不同路径。

图 1-1. 通过工作流逻辑流的不同工作流路径



该工作流示例可采用以下逻辑流路径。

- 标准路径，`true` 决策结果，无异常。
  a 该决策元素返回 `true`。
  b `SnapVMsInResourcePool` 工作流运行成功。
  c `sendHtmlEmail` 操作运行成功。
  d 该工作流成功结束于 `completed` 状态。
- 标准路径，`false` 决策结果，无异常。
  a 该决策元素返回 `false`。
  b 可编辑脚本任务元素定义的操作运行成功。
  c `sendHtmlEmail` 操作运行成功。
  d 该工作流成功结束于 `completed` 状态。
- `true` 决策结果，异常。
  a 该决策元素返回 `true`。
  b `SnapVMsInResourcePool` 工作流发生错误。
  c 工作流返回异常并停止在 `failed` 状态。

- `false` 决策结果，异常。
  a 该决策元素返回 `false`。
  b 可编辑脚本任务元素定义的操作发生错误。
  c 工作流返回异常并停止在 `failed` 状态。

元素链接

链接用于连接架构元素并定义工作流从一个元素向下一个元素运行时的逻辑流。

工作流中的元素通常只能设置一个与另一元素的出站链接，以及一个与用于定义其异常行为的元素的异常链接。出站链接用于定义工作流的标准路径。异常链接用于定义工作流的异常路径。在多数情况下，单个架构元素可以接收来自多个元素的入站标准路径链接。

以下元素是对其之前语句的异常情况。

- “启动工作流”元素无法接收入站链接且无异常链接。
- 异常元素可以接收多个入站异常链接且无出站链接或异常链接。
- 决策元素有两个用于定义工作流应采用路径的出站链接，具体取决于决策结果为 `true` 或是 `false`。决策没有异常链接。
- “结束工作流”元素不能有出站链接或异常链接。

创建标准路径链接

标准路径链接用于确定工作流的正常运行。

将一个元素链接到另一个元素时，您始终需要按照元素在工作流中的运行顺序进行链接。您始终需要从首个运行的元素开始，创建两个元素之间的链接。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 验证工作流编辑器的**架构**选项卡是否包含元素。

步骤

1 将指针放在要与其他元素连接的元素上。

元素的右侧会显示一个蓝色箭头和一个红色箭头。

2 将指针放在蓝色箭头上。

此时蓝色箭头会放大。

3 单击蓝色箭头，按住鼠标左键，然后将指针移动到目标元素。

此时两个元素之间会显示一个蓝色箭头，并且目标元素周围会显示一个绿色矩形。

4 松开鼠标左键。

两个元素之间的蓝色箭头仍将保留。

标准路径此时即链接了两个元素。

后续步骤

元素已经连接，但您尚未定义数据流。您必须定义“输入”和“输出”绑定以绑定工作流属性的输入和输出数据。

工作流的数据流

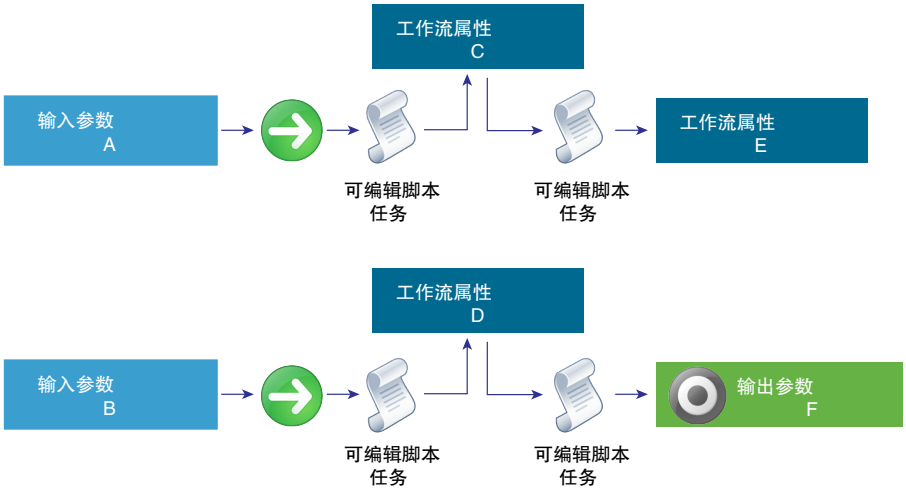
工作流的数据流是在每个工作流元素运行时，工作流元素的输入和输出参数与工作流属性之间的绑定方式。您通过使用架构元素间的绑定来定义工作流的数据流。

工作流架构中的元素在运行时需要以输入参数形式提供的数据。该元素通过与您创建工作流时所设置的工作流属性进行绑定，或与工作流前一个元素运行时所设置的属性进行绑定，获取所需的输入数据。

该元素将处理这些数据，很可能对其进行转换并以输出参数的形式生成运行结果。元素将生成的输出参数它所创建的新工作流属性进行绑定。架构中的其他元素可以与这些新的工作流属性进行绑定，作为自己的输入参数。工作流运行结束时可以生成相关属性作为其输出参数。

下图显示了一个非常简单的工作流。蓝色箭头表示元素链接情况和工作流的逻辑流。红线显示了工作流的数据流。

图 1-2. 工作流数据流示例



通过工作流的数据流如下。

1 工作流从输入参数 a 和 b 开始。

2 第一个元素处理参数 a，并将处理结果绑定到工作流属性 c。

3 第一个元素处理参数 b，并将处理结果绑定到工作流属性 d。

4 第二个元素将工作流属性 c 作为输入参数，对其进行处理，将生成的输出参数绑定到工作流属性 e。

5 第二个元素将工作流属性为输入参数，对其进行处理，并生成输出参数 f。

6 工作流结束，并生成工作流属性 f 作为其输出参数，即其运行结果。

元素绑定

您必须将所有工作流元素的输入和输出参数与工作流属性进行绑定。绑定将设置元素中的数据，定义元素的输出和异常行为。链接将定义工作流的逻辑流，而绑定将定义元素的数据流。

要设置元素中的数据、在处理后根据元素生成输出参数、处理元素运行时可能发生的任何错误，您必须设置元素绑定。

IN 绑定	设置架构元素的传入数据。将元素的本地输入参数与源工作流属性进行绑定。**输入**选项卡会列出“本地参数”列中的元素输入参数。**输入**选项卡会列出“源参数”列中需要与本地参数绑定的工作流属性。该选项卡还会显示参数的类型和说明。
OUT 绑定	元素停止运行时，更改工作流属性并生成输出参数。**输出**选项卡会列出“本地参数”列中的元素输出参数。**输出**选项卡会列出“源参数”列中需要与本地参数绑定的工作流属性。该选项卡还会显示参数的类型和说明。
异常绑定	如果元素在运行时遇到异常，则链接到异常处理程序。

IN 绑定会读取绑定源参数中的值。OUT 绑定会将这些值写入绑定的源参数。

您必须使用 IN 绑定将架构元素中使用的所有属性或输入参数与工作流属性进行绑定。如果元素在运行时更改了收到的输入参数值，您必须使用 OUT 绑定将这些参数与工作流属性进行绑定。将元素的输出参数与工作流元素进行绑定，可允许工作流架构中位于该元素之后的其他元素将这些输出参数作为自己的输入参数。

创建工作流时的常见错误是未绑定输出参数值，因而无法反映该元素对工作流属性做出的更改。

重要事项 当您添加某个需要输入和输出参数的元素且参数类型已在工作流中定义时，Orchestrator 会设置这些参数的绑定。您必须验证 Orchestrator 绑定的参数正确无误，防止工作流对元素可绑定的同一参数类型定义了不同的参数。

定义元素绑定

将不同元素链接以创建工作流的逻辑流程后，您可以定义元素绑定以定义每个元素对其所接收数据和所生成数据的处理方式

前提条件

验证在工作流编辑器的**架构**选项卡中存在工作流架构，而且您已在各元素之间创建了链接。

步骤

1 单击要对其设置绑定的元素的**编辑**图标 ( )。

此时系统会显示一个对话框，列出元素的各种属性。

2 单击**输入**选项卡。

输入选项卡的内容因所选元素类型而异。

- 如果选择了预定义的任务、工作流或操作元素，**输入**选项卡上会列出此元素类型可用的本地输入参数，但未设置绑定。

- 如果选择了另一种类型元素，在**输入**选项卡中单击右键并选择**绑定到工作流参数/属性**，即可从您为该工作流定义的输入参数和属性列表中进行选择。
- 如果所需的属性不存在，您可以在**输入**选项卡中单击右键并选择**绑定到工作流参数/属性 > 在工作流中创建参数/属性**创建该属性。

3 如果合适的参数已存在，则选择要绑定的输入参数，并单击**源参数**文本框中的**未设置**按钮。

此时系统会显示一个列表，列出可用于绑定的源参数和属性。

4 从建议列表中选择要与本地输入参数绑定的源参数。

5 （可选） 如果未定义要绑定的源参数，请单击参数选择对话框中的**在工作流中创建参数/属性**链接进行创建。

6 单击**输出**选项卡。

输出选项卡的内容因所选元素类型而异。

- 如果选择了预定义的任务、工作流或操作元素，**输出**选项卡上会列出此元素类型可用的本地输出参数，但未设置绑定。
- 如果选择了另一种类型元素，在**输出**选项卡中单击右键并选择**绑定到工作流参数/属性**，即可从您为该工作流定义的输出参数和属性列表中进行选择。
- 如果所需的属性不存在，您可以在**输入**选项卡中单击右键并选择**绑定到工作流参数/属性 > 在工作流中创建参数/属性**创建该属性。

7 选择要绑定的参数。

8 单击**源参数 > 未设置**按钮。

9 选择要与输入参数绑定的源参数。

10 （可选） 如果未定义要绑定的参数，请单击参数选择对话框中的**在工作流中创建参数/属性**按钮进行创建。

您即定义了元素接收的输入参数和生成的输出参数，并将它们与工作流属性和参数进行了绑定。

后续步骤

您可通过定义决策的方式在工作流路径中创建分叉。

决策

工作流可以实现相关决策函数，根据布尔 `true` 或 `false` 语句定义不同的操作运行方向。

决策是工作流中的分叉。工作流决策是根据由您本人、其他工作流、应用程序或工作流运行的环境所提供的输入而作出的。决策元素接收的输入参数值决定了工作流应选择的分叉。例如，工作流决策可能会收到给定虚拟机的电源状态作为其输入。如果虚拟机打开电源，工作流将选择某个路径通过其逻辑流。如果虚拟机关闭电源，工作流将选择不同的路径。

决策始终是布尔函数。每个决策的结果只能是 `true` 或 `false`。

自定义决策

自定义决策与标准决策的区别在于您在脚本中定义的决策语句。自定义决策会根据您定义的语句返回 `true` 或 `false`，如下示例所示。

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

创建决策元素链接

决策元素与工作流中的其他元素不同。决策元素只有 `true` 或 `false` 两个输出参数。决策元素没有异常链接。

前提条件

确认工作流编辑器的**架构**选项卡包含相应元素，至少应包含一个未与其他元素链接的决策元素。

步骤

1 将鼠标指针放在决策元素上，准备将其与两个用于定义工作流中两个可能分支的其他元素进行链接。

元素的右侧会显示一个蓝色箭头和一个红色箭头。

2 将指针放在蓝色箭头上，同时按住鼠标左键，将指针移动到目标元素。

此时两个元素之间会显示一个绿色箭头，并且目标元素会变为绿色。绿色箭头表示工作流获取了 `true` 路径，前提是决策元素所收到的输入参数或属性与决策语句匹配。

3 松开鼠标左键。

两个元素之间的绿色箭头仍将保留。您即定义了决策元素在收到预期值时工作流所获取的路径。

4 将指针放在决策元素上，按住鼠标左键，然后将指针移动到目标元素。

此时两个元素之间会显示一个红色点状虚线箭头，并且目标元素会变为绿色。红色箭头表示工作流获取了 `false` 路径，前提是决策元素收到的输入参数或属性与决策语句不匹配。

5 松开鼠标左键。

两个元素之间的红色点状虚线箭头仍将保留。您即定义了决策元素在收到意外输入时工作流所获取的路径。

您即定义了工作流所可能获取的 `true` 或 `false` 路径，具体取决于决策元素收到的输入参数或属性。

后续步骤

定义决策语句。请参见使用决策创建工作流分支。

删除链接的决策元素

从工作流架构删除链接的决策元素时，必须指定要删除的工作流路径。

前提条件

验证工作流编辑器的**架构**选项卡是否包含元素，应至少包含一个具有 `true` 或 `false` 路径的决策元素。

步骤

1 选择决策元素并按 Delete 键。

此时会显示包含可用选项的对话框。

2 选择要删除的决策分支。

选项	描述
成功分支	遵循 `true` 决策路径的决策元素和所有元素都会从工作流架构中删除。
失败分支	遵循 `false` 决策路径的决策元素和所有元素都会从工作流架构中删除。
所有分支	遵循上述两个决策路径的决策元素和所有元素都会从工作流架构中删除。
无	仅决策元素及其链接会从工作流架构中删除。遵循上述两个决策路径的所有元素都会保留在工作流架构中。

3 单击**确定**。

使用决策创建工作流分支

决策元素是简单的布尔函数，您可用来在工作流中创建分支。决策元素用于确定收到的输入是否匹配您设置的决策语句。作为此决策的一个函数，工作流会按两条可能路径中的一条继续运行。

前提条件

在定义决策前，请先确认您的决策元素已与工作流编辑器中架构内的其他两个元素进行了链接。

步骤

1 单击决策元素的**编辑**图标 (✏)。

此时系统会显示一个对话框，列出决策元素的各种属性。

2 单击对话框中的**决策**选项卡。

3 单击**未设置 (空)** 链接以选择该决策的源输入参数。

此时系统会显示一个对话框，列出此工作流中定义的所有属性和输入参数。

4 双击列表中的输入参数将其选中。

5 如果未定义要绑定的源参数，请单击参数选择对话框中的**在工作流中创建属性/参数**链接进行创建。

6 从下拉菜单中选择决策语句。

菜单中建议的语句是上下文关联的，并且根据所选择的输入参数类型而有所不同。

7 添加需要决策语句与之匹配的值。

根据您选择的输入类型和语句，可能会在值文本框中看到**未设置 (空)** 链接。单击此链接可获得一组预定义选择的值。否则，例如对于字符串，就是一个需要您输入值的文本框。

您即定义了决策元素的语句。决策元素收到输入参数时，会比较输入参数的值和语句中的值，并确定语句是 true 还是 false。

后续步骤

您必须设置工作流对异常的处理方式。

异常处理

异常处理会捕捉架构元素运行过程中产生的任何错误。异常处理将定义错误发生时架构元素的行为。

工作流中的所有元素（决策、起始和结束元素除外）都包含一个特定的输出参数类型，专门用来处理异常。如果某个元素在运行过程中发生错误，它可以给异常处理程序发送错误信号。异常处理程序会捕捉该错误并根据收到的错误作出响应。如果您定义的异常处理程序无法处理特定的错误，您可以将元素的异常输出参数绑定到异常元素，该异常元素会结束处于失效状态的工作流。

异常可以充当工作流元素内的 `try` 和 `catch` 顺序。如果不需要处理某个元素中的特定异常，您不需要绑定该元素的异常输出参数。

异常的输出参数类型始终是一个 `errorCode` 对象。

创建异常绑定

元素可设置绑定，用于定义工作流在遇到该元素的错误时应采取的行为。

前提条件

验证工作流编辑器的**架构**选项卡是否包含元素。

步骤

1 将指针放在要为其定义异常绑定的元素上。

元素的右侧会显示一个红色箭头。

2 将指针放在红色箭头上，直到其放大为止，按住鼠标左键，然后将红色箭头拖放到目标元素上。

红色虚线箭头会链接两个元素。目标元素定义了工作流所链接的元素遇到错误时工作流应采取的行为。

3 单击异常处理元素所链接元素的**编辑**图标 (✏)。

4 单击架构元素属性选项卡中的**异常**选项卡。

5 若要设置**输出异常绑定**值，请单击**未设置**。

- 从异常属性绑定对话框中选择一个参数绑定到异常输出参数，然后单击**选择**。
- 单击**在工作流中创建参数/属性**以创建异常输出参数。

6 单击用于定义异常处理行为的目标元素。

7 单击架构元素属性选项卡中的**输入**选项卡。

8 单击**绑定到工作流参数/属性**图标 ( )。

此时系统会显示用于选择输入参数的对话框。

9 选择异常输出参数，然后单击**选择**。

10 单击架构元素属性选项卡中的异常处理元素的**输出**选项卡。

11 定义异常处理元素的行为。

- 单击**绑定到工作流参数/属性**图标 ( ) 以选择异常处理元素要生成的输出参数。
- 单击**脚本**选项卡，并使用 JavaScript 定义异常处理元素的行为。

您即定义了元素对异常的处理方式。

后续步骤

您必须定义如何在用户运行工作流时获取用户的输入参数。

使用错误处理程序

如果在特定工作流架构元素中发生错误，您可以使用标准错误处理程序来定义行为。如果发生了标准错误处理程序未发现的错误，您可以使用全局错误处理程序来定义行为。

添加错误处理程序至工作流

您可以将错误处理程序添加到工作流元素，从而定义在工作流运行期间如何处理特定工作流元素中的错误。您仅可以将错误处理程序添加到没有指定错误路径的工作流元素。

重要事项 包含**处理错误**元素的工作流不兼容 Orchestrator 5.5.x 或更早版本。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**处理错误**元素拖放到工作流架构中适当的元素。

此时将显示一个对话框。

2 在对话框的下拉菜单中，选择如何处理错误。

选项	描述
出现异常	发生错误时，即会出现异常。您可以修改异常绑定。
调用工作流	发生错误时，会运行选定工作流。
自定义脚本	发生错误时，会运行自定义脚本。

3 单击**选择**。

错误处理程序即添加到工作流。工作流到达此元素时，会在结束运行前先执行选定操作。

将全局错误处理程序添加到工作流

您只能将一个全局错误处理程序添加到一个工作流架构，从而定义在工作流运行期间如何处理未被标准错误处理程序发现的错误。您只能将一个全局错误处理程序添加到一个工作流架构。

重要事项　包含**默认错误处理程序**元素的工作流不兼容 Orchestrator 5.5.x 或更早版本。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1. 将**默认错误处理程序**元素拖放到工作流架构。
2. （可选） 在**默认错误处理程序**元素和**出现异常**元素之间添加架构元素，指定如何处理全局工作流错误。

全局错误处理程序即添加到工作流。如果工作流中出现的错误未被标准错误处理程序发现，则全局错误处理程序会在结束工作流运行前，先执行指定的操作。

Foreach 元素和复合类型

您可以在开发的工作流中插入 Foreach 元素以运行对参数或属性数组进行迭代的子工作流。为改进工作流的理解性和可读性，您可以将类型不同但在逻辑上相连的多个工作流参数编入单个类型（即复合类型）。

使用 Foreach 元素

Foreach 元素会对输入参数或属性数组迭代运行子工作流。您可以选择要对其运行子工作流的数组，并可在运行工作流时传递此类数组的元素值。子工作流的运行次数可与您在数组中定义的元素数量一样多。

如果您拥有包含属性数组的配置元素，可以运行工作流，在 Foreach 元素中对这些属性进行迭代。

例如，假设您想要重命名文件夹中的 10 台虚拟机。为此，您必须在工作流中插入 Foreach 元素并将重命名虚拟机工作流定义为元素中的子工作流。重命名虚拟机工作流会采用两个输入参数，即虚拟机及其新名称。您可以将这些参数升级为当前工作流的输入参数，这样它们就会成为重命名虚拟机工作流要迭代的数组。运行工作流时，您可以在文件夹中指定 10 台虚拟机及其新名称。每次工作流运行时，都会从虚拟机数组中选取一个元素并从虚拟机的新名称数组中选取一个元素。

使用复合类型

复合类型是一组类型不同但逻辑上相连的输入参数或属性。在 Foreach 元素中，您可以将一组参数绑定为复合值。这样，Foreach 元素就会在工作流每次后续运行时一次性采用分组的参数值。

例如，假设您要重命名一台虚拟机。您需要虚拟机对象及其新名称。如果要重命名多台虚拟机，则您需要两个数组，一个是虚拟机数组，另一个是虚拟机名称数组。这两个数组不会显式连接。复合类型可让您拥有一个数组，其中每个元素都同时包含虚拟机及其新名称。这样，如果有多个值，那么这两个参数之间的连接是显式指定，而不是由工作流架构暗示。

注 如果工作流包含 vSphere Web Client 中的复合类型，则您无法运行该工作流。

定义 Foreach 元素

如果想要通过在每次后续运行时为其参数或属性传递不同值来多次运行某个子工作流，您可以在父工作流中插入 Foreach 元素。

插入 Foreach 元素时，您必须至少选择一个 Foreach 元素要对其进行迭代的数组。数组元素对于每次后续工作流运行都可以拥有不同的值。

如果子工作流包含输出参数，您应选择要在其中累积工作流输出的 Foreach 元素的输出参数，以便子工作流也对其进行迭代。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 在工作流编辑器中，选择**架构**选项卡。

2 通过**常规**菜单，将 Foreach 元素拖进工作流架构。

3 从选择器对话框中选择工作流。

以下通知会显示在架构窗格的顶部。

是否要将活动参数作为输入/输出添加到当前工作流？(Do you want to add the activity's parameters as input/output to the current workflow?)

4 在通知上，单击**设置**。

此时会显示包含可用选项的弹出窗口。

5 选择每个输入参数的映射类型。

选项	描述
输入	参数映射到输入工作流参数。
跳过	参数映射到 NULL 值。
值	参数映射到属性，其中属性包含您可以在“值”列中进行设置的值。

6 选择每个输出参数的映射类型。

选项	描述
输出	参数映射到输出工作流参数。
跳过	参数映射到 NULL 值。
本地变量	参数映射到属性。

7 单击**升级**。

8 右键单击 Foreach 元素并选择**同步 > 同步展示**。

此时将显示确认对话框。

9 单击**确定**将 Foreach 元素的展示传播到当前工作流。

对话框会显示有关操作结果的信息。

10 在**输入**选项卡中，验证子工作流的参数已添加为类型数组的元素。

11 在**输出**选项卡中，验证子工作流的参数已添加为类型数组的元素。

您即在工作流中定义了 Foreach 元素。Foreach 元素运行的工作流会将参数视为来自您已定义的参数或属性数组中的每个元素。

对于未定义为数组的参数或属性，工作流会在每次后续运行中采用相同的值。

示例： 使用 Foreach 元素重命名虚拟机

您可以使用 Foreach 元素一次性重命名多台虚拟机。您需要在工作流中插入 Foreach 元素并将 vm 和 newName 参数升级为当前工作流的输入。这样，在您运行工作流时，指定要重命名的虚拟机和虚拟机新名称。虚拟机会作为元素随附在您为 vm 参数创建的数组中。虚拟机的新名称会随附在您为 newName 参数创建的数组中。

在 Foreach 元素中定义复合类型

您可以将逻辑连接的多个工作流参数编组为一个新类型，即复合类型。您可以使用 Foreach 元素将一组参数绑定为复合值，从而在单个数组中连接多个数组的参数。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 验证工作流中是否有 Foreach 元素。

步骤

1 选择 Foreach 元素的**入**或**出**选项卡。

2 选择要在复合类型中与其他本地参数编组的本地参数。

3 单击**入**或**出**选项卡顶部的**将一组参数绑定为复合值**。

4 在绑定窗格中，选择要编组为复合类型的参数。

5 选择**绑定为迭代器**。

您即设置了 Foreach 元素对复合类型数组进行迭代。

6 单击**接受**。

您即定义了复合类型并确保了工作流会对此复合类型数组进行迭代。编组为复合类型的参数会被命名为 *composite_type_name.parameter_name*。例如，如果创建 `snapshots` 复合类型，则在类型中编组的参数可以是 `snapshots.vm[in-parameter]` 或 `snapshots.name[in-parameter]`。复合类型数组中的每个元素都包含在复合类型中编组的每个参数的单一实例。

示例： 重命名虚拟机

假设您想要一次性重命名 10 台虚拟机。为此，您需要在工作流中插入 Foreach 元素并在元素中选择“重命名”虚拟机工作流。创建复合类型以明确连接 `vm` 和 `newName` 参数。将复合类型绑定为迭代器，这样就创建了同时包含 `vm` 和 `newName` 参数的单个数组。

将切换活动添加到工作流

您可以将基本切换活动添加到根据工作流属性或参数定义了切换案例的工作流架构。

每个切换活动都可以包含多个切换案例。每个切换案例都由属性或参数相关的条件定义。如果满足条件，则工作流运行会切换到您定义的相应工作流元素。如果未满足任何指定的条件，则工作流运行会切换到您定义的默认工作流元素。

重要事项 包含**切换**元素的工作流不兼容 Orchestrator 5.5.x 或更早版本。

前提条件

验证工作流编辑器的**架构**选项卡是否包含元素。

步骤

1 将**切换**元素拖放到工作流架构中适当的元素。

2 单击**切换**元素的**编辑**图标 (✏)。

3 在**案例**选项卡中，添加或删除切换案例。

您可以更改切换案例的优先级。

4 定义每个切换案例的条件。

5 选择每个切换案例的对应工作流元素。

6 选择要切换的目标默认工作流元素。

7 单击**关闭**。

8 单击**保存**。

您即定义了切换案例条件和工作流路径。

开发插件

Orchestrator 可通过其开放插件架构与管理解决方案进行集成。您可以使用 Orchestrator 客户端运行并创建插件工作流并访问插件 API。

插件概览

Orchestrator 插件必须包含一套标准组件并符合标准架构。以下这些做法有助于您创建各种插件，最大限度地利用种类丰富的外部技术。

- Orchestrator 插件的结构

  Orchestrator 插件具有通用结构，即由实施了特定功能的各种层类型组成。

- 将外部 API 公开至 Orchestrator

  您可以创建 Orchestrator 插件，将外部产品的 API 公开至 Orchestrator 平台。您可以为任何公开 API 的技术创建插件，其中该 API 可被映射到 Orchestrator 可以使用的 JavaScript 对象中。

- 插件的组件

  插件由一套标准组件组成，可以向 Orchestrator 平台公开相关插件技术中的各种对象。

- vso.xml 文件的角色

  您使用 `vso.xml` 文件将插件技术的对象、类、方法和属性映射到 Orchestrator 清单对象、脚本类型、脚本类、脚本方法和属性。`vso.xml` 文件还定义了插件的配置和启动行为。

- 插件适配器的角色

  插件适配器是插件进入 Orchestrator 服务器的入口点。插件适配器可以充当 Orchestrator 服务器中插件技术的数据存储、创建插件工厂以及管理插件技术中发生的事件。

- 插件工厂的角色

  插件工厂用于定义 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。

- 查找器对象的角色

  查找器对象识别并查找插件技术中受管对象类型的特定实例。Orchestrator 可以通过在查找器对象上运行工作流，修改其在插件技术中找到的对象并与之交互。

- 脚本对象角色

  脚本对象是插件技术中对象的 JavaScript 表现形式。插件中的脚本对象会显示在 Orchestrator JavaScript API 中，您可以将其用于工作流和操作中的脚本元素。

- 事件处理程序的角色

  事件即 Orchestrator 在插件技术中发现的对象的状态或属性的更改。Orchestrator 通过实施事件处理程序来监控事件。

Orchestrator 插件的结构

Orchestrator 插件具有通用结构，即由实施了特定功能的各种层类型组成。

Orchestrator 插件的底部三层（即基础架构类、封装类和脚本对象），实现了插件技术和 Orchestrator 之间的连接。

Orchestrator 插件的用户可见部分为顶部三层，即操作、构建块和高级别工作流。

图 1-3. Orchestrator 插件的结构

基础架构类	一组提供了插件技术和 Orchestrator 之间连接的类。基础架构类包含了根据插件定义（例如插件工厂、插件适配器等）实现的类。基础架构类还包含了为通用任务和对象（例如帮助程序、缓存、清单等）提供功能的类。
封装类	一组将插件技术的对象模型调整为要在 Orchestrator 中公开的对象模型的类。
脚本对象	提供了插件技术中封装类、方法和属性等访问权限的 JavaScript 对象类型。在 `vso.xml` 文件中，您定义了插件技术中的哪些封装类、属性和方法将公开到 Orchestrator。
操作	一组可直接在工作流和脚本任务中使用的 JavaScript 函数。操作可使用多个输入参数并拥有单个返回值。
构建块工作流	一组涵盖了所有要随插件一同提供的常规功能的工作流。通常，构建块工作流代表编排技术的用户界面中的操作。构建块工作流可直接使用或包含在高级别工作流内。
高级别工作流	一组涵盖了插件特定功能的工作流。您可以提供高级别工作流来满足具体要求或显示插件使用情况的复杂示例。

将外部 API 公开至 Orchestrator

您可以创建 Orchestrator 插件，将外部产品的 API 公开至 Orchestrator 平台。您可以为任何公开 API 的技术创建插件，其中该 API 可被映射到 Orchestrator 可以使用的 JavaScript 对象中。

插件会将 Java 对象和方法映射到被添加至 Orchestrator 脚本 API 的 JavaScript 对象。如果外部技术公开了 Java API，您可以将 API 直接映射到 JavaScript，以便 Orchestrator 将其用于工作流和操作。

您可以使用 WSDL（Web 服务定义语言）、REST（表述性状态转移）或消息传递服务将公开的 API 与 Java 对象集成，从而以非 Java 语言为公开了 API 的应用程序创建插件。随后将集成的 Java 对象映射到 JavaScript 以供 Orchestrator 使用。

插件技术独立于 Orchestrator。即使您只有（例如 Java 存档，即 JAR 文件中）二进制代码而不是源代码的访问权限，您也可以为外部产品创建 Orchestrator 插件。

插件的组件

插件由一套标准组件组成，可以向 Orchestrator 平台公开相关插件技术中的各种对象。

插件的主要组件是插件适配器、插件工厂和事件实现。您需要将插件适配器、插件工厂和事件实现中所定义的对象和操作映射到 Orchestrator 对象，相关映射关系保存在一个名为 `vso.xml` 的 XML 定义文件中。`vso.xml` 文件会将来自插件技术的对象和功能函数映射到 Orchestrator JavaScript API 中显示的 JavaScript 脚本对象。`vso.xml` 文件还会将来自插件技术的对象类型映射到相应的查找器，这些查找器会显示在 Orchestrator **清单**选项卡中。

插件由以下组件组成。

插件模块	即插件本身，由一组 Java 类、`vso.xml` 文件和软件包进行定义，通过软件包中的各种工作流和操作与通过该插件访问的各种对象进行交互。插件模块为必需组件。
插件适配器	用于定义 Orchestrator 服务器和插件技术之间的接口。适配器是插件在 Orchestrator 平台中应用时的入口点。适配器可用于创建插件工厂、管理插件的加载和卸载、并管理插件技术中各对象所发生的事件。插件适配器为必需组件。
插件工厂	用于定义了 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。适配器会针对 Orchestrator 和插件技术之间打开的客户端会话创建相应的插件工厂。通过该工厂，您可以在所有客户端连接之间共享一个会话，也可以对每个客户端连接打开一个会话。插件工厂为必需组件。
配置	Orchestrator 并不定义插件对其配置的标准存储方式。您可以使用 Windows 注册表或静态配置文件来存储配置信息，也可将信息存储在数据库或 XML 文件中。Orchestrator 插件可以通过在 Orchestrator 客户端中运行配置工作流的方式进行配置。
查找器	即一套交互规则，用于定义 Orchestrator 在插件技术中查找对象和表示对象的方式。查找器会从插件技术向 Orchestrator 公开的一组对象中检索相关对象。您可在 `vso.xml` 文件中定义各对象之间的关系以允许在对象网络中自由浏览。Orchestrator 通过**清单**选项卡来表示插件技术的对象模型。如果需要向 Orchestrator 公开插件技术中的对象，则查找器为必需组件。
脚本对象	即 JavaScript 对象类型，用于向插件技术中对象、操作和属性等提供访问权限。脚本对象可定义 Orchestrator 通过 JavaScript 访问插件技术对象模型时的访问方式。您可通过 `vso.xml` 文件将插件技术的类和方法映射到 JavaScript 对象。您可以访问 Orchestrator 脚本 API 中的 JavaScript 对象，并将这些对象集成到 Orchestrator 脚本任务、操作和工作流中。如果需要向

	Orchestrator JavaScript API 添加脚本类型、类和方法，则脚本对象为必需组件。
清单	即插件技术中的对象实例，Orchestrator 会使用 Orchestrator 客户端**清单**视图中显示的查找器对这些实例进行查找定位。您可以通过运行工作流的方式对清单中的对象执行相应的操作。清单为可选组件。您可以创建这样一个插件，只向 Orchestrator JavaScript API 添加脚本类型和类，而不在清单中公开任何对象实例。
事件	插件技术中某个对象的状态更改。Orchestrator 可以被动侦听插件技术中所发生的事件。Orchestrator 还可以在插件技术中主动触发某些事件。事件为可选组件。

vso.xml 文件的角色

您使用 `vso.xml` 文件将插件技术的对象、类、方法和属性映射到 Orchestrator 清单对象、脚本类型、脚本类、脚本方法和属性。`vso.xml` 文件还定义了插件的配置和启动行为。

`vso.xml` 文件执行以下主体角色。

启动和配置行为	定义了插件的启动方式，并确定插件定义的任何配置实现的位置。加载插件适配器。
清单对象	定义了插件在插件技术中访问的对象类型。插件工厂实现的查找器方法确定这些对象实例的位置，并在 Orchestrator 清单中显示。
脚本类型	将脚本类型添加到 Orchestrator JavaScript API 以表示清单中不同的对象类型。您可以将这些脚本类型用作工作流中的输入参数。
脚本类	将类添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些类。
脚本方法	将方法添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些方法。
脚本属性	将插件技术中的对象属性添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些属性。

插件适配器的角色

插件适配器是插件进入 Orchestrator 服务器的入口点。插件适配器可以充当 Orchestrator 服务器中插件技术的数据存储、创建插件工厂以及管理插件技术中发生的事件。

若要创建插件适配器，您需要创建将可实现该 `IPluginAdaptor` 接口的 Java 类。

您创建的插件适配器类可以管理插件工厂、事件以及插件技术中的触发器。`IPluginAdaptor` 接口提供您可用来执行这些任务的方法。

插件适配器执行以下主要角色。

创建工厂	插件适配器最重要的角色就是对 Orchestrator 与插件技术之间的每个连接加载或卸载一个插件工厂实例。插件适配器类会调用 `IPluginAdaptor.createPluginFactory()` 方法来创建可实现该 `IPluginFactory` 接口的类的实例。
管理事件	插件适配器是 Orchestrator 服务器和插件技术之间的接口。插件适配器会管理 Orchestrator 在插件技术对象上执行或观察的事件。适配器通过事件发布程序来管理各种事件。事件发布程序是 `IPluginEventPublisher` 接口的实例，该接口由适配器通过调用 `IPluginAdaptor.registerEventPublisher()` 方法而创建。事件发布程序会在插件技术的对象上设置触发器和计量器，从而在对象上发生特定事件或对象值传递特定阈值时允许 Orchestrator 启动定义的操作。类似地，您可以定义 `PluginTrigger` 和 `PluginWatcher` 实例，从而定义在长时间运行工作流中“等待事件”元素所等待的事件。
设置插件名称	您在 `vso.xml` 文件中为插件输入一个名称。插件适配器会从 `vso.xml` 文件中获取该名称并将其发布到 Orchestrator 客户端的**清单**视图中。
安装许可证	您可以调用方法来安装插件技术在适配器实现过程中所需的任何许可证文件。

有关 `IPluginAdaptor` 接口及其所有方法，以及插件 API 的所有其他类的完整详细信息，请参见 Orchestrator 插件 API 参考。

插件工厂的角色

插件工厂用于定义 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。

若要创建插件工厂，您必须从 Orchestrator 插件 API 实现并扩展 `IPluginFactory` 接口。您创建的插件工厂类用于定义查找器函数，Orchestrator 可通过这些函数来访问插件技术中的对象。该工厂允许 Orchestrator 服务器按对象 ID、按对象与其他对象的关系以及搜索查询字符串来查找对象。

插件工厂执行以下主要任务。

查找对象	您可以创建相关函数，按名称和类型来查找对象。使用 `IPluginFactory.find()` 方法按名称和类型查找对象。
查找与其他对象相关的对象	您可以创建相关函数，按给定关系类型来查找与给定对象相关的对象。关系可在 `vso.xml` 文件中定义。您还可以创建相关查找器，按给定关系类型查找与所有父对象相关的从属子对象。实现 `IPluginFactory.findRelation()` 方法按给定关系类型来查找与给定父对象相关的任何对象。实现 `IPluginFactory.hasChildrenInRelation()` 方法来发现父对象实例是否至少存在一个子对象。
定义相关查询以根据自己的条件来查找对象	您可以创建对象查找器，实现您定义的查询规则。实现 `IPluginFactory.findAll()` 方法从而当工厂调用此方法时，会查找所有满足您定义的查询规则的对象。您会在 `QueryResult` 对象中获取 `findAll()` 方法的结果，该对象列出了找到的与您所定义查询规则匹配的所有对象。

有关 IPluginFactory 接口及其所有方法、以及插件 API 的所有其他类的更多信息，请参见 Orchestrator 插件 API 参考。

查找器对象的角色

查找器对象识别并查找插件技术中受管对象类型的特定实例。Orchestrator 可以通过在查找器对象上运行工作流，修改其在插件技术中找到的对象并与之交互。

插件技术中给定受管对象类型的每个实例都必须具有唯一标识符，以便 Orchestrator 查找器对象可以找到它们。插件技术为对象实例提供字符串形式的唯一标识符。工作流在运行时，Orchestrator 会将其找到的对象的唯一标识符设置为工作流属性值。需要给定类型对象作为输入参数的工作流会在该对象类型的特定实例上运行。

插件添加到 Orchestrator JavaScript API 的查找器对象会将插件名称作为其名称前缀。例如，vCenter Server API 中的 VirtualMachine 受管对象类型会在 Orchestrator 中显示为 VC:VirtualMachine JavaScript 类型。

例如，Orchestrator 会实施将虚拟机 id 属性用作其唯一标识符的查找器对象，从而通过 vCenter Server 插件访问特定 VC:VirtualMachine 实例。您可以将此对象实例作为属性值传递到工作流元素。

Orchestrator 插件会将插件技术的对象映射到 vso.xml 文件 <finder> 元素中对应的 Orchestrator 查找器对象。<finder> 元素确定了插件技术中可获取对象特定实例唯一标识符的方法或函数。<finder> 元素还定义了对象之间的关系，从而按照对象之间的关联方式来查找对象。

查找器对象会显示在包含这些对象的插件下的 Orchestrator **清单**选项卡中。

脚本对象角色

脚本对象是插件技术中对象的 JavaScript 表现形式。插件中的脚本对象会显示在 Orchestrator JavaScript API 中，您可以将其用于工作流和操作中的脚本元素。

插件中的脚本对象在 Orchestrator JavaScript API 中显示为 JavaScript 模块、类型和类。大多数查找器对象具有脚本对象表现形式。JavaScript 类可将方法和属性添加到 Orchestrator JavaScript API，后者表示插件技术 API 中对象的方法和属性。插件技术提供了对象、类型、类、属性和独立于 Orchestrator 的方法的实现。例如，vCenter Server 插件将 vCenter Server API 中的所有对象表示为 Orchestrator JavaScript API 中的 JavaScript 对象，并以 JavaScript 的形式表示 vCenter Server API 定义的所有类、方法和属性。您可以使用 vCenter Server 脚本类以及其在 Orchestrator 脚本函数中定义的方法和属性。

例如，vCenter Server API 的 VirtualMachine 受管对象类型由 VC:VirtualMachine 查找器发现并在 Orchestrator JavaScript API 中显示为 VcVirtualMachine JavaScript 类。Orchestrator JavaScript API 中的 VcVirtualMachine JavaScript 类将所有相同方法和属性定义为 vCenter Server API 中的 VirtualMachine 受管对象。

Orchestrator 插件会将插件技术中的对象、类型、类、属性和方法映射到 vso.xml 文件 <scripting-objects> 元素中对应的 Orchestrator JavaScript 对象、类型、类、属性和方法。

事件处理程序的角色

事件即 Orchestrator 在插件技术中发现的对象的状态或属性的更改。Orchestrator 通过实施事件处理程序来监控事件。

Orchestrator 插件可让您通过多种方式监控插件技术中的事件。Orchestrator 插件 API 可让您创建以下类型的事件处理程序来监控插件技术中的事件。

侦听器	被动监控插件技术中对象的状态更改。插件技术或插件实现定义了侦听器监控的事件。侦听器不会启动事件，但会在事件发生时通知 Orchestrator。侦听器会通过轮询插件技术或接收插件技术的通知来检测事件。事件发生后，等待事件的 Orchestrator 策略或工作流可通过启动 Orchestrator 服务器中的操作来作出响应。侦听器组件为可选。
策略	监控插件技术中的特定事件并在事件发生时启动 Orchestrator 服务器中的操作。策略可以监控策略触发器和策略计量器。策略触发器定义了插件技术中的事件，即在事件发生后，使正在运行的策略启动 Orchestrator 服务器中的操作（例如运行工作流）。策略计量器定义了插件技术中某个对象属性的值范围，即超出该范围后，Orchestrator 会启动操作。策略为可选。
工作流触发器	如果正在运行的工作流包含“等待事件”元素，则当其到达该元素时，会挂起运行并等待插件技术中发生事件。工作流触发器定义了工作流中“等待事件”元素所等待的插件技术中事件。您可以向观察程序注册工作流触发器。工作流触发器为可选。
观察程序	代表工作流中的“等待事件”元素，观察插件技术中特定事件的工作流触发器。事件发生后，观察程序会通知等待该事件的任何工作流。观察程序为可选。

插件的内容和结构

Orchestrator 插件必须包含一组标准组件并遵守标准文件结构。对于符合标准文件结构的插件，必须包含特定文件夹和文件。

若要创建 Orchestrator 插件，您需要定义 Orchestrator 如何访问并与插件技术中的对象交互。并且，您需要将插件技术的所有对象和函数映射到 `vso.xml` 文件中对应的 Orchestrator 对象和函数。

`vso.xml` 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。插件在插件技术中发现的每个对象都必须具有您提供的唯一标识符。您需要在 `vso.xml` 文件内的 `finder` 元素以及对象元素中定义对象名称。

插件可以交付为标准 Java 存档文档 (JAR) 或 ZIP 文件，但无论哪种交付方式，文件名都必须使用 `.dar` 扩展名进行重命名。

注 您可以使用 Orchestrator 控制中心将 DAR 文件导入 Orchestrator 服务器。

- 定义 vso.xml 文件中的应用程序映射

  `vso.xml` 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或是在 Orchestrator **清单**选项卡中显示为查找器对象。

- vso.xml 插件定义文件的格式

  vso.xml 文件用于定义 Orchestrator 服务器与插件技术的交互方式。vso.xml 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。

- 命名插件对象

  您必须为该插件在插件技术中发现的每个对象提供唯一标识符。您需要分别在 vso.xml 文件中的 <finder> 元素和 <object> 元素中定义对象名称。

- 插件对象命名约定

  在命名插件中的所有对象时，必须遵守 Java 类命名约定。

- 插件的文件结构

  插件必须遵守标准文件结构，并且必须包含某些特定文件夹和文件。作为标准 Java 存档 (JAR) 或 ZIP 文件交付的插件，必须使用 .dar 扩展名对其重命名。

定义 vso.xml 文件中的应用程序映射

vso.xml 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或是在 Orchestrator **清单**选项卡中显示为查找器对象。

vso.xml 文件向 Orchestrator 服务器提供以下信息：

- 插件的版本、名称和说明
- 插件技术的类和关联插件适配器的参考
- 在启动 Orchestrator 服务器时初始化插件
- 脚本类型，用来在插件技术中表示对象类型
- 对象类型之间的关系，用于定义对象在 Orchestrator 清单中的显示方式
- 脚本类，将插件技术中的对象和操作映射到 Orchestrator JavaScript API 中的函数和对象类型
- 枚举，定义适用于特定类型的所有对象的常量值列表
- Orchestrator 在插件技术中监视的事件

vso.xml 文件必须遵守 Orchestrator 插件的 XML 架构定义。您可以在 VMware 支持站点上访问架构定义。

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

关于 vso.xml 文件所有元素的说明，请参见 vso.xml 插件定义文件的元素。

vso.xml 插件定义文件的格式

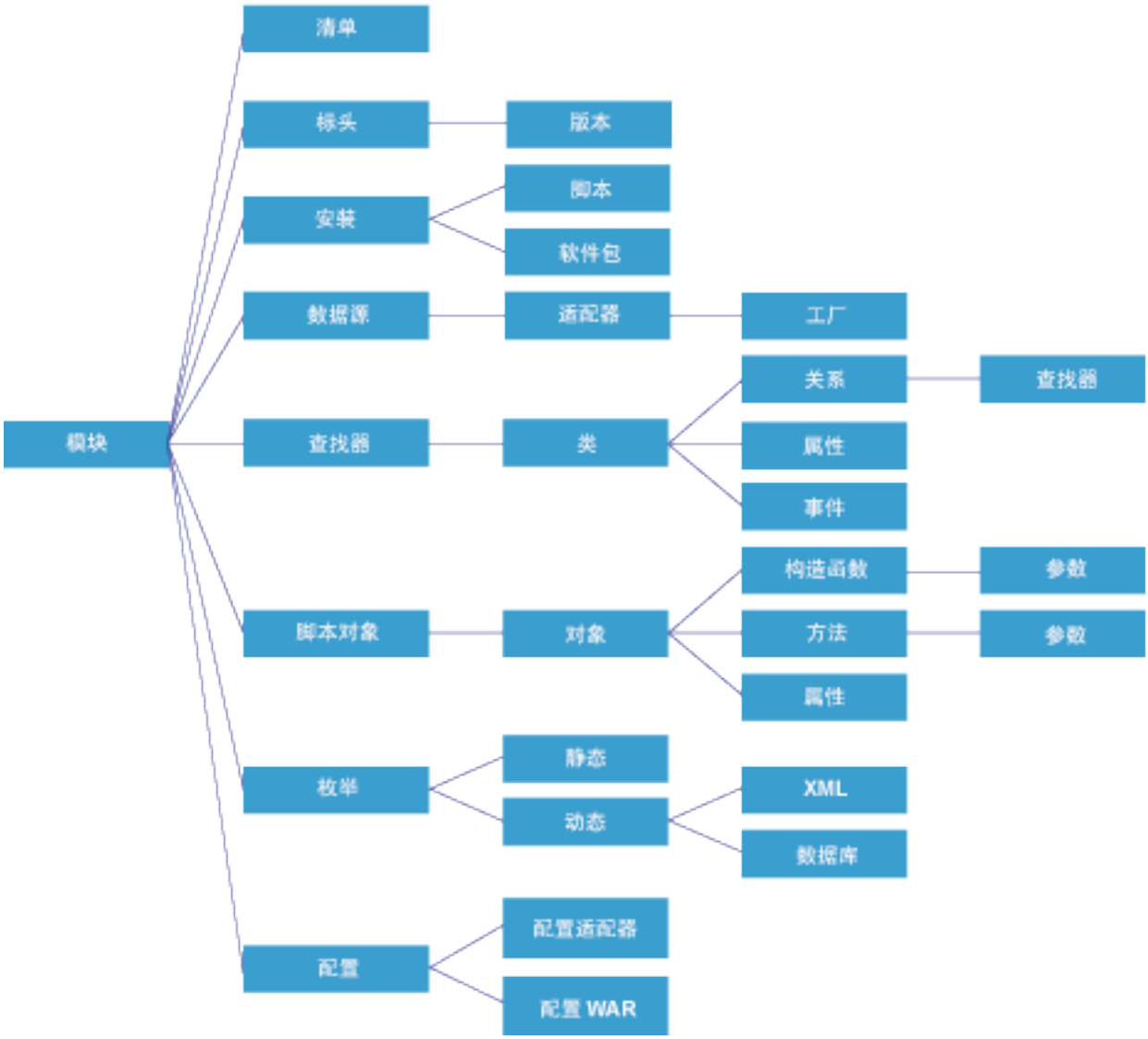
vso.xml 文件用于定义 Orchestrator 服务器与插件技术的交互方式。vso.xml 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。

vso.xml 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或在 Orchestrator **清单**选项卡中显示为查找器对象。

作为插件开放式架构和标准化实现的一部分，vso.xml 文件必须使用标准格式。

以下图标显示了 vso.xml 插件定义文件的格式以及各要素之间相互嵌套的方式。

图 1-4. vso.xml 插件定义文件的格式



命名插件对象

您必须为该插件在插件技术中发现的每个对象提供唯一标识符。您需要分别在 vso.xml 文件中的 <finder> 元素和 <object> 元素中定义对象名称。

您在工厂实现中定义的查找器操作可以查找插件技术中的对象。在插件找到对象后，您可以在 Orchestrator 工作流中使用并在工作流之间传递这些对象。您为这些对象提供的唯一标识符可以让对象在工作流的不同元素之间进行传递。

Orchestrator 服务器仅存储其处理的每个对象的类型和标识符，并不存储 Orchestrator 获取此对象的方式和位置信息。您在插件实现中命名对象时必须保持一致，方便跟踪从插件中获得的对象。

如果 Orchestrator 服务器在工作流运行过程中停止，则在服务器重启时，工作流会从服务器停止时所运行的工作流元素处恢复运行。工作流会使用标识符来检索服务器停止时正在处理的元素。

插件对象命名约定

在命名插件中的所有对象时，必须遵守 Java 类命名约定。

重要事项 鉴于工作流引擎执行数据序列化的方式，在对象名称中不要使用以下字符串序列。在对象标识符中使用以下字符序列会导致工作流引擎错误地解析工作流，这样会造成在运行工作流时发生意外行为。

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

在命名插件中的对象时，使用以下准则。

- 名称中每个词的首字母使用大写字母。
- 不要使用空格来分隔字词。
- 对于字母，仅使用标准字符 `A` 到 `Z` 和 `a` 到 `z`。
- 不要使用特殊字符，例如重音符。
- 不要使用数字作为名称的首字符。
- 尽可能少于 10 个字符。

表 1-5. 插件对象命名规则 显示了应用于各个对象类型的规则。

表 1-5. 插件对象命名规则

对象类型	命名规则
插件	■ 在 `vso.xml` 文件的 `<module>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 必须唯一。您无法在 Orchestrator 服务器中运行两个同名的插件。
查找器对象	■ 在 `vso.xml` 文件的 `<finder>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 在插件中必须唯一。 Orchestrator 会将插件名称和冒号添加到 Orchestrator 脚本 API 的查找器对象类型中的查找器对象名称。例如，vCenter Server 插件中的 `VirtualMachine` 对象类型在 Orchestrator 脚本 API 中显示为 `VC:VirtualMachine`。
脚本对象	■ 在 `vso.xml` 文件的 `<scripting-object>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 在 Orchestrator 服务器中必须唯一。 ■ 为避免混淆脚本对象与同名查找器对象或其他插件中的脚本对象，请始终在脚本对象名称前加上插件名称作为前缀，但不要加冒号。例如，vCenter Server 插件中的 `VirtualMachine` 类在 Orchestrator 脚本 API 中显示为 `VcVirtualMachine` 类。

插件的文件结构

插件必须遵守标准文件结构，并且必须包含某些特定文件夹和文件。作为标准 Java 存档 (JAR) 或 ZIP 文件交付的插件，必须使用 `.dar` 扩展名对其重命名。

DAR 存档的内容必须使用以下文件夹结构和命名约定。

表 1-6. DAR 存档结构

文件夹	说明
plug-in_name\VSO-INF\	包含 vso.xml 文件，用于定义将插件技术中的对象映射到 Orchestrator 对象。 VSO-INF 文件夹和 vso.xml 文件必不可少。
plug-in_name\lib\	包含 JAR 文件，其中包含插件技术的二进制文件。包含的其他 JAR 文件，其中应包含适配器实现、工厂、通知处理程序以及插件中的其他接口。 lib 文件夹和 JAR 文件必不可少。
plug-in_name\resources\	包含插件所需的资源文件。resources 文件夹可以包含以下类型的元素： ■ 图像文件，用来在 Orchestrator **清单**选项卡中表示插件的对象。 ■ 脚本，用来定义插件启动时的初始化行为。 ■ Orchestrator 软件包，可包含自定义工作流、操作和其他资源，能与通过插件访问的对象进行交互。 您可以将资源整理到子文件夹中。例如：resources\images\、resources\scripts\ 或 resources\packages\。 resources 文件夹为可选。

您可以使用 Orchestrator 控制中心将 DAR 文件导入到 Orchestrator 服务器中。

Orchestrator 插件 API 参考

Orchestrator 插件 API 用于定义您在开发 IPluginAdaptor 和 IPluginFactory 实施以创建插件时需要实现和扩展的 Java 接口和类。

所有类均包含在 ch.dunes.vso.sdk.api 软件包中，另有说明的除外。

IAop 接口

IAop 接口提供的方法可获取并设置插件技术中对象上的属性。

```
public interface IAop
```

IAop 接口定义了以下方法：

方法	返回	描述
get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)	java.lang.Object	获取插件中给定对象的属性。
set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)	空	设置插件中给定对象上的属性。

IDynamicFinder 界面

IDynamicFinder 界面会以编程方式返回查找器的 ID 和属性，而无需在 vso.xml 文件中定义这些 ID 和属性。

IDynamicFinder 接口定义了以下方法。

方法	返回	说明
getIdAccessor(java.lang.String type)	java.lang.String	提供 OGNL 表达式，从而以编程方式获取对象 ID。
getProperties(java.lang.String type)	java.util.List<SDKFinderProperty>	以编程方式提供多种对象属性。

IPluginAdaptor 接口

您可实现 IPluginAdaptor 接口以管理插件工厂、事件和观察程序。IPluginAdaptor 接口用于定义插件和 Orchestrator 服务器之间的适配器。

IPluginAdaptor 实例负责会话管理。IPluginAdaptor 接口定义了以下方法。

方法	返回	描述
addWatcher(PluginWatcher watcher)	空	添加观察程序从而监视特定事件。
createPluginFactory(java.lang.String sessionID, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)	IPluginFactory	创建 IPluginFactory 实例。Orchestrator 服务器使用此工厂从插件技术获取对象（按对象 ID 或按对象与其他对象的关系等）。 会话 ID 可让您识别运行的会话。例如，用户可同时登录两个不同的 Orchestrator 客户端并运行两个会话。 同样，启动工作流会创建一个独立会话，独立于该工作流启动时所在的客户端。即使关闭该 Orchestrator 客户端，工作流也会继续运行。
installLicenses(PluginLicense[] licenses)	空	为 VMware 提供的插件安装许可证信息
registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)	空	对清单中的元素设置触发器和计量器
removeWatcher(java.lang.String watcherId)	空	移除观察程序
setPluginName(java.lang.String pluginName)	空	从 vso.xml 文件获取插件名称
setPluginPublisher(IPluginPublisher pluginPublisher)	空	设置插件的发布者
uninstallPluginFactory(IPluginFactory plugin)	空	卸载插件工厂。
unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)	空	从清单中的元素移除触发器和计量器

IPluginEventPublisher 接口

IPluginEventPublisher 接口用于在事件通知总线上对要监视的 Orchestrator 策略发布触发器和计量器。

您可以直接在插件适配器实现中创建 IPluginEventPublisher 实例，或者可以在不同的事件生成器类中创建。

您可以实现 IPluginEventPublisher 接口将插件技术中的事件发布到 Orchestrator 策略引擎。您需要创建方法来设置策略触发器和插件技术中对象的计量器，以及事件侦听器来侦听这些对象的事件。

策略可实现计量器或触发器来监控插件技术中的对象。策略计量器会监控对象的属性并且当对象的值超出特定限制时在 Orchestrator 服务器中推送事件。策略触发器会监控对象并且当对象上发生定义的事件时在 Orchestrator 服务器中推送事件。您可以向 IPluginEventPublisher 实例注册策略计量器和触发器，以便 Orchestrator 策略可对其进行监控。

IPluginEventPublisher 接口定义了以下方法。

类型	返回	说明
pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)	空	针对要监视的策略发布计量器。采用以下参数： ■ type：要监视的对象类型。 ■ id：要监视的对象标识符。 ■ gaugeName：该计量器的名称。 ■ deviceName：该计量器监视的属性类型的名称。 ■ gaugeValue：该计量器监视的对象值。
pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)	空	针对要监视的策略发布触发器。采用以下参数： ■ type：要监视的对象类型。 ■ id：要监视的对象标识符。 ■ triggerName：该触发器的名称。 ■ additionalProperties：需要该触发器监视的任何其他属性。

IPluginFactory 接口

IPluginAdaptor 会返回 IPluginFactory 实例。IPluginFactory 实例可在插件应用程序中运行命令，并查找要执行 Orchestrator 操作的对象。

IPluginFactory 接口定义了以下字段：

static final java.lang.String RELATION_CHILDREN

IPluginFactory 接口定义了以下方法。

方法	返回	说明
executePluginCommand(java.lang.String cmd)	空	使用此插件运行命令。VMware 建议不要使用此方法。
find(java.lang.String type, java.lang.String id)	java.lang.Object	使用此插件查找对象。按对象 ID 和类型识别对象。
findAll(java.lang.String type, java.lang.String query)	QueryResult	使用此插件查找与查询字符串匹配的特定类型的对象。在插件的 IPluginFactory 实现中定义查询语法。如果不定义查询语法，则 findAll() 会返回指定类型的所有对象。
findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	java.util.List	确定对象是否具有子项。
hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	HasChildrenResult	按特定关系查找与给定父项相关的所有子项。
invalidate(java.lang.String type, java.lang.String id)	空	按类型和 ID 使对象无效。
void invalidateAll()	空	使缓存中的所有对象无效。

IPluginNotificationHandler 接口

IPluginNotificationHandler 用于定义针对在 Orchestrator 通过插件访问的对象上所发生的不同类型事件而向 Orchestrator 发出通知的方法。

IPluginNotificationHandler 接口定义了以下方法。

方法	返回	说明
getSessionID()	java.lang.String	返回当前会话 ID。
notifyElementDeleted(java.lang.String type, java.lang.String id)	空	通知系统具有给定类型和 ID 的对象已被删除。
notifyElementInvalidate(java.lang.String type, java.lang.String id)	空	通知系统某个对象的关系已被更改。您可以使用 notifyElementInvalidate() 方法通知 Orchestrator 有关对象之间关系的所有变更情况，而不仅仅是使对象无效的关系变更。例如，向父项添加子对象即表示这两个对象之间关系发生了变更。
notifyElementUpdated(java.lang.String type, java.lang.String id)	空	通知系统某个对象的属性已被更改。
notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)	空	发布与当前模块相关的错误消息

IPluginPublisher 接口

IPluginPublisher 接口用于在事件通知总线上针对“等待事件”元素要监视的长时间运行工作流发布观察程序事件。

当工作流触发器在插件技术中启动事件后，负责观察该触发器（已向 IPluginPublisher 实例注册）的插件观察程序会将事件已发生的消息通知任何正在等待的工作流。

IPluginPublisher 接口定义了以下方法。

类型	值	描述
pushWatcherEvent(java.lang.String id, java.util.Properties properties)	空	在事件通知总线上发布观察程序事件

WebConfigurationAdaptor 接口

WebConfigurationAdaptor 接口实现了 IConfigurationAdaptor，还定义了查找并在插件配置选项卡中安装 Web 应用程序的方法。

注 从 Orchestrator 4.1 开始，WebConfigurationAdaptor 接口已被弃用。若要将 Web 应用程序添加到配置，请实现 IConfigurationAdaptor 并使用 vso.xml 文件中的 configuration-war 属性来识别 Web 应用程序。

WebConfigurationAdaptor 接口定义了以下方法。

方法	返回	描述
getWebAppContext()	字符串	找到配置选项卡的 Web 应用程序的 WAR 文件。以字符串形式提供 DAR 文件中 / webapps 目录内 WAR 文件的名称和路径。
setWebConfiguration(boolean webConfiguration)	布尔	确定配置选项卡的内容是否由 Web 应用程序定义。

PluginTrigger 类

PluginTrigger 类会创建一个触发器模块，代表工作流中的“等待事件”元素，用于获取要在插件技术中监视的对象和事件的相关信息。

PluginTrigger 类定义了获取或设置要监控对象的类型和名称的方法、事件的特性和超时时间段。

您可以创建专门供工作流中“等待事件”元素使用的 PluginTrigger 类实现。您可以按定义了事件和实现了 IPluginEventPublisher.pushTrigger() 方法的类定义 Orchestrator 策略的策略触发器。

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

PluginTrigger 类定义了以下方法：

方法	返回	说明
getModuleName()	java.lang.String	获取触发器模块的名称。
getProperties()	java.util.Properties	获取触发器的属性列表。
getSdkId()	java.lang.String	获取要在插件技术中监视的对象 ID。
getSdkType()	java.lang.String	获取要在插件技术中监视的对象类型。
getTimeout()	长	获取触发器超时时段。
setModuleName(java.lang.String moduleName)	空	设置触发器模块的名称。
setProperties(java.util.Properties properties)	空	设置触发器的属性列表。
setSdkId(java.lang.String sdkId)	空	设置要在插件技术中监视的对象 ID。
setSdkType(java.lang.String sdkType)	空	设置要在插件技术中监视的对象类型。
setTimeout(long timeout)	空	设置超时时段（以秒为单位）。负值将取消激活超时。

构造函数

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

PluginWatcher 类

PluginWatcher 类会代表工作流中的“等待事件”元素来观察插件技术中已定义事件的触发器模块。

PluginWatcher 类定义了您可用来创建插件观察程序实例的构造函数。PluginWatcher 类定义了获取或设置要观察工作流触发器名称的方法，以及超时时间段。

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

PluginWatcher 类定义了以下方法：

方法	返回	说明
getId()	java.lang.String	获取触发器 ID
getModuleName()	java.lang.String	获取触发器模块的名称
getTimeoutDate()	长	获取触发器超时日期
getTrigger()	空	获取触发器
setId(java.lang.String id)	空	设置触发器 ID
setTimeoutDate()	空	设置触发器超时日期

构造函数

`PluginWatcher(PluginTrigger trigger)`

QueryResult 类

`QueryResult` 类包含对 Orchestrator 通过插件所访问对象进行 `find` 查询时的结果。

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

如果已找到结果的总数超过查询返回的结果数，`totalCount` 值可以大于 `QueryResult` 返回的元素数量。查询返回的结果数应在 `vso.xml` 文件中定义。

`QueryResult` 类定义了以下方法：

方法	返回	说明
`addElement(java.lang.Object element)`	空	向 `QueryResult` 添加元素
`addElements(java.util.List elements)`	空	向 `QueryResult` 添加元素列表
`getElements()`	`java.util.List`	从插件应用程序中获取元素
`getTotalCount()`	长	获取插件技术中所有可用元素的数量
`isPartialResult()`	布尔	确定已获取的结果是否完整
`removeElement(java.lang.Object element)`	空	从插件技术中移除元素
`setElements(java.util.List elements)`	空	设置插件技术中的元素
`setTotalCount(long totalCount)`	空	设置插件技术中可用元素的总数

构造函数

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

SDKFinderProperty 类

`SDKFinderProperty` 类定义用于从相关对象中获取并设置其属性的方法，这些对象是由 Orchestrator 查找器对象在插件技术中发现的。`IDynanmicFinder.getProperties` 方法会返回 `SDKFinderProperty` 对象。

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

`SDKFinderProperty` 类定义了以下方法：

方法	返回	说明
`getAttributeName()`	`java.lang.String`	获取对象属性名称
`getBeanProperty()`	`java.lang.String`	获取 Java Bean 的属性

方法	返回	说明
getDescription()	java.lang.String	获取对象说明
getDisplayName()	java.lang.String	获取对象显示名称
getPossibleResultType()	java.lang.String	获取查找器可能返回的结果类型
getPropertyAccessor()	java.lang.String	获取对象属性访问器
getPropertyAccessorTree()	java.lang.Object	获取对象属性访问器树
isHidden()	布尔	显示或隐藏对象
isShowInColumn()	布尔	在数据库列中显示或隐藏对象
isShowInDescription()	布尔	显示或隐藏对象说明
setAttributeName(java.lang.String attributeName)	空	设置对象属性名称
setBeanProperty(java.lang.String beanProperty)	空	设置 Java Bean 中的属性
setDescription(java.lang.String description)	空	设置对象说明
setDisplayName(java.lang.String displayName)	空	设置对象显示名称
setHidden(boolean hidden)	空	显示或隐藏对象
setPossibleResultType(java.lang.String possibleResultType)	空	设置查找器可能返回的结果类型
setPropertyAccessor(java.lang.String propertyAccessor)	空	设置对象属性访问器
setPropertyAccessorTree(java.lang.Object propertyAccessorTree)	空	设置对象属性访问器树
setShowInColumn(boolean showInTable)	空	在数据库列中显示或隐藏对象
setShowInDescription(boolean showInDescription)	空	显示或隐藏对象说明

构造函数

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

PluginExecutionException 类

如果插件在运行操作时遇到异常，PluginExecutionException 类会返回一条错误消息。

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

PluginExecutionException 类从 class java.lang.Throwable 继承以下方法：

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toStringfillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace

构造函数

PluginExecutionException(java.lang.String message)

PluginOperationException 类

PluginOperationException 类用于处理插件在操作过程中遇到的错误。

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

PluginOperationException 类从 class java.lang.Throwable 继承以下方法：

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

构造函数

PluginOperationException(java.lang.String message)

HasChildrenResult 枚举

HasChildrenResult 枚举用于声明给定父项中是否具有子项。IPluginFactory.hasChildrenInRelation 方法会返回 HasChildrenResult 对象。

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

HasChildrenResult 枚举定义以下常量：

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

HasChildrenResult 枚举定义了以下方法：

方法	返回	说明
`getValue()`	int	返回以下其中一个值： **1** 具有子项的父项 **-1** 不具子项的父项 **0** 未知或无效参数
`valueOf(java.lang.String name)`	`static HasChildrenResult`	返回此类型一个指定名称的枚举常量。该字符串必须匹配用于声明此类型枚举常量的标识符。请勿在枚举名称中使用空白字符。
`values()`	`static HasChildrenResult[]`	按声明顺序返回一组包含此枚举类型的常量。此方法可以按以下方式对这些常量进行迭代： `for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);`

`HasChildrenResult` 枚举会继承以下来自 `class java.lang.Enum` 的方法：

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

ScriptingAttribute 注释类型

`ScriptingAttribute` 注释类型可对插件技术中某个对象的属性添加注释，将其作为属性在脚本中使用。

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

`ScriptingAttribute` 注释类型具有以下值：

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

ScriptingFunction 注释类型

`ScriptingFunction` 注释类型可对某个方法添加注释，将其作为属性在脚本中使用。

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

`ScriptingFunction` 注释类型具有以下值：

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

ScriptingParameter 注释类型

ScriptingParameter 注释类型可对某个参数添加注释，将其作为属性在脚本中使用。

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

ScriptingParameter 注释类型具有以下值：

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

vso.xml 插件定义文件的元素

vso.xml 文件包含一组标准元素。有些元素是必选项，有些则是可选项。每个元素包含多种属性，用于定义映射到 Orchestrator 对象和操作的相关对象和操作的值。

此外，元素可拥有零个或多个子元素。子元素用于进一步定义父元素。相同的子元素可以出现在多个父元素中。例如，description 元素没有子元素，但可以显示为多个父元素的子元素，例如 module、example、trigger、gauge、finder、constructor、method、object 和 enumeration。

其后的每个元素定义均会列出其属性、父项和子项。

module 元素

模块是提供给 Orchestrator 使用的一组插件对象。

模块中的信息包括：插件技术中的数据向 Java 类映射的方式、版本控制、模块部署方式以及插件在 Orchestrator 清单中的显示方式。

<module> 元素为可选。<module> 元素具有以下属性：

属性	值	描述
name	字符串	定义插件中所有<finder>元素的类型。此为必需属性。
version	编号	插件版本号，在新版本插件中重新加载安装包时会用到。此为必需属性。
build-number	编号	插件内部版本号，在新版本插件中重新加载安装包时会用到。此为必需属性。
image	图像文件	Orchestrator 清单中显示的图标。此为必需属性。
display-name	字符串	Orchestrator 清单中显示的名称。此为可选属性。
interface-mapping-allowed	true 或 false	Vmware 不建议进行接口映射。此为可选属性。

表 1-7. 元素层次结构

父元素	子元素
无	■ <description> ■ <installation> ■ <configuration> ■ <finder-datasources> ■ <inventory> ■ <finders> ■ <scripting-objects> ■ <enumerations>

description 元素

<description> 元素提供了 API Explorer 文档中所显示插件的元素的说明。

您可在 <description> 和 </description> 标记之间添加在 API Explorer 文档中显示的文本。

<description> 元素为可选。<description> 元素不包含属性。

表 1-8. 元素层次结构

父元素	子元素
■ <module> ■ <example> ■ <trigger> ■ <gauge> ■ <finder> ■ <constructor> ■ <method> ■ <object> ■ <enumeration>	无

deprecated 元素

<deprecated> 元素标记了 API Explorer 文档中已经弃用的对象和方法。

您可在 <deprecated> 和 </deprecated> 标记之间添加在 API Explorer 文档中显示的文本。

<deprecated> 元素为可选。<deprecated> 元素不包含属性。

表 1-9. 元素层次结构

父元素	子元素
■ <method> ■ <object>	无

url 元素

<url> 元素可提供指向与对象或枚举相关的外部文档的 URL。

您需要在 <url> 和 </url> 标记之间提供 URL。

<url> 元素为可选。<url> 元素不包含属性。

表 1-10. 元素层次结构

父元素	子元素
■ <enumeration> ■ <object>	无

installation 元素

<installation> 元素可让您在服务器启动时安装软件包或运行脚本。

<installation> 元素为可选。<installation> 元素具有以下属性：

属性	值	描述
mode	always、never 或 version	在 Orchestrator 服务器启动时设置 mode 值会导致以下行为： ■ 操作 always 运行。 ■ 操作 never 运行。 ■ 操作会在服务器检测到较新版本插件时运行。 此为必需属性。

表 1-11. 元素层次结构

父元素	子元素
<module>	<action>

action 元素

<action> 元素用于指定在 Orchestrator 服务器启动时运行的操作。

<action> 元素属性会提供指向 Orchestrator 软件包或特定脚本的路径，该脚本定义了插件启动时的行为。

<action> 元素为可选。插件可以拥有不限数量的 <action> 元素。<action> 元素具有以下属性。

属性	值	描述
resource	字符串	指向 dar 文件根目录下 Java 软件包或脚本的路径。此为必需属性。
type	install-package 或 execute-script	在 Orchestrator 服务器中安装指定的 Orchestrator 软件包，或运行指定脚本。此为必需属性。

表 1-12. 元素层次结构

父元素	子元素
<installation>	无

finder-datasources 元素

<finder-datasources> 元素是 <finder-datasource> 元素的容器。

<finder-datasources> 元素为可选。<finder-datasources> 元素不包含属性。

表 1-13. 元素层次结构

父元素	子元素
<module>	<finder-datasource>

finder-datasource 元素

<finder-datasource> 元素指向为插件创建的 IPluginAdaptor 实现的 Java 类文件。

您需要设置 Orchestrator 如何访问 <finder-datasource> 元素中插件技术的对象。<finder-datasource> 元素标识了您创建的插件适配器的 Java 类。插件适配器类将您创建的插件工厂实例化。插件工厂定义了插件技术中查找对象的方法。您可以为工厂执行的查找器方法调用的 <finder-datasource> 元素设置超时。不同超时会应用到 IPluginFactory 接口中的不同查找器方法。

<finder-datasource> 元素为可选。插件可以拥有不限数量的 <finder-datasources> 元素。<finder-datasource> 元素具有以下属性。

属性	值	描述
name	字符串	标识了 <finder> 元素 datasource 属性中的数据源。相当于 XML id。此为必需属性。
adaptor-class	Java 类	指向您定义的用来创建插件适配器的 IPluginAdaptor 实现，例如，com.vmware.plugins.sample.Adaptor。此为必需属性。
concurrent-call	true（默认）或 false	允许多个用户同时访问适配器。如果插件不支持并发调用，您必须将 concurrent-call 设置为 false。此为可选属性。
invoker-mode	direct（默认）或 timeout	设置查找器函数的超时。如果设置为 direct，则查找器函数的调用永不会超时。如果设置为 timeout，则 Orchestrator 服务器会应用与查找器方法对应的超时时间段。此为可选属性。
anonymous-login-mode	never（默认）或 always	传递或不传递用户的用户名和密码至插件。此为可选属性。
timeout-fetch-relation	数字；默认为 30 秒	应用于来自 findRelation() 的调用。此为可选属性。
timeout-find-all	数字；默认为 60 秒	应用于来自 findAll() 的调用。此为可选属性。
timeout-find	数字；默认为 60 秒	应用于来自 find() 的调用。此为可选属性。

属性	值	描述
timeout-has-children-in-relation	数字；默认为 2 秒	应用于来自 findChildrenInRelation() 的调用。此为可选属性。
timeout-execute-plugin-command	数字；默认为 30 秒	应用于来自 executePluginCommand() 的调用。此为可选属性。

表 1-14. 元素层次结构

父元素	子元素
<finder-datasources>	无

inventory 元素

<inventory> 元素用于对在 Orchestrator 客户端**清单**视图和对象选择对话框中显示的插件定义其层次结构列表的根。

<inventory> 元素并不表示插件应用程序中的对象，而是表示将插件本身作为 Orchestrator 脚本 API 中的一个对象。

<inventory> 元素为可选。<inventory> 元素具有以下属性。

属性	值	描述
type	Orchestrator 对象类型	<finder> 元素类型，表示对象层次结构的根。此为必需属性。

表 1-15. 元素层次结构

父元素	子元素
<module>	无

finders 元素

<finders> 元素是所有 <finder> 元素的容器。

<finders> 元素为可选。<finders> 元素不包含属性。

表 1-16. 元素层次结构

父元素	子元素
<module>	<finder>

finder 元素

<finder> 元素在 Orchestrator 客户端中代表一种通过该插件发现的对象类型。

<finder> 元素能够识别用于定义由对象查找器表示的对象的 Java 类。<finder> 元素用于定义该对象在 Orchestrator 客户端界面中的显示方式，同时还能够识别由 Orchestrator 脚本 API 所定义的用于表示这一对象的脚本对象。

查找器可以充当不同类型插件技术所用对象格式之间的接口。

<finder> 元素为可选。插件可以拥有不限数量的 <finder> 元素。<finder> 元素定义以下属性：

属性	值	描述
type	Orchestrator 对象类型	查找器表示的对象类型。此为必需属性。
datasource	<finder-datasource name> 属性	识别使用数据源 refid 定义的对象的 Java 类。此为必需属性。
dynamic-finder	Java 方法	用于定义您在 IDynamicFinder 实例中实现的自定义查找器方法，从而以编程方式返回查找器的 ID 和属性，无需在 vso.xml 文件中进行定义。此为可选属性。
hidden	true 或 false（默认值）	如果为 true，则会在 Orchestrator 客户端隐藏该查找器。此为可选属性。
image	图形文件的路径	16x16 图标，表示 Orchestrator 客户端层次结构列表中的查找器。此为可选属性。
java-class	Java 类的名称	用于定义由查找器发现并映射到脚本对象的对象的 Java 类。此为可选属性。
script-object	<scripting-object type> 属性	该查找器要映射到的 <scripting-object> 类（如有）。此为可选属性。

表 1-17. 元素层次结构

父元素	子元素
<finders>	■ <id> ■ <description> ■ <properties> ■ <default-sorting> ■ <inventory-children> ■ <relations> ■ <inventory-tabs> ■ <events>

properties 元素

<properties> 元素是 <finder><property> 元素的容器。

<properties> 元素为可选。<properties> 元素不包含属性。

表 1-18. 元素层次结构

父元素	子元素
<finder>	<property>

properties 元素

<property> 元素会将已发现对象的属性映射到 Java 属性或方法调用。

您可以在实现插件工厂时，调用 SDKFinderProperty 类的方法以获取要处理的插件工厂实现的属性。

您可以在 Orchestrator 客户端的各视图中显示或隐藏对象属性。你也可以使用枚举来定义相关对象属性。

<property> 元素为可选。插件可以拥有不限数量的 <property> 元素。<property> 元素具有以下属性。

属性	值	描述
name	查找器名称	FinderResult 用来存储元素的查找器名称。此为必需属性。
display-name	查找器名称	显示的属性名称此为可选属性。
bean-property	属性名称	您可使用 bean-property 属性来识别要使用 get 和 set 操作获取的属性。如果识别出名为 MyProperty 的属性，插件会定义 getMyProperty 和 setMyProperty 操作。 您可以设置 bean-property 或 property-accessor 的其中之一，但不能同时设置两个。此为可选属性。
property-accessor	获取对象属性值时所用的方法	property-accessor 属性允许您定义 OGNL 表达式来验证对象的属性。 您可以设置 bean-property 或 property-accessor 的其中之一，但不能同时设置两个。此为可选属性。
show-in-column	true（默认）或 false	如果为 true，该属性会显示在 Orchestrator 客户端的示结果表中。此为可选属性。
show-in-description	true（默认）或 false	如果为 true，该属性会显示在对象说明中。此为可选属性。
hidden	true 或 false（默认值）	如果为 true，该属性会在所有情况下隐藏。此为可选属性。
linked-enumeration	枚举名称	将查找器属性与枚举进行链接。此为可选属性。

表 1-19. 元素层次结构

父元素	子元素
<properties>	子元素

relations 元素

<relations> 元素是 <finder><relation> 元素的容器。

<relations> 元素为可选。<relations> 元素不包含属性。

表 1-20. 元素层次结构

父元素	子元素
<finder>	<relation>

relation 元素

<relation> 元素用于定义对象与其他对象之间的相关性。

您可以在 <relation> 元素中定义关系名称。

<relation> 元素为可选。插件可以拥有不限数量的 <relation> 元素。<relation> 元素具有以下属性。

属性	值	描述
name	关系名称	该关系的名称。此为必需属性。
type	Orchestrator 对象类型	通过此关系与其他对象相关的对象的类型。此为必需属性。
cardinality	to-one 或 to-many	定义对象之间的关系是一对一还是一对多。此为可选属性。

表 1-21. 元素层次结构

父元素	子元素
<relations>	无

id 元素

<id> 元素用于定义获取查找器所识别对象唯一 ID 的方法。

<id> 元素为可选。<id> 元素具有以下属性。

属性	值	描述
accessor	方法名称	accessor 属性允许您定义 OGNL 表达式来验证对象的特性。此为必需属性。

表 1-22. 元素层次结构

父元素	子元素
<finder>	无

inventory-children 元素

<inventory-children> 元素用于定义一个层次结构列表，显示在 Orchestrator 客户端**清单**视图和对象选择框中的对象。

<inventory-children> 元素为可选。<inventory-children> 元素不包含属性。

表 1-23. 元素层次结构

父元素	子元素
<finder>	<relation-link>

relation-link 元素

<relation-link> 元素用于定义**清单**选项卡中父对象和子对象之间的层次结构。

<relation-link> 元素为可选。插件可以拥有不限数量的 <relation-link> 元素。<relation-link> 元素具有以下属性。

类型	值	描述
name	关系名称	关系名称的 refid。此为必需属性。

表 1-24. 元素层次结构

父元素	子元素
<inventory-children>	无

events 元素

<events> 元素是 <trigger> 和 <gauge> 元素的容器。

<events> 元素可以包含不限数量的触发器和计量器。

<events> 元素为可选。<events> 元素不包含属性。

表 1-25. 元素层次结构

父元素	子元素
<finder>	■ <trigger> ■ <gauge>

trigger 元素

<trigger> 元素声明了您可用于该查找器的触发器。您必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法来设置触发器。

<trigger> 元素为可选。<trigger> 元素具有以下属性。

类型	值	描述
name	触发器名称	该触发器的名称。此为必需属性。

表 1-26. 元素层次结构

父元素	子元素
<events>	■ <description> ■ <trigger-properties>

trigger-properties 元素

<trigger-properties> 元素是 <trigger-property> 元素的容器。

<trigger-properties> 元素为可选。<trigger-properties> 元素不包含属性。

表 1-27. 元素层次结构

父元素	子元素
<trigger>	<trigger-property>

trigger-property 元素

<trigger-property> 元素用于定义可识别触发器对象的属性。

<trigger-property> 元素为可选。插件可以拥有不限数量的 <trigger-property> 元素。<trigger-property> 元素具有以下属性。

类型	值	描述
name	触发器名称	触发器的名称。此为可选属性。
display-name	触发器名称	Orchestrator 客户端中显示的名称。此为可选属性。
type	触发器类型	用于定义触发器的对象类型。此为必需属性。

表 1-28. 元素层次结构

父元素	子元素
<trigger-properties>	无

gauge 元素

<gauge> 元素定义您可对该查找器使用的计量器。您必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法来设置计量器。

<gauge> 元素为可选。插件可以拥有不限数量的 <gauge> 元素。<gauge> 元素具有以下属性。

类型	值	描述
name	计量器名称	计量器的名称。此为必需属性。
min-value	数字	最低阈值。此为可选属性。
max-value	编号	最高阈值。此为可选属性。
unit	对象类型	用于定义计量器的对象类型。此为必需属性。
format	字符串	受监视的值的格式。此为可选属性。

表 1-29. 元素层次结构

父元素	子元素
<events>	<description>

scripting-objects 元素

<scripting-objects> 元素是 <object> 元素的容器。

<scripting-objects> 元素为可选。<scripting-objects> 元素不包含属性。

表 1-30. 元素层次结构

父元素	子元素
<module>	<object>

object 元素

<object> 元素用于将插件技术的构造函数、属性和方法映射到 Orchestrator 脚本 API 所公开的 JavaScript 对象类型。

有关对象命名的约定信息，请参见 命名插件对象。

<object> 元素为可选。插件可以拥有不限数量的 <object> 元素。<object> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 名称	类的脚本名称。必须全局唯一。此为必需属性。
java-class	Java 类	由该 JavaScript 类封装的 Java 类。此为必需属性。
create	true（默认）或 false	如果为 true，您可以为该类创建一个新实例。此为可选属性。
strict	true 或 false（默认值）	如果为 true，您只能调用在 vso.xml 文件中注释或声明的方法。此为可选属性。
is-deprecated	true 或 false（默认值）	如果为 true，此对象映射的方法已弃用。此为可选属性。
since-version	字符串	Java 类被弃用时的版本。此为可选属性。

表 1-31. 元素层次结构

父元素	子元素
<scripting-objects>	■ <description> ■ <deprecated> ■ <url> ■ <constructors> ■ <attributes> ■ <methods> ■ <singleton>

constructors 元素

<constructors> 元素是 <object><constructor> 元素的容器。

<constructors> 元素为可选。<constructors> 元素不包含属性。

表 1-32. 元素层次结构

父元素	子元素
<object>	<constructor>

constructor 元素

<constructor> 元素用于定义构造函数方法。<constructor> 方法会在 API Explorer 中生成文档。

<constructor> 元素为可选。插件可以拥有不限数量的 <constructor> 元素。<constructor> 元素不包含属性。

表 1-33. 元素层次结构

父元素	子元素
<constructors>	■ <description> ■ <parameters>

Constructor parameters 元素

<parameters> 元素是 <constructor><parameter> 元素的容器。

<parameters> 元素为可选。<parameters> 元素不包含属性。

表 1-34. 元素层次结构

父元素	子元素
<constructor>	<parameter>

Constructor parameter 元素

<parameter> 元素用于定义构造函数的参数。

<parameter> 元素为可选。插件可以拥有不限数量的 <parameter> 元素。<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	要在 API 文档中使用的参数名称。此为必需属性。
type	Orchestrator 参数类型	要在 API 文档中使用的参数类型。此为必需属性。
is-optional	true 或 false	如果为 true，值可以为空。此为可选属性。
since-version	字符串	方法版本。此为可选属性。

表 1-35. 元素层次结构

父元素	子元素
<parameters>	无

attributes 元素

<attributes> 元素是 <object><attribute> 元素的容器。

<attributes> 元素为可选。<attributes> 元素不包含属性。

表 1-36. 元素层次结构

父元素	子元素
<object>	<attribute>

attribute 元素

<attribute> 元素会将 Java 类的属性从插件技术映射到 Orchestrator JavaScript 引擎提供的 JavaScript 属性。

<attribute> 元素为可选。插件可以拥有不限数量的 <attribute> 元素。<attribute> 元素具有以下属性。

类型	值	描述
java-name	Java 属性	Java 属性的名称此为必需属性。
script-name	JavaScript 对象	对应 JavaScript 对象的名称。此为必需属性。
return-type	字符串	此属性返回的对象类型。会显示在 API Explorer 文档中。此为可选属性。 注 如果 JavaScript 返回类型为 Properties，其支持的基础 Java 实现则为 java.util.HashMap 和 java.util.Hashtable。
read-only	true 或 false	如果为 true，您无法修改此属性。此为可选属性。
is-optional	true 或 false	如果为 true，此字段可以为空。此为可选属性。
show-in-api	true 或 false	如果为 false，此属性不会显示在 API 文档中。此为可选属性。
is-deprecated	true 或 false	如果为 true，此对象映射的属性已弃用。此为可选属性。
since-version	数字	属性被弃用时的版本。此为可选属性。

表 1-37. 元素层次结构

父元素	子元素
<attributes>	无

methods 元素

<methods> 元素是 <object><method> 元素的容器。

<methods> 元素为可选。<methods> 元素不包含属性。

表 1-38. 元素层次结构

父元素	子元素
<object>	<method>

method 元素

<method> 元素可以将 Java 方法从插件技术映射到 Orchestrator JavaScript 引擎公开的 JavaScript 方法。

<method> 元素为可选。插件可以拥有不限数量的 <method> 元素。<method> 元素具有以下属性。

类型	值	描述
java-name	Java 方法	Java 方法签名的名称，其参数类型放在括弧中，例如，getVms(DataStore)。此为必需属性。
script-name	JavaScript 方法	JavaScrip 方法相应的名称。此为必需属性。
return-type	Java 对象类型	此方法所获取的类型。此为可选属性。 注 如果 JavaScript 返回类型为 Properties，其支持的基础 Java 实现则为 java.util.HashMap 和 java.util.Hashtable。
static	true 或 false	如果为 true，此方法为静态。此为可选属性。
show-in-api	true 或 false	如果为 false，此属性不会显示在 API 文档中。此为可选属性。
is-deprecated	true 或 false	如果为 true，此对象映射的方法已弃用。此为可选属性。
since-version	编号	方法被弃用时的版本。此为可选属性。

表 1-39. 元素层次结构

父元素	子元素
<methods>	■ <deprecated> ■ <description> ■ <example> ■ <parameters>

example 元素

<example> 元素可让您将代码示例添加到 API Explorer 文档中显示的 JavaScript 方法。

<example> 元素为可选。<example> 元素不包含属性。

表 1-40. 元素层次结构

父元素	子元素
<method>	■ <code> ■ <description>

code 元素

<code> 元素提供了显示在 API Explorer 文档中的示例代码。

您需要在 <code> 和 </code> 标记之间提供代码示例。<code> 元素为可选。<code> 元素不包含属性。

表 1-41. 元素层次结构

父元素	子元素
<example>	无

Method parameter 元素

<parameters> 元素是 <method><parameter> 元素的容器。

<parameters> 元素为可选。<parameters> 元素不包含属性。

表 1-42.

父元素	子元素
<method>	<parameter>

Method parameter 元素

<parameter> 元素用于定义该方法的输入参数。

<parameter> 元素为可选。插件可以拥有不限数量的 <parameter> 元素。<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	参数名称。此为必需属性。
type	Orchestrator 参数类型	参数类型。此为必需属性。
is-optional	true 或 false	如果为 true，值可以为空。此为可选属性。
since-version	字符串	方法版本。此为可选属性。

表 1-43. 元素层次结构

父元素	子元素
<parameters>	无

singleton 元素

<singleton> 元素会将 JavaScript 脚本对象创建为单一实例。

单一对象与静态 Java 类的行为相同。单一对象定义了插件要使用的通用对象，而不是定义 Orchestrator 在插件技术中访问的对象的特定实例。例如，您可以使用单一对象与插件技术建立连接。

<singleton> 元素为可选。<singleton> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 对象	对应 JavaScript 对象的名称。此为必需属性。
datasource	Java 对象	此 JavaScript 对象的源 Java 对象。此为必需属性。

表 1-44. 元素层次结构

父元素	子元素
<object>	无

enumerations 元素

<enumerations> 元素是 <enumeration> 元素的容器。

<enumerations> 元素为可选。<enumerations> 元素不包含属性。

表 1-45. 元素层次结构

父元素	子元素
<module>	<enumeration>

enumeration 元素

<enumeration> 元素用于定义适用于特定类型所有对象的常用值。

如果特定类型的所有对象需要某个特定属性，且该属性值的范围有限，您可以将这些不同值定义为枚举条目。例如，如果某个对象类型需要 color 属性，且可用颜色只有红、蓝、绿，您就可以定义将这三种颜色值定义为三个枚举条目。这些条目将被定义为枚举元素的子元素。

<enumeration> 元素为可选。插件可以拥有不限数量的 <enumeration> 元素。<enumeration> 元素具有以下属性。

类型	值	描述
type	Orchestrator 对象类型	枚举类型。此为必需属性。

表 1-46. 元素层次结构

父元素	子元素
<enumerations>	■ <url> ■ <description> ■ <entries>

entries 元素

<entries> 元素是 <enumeration><entry> 元素的容器。

<entries> 元素为可选。<entries> 元素不包含属性。

表 1-47. 元素层次结构

父元素	子元素
<enumeration>	<entry>

entry 元素

<entry>元素用于为枚举属性提供值。

<entry> 元素为可选。插件可以拥有不限数量的 <entry> 元素。<entry> 元素具有以下属性。

类型	值	描述
id	Text	对象用于将枚举条目设置为属性时的标识符。此为必需属性。
name	Text	entry 名称此为必需属性。

表 1-48. 元素层次结构

父元素	子元素
<entries>	无

Orchestrator 插件开发的最佳做法

了解 Orchestrator 插件的结构和内容以及如何避免具体问题，可帮助您对您所开发的 Orchestrator 插件的相关方面进行改进。

- 构建 Orchestrator 插件的方法

  您可以使用不同方法来构建 Orchestrator 插件。您可以逐层开始构建插件，也可以同时开始构建插件的所有层。

- Orchestrator 插件类型

  通过使用插件，您可以借助 Orchestrator 集成常规用途库或实用程序（例如 XML 或 SSH）以及整个系统（例如 vCloud Director）。根据与 Orchestrator 集成的具体技术，插件可归类为服务插件、常规用途插件及系统插件。

- 插件实现

  在构建插件、实现所需 Java 类和 JavaScript 对象、开发插件工作流和操作以及提供工作流展示时，您可以使用一些实用做法和技巧。

- Orchestrator 插件开发建议

  在开发 Orchestrator 插件的不同组件时，遵循部分特定做法可帮助您改进插件的质量。

- 编写插件用户界面字符串和 API 的文档

  在编写 Orchestrator 插件的用户界面 (UI) 字符串和相关 API 的文档时，请遵循以下广受认可的样式和格式规则。

构建 Orchestrator 插件的方法

您可以使用不同方法来构建 Orchestrator 插件。您可以逐层开始构建插件，也可以同时开始构建插件的所有层。

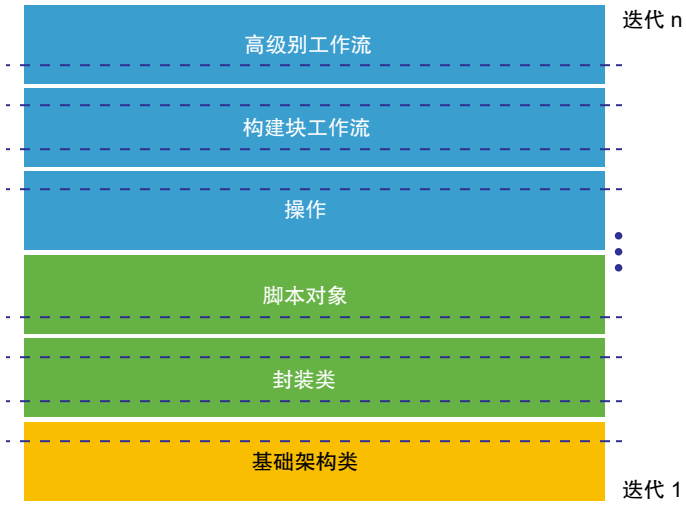
有关插件层的信息，请参阅 Orchestrator 插件的结构。

自下而上的插件开发

您可以使用自下而上的开发方法逐层构建插件。

自下而上的开发方法从低级别的层开始，随后逐层递增到更高级别的层，最终完成插件构建。如果该方法混合了交互式和迭代式开发方法，则每次迭代都会交付部分或整个层。在 N 次迭代结束后，即完全完成了插件。

图 1-5. 自下而上的插件开发



自下而上的插件开发方法的优势在于开发时每次都只关注于一层。

请注意自下而上插件开发方法的以下不利因素。

- 很难显示插件开发的进度，除非完成部分插入。
- 不太适合敏捷开发做法。

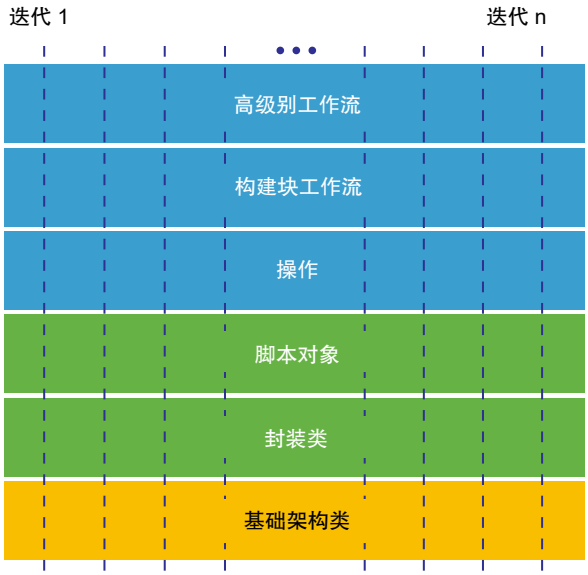
自下而上开发过程非常适合小型插件的开发，减少甚至无需多组封装类、脚本对象、操作或工作流。

自上而下的插件开发

使用自上而下的开发方法，可以将插件自上而下拆分为多个功能，从而进行构建。

如果自上而下的方法与敏捷开发过程混合使用，则在每次迭代时都会交付新功能。最后，在 N 次迭代结束后，即完全实现了插件。

图 1-6. 自上而下的插件开发



自上而下的插件开发方法具有以下优势。

- 插件开发的进度从首次迭代开始就易于查看，因为每次迭代后都会完成新功能并且每次迭代后都能发行并使用插件。
- 完成功能垂直拆分可非常清晰地定义成功条件和完成条件，以及在开发人员、产品管理人员和质量保证 (QA) 工程师之间加强交流。
- 可让 QA 工程师从开发流程的一开始就进行测试和自动化执行。此类方法可获得宝贵的反馈并缩短整个项目交付的周期。

自上而下的插件开发方法其不足之处在于不同层都在同时进行开发。

对于大部分插件，都可以应用自上而下的插件开发方法。此方法同样还适用于具有动态要求的插件。

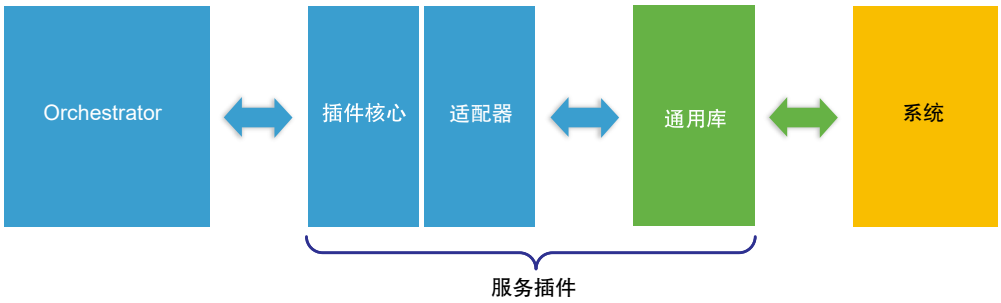
Orchestrator 插件类型

通过使用插件，您可以借助 Orchestrator 集成常规用途库或实用程序（例如 XML 或 SSH）以及整个系统（例如 vCloud Director）。根据与 Orchestrator 集成的具体技术，插件可归类为服务插件、常规用途插件及系统插件。

服务插件

服务插件或常规用途插件提供的功能可视为 Orchestrator 中的服务。

图 1-7. 服务插件的架构



服务插件将通用库或实用程序（例如 XML、SSH 或 SOAP）公开到 Orchestrator。例如，以下 Orchestrator 中可用的插件即为服务插件。

JDBC 插件	可让您在工作流中使用任意数据库。
Mail 插件	可让您在工作流中发送电子邮件。
SSH 插件	可让您在工作流中打开 SSH 连接并运行命令。
XML 插件	可让您在工作流中管理 XML 文档。

服务插件具有以下特征。

复杂度	服务插件的复杂度从低到中不等。服务插件在 Orchestrator 中公开特定库或库的一部分，以便提供具体功能。例如，XML 插件将文档对象模型 (DOM) XML 解析程序的实现添加到 Orchestrator JavaScript API。
大小	服务插件相对较小。除了所有插件都使用的相同基本类集，它们还需要能够提供新脚本对象的其他类，便于增加新功能。
清单	服务插件需要一份简短的对象清单就能运行，或者根本无需清单。服务插件具有通用的小型对象模型，因此，无需在 Orchestrator 清单中显示该模型。

系统插件

系统插件会将 Orchestrator 工作流引擎连接到外部系统，以便您可以编排外部系统。

以下是系统插件的示例。

vCenter Server 插件	可让您使用工作流来管理 vCenter Server 实例。
vCloud Director 插件	可让您与工作流内的 vCloud Director 安装进行交互。
Cisco UCSM 插件	可让您与工作流内的 Cisco 实体进行交互。

以下是系统插件的主要特征。

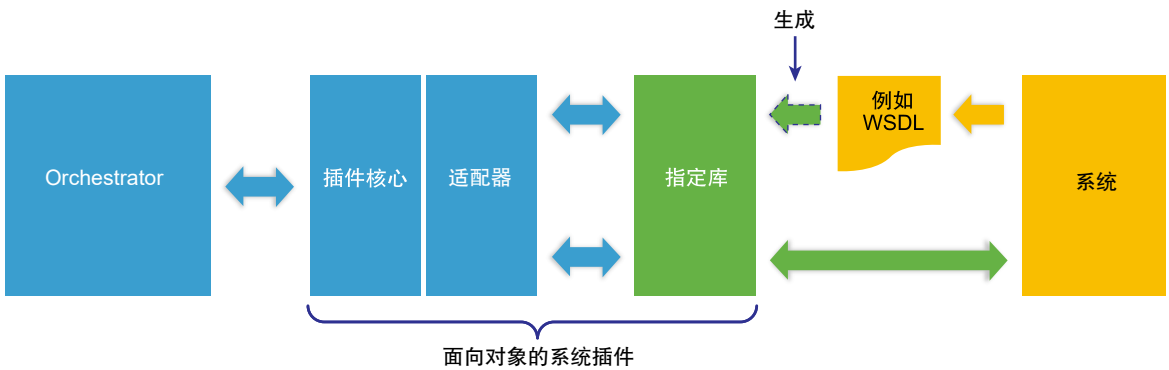
复杂度	相比常规用途的插件，系统插件的复杂度更高，因为其中的技术相对较为复杂。系统插件必须代表 Orchestrator 中外部系统的所有元素与外部系统进行交互，并在 Orchestrator 中提供其功能。如果外部系统提供集成机制，您可以用其在 Orchestrator 中更加轻松地公开系统的功能。但是，除了代表 Orchestrator 中外部系统的元素外，系统插件还可能需要提供高扩展性、提供缓存机制，以及处理事件和通知等。
大小	系统插件通常为中等大小或较大。除了基本类集，系统插件还需要许多其他类，因为其通常会提供大量脚本对象。系统插件可能需要部分可与之交互的其他帮助程序和辅助类。
清单	系统插件通常包含大量对象，您必须在清单中正确公开这些对象以便在 Orchestrator 中轻松找到并进行处理。由于需要公开系统插件的大量对象，您应该构建辅助工具或流程以尽可能多地为插件自动生成代码。例如，vCenter Server 插件会提供此类工具。

面向对象的系统插件

面向对象的系统提供了基于对象和 RPC 的交互机制。

面向对象的系统最常使用的模型为使用 SOAP 的 Web 服务模型。此模型中的对象具有一组与对象状态相关的属性，并提供一组可在目标系统调用的远程方法。

图 1-8. 面向对象的系统插件



在为面向对象的系统实施插件时，可以考虑以下内容。

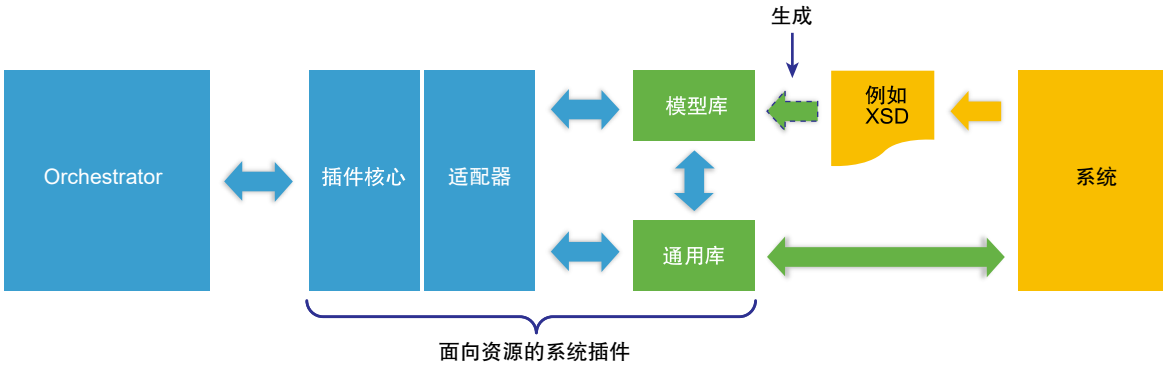
- 如果使用 SOAP，则可以使用 WSDL 文件来生成一组合并了对象模型和通信机制的类。
- 此对象模型几乎涵盖了要在 Orchestrator 中公开的所有内容。

面向资源的系统插件

面向资源的系统提供了基于使用 HTTP 方法的资源和简单操作的交互机制。

面向资源的系统最具代表性的模型是 REST 模型，例如合并了 XML。此模型中的对象具有一组与对象状态相关的属性。若要在目标系统上调用方法（通信机制），您必须使用标准 HTTP 方法（例如 GET、POST、PUT 等）并遵守相关约定。

图 1-9. 面向资源的系统插件



在为面向资源的系统开发插件时，可以考虑以下内容。

- 如果使用 REST 或仅 HTTP 与 XML，您会获得一个或多个可以读写消息的 XML 架构文件。通过这些架构，您可以生成定义对象模型的一组类。这组类仅定义了对象状态，因为操作由 HTTP 方法隐性定义（例如在 vCloud Director 插件中定义）或由特定 XML 消息显式定义（例如 Cisco UCSM 插件）。
- 您需要在另一组类中实施通信机制。这组类定义了与原始对象模型进行交互的新对象模型。通信机制的对象模型仅由对象和方法组成。
- 您可以在 Orchestrator 中同时公开原始对象模型和通信机制的对象模型。这样可能会增加复杂度，具体取决于如何公开两种对象模型，以及您是合并了这两种模型中的相关对象（以模拟面向对象的系统）还是将其保持分离。

插件实现

在构建插件、实现所需 Java 类和 JavaScript 对象、开发插件工作流和操作以及提供工作流展示时，您可以使用一些实用做法和技巧。

- 项目结构

  您可以为 Orchestrator 插件的项目应用标准结构。

- 项目内部机制

  您可以在实现插件时应用特定方法，例如缓存对象、将对象置于后台、克隆对象等。按照此类方法操作，您可以改进插件的性能、避免并发问题并改进 Orchestrator 客户端的响应能力。

- 工作流内部机制

  您可以执行工作流来监控 Orchestrator 插件执行的长时间操作。

- 工作流和操作

  若要使工作流开发和使用更加简单，您可以参考一些比较好的做法。

- 工作流展示

  创建工作流展示时，需应用特定结构和规则。

项目结构

您可以为 Orchestrator 插件的项目应用标准结构。

您可以使用标准 Maven 结构以及插件项目模块以清楚了解每个功能的具体位置。

表 1-49. 插件项目结构

模块	描述
`/myAwesomePlugin-plugin`	插件项目的根。
`/o11nplugin-myAwesomePlugin`	构成最终插件 DAR 文件的模块。
`/o11nplugin-myAwesomePlugin-config`	包含插件配置 Web 应用程序的模块。它会生成标准 WAR 文件。
`/o11nplugin-myAwesomePlugin-core`	此模块包含实现任一标准 Orchestrator 插件接口的所有类以及所用的其他辅助类。它会生成标准 JAR 文件。
`/o11nplugin-myAwesomePlugin-model`	此模块包含可帮助您通过插件将第三方技术与 Orchestrator 集成的所有类。相关类不应包含对标准 Orchestrator 插件 API 的任何直接引用。
`/o11nplugin-myAwesomePlugin-package`	此模块会导入包含操作和工作流的外部 Orchestrator 软件包文件，从而将其包含在最终插件 DAR 文件内。此模块可选

项目内部机制

您可以在实现插件时应用特定方法，例如缓存对象、将对象置于后台、克隆对象等。按照此类方法操作，您可以改进插件的性能、避免并发问题并改进 Orchestrator 客户端的响应能力。

缓存对象

您的插件可与远程服务交互，这一交互操作由代表服务端远程对象的本地对象提供支持。为实现插件的最佳性能以及 Orchestrator 用户界面的卓越响应能力，您可以缓存本地对象而不是每次都从远程服务上获取。您可以考虑缓存的范围，例如所有插件客户端使用一个缓存、插件按用户各使用一个缓存以及第三方服务按用户各使用一个缓存。实施后，缓存机制会与插件接口集成以查找并使对象失效。

将对象置于后台

如果必须在插件清单中显示较长的对象列表，并且没有较快途径来检索这些对象，您可以将对象置于后台。例如，通过使对象具有 `fake` 和 `loaded` 状态，您可以将其置于后台。假设 `fake` 对象易于创建并且在清单中要显示的信息最少，例如名称和 ID。那么，就可以始终返回 `fake` 对象，并且当确实需要所有信息（真实对象）时，正在使用的实体或插件可以自动调用方法 `load` 来获取真实对象。您甚至可以配置在返回假对象后自动启动加载对象流程，从而预测使用中实体的操作。

克隆对象以避免并发问题

如果为插件使用缓存，则必须克隆对象。使用的缓存如果始终将对象的同一实例返回至每个发起请求的实体，则可能会发生意料之外的结果。例如，实体 A 请求了对象 O，并且实体查看了清单中的对象及其所有属性。与此同时，实体 B 也请求了对象 O，并且实体 A 运行了开始更改对象 O 属性的工作流。在运行结束时，工作流调用了对象的 `update` 方法来更新服务器端的对象。如果实体 A 和实体 B 获得对象 O 的同一实例，则实体 A 会在清单中看到实体 B 执行的所有更改，甚至在更改内容提交到服务器之前也能看到。如果运行正常，这应该不是问题。但如果运行失败，则实体 A 请求的对象 O 属性无法恢复。在这种情况下，如果缓存（插件的 `find` 操作）返回对象的克隆内容而不是始终返回同一实例，即每个都使用实体查看并修改各自的副本，则至少可在 Orchestrator 内避免并发问题。

向他人通知更改

同时使用缓存和克隆对象时可能会发生问题。最大的问题是使用实体视图的对象可能不是可用于对象的最新版本。例如，如果某个实体显示了清单、已经加载一次对象，但同时如果另一实体更改了部分对象，则首个实体不会看到更改内容。为避免此问题，您可以使用 Orchestrator 插件 API 中的 `PluginWatcher` 和 `IPluginPublisher` 方法通知发生的更改，从而使 Orchestrator 客户端的其他实例能看到更改内容。如果清单中某个对象的更改影响了清单的其他对象，并且受影响对象也需要获得通知，那么上述情况也适用于 Orchestrator 客户端的唯一实例。倾向于使用通知的操作包含添加、更新和删除对象（如果这些对象或其部分属性显示在清单内）。

启用随时查找任意对象

您必须实现 `IPluginFactory` 接口的 `find` 方法来仅按类型和 ID 查找对象。重新启动 Orchestrator 并恢复工作流后，可以直接调用 `find` 方法。

模拟查询服务（如无）

Orchestrator 客户端可以要求在特定情况下查询部分对象，或要求将对象显示为列表或表格而非树形图。这意味着您的插件必须可以随时查询部分对象集。如果第三方技术提供了查询服务，则您需要适应并使用该服务。否则，无论解决方案的复杂度有多高，性能有多低，您都应可模拟查询服务。

查找方法不应返回运行时异常

`IPluginFactory` 接口中用于在插件中实施搜索的方法不应出现受控或不受控的运行时异常。这可能是当工作流在运行时出现异常验证错误失败的原因。例如，在工作流的两个节点之间，如果首个节点的输出是第二个节点的输入，则会调用 `find` 方法。此时，如果由于任何运行时异常而无法找到对象，您可能在 Orchestrator 客户端中不会收到除验证错误之外的更多信息。在此之后，根据插件记录异常的方式，可以在日志文件内获取或多或少的信息。

工作流内部机制

您可以执行工作流来监控 Orchestrator 插件执行的长时间操作。

您可以部署用于监控长时间运行操作（例如任务监控）的工作流。该工作流可基于 Orchestrator 触发器和等待事件。您必须考虑到正等待任务的被阻挡工作流可以在 Orchestrator 服务器启动后立即恢复。插件必须能够获取所有必需信息以正确恢复监控流程。

监控工作流或其可在内部使用的任务应提供相关机制，从而指定池化率和可能的超时。

对工作流内的一段脚本代码进行调试并非易事，特别是当代码未调用任何 Java 代码时。因此，有时只能选择使用默认 Orchestrator 脚本对象提供的日志记录方法。

工作流和操作

若要使工作流开发和使用更加简单，您可以参考一些比较好的做法。

开始以构建块形式开发工作流

构建块可以是一个简单的工作流，需要一些输入参数并会返回简单输出。如果您有一组丰富的构建块，可以轻松创建级别更高的工作流，并且可以使用一组更好的工具来构建复杂工作流。

基于较小组件创建级别更高的工作流

如果您需要开发具有多个输入和内部步骤的复杂工作流，您可以将其拆分为较小且较简单的构建块工作流和操作。

可随时创建操作

您可以在开发工作流时创建操作来获得额外的灵活性。

- 轻松为脚本方法创建复杂对象或参数
- 避免始终重复使用常用的代码片段
- 执行用户界面验证

工作流应可随时调用操作

操作可以在工作流架构中直接调用为节点。这样能使工作流架构更简单，因为您无需添加脚本代码块以调用单个操作。

填写所需信息

提供工作流或操作的每个元素的相关信息。

- 提供工作流或操作的说明。
- 提供输入参数的说明。
- 提供输出的说明。
- 提供工作流属性的说明。

保持版本信息始终为最新

为插件添加版本信息时，添加实用注释，例如，插件的主要更新、重要的实现详情等。

工作流展示

创建工作流展示时，需应用特定结构和规则。

在工作流展示中对工作流输入使用以下属性。

表 1-50. 工作流输入属性

属性	使用情况
`Show in Inventory`	使用该属性帮助用户通过清单视图运行工作流。
`Specify a root object to be shown in the chooser`	使用该属性帮助用户选择输入。如果根对象可在展示中刷新、是一个属性或由某个对象方法检索，则您需要创建或设置相应的操作来刷新展示中的对象。
`Maximum string length`	对长字符串（例如名称、描述、文件路径等）使用该属性。

表 1-50. 工作流输入属性 （续）

属性	使用情况
`Minimum string length`	使用该属性避免测试工具中包含空字符串。
`Custom validation`	实施通过操作完成的复杂验证。

使用步骤和显示组整理输入。此类整理可帮助用户识别和区分工作流的所有输入参数。

Orchestrator 插件开发建议

在开发 Orchestrator 插件的不同组件时，遵循部分特定做法可帮助您改进插件的质量。

表 1-51. 插件实现实用做法

组件	条目	描述
常规	访问第三方 API	插件应尽可能提供简化的第三方 API 访问方法。
	接口	插件应向用户提供一致且标准的接口，即使 API 未能做到也是如此。
操作	脚本对象	您应该为每个创建、修改、删除及所有其他对脚本对象可用的方法创建操作。
	描述	操作的描述应说明操作有何用处而不是如何运作。
	脚本	使用脚本来获取对象的属性或方法时，可以检查对象值是否与 `null` 或 `undefined` 不同。
	弃用	如果某个操作已被弃用，`comment` 或 `throw` 语句应指示替代操作，或操作应调用新的替代操作，从而使得在操作的已弃用版本上构建的解决方案不会失效。
工作流	编排技术中的用户界面操作	您应当为编排技术用户界面中可用的每个操作创建工作流。
	描述	工作流的描述应说明工作流有何用处而不是如何运作。
	展示属性 `mandatory input`	您必须为所有强制工作流输入设置 `mandatory input` 属性。
	展示属性 `default value`	如果开发了一个配置实体的工作流，则工作流展示应加载该实体的默认配置值。例如，如果开发名为“主机配置”的工作流，则工作流的展示必须加载主机配置的默认值。
	展示属性 `Show in inventory`	您必须设置 `Show in inventory` 属性以便在清单对象上拥有上下文工作流。
	展示属性 `specify a root parameter`	如果无需从树根浏览清单，则应在工作流中使用该属性。
	工作流验证	您必须验证工作流并修复所有错误。
	对象创建	所有创建了新对象的工作流都应将新对象返回为输出参数。
	弃用	如果某个工作流已被弃用，则 `comment` 或 `throw` 语句应指示替代工作流，或弃用的工作流应调用新的替代工作流以确保在工作流先前版本上构建的解决方案不会失效。
清单	主机断开连接	如果清单包含主机连接并且该主机不再可用，则应表明主机已断开连接。您可以通过重命名根对象（附加 `– disconnected`）或移除该对象下的对象树来执行此操作，具体方法与 vCloud Director 插件的方法相同。

表 1-51. 插件实现实用做法 （续）

组件	条目	描述
	`Select value as list` 属性	清单对象必须可选择为 `treeview` 或 `list`。
	主机管理器	如果插件为目标系统实现了 `host` 对象，则应存在一个父 `hostmanager` 根对象，具有用于添加、移除或编辑主机属性的属性。
	获取或更新对象	如果查询服务在编排技术上运行，您应使用该服务来获取多个对象。
	子对象发现	如果您需要单独检索子对象，则检索过程必须采取多线程并且不能受单个错误阻止。
	Orchestrator 对象更改	所有可以更改清单中元素状态的工作流必须更新清单以避免对象未能同步。
	外部对象更改	您可以使用通知机制来通知编排技术中由于在 Orchestrator 外执行的操作所产生的更改。如果此类操作会将对象从编排技术中移除，您必须相应刷新清单以避免发生故障或丢失数据。例如，如果从 vCenter Server 中删除虚拟机，则 vCenter Server 插件将更新清单以移除已移除虚拟机的对象。
	查找器对象	查找器对象应具有可用于区分对象的属性。这些通常都是位于用户界面中的属性。
脚本对象	实现	必须实现 `equals` 方法以确保 `==` 操作在同一对象上运行，这是因为在某些情况下对象可能拥有两个实例。
	插件对象属性	具有父对象的对象应实现 `parent` 属性。
	插件对象属性	具有子对象的对象应实现 `GET` 方法，以返回子对象数组。
	清单对象	清单对象应可使用 `Server.find` 搜索。
		所有清单对象应可序列化以便用作工作流中的输入或输出属性。
	构造函数和方法	在大多数情况下，可编辑脚本的对象应具有构造函数，或应通过其他对象属性或方法返回。
	对象 ID	如果对象具有从外部系统颁发的 ID，则在编排多台服务器时应使用内部 ID 以确保不会发生 ID 重复。
	搜索对象	`search` 或 `find` 方法应实现筛选器，以便可以查找指定的名称或 ID，而不是仅查找所有对象。例如，Orchestrator 服务器具有 `Server.FindForId` 方法，可按 ID 查找插件对象。为此，必须为插件中的每个可查找对象实现该方法。
	触发器	如有可能，触发器应可用于发生更改的对象，这样 Orchestrator 可以对各类事件触发策略。例如，为确定新虚拟机何时添加、打开电源、关闭电源等，Orchestrator 可以监控 `Datacenter` 对象上 vCenter 插件中的触发器或事件。
	对象属性	驻留在其他插件中的对象所具有的属性应可从一个插件对象轻松转换到另一个。例如，虚拟机对象需要拥有 `moref`（受管对象引用 ID）。
	会话管理器	如果您要连接到可能存在不同会话的远程服务器，插件应实现两种会话：共享会话和单用户会话。
触发器	触发器	所有长操作和阻止方法都应能异步启动并返回任务，并在完成时生成触发器事件。
枚举	枚举	给定类型的枚举应拥有可在枚举中选择不同值的清单对象。

表 1-51. 插件实现实用做法 （续）

组件	条目	描述
日志记录	日志	方法应实现不同日志级别。
版本控制	插件版本	插件版本应遵循相关标准并随插件更新一同更新。
API 文档	方法	API 文档中描述的方法不得在对象上引发异常 `no xyz method / property`。相反，方法应在无可用属性时返回 `null`，并在这些属性不可用时详细记录。
	`vso.xml`	所有对象、方法和属性都必须记录在 `vso.xml` 内。

编写插件用户界面字符串和 API 的文档

在编写 Orchestrator 插件的用户界面 (UI) 字符串和相关 API 的文档时，请遵循以下广受认可的样式和格式规则。

常规建议

- 若插件中涉及 VMware 产品，请使用其官方名称。例如，使用以下产品的官方名称和 VMware 术语。

正确术语	请勿使用
vCenter Server	VC 或 vCenter
vCloud Director	vCloud

- 每句工作流描述都以句点结尾。例如，`Creates a new Organization.` 是一句工作流描述。
- 使用文本编辑器和拼写检查器来编写描述，然后将其移至插件。
- 确保插件名称完全匹配所关联第三方产品已批准的名称。

工作流和操作

- 编写信息性描述。大多数操作和工作流只需一两个句子即可。
- 较高级别工作流可能要包含更宽泛的描述和备注。
- 编写描述时直接以动词开头，例如：`Creates…`。不要使用自引用语言，例如 `This workflow creates`。
- 在句子完整的描述结尾使用句点。
- 描述工作流或操作有何用处而不是如何实现。
- 工作流和操作通常随附在文件夹和软件包中。同时也随附了这些文件夹和软件包的简短描述。例如，工作流文件夹的描述可类似 `Set of workflows related to vApp Template management`。

工作流和操作的参数

- 使用描述性的名词短语（例如 `Name of`）作为工作流和操作描述的开头。请勿使用诸如 `It's the name of` 等短语。
- 请勿在参数和操作描述的结尾添加句点。这些不是完整的句子。

- 工作流的输入参数必须指定标签，在展示视图中包含相应名称。在许多情况下，您可以合并显示组中的相关输入。例如，您可以创建标签为“组织”的显示组并将名称和完整名称输入放置在“组织”组内，而不是分别为两种输入内容创建“组织名称”和“组织全名”标签。
- 对于步骤和显示组，还将添加工作流展示中显示的描述或备注。

插件 API

- API 文档指 `vso.xml` 文件和 Java 源文件中的所有文档。
- 对于 `vso.xml` 文件，查找器对象和脚本对象的描述采用与工作流和操作描述相同的规则。对象属性和方法参数的描述采用与工作流和操作参数描述相同的规则。
- 避免在 `vso.xml` 文件中使用特殊字符，并且需要在 `<![CDATA[insert your description here!]]>` 标记中包含描述。
- 对 Java 源文件使用标准 Javadoc 样式。

工作流启动时获取用户的输入参数

如果某工作流需要输入参数，它在启动时会打开一个对话框，让用户输入必需的输入参数值。您可以在工作流编辑器的**展示**选项卡中设置该对话框的内容、布局或展示。

工作流运行时，您在**展示**选项卡中对参数的设置将转换为输入参数对话框。

展示选项卡还允许您对输入参数添加说明，帮助用户正确提供输入参数。您还可以在**展示**选项卡中设置参数的属性和限制，以限制用户可提供的参数。如果用户提供的参数无法满足**展示**选项卡中设置的限制，工作流将不会运行。

- 在展示选项卡中创建输入参数对话框

  您可以定义对话框的布局，当用户在工作流编辑器的**展示**选项卡运行工作流时可以在该对话框中提供输入参数。

- 设置参数属性

  Orchestrator 可让您定义属性来限定用户在运行工作流时提供的输入参数值。您定义的参数属性会对用户提供的输入参数的类型和值进行强制性限制。

在展示选项卡中创建输入参数对话框

您可以定义对话框的布局，当用户在工作流编辑器的**展示**选项卡运行工作流时可以在该对话框中提供输入参数。

展示选项卡允许您对输入参数进行分类，并定义这些类别在输入参数对话框中的显示顺序。

展示说明

您可以对输入参数对话框中显示的每个参数或参数组添加相关说明。此说明将为用户提供相关信息，帮助他们输入正确的参数。您可以使用 HTML 格式来增强说明文本的布局。

定义展示输入步骤

默认情况下，输入参数对话框会采用一个列表列出所有必需的输入参数。为帮助用户键入输入参数，您可以在展示选项卡中定义相关节点，称为输入步骤。输入步骤会将同类输入参数归为一组。工作流运行时，同一输入步骤下的输入参数会显示在输入参数对话框中的不同部分。

定义展示显示组

每个输入步骤可以拥有自己的节点，称为显示组。显示组用于定义各参数输入文本框在输入参数对话框中各自部分的显示顺序。您可以独立于输入步骤对显示组进行定义。

创建输入参数对话框的展示

创建对话框的展示，用户可在工作流编辑器的**展示**选项卡中运行工作流时，在其中提供输入参数。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 验证工作流是否拥有输入参数的定义列表。

步骤

1. 在工作流编辑器中，单击**展示**选项卡。

   默认情况下，工作流的所有参数都会按创建时的顺序显示在**展示**节点下。

2. 右键单击**展示**节点并选择**创建新步骤**。

   展示节点下会显示一个**新步骤**节点。

3. 为步骤输入适当的名称，然后按 Enter 键。

   工作流运行时，此名称会在输入参数对话框中显示为某个部分的标题。

4. 单击输入步骤，并在**展示**选项卡下半部分的**常规**选项卡中添加说明。

   此说明会显示在输入参数对话框中，其中的信息可帮助用户提供正确的输入参数。您可以使用 HTML 格式来增强说明文本的布局。

5. 右键单击所创建的输入步骤，然后选择**创建显示组**。

   在输入步骤节点下会显示一个**新组**节点。

6. 为显示组输入适当的名称，然后按 Enter 键。

   工作流运行时，此名称会在输入参数对话框中显示为某个子部分标题。

7. 单击显示组，并在**展示**选项卡下半部分的**常规**选项卡中添加说明。

   此说明会显示在输入参数对话框中。您可以使用 HTML 格式来增强说明文本的布局。您使用使用 OGNL 语句（例如 `${#param}`）将参数值添加到组说明中。

8. 重复上述步骤，直到您已创建在工作流运行时将显示在输入参数对话框中的所有输入步骤和显示组为止。

9. 将**展示**节点下的参数拖放到您选择的步骤和组。

您即创建了输入参数对话框的布局，用户可在工作流运行时在其中提供输入参数值。

后续步骤

您必须设置参数属性。

设置参数属性

Orchestrator 可让您定义属性来限定用户在运行工作流时提供的输入参数值。您定义的参数属性会对用户提供的输入参数的类型和值进行强制性限制。

每个参数可有多个属性。您需要在**展示**选项卡中给定参数的**属性**选项卡内定义输入参数的属性。

参数属性会验证输入参数并修改文本框在输入参数对话框中的显示方式。有些参数属性可以在参数之间创建依赖关系。

静态和动态参数属性值

参数属性值可以是静态或动态的。静态属性值保持不变。如果将属性值设置为静态，则需要从工作流编辑器根据参数类型生成的列表中设置或选择属性值。

动态属性值则会取决于其他参数或属性的值。您需要使用对象图导航语言 (OGNL) 表达式定义动态属性获取值时所用的函数。如果动态参数属性值取决于其他参数属性值的值，并且该其他参数属性值会发生更改，那么 OGNL 表达式会重新计算并更改动态属性值。

设置参数属性

工作流启动时，会根据您设置的任何参数属性验证用户的输入参数值。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 验证工作流是否拥有输入参数的定义列表。

步骤

1. 在工作流编辑器中，单击**展示**选项卡。
2. 单击**展示**选项卡中的某个参数。

   该参数的**常规**和**属性**选项卡会显示在**展示**选项卡的下半部分。
3. 单击参数的**属性**选项卡。
4. 在**属性**选项卡中单击右键，然后选择**添加属性**。

   此时系统会打开一个对话框，列出了所选类型参数的可能属性。
5. 选择对话框中列出的某个属性，然后单击**确定**。

   该属性会显示在**属性**选项卡中。

6 在**值**项下，使用下拉菜单中的对应符号将属性值设置为静态或动态。

选项	描述
	静态属性
	动态属性

7 如果将属性值设置为静态，需要根据所设置属性的参数的类型选择一个属性值。

8 如果将属性值设置为动态，需要定义一个函数以使用 OGNL 表达式来获取参数属性值。

工作流编辑器会提供编写 OGNL 表达式的帮助。

a 单击  图标获取由工作流定义且可由此表达式调用的所有属性和参数列表。

b 单击  图标获取在 Orchestrator API 中可返回特定类型（您要为其定义属性）输出参数的所有操作列表，

单击参数和操作建议列表中的项目，可将其添加到 OGNL 表达式。

9 单击工作流编辑器底部的**保存**。

您即定义了工作流输入参数的属性。

后续步骤

验证和调试工作流。

Object Missing

This object is not available in the repository.

OGNL 表达式的预定义常量值

在创建 OGNL 表达式以获取动态参数属性值时，您可以使用预定义的常量值。

Orchestrator 定义以下常量值，可用于 OGNL 表达式。

表 1-52. 预定义 OGNL 常量值

常量值	说明
`${#__current}`	自定义验证属性或匹配表达式属性的当前值
`${#__username}`	启动工作流的用户名称
`${#__userdisplayname}`	显示启动工作流的用户名称
`${#__serverurl}`	包含用户启动工作流时所用服务器 IP 地址的 URL。该 URL 包含服务器 IP 地址和一个查找端口： `{ServerIP}:{lookupPort}`
`${#__datetime}`	当前日期和时间

表 1-52. 预定义 OGNL 常量值 （续）

常量值	说明
`${#__date}`	当前日期，时间设为 00:00:00
`${#__timezone}`	当前时区

在工作流运行时请求用户交互

工作流在运行时有时需要外部源中的其他输入参数。这些输入参数可来自其他应用程序或工作流，或者用户可直接提供。

例如，如果在工作流运行时发生特定事件，工作流可以请求人工交互以决定要采取什么操作。工作流在继续前会等待，直到用户响应信息请求或等待时间超出超时时间段为止。如果等待时间超出超时时间段，则工作流会返回异常。

用户交互的默认属性为 `security.group` 和 `timeout.date`。如果将 `security.group` 属性设置为指定 LDAP 用户组，即将用户交互请求的响应权限限制为该用户组成员。

如果设置 `timeout.date` 属性，您需要设置工作流等待用户信息的截止时间和日期。您可以设置绝对日期，或创建脚本工作流元素来计算相对于当前时间的相对时间。

步骤

1 （可选） 将用户交互添加到工作流

在工作流运行期间，您可通过向工作流添加**用户交互**架构元素的方式，请求用户输入参数。工作流在遇到**用户交互**元素时会挂起运行，等待用户提供所需数据。

2 （可选） 设置用户交互 security.group 属性

用户交互元素的 `security.group` 属性设置了哪些用户或用户组具有响应用户交互的权限。

3 （可选） 将 timeout.date 属性设置为绝对日期

设置用户交互的 `timeout.date` 属性以设置工作流需等待多长时间获得用户对于用户交互的响应。

4 （可选） 计算用户交互的相对超时

您可以在 `Date` 对象中计算用户交互超时的相对时间和日期。

5 （可选） 将 timeout.date 属性设置为相对日期

您可以将**用户交互**元素的 `timeout.date` 属性设置为相对时间和日期，方法是将其绑定到 `Date` 对象。您可在脚本函数中定义该对象。

6 （可选） 定义用户交互的外部输入

您需要指定在工作流运行期间用户必须提供的作为用户交互输入参数的信息。

7 （可选） 定义用户交互异常行为

如果用户未在超时时间段内提供输入参数，则用户交互会返回异常。您可以在脚本函数中定义异常行为。

8 （可选） 创建用户交互的输入参数对话框

在工作流运行期间，用户在输入参数对话框中提供输入参数，方法与工作流首次启动时提供输入参数一样。

9 （可选） 响应用户交互请求

要求在运行期间进行用户交互的工作流会挂起运行直至用户提供所需信息或工作流超时。

将用户交互添加到工作流

在工作流运行期间，您可通过向工作流添加**用户交互**架构元素的方式，请求用户输入参数。工作流在遇到**用户交互**元素时会挂起运行，等待用户提供所需数据。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**用户交互**元素拖放到工作流架构中的适当位置。

2 单击**用户交互**元素的**编辑**图标 (✏)。

3 在**信息**选项卡中提供用户交互的名称和说明，然后单击**关闭**。

4 单击**保存**。

您即向工作流添加了用户交互元素。工作流在遇到此元素时会挂起运行，等待来自用户的信息。

后续步骤

设置用户交互的 `security.group` 属性，将用户交互的响应权限限制为用户或用户组。请参见设置用户交互 security.group 属性。

设置用户交互 security.group 属性

用户交互元素的 `security.group` 属性设置了哪些用户或用户组具有响应用户交互的权限。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素和一个用户交互。
- 确定要响应用户交互请求的 LDAP 用户组。

步骤

1 单击工作流架构中**用户交互**元素的**编辑**图标 (✏)。

2 单击用户交互的**属性**选项卡。

3 针对 `security.group` 源参数单击**未设置**，设置哪些用户可以响应用户交互。

4 （可选） 选择**空**以允许所有用户响应用户交互请求。

5 若要将响应权限限制为特定用户或用户组，请单击**在工作流中创建参数/属性**。

此时会打开**参数信息**对话框。

6 命名参数。

7 选择**创建同名的工作流属性**在工作流中创建 `LdapGroup` 属性。

8 针对参数值单击**未设置**以打开 **LdapGroup** 选择框。

9 在**筛选器**文本框中输入 LDAP 用户组的名称。

10 从列表中选择 LDAP 用户组并单击**选择**。

例如，选择**管理员**组即表示只有该组内的成员才能响应用户交互请求。

您即限制了用户交互请求的响应权限。

11 单击**确定**以关闭**参数信息**对话框。

您即设置了用户交互的 `security.group` 属性。

后续步骤

设置 `timer.date` 属性以设置用户交互的超时时间段。

- 若要将超时设置为绝对日期和时间，请参见将 timeout.date 属性设置为绝对日期。
- 若要创建函数以计算相对于当前日期和时间的超时，请参见计算用户交互的相对超时。

将 timeout.date 属性设置为绝对日期

设置用户交互的 `timeout.date` 属性以设置工作流需等待多长时间获得用户对于用户交互的响应。

您可以在 `Date` 对象中设置绝对时间和日期。达到给定日期的时间后，等待用户交互的工作流会超时并以 `Failed` 状态结束。例如，您可以将用户交互设置为在 2 月 12 日中午超时。若要计算相对于当前时间和日期的超时，请参见计算用户交互的相对超时。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加用户交互元素。
- 设置用户交互的 `security.group` 属性。

步骤

1 单击工作流架构中**用户交互**元素的**编辑**图标 ( )。

2 单击用户交互的**属性**选项卡。

3 针对 `timeout.date` 源参数单击**未设置**，设置超时参数值。

4 （可选） 选择**空**以允许用户交互将工作流设置为无限期等待用户响应用户交互。

5 单击**在工作流中创建参数/属性**以设置工作流在超时后失效。

此时会打开**参数信息**对话框。

6 命名参数。

7 选择**创建同名的工作流属性**以在工作流中创建 `Date` 属性。

8 针对参数**值**单击**未设置**。

9 使用日历以选择工作流等待用户响应的截止绝对日期和时间。

10 单击**确定**关闭日历。

11 单击**确定**以关闭**参数信息**对话框。

`timeout.date` 属性即设置为绝对日期。如果用户未在此日期和时间前响应用户交互，则工作流将超时。

后续步骤

定义用户需提供的用户交互外部输入参数。请参见定义用户交互的外部输入。

计算用户交互的相对超时

您可以在 `Date` 对象中计算用户交互超时的相对时间和日期。

您可以在 `Date` 对象中设置绝对时间和日期。达到给定日期的时间后，用户交互的请求即超时。或者，您可以创建工作流元素，根据定义的函数计算并生成相对 `Date` 对象。例如，您可以创建将 24 小时添加到当前时间的相对 `Date` 对象。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加用户交互元素。
- 设置用户交互的 `security.group` 属性。

步骤

1 将**可编辑脚本任务**元素从**通用**菜单拖放到工作流的架构中，放在需要相对 `Date` 对象作为自己 `timeout.date` 属性的元素之前。

2 单击工作流架构中**可编辑脚本任务**元素的**编辑**图标 ( )。

3 在**信息**属性选项卡中，为脚本工作流元素输入名称和说明。

4 单击**输出**属性选项卡，然后单击**绑定到工作流参数/属性**图标 ( )。

5 单击**在工作流中创建参数/属性**以创建工作流属性。

a 为属性 `timerDate` 命名。

b 从属性类型列表中选择 `Date`。

c 选择**创建同名的工作流属性**。

d 将属性值设置保留为**未设置**，因为脚本函数会提供该值。

e 单击**确定**。

6 单击脚本工作流元素的**脚本**选项卡。

7 在**脚本**选项卡的脚本编辑器中，定义一个函数来计算并生成名为 `timerDate` 的 `Date` 对象。

例如，您可以实现以下 JavaScript 函数来创建 `Date` 对象，其中超时时间段是以毫秒为单位的相对延迟。

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

上述示例 JavaScript 函数定义了 `Date` 对象，该对象使用 `getTime` 方法获取当前日期和时间，并添加 86,400,000 毫秒或 24 小时。**可编辑脚本任务**元素会将该值生成为其输出参数。

8 单击**关闭**。

9 单击**保存**。

您即创建了可计算相对于当前时间和日期的相对时间和日期并生成 `Date` 对象的函数。**用户交互**元素可以以输入参数的形式接收该 `Date` 对象，从而设置等待用户输入的截止超时时间段。工作流到达**用户交互**元素时，会挂起其运行并等待用户提供所需信息或等待 24 小时后超时。

后续步骤

您必须将 `Date` 对象绑定到**用户交互**元素的 `timeout.date` 参数。请参见将 timeout.date 属性设置为相对日期。

将 timeout.date 属性设置为相对日期

您可以将**用户交互**元素的 `timeout.date` 属性设置为相对时间和日期，方法是将其绑定到 `Date` 对象。您可在脚本函数中定义该对象。

如果在脚本函数中创建相对 `Date` 对象，则可以将用户交互的 `timeout.date` 属性绑定到该 `Date` 对象。例如，如果将 `timeout.date` 属性绑定到会将 24 小时添加到当前时间的 `Date` 对象，则用户交互会在等待 24 小时后超时。

前提条件

- Add a user interaction element to the workflow schema.
- Set the `security.group` attribute for the user interaction.
- 创建可计算相对时间和日期的脚本函数并将其封装进工作流的 `Date` 对象中。请参见计算用户交互的相对超时。

步骤

1 单击工作流架构中**用户交互**元素的**编辑**图标 (✏️)。

2 单击用户交互的**属性**选项卡。

3 针对 `timeout.date` 源参数单击**未设置**，设置超时参数值。

4 选择封装了脚本函数中所定义相对时间和日期的 `Date` 对象，然后单击**选择**。

您即将 `timeout.date` 属性设置为脚本函数计算的相对日期和时间。

后续步骤

定义用户需提供的用户交互外部输入参数。请参见定义用户交互的外部输入。

定义用户交互的外部输入

您需要指定在工作流运行期间用户必须提供的作为用户交互输入参数的信息。

工作流到达用户交互元素时，会等待用户提供用户交互所需的作为其输入参数的信息。

前提条件

- 向工作流架构添加用户交互元素。
- Set the `security.group` attribute for the user interaction.
- 设置用户交互的 `timer.date` 属性。

步骤

1 单击工作流架构中**用户交互**元素的**编辑**图标 ( )。

2 单击**外部输入**选项卡。

3 单击**绑定到工作流参数/属性**图标 ( ) 以定义用户必须在用户交互中提供的参数。

4 （可选） 如果已经在工作流中定义了输入参数，请从建议列表中选择参数。

5 单击**在工作流中创建参数/属性**以创建工作流属性，从而绑定到用户提供的输入参数。

6 为参数命名适当的名称。

7 通过在**筛选**对话框中搜索对象类型，从类型列表中选择输入参数类型。

例如，如果用户交互需要用户提供虚拟机作为输入参数，则选择 `VC:VirtualMachine`。

8 选择**创建同名的工作流属性**，将用户提供的输入参数绑定到工作流中的新属性。

9 将输入参数值设置保留为**未设置**。

用户在工作流运行期间响应用户交互时，需提供此值。

10 单击**确定**以关闭**参数信息**对话框。

您即定义了用户在用户交互时要提供的输入参数。

后续步骤

定义用户交互发生错误时的异常行为。请参见定义用户交互异常行为。

定义用户交互异常行为

如果用户未在超时时间段内提供输入参数，则用户交互会返回异常。您可以在脚本函数中定义异常行为。

如果未定义用户交互超时后工作流要采取的操作，则工作流会以 `Failed` 状态结束。定义异常行为是一个很好的工作流开发做法。

前提条件

- 向工作流架构添加用户交互元素。
- 设置用户交互的 `security.group` 和 `timer.date` 属性。
- 定义用户交互的外部输入参数。

步骤

1. 单击工作流架构中**用户交互**元素的**编辑**图标 (✏️)。
2. 单击**异常**选项卡。
3. 对于输出异常绑定，单击**未设置**。
4. 单击**在工作流中创建参数/属性**来创建要将用户交互绑定到的异常属性。

   此时会打开**参数信息**对话框。
5. 创建 `errorCode` 属性。

   `errorCode` 属性具有以下默认属性：

   - 名称：**errorCode**
   - 类型：字符串
   - 创建：**创建同名的工作流属性**
   - 值：输入相应的错误消息。
6. 单击**确定**以关闭**参数信息**对话框。
7. 将可编辑脚本的任务元素拖放到工作流架构中用户交互元素上。

   两个元素之间会显示一个红色虚线箭头，表示异常链接。可编辑脚本的任务元素会自动绑定到用户交互的 `errorCode` 属性。
8. 双击可编辑脚本的任务元素并提供相应名称。

   例如，**`Log timeout`**。
9. 在可编辑脚本任务元素的**脚本**选项卡中，编写 JavaScript 函数以处理异常。

   例如，若要在 Orchestrator 日志中记录超时，请编写以下函数：

   ```
   System.log("No response from user. Timed out.");
   ```
10. 将用于处理异常的可编辑脚本任务元素链接并绑定到在工作流中跟随该元素的元素。

    例如，将可编辑脚本任务元素链接并绑定到**出现异常**元素以结束发生错误的工作流。

您即定义了用户交互超时后的异常行为。

后续步骤

创建用户可在其中提供输入参数的对话框。请参见创建用户交互的输入参数对话框。

创建用户交互的输入参数对话框

在工作流运行期间，用户在输入参数对话框中提供输入参数，方法与工作流首次启动时提供输入参数一样。

您需要在用户交互元素的**展示**选项卡中创建对话框的布局，而不是在整个工作流的**展示**选项卡中创建。整个工作流的**展示**选项卡用于创建在启动工作流时显示的输入参数对话框布局。用户交互元素的**展示**选项卡用于创建当工作流在运行时到达用户交互元素时打开的输入参数对话框布局。

前提条件

- 向工作流架构添加用户交互元素。
- 设置用户交互的 `security.group` 和 `timer.date` 属性。
- 定义用户交互的外部输入参数。
- 定义异常行为。

步骤

1. 单击工作流架构中**用户交互**元素的**编辑**图标 (✏)。
2. 单击用户交互元素的**展示**选项卡。

   展示选项卡显示了为用户交互创建的外部输入参数。
3. （可选） 右键单击**展示**选项卡中的**展示**节点并选择**创建新步骤**。

   这些步骤可让您创建对话框中的各个部分，包含可在其中整理输入参数的描述和标题。
4. （可选） 右键单击**展示**选项卡中的**展示**节点并选择**创建显示组**。

   显示组可让您对步骤中显示的输入参数进行排序，并允许您将子标题和说明添加到对话框。
5. 单击列表中的输入参数并在该参数的**常规**选项卡中添加输入参数的描述。

   您输入的描述文本会在输入参数对话框中显示为标签，以通知用户在响应用户交互时必须提供的信息。
6. 定义输入参数属性。

   输入参数属性可让您限定用户可以提供的输入参数值，并使用 OGNL 表达式动态确定参数值。
7. 单击**保存并关闭**以关闭工作流编辑器。

您即创建了输入参数对话框，用户可在其中提供用于在工作流运行时响应用户交互的输入参数。

后续步骤

有关创建展示步骤和组以及设置输入参数属性的信息，请参见在展示选项卡中创建输入参数对话框。

响应用户交互请求

要求在运行期间进行用户交互的工作流会挂起运行直至用户提供所需信息或工作流超时。

需要用户交互的工作流定义了哪些用户可以提供所需的信息并引导交互请求。

前提条件

验证至少有一个工作流处于“等待用户交互”状态。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**运行**。

2 单击 Orchestrator 客户端中的**我的 Orchestrator** 视图。

3 单击**等待输入**选项卡。

等待输入选项卡列出了正等待您或具有权限的用户组成员进行输入的工作流。

4 双击正等待输入的工作流。

等待输入的工作流令牌显示在**工作流**层次结构列表中，并一同显示以下符号：  。

5 右键单击工作流令牌并选择**响应**。

6 按照输入参数对话框中的说明操作，提供工作流所需的信息。

您即提供了工作流在运行时等待用户输入的信息。

在工作流中调用工作流

工作流可以在运行期间调用其他工作流。一个工作流可以启动另一个工作流，可能因为它需要将其他工作流的运行结果作为自己的输入参数，或者因为它可以启动一个工作流并让其独立运行。工作流也可在未来给定时间启动另一工作流或同时启动多个工作流。

- 用于调用工作流的工作流元素

  您可以使用四种方法在一个工作流中调用其他工作流。每种调用工作流的方法都由不同的工作流架构元素来表示。

- 同步调用工作流

  同步调用工作流可以将被调用的工作流作为调用方工作流的一部分来运行。调用方工作流在运行后续架构元素时可以使用被调用工作流的输出参数作为自己的输入参数。

- 异步调用工作流

  异步调用工作流可以将被调用的工作流独立于调用方工作流来运行。调用方工作流则可继续运行，无需等待被调用的工作流完成运行。

- 调度工作流

  您可以从某个工作流调用一个工作流，并调度其在稍后的时间和日期启动。

- 从其他工作流内部调用远程工作流的必备条件

  如果您开发的工作流调用了驻留在远程 Orchestrator 服务器的其他工作流，必须满足特定的必备条件才能成功运行远程工作流。

- 同时调用多个工作流

  同时调用多个工作流可以将被调用的工作流同步作为调用方工作流的一部分来运行。调用方工作流会等待所有被调用工作流完成运行后继续运行。调用方工作流在运行后续架构元素时可以使用被调用工作流的运行结果作为自己的输入参数。

用于调用工作流的工作流元素

您可以使用四种方法在一个工作流中调用其他工作流。每种调用工作流的方法都由不同的工作流架构元素来表示。

同步工作流

一个工作流可以同步启动另一工作流。被调用的工作流在运行时会作为调用方工作流运行的组成部分，并与调用方工作流在同一内存空间中运行。调用方工作流会启动另一工作流，等待被调用工作流运行结束，然后再继续运行其架构中的下一元素。通常，同步调用工作流的目的在于，调用方工作流需要将被调用工作流的输出结果作为后续架构元素的输入参数。例如，工作流可以调用“启动虚拟机并等待”工作流来启动虚拟机，然后获取该虚拟机的 IP 地址以传递给其他元素，或通过电子邮件传递给用户。

异步工作流

一个工作流可以异步启动另一工作流。调用方工作流会启动另一工作流，但调用方工作流会立即运行其架构中的下一元素，无需等待被调用工作流的结果。被调用工作流会使用由调用方工作流定义的输入参数运行，但被调用工作流的生命周期则与调用方工作流的生命周期无关。异步工作流可让您创建工作流链，将输入参数从一个工作流传递到下一个工作流。例如，工作流可以在运行期间创建各种对象。该工作流随后可启动异步工作流，这些异步工作流会在运行中使用这些对象作为自己的输入参数。原始工作流会在启动所有必需工作流并运行完剩余的元素后结束。但是，其启动的异步工作流会继续独立运行。

若要设置调用方工作流等待被调用工作流的结果，请使用嵌套工作流或创建一个可编辑脚本任务，用于检索被调用工作流的工作流令牌状态，然后检索工作流完成后的结果。

调度工作流

工作流在调用另一工作流时，可将该工作流推迟到稍后的时间和日期再启动。调用方工作流会继续运行，直到结束为止。调用被调度的工作流时，会创建一个任务，用于在给定的时间和日期启动该工作流。在调用方工作流运行后，您可以在 Orchestrator 客户端的**调度程序**和**我的 Orchestrator** 视图中查看被调度的工作流。

被调度的工作流只会运行一次。您可以在同步工作流的可编辑脚本任务元素中调用 `Workflow.scheduleRecurrently` 方法，来调度某个工作流重复运行。

嵌套工作流

一个工作流可在单个架构元素中嵌套多个工作流，从而同时启动多个工作流。当调用方工作流到达架构中的嵌套工作流元素时，嵌套工作流元素中所

列的所有工作流就会同时启动。值得注意的是，每个嵌套工作流启动时的内存空间均与调用方工作流不同。调用方工作流会等待所有嵌套工作流完成运行后，再启动自己架构中的下一元素。调用方工作流在运行剩余元素时，会使用嵌套工作流的结果作为输入参数。

将工作流更改传播到其他工作流

如果从其它工作流中调用工作流，Orchestrator 会在您向架构添加工作流元素时，导入父工作流中的子工作流输入参数。

如果您在将子工作流添加到另一个工作流之后又对其进行了修改，则父工作流会调用新版子工作流，但不会导入任何新的输入参数。为防止这些工作流更改影响调用该工作流的其他工作流的行为，Orchestrator 不会自动将新输入参数传播到调用方工作流。

若要将一个工作流的参数传播到调用它的其他工作流中，您必须找到调用该工作流的工作流，然后手动同步两个工作流。

前提条件

验证您有一个被其它工作流调用的工作流。

步骤

1 修改并保存由其他工作流调用的工作流。

2 关闭工作流编辑器。

3 在 Orchestrator 客户端的**工作流**视图中，从层次结构列表中导航至您更改过的工作流。

4 右键单击该工作流并选择**参考 > 查找使用该元素的元素**。

此时系统将显示调用此工作流的工作流列表。

5 双击列表中的某个工作流，使其在 Orchestrator 客户端的**工作流**视图中突出显示。

6 右键单击该工作流并选择**编辑**。

此时系统会打开工作流编辑器。

7 单击工作流编辑器中的**架构**选项卡。

8 右键单击工作流架构中被更改工作流的工作流元素，然后选择**同步 > 同步参数**。

9 在确认对话框中选择**继续**。

10 保存并关闭工作流编辑器。

11 对使用已修改工作流的所有工作流，重复步骤 5 到步骤 10。

您即成功将工作流更改传播到了调用它的其他工作流中。

将子工作流的输入参数和展示传播到父工作流

如果您开发了一个工作流用于调用其他工作流，则可以将子工作流的输入参数和展示传播到父工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**运行**。

2 右键单击想要修改的工作流并选择**编辑**。

此时会打开工作流编辑器。

3 选择**架构**选项卡。

4 右键单击要将其输入参数和展示传播到父工作流的子工作流元素，然后选择**同步 > 同步展示**。

5 在确认对话框中，选择**确定**。

6 （可选） 对于要将其输入参数和展示传播到父工作流的所有子工作流重复 步骤 4 和 步骤 5。

子工作流的输入参数会添加到父工作流的输入参数。父工作流的展示会使用子工作流的展示进行扩展。

同步调用工作流

同步调用工作流可以将被调用的工作流作为调用方工作流的一部分来运行。调用方工作流在运行后续架构元素时可以使用被调用工作流的输出参数作为自己的输入参数。

您可以使用**工作流**元素从另一个工作流中同步调用工作流。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**工作流**元素从**通用**菜单拖放到工作流架构中的适当位置。

此时系统会显示**选择工作流**选择对话框。

2 搜索并选择所需的工作流，然后单击**确定**。

如果搜索只返回了部分结果，请从客户端的**工具 > 用户首选项**菜单中扩大搜索条件或增加搜索结果数量。

3 单击**工作流**元素以在**架构**选项卡的下半部分显示元素属性。

4 单击工作流架构中**工作流**元素的**编辑**图标 ( )。

5 将所需的输入参数与工作流架构元素的**输入**选项卡绑定。

6 将所需的输出参数与工作流架构元素的**输出**选项卡绑定。

7 在**异常**选项卡上定义工作流的异常行为。

8 单击**关闭**。

9 单击工作流编辑器底部的**保存**。

您即从其他工作流中同步调用了一个工作流。当该工作流在运行过程中到达同步工作流时，则同步工作流会启动，而初始工作流会等待该同步工作流结束运行后再继续运行。

后续步骤

您可以从工作流中异步调用工作流。

异步调用工作流

异步调用工作流可以将被调用的工作流独立于调用方工作流来运行。调用方工作流则可继续运行，无需等待被调用的工作流完成运行。

您可以使用**异步工作流**元素从另一个工作流中异步调用工作流。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1. 将**异步工作流**元素从**通用**菜单拖放到工作流架构中的适当位置。

   此时系统会显示**选择工作流**选择对话框。
2. 从列表中搜索并选择所需的工作流，然后单击**确定**。
3. 单击工作流架构中**异步工作流**元素的**编辑**图标 ( )。
4. 将所需的输入参数与异步工作流元素的**输入**选项卡绑定。
5. 将所需的输出参数与异步工作流元素的**输出**选项卡绑定。

   您可将输出参数绑定至被调用的工作流或该工作流的结果。
   - 绑定到被调用的工作流会将该工作流作为输出参数进行返回。
   - 绑定到被调用工作流的工作流令牌会返回被调用工作流的运行结果。
6. 在**异常**选项卡上定义异步工作流元素的异常行为。
7. 单击**关闭**。
8. 单击工作流编辑器底部的**保存**。

您即从其他工作流中异步调用了一个工作流。当该工作流在运行过程中到达异步工作流时，则异步工作流会启动，而初始工作流会继续运行，无需等待异步工作流结束运行。

后续步骤

您可调度工作流在稍后的时间和日期启动。

调度工作流

您可以从某个工作流调用一个工作流，并调度其在稍后的时间和日期启动。

您可使用**调度工作流**元素，在其他工作流中调度工作流

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1. 将**调度工作流**元素从**通用**菜单拖放到工作流架构中的适当位置。
2. 在文本框中输入要调用的工作流的部分名称即可搜索该工作流。
3. 从列表中选择工作流，然后单击**确定**。
4. 单击工作流架构中**调度工作流**元素的**编辑**图标 (✏)。
5. 单击**输入**属性选项卡。

   在要定义的属性列中会显示一个名为 `workflowScheduleDate` 的参数，还会显示调用方工作流的输入参数。
6. 为 `workflowScheduleDate` 参数单击**未设置**以设置参数。
7. 单击**在工作流中创建参数/属性**以创建参数并设置参数值。
8. 为**值**单击**未设置**以设置参数值。
9. 使用显示的日历来设置用于启动已调度工作流的日期和时间，然后单击**确定**。
10. 将剩余输入参数与已调度工作流元素的**输入**选项卡中的已调度工作流绑定。
11. 将所需的输出参数与已调度工作流元素的**输出**选项卡中的 `Task` 对象绑定。
12. 在**异常**选项卡中定义已调度工作流元素的异常行为。
13. 单击**关闭**。
14. 单击工作流编辑器底部的**保存**。

您即调度了工作流在给定时间和日期从其他工作流中启动。

后续步骤

您可以从一个工作流中同时调用多个工作流。

从其他工作流内部调用远程工作流的必备条件

如果您开发的工作流调用了驻留在远程 Orchestrator 服务器的其他工作流，必须满足特定的必备条件才能成功运行远程工作流。

- 远程工作流的所有输入参数必须可在远程 Orchestrator 服务器上进行解析。

- 远程工作流的所有输出参数必须可在本地 Orchestrator 服务器上进行解析。

为确保远程工作流的参数可进行解析，工作流使用的清单对象必须可同时用于远程和本地 Orchestrator 服务器。如果远程工作流使用某个插件中的对象，则同一插件必须可同时用于这两台 Orchestrator 服务器。远程插件和本地插件的清单必须相同。如果远程工作流使用 Orchestrator 中的系统对象（例如工作流和操作），则远程和本地 Orchestrator 服务器的清单中必须存在相同的工作流和操作。

例如，假设您在开发的测试工作流中的嵌套工作流元素内插入了重命名虚拟机工作流。您想要在远程 Orchestrator 服务器上运行重命名虚拟机工作流。运行测试工作流时，重命名虚拟机工作流会在测试工作流运行时被调用。您指定了本地 Orchestrator 服务器清单中要重命名的虚拟机。由于重命名虚拟机工作流在远程 Orchestrator 服务器上运行，因此同一虚拟机必须也存在于该服务器的清单中。否则，重命名虚拟机工作流无法解析其 `vm` 输入参数。因此，本地 vCenter Server 插件和远程 Orchestrator 服务器必须连接到同一 vCenter Server 实例。

同时调用多个工作流

同时调用多个工作流可以将被调用的工作流同步作为调用方工作流的一部分来运行。调用方工作流会等待所有被调用工作流完成运行后继续运行。调用方工作流在运行后续架构元素时可以使用被调用工作流的运行结果作为自己的输入参数。

您可以使用**嵌套工作流**元素从另一个工作流中同时调用多个工作流。您可以使用嵌套工作流运行这些用户凭据与调用方工作流所用凭据不同的工作流。

前提条件

- 打开要在工作流编辑器中编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**嵌套工作流**元素从**操作和工作流**菜单拖放到工作流架构中的适当位置。

此时系统会显示**选择工作流**选择对话框。

2 搜索并选择要启动的工作流，然后单击**确定**。

3 单击工作流架构中**嵌套工作流**元素的**编辑**图标 ( )。

4 单击**工作流**选项卡。

您在 步骤 2 选中的工作流会显示在选项卡中。

5 在**工作流**架构元素属性选项卡右侧面板的**输入**和**输出**选项卡中设置该工作流的“输入”和“输出”绑定。

6 单击**工作流**架构元素属性选项卡右侧面板中的**连接信息**选项卡。

连接信息选项卡可让您使用合适的凭据访问非本地服务器的其他服务器上存储的工作流。

7 要访问远程服务器上的工作流，请选择**远程**并单击**未设置**以提供远程服务器的主机名或 IP 地址。

注 您可以使用 vRealize OrchestratorMulti-Node 插件来调用远程服务器上的工作流。

8 定义访问远程服务器时所需的凭据。

- 选择**继承**可使用与运行调用方工作流的用户相同的凭据。
- 选择**动态**并单击**未设置**可选择使用由 `credentials` 类型的参数在工作流其他地方定义的一组凭据。
- 选择**静态**并单击**未设置**可直接输入凭据。

9 单击**工作流**选项卡中的**添加工作流**按钮，可选择多个要添加到嵌套工作流元素中的工作流。

10 重复 步骤 2 至 步骤 8 定义所添加的每个工作流的设置。

11 单击工作流架构中的嵌套工作流元素。

元素中所嵌套的工作流数量会以数字形式显示在嵌套工作流元素上。

您即从一个工作流中同时调用了多个工作流。

后续步骤

您可以定义长时间运行的工作流。

在选择的对象上运行工作流

您可以在选择的对象上运行工作流来自动执行重复任务。例如，您可以创建一个工作流来为虚拟机文件夹中的所有虚拟机拍摄快照，或者创建一个工作流来关闭指定主机上所有虚拟机的电源。

您可以使用以下任一方法在选择的对象上运行工作流。

- 运行**库 > vCenter > 批处理 > 在选择的对象上运行工作流**工作流。
- 创建一个工作流来调用**库 > Orchestrator > 串行启动工作流**或**并行启动工作流**的工作流。
- 创建一个工作流来获取对象数组并在工作流元素循环的数组中每个对象上运行工作流。
- 在工作流中脚本元素的 `For` 循环中调用 `Workflow.execute()` 方法以通过 JavaScript 运行工作流。

选择哪种方法在一组选定对象上运行工作流取决于要运行的工作流，并且会影响工作流的性能。例如，运行“在选择的对象上运行工作流”工作流是在多个对象上运行工作流最简单的方法，并且无需开发工作流，但只能运行采用单个输入参数的工作流。

创建一个工作流来调用“串行启动工作流”或“并行启动工作流”的工作流，可让您在多个对象上运行采用多个输入参数的工作流。调用工作流必须创建属性数组以将输入参数传递到“串行启动工作流”或“并行启动工作流”工作流。这些工作流仅可在其他工作流中使用。请勿直接运行。

在脚本元素的 `For` 循环中运行工作流比在工作流元素循环中运行工作流要快得多，但却受制于灵活性以及重复使用的可能性。最重要的是，在脚本循环中运行工作流会丢失检查点，Orchestrator 在启动工作流运行中的每个元素时，都会执行这些检查点。因此，如果 Orchestrator 服务器在脚本循环运行时停止，则在服务器重新启动后，工作流会从脚本元素的起始位置恢复并重复整个循环。如果 Orchestrator 服务器在通过工作流元素循环运行工作流时停止，则工作流会在服务器停止时正在运行的循环中特定元素处恢复。

有关批处理工作流的更多信息，请参见《使用 VMware vRealize Orchestrator 插件》。

有关如何创建一个工作流以在工作流元素循环内的对象数组上运行工作流，请参见开发复杂工作流。

有关如何在脚本 `For` 循环中运行工作流，请参见工作流脚本示例。

实现“串行启动工作流”和“并行启动工作流”工作流

您可以使用“串行启动工作流”和“并行启动工作流”工作流在一组选定的对象上运行工作流。

您无法直接运行“串行启动工作流”和“并行启动工作流”工作流。您必须将其包含在您创建的另一工作流中。若要使用“串行启动工作流”和“并行启动工作流”工作流在一组选定的对象上运行工作流，您必须获取要在其上运行工作流的对象。将这些对象以及工作流所需的任何其他输入参数作为属性数组传递到工作流。“串行启动工作流”和“并行启动工作流”工作流会将在一组选定对象上运行的工作流运行结果处理为一个 `WorkflowToken` 对象数组。

“串行启动工作流”和“并行启动工作流”工作流的实现方法相同。“串行启动工作流”工作流会在每个对象上按顺序运行工作流。“并行启动工作流”工作流会在所有对象上同时运行工作流。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 在工作流架构中，添加可编辑脚本任务元素或操作以获取要在其上运行工作流的对象列表。

例如，若要在某个虚拟机文件夹中的所有虚拟机上运行工作流，您可以将 `getAllVirtualMachinesByFolder` 操作添加到工作流。

2 链接脚本元素或操作并将其输入和输出绑定到工作流输入或属性。

例如，您可以将 `getAllVirtualMachinesByFolder` 操作的 `vmFolder` 输入绑定到工作流输入参数并将 `actionResult` 输出绑定到调用工作流的工作流属性中。

3 添加可编辑脚本任务元素以将对象列表转换为属性数组。

例如，如果要在其上运行工作流的对象是虚拟机阵列（即 `allVMs`），且由 `getAllVirtualMachinesByFolder` 操作的 `actionResult` 输出返回，您可以编写以下脚本将对象转换为属性数组。

```
propsArray = new Array();

for each (var vm in allVMs) {
 var prop = new Properties();
 prop.put("vm", vm);
 propsArray.push(prop);
}
```

4 将可编辑脚本任务元素的输入和输出绑定到工作流属性。

在步骤 3 的示例可编辑脚本任务元素中，将输入绑定到虚拟机的 `allVMs` 阵列并将 `propsArray` 输出属性创建为一个 `Properties` 对象数组。

5 将工作流元素添加到工作流架构。

6 选择“串行启动工作流”或“并行启动工作流”工作流，并将工作流元素链接到其他元素。

7 将“串行启动工作流”或“并行启动工作流”工作流的 `wf` 输入绑定到要在对象上运行的工作流。

例如，若要移除 `getAllVirtualMachinesByFolder` 操作返回的所有虚拟机的任何快照，请选择“移除所有快照”工作流。

8 将“串行启动工作流”或“并行启动工作流”工作流的 `parameters` 输入绑定到 `Properties` 对象数组，其中该数组包含了工作流要在其上运行的对象。

例如，将 `parameters` 输入绑定到步骤 4 中定义的 `propsArray` 属性。

9 （可选） 将“串行启动工作流”或“并行启动工作流”的工作流 `workflowTokens` 输出绑定到工作流的属性。

10 （可选） 继续添加更多使用“串行启动工作流”或“并行启动工作流”工作流运行结果的元素。

您即创建了使用“串行启动工作流”或“并行启动工作流”的工作流，从而在一组选定对象上运行工作流。

开发长时间运行的工作流

处于等待状态的工作流会消耗系统资源，因为该工作流会不断轮询能向其提供响应的对象。如果您知道某个工作流可能需要很长时间才能收到所需的响应，您可以向该工作流添加长时间运行的工作流元素。

每个正在运行的工作流都会消耗一个系统线程。当工作流遇到长时间运行的工作流元素时，长时间运行的工作流元素会将工作流设置为被动状态。然后，长时间运行的工作流元素会将工作流信息传递给负责向系统轮询服务器中所有正在运行的长时间运行工作流元素的单个线程。长时间运行的工作流元素会在设定的时间内保持被动状态，由长时间运行的工作流线程代其轮询系统，而不是每个长时间运行的工作流元素都不断尝试从系统检索信息。

您可以通过以下任一方法设置等待时间：

- 设置一个封装到 `Date` 对象的定时器，它会挂起工作流并直到特定时间和日期为止。您可以在架构中添加**等待定时器**元素，实现基于计时器的长时间运行的工作流元素。
- 定义一个封装到 `Trigger` 对象的触发器事件，它会在触发器事件发生后重新启动工作流。您可以在架构中添加**等待事件**或**用户交互**元素，实现基于触发器的长时间运行的工作流元素。

为基于定时器的工作流设置相对时间和日期

您可以将“等待定时器”元素的 `timer.date` 属性设置为相对时间和日期，方法是将其与 `Date` 对象绑定。您可在脚本函数中定义 `Date` 对象。

到达给定日期的时间时，基于定时器的长时间运行工作流会重新激活并继续运行。例如，您可以将工作流设置在 2 月 12 日中午重新激活。或者，您可以创建工作流元素，根据定义的函数计算并生成相对 `Date` 对象。例如，您可以创建将 24 小时添加到当前时间的相对 `Date` 对象。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**可编辑脚本任务**元素从**通用**菜单拖放到工作流的架构中，放在需要相对 Date 对象作为自己 timeout.date 属性的元素之前。

2 单击工作流架构中**可编辑脚本任务**元素的**编辑**图标 ( )。

3 在**信息**属性选项卡中，为脚本工作流元素输入名称和说明。

4 单击**输出**属性选项卡，然后单击**绑定到工作流参数/属性**图标 ( )。

5 单击**在工作流中创建参数/属性**以创建工作流属性。

   a 为属性 timerDate 命名。

   b 从属性类型列表中选择 Date。

   c 选择**创建同名的工作流属性**。

   d 将属性值设置保留为**未设置**，因为脚本函数会提供该值。

   e 单击**确定**。

6 单击脚本工作流元素的**脚本**选项卡。

7 在**脚本**选项卡的脚本编辑器中，定义一个函数来计算并生成名为 timerDate 的 Date 对象。

   例如，您可以实现以下 JavaScript 函数来创建 Date 对象，其中超时时间段是以毫秒为单位的相对延迟。

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

   上述示例 JavaScript 函数定义了 Date 对象，该对象使用 getTime 方法获取当前日期和时间，并添加 86,400,000 毫秒或 24 小时。**可编辑脚本任务**元素会将该值生成为其输出参数。

8 单击**关闭**。

9 单击**保存**。

您即创建了用于计算并生成 Date 对象的函数。**等待定时器**元素会将该 Date 对象接收为输入参数，挂起某个长时间运行的工作流，直到该对象中封闭的日期为止。工作流在到达**等待定时器**元素时会挂起运行并等待 24 小时后继续运行。

后续步骤

您必须将**等待定时器**元素添加到工作流以实现基于定时器的长时间运行工作流。

创建基于定时器的长时间运行工作流

如果您知道某个工作流必须在可预测的时间内等待外部源的响应，可将其实现为基于定时器的长时间运行工作流。基于定时器的长时间运行工作流会等待给定时间和日期，然后才会恢复运行。

使用**等待定时器**元素将工作流实现为基于定时器的长时间运行工作流。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1 将**等待定时器**元素从**通用**菜单拖放到工作流架构中要挂起工作流运行的位置。

如果使用可编辑脚本任务来计算时间和日期，则此元素必须置于**等待定时器**元素之前。

2 单击工作流架构中**等待定时器**元素的**编辑**图标 ( )。

3 在**信息**特性选项卡中输入实现定时器的原因说明。

4 单击**属性**特性选项卡。

此时 `timer.date` 参数会显示在属性列表中。

5 单击 `timer.date` 参数的**未设置**按钮将参数绑定到适当的 `Date` 对象。

此时系统会打开**等待定时器**选择对话框，显示可能的绑定列表。

- 从建议列表中选择预定义的 `Date` 对象，例如由某个**可编辑脚本任务**元素在工作流其他地方定义的对象。
- 或者，创建一个设置了工作流需要等待的特定日期和时间的 `Date` 对象。

6 （可选） 创建一个设置了工作流需要等待的特定日期和时间的 `Date` 对象。

a 在**等待定时器**选择对话框中单击**在工作流中创建参数/属性**。

此时系统会显示**参数信息**对话框。

b 为参数输入适当的名称。

c 将类型设置保留为 `Date`。

d 单击**创建同名的工作流属性**。

e 单击**值**特性的**未设置**按钮以设置参数值。

此时系统会打开一个日历。

f 使用该日历设置需要重新启动工作流的日期和时间。

g 单击**确定**。

7 单击**关闭**。

8 单击工作流编辑器底部的**保存**。

您即定义了一个定时器，它会挂起基于定时器的长时间运行工作流，直到设定的时间和日期为止。

后续步骤

您可以创建一个长时间运行工作流，会等待触发器事件后再运行。

创建触发器对象

触发器对象会监视插件所定义的事件触发器。例如，vCenter Server 插件将这些事件定义为 Task 对象。当任务结束时，触发器会向基于等待触发器的长时间运行工作流元素发送一条消息，重新启动该工作流。

基于触发器的长时间运行工作流所等待的耗时事件必须返回一个 VC:Task 对象。例如，用来启动虚拟机的 startVM 操作会返回一个 VC:Task 对象，以便工作流中的后续元素可监视其进度。基于触发器的长时间运行工作流的触发器事件需将此 VC:Task 对象用作输入参数。

您在**可编辑脚本**元素的 JavaScript 函数中创建 Trigger 对象。此**可编辑脚本任务**元素可属于等待触发器事件的基于触发器的长时间运行工作流。或者，可属于另一个为基于触发器的长时间运行工作流提供输入参数的工作流。触发器函数必须实施 Orchestrator API 中的 createEndOfTaskTrigger() 方法。

重要事项 您必须为所有触发器定义超时时段，否则工作流会无限期等待。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。
- 在工作流中，将 VC:Task 对象声明为属性或输入参数，例如工作流中的 VC:Task 对象或会启动或克隆虚拟机的工作流元素。

步骤

1 将**可编辑脚本任务**从**通用**菜单拖放到工作流的架构中。

可编辑脚本任务之前的任一元素必须生成一个 VC:Task 对象作为其输出参数。

2 单击工作流架构中**可编辑脚本任务**元素的**编辑**图标 ( )。

3 在**信息**属性选项卡中，为触发器输入名称和说明。

4 单击**输入**属性选项卡。

5 单击**绑定到工作流参数/属性**图标 ( )。

此时系统会打开输入参数选择对话框。

6 搜索或创建一个类型为 **VC:Task** 的输入参数。

此 VC:Task 对象表示由另一工作流或元素启动的耗时事件。

7 （可选） 搜索或创建一个类型为数字的输入参数，以秒为单位定义超时时段。

8 单击**输出**属性选项卡。

9 单击**绑定到工作流参数/属性**图标 ( )。

此时系统会打开输出参数选择对话框。

10 创建包含以下属性的输出参数。

   a 创建值为 `trigger` 的名称属性。

   b 创建值为 `Trigger` 的类型属性。

   c 单击**创建同名的属性**来创建属性。

   d 将值保留为**未设置**。

11 在**异常**属性选项卡中定义任何异常行为。

12 定义一个可在**脚本**选项卡中生成 `Trigger` 对象的函数。

   例如，可以实现以下 JavaScript 函数来创建 `Trigger` 对象。

```
trigger = task.createEndOfTaskTrigger(timeout);
```

   `createEndOfTaskTrigger()` 方法会返回一个 `Trigger` 对象，用于监视名为 `task` 的 `VC:Task` 对象。

13 单击**关闭**。

14 单击工作流编辑器底部的**保存**。

您即定义了一个可为基于触发器的长时间运行工作流创建触发器事件的工作流元素。触发器元素会生成一个 `Trigger` 对象作为其输出参数，**等待事件**元素可绑定到该输出参数。

后续步骤

您必须将此触发器事件绑定到基于触发器的长时间运行工作流中的**等待事件**元素。

创建基于触发器的长时间运行工作流

如果您知道某个工作流在运行期间必须等待外部源的响应，但又不知道会等待多长时间，可将其实现为基于触发器的长时间运行工作流。基于触发器的长时间运行工作流会等待所定义的触发器事件发生，然后再恢复运行。

使用**等待事件**元素将工作流实现为基于触发器的长时间运行工作流。基于触发器的长时间运行工作流在到达**等待事件**元素时，会挂起运行并进入被动等待状态，直到收到来自触发器的消息为止。在等待期间，被动工作流不会消耗线程，相反，长时间运行的工作流会将工作流信息传递到负责监视服务器中所有长时间运行工作流的单个线程。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。
- 定义封装在 `Trigger` 对象中的触发器事件。

步骤

1 将**等待事件**元素从**通用**菜单拖放到工作流架构中要挂起工作流运行的位置。

   用于声明触发器的可编辑脚本任务必须紧靠在**等待事件**元素之前。

2 单击工作流架构中**等待事件**元素的**编辑**图标 ( )。

3 在**信息**属性选项卡中输入等待的原因说明。

4 单击**属性**特性选项卡。

此时 `trigger.ref` 参数会显示在属性列表中。

5 单击 `trigger.ref` 参数的**未设置**链接将参数绑定到适当的 `Trigger` 对象。

此时系统会打开**等待事件**选择对话框，显示可能绑定的参数列表。

6 从建议列表中选择预定义的 `Trigger` 对象。

此 `Trigger` 对象表示由另一工作流或工作流元素定义的触发器事件。

7 在**异常**属性选项卡中定义任何异常行为。

8 单击**关闭**。

9 单击工作流编辑器底部的**保存**。

您即定义了一个可挂起基于触发器的长时间运行工作流并等待特定触发器事件发生后再重新启动的工作流元素。

后续步骤

您可以运行工作流。

配置元素

配置元素是一个属性列表，您可用这些属性在整个 Orchestrator 服务器范围内配置各种常量。

在特定 Orchestrator 服务器中运行的所有工作流、操作和策略均可使用您在配置元素中所设置的属性。在配置元素中设置属性可让您将相同的属性值用于在 Orchestrator 服务器中运行的所有工作流、操作和策略。

如果您创建了一个包含工作流、操作或策略的软件包，其中需要使用配置元素中的属性，则 Orchestrator 会自动在此软件包中包含相应的配置元素。如果将某个包含配置元素的软件包导入至另一个 Orchestrator 服务器中，则您可以同时导入该配置元素的属性值。例如，如果您创建了一个工作流，其中所需的属性值取决于该工作流运行时所在的 Orchestrator 服务器，则在配置元素中设置这些属性可让您导出该工作流，以用于另一台 Orchestrator 服务器。因此，配置元素可让您在两个服务器之间更为轻松地交换相关工作流、操作和策略。

注 您无法从 Orchestrator 5.1 或更早期版本导出的配置元素中导入配置元素属性值。

创建配置元素

配置元素可让您在整个 Orchestrator 服务器范围内设置常用属性。服务器中运行的所有元素都可以调用您在配置元素中设置的属性。创建配置元素可让您在服务器中一次性定义各种常用属性，而不是在每个元素中分别定义。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**配置**视图。

3 在文件夹层次结构列表中右键单击某个文件夹，然后选择**新建文件夹**以创建新文件夹。

4 输入文件夹的名称，然后单击**确定**。

5 右键单击所创建的文件夹，然后选择**新建元素**。

6 输入配置元素的名称，然后单击**确定**。

此时系统会打开配置元素编辑器。

7 在**常规**选项卡中单击版本数字以增加版本号，并提供版本备注。

8 在**常规**选项卡的**说明**文本框中，输入配置元素的说明。

9 单击**属性**选项卡。

10 单击**添加属性**图标 () 以创建新属性。

11 在**名称**、**类型**、**值**和**说明**下分别单击属性值以设置属性名称、类型、值和说明。

12 单击**权限**选项卡。

13 单击**添加访问权限**图标 ()，向一组用户授予该配置元素的访问权限。

14 在**筛选**文本框中搜索用户组，然后从建议列表中选择相关用户组。

15 勾选相应复选框，为选定用户组设置访问权限。

您可以对配置元素设置以下权限。

权限	说明
查看	用户可以查看配置元素，但无法查看架构或脚本。
检查	用户可以查看配置元素，包括架构和脚本。
管理员	用户可对配置元素中的元素设置权限，并拥有所有其他权限。
执行	用户可以运行配置元素中的元素。
编辑	用户可以编辑配置元素中的元素。

16 单击**选择**。

17 单击**保存并关闭**以退出配置元素编辑器。

您即定义了可在整个 Orchestrator 服务器范围内设置常用属性的配置元素。

后续步骤

您可以使用配置元素来提供工作流或操作的属性。

工作流用户权限

Orchestrator 会定义您对用户组可应用的权限级别，从而允许或拒绝其访问工作流。

查看	用户可以查看工作流中的元素，但无法查看架构或脚本。
检查	用户可以查看工作流中的元素，包含架构和脚本。
执行	用户可以运行工作流。
编辑	用户可以编辑工作流。
管理员	用户可以对工作流设置权限并拥有所有其他权限。

管理员权限包含**查看**、**检查**、**编辑**和**执行**权限。所有权限都需要**查看**权限。

如果未对工作流设置任何权限，则工作流会继承包含该工作流的文件夹的权限。如果对工作流设置了权限，则这些权限会覆盖包含该工作流的文件夹的权限，即使文件夹权限的限制更严格也是如此。

对工作流设置用户权限

您可以对工作流设置权限级别，限制用户组对该工作流的访问权限。

您可以从 Orchestrator LDAP 服务器选择要为其设置权限的用户和用户组。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。
- 向工作流架构添加一些元素。

步骤

1. 单击**权限**选项卡。
2. 单击**添加访问权限**图标 () 为新用户组定义权限。
3. 搜索用户组。

   搜索结果会包含 Orchestrator LDAP 服务器中匹配搜索项的所有用户组。
4. 选择用户组并选择相应复选框以对该用户组设置权限级别。

   若要允许该用户组中的某个用户查看工作流、检查架构和脚本、运行和编辑工作流以及更改权限，您必须选择所有复选框。
5. 单击**选择**。

   用户组会显示在权限列表中。

6 单击**保存并关闭**以退出该编辑器。

验证工作流

Orchestrator 提供了工作流验证工具。验证工作流有助于识别工作流中的错误，并检查相邻元素之间的数据流动是否正确。

验证工作流时，验证工具会创建一个包含所有错误或警告的列表。在列表中单击某个错误可突出显示包含该错误的工作流元素。

如果在工作流编辑器中运行验证工具，则工具会对其检测到的错误提供快速修复建议。部分快速修复需要您提供其他信息或输入参数，而其他快速修复则会为您解决错误。

工作流验证会检查元素之间的数据绑定和连接。工作流验证不会检查每个元素在工作流中执行的数据处理。因此，如果架构元素中的某个函数不正确，那么即使有效的工作流也会出现运行错误并产生错误结果。

默认情况下，Orchestrator 会在运行工作流时始终执行工作流验证。您可以在 Orchestrator 客户端中更改默认验证行为。请参见在开发过程中测试工作流。例如：在工作流开发过程中，您有时可能会出于测试目的运行明知无效的工作流。

验证工作流并修复验证错误

运行工作流之前必须先对其进行验证。您可以在 Orchestrator 客户端或工作流编辑器中验证工作流。但是，您只能在工作流编辑器中打开了要编辑的工作流，对验证错误进行修复。

前提条件

确认要验证的工作流是否完整，架构元素是否已链接相关绑定是否已定义。

步骤

1 单击**工作流**视图。

2 在**工作流**层次结构列表中导航到工作流。

3 （可选） 右键单击该工作流并选择**验证工作流**。

如果工作流有效，则会显示确认消息。如果工作流无效，则会显示错误列表。

4 （可选） 关闭“工作流验证”对话框。

5 右键单击该工作流并选择**编辑**以打开工作流编辑器。

6 单击**架构**选项卡。

7 单击**架构**选项卡工具栏中的**验证**按钮。

如果工作流有效，则会显示确认消息。如果工作流无效，则会显示错误列表。

8 对于无效的工作流，请单击错误消息。

验证工具会通过添加红色图标的方式来突出显示发生错误的架构元素。在可能的情况下，验证工具会显示快速修复操作。

- 如果同意建议的快速修复操作，请单击以执行该操作。
- 如果不同意建议的快速修复操作，请关闭“工作流验证”对话框并手动修复架构元素。

重要事项 请始终检查 Orchestrator 建议的修复是否合适。

例如：建议的操作可能要删除未使用的属性，而事实上该属性却可能是绑定不正确。

9 重复上述步骤，直到消除所有验证错误为止。

您即验证了工作流并修复了验证错误。

后续步骤

您可以运行工作流。

调试工作流

Orchestrator 提供了工作流调试工具。您可以调试工作流以在任何活动开始时检查输入和输出参数以及属性、以编辑模式在工作流运行时替换参数或属性值，以及从上次失败活动中恢复工作流。

您可以调试标准工作流库中的工作流以及自定义工作流。您可以在工作流编辑器中开发自定义工作流，同时对其进行调试。

调试工作流

您可以将断点添加到工作流架构中的元素来调试元素。

到达断点时，对于如何继续调试操作，您有多种选择。调试工作流架构的元素时，您可以查看有关工作流运行的常规信息、修改工作流变量和查看日志消息。

前提条件

以能够运行工作流的用户身份登录到 Orchestrator 客户端。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 选择工作流库中的工作流，然后单击**架构**选项卡。

4 若要将断点添加到想要调试的架构元素，请右键单击工作流元素并选择**切换断点**。

您可以启用或禁用已切换的断点。

5 单击**调试工作流**图标 ( )。

如果工作流需要输入参数，则您必须提供。

6 当工作流到达断点并暂停运行时，请选择任一可用选项。

选项	描述
恢复	恢复工作流运行，直到到达另一个断点为止。
跳入	可让您跳入工作流元素。 **注** 当您在工作流编辑器中调试工作流时，您不可以跳入嵌套的工作流元素。
跳过	跳过架构中的当前元素并在下一元素上暂停工作流运行。
跳出	退出您已跳入的工作流元素

7 （可选） 在**断点**选项卡上，修改断点。

您可以启用、禁用或移除现有断点。

8 （可选） 在**变量**选项卡上，查看变量。

您可以在调试过程中修改部分变量的值。

示例工作流调试

您可以调试标准工作流库中的工作流。

例如，如果提供错误的收件人地址，可以在使用电子邮件工作流调试示例交互时更正相关值。

前提条件

以能够运行邮件工作流的用户身份登录到 Orchestrator 客户端。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 在工作流层次结构列表中，打开**库 > 邮件**。

4 通过电子邮件工作流选择示例交互，然后单击**架构**选项卡。

5 右键单击**电子邮件发送(交互)**工作流元素并选择**切换断点**。

6 单击**调试工作流**图标 ( )。

7 提供所需信息。

a 在**目标地址**文本框中，输入不完整的收件人地址。

例如：*name@company*.c。

b 选择有权回答查询的 LDAP 用户组。

c 单击**提交**。

8 到达断点时，单击**跳入**图标 ( )。

9 在**变量**选项卡上，验证相关值。

10 在 **toAddress** 文本框中，输入正确的收件人地址值。

例如：*name@company*.com。

11 单击**恢复**图标 ( ) 以继续工作运行。

工作流会使用调试过程中提供的值并继续工作流运行。

运行工作流

Orchestrator 工作流会根据事件的逻辑流来运行。

运行工作流时，工作流中的每个架构元素会根据以下顺序运行。

1 工作流会将工作流令牌属性和输入参数与架构元素的输入参数进行绑定。

2 架构元素运行。

3 架构元素的输出参数会复制到工作流令牌属性和工作流输出参数。

4 工作流令牌属性和输出参数存储在数据库中。

5 下一架构元素开始运行。

每个架构元素都会重复此顺序，直到工作流结束为止。

工作流令牌检查点

工作流运行时，每个架构元素都是一个检查点。在每个架构元素运行后，Orchestrator 会在数据库中存储工作流令牌属性，同时下一架构元素开始运行。如果工作流意外停止，那么在 Orchestrator 服务器下次重新启动时，当前活动的架构元素会再次运行，工作流会从发生中断时正在运行的架构元素开头继续运行。但是，Orchestrator 不会实现事务管理或回滚函数。

工作流结束

如果当前活动的架构元素为结束元素，则工作流结束。在工作流到达结束元素后，其他工作流或应用程序可以使用该工作流的输出参数。

在工作流编辑器中运行工作流

您可以在开发工作流的同时就运行该工作流。

在工作流编辑器中运行工作流可让您验证工作流是否能正确运行，同时无需中断开发过程。您可以查看日志消息获得有关工作流运行的信息。如果工作流运行返回意外结果，您可以修改工作流并再次运行，且无需关闭工作流编辑器。

前提条件

- 创建工作流。
- 在工作流编辑器中打开要编辑的工作流。

- 验证工作流。

步骤

1 单击**架构**选项卡。

2 单击**运行**。

3 （可选） 在**日志**选项卡中查看消息。

运行工作流

您可以运行标准库中的工作流或自己创建的工作流，从而在 vCenter Server 中执行自动化操作。

例如，可以运行“创建简单虚拟机”工作流来创建虚拟机。

前提条件

确认已配置了 vCenter Server 插件。有关详细信息，请参见《安装和配置 VMware vCenter Orchestrator》。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**运行**或**设计**。

2 单击**工作流**视图。

3 在工作流层次结构列表中，打开**库 > vCenter > 虚拟机管理 > 基本**以导航到“创建简单虚拟机”工作流。

4 右键单击“创建简单虚拟机”工作流，然后选择**启动工作流**。

5 在**启动工作流**输入参数对话框中提供以下信息，从而在 Orchestrator 连接的 vCenter Server 中创建虚拟机。

选项	操作
虚拟机名称	将虚拟机命名为 `orchestrator-test`。
虚拟机文件夹	a 单击**未设置**作为**虚拟机文件夹**的值。 b 从清单中选择虚拟机文件夹。 **选择**按钮为非活动，直到您选择了正确类型的对象为止，在本例中为 `VC:VmFolder`。
新磁盘大小 (GB)	输入适当的数值。
内存大小 (MB)	输入适当的数值。
虚拟 CPU 的数量	从**虚拟 CPU 的数量**下拉菜单中选择适当的 CPU 数量。
虚拟机客户机操作系统	单击**未设置**链接并从列表中选择客户机操作系统。
要在其上创建虚拟机的主机	单击**未设置**作为**要在其上创建虚拟机的主机**的值，然后通过 vCenter Server 基础架构层次结构导航到主机。
资源池	单击**未设置**作为**资源池**的值，然后通过 vCenter Server 基础架构层次结构导航到资源池。

选项	操作
要连接到的网络	单击**未设置**作为**要连接到的网络**的值，然后选择网络。 在**筛选器**文本框中按 Enter 键以查看所有可用网络。
要在其中存储虚拟机文件的数据存储	单击**未设置**作为**要在其中存储虚拟机文件的数据存储**的值，然后通过 vCenter Server 基础架构层次结构导航到数据存储。

6 单击**提交**以运行工作流。

此时在“创建简单虚拟机”工作流下会显示一个工作流令牌，显示工作流正在运行的图标。

7 单击该工作流令牌可查看工作流在运行时的状态。

8 单击工作流令牌视图中的**事件**选项卡可跟踪工作流令牌的进度，直到其完成为止。

9 单击**清单**视图。

10 通过 vCenter Server 基础架构层次结构导航到您定义的资源池。

如果列表中未显示虚拟机，请单击刷新按钮以重新加载清单。

`orchestrator-test` 虚拟机会显示在资源池中。

11 （可选） 在**清单**视图中右键单击 `orchestrator-test` 虚拟机可查看一张上下文列表，其中列出了您可在 `orchestrator-test` 虚拟机上运行的工作流。

“创建简单虚拟机”工作流即运行成功。

后续步骤

您可以登录 vSphere Client 并管理新虚拟机。

恢复失败的工作流运行

如果工作流失败，Orchestrator 会提供一个选项，可从上次失败的活动中恢复工作流运行。

您可以更改工作流的参数并尝试恢复工作流运行，或者保留参数并对影响工作流运行的外部组件进行更改。例如，如果因为第三方系统问题导致工作流运行失败，您可以对系统进行更改并从失败的活动中恢复工作流运行，无需更改工作流参数和重复成功的活动。

设置失败工作流运行的恢复行为

您可以为每个自定义工作流设置恢复失败运行时的行为。库中默认工作流在恢复失败工作流运行时会使用默认的系统设置。

您可以修改配置文件来更改默认系统行为。请参见设置用于恢复失败工作流运行的自定义属性。

前提条件

验证您已具有工作流的编辑权限。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 展开工作流层次结构列表以导航到要为其设置行为的工作流。

4 右键单击工作流并选择**编辑**。

此时会打开工作流编辑器。

5 在**常规**选项卡上，从**从失败恢复的行为**下拉菜单中选择选项。

选项	描述
系统默认值	按照默认行为恢复。
已启用	如果工作流运行失败，弹出窗口会显示恢复工作流运行的选项。
已禁用	如果工作流运行失败，则无法恢复。

6 单击**保存并关闭**。

设置用于恢复失败工作流运行的自定义属性

默认情况下，Orchestrator 未设置如何恢复失败的工作流运行。您可以启用 Orchestrator 来恢复失败的工作流运行并自定义设置在超时多久后失败的工作流运行将无法恢复。

步骤

1 在 Orchestrator 服务器系统中，导航到 `/etc/vco/app-server/`。

2 在文本编辑器中打开 `vmo.properties` 配置文件。

3 在 `vmo.properties` 文件中编辑以下行以设置 Orchestrator 来恢复失败的工作流运行。

```
com.vmware.vco.engine.execute.resume-from-failed=true
```

4 在 `vmo.properties` 文件中编辑以下行以设置恢复失败工作流运行的自定义超时时间段。

```
com.vmware.vco.engine.execute.resume-from-failed.timeout-sec=<seconds>
```

您设置的值会替换现有的 86400 秒默认超时设置。

5 保存 `vmo.properties` 文件。

6 重新启动 Orchestrator 服务器。

恢复失败的工作流运行

如果工作流启用了恢复失败运行，您可以将工作流运行从上次失败的活动中恢复。

启用了恢复失败的工作流运行选项后，您可以使用工作流失败后显示的弹出窗口中的选项，更改工作流的参数并尝试进行恢复。您还可以保留参数并对影响工作流运行的外部组件进行更改。如果未选择任何选项，工作流运行会超时并且无法恢复。有关修改超时时间段的信息，请参见设置用于恢复失败工作流运行的自定义属性。

步骤

1 在弹出窗口的下拉菜单中，选择**恢复**并单击**下一步**。

如果选择**取消**，则工作流运行稍后将无法恢复。

2 （可选） 修改工作流参数。

3 单击**提交**。

生成工作流文档

您可以将在任意时间选择的工作流或工作流文件夹的相关文档以 PDF 格式导出。

导出的文档包含有关选定工作流或文件夹中工作流的详细信息。每个工作流相关的信息包括名称、工作流版本历史记录、属性、参数展示、工作流架构和工作流操作。此外，文档还提供了所用操作的源代码。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**运行**或**设计**。

2 单击**工作流**视图。

3 导航到要生成文档的工作流或工作流文件夹，然后右键单击。

4 选择**生成文档**。

5 浏览并找到要在其中保存 PDF 文件的文件夹，然后提供文件名并单击**保存**。

包含选定工作流或文件夹中工作流相关信息的 PDF 文件会保存在您的系统中。

使用工作流版本历史记录

您可以使用版本历史记录将工作流恢复为先前保存的状态。您可以将工作流状态恢复为较早或较新的工作流版本。您还可以比较当前状态的工作流与已保存版本的工作流之间的差异。

在您增加并保存工作流版本时，Orchestrator 会为每个工作流创建新的版本历史记录项目。对工作流的后续更改不会更改当前保存的版本。例如，如果您创建工作流版本 1.0.0 并进行保存，工作流的状态会存储在版本历史记录中。如果对工作流进行任意更改，可以在 Orchestrator 客户端中保存工作流状态，但您无法将更改应用到工作流版本 1.0.0。如果要在版本历史记录中存储更改，必须创建并保存一个后续工作流版本。版本历史记录会连同工作流一起保存在数据库中。

当您删除工作流时，Orchestrator 会在数据库中将元素标记为已删除，而不会将元素的版本历史记录从数据库中删除。这样您就可以还原已删除的工作流。请参见还原已删除工作流。

前提条件

打开要在工作流编辑器中编辑的工作流。

步骤

1 单击工作流编辑器中的**常规**选项卡，然后单击**显示版本历史记录**。

2 选择工作流版本并单击**与当前版本的差异**以比较差异。

窗口会显示当前工作流版本与选定工作流版本之间的差异。

3 选择工作流版本并单击**恢复**以还原工作流状态。

小心 如果尚未保存当前工作流版本，则其会从版本历史记录中删除，并且无法恢复到当前版本。

工作流状态会恢复到选定版本的状态。

还原已删除工作流

您可以还原已从工作流库中删除的工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**运行**或**设计**。

2 单击**工作流**视图。

3 导航到要在其中还原已删除工作流的工作流文件夹。

4 右键单击文件夹并选择**还原已删除工作流**。

5 选择要还原的工作流并单击**还原**。

还原的工作流随即会显示在选定文件夹中。

开发简单示例工作流

开发简单示例工作流可以展示工作流开发过程中最常见的步骤。

您将要创建的示例工作流应启动 vCenter Server 中一台现有的虚拟机，并向管理员发送电子邮件以确认虚拟机已启动。

示例工作流应执行以下任务：

1 提示用户选择一台要启动的虚拟机。

2 提示用户输入要接收通知的电子邮件地址。

3 检查选定的虚拟机是否已打开电源。

4 向 vCenter Server 实例发送启动虚拟机的请求。

5 等待 vCenter Server 启动虚拟机，如果虚拟机无法启动或启动时间过长，则返回错误。

6 等待 vCenter Server 在虚拟机上启动 VMware Tools，如果虚拟机无法启动或 VMware Tools 启动时间过长，则返回错误。

7 确认虚拟机正在运行。

8 向提供的电子邮件地址发送通知，通知虚拟机已启动或发生错误。

从 Orchestrator 文档登录页面下载的 Orchestrator 示例 ZIP 文件包含完整版的“启动虚拟机并发送电子邮件”工作流。

开发示例工作流的过程包含多项任务。

前提条件

在尝试开发简单示例工作流之前，请阅读工作流的主要概念。

步骤

1 创建简单工作流示例

您必须在 Orchestrator 客户端中创建工作流来开始工作流开发过程。

2 创建简单工作流示例的架构

您可以在工作流编辑器中创建工作流的架构。工作流架构包含工作流运行的元素，并确定工作流的逻辑流。

3 （可选） 创建简单工作流示例区域

您可以通过添加不同颜色的工作流备注，突出显示工作流中的不同区域。创建不同的工作流区域有助于使复杂工作流架构更易于阅读和理解。

4 定义简单工作流示例的参数

在此阶段的工作流开发中，您需要定义工作流运行时所需的输入参数。在本示例工作流中，您需要虚拟机打开电源时的输入参数，以及向用户通知该工作流结果时所用电子邮件地址的参数。用户在运行工作流时需要指定要打开电源的虚拟机，并提供电子邮件地址。

5 定义简单工作流示例决策绑定

您可在工作流编辑器的**架构**选项卡中将工作流的元素绑定在一起。决策绑定用于定义决策元素将收到的输入参数与决策语句进行比较的方式，并根据这两者是否匹配来生成输出参数。

6 绑定简单工作流示例的操作元素

您可在工作流编辑器中将工作流的元素绑定在一起。绑定用于定义这些操作元素对输入参数的处理方式和输出参数的生成方式。

7 绑定简单工作流示例的脚本任务元素

您可在工作流编辑器的**架构**选项卡中将工作流的元素绑定在一起。绑定用于定义这些脚本任务元素对输入参数的处理方式和输出参数的生成方式。您还可以将可编辑脚本任务元素与其 JavaScript 函数进行绑定。

8 定义简单工作流示例异常绑定

您可以在工作流编辑器中的**架构**选项卡中定义异常绑定。异常绑定用于定义元素对错误的处理方式。

9 设置简单工作流示例属性的读写特性

您可以将参数和属性定义为只读常量或是可写变量。您还可以限制用户可以作为输入参数所提供的值。

10 设置简单工作流示例参数属性

您可以在工作流编辑器中设置参数属性。设置参数属性会影响参数的行为，并会对该参数的可能值产生限制。

11 设置简单工作流示例输入参数对话框的布局

在工作流编辑器中创建输入参数对话框的布局或展示。用户在运行需要输入参数才能运行的工作流时，输入参数对话框即会打开。

12 验证并运行简单工作流示例

创建工作流后，可以对其进行验证以发现任何可能的错误。如果不包含任何错误，则可以运行该工作流。

创建简单工作流示例

您必须在 Orchestrator 客户端中创建工作流来开始工作流开发过程。

前提条件

确认系统上已安装并配置了以下组件。

- vCenter Server，控制部分虚拟机，其中至少一台虚拟机已关闭电源
- SMTP 服务器的访问权限
- 有效的电子邮件地址

有关如何安装和配置 vCenter Server 的信息，请参见《vSphere 安装和设置》文档。有关如何配置 Orchestrator 使用 SMTP 服务器的信息，请参见《安装和配置 VMware vRealize Orchestrator》。

若要编写工作流，您的 Orchestrator 用户帐户必须对服务器或在用工作流文件夹至少拥有**查看**、**执行**、**检查**、**编辑**以及**管理**（最好拥有）权限。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 右键单击工作流列表的根，然后选择**添加文件夹**。

4 将新文件夹命名为**工作流示例**，然后单击**确定**。

5 右键单击**工作流示例**文件夹，然后选择**新建工作流**。

6 将新工作流命名为**启动虚拟机并发送电子邮件**，然后单击**确定**。

此时系统会打开工作流编辑器。

7 在**常规**选项卡上，单击版本号数字以增加版本号。

由于这是初次创建的工作流，请将版本设置为 **0.0.1**。

8 单击**常规**选项卡中的**服务器重新启动行为**以设置工作流是否在服务器重新启动后恢复。

9 在**常规**选项卡的**说明**文本框中，输入工作流的功能说明。

例如，您可以添加以下说明。

此工作流会启动虚拟机并向 Orchestrator 管理员发送确认电子邮件。

10 单击**常规**选项卡底部的**保存**。

您即创建了名为“启动虚拟机并发送电子邮件”的工作流，但您尚未定义其函数。

后续步骤

创建工作流的架构。

创建简单工作流示例的架构

您可以在工作流编辑器中创建工作流的架构。工作流架构包含工作流运行的元素，并确定工作流的逻辑流。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 在**通用**菜单中，将决策元素拖放到架构中链接 Start 元素和 End 元素的箭头上。

3 双击决策元素并将其更名为**虚拟机已打开电源?**。

决策元素对应的布尔函数会检查虚拟机是否已打开电源。

4 在**通用**菜单中，将操作元素拖放到链接决策元素和 End 元素的红色箭头上。

此时系统会显示操作选择的对话框。

5 在**筛选**文本框中输入 **start**，从筛选的操作列表中选择 **startVM** 操作，然后单击**选择**。

6 将以下操作元素按顺序分别拖放到链接 startVM 操作元素和 End 元素的蓝色箭头上。

vim3WaitTaskEnd	挂起工作流运行，并对正在进行的 vCenter Server 任务定期执行 Ping 操作，直到任务完成为止。startVM 操作会启动虚拟机，vim3WaitTaskEnd 操作会在虚拟机启动时将工作流设置为等待。虚拟机启动后，vim3WaitTaskEnd 会让工作流恢复运行。
vim3WaitToolsStarted	挂起工作流运行并等待，直到 VMware Tools 在目标虚拟机上启动为止。

7 在**通用**菜单中，将可编辑脚本任务元素拖放到链接 vim3WaitToolsStarted 操作元素和 End 元素的蓝色箭头上。

8 双击可编辑脚本任务元素，并将其重命名为**确定**。

9 将另一可编辑脚本任务元素拖放到绿色箭头处，链接 VM powered on? 决策元素和 End 元素，并将此可编辑脚本任务元素命名为**已启动**。

10 修改 Already started 可编辑脚本任务元素的链接。

a 将 Already started 可编辑脚本任务元素拖放到 startVM 操作元素的左侧。

b 删除连接 Already started 可编辑脚本任务元素和 End 元素的蓝色箭头。

c 使用蓝色箭头将 Already started 可编辑脚本任务元素链接到 vim3WaitToolsStarted 操作元素。

11 在**通用**菜单中，将以下可编辑脚本任务元素拖放到架构中。

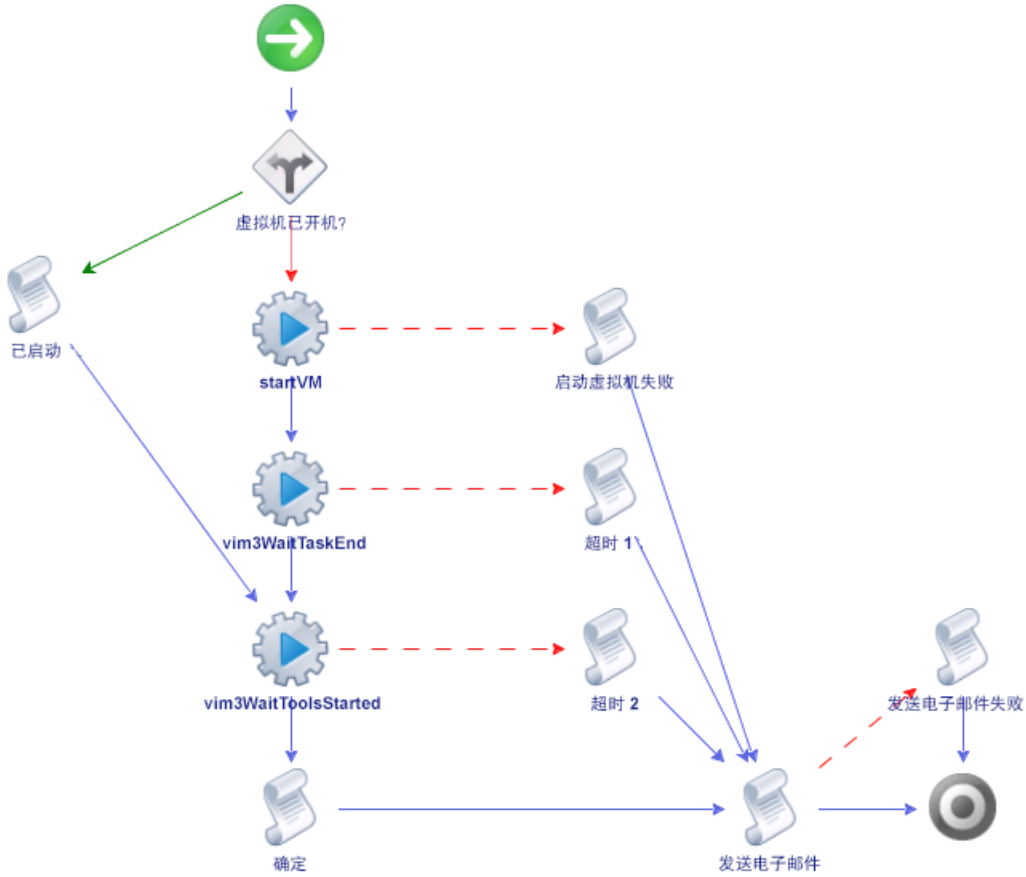
- 将可编辑脚本任务元素拖放到 startVM 操作元素上，并将该可编辑脚本任务元素命名为**启动虚拟机失败**。
- 将可编辑脚本任务元素拖放到 vim3WaitTaskEnd 操作元素上，并将该可编辑脚本任务元素命名为**超时 1**。
- 将可编辑脚本任务元素拖放到 vim3WaitToolsStarted 操作元素上，并将该可编辑脚本任务元素命名为**超时 2**。
- 将可编辑脚本任务元素拖放到链接 OK 可编辑脚本任务元素和 End 元素的蓝色箭头上，将新的可编辑脚本任务元素命名为**发送电子邮件**，并将其拖放到 OK 可编辑脚本任务元素的右侧。
- 使用蓝色箭头将 Start VM Failed、Timeout 1 和 Timeout 2 可编辑脚本任务元素链接到 Send Email 可编辑脚本任务元素。
- 将可编辑脚本任务元素拖放到 Send Email 可编辑脚本任务元素上，将新的可编辑脚本任务元素命名为**发送电子邮件失败**，将其拖放到 Timeout 2 可编辑脚本任务元素的右侧，然后使用蓝色箭头将其链接到 End 元素。

12 将 End 元素拖放到 Send Email 可编辑脚本任务元素的右侧。

13 单击**架构**选项卡底部的**保存**。

下图显示了“启动虚拟机并发送电子邮件”工作流架构各元素的布局。

图 1-10. “启动虚拟机并发送电子邮件示例”工作流中各元素的链接



后续步骤

您可以突出显示工作流中的不同区域。

创建简单工作流示例区域

您可以通过添加不同颜色的工作流备注，突出显示工作流中的不同区域。创建不同的工作流区域有助于使复杂工作流架构更易于阅读和理解。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1. 将工作流备注元素从**通用**菜单拖放到工作流编辑器中。
2. 将工作流备注置于 `Already started` 脚本任务元素上方。
3. 拖动工作流备注边缘调整其大小，将其环绕在 `Already started` 脚本任务元素周围。

4 双击文本并添加说明。

例如，**虚拟机已打开电源时的路径**。

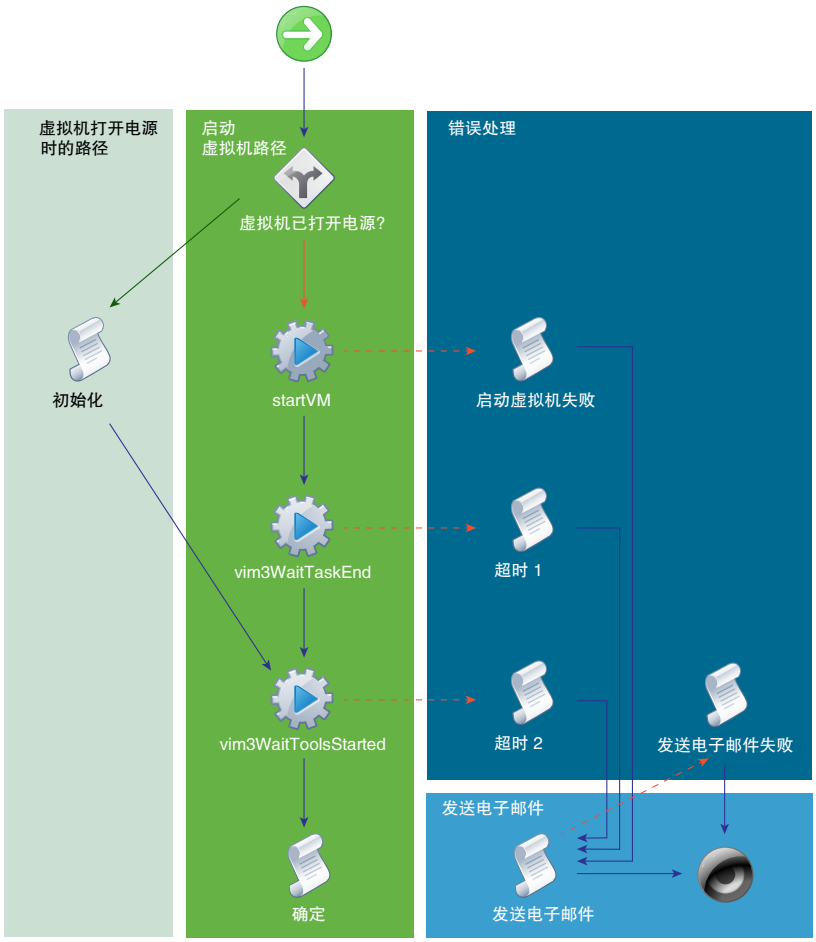
5 按 Ctrl+E 选择背景颜色。

6 重复上述步骤以突出显示工作流中的其他区域。

- 将备注置于从 `VM powered on?` 决策元素到 `OK` 元素的元素垂直序列上方。添加说明**启动虚拟机路径**。
- 将备注置于 `startVM failed`（包括 `Timeout` 可编辑脚本任务元素和 `Send Email Failed` 可编辑脚本任务元素）的上方。添加说明**错误处理**。
- 将备注置于 `Send Email` 可编辑脚本任务元素上方。添加说明**发送电子邮件**。

下图显示了示例工作流区域的表现形式。

图 1-11. “启动虚拟机并发送电子邮件”示例工作流区域



后续步骤

您必须定义工作流的属性及输入和输出参数。

定义简单工作流示例的参数

在此阶段的工作流开发中，您需要定义工作流运行时所需的输入参数。在本示例工作流中，您需要虚拟机打开电源时的输入参数，以及向用户通知该工作流结果时所用电子邮件地址的参数。用户在运行工作流时需要指定要打开电源的虚拟机，并提供电子邮件地址。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1. 单击工作流编辑器中的**输入**选项卡。
2. 在**输入**选项卡中单击右键，然后选择**添加参数**。

   一个名为 `arg_in_0` 的参数会显示在**输入**选项卡中。
3. 单击 **`arg_in_0`**。
4. 在“选择属性名称”对话框中输入 **`vm`** 作为名称，然后单击**确定**。
5. 单击**类型**文本框，然后在参数类型对话框的搜索文本框中输入 **`vc:virtualm`**。
6. 从建议参数类型列表中选择 **`VC:VirtualMachine`**，然后单击**接受**。
7. 在**说明**文本框中添加关于参数的说明。

   例如，输入**要打开电源的虚拟机**。
8. 重复步骤 2 至步骤 7，使用以下值创建第二个输入参数。
   - 名称：`toAddress`
   - 类型：字符串
   - 说明：**接收该工作流结果的电子邮件地址**
9. 单击**输入**选项卡底部的**保存**。

您即定义了工作流的输入参数。

后续步骤

定义元素参数之间的绑定。

定义简单工作流示例决策绑定

您可在工作流编辑器的**架构**选项卡中将工作流的元素绑定在一起。决策绑定用于定义决策元素将收到的输入参数与决策语句进行比较的方式，并根据这两者是否匹配来生成输出参数。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 在**架构**选项卡上，单击 **VM Powered On?** 决策元素的**编辑**图标 ( )。

2 在**决策**选项卡上，单击**未设置 (空)** 按钮，在建议参数列表中选择 **vm** 作为决策元素的输入参数。

3 从下拉菜单中建议的决策语句列表中选择**电源状态等于**语句。

值文本框中会显示**未设置**按钮，提供有限的可能值供您选择。

4 选择 **poweredOn**。

5 单击工作流编辑器**架构**选项卡底部的**保存**。

您即定义了 true 或 false 语句，决策元素会据此比较所收到输入参数的值。

后续步骤

您必须为工作流中的其他元素定义绑定。

绑定简单工作流示例的操作元素

您可在工作流编辑器中将工作流的元素绑定在一起。绑定用于定义这些操作元素对输入参数的处理方式和输出参数的生成方式。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 在**架构**选项卡上，单击 `startVM` 操作元素的**编辑**图标 (✏)。

2 在**信息**选项卡上设置以下常规信息。

选项	操作
交互	选择**无外部交互**。
业务状态	选中该复选框并添加文本**发送启动虚拟机**。
说明	留下文本“启动/恢复虚拟机”。返回启动任务。

3 单击**输入**选项卡。

输入选项卡会显示 `startVM` 操作可用的两个输入参数 `vm` 和 `host`。

Orchestrator 会自动将 `vm` 参数与 `vm[in-parameter]` 绑定，因为 `startVM` 操作只能取 `VC:VirtualMachine` 作为输入参数。Orchestrator 会检测到您在设置工作流输入参数时所定义的 `vm` 参数，因此会自动将其与操作绑定。

4 将 `host` 设置为**空**。

这是一个可选参数，因此可设置为空。但如果将其保留为**未设置**，则工作流无法验证。

5 单击**输出**选项卡。

系统会显示所有操作可生成的默认输出参数 `actionResult`。

6 对于 `actionResult` 参数，单击**未设置**。

7 单击**在工作流中创建参数/属性**。

“参数信息”对话框会显示您可对此输出参数所设置的值。`startVM` 操作的输出参数类型是 `VC:Task` 对象。

8 将该参数命名为 **powerOnTask** 并对其提供说明。

例如，**包含启动虚拟机后的结果**。

9 单击**创建同名的工作流属性**并单击**确定**退出“参数信息”对话框。

10 重复上述步骤将输入和输出参数分别与 `vim3WaitTaskEnd` 和 `vim3WaitToolsStarted` 操作元素绑定。

简单工作流示例操作元素绑定 会列出 `vim3WaitTaskEnd` 和 `vim3WaitToolsStarted` 操作元素的绑定信息。

11 单击工作流编辑器**架构**选项卡底部的**保存**。

操作元素的输入和输出参数即绑定了合适的参数类型和值。

后续步骤

绑定可编辑脚本任务元素并定义其函数。

简单工作流示例操作元素绑定

绑定用于定义简单工作流示例的操作元素对输入和输出参数的处理方式。

定义绑定时，Orchestrator 会显示您在工作流中已定义的参数，并将其作为绑定的候选内容。如果尚未在工作流中定义所需的参数，则参数选择只能为 NULL。单击**在工作流中创建参数/属性**以创建新参数。

vim3WaitTaskEnd 操作

vim3WaitTaskEnd 操作元素会声明常量以跟踪任务进度和轮询率。下表中显示 vim3WaitTaskEnd 操作所需的输入和输出参数绑定。

表 1-53. 绑定 vim3WaitTaskEnd 操作的值

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
task	输入	绑定	■ 本地参数：powerOnTask ■ 源参数：task[attribute] ■ 类型：VC:Task ■ 说明：**包含启动虚拟机后的结果。**
progress	输入	创建	■ 本地参数：progress ■ 源参数：progress[attribute] ■ 类型：布尔 ■ 值：No (false) ■ 说明：**等待 vCenter Server 任务完成时记录进度。**
pollRate	输入	创建	■ 本地参数：pollRate ■ 源参数：pollRate[attribute] ■ 类型：数字 ■ 值：**2** ■ 说明：**以秒为单位的轮询率，vim3WaitTaskEnd 会按此检查 vCenter Server 任务的进度。**
actionResult	输出	创建	■ 本地参数：actionResult[attribute] ■ 源参数：returnedManagedObject[attribute] ■ 类型：任何 ■ 说明：**waitTaskEnd 操作返回的受管对象。**

vim3WaitToolsStarted 操作

vim3WaitToolsStarted 操作元素会等待在虚拟机上安装 VMware Tools，然后定义轮询率和超时时间段。下表中显示 vim3WaitToolsStarted 操作所需的输入参数绑定。

vim3WaitToolsStarted 操作元素无输出，因此无需输出绑定。

表 1-54. 绑定 vim3WaitToolsStarted 操作的值

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
vm	输入	自动绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 值：不可编辑，变量不是工作流属性。 ■ 说明：**要启动的虚拟机。**
pollingRate	输入	绑定	■ 本地参数：pollRate ■ 源参数：pollRate[attribute] ■ 类型：数字 ■ 说明：**以秒为单位的轮询率，vim3WaitTaskEnd 会按此检查 vCenter Server 任务的进度。**
timeout	输入	创建	■ 本地参数：timeout ■ 源参数：timeout[attribute] ■ 类型：数字 ■ 值：**10** ■ 说明：**vim3WaitToolsStarted 等待的超时限制，超出此值即会引发异常。**

绑定简单工作流示例的脚本任务元素

您可在工作流编辑器的**架构**选项卡中将工作流的元素绑定在一起。绑定用于定义这些脚本任务元素对输入参数的处理方式和输出参数的生成方式。您还可以将可编辑脚本任务元素与其 JavaScript 函数进行绑定。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 在**架构**选项卡上，单击 Already Started 可编辑脚本任务元素的**编辑**图标 ( )。

2 在**信息**选项卡上设置以下常规信息。

选项	操作
交互	选择**无外部交互**。
业务状态	选中该复选框并添加文本**虚拟机已打开电源**。
说明	留下文本“虚拟机已打开电源”，跳过 startVM 和 waitTaskEnd，检查虚拟机工具是否已启动并正在运行。

3 单击**输入**选项卡。

由于是自定义的可编辑脚本任务元素，因此未预定义任何属性。

4 单击**绑定到工作流参数/属性**图标 ( )。

5 从建议的参数列表中选择 `vm`。

6 将**输出**和**异常**两个选项卡留空。

该元素不会生成任何输出参数或异常。

7 单击**脚本**选项卡。

8 添加以下 JavaScript 函数。

```
//Writes the following event in the Orchestrator database
Server.log("VM '"+ vm.name +"' already started");
```

9 重复上述步骤将剩余的输入参数分别与其他可编辑脚本任务元素绑定。

简单工作流示例可编辑脚本任务元素绑定 列出了 `Start VM failed`（含 `Timeout` 或 `Error`）、`Send Email Failed` 和 `OK` 等可编辑脚本任务元素的绑定情况。

10 单击工作流编辑器**架构**选项卡底部的**保存**。

您现已将可编辑脚本任务元素与其输入/输出参数进行了绑定，并提供了用于定义其函数的脚本。

后续步骤

您必须定义异常行为处理。

简单工作流示例可编辑脚本任务元素绑定

绑定用于定义简单工作流示例的可编辑脚本任务元素对输入参数的处理方式。您还可以将可编辑脚本任务元素与其 JavaScript 函数进行绑定。

定义绑定时，Orchestrator 会显示您在工作流中已定义的参数，并将其作为绑定的候选内容。如果尚未在工作流中定义所需的参数，则参数选择只能为 `NULL`。单击**在工作流中创建参数/属性**以创建新参数。

“启动虚拟机失败”可编辑脚本任务

“启动虚拟机失败”可编辑脚本任务元素通过设置有关启动虚拟机失败的电子邮件通知内容来处理 `startVM` 操作引发的任何异常，并将事件写入到 Orchestrator 日志中。

下表中显示了“启动虚拟机失败”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-55. “启动虚拟机失败”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要打开电源的虚拟机。**
errorCode	输入	创建	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 说明：**在打开虚拟机电源时发现任何异常。**
body	输出	创建	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 说明：**电子邮件正文**

“启动虚拟机失败”可编辑脚本任务元素会执行以下脚本函数。

```
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);
```

“超时 1”可编辑脚本任务元素

“超时 1”可编辑脚本任务元素通过设置有关任务失败的电子邮件通知内容来处理 vim3WaitTaskEnd 操作引发的任何异常，并将事件写入到 Orchestrator 日志中。

下表中显示“超时 1”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-56. “超时 1”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要启动的虚拟机。**
errorCode	输入	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 说明：**在打开虚拟机电源时发现任何异常。**
body	输出	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 说明：**电子邮件正文**

“超时 1”可编辑脚本任务元素需要以下脚本函数。

```
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);
```

“超时 2”可编辑脚本任务元素

“超时 2”可编辑脚本任务元素通过设置有关任务失败的电子邮件通知内容来处理 vim3WaitToolsStarted 操作引发的任何异常，并将事件写入到 Orchestrator 日志中。

下表中显示“超时 2”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-57. “超时 2”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要打开电源的虚拟机。**
errorCode	输入	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 说明：**在打开虚拟机电源时发现任何异常。**
body	输出	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 说明：**电子邮件正文**

“超时 2”可编辑脚本任务元素需要以下脚本函数。

```
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);
```

“确定”可编辑脚本任务元素

“确定”可编辑脚本任务元素会接收虚拟机已启动成功的消息、设置有关虚拟机成功启动的电子邮件通知内容以及将事件写入到 Orchestrator 日志中。

下表中显示“确定”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-58. “确定”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要打开电源的虚拟机。**
body	输出	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 说明：**电子邮件正文**

“确定”可编辑脚本任务元素需要以下脚本函数。

```
body = "The VM '"+vm.name+"' has started successfully and is ready for use";
//Writes the following event in the Orchestrator database
Server.log(body);
```

“发送电子邮件失败”可编辑脚本任务元素

“发送电子邮件失败”可编辑脚本任务元素会接收电子邮件发送失败的通知，并将事件写入到 Orchestrator 日志中。

下表中显示了“发送电子邮件失败”可编辑脚本任务元素所需的输入参数绑定。

表 1-59. “发送电子邮件失败”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要打开电源的虚拟机。**
toAddress	输入	绑定	■ 本地参数：toAddress ■ 源参数：toAddress[in-parameter] ■ 类型：字符串 ■ 说明：**要接收该工作流结果通知的用户电子邮件地址**
emailErrorCode	输入	创建	■ 本地参数：emailErrorCode ■ 源参数：emailErrorCode[attrbute] ■ 类型：字符串 ■ 说明：**在发送电子邮件时发现任何异常**

“发送电子邮件失败”可编辑脚本任务元素需要以下脚本函数。

```
//Writes  the following event in the Orchestrator database
Server.error("Couldn't send result email to '"+toAddress+"' for VM '"+vm.name+"', exception found:
"+emailErrorCode);
```

“发送电子邮件”可编辑脚本任务元素

“启动虚拟机并发送电子邮件”工作流的目的是在启动虚拟机时通知管理员。为此，您必须定义用于发送电子邮件的可编辑脚本任务。若要发送电子邮件，“发送电子邮件”可编辑脚本任务元素需要 SMTP 服务器、发件人地址和收件人地址、邮件主题和邮件内容。

下表中显示“发送电子邮件”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-60. “发送电子邮件”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[in-parameter] ■ 类型：VC:VirtualMachine ■ 说明：**要打开电源的虚拟机。**
toAddress	输入	绑定	■ 本地参数：toAddress ■ 源参数：toAddress[in-parameter] ■ 类型：字符串 ■ 说明：**要接收该工作流结果通知的用户电子邮件地址**
body	输入	绑定	■ 本地参数：body ■ 源参数：body[attribute] ■ 类型：字符串 ■ 说明：**电子邮件正文**
smtpHost	输入	创建	■ 本地参数：smtpHost ■ 源参数：smtpHost[attribute] ■ 类型：字符串 ■ 说明：**电子邮件 SMTP 服务器**
fromAddress	输入	创建	■ 本地参数：fromAddress ■ 源参数：fromAddress[attribute] ■ 类型：字符串 ■ 说明：**发件人的电子邮件地址**
subject	输入	创建	■ 本地参数：subject ■ 源参数：subject[attribute] ■ 类型：字符串 ■ 说明：**电子邮件主题**

“发送电子邮件”可编辑脚本任务元素需要以下脚本函数。

```
//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+"'");
```

定义简单工作流示例异常绑定

您可以在工作流编辑器中的**架构**选项卡中定义异常绑定。异常绑定用于定义元素对错误的处理方式。

工作流中会返回异常的元素包括：`startVM`、`vim3WaitTaskEnd`、`Send Email` 和 `vim3WaitToolsStarted`。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 绑定简单工作流示例的操作元素。
- 绑定简单工作流示例的脚本任务元素。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 在**架构**选项卡上，单击 **startVM** 操作元素的**编辑**图标 (✏)。

2 单击**异常**选项卡。

3 单击**未设置**按钮。

4 从建议列表中选择 **errorCode**。

5 重复上述步骤，为 `vim3WaitTaskEnd` 和 `vim3WaitToolsStarted` 设置 `errorCode` 异常绑定。

6 单击 **Send Email** 可编辑脚本任务元素中的**编辑**图标 (✏)。

7 单击**异常**选项卡。

8 单击**未设置**按钮。

9 从建议列表中选择 **emailErrorCode**。

10 单击工作流编辑器**架构**选项卡底部的**保存**。

您即定义了会返回异常的元素的异常绑定。

后续步骤

您必须设置属性和参数的读取和写入特性。

设置简单工作流示例属性的读写特性

您可以将参数和属性定义为只读常量或是可写变量。您还可以限制用户可以作为输入参数所提供的值。

将某些参数设置为只读能让其他开发人员调整或修改工作流而不会破坏工作流的核心函数。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 绑定简单工作流示例的操作元素。
- 绑定简单工作流示例的脚本任务元素。
- 定义简单工作流示例异常绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器顶部的**常规**选项卡。

在**属性**下列出了所有已定义的属性，每个属性旁都有一个复选框。选中这些复选框即把相应属性设置为只读。

2 选中复选框将以下参数设置为只读常量：

- progress
- pollRate
- timeout
- smtpHost
- fromAddress
- subject

您即定义了工作流的哪些属性为常量，哪些为变量。

后续步骤

设置参数属性并对该参数的可能值进行限制。

设置简单工作流示例参数属性

您可以在工作流编辑器中设置参数属性。设置参数属性会影响参数的行为，并会对该参数的可能值产生限制。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。

- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 绑定简单工作流示例的操作元素。
- 绑定简单工作流示例的脚本任务元素。
- 定义简单工作流示例异常绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**展示**选项卡。

此处会列出您为该工作流定义的两个输入参数。

2 单击 **(VC:VirtualMachine)vm** 参数。

3 在屏幕下半部分的**常规**选项卡中添加说明。

例如：输入**要启动的虚拟机**。

4 单击屏幕下半部分中的**属性**选项卡。

您可在此选项卡上为 (VC:VirtualMachine)vm 参数设置属性。

5 单击**添加属性**图标 ( )。

6 在建议属性的列表中，选择**必需输入**属性，然后单击**确定**，并将其值设置为**是**。

启用该属性时，用户如果未提供要启动的虚拟机，会无法运行“启动虚拟机并发送电子邮件”工作流。

7 单击**添加属性**图标 ( )。

8 在建议属性的列表中，选择**将值选择为**，然后单击**确定**，并从可能值的列表中选择**列表**。

设置该属性后，您即设置了用户选择 (VC:VirtualMachine)vm 输入参数值的方式。

9 单击**展示**选项卡上半部分中的 **(string)toAddress** 参数。

10 在屏幕下半部分的**说明**选项卡中添加说明。

例如：输入**要通知的用户的电子邮件地址**。

11 单击 (string)toAddress 的**属性**选项卡，然后单击**添加属性**图标 ( )。

12 在建议属性的列表中，选择**必需输入**属性，然后单击**确定**，并将其值设置为**是**。

13 单击**添加属性**图标 ( )。

14 在建议属性的列表中，选择**匹配正则表达式**，然后单击**确定**。

此属性可让您限制用户可以提供为输入参数的内容。

15 单击**匹配正则表达式**的**值**文本框，然后将限制设置为 `[a-zA-Z0-9_%-+.]+@[a-zA-Z0-9-.]+\.[a-zA-Z]{2,4}`。

设置这些限制可将用户输入限制为适用于电子邮件地址的字符。如果用户在启动工作流时尝试为收件人电子邮件地址输入任何其他字符，则该工作流不会启动。

您已将两个参数都设置为必需参数，定义了用户对要启动的虚拟机的选择方式，并限制了收件人电子邮件地址可输入的字符。

后续步骤

您必须为输入参数对话框创建布局或展示，用户可在其中指定运行工作流时的工作流输入参数值。

设置简单工作流示例输入参数对话框的布局

在工作流编辑器中创建输入参数对话框的布局或展示。用户在运行需要输入参数才能运行的工作流时，输入参数对话框即会打开。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 绑定简单工作流示例的操作元素。
- 绑定简单工作流示例的脚本任务元素。
- 定义简单工作流示例异常绑定。
- 设置简单工作流示例属性的读写特性。
- 设置简单工作流示例参数属性。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**展示**选项卡。

2 右键单击展示层次结构列表中的**展示**节点，然后选择**创建显示组**。

展示节点下会显示一个**新步骤**节点和一个**新组**子节点。

3 右键单击**新步骤**并选择**删除**。

由于此工作流只有两个参数，输入参数对话框中的显示部分无需多个层。

4 双击**新组**以编辑组名称，然后按 Enter 键。

例如：将显示组命名为**虚拟机**。

在此输入的文本在用户启动工作流时会显示为输入参数对话框的标题。

5 在**展示**选项卡底部的**常规**选项卡的**说明**文本框中，输入新显示组的说明。

例如：输入**选择要启动的虚拟机**。

在此输入的文本在用户启动工作流时会显示为输入参数对话框的提示。

6 将 **(VC:VirtualMachine)vm** 参数拖放到**虚拟机**显示组下。

在输入参数对话框中的虚拟机标题下会显示一个文本框，用于可在其中输入虚拟机名称。

7 重复上述步骤为 `toAddress` 参数创建显示组，并设置以下属性：

a 创建显示组并将其命名为**收件人的电子邮件**。

b 添加显示组的说明，例如：**输入此虚拟机打开电源时要通知的用户的电子邮件地址**。

c 将 **toAddress** 参数拖放到**收件人电子邮件地址**显示组下。

您即设置了用户运行工作流时显示的输入参数对话框的布局。

后续步骤

您已完成简单工作流示例的开发。现在可以验证和运行工作流了。

验证并运行简单工作流示例

创建工作流后，可以对其进行验证以发现任何可能的错误。如果不包含任何错误，则可以运行该工作流。

前提条件

完成以下任务。

- 创建简单工作流示例。
- 创建简单工作流示例的架构。
- 定义简单工作流示例的参数。
- 定义简单工作流示例决策绑定。
- 绑定简单工作流示例的操作元素。
- 绑定简单工作流示例的脚本任务元素。
- 定义简单工作流示例异常绑定。
- 设置简单工作流示例属性的读写特性。
- 设置简单工作流示例参数属性。
- 设置简单工作流示例输入参数对话框的布局。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器的**架构**选项卡中的**验证**。

验证工具会找到工作流定义中的任何错误。

2 消除所有错误后，单击工作流编辑器底部的**保存并关闭**。

此时系统会返回 Orchestrator 客户端。

3 单击**工作流**视图。

4 在工作流层次结构列表中，选择**工作流示例 > 启动虚拟机并发送电子邮件**。

5 右键单击**启动虚拟机并发送电子邮件**工作流，然后选择**启动工作流**。

此时系统会打开输入参数对话框，提示您输入要启动的虚拟机以及用来接收通知的电子邮件地址。

6 从 vCenter Server 清单中选择要启动的虚拟机。

7 输入要接收电子邮件通知的电子邮件地址。

8 单击**提交**以启动工作流。

此时在“启动虚拟机并发送电子邮件”工作流下会显示一个工作流令牌。

9 单击该工作流令牌以跟踪工作流的运行进度。

如果工作流运行成功，则您选择的虚拟机会处于电源打开状态，并且您定义的电子邮件收件人会收到确认电子邮件。

后续步骤

您可以生成一个文档以在其中查看有关工作流的信息。请参见生成工作流文档。

开发复杂工作流

开发复杂示例工作流可以展示工作流开发过程中最常见的步骤和更高级的场景，例如创建自定义决策和循环。

在复杂工作流开发练习中，您要开发一个可对给定资源池中所有虚拟机创建快照的工作流。创建的工作流应执行以下任务：

1 提示用户输入要对其所包含的虚拟机创建快照的资源池。

2 确定资源池是否包含正在运行的虚拟机。

3 确定资源池所包含的正在运行的虚拟机数量。

4 验证资源池中运行的单个虚拟机是否符合创建快照的具体条件。

5 创建虚拟机快照。

6 确定资源池中是否存在更多要对其创建快照的虚拟机。

7 重复验证和创建快照流程，直到工作流为资源池中所有符合条件的虚拟机创建快照为止。

从 Orchestrator 文档登录页面下载的 Orchestrator 示例 ZIP 文件包含完整版的“创建资源池中所有虚拟机的快照”工作流。

前提条件

在尝试开发此复杂工作流之前，请遵循开发简单示例工作流中的练习做法。以下开发复杂工作流的步骤大致介绍了开发过程，但详细程度不及开发简单工作流中的练习做法。

步骤

1 创建复杂工作流示例

您必须在 Orchestrator 客户端中创建工作流来开始工作流开发过程。

2 为复杂工作流示例创建自定义操作

`Check VM` 可编辑脚本元素会调用 Orchestrator API 上不存在的操作。您必须创建 `getVMDiskModes` 操作。

3 创建复杂工作流示例的架构

您可以在工作流编辑器中创建工作流的架构。工作流架构包含工作流运行的元素，并确定工作流的逻辑流。

4 （可选） 创建复杂工作流示例区域

（可选）您可以通过添加工作流备注的方式来突出显示工作流的不同区域。创建不同的工作流区域有助于使复杂工作流架构更易于阅读和理解。

5 定义复杂工作流示例的参数

您可在工作流编辑器中定义工作流参数。输入参数提供需要工作流处理的数据。输出参数是工作流运行完成后返回的数据。

6 定义复杂工作流示例的绑定

您可在工作流编辑器中将工作流的元素绑定在一起。绑定将定义工作流的数据流。您还可以将可编辑脚本任务元素与其 JavaScript 函数进行绑定。

7 设置复杂工作流示例属性的特性

您可以在工作流编辑器的**常规**选项卡中设置属性的特性。

8 创建复制工作流示例输入参数的布局

在工作流编辑器的**展示**选项卡中创建输入参数对话框的布局或展示。用户在运行工作流时，输入参数对话框会打开，用户可在此输入工作流运行时使用的输入参数。

9 验证并运行复杂工作流示例

创建工作流后，可以对其进行验证以检测任何可能的错误。如果不包含任何错误，则可以运行该工作流。

创建复杂工作流示例

您必须在 Orchestrator 客户端中创建工作流来开始工作流开发过程。

有关如何安装和配置 vCenter Server 的信息，请参见《vSphere 安装和设置》文档。有关如何配置 Orchestrator 的信息，请参见《安装和配置 VMware vRealize Orchestrator》。

前提条件

确认系统上已安装并配置了以下组件。

- vCenter Server，控制包含部分虚拟机的资源池
- 工作流层次结构列表中的工作流示例文件夹（即在 创建简单工作流示例 中创建的）。

步骤

1. 从 Orchestrator 客户端的下拉菜单中，选择**设计**。
2. 选择**工作流 > 工作流示例**。
3. 右键单击**工作流示例**文件夹，然后选择**新建工作流**。
4. 将新工作流命名为**创建资源池中所有虚拟机的快照**，然后单击**确定**。

   此时系统会打开工作流编辑器。
5. 在工作流编辑器的**常规**选项卡上，单击版本号数字以增加版本号。

   对于初次创建的工作流，请将版本设置为 **0.0.1**。
6. 单击**服务器重新启动行为**值以设置工作流是否在服务器重新启动后恢复。
7. 在**说明**文本框中，输入工作流的功能说明。
8. 单击**常规**选项卡底部的**保存**。

您即创建了“创建虚拟机中所有虚拟机的快照”工作流。

后续步骤

您必须创建自定义操作。

为复杂工作流示例创建自定义操作

`Check VM` 可编辑脚本元素会调用 Orchestrator API 上不存在的操作。您必须创建 `getVMDiskModes` 操作。

有关创建操作的更多详细信息，请参见第 3 章 开发操作。

前提条件

创建“创建资源池中所有虚拟机的快照”工作流。请参见创建复杂工作流示例。

步骤

1. 单击**保存并关闭**以关闭工作流编辑器。
2. 单击 Orchestrator 客户端中的**操作**视图。
3. 右键单击操作层次结构列表的根，然后选择**新建模块**。
4. 将新模块命名为 **`com.vmware.example`**。
5. 右键单击 **com.vmware.example** 模块，然后选择**添加操作**。
6. 创建名为 `getVMDiskModes` 的操作。

7 在操作编辑器的**常规**选项卡中单击版本数字来增加版本号。

8 在**常规**选项卡中添加以下操作说明。

```
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 单击**脚本**选项卡。

10 右键单击**脚本**选项卡的顶部窗格，然后选择**添加参数**来创建以下输入参数。

- 名称：`vm`
- 类型：`VC:VirtualMachine`
- 说明：**要返回磁盘模式的虚拟机**

11 在**脚本**选项卡的底部添加以下脚本。

以下代码会返回虚拟机磁盘的磁盘模式数组。

```
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 单击**保存并关闭**以退出**操作**调色板。

您即定义了“创建资源池中所有虚拟机的快照”工作流所需的自定义操作。

后续步骤

创建工作流的架构。

创建复杂工作流示例的架构

您可以在工作流编辑器中创建工作流的架构。工作流架构包含工作流运行的元素，并确定工作流的逻辑流。

前提条件

完成以下任务。

- 创建复杂工作流示例。
- 为复杂工作流示例创建自定义操作。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 将以下架构元素添加到工作流架构中。

元素类型	元素名称	在架构中的位置
可编辑脚本任务	**Initializing**	在 Start 元素下方
决策	**VMs to Process?**	在 Initializing 可编辑脚本任务元素下方
可编辑脚本任务	**Pool Has No VMs**	在 VMs to Process? 自定义决策元素下方，使用红色箭头链接
自定义决策	**Remaining VMs?**	在 VMs to Process? 自定义决策元素右侧，使用绿色箭头链接
操作	getVMDiskModes	在 Remaining VMs? 自定义决策元素右侧，使用绿色箭头链接
自定义决策	**Create Snapshot?**	在 getVMDiskModes 操作元素右侧，使用蓝色箭头链接
工作流	Create a snapshot	在 Create Snapshot? 自定义决策元素上方，使用绿色箭头链接
可编辑脚本任务	**VM Snapshots**	在 Create a snapshot 工作流左侧，使用蓝色箭头链接
可编辑脚本任务	**Increment**	在 VM Snapshots 可编辑脚本任务元素左侧，使用蓝色箭头链接
可编辑脚本任务	**Set Output**	在 Pool Has No VMs 可编辑脚本任务右侧，使用蓝色箭头链接

3 添加 Log Exception 可编辑脚本任务元素。

   a 在“创建快照”工作流和 End 元素之间创建一个异常处理链接。

   b 将可编辑脚本任务元素拖放到红色虚线箭头上，该箭头会将“创建快照”工作流链接到 End 元素。

   c 双击可编辑脚本任务，并将其重命名为**记录异常**。

   d 将 Log Exception 可编辑脚本任务元素移动到 VM Snapshots 可编辑脚本任务元素的上方。

4 取消链接所有 End 元素，但位于 Set Output 可编辑脚本任务元素右侧的 End 元素除外。

5 按下表所述链接剩余元素。

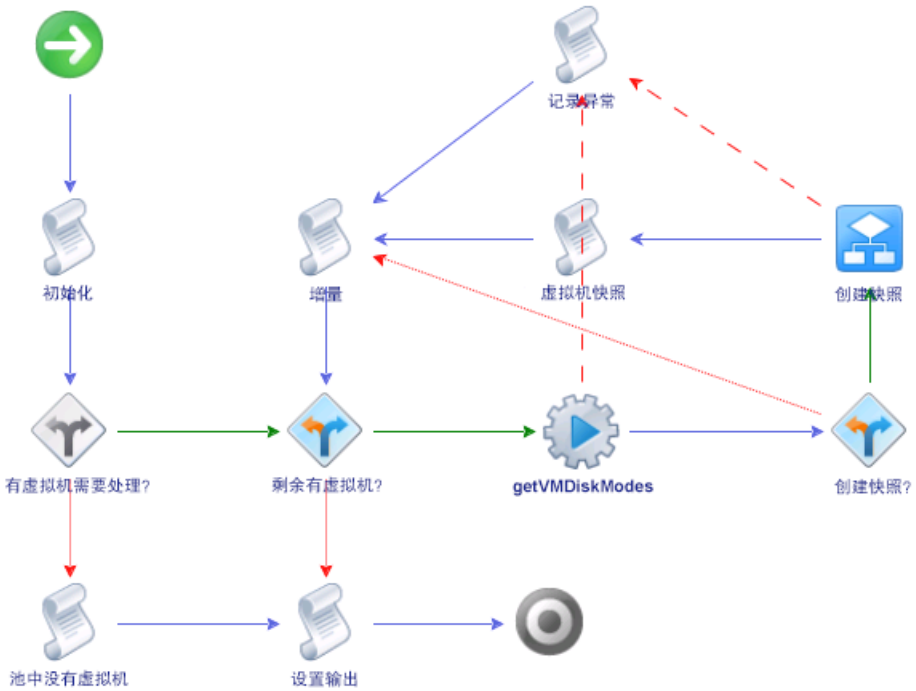
元素	链接到	箭头类型	说明
getVMDiskModes 操作元素	Log Exception 可编辑脚本任务元素	红色虚线	异常处理
Create Snapshot? 自定义决策元素	Increment 可编辑脚本任务元素	红色	False 结果

元素	链接到	箭头类型	说明
Log Exception 可编辑脚本任务元素	Increment 可编辑脚本任务元素	蓝色	正常工作流进度
Increment 可编辑脚本任务元素	Remaining VMs? 自定义决策元素	蓝色	正常工作流进度
Remaining VMs? 自定义决策元素	Set Output 可编辑脚本任务元素	红色	False 结果

6 单击**架构**选项卡底部的**保存**。

下图显示了“创建资源池中所有虚拟机的快照”工作流中各元素之间的链接形式。

图 1-12. “创建资源池中所有虚拟机的快照”示例工作流中的链接



后续步骤

（可选）您可以使用工作流备注来定义工作流区域。

创建复杂工作流示例区域

（可选）您可以通过添加工作流备注的方式来突出显示工作流的不同区域。创建不同的工作流区域有助于使复杂工作流架构更易于阅读和理解。

前提条件

完成以下任务。

- 创建复杂工作流示例。
- 创建复杂工作流示例的架构。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 使用工作流备注创建以下工作流区域。

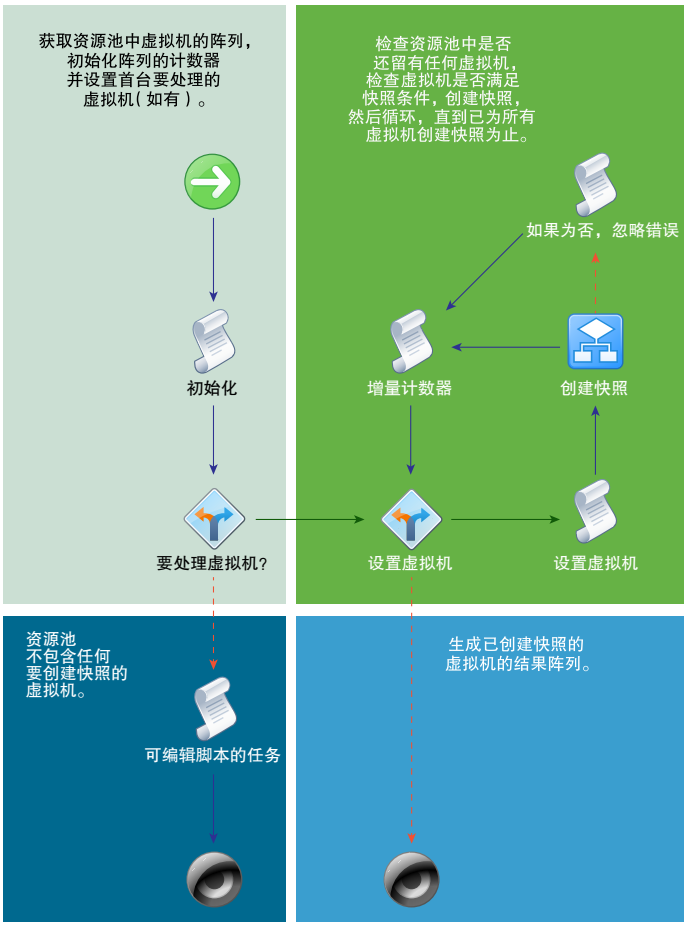
区域中的元素	说明
“起始”元素；“初始化”可编辑脚本任务；“要处理虚拟机?”自定义决策	获取资源池中的虚拟机阵列，初始化阵列计数器并设置首台要处理的虚拟机（如有）。
“池中没有虚拟机”可编辑脚本任务。	资源池不包含任何要创建快照的虚拟机。
“剩余有虚拟机?”自定义决策；`getVMDisksModes` 操作；“创建快照?”决策；“创建快照”工作流；“虚拟机快照”可编辑脚本任务；“增量”可编辑脚本任务；“记录异常”可编辑脚本任务	检查资源池中是否还留有任何虚拟机，检查虚拟机是否满足快照条件，创建快照，然后循环，直到为所有虚拟机创建快照为止。
“设置输出”可编辑脚本任务；“结束”元素	生成已创建快照的虚拟机的结果阵列。

2 选择工作流备注并按 Ctrl+E 选择背景颜色。

3 单击工作流编辑器**架构**选项卡底部的**保存**。

工作流区域的表现形式如下图所示。

图 1-13. “创建资源池中所有虚拟机的快照”示例工作流的架构图



后续步骤

您必须定义工作流的输入和输出参数。

定义复杂工作流示例的参数

您可在工作流编辑器中定义工作流参数。输入参数提供需要工作流处理的数据。输出参数是工作流运行完成后返回的数据。

前提条件

完成以下任务。

- 创建复杂工作流示例。
- 创建复杂工作流示例的架构。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**输入**选项卡。

2 定义以下输入参数。

   - 名称：`resourcePool`
   - 类型：`VC:ResourcePool`
   - 说明：**资源池包含要对其创建快照的虚拟机。**

3 单击工作流编辑器中的**输出**选项卡。

4 定义以下输出参数。

   - 名称：`snapshotVmArrayOut`
   - 类型：`Array/VC:VirtualMachine`
   - 说明：**已对其创建了快照的虚拟机阵列。**

您即定义了工作流的输入和输出参数。

后续步骤

您必须定义元素参数之间的绑定。

定义复杂工作流示例的绑定

您可在工作流编辑器中将工作流的元素绑定在一起。绑定将定义工作流的数据流。您还可以将可编辑脚本任务元素与其 JavaScript 函数进行绑定。

前提条件

完成以下任务。

- 创建复杂工作流示例.

- 创建复杂工作流示例的架构
- 定义复杂工作流示例的参数
- 检查您必须定义的绑定。请参见复杂工作流示例绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**架构**选项卡。

2 定义绑定。

3 单击**架构**选项卡底部的**保存**。

元素的所有输入和输出参数即绑定了合适的参数类型和值。

后续步骤

设置属性的特性。

复杂工作流示例绑定

绑定用于定义简单工作流示例的操作元素对输入和输出参数的处理方式。

“创建资源池中所有虚拟机的快照”工作流需要使用以下输入和输出参数绑定。您还可以定义可编辑脚本任务元素的 JavaScript 函数。

如果与现有参数绑定，则绑定后会继承原始参数的类型和说明值。

“初始化”可编辑脚本任务

“初始化”可编辑脚本任务元素会初始化该工作流的各种属性。下表中显示了“初始化”可编辑脚本任务元素所需的输入和输出参数绑定。

表 1-61. “初始化”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数？	绑定值
`resourcePool`	输入	绑定	■ 本地参数：`resourcePool` ■ 源参数：`resourcePool[in-parameter]` ■ 类型：`VC:ResourcePool` ■ 说明：**资源池包含要对其创建快照的虚拟机**
`allVMs`	输出	创建	■ 本地参数：`allVMs` ■ 源参数：`allVMs[attribute]` ■ 类型：`Array/VC:VirtualMachine` ■ 说明：**资源池中的虚拟机。**
`numberOfVMs`	输出	创建	■ 本地参数：`numberOfVMs` ■ 源参数：`numberOfVMs[attribute]` ■ 类型：数字 ■ 说明：**资源池中找到的虚拟机数量**

表 1-61. “初始化”可编辑脚本任务元素的绑定 （续）

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vmCounter	输出	创建	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 说明：**阵列中虚拟机的计数器**
vm	输出	创建	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**当前已创建了快照的虚拟机**
snapshotVmArray	输出	创建	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 说明：**已对其创建了快照的虚拟机阵列**

“初始化”可编辑脚本任务元素会执行以下脚本函数。

```
//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();
```

“要处理虚拟机?”决策元素

“要处理虚拟机?”决策元素用于确定资源池中是否存在任何要对其创建快照的虚拟机。下表中显示了“要处理虚拟机?”决策元素所需的绑定。

表 1-62. “要处理虚拟机?”决策元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
numberOfVMs	决策	绑定	■ 源参数：numberOfVMs[attribute] ■ 决策声明：大于 ■ 值：0.0 ■ 说明：**资源池中找到的虚拟机数量**

“池中没有虚拟机”可编辑脚本任务元素

“池中没有虚拟机”可编辑脚本任务元素会记录资源池中不包含可合格用于 Orchestrator 数据库的虚拟机这一事实。下表中显示了“池中没有虚拟机”可编辑脚本任务元素所需的绑定。

表 1-63. “池中没有虚拟机”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
resourcePool	输入	绑定	■ 本地参数：resourcePool ■ 源参数：resourcePool[in-parameter] ■ 类型：VC:ResourcePool ■ 说明：**资源池包含要对其创建快照的虚拟机。**

“池中没有虚拟机”可编辑脚本任务元素会执行以下脚本函数。

```
//Writes the following event in the Orchestrator database
Server.warn("The specified ResourcePool "+resourcePool.name+" does not contain any VMs.");
```

“剩余有虚拟机?”自定义决策元素

“剩余有虚拟机?”自定义决策元素用于确定资源池中是否剩余任何要创建快照的虚拟机。下表中显示了“剩余有虚拟机?”自定义决策元素所需的绑定。

表 1-64. “剩余有虚拟机?”自定义决策元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
numberOfVMs	输入	绑定	■ 源参数：numberOfVMs[attribute] ■ 决策声明：大于 ■ 值：0.0 ■ 说明：**资源池中找到的虚拟机数量**
vmCounter	输入	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 说明：**阵列中虚拟机的计数器**

“剩余有虚拟机?”自定义决策元素会执行以下脚本函数。

```
//Checks if the workflow has reached the end of the array of VMs
if (vmCounter < numberOfVms) {
    return true;
} else {
    return false;
}
```

getVMDisksModes 操作元素

getVMDisksModes 操作元素用于获取正在虚拟机中运行的磁盘的模式。下表中显示了 getVMDisksModes 操作元素所需的绑定。

表 1-65. getVMDisksModes 操作元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**当前已创建了快照的虚拟机**
actionResult	输出	创建	■ 本地参数：actionResult ■ 源参数：vmDisksModes[attribute] ■ 类型：阵列/字符串 ■ 说明：**虚拟机的当前磁盘模式**
errorCode	异常	创建	本地参数：errorCode

“创建快照?”自定义决策元素

“创建快照?”自定义决策元素用于根据虚拟机的磁盘模式来确定是否对虚拟机创建快照。下表中显示了“创建快照?”自定义决策元素所需的绑定。

表 1-66. “创建快照?”决策元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vmDisksMode	输入	绑定	■ 本地参数：vmDisksMode ■ 源参数：vmDisksMode[attribute] ■ 类型：阵列/字符串 ■ 说明：**虚拟机的当前磁盘模式**
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**当前已创建了快照的虚拟机**

“创建快照?”自定义决策元素会执行以下脚本函数。

```
//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("Snapshoting '"+vm.name+"'");
    return true;
}
```

“创建快照”工作流元素

“创建快照”工作流元素用于创建虚拟机快照。下表中显示了“创建快照”工作流元素所需的绑定。

表 1-67. “创建快照”工作流元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**要对其创建快照的活动虚拟机。**
name	输入	创建	■ 本地参数：name ■ 源参数：snapshotName[attribute] ■ 类型：字符串 ■ 说明：**此快照的名称。此名称无需对该虚拟机唯一。**
description	输入	创建	■ 本地参数：description ■ 源参数：snapshotDescription[attribute] ■ 类型：字符串 ■ 说明：**此快照的说明。**
memory	输入	创建	■ 本地参数：memory ■ 源参数：snapshotMemory[attribute] ■ 类型：布尔 ■ 值：no ■ 说明：**如果为 TRUE，则快照中会包含该虚拟机内部状态的转储（内存转储）。**

表 1-67. “创建快照”工作流元素的绑定 （续）

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
quiesce	输入	创建	■ 本地参数：quiesce ■ 源参数：snapshotQuiesce[attribute] ■ 类型：布尔 ■ 值：yes ■ 说明：**如果为 TRUE 且该虚拟机在创建快照时处于启动状态，则 VMware Tools 将用于静默虚拟机中的文件系统。**
snapshot	输出	创建	■ 本地参数：snapshot ■ 源参数：NULL ■ 类型：VC:VirtualMachineSnapshot ■ 说明：**创建的快照。**
errorCode	异常	创建	本地参数：errorCode

“虚拟机快照”可编辑脚本任务元素

“虚拟机快照”可编辑脚本任务元素用于向阵列添加快照。下表中显示了“虚拟机快照”可编辑脚本任务元素所需的绑定。

表 1-68. “虚拟机快照”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**要对其创建快照的活动虚拟机。**
snapshotVmArray	输入	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 说明：**已对其创建了快照的虚拟机阵列**
snapshotVmArray	输出	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 说明：**已对其创建了快照的虚拟机阵列**

“虚拟机快照”可编辑脚本任务元素会执行以下脚本函数。

```
//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);
```

“增量”可编辑脚本任务元素

“增量”可编辑脚本任务元素用于对统计阵列中虚拟机数目的计数器进行递增操作。下表中显示了“增量”可编辑脚本任务元素所需的绑定。

表 1-69. “增量”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vmCounter	输入	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 说明：**阵列中虚拟机的计数器**
allVMs	输入	绑定	■ 本地参数：allVMs ■ 源参数：allVMs[attribute] ■ 类型：Array/VC:VirtualMachine ■ 说明：**资源池中的虚拟机。**
vmCounter	输出	绑定	■ 本地参数：vmCounter ■ 源参数：vmCounter[attribute] ■ 类型：数字 ■ 说明：**阵列中虚拟机的计数器**
vm	输出	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**当前已创建了快照的虚拟机**

“增量”可编辑脚本任务元素会执行以下脚本函数。

```
//Increases the array VM counter
vmCounter++;
//Sets the next VM to be snapshot in the attribute vm
vm = allVMs[vmCounter];
```

“记录异常”可编辑脚本任务元素

“记录异常”可编辑脚本任务元素用于处理工作流和操作元素中的异常行为。下表中显示了“记录异常”可编辑脚本任务元素所需的绑定。

表 1-70. “记录异常”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
vm	输入	绑定	■ 本地参数：vm ■ 源参数：vm[attribute] ■ 类型：VC:VirtualMachine ■ 说明：**当前已创建了快照的虚拟机**
errorCode	输入	绑定	■ 本地参数：errorCode ■ 源参数：errorCode[attribute] ■ 类型：字符串 ■ 说明：**创建虚拟机快照时发现的异常行为**

“记录异常”可编辑脚本任务元素会执行以下脚本函数。

```
//Writes  the following event in the Orchestrator database
Server.error("Coudln't snapshot the VM '"+vm.name+"', exception: "+errorCode);
```

“设置输出”可编辑脚本任务元素

“设置输出”可编辑脚本任务元素用于生成工作流的输出参数，其中包含已对其创建了快照的虚拟机阵列。下表中显示了“设置输出”可编辑脚本任务元素所需的绑定。

表 1-71. “设置输出”可编辑脚本任务元素的绑定

参数名称	绑定类型	绑定到现有参数或创建参数?	绑定值
snapshotVmArray	输入	绑定	■ 本地参数：snapshotVmArray ■ 源参数：snapshotVmArray[attribute] ■ 类型：Array/VC:VirtualMachine ■ 说明：**已对其创建了快照的虚拟机阵列**
snapshotVmArrayOut	输出	绑定	■ 本地参数：snapshotVmArrayOut ■ 源参数：snapshotVmArrayOut[out-parameter] ■ 类型：Array/VC:VirtualMachine ■ 说明：**已对其创建了快照的虚拟机阵列。**

“设置输出”可编辑脚本任务元素会执行以下脚本函数。

```
//Passes the value of the internal attribute to a workflow output parameter
snapshotVmArrayOut = snapshotVmArray;
```

设置复杂工作流示例属性的特性

您可以在工作流编辑器的**常规**选项卡中设置属性的特性。

前提条件

完成以下任务。

- 创建复杂工作流示例。
- 创建复杂工作流示例的架构。
- 定义复杂工作流示例的绑定。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击**常规**选项卡。

2 选择以下属性的只读复选框将其设置为只读常量：

- snapshotName
- snapshotDescription
- snapshotMemory
- snapshotQuiesce

您即定义了工作流的哪些属性为常量，哪些为变量。

后续步骤

您必须创建工作流展示，其应创建输入参数对话框的布局，用户可在工作流运行时在其中指定输入参数值。

创建复制工作流示例输入参数的布局

在工作流编辑器的**展示**选项卡中创建输入参数对话框的布局或展示。用户在运行工作流时，输入参数对话框会打开，用户可在此输入工作流运行时使用的输入参数。

前提条件

完成以下任务。

- 创建复杂工作流示例。
- 创建复杂工作流示例的架构。
- 定义复杂工作流示例的参数。
- 定义复杂工作流示例的绑定。
- 设置复杂工作流示例属性的特性。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器中的**展示**选项卡。

“创建资源池中所有虚拟机的快照”工作流只有一个输入参数，因此创建展示很简单。

2 右键单击展示层次结构列表中的**展示**节点，然后选择**创建显示组**。

3 删除**新组**元素上方显示的**新步骤**元素。

4 右键单击**新组**元素，并将组名改为**资源池**。

5 在**展示**选项卡底部的**常规**选项卡的**说明**文本框中，为**资源池**显示组输入相关说明。

例如：**输入包含要对其创建快照的虚拟机的资源池名称。**

6 单击 `(VC:ResourcePool)resourcePool` 参数。

7 单击 `(VC:ResourcePool)resourcePool` 的**属性**选项卡。

8 在**属性**选项卡中单击右键，然后选择**添加属性 > 必需输入**。

9 在**属性**选项卡中单击右键，然后选择**添加属性 > 将值选择为**。

设置该属性后，您即设置了用户选择 `(VC:ResourcePool)resourcePool` 输入参数值的方式。

10 将 `(VC:ResourcePool)resourcePool` 参数拖放到**资源池**显示组下。

您即创建了用户运行工作流时显示的对话框的布局。

后续步骤

您已完成复杂工作流示例的开发。现在可以验证和运行工作流了。

验证并运行复杂工作流示例

创建工作流后，可以对其进行验证以检测任何可能的错误。如果不包含任何错误，则可以运行该工作流。

前提条件

创建工作流、布局其架构、定义链接和绑定、定义参数属性并创建输入参数对话框展示。

完成以下任务。

- 创建复杂工作流示例。
- 为复杂工作流示例创建自定义操作。
- 创建复杂工作流示例的架构。
- 定义复杂工作流示例的参数。
- 定义复杂工作流示例的绑定。
- 设置复杂工作流示例属性的特性。
- 创建复制工作流示例输入参数的布局。
- 在工作流编辑器中打开要编辑的工作流。

步骤

1 单击工作流编辑器的**架构**选项卡中的**验证**。

验证工具会检测工作流定义中的任何错误。

2 消除所有错误后，单击工作流编辑器底部的**保存并关闭**。

此时系统会返回 Orchestrator 客户端。

3 单击**工作流**视图。

4 在工作流层次结构列表中，选择**工作流示例 > 创建资源池中所有虚拟机的快照**。

5 右键单击**创建资源池中所有虚拟机的快照**工作流，然后选择**启动工作流**。

此时系统会打开输入参数对话框，提示您输入包含要对其创建快照的虚拟机的资源池。

6 单击**提交**以运行工作流。

此时会在“创建资源池中所有虚拟机的快照”工作流下显示一个工作流令牌。

7 单击该工作流令牌以跟踪工作流的运行进度。

如果工作流运行成功，则工作流会在选定资源池中创建所有虚拟机的快照。

后续步骤

您可以生成一个文档以在其中查看有关工作流的信息。请参见生成工作流文档。

脚本

2

Orchestrator 使用 JavaScript 创建构建块，您可以从中创建操作、工作流元素以及相关策略（用于访问 Orchestrator 所嵌入技术的 API）。

Orchestrator 使用 Mozilla Rhino 1.7R4 JavaScript 引擎作为其脚本引擎。脚本引擎提供变量类型检查、名称空间管理、自动完成和异常处理。

Orchestrator 工作流引擎可让您使用基本 JavaScript 语言特性，例如 if 语句、循环 (loops)、数组 (arrays) 和字符串 (strings)。您可以使用 Orchestrator API 所提供脚本中的对象，或通过插件导入到 Orchestrator 并映射到 JavaScript 对象的任何其他 API 中的对象。有关 Rhino 的信息，请参见 Mozilla Rhino 网站。

本章讨论了以下主题：

需要编写脚本的 Orchestrator 元素

并非所有 Orchestrator 元素都需要编写脚本。为了最大限度实现应用程序灵活性，您可以通过添加 JavaScript 函数的方式对某些元素进行自定义。

您可以在以下 Orchestrator 元素中添加脚本。

操作

操作是脚本函数。您可以将为某个操作编写的脚本限制为单个操作步骤，最大程度提升该操作被其他元素（如其他工作流）重复使用的可能性。此外，一个操作也可包含多个操作步骤，以降低工作流的复杂度，但是这种方式也会降低该操作的可重复使用性。

策略

您可以使用触发事件监视脚本对策略进行设置。发生触发事件时，策略会启动您在脚本中定义的编排操作。

工作流

可编辑脚本工作流元素允许您编写可在工作流中使用的自定义脚本操作或操作序列。您还可以对脚本中的自定义决策元素定义布尔决策语句，使其返回 `true` 或 `false`。

Orchestrator 中的 Mozilla Rhino 实现限制

Orchestrator 使用 Mozilla Rhino 1.7R4 JavaScript 引擎。但是，在 Orchestrator 中实现 Rhino 却存在一些限制。

为工作流编写脚本时，您必须考虑以下在 Orchestrator 中实现 Mozilla Rhino 的限制。

- 当工作流运行时，从一个工作流元素传递到另一个工作流元素的对象不是 JavaScript 对象。传递到下一元素的是具有 JavaScript 映像的 Java 序列化对象。因此，您无法使用完整 JavaScript 语言，只能使用 API Explorer 中存在的类。您无法将函数对象在工作流元素之间传递。
- Orchestrator 在不是 Rhino 根上下文的上下文中以可编辑脚本任务元素的形式运行代码。Orchestrator 将可编辑脚本任务元素和操作以透明方式封装到 JavaScript 函数，随后运行该函数。包含 `System.log(this);` 的可编辑脚本任务元素不会像标准 Rhino 实现那样显示全局对象 `this`。
- 您仅可以调用从脚本而非工作流返回不可序列化对象的操作。若要调用返回不可序列化对象的操作，您必须使用 `System.getModule`*`ModuleName`*`.action()` 方法编写可调用此操作的可编辑脚本任务元素。
- 工作流验证不会检查工作流属性类型是否与操作或子工作流的输入类型不同。如果更改工作流输入参数的类型（例如从 `VIM3:VirtualMachine` 更改为 `VC:VirtualMachine`），但未更新使用原始输入类型的任何可编辑脚本任务或操作，则工作流会进行验证但不会运行。

使用 Orchestrator 脚本 API

对于 Orchestrator 通过其插件访问的技术，Orchestrator API 会将这些技术的所有对象和功能作为 JavaScript 对象和方法进行公开。

例如，您可以通过 Orchestrator API 访问 vCenter Server API 的 JavaScript 实现，从而将 vCenter 操作包含到您创建的脚本元素中。您还可以通过 Orchestrator 服务器上安装的所有其他插件访问对象的 JavaScript 实现。如果为第三方应用程序创建自定义插件，则会将这些对象从其 API 映射到 JavaScript 对象，并由 Orchestrator API 随后公开。

步骤

1 从工作流编辑器访问脚本引擎

Orchestrator 脚本引擎使用 Mozilla Rhino 1.7R4 JavaScript 引擎帮助您对工作流中的脚本元素编写脚本。您可以从工作流编辑器中的**脚本**选项卡访问适用于脚本工作流元素的脚本引擎。

2 从操作或策略编辑器访问脚本引擎

Orchestrator 脚本引擎使用 Mozilla Rhino JavaScript 引擎帮助您编写相关操作或策略的脚本。您可从操作编辑器或策略编辑器中的**脚本**选项卡访问相关操作或策略的脚本引擎。

3 访问 Orchestrator API Explorer

Orchestrator 提供 API Explorer，可用于搜索 Orchestrator API 并查看有关 JavaScript 对象文档，了解可在脚本元素中使用的 JavaScript 对象。

4 使用 Orchestrator API Explorer 查找对象

Orchestrator API 会公开所有插件技术的 API，包括整个 vCenter Server API。Orchestrator API 可帮助您查找需要添加到脚本的对象。

5 编写脚本

Orchestrator 脚本引擎可帮助您编写脚本。自动插入函数和自动完成多行脚本等功能可以提高脚本编写速度，最大程度降低脚本编写出错的可能性。

6 向脚本添加参数

Orchestrator 脚本引擎可帮助您将可用参数导入至脚本中。

7 从 JavaScript 和工作流访问 Orchestrator 服务器文件系统

Orchestrator 会限制仅可从 JavaScript 和工作流访问特定目录中的 Orchestrator 服务器文件系统。

8 访问 JavaScript 中的 Java 类

默认情况下，Orchestrator 会将 JavaScript 的访问权限限制为一组 Java 类。如果想要 JavaScript 访问范围更广的 Java 类，您必须设置 Orchestrator 系统属性来允许相关访问权限。

9 从 JavaScript 访问操作系统命令

Orchestrator API 提供了脚本类 (`Command`)，可在 Orchestrator 服务器主机操作系统中运行命令。为防止对 Orchestrator 服务器主机未经授权的访问，默认情况下，Orchestrator 应用程序没有 `Command` 类的运行权限。

从工作流编辑器访问脚本引擎

Orchestrator 脚本引擎使用 Mozilla Rhino 1.7R4 JavaScript 引擎帮助您对工作流中的脚本元素编写脚本。您可以从工作流编辑器中的**脚本**选项卡访问适用于脚本工作流元素的脚本引擎。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 在 Orchestrator 客户端**工作流**视图中，右键单击一个工作流，然后选择**编辑**。

3 单击工作流编辑器中的**架构**选项卡。

4 将可编辑脚本任务元素或自定义决策元素添加到工作流架构。

5 单击可编辑脚本元素的**脚本**选项卡。

您即访问了脚本引擎，并可使用其对工作流元素的脚本函数进行定义。使用**脚本**选项卡可以浏览 API、查阅相关对象的文档、搜索对象并编写 JavaScript。

后续步骤

使用 API Explorer 搜索 Orchestrator API。

从操作或策略编辑器访问脚本引擎

Orchestrator 脚本引擎使用 Mozilla Rhino JavaScript 引擎帮助您编写相关操作或策略的脚本。您可从操作编辑器或策略编辑器中的**脚本**选项卡访问相关操作或策略的脚本引擎。

步骤

1 从 Orchestrator 客户端的下拉菜单中选择一个选项，具体视想要对其编辑脚本的元素类型而定。

选项	描述
设计	选择此选项可编辑操作元素的脚本。
运行	选择此选项可编辑策略的脚本。

2 右键单击**操作**或**策略**视图中的操作或策略，然后选择**编辑**。

3 单击操作编辑器或策略编辑器中的**脚本**选项卡。

您即访问了脚本引擎，并可使用其对操作或策略元素的脚本函数进行定义。使用**脚本**选项卡可以浏览 API、查阅相关对象的文档、搜索对象并编写 JavaScript。

后续步骤

使用 API Explorer 搜索 Orchestrator API。

访问 Orchestrator API Explorer

Orchestrator 提供 API Explorer，可用于搜索 Orchestrator API 并查看有关 JavaScript 对象文档，了解可在脚本元素中使用的 JavaScript 对象。

您可以在 Orchestrator 文档主页查看 vCenter Server 插件的在线版脚本 API。

步骤

1 登录到 Orchestrator 客户端。

2 选择**工具 > API Explorer**。

系统会显示 API Explorer。您可用其搜索 Orchestrator API 的所有对象和函数。

后续步骤

使用 API Explorer 对可编辑脚本元素编写脚本。

使用 Orchestrator API Explorer 查找对象

Orchestrator API 会公开所有插件技术的 API，包括整个 vCenter Server API。Orchestrator API 可帮助您查找需要添加到脚本的对象。

前提条件

打开 API Explorer。

步骤

1 在 API Explorer **搜索**文本框中输入对象的名称或部分名称，然后单击**搜索**。

若要将搜索限制为特定对象类型，请取消选中或选中**脚本类**、**属性和方法**以及**类型和枚举**复选框。

2 双击建议列表中的元素。

对象会在左侧的层次结构列表中突出显示。层次结构列表下的文档窗格会显示有关对象的信息。

后续步骤

使用您在脚本中找到的对象。

API Explorer 中的 JavaScript 对象

Orchestrator API Explorer 识别**脚本**选项卡左侧层次结构树或 API Explorer 对话框中的不同种类 JavaScript 对象，并将这些对象组合到一起。API Explorer 使用图标帮助识别不同种类的对象。

下表描述了 Orchestrator API 的对象及其图标。

表 2-1. Orchestrator API 中的 JavaScript 对象

对象	层次结构列表中的图标	说明
类型		类型
函数集		包含一组静态方法的内部类型
原始		原始类型
对象		标准 Orchestrator 脚本对象
属性		JavaScript 属性
方法		JavaScript 方法
构造函数		JavaScript 构造函数
枚举		JavaScript 枚举
字符串集		字符串集，默认值
模块		操作的集合
插件	插件定义的图像	插件向 Orchestrator 公开的 API

编写脚本

Orchestrator 脚本引擎可帮助您编写脚本。自动插入函数和自动完成多行脚本等功能可以提高脚本编写速度，最大程度降低脚本编写出错的可能性。

前提条件

打开要编辑的脚本元素，然后单击其**脚本**选项卡。

步骤

1 浏览**脚本**选项卡左侧的对象层次结构列表或使用 API Explorer 搜索功能选择要添加到此脚本的类型、类或方法。

2 右键单击相应类型、类或方法，然后选择**复制**。

如果脚本引擎不允许复制所选的元素，说明此对象将不适用于该脚本上下文。

3 在脚本编辑器中单击右键，然后将已复制的元素粘贴到脚本中的适当位置。

脚本引擎会将此元素输入至脚本中，包括相应的构造函数和实例名称。

例如，如果您复制了 `Date` 对象，脚本引擎会将以下代码粘贴至脚本中。

```
var myDate = new Date();
```

4 复制并粘贴要添加至脚本的方法。

脚本引擎会完成方法调用，同时添加所需的属性。

例如，如果您从 `com.vmware.library.vc.vm` 模块复制了 `cloneVM()` 方法，脚本引擎会将以下代码粘贴至脚本中。

```
System.getModule("com.vmware.library.vc.vm").cloneVM(vm,folder,name,spec)
```

脚本引擎会突出显示您已在元素中定义的参数。未定义的参数则不会突出显示。

5 将光标放置在已粘贴至脚本中的元素结尾处，然后按“Ctrl + 空格”即可调出包含该对象可调用的方法和属性的上下文列表并从中选择。

注 自动完成功能目前处于试验阶段。

您即向脚本添加了对象和函数。

后续步骤

向脚本添加参数。

脚本关键字颜色编码

在脚本工作流元素的**脚本**选项卡上添加脚本时，有些类型的关键字会显示为不同颜色以增强代码的可读性。

所有脚本均会显示为标准黑色字体，除非另有说明。

表 2-2. 脚本关键字颜色编码

关键字类型	脚本选项卡中的文本颜色
标准 JavaScript 关键字，例如：`if`、`else`、`for` 和 `new`	黑色粗体
变量声明，即 `var`	绿色
循环中的修饰符，例如 `in`	红色
空变量值	紫色
非空变量值	绿色

表 2-2. 脚本关键字颜色编码 （续）

关键字类型	脚本选项卡中的文本颜色
代码备注	灰色斜体
Orchestrator 插件对象类型，例如 `VC:VirtualMachine` 或 `VC:Host`	绿色
输出文本	绿色
工作流属性	粉色
工作流输入	粉色
工作流输出	粉色

向脚本添加参数

Orchestrator 脚本引擎可帮助您将可用参数导入至脚本中。

如果您已为要编辑的元素定义了相关参数，这些参数将作为链接显示在**脚本**选项卡的工具栏中。

前提条件

打开要编辑的脚本元素及其**脚本**选项卡。

步骤

1. 在**脚本**选项卡的脚本编辑器中，将光标移动到脚本的适当位置。
2. 单击**脚本**选项卡工具栏中的参数链接。

   Orchestrator 会将此参数插入到光标位置。
3. 向脚本插入空值参数。

   如果将空值传递到原始类型（例如整数、布尔和字符串），Orchestrator 脚本 API 会自动设置此参数的默认值。

您即向此脚本添加了参数。

后续步骤

在脚本中添加 Java 类的访问权限。

从 JavaScript 和工作流访问 Orchestrator 服务器文件系统

Orchestrator 会限制仅可从 JavaScript 和工作流访问特定目录中的 Orchestrator 服务器文件系统。

JavaScript 函数和工作流仅在永久目录 `c:\orchestrator` 中具有读取、写入和执行权限。

Orchestrator 管理员可以通过设置系统属性，修改 JavaScript 函数和工作流在哪些文件夹中具有读取、写入和执行访问权限。有关设置系统属性的信息，请参见《安装和配置 VMware vRealize Orchestrator》。

JavaScript 函数和工作流在服务器系统默认的临时 I/O 文件夹中也具有读取、写入和执行权限。对于访问具有完整权限的文件系统，写入默认临时 I/O 文件夹是唯一具有可移植性、可靠性和独立于配置的方法。但是，写入临时 I/O 文件夹的文件会在您重新引导服务器时丢失。

您可以通过在 JavaScript 函数中调用 `System.getTempDirectory` 方法来获取默认临时 I/O 文件夹。

使用 System.getTempDirectory 方法访问服务器文件系统

作为写入 Orchestrator 服务器系统上文件夹（管理员在其中设置了相应权限）的备用方法，您可以写入默认临时 I/O 文件夹。

默认情况下，Orchestrator 在默认临时 I/O 文件夹拥有完全读取、写入和执行权限。您可以使用 JavaScript 函数中的 `System.getTempDirectory` 方法来获取默认临时 I/O 文件夹。

步骤

- 包含以下 JavaScript 函数中的代码行来访问 `java.io.temp-dir` 文件夹。

```
var tempDir = System.getTempDirectory()
```

访问 JavaScript 中的 Java 类

默认情况下，Orchestrator 会将 JavaScript 的访问权限限制为一组 Java 类。如果想要 JavaScript 访问范围更广的 Java 类，您必须设置 Orchestrator 系统属性来允许相关访问权限。

默认情况下，Orchestrator JavaScript 引擎仅能访问 `java.util.*` 软件包中的类。

Orchestrator 管理员可设置系统属性允许访问 JavaScript 函数中的其他 Java 类。有关设置系统属性的信息，请参见《安装和配置 VMware vRealize Orchestrator》。

从 JavaScript 访问操作系统命令

Orchestrator API 提供了脚本类 (`Command`)，可在 Orchestrator 服务器主机操作系统中运行命令。为防止对 Orchestrator 服务器主机未经授权的访问，默认情况下，Orchestrator 应用程序没有 `Command` 类的运行权限。

Orchestrator 管理员可通过设置 `com.vmware.js.allow-local-process=true` 系统属性来允许访问 `Command` 脚本类。

有关设置系统属性的信息，请参见《安装和配置 VMware VCenter Orchestrator》。

有关设置系统属性的信息，请参见《安装和配置 VMware VCenter Orchestrator》。

使用 xPath 表达式与 vCenter Server 插件

您可以在 vCenter Server 插件中使用查找器方法来查询 vCenter Server 清单对象。您可以使用 XPath 表达式来定义搜索参数。

vCenter Server 插件包含一组对象查找器方法，例如：`getAllDatastores()`、`getAllResourcePools()`、`findAllForType()`。您可以使用这些方法来访问连接到 Orchestrator 服务器的 vCenter Server 实例的清单，并按 ID、名称或其他属性搜索对象。

出于性能考虑，查找器方法不返回查询对象的任何属性，除非您在搜索查询中指定一组属性。

您可以在 Orchestrator 文档主页参考在线版本的 vCenter Server 插件脚本 API。

重要事项 基于 XPath 表达式的查询可能会影响 Orchestrator 性能，因为查找器对象会返回 vCenter Server 端给定类型的所有对象，并会将查询筛选器应用到 vCenter Server 插件端。

使用 xPath 表达式与 vCenter Server 插件

调用查找器方法时，可以使用基于 XPath 查询语言的表达式。搜索将返回与 XPath 表达式匹配的所有清单对象。如果想查询任何属性，可以将其以字符串数组形式包含到搜索脚本中。

以下 JavaScript 示例使用了 `VcPlugin` 脚本对象以及 XPath 表达式，用于返回 vCenter Server 所属受管对象的所有数据存储对象的名称，并且在其名称中包含字符串 **ds**。

```
var datastores = VcPlugin.getAllDatastores(null, "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
     System.log(datastore.name);
 }
```

使用 `Server` 脚本对象和 `findAllForType` 查找器方法可以调用同一 XPath 表达式。

```
var datastores = Server.findAllForType("VC:Datastore", "xpath:name[contains(.,'ds')]");
for each (datastore in datastores){
     System.log(datastore.name);
 }
```

以下脚本示例会返回其 ID 以数字 **1** 开头的所有主机系统对象的名称。

```
var hosts = VcPlugin.getAllHostSystems(null, "xpath:id[starts-with(.,'1')]");
for each (host in hosts){
     System.log(host.name);
}
```

以下脚本会返回其名称中包含字符串 **DC**（无论大写或小写）的所有数据中心对象的名称和 ID。该脚本还可检索**标记**属性。

```
var datacenters = VcPlugin.getAllDatacenters(['tag'], "xpath:name[contains(translate(., 'DC', 'dc'),
'dc')]");
for each (datacenter in datacenters){
     System.log(datacenter.name + " "  +  datacenter.id);
}
```

异常处理准则

Orchestrator 的 Mozilla Rhino JavaScript 引擎实现支持异常处理，从而处理错误。在脚本中编写异常处理程序时，必须使用以下准则。

- 使用以下欧洲计算机制造商协会 (ECMA) 的错误类型。将 `Error` 用作插件函数返回的通用异常，以及以下特定错误类型。
  - `TypeError`

- RangeError
- EvalError
- ReferenceError
- URIError
- SyntaxError

以下示例显示了 URIError 的定义。

```
try {
     ...
        throw new URIError("VirtualMachine with ID 'vm-0056'
                       not found on 'vcenter-test-1'") ;
     ...
} catch ( e if e instanceof URIError ) {

}
```

- 脚本捕捉不到的所有异常一定是 <type>:SPACE<human readable message> 形式的简单字符串对象，如下所示。

```
throw "ValidationError: The input parameter 'myParam' of type 'string' is too short."
```

- 尽可能清晰地编写可供用户阅读的消息。
- 简单字符串异常类型检查必须使用以下模式。

```
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
}
```

- 简单字符串异常类型检查必须在工作流的脚本元素中使用以下模式。

```
if (typeof(errorCode)=="string"
      && errorCode.indexOf("VMwareNoSpaceLeftOnDatastore:")
      == 0) {
      // Do something useful here
}
```

Orchestrator JavaScript 示例

您可以对 Orchestrator JavaScript 示例进行剪切、粘贴和调整，帮助编写常用编排任务的 JavaScript。

- 基本脚本示例

  工作流脚本元素、操作和策略需要对常用任务进行基本脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- 电子邮件脚本示例

  工作流脚本元素可以包括对常见电子邮件任务的脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- 文件系统脚本示例

  工作流脚本元素、操作和策略需要对常用文件系统任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- LDAP 脚本示例

  工作流脚本元素、操作和策略需要对常用 LDAP 任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- 日志记录脚本示例

  工作流脚本元素、操作和策略需要对常用日志记录任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- 网络连接脚本示例

  工作流脚本元素、操作和策略需要对常用网络连接任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

- 工作流脚本示例

  工作流脚本元素、操作和策略需要对常用工作流任务的脚本编辑示例。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

基本脚本示例

工作流脚本元素、操作和策略需要对常用任务进行基本脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

访问 XML 文档

以下 JavaScript 示例可让您使用 Orchestrator JavaScript API 中的 ECMAScript for XML (E4X) 实现从 JavaScript 访问 XML 文档。

注 除了在 JavaScript API 中实现 E4X 外，Orchestrator 还在 XML 插件中提供了一个文档对象模型 (Document Object Model, DOM) XML 实现。有关 XML 插件及其示例工作流的详细信息，请参见《使用 vRealize Orchestrator 插件》。

```
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 + ")");
}
```

从哈希表获取并设置属性

以下 JavaScript 示例用于在哈希表中设置属性和从哈希表获取属性。在下例中，键始终是一个“字符串”，而值则可能是对象、数字、布尔值或字符串。

```
var table = new Properties() ;
table.put("myKey",new Date()) ;
// get the object back
var myDate= table.get("myKey") ;
System.log("Date is : "+myDate) ;
```

替换字符串的内容

以下 JavaScript 示例用于替换字符串的内容并将其替换成新内容。

```
var str1 = "'hello'" ;
var reg = new RegExp("(')", "g");
var str2 = str1.replace(reg,"\\'") ;
System.log(""+str2) ; // result : \'hello\'
```

比较类型

以下 JavaScript 示例用于检查一个对象与给定的对象类型是否匹配。

```
var path = 'myurl/test';
if(typeof(path, string)){
  throw("string");
else {
```

```
  throw("other");
}
```

在 Orchestrator 服务器中运行命令

以下 JavaScript 示例可在 Orchestrator 服务器上运行命令行。使用与启动服务器所用的同一凭据即可。

注 默认情况下，访问文件系统会受到限制。

```
var cmd = new Command("ls -al") ;
cmd.execute(true) ;
System.log(cmd.output) ;
```

电子邮件脚本示例

工作流脚本元素可以包括对常见电子邮件任务的脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

运行邮件工作流时，它会使用您在“配置邮件”工作流中设置的默认邮件服务器配置。您可以使用输入参数或在工作流脚本元素中定义自定义值以替代默认值。

获取电子邮件地址

以下 JavaScript 示例会获取当前运行脚本的所有者的电子邮件地址。

```
var emailAddress = Server.getRunningUser().emailAddress ;
```

发送电子邮件

以下 JavaScript 示例会通过 SMTP 服务器向规定的收件人发送规定内容的电子邮件。

```
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() ;
```

检索电子邮件

以下 JavaScript 示例会使用 `MailClient` 类提供的脚本 API 检索特定电子邮件帐户的消息，而不将其删除。

```
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();
}
```

文件系统脚本示例

工作流脚本元素、操作和策略需要对常用文件系统任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

向简单文本文件添加内容

以下 JavaScript 示例会向文本文件添加内容。

```
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() ;
```

获取文件内容

以下 JavaScript 示例会从安装了 Orchestrator 服务器的计算机中获取文件内容。

```
var tempDir = System.getTempDirectory() ;
var fileReader = new FileReader(tempDir + "/readme.txt") ;
fileReader.open() ;
var fileContentAsString = fileReader.readAll();
fileReader.close() ;
```

LDAP 脚本示例

工作流脚本元素、操作和策略需要对常用 LDAP 任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

将 LDAP 对象转换为 Active Directory 对象

以下 JavaScript 示例可将 LDAP 组元素转换成 Active Directory 用户组对象，以及反向转换。

```
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) ;
```

日志记录脚本示例

工作流脚本元素、操作和策略需要对常用日志记录任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

持久性日志记录

以下 JavaScript 示例会创建持久性日志条目。

```
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");
```

非持久性日志记录

以下 JavaScript 示例会创建非持久性日志条目。

```
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");
```

网络连接脚本示例

工作流脚本元素、操作和策略需要对常用网络连接任务进行脚本编辑。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

从 URL 获取文本

以下 JavaScript 示例会访问 URL、获取文本并将其转换为字符串。

```
var url = new URL("http://www.vmware.com") ;
var htmlContentAsString = url.getContent() ;
```

工作流脚本示例

工作流脚本元素、操作和策略需要对常用工作流任务的脚本编辑示例。您可以将这些示例剪切、粘贴和调整应用到自己的脚本元素中。

按当前用户返回所有工作流运行

以下 JavaScript 示例会从服务器获取所有工作流运行，并检查其是否属于当前用户。

```
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;
```

访问当前工作流令牌

您可以使用 `workflow` 变量访问当前工作流令牌。该变量是类型 `WorkflowToken` 的一个对象，提供当前工作流运行的访问权限。以下 JavaScript 示例可获取工作流令牌的 ID 及其开始日期。

```
System.log("Current workflow run ID: " + workflow.id);
System.log("Current workflow run start date: "+workflow.startDate);
```

调度工作流

以下 JavaScript 示例会通过一组给定属性来启动工作流，并会调度其在一个小时后启动。

```
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);
```

对循环中的选择对象运行工作流

以下 JavaScript 示例会获取虚拟机阵列，并对 For 循环中的每台虚拟机运行工作流。VMs 和 workflowToRun 为工作流输入。

```
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);
}
```

开发操作

3

Orchestrator 提供预定义操作库。操作表示您在工作流和脚本中用作构建块的各个函数。

操作是 JavaScript 函数，可获取多个输入参数并拥有单个返回值。这些操作可以在 Orchestrator API 中的任何对象上调用，或在您使用插件导入 Orchestrator 的任何 API 中的对象上调用。

在工作流运行时，操作会从工作流属性获取输入参数。这些属性可以是工作流的初始输入参数，或是其他元素在工作流中运行时设置的属性。

本章讨论了以下主题：

重用操作

如果将单个函数定义为一个操作而不是将其直接编码到可编辑脚本任务工作流元素，您即在库中公开了该操作。如果操作在库中可见，则其他工作流也可以使用该操作。

如果在定义操作时独立于调用这些操作的工作流，那么可以更加轻松地更新或优化这些操作。逐个定义操作也能允许其他工作流重用这些操作。工作流运行时，Orchestrator 会在工作流首次运行操作时对每个操作进行缓存。Orchestrator 随后可以重用这些缓存的操作。缓存操作对于在工作流中递归调用以及加速循环非常有用。

您可以复制操作、将其导出到其他工作流或软件包，或将其移动到操作层次结构列表中的不同模块。

访问“操作”视图

Orchestrator 客户端界面提供**操作**视图，用于访问 Orchestrator 服务器的操作库。

Orchestrator 客户端界面的**操作**视图采用层次结构列表形式，列出了 Orchestrator 服务器中所有可用的操作。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**操作**视图。

3 展开操作层次结构列表的各个节点即可浏览操作库。

您可以使用**操作**视图查看库中相关操作的信息，以及创建和编辑这些操作。

“操作”视图的组件

单击操作层次结构列表中的某个操作时，此操作的相关信息会显示在 Orchestrator 客户端的右侧窗格中。

操作视图提供四个选项卡。

常规	显示此操作的常规信息，包括操作名称、版本号、权限和说明。
脚本	显示此操作的返回类型、输入参数、以及定义此操作功能用于的 JavaScript 代码。
事件	显示此操作遇到或触发的所有事件。
权限	显示有权访问此操作的用户和用户组。

创建操作

您可以将单个函数定义可由其他元素（如工作流）使用的操作。操作是包含已定义的输入/输出参数与权限的 JavaScript 函数。

- 创建操作

  在将单个函数定义为一个操作（而不是直接将其编码到可编辑脚本任务工作流元素中）时，您可在库中将其公开以供其他工作流使用。

- 查找实现某个操作的元素

  在对操作进行编辑或改变其行为的过程中，您可能会无意间中断了实现该操作的工作流或应用程序。Orchestrator 能够查找出实现特定元素的所有操作、工作流或软件包。您可以检查对该元素的修改是否会影响其他元素的操作。

- 操作编码准则

  为了优化工作流性能和最大限度地实现操作重用，您在创建操作时应当遵守基本的编码准则。

创建操作

在将单个函数定义为一个操作（而不是直接将其编码到可编辑脚本任务工作流元素中）时，您可在库中将其公开以供其他工作流使用。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**操作**视图。

3 展开操作层次结构列表的根，然后导航到要在其中创建操作的模块。

4 右键单击该模块，然后选择**添加操作**。

5 在文本框中输入操作的名称，然后单击**确定**。

您的自定义操作即会添加到操作库中。

6 右键单击该操作，然后选择**编辑**。

7 单击**脚本**选项卡。

8 若要更改默认的返回类型，请单击**空**链接。

9 单击箭头图标以添加操作输入参数。

10 编写操作脚本。

11 设置操作权限。

12 单击**保存并关闭**。

您即创建了自定义操作并添加了操作输入参数。

后续步骤

您可以在工作流中使用新的自定义操作。

查找实现某个操作的元素

在对操作进行编辑或改变其行为的过程中，您可能会无意间中断了实现该操作的工作流或应用程序。Orchestrator 能够查找出实现特定元素的所有操作、工作流或软件包。您可以检查对该元素的修改是否会影响其他元素的操作。

重要事项 **查找使用该元素的元素**功能能够检查所有软件包、工作流和策略，但不会在脚本中检查。因此，修改操作时可能会影响某个在脚本中调用该操作的元素，而**查找使用该元素的元素**功能并未识别出该脚本。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**操作**视图。

3 展开操作层次结构列表中的节点以导航到特定操作。

4 右键单击该操作并选择**查找使用该元素的元素**。

对话框会显示实现这一操作的所有元素，例如工作流或软件包。

5 双击结果列表中的一个元素，以在 Orchestrator 客户端中显示该元素。

您即找到了实现该操作的所有元素。

后续步骤

您可以检查对该元素的修改是否会影响任何其他元素。

操作编码准则

为了优化工作流性能和最大限度地实现操作重用，您在创建操作时应当遵守基本的编码准则。

基本操作准则

创建操作时，必须使用基本准则。

- 每一项操作都必须包含其角色和功能的说明。
- 编写简短的基础性操作，并将其组合到一个工作流中。
- 避免编写需要完成多项功能的复杂操作，因为这样会限制操作重用的潜力。
- 避免编写需要长时间运行的操作，反之，应在工作流中创建一个循环，将操作元素之后的“等待事件”和“等待定时器”元素纳入其中。
- 不要在操作中编写检查点，工作流会在每个元素运行的开始和结束位置设置检查点。
- 避免在操作中编写任何循环，反之，应在工作流中创建循环。如果服务器重启，正在运行的工作流会从上个检查点（即某个元素的开始位置）恢复运行。如果在操作中编写了循环，并且在工作流正在运行该操作时发生了服务器重启，则工作流会在该操作开始处的检查点恢复运行，而循环则会重新从头开始。

操作命名准则

命名操作时应遵守基本准则

- 操作名称应使用英文。
- 操作名称首字母应小写。名称中每个连结单词的首字母应大写。例如 `myAction`。
- 操作名称应尽量含义明确，清楚说明此操作的功能。例如 `backupAllVMsInPool`。
- 模块名称应尽量含义明确。
- 各模块的名称必须唯一。
- 模块名称应使用与互联网地址相反的格式。例如 `com.vmware.myactions.myAction`。

操作参数准则

编写操作参数定义时应遵守基本准则。

- 参数名称应使用英文。
- 参数名称首字母应小写。
- 参数名称尽量含义明确。
- 参数名称最好限制为一个单词。如果名称中必须包含多个单词，则名称中每个连结单词的首字母应大写。例如 `myParameter`。
- 表示对象数组的参数应使用复数形式。
- 变量名称应含义明确，不存在歧义，例如 `displayName`。
- 每个参数均应包含一项描述，说明其用途。

- 请勿在一个操作中使用过多参数。

使用操作版本历史记录

您可以使用版本历史记录将操作恢复为先前版本。您可以将操作状态恢复为较早或较新的操作版本。您还可以比较当前状态的操作与已保存版本的操作之间的差异。

在您增加并保存操作版本时，Orchestrator 会为每个操作创建新的版本历史记录项目。对操作的后续更改不会更改当前版本项目。例如，如果您创建操作版本 1.0.0 并进行保存，操作的状态会存储在数据库中。如果对操作进行任何更改，可以在 Orchestrator 客户端中保存操作状态，但您无法将更改应用到操作版本 1.0.0。如果要在数据库中存储更改，必须创建并保存一个后续操作版本。版本历史记录会连同操作一起保存在数据库中。

当您删除操作时，Orchestrator 会在数据库中将元素标记为已删除，而不会将元素的版本历史记录从数据库中删除。这样您就可以还原已删除的操作。请参见还原已删除操作。

前提条件

打开操作进行编辑。

步骤

1. 单击操作编辑器中的**常规**选项卡。
2. 单击**显示版本历史记录**。

   此时会显示版本历史记录窗口。
3. 选择操作版本并单击**与当前版本的差异**以比较差异。

   此时显示的窗口会显示当前操作版本与选定操作版本之间的差异。
4. 选择操作版本并单击**恢复**以还原操作状态。

   小心　如果尚未保存当前操作版本，则其会从版本历史记录中删除，并且无法恢复到当前版本。

   操作状态会恢复到选定版本的状态。

还原已删除操作

您可以还原已从库中删除的操作。

步骤

1. 从 Orchestrator 客户端的下拉菜单中，选择**设计**。
2. 单击**操作**视图。
3. 导航到要在其中还原已删除操作的文件夹。
4. 右键单击文件夹并选择**还原已删除操作**。
5. 选择要还原的操作并单击**还原**。

相关操作随即会显示在选定文件夹中。

创建资源元素

4

工作流可能需要您创建独立于 Orchestrator 的对象以将其用作属性。若要在工作流中将外部对象用作属性，您需要将其作为资源元素导入 Orchestrator 服务器。

工作流可用作资源元素的对象包含图像文件、脚本、XML 模板、HTML 文件等。Orchestrator 服务器中运行的任意工作流都可以使用您导入 Orchestrator 的任意资源元素。

将对象作为资源元素导入 Orchestrator 可让您在单一位置对对象进行更改，并且将这些更改自动传播到使用该资源元素的所有工作流。

您可以将资源元素整理到文件夹。资源元素的最大大小为 16 MB。

本章讨论了以下主题：

查看资源元素

您可以在 Orchestrator 客户端中查看现有资源元素，以检查其内容并了解哪个工作流使用了该资源元素。

步骤

1. 从 Orchestrator 客户端的下拉菜单中，选择**设计**。
2. 单击**资源**视图。
3. 展开层次结构树形查看器，导航到资源元素。
4. 单击资源元素以在右侧窗格中显示其相关信息。
5. 单击**查看器**选项卡以显示资源元素的内容。
6. 右键单击资源元素并选择**查找使用该元素的工作流**。

   Orchestrator 会列出使用该资源元素的所有工作流。

后续步骤

导入并编辑资源元素。

导入外部对象以用作资源元素

工作流可能需要您创建独立于 Orchestrator 的对象以将其用作属性。若要在工作流中将外部对象用作属性，您必须将其作为资源元素导入 Orchestrator 服务器。

前提条件

验证您是否有映像文件、脚本、XML 模板、HTML 文件或其他类型的对象要导入。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**资源**视图。

3 在层次结构列表或根目录中右键单击资源文件夹，然后选择**新建文件夹**以创建要在其中存储资源元素的文件夹。

4 右键单击要在其中导入资源元素的资源文件夹，然后选择**导入资源**。

5 选择要导入的资源并单击**打开**。

Orchestrator 随即会将资源元素添加到您选定的文件夹。

资源元素即导入到 Orchestrator 服务器。

后续步骤

编辑资源元素的常规信息并设置用户访问权限。

编辑资源元素信息和访问权限

将对象作为资源元素导入 Orchestrator 服务器后，您可以编辑资源元素的详细信息和权限。

前提条件

确认您已将映像、脚本、XML 或 HTML 文件或者任何其他类型的对象作为资源元素导入 Orchestrator 中。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**资源**视图。

3 右键单击资源元素，然后选择**编辑**。

4 单击**常规**选项卡并设置资源元素名称、版本和描述。

5 单击**权限**选项卡并单击**添加访问权限**图标 () 以定义用户组的权限。

6 在**筛选**文本框中输入用户组名称。

7 选择用户组并单击**确定**。

8 右键单击用户组并选择**添加访问权限**。

9 勾选相应复选框以设置该用户组的权限级别，然后单击**确定**。

权限不会累积。若要允许用户查看资源元素、在其工作流中使用资源元素以及更改权限，您必须勾选所有复选框。

10 单击**保存并关闭**以退出该编辑器。

编辑有关资源元素的常规信息并设置用户访问权限。

后续步骤

将资源元素保存到文件以进行更新，或将其添加到工作流。

将资源元素保存为文件

您可以将资源元素保存为本地系统上的文件。将资源元素保存为文件可让您进行编辑。

您无法在 Orchestrator 客户端中编辑资源元素。例如，如果资源元素为 XML 配置文件或脚本，则必须本地保存以进行修改。

前提条件

验证 Orchestrator 服务器是否包含您可以保存为文件的资源元素。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**资源**视图。

3 右键单击资源元素，然后选择**保存为文件**。

4 对文件进行必要的修改。

您即把资源元素保存为文件。

后续步骤

在 Orchestrator 服务器中更新资源元素。

更新资源元素

如果想要更新资源元素，必须将其导出到文件系统、使用适当的工具编辑导出的文件，并导入编辑后的文件来更新资源元素。

前提条件

确认您已将映像、脚本、XML 或 HTML 文件或者任何其他类型的对象作为资源元素导入 Orchestrator 中。

步骤

1 修改本地系统中资源元素的源文件。

2 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

3 单击**资源**视图。

4 在层次结构列表中导航到您更新的资源元素。

5 右键单击资源元素并选择**更新资源**。

6 （可选） 单击**查看器**选项卡以验证 Orchestrator 是否已更新资源元素。

您即更新了 Orchestrator 服务器包含的资源元素。

将资源元素添加到工作流

资源元素是外部对象，您可以将其导入 Orchestrator 服务器以便在工作流运行时将其用作属性。例如，工作流可以在运行时使用导入的定义了映射的 XML 文件将一种类型的数据转换为另一种，或者使用定义了函数的脚本。

前提条件

确认 Orchestrator 服务器中具有以下对象：

- 作为资源元素导入 Orchestrator 中的映像、脚本、XML 或 HTML 文件或者任何其他类型的对象。
- 需要将资源元素用作属性的工作流。

步骤

1 从 Orchestrator 客户端的下拉菜单中，选择**设计**。

2 单击**工作流**视图。

3 展开层次结构树形查看器以导航到需要将资源元素用作属性的工作流。

4 右键单击工作流并选择**编辑**。

5 在**常规**选项卡上的“属性”面板中，单击**添加属性**图标 ( )。

6 单击属性名称，并为该属性键入新名称。

7 单击**类型**以设置属性类型。

8 在**选择类型**对话框中，在**筛选器**框中键入**资源**，以搜索对象类型。

选项	操作
将单个资源元素定义为属性	从列表中选择 `ResourceElement`。
将包含多个资源元素的文件夹定义为属性	从列表中选择 `ResourceElementCategory`。

9 单击**值**，并在**筛选器**文本框中键入资源元素的名称或类别。

10 在建议的列表中，选择资源元素或包含资源元素的文件夹，然后单击**选择**。

11 单击**保存并关闭**以退出该编辑器。

这样即将资源元素或资源元素文件夹作为属性添加到工作流。

创建软件包

5

软件包可用于在 Orchestrator 服务器之间分发内容。软件包中可包含工作流、操作、策略模板、配置或资源。

向软件包中添加元素时，Orchestrator 会检查依赖关系并将任何从属元素也添加到软件包中。例如：如果您添加使用了某些操作或其他工作流的工作流，Orchestrator 会将这些操作和工作流也添加到软件包中。

导入软件包时，服务器会将软件包不同内容元素与匹配的本地元素进行版本比较。比较结果会显示本地元素和导入元素之间的版本差异。管理员可以决定是导入软件包，还是选择导入特定元素。

软件包通过数字化权限管理来控制接收服务器对软件包内容的使用方式。Orchestrator 会对软件包签名并进行加密以保护数据。软件包会使用 X509 证书跟踪哪些用户导出并重新分发了元素。

有关使用软件包的详细信息，请参见《使用 VMware vRealize Orchestrator 客户端》。

- 创建软件包

  您能以软件包的形式导出工作流、策略模板、操作、插件参考、资源和配置元素。软件包中的某个元素所实现的所有元素会自动添加到该软件包中，确保不同版本之间的兼容性。如果不想添加参考元素，可以在软件包编辑器中将其删除。

- 对软件包设置用户权限

  您可以对软件包设置不同权限级别，限制不同用户或用户组对该软件包内容的访问权限。

创建软件包

您能以软件包的形式导出工作流、策略模板、操作、插件参考、资源和配置元素。软件包中的某个元素所实现的所有元素会自动添加到该软件包中，确保不同版本之间的兼容性。如果不想添加参考元素，可以在软件包编辑器中将其删除。

前提条件

确认 Orchestrator 服务器包含可向软件包中添加的元素，例如工作流、操作和策略模板。

步骤

1. 从 Orchestrator 客户端的下拉菜单中，选择**管理**。
2. 单击**软件包**视图。
3. 在左侧窗格中单击右键，然后选择**添加软件包**。

4 输入新软件包的名称，然后单击**确定**。

软件包名称的语法为 *domain.your_company.folder.package_name*。

例如：`com.vmware.myfolder.mypackage`。

5 右键单击该软件包并选择**编辑**。

此时系统会打开软件包编辑器。

6 在**常规**选项卡上，添加软件包的说明。

7 在**工作流**选项卡上，将工作流添加到软件包。

- 单击**插入工作流 (列表搜索)** 以在选择对话框中搜索并选择工作流。
- 单击**插入工作流 (树浏览)** 以从层次结构列表中浏览并选择工作流的文件夹。

8 在**策略模板**、**操作**、**配置**、**资源**和**所用插件**选项卡上，将策略模板、操作、配置元素、资源元素和插件添加到软件包。

9 单击**保存并关闭**以退出该编辑器。

您即创建了软件包并为其添加了元素。

后续步骤

设置该软件包的用户权限。

对软件包设置用户权限

您可以对软件包设置不同权限级别，限制不同用户或用户组对该软件包内容的访问权限。

您可以从 Orchestrator LDAP 或 vCenter Single Sign-On 服务器中的用户和用户组选择要为其设置权限的不同用户和用户组。Orchestrator 会定义您对用户或用户组可应用的权限级别。

查看	用户可以查看软件包中的元素，但无法查看架构或脚本。
检查	用户可以查看软件包中的元素，包含架构和脚本。
编辑	用户可以编辑软件包中的元素。
管理员	用户可以对软件包中的元素设置权限。

前提条件

创建软件包，在软件包编辑器中将其打开进行编辑，并向软件包添加必要的元素。

步骤

1 在软件包编辑器中单击**权限**选项卡。

2 单击**添加访问权限**图标 () 对新用户或用户组定义权限。

3 搜索用户或用户组。

搜索结果会显示匹配搜索的所有用户和用户组。

4 选择用户或用户组。

5 勾选相应复选框以设置该用户的权限级别，然后单击**选择**。

若要允许用户查看元素、检查架构和脚本、运行和编辑元素以及更改权限，您必须勾选所有复选框。

6 单击**保存并关闭**以退出该编辑器。

您即创建了软件包并设置了相应的用户权限。

开发插件

6

Orchestrator 可通过其开放插件架构与管理解决方案进行集成。您可以使用 Orchestrator 客户端运行并创建插件工作流并访问插件 API。

本章讨论了以下主题：

- 插件概览
- 插件的内容和结构
- Orchestrator 插件 API 参考
- vso.xml 插件定义文件的元素
- Orchestrator 插件开发的最佳做法

插件概览

Orchestrator 插件必须包含一套标准组件并符合标准架构。以下这些做法有助于您创建各种插件，最大限度地利用种类丰富的外部技术。

- Orchestrator 插件的结构

  Orchestrator 插件具有通用结构，即由实施了特定功能的各种层类型组成。

- 将外部 API 公开至 Orchestrator

  您可以创建 Orchestrator 插件，将外部产品的 API 公开至 Orchestrator 平台。您可以为任何公开 API 的技术创建插件，其中该 API 可被映射到 Orchestrator 可以使用的 JavaScript 对象中。

- 插件的组件

  插件由一套标准组件组成，可以向 Orchestrator 平台公开相关插件技术中的各种对象。

- vso.xml 文件的角色

  您使用 `vso.xml` 文件将插件技术的对象、类、方法和属性映射到 Orchestrator 清单对象、脚本类型、脚本类、脚本方法和属性。`vso.xml` 文件还定义了插件的配置和启动行为。

- 插件适配器的角色

  插件适配器是插件进入 Orchestrator 服务器的入口点。插件适配器可以充当 Orchestrator 服务器中插件技术的数据存储、创建插件工厂以及管理插件技术中发生的事件。

- 插件工厂的角色

  插件工厂用于定义 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。

- 查找器对象的角色

  查找器对象识别并查找插件技术中受管对象类型的特定实例。Orchestrator 可以通过在查找器对象上运行工作流，修改其在插件技术中找到的对象并与之交互。

- 脚本对象角色

  脚本对象是插件技术中对象的 JavaScript 表现形式。插件中的脚本对象会显示在 Orchestrator JavaScript API 中，您可以将其用于工作流和操作中的脚本元素。

- 事件处理程序的角色

  事件即 Orchestrator 在插件技术中发现的对象的状态或属性的更改。Orchestrator 通过实施事件处理程序来监控事件。

Orchestrator 插件的结构

Orchestrator 插件具有通用结构，即由实施了特定功能的各种层类型组成。

Orchestrator 插件的底部三层（即基础架构类、封装类和脚本对象），实现了插件技术和 Orchestrator 之间的连接。

Orchestrator 插件的用户可见部分为顶部三层，即操作、构建块和高级别工作流。

图 6-1. Orchestrator 插件的结构

基础架构类 一组提供了插件技术和 Orchestrator 之间连接的类。基础架构类包含了根据插件定义（例如插件工厂、插件适配器等）实现的类。基础架构类还包含了为通用任务和对象（例如帮助程序、缓存、清单等）提供功能的类。

封装类 一组将插件技术的对象模型调整为要在 Orchestrator 中公开的对象模型的类。

脚本对象 提供了插件技术中封装类、方法和属性等访问权限的 JavaScript 对象类型。在 `vso.xml` 文件中，您定义了插件技术中的哪些封装类、属性和方法将公开到 Orchestrator。

操作 一组可直接在工作流和脚本任务中使用的 JavaScript 函数。操作可使用多个输入参数并拥有单个返回值。

构建块工作流 一组涵盖了所有要随插件一同提供的常规功能的工作流。通常，构建块工作流代表编排技术的用户界面中的操作。构建块工作流可直接使用或包含在高级别工作流内。

高级别工作流 一组涵盖了插件特定功能的工作流。您可以提供高级别工作流来满足具体要求或显示插件使用情况的复杂示例。

将外部 API 公开至 Orchestrator

您可以创建 Orchestrator 插件，将外部产品的 API 公开至 Orchestrator 平台。您可以为任何公开 API 的技术创建插件，其中该 API 可被映射到 Orchestrator 可以使用的 JavaScript 对象中。

插件会将 Java 对象和方法映射到被添加至 Orchestrator 脚本 API 的 JavaScript 对象。如果外部技术公开了 Java API，您可以将 API 直接映射到 JavaScript，以便 Orchestrator 将其用于工作流和操作。

您可以使用 WSDL（Web 服务定义语言）、REST（表述性状态转移）或消息传递服务将公开的 API 与 Java 对象集成，从而以非 Java 语言为公开了 API 的应用程序创建插件。随后将集成的 Java 对象映射到 JavaScript 以供 Orchestrator 使用。

插件技术独立于 Orchestrator。即使您只有（例如 Java 存档，即 JAR 文件中）二进制代码而不是源代码的访问权限，您也可以为外部产品创建 Orchestrator 插件。

插件的组件

插件由一套标准组件组成，可以向 Orchestrator 平台公开相关插件技术中的各种对象。

插件的主要组件是插件适配器、插件工厂和事件实现。您需要将插件适配器、插件工厂和事件实现中所定义的对象和操作映射到 Orchestrator 对象，相关映射关系保存在一个名为 `vso.xml` 的 XML 定义文件中。`vso.xml` 文件会将来自插件技术的对象和功能函数映射到 Orchestrator JavaScript API 中显示的 JavaScript 脚本对象。`vso.xml` 文件还会将来自插件技术的对象类型映射到相应的查找器，这些查找器会显示在 Orchestrator **清单**选项卡中。

插件由以下组件组成。

插件模块 即插件本身，由一组 Java 类、`vso.xml` 文件和软件包进行定义，通过软件包中的各种工作流和操作与通过该插件访问的各种对象进行交互。插件模块为必需组件。

插件适配器 用于定义 Orchestrator 服务器和插件技术之间的接口。适配器是插件在 Orchestrator 平台中应用时的入口点。适配器可用于创建插件工厂、管理插件的加载和卸载、并管理插件技术中各对象所发生的事件。插件适配器为必需组件。

插件工厂 用于定义了 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。适配器会针对 Orchestrator 和插件技术之间打开的客户端会话创

建相应的插件工厂。通过该工厂，您可以在所有客户端连接之间共享一个会话，也可以对每个客户端连接打开一个会话。插件工厂为必需组件。

配置 Orchestrator 并不定义插件对其配置的标准存储方式。您可以使用 Windows 注册表或静态配置文件来存储配置信息，也可将信息存储在数据库或 XML 文件中。Orchestrator 插件可以通过在 Orchestrator 客户端中运行配置工作流的方式进行配置。

查找器 即一套交互规则，用于定义 Orchestrator 在插件技术中查找对象和表示对象的方式。查找器会从插件技术向 Orchestrator 公开的一组对象中检索相关对象。您可在 `vso.xml` 文件中定义各对象之间的关系以允许在对象网络中自由浏览。Orchestrator 通过**清单**选项卡来表示插件技术的对象模型。如果需要向 Orchestrator 公开插件技术中的对象，则查找器为必需组件。

脚本对象 即 JavaScript 对象类型，用于向插件技术中对象、操作和属性等提供访问权限。脚本对象可定义 Orchestrator 通过 JavaScript 访问插件技术对象模型时的访问方式。您可通过 `vso.xml` 文件将插件技术的类和方法映射到 JavaScript 对象。您可以访问 Orchestrator 脚本 API 中的 JavaScript 对象，并将这些对象集成到 Orchestrator 脚本任务、操作和工作流中。如果需要向 Orchestrator JavaScript API 添加脚本类型、类和方法，则脚本对象为必需组件。

清单 即插件技术中的对象实例，Orchestrator 会使用 Orchestrator 客户端**清单**视图中显示的查找器对这些实例进行查找定位。您可以通过运行工作流的方式对清单中的对象执行相应的操作。清单为可选组件。您可以创建这样一个插件，只向 Orchestrator JavaScript API 添加脚本类型和类，而不在清单中公开任何对象实例。

事件 插件技术中某个对象的状态更改。Orchestrator 可以被动侦听插件技术中所发生的事件。Orchestrator 还可以在插件技术中主动触发某些事件。事件为可选组件。

vso.xml 文件的角色

您使用 `vso.xml` 文件将插件技术的对象、类、方法和属性映射到 Orchestrator 清单对象、脚本类型、脚本类、脚本方法和属性。`vso.xml` 文件还定义了插件的配置和启动行为。

`vso.xml` 文件执行以下主体角色。

启动和配置行为 定义了插件的启动方式，并确定插件定义的任何配置实现的位置。加载插件适配器。

清单对象 定义了插件在插件技术中访问的对象类型。插件工厂实现的查找器方法确定这些对象实例的位置，并在 Orchestrator 清单中显示。

脚本类型 将脚本类型添加到 Orchestrator JavaScript API 以表示清单中不同的对象类型。您可以将这些脚本类型用作工作流中的输入参数。

脚本类	将类添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些类。
脚本方法	将方法添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些方法。
脚本属性	将插件技术中的对象属性添加到 Orchestrator JavaScript API，而且您可以在工作流、操作、策略等的脚本元素中使用这些属性。

插件适配器的角色

插件适配器是插件进入 Orchestrator 服务器的入口点。插件适配器可以充当 Orchestrator 服务器中插件技术的数据存储、创建插件工厂以及管理插件技术中发生的事件。

若要创建插件适配器，您需要创建将可实现该 `IPluginAdaptor` 接口的 Java 类。

您创建的插件适配器类可以管理插件工厂、事件以及插件技术中的触发器。`IPluginAdaptor` 接口提供您可用来执行这些任务的方法。

插件适配器执行以下主要角色。

创建工厂	插件适配器最重要的角色就是对 Orchestrator 与插件技术之间的每个连接加载或卸载一个插件工厂实例。插件适配器类会调用 `IPluginAdaptor.createPluginFactory()` 方法来创建可实现该 `IPluginFactory` 接口的类的实例。
管理事件	插件适配器是 Orchestrator 服务器和插件技术之间的接口。插件适配器会管理 Orchestrator 在插件技术对象上执行或观察的事件。适配器通过事件发布程序来管理各种事件。事件发布程序是 `IPluginEventPublisher` 接口的实例，该接口由适配器通过调用 `IPluginAdaptor.registerEventPublisher()` 方法而创建。事件发布程序会在插件技术的对象上设置触发器和计量器，从而在对象上发生特定事件或对象值传递特定阈值时允许 Orchestrator 启动定义的操作。类似地，您可以定义 `PluginTrigger` 和 `PluginWatcher` 实例，从而定义在长时间运行工作流中“等待事件”元素所等待的事件。
设置插件名称	您在 `vso.xml` 文件中为插件输入一个名称。插件适配器会从 `vso.xml` 文件中获取该名称并将其发布到 Orchestrator 客户端的**清单**视图中。
安装许可证	您可以调用方法来安装插件技术在适配器实现过程中所需的任何许可证文件。

有关 `IPluginAdaptor` 接口及其所有方法，以及插件 API 的所有其他类的完整详细信息，请参见 Orchestrator 插件 API 参考。

插件工厂的角色

插件工厂用于定义 Orchestrator 在插件技术中查找对象以及在对象上执行操作时所用的方法。

若要创建插件工厂，您必须从 Orchestrator 插件 API 实现并扩展 `IPluginFactory` 接口。您创建的插件工厂类用于定义查找器函数，Orchestrator 可通过这些函数来访问插件技术中的对象。该工厂允许 Orchestrator 服务器按对象 ID、按对象与其他对象的关系以及搜索查询字符串来查找对象。

插件工厂执行以下主要任务。

查找对象	您可以创建相关函数，按名称和类型来查找对象。使用 `IPluginFactory.find()` 方法按名称和类型查找对象。
查找与其他对象相关的对象	您可以创建相关函数，按给定关系类型来查找与给定对象相关的对象。关系可在 `vso.xml` 文件中定义。您还可以创建相关查找器，按给定关系类型查找与所有父对象相关的从属子对象。实现 `IPluginFactory.findRelation()` 方法按给定关系类型来查找与给定父对象相关的任何对象。实现 `IPluginFactory.hasChildrenInRelation()` 方法来发现父对象实例是否至少存在一个子对象。
定义相关查询以根据自己的条件来查找对象	您可以创建对象查找器，实现您定义的查询规则。实现 `IPluginFactory.findAll()` 方法从而当工厂调用此方法时，会查找所有满足您定义的查询规则的对象。您会在 `QueryResult` 对象中获取 `findAll()` 方法的结果，该对象列出了找到的与您所定义查询规则匹配的所有对象。

有关 `IPluginFactory` 接口及其所有方法、以及插件 API 的所有其他类的更多信息，请参见 Orchestrator 插件 API 参考。

查找器对象的角色

查找器对象识别并查找插件技术中受管对象类型的特定实例。Orchestrator 可以通过在查找器对象上运行工作流，修改其在插件技术中找到的对象并与之交互。

插件技术中给定受管对象类型的每个实例都必须具有唯一标识符，以便 Orchestrator 查找器对象可以找到它们。插件技术为对象实例提供字符串形式的唯一标识符。工作流在运行时，Orchestrator 会将其找到的对象的唯一标识符设置为工作流属性值。需要给定类型对象作为输入参数的工作流会在该对象类型的特定实例上运行。

插件添加到 Orchestrator JavaScript API 的查找器对象会将插件名称作为其名称前缀。例如，vCenter Server API 中的 `VirtualMachine` 受管对象类型会在 Orchestrator 中显示为 `VC:VirtualMachine` JavaScript 类型。

例如，Orchestrator 会实施将虚拟机 `id` 属性用作其唯一标识符的查找器对象，从而通过 vCenter Server 插件访问特定 `VC:VirtualMachine` 实例。您可以将此对象实例作为属性值传递到工作流元素。

Orchestrator 插件会将插件技术的对象映射到 `vso.xml` 文件 `<finder>` 元素中对应的 Orchestrator 查找器对象。`<finder>` 元素确定了插件技术中可获取对象特定实例唯一标识符的方法或函数。`<finder>` 元素还定义了对象之间的关系，从而按照对象之间的关联方式来查找对象。

查找器对象会显示在包含这些对象的插件下的 Orchestrator **清单**选项卡中。

脚本对象角色

脚本对象是插件技术中对象的 JavaScript 表现形式。插件中的脚本对象会显示在 Orchestrator JavaScript API 中，您可以将其用于工作流和操作中的脚本元素。

插件中的脚本对象在 Orchestrator JavaScript API 中显示为 JavaScript 模块、类型和类。大多数查找器对象具有脚本对象表现形式。JavaScript 类可将方法和属性添加到 Orchestrator JavaScript API，后者表示插件技术 API 中对象的方法和属性。插件技术提供了对象、类型、类、属性和独立于 Orchestrator 的方法的实现。例如，vCenter Server 插件将 vCenter Server API 中的所有对象表示为 Orchestrator JavaScript API 中的 JavaScript 对象，并以 JavaScript 的形式表示 vCenter Server API 定义的所有类、方法和属性。您可以使用 vCenter Server 脚本类以及其在 Orchestrator 脚本函数中定义的方法和属性。

例如，vCenter Server API 的 `VirtualMachine` 受管对象类型由 `VC:VirtualMachine` 查找器发现并在 Orchestrator JavaScript API 中显示为 `VcVirtualMachine` JavaScript 类。Orchestrator JavaScript API 中的 `VcVirtualMachine` JavaScript 类将所有相同方法和属性定义为 vCenter Server API 中的 `VirtualMachine` 受管对象。

Orchestrator 插件会将插件技术中的对象、类型、类、属性和方法映射到 `vso.xml` 文件 `<scripting-objects>` 元素中对应的 Orchestrator JavaScript 对象、类型、类、属性和方法。

事件处理程序的角色

事件即 Orchestrator 在插件技术中发现的对象的状态或属性的更改。Orchestrator 通过实施事件处理程序来监控事件。

Orchestrator 插件可让您通过多种方式监控插件技术中的事件。Orchestrator 插件 API 可让您创建以下类型的事件处理程序来监控插件技术中的事件。

侦听器	被动监控插件技术中对象的状态更改。插件技术或插件实现定义了侦听器监控的事件。侦听器不会启动事件，但会在事件发生时通知 Orchestrator。侦听器会通过轮询插件技术或接收插件技术的通知来检测事件。事件发生后，等待事件的 Orchestrator 策略或工作流可通过启动 Orchestrator 服务器中的操作来作出响应。侦听器组件为可选。
策略	监控插件技术中的特定事件并在事件发生时启动 Orchestrator 服务器中的操作。策略可以监控策略触发器和策略计量器。策略触发器定义了插件技术中的事件，即在事件发生后，使正在运行的策略启动 Orchestrator 服务器中的操作（例如运行工作流）。策略计量器定义了插件技术中某个对象属性的值范围，即超出该范围后，Orchestrator 会启动操作。策略为可选。
工作流触发器	如果正在运行的工作流包含“等待事件”元素，则当其到达该元素时，会挂起运行并等待插件技术中发生事件。工作流触发器定义了工作流中“等待事件”元素所等待的插件技术中事件。您可以向观察程序注册工作流触发器。工作流触发器为可选。
观察程序	代表工作流中的“等待事件”元素，观察插件技术中特定事件的工作流触发器。事件发生后，观察程序会通知等待该事件的任何工作流。观察程序为可选。

插件的内容和结构

Orchestrator 插件必须包含一组标准组件并遵守标准文件结构。对于符合标准文件结构的插件，必须包含特定文件夹和文件。

若要创建 Orchestrator 插件，您需要定义 Orchestrator 如何访问并与插件技术中的对象交互。并且，您需要将插件技术的所有对象和函数映射到 `vso.xml` 文件中对应的 Orchestrator 对象和函数。

`vso.xml` 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。插件在插件技术中发现的每个对象都必须具有您提供的唯一标识符。您需要在 `vso.xml` 文件内的 `finder` 元素以及对象元素中定义对象名称。

插件可以交付为标准 Java 存档文档 (JAR) 或 ZIP 文件，但无论哪种交付方式，文件名都必须使用 `.dar` 扩展名进行重命名。

注 您可以使用 Orchestrator 控制中心将 DAR 文件导入 Orchestrator 服务器。

- 定义 vso.xml 文件中的应用程序映射

  `vso.xml` 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或是在 Orchestrator **清单**选项卡中显示为查找器对象。

- vso.xml 插件定义文件的格式

  `vso.xml` 文件用于定义 Orchestrator 服务器与插件技术的交互方式。`vso.xml` 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。

- 命名插件对象

  您必须为该插件在插件技术中发现的每个对象提供唯一标识符。您需要分别在 `vso.xml` 文件中的 `<finder>` 元素和 `<object>` 元素中定义对象名称。

- 插件对象命名约定

  在命名插件中的所有对象时，必须遵守 Java 类命名约定。

- 插件的文件结构

  插件必须遵守标准文件结构，并且必须包含某些特定文件夹和文件。作为标准 Java 存档 (JAR) 或 ZIP 文件交付的插件，必须使用 `.dar` 扩展名对其重命名。

定义 vso.xml 文件中的应用程序映射

`vso.xml` 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或是在 Orchestrator **清单**选项卡中显示为查找器对象。

`vso.xml` 文件向 Orchestrator 服务器提供以下信息：

- 插件的版本、名称和说明
- 插件技术的类和关联插件适配器的参考
- 在启动 Orchestrator 服务器时初始化插件
- 脚本类型，用来在插件技术中表示对象类型

- 对象类型之间的关系，用于定义对象在 Orchestrator 清单中的显示方式
- 脚本类，将插件技术中的对象和操作映射到 Orchestrator JavaScript API 中的函数和对象类型
- 枚举，定义适用于特定类型的所有对象的常量值列表
- Orchestrator 在插件技术中监视的事件

`vso.xml` 文件必须遵守 Orchestrator 插件的 XML 架构定义。您可以在 VMware 支持站点上访问架构定义。

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

关于 `vso.xml` 文件所有元素的说明，请参见 vso.xml 插件定义文件的元素。

vso.xml 插件定义文件的格式

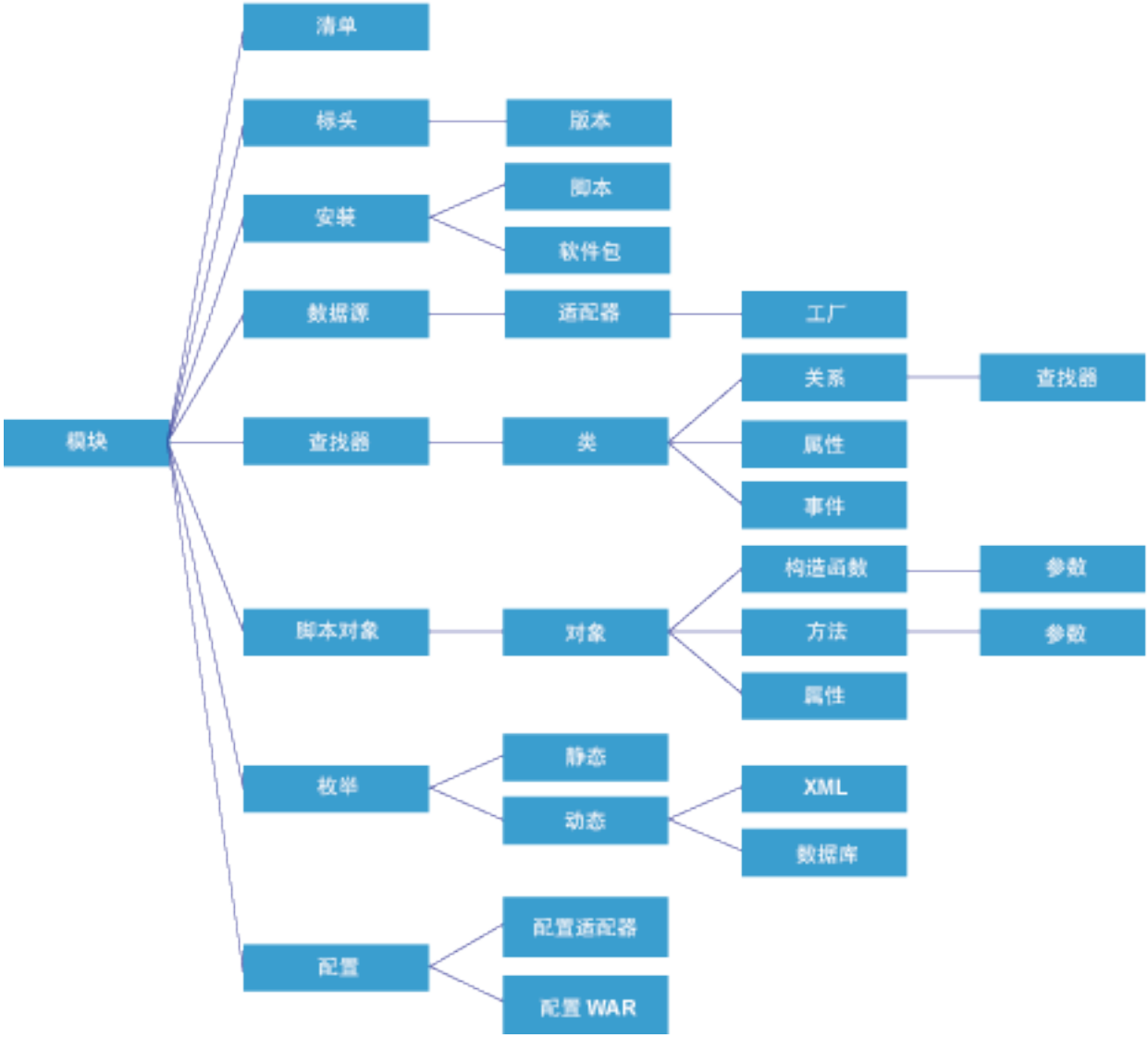
`vso.xml` 文件用于定义 Orchestrator 服务器与插件技术的交互方式。`vso.xml` 文件必须包含要公开到 Orchestrator 的每类对象或操作的引用。

`vso.xml` 文件中包含的对象在 Orchestrator 脚本 API 中会显示为脚本对象，或在 Orchestrator **清单**选项卡中显示为查找器对象。

作为插件开放式架构和标准化实现的一部分，`vso.xml` 文件必须使用标准格式。

以下图标显示了 `vso.xml` 插件定义文件的格式以及各要素之间相互嵌套的方式。

图 6-2. vso.xml 插件定义文件的格式



命名插件对象

您必须为该插件在插件技术中发现的每个对象提供唯一标识符。您需要分别在 `vso.xml` 文件中的 `<finder>` 元素和 `<object>` 元素中定义对象名称。

您在工厂实现中定义的查找器操作可以查找插件技术中的对象。在插件找到对象后，您可以在 Orchestrator 工作流中使用并在工作流之间传递这些对象。您为这些对象提供的唯一标识符可以让对象在工作流的不同元素之间进行传递。

Orchestrator 服务器仅存储其处理的每个对象的类型和标识符，并不存储 Orchestrator 获取此对象的方式和位置信息。您在插件实现中命名对象时必须保持一致，方便跟踪从插件中获得的对象。

如果 Orchestrator 服务器在工作流运行过程中停止，则在服务器重启时，工作流会从服务器停止时所运行的工作流元素处恢复运行。工作流会使用标识符来检索服务器停止时正在处理的元素。

插件对象命名约定

在命名插件中的所有对象时，必须遵守 Java 类命名约定。

重要事项 鉴于工作流引擎执行数据序列化的方式，在对象名称中不要使用以下字符串序列。在对象标识符中使用以下字符序列会导致工作流引擎错误地解析工作流，这样会造成在运行工作流时发生意外行为。

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

在命名插件中的对象时，使用以下准则。

- 名称中每个词的首字母使用大写字母。
- 不要使用空格来分隔字词。
- 对于字母，仅使用标准字符 `A` 到 `Z` 和 `a` 到 `z`。
- 不要使用特殊字符，例如重音符。
- 不要使用数字作为名称的首字符。
- 尽可能少于 10 个字符。

表 6-1. 插件对象命名规则 显示了应用于各个对象类型的规则。

表 6-1. 插件对象命名规则

对象类型	命名规则
插件	■ 在 `vso.xml` 文件的 `<module>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 必须唯一。您无法在 Orchestrator 服务器中运行两个同名的插件。
查找器对象	■ 在 `vso.xml` 文件的 `<finder>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 在插件中必须唯一。 Orchestrator 会将插件名称和冒号添加到 Orchestrator 脚本 API 的查找器对象类型中的查找器对象名称。例如，vCenter Server 插件中的 `VirtualMachine` 对象类型在 Orchestrator 脚本 API 中显示为 `VC:VirtualMachine`。
脚本对象	■ 在 `vso.xml` 文件的 `<scripting-object>` 元素中定义。 ■ 必须遵守 Java 类命名约定。 ■ 在 Orchestrator 服务器中必须唯一。 ■ 为避免混淆脚本对象与同名查找器对象或其他插件中的脚本对象，请始终在脚本对象名称前加上插件名称作为前缀，但不要加冒号。例如，vCenter Server 插件中的 `VirtualMachine` 类在 Orchestrator 脚本 API 中显示为 `VcVirtualMachine` 类。

插件的文件结构

插件必须遵守标准文件结构，并且必须包含某些特定文件夹和文件。作为标准 Java 存档 (JAR) 或 ZIP 文件交付的插件，必须使用 `.dar` 扩展名对其重命名。

DAR 存档的内容必须使用以下文件夹结构和命名约定。

表 6-2. DAR 存档结构

文件夹	说明
plug-in_name\VSO-INF\	包含 `vso.xml` 文件，用于定义将插件技术中的对象映射到 Orchestrator 对象。 `VSO-INF` 文件夹和 `vso.xml` 文件必不可少。
plug-in_name\lib\	包含 JAR 文件，其中包含插件技术的二进制文件。包含的其他 JAR 文件，其中应包含适配器实现、工厂、通知处理程序以及插件中的其他接口。 `lib` 文件夹和 JAR 文件必不可少。
plug-in_name\resources\	包含插件所需的资源文件。`resources` 文件夹可以包含以下类型的元素： ■ 图像文件，用来在 Orchestrator **清单**选项卡中表示插件的对象。 ■ 脚本，用来定义插件启动时的初始化行为。 ■ Orchestrator 软件包，可包含自定义工作流、操作和其他资源，能与通过插件访问的对象进行交互。 您可以将资源整理到子文件夹中。例如：`resources\images\`、`resources\scripts\` 或 `resources\packages\`。 `resources` 文件夹为可选。

您可以使用 Orchestrator 控制中心将 DAR 文件导入到 Orchestrator 服务器中。

Orchestrator 插件 API 参考

Orchestrator 插件 API 用于定义您在开发 `IPluginAdaptor` 和 `IPluginFactory` 实施以创建插件时需要实现和扩展的 Java 接口和类。

所有类均包含在 `ch.dunes.vso.sdk.api` 软件包中，另有说明的除外。

IAop 接口

`IAop` 接口提供的方法可获取并设置插件技术中对象上的属性。

```
public interface IAop
```

`IAop` 接口定义了以下方法：

方法	返回	描述
`get(java.lang.String propertyName, java.lang.Object object, java.lang.Object sdkObject)`	`java.lang.Object`	获取插件中给定对象的属性。
`set(java.lang.String propertyName, java.lang.String propertyValue, java.lang.Object object)`	空	设置插件中给定对象上的属性。

IDynamicFinder 界面

`IDynamicFinder` 界面会以编程方式返回查找器的 ID 和属性，而无需在 `vso.xml` 文件中定义这些 ID 和属性。

`IDynamicFinder` 接口定义了以下方法。

方法	返回	说明
`getIdAccessor(java.lang.String type)`	`java.lang.String`	提供 OGNL 表达式，从而以编程方式获取对象 ID。
`getProperties(java.lang.String type)`	`java.util.List<SDKFinderProperty>`	以编程方式提供多种对象属性。

IPluginAdaptor 接口

您可实现 `IPluginAdaptor` 接口以管理插件工厂、事件和观察程序。`IPluginAdaptor` 接口用于定义插件和 Orchestrator 服务器之间的适配器。

`IPluginAdaptor` 实例负责会话管理。`IPluginAdaptor` 接口定义了以下方法。

方法	返回	描述
`addWatcher(PluginWatcher watcher)`	空	添加观察程序从而监视特定事件。
`createPluginFactory(java.lang.String sessionID, java.lang.String username, java.lang.String password, IPluginNotificationHandler notificationHandler)`	`IPluginFactory`	创建 `IPluginFactory` 实例。Orchestrator 服务器使用此工厂从插件技术获取对象（按对象 ID 或按对象与其他对象的关系等）。 会话 ID 可让您识别运行的会话。例如，用户可同时登录两个不同的 Orchestrator 客户端并运行两个会话。 同样，启动工作流会创建一个独立会话，独立于该工作流启动时所在的客户端。即使关闭该 Orchestrator 客户端，工作流也会继续运行。
`installLicenses(PluginLicense[] licenses)`	空	为 VMware 提供的插件安装许可证信息
`registerEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)`	空	对清单中的元素设置触发器和计量器
`removeWatcher(java.lang.String watcherId)`	空	移除观察程序

方法	返回	描述
setPluginName(java.lang.String pluginName)	空	从 vso.xml 文件获取插件名称
setPluginPublisher(IPluginPublisher pluginPublisher)	空	设置插件的发布者
uninstallPluginFactory(IPluginFactory plugin)	空	卸载插件工厂。
unregisterEventPublisher(java.lang.String type, java.lang.String id, IPluginEventPublisher publisher)	空	从清单中的元素移除触发器和计量器

IPluginEventPublisher 接口

IPluginEventPublisher 接口用于在事件通知总线上对要监视的 Orchestrator 策略发布触发器和计量器。

您可以直接在插件适配器实现中创建 IPluginEventPublisher 实例，或者可以在不同的事件生成器类中创建。

您可以实现 IPluginEventPublisher 接口将插件技术中的事件发布到 Orchestrator 策略引擎。您需要创建方法来设置策略触发器和插件技术中对象的计量器，以及事件侦听器来侦听这些对象的事件。

策略可实现计量器或触发器来监控插件技术中的对象。策略计量器会监控对象的属性并且当对象的值超出特定限制时在 Orchestrator 服务器中推送事件。策略触发器会监控对象并且当对象上发生定义的事件时在 Orchestrator 服务器中推送事件。您可以向 IPluginEventPublisher 实例注册策略计量器和触发器，以便 Orchestrator 策略可对其进行监控。

IPluginEventPublisher 接口定义了以下方法。

类型	返回	说明
pushGauge(java.lang.String type, java.lang.String id, java.lang.String gaugeName, java.lang.String deviceName, java.lang.Double gaugeValue)	空	针对要监视的策略发布计量器。采用以下参数： ■ type：要监视的对象类型。 ■ id：要监视的对象标识符。 ■ gaugeName：该计量器的名称。 ■ deviceName：该计量器监视的属性类型的名称。 ■ gaugeValue：该计量器监视的对象值。
pushTrigger(java.lang.String type, java.lang.String id, java.lang.String triggerName, java.util.Properties additionalProperties)	空	针对要监视的策略发布触发器。采用以下参数： ■ type：要监视的对象类型。 ■ id：要监视的对象标识符。 ■ triggerName：该触发器的名称。 ■ additionalProperties：需要该触发器监视的任何其他属性。

IPluginFactory 接口

IPluginAdaptor 会返回 IPluginFactory 实例。IPluginFactory 实例可在插件应用程序中运行命令，并查找要执行 Orchestrator 操作的对象。

IPluginFactory 接口定义了以下字段：

static final java.lang.String RELATION_CHILDREN

IPluginFactory 接口定义了以下方法。

方法	返回	说明
executePluginCommand(java.lang.String cmd)	空	使用此插件运行命令。VMware 建议不要使用此方法。
find(java.lang.String type, java.lang.String id)	java.lang.Object	使用此插件查找对象。按对象 ID 和类型识别对象。
findAll(java.lang.String type, java.lang.String query)	QueryResult	使用此插件查找与查询字符串匹配的特定类型的对象。在插件的 IPluginFactory 实现中定义查询语法。如果不定义查询语法，则 findAll() 会返回指定类型的所有对象。
findRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	java.util.List	确定对象是否具有子项。
hasChildrenInRelation(java.lang.String parentType, java.lang.String parentId, java.lang.String relationName)	HasChildrenResult	按特定关系查找与给定父项相关的所有子项。
invalidate(java.lang.String type, java.lang.String id)	空	按类型和 ID 使对象无效。
void invalidateAll()	空	使缓存中的所有对象无效。

IPluginNotificationHandler 接口

IPluginNotificationHandler 用于定义针对在 Orchestrator 通过插件访问的对象上所发生的不同类型事件而向 Orchestrator 发出通知的方法。

IPluginNotificationHandler 接口定义了以下方法。

方法	返回	说明
getSessionID()	java.lang.String	返回当前会话 ID。
notifyElementDeleted(java.lang.String type, java.lang.String id)	空	通知系统具有给定类型和 ID 的对象已被删除。

方法	返回	说明
`notifyElementInvalidate(java.lang.String type, java.lang.String id)`	空	通知系统某个对象的关系已被更改。您可以使用 `notifyElementInvalidate()` 方法通知 Orchestrator 有关对象之间关系的所有变更情况，而不仅仅是使对象无效的关系变更。例如，向父项添加子对象即表示这两个对象之间关系发生了变更。
`notifyElementUpdated(java.lang.String type, java.lang.String id)`	空	通知系统某个对象的属性已被更改。
`notifyMessage(ch.dunes.vso.sdk.api.ErrorLevel severity, java.lang.String type, java.lang.String id, java.lang.String message)`	空	发布与当前模块相关的错误消息

IPluginPublisher 接口

`IPluginPublisher` 接口用于在事件通知总线上针对“等待事件”元素要监视的长时间运行工作流发布观察程序事件。

当工作流触发器在插件技术中启动事件后，负责观察该触发器（已向 `IPluginPublisher` 实例注册）的插件观察程序会将事件已发生的消息通知任何正在等待的工作流。

`IPluginPublisher` 接口定义了以下方法。

类型	值	描述
`pushWatcherEvent(java.lang.String id, java.util.Properties properties)`	空	在事件通知总线上发布观察程序事件

WebConfigurationAdaptor 接口

`WebConfigurationAdaptor` 接口实现了 `IConfigurationAdaptor`，还定义了查找并在插件配置选项卡中安装 Web 应用程序的方法。

注　从 Orchestrator 4.1 开始，`WebConfigurationAdaptor` 接口已被弃用。若要将 Web 应用程序添加到配置，请实现 `IConfigurationAdaptor` 并使用 `vso.xml` 文件中的 `configuration-war` 属性来识别 Web 应用程序。

`WebConfigurationAdaptor` 接口定义了以下方法。

方法	返回	描述
`getWebAppContext()`	字符串	找到配置选项卡的 Web 应用程序的 WAR 文件。以字符串形式提供 DAR 文件中 `/webapps` 目录内 WAR 文件的名称和路径。
`setWebConfiguration(boolean webConfiguration)`	布尔	确定配置选项卡的内容是否由 Web 应用程序定义。

PluginTrigger 类

PluginTrigger 类会创建一个触发器模块，代表工作流中的“等待事件”元素，用于获取要在插件技术中监视的对象和事件的相关信息。

PluginTrigger 类定义了获取或设置要监控对象的类型和名称的方法、事件的特性和超时时间段。

您可以创建专门供工作流中“等待事件”元素使用的 PluginTrigger 类实现。您可以按定义了事件和实现了 IPluginEventPublisher.pushTrigger() 方法的类定义 Orchestrator 策略的策略触发器。

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

PluginTrigger 类定义了以下方法：

方法	返回	说明
getModuleName()	java.lang.String	获取触发器模块的名称。
getProperties()	java.util.Properties	获取触发器的属性列表。
getSdkId()	java.lang.String	获取要在插件技术中监视的对象 ID。
getSdkType()	java.lang.String	获取要在插件技术中监视的对象类型。
getTimeout()	长	获取触发器超时时段。
setModuleName(java.lang.String moduleName)	空	设置触发器模块的名称。
setProperties(java.util.Properties properties)	空	设置触发器的属性列表。
setSdkId(java.lang.String sdkId)	空	设置要在插件技术中监视的对象 ID。
setSdkType(java.lang.String sdkType)	空	设置要在插件技术中监视的对象类型。
setTimeout(long timeout)	空	设置超时时段（以秒为单位）。负值将取消激活超时。

构造函数

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

PluginWatcher 类

PluginWatcher 类会代表工作流中的“等待事件”元素来观察插件技术中已定义事件的触发器模块。

PluginWatcher 类定义了您可用来创建插件观察程序实例的构造函数。PluginWatcher 类定义了获取或设置要观察工作流触发器名称的方法，以及超时时间段。

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

PluginWatcher 类定义了以下方法：

方法	返回	说明
getId()	java.lang.String	获取触发器 ID
getModuleName()	java.lang.String	获取触发器模块的名称
getTimeoutDate()	长	获取触发器超时日期
getTrigger()	空	获取触发器
setId(java.lang.String id)	空	设置触发器 ID
setTimeoutDate()	空	设置触发器超时日期

构造函数

PluginWatcher(PluginTrigger trigger)

QueryResult 类

QueryResult 类包含对 Orchestrator 通过插件所访问对象进行 find 查询时的结果。

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

如果已找到结果的总数超过查询返回的结果数，totalCount 值可以大于 QueryResult 返回的元素数量。查询返回的结果数应在 vso.xml 文件中定义。

QueryResult 类定义了以下方法：

方法	返回	说明
addElement(java.lang.Object element)	空	向 QueryResult 添加元素
addElements(java.util.List elements)	空	向 QueryResult 添加元素列表
getElements()	java.util.List	从插件应用程序中获取元素
getTotalCount()	长	获取插件技术中所有可用元素的数量
isPartialResult()	布尔	确定已获取的结果是否完整
removeElement(java.lang.Object element)	空	从插件技术中移除元素
setElements(java.util.List elements)	空	设置插件技术中的元素
setTotalCount(long totalCount)	空	设置插件技术中可用元素的总数

构造函数

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

SDKFinderProperty 类

SDKFinderProperty 类定义用于从相关对象中获取并设置其属性的方法，这些对象是由 Orchestrator 查找器对象在插件技术中发现的。IDynanmicFinder.getProperties 方法会返回 SDKFinderProperty 对象。

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

SDKFinderProperty 类定义了以下方法：

方法	返回	说明
getAttributeName()	java.lang.String	获取对象属性名称
getBeanProperty()	java.lang.String	获取 Java Bean 的属性
getDescription()	java.lang.String	获取对象说明
getDisplayName()	java.lang.String	获取对象显示名称
getPossibleResultType()	java.lang.String	获取查找器可能返回的结果类型
getPropertyAccessor()	java.lang.String	获取对象属性访问器
getPropertyAccessorTree()	java.lang.Object	获取对象属性访问器树
isHidden()	布尔	显示或隐藏对象
isShowInColumn()	布尔	在数据库列中显示或隐藏对象
isShowInDescription()	布尔	显示或隐藏对象说明
setAttributeName(java.lang.String attributeName)	空	设置对象属性名称
setBeanProperty(java.lang.String beanProperty)	空	设置 Java Bean 中的属性
setDescription(java.lang.String description)	空	设置对象说明
setDisplayName(java.lang.String displayName)	空	设置对象显示名称
setHidden(boolean hidden)	空	显示或隐藏对象
setPossibleResultType(java.lang.String possibleResultType)	空	设置查找器可能返回的结果类型
setPropertyAccessor(java.lang.String propertyAccessor)	空	设置对象属性访问器
setPropertyAccessorTree(java.lang.Object propertyAccessorTree)	空	设置对象属性访问器树
setShowInColumn(boolean showInTable)	空	在数据库列中显示或隐藏对象
setShowInDescription(boolean showInDescription)	空	显示或隐藏对象说明

构造函数

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

PluginExecutionException 类

如果插件在运行操作时遇到异常，PluginExecutionException 类会返回一条错误消息。

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

PluginExecutionException 类从 class java.lang.Throwable 继承以下方法：

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toStringfillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace

构造函数

PluginExecutionException(java.lang.String message)

PluginOperationException 类

PluginOperationException 类用于处理插件在操作过程中遇到的错误。

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

PluginOperationException 类从 class java.lang.Throwable 继承以下方法：

fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

构造函数

PluginOperationException(java.lang.String message)

HasChildrenResult 枚举

HasChildrenResult 枚举用于声明给定父项中是否具有子项。IPluginFactory.hasChildrenInRelation 方法会返回 HasChildrenResult 对象。

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

HasChildrenResult 枚举定义以下常量：

- public static final HasChildrenResult Yes

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

`HasChildrenResult` 枚举定义了以下方法：

方法	返回	说明
`getValue()`	int	返回以下其中一个值： **1** 具有子项的父项 **-1** 不具子项的父项 **0** 未知或无效参数
`valueOf(java.lang.String name)`	`static HasChildrenResult`	返回此类型一个指定名称的枚举常量。该字符串必须匹配用于声明此类型枚举常量的标识符。请勿在枚举名称中使用空白字符。
`values()`	`static HasChildrenResult[]`	按声明顺序返回一组包含此枚举类型的常量。此方法可以按以下方式对这些常量进行迭代： `for (HasChildrenResult c : HasChildrenResult.values()) System.out.println(c);`

`HasChildrenResult` 枚举会继承以下来自 `class java.lang.Enum` 的方法：

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

ScriptingAttribute 注释类型

`ScriptingAttribute` 注释类型可对插件技术中某个对象的属性添加注释，将其作为属性在脚本中使用。

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

`ScriptingAttribute` 注释类型具有以下值：

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

ScriptingFunction 注释类型

`ScriptingFunction` 注释类型可对某个方法添加注释，将其作为属性在脚本中使用。

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

ScriptingFunction 注释类型具有以下值：

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

ScriptingParameter 注释类型

ScriptingParameter 注释类型可对某个参数添加注释，将其作为属性在脚本中使用。

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

ScriptingParameter 注释类型具有以下值：

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

vso.xml 插件定义文件的元素

vso.xml 文件包含一组标准元素。有些元素是必选项，有些则是可选项。每个元素包含多种属性，用于定义映射到 Orchestrator 对象和操作的相关对象和操作的值。

此外，元素可拥有零个或多个子元素。子元素用于进一步定义父元素。相同的子元素可以出现在多个父元素中。例如，description 元素没有子元素，但可以显示为多个父元素的子元素，例如 module、example、trigger、gauge、finder、constructor、method、object 和 enumeration。

其后的每个元素定义均会列出其属性、父项和子项。

module 元素

模块是提供给 Orchestrator 使用的一组插件对象。

模块中的信息包括：插件技术中的数据向 Java 类映射的方式、版本控制、模块部署方式以及插件在 Orchestrator 清单中的显示方式。

<module> 元素为可选。<module> 元素具有以下属性：

属性	值	描述
name	字符串	定义插件中所有<finder>元素的类型。此为必需属性。
version	编号	插件版本号，在新版本插件中重新加载安装包时会用到。此为必需属性。
build-number	编号	插件内部版本号，在新版本插件中重新加载安装包时会用到。此为必需属性。
image	图像文件	Orchestrator 清单中显示的图标。此为必需属性。

属性	值	描述
display-name	字符串	Orchestrator 清单中显示的名称。此为可选属性。
interface-mapping-allowed	true 或 false	Vmware 不建议进行接口映射。此为可选属性。

表 6-3. 元素层次结构

父元素	子元素
无	■ <description> ■ <installation> ■ <configuration> ■ <finder-datasources> ■ <inventory> ■ <finders> ■ <scripting-objects> ■ <enumerations>

description 元素

<description> 元素提供了 API Explorer 文档中所显示插件的元素的说明。

您可在 <description> 和 </description> 标记之间添加在 API Explorer 文档中显示的文本。

<description> 元素为可选。<description> 元素不包含属性。

表 6-4. 元素层次结构

父元素	子元素
■ <module> ■ <example> ■ <trigger> ■ <gauge> ■ <finder> ■ <constructor> ■ <method> ■ <object> ■ <enumeration>	无

deprecated 元素

<deprecated> 元素标记了 API Explorer 文档中已经弃用的对象和方法。

您可在 <deprecated> 和 </deprecated> 标记之间添加在 API Explorer 文档中显示的文本。

<deprecated> 元素为可选。<deprecated> 元素不包含属性。

表 6-5. 元素层次结构

父元素	子元素
■ <method> ■ <object>	无

url 元素

<url> 元素可提供指向与对象或枚举相关的外部文档的 URL。

您需要在 <url> 和 </url> 标记之间提供 URL。

<url> 元素为可选。<url> 元素不包含属性。

表 6-6. 元素层次结构

父元素	子元素
■ <enumeration> ■ <object>	无

installation 元素

<installation> 元素可让您在服务器启动时安装软件包或运行脚本。

<installation> 元素为可选。<installation> 元素具有以下属性：

属性	值	描述
mode	always、never 或 version	在 Orchestrator 服务器启动时设置 mode 值会导致以下行为： ■ 操作 always 运行。 ■ 操作 never 运行。 ■ 操作会在服务器检测到较新版本插件时运行。 此为必需属性。

表 6-7. 元素层次结构

父元素	子元素
<module>	<action>

action 元素

<action> 元素用于指定在 Orchestrator 服务器启动时运行的操作。

<action> 元素属性会提供指向 Orchestrator 软件包或特定脚本的路径，该脚本定义了插件启动时的行为。

<action> 元素为可选。插件可以拥有不限数量的 <action> 元素。<action> 元素具有以下属性。

属性	值	描述
resource	字符串	指向 dar 文件根目录下 Java 软件包或脚本的路径。此为必需属性。
type	install-package 或 execute-script	在 Orchestrator 服务器中安装指定的 Orchestrator 软件包，或运行指定脚本。此为必需属性。

表 6-8. 元素层次结构

父元素	子元素
<installation>	无

finder-datasources 元素

<finder-datasources> 元素是 <finder-datasource> 元素的容器。

<finder-datasources> 元素为可选。<finder-datasources> 元素不包含属性。

表 6-9. 元素层次结构

父元素	子元素
<module>	<finder-datasource>

finder-datasource 元素

<finder-datasource> 元素指向为插件创建的 IPluginAdaptor 实现的 Java 类文件。

您需要设置 Orchestrator 如何访问 <finder-datasource> 元素中插件技术的对象。<finder-datasource> 元素标识了您创建的插件适配器的 Java 类。插件适配器类将您创建的插件工厂实例化。插件工厂定义了插件技术中查找对象的方法。您可以为工厂执行的查找器方法调用的 <finder-datasource> 元素设置超时。不同超时会应用到 IPluginFactory 接口中的不同查找器方法。

<finder-datasource> 元素为可选。插件可以拥有不限数量的 <finder-datasources> 元素。<finder-datasource> 元素具有以下属性。

属性	值	描述
name	字符串	标识了 <finder> 元素 datasource 属性中的数据源。相当于 XML id。此为必需属性。
adaptor-class	Java 类	指向您定义的用来创建插件适配器的 IPluginAdaptor 实现，例如，com.vmware.plugins.sample.Adaptor。此为必需属性。
concurrent-call	true（默认）或 false	允许多个用户同时访问适配器。如果插件不支持并发调用，您必须将 concurrent-call 设置为 false。此为可选属性。

属性	值	描述
invoker-mode	direct（默认）或 timeout	设置查找器函数的超时。如果设置为 direct，则查找器函数的调用永不会超时。如果设置为 timeout，则 Orchestrator 服务器会应用与查找器方法对应的超时时间段。此为可选属性。
anonymous-login-mode	never（默认）或 always	传递或不传递用户的用户名和密码至插件。此为可选属性。
timeout-fetch-relation	数字：默认为 30 秒	应用于来自 findRelation() 的调用。此为可选属性。
timeout-find-all	数字：默认为 60 秒	应用于来自 findAll() 的调用。此为可选属性。
timeout-find	数字：默认为 60 秒	应用于来自 find() 的调用。此为可选属性。
timeout-has-children-in-relation	数字：默认为 2 秒	应用于来自 findChildrenInRelation() 的调用。此为可选属性。
timeout-execute-plugin-command	数字：默认为 30 秒	应用于来自 executePluginCommand() 的调用。此为可选属性。

表 6-10. 元素层次结构

父元素	子元素
<finder-datasources>	无

inventory 元素

<inventory> 元素用于对在 Orchestrator 客户端**清单**视图和对象选择对话框中显示的插件定义其层次结构列表的根。

<inventory> 元素并不表示插件应用程序中的对象，而是表示将插件本身作为 Orchestrator 脚本 API 中的一个对象。

<inventory> 元素为可选。<inventory> 元素具有以下属性。

属性	值	描述
type	Orchestrator 对象类型	<finder> 元素类型，表示对象层次结构的根。此为必需属性。

表 6-11. 元素层次结构

父元素	子元素
<module>	无

finders 元素

<finders> 元素是所有 <finder> 元素的容器。

<finders> 元素为可选。<finders> 元素不包含属性。

表 6-12. 元素层次结构

父元素	子元素
<module>	<finder>

finder 元素

<finder> 元素在 Orchestrator 客户端中代表一种通过该插件发现的对象类型。

<finder> 元素能够识别用于定义由对象查找器表示的对象的 Java 类。<finder> 元素用于定义该对象在 Orchestrator 客户端界面中的显示方式，同时还能够识别由 Orchestrator 脚本 API 所定义的用于表示这一对象的脚本对象。

查找器可以充当不同类型插件技术所用对象格式之间的接口。

<finder> 元素为可选。插件可以拥有不限数量的 <finder> 元素。<finder> 元素定义以下属性：

属性	值	描述
type	Orchestrator 对象类型	查找器表示的对象类型。此为必需属性。
datasource	<finder-datasource name> 属性	识别使用数据源 refid 定义的对象的 Java 类。此为必需属性。
dynamic-finder	Java 方法	用于定义您在 IDynamicFinder 实例中实现的自定义查找器方法，从而以编程方式返回查找器的 ID 和属性，无需在 vso.xml 文件中进行定义。此为可选属性。
hidden	true 或 false（默认值）	如果为 true，则会在 Orchestrator 客户端隐藏该查找器。此为可选属性。
image	图形文件的路径	16x16 图标，表示 Orchestrator 客户端层次结构列表中的查找器。此为可选属性。
java-class	Java 类的名称	用于定义由查找器发现并映射到脚本对象的对象的 Java 类。此为可选属性。
script-object	<scripting-object type> 属性	该查找器要映射到的 <scripting-object> 类（如有）。此为可选属性。

表 6-13. 元素层次结构

父元素	子元素
<finders>	■ <id> ■ <description> ■ <properties> ■ <default-sorting> ■ <inventory-children> ■ <relations> ■ <inventory-tabs> ■ <events>

properties 元素

<properties> 元素是 <finder><property> 元素的容器。

<properties> 元素为可选。<properties> 元素不包含属性。

表 6-14. 元素层次结构

父元素	子元素
<finder>	<property>

properties 元素

<property> 元素会将已发现对象的属性映射到 Java 属性或方法调用。

您可以在实现插件工厂时，调用 SDKFinderProperty 类的方法以获取要处理的插件工厂实现的属性。

您可以在 Orchestrator 客户端的各视图中显示或隐藏对象属性。你也可以使用枚举来定义相关对象属性。

<property> 元素为可选。插件可以拥有不限数量的 <property> 元素。<property> 元素具有以下属性。

属性	值	描述
name	查找器名称	FinderResult 用来存储元素的查找器名称。此为必需属性。
display-name	查找器名称	显示的属性名称此为可选属性。
bean-property	属性名称	您可使用 bean-property 属性来识别要使用 get 和 set 操作获取的属性。如果识别出名为 MyProperty 的属性，插件会定义 getMyProperty 和 setMyProperty 操作。 您可以设置 bean-property 或 property-accessor 的其中之一，但不能同时设置两个。此为可选属性。
property-accessor	获取对象属性值时所用的方法	property-accessor 属性允许您定义 OGNL 表达式来验证对象的属性。 您可以设置 bean-property 或 property-accessor 的其中之一，但不能同时设置两个。此为可选属性。
show-in-column	true（默认）或 false	如果为 true，该属性会显示在 Orchestrator 客户端的示结果表中。此为可选属性。
show-in-description	true（默认）或 false	如果为 true，该属性会显示在对象说明中。此为可选属性。
hidden	true 或 false（默认值）	如果为 true，该属性会在所有情况下隐藏。此为可选属性。
linked-enumeration	枚举名称	将查找器属性与枚举进行链接。此为可选属性。

表 6-15. 元素层次结构

父元素	子元素
<properties>	子元素

relations 元素

<relations> 元素是 <finder><relation> 元素的容器。

<relations> 元素为可选。<relations> 元素不包含属性。

表 6-16. 元素层次结构

父元素	子元素
<finder>	<relation>

relation 元素

<relation> 元素用于定义对象与其他对象之间的相关性。

您可以在 <relation> 元素中定义关系名称。

<relation> 元素为可选。插件可以拥有不限数量的 <relation> 元素。<relation> 元素具有以下属性。

属性	值	描述
name	关系名称	该关系的名称。此为必需属性。
type	Orchestrator 对象类型	通过此关系与其他对象相关的对象的类型。此为必需属性。
cardinality	to-one 或 to-many	定义对象之间的关系是一对一还是一对多。此为可选属性。

表 6-17. 元素层次结构

父元素	子元素
<relations>	无

id 元素

<id> 元素用于定义获取查找器所识别对象唯一 ID 的方法。

<id> 元素为可选。<id> 元素具有以下属性。

属性	值	描述
accessor	方法名称	accessor 属性允许您定义 OGNL 表达式来验证对象的特性。此为必需属性。

表 6-18. 元素层次结构

父元素	子元素
<finder>	无

inventory-children 元素

<inventory-children> 元素用于定义一个层次结构列表，显示在 Orchestrator 客户端**清单**视图和对象选择框中的对象。

<inventory-children> 元素为可选。<inventory-children> 元素不包含属性。

表 6-19. 元素层次结构

父元素	子元素
<finder>	<relation-link>

relation-link 元素

<relation-link> 元素用于定义**清单**选项卡中父对象和子对象之间的层次结构。

<relation-link> 元素为可选。插件可以拥有不限数量的 <relation-link> 元素。<relation-link> 元素具有以下属性。

类型	值	描述
name	关系名称	关系名称的 refid。此为必需属性。

表 6-20. 元素层次结构

父元素	子元素
<inventory-children>	无

events 元素

<events> 元素是 <trigger> 和 <gauge> 元素的容器。

<events> 元素可以包含不限数量的触发器和计量器。

<events> 元素为可选。<events> 元素不包含属性。

表 6-21. 元素层次结构

父元素	子元素
<finder>	■ <trigger> ■ <gauge>

trigger 元素

<trigger> 元素声明了您可用于该查找器的触发器。您必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法来设置触发器。

<trigger> 元素为可选。<trigger> 元素具有以下属性。

类型	值	描述
name	触发器名称	该触发器的名称。此为必需属性。

表 6-22. 元素层次结构

父元素	子元素
<events>	■ <description> ■ <trigger-properties>

trigger-properties 元素

<trigger-properties> 元素是 <trigger-property> 元素的容器。

<trigger-properties> 元素为可选。<trigger-properties> 元素不包含属性。

表 6-23. 元素层次结构

父元素	子元素
<trigger>	<trigger-property>

trigger-property 元素

<trigger-property> 元素用于定义可识别触发器对象的属性。

<trigger-property> 元素为可选。插件可以拥有不限数量的 <trigger-property> 元素。<trigger-property> 元素具有以下属性。

类型	值	描述
name	触发器名称	触发器的名称。此为可选属性。
display-name	触发器名称	Orchestrator 客户端中显示的名称。此为可选属性。
type	触发器类型	用于定义触发器的对象类型。此为必需属性。

表 6-24. 元素层次结构

父元素	子元素
<trigger-properties>	无

gauge 元素

<gauge> 元素定义您可对该查找器使用的计量器。您必须实现 IPluginAdaptor 的 registerEventPublisher() 和 unregisterEventPublisher() 方法来设置计量器。

<gauge> 元素为可选。插件可以拥有不限数量的 <gauge> 元素。<gauge> 元素具有以下属性。

类型	值	描述
name	计量器名称	计量器的名称。此为必需属性。
min-value	数字	最低阈值。此为可选属性。
max-value	编号	最高阈值。此为可选属性。

类型	值	描述
unit	对象类型	用于定义计量器的对象类型。此为必需属性。
format	字符串	受监视的值的格式。此为可选属性。

表 6-25. 元素层次结构

父元素	子元素
<events>	<description>

scripting-objects 元素

<scripting-objects> 元素是 <object> 元素的容器。

<scripting-objects> 元素为可选。<scripting-objects> 元素不包含属性。

表 6-26. 元素层次结构

父元素	子元素
<module>	<object>

object 元素

<object> 元素用于将插件技术的构造函数、属性和方法映射到 Orchestrator 脚本 API 所公开的 JavaScript 对象类型。

有关对象命名的约定信息，请参见 命名插件对象。

<object> 元素为可选。插件可以拥有不限数量的 <object> 元素。<object> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 名称	类的脚本名称。必须全局唯一。此为必需属性。
java-class	Java 类	由该 JavaScript 类封装的 Java 类。此为必需属性。
create	true（默认）或 false	如果为 true，您可以为该类创建一个新实例。此为可选属性。
strict	true 或 false（默认值）	如果为 true，您只能调用在 vso.xml 文件中注释或声明的方法。此为可选属性。
is-deprecated	true 或 false（默认值）	如果为 true，此对象映射的方法已弃用。此为可选属性。
since-version	字符串	Java 类被弃用时的版本。此为可选属性。

表 6-27. 元素层次结构

父元素	子元素
<scripting-objects>	■ <description> ■ <deprecated> ■ <url> ■ <constructors> ■ <attributes> ■ <methods> ■ <singleton>

constructors 元素

<constructors> 元素是 <object><constructor> 元素的容器。

<constructors> 元素为可选。<constructors> 元素不包含属性。

表 6-28. 元素层次结构

父元素	子元素
<object>	<constructor>

constructor 元素

<constructor> 元素用于定义构造函数方法。<constructor> 方法会在 API Explorer 中生成文档。

<constructor> 元素为可选。插件可以拥有不限数量的 <constructor> 元素。<constructor> 元素不包含属性。

表 6-29. 元素层次结构

父元素	子元素
<constructors>	■ <description> ■ <parameters>

Constructor parameters 元素

<parameters> 元素是 <constructor><parameter> 元素的容器。

<parameters> 元素为可选。<parameters> 元素不包含属性。

表 6-30. 元素层次结构

父元素	子元素
<constructor>	<parameter>

Constructor parameter 元素

<parameter> 元素用于定义构造函数的参数。

<parameter> 元素为可选。插件可以拥有不限数量的 <parameter> 元素。<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	要在 API 文档中使用的参数名称。此为必需属性。
type	Orchestrator 参数类型	要在 API 文档中使用的参数类型。此为必需属性。
is-optional	true 或 false	如果为 true，值可以为空。此为可选属性。
since-version	字符串	方法版本。此为可选属性。

表 6-31. 元素层次结构

父元素	子元素
<parameters>	无

attributes 元素

<attributes> 元素是 <object><attribute> 元素的容器。

<attributes> 元素为可选。<attributes> 元素不包含属性。

表 6-32. 元素层次结构

父元素	子元素
<object>	<attribute>

attribute 元素

<attribute> 元素会将 Java 类的属性从插件技术映射到 Orchestrator JavaScript 引擎提供的 JavaScript 属性。

<attribute> 元素为可选。插件可以拥有不限数量的 <attribute> 元素。<attribute> 元素具有以下属性。

类型	值	描述
java-name	Java 属性	Java 属性的名称此为必需属性。
script-name	JavaScript 对象	对应 JavaScript 对象的名称。此为必需属性。
return-type	字符串	此属性返回的对象类型。会显示在 API Explorer 文档中。此为可选属性。 注 如果 JavaScript 返回类型为 Properties，其支持的基础 Java 实现则为 java.util.HashMap 和 java.util.Hashtable。
read-only	true 或 false	如果为 true，您无法修改此属性。此为可选属性。
is-optional	true 或 false	如果为 true，此字段可以为空。此为可选属性。

类型	值	描述
show-in-api	true 或 false	如果为 false，此属性不会显示在 API 文档中。此为可选属性。
is-deprecated	true 或 false	如果为 true，此对象映射的属性已弃用。此为可选属性。
since-version	数字	属性被弃用时的版本。此为可选属性。

表 6-33. 元素层次结构

父元素	子元素
<attributes>	无

methods 元素

<methods> 元素是 <object><method> 元素的容器。

<methods> 元素为可选。<methods> 元素不包含属性。

表 6-34. 元素层次结构

父元素	子元素
<object>	<method>

method 元素

<method> 元素可以将 Java 方法从插件技术映射到 Orchestrator JavaScript 引擎公开的 JavaScript 方法。

<method> 元素为可选。插件可以拥有不限数量的 <method> 元素。<method> 元素具有以下属性。

类型	值	描述
java-name	Java 方法	Java 方法签名的名称，其参数类型放在括弧中，例如，getVms(DataStore)。此为必需属性。
script-name	JavaScript 方法	JavaScrip 方法相应的名称。此为必需属性。
return-type	Java 对象类型	此方法所获取的类型。此为可选属性。 注 如果 JavaScript 返回类型为 Properties，其支持的基础 Java 实现则为 java.util.HashMap 和 java.util.Hashtable。
static	true 或 false	如果为 true，此方法为静态。此为可选属性。
show-in-api	true 或 false	如果为 false，此属性不会显示在 API 文档中。此为可选属性。

类型	值	描述
is-deprecated	true 或 false	如果为 true，此对象映射的方法已弃用。此为可选属性。
since-version	编号	方法被弃用时的版本。此为可选属性。

表 6-35. 元素层次结构

父元素	子元素
<methods>	■ <deprecated> ■ <description> ■ <example> ■ <parameters>

example 元素

<example> 元素可让您将代码示例添加到 API Explorer 文档中显示的 JavaScript 方法。

<example> 元素为可选。<example> 元素不包含属性。

表 6-36. 元素层次结构

父元素	子元素
<method>	■ <code> ■ <description>

code 元素

<code> 元素提供了显示在 API Explorer 文档中的示例代码。

您需要在 <code> 和 </code> 标记之间提供代码示例。<code> 元素为可选。<code> 元素不包含属性。

表 6-37. 元素层次结构

父元素	子元素
<example>	无

Method parameter 元素

<parameters> 元素是 <method><parameter> 元素的容器。

<parameters> 元素为可选。<parameters> 元素不包含属性。

表 6-38.

父元素	子元素
<method>	<parameter>

Method parameter 元素

<parameter> 元素用于定义该方法的输入参数。

<parameter> 元素为可选。插件可以拥有不限数量的 <parameter> 元素。<parameter> 元素具有以下属性。

类型	值	描述
name	字符串	参数名称。此为必需属性。
type	Orchestrator 参数类型	参数类型。此为必需属性。
is-optional	true 或 false	如果为 true，值可以为空。此为可选属性。
since-version	字符串	方法版本。此为可选属性。

表 6-39. 元素层次结构

父元素	子元素
<parameters>	无

singleton 元素

<singleton> 元素会将 JavaScript 脚本对象创建为单一实例。

单一对象与静态 Java 类的行为相同。单一对象定义了插件要使用的通用对象，而不是定义 Orchestrator 在插件技术中访问的对象的特定实例。例如，您可以使用单一对象与插件技术建立连接。

<singleton> 元素为可选。<singleton> 元素具有以下属性。

类型	值	描述
script-name	JavaScript 对象	对应 JavaScript 对象的名称。此为必需属性。
datasource	Java 对象	此 JavaScript 对象的源 Java 对象。此为必需属性。

表 6-40. 元素层次结构

父元素	子元素
<object>	无

enumerations 元素

<enumerations> 元素是 <enumeration> 元素的容器。

<enumerations> 元素为可选。<enumerations> 元素不包含属性。

表 6-41. 元素层次结构

父元素	子元素
<module>	<enumeration>

enumeration 元素

<enumeration> 元素用于定义适用于特定类型所有对象的常用值。

如果特定类型的所有对象需要某个特定属性，且该属性值的范围有限，您可以将这些不同值定义为枚举条目。例如，如果某个对象类型需要 color 属性，且可用颜色只有红、蓝、绿，您就可以定义将这三种颜色值定义为三个枚举条目。这些条目将被定义为枚举元素的子元素。

<enumeration> 元素为可选。插件可以拥有不限数量的 <enumeration> 元素。<enumeration> 元素具有以下属性。

类型	值	描述
type	Orchestrator 对象类型	枚举类型。此为必需属性。

表 6-42. 元素层次结构

父元素	子元素
<enumerations>	■ <url> ■ <description> ■ <entries>

entries 元素

<entries> 元素是 <enumeration><entry> 元素的容器。

<entries> 元素为可选。<entries> 元素不包含属性。

表 6-43. 元素层次结构

父元素	子元素
<enumeration>	<entry>

entry 元素

<entry>元素用于为枚举属性提供值。

<entry> 元素为可选。插件可以拥有不限数量的 <entry> 元素。<entry> 元素具有以下属性。

类型	值	描述
id	Text	对象用于将枚举条目设置为属性时的标识符。此为必需属性。
name	Text	entry 名称此为必需属性。

表 6-44. 元素层次结构

父元素	子元素
<entries>	无

Orchestrator 插件开发的最佳做法

了解 Orchestrator 插件的结构和内容以及如何避免具体问题，可帮助您对您所开发的 Orchestrator 插件的相关方面进行改进。

- 构建 Orchestrator 插件的方法

  您可以使用不同方法来构建 Orchestrator 插件。您可以逐层开始构建插件，也可以同时开始构建插件的所有层。

- Orchestrator 插件类型

  通过使用插件，您可以借助 Orchestrator 集成常规用途库或实用程序（例如 XML 或 SSH）以及整个系统（例如 vCloud Director）。根据与 Orchestrator 集成的具体技术，插件可归类为服务插件、常规用途插件及系统插件。

- 插件实现

  在构建插件、实现所需 Java 类和 JavaScript 对象、开发插件工作流和操作以及提供工作流展示时，您可以使用一些实用做法和技巧。

- Orchestrator 插件开发建议

  在开发 Orchestrator 插件的不同组件时，遵循部分特定做法可帮助您改进插件的质量。

- 编写插件用户界面字符串和 API 的文档

  在编写 Orchestrator 插件的用户界面 (UI) 字符串和相关 API 的文档时，请遵循以下广受认可的样式和格式规则。

构建 Orchestrator 插件的方法

您可以使用不同方法来构建 Orchestrator 插件。您可以逐层开始构建插件，也可以同时开始构建插件的所有层。

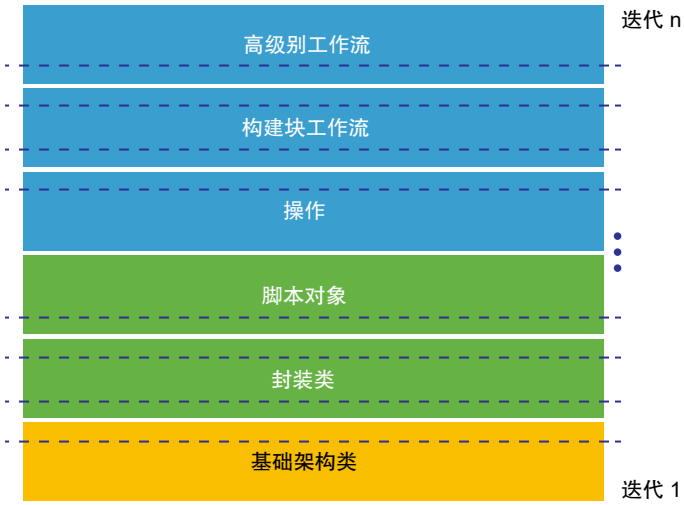
有关插件层的信息，请参阅 Orchestrator 插件的结构。

自下而上的插件开发

您可以使用自下而上的开发方法逐层构建插件。

自下而上的开发方法从低级别的层开始，随后逐层递增到更高级别的层，最终完成插件构建。如果该方法混合了交互式和迭代式开发方法，则每次迭代都会交付部分或整个层。在 N 次迭代结束后，即完全完成了插件。

图 6-3. 自下而上的插件开发



自下而上的插件开发方法的优势在于开发时每次都只关注于一层。

请注意自下而上插件开发方法的以下不利因素。

- 很难显示插件开发的进度，除非完成部分插入。
- 不太适合敏捷开发做法。

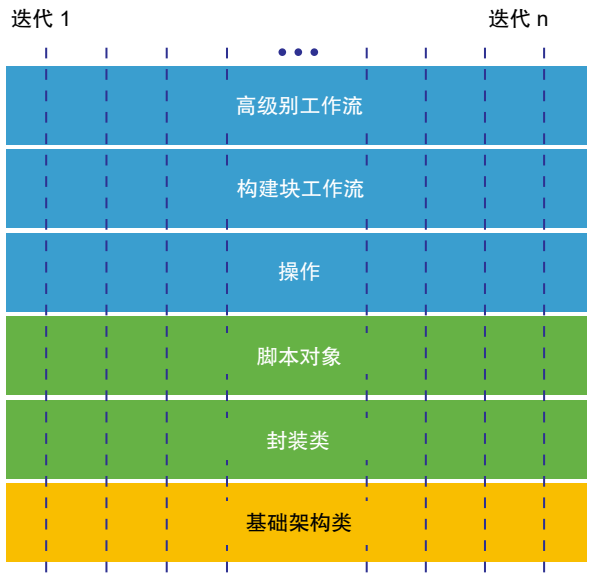
自下而上开发过程非常适合小型插件的开发，减少甚至无需多组封装类、脚本对象、操作或工作流。

自上而下的插件开发

使用自上而下的开发方法，可以将插件自上而下拆分为多个功能，从而进行构建。

如果自上而下的方法与敏捷开发过程混合使用，则在每次迭代时都会交付新功能。最后，在 N 次迭代结束后，即完全实现了插件。

图 6-4. 自上而下的插件开发



自上而下的插件开发方法具有以下优势。

- 插件开发的进度从首次迭代开始就易于查看，因为每次迭代后都会完成新功能并且每次迭代后都能发行并使用插件。
- 完成功能垂直拆分可非常清晰地定义成功条件和完成条件，以及在开发人员、产品管理人员和质量保证 (QA) 工程师之间加强交流。
- 可让 QA 工程师从开发流程的一开始就进行测试和自动化执行。此类方法可获得宝贵的反馈并缩短整个项目交付的周期。

自上而下的插件开发方法其不足之处在于不同层都在同时进行开发。

对于大部分插件，都可以应用自上而下的插件开发方法。此方法同样还适用于具有动态要求的插件。

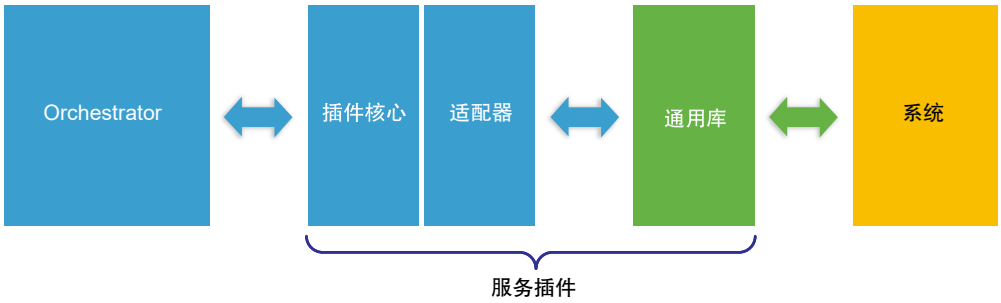
Orchestrator 插件类型

通过使用插件，您可以借助 Orchestrator 集成常规用途库或实用程序（例如 XML 或 SSH）以及整个系统（例如 vCloud Director）。根据与 Orchestrator 集成的具体技术，插件可归类为服务插件、常规用途插件及系统插件。

服务插件

服务插件或常规用途插件提供的功能可视为 Orchestrator 中的服务。

图 6-5. 服务插件的架构



服务插件将通用库或实用程序（例如 XML、SSH 或 SOAP）公开到 Orchestrator。例如，以下 Orchestrator 中可用的插件即为服务插件。

JDBC 插件	可让您在工作流中使用任意数据库。
Mail 插件	可让您在工作流中发送电子邮件。
SSH 插件	可让您在工作流中打开 SSH 连接并运行命令。
XML 插件	可让您在工作流中管理 XML 文档。

服务插件具有以下特征。

复杂度	服务插件的复杂度从低到中不等。服务插件在 Orchestrator 中公开特定库或库的一部分，以便提供具体功能。例如，XML 插件将文档对象模型 (DOM) XML 解析程序的实现添加到 Orchestrator JavaScript API。
大小	服务插件相对较小。除了所有插件都使用的相同基本类集，它们还需要能够提供新脚本对象的其他类，便于增加新功能。
清单	服务插件需要一份简短的对象清单就能运行，或者根本无需清单。服务插件具有通用的小型对象模型，因此，无需在 Orchestrator 清单中显示该模型。

系统插件

系统插件会将 Orchestrator 工作流引擎连接到外部系统，以便您可以编排外部系统。

以下是系统插件的示例。

vCenter Server 插件	可让您使用工作流来管理 vCenter Server 实例。
vCloud Director 插件	可让您与工作流内的 vCloud Director 安装进行交互。
Cisco UCSM 插件	可让您与工作流内的 Cisco 实体进行交互。

以下是系统插件的主要特征。

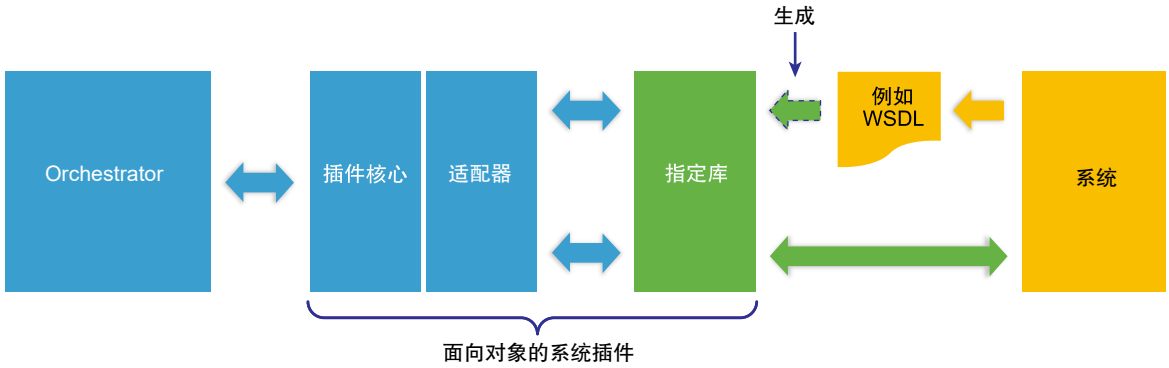
复杂度	相比常规用途的插件，系统插件的复杂度更高，因为其中的技术相对较为复杂。系统插件必须代表 Orchestrator 中外部系统的所有元素与外部系统进行交互，并在 Orchestrator 中提供其功能。如果外部系统提供集成机制，您可以用其在 Orchestrator 中更加轻松地公开系统的功能。但是，除了代表 Orchestrator 中外部系统的元素外，系统插件还可能需要提供高扩展性、提供缓存机制，以及处理事件和通知等。
大小	系统插件通常为中等大小或较大。除了基本类集，系统插件还需要许多其他类，因为其通常会提供大量脚本对象。系统插件可能需要部分可与之交互的其他帮助程序和辅助类。
清单	系统插件通常包含大量对象，您必须在清单中正确公开这些对象以便在 Orchestrator 中轻松找到并进行处理。由于需要公开系统插件的大量对象，您应该构建辅助工具或流程以尽可能多地为插件自动生成代码。例如，vCenter Server 插件会提供此类工具。

面向对象的系统插件

面向对象的系统提供了基于对象和 RPC 的交互机制。

面向对象的系统最常使用的模型为使用 SOAP 的 Web 服务模型。此模型中的对象具有一组与对象状态相关的属性，并提供一组可在目标系统调用的远程方法。

图 6-6. 面向对象的系统插件



在为面向对象的系统实施插件时，可以考虑以下内容。

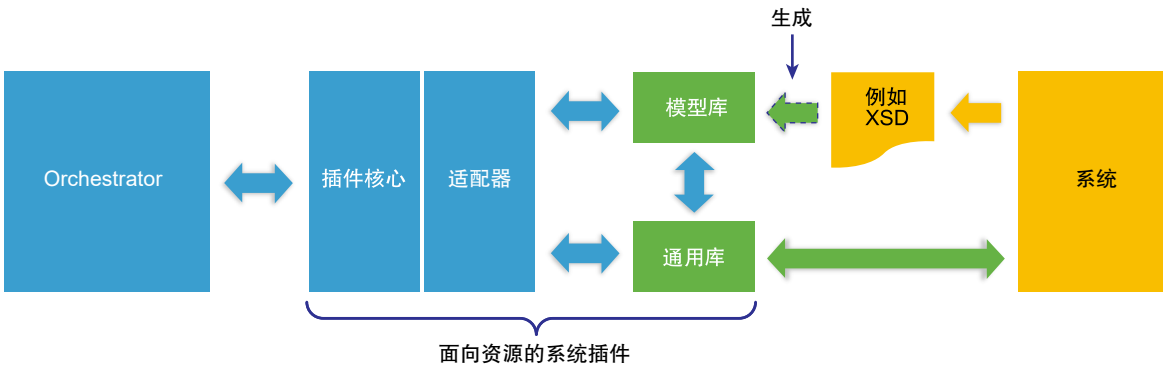
- 如果使用 SOAP，则可以使用 WSDL 文件来生成一组合并了对象模型和通信机制的类。
- 此对象模型几乎涵盖了要在 Orchestrator 中公开的所有内容。

面向资源的系统插件

面向资源的系统提供了基于使用 HTTP 方法的资源和简单操作的交互机制。

面向资源的系统最具代表性的模型是 REST 模型，例如合并了 XML。此模型中的对象具有一组与对象状态相关的属性。若要在目标系统上调用方法（通信机制），您必须使用标准 HTTP 方法（例如 GET、POST、PUT 等）并遵守相关约定。

图 6-7. 面向资源的系统插件



在为面向资源的系统开发插件时，可以考虑以下内容。

- 如果使用 REST 或仅 HTTP 与 XML，您会获得一个或多个可以读写消息的 XML 架构文件。通过这些架构，您可以生成定义对象模型的一组类。这组类仅定义了对象状态，因为操作由 HTTP 方法隐性定义（例如在 vCloud Director 插件中定义）或由特定 XML 消息显式定义（例如 Cisco UCSM 插件）。
- 您需要在另一组类中实施通信机制。这组类定义了与原始对象模型进行交互的新对象模型。通信机制的对象模型仅由对象和方法组成。

- 您可以在 Orchestrator 中同时公开原始对象模型和通信机制的对象模型。这样可能会增加复杂度，具体取决于如何公开两种对象模型，以及您是合并了这两种模型中的相关对象（以模拟面向对象的系统）还是将其保持分离。

插件实现

在构建插件、实现所需 Java 类和 JavaScript 对象、开发插件工作流和操作以及提供工作流展示时，您可以使用一些实用做法和技巧。

- 项目结构

  您可以为 Orchestrator 插件的项目应用标准结构。

- 项目内部机制

  您可以在实现插件时应用特定方法，例如缓存对象、将对象置于后台、克隆对象等。按照此类方法操作，您可以改进插件的性能、避免并发问题并改进 Orchestrator 客户端的响应能力。

- 工作流内部机制

  您可以执行工作流来监控 Orchestrator 插件执行的长时间操作。

- 工作流和操作

  若要使工作流开发和使用更加简单，您可以参考一些比较好的做法。

- 工作流展示

  创建工作流展示时，需应用特定结构和规则。

项目结构

您可以为 Orchestrator 插件的项目应用标准结构。

您可以使用标准 Maven 结构以及插件项目模块以清楚了解每个功能的具体位置。

表 6-45. 插件项目结构

模块	描述
`/myAwesomePlugin-plugin`	插件项目的根。
`/o11nplugin-myAwesomePlugin`	构成最终插件 DAR 文件的模块。
`/o11nplugin-myAwesomePlugin-config`	包含插件配置 Web 应用程序的模块。它会生成标准 WAR 文件。
`/o11nplugin-myAwesomePlugin-core`	此模块包含实现任一标准 Orchestrator 插件接口的所有类以及所用的其他辅助类。它会生成标准 JAR 文件。
`/o11nplugin-myAwesomePlugin-model`	此模块包含可帮助您通过插件将第三方技术与 Orchestrator 集成的所有类。相关类不应包含对标准 Orchestrator 插件 API 的任何直接引用。
`/o11nplugin-myAwesomePlugin-package`	此模块会导入包含操作和工作流的外部 Orchestrator 软件包文件，从而将其包含在最终插件 DAR 文件内。此模块可选

项目内部机制

您可以在实现插件时应用特定方法，例如缓存对象、将对象置于后台、克隆对象等。按照此类方法操作，您可以改进插件的性能、避免并发问题并改进 Orchestrator 客户端的响应能力。

缓存对象

您的插件可与远程服务交互，这一交互操作由代表服务端远程对象的本地对象提供支持。为实现插件的最佳性能以及 Orchestrator 用户界面的卓越响应能力，您可以缓存本地对象而不是每次都从远程服务上获取。您可以考虑缓存的范围，例如所有插件客户端使用一个缓存、插件按用户各使用一个缓存以及第三方服务按用户各使用一个缓存。实施后，缓存机制会与插件接口集成以查找并使对象失效。

将对象置于后台

如果必须在插件清单中显示较长的对象列表，并且没有较快途径来检索这些对象，您可以将对象置于后台。例如，通过使对象具有 `fake` 和 `loaded` 状态，您可以将其置于后台。假设 `fake` 对象易于创建并且在清单中要显示的信息最少，例如名称和 ID。那么，就可以始终返回 `fake` 对象，并且当确实需要所有信息（真实对象）时，正在使用的实体或插件可以自动调用方法 `load` 来获取真实对象。您甚至可以配置在返回假对象后自动启动加载对象流程，从而预测使用中实体的操作。

克隆对象以避免并发问题

如果为插件使用缓存，则必须克隆对象。使用的缓存如果始终将对象的同一实例返回至每个发起请求的实体，则可能会发生意料之外的结果。例如，实体 A 请求了对象 O，并且实体查看了清单中的对象及其所有属性。与此同时，实体 B 也请求了对象 O，并且实体 A 运行了开始更改对象 O 属性的工作流。在运行结束时，工作流调用了对象的 `update` 方法来更新服务器端的对象。如果实体 A 和实体 B 获得对象 O 的同一实例，则实体 A 会在清单中看到实体 B 执行的所有更改，甚至在更改内容提交到服务器之前也能看到。如果运行正常，这应该不是问题。但如果运行失败，则实体 A 请求的对象 O 属性无法恢复。在这种情况下，如果缓存（插件的 `find` 操作）返回对象的克隆内容而不是始终返回同一实例，即每个都使用实体查看并修改各自的副本，则至少可在 Orchestrator 内避免并发问题。

向他人通知更改

同时使用缓存和克隆对象时可能会发生问题。最大的问题是使用实体视图的对象可能不是可用于对象的最新版本。例如，如果某个实体显示了清单、已经加载一次对象，但同时如果另一实体更改了部分对象，则首个实体不会看到更改内容。为避免此问题，您可以使用 Orchestrator 插件 API 中的 `PluginWatcher` 和 `IPluginPublisher` 方法通知发生的更改，从而使 Orchestrator 客户端的其他实例能看到更改内容。如果清单中某个对象的更改影响了清单的其他对象，并且受影响对象也需要获得通知，那么上述情况也适用于 Orchestrator 客户端的唯一实例。倾向于使用通知的操作包含添加、更新和删除对象（如果这些对象或其部分属性显示在清单内）。

启用随时查找任意对象

您必须实现 `IPluginFactory` 接口的 `find` 方法来仅按类型和 ID 查找对象。重新启动 Orchestrator 并恢复工作流后，可以直接调用 `find` 方法。

模拟查询服务（如无）

Orchestrator 客户端可以要求在特定情况下查询部分对象，或要求将对象显示为列表或表格而非树形图。这意味着您的插件必须可以随时查询部分对象集。如果第三方技术提供了查询服务，则您需要适应并使用该服务。否则，无论解决方案的复杂度有多高，性能有多低，您都应可模拟查询服务。

查找方法不应返回运行时异常

`IPluginFactory` 接口中用于在插件中实施搜索的方法不应出现受控或不受控的运行时异常。这可能是当工作流在运行时出现异常验证错误失败的原因。例如，在工作流的两个节点之间，如果首个节点的输出是第二个节点的输入，则会调用 `find` 方法。此时，如果由于任何运行时异常而无法找到对象，您可能在 Orchestrator 客户端中不会收到除验证错误之外的更多信息。在此之后，根据插件记录异常的方式，可以在日志文件内获取或多或少的信息。

工作流内部机制

您可以执行工作流来监控 Orchestrator 插件执行的长时间操作。

您可以部署用于监控长时间运行操作（例如任务监控）的工作流。该工作流可基于 Orchestrator 触发器和等待事件。您必须考虑到正等待任务的被阻挡工作流可以在 Orchestrator 服务器启动后立即恢复。插件必须能够获取所有必需信息以正确恢复监控流程。

监控工作流或其可在内部使用的任务应提供相关机制，从而指定池化率和可能的超时。

对工作流内的一段脚本代码进行调试并非易事，特别是当代码未调用任何 Java 代码时。因此，有时只能选择使用默认 Orchestrator 脚本对象提供的日志记录方法。

工作流和操作

若要使工作流开发和使用更加简单，您可以参考一些比较好的做法。

开始以构建块形式开发工作流

构建块可以是一个简单的工作流，需要一些输入参数并会返回简单输出。如果您有一组丰富的构建块，可以轻松创建级别更高的工作流，并且可以使用一组更好的工具来构建复杂工作流。

基于较小组件创建级别更高的工作流

如果您需要开发具有多个输入和内部步骤的复杂工作流，您可以将其拆分为较小且较简单的构建块工作流和操作。

可随时创建操作

您可以在开发工作流时创建操作来获得额外的灵活性。

- 轻松为脚本方法创建复杂对象或参数
- 避免始终重复使用常用的代码片段
- 执行用户界面验证

工作流应可随时调用操作

操作可以在工作流架构中直接调用为节点。这样能使工作流架构更简单，因为您无需添加脚本代码块以调用单个操作。

填写所需信息

提供工作流或操作的每个元素的相关信息。

- 提供工作流或操作的说明。
- 提供输入参数的说明。
- 提供输出的说明。
- 提供工作流属性的说明。

保持版本信息始终为最新

为插件添加版本信息时，添加实用注释，例如，插件的主要更新、重要的实现详情等。

工作流展示

创建工作流展示时，需应用特定结构和规则。

在工作流展示中对工作流输入使用以下属性。

表 6-46. 工作流输入属性

属性	使用情况
`Show in Inventory`	使用该属性帮助用户通过清单视图运行工作流。
`Specify a root object to be shown in the chooser`	使用该属性帮助用户选择输入。如果根对象可在展示中刷新、是一个属性或由某个对象方法检索，则您需要创建或设置相应的操作来刷新展示中的对象。
`Maximum string length`	对长字符串（例如名称、描述、文件路径等）使用该属性。
`Minimum string length`	使用该属性避免测试工具中包含空字符串。
`Custom validation`	实施通过操作完成的复杂验证。

使用步骤和显示组整理输入。此类整理可帮助用户识别和区分工作流的所有输入参数。

Orchestrator 插件开发建议

在开发 Orchestrator 插件的不同组件时，遵循部分特定做法可帮助您改进插件的质量。

表 6-47. 插件实现实用做法

组件	条目	描述
常规	访问第三方 API	插件应尽可能提供简化的第三方 API 访问方法。
	接口	插件应向用户提供一致且标准的接口，即使 API 未能做到也是如此。
操作	脚本对象	您应该为每个创建、修改、删除及所有其他对脚本对象可用的方法创建操作。
	描述	操作的描述应说明操作有何用处而不是如何运作。
	脚本	使用脚本来获取对象的属性或方法时，可以检查对象值是否与 `null` 或 `undefined` 不同。

表 6-47. 插件实现实用做法 （续）

组件	条目	描述
	弃用	如果某个操作已被弃用，`comment` 或 `throw` 语句应指示替代操作，或操作应调用新的替代操作，从而使得在操作的已弃用版本上构建的解决方案不会失效。
工作流	编排技术中的用户界面操作	您应当为编排技术用户界面中可用的每个操作创建工作流。
	描述	工作流的描述应说明工作流有何用处而不是如何运作。
	展示属性 `mandatory input`	您必须为所有强制工作流输入设置 `mandatory input` 属性。
	展示属性 `default value`	如果开发了一个配置实体的工作流，则工作流展示应加载该实体的默认配置值。例如，如果开发名为“主机配置”的工作流，则工作流的展示必须加载主机配置的默认值。
	展示属性 `Show in inventory`	您必须设置 `Show in inventory` 属性以便在清单对象上拥有上下文工作流。
	展示属性 `specify a root parameter`	如果无需从树根浏览清单，则应在工作流中使用该属性。
	工作流验证	您必须验证工作流并修复所有错误。
	对象创建	所有创建了新对象的工作流都应将新对象返回为输出参数。
	弃用	如果某个工作流已被弃用，则 `comment` 或 `throw` 语句应指示替代工作流，或弃用的工作流应调用新的替代工作流以确保在工作流先前版本上构建的解决方案不会失效。
清单	主机断开连接	如果清单包含主机连接并且该主机不再可用，则应表明主机已断开连接。您可以通过重命名根对象（附加 `– disconnected`）或移除该对象下的对象树来执行此操作，具体方法与 vCloud Director 插件的方法相同。
	`Select value as list` 属性	清单对象必须可选择为 `treeview` 或 `list`。
	主机管理器	如果插件为目标系统实现了 `host` 对象，则应存在一个父 `hostmanager` 根对象，具有用于添加、移除或编辑主机属性的属性。
	获取或更新对象	如果查询服务在编排技术上运行，您应使用该服务来获取多个对象。
	子对象发现	如果您需要单独检索子对象，则检索过程必须采取多线程并且不能受单个错误阻止。
	Orchestrator 对象更改	所有可以更改清单中元素状态的工作流必须更新清单以避免对象未能同步。
	外部对象更改	您可以使用通知机制来通知编排技术中由于在 Orchestrator 外执行的操作所产生的更改。如果此类操作会将对象从编排技术中移除，您必须相应刷新清单以避免发生故障或丢失数据。例如，如果从 vCenter Server 中删除虚拟机，则 vCenter Server 插件将更新清单以移除已移除虚拟机的对象。
	查找器对象	查找器对象应具有可用于区分对象的属性。这些通常都是位于用户界面中的属性。
脚本对象	实现	必须实现 `equals` 方法以确保 `==` 操作在同一对象上运行，这是因为在某些情况下对象可能拥有两个实例。
	插件对象属性	具有父对象的对象应实现 `parent` 属性。
	插件对象属性	具有子对象的对象应实现 `GET` 方法，以返回子对象数组。

表 6-47. 插件实现实用做法 （续）

组件	条目	描述
	清单对象	清单对象应可使用 `Server.find` 搜索。
		所有清单对象应可序列化以便用作工作流中的输入或输出属性。
	构造函数和方法	在大多数情况下，可编辑脚本的对象应具有构造函数，或应通过其他对象属性或方法返回。
	对象 ID	如果对象具有从外部系统颁发的 ID，则在编排多台服务器时应使用内部 ID 以确保不会发生 ID 重复。
	搜索对象	`search` 或 `find` 方法应实现筛选器，以便可以查找指定的名称或 ID，而不是仅查找所有对象。例如，Orchestrator 服务器具有 `Server.FindForId` 方法，可按 ID 查找插件对象。为此，必须为插件中的每个可查找对象实现该方法。
	触发器	如有可能，触发器应可用于发生更改的对象，这样 Orchestrator 可以对各类事件触发策略。例如，为确定新虚拟机何时添加、打开电源、关闭电源等，Orchestrator 可以监控 `Datacenter` 对象上 vCenter 插件中的触发器或事件。
	对象属性	驻留在其他插件中的对象所具有的属性应可从一个插件对象轻松转换到另一个。例如，虚拟机对象需要拥有 `moref`（受管对象引用 ID）。
	会话管理器	如果您要连接到可能存在不同会话的远程服务器，插件应实现两种会话：共享会话和单用户会话。
触发器	触发器	所有长操作和阻止方法都应能异步启动并返回任务，并在完成时生成触发器事件。
枚举	枚举	给定类型的枚举应拥有可在枚举中选择不同值的清单对象。
日志记录	日志	方法应实现不同日志级别。
版本控制	插件版本	插件版本应遵循相关标准并随插件更新一同更新。
API 文档	方法	API 文档中描述的方法不得在对象上引发异常 `no xyz method / property`。相反，方法应在无可用属性时返回 `null`，并在这些属性不可用时详细记录。
	`vso.xml`	所有对象、方法和属性都必须记录在 `vso.xml` 内。

编写插件用户界面字符串和 API 的文档

在编写 Orchestrator 插件的用户界面 (UI) 字符串和相关 API 的文档时，请遵循以下广受认可的样式和格式规则。

常规建议

- 若插件中涉及 VMware 产品，请使用其官方名称。例如，使用以下产品的官方名称和 VMware 术语。

正确术语	请勿使用
vCenter Server	VC 或 vCenter
vCloud Director	vCloud

- 每句工作流描述都以句点结尾。例如，`Creates a new Organization.` 是一句工作流描述。

- 使用文本编辑器和拼写检查器来编写描述，然后将其移至插件。
- 确保插件名称完全匹配所关联第三方产品已批准的名称。

工作流和操作

- 编写信息性描述。大多数操作和工作流只需一两个句子即可。
- 较高级别工作流可能要包含更宽泛的描述和备注。
- 编写描述时直接以动词开头，例如：`Creates…`。不要使用自引用语言，例如 `This workflow creates`。
- 在句子完整的描述结尾使用句点。
- 描述工作流或操作有何用处而不是如何实现。
- 工作流和操作通常随附在文件夹和软件包中。同时也随附了这些文件夹和软件包的简短描述。例如，工作流文件夹的描述可类似 `Set of workflows related to vApp Template management`。

工作流和操作的参数

- 使用描述性的名词短语（例如 `Name of`）作为工作流和操作描述的开头。请勿使用诸如 `It's the name of` 等短语。
- 请勿在参数和操作描述的结尾添加句点。这些不是完整的句子。
- 工作流的输入参数必须指定标签，在展示视图中包含相应名称。在许多情况下，您可以合并显示组中的相关输入。例如，您可以创建标签为“组织”的显示组并将名称和完整名称输入放置在“组织”组内，而不是分别为两种输入内容创建“组织名称”和“组织全名”标签。
- 对于步骤和显示组，还将添加工作流展示中显示的描述或备注。

插件 API

- API 文档指 `vso.xml` 文件和 Java 源文件中的所有文档。
- 对于 `vso.xml` 文件，查找器对象和脚本对象的描述采用与工作流和操作描述相同的规则。对象属性和方法参数的描述采用与工作流和操作参数描述相同的规则。
- 避免在 `vso.xml` 文件中使用特殊字符，并且需要在 `<![CDATA[insert your description here!]]>` 标记中包含描述。
- 对 Java 源文件使用标准 Javadoc 样式。

使用 Maven 创建插件

7

Orchestrator Appliance 提供了一个包含 Maven 项目的存储库，您可使用其从原型创建插件项目。

存储库托管在 `https://orchestrator_server:8281/vco-repo/` 或 `http://orchestrator_server:8280/vco-repo/` 上，以防您的 Maven 版本不支持 HTTPS 协议。此位置内嵌在标准 Orchestrator Maven 插件项目的 `pom.xml` 文件中。仅当部署 Orchestrator Appliance 之后，您才能访问存储库。

本章讨论了以下主题：

- 使用 Maven 从原型创建 Orchestrator 插件
- Maven 原型
- 基于 Maven 的插件开发最佳做法

使用 Maven 从原型创建 Orchestrator 插件

您可以通过在命令行界面中运行命令，从原型创建标准 Orchestrator Maven 插件。

前提条件

- 验证是否安装了 Orchestrator Appliance 5.5.1 或更高版本。
- 验证是否安装了 Apache Maven 3.0.4 或 3.0.5。

步骤

1 通过选择原型以交互式模式创建项目。

```
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 仅当已部署 Orchestrator Appliance 之后，您才能访问 Maven 存储库。

2 （可选） 如果无法通过 HTTPS 访问存储库，可通过 HTTP 进行访问。如果通过 HTTP 访问存储库或拥有有效的 SSL 证书，可以创建项目而无需使用 `-Dmaven.wagon.http.ssl.allowall=true` 标记。

```
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 导航到项目目录并构建插件。

```
cd project_dir && mvn clean install -Dmaven.wagon.http.ssl.insecure=true -
Dmaven.wagon.http.ssl.allowall=true
```

如果构建过程成功，则会在 DAR 模块的 target/ 目录中生成 .dar 文件。

Maven 原型

您可以将一组预定义的 Maven 原型用作开发 Orchestrator 插件的模板。

下表介绍了 Orchestrator 中可用的默认 Maven 原型。

表 7-1. 默认 Maven 原型

原型	描述
com.vmware.o11n:o11n-plugin-archetype-simple	com.vmware.o11n:o11n-plugin-archetype-simple
com.vmware.o11n:o11n-package-archetype	仅包含内容的 Maven 项目，可用来将软件包保存为源表单以更好地与 RCS、diff、后期处理等交互。
com.vmware.o11n:o11n-client-archetype-rest	一种简单的命令行工具，可与 Orchestrator REST API 通信并调用工作流。
com.vmware.o11n:o11n-plugin-archetype-inventory	一个演示清单使用的插件。该插件为单一类型实现了存储库、适配器和工厂。清单存储在磁盘上的文件内。
com.vmware.o11n:o11n-archetype-inventory-annotation	一个插件，其 vso.xml 描述符生成在所有注释顶层。
com.vmware.o11n:o11n-archetype-spring	一个插件，使用基于 Spring 的 SDK，提供了支持 DI 的环境，且相比标准插件 API，添加了更高级别服务。
com.vmware.o11n:o11n-plugin-archetype-modeldriven	一个生成插件框架的原型，用于通过 ModelDriven 构建插件。

基于 Maven 的插件开发最佳做法

您可以执行一组任务来改进使用 Maven 创建的 Orchestrator 插件的交付过程。

使用存储库管理器

如果在较大组织中创建插件，请使用企业存储库管理器来设置要添加为代理存储库的默认 Orchestrator Appliance 存储库。使用中央存储库改进管理和插件项目协作。在新存储库中完成首个内部版本，存储库管理器会缓存 Orchestrator Appliance 存储库中的项目，您可以关闭默认存储库。

锁定工作流

在验证插件中的所有工作流都正常运行后，请将其锁定以防止未经授权的修改。通过锁定工作流，您可以确保插件的基本函数不会遭到破坏。如果用户出于特定目的必须修改默认工作流，可以创建原始工作流的副本并编辑该副本。

可通过以下两种方法使用锁定工作流生成发布版本。

- 将 -DallowedMask=vf 参数传输到 Maven。

- 编辑 pom.xml 并将 allowedMask 参数的值更改为 vf。

```
<allowedMask>vf</allowedMask>
```

使用软件包签名证书

使用自签名的证书或由证书颁发机构签名的证书，确保插件的完整性和可靠性。将证书存储在 _dunesrsa_alias_ 别名的密钥库中，方法是使用 JDK 中的密钥工具将其导入。

有两种方法可以指定密钥库文件路径和密钥库密码。

- 定义 *MAVEN_OPTS* 变量的 –DkeystoreLocation 和 –DkeystorePassword 命令行参数。
- 编辑 pom.xml 文件以手动插入值。例如，

```
<keystore>密钥库文件的路径</keystore>
<storepass>密钥库密码</storepass>
```

如果未导入任何密钥库，.package 文件会由 archetype.keystore 文件签名。