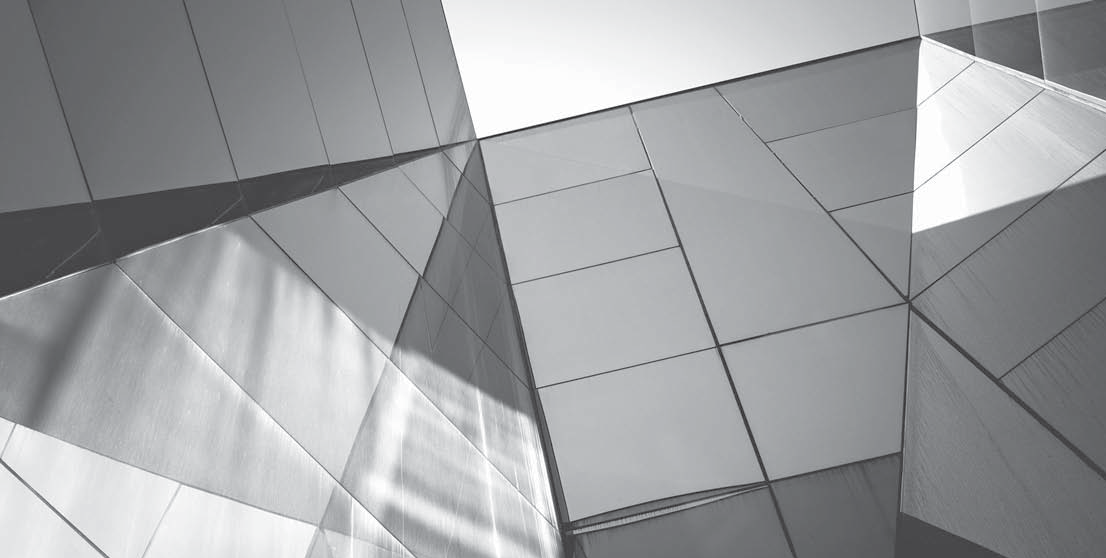


部分

# 四

## 异步 服务和事件

*本页故意留空*



章

# 12

## 具有 SOA 组合和服务总线的异步服务Asynchronous Services with SOA Composites and Service Bus

我写下你的号码并回复你“可能比”请保持排队“更有效率。前者是异步通信的典型例子。

**“L**

这可能比实际保持线路更有效，这是同步的交谈方式。在排队等待响应时，您被阻止了——无法 参与有用的活动。

在现实生活中，当你告诉一个人一些事情时，你不介意等待答案（除了某种形式的承认你的信息被听到和理解之外），如果和时间：

* 您问了一个想要答案的问题。
* 在你收到回应之前，你无法有意义地继续你的生活。
* 您不必等待太久（有点主观）。
* 如果你不等待，你将失去获得答案的机会。
* 您可以承受等待（因为等待所涉及的时间或资源成本）。

但是，如果这些考虑不适用，您希望在说出您的作品或提出您的问题后尽快继续您的生活。

在服务领域，也存在着非常相似的考虑因素。将这些项目符号中的“you”替换为“服务使用者”，你就有了一个清单，它将帮助您决定何时必须同步交互， 以及何时可以异步处理交互。

同步等待并非绝对需要的响应不仅是浪费时间：也是对稀缺系统资源（如内存和主要处理线程）的浪费。

同步交互也增加了系统的脆弱性。当我们可以使用多个更小、更简单、解耦的链而不是长、复杂、全依赖的链时，这将是更可取的。因此，在前面的大部分章节中讨论了同步交互之后，我们将在本章中重点讨论如何从 SOA 套件中公开异步接口。这些 接口将接受请求，停止通信，继续执行请求 ，并在回调中提供响应。正如我们将看到的，BPEL 提供了实现此类操作的最简单方法，尽管仅使用 Service Bus 或 Mediator，我们可以获得类似的结果。

由于 BPEL 流程可以长时间运行，因此我们可以谈论流程的实例，并且可以与这样的实例进行通信。例如，这可以是了解有关实例（状态）的信息、向实例提供其他信息或中止实例。确保请求最终到达正确的实例是我们面临的挑战之一 - 在 BPEL 中通过一种称为“关联”的机制进行处理。

作为一个实际的展示案例，我们将研究 *AircraftServiceOrchestrator*——航空公司与赛博特机场燃料、清洁、餐饮和维修服务提供商之间的中间人，帮助各方协商在特定条件下提供服务的同意条款。

### 公开异步服务

异步服务这个术语可能有些误导。我们用它来指示与两个接口关联的服务类别。一种是由服务本身实现的接口，具有一个或多个仅接受输入的单向操作。另一个接口是回调接口：服务在回调以传递响应消息时所需的接口。此接口必须由调用服务的一方实现。实现的地址必须与请求消息一起传递，以便服务知道在何处调用响应。在人机交互中，回调接口是电话或传真机及其有线或无线连接。

只有当您能够以服务提供商描述的方式被回调时 ，您才能成功调用异步服务并获得所需的响应 。

###### 注意

*服务操作可以定义输出（同步返回），还可以指定回调接口。服务操作的实现方式是，它首先返回同步响应，然后调用回调服务并传递其他消息。在本章中，我们将假设异步服务操作是单向的，并向回调服务传递单个异步响应消息。*

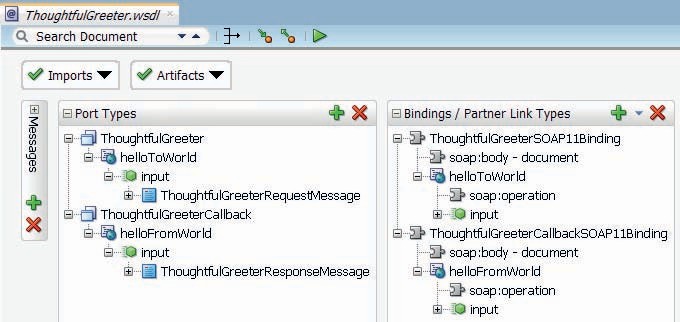
异步 Web 服务的实现除了同步服务实现必须执行的操作外，还需要执行一些步骤 。这些步骤涉及对请求标头的解释 ，以提取要向其发送响应的地址，以及用于标识会话的消息标识符。此外，服务实现必须 对动态提取的地址进行服务调用。

本部分讨论如何使用服务总线项目、中介器（首先是 BPEL）实现异步服务。我们还将研究如何从这三个环境中调用异步服务。中介和服务总线都可以在同步和异步后端或业务服务之上公开异步接口。此外，BPEL 还可以在异步后端之上提供同步接口。

本章的联机资源描述了仅使用 Java 和 JAX-WS 实现异步 Web Service，以及从 Java SE 程序调用异步 Web Service。

我们将实现的服务是 hello world 的异步形式： *ThoughtfulGreeterService*

（图 12-1）。



**图 12-1。** *ThoughtfulGreeterService 的 WSDL 定义*

服务定义描述了两个接口：主接口由服务本身实现，回调接口由服务使用者实现。前一个接口有一个单向操作 - *helloToWorld* - 将带有 name 元素的简单请求消息作为关键方面。它不返回任何输出。

回调接口还包含一个操作，该操作称为 *helloFromWorld*。此操作也是一种方式：它将来自 *ThoughtfulGreeterService*  的响应消息作为输入，并且不返回任何输出。此操作只能由 *ThoughtfulGreeterService* 主接口的实现调用。

**使用 BPEL 组件实现异步 ThoughtfulGreeterService**

*ThoughtfulGreeterService* 只是一个需要时间来响应的 *hello world* 服务。更准确地说：它完成与接受请求消息的使用者的同步交互，它通过必须执行的任何处理来组合响应消息，然后回调到指定的地址以移交响应 - 该响应作为请求发送到回调服务。这确实可能需要一些时间。但是，它也可能几乎是瞬间的。异步并不一定意味着缓慢或延迟。这*确实*意味着解耦：接收响应的交互发生在与首次调用服务的交互不同的交互中。

此服务的 BPEL 实现步骤非常简单（参见 图 12-2 中的结果）：

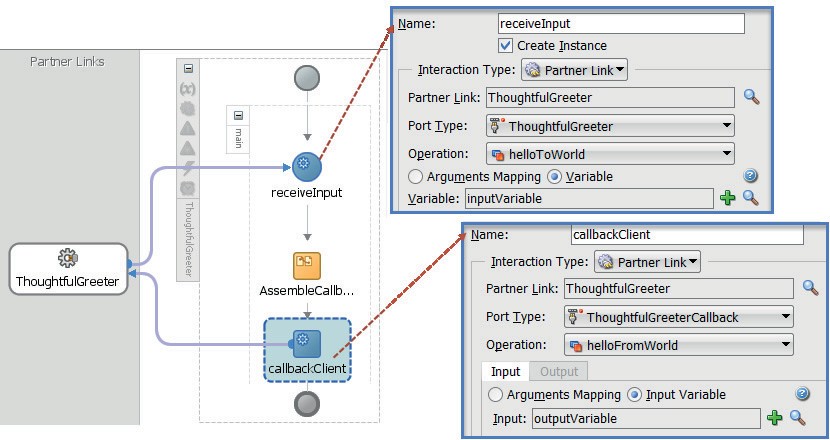
* 创建新的 SOA 组合应用程序 - 空模板。
* 添加基于 ThoughtfulGreeterService.wsdl 的 BPEL 组件。
* 检查创建的 BPEL 流程（Receive 和 Invoke 而不是 Reply）。
* 选择要实现的角色。
* 基于 WSDL 在组合级别创建 SOAP 服务绑定。
* 添加 Assign 活动以设置输出变量。

请注意，在 Receive 和 Invoke 活动之间，我们可以包含任何其他活动，包括同步或异步服务调用、迭代（针对每个）和并行作用域（流）。不存在超时的风险，因为没有组件在等待结果，至少在被阻止的同步会话中不会。服务使用者是否可能在回调接口后面等待响应到达（具有固定的截止时间和自己的超时 机制），这超出了本章的范围。

##### 从 SoapUI 测试异步服务

与测试同步服务相比，测试异步 Web 服务有些特殊，因为除了要发送到公开的服务端点的请求消息之外，SoapUI 还需要提供回调服务，并且必须在请求标头中设置适当的 WS-Addressing 标头，以将异步服务定向到回调服务。

SoapUI 有一个模拟服务工具。这允许我们发布 Web 服务 — 基于 WSDL 文档中的端口类型，并在我们选择的端口和端点上发布。回调接口只有一种方式：它接收来自异步服务的响应消息，这就是故事的结尾。在 SoapUI 中，我们可以检查 Mock Service 接收到的所有消息， 因此我们将知道异步服务是否以及何时发送响应。



**图 12-2。** *实现异步接口的 BPEL 流程（最终活动*

*是 Invoke 而不是 Reply，它在 ThoughtfulGreeter partnerLink 上调用不同的 portType ）*

###### 提示

*在线资源详细解释并展示了如何使用 ThoughtfulGreeterService 的模拟服务设置测试项目。*

创建一个新的 SoapUI 项目 - 基于已部署的 *ThoughtfulGreeter*  服务公开的 WSDL 文档（在部署 SOA 组合后，在 EM FMW 控制台中显示为 WSDL URL 的地址，例如：**http://localhost:7101/soa-infra/ services/default/AsynchronousSOAComposite/ThoughtfulGreeter\_ep？WSDL**）。

该项目是使用 WSDL 中两个绑定的测试用例和请求创建的。

接下来，我们需要 SoapUI 中的 MockService，用于实现 ThoughtfulGreeterService 的回调接口 （PortType *ThoughtfulGreeterCallback*）。 右键单击 SoapUI 中的项目。从上下文菜单中选择“新建 SOAP MockService”。将 MockService 的名称设置为 *ThoughtfulGreeterCallbackService*，然后单击“确定”。 从 helloFromWorld *操作的上下文*菜单中添加 MockOperation。

打开 MockService 编辑器，从 Mock Service 的上下文菜单中打开 Mock Service 的选项窗口。将模拟服务的路径和端口设置为适当的值，例如 /*MockThoughfulGreeterCallback* 和 8081。

启动模拟服务。请注意，我们不必为此模拟服务和操作指定响应消息。该操作是一种方式，只是为了异步服务提供其响应而调用。

使用 “Mock Editor”（模拟编辑器）窗口中的图标，打开 Mock Service 的 WSDL 页面。这是获取 MockService 终结点的最简单方法。我们需要此终结点，以将其作为 ReplyTo 地址在请求中传递给 ThoughtfulGreeterService.helloToWorld 操作。

##### 在 ThoughfulGreeterService 的测试请求中配置 WS-Addressing

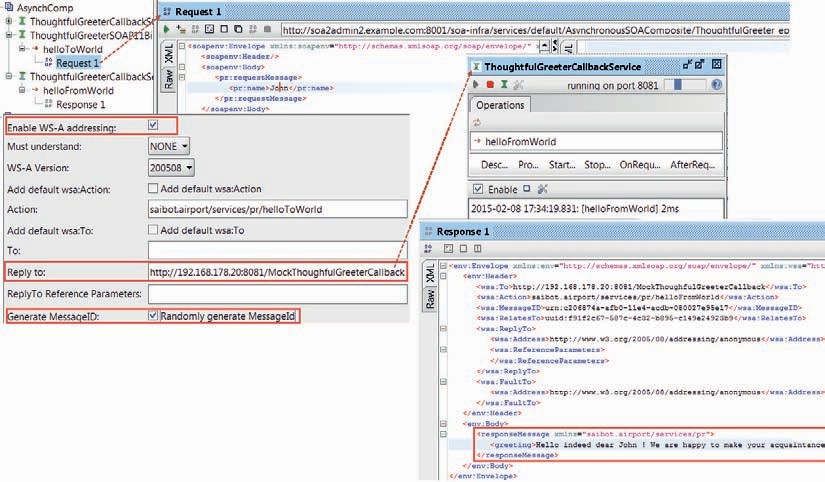
为了让异步服务（在本例中为 BPEL 引擎）知道将其响应消息发送到何处，以及要包含哪些 messageId，以便允许请求和异步响应之间的关联，必须在请求中包含多个 WS-Addressing 标头。

打开 helloToWorld  *操作的测试请求*。将终端节点设置为部署后在 JDeveloper 控制台中找到的终端节点。为请求消息中的 name 元素设置一个值。

打开“寻址详细信息”选项卡。选中标记为 Enable WS-Addressing 的复选框。将 ReplyTo 属性设置为剪贴板缓冲区中的终结点地址（从模拟服务的 WSDL 页复制）。选中标记为“随机生成 MessageId”的复选框，如图 12-3 所示。

这基本上意味着 SoapUI 将向它发送到 *ThoughtfulGreeterService* 的请求添加多个 WS-Addressing 标头，其中包括一个用于 *ReplyTo* 地址（回调端点）和要在回调中使用的消息 ID。

现在从 SoapUI 进行调用。在 MockService 编辑器中签入 SoapUI。在这里，您将 找到传入请求的日志行。当您检查消息交换结果时，您将



**图 12-3。** *配置 SoapUI 测试请求（将 WS-Addressing 属性配置为*

*guide 回调消息）和 Mock Service 接收来自 ThoughtfulGreeterService 的异步响应*

查找响应消息以及 *RelatesTo*  标头属性，该属性与请求消息中发送的消息 ID 相同。

当然，您还可以检查在为异步服务发送 SoapUI 测试请求时创建的 SOA 复合实例的流跟踪和 BPEL 审计跟踪。

请注意，在 BPEL 中实现服务时，我们完全不必处理 WS-Addressing 标头或其他有关回调位置或相关 ID 的低级细节 。BPEL 引擎负责处理所有这些问题。

#### 使用调解器实现异步服务

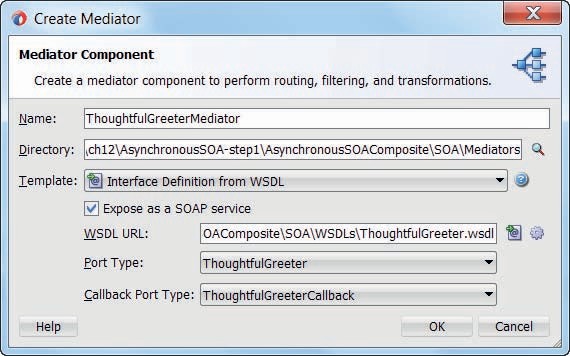
调解器还可以实现异步服务。它不像 BPEL 组件那样优雅。开发人员必须从请求和作为回调服务响应发送的消息中读取和设置 WS-Addressing 设置。我们将简要讨论这种方法。

将 Mediator 组件添加到 SOA 组合中。称其为 *ThoughtfulGreeterMediator*。选择 *Interface Definition from WSDL 模板，选中* Expose as a SOAP service *复选框，*然后选择 ThoughtfulGreeter.wsdl 文档。选择“*ThoughtFulGreeter*”接口作为“PortType”，选择“*ThoughtfulGreeterCallback*”作为“Callback Port Type”，如图12-4所示。

此外，根据 WSDL 文档中的回调接口，将 SOAP 引用绑定添加到组件。注意：在设计时为此绑定设置的终结点无关紧要。在运行时，将根据从服务使用者收到的 WS-Addressing ReplyTo 标头设置端点 ，以进行回调以传递异步响应。

将调解器连接到此 SOAP 绑定，如图 12-5 所示。

打开 Mediator 编辑器。它显示实现 *ThoughtfulGreeterCallback* 接口的（虚拟）SOAP 引用的路由规则。



**图 12-4。** *配置具有异步接口的调解器*

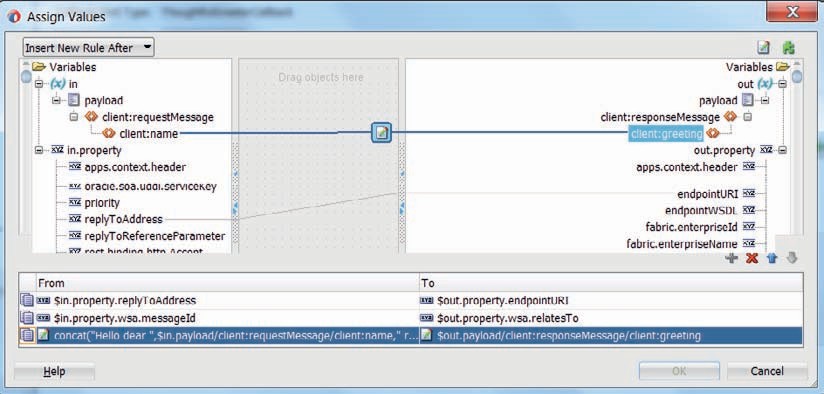


**图 12-5。** *ThoughtfulGreeterMediator 公开异步接口并使用*

*用于执行回调的 SOAP 引用绑定。*

打开“赋值”对话框，请参见图 12-6。将 *endpointURI*  属性（在 $out 变量中）设置为 $in 变量的 replyToAddress  *属性中的值（在*请求消息中作为标头接收）。将 *$in 变量中的* wsa.messageId 属性映射到 $out 变量中的 *wsa.relatesTo* 属性。此外，设置一个 XPath 表达式来派生响应消息中的 greeting 元素的值。

部署 SOA 组合。使用与之前相同的 SoapUI 测试请求来测试服务，并验证中介是否向回调（模拟）服务发送响应消息。



**图 12-6。** *通过在 $out 变量上设置 endpointURI 属性，响应为*

*引导至正确的回调服务终结点;relatesTo 属性使用传入的 messageId 进行设置，以允许使用者在请求和对回调服务的异步响应之间建立关联*

#### 使用服务总线公开异步服务接口

服务总线以与中介类似的方式实现异步接口：在初始单向传入对话之后，在管道中处理请求，并将该处理的结果（可能涉及对一个或多个业务服务的调用）发送到表示回调服务的业务服务。请求消息中的标头包含指示业务服务将响应发送到哪个端点所需的信息。

使用服务总线*实现* ThoughtfulGreeterService 的步骤相当简单。

创建新的服务总线项目。将 WSDL 和 XSD 导入到项目中。创造

基于 ThoughtfulGreeter *端口绑定*的代理服务和基于 *ThoughtfulGreeterCallback* 的业务服务。请注意，为业务服务设置的端点地址是无关紧要的，因为与中介一样，此端点将在运行时从服务使用者接收的 ReplyTo 标头属性进行设置。

创建一个管道，也基于 *ThoughtfulGreeter*。将代理服务连接到此管道，并将管道连接到业务服务，如图 12-7 所示。

管道从请求标头中收集 ReplyTo 地址和 MessageId，然后处理正文以准备响应，然后使用 Publish 操作调用

单向回调接口，使用适当的值操作标头后， 如图 12-8 所示。

###### 注意

*此 Publish 操作还可以调用服务总线代理服务，而不是调用传出业务服务，该服务使用自己的管道在分离*的*单独线程*上执行处理*。只要*

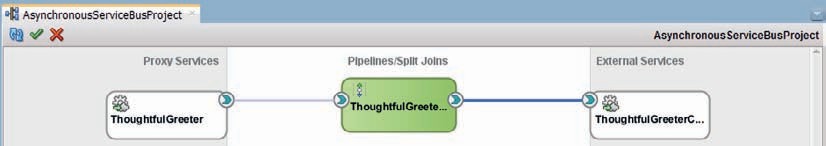
*随着 WS-Addressing 属性的传递，此管道可以路由到调用 Callback 接口的业务服务。*

第一阶段 GatherCallbackToandMessageId 包含将值从$header变量分配给局部变量的 Assign 活动：

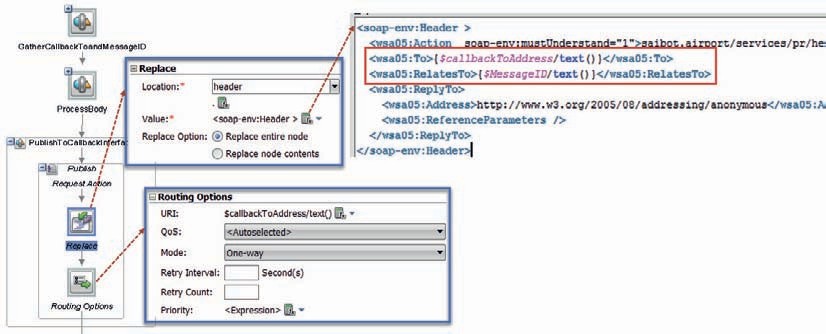
 $header/wsa05：ReplyTo/wsa05：Address/。=> callbackToAddress

和

 $header/wsa05：MessageID/。=> 消息 Id



**图 12-7。** *用于异步响应的 ThoughtfulGreeter 服务总线项目*



**图 12-8。** *通过三个步骤实现 ThoughtfulGreeter 功能的管道：*

*检索 WS-Addressing 属性，准备响应并发布到执行回调的业务服务*

请注意，在服务总线管道中，我们始终可以分别在 $body 和 $header 变量中访问请求消息中的正文和标头元素。

响应消息在第二阶段 *ProcessBody* 中使用 $body 变量上的替换活动准备，将节点内容替换为包含请求名称的问候消息 。

第三阶段通过对业务服务的发布操作发起回调。 发布操作配置为调用业务服务。它包含一个替换活动，用于使用收集的 WS-Addressing 信息替换标头变量中的整个节点。

“路由选项”活动是设置终结点地址的位置，发布操作使用该地址将其消息发送到该地址。URI 属性是根据我们之前基于请求标头设置的 $callbackToAddress 变量设置的。请注意，在业务服务上指定的端点地址将被完全忽略，因为它被路由中设置的值覆盖

期权活动。

这样就完成了 ThoughtfulGreeterService *的实现服务总线样式*。从 JDeveloper 运行它，然后使用用于测试 SOA 组合的相同测试项目从 SoapUI 调用它，但只为服务总线上公开的服务设置了终结点。当然，结果应该完全相同。

###### 提示

*我们倾向于将 ReplyTo 地址视为“返回发件人”位置，该位置应以某种方式与异步服务组件相关并位于异步服务附近。但是，我们也可以将其视为“转发地址”，即与调用服务的位置完全无关的服务终结点。*

此外，使用 WS-Addressing 标头属性 *RelatesTo* 和 *ReplyTo* 来指定消息 ID 或相关标识符以及异步响应应发送到的终结点当然是自定义的。为了在 BPEL 中自动处理异步交互，这是必需的。但是，你已了解了如何针对中介器和服务总线实现，我们已指定如何检索和分配这些值。这意味着，如果我们愿意，我们可以通过其他方式读取转发地址和相关标识符，而不是从 WS-Addressing 标头属性中读取。

### 与异步接互

与同步服务相比，公开和实现异步接口需要一些额外的东西。调用这些接口并为回调响应做准备是一个更大的挑战，正如我们接下来从 BPEL 组件和服务总线项目调用异步操作时所看到的那样。

#### 从 BPEL 组件调用异步服务 — 异步后端之上的同步接口

BPEL 是一种专为 Web 服务编排而设计的语言。这包括处理与异步 Web 服务交互的能力。WS-Addressing 标头的操作以及从异步服务返回到调用 BPEL 流程实例的异步 [回调] 响应的关联都由 BPEL 引擎处理。开发人员 只需处理被调用和接收的 partnerLink。

异步降压随着 BPEL 流程的发生而停止：BPEL 流程可以向其使用者公开一个同步接口，即使它在内部进行异步交互也是如此。当然，当异步响应需要很长时间才能到达时，同步会话 可能会超时。

我们演练了一个简单的 BPEL 流程示例，该示例公开了 *ThoughtfulGreeter* 的同步变体：它接收带有名称的请求消息，并返回带有问候语的同步响应。这个 BPEL 组件调用异步 *ThoughtfulGreetingService*，这是本章前面分别使用 BPEL、Mediator 和服务总线实现的组件之一。

步骤同样简单明了：

* + 创建一个新的 SOA 组合应用程序 *LessThoughtfulGreeterService。*
  + 为 LessThoughtfulGreeterService *创建或导入同步 WSDL 定义*

（当然，即使这个服务是同步的，也不能真正考虑

不那么深思熟虑，因为为了创建响应，它征集了所谓的

*周到的GreeterService*）。

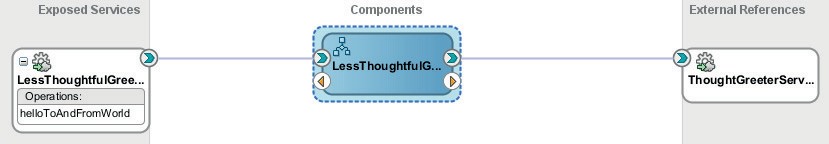
* + 为 ThoughtfulGreeterService 创建 Web 服务引用绑定。
  + 创建一个名为 *LessThoughtfulGreeterProcess*  的 BPEL 组件，该组件基于

*LessThoughtfulGreeterService.wsdl* 文档;将此组件作为 SOAP 公开

Web 服务。

* + 将 BPEL 组件连接到 *ThoughtfulGreeterService* 引用。

复合现在如图 12-9 所示。

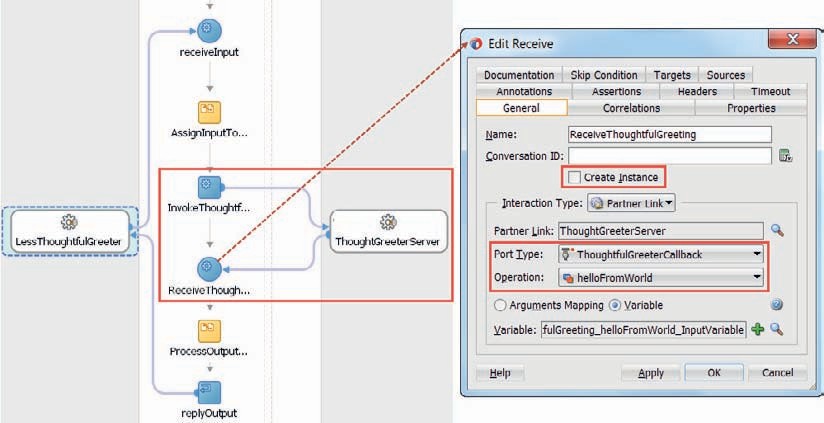


**图 12-9。** *公开由*

*异步后端 — 由 BPEL 组件处理同步-异步解耦*

打开 BPEL 组件的编辑器。添加一个调用活动，该活动*调用* ThoughtfulGreeterService 的合作伙伴链接 。将活动配置为*在* ThoughtfulGreeter  *portType 上调用 helloToWorld*  操作。为此调用创建输入变量。由于这是单向操作，因此无需配置输出变量。

随后，BPEL 组件将不得不等待异步响应的传入。为此，我们需要在*调用下方添加一个*接收*活动*，并将其连接到 *ThoughtfulGreeterService* 合作伙伴链接。但是，这次活动是在 *ThoughtfulGreeterCallback*  Port Type 中为 helloFromWorld *操作配置的*，如图 12-10 所示。



**图 12-10。** *配置处理异步接收的 Receive 活动*

*回调*

确保在接收配置中*取消选中*“创建实例”复选框：在本例中，我们希望通过现有实例中的合作伙伴链接接收消息，而不是创建新实例。取消选中此复选框意味着 BPEL 引擎在接收到异步回调消息时将尝试与正在运行的实例关联 — 基于 RelatesTo WS-Addressing 标头属性。

###### 注意

*当调用的异步服务不支持 使用 WS-Addressing Relate To 属性时，我们可以在回调消息中手动定义相关标识符，该标识符应用于将该消息与正在运行的实例相关联。下一 节将介绍这种关联机制。*

最后，添加*分配*活动以准备 *ThoughtfulGreeterService*  的输入，并 根据收到的回调消息设置 outputVariable。图 12-9 显示了完整的 BPEL 流程。

现在可以部署 SOA 组合并测试其（同步）服务接口。这将演示 BPEL 如何轻松地从异步对话风格适应同步对话风格。它也可以做相反的事情：发布一个异步接口，同时与 同步服务进行交互。

###### 提示

*没有理由在调用和接收活动之间不能有其他活动。当执行 BPEL 流程实例的线程到达接收活动时，该实例将被冻结或压缩并存储，然后释放线程。*

*当回调消息到达时，另一个线程被赋予恢复实例和执行接收活动的任务。*

#### 在服务总线中调用异步服务Invoking an Asynchronous Service in Service Bus

上一节描述了 BPEL 组件如何优雅地与异步服务交互。繁重的工作是由引擎完成的，作为开发人员，调用同步服务或异步服务之间几乎没有任何区别。

服务总线是无状态的，并且没有用于关联传入消息的相同功能。当回调消息进入服务总线运行时时，它与任何其他（请求）消息没有什么不同。回调消息需要由代理服务处理，就像任何其他消息一样。请注意，异步业务服务可以是 HTTP/SOAP，也可以是 JMS、EJB、AQ 等。

服务总线无法在同步接口的实现中调用异步服务。原始同步请求与异步业务服务回调消息的处理之间不能有任何链接。

在实现单向服务时，从服务总线调用异步服务是完全可能的。回调代理服务的终结点在 ReplyTo 标头属性中传递。此代理由异步服务调用，并可能继续单向服务实现的下一阶段。

如果服务总线应该虚拟化异步后端，它只能使用异步接口来实现，如图 12-11 所示;与 BPEL 不同，它不能从异步适应到同步。在处理回调消息的代理服务开始的处理结束时，我们需要调用由启动整个会话的外部使用者指定的回调服务终结点。这反过来又要求我们将原始调用中收到的 ReplyTo 和 MessageId 属性传输到异步业务服务，并希望在回调消息中接收它们。图 12-12 显示了我们必须在服务总线中执行的两步舞。

这意味着，为了使服务总线完全虚拟化异步业务服务，需要以某种方式将原始 ReplyTo 值（*图 12-12 中* ThoughtfulGreeterCallback 服务的地址）传递给业务服务随回调消息调用的代理服务。

似乎有两种主要方法可以应对这一挑战：

1. 让异步业务服务传递原始 ReplyTo 值，例如，将其包含在原封不动地传递的（自定义）标头属性中，或者将此值连接到消息标识符。
2. 让第一个代理服务将 ReplyTo 地址保存到（跨群集）缓存（与消息标识符关联为键），并让第二个（回调）代理服务使用回调消息中的 RelatesTo 属性作为键值从缓存中检索此 ReplyTo 地址 。

如果您可以控制异步服务，则第一种方法是最简单的。只需就调用程序可能传入的标头属性和异步业务服务达成一致

Client of One-Way Service

One-Way Proxy Stage1

SomeBusinessService (asynchronous)

CallbackProxy (Stage 2 of One-Way)

**Service Bus**

WS-Addressing RelatesTo

WS-Addressing ReplyTo, Messageld

**图 12-11。** *从服务总线项目调用异步业务服务;将*

*“下一阶段代理服务”的终结点作为 ReplyTo 标头属性中的回调服务*

ThoughtfulGreeter Client

WS-Addressing ReplyTo Messageld

helloTo ThoughtfulGreeterProxy

World (one-way)

helloTo World

ThoughtfulGreeterService (asynchronous)

ThoughtfulGreeterCallback

helloFrom World

helloFrom ThoughtfulGreeter

World CallbackProxy (one-way)

**Service Bus**

WS-Addressing ReplyTo, RelatesTo

WS-Addressing ReplyTo & originalReplyTo,

Messageld

**图 12-12。** *使用双单向在服务总线中虚拟化异步业务服务*

*代理服务*

会通过。如果无法以这种方式更改要虚拟化的异步服务，则可以 操作消息标识符（如果业务服务基于 MessageId 设置 RelatesTo 属性 ），也可以采用第二种方法。

调解器组件的情况大致相同。该组件也是无状态的，并且没有引擎级基础结构来处理异步回调关联。该 解决方案类似于服务总线所描述的解决方案。

### 索取机场服务建议书

赛博特机场的首席运营官对机场的运营顺利进行感到焦虑。不仅当赛博特机场本身与航空公司或在机场内或机场周围开展业务活动的其他方进行互动时，而且当各方在没有赛博特参与的情况下进行直接互动时。为了增加 Saibot 对乘客、业务合作伙伴，尤其是航空公司的吸引力，促进快速、高效、灵活和一致的业务运营至关重要。

希望业务顺利运行的一个领域是围绕飞机服务。每架飞机在抵达后和起飞前都需要一些服务。这些例子包括行李处理、加油、电力和供水、清洁（内部和外部）、小型维修和餐饮。许多航空公司都要求提供这些服务，活跃在塞博特机场的大量公司也提供这些服务（其中的一部分）。

Saibot率先组织了围绕这些服务的自动化电子交换，指定了一个Web服务接口，如果任何公司都想有资格为航空公司提供飞机服务，那么他们就必须提供该接口 。当承运商安排时

如果他们需要餐饮或清洁方面的帮助，他们将向 所有可以提供飞机设施的公司公开的 Web 服务发送提案请求。 Saibot 指定的服务是一种异步服务，承运人可以请求 有关航班的某些（组合）服务的报价。在飞机服务提供商必须经过任何内部处理以 准备对 RFP 的响应后，将向发起 RFP 的航空公司发出带有报价的回调 。

在本节中，我们将简要介绍 Saibot 的 IT 人员定义的服务接口。

接下来，我们将介绍 使用 BPEL 进行异步回调的 AircraftService 实现。然后，我们创建第二个 SOA 组合来充当提案的请求者。此组合还包含一个 BPEL 流程，该流程调用 *AircraftService* 提供程序 并等待异步回调到达。

最后，我们将这个 BPEL 流程扩展到同时调用多个

*AircraftServiceProviders* ，并并行等待他们的建议。

#### AircraftService 简介

*AircraftService 定义了两个 portType：一个由飞机服务提供商实现的“普通”端口类型*，以及将由该提供程序调用的回调 portType，因此必须由调用 AircraftService *并希望接收*异步响应的任何人实现。

*调用 requestProposal* 操作以向飞机服务供应商请求建议。该请求包含有关要提供的服务（燃料、清洁、餐饮、行李处理等）的信息，以及哪个航班的飞机类型、时间段、航班号、目的地、乘客人数以及报价的截止日期。回调接口中的单向 *submitProposal* 操作使用一条消息调用，该消息包含供应商的指示，他是否能够提供请求的服务、这些服务的价格是多少以及报价有效期。

#### 使用 BPEL 实现异步 AircraftService

机场人员提出的 *AircraftService*  接口实现的起点显然是 WSDL 及其关联的 XSD 文档。使用这些，我们可以创建一个 SOA 组合应用程序，它公开一个基于该接口的服务，该服务由遵循同一接口的 BPEL 组件实现。

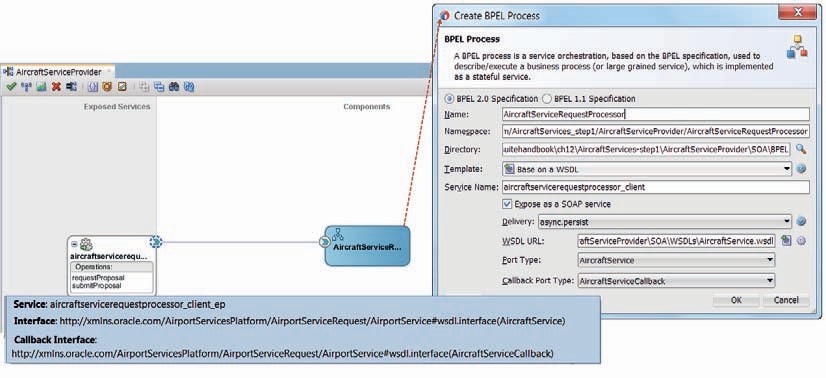
只需几个步骤，我们就拥有了复合应用程序的基础：

* 在 JDeveloper 中创建一个新的 SOA 应用程序（称为 *AircraftServices*），并使用名为 *AircraftServiceProvider* 的 SOA 组合（基于*空模板*）。
* 添加一个名为 AircraftServiceRequestProcessor *的 BPEL 组件*;该组件基于 AircraftService.wsdl 文档。BPEL 组件作为 SOAP 公开

复合级别的服务

SOA 组合如图 12-13 所示。BPEL 组件公开了单向接口，并调用回调接口来传递提案。

打开 AircraftServiceRequestProcessor  *组件的 BPEL 编辑器*。接收和异步调用（即回调）活动已经创建并连接到 PartnerLink。我们需要添加的只是一个 Assign 活动，以提供一些半有意义的内容



**图 12-13。** *基于 BPEL 的异步 AircraftService 接口实现*

用于回调消息和 Wait 活动 - 看起来不太急切，并且至少在一定程度上模拟现实生活。

这就是实现异步交互所需的全部内容。因为 BPEL 引擎负责异步细节，比如操作 WS-Addressing 标头，所以开发人员几乎不会注意到交互实际上是异步的，而不是直接的同步请求/回复。

您可以使用 EM FMW Control 部署复合并对其进行测试。不会有可见的响应，但通过检查流跟踪，您将能够验证是否发生了回调响应。将 SoapUI 与 mock 服务一起使用来实现回调接口并提供

正确的 WS-Addressing 标头属性，我们可以确定回调消息是否 正确返回。

#### 实现 AircraftServiceOrchestrator

航空公司或作为承运商和服务提供商之间中间商的一方必须实施一个协调器：一种协调寻找一方以提供所需飞机服务的机制。此机制必须调用*尽可能多的适用服务提供商公开的* AircraftService，处理他们通过异步回调提交的实际建议，密切关注截止日期并收集结果并做出或推荐选择。

我们已经了解了 BPEL 如何擅长调用异步服务和处理 传递响应的回调。处理多个并行活动并在一段时间内保持整体活动的状态绝对是 BPEL 所擅长的。因此， 我们将创建一个 *AircraftServiceOrchestrator* 复合来组织航空公司的 RFP 交互 。

此组合公开了具有两个操作的 Web 服务：*arrangeAircraftService* 和

*cancelAircraftService*。第一个发起并协调程序，要求飞机服务

供应商的建议并收集他们的回复。第二种方法用于中止程序， 将在后面的章节中讨论。

##### 创建 AircraftServiceOrchestrator SOA 组合

创建此复合的步骤与上一节中看到的步骤非常相似：

* 将名为 AircraftServiceOrchestrator  *的 SOA 项目添加到*应用程序 AircraftServices; 使项目基于*空模板*。
* 添加一个名为 AircraftServiceNegotiationProcessor 的 BPEL 组件*;该组件基于* AircraftServiceNegotiation.wsdl 文档。BPEL 组件已公开

作为复合级别的 SOAP 服务

我们希望 BPEL 组件能够调用飞机服务供应商，这些供应商公开了他们在*上一节中实现和部署的* AircraftService 的实现。显然，在现实生活中，这个服务会有很多实现，每个实现都有相同的接口和自己的端点。我们稍后会处理这种情况。现在，我们只从 BPEL 组件创建与我们所知道的一个 *AircraftService*  实例的交互，该实例就是我们在上一节中部署的实例。

将 SOAP 组件从组件面板拖动到引用泳道，以便为外部 *AircraftService* 创建引用绑定。将引用的名称设置为 *AirportServiceProvider*。选择 AircraftService.wsdl 文档。为“端口类型”和“回调端口类型”属性分别选择“*AircraftService”*和*“AircraftServiceCallback*”。按 OK。

JDeveloper 将显示一个关于 WSDL 文档的抽象性质的警告：因为它只描述了功能接口，而没有提供具有实际端点的端口，在该端口中，服务可用，因此我们还没有足够的信息来成功部署和运行我们的组合。但是，要继续实施，没有问题。确认 警告并继续。

将 BPEL 流程连接到 *AircraftServiceProvider* 引用绑定。SOA 组合如图 12-14 所示。

##### 实现 AircraftServiceNegotiationProcessor 组件

BPEL 流程接受一条请求消息，其中包含服务请求标识符和有关所需飞机设施的详细信息，并立即回复包含该标识符的响应，以及该流程汇总所有建议的最后时间。



**图 12-14。** *使用 AircraftService Providers 收集的 AircraftServiceOrchestrator*

*提供某种飞机服务的建议*

当此同步回复返回给调用者时，BPEL 流程将调用一个 *AircraftServiceProvider*，传递飞机服务请求的详细信息以及 RFP 标识符和响应的截止日期。此调用是单向的;它启动一个异步会话，该会话以服务提供商的回调结束。但是，如果在截止日期之前未收到异步响应，则该过程将停止等待并立即完成。在这一点上，我们专注于异步交互和

wait-until-deadline-or-callback，BPEL 流程实际上并不收集和进一步处理 提案。

打开 BPEL 编辑器，实现如上所述的流程。*接收*和*答复*活动已创建。定义两个全局变量 - xsd：string 类型的 *RFPIdentification* 和 xsd： dateTime 类型的 Deadline。*在 Receive 之后*添加一个名为 SetGlobals 的 Assign 活动。将这两个变量设置为：

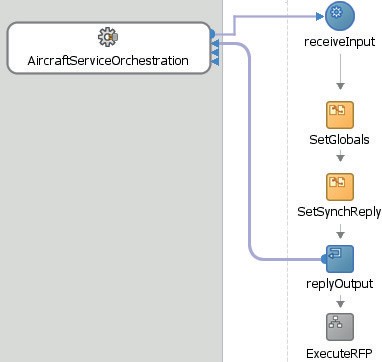
* + RFPACS<ServiceRequestIdentifier><DDMMYY>
  + 实际服务交付日期前 36 小时 - 使用表达式完成：xp20：add-dayTimeDuration-to-dateTime（$inputVariable.part1/

ns3：ServiceDeliveryDate， '-P1DT12H'）;出于测试目的，您也可以只设置 当前时间加 5 分钟

接下来，添加一个 Assign 活动以设置同步响应的 *outputVariable*。此变量包含服务请求标识符和收集建议的截止时间。

最后，在 Reply 活动下添加一个名为 *ExecuteRFP*  的作用域。这是实际收集提案的地方。BPEL 流程现在如图 12-15 所示。

接下来，添加一个 Invoke 活动 - 连接到 *AircraftServiceProvider*  合作伙伴链接的单向接口;让此活动使用本地输入变量。此外，添加一个 Receive 活动，用于使用来自 AircraftServiceProvider  *合作伙伴链接的回调消息*。这就是 BPEL 的用武之地



**图 12-15。** *AircraftServiceNegotiationProcessor 的同步第一部分*

真正闪耀的是：接收回调消息（即异步响应）就像接收同步回复一样简单。创建另一个局部变量，用于保存回调响应。

在请求之前添加一个名为 *SetInputForRFP* 的 Assign 活动。在此 Assign 步骤中，在 inputVariable 和*两个全局变量（*RFPIdentifier *和* Deadline*）之间进行必要的映射*，以调用 *AircraftServiceProvider*。

BPEL 流程的第二部分如图 12-16 所示。

此时，我们几乎可以部署和运行复合。在后面的部分中，我们将添加当前缺少的“等待截止日期”功能。此时，我们将添加对第二个 *AircraftServiceProvider 的并行调用*。

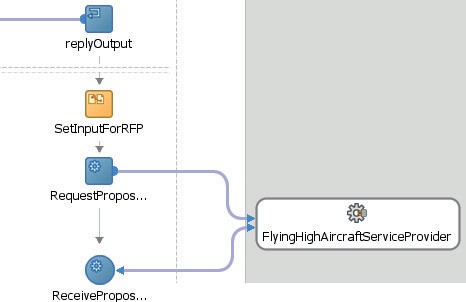
##### 完成、部署和运行 AircraftServiceOrchestrator

该组合尚未准备好进行部署。请记住，*AircraftServiceProvider*  的引用绑定是如何基于我们从文件系统中选择的抽象 WSDL 的。这个 WSDL 没有提供具体的端口和端点，这意味着我们的组合知道服务是什么样子的，但不知道在哪里可以找到它。在成功部署之前，我们必须指定服务的确切地址。

在后面的章节中，我们将讨论如何使用配置计划，以便在部署期间为外部服务提供特定于环境的端点。现在，让我们对 composite.xml 文件进行简单的修改。

首先，打开*本章前面部署的* AircraftService 组合的 WSDL 文档。您可以从 EM FMW 控件执行此操作。在 WSDL 文档中找到 Port 元素。*实时 WSDL 文档中 Service 和 Port*  元素的 name 属性组合提供了 binding.ws 元素的 port *属性*中 wsdl.endpoint 的值（图 12-17）。binding.ws 元素的 location 属性是根据实时 WSDL 的 port 元素*中 soap：address 元素中的* location 属性设置的*，并指示*实际的端点地址。

当 composite.xml 文件中的 binding.ws 元素完成后，我们可以将该组合部署到 SOA Suite 运行时。部署后，我们可以调用 *arrangeAircraftService* 操作来启动程序，以招募飞机服务提供商来帮助特定航班的燃料、清洁、餐饮和行李处理。



**图 12-16。** *与单个 AircraftServiceProvider 的异步交互*



**图 12-17。** *从.xml*

*AircraftServiceProvider 的实时 WSDL 文档*

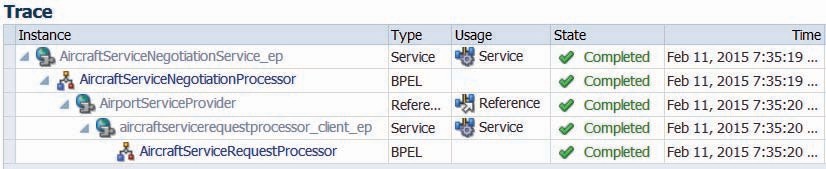
此操作请求的跟踪流可能如图 12-18 所示。

#### 使用 ServiceBus 实现异步 AircraftService

可以使用服务总线创建 AircraftService *的第二个*实现。 在本章前面，我们了解了在基于原始请求中的 WS-Addressing 标头设置终结点地址后，如何使用业务服务的标注在服务总线中实现异步接口。 步骤如下：

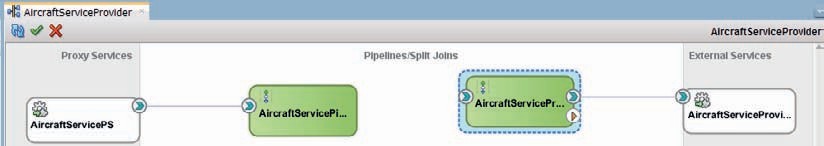
* + 使用服务总线项目 *AircraftServiceProvider 创建新应用程序。*
  + 添加 AircraftService.wsdl 和关联的 XSD 文档。
  + 为这两种端口类型向 WSDL 添加绑定。
  + 为单向端口类型*创建代理服务* AircraftServicePS 并生成关联的管道;将终结点设置为 /*FlyingHighAircraftService/*

*AircraftServiceProvider。*



**图 12-18。** *对 AircraftServiceOrchestration 上的操作 arrangeAircraftService 的调用*

*服务触发了 AircraftNegotiationProcessor 及其与 AircraftServiceProvider 的异步交互;来自 AircraftServiceProvider 的回调消息由 BPEL 运行时框架顺利反馈到 AircraftNegotiationProcessor 的正确实例中*



**图 12-19。** *实现异步 AircraftService 的服务总线项目 - 使用*

*单向请求和延时回调*

* 根据回调端口类型创建业务服务（*AircraftServiceProviderCallbackBS）。* 此外，根据

单向接口也由代理服务和另一个管道实现。 在第二个管道中添加发布活动，并将其发布到业务服务及其 *submitProposal* 操作。

服务总线项目如图 12-19 所示。

第一个管道会将其请求发布到第二个管道。通过此发布操作，将立即释放处理传入请求接收的线程。

第二个管道包含三个阶段，如图 12-20 所示：

1. 在局部变量中收集 WS-Addressing 详细信息。
2. 准备包含回调消息的$body。
3. 发布到回调业务服务;但是，在实际调用业务服务之前，我们会动态设置此调用的端点，使其路由到原始请求消息的 WS-Addressing 标头中找到的回调端点，而不是回调管道。

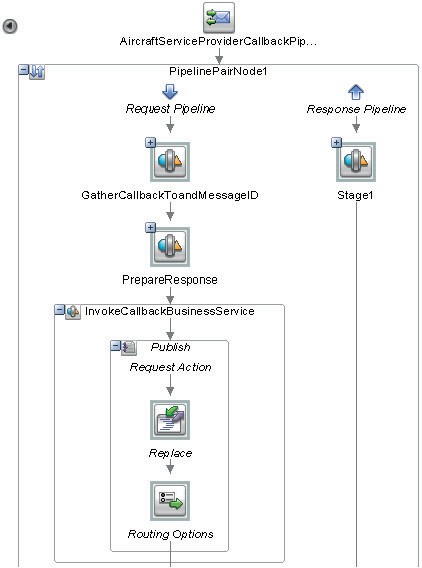
有关确切实施的详细信息可以在线找到。

我们可以使用 SoapUI 测试这个异步服务，就像我们之前看到的那样。

#### 扩展 AircraftServiceOrchestrator

我们现在有两项服务可供调用，大概是由活跃在赛博特机场的两家不同公司提供的。让我们扩展 *AircraftServiceOrchestrator* 来支持这种情况。这意味着我们的 BPEL 流程将有一个 Flow 活动。这是一个标准的 BPEL 活动，支持并行活动。在这种情况下，Flow 中的两个分支都将对 AircraftService  *的单向接口*执行 Invoke 以请求有关我们飞机的帮助，然后它们都将等待异步回调。

稍后，我们将建立截止时间观察，以确保 BPEL 流程等待来自飞机服务提供商的回调消息的时间不会超过设定的截止时间。我们甚至可以进一步扩展此示例，以动态派生要调用的服务列表，而不是 仅使用这两个硬编码的服务。



**图 12-20。** *回调管道的实现分为三个阶段：提取 WS-*

*寻址标头、撰写回调消息并将回调消息发布到动态 选择的端点*

**将基于异步服务总线的 AircraftService 集成到复合中** 首先，我们必须确定*从服务总线公开的* AircraftServicePS 的终结点。导航到“应用程序服务器”窗口，然后打开集成 WebLogicServer 的“服务总线”节点。在此节点下，展开 *AircraftServiceProvider* 的项目文件夹。右键单击 *AircraftServicePS* 节点，然后从上下文菜单中选择“打开”。

WSDL 文档从运行时检索并在 JDeveloper 中打开。端口中 address 元素中的 location 属性生成终结点。加？WSDL 复制到此 URL 以获取 WSDL 位置。

打开 AircraftServiceOrchestrator  *复合的 SOA 复合编辑器*。在“外部引用”泳道中添加 SOAP 引用绑定。将名称设置为

*FlyingHighAircraftServiceProvider*。将上一步中确定的端点粘贴到 WSDL URL 字段中。端口类型是自动选择的。按 OK 创建此 Reference 绑定。

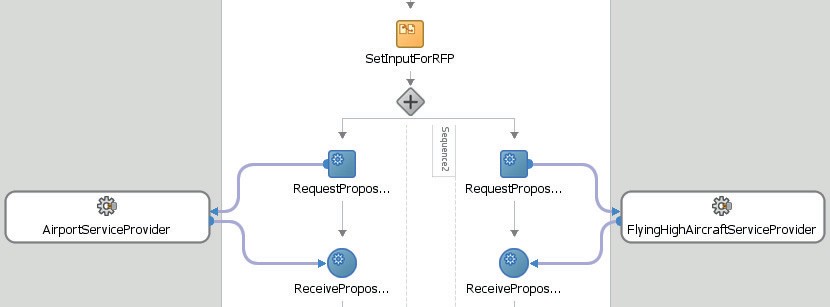
将 BPEL 流程连接到新的引用绑定;毕竟，我们打算从 BPEL 流程中调用此引用 。

##### 从 BPEL 流程调用第二个 AircraftService 提供程序

并行活动是通过 Flow 活动在 BPEL 流程中实现的。流包含两个或多个同时执行的 Sequence 活动。这意味着，例如，在 Flow 活动的两个分支中，可以对外部服务进行异步调用，两个分支同时等待回复。显然，这比在发出第二个请求之前等待第一个响应进来更有效。对于流，最慢的服务响应所需的时间决定了整个流的处理时间，而不是所有调用加在一起的时间。

打开 *AircraftServiceNegotiationProcessor* BPEL 流程。打开范围 *ExecuteRFP*。在 *SetInputForRFP 分配*活动之后，立即将“流”活动添加到流程中。将 invoke 和 receive 活动（连接到 *AircraftServiceProvider* 合作伙伴链接）拖动到流中的第一个分支。将调用和接收活动添加到第二个流分支。将这些活动与 *FlyingHighAircraftService* 合作伙伴链接相关联。可用于调用此新合作伙伴链接的输入变量与用于调用 *AircraftServiceProvider* 合作伙伴链接的输入变量相同。该过程的第二部分现在有两个并行的活动分支，每个 *AircraftService*  提供程序一个分支，请参见图 12-21。

此时，只有当整个流程完成时，BPEL 流程才能继续处理*流*活动之后的活动。也就是说，当从两个服务提供商收到两个回调建议时。我们需要添加一种机制来监控我们的截止日期：当我们达到截止日期时，我们必须继续前进，停止等待这些回调消息到达。



**图 12-21。** *具有并行分支的流活动，以与两个分支进行异步交互*

*同时使用不同的服务提供商*

### 在 BPEL 流程中引入一些异步交互

BPEL 流程可以参与各种形式的异步活动。我们已经了解了该进程如何等待回调消息，该消息实际上是来自异步 合作伙伴的回复 [链接]。这样的回调消息是意料之中的，即使我们不知道它何时到达，甚至不知道它是否会到达。

我们可以在 BPEL 流程中引入的另一个异步活动是后台计时器，它会运行一段时间或直到某个截止日期。当计时器触发时，将执行指定的操作。*AircraftServiceOrchestrator*  的等待时间不会超过决定服务提供商的截止时间。

以类似的方式，我们可以有后台消息侦听器来监听请求消息，而不是响应 BPEL 流程中先前服务调用的回调响应 ，而是意外的新消息。

在本节中，我们将允许最初调用 AircraftServiceOrchestrator 复合的航空公司在 *AircraftServiceOrchestrator*  服务上调用取消操作 ，以停止 RFP 进程。这应该会导致向所有飞机服务提供商发送类似的消息，通知他们取消 RFP。然而，这最后一个电话超出了本章的范围。

按照类似的思路，我们可以让承运人询问提案的状态，甚至提供其他信息，例如更新的乘客人数或更改的时段时间。

调用 SOA 组合的运行实例的能力非常特殊。它依赖于 BPEL 中一种称为相关性的机制。通过关联，请求消息可以与现有的 BPEL 流程实例相关联，而不是像往常那样启动新的实例。

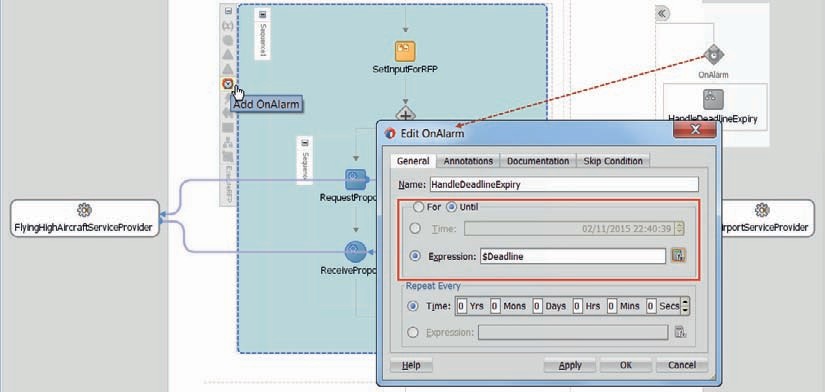
#### 将截止时间监视器添加到 AircraftServiceOrchestrator

BPEL 中的范围就像一个封装的单元，除了一组活动之外，它还可以有自己的局部变量、错误处理程序、补偿处理程序以及（异步）事件处理程序。事件处理程序指定一个异步代理，该代理在作用域运行期间运行，并且可以执行以下两项操作之一：等待特定时刻然后执行操作，或者接收特定 类型的传入消息并对其进行操作。

在 AircraftServiceOrchestrator BPEL 流程中*选择*包含对 *AircraftServiceProviders*  的两次调用的流的范围。在示波器左侧的图标面板中，单击闹钟图标，添加OnAlarm事件处理程序，如图12-22所示。JDeveloper 将处理程序添加到作用域中。单击 OnAlarm 活动以打开其编辑器。将单选按钮设置为“直到”，以在特定时刻触发警报，并将 $Deadline 指定为应触发警报的时间表达式。将事件处理程序中的范围重命名为 *HandleDeadlineExpiry*。作为处理截止期限到期的第一种也是最终粗略的方法，请将*退出*活动添加到此范围。请注意，只有当并非所有飞机服务提供商在截止日期之前发送其提案回调消息时，OnAlarm 处理程序才会被激活。

为了查看 OnAlarm 的运行情况，您需要稍微更改截止时间值和操作，或者使 *AircraftServiceProvider* 速度稍慢一些。

重新部署 *AircraftServiceOrchestrator* 复合后，我们可以尝试添加的逻辑。调用 *arrangeAircraftService* 操作，并可能提前截止时间。如果 OnAlarm



**图 12-22。** *为要触发的作用域 ExecuteRFP 配置 OnAlarm 事件处理程序*

*当达到最后期限时*

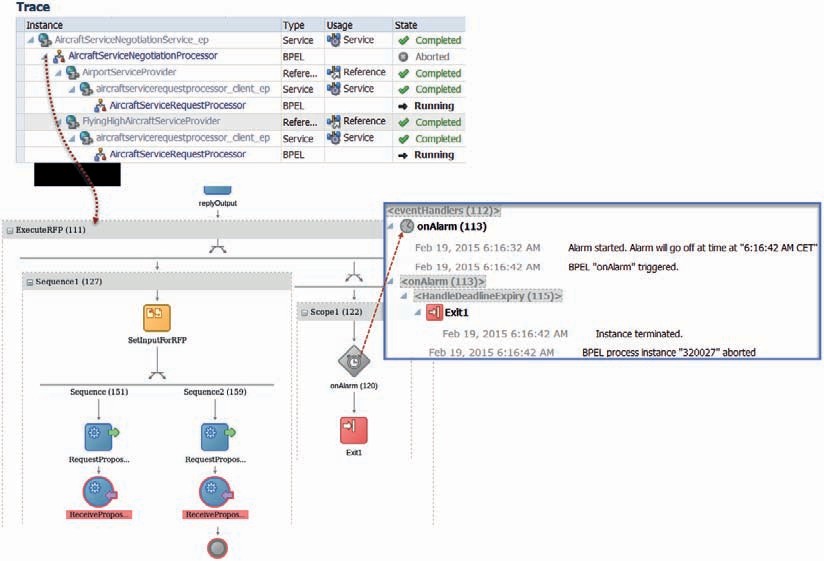
触发触发时，流跟踪和审核流将指示这一点。请参见图 12-23。注意：由于我们没有采取任何措施来通知 *AircraftServiceProviders* 截止日期到期，因此他们的流程实例正常完成，即使由于终止正常处理的警报，协商者无法处理他们的响应。

#### 使用关联向 AircraftServiceOrchestrator 添加取消操作

BPEL 在 SOA 套件中的所有服务引擎和语言中都具有独特的功能：BPEL 流程甚至可以在开始运行之后接收，这不仅仅是处理对服务调用的同步甚至异步响应。大多数计算机程序都是由原始调用启动的，一旦完成，就会返回结果——在运行时，它们不能轻易地或根本无法从外部访问。BPEL 流程可以公开多个操作，其中一个操作将启动实例，而其他操作则将消息馈送到 正在运行的实例中。

我们已经了解了 BPEL 流程在返回同步或异步响应消息后如何继续运行。对于这种特殊行为，我们现在添加了接收后续消息的功能 - 通过显式等待消息到达或将它们作为未经请求的事件处理。在这两种情况下，此功能的关键要素都是称为 correlation 的 BPEL 机制，即将传入的消息与可能正在运行的实例之一进行匹配。

可以使用 Receive 活动将消息接收到正在运行的实例中。另一种方法是通过一个 onMessage 事件处理程序，我们可以将它附加到 BPEL 流程中的一个作用域。



**图 12-23。** *之后 AircraftServiceOrchestrator 的消息流跟踪和审核跟踪*

*异步 OnAlarm 事件处理程序已触发，并缩短了作用域的执行*

###### 提示

*正在运行的 BPEL 流程实例处理 传入消息的第三种方法是在 Pick 活动中。BPEL Pick 活动与任何其他 BPEL 活动一样包含在序列中。它也处理事件：它指示 BPEL 引擎暂停 BPEL 流程实例，直到发生潜在的许多事件之一。与事件处理程序的情况一样，事件要么是经过特定持续时间，要么是接收到特定消息。与事件处理程序不同*

*在作用域的整个生命周期内，它们在后台处于空闲状态， 仅当它们正在侦听的事件发生时才影响 BPEL 进程，Pick 活动会使该进程（或者至少是它所在的分支，因为公共 Flow 父级中可能存在其他分支）停止，直到为其配置的一个事件发生。 除非发生其中一个 Pick 事件，否则不会执行 Pick 活动之后的任何活动 。*

取消可以在任何时候到达，从最初提交飞机服务请求的那一刻起，直到 RFP 完成并选择服务提供商。它作为 Web 服务请求进入我们的 SOA 基础架构，该请求应该被馈送到 为该特定飞机创建并仍在运行的 BPEL 流程实例中。BPEL 流程实例在相关事件处理程序中接收请求，然后应该完全完成该实例 。

取消 Web 服务请求需要准确指定必须取消哪个飞机服务请求，就像我们通过电话或电子邮件取消此类较早请求时必须做的那样 。服务请求由初始请求消息中的标识符标识 。或者，我们也可以在 BPEL 流程中确定一个标识符，并在 同步回复中返回它。

SOA Suite 运行时使用此标识符将传入的取消请求与正确运行的 SOA 组合应用程序实例相关联。这样做的前提条件是，使用的标识符在所有流程实例中都应该是唯一的。在入站请求和现有实例之间进行这种匹配的机制称为关联。

**AircraftServiceOrchestrator 的关联**

关联通常处理以下场景：请求消息到达 SOA 套件。它不打算启动新的复合应用程序实例。相反，它需要路由到已经运行的实例。由引擎来找到正确的实例来传递消息。在这种情况下，需要将取消 RFP 的请求传递给为 该特定飞机服务请求创建的实例。有关说明，请参阅图 12-24。

当然，引擎需要能够从请求消息中提取某种标识符，以便将该消息与正在运行的实例相关联。在我们的示例中，最初*使用包含* ServiceRequestIdentifier  *的消息调用 AircraftServiceOrchestrator，出于本讨论的目的，我们假设该消息在*对 AircraftServiceOrchestrator *的所有调用中是唯一的。*

有关服务请求的任何后续交互都应包含此标识符，并且 AircraftServiceOrchestrator *的每个正在运行的实例*也应通过该标识符进行标识。它是将新请求消息与正在运行的实例相关联的链接引脚。

**197**

**图 12-24。** *传入消息与正在运行的复合应用程序之间的关联*

*通过 BPEL 组件的实例*



ServiceRequest Identifier #361

**SOA Suite**

**...**

Cancellation request

**AircraftServiceN...**

**AircraftServiceN...**

**AircraftServiceN...**

**?**

**AircraftServiceN**

**183**

**361**

**23**

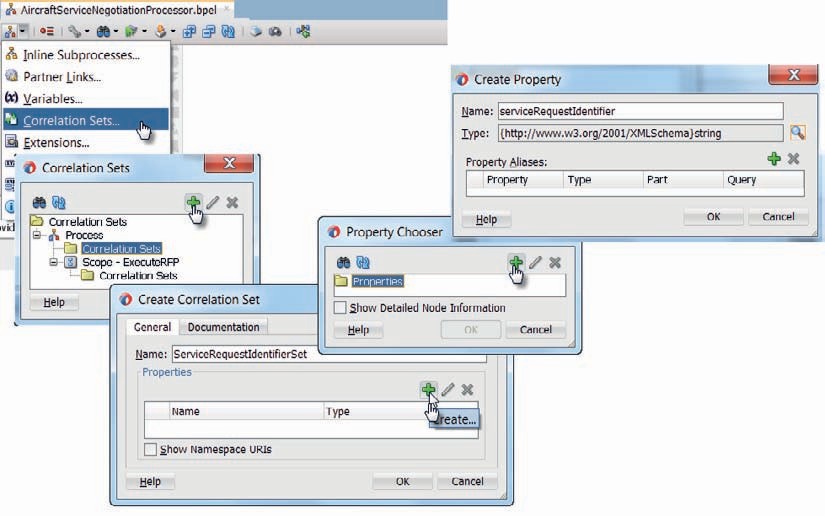
组合应用程序实例的关联是建立在 BPEL 流程实例的关联之上的;没有 BPEL 服务组件的复合应用程序不支持关联 。需要与正在运行的复合应用程序实例关联的消息需要 发送到 BPEL 组件中，因此需要发送到连接到 BPEL 组件的应用程序公开的服务 。

为了使关联机制起作用，我们必须配置 BPEL 流程以将 *serviceRequestIdentifier* 识别为该关联键。

可以使用关联集来标识 BPEL 流程的实例以进行关联。这种关联集是一个或多个属性的组合，其方式与复合主键或唯一键数据库约束非常相似。属性在进程级别定义，属于特定类型，并映射到进程发送或接收的消息中的值。BPEL 流程可以有多个关联集，就像数据库表可以有多个 唯一键一样。

*AircraftServiceNegotiationProcessor* 具有由单个属性组成的单个关联集。我们将此集合称为 *ServiceRequestIdentifierSet*。我们可以从 BPEL 流程编辑器顶部栏中的下拉菜单中创建一个关联集，如图 12-25 所示。 添加一个名为 *ServiceRequestIdentifier* 的属性，其类型为 String。

在此，我们指定 *AircraftServiceNegotiationProcessor* BPEL 流程的实例可以通过仅*使用此*属性的值的关联集 ServiceRequestIdentifierSet 进行唯一标识。



**图 12-25。** *创建关联集 ServiceRequestIdentifierSet 和*

*AircraftServiceNegotiationProcessor 的 ServiceRequestIdentifier 属性*

但是，该属性的价值是多少？何时以及如何确定？该属性与 BPEL 流程中的变量或发送到流程或从流程发送的消息有何关系？

关联始终发生在消息交换的上下文中。当 BPEL 流程接收消息（onMessage 和 Receive 活动）或发送消息（Invoke 和 + Reply）时，相关性都会发挥作用。只有在这种时候，引擎才需要在附加到消息交换的关联集中建立属性值 。

属性的值与在此类交换时刻发送到进程或从进程发送的消息的内容相关联。例如，*serviceRequestIdentifier*  属性从传出的 ArrrangeAicraftServiceResponseMessage 中获取其值*，该传出 ArrrangeAicraftServiceResponseMessage*  是从第一个同步 Reply 活动中的进程返回的。当取消消息交换发生时，该属性将从传入的 *CancelAircraftServiceRequestMessage 中获取其值*。

属性与特定消息交换之间的这些关联是使用属性别名定义指定的。BPEL 流程可以包含一个或多个属性别名，这些别名将属性映射到特定的消息部分，准确地说，可以包含一个特定的 XPath 表达式，以便从该消息部分中提取值。此消息部分通过流程中的 partnerLink 之一在交换中使用。

对于 *AircraftServiceNegotiationProcessor*，我们将有两个属性别名，因为 *ServiceRequestIdentifier* 与两个消息交换（初始飞机服务请求和取消）相关联。这些属性别名可以从 编辑属性窗口创建，可以使用关联集窗口上的编辑属性图标打开该窗口。如图 12-26 所示。在 Query 字段中，我们必须指定 XPath 表达式来检索



**图 12-26。** *AircraftServiceNegotiationProcessor 的关联定义：*

*关联集、属性和两个属性别名*

属性的值。提示：按 ctrl-空格键会显示要添加到 XPath 表达式的可用 XML 元素列表。

流程实例的标识（关联集中的值）只建立一次，显然，因为以后不会更改。通过初始化关联集并捕获该时刻集合中属性的值来建立标识。在我们的示例中，当标记为 *arrrangeAicraftService* 的同步回复操作发送 *ArrrangeAicraftServiceResponseMessage*  时，会发生这种情况。*响应消息中* ArrrangeAicraftServiceResponseMessage  *元素中 ServiceRequestId 元素*的值被读取并设置为 ServiceRequestIdentifier *的值，该值*对于 BPEL 流程的实例永远不会更改。

在每次后续消息交换中，可以将关联集中确定的流程实例的标识与从传入消息中提取的值进行比较。这允许引擎将传入消息链接到具有相同关联集值的实例。

关联过程中的步骤如下：

* 1. 同步 Reply 活动启动关联集，并从响应消息中提取值并用于设置实例标识符。
  2. ServiceRequestIdentifierSet *关联集*的值基于为该消息定义的属性别名从传入的取消请求消息中提取，并与所有正在运行的实例的标识符进行比较，以查找匹配的实例。

使关联像这样工作的最后一步是配置发送（回复） 和接收（onMessage 处理程序）需要关联的消息的活动。

首先是 Reply 活动。此活动很特殊，因为它需要实例化 关联集。双击“回复”活动打开编辑器。转到“关联”选项卡。单击绿色加号图标以添加与此消息交换关联的关联集。 选择 *ServiceRequestIdentifierSet*。您需要将 Initiate 设置为 *yes* ，以指示此 Reply 步骤是实例化此关联集并设置此流程实例的标识符的时刻 。图 12-27 演示了这些步骤。

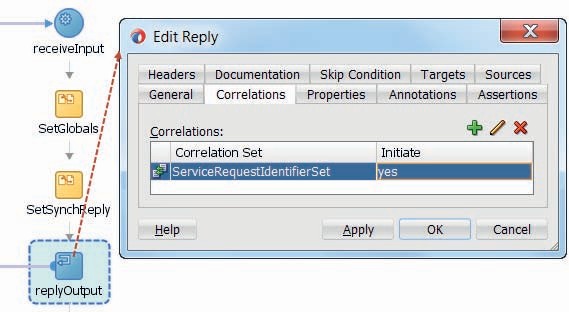
通过这个定义，我们确保在执行此 Reply 活动时，为该 BPEL 流程的实例分配一个可用于关联目的的标识。

顺便说一句，BPEL 流程可以通过具有不同属性和不同值的多个关联集来具有多个身份，并且可以在不同的时间点建立。例如，*AircraftServiceNegotiationProcessor* 可以具有第二个关联集，该关联集还通过 flightNumber 和插槽的日期和时间的组合来标识飞机服务请求。

##### 飞机服务请求取消的相关性

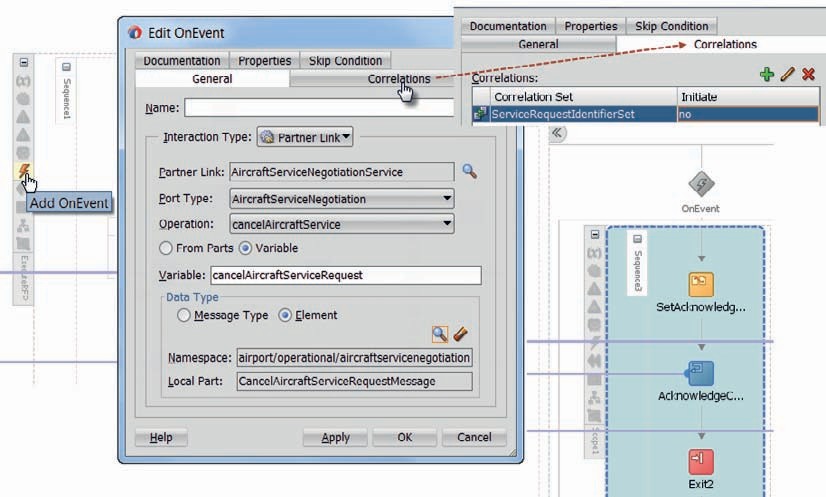
我们已经为接收飞机服务请求的取消请求奠定了基础。我们配置了一个关联集，并确保在进行同步回复时确定实例标识。

接下来，我们必须将 on [message] 事件处理程序添加到作用域 *ExecuteRFP*，以便在对 AircraftServiceNegotiation 服务调用 cancelAircraftService 操作时*处理* CancelAircraftServiceRequestMessage  *的接收*  。然后，我们需要配置这个 onMessage 处理程序来支持此消息交换的关联。双击 on 事件图标以配置它将实现的消息交换。图 12-28 演示了这些步骤。



**图 12-27。** *配置关联集 ServiceRequestIdentifierSet 及其启动*

*“答复”活动*



**图 12-28。** *为 onMessage 分支配置消息交换和关联*

*将处理取消请求*

开启（消息）事件活动中涉及的合作伙伴链接是 *AircraftServiceOrchestration*。

与此伙伴链接关联的端口类型中的相关操作是 *cancelAircraft* 操作。*根据*请求消息所基于的 *CancelAircraftServiceRequest*  元素，将 （input） 变量指定为 cancelAircraftServiceRequest。

接下来，单击“关联”选项卡。与此事件活动有关的唯一关联集是 *ServiceRequestIdentifierSet*。不应启动它，因为这已经 通过从 Reply 活动发送的传出消息发生;对于取消，我们将使用当时在关联集中分配给属性的值来与 传入取消消息的属性值相关联。

若要添加一定程度的适当取消处理，请向 onEvent 处理程序*内的作用域*添加回复活动。此活动提供对取消请求的同步回复;它与 *AircraftServiceOrchestration* 合作伙伴链接相关联。生成一个局部变量来保存响应，并添加一个*分配*活动来设置此变量的值，并具有某种形式的取消请求确认。然后，作为一种非常粗略的方法，现在就足够了，添加一个*退出*活动来终止 *AircraftServiceNegotiationProcessor 的实例*。

部署 AircraftServiceOrchestrator 复合。在其 arrangeAircraftService 操作上调用服务。然后，在该调用后不久，再次调用该服务，这次是在其 cancelAircraftService 操作上，传入与第一次调用中相同的 ServiceRequestId 值。这应导致 AircraftServiceOrchestration 进程实例被启动，并被取消请求中断和中止。图 12-29 显示了此流程实例的流跟踪和审计跟踪。请注意对 AircraftServiceOrchestration\_ep 服务绑定的第二次调用;此实例不会启动复合实例的新实例，而是根据 ServiceRequestId 与正在运行的实例关联并馈送到正在运行的实例中。



**图 12-29。** *AircraftServiceOrchestration 组合的流跟踪和 BPEL 审计跟踪*

*通过相关服务调用取消的实例*

##### 关联和异步服务调用

关联是 SOA Suite 运行时用来将传入消息与现有复合应用程序实例进行匹配的主要机制。然而，当我们引入对异步服务的调用的实现时，我们没有讨论相关性，即使异步服务（如 AircraftServiceProvider）的响应以对 AircraftService *回调 portType* 的服务调用的形式返回，由 *AircraftServiceNegotiationProcessor* 通过 partnerlink 处理。

这样做的原因是 BPEL 引擎使用 WS-Addressing 标准在后台自动处理这个问题。在其他场景中，我们必须自己实现/配置它，因为我们与之交互的组件不是 BPEL 组件。我们不需要对 BPEL 流程、组合定义或 WSDL 文件进行任何更改，就可以利用 WS-Addressing 方法在 BPEL 流程实例和它们调用的异步服务之间建立关联。BPEL 引擎运行时框架将向调用异步 Web 服务时发送的 SOAP 消息添加标头。这些标头（基于 WS-Addressing 规范）包含端点位置（回复地址），前者指定 BPEL 客户端侦听回调消息的位置，后者是发送请求的 BPEL 流程实例的唯一标识符。

当异步服务通过调用回调服务发送响应时，它可以使用 WS-Addressing 标头中的信息将响应定位到正确的客户端。当异步服务本身就是一个 BPEL 流程时，就像我们的 *AircraftService*  提供商一样，BPEL 引擎会自动利用标头，这对我们作为开发人员来说是完全透明的。

在几种情况下，当我们调用异步服务时，内置的、默认的 WS-Addressing 关联机制是不够的。其中一种情况是异步 Web 服务提供程序不支持 WS-Addressing，并且需要关联才能将响应消息映射到流程实例。另一种情况是更复杂的对话模式，该模式涉及两个以上的通信伙伴，以及收到开始对话的原始呼叫的伙伴未返回的最终响应。

请参阅 FMW 文档，以获取有关 WS-Addressing、它在 BPEL 引擎中的使用方式以及使用 OWSM 日志记录策略或 TCP/IP 侦听器检查 SOAP 消息（和 WS-Addressing 标头）内容的方法的更多信息。

#### 从正在运行的 AircraftServiceOrchestrator 中检索当前 RFP 状态

例如，一个 BPEL 流程不能只接收一次相关的消息，以结束该流程，它还可以接收多个请求并为每个请求返回同步响应。在本节中，我们将扩展该过程以支持航空公司检索

业务流程协调程序仍在进行协商时 RFP 的状态。我们将利用从流程中的每个并行分支调用的 BPEL 内联子流程，将飞机服务提案添加到所有提案的全局集合中。然后，我们将向 ExecuteRFP  *作用域添加第二个 OnEvent 处理程序，该处理程序*接收另一条相关消息，用于*我们将添加到 WSDL 中的 portType* 的 retrieveStatus 操作。

##### 扩展服务接口定义

打开 AircraftServiceNegotiation.xsd 文档。它包含在 RetrieveStatusAircraftServiceNegotiationResponseMessage 中使用的 proposalsListType。RetrieveStatusAircraftServiceNegotiationRequestMessage 包含一个 ServiceRequestId 元素， 现在应该有点熟悉了。

打开 AircraftServiceNegotiation.wsdl 文档。在 portType 中添加两条消息和一个操作：

 <wsdl：消息名称=“RetrieveStatusAircraftServiceNegotiationRequestMessage”>

<wsdl：part name=“part1” element=“acsn：RetrieveStatusAircraftServiceNegotiationRequestMessage”/>

</wsdl：消息>

<wsdl：消息名称=“RetrieveStatusAircraftServiceNegotiationResponseMessage”>

<wsdl：part name=“part1” element=“acsn：RetrieveStatusAircraftServiceNegotiationResponseMessage”/>

</wsdl：消息>

<wsdl：操作名称=“retrieveStatusAircraftServiceNegotiation”>

<wsdl：input message=“tns：RetrieveStatusAircraftServiceNegotiationRequestMessage”/>

<wsdl：输出 消息=“tns：RetrieveStatusAircraftServiceNegotiationResponseMessage”/>

</wsdl：操作>

##### 扩展 AircraftNegotiationProcessor 进程

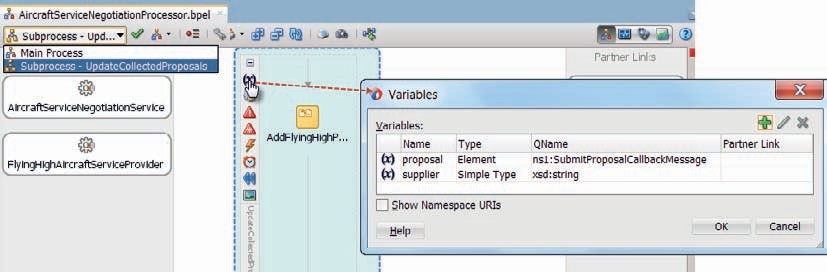
打开 AircraftNegotiationProcessor *的 BPEL 编辑器*。根据 XSD 文档中的 proposalsListType 创建一个名为 collectedProposals 的新变量。在此变量中，我们将收集从飞机服务供应商那里收到的所有建议。

将“作用域”活动添加到其中一个流分支。在此范围内，添加“分配”活动。此活动必须将 proposal 元素追加到 *collectedProposals* 变量。右键单击范围，然后选择*“转换为子进程”选项*。

为子流程定义两个变量，作为其输入参数;详见图12-30。

一个称为 *proposal*（基于从飞机服务提供商处收到的 SubmitProposalCallbackMessage），另一个称为 *supplier* ，类型为字符串。

编辑“分配”活动，将提案变量中的所有值复制到全局 *collectedProposals* 变量，包括供应商的名称。图 12-31 显示了此 Assign 活动。



**图 12-30。** *定义 UpdateCollectedProposals 子进程的输入参数：*

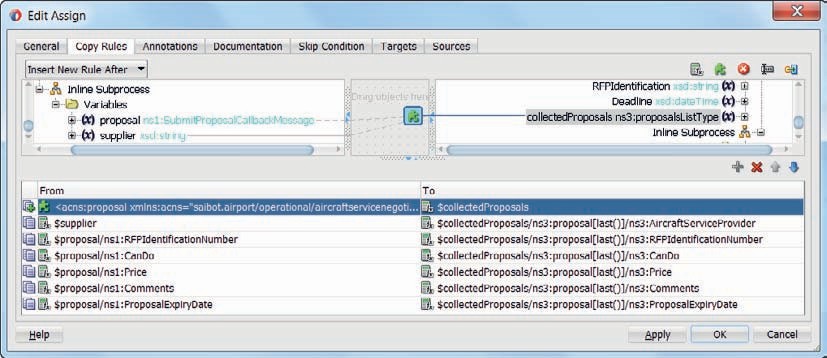
*收到的提案和提出提案的供应商的名称*

添加和编辑每个流程分支对子流程的调用。将局部变量映射到 子流程的输入参数，如图12-32所示。

为属性 ServiceRequestIdentifier *添加属性别名*，以允许关联发送到 *getStatusRFP* 操作的消息。这一次，该属性映射到 *RetrieveStatusAircraftServiceNegotiationRequestMessage* 中的 ServiceRequestId *元素*，如图 12-33 所示。

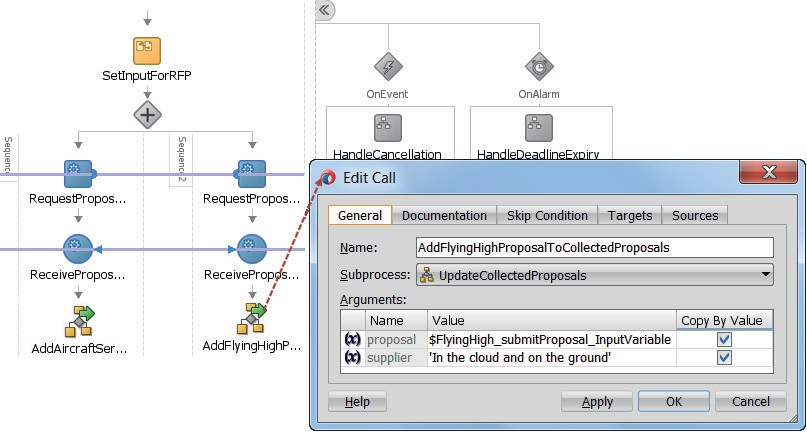
添加 OnEvent 处理程序，以限定 *AircraftServiceOrchestrator*  服务中 retrieveStatus *操作*的 *ExecuteRFP*  范围。配置关联集以挂接到此事件处理程序，如图12-34所示。

使用生成的局部变量*为* retrieveStatus 操作添加回复活动，并添加分配活动以将 *collectedProposals*  变量中的内容添加到此输出变量中。



**图 12-31。** *将有关供应商建议的详细信息复制到*

*collectedProposals 集合*

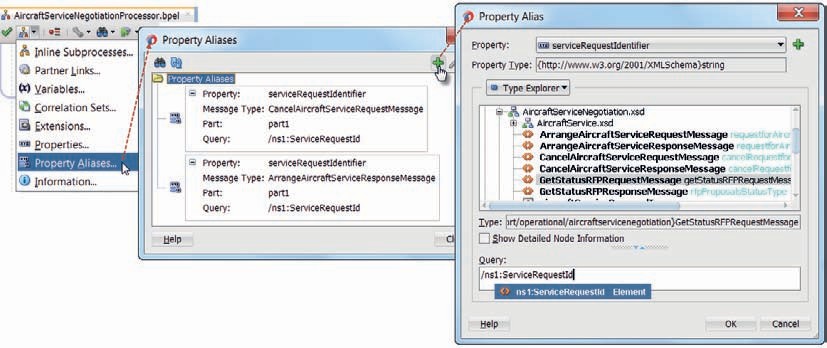


**图 12-32。** *配置对 BPEL 内联子流程的调用以添加收到的提案*

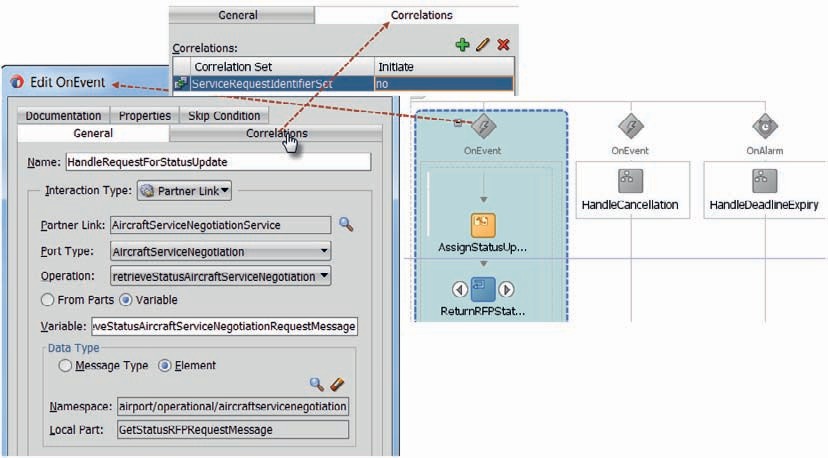
*到提案集*

###### 提示

*为了有更多时间询问正在运行的实例的状态，您可以在范围 ExecuteRFP* 内的流活动之后*添加一个*等待*活动，以强制实例运行更长时间。*



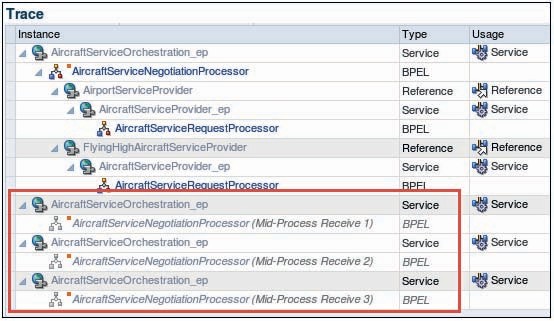
**图 12-33。** *为 GetStatusRFPRequestMessage 添加属性别名*



**图 12-34。** *配置 OnEvent 处理程序以处理到*

*启用关联时的 retrieveStatusAircraftServiceNegotiation 操作*

部署复合。像以前一样调用 *AircraftServiceOrchestrator* 复合，调用 arrange 操作。现在，您可以再次调用复合，这次调用 *retrieveStatus*  操作并传递与对 arrange 操作的调用中相同的标识符。应返回当前状态;当然，它的价值取决于飞机服务提供商是否已经将他们的提案发送给协调者。如图 12-35 所示



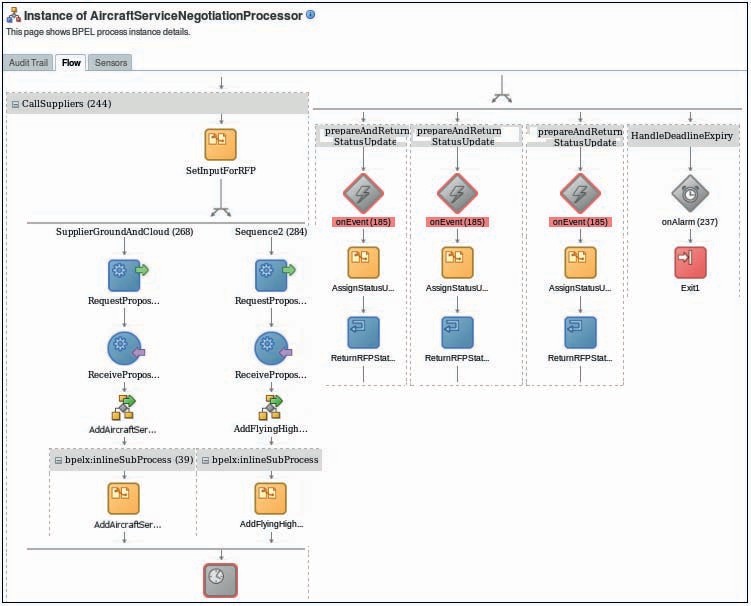
**图 12-35。** *消息流跟踪，其中包含对检索的三个进程中异步调用*

*状态操作*

AircraftServiceOrchestrator *的消息流跟踪*;指示了三个异步的进程中间状态检索调用。

请注意，您可以多次调用 *retrieveStatus* 操作。当您在*调用*  arrange 操作（启动组合）之前调用该操作时，BPEL 引擎将记住该请求，并在*调用 arrange* 操作以实例化 BPEL 流程后立即处理它。

图 12-35 所示交互的 BPEL 审计跟踪如图 12-36 所示。



**图 12-36。** *AircraftServiceNegotiationProcessor 的 BPEL 流跟踪，其中包含两个*

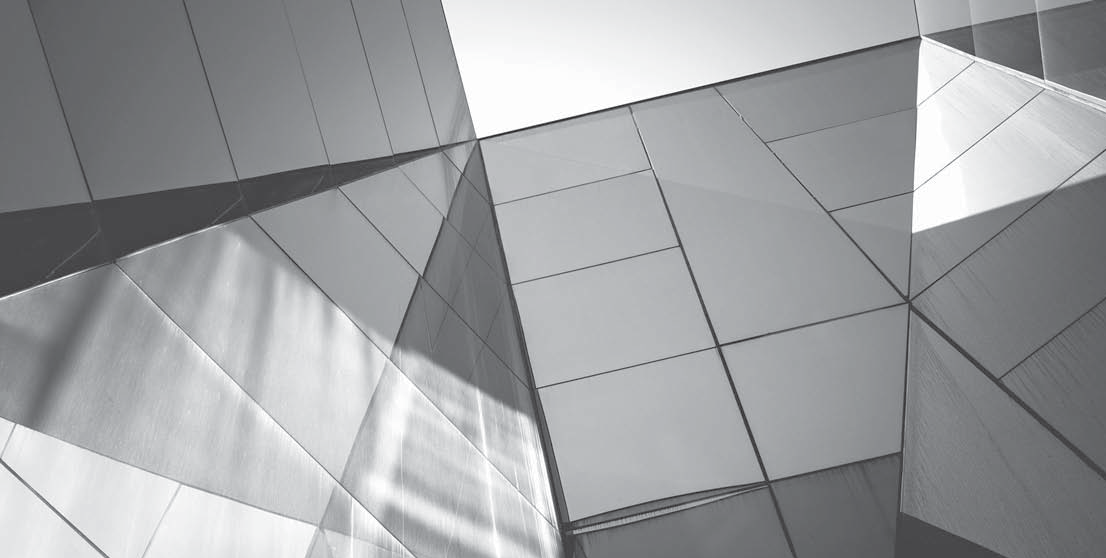
*与飞机服务提供商的异步交互和三个相关的进程中状态 检索请求*

### 总结

异步交互在现实世界中以及在面向服务的体系结构中都很重要。在不需要同步性的情况下，在同步交换中占用资源是一种浪费。SOA Suite 为异步消息交换提供了良好的支持，既可用于访问异步服务，也可用于实现异步服务接口。

虽然调解器和服务总线对异步交互有一定的支持，但它的级别较低，需要操作 WS-Addressing 标头。BPEL 非常适合异步交换 — 既可以实现它们，也可以启动它们。BPEL 引擎处理所有寻址和关联细节;因此，开发人员甚至几乎不会 注意到接口是异步的，而不是同步的。

除了异步的“请求后跟回调”模式之外，在本章中，我们还看到了 BPEL 如何通过关联将后续请求接受到已经运行的流程实例中。此类请求可以在主进程“线程”中显式等待，也可以作为进程中的异步事件到达，由事件处理程序处理。另一种类型的异步或后台事件处理程序由 OnAlarm 机制提供，该机制可以在达到截止时间或经过一段时间后中断主流。



章

# 13

## 入站适配器 — 轮询数据库、使用 JMS 和接收电子邮件

OA复合或服务总线项目不必通过来自外部使用者的请求消息触发。我们讨论过的用于对数据库、文件系统、JMS 目标、高级队列和人员的出站操作的适配器

**S**

（通过电子邮件、短信和聊天的 UMS）也朝入站方向运行。在这个方向上，外部事件激活适配器以触发代理服务或 SOA 组合。这包括出现新文件、更新数据库记录、在 JMS 队列或主题上发布消息或者电子邮件或聊天消息到达等事件。

本章演示如何使用这些入站适配器操作。这些对于让我们的 SOA 应用程序快速响应边界系统中的事件非常有价值——即使这些事件没有明确地作为业务事件发布。适配器的入站模式意味着我们不必编写各种轮询逻辑来跟踪我们周围各种通道和系统中发生的事情。

具体而言，本章将讨论用于发现新的或更改的数据库记录的入站数据库适配器模式、JMS 队列侦听器以及 UMS 适配器的入站模式，该模式 将监视传入的电子邮件和聊天消息。

### 暂存槽请求的数据库轮询

理想情况下，数据操作仅通过服务完成。然而，现实情况是，在许多组织中，现有的应用程序直接更新数据库表，而没有采用正确的服务路线。这些直接插入或更新数据库记录很可能会触发服务和进程级别的活动——在 SOA Suite 中。这些数据操作可能代表我们的服务需要响应的业务事件。

数据库适配器在第 4 章中首次介绍。然后，我们使用此适配器 从服务总线项目或 SOA 组合访问数据库，以调用 PL/SQL 存储程序或从表和视图查询数据。此适配器还支持入站模式，在这种模式下，主动权在于适配器，而不是服务。在本节中，我们将使用入站模式来检查一个表，在该表中，外部门户应用程序和 Saibot 客户经理的内部应用程序可以写入插槽请求 - 数据库记录表示 航空母舰拥有某个插槽的愿望。

在本节中，我们必须开发一个 SOA 组合，它将在数据库表中找到这些槽请求记录并对其进行处理 - 调用 *SlotService* ，以便在 Future 数据域中正确注册槽分配。此复合将轮询具有槽请求的表。 对于找到的每条记录，将启动 SOA 组合的新实例。

#### 准备环境

对于我们将要开发的 SOA 组合，用于从数据库表轮询记录，我们当然需要 首先创建该表。我们将检查新记录的表称为 *PORTAL\_SLOT\_ALLOCATIONS*。它是通过第 13 章在线资源中的 DDL 脚本创建的。该表包含 10 列，用于描述特定航空公司的机位分配请求 。

此外，就像我们在第 5 章中讨论的出站数据库适配器绑定一样，入站适配器绑定引用数据库适配器部署中指定的连接（JNDI 名称），该连接本身使用连接到包含表的数据库模式的 JDBC 数据源 。联机描述了数据库适配器所需的配置步骤。

#### 创建 SOA 组合

使用名为 *PortalSlotRequestProcessor* 的 SOA 复合项目创建新的 SOA 应用程序 *InboundProcessing*。选择模板 *Composite with Mediator*。将调解器的名称设置为 *PortalSlotRequestMediator*，并选择稍后定义接口。

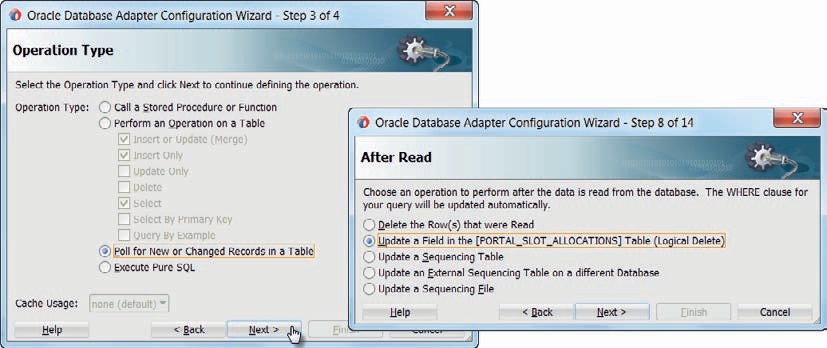
稍后，此调解器的入站接口将由入站数据库适配器绑定定义 。

##### 配置入站数据库适配器

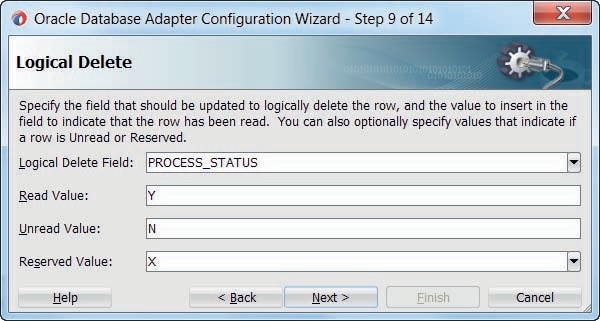
将数据库适配器拖到“服务”泳道上，因为在这种情况下，适配器绑定将启动 SOA 复合实例，并因此充当该术语的 SCA 含义中的服务。将适配器绑定的名称设置为 *PortalSlotRequestPoller*。使用与第 4 章中相同的 Saibot 数据库连接，并将 JNDI 名称设置为 eis/DB/SaibotCommonDB，对应于在运行时 SOA Suite 环境中为数据库适配器部署定义的连接之一的 JNDI 名称。在步骤 3 中，我们有一个新颖之处：选择“*轮询表中的新记录或更改的记录*”作为操作类型。此操作仅对入站数据库适配器绑定有意义。

在下一页上，您必须指明应轮询的表：PORTAL\_SLOT\_分配。然后，在步骤 8 中，可以指定数据库适配器在处理记录后的行为，请参见图 13-1。根据要处理的记录的识别方式以及 SOA 组合应用程序对源表的特权，我们可以选择直接更新已处理的数据库记录（更新甚至删除），或者我们可以指示已处理的每条记录的标识符入到不同的表中（与被轮询的记录在同一数据库或另一个数据库中）或在文件中 （在 SOA Suite 环境中本地）。

在可能的情况下，通常使用逻辑删除选项来更新 已处理记录的特定列。



**图 13-1。** *数据库适配器的入站模式将轮询新的或更改的记录*



**图 13-2。** *配置已处理数据库记录的逻辑删除的详细信息*

如果我们选择“处理后逻辑删除记录”选项，我们将在 步骤 9 中指定此逻辑删除的详细信息（图 13-2）。选择第 PROCESS\_STATUS 列作为 指示记录状态的列。值 Y 将指示记录已被读取和处理。 此列的值 N 标识尚未处理的记录。保留值 X 用于标记当前正在处理的记录。记录将仅短暂地具有此值。

向导的下一步是轮询选项（图 13-3）。在此步骤中，我们指定轮询频率;在此示例中，每 20 秒即可。我们设置了在单个 SOA 组合实例化中传递的数据库记录数;我们将此属性设置为 1，通常情况如此。在本例中，在单个事务中选取的数据库行数设置为 2;这意味着，每 20 秒，此数据库适配器的每个实例将从表中获取最多两条记录，并将为每个记录创建一个 SOA 组合实例。工作线程数（数据库适配器的实例数）由两个附加属性确定：*群集环境中的* activationInstances  *和多线程单节点设置中的* NumberOfThreads。

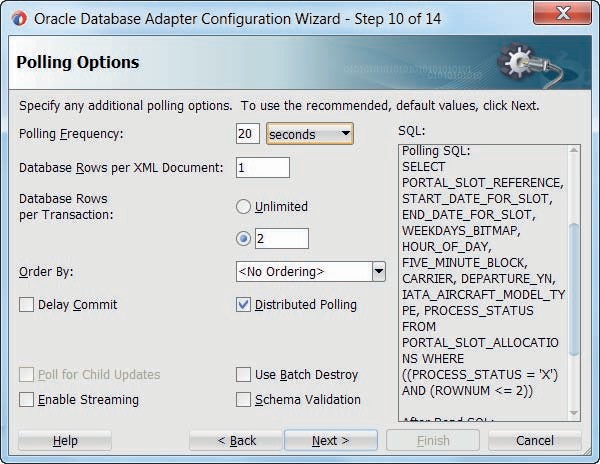
###### 提示

*如果要实现自己的逻辑删除机制，可以。在这种情况下，请选择“删除在步骤 8 中读取的行”，并在向导完成后编辑生成的 ..-or-mappings*

*.xml文件并编辑 <delete-query> 部分，该部分指定要为要逻辑删除的每条记录执行的 SQL。*

排序依据可用于指定理想情况下应处理未读记录的排序顺序。但是，对于处理结果消息的实际顺序，无法保证， 因此最好完全避免使用此选项。

选中 Distributed Polling 复选框。这将指示数据库适配器使用 SELECT ..FOR UPDATE SKIP LOCKED 语法，该语法可优化吞吐量并减少来自多个工作线程的负面影响。



**图 13-3。** *配置轮询选项以指示检查表的频率*

*以及每个服务实例和每个事务或轮询事件应处理多少条记录*

查看文档，了解其他更高级和/或异国情调的选项。按“下一步”转到步骤 10 - 定义的选择标准。此步骤允许我们进一步优化应执行的 SQL 语句，以获取要处理的记录。接受默认语句并继续按“下一步”，直到启用“完成”按钮。接受所有中间缺省值，然后按“完成”以完成此入站数据库适配器绑定的配置。

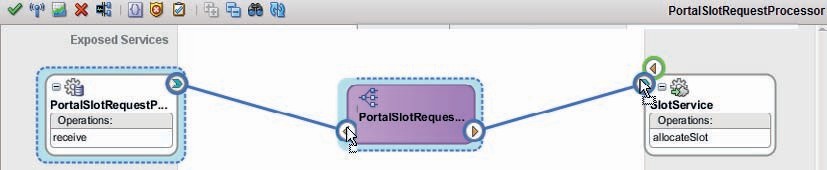
##### 为插槽服务创建引用绑定

我们希望调用在第 4 章中创建的 *SlotService*（并在第 7 章中重用），用于入站数据库适配器将传递给当前 SOA 组合的每个槽分配请求。因此，应将 Reference 绑定添加到 *SlotService 的复合中*。

在“外部参照”（External References） 泳道中单击鼠标右键，可以选择“插入”（Insert）|”SOAP 从下拉菜单中。输入 *SlotService*  的 WSDL 的 URL：http://localhost:7101/flight/SlotService?wsdl。将引用的名称设置为 *SlotService*。按 OK。 允许本地化所有文件。因此，WSDL 文档和关联的资源将复制到项目中。

##### 配置调解器

在这种情况下，像往常一样，调解人组件实际上是中间人。将入站数据库适配器绑定连接到调解器，如图 13-4 所示。将中介连接到 *SlotService* 引用绑定。



**图 13-4。** *具有入站数据库适配器的 PortalSlotRequestProcessor 复合*

我们现在需要做的就是将数据库适配器将馈送到 SOA 组合中的 XML 消息（结构）映射到 *SlotService* 预期的消息结构。前者由数据库适配器向导生成的 PortalSlotRequestPoller\_table.xsd 文件描述。创建一个新的 XSL 映射文件 inboundPortalSlotRequestToallocateSlotRequestMessage.xsl，该文件在 *PortalSlotAllocationsCollection* 和 *allocateSlotRequestMessage 之间进行映射*。

编辑调解器：配置路由规则以使用此 XSL 映射来转换 请求消息。

#### 部署和测试

随着 SOA 组合应用程序的完成，我们可以部署它了。完成后，没有

我们可以为此复合调用的服务接口。它的入口是通过轮询数据库表的入站 数据库适配器。

部署后，数据库的轮询将立即开始。轮询间隔过后， 将检查PORTAL\_SLOT\_ALLOCATIONS表中是否有状态等于 N 的任何（新）记录。对于检索到的每条记录，将创建一个 SOA 组合的新实例，该实例将数据库记录的 XML 表示形式作为其请求消息传递，每个轮询间隔最多有两个实例（对于单个适配器实例）。复合转换消息并将其传递给 *SlotService* 进行正确处理。整个事务的结果是将新记录添加到表 *FUT\_SLOTS*。

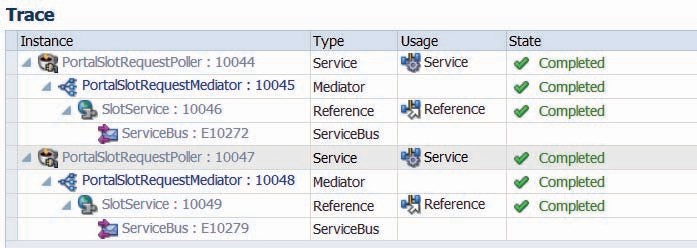
当我们在表 PORTAL\_SLOT\_ALLOCATIONS 中插入三条或更多条记录并 等待 大约 20 秒时，我们可以在 EM FMW 控件中找到图 13-5 所示的流迹。

其他未处理的记录必须等待下一个轮询间隔也被 检索和处理。

#### 入站适配器和异常

当入站消息处理失败时，没有要向其发送错误消息的显式参与方。入站数据库适配器只是中间人 — 从表中获取记录并将它们交给 SOA 组合。当 SOA 组合内部或外部发生异常时，SOA 组合实例将失败。我们必须设置故障策略或手动 程序来处理这些失败的实例。

此外，还可以使用拒绝处理程序配置入站适配器来处理被拒绝的消息。可重试错误之间是有区别的，例如，由



**图 13-5。** *入站数据库的单个轮询操作已提取两个数据库*

*记录并为每个实例创建一个 SOA 组合实例;这些实例调用了 SlotService（在服务总线上），该实例使用出站数据库适配器在表 FUT\_SLOTS 中创建记录*

连接问题 - 以及不可重试的错误，例如转换错误，甚至是未处理的业务异常。可以处理的一类特殊拒绝是，由于内部问题，入站适配器甚至没有进入将消息传递给 SOA 组合的阶段，例如，将本机数据转换为 使用适配器绑定配置的 XML 格式。

当处理导致 SOA 复合实例失败时，入站适配器读取的数据库记录将在逻辑上被删除（其逻辑删除状态设置为*“读取*”）。必须对 SOA 组合进行恢复;此时，数据库记录被视为已完成。

#### 扩展入站适配器活动

入站数据库适配器可能会导致 SOA Suite 服务器上出现大量负载：例如，如果导入操作或某些批处理创建或更新了大量数据库记录，或者入站数据库适配器在长时间停机后变为活动状态，则您可能认为几乎可以立即创建大量 SOA 组合实例。

幸运的是，适配器绑定定义上有可用的优化参数：

* *activationInstances –* 为在分布式集群场景中处理数据库记录而创建的数据库适配器的并发实例数。在多线程中

单节点场景，请改为配置 *NumberOfThreads* 。

* *MaxTransactionSize —* 数据库适配器实例将在单个事务中处理的最大记录数，因此在单个轮询间隔内处理的最大记录数。
* *PollingInterval —* 数据库适配器实例读取之间的等待时间。
* *MaxRaiseSize -* 单个 SOA 复合实例要处理的数据库记录数。

根据这些参数，我们可以计算出 SOA Suite 引擎因入站数据库适配器绑定而承受的最大负载：

每秒处理的最大数据库记录数计算为 MaxTransactionSize/PollingInterval \* activationInstances。例如：如果 MaxTransactionSize 为 150，则 6 个 activationInstances，PollingInterval 为 5 分钟，则每秒可以处理的记录不超过 150/300 \* 6 = 3 条记录（尽管我们每 5 分钟处理一次

每个实例）。

我们可以将其与我们期望的流量进行比较，看看我们的设置是否能够长期应对。但是，这并不能告诉我们处理单个数据库记录的工作量有多大，因此我们不知道在吞吐量和资源方面，我们是否真的每秒处理这三条记录。

要提高可扩展性并减少数据库适配器并发实例的相互影响，请使用分布式轮询策略。在 Oracle 数据库上，这意味着使用“SELECT FOR UPDATE SKIP LOCKED”获取记录，这意味着轮询的记录将被锁定，锁定的记录将被忽略，并且不会因现有锁定而产生等待时间。这是迄今为止为数据库适配器实例保留和提取批量记录的最有效方法，而不会对其他实例产生任何影响。

如您所知：入站适配器可以通过嵌入式 Enterprise Scheduler 打开和关闭。例如，这使我们能够尊重外部系统的维护窗口 。第14章将对此提供更多细节。

##### 使用入站数据库适配器分解 BPEL 流程

BPEL 流程可以长时间运行。这通常发生在 BPEL 流程等待外部操作*时 —* 消息从外部系统到达或由人工代理执行的操作。然而，BPEL 基础结构并不适合于长时间运行的流程实例。一些考虑因素包括冻结存储的大小、新版本的 SOA 组合的部署以及 SOA Suite 运行时环境的升级。理想情况下，避免运行数天或（更）长时间的实例。

有时，BPEL 流程的逻辑决定了长期运行的性质。这就是 定义业务功能的方式。在不同的时刻，随着时间的推移，必须执行该过程的各个阶段，每个阶段都由某个外部事件触发*——*这可能 只是时间的流逝。每个阶段都需要访问至少一些 进程（实例）的状态。

如果 BPEL 流程中的各个阶段是由截止时间或相关消息交换以外的事件激活的，那么可以选择为每个阶段创建单独的 BPEL 流程，并通过监视截止时间和状态更改的入站数据库适配器激活每个阶段流程。每个阶段从检索状态信息开始，最后通过记录流程实例的更新状态信息结束。

图 13-6 直观地显示了此方法。原始的 BPEL 流程*（*由周围的圆角矩形表示*）*已拆分为多个特定于阶段的流程定义。阶段 0 根据 SOAP 请求以常规方式启动流程实例。当阶段 0 完成时，BPEL 组件使用可重用的 BPEL 子流程记录应延续到下一阶段的状态。此子进程使用自定义格式（可能基于 XML）在数据库架构的自定义表中记录相关状态。

肥皂

服务绑定

*整体业务流程*

BPEL公司

阶段 0

BPEL公司

第 1 阶段

external systems

BPEL 子

检索状态

BPEL 子

记录状态

流程 实例 状态

流程实例 截止时间和状态注册表

BPEL公司

协调者

BPEL公司

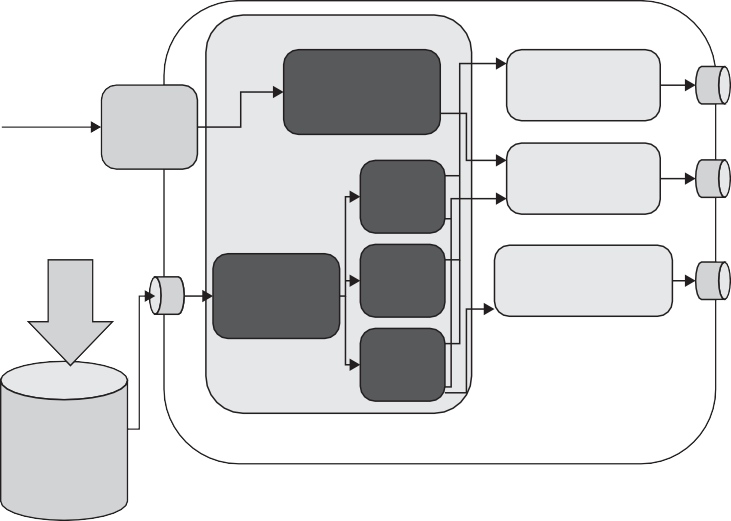
第 2 阶段

BPEL公司

第 3 阶段

BPEL 子

报名截止日期



**图 13-6。** *将长时间运行的 BPEL 流程分解为多个阶段 — 由基于截止时间或状态更改的入站数据库适配器触发*

入站数据库适配器监视一个表，该表保存此特定 SOA 组合的流程实例的记录。每条记录代表一个（正在运行的）流程实例。记录可以通过状态或截止时间触发数据库适配器。

当截止时间过后或状态设置为特定值时，入站数据库适配器将触发 SOA 组合的新实例并调用协调器 BPEL 流程。协调器根据从数据库中读取的当前进程状态确定应执行哪个阶段。然后，它调用该阶段的 BPEL 流程组件*，*传递流程的实例标识符。

选定状态的 BPEL 流程使用此实例标识符从定制流程状态存储库中检索流程状态。恢复状态后，此阶段的操作将由 BPEL 流程执行。当该阶段完成时，组件将记录流程实例现在已达到的状态。它还可以为流程实例写入截止时间记录或新的状态值。此时，复合实例已完成处理。在 SOA 套件的级别上不会留下任何正在运行的实例。只有在自定义表中，我们可能仍然有可能为业务流程执行下一阶段。

每个阶段特定的流程实例将运行很短的持续时间。整个业务流程由所有阶段组成，可能运行数天、数周甚至数月，而不会引入这种长时间运行的实例的所有缺点（但引入了一种非标准的流程协调机制，不会产生端到端的消息流跟踪，因为对于 SOA 套件来说，每个阶段都是一个新的对话）。使用复合传感器至少是一种能够快速找到单个业务流程实例的所有相关 SOA 组合实例的方法。请注意，这种方法类似于我们在 BPM Suite 中对 BPMN 流程所做的工作，对于自适应案例可能更是如此。

### 代表财务部门读取 JMS 队列

第 6 章中放在 JMS 队列中的 AircraftMovement 消息在通过 JMS 适配器的入站模式触发财务部门的服务总线服务时，会在此处重新出现。注意：我们在 Service Bus 项目中使用 JMS 的 JCA 适配器，只是为了了解入站适配器如何像 SOA 组合一样轻松地用于 Service Bus 项目 *-* 产品本身没有特别的偏好将入站 JCA 适配器与其中一个一起使用。服务总线项目可以在入站模式下利用 JMS 传输，而不是使用用于 JMS 的 JCA 适配器*，*这比基于 JCA 的方法轻一些。

我们将看到如何在服务总线项目中使用管道，无论其输入源自入站适配器还是更常见的传入 SOAP 请求，这完全没有区别 。

#### 创建服务总线项目Create the Service Bus Project

创建新的服务总线应用程序和项目，两者都称为 *FinancialAircraftMovementProcessor*。 将 Schemas 文件夹从*您在第 6 章中处理的* AircraftMovementService 项目复制到此新项目中。此文件夹包含新项目将从 JMS 队列中使用的消息的模式定义。

#### 添加入站 JMS 适配器绑定

将 JMS 适配器从 “Component”（组件）面板中的“Advanced”（高级）类别拖到 Proxy Services 泳道。

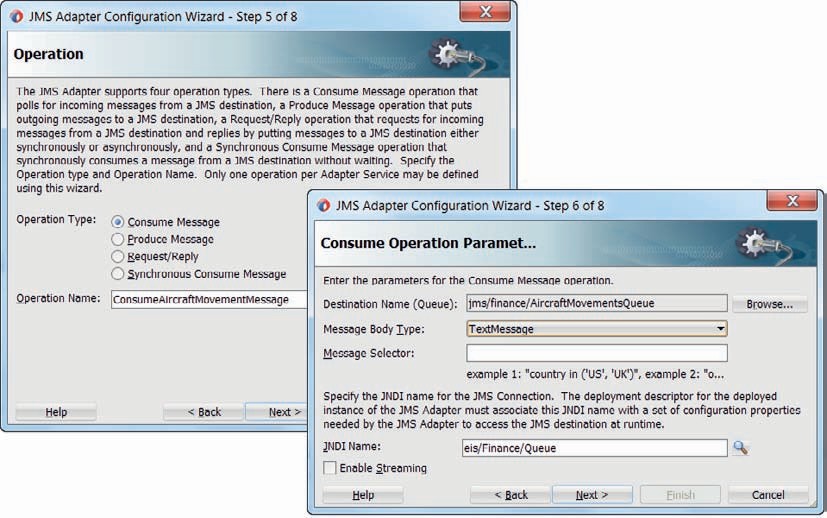
此时将显示适配器配置向导。将入站服务的名称设置为 *ConsumeAircraftMovementMessages*。在步骤 2 中，选择 Oracle Enterprise Messaging Service，然后选择 Oracle WebLogic JMS 作为实现。

在“适配器接口”步骤中，接受默认值“从操作和架构定义（稍后指定）”。按“下一步”。在“操作”页中，选择“*使用消息*”单选按钮。将操作名称设置为 *ConsumeAircraftMovementMessage*（此值非常任意）。单击“下一步”，进入“消费操作参数”页面，如图13-7所示。

选择 jms/finance/AircraftMovementsQueue 作为 [JMS] 目标名称。尽管名称Destination 令人困惑，但我们在这里配置了 JMS 队列或主题，我们希望此适配器绑定从中获取消息。邮件正文类型为 *TextMessage*。我们不会使用 MessageSelector。JMS [适配器] 连接的 JNDI 名称是 eis/Finance/Queue（另请参阅下一节）。单击“下一步”继续。

向导的最后一页用于指定描述从队列中使用的消息的架构。*从 AircraftMovementService.xsd 中选择* reportAircraftMovementRequestMessage 元素。单击“下一步”，然后单击“完成”。现在已创建代理服务，就像使用适配器配置的 JCA 文件一样。

右键单击服务总线概述编辑器的中心区域。选择“插入”|”管道。调用新管道 *ProcessAircraftMovementPipeline*。单击“下一步”转到向导的第二页，请参见图 13-8。选择 WSDL 单选按钮。单击齿轮图标以生成



**图 13-7。** *入站 JMS 适配器配置向导中的关键步骤*

此管道将公开的 WSDL。接受所有默认值*，*包括“单向接口”类型。 根据我们在 JMS 适配器配置中使用的 AircraftMovementService.xsd *中的相同* reportAircraftMovementRequestMessage 指定单个输入参数。

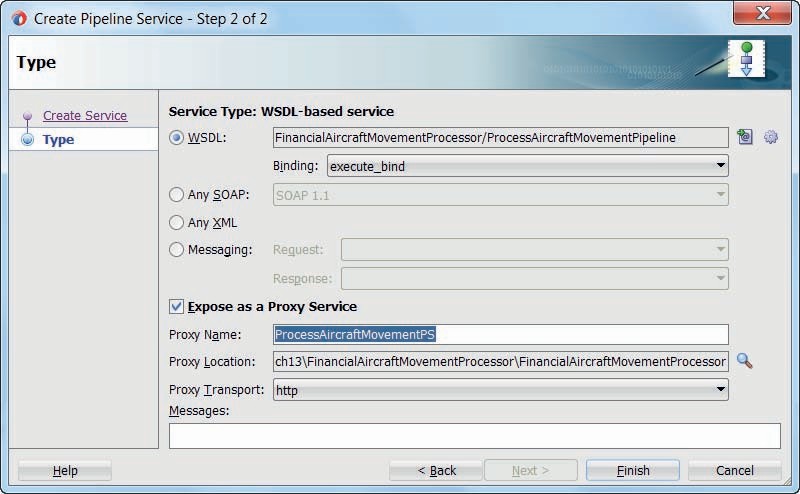
此外，通过名为 ProcessAircraftMovementPS *的代理服务公开管道*。单击“完成”以创建管道。

将代理服务 *ConsumeAircraftMovementMessage（*使用入站 JMS 适配器创建*）*连接到此管道。这意味着现在有两个管道通道：一个通过 ProcessAircraftMovementPS 公开的单向 SOAP Web 服务，*另一个*通过入站适配器和 ConsumeAircraftMovementMessage *代理服务从 JMS 队列，如图* 13-9 所示。

双击管道以打开其编辑器。现在创建一个非常简单的实现，因为我们要展示的是入站 JMS 适配器如何允许 Saibot 机场的财务部门使用由完全不同部门的服务发布的飞机移动消息。

添加管道对。将 [Message] Report 操作添加到请求管道。将内容设置为

$body *-* 这样您就能够看到从 JMS 队列中检索到的整个消息。 定义一个名为 *FlightnumberForAircraftMovement*  的搜索键，并让它从 body 变量返回航班号。保存更改。

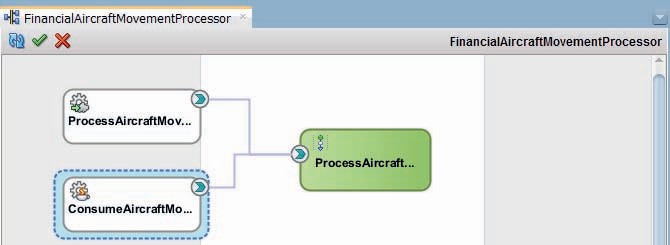


**图 13-8。** *管道服务 ProcessAircraftMovementPipeline 的配置*

#### 配置 JMS 适配器

在有意义地部署服务总线项目之前，我们必须准备 JMS 适配器部署。

在第 6 章中，我们已经在 WebLogic Server 上为我们正在讨论的队列配置了 JMS 适配器。已通过 WebLogic 控制台将连接添加到



**图 13-9。** *进入管道的两个通道，包括一个来自入站 JMS 适配器的通道*

JMS 适配器的连接集，JNDI 名称为 *eis/Finance/Queue*。此连接是指 *FinanceConnectionFactory*，它有助于访问资源，例如*我们现在将使用 JMS 适配器的入站模式侦听的* AircraftMovementsQueue。如果我们只是重用此连接，则不需要更多配置。

#### 部署和测试

您可以直接测试管道*（*右键单击并按运行*），*也可以通过 *ProcessAircraftMovementPS*  代理服务间接测试管道。无法直接测试基于入站 JMS 适配器的代理服务 ConsumeAircraftMovementMessage。相反，我们必须部署服务总线项目*，*该项目会将适配器绑定注册为 *AircraftMovementsQueue* 上的订阅者。当我们下次在该队列上发布消息时，这应该激活适配器绑定并间接调用管道。

通过调用第 6 章中开发的 *AircraftMovementService*，我们可以将消息发布到 JMS 队列上，前提是此服务在条件分支中选择正确的路径。或者，我们可以使用 QBrowser、SoapUI（带有 HermesJMS）或 WebLogic 管理控制台等工具在 *AircraftMovementsQueue 上发布消息*。

在队列上发布消息后，在 EM FMW 控件中检查是否发布了预期的新消息报告，其中包含放入 JMS 队列的消息的详细信息，以证明入站适配器确实使用了该消息并激活了服务总线项目中的管道。

### 机场问讯处 - 接收电子邮件

有关航班状态的问题可以通过即时消息（聊天）和电子邮件回答，正如我们在第 6 章中看到的那样。在本节中，我们将了解如何 通过人机交互渠道提出这些问题，重点是电子邮件。

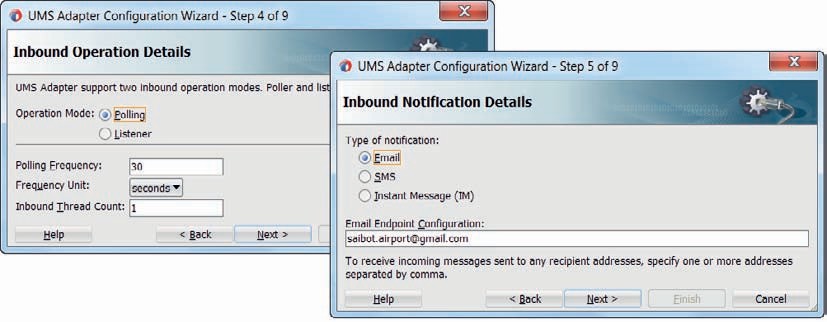
UMS 适配器不仅具有用于启动与人员交互的出站模式。它还支持入站模式，在该模式下，它轮询电子邮件服务器、XMPP 服务器和其他消息传递引擎，以允许传入消息触发新的 SOA 复合实例或对服务总线的调用。我们将使用此 UMS 适配器的入站模式来监视邮件服务器上邮件帐户的收件箱。对于此收件箱中收到的每封电子邮件，都会有一个 SOA 组合的实例。

#### 创建 SaibotHelpdesk SOA 组合

创建一个名为 SaibotHelpdesk *的新 SOA 应用程序*。使用空模板创建具有相同标题的新项目。

将 UMS 适配器拖动到左侧的服务泳道，以创建入站适配器 绑定。此时将显示配置向导。将名称设置为 *HelpdeskEmailProcessor*。接受 缺省入站连接 JNDI 名称 （eis/ums/UMSAdapterInbound）。在步骤 3 中， 操作类型应为 *“入站接收通知*”。接受默认操作名称 *ReceiveNotification*。

步骤4用于配置入站操作详细信息，例如轮询设置，如图13-10所示。将 30 秒设置为轮询间隔，或您认为合适的任何其他值。鉴于我们的开发环境中的数据量很慢，我们不需要配置多个线程来处理传入的电子邮件。注意：除了轮询模式，我们还可以选择 侦听器模式。后一种模式允许我们近乎实时地接收电子邮件，让 UMS 服务器为此复合执行专门的工作。



**图 13-10。** *配置轮询操作和电子邮件终结点（又名电子邮件帐户）*

*要轮询的收件箱*

在步骤 5 中，我们指定要侦听其收件箱的电子邮件帐户。我们可以在逗号分隔的列表中提供多个帐户。在“电子邮件终结点配置”属性中提供电子邮件地址，在图 13-10 中设置为 [saibot.airport@gmail.com。](mailto:saibot.airport@gmail.com) 此处提供的电子邮件地址必须在 UMS 电子邮件驱动程序中配置，如下一节所述。

向导的第 6 步用于配置消息处理。电子邮件的有效负载可以作为字符串甚至 base64 二进制对象传递到复合应用程序中，或者当电子邮件内容是 XML、JSON 或其他一些可以转换为正确 XML 的结构化格式时，它可以进行本机转换。请注意，除了电子邮件内容之外，SOA 组合还接收了广泛的 JCA 标头属性，这些属性提供从主题到和抄送到 messageID、回复、语言、编码和附件指示的详细信息。邮件 附件也可用于 SOA 组合以供进一步处理。

在这种情况下，请选中“消息是字符串类型”框。

在步骤 7 中，我们可以指定消息过滤器。它们用于指示 UMS 适配器实际处理哪些电子邮件，并为哪些电子邮件创建 SOA 组合的实例，以及忽略哪些电子邮件。邮件过滤器可以使用白名单（特别是处理来自这些发件人的邮件）、黑名单（处理来自这些发件人的电子邮件以外的所有邮件）以及按主题、内容、发件人、收件人和回复进行文本过滤。此外，在步骤 8 中，我们甚至可以配置一个自定义 Java 类的标注，该类对传入的消息执行我们自己的特定检查，以确定是否将其转发到 SOA 组合。

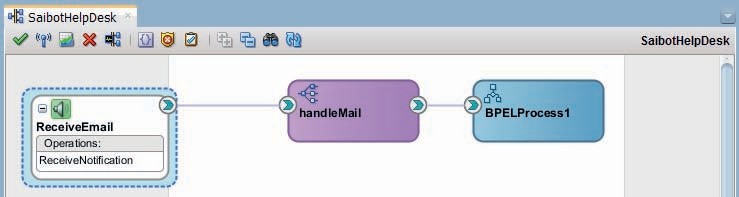
在这种情况下，根本不配置任何筛选。

按“下一步”和“完成”，完成向导并创建适配器绑定。

将 Mediator 组件拖动到复合应用程序中。称其为 *EmailMediator*。将 入站适配器绑定连接到调解器。

将 BPEL 流程组件添加到应用程序，称为 *EmailProcessor*。使用单向接口配置此组件，接受默认的简单单字符串输入。

将 *EmailMediator* 连接到 *EmailProcessor*。应用程序现在看起来如图 13-11 所示。



**图 13-11。** *SaibotHelpdesk 与要拦截的入站 UMS 适配器复合*

*传入的电子邮件，将它们移交给调解器进行过滤、转换和路由， 最后是 BPEL 流程进行处理*

在 EmailMediator 中编辑传递规则：使用 Assign Values 选项将电子邮件的内容传递给 BPEL 流程。这将允许我们看到我们是否能够 拦截传入的电子邮件，让它们触发新的 SOA 组合实例，并将 电子邮件有效负载传递给 BPEL 流程进行处理。稍后将讨论电子邮件的更有意义的用法 。

#### 配置电子邮件驱动程序

为了配置我们的 SOA Suite 运行时环境，以便在某个电子邮件服务器上轮询收件箱中的特定电子邮件帐户，我们必须在受控服务器上配置 UMS 电子邮件驱动程序属性。根据所使用的邮件服务器，将配置 IMAP 或 POP3 协议，以读取邮件服务器上收到的邮件。必须配置主机和端口，以及要轮询其邮件帐户的用户、应轮询的文件夹以及应轮询的帐户的密码。

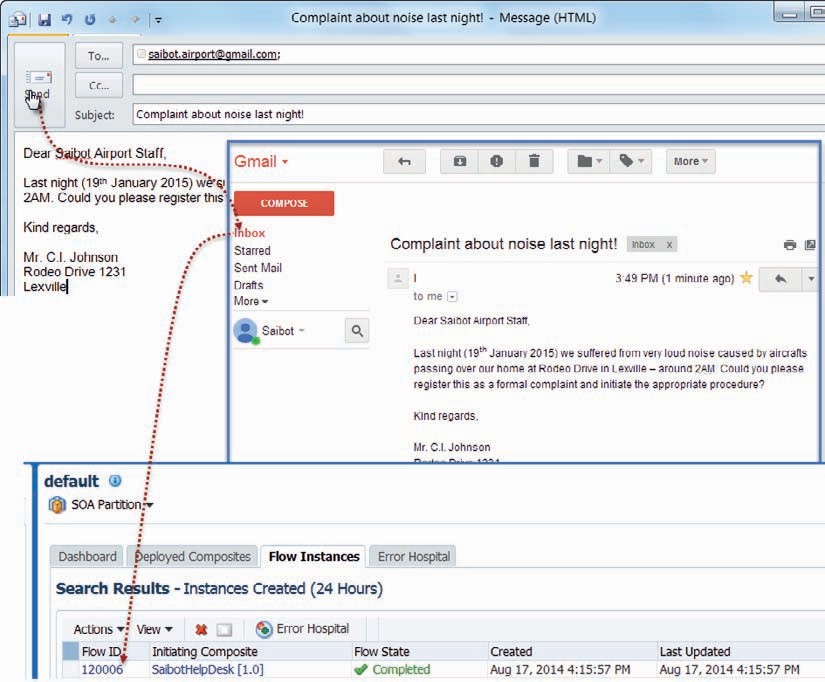
联机资源描述了此配置的详细信息。

#### 部署和测试

将 SOA 组合部署到 SOA Suite 运行时。与具有入站数据库适配器的复合体一样，没有直接的方法来测试复合体。它没有服务接口。它只能通过通过接收电子邮件触发的入站 UMS 适配器激活。

将电子邮件发送到为入站交互配置的电子邮件地址。过了一会儿，根据为 UMS 入站适配器绑定配置的轮询时间，电子邮件将被选取并传递给 SOA 组合的新实例，如图 13-12 所示。

我们创建的用于处理传入电子邮件的 SOA 组合非常简单， 特别是因为我们现在没有真正进行任何消息处理。但即便如此，通过电子邮件激活 SOA 套件所需的工作量却如此之少，这还是令人惊讶的。请注意 ，对于传入的聊天消息，我们必须经历非常相似的步骤。



**图 13-12。** *向* [*saibot.airport@gmail.com*](mailto:saibot.airport@gmail.com)  *发送电子邮件会触发*

*SaibotHelpdesk 通过从收件箱中选取电子邮件的入站 UMS 适配器绑定进行复合*

### 其他入站交互

除了通过数据库适配器、JMS 适配器和 UMS 适配器（用于电子邮件）进行的入站交互之外，还有其他几个入站操作。我们将简要介绍其中的一些。

AQ 适配器显然与 JMS 适配器非常相似;它订阅数据库中的高级队列，并在队列中使用消息时触发 SOA 复合或服务总线代理服务。在此区域中，还有 MS （Microsoft）、MQ 和 MQ 系列适配器，它们具有由到达队列的消息触发的入站模式。

文件适配器和 FTP 适配器都支持文件轮询。每当新文件到达配置的目录时，文件适配器（从本地 [共享] 文件系统）或 FTP 适配器（可能远程）FTP 服务器（在轮询间隔结束时）都会检索该文件。

文件中的内容按指定方式进行处理，例如，使用本机格式转换或对二进制文件不透明。此外，我们可以配置文件 Valves，即对检索到的文件执行预处理的自定义 Java 类。

###### 提示

*我们可以指示入站模式下的文件适配器仅在找到“触发器文件”（具有预定义名称的虚拟文件）后才开始轮询。*

*通过这种方式，我们可以确保文件仅在完成时才被处理 。*

除了开箱即用的 JCA 适配器之外，还可以创建自己的自定义 JCA 适配器，例如，轮询 DropBox 文件夹、RSS 源或您可能希望触发服务总线项目或 SOA 复合实例的更改的任何其他资源。定制适配器利用 JCA 1 路入站交互与轮询活动的激活机制。JDeveloper 中的“自定义适配器”向导可用于配置自定义入站适配器的适配器绑定。

当然，轮询、收集、聚合然后触发服务总线或 SOA 组合公开的服务的任何计划进程都可以执行与入站适配器非常相似的操作。在下一章中，我们将介绍企业调度服务 （ESS）。我们可以使用此服务来计划使用服务总线和 SOA 组合实现的服务的执行、（入站）适配器的激活和停用以及外部程序的执行，这些程序可以执行轮询操作，并根据结果调用服务总线或 SOA 复合服务。

事件交付网络 （EDN） 是 SOA 组合的重要入站机制的基础。发布到 EDN 的事件可以传送到正在运行的实例或新创建的实例。EDN 在技术性更强的 JMS 消息传递基础结构之上提供了一个功能强大、更面向业务的层。我们将在第 15 章中讨论 EDN。

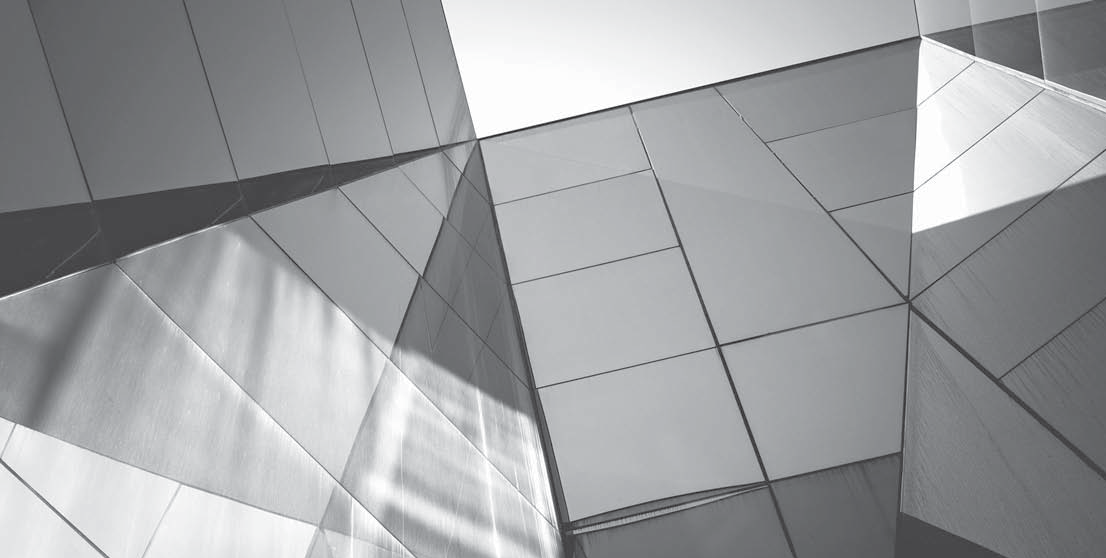
### 总结

如果服务总线或 SOA 组合中的操作主动权不属于某个对响应感兴趣或认为有义务调用单向服务的外部方，则使用入站适配器。相反，适配器配置为定期检查外部系统是否存在相关更改（例如新文件、新电子邮件或聊天消息、已更改或新的数据库记录或队列中的消息），并在发现此类更改时启动 SOA 复合实例或调用服务总线代理服务。入站适配器可以在（异步）直通式处理 （STP） 的实现中发挥重要作用。

本章介绍了各种入站适配器绑定的配置。很明显，一旦适配器转发了它从外部系统检索到的有效负载，在管道（在服务总线中）或中介器或 BPEL 流程（在 SOA 组合中）中处理该有效负载与处理由外部合作伙伴主动发送的正常请求消息中接收的有效负载没有什么不同。

入站适配器的配置可能需要非常小心。为确保及时采取行动，不应将轮询时间设置为高值。事务的大小（一次处理的块有多大）以及并发活动的轮询线程数 对于调整工作负载和防止服务器过载非常重要。





章

# 14

## 使用 Enterprise Scheduler 服务

Fusion Middleware 12.1.3 平台包含 ESS 或 Enterprise Scheduler Service。此服务可用作基于计划的异步作业业务流程协调程序。它可以 执行操作系统操作、Java 调用（本地 Java 或 EJB）、PL/SQL 调用、

**T**

以及 Web 服务调用（同步、异步和单向），包括对 SOA 组合、服务总线和 ADF BC Web 服务的调用。用于规划作业执行的计划可以定义单个未来时间点，也可以描述作业执行的周期性重复模式。计划的最精细粒度是分钟和小时级别。

ESS 显然不适用于非常快速的轮询或高频任务。

ESS 具有作业集的概念。作业集由多个作业步骤组成，其中作业集步骤可以是一个作业，也可以是另一个作业集。作业集步骤的执行可以配置为顺序或并行。使用串行作业集，Enterprise Scheduler 支持根据上一步的执行状态在步骤之间进行有条件的分支。此外，一个作业集步骤的输出可用作下一步的输入。

作业和计划可以从访问 Java API 的客户机应用程序、 ESS 的 Enterprise Manager FMW Control 用户界面和 Web 服务来定义。这个 Web 服务可以通过它来调度（预定义的）作业，可以使用一个特殊的 Schedule Job 活动从 SOA 组合中的 BPEL 流程中调用。此外，还可以创建用于调用 ESS Web 服务的服务总线业务服务，并且中介器或 Spring 组件可以从 SOA 组合进行该调用。

在 SOA Suite 12c 中，除了从调度作业调用服务和从服务调用来调度作业之外，ESS 至少还有两种使用方式 。

一种是故障实例的批量恢复。这可以使用计划进行计划，以确保在不久的将来在方便的时刻执行恢复，并随着时间的推移分散进行，以防止突然出现突然的峰值负载。在幕后，Enterprise Scheduler Service 用于根据计划协调恢复的执行。

ESS 的第二个用例是根据计划激活和停用（入站）适配器。安排适配器可用性的关键原因是后端系统的可用性（为什么当我们知道需要调用的 ERP 系统每晚都关闭时，随时开始处理队列上的消息）以及系统上的负载 - 让我们在系统没有被更重要流程的峰值负载淹没的时候处理低优先级的文档入站流 。

ESS 在“中间件作业”方面补充了数据库调度程序。它与数据库调度程序协作以执行 PL/SQL 作业。请注意，SOA 组合实例的计划清除是直接通过数据库调度程序完成的，没有 ESS 参与。相比之下，历史作业数据的定期清理由 ESS 本身处理。

### 关闭用于处理门户槽请求的入站数据库适配器

Enterprise Scheduler Service 支持围绕 SOA 套件的许多管理活动。虽然我们将在第 6 部分中更详细地讨论管理，但我们将研究一个 关于使用 ESS 进行环境管理的特定用例。

在上一章中，我们介绍了入站数据库适配器。我们创建了

*PortalSlotRequestProcessor* SOA 组合，它使用数据库轮询器在

表 *PORTAL\_SLOT\_ALLOCATIONS*.轮询频率设置为每 20 秒一次。只要 SOA 组合保持部署和活动状态，这种情况就会一直持续下去。

想象一下，在某个时期内，SOA 套件每天都有大量的负载，我们更愿意减少非关键进程的资源使用。

进一步假设从门户到达的插槽分配请求被视为不紧急，例如，因为与我们的客户经理商定的业务服务级别是必须在 24 小时内处理这些请求，而不是每 20 秒处理一次。我们不想创造一个大批量，只要有可能，我们就会努力实现直 通式处理。但是在每天凌晨 1 点到 2 点之间，我们希望暂停入站数据库适配器。

在本节中，我们将使用 Enterprise Scheduler Service 来实现此目的。我们将创建每天凌晨 1 点触发的计划，用于停用适配器，凌晨 2 点触发，用于激活适配器。事实上，为了让测试更有趣，我们将使用在整点过后 10 点和整点过后 30 点触发的时间表。然后，这些计划在 Enterprise Manager Fusion Middleware Control 中与入站数据库适配器绑定 *PortalSlotRequestPoller 相关联*。

#### 创建时间表

ESS 计划用于描述一个或一系列时间时刻。计划可以与一个或多个作业定义相关联，以描述何时应执行这些作业。定期计划具有描述时间时刻如何随时间分布的频率。定期计划可以具有开始时间和结束时间，以指定 应进行重复的时间段。

要创建将控制入站数据库适配器的计划，请打开 EM FMW 控件并选择节点 Scheduling Services |ESSAPP的。从页面顶部的下拉列表中，选择“作业请求”|”定义计划，如图 14-1 所示。

单击该图标以创建新计划。将计划的名称指定为 At10minPastTheHour。将显示名称设置为“每小时过去 10 分钟”。必须在软件包 [/oracle/apps/ess/custom/]soa 中创建调度。这是用于适配器激活的计划的要求。

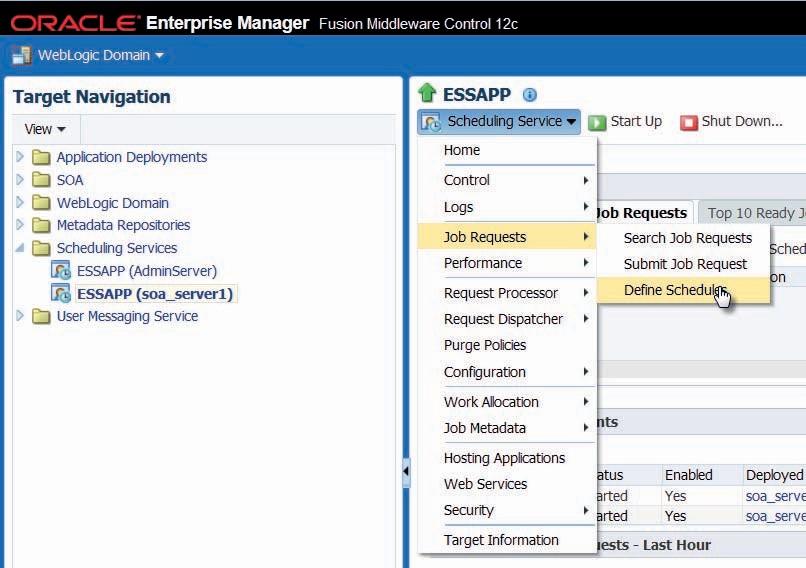
选择频率为每小时/分钟、每 1 小时 0 分钟，将开始日期选择为未来（甚至过去）不太远的任何日期，并将时间设置为任何小时后 10 分钟。示例如图 14-2 所示。

请注意，使用“自定义时间”按钮，我们可以生成一长串时间点，然后根据需要对模式进行一些例外时手动修改它们 。

单击“确定”以保存此计划。

创建名为 *At30minPastTheHour* 的第二个计划。该定义与前一个定义非常相似，只是开始时间应该在某个小时后 30 分钟。单击“确定”保存此计划定义。

请注意，可以通过 ESS 公开的 Java API 以及 JDeveloper 中的 IDE 支持来创建更复杂的重复计划。这些允许包含或排除特定工作日或月份的选项目前无法通过 EM FMW 控件进行设置。



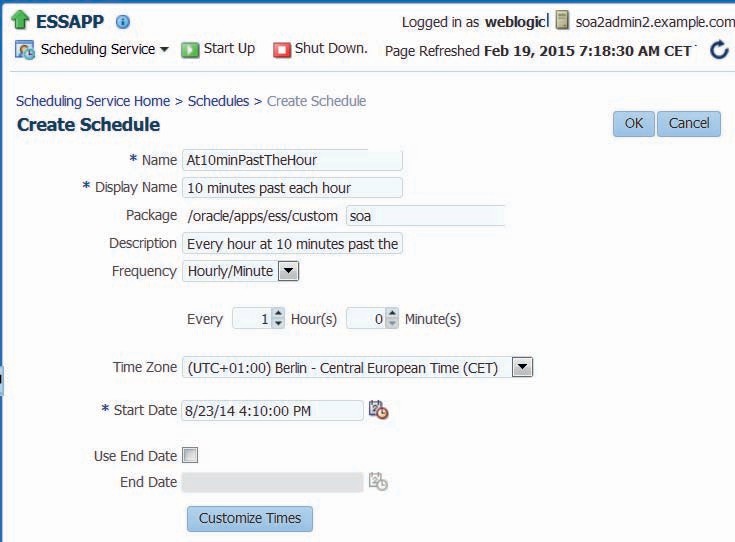
**图 14-1。** *打开页面以定义 ESS 计划*

#### 应用入站数据库适配器的激活和停用计划

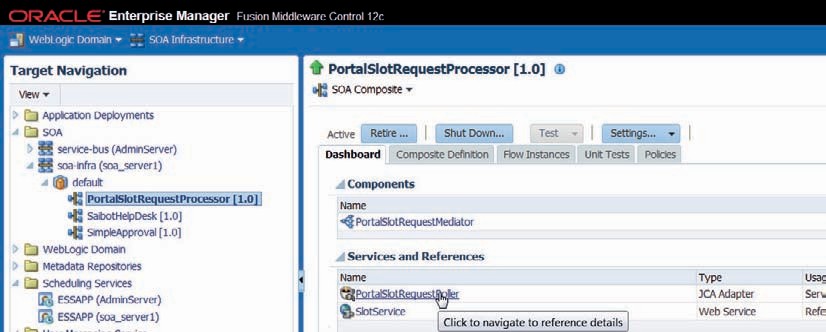
选择节点 SOA | soa-infra | default |PortalSlotRequestProcessor - 我们在上一章中创建的复合。在“Services and References”下，单击入站数据库适配器绑定 *PortalSlotRequestPoller*，如图 14-3 所示。

此时将显示 *PortalSlotRequestProcessor*。单击适配器时间表的图标。在出现的“适配器计划”弹出窗口中（图 14-4），我们可以选择要用于停用和激活适配器绑定的计划。使用 At10minPastTheHour 计划*停用，使用* At30minPastTheHour  *计划*进行激活。按 Apply Schedules 确认新配置。

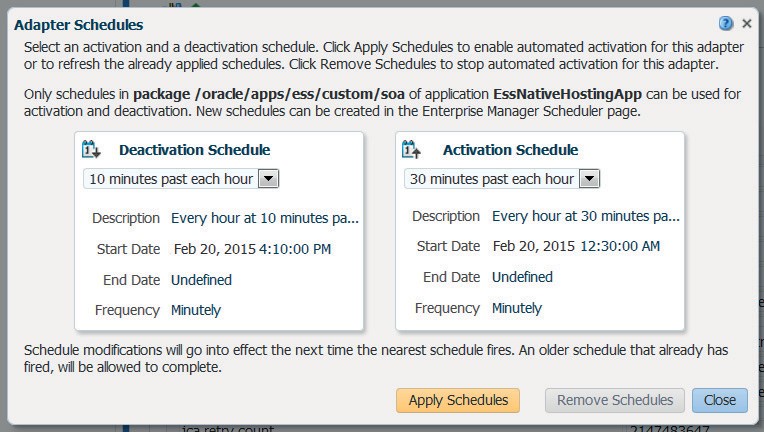
从这一刻起，轮询表 PORTAL\_ SLOT\_ALLOCATIONS 的入站数据库适配器绑定 在每小时仅活动 40 分钟，从整点后 30 分钟开始。



**图 14-2。** *在 10minPastTheHour 配置计划*



**图 14-3。** *向下钻取到入站数据库适配器绑定的配置*

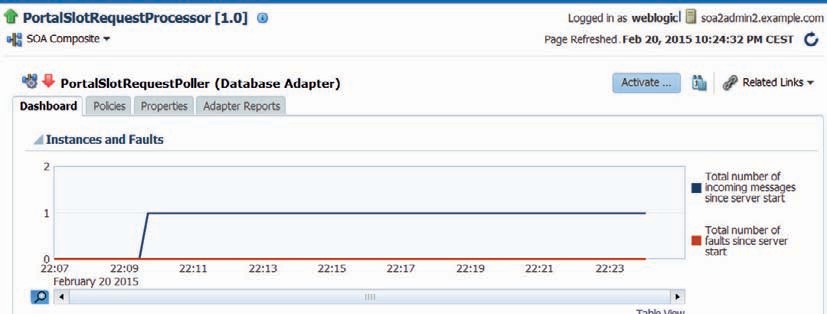


**图 14-4。** *将 ESS 调度与绑定到调度的数据库适配器相关联*

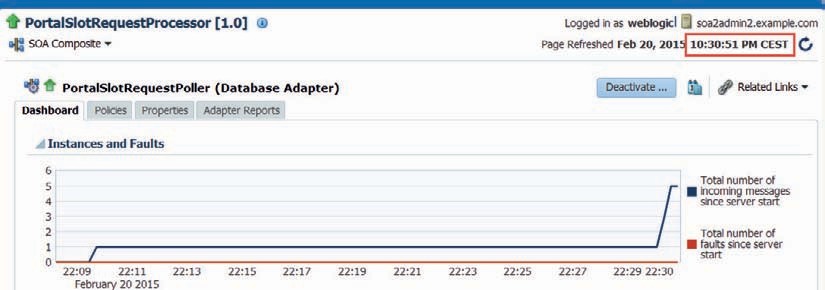
*停用和激活*

#### 测试数据库适配器绑定的关闭和打开

应用激活和停用计划后，它们将立即 生效。您可以在 Dashboard 页面中测试入站数据库适配器绑定，如图 14-5 所示。



**图 14-5。** *显示 PortalSlotRequestPoller 数据库适配器绑定状态的仪表板*



**图 14-6。** *在整点后 30 分钟，适配器绑定将再次激活并处理*

*在两个轮询周期中插入的 4 条记录（分别在 22：11 和 22：12 插入）*

在这里，我们看到适配器绑定如何处理单个记录，在晚上 10：09 插入。在10：11和10：12，又有四条记录入到表PORTAL\_SLOT\_ALLOCATIONS中。但是，由于适配器绑定当前未处于活动状态，因此尚未处理这些记录。

在整点后 30 分钟（在本例中为 10：30），适配器将再次变为活动状态，并开始处理随后将在表中找到的记录。由于适配器配置为仅将单个记录传递给 SOA 组合，并且不会在单个事务中处理超过两条记录，因此需要两个轮询周期来处理在 10：10 到 10：30 之间插入的四条记录。图 14-6 对此进行了说明。

当您在 EM FMW Control 中签入 ESS UI 时，您会发现两个新的作业定义，即用于执行 SOA Suite 管理任务的通用作业。在“作业请求”概述中，这些作业的实例每小时显示一次。这些作业请求的详细信息指定了哪个组合中的适配器绑定是该作业执行的 SOA 管理操作的目标。

您可以自己提交对这些作业定义的请求，为应用程序属性提供不同的值 ，以激活或停用选定的入站适配器。

### 安排每日航班数据传输

在 Saibot，Future 数据库 [schema] 保存有关当前和未来插槽分配的数据。根据此架构中的数据，Saibot 每天推导出未来 24 小时内将要进行的航班。有关这些航班的详细信息被加载到当前数据库 [schema] 中，该数据库是在航站楼、网站和移动应用程序中向旅客展示的航班信息系统的来源。Saibot 使用的第三个数据库 [schema] 称为 Past。它保存了一周或更长时间前的航班和飞机移动数据。每天，数据都会从“现在”移动到“过去”，以便尽可能精简“现在”（实时操作数据库）。

上面描述的两次每日数据移动是 ESS 计划编排的良好候选者。我们将重点关注第一个。在线源中提供了一个 PL/SQL 包，该包使用日期参数调用，并将根据 Future 中的时隙分配详细信息在 Present 中创建飞行记录。

最初，我们将创建一个简单的 SOA 组合应用程序，该应用程序使用数据库适配器来调用此包。然后，我们将创建一个用于调用 SOA 组合的 ESS 作业定义，并创建一个将该作业定义与每日计划相关联的作业请求。随后，我们将把 SOA 组合从等式中剔除，并创建一个 PL/SQL 作业定义，以直接从 ESS 到 PL/SQL 执行日常数据创建作业。

#### 创建 SOA 组合 PresentFromFuture

创建名为 PresentFromFuture *的 SOA 组合应用程序和项目*。创建一个名为 *PreparePresentFromFutureMediator* 的调解器，最初没有接口。

添加一个名为 CallFlightDataManager *的出站数据库适配器*，以调用包 flight\_data\_manager 中present\_from\_future的过程derive\_。

在公开的服务泳道中单击鼠标右键，然后选择“插入”|”肥皂。调用服务 *PreparePresentFromFutureService，并让它在* PresentFromFuture.wsdl 中实现单个端口类型。

将服务连接到调解器，将调解器连接到出站数据库适配器绑定。

图 14-7 显示了复合材料的状态。

使用调解器中的“赋值”设置将值从请求消息传递到 数据库适配器绑定，然后传递到 PL/SQL 过程。

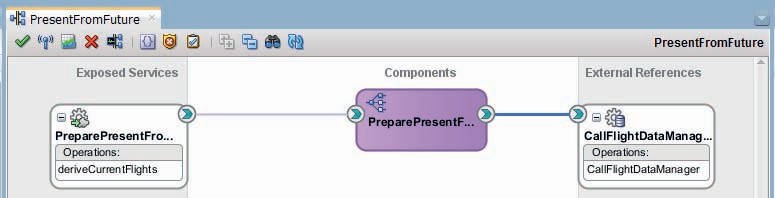
部署 SOA 组合。进行测试调用，看看它是否成功完成，并在数据库中执行您期望它执行的操作。 注意：我们不打算在将来进行此类直接调用 ，这就是 ESS 将为我们所做的。

#### 使用 ESS 创建和计划 PresentFromFuture 的每日执行

现在，我们将创建作业定义来调用 *PresentFromFuture* 组合，并计划它 每天执行一次。

##### 创建作业定义 CreateCurrentFlightData

打开 EM FMW 控件并转到 ESS 页面。从下拉菜单中，选择“作业 元数据”|”作业定义。单击创建按钮。



**图 14-7。** *SOA 组合 PresentFromFuture — 一个简单的 Web 服务到 PL/SQL 包的流程*

编辑新的作业定义。名称为 *CreateCurrentFlightData*，显示名称为“派生当前航班”，包为 [oracle/apps/ess/custom/]saibot/flight。作业类型为 *SynchWebserviceJobType*。单击“选择 Web 服务”。此时将显示一个弹出窗口。输入 PresentFromFuture  *复合公开的服务的 WSDL 的*  URL。将“Web 服务类型”设置为“SOA”。按 Go 按钮。

检索并处理 WSDL。将显示（一个）服务的列表 - 单击 *PreparePresentFromFutureService*。标记为“端口类型”的下拉元素 现已填充。选择 *“PreparePresentFromFutureService\_pt* ”选项。这将扩展对话框以包含包含操作的下拉列表。选择操作 *deriveCurrentFlights*。 将显示此操作的示例请求消息。将 *PeriodDuration* 元素的值设置为 ${ESS\_REQ：NUMBER\_ OF\_HOURS}。这是一个占位符，指示 ESS 将其替换为 为特定作业请求设置的应用程序属性的实际值NUMBER\_OF\_HOURS。有关完整的作业定义，请参见图 14-8 。

最后，将应用程序定义的属性添加到作业定义中。此属性称为 NUMBER\_OF\_HOURS，其类型为 Integer。初始值为 24，并且该属性不是 只读的。

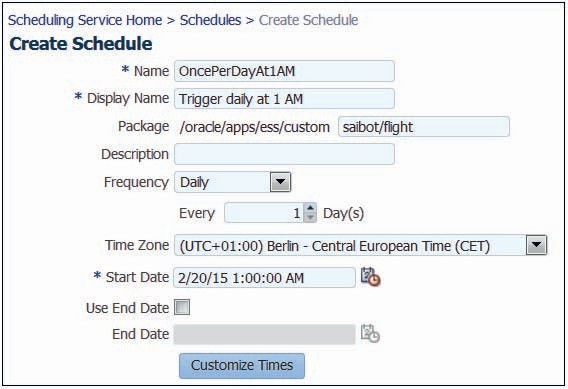
##### 创建“每天一次”计划

使用作业请求 |在 EM FMW Control 的“计划服务”页面上的下拉菜单中定义计划。单击创建图标。此时将显示创建新计划的页面，请参见图 14-9。将名称设置为 OncePerDayAt1AM，将显示名称设置为类似名称。



**图 14-8。** *配置涉及调用*

*由 SOA 组合 PresentFromFuture 公开的同步 Web 服务*



**图 14-9。** *配置 OncePerDayAt1AM 计划*

将 package 设置为 [/oracle/apps/ess/custom/]saibot/flight。将频率设置为每天，将开始日期设置为过去或不久的将来的任何日期。将开始日期的时间部分设置为 01：00：00 AM ，以确保此计划在每天的某个时间触发。按 OK 完成计划 定义。

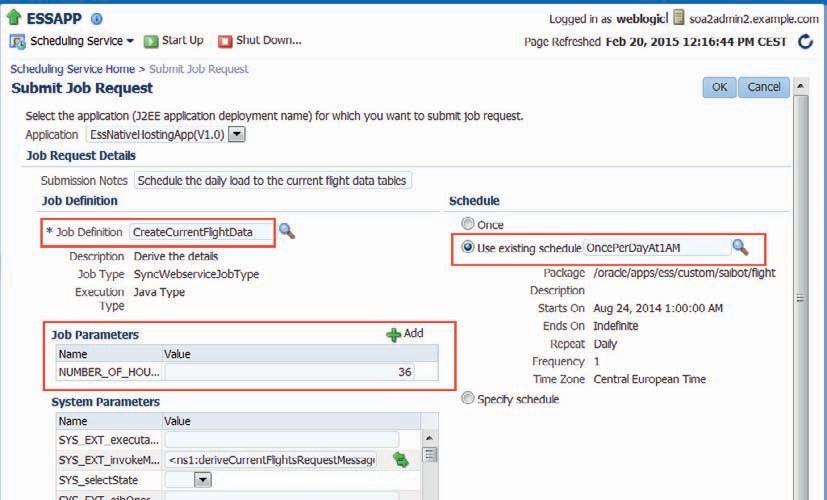
通过仅创建此计划，我们尚未完成任何工作，除了为想要提交作业请求的任何人提供此计划定义 - 其中包括计划和作业定义的组合（可选）以及应用程序和系统属性的特定值。

##### 创建作业请求以每天执行一次 CreateCurrentFlightData

在 EM FMW Control 中打开“调度服务”页面上的下拉菜单。点击“Job Requests”|“工作请求”提交工作请求。

此时将显示用于创建作业请求提交的表单，请参见图 14-10。选择作业 定义 *CreateCurrentFlightData*。选择 *OncePerDayAt1AM* 计划。提供一些 提交说明，并可能为应用程序属性 NUMBER\_OF\_ HOURS 提供特殊值，以覆盖此提交导致的每次作业执行的默认值 24。 按 OK 填写表单并提交工作请求。

提交此请求后，将分配一个请求 ID 并报告回来。具有此 ID 的请求将是 ESS 将生成的 CreateCurrentFlightData 作业的特定作业请求的父请求。第一个立即创建，计划在明天凌晨 1 点执行，请参见图 14-11。

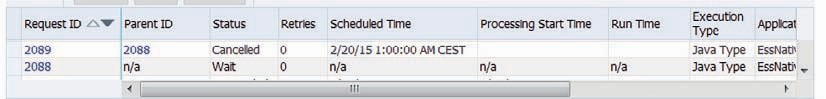


**图 14-10。** *根据 CreateCurrentFlightData 的作业请求根据*

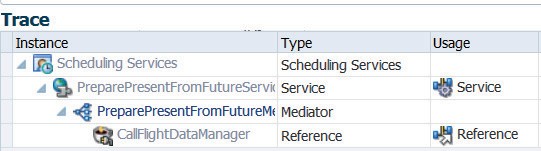
*OncePerDayAt1AM 时间表*

##### 测试我们的工作请求

在测试方面，我们真的无能为力，除了等待第一个凌晨 1 点到来，看看是否创建了 *PresentFromFuture* SOA 组合的实例。当然，我们可以稍微加快速度，只需稍微修改一下时间表，例如，将频率从每天更改为每小时。请注意，您必须将（相同但现在已修改）计划重新分配给作业请求，以使频率更改具有



**图 14-11。** *ESS 在提交作业请求后创建的作业请求记录*



**图 14-12。** *SOA 组合的定时执行（调用）的流跟踪*

*现在从未来*

影响。或者，您可以为作业定义提交单个执行请求。这不会 测试调度，但它将确保作业的执行，从而确保 SOA 组合的执行。

在预定时间，ESS 将触发作业的执行，从而调用 SOA 组合，该组合将调用 PL/SQL 包，以在包含当前飞行信息的表中执行数据创建，请参见图 14-12。在下一个预定的时刻，这种情况将 再次发生。

#### 将当前飞行数据创建转变为 ESS PL/SQL 作业

在上一节中，我们创建了一个 SOA 组合来调用 PL/SQL 过程来执行数据创建。随后，我们安排了对 SOA 组合公开的服务的调用。果然，一切都很顺利。但是，在 Enterprise Scheduler Service 和 PL/SQL 过程之间实现 SOA 组合的价值非常小。它会产生一些额外的日志记录，基本上就是这样。它可以进行扩充、错误处理和后处理，但这些目前不是必需的。因此，我们给 SOA 套件施加了一点都没有价值的负载。此外，在 PL/SQL 包中完成的工作可能需要相当长的时间，甚至可能导致 SOA 组合的事务超时。

简言之，最好将当前飞行数据创建作为 PL/SQL 作业类型进行调度，而不涉及 SOA 套件。通过几个简单的步骤，这正是 我们将在本节中要做的事情。

##### 为 ESS PL/SQL 作业准备 Saibot 数据库架构

为了成功创建并执行 ESS 可以调用的 PL/SQL 包flight\_data\_manager的 PL/SQL 包装器，我们必须将 ESS 数据库架构中数据库对象的一些权限授予 Saibot 架构：在 ESS\_RUNTIME、ESS\_JOB 和 ESS\_ SCHJOB\_PROC 上执行。 此外，在 Saibot 架构中为 ESS 架构中的某些对象创建私有同义词： ESS\_RUNTIME和ESS\_JOB。

##### 创建 PL/SQL 包装过程

要从 ESS 作为 PL/SQL 作业调用的 PL/SQL 过程必须实现特定签名：提供 ESS 请求标识符的 varchar2 类型的单个输入变量。输入

PL/SQL 过程可以使用该标识符通过包ess\_runtime从 ESS 请求上下文中读取值。同一包ess\_runtime可用于将结果从 PL/SQL 过程传递回 ESS 引擎。

请注意，PL/SQL 过程处理自己的事务管理，ESS 引擎不协调事务。下一个代码片段是本章源代码中的精简版本：出于可读性目的，应删除其中一部分的所有异常处理。

 创建或替换

过程 ess\_job\_derive\_pres\_from\_fut （ varchar2 中的request\_handle

） 是

l\_number\_of\_hours 数字（5,0）;l\_number\_of\_flights数（5,0）;l\_request\_id 数字 ：= null;

开始

l\_request\_id ：= ess\_runtime.get\_request\_id（request\_handle）;l\_number\_of\_hours ：= ess\_runtime.get\_reqprop\_int（l\_request\_id， 'NUMBER\_OF\_

小时'）;

flight\_data\_manager.derive\_present\_from\_future （ p\_start\_time => trunc（sysdate）

、p\_hours => l\_number\_of\_hours

，p\_number\_of\_flights => l\_number\_of\_flights

);

-- 使用新值更新现有请求属性。ess\_runtime.update\_reqprop\_int（l\_request\_id， 'NUMBER\_OF\_FLIGHTS\_PROCESSED'，

l\_number\_of\_flights）;

犯;

结束ess\_job\_derive\_pres\_from\_fut;

##### 创建 PL/SQL 作业类型的 ESS 作业定义

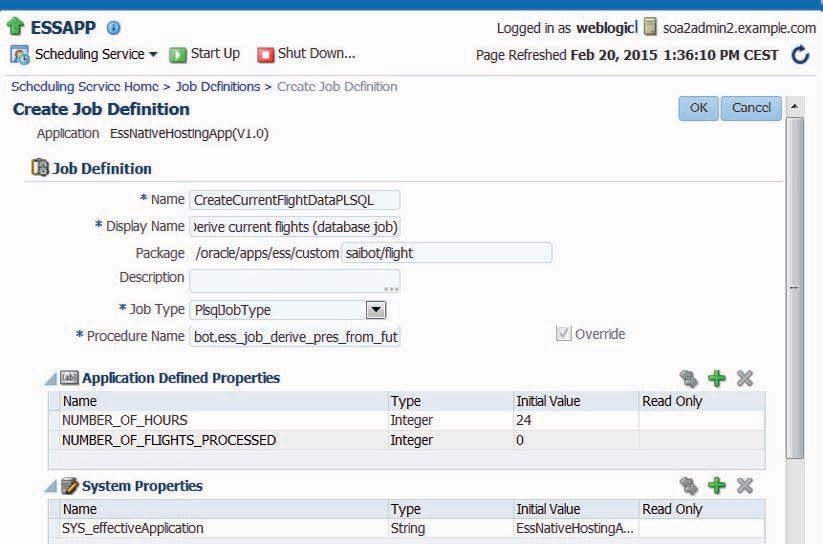
打开 EM FMW 控件并转到 ESS 页面。从下拉菜单中，选择“作业 元数据”|”作业定义。单击创建按钮。

编辑新的作业定义，如图 14-13 所示。名称为 *CreateCurrentFlightDataPLSQL*，包为 [oracle/apps/ess/custom/]saibot/flight。选择您自己的显示名称和说明。将“作业类型”设置为“PlsqlJobType”。将过程名称设置为“saibot.ess\_job\_derive\_pres\_from\_fut. Create”，设置为“应用程序定义的属性”，称为“NUMBER\_OF\_HOURS”和“NUMBER\_OF\_FLIGHTS\_PROCESSED”。两者都不是只读的，并且都是 Integer 类型。

按 OK 按钮创建作业定义。

##### 将 PL/SQL 作业安排为每日执行

我们刚刚创建的 PL/SQL 作业将接替我们之前创建的 SOA 作业 *CreateCurrentFlightData*，并为其提交了作业请求。可以使用“操作”|”请求详细信息页面上的“取消”选项。



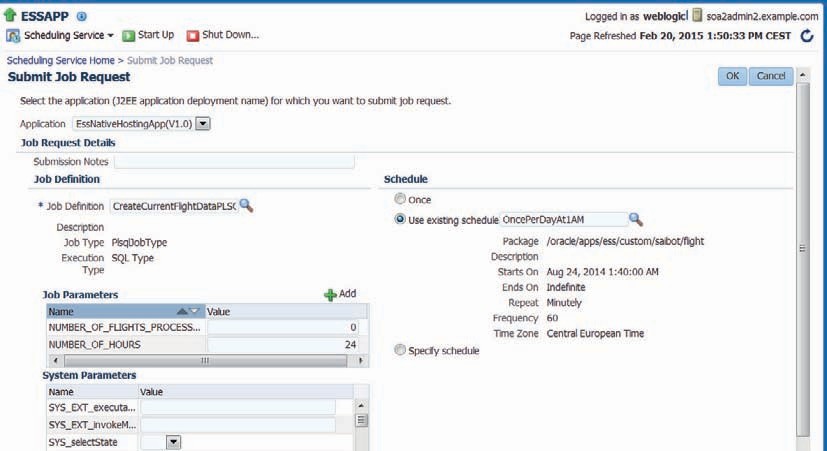
**图 14-13。** *创建 PL/SQL 作业类型的 CreateCurrentFlightDataPLSQL 作业定义*

*用于调用 PL/SQL 包装过程的类型ess\_job\_derive\_pres\_from\_fut*

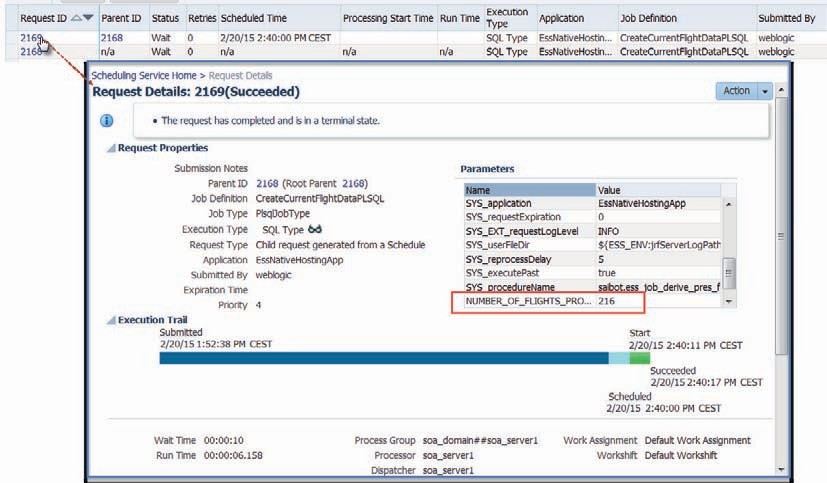
以与之前相同的方式为 PL/SQL 作业 *CreateCurrentFlightDataPLSQL*  提交新的作业请求：单击“Job Requests”（作业请求） |在下拉菜单中提交工作请求。此时将显示用于创建作业请求提交的表单。选择作业定义 *CreateCurrentFlightDataPLSQL*。选择 *OncePerDayAt1AM* 计划。提供一些提交说明，并可能为应用程序属性 NUMBER\_OF\_ HOURS 提供特殊值，以覆盖此提交导致的每次作业执行的默认值 24。 单击“确定”完成表单并提交作业请求，如图 14-14 所示。

再一次，除了等待计划触发作业的第一次执行之外，我们几乎无法进行任何测试。或者，您可以提交仅执行一次的第二个作业请求，只是为了尝试作业 定义本身。

作业请求完成后，作业请求参数 NUMBER\_OF\_FLIGHTS\_ PROCESSED 将包含从 PL/SQL 过程返回并由包装过程存储在作业请求中的编号。在 SOA 组合流跟踪中，我们找不到请求及其执行的痕迹，因为 SOA 套件不参与其中。EM FMW Control 中的 ESS 页面提供了有关我们在图 14-15 中提交的请求生成的每个子 Job Request 的详细信息。



**图 14-14。** *提交作业定义 CreateCurrentFlightDataPLSQL 的作业请求*



**图 14-15。** *有关请求 PL/SQL 作业定义 CreateCurrentFlightDataPLSQL 的详细信息*

### 从 SOA 组合延迟调用 ESS 作业

在本章的简介中，我们讨论了如何从 SOA 组合或服务总线管道调用 ESS 计划程序 Web 服务。在本节中，我们将看到一个示例。*我们在第 13 章中创建*的用于处理从门户应用程序提交的插槽分配请求的 SOA 组合 PortalSlotRequestProcessor 将被扩展。

如果插槽分配请求中有在接下来的 48 小时内有插槽请求，则复合将指示 ESS 运行*上一节*中定义的 CreateCurrentFlightDataPLSQL 作业。

这确保了在不久的将来从这些请求派生的新航班也被加载到当前航班表中。这是在 SOA 套件之外异步发生的，对 PortalSlotRequestProcessor *的执行影响很小*。

#### 将 BPEL 流程添加到组合中

将一个名为 IfDoRunPresFromFutJob *的 BPEL 组件*添加到组合中。它应该基于单向 BPEL 流程模板，而不是作为 SOAP 服务公开，并且它可以使用预定义的输入。

编辑 JDeveloper 生成的 XSD 文件 IfDoRunPresFromFutJob.xsd。将输入元素的名称更改为 *slotStartDate* ，并将其类型更改为 *dateTime*。

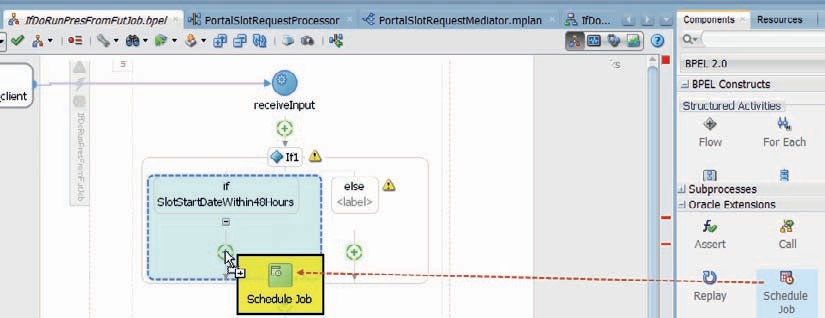
打开 BPEL 流程进行编辑。添加一个 If 活动，其中包含一个表达式，用于测试

*slotStartDate* 距离现在起不到 48 小时。

 $inputVariable.payload/client：slotStartDate <= xp20：add-dayTimeDuration-to- dateTime（xp20：current-dateTime（），'PT48H'）

将*“计划作业*”活动添加到“*if*”分支，请参见图 14-16。将“空”活动拖到

*else* 分支。



**图 14-16。** *向 BPEL 流程添加“计划作业”活动*

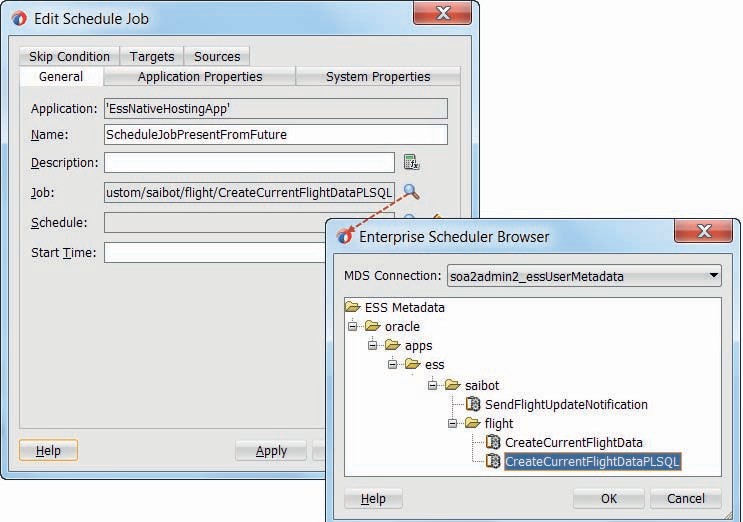
下一步是编辑*“计划作业*”活动。请注意，为此，JDeveloper 中需要有一个可用的特殊连接：连接到 MDS 中的 *essUserMetadata* 分区的 MDS 数据库连接。

设置该连接后，双击“计划作业”活动。将活动的名称设置为 *ScheduleJobPresentFromFuture*。选择作业 *CreateCurrentFlightDataPLSQL;*这将 填充 Application 字段。不要选择计划：这意味着我们希望 ESS 立即执行此作业。有关完整对话框，请参见图 14-17。单击“确定”完成 活动的定义。

关闭对话框时，将显示一条警告，指出 ESS 服务的 WSDL 是抽象的。单击“是”确认此消息 - 我们稍后将修复此问题 。

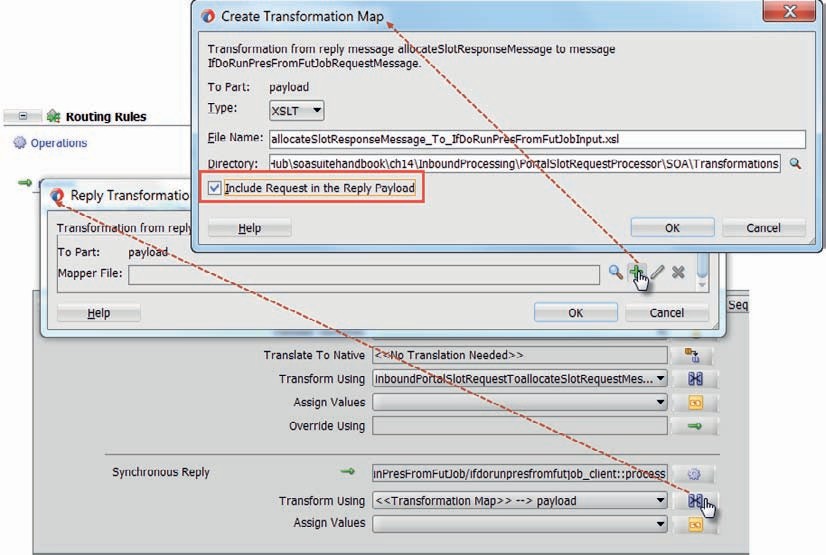
#### 从数据库适配器到 BPEL 流程的转发响应

返回到复合编辑器，然后双击调解器。在路由规则中，单击“同步应答”字段后面的齿轮图标，以设置从出站数据库适配器收到应答后将请求路由到的目标。选择 BPEL 流程公开的（现有）服务 作为此目标。



**图 14-17。** *编辑“计划作业”活动以使 ESS 立即运行*

*CreateCurrentFlightDataPLSQL 作业*



**图 14-18。** *使用原始请求为 BPEL 流程的输入创建映射*

*从入站数据库适配器作为输入参数*

单击转换图标以指定要用于转换的文件。在弹出窗口中，单击 加号图标以创建新的映射文件。在第二个弹出窗口中，将新映射器文件的名称设置为 allocateSlotResponseMessage\_To\_IfDoRunPresFromFutJobInput.xsl。选中复选框 *Include Request in the Reply Payload*。这指示中介在 响应转换发生时将输入路由规则的原始请求作为参数提供，这样我们就可以从请求中获取输入并将其传递给 BPEL 流程，参见图 14-18。

在 XSL 映射器中，将*原始请求中的* startDateForSlot 元素映射到

*slotStartDate* 元素。

#### 环境准备

除了功能实现之外，我们还必须做一些事情才能使这一切发挥作用。 有些与安全性有关，第 24 章对此进行了更详细的讨论。

##### 将策略附加到 ESS Web 服务

ESS中的作业不能以匿名用户身份运行。因此，如果 ESS 不知道谁发出了 调度作业的请求，则不会执行该请求。确保用户的身份是

当我们的 BPEL 流程调度作业时，我们必须将 WSM 安全策略附加 到 ESS Web Service，并在请求中传递一个带有有效用户名和密码的 WS Security Header。

步骤如下：在 EM FMW Control 中，单击 Scheduling Service |ESSAPP 在相关受控服务器上。从页面右侧的下拉菜单中，选择“Web 服务”选项。您将转到 Web 服务概述页。单击 *SchedulerServiceImplPort* 的链接。这会将您带到 *SchedulerServiceImplPort* 的另一个概述页面。打开标记为 WSM 策略的选项卡。单击标有“附加/分离”的图标。现在，您会发现自己位于可以将策略附加到此 Web 服务（端口绑定）的页面上。在可用策略列表中查找安全策略 oracle/wss\_username\_token\_service\_policy。单击“附加”按钮，将此策略附加到 ESS Web 服务。单击“确定”以确认此新策略附件。

此时，ESS 调度程序服务只能由提供有效用户名和密码的各方调用。因此，Web 服务的操作是在真实用户的上下文中执行的，就像通过 EM FMW 控件的 ESS UI 执行的与作业相关的操作一样，或者通过 Java API 从客户端应用程序执行的操作。

##### 为 ESS Web 服务绑定设置具体的 WSDL URL

要解决有关 *EssService* 的抽象 WSDL 的警告，我们需要获取实时 ESS Web 服务的 WSDL。在刚刚添加安全策略的 EM FMW 控件的页面上，还有一个指向 WSDL 文档的链接，该链接称为 *SchedulerServiceImplPort*。单击该链接可查看 WSDL 文档，更重要的是，要获取其 URL。

返回到复合编辑器。右键单击 *ESS Service* Reference Binding，然后从菜单中选择 Edit。在“更新引用”对话框的字段中设置 WSDL URL。

##### 将策略附加到 ESS 引用绑定

由于 ESS 计划程序 Web 服务受 WSM 安全策略保护，因此它要求调用方传递相应的 WS 安全标头。我们可以简单地附加一个（我们自己的）WSM策略来实现这种效果。我们甚至可以在运行时环境中通过 EM FMW 控制来做到这一点，而不是在设计时就在这里。但这次我们将选择设计时间， 开发人员路线。

右键单击 *EssService* 引用绑定。选择配置 SOA WS 策略 |对于菜单中的请求。此时将显示用于配置 SOA WS 策略的对话框。单击 “安全”类别的加号图标。从安全策略列表中，选择 oracle/wss\_username\_token\_client\_ 策略。然后按 OK。该策略附加到引用绑定。再次按 OK。

我们此时的配置将导致 OWSM 框架拦截从 SOA 组合到 *EssService*  的调用，并将 WS 安全策略注入其中。或者至少，这是它想要做的。但是，策略框架需要访问凭据才能放入 WS Security 标头中。这样做的正常方法是让策略框架检查配置的凭据存储中是否有要使用的用户名和密码。

但是，有一条捷径，我们将在这里使用。我们的安全策略可以简单地使用用户名和密码，而不是使用凭据存储，这些用户名和密码配置为策略附加到的引用绑定的属性。再次单击引用绑定。找到“复合属性”|”属性选项板中的绑定属性， 如图 14-19 所示。



**图 14-19。** *在引用绑定上设置属性 oracle.webservices.auth.username*

*EssService（英语：EssService）*

单击绿色加号图标以添加新属性。它的名称是 oracle.webservices.auth

.username，值例如 weblogic。然后添加第二个属性，称为 oracle

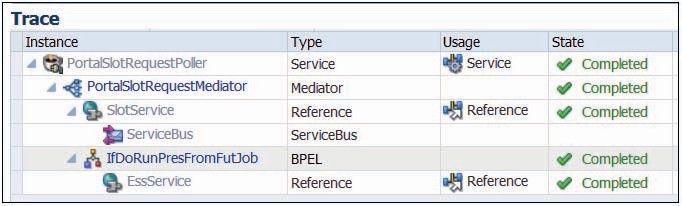
.webservices.auth.password 并设置其值。您会注意到，这两个属性不会显示在属性选项板中。无论多么烦人，这都不是问题：属性都添加到复合.xml文件中。

#### 部署和测试

部署并确保覆盖在第 13 章中部署的现有 PortalSlotRequestProcessor。

接下来，在表 PORTAL\_SLOT\_ALLOCATIONS 中插入一条记录，并确保START\_DATE\_FOR\_SLOT列在接下来的 48 小时内有一个值，因为这应该会触发 BPEL 流程调度作业。在提交事务并等待一段时间让入站数据库适配器执行轮询（确保它当前未被停用！）之后，您应该 找到一个新的 SOA 复合实例，其流跟踪如图 14-20 所示。

当您在 ESS 管理页面中检查作业请求时，您将看到 CreateCurrentFlightDataPLSQL  *的一次性请求*已创建并已执行。当您检查此请求的 ESS 请求日志时，您将找到触发此作业请求的 SOA 复合实例的 SOA 流 ID，该 ID 肯定与*入站数据库适配器创建的* PortalSlotRequestProcessor 实例相对应。



**图 14-20。** *由新的门户网站槽分配请求生成的消息流跟踪*

*由入站数据库适配器处理并导致将作业请求提交到 ESS 的记录*

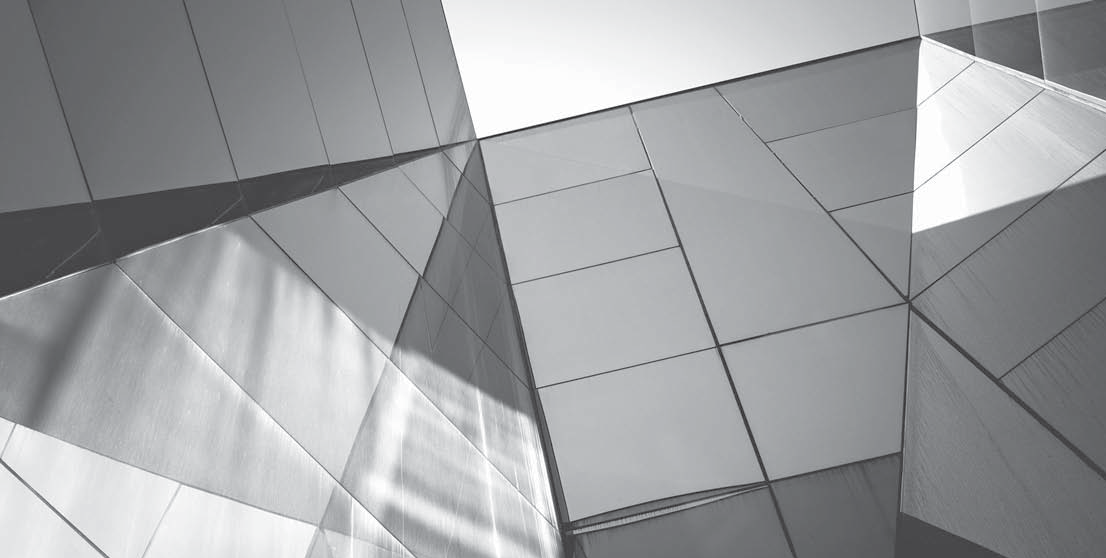
### 总结

Enterprise Scheduler Service 提供了一种根据特定计划来计划 SOA 组合以及其他几种作业类型的执行的方法。这为在后台、定期和在适当时刻完成工作提供了选项，例如实现自定义入站适配器，以轮询 Dropbox、Twitter 或 RSS 源。

一个强大的选项是能够将入站适配器与 ESS 计划相关联，以便停用和激活。这意味着我们可以根据外部系统的可用性和/或我们自己环境的容量来启动和关闭适配器绑定。激活或停用适配器绑定可以从 ESS 控制台或 SOA 组合完成 。

在 SOA 组合或服务总线管道中，我们可以调用 ESS 计划程序 Web 服务来提交作业请求。这意味着我们的服务实现可以轻松地 请求在服务执行的（事务）边界之外（在选定的时间点）异步执行各种类型的后处理。例如，我们可以将绩效评估流程服务安排在行李处理部门新员工的第一个工作日后 30 天调用。并且还可以运行 SOA 组合无法直接访问的操作系统进程。





章

# 15

## 事件 传送网络

OA是关于解耦和重用的，从而实现业务敏捷性。在前面的章节中，我们已经看到了许多解耦的例子，包括我们的 SOA 组合应用程序内部和它们之间，以及这些组合与外部服务和系统之间的解耦。

**S**

使用 XML 和 Web 服务标准有利于跨 异构技术堆栈（例如，文件系统、数据库、Java 应用程序、打包应用程序和 SOA 套件）的解耦互操作性。基于 WSDL 契约的服务组件和 SCA 基础架构之间的集成允许我们使用最好的工具来完成这项工作 — 业务逻辑的业务规则、路由和转换的 Mediator、有状态流程的 BPEL，以及利用其他平台功能的技术适配器。

将 JMS 和 AQ 等排入队列的异步功能，以及本章中介绍的事件，还为我们提供了时间解耦，其中使用者和提供者可以进行通信，而不必同时可用。

在解耦方面，我们至少还有一个挑战：如何确保在正确的时间调用服务？某些服务为其调用方提供明确的值，例如对最新航班信息的请求。每当应用程序需要此类服务的功能时，都会调用此类服务。应用程序在功能上与服务分离，规范模型和可重用的 Web 服务协定以及服务总线的终结点虚拟化负责处理这一点。但是，应用程序仍然需要显式调用服务，需要知道一些终结点位置，并根据服务协定工作。

对于单程服务来说，这个故事更有趣，当航班被取消或没有服务提供商对某个航空公司提供的合同进行投标时，可能需要执行一些操作。谁负责调用这些服务？没有人会打电话给他们从他们那里得到一些东西——因为他们没有回复。其他服务和应用程序可能拥有单向服务需要获取的信息。但是，谁有责任把它送到他们手中呢？这些信息所有者应该如何知道哪些单向服务对他们的数据感兴趣？他们是否应该有责任明确调用这些服务？当然，我们不希望在出现新的信息使用者或现有消费者失去兴趣时修改和重新部署应用程序。那根本不是脱钩！

本章将介绍 SOA Suite 中的事件交付网络 （EDN），它是一种通过调解生产者和消费者之间互不了解的事件来提供高级解耦的工具。复合应用程序可以订阅一个或多个集中定义的业务事件类型，并在发布其中一种类型的实例时收到 EDN 的通知。更具体地说，调解器和 BPEL 组件都可以生成和使用事件。还可以通过 BPEL 组件将事件关联到正在运行的复合实例中。可以从本地和远程 Java 客户端以及 Oracle Event Processor（请参阅下一章）和 BPM 流程（参见第 19 章）监听或发布 EDN。注意：服务总线不会以本机方式与 EDN 交互;但是，EDN 的基础实现是 JMS，当然服务总线可以与 JMS 主题进行交互。

### 用于超级解耦的事件驱动架构

事件驱动架构（EDA的首字母缩写）引起了很多人的注意。 EDA 被一些人看作是 SOA 的继承者，而另一些人则认为它是 SOA 的受欢迎的补充，EDA 显然正在 做一些事情。极度解耦架构将是首字母缩略词的完美次要含义，因为极端解耦是 EDA 在“传统”中增加的内容之一

SOA。在 EDA 的世界里，事件就是消息，取代了直接的服务调用。事件不针对特定的服务提供商。事件在事件生成者以即发即弃模式使用的一些中央通用基础结构上以某种逻辑名称发布。发布事件后，这些制作者不再对此负责。他们已经完成了他们的工作，将活动交给了中央活动协调设施，并且不再对活动附加任何条件。事实上，它们以后甚至可能像任何其他使用者一样使用自己的事件，因为事件使用者不知道它们的来源。事件通常具有标头和有效负载。标头包含元数据，例如事件类型和时间戳。正文或有效负载包含描述使用者中的业务逻辑将处理的事件实例的事实的详细信息 。

任何对特定类型事件的发生感兴趣的人都可以订阅它，通过协调事件的中央设施注册他们对该事件类型的兴趣。此协调工具接收发布的所有事件，从而减轻事件发布者对事件的责任。收到事件后，它将遍历特定事件类型的所有订阅，同时考虑可能已在此类订阅上定义的任何筛选器，以查看特定事件是否实际应转发给订阅的使用者，并将事件传播到所有符合条件的使用者。事件协调器可以支持在初始失败时重试传递事件的机制，或者为当前不可用的使用者延迟传递事件。

###### 注意

*没有一个事件的制作者知道事件的订阅。 即使根本不存在订阅，他们也应继续发布其事件。他们既不应该知道，也不应该关心。*

SOA 2.0 和事件驱动 SOA 这两个术语已经在 SOA 社区中取得了一些进展。

它们都表示一种面向服务的体系结构，其中事件机制用于进一步将应用程序和服务彼此分离。事件不是将作为业务事件源的应用程序和服务与需要信息的（通常是单向的）服务耦合在一起，而是使用事件通过通用的促进媒介（事件协调器）来传递数据。在 SOA 套件中，事件协调者的角色由事件交付网络实现。

**事件传送网络简介**

Oracle SOA Suite 附带事件交付网络 （EDN），这是一种基础结构，它提供了一种声明性方式来定义、发布和注册业务事件的使用。 EDN 支持在 SOA 套件中实现 EDA 模式。

事件传递网络是一个中间人，一个与三种类型的实体交互的中央协调器：事件的发布者、事件的使用者和事件本身。发布者（复合 SOA 应用程序或外部各方，如 Java 应用程序或运行在数据库中的 PL/SQL 代码）创建一个事件，并通过告诉中间人来发布它。

但是，在发布事件之前，需要对其元定义到位，包括（完全限定的）名称和其有效负载的 XSD 定义。事件的有效负载是与其关联的数据，由发布者提供并可供使用者使用。业务事件的元定义在 EDL（事件定义语言）中定义。EDL 是

部署在复合材料内部或存储在 MDS 中。这些事件定义文件通常导入一个或多个 XSD 文档，这些文档提供事件负载所基于的元素定义。

EDL 只是另一种 XML 语言 — 它本身基于在 JDeveloper 中注册的 XSD（JAR 文件 bpm-ide- common.jar 中的 edl.xsd）。请注意，没有对 EDL 文件的显式引用，不是来自 composite.xml 或任何组件的定义文件。项目中的所有 EDL 文件都有助于提供组件可以订阅或发布的事件，并且部署到 SOA Suite 实例的所有 EDL 文件都可用于该 SOA Suite 中的所有组合。一个 EDL 文件可以包含事件类型的多个定义。

事件交付网络在 SOA 套件之外工作，协调来自 SCA 容器中运行的所有组合的事件。因此，理想情况下，事件定义应该是 通用的，基于规范数据模型定义。

向 EDN 注册事件定义后，可以针对这些事件创建订阅。SOA 组合应用程序 通过使用特定事件的 Mediator 或 BPEL 组件向 EDN 注册它们对特定类型事件的兴趣。BPMN 组件（ 将在第 19 章中介绍）也可以订阅作为 BPMN 信号事件使用的 EDN 事件。许多不同的复合应用程序可以订阅同一事件。

其中每个应用程序都将收到来自事件传送网络的事件通知。

在部署具有事件消耗组件的应用程序时，EDN 会自动检测订阅，并用于分发这些事件的已发布实例。这些订阅是持久的 - 当复合由于某种原因被停用时，它们将继续存在。重新激活后，将传递在复合停机期间发布的事件。

取消部署将结束对 SOA 组合的订阅，而 Retire 也会结束订阅。

事件的发布者不必事先注册;任何人都可以发布通过 EDL 定义的类型的事件。特定类型的事件可以由许多不同的发布者发布，无论是在 SCA 容器内部还是在容器外部。事件交付网络事件的主要发布者 是 Mediator、BPMN 和 BPEL 服务组件，以及 OEP 应用程序和 Java 客户端。此外，可以将 ADF Business Components 配置为 在对实体对象进行数据操作时将事件发布到 EDN。EM FMW 控件具有用于发布事件以进行测试的工具，我们将在本章后面使用该功能 。

### FlightStatusUpdate 事件

FlightStatusUpdate 事件是 Saibot 机场的主要业务事件 之一。每当航班周围发生某些事情时，当它在着陆后 20 分钟内、着陆或到达登机口时，当清洁人员完成工作或行李撞到皮带时，都会发布一个事件。机场的许多不同方都利用这些活动来协调自己的活动。例如，所有公司都签订了为特定航班提供某种飞机服务的合同。以及所有参与通知对飞行表示兴趣的人。

在本节中，我们将*在 EDL 文件（EDN 的定义格式）中*定义 FlightStatusUpdate 事件。然后，我们将创建一个新的 SOA 组合，该组合使用中介器注册此事件，并为注册了对航班的积极兴趣的每个人和渠道*（电子邮件、Twitter、聊天）*调用 **FlightUpdateNotificationService**。我们将首先使用 EM FMW 控件中的测试工具在 EDN 上发布事件，然后我们将了解如何从服务总线服务发布事件。

#### 定义业务事件 FlightStatusUpdate

*FlightStatusUpdate*  事件在 EDL 文件中定义为具有规范有效负载的机场范围事件类型，不特定于任何应用程序或服务，并且基于规范数据模型中定义的元素。请注意，EDL 文件必须作为 SOA 组合应用程序的一部分进行创建和部署，即使它实际上不属于任何一个应用程序——它适用于整个 SOA Suite 容器，并且或多或少是“社区财产”。 在第 21 章中，我们将了解如何使用 MDS 来集中存储、共享和管理 EDL 定义。

使用 Empty Composite 模板*创建具有相同名称的 SOA 复合项目的 SOA* 应用程序 FlightStatusRelay。

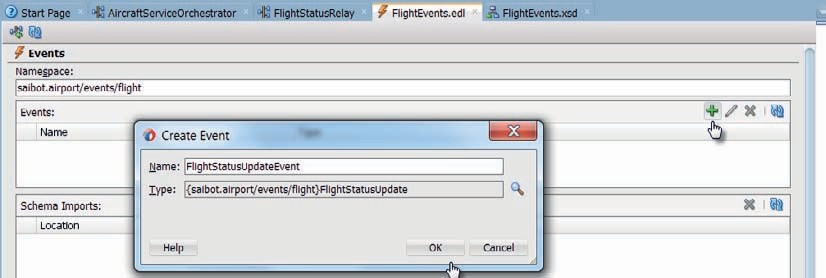
右键单击文件夹 SOA |事件，然后选择新建 |事件。输入 *FlightEvents* 作为 EDL 文件的名称。将命名空间设置为 saibot.airport/events/flight。单击绿色加号图标以添加第一个事件定义。在“类型选择器”弹出窗口中*，从 XSD FlightEvents.xsd*  中选择元素 FlightStatusUpdate。输入*“FlightStatusUpdateEvent*”作为事件名称，如图15-1所示。单击“确定”，完成事件创建。

在此阶段，业务事件已定义完毕。目前还没有使用者，运行时 EDN 不知道它，也没有人发布事件。

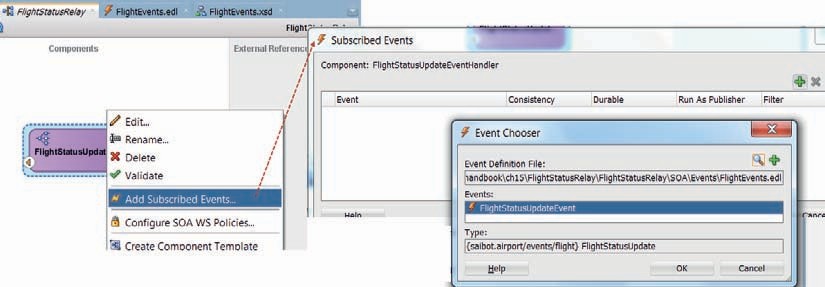
#### 使用 EDN 中的 FlightStatusUpdate

为了使用特定事件，例如*来自事件传送网络的* FlightStatusUpdateEvent，SOA 组合需要通过 EDN 订阅该事件。可以通过两种方式创建订阅：通过 Mediator 或 BPEL 组件（或 BPMN 组件，我们将在第 19 章中看到）。调解器路由规则可以将事件作为其源，而不是通过网络从服务绑定或其他 SCA 组件传递的请求消息。BPEL 组件可以有一个 Receive 或（中间进程）OnMessage 活动来侦听事件，而不是一个伙伴链接。

将名为 FlightStatusUpdateEventHandler  *的 Mediator 组件添加到*应用程序，此时没有接口。右键单击已创建的调解器，然后选择添加订阅的事件。单击出现的对话框中的绿色加号图标，然后浏览 *FlightStatusUpdateEvent*，如下所示



**图 15-1。** *在 FlightEvents.edl 文件中定义 FlightStatusUpdateEvent*



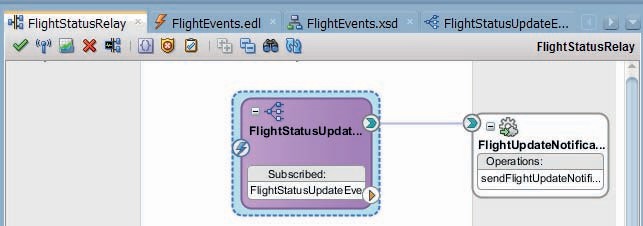
**图 15-2。** *将调解器配置为使用 FlightStatusUpdate 事件*

如图 15-2 所示。然后单击“确定”和“确定”以确认调解员对事件的订阅，默认设置为“一致性”（一个且只有一个）、“持久”（是）、“以发布者身份运行”（是）和“ 筛选器”（无）。

使用此事件订阅，每当在 EDN 上发布 *FlightStatusUpdateEvent* 时，都会创建此 *FlightStatusRelay*  复合的新实例。事件的有效负载将传输到此新实例，并将成为调解器的输入。此时，事件有效负载的后续处理与到达的常规请求消息没有区别。

由于我们接受的默认设置将订阅配置为 *“持久”*，因此可以保证将事件传递到复合。即使它暂时处于离线状态，事件也会被存储起来，并在它重新联机后转发到复合体。我们接受的另一个默认设置是在事件发布者的安全上下文中运行事件处理 。

接下来，在 SOA 组合中为 *FlightUpdateNotificationService* 创建一个引用绑定，上次出现在第 10 章中。将调解器连接到此参考，如图 15-3 所示。接下来，打开调解器进行编辑。使用“赋值”对话框进行复制



**图 15-3。** *由 FlightStatusUpdateEvent 启动的 FlightStatusRelay 复合*

*并调用 FlightUpdateNotificationService*

将*事件中的 Carrier*、*FlightNumber* 和 *FlightStatus 更改为* FlightUpdateNotificationService *的输入*。此外，定义通信元数据的值，尤其是 *收件人*地址和*通道*（电子邮件、即时消息、推特）。

部署 SOA 组合。这将使运行时 EDN 环境意识到

*FlightStatusUpdateEvent* 及其目前唯一的使用者。是时候让事件进来了。

##### 从 EM FMW 控件发布测试事件

打开 EM FMW 控件。单击节点 SOA |SOA-基础设施。从下拉菜单中（如图 15-4 所示）中，选择 *Business Events*。

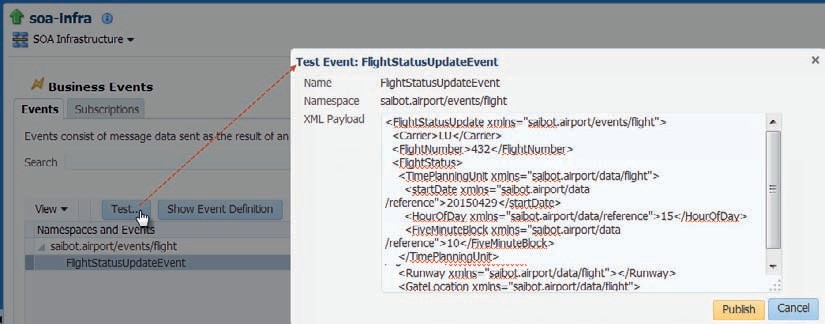
您将找到所有已知事件定义的列表，其中包含单个事件：*FlightStatusUpdateEvent*。您可以检查事件的定义，浏览此事件的订阅及其状态。在 EM FMW 控件中，您可以查看已订阅业务事件的服务组件。服务组件预订是在设计时在 Oracle JDeveloper 中创建的，无法在 Oracle Enterprise Manager Fusion Middleware Control 中修改。

您还可以通过单击“测试”按钮来发布事件。您必须提供构成事件有效负载的 XML 消息。在本例中，这意味着基于 FlightStatusUpdate  *元素的 XML 片段*，请参见图 15-5。

按“发布”将事件传输到事件传送网络。将出现一个弹出窗口 ，通知您事件已成功发布。



**图 15-4。** *在 EM FMW Control 中打开 Business Events 控制台*



**图 15-5。** *将 FlightStatusUpdateEvent 从 EM FMW 控件发布到 EDN*

EDN 吸收事件，检查订阅者并将事件传递给每个订阅者。

在本例中， *FlightStatusRelay* 复合由于其订阅而实例化，并且事件的有效负载将传输到 *FlightStatusUpdateEventHandler* Mediator 组件，请参见 图 15-6，了解从该事件开始的消息流跟踪。

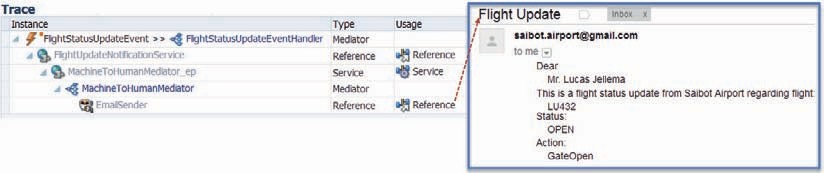
##### 幕后花絮：JMS

当您打开 WLS 管理控制台并在 *SOAJMSModule* 中检查 EDNTopic  时，您将在 Monitoring 选项卡中找到一个持久订户。这表示 FlightStatusRelay 复合中调解器组件中的事件订阅 。

打开 EM FMW 控件，然后单击节点 SOA | soa-infra。导航到 “已部署的复合材料”选项卡。单击 *FlightStatusRelay* 复合。然后单击“关机”按钮 ，以暂时停用复合。

再次发布 FlightStatusUpdateEvent。

复合的订阅仍然存在。但是该复合体尚未启动并运行。现在发生的情况是，该事件在 JMS 主题中举行，直到组合准备好为止



**图 15-6。** *FlightStatusUpdateEvent 触发了 FlightStatusRelay 复合*

*调用了将电子邮件发送给感兴趣的旅客的 FlightUpdateNotificationService*

再次食用。在 WLS 管理控制台中，在 *EDNTopic* 的“监视”选项卡中，当您单击持久订阅的*“显示消息*”按钮时，您将找到此传输中的消息。

下次再次启动复合时，此消息将从主题中消失，因为它可以传递到复合。您将找到一个新的复合实例，该实例 在复合再次处于活动状态后立即启动。

### 航空公司暂停事件

由于国际紧张局势和随之而来的制裁，以及金融动荡和商业纠纷升级，航空公司确实有可能被暂停。当这种情况发生时， 应该发布与此相关的业务事件。

在本节中，BPEL 流程将订阅此事件。当该事件被使用时，BPEL 流程将检索该航空公司当前航班的所有航班号，并通过发布 *FlightStatusUpdateEvent 的实例来发布这些航班的取消*。

随后，*第 12 章中引入的* AircraftServiceOrchestrator BPEL 流程将扩展为也使用该事件。当它这样做时，它将立即停止它为航空承运人启动的 RFP 流程。该事件通过关联到航空承运人的正在运行的实例中使用，它不会启动新实例。

#### 将 EDN 事件使用到新的 BPEL 流程中

查找所有未完成的航班，并发布已*暂停的承运商*现已取消的所有航班的 FlightStatusUpdateEvent。

##### 制备

首先，我们定义 SOA 应用程序、项目和 *AirCarrierSuspended* 事件定义。我们还 创建了一个简单的 BPEL 流程，该流程返回给定承运商的（静态、虚拟）航班列表。

**创建 SOA 应用程序和项目 基于空模板，使用名为** CarrierWatcher *的 SOA 组合（项目）创建名为* CarrierWatcher *的新 SOA 应用程序*。将文件 CarrierWatcher.xsd 复制到 Schemas 文件夹。

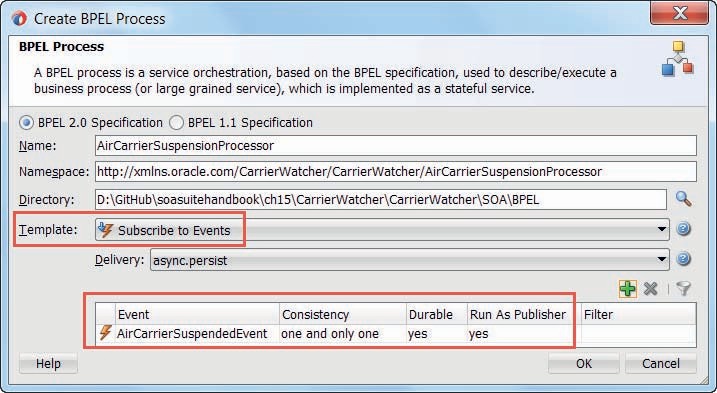
**定义 AirCarrierSuspendedEvent EDN**  事件 右键单击“事件”文件夹。选择新建 |上下文菜单中的事件定义。将 [edl 文件] 的名称设置为 *AircarrierEvents*，将命名空间设置为 saibot.airport/events/aircarrier。

单击绿色加号图标以创建新事件。将事件名称定义为 *AirCarrierSuspendedEvent*。

从 CarrierWatcher.xsd 中  *选择元素* carrierSuspension 作为事件负载。单击“确定”和“确定”以完成事件和 .edl 文件的定义。

**创建 BPEL 组件以收集所有当前航班 创建一个名为** CurrentFlightsRetriever *的同步 BPEL 流程*组件，该组件未公开为 SOAP 服务。输入基于 *CarrierWatcher.xsd 中的 CurrentFlightsRequest 元素和*  同一文件中 *CurrentFlightsResponse*  的输出。此组件接受当前航班的条件（在本例中仅接受航空承运人），并返回 48 小时时间窗口内所有当前航班的列表。

在真实世界的机场和真实的端到端应用程序中，该组件将调用 各种业务服务，这些业务服务反过来查询 存储航班详细信息的当前数据库。 但是，在本章的范围内，我们重点介绍事件，我们创建了一个非常简单的虚拟实现，该实现为传入IATA代码的航空公司返回一些航班。



**图 15-7。** *创建一个 BPEL 流程组件，该组件由*

*事件传送网络中的 AirCarrierSuspendedEvent*

打开 BPEL 流程。添加“分配”活动。打开编辑器。将文本 [XML 片段] 的图标拖动到 *outputVariable/payload* 元素。使用 CurrentFlights 中的内容。 xml 文档来填充片段。

##### 创建 BPEL 以使用 AirCarrierSuspendedEvent

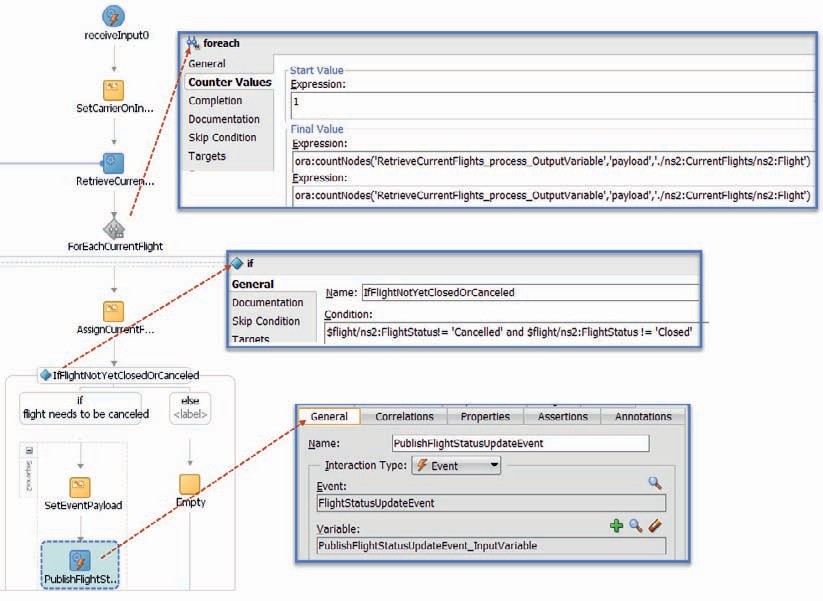
将新的 BPEL 组件添加到组合中。将名称设置为 *AirCarrierSuspensionProcessor*。选择模板 *Subscribe to Events*。单击绿色加号图标以创建事件订阅。将 BPEL 组件订阅到 *AirCarrierSuspendedEvent*，如图 15-7 所示。接受 默认设置。单击“确定”完成 BPEL 组件的创建。

将 BPEL 组件连接到 *CurrentFlightsRetriever*。接下来，打开组件进行编辑。将 Invoke 活动添加到 *CurrentFlightsRetriever* 组件。生成输入和输出变量。添加一个 Assign 活动，将触发 BPEL 组件的事件中的航空承运人代码映射到流向 *CurrentFlightsRetriever 的输入变量*。

添加 For-Each 活动。将活动配置为循环遍历从 invoke 活动返回的 *CurrentFlights*  列表中的所有 Flight 元素。在“If”活动中检查每个航班的状态是否等于“已取消”和“已关闭”，因为这些航班不需要进一步操作。 将 Invoke 活动添加到 If 分支。将此活动配置为不调用合作伙伴链接，而是发布 *FlightStatusUpdateEvent*。创建已使用的输入变量

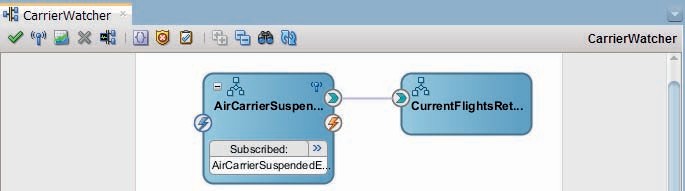
设置事件的有效负载。添加一个 Assign 活动（就在发布事件的 Invoke 活动的上方），以从 For Each 循环中的当前 flight 元素设置此输入变量的值 。图 15-8 显示了 BPEL 流程的样子。

整个复合如图 15-9 所示。没有服务或引用绑定。取而代之的是：进出此组合的输入和输出由第一个 BPEL 组件使用和发布的 EDN 事件组成。



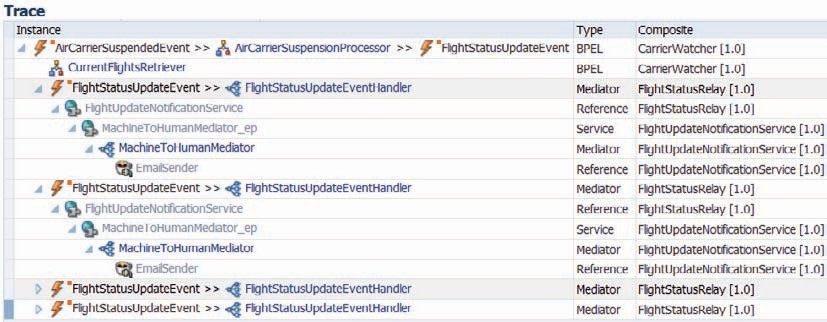
**图 15-8。** *BPEL流程，用于检索已暂停承运商的当前航班，以及*

*发布了每个相关航班的 FlightStatusUpdateEvent*



**图 15-9。** *订阅 AirCarrierSuspendedEvent 的 CarrierWatcher 复合*

*并发布了航空公司当前航班的 FlightStatusUpdateEvent*



**图 15-10。** *消息流跟踪，由 AirCarrierSuspendedEvent 启动，结果*

*在发布到 EDN 并由 FlightStatusRelay 复合处理的四个 FlightStatusUpdateEvent 实例中*

##### 部署和测试

部署 *CarrierWatcher* SOA 组合。然后从 EM FMW 控件发布 *AirCarrierSuspendedEvent* 实例。*应触发 CarrierWatcher* 组合，BPEL 组件应使用该事件，检索要取消的航班列表，并为每个航班发布 *FlightStatusUpdateEvent*。在 CarrierWatcher 组合的消息流跟踪中验证此活动 ，如图 15-10 所示。

##### 在事件订阅上定义复合传感器

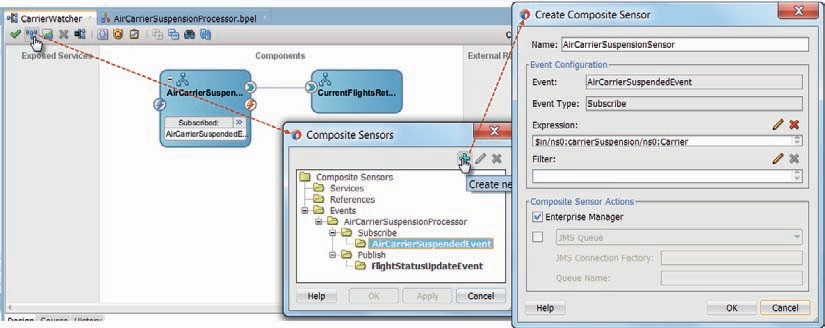
复合传感器（在第 5 章中首次出现）可以在服务和引用绑定上定义，以便在 SOA 复合实例上发布相关且通常标识的详细信息，这与服务总线管道中的消息报告非常相似。还可以在事件订阅中定义此类传感器，以跟踪已使用事件的复合实例的（相关信息）。

返回到 JDeveloper 并打开 *CarrierWatcher*  组合的复合编辑器。单击图标栏中的传感器图标，如图15-11所示。在弹出窗口中，选择已使用的 *AirCarrierSuspendedEvent*。单击加号图标以创建复合传感器。将传感器的名称设置为 *AirCarrierSuspensionSensor*。指定一个 XPath 表达式，该表达式检索（挂起的载波的）载波代码作为传感器的值。

###### 注意

*可以使用*  相同的 XPath 将表达式创建为*变量或表达式。但是，第一个选项将生成 iataAirlineType 作为输出，第二个方法将生成 String。后者更容易搜索。*

重新部署复合。发布 *AirCarrierSuspendedEvent* 的另一个实例。检查是否创建了复合实例。然后找出复合传感器值在 EM FMW 控制器中是否可用，如图 15-12 所示。



**图 15-11。** *创建一个复合传感器，该传感器在所有实例上公开载体代码*

*由 AirCarrierSuspendedEvent 触发的 CarrierWatcher 复合*

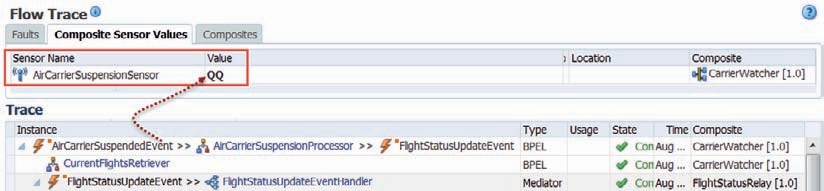
#### 消费到正在运行的 BPEL 流程 AircraftServiceNegotiationProcessor

*AircraftServiceOrchestrator* 复合让一家或多家飞机服务提供商参与，为航空公司征集建议。复合调用的服务是异步的，响应可能需要几个小时到几天才能到位。一旦请求的航空公司被暂停，整个 RFP 可以而且应该终止。这意味着 BPEL 组件 *AircraftServiceNegotiationProcessor* 应将 *AirCarrierSuspendedEvent* 使用到已挂起的承运人（如果有）的正在运行的实例中，并终止该实例。

在上一节中，我们已经了解了如何通过接收事件来实例化 BPEL 流程。现在，我们将更进一步，使用事件而不是 中间进程传入请求消息来关联到实例中。

##### 定义 BPEL 流程的关联集

打开 AircraftServiceOrchestrator 复合。打开 BPEL 组件 AircraftServiceNegotiationProcessor 进行编辑。



**图 15-12。** *派生自 AirCarrierSuspendedEvent 的复合传感器值*

添加一个名为 AirCarrierCorrelation *的关联集*。将 String 类型的属性 *AirCarrierIataCode* 添加到该集中。根据*传入的* ArrangeAircraftServiceRequestMessage *中的 RequestingCarrierIATACode 元素定义此属性的属性别名*。

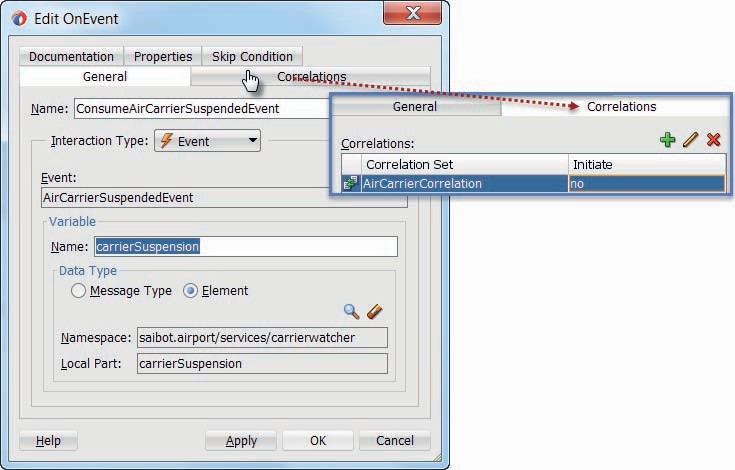
编辑启动 BPEL 流程的 Receive 活动。在“关联”选项卡上，添加新的 *AirCarrierCorrelation* 关联集。将 initiate 设置为 true - 此活动接收请求消息是为此 *BPEL 流程指定* AirCarrierIataCode 标识值的触发器。

##### 为 AirCarrierSuspendedEvent 添加异步事件处理程序

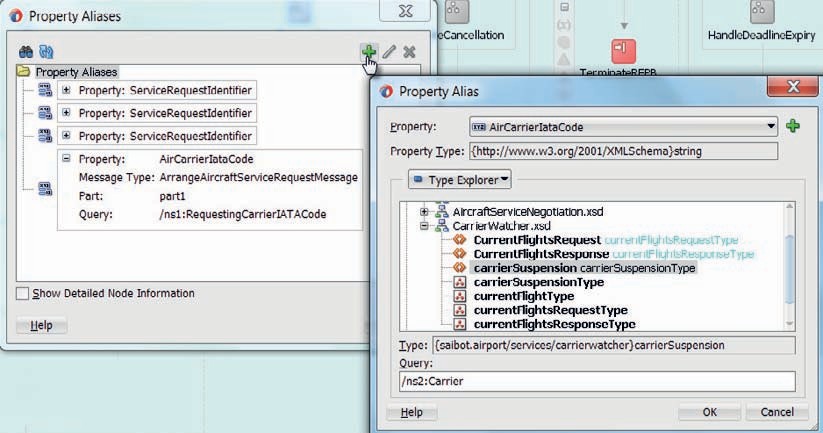
单击范围 *ExecuteRFP* 的 onEvent *图标*。编辑添加到作用域的 *onEvent* 处理程序。将名称设置为 *ConsumeAirCarrierSuspendedEvent*。从 *CarrierWatcher*  项目的 AircarriersEvents.edl 文件中选择 *AirCarrierSuspendedEvent*。允许将 edl 文件和关联的 XSD 文件复制到当前项目。

根据 CarrierWatcher.xsd 文档中该名称的元素*，将变量的名称设置为* carrierSuspension。onEvent处理程序的配置如图15-13  *所示*。

将 Exit 活动添加到 *onEvent* 范围。这是我们处理 *AirCarrierSuspendedEvent* 的第一种相当粗略的方法。当然，理想情况下，我们会通知所有参与 RFP 的飞机服务提供商，我们仍在等待他们的异步响应。这些是我们应该在现实世界中添加的改进。在这个例子中，它们并没有增加我们对将 EDN 事件关联到正在运行的 BPEL 流程实例的理解，因此我们忽略了它们。



**图 15-13。** *配置 onEvent 处理程序以将 AirCarrierSuspendedEvent 使用到具有关联集 AirCarrierCorrelation 的相应航空承运人值的实例中*



**图 15-14。** *基于承运商为属性 AirCarrierIataCode 添加属性别名*

*元素，元素在 AirCarrierSuspendedEvent 的有效负载中*

最后，将属性别名添加到关联集，该关联集将 *AirCarrierSuspendedEvent*  的有效负载中的 Carrier 元素映射到属性 *AirCarrierIataCode*，如图 15-14 所示。

##### 部署和测试

部署 SOA 复合 *AircraftServiceOrchestrator*。调用操作 *arrangeAircraftService*

请求消息中元素 *RequestingCarrierIATACode* 的值为 XX。

当您在 EM FMW 控制器中检查时，您会发现正在运行

*AircraftServiceOrchestrator* 实例，由您的请求启动。

使用 EM FMW 控件中的 Business Events 控制台发布 *AirCarrierSuspendedEvent* 的实例。将此事件的有效负载中的 Carrier 元素设置为 XX。正在运行的 *AircraftServiceOrchestrator* 实例应在我们刚刚添加的新 *onEvent*  处理程序中使用此事件。这反过来又会导致实例立即终止。

###### 注意

*BPEL 中的关联假定所有正在运行的实例之间设置的关联是唯一的。这意味着在这种情况下，* 任何时候都只能为特定航空承运人运行一个 *AircraftServiceOrchestrator 实例。在现实中，经常会出现这样的情况：一个事件应该被消耗到多个正在运行的实例中***，***我们需要一种比上面描述的更复杂的方法。该书的网站引用了一篇文章，介绍了一种*

*“扇出 EDN 事件”，以便一个事件影响多个实例。*

### 有关事件交付网络的更多信息

本节包含有关事件传送网络及其使用方式的许多有用详细信息。当然，这只是一个有限的收藏。查看在线资源了解更多详情。

#### 事务、安全性和可扩展性

使用事件订阅上的“一致性”设置，我们是否指示事件的传递是响应事件的事务的一部分（一个且只有一个），还是传递是在 单独的事务中（保证）。前者（和默认）设置意味着 当使用者中发生错误时回滚事件传递（并且可以再次传递事件），而后者会导致在回滚消费事务时此订阅的事件丢失。第三个值 immediate 指定在与发布者相同的全局事务和同一线程中将事件传递给订阅者，从而有效地（通常非常不希望）将发布者与使用者耦合。远离该选项， 除非您有充分的理由使用它。

将“以发布者身份运行”设置为“是”，以使用与发布事件的复合相同的安全上下文（标识）执行使用者。根据文档， 也可以设置一个企业角色来执行使用组合。但是， IDE 对此没有适当的支持。

SOA 套件不会以存档或审计跟踪或类似内容的形式跟踪 EDN 事件。当事件被复合使用时，复合流跟踪将反映事件的发生。与当前不可用的使用者的持久订阅关联的事件 保留在 JMS 级别。

如果 SOA 组合应用程序包含业务事件预订，并且部署了该组合的不同修订版，那么该组合的所有修订版中的所有事件预订都将处于 活动状态并接收消息。

安装后，将自动配置 Oracle WebLogic Server JMS 主题（缺省值）和 AQ JMS 主题以供 EDN 使用。可以在 Oracle Enterprise Manager Fusion Middleware Control 中将默认 JMS 类型从 Oracle WebLogic Server JMS（缺省值）切换到 AQ JMS。您可以在 Oracle Enterprise Manager Fusion Middleware Control 的“业务事件”页面上将业务事件映射到 JMS 主题目标，以降低所有事件流经单个 JMS 主题时出现瓶颈的风险。

###### 提示

*人工任务组件（将在第 17 章中介绍）可以配置为将每个任务和工作流事件的事件发布到事件交付网络 。这包括已分配、已认领、已拒绝、已上报和已完成的任务。*

###### 注意

*事件不受授权机制的约束：任何复合都可以在 EDN 上注册任何事件类型，并且每个事件实例都会传递给每个使用者（如果订阅上的可选筛选器表达式允许）。不能指定或强制执行任何授权。*

#### 缺失事件检测

事件表明发生了某些事情——或者你会这么认为。但是，当某些事情没有发生时，很有可能发生有意义的事件。当预期的没有发生时，这确实可能是一个相当大的事件。如果太阳或潮汐忘记升起，我们确实会有重大 事件。

在稍微小一点的范围内，我们也会遇到这样的情况，即我们预期的事情本身并没有真正发生，这是一个有意义的事件。发票已发送，但付款从未兑现;行李箱进入了行李处理系统，但从未出来;投诉已提交，但从未发出答复。此类事件通常称为非事件。没有事件本身就是一个事件。

事件处理中的一个常见挑战是检测非事件：应该发生（在某个时刻之前）但不会发生的事件。SOA Suite 不支持这种开箱即用的检测。我们必须实现一些逻辑来查找非事件，并将它们转换为真正的 EDN 业务事件。这是应对这一挑战的一种方法。

当飞机降落并开始滑行到登机口时，会将事件发布到事件传送网络（例如，*AircraftLandedEvent*），并将航班号作为有效负载。 复合应用程序订阅此事件，并在使用时启动一个新实例。这个组合包含一个使用事件的 BPEL 组件。它进入具有两个分支的 Pick 活动：一个是等待 30 分钟的 Wait 活动。另一个分支是 *onMessage*，它将等待 *AircraftAtGateEvent*。*onMessage* 会将后一个事件关联到航班号上。

当飞机到达登机口时，应发布此 *AircraftAtGateEvent*。当它出现时，BPEL 组件将使用它，结束 Pick 活动，并完全终止实例。但是，如果 30 分钟后仍未收到此事件，则会激活 Pick 活动的等待分支，并将发布“非事件”*AircraftDidNotArriveAtGateEvent*。BPEL 组件将业务事件的缺失（在指定时间段之后）转换为另一个业务事件，随后所有使用者都可以响应该事件。

### 从其他渠道与EDN进行事件交换

发布和使用 EDN 事件可以很容易地通过 Mediator 和 BPEL 流程完成。 Oracle Event Processor 具有与 EDN 交互的适配器，可以将 ADF 业务组件配置为发布到 EDN，并且任何能够通过 JMS 进行通信的人都可以与 EDN 交互。例如，这使得从服务总线将事件发布到 EDN 成为可能，尽管服务总线不了解 EDL 文件，也没有本机 EDN 集成。

在本节中，我们将首先了解一个简单的 Java 客户端与事件交付网络的交互。接下来，我们将特别关注作为事件的使用者和生产者的服务总线 。

#### 将 FlightStatusUpdateEvent 从 Java 客户端发布到 EDN

Java 客户端可以使用 Oracle 随 SOA 套件一起提供的 EDN Java API。通过使用此 API，可以处理有关通信和事件定义的大多数细节。或者，我们可以在普通的 JMS 级别进行通信，这让我们在 EDN 复杂性方面承担了更多的责任。

创建名为 EDNandAlternativeChannels *的服务总线应用程序*。创建一个名为 *EDNJavaClient* 的新 Java 项目。将以下库和 jar 文件添加到项目中：

* 库：SOA 运行时、WebLogic 12.1 远程客户端、JRF 运行时
* jar 文件：tracking-core.jar 和 Jca-binding-api.jar（注意：这些 jar 文件可以在目录 FMW\_HOME\Oracle\_Home\soa\soa\modules\oracle.soa.adapter\_11.1.1 和

FMW\_HOME\ Oracle\_Home\soa\soa\modules\oracle.soa.fabric\_11.1.1

创建一个名为 FlightStatusUpdateEventPublisher *的 Java 类*。添加以下代码，用于将 *FlightStatusUpdateEvent* 发布到事件传递网络。记下所提供的 SOA Suite 托管服务器的地址和登录详细信息;您需要根据自己的环境调整它们。请注意，为了简洁起见，省略了 import 语句。当然，您会在在线源中找到正确的导入：

 软件包 saibot.airport.events.client;

公共类 FlightStatusUpdateEventPublisher { public static void main（String[] args） {

属性 props = new Properties（）;props.put（Context.PROVIDER\_URL， “t3：//soa2admin2：8001/soa-infra”）;props.put（Context.INITIAL\_CONTEXT\_FACTORY， “weblogic.jndi.

WLInitialContextFactory“）;props.put（Context.SECURITY\_PRINCIPAL， “weblogic”）;props.put（Context.SECURITY\_CREDENTIALS， “weblogic1”）;字符串 FlightStatusUpdateEventBody =

“<FlightStatusUpdate xmlns=\”saibot.airport/events/flight\“>\n” + ”

<运营商>ZQ</运营商>\n“ +

“ <航班号>1134</航班号>\n” + “ <航班状态>\n” +

" <操作 xmlns=\“saibot.airport/data/flight\”>GateOpen</Action>\n“ + ” </FlightStatus>\n“ + ”</FlightStatusUpdate>“;

尝试 {

元素 eventBody = XMLUtil.parseDocumentFromXMLString（FlightStatusUpdateEv entBody.toString（））.getDocumentElement（）;

BusinessEventConnectionFactorySupport.setJndiProperties（props）;EdnConnection ec = （EdnConnection） BusinessEventConnectionFactorySupport。

findRelevantBusinessEventConnectionFactory（true）。createBusinessEventConnection（）;

字符串命名空间 = “saibot.airport/events/flight”;字符串 eventName = “FlightStatusUpdateEvent”;

BusinessEventBuilder 生成器 = BusinessEventBuilder.newInstance（）;builder.setEventName（new QName（命名空间， eventName））;builder.setBody（eventBody）;

ec.publishEvent（builder.createEvent（）， 4）;ec.close（）;

} catch （异常 e） {

e.printStackTrace（）;

}

}

}

运行此 Java 类时，将执行其 main 方法。它构建了一个与 SOA Suite 运行时服务器的 EdnConnection，而不必在 JMS 主题和连接工厂方面了解很多关于 EDN 配置的知识。通过 EdnConnection，可以轻松发布事件。当然，设置正确的命名空间和事件名称对于指示 EDN 了解 事件的性质至关重要。

运行该类并验证是否创建了 SOA 复合 *FlightStatusRelay*  的新实例。

#### 将 FlighStatusUpdateEvent 从 EDN 使用 到 Java 客户端

使用我们在上一节中创建的用于发布 EDN 事件的 JDeveloper 项目，我们已经很好地使用了来自 EDN 的事件。我们可以相当容易地创建一个事件订阅，并将我们自己的一个对象注册为侦听器。我们可以指定 要侦听的事件类型，也可以使用事件过滤器。EDN API 负责创建适当的 持久 JMS 主题订阅，并将事件传输到我们的侦听器。通过订阅，我们可以指定一个超时，指示我们的代码是否以及应该在多长时间内阻止等待 我们订阅的事件。

创建一个 Java 类 *FlightStatusUpdateEventSubscriber*。以下代码还建立了 EDN 连接。然后，它在 *FlightStatusUpdateEvent* 上创建一个订阅，将自身注册为侦听器。该类实现 EDN API 中定义的接口 *BusinessEventHandler*。*onEvent*  方法必须在 Class 中实现，以满足接口要求。每当事件在 EDN 上发布时，都会调用此方法。

 public class FlightStatusUpdateEventSubscriber implements BusinessEventHandler { public static void main（String[] args） {

属性 props = new Properties（）; props.put（Context.PROVIDER\_URL， “t3：//soa2admin2：8001/soa-infra”）; props.put（Context.INITIAL\_CONTEXT\_FACTORY， “weblogic.jndi.

WLInitialContextFactory“）;props.put（Context.SECURITY\_PRINCIPAL， “weblogic”）; props.put（Context.SECURITY\_CREDENTIALS， “weblogic1”）;

BusinessEventConnectionFactorySupport.setJndiProperties（props）;EdnConnection ec = （EdnConnection） BusinessEventConnectionFactorySupport。

findRelevantBusinessEventConnectionFactory（true）。 createBusinessEventConnection（）;

字符串命名空间 = “saibot.airport/events/flight”;字符串 eventName = “FlightStatusUpdateEvent”;

ec.setEventHandler（new FlightStatusUpdateEventSubscriber（））; ec.set超时（12000）;

ec.subscribe（new QName（namespace， eventName）， “mySubscriptionForFlightStatusUpdateEvent”）;

}

public void onEvent（BusinessEvent be） 抛出 FabricInvocationException { System.out.println（“嘿，我收到了一个类型为 ” + be.getEventName（）+“;

有效载荷 = “ + be.getBodyAsText（））;

}

}

运行此类。现在，*可以从 EM FMW 控件*发布 FlightStatusUpdateEvent，或者使用在上一节中创建的 Java 类 FlightStatusUpdateEventPublisher 。检查 *onEvent* 方法是否确实接收到该事件。

#### 从服务总线与 EDN 交互

如前所述，服务总线没有与事件传送网络的现成集成。话虽如此，由于可以通过 Java API 访问 EDN 来发布事件，并使用 JMS 来传递事件，因此从服务总线与 EDN 进行交互并不非常困难。 使用与上一部分中创建的类非常相似的 Java 类的 Java 标注，我们可以从服务总线管道发布 EDN 事件。随后，我们将 使用入站 JMS 传输将服务总线代理服务订阅到特定的 EDN 事件。

##### 使用 Java 标注发布到 EDN

创建 Java 项目 *EDNServiceBusCalloutLibrary*。添加与 *EDNJavaClient*  项目中相同的库和 JAR 文件。还要添加我们之前在第 7 章中用于构造 Java 标注的 XML Beans jar 文件（com.bea.core.xml.xmlbeans\_x.y-z.p.jar）。最后，为名为 *EDNServiceBusCalloutLibraryJAR*  的项目创建一个新的部署配置文件，其配置文件类型为 JAR 文件。

使用静态方法 *publishEDNEvent* 创建 Java 类 *EDNEventPublisher*，该方法采用 *XmlObject*  以及 EDN 事件的有效负载以及 EDN 事件的命名空间和 [local] 事件名称的字符串。（最后一个输入参数是根元素的名称 - 要工作 围绕使用 xml-fragment 作为根元素的 SB 特性。

实现此方法与前面讨论的 Java 类 FlightStatusUpdateEventPublisher 中的 main 方法非常相似*，只是使代码泛型：不绑定到* FlightStatusUpdateEvent*。*

 public static void publishEDNEvent（XmlObject eventPayload， String ednEventNamespace， String ednEventName，

String rootElementName） { String xml = eventPayload.toString（）.replaceAll（“xml-fragment”，

rootElementName）;

属性道具;像以前一样定义属性 try {

元素 eventBody = XMLUtil.parseDocumentFromXMLString（eventBodyXML. toString（））.getDocumentElement（）;

BusinessEventConnectionFactorySupport.setJndiProperties（props）;EdnConnection ec =

（Edn连接）BusinessEventConnectionFactorySupport.findRelevantBusi nessEventConnectionFactory（true）.createBusinessEventConnection（）;

BusinessEventBuilder 生成器 = BusinessEventBuilder.newInstance（）;builder.setEventName（new QName（ednEventNamespace， ednEventName））;builder.setBody（eventBody）;

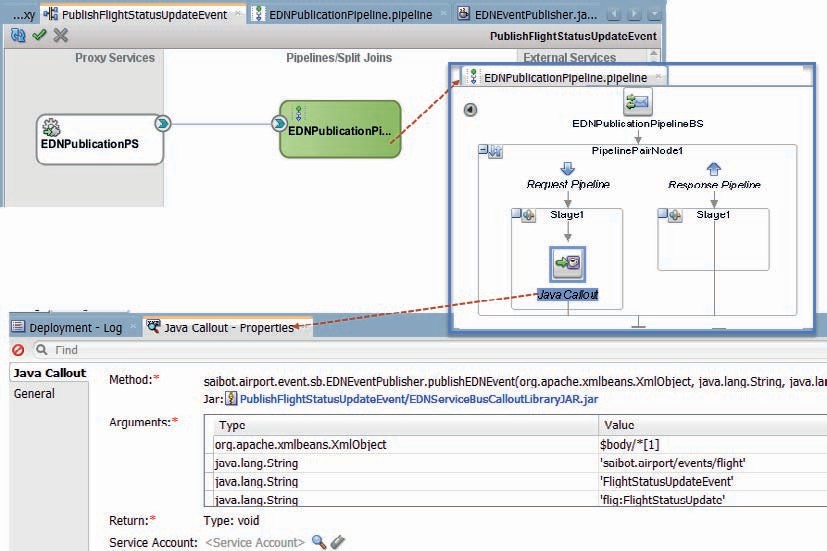
ec.publishEvent（builder.createEvent（）， 4）;ec.close（）;

} catch （异常 e） {

实施异常处理...

}

}



**图 15-15。** *配置服务总线项目以将事件发布到 EDN*

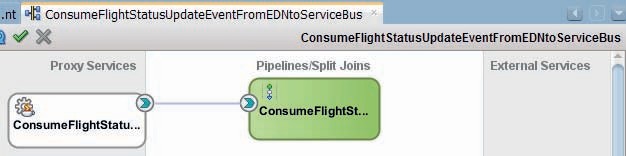
根据 *EDNServiceBusCalloutLibraryJAR*  部署配置文件部署项目，生成 JAR 文件。

创建一个名为 PublishFlightStatusUpdateEvent *的新服务总线项目*。创建服务类型为“任何 XML”的 ProxyService *EDNPublicationPS*。为此代理服务生成名为 *EDNPublicationPipeline*  的管道。在管道中添加管道对。在请求分支中添加 Java 标注。在 JAR 文件 EDNServiceBusCalloutLibraryJAR.jar 的 EDNEventPublisher 类*中配置标注以调用方法* publishEDNEvent 。第一个参数是用 $body 变量中的第一个元素设置的，其他三个参数分别设置为 saibot.airport/events/flight、FlightStatusUpdateEvent 和 flig：FlightStatusUpdate，如图 15-15 所示。

运行管道或代理服务。将输入设置为作为 *FlightStatusUpdateEvent* 有效负载的 XML 文档。例如，应发布该事件，并由 FlightStatusRelay *复合*的新实例使用该事件。

##### 使用入站 JMS 传输在服务总线中使用 EDN 事件

可以使用简单的入站 JMS 传输在服务总线项目中使用 EDN 事件。服务总线代理服务订阅为 *EDNTopic 上的持久侦听器*，并且



**图 15-16。** *使用入站 JMS 使用 EDN 事件的服务总线项目*

*运输*

使用消息过滤器，我们确保（仅）特定 EDN 事件类型的所有实例都 传递到代理服务中。

创建服务总线项目 *ConsumeFlightStatusUpdateEventFromEDNtoServiceBus*。在“代理服务”泳道中单击鼠标右键，然后选择“插入传输”|”JMS 传输。将代理服务的名称设置为 *ConsumeFlightStatusUpdateEventFromEDN*。将服务公开为名为 *ConsumeFlightStatusUpdateEventFromEDNPipeline* 的管道。在代理服务向导的第 2 步中，将服务类型设置为“任何 XML”。在第三步中，将端点 URI 设置为 jms://<host\_of\_ SOA\_managed\_server>：<port\_of\_SOA\_managed\_server>/jms.fabric.xaEDNConnectionFactory/ jms.fabric.EDNTopic（例如，jms://localhost:7101/jms.fabric.xaEDNConnectionFactory/jms.fabric. EDNTopic）。

完成向导。该项目现在如图 15-16 所示。

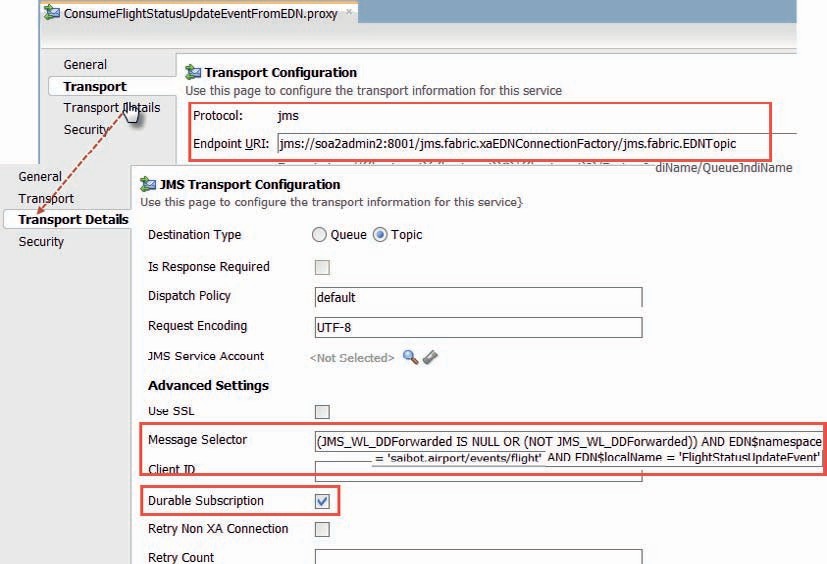
使用当前配置时，此服务总线项目将使用在 EDN 上发布的所有事件，无论类型如何。我们需要在代理服务的“传输详细信息”页面上设置消息选择器。我们可以通过检查 FlightStatusRelay *复合*中 *Mediator* FlightStatusUpdateEventHandler 的持久订阅来找到所需的表达式：

 （JMS\_WL\_DDForwarded为 NULL 或（不是 JMS\_WL\_DDForwarded））AND EDN$namespace = 'saibot.airport/events/flight' 和 EDN$localName = 'FlightStatusUpdateEvent'

同时将“目标类型”设置为“主题”，并选中“持久订阅”框，如图 15-17 所示。

此时，事件已使用，但未对它执行任何操作。为了至少证明 FlightStatusUpdateEvent 类型的 EDN 事件*确实触发了 ConsumeFlightStatusUpdateEventFromEDN 代理服务，我们可以在* ConsumeFlightStatusUpdateEventFromEDNPipeline 的管道对中添加一个 Report 活动， 该活动发布包含整个事件有效负载的报告（加载到管道内部的 $body 变量中）。

部署项目。然后，使用*在前面部分*中创建*的 EM FMW 控件或 Java 客户端或服务总线项目* PublishFlightStatusUpdateEvent 发布 FlightStatusUpdateEvent。



**图 15-17。** *配置入站 JMS 传输，以使用以下类型的 EDN 事件*

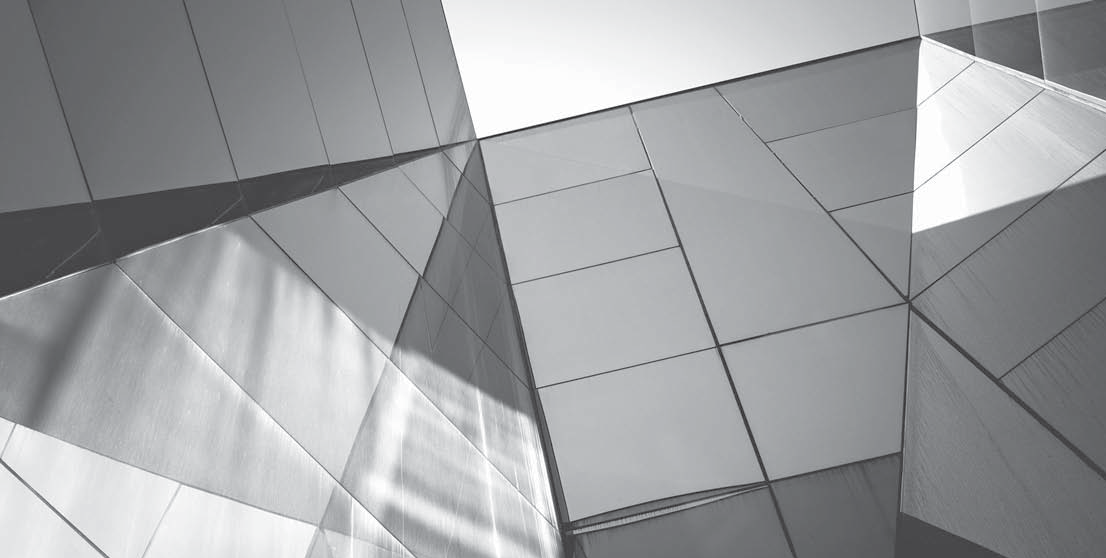
*FlightStatusUpdate事件*

### 总结

事件为面向服务的环境增加了一个重要元素：它们提供了一种以真正解耦的方式进行交互的方法。消费者和出版商彼此不了解，但他们对彼此都非常重要。事件传送网络支持从 SOA 组合到 Mediator 和 BPEL 组件以及其他各方（包括服务总线和远程 Java 客户端）的事件交换。本章演示了使用 EDN 发布、订阅和使用事件的一些关键方面。

然而，这只是触及了表面。将事件作为 SOA 不可或缺的一部分会带来许多需要仔细研究的问题、挑战和机遇。请注意，事件 和 EDN 将出现在一些即将到来的章节中，包括下一章（关于持续事件处理）和第 19 章（BPM）。





章

# 16

## 使用 Oracle Event Processor 实现快速数据处理

在上一章讨论的业务事件层之外，存在一个由 细粒度信号、度量和事实组成的世界。这是一个快速数据的世界， 其中带有时间戳的数据语句流在 IT 系统中不断流动。住

**U**

数据由 IoT（物联网）——各种连接的传感器、摄像头、可穿戴设备和其他设备，以及 Twitter 和 Facebook 等社交媒体以及移动应用程序、Web 应用程序、管理软件系统和业务流程引擎上的传感器产生 。

通过处理这些流，我们可能能够了解重要的业务信息。

聚合、过滤、缺失事件检测和模式匹配是我们可以应用的一些操作，用于将这种雪崩式信号的结果提升到业务级别。事件处理通常会生成业务事件，这些事件可提供见解并触发操作。对数据事实流的解释通常必须（近乎）实时地完成，这增加了 挑战，需要使用专门的技术。

事件处理的典型配套工具是业务活动监视 （BAM）。通过 BAM（监控行为和产品（将在第 20 章中讨论），通过实时仪表板和近乎实时的报告提供运营洞察力。 事件处理产生的结果通常通过 BAM 仪表板进行报告。

Oracle 事件处理器 （OEP） 是集成在 SOA Suite 中的一个组件，它提供了 处理快速数据和从连续的细粒度数据事件流中生成业务事件的能力 。OEP 可以使用来自各种渠道（包括 JMS、EDN、HTTP/REST、RMI）的快速数据，并支持创建自定义适配器，以便 从 Twitter、消息传递系统（如 Tibco Rendezvous、Sockets、财务数据源等）收集快速数据。OEP 和 SOA 组合与服务总线之间的集成取决于

JMS 和 EDN。

OEP 不仅用于处理大量事件。它还将是检测跨进程实例和服务调用的预定义模式的首选工具。OEP 可以记住业务事件相当长一段时间，并快速关联其他（或缺失）事件以生成相关的见解或警报。

Stream Explorer （SX） 是 OEP 之上的工具，面向业务线用户（非技术 IT 使用者）。此工具围绕 Oracle Event Processor 和 Oracle BAM 提供基于浏览器的可视化、声明性包装器。借助 Stream Explorer，您可以轻松地在实时（数据流）上创建探索和仪表板，实时报告模式、 相关性、聚合和偏差。

在本章中，我们将使用 OEP 处理包含停车场事件的模拟源，这些事件报告汽车进出 Saibot 机场的一个停车场。处理这些事件，以便在停车场快要填满时在 JMS 上生成消息。使用事件流，OEP 将检测拖车候选者：已被遗弃或至少在停车场停留超过 48 小时的汽车。OEP 还将生成汽车停留事件*（源自*基本汽车进入*和*汽车离开*事件），然后汇总这些事件以生成*平均汽车停留*持续时间报告。*

最后，塞博特机场的安全部门使用 OEP 来检测机场购物区的一系列可疑信用卡交易，遵循早期盗窃案件中发现的特定模式。这些发现被报告为 EDN 上的业务事件，由 一个或多个 SOA 组合应用程序处理。

### 整理实时数据雪崩

“持续事件”一词可以指许多不同的事物——从相当高级的业务事件，例如与某些供应商签订长期协议或批发商的大订单，到微小的信息碎片，例如冰箱中的当前温度、单击患者自助服务 Web 应用程序中的导航链接、 某些设备的心跳，或者扫描某些安全徽章以进入储藏室的事实。与最低级别的数据粒度相比，较高级别的事件通常具有更低的频率（以每小时或每天的发生次数计数），最多每秒数千次。事件频谱的这两端的信息内容往往从高级别、低频事件的丰富到高频数据包的相当稀疏。

处理这些事件的一个重要方面是，其中大部分必须实时完成。持续的业务会导致连续的事件流，可能需要立即采取行动。我们不能只是将所有数据转储到我们稍后将要分析的数据仓库中，即使对于某些（派生的）数据，当然，这正是我们将要做的事情。事件可能促使我们做出的一些反应的紧迫性是一个原因。数据量之大是另一回事：每秒发生的数千个低级事件代表了数据仓库无法轻松处理的大量数据，也不应该处理这些数据，因为这些事件中的绝大多数代表了完全无用的信息，尤其是在单独考虑时。

因此，我们需要一种方法来处理业务运营不断产生的数以百万计的事件。将这些事件转换为信息的处理器通常通过生成具有某些特殊业务意义的更高级别的事件类型，携带更大的信息有效负载，并且以低得多的频率生成。如图 16-1 所示。通常，处理器生成的事件在协议和通道以及信息结构方面都以更通用、更规范的方式传播。

在事件处理的第一阶段之后，下游消费者吸收更高级别的事件。这些使用者可以执行多项操作，包括发送警报或通知、更新实时报表或仪表板、执行调用 Web 服务或启动人工任务等操作，或者将数据加载到数据仓库中。这些下游使用者还可以进一步处理事件，将其处理为更高级别、更有意义的事件。然后，这些事件将 通过事件处理的下一次迭代进行传播。

Events

* Information
* Conclusion
* Alert
* Recommendation
* Action

Smart Processing:

* Filter & Discard
* Enrich
* Aggregate
* Pattern detection
* Promote/Propagate

**图 16-1。** *事件处理 — 增加信息内容并降低频率*

#### 复杂事件处理

在大量连续到达的、基本上是单独无意义的事件的大海捞针中找到（缺失的）针是复杂事件处理 （CEP） 的目的，也称为智能事件处理 （IEP）。CEP 是一个在 1990 年代初期出现的领域，首先出现在学术界，例如斯坦福大学和加州理工学院，后来出现在 IT 行业。许多大型供应商都活跃在这一领域（其中一些是在最近收购之后），包括SAP，Microsoft，Software AG，IBM，Tibco，JBoss和EsperTech。这些供应商提供各种软件产品，这些产品通过实时计算提供“运营智能”，这些计算涉及与其各自的事件流智能或复杂处理器进行模式匹配 。

事件处理器使用来自异构源的事件，例如 JMS 队列、JMX MBean、传入的 HTTP 请求和 WebSocket 以及本机 API 调用、消息驱动 Bean、数据网格、Hadoop/NoSQL 和关系数据库。处理的结果主要包括另一个高级事件生成，这些事件通常以规范数据格式通过更同质的通道发送。请注意，事件处理器与它们所处理的事件的生产者和它们生成的事件的使用者是分离的。

处理器对传入事件执行以下操作，从而生成 它们发布的事件：

* *过滤器* 大海捞针，那些表明异常的罕见事件。
* *丰富* 为事件添加上下文，从而（业务）意义。
* *聚合*计数事件或确定事件在其有效负载中携带的指标的总和、上限或下限或平均值。

###### 注意

*聚合通常是在包含特定时间范围内的所有事件或指定数量的最新事件的窗口上计算的 。*

* *模式检测* 通过比较一段时间内的相关事件来发现趋势， 找到已知预示着某个特定值或值变化的特殊序列

事件，以及发现本应发生的事件的缺失，都是在一串相关事件中检测有意义模式的示例。

* *升级*或传播 上述操作的结果可以是按原样传播事件，也可以是以扩充或提升状态传播事件，也可以是发布全新的事件

带有聚合值、某些指示器或警报。

###### 需要明确的是

*复杂事件处理中的“复杂”一词不是指事件（它们通常非常简单），而是指发生的处理。特别是模式的检测可以非常先进，考虑到模式的复杂性以及连续实时处理大量事件的挑战。*

**事件处理语言**

需要对事件处理器进行编程。它们就像查询引擎一样，需要查询事件流，而不是静态的记录集，而是不断变化的数据颗粒集合。事件查询不会只执行一次。相反，它们被附加到事件流并持续执行。每个新到达的事件都可能导致事件查询产生新的结果。结果不断以事件的形式发布。

随着时间的流逝，在事件处理语言 （EPL） 的通用标签下对事件处理器进行编程的特殊语言已经发展起来。然而，业界尚未出现单一标准的统一EPL。几种完全不同的 EPL 用于各种 CEP 产品。某些 EPL 源自业务规则语言或受其启发。其他 EPL 的灵感来自 SQL，源自 SQL，甚至与 SQL 集成。在对事件处理查询进行编程时，可以利用 SQL 的广泛知识。事实证明，能够在单个查询中组合事件流和关系数据源也非常方便（例如，将历史数据和参考数据与实时事件源联接）。SQL 和 EPL 的这种结合导致了 CQL，即连续查询语言，它起源于斯坦福大学。CQL 是 OEP 应用程序中用于处理和生成事件的核心技术。

###### 提示

*Oracle Database 12c 中引入的用于对关系数据集执行模式匹配的 Match\_Recognize 运算符也利用 CQL 语法来构造模式查询。*

CQL 查询从事件通道中进行选择，通常应用某些范围或时间窗口， 使用 where 子句筛选查询返回的事件。select 子句引用 事件属性，并使用函数和运算符来派生结果。请注意，可以在单个 CQL 查询中将多个通道和其他数据源联接在一起（是的：甚至是外部联接）。

下面是 CQL 查询的简单示例，每当*有效负载值大于 15 的* someEventStream 中的输入事件数发生变化时，就会生成输出事件，并考虑过去 5 秒的时间窗口：

 **选择 count**（payloadValue） **作为** countEvents

**从** someEventStream [范围 5]

**其中** payloadValue > 15

此查询生成的结果数无法预测：这取决于事件在事件流中出现的频率和不规则性。例如，如果每秒只发布一个事件到流中，则 5 秒窗口中的事件数不会更改，并且在运行查询的前 5 秒后，查询不会生成其他事件 。

##### 下游事件使用者

事件处理器只为我们提供了从数据到见解和行动的部分路径。它只 在 JMS 等通道上的事件中说话，而不是直接与人甚至 Web 服务交谈，并且它没有 自己的用户界面，除了用作管理和监视控制台的基于浏览器的可视化工具。OEP 发布的事件（无论它们变得多么先进和丰富）都需要使用并转换为其他内容：报表或仪表板、 警报或操作。

OEP 应用程序通常以 JMS 队列上相当通用的消息、发送到 EDN 或 HTTP 通道的消息、数据网格上的数据包或记录到 NoSQL 数据库、或对 Web 服务接口或其他一些标准设施的请求的形式发布其结果。因此，许多不同的产品可以在 OEP 的下游使用，以吸收事件并处理它们：加载到数据仓库中、用于刷新实时仪表板、更新数据库记录、通知 Web 服务、发送电子邮件或文本消息、创建人工任务或在网站或 RSS 源上发布。请注意，具有多个链式 OEP 实例的情况并不少见，每个实例处理的较高业务级别事件的数量都在减少。

在本章中，我们将使用最常见的渠道：JMS 和 SOA Suite Event Delivery Network （EDN） 来发布 OEP 结果。

#### Oracle 事件处理器和 SOA 套件

Oracle 事件处理器是 SOA Suite 许可证的一部分。但是，它不是 SOA Suite SCA 容器的一部分，它甚至不像 SOA Suite 那样运行在同一个 WebLogic Server 中。它 运行在自己的精简、轻量级服务器上，该服务器基于 POJO，基于 Spring DM 和基于 OSGi 的框架来管理服务。该服务器附带了 Jetty，一个用于运行 servlet 的 HTTP 容器，并支持 JMS 和 JDBC。它具有缓存和集群 功能，可能由 Oracle Coherence 网格提供支持。OEP 可以处理数千个 并发查询，每秒处理数十万个事件。平均消息 延迟可以低于 1 毫秒。

消息通常从 JMS（例如，SOA Suite WebLogic Server 中的队列）或 EDN（由 JMS 支持）读取。但是，表也可以用作事件源，并且通过自定义适配器，我们几乎可以使用来自任何源的消息，包括 文件、套接字、NoSQL 数据库和 JMX。

OEP 服务器提供基于 Bayeux 协议的 HTTP Pub/Sub 事件通道;这允许 OEP 应用程序使用从符合 Bayeux 的服务器推送的消息，并允许客户端订阅 OEP 服务器发布的消息。客户端还可以发布到 HTTP 通道，而无需显式连接到 HTTP Pub/Sub 服务器。

事件消息通常由 CQL 处理器处理，CQL 处理器是事件处理网络 （EPN） 中的节点，用于接收来自流源的消息并对其执行操作以生成新的事件消息。CQL 查询的结果通常是事件流。

OEP 提供了其他功能，例如记录流经 事件处理网络的事件并存储它们的能力。以后可以从事件存储库中回放它们 ，这对于测试非常有用。

OEP 可视化工具是一个 Web 应用程序，它是管理控制台，可用于查看、开发、配置和监视 Oracle 事件处理应用程序的各个方面。可以从可视化工具中部署、暂停和恢复 OEP 应用程序，以及卸载 OEP 应用程序。可以监控当前操作，包括所有控制台输出。它还支持视觉和

运行时 CQL 查询编辑。可以在可视化工具中监视在 HTTP Pub/Sub 通道上发布的事件，并由服务器异步推送到浏览器。

### 使用事件处理器管理停车场

塞博特机场从机场周围的停车场赚了很多钱。因此，对停车过程进行密切监控和仔细管理非常重要，因为它是 旅客在机场调查中评估的一个重要方面。在适当的时候表明

例如，停车场已满，可以防止人们不必要地绕圈子寻找空闲位置。然而，过早的警告意味着人们不必要地被迫将车停在 更偏远的停车场，这并不一定会改善他们的体验。

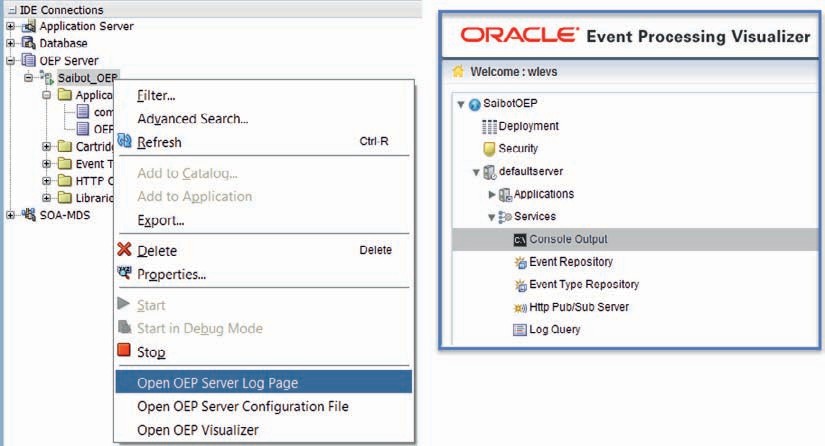
在本节中，我们将了解如何创建 OEP 应用程序来处理汽车进入或离开停车场的所有事件，从而得出当前停车场内的汽车数量。当停车场宣布已满时，应用程序将发布一个特殊事件。 当空间再次可用时，它将发布类似的事件。

#### 开发环境的准备

OEP 12c 应用程序是在 JDeveloper 中开发的。这些应用程序部署到 OEP 服务器实例并在其上运行。此服务器不是集成 WebLogic Server 的一部分，并且独立于基于 WebLogic 的 SOA Suite 12*c* 运行时环境。OEP 运行时 是需要单独安装的 OSGi 服务器。

根据本书网站上的在线说明安装 OEP 12c，并按照所述创建 *SaibotOEP* 域。在 JDeveloper 中创建与本地 OEP 服务器的连接。使用此连接，您可以从上下文菜单中启动和停止服务器，查看其日志记录输出，以及部署、挂起、激活和取消部署 OEP 应用程序。

当 OEP 服务器运行时，您可以使用鼠标右键菜单选项打开 OEP 服务器日志页，在 JDeveloper 中打开日志窗口。或者，您可以使用在浏览器中运行的 OEP 可视化工具：host：port/wlevs/。此工具包括控制台输出窗口 - 在“域”|“defaultServer”|”服务 |控制台输出，如图16-2所示。



**图 16-2。** *左侧显示 JDeveloper 中的 OEP 连接（以及可用操作）*

*以及右侧的 Event Processing Visualizer 浏览器应用程序*

#### Saibot 停车 OEP 应用程序

在 JDeveloper 中，基于模板 OEP 应用程序创建一个新应用程序。将应用程序命名为 *SaibotParking*。在应用程序内部创建一个 OEP 项目*CarParking\_step1* - 基于空 OEP 项目模板。该应用程序是使用一个名为 CarParking\_step1.context.xml 的空 Spring 上下文文件创建的，该文件将包含 bean、事件、通道和处理器的定义，以及一个名为 processor.xml用于存储 CQL 查询的文件。EPN 图可视化上下文文件中定义的事件处理网络。MANIFEST.MF 文件最终包含应用程序本身及其所有 bundle 依赖项的 OSGi 定义。

##### 定义事件 CarMovementEvent

OEP 中的事件可以完全以声明方式定义，也可以基于 Java Bean（POJO）定义。我们将采用后一种方法。创建一个 Java 类 *CarMovementEvent* ，它是具有以下属性的 POJO ：

 private 字符串 carparkIdentifier;

private Integer entryOrExit;1 表示到达，-1 表示退出私人 String 车牌;

私人长到货时间;

为这些属性生成 getter 和 setter 方法。

打开 EPN 图。单击“事件类型”选项卡。向 CarParking\_step1.context.xml 添加新的事件类型。新事件称为 *CarMovementEvent* ，它基于 具有该名称的 Java bean。

创建第二个 Java 类，名为 *CarParkEvent* ，具有以下属性：

 private 字符串 carparkIdentifier;

private 整数 carCount;私人停车场现时车辆数目 浮动百分比Full;目前占用的私人容量百分比 Long， averageStayDuration;以纳秒为单位，OEP 时间单位

同样，生成 getter 和 setter 方法。并且，像以前一样，创建一个 OEP 事件类型，称为

*CarParkEvent* - 基于此 Java bean。

##### 创建汽车运动模拟器

OEP 使用适配器将事件馈送到 EPN 中。适配器可以从外部源（如 JMS、HTTP、RMI 和 EDN）收集事件。在开发和测试过程中，创建生成事件的适配器（而不是从外部源检索事件）可能非常方便。任何 Java 类都可以充当这样的事件生成器，只要它实现了 *StreamSource* 接口。

创建一个名为 *CarMovementController*  的 Java 类来实现此接口，以及 *Runnable* 接口以允许实例计划对自身的调用。*CarMovementEvents*  生成器的非常简单的实现如下所示：

 public class CarMovementController implements RunnableBean， StreamSource { private static final int SLEEP\_MILLIS = 100; // 事件之间的时间 private boolean suspended;

私有 StreamSender eventSender;

私有静态 Random rand = new Random（）;

公共静态 int randInt（int min， int max） {

int randomNum = rand.nextInt（（最大值 - 最小值） + 1） + 最小值;返回 randomNum;

}

私有字符串 randChar（） {

返回 new Character（（char） randInt（65， 90））.toString（）;

}

public void run（） { suspended = false;

while （！isSuspended（）） { // 永久生成消息...生成CarMovementEvent（）;

try { synchronized （this） {

wait（SLEEP\_MILLIS \* randInt（1， 5） / 2）;

}

} catch （InterruptedException e） { e.printStackTrace（）;

}

}

}

private void generateCarMovementEvent（） { CarMovementEvent 事件 = new CarMovementEvent（）; event.setEntryOrExit（1）; event.setCarparkIdentifier（“1”）;

event.setLicencePlate（randChar（） + randChar（） + “-” + randInt（1， 999） + “-” + randChar（） + randChar（））;

eventSender.sendInsertEvent（事件）;

}

public void setEventSender（StreamSender sender） { eventSender = 发送者;

}

public synchronized void suspend（） { suspended = true;

}

私有同步布尔值 isSuspended（） { return suspended;

}

}

此类以某种随机速率生成具有随机生成车牌的汽车到达事件。稍后，该类的更高级化身也将派生出车事件， 并将生成涵盖五个不同停车场的事件。

###### 注意

*花费大量精力创建事件生成器是值得的。 它是 OEP 应用程序开发和测试的关键元素，理想情况下，它应该涵盖您希望能够处理的所有用例。*

##### 创建事件报告 Bean

OEP 应用程序的输出可以通过出站适配器报告给外部接收器，例如 JMS 目标、RMI 客户机、套接字端点或事件交付网络。在开发和测试过程中，使用内部事件接收器通常很方便。实现 *StreamSink*  的简单 Java 类可用于接收来自事件处理器的结果，例如，将它们写入控制台。

创建 Java 类 *CarParkReporter*，该类实现该接口，并根据它收到的事件编写简单的日志记录行：

 public 类 CarParkReporter 实现 StreamSink {

public void onInsertEvent（Object event） { if （事件实例的 CarParkEvent） {

CarParkEvent cpe = （CarParkEvent）事件;System.out.println（“目前停在停车场的汽车数量”

+cpe.getCarparkIdentifier（）+“ ： ”+cpe.getCarCount（）

+“ 百分比满 ”+（cpe.getPercentageFull（））+“%”

+“ 汽车的平均停留时间为”

+（0.1\*Math.round（1E-08\*cpe.getAverageStayDuration（）））

+“秒。

);

}

if （事件实例CarMovementEvent） { CarMovementEvent cpe = （CarMovementEvent）event;

System.out.println（“带车牌的汽车”+cpe.getLicencePlate（）

+“ 只是 ”+ （ cpe.getEntryOrExit（）==1？”到达“：”left from“）

+“ 停车场 ”+cpe.getCarparkIdentifier（）

);

}

}

}

##### 创建事件处理网络

打开 EPN 图。将 Adapter 组件拖到关系图中。调用适配器 *carMovementEventGenerator* 并选择 *CarMovementController* 作为 Java 类来实现 适配器。

将“通道”组件拖到关系图中。将通道名称设置为 *carMovementInputChannel*，并选择 *CarMovementEvent* 作为流经此通道的事件类型。将适配器连接到通道的左侧 - 指定适配器的输出是通道的输入。

将 Processor 组件添加到关系图中。将其名称设置为 *carMovementProcessor*，并接受配置位置的默认 processor.xml 文件。将 *carMovementInputChannel* 连接到处理器的输入（左侧）。

添加第二个 Channel 组件，称为 *carParEventChannel*。将其事件类型设置为 *CarParkEvent*。

将 *carMovementProcessor*  的输出连接到此通道的输入端。将 Bean 组件添加到关系图中。将其命名为 *carParReporter* 并选择 Java 类

*CarParkReporter* 作为其实现类。

将 carParEventChannel *的输出连接到*此 bean。此外，将 *carMovementInputChannel*  的输出连接到 *carParkReporter*。

该图现在应如图 16-3 所示。这是 CarParking\_step1.context.xml 中定义的配置的可视化。

应用程序中仍然缺少的元素是实际的 CQL 查询，该查询处理并在这种情况下聚合 *carMovement* 事件。打开文件处理器 .xml。它包含 *carMovementProcessor* 元素，其中包含 ID *为 ExampleQuery* 的查询。将 query 元素替换为以下代码片段：

 <query id=“CarparkSummary”><！[CDATA[

IStream（选择 sum（car.entryOrExit） 作为 carCount， car.carparkIdentifier 从 carMovementInputChannel 作为 car

group by carparkIdentifier） ]]>

</查询>

##### 部署和测试

可以部署应用程序。部署后， *CarMovementController* 将立即开始生成事件，处理器也将立即启动操作。

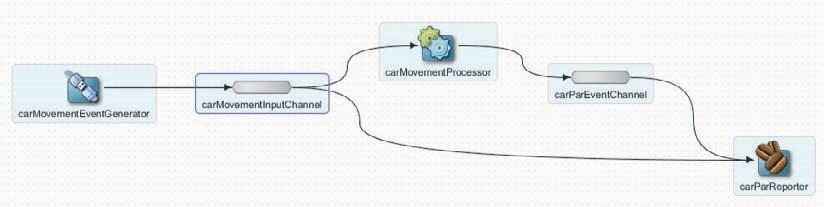
右键单击应用程序导航器中的项目。从上下文菜单中，选择“部署”|”oep\_profile−1。此时将显示部署向导。接受默认部署操作 - *将 OSGi 捆绑包部署到目标平台*。按“下一步”。在第二页上，检查*Saibot\_OEP*服务器是否为目标服务器。如果是，并且服务器正在运行，请按“完成”执行部署。

检查日志窗口 - 在 JDeveloper 或 Visualizer 控制台中。您会注意到源源不断的消息流，一条宣布有一辆车到达，另一条则提供一号停车场当前汽车数量的最新信息。在这种情况下，每当之前的结果被新结果取代时，CQL 就会发布结果，也就是说，每辆进入停车场的新车都会发布结果 。

对于每辆进入的汽车来说，并不是真正需要对汽车数量进行这样的更新。我们可能希望每 5 秒更新一次摘要。这可以通过将 from 子句重写为：

 从 carMovementInputChannel [范围 1 DAY slide 5 秒] 作为汽车

在这里，我们指定了查询的结果将在 1 天的（输入）事件中计算，并且应每 5 秒生成一次。进行此更改并重新部署应用程序。摘要消息将不如以前丰富。



**图 16-3。** *用于处理 carMovementEvents 并将所有输出发送到*

*carParkReporter bean*

### 更多事件智能

OEP 可以做的不仅仅是计数和聚合。在本节中，我们将了解如何关联事件、检测“缺失”事件并从中获取商业智能。

在应用程序内部或跨应用程序链接 OEP 处理器通常是为了 从不同粒度级别的某个基本数据信号集得出不同的结论;这可以跨应用程序或内部完成。我们将在应用程序中使用多个处理器。

在本节中，我们确定离开的汽车在停车场停留了多长时间;我们可以用它来自动计算停车费。接下来，我们用汽车的平均停留时间来扩展我们的摘要。我们寻找已经过时的汽车：我们的停车场仅限短期停车，最多可停车 36 小时。在停车场停留超过 48 小时的汽车将成为拖车的候选者。

恳请读者下载本书的资源，尤其是第 16 章源文件夹下CarParking\_step2和\_step3的项目，而不是逐一完成每个详细的小步骤并用大量长代码片段填充本章。这些 项目包含所有来源，其要点将在下面讨论。

CarMovementController*（事件生成器）扮演着重要角色*。此类已扩展至：

* 为离开停车场的汽车生成事件（当生成汽车到达的初始事件时，随机派生的离开时间将记录在 *TreeSet 中*

按出发时间排序;每当汽车到达停车场时，*都会检查* TreeSet 以查看列表顶部是否应该离开，在这种情况下，将生成汽车退出事件。

* 为一组（目前有五个）不同的停车场生成事件，每个停车场都有特定的容量;生成器尊重容量，确保不会有更多的汽车被允许进入

停车场超过容量允许的容量。

It is common to want to enrich events at some point during the processing. For example, because events have to be placed in context, such as which car park did the car arrive at, what is that car park’s capacity or its location, and what name should be passed in the business event published to JMS or EDN.

Having to call out to, for example, a relational database each time an event has to be enriched can be quite expensive and is quite unnecessary for (fairly) static data. OEP supports the use of caches to store such data and allows joining to caches in CQL queries. OEP has its own internal (local) cache implementation system. Alternatively it supports use of Coherence (local and distributed) as well as third party caching solutions.

In the current example of the car parks, a cache is defined in the context file, using the internal caching mechanism. A Java class—*CarParkCacheLoader*—is created to provide the static data for the cache. A bean based on this class is configured as the cache-loader for the cache. The CQL query in the *carParkEvent* processor uses the cache to enrich the

*CarParkEvent* with the name of the car park. In the next section, the car park’s capacity is also retrieved from the cache and used to determine to what extent the car park is filled with cars.

In order to make use of caches, the MANIFEST.MF file has to be edited. The bundle com.bea.cache.jcache has to be added to the list of dependencies.

#### 汽车停留时间 - 按车计算，按 停车场升级

OEP 擅长的一件事是关联事件。它可以在不同时间点到达的事件中找到模式。在此示例中，我们将让 OEP 关联同一辆车的到达和退出事件 。这些事件共同构成了我们可以计算停车费的“汽车停留 ”。通过考虑所有这些停车事件，我们可以让 OEP 计算 每个停车场的平均停车时间。图 16-4 显示了此方法的 EPN。

##### CQL 编写 CarStayDone 事件

我们用来将事件关联到有意义的模式的关键 CQL 运算符称为 MATCH\_ RECOGNIZE。carStayDoneProcessor 中使用的查询使用以下子句：

 ...从 carInputChannel match\_recognize （

按停车场分区标识符，车牌测量 A.车牌作为车牌

, A.carparkIdentifier 作为 carparkIdentifier

, A.arrivalTime 作为 arrivalTime

, B.element\_time - A.element\_time作为 stayduration

**模式 （A， B） 定义**

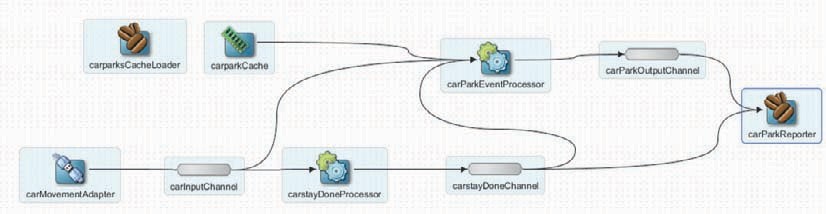
**A 作为 entryOrExit= 1**

**，B AS entryOrExit = -1**

） 如保留

MATCH\_RECOGNIZE表达式中的关键元素是应 在事件流中检测到的模式。在本例中，该模式由事件 A 和 B 组成，这两个事件都适用于分区 依据子句定义的同一停车场和汽车（通过车牌）。此外，对于 这辆车，A 是到达事件（*entryOrExit* 等于 1），B 是后续退出事件。

每当找到这两个事件时（中间没有其他事件，且 entryOrExit  *属性的值*不同），查询将生成 *CarStayDone* 类型的事件。*stayduration* 度量值是使用伪列*element\_time*计算的。每个事件都由 OEP 加盖时间戳，使两个事件的*element\_time*值之差等于两个事件之间的时间间隔（以纳秒为单位）。



**图 16-4。** *停车场事件的事件处理图 - 派生汽车停留*

*通过基于缓存的扩充，提供完整的事件和平均汽车停留时间洞察*

##### CQL 计算平均汽车停留时间

使用*发布到* carstayDoneChannel 的 *CarStayDone*  事件，可以非常简单地派生*报告每个停车场平均停留时间的* CarParkEvents，并在汽车离开停车场时发布更新。下面的 CQL 代码段处理了这个问题。请注意，在这种情况下，平均持续时间是根据最近 100 辆离开停车场的汽车计算的。

 <query id=“CarparkAvgStayQuery”><！[CDATA[ IStream（选择 -1 作为 carCount）

, carparkIdentifier 作为 carparkIdentifier

, '' 作为描述

, avg（carstayDuration as averageStayDuration from carstayDoneChannel [rows 100]

group by carparkIdentifier

)

]]></查询>

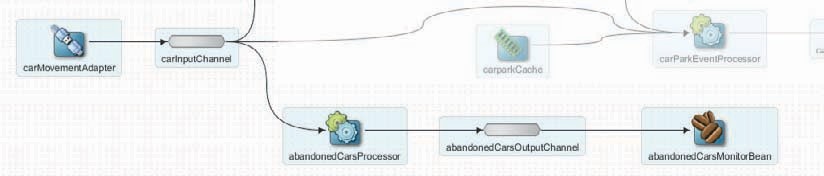
当您检查 processor.xml 文件的内容时，您会注意到对于 *carParkEventProcessor*，<rules> 元素中有两个查询元素。您可以在一个处理器中定义多个查询，并且每个查询都会持续执行。显然，它们都需要提供相同的事件类型才能馈送到输出通道中。请注意视图 *CarparkStatusView 如何包含* CarparkStatus *视图和* carparkCache *之间的联接*，后者用于使用从缓存中检索到的停车场名称*来扩充* CarParkEvent。

#### 寻找拖车候选人

如前所述，通常情况下，我们应该检测预期事件的缺失，以及导致发布新业务事件的原因。在我们的机场停车案例中，缺少的事件是没有及时离开我们短期停车场的汽车。

事件生成器已经扩展，也可以提供不会很快离开的汽车。我们的 OEP 应用程序应该能够检测到此类汽车，这些汽车不会在 120 秒内离开（在我们的示例应用程序的时间尺度上）。有关此步骤，请参阅项目CarParking\_step3中的源代码 。

定义了一个新事件 — *AbandonedCarEvent*，该事件基于具有该名称的 POJO。创建了实现 *StreamSink 接口的 Java* 类 *AbandonedCarsMonitor* 来接收这些事件并为它们编写日志消息。新增 *abandonedCarsProcessor*，用于处理 carInputChannel *的事件*，检测废弃的汽车，并向 *abandonedCarsOutputChannel* 报告，如图 16-5 所示。



**图 16-5。** *用于检测和报告废弃汽车的 EPN 流程 - 使用失踪的汽车*

*事件检测模式*

下一个 CQL 代码段是处理器的核心。

 从 carInputChannel match\_recognize （

按停车场分区标识符，车牌测量 A.车牌作为车牌

, A.carparkIdentifier 作为 carparkIdentifier

, A.arrivalTime 作为 arrivalTime

**包括计时器事件**

**pattern （A NotExit\*） 持续时间 120 秒定义**

**A 作为 entryOrExit= 1**

**， NotExit AS entryOrExit！= -1**

） 被遗弃

同样，我们检查 *carInputChannel* 上的 *CarMovementEvents*。我们正在寻找模式 A——某辆车的进入，然后是 *NotExit*  事件——该事件被定义为 *entryOrExit* 值不等于 −1 的事件。进入停车场的汽车只能在某个时间点再次离开该停车场，并在该点生成 entryOrExit = −1 的事件。如果是这样，它就不会被抛弃。

OEP 允许我们将系统生成的计时器事件与常规事件流混合在一起 — 使用 INCLUDE TIMER EVENTS 指令，并结合子句 DURATION 120 SCONDS。这些告诉 OEP，在事件流的每个分区中（对于我们看到第一个 CarMovementEvent *的每个车牌和停车场组合*），如果在两分钟的时间范围内没有实际事件，则应每 2 分钟添加一次系统事件。随后，当 为某辆车添加这样的计时器事件时——因为 120 秒内没有发生真正的事件，而这应该是

“car exit”事件 - 模式将与该汽车匹配，因为系统生成的计时器事件的 *entryOrExit* 不等于 −1。事实上，计时器事件根本没有属性 值。

### 将 Car Park Almost Full 事件发布到 JMS Destination

作为停车场管理的一部分，Saibot 员工必须对每个停车场的满满情况有很好的了解。当停车场超过90%的满员标记时，这不仅仅是一个侥幸的高峰，而且已经超过85%一段时间了，停车场被宣布为满员。另一个 停车场可能会开放，道路标志也会相应更新。

我们将首先扩展 OEP 应用程序，以使用已计算的当前汽车计数和从缓存中检索到的停车场容量来计算所有停车场的状态 。填充的百分比将添加到已报告汽车计数的摘要中。接下来， 将一条消息发布到 JMS 目标，以提醒任何侦听器有关停车场的情况。

*有关此步骤，*请参阅项目CarParking\_step4中的源代码。

#### 计算每个停车场的满员百分比

*CarParkEvent* 处理器的 CQL 语句与 *carparkCache*  联接，用于计算停车场的满载百分比。此数字被添加到

*CarParkEvent 并在* CarParkReporter  *bean 中报告*。基本 CQL 代码段如下所示：

 <view id=“CarparkStatusView”><！[CDATA[

RStream（ select carCount， car.carparkIdentifier as carparkIdentifier

, R.name 作为描述

**, to\_char（100\*carCount/R.capacity） 作为百分比满**

从 停车场状态[现在] 作为汽车

, carparkCache 作为 R

其中 car.carparkIdentifier = R.identifier

)

]]></查看>

#### 发布 CarParkNearlyFull 事件

定义了一个新事件：*CarParkNearFullEvent*，该事件基于具有该名称的 Java 类。它由扩展的 *CarParkReporter* 报道。添加了一个新处理器（如图 16-6 所示）：*carParkNearlyFullProcessor*，它使用 *CarMovementEvents* 从 *carparkCache* 和 *carInputChannel*  读取并发布到新的 *carParkNearlyFullChannel*。

该处理器必须检查停车场的汽车计数（第一个视图），并将其与 可以从缓存中提取的停车场容量进行比较（第二个视图）。第三个视图 在最后 30 秒内产生最低的汽车计数（相当于现实生活中的 5 分钟）。最后，在视图本身中，当当前停车场满率百分比为 >= 90 且过去 30 秒的最低百分比为 85 或更高时，我们会生成一个停车场几乎已满事件，因此我们不会在第一次越过 90% 边界时开始发布恐慌事件，只是很短的时间。请注意如何在处理器中定义多个视图，这些视图相互引用， 类似于 SQL 语句中的内联视图。

 <处理器>

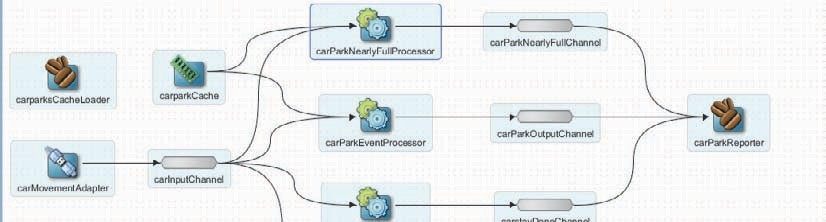
<name>carParkNearlyFullProcessor</name>

<规则>

<view id=“CarparkStatus”><！[CDATA[

IStream（选择 sum（car.entryOrExit） 作为 carCount

, car.carpark标识符



**图 16-6。** *EPN 图，其中包含用于发现几乎满员的停车场的流程*

</查看>

从 carInputChannel 作为 carparkIdentifier） >]] 的汽车组

<view id=“CarparkNearlyFullView”><！[CDATA[ RStream（ select car.carCount

, car.carpark标识符

, to\_char（100\*car.carCount/R.capacity） 作为百分比Full

, R.capacity as carPark容量从 CarparkStatus [now] 作为汽车

, carparkCache 作为 R

哪里 car.carparkIdentifier = R.标识符

)

]]></查看>

<view id=“CarparkStatusView”><！[CDATA[

RStream（ select min（carCount） as recentLowCarCount

, carpark标识符

从 CarparkStatus [范围 30 秒] 作为 carparkIdentifier 的汽车组

)

]]></查看>

<query id=“CarparkNearlyFullQuery”><！[CDATA[ IStream（ 选择 carCount

, c1.carparkIdentifier 作为 carparkIdentifier

, 百分比已满

从 停车场NearlyFullView[现在] 作为 c1

, CarparkStatusView[now] 作为 c2

哪里 c1.carparkIdentifier = c2.carparkIdentifier 和 百分比完整>= 90

和 （100\*c2.recentLowCarCount/c1.carParkCapacity） >= 85

)

]]></查询>

#### 从 OEP 发布到 JMS

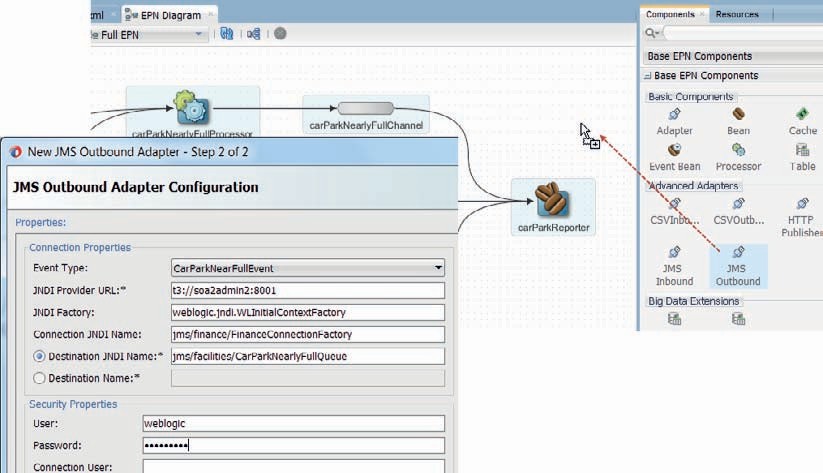
*CarParkNearFull* 事件必须推送到 OEP 应用程序之外，以便全世界（或至少是 Saibot 机场的停车场管理人员）了解它。OEP 可以将事件发布到各种出站通道。在本例中，我们将使用 JMS。我们必须配置 JMS Destination，在 OEP 应用程序中添加一个出站 JMS 适配器并扩展其清单文件，并 创建一个将 OEP 事件转换为 JMS 消息的 bean。

##### 在 WebLogic 上配置 JMS 目标

根据需要添加一个 JMS 队列（或主题），例如，在*运行 SOA Suite 的 WebLogic Server 的 SaibotJMSModule（参见第 6 章）中，以及名为 jms/facilities/*CarParkNearlyFullQueue 的队列。

##### 添加出站 JMS 适配器

打开 EPN 图，并将出站 JMS 适配器拖到该图上。此时将显示 JMS 适配器向导，尽管该向导与 SOA 组合和服务总线服务中使用的向导不同。在 OEP 中



**图 16-7。** *配置出站 JMS 适配器 - 连接到 CarParkNearlyFullQueue*

我们不使用相同的 JCA 适配器。将名称设置为 carpark-nearly-full-jms-outbound-adapter。 按“下一步”。配置适配器如图16-7所示。

将 *carParkNearFullChannel* 连接到新添加的 JMS 适配器。

##### 向项目和 OSGi 清单添加依赖项

必须编辑 OEP 项目属性。添加对 Java EE 库的依赖关系（以涵盖标准 JMS API 依赖关系）。必须在 JAR com.bea.wlevs 上添加依赖项。adapters.jms\_12.1.3.0\_0.jar。此文件位于 OEP 主目录下的 oep\modules 目录中 。

同样，为了成功部署 OEP 应用程序，其 MANIFEST.由于使用了 JMS，需要使用引入的两个新的 bundle 依赖项来扩展 MF 文件。

添加以下两行：

 com.bea.wlevs.adapters.jms.api;version=“12.1.3”， javax.jms，

##### 创建消息转换器

JMS 适配器可以将 OEP 应用程序中的 EPN 生成的事件处理到 JMS MapMessages 中，其条目以事件属性命名。为此，必须在 Spring 上下文文件中*的 JMS 适配器上配置*名称为 eventType *和值*为 CarParkNearFullEvent 的 instance- 属性。

这种自动转换不适用于复杂的事件属性（如 List），并且可能并不总是产生所需的结果。创建起来既简单又常见

并在 JMS 适配器上配置一个消息转换器，该转换器会将事件处理到所需的 *MapMessage 中*。

在这种情况下，请创建实现*接口* OutboundMessageConverter 的 Java 类 *CarParkNearlyFullEventConverter*。在 convert 方法中提供代码 ，该方法将 JMS 会话和您自己的 OEP Event 对象作为输入，并返回 List<Message>。下面显示了一个简单的代码片段：

 public List<Message> convert（Session session， Object inputEvent） 抛出 MessageConverterException， JMSException {

CarParkNearFullEvent 事件 = （CarParkNearFullEvent） inputEvent;MapMessage 消息 = session.createMapMessage（）;message.setString（“carparkIdentifier”， event.getCarparkIdentifier（））;message.setFloat（“percentageFull”， event.getPercentageFull（））;List<Message> messages = new ArrayList<Message>（）;messages.add（消息）;

返回消息;

}

然后，在 Spring 上下文文件中定义一个 bean，并使用以下配置将该 bean 配置为 JMS 适配器的转换器：

 <bean id=“carParkNearlyFullEventConverter” class=“saibot.airport.facilities.parking.jms.CarParkNearlyFullEventConverter”/>

<wlevs：adapter id=“carpark-nearly-full-jms-outbound-adapter” provider=“jms- outbound”>

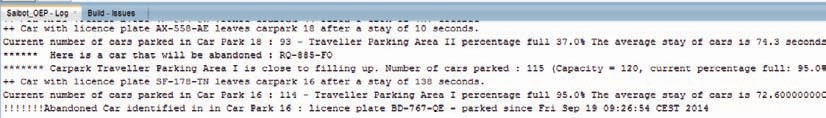
<wlevs：instance-property name=“converterBean” ref=“carParkNearlyFullEventCon verter”/>

</wlevs：适配器>

#### 部署和运行，并监听

应用程序完成后，现在可以将其部署到 OEP 服务器。报告 Bean 的输出将立即显示在日志窗口中。从这一刻起，消息也可以发布到 JMS 队列 — 每当看到停车场几乎已满时。可以通过多种方式检查 JMS 队列的内容：在 WebLogic 控制台中，使用 QBrowser，通过具有入站 JMS 适配器或传输的 SOA 组合或服务总线服务，或者通过自定义 JMS 客户端。后者包含在联机资源中，即 CarParkNearlyFullQueueListener *项目中*。请注意，您可能需要更改源中的连接属性。

图 16-8 显示了 OEP 应用程序生成的日志记录输出片段。



**图 16-8。** *OEP应用程序为各种停车场产生的一些伐木 -*

*相关活动*

### 将停车场几乎已满的活动发布到活动 交付网络

当然，发布到 JMS 目标的消息可以由 SOA 组合应用程序中的入站 JMS 适配器读取，然后在事件交付网络 （EDN） 上转换为业务事件。但是，OEP 可以将事件直接发布到 EDN 上，如下所述。请注意，OEP 还具有一个入站 EDN 适配器，该适配器允许它读取和处理在 Eevent 交付网络上发布的业务事件。

EDN 事件是使用 EDL 文件和负载的关联 XSD 定义来定义的，如第 15 章中所述。这些文件将导入到 OEP 应用程序中，并从出站 EDN 适配器引用。将少量 SQL/XML 添加到 CQL 查询中，以正确的格式生成 EDN 事件的有效负载。每当 OEP 生成 CarParkNearFullEvent 时，都会触发订阅 EDN 事件的 SOA 组合应用程序。

下一步的源代码可在项目 *CarParking-step5 中找到*。

#### 用于响应几乎满员的停车场的 SOA 复合应用程序

创建一个名为 SaibotParkingStaff *的新 SOA 应用程序*，其中包含基于 *空*模板的同名项目。在他的应用程序中，我们将首先定义 EDN 事件，然后创建一个订阅该事件的调解器，并通过文件适配器将一行写入日志文件。

##### 定义 EDN 事件

在 Schemas 文件夹中创建名为 FacilitiesEvents.xsd 的 XSD 文件，其目标命名空间为 *saibot。 机场/设施*。定义一个元素 *CarParkAlmostFullEvent* 和与子项关联的类型 *carParkAlmostFullEventType*，以保存事件的关键属性。它们描述了 EDN 事件的有效负载。请查看联机资源以获取示例。

右键单击“事件”文件夹，然后选择“新建”|”事件定义。输入 *FacilitiesEvents* 作为 EDL 文件的名称，并使用与 XSD 文档中相同的命名空间。基于*具有相同名称*的 XSD 元素创建名为 CarParkAlmostFullEvent 的新事件。

##### 使用事件的调解器

将调解器添加到复合中，称为 *CarParkAlmostFullHandler*。选择模板 *Subscribe to Events*。选择 *CarParkAlmostFull* 事件作为要订阅的事件。

创建一个文件适配器绑定 *writeCarParkAlmostFullEntryInAlertLog*，该绑定将一行追加到名为 *CarParkAlertLog.txt* 的文件。每行都包含从 EDN 事件中获取的详细信息，描述了一个几乎已满的停车场。将调解器连接到文件适配器绑定。创建 XSL 映射以从 EDN 事件转换为文件适配器绑定的输入，并将其添加到路由规则中。

##### 部署和测试

将 SOA 组合部署到运行时。转到 EM FMW 控件中的“业务事件”页面 。发布事件的测试实例。然后检查是否创建了复合的新实例 ，以及测试事件的内容是否正确写入日志文件。如果成功 ，您可以继续下一部分。

#### 通过 EDN 集成扩展 OEP 应用程序

为了将 OEP 应用程序与 SOA Suite 事件交付网络连接起来，我们必须配置 EDN 出站适配器，并添加一个处理器，以将 OEP 消息转换为 EDN 事件 所需的 XML 格式。

##### 添加和配置 EDN 出站适配器

将 EDN 出站适配器添加到 EPN。将名称设置为 *CarparkAlmostFull-edn-outbound- adapter*。单击“下一步”。配置 EDN 适配器，将 JNDI 提供程序 URL 设置为 t3：//host：port（例如，集成 WLS 的 localhost：7101）。在 SOA 复合应用程序 SaibotParkingStaff 中浏览到在上一节中创建的 FacilitiesEvents.edl 文件。选择 *CarParkAlmostFullEvent* 作为要从 OEP 发布到 EDN 的事件。选中*“原始 XML 内容*”复选框。这告诉 EDN 适配器，传递到其中的事件已经具有正确的 XML 有效负载。

单击“完成”以完成 EDN 出站适配器的创建。您将收到一条消息，大意是 JDeveloper 已将 EDL 文件和关联的 XSD 文档导入 到 OEP 应用程序中。

##### 创建处理器以准备具有适当有效负载的 EDN 事件

向 EPN 添加一个名为 *carParkNearlyFullEventToEDNProcessor* 的新处理器。此处理器将为 *EDN 适配器准备 CarParkAlmostFullEvent*，并具有适当的 XML 有效负载。

将 *carParkNearlyFullChannel* 连接到此处理器的输入。创建一个新通道 *CarParkAlmostFullEventChannel* ，该通道从此处理器读取并发布到出站 EDN 适配器。

现在，我们需要为处理器准备 CQL 语句，以创建 EDN 事件的有效负载。由于我们已将 EDN 适配器配置为处理原始 XML，因此我们必须为 EDN 事件提供确切的有效负载。下面的代码片段显示了如何使用 SQL/ XML 指令在正确的命名空间中编写 XML。

 <query id=“CarParkAlmostFullEventEDNEventQuery”><！[CDATA[ 选择 XMLELEMENT（

NAME “CarParkAlmostFullEvent”，XMLATTRIBUTES（'saibot.airport/ facilities' as “xmlns”），

XMLELEMENT（名称“CarParkIdentifier”）

, ''||carparkIdentifier）、XMLELEMENT（NAME “CarParkName”）

，描述），

XMLELEMENT（名称 “PercentageFull” ，to\_char（百分比完整））， XMLELEMENT（名称 “容量” ，to\_char（容量）），

XMLELEMENT（名称 “CurrentCarCount” ，to\_char（lowestRecentCount））， XMLELEMENT（名称 “LowestRecentCarCount” ，to\_char （百分比完整））

） 作为 xmlContent

来自 carParkNearlyFullChannel [now] ]]></query>

此查询将使我们的处理器使用完全符合 EDN 事件的有效负载架构的 XML 内容创建 OEP 事件。EDN 适配器将接收这些事件，并将它们转换为 EDN 上的常规事件。

#### 部署和测试/运行

在我们的 OEP 服务器能够成功地与 SOA Suite 12c 事件交付网络通信之前，我们必须正确地设置它对 EDN 设置的期望。在默认情况下，EDN 基于 WebLogic JMS，我们必须添加 JVM 启动参数才能让 EDN 知道这一点。

打开文件 startwlevs.sh（或 .cmd，具体取决于您的操作系统）。几乎在文件末尾是运行 wlevs.jar 的调用。在此行中，您需要在“–jar”之前添加以下字符串：

 -Dedn.jms.topic=“jms/fabric/EDNTopic” -Dedn.jms.connection-factory=“jms/ fabric/EDNConnectionFactory”

进行此更改后，（重新）启动 OEP 服务器。

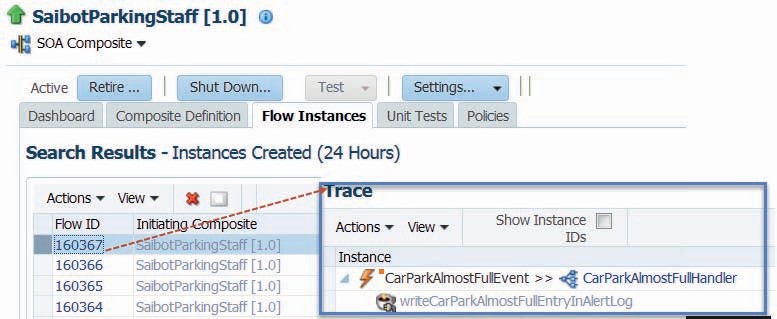
现在，您可以部署 OEP 应用程序。日志控制台应显示一条消息，指示 OEP 已创建 *EdnJmsConnectionFactory*  对象，通过该对象与 EDN 进行通信。

高枕无忧 - 在日志控制台中查看事件报告。然后检查 EM FMW 控件中是否有由 EDN 事件触发的 SOA 组合的新实例，该事件由 OEP 应用程序发布，如图 16-9 所示。最后一部分在 EM FMW 的流迹中不可见：EDN 事件的起源尚不清楚。

### 信用卡盗窃检测

在线资源以及所有代码示例中详细介绍了 OEP 的另一个有趣的用例。它涉及塞博特机场的购物区，那里发生了一系列信用卡盗窃案。经过调查，很明显，肇事者每次都在拥挤的主要航站楼区域偷走信用卡，走向

退出并在 15 分钟内使用信用卡进行了三次或更多次购买



**图 16-9。** *由*

*CarParkOEP 应用程序 CarParking 发布的 AlmostFull EDN 事件*

范围在 200 美元到 500 美元之间。他有时会往后走一点，进入一家比前一个出口离主出口稍远的商店。

OEP 设置为处理信用卡购买事件流，其中包含金额、信用卡号和商店标识符。使用OEP时，可以检测上述图16-10所示的模式。当它被发现时，它将以 EDN 事件的形式报告，由 SOA 组合处理。您还可以在线找到 ADF Web 应用程序的源代码，该应用程序将可疑事务可视化，其方式与 图 16-10 非常相似。

此 CreditCard *处理应用程序*的一些突出方面：

* + 使用 listAgg 运算符将多个事件聚合到 Java 集合中
  + 在 CQL 查询中使用自定义 Java 类
  + 使用 XMLParse 在带有子元素的 XML 元素中添加 XML 内容，基于字符串 [在自定义 Java 类中准备的商店标识符列表和购买

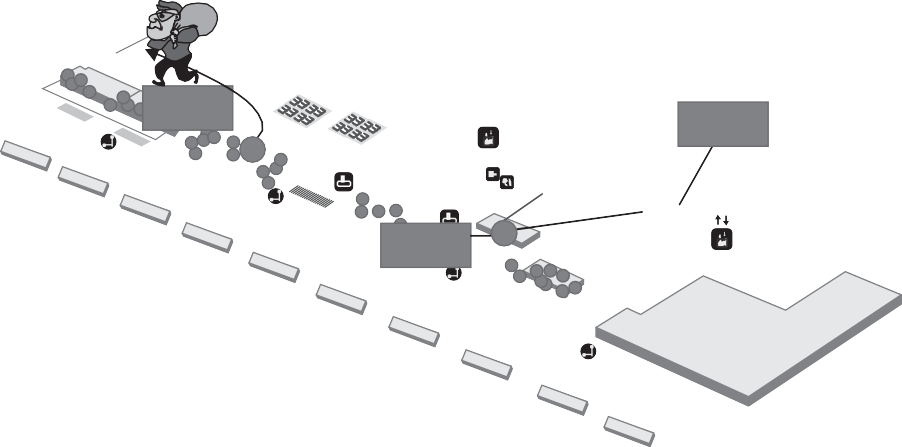
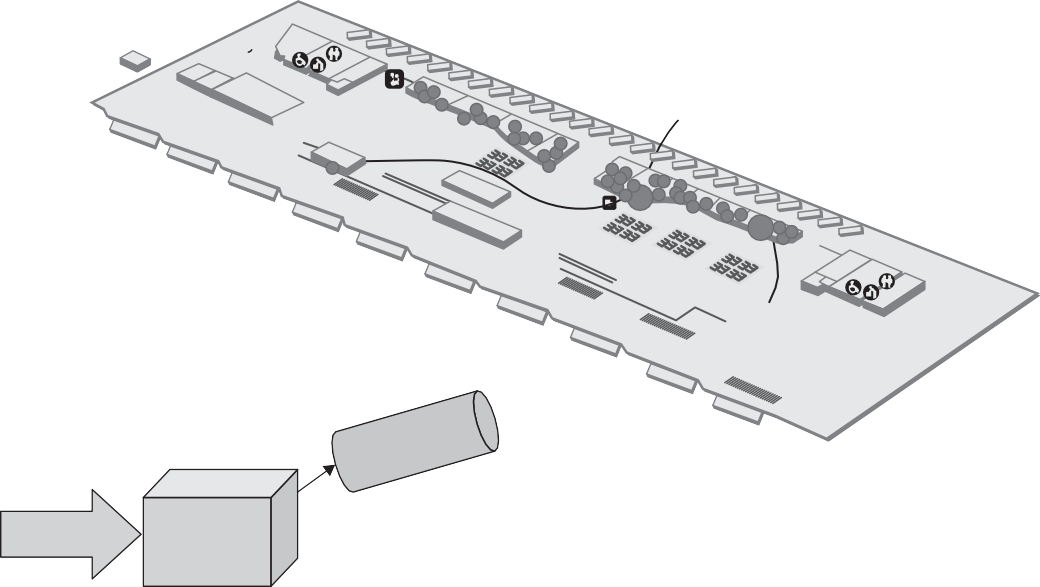
amounts] 生成更丰富、更复杂的 EDN 事件有效负载

* + 在 CQL 查询中使用 within *和* subset  *中的关键字*，以分别查找仅在特定时间段（15 分钟）内出现的模式，并允许聚合所有或

构成该模式的一些事件

**图 16-10。** *信用卡交易模式 OEP 的设置用于检测：一系列中间金额*

*在很短的时间内，大致沿着从主航站楼到出口的路线购买范围*



EDN

Medi- ator

EXIT

SPAD

JPJ L1-1

TAXI BAY

$250

$300

$380

L1-13-17

BUS BAY

SOA Suite 12c

Main Terminal

Fast Data

OEP

EDN

TransactionLog.txt

$440

下一个代码片段显示了 CQL 查询的一些关键方面，该查询用于查找可疑事务：

 选择 its.creditCardNumber 作为 creditCardNumber，totalSum 作为 sumAmount

，delta 作为 shoppingDuration，商店作为商店，purchaseAmount 作为 purchaseAmounts

从 creditCardMidRangePurchaseChannel MATCH\_RECOGNIZE （

按 creditCardNumber 划分

将 C.creditCardNumber 测量为 creditCardNumber

，sum（allPurchases.amount） 作为 totalSum

， C.element\_time - A.element\_time 作为 delta

，listAgg（allPurchases.shopIdentifier） 作为商店

， listAgg（allPurchases.amount） as purchaseAmount PATTERN （ A+ B？A+ C） 在 30000 毫秒内 SUBSET allPurchases = （A，B，C）

定义

A 作为 A.shopIdentifier >= A.shopIdentifier

、B 作为 B.shopIdentifier < A.shopIdentifier 和 A.shopIdentifier - B.shopIdentifier < 7

，C as A.shopIdentifier - C.shopIdentifier < 7

） 作为其

### 流资源管理器

Stream Explorer 是 Oracle 作为 OEP 12c 之上的扩展发布的。 Stream Explorer 是一个面向业务用户的基于浏览器的用户界面。通过极具吸引力的用户界面（如图 16-11 所示），用户可以通过关注功能细节，在流之上快速创建探索，并定义可视化这些探索的方法。在此过程中，他们正在创建 OEP 应用程序，如有必要，JDeveloper 中的开发人员可以进一步完善这些应用程序。当 OEP 服务器运行时，可以使用 URL http：//host：port/sx（例如 http://127.0.0.1:9002/sx）从浏览器访问流资源管理器。

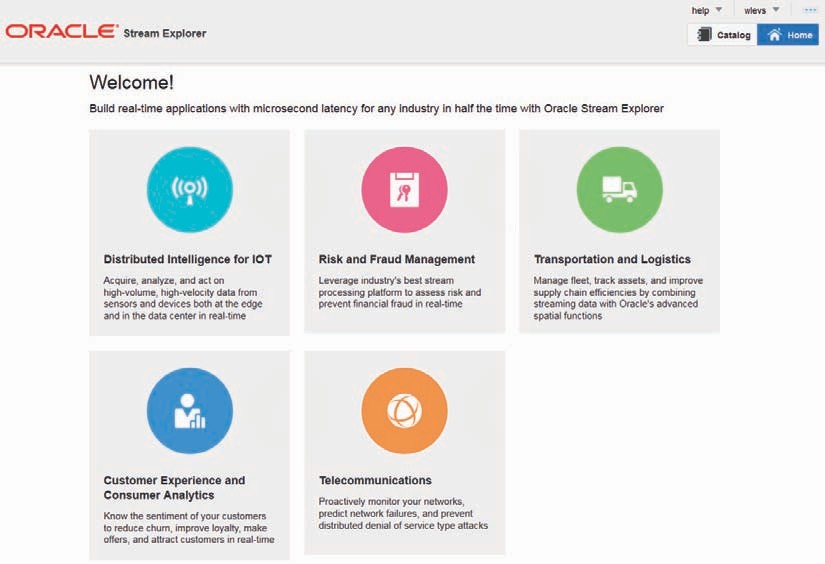
业务用户为输入定义 Streams。此类流可以是 （CSV） 文件、HTTP 订阅服务器、EDN 事件、JMS 目标、数据库表或 REST 服务。这些流 携带形状实例。形状是由属性及其类型组成的记录（事件）定义。精通技术的 IT 同事也许可以为 LoB 用户预配置 Streams 。然后，业务用户可以在基于预定义模板的流上定义探索 。业务用户可以使用许多按行业分类的预定义模板。这些模板表示处理事件的特定方式。

业务用户必须为模板参数（如聚合间隔、聚合函数或事件属性）提供值以分组依据。可以链接多个探索;这意味着一次探索的结果可以为下一次探索提供输入。

可以将探索结果发送到的下游目的地包括

CSV 文件、HTTP 发布者、EDN、JMS 目标、REST 服务或 BAM（业务活动监视器）。

Stream Explorer 是安装在 Jetty Server 中的 Web 应用程序，它还运行 OEP 引擎和 OEP 可视化工具管理控制台。可以使用 Stream Explorer 配置的 OEP 应用程序导出以进行导入和优化 |extended |在 JDeveloper 中通过以下方式增强



**图 16-11。** *流资源管理器的用户友好主页*

IT 专家。流资源管理器应用程序包含事件处理网络和 使用模板构建的基于 CQL 的事件处理器。

流资源管理器在不久的将来会包含与 NoSQL 源和目标的交互、BAM 可视化效果、对机器学习算法的支持、敏感数据的数据编辑、 Coherence 缓存集成和基于位置的功能。

### 总结

简而言之，Oracle Event Processor 可以为我们做些什么，这些发现基于来自各种来源的大量数据，并且可能很快，可以发布到各种类型的渠道进行后续处理。通过不断扫描在不可预测的时刻到达的事件流，OEP 能够寻找异常情况、有意义的模式、 相关聚合，甚至是重要的事件缺席。

事件处理应用程序使用 CQL（连续查询语言），该语言在语法上类似于 SQL，可以联接事件通道、缓存和关系数据源，以生成更粗粒度、更高价值的输出事件。OEP 通过 JMS、Coherence 和事件交付网络在入站和出站边缘与 SOA Suite 和 Java 应用程序进行连接。

通过关注OEP的核心竞争力——轻量级、多渠道交互、跨接口|会话|交易|对话——不难看出，例如，OEP如何在监控和跨业务流程方面发挥重要作用。请注意，OEP 引擎的核心 已嵌入到 Oracle BAM 中，以精确地帮助实现这一点。

SOA Suite 中对 OEP 与事件交付网络集成的支持为使用 SOA Suite 实现 EDA（事件驱动架构）提供了巨大的潜力，或者至少使服务架构更加支持事件。最后，从业务角度来看，Stream Explorer 产品降低了开始利用事件处理的门槛。