Web 服务的业务流程执行语言 2.0

OASIS 标准

2007.4.11

规范 URI：

这个版本：

http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.doc
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

先前版本：

http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html
http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.doc
http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf

最新版本：

http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.doc
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

技术委员会：

OASIS Web Services Business Process Execution Language (WSBPEL) TC

主讲：

Diane Jordan, IBM
John Evdemon, Microsoft

编辑：

Alexandre Alves, BEA
Assaf Arkin, Intalio
Sid Askary, Individual
Charlton Barreto, Adobe Systems
Ben Bloch, Systinet
Francisco Curbera, IBM
Mark Ford, Active Endpoints, Inc.
Yaron Goland, BEA
Alejandro Guízar, JBoss, Inc.
Neelakantan Kartha, Sterling Commerce
Canyang Kevin Liu, SAP
Rania Khalaf, IBM
Dieter König, IBM
Mike Marin, IBM, formerly FileNet Corporation
Vinkesh Mehta, Deloitte

Satish Thatte, Microsoft
Danny van der Rijn, TIBCO Software
Prasad Yendluri, webMethods
Alex Yiu, Oracle

相关工作：

- 见第 3 章

声明的 XML 命名空间：

http://docs.oasis-open.org/wsbpel/2.0/process/abstract
http://docs.oasis-open.org/wsbpel/2.0/process/executable
http://docs.oasis-open.org/wsbpel/2.0/plnktype
http://docs.oasis-open.org/wsbpel/2.0/serviceref
http://docs.oasis-open.org/wsbpel/2.0/varprop

摘要:

本文档定义了基于 Web 服务的指定的业务流程语言。称作 Web 服务的业务流程执行语言（此后在文档中简称 WS-BPEL）。WS-BPEL 中的流程通过专门的 Web 服务接口实现功能性的输出和输入。
业务流程有两种描述方法。可执行业务流程模拟业务交互中参与者的真实行为。抽象业务流程部分说明没有被确定执行的流程。抽象程序可能隐藏某些必需的具体操作细节。它提供一个角色说明，使用多个可能的案例，包括可视化行为和程序模板。WS-BPEL 被用来模拟可执行的和抽象流程的行为。
WS-BPEL 提供可执行以及抽象业务流程规范的语言。通过这个，它扩展了 Web 服务交互模型并且能够支持业务事务。WS-BPEL 定义了一个可兼容的集成模型，可以促进组织内部或者 B2B 模式中的自动化流程扩展。

地位:

本文档在 2007 年 4 月 11 日被 WSBPEL TC 最后修订和核准。认可级别也如上所示。
注意检查文档的当前版本信息。文档会定时更新但没有确切的进度表。
技术委员会成员应将有关这个规范的评论发送到技术委员会的邮箱列表中。其他人可以通过点击 http://www.oasis-open.org/committees/wsbpel网页上的“send A Comment”按钮来发送评论。
相关专利权信息以及专利授权条件，请查阅技术委员会网页 http://www.oasis-open.org/committees/wsbpel/ipr.php.上的知识产权章节。
本规范的非标准化勘误表可以在 http://www.oasis-open.org/committees/wsbpel.查找。

注意事项：

OASIS 没有在任何知识产权或者其他能够被用来声明保持文档中所描述技术的实施和应用权利或者任何的上述权利下认证的适用范围。同样地，它也没有展示出它为了确认这些权利做了任何努力。有关 OASIS 规范权利的手续信息可以在 OASIS 的网站上找到。可以进行合法的出版和拷贝，以及授权文件的担保。

OASIS 希望任何当事人注意版权，专利权或者专利申请，或者其他所有权中可能会涉及到本规范的技术。请将信息告知 OASIS 执行理事会。

目录：

1. 简介

Web 服务的目的是通过使用 Web 标准来完成应用程序之间的互操作性。Web 服务使用低耦合的集成模型来允许灵活的不同系统间的集成，包括企业-消费者，企业-企业以及企业应用程序集成等领域。下列技术规范定义了最初的 Web 服务空间：SOAP [SOAP 1.1],Web 服务描述语言（WSDL）[WSDL 1.1],以及统一描述、发现，以及集成（UDDI）[UDDI]. SOAP 为基本服务的互操作性定义了 XML 消息传递协议。WSDL 介绍了描述服务的基本语法。UDDI 为系统地发布和发现服务提供了所需的基础结构。这些规范允许应用程序在低耦合，独立平台模型下互相查找和交互。

系统集成要求更多的通过使用标准协议来进行简单交互的能力。仅当应用程序和业务流程能够通过使用标准流程集成模型来集成复杂的交互时才能发挥 Web 服务作为集成平台的全部潜力。由 WSDL 直接提供的交互模型本质上是请求-响应或者单项无约束的交互模型。

典型的业务交互模型假定对等消息交换序列，不管是请求-响应还是单向，在有状态的长时间运行的交互里包括两方或多方。为了定义这样一个业务交互，需要一个由业务流程在交互中使用的正式的消息交换协议描述。可以使用抽象消息来描述各个群里调用的显示消息交换行为，不包括显示它们的内在执行。分离业务流程的公共特征和内部或者私有特征有两个好处。一个原因是企业显然不想让它的业务伙伴知道它的所有的内部决策和数据管理。另一个原因是（即便并不是这种情况）通过把公共流程和私有流程分开，您能够改变流程实现的私有部分而不会影响到公共业务协议。必须用平台无关的方式来明确地描述业务协议，业务协议必须包括在跨企业业务中的所有重要行为部分。

以下描述业务流程的概念应该注意：

- 业务流程包括数据相关行为。比如：供应链程序依赖于这样的数据-同一个订单中大量的行式条目，订单的总价值，或者交付的最终期限。这些案例中的业务意向要求使用有条件的和超时构造器。
- 指定异常条件及其结果，包括恢复序列的能力最低也同业务流程在“一切运行良好”的情况下定义行为的能力一样重要。
- 长期运行的交互包括多个工作单元，这些单元常常是嵌套的，每个都有其自身的数据要求。业务流程经常要求多级粒度的跨伙伴输出（成功或者失败）工作单元配合。

WS-BPEL 的基本概念可以被应用于两种方式，抽象或者可执行。

WS-BPEL 抽象程序是一个部分指定程序并不规定为可执行的并且必须声明为“抽象的”。然而可执行程序是完整指定的因此可以被执行，抽象程序可以隐藏某些具体的必须通过可执行人为因素来表达的操作细节。

所有的可执行流程的构造器对抽象流程都是可用的，因此可执行的和抽象的 WS-BPEL 流程共享同样的表达力。除了可执行流程的可用特性之外，抽象流程提供两种隐藏操作细节的机

制：（1）使用显示不透明标记；（2）省略特殊的可能包含交付给可执行流程的完整信息的抽象流程定义，它已经给出的具体实现的抽象状态，被允许执行额外处理的步骤同观众并不相关。

抽象流程提供了一个描述角色，使用多个用例。一个用例可能描述可执行程序提供的部分或者全部服务的显示行为。另一个用例可能定义补充特殊域最优方法的程序模板。这样一个程序模板可能捕获在某种意义上和设计时间表达相兼容的基本程序逻辑，同时排除映射到可执行程序的被完成的执行细节。

共基极指定了定义抽象流程语法域的特征。共基极中的应用轮廓提供了基于抽象程序特殊用法的可执行 WS-BPEL 的必要的专业性和语义。

如上所提及的，使用 WS-BPEL 来定义可执行业务程序是可能的。然从私有实现的角度来看并不需要完整地定义 WS-BPEL 流程，但是 WS-BPEL 为仅依赖于 Web 服务资源和 XML 数据的业务流程有效地定义了可移植的执行格式。此外，这种可兼容的流程执行和伙伴交互可以无关于主机环境设备使用的支持平台或者编程模型。

WS-BPEL 抽象和可执行流程之间的基本概念模型的连续性使得可以像程序或者角色模板一样导入和导出抽象流程的公共特征，同时维护了外部可视化行为的意图和结构。这个甚至被应用到私有工具方面使用平台相关函数操作。这是 WS-BPEL 应用的一个关键特征-开放 Web 服务，因为它允许扩展工具和其他技术，极大的提高了自动化水平并且通过建立跨企业自动化业务流程来降低成本。

在本规范中，抽象业务流程的描述在可执行流程之后。我们将在 13.WS-BPEL 抽象流程章节中清楚的区分抽象业务程序和可执行程序的概念。

WS-BPEL 定义了描述基于程序和伙伴交互的业务程序行为的模型和语法。同每个伙伴的交互通过 Web 服务接口发生，并且接口级别的架构关系被压缩在伙伴链接中。WS-BPEL 程序定义了与伙伴的多服务交互是怎样协调一致来完成业务目标的，也定义了这个协调的状态和必要逻辑。同样，WS-BPEL 介绍了处理业务异常和编程故障的系统机制。此外，WS-BPEL 还介绍了一种机制来定义当异常发生或者伙伴请求撤销时，单元功能内的单个或复合活动是如何被赔偿的。

WS-BPEL 利用了若干 XML 规范：WSDL 1.1，XML Schema 1.0，Xpath 1.0 和 XSLT 1.0。WSDL 消息和 XML Schema 类型定义提供了由 WS-BPEL 流程使用的数据模型。Xpath 和 XSLT 提供了数据操作支持。所有的资源和伙伴被描绘为 WSDL 服务。WS-BPEL 所提供的可扩展性能支持这些标准的未来版本，即用于 XML 计算的 XPath 和相关标准。

WS-BPEL 程序是一个可重复使用的定义，可以在不同的方法和不同的场景中配置，同时在上面维持一致的应用程序级的行为。在本规范中 WS-BPEL 程序的部署描述是在作用域之外的。

2.符号约定

本文中的关键字“**必须（MUST）**”、“**绝不可以（MUST NOT）**”、“**需要的（REQUIRED）**”、“**应该（SHALL）**”、“**将不（SHALL NOT）**”、“**应该（SHOULD）**”、“**不应该（SHOULD NOT）**”、“**推荐的（RECOMMENDED）**”、“**可以（MAY）**”和“**可选的（OPTIONAL）**”将按 [RFC 2119] 中的描述来解释。

名称空间 URI（常规形式是“some-URI”）表示 [RFC 2396]中定义的与应用程序相关或与内容相关的某个 URI。

本规范使用非正式的语法来描述 XML 片段的 XML 语法，如下：

- 本语法以 XML 实例的形式出现，但其中的值代表数据类型而不是值。
- 粗体显示的语法是还未在本文中介绍过的语法，或在示例中有特别的意义。
- <-- description --> 是某些“其它”名称空间的元素的占位符（象 XSD 中的 ##other）。
- 字符按以下方式被附加到元素、属性和 <!-- descriptions -->：“?”（0 个或 1 个）、“*”（0 个或更多个）、“+”（1 个或更多个）。字符“[”和“]”用以表示所包含的项应作为一个与“?”、“*”或“+”字符有关的组被处理。
- 被“|”分隔且被“(”和“)”并排在一起的元素和属性应被理解为语法的候选式。
- XML 名称空间前缀（在下文中定义）被用来指出被定义的元素的名称空间。
- 使用者定义的扩展活动的名字由 anyElementQName 指出。

语法规范如下颜色显示：

```
<variables>
   <variable name="BPELVariableName"
             messageType="QName"?
             type="QName"?
             element="QName"?>+
      from-spec?
   </variable>
</variables>
```

以 <?xml 开头的示例包含足够的信息，这些示例符合本规范；其它示例是片断，需指定更多信息才能符合本规范。

本文档中的例子和其他解释资料不是完整的除非特别声明。例如,某些例子导入 WSDL 定义并未在本文档中指定。

例子如下颜色显示：

```
<variable xmlns:ORD="http://example.com/orders"
          name="orderDetails" messageType="ORD:orderDetails" />
```

所提供的 XSD schema 定义是语法的正式定义 [xml-schema1 Part 1]。独立XML Schema文件，附录中的XML Schema，说明文本中的伪XML Schema，以及标准说明文本并不一致，标准说明文本优先权高于独立XML Schema文件。WS-BPEL XML Schemas提供了标准化的XML语法细节补充，例如关于扩展WS-BPEL程序定义的详细资料，只要那些XML语法细节不会违反显示的标准说明文本。

XML Schemas 只实施在标准说明文本中描述的约束子集。因此 WS-BPEL 的人工制品，比如程序定义，只对 XML Schemas 有效，而对标准说明文本无效。

本规范使用许多的名称空间前缀，它们关联的 URI 列举如下。注意名称空间前缀的选择是任意的、非标准化的及非重要的语义。

- **xsi** - "http://www.w3.org/2001/XMLSchema-instance"
- **xsd** - "http://www.w3.org/2001/XMLSchema"
- **wsdl** - "http://schemas.xmlsoap.org/wsdl/"
- **vprop** - "http://docs.oasis-open.org/wsbpel/2.0/varprop"
- **sref** - "http://docs.oasis-open.org/wsbpel/2.0/serviceref"
- **plnk** – "http://docs.oasis-open.org/wsbpel/2.0/plnktype"
- **bpel** – "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
- **abstract** – "http://docs.oasis-open.org/wsbpel/2.0/process/abstract"

3. 同其他规范的联系

WS-BPEL依赖于以下基于 XML 的规范：WSDL 1.1、XML Schema 1.0 和 XPath 1.0。所有的WS-BPEL执行应该是可配置的以便它们可以共享Basic Profile 1.1[WS-I Basic Profile]使得交互一致。WS-BPEL执行可以允许Basic Profile1.1 配置停用，甚至对Basic Profile 1.1 包含的情景也是相同的。

WSDL 对 WS-BPEL 语言的影响最大。WS-BPEL 流程模型位于由 WSDL 1.1 所定义的服务模型之上。位于 WS-BPEL 流程模型核心的是由 WSDL 描述的服务间的对等交互概念；流程及其伙伴都被建模成 WSDL 服务。业务流程定义了怎样协调流程实例与它的伙伴间的交互。在这个意义上，一个 WS-BPEL 流程定义提供和 / 或使用一个或多个 WSDL 服务，还通过 Web 服务接口提供流程实例相对于它的伙伴和资源的行为和交互的描述。也就是说，WS-BPEL 描述了交互中某个角色的业务流程遵守的消息交换协议。

WS-BPEL 业务流程的定义也遵循 WSDL 的分离模型，即把业务流程使用的抽象消息内容与部署信息（消息和 portType 与绑定和地址信息）分开。具体地说，WS-BPEL 流程用抽象 WSDL 接口（portType 和操作）来表示所有的伙伴以及与这些伙伴的交互；它并不引用流程实例使用的实际服务。WS-BPEL 不会对 WSDL 绑定做任何假设。约束，多义性，提供或者缺少 WSDL 绑定的权能超出了本规范的范围。

不管怎样，WSDL 的抽象部分没有定义约束利用具体绑定支持的通信模式。因此，WS-BPEL 可以定义相对于伙伴服务的并不被所有可能的绑定支持的行为，并且可以出现某些绑定对 WS-BPEL 程序定义失效的情况。

尽管 WS-BPEL 努力提供尽可能多的同 WSDL 的兼容性，但是下列三种情况无效：

- 故障命名的约束，本章稍后将会讨论（见 10.3 调用 Web 服务操作-Invoke）
- [SA00002]WSDL端口类型的超载操作名字。不管WS-I Basic Profile配置是否被激活，WS-BPEL处理器必须拒绝包含超载操作名字的WSDL端口类型定义。这个约束被视为合理的因为超载操作是很少见的，实际上它们在WS-I Basic Profile中是被禁止的，并且支持它们带来的复杂性大于好处。
- [**错误！未找到引用源。**SA00001]在WSDL 1.1 规范中定义了包含请求-响应或者标识操作的端口类型。不管WS-I Basic Profile配置是否被激活，WS-BPEL处理器必须拒绝依赖于这样的端口类型的WS-BPEL。

在这个规范被完成的时候，各种 Web 服务标准工作，比如 WSDL 2.0 和 WS-Addressing 还正在进行并且没有为 WS-BPEL 2.0 做好准备。WS-BPEL 以后的版本可能会提供这些标准的支持。

应该注意这个规范中提供的例子选定的 Schema 在 "http://schemas.xmlsoap.org/wsdl/2004-08-24.xsd"，名称空间 URI 是 http://schemas.xmlsoap.org/wsdl/ [WSDL 1.1].这个XML Schema合并已知错误，并且是通过

[WS-I Basic Profile 1.1 Errata] (2005.10.25)选择的XML Schema。

4. 业务流程的静态分析

WS-BPEL 将一致实现必须执行基本静态分析（附录 B）来检测和拒绝那些缺少静态分析检查的程序定义作为总原则。请注意这样的分析可能在某些情况下阻止流程的作用，实际上，创建有错误的情况，要么有具体用途要么有其他用途。例如：WS-BPEL 实现将拒绝带有访问定义变量的<invoke>活动的程序，实际上在程序执行期间<invoke>活动可能无法达到那里。

WS-BPEL 实现可以执行额外的超出本规范要求的基本静态分析如发送警告信号或者拒绝程序定义的静态分析检查。这样一个实现应该是可配置的以便禁止那些非规范的静态分析检查。

5. 定义业务程序

5.1. 初始例子

在详细描述业务流程的结构之前，本章描述了 WS-BPEL 的一个简单例子-处理订单。目的是介绍最基本的结构和语言的某些最基本概念。

程序操作非常简单，在 Figure 1: Purchase Order Process Outline中描述。点线代表顺序。任意地把序列组成一组表示并发序列。实心箭头表示用于并发活动间的同步的控制链接。请注意这张图并不是 WS-BPEL流程的正式图解表示法。这张非正式的图被用来帮助读者理解。

当收到客户的购买定单后，流程初始化三个并行的任务：计算定单的最终价格、选择承运人以及为定单安排生产和运输。虽然有些处理可以并行地进行，但是三个任务之间存在相互依赖的控制和数据。具体地说，在计算最终价格时需要运输价格，在全面安排实现计划时需要运输日期。在完成这三个任务后就可以开始处理发票并把发票交给客户。

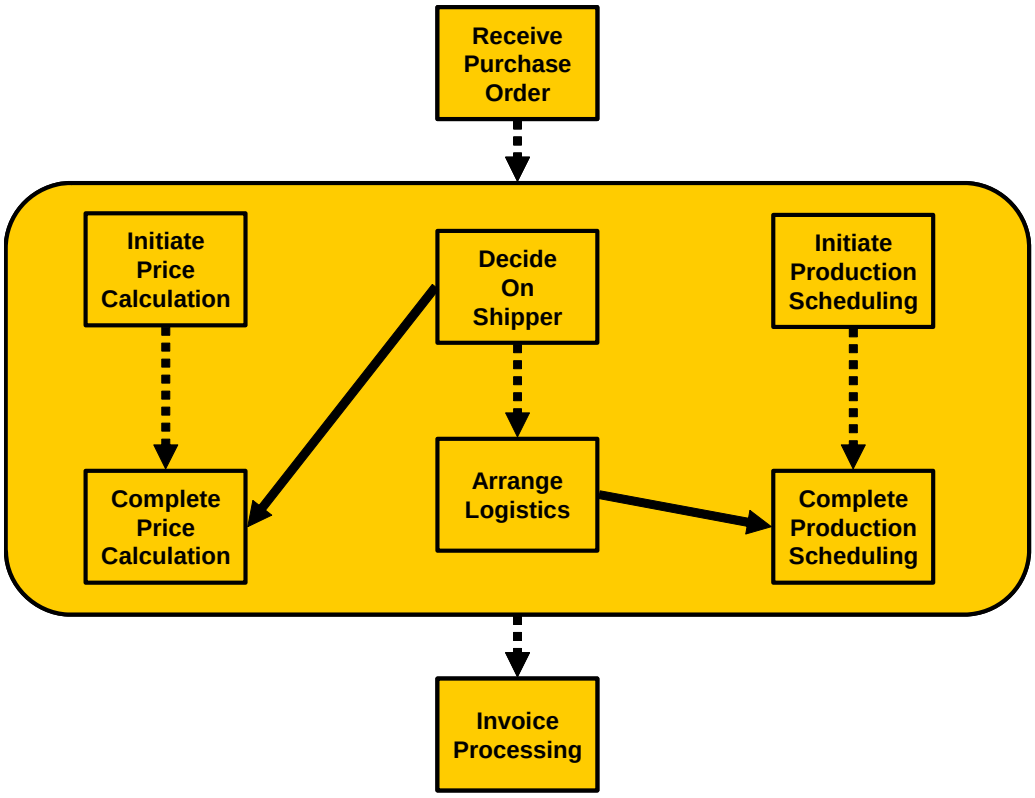
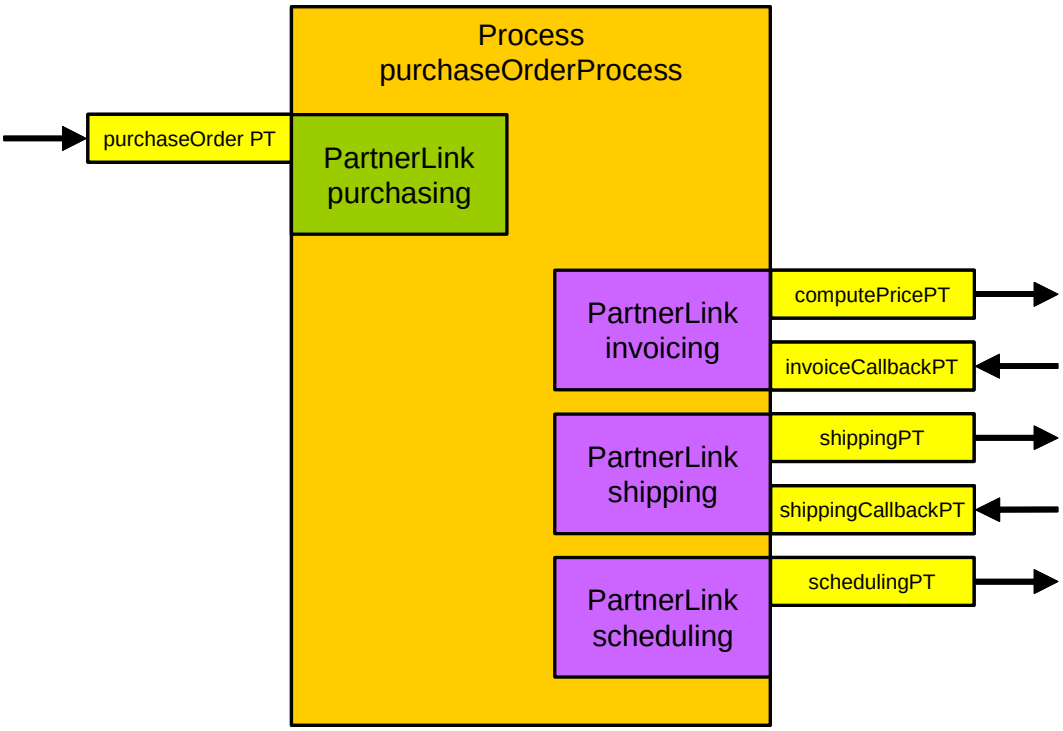


Figure 1: Purchase Order Process - Outline

在下面的 WSDL 文档中显示了由服务提供给客户的 WSDL 端口类型(purchaseOrderPT)。简单起见，其他业务流程必需的 WSDL 定义包含在同一个 WSDL 文档中，特别地，Web 服务的端口类型提供了价格计算，运输选择和调度，以及生产调度功能。请注意 WSDL 文档中没有绑定或服务元素。WS-BPEL 程序通过引用程序中被调用服务的端口类型来定义，并且不引用

它们可能的部署。以这种方式定义的业务流程允许在兼容的服务的多次部署中再次使用业务流程定义。

WSDL 文档末尾的<partnerLinkType>表示了订单服务和每个与其交互的群之间的交互（见 6.伙伴链接类型，伙伴链接以及端点引用）。<PartnerLinkType>可以用来表示服务间的依存关系，不管 WS-BPEL 业务程序是否定义一个或者多个服务。每个<partnerLinkType>最多定义两个“角色”名称，并且列举每个角色必须支持的端口类型以便交互成功执行。在本例中，两个<partnerLinkType>，”purchasingLT”和”schedulingLT”<partnerLinkType>表示了订单服务和调度服务之间的交互，只有后者的操作被调用。另外两个<partnerLinkType>，”invoicingLT”和”shippingLT”，定义两个角色因为发票计算和运送服务的用户（发票或者运送调度）必须都提供回调操作来启动发送通知报告（”invoiceCallbackPT”和”shippingCallbackPT”端口类型）。




```
<wsdl:definitions
   targetNamespace="http://manufacturing.org/wsdl/purchase"
   xmlns:sns="http://manufacturing.org/xsd/purchase"
   xmlns:pos="http://manufacturing.org/wsdl/purchase"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:types>
      <xsd:schema>
         <xsd:import
namespace="http://manufacturing.org/xsd/purchase"
          schemaLocation="http://manufacturing.org/xsd/purchase.xsd"
```

```
/>
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="POMessage">
      <wsdl:part name="customerInfo" type="sns:customerInfoType" />
      <wsdl:part name="purchaseOrder" type="sns:purchaseOrderType" />
   </wsdl:message>
   <wsdl:message name="InvMessage">
      <wsdl:part name="IVC" type="sns:InvoiceType" />
   </wsdl:message>
   <wsdl:message name="orderFaultType">
      <wsdl:part name="problemInfo" element=”sns:OrderFault " />
   </wsdl:message>
   <wsdl:message name="shippingRequestMessage">
      <wsdl:part name="customerInfo" element="sns:customerInfo" />
   </wsdl:message>
   <wsdl:message name="shippingInfoMessage">
      <wsdl:part name="shippingInfo" element="sns:shippingInfo" />
   </wsdl:message>
   <wsdl:message name="scheduleMessage">
      <wsdl:part name="schedule" element="sns:scheduleInfo" />
   </wsdl:message>

   <!-- portTypes supported by the purchase order process -->
   <wsdl:portType name="purchaseOrderPT">
      <wsdl:operation name="sendPurchaseOrder">
         <wsdl:input message="pos:POMessage" />
         <wsdl:output message="pos:InvMessage" />
         <wsdl:fault name="cannotCompleteOrder"
            message="pos:orderFaultType" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:portType name="invoiceCallbackPT">
      <wsdl:operation name="sendInvoice">
         <wsdl:input message="pos:InvMessage" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:portType name="shippingCallbackPT">
      <wsdl:operation name="sendSchedule">
         <wsdl:input message="pos:scheduleMessage" />
      </wsdl:operation>
   </wsdl:portType>
```

```
<!-- portType supported by the invoice services -->
<wsdl:portType name="computePricePT">
   <wsdl:operation name="initiatePriceCalculation">
      <wsdl:input message="pos:POMessage" />
   </wsdl:operation>
   <wsdl:operation name="sendShippingPrice">
      <wsdl:input message="pos:shippingInfoMessage" />
   </wsdl:operation>
</wsdl:portType>

<!-- portType supported by the shipping service -->
<wsdl:portType name="shippingPT">
   <wsdl:operation name="requestShipping">
      <wsdl:input message="pos:shippingRequestMessage" />
      <wsdl:output message="pos:shippingInfoMessage" />
      <wsdl:fault name="cannotCompleteOrder"
         message="pos:orderFaultType" />
   </wsdl:operation>
</wsdl:portType>

<!-- portType supported by the production scheduling process -->
<wsdl:portType name="schedulingPT">
   <wsdl:operation name="requestProductionScheduling">
      <wsdl:input message="pos:POMessage" />
   </wsdl:operation>
   <wsdl:operation name="sendShippingSchedule">
      <wsdl:input message="pos:scheduleMessage" />
   </wsdl:operation>
</wsdl:portType>

<plnk:partnerLinkType name="purchasingLT">
   <plnk:role name="purchaseService"
      portType="pos:purchaseOrderPT" />
</plnk:partnerLinkType>

<plnk:partnerLinkType name="invoicingLT">
   <plnk:role name="invoiceService"
      portType="pos:computePricePT" />
   <plnk:role name="invoiceRequester"
      portType="pos:invoiceCallbackPT" />
</plnk:partnerLinkType>

<plnk:partnerLinkType name="shippingLT">
   <plnk:role name="shippingService"
```

```
         portType="pos:shippingPT" />
      <plnk:role name="shippingRequester"
         portType="pos:shippingCallbackPT" />
   </plnk:partnerLinkType>

   <plnk:partnerLinkType name="schedulingLT">
      <plnk:role name="schedulingService"
         portType="pos:schedulingPT" />
   </plnk:partnerLinkType>

</wsdl:definitions>
```

下面定义的是订单服务的业务程序。程序定义中有四个主要部分。请注意例子提供了一个简单的情况。为了完成它，需要额外的元素比如<correlationSets>.

- <partnerLinks>部分定义了不同的在处理订单期间与业务程序交互的群。这里显示的四个<partnerLink>定义对应于订单发送器(customer),价格提供者(invoicing),出货方（shipping provider），生产调度服务（scheduling provider）。每个<partnerLink>由 partnerLinkType 和一个或两个角色名字来标识。这个信息表示了必须由业务程序和伙伴服务来提供的使该关系成功的功能，即订单程序和伙伴需要执行的端口类型。
- <variables>部分定义了程序使用的数据变量，根据 WSDL 消息类型、XML Schema 类型（简单或复杂）、XMLSchema 元素提供的定义。变量允许流程在消息交换时维持状态。
- <faultHandlers>部分包括故障处理器定义了在调用评估和认可服务时对产生的故障必须响应的活动。在 WS-BPEL 中，所有的故障，不管是内部的还是服务调用产生的，都由一个修饰名标识。特别是 WS-BPEL 中的每个 WSDL 故障都由修饰名识别,这个修饰名由同被定义的端口类型和故障相关联的 WSDL 文档的目标名称空间以及故障的 NCName 组成。
- 余下的<process>定义包含购买请求的正常执行的描述。其主要元素将在下面的程序定义部分解释。

```
<process name="purchaseOrderProcess"
   targetNamespace="http://example.com/ws-bp/purchase"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:lns="http://manufacturing.org/wsdl/purchase">

   <documentation xml:lang="EN">
      A simple example of a WS-BPEL process for handling a purchase
      order.
   </documentation>

   <partnerLinks>
      <partnerLink name="purchasing"
         partnerLinkType="lns:purchasingLT"  myRole="purchaseService"
```

```
/>
      <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"
         myRole="invoiceRequester" partnerRole="invoiceService" />
      <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
         myRole="shippingRequester" partnerRole="shippingService" />
      <partnerLink name="scheduling"
         partnerLinkType="lns:schedulingLT"
         partnerRole="schedulingService" />
   </partnerLinks>

   <variables>
      <variable name="PO" messageType="lns:POMessage" />
      <variable name="Invoice" messageType="lns:InvMessage" />
      <variable name="shippingRequest"
         messageType="lns:shippingRequestMessage" />
      <variable name="shippingInfo"
         messageType="lns:shippingInfoMessage" />
      <variable name="shippingSchedule"
         messageType="lns:scheduleMessage" />
   </variables>

   <faultHandlers>
      <catch faultName="lns:cannotCompleteOrder"
         faultVariable="POFault"
         faultMessageType="lns:orderFaultType">
         <reply partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder" variable="POFault"
            faultName="cannotCompleteOrder" />
      </catch>
   </faultHandlers>

   <sequence>
      <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
         operation="sendPurchaseOrder" variable="PO"
         createInstance="yes">
         <documentation>Receive Purchase Order</documentation>
      </receive>

      <flow>
         <documentation>
            A parallel flow to handle shipping, invoicing and
            scheduling
         </documentation>
```

```
        <links>
           <link name="ship-to-invoice" />
           <link name="ship-to-scheduling" />
        </links>
        <sequence>
           <assign>
              <copy>
                 <from>$PO.customerInfo</from>
                 <to>$shippingRequest.customerInfo</to>
              </copy>
           </assign>
           <invoke partnerLink="shipping" portType="lns:shippingPT"
              operation="requestShipping"
              inputVariable="shippingRequest"
              outputVariable="shippingInfo">
              <documentation>Decide On Shipper</documentation>
              <sources>
                 <source linkName="ship-to-invoice" />
              </sources>
           </invoke>
           <receive partnerLink="shipping"
              portType="lns:shippingCallbackPT"
              operation="sendSchedule" variable="shippingSchedule">
              <documentation>Arrange Logistics</documentation>
              <sources>
                 <source linkName="ship-to-scheduling" />
              </sources>
           </receive>
        </sequence>
        <sequence>
           <invoke partnerLink="invoicing"
              portType="lns:computePricePT"
              operation="initiatePriceCalculation"
              inputVariable="PO">
              <documentation>
                 Initial Price Calculation
              </documentation>
           </invoke>
           <invoke partnerLink="invoicing"
              portType="lns:computePricePT"
              operation="sendShippingPrice"
              inputVariable="shippingInfo">
              <documentation>
                 Complete Price Calculation
```

```
            </documentation>
            <targets>
               <target linkName="ship-to-invoice" />
            </targets>
         </invoke>
         <receive partnerLink="invoicing"
            portType="lns:invoiceCallbackPT"
            operation="sendInvoice" variable="Invoice" />
      </sequence>
      <sequence>
         <invoke partnerLink="scheduling"
            portType="lns:schedulingPT"
            operation="requestProductionScheduling"
            inputVariable="PO">
            <documentation>
               Initiate Production Scheduling
            </documentation>
         </invoke>
         <invoke partnerLink="scheduling"
            portType="lns:schedulingPT"
            operation="sendShippingSchedule"
            inputVariable="shippingSchedule">
            <documentation>
               Complete Production Scheduling
            </documentation>
            <targets>
               <target linkName="ship-to-scheduling" />
            </targets>
         </invoke>
      </sequence>
   </flow>
   <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="Invoice">
      <documentation>Invoice Processing</documentation>
   </reply>
</sequence>

</process>
```

5.2. 业务程序结构

本章提供了 WS-BPEL 语义的快速摘要。它只提供了一个简短的概述，每个语言结构的详细内容在文档的其余部分描述。

```
<process name="NCName" targetNamespace="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <extensions>?
      <extension namespace="anyURI" mustUnderstand="yes|no" />+
   </extensions>

   <import namespace="anyURI"?
      location="anyURI"?
      importType="anyURI" />*

   <partnerLinks>?
      <!-- Note: At least one role must be specified. -->
      <partnerLink name="NCName"
         partnerLinkType="QName"
         myRole="NCName"?
         partnerRole="NCName"?
         initializePartnerRole="yes|no"?>+
      </partnerLink>
   </partnerLinks>

   <messageExchanges>?
      <messageExchange name="NCName" />+
   </messageExchanges>

   <variables>?
      <variable name="BPELVariableName"
         messageType="QName"?
         type="QName"?
         element="QName"?>+
         from-spec?
      </variable>
   </variables>

   <correlationSets>?
      <correlationSet name="NCName" properties="QName-list" />+
   </correlationSets>

   <faultHandlers>?
      <!-- Note: There must be at least one faultHandler -->
```

```
      <catch faultName="QName"?
         faultVariable="BPELVariableName"?
         ( faultMessageType="QName" | faultElement="QName" )? >*
         activity
      </catch>
      <catchAll>?
         activity
      </catchAll>
   </faultHandlers>

   <eventHandlers>?
      <!-- Note: There must be at least one onEvent or onAlarm. -->
      <onEvent partnerLink="NCName"
         portType="QName"?
         operation="NCName"
         ( messageType="QName" | element="QName" )?
         variable="BPELVariableName"?
         messageExchange="NCName"?>*
         <correlations>?
            <correlation set="NCName" initiate="yes|join|no"? />+
         </correlations>
         <fromParts>?
            <fromPart part="NCName" toVariable="BPELVariableName" />+
         </fromParts>
         <scope ...>...</scope>
      </onEvent>
      <onAlarm>*
         <!-- Note: There must be at least one expression. -->
         (
         <for expressionLanguage="anyURI"?>duration-expr</for>
         |
         <until expressionLanguage="anyURI"?>deadline-expr</until>
         )?
         <repeatEvery expressionLanguage="anyURI"?>
            duration-expr
         </repeatEvery>?
         <scope ...>...</scope>
      </onAlarm>
   </eventHandlers>
   activity
</process>
```

最高级别的属性如下：

- queryLanguage。这个属性指定了赋值程序中选择节点的查询语言。属性的默认值是：“urn：oasis：name：tc：wsbpel：2.0：sublang：xpath1.0”，表示 WS-BPEL2.0 中[XPath 1.0]的用法。
- expressionLanguege。这个属性指定了<process>中指定的表达式语言。默认属性值是：”urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0”，表示 WS-BPEL2.0 中[XPath 1.0]的用法。
- suppressJoinFailure。这个属性决定是否抑制流程中的所有活动的 joinFailure 故障。这个程序级属性的作用是活动使用属性的不同值时可以被忽略。属性的缺省值是“no”。当活动没有指定这个属性时，它从最靠近的附属活动或者无附属活动指定这个属性时则从<process>继承值。
- exitOnStandardFault。如果这个属性值被设为“yes”，则程序必须立即退出就像遇到一个 WS-BPEL 标准故障除了 bpel：joinFailure，<exit>活动到达一样。如果属性值被设为“no”，则程序可以使用故障处理器来处理标准故障。属性的缺省值是“no”。当<scope>没有指定这个属性时，它从附属的<scope>或者<process>继承值。

[SA00003]<scope>或者<process>的exitOnStandardFault属性值被设为“no”，则故障处理器明确标记WS-BPEL的标准故障不能在那个作用域中使用。如果程序定义违反了这个条件，肯定会被静态分析找到并且被一致实现拒绝。

- 抽象程序的语义拥有其独特的目标名称空间。额外的最高层属性会为抽象流程定义。

在<process>元素上的 queryLanguage 和 expressionLanguege 的属性值是全局缺省的并且可以由指定的构造器忽略，比如<while>活动的<condition>，稍后会在本规范中定义。另外 queryLanguage 属性在 WSDL 中定义 WS-BPEL<vprop:propertyAlias>的用法也是可行的。

- 在与 WSDLs 相关联的 WS-BPEL 自身程序定义中或者 WS-BPEL 属性定义中通过 queryLanguage 或 expressionLanguege 属性来引用的语言的静态终止以及
- [SA00004]如果任何被引用语言不被WS-BPEL处理器支持，则处理器必须拒绝提交WS-BPEL程序定义。

请注意：<documentation>结构可能被正式的添加到所有的 WS-BPEL 构造器中同人工文档一起来注释流程定义。<documentation>结构例子可以在先前的章节找到。有关<documentation>的详细描述会在 5.3 扩展性语言 中提供。

每个业务程序有一个主要的活动

一个 WS-BPEL 活动可以是以下的任何一个：

- <receive>
- <reply>
- <invoke>
- <assign>
- <throw>
- <exit>

- <wait>
- <empty>
- <sequence>
- <if>
- <while>
- <repeatUntil>
- <forEach>
- <pick>
- <flow>
- <scope>
- <compensate>
- <compensateScope>
- <rethrow>
- <validate>
- <extensionActivity>

每个这些元素的语义在下面的段落中说明。

<receive>活动允许业务程序等待一个匹配的消息到达。当消息到达时<receive>活动完成。<receive>活动的portType属性是可选的。[SA00005]如果portType属性可读，则portType的属性值必须匹配由指定的partnerLink和由活动明确指定的role（见下章的partnerLink描述）组合默认的portType值。可选的messageExchange属性用于关联<reply>活动和<receive>活动。

```
<receive partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   variable="BPELVariableName"?
   createInstance="yes|no"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</receive>
```

<reply>活动允许业务程序发送消息回答入站式消息活动（IMA）接收的消息，即<receive>,<onMessage>或者<onEvent>。一个 IMA 和一个<reply>的组合形成了一个 WSDL portType 程序的请求-响应操作。<reply>活动的 portType 属性是可选择的。如果 portType 属性可读，则 portType 的属性值必须匹配由指定的 partnerLink 和由活动明确指定的 role（见下章的 partnerLink 描述）组合默认的 portType 值。可选的 messageExchange 属性用于关联<reply>活动和 IMA。

```
<reply partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   variable="BPELVariableName"?
   faultName="QName"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
</reply>
```

〈invoke〉活动允许业务程序在伙伴提供的 portType 上调用单向或者请求-响应操作。在请求-响应情况下，响应被接收则 invoke 活动完成。〈invoke〉活动上的 portType 属性是可选择的。如果 portType 属性可读，则 portType 的属性值必须匹配由指定的 partnerLink 和由活动明确指定的 role（见下章的 partnerLink 描述）组合默认的 portType 值。

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"?
         pattern="request|response|request-response"? />+
   </correlations>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      faultMessageType="QName"?
      faultElement="QName"?>*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
   <compensationHandler>?
      activity
   </compensationHandler>
   <toParts>?
```

```
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</invoke>
```

<assign>活动是用新的数据更新变量值。<assign>构造可以包括任意数量的基本赋值，包括在其他名称空间下扩展定义的<copy>赋值元素或者数据更新操作。

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
   (
   <copy keepSrcElementName="yes|no"? ignoreMissingFromData="yes|no"?>
      from-spec
      to-spec
   </copy>
   |
   <extensionAssignOperation>
      assign-element-of-other-namespace
   </extensionAssignOperation>
   )+
</assign>
```

<validate>活动用于验证变量值通过和它们相关的 XML 和 WSDL 数据定义。构造拥有一个 variables 属性，指向正在被验证的变量。

```
<validate variables="BPELVariableNames" standard-attributes>
   standard-elements
</validate>
```

<throw>活动用于生成来自业务程序内部的故障。

```
<throw faultName="QName"
   faultVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
</throw>
```

<wait>活动用于等待给定的时间段或者直到某个时刻到来。必须指定一个截止时间条件。

```
<wait standard-attributes>
   standard-elements
   (
   <for expressionLanguage="anyURI"?>duration-expr</for>
```

```
   |
   <until expressionLanguage="anyURI"?>deadline-expr</until>
   )
</wait>
```

<empty>活动在业务程序中是一个空操作。它对于同步并发活动很有用，例如：

```
<empty standard-attributes>
   standard-elements
</empty>
```

<sequence>活动用于定义一组按词法顺序先后被执行的活动。

```
<sequence standard-attributes>
   standard-elements
   activity+
</sequence>
```

<if>活动用于从一套选择中精确选择执行某个活动。

```
<if standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
   activity
   <elseif>*
      <condition expressionLanguage="anyURI"?>bool-expr</condition>
      activity
   </elseif>
   <else>?
      activity
   </else>
</if>
```

<while>活动用于定义当<condition>为 true 时重复执行子活动。

```
<while standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
   activity
</while>
```

<repeatUntil>活动用于定义需要重复执行子活动直到指定的<condition>变为 true。子活动完成后检验<condition>。<repeatUntil>活动用于执行子活动至少一次。

```
<repeatUntil standard-attributes>
```

```
   standard-elements
   activity
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

<forEach>活动精确地重复它的子作用域活动 N+1 次，N=<finalCounterValue>-<startCounterValue>。如果 parallel=“yes”则<forEach>中附属<scope>活动的 N+1 个实例应该并行发生。实际上隐式流同 N+1 个<forEach>的<scope>活动的拷贝一起被动态创建。一个在<forEach>中的<completionCondition>可能允许<forEach>活动在没有执行或者完成所有指定分支的情况下结束。

```
<forEach counterName="BPELVariableName" parallel="yes|no"
   standard-attributes>
   standard-elements
   <startCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </finalCounterValue>
   <completionCondition>?
      <branches expressionLanguage="anyURI"?
         successfulBranchesOnly="yes|no"?>?
         unsigned-integer-expression
      </branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

<pick>活动用来等待若干可能的消息之一到达或者发生超时。当其中某个触发器发生时，相关联的子活动被执行。当子活动完成后<pick>活动完成。

<onMessage>活动的 portType 属性是可选的。如果 portType 属性可读，则 portType 的属性值必须匹配指定的 partnerLink 和活动明确指定的 role（见下章的 partnerLink 描述）组合默认的 portType 值。可选的 messageExchange 属性用于关联<reply>活动和<onMessage>事件。

```
<pick createInstance="yes|no"? standard-attributes>
   standard-elements
   <onMessage partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      variable="BPELVariableName"?
      messageExchange="NCName"?>+
      <correlations>?
```

```
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      activity
   </onMessage>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )
      activity
   </onAlarm>
</pick>
```

<flow>活动用来指定并发执行的一个或多个活动。<flow>里的<links>用来定义嵌套子活动间的精确控制依赖。

```
<flow standard-attributes>
   standard-elements
   <links>?
      <link name="NCName" />+
   </links>
   activity+
</flow>
```

<scope>活动通过与其相关联的<partnerLinks>,<messageExchange>,<variables>,<correlationSets>,<faultHandlers>,<compensationHandlers>, <terminationHandler>和<eventHandlers>定义嵌套活动。

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
   standard-attributes>
   standard-elements
   <partnerLinks>?
      ... see above under <process> for syntax ...
   </partnerLinks>
   <messageExchanges>?
      ... see above under <process> for syntax ...
   </messageExchanges>
   <variables>?
      ... see above under <process> for syntax ...
   </variables>
   <correlationSets>?
```

```
      ... see above under <process> for syntax ...
   </correlationSets>
   <faultHandlers>?
      ... see above under <process> for syntax ...
   </faultHandlers>
   <compensationHandler>?
      ...
   </compensationHandler>
   <terminationHandler>?
      ...
   </terminationHandler>
   <eventHandlers>?
      ... see above under <process> for syntax ...
   </eventHandlers>
   activity
</scope>
```

<compensateScope>活动用于在指定的已经成功完成的内部作用域中启动赔偿。[SA00007]这个活动必须从故障处理器，另外的赔偿处理器或者终止处理器内部调用。

```
<compensateScope target="NCName" standard-attributes>
   standard-elements
</compensateScope>
```

<compensate>活动用于默认请求在所有的已经成功完成的内层作用域上启动赔偿。[SA00008] 这个活动必须从故障处理器，另外的赔偿处理器或者终止处理器内部调用。

```
<compensate standard-attributes>
   standard-elements
</compensate>
```

<exit>活动用于立即结束包含<exit>活动的程序里的业务程序实例。

```
<exit standard-attributes>
   standard-elements
</exit>
```

<rethrow>活动用于重新抛出最初被立即附属故障处理器捕获的故障。[SA00006]<rethrow>活动必须在故障处理器内（比如<catch>和<catchall>元素）调用。

```
<rethrow standard-attributes>
   standard-elements
</rethrow>
```

<extensionActivity>元素通过引入新的活动类型来扩充 WS-BPEL。<extensionActivity>元素的内容必须是单个能使 WS-BPEL 的标准属性和标准元素可用的元素。

```
<extensionActivity>
   <anyElementQName standard-attributes>
      standard-elements
   </anyElementQName>
</extensionActivity>
```

上面引用的“standard-attributes”是：

```
name="NCName"? suppressJoinFailure="yes|no"?
```

默认值如下：

- Name:没有缺省值（即默认是无名的）
- suppressJoinFailure：当活动没有指定这个属性，它从最靠近的附属活动集成值，如没有附属活动指定时则从程序继承值。

上面引用的“standard-elements”为：

```
<targets>?
   <joinCondition expressionLanguage="anyURI"?>?
      bool-expr
   </joinCondition>
   <target linkName="NCName" />+
</targets>
<sources>?
   <source linkName="NCName">+
      <transitionCondition expressionLanguage="anyURI"?>?
         bool-expr
      </transitionCondition>
   </source>
</sources>
```

5.3 语言扩展性

为了支持扩展性，WS-BPEL 允许名称空间限定的属性出现在任何 WS-BPEL 元素上，也允许其它名称空间的元素出现在 WS-BPEL 定义的元素中。这在 WS-BPEL 的 XML Schema 规范中是允许的。

扩展要么是强制的要么是可选的（见章节 14.扩展声明）。[SA00009]在WS-BPEL实现不支持强制性扩展的情况下，程序定义必须被拒绝。不被WS-BPEL支持的可选择扩展必须被忽略。

另外，WS-BPEL 提供两种显示扩展构造器：<extensionAssignOperation>和<extensionActivity>。这些构造器的具体规则在章节 8.4 赋值和 10.9 添加新的活动类型-扩展活动 会介绍。

扩展禁止同 WS-BPEL 规范定义的元素或属性的语义相抵触。

在 WS-BPEL 构造器中，扩展允许使用 WSDL 定义，比如<partnerLink>，<role>，<vprop：property>和<vprop：propertyAlias>。WS-BPEL 构造器扩展的句法样式和语义规则也同样适用于那些扩展。因为 WSDL 定义可以及物地被 WS-BPEL 程序引用，WS-BPEL 程序的扩展声明命令也适用于 WSDL 定义里的 WS-BPEL 构造器使用的扩展（见章节 14.扩展声明）。

可选择<documentation>构造器适用于任何 WS-BPEL 可扩展的构造器。典型地，<documentation>的内容是人工注释标记的。那些注释的示例类型有：纯文本，HTML 和 XHTML。工具实现通过使用通用的 WS-BPEL 可扩展性机制来指定了应该由其他名称空间的元素和属性来添加的信息（比如图形设计详细资料）。

5.4 文档链接

WS-BPEL 程序定义依赖 XML Schema 和 WSDL 1.1 的数据类型和服务接口定义。程序定义也依赖其他构造器比如 WSDL 文档中使用 WSDL 1.1 语言扩展特征的伙伴链接类型，变量属性和属性别名（本规范稍后定义）等。

```
<import namespace="anyURI"?
   location="anyURI"?
   importType="anyURI" />*
```

WS-BPEL 程序中的<import>元素用来声明外部 XML Schema 或 WSDL 定义上的依赖关系。许多<import>元素可以以<process>元素的子集出现。每个<import>元素包括一个强制性的和两个可选择的属性。

- namespace。namespace属性指定纯粹的识别引入定义的URI。属性是可选择的。没有namespace属性的引入元素指出没有名称空间资格的外部定义正在使用中。[SA00011]如果名称空间被指定则引入定义必须在那个名称空间内。[SA00012]如果没有名称空间被指定则引入定义禁止包含目标名称空间的规范。如果上述情况都未出现则程序定义必须被一致的WS-BPEL实现拒绝。名称空间 http://www.w3.org/2001/XMLSchema是被暗中引入的。请注意不管怎样，没有替 http://www.w3.org/2001/XMLSchema定义隐式的XML Namespace 。
- location。location 属性包含了 URI 指示的含有相关定义的文档位置。location URI 可能是相对 URI，遵循 URI base (XML Base and RFC 2396)确定的惯例规则。location 属性是可选的。没有 location 属性的<import>元素指出了由程序使用的外部定义但

是没有声明哪里可以找到这些定义。location 属性是一个提示并且 WS-BPEL 处理器不需要检索从指定位置引入的文件。

- importType。强制的importType属性识别了正在通过纯URI被引入的文档类型，纯URI识别了文档使用的编码语言。[SA00013]当引入XML Schema 1.0 文档时importType的属性值必须设为 http://www.w3.org/2001/XMLSchema，当引入WSDL 1.1 文档时必须设为 http://schemas.xmlsoap.org/wsdl/。其他importType URI值可以在这里被使用。

请注意根据这些规则，只包括 importType 属性不包含 namespace 和 location 属性的<import>元素是允许的。这样一个<import>元素指出了没有名称空间资格的外部定义正在使用中，并且没有说明可以找到那些定义的地址。

[SA00010]WS-BPEL程序定义必须引入使用的所有XML Schema和WSDL定义。这包含所有XML Schema类型和元素定义、WSDL端口类型和消息类型，以及程序使用的<vrop:property>和<vprop:propertyAlias>定义。[SA00053], [SA00054]WS-BPEL处理器必须验证在它们各自的WSDL消息定义里找到的<vprop:propertyAlias>,<from>,<to>,<fromPart>以及<toPart>部分引用的所有消息。为了支持来自名称空间生成的多个文档的定义使用，WS-BPEL程序可以包含多于 1 个相同的namespace和importType引入声明,假设那些声明包含不同的location值。<import>元素概念上是不可排序的。[SA00014]如果引入的文档包含引入程序定义（可能被导致的，比如当XSD重定义机制被使用时）使用的成分相冲突的定义，则WS-BPEL程序定义必须被拒绝。

由 WS-BPEL 程序定义引入的 WSDL 文档的类型部分中定义的 Schema 定义被认为是可以高效引入自身并且对程序定义 XML Schema 变量可用。不管怎样，由引入文档（或名称空间）引入的文档（或名称空间）禁止被 WS-BPEL 处理器直接引入。具体的说，就是如果一个外部项目被 WS-BPEL 程序使用，则定义那个项目的文档（或名称空间定义）禁止被程序直接引入；请注意这个要求没有限制被引入的文档本身引入其他文档或者名称空间的能力。下面的例子阐明了某些涉及到缺乏引入传递性的问题。

假设文档 D1 定义了一个类型称为 d1：Type。不管怎样，d1：Type 的定义可以依赖文档 D2 定义的 d2：Type。D1 可以包含 D2 的引入因此使得在 d1：Type 的定义内 d2：Type 的定义可用。如果一个 WS-BPEL 程序涉及到 d1：Type，则它必须引入文档 D1。通过引入文档 D1，WS-BPEL 程序可以合法的引用 d1：Type。但是即使 D1 引入了 D2，WS-BPEL 程序不能引用 d2：Type。这是因为 WS-BPEL 不支持引入的传递性。请注意，不管怎样，D1 仍然可以引入 D2 并且 d1：Type 在它的定义里任然可以使用 d2：Type。为了让 WS-BPEL 程序允许引用 d2：Type，WS-BPEL 程序直接引入 D2 文档是必要的。

5.5 可执行业务程序的生命周期

正如简介里提到的，由 WSDL 直接支持的交互作用模型本质上是一个无状态的请求-响应或者单向非约束交互的客户端服务模型。WS-BPEL 建立在 WSDL 之上通过假设所有业务程序的外部交互通过 Web 服务操作发生。不管怎样，WS-BPEL 业务流程表现了有状态的长

时运行的交互，每个交互有一个开始和结束，并且在生命周期内定义行为。例如在一个供应链中，销售者的业务程序可能提供一个服务-通过接受因输入消息而得到的订购单来启动交互，如果请求被执行则返回一个确认给买家。稍后它可能发送进一步的消息给买家，比如通知单和发票等。销售者的业务程序记得每个订购单的交互状态，从而将它们同其他类似交互区别开来。这是必要的，因为买家可能同时和同一个卖家执行多次订购业务。简而言之，WS-BPEL 业务程序定义可以被认为是创建业务程序实例的模板。

WS-BPEL 程序实例的创建总是隐式的，接收消息的活动（即<receive>活动和<pick>活动）可以通过注释来指出活动导致的新的被创建业务程序实例的发生。这个通过设置活动的creatInstance 属性为“yes”来完成。当活动接收到消息时，业务程序的实例如果不存在的话则被创建（见章节 10.4 提供 Web 服务操作-Receive 和 Reply 及章节 11.5 选择事务处理-Pick）。

开始活动是<receive>或<pick>活动的属性creatInstance=“yes”。[SA00015]每个可执行的业务程序必须包括最少一个开始活动（见章节 10.4 提供Web服务操作-Receive和Reply获得更多有关开始活动的详细内容）。

如果一个程序里包含多于一个的开始活动并且这些开始活动包含<correlationSets>则它们必须共享至少一个公用的<correlation>.(见章节 9.2 声明和使用相关装置中的例子)

如果一个程序只包含一个开始活动则<correlationSets>的使用不是强制的。它包含了有多个 <onMessage> 分支的 pick；每个分支可以使用不同的 <correlationSets> 或者无 <correlationSets>。

业务程序实例要么正常终止，要么非正常的终止。当程序的主要活动和所有事件处理器实例没有任何故障的完成则程序正常结束。当出现以下情况时，程序异常结束：

- 程序级（显示或默认的）故障处理器完成（没有传送任何故障）
- 程序级故障处理器自身故障的执行（这个特殊情况的效果同<exit>活动类似）
- 程序实例通过<exit>活动显示结束（见章节 10.10 立即终止程序-Exit）

5.6 再访初始例子

在 5.1 节的订单程序例子中。初始化例子，程序主要活动的结构由外部<sequence>元素定义，里面包含的三个活动按顺序执行。客户请求被接收(<receive>元素)后，开始处理（里面的<flow>部分激活并发行为），最后带有请求的最后确认状态的回复消息被返给客户（<reply>）。请注意<receive>和<reply>元素分别匹配客户调用的”sendPurchaseOrder”操作的<input>和<output>消息，同时程序从请求被接受的时间直到响应被返回的时间（reply）里在那些表示响应客户请求行为的元素中执行活动。

这个处理发生在由三个并发<sequence>活动组成的<flow>元素内部。三个并发序列里的活动之间的同步依赖通过使用<links>连接表示。<links>在<flow>内部定义并且用于将资源活动连接到目标活动。请注意每个活动如资源或者目标<link>一样使用嵌套的<source>和<target>元素来声明自身。当缺乏<links>时，<flow>内部直接嵌套的活动继续并行进行。在

例子中，不管怎样，两个<link>的存在说明了活动间的控制依赖内部执行了每个序列段。例如，当价格计算可以在接收到请求后立即启动，运送价格只能在得到运送信息后添加到发票，这个依赖通过<link>（名为“ship-to-invoice”）表现，第一个<link>将运送提供者（“requestShipping”）的访问和发送运送信息给价格计算服务（“sendShippingPrice”）连接起来。同样地，运送调度信息只能在接收到运送服务后被送给生产调度服务，因此需要第二个<linlk>（“ship-to-scheduling”）。

不同活动间通过共享变量实现数据共享，例如，两个<variable>”shippingInfo”和”shippingSchedule”。

某些操作可能返回故障，正如它们的 WSDL 定义里定义的一样。简单的说，假定两个操作返回相同的故障（“cannotCompleteOrder”）。当故障发生时，如<faultHandlers>部分定义的应该正常处理终止并且将控制转移给相应的故障处理器。在这个例子中故障处理器使用<reply>元素返回故障给客户（请注意<reply>元素中的 faultName 属性）。

最后，请注意赋值活动是怎样在数据变量间传输信息的。这个例子中展示了简单的赋值即将一个来自资源变量的消息部分传送给目标变量的消息部分，但是更复杂的赋值形式也是可能的。

6. 伙伴链接类型，伙伴链接和端点引用

一个 WS-BPEL 的重要案例是描述每个企业的业务流程通过 Web 服务接口实现跨企业业务交互。因此，WS-BPEL 提供了模拟伙伴流程间请求关系的功能。WSDL 已经描述了抽象层和具体层的伙伴提供的服务的功能性。业务流程与伙伴的关系通常是对等的，这需要在服务级别上有双向的相关性。换言之，伙伴表示由业务流程提供的服务的消费者，同时，对于业务流程来说，伙伴又表示服务的提供者。这个尤其是在基于单向操作的交互胜于请求-响应操作的情况时。<partnerLinks>的概念用于直接模拟伙伴间对等模式会话关系。<partnerLink>通过定义双向交互时使用的 portType 来定义同伙伴的关系类型。不管怎样，实际的伙伴服务可能在程序里被动态地定义。WS-BPEL 使用端点引用概念，作为一个服务引用容器<sref:service-ref>，表现数据要求描述伙伴服务端点。

服务引用容器<sref:service-ref>的说明避免了发明私有的 Web 服务端点引用 WS-BPEL 机制。它也提供了不同版本的服务引用的或者 WS-BPEL 里使用的端点寻址方案的可插性。

6.1 伙伴链接类型

<partnerLinkType>通过定义会话里每个服务扮演的角色以及指定由每个服务提供的 portType 接收会话上下文里的消息来刻画了两个服务之间的会话关系。每个<role>确切指定了一个 WSDL portType。下面的例子说明了<partnerLinkType>声明的基本语法。

```
<plnk:partnerLinkType name="BuyerSellerLink">
   <plnk:role name="Buyer" portType="buy:BuyerPortType" />
   <plnk:role name="Seller" portType="sell:SellerPortType" />
</plnk:partnerLinkType>
```

WSDL 1.1 的扩展机制用于将<partnerLinkType>定义为新类型作为<wsdl:definitions>元素的后继子元素的取代物。这允许重用 WSDL 目标名称空间规范以及引入机制来引入 portType 定义。<PartnerLinkType>定义可以人工分离独立于任一服务的 WSDL 文档。二选一地，<partnerLinkType>定义可以被置于不同的被定义角色的 portTypes 的 WSDL 文档中。

定义<PartnerLinkType>的语法为：

```
<wsdl:definitions name="NCName" targetNamespace="anyURI" ...>
   ...
   <plnk:partnerLinkType name="NCName">
      <plnk:role name="NCName" portType="QName" />
      <plnk:role name="NCName" portType="QName" />?
   </plnk:partnerLinkType>
   ...
</wsdl:definitions>
```

这在由 WSDL 文档元素的 targetNamespace 属性值表示的名称空间里定义了<partnerLinkType>。<role>里 portType 标识由根据 WSDL 规范规则使用的 Qnames 来引用。

请注意在某些情况下，定义包含一个而不是两个<role>的<partnerLinkType>是很有意义的。那样定义了一个伙伴不用放置任何需求就可以连接其他伙伴性能的伙伴链接。

本规范中可以找到在不同业务程序里<partnerLinkType>声明的例子。

6.2 伙伴链接

在 WS-BPEL 中，业务程序交互的服务如同伙伴链接一样被模拟。每个<partnerLink>由 partnerLinkType 标记。不止一个<partnerLink>可以被相同的 partnerLinkType 标记。比如，某个获取程序可能为了它的事务使用不止一个售主，但是对所有的售主可能使用相同的 partnerLinkType。

```
<partnerLinks>
   <partnerLink name="NCName"
      partnerLinkType="QName"
      myRole="NCName"?
      partnerRole="NCName"?
      initializePartnerRole="yes|no"? />+
</partnerLinks>
```

每个<partnerLink>都是有命名的，这个名字通过<partnerLink>用于所有的服务交互。这是很关键的，比如说让同类型同时请求的不同<partnerLink>与响应相关联。（见章节 10.3 调用 Web 服务操作-Invoke 和章节 10.4 提供 Web 服务操作-Receive 和 Reply）

在<partnerLink>里，业务程序自身的角色通过属性myRole表示并且伙伴角色通过属性partnerRole表示。当PartnerLinkType只有一个角色时，这些属性之一会被适当的忽略。[SA00016]请注意<partnerLink>必须指定myRole或partnerRole，或者两者都指定。这个语法约束必须静态强制。

<partnerLink>声明指定了 WS-BPEL 程序必须雇用它的行为的关系。为了通过<partnerLink>来利用操作，<partnerLink>的绑定和通信数据，包括端点引用（EPR）必须是可用的（见章节 10.3 调用 Web 服务操作-Invoke）。有关<partnerLink>的信息可以被看做为业务程序部署的一部分。这是在 WS-BPEL 规范的作用域之外的。伙伴链接类型确定了两种 Web 服务的 WSDL 端口类型间的关系。伙伴链接类型的目的是保持程序里关系的清晰，以使有多个伙伴的程序更容易被了解。本规范中没有其他的伙伴链接类型暗指的语法或者语义关系。动态绑定伙伴链接也是可能的。WS-BPEL 通过端点引用的赋值来提供了这种机制（见章节 8.4 赋值）。由于伙伴可能是有状态的，服务端点信息可能需要用具体实例信息来扩展。

initializePartnerRole属性指定WS-BPEL处理器是否需要初始化<partnerLinl>的partnerRole值。属性初始化后对partnerRole的值没有影响。[SA00017]initializePartnerRole属性禁止被用

于没有伙伴角色的伙伴链接；这个约束必须被静态强制。如果initializePartnerRole属性值设为“yes”则WS-BPEL处理器必须在WS-BPEL程序初次利用EPR前初始化partnerRole的EPR。这里有当<invoke>活动使用EPR时的例子。如果initializePartnerRole属性设为“no”则WS-BPEL处理器禁止在WS-BPEL程序初次利用EPR前初始化partnerRole的EPR。如果initializePartnerRole属性被省略则程序角色可以被WS-BPEL处理器初始化。

当 initializePartnerRole=“yes”时，partnerRole 初始状态里使用的 EPR 值典型地指定为 WS-BPEL 部署或者执行环境配置的一部分。因此，initializePartnerRole 属性可能被当作程序部署合同的一部分。

<partnerLink>可以在<process>或者<scope>元素里声明。[SA00018]在同一个直接附属作用域中<partnerLink>的名字必须必须是唯一的。这个约束必须静态地强制。<partnerLink>的使用权遵循公用词法域规则。<partnerLink>的生命周期同作用域声明的<partnerLink>生命周期一样。<partnerLink>的初始绑定信息可以看做业务程序部署的一部分，不管它是否在<process>或<scope>元素上声明。

6.3 端点引用

WSDL 在端口类型和端口间有重要的区别。端口类型通过使用抽象消息定义抽象函数性。端口提供实际访问信息，包含通信服务端口和（通过使用扩展元素的）其他部署相关信息比如加密的公钥。绑定提供了两者之间的粘合剂。当服务使用者必须静态从属于端口类型定义抽象接口时，某些包含端口定义的信息典型地被发现并且被动态使用。

端口引用的主要用法是像指定端口数据服务的动态通信机制一样提供服务。端口引用使得 WS-BPEL 里动态地选择特殊类型的服务提供者以及调用它们的操作是可能的。WS-BPEL 提供了关联消息和服务的有状态实例的通用机制，并且可以充足执行中枢实例端口信息的端口引用。不管怎样，在端口引用自身携带额外的实例辨识标记通常是必要的。

同<partnerLink>的 partnerRole 和 myRole 相关联的端口引用被看做服务引用容器（<sref:service-ref>）。这个容器作为交换实际的端口引用值的封装被使用。这个实际样式类似于那些表达式语言，也被看做是内容开放式模型，例如：

```
<sref:service-ref reference-scheme="http://example.org">
   <foo:barEPR xmlns:foo="http://example.org">...</foo:barEPR>
</sref:service-ref>
```

<sref:service-ref>有一个名为 reference-schema 的可选择属性用来表示端点服务的引用解释模式的 URI，是<sref:service-ref>的子元素。

reference-schema 的 URI 和<sref:service-ref>子元素的名称空间的 URI 不一定相同。当<sref:service-ref>的子元素模糊不清时，应该使用可选择的 reference-schema 属性。这个属性提供了消除歧义的方法。比如，如果 wsdl：service 被作为端口引用使用，可能发生不同的 wsdl：service 元素的处理。

如果属性没有指定，包装里的内容元素的名称空间 URI 必须用于终止端口服务的引用模式。

如果指定了属性，URI 应该被作为被处理过的包装里的端口服务的引用模式以及内容元素。

当 WS-BPEL 实现解释 reference-scheme 属性和内容元素的组合或者只是单独的内容元素失败时，标准异常“unsupportedReference”必须被抛出。

<sref:service-ref>元素不总是被 WS-BPEL 程序定义暴露的。比如，在从 partnerLink-A 的 myRole 的端点引用到 partnerLink-B 的 partnerRole 的端点引用的赋值中，它没有被暴露。相反的是，在来自基于变量的 messageType 或元素的赋值，或者来自常量<sref:service-ref>的赋值中被暴露。

7. 变量属性

7.1 动机

7.1.1 消息属性的动机

消息数据概念上由两部分组成：应用程序数据和协议有关数据，协议可以是提供更高质量服务的业务协议或者基础协议。有关业务协议数据的例子在<correlationSets>中使用的相关标记（见章节 9.2 声明以及使用相关装置）。基础协议的例子就是安全性、事务以及可靠通信协议。业务协议数据通常被嵌入在可视应用程序消息部分，而基础协议几乎总给消息类型添加额外的隐式部分来表现协议头部同应用程序数据是分离的。这样一个内隐部分经常被称作消息上下文因为它们涉及到安全性上下文、事务上下文以及其他类似交互中间件的上下文。业务流程可能需要存取和操作两种协议相关数据。消息属性的概念被定义为在消息里命名和表现数据元素的通用方法，不管是否在可视应用程序或消息上下文中。为了对基础协议的服务描述有全方位的统计，定义服务保险单、端点属性以及消息上下文的标记是必要的。这个工作是在 WS-BPEL 工作域之外的。这里定义的消息属性能充分覆盖由内隐部分组成的消息上下文，但是本规范中的用法集中了嵌入在可视应用程序用于抽象和可执行业务流程定义数据的属性，用于抽象和可执行业务流程定义。

7.1.2 变量属性的动机

消息属性是其他类机制的实例，<viriable>属性。WS-BPEL 中的所有变量可以定义在它们上的属性。属性作为一种把 WS-BPEL 程序的逻辑从特殊变量的定义细节中分离出来的途径，在没有消息的变量上很有用。使用属性，WS-BPEL 程序可以在某个适当的位置分离它的初始化可变逻辑，之后可以在<variable>上设置和得到属性并且操作。如果<variable>的定义稍后改变了 WS-BPEL 程序定义的余下部分，则变量操作可以保持不变。

7.2 定义属性

<vprop:property>定义创建了 WS-BPEL 程序定义的唯一名称并且将其和 XML Schema 类型相连。意图指定名称来拥有一个越过类型本身的语义有效性。比如，序列号可以是整数，但是整数类型并不代表序列号，然而一个命名的序列号属性可以。属性可以访问变量的任意部分。

<vprop：property>在 WS-BPEL 里的典型应用是为带有消息的服务实例的相关性命名一个标记。比如，社会安全号可能在有关纳税问题的长时运行的多处理业务程序里用于识别个体纳税人。社会安全号可以有多种消息类型，但是在与纳税相关的程序上下文里只能是一个指定的有效数作为纳税人 ID。因此名字被给出通过定义<vporp:property>，如下所示：

```
<wsdl:definitions name="properties"
   targetNamespace="http://example.com/properties.wsdl"
   xmlns:tns="http://example.com/properties.wsdl"
   xmlns:txtyp="http://example.com/taxTypes.xsd"
   xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <!-- import schema taxTypes.xsd -->

   <!-- define a correlation property -->
   <vprop:property name="taxpayerNumber" type="txtyp:SSN" />
   ...

</wsdl:definitions>
```

在相关性中，属性名称在所有的应用中必须有程序级的有效性。属性比如价格，风险，响应反应时间等被用于业务程序中的条件性行为，有类似的有效性。它们将被映射到多个消息中，并且因此在相关属性中需要被命名。

甚至在普通 XML 类型的 WS-BPEL 变量中，属性名称应该保持它的一般类属。名称用于识别某一类型值，经常同暗指的语义一起。任何属性可用的变量因而期望提供不仅符合属性定义语法而且符合语义的值。

WSDL 可扩展机制被用于定义属性。目标名称空间以及其他可用的 WSDL 特征对于它们也是可用的。

属性定义的语义是 WSDL 中一种新型定义，如下：

```
<wsdl:definitions name="NCName">
   <vprop:property name="NCName" type="QName"? element="QName"? />
   ...
</wsdl:definitions>
```

[SA00019]类型或者元素属性必须被显示其中之一。业务协议中使用的属性典型地嵌入在可视应用程序消息数据中。

7.3 定义属性别名

介绍别名的使用概念用于映射属性到指定消息部分或者变量值的域中。属性名称成为消息部分及/或位置的别名，并且可以被用于表达式或者赋值中。WSDL 消息定义例子如下：

```
<wsdl:definitions name="messages"
   targetNamespace="http://example.com/taxMessages.wsdl"
   xmlns:txtyp="http://example.com/taxTypes.xsd"
```

```
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <!-- define a WSDL application message -->
   <wsdl:message name="taxpayerInfoMsg">
      <wsdl:part name="identification"
         element="txtyp:taxPayerInfoElem" />
   </wsdl:message>
   ...

</wsdl:definitions>
```

指定消息域的属性定义和它的位置在接下的 WSDL 片段显示：

```
<wsdl:definitions name="properties"
   targetNamespace="http://example.com/properties.wsdl"
   xmlns:tns="http://example.com/properties.wsdl"
   xmlns:txtyp="http://example.com/taxTypes.xsd"
   xmlns:txmsg="http://example.com/taxMessages.wsdl" ...>

   <!-- define a correlation property -->
   <vprop:property name="taxpayerNumber" type="txtyp:SSN" />
   ...

   <vprop:propertyAlias propertyName="tns:taxpayerNumber"
      messageType="txmsg:taxpayerInfoMsg" part="identification">
      <vprop:query>txtyp:socialsecnumber</vprop:query>
   </vprop:propertyAlias>

   <vprop:propertyAlias propertyName="tns:taxpayerNumber"
      element="txtyp:taxPayerInfoElem">
      <vprop:query>txtyp:socialsecnumber</vprop:query>
   </vprop:propertyAlias>

</wsdl:definitions>
```

第一个<vprop:propertyAlias>定义一个有名属性 tns：taxpayerNumber 作为消息类型 txmsg：taxpayerInfoMsg 身份识别的定位。

第二个<vprop:propertyAlias>提供了第二个相同定义 tns：taxpayerNumber，但这次是作为元素 txtyp：taxPayerInfoElem 内部定位的别名。

两个别名的存在说明了不仅可以从持有 txmsg：taxpayerInfo 的消息中也可以从使用 txtyp：taxPayerInfoElem 的元素中检索社会安全号。

<vprop:propertyAlias>定义的语法是：

```
<wsdl:definitions name="NCName" ...>

   <vprop:propertyAlias propertyName="QName"
      messageType="QName"?
      part="NCName"?
      type="QName"?
      element="QName"?>
      <vprop:query queryLanguage="anyURI"?>?
         queryContent
      </vprop:query>
   </vprop:propertyAlias>
   ...

</wsdl:definitions>
```

messageType 和 part 属性的解释，同<query>元素的解释一样在拷贝赋值中同 from-spec 相对应（见章节 8.4 赋值）。唯一一个异常是<vprop:propertyAlias>里<query>元素的 queryLanguage 属性的缺省值是 urn：oasis：names：tc：wsbpel：2.0：sublang：xpath1.0.

[SA00020]<vprop:propertyAlias>元素必须使用下面三种属性组合之一：

- `messageType` and `part`,
- `type` or
- `element`.

如果<vprop:propertyAlias>定义使用了 messageType/part 组合，则属性必须在所有 WS-BPEL 变量上可用，其中变量声明的 messageType QName 与<vprop:propertyAlias>相同。part 属性和<query>元素的应用同 WS-BPEL messageType 变量不同，不管是设置或者获得属性变量，part 属性和<query>元素都用于<copy>赋值中第一个 from 和 to 规范。

如果<vprop:propertyAlias>定义使用了 type 属性则属性必须在所有 WS-BPEL 变量上可用，其中变量声明的 type QName 与<vprop:propertyAlias>相同。查询和 WS-BPEL 变量不同，不管是设置还是获得属性变量，查询用于拷贝赋值中第一个 from 和 to 规范。

如果<vprop:propertyAlias>定义使用了 element 属性则属性必须在所有 WS-BPEL 变量上可用，其中变量声明的 element QName 与<vprop:propertyAlias>相同。查询和 WS-BPEL 变量不同，不管是设置还是获得属性变量，查询用于拷贝赋值中第一个 from 和 to 规范。

使用与上面相同的例子“`tns:taxpayerNumber`”，消息类型为 `txmsg:taxpayerInfoMsg` 的消息变量“`myTaxPayerInfoMsg`”

```
   <from variable="myTaxPayerInfoMsg" property="tns:taxpayerNumber" />
```

以及

```
<from>$myTaxPayerInfoMsg.identification/txtyp:socialsecnumber</from>
```

有相同的输出（见章节 8.4 赋值 的详细内容）

[SA00022]WS-BPEL程序定义如果为同一个属性名称和WS-BPEL变量类型定义了两个或两个以上属性别名，则禁止被处理。比如，替属性`tns:taxpayerNumber`和消息类型`txmsg:taxpayerInfoMsg`定义两个属性别名是不合法的。同样的逻辑也禁止相同的`element` QName和属性命名值或者相同的type QName和属性命名值有两个属性别名。

[SA00021]静态分析必须阻止同变量类型相关的属性别名在WS-BPEL程序直接引入的WSDL定义中无法查找的属性应用。正如在章节 8.数据处理 和 章节 9.相关性 中所描述的，WS-BPEL中的属性应用包括`<correlationSets>`, `getVariableProperty`函数以及赋值活动拷贝`<from>`和`<to>`属性格式。

8. 数据处理

业务流程指定有状态的交互调用伙伴间的消息交换。业务程序状态包括像在业务逻辑和送给伙伴的组装消息中使用的中间数据交换的消息。业务程序的状态维护要求变量的使用。此外，状态数据源需要被提取和组装来控制程序行为，需要数据表达式。最后，状态更新需要赋值标记。WS-BPEL 提供 XML 数据类型和 WSDL 消息类型的特征。在这些区域的 XML 规范仍然是展开的，并且考虑到未来的规范为查询和表达式语言使用程序级属性。

可执行的和抽象流程允许充分使用数据选择和赋值。可执行流程不允许使用不透明的表达式，而抽象流程允许使用它们来隐藏行为。详细区别会在下面的章节说明。

8.1 变量

变量提供了持有消息的方法，组成业务程序状态的一部分。被持有的消息经常是从伙伴那里接收到的或者即将被送给伙伴的。变量也可以持有保持相关程序状态需要的数据并且不会同伙伴交换。

WS-BPEL 使用三种类型的变量声明：WSDL 消息类型，XML Schena 类型（简单或复杂的）以及 XML Schema 元素。<variables>声明的语法如下：

```
<variables>
   <variable name="BPELVariableName"
      messageType="QName"?
      type="QName"?
      element="QName"?>+
      from-spec?
   </variable>
</variables>
```

<variable>声明的例子使用了 WSDL 文档中声明的消息类型，目标名称空间是"http://example.com/orders":

```
<variable xmlns:ORD="http://example.com/orders"
   name="orderDetails"
   messageType="ORD:orderDetails" />
```

每个<variable>在<scope>中被声明并且声称属于那个作用域。属于程序全局的变量被称作全局变量。程序也可能不属于全局作用域，这样的程序被称作局部变量。每个变量只在它被定义的作用域以及被嵌套在那个作用域中的作用域内可见的。因此，全局变量在整个程序中都是可见的。可以通过在内层作用域中声明统一的变量名来隐藏外部作用域中的变量声明。这些规则类似于在编程语言中变量的词法域。

[SA00023]在统一立即附属作用域中，<variable>的名字必须是唯一的。这个要求必须被静态地强制。[SA00024]变量名是NCNames（如同XML Schema规范里定义的），但禁止包含“.”字符。这个约束是必要的因为“.”字符是WS-BPEL对XPath 1.0 的缺省绑定分隔符（也就是通过"urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"识别绑定）。分隔符将WS-BPEL消息类型变量名同它的其中一个WSDL消息部分的名字分离开来。WSDL消息变量名字的拼接，分隔符以及WSDL部分名字作为XPath变量引用使用，这证明了XML Infoset对应了WSDL消息部分。

在这个规范中，`BPELVariableName` 类型用于描述`<variabel>`的名字。它源自于如下定义的 `XML Schema NCName`。`BPELVariableName` 类型用于描述一列变量名。

```
<xsd:simpleType name="BPELVariableName">
   <xsd:restriction base="xsd:NCName">
      <xsd:pattern value="[^\.]+" />
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BPELVariableNames">
   <xsd:restriction>
      <xsd:simpleType>
         <xsd:list itemType="tns:BPELVariableName" />
      </xsd:simpleType>
      <xsd:minLength value="1" />
   </xsd:restriction>
</xsd:simpleType>
```

变量存取遵循公共的词法域规则。变量解析最近的附属作用域，不管变量是何种类型，除了在 12.7 事件处理器中描述的。如果局部变量同在附属作用域中定义的变量有相同的名称，则局部变量将被用于局部赋值及/或 `bpel:getVariableProperty` 函数中（如下定义的）。

[SA00025]messageType，type或者element属性用于指定变量类型。必须指定其中之一。messageType属性引用WSDL消息类型定义。Type属性引用XML Schema类型（简单或者复杂的）。Element属性引用XML Schema元素。

使用[Infoset]术语，WS-BPEL元素变量的infoset由包含一个子集的文档信息条目（DII）和通过文档元素属性引用的元素信息条目（EII）组成。EII是元素变量值。

如果 WS-BPEL 实现选择表示一个简单类型变量作为 XML infuset，则 infoset 应该由包含一个子集的 DII 组成，EII 通过文档元素属性引用。本规范中没有定义文档元素的属性，特定的名称空间名字和局部命名属性。实现必须指定名称空间名字/局部命名值。不管怎样，文档元素的孩子必须由表示简单类型值的字符信息条目（CIIs）组成。WS-BPEL 实现可以选择将简单类型值映射到正在被使用的表达式/查询语言定义的 non-XML-infoset 数据模型

（如 XPath1.0 中的布尔类型）。

复杂类型的 infoset 由包含一个子集的 DII 组成，EII 通过文档元素属性引用。本规范中没有定义文档元素的属性，特定的名称空间名字和局部命名属性。实现必须指定名称空间名字/局部命名值。不管怎样，文档元素的孩子必须由赋值给变量的负责类型值组成。

为了简化数据存取，WSDL 消息变量的 WSDL 部件在 WS-BPEL 中作为 infosets 出现，每个 WSDL 消息部件中有一个 infoset。当 WSDL 消息部件作为 infoset 出现时，WS-BPEL 引擎必须使用如下算法：

WSDL 消息定义的每个部分：

步骤 1 - 创建合成的没有孩子的 DII 除了在步骤 2 中指定的。

步骤 2a - 如果 WSDL 消息部件被定义使用 type 属性则创建一个 EII 作为文档元素的孩子。新建 EII 的局部名字和名称空间名字由 WS-BPEL 处理器声明并且本规范没有指定。EII 的处理类似于 WS-BPEL 如何处理复杂或者简单类型 XML 变量的容器。新建 EII 的内容需要符合引用类型定义规定的内容。

步骤 2b – 如果 WSDL 消息部件被定义使用 element 属性则创建一个 EII 作为通过引用类型定义规定的文档元素的孩子。

先前的模型是概念上的；它们定义了 WS-BPEL 如何使用 infoset 定义来提交及检索 XML 变量值。WS-BPEL 处理器不需要实现 infoset 模型。不管变量绑定是如何被处理的，最后结果应该使用上面的 infoset 模型重复被定义的行为。例如，WS-BPEL 实现可能选择将简单类型 WS-BPEL 变量 xsd:string 类型直接绑定到 XPath 1.0 里的字符串对象。映射选择必须始终如一地被应用到变量以及相同的 XML Schema 类型的 WSDL 消息部件值里。比如，如果一个 `xsd:string` 变量作为一个字符串对象出现，`xsd:string` 消息部件必须也作为一个字符串出现。更多有关 Xpath 1.0 中的 WS-BPEL 变量显示定义信息，请见章节 8.2.2 在 Xpath1.0 中绑定 WS-BPEL 变量。

总结，WS-BPEL 变量作为 XML Infoset 出现有以下途径：

（1） 单个 XML infoset 条目：比如一个元素或者复杂类型变量或者 WSDL 消息部件

（2） 简单类型数据 CIIs 的序列段：比如，用来表示字符串（如果有需要，这些条目可能以非 XML infoset 条目出现，如 Boolean）

变量可以通过使用直接插入的 form-spec 来选择是否被初始化。From-spec 在章节 8.4 中定义。概念上直接插入变量初始化如同虚拟的包含一连串虚拟<assign>活动的<sequence>活动一样，每个变量按照列举的变量声明顺序被初始化。每个虚拟<assign>活动包含一个单独的其 from-spec 已在变量初始化时被给出的虚拟<copy>和一个指向被创建变量的 to-spec。

[SA00026]包含在作用域里的变量初始化逻辑或者其孩子包括的开始活动在from-spec里必须只能使用幂等函数。幂等函数允许变量值在每个程序实例中被预处理及再利用。

全局变量在程序初是未被初始化的状态。局部变量在其属于的作用域启动时也未被初始

化。请注意非全局作用域通常在它们属于的程序实例的生命周期内不止一次启动和完成。变量可以被多种包含赋值和接收消息的方法初始化。

在程序处理期间试图在变量初始化前读取变量或者消息类型变量或者变量的某部分一定会导致标准故障 bpel:uninitializedVariable。

变量验证

变量里存储的值在程序处理期间可以改变。<validate>活动用于确认变量值对于它们相关联的 XML 数据定义是否有效，包括 XML Schema 简单类型，复杂类型，元素定义以及 XML 的 WSDL 部分定义。<validate>活动有一个 variables 属性，列举变量验证。属性接收一个或多个变量名（BPELVariableName），用空格分离。Validate 活动的语法是：

```
<validate variables="BPELVariableNames" standard-attributes>
   standard-elements
</validate>
```

当一个或多个变量对于它们相关的 XML 定义无效时，必须抛出标准故障 bpel:invalidVariables fault

WS-BPEL 实现可以提供开/关任意显示验证的机制，例如<validate>活动。

WS-BPEL 实现可以在消息相关的活动执行期间验证输入和输出消息，比如<receive>,<reply>,<pick>,<onEvent>以及<invoke>活动。如果这样的 Schema 验证被激活并且消息无效，应该在这些下平息活动期间抛出 bpel:invalidVariables 故障。

8.2 查询和表达式语言的用法

本章描述了查询/表达式语言同来自两种不同透视的 WS-BPEL 间的关系。第一种透视是 WS-BPEL 的查询/表达式语言窗口。窗口受约束于 WS-BPEL 通过查询/表达式语言使用什么样的可用信息。第二种透视是查询/表达式语言的 WS-BPEL 窗口，指定 XPath 1.0 的处理上下文是如何被 WS-BPEL 初始化的。

WS-BPEL 提供在查询和表达式中使用语言的扩展机制。程序元素的 queryLanguage 和 expressionLanguege 属性指定语言。WS-BPEL 构造器需要或者允许查询或表达式为专用的查询/表达式提供覆盖缺省查询/表达式语言的能力。WS-BPEL 实现必须支持将 XPath 1.0 作为查询和表达式语言。XPath 1.0 通过 queryLanguage 和 expressionLanguege 的缺省值表示，如下：

```
urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0
```

表示了在 WS-BPEL 2.0 中 XPath 1.0 的用法。

如果查询或者表达式的处理产生一个未处理语言故障，必须抛出 WS-BPEL 标准故障 bpel:subLanguageExecutionFault。

8.2.1 附属元素

为了描述 WS-BPEL 提供给查询/表达式语言的视图，有必要接收一个新的项目-附属元素。

定义（附属元素）。附属元素被定义为包含查询或者表达式的 WS-BPEL 程序定义中的父元素。在下面的例子中，<form>元素就是附属元素。

```
<process>
   ...
   <from>$myVar/abc/def</from>
   ...
</process>
```

附属元素的输入作用域名称空间对于查询/表达式语言可见的名称空间。（请注意：XPath 1.0 没有缺省名称空间的概念）

链接、变量、伙伴链接等等被定义为基于实体的对于包含附属元素活动的可见性，对于查询/表达式语言都是可见的。查询/表达式语言不需要表明所有不同的对象。只有在的附属元素的附属活动的作用域中的对象应该在查询/表达式语言中可见。

WS-BPEL 表达式或者查询的评估将产生以下几种情况（这里我们使用 XPath 1.0 表达式作为例子）：

- 单个 XMLinfoset 条目：如 $myFooVar/lines/line[2]
- XML infoset 条目的集合：如 $myFooVar/lines/*
- 简单类型数据的 CIIs 的序列段：如$myFooVar/lines/line[2]/text()（请注意当需要时，这个条目的序列段可能作为一个非 XMLinfoset 出现，如 Boolean）
- 变量引用：如<from>$myFooVar</from>

8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量

除了其变量存取语法和语义学在章节 8.2.4 表达式语言的 XPath 1.0 的默认用法描述的连接表达式外，在 XPath 表达式中的 WS-BPEL 变量可以通过 XPath 变量绑定存取。特别指出的是，所有来自 XPath 表达式附属元素可见的 WS-BPEL 变量必须对 XPath 处理器可用，通过将 WS-BPEL 变量表示为与 WS-BPEL 变量名相同的 XPath 变量绑定的方式，除了变量有 WSDL messageType 声明，那个要求某些特殊操作（下面将会讨论）。

使用元素声明的 WS-BPEL 变量必须被表示为有一个单个成员节点的 node-set XPath

变量。那个节点是个合成的 DII 包含一个单独的孩子，值为 WS-BPEL 变量的文档元素。XPath 变量绑定将会绑定到文档元素。比如下面被给出的 Schema 定义：

```
<xsd:element name="StatusContainer">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="statusDescription" type="xsd:string"
            form="qualified" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

以及下面给出的变量声明：

```
<variable name="AStatus" element="e:StatusContainer" />
```

然后存取 statusDescription 元素值的 WS-BPEL XPath 表达式，假设 AStatus 变量在作用域中，如下：

```
$AStatus/e:statusDescription
```

$AStatus 指向变量的文档元素 StatusContainer。所以存取 StatusContainer 的孩子 statusDescription 只需指定孩子的元素值。

使用复杂类型声明的 WS-BPEL 变量必须被表示为有一个带有 WS-BPEL 复杂类型变量值的匿名文档元素的成员节点的 node-set XPath 变量。XPath 变量绑定给文档元素。比如，下面给出的 Schema 定义：

```
<xsd:complexType name="AuctionResults">
   <xsd:sequence>
      <xsd:element name="AuctionResult" maxOccurs="unbounded"
         form="qualified">
         <xsd:complexType>
            <xsd:attribute name="AuctionID" type="xsd:int" />
            <xsd:attribute name="Result" type="xsd:string" />
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
```

以及下面给出的变量声明：

```
<variable name="Results" type="e:AuctionResults" />
```

之后存取第二个 AuctionID 属性值的 WS-BPEL XPath 表达式如下：

```
$Results/e:AuctionResult[2]/@AuctionID
```

$Results 指向变量的文档元素，AuctionResult[2]指向第二个文档元素的孩子 AuctionResult，@AuctionID 指向在选中的 AuctionResult 元素上的 AuctionID 属性。

WS-BPEL messageType 变量必须在 XPath 中作为一连串的变量出现，messageType 的每个部分一个变量。每个变量通过链接消息变量的名字来命名，“.”字符和部件的名字。WS-BPEL messageType 变量中的数据作为单个 XPath 变量在默认的查询和表达式语言绑定下对一般的 XPath 处理是无法使用的。比如，如果一个 messageType 变量被命名为“myMessageType”并且包含两部分，“msgPart1”和“msgPart2”则在作用域中带有“myMessageType”的 XPath 绑定将会表示两个 XPath 变量，$myMessageTypeVar.msgPart1 和$myMessageTypeVar.msgPart2.

WSDL 消息部分定义总是使用 WSD 元素，XSD 复杂类型或 XSD 简单类型。这些消息部分在 XPath 中的表现对于在这里被指定的元素-复杂类型和简单类型 WS-BPEL 变量，可以用相同的方式处理。

下面是在 WS-BPEL XPath 中怎样表现 WSDL 消息类型的完整例子。

```
<message name="StatusMessage">
   <part name="StatusPart1" element="e:StatusContainer" />
   <part name="StatusPart2" element="e:StatusContainer" />
</message>
```

以及给出下面的变量声明：

```
<variable name="StatusVariable" messageType="e:StatusMessage" />
```

之后存取第二个部分的 statusDescription 元素的 WS-BPEL 表达式如下：

```
$StatusVariable.StatusPart2/e:statusDescription
```

写出通过 WSDL 消息变量多个部件实现同时查询的 WSDL 查询是可能的，通过应用一个合并操作符来创建一个单个的节点集。例如：

```
( $StatusVariable.StatusPart1
| $StatusVariable.StatusPart2 )//e:statusDescription
```

WS-BPEL 简单类型变量必须被直接表现为 XPath 字符串，布尔类型或者浮点数对象。如果 WS-BPEL 简单类型变量的 XML Schema 类型是 xsd:boolean 或者其他 xsd:boolean 约束的类型则 WS-BPEL 变量必须被表示为 XPath 布尔对象。如果 WS-BPEL 简单类型变量的 XML Schema 为 xsd:float, xsd:int, xsd:unsignedInt 或者这些类型约束的类型，则 WS-BPEL 变量必须被表示为 XPath 浮点数对象。其他类型的 XML Schema 必须被表示为 XPath 字符串对象。

XPath 1.0 中浮点数对象的精度不能精确表示某些 XML Schema 数据类型值，比如 xsd:decimal.XSD 数值没有丢失精度就不能被表示，因此 XPath 浮点数对象必须通过 WS-BPEL 处理器转化为 XPath 字符串对象。

8.2.3 XPath 1.0 远景和 WS-BPEL

XPath 1.0 规范[XPATH 1.0]定义了五个定义在被估值XPath表达式中的上下文的要点。这些要点介绍如下：

- 节点(上下文节点)
- 一对非零正整数（上下文的位置和长度）
- 一套变量绑定
- 函数库
- 在作用域中一套表达式的名称空间声明

下面的片段定义了在 WS-BPEL 中，不同类型的 WS-BPEL 表达式和查询语言的上下文是如何被初始化的。

8.2.4 XPath 表达式语言的默认用法

当 XPath 1.0 被用于表达式语言时，除了在章节 8.2.5 XPath 1.0 表达式语言在连接条件中的用法 指定的外，XPath 上下文都被如下初始化：

Context node	None
Context position	None
Context size	None
A set of variable bindings	Variables visible to the Enclosing Element as defined by the WS-BPEL scope rules
A function library	WS-BPEL and core XPath 1.0 functions MUST be available and processor-specific functions MAY be available
Namespace declaration	In-scope namespace declarations from Enclosing Element

值得强调的是正如 XPath 1.0 标准定义的当解析 XPath 时，变量里使用的名称空间前缀是不相关的。唯一有关系的前缀是在作用域中的名称空间。

比如，假设一个“foo”元素类型的名为“FooVar”的 WS-BPEL 变量带有如下值：

```
<a:foo xmlns:a="http://example.com">
   <a:bar>23</a:bar>
```

```
</a:foo>
```

下面的 XPath 将会返回值 23：

```
<from xmlns:b="http://example.com">$FooVar/b:bar/text()</from>
```

请注意在先前的例子中 bar 元素中有实际值时使用“b”名称空间前缀优于使用“a”名称空间前缀。

同样值得强调的是 XPath 1.0 明确地要求在 XPath 表达式中使用的元素或属性如果没有名称空间前缀必须作为不合格的名称空间对待。也就是说，即使附属元素上定义了缺省名称空间，缺省名称空间仍然不会被应用。

在下面使用先前提供的相同的 Foo 值，将会返回一个 bpel：selectionFailure 故障（在可执行 WS-BPEL 中），因为它不能在<copy>操作上的下文中选择节点：

```
<from xmlns="http://example.com">$FooVar/bar/text()</from>
```

在XPath里面的值不会继承附属元素的缺省名称空间。所以XPath里引用的‘bar’元素里面不曾有任何名称空间值，因此不会匹配FooVar变量中名称空间值为http://example.com的bar元素。

允许 WS-BPEL 变量表示为 XPath 变量绑定使得 WS-BPEL 程序员可以创建强大的 XPath 表达式包括多个 WS-BPEL 变量。比如：

```
<assign>
   <copy>
      <from>$po/lineItem[@prodCode=$myProd]/amt * $exchangeRate</from>
      <to>$convertedPO/lineItem[@prodCode=$myProd]/amt</to>
   </copy>
</assign>
```

[SA00027]当XPath1.0 在WS-BPEL中被用为表达式语言时，没有可用的上下文节点。因此XPath Expr生成式的法定值(http://www.w3.org/TR/xpath#NT-Expr)必须被约束以便阻止存取上下文节点。

特别地说，当XPath被用为表达式语言时，“PathExpr”(http://www.w3.org/TR/xpath#NT-PathExpr)生产式规则的“LocationPath”(http://www.w3.org/TR/xpath#NT-LocationPath)生产式规则禁止被使用。前面的将XPath作为表达式语言的XPath Expr生产式约束必须被静态地强制。

这个约束的结果就是WS-BPEL表达式或者查询语言XPath的“PathExpr”将总是以“PrimaryExpr” (http://www.w3.org/TR/xpath#NT-PrimaryExpr)开始。值得记住的是PrimaryExpr不管是变量引用，表达式，常量，数字还是函数调用，都不能存取上下文节点。

额外的约束作为包含 to-spec 的表达式语言应用于 XPath 中。

8.2.5 表达式语言在连接条件中 XPath 1.0 的使用

当 XPath 1.0 作为表达式语言在连接条件中使用时，XPath 上下文初始化如下：

Context node	None
Context position	None
Context size	None
A set of variable bindings	Links that target the activity that the Enclosing Element is contained within
A function library	Core XPath functions MUST be available, [SA00028] WS-BPEL functions MUST NOT be available, and processor-specific functions MAY be available.
Namespace declaration	In-scope namespace declarations from Enclosing Element

正如章节 11.5.1 中解释的一样，表达式在连接条件中可能只访问目标连接条件的附属活动的链接状态。没有其他的数据可能有用。到最后连接条件可用的变量绑定只有访问链接状态：

<link>状态通过 XPath 变量绑定获得，表示以活动为目标的<link>包含一个相同名字的 XPath 变量绑定。也就是说，如果有一个名叫“ABC”的以活动为目标的<link>，则一定有一个叫“ABC”的 XPath 变量绑定。链接变量被表示为 XPath 布尔对象，其值被设置给链接。

下面是<targets>元素内部的<joinCondition>示例：

```
<targets>
   <target linkName="link1" />
   <target linkName="link2" />
   <joinCondition>$link1 and $link2</joinCondition>
</targets>
```

8.2.6 拷贝操作以及属性别名中 XPath 1.0 查询语言的用法

当 XPath 1.0 作为查询语言在<copy>赋值或者<vprop:propertyAlias>中 from-spec 及 to-spec 的第一个变量使用时，XPath 上下文被如下初始化：

	Variable variant from-spec or to-spec	<vprop:propertyAlias>
Context node	See below	See below
Context position	1	1
Context size	1	1
A set of variable bindings	Variables visible to the Enclosing Element as defined by the WS-BPEL scope rules	There MUST NOT be any variable bindings available when XPath is used as the query language in a <vprop:propertyAlias>
A function library	WS-BPEL and core XPath 1.0 functions MUST be available and processor-specific functions MAY be available	Core XPath functions MUST be available, [SA00029] WS-BPEL functions MUST NOT be available, and processor-specific functions MAY be available.
Namespace declaration	In-scope namespace declarations from Enclosing Element	In-scope namespace declarations from Enclosing Element (note that the Enclosing Element is in a <vprop:propertyAlias> defined in a WSDL definition)

上下文节点被如下定义:

- 当 from-spec 或 to-spec 引用 messageType 变量或者 <vprop:propertyAlias>的 messageType/part 属性被使用时:
  - ✧ 如果消息部分基于复杂类型或者元素，则上下文节点必须指向包含单个的已引用部分是 EII 的节点的节点列表(在章节 8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量 指定)。
  - ✧ 如果消息部分基于简单类型，则上下文节点必须指向 XPath 对象（在章节 8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量 指定)。
- 当 from-spec 或 to-spec 引用 XML Schema 类型变量或者 <vprop:propertyAlias>的类型属性被使用时:
  - ✧ 如果类型是复杂类型，则上下文及诶但必须指向包含单个的已引用部分是 EII 的节点的节点列表（在章节 8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量 指定)。
  - ✧ 如果类型是简单类型，则上下文节点必须指向 XPath 对象（在章节 8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量 指定)。
- 当 from-spec 或 to-spec 引用 XML Schema 元素变量或者 <vprop:propertyAlias>的元素属性被使用时，上下文节点必须指向包含单个的已引用部分是 EII 的节点的节点列表（在章节 8.2.2 在 XPath 1.0 中绑定 WS-BPEL 变量 指定)。

先前列出的有关 XPath 表达式的约束不能应用于 from-spec/to-spec 以及

<vprop:propertyAlias> 中的<query>，因为它定义了上下文节点。可以使用合法的 XPath 表达式。可以在<vprop:propertyAlias>中使用绝对或者相对路径解析是根节点的上下文节点。

这个例子演示了使用相对 XPath 查询的<vprop:propertyAlias>。它返回了个左值：

```
<vprop:propertyAlias propertyName="p:price"
   messageType="my:POMsg"
   part="poPart">
   <vprop:query>price</vprop:query>
</vprop:propertyAlias>
```

相反地，这个例子演示了使用绝对路径查询的<vprop:propertyAlias>。它没有返回左值：

```
<vprop:propertyAlias propertyName="p:goldCustomerPrice"
   messageType="my:POMsg"
   part="poPart">
   <vprop:query>(/p:po/price * 0.9)</vprop:query>
</vprop:propertyAlias>
```

<query>并不要求返回左值。当<query>被用于变量不同的 to-spec 或者<vprop:propertyAlias>的<query>被用于属性不同的 to-spec 时不会返回左值，试图赋值给这样的 to-spec 一定会失败，并且抛出 bpel:selectionFailure 异常（正如章节 8.4 赋值 中定义的）。多节点可以用<vprop:propertyAlias>特性选择。不管怎样，这些选择稍后可能在余下的表达式中已筛选并且导致返回一个节点。

8.3 表达式

WS-BPEL 使用若干类型的表达式，如下（相关应用上下文在圆括号中列出）：

- 布尔表达式（事务，连接，while 以及 if 条件）
- 最终期限表达式（until 表达式的<onAlarm>和<wait>）
- 持续时间表达式（for 表达式的<onAlarm>和<wait>，<repeatUtill>表达式的<onAlarm>）
- 无符号整数表达式（<startCounterValue>, <finalCounterValue>, 以及<forEach>中的<branches>）
- 普通表达式（<assign>）

当上述前四种类型的表达式被使用时，相应的表达式应该根据相应 XML Schema 类型返回有效值：

- 布尔表达式应该返回 xsd:boolean 的有效值
- 最终期限表达式应该返回 xsd:date 和 xsd:dateTime 的有效值

- 持续时间表达式应该返回 xsd:duration 的有效值
- 无符号整数表达式应该返回 xsd:unsignedInt 的有效值

另外，bpel:invalidExpressionValue 故障应该被抛出。当数据处理从表达式返回值时，隐式数据转换或者挖掘可以被应用，基于下层表达式语言确定的数据模型或类型转换语义。

下面的值转换以及有效性验证语义必须被应用当 WS-BPEL 到 XPath 1.0 的缺省绑定用为表达式语言时：

- 对于 WS-BPEL 布尔表达式，XPath 的 boolean（对象）函数用于将表达式结果转换为布尔值（如果需要的话）。
- 对于 WS-BPEL 最后期限表达式，XPath 的 string（对象）函数用于将表达式结果转换为字符串值（如果需要的话）。字符串值必须是 xsd:date 和 xsd:dateTime.的有效值。另外，bpel:invalidExpressionValue 故障必须被抛出。
- 对于 WS-BPEL 有效期限表达式，XPath 的 string（对象）函数用于将表达式结果转换为字符串值（如果需要的话）。字符串值必须是 xsd:duration 的有效值。另外，bpel:invalidExpressionValue 故障必须被抛出。
- 对于 WS-BPEL 无符号整数表达式，XPath 的号码（对象）函数用于将表达式结果转化为数值（如果需要的话）。数值必须是 xsd:unsignedInt 的有效值（也就是不管是负数还是网络应用节点并且必须是整数值）。另外，bpel:invalidExpressionValue 故障必须被抛出。

下列 WS-BPEL 定义的 XPath 扩展函数必须被 WS-BPEL 实现支持。

- getVariableProperty,下面将描述
- doXslTransform，在章节 8.4 赋值 中描述

这些扩展在标准 WS-BPEL 名称空间中定义（见章节 5.3 语言扩展性 可以得到有关 WS-BPEL 语言扩展性的全面论述）

XPath表达式里使用的修饰名必须通过在表达式定位的WS-BPEL文档作用域中使用的名称空间声明解析。空值前缀必须作为在[XSLT 1.0] 章节 2.4 中指定的情况处理（空值前缀是指空的名称空间被使用）。

bpel:getVariableProperty 函数签名是：

```
object bpel:getVariableProperty(string, string)
```

这个函数从变量里提取属性值。第一个参数代表数据资源变量名，第二个是从那个变量中选取的属性的QName（见章节 7.变量属性）。[SA00031]第二个参数必须是遵照Qname定义的字符串文字（见章节 3.同其他名称空间的关系），并且这些约束必须由静态分析强制。

这个函数的返回值通过提供请求属性适当的<vprop:propertyAlias> 给被提交变量的当前值来计算。

[SA00030] bpel:getVariableProperty的参数必须作为引用字符串被给出。先前的要求必须被静态地强制。因此禁止将XPath变量，XPath函数的输出，XPath定位路径或者其他任何不是引用字符串的值考进WS-BPEL XPath函数中。比如bpel:getVariableProperty("varA","b:propB")满足先前的定义但是bpel:getVariableProperty($varA, string(bpel:getVariableProperty("varB", "b:propB"))不符合。请注意先前的请求制定了个在XPath标准中不存在的约束。

8.3.1 布尔表达式

这些是遵照 XPath 1.0 Expr 生成式中其评估导致布尔值的表达式。

8.3.2 最终期限表达式

这些是遵照 XPath 1.0 Expr 生成式中其评估导致 XML Schema 类型 dataTime 或者 date 的表达式。

请注意 XPath 1.0 不是明确的 XML Schema。同样地，XPath 1.0 的内置函数不能生成或者操作 dataTime 或者 date 值。尽管如此，它还是可以写入遵照 XML Schema 定义的变量并且将其作为最终期限值或者从这些类型之一的（部分）变量中提取字段并且作为最终期限值使用。XPath 1.0 将常量当做字符串常量处理，但是结果可以被解释为 dateTime 或 date 值的词汇表示法。

8.3.3 持续时间表达式

这些是遵照 XPath 1.0 Expr 生成式中其评估导致 XML Schema 类型持续时间值的表达式。前面有关 XPath 1.0 的缺乏 XML Schema 明确表示的讨论也适用于这里。

8.3.4 无符号整数表达式

这些是遵照 XPath 1.0 Expr 生成式中其评估导 XPath 无符号整数类型的编号对象值的表达式。

8.3.5 一般表达式

这些是遵照 XPath 1.0 Expr 生成式中其值导致任何 XPath 类型值（string，number 或 Boolean）的表达式。

8.4 赋值

<assign>活动可以用于从一个变量拷贝数据给其他变量，也可以使用表达式来构建以及插入新数据。首先是需要执行简单计算（比如递增字段编号）才使用表达式。表达式通过操作变量，属性以及常量来生成一个新值。<assign>活动也可以用于从伙伴链接中拷贝端点引用，或者将端点引用拷贝给伙伴链接。同样它也可以包含可扩展数据处理操作正如在名称空间下定义的不同于 WS-BPEL 名称空间的可扩展元素。如果元素包含在 extensionAssignOperation 元素中则不会被 WS-BPEL 处理器识别并且不会服从来自扩展声明里的 mustUnderstand=“yes”的请求，于是 extensionAssignOperation 操作必须被忽略（见章节 14 扩展声明）。

最后，它可以包含可扩展数据处理操作如同在名称空间下定义的不同于 WS-BPEL 名称空间的可扩展元素。

<assign>活动包含一个或多个初等运算。

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
   (
   <copy keepSrcElementName="yes|no"? ignoreMissingFromData="yes|no"?>
      from-spec to-spec
   </copy>
   |
   <extensionAssignOperation>
      assign-element-of-other-namespace
   </extensionAssignOperation>
   )+
</assign>
```

<assign>活动从资源（“from-spec”）中拷贝可兼容类型的值给目的地（“to-spec”），使用<copy>元素。[SA00032]除了在抽象流程中，from-spec必须是以下几种变量之一：

```
<from variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"?>?
      queryContent
   </query>
</from>
<from partnerLink="NCName" endpointReference="myRole|partnerRole" />
<from variable="BPELVariableName" property="QName" />
<from expressionLanguage="anyURI"?>expression</from>
<from><literal>literal value</literal></from>
<from/>
```

在抽象流程中，from-spec 必须是上述几种类型之一或者是在章节 13.1.3 隐藏语法元素中描述的不透明变量。

To-spec 必须是以下几种变量之一：

```
<to variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"?>?
      queryContent
   </query>
</to>
<to partnerLink="NCName" />
<to variable="BPELVariableName" property="QName" />
<to expressionLanguage="anyURI"?>expression</to>
<to/>
```

To-spec必须返回左值。如果to-spec没有返回左值则bpel:selectionFailure必须被抛出。左值就是在XPath上下文中的包括由to-spec识别的来自variable或者partnerLink的单个节点的节点列表。在章节8.2.4 XPath表达式语言的默认用法 中列出的约束必须将XPath作为查询语言应用。 [SA00033]另外，XPath 查询必须以XPath VariableReference开始。这个约束必须被静态地强制。

变量体：在第一个from-spec和to-spec变体中，variable属性提供变量的名字。如果变量的类型是WSDL messageType，则可选择的part属性可能被用于提供变量里的部件名字。[SA00034]当变量使用XML Schema类型（简单或复杂）或者元素定义时，part属性禁止被使用。可选择的<query>元素可以用于选择从资源或目标变量或者消息部分中选择值。Query的计算值必须是如下之一：

- 单个 XML 信息条目除了 CII，如 EII 和 AII
- 零个或多个 CII 的字段：这个是被映射到 XPath 1.0 数据模型中的 TextNode 或者字符串

伙伴链接变量体：第二个from-spec和to-spec变体中允许操作同伙伴链接相关联的端点引用。partnerLink属性值是在作用域中的partnerLink的名字。在from-spec中，角色必须被指定。“myRole”值是指程序关于伙伴链接的端点引用是源，“partnerRole”值是指伙伴链接中伙伴的端点引用是源。[SA00035] [SA00036]如果值“myRole”或“partnerRole”被使用，则对应的<partnerLink>声明必须指定对应的myRole或partnerRole属性。这个约束必须被静态地强制。对于to-spec，只能对partnerRole赋值，因此没有必要指定角色。[SA00037]因此，to-spec只能访问其声明指定了partnerRole属性的<partnerLink>。这个约束必须被静态地强制。partnerLink式样的from-spec/to-spec引用值的类型是<sref:service-ref>元素（见章节 6.伙伴链接类型，伙伴链接和端点引用）。

PartnerRole EPR 被初始化之前，试图在程序执行期间读伙伴链接一定会导致bpel:uninitializedPartnerRole 标准故障。当伙伴链接的伙伴角色在<assign>活动的<invoke>或者部分<copy>的<from>中被引用时被读取。

属性变量体：第三个 from-spec 和 to-spec 变量体允许数据操作使用属性（见章节 7.变量属性）。from-spec 生成的属性值的产生同 bpel：getVariableProperty（）函数返回值的方式一样。属性变量体提供了清晰定义消息里独特的数据元素正在被如何使用的方法。

表达式变量体：在第四个 from-spec 变量体中，由可选择的 expressionLanguege 属性识别的表达式语言，可以用来返回和选择值。这个表达式的计算值必须是如下之一:

- 单个 XML 信息条目除了 CII，如 EII 和 AII
- 零个或多个 CII 的字段：这个是被映射到 XPath 1.0 数据模型中的 TextNode 或者字符串

不管是使用第一种形式的 from-spec/to-spec 还是第四种形式的 from-spec/to-spec，都可以在非消息变量和部分消息变量上执行拷贝，因为本规范定义了如何将非消息变量部分消息变量辨识为 XML Infoset 信息条目。尽管如此，只有第一种形式的 from-spec/to-spec 可以拷贝包括所有部分的完整消息变量。其他 from-spec 和 to-spec 变量体只能访问 WSDL 消息类型变量的单个部分并且不能一次性拷贝所有部分。

常量变量体：第五个 from-spec 变量体允许常量被当作赋值给目的的源值来给出。为了防止同<from>下标准扩展元素相冲突，被赋值的常量值包括<literal>元素。<literal>元素自身不允许标准可扩展性。常量值的类型可以在子过程中被随意地指定连同 XML Schema 的实例类型机制（xsi：type）使用的值。

第五个from-spec变量体返回值如同它是选取WS-BPEL源代码中的<literal>元素孩子的from-spec一样。[SA00038]返回值只能是单个EII或者文本信息条目（TII）。这个约束必须在静态分析中强制（见章节 8.4.1 为TII定义选取拷贝操作结果）。在源代码中<literal>元素的XML语法分析上下文如XML 名称空间，被带入<literal>元素内部的孩子的语法分析。空<literal/>元素返回空的TII。这里是一些说明的例子：

```
<assign>
   <copy>
      <from>
         <literal xmlns:foo="http://example.com">
            <foo:bar />
         </literal>
      </from>
      <to variable="myFooBarElemVar" />
   </copy>
   <copy>
      <from>
         <literal>
            <![CDATA[<foo:bar/>]]>
         </literal>
      </from>
      <to variable="myStringVar" />
```

```
   </copy>
   <copy>
      <from>
         <literal />
      </from>
      <to variable="myStringVar" />
   </copy>
</assign>
```

第一个<copy>将带有”foo”前缀同“http://example.com”namespace into “myFooBarElemVar”相关联的<foo：bar/>元素复制给“myFooBarElemVar”。第二个<copy>复制了值为“<foo：bar/>”的字符串给“myStringVar”。最后一个<copy>复制一个空字符串给“myStringVar”。

当同 to-spec 的伙伴链接变量体一起使用时，常量 from-spec 变量体也允许将常量<sref:service-ref>值赋值给伙伴链接。

空变量体：第六个 from-spec 变量体和第五个 to-spec 变量体明确显示 from-spec 和 to-spec 是可扩展的。请注意如果这些变量体是不可扩展的，或者扩展性没有被接受，则它们必须像表达式变量体一样返回零个节点。

除<copy>规范外，其他可扩展的数据操作元素可以被包括在 assign 活动中，内置<extensionAssignOperation>元素。可扩展数据操作元素必须属于不同于 WS-BPEL 名称空间的名称空间。

Assign 和 Copy 的属性

<copy>构造器中可选择的 keepSrcElementName 属性指定在拷贝操作中目的元素名字是否会被源元素名称（如 from-spec 选择的那样）取代（见章节 8.4.2 拷贝操作的替换逻辑）。

<copy>构造器中可选择属性 ignoreMissingFromData 用于指定是否抑制 bpel：selectionFailure 标准故障如章节 8.4.1 选择拷贝操作结果 中指定的一样。ignoreMissingFromData 属性的缺省值是“no”。

可选择属性 validate 可以同<assign>活动一起使用。缺省值为“no”。当 validate 设为“yes”时，<assign>活动验证所有正在被活动修改的变量。WS-BPEL 实现可以提供开/关显示验证的机制。如 assign 中的 validate 属性。

如果<assign>活动中的“validate”部分失败，则表示有变量对于其对应的 XML 定义无效，标准故障 bpel：invalidVariables 必须被抛出。

如果在赋值活动的处理过程中发生故障，则目的变量不会发生改变，如同在活动的开始状态（如同赋值活动是原子的）。这个在整体赋值活动中应用与赋值元素的数目无关。

赋值活动在整段时期内必须像唯一正在被执行的活动一样执行。

迄今描述的拷贝机制，当与默认的 XPath 1.0 表达式语言结合时，不会执行复杂 XML 变换。为了能够可移植的寻址这个约束，WS-BPEL 处理器必须支持 bpel：doXslTrasform（）XPath 1.0 可扩展函数。函数的 bpel：doXslTrasform 签名是：

```
object bpel:doXslTransform(string, node-set, (string, object)*)
```

第一个参数是XPath字符串提供了命名WS-BPEL处理器使用的样式表的URI。[SA00039]这必须表现为字符串常量的形式。这个约束的目的是允许实现为变量依赖静态地分析程序（并且命名样式表）；必须被静态分析强制。

第二个参数是 XPath 节点集提供 WS-BPEL 处理器执行转换的资源文档。集合必须包括单个的 EII（也就是 在 XPath 1.0 数据模型的元素节点）。如果没有，则 WS-BPEL 处理器必须抛出 bpel：xsltInvalidSource 故障。这个参数指定的单个 EII 在 XSLT 处理中必须作为资源树根结点的单个孩子对待。

后面的可选择参数必须成对出现。每对定义为：

- XPath 字符串参数提供 XSLT 参数的限定名
- XPath 对象参数提供已命名 XSLT 参数值。可以是 XPath Expr.

[SA00040]WS-BPEL处理器必须通过静态分析来强制这些成对的参数（也就是奇数个参数必须导致静态分析错误）。

函数必须返回转换结果。结果为以下 infoset 条目之一，这取决于被选择的样式表的 WSLT 输出方法：

- 单个 TII（XPath 1.0 文本节点），由 XSLT “text”或者“html”输出方法创建，或者
- 单个 EII（XPath 元素节点，是生成树根部的单个孩子），由 XSLT“xml”输出方法创建。

WS-BPEL 处理器必须执行下面提供的 bpel：doXslTrasform 函数：

- 第一个参数命名被使用的样式表，必须用于寻找与被给出的 URI 对应的样式表。这个适合于在执行相关样式中。如果样式表对应的 URI 没有找到，则 WS-BPEL 处理器必须抛出 bpel：xsltStylesheetNotFound 故障。
- 处理器必须执行 XSLT 1.0 转换，正如在 XSLT 1.0 规范的 章节 5.1 处理模型 中表述的一样，使用已命名的样式表作为最初的图表，提供的消息 EII 作为资源文档，并且生成树作为转换的结果。
- XSLT 全局参数([XSLT 1.0], XSLT 1.0 规范的 11.4 章节)用于传递附加值从WS-BPEL程序到XSLT处理器。doXslTrasform函数的可选参数以名字-值的方式成对出现在自变量表中，如上所描述的一样。它们通过QName识别XSLT全局参数，并且提供值给已命名的全局参数。[SA00041]全局参数的名字必须是符合[Namespaces in XML]章节 3 的常量字符串，并且这些约束必须由静态分析强制。WS-BPEL处理器必须传递被给出的全局参数和值给XSLT处理器。
- 如果 XSLT 处理故障在转换期间发生，则 bpel：subLanguageExecutionFault 必须

被抛出。

因为 XSLT 是单边自由效应语言，所以转换的执行不会导致样式表中访问的 WS-BPEL 变量改变。

第一个 XPath 函数参数命名样式表，和<import>元素的局部属性有类似的语义。和程序相关的样式表（通过它的 doXslTransform 调用）应该被看做程序定义的一部分，就像<import>元素引用的 WSDL 定义和 XML Schemas。

bpel：doXslTransform 例子

下面的例子显示了复杂文档转换和迭代文档构造器。

复杂文档转换。WS-BPEL 程序中的公共样式包括从一个服务接收 XML 文档，将其转换成一个不同的 Schema 来形成新的请求消息，以及发送新的请求给其他的服务。这样的文件转换可以用 XSLT 通过 bpel：doXslTransform 函数完成。

```
<variables>
   <variable name="A" element="foo:AElement" />
   <variable name="B" element="bar:BElement" />
</variables>
...
<sequence>
   <invoke ... inputVariable="..." outputVariable="A" />
   <assign>
      <copy>
         <from>
            bpel:doXslTransform("urn:stylesheets:A2B.xsl", $A)
         </from>
         <to variable="B" />
      </copy>
   </assign>
   <invoke ... inputVariable="B" ... />
</sequence>
```

在字段中，服务被调用，并且结果（foo：AElement）拷贝给变量 A。<assign>活动用于转换变量 A 的内容给 bar：BElement，并且将转换的结果拷贝给变量 B。变量 B 用于调用其他服务。

样式表 A2B.xsl 包括将 Schema foo：AElement 的文档转换给 Schema bar：BElement 的 XSL 规则。

迭代文档构造器：假定文档通过重复调用服务来构造，并且将结果累加到 XML 变量中。循环可能如下所示：

```
<variables>
   <variable name="PO" element="foo:POElement" />
   <variable name="OutVar" element="foo:ItemElement" />
</variables>

<!-- ... PO is initialized ... -->

<!-- Iteratively add more items to PO until complete -->
<while>
   <condition>...</condition>
   <sequence>
      <!-- Fetch next chunk into OutVar -->
      <invoke ... inputVariable="..." outputVariable="OutVar" />
      <assign>
         <copy>
            <from>
               bpel:doXslTransform( "urn:stylesheets:AddToPO.xsl",
                                    $PO, "NewItem", $OutVar)
            </from>
            <to variable="PO" />
         </copy>
      </assign>
   </sequence>
</while>
```

doXslTransform 调用中给出的可选择参数指定了名为“NewItem”的 XSLT 参数设置为 WS-BPEL 变量 OutVar 的值。为了允许 XSLT 样式表访问这个值，它包括了一个与上述函数的第三个参数名字相匹配的（最高级别的）全局参数。

```
<xsl:transform version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ...>
   <!-- NewItem variable set by WS-BPEL process;
        defaults to empty item -->
   <xsl:param name="NewItem">
      <foo:itemElement />
   </xsl:param>
   ...
</xsl:transform>
```

样式表包括一个将全局参数 NewItem 值（来自程序实例的 OutVar 值）附加到 PO 变量里存在的条目表的模板。

```
<xsl:template match="foo:itemElement">  <!-- line 1 -->
```

```
   <xsl:copy-of select="." />           <!-- line 2 -->
   <xsl:if test="position()=last()">    <!-- line 3 -->
      <xsl:copy-of select="$NewItem" /> <!-- line 4 -->
   </xsl:if>                            <!-- line 5 -->
</xsl:template>                         <!-- line 6 -->
```

这个模板拷贝所有存在于源文件中（第一行和第二行）的条目并且将 XSLT 参数 NewItem 的上下文附加给条目列表（第 3 行和第 4 行）。它检验当前节点是否在样式表的末端并且从 XSLT 参数 NewItem 中拷贝结果树片段给最后的条目（第 4 行）。

如果 PO 有下列的值：

```
<foo:poElement>
   <foo:itemElement>item 1</foo:itemElement>
</foo:poElement>
```

在<while>循环的迭代开始时且<invoke>活动返回<foo：itemElement>item 2</foo：itemElement>的值，则<from>的验证会导致以下的值：

```
<foo:poElement>
   <foo:itemElement>item 1</foo:itemElement>
   <foo:itemElement>item 2</foo:itemElement>
</foo:poElement>
```

当<copy>操作完成时，变为新的 PO 变量值。

8.4.1 拷贝操作的选择结果

<copy>操作里使用的from-spec或者to-spec的选择结果必须是下面三种信息条目之一：元素信息条目（EII），属性信息条目（AII），或者文本信息条目（TII）。EII和AII在[Infoset]中定义，而在本规范中TII定义成XML Infoset模型和其他公用XML数据模型的桥梁，如XPath 1.0.

文本信息条目（TII）是零个或多个字符信息条目的序列，根据文档顺序 TII 不是显式的且自身直接在 XML 串行化中。当 TII 映射给 XPath 1.0 模型时，可以归纳为一个字符串对象（有零个或多个字符）和文本节点（有零个或多个字符）。TII 左值禁止为空。TII 右值可以映射给 XPath1.0 中的文本节点，字符串/布尔/数字对象，同时 TII 左值必须映射给文本节点。

如果 from-spec 或 to-spec 的选择结果属于信息条目而不是 EII,AII 或 TII，则 bpel：selectionFailure 故障必须被抛出。如果选择结果里包括任一不支持的信息条目，则它们必须被保持；唯一的约束就是当它们为最高阶层的条目时，禁止被 from-spec 或 to-spec 直接选择。

<copy>操作是一对一置换操作。如果可选择属性 ignoreMissingFromData 属性值设为“yes”并且 from-spec 返回零个 XML 信息条目，则<copy>必须是空操作，没有 bpel：selectionFailure 被抛出。在这种情况下，to-spec 禁止被验证。如果遇到以下的情况，即使

ignoreMissingFromData 属性值设为“yes”，bpel：selectionFailure 也必须被抛出：

1. from-spec 选择多个 XML 信息条目
2. from-spec 选择一个 XML 信息条目并且 to-spec 没有选择确定的 XML 信息条目。

如果 ignoreMissingFromData 属性值为“no”，则要求 from-spec 和 to-spec 都必须明确地选择上述三种信息条目之一。如果 from-spec 或 to-spec 在执行期间没有选择信息条目，则标准故障 bpel：selectionFailure 必须被抛出。下表说明了 ignoreMissingFromData 属性在<copy>操作中的行为：

returned nodes		**ignoreMissingFromData**	
from	**to**	**“no”**	**“yes”**
0	0	selectionFailure	no-op
0	1	selectionFailure	no-op
0	N	selectionFailure	no-op
1	0	selectionFailure	selectionFailure
1	1	copy	copy
1	N	selectionFailure	selectionFailure
N	0	selectionFailure	selectionFailure
N	1	selectionFailure	selectionFailure
N	N	selectionFailure	selectionFailure

ignoreMissingFromData 逻辑表

字面值（from-spec 的字面变量体）必须包括作为最级别值的单个的 TII 或 EII。当 from-spec 的右值是 AII 时，to-spec 设为在章节 8.4.2 拷贝操作的置换逻辑 中指定 AII 的规格化值属性构造的 TII。

当在<copy>操作中同 from-spec 和 to-spec 非伙伴链接变量体一起使用 from-spec 和 to-spec 的伙伴链接变量体时，partnerLink 变量体应该就像它们创建了 EII 的右值和左值一样被处理，EII 的 [local name] 为 “ service-ref ” 且 [namespace name] 为 "http://docs.oasis-open.org/wsbpel/2.0/serviceref".

8.4.2 拷贝操作的置换逻辑

这章提供了<copy>操作中 to-spec 引用的数据置换的规则。附录 D 置换逻辑例子 中提供了例子的详细信息。

WSDL 消息变量的置换逻辑

当<copy>操作的 from-spec 和 to-spec 都选择了 WSDL 消息变量时，from-spec 消息变量值

必须被复制为 to-spec 的消息变量值。如果 from-spec 消息变量完全未被初始化则标准故障 bpel：uninitializedVariable 被抛出。如果 from-spec 消息变量被部分初始化则 from-spec 变量中未被初始化的部分将导致 to-spec 变量中相同部分变成未初始化。To-spec 消息变量最初的消息部分在<copy>操作后无法使用。

XML 数据条目替换表

当 from-spec（源）和 to-spec（目的）选择三种信息条目类型之一时，WS-BPEL 处理器必须使用下表中的置换规则：

Source\Destination	EII	AII	TII
EII	RE	RC	RC
AII	RC	RC	RC
TII	RC	RC	RC

置换逻辑表

- RE（置换-元素-属性）：
  - ✧ 和源的全部元素的拷贝一起在目的中置换元素，包括[children]和[attribute]属性。

    可选择的属性keepSrcElementName提供了更进一步的优化行为。[SA00042]只有当from-spec和to-spec的结果都是EII时属性可用，并且在其他情况下禁止被明确设置。WS-BPEL处理器可以通过表达式/查询语言的静态分析强制检查这个情况。如果在运行期间检测到违反情况，bpel：selectionFailure故障必须被抛出。
    - ■ 当 keepSrcElementName 属性设置为“no”时，最初目的元素的名字（比如说[namespace name]和[local name]属性）被作为结果元素的名字使用。这是默认值。
    - ■ 当 keepSrcElementName 属性设置为“yes”时，源元素名字作为结果目的元素的名字使用。

    当keepSrcElementName属性设为“yes”并且目的元素是基于元素变量的文档EII或者基于WSDL消息类型变量的基于元素部分，WS-BPEL处理器必须确认源元素的名字属于变量声明或WSDL部分定义里使用的目的元素的置换组。置换组的相关性是由WS-BPEL处理器明白的XML Schemas决定。[SA00094]WS-BPEL处理器可以通过表达式/查询语言的静态分析强制检查这个情况。如果在程序运行期间检测出违例情况，bpel：mismatchedAssignmentFailure故障必须被抛出。

- RC（置换-目录）
  - ✧ 得到源目录：
    - ■ 一旦信息条目从源中返回，TII 将根据它来计算。这个源目录 TII 是基于一连串的 CII，通常是基于文档请求（除非是表达式或查询中的分类（或排序 sorting）规范），从返回信息条目中取得。CII 被一起拷贝、连接，并且产生的结果值被赋值给 TII。这个同 XPath 1.0 string（）函数类似。
    - ■ 如果源有 xsi：nil=“true”的 EII，则 selectionFailure 故障必须被抛出。这个检查在 EII-to-AII 或者 EII-to-TII 拷贝期间执行。

- 置换目的目录
  - 如果目的是 EII，则所有的[children]属性（即便要）被移除且源目录 TII 作为 EII 的孩子被添加。
  - 如果目的是AII，则AII的值同来自源的TII置换。值必须是规格化的，与 XML 1.0 标准一致（见章节 3.3.3 属性值标准化：http://www.w3.org/TR/1998/REC-xml-19980210#AVNormalize).
  - 如果目的是 TII，则在目的中的 TII 与来自源的 TII 置换。
- 另外，下列的规则适用于：
  - To-spec 引用的信息条目必须是左值。在 XPath 1.0 数据模型中，TII 左值必须是文本节点。
  - 当 to-spec 选择 TII 作为左值且不属于 XSD 字符串类型（或由 XSD 字符串衍生出的类型）的 WS-BPEL 变量时，bpel:mismatchedAssignmentFailure 故障必须被抛出，并且下列情况之一作为来自 from-spec 的右值计算：
    - 有零个 CII 的 TII
    - [normalized value]为空字符串的 AII
    - 有零个 CII 作为其子节点的 EII，即是它的[children]和嵌套的[children]。请注意，将 XPath 1.0 string（）函数应用于这类 EII 将会产生一个空字符串。
  - 在 XPath 1.0 中，属性值不是文本节点。属性节点有一个与 XML 规格化属性值相关的字符串值，即 TII。

使用<copy>初始化变量

当<copy>操作中 to-spec 选择的目的没有被初始化时，即它不是一个完整的 WS-BPEL 变量或者消息部分时，目的必须在执行上述置换规则之前被初始化，如下列被应用的：

- 对于复杂类型和简单类型变量或消息部分，初始化结构框架由 DII 和匿名的文档元素 EII 组成
- 对于基于元素的变量或消息部分，初始化结构框架由 DII 和匹配变量声明里使用的元素名称的文档元素 EII 组成

这个初始化行为是原子级<assign>活动的主要部分。

在<copy>中处理非-XML Infoset 数据对象

简单类型变量和值可以表示为非 XML Infoset 数据对象，如在 XPath 1.0 中定义的布尔，字符串或浮点数。同样地，表达式也可以返回非 XML Infoset 数据对象，比如：

```
<from>number($order/amt) * 0.8</from>
```

为了能够贯彻应用上述置换规则，这样的非XML Infoset数据像TII一样被处理。这是通过将数据转换为字符串完成的，因为TII类似于一个字符串对象。尤其是当XPath 1.0 数据模型在 WS-BPEL 中使用时，“字符串（对象）”(http://www.w3.org/TR/1999/REC-xpath-19991116#function-string)强制转换必须用于把布尔或数字对象转换为字符串。如果<copy>的结果保持一致，WS-BPEL处理器可以跳过实际转换。

XML 名称空间保存

在<copy>操作中，来自源的[in-scope namespace]属性（类似于其他 XML Infoset 条目属性）必须在目的结果中被阻止。WS-BPEL 处理器可以使用名称空间明确的 XML 基础构造来维持 XML 名称空间一致性。

在某些 XML Schema 设计中，可以对属性或元素值使用 QName。当 TII 或 AII 包括一个通过 Schema 不明确的表达式/查询语言选择的 QName 时，它的数据模型将不能捕获 QName 值的名称空间属性。因此，XML 名称空间可能丢失。请注意 XPath 1.0 的 Schema 是不明确的。

例如，attrX的值为QName（“myPrefix：somename”）并且“foo：bar3”的值为另外的QName（“myPrefix：somename2”）。当“foo：bar2/@attrX”同XPath 1.0 数据模型的源一起被拷贝时，“myPrefix”的名称空间声明可能在目的中缺失。

```
<foo:bar1 xmlns:myPrefix="http://example.org"
   xmlns:foo="http://example.com ">
   <foo:bar2 attrX="myPrefix:somename" />
   <foo:bar3>myPrefix:somename2</foo:bar3>
</foo:bar1>
```

8.4.3 拷贝操作中的类型兼容性

[SA00043]为了使拷贝操作有效，from-spec和to-spec访问的数据必须是可兼容的类型。

下列情况被看做类型不兼容：

- From-spec 和 to-spec 的选择结果都是 WSDL 消息类型的变量，并且两个变量不是相同的 WSDL 消息类型（如果 Qname 相等则两种 WSDL 消息类型一致）。
- From-spec 的选择结果是 WSDL 消息类型变量而 to-spec 的结果不是，或者反之亦然（变量部分，变量部分的选择，或者端点引用不能被直接赋值给/从 WSDL 消息类型变量）。
- From-spec 的选择结果是 EII，to-spec 的选择结果是基于元素变量的文档 EII 或者基于 WSDL 消息类型变量的基于元素部分 I，keepSrcElementName 属性设置为“yes”且源元素的名字不属于目的的 subsitutionGroup（见章节 8.4.2 拷贝操作的置换逻辑）。

如果在赋值期间检测到不兼容的类型，则标准故障 bpel：mismatchedAssignmentFailure 必须被抛出。

8.4.4 赋值例子

假设下面复杂类型定义在名称空间“http://example.org/bpel/example”中：

```
<complexType name="tAddress">
   <sequence>
      <element name="number" type="xsd:int" />
      <element name="street" type="xsd:string" />
      <element name="city" type="xsd:string" />
      <element name="phone">
         <complexType>
            <sequence>
               <element name="areacode" type="xsd:int" />
               <element name="exchange" type="xsd:int" />
               <element name="number" type="xsd:int" />
            </sequence>
         </complexType>
      </element>
   </sequence>
</complexType>

<element name="address" type="tAddress" />
```

假设下面的 WSDL 消息定义在相同的目标名称空间中存在：

```
<message name="person" xmlns:x="http://example.org/bpel/example">
   <part name="full-name" type="xsd:string" />
   <part name="address" element="x:address" />
</message>
```

同样地，假定下面的 WS-BPEL 变量声明：

```
<variable name="c1" messageType="x:person" />
<variable name="c2" messageType="x:person" />
<variable name="c3" element="x:address" />
```

例子说明了把一个变量赋值给其他变量与将一个变量部分拷贝给相兼容元素类型的变量一样：

```
<assign>
   <copy>
      <from variable="c1" />
      <to variable="c2" />
   </copy>
   <copy>
      <from>$c1.address</from>
      <to variable="c3" />
   </copy>
```

```
</assign>
```

9.相关性

到目前为止提供的信息建议转交给业务程序服务的消息目标是收件人服务的 WSDL 端口。这只能是幻想因为，由它们特殊的实际情况，有状态的业务流程被用于执行与扩展交互历史一致的行为。因此，送到这样流程的消息不仅需要被送到正确的目的端口，也需要被送到正确的提供端口的业务程序的实例。创建新的业务程序实例的消息是一个特殊情况，在章节 5.5 可执行业务程序的生命周期 中描述。

在面向对象的世界，这样的有状态交互由对象引用调解，即本质提供到达指定对象（实例）其状态和交互历史正确的能力。这在高耦合的实现即正常依赖实现的架构中工作的更好。（This works reasonably well in tightly coupled implementations where a dependency on the structure of the implementation is normal.）在 Web 服务的低耦合世界中，这样的引用用法将会导致实现依赖集不稳固，不能在每个业务伙伴中将进化独立在业务程序实现细节之外。（the use of such references would create a fragile set of implementation dependencies that would not survive the independent evolution of business process implementation details at each business partner.）在这个世界中，答案依赖于定义伙伴间线程级交互的业务数据和通信协议域名，以及为了避免特殊实现标记的用法比如随时可能的路由选择。

设想买家发送订购单给卖家的供应链情况。假设买家和卖家由一个稳定的业务关系并且被静态地设置为发送同订单交互有关的文档给相关的同 WSDL 服务端口相连的 URI。卖家需要返回一个对订单的确认信息，并且这个确认信息必须选择正确的到买家的业务程序实例路径。很明显，完成这个工作的标准机制就是在订单消息中携带一个业务标记（如订单号），将其拷贝到确认消息中以便关联。标记可以在消息包封中，在头部，或者在业务文件（订单）本身中。前面任何一种情况，在相关消息中精确的位置和标记类型是固定的并且实例独立。只有标记的值是实例独立的。因此，各个消息中相关标记的结构和位置可以在业务程序描述中声明。下面描述的 WS-BPEL 相关集的概念提供了这个特征。说明信息允许符合 WS-BPEL 的基础结构使用相关标记来提供实例自动化选择路由。

相关声明依赖于说明性的消息属性。属性是简单的由查询表示的消息里的“域”。只有当良好地定义消息结构时（如使用 XML Schema 来描述），这才是可能的。相关标记的用法受限于这样描述的消息部件。这些消息实际的线的格式可以是非 XML，如 EDI 展开文件，基于端口的不同绑定。

9.1 消息相关性

在它的生命周期中，业务流程实例典型地会保存一个或多个与伙伴的会话。会话可能基于复杂的传送基础结构即用会话特性的某种形式来关联会话中包括的消息并且不需要指定任何业务流程里的相关信息即可让它们自动选择到正确的程序实例路线。尽管如此，很多情况下会话包括大于两个群或使用相关标记直接嵌入于正在被交换的应用程序数据中的轻量传

输基础结构。在这样的情况下，经常需要提供额外的应用程序级机制来匹配业务流程实例预期的消息和会话。

相关性样式可以非常复杂。特殊的相关标记集的使用通常不会生成程序实例或跨越伙伴间的完整交互，但可以生成部分交互。相关交换可以嵌套和覆盖，并且消息可以携带几种相关标记集。比如买家可以通过发送订购单（PO）消息及使用嵌套在消息中的订购单号码作为相关标记来启动同卖家的相关交换。PO 号码被卖家用于确认消息。卖家稍后可能发送一个携带 PO 号码的发票消息，使它和最初的 PO 相关联，并且携带一个发票号以便将来相关的付款消息只需携带发票号作为相关标记即可。发票消息因此携带两个不同的相关标记并且参与两个重叠的相关消息的交换。

WS-BPEL 标记相关性情况通过提供说明性的机制来指定在程序实例中相关的操作组。相关标记集被定义为在相关组中所有消息共享的属性集。这样的一个属性集被称作相关集。

```
<correlationSets>?
   <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>
```

<correlationSet>可以在程序或作用域元素中声明，某种意义上类似于变量声明。[SA00044]<correlationSet>的名字在同一个立即附属作用域中必须是唯一的。这个要求必须被静态地强制。使用<correlationSet>遵循普通的词法域规则。

程序的<correlationSet>在程序初是未初始化状态。作用域的<correlationSet>在其附属的作用域开始是未初始化状态。请注意作用域如果被包含在可重复的构造器或事件处理器中，在其生命周期内可能不止一次启动和结束它们的行为。在这种情况下，<correlationSet>启动语义应用于作用域的每个实例。

<correlationSet>比起变量更类似于一个有边界的常量。<correlationSet>的绑定值通过特定的有记号的发送或接收消息操作触发。<correlationSet>在其所属作用域的生命周期内只能被初始化一次。一旦被初始化，<correlationSet>必须保持它的值，不管变量是否更新。因此，程序的<correlationSet>在程序实例的生命周期内最多只能被初始化一次。它的值一旦被初始化，可以被看做业务程序实例的特征。作用域的<correlationSet>实例每次在相应的作用域启动时对绑定可用。

在多方业务会话中，每个相关消息交换里的参与程序要么像发起者要么像交换的追随者一样活动。发起者程序发送第一个消息（如同操作调用的一部分）启动会话，并且因此来定义在<correlationSet>里标记会话的属性值。其它参与者全部成为追随者，通过接收一个提供<correlationSet>属性值的输入报文来在会话中绑定它们的<correlationSet>。发起者和追随者都要在活动初始化<correlationSet>时，在各自的组中标记它们的第一个活动。

9.2 声明和使用相关集

相关性可以在每个通信活动中（<receive>，<onMessage>，<onEvent>，<reply>和<invoke>）

使用。WS-BPEL 没有采取消息的复杂会话式传输协议用法。在这样的协议使用的地方，WS-BPEL 中相关性的显示用法可以归纳为那些建立会话式连接的活动。这些协议机制可以与或者不与任何明确的相关性一起使用。

[SA00045]<correlationSet>中使用的属性必须使用XML Schema简单类型来定义。这个约束必须是静态强制的。每个<correlationSet>是一个已命名的属性组，一起用来识别会话。一个被给出的消息可以携带匹配或者初始化一个或多个相关集的信息。

相关集规范在<invoke>，<receive>和<reply>活动中使用（见章节 10.3 调用 Web 服务操作及 10.4 提供 Web 服务操作-Receive 和 Reply），在<pick>活动的<onMessage>分支以及<eventHandlers>的<onEvent>变量体中使用（见章节 11.5 Pick 及 12.5.1 消息事件）。这些<correlation>规范通过名字识别相关集并且用于指出消息里哪个相关集正在被发送和接收。<correlation>规范上的初始化属性用于指出相关集是否正在被初始化。

在相关集被初始化后，相关集的属性值在携带相关集以及在对应的作用域中出现直到它完成的所有操作中的所有消息必须是相等的。这个相关性的一致性约束在所有 initiate 值中必须注意。合法的 initiate 属性值有：“yes”，“join”，“no”。initiate 的缺省值是“no”。

- 当 initiate 属性值设为“yes”，则有关活动必须试图初始化相关集。
  - ✧ 如果相关集已经被初始化，则标准故障 bpel：correlationViolation 必须被抛出。
- 当 initiate 属性值设为“join”，如果相关集尚未被初始化，则有关活动必须试图初始化相关集。
  - ✧ 如果相关集已经初始化并且违反相关性的一致性约束，则标准故障：bpel：correlationViolation 必须被抛出。
- 当 initiate 属性值设为“no”或者没有明确的指出，则禁止有关活动试图初始化相关集。
  - ✧ 如果相关集没有被预先初始化，标准故障 bpel：correlationViolation 必须被抛出。
  - ✧ 如果相关集已经被初始化并且违反相关性的一致性约束，标准故障 bpel：correlationViolation 必须被抛出。

上面描述了相关集的启动约束。如果在消息输出活动中（如<invoke>）使用多个相关集，则必须注意相关集使用的启动约束和一致性约束。如果在消息输入活动（IMA）中（如<receive>）使用多个相关集，则必须注意相关集中使用的启动约束。如果任何一个相关集没有遵守上面的约束，则标准故障 bpel：correlationViolation 必须被抛出。

当 initiate=“no”的 IMA 中使用多个相关集时，消息必须匹配所有这样的相关集——在给出的程序实例中消息被转送给活动。当消息中的相关集没有匹配程序实例中已经初始化的相关集或如果相关集未初始化，则消息禁止被转送给 IMA.因此，相关集的一致性约束检查不适用于 IMA.

如果 Web 服务的输入请求消息到达并且（1）没有正在运行的程序实例可以被消息相关集机制识别以及（2）所有正在引用 Web 服务操作的输入消息活动的 creatInstance 属性值设为“no”，则这个情况不在本规范的作用域之内因为没有程序实例能够处理它。

当 bpel：correlationViolation 故障因为请求/响应操作中的违规响应被<invoke>活动抛出时，响应必须在 bpel：correlationViolation 被抛出之前接收。在其他任何抛出 bpel：correlationViolation 故障的情况下，导致故障的消息禁止被发送或接收。

为了检索来自消息的相关值，处理器必须找到匹配的<vprop:propertyAlias>并将其应用于消息中。<vprop:propertyAlias>被看做同消息匹配如下：

1. <vprop：propertyAlias>定义中使用的 messageType 属性值匹配同消息相关联的 WSDL 消息类型的 QName；或
2. 消息包括<vprop：propertyAlias>中使用的元素以及元素属性值定义的单个部分的 WSDL 消息类型相关联的消息匹配定义 WSDL 部分的元素的 QName。

这个匹配<vprop：propertyAlias>约束必须被静态强制。如果基于<vprop:propertyAlias>的 messageType 和元素都同消息匹配，则基于<vprop:propertyAlias>的 messageType 必须有优先级。基于<vprop:propertyAlias>的类型从来不用考虑检索相关值。这些匹配规则只应用于检索相关值并且对 from-spec，to-spec 或 bpel：getVariableProperty 中使用的选择<vprop:propertyAlias>无效。

<vprop:propertyAlias>的应用程序导致包括任何除了一个信息条目以及/或 CII 的收集响应的情况时，bpel：selectionFailure 故障必须被抛出。

对于<invoke>，当调用的操作是请求/响应操作时，<correlation>规范上的pattern属性用于指出是否将相关性应用于输出消息（“request”），输入消息（“response”），或者两者都（”request-response”）。[SA00046]<invoke>中使用的pattern属性要求请求-响应操作，并且在单向操作被调用时禁用。任何违规情况的必须在静态分析期间找出。因为有这个，业务流程可以确保被送出的消息带有预期的相关标记。

```
<correlations>
   <correlation set="NCName"
      initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
</correlations>
```

以下是相关性的一个扩展例子。以定义四个消息属性开始：customerID,orderName,vendorID 以及 invoiceNumber。所有的这些属性被定义为文档的名称空间"http://example.com/supplyCorrelation"的一部分。

```
<wsdl:definitions name="properties"
   targetNamespace="http://example.com/supplyCorrelation"
   xmlns:tns="http://example.com/supplyCorrelation" ...>

   <!-- define correlation properties -->
   <vprop:property name="customerID" type="xsd:string" />
   <vprop:property name="orderNumber" type="xsd:int" />
```

```
   <vprop:property name="vendorID" type="xsd:string" />
   <vprop:property name="invoiceNumber" type="xsd:int" />

</wsdl:definitions>
```

这些属性是带有 XML Schema 简单类型的名字。它们是抽象的以致于在变量中的具体值需要单个地指定（见章节 7.变量属性）。例子通过定义订单以及发票消息继续并且通过使用别名概念来映射抽象属性给选择识别的消息数据里的字段。

```
<wsdl:definitions name="correlatedMessages"
   targetNamespace="http://example.com/supplyMessages"
   xmlns:tns="http://example.com/supplyMessages"
   xmlns:cor="http://example.com/supplyCorrelation"
   xmlns:po="http://example.com/po.xsd" ...>

   <wsdl:import namespace="http://example.com/supplyCorrelation"
      location="..." />

   <!-- define schema types for PO and invoice information -->
   <wsdl:types>
      <xsd:schema targetNamespace="http://example.com/po.xsd">
         <xsd:complexType name="PurchaseOrder">
            <xsd:element name="CID" type="xsd:string" />
            <xsd:element name="order" type="xsd:int" />
            ...
         </xsd:complexType>
         <xsd:complexType name="PurchaseOrderResponse">
            <xsd:element name="CID" type="xsd:string" />
            <xsd:element name="order" type="xsd:int" />
            <xsd:element name="VID" type="xsd:string" />
            <xsd:element name="invNum" type="xsd:int" />
            ...
         </xsd:complexType>
         <xsd:complexType name="PurchaseOrderRejectType">
            <xsd:element name="CID" type="xsd:string" />
            <xsd:element name="order" type="xsd:int" />
            <xsd:element name="reason" type="xsd:string" />
            ...
         </xsd:complexType>
         <xsd:complexType name="InvoiceType">
            <xsd:element name="VID" type="xsd:string" />
            <xsd:element name="invNum" type="xsd:int" />
         </xsd:complexType>
         <xsd:element name="PurchaseOrderReject"
```

```
          type="po:PurchaseOrderRejectType" />
       <xsd:element name="Invoice" type="po:invoiceType" />
    </xsd:schema>
</wsdl:types>

<wsdl:message name="POMessage">
   <wsdl:part name="PO" type="po:PurchaseOrder" />
</wsdl:message>
<wsdl:message name="POResponse">
   <wsdl:part name="RSP" type="po:PurchaseOrderResponse" />
</wsdl:message>
<wsdl:message name="POReject">
   <wsdl:part name="RJCT" element="po:PurchaseOrderReject" />
</wsdl:message>
<wsdl:message name="InvMessage">
   <wsdl:part name="IVC" element="po:Invoice " />
</wsdl:message>

<vprop:propertyAlias propertyName="cor:customerID"
   messageType="tns:POMessage" part="PO">
   <vprop:query>CID</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:orderNumber"
   messageType="tns:POMessage" part="PO">
   <vprop:query>Order</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:customerID"
   messageType="tns:POResponse" part="RSP">
   <vprop:query>CID</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:orderNumber"
   messageType="tns:POResponse" part="RSP">
   <vprop:query>Order</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:vendorID"
   messageType="tns:POResponse" part="RSP">
   <vprop:query>VID</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:invoiceNumber"
   messageType="tns:POResponse" part="RSP">
   <vprop:query>InvNum</vprop:query>
</vprop:propertyAlias>
<vprop:propertyAlias propertyName="cor:vendorID"
   messageType="tns:InvMessage" part="IVC">
```

```
      <vprop:query>VID</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:invoiceNumber"
      messageType="tns:InvMessage" part="IVC">
      <vprop:query>InvNum</vprop:query>
   </vprop:propertyAlias>
   ...
</wsdl:definitions>
```

最后，在单独的 WSDL 文档中定义被使用的 portType，。

```
<wsdl:definitions name="purchasingPortType"
   targetNamespace="http://example.com/purchasing"
   xmlns:smsg="http://example.com/supplyMessages"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <wsdl:import namespace="http://example.com/supplyMessages"
      location="..." />

   <wsdl:portType name="PurchasingPT">
      <wsdl:operation name="Purchase">
         <wsdl:input message="smsg:POMessage" />
         <wsdl:output message="smsg:POResponse" />
         <wsdl:fault name="tns:RejectPO" message="smsg:POReject" />
      </wsdl:operation>
      <wsdl:operation name="PurchaseRequest">
         <wsdl:input message="smsg:POMessage" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:portType name="BuyerPT">
      <wsdl:operation name="PurchaseResponse">
         <wsdl:input message="smsg:POResponse" />
      </wsdl:operation>
      <wsdl:operation name="PurchaseReject">
         <wsdl:input message="smsg:POReject" />
      </wsdl:operation>
   </wsdl:portType>

</wsdl:definitions>
```

属性以及它们对订单的映射以及发票消息都会在下列相关性例子中使用：

```
<correlationSets xmlns:cor="http://example.com/supplyCorrelation">
```

```
   <!-- Order numbers are particular to a customer,
        this set is carried in application data -->
   <correlationSet name="PurchaseOrder"
      properties="cor:customerID cor:orderNumber" />

   <!-- Invoice numbers are particular to a vendor,
      this set is carried in application data -->
   <correlationSet name="Invoice"
      properties="cor:vendorID cor:invoiceNumber" />

</correlationSets>
```

消息可以携带一个或多个相关集的标记。第一个例子显示了订单接收单向输入请求和涉及被发送发票单向响应的确认之间的交互。两个活动都使用 PurchaseOrder <correlationSet>以致于单向响应通过相关集验证将买家的请求联系起来。<receive>活动初始化 PurchaseOrder <correlationSet>。买家因此成为领导者并且接收业务程序成为<correlationSet>的跟随者。<invoke>活动发送单向响应也初始化了一个名为 Invoice 的新的<correlationSet>。业务流程是这个相互交换的领导者且买家是跟随者。因此响应消息是两个单独会话的一部分，并且在两者之间建起了一座桥。

下面，prefix SP:表示名称空间"http://example.com/purchasing".

```
<receive partnerLink="Buyer" portType="SP:PurchasingPT"
   operation="PurchaseRequest" variable="PO">

   <correlations>
      <correlation set="PurchaseOrder" initiate="yes" />
   </correlations>
</receive>
...
<invoke partnerLink="Buyer" portType="SP:BuyerPT"
   operation="PurchaseResponse" inputVariable="POResponse">

   <correlations>
      <correlation set="PurchaseOrder" initiate="no" />
      <correlation set="Invoice" initiate="yes" />
   </correlations>
</invoke>
```

二选择一的，响应可能成为一个障碍（比如作为一个”不在存储中”的消息），在这种情况下 PurchaseOrder <correlationSet>相关的会话不会触发新的与 Invoice 相关的会话。Pattern 属性没有被使用，因为操作是单向的。

```
<invoke partnerLink="Buyer" portType="SP:BuyerPT"
```

```
   operation="PurchaseReject" inputVariable="POReject">

   <correlations>
      <correlation set="PurchaseOrder" initiate="no" />
   </correlations>
</invoke>
```

从买家业务流程的透视图中，相关集被定义为用于发送订单的单向调用活动用以及用于接收订单的接收活动来分别响应或拒绝消息。

```
<invoke partnerLink="Seller" portType="SP:PurchasingPT"
   operation="PurchaseRequest" variable="PO">

   <correlations>
      <correlation set="PurchaseOrder" initiate="yes" />
   </correlations>
</invoke>
...
<pick>
   <onMessage partnerLink="Seller" portType="SP:BuyerPT"
      operation="PurchaseResponse" variable="POResponse">
      <correlations>
         <correlation set="PurchaseOrder" initiate="no" />
         <correlation set="Invoice" initiate="yes" />
      </correlations>
      ...
      <!-- handle the response message -->
   </onMessage>

   <onMessage partnerLink="Seller" portType="SP:BuyerPT"
      operation="PurchaseReject" variable="POReject">
      <correlations>
         <correlation set="PurchaseOrder" initiate="no" />
      </correlations>
      ...
      <!-- handle the reject message -->
   </onMessage>
</pick>
```

二选择一的，如果请求-响应订单操作在买家业务流程中使用，则相关集分别为请求和响应调用活动指定。来自卖家的 PO 通过故障消息发送。

```
<invoke partnerLink="Seller" portType="SP:PurchasingPT"
   operation="Purchase" inputVariable="sendPO"
```

```
   outputVariable="getResponse">

   <correlations>
      <correlation set="PurchaseOrder" initiate="yes"
         pattern="request" />
      <correlation set="Invoice" initiate="yes" pattern="response" />
   </correlations>

   <catch faultName="SP:RejectPO" faultVariable="POReject"
      faultMessageType="smsg:POReject">
      ...
      <!-- handle the fault -->
   </catch>
</invoke>
```

<invoke>可以由两个消息组成：输出请求消息和输入回复消息。对每个消息适用的<correlationSet>必须单个考虑，因为它们是可以相异的。在这种情况下PurchaseOrder相关性将应用于启动它的输出请求，Invoice相关性将应用于输入回复并且被回复初始化。因为PurchaseOrder相关性是由输出消息初始化，则买家是那个相关性的领导者。尽管如此，买家也是Invoice相关性的跟随者因为Invoice的相关属性值由买家接收的卖家的回复消息初始化。

10.基本活动

WS-BPEL活动执行了程序逻辑。活动被分为两个类：基本类和架构类。基本活动即描述程序行为的基本步骤。架构活动将控制流逻辑转换成代码，以及因此递归地包括其他基本或/及架构活动。架构活动在章节11.架构活动中描述。

10.1 所有活动的标准属性

每个活动有两个可选择的标准属性：活动的 name 以及指出当连接故障发生时是否抑制的 suppressJoinFailure（见章节 5.2 业务流程定义的架构）。WS-BPEL 语言扩展性允许添加其他合法的名称空间属性。Name 属性用于向活动提供程序可处理的名字。WS-BPEL 只标志性的使用作用域活动的名字。见章节 12.4.3 调用赔偿处理器 有关 name 属性的唯一性约束。有关 suppressJoinFailure 属性的全面讨论，见章节 11.6 并行以及控制依赖流程-Flow。

```
name="NCName"?
suppressJoinFailure="yes|no"?
```

10.2 所有活动的标准元素

每个活动有可选择的容器<source>以及<targets>，分别包括了标准元素<source>和<target>。WS-BPEL 语言扩展性允许这些通过添加合法的名称空间元素来扩展。这些源和目的，元素通过链接来确立同步关系（见章节 11.6 并行以及控制依赖流程-Flow）。

```
<targets>?
   <joinCondition expressionLanguage="anyURI"?>?
      bool-expr
   </joinCondition>
   <target linkName="NCName" />+
</targets>

<sources>?
   <source linkName="NCName">+
      <transitionCondition expressionLanguage="anyURI"?>?
         bool-expr
      </transitionCondition>
   </source>
</sources>
```

10.3 调用Web服务操作-Invoke

<invoke>活动用于调用服务提供者提供的 Web 操作（见章节 6.伙伴连接类型，伙伴链接以及端点引用）。典型的应用是在服务上调用一个操作，这被看做基本的活动。<invoke>活动可以附属于其他的活动，嵌入于会话处理器和故障处理器中，如下所示。操作可以是请求-响应或者单向操作，对应于 WSDL 1.1 操作定义。WS-BPEL 对两者使用相同的基本语法，不过请求-响应的情况有一些额外的选项。

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"?
         pattern="request|response|request-response"? />+
   </correlations>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
```

```
      faultMessageType="QName"?
      faultElement="QName"?>*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
   <compensationHandler>?
      activity
   </compensationHandler>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</invoke>
```

单向操作只要求inputVariable（或者它的等效<toPart>元素），因为不期望响应作为操作的一部分（见章节 10.4 提供Web服务操作-Receive和Reply）。请求-响应调用需要inputVariable（或者其等效元素<toPart>）和outputVariable（或者其等效元素<fromPart>）。如果WSDL消息定义没有包括任何一个部分，则相关属性，inputVariable或者outputVariable，可以被省略，[SA00047]并且<fromPart>或<toPart>必须被省略。零个或多个correlationSets可以被指定给相关的在伙伴方带有一个有状态服务的业务程序实例（见章节 9 相关性）。

如果一个<invoke>活动用在 partnerRole EPR 未被初始化的 partnerLink 上，则 bpel：uninitialPartnerRole 故障必须被抛出。

使用请求-响应的情况下，操作可能返回一个 WSDL 故障消息。这导致 WS-BPEL 中通过相应端口类型和故障名称的目标名称空间而形成的 QName 的故障识别。为了确保故障标识的一致性，必须遵循统一命名机制即使它不匹配 WSDL 的故障命名模型。WSDL 1.1 不要求已定义服务操作的名称空间中的故障名称唯一。因此，在 WSDL 1.1 中必须指定端口类型名，操作名以及故障名来唯一识别故障。使用 WSDL 1.1 的模式将会限制使用故障识别以及处理调用故障的处理机制的能力。在 WSDL 中可以定义声明大于一个使用相同数据类型故障的操作。某些 WSDL 绑定不会为 WS-BPEL 处理器提供足够的信息来确定哪个故障发生。在这种情况下，WS-BPEL 处理器必须自己选择故障：

- 匹配发送的数据并且
- 在操作定义的词法要求中第一个发生。

这个要求的结果是一个使用基于 faultName 的<catch>并且处理这样的操作定义的程序，当配置不同绑定时可以有不同的行为。

WS-BPEL 中的故障只根据故障名和可选择的故障数据来定义。这就意味着，如故障产生

于消息发送活动（不同于<throw>活动（见章节 10.6 发送内部故障信号）或者系统故障），则没有必要记住端口类型或操作，当故障被接收时使用消息活动。结果，所有的故障共享一个公用的名字，在同一名称空间中定义并且共享同一个数据类型（或者省略），在 WS-BPEL 中无法辨别。有特定名字的故障可以同多个变量类型相关联。WS-BPEL 中<catch>构造促进区别名字相同，但是带有的消息和变量类型不同的故障。有关故障处理和<catch>的详细信息，见章节 12.5 故障处理器。

<invoke>活动可以同其他担当补偿作用的活动相关联。因此，<compensationHandler>可以被明确地或者通过缺省的附属作用域的<compensationHandler>调用（见章节 12.作用域 及章节 12.3 业务流程中的错误处理）。

从语义上说，局部故障处理器及/或局部补偿处理器的规范相当于存在一个直接附属于提供这些处理器的<invoke>活动的隐式<scope>活动。隐式<scope>活动采用其附属的<invoke>活动名字以及它的 suppressJoinFailure 属性，连同它的<sources>以及<targets>。举例如下：

```
<invoke name="purchase"
   suppressJoinFailure="yes"
   partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
   <targets>
      <target linkName="linkA" />
   </targets>
   <sources>
      <source linkName="linkB" />
   </sources>
   <catch faultName="SP:rejectPO">...</catch>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation" />
   </compensationHandler>
</invoke>
```

相当于：

```
<scope name="purchase" suppressJoinFailure="yes">
   <targets>
      <target linkName="linkA" />
   </targets>
```

```
   <sources>
      <source linkName="linkB" />
   </sources>
   <faultHandlers>
      <catch faultName="SP:rejectPO">...</catch>
   </faultHandlers>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation" />
   </compensationHandler>

   <invoke name="purchase"
      partnerLink="Seller"
      portType="SP:Purchasing"
      operation="Purchase"
      inputVariable="sendPO"
      outputVariable="getResponse" />
</scope>
```

在这个例子中，如果必要的话，调用订单操作可以被补偿，通过调用 CancelPurchase 操作（见章节 12.4 补偿处理器详细资料）。

[SA00048]当<invoke>活动中使用可选择属性inputVariable和outputVariable时，inputVariable和outputVariable引用的变量必须是其Qname分别匹配操作中使用的input和output消息类型的messageType变量，以下情况除外：如果<invoke>活动中使用的WSDL操作使用明确包括某部分自身使用一个元素定义的消息，则用于定义这部分的相同元素类型的变量可以分别通过inputVariable和outputVariable引用。使用在预先定义好的环境中的变量的结果必须等价于声明匿名的基于同WSDL相关联的消息类型的临时WSDL消息变量。匿名临时WSDL消息变量和元素变量之间的元素数据拷贝作为带有一个<copy>操作的其keepSrcElementName属性值设为”yes”的单个虚拟<assign>一样运行。虚拟的<assign>必须像真正的<assign>一样遵循相同的语义以及抛出相同的故障。在inputVariable的情况下，属性引用的变量值将会被设置为匿名临时WSDL消息变量的部分值。在outputVariable的情况下，临时WSDL消息变量中接收的部分值将会被设置为属性引用的变量值。

10.3.1 映射WSDL消息部分

<toParts>元素提供交替式用来明确创建多部件的来自WS-BPEL变量内容的WSDL消息。通过使用<toParts>元素，匿名临时的WSDL变量在基于有关WSDL操作的输入消息上被声明。<toPart>元素，作为一个组，如同单个虚拟的<assign>一样运行，每个<toPart>作为一个<copy>。WSDL消息定义中每个部件最多存在一个<toPart>。每个<copy>操作从fromVariable属性指出的变量中拷贝数据到<toPart>元素的part属性中引用的匿名临时WSDL变量的部件

中（见章节 8.4 赋值活动）。如果<copy>操作正在赋值元素变量给元素部件则操作的keepSrcElementName选项设为“yes”。虚拟<assign>必须像真正的<assign>一样遵循相同的语义和故障。[SA00050]当<toParts>存在，在WSDL消息定义中每个部件都要有一个<toPart>；命令在哪个部件被指定是不相关的。<toPart>元素不明确地表示部件会导致<invoke或<reply>活动使用目标匿名WSDL变量中未初始化的部件。这种缺少<toPart>元素的流程必须在静态分析期间被拒绝。[SA00051]inputVariable属性禁止在包含<toPart>元素的<invoke>活动上使用。

<fromPart>元素类似于<toPart>元素。<fromPart>元素用于从输入多部件 WSDL 消息中检索数据并且将其放入单个的 WS-BPEL 变量中。当 WSDL 消息在使用<fromPart>元素的<invoke>活动上被接收，则消息被放入相关 WSDL 操作的输入消息指定类型的匿名临时 WSDL 变量。<fromPart>元素，作为一个组，如同单个虚拟的<assign>一样运行，每个<fromPart>如同一个<copy>。每个<copy>操作在<fromPart>的 part 属性引用的匿名临时 WSDL 变量中拷贝数据至 toVariable 属性指出的变量里。如果<copy>操作拷贝元素部件给元素变量则操作的 keepSrcElementName 选项设为”yes”。虚拟的<assign>必须像真正的<assign>一样遵循相同的语义和故障（见章节 8.4 赋值活动）。当<invoke>中存在<fromPart>，不要求 WSDL 消息定义的每个部件里有一个<fromPart>，也不用指定相关命令具体在哪个部件里（nor is the order in which parts are specified relevant.）。

使用 inputVariable 表单代替<toParts>表单，不会在被使用的 outputVariable 或<fromParts>表单上创建约束，反之亦然。类似的，使用 outputVariable 表单代替<fromParts>表单也是如此，反之亦然。

虚拟<assign>作为<fromPart>或<toPart>元素的结果创建作为<invoke>活动的作用域的部件出现，因此任何抛出的故障都由<invoke>的嵌入式故障处理器捕获。<toPart>或者<fromPart>元素可以同只拥有单个部件的 WSDL 消息一起使用。

见章节 9.相关性 中有关相关性语义的说明。

10.4 提供Web服务操作——Receive和Reply

业务流程通过输入消息活动（IMA-<receive>,<pick>,<onEvent>）以及相应的<reply>活动提供服务给它的伙伴。本章介绍了<receive>和<reply>活动的详细情况（见章节 11.5 选择性事件处理-Pick 以及 12.7.1 消息事件 中的<onEvent>）。

<receive>活动指定了partnerLink，其中包括用来接收消息的myRole，portType（可选择的）以及期望伙伴调用的operation。当处理<receive>活动时，不会使用partnerLink中partnerRole的值。另外，<receive>通过使用variable属性指定变量，用来接收消息数据。Variable属性可以选择是否使用<fromPart>元素。<receive>活动上使用<fromPart>元素的语义和语法与章节 10.3.1 映射WSDL消息部件中指定的<invoke>活动相同。[SA00055]如果<fromPart>元素在<receive>活动上使用，则variable属性禁止在相同的活动上使用。如果WSDL消息定义不包括任何部件，则相关联的variable属性可以被省略，[SA00047]且<fromParts>构造器必须被省略。<receive>活动的语法摘要如下：

```
<receive partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   variable="BPELVariableName"?
   createInstance="yes|no"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</receive>
```

<receive>活动在业务流程的生命周期执行一个任务。例示WS-BPEL中的业务流程的唯一途径是creatInstance属性设为“yes”的<receive>活动（或<Pick>活动）（见章节 11.5 选择性事件处理——Pick）。这个属性的缺省值是“no”。开始活动是其creatInstance=“yes”或<extensionActivity>子元素被注释的<receive>或<pick>活动。为了<extensionActivity>子元素符合开始活动的要求，必须展示接收输入消息的行为。[SA00056]非开始活动除了<scope>，<flow>，<sequence>或<extensionActivity>活动外必须在开始活动有一个控制依赖（见章节 12.5.2 缺省补偿命令 有关控制依赖的定义）。如果<extensionActivity>在开始活动没有控制依赖则<extensionActivity>子元素必须是一个包括开始活动的框架活动。这个框架活动必须同WS-BPEL程序实例化模型一致，即它禁止是可重复的活动。如果<extensionActivity>子元素自身是个开始活动或包括开始活动则<extensionActivity>子元素的名称空间必须声明mustUnderstand=“yes”。其他的语义约束，见章节 5.3 语言扩展性。执行活动的逻辑命令由静态分析决定。messageExchange属性的解释，见章节 10.4.1 消息交换 中<reply>活动的描述。

有多个开始活动是可能的。初始化开始活动是引起特殊的程序实例被初始化的开始活动。如同在章节 12.作用域中指定的，初始化开始活动必须在开始活动允许被执行之前完成执行。这就允许开始活动中的输入消息创建程序实例因为这些消息达到的命令是不可预知的。[SA00057]如果程序有多个带有相关集的开始活动则所有这些活动必须共享至少一个公共的相关集并且所有在活动上定义的公共相关集的initiate属性值设为“join”（见章节 9 相关性）。一致性执行必须确保只有一个匹配单个程序实例的输入消息例示了业务流程（常常必须是第一个到达的消息，但这是由执行决定的）。其他并行初始化集中的输入消息必须被转交给相应的已经创建实例的<receive>活动。

下面的例子是不被允许的，因为<assign>活动不是开始活动：

```
<flow>
   <!-- this example is illegal -->
```

```
   <receive ... createInstance="yes" />
   <assign ... />
</flow>
```

下面的例子是允许的，因为<assign>活动不会优先于或者同时与<receive>活动执行：

```
<flow>
   <links>
      <link name="RecvToAssign" />
   </links>
   <receive ... createInstance="yes">
      <sources>
         <source linkName="RecvToAssign" />
      </sources>
   </receive>
   <assign>
      <targets>
         <target linkName="RecvToAssign" />
      </targets>
      ...
   </assign>
</flow>
```

[SA00058]在<receive>或<reply>活动中，variable属性引用的变量必须是其QName匹配操作中使用的input（对于<receive>）或output（对于<reply>）消息类型的messageType变量。除了以下情况：如果WSDL操作使用其部件自身使用元素定义的消息，则与用来定义部件的相同元素类型的WS-BPEL变量可以通过<receive>或<reply>活动的variable属性被引用。使用预先定义环境的WS-BPEL变量的结果必须等效于声明一个基于相关WSDL消息类型的匿名临时WSDL消息变量。匿名临时WSDL消息变量和元素变量之间的元素数据拷贝作为带有一个<copy>操作的其keepSrcElementName属性值设为”yes”的单个虚拟<assign>一样运行。虚拟的<assign>必须像真正的<assign>一样遵循相同的语义以及抛出相同的故障。在<receive>活动的情况下，输入部件值将会被设置为variable属性引用的变量值。在<reply>活动的情况下，variable属性引用的变量值将会设置为被发送的匿名临时WSDL消息变量中的部件值。在<reply>发送故障时，应用相同的逻辑。

<receive>活动中的<fromParts>元素交替使用以指出来自接收消息中的数据是否直接拷贝给来自相应匿名 WSDL 消息变量的 WS-BPEL 变量。类似的，<toParts>元素交替使用以使来自 WS-BPEL 变量中的数据直接拷贝给<reply>活动使用的匿名 WSDL 消息（见章节 10.3.1 映射 WSDL 消息部件 有关使用这两个元素的规则）。

<receive>是模块化活动因为它直到程序实例接收到匹配的消息才会完成。业务程序实例禁止为同一个 partnerLink，portType，operation 以及 correlationSet（s）（包括 WS-BPEL 处理器指定的相关性）同时启动两个或多个<receive>活动。如果在业务流程实例执行期间，两个或多个相同 partnerLink，operation 以及 correlationSet（s）的接收活动实例同时启动，则标

准故障 bpel：conflictingReceive 必须被抛出（注意 bpel：conflictingReceive 不同于 bpel：conflictingRequest，见章节 10.4.1 消息交换）。接收活动实例可能在 partnerLink 和 correlationSet（s）相异的操作上，然而 WS-BPEL 处理器在运行时不能辨别。在这些情况下,WS-BPEL 处理器应该抛出 bpel：conflictingReceive 故障。如果业务程序实例为相同 partnerLink，portType，operation 但是 correlationSet（s）不同，同时启动两个或多个 IMA，且多个活动的相关性匹配输入请求消息，则 bpel：ambiguousReceive 标准故障必须被所有与相关集输入消息匹配的 IMA 抛出。为了这些约束，<pick>和<onEvent>事件处理器中的<onMessage>子句等效于<receive>（见章节 11.5 选择性事件处理-Pick 以及 章节 12.7.1 消息事件）。

紊乱情况（Race conditions）可能发生在业务程序执行中。以特殊程序实例为目标的消息可能在对应的<receive>活动启动之前到达。比如，假设循环接收一连串消息的程序，所有的消息使用同一个相关性。在运行时，消息将会到达且不依赖于循环的迭代次数。事实上是相关性已经被初始化，不管怎样，应该启动运行时刻的引擎并且接发消息至平台来识别这些消息同程序实例相关，以及适当地处理那些消息。另一个例子是程序可能调用远程服务稍后为预期的回调消息启动一个相关集。因种种理由，回调消息可能在相应的<receive>活动开始前到达。到达消息中的相关数据应该启动引擎来识别消息是否以这个程序实例为目标。程序引擎可以使用不同的机制来处理这些紊乱情况。本规范没有授权任何特殊机制。消息传送机制的详细情况不在本规范的作用域之内。尽管如此，WS-BPEL 处理器应该根据优先消息发送和运输机制的服务质量来将消息传送至程序实例。为了处理紊乱情况，<pick>和<onEvent>事件处理器中的<onMessage>子句等效于一个接收活动（见章节 11.5 选择性事件处理-Pick 和 12.7.1 消息事件）。

<reply>活动用于通过输入消息活动如<receive>活动发送响应给先前的请求。这些响应只对请求-响应交互有效。单向“response”可以通过在partnerLink上调用相应的单向操作发送。<reply>活动可以指定包含被发送消息数据的引用变量的variable属性。如果WSDL消息定义没有包括任何部分，则相关variable属性可以被省略，[SA00047]且<toParts>构造必须被省略。在<reply>活动上使用的<toPart>元素的语义和语法与在章节 10.3.1 映射WSDL消息部分 中有关<invoke>活动的相同，[SA00059]包括如果<reply>活动使用<toPart>元素则禁止在相同活动上使用variable属性的约束。

```
<reply partnerLink="NCName"
   portType="QName"? operation="NCName"
   variable="BPELVariableName"?
   faultName="QName"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
```

```
</reply>
```

<reply>活动有两种潜在形式。第一种，在普通响应中，不使用 faultName 属性且 variable 属性（或者其等效<toPart>元素）存在时，将指出有响应消息的变量。第二种，当响应出现故障，使用 faultName 属性且 variable 属性（或者其等效<toPart>元素）存在时，将指出相应故障的变量。faultName 属性应该参照在<reply>活动使用的操作中定义的故障且变量最好应匹配同引用故障相关联的消息类型（注意：这里的匹配语义参照章节 12.5 故障处理器中<catch>相关匹配规则的#1 和#2）。WS-BPEL 处理故障基于抽象 WSDL 1.1 操作定义。这限制了 WS-BPEL 程序在故障通过 SOAP 绑定被送回时决确定信息传送的能力（见章节 10.3 调用 Web 服务操作——Invoke）。

10.4.1 消息交换

可选择的messageExchange属性用于消除输入消息活动（IMA）和<reply>活动间的关系歧义。messageExchange的明确用法只有在相同的正在被同时执行的partnerLink和operation上执行导致多个IMA-<reply>对（如<receive>-<reply>对）时需要。[SA00060]在这些情况下，程序定义必须明确标记这些配成一对的关系。

<reply>活动同IMA相关联，如基于partnerLink，operation和messageExchange元组的<receive>,<onMessage>和<onEvent>。[SA00061]messageExchange属性中使用的名字必须解析成封装<reply>活动以及其相应IMA的作用域（程序的根作用域）中声明的messageExchange。这个解析同相关集解析一样遵循相同的作用域规则。

开放式 IMA 描述了来自启动执行直到相关<reply>活动成功完成的请求-响应 IMA 的 Web 服务操作的状态。如果<reply>活动出现故障，IMA 继续开启且可以尝试其它的<reply>活动，例如来自故障处理器。带有多个同时开放其 partnerLink，operation 和 messageExchange 元组相同的 IMA 是非法的。WS-BPEL 处理器必须在 IMA 开始执行时抛出 bpel:conflictingRequest 故障。在多个同时开放的 IMA 里使用相同的 messageExchange 也是非法的，即使 partnerLink 和 operation 的组合都各不相同。请注意 bpel：conflictingRequest 在语义上不同于 bpel：conflictingReceive，因为通过在特殊的 partnerLink，operation 和 messageExchange 元组上连续接收相同请求来创建 conflictingRequest 是可能的，且不会触发 conflictingReceive（见章节 10.4 提供 Web 服务操作-Receive 和 Reply 上有关 conflictingReceive 的语义）。

如果<reply>活动不能通过匹配 partnerLink，operation 和 messageExchange 元组同开放式 IMA 活动相关联，则 WS-BPEL 处理器必须在<reply>活动上抛出 bpel：missingRequest 故障。因为冲突的请求在 IMA 开始处理时被阻止，则在<reply>活动被执行时不能有多于一个的相应的 IMA。

当<scope>的主要活动和时间处理器完成则需要完成所有<scope>内部声明的依赖于伙伴链接或消息交换的 Web 服务交互。开放式 IMA 使用正在被完成或已经完成的<scope>中的伙伴链接或消息交换被称作孤立 IMA。孤立 IMA 的检测将会导致抛出 bpel：missingReply 故障。更多有关孤立 IMA 的详细资料在章节 12.2 消息交换处理 中讨论。因此，如果程序实例包括一个或多个开放式 IMA 完成，则 bpel：missingReply 故障必须被抛出。

如果 messageExchange 属性没有在 IMA 或<reply>上指定，则活动的 messageExchange 自动同缺省的没有名字的 messageExchange 相关联。缺省 messageExchange 通过<process>和<onEvent>的直接子作用域及<forEach>的并行形式隐式声明。<scope>活动其它的具体值不提供缺省的 messageExchange。缺省 messageExchange 实例，正如非缺省的 messageExchange 元素，每次在作用域声明的缺省 messageExchange 被执行时创建。例如每次<onEvent>被执行时（也就是当处理的新消息到达时）创建新的同每个<onEvent>实例相关联的缺省 messageExchange 实例。这允许请求-响应<onEvent>事件处理器在没有错误或明确指定 messageExchange 时并行接收消息。类似地允许<forEach>并行形式中的<receive>-<reply>或<onMessage>-<reply>对不用明确指定 messageExchange 的用法。

10.5 更新变量和伙伴链接——Assign

变量更新通过<assign>活动发生，在章节 8.4 赋值活动 中描述。

10.6 发送内部故障信号——Throw

当业务程序需要显式发送内部故障信号时，使用<throw>活动。故障必须通过 QName 识别（见章节 10.3 调用 Web 服务操作）。<throw>活动提供故障的名字，并且可以选择提供故障更进一步的信息数据。故障处理器可以使用这样的数据来处理故障及移植需要被送到其他服务的故障消息。

WS-BPEL 不要求在<throw>活动中预先定义故障名。这提供了介绍业务程序故障的轻量级机制。可以通过适当的 QName 直接使用业务程序中定义的故障名，WSDL 定义或 WS-BPEL 标准故障，如 faultName 属性值和提供带有故障数据的 variable，如果需要的话。

```
<throw faultName="QName" faultVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
</throw>
```

<throw>活动不提供故障数据的简单例子：

```
<throw xmlns:FLT="http://example.com/faults"
   faultName="FLT:OutOfStock" />
```

10.7 延迟处理——Wait

<wait>活动指定了某个时间段或者直到某个最后期限到达的延迟（见章节 8.3 表达式 中有关持续时间表达式和最终期限表达式的语法）。如果<for>中指定的持续时间值为 0 或负数，

或<until>中指定的最终期限已经到达或过去，则<wait>活动立即完成。

```
<wait standard-attributes>
   standard-elements
   (
   <for expressionLanguage="anyURI"?>duration-expr</for>
   |
   <until expressionLanguage="anyURI"?>deadline-expr</until>
   )
</wait>
```

这个活动典型的应用是在某个时刻调用操作（在这个例子中是个常数，但更多是依赖于程序状态的表达式）：

```
<sequence>
   <wait>
      <until>'2002-12-24T18:00+01:00'</until>
   </wait>
   <invoke partnerLink="CallServer" portType="AutomaticPhoneCall"
      operation="TextToSpeech" inputVariable="seasonalGreeting" />
</sequence>
```

10.8 不做任何事——Empty

经常需要使用一个活动不做任何事，如当故障需要被捕获和抑制时。为了这个目的，可以使用<empty>活动。<empty>的另外一个用法是在<flow>中提供同步点。

```
<empty standard-attributes>
   standard-elements
</empty>
```

10.9 添加新的活动类型——ExtensionActivity

WS-BPEL 程序定义可以包括没有在本规范中定义的新活动，通过将它们置于<extensionActivity>元素中。这些活动被看作是扩展活动。<extensionActivity>元素的内容必须是合法的有不同于 WS-BPEL 名称空间的名称空间的单个元素。单个元素必须使得 WS-BPEL 的 standard-attributes 和 standard-elements 可用。如果包括在<extensionActivity>中的元素不能让 WS-BPEL 处理器识别且不能服从来自扩展声明的 mustUnderstand=“yes”请求，则未知活动必须被当作有 standard-attributes 和 standard-elements 的未被识别元素的<empty>活动；它的所有其他属性和子元素被忽略。standard-attributes 和 standard-elements 必须如同本规范中定义的一样处理，不管扩展是否被理解。

WS-BPEL 处理器执行的静态分析在它被忽略后，未被识别的扩展活动的非标准属性和非标准元素不遵守 mustUnderstand=“yes”。可能会检测到某些违反 WS-BPEL 要求的语义的情况。如：

- 至少必须存在一个开始活动——如果<extensionActivity>有一个嵌套的开始活动，则若<extensionActivity>的非标准子构造器被忽略，要求被破坏。
- 链接必须有一个确定的资源和目标——如果<extensionActivity>有一个嵌套的活动是跨越<extensionActivity>边界的链接的资源或目标，则若<extensionActivity>的非标准子构造器被忽略，要求被破坏。

<extensionActivity>也可以是构造活动，即可以包括其他活动。如果<extensionActivity>允许嵌套活动，其相应的扩展声明应该遵守 mustUnderstand=“yes”。

```
<extensionActivity>
   <anyElementQName standard-attributes>
      standard-elements
   </anyElementQName>
</extensionActivity>
```

10.10 立即结束程序——Exit

<exit>活动用于立即结束业务程序实例。所有当前正在运行的活动必须立即结束不涉及终止处理，错误处理或补偿行为。

```
<exit standard-attributes>
   standard-elements
</exit>
```

10.11 传播错误——Rethrow

<rethrow>活动用于故障处理器重新抛出捕获的故障，也就是说，故障名以及存在的原始故障的故障数据。它只能在故障处理器内部使用（<catch>和<catchall>）。故障数据的修改必须通过<rethrow>忽略。比如如果故障处理器内的逻辑修改了故障数据且稍后调用<rethrow>，则原始故障数据被重新抛出且不是被修改后的故障数据。类似地如果使用简化操作捕获故障允许某部分使用元素定义的消息类型故障被寻找相同元素类型的故障处理器捕获，则<rethrow>会重新抛出原始消息类型数据（见章节 12.5 故障处理器）。

```
<rethrow standard-attributes>
   standard-elements
</rethrow>
```

11. 构造活动

构造活动指示了活动集执行的命令。它们描述了如何创建业务流程，通过设计基本活动（见章节 10 基本活动）。它执行到表示控制模式的结构内，处理故障和外部事件，并且协调包括在业务协议中的程序实例间的消息交换。

WS-BPEL 为构造活动定义了多种控制流模式：

- <sequence>，<if>，<while>，<repeatUtill>和<forEach>的串行变量体提供的活动间的顺序控制。
- <flow>和<forEach>的并行变量体提供的活动间的并发和同步。
- <pick>提供的内部和外部事件控制的延迟选择。

WS-BPEL 中的构造活动集没有被规定为最小的。一个活动的语义可以使用其它的活动来表示。例如，顺序处理可以使用<sequence>或者适当定义链接的<flow>模拟。

构造活动可以任意的嵌入或组合。这提供了传统意义上被看做优于正交混合特性的交替式构造图表和构造体模拟式样的混合体。在章节 5.1 初始化例子 中可以找到有关混合应用的简单例子。

下面所说的活动包括基本和构造活动。

11.1 顺序处理-Sequence

<sequence>活动包括顺序执行的一个或多个活动，词法要求它们应该在<sequence>元素中出现。当序列中最后一个活动完成时<sequence>活动完成。

```
<sequence standard-attributes>
   standard-elements
   activity+
</sequence>
```

例子：

```
<sequence>
   <flow>...</flow>
   <scope>...</scope>
   <pick>...</pick>
</sequence>
```

11.2 条件行为-If

<if>活动提供条件性行为。活动由<if>和可选择的<elseif>定义的一个或多个条件转移的有序列表组成，后面跟随可选的<else>元素。<if>和<elseif>转移在它们出现的地方按顺序考虑。第一个其<condition>标记为 true 的转移被取走时，它包括的活动被执行。如果无有条件的转移被取走，则若<else>转移存在的话就被取走。当包含的选择性转移的活动完成时，或没有<condition>被验证为 true 且没有指定<else>转移，<if>活动完成。

```
<if standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
   activity
   <elseif>*
      <condition expressionLanguage="anyURI"?>bool-expr</condition>
      activity
   </elseif>
   <else>?
      activity
   </else>
</if>
```

例子：

```
<if xmlns:inventory="http://supply-chain.org/inventory"
   xmlns:FLT="http://example.com/faults">
   <condition>
      bpel:getVariableProperty('stockResult','inventory:level') > 100
   </condition>
   <flow>
      <!-- perform fulfillment work -->
   </flow>
   <elseif>
      <condition>
         bpel:getVariableProperty('stockResult','inventory:level') >=
0
      </condition>
      <throw faultName="FLT:OutOfStock" variable="RestockEstimate" />
   </elseif>
   <else>
      <throw faultName="FLT:ItemDiscontinued" />
   </else>
</if>
```

11.3 重复执行-While

<while>活动提供被包含活动的重复执行。在每个迭代开始，只要布尔<condition>验证为true，则被包含活动一直被执行。

```
<while standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
   activity
</while>
```

例子：

```
<while>
   <condition>$orderDetails > 100</condition>
   <scope>...</scope>
</while>
```

11.4 重复执行-RepeatUntil

<repeatUntil>活动为被包含的活动提供重复执行。被包含活动执行直到给出的布尔<condition>变成 true。每个循环体被执行后检查条件。对比<while>活动，<repeatUntil>循环执行活动至少一次。

```
<repeatUntil standard-attributes>
   standard-elements
   activity
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
</repeatUntil>
```

11.5 选择性事件处理-Pick

<pick>活动等待事件集中某个具体事件发生，然后处理同那个事件相关联的活动。当事件被选中后，其他的事件不再被<pick>接受。如果在多个事件中发生紊乱情况，事件的选择是相关实现的（the choice of the event is implementation dependent）（见章节 10.4 提供 Web 服务操作-Receive 和 Reply 中有关紊乱情况的描述）。

<pick>活动由转移集组成，每个包括一个成对的事件-活动。当选中的活动完成时，<pick>活动完成。<pick>活动的事件有两种形式到达：

- <onMessage>类似于<receive>活动，因为它等待入站式消息的接收。
- <onAlarm>相当于基于计时器的警报信号。如果<for>中指定的持续时间值为 0 或负数，或<until>中指定的最终期限已经到达或过去，则立即执行<onAlarm>事件。此外，紊乱情况的处理是相关实现的。（implementation dependent.）

每个 pick 活动必须至少包括一个<onMessage>。

当业务流程新实例被创建在<onMessage>事件的接收上时，使用<pick>的特殊形式。这种形式的<pick>有一个creatInstance=“yes”的属性（缺省属性值为no）。[SA00062]在这样的情况下，<pick>中的事件必须全部为<onMessage>事件。这个要求必须静态强制。

[SA00063]<onMessage>事件的语义等同于<receive>活动关于variable属性或<fromPart>元素（见[SA00047]，紊乱情况的处理、相关集的处理、单个基于元素部分消息的捷径和关于同时启动相冲突的接收操作约束的选择性）。在最后一种情况里，如果相同partnerLink，portType和correlationSet（s）在执行期间启动两个或多个receive操作，则标准故障bpel：conflictingReceive 必须被抛出（见章节 10.4 提供Web服务操作-Receive和Reply）。<onMessage>事件的启动等同于相应<receive>活动的启动。

可选项 messageExchange 属性用于关联<onMessage>构造器和<reply>活动（更多详细资料见章节 10.4 消息交换）。

```
<pick createInstance="yes|no"? standard-attributes>
   standard-elements

   <onMessage partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      variable="BPELVariableName"?
      messageExchange="NCName"?>+
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      activity
   </onMessage>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )
      activity
```

```
   </onAlarm>
</pick>
```

下面的例子展示了<pick>的典型应用。循环中发生的<pick>活动从命令中接收条目列表。超时请求完成由<onAlarm>事件激活。

```
<pick>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="inputLineItem"
      variable="lineItem">
      <!-- activity to add line item to order -->
   </onMessage>
   <onMessage partnerLink="buyer"
      portType="orderEntry"
      operation="orderComplete"
      variable="completionDetail">
      <!-- activity to perform order completion -->
   </onMessage>
   <!-- set an alarm to go off
        3 days and 10 hours after the last order line -->
   <onAlarm>
      <for>'P3DT10H'</for>
      <!-- handle timeout for order completion -->
   </onAlarm>
</pick>
```

11.6 并行和控制依赖处理-Flow

<flow>活动提供并发和同步。<flow>的语义如下：

```
<flow standard-attributes>
   standard-elements
   <links>?
      <link name="NCName">+
   </links>
   activity+
</flow>
```

在<flow>中将活动分组的效果是能够启用并发。当所有附属<flow>的活动完成时，<flow>结束。如果它的启动条件验证为 false 则活动被跳过，并且也被看做完成（见章节 11.6.3 消除无用的路径）。

在下面的例子中，当<flow>开始时激活两个<invoke>活动来启动并发。假定<invoke>操作是请求-响应操作，则<flow>在卖家和发货人响应后完成。“transferMoney”活动在<flow>完成后执行。

```
<sequence>
   <flow>
      <invoke partnerLink="Seller" ... />
      <invoke partnerLink="Shipper" ... />
   </flow>
   <invoke partnerLink="Bank" name="transferMoney" ... />
</sequence>
```

<flow>活动创建了直接嵌套在内的并发活动集。它可以启动内嵌的活动间的同步依赖至任何深度。<link>构造器用于表示这些同步依赖。通过<flow>活动附属<link>的声明。[SA00064]<link>有一个强制的name属性，在同一个直接附属的<flow>中必须是唯一标识的。这个要求必须被静态地强制。

11.6.1 Flow的相关属性和元素

<flow>内嵌活动的 standard-attributes 和 standard-elements 是很重要的因为存在标准属性和元素为活动提供链接语义。每个 WS-BPEL 活动有可选择的容器<source>和<targets>，分别包括<source>和<target>元素的集合。这些元素用于通过<link>建立同步关系。

```
<targets>?
   <joinCondition expressionLanguage="anyURI"?>?
      bool-expr
   </joinCondition>
   <target linkName="NCName" />+
</targets>

<sources>?
   <source linkName="NCName">+
      <transitionCondition expressionLanguage="anyURI"?>?
         bool-expr
      </transitionCondition>
   </source>
</sources>
```

[SA00065]<source>或<target>的linkName属性值必须是在附属<flow>活动中声明的<link>的名字。[SA00068]活动通过包含一个或多个<source>元素可以声明自身是一个或多个链接的资源。每个同给出活动相关联的<source>元素必须使用linkName来区别活动中其他的<source>元素。类似的，[SA00069]活动可以声明自身是一个或多个链接的目标，通过包含一个或多个<target>元素。每个同活动相关联的<target>元素必须使用linkName来区别活动中其他的<target>元素。[SA00067]两个不同的链接禁止共享同一个资源和目标活动，即连接两

个活动最多使用一个链接。[SA00066]每个<flow>活动内声明的链接在<flow>里必须有一个明确的活动作为它的资源和一个明确的活动作为它的目标。链接的资源和目标可以嵌入到<flow>内嵌构造性活动的任意深度，除了下面定义的越界约束。本段中指定的所有要求必须被静态地强制。

总体上<targets>可以指定一个可选择的<joinCondition>。<joinCondition>元素值是表达式语言中由 expressionLanguege 属性指出的或者这个程序默认的表达式语言的布尔表达式（见章节 8.3 表达式）。如果没有指定<joinCondition>，则<joinCondition>是这个活动所有输入链接的链接状态的逻辑和（也就是逻辑或操作）。

每个<source>元素可以指定可选择的<transitionCondition>作为下列指定链接的防护。如果<transitionCondition>被省略，则假定验证为 true。

每个活动上的 standard-attributes 选项，suppressJoinFailure，都涉及到链接。这个属性指出如果发生连接故障，是否应该被抑制（见章节 11.6.3 消除不可用的路径）。当活动没有指定 suppressJoinFailure 属性时，属性从最近的附属构造器（也就是活动或者程序本身）中继承值。

<joinCondition>，<transitionCondition>以及 suppressJoinFailure 的语义在章节 11.6.2 链接语义 中讨论。

假设链接的资源以任何级别内嵌在语法构造器中，并且构造器内部的链接没有以任何级别被声明。我们称这样的链接离开构造器。同样地，链接的目标以任何级别内嵌在语法构造器中，且构造器内部的链接没有以任何级别被声明。我们称这样的链接插入构造器。不管插入还是离开构造器的链接，都看作是构造器的越界。当链接的资源或目标活动内嵌于构造器 X，同时链接被声明在构造器 X 之外，则链接被称作既插入又离开构造器。

下面的例子演示了构造性活动越界的链接。名为 CtoD 的<link>在<sequence>Y 中启动活动 C 并且通过<flow>活动直接附属的活动 D 结束。例子进一步的说明了<sequence>X 必须先于<sequence>Y 执行因为 X 是名为 XtoY（被<sequence>Y 作为目标）的<link>的资源。XtoY 链接跨越了<sequence>X 和<sequence>Y 的边界。

```
<flow>
   <links>
      <link name="XtoY" />
      <link name="CtoD" />
   </links>
   <sequence name="X">
      <sources>
         <source linkName="XtoY" />
      </sources>
      <invoke name="A" ... />
      <invoke name="B" ... />
   </sequence>
```

```
   <sequence name="Y">
      <targets>
         <target linkName="XtoY" />
      </targets>
      <receive name="C" ...>
         <sources>
            <source linkName="CtoD" />
         </sources>
      </receive>
      <invoke name="E" ... />
   </sequence>
   <invoke name="D" ...>
      <targets>
         <target linkName="CtoD" />
      </targets>
   </invoke>
</flow>
```

可重复构造器（<while>，<repeatUntil>，<forEach>，<eventHandlers>）中使用的链接必须在自身嵌套在可重复构造器或<compensationHandlers>的<flow>中声明。[SA00070]链接禁止跨越可重复构造器或<compensationHandlers>元素的边界。[SA00071]跨越<catch>，<catchAll>或<terminationHandler>元素边界的链接必须只能被输出，即必须在<faultHandlers>或<terminationHandler>中有它的资源活动，并且它的目标活动在处理器相关联的作用域之外（见章节 12. <eventHandlers>，<faultHandlers>，<terminationHandler>以及<compensationHandler>的作用域规范）。

[SA00072]<flow>中声明的<link>禁止创建控制周期，即资源活动禁止有作为上述逻辑活动的目标活动。这暗示了这些定向图标总是非循环的。如果语法上B的初始化要求A的完成，则称活动A逻辑优先于活动B。特别的是，链接禁止有活动作为目标如果资源活动封装目标活动，反之亦然。这些要求必须被静态地强制。

为了举例说明上述内容，下面的例子演示了链接的非法使用，因为违反了链接禁止在资源活动中封装目标活动的约束：

```
<sequence>
   <sources>
      <source linkName=”L1”>
   </sources>
   ...
   <invoke ...>
      <targets>
         <target linkName=”L1” />
      </targets>
   </invoke>
```

```
    ...
</sequence>
```

11.6.2 链接语义

在余下的章节中，资源活动 A 的链接将会作为 A 的输出链路被引用，目标活动 A 的链接将会作为输入链路被访问。如果活动 X 是以活动 Y 为资源链接的目标，我们说 X 同步依赖于 Y。

每个作为链接目标的活动有一个显式或隐式的同活动相关联的连接条件。这个即使在活动只有一个输入链路时也适用。显式连接条件由<targets>元素下的<joinCondition>元素提供。如果显式连接条件缺损，隐式条件要求最少一个输入链路为true（见下面链接状态的解释）。连接条件是布尔表达式（见章节 8.3.1 布尔表达式）。[SA00073]连接条件的表达式必须只能使用布尔运算符和活动输入链路的状态值构造。

忽略链接，当给出的活动准备启动时决定业务流程、<scope>以及构造活动的语义。例如，<sequence>中的第二个活动在第一个活动完成时准备启动。活动包括在<if>的转移中当转移被选择时启动。类似的，直接嵌入在<flow>中的活动当<flow>自身启动时启动。

如果准备启动的活动有一个输入链路，则它禁止启动直到确定所有输入链路并且验证显式或隐式的连接条件。为了防止违反控制依赖，连接条件的验证只有在确定所有输入链路的状态后执行。

链接状态是同每个已声明链接相关联的三态标记。这个标记可能是以下三种状态：true，false 或者 unset。<link>状态的生命周期是声明<link>的<flow>活动的生命周期。每次激活<flow>活动，其中声明的所有链接状态为 unset。

验证链接状态的语义在下面的段中描述。

当活动 A 完成且没有传播任何故障时，则必须执行下面的步骤来确定其他活动上链接的效果：

- 确定 A 中所有输出链路的状态。状态应该是 true 或者 false。确定每个链接的<transitionCondition>状态是否被验证。如果<transitionCondition>引用的某些变量在当前路径被更改，则过渡条件验证的结果可以不依赖于当前活动中的时间特性。
- 对于每个同步依赖 A 的活动 B，检查：
  - ✧ 在上述情况下，B 是否已经为启动做好准备（除了它在输入链路上的依赖）。
  - ✧ B 所有的输入链路是否已经确定。请注意如果输入链路离开独立的作用域，则无法知道链路的最终状态直到独立作用域完成（见章节 12.8 独立作用域）。

如果上面的两个条件都为 true，则验证 B 的<joinCondition>，如果验证为 true，启动活动 B。否则标准故障 bpel：joinFailure 必须被抛出，直到 bpel：joinFailure 未被抛出的情况下 suppressJoinFailure 的值为“yes”（见章节 11.6.3 消除无用路径）。

当活动有多个输出链路时，链接和相关过渡条件的状态验证的指令被定义为有序的，根据<source>元素中声明的链接顺序。

相关资源活动必须在链接的<transitionCondition>被验证之前完成。如果资源活动自身是<scope>的话，不要求成功完成。即<scope>可能遇到一个内部错误也能（不成功地）完成如果有同<scope>相关联的故障处理器且故障处理器没有抛出故障的完成。如果在验证<transitionCondition>的同时发生错误，则错误不会影响活动的完成状态并且由资源活动附属的作用域处理。如果链接的目标在资源活动的附属作用域之外则链接的状态为 false。验证过渡条件时发生故障和没有验证过渡条件的链接状态是相同的。如果目标是在附属作用域内，则从作用域发生故障起与状态不相关（见章节 11.6.3 消除无用的路径）。在链接 L 将<scope>X 作为它的资源活动的情况下，在为 L 验证过渡条件中的错误产生的故障将被传播给<scope>X 的附属<scope>。

如果在验证输出链路的活动的某个过渡条件时发生错误，则剩余的所有目标在资源活动的附属作用域中的输出链路禁止验证它们的过渡条件并且保持 unset 状态。不管怎样，如果剩余输出链路的目标在资源活动的附属作用域之外，则链接的状态必须被设置为 false。

如果，在构造性活动 A 的执行期间，A 的语义指示嵌套在 A 中的活动 B 将不会作为 A 的处理部分被执行，则来自 B 的所有输出链路的状态必须被设置为 false。尽管如此，为了防止违反控制依赖，这个规则只能适用于所有 B 的输入链路的状态确定之后，同活动的输入链路一样，在 B 有控制依赖的基础上已经被决定。有关这个规则的应用例子是在<if>活动的转移内部的<condition>为 false 的活动。其他的例子可以在因为故障的<scope>而未完成的活动中看到（见章节 12.作用域 和 12.4 补偿处理器）。控制依赖上的规则也用于<faultHandlers>和<terminationHandler>输出的链接上。如果确定这些处理器之一不会运行，则所有输出链接的状态被设置为 false。

在下面的例子中，toSkipped 链接创建了<if>中从<receive>活动到<empty>活动的控制依赖。fromSkipped 链接创建从<empty>活动到<reply>活动的依赖。这两个链接创建了从<receive>活动到<reply>活动的传递相关性。即使<if>条件验证为 false 而跳过<empty>活动，还是保留传递相关性，并且 fromSkipped 的状态因此未设置为 false 直到明确 toSkipped 的状态。

```
<flow>
   <links>
      <link name="toSkipped" />
      <link name="fromSkipped" />
   </links>

   <receive ...>
      <sources>
```

```
         <source linkName="toSkipped" />
      </sources>
      ...
   </receive>

   <if>
      <condition>
         ... <!-- evaluates to false -->
      </condition>

      <empty name="skipped">
         <targets>
            <target linkName="toSkipped">
         </targets>
         <sources>
            <source linkName="fromSkipped">
         </sources>
      </empty>
   </if>

   <reply ...>
      <targets>
         <target linkName="fromSkipped" />
      </targets>
   </reply>
</flow>
```

<onEvent>和<onAlarm>处理器，同并行的<forEach>活动一样可以有当前同步的实例。这些构造器的子作用域中声明的数据和资源包括链接，必须在每个实例中单独处理。

当<flow>活动嵌套在其他的<flow>活动中时，内部的<flow>活动可以定义为有相同名字的<link>，和在附属的<flow>活动中一样。资源或目标引用这样的<link>匹配活动可视的最内层的<link>。

11.6.3 消除无用的路径

当链接定义的控制流以及 suppressJoinFailure 属性设置为“yes”，如果活动 A 的连接条件的解析验证为 false，则 A 禁止被执行。在这种情况下，禁止生成故障 bpel：joinFailure。这个属性值被所有嵌套的活动继承，除了被另外的 suppressJoinFailure 属性设置覆盖的地方。

当目标活动由于<joinCondition>（显式或隐式）的值为 false 而没有执行时，它的输出链路根据章节 11.6.2 链接语义 必须被赋值为 false。这会顺着整个由成功链路形成的路径及物地传播 false 链接状态直到连接条件被验证为 true。这个方法被称作消除无用的路径（DPE）。

<process>元素的 suppressJoinFailure 属性缺省值为“no”。这通过故障设置避免了抑制明确定义的故障。考虑在章节 5.1 初始化例子 中 suppressJoinFailure 设置为“yes”的例子的解析。进一步假设运送提供者的启动被附属在由错误处理器提供的作用域中（见章节 12. 作用域 和章节 12.5 错误处理器）。如果启动之一失败，则来自这个启动的输出链路状态为 false，且（隐式的）链接目标中的<joinCondition>为 false，但是作为结果 bpel：joinFailure 将被隐式地抑制并且目标活动在序列中将会被默默的跳过代替导致预期的故障。

如果想要得到 bpel：joinFailure 故障的通用表达式，可以通过在<process>元素中设置属性 suppressJoinFailure 为“yes”来完成。

11.6.4 流程图例子

在下面的例子中，名为 receiveBuyerInformation，receiveSellerInformation，settleTrade，confirmBuyer 以及 confirmSeller 的活动是在<flow>活动中定义的图的节点。

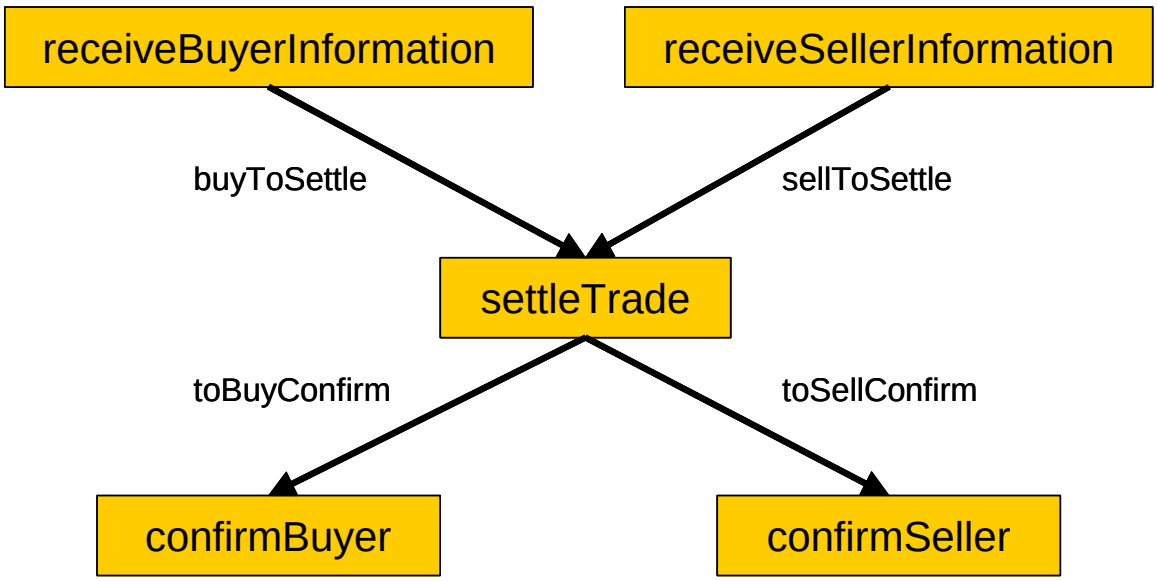


Figure 2: Flow Graph

下面的<link>被定义为：

- buyToSettle 在 receiveBuyerInformation 中开始（在 receiveBuyerInformation 内嵌的相应<source>元素中指定），并且在 settleTrade 中结束（在 settleTrade 内嵌的相应<target>元素中指定）
- sellToSettle 在 receiveSellerInformation 中开始并且在 settleTrade 中结束。
- toBuyConfirm 在 settleTrade 中开始且在 confirmBuyer 中结束。
- toSellConfirm 在 settleTrade 中开始且在 confirmSeller 中结束。

基于<flow>定义的结构图，活动 receiveBuyerInformation 和 receiveSellerInformation 可以并发运行。settleTrade 活动只有在这两个活动完成后执行。在 settleTrade 完成两个活动后，confirmBuyer 和 confirmSeller 再次并发执行。

```
<flow suppressJoinFailure="yes">
   <links>
      <link name="buyToSettle" />
      <link name="sellToSettle" />
      <link name="toBuyConfirm" />
      <link name="toSellConfirm" />
   </links>
   <receive name="receiveBuyerInformation" ...>
      <sources>
         <source linkName="buyToSettle" />
      </sources>
   </receive>
   <receive name="receiveSellerInformation" ...>
      <sources>
         <source linkName="sellToSettle" />
      </sources>
   </receive>
   <invoke name="settleTrade" ...>
      <targets>
         <joinCondition>$buyToSettle and $sellToSettle</joinCondition>
         <target linkName="buyToSettle" />
         <target linkName="sellToSettle" />
      </targets>
      <sources>
         <source linkName="toBuyConfirm" />
         <source linkName="toSellConfirm" />
      </sources>
   </invoke>
   <reply name="confirmBuyer" ...>
      <targets>
         <target linkName="toBuyConfirm" />
      </targets>
   </reply>
   <reply name="confirmSeller" ...>
      <targets>
         <target linkName="toSellConfirm" />
      </targets>
   </reply>
</flow>
```

11.6.5 链接和构造性活动

链接可以跨越构造性活动的边界（见章节 11.6.1 Flow 相关标准属性和元素）。下面的例子说明了当目标活动在构造器内时的链接活动。

<flow>预备按顺序执行活动 A,B,C。活动 B 有一个在序列外的在活动 X 和 Y 上的同步依赖。即来自 X 和 Y 的链接的目标是 B。B 中没有指定<joinCondition>，因此将使用以 B 为目标的链接的逻辑和（也就是逻辑或）。如果最少一个输入链路为 true 状态，则条件为 true。在这种情况下，条件变为布尔条件（`P:funcXB()` 或 `P:funcYB()`）。

在<flow>中，<sequence>命名为 S 并且两个<receive>活动 X 和 Y 在<flow>开始时都同时激活。在 S 里活动 A 完成之后，B 不能启动直到确定来自 X 和 Y 的输入链路状态且隐式链接条件被验证。当活动 X 和 Y 完成，B 的连接条件被验证。

假设过渡条件 `P:funcXB()` 和 `P:funcYB()`都验证为 `false`，则抛出标准故障 `bpel：joinFailure`，因为附属`<flow>`活动的 `suppressJoinFailure` 属性值设为“no”。因此`<flow>`的行为被中断且 `B` 和 `C` 都不会被执行。

如果附属`<flow>`活动的 `suppressJoinFailure` 属性值设为”yes”,则 `B` 将会被跳过但是 `C` 会被执行因为 `bpel：joinFailure` 故障被抑制。

```
<flow suppressJoinFailure="no">
   <links>
      <link name="XtoB" />
      <link name="YtoB" />
   </links>
   <receive name="X" ...>
      <sources>
         <source linkName="XtoB">
            <transitionCondition>P:funcXB()</transitionCondition>
         </source>
      </sources>
      ...
   </receive>
   <receive name="Y" ...>
      <sources>
         <source linkName="YtoB">
            <transitionCondition>P:funcYB()</transitionCondition>
         </source>
      </sources>
      ...
   </receive>
   <sequence name="S">
      <receive name="A" ...>...</receive>
      <receive name="B" ...>
         <targets>
            <target linkName="XtoB" />
            <target linkName="YtoB" />
```

```
         </targets>
      </receive>
      <receive name="C" ... />
   </sequence>
</flow>
```

最后，假设先前的<flow>被连接的 A,B,C 通过链接（带有常数为 true 的缺省<transitionCondition>元素）轻微地改动，则修改后的值将代替它们进入<sequence>。因为默认的连接条件为逻辑和且 A 到 B 的链接的<transitionCondition>为常量 true，则连接条件总会被验证为 true，同来自 P:funcXB() 和 P:funcYB()的值无关。现在，B 和后面的 C 将总会被处理。

```
<flow suppressJoinFailure="no">
   <links>
      <link name="XtoB" />
      <link name="YtoB" />
      <link name="AtoB" />
      <link name="BtoC" />
   </links>
   <receive name="X">
      <sources>
         <source linkName="XtoB">
            <transitionCondition>P:funcXB()</transitionCondition>
         </source>
      </sources>
   </receive>
   <receive name="Y">
      <sources>
         <source linkName="YtoB">
            <transitionCondition>P:funcYB()</transitionCondition>
         </source>
      </sources>
   </receive>
   <receive name="A">
      <sources>
         <source linkName="AtoB" />
      </sources>
   </receive>
   <receive name="B">
      <targets>
         <target linkName="AtoB" />
         <target linkName="XtoB" />
         <target linkName="YtoB" />
      </targets>
```

```
      <sources>
         <source linkName="BtoC" />
      </sources>
   </receive>
   <receive name="C">
      <targets>
         <target linkName="BtoC" />
      </targets>
   </receive>
</flow>
```

11.7 处理多分支—ForEach

<forEach>活动总会执行它包含的<scope>活动 N+1 次，N=<finalCounterValue>-<startCounterValue>。

```
<forEach counterName="BPELVariableName" parallel="yes|no"
   standard-attributes>
   standard-elements
   <startCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </finalCounterValue>
   <completionCondition>?
      <branches expressionLanguage="anyURI"?
         successfulBranchesOnly="yes|no"?>?
         unsigned-integer-expression
      </branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

当<forEach>活动开始时，验证<startCounterValue>和<finalCounterValue>中的表达式。一旦两个值被返回则它们在活动的生命期限内保持常量。[SA00074]两个表达式都必须返回一个可以像xsd:unsignedInt 一样被验证的TII（意思是它们至少包含一个字符）。如果这些表达式没有返回一个有效值，则必须抛出bpel：invalidExpressionValue故障（见章节 8.3 表达式）。如果<startCounterValue>大于<finalCounterValue>，则禁止执行子<scope>活动且结束<forEach>活动。

<forEach>的子活动必须是<scope>活动。<forEach>构造器引入了隐式计数器变量，还引入

了动态并发操作（也就是有不能提前知道计数的并行分支）。<scope>活动提供一个定义明确的作用域快照语义以及命名用于补偿的动态并发工作的方法（见章节 12.4.2 补偿处理器中程序状态应用 有关作用域快照定义）。

如果 parallel 属性值设为“no”则活动是串行的<forEach>。附属的<scope>活动必须被执行 N+1 次，每个实例只能在先前的重复完成后开始。如果程序提前结束如发生故障或者完成条件被验证为 true，则 N+1 次要求不适用。在每个重复期间，xsd：unsignedInt 型的变量在<forEach>活动的子<scope>中隐式声明。这个隐式变量的名字在 counterName 属性中指定。作用域的第一个迭代将在初始化至<startCounterValue>的计数器变量中出现。下一个迭代将会导致计数器变量被初始化给<startCounterValue>+1。后来每个迭代将会在先前被初始化的计数器变量值基础上加 1 直到最后一个计数器设置给<finalCounterValue>的迭代。计数器变量是局部定义在附属<scope>中且它的值虽然在迭代期间可以改变，但是值会在每个迭代的最后丢失。因此，计数器改变值不会影响下一个迭代的计数器的值。

如果 parallel 属性设为 yes 则活动是并行的<forEach>。附属的<scope>活动必须被同时执行 N+1 次。本质上隐式<flow>是同 N+1 个子<forEach>的附属<scope>活动的拷贝一起被动态创建。每个<scope>活动的拷贝将包含和串行<forEach>同样方式声明的计数器变量。每个实例的计数器变量必须被唯一并行初始化至以 <startCounterValue> 开始一直到包括 <finalCounterValue>的整数值之一，作为<scope>实例化的一部分。

[SA00076]如果相同名字的变量作为在隐式作用域中声明的counterName的属性值，将会被看做重复变量声明在静态分析期间必须被报错。

没有 <completionCondition> 的 <forEach> 活动当它所有的子 <scope> 结束后完成。<completionCondition>元素可随意指定来阻止某些正在执行的孩子（在串行的情况下），或者强制提前中断某些孩子（在并行的情况下）。

<branches>元素表示一个用于定义“最少 N 在 M 之外”形式的完成条件无符号整数的表达式（见章节 8.3.4 无符号整数表达式）。表达式的实际值 B 在<forEach>活动开始时被计算一次。作为<forEach>活动的执行结果，它不会被改变。在每个直接附属的<scope>活动执行末端，完成孩子的数量比喻为 B—<branches>表达式的值。如果最少 B 个孩子已经完成，则<completionCondition>被触发：没有更深一层的孩子被启动，并且正在并发运行的孩子将被中断（见章节 12.6 中断处理器）。请注意强制并行<forEach>中“确定的多少 N 在 M 之外”的语义将会涉及紊乱情况，因此没有指定。

当完成条件B被计算，如果它的值大于直接附属活动N+1 的数字，则必须抛出标准故障 bpel：invalidBranchCondition。[SA00075]B和N+1 都可以是常量表达式，并且在这样的情况下，静态分析应该阻止B大于N+1 的流程。

<branches> 元素的可选 successfulBranchesOnly 属性缺省值为 no。如果 successfulBranchesOnly 值为 no，所有（成功或不成功）完成的<scope>必须被计数。如果 successfulBranchesOnly 为 yes，只有成功完成的<scope>被计数。

当每次直接附属的 <scope> 活动完成时验证 <completionCondition>。如果

<completionConditions>验证为 true，则<forEach>活动结束：

- 当<completionCondition>为并行<forEach>活动完成时，所有正在运行的直接附属<scope>活动必须被中断（见章节 12.6 中断处理器）。
- 当<completionCondition>为串行<forEach>活动完成时，更深一层的子<scope>禁止被实例化，且<forEach>活动结束。

如果在直接附属<scope>活动的完成上可以被决定<completionCondition>从不为 true，则标准故障 bpel：completionConditionFailure 必须被抛出。

当<completionCondition>没有任何子元素或属性被 WS-BPEL 处理器接受，则必须被当作<completionCondition>不存在来处理。

---

12. 作用域

<scope>提供影响它的附属活动行为处理行为的上下文。这个行为上下文包括变量，伙伴链接，消息交换，相关集，事件处理器，故障处理器，补偿处理器和中断处理器。<scope>活动提供的上下文可以分级嵌套，同时<process>构造器提供“root”根结点（见章节 8.1 变量，12.4 补偿处理器以及 12.5 故障处理器）。

有相同的语义的<process>和<scope>元素共享语法构造。尽管如此，它们还是有下列差异：

- <process>构造器不是活动，因此标准属性和元素不适用于<process>构造器。
- 补偿处理器和中断处理器不能附属于<process>构造器
- Isolated 属性不适用于<process>构造器（见章节 12.8 独立作用域）

每个<scope>有一个必需的定义普通行为的主要活动。主要活动可以是复杂构造性活动，有很多嵌套到各种深度的活动。<scope>活动的所有其他语法构造器是可选的，并且其中某些有缺省语义。<scope>提供的上下文被它所有的嵌套活动共享。

作用域的语法如下：

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
   standard-attributes>
   standard-elements
   <variables>?
      ...
   </variables>
   <partnerLinks>?
      ...
   </partnerLinks>
   <messageExchanges>?
```

```
      ...
   </messageExchanges>
   <correlationSets>?
      ...
   </correlationSets>
   <eventHandlers>?
      ...
   </eventHandlers>
   <faultHandlers>?
      ...
   </faultHandlers>
   <compensationHandler>?
      ...
   </compensationHandler>
   <terminationHandler>?
      ...
   </terminationHandler>
   activity
</scope>
```

<scope>上的所有处理器从属于<scope>并且可以访问<scope>上定义的所有变量，伙伴链接，消息交换，相关集以及其线性祖先。这个服从任何约束，对于称号类型是唯一的，在本文档的别处指定。

<scope>可以声明变量，伙伴链接，消息交换和相关集在<scope>内可见。更详细的信息，分别见章节 6.2 伙伴链接，章节 8.1 变量，章节 9.相关性 以及 章节 10.4 提供 Web 服务操作—Receive 和 Reply。

12.1 作用域初始化

当<process>或者<scope>启动时作用域开始初始化。作用域初始化由例示及正在初始化的作用域变量以及伙伴链接组成；例示相关集；以及安装的故障处理器，中断处理器以及事件处理器。任何在<scope>中定义的伙伴链接必须在相同的初始化涉及那些伙伴链接的<scope>定义的变量之前设置。作用域初始化是要么全有要么全无行为：要么全部成功发生要么抛出 bpel：scopeInitializationFailure 故障给错误<scope>的父作用域。如果发生程序级错误，则整个程序被看做有故障的。一旦作用域初始化完成，则执行<scope>的主要活动且同时安装事件处理器。先前应用到<scope>的规则的异常包括程序的初始化开始活动。初始化开始活动是导致特殊程序实例被初始化的开始活动。如果作用域包括初始化开始活动则开始活动必须在安装事件处理器之前完成。

在下面的例子中，<scope>有一个主要的包括两个并发的<invoke>活动的<flow>活动。任一<invoke>活动可以接收故障响应。<scope>的<faultHandlers>被两个<invoke>活动共享并且可以用于捕获由可能的故障相应引起的故障。

```
<scope>
   <faultHandlers>...</faultHandlers>
   <flow>
      <invoke partnerLink="Seller"
         portType="Sell:Purchasing"
         operation="Purchase"
         inputVariable="sendPO"
         outputVariable="getResponse" />
      <invoke partnerLink="Shipper"
         portType="Ship:TransportOrders"
         operation="OrderShipment"
         inputVariable="sendShipOrder"
         outputVariable="shipAck" />
   </flow>
</scope>
```

12.2 消息交换处理

当<scope>的主要活动和事件处理器完成则<scope>内部声明的依赖于伙伴链接或消息交换的所有 Web 服务交互必须结束。当 IMA 使用伙伴链接或消息交换时发生孤儿 IMA，在完成的<scope>或它的子孙里声明，保持公开。在这个情况下，标准故障 bpel：missingReply 必须被抛出。孤儿 IMA 的定义以及它们如何被检测的情况如下：

- 如果被包括的作用域的主要活动和事件处理器没有任何未处理故障地完成则孤儿 IMA 的检测必须执行。如果一个或多个孤儿 IMA 被检测到则 bpel：missingReply 故障必须抛出至父作用域（类似于从故障处理器中抛出或重新抛出其他故障）。最近被抛出的 bpel：missingReply 故障必须包括所有孤儿 IMA，并且它们不再被看做无双亲的。
- 如果故障处理器没有任何未处理故障的完成则孤儿 IMA 的检测必须执行。如果任何孤儿 IMA 被检测则新的 bpel：missingReply 被抛出至父作用域（类似于抛出或重新抛出来自故障处理器的其他故障）。最近被抛出的 bpel：missingReply 故障必须包括所有孤儿 IMA，并且它们不再被看做无双亲的。
- 如果故障处理器本身抛出或重新抛出不是 bpel：missingReply 的故障至父作用域则不用检测孤儿 IMA，并且检测遵从父<scope>。孤儿 IMA 保持原状。
- 上述条目行为可以应用于当中断处理器执行时。
- 在补偿处理器没有任何未处理故障的完成后执行相同的孤儿 IMA 的检查。如果任一孤儿 IMA 被检测到，bpel：missingReply 故障必须被传播给调用的 FCT-处理器并且那些 IMA 不在被看做无双亲的。

如果未处理的不是 bpel：missingReply 的故障在补偿处理器执行期间发生，则故障传播给正在调用的 FCT-处理器。孤儿 IMA 的检查遵循调用的 FCT-处理器。如果是任何由补偿处理的执行产生的孤儿 IMA，保持孤儿状态。

12.3. 业务流程中的错误处理

业务流程经常有一个长的持续时间。它们可以在后台数据库和业务链应用程序中熟练地操作数据。这种环境下的错误处理不仅困难而且有商业上的限制。ACID事务的应用通常局限于局部更新因为信任问题以及锁和隔绝性不能在业务流程示例中发生错误条件，技术和业务错误时长时间维持。结果，所有的业务事务在很多ACID事务被提交后失败或者取消。部分已经完成的工作必须最好是未完成的。WS-BPEL流程中的错误处理因此补充了补偿的概念，即特殊应用程序的活动试图逆转事务[Trends]。WS-BPEL通过提供逆转的柔韧性控制能力来提供提供这样的补偿机制的变量体。WS-BPEL通过提供在特殊应用方式中定义故障处理和补偿的能力来完成支持长时运行的事务（LRT's）。

这里描述的 LRT 的概念是纯局部的并且在单个业务程序实例中发生。没有关于在多方服务中的结果意见一致的分布式协调。（There is no distributed coordination necessary regarding an agreed-upon outcome among multiple-participant services.）有关分布式协议的优点是在本规范作用域外的问题。

作为一个例子，考虑移动路线的计划和实施。这个可以被看做在个体服务保留中的 LRT 可以使用总体 LRT 作用域中的嵌套事务。如果路线被取消，则保留事务必须通过注销事务赔偿，并且相应的支付事务也必须因此而赔偿。因为 ACID 事务在数据库中，事务协调程序和它们控制的资源知道所有必须被逆转的未授权的更新和命令，并且它们可以完全控制这样的逆转。在商务处理中，补偿行为自身是业务逻辑和协议的一部分，并且必须被明确指定。在这个例子中，取消依赖于票类和取消时间的航线预约可能产生损失或者费用。如果预先交付订金，则预定可以在订金以扣除订金的形式被逆转之前成功地取消。这就意味着补偿活动可能需要与初始活动相同顺序运行，这在大部分 ACID 事务系统中不是标准或者缺省的。使用<scope>活动作为工作逻辑单元的定义，WS-BPEL 标记 LRT 的这些要求的地址。

12.4 补偿处理器

声明在转发工作逻辑旁的补偿逻辑能力是 WS-BPEL 应用程序控制的错误处理框架的基础。WS-BPEL 允许作用域描述通过指定补偿处理器以应用程序定义的方式可逆的部分行为。有故障处理器和补偿处理器的作用域可以任意深度地嵌套。

12.4.1. 定义补偿处理器

语义上，<compensationHandler>是执行补偿活动的简单封装，如下所示：

```
<compensationHandler>
   activity
</compensationHandler>
```

正如在章节 10.3 调用 Web 服务操作—Invoke 中解释的，<invoke>活动有一个特殊的简化操作来内联<compensationHandler>胜于明确地使用立即附属的<scope>。如下：

```
<invoke partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
   <correlations>
      <correlation set="PurchaseOrder" initiate="yes"
         pattern="request" />
   </correlations>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
         <correlations>
            <correlation set="PurchaseOrder" pattern="request" />
         </correlations>
      </invoke>
   </compensationHandler>
</invoke>
```

在这个例子中，最初的<invoke>活动产生一个购买且如果购买需要被补偿，则<compensationHandler>在同一个伙伴链接的相同端口上调用注销操作，使用购买请求的响应作为输入。

没有<invoke>捷径这个例子如下所示：

```
<scope>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
         <correlations>
            <correlation set="PurchaseOrder" pattern="request" />
         </correlations>
      </invoke>
   </compensationHandler>
```

```
   <invoke partnerLink="Seller"
      portType="SP:Purchasing"
      operation="Purchase"
      inputVariable="sendPO"
      outputVariable="getResponse">
      <correlations>
         <correlation set="PurchaseOrder" initiate="yes"
            pattern="request" />
      </correlations>
   </invoke>
</scope>
```

请注意变量 getResponse 不是<scope>局部的而是<compensationHandler>附加的，并且可以为了另外的目的在这个<scope>调用补偿之前重新使用。非局部变量的当前状态在补偿处理器中是可用的，下面将会有更详细的解释。假设补偿处理器对于被逆转的<invoke>操作需要特殊的响应，则响应最方便的是存放在<scope>的局部变量中，也就是说通过让 getResponse 局限于<scope>。在这种情况下，变量声明需要显式<scope>。

如果作用域的<compensationHandler>没有被指定，则提供作用域的缺省补偿处理（见章节 12.5.2 缺省补偿命令 获得更详细信息）。

12.4.2. 补偿处理器中的程序状态应用

在补偿处理器被执行时，补偿处理器总是使用程序的当前状态。这个状态来自于相关的作用域和所有附属的作用域，而且包括变量，伙伴链接和相关集的状态。补偿处理器既可以读又可以写入这样的数据值。程序的其他部分将会看到补偿处理器共享的数据变化以及逆转，补偿处理器也会看到程序的其他部分共享的数据变化。万一补偿处理器同程序的其他部分同时运行，当它们涉及到附属<scope>中的状态时，补偿处理器可能需要使用隔离的作用域来避免干扰。

程序状态是由所有已经启动的作用域的当前状态组成。包括已经成功完成的但是相关补偿处理器没有被调用的作用域。对于成功完成的（但是没有补偿的）作用域，它们的状态保持在结束的时刻。这样的作用域没有正在运行，然而仍然是可获得的。因为它们的补偿处理器仍然可用，并且这种作用域的处理在它们的补偿处理器执行期间可以一直继续，可以被视为相关作用域的行为的可选择延续。作用域可以被执行多次（如在<while>或者<forEach>中），所以程序的状态包括所有成功完成（未被补偿）的迭代的作用域实例。我们访问成功完成未被补偿的作用域的保持状态作为作用域的快照。

补偿处理器的行为可以使用如同它遗留的相关作用域的状态。包括在相关作用域及所有附属作用域中的变量，伙伴链接，消息交换和相关集。因为变量在相关作用域中，补偿处理器启动执行连同作用域快照一起。补偿处理器也可以访问每个附属作用域的当前状态。状态被当前所有逻辑单元共享。补偿处理器自身可以被来自父作用域的补偿处理器调用。稍后将会共享调用者使用的附属作用域的状态的延续。

下面的图片显示了三个嵌套的作用域 P,S2 和 S3，程序的故障处理器 FH(P)和补偿处理器 CH(S2)和 CH(S3)。

下面的图片是基于 XML 的。当处理程序时，第一个作用域 P（程序本身）声明了一个变量 V1 并且将它的值初始化为 0。作用域 S2 和 S3 被处理。当 S2 和 S3 成功完成时，所有的变量被设置为 1 并且冻结到快照中（虚线显示的时间线）。随后，程序 P 中发生故障（图中有事件“1”指出的），被程序 P 的故障处理器 FH(P)捕获。当程序的故障处理器调用作用域 S2 的补偿处理器 CH(S2)（图中“2”指出的），S2 状态的快照被逆转并且同时使用补偿。同样的过程应用于补偿作用域 S3 时（图中事件“3”指出的）。

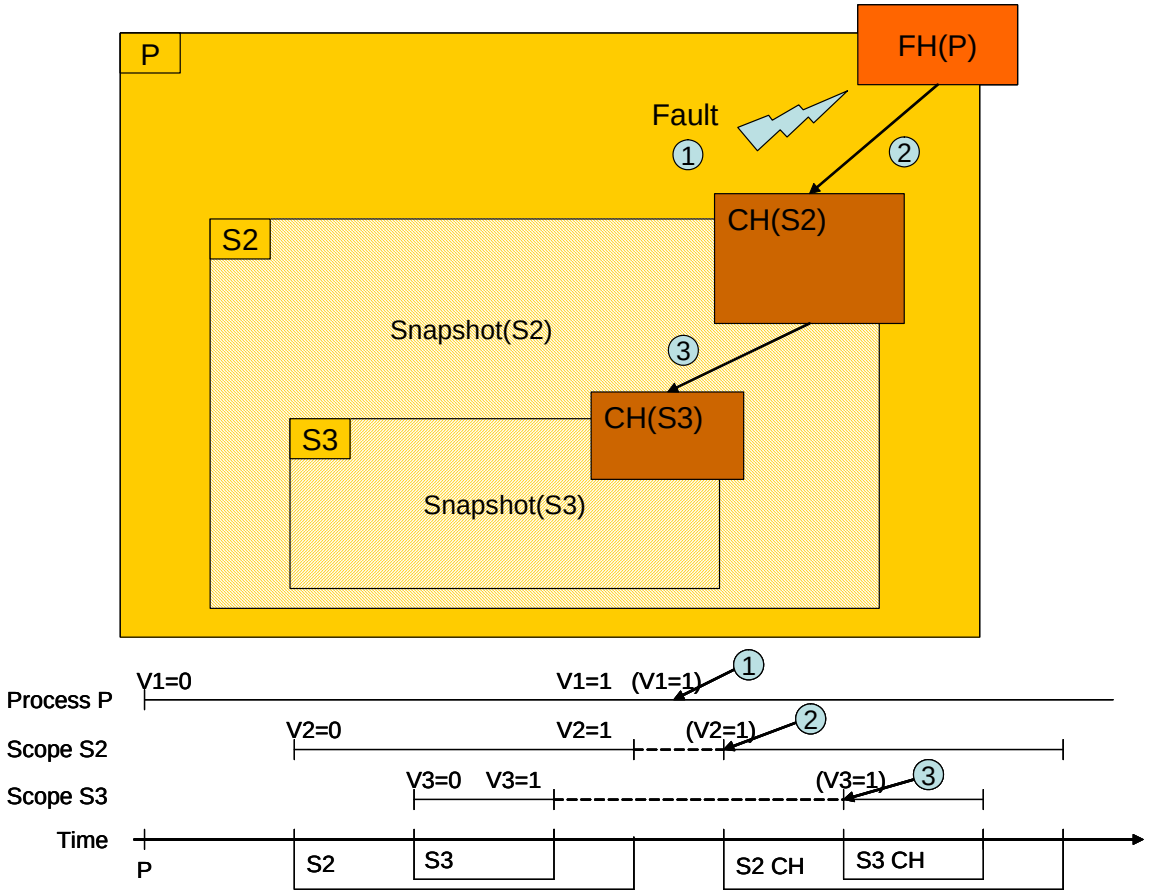


Figure 3: Variable Access in Compensation Handlers

```
<process name="P">
   <variables>
      <variable name="V1" type="xsd:int">
         <from>0</from>
      </variable>
   </variables>
   <faultHandlers>
      <catch faultName="prefix:someFault">
         <compensate />
      </catch>
```

```
   </faultHandlers>
   <scope name="S2">
      <variables>
         <variable name="V2" type="xsd:int">
            <from>0</from>
         </variable>
      </variables>
      <compensationHandler>...</compensationHandler>
      <sequence>
         <scope name="S3">
            <variables>
               <variable name="V3" type="xsd:int">
                  <from>0</from>
               </variable>
            </variables>
            <compensationHandler>
               ...
               <!-- V1, V2, and V3 ALL have the value 1
                    when this logic is reached -->
               ...
            </compensationHandler>
            <assign>
               <copy>
                  <from>1</from>
                  <to variable="V3" />
               </copy>
            </assign>
         </scope> <!-- end of scope S3 -->
         <assign>
            <copy>
               <from>1</from>
               <to variable="V1" />
            </copy>
            <copy>
               <from>1</from>
               <to variable="V2" />
            </copy>
         </assign>
         <throw faultName="prefix:someFault" />
      </sequence>
   </scope> <!-- end of scope S2 -->
</process>
```

12.4.3 调用补偿处理器

补偿处理器可以使用<compensationScope>或<compensate>调用（共同作为“补偿活动”引用）。作用域的补偿处理器必须只能在作用域成功完成时对调用可用。任何补偿作用域的尝试，不管是否安装执行补偿处理器，必须像处理的<empty>活动一样。因此，处理器不依赖状态来决定已经成功完成的嵌套作用域。

```
<compensateScope target="NCName" standard-attributes>
   standard-elements
</compensateScope>
```

```
<compensate standard-attributes>
   standard-elements
</compensate>
```

故障处理器，补偿处理器以及中断处理器作为 FCT-处理器被引用。为了指定<compensate>和<compensateScope>的语义，作用域 A 被看做直接附属作用域 B，如果 B 附属在 A 中且 B 不附属于其他作用域或 FCT-处理器，则 B 本身附属在作用域 A 外部。其他附属在 A 中的构造活动（如<sequence>或<forEach>）以及事件处理器不会影响即时封装的概念。这个定义包括由<invoke>的故障处理器和补偿处理器的简化操作标记产生的作用域。

[SA00092]在作用域内，所有已命名的附属作用域的名字必须是唯一的。这个要求必须被静态地强制。

在FCT-处理器中<compensateScope>或<compensate>活动用于补偿成功完成的附属在与FCT-处理器相关的作用域中的行为。[SA00077]<compensateScope>活动上的target属性值必须引用直接附属作用域的名字。包括与同一个作用域相关联的事件处理器的立即附属的作用域（<onEvent>或<onAlarm>）（见章节 12.7 事件处理器）。这个规则必须被静态地强制。

FCT-处理器自身可以包含作用域。补偿活动的调用基于立即附属 FCT-处理器解析并且用于已成功完成的直接附属于同 FCT-处理器相关联的作用域的补偿行为。因此无法使用补偿活动来补偿附属在 FCT-处理器内部的作用域。

12.4.3.1. 制定作用域的补偿

<compensateScope>活动导致一个指定的子作用域被补偿。例如：

```
<compensateScope target="RecordPayment"/>
```

直接附属在作用域内的已命名活动的名字必须是唯一的（见章节 10.1 所有活动的标准属性）。[SA00078]<compensateScope>活动的target属性必须引用有故障处理器或补偿处理器的作用域或调用活动。已引用的活动必须通过包含<compensateScope>活动的FCT-处理器的作用域直接附属。如果没有发现这些要求则WS-BPEL程序必须被阻止。这些要求必须被静态地强制。

12.4.3.2. 调用缺省补偿行为

<compensate>活动导致所有直接附属的作用以默认顺序被补偿（见章节 12.5.2 默认补偿顺序）。

当 FCT-处理器需要执行额外的工作时使用这个活动，如更新变量，除了为目标直接附属作用域执行默认补偿外。

用户定义的 FCT-处理器可以使用<compensateScope>活动以默认顺序来补偿指定的直接附属作用域及/或<compensate>来补偿所有直接附属的作用域。任何补偿立即附属作用域的重复行为被当作<empty>活动处理（见章节 12.4.3 调用补偿处理器）。

当用户定义的 FCT-处理器被执行时，WS-BPEL 处理器禁止补偿立即附属作用域除非<compensate>或<compensateScope>活动被使用。

12.4.4 在重复构造器或处理器内的补偿

12.4.4.1 补偿处理器实例组

放置在重复构造器内的作用域，如循环或事件处理器经常导致作用域的作用域的多实例。每次重复或事件处理器实例化时分别创建一个作用域实例。

当<compensateScope>或<compensate>活动用于调用包含重复构造器的作用域的补偿处理器时，补偿活动运行安装补偿处理器实例组并且引发子作用域的相关集被补偿。所有这样安装的补偿处理器实例化集称作补偿处理器实例组。

在作用域指定补偿（<compensateScope>）的情况下，补偿处理器实例组包括在重复构造器内执行的特殊目标作用域安装的补偿处理器实例。对于缺省补偿（<compensate>）的情况，补偿处理器实例组包括所有已经成功完成的直接附属作用域的补偿处理器实例。直接附属作用域的补偿处理器实例被当作单个组处理。

如果在执行组的补偿处理器实例时发生未捕获的故障，或者补偿活动被中断，则所有正在允许的实例必须依照标准 WS-BPEL 活动中断语义中断。所有组的补偿处理器实例和直接附属作用域的补偿处理器实例组被卸载。在补偿处理器实例组内完成的补偿处理器实例不遵从更进一步的补偿。

12.4.4.2 可重复构造器内的补偿

如果名字嵌套在<while>，<repeatUntil>或非并行的<forEach>循环正在补偿作用域，在逐次迭代里安装的补偿处理器的实例的调用必须被颠倒次序。

在并行<forEach>和事件处理器的情况下，同作用域相关的补偿不要求顺序。

12.4.4.3 FCT-处理器中的补偿

如果作用域附属在FCT-处理器内，则附属作用域的补偿处理器只在附属处理器的生存期可用。一旦处理器完成，任何安装在内的补偿处理器被卸载。[SA00079]上述三种类型处理器内部的根作用域不能有相关补偿处理器因为它根本不可能到达程序内的任何地方。因此，上述三种类型处理器内部的根作用域禁止有补偿处理器。这个规则必须被静态地强制。请注意时间处理器的根作用域（<onEvent>或<onAlarm>）可以有补偿处理器。

图形 4：故障处理器内部的补偿展示了包含作用域 S2 的故障处理器 FH(S1)。作用域 S2 不能有补偿处理器 CH(S2)因为这个补偿处理器是取不到的，但是它可以有允许补偿内部作用域 S3 的故障处理器 FH(S2)。

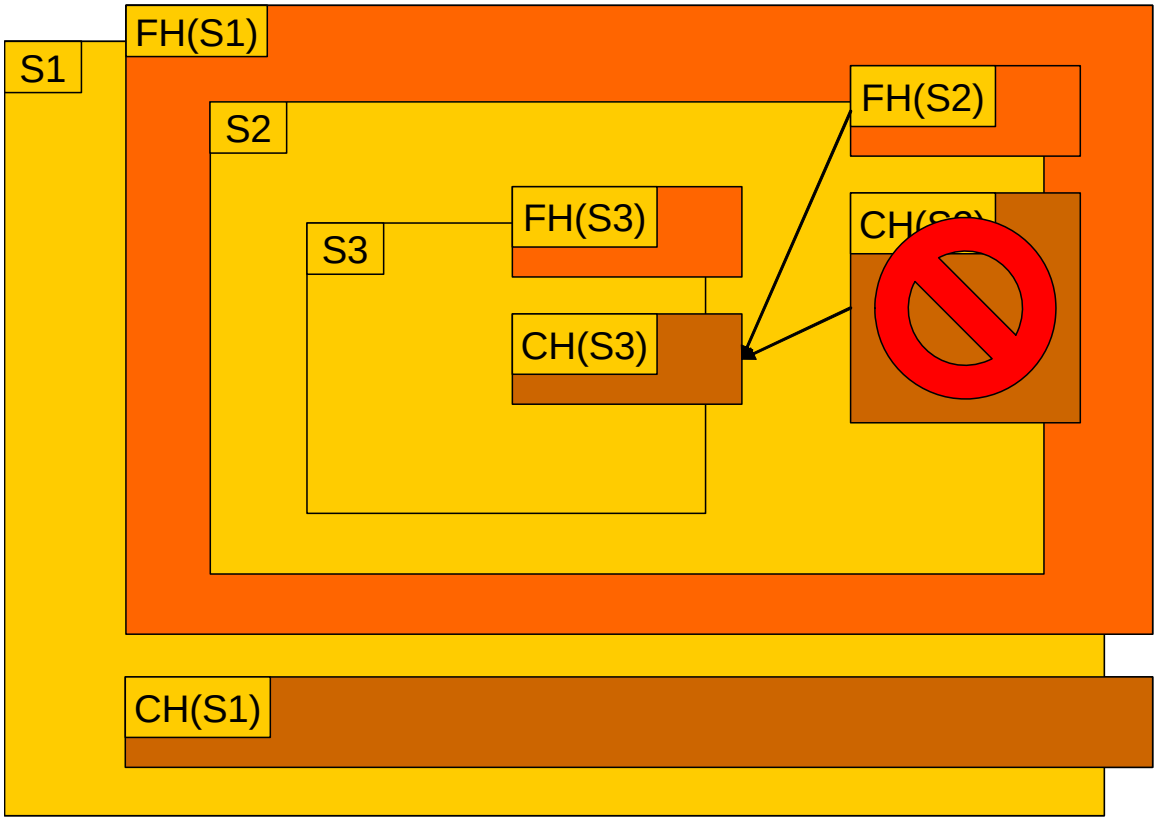


Figure 4: Compensation within Fault Handlers

故障处理器内的补偿

故障处理器内的故障必须引起所有运行包括的活动中断（见章节 12.6 中断处理器）。所有在故障处理器里的补偿处理器必须被卸载。故障被传播给附属作用域。

补偿处理器内的补偿

补偿处理器附属的根作用域可以被用于确定“全有或全无”语义，但是不能反转已经成功完成的补偿处理器。如果补偿处理器成功完成则任一作用域内部安装的补偿处理器被卸载。

成功完成的补偿不能被逆转，因为补偿处理器内部的根作用域不能有自己的相关补偿处理器因为在程序内是无法获得的。

补偿处理器出现错误（“内部故障”）将取消它的部分工作通过补偿所有根据根作用域故障处理器的根目录直接附属的作用域。如果没有明确指定这样的故障处理器，部分工作将以默认顺序补偿（见章节 12.5.2 默认补偿顺序）。这个方法可以为补偿处理器用于提供全有或全无语义。在部分工作被取消后，补偿处理器必须被卸载。故障必须被传播给补偿处理器的调用者。这个调用者是用户定义的处理器内的附属作用域或补偿活动默认的 FCT-处理器。

图形 5：补偿处理器内的补偿展示了一个包含作用域 S2 的补偿处理器 CH(S1)。正如在先前图片中，S2 不能有补偿处理器 CH（S2）自身但是可以有允许补偿内部作用域 S3 的故障处理器 FH(S2)。

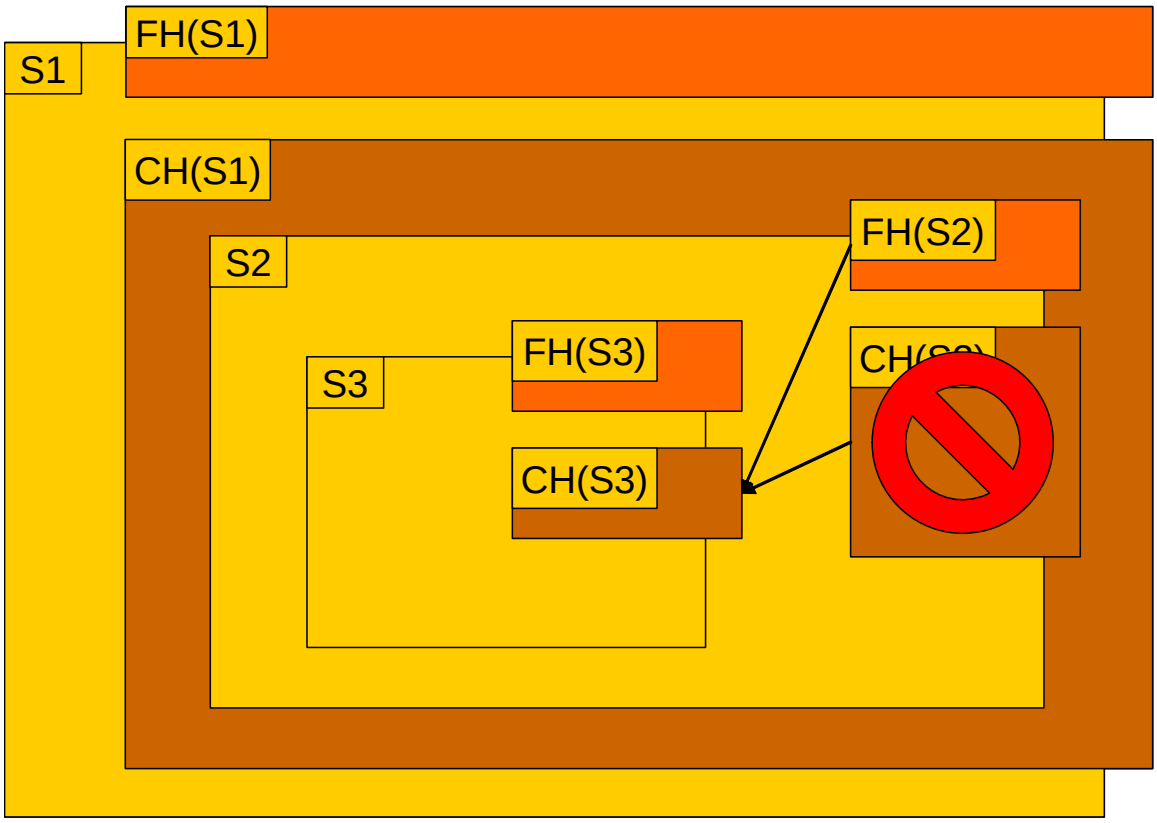


Figure 5: Compensation within Compensation Handlers

中断处理器中的补偿

中断处理器内部故障禁止传播给附属作用域（见章节 12.6 中断处理器）。除了这个，所有有关故障处理器的语句将同样应用于中断处理器。

12.5 故障处理器

业务流程中的故障处理可以被看做作用域中普通处理的开关模式。WS-BPEL 中的故障处

理被设计当作“逆转工作”一样处理，因为它的目的是取消发生故障的作用域内的部分以及未成功的工作。故障处理器活动的完成，即使在没有重新抛出故障时，不会被看作为附加作用域的成功完成。已经有相关故障处理器被调用的作用域内不能启用补偿。

如果使用显式故障处理器，附加在作用域上的提供定义客户故障处理活动集的方法，通过<catch>和<catchall>构造器定义。每个<catch>构造器被定义为截取某种具体的故障类型，通过故障 QName 定义。可选择的变量可以被提供为携带同故障相关联的数据。如果故障名缺失，则捕获器将截取所有同故障数据类型相同的故障。在<catch>故障处理器中使用 faultVariable 属性指定故障变量。变量被看做是依赖于正在被使用的这个属性值来隐式声明并且局限于故障处理器。在其声明的故障处理器外变量是不可见并且不可用的。<catchall>子句可以被添加给任意未被具体故障处理器捕获的故障。

在 WS-BPEL 中有各种各样的故障。响应<invoke>活动的故障是故障的资源之一，故障名和数据基于在 WSDL 操作中的故障定义。<throw>活动是另外一个资源，有明确给出的名字及/或数据。WS-BPEL 定义了若干标准故障的名字，并且可以有其他特殊平台的故障比如通信故障。

在 WS-BPEL 程序中可以使用的没有在别处定义的故障名，如 WSDL 操作中的例子；或者故障名可以缺失。

```
<faultHandlers>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
</faultHandlers>
```

[SA00080]在<faultHandlers>元素里至少必须有一个<catch>或<catchall>元素，将QName指定为faultName的属性值，只能捕获匹配QName的故障（见章节 10.3 调用Web服务操作-Invoke 获得有关如何从WSDL定义的故障中构造QName的信息）。在同一个WSDL名称空间中的多个WSDL操作中定义的名字相同的故障可以被同一个<catch>故障处理器捕获。如果被捕获的数据是WSDL消息则faultMessageType属性用于指定消息类型的QName。如果被捕获的数据是XML元素则faultElement属性用于指定元素定义的QName。

[SA00081]为了有同故障变量相关联的已定义类型，faultVariable属性必须只能在faultMessageType或faultElement属性之一陪伴下使用。faultMessageType和faultElement属性禁止被使用除非有faultVariable属性陪伴。

因为表达故障的灵活性所以<catch>构造可以处理一个故障可以匹配多个故障处理器。[SA00093]因为多个故障处理器可以匹配一个故障，<faultHandlers>元素禁止包括同一个

<catch>构造器。当<catch>构造器有同一个faultName，faultElement和faultMessageType属性时被看做是相同的。如果<catch>中没有表示属性，它的值被认为缺失的且只等于另一个缺失属性的<catch>。违反这个条件的程序定义必须被静态分析阻止且必须被实现拒绝。

当故障被抛出且没有任何相关数据时，故障必须被如下捕获：

1.如果有匹配没有指定 faultVariable 属性的 faultName 值的<catch>构造器则故障传给标识的捕获活动。

2.否则如果有<catchAll>故障处理器则故障被传给<catchAll>故障处理器。

3.否则，故障由默认的故障处理器处理(见章节 12.5.1 默认故障，补偿以及中断处理器)。

如果故障带有相关数据地抛出则故障必须被如下捕获：

1. 如果有匹配 faultName 值的<catch>构造器，其 faultVariable 的类型匹配运行时刻故障数据的类型则故障被传给标识的<catch>构造器（见下面的匹配协议准则）。
2. 否则如果故障数据是 WSDL 消息类型，其消息包含由元素定义的单个部分且存在同 faultName 值匹配的<catch>构造器，且构造器的 faultVariable 相关 faultElement 的 QName 匹配运行时期单个 WSDL 消息部分的元素数据的 QName，则故障被传给标识的 faultVariable 初始化至单个部分元素值的<catch>构造器（见下面的匹配协议准则）。
3. 否则如果有匹配 faultName 值的<catch>构造器没有指定 faultVariable 属性则故障被传给标识的<catch>构造器。请注意在这种情况下故障值在故障处理器内部不可用但是对于<rethrow>活动可用。
4. 否则如果有不带 faultName 属性的<catch>构造器有类型匹配运行时期故障数据类型的 faultVariable 则故障被传给标识的<catch>构造器（见下面的匹配协议准则）。
5. 否则如果故障数据是消息包含元素定义的单个部分的 WSDL 消息类型且存在没有 faultName 属性的<catch>构造器有 faultVariable 相关的 faultElement 的 QName 匹配 WSDL 消息类型部分的运行时期元素数据的 QName，则故障被传给标识的 faultVariable 被初始化至单个部分元素值的<catch>构造器（见下面的匹配协议准则）。
6. 否则如果有<catchAll>故障处理器则故障被传给<catchall>故障处理器。
7. 否则，故障将由默认的故障处理器处理（见章节 12.5.1 默认故障，补偿和中断处理器）。

在上述条目 1 和 4 中提到的将 faultVariable 的类型匹配给运行时期故障数据比条目 2 和 5 中的更严格。在条目 1 和 4 的情况下，WSDL 消息类型变量只能匹配 WSDL 消息类型故障数据，同时元素变量只能匹配基于元素的故障数据。在 WSDL 基于消息故障的情况下，只有它们的 QName 唯一时可以匹配。条目 1 和 4 使用 faultElement，条目 2 和 5 通过比较运行时期基于元素的数据和 faultElement 的 QName 来匹配。

运行时期基于元素的数据，源于带有基于 XSD 元素变量、基于 XSD 类型变量和基于 XSD 元素的单个 WSDL 消息的抛出故障，如下情况时被看做与全局声明的 faultElement 引用元素一致：

- 基于元素数据的 QName 精确匹配被引用元素的 QName，或

- 基于元素数据是引用元素为头部的置换组的成员（请注意：从属关系的关系式是可传递的但不对称）。

如果多个基于 faultElement 的<catch>构造器同基于元素故障数据一致则它们匹配优先级如下：

- 带有确切 QName 的<catch>构造器享有优先权。
- 如果没有确切匹配存在则匹配优先权给 XML 元素声明里置换组关系式的层次最少的 faultElement 的<catch>（见下面的例子）。

例如，foo：Elem1，foo：Elem2，foo：Elem3，foo：Elem4，foo：Elem5 都是全局声明的变量。Elem2 的置换组涉及到 Elem1 地被声明。相关的关系式声明在 Elem3 和 Elem2，Elem4 和 Elem3，Elem5 和 Elem4 中。假设作用域有如下故障处理器：

```
<scope>
  <faultHandlers>
    <catch faultName="foo:BarFaultName" faultElement="foo:Elem2">
      ... catch-logic-A ...
    </catch>
    <catch faultName="foo:BarFaultName" faultElement="foo:Elem4">
      ... catch-logic-B ...
    </catch>
    <catch faultName="foo:BarFaultName">
      ... catch-logic-C ...
    </catch>
  </faultHandlers>
</scope>
```

如果故障数据元素是“foo：Elem5”，则基于“foo：Elem4”的<catch>-logic-B 将被匹配。如果故障数据元素是“foo：elem3”，则基于“foo：Elem2”的<catch>-logic-A 将被匹配。如果故障元素是“foo：Elem1”，则<catch>-logic-C 将被匹配。

假设有如下例子：

```
<faultHandlers>
   <catch faultName="x:foo">
      <empty />
   </catch>
   <catch faultVariable="bar" faultMessageType="tns:barType">
      <empty />
   </catch>
   <catch faultName="x:foo"
      faultVariable="bar"
      faultMessageType="tns:barType">
```

```
      <empty />
   </catch>
   <catchAll>
      <empty />
   </catchAll>
</faultHandlers>
```

假设名为“x：foo”故障从作用域内部被抛给这个附加的<faultHandlers>构造器。如果故障没有携带故障数据将选择第一个<catch>。如果有同故障相关的故障数据，则选择第三个<catch>并且只有当故障的数据类型匹配变量“bar”的类型时，选择另外的<catchall>故障处理器。最后，故障变量类型匹配“bar”以及名字不是“x：foo”的故障将由第二个 catch 处理。所有其他的故障由<catchAll>故障处理器处理。

WS-BPEL 程序允许使用<rethrow>活动重新抛出最近的附属故障处理器捕获的原始故障。<rethrow>活动允许在任何故障处理器内部使用并且只能在故障处理器内部使用。不管故障如何被捕获以及故障处理器是否修改故障数据，<rethrow>活动总是抛出原始故障数据并且保存它的类型。

尽管补偿的使用可以是故障处理器行为的关键特征，但是故障处理器内部的活动是任意的，甚至可以是<empty>活动。当故障处理器存在时，它管理故障的处理。它可以重新抛出相同的或者不同的故障，或者可以通过执行清空来处理故障并且允许在附属作用域内继续正常的处理。

故障发生的程序或作用域被认为非正常结束（也就是说未能成功地完成），无论故障被捕获后是否重新抛出故障。补偿处理器从不在发生故障的作用域内安装。

当作用域的故障处理器完成处理发生在那个作用域内的故障，自身没有抛出故障时，将那个作用域作为资源的链接必须受状态验证结果的支配。

正如在章节 10.3. 调用 Web 服务操作—Invoke 中说明的，对于 invoke 活动有一个特殊的简化操作来内联故障处理器优于明确使用直接附属作用域。

当作用域 C 正常完成时，作用域 C 的补偿处理器通过 FCT-处理器在它的直接附属作用域内可以调用。作用域 C 的故障处理器在 C 开始且未结束时可以调用。如果作用域在完成前发生故障，则合适的故障处理器将得到控制权且其他的故障处理器和中断处理器被卸载。WS-BPEL 处理器在任何情况下禁止在一个作用域内运行多个显式或默认的 FCT-处理器。

作用域 C 的故障处理行为必须在中断所有附属在内的当前活跃的活动前开始（见章节 12.6 中断处理器）。这些活动的中断必须在故障处理器的具体活动开始前发生。这也同样用于下面描述的故障处理器。故障处理器的活动被认为发生在其附属的作用域内。

12.5.1 默认故障，补偿和中断处理器

作用域的可视化命名且因此补偿处理器被限于直接附属作用域内。因而如果直接附属作用

域没有 FCT-处理器，则补偿作用域的能力丢失。同样地很多故障不是遵循计划的或者是调用操作的结果，所以期望在每个作用域内的每个故障有一个显式的故障处理器是不合理的。因此当它们遗漏时，WS-BPEL 提供默认的故障处理器。类似的便利特征也提供给补偿处理器和中断处理器。

不论何时如果给出的<scope>缺少<catchAll>故障处理器（对于任何故障），<compensationHandler>或者<terminationHandler>，则它们必须被如下隐式地创建：

默认故障处理器：

```
<catchAll>
   <sequence>
      <compensate />
      <rethrow />
   </sequence>
</catchAll>
```

默认补偿处理器：

```
<compensationHandler>
   <compensate />
</compensationHandler>
```

默认中断处理器：

```
<terminationHandler>
   <compensate />
</terminationHandler>
```

12.5.2 默认补偿顺序

默认补偿顺序有两个规则来标记次序关系的不同特征。请注意它们是累加的，也就是说在执行默认顺序时的每种情况下都必须同时遵守它们。

非正式地，规则 1 声明在作用域正在被补偿时，默认补偿必须尊重执行的前一个命令，但是只有在活动被程序定义委托统治时。作为使用<flow>，并行<forEach>或<eventHandlers>的结果是允许并发性，并且不受链接的约束，任何处理期间的暂时逻辑命令不是第一个规则定义的约束的一部分。更正式地，我们声明规则基于严格的控制依赖标记。

定义（控制依赖）。如果活动 A 必须在活动 B 开始前完成，作为存在程序定义里从 A 到 B 的控制路径的结果，则我们称 B 在 A 上有控制依赖。请注意由于控制链接在<flow>里或者构造器类似于<sequence>，可能发生控制依赖。控制流由于显式<throw>不被看做控制依赖。

Rule1：假设作用域 A 和 B 且 B 在 A 上有一个控制依赖。假设 A 和 B 都成功完成并且都必须作为单个默认补偿行为被补偿，B 的补偿处理器必须在 A 的处理器启动前运行至完成。

在某些情况下，单个故障信号可以触发多个默认补偿行为。上述规则 1 将应用于每个补偿行为。

规则 1 允许作用域被正向并发处理，也允许被逆向并发补偿。规则利用命令上的约束在补偿处理器在补偿期间运行在附属作用域的默认处理器中，并且不是意图充分说明有关确切命令和并发性。

当然，如果遵循严格的完成的逆顺序，则有必要考虑控制依赖并且是同这个规则一致的。

非正式地，因为不是所有的作用域都是孤立的所以需要第二个规则（见章节 12.8 孤立作用域）。依照语法两个作用域可以有跨活动的链接，并且这样的链接可以双向跨越（见章节 11.6.2 链接语义）。如果作用域都是孤立的则这样是非法的。跨孤立作用域边界的链接语义暗示这样的双向链接组成一个圈。规则 2 的目的是处理所有的作用域就像它们是孤立的一样，只是为了循环检测有关跨越作用域边界的链接。这允许我们将规则 1 用于任何成对的作用域来模糊判定在两者之间是否有控制依赖，如果有，是在哪个方向。正式地，我们需要三种定义来说明规则。

定义（同级作用域）.如果作用域 S1 和 S2 都直接附属在同一个作用域内（包括程序作用域）则它们是同级作用域。

定义（受控作用域集）.如果活动 A 是作用域 S 本身，或者 A 任意深度地附属在 S 内部，则活动 A 在作用域 S 的活动的受控作用域集内。

定义（同级作用域依赖关系）.如果 S1 和 S2 是同级作用域且 S2 受控作用域集中的活动 B 对 S1 受控作用域集中的活动 A 有控制依赖关系，则称 S2 对 S1 有直接同级作用域依赖关系。同级作用域依赖相关性是直接同级作用域以来相关性的传递闭包。

规则 2：[SA00082]同级作用域依赖相关性禁止包括圆圈。换句话说，WS-BPEL禁止有两个作用域S1 和S2 的程序其中S1 对S2 有同级作用域依赖且S2 对S1 也有同级作用域依赖。包含循环的同级作用域以来相关性的程序定义必须被拒绝。这个必须由静态分析强制。

在下面的例子中，作用域“SC1“和“SC2”是同级作用域且它们的附属作用域是程序作用域“P1”。活动“InvA”和“RcvB”在作用域“SC1”的活动的受控作用域集内，同时“InvB”和“RcvA”在作用域“S2”的活动的受控作用域集内。作用域“SC1” 因为控制链接“LinkA”被看做对作用域 “SC2”有同级作用域依赖。因为控制连接 “LinkB”，在反方向上有一个同级作用域依赖。因此，因为这个循环依赖，程序定义不会被 WS-BPEL 处理器接受。

```
<process name="P1">
   ...
   <flow name="F1">
      ...
      <scope name="SC1">
         <flow name="F2">
```

```
            ...
            <invoke name="InvA" ...>
               <targets>
                  <target linkName="LinkA" />
               </targets>
            </invoke>
            ...
            <receive name="RcvB" ...>
               <sources>
                  <source linkName="LinkB" />
               </sources>
            </receive>
            ...
         </flow>
      </scope>
      <scope name="SC2">
         <flow name="F3">
            ...
            <invoke name="InvB" ...>
               <targets>
                  <target linkName="LinkB" />
               </targets>
            </invoke>
            ...
            <receive name="RcvA" ...>
               <sources>
                  <source linkName="LinkA" />
               </sources>
            </receive>
            ...
         </flow>
      </scope>
   </flow>
   ...
</process>
```

规则 2 的作用之一是允许深度优先遍历默认补偿，遵守在规则 1 规定的同级作用域之间的控制依赖相关性。由这些规则产生的作用域的默认补偿顺序只依赖于其嵌套作用域的补偿。这些规则委任的默认补偿顺序与严格的完成的逆转顺序是一致的。严格的完成的逆转顺序应用于所有作用域的补偿可能不会执行深度优先顺序，并且可以要求分解嵌套的跨越同级作用域的补偿。要求分解嵌套的跨越统计作用域的流程是被上面的规则禁止的。

12.5.3 补偿处理器和孤立作用域之间的相关性

补偿处理器可以同其他活动包括其他的补偿处理器一起并发运行，因此有必要允许补偿处理器使用孤立作用域语义（见章节 12.8 孤立作用域）。补偿处理器不会在相关联的作用域的隔离域内运行，但是故障处理器会。这会对在作用域内嵌套一个隔离作用域的补偿处理器的隔离语义制造困难。这样的补偿处理器禁止使用孤立作用域本身因为孤立作用域不可以被嵌套。不管怎样，它们的隔离环境是不确定的因为它们可以从附属作用域的隔离域内的故障处理器或从同一个附属作用域但是没有在隔离域内的补偿处理器中调用。

为了确定行为一致性，WS-BPEL 委任孤立作用域的补偿处理器自身可以有隐式的隔离行为，虽然这会从它相关的作用域中创建一个单独的隔离域。

12.5.4 处理WS-BPEL标准故障

如果作用域上的 exitOnStandardFault 属性值为“yes”，当任何 WS-BPEL 标准故障除了 bpel：joinFailure 到达作用域时，程序必须立即退出，就像<exit>活动已经到达一样。如果属性值设为“no”，则程序可以使用故障处理器处理 WS-BPEL 标准故障。这个属性的默认值是“no”。当<Scope>没有指定这个属性时，它从附属的<scope>或<process>继承值。

12.6 中断处理器

作用域 C 的故障处理器的行为通过禁止作用域的事件处理器开始并且隐式地终止所有当前运行的附属在 C 内的活动（包括所有正在运行的事件处理器实例）。请注意<forEach>中的完成条件也可以触发附属作用域中断。下面的段定义了必须被所有 WS-BPEL 活动类型遵守的规则。

<assign>活动是十分短命的，因此当被迫中断时它们可以被允许完成而不是中断。已经启动的验证表达式允许完成。强迫中断也可以被允许如果 WS-BPEL 没有要求赋值和表达式验证的特殊活动。

每个<wait>，<receive>，<reply>和<invoke>活动必须被过早地中断。当请求-响应<invoke>被过早地中断时，这样的中断活动的响应（如果接收到）必须被忽略。

<empty>，<throw>和<rethrow>活动可以被允许完成。一旦启动<exit>活动，禁止被中断。

所有构造活动行为是可以被中断的。<while>，<repeatUntil>和串行<forEach>的重复必须是可中断的并且中断必须应用到循环体活动。对于并行<forEach>，中断必须应用到所有并行执行分支。如果<if>或<pick>活动已经选择了分支，则中断必须应用于被选择分支的活动。如果这些活动还没有选择分支，则<if>或<pick>活动本身必须被立即中断。<sequence>和<flow>构造器必须终止，通过中断它们的行为以及将中断应用到所有当嵌套的当前活跃的活动。

<compensationScope>和<compensate>活动必须被中断，通过将中断传播给被调用的补偿处理器实例以及将中断应用到补偿处理器的活动中。

中断处理器提供作用域控制某种程度的强迫中断语法的能力。语法如下：

```
<terminationHandler>
   activity
</terminationHandler>
```

作用域的强迫中断通过禁用作用域的事件处理器以及中断它的主要活动和所有正在运行的事件处理器实例开始。根据这个，为作用域定制的<terminationHandler>，如果存在的话是运行状态。另外，默认的中断处理器也是运行状态。

作用域的强迫中断只应用在作用域处于正常处理模式下。如果作用域以及调用故障处理行为，则中断处理器被卸载，并且强制中断不生效。已经激活的故障处理被允许完成。如果故障处理器自身抛出故障，这个故障被传播给下一个附属作用域。

作用域的中断处理器允许使用同故障处理器相同范围的活动，包括<compensateScope>或<compensate>。尽管如此，中断处理器不能抛出任何故障。即使在它的行为期间发生未捕获的故障，也不能被重抛给下一个附属作用域。这是因为：（a）附属作用域已经有故障或者在正在中断的程序内，这导致嵌套作用域的强制中断或（b）正在被中断的作用域是并行<forEach>的分支以及提前完成机制触发中断，如<forEach>的<completionCondition>被实现。

中断处理器中的故障必须导致所有正在运行的活动被中断（见章节 12.4.4.3 FCT-处理器内的补偿）。

嵌套作用域的强制中断发生按照里层优先的顺序是（上述）规则—中断处理器在中断它的主要活动后运行的结果。

12.7 事件处理器

每个作用域，包括程序作用域可以有事件处理器集。这些事件处理器可以并发运行并且当相应事件发生时可以被调用。这个在事件处理器内的子活动必须是<scope>活动。事件有两种类型。第一种，事件可以是符合 WSDL 操作的输入式消息。第二种，事件可以是警报器，在用户设置的时间后离开。同作用域或程序相关联的事件处理器集的语法是：

```
<eventHandlers>?
   <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
```

```
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>
   </onEvent>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )?
      <repeatEvery expressionLanguage="anyURI"?>?
         duration-expr
      </repeatEvery>
      <scope ...>...</scope>
   </onAlarm>
</eventHandlers>
```

[SA00083]事件处理器必须包括至少一个<onEvent>或<onAlarm>元素。这个必须被静态地强制。

<onEvent>里的 portType 属性是可选的。如果包括 portType 属性，则 portType 的属性值必须匹配伙伴链接的 myRole 属性暗指的 portType 值。<onEvent>的所有实例必须确切使用 messageType，element 或<fromParts>中的一个。

事件处理器被看做作用域标准行为的一部分，不像 FCT-处理器。

<onEvent>和<onAlarm>里附属的活动必须是<scope>。

当讨论事件处理器，下面两项用于解释语义：

- 相关作用域：作用域直接定义在<onEvent>或<onAlarm>内
- 祖先作用域：事件处理器的附属作用域<scope>或<process>元素链

12.7.1 消息事件

<onEvent>元素指出等待消息到达的指定事件。这个元素的解释和它的属性非常类似于<receive>活动。partnerLink属性引用期望到达的消息上包含myRole端点引用的伙伴链接。[SA00084]partnerLink引用必须按照以下顺序解析在程序里声明的伙伴链接：相关作用域第一，然后才是祖先作用域。这个要求必须在静态分析期间强制。带有<receive>的partnerRole端点引用被忽略目的是执行事件处理器的接收语义。portType和operation属性定义端口类型

并且伙伴调用的操作是为了使事件发生。

Viriable属性，如果存在的话，定义了属于包含从伙伴接收的消息的事件处理器的局部变量。[SA00087]messageType属性通过引使用Qname的消息类型定义指定了变量的类型。变量（由messageType属性指定）的类型必须同operation属性引用的操作定义的输入消息类型一致。如果被接收的消息有一个单独部分并且那个部分同元素类型一起被定义，则messageType属性可以被省略以及代替element属性。元素类型必须精确匹配元素属性引用的元素类型。Variable和messageType/element属性组成在事件处理器相关作用域内的指定名字和类型的变量的隐式声明。如果element属性被使用，则产生输入消息到<onEvent>事件处理器中声明变量的绑定，如章节 10.4 提供Web服务操作-Receive和Reply中指定的接收活动。

Variable属性其中之一的用法是<fromPart>元素集合的用法。在<onEvent>元素上使用的<fromPart>的语法和语义同章节 10.4 提供Web服务操作—Receive和Reply 中receive活动的相同。[SA00085]这包括如果<fromPart>元素在<onEvent>元素上使用则variable，element和messageType属性禁止在同样的元素上使用的约束，以及[SA00047]有关variable属性或<fromPart>元素的非强制性规则。当使用<fromPart>元素时，每个<fromPart>元素组成一个事件处理器相关作用域内指定变量的隐式声明。变量类型源自于相关消息部分的类型。WSDL操作的消息类型可以不带歧义地被演绎，因为WS-BPEL不支持WSDL重载操作（见章节 3. 同其他规范的关系）。

<fromPart>元素引用的variable属性或<onEvent>元素的variable属性在事件处理器的相关作用域内隐式声明。[SA00086]相同名字的变量禁止在相关作用域内显式声明。这个要求必须被静态分析强制。

[SA00090]如果在<onEvent>元素里使用variable属性，则必须在<onEvent>元素里提供messageType或element属性。这个要求必须在静态分析期间被强制。

在输入消息的接收上，事件处理器在执行下一个事件处理器附属的<scope>活动之前将输入消息赋值给变量。自从变量在事件处理器相关联的作用域内声明，事件处理器的每个实例（不管相对于其他实例是并行处理还是串行处理）包括变量的私有拷贝，不与其他实例共享。[SA00095]变量引用值被解析给相关作用域并且禁止被解析给祖先作用域。

<onEvent>事件处理器指定的操作可以是单向或者请求-响应操作。在后面的案例中，事件处理器期望使用<reply>活动来发送响应。

<correlation>的用法在以下情况同<receive>活动相同：可以从事件处理器的输入式消息操作，到使用在相关作用域内声明的相关集。. [SA00088]<correlation>引用的相关集的分解顺序必须是：相关作用域在前，祖先作用域在后。

```
<scope name="S1">
   <compensationHandler>
      <sequence>
         <compensateScope target="S2" />
      </sequence>
```

```
   </compensationHandler>
   <eventHandlers>
      <onEvent partnerLink="travelAgency"
         portType="ns:agent"
         operation="travelUpdate"
         messageType="ns:travelStatsUpdate"
         variable="travelUpdate">
         <correlations>
            <correlation set="travelCode" initialize="no" />
            <correlation set="updateCode" initialize="yes" />
         </correlations>
         <scope name="S2">
            ...
            <correlationSets>
               <correlationSet name="updateCode"
                  properties="ns:updateCode" />
            </correlationSets>
            ...
         </scope>
      </onEvent>
   </eventHandlers>
   ...
</scope>
```

在这个例子中程序管理客户的旅行记录并且需要处理来自旅行预定系统的记录更新。<onEvent>构造器用于接收和更新使用同 travelCode 相关集相关的消息，在程序别处定义和初始化。尽管如此，有时候事件处理器需要接触旅行预订系统来探查消息更新。为了做这个输出消息不仅需要 travelCode 相关集中的值，还需要旅行更新消息包括的更新代码里的值。updateCode 相关集，在<onEvent>构造器进来时局部声明。当更新消息被接收时初始化 updateCode 相关集并且它的值只对<onEvent>事件处理器实例可用。

作用域 S2 直接附属于作用域 S1。作用域 S1 的补偿处理器调用作用域 S2 上同<onEvent>事件处理器相关联的补偿处理器。如果 S2 的补偿处理器被调用，则变量接收<onEvent>事件处理器的消息且在相关作用域内声明的相关集对补偿处理器可见，并且作为作用域快照的一部分。

<onEvent>的语义同关于 variable 属性或<fromPart>元素选择性的接收活动，紊乱条件的处理以及关于矛盾接收操作的同时启动的约束相等。最后一种情况，见章节 10.4 提供 Web 服务操作—Receive 和 Reply 中 bpel：conflictingReceive 故障和它的相关语义。

当<onEvent>元素中指定的操作是请求-响应操作时，消息交换用于关联来自<reply>活动的响应和<onEvent>元素指定的输入式消息操作。消息交换总是用于把请求和响应操作配成一对。即使当messageExchange属性没有在<onEvent>元素上显式指定也是如此，因为默认消息交换的属性表示用法省略（见章节 10.4.1 消息交换）。[SA00089]当messageType属性被显式

指定时，messageType属性引用的消息交换的解析命令必须是先相关作用域后祖先作用域。

事件处理器不会带有 creatInstance 属性，因为事件处理器不能被激活直到创建程序。

当消息组成的事件到达，在相关事件处理器指定的<scope>活动被执行。激活业务流程同时接收这样的消息和事件处理器附属的作用域的正常活动以及其他事件处理器实例。这允许在相应作用域（可以是整个业务程序实例）活跃时，事件可以在任意时刻任意次数地发生。

下面的例子显示了事件处理器通过外部消息立即暂停程序实例的用法。这个事件处理器附属于<process>作用域并且因此在整个业务流程实例的生命周期内可用。

```
<process name="orderCar">
   ...
   <eventHandlers>
      <onEvent partnerLink="buyer"
         portType="ns:car"
         operation="haltOrder"
         messageType="ns:haltOrderMsgType"
         variable="haltDetails">
         <scope>
            <exit />
         </scope>
      </onEvent>
      ...
   </eventHandlers>
   ...
</process>
```

在这个例子中，如果买家调用 haltOrder 操作，则执行<exit>活动，导致程序实例的立即中断除了正在被补偿的工作。作为选择，事件处理器可以抛出故障使正在进行的工作中断并且被补偿。

12.7.2 时钟事件

<onAlarm>元素标记一个受时间控制的事件。在<onAlarm>元素中，<for>和<until>表达式是互斥的。至少必须有一个<for>，<until>或<repeatEvery>表达式。<for>表达式指定了时间被发送后的时间期限。当父母作用域（直接附属在事件处理器内的作用域）启动时在整段时期内的时钟及时启动。可选择的<until>表达式当时钟被激活时指定具体时间点。在任意的<onAlarm>事件里只能发生这两个表达式之一。如果<for>指定的持续时间值为 0 或负数，或者在<until>中指定的最终期限已经到达或者过去，则马上处理<onAlarm>事件。可选择的<repeatEvery>表达式也指定时间期限。当<repeatEvery>表达式被指定，时钟每次都会被激活直到持续时间周期期满，当父母作用域活跃时。<repeatEvery>表达式可以同<for>或<until>表达式一起指定。如果<repeatEvery>表达式被单独指定，则第一个持续时间的时钟在父母作用域开始时启动。如果<repeatEvery>表达式同<for>或者<until>表达式一起被指定时，第一

个时钟不会被激活直到<for>或<until>表达式指定的时间期满；其后在<repeatEvery>表达式指定的时间间隔内被重复激活。当父母作用域启动时开始计算<repeatEvery>的持续时间。如果<repeatEvery>指定的持续时间值为 0 或者负数，则标准故障 bpel：invalidExpressionValue 必须被抛出。

12.7.3 事件的启动

当父母作用域启动时激活同作用域相关的事件处理器。如果事件处理器附属于<process>作用域，则程序实例被创建时启动事件处理器。这允许全局时钟事件的时钟时间被指定使用创建程序实例的消息内提供的数据，如下所示：

```
<wsdl:definitions
   targetNamespace="http://www.example.com/wsdl/example" ...>
   <wsdl:message name="orderDetails">
      <wsdl:part name="processDuration" type="xsd:duration" />
   </wsdl:message>
</wsdl:definitions>
```

上面的消息用在：

```
<process name="orderCar"
   xmlns:def="http://www.example.com/wsdl/example" ...>
   ...
   <eventHandlers>
      <onAlarm>
         <for>$orderDetails.processDuration</for>
         ...
      </onAlarm>
      ...
   </eventHandlers>
   ...
   <variables>
      <variable name="orderDetails" messageType="def:orderDetails" />
   </variables>
   ...
   <receive name="getOrder"
      partnerLink="buyer"
      operation="order"
      variable="orderDetails"
      createInstance="yes" />
   ...
</process>
```

<onAlarm>元素指定了一个计时器事件当越过 orderDetails 变量的 processDuration 部件中指

定的持续时间时激活。部件值通过接收包含请求细节消息的 getOrder 活动提供并且为请求创建程序实例。

12.7.4 事件的处理

下面的子目录提供在时钟或消息事件的处理期间必须遵守的规则。

12.7.4.1 时钟事件

当父母作用域开始时启动持续时间的时钟。当指定时刻或持续时间已经到达时，时钟事件消失。除了<repeatEvery>时钟，时钟事件在包含的活跃作用域内最多只执行一次；在父母作用域启动后余下的生命周期内禁用事件并且已经处理指定的流程。当父母作用域活跃，<repeatEvery>时钟事件在持续时间期满前被重复创建。如果<repeatEvery>指定的持续时间值为 0 或者负数则标准故障 bpel：invalidExpressionValue 必须被抛出。

12.7.4.2 消息事件

当合适的消息被接收时发生消息事件。当消息事件发生时，相关联的<scope>活动被处理。不管怎样，事件处理器保持激活，即使在并发应用中。当父母作用域活跃，特定的消息事件可以发生多次（见章节 12.7.7 并发考虑 中有关并发的考虑）。

12.7.5 事件的无力化

当作用域的主要活动结束时，它包含的所有事件处理器停用。已经运行的事件处理器实例必须完成，并且作用域的完成整体上延迟直到它们结束。

12.7.6 故障处理的考虑

当<onEvent>自身或者其相关作用域指定的输入式消息操作中发生故障（如 bpel：invalidVariables 或 bpel：conflictingReceive），则故障必须首先被传播给相关作用域。如果没有处理，则故障将会被传播到祖先作用域链。

12.7.7 并发考虑

多个<onEvent>或<onAlarm>事件可以同时发生并且它们被当作并发活动处理即使它们同来自相同伙伴链接的请求-响应操作对应。它们同样适用于对给出的伙伴链接上的请求-响应操作最多有一个突出的请求的约束（见章节 10.4 提供 Web 服务操作—Receive 和 Reply 中 bpel：conflictingRequest 的相关语义）。

当考虑到事件处理器的并发调用时，包括有<repeatEvery>表达式的<onEvent>和

<onAlarm>，孤立作用域可以用于控制共享变量的使用权限（见章节 12.8 孤立作用域）。

12.8 孤立作用域

当作用域的 isolated 属性值为“yes”时，提供共享资源的并发访问控制：变量，伙伴链接以及控制依赖链接。这样的作用域叫孤立作用域。Isolated 的缺省属性值是“no”。

假设两个并发孤立作用域 S1 和 S2，选取变量和伙伴链接的公共集来进行读取或者写入操作。孤立作用域的语义确定结果不会是相异的如果所有在共享变量和伙伴链接上的冲突活动（读/写 和 写/写活动）概念地重新排序以致于 S1 中所有这样的活动在 S2 中的活动之前结束，反之亦然。相同的隔离语义也应用在属性中。属性仅仅是变量的影射并且因此总是成对地出现。访问属性与访问变量是一致的，由附属的孤立作用域控制。请注意孤立作用域的语义非常类似于数据库事务中使用的标准隔离级“serializable”。用于确认这个的实际机制是相关实现。

[SA00091]孤立作用域禁止包括其他孤立作用域，但是可以包括没有被标记为孤立的作用域。在后面的情况下，访问来自于这样一个附属作用域的共享变量有附属孤立作用域控制。

作用域引用的消息交换只提供访问相关伙伴链接的某一面状态的处理并且本质上是无状态的。因此，附属孤立作用域的外部控制不提供消息交换。

孤立作用域引用的任何伙伴链接有自己的被附属作用域保护的通路。保护明确被应用于 endpointReference 部件，而不是伙伴链接状态的消息交换部件。

定义上，相关集只有在初始化时可变并且在剩下的生命周期内是不可变的。因此孤立作用域内引用的任何相关集不会有自己的由附属作用域控制的通路。尽管如此，相关集的初始化以原子的方式执行—同<assign>操作方式相同—确保相关集不会被部分初始化。

孤立作用域使用的处理器必须遵循以下规则：

- 孤立作用域的事件处理器共享相关作用域的隔离域。孤立作用域禁止被嵌套的规则也适用于事件处理器的相关作用域。
- 孤立作用域的故障处理器共享相关作用域的隔离域。万一在孤立作用域内发生故障，故障处理器的行为被看做是隔离行为的一部分。
- 孤立作用域的中断处理器共享相关作用域的隔离域。当调用孤立作用域的中断处理器时，它的行为被看做是隔离行为的一部分。
- 孤立作用域的补偿处理器不会共享相关作用域的隔离域。当孤立作用域的普通处理完成时，隔离域结束并且作用域快照被创建。然后，补偿处理器被卸载，如果补偿处理器的调用者（如<compensate>或<compensateScope>活动或者隐式调用直接附属作用域的FCT-处理器）没有在隔离域内，则同孤立作用域相关的补偿处理器的处理将会被隐式地隔离。当补偿处理器结束时隐式隔离域结束（见下面例子中的作用域“FH_P”和作用域“Q”）。如果补偿处理器的调用者已经在隔离域中并且被调用的补偿处理器同孤立作用域相关联，则作用域是嵌套的孤立作用域，必须被静

态分析禁止（如果下面的作用域“FH_P”是孤立的，则这样的作用域定义是被禁止的）。（见[SA00091]）。

```
<scope name="P">
  <faultHandler>
    <catchAll>
      <scope name="FH_P">
        <sequence>
          ...
          <compensate/>
          ...
        </sequence>
      </scope>
    </catchAll>
  </faultHandler>
  <sequence>
    ...
    <scope name="Q" isolated="true">
      <compensationHandler>
        <sequence name="undoQ_Seq">...</sequence>
      </compensationHandler>
      <sequence name="doQ_Seq">...</sequence>
    </scope>
    ...
  </sequence>
</scope>
```

在上面的例子中，<compensate/>活动没有在隔离域中（假设作用域“P”不是程序的根目录）。作用域“Q”的补偿处理器的处理将会被自动地隔离。当作用域“Q”的补偿处理器处理结束时，隔离域结束。

当调用者已经在隔离域内时，带有非孤立作用域的补偿处理器实际上共享补偿处理器的调用者的隔离域（见下面例子中的作用域“FH_X”）。

```
<scope name="X">
  <faultHandler>
    <catchAll>
      <scope name="FH_X" isolated="true">
        <sequence>
          ...
          <compensate />
          ...
        </sequence>
      </scope>
```

```
    </catchAll>
  </faultHandler>
  <sequence>
    ...
    <scope name="Y">
      <compensationHandler>
        <sequence name="undoY_Seq">...</sequence>
      </compensationHandler>
      <sequence name="doY_Seq"></sequence>
    </scope>
    ...
  </sequence>
</scope>
```

在上面的例子中，<compensate/>活动将在作用域“FH_X”的隔离域中调用作用域“Y”的补偿处理器（执行序列“undo_seq”）。

离开孤立作用域的链接状态（见章节 11.6.2 链接语义）在目标不可见直到作用域完成，不管成功或未成功地。如果作用域未成功完成，则离开作用域的链接状态为 false 不管那个时候源活动结束。进入孤立作用域的链接没有特定的规则。

---

13. WS-BPEL抽象流程

抽象流程有多种使用情况。因此，为定义使用公共基准的抽象流程提供了一个方法，可以用标记来区分不同的使用情况。公共基准在章节 13.1 公共基准中定义，指出了定义抽象流程语法域的特征。尽管如此，公共基准没有明确定义的语义。指定的这个公共基准中应用标记定义了必要语法约束和基于可执行抽象流程的特殊使用情况的 WS-BPEL 处理器的语义。每个抽象流程必须识别定义它的意图的应用标记。标记使用 URI 识别。这个方法是可扩展的；可以定义新的标记因为可以识别不同的区域。这些标记可以在别处定义，不在本规范之内。

来自公共基准创建的标记以及它们的特点在章节 13.2 抽象程序标记以及抽象流程的语义中定义。本规范中提供两个标记。

13.1 公共基准

公共基准是所有 WS-BPEL 抽象流程必须遵守的“语法形式”。公共基准的语法特性是：

1. abstractProcessProfile 属性必须存在。它的值引用存在的标记定义。
2. 可执行流程的构造器被允许。因此，在抽象和可执行流程之间没有基本的表达力区

别。

3. 某些 WS-BPEL 可执行流程中的语法构造器可以被隐藏，显式地通过包含 opaque language extension，并且隐式地通过 omissions，如下在章节 13.1.3 隐藏语法元素 中所示。
4. 抽象程序必须依照章节 13.1.4 语义有效性约束 定义的语义有效性约束。
5. 抽象程序可以省略可执行 WS-BPEL 流程命令的 creatInstance 活动（<receive>或<pick>）。

13.1.1 URI

抽象流程语义在下面的名称空间中表示：

```
http://docs.oasis-open.org/wsbpel/2.0/process/abstract
```

13.1.2 抽象程序的架构

抽象程序的架构只在允许的属性上不同于可执行程序，如下所示：

```
<process name="NCName"
   targetNamespace="anyURI"
   abstractProcessProfile="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract">
   ...
</process>
```

抽象流程额外的顶层属性如下：

- abstractProcessProfile。这个抽象流程的强制属性提供识别抽象程序标记的 URI。

13.1.3 隐藏语义元素

章节 13.1 公共基准 条款[3]中提到的隐藏语义元素，如下表示：

不透明语言扩充

由不透明标记组成语言扩展性作为缺损详细资料的显式占位符使用。请注意不透明标记不是新的语法上有意义的构造器而是指示不完全性的语法设备。同样地，不透明实体没有自己的语法。

不透明占位符有四种：表达式，活动，属性和 from-spec。用法标记可以限制不透明标记的种类。例如，标记可以指定只允许不透明活动，而不是其他类型的不透明标记，或者标记可以指定允许所有的属性是不透明的除了 partnerLink 属性。不管怎样，用法标记禁止在“公共基准”上允许的基础上扩充不透明性。例如，标记不能指定故障处理器为不透明的。

每个不透明标记是相应的可执行 WS-BPEL 构造器的占位符，下面将会描述。在每个抽象程序的可执行完成中构造器可以是不同的（见章节 13.1.4 语义有效性约束）。例如，抽象程序中的不透明活动可以在一个可执行程序中表示<assign>和在其他可执行流程中的<empty>，都是抽象程序的有效完成。

公共基准允许下列在抽象流程中的不透明用法：

- 允许不透明活动。
- 所有 WS-BPEL 表达式允许是不透明的。
- 所有 WS-BPEL 属性在公共基准中允许是不透明的。
- From-spec（如在<assign>中）允许是不透明的。

在抽象流程中允许四种不透明标记类型的函数（活动，表达式，属性和 from-spec）如下描述，连同例子一起：

不透明活动

不透明活动是一个明确的可执行 WS-BPEL 活动的显式占位符，并且任何活动可以被嵌套在那个活动中。可执行 WS-BPEL 活动使用所有由它替换的不透明活动定义的非不透明的元素/属性。它也替换是不透明活动某部分的不透明属性或者表达式。

不透明活动有同所有 WS-BPEL 活动相同的标准元素和属性（见章节 10.1 所有活动的标准属性 和章节 10.2 所有活动的标准元素）。不透明活动有如下形式：

```
<opaqueActivity standard-attributes>
   standard-elements
</opaqueActivity>
```

一个使用不透明活动的例子是在标记程序扩展点的程序模板创建中。另一个是在从已知的可执行程序中创建抽象流程时隐藏若干链接连接点的活动。另一方面，如果那个活动是在<sequence>里的没有链接的活动，它可以被来自合成的抽象程序省略。这个不能使用<empty>活动来完成，因为<empty>明确地意味“这里没有发生任何事”。然而<opaqueActivity>表示“这里发生了某些事，但是故意隐藏了”。

为一个活动（并且不能是 0 个或者多个）提供一个不透明活动的原因是在活动没有模糊的关于存储任何在不透明活动上或者是在它的父母的相关性或者兄弟姐妹活动中定义的属性或者元素。

不透明表达式

不透明表达式是相应可执行 WS-BPEL 表达式的占位符。

不透明表达式的典型用法是赋值一个隐藏值到已知变量中。不透明表达式可以用于不确定的情况中：明显的案例是一个程序需要展示出口的判定点且不指定地区昂判定怎样到达。在这种情况下，表达式强迫每个分支可能需要被遗弃。尽管如此，它也可以很方便地使指定值或者数量就像为指定的价格限值一样，所以显式指定设计限值的条件可能会称为非确定的因为限值的值是不确定的。

WS-BPEL 中的所有表达式，以及相关不透明表示法列举如下：

1. 布尔值表达式：
   - <source>的<transitionCondition>元素：

```
<transitionCondition expressionLanguage="anyURI"? opaque="yes"/>
```

   - <targets>的<joinCondition>元素：

```
<joinCondition expressionLanguage="anyURI"? opaque="yes"/>
```

   - <while>，<repeatUntil>，<if>和<elseif>的<condition>元素：

```
<condition expressionLanguage="anyURI"? opaque="yes"/>
```

2. 最终期限值表达式：
   - <onAlarm>和<wait>的<until>元素：

```
<until expressionLanguage="anyURI"? opaque="yes"/>
```

3. 持续时间值表达式：
   - <onAlarm>和<wait>的<for>元素：

```
<for expressionLanguage="anyURI"? opaque="yes"/>
```

   - <onAlarm>的<repeatEvery>元素：

```
<repeatEvery expressionLanguage="anyURI"? opaque="yes"/>
```

4. 无符号整数值表达式：
   - <forEach>的<startCounterValue>，<finalCounterValue>以及<branches>元素：

```
<startCounterValue expressionLanguage="anyURI"? opaque="yes"/>
```