SOAの中核技術としてのBPEL入門（3）

制御構造と各種のハンドラ

丸山不二夫 | （稚内北星学園大学）

BPELの制御構造

連載第6回目からWS-BPEL（Web Services Business Process Execution Language．以下BPEL）[1]について解説してきましたが，今回はその最後です．BPELは，WSDL（Web Services Description Language）によりサービスとして記述された個々のビジネスロジックを，ビジネスプロセス（業務プロセス）として統合するためのプログラミング言語です（**図-1**）．BPELの回の最後として，BPELの制御構造と各種のハンドラを見てみたいと思います．BPELは，XMLで書かれたプログラム言語ですので，シーケンシャルな実行，条件分岐や繰り返しといった処理が定義されているのは当然のことです．

ただ，XMLでプログラムを書くのは，とても骨が折れます．最初にお断りしたいのですが，実際には，GUIを使って，基本的なアクティビティのアイコンを配置し，ワイアリングでこれらを結び付けてゆくというBPELの開発環境を使うのが一般的です．GUI用のアイコン等の表記はBPMN（Business Process Modeling Notation）として標準化されており，BPMNをサポートする開発環境も増えてきました．開発環境上でビジネスフローを記述するとBPELコードに変換してくれます．今回，いくつかのBPELのコードを示しますが，それらを開発者が，そのままエディタで入力するわけではありません．

最初に，代表的なBPELの制御構造を見てみたいと思います．リスト1に，その一覧を挙げておきます．

リスト1　BPELの制御構造

```
sequence  :  シーケンシャルな実行
flow      :  並列処理
switch    :  条件分岐
if        :  条件分岐 (BPEL2.0)
while     :  繰り返し
wait      :  待機
pick      :  非決定的な動作（早い者勝ち）
```

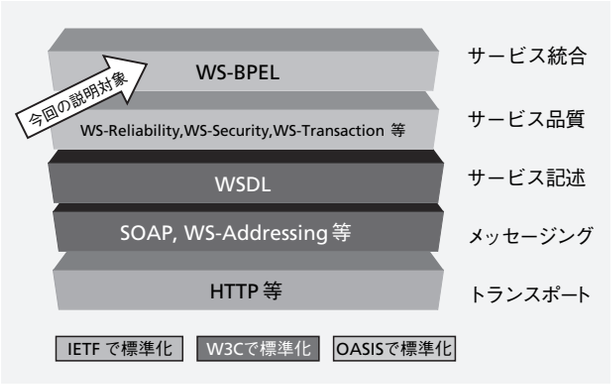


図-1　WS-BPELの位置付け

sequenceとflow（逐次実行と並列実行）

制御構造の基本は，プログラムの逐次実行です．BPELでは，そのために，<sequence>タグを明示的に指定します．XMLで書かれたプログラム言語であるBPELは，さまざまな命令を表すタグをアクティビティと呼んでいます．sequenceアクティビティは，その内部のトップエレメントのアクティビティを，上から下へと順番に実行せよという命令です．

sequenceアクティビティは，partnerLinksやvariablesといった，BPELの宣言部が終わって，そこからBPELの実行部分が始まるという目印の役割も果たしています．BPELプログラムの中で，最初に登場するsequenceアクティビティを見つければ，そこからBPELの実行が始まると考えていいのです．

BPELには，逐次実行の命令と並んで，他の言語ではあまり見られない並列実行の命令があります．flowアクティビティは，その直下にある複数のエレメントのアクティビティを，同時に並列に実行せよという命令です．

また，BPELには，いくつかのプログラムの部分を1つのスコープとしてまとめて，名前を付けるscopeアクティビティがあります．以下では，表現を簡単にするために，「アクティビティ」を「命令」と呼ぶことにします．たとえば，<scope>タグで表されるscopeアクティ

```
<scope name="GetLoanOffer"… . >
  <sequence>
    <assign>   </assign>
    <flow>  AとBが並列に実行される
      <sequence>
A       <invoke name="invokeUnitedLoan"… ./>
        <receive name="receive_invokeUnitedLoan"./>
      </sequence>
      <sequence>
B       <invoke name="invokeStarLoan"… ./>
        <receive name="receive_invokeStarLoan"… ./>
      </sequence>
    </flow>
  </sequence>
</scope>
```

図 -2　sequence と flow（並列実行）

ビティを，scope 命令と呼ぶことにします．

図 -2 を見てください．

冒頭の scope は，このコード全体が GetLoanOffer という名前を持つことを表しています．scope 内部は，1 個の sequence 命令からできていますので，このスコープに制御がわたると，assign 命令が，続いて，flow 命令が，シーケンシャルに実行されることが分かります．

flow 命令の実行とは，内部の 2 つの sequence 命令を並列に実行することです．図 -2 での，A と B が，flow 命令によって，同時に並列に実行されます．A は，sequence 命令からできていますので，A の実行とは，invoke 命令を呼び出して，その後に，receive 命令を実行することになりますし，同様に，B の実行も，sequence 命令の実行ですので，この例の場合では，invoke 命令を呼び出して，その後に，receive 命令を実行することになります．

flow 命令は，基本的には同時に実行された複数のプロセスがすべて終了するまで，ブロックして，待っています．

switchによる条件分岐

今度は，条件に応じて処理を分岐させる命令，switch を見てみましょう．switch 命令は，条件を指定して，その条件が満たされた場合の処理を記述する <case> 部と，条件が満たされなかった場合の処理を記述する <otherwise> 部の 2 つのパートからできています．

実際のサンプルを見てみましょう．まず，switch 命令の内部に，case エレメントと otherwise エレメントがあることを確認してください．case エレメントの condition 属性が，この switch 命令全体の条件を，文字列として指定しています．

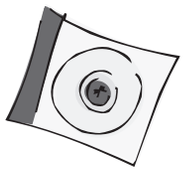
condition 属性で示される条件部では，getVariableData という BPEL の関数がよく利用されます．この関数は，変数名と part 名と XPath 指定の 3 つの引数を取って，変数のその指定された部分の値を返します．この例の場合には，input という名前の変数の，payload パートの，inputMsg エレメント内部の value エレメントの値を取り出しています．この値が，ゼロと，不等号の '>' で比較されています．

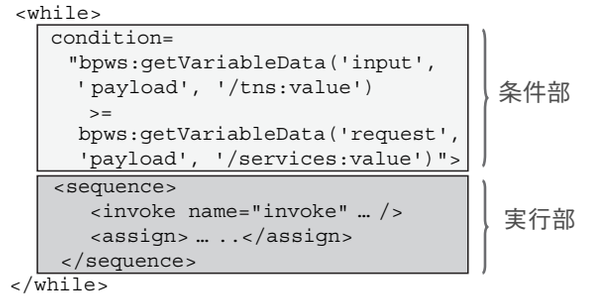
この条件が満たされた場合には，case 内部のトップエレメントが実行されます．ここでは，assign 命令が実行され，' 値は，ゼロより大きい ' という文字列が，output という名前の変数の，payload パートの，resultMsg エレメントの valueResult エレメントに，代入されます．

条件が満たされなかった場合には，<otherwise> タグに制御が移って，前と同じところに，' 値は，ゼロより小さいか等しい ' というメッセージが，assign 命令で実行されることになります．

リスト 2　switch 命令サンプル

```
    <switch>
      <case condition="bpws:getVariableData('input',
                      'payload',
                      '/tns:inputMsg/tns:value') > 0">
        <assign>
          <copy>
            <from expression="' 値は，ゼロより大きい '"/>
            <to variable="output" part="payload"
                query="/tns:resultMsg/tns:valueResult"/>
          </copy>
        </assign>
      </case>
      <otherwise>
        <assign>
          <copy>
            <from expression="' 値は，ゼロより小さいか等しい '"/>
            <to variable="output" part="payload"
                query="/tns:resultMsg/tns:valueResult"/>
          </copy>
        </assign>
      </otherwise>
    </switch>
```






```
<while>
   condition=
    "bpws:getVariableData('input',
     'payload', '/tns:value')
      >=
     bpws:getVariableData('request',
     'payload', '/services:value')">
   <sequence>
      <invoke name="invoke" … />
      <assign> … ..</assign>
   </sequence>
</while>
```

図 -3　while での繰り返し

BPEL2.0でのif命令

BPEL2.0 からは，switch 命令に代わって，次のような構文規則に従う if 命令がサポートされています．その動作は，容易に理解できると思います．

リスト 3　if 命令の構文規則

```
<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での繰り返し

今度は，BPEL での while 命令を使った繰り返しの定義を見ておきましょう．繰り返しの条件は，while 要素の condition 属性で指定されます．繰り返し実行される部分は，while 要素のトップエレメントです．

図 -3 を見てください．

まず，条件部分を見てください．先に見た getVariableData 関数が，2 度使われているのが分かると思います．この条件部分は，input 変数の value エレメントの値を，request 変数の value エレメントの値と '>=' で比較しています．おそらく，後者の値がインクリメントされていくのでしょう．前者の値が大きい限り，while 命令の，実行部分が繰り返し実行されます．

この実行部分は，sequence 命令です．ですので，条件が満たされている限り，次の命令実行が繰り返されることになります．すなわち，invoke 命令が呼ばれ，引き続いて assign 命令が，呼ばれることが繰り返されます．

wait until と wait for

BPEL には，プログラムの動作を一定時間止める wait 命令があります．wait 命令は，until 属性を取ると，いついつの時間まで止まるという指定になり，for 属性を取ると，指定された期間，動作を止めます．

次のリスト 4 を見てください．

最初の until の例は，具体的に指定した期日まで，動作が止まります．until の 2 つ目の例は，時間指定に getVariableData 関数を使っています．input 変数の waitValue エレメントの until エレメントで指定された値まで，動作が止まります．

for 属性は，先頭が PT である文字列で期間（period of time）を指定します．"PT1H2M3S" は，1 時間 2 分 3 秒の間という期間指定になります．for 属性についても，getVariableData 関数を使った指定が有効です．

リスト 4　wait until と wait for

```
<!-- 2007年3月9日15:50まで待つ -->
<wait until="2007-03-09T15:50:00"/>

<!-- ユーザが指定した時間になるまで待つ -->
<wait until="bpws:getVariableData('input', 'payload',
                    '/tns:waitValue/tns:until')"/>

<!-- 指定した期間（period of time）の間待つ この場合には，10秒間 -->
<wait for="PT10S"/>

<!-- ユーザが指定した間待つ -->
<wait for="bpws:getVariableData('input', 'payload',
                  '/tns:waitValue/tns:for')"/>
```

BPELの各種のハンドラ

BPEL には，何種類かの強力なハンドラが用意されています．これは，BPEL の 1 つの特徴だと思います．外部からのメッセージやイベントを処理する eventHandlers，エラーや例外の処理をする faultHandlers，ロールバックよりもう少し内容的に複雑な修復・補修の処理をする compensateHandlers など

です．

以下，代表的なハンドラとして，eventHandlers と faultHandlers を見ていきたいと思います．紙幅の都合で，BPEL の特徴的なハンドラである compensateHandlers については，詳細に解説することができませんがここで簡単に説明しておきます．compensateHandlers は“補償処理”を行うハンドラです．処理としてはデータベースのコミット・ロールバック処理のようなキャンセル処理を行うのですが，“一旦正常に完了した処理を前の状態に戻す”という点が異なります．コミット・ロールバック処理ではコミットするまで処理完了とせず状態を保持しているのに対し，補償処理では処理の状態は解放されているため少し複雑な戻し処理が必要になるのです．一方，補償処理により状態を保持するためのリソースを解放することが可能になるため，ロングトランザクションサポートの点でメリットがあります．

eventHandlers

eventHandlers のサンプルを見てみましょう．次のリストを見てください．まず，全体が scope 命令で囲まれていることを，確認してください．ついで，この scope 命令の内部が，eventHandlers 命令と sequence 命令という 2 つの部分から構成されていることを見てください．ここは，大事なポイントです．

この scope に制御が移ったとき，eventHandlers 命令内のイベント・ハンドラがイベントごとにセットされ，ついで sequence 命令が実行されます．この sequence 命令の実行中に，セットされたイベントが起きたとき，制御はこのイベントのハンドラに移ります．その後，イベント・ハンドラの処理の終了とともに，制御は scope から抜け出します．もしも，sequence 命令の実行中に何のイベントも起きなかった場合には，sequence 命令の終了とともに，制御は scope から抜け出します．

onMessage と onAlarm

eventHandlers には，任意の数だけイベント・ハンドラを登録することができます．この例では，eventHandlers 内に置かれているイベント・ハンドラは，onMessage 命令と onAlarm 命令の 2 つです．

onMessage 命令は，外部からの非同期メッセージをイベントとして，そのメッセージで呼び出されます．外部の非同期メッセージの指定には，invoke 命令や receive 命令と同じように，partnerLink と portType と operation の三つ組みが利用されます．受け取ったメッセージは，onMessage 要素の variable 属性で指定された変数に入れられます．

onAlarm 命令は，先に見た wait と同じようなシンタックスを持ち，一定期間の経過後（for 属性を利用）あるいは，特定の時刻（until 属性を利用）に，タイム・イベントを発生させます．

onMessage 命令も onAlarm 命令も，具体的な処理は，内部の sequence 命令に記述されます．

リスト 5　eventHandlers 命令サンプル

```
-------------------------------------------------------------------------
    <!--  receive the result of the remote process -->
    <scope name="handleEvents">

      <eventHandlers>

        <!-- wait for event callback -->
        <onMessage partnerLink="AsyncBPELService"
                   portType=
                   "services:AsyncBPELServiceCallback"
                   operation="onEvent" variable="event">
          <sequence>
            <assign>
              <copy>
                 <!-- 以下２行は実際には１行 -->
                 <from expression="'Est processing time
event received, est time remaining = 15 seconds'"/>
                 <to variable="notes"/>
               </copy>
            </assign>

            <wait for="'PT15S'"/>

          </sequence>
        </onMessage>

        <!-- test alarm -->
        <onAlarm for="'PT10S'">
          <sequence>
            <assign>
              <copy>
                 <!-- 以下２行は実際には１行 -->
                 <from expression="'Alarm timeout: no
response from AsyncBPELService after 15 seconds'"/>
                 <to variable="notes"/>
               </copy>
            </assign>

            <wait for="'PT5S'"/>

          </sequence>
        </onAlarm>
```

```
    </eventHandlers>

    <!-- wait for result callback -->
    <sequence>
      <receive name="receiveResponse"
          partnerLink="AsyncBPELService"
          portType="services:AsyncBPELServiceCallback"
          operation="onResult" variable="response"/>

      <assign>
        <copy>
          <from variable="response" part="payload"/>
          <to variable="output" part="payload"/>
        </copy>
      </assign>

    </sequence>
  </scope>
```

サンプルの動きを追う

もう少し詳しく，このリスト 5 を見て行きましょう．まず，この scope に制御が移ると，eventHandlers 内に定義されている 2 つのハンドラがセットされます．1 つは，onMessage ハンドラで，AsyncBPELService という partnerLink で，onEvent という operation が呼ばれるのを待っています．もう 1 つは，onAlarm ハンドラで，ハンドラがセットされてから 10 秒後に呼び出されます．

この scope の実行の本体は，eventHandlers 命令の後ろにある sequence 命令です．イベント・ハンドラがセットされると直ちに，この sequence 命令が実行されます．この例では，AsyncBPELService という partnerLink の onResult という operation の呼び出しを receive 命令で待っていて，メッセージが届くと，それを assign 命令で output という変数に代入する処理を行います．

この scope の実行の本体の処理中に，先にセットした 2 つのハンドラが呼び出されることがあります．

onMessage ハンドラが呼び出されるのは，AsyncBPELService という partnerLink の相手側が，onResult ではなく onEvent という operation を呼び出した場合です．このとき，ある文字列を notes という変数に代入して，15 秒待ちます．

onAlarm ハンドラが呼び出されるのは，scope の実行が始まってから 10 秒後です．このとき，ある文字列が notes 変数に代入されて，さらに 5 秒だけ待ちます．

BPEL2.0でのonEvent命令

BPEL2.0 からは，onMessage 命令に代わって，次の構文規則に従う，onEvent 命令が用いられます．本体の実行部分が scope 命令になっていることに注意してください．

リスト 6　onEvent 命令の構文規則

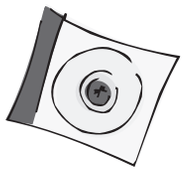
```
<onEvent partnerLink="NCName"
  portType="QName"?
  operation="NCName"
  ( messageType="QName" | element="QName" )?
  variable="BPELVariableName"?
  messageExchange="NCName"?>*
  ….
  <scope …>…</scope>
</onEvent>
```

faultHandlers

faultHandlers は，scope の実行部分で発生したエラーや例外に対応した処理を行うフォールト・ハンドラを定義します．次のリスト 7 を見てください．フォールト・ハンドラとして，catch 命令が用いられていることを確認してください．

この例では，1 つの catch 命令しかありませんが，faultHandlers には，複数個の catch 命令を置くことができます．これらの複数のフォールト・ハンドラの定義の中から，catch 命令の faultName 属性，faultVariable 属性の値にマッチするハンドラが起動されることになります．

この例で定義されているフォールト・ハンドラがどういう場合に呼び出されるかは，このコードを見ているだけでは，明示的には示されていません．実は，scope の実行部分である sequence の中の，invokeCR という名前の invoke 命令の実行中にフォールトが発生した場合に呼び出されるのですが，そのことを次に説明しようと思います．



リスト７　faultHandlers サンプル

```
<variables>
  <variable name="crError"
            messageType=
            "services:CreditRatingServiceFaultMessage"/>
  ……
</variables>

<scope name="GetCreditRating" ……>

   <faultHandlers>
      <catch faultName=
      "services:NegativeCredit" faultVariable="crError">
         <assign>
           <copy>
             <from expression="number(-1000)"/>
             <to variable="input" part="payload"
                query=
                "/auto:loanApplication/auto:creditRating"/>
           </copy>
         </assign>
      </catch>
   </faultHandlers>

   <sequence>
      ……
      ……
      <invoke name="invokeCR"
         partnerLink="creditRatingService"
         portType="services:CreditRatingService"
         operation="process"
         inputVariable="crInput"
         outputVariable="crOutput"/>
      ……
      ……
   </sequence>

</scope>
```

faultName と faultVariable

ここでは，invokeCR という名前の invoke 命令の実行が，フォールト・ハンドラを起動する場合を少し詳しく見ておきましょう．そのためには，この invoke 命令を定義している WSDL ファイルの参照が必要になります．関連部分をリスト 8 に抜き出しました．

これを見ると，invokeCR という名前の invoke 命令が，"NegativeCredit" という名前を持つフォールト・メッセージを返すことが分かります．これは，先の catch 命令の faultName 属性の値に一致しています．また，このフォールト・メッセージの型は，WSDL の message 節では，"tns:CreditRatingServiceFaultMessage" という名前で定義されているのですが，これは，先の catch 命令の faultVariable で指定された変数 crError の定義された型，"services:CreditRatingServiceFaultMessage" と一致しています．

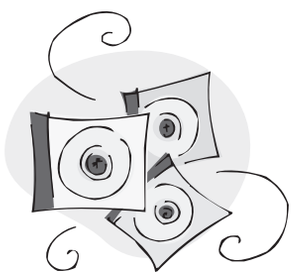
リスト８　CreditRatingService.wsdl

```
……
<message name="CreditRatingServiceFaultMessage">
  <part name="payload" element="tns:error" />
</message>

<portType name="CreditRatingService">
    <operation name="process">
        <input message=
        "tns:CreditRatingServiceRequestMessage"/>
        <output message=
        "tns:CreditRatingServiceResponseMessage"/>
        <fault name="NegativeCredit"
               message=
               "tns:CreditRatingServiceFaultMessage" />
    </operation>
</portType>
```

throw と catch

次の例のように，scope の実行部分で明示的に throw 命令を使って，フォールト・ハンドラの catch 命令に制御を移すことができます．throw 命令の faultName 属性で，faultName を設定できます．ここでは，error という名前に設定されていますね．これが，同じ faultName を持つ catch を呼び出します．throw と catch にあらわれる 2 つの faultVariable である，error と error2 も，variables での変数の定義を見れば，どちらも同じ型を持つことが確認できます．



リスト9　throw 命令と catch 命令

```
------------------------------------------------------------------------

<variables>
   ....
  <variable name="error"
            messageType="tns:InvalidFlowExceptionMessage"/>
  <variable name="error2"
            messageType="tns:InvalidFlowExceptionMessage"/>
</variables>
.....

<scope name="ft">
    <faultHandlers>
        <catch faultName="tns:error" faultVariable="error2">
            <empty/>
        </catch>
    </faultHandlers>

    <sequence>
         <receive name="receiveInput" ..... />
         <assign>
            <copy>
               <from expression=
               "string('some error')"/>
               <to variable="error" part="payload"/>
            </copy>
         </assign>
         <throw faultName="tns:error" faultVariable="error"/>
         <invoke name="replyOutput" ….>
    </sequence>
</scope>

------------------------------------------------------------------------
```

3回にわたって BPEL の解説をしてきましたが，それでも BPEL 仕様のほんの一部しか説明できませんでした．BPEL は XML をベースにしたプログラミング言語であるため少し難しかったかもしれませんが，制御構造は Java をはじめとするプログラミング言語と類似しています．今回説明した，flow, wait, pick アクティビティや各種ハンドラなど，Web サービス（通信）を前提とした言語であることによる特徴的なアクティビティを中心に理解すると，BPEL 仕様 [1] を効率的に理解することができます．

次回は連載を1回お休みし，第9回目からは，ビジネスで Web サービスを利用する際に必須となるサービス品質についての標準を，最新動向もふまえて解説していきます．第9回目は Web サービスのセキュリティ標準，WS-Security/WS-SX（Web Services Secure Exchange）について解説する予定です．

1）OASIS WS-BPEL TC：http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

（平成 19 年 3 月 11 日受付）

丸山不二夫（正会員）
maruyama@wakhok.ac.jp

東大教育学部卒業．一橋大学大学院社会学研究科博士課程修了．「最北端・最先端」をモットーに，稚内で新しいスタイルとコンテンツの情報教育を展開．「新しい時代の新しい大学」を目指して，社会人 IT 技術者をターゲットとしたサテライト校を秋葉原に設置．アジアでの IT 教育も熱心に展開している．現在，稚内北星学園大学学長．