WebSphere® Process Server for z/OS

バージョン 6.0.1

モジュールの開発とデプロイ

お願い

本書をご使用になる前に、 141 ページの『特記事項』に記載されている情報をお読みください。

本書は、WebSphere Process Server for z/OS バージョン 6、リリース 0、モディフィケーション 1 (製品番号 5655-N53) および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

`http://www.ibm.com/jp/manuals/main/mail.html`

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

`http://www.ibm.com/jp/manuals/` の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原　典： WebSphere® Process Server for z/OS
Developing and Deploying Modules
Version 6.0.1

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2006.6

この文書では、平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、平成角ゴシック体™W7

目次

モジュールの開発とデプロイ

モジュールの開発とデプロイは、基本的な作業です。

WebSphere Process Server 資料 (PDF 形式)

以下のトピックは、WebSphere® Process Server で使用するためのモジュールの開発と、そのモジュールのサーバーへのデプロイに関する概念とタスクについて説明したものです。

モジュールの開発の概要

モジュールは、WebSphere Process Server アプリケーションのデプロイメントの基本単位です。モジュールは、アプリケーションが使用する 1 つ以上のコンポーネント・ライブラリーとステージング・モジュールで構成されています。コンポーネントは、ほかのサービス・コンポーネントを参照することができます。モジュールを開発するには、アプリケーションが必要とするコンポーネント、ステージング・モジュール、およびライブラリー (モジュールによって参照される成果物の集合) が実動サーバー上で使用可能であることを確認する必要があります。

WebSphere Integration Developer は、WebSphere Process Server にデプロイするモジュールを開発するための主要なツールです。ほかの環境でモジュールを開発することもできますが、WebSphere Integration Developer を使用するのが最適な方法です。

6.0.1+ WebSphere Process Server は、2 つのタイプのサービス・モジュールをサポートします。ビジネス・サービス用モジュールおよびメディエーション・モジュールです。ビジネス・サービス用モジュールは、プロセスのロジックをインプリメントします。メディエーション・モジュールは、サービス起動をターゲットが理解する形式に変換し、要求をターゲットに渡して結果をオリジネーターに戻すことによって、アプリケーション間の通信を可能にします。

以降のセクションでは、WebSphere Process Server 上でモジュールをインプリメントおよび更新する方法について説明します。

コンポーネントの概要

コンポーネントは、再使用可能なビジネス・ロジックをカプセル化するための基本要素です。サービス・コンポーネントは、インターフェース、参照、インプリメンテーションに関連付けられます。インターフェースは、サービス・コンポーネントと呼び出し側コンポーネントの間の取り決めを定義します。サービス・モジュールは、WebSphere Process Server を使用して、他のモジュールが使用できるようにサービス・コンポーネントをエクスポートしたり、サービス・コンポーネントをインポートして使用したりすることができます。サービス・コンポーネントを呼び出すために、呼び出し側のモジュールはサービス・コンポーネントとのインターフェースを参照します。呼び出し側モジュールからそれぞれのインターフェースへの参照を構成することによって、インターフェースに対する参照が解決されます。

モジュールを開発するには、以下の作業を行う必要があります。

1. モジュール内のコンポーネント用のインターフェースを定義します。
2. サービス・コンポーネントで使用されるビジネス・オブジェクトを定義、変更、または操作します。
3. インターフェースを使用して、サービス・コンポーネントを定義または変更します。

   注: サービス・コンポーネントは、インターフェースを使用して定義されます。
4. 必要に応じて、サービス・コンポーネントをエクスポートまたはインポートします。
5. コンポーネントを使用するモジュールをインストールするために使用する EAR ファイルを作成します。WebSphere Integration Developer のエクスポート EAR 機能を使用してファイルを作成するか、serviceDeploy コマンドを使用して EAR ファイルを作成し、サービス・コンポーネントを使用するサービス・モジュールをインストールします。

開発タイプ

WebSphere Process Server では、サービス指向のプログラミング・パラダイムを促進するコンポーネント・プログラミング・モデルを提供します。このモデルを使用するために、提供者はサービス・コンポーネントのインターフェースをエクスポートします。これにより、利用者はそのインターフェースをインポートして、そのサービス・コンポーネントがローカルであるかのように使用できるようになります。開発者は厳密に型指定されたインターフェースまたは動的型付きインターフェースのいずれかを使用して、サービス・コンポーネントをインプリメントしたり、呼び出したりします。インターフェースとそのメソッドについては、このインフォメーション・センターの『References』のセクションに説明があります。

サービス・モジュールをサーバーにインストールした後、管理コンソールを使用して、アプリケーションからの参照のターゲット・コンポーネントを変更することができます。新しいターゲットは、アプリケーションからの参照が要求しているものと同じビジネス・オブジェクト・タイプを受け入れ、同じ操作を実行する必要があります。

サービス・コンポーネントの開発に関する考慮事項

サービス・コンポーネントを開発する場合は、以下の点を検討してください。

- このサービス・コンポーネントがエクスポートされ、ほかのモジュールによって使用されるかどうか。

  使用される場合、そのコンポーネントに定義したインターフェースを別のモジュールが使用できることを確認してください。
- サービス・コンポーネントを実行するのに比較的長い時間がかかるかどうか。

  長時間かかる場合は、サービス・コンポーネントに非同期のインターフェースをインプリメントすることを検討してください。
- サービス・コンポーネントを分散化することが有益かどうか。

有益である場合は、サーバーのクラスター上にデプロイされているサービス・モジュール内にサービス・コンポーネントのコピーを配置して、並列処理の利点を活かすことを検討してください。

- アプリケーションが、一相および二相コミット・リソースの混用を必要とするか。

  必要とする場合、アプリケーションの Last Participant サポートを使用可能にしてください。

  注: WebSphere Integration Developer を使用してアプリケーションを作成したか、または serviceDeploy コマンドを使用してインストール可能な EAR ファイルを作成した場合、これらのツールは自動的にアプリケーションのサポートを使用可能にします。WebSphere Application Server for z/OS インフォメーション・センターで、トピック『同一トランザクション内での 1 フェーズ・コミットおよび 2 フェーズ・コミットのリソースの使用』を参照してください。

サービス・モジュールの開発

サービス・コンポーネントは、サービス・モジュール内に含まれていなければなりません。サービス・コンポーネントを含むためのサービス・モジュールを開発することが、ほかのモジュールにサービスを提供するための鍵となります。

以下のタスクでは、要件を分析した結果、ほかのモジュールで使用できるように、サービス・コンポーネントをインプリメントすると有益であると判断されていることが前提となっています。

要件を分析した結果、サービス・コンポーネントの提供と利用が効率的な情報処理手段であると判断できる場合があります。ご使用の環境にとって再使用可能なサービス・コンポーネントが有効であると判断したうえで、サービス・コンポーネントを含むためのサービス・モジュールを作成してください。

1. ほかのサービス・モジュールで使用できるコンポーネントを特定します。

   サービス・コンポーネントを特定したら、『サービス・コンポーネントの開発』に進みます。
2. ほかのサービス・モジュール内のサービス・コンポーネントを使用できる、アプリケーション内のサービス・コンポーネントを特定します。

   サービス・コンポーネントとそれぞれのターゲット・コンポーネントを特定したら、『コンポーネントの呼び出し』に進みます。
3. クライアント・コンポーネントをワイヤー経由でターゲット・コンポーネントに接続します。

サービス・コンポーネントの開発

ご使用のサーバー内の複数のアプリケーションに再使用可能なロジックを提供するための、サービス・コンポーネントを作成します。

この作業では、複数のモジュールで使用できる処理がすでに作成され、特定されていることが前提になっています。

複数のモジュールで 1 つのサービス・コンポーネントを使用することができます。サービス・コンポーネントをエクスポートすると、インターフェースを介してそのコンポーネントを参照するほかのモジュールが、そのサービス・コンポーネントを利用できるようになります。この作業では、ほかのモジュールがコンポーネントを使用できるように、そのサービス・コンポーネントを作成する方法を説明します。

注: 1 つのサービス・コンポーネントに、複数のインターフェースを設定することができます。

1. 呼び出し元とサービス・コンポーネントの間のデータの移動のためのデータ・オブジェクトを定義します。

   データ・オブジェクトおよびそのタイプは、呼び出し元とサービス・コンポーネント間のインターフェースの一部となります。
2. 呼び出し元がサービス・コンポーネントを参照するときに使用するインターフェースを定義します。

   このインターフェース定義で、サービス・コンポーネントを指定し、サービス・コンポーネント内のすべての使用可能なメソッドをリストします。
3. インプリメンテーションを定義するクラスを開発します。
   - コンポーネントが長期にわたって実行される (非同期) 場合は、ステップ 4 に進みます。
   - コンポーネントが長期にわたって実行されるものでない (同期) 場合は、ステップ 5 に進みます。
4. 非同期インプリメンテーションを開発します。

   重要: 非同期型コンポーネント・インターフェースでは、joinsTransaction プロパティーを `true` に設定できません。

   a. 同期型サービス・コンポーネントを示すインターフェースを定義します。
   b. サービス・コンポーネントのインプリメンテーションを定義します。
   c. ステップ 6 に進みます。
5. 同期インプリメンテーションを開発します。
   a. 同期型サービス・コンポーネントを示すインターフェースを定義します。
   b. サービス・コンポーネントのインプリメンテーションを定義します。
6. コンポーネントのインターフェース、およびインプリメンテーションを拡張子が .java のファイルに保管します。
7. サービス・モジュールと必要なリソースを JAR ファイルにパッケージ化します。

   ステップ 7 から 9 までの詳しい説明については、このインフォメーション・センターの『実動サーバーへのモジュールのデプロイ』のセクションを参照してください。
8. serviceDeploy コマンドを実行して、アプリケーションを格納するインストール可能な EAR ファイルを作成します。
9. サーバー・ノード上にアプリケーションをインストールします。

10. **オプション:** ほかのサービス・モジュール内のサービス・コンポーネントを呼び出す場合は、呼び出し元とそれに対応するサービス・コンポーネント間のワイヤーを構成します。

    このインフォメーション・センターの『管理』セクションに、ワイヤーの構成についての説明があります。

コンポーネントの開発例

この例では、1 つのメソッド CustomerInfo をインプリメントする同期型サービス・コンポーネントを示しています。最初のセクションでは、getCustomerInfo というメソッドをインプリメントするサービス・コンポーネントに対するインターフェースを定義しています。

```
public interface CustomerInfo {
 public Customer getCustomerInfo(String customerID);
}
```

以下のコード・ブロックで、サービス・コンポーネントをインプリメントします。

```
public class CustomerInfoImpl implements CustomerInfo {
 public Customer getCustomerInfo(String customerID) {
  Customer cust = new Customer();

  cust.setCustNo(customerID);
  cust.setFirstName("Victor");
  cust.setLastName("Hugo");
  cust.setSymbol("IBM");
  cust.setNumShares(100);
  cust.setPostalCode(10589);
  cust.setErrorMsg("");

  return cust;
 }
}
```

この例では、非同期型サービス・コンポーネントを作成します。コードの最初のセクションでは、getQuote というメソッドをインプリメントするサービス・コンポーネントに対するインターフェースを定義しています。

```
public interface StockQuote {

 public float getQuote(String symbol);
}
```

以下のセクションは、StockQuote に関連したクラスのインプリメンテーションです。

```
public class StockQuoteImpl implements StockQuote {

 public float getQuote(String symbol) {


     return 100.0f;
 }
}
```

以下のコード・セクションは、非同期インターフェース StockQuoteAsync をインプリメントします。

```
public interface StockQuoteAsync {

 // deferred response
 public Ticket getQuoteAsync(String symbol);
 public float getQuoteResponse(Ticket ticket, long timeout);

 // callback
 public Ticket getQuoteAsync(String symbol, StockQuoteCallback callback);
}
```

以下のセクションは、onGetQuoteResponse メソッドを定義するインターフェース StockQuoteCallback です。

```
public interface StockQuoteCallback {

 public void onGetQuoteResponse(Ticket ticket, float quote);
}
```

サービスを起動します。

コンポーネントの呼び出し

モジュールを含むコンポーネントは、WebSphere Process Server クラスターの任意のノード上でコンポーネントを使用することができます。

コンポーネントを呼び出す前に、WebSphere Process Server に、コンポーネントを含むモジュールがインストールされていることを確認してください。

コンポーネントは、コンポーネントの名前を使用し、コンポーネントに適したデータ型を渡すことによって、WebSphere Process Server クラスター内で使用可能なすべてのサービス・コンポーネントを使用することができます。この環境内でコンポーネントを呼び出すには、必要なコンポーネントを見つけてから、そのコンポーネントへの参照を作成する操作が必要です。

注: モジュール内のコンポーネントは、同一のモジュール内のコンポーネントを呼び出すことができ、これはモジュール内呼び出しと呼ばれます。提供側コンポーネント内のインターフェースをエクスポートし、呼び出し側コンポーネント内でインターフェースをインポートすることによって、外部呼び出し (モジュール内呼び出し) をインプリメントしてください。

重要: 呼び出し側モジュールが稼動するサーバーと異なるサーバー上に存在するコンポーネントを呼び出す場合は、サーバーへの追加構成を実行する必要があります。必要な構成は、コンポーネントが非同期に呼び出されるか、同期して呼び出されるかによって異なります。この場合のアプリケーション・サーバーの構成方法は、関連タスクで説明されています。

1. 呼び出し側モジュールに必要なコンポーネントを判別します。

   コンポーネント内のインターフェースの名前と、そのインターフェースに必要なデータ型を書き留めます。
2. データ・オブジェクトを定義します。

   入力または戻りは Java™ クラスでかまいませんが、サービス・データ・オブジェクトが最適です。
3. コンポーネントを探します。

a. ServiceManager クラスを使用して、呼び出し側モジュールが使用できる参照を取得します。
b. locateService() メソッドを使用して、コンポーネントを探します。

   インターフェースは、コンポーネントに応じて、Web サービス記述言語 (WSDL) ポート・タイプまたは Java インターフェースのいずれかを使用することができます。

4. コンポーネントを同期式、または非同期に呼び出します。

   Java インターフェースを使用してコンポーネントを呼び出すことも、 invoke() メソッドを使用してコンポーネントを動的に呼び出すこともできます。

5. 戻り値を処理します。

   コンポーネントが例外を生成することがあるので、クライアントでは例外の処理が可能である必要があります。

コンポーネントの呼び出し例

次の例では、 ServiceManager クラスを作成します。

```
ServiceManager serviceManager = new ServiceManager();
```

以下の例は、 ServiceManager クラスを使用して、コンポーネントの参照を含んでいるファイルからコンポーネントのリストを取得します。

```
InputStream myReferences = new FileInputStream("MyReferences.references");
ServiceManager serviceManager = new ServiceManager(myReferences);
```

以下のコードは、StockQuote Java インターフェースをインプリメントするコンポーネントを探します。

```
StockQuote stockQuote = (StockQuote)serviceManager.locateService("stockQuote");
```

以下のコードは、Java または WSDL ポート・タイプ・インターフェースをインプリメントするコンポーネントを探します。呼び出し側モジュールは、Service インターフェースを使用して、コンポーネントと対話します。

ヒント: コンポーネントが Java インターフェースをインプリメントする場合は、コンポーネントをインターフェースまたは invoke() メソッドのいずれかを使用して呼び出すことができます。

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
```

次の例は、別のコンポーネントを呼び出すコード MyValue を示しています。

```
public class MyValueImpl implements MyValue {

 public float myValue throws MyValueException {

  ServiceManager serviceManager = new ServiceManager();

     // variables
         Customer customer = null;
         float quote = 0;
         float value = 0;

         // invoke
         CustomerInfo cInfo =
     (CustomerInfo)serviceManager.locateService("customerInfo");
```

```
        customer = cInfo.getCustomerInfo(customerID);

    if (customer.getErrorMsg().equals("")) {

        // invoke
      StockQuoteAsync sQuote =
     (StockQuoteAsync)serviceManager.locateService("stockQuote");
      Ticket ticket =  sQuote.getQuoteAsync(customer.getSymbol());
   // ... do something else ...
      quote =  sQuote.getQuoteResponse(ticket, Service.WAIT);

        // assign
         value = quote * customer.getNumShares();
    } else {

        // throw
        throw new MyValueException(customer.getErrorMsg());
    }
    // reply
        return value;
 }
}
```

呼び出し側モジュールの参照とコンポーネントのインターフェースの間のワイヤーを構成します。

コンポーネントの動的呼び出し:

Web サービス記述言語 (WSDL) ポート・タイプ・インターフェースを指定したコンポーネントをモジュールから呼び出す場合、モジュールは invoke() メソッドを使用して、そのコンポーネントを動的に呼び出す必要があります。

この操作では、呼び出し側コンポーネントがコンポーネントを動的に呼び出すことが前提となっています。

WSDL ポート・タイプ・インターフェースの場合は、呼び出し側コンポーネントは invoke() メソッドを使用して、コンポーネントを呼び出す必要があります。呼び出し側モジュールから、この方法で Java インターフェースを指定したコンポーネントも呼び出すことができます。

1. 必要なコンポーネントを含んでいるモジュールを判別します。
2. コンポーネントが必要とする配列を判別します。

   入力配列は、次の 3 つのタイプのいずれかです。
   - 大文字の Java プリミティブ型、またはこの型の配列
   - 通常の Java クラス、またはクラスの配列
   - サービス・データ・オブジェクト (SDO)
3. コンポーネントからの応答を収容する配列を定義します。

   応答配列は、入力配列と同じタイプでかまいません。
4. invoke() メソッドを使用して、必要なコンポーネントを呼び出し、配列オブジェクトをそのコンポーネントに渡します。
5. 結果を処理します。

コンポーネントの動的呼び出しの例

以下の例では、モジュールは invoke() メソッドを使用して、大文字の Java プリミティブ・データ型を使用するコンポーネントを呼び出します。

```
Service service = (Service)serviceManager.locateService("multiParamInf");

  Reference reference = service.getReference();

  OperationType methodMultiType =
    reference.getOperationType("methodWithMultiParameter");

  Type t = methodMultiType.getInputType();

  BOFactory boFactory = (BOFactory)serviceManager.locateService
    ("com/ibm/websphere/bo/BOFactory");

  DataObject paramObject = boFactory.createbyType(t);

  paramObject.set(0,"input1")
  paramObject.set(1,"input2")
  paramObject.set(2,"input3")

  service.invoke("methodMultiParamater",paramObject);
```

次の例では、WSDL ポート・タイプ・インターフェースをターゲットとして持つ呼び出しメソッドを使用します。

```
Service serviceOne = (Service)serviceManager.locateService("multiParamInfWSDL");

 DataObject dob = factory.create("http://MultiCallWSServerOne/bos", "SameBO");
   dob.setString("attribute1", stringArg);

 DataObject wrapBo = factory.createByElement
  ("http://MultiCallWSServerOne/wsdl/ServerOneInf", "methodOne");
   wrapBo.set("input1", dob); //wrapBo encapsulates all the parameters of methodOne
   wrapBo.set("input2", "XXXX");
   wrapBo.set("input3", "yyyy");

 DataObject resBo= (DataObject)serviceOne.invoke("methodOne", wrapBo);
```

異なるサーバー上のサービス起動時の考慮事項

サービス指向アーキテクチャーの利点の 1 つは、利用者が他のサービス・モジュールに既に存在するサービスを使用できることです。公平にワークロードのバランスを取るために、セル内の異なるサーバーにアプリケーションをインストールすることができ、それらのアプリケーションは異なる物理サーバー上に存在してもかまいません。

WebSphere Process Server の利点の 1 つは、アプリケーションのワークロードをセル内の複数のサーバーに分散させる機能です。このように分散することにより、セル内のさまざまなサーバー間のワークロード・バランシングを改善することができ、サーバー内にアプリケーションまたはサービスのコピーが 1 つしかないため、コンピューティング・リソースの保守容易性が最大になります。したがって、サーバー A 上のアプリケーションが、セル内のサーバー B にインストールされたサービスを必要とする場合もあります。そうした場合にサービスを使用するには、サーバー間の通信を構成する必要があります。実行する構成のタイプは、呼び出し側サービス・コンポーネントがサービスを非同期に起動するか、同期して起動するかによって異なります。

関連トピックで、非同期の起動および同期した起動の両方について、システムの構成方法を説明しています。

サービスを非同期に呼び出すサーバーの構成:

異なるサーバー上のサービス・コンポーネントが通信できるようにするには、サーバーを同じように構成する必要があります。このトピックでは、異なるサーバー上のサービスを非同期に起動するアプリケーションの通信を可能にするために実行する構成について説明します。

このタスクでは、通信を構成するシステム上に既に WebSphere Process Server をインストールしているが、関係するアプリケーションをまだインストールしていないことを前提としています。関係する両方のサーバーの構成を検査して変更することのできる管理コンソールを使用します。

別のシステム上にインストールされたサービス・コンポーネントのサービスを必要とするアプリケーションをインストールする前に、要求をやりとりできるようにシステムを構成する必要があります。非同期呼び出しを使用するサービス・モジュールの場合、処理には外部バスおよび Service Integration Bus (SIBus) のメディエーションが含まれます。

注: このタスクでは、呼び出し側サービス・モジュールはシステム A に存在し、ターゲットはシステム B に存在しています。

このタスクで、構成に使用する情報を 11 ページの図 1 に示しています。

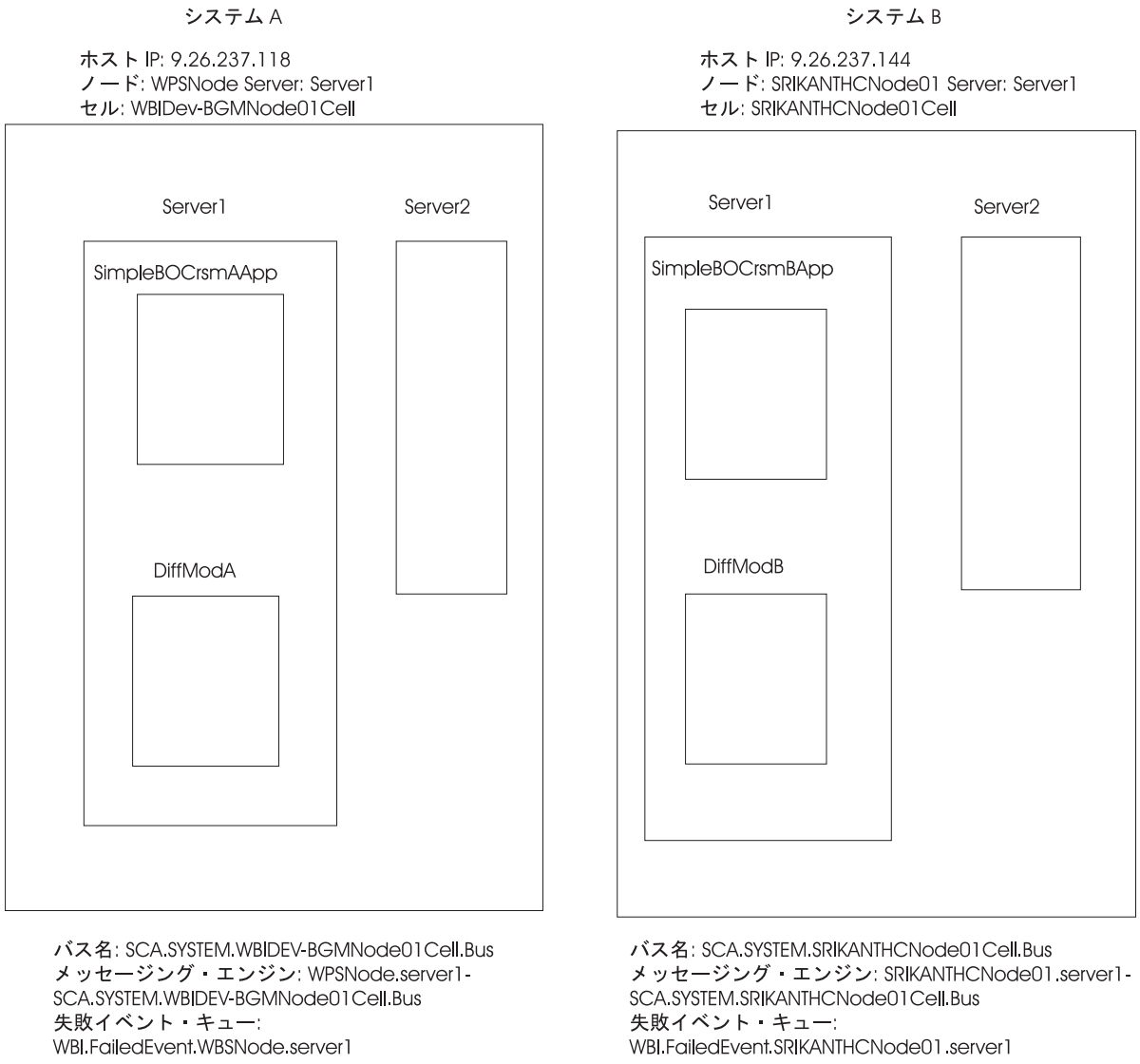


図 1. 異なるシステム上のサービスの呼び出し

注: 説明を簡潔にするため、各セル内にはこの通信に関係するサーバーのみが示され、各サーバーは異なる物理マシン上にあります。

1. 通信に関係する各サーバーに関する情報を収集します。 オリジネーターおよびターゲット・サーバーの両方について、以下の情報が必要になります。
   - ホスト IP アドレス
   - セル
   - ノード
   - サーバー
   - バス名
   - メッセージング・エンジン
   - 失敗イベント・キュー名
2. アプリケーションをインストールします。
3. それぞれのサーバーで、他方のサーバーを指す外部バスを作成し、ルーティング定義タイプを `Direct, service integration bus link` に設定します。

詳しくは、WebSphere Application Server Network Deployment バージョン 6 インフォメーション・センターの『外部バスの追加』を参照してください。

例では、システム A の外部バスは SCA.SYSTEM.SRIKANTHCNode01Cell.Bus になります。システム B の外部バスは SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus となります。

4. それぞれのサーバーで、他方のサーバー上のメッセージング・エンジンを指す SIB メディエーション・リンクをセットアップします。

   詳しくは、WebSphere Application Server Network Deployment バージョン 6 インフォメーション・センターの『サービス統合バス・リンクの追加』を参照してください。

   例では、システム A の SIB メディエーション・リンクは以下のようになります。

   ```
   SIB Link: TestCrossCell
   Remote ME: SRIKANTHCNode01.server1-SCA.SYSTEM.SRIKANTHCNode01Cell.Bus
   Bootstrap: 9.26.237.144:7277:BootstrapBasicMessaging
   ```

   システム B の SIB メディエーション・リンクは以下のようになります。

   ```
   SIB Link: TestCrossCell
   Remote ME: WPSNode.server1.SCA.SYSTEM.WBIDev-BGMNode01.Cell.Bus
   Bootstrap: 9.26.237.118:7276:BootstrapBasicMessaging
   ```

   重要: ブートストラップ内のポート番号は、SIB エンドポイント・アドレス・ポートです。セキュリティーを使用可能にした場合、セキュア SIB エンドポイント・アドレス・ポートを使用する必要があります。

5. サーバーを再始動して、SIB メディエーション・リンクを同期します。

   次のようなメッセージが表示されます。`[8/24/05 11:00:09:741 PDT] 00000086 SibMessage I [SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:WPSNode.server1-SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus] CWSIP0382I: messaging engine 2D7333574B0CD70B responded to subscription request, Publish Subscribe topology now consistent.`

6. 各サービス・モジュールの宛先を表示します。
7. 他のシステム上のターゲットに関連付ける必要のある呼び出し側サービス・モジュールの、出力宛先の転送パスを変更します。

   関連付ける宛先の宛先名には `importlink` が含まれます。例えばシステム A では、宛先は sca/SimpleBoCrsmA/importlink/test/sca/cros/simple/custinfo/CustomerInfo のようになります。外部バス名を宛先名にプレフィックス変換して、パスを変更します。例では、2 番目のシステムの外部バス名は SCA.SYSTEM.SRIKANTHCNode01Cell.Bus になります。結果は以下のようになります。

   ```
   SCA.SYSTEM.SRIKANTHCNode01Cell.Bus:sca/SimpleBoCrsmA/importlink/
   test/sca/cros/simple/custinfo/CustomerInfo
   ```

8. ターゲット・サーバー上に 2 つの宛先を作成し、もう一方のサーバー上の呼び出し側サービス・モジュールを指すように、それらを構成します。

   例では、システム B に以下を作成します。

```
sca/SimpleBOCrsmA/import/test/sca/cros/simple/custinfo/CustomerInfo
sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer
```

次に転送パスを、呼び出し側サーバー上の対応するパスを指すように設定します。 This would look like:

```
SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:
sca/SimpleBOCrsmA/import/test/sca/cros/simple/custinfo/CustomerInfo
SCA.SYSTEM.WBIDev-BGMNode01Cell.Bus:
sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer
```

a. 例外宛先を、作成した両方の宛先用の失敗イベント・キューに設定します。

   例では、値は `WBI.FailedEventSRIKANTHCNode01.server1` です。

9. **オプション:** システムでセキュリティーを使用可能にした場合、送信者のロールを外部バスに追加します。 オペレーティング・システムのコマンド・プロンプトから、両方のシステムで各アプリケーションが使用するユーザーを必ず定義してください。 ロールを追加するコマンドは以下のとおりです。

```
wsadmin $AdminTask addUserToForeignBusRole -bus busName
  -foreignBus foreignBusName -role roleName -user userName
```

各部の意味は、次のとおりです。

busName
: コマンドを入力するシステム上のバスの名前。

foreignBusName
: ユーザーを追加する外部バス。

userName
: 外部バスに追加するユーザー ID。

アプリケーションを始動します。

サービスを同期して呼び出すサーバーの構成:

サービス・コンポーネントが別のサービス・コンポーネントを同期して呼び出す場合、ターゲットを実行するシステムを指すように呼び出し側サービス・コンポーネントを構成して、ターゲット・サービスが呼び出し側サービス・コンポーネントに結果を通信できるようにする必要があります。

このタスクでは、通信を構成するシステム上に既に WebSphere Process Server をインストールしているが、関係するアプリケーションをまだインストールしていないことを前提としています。関係する両方のサーバーの構成を検査して変更することのできる管理コンソールを使用します。

別のサービスを同期して呼び出すサービス・コンポーネントは、ターゲット・システム上のエクスポート Java Naming and Directory Interface (JNDI) 名を、呼び出し側システム上の JNDI 名に構成するだけで、ターゲットと通信できます。

注: このタスクでは、呼び出し側サービス・モジュールはシステム A に存在し、ターゲットはシステム B に存在しています。

このタスクで、構成に使用する情報を 14 ページの図 2 に示しています。

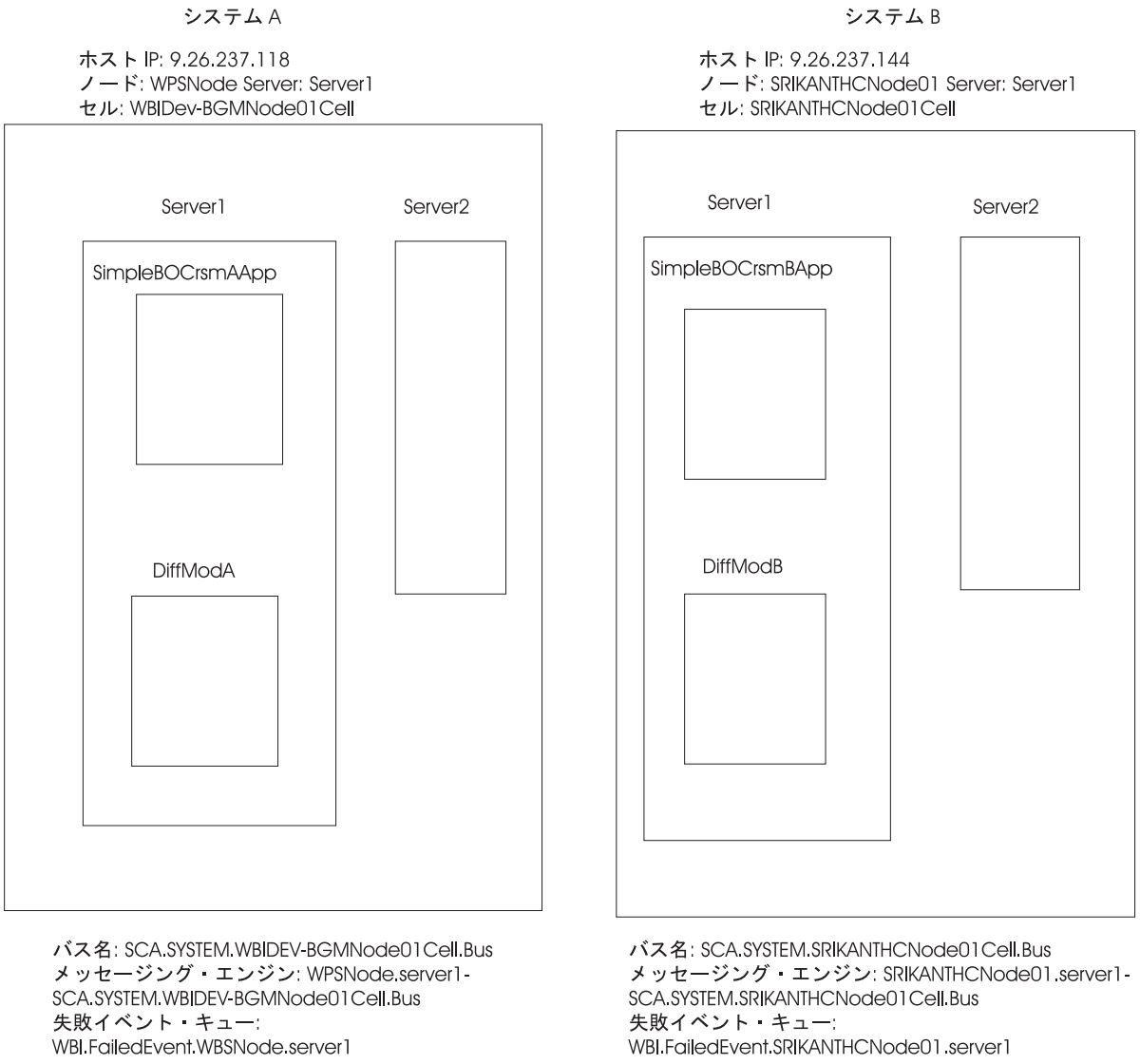


図 2. 異なるシステム上のサービスの呼び出し

注: 説明を簡潔にするため、各セル内にはこの通信に関係するサーバーのみが示され、各サーバーは異なる物理マシン上にあります。

1. それぞれのサーバーにアプリケーションをインストールします。
2. 呼び出し側システム (例ではシステム A) で、ターゲット・システム上のエクスポートを指す新規ネーム・スペース・バインディングを作成します。

   「**ネーム・スペース・バインディング (Name Space Bindings)**」 ペインで、セルのスコープを選択し、「**適用**」をクリックします。変更されたスコープで 、表示内の「**新規**」をクリックして、新規バインディングを作成します。

   ウィザード内で、以下を指定します (値は例の構成に適した値です)。

   a. バインディング・タイプは CORBA です。

   b. 基本プロパティーは以下のとおりです。

- 「バインディング ID (Binding identifier)」は固有のストリングです。この例では`sca_import_test_sca_cross_simple_custinfo_CustomerInfo` となっています。
- 「ネーム・スペース内の名前 (Name in Name space)」は、ターゲット・システム上で呼び出すエンタープライズ Java Bean (EJB) の JNDI 名です。例えば以下のようになります。

  ```
  sca/SimpleBOCrsmB/export/test/sca/cros/simple/custinfo/CustomerInfo
  ```

  これにより、ターゲット・システム上のエクスポート・インターフェースを指定します。
- 「Corbaname URL (Corbaname URL)」は、ターゲット・システム上のネーミング・サービスの IP アドレスおよびポート番号です。例えば、以下のようになります。

  ```
  corbaname:iiop:9.26.237.144:2809/NameServiceServerRoot#sca/
    impleBOCrsmB/export/test/sca/cros/simple/custinfo/CustomerInfo
  ```

  完了したら、「**次へ**」をクリックし、「**要約 (Summary)**」ページで値を検証します。検証後、「**終了**」をクリックします。

システムに、新規バインディングが表示されます。

3. 「**保管**」をクリックして変更を保管します。

アプリケーションを始動します。これで、システム A のサービス・コンポーネントは、システム B のサービスを同期して呼び出すことができるようになりました。

モジュールとターゲットの分離の概要

モジュールを開発する際、複数のモジュールが使用できるサービスを識別します。このようにしてサービスにてこ入れすることにより、開発サイクルとコストを最小化します。多数のモジュールによって使用されるサービスがある場合は、ターゲットがアップグレードされた場合に新規サービスへの切り替えが呼び出しモジュールに対して透過的になるように、呼び出しモジュールをターゲットから分離する必要があります。このトピックでは、単純な呼び出しモデルと分離された呼び出しモデルを対比して、分離がどのように役立つかを示す例を提供します。特定の例について説明しますが、これが、ターゲットからモジュールを分離する唯一の方法というわけではありません。

単純な呼び出しモデル

モジュールを開発する際、その他のモジュールにあるサービスを使用することができます。これは、モジュールにサービスをインポートしてからそのサービスを呼び出すことによって実行します。インポートされたサービスは、WebSphere Integration Developer で、または管理コンソール内のサービスをバインディングすることによって、その他のモジュールによってエクスポートされたサービスに「関連付け」られます。『単純な呼び出しモデル』は、このモデルを示しています。

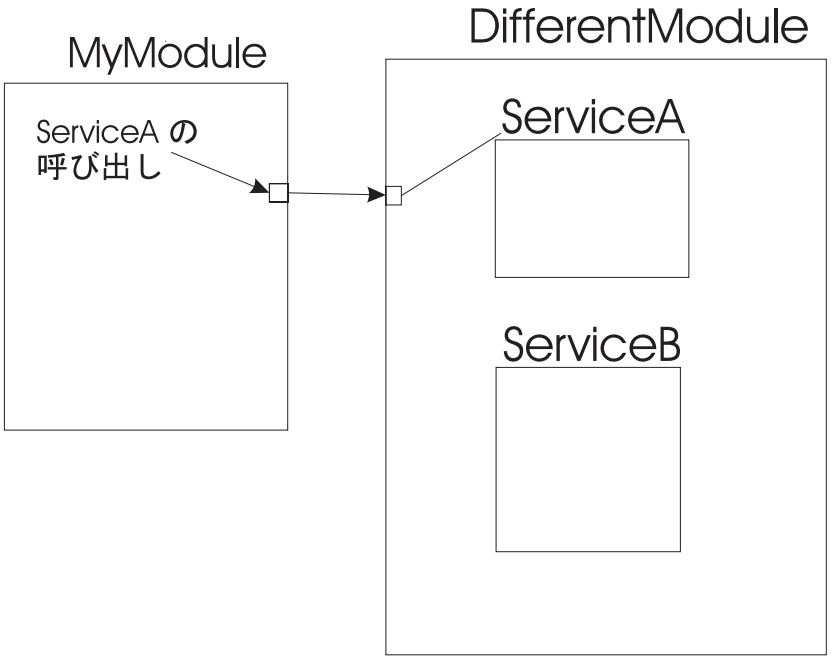


図 3. 単純な呼び出しモデル

分離された呼び出しモデル

呼び出しモジュールを停止せずに呼び出しのターゲットを変更するには、呼び出しのターゲットから呼び出しモジュールを分離します。この場合、モジュールそのものではなくダウンストリーム・ターゲットを変更しているので、ターゲットの変更中もモジュールが処理を続行できます。『アプリケーションの分離の例』は、分離によって、呼び出しモジュールの状況に影響を与えずにターゲットを変更する方法を示します。

アプリケーションの分離の例

単純な呼び出しモデルを使用した場合、同一のサービスを呼び出す複数のモジュールは、『単一のサービスを呼び出す複数のアプリケーション』のようになります。APPA、APPB、および APPC はすべて CalculateFinalCost を呼び出します。

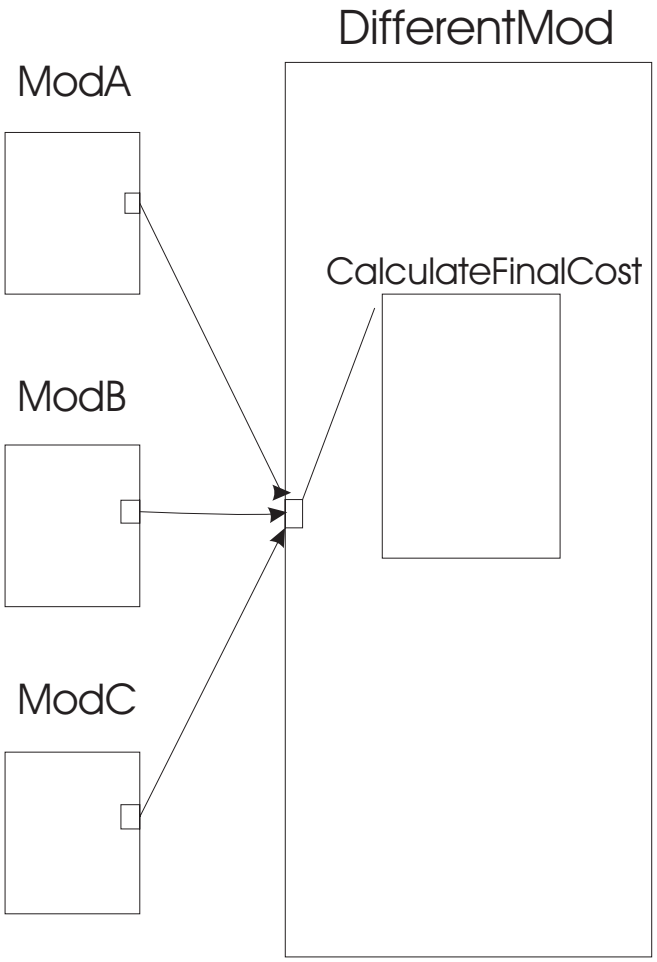


図 4. 単一のサービスを呼び出す複数のアプリケーション

CalculateFinalCost によって提供されるサービスは、そのサービスを使用するすべてのモジュールに新規コストが反映されるように、更新する必要があります。開発チームは、新規サービス UpdatedCalculateFinal を構築およびテストして、変更を取り込みます。新規サービスは実動に移す準備ができています。分離を使用しない場合は、UpdateCalculateFinal を呼び出すために、 CalculateFinalCost を呼び出すモジュールをすべて更新する必要があります。分離を使用した場合、実行する必要があるのは、バッファー・モジュールをターゲットに接続するバインディングを変更することだけです。

注: このようにサービスを変更することにより、オリジナルのサービスを、それを必要とするその他のモジュールに提供し続けることができます。

分離を使用して、アプリケーションとターゲットの間でバッファー・モジュールを作成します (『UpdateCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

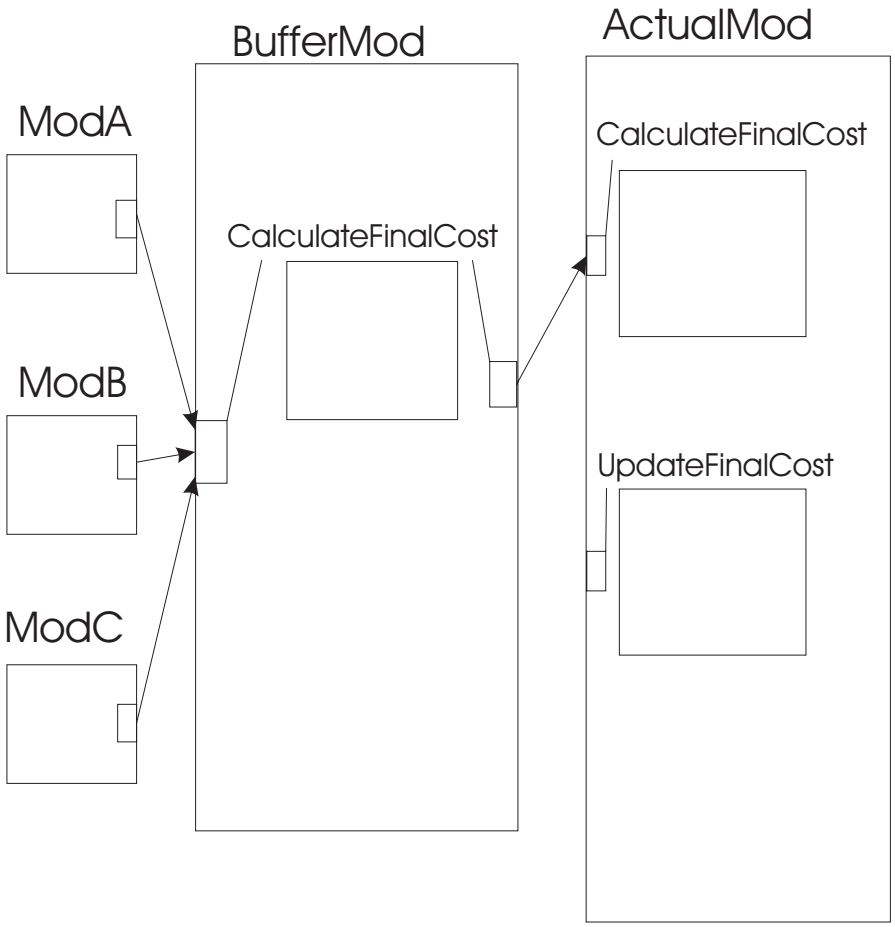


図 5. UpdateCalculateFinal を呼び出す分離された呼び出しモデル

このモデルで、呼び出しモジュールは変わりません。実行する必要があるのは、バインディングをバッファー・モジュール・インポートからターゲットへ変更することだけです (『UpdatedCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

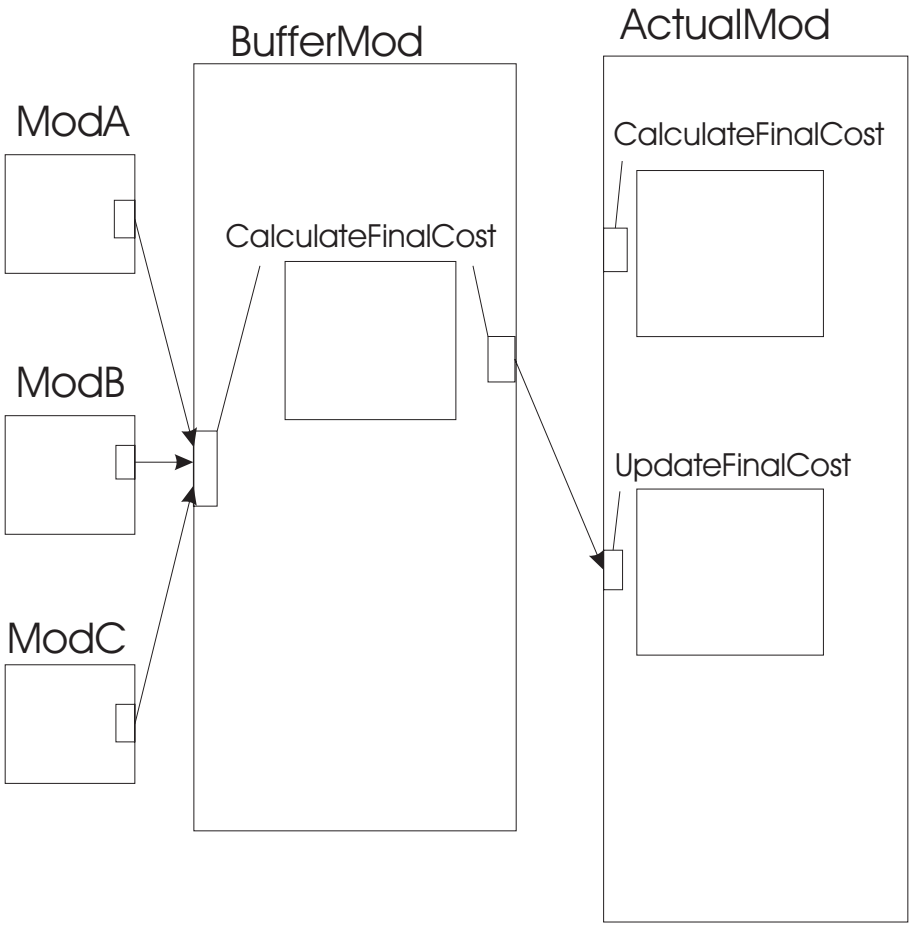


図 6. UpdatedCalculateFinal を呼び出す分離された呼び出しモデル

バッファー・モジュールがターゲットを同期で呼び出す場合、元のアプリケーションに戻される結果は、バッファー・モジュール (メディエーション・モジュール、またはビジネス・モジュール用のサービス) を再始動したときに、新規ターゲットから送信されます。バッファー・モジュールがターゲットを非同期で呼び出す場合、元のアプリケーションに戻される結果は、次の呼び出し時に新規ターゲットから送信されます。

ビジネス・プロセスおよびタスク用アプリケーションの開発

WebSphere Integration Developer などのモデル化ツールを使用して、ビジネス・プロセスやタスクを作成しデプロイすることができます。そのようなプロセスとタスクは、実行時に相互に働きかけます。例えば、プロセスを開始すると、タスクが要求されて完了し、実行されていたプロセスが終了します。プロセスおよびタスクとは、Business Process Choreographer Explorer を使用して対話できますが、Business Process Choreographer API を使用して、このような対話用にカスタマイズしたアプリケーションを開発することもできます。

API には、WebSphere Process Server にインストールされるすべてのプロセスやタスクで使用可能な汎用メソッドがあります。 Business Process Choreographer API は、次の 2 種類のステートレス・セッション・エンタープライズ Bean として提供されます。

- BusinessFlowManagerService インターフェースは、ビジネス・プロセス・アプリケーション用のメソッドを備えています。
- HumanTaskManagerService インターフェースは、タスク・ベースのアプリケーション用のメソッドを備えています。

Business Process Choreographer API の詳細は、com.ibm.bpe.api パッケージおよび com.ibm.task.api パッケージの中の Javadoc を参照してください。

1. アプリケーションが提供する機能を決定します。

   標準的なビジネス・プロセスおよびヒューマン・タスク機能の例が提供されています。
2. どちらの Business Choreographer API を使用するかを決定します。

   アプリケーションでインプリメントするシナリオに応じて、2 つのセッション Bean のうちの 1 つ、または両方を使用することができます。
3. アプリケーションのユーザーが必要とする許可権限を判別します。

   アプリケーションのユーザーは、アプリケーションに組み込まれたメソッドを呼び出すこと、およびこれらのメソッドが戻すオブジェクトとそのオブジェクトの属性を表示することを許可されていなければなりません。該当する Business Process Choreographer API セッション Bean のインスタンスを作成するときに、WebSphere Application Server がセッション・コンテキストとそのインスタンスを関連付けます。セッション・コンテキストには、呼び出し元のプリンシパルのロールが含まれます。この情報は、それぞれの呼び出しごとに、呼び出し元の権限を確認するために使用されます。

   Javadoc には、各メソッドの許可情報が含まれています。アプリケーションのユーザーにとって最適なメソッドを選択してください。
4. アプリケーションをレンダリングする方法を決めます。

   Business Process Choreographer API は、ローカル側でもリモート側でも呼び出すことができます。
5. アプリケーションを開発します。
   a. API にアクセスします。
   b. API を使用して、プロセスまたはタスクと対話します。
      - データを照会します。
      - データで作業を行います。

汎用 API へのアクセス

ビジネス・プロセス・アプリケーションおよびタスク・アプリケーションは、Bean のホーム・インターフェースを介して、適切なセッション Bean にアクセスします。

BusinessFlowManagerService インターフェースおよび HumanTaskManagerService インターフェースは、セッション Bean の共通のインターフェースです。これらのインターフェースは、アプリケーション・プログラムが呼び出すことができる機能を公開します。このアプリケーション・プログラムは、別の Enterprise JavaBeans™ (EJB) アプリケーションを含む任意の Java プログラムを指します。

リモート・セッション Bean またはローカル・セッション Bean のいずれかを使用して、汎用 API にアクセスできます。

リモート・セッション Bean へのアクセス

アプリケーションは、Bean のホーム・インターフェースを介して、適切なリモート・セッション Bean にアクセスします。

セッション Bean は、プロセス・アプリケーションに対しては BusinessFlowManager セッション Bean、タスク・アプリケーションに対しては HumanTaskManager セッション Bean のいずれかである可能性があります。

1. リモート・セッション Bean への参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、`application-client.xml` ファイル
   - Web アプリケーションの場合は、`web.xml` ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、`ejb-jar.xml` ファイル

   プロセス・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-ref>
    <ejb-ref-name>ejb/BusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
    <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
   </ejb-ref>
   ```

   タスク・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-ref>
    <ejb-ref-name>ejb/HumanTaskManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>com.ibm.task.api.HumanTaskManagerHome</home>
    <remote>com.ibm.task.api.HumanTaskManager</remote>
   </ejb-ref>
   ```

   WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。
2. 生成されたスタブをアプリケーションにパッケージします。

   ご使用のアプリケーションが、BPEContainer アプリケーションまたは TaskContainer アプリケーションを実行しているのと異なる Java 仮想マシン (JVM) で実行されている場合、以下のアクションを完了します。

a. プロセス・アプリケーションの場合、
<*install_root*>/ProcessChoreographer/client/bpe137650.jar ファイルを、ご使用のアプリケーションのエンタープライズ・アーカイブ (EAR) ファイルにパッケージします。

b. タスク・アプリケーションの場合、
<*install_root*>/ProcessChoreographer/client/task137650.jar ファイルを、ご使用のアプリケーションの EAR ファイルにパッケージします。

c. アプリケーション・モジュールのマニフェスト・ファイル内の **Class-Path** パラメーターを、JAR ファイルを含めるように設定します。 アプリケーション・モジュールは、J2EE アプリケーション、Web アプリケーション、または EJB アプリケーションの可能性があります。

3. Java Naming and Directory Interface (JNDI) 検索機構を使用して、セッション Bean のホーム・インターフェースがアプリケーションから使用可能になるようにします。 以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the remote home interface of the BusinessFlowManager bean
  Object result =
         initialContext.lookup("java:comp/env/ejb/BusinessFlowManagerHome");

// Convert the lookup result to the proper type
   BusinessFlowManagerHome processHome =
           (BusinessFlowManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,BusinessFlowManagerHome.class);
```

セッション Bean のホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。このメソッドは、セッション Bean のリモート・インターフェースを戻します。

4. セッション Bean のリモート・インターフェースにアクセスします。 以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
BusinessFlowManager process = processHome.create();
```

5. サービス・インターフェースによって公開されたビジネス関数を呼び出します。以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
process.initiate("MyProcessModel",input);
```

アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
   UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

   // Begin a transaction
   transaction.begin();
```

```
          // Applications calls ...

        // On successful return, commit the transaction
        transaction.commit();
```

以下に、ステップ 3 から 5 でタスク・アプリケーションを探す方法の例を示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the remote home interface of the HumanTaskManager bean
  Object result =
         initialContext.lookup("java:comp/env/ejb/HumanTaskManagerHome");

// Convert the lookup result to the proper type
   HumanTaskManagerHome taskHome =
           (HumanTaskManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,HumanTaskManagerHome.class);

...
//Access the remote interface of the session bean.
HumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

ローカル・セッション Bean へのアクセス

アプリケーションは、Bean のホーム・インターフェースを介して、適切なローカル・セッション Bean にアクセスします。

セッション Bean は、プロセス・アプリケーションの場合は LocalBusinessFlowManager セッション Bean、ヒューマン・タスク・アプリケーションの場合は LocalHumanTaskManager セッション Bean になります。

1. ローカル・セッション Bean への参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、application-client.xml ファイル
   - Web アプリケーションの場合は、web.xml ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、ejb-jar.xml ファイル

   プロセス・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalBusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
    <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
   </ejb-local-ref>
   ```

   タスク・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

```
<ejb-local-ref>
 <ejb-ref-name>ejb/LocalHumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
 <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。

2. Java Naming and Directory Interface (JNDI) 検索機構を使用して、ローカル・セッション Bean のローカル・ホーム・インターフェースがアプリケーションから使用可能になるようにします。 以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the local home interface of the LocalBusinessFlowManager bean

   LocalBusinessFlowManagerHome processHome =
        (LocalBusinessFlowManagerHome)initialContext.lookup
        ("java:comp/env/ejb/LocalBusinessFlowManagerHome");
```

ローカル・セッション Bean のホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。このメソッドは、セッション Bean のローカル・インターフェースを戻します。

3. ローカル・セッション Bean のローカル・インターフェースにアクセスします。以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
LocalBusinessFlowManager process = processHome.create();
```

4. サービス・インターフェースによって公開されたビジネス関数を呼び出します。以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
process.initiate("MyProcessModel",input);
```

アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
   UserTransaction transaction=
       (UserTransaction)initialContext.lookup("jta/usertransaction");

   // Begin a transaction
   transaction.begin();

     // Applications calls ...

   // On successful return, commit the transaction
   transaction.commit();
```

以下に、ステップ 2 から 4 でタスク・アプリケーションを探す方法の例を示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

  // Lookup the local home interface of the LocalHumanTaskManager bean

   LocalHumanTaskManagerHome taskHome =
        (LocalHumanTaskManagerHome)initialContext.lookup
        ("java:comp/env/ejb/LocalHumanTaskManagerHome");

...
//Access the local interface of the local session bean
LocalHumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

ビジネス・プロセス用のアプリケーションの開発

ビジネス・プロセスは、ビジネス・ゴールを達成するために特定のシーケンスで呼び出される、ビジネス関連の一連のアクティビティーです。ビジネス・プロセスは、microflow または長期にわたって実行するプロセスのいずれかです。

- microflow は、短期で実行するビジネス・プロセスです。 microflow は、入力パラメーターによって呼び出され、このプロセスの同期実行中は呼び出し元は待機することになります。結果は即時に呼び出し元に戻されます。
- 長期実行の割り込み可能プロセスは、まとめてチェーニングされるアクティビティーのシーケンスとして実行されます。プロセスの並列分岐を、同期式にナビゲートすることができます。アクティビティーのタイプとトランザクションの設定に応じて、アクティビティーを独自のトランザクションで実行することができます。

microflow や長期実行プロセスに対する以下の標準のアクションに対応したアプリケーションを開発する方法を示した例が提供されています。

ビジネス・プロセスの許可のロール

ビジネス・プロセスで実行できるアクションは、許可のロールによって異なります。このロールは、J2EE ロールまたはインスタンス・ベースのロールです。

ロールは、同一レベルの権限を共有する従業員のグループです。 Java 2 Platform Enterprise Edition (J2EE) ロールは、ビジネス・プロセス・コンテナーが構成された場合にセットアップされます。インスタンス・ベースのロールは、プロセスがモデル化された場合にプロセスおよびアクティビティーに割り当てられます。ロール・ベースの許可は、 WebSphere Application Server でグローバル・セキュリティーが使用可能になっている必要があります。

J2EE ロール

次の J2EE ロールがサポートされています。

- J2EE BPESystemAdministrator。このロールに割り当てられたユーザーは、全特権を持ちます。
- J2EE BPESystemMonitor。このロールに割り当てられたユーザーは、すべてのビジネス・プロセス・オブジェクトのプロパティーを表示できます。

管理コンソールを使用して、これらのロールへのユーザーおよびグループの割り当てを変更することができます。

RACF セキュリティーによるロールの設定: この RACF 許可は、以下のセキュリティー・フィールドを指定した場合に適用されます。

- **com.ibm.security.SAF.authorization= true**

```
RDEFINE EJBROLE BPESystemAdministrator  UACC(NONE)
PERMIT BPESystemAdministrator CLASS(EJBROLE)  ID(userid) ACCESS(READ)

RDEFINE EJBROLE BPESystemMonitor  UACC(NONE)
PERMIT BPESystemMonitor  CLASS(EJBROLE)  ID(userid) ACCESS(READ)
```

- **com.ibm.security.SAF.delegation= true**

```
RDEFINE EJBROLE JMSAPIUser  UACC(NONE) APPLDATA(' userid')
```

Security Authorization Facility (SAF) ベースの許可 (RACF EJBROLE プロファイルの使用など) を使用して、ビジネス・プロセス・コンテナーなど、EJB およびエンタープライズ・アプリケーションにおける Java 2 Platform、Enterprise Edition (J2EE) ロールへのクライアントのアクセスを制御できます。SAF の使用についての詳細は、WebSphere Application Server for z/OS インフォメーション・センターの『役割ベースの許可の System Authorization Facility』を参照してください。

インスタンス・ベースのロール

プロセス・インスタンスまたはアクティビティーは、プロセス・モデル内でスタッフ・メンバーに直接割り当てられるのではなく、使用可能なロールのいずれかに割り当てられます。インスタンス・ベースのロールに割り当てられたスタッフ・メンバーは、そのロールのアクションを実行できます。インスタンス・ベースのロールに対するユーザーの関連は、実行時にスタッフ解決を使用して判別されます。

次のインスタンス・ベースのロールがサポートされています。
- プロセスの場合: リーダー、スターター、管理者
- アクティビティーの場合: リーダー、エディター、潜在的な所有者、所有者、管理者

これらのロールは、次のアクションを実行することが許可されています。

ロール	許可されたアクション
アクティビティー・リーダー	関連するアクティビティー・インスタンスのプロパティー、およびその入出力メッセージを表示します。
アクティビティー編集者	アクティビティー・リーダーに権限が与えられたアクション、およびアクティビティーに関連するメッセージやその他のデータに対する書き込みアクセス権限。
潜在的なアクティビティー所有者	アクティビティー・リーダーに対して許可されたアクション。このロールのメンバーは、アクティビティーを要求し、 receive アクティビティーまたは pick アクティビティーにメッセージを送信することができます。
アクティビティー所有者	アクティビティーを処理し、完了します。このロールのメンバーは、所有する作業項目を管理者または潜在的な所有者に転送することができます。

ロール	許可されたアクション
アクティビティー管理者	予期しないエラーによって停止されたアクティビティーの修復、および長期にわたるアクティビティーの強制終了を実行します。
プロセス・スターター	関連するプロセス・インスタンスのプロパティー、およびその入出力メッセージを表示します。
プロセス・リーダー	関連するプロセス・インスタンスのプロパティー、その入出力メッセージ、およびインスタンスに含まれているすべてのアクティビティー (サブプロセスのものは除く) に対してアクティビティー・リーダーがサポートするものすべてを表示します。
プロセス管理者	このロールのメンバーは、プロセス・インスタンスの管理、開始されたプロセスへの介入、および作業項目の作成、削除、転送を実行できます。このロールのメンバーには、アクティビティー管理者権限があります。

プロセス・インスタンスが存在している間は、ユーザー・レジストリーからプロセス・スターターのユーザー ID を削除しないでください。これを行うと、このプロセスのナビゲーションを継続できなくなります。システム・ログ・ファイルに、次のような例外が書き込まれます。

```
no unique ID for: <user ID>
```

プロセス・インスタンスに対するアクションに必要なロール:

LocalBusinessFlowManager または BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がプロセスに対するすべてのアクションを実行できることは保証しません。呼び出し元は、そのアクションの実行も許可されている必要があります。次の表に、それぞれのロールで実行できるプロセス・インスタンス上のアクションを示します。

アクション	呼び出し元のプリンシパルのロール		
	リーダー	スターター	管理者
createMessage	x	x	x
createWorkItem			x
delete			x
deleteWorkItem			x
forceTerminate			x
getActiveHandlers	x	x	x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x		x
getCustomProperties	x	x	x
getCustomProperty	x	x	x
getCustomPropertyNames	x	x	x
getFaultMessage	x	x	x
getInputClientUISettings	x		x
getInputMessage	x	x	x

アクション	呼び出し元のプリンシパルのロール		
	リーダー	スターター	管理者
getOutputClientUISettings	x		x
getOutputMessage	x	x	x
getProcessInstance	x	x	x
getVariable	x	x	x
getWaitingActivities	x	x	x
getWorkItems	x		x
resume			x
restart			x
setCustomProperty		x	x
setVariable			x
suspend			x
transferWorkItem			x

ビジネス・プロセス・アクティビティーのアクションに必要なロール:

LocalBusinessFlowManager または BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がアクティビティーに対するすべてのアクションを実行できることは保証しません。呼び出し元は、そのアクションの実行も許可されている必要があります。次の表に、それぞれのロールで実行できるアクティビティー・インスタンス上のアクションを示します。

アクション	呼び出し元のプリンシパルのロール				
	リーダー	エディター	潜在的な所有者	所有者	管理者
cancelClaim				x	x
claim			x		x
complete				x	x
createMessage	x	x	x	x	x
createWorkItem					x
deleteWorkItem					x
forceComplete					x
forceRetry					x
getActivityInstance	x	x	x	x	x
getAllWorkItems	x				x
getClientUISettings	x	x	x	x	x
getCustomProperties	x	x	x	x	x
getCustomProperty	x	x	x	x	x
getCustomPropertyNames	x	x	x	x	x
getFaultMessage	x	x	x	x	x
getFaultNames	x	x	x	x	x
getInputMessage	x	x	x	x	x
getOutputMessage	x	x	x	x	x

アクション	呼び出し元のプリンシパルのロール				
	リーダー	エディター	潜在的な所有者	所有者	管理者
getVariable	x	x	x	x	x
getWorkItems	x	x	x	x	x
setCustomProperty		x		x	x
setFaultMessage		x		x	x
setOutputMessage		x		x	x
setVariable					x
transferWorkItem				x 潜在的な所有者または管理者に対してのみ	x

ビジネス・プロセスの開始

ビジネス・プロセスを開始する方法は、プロセスが microflow であるか長期実行プロセスであるかによって異なります。プロセスを開始するサービスも、プロセスの開始方法にとって重要です。プロセスに固有の開始サービスを 1 つ設定するか、複数の開始サービスを設定することができます。

microflow や長期実行プロセスの標準の開始シナリオに対応したアプリケーションを開発する方法を示した例が提供されています。

固有の開始サービスを含む microflow の実行:

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。開始サービスが固有であるのは、microflow が receive アクティビティーを使って開始された場合、または pick アクティビティー内に 1 つの onMessage 定義のみがある場合です。

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しでパラメーターとしてプロセス・テンプレート名を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

1. **オプション:** プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
   ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
    "PROCESS_TEMPLATE.NAME",
     new Integer(50),
     null);
   ```

結果は名前でソートされます。call メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
                           (template.getID(),
                            template.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}

//run the process
ClientObjectWrapper output = process.call(template.getName(), input);
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

非固有の開始サービスを含む microflow の実行:

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。 microflow が複数の onMessage 定義を含む pick アクティビティーを使用して開始された場合、開始サービスは固有ではありません。

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しで開始サービスの ID を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

1. **オプション:** プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
("PROCESS_TEMPLATE.EXECUTION_MODE =
       PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
 "PROCESS_TEMPLATE.NAME",
  new Integer(50),
  null);
```

結果は名前でソートされます。長期実行プロセスとして開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

   この例では、最初に検出されたテンプレートを使用します。

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
        process.getStartActivities(template.getID());
```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input =
      process.createMessage(activity.getServiceTemplateID(),
                            activity.getActivityTemplateID(),
                            activity.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//run the process
ClientObjectWrapper output = process.call(activity.getServiceTemplateID(),
                                          activity.getActivityTemplateID(),
                                          input);
//check the output of the process, for example, an order number
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

   このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

固有の開始サービスを含む長期実行プロセスの開始:

開始サービスが固有の場合、initiate メソッドを使用して、プロセス・テンプレート名をパラメーターとして渡すことができます。これは、長期実行プロセスが、単一の receive アクティビティーまたは pick アクティビティーのいずれかを使用して開始する、および単一の pick アクティビティーが 1 つのみの onMessage 定義を持つ場合にあてはまります。

1. **オプション:** プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
  ("PROCESS_TEMPLATE.EXECUTION_MODE =
       PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
   "PROCESS_TEMPLATE.NAME",
    new Integer(50),
    null);
```

結果は名前でソートされます。initiate メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

```
ProcessTemplateData template = processTemplates[0];
//create a message for the single starting receive activity
ClientObjectWrapper input = process.createMessage
                            (template.getID(),
                            template.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.initiate(template.getName(), "CustomerOrder", input);
```

   このアクションによって、インスタンス CustomerOrder が作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

   プロセス・インスタンスのスターターは、要求の呼び出し元に設定されます。このユーザーは、このプロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、およびエディターが決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または staff、receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

非固有の開始サービスを含む長期実行プロセスの開始:

長期実行プロセスは、複数の開始 receive アクティビティーまたは pick アクティビティーを介して開始することができます。initiate メソッドを使用して、プロセスを開始することができます。例えば、プロセスが複数の receive または pick アクティビティー、または複数の onMessage 定義を持つ pick アクティビティーから開始される場合など、開始サービスが固有のものではない場合、呼び出されるサービスを識別する必要があります。

1. **オプション:** プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

```
ProcessTemplateData[] processTemplates = process.queryProcessTemplates
  ("PROCESS_TEMPLATE.EXECUTION_MODE =
       PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
   "PROCESS_TEMPLATE.NAME",
    new Integer(50),
    null);
```

結果は名前でソートされます。長期実行プロセスとして開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

```
ProcessTemplateData template = processTemplates[0];
ActivityServiceTemplateData[] startActivities =
       process.getStartActivities(template.getID());
```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

```
ActivityServiceTemplateData activity = startActivities[0];
//create a message for the service to be called
ClientObjectWrapper input = process.createMessage
                           (activity.getServiceTemplateID(),
                            activity.getActivityTemplateID(),
                            activity.getInputMessageTypeName());
DataObject myMessage = null;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.initiate(activity.getServiceTemplateID(),
                            activity.getActivityTemplateID(),
                            null,
                            input);
```

このアクションによって、インスタンスが作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

プロセス・インスタンスの開始は、要求の呼び出し元に設定され、プロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、およびエディターが決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または staff、receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

スタッフ・アクティビティーの処理

ビジネス・プロセス内のスタッフ・アクティビティーは、作業項目を通じて、組織内のさまざまな人に割り当てられます。プロセスが開始されると、潜在的な所有者

に対して作業項目が作成されます。これらの所有者のいずれかが、アクティビティーを要求します。このユーザーは、関係のある情報の提供とアクティビティーの完了に対して責任があります。

1. 作業の準備ができている、ログオン・ユーザーに属するアクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
                    ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
                    WORK_ITEM.REASON =
                         WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                    null, null, null);
```

   このアクションは、ログオン・ユーザーが作業することができるアクティビティーが含まれる照会結果セットを戻します。

2. 作業対象のアクティビティーを要求します。

```
if (result.size() > 0)
{
 result.first();
 AIID aiid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   アクティビティーが要求されると、アクティビティーの入力メッセージが戻されます。

3. アクティビティーの作業が終了したら、アクティビティーを完了します。 アクティビティーは、正常に完了することも、障害メッセージが表示されて完了することもあります。アクティビティーが正常に完了した場合、出力メッセージが渡されます。アクティビティーが失敗した場合、アクティビティーは失敗状態または停止状態に置かれ、障害メッセージが渡されます。これらのアクションに対して、適切なメッセージを作成する必要があります。メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   a. アクティビティーを正常に完了するには、出力メッセージを作成します。

```
ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
      process.createMessage(aiid, activity.getOutputMessageTypeName());
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the activity
process.complete(aiid, output);
```

      このアクションは、オーダー番号が含まれる出力メッセージを設定します。

b. 障害が発生した場合にアクティビティーを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the staff activity
List faultNames = process.getFaultNames(aiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      process.createMessage(aiid, faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

process.complete(aiid, (String)faultNames.get(0), myFault);
```

このアクションは、アクティビティーを失敗状態または停止状態のいずれかに設定します。プロセス・モデル内のアクティビティーの **continueOnError** パラメーターが真に設定されている場合、アクティビティーは失敗状態に置かれ、ナビゲーションが続行されます。 **continueOnError** パラメーターが偽に設定されている場合、アクティビティーは停止状態に置かれます。この状態では、強制終了または強制再試行を使用してアクティビティーを修復できます。

待機中のアクティビティーへのメッセージの送信

pick アクティビティー (receive choice アクティビティーとも呼ばれる) および receive アクティビティーを使用すると、実行中のプロセスを「外の世界」からのイベントと同期化することができます。例えば、情報に対する要求に応えたお客様からの E メールの受信は、このようなイベントとみなされます。

1. ログオンしたユーザーからのメッセージを待機しているアクティビティー・サービス・テンプレートをリストします。

```
QueryResultSet result =
   process.query("ACTIVITY_SERVICE.VTID,ACTIVITY.ATID",
                 "ACTIVITY.STATE=ACTIVITY.STATE.STATE_WAITING AND
                  ACTIVITY_SERVICE.PORT_TYPE='Confectionery' AND
                  ACTIVITY_SERVICE.OPERATION='OrderRequest' AND
                  WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                  null, null, null);
```

2. メッセージを送信します。

呼び出し元は、メッセージを受信するアクティビティーの潜在的な所有者、またはプロセス・インスタンスの管理者である必要があります。

```
if ( result.size() > 0 )
{
  result.first();
  VTID vtid = (VTID)result.getOID(1);
  ATID atid = (ATID)result.getOID(2);
  ActivityServiceTemplateData activity =
          process.getActivityServiceTemplate(vtid,atid);

  // create a message for the service to be called
  ClientObjectWrapper message =
        process.createMessage(vtid,atid,activity.getInputMessageTypeName());
```

```
  DataObject myMessage = null;
  if ( message.getObject()!= null && message.getObject() instanceof DataObject )
  {
    myMessage = (DataObject)message.getObject();
    //set the strings in the message, for example, chocolate is to be ordered
    myMessage.setString("Order", "chocolate");
  }

  // send the message to the waiting activity
  process.sendMessage(vtid, atid, message);
}
```

このアクションによって、指定されたメッセージを待機アクティビティー・サービスに送信し、一部のオーダー・データを渡します。

また、プロセス・インスタンス ID を指定して、メッセージが指定されたプロセス・インスタンスに送信されたことを確認することもできます。プロセス・インスタンス ID が指定されていない場合、メッセージは、アクティビティー・サービス、およびメッセージの相関値によって識別されたプロセス・インスタンスに送信されます。プロセス・インスタンス ID が指定された場合、相関値を使用して検出されたプロセス・インスタンスがチェックされ、指定されたプロセス・インスタンス ID であることが確認されます。

イベントの処理

ビジネス・プロセス全体とビジネス・プロセスの各スコープを、関連するイベントの発生時に呼び出されるイベント・ハンドラーと関連付けることができます。 プロセスによりイベント・ハンドラーを使用して、Web サービス操作を提供できるという点で、イベント・ハンドラーは、receive アクティビティーや pick アクティビティーと似ています。イベント・ハンドラーは、対応するスコープが実行中である限り、何度でも呼び出すことができます。また、イベント・ハンドラーの複数インスタンスを並行して活動化することができます。

以下のコードの断片では、あるプロセス・インスタンス用のアクティブなイベント・ハンドラーを取得する方法、および入力メッセージを送信する方法を示しています。

1. プロセス・インスタンス ID のデータを判別し、そのプロセスのアクティブなイベント・ハンドラーをリストします。

```
ProcessInstanceData processInstance =
       process.getProcessInstance( "CustomerOrder2711");
EventHandlerTemplateData[] events = process.getActiveEventHandlers(
                                    processInstance.getID() );
```

2. 入力メッセージを送信します。

   この例では、最初に検出されたイベント・ハンドラーを使用します。

```
EventHandlerTemplateData event = null;
if ( events.length > 0 )
{
    event = events[0];

    // create a message for the service to be called
    ClientObjectWrapper input = process.createMessage(
    event.getID(), event.getInputMessageTypeName());

    if (input.getObject() != null && input.getObject() instanceof DataObject )
    {
```

```
        DataObject inputMessage = (DataObject)input.getObject();
        // set content of the message, for example, a customer name, order number
        inputMessage.setString("CustomerName", "Smith");
        inputMessage.setString("OrderNo", "2711");

         // send the message
        process.sendMessage( event.getProcessTemplateName(),
                             event.getPortTypeNamespace(),
                             event.getPortTypeName(),
                             event.getOperationName(),
      input );
      }
  }
```

このアクションにより、指定されたメッセージがプロセスのアクティブなイベント・ハンドラーに送信されます。

プロセスの結果の分析

長期実行プロセスは、非同期に実行されます。出力メッセージは、プロセスの完了時に、自動的に戻されることはありません。メッセージを明示的に検索する必要があります。プロセスの結果は、プロセス・インスタンスが派生したプロセス・テンプレートが、派生したプロセス・インスタンスの自動削除を指定しない場合にのみ、データベースに保管されます。

プロセスの結果を分析し、例えば、オーダー番号などを確認します。

```
QueryResultSet result = process.query
                       ("PROCESS_INSTANCE.PIID",
                        "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                         PROCESS_INSTANCE.STATE =
                                 PROCESS_INSTANCE.STATE.STATE_FINISHED",
                         null, null, null);
if (result.size() > 0)
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ClientObjectWrapper output = process.getOutputMessage(piid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject )
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

ビジネス・プロセスのライフ・サイクルの管理

プロセスを開始できる Business Process Choreographer API メソッドが呼び出されると、プロセス・インスタンスが生成されます。プロセス・インスタンスのすべてのアクティビティーが終了状態になるまで、プロセス・インスタンスのナビゲーションは続きます。有効な終了状態は、完了、スキップ、失敗、期限切れ、または終了の状態です。

プロセス・インスタンス、またはそのアクティビティーのうちの 1 つで、プロセス・ロジックの一部として処理できない障害が起きることがあります。この場合、プロセス管理者は、いくつかの方法でそのアクティビティーやプロセス・インスタンスに対する処置をとることができます。

プロセスに対する以下の標準のライフ・サイクル・アクションに対応したアプリケーションを開発する方法を示した例が提供されています。

ビジネス・プロセスの中断と再開:

プロセス・インスタンスを一時的に中断して、再度それを再開し完了することができます。

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。プロセス・インスタンスを中断するには、プロセス・インスタンスが実行状態または失敗状態でなければなりません。

長期にわたって実行するトップレベルのプロセス・インスタンスを、そのプロセス・インスタンスが実行中の間に中断することができます。例えば、プロセスで後で使用されるバックエンド・システムへのアクセスを構成するために、プロセス・インスタンスを中断することがあります。プロセスの前提条件を満たしていれば、そのプロセス・インスタンスを再開することができます。

1. 中断する実行中のプロセス CustomerOrder を取得します。

   ```
   ProcessInstanceData processInstance =
                       process.getProcessInstance("CustomerOrder");
   ```

2. プロセス・インスタンスを中断します。

   ```
   PIID piid = processInstance.getID();
   process.suspend( piid );
   ```

   このアクションにより、指定したトップレベルのプロセス・インスタンスが中断します。プロセス・インスタンスは、中断状態になります。 autonomy 属性が child に設定されたサブプロセスは、実行中、失敗、終了中、または補正中の状態であれば、中断されます。

3. プロセス・インスタンスを再開します。

   ```
   process.resume( piid );
   ```

   このアクションにより、プロセス・インスタンスとそのサブプロセスが中断前の状態に戻ります。

ビジネス・プロセスの再開:

完了、終了、失敗、補正のいずれかの状態にあるプロセス・インスタンスを再開させることができます。

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。

プロセス・インスタンスの再開は、プロセス・インスタンスを初めて開始する手順と同様です。ただし、プロセス・インスタンスの再開時には、プロセス・インスタンス ID が認識されているため、インスタンスの入力メッセージが使用可能です。

プロセスに、プロセス・インスタンスを作成可能な複数の receive アクティビティーまたは pick アクティビティー (receive choice アクティビティーとも呼ばれる) が含まれる場合、これらのアクティビティーに属するすべてのメッセージを使用して、プロセス・インスタンスを再始動します。これらのアクティビティーのいずれ

かが、要求/応答操作をインプリメントする場合、関連する reply アクティビティーがナビゲートされると、応答が再度送信されます。

1. 再開させるプロセスを取得します。

```
ProcessInstanceData processInstance =
                    process.getProcessInstance("CustomerOrder");
```

2. プロセス・インスタンスを再開します。

```
PIID piid = processInstance.getID();
process.restart( piid );
```

このアクションにより、指定されたプロセス・インスタンスが再開されます。

プロセス・インスタンスの終了:

プロセス管理者権限を持つユーザーが、リカバリー不能状態として認識されているトップレベルのプロセス・インスタンスを終了する必要がある場合があります。例えば、アプリケーションが起動されて障害が発生したときに、休止状態に戻らない場合などです。

プロセス・インスタンスは、未解決のサブプロセスやアクティビティーがあってもこれらを待たずに即時に終了するため、プロセス・インスタンスの終了は例外的な場合にのみ行ってください。

1. 終了するプロセス・インスタンスを検索します。

```
ProcessInstanceData processInstance =
       process.getProcessInstance("CustomerOrder");
```

2. プロセス・インスタンスを終了します。

プロセス・インスタンスを終了する場合、補正を使用してプロセス・インスタンスを終了することも、補正を使用せずに終了することもできます。

補正を使用してプロセス・インスタンスを終了するには、以下のようにします。

```
PIID piid = processInstance.getID();
process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
```

補正を使用しないでプロセス・インスタンスを終了するには、以下のようにします。

```
PIID piid = processInstance.getID();
process.forceTerminate(piid);
```

プロセス・インスタンスを補正を使用して終了する場合、プロセス・テンプレートに定義された補正ハンドラーが呼び出されます。プロセス・テンプレートに補正ハンドラーが定義されていない場合は、デフォルトの補正ハンドラーが呼び出されます。補正を使用せずにプロセス・インスタンスを終了する場合、プロセス・インスタンスはアクティビティーが正常に終了するのを待たずに、即時に終了されます。

プロセスによって開始されるアプリケーションは、強制終了要求の影響を受けません。そのようなアプリケーションを終了させる場合は、プロセスによって開始されるアプリケーションを明示的に終了するステートメントをプロセス・アプリケーションに追加する必要があります。

プロセス・インスタンスの削除:

完了済みのプロセス・インスタンスは、プロセス・モデル内のプロセス・テンプレートに対応するプロパティーが設定されていれば、Business Process Choreographer データベースから自動的に削除されます。

監査ログに書き込まれていないプロセス・インスタンスのデータを照会する場合や、プロセスの削除をオフピーク時に遅らせる場合などに、プロセス・インスタンスをデータベースに保存しておくことができます。ただし、不要になったプロセス・インスタンス・データがディスク・スペースやパフォーマンスに影響を与える可能性があります。したがって、プロセス・インスタンス・データを定期的に削除する必要があります。プロセス・インスタンスを削除するには、プロセス管理者権限が必要であり、そのプロセス・インスタンスは、トップレベルのプロセス・インスタンスでなければなりません。

以下の例では、完了したプロセス・インスタンスをすべて削除する方法が示されています。

1. 完了したプロセス・インスタンスをリストします。

   ```
   QueryResultSet result =
        process.query("DISTINCT PROCESS_INSTANCE.PIID",
                      "PROCESS_INSTANCE.STATE =
                               PROCESS_INSTANCE.STATE.STATE_FINISHED",
                       null, null, null);
   ```

   このアクションは、完了したプロセス・インスタンスをリストした照会結果セットを戻します。

2. 完了したプロセス・インスタンスを削除します。

   ```
   while (result.next() )
   {
      PIID piid = (PIID) result.getOID(1);
      process.delete(piid);
   }
   ```

   このアクションは、選択されたプロセス・インスタンスをデータベースから削除します。

アクティビティーの修復

長期実行プロセスには、やはり長期間実行されるアクティビティーが含まれる場合があります。これらのアクティビティーでは、catch されていないエラーが発生して、停止状態になる可能性があります。実行状態のアクティビティーが、反応していないように見える可能性もあります。どちらの場合でも、プロセスのナビゲーションを継続できるように、アクティビティーを修復することができます。

Business Process Choreographer API は、アクティビティーの修復のために、forceRetry メソッドおよび forceComplete メソッドを提供しています。アクティビティーの修復アクションをアプリケーションに追加する方法を示した例が提供されています。

アクティビティーの強制完了:

長期実行プロセスのアクティビティーで、障害が発生することがあります。これらの障害が、囲んでいるスコープ内で障害ハンドラーによって catch されておらず、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーが停止するように指定している場合、アクティビティーは修復することができるように停止状態になります。この状態で、アクティビティーの完了を強制することができます。

例えば、アクティビティーが応答しない場合、実行状態のアクティビティーを強制的に完了することもできます。

特定のタイプのアクティビティーでは、追加要件が存在します。

staff アクティビティー
送信されるはずだったメッセージ、または引き起こされるはすだった障害など、強制完了呼び出しでパラメーターを渡すことができます。

script アクティビティー
強制完了呼び出しで、パラメーターを渡すことはできません。ただし、修復する必要がある変数を設定する必要があります。

invoke アクティビティー
invoke アクティビティーが実行状態の場合、サブプロセスでない非同期サービスを呼び出す invoke アクティビティーを強制的に完了することもできます。例えば、非同期サービスが呼び出されて応答がない場合、こうすることがあります。

1. 停止状態の停止アクティビティーをリストします。

```
QueryResultSet result =
     process.query("DISTINCT ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    PROCESS_INSTANCE.NAME='CustomerOrder'",
                    null, null, null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止したスタッフ・アクティビティーなどのアクティビティーを完了します。

   この例では、出力メッセージが渡されます。

```
if (result.size() > 0)
{
   result.first();
   AIID aiid = (AIID) result.getOID(1);
   ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper output =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   boolean continueOnError = true;
   process.forceComplete(aiid, output, continueOnError);
}
```

このアクションによって、アクティビティーが完了します。エラーが発生した場合、**continueOnError** パラメーターによって、forceComplete 要求の処理中にエラーが発生した場合に実行するアクションが決まります。

例では、**continueOnError** が true です。この値は、forceComplete 要求の処理中にエラーが発生した場合、アクティビティーは失敗状態になることを意味します。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、最終的に失敗状態になります。

停止されたアクティビティーの再試行:

長期実行プロセスのアクティビティーに、囲んでいるスコープ内で catch されていない障害が発生した場合、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーの停止を指定しているときは、アクティビティーは停止状態になり、修復することができます。アクティビティーの実行を再試行することができます。

アクティビティーが使用する変数を設定することができます。また、script アクティビティーの例外と共に、アクティビティーが予想したメッセージなど、強制再試行呼び出しのパラメーターを渡すこともできます。

1. 停止アクティビティーをリストします。

```
QueryResultSet result =
      process.query("DISTINCT ACTIVITY.AIID",
                    "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                     PROCESS_INSTANCE.NAME='CustomerOrder'",
                     null, null, null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止したスタッフ・アクティビティーなどのアクティビティーの実行を再試行します。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper input =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
    {
      myMessage = (DataObject)input.getObject();
      //set the strings in your message, for example, chocolate is to be ordered
      myMessage.setString("OrderNo", "chocolate");
    }

   boolean continueOnError = true;
   process.forceRetry(aiid, input, continueOnError);
}
```

   このアクションによって、アクティビティーが再試行されます。エラーが発生した場合、**continueOnError** パラメーターによって、forceRetry 要求の処理中にエラーが発生した場合に実行するアクションが決まります。

例では、**continueOnError** が true です。この場合、forceRetry 要求の処理中にエラーが発生すると、アクティビティーは失敗状態になります。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、最終的に失敗状態になります。

BusinessFlowManagerService インターフェース

BusinessFlowManagerService インターフェースは、クライアント・アプリケーションから呼び出すことができるビジネス・プロセス機能を公開します。

BusinessFlowManagerService インターフェースから呼び出すことができるメソッドは、プロセスまたはアクティビティーの状態、およびそのメソッドが含まれているアプリケーションを使用するユーザーの権限によって異なります。ビジネス・プロセス・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび BusinessFlowManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.bpe.api パッケージ内の Javadoc を参照してください。

プロセス・テンプレート

プロセス・テンプレートは、バージョン付けされ、デプロイされ、インストールされるプロセス・モデルで、ビジネス・プロセスの仕様を含んでいます。これは、例えば initiate() などの適切な要求を発行することによって、インスタンス化および開始することができます。プロセス・インスタンスの実行は、サーバーによって自動的に駆動されます。

表 1. プロセス・テンプレート用の API メソッド

メソッド	説明
getProcessTemplate	指定されたプロセス・テンプレートを取得します。
queryProcessTemplate	データベースに保管されているプロセス・テンプレートを取得します。

プロセス・インスタンス

以下の API メソッドは、プロセス・インスタンスを開始します。

表 2. プロセス・インスタンスを開始するための API メソッド

メソッド	説明
call	マイクロフローを作成および実行します。
callWithReplyContext	指定されたプロセス・テンプレートから、固有の開始サービスでのマイクロフローまたは固有の開始サービスでの長期実行プロセスを作成および実行します。呼び出しは、結果を非同期で待ちます。
callWithUISettings	マイクロフローを作成および実行し、出力メッセージとクライアント・ユーザー・インターフェース (UI) の設定を戻します。

表 2. プロセス・インスタンスを開始するための API メソッド (続き)

メソッド	説明
initiate	プロセス・インスタンスを作成し、そのプロセス・インスタンスの処理を開始します。このメソッドは、長時間実行プロセスに使用します。このメソッドは、応答不要送信を適用するマイクロフローに対しても使用できます。
sendMessage	指定されたメッセージを、指定されたアクティビティー・サービスおよびプロセス・インスタンスに送信します。プロセス・インスタンスは、マイクロフローまたは長期実行プロセスのいずれかです。これらのプロセスは、固有または非固有の開始サービスです。
getStartActivities	指定されたプロセス・テンプレートからプロセス・インスタンスを開始できるアクティビティーに関する情報を戻します。
getActivityServiceTemplate	指定されたアクティビティー・サービス・テンプレートを取得します。

表 3. プロセス・インスタンスのライフ・サイクルを制御するための API メソッド

メソッド	説明
suspend	実行状態または失敗状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を中断します。
resume	中断状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を再開します。
restart	完了、失敗、または終了状態にある、長期実行中のトップレベルのプロセス・インスタンスを再始動します。
forceTerminate	指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセス、およびその実行中のアクティビティー、要求済みのアクティビティー、または待機中のアクティビティーを終了します。
delete	指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセスを削除します。
query	データベースから、検索基準に一致するプロパティーを取得します。

アクティビティー

invoke アクティビティーの場合、プロセス・モデルで、それらのアクティビティーがエラー状態でも続行されるように指定できます。 continue-on-error フラグを false に設定し、未処理エラーが発生すると、そのアクティビティーは停止状態になります。その場合は、プロセス管理者が、そのアクティビティーを修復することができ

ます。continue-on-error フラグおよびそれに関連する修復機能は、例えば、invoke アクティビティーが失敗することがある長期実行プロセスなどで使用することができますが、補正および障害処理のモデル化には、かなりの労力が必要です。

アクティビティーの操作および修復には、以下のメソッドが使用可能です。

表 4. アクティビティー・インスタンスのライフ・サイクルを制御するための API メソッド

メソッド	説明
claim	準備ができたアクティビティー・インスタンスを要求し、ユーザーがそのアクティビティーを使用できるようにします。
cancelClaim	アクティビティー・インスタンスの要求を取り消します。
complete	アクティビティー・インスタンスを完了します。
forceComplete	実行状態または停止状態にあるアクティビティー・インスタンスを強制的に完了します。
forceRetry	実行状態または停止状態にあるアクティビティー・インスタンスを強制的に反復します。
query	データベースから、検索基準に一致するプロパティーを取得します。

変数およびカスタム・プロパティー

このインターフェースは、変数の値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスに関連付けたり、指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。

表 5. 変数およびカスタム・プロパティーの API メソッド

メソッド	説明
getVariable	指定された変数を取得します。
setVariable	指定された変数を設定します。
getCustomProperty	指定されたアクティビティーまたはプロセス・インスタンスの指定されたカスタム・プロパティーを取得します。
getCustomProperties	指定されたアクティビティーまたはプロセス・インスタンスの指定されたカスタム・プロパティーを取得します。
getCustomPropertyNames	指定されたアクティビティーまたはプロセス・インスタンスのカスタム・プロパティーの名前を取得します。
setCustomProperty	指定されたアクティビティーまたはプロセス・インスタンスのカスタム固有値を保管します。

ヒューマン・タスク用のアプリケーションの開発

タスクは、コンポーネントが人をサービスとして呼び出したり、人がサービスを呼び出すための手段となります。ヒューマン・タスクに関する標準的なアプリケーションの例が提供されています。

Business Process Choreographer API について詳しくは、com.ibm.task.api パッケージにある Javadoc を参照してください。

ヒューマン・タスク用の許可のロール

ヒューマン・タスクで実行できるアクションは、許可のロールによって異なります。このロールは、J2EE ロールまたはインスタンス・ベースのロールです。

ロールは、同一レベルの権限を共有する従業員のグループです。 Java 2 Platform Enterprise Edition (J2EE) ロールは、ヒューマン・タスク・コンテナーが構成された場合にセットアップされます。インスタンス・ベースのロールは、ヒューマン・タスクに割り当てられ、タスクがモデル化されるとエスカレーションします。ロール・ベースの許可は、 WebSphere Application Server でグローバル・セキュリティーが使用可能になっている必要があります。

J2EE ロール

次の J2EE ロールがサポートされています。

- J2EE TaskSystemAdministrator。このロールに割り当てられたユーザーは、全特権を持ちます。
- J2EE TaskSystemMonitor。このロールに割り当てられたユーザーは、すべてのタスク・オブジェクトのプロパティーを表示できます。

管理コンソールを使用して、これらのロールへのユーザーおよびグループの割り当てを変更することができます。

RACF セキュリティーによるロールの設定: この RACF 許可は、以下のセキュリティー・フィールドを指定した場合に適用されます。

- **com.ibm.security.SAF.authorization= true**

```
RDEFINE EJBROLE TaskSystemAdministrator  UACC(NONE)
PERMIT TaskSystemAdministrator CLASS(EJBROLE)  ID(userid) ACCESS(READ)

RDEFINE EJBROLE TaskSystemMonitor  UACC(NONE)
PERMIT TaskSystemMonitor  CLASS(EJBROLE)  ID(userid) ACCESS(READ)
```

- **com.ibm.security.SAF.delegation= true**

```
RDEFINE EJBROLE JMSAPIUser  UACC(NONE) APPLDATA(' userid')
```

Security Authorization Facility (SAF) ベースの許可 (RACF EJBROLE プロファイルの使用など) を使用して、WebSphere Application Server 管理コンソール・アプリケーションなど、EJB および Web アプリケーションにおける Java 2 Platform、Enterprise Edition (J2EE) ロールへのクライアントのアクセスを制御できます。詳細については、WebSphere Application Server for z/OS インフォメーション・センターの『役割ベースの許可の System Authorization Facility』を参照してください。

インスタンス・ベースのロール。

タスク・インスタンスまたはエスカレーション・インスタンスは、タスク・モデル内でスタッフ・メンバーに直接割り当てられるのではなく、使用可能なロールのいずれかに割り当てられます。インスタンス・ベースのロールに割り当てられたスタッフ・メンバーは、そのロールのアクションを実行できます。インスタンス・ベースのロールに対するユーザーの関連は、実行時にスタッフ解決を使用して判別されます。

次のインスタンス・ベースのロールがサポートされています。
- タスクの場合: 潜在的なインスタンス作成者、オリジネーター、潜在的なスターター、スターター、潜在的な所有者、所有者、リーダー、エディター、管理者
- エスカレーションの場合: エスカレーション受信側

これらのロールは、次のアクションを実行することが許可されています。

ロール	許可されたアクション
潜在的なインスタンス作成者	このロールのメンバーは、タスクのインスタンスを作成できます。タスク・テンプレートまたはアプリケーション・コンポーネントに対して潜在的なインスタンス作成者が定義されていない場合は、すべてのユーザーがこのロールのメンバーであると見なされます。
オリジネーター	このロールのメンバーは、タスク開始まで管理権限を持ちます。タスクが開始されると、オリジネーターはリーダーの権限を持ち、タスクの中断と再開や作業項目の転送など、いくつかの管理アクションを実行できます。
潜在的なスターター	このロールのメンバーは、既存のタスク・インスタンスを開始できます。潜在的なスターターを指定しない場合、オリジネーターが潜在的なスターターになります。潜在的なスターターのいないインライン・タスクでは、デフォルトは全ユーザーになります。
スターター	このロールのメンバーは、リーダーの権限を持ち、作業項目の転送など、いくつかの管理アクションを実行できます。
潜在的な所有者	このロールのメンバーは、タスクを要求できます。タスク・テンプレートまたはアプリケーション・コンポーネントに対して潜在的な所有者が定義されていない場合は、すべてのユーザーがこのロールのメンバーであると見なされます。
所有者	タスクを処理し、完了します。
リーダー	すべてのタスク・オブジェクトのプロパティーを表示できますが、それらを処理することはできません。
エディター	このロールのメンバーは、タスクの内容を処理できますが、タスクを要求または完了することはできません。
管理者	このロールのメンバーは、タスク、タスク・テンプレート、およびエスカレーションを管理できます。
エスカレーションの受信側	このロールのメンバーは、リーダーの権限を持ちます。

タスクに対するアクションに必要なロール:

LocalHumanTaskManager または HumanTaskManager インターフェースへのアクセス権は、呼び出し元がタスクに対するすべてのアクションを実行できることは保証しません。呼び出し元は、そのアクションの実行も許可されている必要があります。次の表に、それぞれのロールで実行できるアクションを示します。

アクション	呼び出し元のプリンシパルのロール								
	所有者	(潜) 所有者	スターター	(潜) スターター	Origin	Admin	エディター	リーダー	Esc 受信側
callTask				X[1]	X[1]	X[1]			
cancelClaim	X					X			
claim		X				X			
complete	X					X			
createFaultMessage	X	X	X	X	X[1]	X	X	X	X
createInputMessage	X	X	X	X	X[1]	X	X	X	X
createOutputMessage	X	X	X	X	X[1]	X	X	X	X
createWorkItem					X[1、2]	X			
delete					X[3]	X			
deleteWorkItem					X[1、2]	X			
getCustomProperty	X	X	X	X	X[1]	X	X	X	X
getDocumentation	X	X	X	X	X[1]	X	X	X	X
getFaultMessage	X	X	X	X	X[1]	X	X	X	X
getFaultNames	X	X	X	X	X[1]	X	X	X	X
getInputMessage	X	X	X	X	X[1]	X	X	X	X
getOutputMessage	X	X	X	X	X[1]	X	X	X	X
getRoleInfo	X	X	X	X	X[1]	X	X	X	X
getTask	X	X	X	X	X[1]	X	X	X	X
getUISettings	X	X	X	X	X[1]	X	X	X	X
resume	X				X[1]	X			
setCustomProperty	X		X			X	X		
setFaultMessage	X					X	X		
setOutputMessage	X					X	X		
startTask				X	X[1]	X			
suspend	X				X[1]	X			
terminate	X		X[1]			X			
transferWorkItem	X		X		X[1]	X			
update	X		X			X	X		

アクション	呼び出し元のプリンシパルのロール								
	所有者	(潜) 所有者	スターター	(潜) スターター	Origin	Admin	エディター	リーダー	Esc 受信側

注:

1. スタンドアロンのタスクおよびタスク・テンプレートの場合のみ。
2. 非アクティブ状態のタスクの場合のみ。
3. オリジネーターは、非アクティブ状態のタスクのみを削除することができます。

略語:

Admin 管理者

Esc 受信側
エスカレーションの受信側

Origin オリジネーター

(潜) 所有者
潜在的な所有者

(潜) スターター
潜在的なスターター

同期インターフェースを起動する親タスクの開始

同期インターフェースを起動する親タスクには、microflow 内のインライン親タスク、microflow 内のスタンドアロン親タスク、および単純 Java クラスなどを開始する親タスクが含まれます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。両方向操作が戻るまで、タスクは実行状態のままです。タスクの結果である OrderNo が呼び出し元に戻されます。

1. **オプション:** タスク・テンプレートをリストして、実行する親タスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
    "TASK_TEMPL.NAME",
     new Integer(50),
     null);
   ```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
   ```

```
  myMessage = (DataObject)input.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
```

3. タスクを作成して、タスクを同期実行します。

   タスクを同期実行するには、両方向の操作であることが必要です。例では、createAndCallTask メソッドを使用してタスクを作成および実行します。

```
ClientObjectWrapper output = task.createAndCallTask( template.getName(),
                                                     template.getNamespace(),
                                                     input);
```

4. タスクの結果を分析します。

```
DataObject myOutput = null;
if ( output.getObject() != null && output.getObject() instanceof DataObject )
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

非同期インターフェースを起動する親タスクの開始

同期インターフェースを起動する親タスクには、microflow 内のインライン親タスク、microflow 内のスタンドアロン親タスク、および単純 Java クラスなどを開始する親タスクが含まれます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。

1. **オプション:** タスク・テンプレートをリストして、実行する親タスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

```
TaskTemplate[] taskTemplates = task.queryTaskTemplates
  ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
   "TASK_TEMPL.NAME",
    new Integer(50),
    null);
```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

```
TaskTemplate template = taskTemplates[0];

// create a message for the selected task
ClientObjectWrapper input = task.createInputMessage( template.getID());
DataObject myMessage = null ;
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
```

3. タスクを作成して、非同期に実行します。

   例では、createAndStartTask メソッドを使用してタスクを作成および実行します。

```
task.createAndStartTask( template.getName(),
                         template.getNamespace(),
                         input,
                         null);
```

タスク・インスタンスの作成と開始

このシナリオでは、ヒューマン・タスクを定義するタスク・テンプレートのインスタンスを作成し、タスク・インスタンスを開始する方法を示します。

1. **オプション:** タスク・テンプレートをリストして、実行する親タスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_HUMAN",
    "TASK_TEMPL.NAME",
     new Integer(50),
     null);
   ```

   結果は名前でソートされます。ソート済みのヒューマン・タスク・テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. ヒューマン・タスクを作成し、開始します。この例では、応答ハンドラーは指定されません。

   この例では、createAndStartTask メソッドを使用してタスクを作成し、開始します。

   ```
   TKIID tkiid = task.createAndStartTask( template.getName(),
                                          template.getNamespace(),
                                          input
                                          null);
   ```

   タスク・インスタンスに関連する人に対して作業項目が作成されます。例えば、潜在的な所有者は、新規タスク・インスタンスを要求できます。

4. タスク・インスタンスを要求します。

   ```
   ClientObjectWrapper input2 = task.claim(tkiid);
   DataObject taskInput = null ;
   if ( input2.getObject()!= null && input2.getObject() instanceof DataObject )
   {
     taskInput = (DataObject)input2.getObject();
     // read the values
     ...
   }
   ```

   タスク・インスタンスが要求されると、タスクの入力メッセージが戻されます。

参加タスクまたは純粋なヒューマン・タスクの処理

参加タスクや純粋なヒューマン・タスクは、作業項目を通じて、組織内のさまざまな人に割り当てられます。プロセスが staff アクティビティーにナビゲートしたときなどに、参加タスクとそれに関連した作業項目が作成されます。潜在的な所有者の中の 1 人が、作業項目に関連したタスクを要求します。このユーザーは、関係のある情報の提供とタスクの完了に対して責任があります。

1. 作業の準備ができている、ログオン・ユーザーに属するタスクをリストします。

```
QueryResultSet result =
     task.query("TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_READY AND
                (TASK.KIND = TASK.KIND.KIND_PARTICIPATING OR
                 TASK.KIND = TASK.KIND.KIND_HUMAN)AND
                 WORK_ITEM.REASON =
                   WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                 null, null, null);
```

   このアクションは、ログオン・ユーザーが作業することができるタスクが含まれる照会結果セットを戻します。

2. 作業対象のタスクを要求します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper input = task.claim(tkiid);
  DataObject taskInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   タスクが要求されると、タスクの入力メッセージが戻されます。

3. タスクの作業が完了した場合、タスクを完了します。

   タスクは、正常に完了することも、障害メッセージが表示されて完了することもあります。タスクが正常に完了した場合、出力メッセージが渡されます。タスクが正常に完了しなかった場合、障害メッセージが渡されます。これらのアクションに対して、適切なメッセージを作成する必要があります。

   a. タスクを正常に完了するには、出力メッセージを作成します。

```
ClientObjectWrapper output =
      task.createOutputMessage(tkiid);
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the task
task.complete(tkiid, output);
```

      このアクションは、オーダー番号が含まれる出力メッセージを設定します。タスクは、完了状態になります。

b. 障害が発生した場合にタスクを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the task
List faultNames = task.getFaultNames(tkiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      task.createFaultMessage(tkiid, (String)faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

task.complete(tkiid, (String)faultNames.get(0), myFault);
```

このアクションにより、エラー・コードを含む障害メッセージが設定されます。タスクは、失敗状態になります。

タスク・インスタンスの中断と再開

ヒューマン・タスク・インスタンスまたは参加しているタスク・インスタンスを中断し、それらを再び再開して完了することができます。

タスク・インスタンスは、作動可能状態または要求済み状態にすることができます。これをエスカレートすることが可能です。呼び出し元は、タスク・インスタンスの所有者、オリジネーター、または管理者である必要があります。

タスク・インスタンスは、実行中に中断することができます。そうすることによって、例えば、タスクの完了に必要な情報を収集することができます。情報が使用可能になったら、タスク・インスタンスを再開できます。

1. ログオン・ユーザーによって要求されたタスクのリストを取得します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.STATE = TASK.STATE.STATE_CLAIMED",
                                   null, null, null);
```

このアクションにより、ログオン・ユーザーによって要求されたタスクのリストを含む照会結果セットが戻されます。

2. タスク・インスタンスを中断します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  task.suspend(tkiid);
}
```

このアクションにより、指定されたタスク・インスタンスが中断されます。タスク・インスタンスは中断状態になります。

3. プロセス・インスタンスを再開します。

```
task.resume( tkiid );
```

このアクションにより、タスク・インスタンスが中断前の状態になります。

タスクの結果の分析

参加タスクまたは純粋なヒューマン・タスクは非同期に実行されます。タスク開始時に応答ハンドラーが指定された場合、タスク完了時に自動的に出力メッセージが戻されます。応答ハンドラーが指定されていない場合、メッセージを明示的に検索する必要があります。

タスクの結果は、そのタスク・インスタンスの派生元となったタスク・テンプレートに、派生したタスク・インスタンスの自動削除が指定されていない場合にのみ、データベースに保管されます。

タスクの結果を分析します。

例では、正常に完了したタスクのオーダー番号を確認する方法を示します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.NAME = 'CustomerOrder' AND
                                    TASK.STATE = TASK.STATE.STATE_FINISHED",
                                    null, null, null);
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper output = task.getOutputMessage(tkiid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject)
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

タスク・インスタンスの終了

管理者権限を有する担当者が、リカバリー不能状態になったと分かったタスク・インスタンスを終了する必要が生じる場合があります。 例えば、アプリケーションが起動されて障害が発生し、休止状態に戻らない場合などです。

例外的な状況の場合のみ、タスク・インスタンスを終了することをお勧めします。タスク・インスタンスは即時に終了されます。

1. 終了するタスク・インスタンスを検索します。

   ```
   Task taskInstance = task.getTask(tkiid);
   ```

2. タスク・インスタンスを終了します。

   ```
   TKIID tkiid = taskInstance.getID();
   task.terminate(tkiid);
   ```

   タスク・インスタンスは、未解決のタスクを待機しないで即時に終了します。

タスク・インスタンスの削除

タスク・インスタンスは、インスタンスの派生元となった関連タスク・テンプレートに自動削除が指定されている場合にのみ、完了時に自動的に削除されます。以下の例では、完了して自動的に削除されなかったタスク・インスタンスすべてを削除する方法を示します。

1. 完了したタスク・インスタンスをリストします。

```
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_FINISHED",
                 null, null, null);
```

このアクションにより、完了したタスク・インスタンスをリストした照会の結果セットが戻されます。

2. 完了したタスク・インスタンスを削除します。

```
while (result.next() )
{
 TKIID tkiid = (TKIID) result.getOID(1);
 task.delete(tkiid);
}
```

要求されたタスクの解放

管理者権限を持つユーザーが、別のユーザーによって要求されたタスクの解放を実行する必要がある場合があります。この状態は、例えば、タスクを完了する必要があるが、タスクの所有者が不在の場合に発生する可能性があります。タスクの所有者も、要求されたタスクを解放できます。

1. 特定のユーザー (例では、Smith) 所有の、要求されたタスクをリストします。

```
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_CLAIMED AND
                 TASK.OWNER = 'Smith'",
                 null, null, null);
```

このアクションは、指定されたユーザーの Smith が要求したタスクをリストする照会結果セットを戻します。

2. 要求されたタスクを解放します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  task.cancelClaim(tkiid);
}
```

このアクションは、タスクを作動可能状態に戻すので、他の潜在的な所有者の 1 人が要求することができます。

作業項目の管理

作業項目は、特定の理由でのユーザーまたはユーザーのグループへのオブジェクトの割り当てを表します。通常、このオブジェクトは、staff アクティビティー・インスタンス、プロセス・インスタンス、ヒューマン・タスクのいずれかです。理由は、ユーザーがアクティビティーまたはタスクに対して持っているロールから派生します。ユーザーは、アクティビティーまたはタスクとの関連において異なるロールを持つことができ、作業項目はこのようなロールそれぞれに対して作成されるため、1 つのアクティビティーやタスクが複数の作業項目を持つことが可能です。

アクティビティー・インスタンスまたはタスク・インスタンスの存続時間中に、オブジェクトに関連したユーザーのセットが変わる場合があります。例えば、社員が休暇に入ったり、新しい社員を雇用したり、またはワークロードを異なった方法で

分散させる必要がある場合です。このような変更に対応するために、作業項目を作成、削除、または転送するようにアプリケーションを開発することができます。

作業項目を管理するために実行可能なアクションは、ユーザーのロールによって異なります。例えば、管理者は作業項目を作成、削除、および転送できますが、タスク所有者は作業項目の転送のみが可能です。

- 作業項目を作成します。

```
// query the task instance for which an additional
// administrator is to be specified
QueryResultSet result = task.query("TASK.TKIID",
                                   "TASK.NAME='CustomerOrder'",
                                    null, null, null);
if ( result.size() > 0 )
{
  result.first();
  // create the work item
  task.createWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_ADMINISTRATOR,"Smith");
}
```

  このアクションによって、管理者のロールがあるユーザー Smith の作業項目が作成されます。

- 作業項目を削除します。

```
// query the task instance for which a work item is to be deleted
QueryResultSet result = task.query("TASK.TKIID",
                                   "TASK.NAME='CustomerOrder'",
                                    null, null, null);
if ( result.size() > 0 )
{
  result.first();
  // delete the work item
  task.deleteWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_READER,"Smith");
}
```

  このアクションによって、リーダーのロールがあるユーザー Smith の作業項目が削除されます。

- 作業項目を転送します。

```
// query the task that is to be rescheduled
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.NAME='CustomerOrder' AND
                 TASK.STATE=TASK.STATE.STATE_READY AND
                 WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER AND
                 WORK_ITEM.OWNER_ID='Miller'",
                 null, null, null);
if ( result.size() > 0 )
{
  result.first();
  // transfer the work item from user Miller to user Smith
  // so that Smith can work on the task
  task.transferWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_POTENTIAL_OWNER,"Miller","Smith");
}
```

  このアクションによって、作業項目をユーザー Smith に転送するので、 Smith は作業項目で作業することができます。

実行時のタスク・テンプレートおよびタスク・インスタンスの作成

通常、WebSphere Integration Developer などのモデル化ツールを使用して、タスク・テンプレートを作成します。次に、タスク・テンプレートを WebSphere Process Server にインストールし、例えば Business Process Choreographer Explorer を使用して、これらのテンプレートからインスタンスを作成します。ただし、実行時にヒューマン・タスクまたは参加タスクのインスタンスまたはテンプレートを作成することもできます。例えば、アプリケーションのデプロイ時にタスク定義が使用不可である場合、ワークフローに含まれるタスクがまだ認識されていない場合、またはタスクがユーザーのグループ間の随時のコラボレーションを対象とする必要がある場合などに、このようにすることがあります。

1. **オプション:** ご使用のインターフェースに、単純 Java 型ではない型が含まれる場合、ランタイム・タスクまたはテンプレートが使用するデータ型を含むアプリケーションを作成または特定します。

   ランタイム・タスクまたはタスク・テンプレートは、アプリケーションのコンテキストで実行され、データ型へアクセスできるようになります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにタスク定義またはプロセス定義も含めるようにしてください。これらのタスクまたはプロセスは、ダミー・タスクまたはダミー・プロセスでもかまいません。
2. タスク・モデルを作成します。

   このモデルは、ステップ 1 で特定されたアプリケーション内のデータ型を参照します。
3. タスク・モデルを検証します。
4. タスク・テンプレートまたはタスク・インスタンスを作成します。

   HumanTaskManagerService インターフェースを使用して、このアクションを完了します。インターフェースに単純 Java 型以外の型が含まれる場合は、タスク・インスタンスまたはタスク・テンプレートを作成するときに、そのデータ型定義を含むアプリケーションの名前を指定してください。

単純 Java 型を使用するランタイム・タスクの作成:

この例では、インターフェースで単純 Java 型 (例えば String オブジェクト) のみを使用するランタイム・タスクを作成します。

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

   ```
   ClientTaskFactory factory = ClientTaskFactory.newInstance();
   ResourceSet resourceSet = factory.createResourceSet();
   ```
2. 操作の WSDL 定義を作成し、記述を追加します。

   ```
   // create the WSDL interface
   Definition definition = factory.createWSDLDefinition
            ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

   // create a port type
   ```

```
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input and output messages are of type String:
// a fault message is not specified
Operation operation = factory.createOperation
        ( definition, portType, "doIt",
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          null );
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。アプリケーションが単純 Java 型のみを使用するため、アプリケーション名を指定する必要はありません。

   - 以下の断片はタスク・インスタンスを作成します。

```
task.createTask( taskModel, null, "HTM" );
```

   - 以下の断片はタスク・テンプレートを作成します。

```
task.createTaskTemplate( taskModel, null );
```

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

複合タイプを使用するランタイム・タスクの作成:

この例では、インターフェースで複合タイプを使用するランタイム・タスクを作成します。複合タイプは既に定義されています。すなわち、クライアントのローカル・ファイル・システムには、複合タイプの記述を含む XSD ファイルがあります。

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

   ```
   ClientTaskFactory factory = ClientTaskFactory.newInstance();
   ResourceSet resourceSet = factory.createResourceSet();
   ```

2. 複合タイプの XSD 定義をリソース・セットに追加して、操作の定義時に使用できるようにします。

   ファイルは、コードが実行されたロケーションの相対位置に配置されます。

   ```
   factory.loadXSDSchema( resourceSet, "InputBO.xsd" );
   factory.loadXSDSchema( resourceSet, "OutputBO.xsd" );
   ```

3. 操作の WSDL 定義を作成し、記述を追加します。

   ```
   // create the WSDL interface
   Definition definition = factory.createWSDLDefinition
            ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

   // create a port type
   PortType portType = factory.createPortType( definition, "doItPT" );

   // create an operation; the input message is an InputBO and
   // the output message an OutputBO;
   // a fault message is not specified
   Operation operation = factory.createOperation
           ( definition, portType, "doIt",
             new QName( "http://Input", "InputBO" ),
             new QName( "http://Output", "OutputBO" ),
             null );
   ```

4. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

   ```
   TTask humanTask = factory.createTTask( resourceSet,
                                          TTaskKinds.HTASK_LITERAL,
                                          "TestTask",
                                          new UTCDate( "2005-01-01T00:00:00" ),
                                          "http://www.ibm.com/task/test/",
                                          portType,
                                          operation );
   ```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

5. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

6. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

7. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

8. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "BOapplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "BOapplication" );
     ```

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

既存のインターフェースを使用するランタイム・タスクの作成:

この例は、既に定義されているインターフェースを使用するランタイム・タスクを作成します。すなわち、クライアントのローカル・ファイル・システムには、インターフェースの記述を含むファイルがあります。

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. 操作の WSDL 定義および記述にアクセスします。

   インターフェース記述は、コードが実行されたロケーションの相対位置に配置されます。

   ```
   Definition definition = factory.loadWSDLDefinition(
                             resourceSet, "interface.wsdl"  );
   PortType portType = definition.getPortType(
                        new QName( definition.getTargetNamespace(), "doItPT" ) );
   Operation operation = portType.getOperation("doIt", null, null );
   ```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

   ```
   TTask humanTask = factory.createTTask( resourceSet,
                                          TTaskKinds.HTASK_LITERAL,
                                          "TestTask",
                                          new UTCDate( "2005-01-01T00:00:00" ),
                                          "http://www.ibm.com/task/test/",
                                          portType,
                                          operation );
   ```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

   ```
   // use the methods from the com.ibm.wbit.tel package, for example,
   humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

   // retrieve the task factory to create or modify composite task elements
   TaskFactory taskFactory = factory.getTaskFactory();

   // specify escalation settings
   TVerb verb = taskFactory.createTVerb();
   verb.setName("John");

   // create escalationReceiver and add verb
   TEscalationReceiver escalationReceiver =
                       taskFactory.createTEscalationReceiver();
   escalationReceiver.setVerb(verb);

   // create escalation and add escalation receiver
   TEscalation escalation = taskFactory.createTEscalation();
   escalation.setEscalationReceiver(escalationReceiver);
   ```

5. すべてのリソース定義を含むタスク・モデルを作成します。

   ```
   TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
   ```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

   ```
   ValidationProblem[] validationProblems = taskModel.validate();
   ```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

- 以下の断片はタスク・インスタンスを作成します。

  ```
  task.createTask( taskModel, "BOapplication", "HTM" );
  ```

- 以下の断片はタスク・テンプレートを作成します。

  ```
  task.createTaskTemplate( taskModel, "BOapplication" );
  ```

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

呼び出し側アプリケーションのインターフェースを使用するランタイム・タスクの作成:

この例では、呼び出し側アプリケーションに属するインターフェースを使用するランタイム・タスクを作成します。例えば、ランタイム・タスクがビジネス・プロセスの Java 断片で作成され、プロセス・アプリケーションからのインターフェースを使用します。

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

   ```
   ClientTaskFactory factory = ClientTaskFactory.newInstance();

   // specify the context class loader so that following resources are found
   ResourceSet resourceSet = factory.createResourceSet
                ( Thread.currentThread().getContextClassLoader() );
   ```

2. 操作の WSDL 定義および記述にアクセスします。

   収容パッケージ JAR ファイル内でパスを指定します。

   ```
   Definition definition = factory.loadWSDLDefinition( resourceSet,
                           "com/ibm/workflow/metaflow/interface.wsdl" );
                        PortType portType = definition.getPortType(
                        new QName( definition.getTargetNamespace(), "doItPT" ) );
   Operation operation = portType.getOperation("doIt", null, null );
   ```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

   ```
   TTask humanTask = factory.createTTask( resourceSet,
                                          TTaskKinds.HTASK_LITERAL,
                                          "TestTask",
                                          new UTCDate( "2005-01-01T00:00:00" ),
                                          "http://www.ibm.com/task/test/",
                                          portType,
                                          operation );
   ```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

   ```
   // use the methods from the com.ibm.wbit.tel package, for example,
   humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

   // retrieve the task factory to create or modify composite task elements
   ```

```
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

   ```
   TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
   ```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

   ```
   ValidationProblem[] validationProblems = taskModel.validate();
   ```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "WorkflowApplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "WorkflowApplication" );
     ```

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

HumanTaskManagerService インターフェース

HumanTaskManagerService インターフェースは、ローカルまたはリモート・クライアントから呼び出すことができるタスク関連機能を公開します。

呼び出すことができるメソッドは、タスクの状態、およびそのメソッドが含まれているアプリケーションを使用する人物の権限によって異なります。タスク・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび HumanTaskManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.task.api パッケージ内の Javadoc を参照してください。

タスク・テンプレート

タスク・テンプレートを扱う作業には、以下のメソッドが使用可能です。

表 6. タスク・テンプレート用の API メソッド

メソッド	説明
getTaskTemplate	指定されたタスク・テンプレートを取得します。

表 6. タスク・テンプレート用の API メソッド (続き)

メソッド	説明
createAndCallTask	指定されたタスク・テンプレートからタスク・インスタンスを作成および実行し、同期式に結果を待機します。
createAndStartTask	指定されたタスク・テンプレートからタスク・インスタンスを作成および開始します。
createTask	指定されたタスク・テンプレートからタスク・インスタンスを作成します。
createInputMessage	指定されたタスク・テンプレート用の入力メッセージを作成します。例えば、タスクの開始に使用できるメッセージを作成します。
queryTaskTemplates	データベースに保管されているタスク・テンプレートを取得します。

タスク・インスタンス

タスク・インスタンスを扱う作業には、以下のメソッドが使用可能です。

表 7. タスク・インスタンス用の API メソッド

メソッド	説明
getTask	タスク・インスタンスを取得します。任意の状態のタスク・インスタンスを取得できます。
callTask	親タスクを同期で開始します。
startTask	既に作成済みのタスクを開始します。
suspend	ヒューマン・タスクまたは参加タスクを中断します。
resume	ヒューマン・タスクまたは参加タスクを再開します。
terminate	指定されたタスク・インスタンスを終了します。親タスクを終了する場合、このアクションは、呼び出されたサービスには影響しません。
delete	指定されたタスク・インスタンスを削除します。
claim	処理のためタスクを要求します。
update	タスク・インスタンスを更新します。
complete	タスク・インスタンスを完了します。
cancelClaim	別の潜在的な所有者が処理できるように、要求されたタスク・インスタンスをリリースします。
createWorkItem	タスク・インスタンスの作業項目を作成します。
transferWorkItem	指定された所有者に作業項目を転送します。
deleteWorkItem	作業項目を削除します。

エスカレーション

エスカレーションを扱う作業には、以下のメソッドが使用可能です。

表 8. エスカレーションで使用できる API メソッド

メソッド	説明
getEscalation	指定されたエスカレーション・インスタンスを取得します。

変数およびカスタム・プロパティー

このインターフェースは、変数の値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスに関連付けたり、指定されたプロパティーをタスク・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。

表 9. 変数およびカスタム・プロパティーの API メソッド

メソッド	説明
getCustomProperty	指定されたタスク・インスタンスの指定された 1 つのカスタム・プロパティーを取得します。
getCustomProperties	指定されたタスク・インスタンスの指定されたカスタム・プロパティー (複数) を取得します。
getCustomPropertyNames	タスク・インスタンスのすべてのカスタム・プロパティーの名前を取得します。
setCustomProperty	指定されたタスク・インスタンスのカスタム固有値を保管します。

各タスクで許可されるアクション:

タスクに対して実行可能なアクションは、そのタスクが参加タスク、純粋なヒューマン・タスク、親タスク、管理用タスクのどのタイプであるかによって異なります。

LocalHumanTaskManager または HumanTaskManager インターフェースによって提供されるすべてのアクションが、すべての種類のタスクに対して使用できるわけではありません。次の表に、タスクの種類ごとに実行可能なアクションを示します。

アクション	タスクの種類			
	参加タスク	ヒューマン・タスク	親タスク	管理用タスク
callTask			X[1]	
cancelClaim	X	X[1]		
claim	X	X[1]		
complete	X	X[1]		X
createFaultMessage	X	X	X	X

アクション	タスクの種類			
	参加タスク	ヒューマン・タスク	親タスク	管理用タスク
createInputMessage	X	X	X	X
createOutputMessage	X	X	X	X
createWorkItem	X	X[1]	X	X
delete	X[1]	X[1]	X	X[1]
deleteWorkItem	X	X[1]	X	X
getCustomProperty	X	X[1]	X	X
getDocumentation	X	X[1]	X	X
getFaultMessage	X	X[1]	X	
getInputMessage	X	X[1]	X	
getOutputMessage	X	X[1]	X	
getRoleInfo	X	X[1]	X	X
getTask	X	X[1]	X	X
getUISettings	X	X[1]	X	X
resume	X	X[1]		
setCustomProperty	X	X[1]	X	X
setFaultMessage	X	X[1]		
setOutputMessage	X	X[1]		
startTask	X[1]	X[1]	X	X
suspend	X	X[1]		
terminate	X[1]	X[1]	X[1]	
transferWorkItem	X	X[1]	X	X
updateInactiveTask	X[2]	X[3]	X[2]	X[2]
updateTask	X	X[1]	X	X

注:

1. スタンドアロンのタスク、ランタイム・タスク、およびタスク・テンプレートの場合のみ
2. スタンドアロンのタスク、ビジネス・プロセス内のインライン・タスク、およびランタイム・タスクの場合のみ
3. スタンドアロンのタスクおよびランタイム・タスクの場合のみ

ビジネス・プロセスおよびタスク関連のオブジェクトの照会

データベース内のビジネス・プロセス・オブジェクトおよびタスク関連のオブジェクトを照会して、これらのオブジェクトの特定のプロパティーを取得することができます。

Business Process Choreographer を構成すると、リレーショナル・データベースは、ビジネス・プロセス・コンテナーおよびタスク・コンテナーの両方に関連付けられます。このデータベースは、ビジネス・プロセスとタスクの管理用のすべてのテンプレート (モデル) とインスタンス (ランタイム) のデータを保管します。そのデータを照会するには、SQL 形式の構文を使用します。

単発の照会を実行して、オブジェクトの特定のプロパティーを取得することができます。また、頻繁に使用する照会を保管しておいて、この保管照会文をアプリケーションに組み込むこともできます。

ビジネス・プロセスおよびタスク関連オブジェクトに対する照会

サービス API の QUERY インターフェースを使用して、ビジネス・プロセスおよびタスクに関する保管情報を取得します。

オブジェクトのプロパティーを照会するために、事前定義データベース・ビューが提供されています。プロセス・テンプレートの場合、照会関数には以下の構文があります。

```
ProcessTemplateData[] queryProcessTemplates
                       (java.lang.String whereClause,
                        java.lang.String orderByClause,
                        java.lang.Integer threshold,
                        java.util.TimeZone timezone);
```

タスク・テンプレートの場合、照会関数には以下の構文があります。

```
TaskTemplate[] queryTaskTemplates
                (java.lang.String whereClause,
                 java.lang.String orderByClause,
                 java.lang.Integer threshold,
                 java.util.TimeZone timezone);
```

他のビジネス・プロセスおよびタスク関連オブジェクトの場合、照会関数には以下の構文があります。

```
QueryResultSet query (java.lang.String selectClause,
                      java.lang.String whereClause,
                      java.lang.String orderByClause,
                      java.lang.Integer skipTuples
                      java.lang.Integer threshold,
                      java.util.TimeZone timezone);
```

照会は以下のもので構成されます。

- select 文節
- where 文節
- order-by 文節
- スキップ・タプル・パラメーター
- しきい値パラメーター
- 時間帯パラメーター

例えば、関数の呼び出し元にアクセス可能な作業項目 ID のリストは、次のようにして取得します。

```
QueryResultSet result = process.query("WORK_ITEM.WIID",
                                      null, null, null, null, null);
```

照会関数は、呼び出し元の権限に応じてオブジェクトを戻します。照会結果セットには、呼び出し元が表示を許可されているオブジェクトのプロパティーのみが含まれます。

QUERY インターフェースには、queryAll メソッドも含まれています。このメソッドを使用して、オブジェクトに関係のあるデータすべてを、モニターなどの目的で取得することができます。queryAll メソッドの呼び出し元には、 Java 2 Platform

Enterprise Edition (J2EE) ロールの、BPESystemAdministrator、BPESystemMonitor、TaskSystemAdministrator、または TaskSystemMonitor のいずれかが必要です。オブジェクトの対応する作業項目を使用した許可検査は適用されません。

Business Process Choreographer API について詳しくは、プロセス関連メソッドの com.ibm.bpe.api パッケージおよびタスク関連メソッドの com.ibm.task.api パッケージ内にある Javadoc を参照してください。

select 文節:

照会関数の select 文節は、照会によって戻されるオブジェクト・プロパティーを示します。

select 文節は、照会結果を記述します。これは、戻すオブジェクト・プロパティー (結果の列) を識別する名前のリストを指定します。構文は SQL select 文節と同じで、コンマを使用して文節のパーツを分離します。文節の各パーツは、事前定義されているビューのいずれか 1 つのプロパティーを指定する必要があります。QueryResultSet オブジェクトで戻される列は、select 文節で指定されているプロパティーと同じ順序で表示されます。

select 文節は、AVG()、SUM()、MIN()、または MAX() などの SQL 集約関数はサポートしていません。

名前と値の対のプロパティー (カスタム・プロパティーなど) を選択する場合は、ビュー名に 1 桁のサフィックスを追加します。

select 文節の例

- ″WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON″

  関連オブジェクトのオブジェクト・タイプ、および作業項目の割り当て理由を取得します。

- ″DISTINCT WORK_ITEM.OBJECT_ID″

  呼び出し元が作業項目を所有しているオブジェクトの ID すべてを重複なしで取得します。

- ″ACTIVITY.TEMPLATE_NAME, WORK_ITEM.REASON″

  呼び出し元が作業項目を所有しているアクティビティーの名前、およびその割り当て理由を取得します。

- ″ACTIVITY.STATE, PROCESS_INSTANCE.STARTER″

  アクティビティーの状態、およびその関連プロセス・インスタンスのスターターを取得します。

- ″DISTINCT TASK.TKIID, TASK.NAME″

  呼び出し元が作業項目を所有しているタスクの ID と名前すべてを重複なしで取得します。

- ″TASK_CPROP1.STRING_VALUE, TASK_CPROP2.STRING_VALUE″

さらに where 文節でも指定されているカスタム・プロパティーの値を取得します。

- ″COUNT(DISTINCT TASK.TKIID)″

  where 文節の条件を満たす固有のタスクの作業項目の数を数えます。

select 文節の処理中にエラーが発生した場合は、QueryUnknownTable または QueryUnknownColumn 例外が、テーブルまたは列名として認識されないプロパティーの名前とともにスローされます。

where 文節:

照会関数の中の where 文節は、照会ドメインに適用するフィルター基準を記述します。

where 文節の構文は、SQL の where 文節と同じです。文節から明示的に SQL を追加したり、where 文節に述部を結合したりする必要はありません。これらの構成要素は照会の実行時に自動的に追加されます。フィルター基準を適用しない場合は、where 文節に `null` を指定する必要があります。

where 文節構文は以下のものをサポートします。

- キーワード: AND、OR、NOT
- 比較演算子: =、<=、<、<>、>、>=、LIKE
- 設定操作: IN

  LIKE 操作では、照会されるデータベースに定義されているワイルドカード文字がサポートされます。

以下の規則も適用されます。

- オブジェクト ID 定数を `ID('string-rep-of-oid')` に指定します。
- `BIN('UTF-8 string')` としてバイナリー定数を指定します。
- 整数列挙型の代わりにシンボリック定数を使用します。例えば、アクティビティー状態式 `ACTIVITY.STATE=2` を指定する代わりに、`ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY` を指定します。
- 比較ステートメント内のプロパティーの値に単一引用符 (') が含まれる場合、例えば `″TASK_CPROP.STRING_VALUE='d''automatisation'″` のように、引用符を二重にしてください。
- ビュー名に 1 桁のサフィックスを追加して、名前と値の対のプロパティー (カスタム・プロパティーなど) を参照します。例: `"TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"`
- タイム・スタンプ定数を `TS('yyyy-mm-ddThh:mm:ss')` に指定します。現在日付を参照するには、タイム・スタンプを `CURRENT_DATE` に指定します。

  タイム・スタンプには、最低でも日付または時間の値を指定する必要があります。

  – 日付のみを指定すると、時間値はゼロに設定されます。
  – 時間のみを指定すると、日付は現在の日付に設定されます。

– 日付を指定する場合、年は 4 桁の定数で構成する必要があります。月および日の値はオプションです。欠落している月および日の値は、01 に設定されます。例えば、TS('2003') は TS('2003-01-01T00:00:00') と同じです。
– 日付を指定する場合、この値は 24 時間制で記述されます。例えば、現在日付が 2003 年 1 月 1 日の場合、TS('T16:04') または TS('16:04') は、TS('2003-01-01T16:04:00') と同じです。

where 文節の例

- オブジェクト ID と既存の ID の比較

```
"WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"
```

この型の where 文節は、通常、直前の呼び出しの既存オブジェクト ID を使用して、動的に作成されます。このオブジェクト ID が *wiid1* 変数に保管されている場合、文節の構文は次のようになります。

```
"WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
```

- タイム・スタンプの使用

```
"ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"
```

- シンボリック定数の使用

```
"WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"
```

- ブール値 true および false の使用

```
"ACTIVITY.BUSINESS_RELEVANCE = TRUE"
```

- カスタム・プロパティーの使用

```
"TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' OR
 TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"
```

order-by 文節:

照会関数内の order-by 文節は、照会結果セットのソート基準を指定します。

order-by 文節の構文は、SQL の order-by 文節と同じで、コンマを使用して文節の各パーツを分離します。文節の各パーツは、事前定義されているビューのいずれか 1 つのプロパティーを指定する必要があります。

ソート基準はサーバーに適用されます。つまり、ソートにサーバーのロケールが使用されます。複数のプロパティーを指定すると、照会結果セットはまず最初のプロパティーの値で順序付けされ、次に 2 番目のプロパティーの値で順序付けされる、という具合に続きます。

照会結果セットをソートしない場合は、order-by 文節で null を指定する必要があります。

order-by 文節の例

- ″PROCESS_TEMPLATE.NAME″

照会結果を、プロセス・テンプレート名でアルファベット順にソートします。

- ″PROCESS_INSTANCE.CREATED, PROCESS_INSTANCE.NAME DESC″

照会結果を作成日でソートし、特定の日付の場合はその結果を、プロセス・インスタンス名でアルファベット順の逆順にソートします。

- ″ACTIVITY.OWNER, ACTIVITY_TEMPLATE.NAME, ACTIVITY.STATE″

  照会結果を、アクティビティー所有者、アクティビティー・テンプレート名、アクティビティーの状態の順でソートします。

スキップ・タプル・パラメーター:

スキップ・タプル・パラメーターは、照会の結果セットの中の、無視され、呼び出し元に戻されない照会結果セット・タプルの数を指定します。

このパラメーターは、しきい値パラメーターと併用して、クライアント・アプリケーションでのページングをインプリメントします。

このパラメーターを null に設定したときに、しきい値パラメーターを設定していないと、すべての適格なタプルが戻されます。

スキップ・タプル・パラメーターの例
- new Integer(5)

  最初の 5 つの適格なタプルを戻さないように指定します。

しきい値パラメーター:

照会関数のしきい値パラメーターは、照会の結果セットとしてサーバーからクライアントに戻されるオブジェクトの数を制限します。

しきい値パラメーターは、例えば少数の項目のみが表示されるグラフィカル・ユーザー・インターフェースなどで有用な場合があります。しきい値パラメーターを適宜設定すると、データベース照会が高速化し、サーバーからクライアントへ転送する必要のあるデータが少なくなります。

このパラメーターを null に設定したときに、スキップ・タプル・パラメーターを設定していないと、適格なオブジェクトがすべて戻されます。

しきい値パラメーターの例
- new Integer(50)

  50 個の適格なタプルを戻すように指定します。

時間帯パラメーター:

照会関数の時間帯パラメーターは、照会内のタイム・スタンプ定数の時間帯を定義します。

照会を開始するクライアントと照会を処理するサーバーの間で、時間帯が異なることがあります。時間帯パラメーターを使用して、where 文節で使用されるタイム・スタンプ定数の時間帯を、例えば地方時を指定するように指定します。照会の結果セットで戻される日付には、照会で指定したものと同じ時間帯が設定されます。

このパラメーターを null に設定すると、タイム・スタンプ定数は協定世界時 (UTC) と想定されます。

時間帯パラメーターの例

- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null,
                 null,
                 java.util.TimeZone.getDefault() );
  ```

  2005 年 1 月 1 日の 17:40 地方時より後に開始されたアクティビティーのオブジェクト ID を戻します。
- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 null, null, null);
  ```

  2005 年 1 月 1 日の 17:40 UTC より後に開始されたアクティビティーのオブジェクト ID を戻します。この指定は、例えば東部標準時より 6 時間早い時間です。

照会結果:

照会結果セットには、照会の結果が入ります。

結果セットのエレメントは、呼び出し元が表示を許可されているオブジェクトです。 エレメントは、次のメソッドを使用して相対的に読み取るか、あるいは最初と最後のメソッドを使用して絶対的に読み取ります。照会結果セットの暗黙カーソルは初めは最初のエレメントの前に配置されるため、エレメントを読み取る前に、最初または次のメソッドのいずれかを呼び出す必要があります。 size メソッドを使用して、セット内のエレメント数を判別することができます。

照会結果セットのエレメントは、作業項目とそれに関連する参照オブジェクト (アクティビティー・インスタンスやプロセス・インスタンスなど) の選択済み属性を構成します。 QueryResultSet エレメントの最初の属性 (列) は、照会要求の select 文節で指定されている最初の属性の値を指定します。 QueryResultSet エレメントの 2 番目の属性 (列) は、照会要求の select 文節で指定されている 2 番目の属性の値を指定する、という具合に続きます。

属性の値は、その属性タイプと互換性のあるメソッドを呼び出すことによって、また、適切な列インデックスを指定することによって、取得することができます。列インデックスの番号付けは 1 から始まります。

属性タイプ	**メソッド**
ストリング	getString
ID	getOID
タイム・スタンプ	getTimestamp getString
整数	getInteger getShort getLong getString getBoolean

属性タイプ	メソッド
ブール	getBoolean getShort getInteger getLong getString
CHAR FOR BIT DATA	getBinary

例:

以下の照会が実行されます。

```
QueryResultSet resultSet = process.query("ACTIVITY.STARTED,
                                         ACTIVITY.TEMPLATE_NAME AS NAME,
                                         WORK_ITEM.WIID, WORK_ITEM.REASON",
                                         null, null, null, null);
```

戻される照会結果セットには、以下の 4 つの列があります。

- 列 1 はタイム・スタンプ
- 列 2 はストリング
- 列 3 はオブジェクト ID
- 列 4 は整数

以下のメソッドを使用して、属性値を取得することができます。

```
while (resultSet.next())
{
 java.util.Calendar activityStarted = resultSet.getTimestamp(1);
 String templateName = resultSet.getString(2);
 WIID wiid = (WIID) resultSet.getOID(3);
 Integer reason = resultSet.getInteger(4);
}
```

結果セットの表示名を、例えば、印刷されるテーブルの見出しなどに使用することができます。これらの名前は、ビューの列名、または照会の AS 文節で定義された名前です。以下のメソッドを使用して、例中の表示名を取得することができます。

```
resultSet.getColumnDisplayName(1) returns "STARTED"
resultSet.getColumnDisplayName(2) returns "NAME"
resultSet.getColumnDisplayName(3) returns "WIID"
resultSet.getColumnDisplayName(4) returns "REASON"
```

保管照会文の管理

保管照会文は、データベースに保管され、名前で識別される照会のことです。照会の定義はデータベースに保管されますが、保管照会文に収容されている項目は、照会されたときに動的にアセンブルされます。すべての保管照会文は、公開されてアクセス可能です。ただし、ビジネス・プロセス管理者権限またはタスク管理者権限を持つ場合のみ、これらの保管照会文を作成および削除できます。保管照会文は、ビジネス・プロセス・オブジェクト、タスク・オブジェクト、またはこの 2 つのオブジェクト・タイプの組み合わせたものを対象とします。

1. 保管照会文を作成します。

   例えば、以下のコードの断片は、プロセス・インスタンスの照会を作成して、固有の名前を付けて保管します。

```
process.createStoredQuery("CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              null,null);
```

この照会は、文字 A で始まるプロセス・インスタンス名、および関連したプロセス・インスタンス ID (PIID) をすべてソートしたリストにして戻します。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
              new Integer(0));
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. **オプション:** 使用可能な保管照会文の一覧を示します。

例えば、以下のコードの断片は、プロセス・オブジェクトの保管照会文リストを取得する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames();
```

4. **オプション:** 特定の保管照会文で定義された照会を検査します。

```
StoredQuery storedQuery = process.getStoredQuery("CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
```

5. 保管照会文を削除します。

以下のコードの断片では、ステップ 1 で作成した保管照会文の削除方法を示しています。

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクトの照会のための事前定義ビュー

ビジネス・プロセス・オブジェクトおよびヒューマン・タスク・オブジェクト用に、事前定義データベース・ビューが提供されています。これらのオブジェクトの参照データを照会する場合は、これらのビューを使用します。これらのビューを使用する場合は、ビューの列に明示的に述部を結合する必要はありません。これらの構成要素は自動的に追加されます。このデータを照会するには、サービス API (BusinessFlowManagerService または HumanTaskManagerService) の汎用照会関数を使用することができます。HumanTaskManagerDelegate API の対応するメソッド、または ExecutableQuery インターフェースのインプリメンテーションによって提供される定義済み照会を使用することもできます。

ACTIVITY ビュー:

この定義済みデータベース・ビューは、アクティビティーの照会に使用します。

表 10. ACTIVITY ビュー内の列

列名	タイプ	コメント
PIID	ID	プロセス・インスタンス ID。
AIID	ID	アクティビティー・インスタンス ID。

表 10. ACTIVITY ビュー内の列 (続き)

列名	タイプ	コメント
PTID	ID	プロセス・テンプレート ID。
ATID	ID	アクティビティー・テンプレート ID。
KIND	整数	アクティビティーの種類。指定可能な値は、以下のとおりです。 KIND_INVOKE KIND_RECEIVE KIND_REPLY KIND_THROW KIND_RETHROW KIND_TERMINATE KIND_WAIT KIND_COMPENSATE KIND_SEQUENCE KIND_EMPTY KIND_SWITCH KIND_WHILE KIND_PICK KIND_FLOW KIND_SCOPE KIND_SCRIPT KIND_STAFF KIND_ASSIGN KIND_CUSTOM
COMPLETED	タイム・スタンプ	アクティビティーの完了時刻。
ACTIVATED	タイム・スタンプ	アクティビティーが活動化された時刻。
FIRST_ACTIVATED	タイム・スタンプ	そのアクティビティーが初めて活動化された時刻。
STARTED	タイム・スタンプ	アクティビティーの開始時刻。

表 10. ACTIVITY ビュー内の列 (続き)

列名	タイプ	コメント
STATE	整数	アクティビティーの状態。指定可能な値は、以下のとおりです。 STATE_INACTIVE STATE_READY STATE_RUNNING STATE_PROCESSING_UNDO STATE_SKIPPED STATE_FINISHED STATE_FAILED STATE_TERMINATED STATE_CLAIMED STATE_TERMINATING STATE_FAILING STATE_WAITING STATE_EXPIRED STATE_STOPPED
OWNER	ストリング	所有者のプリンシパル ID。
DESCRIPTION	ストリング	アクティビティー・テンプレートの説明にプレースホルダーが含まれている場合、この列には、解決済みのプレースホルダーを所有するアクティビティー・インスタンスの説明が入ります。
TEMPLATE_NAME	ストリング	関連するアクティビティー・テンプレートの名前。
TEMPLATE_DESCR	ストリング	関連するアクティビティー・テンプレートの説明。
BUSINESS_RELEVANCE	ブール	アクティビティーがビジネスと関係があるかどうかを指定します。 属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** アクティビティーはビジネスと関係があり、監査されます。 **FALSE** アクティビティーはビジネスとの関係がなく、監査されません。

ACTIVITY_ATTRIBUTE ビュー:

この定義済みデータベース・ビューは、アクティビティーのカスタム・プロパティーの照会に使用します。

表 11. ACTIVITY_ATTRIBUTE ビュー内の列

列名	タイプ	コメント
AIID	ID	カスタム・プロパティーを所有するアクティビティー・インスタンスの ID。

表 11. ACTIVITY_ATTRIBUTE ビュー内の列 (続き)

列名	タイプ	コメント
NAME	ストリング	カスタム・プロパティーの名前。
VALUE	ストリング	カスタム・プロパティーの値。

ACTIVITY_SERVICE ビュー:

この定義済みデータベース・ビューは、アクティビティー・サービスの照会に使用します。

表 12. ACTIVITY_SERVICE ビュー内の列

列名	タイプ	コメント
EIID	ID	イベント・インスタンスの ID。
AIID	ID	イベントを待機しているアクティビティーの ID。
PIID	ID	イベントが含まれているプロセス・インスタンスの ID。
VTID	ID	イベントを説明するサービス・テンプレートの ID。
PORT_TYPE	ストリング	ポート・タイプの名前。
NAME_SPACE_URI	ストリング	ネーム・スペースの URI。
OPERATION	ストリング	サービスのオペレーション名。

APPLICATION_COMP ビュー:

この定義済みデータベース・ビューは、アプリケーション・コンポーネント ID、およびタスクのデフォルト設定の照会に使用します。

表 13. APPLICATION_COMP ビュー内の列

列名	タイプ	コメント
ACOID	ストリング	アプリケーション・コンポーネントの ID。
BUSINESS_ RELEVANCE	ブール	コンポーネントのデフォルトのタスク・ビジネス関連ポリシー。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。
NAME	ストリング	アプリケーション・コンポーネントの名前。
SUPPORT_ AUTOCLAIM	ブール	コンポーネントのデフォルトの自動要求ポリシー。この属性が TRUE に設定されている場合、潜在的な所有者が単一のユーザーであれば、タスクを自動的に要求することができます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。

表 13. APPLICATION_COMP ビュー内の列 (続き)

列名	タイプ	コメント
SUPPORT_CLAIM_ SUSP	ブール	中断されているタスクを要求できるかどうかを判断するコンポーネントのデフォルト設定。この属性が TRUE に設定されている場合は、中断されているタスクを要求することができます。この値は、タスク・テンプレートまたはタスクの定義によって上書きされる場合があります。
SUPPORT_ DELEGATION	ブール	コンポーネントのデフォルトのタスク代行サポート・ポリシー。この属性が TRUE に設定されている場合は、タスクの作業項目割り当てを変更することができます。すなわち、作業項目の作成、削除、または転送が可能です。

ESCALATION ビュー:

この定義済みデータベース・ビューは、エスカレーションのデータの照会に使用します。

表 14. ESCALATION ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション・インスタンスの ID。
ACTION	整数	エスカレーションによって起動されるアクション。指定可能な値は、以下のとおりです。 **ACTION_CREATE_WORK_ITEM** それぞれのエスカレーションに対する受信側の作業項目を作成します。 **ACTION_SEND_EMAIL** それぞれのエスカレーションの受信側に E メールを送信します。 **ACTION_CREATE_EVENT** イベントを作成および公表します。
ACTIVATION_STATE	整数	対応するタスクが以下のいずれかの状態になると、エスカレーション・インスタンスが作成されます。 **ACTIVATION_STATE_READY** ヒューマン・タスクまたは参加タスクは、要求を受ける準備ができていることを示します。 **ACTIVATION_STATE_RUNNING** 親タスクが開始され、実行中であることを示します。 **ACTIVATION_STATE_CLAIMED** タスクが要求されることを明示します。
ACTIVATION_TIME	タイム・スタンプ	エスカレーションが活動化される時刻。

表 14. ESCALATION ビュー内の列 (続き)

列名	タイプ	コメント
AT_LEAST_ EXP_STATE	整数	エスカレーションによって予期されるタスクの状態。タイムアウトが発生した場合、タスク状態がこの属性の値と比較されます。指定可能な値は、以下のとおりです。 **AT_LEAST_EXPECTED_STATE _CLAIMED** タスクが要求されることを明示します。 **AT_LEAST_EXPECTED_STATE _ENDED** タスクが最終状態 (FINISHED、FAILED、TERMINATED、または EXPIRED) にあることを明示します。
ESTID	ストリング	対応するエスカレーション・テンプレートの ID。
FIRST_ESIID	ストリング	チェーン内の最初のエスカレーションの ID。
INCREASE_PRIORITY	整数	タスクの優先度を増やす方法を示します。指定可能な値は、以下のとおりです。 **INCREASE_PRIORITY_NO** タスクの優先度は増えません。 **INCREASE_PRIORITY_ONCE** タスクの優先度は 1 度に 1 ずつ増えます。 **INCREASE_PRIORITY_REPEATED** タスクの優先度は、エスカレーションが繰り返されるごとに 1 ずつ増えます。
NAME	ストリング	エスカレーションの名前。
STATE	整数	エスカレーションの状態。指定可能な値は、以下のとおりです。 STATE_INACTIVE STATE_WAITING STATE_ESCALATED STATE_SUPERFLUOUS
TKIID	ストリング	エスカレーションが所属するタスク・インスタンス ID。

ESCALATION_CPROP ビュー:

この定義済みデータベース・ビューは、エスカレーションのカスタム・プロパティーを照会するために使用します。

表 15. ESCALATION_CPROP ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション ID。
NAME	ストリング	プロパティーの名前。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

ESCALATION_DESC ビュー:

この定義済みデータベース・ビューは、エスカレーションのマルチリンガル記述データを照会するために使用します。

表 16. ESCALATION_DESC ビュー内の列

列名	タイプ	コメント
ESIID	ストリング	エスカレーション ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスク・テンプレートの説明。
DISPLAY_NAME	ストリング	エスカレーションの記述名。

PROCESS_ATTRIBUTE ビュー:

この定義済みデータベース・ビューは、プロセスのカスタム・プロパティーの照会に使用します。

表 17. PROCESS_ATTRIBUTE ビュー内の列

列名	タイプ	コメント
PIID	ID	カスタム・プロパティーを所有するプロセス・インスタンスの ID。
NAME	ストリング	カスタム・プロパティーの名前。
VALUE	ストリング	カスタム・プロパティーの値。

PROCESS_INSTANCE ビュー:

この定義済みデータベース・ビューは、プロセス・インスタンスの照会に使用します。

表 18. PROCESS_INSTANCE ビュー内の列

列名	タイプ	コメント
PTID	ID	プロセス・テンプレート ID。
PIID	ID	プロセス・インスタンス ID。
NAME	ストリング	プロセス・インスタンスの名前。

表 18. PROCESS_INSTANCE ビュー内の列 (続き)

列名	タイプ	コメント
STATE	整数	プロセス・インスタンスの状態。指定可能な値は、以下のとおりです。 STATE_READY STATE_RUNNING STATE_FINISHED STATE_COMPENSATING STATE_INDOUBT STATE_FAILED STATE_TERMINATED STATE_COMPENSATED STATE_COMPENSATION_FAILED STATE_TERMINATING STATE_FAILING STATE_SUSPENDED
CREATED	タイム・スタンプ	プロセス・インスタンスの作成時刻。
STARTED	タイム・スタンプ	プロセス・インスタンスの開始時刻。
COMPLETED	タイム・スタンプ	プロセス・インスタンスの完了時刻。
PARENT_NAME	ストリング	親プロセス・インスタンスの名前。
TOP_LEVEL_NAME	ストリング	トップレベル・プロセス・インスタンスの名前。トップレベルのプロセス・インスタンスがない場合、これは、現行プロセス・インスタンスの名前になります。
STARTER	ストリング	プロセス・インスタンスのスターターのプリンシパル ID。
DESCRIPTION	ストリング	プロセス・テンプレートの説明にプレースホルダーが含まれている場合、この列には、解決済みのプレースホルダーを所有するプロセス・インスタンスの説明が入ります。
TEMPLATE_NAME	ストリング	関連するプロセス・テンプレートの名前。
TEMPLATE_DESCR	ストリング	関連するプロセス・テンプレートの説明。

PROCESS_TEMPLATE ビュー:

この定義済みデータベース・ビューは、プロセス・テンプレートの照会に使用します。

表 19. PROCESS_TEMPLATE ビュー内の列

列名	タイプ	コメント
PTID	ID	プロセス・テンプレート ID。
NAME	ストリング	プロセス・テンプレートの名前。
VALID_FROM	タイム・スタンプ	プロセス・テンプレートのインスタンス化が可能になる時刻。
TARGET_NAMESPACE	ストリング	プロセス・テンプレートのターゲット・ネーム・スペース。
APPLICATION_NAME	ストリング	プロセス・テンプレートが所属するエンタープライズ・アプリケーションの名前。
VERSION	ストリング	ユーザー定義のバージョン。
CREATED	タイム・スタンプ	プロセス・テンプレートがデータベース内に作成される時刻。
STATE	整数	プロセス・インスタンスの作成にプロセス・テンプレートを使用できるかどうかを指定します。指定可能な値は、以下のとおりです。 STATE_STARTED STATE_STOPPED
EXECUTION_MODE	整数	このプロセス・テンプレートから派生したプロセス・インスタンスの実行方法を指定します。指定可能な値は、以下のとおりです。 EXECUTION_MODE_MICROFLOW EXECUTION_MODE_LONG_RUNNING
DESCRIPTION	ストリング	プロセス・テンプレートの説明。
COMP_SPHERE	整数	プロセス・テンプレート内の microflow のインスタンスの補正の振る舞いを指定します。既存の補正範囲を結合するか、補正範囲を作成するかのいずれかです。 指定可能な値は、以下のとおりです。 COMP_SPHERE_REQUIRED COMP_SPHERE_REQUIRES_NEW COMP_SPHERE_SUPPORTS COMP_SPHERE_NOT_SUPPORTED

TASK ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトの照会に使用します。

表 20. TASK ビュー内の列

列名	タイプ	コメント
TKIID	ID	タスク・インスタンスの ID。
ACTIVATED	タイム・スタンプ	タスクが活動化された時刻。

表 20. TASK ビュー内の列 (続き)

列名	タイプ	コメント
APPLIC_ DEFAULTS_ID	ID	タスクのデフォルト値を指定するアプリケーション・コンポーネントの ID。
APPLIC_NAME	ストリング	タスクが所属するエンタープライズ・アプリケーションの名前。
BUSINESS_ RELEVANCE	ブール	タスクがビジネスと関係があるかどうかを指定します。 属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。
COMPLETED	タイム・スタンプ	タスクが完了した時刻。
CONTAINMENT_ CTX_ID	ID	このタスクの包含コンテキスト。この属性は、タスクのライフ・サイクルを決定します。タスクの包含コンテキストを削除すると、タスク・テンプレートも削除されます。
CTX_ AUTHORIZATION	整数	タスクの所有者がタスク・コンテキストにアクセスできるようにします。指定可能な値は、以下のとおりです。 **AUTH_NONE** 関連コンテキスト・オブジェクトに対する許可権限はありません。 **AUTH_READER** 関連コンテキスト・オブジェクトに関する操作に、プロセス・インスタンスのプロパティーの読み取りなどのリーダー権限が必要です。
DUE	タイム・スタンプ	タスクの期限時刻。
EXPIRES	タイム・スタンプ	タスクの有効期限が切れる日付。
FIRST_ACTIVATED	タイム・スタンプ	タスクが初めて活動化された時刻。
IS_ESCALATED	ブール	このタスクのエスカレーションが発生済みかどうかを示します。
IS_INLINE	ブール	タスクがビジネス・プロセス内のインラインのタスクであるかどうかを示します。

表 20. TASK ビュー内の列 (続き)

列名	タイプ	コメント
KIND	整数	タスクの種類。指定可能な値は、以下のとおりです。 **KIND_HUMAN** タスクが人の手で作成され、処理されることを示します。 **KIND_WPC_STAFF_ACTIVITY** タスクが、WebSphere Business Integration Server Foundation バージョン 5 ビジネス・プロセスのスタッフ・アクティビティーであるヒューマン・タスクであることを示します。 **KIND_ORIGINATING** 人からコンピューターへの対話をタスクがサポートし、人がサービスを作成、開始、および始動できることを示します。 **KIND_PARTICIPATING** コンピューターから人への対話をタスクがサポート、人がサービスをインプリメントできることを示します。 **KIND_ADMINISTRATIVE** タスクが管理用タスクであることを示します。
LAST_MODIFIED	タイム・スタンプ	タスクの最終変更時刻。
LAST_STATE_ CHANGE	タイム・スタンプ	タスクの状態の最終変更時刻。
NAME	ストリング	タスクの名前。
NAME_SPACE	ストリング	タスクのカテゴリー化に使用されるネーム・スペース。
ORIGINATOR	ストリング	タスク・オリジネーターのプリンシパル ID。
OWNER	ストリング	タスクの所有者のプリンシパル ID。
PARENT_ CONTEXT_ID	ストリング	このタスクの親コンテキスト。この属性は、呼び出し側アプリケーション・コンポーネント内の対応するコンテキストに対するキーを提供します。親コンテキストは、そのタスクを作成するアプリケーション・コンポーネントによって設定されます。
PRIORITY	整数	タスクの優先順位。
STARTED	タイム・スタンプ	タスクが開始された時刻 (STATE_RUNNING、STATE_CLAIMED)。
STARTER	ストリング	タスク・スターターのプリンシパル ID。

表 20. TASK ビュー内の列 (続き)

列名	タイプ	コメント
STATE	整数	タスクの状態。指定可能な値は、以下のとおりです。 **STATE_READY** そのタスクは要求を受ける準備ができたことを示します。 **STATE_RUNNING** タスクが開始され、実行中であることを示します。 **STATE_FINISHED** タスクが正常に完了したことを示します。 **STATE_FAILED** タスクが正常に完了しなかったことを示します。 **STATE_TERMINATED** 外部要求または内部要求が原因で、タスクが終了したことを示します。 **STATE_CLAIMED** タスクが要求されることを示します。 **STATE_EXPIRED** 指定された期間を超えたため、タスクが終了したことを示します。
SUPPORT_ AUTOCLAIM	ブール	このタスクが単一ユーザーに割り当てられている場合に、自動的に要求されるかどうかを示します。
SUPPORT_CLAIM_ SUSP	ブール	このタスクが中断されている場合に、要求可能かどうかを示します。
SUPPORT_ DELEGATION	ブール	このタスクが、作業項目の作成、削除、または転送によって、作業代行をサポートするかどうかを示します。
SUSPENDED	ブール	タスクが中断されているかどうかを示します。
TKTID	ストリング	タスク・テンプレート ID。
TYPE	ストリング	タスクのカテゴリー化に使用されるタイプ。

TASK_CPROP ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトのカスタム・プロパティーを照会するために使用します。

表 21. TASK_CPROP ビュー内の列

列名	タイプ	コメント
TKIID	ストリング	タスク・インスタンス ID。
NAME	ストリング	プロパティーの名前。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

TASK_DESC ビュー:

この定義済みデータベース・ビューは、タスク・オブジェクトのマルチリンガル記述データを照会するために使用します。

表 22. TASK_DESC ビュー内の列

列名	タイプ	コメント
TKIID	ストリング	タスク・インスタンス ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスクの説明。
DISPLAY_NAME	ストリング	タスクの記述名。

TASK_TEMPL ビュー:

この定義済みデータベース・ビューは、タスクのインスタンスを生成するために使用できるデータを保持します。

表 23. TASK_TEMPL ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
VALID_FROM	タイム・スタンプ	インスタンス化にタスク・テンプレートが使用できるようになる時刻。
APPLIC_ DEFAULTS_ID	ストリング	タスク・テンプレートのデフォルト値を指定するアプリケーション・コンポーネントの ID。
APPLIC_NAME	ストリング	タスク・テンプレートが所属するエンタープライズ・アプリケーションの名前。
BUSINESS_ RELEVANCE	ブール	タスク・テンプレートがビジネスと関係があるかどうかを指定します。 属性は、監査証跡へのロギングに影響します。指定可能な値は、以下のとおりです。 **TRUE** タスクはビジネスと関係があり、監査されます。 **FALSE** タスクはビジネスとの関係がなく、監査されません。
CONTAINMENT_ CTX_ID	ID	このタスク・テンプレートの包含コンテキスト。この属性は、タスク・テンプレートのライフ・サイクルを決定します。包含コンテキストを削除すると、タスク・テンプレートも削除されます。

表 23. TASK_TEMPL ビュー内の列 (続き)

列名	タイプ	コメント
CTX_ AUTHORIZATION	整数	タスクの所有者がタスク・コンテキストにアクセスできるようにします。指定可能な値は、以下のとおりです。 **AUTH_NONE** 関連コンテキスト・オブジェクトに対する許可権限はありません。 **AUTH_READER** 関連コンテキスト・オブジェクトに関する操作に、プロセス・インスタンスのプロパティーの読み取りなどのリーダー権限が必要です。
IS_INLINE	ブール	このタスク・テンプレートがビジネス・プロセス内のタスクとしてモデル化されるかどうかを示します。
KIND	整数	このタスク・テンプレートから派生したタスクの種類。指定可能な値は、以下のとおりです。 **KIND_HUMAN** タスクが人の手で作成され、処理されることを明示します。 **KIND_ORIGINATING** 人によるコンピューターへのタスクの割り当てが可能であることを示します。この場合、人間が自動サービスを呼び出します。 **KIND_PARTICIPATING** サービス・コンポーネント (ビジネス・プロセスなど) により、人にタスクが割り当てられることを示します。 **KIND_ADMINISTRATIVE** タスクが管理用タスクであることを明示します。
NAME	ストリング	タスク・テンプレートの名前。
NAMESPACE	ストリング	タスク・テンプレートのカテゴリー化に使用されるネーム・スペース。
PRIORITY	整数	タスク・テンプレートの優先順位。

表 23. TASK_TEMPL ビュー内の列 (続き)

列名	タイプ	コメント
STATE	整数	タスク・テンプレートの状態。指定可能な値は、以下のとおりです。 **STATE_STARTED** タスク・テンプレートをタスク・インスタンスの作成に使用できることを明示します。 **STATE_STOPPED** タスク・テンプレートが停止されたことを明示します。この状態のタスク・テンプレートからタスク・インスタンスを作成することはできません。
SUPPORT_ AUTOCLAIM	ブール	このタスク・テンプレートから派生したタスクが 1 人のユーザーに割り当てられた場合、タスクを自動的に要求できるかどうかを示します。
SUPPORT_CLAIM_ SUSP	ブール	このタスク・テンプレートから派生したタスクが中断された場合、タスクを自動的に要求できるかどうかを示します。
SUPPORT_ DELEGATION	ブール	このタスク・テンプレートから派生したタスクが、作業項目の作成、削除、または転送を使用して作業代行をサポートするかどうかを示します。
TYPE	ストリング	タスク・テンプレートのカテゴリー化に使用されるタイプ。

TASK_TEMPL_CPROP ビュー:

この定義済みデータベース・ビューは、タスク・テンプレートのカスタム・プロパティーを照会するために使用します。

表 24. TASK_TEMPL_CPROP ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
NAME	ストリング	プロパティーの名前。
STRING_VALUE	ストリング	String 型のカスタム・プロパティーの値。

TASK_TEMPL_DESC ビュー:

この定義済みデータベース・ビューは、タスク・テンプレート・オブジェクトのマルチリンガル記述データを照会するために使用します。

表 25. TASK_TEMPL_DESC ビュー内の列

列名	タイプ	コメント
TKTID	ストリング	タスク・テンプレート ID。
LOCALE	ストリング	説明または表示名に関連付けられているロケールの名前。
DESCRIPTION	ストリング	タスク・テンプレートの説明。

表 25. TASK_TEMPL_DESC ビュー内の列 (続き)

列名	タイプ	コメント
DISPLAY_NAME	ストリング	タスク・テンプレートの記述名。

WORK_ITEM ビュー:

この定義済みデータベース・ビューは、作業項目の照会や、プロセス、タスクおよびエスカレーション用の許可データの照会に使用します。

表 26. WORK_ITEM ビュー内の列

列名	タイプ	コメント
WIID	ID	作業項目 ID。
OWNER_ID	ストリング	所有者のプリンシパル ID。
GROUP_NAME	ストリング	関連するグループ・ワーク・リストの名前。
EVERYBODY	ブール	この作業項目を全員が所有するかどうかを指定します。
OBJECT_TYPE	整数	関連オブジェクトのタイプ。指定可能な値は、以下のとおりです。 **OBJECT_TYPE_ACTIVITY** その作業項目がアクティビティー用に作成されたものであることを明示します。 **OBJECT_TYPE_PROCESS_INSTANCE** その作業項目がプロセス・インスタンス用に作成されたものであることを明示します。 **OBJECT_TYPE_TASK_INSTANCE** その作業項目がタスク用に作成されたものであることを明示します。 **OBJECT_TYPE_TASK_TEMPLATE** その作業項目がタスク・テンプレート用に作成されたものであることを明示します。 **OBJECT_TYPE_ESCALATION_ INSTANCE** その作業項目がエスカレーション・インスタンス用に作成されたものであることを明示します。 **OBJECT_TYPE_APPLICATION_ COMPONENT** その作業項目がアプリケーション・コンポーネント用に作成されたものであることを示します。
OBJECT_ID	ID	関連オブジェクト (例えば、関連プロセスやタスクなど) の ID。

表 26. *WORK_ITEM* ビュー内の列 (続き)

列名	タイプ	コメント
ASSOC_OBJECT_TYPE	整数	ASSOC_OID 属性によって参照されるオブジェクトのタイプ。例えば、タスク、プロセス、または外部オブジェクトなど。OBJECT_TYPE 属性の値を使用します。
ASSOC_OID	ID	作業項目のあるオブジェクト関連オブジェクトの ID。例えば、この作業項目が作成されたアクティビティー・インスタンスを含んでいるプロセス・インスタンスの、プロセス・インスタンス ID など。
REASON	整数	作業項目の割り当て理由。指定可能な値は、以下のとおりです。 REASON_POTENTIAL_STARTER REASON_POTENTIAL_INSTANCE_ CREATOR REASON_POTENTIAL_OWNER REASON_EDITOR REASON_READER REASON_ORIGINATOR REASON_OWNER REASON_STARTER REASON_ESCALATION_RECEIVER REASON_ADMINISTRATOR
CREATION_TIME	タイム・スタンプ	作業項目が作成された日時。

例外および障害の処理

障害は、プロセス・インスタンスが作成される、またはプロセス・インスタンスのナビゲーションのパーツとして呼び出される命令が失敗する場合に発生することがあります。これらの障害を処理する以下のような機構が存在します。

- 対応する障害ハンドラーへの制御の引き渡し
- プロセスの停止と状態の修復の依頼 (強制再試行、強制完了)
- プロセスの補正
- 障害の API 例外としてのクライアント・アプリケーションへの引き渡し (例えば、インスタンスの作成元のプロセス・モデルが存在しない場合に例外がスローされる)

障害および例外の処理は、以下のタスクで説明します。

API 例外の処理

BusinessFlowManagerService インターフェースまたは HumanTaskManagerService インターフェースのメソッドが正常に完了しない場合、エラーの原因を示す例外がスローされます。この例外を特別に処理して、呼び出し元にガイダンスを提供することができます。

ただし、例外のサブセットのみを特別に処理し、その他の潜在的な例外に対しては汎用のガイダンスを提供するのが一般的です。すべての固有の例外は、汎用の ProcessException または TaskException から継承しています。最終の catch(ProcessException) または catch(TaskException) ステートメントを使用して汎用の例外を catch することが*最良実例* です。このステートメントでは、発生する可能性があるその他すべての例外を考慮に入れるため、このステートメントによって、ご使用のアプリケーション・プログラムの上位互換性を確保することができます。

アクティビティーに設定された障害の検査

1. 失敗状態または停止状態のタスク・アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "(ACTIVITY.STATE = ACTIVITY.STATE.STATE_FAILED OR
                     ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED) AND
                     ACTIVITY.KIND=ACTIVITY.KIND.KIND_STAFF",
                     null, null, null);
```

   このアクションは、失敗または停止のアクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

   この障害名は、障害のあるキュー名のローカル・パーツです。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ClientObjectWrapper faultMessage = process.getFaultMessage(aiid);
  DataObject fault = null ;
  if ( faultMessage.getObject() != null && faultMessage.getObject()
       instanceof DataObject )
  {
    fault = (DataObject)faultMessage.getObject();
    Type type = fault.getType();
    String name = type.getName();
    String uri = type.getURI();
  }
}
```

   これは、障害名を戻します。また、障害名を取得する代わりに、停止アクティビティーの未処理の例外を分析することもできます。

停止した invoke アクティビティーで発生した障害の検査

アクティビティーで障害が発生した場合、障害タイプによってそのアクティビティーの修復のために実行できるアクションが決まります。

1. 停止状態の staff アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
                    null, null, null);
```

   このアクションは、停止された invoke アクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

これは、障害キュー名のローカル・パーツです。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);

  ProcessException excp = activity.getUnhandledException();
  if ( excp instanceof ApplicationFaultException )
  {
   ApplicationFaultException fault = (ApplicationFaultException)excp;
   String faultName = fault.getFaultName();
  }
}
```

JSF コンポーネントを使用した、ビジネス・プロセスおよびヒューマン・タスク用 Web アプリケーションの開発

Business Process Choreographer Explorer は、いくつかの JavaServer Faces (JSF) コンポーネントを提供します。これらのコンポーネントを拡張および統合して、ビジネス・プロセスおよびヒューマン・タスク機能を Web アプリケーションに追加することができます。

WebSphere Integration Developer を使用して Web アプリケーションを作成することができます。

1. 動的プロジェクトを作成し、Web プロジェクトの Web プロジェクト・フィーチャー・プロパティーを変更して Faces 基本コンポーネントを組み込みます。

   Web プロジェクトの作成の詳細については、WebSphere Integration Developer インフォメーション・センターにアクセスしてください。

2. 前提条件である Business Process Choreographer Explorer Java アーカイブ (JAR ファイル) を追加します。

   以下のファイルをプロジェクトの WEB-INF/lib ディレクトリーに追加してください。

   - bpcclientcore.jar
   - bfmclientmodel.jar
   - htmclientmodel.jar
   - bpcjsfcomponents.jar

   これらのファイルは *install_root*/ProcessChoreographer/client ディレクトリーにあります。

3. 必要な EJB 参照を、Web アプリケーション・デプロイメント記述子 web.xml ファイルに追加します。

```
<ejb-ref id="EjbRef_1">
  <ejb-ref-name>ejb/BusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
  <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
<ejb-ref id="EjbRef_2">
  <ejb-ref-name>ejb/HumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.task.api.HumanTaskManagerHome</home>
```

```
  <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
<ejb-local-ref id="EjbLocalRef_1">
  <ejb-ref-name>ejb/LocalBusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
  <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
</ejb-local-ref>
<ejb-local-ref id="EjbLocalRef_2">
  <ejb-ref-name>ejb/LocalHumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
  <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

4. Business Process Choreographer Explorer JSF コンポーネントを JSF アプリケーションに追加します。
   a. アプリケーションに必要なタグ・ライブラリーを JavaServer Pages (JSP) ファイルに追加します。 通常、JSF および HTML タグ・ライブラリーと、JSF コンポーネントに必要とされるタグ・ライブラリーが必要です。
      - `<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>`
      - `<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>`
      - `<%@ taglib uri="http://com.ibm.bpe.jsf/taglib" prefix="bpe" %>`

   b. JSP ページの本体に `<f:view>` タグを追加し、`<f:view>` タグに `<h:form>` タグを追加します。
   c. JSP ファイルに JSF コンポーネントを追加します。

      アプリケーションに応じて、List コンポーネント、Details コンポーネント、CommandBar コンポーネント、または Message コンポーネントを JSP ファイル内に追加します。各コンポーネントの複数のインスタンスを追加することができます。
   d. JSF 構成ファイル内で管理対象 Bean を構成します。

      デフォルトでは、構成ファイルは faces-config.xml ファイルです。このファイルは、Web アプリケーションの WEB-INF ディレクトリーにあります。 JSP ファイルに追加するコンポーネントに応じて、照会およびその他のラッパー・オブジェクトへの参照を JSF 構成ファイルに追加する必要もあります。
   e. JSF コンポーネントをサポートするために必要なカスタム・コードをインプリメントします。
5. アプリケーションをデプロイします。

   EJB 参照を Java Naming and Directory Interface (JNDI) 名へマップするか、参照を ibm-web-bnd.xmi ファイルへ手動で追加します。以下の表に、参照バインディングおよびそのデフォルト・マッピングを示します。

表 27. 参照バインディングから JNDI 名へのマッピング

参照バインディング	JNDI 名	コメント
ejb/BusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	リモート・セッション Bean
ejb/LocalBusinessProcessHome	com/ibm/bpe/api/BusinessFlowManagerHome	ローカル・セッション Bean
ejb/HumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	リモート・セッション Bean
ejb/LocalHumanTaskManagerEJB	com/ibm/task/api/HumanTaskManagerHome	ローカル・セッション Bean

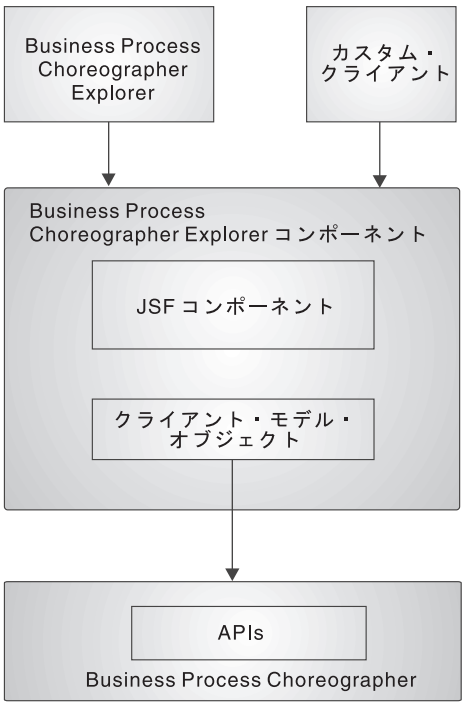
デプロイした Web アプリケーションには、Business Process Choreographer Explorer コンポーネントが提供する機能が含まれています。

Business Process Choreographer Explorer コンポーネント

Business Process Choreographer Explorer コンポーネントは、JavaServer Faces (JSF) テクノロジーに基づく構成可能かつ再使用可能なエレメントの集合です。

これらのエレメントを Web アプリケーションに組み込むことができます。これにより、これらのアプリケーションは、インストール済みビジネス・プロセスおよびヒューマン・タスク・アプリケーションにアクセスできるようになります。

コンポーネントは、JSF コンポーネントのセットおよびクライアント・モデル・オブジェクトのセットで構成されています。コンポーネントから Business Process Choreographer、Business Process Choreographer Explorer、およびその他のカスタム・クライアントへの関係を、次の図に示します。



JSF コンポーネント

Business Process Choreographer Explorer コンポーネントには、以下の JSF コンポーネントが含まれます。ビジネス・プロセスおよびヒューマン・タスクを操作するための Web アプリケーションをビルドするときに、これらの JSF コンポーネントを JavaServer Pages (JSP) ファイルに組み込みます。

- List コンポーネント

  List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、またはエスカレーションなどの、アプリケーション・オブジェクトのリストをテーブル内に表示します。このコンポーネントには、関連付けられたリスト・ハンドラーがあります。
- Details コンポーネント

Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。このコンポーネントには、関連付けられた詳細ハンドラーがあります。

- CommandBar コンポーネント

  CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトに作動するコマンドを表します。これらのオブジェクトは、リスト・ハンドラーまたは詳細ハンドラーによって提供されます。

- Message コンポーネント

  Message コンポーネントは、サービス・データ・オブジェクト (SDO) または単純型のいずれかを含むことのできるメッセージを表示します。

クライアント・モデル・オブジェクト

クライアント・モデル・オブジェクトは、JSF コンポーネントと共に使用されます。これらのオブジェクトは、基盤となる Business Process Choreographer API のインターフェースの一部をインプリメントし、元のオブジェクトをラップします。クライアント・モデル・オブジェクトは、ラベルの各国語サポートと、一部のプロパティーのコンバーターを提供します。

List コンポーネントでのリスト処理:

List コンポーネントのインスタンスはすべて com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスに関連しています。

このリスト・ハンドラーは、関連するリスト内でどの項目が選択されているかをトラッキングし、通知メカニズムを提供します。リスト・ハンドラーは、`bpe:list` タグのモデル属性を介して List コンポーネントにバインドされます。

リスト・ハンドラーの通知メカニズムは、 com.ibm.bpe.jsf.handler.ItemListener インターフェースを使用してインプリメントされます。 Business Process Choreographer Explorer はこの通知メカニズムを使用して、さまざまな種類の項目用の詳細ページにリスト項目を関連付けます。通知イベントのトリガーは、通常、現在のリストで表示される項目のプロパティーの 1 つです。

通知メカニズムを活用するには、プロパティーの `bpe:column` タグのアクション属性の値を、通知イベントがトリガーされたときにアプリケーションが動作を続行する JSF ナビゲーション・ターゲットに設定します。 List コンポーネントは、列内の項目を JSF コマンド・リンクとしてレンダリングします。リンクがトリガーされると、リスト内の項目を表すオブジェクトが判別され、すべての登録済み項目リスナーに渡されます。このインターフェースのインプリメンテーションは、JavaServer Faces (JSF) アプリケーションの構成ファイルに登録できます。

BPCListHandler クラスは、refreshList メソッドを提供します。このメソッドを JSF メソッド・バインディングで使用して、照会を再実行するためのユーザー・インターフェース制御をインプリメントすることができます。

照会のインプリメンテーション

リスト・ハンドラーを使用すると、すべての種類のオブジェクトおよびそれらのプロパティーを表示できます。表示されるリストの内容は、リスト・ハンドラー用に構成された com.ibm.bpc.clientcore.Query インターフェースのインプリメンテーションによって戻されるオブジェクトのリストによって異なります。照会は、BPCListHandler クラスの setQuery メソッドを使用してプログラマチックに設定することもできますし、アプリケーションの JSF 構成ファイルで構成することもできます。

照会は、Business Process Choreographer API に対してだけでなく、コンテンツ管理システムやデータベースなど、アプリケーションからアクセスできるその他の情報ソースに対しても実行できます。要件は、照会の結果が execute メソッドでオブジェクトの `java.util.List` として戻されることだけです。

戻されるオブジェクトのタイプは、照会が定義されたリストの列に表示されるすべてのプロパティーに対して適切な getter メソッドを使用できることを保証する必要があります。戻されるオブジェクトのタイプがリスト定義に適合することを確認するには、 faces 構成ファイルで定義されている BPCListHandler インスタンス上のタイプ・プロパティーの値を、戻されるオブジェクトの完全修飾クラス名に設定します。この名前は、照会インプリメンテーションの getType 呼び出しで戻すことができます。実行時に、リスト・ハンドラーはオブジェクト・タイプが定義に準拠していることを確認します。

リスト内の特定の項目にエラー・メッセージをマップするには、照会によって戻されたオブジェクトが署名 `public Object getID()` でメソッドをインプリメントする必要があります。

エラー処理

次のエラー状態では、BPCListHandler クラスが提供するエラー処理機能を利用できます。

- 照会の実行時またはコマンドの実行時に発生するエラー

  照会の実行中にエラーが発生した場合、 BPCListHandler クラスは、不十分なアクセス権限によるエラーとその他の例外とを区別します。不十分なアクセス権限によるエラーをキャッチするには、照会の実行メソッドによってスローされる ClientException の rootCause パラメーターが com.ibm.bpe.api.EngineNotAuthorizedException または com.ibm.task.api.NotAuthorizedException 例外である必要があります。 List コンポーネントは、照会の結果の代わりにエラー・メッセージを表示します。

  エラーが不十分なアクセス権限によるものでない場合、 BPCListHandler クラスは例外オブジェクトを、 JSF アプリケーション構成ファイルの BPCError キーで定義した com.ibm.bpc.clientcore.util.ErrorBean インターフェースのインプリメンテーションに渡します。例外が設定されている場合は、エラー・ナビゲーション・ターゲットが呼び出されます。
- リストに表示される項目の処理時に発生するエラー

BPCListHandler クラスは、 com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントします。利用者は、setErrors メソッドでタイプ `java.util.Map` のマップ・パラメーターを使用して、これらのエラーに関する情報を提供することができます。このマップには、キーとして ID が、値として例外が含まれています。 ID は、エラーの原因となったオブジェクトの getID メソッドによって戻された値である必要があります。リストが再びレンダリングされた場合は、リスト・ハンドラーによって、修飾リスト項目用のエラー・メッセージが別の列に表示されます。マップが設定されていて、リスト内に表示されている項目のいずれかに ID のいずれかが一致する場合は、リスト・ハンドラーによって、エラー・メッセージを含む列が自動的にリストに追加されます。

リスト内のエラー・メッセージが古くなるのを避けるため、エラー・マップをリセットしてください。次の状況では、マップは自動的にリセットされます。

- refreshList メソッドの BPCListHandler クラスが呼び出される。
- BPCListHandler クラスで新規照会が設定されている。
- CommandBar コンポーネントを使用して、リストの項目でアクションがトリガーされている。 CommandBar コンポーネントは、エラー処理のメソッドの 1 つとしてこのメカニズムを使用します。

CommandBar コンポーネント:

CommandBar コンポーネントを使用して、アプリケーション内のアクション・ボタンを統合します。コンポーネントは、ユーザー・インターフェースでのアクション用のボタンを作成し、ボタンがクリックされたときに作成されるイベントを処理します。

これらのボタンは、BPCListHandler クラスや BPCDetailsHandler クラスなど、com.ibm.bpe.jsf.handler.ItemProvider インターフェースによって戻されるオブジェクトで動作する機能を起動します。 CommandBar コンポーネントは、 `bpe:commandbar` タグでモデル属性の値によって定義された項目プロバイダーを使用します。

コマンドの処理方法

アプリケーションのユーザー・インターフェースのコマンド・バー・セクションにあるボタンをクリックすると、関連するイベントが CommandBar コンポーネントによって次のように処理されます。

1. CommandBar コンポーネントは、イベントを生成したボタンに対して指定された com.ibm.bpc.clientcore.Command インターフェースのインプリメンテーションを示します。
2. CommandBar コンポーネントに関連するモデルが com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントすると、前のイベントからのエラー・メッセージを削除するため、clearErrorMap メソッドが呼び出されます。
3. ItemProvider インターフェースの getSelectedItems メソッドが呼び出されます。戻された項目のリストは、コマンドの execute メソッドに渡され、コマンドが呼び出されます。
4. CommandBar コンポーネントは、 JavaServer Faces (JSF) ナビゲーション・ターゲットを決定します。 `bpe:commandbar` タグでアクション属性が指定されていな

い場合は、 execute メソッドの戻り値によってナビゲーション・ターゲットが指定されます。アクション属性が JSF メソッド・バインディングに設定されている場合は、メソッドによって戻されたストリングがナビゲーション・ターゲットと解釈されます。アクション属性は、明示的なナビゲーション・ターゲットも指定します。

エラー処理

コマンドがトリガーされるのは、次の条件のいずれかが真である場合だけです。

- 例外がスローされない。
- 例外がスローされる場合は、ErrorsInCommandException 例外である。

CommandBar コンポーネントでエラー処理をインプリメントする方法をいくつか示します。

- CommandBar コンポーネントの機能を使用しないことが可能です。例えば、選択したコマンドに固有のページにエラーを表示する場合は、コマンドのインプリメンテーションによって、発生した例外をキャッチし、エラー・ページに使用するページ Bean にそれを伝搬することができます。 `bpe:commandbar` タグのコンテキスト属性を使用して、ページ Bean をコマンド・インプリメンテーションに使用できるようにすることができます。ページ Bean で例外が設定された後、コマンドは、エラー・ページ用に定義された JSF ナビゲーション・ルールのストリングを戻します。
- ユーザー・インターフェースのコマンド・バー・セクションの下にエラー・メッセージを表示する場合は、com.ibm.bpc.clientcore.exception.CommandBarMessage マーク文字インターフェースをインプリメントする例外クラスを作成します。このインターフェースは、エラー・メッセージのメッセージ・カタログを提供します。
- コマンドが項目のリストで動作する場合は、リスト内の各項目についてコマンドの成功をトラッキングすることができます。エラーをトラッキングするには、操作が失敗した項目に各例外をマップします。 CommandBar コンポーネントは、キーとしての ID と値としての例外を含むマップを、 CommandBar コンポーネントに対して定義されたモデル・オブジェクトに渡すことができます。

  このメカニズムを機能させるには、モデル・オブジェクトが com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントし、コマンドが com.ibm.bpc.clientcore.exception.ErrorsInCommandException 例外をスローする必要があります。次に、CommandBar コンポーネントは、例外に含まれているマップをエラー・ハンドラーに渡します。エラーは発生しましたが、このアクション・メソッドがトリガーされ、現在のビューが更新されます。 Business Process Choreographer Explorer アプリケーションは、このメソッドを利用して、リストで例外を表示します。
- CommandBarMessage インターフェースをインプリメントしない ClientException 例外をスローし、例外が ErrorsInCommandException ではない場合、 CommandBar コンポーネントは、アプリケーションの構成ファイルで定義された BPCError エラー Bean に例外を伝搬します。エラー・ナビゲーション・ターゲットでエラー処理が続けられます。

Business Process Choreographer Explorer JSF コンポーネントによって提供されるユーティリティー:

JavaServer Faces (JSF) コンポーネントは、ユーザー指定の時間帯情報およびエラー処理のためのユーティリティーを提供します。

ユーザー固有の時間帯情報

BPCListHandler クラスは、 com.ibm.bpc.clientcore.util.User インターフェースを使用して、各ユーザーの時間帯およびロケールに関する情報を取得します。 List コンポーネントは、JavaServer Faces (JSF) 構成ファイルでインターフェースのインプリメンテーションが **user** を管理対象 Bean 名として設定されていると予期します。構成ファイル内でこの項目が欠けている場合は、 WebSphere Process Server が動作している時間帯が戻されます。

com.ibm.bpc.clientcore.util.User インターフェースは次のように定義されています。

```
public interface User {

    /**
     * The locale used by the client of the user.
     * @return Locale.
     */
    public Locale getLocale();
   /**
    * The time zone used by the client of the user.
    * @return TimeZone.
    */
    public TimeZone getTimeZone();

   /**
    * The name of the user.
    * @return name of the user.
    */
    public String getName();
}
```

エラー処理用の ErrorBean インターフェース

JSF コンポーネントが、事前定義された管理対象 Bean の BPCError をエラー処理に使用することがあります。この Bean は、com.ibm.bpc.clientcore.util.ErrorBean インターフェースをインプリメントします。エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。エラー・ページが表示されるのは、次のような状態のときです。

- リスト・ハンドラー用に定義された照会の実行中にエラーが発生し、エラーがコマンドの execute メソッドによって ClientException エラーとしてスローされた場合
- ClientException エラーがコマンドの execute メソッドによってスローされ、このエラーが ErrorsInCommandException エラーではなく、 CommandBarMessage インターフェースのインプリメントもしない場合
- コンポーネント内でエラー・メッセージが表示され、メッセージのハイパーリンクに従っている場合

com.ibm.bpc.clientcore.util.ErrorBeanImpl インターフェースのデフォルト・インプリメンテーションを使用できます。

インターフェースは次のように定義されます。

```
public interface ErrorBean {

    public void setException(Exception ex);

    /*
     * This setter method call allows a locale and
     * the exception to be passed. This allows the
     * getExceptionMessage methods to return localized Strings
     *
     */
    public void setException(Exception ex, Locale locale);

    public Exception getException();
    public String getStack();
    public String getNestedExceptionMessage();
    public String getNestedExceptionStack();
    public String getRootExceptionMessage();
    public String getRootExceptionStack();

    /*
     * This method returns the exception message
     * concatenated recursively with the messages of all
     * the nested exceptions.
     */
    public String getAllExceptionMessages();

    /*
     * This method is returns the exception stack
     * concatenated recursively with the stacks of all
     * the nested exceptions.
     */
    public String getAllExceptionStacks();
}
```

JSF アプリケーションへの List コンポーネントの追加

Business Process Choreographer Explorer List コンポーネントを使用して、例えばビジネス・プロセス・インスタンスやタスク・インスタンスなどの、クライアント・モデル・オブジェクトのリストを表示します。

1. List コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:list タグを h:form タグに追加します。 bpe:list タグには、モデル属性が含まれていなければなりません。bpe:column タグを bpe:list に追加して、リスト内の各行に表示されるオブジェクトのプロパティーを追加します。

   以下の例では、List コンポーネントを追加してタスク・インスタンスを表示する方法を示します。

   ```
   <h:form>

      <bpe:list model="#{TaskPool}">
         <bpe:column name="name" action="taskInstanceDetails" />
         <bpe:column name="state" />
         <bpe:column name="kind" />
         <bpe:column name="owner" />
         <bpe:column name="originator" />
      </bpe:list>

   </h:form>
   ```

モデル属性は、TaskPool という管理対象 Bean を参照します。管理対象 Bean は、リストが操作を繰り返す対象となる Java オブジェクトのリストを提供し、次に個々の行を表示します。

2. bpe:list タグで参照されている管理対象 Bean を構成します。

   List コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスでなければなりません。

   以下の例では、TaskPool 管理対象 Bean を構成ファイルに追加する方法を示します。

```
<managed-bean>
<managed-bean-name>TaskPool</managed-bean-name>
<managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>

   <managed-property>
      <property-name>query</property-name>
      <value>#{TaskPoolQuery}</value>
   </managed-property>

   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>

</managed-bean>
<managed-bean>
<managed-bean-name>htmConnection</managed-bean-name>
<managed-bean-class>com.ibm.task.clientmodel.HTMConnection</managed-bean-class>
<managed-bean-scope>application</managed-bean-scope>

   <managed-property>
      <property-name>jndiName</property-name>
      <value>java:comp/env/ejb/LocalHumanTaskManagerEJB</value>
   </managed-property>
```

   例では、照会およびタイプの 2 つの構成可能プロパティーが TaskPool に含まれていることを示しています。照会プロパティーの値は、TaskPoolQuery という別の管理対象 Bean を参照しています。タイプ・プロパティーの値は、Bean クラスを指定します。そのクラスのプロパティーは、表示されたリストの列に示されます。関連する照会インスタンスは、プロパティー型を持つことも可能です。プロパティー型が指定された場合、それはリスト・ハンドラーに指定された型と同一でなければなりません。

   TaskPool 管理対象 Bean は、Human Task Manager への接続を提供するため、htmConnection 管理対象 Bean を使用してインプリメントされます。

3. リスト・ハンドラーによって参照される管理対象 Bean 用のカスタム・コードを追加します。

   以下の例では、TaskPool 管理対象 Bean 用のカスタム・コードを追加する方法を示します。

```
public class MyTaskQuery implements Query {

  public List execute throws ClientException {

    // Examine the faces-config file for a managed bean "htmConnection".
```

```
    //
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application  app = ctx.getApplication();
    ValueBinding htmVb = app.createValueBinding("#{htmConnection}");
    htmConnection = (HTMConnection) htmVb.getValue(ctx);
    HumanTaskManagerService taskService =
         htmConnection.getHumanTaskManagerService();

    // Then call the actual query method on the Human Task Manager service.
    //
    QueryResultSet queryResult = taskService.query(
     "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
      + "TASK.STARTED, TASK.ACTIVATED, TASK.DUE, TASK.EXPIRES, TASK.PRIORITY" ,
      "TASK.KIND IN(101,102,105) AND TASK.STATE IN(2)
        AND WORK_ITEM.REASON IN (1)",
      null,
      null,
      null);
    List applicationObjects = transformToTaskList ( queryResult );
    return applicationObjects ;
  }


   private List transformToTaskList(QueryResultSet result) {

ArrayList array = null;
int entries = result.size();
array = new ArrayList( entries );

// Transforms each row in the QueryResultSet to a task instance beans.
 for (int i = 0; i < entries; i++) {
   result.next();
   array.add( new TaskInstanceBean( result, connection ));
 }
 return array ;
  }
}
```

TaskPoolQuery Bean は、Java オブジェクトのプロパティーを照会します。この Bean は、com.ibm.bpc.clientcore.Query インターフェースをインプリメントする必要があります。リスト・ハンドラーは、内容を最新表示するときに、照会の実行メソッドを呼び出します。呼び出しによって、Java オブジェクトのリストが戻されます。getType メソッドは、戻された Java オブジェクトのクラス名を戻す必要があります。

これで、JSF アプリケーションは、例えば状態、種類、所有者、ユーザーが使用可能なタスク・インスタンスのオリジネーターなどの、要求されたオブジェクトのリストのプロパティーを表示する JavaServer ページを含むようになります。

List コンポーネント: タグ定義:

Business Process Choreographer Explorer List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、およびエスカレーションなどの、オブジェクトのリストをテーブル内に表示します。

List コンポーネントは、JSF コンポーネント・タグである `bpe:list` および `bpe:column` から構成されます。 `bpe:column` タグは、`bpe:list` タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.ListComponent

構文例

```
<bpe:list model="#{ProcessTemplateList}">
          rows="20"
          styleClass="list"
          headerStyleClass="listHeader"
          rowClasses="normal">

    <bpe:column name="name" action="processTemplateDetails"/>
    <bpe:column name="validFromTime"/>
    <bpe:column name="executionMode" label="Execution mode"/>
    <bpe:column name="state" converterID="my.state.converter"/>
    <bpe:column name="autoDelete"/>
    <bpe:column name="description"/>

</bpe:list>
```

タグ属性

bpe:list タグの本体には、bpe:column タグのみを含めることができます。テーブルがレンダリングされるとき、List コンポーネントは、アプリケーション・オブジェクトのリストについて処理を繰り返し、列ごとに特定のオブジェクトを提供します。

表 28. bpe:list 属性

属性	必須	説明
model	はい	com.ibm.bpe.jsf.handler.BPCListHandler クラスの管理対象 Bean 用の値バインディング。
styleClass	いいえ	タイトル、行、およびページ送りボタンを含むテーブル全体のレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル。
headerStyleClass	いいえ	テーブル・ヘッダーのレンダリング用の CSS スタイル・クラス。
cellStyleClass	いいえ	個々のテーブル・セルのレンダリング用の CSS スタイル・クラス。
buttonStyleClass	いいえ	フッター領域内のボタンのレンダリング用の CSS スタイル・クラス。
rowClasses	いいえ	テーブル内の行のレンダリング用の CSS スタイル・クラス。
rows	いいえ	ページに表示される行数。項目数が行数を超える場合は、テーブルの最後にページ送りボタンが表示されます。
checkbox	いいえ	複数の項目を選択するためのチェック・ボックスを提供するかどうかを決定します。この属性に使用される値は true または false です。

表 29. bpe:column 属性

属性	必須	説明
name	はい	この列に表示されるオブジェクト・プロパティーの名前。この名前は、対応するクライアント・モデル・クラスで定義されているように、名前付きプロパティーに対応していなければなりません。
action	いいえ	この属性は、結果ストリングとして指定された場合、JavaServer Faces (JSF) ナビゲーション・ハンドラーによって使用される結果を定義して、次のページを決定します。 この属性がメソッド・バインディング (#{.....}) として指定された場合、呼び出されるメソッドのシグニチャーは `String method()` であり、その戻り値は、JSF ナビゲーション・ハンドラーによって次のページを決定するために使用されます。
label	いいえ	列のヘッダー内またはテーブル・ヘッダー行のセル内に表示されるラベル。この属性が設定されていない場合、クライアント・モデル・クラスによってデフォルト・ラベルが提供されます。
converterID	いいえ	JSF 構成ファイルでコンバーターを登録するために使用される ID。コンバーター ID が指定されない場合、リストに表示されるオブジェクトのインプリメンテーションには、現在のプロパティー用のコンバーターの定義が含まれます。List コンポーネントはこのコンバーターを使用します。

JSF アプリケーションへの Details コンポーネントの追加

Business Process Choreographer Explorer Details コンポーネントを使用して、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

1. Details コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   `bpe:details` タグを `<h:form>` タグに追加します。 `bpe:details` タグには、モデル属性が含まれていなければなりません。`bpe:property` タグを使用して Details コンポーネントにプロパティーを追加することができます。 Details コンポーネントにプロパティーが含まれていない場合は、オブジェクトのすべてのプロパティーが表示されます。

   以下の例では、Details コンポーネントを追加して、タスク・インスタンスのプロパティーのいくつかを表示する方法を示します。

```
<h:form>

   <bpe:details model="#{TaskInstanceDetails}">
      <bpe:property name="displayName" />
```

```
            <bpe:property name="owner" />
            <bpe:property name="kind" />
            <bpe:property name="state" />
            <bpe:property name="escalated" />
            <bpe:property name="suspended" />
            <bpe:property name="originator" />
            <bpe:property name="activationTime" />
            <bpe:property name="expirationTime" />
         </bpe:details>

      </h:form>
```

モデル属性は、TaskInstanceDetails という管理対象 Bean を参照します。Bean は、Java オブジェクトのプロパティーを提供します。

2. bpe:details タグで参照されている管理対象 Bean を構成します。

   Details コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスのインスタンスでなければなりません。このハンドラー・クラスは、Java オブジェクトをラップし、そのパブリック・プロパティーを Details コンポーネントに公開します。

   以下の例では、TaskInstanceDetails 管理対象 Bean を構成ファイルに追加する方法を示します。

```
<managed-bean>
   <managed-bean-name>TaskInstanceDetails</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.jsf.handler.BPCDetailsHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>
```

   例では、TaskInstanceDetails Bean に構成可能な type プロパティーが含まれることを示しています。タイプ・プロパティーの値は、Bean クラス (com.ibm.task.clientmodel.bean.TaskInstanceBean) を指定します。そのクラスのプロパティーは、表示された詳細の行に示されます。

これで、JSF アプリケーションは、例えばタスク・インスタンスの詳細などの、指定されたオブジェクトの詳細を表示する JavaServer ページを含むようになります。

Details コンポーネント: タグ定義:

Business Process Choreographer Explorer Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

Details コンポーネントは、JSF コンポーネント・タグである bpe:details および bpe:property から構成されます。 bpe:property タグは、 bpe:details タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.DetailsComponent

構文例

```
<bpe:details model="#{MyActivityDetails}">
    <bpe:property name="name"/>
    <bpe:property name="owner"/>
    <bpe:property name="activated"/>
</bpe:details>
```

```
<bpe:details model="#{MyActivityDetails}" style="style" styleClass="cssStyle">
             style="style"
             styleClass="cssStyle"
</bpe:details>
```

タグ属性

bpe:property タグを使用して、表示される属性のサブセットおよびこれらの属性が表示される順序の両方を指定します。詳細タグに属性タブが含まれていない場合、モデル・オブジェクトの使用可能な属性がすべてレンダリングされます。

表 30. bpe:details 属性

属性	必須	説明
model	はい	com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスの管理対象 Bean 用の値バインディング。
styleClass	いいえ	HTML エレメントのレンダリング用の、カスケーディング・スタイル・シート・スタイル・クラス。
columnClasses	いいえ	列のレンダリング用の、コンマで区切られた CSS スタイルのリスト。
rowClasses	いいえ	行のレンダリング用の、コンマで区切られた CSS スタイルのリスト。

表 31. bpe:property 属性

属性	必須	説明
name	はい	表示されるプロパティーの名前。この名前は、対応するクライアント・モデル・クラスで定義されているように、名前付きプロパティーに対応していなければなりません。
label	いいえ	プロパティーのラベル。この属性が設定されていない場合、クライアント・モデル・クラスによってデフォルト・ラベルが提供されます。
converterID	いいえ	JavaServer Faces (JSF) 構成ファイルでコンバーターを登録するために使用される ID。

JSF アプリケーションへの CommandBar コンポーネントの追加

Business Process Choreographer Explorer CommandBar コンポーネントを使用して、ボタンを含むバーを表示します。これらのボタンは、オブジェクトの詳細ビューまたはリスト内の選択されたオブジェクトで作動するコマンドを表します。

ユーザーがユーザー・インターフェースのボタンをクリックすると、対応するコマンドが選択されたオブジェクトで実行されます。JSF アプリケーション内の CommandBar コンポーネントを追加および拡張することができます。

1. CommandBar コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:commandbar タグを <h:form> タグに追加します。 bpe:commandbar タグには、モデル属性が含まれていなければなりません。

   以下の例では、CommandBar コンポーネントを追加して、タスク・インスタンスのプロパティーのいくつかを表示する方法を示します。

   ```
   <h:form>

      <bpe:commandbar model="#{TaskInstanceList}">
         <bpe:command commandID="Refresh" >
                      action="#{TaskInstanceList.refreshList}"
                      label="Refresh"/>

         <bpe:command commandID="MyClaimCommand" >
                      label="Claim" >
                      commandClass="<customcode>"/>
      </bpe:commandbar>

   </h:form>
   ```

   model 属性は、管理対象 Bean を参照します。この Bean は、ItemProvider インターフェースをインプリメントし、選択された Java オブジェクトを提供する必要があります。CommandBar コンポーネントは、通常、同一の JSP ファイル内の List コンポーネントまたは Details コンポーネントのいずれかと共に使用されます。一般に、タグで指定されたモデルは、同一ページの List コンポーネントまたは Details コンポーネントで指定されたモデルと同じです。そのため、例えば List コンポーネントの場合、コマンドはリスト内の選択された項目に対して作動します。

   この例では、 model 属性は TaskInstanceList 管理対象 Bean を参照します。この Bean は、Java オブジェクトのプロパティーを提供しており、ItemProvider インターフェースをインプリメントしなければなりません。このインターフェースは、BPCListHandler クラスおよび BPCDetailsHandler クラスによってインプリメントされます。また、カスタム要求コマンドも含まれています。

2. **オプション:** bpe:commandbar タグで参照されている管理対象 Bean を構成します。

   CommandBar モデル属性が、例えばリスト・ハンドラーまたは詳細ハンドラー用に既に構成済みの管理対象 Bean を参照する場合、それ以上の構成は必要ありません。これらのハンドラーのいずれかの構成を変更した場合、または異なる管理対象 Bean を使用した場合は、ItemProvider インターフェースをインプリメントする管理対象 Bean を JSF 構成ファイルに追加してください。

3. カスタム・コマンドをインプリメントするコードを JSF アプリケーションに追加します。

   以下のコード断片は、コマンド・バーを拡張するコマンド・クラスの作成方法を示します。このコマンド・クラス (MyClaimCommand) は、JSP ファイル内の bpe:command タグで参照されます。

このコマンドは、正しい数の項目が選択されていることなどの前提条件を検査します。次に、ヒューマン・タスク API の HumanTaskManagerService への参照を検索します。コマンドは、選択されたオブジェクトに対して操作を繰り返し、それらの処理を試みます。HumanTaskManagerService API を通じて、ID によってタスクが要求されます。例外が発生しなければ、対応する TaskInstanceBean オブジェクトの状態が更新されます。このアクションでは、オブジェクトの値を再度サーバーから検索することが避けられます。

```
public class MyClaimCommand implements Command {

  public String execute(List selectedObjects) throws ClientException {
     if( selectedObjects != null && selectedObjects.size() > 0 ) {
        try {
         // Determine HumanTaskManagerService from an HTMConnection bean.
         // Configure the bean in the faces-config.xml for easy access
         // in the JSF application.
         FacesContext ctx = FacesContext.getCurrentInstance();
         ValueBinding vb =
           ctx.getApplication().createValueBinding("{htmConnection}");
         HTMConnection htmConnection = (HTMConnection) htmVB.getValue(ctx);
         HumanTaskManagerService htm =
            htmConnection.getHumanTaskManagerService();

         Iterator iter = selectedObjects.iterator() ;
         while( iter.hasNext() ) {
           try {
                TaskInstanceBean task = (TaskInstanceBean) iter.next() ;
                TKIID tiid = task.getID() ;

                htm.claim( tiid ) ;
                task.setState( new Integer(TaskInstanceBean.STATE_CLAIMED ) ) ;

           }
           catch( Exception e ) {
             ;   // Error while iterating or claiming task instance.
                  // Ignore for better understanding of the sample.
           }
         }
       }
       catch( Exception e ) {
         ;  // Configuration or communication error.
           // Ignore for better understanding of the sample
       }
     }
     return null;
  }

 // Default implementations
 public boolean isMultiSelectEnabled() { return false; }
 public boolean[] isApplicable(List itemsOnList) {return null; }
 public void setContext(Object targetModel) {; // Not used here }
}
```

コマンドは以下のように処理されます。

a. ユーザーがコマンド・バーの対応するボタンをクリックすると、コマンドが起動されます。 CommandBar コンポーネントは、model 属性で指定された項目プロバイダーから選択された項目を検索し、選択されたオブジェクトのリストを commandClass インスタンスの execute メソッドに渡します。

b. commandClass 属性は、コマンド・インターフェースをインプリメントするカスタム・コマンド・インプリメンテーションを参照します。コマンドは、public String execute(List selectedObjects)throws ClientException メソッドをイ

ンプリメントしていなければなりません。コマンドが戻した結果は、JSF アプリケーションの次のナビゲーション規則を決定するために使用されます。

c. コマンドの完了後、CommandBar コンポーネントは action 属性を評価します。 action 属性は、静的ストリングである場合も、public String Method() というシグニチャーの JSF アクション・メソッドへのメソッド・バインディングである場合もあります。 action 属性を使用して、コマンド・クラスの結果をオーバーライドするか、またはナビゲーション規則の結果を明示的に指定します。コマンドが ErrorsInCommandException 例外以外の例外をスローした場合、 action 属性は処理されません。

これで、JSF アプリケーションは、カスタマイズされたコマンド・バーをインプリメントする JavaServer ページを含むようになります。

CommandBar コンポーネント: タグ定義:

Business Process Choreographer Explorer CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトに作動します。

CommandBar コンポーネントは、JSF コンポーネント・タグである `bpe:commandbar` および `bpe:command` から構成されます。 `bpe:command` タグは、 `bpe:commandbar` タグのサブエレメントです。

コンポーネント・クラス

com.ibm.bpe.jsf.component.CommandBarComponent

構文例

```
<bpe:commandbar model="#{TaskInstanceList}">

    <bpe:command
       commandID="Work on"
       label="Work on..."
       commandClass="com.ibm.bpc.explorer.command.WorkOnTaskCommand"
       context="#{TaskInstanceDetailsBean}"/>

    <bpe:command
       commandID="Cancel"
       label="Cancel"
       commandClass="com.ibm.task.clientmodel.command.CancelClaimTaskCommand"
       context="#{TaskInstanceList}"/>

</bpe:commandbar>
```

タグ属性

表 32. bpe:commandbar 属性

属性	必須	説明
model	はい	ItemProvider インターフェースをインプリメントする管理対象 Bean に対する値バインディング式。通常、この管理対象 Bean は、同一の JavaServer Pages (JSP) ファイル内の List コンポーネントまたは Details コンポーネントによって CommandBar コンポーネントとして使用される com.ibm.bpe.jsf.handler.BPCListHandler クラスまたは com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスです。
styleClass	いいえ	バーのレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル。
buttonStyleClass	いいえ	コマンド・バー内のボタンのレンダリング用の CSS スタイル。

表 33. bpe:command 属性

属性	必須	説明
commandID	はい	コマンドの ID。
commandClass	はい	起動されるコマンド・クラス。
action	いいえ	シグニチャーが `String method()` である JavaServer Faces (JSF) アクション・メソッド。このアクション・メソッドによって戻される値、または直接リテラルとして指定された値が、コマンドの execute メソッドによって戻されるターゲットをオーバーライドします。コマンドが ErrorsInCommandException 例外以外の例外をスローした場合、 action 属性は処理されません。 この属性は、結果ストリングとして指定された場合、JSF ナビゲーション・ハンドラーによって使用される結果を定義して、ナビゲーション規則と次に表示するページを決定します。 この属性がメソッド・バインディング (#{.....}) として指定された場合、呼び出されるメソッドのシグニチャーは `String method()` です。その戻り値が JSF ナビゲーション・ハンドラーによって使用され、ナビゲーション規則と次に表示するページを決定します。
label	はい	コマンド・バーでレンダリングされるボタンのラベル。

表 33. bpe:command 属性 (続き)

属性	必須	説明
styleClass	いいえ	ボタンのレンダリング用の CSS スタイル。このスタイルは、コマンド・バーに定義されたボタン・スタイルをオーバーライドします。
context	いいえ	管理対象 Bean を参照する値バインディング式。コマンドがターゲット・ページまたは Bean を初期化する必要がある場合に、この属性を使用します。

JSF アプリケーションへの Message コンポーネントの追加

Business Process Choreographer Explorer Message コンポーネントを使用して、JavaServer Faces (JSF) アプリケーション内で、データ・オブジェクトおよびプリミティブ型をレンダリングします。

メッセージ型がプリミティブ型である場合、ラベルおよび入力フィールドがレンダリングされます。メッセージ型がデータ・オブジェクトである場合、コンポーネントはオブジェクトを全探索し、オブジェクト内のエレメントをレンダリングします。

1. Message コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   `bpe:form` タグを `<h:form>` タグに追加します。 `bpe:form` タグには、 model 属性が含まれていなければなりません。

   以下の例では、Message コンポーネントを追加する方法を示します。

   ```
   <h:form>

      <h:outputText value="Input Message" />
      <bpe:form model="#{MyHandler.inputMessage}" readOnly="true" />

      <h:outputText value="Output Message" />
      <bpe:form model="#{MyHandler.outputMessage}" />

   </h:form>
   ```

   Message コンポーネントの model 属性は、com.ibm.bpc.clientcore.MessageWrapper オブジェクトを参照します。このラッパー・オブジェクトは、サービス・データ・オブジェクト (SDO) オブジェクトか、または `int` や `boolean` などの Java プリミティブ型のいずれかをラップします。例では、メッセージは MyHandler 管理対象 Bean のプロパティーによって提供されます。

2. `bpe:form` タグで参照されている管理対象 Bean を構成します。

   以下の例では、MyHandler 管理対象 Bean を構成ファイルに追加する方法を示します。

   ```
   <managed-bean>
   <managed-bean-name>MyHandler</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.sample.jsf.MyHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>

      <managed-property>
         <property-name>type</property-name>
   ```

```
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
    </managed-property>

</managed-bean>
```

3. JSF アプリケーションにカスタム・コードを追加します。

   以下の例では、入力メッセージおよび出力メッセージをインプリメントする方法を示します。

```
public class MyHandler implements ItemListener {

  private TaskInstanceBean taskBean;
  private MessageWrapper inputMessage, outputMessage

  /* Listener method, e.g. when a task instance was selected in a list handler.
   * Ensure that the handler is registered in the faces-config.xml or manually.
   */
  public void itemChanged(Object item) {
    if( item instanceof TaskInstanceBean ) {
        taskBean = (TaskInstanceBean) item ;
    }
  }

  /* Get the input message wrapper
   */
  public MessageWrapper getInputMessage() {
    try{
        inputMessage = taskBean.getInputMessageWrapper() ;
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return inputMessage;
  }

  /* Get the output message wrapper
   */
  public MessageWrapper getOutputMessage() {
    // Retrieve the message from the bean. If there is no message, create
    // one if the task has been claimed by the user. Ensure that only
    // potential owners or owners can manipulate the output message.
    try{
        outputMessage = taskBean.getOutputMessageWrapper();
        if( outputMessage == null
         && taskBean.getState() == TaskInstanceBean.STATE_CLAIMED ) {
            HumanTaskManagerService htm = getHumanTaskManagerService();
            outputMessage = new MessageWrapperImpl();
            outputMessage.setMessage(
               htm.createOutputMessage( taskBean.getID() ).getObject()
            );
         }
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return outputMessage
  }
}
```

   MyHandler 管理対象 Bean は、リスト・ハンドラーへの項目リスナーとして登録できるように、com.ibm.jsf.handler.ItemListener インターフェースをインプリメントします。ユーザーがリスト内の項目をクリックすると、選択された項目について MyHandler Bean が itemChanged(Object item) メソッドで通知されます。ハ

ンドラーは、項目タイプを検査してから、関連した TaskInstanceBean オブジェクトへの参照を保管します。このインターフェースを使用するには、faces-config.xml ファイル内の適切なリスト・ハンドラーにエントリーを追加します。

MyHandler Bean は、getInputMessage および getOutputMessage メソッドを提供します。これらのメソッドはどちらも、MessageWrapper オブジェクトを戻します。メソッドは、参照されたタスク・インスタンス Bean への呼び出しを委任します。例えばメッセージが設定されていないなどの理由で、タスク・インスタンス Bean がヌルを戻した場合、ハンドラーは新規に空のメッセージを作成して保管します。Message コンポーネントは MyHandler Bean が提供するメッセージを表示します。

これで、JSF アプリケーションは、データ・オブジェクトおよびプリミティブ型をレンダリング可能な JavaServer ページを含むようになります。

Message コンポーネント: タグ定義:

Business Process Choreographer Explorer Message コンポーネントは、JavaServer Faces (JSF) アプリケーション内で、`commonj.sdo.DataObject` オブジェクトと、整数およびストリングなどのプリミティブ型をレンダリングします。

Message コンポーネントは、JSF コンポーネント・タグである `bpe:form` から構成されます。

コンポーネント・クラス

com.ibm.bpe.jsf.component.MessageComponent

構文例

```
<bpe:form model="#{TaskInstanceDetailsBean.inputMessageWrapper}"
          simplification="true" readOnly="true"
          styleClass4table="messageData"
          styleClass4output="messageDataOutput">
</bpe:form>
```

タグ属性

表 34. bpe:form 属性

属性	必須	説明
model	はい	`commonj.sdo.DataObject` オブジェクトまたは `com.ibm.bpc.clientcore.MessageWrapper` オブジェクトを参照する値バインディング式。
simplification	いいえ	この属性が `true` に設定されている場合は、カーディナリティーがゼロまたは 1 のプロパティーが表示されます。デフォルトでは、この属性は `false` に設定されています。

表 34. bpe:form 属性 (続き)

属性	必須	説明
readOnly	いいえ	この属性が true に設定されている場合は、読み取り専用フォームのみがレンダリングされます。デフォルトでは、この属性は false に設定されています。
style4validinput	いいえ	有効な入力のレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル。
style4invalidinput	いいえ	無効な入力のレンダリング用の CSS スタイル。
styleClass4validInput	いいえ	有効な入力のレンダリング用の CSS クラス名。
styleClass4invalidInput	いいえ	無効な入力のレンダリング用の CSS クラス名。
styleClass4output	いいえ	出力エレメントのレンダリング用の CSS スタイル・クラス名。
styleClass4table	いいえ	Message コンポーネントによって提供されるテーブルのレンダリング用の、CSS テーブル・スタイルのクラス名。
buttonStyleClass	いいえ	配列またはリストに作動するボタン用の CSS スタイル。

クライアント・モデル・オブジェクトのマッピング

クライアント・モデル・オブジェクトは、Business Process Choreographer API の対応するインターフェースをインプリメントします。

インターフェースをこのようにラップすることにより、ロケールを区別するラベルと、プロパティーのセット用のコンバーターが提供されます。以下の表に、Business Process Choreographer インターフェースから対応するクライアント・モデル・オブジェクトへのマッピングを示します。

表 35. Business Process Choreographer インターフェースからクライアント・モデル・オブジェクトへのマッピング

Business Process Choreographer インターフェース	クライアント・モデル・オブジェクト・クラス
com.ibm.bpe.api.ActivityInstanceData	com.ibm.bpe.clientmodel.bean.ActivityInstanceBean
com.ibm.bpe.api.ActivityServiceTemplateData	com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean
com.ibm.bpe.api.ProcessInstanceData	com.ibm.bpe.clientmodel.bean.ProcessInstanceBean
com.ibm.bpe.api.ProcessTemplateData	com.ibm.bpe.clientmodel.bean.ProcessTemplateBean
com.ibm.task.api.Escalation	com.ibm.task.clientmodel.bean.EscalationBean
com.ibm.task.api.Task	com.ibm.task.clientmodel.bean.TaskInstanceBean
com.ibm.task.api.TaskTemplate	com.ibm.task.clientmodel.bean.TaskTemplateBean

ヒューマン・タスク・イベント用のイベント・ハンドラーの開発

ヒューマン・タスク API イベントとエスカレーション通知イベント用のプラグインを作成することができます。

タスクの処理時に発生するイベントを処理するには、タスク・モデルでイベント・ハンドラー名を指定する必要があります。

ヒューマン・タスク・イベントに対しては、次のタイプのイベント・ハンドラーを作成できます。

通知イベント・ハンドラー
エスカレーション通知用のイベント・ハンドラーを作成するには、com.ibm.task.spi.NotificationEventHandlerPlugin インターフェースをインプリメントする必要があります。

API イベント・ハンドラー
ヒューマン・タスク・イベント用のイベント・ハンドラーを作成するには、com.ibm.task.spi.APIEventHandlerPlugin interface インターフェースをインプリメントする必要があります。

1. イベント・ハンドラーを JAR ファイルとしてインプリメントします。

   JAR ファイルには、次のものが必要です。

   - イベント・ハンドラー・インターフェースをインプリメントするクラス。例えば、通知イベント用の com.ibm.task.spi.NotificationEventHandlerPlugin または API イベント用の com.ibm.task.spi.APIEventHandlerPlugin。以下に例を示します。

     ```
     package com.ibm.task.spi ;

     public interface NotificationEventHandlerPlugin
     {
        public void interFaceMethod(Param param) ;
        :
     }
     ```

   - JAR ファイルで、META-INF/services/ ディレクトリーに com.ibm.task.spi.*%identifier%*_*%type%*EventHandlerPlugin というプロパティー・ファイルが必要です。ここで、*%identifier%* は、例えば MyEventHandler など、モデル内で指定されたイベント・ハンドラー名です。*%type%* は、イベント・タイプ (Notification または API) です。

     コメント行やブランク行ではないこのファイルの最初の行は、プラグイン・インプリメンテーションの名前を指定する必要があります。例えば、

     ```
     META-INF/services/com.ibm.task.spi.MyEventHandlerNotificationEventHandlerPlugin
     ```

     ファイルには次のような行が含まれます。

     ```
     myevents.EventHandlerImplementation
     ```

2. JAR ファイルをアプリケーションで使用できるようにします。
   - イベント・ハンドラーを 1 つの Java 2 Enterprise Edition (J2EE) アプリケーションでのみ使用したい場合は、 JAR ファイルをアプリケーション EAR ファイルに組み込みます。
   - イベント・ハンドラーをいくつかのアプリケーションで使用したい場合は、JAR ファイルを WebSphere Application Server 共用ライブラリーに置くことを検討してください。その後、イベント・ハンドラーにアクセスする必要のあるアプリケーションをライブラリーに明示的に関連付けることができます。

イベント・ハンドラーは、アプリケーション内で発生したヒューマン・タスク・イベントを処理します。

モジュールの準備とインストールの概要

モジュールのインストール (デプロイとも呼ばれる) では、モジュールをテスト環境または実稼働環境のいずれかで活動化します。この概要では、テストおよび実稼働環境と、モジュールのインストールに必要な手順の一部について簡単に説明します。

注: アプリケーションを実稼働環境でインストールするプロセスは、WebSphere Application Server Network Deployment バージョン 6 インフォメーション・センターの『アプリケーションの開発とデプロイ』の項で説明されているプロセスと同様です。これらのトピックをお読みになったことがない場合、まず目を通してください。

モジュールを実稼働環境にインストールする前に、必ずテスト環境での変更内容を確認してください。モジュールをテスト環境にインストールする場合は、WebSphere Integration Developer を使用します。詳しくは、WebSphere Integration Developer インフォメーション・センターを参照してください。モジュールを実稼働環境にインストールする場合は、WebSphere Process Server を使用します。

このトピックでは、モジュールの実稼働環境へのインストールおよびその準備に必要な概念と作業について説明します。モジュールで使用されるオブジェクトを収容するファイルや、モジュールをテスト環境から実稼働環境へ移行する方法について説明しているその他のトピックがあります。モジュールを正しくインストールするためには、これらのファイルとファイルに格納されている内容を理解することが大切です。

ライブラリーと JAR ファイルの概要

モジュールでは、ライブラリー内の成果物を使用することがよくあります。成果物およびライブラリーは、モジュールをデプロイするときに指定する Java アーカイブ (JAR) ファイルに含まれています。

モジュールの開発時に、そのモジュールのさまざまな部分で使用する特定のリソースまたはコンポーネントを指定する場合があります。これらのリソースまたはコンポーネントは、モジュールの開発時に作成したオブジェクトである場合と、既にサーバー上にデプロイされているライブラリー内のオブジェクトである場合があります。ここでは、アプリケーションのインストール時に必要となるライブラリーおよびファイルについて説明します。

ライブラリーの概要

ライブラリーには、WebSphere Integration Developer 内の複数のモジュールで使用されるオブジェクトおよびリソースが格納されています。これらの成果物は、JAR ファイル、リソース・アーカイブ (RAR) ファイル、または Web サービス・アーカイブ (WAR)ファイルに格納されています。これらの成果物の例としては、次のものがあります。

- インターフェースまたは Web サービス記述子 (拡張子 .wsdl のファイル)

- ビジネス・オブジェクトの XML スキーマ定義 (拡張子 .xsd のファイル)
- ビジネス・オブジェクト・マップ (拡張子 .map のファイル)
- リレーションシップ定義とロール定義 (拡張子 .rel および .rol のファイル)

ある成果物がモジュールで必要になると、EAR クラス・パスに基づいてサーバーがこれを探し出し、メモリーにまだロードされていない場合は、ロードします。それ以降は、成果物が置き換えられないかぎりは、その成果物に対する要求では、メモリーにロードしたコピーが使用されます。図 7 に、アプリケーションとコンポーネントおよび関連するライブラリーの包含関係を示します。

図 7. モジュール、コンポーネント、およびライブラリー間の関係

JAR、RAR、および WAR ファイルの概要

モジュールのコンポーネントを格納できるファイルは複数存在します。これらのファイルについては、Java 2 Enterprise Edition (J2EE) 仕様に詳しい説明があります。JAR ファイルの詳細については、JAR 仕様に説明があります。

WebSphere Process Server では、JAR ファイルにアプリケーション (モジュールで使用するほかのサービス・コンポーネントへの支援的な参照およびインターフェースをすべて含んだ、モジュールのアセンブル・バージョン) も格納されています。

アプリケーションを完全にインストールするには、この JAR ファイル、その他のすべてのライブラリー (JAR ファイル、Web サービス・アーカイブ (WAR) ファイル、リソース・アーカイブ (RAR) ファイル、ステージング・ライブラリー (Enterprise Java Beans - EJB) JAR ファイル、またはその他のアーカイブなど) が必要です。また、serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを作成する必要があります (『実動サーバーへのモジュールのインストール』を参照)。

ステージング・モジュールの命名規則

ライブラリー内では、ステージング・モジュールの名前についての要件があります。ステージング・モジュールの名前は、それぞれのモジュールで固有でなければなりません。アプリケーションをデプロイするために必要なその他のモジュールには、ステージング・モジュールの名前との間に競合が発生しないような名前を付けてください。*myService* という名前のモジュールの場合、ステージング・モジュール名は、次のようになります。

- *myService*App
- *myService*EJB
- *myService*EJBClient
- *myService*Web

注: serviceDeploy コマンドは、サービスに WSDL ポート・タイプ・サービスが含まれている場合にのみ、*myService* Web ステージング・モジュールを作成します。

ライブラリー使用時の考慮事項

ライブラリーを使用することにより、ビジネス・オブジェクトの整合性およびモジュール間での処理の整合性が保証されます。なぜなら、それぞれの呼び出し側モジュールは、特定のコンポーネントの専用コピーを所有するからです。不整合や障害が発生しないようにするために、呼び出し側モジュールで使用するコンポーネントおよびビジネス・オブジェクトに対する変更は、すべての呼び出し側モジュール間で整合がとれている必要があります。呼び出し側モジュールを更新するには、次の手順を実行します。

1. モジュールおよびライブラリーの最新コピーを実動サーバーにコピーします。
2. serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを再作成します。
3. 呼び出し側モジュールを含む実行中のアプリケーションを停止し、再インストールします。
4. 呼び出し側モジュールを含むアプリケーションを再始動します。

EAR ファイルの概要

EAR ファイルは、サービス・アプリケーションの実動サーバーへのデプロイにおいて、重要な要素です。

エンタープライズ・アーカイブ (EAR) ファイルとは、アプリケーションがデプロイメントに必要とするライブラリー、エンタープライズ Bean、および JAR ファイルを格納した圧縮ファイルです。

アプリケーション・モジュールを WebSphere Integration Developer からエクスポートするときに、JAR ファイルを作成します。この JAR ファイルおよびその他の成果物ライブラリーまたはオブジェクトを、インストール・プロセスの入力として使用します。 serviceDeploy コマンドは、アプリケーションを構成する、コンポーネントの記述と Java コードを含む入力ファイルから EAR ファイルを作成します。

サーバーへのデプロイの準備

モジュールの開発およびテストが終了したら、モジュールをテスト・システムからエクスポートし、実稼働環境にデプロイする必要があります。アプリケーションをインストールするには、モジュールのエクスポート時に必要となるパスと、モジュールが必要とするライブラリーを認識している必要があります。

このタスクを開始する前に、テスト・サーバーでモジュールの開発およびテストを完了し、各種問題およびパフォーマンス問題を解決しておく必要があります。

このタスクでは、アプリケーションの必要な部分がすべて使用可能かどうか、および実動サーバーに移せるように正しくパッケージされているかどうか検証します。

注: WebSphere Integration Developer からエンタープライズ・アーカイブ (EAR) ファイルをエクスポートし、そのファイルを直接 WebSphere Process Server にインストールすることもできます。

重要: コンポーネント内部のサービスがデータベースを使用する場合、データベースに直接接続されたサーバーにアプリケーションをインストールします。

1. デプロイするモジュール用のコンポーネントを含むフォルダーを見つけます。

   コンポーネント・フォルダーの名前は *module-name* で、その中に *module*.module という名前のファイル (基本モジュール) があります。
2. モジュールに含まれるすべてのコンポーネントが、モジュール・フォルダーの下のコンポーネント・サブフォルダー内にあることを確認します。

   使いやすくするために、サブフォルダーには *module/component* のような名前を付けます。
3. 各コンポーネントを構成するすべてのファイルが適切なコンポーネント・サブフォルダーに格納されていて、*component-file-name*.component といった名前が付けられていることを確認します。

   コンポーネント・ファイルには、モジュール内の個々のコンポーネントの定義が記述されています。
4. 他のすべてのコンポーネントおよび成果物が、それらを必要とするコンポーネントのサブフォルダーに格納されていることを確認します。

   このステップで、コンポーネントが必要とする成果物への参照が使用可能であることを確認します。 serviceDeploy コマンドがステージング・モジュールに対して使用する名前と、これらのコンポーネントの名前が競合していてはなりません。『ステージング・モジュールの命名規則』を参照してください。
5. 参照ファイル (*module*.references) が、ステップ 1 のモジュール・フォルダー内に存在することを確認します。

参照ファイルは、モジュール内の参照およびインターフェースを定義します。

6. ワイヤー・ファイル (*module*.wires) がコンポーネント・フォルダーに存在することを確認します。

   ワイヤー・ファイルは、モジュール内の参照とインターフェース間の接続を完成させます。

7. マニフェスト・ファイル (*module*.manifest) がコンポーネント・フォルダーに存在することを確認します。

   マニフェストは、モジュールと、モジュールを構成するすべてのコンポーネントをリストします。マニフェストには、クラスパス・ステートメントも記述されています。これは、serviceDeploy コマンドが、モジュールが必要とする他のモジュールを検出できるようにするためです。

8. モジュールの圧縮ファイルまたは JAR ファイルを作成します。このファイルは serviceDeploy コマンドの入力として使用します。このコマンドは、実動サーバーにインストールするモジュールを準備してくれます。

デプロイメント前の MyValue モジュールのフォルダー構造の例

以下の例は、MyValue、CustomerInfo、および StockQuote というコンポーネントから構成される、MyValueModule モジュールのディレクトリー構造を示しています。

```
MyValueModule
   MyValueModule.manifest
   MyValueModule.references
   MyValueModule.wiring
   MyValueClient.jsp
process/myvalue
   MyValue.component
   MyValue.java
   MyValueImpl.java
service/customerinfo
   CustomerInfo.component
   CustomerInfo.java
   Customer.java
   CustomerInfoImpl.java
service/stockquote
   StockQuote.component
   StockQuote.java
   StockQuoteAsynch.java
   StockQuoteCallback.java
   StockQuoteImpl.java
```

『実動サーバーへのモジュールのインストール』の説明に従って、モジュールを実動システムにインストールします。

クラスター上のサービス・アプリケーションのインストールに関する考慮事項

クラスターにサービス・アプリケーションをインストールする場合、追加要件が発生します。クラスターにサービス・アプリケーションをインストールする際に、これらの考慮事項に注意することが重要です。

クラスターは、スケール・メリットを提供して、サーバー間の要求ワークロードのバランスを取り、アプリケーションのクライアントに対して一定レベルの可用性を

提供できるようにすることで、処理環境に多くのメリットをもたらす可能性があります。クラスター上で、サービスを含むアプリケーションをインストールする前に、以下の事項を検討してください。

- アプリケーションのユーザーが、クラスタリングにより提供される処理能力と可用性を必要としているか。

  その場合、クラスタリングが正しい解決策です。クラスタリングにより、アプリケーションの可用性と能力が向上します。
- クラスターがサービス・アプリケーション用に正しく準備されているか。

  サービスを含む最初のアプリケーションをインストールして開始する前に、クラスターを正しく構成する必要があります。クラスターを正しく構成していない場合、アプリケーションは要求を正しく処理できません。
- クラスターのバックアップはあるか。

  バックアップ・クラスターにもアプリケーションをインストールする必要があります。

関連タスク

クラスター環境の作成

実動サーバーへのモジュールのインストール

このトピックでは、テスト・サーバーからアプリケーションを取り出して実稼働環境にデプロイする際に行うステップについて説明します。

サービス・アプリケーションを実動サーバーにデプロイする前に、テスト・サーバー上でアプリケーションをアセンブルし、テストします。テストが終了したら、『サーバーへのデプロイの準備』の説明のとおりに関連するファイルをエクスポートし、このファイルをデプロイする実動システムに移します。詳しくは、WebSphere Integration Developer および WebSphere Application Server Network Deployment バージョン 6 のインフォメーション・センターを参照してください。

1. モジュールおよびその他のファイルを実動サーバーにコピーします。

   アプリケーションに必要なモジュールおよびリソース (EAR、JAR、RAR、および WAR ファイル) が、実稼働環境に移動します。
2. serviceDeploy コマンドを実行して、インストール可能な EAR ファイルを作成します。

   このステップでは、アプリケーションを実稼働にインストールする準備として、サーバーにモジュールを定義します。

   a. デプロイするモジュールを含む JAR ファイルを見つけ出します。
   b. 前のステップで見つかった JAR ファイルを入力として使用して、 コマンドを実行します。
3. ステップ 2 で作成した EAR ファイルをインストールします。 アプリケーションのインストール方法は、アプリケーションをスタンドアロン・サーバーにインストールするか、セル内のサーバーにインストールするかによって異なります。

注: 管理コンソールを使用しても、スクリプトを使用しても、アプリケーションをインストールすることができます。追加情報については、WebSphere Application Server インフォメーション・センターを参照してください。

4. 構成を保管します。 これで、モジュールがアプリケーションとしてインストールされました。
5. アプリケーションを開始します。

これで、アプリケーションがアクティブになり、モジュールを通して処理が行われるようになります。

アプリケーションをモニターし、サーバーが要求を正しく処理していることを確認します。

serviceDeploy を使用したインストール可能な EAR ファイルの作成

アプリケーションを実稼働環境にインストールするには、実働サーバーにコピーされたファイルを取得して、インストール可能な EAR ファイルを作成します。

このタスクを開始する前に、サーバーに対してデプロイしようとしているモジュールとサービスを含む JAR ファイルを用意する必要があります。詳しくは、『サーバーへのデプロイの準備』を参照してください。

serviceDeploy コマンドは、JAR ファイルや、これに依存するそのほかの EAR、JAR、RAR、WAR、および ZIP ファイルを取得して、サーバーにインストールできる EAR ファイルを作成します。

1. デプロイするモジュールを含む JAR ファイルを見つけ出します。
2. 前のステップで見つかった JAR ファイルを入力として使用して、 コマンドを実行します。

   このステップでは、EAR ファイルが作成されます。

   注: 管理コンソールで以下のステップを実行します。
3. サーバーの管理コンソールでインストールする EAR ファイルを選択します。
4. 「**保管**」をクリックして、EAR ファイルをインストールします。

ANT タスクを使用したアプリケーションのデプロイ

このトピックでは、ANT タスクを使用して、WebSphere Process Server に対するアプリケーションのデプロイメントを自動化する方法について説明します。 ANT タスクを使用すると、複数のアプリケーションのデプロイメントを定義し、サーバー上でこれらのアプリケーションのデプロイメントを無人で実行することができます。

このタスクでは、以下が前提となります。

- デプロイしようとしているアプリケーションは、すでに開発およびテスト済みである。
- アプリケーションのインストール先が、同じ 1 つまたは複数のサーバーである。
- ANT タスクについての知識がある。
- デプロイメント・プロセスを理解している。

アプリケーションの開発およびテストについての詳細は、WebSphere Integration Developer インフォメーション・センターにあります。

WebSphere Application Server Network Deployment バージョン 6 のインフォメーション・センターのリファレンス部分に、アプリケーション・プログラミング・インターフェースに関するセクションがあります。ANT タスクについては、パッケージ com.ibm.websphere.ant.tasks に説明があります。このトピックに関連するタスクとして、ServiceDeploy および InstallApplicationがあります。

複数のアプリケーションを並行インストールする必要がある場合、デプロイメントの前に ANT タスクを開発します。その後 ANT タスクは、プロセスへのユーザーの介入なしにアプリケーションをサーバーにデプロイし、インストールすることができます。

1. デプロイするアプリケーションを特定します。
2. 各アプリケーションごとに JAR ファイルを作成します。
3. JAR ファイルをターゲット・サーバーにコピーします。
4. ServiceDeploy コマンドを実行する ANT タスクを作成し、サーバーごとに EAR ファイルを作成します。
5. 該当するサーバー上で、ステップ 4 で作成した各 EAR ファイルごとに、InstallApplication コマンドを実行する ANT タスクを作成します。
6. ServiceDeploy ANT タスクを実行し、アプリケーション用の EAR ファイルを作成します。
7. InstallApplication ANT タスクを実行し、ステップ 6 で作成した EAR ファイルをインストールします。

アプリケーションは、ターゲット・サーバー上に正しくデプロイされます。

アプリケーションの無人デプロイの例

以下の例は、ファイル myBuildScript.xml に入っている ANT タスクを示しています。

```
<?xml version="1.0">

<project name="OwnTaskExample" default="main" basedir=".">
 <taskdef name="servicedeploy"
   classname="com.ibm.websphere.ant.tasks.ServiceDeployTask" />
 <target name="main" depends="main2">
  <servicedeploy scaModule="c:/synctest/SyncTargetJAR"
   ignoreErrors="true"
   outputApplication="c:/synctest/SyncTargetEAREAR"
   workingDirectory="c:/synctest"
   noJ2eeDeploy="true"
   cleanStagingModules="true"/>
 </target>
</project>
```

以下のステートメントは、ANT タスクを呼び出す方法を示しています。

```
${WAS}/bin/ws_ant -f myBuildScript.xml
```

ヒント: このファイルにさらに別のプロジェクト・ステートメントを追加して、複数のアプリケーションを無人デプロイすることができます。

管理コンソールを使用して、新しくインストールしたアプリケーション・サーバーが始動され、ワークフローを正しく処理していることを確認してください。

ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール

ビジネス・プロセスまたはヒューマン・タスク、あるいはこの両方を含む Service Component Architecture (SCA) Enterprise JavaBeans (EJB) モジュールをデプロイメント・ターゲットに配布することができます。 サーバーまたはクラスターをデプロイメント・ターゲットとすることができます。

アプリケーションのインストール先とするアプリケーション・サーバーまたはクラスターごとに、ビジネス・プロセス・コンテナーまたはタスク・コンテナーがインストールされ、構成されていることを確認してください。

ビジネス・プロセスまたはヒューマン・タスク・アプリケーションをインストールする前に、以下の条件をすべて満たしていることを確認してください。

- アプリケーションのインストール先とするサーバーが稼動している。
- 各クラスターで、プロセスまたはタスクを使用して Enterprise JavaBeans モジュールをインストールする対象のサーバーが少なくとも 1 台実行されている。

ビジネス・プロセスおよびタスク・アプリケーションを、管理コンソールやコマンド行から、または管理スクリプトを実行してインストールすることができます。管理スクリプトを実行してビジネス・プロセス・アプリケーションまたはヒューマン・タスク・アプリケーションをインストールする場合、サーバー接続が必要です。 `-conntype NONE` オプションをインストール・オプションとして使用しないでください。

1. アプリケーションをクラスター上にインストールする場合、アプリケーションがクラスターにちなんで名前を付けられたデータ・ソースを使用していることを確認します。

   例えば、アプリケーションがデフォルト・データ・ソース `BPEDB` を使用して生成された場合、アプリケーションのデータ・ソースを BPEDB_*cluster_name* に変更します。ここで、*cluster_name* は、アプリケーションのインストール先のクラスターの名前です。
2. アプリケーションをインストールします。 詳しくは、WebSphere Application Server のインフォメーション・センターの『コンソールへのアプリケーション・ファイルのインストール』を参照してください。

すべてのビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートが、開始状態になります。

プロセス・インスタンスまたはタスク・インスタンスを作成するには、アプリケーションを始動する必要があります。

モデルのデプロイ

WebSphere Integration Developer がプロセスのデプロイメント・コードを生成するとき、プロセスまたはタスク・モデルの構成要素は、さまざまな Java 2 Enterprise

Edition (J2EE) 構成要素および成果物にマップされます。すべてのデプロイメント・コードは、エンタープライズ・アプリケーション (EAR) ファイルにパッケージ化されます。デプロイするモデルの新規バージョンごとに、新しいエンタープライズ・アプリケーションにパッケージ化する必要があります。

ビジネス・プロセス・モデルまたはヒューマン・タスク・モデルの J2EE 構成要素で構成されるエンタープライズ・アプリケーションをインストールすると、モデルの構成要素が必要に応じて、プロセス・テンプレート またはタスク・テンプレートとして Business Process Choreographer データベースに格納されます。データベース・システムが稼働していない場合、またはデータベース・システムにアクセスできない場合、デプロイは失敗します。デフォルトでは、新しくインストールされたテンプレートは、開始済み状態となります。ただし、新しくインストールされたエンタープライズ・アプリケーションの場合は、停止状態になります。インストール済みのエンタープライズ・アプリケーションは、個々に開始したり停止したりすることができます。

新バージョンのプロセス・テンプレートまたはタスク・テンプレートの名前は同じですが、有効開始日属性が異なります。異なるアプリケーションそれぞれに、多数のバージョンのプロセス・テンプレートやタスク・テンプレートをデプロイできます。ただし、1 つのプロセスの 2 つのバージョンが同じ有効開始日を持つことはできません。1 つのプロセスの異なるバージョンをインストールする場合は、バージョンごとに異なる有効開始日を指定してください。データベースには、異なるプロセス・バージョンのすべてが格納されます。

有効開始日を指定しない場合、日付は次のように決定されます。

- ヒューマン・タスクの場合、有効開始日はアプリケーションがインストールされた日付です。
- ビジネス・プロセスの場合、有効開始日はプロセスがモデル化された日付です。

サーバーが稼動していないクラスターにプロセス・アプリケーションをインストール可能な場合

このトピックでは、サーバーが稼動していないクラスターにアプリケーションをインストールする必要があるような、例外的な状況について説明します。

サーバー上のビジネス・プロセス・アプリケーションのインストール時に、対応するビジネス・プロセス・コンテナーのデータ・ソースの Java Naming and Directory Interface (JNDI) 名を解決する必要があります。そのため、サーバー接続がなければ、アプリケーションをインストールできません。Network Deployment (ND) 環境では、このサーバーはデプロイメント・マネージャーです。

撤廃された制約事項

ND 環境でクラスターにビジネス・プロセス・アプリケーションをインストールする場合、以下の条件が真である場合は、クラスター内のサーバーが稼動している必要はありません。

- 必要なデータ・ソースがセル・レベルで定義されている。
- プロセス・アプリケーションがヒューマン・タスクを指定していない。

ヒューマン・タスクを持たないプロセス・アプリケーションでは、アプリケーション・サーバーのネーム・スペースでの検索操作が前に失敗した場合、データ・ソース検索操作はデプロイメント・マネージャーのネーム・スペース内で実行されます。アプリケーションが正常にインストールされた場合は、アプリケーション・サーバー・ネーム・スペース内部のデータ・ソース検索操作の失敗を示す SystemOut.log ファイル内のエラー・メッセージは無視してください。

機能する場合

- デプロイメント・マネージャー・ネーム・スペース内部の検索操作は、データ・ソース JNDI 名がセル・レベルで定義されている場合のみ、正常に実行されます。
- ウィザードを使用して、スタンドアロン・サーバー上のビジネス・プロセス・コンテナーまたはヒューマン・タスク・コンテナーを構成した場合、データ・ソースはサーバー・レベルで定義されます。アプリケーション・サーバーのインストール済み環境の ProcessChoreographer/sample ディレクトリーで提供されている構成スクリプト bpeconfig.jacl を使用した場合にも、同じことが当てはまります。この場合、手動でデータ・ソースをセル・レベルで定義し、ビジネス・プロセス・コンテナーのインストール時にこのデータ・ソースを使用する必要があります。
- クラスター・メンバーでウィザードを使用してビジネス・プロセス・コンテナーを構成した場合、データ・ソースは自動的にセル・レベルで定義されます。JNDI 名は、クラスター名によってスコープが決まります。アプリケーション・サーバーのインストール済み環境の ProcessChoreographer/sample ディレクトリーで提供されている構成スクリプト bpeconfig.jacl を使用した場合にも、同じことが当てはまります。この場合、何も手動で変更する必要はありません。

機能しない場合

ヒューマン・タスクを持つプロセス・アプリケーションでは、追加の JNDI 名検索操作によって、スタッフ・プラグイン・プロバイダーを見つける必要があります。したがって、このようなアプリケーションのインストールを確実に正常に実行するため、稼動しているサーバーがクラスターに含まれるようにしてください。

スコープによる副次作用

名前検索の副次作用としては、アプリケーション・サーバーが稼動しておらず、データ・ソースが、サーバーまたはノード・レベルでもセル・レベルのデータ・ソースと同じ名前で定義されている場合、セル・レベルのデータ・ソースが優先されます。すなわち、配置中と実行時とで異なるデータ・ソースを使用することになります。

重要: 名前の競合を回避してください。手動により、セル・レベルでデータ・ソースを定義する場合、クラスター名、またはサーバー名およびノード名によってスコープの決められた JNDI 名 (例えば jdbc/BPEDB_) を使用します。

管理コンソールを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

ビジネス・プロセスまたはヒューマン・タスクが含まれるエンタープライズ・アプリケーションをアンインストールするには、以下のアクションを実行します。

1. アプリケーションのすべてのプロセスおよびタスクのテンプレートを停止します。

   このアクションにより、プロセスおよびタスクのインスタンスの作成が回避されます。

   a. 管理コンソールのナビゲーション・ペインで、「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」をクリックします。
   b. 停止するアプリケーションを選択します。
   c. 「関連項目」の下で、「**EJB モジュール**」をクリックして、Enterprise JavaBeans モジュールを選択します。 複数の EJB モジュールがある場合は、ビジネス・プロセスまたヒューマン・タスクが含まれる Service Component Architecture (SCA) モジュールに対応する EJB モジュールを選択してください。 SCA モジュール名の後に EJB を付けると、対応する EJB モジュールを見つけることができます。例えば、SCA モジュールの名前が `TestProcess` であれば、 EJB モジュールは `TestProcessEJB.jar` になります。
   d. 「追加プロパティー」の下で、「**ビジネス・プロセス**」か「**ヒューマン・タスク**」、または両方を必要に応じてクリックします。
   e. 適切なチェック・ボックスをクリックして、プロセスおよびタスクのテンプレートをすべて選択します。
   f. 「**停止**」をクリックします。

   ビジネス・プロセス・テンプレートまたはタスク・テンプレートが含まれるすべての EJB モジュールで、このステップを繰り返します。
2. データベース、クラスターごとに少なくとも 1 つのアプリケーション・サーバー、アプリケーションがデプロイされているスタンドアロン・サーバーが動作していることを確認してください。

   Network Deployment (ND) 環境では、デプロイメント・マネージャー、すべての ND 管理対象スタンドアロン・アプリケーション・サーバー、および少なくとも 1 台のアプリケーション・サーバーが、アプリケーションがインストールされているそれぞれのクラスターごとに実行されている必要があります。
3. プロセス・インスタンスまたはタスク・インスタンスが存在していないことを確認します。

   必要であれば、管理者は、Business Process Choreographer Explorer を使用して、残存するプロセス・インスタンスまたはタスク・インスタンスを削除することができます。
4. アプリケーションを停止してアンインストールするには、以下のようにします。
   a. 管理コンソールのナビゲーション・ペインで、「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」をクリックします。

b. アンインストールするアプリケーションを選択し、「**停止**」をクリックします。

   アプリケーションでプロセス・インスタンスまたはタスク・インスタンスがまだ存在する場合は、このステップは失敗します。

c. アンインストールするアプリケーションを再度選択し、「**アンインストール**」をクリックします。

d. 「**保管**」をクリックして、変更を保管します。

アプリケーションはアンインストールされます。

管理コマンドを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

管理コマンドには、ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするための、管理コンソールの代替方法が用意されています。

グローバル・セキュリティーが使用可能な場合は、ユーザー ID にオペレーター権限があることを確認します。

管理クライアントが接続しているサーバー・プロセスが動作していることを確認します。

- ND 環境では、サーバー・プロセスはデプロイメント・マネージャーです。
- スタンドアロン環境では、サーバー・プロセスはアプリケーション・サーバーです。

管理クライアントが自動的にサーバー・プロセスに接続できるよう、 `-conntype NONE` オプションをコマンド・オプションとして使用しないでください。

Business Process Choreographer Explorer を使用するなどして、アプリケーション内のテンプレートに関連するプロセス・インスタンスまたはタスク・インスタンスを必ず削除してください。

次の手順で、bpcTemplates.jacl スクリプトを使用して、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートを含むアプリケーションをアンインストールする方法を示します。また、アプリケーションをアンインストールするには、そのアプリケーションに属しているテンプレートを停止する必要があります。 bpcTemplates.jacl スクリプトを使用して、1 つの手順でテンプレートを停止およびアンインストールできます。

1. Business Process Choreographer サンプル・ディレクトリーに移動します。 次のように入力します。

   ```
   cd install_root/ProcessChoreographer/sample
   ```

2. テンプレートを停止して、対応するアプリケーションをアンインストールします。

   ```
   install_root/bin/wsadmin -f bpcTemplates.jacl
                                [-user user_name]
                                [-password user password]
                                -uninstall application_name
   ```

各部の意味は、次のとおりです。

user_name
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー ID を提供します。

user_password
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー・パスワードを提供します。

application_name
グローバル・セキュリティーが使用可能な場合は、認証用のユーザー・パスワードを提供します。

アプリケーションはアンインストールされます。

アプリケーションおよび組み込み WebSphere アダプターのインストール

アプリケーションに WebSphere アダプターを組み込んで開発する場合、そのアダプターはアプリケーションと共にデプロイされます。アダプターを別途インストールする必要はありません。アプリケーションを組み込みアダプターと共にインストールする手順を以下に示します。

アプリケーションを組み込みの WebSphere アダプターを使用して開発する場合にのみ、この作業を行ってください。

1. アプリケーションとその中のリソース・アダプター・アーカイブ (RAR) モジュールをアセンブルします。『アプリケーションのアセンブル』を参照してください。
2. 『新規アプリケーションのインストール』のステップに従って、アプリケーションをインストールします。『モジュールのサーバーへのマップ』ステップで、RAR ファイルごとにターゲット・サーバーまたはクラスターを指定します。RAR モジュールに定義されているリソース・アダプターを使用するその他のモジュールはすべて、必ず同じターゲットにマップしてください。また、このアプリケーションに対する要求のルーターとして機能するターゲットとして、Web サーバーを指定します。各 Web サーバーのプラグイン構成ファイル (plugin-cfg.xml) は、Web サーバーを経由するアプリケーションに基づいて生成されます。

   注: サーバーに RAR ファイルをインストールする場合、WebSphere Application Server はコネクター・モジュールのマニフェスト (MANIFEST.MF) を探します。最初、RAR ファイルの connectorModule.jar ファイルで探し、_connectorModule.jar ファイルからマニフェストをロードします。クラスパス・エントリーが connectorModule.jar ファイルから得たマニフェストにある場合は、 RAR はそのクラスパスを使用します。インストール済みコネクター・モジュールが必要なクラスおよびリソースを見つけることを確認するには、コンソールを使用して、RAR の「クラスパス」設定を確認します。詳しくは、リソース・アダプターの設定および WebSphere リレーショナル・リソース・アダプターの設定を参照してください。
3. 変更を保管します。 **「終了」>「保管」**をクリックします。
4. 新規にインストールされたアプリケーションの接続ファクトリーを作成します。

a. 管理コンソールを開きます。
b. 新しくインストールされたアプリケーションを選択します。 「**アプリケーション**」 >「**エンタープライズ・アプリケーション**」>「***application name***」をクリックします。
c. このページの「関連項目」セクションの「**コネクター・モジュール**」をクリックします。
d. RAR ファイルを選択します。 「***filename.rar***」をクリックします。
e. このページの「追加プロパティー」セクションの「**リソース・アダプター**」をクリックします。
f. このページの「追加プロパティー」セクションの「**J2C 接続ファクトリー**」をクリックします。
g. 既存の接続ファクトリーを更新する場合はそれをクリックし、新規に作成する場合は「**新規**」をクリックします。

注: WebSphere アダプターが EIS インポートまたは EIS エクスポートを使用して構成されている場合は、ConnectionFactory または ActivationSpec が存在し、これらを更新することができます。

ネイティブ・パス・エレメントが含まれているアダプターをインストールする場合は、次のことを考慮してください。複数のネイティブ・パス・エレメントがあり、ネイティブ・ライブラリーの 1 つ (ネイティブ・ライブラリー A) が別のライブラリー (ネイティブ・ライブラリー B) に依存している場合、ネイティブ・ライブラリー B をシステム・ディレクトリーにコピーする必要があります。ほとんどの UNIX システムでは制限があるため、ネイティブ・ライブラリーをロードしようとする際に現行ディレクトリーを調べません。

接続ファクトリーを作成および保管した後、適宜に、アプリケーションの各種モジュールで定義されているリソース参照を変更し、接続ファクトリーの Java Naming and Directory Interface (JNDI) 名を指定します。

注:

所定のネイティブ・ライブラリーは、 Java 仮想マシン (JVM) のインスタンスごとに 1 度だけロードできます。アプリケーションごとに独自のクラス・ローダーがあるため、組み込み RAR ファイルを持つ異なるアプリケーション同士が同じネイティブ・ライブラリーを使用することはできません。2 番目のアプリケーションは、ライブラリーをロードしようとすると例外を受け取ります。

アプリケーション・サーバーにデプロイされているアプリケーションが、ネイティブ・パス・エレメントを含む組み込み RAR ファイルを使用する場合、未解決のトランザクションがない状態で、アプリケーション・サーバーが完全にシャットダウンしたことを必ず確認する必要があります。アプリケーション・サーバーが完全にシャットダウンしない場合は、サーバーの再始動時にリカバリーを実行し、必要な RAR ファイルおよびネイティブ・ライブラリーをロードします。リカバリーが完了したら、アプリケーションに関する作業は行わないようにしてください。サーバーをシャットダウンして、

再始動します。この再始動時にアプリケーション・サーバーによってこれ以上リカバリーが行われることはなく、通常のアプリケーション処理を続行できます。

WebSphere アダプター

WebSphere アダプター (または JCA アダプター、J2C アダプター) は、Java アプリケーションがエンタープライズ情報システム (EIS) への接続に使用するシステム・レベルのソフトウェア・ドライバーです。WebSphere アダプターは、JCA 仕様のバージョン 1.5 に準拠しています。

WebSphere アダプターは、アプリケーション・サーバーにプラグインし、EIS、アプリケーション・サーバー、およびエンタープライズ・アプリケーション間の接続を提供します。

アプリケーション・サーバーのベンダーは、J2EE コネクター・アーキテクチャー (JCA) をサポートするように一度そのシステムを拡張すれば、その後は複数の EIS へのシームレスな接続を保証されます。同様に、EIS ベンダーは、コネクター・アーキテクチャーをサポートするアプリケーション・サーバーにプラグインするための機能を備えた 1 つの標準の WebSphere アダプターを提供します。

WebSphere Process Server では、WebSphere リレーショナル・リソース・アダプター (RRA) のインプリメンテーションを提供します。この WebSphere アダプターは、JDBC 呼び出しによってデータ・アクセスを実現し、データベースに動的にアクセスします。接続管理は、JCA 接続管理アーキテクチャーに基づいて行われます。これは、接続プール、トランザクション、およびセキュリティー・サポートを提供します。 WebSphere Process Server バージョン 6.0 では、JCA バージョン 1.5 をサポートします。

コンテナー管理パーシスタンス (CMP) Bean のデータ・アクセスは、 WebSphere Persistence Manager から間接的に管理されます。JCA 仕様により、パーシスタンス・マネージャーは、バックエンド・ストアを特に認識せずに、WebSphere アダプターへのデータ・アクセスを代行することができます。リレーショナル・データベース・アクセスの場合、パーシスタンス・マネージャーは、リレーショナル・リソース・アダプターを使用して、データベースのデータにアクセスします。 JDBC API でサポートされるデータベース・プラットフォームに関しては、 WebSphere Process Server の前提条件を記載した Web サイトを参照してください。

IBM は、WebSphere Process Server パッケージとは別に、多数のエンタープライズ・システム用のリソース・アダプターを用意しています。これには顧客情報管理システム (CICS)、Host On-Demand (HOD)、情報管理システム (IMS)、および SAP (Systems, Applications, and Products) R/3 など (ただし、これらに限定されない) が含まれます。

WebSphere Process Server では、WebSphere アダプターとのインターフェースに、EIS インポートおよび EIS エクスポートを使用します。これに代わる方法として、Rational Application Developer などのツールを使用して、EJB セッション Bean やサービスを開発することにより、WebSphere アダプターを使用するアプリケーショ

ンを作成することもできます。セッション Bean は、javax.resource.cci インターフェースを使用して、WebSphere アダプターを介したエンタープライズ情報システムとの通信を行います。

WebSphere アダプターのデプロイメントに関する考慮事項

WebSphere アダプターのデプロイメントには、スコープに関する特定のオプションが必要です。

WebSphere アダプターは、管理コンソールを使用して、次の 2 とおりの方法でデプロイすることができます。

- スタンドアロン: アダプターがノード・レベルでインストールされ、特定のアプリケーションに関連付けられない。

  注: スタンドアロンの WebSphere アダプターのデプロイメントは、WebSphere Process Server v6.0 ではサポートされません。
- 組み込み: アダプターがアプリケーションの一部であり、アプリケーションをデプロイすると、アダプターも一緒にデプロイされる。

組み込み WebSphere アダプターの場合:

- RAR ファイルを、(EIS インポートまたはエクスポートを使用する) SCA モジュール内でアプリケーション・スコープを持つように設定できます。
- RAR ファイルを、SCA 以外のモジュール内でアプリケーション・スコープを持つように設定できます。EIS インポートまたはエクスポートを含むアプリケーション自体は、別個の SCA モジュールです。

スタンドアロンの WebSphere アダプターは、インストールしないでください。

注: 管理コンソールによって、スタンドアロンの WebSphere アダプターのインストールが妨げられることはありませんが、これはインストールすべきではありません。WebSphere アダプターは、アプリケーションに組み込んでください。

組み込み WebSphere アダプターのみ、WebSphere Process Server のデプロイメントに適しています。また、組み込みの WebSphere アダプターのデプロイメントは、SCA モジュール内でアプリケーション・スコープを持つように設定された RAR ファイルでしかサポートされません。SCA モジュール以外でのデプロイメントはサポートされません。

スタンドアロンの WebSphere アダプターのインストール

WebSphere アダプターは、アプリケーションに組み込んでください。スタンドアロンの WebSphere アダプターは、WebSphere Process Server v6.0 ではサポートされません。ここで示す説明は、参考のため記載しています。スタンドアロンの WebSphere アダプターを使用する必要がある場合は、この説明に従ってインストールしてください。代わりに、関連したアプリケーションのインストールの一部として自動的にインストールされる、組み込みアダプターを使用することができます。

アダプターをインストールする前に、データベースを構成する必要があります。

この操作を実行するには、管理コンソールへのアクセス権限を持ち、管理コンソールに必要なセキュリティーのロールに属していることが必要です。

1. 「RAR ファイルのインストール (Install RAR file)」ダイアログ・ウィンドウを開きます。

   管理コンソールで、以下の手順を実行します。

   a. 「**リソース**」を展開します。

   b. 「**リソース・アダプター**」をクリックします。

   c. このリソース・アダプターを定義する有効範囲を選択します。この有効範囲は、ご使用の接続ファクトリーの有効範囲になります。セル、ノード、クラスター、またはサーバーを選択することができます。

   d. 「**Install RAR**」をクリックします。

   ウィンドウが開きます。ここで JCA コネクターをインストールし、JCA コネクター用に WebSphere アダプターを作成することができます。「新規」ボタンを使用することもできますが、「新規」ボタンによって作成されるのは、新規リソース・アダプターだけです (JCA コネクターがシステムにインストール済みである必要があります)。

   注: このダイアログを使用して RAR ファイルをインストールすると、「リソース・アダプター」ページで定義する有効範囲は、RAR ファイルのインストール場所では無効になります。 RAR ファイルは、ノード・レベルでのみインストールできます。ファイルがインストールされているノードは、「Install RAR」ページ上の有効範囲によって決定されます。 (「リソース・アダプター」ページで設定する有効範囲により、新規リソース・アダプターの有効範囲が決まります。新規リソース・アダプターは、サーバー、ノード、またはセルの各レベルでインストールすることができます。)

2. RAR ファイルをインストールします。

   ダイアログから、WebSphere アダプターを以下のようにインストールします。

   a. JCA コネクターのロケーションをブラウズします。 RAR ファイルがローカル・ワークステーション上にある場合、「ローカル・パス」を選択してブラウズし、ファイルを探します。RAR ファイルがネットワーク・サーバー上にある場合は、「**サーバー・パス**」を選択し、ファイルの完全修飾パスを指定します。

   b. 「**次へ**」をクリックします。

   c. 「一般プロパティー」の下で、リソース・アダプター名およびその他の必要なすべてのプロパティーを入力します。ネイティブ・パス・エレメントが含まれている J2C リソース・アダプターをインストールする場合は、次のことを考慮してください。複数のネイティブ・パス・エレメントがあり、ネイティブ・ライブラリーの 1 つ (ネイティブ・ライブラリー A) が別のライブラリー (ネイティブ・ライブラリー B) に依存している場合、ネイティブ・ライブラリー B をシステム・ディレクトリーにコピーする必要があります。ほとんどの UNIX システムでは制限があるため、ネイティブ・ライブラリーをロードしようとする際に現行ディレクトリーを調べません。

   d. 「**OK**」をクリックします。

クラスターのメンバーとしての WebSphere アダプター・アプリケーション

WebSphere アダプター・モジュール・アプリケーションは、一定の条件下で、クラスターのメンバーとして複製することができます。

WebSphere アダプター・モジュール・アプリケーションには、アダプターを介した情報の流れによって、次の 3 つのタイプがあります。

- EIS エクスポートのみを使用する WebSphere アダプター・アプリケーション: インバウンド・トラフィックのみ。
- EIS インポートのみを使用する WebSphere アダプター・アプリケーション:アウトバウンド・トラフィックのみ。
- EIS インポートとエクスポートの両方を使用する WebSphere アダプター・アプリケーション: 両方向トラフィック。

Network Deployment 環境でのアプリケーションのスケーラビリティーと可用性を実現するために、クラスターが使用されます。

インバウンド・トラフィックまたは両方向のトラフィックが発生する WebSphere アダプター・モジュール・アプリケーションは、クラスターのメンバーとして複製できません。純粋なアウトバウンド・トラフィックのみが発生するアプリケーションを、クラスターのメンバーとして複製することができます。

外部のオペレーティング・システム高可用性 (HA) 管理ソフトウェア・パッケージ (HACMP™、Veritas、Microsoft® Cluster Server など) を使用すると、インバウンドまたは双方向の (すなわち、EIS エクスポートを含む) WebSphere アダプターを使用するアプリケーションでも、Network Deployment 環境での可用性を提供することができます。

クラスター・メンバーとしての WebSphere Business Integration Adapter アプリケーション

WebSphere Business Integration Adapter モジュール・アプリケーションは、一定の条件下で、クラスターのメンバーとして複製することができます。

WebSphere Business Integration Adapter モジュール・アプリケーションには、アダプターを介した情報の流れによって、次の 3 つのタイプがあります。

- EIS エクスポートのみを使用する WebSphere Business Integration Adapter アプリケーション: インバウンド・トラフィックのみ。
- EIS インポートのみを使用する WebSphere Business Integration Adapter アプリケーション: アウトバウンド・トラフィックのみ。
- EIS インポートとエクスポートの両方を使用する WebSphere Business Integration Adapter アプリケーション: 両方向トラフィック。

Network Deployment 環境でのアプリケーションのスケーラビリティーと可用性を実現するために、クラスターが使用されます。

インバウンド・トラフィックまたは両方向のトラフィックが発生する WebSphere Business Integration Adapter モジュール・アプリケーションは、クラスターのメンバ

ーとして複製できません。純粋なアウトバウンド・トラフィックのみが発生するアプリケーションを、クラスターのメンバーとして複製することができます。

外部のオペレーティング・システム高可用性 (HA) 管理ソフトウェア・パッケージ (HACMP、Veritas、Microsoft Cluster Server など) を使用すると、インバウンドまたは両方向の (すなわち、EIS エクスポートを含む) WebSphere Business Integration Adapter を使用するアプリケーションでも、Network Deployment 環境での可用性を提供することができます。

EIS アプリケーションのインストール

EIS アプリケーション・モジュール、すなわち、EIS アプリケーション・モジュール・パターンに従った Service Component Architecture (SCA) モジュールは、 J2SE プラットフォームまたは J2EE プラットフォームにデプロイすることができます。

EIS モジュールへのデプロイに必要な手順は、プラットフォームによって異なります。

詳しくは、これ以降の作業を参照してください。

J2SE プラットフォームへの EIS アプリケーション・モジュールのデプロイ

EIS モジュールを J2SE プラットフォームにデプロイすることはできますが、EIS インポートのみがサポートされます。

この作業を開始する前に、 WebSphere 統合開発環境で、JMS インポート・バインディングを使用して、EIS アプリケーション・モジュールを作成する必要があります。

メッセージ・キューを使用して EIS システムに非同期にアクセスしたい場合は、EIS アプリケーション・モジュールに JMS インポート・バインディングを提供します。

J2SE プラットフォームにデプロイするのは、バインディングのインプリメンテーションを管理対象外モードで実行できる場合のみです。 JMS バインディングでは、非同期および JNDI のサポートが必要です。このどちらも、基本の Service Component Architecture または J2SE では提供されません。J2EE Connector Architecture では、管理対象外のインバウンド通信がサポートされないため、EIS エクスポートは除去されます。

EIS インポートを使用した EIS アプリケーション・モジュールを J2SE にデプロイする場合、モジュールの依存関係だけでなく、インポートで使用される WebSphere アダプターを依存関係として、マニフェストまたは SCA がサポートするその他の形式で指定する必要があります。

J2EE プラットフォームへの EIS アプリケーション・モジュールのデプロイ

EIS モジュールを J2EE プラットフォームにデプロイすると、EAR ファイルとしてパッケージ化されたアプリケーションがサーバーにデプロイされます。J2EE のすべての成果物およびリソースが作成され、アプリケーションが構成されて、実行準備ができます。

この作業を開始する前に、 WebSphere 統合開発環境で、JMS インポート・バインディングを使用して、EIS モジュールを作成する必要があります。

J2EE プラットフォームへのデプロイでは、以下の J2EE 成果物およびリソースが作成されます。

表 36. バインディングから J2EE 成果物への対応

SCA モジュールでのバインディング	生成される J2EE 成果物	作成される J2EE リソース
EIS インポート	モジュールのセッション EJB で生成されるリソース参照。	ConnectionFactory
EIS エクスポート	リソース・アダプターによってサポートされるリスナー・インターフェースに応じて生成またはデプロイされるメッセージ駆動型 Bean。	ActivationSpec
JMS インポート	ランタイムにより提供されるメッセージ駆動型 Bean (MDB) がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。MDB はインポートに受信宛先が指定されている場合にのみ作成される。	• ConnectionFactory • ActivationSpec • Destination
JMS エクスポート	ランタイムにより提供されるメッセージ駆動型 Bean がデプロイされ、モジュールのセッション EJB でリソース参照が生成される。	• ActivationSpec • ConnectionFactory • Destination

インポートまたはエクスポートで、ConnectionFactory のようなリソースを定義している場合、リソース参照はモジュールのステートレス・セッション EJB のデプロイメント記述子内に生成されます。また、適切なバインディングが EJB バインディング・ファイル内に生成されます。リソース参照がバインドされる名前は、モジュール名とインポート名を基にして、ターゲット属性の値 (値が存在する場合)、またはリソースに与えられたデフォルトの JNDI 検索名のいずれかになります。

デプロイ時に、実装環境により、モジュールのセッション Bean が検出され、これを使用して、リソースが検索されます。

サーバーへのアプリケーションのデプロイ中に、EIS インストール・タスクにより、バインドされたエレメント・リソースが存在するかどうかの確認が行われます。このリソースが存在しない場合、SCDL ファイルで 1 つ以上のプロパティーを

指定しているときは、EIS インストール・タスクによって、このリソースが作成され、構成されます。リソースが存在しない場合は、何も処置は行われず、リソースを作成してからアプリケーションを実行するものとみなされます。

JMS インポートが受信宛先を指定してデプロイされた場合、メッセージ駆動型 Bean (MDB) がデプロイされます。これにより、送信された要求に対する応答を listen します。MDB は、JMS メッセージの JMSreplyTo ヘッダー・フィールドにある、要求で送信された Destination に関連付けられます (この Destination で listen します)。MDB は、応答メッセージが到着すると、メッセージの相関 ID を使用して、コールバック Destination に保管されているコールバック情報を取得して、コールバック・オブジェクトを呼び出します。

インストール・タスクでは、インポート・ファイル内の情報から ConnectionFactory と 3 つの宛先が作成されます。これに加えて、ActivationSpec を作成して、ランタイム MDB が受信宛先で応答を listen できるようにします。ActivationSpec のプロパティーは、Destination/ConnectionFactory プロパティーから派生しています。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

JMS エクスポートをデプロイする場合、メッセージ駆動型 Bean (MDB) (JMS インポートの場合にデプロイされる MDB とは異なる MDB) がデプロイされます。MDB は、受信宛先で着信要求を listen して、その要求が SCA で処理されるようにディスパッチします。インストール・タスクでは、JMS インポートの場合と同様のリソース・セット、ActivationSpec、応答の送信に使用される ConnectionFactory、および 2 つの宛先が作成されます。これらのリソースのプロパティーはすべて、エクスポート・ファイルに指定されます。JMS プロバイダーが SIBus リソース・アダプターである場合、JMS の Destination に対応する SIBus の Destination が作成されます。

失敗したデプロイメントのトラブルシューティング

このトピックでは、アプリケーションのデプロイ時の問題の原因を判別するために行うステップについて説明します。また、参考になるいくつかのソリューションも示されています。

このトピックは、以下の事項を前提としています。
- モジュールのデバッグの基本について理解している。
- モジュールのデプロイ中にロギングおよびトレースがアクティブになっている。

デプロイメントのトラブルシューティングのタスクは、エラーの通知を受け取った後に開始します。失敗したデプロイメントには、アクションをとる前に検査する必要のあるさまざまな症状があります。

1. アプリケーションのインストールが失敗したかどうか判別します。

   system.out ファイルを調べて、失敗の原因を示すメッセージを探します。アプリケーションをインストールできない理由には、以下のようなものがあります。

   - 同一の Network Deployment セル内の複数のサーバーにアプリケーションをインストールしようとしている。

- アプリケーションの名前が、アプリケーションをインストールする Network Deployment セル上の既存のモジュールの名前と同じである。
- EAR ファイル内部の J2EE モジュールを異なるターゲット・サーバーにデプロイしようとしている。

重要: インストールが失敗し、アプリケーションにサービスが含まれる場合、アプリケーションの再インストールを試みる前に、失敗の前に作成された SIBus 宛先または J2C 活動化仕様を除去する必要があります。これらの成果物を除去する最も簡単な方法は、失敗後に「**保管**」>「**すべて廃棄 (Discard all)**」をクリックする方法です。不注意で変更を保管した場合、SIBus 宛先および J2C 活動化仕様を手動で除去する必要があります (『管理』セクションの『SIBus 宛先の削除』および『J2C 活動化仕様の削除』を参照)。

2. アプリケーションが正常にインストールされている場合、そのアプリケーションが開始されているかどうか調べます。

   アプリケーションが稼働していない場合、サーバーがそのアプリケーション用のリソースを始動しようとしたときに障害が発生したということです。

   a. system.out ファイルを調べて、対処法を指示するメッセージを探します。
   b. リソースが開始しているかどうかを判別します。

      リソースが開始されていないと、アプリケーションは実行されません。これは情報の損失を防ぐためです。リソースが開始しない理由には次のものがあります。

      - 指定されたバインディングが正しくない。
      - リソースが正しく構成されていない。
      - リソースがリソース・アーカイブ (RAR) ファイルに含まれていない。
      - Web リソースが Web サービス・アーカイブ (WAR) ファイルに含まれていない。

   c. コンポーネントが欠落していないかどうか判別します。

      コンポーネント欠落の原因は、エンタープライズ・アーカイブ (EAR) ファイルが正しく作成されなかったことにあります。モジュールが必要とするすべてのコンポーネントが、Java アーカイブ (JAR) ファイルを作成したテスト・システムの正しいフォルダーに格納されているか確認します。追加情報については、**『モジュールの開発とデプロイ』>『モジュールの準備とインストールの概要』>『サーバーへのデプロイの準備』**を参照してください。

3. アプリケーションで情報が処理されているかどうかを調べます。

   実行中のアプリケーションでも、情報の処理に失敗することがあります。この理由は、ステップ 2b で示した理由と同様です。

   a. アプリケーションが、別のアプリケーションに含まれるサービスを使用するかどうかを判別します。 他方のアプリケーションがインストールされ稼働していることを確認してください。
   b. 障害が発生したアプリケーションが使用する、他のアプリケーションに含まれるすべてのサービスのインポート・バインディングおよびエクスポート・

バインディングが、正しく構成されていることを判別します。 管理コンソールを使用して、バインディングを調べ、訂正してください。

4. 問題を解決してから、アプリケーションを再始動します。

J2C 活動化仕様の削除

システムは、サービスを含むアプリケーションをインストールする際に、J2C 活動化仕様を作成します。アプリケーションを再インストールする前に、これらの仕様を削除しなければならない場合があります。

アプリケーションのインストールが失敗したために仕様を削除する場合は、Java Naming and Directory Interface (JNDI) 名のモジュールが、インストールに失敗したモジュールの名前と一致することを確認してください。JNDI 名の 2 番目の部分は、宛先をインプリメントしたモジュールの名前です。例えば sca/SimpleBOCrsmA/ActivationSpec では、**SimpleBOCrsmA** がモジュール名です。

サービスを含むアプリケーションのインストール後に不注意で構成を保管した場合、J2C 活動化仕様を必要としなければ、活動化仕様を削除します。

1. 削除する活動化仕様を見つけます。

   仕様はリソース・アダプター・パネルに含まれています。「**リソース**」>「**リソース・アダプター (Resource adapters)**」をクリックして、このパネルにナビゲートします。

   a. **Platform Messaging Component SPI Resource Adapter** を見つけます。

      このアダプターを見つけるには、スタンドアロン・サーバーの**ノード**・スコープまたは Network Deployment 環境の**サーバー**・スコープにいる必要があります。

2. Platform Messaging Component SPI Resource Adapter に関連付けられた J2C 活動化仕様を表示します。

   リソース・アダプター名をクリックすると、関連する仕様が次のパネルに表示されます。

3. 削除するモジュール名に一致する **JNDI 名**の仕様をすべて削除します。
   a. 該当する仕様の横のチェック・ボックスをクリックします。
   b. 「**削除**」をクリックします。

システムが、選択された仕様を表示から除去します。

変更を保管します。

SIBus 宛先の削除

SIBus 宛先とは、アプリケーションに対してサービスを使用可能にする接続です。宛先を除去しなければならない場合もあります。

アプリケーションのインストールが失敗したために宛先を削除する場合は、宛先名のモジュールが、インストールに失敗したモジュールの名前と一致することを確認してください。宛先の 2 番目の部分は、宛先をインプリメントしたモジュールの名

前です。例えば sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer では、**SimpleBOCrsmA** がモジュール名です。

サービスを含むアプリケーションのインストール後に不注意で構成を保管した場合、SIBus 宛先を必要としなくなったのであれば、その宛先を削除します。

注: このタスクは、SCA システム・バスからのみ宛先を削除します。また、サービスを含むアプリケーションを再インストールする前にも、アプリケーション・バスからエントリーを除去する必要があります。このインフォメーション・センターの『管理』セクションの『J2C 活動化仕様の削除』を参照してください。

1. 管理コンソールにログインします。
2. SCA システム・バスの宛先を表示します。

   「**サービス統合**」>「**バス**」をクリックしてパネルにナビゲートします。
3. SCA システム・バス宛先を選択します。

   表示の中で、**SCA.SYSTEM.*cellname*.Bus** をクリックします。ここで *cellname* は、削除する宛先を含むモジュールが含まれているセルの名前です。
4. 除去するモジュールに一致するモジュール名を含む宛先を削除します。
   a. 該当する宛先の横のチェック・ボックスをクリックします。
   b. 「**削除**」をクリックします。

パネルには残った宛先のみが表示されます。

これらの宛先を作成したモジュールに関連する J2C 活動化仕様を削除してください。

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

プログラミング・インターフェース情報がある場合、それらはこのプログラムを使用してアプリケーション・ソフトウェアを作成する際に役立つよう提供されています。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、IBM Corporation の商標です。IBM、 IBM LOGO、 AIX、 CICS、 Cloudscape、 DB2、 DB2 Connect、 DB2 Universal Database 、 developerWorks、 IMS、 Informix、 iSeries、 Lotus、 Lotus Domino、 MQSeries、 MVS、 OS/390、 Passport Advantage、 pSeries、 Rational 、 Redbooks、 Tivoli、 WebSphere、 z/OS、 zSeries

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

Microsoft および Windows は、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

この製品には、Eclipse Project (http://www.eclipse.org/) により開発されたソフトウェアが含まれています。

IBM Websphere Process Server for z/OS バージョン 6.0.1

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12