WebSphere®

IBM WebSphere Process Server for Multiplatforms

バージョン 7.0.0

モジュールの開発とデプロイ

IBM®

WebSphere®

IBM WebSphere Process Server for Multiplatforms

バージョン 7.0.0

モジュールの開発とデプロイ

IBM®

本書は、WebSphere Process Server for Multiplatforms バージョン 7、リリース 0、モディフィケーション 0 (製品番号 5724-L01)、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。

本書についてのご意見は、doc-comments@us.ibm.com へ E メールでお寄せください。皆様の率直なご意見をお待ちしています。

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。

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 Multiplatforms
Version 7.0.0
Developing and Deploying Modules

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2010.4

目次

表

第 1 部 アプリケーション開発

第 1 章 ビジネス・プロセス・マネジメント・ソリューションの開発

このセクションでは、ビジネス・プロセス・マネジメント (BPM) プログラミング・モデルの基礎について説明します。ここでは、Service Component Architecture (SCA) を紹介し、ビジネス・インテグレーションに関連するパターンについて説明します。

BPM とは、企業がビジネス・プロセスを識別して統合し、最適化することを可能にする 1 つの専門分野です。その目標は、生産性を向上させ、組織の有効性を最大限にすることです。企業の吸収合併が進み、ライブラリーのさまざまな情報資産が増えていくにつれ、BPM への関心が強くなってきています。多くの場合、これらの資産には一貫性がなく、資産間で調整が取れていないことから、「情報の孤島」が次第に増えていきます。

BPM は、サービス指向アーキテクチャー (SOA) と強く結び付いています。企業の性質、そしてどの程度の統合が必要かによって、BPM が IT 部門に課す要件は異なります。数少ない側面に対処すればいいだけのプロジェクトもあれば、これらの要件の多くを包含する大規模なプロジェクトもあります。BPM プロジェクトで最も一般的な側面のいくつかを以下に示します。

- **アプリケーションの統合**は、共通要件です。アプリケーション統合プロジェクトの複雑さは、場合によって異なります。単純な場合は、少数のアプリケーションが情報を確実に共用できるようにするだけでかまいません。一方、複雑な場合には、複数のバックエンド・アプリケーションで、トランザクションとデータ交換が同時に反映されるようにしなければなりません。複雑なアプリケーション統合には大抵、複雑な作業単位の管理、そして変換とマッピングが必要になります。
- **プロセスの自動化**は、個人または組織が行うアクティビティーによって、派生するアクティビティーが別の場所で体系的にトリガーされるようにする、もう 1 つの重要な側面です。これにより、ビジネス・プロセス全体が正常に完了することが確実になります。例えば、企業が従業員を雇用するときには、給与計算情報を更新する、セキュリティー部門が適切な措置を講じる、必要なツールを従業員に提供するなどの必要があります。プロセスのなかには、人間による入力と対話を取り込むアクティビティーもあれば、バックエンド・システムのスクリプトや、環境内のその他のサービスを呼び出すアクティビティーが含まれることもあります。
- **接続**は、抽象的ながらも企業だけでなく、ビジネス・パートナーの点から見ても重要な側面です。接続とは、組織間または企業間での情報のフローと、分散 IT サービスへのアクセスが可能であることの両方を指します。

ビジネス・インテグレーション実装におけるいくつかの技術的課題は、以下のように要約できます。

- フォーマットが異なるために効率的なデータ変換を不可能にしているデータに対処すること

- 異なるテクノロジーを使用して開発された IT サービスにアクセスするための各種プロトコルとメカニズムに対処すること
- 地理的に分散されていたり、異なる組織によって提供されている各種 IT サービスの調整を取ること
- 使用可能なサービスを分類および管理するルールとメカニズムを提供すること (ガバナンス)

以上のように、BPM には SOA にも共通する多くの主題と要素があります。IBM® の BPM の構想は、SOA で見られる基本的な概念の多くをベースに構築されています。この構想による直接の結果として、BPM ソリューションの実現には、さまざまな製品が必要となる可能性があります。IBM では、さまざまなステージと操作の側面すべてをサポートするツールとランタイム・プラットフォームを豊富に揃えています。

簡単に言い換えると、IBM の BPM の構想によって、企業は SOA IT インフラストラクチャーで実行されるアプリケーションを使用して、ビジネス・プロセスを定義、作成、マージ、統合、そして簡素化できるようになります。BPM の作業は、まさにロール・ベースです。マクロ・レベルでは、ビジネス・インテグレーションにはビジネス・プロセス・アプリケーションのモデル化、開発、ガバナンス、管理、およびモニターが必要です。適切なツールと手順の支援により、企業内外の人と異種のシステムが関係するビジネス・プロセスを自動化することが可能になります。BPM の主要な側面の 1 つは、効率的でスケーラブルかつ信頼性に優れ、変更に対処できるだけの柔軟性を持つようにビジネス・オペレーションを最適化できるかどうかです。

BPM には開発ツール、ランタイム・サーバー、モニター・ツール、サービス・リポジトリー、ツールキット、およびプロセス・テンプレートが必要になります。 BPM には多くの側面があるため、ソリューションを開発するには複数の開発ツールを使用する必要が出てきます。これらのツールにより、統合開発者が複雑なビジネス・ソリューションを組み立てることが可能になります。サーバーは、複雑なアプリケーションを実行するハイパフォーマンス・ビジネス・エンジンまたはサービス・コンテナーです。管理には常に、組織内で誰が何を行っているかを知る必要があります。そこで活躍するのが、モニター・ツールです。企業がこれらのビジネス・プロセスまたはサービスを作成するときには、サービスのガバナンス、分類、および保管が重要になってきます。この機能を提供するのが、サービス・リポジトリーです。既存のシステムに対するコネクターやアダプターなど、ソリューションの特化した部分を作成するために、特定のツールキットが必要になることもよくあります。

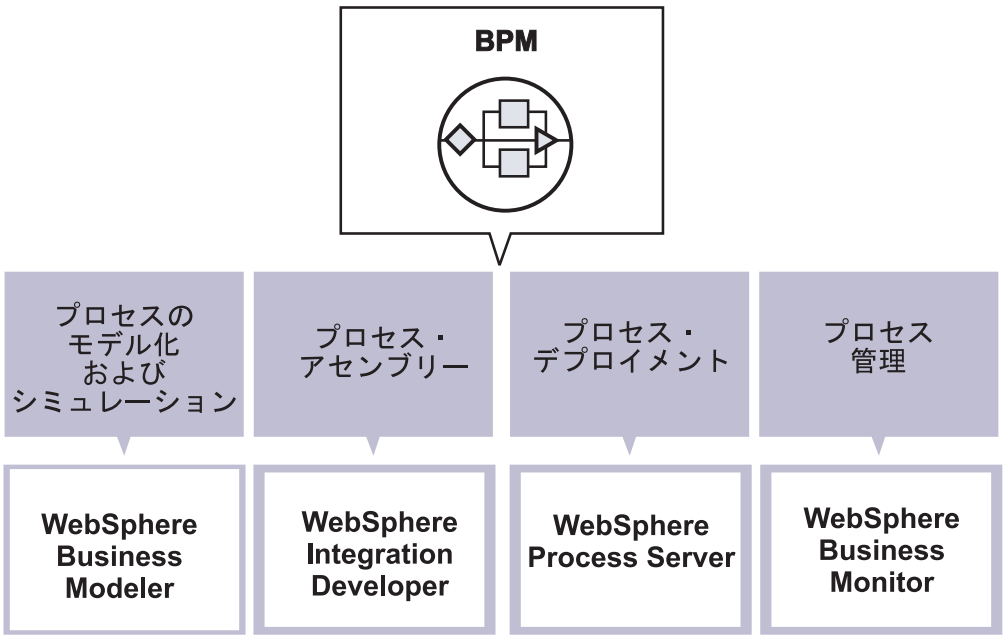


図 1. IBM ツールは BPM ライフ・サイクル全体を網羅するため、プロセスのモデル化からアセンブル、デプロイ、および管理までを行うことができます。

BPM は、単一の製品をベースとするわけではありません。ビジネス・インテグレーションには、ほぼすべての人、そして組織内および組織間のビジネスの側面すべてが関連します。 BPM には、SOA リファレンス・アーキテクチャー内のサービスとエレメントの多くが含まれます。

これらの概念の詳細およびプログラミング例については、以下を参照してください。

- 「*WebSphere® Business Integration Primer: Process Server, BPEL, SCA, and SOA*」(IBM Press、2008 年)
- 「*Getting Started with IBM WebSphere Process Server and IBM WebSphere Enterprise Service Bus Part 1: Development*」(IBM Redbooks®、SG24-7608-00、2008 年 6 月)
- 「*IBM Business Process Management Reviewer's Guide*」(IBM Redpapers、REDP-4433-01、2009 年 4 月)

ビジネス・インテグレーションのアーキテクチャーおよびパターン

標準的なビジネス管理プロジェクトでは、数種類の IT 資産を調整する必要があります。これらの IT 資産はおそらく異なるプラットフォームで実行されていて、開発されたタイミングも、開発に使用されたテクノロジーもさまざまに異なります。そのため、一連の多様なコンポーネントで、情報を簡単に操作して交換できるようにすることが、技術上の大きな課題となります。最適な対処方法は、ビジネス・インテグレーション・ソリューションの開発に使用されるプログラミング・モデルを使用することです。

このセクションでは、Service Component Architecture (SCA) を紹介し、ビジネス・インテグレーションに関連するパターンについて説明します。パターンは私たちの

生活に浸透しています。裁縫のパターン、子供たちの学習パターン、住宅建設のパターン、木彫パターン、飛行パターン、風のパターン、医療活動パターン、顧客の購買パターン、ワークフロー・パターン、情報科学のデザイン・パターンなど、パターンはさまざまなところに存在します。

パターンがソリューションの設計者と開発者に役立つことは実証されています。したがって、ビジネス・インテグレーションとエンタープライズ・インテグレーションにパターンを使用するのも当然です。要求と応答のルーティング・パターン、チャネル・パターン (パブリッシュ/サブスクライブなど) をはじめ、ビジネス・インテグレーションに適用できるパターンは幅広くあります。抽象パターンは特定の問題カテゴリーを解決する際のテンプレートとなる一方、具体的パターンは固有のソリューションを実装する方法を具体的に指示します。このセクションでは、データおよびサービス呼び出しに対処するパターンに焦点を当てます。WebSphere ビジネス・インテグレーションに対する IBM ソフトウェア戦略では、これらのパターンをプログラミング・モデルの基盤としています。

ビジネス・インテグレーションのシナリオ

企業はさまざまなソフトウェア・システムを使用してビジネスを運営します。さらに、これらのビジネス・コンポーネントは企業独自の方法で統合されます。

最も一般的なビジネス・インテグレーションのシナリオには、以下の 2 つがあります。

- **統合ブローカー**: このシナリオでは、ビジネス・インテグレーション・ソリューションが、各種バックエンド・アプリケーションの間に位置する仲介の役割を果たします。例えば、顧客がオンライン注文管理アプリケーションを使用して注文を行うと、トランザクションがカスタマー・リレーションシップ・マネージメント (CRM) バックエンドで関連する情報を更新するようにしなければなりません。このシナリオでは、統合ソリューションが注文管理アプリケーションから必要な情報を取り込み、場合によっては情報を変換した上で、CRM アプリケーションの適切なサービスを呼び出すことが可能でなければなりません。
- **プロセス自動化**: このシナリオでは、統合ソリューションが関連付けられていない異なる IT サービス間のグルーとしての役割を果たします。例えば、企業が従業員を雇用する場合、以下の一連のアクションが発生する必要があります。
  - 従業員の情報を給与計算システムに追加します。
  - 従業員に設備への物理アクセスを認可し、バッジを提供する必要があります。
  - 企業が一連の物理資産 (オフィスのスペース、コンピューターなど) を従業員に割り当てる必要がある場合もあります。
  - IT 部門が従業員のユーザー・プロファイルを作成し、一連のアプリケーションへのアクセスを認可する必要があります。

    このプロセスの自動化も、ビジネス・インテグレーションのシナリオでは一般的なユース・ケースです。このシナリオでは、ソリューションが実装する自動フローは、給与計算システムへの従業員の追加によってトリガーされます。続いて、このフローはアクションを実行する担当者の作業項目を作成するか、または適切なサービスを呼び出すことによって、他のステップをトリガーします。

いずれのシナリオでも、統合ソリューションは以下のことを行う必要があります。

1. さまざまな情報ソースと各種のデータ・フォーマットを操作し、情報のフォーマット変換を行えるようにする。
2. 異なる呼び出しメカニズムとプロトコルを使用する各種のサービスを呼び出せるようにする。

役割、製品、および技術に関する課題

ビジネス・インテグレーション・プロジェクトの成功は、特化した開発役割、プログラミング手法、およびツール・スイートの組み合わせによって左右されます。

ビジネス・インテグレーション・プロジェクトには、以下の基本要素が必要です。

- 開発組織内での明確な役割の分離。通常、役割を明確に分離して特殊化を促進することで、開発される個々のコンポーネントの品質が改善されます。
- 共通ビジネス・オブジェクト (BO) モデル。共通の論理モデルでビジネス情報を表現することを可能にします。
- プログラミング・モデル。インターフェースを実装から明確に分離するとともに、実装とは完全に独立し、インターフェースの操作だけが必要な汎用サービス呼び出しメカニズムをサポートします。
- 統合されたツールと製品のセット。開発役割をサポートし、それぞれの役割の分離を維持します。

以降のセクションでは、上記の構成要素のそれぞれについて詳しく説明します。

明確な役割の分離

ビジネス・インテグレーション・プロジェクトには、互いに協調しながらも明確に分離した役割を持つ担当者が必要です。これらの役割には以下があります。

- **ビジネス・アナリスト**: ビジネス・アナリストはドメイン・エキスパートとして、プロセスのビジネスの側面を捉え、プロセス自体を的確に表現するプロセス・モデルを作成します。この役割が焦点とするのは、プロセスの財務的パフォーマンスを最適化することです。ビジネス・アナリストは、プロセス実装の技術的な側面には関与しません。
- **コンポーネント開発者**: コンポーネント開発者は、個々のサービスとコンポーネントの実装を担当します。この役割が焦点とするのは、実装に使用する特定のテクノロジーです。この役割には、プログラミングの経歴を積んでいることが必要となります。
- **統合スペシャリスト**: これは比較的新しい役割で、この役割が表すのは、既存の一連のコンポーネントをまとめて 1 つの大きなビジネス・インテグレーション・ソリューションにアセンブルするという職務を課せられた担当者です。統合開発者は、再使用およびワイヤリングするコンポーネントやサービスそれぞれの技術詳細を知る必要はありません。理想的には、統合開発者は、アセンブルしているサービスのインターフェースについて理解することだけに専念します。統合開発者は、アセンブル・プロセス用の統合ツールに依存する必要があります。
- **ソリューション・デプロイヤー**: ソリューション・デプロイヤーおよび管理者が対象とするのは、エンド・ユーザーが使用可能なビジネス・インテグレーション・ソリューションの作成です。理想的には、ソリューション・デプロイヤーは主にソリューションを作動準備の整った物理リソース (データベース、キュー・

マネージャーなど) にバインドすることに専念し、ソリューションの内部構造を深く理解することはしません。ソリューション・デプロイヤーが重点を置くのは、サービスの品質 (QoS) です。

共通ビジネス・オブジェクト・モデル

前述のとおり、ビジネス・インテグレーション・プロジェクトの重要な側面には、複数のコンポーネントの呼び出しを調整できること、そしてこれらのコンポーネントの間でのデータ交換を処理できることが含まれます。これは特に、複数のコンポーネントがそれぞれに異なる手法で、注文や顧客情報に含まれるデータなどのビジネス・アイテムを表現する可能性があるからです。例えば、Enterprise Java™Bean (EJB) エンティティーを使用してビジネス・アイテムを表す Java アプリケーションと、COBOL コピーブックで情報を編成しているレガシー・アプリケーションを統合しなければならない場合も考えられます。そのため、統合ソリューションの作成を単純化することを意図するプラットフォームは、バックエンド・システムがデータ処理に使用する手法とは関係なくビジネス・アイテムを表現する汎用方法も提供しなければなりません。この目標は、WebSphere Process Server および WebSphere Enterprise Service Bus では、*ビジネス・オブジェクト・フレームワーク* によって達成されています。

ビジネス・オブジェクト・フレームワークは、開発者が XML スキーマを使用してビジネス・データの構造を定義し、これらのデータ構造 (ビジネス・オブジェクト) のインスタンスに XPath または Java コードでアクセスして操作することを可能にします。ビジネス・オブジェクト・フレームワークは、サービス・データ・オブジェクト (SDO) 標準に基づきます。

Service Component Architecture (SCA) プログラミング・モデル

SCA プログラミング・モデルは、WebSphere Process Server および WebSphere Enterprise Service Bus で開発されるすべてのソリューションの基盤となります。SCA は、開発者にサービス実装を再使用可能コンポーネントにカプセル化する方法を提供します。SCA を使用すると、開発者は使用するテクノロジーに依存しない方法で、インターフェース、実装、および参照を定義できます。この手法により、選択したテクノロジーの種類を問わず、要素をバインドする機会が得られます。また、これらのコンポーネントの呼び出しを可能にする SCA クライアント・プログラミング・モデルもあります。このモデルは具体的には、Java ベースのランタイム・インフラストラクチャーが Java 以外のランタイムと対話することを可能にします。SCA は、サービスを呼び出すためにビジネス・オブジェクトをデータ項目として使用します。

ツールおよび製品

IBM WebSphere Integration Developer は、上述のテクノロジーをベースとしたビジネス・インテグレーション・ソリューションを作成および構成するために必要なすべてのツールを備えた統合開発環境です。これらのソリューションは通常、WebSphere Process Server にデプロイされますが、場合によっては WebSphere Enterprise Service Bus にデプロイされることもあります。

第 2 章 バインディング

サービス指向アーキテクチャーの中核をなすのは「サービス」の概念で、これはコンピューター・デバイス間での対話によって実行される機能単位のことです。「エクスポート」はモジュールの外部インターフェース (またはアクセス・ポイント) を定義し、これによってモジュール内の Service Component Architecture (SCA) コンポーネントは外部クライアントにサービスを提供できます。「インポート」はモジュール外部のサービスへのインターフェースを定義し、これによってサービスはモジュール内部から呼び出すことができます。インポートおよびエクスポートとともにプロトコル固有の「バインディング」を使用して、データをモジュールの内部または外部に移送する方法を指定します。

エクスポート

外部クライアントがインテグレーション・モジュール内の SCA コンポーネントを呼び出すとき、さまざまなプロトコル (HTTP、JMS、MQ、および RMI/IIOP など) を経由して、さまざまなデータ・フォーマット (XML、CSV、COBOL、および JavaBean など) が使用されます。エクスポートはこれらの要求を外部ソースから受信し、SCA プログラミング・モデルを使用して WebSphere Process Server コンポーネントを呼び出すコンポーネントです。

例えば以下の図で、エクスポートはクライアント・アプリケーションから HTTP プロトコルを経由して要求を受信します。データは SCA コンポーネントによって使用されるフォーマットであるビジネス・オブジェクトに変換されます。そのコンポーネントはそのデータ・オブジェクトによって呼び出されます。

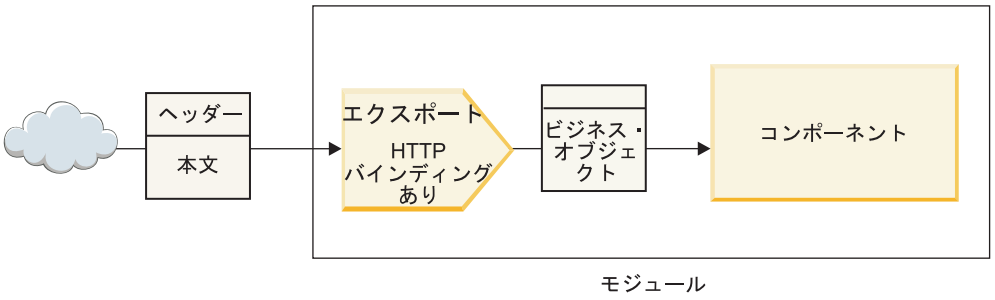


図 2. HTTP バインディングを持つエクスポート

インポート

SCA コンポーネントから、SCA とは異なるフォーマットのデータを要求する SCA 以外の外部サービスを呼び出すことが必要な場合があります。インポートは SCA プログラミング・モデルを使用して外部サービスを呼び出すために、SCA コンポーネントによって使用されます。インポートは次に、ターゲット・サービスが要求する方法でサービスを呼び出します。

例えば以下の図で、SCA コンポーネントからの要求がインポートによって外部サービスに送信されます。SCA コンポーネントによって使用されるフォーマットである

ビジネス・オブジェクトは、サービスが要求するフォーマットに変換され、サービスが呼び出されます。

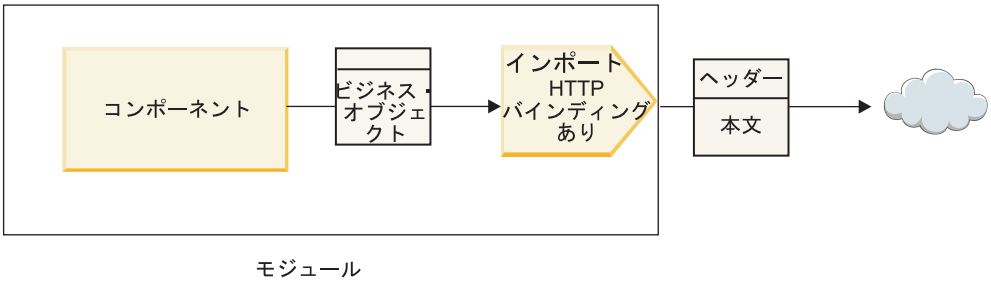


図 3. HTTP バインディングを持つインポート

バインディングのリスト

インポートまたはエクスポートのバインディングを生成して、バインディングを構成するには、WebSphere Integration Developer を使用します。使用可能なバインディングのタイプを以下の一覧に記載します。

- SCA

  SCA バインディングはデフォルトのバインディングで、使用するサービスを他の SCA モジュールのサービスと通信させることができます。インポートと SCA バインディングを使用して、別の SCA モジュールのサービスにアクセスします。エクスポートと SCA バインディングを使用して、別の SCA モジュールにサービスを提供します。

- Web サービス

  Web サービス・バインディングを使用すれば、相互運用可能な SOAP メッセージおよびサービス品質を使用して外部サービスにアクセスすることができます。Web サービス・バインディングを使用すると、添付ファイルを SOAP メッセージの一部として組み込むこともできます。

  Web サービス・バインディングでは SOAP/HTTP (SOAP over HTTP) または SOAP/JMS (SOAP over JMS) のいずれかのトランスポート・プロトコルを使用できます。SOAP メッセージの伝達に使うトランスポート (HTTP または JMS) に関係なく、Web サービス・バインディングは常に要求/応答の対話を同期的に処理します。

- HTTP

  HTTP バインディングを使用すれば、SOAP 以外のメッセージが使用されている場合、または直接 HTTP アクセスが必要な場合に、HTTP プロトコルを使用して外部サービスにアクセスすることができます。このバインディングは、HTTP モデルに基づく Web サービス (GET、PUT、DELETE などのよく知られた HTTP インターフェース操作を使用するサービス) を処理するときに使用されます。

- Enterprise JavaBeans™ (EJB)

  EJB バインディングにより、SCA コンポーネントは、Java EE サーバー上で実行される Java EE ビジネス・ロジックで提供されるサービスと対話できるようになります。

- EIS

  EIS (エンタープライズ情報システム) バインディングを JCA リソース・アダプターと一緒に使用すると、エンタープライズ情報システム上のサービスにアクセスしたり、サービスを EIS で使用可能にすることができます。
- JMS バインディング

  Java Message Service (JMS)、汎用 JMS、および WebSphere MQ JMS (MQ JMS) バインディングは、メッセージ・キューを経由した非同期通信が信頼性の維持に欠かせない場合に、メッセージング・システムとの対話に使用されます。

  これらの JMS バインディングのいずれかを使用するエクスポートは、キューにメッセージが到着するのを監視し、応答 (該当する場合) を応答キューに非同期に送信します。これらの JMS バインディングのいずれかを使用するインポートは、メッセージを構築して JMS キューに送信し、キューに応答 (該当する場合) が到着するのを監視します。

  – JMS

    JMS バインディングを使用すれば、WebSphere 組み込み JMS プロバイダーにアクセスすることができます。

  – 汎用 JMS

    汎用 JMS バインディングでは、IBM 以外のベンダーのメッセージング・システムにアクセスできます。

  – MQ JMS

    MQ JMS バインディングを使用すれば、WebSphere MQ メッセージング・システムの JMS サブセットにアクセスできます。ご使用のアプリケーションにとって JMS サブセットの機能で十分な場合、このバインディングを使用します。
- MQ

  WebSphere MQ バインディングにより、MQ ネイティブ・アプリケーションと通信できるようになるため、これらのアプリケーションがサービス指向アーキテクチャー・フレームワークに組み込まれ、MQ 固有のヘッダー情報にアクセスできるようになります。MQ のネイティブ機能を使用する必要がある場合に、このバインディングを使用します。

エクスポートおよびインポート・バインディングの概要

エクスポートによってインテグレーション・モジュール内のサービスが外部クライアントに使用可能となり、インポートによってインテグレーション・モジュール内の SCA コンポーネントが外部サービスを呼び出すことができるようになります。エクスポートまたはインポートに関連付けられたバインディングは、プロトコル・メッセージとビジネス・オブジェクトとの関係を指定します。また、操作と障害を選択する方法も指定します。

エクスポートによる情報のフロー

エクスポートは、エクスポートがワイヤリングされたコンポーネントを対象とする要求を受信します。要求の受信に使用される特定のトランスポート (例えば、HTTP) は、エクスポートに関連付けられたバインディングによって決まります。

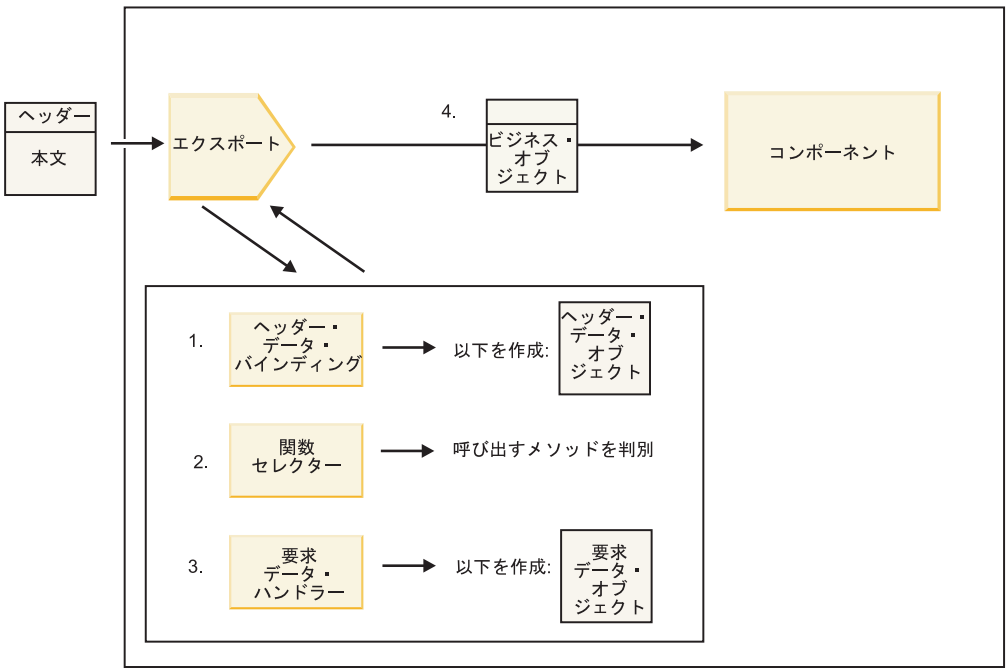


図 4. エクスポートを経由したコンポーネントへの要求のフロー

エクスポートが要求を受信するとき、以下の一連のイベントが発生します。

1. WebSphere MQ バインディングの場合に限り、ヘッダー・データ・バインディングがプロトコル・ヘッダーをヘッダー・データ・オブジェクトに変換します。
2. 関数セレクターがプロトコル・メッセージからネイティブ・メソッド名を判別します。ネイティブ・メソッド名は、エクスポート構成により、エクスポートのインターフェース上の操作名にマップされています。
3. メソッドの要求データ・ハンドラーまたは要求データ・バインディングは、要求を要求ビジネス・オブジェクトに変換します。
4. エクスポートは要求ビジネス・オブジェクトでコンポーネント・メソッドを呼び出します。
   - HTTP エクスポート・バインディング、Web サービス・エクスポート・バインディング、および EJB エクスポート・バインディングは、SCA コンポーネントに対して同期呼び出しを行います。
   - JMS、汎用 JMS、MQ JMS、および WebSphere MQ エクスポート・バインディングは、SCA コンポーネントに対して非同期呼び出しを行います。

エクスポートは、コンテキスト伝搬が有効になっている場合、プロトコル経由で受信したヘッダーおよびユーザー・プロパティーを伝搬できます。その後、エクスポートにワイヤリングされたコンポーネントは、これらのヘッダーおよびユーザー・プロパティーにアクセスできます。詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『伝搬』を参照してください。

これが両方向操作の場合、コンポーネントは応答を返します。

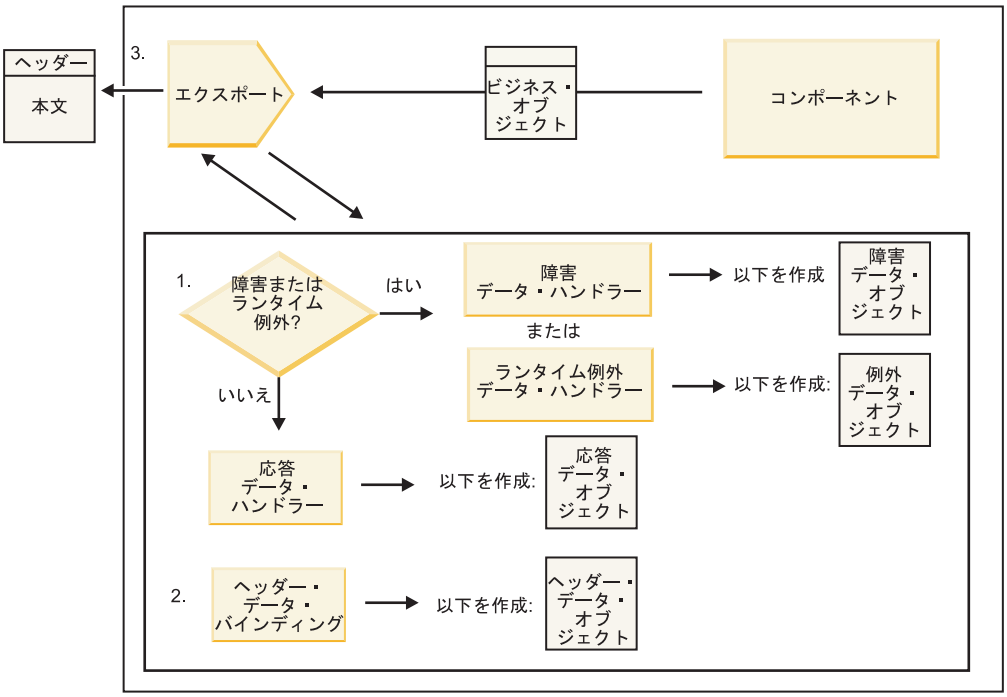


図 5. エクスポートを経由して戻る応答のフロー

以下の一連のステップが発生します。

1. 通常の応答メッセージがエクスポート・バインディングによって受信されると、メソッドの応答データ・ハンドラーまたは応答データ・バインディングは、ビジネス・オブジェクトを応答に変換します。

   応答が障害の場合、メソッドの障害データ・ハンドラーまたは障害データ・バインディングは障害を障害応答に変換します。

   HTTP エクスポート・バインディングにのみ適用されることですが、応答がランタイム例外の場合、ランタイム例外データ・ハンドラー (構成済みの場合) が呼び出されます。
2. WebSphere MQ バインディングの場合に限り、ヘッダー・データ・バインディングがヘッダー・データ・オブジェクトをプロトコル・ヘッダーに変換します。
3. エクスポートはサービス応答をトランスポート経由で送信します。

インポートによる情報のフロー

コンポーネントはインポートを使用して、モジュール外部のサービスに要求を送信します。要求は、関連付けられたバインディングによって決まる特定のトランスポートを使用して送信されます。

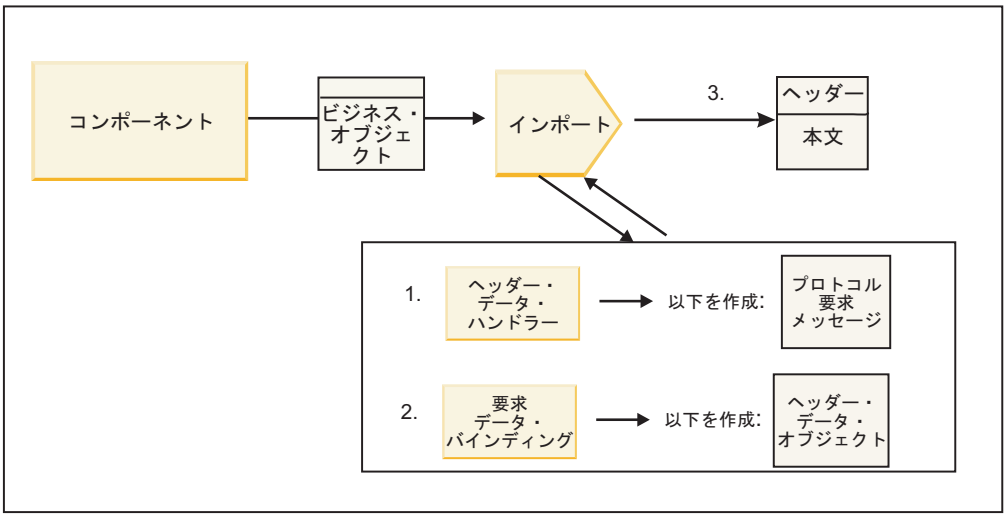


図 6. インポートを経由したコンポーネントからサービスへのフロー

コンポーネントは、要求ビジネス・オブジェクトでインポートを呼び出します。

注:

- HTTP インポート・バインディング、Web サービス・インポート・バインディング、および EJB インポート・バインディングは、呼び出し側コンポーネントが同期方式で呼び出す必要があります。
- JMS、汎用 JMS、MQ JMS、および WebSphere MQ インポート・バインディングは、非同期方式で呼び出す必要があります。

コンポーネントがインポートを呼び出した後、以下の一連のイベントが発生します。

1. メソッドの要求データ・ハンドラーまたは要求データ・バインディングは、要求ビジネス・オブジェクトをプロトコル要求メッセージに変換します。
2. WebSphere MQ バインディングの場合に限り、メソッドのヘッダー・データ・バインディングがヘッダー・ビジネス・オブジェクトをプロトコル・ヘッダーに設定します。
3. インポートはトランスポートを経由して、サービス要求でサービスを呼び出します。

これが両方向操作の場合、サービスは応答を戻し、以下の一連のステップが発生します。

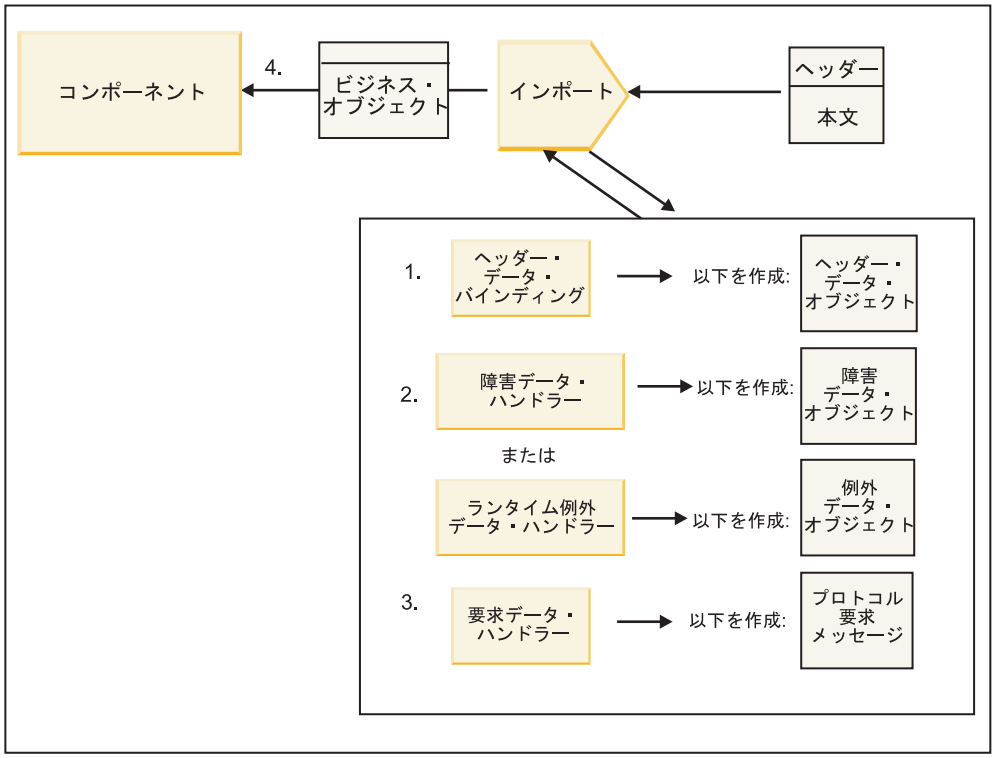


図 7. インポートを経由して戻る応答のフロー

1. WebSphere MQ バインディングの場合に限り、ヘッダー・データ・バインディングがプロトコル・ヘッダーをヘッダー・データ・オブジェクトに変換します。
2. 応答が障害かどうかを判別します。
   - 応答が障害の場合、障害セレクターは障害を検査して、どの WSDL 障害にマップするかを決定します。次にメソッドの障害データ・ハンドラーは、障害を障害応答に変換します。
   - 応答がランタイム例外の場合、ランタイム例外データ・ハンドラー (構成済みの場合) が呼び出されます。
3. メソッドの応答データ・ハンドラーまたは応答データ・バインディングは、応答を応答ビジネス・オブジェクトに変換します。
4. インポートは応答ビジネス・オブジェクトをコンポーネントに返します。

エクスポートおよびインポート・バインディング構成

エクスポートおよびインポート・バインディングの重要な側面の 1 つはデータ・フォーマットの変換です。これによって、データをネイティブ・ワイヤー・フォーマットからビジネス・オブジェクトにマップ (非直列化) する方法、またはビジネス・オブジェクトからネイティブ・ワイヤー・フォーマットにマップ (直列化) する方法が指定されます。 エクスポートに関連したバインディングでは、データに対して実行すべき操作を指示する関数セレクターも指定できます。エクスポートまたはインポートに関連したバインディングでは、処理中に発生する障害の処理方法を指示することができます。

さらに、トランスポート固有情報をバインディングに指定できます。例えば HTTP バインディングでは、エンドポイント URL を指定します。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。例えば HTTP バインディングの場合、トランスポート固有の情報は『HTTP インポート・バインディングの生成』トピックと『HTTP エクスポート・バインディングの生成』に記載されています。

インポートおよびエクスポートでのデータ・フォーマット変換

WebSphere Integration Developer でエクスポートまたはインポート・バインディングを構成するときには、構成プロパティーの 1 つとしてバインディングで使用するデータ・フォーマットを指定します。

- エクスポート・バインディングでは、クライアント・アプリケーションが要求を SCA コンポーネントに送信し、応答を SCA コンポーネントから受信しますが、この場合、ネイティブ・データのフォーマットを指示します。システムはこのフォーマットに基づいて、ネイティブ・データからビジネス・オブジェクト (SCA コンポーネントによって使用される) に変換し、逆にビジネス・オブジェクトをネイティブ・データ (クライアント・アプリケーションへの応答) に変換するための適切なデータ・ハンドラーまたはデータ・バインディングを選択します。
- インポート・バインディングでは、SCA コンポーネントが要求をモジュール外部のサービスに送信し、応答をモジュール外部のサービスから受信しますが、この場合、ネイティブ・データのフォーマットを指示します。システムはこのフォーマットに基づいて、ビジネス・オブジェクトからネイティブ・データへ、およびその逆に変換するための適切なデータ・ハンドラーまたはデータ・バインディングを選択します。

WebSphere Process Server には事前定義された一連のデータ・フォーマットと、そのフォーマットをサポートする対応データ・ハンドラーまたはデータ・バインディングが提供されています。また、固有のカスタム・データ・ハンドラーを作成して、そのデータ・ハンドラーのデータ・フォーマットを登録することもできます。詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『データ・ハンドラーの開発』を参照してください。

- データ・ハンドラーはプロトコルに中立的で、1 つのフォーマットから別のフォーマットにデータを変換します。WebSphere Process Server では、データ・ハンドラーはネイティブ・データ (XML、CSV、および COBOL) からビジネス・オブジェクトへ、およびビジネス・オブジェクトからネイティブ・データへの変換を行うのが標準的です。これらはプロトコルに中立的なため、いろいろなエクスポートおよびインポート・バインディングで同じデータ・ハンドラーを再利用することができます。例えば、同一の XML データ・ハンドラーを HTTP エクスポートまたはインポート・バインディング、あるいは JMS エクスポートまたはインポート・バインディングで使用できます。
- データ・バインディングもネイティブ・データからビジネス・オブジェクトへの変換 (およびその逆) を行いますが、これらはプロトコルに固有です。例えば、HTTP データ・バインディングは HTTP エクスポートまたは HTTP インポート・バインディングのみで使用できます。データ・ハンドラーとは異なり、HTTP データ・バインディングは MQ エクスポートまたは MQ インポート・バインディングでは再利用できません。

注: 3 つの HTTP データ・バインディング (HTTPStreamDataBindingSOAP、HTTPStreamDataBindingXML、および HTTPServiceGatewayDataBinding) は、WebSphere Process Server バージョン 7.0 では推奨されていません。可能な場合はデータ・ハンドラーを使用してください。

前に述べたように、必要に応じてカスタム・データ・ハンドラーを作成することができます。また、カスタム・データ・バインディングを作成することもできますが、カスタム・データ・ハンドラーは複数のバインディングで使用できるため、カスタム・データ・ハンドラーを作成することをお勧めします。

データ・ハンドラー

データ・ハンドラーは、プロトコルに中立的な方法でデータ・フォーマットを変換するために、エクスポートおよびインポート・バインディングに対して構成されます。いくつかのデータ・ハンドラーが製品の一部として提供されていますが、必要に応じて固有のデータ・ハンドラーを作成することもできます。データ・ハンドラーをエクスポート・バインディングまたはインポート・バインディングに関連付けるレベルは 2 つあります。一方のレベルでは、エクスポートまたはインポートのインターフェースにあるすべての操作に関連付けることが可能です。もう一方のレベルでは、要求または応答に対する特定の操作に関連付けることができます。

事前定義データ・ハンドラー

使用するデータ・ハンドラーの指定は、WebSphere Integration Developer を使用して行います。

使用できるように事前定義されたデータ・ハンドラーを以下の表で示します。表には、各データ・ハンドラーがインバウンドおよびアウトバウンドのデータを変換する方法についても記載されています。

注: 注記されていない限り、これらのデータ・ハンドラーは、JMS、汎用 JMS、MQ JMS、WebSphere MQ、および HTTP バインディングで使用できます。
詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『データ・ハンドラー』を参照してください。

表 1. 事前定義データ・ハンドラー

データ・ハンドラー	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
ATOM	ATOM フィードを ATOM フィード・ビジネス・オブジェクトに解析します。	ATOM フィード・ビジネス・オブジェクトを ATOM フィードに直列化します。
Delimited	区切り文字で区切られているデータをビジネス・オブジェクトに解析します。	ビジネス・オブジェクトを区切り文字で区切られているデータ (CSV など) に直列化します。
Fixed Width	固定長データをビジネス・オブジェクトに解析します。	ビジネス・オブジェクトを固定長データに直列化します。

表 1. 事前定義データ・ハンドラー (続き)

データ・ハンドラー	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
WTX により処理	データ・フォーマット変換を WebSphere Transformation Extender (WTX) に委任します。WTX マップ名は、データ・ハンドラーから派生します。	データ・フォーマット変換を WebSphere Transformation Extender (WTX) に委任します。WTX マップ名は、データ・ハンドラーから派生します。
WTX 呼び出し側により処理	データ・フォーマット変換を WebSphere Transformation Extender (WTX) に委任します。WTX マップ名は、ユーザーが提供します。	データ・フォーマット変換を WebSphere Transformation Extender (WTX) に委任します。WTX マップ名は、ユーザーが提供します。
JAXB	Java Architecture for XML Binding (JAXB) 仕様で定義されているマッピング・ルールを使用して、Java Bean をビジネス・オブジェクトに直列化します。	JAXB 仕様で定義されているマッピング・ルールを使用して、ビジネス・オブジェクトを Java Bean に非直列化します。
JAXWS **注:** JAXWS データ・ハンドラーは、EJB バインディングでのみ使用できます。	Java API for XML Web Services (JAX-WS) 仕様で定義されているマッピング・ルールを使用して、応答 Java オブジェクトまたは例外 Java オブジェクトを応答ビジネス・オブジェクトに変換するために EJB バインディングによって使用されます。	JAX-WS 仕様で定義されているマッピング・ルールを使用して、ビジネス・オブジェクトを発信 Java メソッド・パラメーターに変換するために EJB バインディングによって使用されます。
JSON	JSON データをビジネス・オブジェクトに解析します。	ビジネス・オブジェクトを JSON データに直列化します。
ネイティブ本体	ネイティブのバイト、テキスト、マップ、ストリーム、またはオブジェクトを 5 つの基本ビジネス・オブジェクト (テキスト、バイト、マップ、ストリーム、またはオブジェクト) のいずれかに解析します。	5 つの基本ビジネス・オブジェクトをバイト、テキスト、マップ、ストリーム、またはオブジェクトに変換します。
SOAP	SOAP メッセージ (およびヘッダー) をビジネス・オブジェクトに解析します。	ビジネス・オブジェクトを SOAP メッセージに直列化します。
XML	XML データをビジネス・オブジェクトに解析します。	ビジネス・オブジェクトを XML データに直列化します。

表 1. 事前定義データ・ハンドラー (続き)

データ・ハンドラー	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
UTF8XMLDataHandler	UTF-8 にエンコードされた XML データをビジネス・オブジェクトに解析します。	メッセージの送信時に、ビジネス・オブジェクトを UTF-8 にエンコードされた XML データに直列化します。

データ・ハンドラーの作成

データ・ハンドラーの作成に関する詳細情報については、WebSphere Integration Developer インフォメーション・センターのトピック『データ・ハンドラーの開発』を参照してください。

データ・バインディング

データ・バインディングは、データ・フォーマットを変換するために、エクスポートおよびインポート・バインディングに対して構成されます。データ・バインディングは、それぞれのプロトコルに固有です。いくつかのデータ・バインディングが製品の一部として提供されていますが、必要に応じて固有のデータ・バインディングを作成することもできます。データ・バインディングをエクスポートまたはインポート・バインディングに関連付けるレベルは 2 つあります。一方のレベルでは、エクスポートまたはインポートのインターフェースにあるすべての操作に関連付けることが可能です。もう一方のレベルでは、要求または応答に対する特定の操作に関連付けることができます。

使用するデータ・バインディングの指定または固有のデータ・バインディングの作成は、WebSphere Integration Developer を使用して行います。データ・バインディングの作成については、WebSphere Integration Developer インフォメーション・センターの『JMS、MQ JMS、および汎用 JMS データ・バインディングの概要』セクションを参照してください。

JMS バインディング

次の表は、以下のバインディングで使用できるデータ・バインディングの一覧です。

- JMS バインディング
- 汎用 JMS バインディング
- WebSphere MQ JMS バインディング

また表には、データ・バインディングが実行するタスクの説明も記載されています。

表 2. JMS バインディング用の事前定義データ・バインディング

データ・バインディング	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
直列化 Java オブジェクト	Java 直列化オブジェクトをビジネス・オブジェクト (WSDL で入力または出力タイプとしてマップされているビジネス・オブジェクト) に変換します。	ビジネス・オブジェクトを JMS オブジェクト・メッセージの Java 直列化オブジェクトに直列化します。
ラップされたバイト	着信 JMS バイト・メッセージからバイトを抽出して、これらを JMSBytesBody ビジネス・オブジェクトにラップします。	JMSBytesBody ビジネス・オブジェクトからバイトを抽出して、これらを発信 JMS バイト・メッセージにラップします。
ラップされたマップ項目	着信 JMS マップ・メッセージの各項目の名前、値、およびタイプ情報を抽出して、MapEntry ビジネス・オブジェクトのリストを作成します。次に、そのリストを JMSMapBody ビジネス・オブジェクトにラップします。	JMSMapBody ビジネス・オブジェクトの MapEntry リストから名前、値、およびタイプ情報を抽出して、発信 JMS マップ・メッセージ内に対応する項目を作成します。
ラップされたオブジェクト	着信 JMS オブジェクト・メッセージからオブジェクトを抽出して、これを JMSObjectBody ビジネス・オブジェクトにラップします。	JMSObjectBody ビジネス・オブジェクトからオブジェクトを抽出して、これを発信 JMS オブジェクト・メッセージにラップします。
ラップされたテキスト	着信 JMS テキスト・メッセージからテキストを抽出して、これを JMSTextBody ビジネス・オブジェクトにラップします。	JMSTextBody ビジネス・オブジェクトからテキストを抽出して、これを発信 JMS テキスト・メッセージにラップします。

WebSphere MQ バインディング

次の表は、WebSphere MQ で使用できるデータ・バインディングの一覧と、データ・バインディングが実行するタスクを説明したものです。

表 3. WebSphere MQ バインディング用の事前定義データ・バインディング

データ・バインディング	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
直列化 Java オブジェクト	着信メッセージからの Java 直列化オブジェクトをビジネス・オブジェクト (WSDL で入力または出力タイプとしてマップされているビジネス・オブジェクト) に変換します。	ビジネス・オブジェクトを出力メッセージの Java 直列化オブジェクトに変換します。

表 3. WebSphere MQ バインディング用の事前定義データ・バインディング (続き)

データ・バインディング	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
ラップされたバイト	構造化されていない MQ バイト・メッセージからバイトを抽出して、これらを JMSBytesBody ビジネス・オブジェクトにラップします。	JMSBytesBody ビジネス・オブジェクトからバイトを抽出して、バイトを構造化されていない発信 MQ バイト・メッセージにラップします。
ラップされたテキスト	構造化されていない MQ テキスト・メッセージからテキストを抽出して、これを JMSTextBody ビジネス・オブジェクトにラップします。	JMSTextBody ビジネス・オブジェクトからテキストを抽出して、これを構造化されていない MQ テキスト・メッセージにラップします。
ラップされたストリーム項目	着信 JMS ストリーム・メッセージの各項目の名前およびタイプ情報を抽出して、StreamEntry ビジネス・オブジェクトのリストを作成します。次に、そのリストを JMSStreamBody ビジネス・オブジェクトにラップします。	JMSStreamBody ビジネス・オブジェクトの StreamEntry リストから名前およびタイプ情報を抽出して、発信 JMSStreamMessage 内に対応する項目を作成します。

20 ページの表 3 にリストされたデータ・バインディングの他、WebSphere MQ はヘッダー・データ・バインディングも使用します。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。

HTTP バインディング

次の表は、HTTP で使用できるデータ・バインディングの一覧と、データ・バインディングが実行するタスクを説明したものです。

表 4. HTTP バインディング用の事前定義データ・バインディング

データ・バインディング	ネイティブ・データからビジネス・オブジェクトへ	ビジネス・オブジェクトからネイティブ・データへ
ラップされたバイト	着信 HTTP メッセージの本体からバイトを抽出して、これらを HTTPBytes ビジネス・オブジェクトにラップします。	HTTPBytes ビジネス・オブジェクトからバイトを抽出して、これらを発信 HTTP メッセージの本体に追加します。
ラップされたテキスト	着信 HTTP メッセージの本体からテキストを抽出して、これを HTTPText ビジネス・オブジェクトにラップします。	HTTPText ビジネス・オブジェクトからテキストを抽出して、これを発信 HTTP メッセージの本体に追加します。

エクスポート・バインディングでの関数セレクター

関数セレクターは、要求メッセージのデータに対して実行すべき操作を指示するために使用します。関数セレクターは、エクスポート・バインディングの一部として構成されます。

インターフェースを公開する SCA エクスポートを例として考えてみます。このインターフェースには、作成および更新の 2 つの操作が組み込まれています。エクスポートの JMS バインディングは、キューから読み取られます。

メッセージがキューに到着すると、エクスポートには関連するデータが渡されますが、ワイヤリングされたコンポーネントで、エクスポートのインターフェースからどの操作を呼び出すべきかの情報は渡されません。その操作は、関数セレクターとエクスポート・バインディング構成によって決まります。

関数セレクターはネイティブの関数名 (メッセージを送信したクライアント・システムの関数名) を返します。ネイティブの関数名は、エクスポートが関連付けられたインターフェースの操作名または関数名にマップされます。例えば以下の図で、関数セレクターは着信メッセージからネイティブの関数名 (CRT) を返し、ネイティブの関数名は作成操作にマップされ、ビジネス・オブジェクトは作成操作を持つ SCA コンポーネントに送信されます。

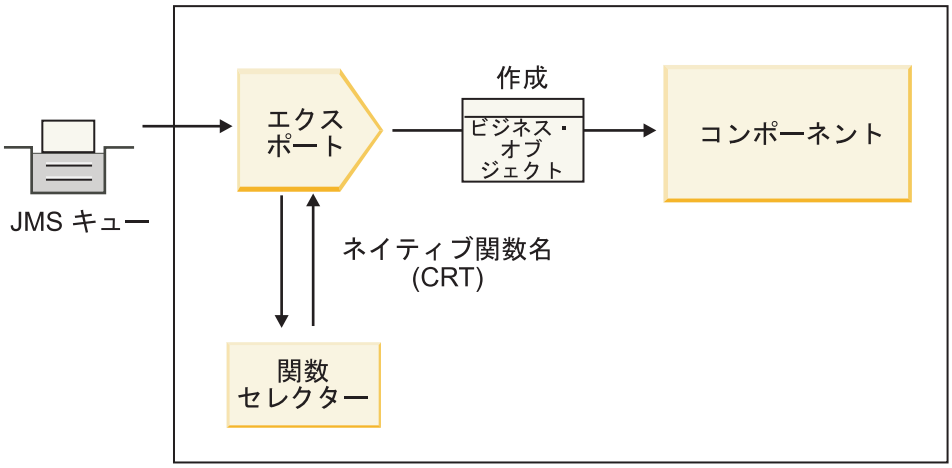


図 8. 関数セレクター

インターフェースに 1 つの操作しかない場合には、関数セレクターを指定する必要はありません。

複数の関数セレクターがプリパッケージされています。以降のセクションで、これらの関数セレクターをリストします。

JMS バインディング

次の表は、以下のバインディングで使用できる関数セレクターの一覧です。

- JMS バインディング
- 汎用 JMS バインディング
- WebSphere MQ JMS バインディング

表 5. JMS バインディング用の事前定義関数セレクター

関数セレクター	説明
単純な JMS データ・バインディング用 JMS 関数セレクター	メッセージの JMSType プロパティーを使用して操作名を選択します。
JMS ヘッダー・プロパティー関数セレクター	ヘッダーの JMS ストリング・プロパティー、TargetFunctionName の値を返します。
JMS サービス・ゲートウェイ関数セレクター	クライアントによって設定された JMSReplyTo プロパティーを調べ、要求が片方向操作と両方向操作のどちらであるかを判別します。

WebSphere MQ バインディング

次の表は、WebSphere MQ バインディングで使用できる関数セレクターの一覧です。

表 6. WebSphere MQ バインディング用の事前定義関数セレクター

関数セレクター	説明
MQ handleMessage 関数セレクター	handleMessage を値として返します。この値は、エクスポート・メソッド・バインディングを使用してインターフェースの操作名にマップされています。
MQ が JMS デフォルト関数セレクターを使用	MQRFH2 ヘッダーのフォルダーの TargetFunctionName プロパティーからネイティブ操作を読み取ります。
MQ がメッセージ本体のフォーマットをネイティブ関数として使用	最後のヘッダーの Format フィールドを検索し、そのフィールドをストリングとして返します。
MQ タイプ関数セレクター	MQRFH2 ヘッダーで検出した Msd、Set、Type、および Format プロパティーを含む URL を取得して、エクスポート・バインディング内にメソッドを作成します。
MQ サービス・ゲートウェイ関数セレクター	MQMD ヘッダー内の MsgType プロパティーを使用して操作名を判断します。

HTTP バインディング

次の表は、HTTP バインディングで使用できる関数セレクターの一覧です。

表 7. HTTP バインディング用の事前定義関数セレクター

関数セレクター	説明
TargetFunctionName ヘッダーに基づく HTTP 関数セレクター	クライアントからの TargetFunctionName HTTP ヘッダー・プロパティーを使用して、実行時にエクスポートから呼び出す操作を判断します。

表 7. HTTP バインディング用の事前定義関数セレクター (続き)

関数セレクター	説明
URL および HTTP メソッドに基づく HTTP 関数セレクター	クライアントからの HTTP メソッドに付加された URL の相対パスを使用して、エクスポートに定義されたネイティブ操作を判断します。
操作名が設定された URL に基づく HTTP サービス・ゲートウェイ関数セレクター	要求 URL に「operationMode = oneWay」が付加されている場合、その URL に基づいて呼び出すメソッドを判断します。

注: WebSphere Integration Developer を使用して固有の関数セレクターを作成することもできます。関数セレクターの作成に関する詳細情報については、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。例えば、WebSphere MQ バインディング用の関数セレクターの作成に関する説明は、『MQ 関数セレクターの概要』に記載されています。

障害の処理

ご使用のインポート・バインディングおよびエクスポート・バインディングについて、障害データ・ハンドラーを指定することによって、処理中に発生する障害 (ビジネス例外など) を処理するように構成できます。障害データ・ハンドラーは、3 つのレベルでセットアップできます。具体的には、障害データ・ハンドラーを障害に関連付けるか、操作に関連付けるか、またはバインディングを使用するすべての操作を対象に関連付けます。

障害データ・ハンドラーは障害データを処理し、エクスポート・バインディングまたはインポート・バインディングによって送信される正しいフォーマットに変換します。

- エクスポート・バインディングでは、障害データ・ハンドラーはコンポーネントから送信された例外ビジネス・オブジェクトを、クライアント・アプリケーションによって使用できる応答メッセージに変換します。
- インポート・バインディングでは、障害データ・ハンドラーはサービスから送信された障害データまたは応答メッセージを、SCA コンポーネントによって使用できる例外ビジネス・オブジェクトに変換します。

インポート・バインディングの場合、バインディングは障害セレクターを呼び出します。障害セレクターは、応答メッセージが通常応答、ビジネス障害、またはランタイム例外のいずれであるかを判別します。

障害データ・ハンドラーは、特定の障害、操作、およびバインディングを使用するすべての操作について指定することができます。

- 障害データ・ハンドラーが 3 つすべてのレベルで設定されている場合、特定の障害に関連付けられたデータ・ハンドラーが呼び出されます。
- 障害データ・ハンドラーが操作およびバインディングのレベルで設定されている場合、操作に関連付けられたデータ・ハンドラーが呼び出されます。

WebSphere Integration Developer での障害処理の指定には、2 つのエディターが使用されます。インターフェース・エディターは、操作に障害があるかどうかを示すの

に使用されます。このインターフェースでバインディングが生成された後、プロパティー・ビューのエディターによって、障害をどのように処理するかを構成できます。詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『障害セレクター』を参照してください。

エクスポート・バインディングでの障害の処理方法

クライアント・アプリケーションからの要求の処理中に障害が発生したとき、エクスポート・バインディングは障害情報をクライアントに返すことができます。障害を処理してクライアントに返す方法を指定するようにエクスポート・バインディングを構成します。

エクスポート・バインディングの構成は WebSphere Integration Developer を使用して行います。

要求を処理するとき、クライアントは要求でエクスポートを呼び出し、エクスポートは SCA コンポーネントを呼び出します。要求の処理中に、SCA コンポーネントはビジネス応答を返すか、またはサービス・ビジネス例外あるいはサービス・ランタイム例外をスローすることができます。これが発生すると、エクスポート・バインディングは例外を障害メッセージに変換して、クライアントに送信します。これについて以下の図で示し、その次のセクションで説明します。

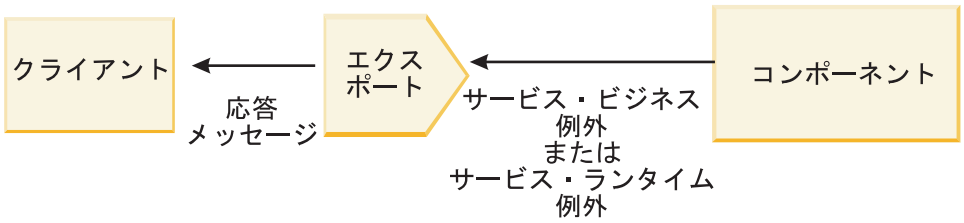


図 9. 障害情報をエクスポート・バインディングを経由してコンポーネントからクライアントに送信する方法

障害を処理するカスタムのデータ・ハンドラーまたはデータ・バインディングを作成することができます。

ビジネス障害

ビジネス障害とは、処理中に発生するビジネス・エラーまたはビジネス例外です。

createCustomer 操作を持つ以下のインターフェースについて考えてみます。この操作には、 CustomerAlreadyExists および MissingCustomerId の 2 つのビジネス障害が定義されています。

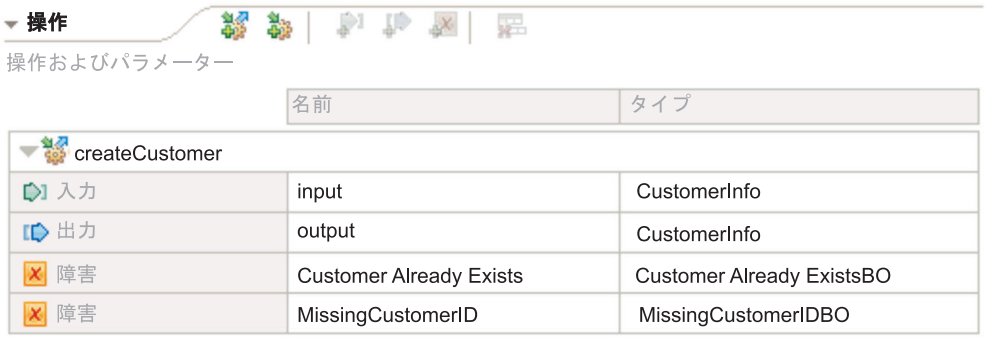


図 10. 2 つの障害を持つインターフェース

この例で、クライアントが顧客を作成する要求を (この SCA コンポーネントに) 送信したときにその顧客が既に存在する場合、コンポーネントは CustomerAlreadyExists 障害をエクスポートにスローします。エクスポートはこのビジネス障害を、呼び出し側のクライアントに伝搬して返す必要があります。エクスポートはこれを行うために、エクスポート・バインディングに設定された障害データ・ハンドラーを使用します。

ビジネス障害がエクスポート・バインディングによって受信されると、以下の処理が行われます。

1. バインディングは、どの障害を障害データ・ハンドラーを呼び出して処理させるかを決定します。サービス・ビジネス例外に障害名が含まれている場合、その障害について設定されているデータ・ハンドラーが呼び出されます。サービス・ビジネス例外が障害名を含まない場合、障害名は障害タイプをマッチングすることによって導き出されます。
2. バインディングは、サービス・ビジネス例外からのデータ・オブジェクトを使用して、障害データ・ハンドラーを呼び出します。
3. 障害データ・ハンドラーは障害データ・オブジェクトを応答メッセージに変換し、これをエクスポート・バインディングに返します。
4. エクスポートは応答メッセージをクライアントに返します。

サービス・ビジネス例外に障害名が含まれている場合、その障害について設定されているデータ・ハンドラーが呼び出されます。サービス・ビジネス例外が障害名を含まない場合、障害名は障害タイプをマッチングすることによって導き出されます。

ランタイム例外

ランタイム例外とは、要求の処理中に SCA アプリケーション内で発生する、ビジネス障害に対応しない例外のことです。ビジネス障害とは異なり、ランタイム例外はインターフェースで定義されません。

シナリオによっては、これらのランタイム例外をクライアント・アプリケーションに伝搬して、クライアント・アプリケーションが適切なアクションを実行できるようにしたい場合もあります。

例えば、クライアントが顧客を作成する要求を (SCA コンポーネントに) 送信したとき、要求の処理中に権限エラーが発生した場合、コンポーネントはランタイム例

外をスローします。このランタイム例外は呼び出し側のクライアントに戻すよう伝搬させて、クライアントが権限に関して適切なアクションを実行できるようにする必要があります。これはランタイム例外データ・ハンドラーをエクスポート・バインディングに構成することによって実現できます。

注: ランタイム例外データ・ハンドラーは、HTTP バインディングでのみ構成できます。

ランタイム例外の処理は、ビジネス障害の処理と似ています。ランタイム例外データ・ハンドラーが設定されている場合、以下の処理が実行されます。

1. エクスポート・バインディングはサービス・ランタイム例外について適切なデータ・ハンドラーを呼び出します。
2. データ・ハンドラーは障害データ・オブジェクトを応答メッセージに変換し、これをエクスポート・バインディングに返します。
3. エクスポートは応答メッセージをクライアントに返します。

障害処理およびランタイム例外処理はオプションです。障害またはランタイム例外を呼び出し側のクライアントに伝搬させたくない場合、障害データ・ハンドラーまたはランタイム例外データ・ハンドラーを構成しないでください。

インポート・バインディングでの障害の処理方法

コンポーネントはインポートを使用して、要求をモジュール外部のサービスに送信します。 要求の処理中に障害が発生すると、サービスはインポート・バインディングに障害を返します。障害を処理してコンポーネントに返す方法は、インポート・バインディングを構成して指定することができます。

インポート・バインディングの構成は WebSphere Integration Developer を使用して行います。障害データ・ハンドラー (または障害データ・バインディング) を指定できますが、障害セレクターを指定することもできます。

障害データ・ハンドラー

要求を処理するサービスがインポート・バインディングに障害情報を送信するときには、例外の形式または障害データが含まれる応答メッセージが使用されます。

インポート・バインディングはサービス例外または応答メッセージをサービス・ビジネス例外またはサービス・ランタイム例外に変換します。これについて以下の図で示し、その次のセクションで説明します。

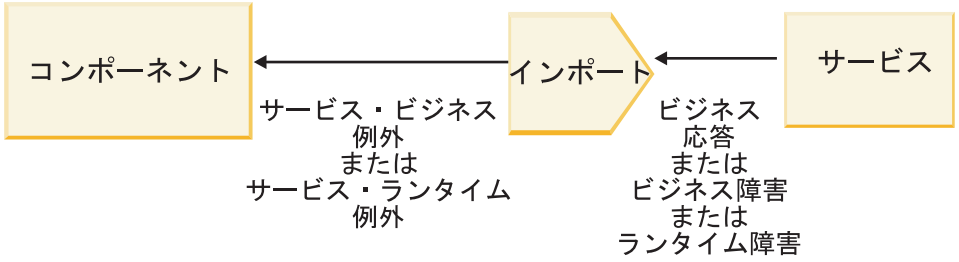


図 11. 障害情報をインポートを経由してサービスからコンポーネントに送信する方法

障害を処理するカスタムのデータ・ハンドラーまたはデータ・バインディングを作成することができます。

障害セレクター

インポート・バインディングを構成するとき、障害セレクターを指定することができます。障害セレクターは、インポートの応答が実際の応答か、ビジネス例外か、またはランタイム障害であるかを判別します。またこれは、応答の本体またはヘッダーからネイティブの障害名を判別します。ネイティブの障害名は、バインディング構成によって、関連するインターフェースの障害名にマップされます。

JMS、MQ JMS、汎用 JMS、WebSphere MQ、および HTTP インポートでは、プリパッケージされた 2 つのタイプの障害セレクターを使用できます。

表 8. プリパッケージされている障害セレクター

障害セレクター・タイプ	説明
ヘッダー・ベース	着信応答メッセージのヘッダーに基づいて、応答メッセージがビジネス障害、ランタイム例外、または通常のメッセージのいずれであるかを判別します。
SOAP	応答 SOAP メッセージが通常応答、ビジネス障害、またはランタイム例外のいずれであるかを判別します。

以下は、ヘッダー・ベースの障害セレクターおよび SOAP 障害セレクターの例です。

- ヘッダー・ベースの障害セレクター

  アプリケーションで、着信メッセージがビジネス障害であることを示す場合、着信メッセージにはビジネス障害に対応する以下の 2 つのヘッダーが含まれます。

  ```
  Header name = FaultType, Header value = Business
  Header name = FaultName, Header value = <user defined native fault name>
  ```

  アプリケーションで、着信応答メッセージがランタイム例外であることを示す場合、着信メッセージには以下に示す 1 つのヘッダーが含まれます。

  ```
  Header name = FaultType, Header value = Runtime
  ```

- SOAP 障害セレクター

  ビジネス障害は、SOAP メッセージの一部として送信できます。それには、SOAP メッセージに以下のカスタム SOAP ヘッダーを設定します。この場合の障害名は、「CustomerAlreadyExists」です。

  ```
  <ibmSoap:BusinessFaultName
  xmlns:ibmSoap="http://www.ibm.com/soap">CustomerAlreadyExists
  <ibmSoap:BusinessFaultName>
  ```

障害セレクターはオプションです。障害セレクターを指定しないと、インポート・バインディングは応答のタイプを判別できません。そのため、バインディングは応答をビジネス応答として扱い、応答データ・ハンドラーまたはデータ・バインディングを呼び出します。

カスタム障害セレクターを作成することができます。カスタム障害セレクターを作成するステップは、WebSphere Integration Developer インフォメーション・センターのトピック『カスタム障害セレクターの開発』にあります。

ビジネス障害

要求の処理にエラーがあると、ビジネス障害が発生することがあります。例えば、顧客を作成する要求を送信したときにその顧客が既に存在する場合、サービスはインポート・バインディングにビジネス例外を送信します。

ビジネス例外がバインディングによって受信された場合、処理ステップは障害セレクターがバインディングに設定されているかどうかに依存します。

- 障害セレクターが設定されていない場合、バインディングは応答データ・ハンドラーまたは応答データ・バインディングを呼び出します。
- 障害セレクターが設定されている場合、以下の処理が実行されます。
  1. インポート・バインディングは障害セレクターを呼び出して、応答がビジネス障害、ビジネス応答、またはランタイム障害のいずれであるかを判別します。
  2. 応答がビジネス障害である場合、インポート・バインディングは、ネイティブ障害名を提供するために障害セレクターを呼び出します。
  3. インポート・バインディングは、障害セレクターによって返されたネイティブ障害名に対応する WSDL 障害を判別します。
  4. インポート・バインディングは、この WSDL 障害用に構成された障害データ・ハンドラーを判別します。
  5. インポート・バインディングは障害データについて、この障害データ・ハンドラーを呼び出します。
  6. 障害データ・ハンドラーは障害データをデータ・オブジェクトに変換し、これをインポート・バインディングに返します。
  7. インポート・バインディングはデータ・オブジェクトおよび障害名を使用して、サービス・ビジネス例外オブジェクトを構成します。
  8. インポートはサービス・ビジネス例外オブジェクトをコンポーネントに返します。

ランタイム例外

ランタイム例外は、サービスとの通信に問題があるときに発生することがあります。ランタイム例外の処理は、ビジネス例外の処理と似ています。障害セレクターが設定されている場合、以下の処理が実行されます。

1. インポート・バインディングは例外データについて適切なランタイム例外データ・ハンドラーを呼び出します。
2. ランタイム例外データ・ハンドラーは例外データをサービス・ランタイム例外オブジェクトに変換し、これをインポート・バインディングに返します。
3. インポートはサービス・ランタイム例外オブジェクトをコンポーネントに返します。

SCA モジュールとオープン SCA サービスの間のインターオペラビリティー

IBM WebSphere Application Server V7.0 Feature Pack for Service Component Architecture (SCA) は、オープン SCA 仕様に基づいてアプリケーションを構築するための、単純であるが強力なプログラミング・モデルを提供します。WebSphere Process Serverの SCA モジュールは、インポートおよびエクスポート・バインディングを使用して、Rational® Application Developer 環境で開発され、WebSphere Application Server Feature Pack for Service Component Architecture によりホストされるオープン SCA サービスと相互運用します。

SCA アプリケーションは、インポート・バインディングを使ってオープン SCA アプリケーションを呼び出します。SCA アプリケーションは、エクスポート・バインディングを使ってオープン SCA アプリケーションからの呼び出しを受け取ります。サポートされるバインディングのリストを、32 ページの『相互運用可能なバインディングを利用したサービスの呼び出し』に示します。

SCA モジュールからのオープン SCA サービスの呼び出し

WebSphere Integration Developer を使って開発した SCA アプリケーションは、Rational Application Developer 環境で開発されたオープン SCA アプリケーションを呼び出すことができます。このセクションでは、SCA インポート・バインディングを使用して SCA モジュールからオープン SCA サービスを呼び出す例を示します。

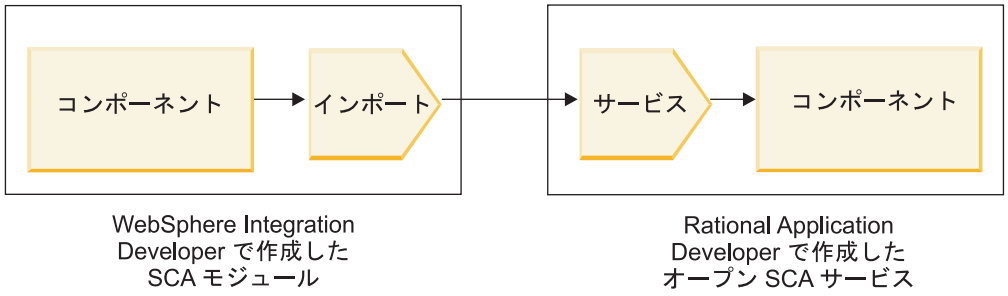


図 12. オープン SCA サービスを呼び出す SCA モジュールのコンポーネント

オープン SCA サービスの呼び出しには特別の構成は必要ありません。

SCA インポート・バインディングを使ってオープン SCA サービスに接続するには、オープン SCA サービスのコンポーネント名とサービス名をインポート・バインディングに指定します。

1. オープン SCA コンポジットからターゲットのコンポーネントおよびサービスの名前を取得するには、以下の手順を実行します。
   a. 「**ウィンドウ**」 → 「**ビューの表示 (Show View)**」 → 「**プロパティー**」をクリックして、「**プロパティー**」タブを開きます。
   b. コンポーネントとサービスが含まれるコンポジット・ダイアグラムをダブルクリックして、コンポジット・エディターを開きます。例えば、**customer** という名前のコンポーネントの場合、コンポジット・ダイアグラムは **customer.composite_diagram** となります。

c. ターゲット・コンポーネントをクリックします。

d. 「**プロパティー**」タブの「**名前**」フィールドに表示されるターゲット・コンポーネントの名前をメモします。

e. このコンポーネントに関連付けられたサービス・アイコンをクリックします。

f. 「**プロパティー**」タブの「**名前**」フィールドに表示されるサービスの名前をメモします。

2. オープン SCA サービスに接続するように WebSphere Process Server インポートを構成するには、以下の手順を実行します。

a. WebSphere Integration Developer で、オープン SCA サービスに接続する SCA インポートの「**プロパティー**」タブにナビゲートします。

b. 「**モジュール名**」フィールドに、ステップ 1d でメモしたコンポーネント名を入力します。

c. 「**エクスポート名**」フィールドに、ステップ 1f でメモしたサービス名を入力します。

d. Ctrl+S キーを押して、作業内容を保存します。

オープン SCA サービスからの SCA モジュールの呼び出し

Rational Application Developer 環境で開発されたオープン SCA アプリケーションは、WebSphere Integration Developer を使って開発された SCA アプリケーションを呼び出すことができます。このセクションでは、オープン SCA サービスから (SCA エクスポート・バインディングを使用して) SCA モジュールを呼び出す例を示します。

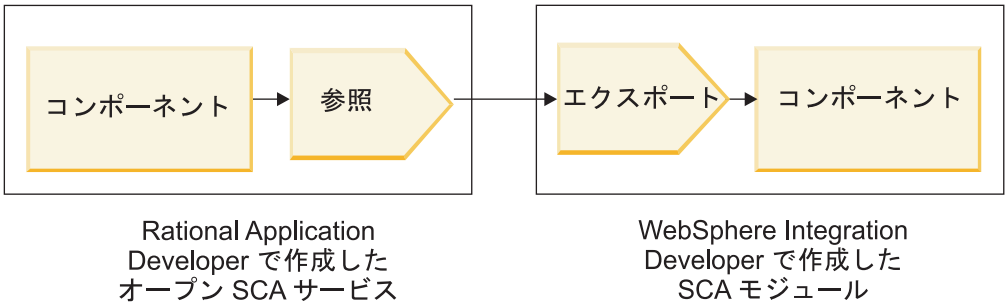


図 13. SCA モジュールのコンポーネントを呼び出すオープン SCA サービス

オープン SCA 参照バインディングにより SCA コンポーネントに接続するには、モジュール名およびエクスポート名を指定します。

1. ターゲットのモジュールおよびエクスポートの名前を取得するには、以下の手順を実行します。

a. WebSphere Integration Developer でモジュールをダブルクリックして、アセンブリー・エディターでそのモジュールを開きます。

b. エクスポートをクリックします。

c. 「**プロパティー**」タブの「**名前**」フィールドに表示されるエクスポートの名前をメモします。

2. WebSphere Process Server モジュールおよびエクスポートに接続するオープン SCA 参照を構成します。
   a. Rational Application Developer で、コンポーネントとサービスが含まれるコンポジット・ダイアグラムをダブルクリックして、コンポジット・エディターを開きます。
   b. コンポーネント参照の参照アイコンをクリックして、「**プロパティー**」タブに参照プロパティーを表示します。
   c. ページの左側にある「**バインディング**」タブをクリックします。
   d. 「**バインディング**」をクリックし、「**追加**」をクリックします。
   e. 「**SCA**」バインディングを選択します。
   f. 「**URI**」フィールドに WebSphere Process Server モジュール名を入力し、それに続いてスラッシュ (『/』) とエクスポート名 (ステップ 1c (31 ページ) で確認した名前) を入力します。
   g. 「**OK**」をクリックします。
   h. Ctrl+S キーを押して、作業内容を保存します。

相互運用可能なバインディングを利用したサービスの呼び出し

オープン SCA サービスとの相互運用のために次のバインディングがサポートされています。

- SCA バインディング

  SCA インポート・バインディングを使って WebSphere Process Server SCA モジュールがオープン SCA サービスを呼び出すときには、次のスタイルの呼び出しがサポートされています。

  – 非同期 (片方向)
  – 同期 (要求/応答)

  SCA インポート・インターフェースおよびオープン SCA サービス・インターフェースは、Web Services Interoperability (WS-I) に準拠した WSDL インターフェースを使用する必要があります。

  SCA バインディングは、トランザクションおよびセキュリティー・コンテキストの伝搬をサポートしている点に留意してください。

- SOAP1.1/HTTP または SOAP1.2/HTTP プロトコルを用いた Web サービス (JAX-WS) バインディング

  SCA インポート・インターフェースおよびオープン SCA サービス・インターフェースは、Web Services Interoperability (WS-I) に準拠した WSDL インターフェースを使用する必要があります。

  また、次のサービス品質がサポートされています。

  – Web Services Atomic Transaction
  – Web サービスのセキュリティー

- EJB バインディング

EJB バインディングが使用されているときは、Java インターフェースを使って SCA モジュールとオープン SCA サービス間の相互作用を定義します。

EJB バインディングは、トランザクションおよびセキュリティー・コンテキストの伝搬をサポートしている点に留意してください。

- JMS バインディング

  SCA インポート・インターフェースおよびオープン SCA サービス・インターフェースは、Web Services Interoperability (WS-I) に準拠した WSDL インターフェースを使用する必要があります。

  次の JMS プロバイダーがサポートされています。

  – WebSphere Platform Messaging (JMS バインディング)
  – WebSphere MQ (MQ JMS バインディング)

注: ビジネス・グラフは SCA バインディングの中で相互運用できません。このため、WebSphere Application Server Feature Pack for Service Component Architecture との相互運用で使用されているインターフェースではサポートされません。

バインディング・タイプ

インポートおよびエクスポートと共にプロトコル固有の「バインディング」を使用して、データをモジュールの内部または外部に移送する方法を指定します。

適切なバインディングの選択

アプリケーションのニーズに合わせて使用できるさまざまなバインディングがあります。

WebSphere Integration Developer で使用できるバインディングには、さまざまな選択肢があります。このリストは、アプリケーションのニーズにより適しているバインディングのタイプを識別する助けになります。

以下の点が当てはまる場合は、*SCA* バインディングを考慮してください。

- すべてのサービスが WebSphere Integration Developer モジュールに組み込まれている (外部サービスがない)。
- 互いに直接対話する複数の SCA モジュールに機能を分けたい。
- モジュールが密結合になっている

次の点が当てはまる場合は、*Web* サービス・バインディングを考慮してください。

- インターネットを介して外部サービスにアクセスする必要があるか、インターネットを介してサービスを提供する必要がある。
- サービスが疎結合になっている。
- 同期通信が望ましい (あるサービスからの要求が別のサービスからの応答を待機できる)
- アクセスする外部サービスまたは提供するサービスのプロトコルが SOAP/HTTP または SOAP/JMS である。

以下の点が当てはまる場合は、*HTTP* バインディングを考慮してください。

- インターネットを介して外部サービスにアクセスする必要があるか、インターネットを介してサービスを提供する必要があり、HTTP モデルに基づく他の Web サービスを処理している (GET、PUT、DELETE などのよく知られた HTTP インターフェース操作を使用している)。
- サービスが疎結合になっている。
- 同期通信が望ましい (あるサービスからの要求が別のサービスからの応答を待機できる)

以下の点が当てはまる場合は、*EJB* バインディングを考慮してください。

- バインディングの対象が、それ自体が EJB であるインポート済みサービスか、EJB クライアントによるアクセスが必要なインポート済みサービスである。
- インポートされたサービスが疎結合になっている。
- ステートフル EJB による対話は不要である。
- 同期通信が望ましい (あるサービスからの要求が別のサービスからの応答を待機できる)

以下の点が当てはまる場合は、*EIS* バインディングを考慮してください。

- リソース・アダプターを使用して EIS システム上のサービスにアクセスする必要がある。
- 非同期データ伝送よりも同期データ伝送が望ましい。

次の点が当てはまる場合は、*JMS* バインディングを考慮してください。

注: JMS バインディングには、いくつかのタイプがあります。JMS を使用して SOAP メッセージを交換する必要がある場合は、SOAP/JMS プロトコルを使用した Web サービス・バインディングを検討してください。 35 ページの『Web サービス・バインディング』を参照してください。

- メッセージング・システムにアクセスする必要がある。
- サービスが疎結合になっている。
- 同期データ伝送よりも非同期データ伝送が望ましい。

次の点が当てはまる場合は、*汎用 JMS* バインディングを考慮してください。

- IBM 以外のベンダーのメッセージング・システムにアクセスする必要がある。
- サービスが疎結合になっている。
- パフォーマンスよりも信頼性が重要な要件になっている (同期データ伝送よりも非同期データ伝送が望ましい)

以下の点が当てはまる場合は、*MQ* バインディングを考慮してください。

- WebSphere MQ メッセージング・システムにアクセスする必要があり、MQ ネイティブ機能を使用する必要がある。
- サービスが疎結合になっている。
- パフォーマンスよりも信頼性が重要な要件になっている (同期データ伝送よりも非同期データ伝送が望ましい)

以下の点が当てはまる場合は、*MQ JMS* バインディングを考慮してください。

- WebSphere MQ メッセージング・システムにアクセスする必要があるが、JMS コンテキスト内でアクセス可能である (アプリケーションで、機能の JMS サブセットを利用すれば十分である)。
- サービスが疎結合になっている。
- パフォーマンスよりも信頼性が重要な要件になっている (同期データ伝送よりも非同期データ伝送が望ましい)

SCA バインディング

Service Component Architecture (SCA) バインディングを使用すると、サービスは他のモジュール内の他のサービスと通信できるようになります。SCA バインディングを持つインポートを使用すると、別の SCA モジュール内のサービスにアクセスできるようになります。SCA バインディングを持つエクスポートを使用すると、サービスを他のモジュールに提供することができます。

SCA モジュールでのインポートおよびエクスポートに対して SCA バインディングを生成および構成するには、WebSphere Integration Developer を使用します。

モジュールが同じサーバー上で実行されている場合、または同じクラスターにデプロイされている場合、最も簡単に使用できる最も速いバインディングは SCA バインディングです。

SCA バインディングが組み込まれているモジュールをサーバーにデプロイすると、管理コンソールを使用して、バインディングに関する情報を表示したり、バインディングの選択済みプロパティーを変更したり (インポート・バインディングの場合) することができます。

Web サービス・バインディング

Web サービス・バインディングとは、Service Component Architecture (SCA) コンポーネントと Web サービスとの間でメッセージを送信する手段です。

Web サービス・バインディングの概要

Web サービス・インポート・バインディングを使用すると、Service Component Architecture (SCA) コンポーネントから外部の Web サービスを呼び出すことができます。 Web サービス・エクスポート・バインディングを使用すると、SCA コンポーネントを Web サービスとしてクライアントに公開できます。

Web サービス・バインディングを使用することにより、相互運用可能な SOAP メッセージおよびサービス品質 (Qos) を使用して外部サービスにアクセスできます。

SCA モジュールでのインポートおよびエクスポートに関して Web サービス・バインディングを生成および構成するには、WebSphere Integration Developer を使用します。以下のタイプの Web サービス・バインディングが使用可能です。

- SOAP1.2/HTTP および SOAP1.1/HTTP

  これらのバインディングは、Java API for XML Web Services (JAX-WS) (Web サービスを作成するための Java プログラミング API) に基づいています。

  – Web サービスが SOAP 1.2 仕様に準拠している場合は、SOAP1.2/HTTP を使用します。

– Web サービスが SOAP 1.1 仕様に準拠している場合は、SOAP1.1/HTTP を使用します。

これらのいずれかのバインディングを選択すると、SOAP メッセージで添付ファイルを送信できます。

Web サービス・バインディングは、標準 SOAP メッセージと連動します。ただし、いずれかの Web サービス JAX-WS バインディングを使用すると、SOAP メッセージを解析または作成する方法をカスタマイズすることができます。例えば、SOAP メッセージで非標準エレメントを処理したり、SOAP メッセージに追加の処理を適用したりすることができます。バインディングの構成時に、このような処理を SOAP メッセージに対して実行するカスタム・データ・ハンドラーを指定します。

- SOAP1.1/HTTP

  Java API for XML-based RPC (JAX-RPC) に基づく SOAP エンコード・メッセージを使用する Web サービスを作成する場合は、このバインディングを使用します。

- SOAP1.1/JMS

  Java Message Service (JMS) 宛先を使用して SOAP メッセージを送信または受信するには、このバインディングを使用します。

SOAP メッセージの伝達に使うトランスポート (HTTP または JMS) に関係なく、Web サービス・バインディングは常に要求/応答の対話を同期的に処理します。サービス・プロバイダー上で呼び出しを実行するスレッドは、プロバイダーから応答を受け取るまでブロックされます。この呼び出しスタイルについて詳しくは、『同期呼び出し』を参照してください。

重要: 以下の Web サービス・バインディングの組み合わせは、同じモジュール内のエクスポートに対しては使用できません。これらのエクスポート・バインディングを複数使用してコンポーネントを公開する必要がある場合は、それぞれを別個のモジュールに分けてから、SCA バインディングを使用してそれらのモジュールをコンポーネントに接続する必要があります。

- JAX-RPC を使用する SOAP 1.1/JMS と、JAX-RPC を使用する SOAP 1.1/HTTP
- JAX-RPC を使用する SOAP 1.1/HTTP と、JAX-WS を使用する SOAP 1.1/HTTP
- JAX-RPC を使用する SOAP 1.1/HTTP と、JAX-WS を使用する SOAP 1.2/HTTP

Web サービス・バインディングが組み込まれている SCA モジュールをサーバーにデプロイすると、管理コンソールを使用してバインディングに関する情報を表示するか、バインディングの選択済みプロパティーを変更することができます。

注: Web サービスを使用すると、複数のアプリケーションが、標準のサービス記述を使用したり、交換するメッセージに標準形式を使用したりすることによって、相互運用できるようになります。例えば、Web 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・バインディングは、Web Services Enhancements (WSE) Version 3.5 および Windows® Communication Foundation (WCF) Version 3.5 for Microsoft® .NET を使用して実装されたサービスと相互運用できます。このようなサービスとの相互運用を行う際には、次のことを確認する必要があります。

- Web サービス・エクスポートへのアクセスに使用される Web サービス記述言語 (WSDL) ファイルに、インターフェースの各操作に対する空ではない SOAP アクション値が含まれていること。
- Web サービス・クライアントが、Web サービス・エクスポートへのメッセージ送信の際に、SOAPAction ヘッダーまたは wsa:Action ヘッダーを設定していること。

SOAP ヘッダーの伝搬

SOAP メッセージを処理するときは、受信したメッセージの特定の SOAP ヘッダー情報にアクセスすること、 SOAP ヘッダーを持つメッセージが特定の値とともに送信されていることを確認すること、または SOAP ヘッダーがモジュールを通過できるようにすることが必要になる場合があります。

WebSphere Integration Developer で Web サービス・バインディングを構成する場合、 SOAP ヘッダーを伝搬するかどうかを指定できます。

- エクスポートで要求を受信する場合、またはインポートで応答を受信する場合は、 SOAP ヘッダー情報にアクセスして、モジュール内のロジックをヘッダー値に基づくようにし、それらのヘッダーを変更できます。
- エクスポートから要求を送信する場合、またはインポートから応答を送信する場合は、 SOAP ヘッダーをこれらのメッセージに含めることができます。

伝搬される SOAP ヘッダーの形式、およびそれが存在するかどうかは、インポートまたはエクスポートで構成されたポリシー・セットの影響を受ける場合があります(39 ページの表 9 を参照)。

インポートまたはエクスポートの SOAP ヘッダーの伝搬を構成するには、(WebSphere Integration Developer の「プロパティー」ビューから) 「**プロトコル・ヘッダーの伝搬**」タブを選択し、必要なオプションを選択します。

WS-Addressing ヘッダー

WS-Addressing ヘッダーは、Web サービス (JAX-WS) バインディングで伝搬できます。

WS-Addressing ヘッダーを伝搬する場合は、以下の情報に注意してください。

- WS-Addressing ヘッダーの伝搬を有効にした場合、以下の状況では、ヘッダーはモジュール内に伝搬されます。
  - 要求がエクスポートで受信される場合
  - 応答がインポートで受信される場合
- WS-Addressing ヘッダーは、WebSphere Process Server からのアウトバウンド・メッセージ内には伝搬されません (つまり、要求がインポートから送信される場合、または応答がエクスポートから送信される場合、ヘッダーは伝搬されません)。

WS-Security ヘッダー

WS-Security ヘッダーは、Web サービス (JAX-WS) バインディング、および Web サービス (JAX-RPC) バインディングの両方で伝搬できます。

Web サービスの WS-Security 仕様には、メッセージ保全性、メッセージ機密性、および単一メッセージ認証を通じて保護品質を提供するための SOAP メッセージングの拡張が記述されています。これらのメカニズムを使用することにより、さまざまなセキュリティー・モデルと暗号化テクノロジーに対応できます。

WS-Security ヘッダーを伝搬する場合は、以下の情報に注意してください。

- WS-Security ヘッダーの伝搬を有効にした場合、以下の状況では、ヘッダーはモジュールを通過して伝搬されます。
  - 要求がエクスポートで受信される場合
  - 要求がインポートから送信される場合
  - 応答がインポートで受信される場合
- デフォルトでは、応答がエクスポートから送信される場合は、ヘッダーは伝搬*されません*。 ただし、JVM プロパティー `WSSECURITY.ECHO.ENABLED` に `true` を設定すると、応答がエクスポートから送信される場合にヘッダーは伝搬されます。この場合、要求パスの WS-Security ヘッダーが変更されないときは、要求から応答に WS-Security ヘッダーが自動的にエコーされることがあります。
- 要求に対してインポートから送信される SOAP メッセージ、または応答に対してエクスポートから送信される SOAP メッセージの厳格な形式は、最初に受信した SOAP メッセージとは正確に一致しない場合があります。このため、すべてのデジタル署名は無効になると想定する必要があります。送信メッセージでデジタル署名が必要な場合は、適切なセキュリティー・ポリシー・セットを使用してデジタル署名を設定し、受信メッセージのデジタル署名に関連する WS-Security ヘッダーをモジュール内で削除する必要があります。

WS-Security ヘッダーを伝搬するには、WS-Security スキーマをアプリケーション・モジュールに組み込む必要があります。スキーマを組み込む手順については、40 ページの『WS-Security スキーマのアプリケーション・モジュールへの組み込み』を参照してください。

ヘッダーの伝搬方法

ヘッダーの伝搬方法は、インポート・バインディングまたはエクスポート・バインディングのセキュリティー・ポリシー設定に依存します。39 ページの表 9 を参照してください。

表 9. セキュリティー・ヘッダーを渡す方法

	セキュリティー・ポリシーを使用しないエクスポート・バインディング	**セキュリティー・ポリシーを使用したエクスポート・バインディング**
セキュリティー・ポリシーを使用しないインポート・バインディング	セキュリティー・ヘッダーは、そのままモジュールを通過します。暗号化解除はされません。 ヘッダーは、受信時と同じ形式でアウトバウンドに送信されます。 デジタル署名は、無効になる場合があります。	セキュリティー・ヘッダーは、暗号化解除され、モジュールを通過します。このとき、署名の検証と認証が実行されます。 暗号化解除されたヘッダーは、アウトバウンドに送信されます。 デジタル署名は、無効になる場合があります。
セキュリティー・ポリシーを使用したインポート・バインディング	セキュリティー・ヘッダーは、そのままモジュールを通過します。暗号化解除はされません。 ヘッダーは、インポートに伝搬してはなりません。そうしないと、重複のためにエラーが発生します。	セキュリティー・ヘッダーは、暗号化解除され、モジュールを通過します。このとき、署名の検証と認証が実行されます。 ヘッダーは、インポートに伝搬してはなりません。そうしないと、重複のためにエラーが発生します。

適切なポリシー・セットをエクスポート・バインディングとインポート・バインディングで構成してください。なぜなら、これにより、サービス・プロバイダーの構成または QoS 要件に対する変更と、サービス要求元が分離されるからです。標準 SOAP ヘッダーをモジュールで可視にすると、モジュールの処理 (ロギングやトレースなど) に影響を与えることができます。モジュールを通過して受信メッセージから送信メッセージに SOAP ヘッダーを伝搬すると、モジュールを分離する利点が減少します。

標準ヘッダー (WS-Security ヘッダーなど) が通常生成されるポリシー・セットがインポートまたはエクスポートに関連付けられているときは、標準ヘッダーをインポート (要求時) またはエクスポート (応答時) に伝搬してはなりません。そうしないと、ヘッダーの重複によりエラーが発生します。代わりに、ヘッダーを明示的に削除するか、もしくはプロトコル・ヘッダーが伝搬されないようにインポート・バインディングまたはエクスポート・バインディングを構成する必要があります。

SOAP ヘッダーへのアクセス

SOAP ヘッダーを含むメッセージが Web サービスのインポートまたはエクスポートから受信されると、それらのヘッダーは、サービス・メッセージ・オブジェクト (SMO) のヘッダー・セクションに配置されます。ヘッダー情報にアクセスするには、『SMO の SOAP ヘッダー情報へのアクセス』を参照してください。

WS-Security スキーマのアプリケーション・モジュールへの組み込み

以下の手順では、スキーマをアプリケーション・モジュールに組み込むステップを説明します。

- WebSphere Integration Developer が実行中のコンピューターがインターネットにアクセスできる場合は、以下のステップを実行します。
  1. ビジネス・インテグレーション・パースペクティブで、プロジェクトの「**依存関係**」を選択します。
  2. 「**事前定義リソース**」を展開し、「**WS セキュリティー 1.0 スキーマ・ファイル**」または「**WS セキュリティー 1.1 スキーマ・ファイル**」を選択して、スキーマをモジュールにインポートします。
  3. プロジェクトをクリーンにし再ビルドします。
- WebSphere Integration Developer を実行中のコンピューターがインターネットにアクセスできない場合は、インターネットにアクセスできる 2 番目のコンピューターにスキーマをダウンロードします。その後、WebSphere Integration Developer を実行中のコンピューターにスキーマをコピーします。
  1. インターネットにアクセスできるコンピューターから、以下の手順でリモート・スキーマをダウンロードします。
     a. 「**ファイル**」 → 「**インポート**」 → 「**ビジネス・インテグレーション**」 → 「**WSDL および XSD**」をクリックします。
     b. 「**リモート WSDL (Remote WSDL)**」または「**XSD ファイル (XSD file)**」を選択します。
     c. 以下のスキーマをインポートします。

        http://www.w3.org/2003/05/soap-envelope/

        http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd

        http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd
  2. インターネットにアクセスできないコンピューターにスキーマをコピーします。
  3. インターネットにアクセスできないコンピューターから、以下の手順でスキーマをインポートします。
     a. 「**ファイル**」 → 「**インポート**」 → 「**ビジネス・インテグレーション**」 → 「**WSDL および XSD**」 をクリックします。
     b. 「**ローカル WSDL (Local WSDL)**」または「**XSD ファイル (XSD file)**」を選択します。
  4. oasis-wss-wssecurity_secext-1.1.xsd のスキーマの場所を以下の手順で変更します。
     a. *workplace_location*/*module_name*/StandardImportFilesGen/oasis-wss-wssecurity-secext-1.1.xsd でスキーマを開きます。
     b. 変更前:

        ```
        <xs:import namespace='http://www.w3.org/2003/05/soap-envelope'
        schemaLocation='http://www.w3.org/2003/05/soap-envelope/'/>
        ```

        変更後:

        ```
        <xs:import namespace='http://www.w3.org/2003/05/soap-envelope'
        schemaLocation='../w3/_2003/_05/soap_envelope.xsd'/>
        ```

c. 変更前:

```
<xs:import namespace='http://www.w3.org/2001/04/xmlenc#'
schemaLocation='http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd'/>
```

変更後:

```
<xs:import namespace='http://www.w3.org/2001/04/xmlenc#'
schemaLocation='../w3/tr/_2002/rec_xmlenc_core_20021210/xenc-schema.xsd'/>
```

5. oasis-200401-wss-wssecurity-secext-1.0.xsd のスキーマの場所を以下の手順で変更します。
   a. *workplace_location*/*module_name*/StandardImportFilesGen/oasis-200401-wss-wssecurity-secext-1.0.xsd でスキーマを開きます。
   b. 変更前:

```
<xsd:import namespace="http://www.w3.org/2000/09/xmldsig#"
schemaLocation="http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd"/>
```

変更後:

```
<xsd:import namespace="http://www.w3.org/2000/09/xmldsig#"
schemaLocation="../w3/tr/_2002/rec_xmldsig_core_20020212/xmldsig-core-schema.xsd"/>
```

6. プロジェクトをクリーンにし再ビルドします。

SOAP メッセージの添付ファイル

添付ファイルとしてバイナリー・データ (PDF ファイルや JPEG イメージ) が含まれている SOAP メッセージを送信および受信することができます。添付ファイルには、*参照されている* 添付ファイル (サービス・インターフェースでメッセージ・パーツとして明示的に表わされる) と、*参照されていない* 添付ファイル (任意の数とタイプの添付ファイルを組み込むことができる) があります。

参照されている添付ファイルは、以下のいずれかの方法で表わされます。

- メッセージ・スキーマの wsi:swaRef タイプのエレメントとして

  wsi:swaRef タイプを使用して定義される添付ファイルは、メッセージ・エレメントがどのように MIME パーツに関連しているかを定義する Web Services Interoperability Organization (WS-I) の「*Attachments Profile Version 1.0*」(http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html)に準拠しています。
- バイナリー・スキーマ・タイプを使用した最上位メッセージ・パーツとして

  最上位メッセージ・パーツとして表わされる添付ファイルは、「*SOAP Messages with Attachments*」(http://www.w3.org/TR/SOAP-attachments)仕様に準拠しています。

参照されていない添付ファイルは、メッセージ・スキーマ内で表わされることなく、SOAP メッセージに組み込まれます。

いずれの場合にも、WSDL SOAP バインディングには、使用する予定の添付ファイルの MIME バインディングを組み込み、添付ファイルの最大サイズは 20 MB を超えないようにしてください。

注: 添付ファイル付きの SOAP メッセージを送信または受信するには、Java API for XML Web Services (JAX-WS) に基づく、いずれかの Web サービス・バインディングを使用する必要があります。

参照されている添付ファイル: swaRef タイプのエレメント:

サービス・インターフェースで swaRef タイプのエレメントとして表わされている添付ファイルが組み込まれた SOAP メッセージを送信および受信することができます。

swaRef タイプのエレメントは、メッセージ・エレメントがどのように MIME パーツに関連しているかを定義する Web Services Interoperability Organization (WS-I) の「*Attachments Profile Version 1.0*」(http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html)で定義されています。

注: 生成またはコンシュームされる SOAP メッセージが WS-I Attachments Profile に準拠していることは保証されません。特に、WS-I Attachments Profile 1.0 のセクション 3.8 に記述されている「コンテンツ ID パーツのエンコード」はサポートされません。

SOAP メッセージ内の SOAP body には、添付ファイルのコンテンツ ID を識別する swaRef タイプのエレメントが含まれています。

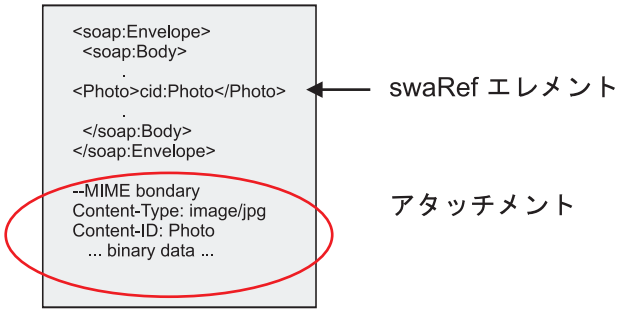


図 14. swaRef エレメントが含まれている SOAP メッセージ

この SOAP メッセージの WSDL では、添付ファイルを識別するメッセージ・パーツ内に swaRef タイプのエレメントが含まれています。

```
<element name="sendPhoto">
  <complexType>
     <sequence>
       <element name="Photo" type="wsi:swaRef"/>
     </sequence>
  </complexType>
 </element>
```

WSDL には、MIME multipart メッセージを使用することを示す MIME バインディングも含まれている必要があります。

注: MIME バインディングは最上位メッセージ・パーツにのみ適用されるため、WSDL には特定の swaRef タイプのメッセージ・エレメント用の MIME バインディングは含まれていません。

swaRef タイプのエレメントとして表わされている添付ファイルは、メディエーション・フロー・コンポーネントを介した場合にのみ伝搬できます。添付ファイルに別のコンポーネント・タイプでアクセスする必要がある場合や、別のコンポーネン

ト・タイプに伝搬する必要がある場合には、メディエーション・フロー・コンポーネントを使用して、そのコンポーネントがアクセスできる場所へ添付ファイルを移動してください。

添付ファイルのインバウンド処理

WebSphere Integration Developer を使用して、添付ファイルを受信するようエクスポート・バインディングを構成します。モジュールと、モジュールに関連付けられたインターフェースや操作 (タイプが swaRef のエレメント) を作成します。次に、Web サービス (JAX-WS) バインディングを作成します。

注: 詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『添付ファイルの操作 (Working with attachments)』を参照してください。

クライアントが swaRef 添付ファイル付きの SOAP メッセージを Service Component Architecture (SCA) コンポーネントに渡すと、Web サービス (JAX-WS) エクスポート・バインディングは、最初に添付ファイルを除去します。次に、メッセージの SOAP パーツを解析して、ビジネス・オブジェクトを作成します。最後に、バインディングは、ビジネス・オブジェクト内で添付ファイルのコンテンツ ID を設定します。

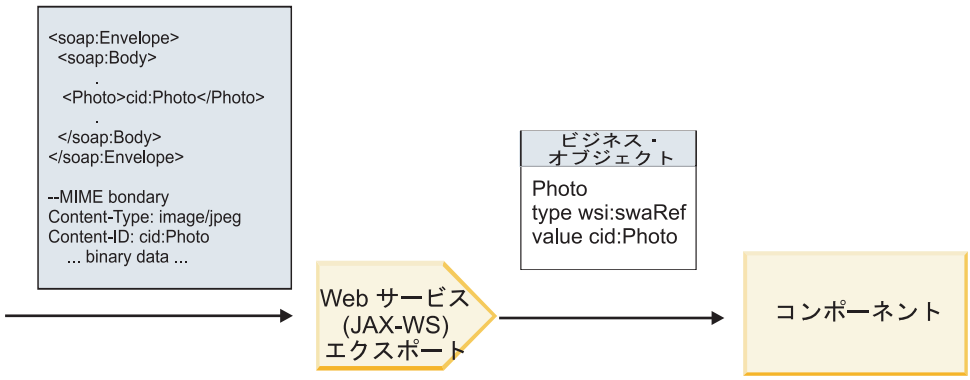


図 15. Web サービス (JAX-WS) エクスポート・バインディングが、swaRef 添付ファイル付きの SOAP メッセージを処理する方法

メディエーション・フロー・コンポーネント内の添付ファイル・メタデータへのアクセス

44 ページの図 16 に示されているとおり、swaRef 添付ファイルがコンポーネントによってアクセスされると、添付ファイル・コンテンツ ID はタイプが swaRef のエレメントとして表現されます。

SOAP メッセージのそれぞれの添付ファイルの SMO には、対応する `attachments` エレメントがあります。 WS-I swaRef タイプを使用する場合、`attachments` エレメントには、添付ファイルの実際のバイナリー・データだけでなく、添付ファイル・コンテンツ・タイプとコンテンツ ID も組み込まれます。

したがって、swaRef 添付ファイルの値を取得するには、swaRef タイプのエレメントの値を取得する必要があり、対応する `contentID` 値が含まれている `attachments` エレメントを見つける必要があります。通常、`contentID` 値では、`cid:` プレフィッ

クスが swaRef 値から除去されています。

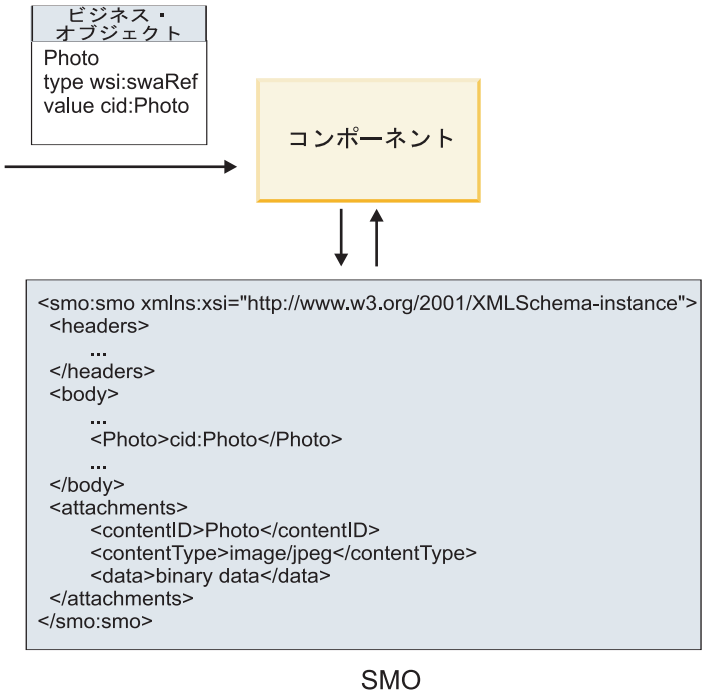


図 16. swaRef 添付ファイルが SMO 内で表わされる方法

アウトバウンド処理

外部 Web サービスを呼び出すように Web サービス (JAX-WS) インポート・バインディングを構成するには、WebSphere Integration Developer を使用します。インポート・バインディングは、WSDL 文書を使用して構成されます。この文書は、呼び出される Web サービスを記述し、Web サービスに渡される添付ファイルを定義しています。

SCA メッセージが Web サービス (JAX-WS) インポート・バインディングによって受信されると、インポートがメディエーション・フロー・コンポーネントにワイヤリングされている場合で、対応する `attachments` エレメントが swaRef タイプのエレメントにある場合は、 swaRef タイプのエレメントが添付ファイルとして送信されます。

アウトバウンド処理の場合、swaRef タイプのエレメントは常にコンテンツ ID 値と一緒に送信されます。ただし、メディエーション・モジュールは、`contentID` 値が一致する、対応する `attachments` エレメントが存在することを確認する必要があります。

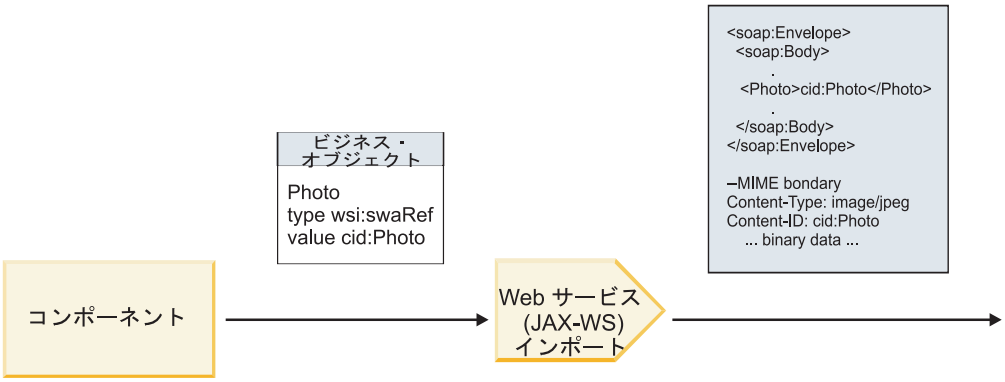


図 17. Web サービス (JAX-WS) インポート・バインディングが、swaRef 添付ファイル付きの SOAP メッセージを生成する方法

メディエーション・フロー・コンポーネント内の添付ファイル・メタデータの設定

SMO 内に、swaRef タイプのエレメント値と `attachments` エレメントがある場合、バインディングは SOAP メッセージ (添付ファイル付き) を作成し、受信側に送信します。

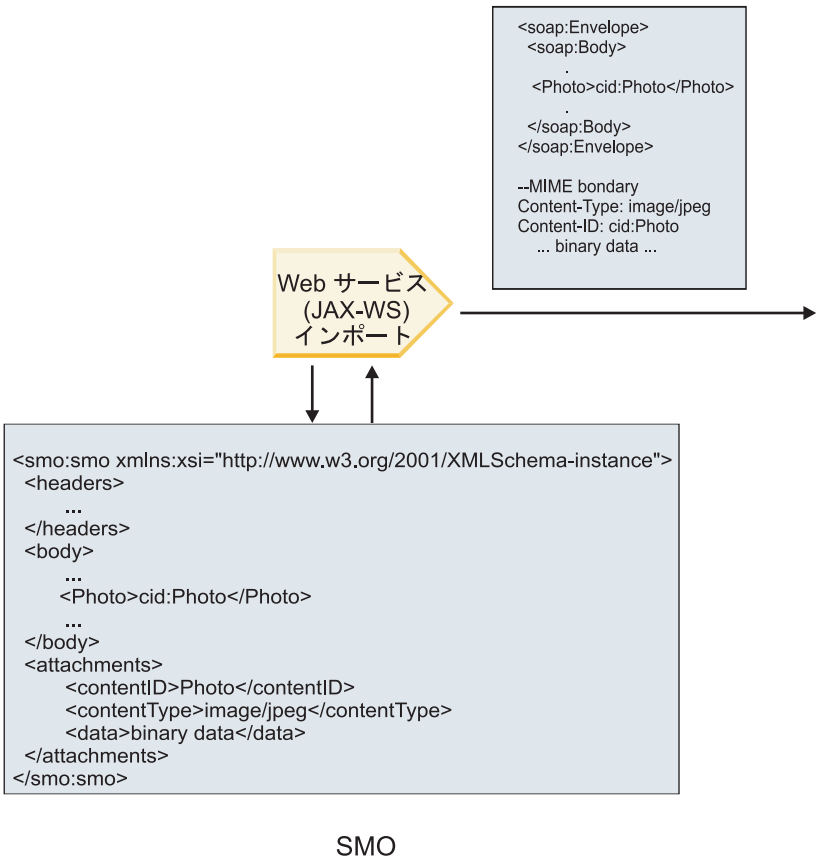


図 18. SOAP メッセージを作成するために SMO 内の swaRef 添付ファイルがアクセスされる方法

`attachments` エレメントは、メディエーション・フロー・コンポーネントがインポートまたはエクスポートに直接接続されている場合にのみ、SMO 内に存在します。

このエレメントは、他のコンポーネント・タイプによって渡されることはありません。他のコンポーネント・タイプが含まれているモジュールで値が必要になる場合は、メディエーション・フロー・コンポーネントを使用して、モジュール内でアクセス可能な場所に値をコピーする必要があり、別のメディエーション・フロー・コンポーネントを使用して、Web サービス・インポートによるアウトバウンド呼び出しの前に、正しい値を設定する必要があります。

重要: 『SMO の XML 表記』で説明しているように、XSL 変換メディエーション・プリミティブは、XSLT 1.0 変換を使用してメッセージを変換します。変換は、SMO の XML 直列化で作動します。XSL 変換メディエーション・プリミティブより、直列化のルートを指定することができます。また、XML 文書のルート・エレメントはこのルートを反映します。

SOAP メッセージを添付ファイルと共に送信する場合、選択したルート・エレメントによって、添付ファイルの伝搬方法が決まります。

- 「/body」を XML マップのルートとして使用した場合は、デフォルトですべての添付ファイルがマップ全体に伝搬します。
- 「/」をマップのルートとして使用した場合は、添付ファイルの伝搬を制御できます。

参照されている添付ファイル: 最上位メッセージ・パーツ:

サービス・インターフェースでパーツとして宣言されているバイナリー添付ファイルが組み込まれた SOAP メッセージを送信および受信することができます。

MIME multipart SOAP メッセージでは、SOAP 本体がメッセージの最初の部分であり、添付ファイルがその後に続きます。添付ファイルへの参照情報は、SOAP 本体に組み込まれています。

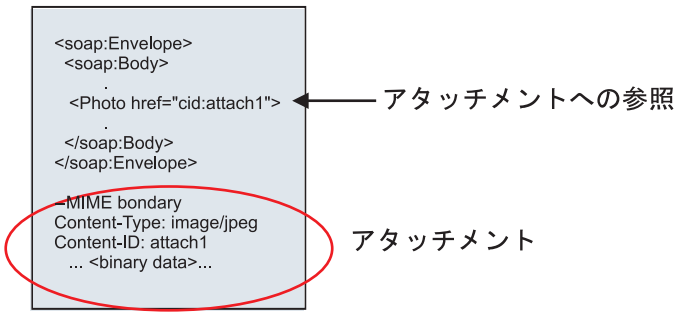


図 19. 参照されている添付ファイル付きの SOAP メッセージ

参照されている添付ファイルを SOAP メッセージで送信または受信することの利点は何でしょうか。 添付ファイルを構成するバイナリー・データ (多くの場合、大容量のデータ) は、SOAP メッセージ本体とは別に保持されているため、XML として解析する必要はありません。このため、バイナリー・データが XML エレメントに保持される場合よりも効率的に処理することができます。

参照されている添付ファイルのインバウンド処理

WebSphere Integration Developer を使用して、エクスポート・バインディングを構成します。モジュールと、モジュールに関連付けられたインターフェースや操作を作

成します。次に、Web サービス (JAX-WS) バインディングを作成します。「参照されている添付ファイル (Referenced attachments)」ページに、作成された操作のすべてのバイナリー・パーツが表示されるため、添付ファイルとして使用するパーツを選択します。

注: 参照されている添付ファイルとして送信または受信できるのは、バイナリー・タイプ (base64Binary または hexBinary) である最上位メッセージ・パーツ (入力メッセージまたは出力メッセージ内のパーツとして WSDL portType で定義されているエレメント) のみです。
詳しくは、WebSphere Integration Developer インフォメーション・センターのトピック『添付ファイルの操作 (Working with attachments)』を参照してください。

クライアントが添付ファイル付きの SOAP メッセージを Service Component Architecture (SCA) コンポーネントに渡すと、Web サービス (JAX-WS) エクスポート・バインディングは、最初に添付ファイルを除去します。次に、メッセージの SOAP パーツを解析して、ビジネス・オブジェクトを作成します。最後に、バインディングは、ビジネス・オブジェクト内で添付ファイル・バイナリーを設定します。

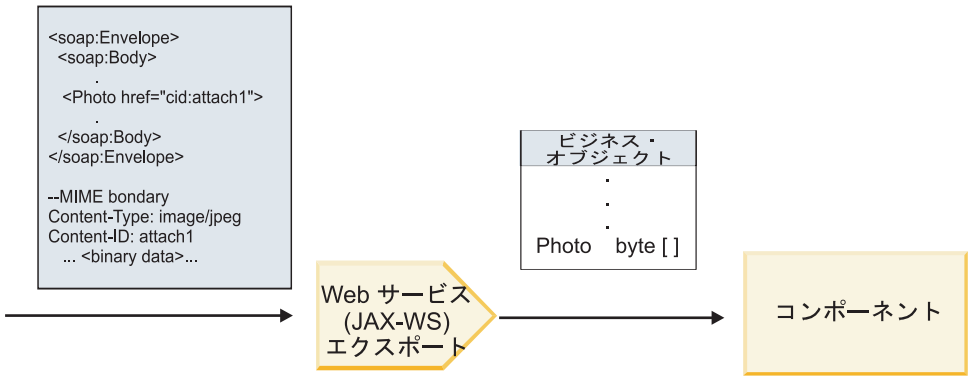


図 20. Web サービス (JAX-WS) エクスポート・バインディングが、参照されている添付ファイル付きの SOAP メッセージを処理する方法

メディエーション・フロー・コンポーネント内の添付ファイル・メタデータへのアクセス

図 20 に示されているとおり、参照されている添付ファイルがコンポーネントによってアクセスされると、添付ファイル・データはバイト配列として表現されます。

SOAP メッセージのそれぞれの参照されている添付ファイルの SMO には、対応する `attachments` エレメントがあります。 `attachments` エレメントには、添付ファイルのコンテンツ・タイプと、添付ファイルが保持されているメッセージの body エレメントへのパスが組み込まれています。

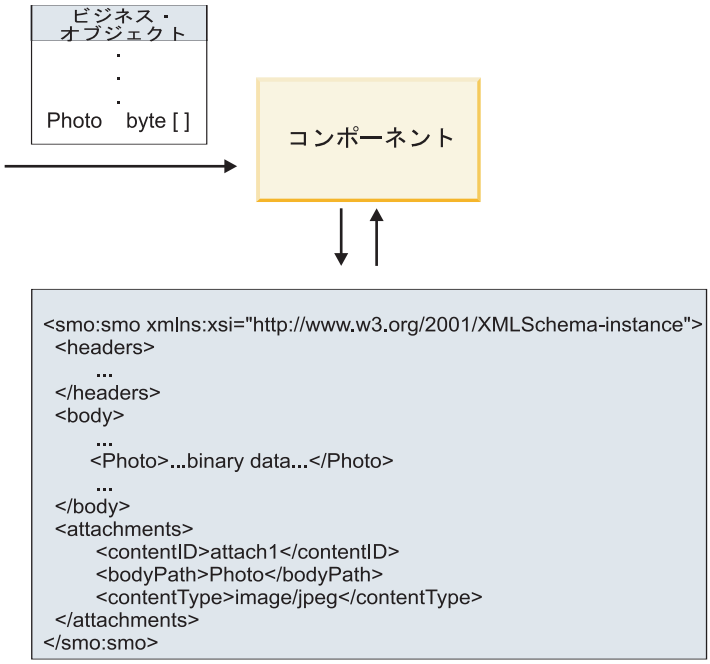


図 21. 参照されている添付ファイルが SMO 内で表わされる方法

重要: メッセージが変換され、添付ファイルが移動されると、メッセージの body エレメントへのパスは、自動的に更新されません。メディエーション・フローを使用することにより、(変換の一環として、または別のメッセージ・エレメント・セッターを使用するなどして) `attachments` エレメントを新しいパスで更新することができます

参照されている添付ファイルのアウトバウンド処理

外部 Web サービスを呼び出すように Web サービス (JAX-WS) インポート・バインディングを構成するには、WebSphere Integration Developer を使用します。インポート・バインディングは、WSDL 文書を使用して構成されます。この文書は、呼び出される Web サービスを記述し、添付ファイルとして渡すメッセージ・パーツを定義しています。

注: 添付ファイルを表すパーツ (WSDL 内で定義されている) は、単純タイプ (base64Binary または hexBinary) でなければなりません。パーツが complexType で定義されている場合、そのパーツは、添付ファイルとして扱われません。

インポート・バインディングでは、SMO の情報を使用して、バイナリーの最上位メッセージ・パーツを添付ファイルとして送信する方法が決定されます。

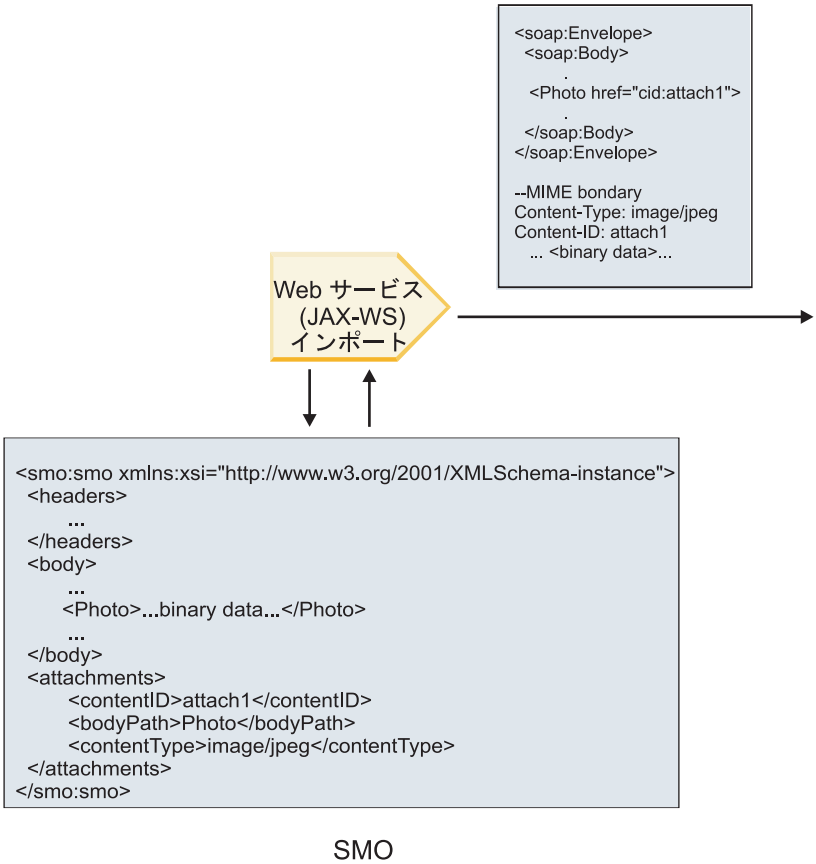


図 22. SOAP メッセージを作成するために SMO 内の参照されている添付ファイルがアクセスされる方法

attachments エレメントは、メディエーション・フロー・コンポーネントがインポートまたはエクスポートに直接接続されている場合にのみ、SMO 内に存在します。このエレメントは、他のコンポーネント・タイプによって渡されることはありません。他のコンポーネント・タイプが含まれているモジュールで値が必要になる場合は、メディエーション・フロー・コンポーネントを使用して、モジュール内でアクセス可能な場所に値をコピーする必要があり、別のメディエーション・フロー・コンポーネントを使用して、Web サービス・インポートによるアウトバウンド呼び出しの前に、正しい値を設定する必要があります。

バインディングでは、以下の条件の組み合わせを使用して、メッセージの送信方法 (または、メッセージを送信するかどうか) が決定されます。

- 最上位バイナリー・メッセージ・パーツの WSDL MIME バインディングが存在するかどうか。存在する場合は、コンテンツ・タイプの定義方法。
- bodyPath 値が最上位バイナリー・パーツを参照している attachments エレメントが SMO に存在するかどうか。

attachment エレメントが SMO に存在するときの添付ファイルの作成方法

bodyPath がメッセージ名パーツに一致する attachment エレメントが SMO に含まれている場合、添付ファイルを作成して送信する方法を以下の表に示します。

表 10. 添付ファイルの生成方法

最上位バイナリー・メッセージ・パーツの WSDL MIME バインディングの状況	**メッセージを作成して送信する方法**
以下のいずれかの場合に存在します。 • メッセージ・パーツのコンテンツ・タイプが定義されていない • 複数のコンテンツ・タイプが定義されている • ワイルドカード・コンテンツ・タイプが定義されている	メッセージ・パーツは、添付ファイルとして送信されます。 `Content-Id` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、Content-Id 値が生成されます。 `Content-Type` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、`application/octet-stream` が設定されます。
メッセージ・パーツのコンテンツが単一で非ワイルドカードの場合に存在します	メッセージ・パーツは、添付ファイルとして送信されます。 `Content-Id` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、Content-Id 値が生成されます。 `Content-Type` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、WSDL MIME コンテンツ・エレメントで定義されたタイプが設定されます。
存在しません	メッセージ・パーツは、添付ファイルとして送信されます。 `Content-Id` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、Content-Id 値が生成されます。 `Content-Type` には、添付ファイル・エレメントの値が設定されます (存在する場合)。それ以外の場合は、`application/octet-stream` が設定されます。 **注:** メッセージ・パーツを添付ファイルとして送信することが WSDL で定義されていないのに、メッセージ・パーツを添付ファイルとして送信すると、WS-I Attachments Profile 1.0 に準拠しなくなる可能性があります。そのため、このようなことは、できる限り避ける必要があります。

`attachment` エレメントが SMO に存在しないときの添付ファイルの作成方法

`bodyPath` がメッセージ名パーツに一致する `attachment` エレメントが SMO に含まれていない場合、添付ファイルを作成して送信する方法を以下の表に示します。

表 11. 添付ファイルの生成方法

最上位バイナリー・メッセージ・パーツの WSDL MIME バインディングの状況	**メッセージを作成して送信する方法**
以下のいずれかの場合に存在します。 • メッセージ・パーツのコンテンツ・タイプが定義されていない • 複数のコンテンツ・タイプが定義されている • ワイルドカード・コンテンツ・タイプが定義されている	メッセージ・パーツは、添付ファイルとして送信されます。 `Content-Id` が生成されます。 `Content-Type` には、`application/octet-stream` が設定されます。
メッセージ・パーツのコンテンツが単一で非ワイルドカードの場合に存在します	メッセージ・パーツは、添付ファイルとして送信されます。 `Content-Id` が生成されます。 `Content-Type` には、WSDL MIME コンテンツ・エレメントで定義されたタイプが設定されます。
存在しません	メッセージ・パーツは、添付ファイルとして送信されません。

重要: 『SMO の XML 表記』で説明しているように、XSL 変換メディエーション・プリミティブは、XSLT 1.0 変換を使用してメッセージを変換します。変換は、SMO の XML 直列化で作動します。XSL 変換メディエーション・プリミティブより、直列化のルートを指定することができます。また、XML 文書のルート・エレメントはこのルートを反映します。

SOAP メッセージを添付ファイルと共に送信する場合、選択したルート・エレメントによって、添付ファイルの伝搬方法が決まります。

- 「/body」を XML マップのルートとして使用した場合は、デフォルトですべての添付ファイルがマップ全体に伝搬します。
- 「/」をマップのルートとして使用した場合は、添付ファイルの伝搬を制御できます。

参照されていない添付ファイル:

サービス・インターフェースで宣言されていない、*参照されていない*添付ファイルを送信および受信できます。

MIME multipart SOAP メッセージでは、SOAP 本体がメッセージの最初の部分であり、添付ファイルがその後に続きます。 SOAP 本体には、添付ファイルへの参照情報は組み込まれていません。

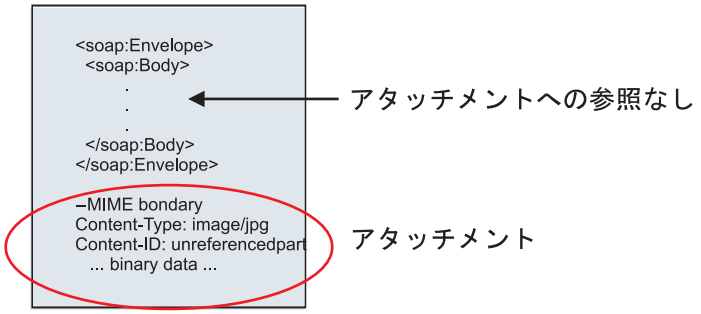


図 23. 参照されていない添付ファイル付きの SOAP メッセージ

参照されていない添付ファイル付きの SOAP メッセージは、Web サービス・エクスポートを介して Web サービス・インポートへ送信できます。ターゲットの Web サービスへ送信される出力メッセージには、添付ファイルが含まれます。

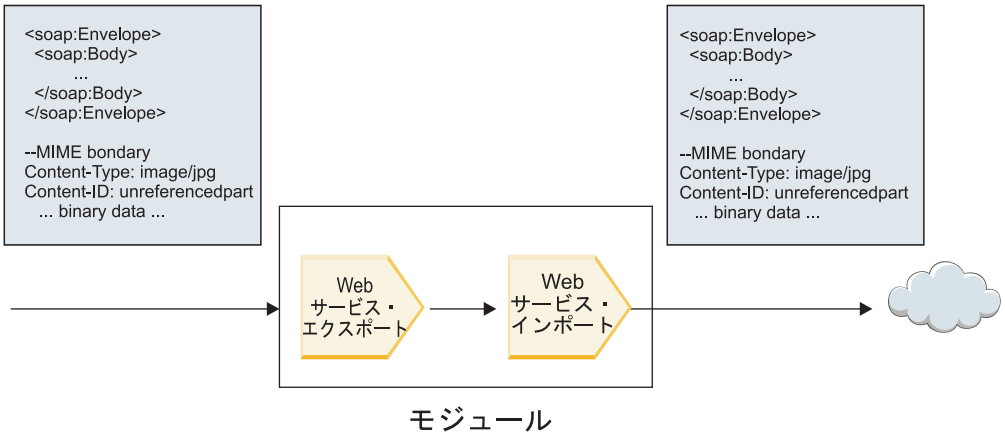


図 24. SCA モジュールを通過する添付ファイル

図 24 は、添付ファイル付きの SOAP メッセージが変更なしで通過する様子を示しています。

SOAP メッセージは、メディエーション・フロー・コンポーネントを使用して変更することもできます。例えば、メディエーション・フロー・コンポーネントを使用して SOAP メッセージからデータ (この場合は、メッセージの本体内部のバイナリー・データ) を抽出し、添付ファイル付きの SOAP メッセージを作成できます。データは、サービス・メッセージ・オブジェクト (SMO) の添付ファイル・エレメントの一部として処理されます。

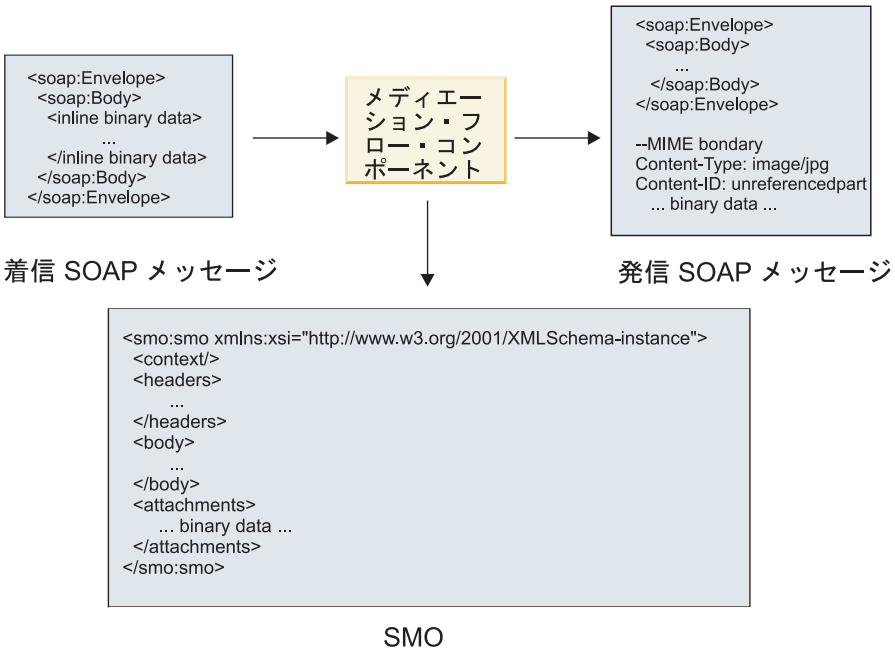


図 25. メディエーション・フロー・コンポーネントによって処理されるメッセージ

逆に、メディエーション・フロー・コンポーネントは、添付ファイルを抽出してエンコードしたあと添付ファイルなしでメッセージを送信することにより、着信メッセージを変換できます。

着信 SOAP メッセージからデータを抽出して添付ファイル付きの SOAP メッセージを作成する代わりに、データベースなどの外部ソースから添付ファイル・データを取得できます。

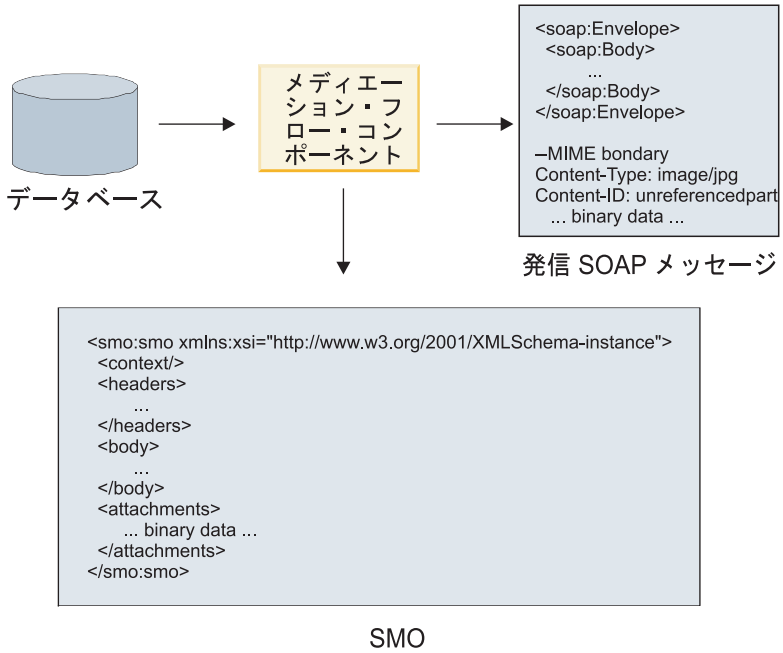


図 26. データベースから取得して SOAP メッセージに追加した添付ファイル

逆に、メディエーション・フロー・コンポーネントは着信 SOAP メッセージから添付ファイルを抽出して、メッセージを処理 (例えば、添付ファイルをデータベースに保管) できます。

参照されていない添付ファイルは、メディエーション・フロー・コンポーネントを介した場合にのみ伝搬できます。添付ファイルに別のコンポーネント・タイプでアクセスする必要がある場合や、別のコンポーネント・タイプに伝搬する必要がある場合には、メディエーション・フロー・コンポーネントを使用して、そのコンポーネントがアクセスできる場所へ添付ファイルを移動してください。

重要: 『SMO の XML 表記』で説明しているように、XSL 変換メディエーション・プリミティブは、XSLT 1.0 変換を使用してメッセージを変換します。変換は、SMO の XML 直列化で作動します。XSL 変換メディエーション・プリミティブより、直列化のルートを指定することができます。また、XML 文書のルート・エレメントはこのルートを反映します。

SOAP メッセージを添付ファイルと共に送信する場合、選択したルート・エレメントによって、添付ファイルの伝搬方法が決まります。

- 「/body」を XML マップのルートとして使用した場合は、デフォルトですべての添付ファイルがマップ全体に伝搬します。
- 「/」をマップのルートとして使用した場合は、添付ファイルの伝搬を制御できます。

複数パーツ・メッセージでの WSDL 文書スタイルのバインディングの使用

Web Services Interoperability Organization (WS-I) 組織により、相互運用性の実現を目的として、WSDL を使用した Web サービスの記述方法と、対応する SOAP メッセージの形成方法に関する一連の規則が定義されました。

これらの規則は、WS-I *Basic Profile バージョン 1.1* (http://www.ws-i.org/Profiles/BasicProfile-1.1.html)に明記されています。

特に、文書スタイルの SOAP バインディングについては、WS-I プロファイルに適合するには、以下の 2 点が要求されます。WSDL 文書内で文書スタイルを使用する操作では、必ず SOAP 本体にメッセージ・パーツが 1 つだけバインドされること。これに対応する SOAP メッセージには、前述のようにバインドされたパーツと一致する単一の子エレメントが含まれること。

つまり、複数のパーツを使用してメッセージ (入力、出力、または障害) が定義されている操作に文書スタイルの SOAP バインディングを使用する場合、WS-I Basic Profile 1.1 に準拠するには、これらのパーツのうち 1 つしか SOAP 本体とバインドできないことになります。

この場合、Web サービス (JAX-WS および JAX-RPC) のバインディングを使用したエクスポート用に WSDL 記述を生成する際には、以下のアプローチが使用されます。

- 最初のメッセージ・パーツが SOAP 本体にバインドされます。

- JAX-WS バインディングの場合、タイプ「hexBinary」または「base64Binary」のその他のメッセージ・パーツはすべて、参照されている添付ファイルとしてバインドされます。 46 ページの『参照されている添付ファイル: 最上位メッセージ・パーツ』を参照してください。
- それ以外のメッセージ・パーツはすべて SOAP ヘッダーとしてバインドされます。

JAX-RPC および JAX-WS インポート・バインディングでは、複数パーツの文書スタイル・メッセージを持つ既存の WSDL 文書において、複数のパーツが SOAP 本体にバインドされる場合でも、SOAP バインディングが尊重されます。ただし、Rational Application Developer でこのような WSDL 文書用の Web サービス・クライアントを生成することはできません。

注: JAX-RPC バインディングでは、添付ファイルはサポートされていません。

したがって、文書スタイルの SOAP バインディングを使用する操作で複数パーツ・メッセージを使用する場合は、以下のパターンが推奨されます。

1. 文書/リテラル折り返しスタイルを使用します。この場合、メッセージに含まれるパーツは必ず 1 つになります。ただしこの場合は、参照されていない添付ファイル (51 ページの『参照されていない添付ファイル』を参照) または swaRef タイプの添付ファイル (42 ページの『参照されている添付ファイル: swaRef タイプのエレメント』を参照) を使用できます。
2. RPC/リテラル・スタイルを使用します。この場合、 SOAP 本体にバインドされるパーツの数について WSDL バインディングに課せられる制限はありません。結果的に SOAP メッセージには必ず、呼び出される操作を表す 1 つの子と、そのエレメントの子となる複数のメッセージ・パーツが含まれます。
3. JAX-WS バインディングの場合は、最初のメッセージ・パーツをタイプ「hexBinary」または「base64Binary」以外のパーツにします。この 2 つのタイプのいずれかに該当するその他のパーツはすべて添付ファイルとしてバインドされます。
4. それ以外の場合は、前述の動作に従います。

注: 参照されている添付ファイルのある複数パーツ文書スタイルの SOAP メッセージを受信する場合、JAX-WS バインディングでは、参照されている添付ファイルのそれぞれが SOAP 本体の子エレメントで表され、その子エレメントは href 属性を持ち、属性値がコンテンツ ID によってその添付ファイルを識別している必要があります。JAX-WS バインディングは、そのようなメッセージの参照されている添付ファイルを同じ方法で送信します。この動作は、WS-I Basic Profile には準拠していません。WS-I Attachments Profile では、href 属性を持つ子エレメントを省略でき、したがってそのようなメッセージを Basic Profile 準拠にすることができる「コンテンツ ID パーツのエンコード」について定義されています。JAX-WS バインディングでは、コンテンツ ID パーツのエンコードを使用するメッセージの送受信をサポートしていません。メッセージを準拠させるには、上記のリストのアプローチ 1 または 2 に従うか、そのようなメッセージに参照されている添付ファイルを使用せず、代わりに、参照されていない添付ファイルまたは swaRef タイプの添付ファイルを使用してください。

HTTP バインディング

HTTP バインディングは、Service Component Architecture (SCA) と HTTP の接続を提供する目的で設計されています。その結果、既存または新規開発の HTTP アプリケーションは、サービス指向アーキテクチャー (SOA) 環境に加わることができます。

Hypertext Transfer Protocol (HTTP) は、Web 上での情報転送プロトコルとして広く使用されています。HTTP プロトコルを使用する外部アプリケーションを扱う場合には、HTTP バインディングが必要です。HTTP バインディングは、メッセージとして渡されたデータのネイティブ・フォーマットから SCA アプリケーションのビジネス・オブジェクトへの変換を処理します。着信メッセージの場合、HTTP バインディングは、ビジネス・オブジェクトとして渡されたデータを、外部アプリケーションが期待するネイティブ・フォーマットに変換することもできます。

注: Web サービスの SOAP/HTTP プロトコルを使用するクライアントおよびサーバーと対話する場合は、Web サービス・バインディングのいずれかを使用することを検討してください。これにより、Web サービスの標準サービス品質の処理に関する追加機能が提供されます。

以下に、HTTP バインディングを使用する一般的なシナリオをリストして説明します。

- SCA によってホストされるサービスは、HTTP インポートを使用して HTTP アプリケーションを呼び出すことができます。
- SCA によってホストされるサービスは、HTTP 対応アプリケーションとして自己公開することができます。その場合、HTTP クライアントは HTTP エクスポートを使用して、これらのサービスを使用できるようになります。
- WebSphere Process Server および WebSphere Enterprise Service Bus は、HTTP インフラストラクチャーを介して互いに通信できるため、ユーザーは企業の標準に従って通信を管理できます。
- WebSphere Process Server および WebSphere Enterprise Service Bus は、HTTP 通信のメディエーターとしてメッセージを変換およびルーティングし、HTTP ネットワークを使用するアプリケーションの統合を強化することができます。
- WebSphere Process Server および WebSphere Enterprise Service Bus は、HTTP とその他のプロトコル (SOAP/HTTP Web サービス、Java Connector Architecture (JCA) ベースのリソース・アダプター、JMS など) との間のブリッジとして使用できます。

HTTP インポートおよびエクスポート・バインディングの作成に関する詳細情報については、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。**『統合アプリケーションの開発』 → 『HTTP を使用した外部サービスへのアクセス』**トピックを参照してください。

HTTP バインディングの概要

HTTP バインディングは、HTTP がホストするアプリケーションに接続を提供します。HTTP アプリケーション間の通信をメディエーションし、既存の HTTP ベースのアプリケーションをモジュールから呼び出せるようにします。

HTTP インポート・バインディング

HTTP インポート・バインディングは、Service Component Architecture (SCA) アプリケーションから HTTP サーバーまたはアプリケーションへのアウトバウンド接続を提供します。

インポートは、HTTP エンドポイントの URL を呼び出します。この URL は、以下の 3 つの方法のいずれかで指定できます。

- 動的オーバーライド URL を使用して、HTTP ヘッダー内に URL を動的に設定します。
- SMO ターゲット・アドレス・エレメント内に URL を動的に設定します。
- インポートの構成プロパティーとして URL を指定します。

本来、この呼び出しは常に同期呼び出しとなります。

HTTP 呼び出しは常に要求/応答となりますが、HTTP インポートは片方向操作と両方向操作の両方をサポートし、操作が片方向の場合には応答を無視します。

HTTP エクスポート・バインディング

HTTP エクスポート・バインディングは、HTTP アプリケーションから SCA アプリケーションへのインバウンド接続を提供します。

URL は、HTTP エクスポートで定義されます。要求メッセージをエクスポートに送信する必要がある HTTP アプリケーションは、この URL を使用してエクスポートを呼び出します。

HTTP エクスポートは ping もサポートします。

実行時の HTTP バインディング

実行時に HTTP バインディングを使用するインポートは、メッセージ本体 (データの有無にかかわらず) を使用して、SCA アプリケーションから外部 Web サービスに要求を送信します。要求は、図 27 に示すように、SCA アプリケーションから外部 Web サービスに対して行われます。

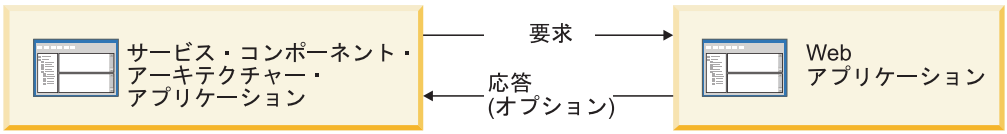


図 27. SCA アプリケーションから Web アプリケーションへの要求フロー

オプションで、HTTP バインディングを使用するインポートは要求に対する応答として Web アプリケーションからのデータを受信することができます。

エクスポートの場合は、58 ページの図 28 に示すように、クライアント・アプリケーションが Web サービスに対して要求を行います。

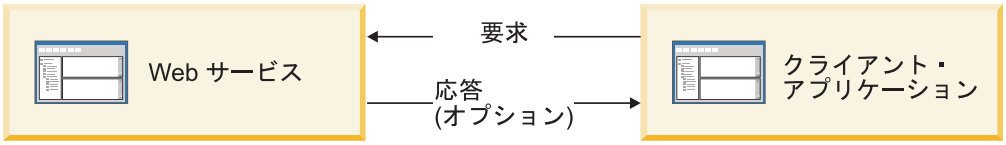


図 28. クライアント・アプリケーションから Web サービスへの要求フロー

Web サービスは、サーバーで稼働する Web アプリケーションです。この Web アプリケーションでは、エクスポートはサーブレットとして実装されるため、クライアントはその要求を URL アドレスに送信します。サーブレットは、実行時に要求を SCA アプリケーションに渡します。

オプションで、エクスポートは要求に対する応答として、クライアント・アプリケーションにデータを送信することができます。

HTTP ヘッダー

HTTP インポートおよびエクスポート・バインディングを使用すると、HTTP ヘッダーを構成し、その値をアウトバウンド・メッセージに使用できます。HTTP インポートはそれらのヘッダーを要求に使用し、HTTP エクスポートは応答に使用します。

静的に構成されたヘッダーと制御情報は、実行時に動的に設定される値よりも優先されます。しかし、動的オーバーライド URL、バージョン、およびメソッド制御の値は、静的な値をオーバーライドします。それ以外の場合、静的な値がデフォルト値と見なされます。

バインディングは、実行時に HTTP ターゲットの URL、バージョン、およびメソッド値を判断することにより、HTTP インポート URL の動的性質をサポートします。これらの値は、エンドポイント参照、動的オーバーライド URL、バージョン、メソッドの値を抽出して判別されます。

- エンドポイント参照については、com.ibm.websphere.sca.addressing.EndpointReference API を使用するか、または SMO ヘッダーの /headers/SMOHeader/Target/address フィールドを設定します。
- 動的オーバーライド URL、バージョン、およびメソッドについては、Service Component Architecture (SCA) メッセージの HTTP 制御パラメーター・セクションを使用します。動的オーバーライド URL は、ターゲットのエンドポイント参照よりも優先されることに注意してください。ただし、エンドポイント参照はバインディング全体に適用されるため、望ましいアプローチです。このため、可能であればエンドポイント参照を使用してください。

注: 動的呼び出しについて、および URL のフォーマット、構文、使用方法について固有の情報は、『関連概念』を参照してください。

HTTP エクスポートおよびインポート・バインディングのアウトバウンド・メッセージに関する制御およびヘッダー情報は、次の順序で処理されます。

1. SCA メッセージからの HTTP 動的オーバーライド URL、バージョン、およびメソッドを除くヘッダーおよび制御情報 (最低の優先順位)
2. エクスポート/インポート・レベルでの管理コンソールからの変更

3. エクスポートまたはインポートのメソッド・レベルでの管理コンソールからの変更
4. エンドポイント参照または SMO ヘッダーによって指定されたターゲット・アドレス
5. SCA メッセージからの動的オーバーライド URL、バージョン、およびメソッド
6. データ・ハンドラーまたはデータ・バインディングからのヘッダーおよび制御情報 (最高の優先順位)

HTTP エクスポートおよびインポートは、プロトコル・ヘッダーの伝搬が `True` に設定されている場合にのみ、インバウンド方向のヘッダーと制御パラメーターに、着信メッセージからのデータ (HTTPExportRequest および HTTPImportResponse) の取り込みを行います。逆に、アウトバウンドのヘッダーと制御パラメーター (HTTPExportResponse および HTTPImportRequest) については、プロトコル・ヘッダーの伝搬が `True` に設定されている場合にのみ、HTTP エクスポートおよびインポートが読み取りと処理を行います。

注: インポート応答またはエクスポート要求のなかでデータ・ハンドラーまたはデータ・バインディングがヘッダーまたは制御パラメーターに変更されても、インポートまたはエクスポート・バインディング内でのメッセージの処理命令が変更されることはありません。そのため、データ・ハンドラーまたはデータ・バインディングは、変更された値をダウンストリームの SCA コンポーネントに伝搬する目的のみで使用する必要があります。

コンテキスト・サービスは、コンテキスト (HTTP ヘッダーなどのプロトコル・ヘッダーや、アカウント ID などのユーザー・コンテキストを含む) を SCA 呼び出しパスに沿って伝搬します。 WebSphere Integration Developer での開発時には、インポート・プロパティーとエクスポート・プロパティーを使用してコンテキストの伝搬を制御できます。詳しくは、WebSphere Integration Developer インフォメーション・センターのインポートおよびエクスポート・バインディングに関する情報を参照してください。

提供される HTTP ヘッダー構造とサポート

表 12 に、HTTP インポートと HTTP エクスポートの要求および応答の要求/応答パラメーターを項目別に示します。

表 12. 提供される HTTP ヘッダー情報

制御名	HTTP インポート要求	HTTP インポート応答	HTTP エクスポート要求	HTTP エクスポート応答
URL	無視される	設定されない	要求メッセージから読み取られる。 **注:** 照会ストリングも URL 制御パラメーターの一部です。	無視される

表 12. 提供される HTTP ヘッダー情報 (続き)

制御名	HTTP インポート要求	HTTP インポート応答	HTTP エクスポート要求	HTTP エクスポート応答
バージョン (可能な値: 1.0、1.1。デフォルトは 1.1)	無視される	設定されない	要求メッセージから読み取られる	無視される
メソッド	無視される	設定されない	要求メッセージから読み取られる	無視される
動的オーバーライド URL	データ・ハンドラーまたはデータ・バインディングに設定された場合、HTTP Import URL をオーバーライドする。要求行のメッセージに書き込まれる。 **注:** 照会ストリングも URL 制御パラメーターの一部です。	設定されない	設定されない	無視される
動的オーバーライド・バージョン	設定された場合、HTTP Import バージョンをオーバーライドする。要求行のメッセージに書き込まれる。	設定されない	設定されない	無視される
動的オーバーライド・メソッド	設定された場合、HTTP Import メソッドをオーバーライドする。要求行のメッセージに書き込まれる。	設定されない	設定されない	無視される

表 12. 提供される HTTP ヘッダー情報 (続き)

制御名	HTTP インポート要求	HTTP インポート応答	HTTP エクスポート要求	HTTP エクスポート応答
メディア・タイプ (この制御パラメーターは、Content-Type HTTP ヘッダーの値の一部を搬送します。)	存在する場合、メッセージに Content-Type ヘッダーの一部として書き込まれる。 **注:** この制御エレメント値は、データ・ハンドラーまたはデータ・バインディングによって提供してください。	応答メッセージ、Content-Type ヘッダーから読み取られる	要求メッセージ、Content-Type ヘッダーから読み取られる	存在する場合、メッセージに Content-Type ヘッダーの一部として書き込まれる。 **注:** この制御エレメント値は、データ・ハンドラーまたはデータ・バインディングによって提供してください。
文字セット (デフォルト: UTF-8)	存在する場合、メッセージに Content-Type ヘッダーの一部として書き込まれる。 **注:** この制御エレメント値は、データ・バインディングによって提供してください。	応答メッセージ、Content-Type ヘッダーから読み取られる	要求メッセージ、Content-Type ヘッダーから読み取られる	サポート対象。メッセージに Content-Type ヘッダーの一部として書き込まれる。 **注:** この制御エレメント値は、データ・バインディングによって提供してください。
転送エンコード (可能な値: chunked、identity。デフォルトは identity)	存在する場合、メッセージにヘッダーとして書き込まれ、メッセージ変換のエンコードの方法を制御する。	応答メッセージから読み取られる	要求メッセージから読み取られる	存在する場合、メッセージにヘッダーとして書き込まれ、メッセージ変換のエンコードの方法を制御する。
コンテンツ・エンコード (可能な値: gzip、x-gzip、deflate、identity。デフォルトは identity)	存在する場合、メッセージにヘッダーとして書き込まれ、ペイロードのエンコードの方法を制御する。	応答メッセージから読み取られる	要求メッセージから読み取られる	存在する場合、メッセージにヘッダーとして書き込まれ、ペイロードのエンコードの方法を制御する。
Content-Length	無視される	応答メッセージから読み取られる	要求メッセージから読み取られる	無視される
StatusCode (デフォルト: 200)	サポート対象外	応答メッセージから読み取られる	サポート対象外	存在する場合、応答行のメッセージに書き込まれる。

表 12. 提供される HTTP ヘッダー情報 (続き)

制御名	HTTP インポート要求	HTTP インポート応答	HTTP エクスポート要求	HTTP エクスポート応答
ReasonPhrase (デフォルト: OK)	サポート対象外	応答メッセージから読み取られる	サポート対象外	制御値は無視される。メッセージ応答行の値は StatusCode から生成される。
認証 (複数のプロパティーを含む)	存在する場合、基本認証ヘッダーの作成に使用される。 **注:** このヘッダーの値は、HTTP プロトコル上でのみエンコードされます。SCA では、これはデコードされ、平文として渡されます。	適用外	要求メッセージの基本認証ヘッダーから読み取られる。このヘッダーの存在は、ユーザーが認証済みであることを示すものではありません。認証は、サーブレット構成で制御する必要があります。 **注:** このヘッダーの値は、HTTP プロトコル上でのみエンコードされます。SCA では、これはデコードされ、平文として渡されます。	適用外
プロキシー (次のような複数のプロパティーを含む: Host、Port、Authentication)	存在する場合、プロキシーによる接続を確立するために使用される。	適用外	適用外	適用外
SSL (次のような複数のプロパティーを含む: Keystore、Keystore Password、Trustore、Trustore Password、ClientAuth)	入力され、宛先 URL が HTTPS の場合、これは SSL による接続を確立するために使用される。	適用外	適用外	適用外

HTTP データ・バインディング

Service Component Architecture (SCA) メッセージと HTTP プロトコル・メッセージ間のデータの異なるマッピングごとに、データ・ハンドラーまたは HTTP データ・バインディングを構成する必要があります。データ・ハンドラーは、バインディングに関して中立的なインターフェースを提供します。 このインターフェースは複数のトランスポート・バインディングに渡って再利用が可能であり、推奨される方法を表します。データ・バインディングは特定のトランスポート・バインディングに固有です。HTTP 固有のデータ・バインディング・クラスが提供されるほか、カスタム・データ・ハンドラーまたはデータ・バインディングを作成することもできます。

注: このトピックで説明した 3 つの HTTP データ・バインディング・クラス (HTTPStreamDataBindingSOAP、HTTPStreamDataBindingXML、および HTTPServiceGatewayDataBinding) は、WebSphere Process Server バージョン 7.0 では推奨されていません。このトピックで説明したデータ・バインディングを使用する代わりに、次のデータ・ハンドラーの使用を検討してください。

- HTTPStreamDataBindingSOAP の代わりに SOAPDataHandler を使用する
- HTTPStreamDataBindingXML の代わりに UTF8XMLDataHandler を使用する
- HTTPServiceGatewayDataBinding の代わりに GatewayTextDataHandler を使用する

HTTP インポートおよび HTTP エクスポートに使用するデータ・バインディングが提供されます。それらは、バイナリー・データ・バインディング、XML データ・バインディング、および SOAP データ・バインディングです。応答データ・バインディングは、片方向の操作には必要ありません。データ・バインディングは、インスタンスが HTTP と ServiceDataObject の間で両方向に変換可能な Java クラスの名前によって表されます。エクスポートでは関数セレクターを使用する必要があります。関数セレクターは、使用されるデータ・バインディングと呼び出される操作を、メソッド・バインディングと連動して判別できます。提供されるデータ・バインディングは、次のとおりです。

- バイナリー・データ・バインディング。本文を非構造化バイナリー・データとして取り扱います。バイナリー・データ・バインディングの XSD スキーマの実装は、次のとおりです。

```
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://com.ibm.websphere.http.data.bindings/schema"
  xmlns:tns="http://com.ibm.websphere.http.data.bindings/schema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="HTTPBaseBody">
    <xsd:sequence/>
  </xsd:complexType>

  <xsd:complexType name="HTTPBytesBody">
    <xsd:complexContent>
      <xsd:extension base="tns:HTTPBaseBody">
        <xsd:sequence>
          <xsd:element name="value" type="xsd:hexBinary"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

- XML データ・バインディング。本文を XML データとしてサポートします。XML データ・バインディングの実装は JMS XML データ・バインディングに類似しており、インターフェース・スキーマについての制限はありません。
- SOAP データ・バインディング。本文を SOAP データとしてサポートします。SOAP データ・バインディングの実装には、インターフェース・スキーマについての制限はありません。

カスタム HTTP データ・バインディングの実装

このセクションでは、カスタム HTTP データ・バインディングを実装する方法を説明します。

注: 推奨される方法は、カスタム・データ・ハンドラーを実装する方法です。カスタム・データ・ハンドラーは複数のトランスポート・バインディング間で再利用できるためです。

HTTPStreamDataBinding は、カスタム HTTP メッセージを処理するための主要なインターフェースです。このインターフェースは大きなペイロードを処理できるように設計されています。しかし、そのような実装を機能させるために、このデータ・バインディングは、ストリームにメッセージを書き込む前に制御情報とヘッダーを返す必要があります。

メソッドとその実行順序 (以下を参照) は、カスタム・データ・バインディングによって実装する必要があります。

データ・バインディングをカスタマイズするには、HTTPStreamDataBinding を実装するクラスを作成します。データ・バインディングには、以下の 4 つの private プロパティーが必要です。

- private DataObject pDataObject
- private HTTPControl pCtrl
- private HTTPHeaders pHeaders
- private yourNativeDataType nativeData

HTTP バインディングは、カスタマイズされたデータ・バインディングを以下の順序で呼び出します。

- アウトバウンド処理 (DataObject からネイティブ・フォーマットへ):
  1. setDataObject(...)
  2. setHeaders(...)
  3. setControlParameters(...)
  4. setBusinessException(...)
  5. convertToNativeData()
  6. getControlParameters()
  7. getHeaders()
  8. write(...)
- インバウンド処理 (ネイティブ・フォーマットから DataObject へ):
  1. setControlParameters(...)

2. setHeaders(...)
3. convertFromNativeData(...)
4. isBusinessException()
5. getDataObject()
6. getControlParameters()
7. getHeaders()

convertFromNativeData(...) で setDataObject(...) を呼び出して、dataObject の値を設定する必要があります。dataObject は、ネイティブ・データから private プロパティー「pDataObject」に変換されます。

```
public void setDataObject(DataObject dataObject)
   throws DataBindingException {
  pDataObject = dataObject;

}
public void setControlParameters(HTTPControl arg0) {
  this.pCtrl = arg0;
}

public void setHeaders(HTTPHeaders arg0) {
  this.pHeaders = arg0;
}
/*
* pHeaders に HTTP ヘッダー「IsBusinessException」を追加します。
* 以下の 2 つのステップを実行します。
* 1. まず IsBusinessException という名前 (大/小文字を区別しない) を
*    持つヘッダーをすべて削除します。
*    これは、1 つのヘッダーのみが存在するようにするためです。
* 2. 新しいヘッダー「IsBusinessException」を追加します。
*/
public void setBusinessException(boolean isBusinessException) {
  // まず IsBusinessException という名前 (大/小文字を区別しない) を
  // 持つヘッダーをすべて削除します。
  // これは、1 つのヘッダーのみが存在するようにするためです。
  // 新しいヘッダー「IsBusinessException」を追加します。コード例は以下のとおりです。
  HTTPHeader header=HeadersFactory.eINSTANCE.createHTTPHeader();
  header.setName("IsBusinessException");
  header.setValue(Boolean.toString(isBusinessException));
  this.pHeaders.getHeader().add(header);

}
public HTTPControl getControlParameters() {
  return pCtrl;
}
public HTTPHeaders getHeaders() {
  return pHeaders;
}
public DataObject getDataObject() throws DataBindingException {
  return pDataObject;
}
/*
* pHeaders からヘッダー「IsBusinessException」を取得し、そのブール値を返す
*/
public boolean isBusinessException() {
  String headerValue = getHeaderValue(pHeaders,"IsBusinessException");
  boolean result=Boolean.parseBoolean(headerValue);
  return result;
}
public void convertToNativeData() throws DataBindingException {
  DataObject dataObject = getDataObject();
  this.nativeData=realConvertWorkFromSDOToNativeData(dataObject);
}
```

```
public void convertFromNativeData(HTTPInputStream arg0){
  //HTTPInputStream からデータを読み取る
  //ユーザー開発のメソッド
  // DataObject に変換する
  DataObject dataobject=realConvertWorkFromNativeDataToSDO(arg0);
  setDataObject(dataobject);
}
public void write(HTTPOutputStream output) throws IOException {
  if (nativeData != null)
  output.write(nativeData);
}
```

EJB バインディング

Enterprise JavaBeans (EJB) インポート・バインディングにより、Service Component Architecture (SCA) コンポーネントは、Java EE サーバー上で実行される Java EE ビジネス・ロジックで提供されるサービスを起動することができます。EJB エクスポート・バインディングにより、 SCA コンポーネントを Enterprise JavaBeans として公開することができます。これにより、Java EE ビジネス・ロジックは、これ以外の方法では使用できない SCA コンポーネントを起動できるようになります。

EJB インポート・バインディング

EJB インポート・バインディングでは、コンシュームする側のモジュールと外部 EJB とをバインドする方法を指定することによって、SCA モジュールが EJB 実装を呼び出すことができます。外部 EJB 実装からサービスをインポートすることによって、ユーザーはビジネス・ロジックを WebSphere Process Server 環境に接続でき、ビジネス・プロセスに参加することができます。

WebSphere Integration Developer を使用して EJB インポート・バインディングを作成します。以下のいずれかの手順により、バインディングを生成することができます。

- 外部サービス・ウィザードを使用して EJB インポート・バインディングを作成する

  WebSphere Integration Developer の外部サービス・ウィザードを使用し、既存の実装に基づいて EJB インポートを作成できます。外部サービス・ウィザードは、指定された基準に基づいてサービスを作成します。次に、ディスカバーされたサービスに基づいて、ビジネス・オブジェクト、インターフェース、インポート・ファイルを生成します。
- アセンブリー・エディターを使用して EJB インポートを作成する

  WebSphere Integration Developer アセンブリー・エディターを使用して、アセンブリー・ダイアグラム内に EJB インポートを作成できます。パレットで、インポートを使用するか、Java クラスを使用して、EJB バインディングを作成できます。

生成されたインポートには、Java ブリッジ・コンポーネントを必要とする代わりに Java-WSDL 接続を行うデータ・バインディングがあります。Web Services Description Language (WSDL) 参照を持つコンポーネントを、Java インターフェースを使って EJB ベースのサービスと通信する EJB インポートに直接接続できます。

EJB インポートは、EJB 2.1 プログラミング・モデルまたは EJB 3.0 プログラミング・モデルを使用して、Java EE ビジネス・ロジックと対話することができます。

Java EE ビジネス・ロジックの呼び出しは、ローカル (EJB 3.0 の場合のみ) でもリモートでもかまいません。

- ローカル呼び出しは、インポートと同じサーバーに存在する Java EE ビジネス・ロジックを呼び出す場合に使用します。

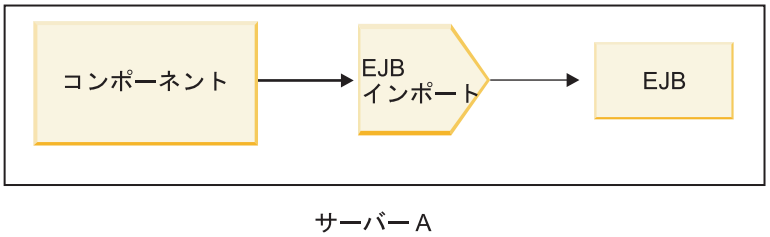


図 29. EJB (EJB 3.0 のみ)のローカル呼び出し

- リモート呼び出しは、インポートと同じサーバーに存在しない Java EE ビジネス・ロジックを呼び出す場合に使用します。

  例えば、次の図で、EJB インポートは、Remote Method Invocation over Internet InterORB Protocol (RMI/IIOP) を使用して、別のサーバーの EJB メソッドを呼び出しています。

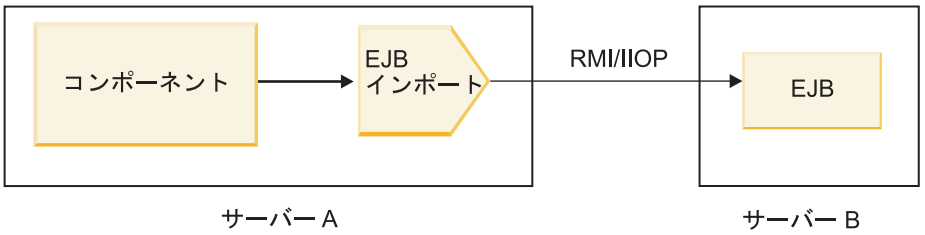


図 30. EJB のリモート呼び出し

EJB バインディングの構成時に、WebSphere Integration Developer は JNDI 名を使用して、EJB プログラミング・モデル・レベルと呼び出しのタイプ (ローカルまたはリモート) を判別します。

EJB インポート・バインディングには、以下の主なコンポーネントが含まれています。

- JAX-WS データ・ハンドラー
- EJB 障害セレクター
- EJB インポート関数セレクター

ユーザー・シナリオが JAX-WS マッピングに基づいていない場合は、タスクを実行するために、カスタム・データ・ハンドラー、関数セレクター、および障害セレクターが必要になることがあります。このタスクは、ユーザー・シナリオがこのマッピングに基づいている場合には、EJB インポート・バインディングを構成するコンポーネントによって実行されるタスクです。これにはカスタム・マッピング・アルゴリズムによって通常実行されるマッピングが含まれます。

EJB エクスポート・バインディング

外部 Java EE アプリケーションは、EJB エクスポート・バインディングによって SCA コンポーネントを呼び出すことができます。 EJB エクスポートを使用することにより、SCA コンポーネントを公開することができます。これにより、外部 Java EE アプリケーションは、EJB プログラミング・モデルを使用してこれらのコンポーネントを呼び出せるようになります。

注: EJB エクスポートは、ステートレス Bean です。

WebSphere Integration Developer を使用して EJB バインディングを作成します。以下のいずれかの手順により、バインディングを生成することができます。

- 外部サービス・ウィザードを使用して EJB エクスポート・バインディングを作成する

  WebSphere Integration Developer の外部サービス・ウィザードを使用し、既存の実装に基づいて EJB エクスポート・サービスを作成できます。外部サービス・ウィザードは、指定された基準に基づいてサービスを作成します。次に、ディスカバーされたサービスに基づいて、ビジネス・オブジェクト、インターフェース、エクスポート・ファイルを生成します。
- アセンブリー・エディターを使用して EJB エクスポート・バインディングを作成する

  WebSphere Integration Developer アセンブリー・エディターを使用して、EJB エクスポートを作成できます。

既存の SCA コンポーネントからバインディングを生成できます。または、Java インターフェース用の EJB バインディングを使ってエクスポートを生成できます。

- 既存の WSDL インターフェースを持つ既存の SCA コンポーネント用のエクスポートを生成するときは、エクスポートは Java インターフェースに割り当てられます。
- Java インターフェース用のエクスポートを生成するときは、そのエクスポート用に WSDL または Java インターフェースを選択できます。

  注: EJB エクスポートの作成で使用する Java インターフェースには、リモート呼び出しでパラメーターとして渡されるオブジェクト (入出力パラメーターと例外) に関して次のような制約があります。
  - 具象タイプである必要がある (インターフェースや抽象タイプではない)。
  - これらは、Enterprise JavaBean の仕様に準拠している必要があります。これらは、直列化可能で、デフォルトの引数のないコンストラクターを持ち、すべてのプロパティーは getter メソッドおよび setter メソッドを使ってアクセスできる必要があります。

    Enterprise JavaBean の仕様については、Sun Microsystems, Inc. の Web サイト (http://java.sun.com) を参照してください。

  また、例外はチェック例外である必要があり、java.lang.Exception から継承され、単数形である必要があります (つまり、複数のチェック例外タイプのスローをサポートしていません)。

Java EnterpriseBean のビジネス・インターフェースは簡潔な Java インターフェースであり、javax.ejb.EJBObject または javax.ejb.EJBLocalObject を拡張してはいけないという点に留意してください。ビジネス・インターフェースのメソッドは、java.rmi.Remote.Exception をスローしてはいけません。

EJB エクスポート・バインディングは、EJB 2.1 プログラミング・モデルまたは EJB 3.0 プログラミング・モデルを使用して、Java EE ビジネス・ロジックと対話することができます。

呼び出しは、ローカル (EJB 3.0 の場合のみ) でもリモートでもかまいません。

- ローカルの呼び出しは、Java EE ビジネス・ロジックがエクスポートと同じサーバー上にある SCA コンポーネントを呼び出したときに使われます。
- リモートの呼び出しは、Java EE ビジネス・ロジックがエクスポートと同じサーバー上にないときに使われます。

例えば、次の図で、EJB は RMI/IIOP を使用して、別のサーバー上の SCA コンポーネントを呼び出しています。

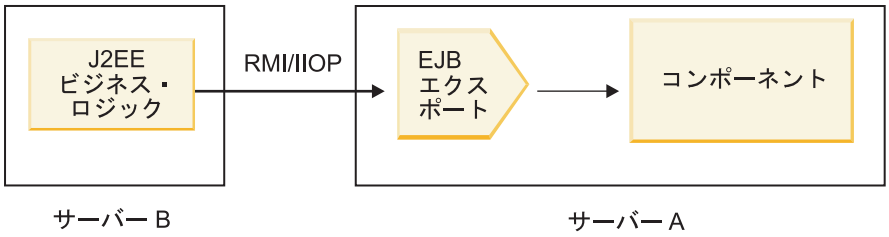


図 31. EJB エクスポートによるクライアントから SCA コンポーネントに対するリモート呼び出し

EJB バインディングの構成時に、WebSphere Integration Developer は JNDI 名を使用して、EJB プログラミング・モデル・レベルと呼び出しのタイプ (ローカルまたはリモート) を判別します。

EJB エクスポート・バインディングには、以下の主なコンポーネントが含まれています。

- JAX-WS データ・ハンドラー
- EJB エクスポート関数セレクター

お使いのユーザー・シナリオが JAX-WS マッピングに基づいていない場合は、タスクを実行するためにカスタム・データ・ハンドラーおよび関数セレクターが必要になることがあります。そうでない場合は、EJB エクスポート・バインディングの一部であるコンポーネントにより処理が完了します。これにはカスタム・マッピング・アルゴリズムによって通常実行されるマッピングが含まれます。

EJB バインディング・プロパティー

EJB インポート・バインディングは、構成済みの JNDI 名を使用して、EJB プログラミング・モデル・レベルと呼び出しのタイプ (ローカルまたはリモート) を判別します。EJB インポートおよびエクスポート・バインディングは、データ変換に JAX-WS データ・ハンドラーを使用します。EJB インポート・バインディングは

EJB インポート関数セレクターと EJB 障害セレクターを使用し、EJB エクスポート・バインディングは EJB エクスポート関数セレクターを使用します。

JNDI 名および EJB インポート・バインディング:

インポートに対する EJB バインディングの構成時に、WebSphere Integration Developer は JNDI 名を使用して、EJB プログラミング・モデル・レベルと呼び出しのタイプ (ローカルまたはリモート) を判別します。

JNDI 名が指定されていない場合、デフォルトの EJB インターフェース・バインディングが使用されます。作成されるデフォルトの名前は、EJB 2.1 JavaBean と EJB 3.0 JavaBean のいずれを呼び出すかによって異なります。

注: 命名規則に関する詳細については、WebSphere Application Server インフォメーション・センターのEJB 3.0 アプリケーション・バインディングの概要を参照してください。

- EJB 2.1 JavaBean

  WebSphere Integration Developer によって事前選択されているデフォルトの JNDI 名は、デフォルトの EJB 2.1 バインディングであり、`ejb/` の後にスラッシュで区切られたホーム・インターフェースが続く形式になっています。

  例えば、com.mycompany.myremotebusinesshome の EJB 2.1 JavaBean のホーム・インターフェースの場合、デフォルトのバインディングは以下のとおりです。

  ```
  ejb/com/mycompany/myremotebusinesshome
  ```

  EJB 2.1 の場合は、リモート EJB 呼び出しのみがサポートされます。

- EJB 3.0 JavaBean

  WebSphere Integration Developer によって事前選択されているローカル JNDI 用のデフォルトの JNDI 名は、先頭に `ejblocal:` が付されたローカル・インターフェースの完全修飾クラス名になります。例えば、ローカル・インターフェース com.mycompany.mylocalbusiness の完全修飾インターフェースの場合、事前選択された EJB 3.0 JNDI は、以下のとおりです。

  ```
  ejblocal:com.mycompany.mylocalbusiness
  ```

  リモート・インターフェース com.mycompany.myremotebusiness の場合、事前選択された EJB 3.0 JNDI は、以下の完全修飾インターフェースになります。

  ```
  com.mycompany.myremotebusiness
  ```

  EJB 3.0 デフォルト・アプリケーション・バインディングの説明は、EJB 3.0 アプリケーション・バインディングの概要に記載されています。

  WebSphere Integration Developer は、バージョン 3.0 のプログラミング・モデルを使用する EJB 用のデフォルトの JNDI の場所として短縮名を使用します。

  注: カスタム・マッピングが使用または構成されていたために、ターゲット EJB のデプロイ済み JNDI 参照が、デフォルトの JNDI バインディングの場所と異なっている場合は、ターゲット JNDI 名を正しく指定する必要があります。デプロイメント前に、WebSphere Integration Developer で名前を指定することができま

す。あるいは、インポート・バインディングの場合は、(デプロイメント後に) ターゲット EJB の JNDI 名と一致するように管理コンソールで名前を変更することもできます。

EJB バインディングの作成について詳しくは、WebSphere Integration Developer インフォメーション・センターで EJB バインディングの処理について扱われているセクションを参照してください。

JAX-WS データ・ハンドラー:

Enterprise JavaBeans (EJB) インポート・バインディングは、JAX-WS データ・ハンドラーを使用して、要求ビジネス・オブジェクトを Java オブジェクト・パラメーターに変換し、Java オブジェクト戻り値を応答ビジネス・オブジェクトに変換します。 EJB エクスポート・バインディングは、JAX-WS データ・ハンドラーを使用して、要求 EJB を要求ビジネス・オブジェクトに変換し、応答ビジネス・オブジェクトを戻り値に変換します。

このデータ・ハンドラーは、Java API for XML Web Services (JAX-WS) 仕様および Java Architecture for XML Binding (JAXB) 仕様を使用して、データを SCA 指定の WSDL インターフェースからターゲット EJB Java インターフェースに (また、逆も同様に) マップします。

注: 現行のサポートは JAX-WS 2.1.1 および JAXB 2.1.3 仕様に制限されています。

EJB バインディング・レベルで指定されたデータ・ハンドラーは、要求、応答、障害、およびランタイム例外を処理するために使用されます。

注: 障害については、faultBindingType 構成プロパティーを指定することによって、障害ごとに特定のデータ・ハンドラーを指定できます。この設定は、EJB バインディング・レベルで指定された値をオーバーライドします。

EJB バインディングに WSDL インターフェースがある場合、デフォルトで JAX-WS データ・ハンドラーが使用されます。このデータ・ハンドラーは、JAX-WS 呼び出しを表す SOAP メッセージからデータ・オブジェクトへの変換には使用できません。

EJB インポート・バインディングは、データ・ハンドラーを使用して、データ・オブジェクトを Java オブジェクト配列 (Object[]) に変換します。アウトバウンド通信中は、以下の処理が実行されます。

1. EJB バインディングは、WSDL で指定された内容と一致するように、期待されるタイプ、期待されるエレメント、およびターゲット・メソッド名を BindingContext に設定します。
2. EJB バインディングが、データ変換の必要なデータ・オブジェクトのための変換メソッドを呼び出します。
3. データ・ハンドラーは、(メソッド内部での定義の順で) メソッドのパラメーターを表す Object[] を返します。
4. EJB バインディングは Object[] を使用して、ターゲット EJB インターフェース上のメソッドを呼び出します。

また、バインディングは EJB 呼び出しからの応答を処理するために Object[] も作成します。

- Object[] の最初のエレメントは、Java メソッド呼び出しからの戻り値です。
- subsequent 値は、メソッドの入力パラメーターを表します。

これは In/Out タイプおよび Out タイプのパラメーターをサポートするために必要です。

Out タイプのパラメーターの場合、値は応答データ・オブジェクト内で返される必要があります。

データ・ハンドラーは Object[] 内で検出した値を処理および変換して、データ・オブジェクトに応答を返します。

データ・ハンドラーは xs:AnyType、xs:AnySimpleType、および xs:Any のほか、他の XSD データ型もサポートします。xs:Any へのサポートを使用可能にするには、以下の例に示すように Java コードの JavaBean プロパティーで `@XmlAnyElement (lax=true)` を使用します。

```
public class TestType {
 private Object[] object;

 @XmlAnyElement (lax=true)
 public Object[] getObject() {
  return object;
 )

 public void setObject (Object[] object) {
  this.object=object;
 )
)
```

これによって、TestType のプロパティー・オブジェクトが xs:any フィールドになります。xs:any フィールド内で使用される Java クラスの値は、`@XmlAnyElement` アノテーションを持つ必要があります。例えば Address がオブジェクト配列の設定に使用される Java クラスである場合、Address クラスはアノテーション `@XmlRootElement` を持つ必要があります。

注: JAX-WS 仕様で定義された XSD タイプから Java タイプへのマッピングをカスタマイズするには、ビジネス・ニーズに合うように JAXB アノテーションを変更します。JAX-WS データ・ハンドラーは xs:any、xs:anyType、および xs:anySimpleType をサポートします。

JAX-WS データ・ハンドラーには以下の制約が適用されます。

- データ・ハンドラーには、ヘッダー属性 `@WebParam` アノテーションのサポートが含まれない。
- ビジネス・オブジェクト・スキーマ・ファイル (XSD ファイル) の名前空間には、Java パッケージ名からのデフォルト・マッピングが含まれない。package-info.java のアノテーション `@XMLSchema` も機能しません。名前空間で XSD を作成するための方法は、`@XmlType` および `@XmlRootElement` アノテーションを使用する方法だけです。`@XmlRootElement` は JavaBean タイプのグローバル・エレメントのターゲット名前空間を定義します。

- EJB インポート・ウィザードは、関連しないクラスについては XSD ファイルを作成しません。バージョン 2.0 では `@XmlSeeAlso` アノテーションがサポートされないため、子クラスが親クラスから直接参照されない場合、XSD は作成されません。この問題の解決方法は、そのような子クラスのために SchemaGen を実行することです。

  SchemaGen は、特定の JavaBean 用の XSD ファイルを作成するためのコマンド行ユーティリティー (場所は *WPS_Install_Home*/bin ディレクトリー) です。この解決方法を実現させるためには、これらの XSD をモジュールに手動でコピーする必要があります。

EJB 障害セレクター:

EJB 障害セレクターは、EJB 呼び出しの結果が、障害、ランタイム例外、または正常な応答のいずれかを判別します。

障害が検出された場合、EJB 障害セレクターがネイティブの障害名をバインディング・ランタイムに返すため、JAX-WS データ・ハンドラーが例外オブジェクトを障害ビジネス・オブジェクトに変換することができます。

正常な (障害のない) 応答の場合、EJB インポート・バインディングは、値を返すために、Java オブジェクト配列 (Object[]) をアセンブルします。

- Object[] の最初のエレメントは、Java メソッド呼び出しからの戻り値です。
- subsequent 値は、メソッドの入力パラメーターを表します。

これは In/Out タイプおよび Out タイプのパラメーターをサポートするために必要です。

例外シナリオの場合、バインディングは Object[] を組み立て、最初のエレメントはメソッドによってスローされた例外を表します。

障害セレクターは以下のどの値でも返すことができます。

表 13. 戻り値

タイプ	戻り値	説明
障害	`ResponseType.FAULT`	渡された Object[] に例外オブジェクトが含まれる場合に返されます。
実行時例外	`ResponseType.RUNTIME`	例外オブジェクトが、メソッド上で宣言されたどの例外タイプとも一致しない場合に返されます。
通常応答	`ResponseType.RESPONSE`	その他のすべてのケースで返されます。

障害セレクターが `ResponseType.FAULT` の値を返す場合、ネイティブの障害名を返します。このネイティブの障害名は、対応する WSDL 障害名をモデルから判別し、正しい障害データ・ハンドラーを呼び出すためにバインディングによって使用されます。

EJB 関数セレクター:

EJB バインディングは、インポート関数セレクター (アウトバウンド処理の場合) またはエクスポート関数セレクター (インバウンド処理の場合) を使用して、呼び出す EJB メソッドを判別します。

インポート関数セレクター

アウトバウンド処理の場合、インポート関数セレクターは、EJB インポートにワイヤリングされている SCA コンポーネントによって呼び出される操作の名前に基づいて、EJB メソッド・タイプを派生させます。関数セレクターは、WebSphere Integration Developer によって生成された JAX-WS アノテーション付きの Java クラスで @WebMethod アノテーションを探し、関連付けられているターゲット操作名を判別します。

- @WebMethod アノテーションが存在する場合、関数セレクターは @WebMethod アノテーションを使用して、WSDL メソッドのための正しい Java メソッド・マッピングを判別します。
- @WebMethod アノテーションがない場合、関数セレクターは、Java メソッド名は呼び出された操作名と同じであると想定します。

注: この関数セレクターは、EJB インポートの Java タイプ・インターフェースではなく、EJB インポートの WSDL タイプ・インターフェースでのみ有効です。

関数セレクターは EJB インターフェースのメソッドを表す java.lang.reflect.Method オブジェクトを返します。

関数セレクターは、ターゲット・メソッドからの応答を入れるために Java オブジェクト配列 (Object[]) を使用します。Object[] の最初のエレメントは、WSDL の名前を持つ Java メソッドで、Object[] の 2 番目のエレメントは入力ビジネス・オブジェクトです。

エクスポート関数セレクター

インバウンド処理の場合、エクスポート関数セレクターは、Java メソッドから呼び出されるターゲット・メソッドを派生させます。

エクスポート関数セレクターは、EJB クライアントによって呼び出される Java 操作名を、ターゲット・コンポーネントのインターフェースの操作名にマップします。メソッド名は、ストリングとして返され、ターゲット・コンポーネントのインターフェース・タイプに基づいて、SCA ランタイムによって解決されます。

EIS バインディング

エンタープライズ情報システム (EIS) のバインディングは、SCA コンポーネントと外部 EIS 間の接続を提供します。この通信を可能にするために使用する手段は、JCA 1.5 リソース・アダプターおよび Websphere Adapters をサポートする EIS エクスポートおよび EIS インポートです。

SCA コンポーネントでは、外部 EIS との間でのデータの転送が必要になる場合があります。このような接続を必要とする SCA モジュールを作成する場合は、SCA コンポーネントに加えて、特定の外部 EIS との通信のために EIS バインディングを使用したインポートまたはエクスポートを組み込む必要があります。

WebSphere Integration Developer のリソース・アダプターは、インポートまたはエクスポートのコンテキスト内で使用されます。インポートまたはエクスポートは外部サービス・ウィザードで開発しますが、これを開発するときにリソース・アダプターを組み込みます。アプリケーションが EIS システムのサービスを呼び出せるようにする EIS インポート、または EIS システムのアプリケーションが WebSphere Integration Developer で開発されたサービスを呼び出せるようにする EIS エクスポートは、特定のリソース・アダプターで作成されます。例えば、JD Edwards システムのサービスを呼び出すインポートを作成するには、JD Edwards アダプターを使用することになります。

外部サービス・ウィザードを使用すると、EIS バインディング情報が自動的に作成されます。また別のツールとして、バインディング情報を追加または変更するにはアセンブリー・エディターを使用できます。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。

EIS バインディングが含まれる SCA モジュールをサーバーにデプロイすると、管理コンソールを使用して、バインディングに関する情報を表示したり、バインディングを構成したりできます。

EIS バインディングの概要

EIS (エンタープライズ情報システム) バインディングを JCA リソース・アダプターと一緒に使用すると、エンタープライズ情報システム上のサービスにアクセスしたり、サービスを EIS で使用可能にすることができます。

以下は、Siebel システムと SAP システム間の接続情報を同期化する方法を説明する ContactSyncModule という名前の SCA モジュールの例です。

1. ContactSync という名前の SCA コンポーネントは、(Siebel Contact という名前の EIS アプリケーション・エクスポートを介して) Siebel への接続に対する変更を listen します。
2. ContactSync SCA コンポーネント自体は、SAP の接続情報を適宜更新するために、(EIS アプリケーション・インポートを介して) SAP アプリケーションを利用します。

接続情報の保管用に使用されるデータ構造は、Siebel システムと SAP システムでは異なるため、ContactSync SCA コンポーネントがマッピングを行う必要があります。

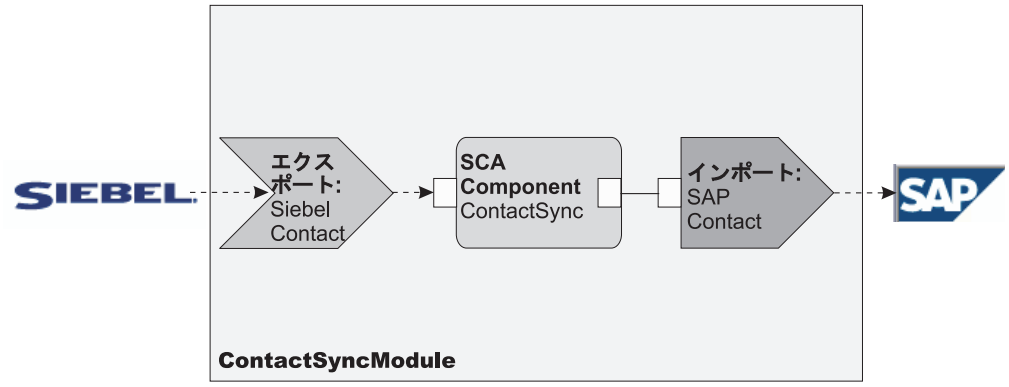


図 32. Siebel システムから SAP システムへのフロー

Siebel Contact エクスポートと SAP Contact インポートには、適切なリソース・アダプターが構成されます。

EIS バインディングの主な特徴

EIS インポートとは、Service Component Architecture (SCA) モジュール内のコンポーネントが SCA モジュール外部で定義された EIS アプリケーションを使用できるようにする SCA インポートです。EIS インポートは、SCA コンポーネントから外部 EIS へのデータ転送に使用されます。EIS エクスポートは、外部 EIS から SCA モジュールへのデータ転送に使用されます。

インポート

EIS インポートの役割は、SCA コンポーネントと外部 EIS システムの間のすき間を埋めることです。外部アプリケーションを EIS インポートとして取り扱うことができます。この場合、EIS インポートはデータを外部の EIS に送信し、必要に応じて応答データを受信します。

EIS インポートは、SCA コンポーネントがモジュール外部のアプリケーションを統一された形式で認識できるようにします。これによりコンポーネントは、SAP、Siebel、PeopleSoft などの外部 EIS と、一貫性のある SCA モデルを使用して通信できるようになります。

インポートのクライアント側には、EIS インポート・アプリケーションによって公開されるインターフェースがあります。ここには 1 つ以上のメソッドがあり、各メソッドがデータ・オブジェクトを引数および戻り値として受け入れます。実装側には、リソース・アダプターによって実装される共通クライアント・インターフェース (CCI) があります。

EIS インポートのランタイム実装は、クライアント側のインターフェースとこの CCI を接続します。インポートにより、インターフェース上のメソッドの呼び出しが CCI 上の呼び出しにマップされます。

バインディングは 3 つのレベルで作成されます (インターフェース・バインディングがそれに含まれるメソッド・バインディングを使用し、さらにそのメソッド・バインディングがデータ・バインディングを使用します)。

インターフェース・バインディングは、インポートのインターフェースを、アプリケーションを提供する EIS システムとの接続に関連付けます。これは、インターフェースによって表されるアプリケーション・セットが EIS の特定のインスタンスによって提供されていて、このインスタンスには接続を介してアクセスできるという事実を反映しています。バインディング・エレメントは、接続を作成するための十分な情報を指定したプロパティーを持ちます (このプロパティーは、javax.resource.spi.ManagedConnectionFactory インスタンスの一部です)。

メソッド・バインディングは、メソッドを、EIS システムとの特定の対話に関連付けます。JCA では、この対話の特徴は、javax.resource.cci.InteractionSpec インターフェース実装のプロパティー・セットによって記述されます。メソッド・バインディングの対話エレメントには、これらのプロパティーとともにクラスの名前が含まれています。これにより、対話を実行するための十分な情報を提供します。メソッド・バインディングは、データ・バインディングを使用して、インターフェース・メソッドの引数と結果の EIS 表現へのマッピングを記述します。

EIS インポートのランタイム・シナリオを以下に示します。

1. インポート・インターフェースのメソッドが、SCA プログラミング・モデルを使用して呼び出されます。
2. EIS インポートに送信される要求には、メソッド名と引数が指定されています。
3. インポートは最初にインターフェース・バインディングの実装を作成し、次にインポート・バインディングのデータを使用して ConnectionFactory を作成し、この 2 つを関連付けます。したがって、インポート により、インターフェース・バインディングの setConnectionFactory が呼び出されることになります。
4. 呼び出されたメソッドと一致するメソッド・バインディングの実装が作成されます。
5. javax.resource.cci.InteractionSpec インスタンスが作成され、データが取り込まれます。次に、データ・バインディングを使用して、リソース・アダプターが認識できるフォーマットにメソッド引数がバインドされます。
6. CCI インターフェースを使用して対話が実行されます。
7. 呼び出しが戻されたら、データ・バインディングによって呼び出しの結果が作成され、結果が呼び出し元に戻されます。

エクスポート

EIS エクスポートの役割は、SCA コンポーネントと外部 EIS 間のギャップを埋めることです。外部アプリケーションを EIS エクスポートとして取り扱うことができます。この場合、外部アプリケーションはそのデータを定期的な通知という形で送信します。EIS エクスポートは、EIS からの外部要求を listen するサブスクリプション・アプリケーションであると考えることができます。EIS エクスポートを使用する SCA コンポーネントは、EIS エクスポートをローカル・アプリケーションとして認識します。

EIS エクスポートは、SCA コンポーネントがモジュール外部のアプリケーションを統一された形式で認識できるようにします。これによりコンポーネントは、SAP、Siebel、PeopleSoft などの EIS と、一貫性のある SCA モデルを使用して通信できるようになります。

エクスポートは、EIS から要求を受け取るリスナー実装を特徴としています。リスナーは、リソース・アダプター固有のリスナー・インターフェースを実装します。また、エクスポートには、エクスポートを介して EIS に公開されるインターフェースを実装するコンポーネントも含まれます。

EIS エクスポートのランタイム実装は、インターフェースを実装するコンポーネントにリスナーを接続します。エクスポートにより、EIS 要求が、コンポーネント上の該当する操作の呼び出しにマップされます。バインディングは 3 つのレベルで作成されます (リスナー・バインディングがそれに含まれるネイティブ・メソッド・バインディングを使用し、さらにそのネイティブ・メソッド・バインディングがデータ・バインディングを使用します)。

リスナー・バインディングは、要求を受け取るリスナーを、エクスポートを介して公開されるコンポーネントに関連付けます。エクスポート定義には、コンポーネントの名前が含まれています。これにより、ランタイムがコンポーネントを検索し、コンポーネントに要求を転送します。

ネイティブ・メソッド・バインディングは、ネイティブ・メソッドまたはリスナーが受け取ったイベント・タイプを、エクスポートを介して公開されたコンポーネントによって実装される操作に関連付けます。リスナーで呼び出されるメソッドとイベント・タイプの間に関係はなく、すべてのイベントはリスナーの 1 つ以上のメソッドを使用して受信されます。ネイティブ・メソッド・バインディングは、エクスポートで定義された関数セレクターを使用してインバウンド・データからネイティブ・メソッド名を取り出し、データ・バインディングを使用して EIS のデータ・フォーマットをコンポーネントが認識できるフォーマットにバインドします。

EIS エクスポートのランタイム・シナリオを以下に示します。

1. EIS 要求は、リスナー実装のメソッドの呼び出しをトリガーします。
2. リスナーは エクスポートを検出して呼び出し、すべての呼び出し引数を渡します。
3. エクスポートは、リスナー・バインディングの実装を作成します。
4. エクスポートは、関数セレクターをインスタンス化してリスナー・バインディング上に設定します。
5. エクスポートは、ネイティブ・メソッド・バインディングを初期化してリスナー・バインディングに追加します。各ネイティブ・メソッド・バインディングについて、データ・バインディングも初期化されます。
6. エクスポートは、リスナー・バインディングを呼び出します。
7. リスナー・バインディングは、エクスポートされたコンポーネントを検出し、関数セレクターを使用してネイティブ・メソッド名を取得します。
8. この名前を使用して、ネイティブ・メソッド・バインディングが検出されます。その後、ネイティブ・メソッド・バインディングによってターゲット・コンポーネントが呼び出されます。

アダプターの対話スタイルでは、EIS エクスポート・バインディングで、ターゲット・コンポーネントを非同期方式 (デフォルト) または同期方式のどちらでも呼び出すことができます。

リソース・アダプター

外部サービス・ウィザードを使用してインポートやエクスポートを開発する際に、リソース・アダプターを組み込みます。WebSphere Integration Developer に付属するリソース・アダプターは、 CICS、IMS、JD Edwards、PeopleSoft、SAP、Siebel の各システムへのアクセスに使用され、開発とテストのみを目的としています。これらのリソース・アダプターを、アプリケーションの開発とテスト以外の目的では使用しないでください。

デプロイしたアプリケーションを実行するには、ライセンスが交付されたランタイム・アダプターが必要になります。ただし、サービスを構築する際に、このアダプターをサービスに組み込むことができます。アダプターのライセンス交付により、組み込まれたアダプターを、ライセンスが交付されたランタイム・アダプターとして使用することができます。これらのアダプターは、Java EE Connector Architecture (JCA 1.5) に準拠しています。オープン・スタンダードである JCA は、EIS 接続のための Java EE 標準です。 JCA は、管理されたフレームワークを提供します。つまり、サービスの品質 (QoS) がアプリケーション・サーバーによって提供され、それによってライフ・サイクル管理とセキュリティーがトランザクションに対して提供されます。また、これらのアダプターは、Enterprise Metadata Discovery 仕様にも準拠しています (IBM CICS ECI リソース・アダプターおよび IBM IMS Connector for Java を除く)。

従来のアダプターのセットである WebSphere Business Integration Adapters もウィザードでサポートされています。

Java EE リソース

EIS モジュール (EIS モジュール・パターンに準拠する SCA モジュール) は、Java EE プラットフォームにデプロイできます。

EIS モジュールを Java EE プラットフォームにデプロイすると、アプリケーションが EAR ファイルとしてパッケージされ、サーバーにデプロイされるので、アプリケーションを実行する準備が整います。すべての Java EE 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、実行の準備が整います。

JCA 対話仕様および接続仕様の動的プロパティー

EIS バインディングは、ペイロードに付随する明確に定義された子データ・オブジェクトを使用することによって指定された InteractionSpec および ConnectionSpec に対する入力を受け入れることができます。これにより、InteractionSpec を介したリソース・アダプターとの動的要求応答対話と、ConnectionSpec を介したコンポーネント認証が可能になります。

javax.cci.InteractionSpec は、リソース・アダプターとの対話要求の処理方法に関する情報を保持します。また、要求後に対話を行う方法に関する情報も保持します。対話によるこれらの両方向通信は、*会話*とも呼ばれます。

EIS バインディングはペイロードを必要とします。このペイロードは、`properties` という子データ・オブジェクトを格納するためのリソース・アダプターに対する引数になります。このプロパティー・データ・オブジェクトは、名前と値のペアを含

みます。この名前は、特定のフォーマットでの対話仕様プロパティーの名前になります。フォーマット設定の規則は以下のとおりです。

- 名前はプレフィックス「IS」で開始し、その後にプロパティー名が続かなければなりません。例えば、InteractionId という JavaBeans プロパティーを持つ対話仕様は、プロパティー名を ISInteractionId と指定します。
- 名前と値のペアは、対話仕様プロパティーの名前と単純型の値を表します。

この例ではインターフェースによって、操作の入力は「Account」データ・オブジェクトと指定されます。このインターフェースは、値が「xyz」である「workingSet」という動的 InteractionSpec プロパティーを送受信するために、EIS インポート・バインディング・アプリケーションを呼び出します。

サーバーのビジネス・グラフまたはビジネス・オブジェクトには下位の「properties」ビジネス・オブジェクトが含まれていて、このビジネス・オブジェクトによってペイロードを持つプロトコル固有データを送信できます。この「properties」ビジネス・オブジェクトは組み込みのものであり、ビジネス・オブジェクトを構成するときに XML スキーマで指定する必要はありません。単に作成するだけで使用できます。XML スキーマに基づいて独自のデータ型を定義している場合は、必要な名前と値のペアを含む「properties」エレメントを指定する必要があります。

```
BOFactory dataFactory = (BOFactory) ¥
serviceManager.locateService("com/ibm/websphere/bo/BOFactory");
 //Wrapper for doc-lit wrapped style interfaces,
 //skip to payload for non doc-lit
 DataObject docLitWrapper = dataFactory.createByElement /
 ("http://mytest/eis/Account", "AccountWrapper");
```

ペイロードを作成します。

```
DataObject account = docLitWrapper.createDataObject(0);
  DataObject accountInfo = account.createDataObject("AccountInfo");
  //Perform your setting up of payload


  //Construct properties data for dynamic interaction

  DataObject properties = account.createDataObject("properties");
```

名前「workingSet」に対して予想される値「xyz」を設定します。

```
properties.setString("ISworkingSet", "xyz");


  //Invoke the service with argument

  Service accountImport = (Service) ¥
  serviceManager.locateService("AccountOutbound");
  DataObject result = accountImport.invoke("createAccount", docLitWrapper);

  //Get returned property
  DataObject retProperties = result.getDataObject("properties");

  String workingset = retProperties.getString("ISworkingSet");
```

ConnectionSpec プロパティーは動的コンポーネント認証に使用できます。上記と同じ規則が適用されます。ただし、プロパティー名のプレフィックスは「CS」にする必要があります (「IS」ではありません)。ConnectionSpec プロパティーは両方向で

はありません。同じ「properties」データ・オブジェクトに IS プロパティーと CS プロパティーの両方を入れることができます。

ConnectionSpec プロパティーを使用するには、インポート・バインディングで指定する resAuth を「Application」に設定します。また、リソース・アダプターがコンポーネント許可をサポートする必要があります。詳しくは、「J2EE Connector Architecture Specification」の第 8 章を参照してください。

EIS バインディングを持つ外部クライアント

サーバーは、EIS バインディングを使用して、外部クライアントとの間でメッセージを送受信できます。

外部クライアント (Web ポータルや EIS など) は、サーバーの SCA モジュールにメッセージを送信するか、サーバー内からコンポーネントによって呼び出される必要があります。

クライアントは、動的起動インターフェース (DII) または Java インターフェースを使用して、その他のアプリケーションの場合と同様に EIS インポートを呼び出します。

1. 外部クライアントが ServiceManager のインスタンスを作成し、その参照名を使用して EIS インポートを検索します。検索の結果は、サービス・インターフェースの実装です。
2. クライアントが入力引数 (汎用データ・オブジェクト) を作成します。この引数は、データ・オブジェクト・スキーマを使用して動的に作成されます。このステップは、サービス・データ・オブジェクトの DataFactory インターフェースの実装を使用して行われます。
3. 外部クライアントが EIS を呼び出して必要な結果を取得します。

代わりに、クライアントは Java インターフェースを使用して EIS インポートを呼び出すこともできます。

1. クライアントが ServiceManager のインスタンスを作成し、その参照名を使用して EIS インポートを検索します。検索の結果は、EIS インポートの Java インターフェースです。
2. クライアントは入力引数および型付きデータ・オブジェクトを作成します。
3. クライアントが EIS を呼び出して必要な結果を取得します。

EIS エクスポート・インターフェースは、外部の EIS アプリケーションで使用可能なエクスポート済み SCA コンポーネントのインターフェースを定義します。このインターフェースは、(SAP や PeopleSoft などの) 外部アプリケーションが EIS エクスポート・アプリケーション・ランタイムの実装を介して呼び出すインターフェースと考えることができます。

エクスポートは EISExportBinding を使用してエクスポート済みサービスを外部の EIS アプリケーションにバインドします。これにより、EIS サービス要求を listen するよう、SCA モジュールに含まれるアプリケーションをサブスクライブできます。EIS エクスポート・バインディングは、リソース・アダプターによって (Java

EE コネクター・アーキテクチャー・インターフェースを使用して) 認識されるインバウンド・イベントの定義と SCA 操作の呼び出しの間のマッピングを指定します。

EISExportBinding では、外部の EIS サービスが Java EE コネクター・アーキテクチャー 1.5 のインバウンド規約に基づいている必要があります。EISExportBinding では、データ・ハンドラーまたはデータ・バインディングをバインディング・レベルまたはメソッド・レベルで指定する必要があります。

JMS バインディング

Java Message Service (JMS) プロバイダーにより、Java Messaging Service API およびプログラミング・モデルに基づいたメッセージングが可能になります。JMS プロバイダーは、JMS 宛先への接続を作成しメッセージを送受信するための Java EE 接続ファクトリーを提供します。

以下の 3 つの JMS バインディングが提供されています。

- JMS JCA 1.5 に準拠したサービス統合バス (SIB) プロバイダー・バインディング (*JMS バインディング*)
- JMS 1.1 に準拠した非 JCA の汎用 JMS バインディング (*汎用 JMS バインディング*)
- WebSphere MQ JMS バインディング。WebSphere MQ に対し JMS プロバイダー・サポートを提供し、Java EE アプリケーションとの相互運用性を実現します。(*WebSphere MQ JMS バインディング*)

JMS エクスポートおよびインポート・バインディングにより、Service Component Architecture (SCA) モジュールは、外部 JMS システムに対する呼び出しを実行したり、外部 JMS システムからメッセージを受信したりできます。

また、WebSphere MQ バインディング (*WebSphere MQ バインディング*) もサポートされています。このバインディングにより、ネイティブ MQ ユーザーは、着信メッセージと発信メッセージの任意のフォーマットを処理できます (WebSphere MQ が必要です)。

JMS インポートおよびエクスポート・バインディングは、WebSphere Application Server の一部である JCA 1.5 ベースの SIB JMS プロバイダーを使用して JMS アプリケーションとの統合を行います。その他の JCA 1.5 ベースの JMS リソース・アダプターはサポートされません。

JMS バインディングの概要

JMS バインディングは、Service Component Architecture (SCA) 環境と JMS システムの間の接続を提供します。

JMS バインディング

JMS インポート・バインディングおよび JMS エクスポート・バインディングの主なコンポーネントは、以下のとおりです。

- リソース・アダプター: SCA モジュールと外部 JMS システムの間の管理された両方向接続を使用可能にします。

- 接続: クライアントとプロバイダー・アプリケーションの間の仮想接続をカプセル化します。
- 宛先: 生成するメッセージの宛先またはコンシュームするメッセージの送信元を指定するためにクライアントが使用します。
- 認証データ: バインディングへのアクセスを保護するために使用します。

JMS インポート・バインディング

JMS インポート・バインディングでは、SCA モジュール内で使用する外部 JMS アプリケーションをインポートできます。JMS インポート・バインディングにより、SCA モジュール内のコンポーネントは、外部 JMS アプリケーションが提供するサービスと通信できるようになります。

JMS 宛先に関連する JMS プロバイダーへの接続を作成するには、JMS 接続ファクトリーを使用します。デフォルト・メッセージング・プロバイダーの JMS 接続ファクトリーを管理するには、接続ファクトリーの管理オブジェクトを使用します。

外部 JMS システムとの対話では、要求を送信し、応答を受信するための宛先が使用されます。

JMS インポート・バインディングでは、呼び出されている操作のタイプに応じた以下の 2 種類の使用シナリオがサポートされています。

- 片方向: JMS インポートは、インポート・バインディングに構成された送信宛先にメッセージを送信します。JMS ヘッダーの replyTo フィールドには何も設定しません。
- 両方向 (要求/応答): JMS インポートは、送信宛先にメッセージを送信し、その後 SCA コンポーネントから受信した応答を維持します。

  (WebSphere Integration Developer の「**応答相関スキーム**」フィールドを使用して) インポート・バインディングを構成して、要求メッセージ ID (デフォルト) または要求メッセージ相関 ID から応答メッセージ相関 ID がコピーされているようにすることができます。インポート・バインディングを構成して、応答と要求を相関させるために一時動的応答宛先を使用することもできます。一時宛先は要求ごとに作成され、インポートはこの宛先を使用して応答を受信します。

  receive 宛先がアウトバウンド・メッセージの replyTo ヘッダー・プロパティーに設定されます。受信宛先で listen するためのメッセージ・リスナーをデプロイします。応答を受信すると、メッセージ・リスナーは応答をコンポーネントに返します。

片方向と両方向のいずれのシナリオを使用する場合も、動的および静的ヘッダー・プロパティーを指定できます。静的プロパティーは、JMS インポート・メソッド・バインディングから設定できます。これらのプロパティーの一部は、SCA JMS ランタイムで特別な意味を持ちます。

重要な点として、JMS は非同期バインディングであることに注意してください。呼び出し側コンポーネントが JMS インポートを同期的に呼び出すと (両方向操作の場合)、呼び出し側コンポーネントは、JMS サービスからの応答が返されるまでブロックされます。

図 33 は、インポートがどのように外部サービスにリンクされているのかを示しています。

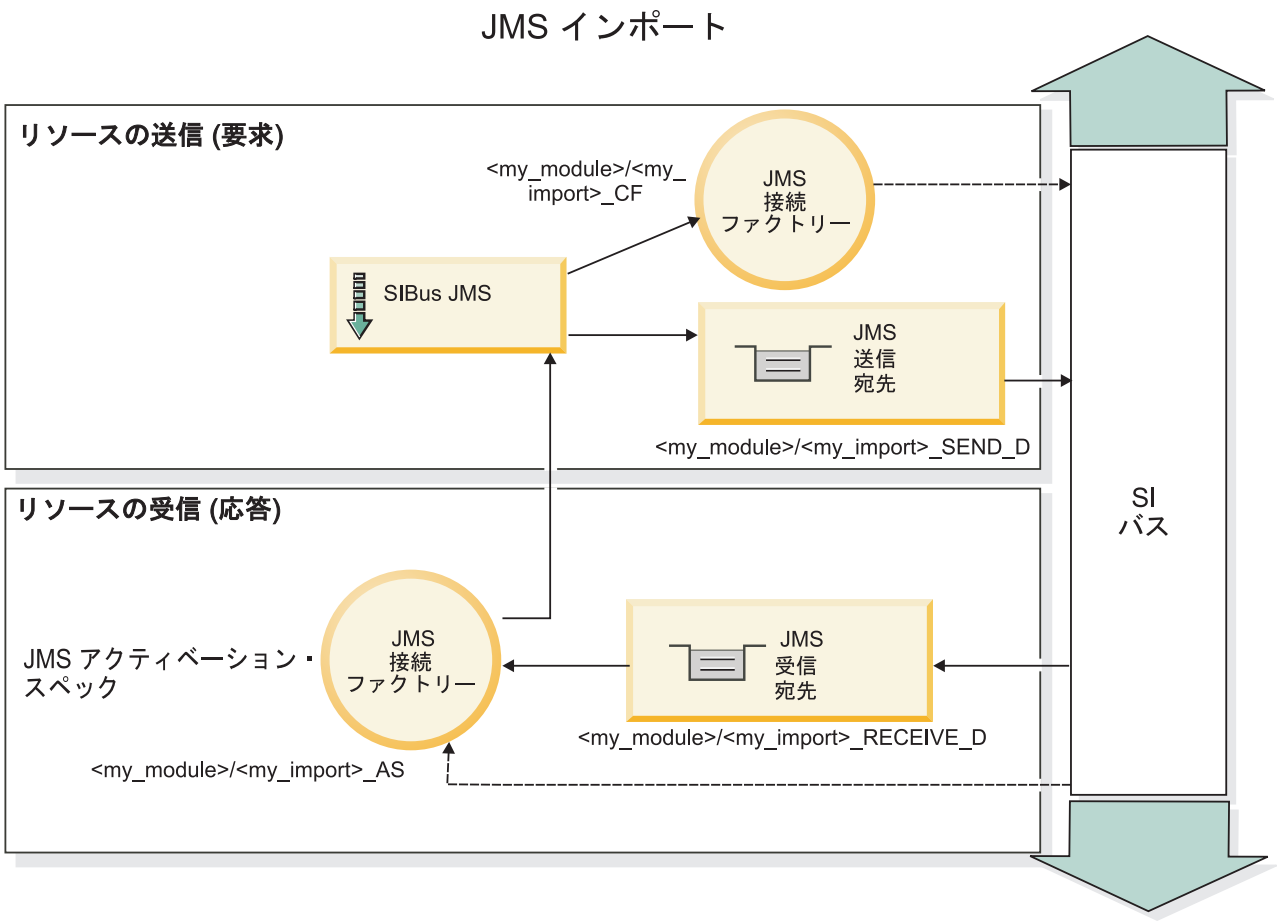


図 33. JMS インポート・バインディングのリソース

JMS エクスポート・バインディング

JMS エクスポート・バインディングは、SCA モジュールが外部 JMS アプリケーションにサービスを提供する手段を提供します。

JMS エクスポートの接続部分は、構成可能なアクティベーション・スペックです。

JMS エクスポートには、send 宛先と receive 宛先があります。

- receive 宛先は、送信先コンポーネントに対する着信メッセージを格納する宛先です。
- send 宛先は、応答を送信する宛先です。ただし、着信メッセージで replyTo ヘッダー・プロパティーを使用してこの宛先をオーバーライドしている場合は、その宛先が優先されます。

エクスポート・バインディングで指定された receive 宛先に着信する要求を listen するため、メッセージ・リスナーがデプロイされます。send フィールドで指定された宛先は、呼び出されたコンポーネントが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。着信メッセージの replyTo フィールドで指定された宛先は、send で指定された宛先をオーバーライドします。

図 34 は、外部の要求側がどのようにエクスポートにリンクされているのかを示しています。

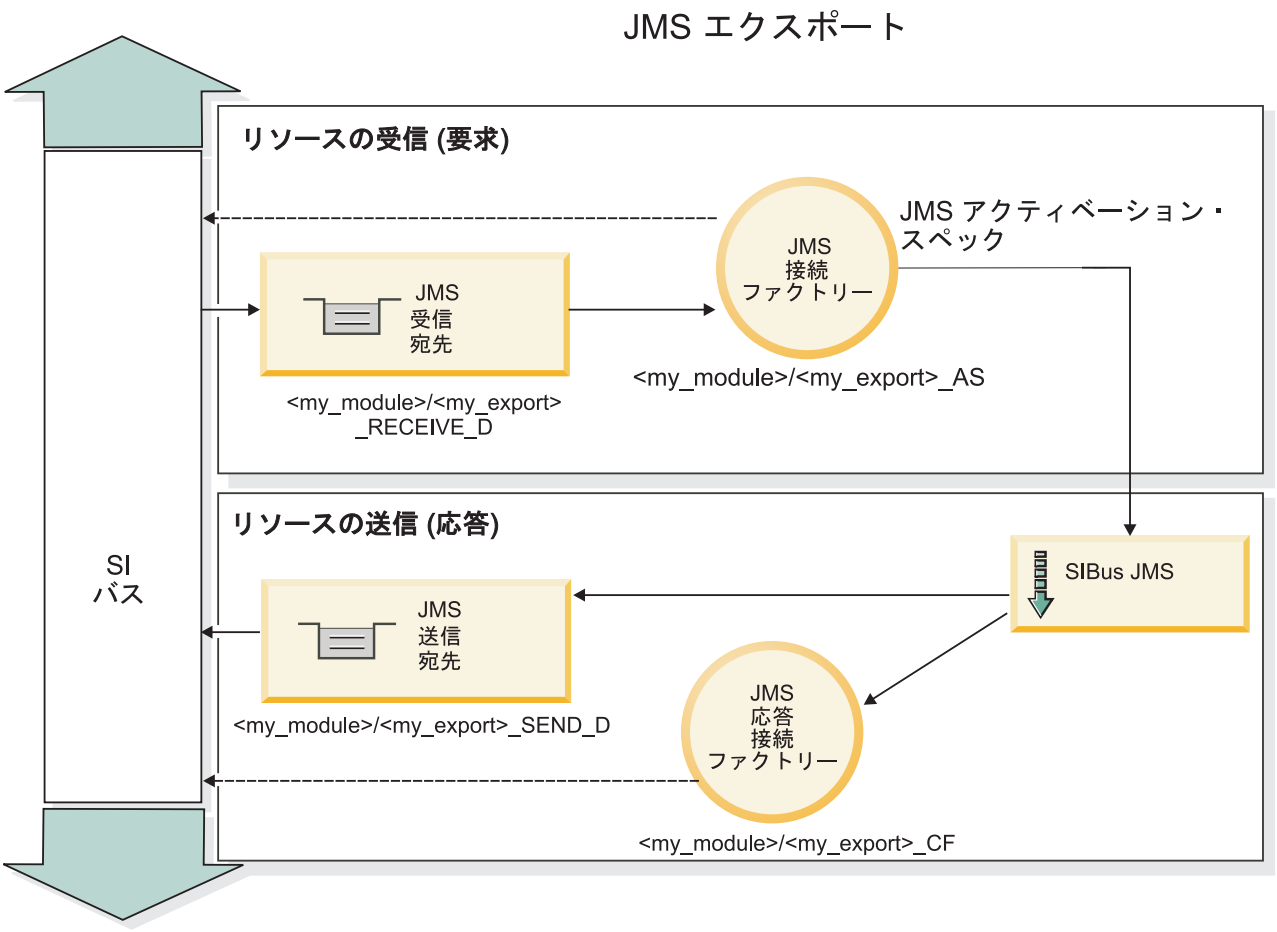


図 34. JMS エクスポート・バインディングのリソース

JMS 統合とリソース・アダプター

Java Message Service (JMS) は、使用可能な JMS JCA 1.5 ベースのリソース・アダプターを介して統合を提供します。JMS 統合のための完全なサポートが、サービス統合バス (SIB) JMS リソース・アダプターに対して用意されています。

JCA 1.5 リソース・アダプター用の JMS プロバイダーは、外部 JCA 1.5 準拠 JMS システムと統合する場合に使用します。JCA 1.5 に準拠した外部サービスは、SIB JMS リソース・アダプターを使用してメッセージを送受信でき、Service Component Architecture (SCA) コンポーネントと統合します。

他のプロバイダー固有の JCA 1.5 リソース・アダプターの使用はサポートされません。

JMS モジュールは Java SE 環境にデプロイできません。それらのモジュールは Java EE 環境にのみデプロイできます。

JMS バインディングの主な特徴

JMS のインポート・バインディングおよびエクスポート・バインディングの主な特徴は、ヘッダーと作成される Java EE リソースです。

特殊ヘッダー

特殊ヘッダーのプロパティーは、JMS インポートおよびエクスポートで使用され、ターゲットに対してメッセージの処理方法を指示します。

例えば、TargetFunctionName がネイティブ・メソッドから操作メソッドにマップされます。

Java EE リソース

JMS インポートおよびエクスポートを Java EE 環境にデプロイすると、さまざまな Java EE リソースが作成されます。

ConnectionFactory
クライアントが JMS プロバイダーとの接続を作成するために使用します。

ActivationSpec
インポートでは、要求に対する応答を受信するときに使用されます。エクスポートでは、メッセージング・システムとの対話でメッセージ・リスナーを表すメッセージ・エンドポイントを構成するときに使用されます。

宛先

- 送信宛先: インポートの場合は、要求または出力メッセージが送信される宛先になります。エクスポートの場合は、応答メッセージが送信される宛先になります。ただし、着信メッセージの JMSReplyTo ヘッダー・フィールドにより置き換えられた場合は、その宛先が優先されます。
- 受信宛先: 着信メッセージが格納される宛先です。インポートの場合は応答になり、エクスポートの場合は要求になります。
- コールバック宛先: 相関情報の保管に使用される SCA JMS システム宛先です。この宛先に対して、読み取りまたは書き込みを行わないでください。

インストール・タスクでは、ConnectionFactory および 3 つの宛先を作成します。また、ActivationSpec を作成して、ランタイム・メッセージ・リスナーが受信宛先で応答を listen できるようにします。これらのリソースのプロパティーは、インポート・ファイルまたはエクスポート・ファイルに指定されます。

JMS ヘッダー

JMS メッセージには、2 つのタイプのヘッダーが含まれます。1 つは JMS システム・ヘッダー、もう 1 つは複数の JMS プロパティーです。メディエーション・モジュールでは、いずれのタイプのヘッダーにも、サービス・メッセージ・オブジェクト (SMO) 内、または ContextService API を使用してアクセスできます。

JMS システム・ヘッダー

SMO では、JMS システム・ヘッダーは JMSHeader エレメントによって表されます。このエレメントには、JMS ヘッダーに通常あるすべてのフィールドが含まれます。これらのフィールドはメディエーション (または ContextService) で変更できま

すが、SMO に設定された一部の JMS システム・ヘッダー・フィールドは、システムまたは静的な値によってオーバーライドされるため、アウトバウンド JMS メッセージでは伝搬されません。

メディエーション (または ContextService) で更新可能な JMS システム・ヘッダーのキー・フィールドには以下があります。

- **JMSType** および **JMSCorrelationID** - 特定の事前定義メッセージ・ヘッダー・プロパティーの値
- **JMSDeliveryMode** - 送達モードの値 (persistent または nonpersistent。デフォルトは persistent)
- **JMSPriority** - 優先度の値 (0 から 9。デフォルトは JMS_Default_Priority)

JMS プロパティー

JMS プロパティーは、SMO ではプロパティー・リスト内のエントリーとして表されます。プロパティーは、メディエーション内で、または ContextService API を使用して追加、更新、または削除できます。

プロパティーは、JMS バインディングに静的に設定することもできます。静的に設定されたプロパティーは、動的に設定される (同じ名前の) 設定をオーバーライドします。

他のバインディング (例えば HTTP バインディング) から伝搬されるユーザー・プロパティーは、JMS バインディングでは JMS プロパティーとして出力されます。

ヘッダー伝搬の設定

JMS システム・ヘッダーおよびプロパティーの、インバウンド JMS メッセージからダウンストリームのコンポーネントへの伝搬、またはアップストリームのコンポーネントからアウトバウンド JMS メッセージへの伝搬は、バインディングのプロトコル・ヘッダー伝搬フラグで制御できます。

プロトコル・ヘッダー伝搬を設定すると、以下のリストに説明するようにヘッダー情報をメッセージまたはターゲット・コンポーネントに流すことができます。

- JMS エクスポート要求

  メッセージ内で受信した JMS ヘッダーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。メッセージ内で受信した JMS プロパティーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。
- JMS エクスポート応答

  コンテキスト・サービスに設定されたすべての JMS ヘッダー・フィールドは、JMS エクスポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。コンテキスト・サービスに設定されたすべてプロパティーは、JMS エクスポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。
- JMS インポート要求

コンテキスト・サービスに設定されたすべての JMS ヘッダー・フィールドは、JMS インポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。コンテキスト・サービスに設定されたすべてプロパティーは、JMS インポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。

- JMS インポート応答

  メッセージ内で受信した JMS ヘッダーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。メッセージ内で受信した JMS プロパティーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。

JMS 一時動的応答宛先の相関スキーム

一時動的応答宛先の相関スキームによって、送信された要求ごとに固有の動的キューまたはトピックが作成されます。

インポートに指定されている静的応答宛先は、一時動的宛先キューまたはトピックの性質を派生させるために使用されます。これは要求の **ReplyTo** フィールドで設定され、JMS インポートはその宛先で応答を listen します。応答が受信されると、その応答は非同期処理のために静的応答宛先に再キューイングされます。応答の **CorrelationID** フィールドは使用されないため、設定する必要はありません。

トランザクションの問題

一時動的宛先を使用する場合、応答は送信された応答と同じスレッドでコンシュームされる必要があります。要求はグローバル・トランザクションの外側で送信される必要があり、バックエンド・サービスによって受信される前、および応答が返される前にコミットされる必要があります。

パーシスタンス

一時動的キューは存続期間の短いエンティティーであるため、静的キューまたはトピックに関連付けられているのと同じレベルの永続性は保障されません。一時動的キューまたはトピックは、サーバーの再始動後も存続することはありません。メッセージも同様です。メッセージが静的応答宛先に再キューイングされると、メッセージに定義されている永続性を保持するようになります。

タイムアウト

インポートは、一定の時間、一時動的応答宛先で応答の受信を待機します。この時間間隔は、SCA 応答有効期限修飾子から取得されますが、これが設定されていない場合はデフォルトで 60 秒になります。待機時間を超過すると、インポートは `ServiceTimeoutRuntimeException` をスローします。

外部クライアント

サーバーは、JMS バインディングを使用して、外部クライアントとの間でメッセージを送受信できます。

外部クライアント (Web ポータルやエンタープライズ情報システムなど) は、サーバーの SCA モジュールにメッセージを送信できます。また、サーバー内からコンポーネントによって外部クライアントを呼び出すこともできます。

JMS エクスポート・コンポーネントは、エクスポート・バインディングで指定された受信宛先に着信する要求を listen するためのメッセージ・リスナーをデプロイします。送信フィールドで指定された宛先は、呼び出されたアプリケーションが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。したがって、外部クライアントは、エクスポート・バインディングを使用してアプリケーションを呼び出すことができます。

JMS インポートは外部クライアントにバインドし、外部クライアントにメッセージを配信できます。このメッセージは、外部クライアントからの応答を要求してもしなくても構いません。

外部クライアントの使用:

外部クライアント (サーバー外部のクライアント) とサーバーにインストールされたアプリケーションの対話が必要になる場合があります。

このタスクについて

外部クライアントとサーバー上のアプリケーションが対話する場合の、非常に簡単なシナリオを以下に示します。以下の図は、標準的な単純なシナリオを表しています。

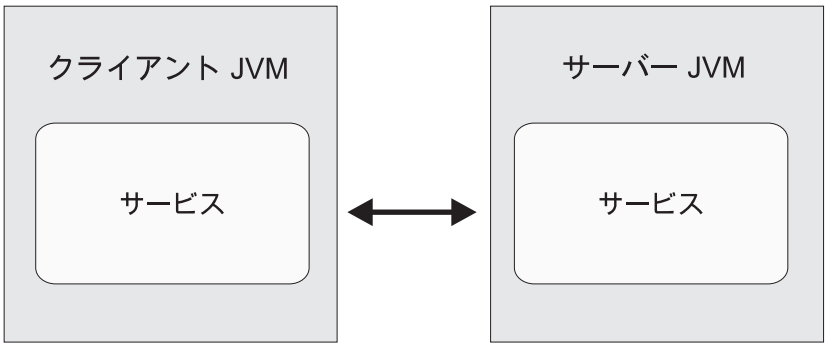


図 35. 単純なユース・ケース・シナリオ: 外部クライアントとサーバー・アプリケーションの対話

SCA アプリケーションには、JMS バインディングによるエクスポートが含まれています。これにより、外部クライアントからアプリケーションを使用できるようになります。

外部クライアントがサーバーとは別個の Java 仮想マシン (JVM) 内にある場合、JMS エクスポートとの接続および対話を作成するためにいくつかのステップを実行する必要があります。クライアントは、正しい値が指定された InitialContext を取得

し、JNDI を使用してリソースを検索します。次にクライアントは、JMS 1.1 仕様のクライアントを使用して宛先にアクセスし、宛先の送信メッセージと受信メッセージにアクセスします。

ランタイムにより自動作成されるリソースのデフォルトの JNDI 名は、このセクションの構成に関するトピックにリストされています。ただし、事前に作成済みのリソースがある場合は、その JNDI 名を使用します。

手順

1. メッセージを送信するため、JMS の宛先と接続ファクトリーを構成します。
2. JNDI コンテキスト、SIB リソース・アダプターのポート、およびメッセージング・ブートストラッピング・ポートが正しいことを確認します。

   サーバーはいくつかのデフォルト・ポートを使用しますが、それより多くのサーバーがそのシステムにインストールされている場合は、他のサーバー・インスタンスとの競合を避けるために、インストール時に代わりのポートが作成されます。管理コンソールを使用して、サーバーが使用しているポートを調べることができます。「**サーバー**」 → 「**アプリケーション・サーバー**」 → ***your_server_name*** → 「**構成**」を選択して、「**通信 (Communication)**」の下で「**ポート**」をクリックします。これで、使用するポートを編集できます。
3. クライアントは、正しい値が指定された初期コンテキストを取得して、JNDI を使用してリソースを検索します。
4. クライアントは、JMS 1.1 仕様に基づいて宛先にアクセスし、宛先の送信メッセージと受信メッセージにアクセスします。

JMS バインディングのトラブルシューティング

JMS バインディングで発生した問題を診断し、修正できます。

実装例外

JMS のインポートおよびエクスポート実装は、さまざまなエラー状態に応じて、以下の 2 種類の例外のうちのいずれかを戻すことがあります。

- サービス・ビジネス例外: サービス・ビジネス・インターフェース (WSDL ポート・タイプ) で指定された障害が発生した場合に、この例外が戻されます。
- サービス・ランタイム例外: その他のすべてのケースで生成されます。ほとんどの場合、cause 例外には元の例外 (JMSException) が含まれます。

  例えばインポートの場合、要求メッセージごとに 1 つの応答メッセージだけが戻されることを前提としています。そのため、複数の応答を受信した場合や遅延応答 (SCA の応答有効期限が切れた応答) を受信した場合は、サービス・ランタイム例外がスローされます。この場合、トランザクションはロールバックされ、応答メッセージはキューからバックアウトされるか、または Failed Event Manager によって処理されます。

主な障害状態

JMS バインディングの主な障害状態は、トランザクションの意味構造、JMS プロバイダー構成、またはその他のコンポーネントの既存動作への参照に基づいて判別されます。主な障害状態は以下のとおりです。

- JMS プロバイダーまたは宛先に接続できない。

  JMS プロバイダーに接続してメッセージを受信できない場合は、メッセージ・リスナーを開始することができません。この状態は、WebSphere Application Server ログに記録されます。正常に取得されるまで (または期限切れとなるまで)、永続メッセージは宛先に残ります。

  JMS プロバイダーに接続してアウトバウンド・メッセージを送信できない場合、送信操作を制御するトランザクションがロールバックされます。
- インバウンド・メッセージを解析できないか、アウトバウンド・メッセージを構成できない。

  データ・バインディングまたはデータ・ハンドラーが失敗すると、作業を制御するトランザクションがロールバックされます。
- アウトバウンド・メッセージを送信できない。

  メッセージを送信できないと、関連するトランザクションがロールバックされます。
- 複数のメッセージまたは予期しない遅延応答メッセージが返される。

  インポートの場合、要求メッセージごとに 1 つの応答メッセージだけが戻されることを前提としています。また、応答を受信できる有効期間は、要求の SCA 応答有効期限修飾子によって決まります。応答を受信したとき、または有効期間を超えたときに、相関レコードが削除されます。予期しない応答メッセージを受信した場合や遅れて応答を受信した場合は、サービス・ランタイム例外がスローされます。
- 一時動的応答宛先相関スキームを使用したとき、遅延応答によりサービス・タイムアウト・ランタイム例外が引き起こされる。

  JMS インポートは、SCA 応答有効期限修飾子によって決められた期間後にタイムアウトします。期間が設定されていない場合は、デフォルトで 60 秒になります。

Failed Event Manager に表示されない JMS ベースの SCA メッセージ

JMS の対話の失敗によって SCA メッセージが発生した場合は、Failed Event Manager でこのメッセージを見つけることになります。Failed Event Manager にこのようなメッセージが表示されない場合は、JMS 宛先の基礎となる SIB 宛先の最大配信失敗回数の値に 1 よりも大きい値が設定されているかどうかを確認してください。この値を 2 以上に設定すると、JMS バインディングに対して SCA を呼び出す際に、Failed Event Manager と対話することができます。

例外の処理

バインディングがどのように構成されているかによって、データ・ハンドラーまたはデータ・バインディングによる例外の処理方法が決まります。また、メディエーション・フローの性質により、そのような例外がスローされたときのシステムの振る舞いが決まります。

データ・ハンドラーまたはデータ・バインディングがバインディングによって呼び出されるときには、さまざまな問題が発生する可能性があります。例えば、データ・ハンドラーが、ペイロードが破損しているメッセージを受信したり、誤った形式のメッセージを読み取るなどです。

バインディングによるそのような例外の処理方法は、データ・ハンドラーおよびデータ・バインディングの実装方法によって決まります。推奨される振る舞いは、`DataBindingException` をスローするようにデータ・バインディングを設計することです。

`DataBindingException` などのランタイム例外がスローされた場合:

- メディエーション・フローがトランザクションとして構成されている場合、デフォルトでは、JMS メッセージは手動でやり直し、または削除できるように Failed Event Manager に保管されます。

  注: バインディングのリカバリー・モードを変更することによって、メッセージが Failed Event Manager に保管される代わりに、ロールバックされるようにできます。
- メディエーション・フローがトランザクションではない場合、例外はログに記録され、メッセージは失われます。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、すべてのデータ・ハンドラー例外はデータ・バインディング例外内にラップされます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

汎用 JMS バインディング

汎用 JMS バインディングにより、サード・パーティーの JMS 1.1 準拠プロバイダーに対する接続が提供されます。汎用 JMS バインディングの操作は、JMS バインディングの操作と似ています。

JMS バインディングを介して提供されるサービスにより、Service Component Architecture (SCA) モジュールは、外部システムに対する呼び出しを実行したり、外部システムからメッセージを受信したりできます。このシステムとして、外部 JMS システムを使用できます。

汎用 JMS バインディングは、JCA 1.5 非準拠 JMS プロバイダー (JMS プロバイダーが JMS 1.1 をサポートし、オプションの JMS Application Server Facility を実装している場合) との統合を実現します。汎用 JMS バインディングは、JCA 1.5 をサポートしないが JMS 1.1 仕様の Application Server Facility をサポートする JMS プロバイダーをサポートしています (これには、Oracle AQ、TIBCO、SonicMQ、WebMethods、BEA WebLogic、WebSphere MQ などが含まれます)。 WebSphere の組み込み JMS プロバイダー (SIBJMS) である JCA 1.5 JMS プロバイダーは、このバインディングではサポートされていません。このプロバイダーを使用する場合は、 82 ページの『JMS バインディング』を使用してください。

SCA 環境内で JCA 1.5 非準拠の JMS ベース・システムと統合するときに、この汎用バインディングを使用します。これにより、ターゲットの外部アプリケーションがメッセージを送受信でき、SCA コンポーネントと統合できます。

汎用 JMS バインディングの概要

汎用 JMS バインディングは、Service Component Architecture (SCA) 環境と JMS システム (JMS 1.1 に準拠し、オプションの JMS Application Server Facility を実装している場合) の間の接続を提供する非 JCA JMS バインディングです。

汎用 JMS バインディング

汎用 JMS インポートおよびエクスポート・バインディングの主な側面は以下のとおりです。

- リスナー・ポート: 非 JCA ベースの JMS プロバイダーがメッセージを受信し、それらをメッセージ・ドリブン Bean (MDB) にディスパッチできるようにします。
- 接続: クライアントとプロバイダー・アプリケーションの間の仮想接続をカプセル化します。
- 宛先: 生成するメッセージの宛先またはコンシュームするメッセージの送信元を指定するためにクライアントが使用します。
- 認証データ: バインディングへのアクセスを保護するために使用します。

汎用 JMS インポート・バインディング

汎用 JMS インポート・バインディングにより、SCA モジュール内のコンポーネントは、外部の JCA 1.5 非準拠の JMS プロバイダーが提供するサービスと通信できるようになります。

JMS インポートの接続部分は、接続ファクトリーです。接続ファクトリーとは、クライアントがプロバイダーへの接続を作成するために使用するオブジェクトであり、管理者によって定義された一連の接続構成パラメーターをカプセル化します。各接続ファクトリーは、ConnectionFactory、QueueConnectionFactory、または TopicConnectionFactory インターフェースのインスタンスです。

外部 JMS システムとの対話では、要求を送信し、応答を受信するための宛先が使用されます。

汎用 JMS インポート・バインディングでは、呼び出されている操作のタイプに応じた以下の 2 種類の使用シナリオがサポートされています。

- 片方向: 汎用 JMS インポートは、インポート・バインディングに構成された送信宛先にメッセージを送信します。JMS ヘッダーの replyTo フィールドには何も送信しません。
- 両方向 (要求/応答): 汎用 JMS インポートは、送信宛先にメッセージを送信し、その後 SCA コンポーネントから受信した応答を維持します。

  receive 宛先がアウトバウンド・メッセージの replyTo ヘッダー・プロパティーに設定されます。受信宛先で listen するためのメッセージ駆動型 Bean (MDB) をデプロイします。応答を受信すると、MDB は応答をコンポーネントに返します。

インポート・バインディングは、要求メッセージ ID (デフォルト) または要求メッセージ相関 ID から応答メッセージ相関 ID がコピーされていることを期待するように (WebSphere Integration Developer の「**応答相関スキーム**」フィールドを使用して) 構成することができます。

片方向と両方向のいずれのシナリオを使用する場合も、動的および静的ヘッダー・プロパティーを指定できます。静的プロパティーは汎用 JMS インポート・メソッド・バインディングから設定できます。これらのプロパティーのなかには、SCA JMS ランタイムにとって特殊な意味を持つものがあります。

重要な点として、汎用 JMS は非同期バインディングであることに注意してください。呼び出し側コンポーネントが汎用 JMS インポートを同期的に呼び出すと (両方向操作の場合)、呼び出し側コンポーネントは、JMS サービスからの応答が返されるまでブロックされます。

図 36 は、インポートがどのように外部サービスにリンクされているのかを示しています。

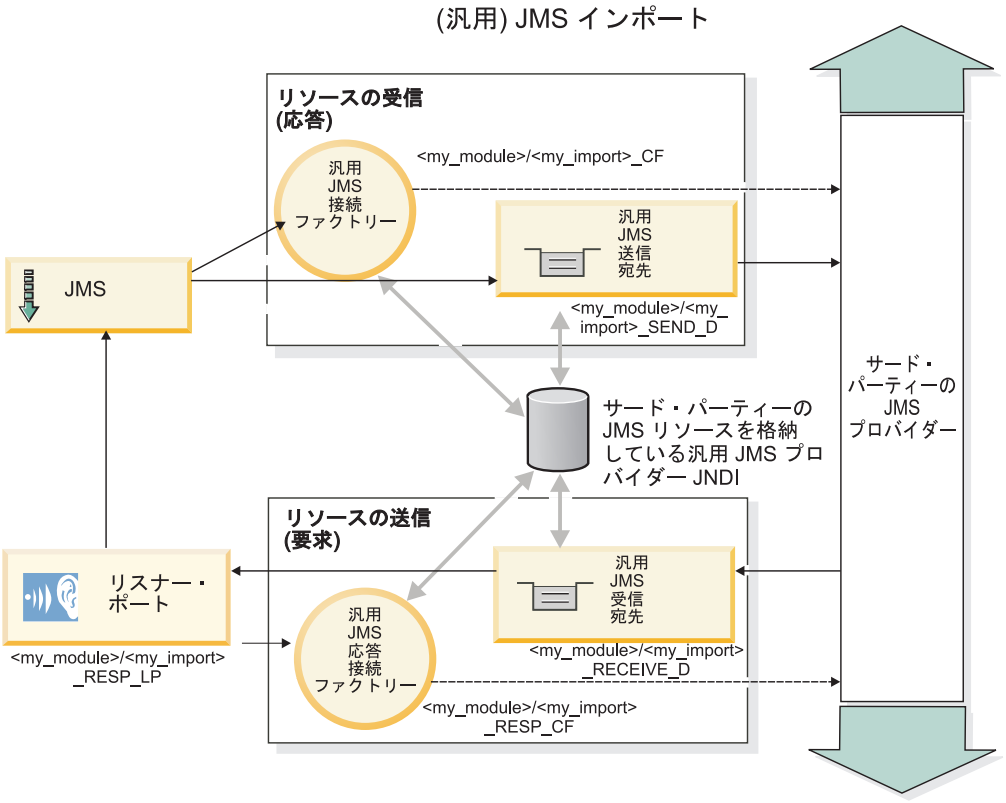


図 36. 汎用 JMS インポート・バインディング・リソース

汎用 JMS エクスポート・バインディング

汎用 JMS エクスポート・バインディングは、SCA モジュールが外部 JMS アプリケーションにサービスを提供する手段を提供します。

JMS エクスポートの接続部分は、ConnectionFactory および ListenerPort から構成されます。

汎用 JMS エクスポートには、send 宛先と receive 宛先があります。

- receive 宛先は、送信先コンポーネントに対する着信メッセージを格納する宛先です。
- send 宛先は、応答を送信する宛先です。ただし、着信メッセージで replyTo ヘッダー・プロパティーを使用してこの宛先をオーバーライドしている場合は、その宛先が優先されます。

エクスポート・バインディングで指定された receive 宛先に着信する要求を listen するため、MDB がデプロイされます。

- send フィールドで指定された宛先は、呼び出されたコンポーネントが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。
- 着信メッセージの replyTo フィールドで指定された宛先は、send フィールドで指定された宛先をオーバーライドします。
- 要求/応答シナリオの場合、応答が要求 message ID を応答メッセージの correlation ID フィールドにコピーする(デフォルト) ことを期待するように、(WebSphere Integration Developer の「**応答相関スキーマ**」フィールドを使用して) インポート・バインディングを構成できます。または、応答が要求の correlation ID を応答メッセージの correlation ID フィールドにコピーすることもできます。

図 37 は、外部の要求側がどのようにエクスポートにリンクされているのかを示しています。

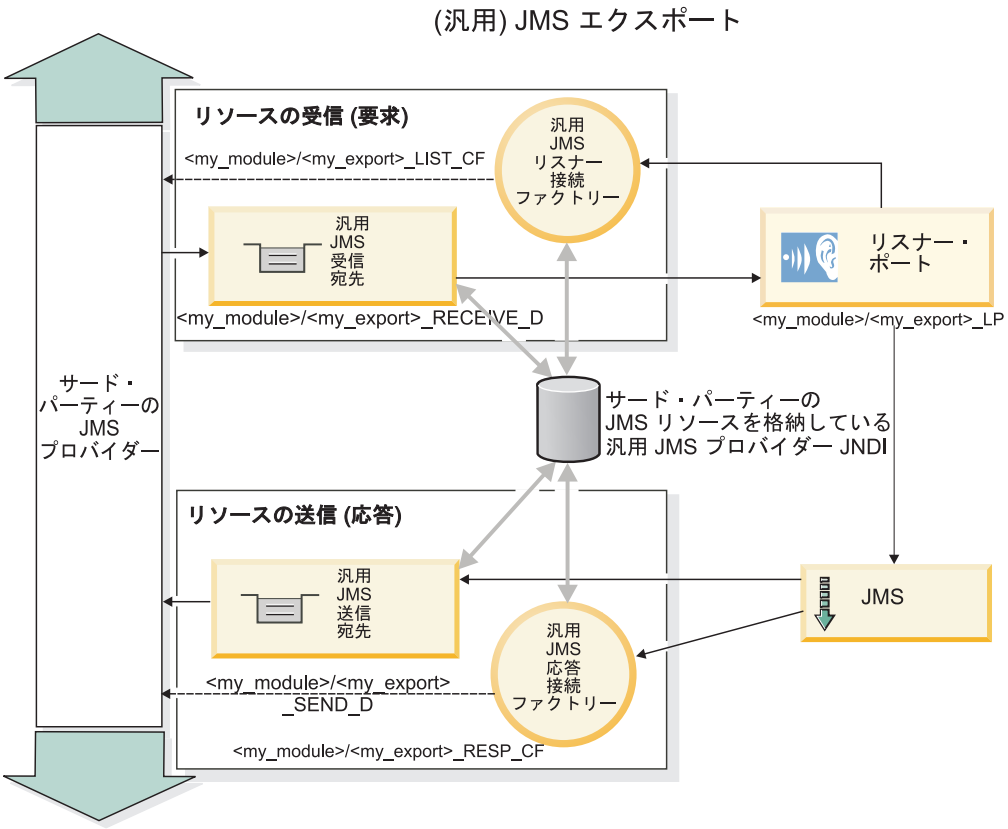


図 37. 汎用 JMS エクスポート・バインディング・リソース

汎用 JMS バインディングの主な機能

汎用 JMS インポート・バインディングとエクスポート・バインディングの機能は、WebSphere 埋め込み JMS と MQ JMS インポート・バインディングの機能と整合性があります。主な機能は、ヘッダー定義および既存の Java EE リソースへのアクセスです。ただし、汎用という性質上、JMS プロバイダー固有の接続オプションはありません。また、このバインディングでは、デプロイメントおよびインストール時にリソースを生成できる機能が限定されています。

汎用インポート

MQ JMS インポート・アプリケーションと同様に、汎用 JMS 実装は非同期であり、3 つの呼び出し (片方向、両方向 (要求/応答ともいう)、コールバック) をサポートしています。

JMS インポートがデプロイされるとき、ランタイム環境によって提供されるメッセージ駆動型 Bean がデプロイされます。MDB は要求メッセージに対する応答を listen します。 MDB は、要求とともに送信される宛先に関連付けられています (この宛先を listen します)。この宛先は、JMS メッセージの replyTo ヘッダー・フィールドで指定されます。

汎用エクスポート

汎用 JMS エクスポート・バインディングは、結果の戻りの処理方法が EIS エクスポート・バインディングとは異なります。汎用 JMS エクスポートは、着信メッセージに指定された replyTo 宛先に応答を明示的に送信します。この宛先が指定されていない場合は、送信宛先が使用されます。

汎用 JMS エクスポートがデプロイされるときに、メッセージ駆動型 Bean (汎用 JMS インポートに使用されるものとは異なる MDB) がデプロイされます。この MDB は受信宛先で着信要求を listen し、次にその要求が SCA ランタイムで処理されるようにディスパッチします。

特殊ヘッダー

特殊ヘッダーのプロパティーは、汎用 JMS インポートとエクスポートで使用され、ターゲット・バインディングに対してメッセージの処理方法を指示します。

例えば、TargetFunctionName プロパティーは、呼び出されているエクスポートのインターフェース内の操作名を識別する際に、デフォルトの関数セレクターによって使用されます。

注: インポート・バインディングを構成して、各操作名に TargetFunctionName ヘッダーを設定することができます。

Java EE リソース

JMS バインディングを Java EE 環境にデプロイすると、いくつかの Java EE リソースが作成されます。

- インポートの場合は受信宛先 (応答、両方向のみ)、エクスポートの場合は受信宛先 (要求) で listen するためのリスナー・ポート

- outboundConnection (インポート) および inboundConnection (エクスポート) の汎用 JMS 接続ファクトリー
- 送信宛先 (インポート) および受信宛先 (エクスポート、両方向のみ) の汎用 JMS 宛先
- responseConnection の汎用 JMS 接続ファクトリー (両方向のみ、オプション。これ以外の場合は、インポートに outboundConnection が使用され、エクスポートに inboundConnection が使用される)
- 受信宛先 (インポート) および送信宛先 (エクスポート) の汎用 JMS 宛先 (両方向のみ)
- SIB コールバック・キュー宛先にアクセスするときに使用されるデフォルトのメッセージング・プロバイダー・コールバック JMS 宛先 (両方向のみ)
- コールバック JMS 宛先にアクセスするときに使用されるデフォルトのメッセージング・プロバイダー・コールバック JMS 接続ファクトリー (両方向のみ)
- 応答処理中に使用される要求メッセージに関する情報を格納するための SIB コールバック・キュー宛先 (両方向のみ)

インストール・タスクにより、インポート・ファイルおよびエクスポート・ファイルの情報から ConnectionFactory、3 つの宛先、および ActivationSpec が作成されます。

汎用 JMS ヘッダー

汎用 JMS ヘッダーは、すべての汎用 JMS メッセージ・プロパティーを含むサービス・データ・オブジェクト (SDO) です。これらのプロパティーはインバウンド・メッセージからのものか、アウトバウンド・メッセージに適用されるプロパティーにすることができます。

JMS メッセージには、2 つのタイプのヘッダーが含まれます。1 つは JMS システム・ヘッダー、もう 1 つは複数の JMS プロパティーです。メディエーション・モジュールでは、いずれのタイプのヘッダーにも、サービス・メッセージ・オブジェクト (SMO) 内、または ContextService API を使用してアクセスできます。

methodBinding で静的に設定されるプロパティーを以下に示します。

- JMSType
- JMSCorrelationID
- JMSDeliveryMode
- JMSPriority

また、汎用 JMS バインディングでは、JMS および MQ JMS バインディングと同じ方法で JMS ヘッダーとプロパティーの動的変更がサポートされます。

一部の汎用 JMS プロバイダーでは、アプリケーションにより設定できるプロパティーとその組み合わせが制限されます。詳しくは、サード・パーティー製品の資料を参照してください。ただし、methodBinding には、新しいプロパティー ignoreInvalidOutboundJMSProperties (例外の伝搬を許可するプロパティー) が追加されています。

汎用 JMS ヘッダーとメッセージのプロパティーは、基本 Service Component Architecture の SCDL バインディング・スイッチがオンになっている場合にのみ使

用されます。スイッチがオンになっているとき、コンテキスト情報が伝搬されます。デフォルトでは、このスイッチはオンになっています。コンテキスト情報の伝搬を回避するには、値を false に変更します。

コンテキスト伝搬を使用可能にすると、ヘッダー情報をメッセージまたはターゲット・コンポーネントに流すことができます。コンテキスト伝搬のオン/オフを切り替えるには、インポート・バインディングおよびエクスポート・バインディングの contextPropagationEnabled 属性に true または false を指定します。以下に例を示します。

```
<esbBinding xsi:type="eis:JMSImportBinding" contextProgagationEnabled="true">
```

デフォルトは true です。

汎用 JMS バインディングのトラブルシューティング

汎用 JMS バインディングで発生した問題は、診断して修正することができます。

実装例外

汎用 JMS のインポート実装とエクスポート実装は、さまざまなエラー状態に応じて、以下の 2 種類の例外のいずれかを戻すことがあります。

- サービス・ビジネス例外: サービス・ビジネス・インターフェース (WSDL ポート・タイプ) で指定された障害が発生した場合に、この例外が戻されます。
- サービス・ランタイム例外: その他のすべてのケースで生成されます。ほとんどの場合、cause 例外には元の例外 (JMSException) が含まれます。

汎用 JMS メッセージの有効期限のトラブルシューティング

JMS プロバイダーの要求メッセージには有効期限が設定されます。

要求有効期限 は、JMS プロバイダーの要求メッセージの JMSExpiration の時刻に達した場合の、その要求メッセージの有効期限を指します。その他の JMS バインディングの場合と同様に汎用 JMS バインディングも、インポートによって格納されたコールバック・メッセージの有効期限を発信要求と同じ有効期限に設定することにより、要求の有効期限を処理しています。コールバック・メッセージの有効期限の通知により、要求メッセージの有効期限が切れていることが示されます。クライアントには、ビジネス例外により期限切れを通知する必要があります。

コールバック宛先がサード・パーティー・プロバイダーに移動すると、このタイプの要求有効期限はサポートされなくなります。

応答有効期限 は、JMS プロバイダーの応答メッセージの JMSExpiration の時刻に達した場合の、その応答メッセージの有効期限を指します。

汎用 JMS バインディングでは応答有効期限はサポートされていません。これは、サード・パーティー JMS プロバイダーの正確な期限切れ動作が定義されていないためです。ただし、応答を受信する場合、応答を受信する時点で応答有効期限が切れていないかどうかを確認できます。

アウトバウンド要求メッセージでは、JMSExpiration 値は待機時間と、asyncHeader の requestExpiration 値 (設定されている場合) から算出されます。

汎用 JMS 接続ファクトリー・エラーのトラブルシューティング

汎用 JMS プロバイダーで特定のタイプの接続ファクトリーを定義すると、アプリケーションの開始時にエラー・メッセージを受信することがあります。この問題を回避するには、外部接続ファクトリーを変更します。

アプリケーションの起動時に、以下のエラー・メッセージを受け取ります。

```
MDB リスナー・ポート JMSConnectionFactory タイプが JMSDestination タイプと
一致しません (MDB Listener Port JMSConnectionFactory type
does not match JMSDestination type)
```

この問題は、外部接続ファクトリーの定義時に発生します。特に、JMS 1.1 (統合) 接続ファクトリー (Point-to-Point 通信およびパブリッシュ/サブスクライブ通信の両方をサポート可能な接続ファクトリー) ではなく JMS 1.0.2 トピック接続ファクトリーを作成すると、例外がスローされることがあります。

この問題を解決するには、以下のステップを実行します。

1. 使用している汎用 JMS プロバイダーにアクセスします。
2. 定義されている JMS 1.0.2 トピック接続ファクトリーを JMS 1.1 (統合) 接続ファクトリーに置き換えます。

新規に定義した JMS 1.1 接続ファクトリーを使用してアプリケーションを起動すると、エラー・メッセージは表示されないはずです。

Failed Event Manager に表示されない汎用 JMS ベースの SCA メッセージ

汎用 JMS の対話の失敗によって SCA メッセージが発生した場合は、Failed Event Manager でこのメッセージを見つけることになります。Failed Event Manager にこのようなメッセージが表示されない場合は、基礎となるリスナー・ポートの最大再試行回数プロパティーの値が 1 以上になっているかどうかを確認してください。この値を 1 以上に設定すると、汎用 JMS バインディングに対して SCA を呼び出す際に、Failed Event Manager と対話することができます。

例外の処理

バインディングがどのように構成されているかによって、データ・ハンドラーまたはデータ・バインディングによる例外の処理方法が決まります。また、メディエーション・フローの性質により、そのような例外がスローされたときのシステムの振る舞いが決まります。

データ・ハンドラーまたはデータ・バインディングがバインディングによって呼び出されるときには、さまざまな問題が発生する可能性があります。例えば、データ・ハンドラーが、ペイロードが破損しているメッセージを受信したり、誤った形式のメッセージを読み取るなどです。

バインディングによるそのような例外の処理方法は、データ・ハンドラーおよびデータ・バインディングの実装方法によって決まります。推奨される振る舞いは、`DataBindingException` をスローするようにデータ・バインディングを設計することです。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、すべてのデータ・ハンドラー例外はデータ・バインディング例外内にラップされます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

`DataBindingException` 例外などのランタイム例外がスローされた場合:

- メディエーション・フローがトランザクションとして構成されている場合、デフォルトでは、手動でやり直し、または削除できるように JMS メッセージは Failed Event Manager に保管されます。

  注: バインディングのリカバリー・モードを変更することによって、メッセージが Failed Event Manager に保管される代わりに、ロールバックされるようにできます。
- メディエーション・フローがトランザクションではない場合、例外はログに記録され、メッセージは失われます。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、データ・ハンドラー例外はデータ・バインディング例外内に生成されます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

WebSphere MQ JMS バインディング

WebSphere MQ JMS バインディングは、WebSphere MQ JMS ベースのプロバイダーを使用する外部アプリケーションとの統合を提供します。

WebSphere MQ JMS エクスポートおよびインポート・バインディングは、サーバー環境から外部 JMS または MQ JMS システムと直接統合する場合に使用します。これにより、サービス統合バスの MQ リンクまたはクライアント・リンク機能を使用する必要がなくなります。

コンポーネントがインポートによって WebSphere MQ JMS ベースのサービスと対話するときには、WebSphere MQ JMS インポート・バインディングはデータの送信先の宛先と応答を受信できる宛先を使用します。JMS メッセージとの間のデータ変換は、JMS データ・ハンドラーまたはデータ・バインディング・エッジ・コンポーネントを介して行います。

SCA モジュールが WebSphere MQ JMS クライアントにサービスを提供するときには、WebSphere MQ JMS エクスポート・バインディングは、要求の受信と応答の送信ができる宛先を使用します。JMS メッセージとの間のデータ変換は、JMS データ・ハンドラーまたはデータ・バインディングを介して行います。

関数セレクターは、呼び出すターゲット・コンポーネントに対する操作へのマッピングを提供します。

WebSphere MQ JMS バインディングの概要

WebSphere MQ JMS バインディングは、WebSphere MQ JMS プロバイダーを使用する外部アプリケーションとの統合を提供します。

WebSphere MQ 管理用タスク

WebSphere MQ システム管理者は、基盤となる WebSphere MQ キュー・マネージャーを作成する必要があります。このキュー・マネージャーは、WebSphere MQ JMS バインディングが含まれるアプリケーションを実行する前に、これらのバインディングによって使用されます。

WebSphere MQ JMS インポート・バインディング

WebSphere MQ JMS インポートにより、SCA モジュール内のコンポーネントは、WebSphere MQ JMS ベースのプロバイダーが提供するサービスと通信できるようになります。サポートされているバージョンの WebSphere MQ を使用していることを確認してください。詳細なハードウェアおよびソフトウェア要件については、IBM サポート・ページを参照してください。

WebSphere MQ JMS インポート・バインディングでは、呼び出されている操作のタイプに応じた以下の 2 種類の使用シナリオがサポートされています。

- 片方向: WebSphere MQ JMS インポートは、インポート・バインディングに構成された送信宛先にメッセージを送信します。 JMS ヘッダーの replyTo フィールドには何も送信しません。
- 両方向 (要求/応答): WebSphere MQ JMS インポートは、送信宛先にメッセージを送信します。

  receive 宛先が replyTo ヘッダー・フィールドに設定されます。受信宛先で listen するためのメッセージ駆動型 Bean (MDB) をデプロイします。応答を受信すると、MDB は応答をコンポーネントに返します。

  インポート・バインディングは、要求メッセージ ID (デフォルト) または要求メッセージ相関 ID から応答メッセージ相関 ID がコピーされていることを期待するように (WebSphere Integration Developer の「**応答相関スキーム**」フィールドを使用して) 構成することができます。

片方向と両方向のいずれのシナリオを使用する場合も、動的および静的ヘッダー・プロパティーを指定できます。静的プロパティーは JMS インポート・メソッド・バインディングから設定できます。 これらのプロパティーの一部は、SCA JMS ランタイムで特別な意味を持ちます。

重要な点として、WebSphere MQ JMS は非同期バインディングであることに注意してください。呼び出し側コンポーネントが WebSphere MQ JMS インポートを同期的に呼び出すと (両方向操作の場合)、呼び出し側コンポーネントは、JMS サービスからの応答が返されるまでブロックされます。

102 ページの図 38 は、インポートがどのように外部サービスにリンクされているのかを示しています。

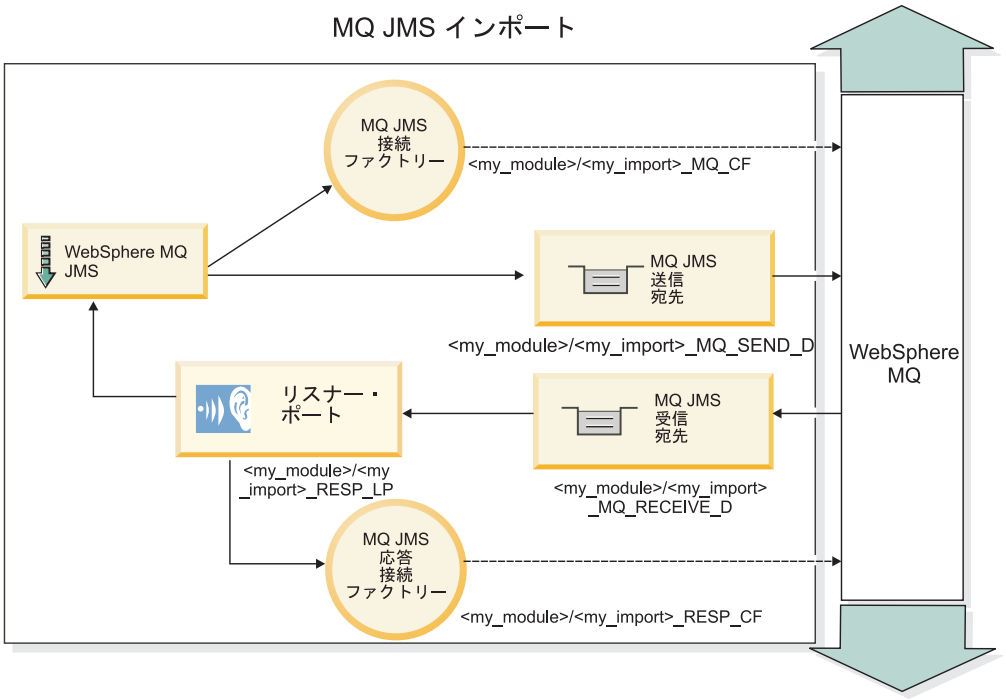


図 38. WebSphere MQ JMS インポート・バインディングのリソース

WebSphere MQ JMS エクスポート・バインディング

WebSphere MQ JMS エクスポート・バインディングは、SCA モジュールが WebSphere MQ ベースの JMS プロバイダーで外部 JMS アプリケーションにサービスを提供する手段を提供します。

エクスポート・バインディングで指定された receive 宛先に着信する要求を listen するため、MDB がデプロイされます。send フィールドで指定された宛先は、呼び出されたコンポーネントが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。応答メッセージの replyTo フィールドで指定された値は、send フィールドで指定された宛先をオーバーライドします。

103 ページの図 39 は、外部の要求側がどのようにエクスポートにリンクされているのかを示しています。

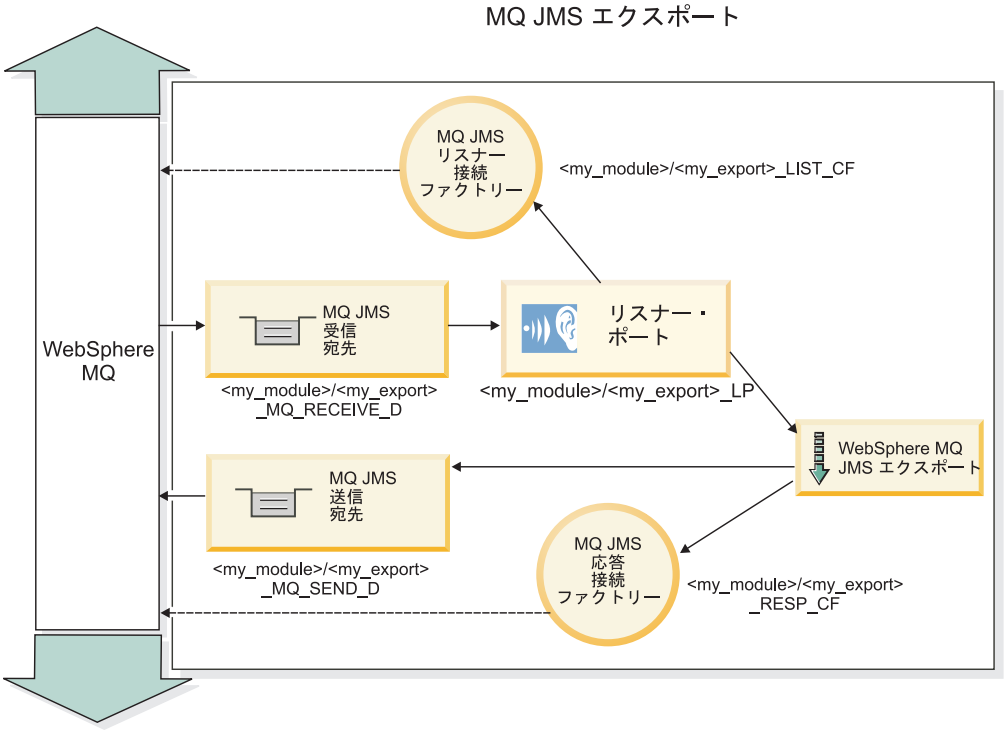


図 39. WebSphere MQ JMS エクスポート・バインディングのリソース

注: 102 ページの図 38 および 図 39 は、旧バージョンの WebSphere Process Server のアプリケーションが外部サービスにリンクされる方法を示しています。WebSphere Process Server バージョン 7.0 用に開発されたアプリケーションについては、リスナー・ポートと接続ファクトリーではなくアクティベーション・スペックが使用されます。

WebSphere MQ JMS バインディングの主な特徴

WebSphere MQ JMS バインディングの主な特徴は、ヘッダー、Java EE 成果物、および作成される Java EE リソースです。

ヘッダー

JMS メッセージ・ヘッダーには、いくつかの定義済みフィールドがあります。これらのフィールドには、クライアントとプロバイダーの両方でメッセージの識別と送付に使用される値が格納されます。バインディング・プロパティーを使用して、固定値でこれらのヘッダーを設定するか、またはヘッダーを実行時に動的に指定することができます。

JMSCorrelationID
関連メッセージへのリンクです。通常、このフィールドは、応答の対象となるメッセージのメッセージ ID のストリングに設定されます。

TargetFunctionName
このヘッダーは、呼び出された操作を識別する際に、指定されたいずれかの関数セレクターによって使用されます。TargetFunctionName JMS ヘッダー・プロパティーを JMS エクスポートに送信されたメッセージ内に設定することにより、この関数セレクターが使用可能になります。このプロパティーは、JMS クライ

アント・アプリケーションに直接設定することも、JMS バインディングが指定されたインポートを JMS エクスポートに接続する際に設定することもできます。この場合、JMS インポート・バインディングを構成し、操作名とのインターフェースにおいて、各操作ごとに TargetFunctionName ヘッダーを設定する必要があります。

相関スキーム

WebSphere MQ JMS バインディングは、要求メッセージと応答メッセージを相関させる方法を決定するためのさまざまな相関スキームを提供します。

RequestMsgIDToCorrelID
JMSMessageID は JMSCorrelationID フィールドにコピーされます。これはデフォルト設定です。

RequestCorrelIDToCorrelID
JMSCorrelationID は JMSCorrelationID フィールドにコピーされます。

Java EE リソース

MQ JMS インポートを Java EE 環境にデプロイすると、いくつかの Java EE リソースが作成されます。

パラメーター

MQ 接続ファクトリー
クライアントが MQ JMS プロバイダーとの接続を作成するために使用します。

応答接続ファクトリー
送信宛先が受信宛先とは異なるキュー・マネージャー上にある場合に、SCA MQ JMS ランタイムが使用します。

アクティベーション・スペック
MQ JMS アクティベーション・スペックは、1 つ以上のメッセージ駆動型 Bean に関連付けられており、これらの Bean がメッセージを受信するのに必要な構成を提供します。

宛先

- 送信宛先:
  - インポート: 要求または出力メッセージが送信される宛先です。
  - エクスポート: 応答メッセージが送信される宛先です。ただし、着信メッセージの JMSReplyTo ヘッダー・フィールドにより置き換えられた場合は、その宛先が優先されます。
- 受信宛先:
  - インポート: 応答メッセージまたは着信メッセージが格納される宛先です。
  - エクスポート: 着信メッセージまたは要求メッセージが格納される宛先です。

JMS ヘッダー

JMS メッセージには、2 つのタイプのヘッダーが含まれます。1 つは JMS システム・ヘッダー、もう 1 つは複数の JMS プロパティーです。メディエーション・モ

ジュールでは、いずれのタイプのヘッダーにも、サービス・メッセージ・オブジェクト (SMO) 内、または ContextService API を使用してアクセスできます。

JMS システム・ヘッダー

SMO では、JMS システム・ヘッダーは JMSHeader エレメントによって表されます。このエレメントには、JMS ヘッダーに通常あるすべてのフィールドが含まれます。これらのフィールドはメディエーション (または ContextService) で変更できますが、SMO に設定された一部の JMS システム・ヘッダー・フィールドは、システムまたは静的な値によってオーバーライドされるため、アウトバウンド JMS メッセージでは伝搬されません。

メディエーション (または ContextService) で更新可能な JMS システム・ヘッダーのキー・フィールドには以下があります。

- **JMSType** および **JMSCorrelationID** - 特定の事前定義メッセージ・ヘッダー・プロパティーの値
- **JMSDeliveryMode** - 送達モードの値 (persistent または nonpersistent。デフォルトは persistent)
- **JMSPriority** - 優先度の値 (0 から 9。デフォルトは JMS_Default_Priority)

JMS プロパティー

JMS プロパティーは、SMO ではプロパティー・リスト内のエントリーとして表されます。プロパティーは、メディエーション内で、または ContextService API を使用して追加、更新、または削除できます。

プロパティーは、JMS バインディングに静的に設定することもできます。静的に設定されたプロパティーは、動的に設定される (同じ名前の) 設定をオーバーライドします。

他のバインディング (例えば HTTP バインディング) から伝搬されるユーザー・プロパティーは、JMS バインディングでは JMS プロパティーとして出力されます。

ヘッダー伝搬の設定

JMS システム・ヘッダーおよびプロパティーの、インバウンド JMS メッセージからダウンストリームのコンポーネントへの伝搬、またはアップストリームのコンポーネントからアウトバウンド JMS メッセージへの伝搬は、バインディングのプロトコル・ヘッダー伝搬フラグで制御できます。

プロトコル・ヘッダー伝搬を設定すると、以下のリストに説明するようにヘッダー情報をメッセージまたはターゲット・コンポーネントに流すことができます。

- JMS エクスポート要求

  メッセージ内で受信した JMS ヘッダーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。メッセージ内で受信した JMS プロパティーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。
- JMS エクスポート応答

コンテキスト・サービスに設定されたすべての JMS ヘッダー・フィールドは、JMS エクスポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。コンテキスト・サービスに設定されたすべてプロパティーは、JMS エクスポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。

- JMS インポート要求

  コンテキスト・サービスに設定されたすべての JMS ヘッダー・フィールドは、JMS インポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。コンテキスト・サービスに設定されたすべてプロパティーは、JMS インポート・バインディングに設定された静的プロパティーによってオーバーライドされていない限り、アウトバウンド・メッセージ内で使用されます。

- JMS インポート応答

  メッセージ内で受信した JMS ヘッダーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。メッセージ内で受信した JMS プロパティーは、コンテキスト・サービスを介してターゲット・コンポーネントに伝搬されます。

外部クライアント

サーバーは、WebSphere MQ JMS バインディングを使用して、外部クライアントとの間でメッセージを送受信できます。

外部クライアント (Web ポータルやエンタープライズ情報システムなど) は、エクスポートによって、アプリケーションの SCA コンポーネントにメッセージを送信できます。また、アプリケーション内の SCA コンポーネントがインポートによって外部クライアントを呼び出すこともできます。

WebSphere MQ JMS エクスポート・バインディングは、エクスポート・バインディングで指定された receive 宛先に着信する要求を listen するためのメッセージ駆動型 Bean (MDB) をデプロイします。send フィールドで指定された宛先は、呼び出されたアプリケーションが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。したがって、外部クライアントは、エクスポート・バインディングを介してアプリケーションを呼び出すことができます。

WebSphere MQ JMS インポートは外部クライアントにバインドし、外部クライアントにメッセージを配信できます。このメッセージは、外部クライアントからの応答を要求してもしなくても構いません。

WebSphere MQ を使用して外部クライアントと対話する方法について詳しくは、WebSphere MQ インフォメーション・センターを参照してください。

WebSphere MQ JMS バインディングのトラブルシューティング

WebSphere MQ JMS バインディングで発生した問題を診断し、修正できます。

実装例外

MQ JMS のインポートおよびエクスポート実装は、さまざまなエラー状態に応じて、以下の 2 種類の例外のうちのいずれかを戻すことがあります。

- サービス・ビジネス例外: サービス・ビジネス・インターフェース (WSDL ポート・タイプ) で指定された障害が発生した場合に、この例外が戻されます。
- サービス・ランタイム例外: その他のすべてのケースで生成されます。ほとんどの場合、cause 例外には元の例外 (JMSException) が含まれます。

  例えばインポートの場合、要求メッセージごとに 1 つの応答メッセージだけが戻されることを前提としています。そのため、複数の応答を受信した場合や遅延応答 (SCA の応答有効期限が切れた応答) を受信した場合は、サービス・ランタイム例外がスローされます。この場合、トランザクションはロールバックされ、応答メッセージはキューからバックアウトされるか、または Failed Event Manager によって処理されます。

Failed Event Manager に表示されない JMS ベースの WebSphere MQ SCA メッセージ

WebSphere MQ JMS の対話の失敗によって SCA メッセージが発生した場合は、Failed Event Manager でこのメッセージを見つけることになります。Failed Event Manager にこのようなメッセージが表示されない場合は、基礎となるリスナー・ポートの最大再試行回数プロパティーの値が 1 以上になっているかどうかを確認してください。この値を 1 以上に設定すると、MQ JMS バインディングに対して SCA を呼び出す際に、Failed Event Manager と対話することができます。

誤用例: WebSphere MQ バインディングとの比較

WebSphere MQ JMS バインディングは、 WebSphere MQ に対してデプロイされている JMS アプリケーションと相互協調処理するよう設計されています。これにより、メッセージは JMS メッセージ・モデルに基づいて公開されます。これに対し、WebSphere MQ インポートおよびエクスポートは、ネイティブ WebSphere MQ アプリケーションと相互協調処理することができ、WebSphere MQ メッセージ本体の内容全体をメディエーションに公開するように設計されています。

以下のシナリオでは、WebSphere MQ バインディングではなく WebSphere MQ JMS バインディングを使用して作成する必要があります。

- JMS メッセージ駆動型 Bean (MDB) を SCA モジュールから呼び出す。この MDB は、WebSphere MQ JMS プロバイダーに対してデプロイされています。WebSphere MQ JMS インポートを使用します。
- SCA モジュールを Java EE コンポーネント・サーブレットまたは EJB から JMS を介して呼び出すことができるようにする。WebSphere MQ JMS エクスポートを使用します。
- WebSphere MQ 上で転送中の JMS MapMessage の内容のメディエーションを実行する。WebSphere MQ JMS エクスポートとインポート、および適切なデータ・ハンドラーまたはデータ ・バインディングを組み合わせて使用します。

WebSphere MQ バインディングと WebSphere MQ JMS バインディングの相互協調処理が予期される状況があります。特に、Java EE WebSphere MQ アプリケーショ

ンと非 Java EE WebSphere MQ アプリケーション間をブリッジングする場合は、WebSphere MQ エクスポートと WebSphere MQ JMS インポート (あるいはこの逆) を、適切なデータ・バインディングまたはメディエーション・モジュール (あるいはこの両方) と組み合わせて使用します。

例外の処理

バインディングがどのように構成されているかによって、データ・ハンドラーまたはデータ・バインディングによる例外の処理方法が決まります。また、メディエーション・フローの性質により、そのような例外がスローされたときのシステムの振る舞いが決まります。

データ・ハンドラーまたはデータ・バインディングがバインディングによって呼び出されるときには、さまざまな問題が発生する可能性があります。例えば、データ・ハンドラーが、ペイロードが破損しているメッセージを受信したり、誤った形式のメッセージを読み取るなどです。

バインディングによるそのような例外の処理方法は、データ・ハンドラーおよびデータ・バインディングの実装方法によって決まります。推奨される振る舞いは、`DataBindingException` をスローするようにデータ・バインディングを設計することです。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、すべてのデータ・ハンドラー例外はデータ・バインディング例外内にラップされます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

`DataBindingException` 例外などのランタイム例外がスローされた場合:

- メディエーション・フローがトランザクションとして構成されている場合、デフォルトでは、手動でやり直し、または削除できるように JMS メッセージは Failed Event Manager に保管されます。

  注: バインディングのリカバリー・モードを変更することによって、メッセージが Failed Event Manager に保管される代わりに、ロールバックされるようにできます。
- メディエーション・フローがトランザクションではない場合、例外はログに記録され、メッセージは失われます。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、データ・ハンドラー例外はデータ・バインディング例外内に生成されます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

WebSphere MQ バインディング

WebSphere MQ バインディングは、WebSphere MQ アプリケーションと Service Component Architecture (SCA) との接続を提供します。

WebSphere MQ エクスポートおよびインポート・バインディングは、サーバー環境から WebSphere MQ ベースのシステムと直接統合する場合に使用します。これにより、サービス統合バスの MQ リンクまたはクライアント・リンク機能を使用する必要がなくなります。

コンポーネントがインポートによって WebSphere MQ サービスと対話するときには、WebSphere MQ インポート・バインディングはデータの送信先のキューと応答を受信できるキューを使用します。

SCA モジュールが WebSphere MQ クライアントにサービスを提供するときには、WebSphere MQ エクスポート・バインディングは、要求の受信と応答の送信ができるキューを使用します。関数セレクターは、呼び出すターゲット・コンポーネントに対する操作へのマッピングを提供します。

MQ メッセージとの間のペイロード・データの変換は、MQ 本体データ・ハンドラーまたはデータ・バインディングを使用して行われます。 MQ メッセージとの間のヘッダー・データの変換は、MQ ヘッダー・データ・バインディングを使用して行われます。

サポートされる WebSphere MQ バージョンについては、Web サイトの WebSphere Process Server システム要件を参照してください。

WebSphere MQ バインディングの概要

WebSphere MQ バインディングにより、ネイティブ MQ ベースのアプリケーションとの統合が実現します。

WebSphere MQ 管理用タスク

WebSphere MQ システム管理者は、基盤となる WebSphere MQ キュー・マネージャーを作成する必要があります。このキュー・マネージャーは、WebSphere MQ バインディングが含まれるアプリケーションを実行する前に、これらのバインディングによって使用されます。

WebSphere 管理用タスク

Websphere の MQ リソース・アダプターの「**ネイティブ・ライブラリー・パス (Native library path)**」プロパティーを、サーバーでサポートされる WebSphere MQ バージョンに設定して、サーバーを再始動する必要があります。これにより、サポートされるバージョンの WebSphere MQ のライブラリーが使用されることが保証されます。詳細なハードウェアおよびソフトウェア要件については、IBM サポート・ページを参照してください。

WebSphere MQ インポート・バインディング

WebSphere MQ インポート・バインディングにより、SCA モジュール内のコンポーネントは、外部の WebSphere MQ ベースのアプリケーションが提供するサービスと通信できるようになります。サポートされているバージョンの WebSphere MQ を使用していることを確認してください。詳細なハードウェアおよびソフトウェア要件については、IBM サポート・ページを参照してください。

外部 WebSphere MQ システムとの対話では、要求を送信し、応答を受信するためのキューが使用されます。

WebSphere MQ インポート・バインディングでは、呼び出されている操作のタイプに応じた以下の 2 種類の使用シナリオがサポートされています。

- 片方向: WebSphere MQ インポートは、インポート・バインディングの「**送信宛先キュー (Send destination queue)**」フィールドに構成されたキューにメッセージを送信します。 MQMD ヘッダーの replyTo フィールドには何も送信しません。
- 両方向 (要求/応答): WebSphere MQ インポートは、「**送信宛先キュー (Send destination queue)**」フィールドに構成されたキューにメッセージを送信します。

  receive キューは、replyTo MQMD ヘッダー・フィールドに設定されます。 受信キューで listen するためのメッセージ駆動型 Bean (MDB) をデプロイします。応答を受信すると、MDB は応答をコンポーネントに返します。

  インポート・バインディングは、要求メッセージ ID (デフォルト) または要求メッセージ相関 ID から応答メッセージ相関 ID がコピーされていることを期待するように (「**応答相関スキーム**」フィールドを使用して) 構成することができます。

重要な点として、WebSphere MQ は非同期バインディングであることに注意してください。呼び出し側コンポーネントが WebSphere MQ インポートを同期的に呼び出すと (両方向操作の場合)、呼び出し側コンポーネントは、 WebSphere MQ サービスからの応答が返されるまでブロックされます。

図 40 は、インポートがどのように外部サービスにリンクされているのかを示しています。

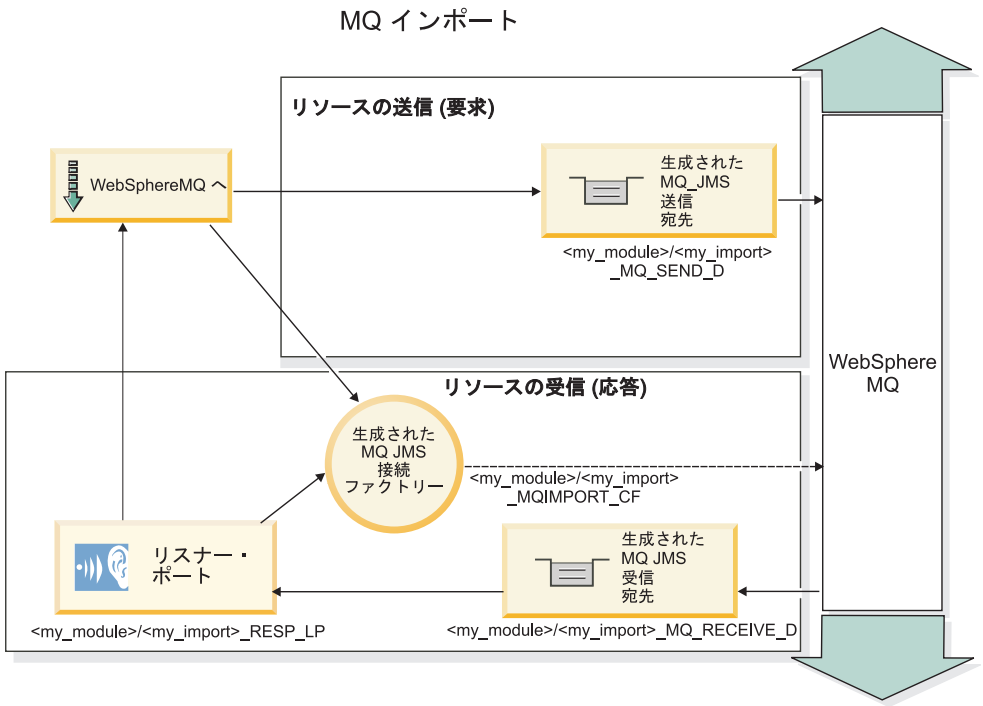


図 40. WebSphere MQ インポート・バインディングのリソース

WebSphere MQ エクスポート・バインディング

WebSphere MQ エクスポート・バインディングは、SCA モジュールが外部の WebSphere MQ ベースにアプリケーションにサービスを提供する手段を提供します。

エクスポート・バインディングで指定された「**受信宛先キュー (Receive destination queue)**」に着信する要求を listen するため、MDB がデプロイされます。「**送信宛先キュー (Send destination queue)**」 フィールドで指定されたキューは、呼び出されたコンポーネントが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。応答メッセージの replyTo フィールドで指定されたキューは、「**送信宛先キュー (Send destination queue)**」フィールドで指定されたキューをオーバーライドします。

図 41 は、外部の要求側がどのようにエクスポートにリンクされているのかを示しています。

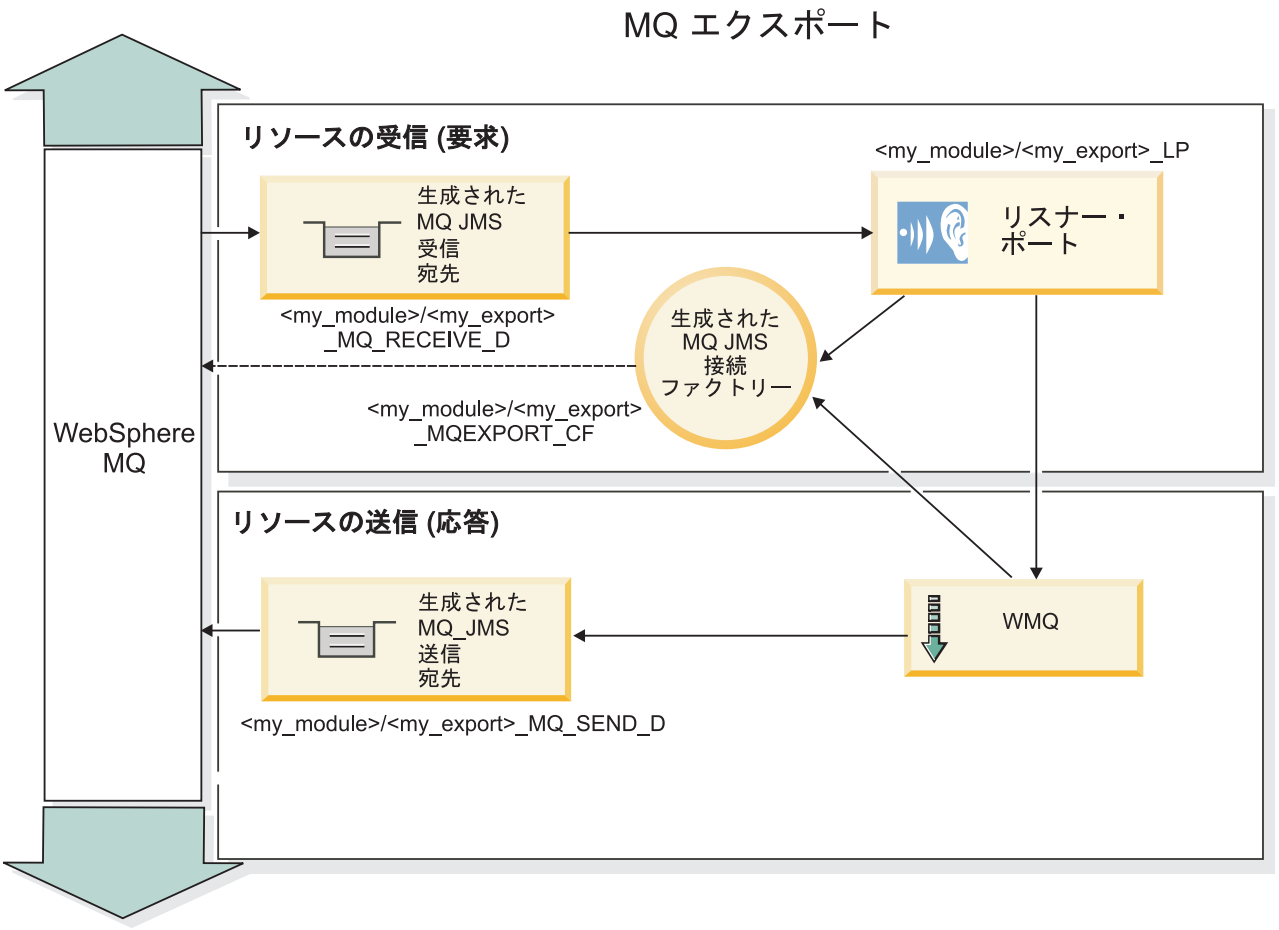


図 41. WebSphere MQ エクスポート・バインディングのリソース

注: 110 ページの図 40 および 図 41 は、旧バージョンの WebSphere Process Server のアプリケーションが外部サービスにリンクされる方法を示しています。WebSphere Process Server バージョン 7.0 用に開発されたアプリケーションについては、リスナー・ポートと接続ファクトリーではなくアクティベーション・スペックが使用されます。

WebSphere MQ バインディングの主な特徴

WebSphere MQ バインディングの主な特徴は、ヘッダー、Java EE 成果物、および作成される Java EE リソースです。

相関スキーム

WebSphere MQ 要求/応答アプリケーションでは、応答メッセージと要求を相関付けるさまざまな手法の 1 つを使用できます。これらの手法には、MQMD MessageID フィールドと CorrelID フィールドが関連します。ほとんどの場合、要求側はキュー・マネージャーに MessageID を選択させ、応答アプリケーションがこれを応答の CorrelID にコピーすることを想定します。多くの場合、要求側と応答アプリケーションは、使用される相関手法を暗黙的に認識しています。応答アプリケーションは、要求の Report フィールドに設定されているさまざまなフラグの指示に従って、これらのフィールドを処理することもあります。

WebSphere MQ メッセージのエクスポート・バインディングを構成するときには、以下のオプションを使用できます。

応答の MsgId のオプション:

新規 MsgID (New MsgID)
キュー・マネージャーが応答の固有の MsgId を選択できるようにします (デフォルト)。

要求の MsgID からコピーする (Copy from Request MsgID)
要求の MsgId フィールドから MsgId フィールドをコピーします。

SCA メッセージからコピーする (Copy from SCA message)
SCA 応答メッセージの WebSphere MQ ヘッダーに含まれる値を MsgId に設定します。この値が存在しない場合は、キュー・マネージャーに新規 Id を定義させます。

Report オプションを使用 (As Report Option)
要求の MQMD の Report フィールドを検査して、MsgId の処理方法を認識します。MQRO_NEW_MSG_ID オプションと MQRO_PASS_MSG_ID オプションがサポートされています。これらのオプションは、それぞれ「新規 MsgID (New MsgID)」および「要求の MsgID からコピーする (Copy from Request MsgID)」と同様に動作します。

応答の CorrelId のオプション:

要求の MsgID からコピーする (Copy from Request MsgID)
要求の MsgId フィールドから CorrelId フィールドをコピーします (デフォルト)。

要求の CorrelID からコピーする (Copy from Request CorrelID)
要求の CorrelId フィールドから CorrelId フィールドをコピーします。

SCA メッセージからコピーする (Copy from SCA message)
SCA 応答メッセージの WebSphere MQ ヘッダーに含まれる値を CorrelId に設定します。または、この値が存在しない場合は、空白のままにします。

Report オプションを使用 (As Report Option)
要求の MQMD の Report フィールドを検査して、CorrelId の処理方法を認識します。MQRO_COPY_MSG_ID_TO_CORREL_ID オプションと MQRO_PASS_CORREL_ID オプションがサポートされています。これらのオプ

ションは、それぞれ「要求の MsgID からコピーする (Copy from Request MsgID)」および「要求の CorrelID からコピーする (Copy from Request CorrelID)」と同様に動作します。

WebSphere MQ メッセージのインポート・バインディングを構成するときには、以下のオプションを使用できます。

要求の MsgId のオプション:

新規 MsgID (New MsgID)
キュー・マネージャーが要求の固有の MsgId を選択できるようにします (デフォルト)。

SCA メッセージからコピーする (Copy from SCA message)
SCA 要求メッセージの WebSphere MQ ヘッダーに含まれる値を MsgId に設定します。この値が存在しない場合は、キュー・マネージャーに新規 Id を定義させます。

応答相関のオプション:

応答の CorrelID を MsgId からコピーする (Response has CorrelID copied from MsgId)
応答メッセージの CorrelId フィールドが、要求の MsgId ごとに設定されていることを想定します (デフォルト)。

応答の MsgID を MsgId からコピーする (Response has MsgID copied from MsgId)
応答メッセージの MsgId フィールドが、要求の MsgId ごとに設定されていることを想定します。

応答の CorrelID を CorrelId からコピーする (Response has CorrelID copied from CorrelId)
応答メッセージの CorrelId フィールドが、要求の CorrelId ごとに設定されていることを想定します。

Java EE リソース

WebSphere MQ バインディングを Java EE 環境にデプロイすると、いくつかの Java EE リソースが作成されます。

パラメーター

MQ 接続ファクトリー
クライアントが WebSphere MQ プロバイダーとの接続を作成するために使用します。

応答接続ファクトリー
送信宛先が受信宛先とは異なるキュー・マネージャー上にある場合に、SCA MQ ランタイムが使用します。

アクティベーション・スペック
MQ JMS アクティベーション・スペックは、1 つ以上のメッセージ駆動型 Bean に関連付けられており、これらの Bean がメッセージを受信するのに必要な構成を提供します。

宛先

- 送信宛先: インポートの場合は、要求または出力メッセージが送信される宛先です。エクスポートの場合は、応答メッセージが送信される宛先です。ただし、着信メッセージの MQMD ReplyTo ヘッダー・フィールドにより置き換えられた場合は、その宛先が優先されます。
- 受信宛先: 応答/要求または着信メッセージが格納される宛先です。

WebSphere MQ ヘッダー

WebSphere MQ ヘッダーには、Service Component Architecture (SCA) メッセージへの変換に関する特定の規則が組み込まれています。

WebSphere MQ メッセージは、システム・ヘッダー (MQMD)、場合によっては 1 つ以上のその他の MQ ヘッダー (システムまたはカスタム)、 およびメッセージ本体で構成されます。メッセージ内に複数のメッセージ・ヘッダーがある場合、ヘッダーの順序が意味を持ちます。

各ヘッダーには、次のヘッダーの構造を記述する情報が含まれています。MQMD は最初のヘッダーを記述します。

MQ ヘッダーの解析方法

MQ ヘッダーの解析には、MQ ヘッダー・データ・バインディングが使用されます。以下のヘッダーは自動的にサポートされます。

- MQRFH
- MQRFH2
- MQCIH
- MQIIH

MQH で始まるヘッダーについては、処理方法が異なります。ヘッダーの特定のフィールドは解析されません。これらのフィールドは未解析バイトとして維持されます。

その他の MQ ヘッダーについては、カスタム MQ ヘッダー・データ・バインディングを作成して解析することができます。

MQ ヘッダーへのアクセス方法

製品内で MQ ヘッダーにアクセスするには、以下の 2 つの方法のいずれかを使用します。

- メディエーション内でサービス・メッセージ・オブジェクト (SMO) を使用
- ContextService API を使用

MQ ヘッダーは、内部では SMO MQHeader エレメントによって表されます。MQHeader は MQControl を拡張するヘッダー・データのコンテナーですが、anyType の値エレメントが含まれています。これには MQMD、MQControl (MQ メッセージ本体制御情報)、およびその他の MQ ヘッダーのリストが含まれます。

- MQMD は、WebSphere MQ メッセージ記述の内容を表します。ただし、本体の構造とエンコードを定義する情報は含まれません。
- MQControl には、メッセージ本体の構造とエンコードを定義する情報が含まれています。

- MQHeaders には、MQHeader オブジェクトのリストが含まれています。

MQ ヘッダー・チェーンはアンワインドされるため、SMO 内部では各 MQ ヘッダーが独自の制御情報 (CCSID、Encoding、および Format) を持つことになります。ヘッダーは簡単に追加または削除できます。他のヘッダー・データを変更する必要はありません。

MQMD のフィールドの設定

MQMD を更新するときは、コンテキスト API を使用するか、またはメディエーション内でサービス・メッセージ・オブジェクト (SMO) を使用します。以下のフィールドは、自動的にアウトバウンド MQ メッセージに伝搬されます。

- エンコード
- CodedCharacterSet
- フォーマット
- レポート
- 有効期限
- フィードバック
- 優先順位
- パーシスタンス
- CorrelId
- MsgFlags

以下のプロパティーがアウトバウンド MQ メッセージに伝搬されるように、インポートまたはエクスポートの MQ バインディングを構成します。

MsgID
「**要求メッセージ ID (Request Message ID)**」に 「SCA メッセージからのコピー」を設定します。

MsgType
「**要求/応答操作のメッセージ・タイプを MQMT_DATAGRAM または MQMT_REQUEST に設定**」チェック・ボックスのチェック・マークを外します。

ReplyToQ
「**要求メッセージの応答先キューのオーバーライド**」チェック・ボックスのチェック・マークを外します。

ReplyToQMgr
「**要求メッセージの応答先キューのオーバーライド**」チェック・ボックスのチェック・マークを外します。

バージョン 7.0 以降では、JNDI 宛先定義のカスタム・プロパティーを使用して、コンテキスト・フィールドをオーバーライドすることが可能です。送信宛先上の値 SET_IDENTITY_CONTEXT を使用してカスタム・プロパティー MDCTX を設定し、以下のフィールドをアウトバウンド MQ メッセージに伝搬します。

- UserIdentifier
- AppIdentityData

送信宛先上の値 SET_ALL_CONTEXT を使用してカスタム・プロパティー MDCTX を設定し、以下のプロパティーをアウトバウンド MQ メッセージに伝搬します。

- UserIdentifier
- AppIdentityData
- PutApplType
- PutApplName
- ApplOriginData

アウトバウンド MQ メッセージに伝搬されないフィールドもあります。以下のフィールドは、メッセージの送信時にオーバーライドされます。

- BackoutCount
- AccountingToken
- PutDate
- PutTime
- Offset
- OriginalLength

WebSphere MQ バインディングへの MQCIH の静的な追加

WebSphere Process Server では、メディエーション・モジュールを使用することなく、MQCIH ヘッダー情報を静的に追加することができます。

メッセージに MQCIH ヘッダー情報を追加するには、いくつかの方法があります (例えば、ヘッダー・セッター・メディエーション・プリミティブを使用するのも、1 つの方法です)。このヘッダー情報を静的に追加すると、追加のメディエーション・モジュールを使用しなくてもよいので便利です。静的なヘッダー情報 (CICS® プログラム名、トランザクション ID、その他のデータ形式ヘッダー詳細情報など) を定義し、WebSphere MQ バインディングの一部として追加することができます。

MQCIH ヘッダー情報を静的に追加するには、WebSphere MQ、MQ CICS Bridge、CICS を構成する必要があります。

WebSphere Integration Developer を使用して、MQCIH ヘッダー情報に必要な静的値を WebSphere MQ インポートに設定できます。

到着したメッセージが WebSphere MQ インポートによって処理される際に、MQCIH ヘッダー情報がメッセージ内に存在するかどうかが確認されます。MQCIH が存在する場合、WebSphere MQ インポート内に定義されている静的値を使用して、メッセージ内の対応する動的値がオーバーライドされます。MQCIH が存在しない場合、メッセージ内に MQCIH が作成され、WebSphere MQ インポート内で定義されている静的値が追加されます。

WebSphere MQ インポート内に定義されている静的値には、メソッドごとに固有の値が設定されます。同じ WebSphere MQ インポート内のさまざまなメソッドごとに、異なる静的 MQCIH 値を指定することができます。

この機能は、MQCIH に特定のヘッダー情報が含まれていない場合にデフォルト値を提供する目的では使用されません。この理由は、WebSphere MQ インポートで定義された静的な値によって、着信メッセージで指定された対応する値が上書きされるためです。

外部クライアント

WebSphere Process Server は、WebSphere MQ バインディングを使用して、外部クライアントとの間でメッセージを送受信できます。

外部クライアント (Web ポータルやエンタープライズ情報システムなど) は、エクスポートによって、アプリケーションの SCA コンポーネントにメッセージを送信できます。また、アプリケーション内の SCA コンポーネントがインポートによって外部クライアントを呼び出すこともできます。

WebSphere MQ エクスポート・バインディングは、エクスポート・バインディングで指定された receive 宛先に着信する要求を listen するためのメッセージ駆動型 Bean (MDB) をデプロイします。send フィールドで指定された宛先は、呼び出されたアプリケーションが応答を返す場合にインバウンド要求に対する応答を送信するために使用されます。したがって、外部クライアントは、エクスポート・バインディングを介してアプリケーションを呼び出すことができます。

WebSphere MQ インポートは外部クライアントにバインドし、外部クライアントにメッセージを配信できます。このメッセージは、外部クライアントからの応答を要求してもしなくても構いません。

WebSphere MQ を使用して外部クライアントと対話する方法について詳しくは、WebSphere MQ インフォメーション・センターを参照してください。

WebSphere MQ バインディングのトラブルシューティング

WebSphere MQ バインディングで発生する障害または失敗の状態を診断し、このような状態を修正できます。

主な障害状態

WebSphere MQ バインディングの主な障害状態は、トランザクションの意味構造、WebSphere MQ 構成、またはその他のコンポーネントの既存の動作への参照に基づいて判別されます。主な障害状態は以下のとおりです。

- WebSphere MQ キュー・マネージャーまたはキューに接続できない。

  WebSphere MQ に接続してメッセージを受信できない場合は、MDB リスナー・ポートを開始することができません。この状態は、WebSphere Application Server ログに記録されます。永続メッセージは、正常に取得されるまで (または WebSphere MQ により期限切れとなるまで) WebSphere MQ キューに残ります。

  WebSphere MQ に接続してアウトバウンド・メッセージを送信できない場合は、送信操作を制御するトランザクションがロールバックされます。
- インバウンド・メッセージを解析できないか、アウトバウンド・メッセージを構成できない。

データ・バインディングが失敗すると、作業を制御するトランザクションがロールバックされます。

- アウトバウンド・メッセージを送信できない。

  メッセージを送信できないと、関連するトランザクションがロールバックされます。

- 複数の応答メッセージまたは予期しない応答メッセージが返される。

  インポートの場合、要求メッセージごとに 1 つの応答メッセージだけが戻されることを前提としています。そのため、複数の応答を受信した場合や遅延応答 (SCA の応答有効期限が切れた応答) を受信した場合は、サービス・ランタイム例外がスローされます。この場合、トランザクションはロールバックされ、応答メッセージはキューからバックアウトされるか、または Failed Event Manager によって処理されます。

誤用例: WebSphere MQ JMS バインディングとの比較

通常、WebSphere MQ インポートおよびエクスポートは、ネイティブ WebSphere MQ アプリケーションと相互協調処理し、WebSphere MQ メッセージ本体の内容全体をメディエーションに公開するように設計されています。一方、WebSphere MQ JMS バインディングは、WebSphere MQ に対してデプロイされている JMS アプリケーションと相互協調処理するよう設計されています。これにより、メッセージは JMS メッセージ・モデルに基づいて公開されます。

以下のシナリオでは、WebSphere MQ バインディングではなく WebSphere MQ JMS バインディングを使用して作成する必要があります。

- JMS メッセージ駆動型 Bean (MDB) を SCA モジュールから呼び出す。この MDB は、WebSphere MQ JMS プロバイダーに対してデプロイされています。WebSphere MQ JMS インポートを使用します。
- SCA モジュールを Java EE コンポーネント・サーブレットまたは EJB から JMS を介して呼び出すことができるようにする。WebSphere MQ JMS エクスポートを使用します。
- WebSphere MQ 上で転送中の JMS MapMessage の内容のメディエーションを実行する。WebSphere MQ JMS エクスポートとインポート、および適切なデータ・バインディングを組み合わせて使用します。

WebSphere MQ バインディングと WebSphere MQ JMS バインディングの相互協調処理が予期される状況があります。特に、Java EE WebSphere MQ アプリケーションと非 Java EE WebSphere MQ アプリケーション間をブリッジングする場合は、WebSphere MQ エクスポートと WebSphere MQ JMS インポート (あるいはこの逆) を、適切なデータ・バインディングまたはメディエーション・モジュール (あるいはこの両方) と組み合わせて使用します。

未配布メッセージ

構成エラーなどが原因で WebSphere MQ がメッセージを対象の宛先に配信できない場合、メッセージは指定されている送達不能キューに送信されます。

このとき、メッセージ本体の先頭には、送達不能ヘッダーが追加されます。このヘッダーには、失敗の原因や元の宛先などの情報が含まれています。

Failed Event Manager に表示されない MQ ベースの SCA メッセージ

WebSphere MQ の対話の失敗によって SCA メッセージが発生した場合は、Failed Event Manager でこのメッセージを見つけることになります。Failed Event Manager にこれらのメッセージが表示されない場合は、基盤となる WebSphere MQ 宛先の最大配信失敗回数の値が 1 よりも大きいことを確認してください。この値を 2 以上に設定すると、WebSphere MQ バインディングに対する SCA 呼び出しの間の Failed Event Manager との対話が可能になります。

誤ったキュー・マネージャーに再生される MQ 失敗イベント

事前定義の接続ファクトリーをアウトバウンド接続に使用する場合、接続プロパティーは、インバウンド接続に使用されるアクティベーション・スペックで定義されている接続プロパティーと一致していなければなりません。

事前定義の接続ファクトリーは、失敗イベントの再生時に接続を作成するために使用されるので、メッセージをもともと受信したのと同じキュー・マネージャーを使用するように構成する必要があります。

例外の処理

バインディングがどのように構成されているかによって、データ・ハンドラーまたはデータ・バインディングによる例外の処理方法が決まります。また、メディエーション・フローの性質により、そのような例外がスローされたときのシステムの振る舞いが決まります。

データ・ハンドラーまたはデータ・バインディングがバインディングによって呼び出されるときには、さまざまな問題が発生する可能性があります。例えば、データ・ハンドラーが、ペイロードが破損しているメッセージを受信したり、誤った形式のメッセージを読み取るなどです。

バインディングによるそのような例外の処理方法は、データ・ハンドラーおよびデータ・バインディングの実装方法によって決まります。推奨される振る舞いは、`DataBindingException` をスローするようにデータ・バインディングを設計することです。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、すべてのデータ・ハンドラー例外はデータ・バインディング例外内にラップされます。したがって、`DataHandlerException` は `DataBindingException` として報告されます。

`DataBindingException` 例外などのランタイム例外がスローされた場合:

- メディエーション・フローがトランザクションとして構成されている場合、デフォルトでは、手動でやり直し、または削除できるように JMS メッセージは Failed Event Manager に保管されます。

  注: バインディングのリカバリー・モードを変更することによって、メッセージが Failed Event Manager に保管される代わりに、ロールバックされるようにできます。
- メディエーション・フローがトランザクションではない場合、例外はログに記録され、メッセージは失われます。

データ・ハンドラーの場合も同様の状況になります。データ・ハンドラーはデータ・バインディングによって呼び出されるため、データ・ハンドラー例外はデータ・バインディング例外内に生成されます。したがって、DataHandlerException は DataBindingException として報告されます。

バインディングの制限

バインディングには、その使用時に以下に示すいくつかの制限があります。

MQ バインディングの制限

MQ バインディングには、その使用時に以下に示すいくつかの制限があります。

パブリッシュ/サブスクライブ方式のメッセージ配布機能なし

WMQ 自体はパブリッシュ/サブスクライブをサポートしますが、メッセージ配布のパブリッシュ/サブスクライブ方式は、現時点では、MQ バインディングによってサポートされていません。ただし、MQ JMS バインディングは、この配布方式をサポートしています。

共用受信キュー

複数の WebSphere MQ エクスポート・バインディングおよびインポート・バインディングでは、構成されている受信キュー上のメッセージはすべて、そのエクスポートまたはインポートの対象であると期待します。インポート・バインディングとエクスポート・バインディングを構成するときは、以下の点に考慮する必要があります。

- 各 MQ インポートにはそれぞれ異なる受信キューが必要です。これは、MQ インポート・バインディングでは、その受信キュー上のすべてのメッセージは自身が送信した要求に対する応答であると想定されるためです。受信キューが複数のインポートで共用されている場合、 誤ったインポートで応答が受信される場合があり、その応答は元の要求メッセージに相関されません。
- 各 MQ エクスポートにはそれぞれ異なる受信キューが必要です。これは、そうでない場合、どのエクスポートで特定の要求メッセージが取得されるのか予測できないためです。
- MQ インポートと MQ エクスポートで、同じ送信キューを指すことはできます。

JMS、MQ JMS、および汎用 JMS バインディングの制限

JMS および MQ JMS バインディングには、いくつかの制限があります。

デフォルトのバインディングを生成する意味

JMS、MQ JMS、および汎用 JMS バインディングを使用する場合の制限については、以下のセクションで説明します。

- デフォルトのバインディングを生成する意味
- 応答相関スキーム
- 双方向言語サポート

バインディングを生成するときに、自分で値を入力することを選択しなかった場合は、いくつかのフィールドにデフォルトとして値が自動的に入力されます。例えば、接続ファクトリー名が自動的に作成されます。アプリケーションをサーバー上に置き、このアプリケーションにクライアントからリモートでアクセスすることが分かっている場合は、デフォルトを採用するのではなく、バインディングの作成時に JNDI 名を入力してください。これらの値を実行時に管理コンソールから管理することになる可能性が高いためです。

ただし、デフォルトを確定した後で、リモート・クライアントからアプリケーションにアクセスできないことが分かった場合は、管理コンソールを使用して接続ファクトリーの値を明示的に設定できます。接続ファクトリー設定のプロバイダー・エンドポイント・フィールドを見つけて、<server_hostname>:7276 (デフォルトのポート番号を使用している場合) などの値を追加します。

応答相関スキーム

要求/応答操作でメッセージの相関をとるために使用される CorrelationId To CorrelationId 応答相関スキームを使用する場合は、メッセージ内に動的相関 ID が必要です。

メディエーション・フロー・エディターを使用してメディエーション・モジュール内に動的相関 ID を作成するには、JMS バインディングを使用するインポートの前に XSLT ノードを追加します。XSLT マッピング・エディターを開きます。ターゲット・メッセージには、既知の Service Component Architecture ヘッダーを使用できます。ソース・メッセージ内の固有 ID が入っているフィールドをドラッグして、ターゲット・メッセージの JMS ヘッダー内にある相関 ID にドロップします。

双方向言語サポート

実行時の Java Naming and Directory Interface (JNDI) 名に対しては、ASCII 文字のみがサポートされます。

共用受信キュー

複数のエクスポート・バインディングとインポート・バインディングでは、構成されている受信キュー上のメッセージはすべて、そのエクスポートまたはインポートの対象であると期待します。インポート・バインディングとエクスポート・バインディングを構成するときは、以下の点に考慮する必要があります。

- 各インポート・バインディングにはそれぞれ異なる受信キューが必要です。これは、インポート・バインディングでは、その受信キュー上のすべてのメッセージは自身が送信した要求に対する応答であると想定されるためです。受信キューが複数のインポートで共用されている場合、誤ったインポートで応答が受信される場合があり、その応答は元の要求メッセージに相関されません。
- 各エクスポートにはそれぞれ異なる受信キューが必要です。これは、そうでない場合、どのエクスポートで特定の要求メッセージが取得されるのか予測できないためです。
- インポートとエクスポートで、同じ送信キューを指すことはできます。

第 3 章 プログラミング・ガイドおよび手法

このセクションでは、プログラミング・ガイドと例を説明します。

以下のサブトピックに、プログラミングの各種コンポーネント、アプリケーション、およびビジネス・インテグレーション・ソリューションに関する情報が記載されています。

重要: WebSphere Process Server および WebSphere Enterprise Service Bus でサポートされるアプリケーション・プログラミング・インターフェース (API) とシステム・プログラミング・インターフェース (SPI) について詳しくは、インフォメーション・センターの『参照』セクションを参照してください。

Service Component Architecture プログラミング

Service Component Architecture (SCA) は、サービス指向アーキテクチャー (SOA) に基づいてアプリケーションを構成するための、単純であるが強力なプログラミング・モデルを提供します。

Service Component Definition Language

Service Component Definition Language (SCDL) は、モジュール、コンポーネント、参照、インポート、およびエクスポートなどの Service Component Architecture (SCA) エレメントを記述するために使用される XML ベースの言語です。

SCA に存在するさまざまな成果物タイプは、このサービス指向アーキテクチャーの基本的な要件のいくつかをサポートするために設計されました。まず始めに、SCA では、基本サービス・コンポーネントを定義するメカニズムが必要です。サービス・コンポーネントを定義するメカニズムが用意されると、現行の SCA モジュールの内部または外部の両方のクライアントに対して、それらのサービスを使用可能にできることが重要になります。これに加えて、現行の SCA モジュールの外部のサービスをインポートおよび参照するように設計された構成体が存在しなければなりません。最後に、SCA は、サービスおよびモジュールを組み合わせて大規模アプリケーションを作成するための構造を提供します。

SCDL 定義は、いくつかのファイルにまたがって編成されます。例えば、クレジット承認アプリケーションでは、インターフェースおよび実装の SCDL を `CreditApproval.component` というファイルに格納できます。参照は、モジュールのルートにある `CreditApproval.component` ファイル (インライン) または独立した `sca.references`ファイルに含めることができます。スタンドアロン参照は、`sca.references` ファイルに入れます。同じ SCA モジュール内部の非 SCA 成果物 (JSP) が、SCA コンポーネントを呼び出すために、スタンドアロン参照を使用できます。

表 14. SCA サービス・モジュールを構成する主な成果物

成果物	SCDL 定義
モジュール定義	• ルート SCA プロジェクト JAR の sca.module ファイルに含まれています。
サービス・コンポーネント	• モジュールには、0 から n 個のサービス定義を含めることができます。 • 各コンポーネント定義は、<SERVICE_NAME>.component ファイルに含まれています。
インポート	• モジュールには、0 から n 個のインポート定義を含めることができます。 • 各インポート定義は、<IMPORT_NAME>.import ファイルに含まれています。
エクスポート	• モジュールには、0 から n 個のエクスポート定義を含めることができます。 • 各エクスポート定義は、<EXPORT_NAME>.export ファイルに含まれています。
参照	• 2 つのタイプの参照 – インライン (サービス・コンポーネント定義内に含まれる) – スタンドアロン • 各コンポーネント定義は、<SERVICE_NAME>.reference ファイルに含まれています。
その他の成果物	• その他の成果物としては、Java クラス、WSDL ファイル、その他の成果物 XSD ファイル、BPEL などがあります。

SCA アプリケーションをビルドするとき、WebSphere Integration Developer が、適切な SCDL 定義の生成を担当します。ただし、SCDL の基本を知っておくと、アーキテクチャー全体の理解に役立ち、アプリケーションのデバッグ時にも役立ちます。

モジュール定義

Service Component Architecture (SCA) は、コンポーネントをサービス・モジュールにパッケージ化するための標準のデプロイメント・モデルを定義します。sca.module ファイルには、モジュールの定義が含まれています。

SCA モジュールは、単なるパッケージのタイプではありません。WebSphere Process Server では、SCA サービス・モジュールは、Java EE EAR ファイルおよびその他のいくつかの Java EE サブモジュールに相当します。WAR ファイルなどの Java EE エレメントは、SCA モジュールと併せてパッケージできます。非 SCA 成果物 (JSP など) も、SCA サービス・モジュールと一緒にパッケージ化することができま

す。これにより、それらの成果物が、スタンドアロン参照と呼ばれる特殊な参照タイプを使用した SCA クライアント・プログラミング・モデルによって SCA サービスを呼び出せるようになります。

以下に sca.module ファイルの例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:module xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
name="CreditApproval"/>
```

図には、WebSphere Integration Developer 内のモジュールと、その関連 SCDL モジュール定義を示します。それらはエディターで表示できます。この例では、モジュール・タイプはメディエーション・モジュールです。

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:module xmlns:mt="http://www.ibm.com/xmlns/prod/websphere/scdl/moduletype/7.0.0"
  Xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"name="StockQuote">
  <mt:moduleType type="com.ibm.ws.sca.scdl.moduletype.mediation" version="1.0.0"/>
</Scdl:module>
```

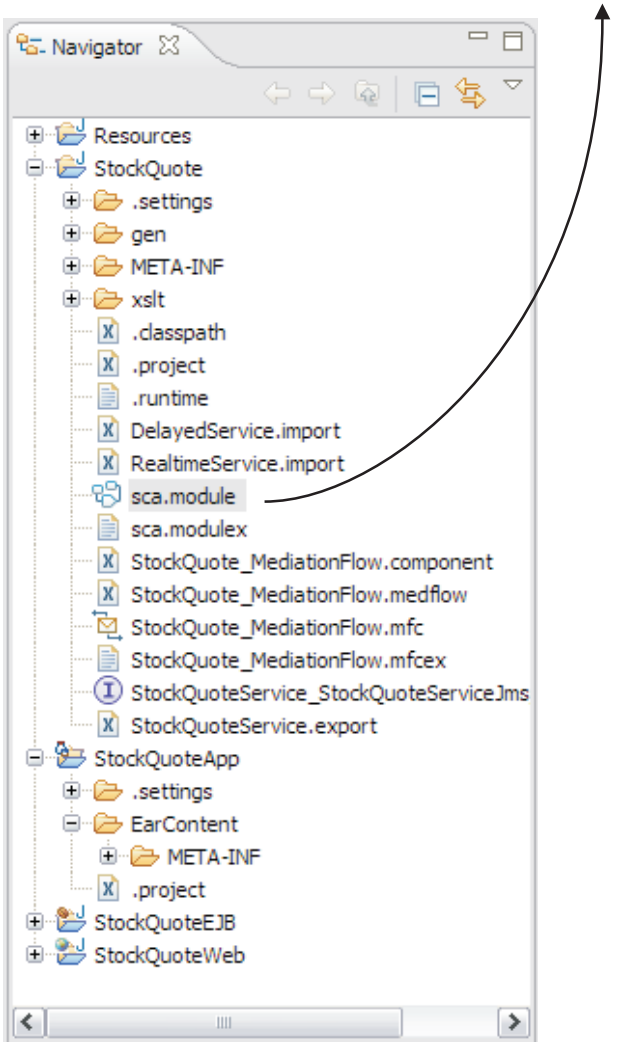


図 42. WebSphere Integration Developer 内のモジュールとモジュール定義の関係

コンポーネント定義

サービス・コンポーネント定義は <SERVICE_NAME>.component というファイルに含まれています。SCA コンポーネントと関連する依存関係を定義して、デプロイ可能なユニットにパッケージ化することができます。

次の図では、サービス・コンポーネント定義をさらに詳細に示しています。各サービス・コンポーネントには、SCA モジュール内での固有名が必要であり、その名前はモジュールのルートからの相対ファイル・パスに一致する必要があります。前のスライドで示したように、サービス・コンポーネント定義は <SERVICE_NAME>.component というファイルに含まれています。

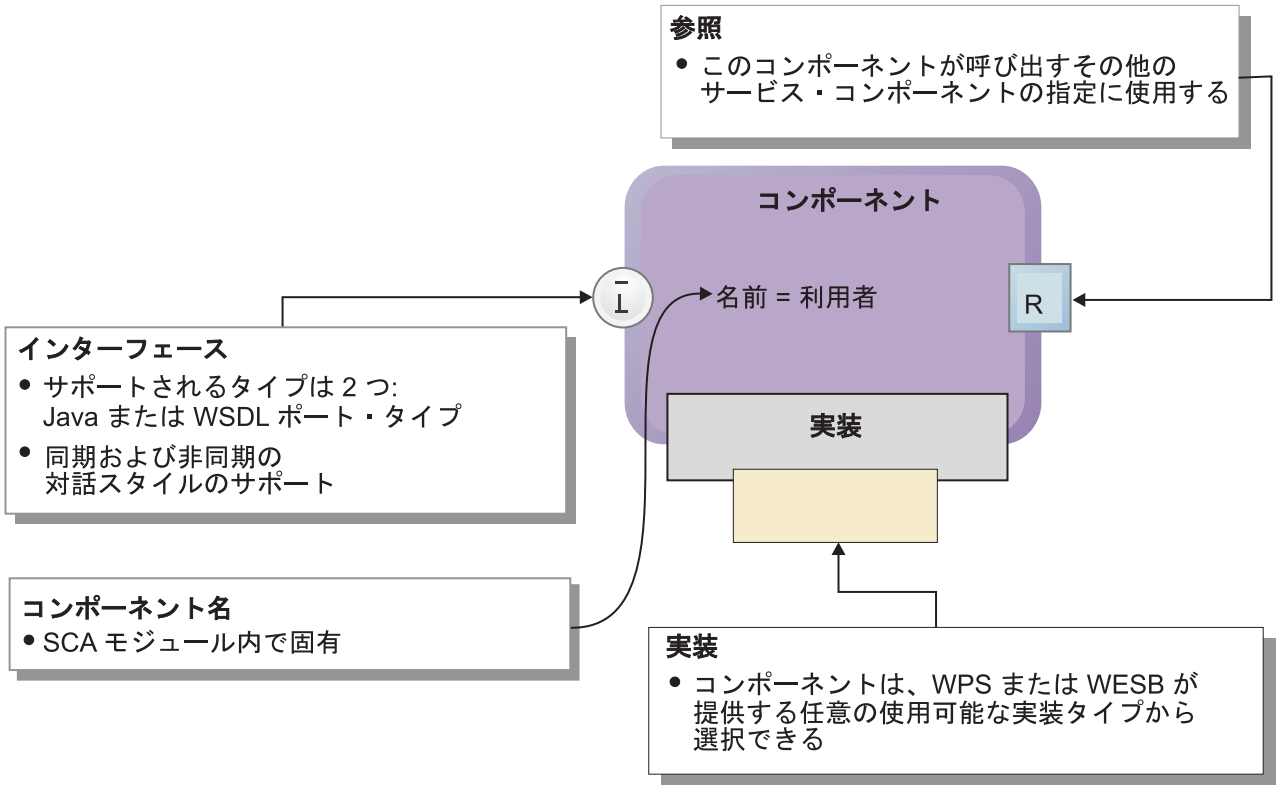


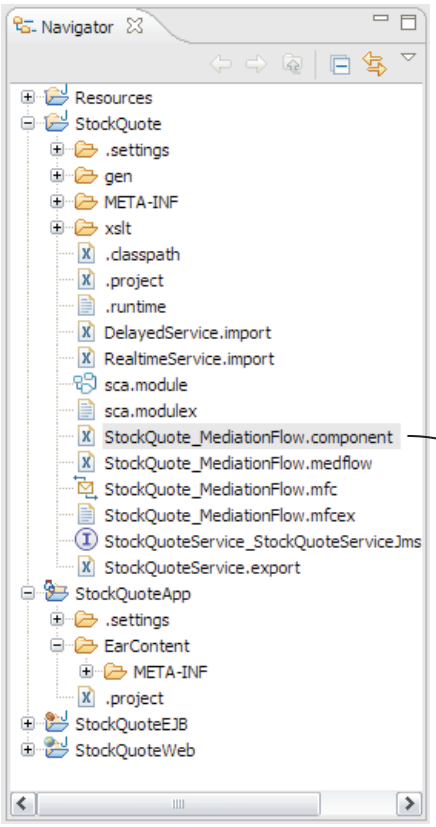
図 43. コンポーネント名、実装、インターフェース、および参照を含むサービス・コンポーネント定義

各サービス・コンポーネントには、SCA モジュール内での固有名が必要であり、その名前はモジュールのルートからの相対ファイル・パスに一致する必要があります。次に、各サービス・コンポーネントには、1 つ以上のインターフェースを関連付けることができます。Java または WSDL ポート・タイプ・インターフェース定義のいずれかを使用できます。サービス・コンポーネントに関連付けられたインターフェースは、サービスを呼び出すクライアントで、同期または非同期対話スタイルをサポートします。

各サービス・コンポーネントは、実装定義によって指定されたさまざまな方法で実装できます。最後に、サービス・コンポーネントは、現行のサービス・モジュールで定義された他のサービス・コンポーネントまたはインポートを呼び出すことがで

きます。この場合、適切な参照を定義して、どのサービスを使用するかを指示する必要があります。多くの場合、このタイプの参照は、サービス・コンポーネント定義でインライン化されていますが、スタンドアロン参照ファイルに入れることもできます。各サービス・コンポーネント定義は、定義されているサービス・コンポーネントによって呼び出される他のサービスへの 0 個以上の参照を持つことができます。

以下に示すのは、StockQuote_MediationFlow コンポーネントの定義を示す例です。WSDL インターフェース、2 つの参照、および実装の定義がコンポーネントに含まれていることが分かります。




```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
        xmlns:ns1="http://Resources/StockQuoteService"
        xmlns:ns2="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
        xmlns:ns3="http://stockquote.samp.sibx.websphere.ibm.com/RealtimeService/"
        xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
        Xmlns:wsdl="http://www.ibm.com/xmlns/prod/websphere/scdl/wsdl/7.0.0"
        DisplayName="StockQuote_MediationFlow" name="StockQuote_MediationFlow">
 <interfaces>
  <interface xsi:type="wsdl:WSDLPortType" portType="ns1:StockQuoteService"/>
 </interfaces>
 <references>
  <reference name="DelayedServicePortTypePartner">
   <interface xsi:type="wsdl:WSDLPortType" portType="ns2:DelayedServicePortType">
    <method name="getQuote"/>
   </interface>
   <wire target="DelayedService"/>
  </reference>
  <reference name="RealtimeServicePortTypePartner">
   <interface xsi:type="wsdl:WSDLPortType" portType="ns3:RealtimeServicePortType">
    <Method name="getQuote"/>
   </interface>
   <wire target="RealtimeService"/>
  </reference>
 </references>
 <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="StockQuote_MediationFlow.mfc"/>
</Scdl:component>
```

図 44. Service Component Definition Language でのコンポーネント定義の例

インポート定義

インポート定義は、<IMPORT_NAME>.import というファイルに含まれています。SCA インポートを使用すると、SCA モジュール内のクライアントが、現行の SCA モジュールの外部にあるサービスにアクセスできます。

サービス・コンポーネントと同様に、インポートには、名前と、インポートが関連付けられた 1 から N 個のインターフェースのセットがあります。インポートには、バインディング属性もあります。これは、外部サービスがどのように現行モジュールにバインドされているかを記述するために使用されます。共通バインディング・タイプを図に示します。

インポートは、SCA モジュール内の特殊なタイプのサービス・コンポーネントと見なすことができます。インポートは、サービス参照のワイヤー定義での有効なターゲットです。すなわち、ターゲット・サービスを呼び出すクライアントにとって、参照がインポートを指すか別のサービス・コンポーネントを指すかに関わらず、クライアント・プログラミング・モデルは同一です。

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:import xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:ns1="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
        xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
        xmlns:webservice="http://www.ibm.com/xmlns/prod/websphere/scdl/webservice/7.0.0"
        xmlns:wsdl="http://www.ibm.com/xmlns/prod/websphere/scdl/wsdl/7.0.0"
        displayName="DelayedService" name="DelayedService">
 <interfaces>
  <interface xsi:type="wsdl:WSDLPortType" portType="ns1:DelayedServicePortType">
   <method name="getQuote"/>
  </interface>
 </interfaces>
 <esbBinding xsi:type="webservice:WebServiceImportBinding"
        endpoint="http://localhost:9080/DelayedService/services/DelayedServiceSOAP"
        port="ns1:DelayedServiceSOAP" service="ns1:DelayedService"/>
</scdl:import>
```

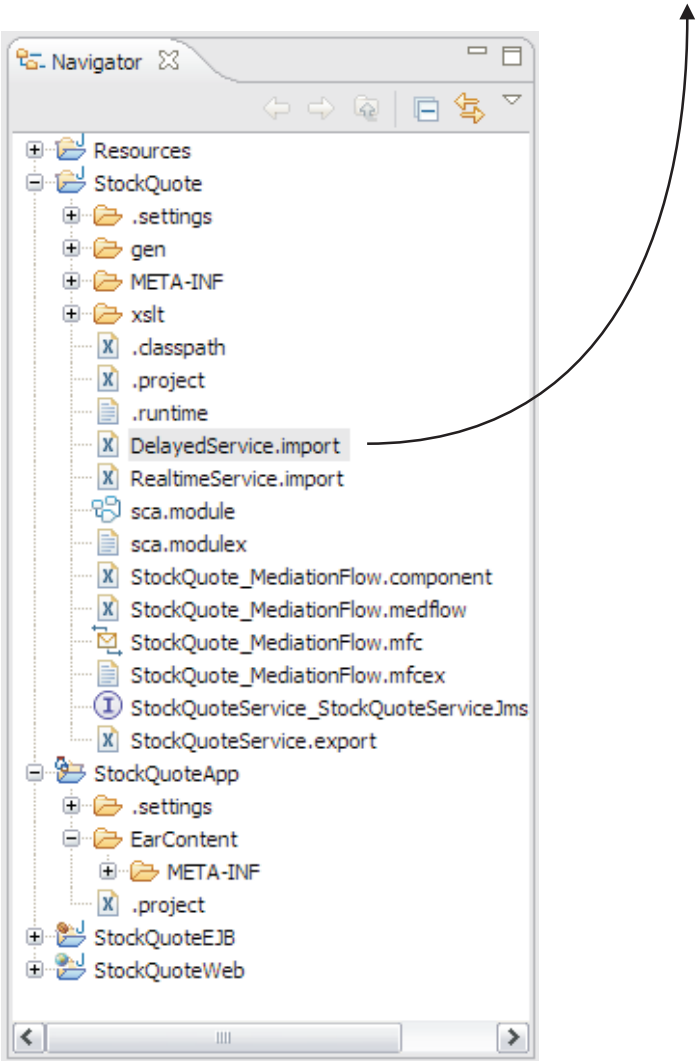
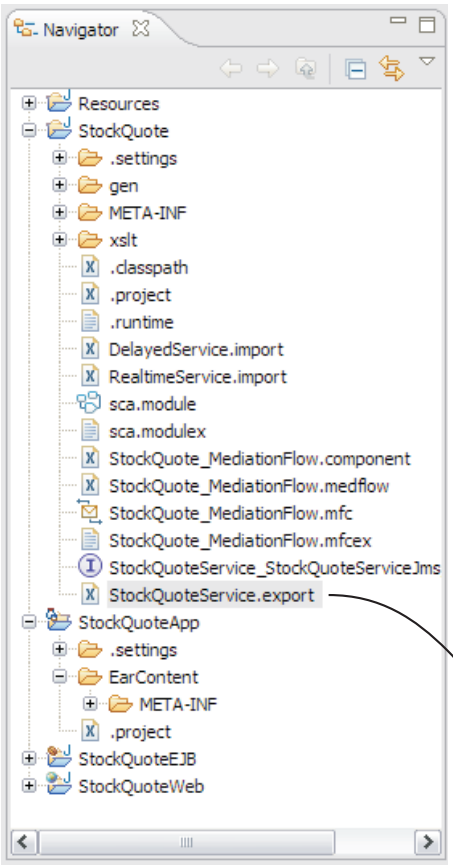


図 45. Service Component Definition Language でのインポート定義の例

エクスポート定義

エクスポート定義は、<EXPORT_NAME>.export というファイルに含まれています。SCA エクスポートは、現行の SCA モジュールの外部にあるクライアントが使用するために、SCA モジュールで定義されたサービス・コンポーネントへのアクセスを提供します。

エクスポート・コンポーネントには、名前とターゲット属性が含まれます。ターゲット属性は、エクスポートされるサービス・コンポーネントを指定します。インポート・コンポーネントと同様に、エクスポートには、サービスがどのように外部にバインドされているかを示すバインディング属性があります。共通バインディング・タイプを図に示します。




```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:export xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:_="http://Resources/StockQuoteService/Binding"
xmlns:ns1="http://Resources/StockQuoteService"
        xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
        xmlns:webservice="http://www.ibm.com/xmlns/prod/websphere/scdl/webservice/7.0.0"
        Xmlns:wsdl="http://www.ibm.com/xmlns/prod/websphere/scdl/wsdl/7.0.0"
        displayName="StockQuoteService" name="StockQuoteService" target="StockQuote_MediationFlow">
 <interfaces>
  <interface xsi:type="wsdl:WSDLPortType" portType="ns1:StockQuoteService">
   <method name="getQuote"/>
  </interface>
 </interfaces>
```

図 46. Service Component Definition Language でのエクスポート定義の例

参照定義

サービス・コンポーネントを呼び出す SCA クライアントおよび非 SCA クライアントでは、呼び出すためにそのサービスへの参照が必要です。参照は、`sca.reference` ファイルでスタンドアロンとして定義するか、サービス構成定義内でインラインで定義することが可能です。

各参照には、クライアント・プログラミング・モデルを使用してクライアントが適切なサービスを検索するために使用される名前があります。その名前のほかに、参照にはインターフェース・エレメントも含まれています。参照の多重度は、どれだ

けの数のワイヤー定義がこの参照をソースとして指定できるかを示します。最後に、ワイヤー定義は、参照を解決するターゲット・サービス・コンポーネントまたはインポートの名前を指定します。

参照を定義するには 2 つの方法があります。1 つ目は、サービス・コンポーネント定義で参照をインライン化する方法です。この方法を使用すると、参照は、参照が組み込まれたサービス・コンポーネントのみで使用可能になります。もう 1 つは、スタンドアロン参照ファイル内に参照定義を組み込む方法です。この方法では、非 SCA クライアントや、モジュール内の別のコンポーネントが参照を使用できます。スタンドアロン参照ファイル内の参照を使用する非 SCA コンポーネントの例としては、特定のサービスを呼び出す機能が必要な JSP などのユーザー・インターフェース・コンポーネントがあります。クライアントがサービス・コンポーネントを呼び出すためには、SCA ランタイムを使用して、呼び出す適切なサービスを検索できるように、参照が必要となります。

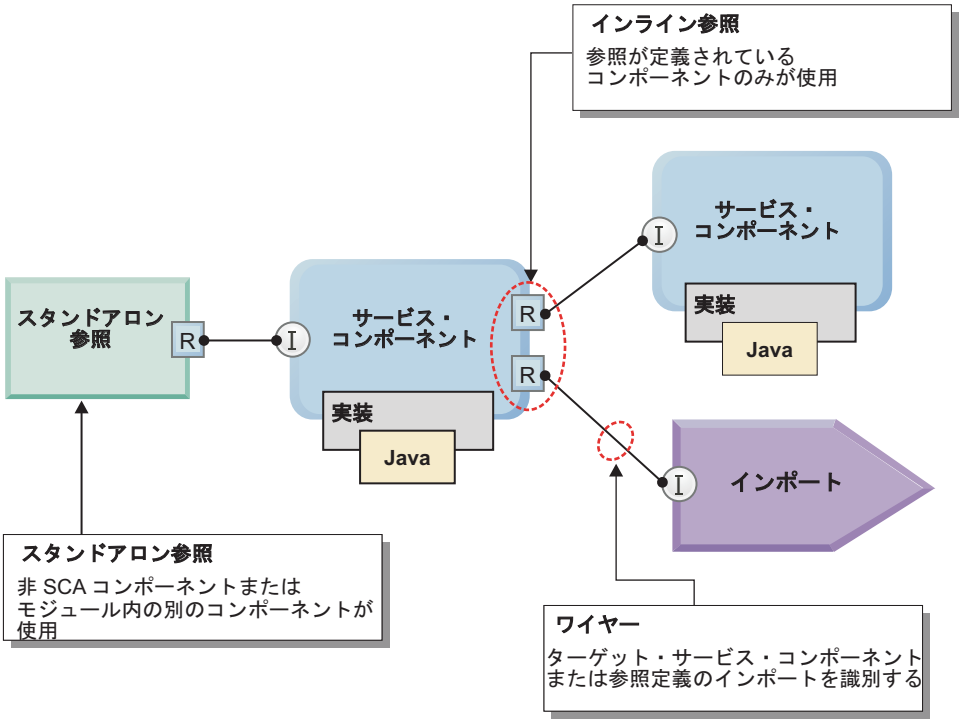


図 47. クライアントが、スタンドアロンまたはインライン参照を使用してのサービス・コンポーネントの呼び出し

以下に示すのは、DelayedServicePortTypePartner および RealtimeServicePortTypePartner という 2 つのインライン参照を含むコンポーネントの定義の例です。WSDL インターフェース、2 つの参照、および実装の定義がコンポーネントに含まれていることが分かります。

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        Xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
        Xmlns:ns1="http://Resources/StockQuoteService"
        xmlns:ns2="http://stockquote.samp.sibx.websphere.ibm.com/DelayedService/"
        xmlns:ns3="http://stockquote.samp.sibx.websphere.ibm.com/RealtimeService/"
        Xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
        Xmlns:wsdl="http://www.ibm.com/xmlns/prod/websphere/scdl/wsdl/7.0.0"
        DisplayName="StockQuote_MediationFlow" name="StockQuote_MediationFlow">
 <Interfaces>
  <interface xsi:type="wsdl:WSDLPortType" portType="ns1:StockQuoteService"/>
 </Interfaces>
 <references>
  <reference name="DelayedServicePortTypePartner">
   <interface xsi:type="wsdl:WSDLPortType" portType="ns2:DelayedServicePortType">
    <method name="getQuote"/>
   </interface>
   <wire target="DelayedService"/>
  </reference>
  <reference name="RealtimeServicePortTypePartner">
   <interface xsi:type="wsdl:WSDLPortType" portType="ns3:RealtimeServicePortType">
    <Method name="getQuote"/>
   </interface>
   <wire target="RealtimeService"/>
  </reference>
 </references>
 <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="StockQuote_MediationFlow.mfc"/>
</Scdl:component>
```

図 48. インライン参照定義の例

SCA プログラミング・モデルの基礎

Service Component Architecture (SCA) プログラミング・モデルの基本となるのは、ソフトウェア・コンポーネント の概念です。コンポーネントとは、あるロジックを実装し、そのロジックをインターフェースを介して他のコンポーネントに使用可能にする単位のことです。コンポーネントには、他のコンポーネントによって使用可能にされたサービスが必要な場合もあります。その場合、コンポーネントは該当するサービスへの*参照* を公開します。

SCA では、すべてのコンポーネントが 1 つ以上のインターフェースを公開する必要があります。 135 ページの図 49 に示すアセンブリー・ダイアグラムには、3 つのコンポーネントがあります。それぞれのコンポーネントのインターフェースは、丸で囲んだ文字 I で表されています。コンポーネントは他のコンポーネントを参照することもできます。参照は、四角で囲んだ文字 R で表されています。参照とインターフェースは、アセンブリー・ダイアグラムでリンクされます。基本的に、統合開発者は、必要なロジックを実装するコンポーネントのインターフェースに参照を接続することによって参照を「解決」します。

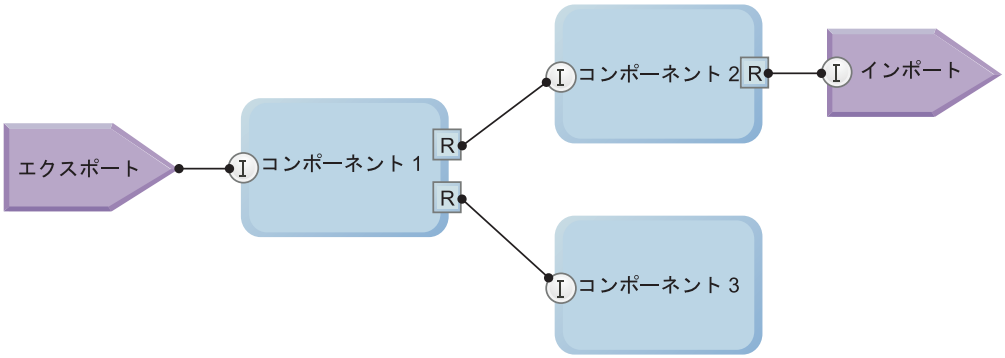


図 49. アセンブリー・ダイアグラム

SCA コンポーネントの呼び出し

呼び出すサービスへのアクセスを提供するために、SCA プログラミング・モデルには *ServiceManager* クラスが組み込まれます。このクラスは、開発者が名前を基準に使用可能なサービスを検索することを可能にします。以下は、サービス検索を説明する典型的な Java コード・フラグメントです。ServiceManager を使用して、BOFactory サービスへの参照を取得します。これはシステム提供のサービスです。

```
//サービス・マネージャー singleton を取得
ServiceManager smgr = new ServiceManager();
//BOFactory サービスにアクセス
BOFactory bof =(BOFactory)
        smgr.locateService("com/ibm/websphere/bo/BOFactory");
```

注: ServiceManager のパッケージは、`com.ibm.websphere.sca` です。

開発者は同様のメカニズムを使用して、*locateService* メソッド内で参照されるサービスの名前を指定することにより、独自のサービスへの参照を取得します。ServiceManager クラスを使用してサービスへの参照を取得した後は、呼び出しプロトコルや実装のタイプに依存しない方法で、そのサービスで使用可能な任意の操作を呼び出せます。

SCA コンポーネントを呼び出すには、以下の 3 種類の呼び出しスタイルを使用できます。

- **同期呼び出し**: この呼び出しスタイルを使用すると、呼び出し側は応答が返されるまで同期的に待機します。このスタイルは従来からの呼び出しメカニズムです。
- **非同期呼び出し**: このメカニズムを使用すると、呼び出し側は、応答が作成されるまで待機することなく、直ちにサービスを呼び出せます。応答を受け取る代わりに、呼び出し側は「チケット」を取得します。このチケットを使用して、後で応答を取得することができます。呼び出し側が応答を取得するために呼び出す特殊な操作は、呼び出し先がこの目的専用に提供する必要があります。
- **コールバックを使用した非同期呼び出し**: この呼び出しスタイルは上記のスタイルと同様ですが、応答のリターンは呼び出し先に委任されます。呼び出し側は、応答の準備ができたときに呼び出し先が呼び出すことのできる特殊な操作 (コールバック操作) を公開する必要があります。

- **据え置き応答を使用した非同期呼び出し**: この呼び出しスタイルでは、クライアントは、サービスを呼び出してから、後で応答を取り込む要求を出すまで処理を続行します。

インポート

場合によっては、レガシー・アプリケーションなどの外部システムやその他の外部実装で使用可能なコンポーネントまたは関数によって、ビジネス・ロジックが提供されることがあります。このような場合、統合開発者は、実装が含まれるコンポーネントに参照を接続するという方法で参照を解決することができません。参照は、該当する外部実装を「指す」コンポーネントに接続する必要があります。これらのコンポーネントはインポート と呼ばれます。インポートを定義するときには、外部サービスにアクセスする方法を、ロケーションおよび呼び出しプロトコルという点で指定する必要があります。

エクスポート

同様に、外部アプリケーションからコンポーネントへのアクセスが必要なこともよくあります。その場合には、コンポーネントをアクセス可能にしなければなりません。アクセス可能にするには、ロジックを「外部世界」に公開する特殊なコンポーネントを使用します。これらのコンポーネントはエクスポート と呼ばれます。エクスポートは、同期的または非同期的に呼び出すことができます。

スタンドアロン参照

WebSphere Process Server では、SCA サービス・モジュールは Java EE EAR ファイルとしてパッケージされます。このファイルには、他の Java EE サブモジュールもいくつか含まれます。WAR ファイルなどの Java EE エレメントは、SCA モジュールと併せてパッケージできます。JSP などの非 SCA 成果物も SCA サービス・モジュールと併せてパッケージできます。このようにパッケージ化すると、これらのエレメントが、スタンドアロン参照と呼ばれる特殊なコンポーネント・タイプを使用した SCA クライアント・プログラミング・モデルによって SCA サービスを呼び出せるようになります。

SCA プログラミング・モデルは厳格な宣言型です。統合開発者は、呼び出しのトランザクション振る舞い、セキュリティー資格認定の伝搬、宣言による同期呼び出しまたは非同期呼び出しの指定などの側面をアセンブリー・ダイアグラムに直接構成できます。これらの修飾子で指定された振る舞いを実装する責任は、開発者ではなく、SCA ランタイムにあります。SCA の宣言に関する柔軟性は、このプログラミング・モデルの最も強力な機能の 1 つです。開発者は、非同期呼び出しメカニズムに対応できるようにするなどの技術的側面に重点を置く代わりに、ビジネス・ロジックの実装に専念できます。これらの側面すべては、SCA ランタイムによって自動的に対処されます。

修飾子

修飾子は、サービス・クライアントとターゲット・サービスとの間の相互作用を管理します。修飾子はサービス・コンポーネント参照、インターフェース、および実装に指定することが可能で、通常は実装外部にあります。

修飾子には以下の各種カテゴリーがあります。

- トランザクション。SCA 呼び出しでトランザクション・コンテキストを処理する方法を指定します。
- アクティビティー・セッション。アクティビティー・セッションの伝搬方法を指定します。
- セキュリティー。アクセス権を指定します。
- 非同期信頼性。非同期メッセージ配信のルールを指定します。

SCA では、これらのサービスの品質 (QoS) 修飾子を宣言によってコンポーネントに適用することができます (プログラミングの必要も、サービス実装コードを変更する必要もありません)。WebSphere Integration Developer を使用して、サービス修飾子を追加できます。通常、QoS 修飾子を適用するのは、ソリューション・デプロイメントを検討する準備が整った時点です。

クライアント・プログラミング・モデル

SCA クライアント・プログラミング・モデルは、サービスを見つけ、データ・オブジェクトを作成し、サービスを呼び出し、呼び出されたコンポーネントで発生した例外を処理するように設計されています。

SCA クライアント・プログラミング・モデルは、クライアント用の 2 つの主要な機能を提供します。プログラミング・モデルでは、クライアントが現行モジュール内でサービスを見つけることができるインターフェースを公開しており、サービスが見つかると、クライアント・プログラミング・モデルは、クライアントがそのサービスで操作を起動する手段を提供します。

クライアントは ServiceManager クラスを使用してサービスを見つけます。希望するサービス検索範囲に応じて、ServiceManager クラスをインスタンス化する方法がいくつかあります。

サービスを見つけるためにクライアントが知っておくべき主要なインターフェースは、`com.ibm.websphere.sca.ServiceManager` です。このインターフェースには、要求されたサービスのサービス実装への参照を返す locateService メソッドが含まれています。locateService メソッドに渡されるストリング・パラメーターは、クライアントが見つけようとするサービスの参照名を表します。 SCA プログラミング・モデル用の Java 資料は、WebSphere Process Server インフォメーション・センターに含まれています。WebSphere Process Server インストールの一環として Java 資料をインストールすることを選択した場合にも含まれます。

クライアントが適切なサービスを見つけると、そのサービスが提供する操作またはメソッドを呼び出すために使用できる 2 つの呼び出しモデルがあります。まず、対話の*動的起動* スタイルがあります。このスタイルの対話の主要なインターフェースは、`com.ibm.websphere.sca.Service` です。このインターフェースの invoke() メソッドは、呼び出そうとする操作の名前を、その操作を呼び出すために必要なパラメーターと共に取り込みます。

```
public Interface MyService {
public String someMethod(String input);

Service myService = (Service) serviceManager.locateService("myService");
DataObject input = ...
DataObject result = (DataObject) myService.invoke("someMethod", input);
```

クライアントは、*静的 (タイプ・セーフな) 起動* メソッドを使用して、サービスに関連付けられた特定の操作を呼び出すこともできます。このタイプの起動は、Java として指定されたインターフェース定義のみで機能します。この状況では、クライアントは locateService() 呼び出しからの戻りを適切なインターフェースにキャストします。続いてそのインターフェースで適切なタイプ・セーフ・メソッド呼び出しを行うことができます。

```
public Interface MyService {
public String someMethod(String input);

MyService myService = (MyService) serviceManager.locateService("myService");
String input = ...
String result = myService.someMethod(input);
```

インターフェース

サービス・コンポーネントには、関連付けられた 1 つ以上のインターフェースがあります。サービス・コンポーネントに関連付けられたインターフェースは、このサービスに関連付けられたビジネス・オペレーションを通知します。

すべてのコンポーネントは、WSDL 型のインターフェースを持ちます。Java コンポーネントのみが Java 型インターフェースをサポートします。コンポーネント、インポート、またはエクスポートに複数のインターフェースがある場合、すべてのインターフェースの型は同じでなければなりません。

```
<?xml version="1.0" encoding="UTF-8"?>
<scdl:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:mfc="http://www.ibm.com/xmlns/prod/websphere/scdl/mfc/7.0.0"
        xmlns:ns1="http://HelloWorldLibrary/HelloWorld" xmlns:ns2="http://HelloService/HelloService"
        xmlns:scdl="http://www.ibm.com/xmlns/prod/websphere/scdl/7.0.0"
        Xmlns:wsdl="http://www.ibm.com/xmlns/prod/websphere/scdl/wsdl/7.0.0"
        displayName="HelloWorldMediation" name="HelloWorldMediation">
 <Interfaces>
  <interface xsi:type="wsdl:WSDLPortType" portType="ns1:HelloWorld">
   <scdl:interfaceQualifier xsi:type="scdl:JoinTransaction" value="true"/>
  </Interface>
 </interfaces>
 <references>
  <reference name="HelloServicePartner">
   <interface xsi:type="wsdl:WSDLPortType" portType="ns2:HelloService"/>
   <scdl:referenceQualifier xsi:type="scdl:SuspendTransaction" value="false"/>
   <scdl:referenceQualifier xsi:type="scdl:Reliability"/>
   <scdl:referenceQualifier xsi:type="scdl:DeliverAsyncAt" value="commit"/>
   <wire target="HelloServiceImport"/>
  </Reference>
 </references>
 <implementation xsi:type="mfc:MediationFlowImplementation" mfcFile="HelloWorldMediation.mfc">
  <scdl:implementationQualifier xsi:type="scdl:Transaction" value="global"/>7
```

図 50. SCDL コンポーネント定義内のインターフェース定義の例

これらのインターフェースは、Java インターフェースとしても WSDL ポート・タイプ・インターフェースとしても指定できます。ただし、同じサービス・コンポーネント定義で Java および WSDL ポート・タイプ・インターフェースを混用することはできません。これらのインターフェースの引数と戻りの型は、単純 Java 型、Java クラス、Service DataObject、または XML スキーマ (WSDL ポート・タイプ・インターフェース用) として指定できます。

実装が同期的であるか非同期的であるかに関わらず、コンポーネントは同期的または非同期的に呼び出すことができます。コンポーネント・インターフェースが同期

形式で定義され、それらの非同期サポートも生成されます。優先対話スタイルを同期または非同期として指定できます。非同期タイプは、完了するのにかなりの時間がかかる可能性のある 1 つ以上の操作を含むインターフェースを、ユーザーに通知します。その結果として、呼び出し側のサービスでは、操作が完了して応答を送信するまで待機する間、トランザクションを開いたままにしないようにする必要があります。対話スタイルは、インターフェース内のすべての操作に適用されます。

インターフェースを持つエクスポートとしてサービスを公開し、インポートの作成に必要なアセンブリー・ダイアグラムにエクスポートをドラッグすることによって、さまざまなモジュールのコンポーネントをワイヤリングすることができます。コンポーネントをワイヤリングするときに、実装、パートナー参照、およびコンポーネントのインターフェースに対して、サービス品質修飾子を指定することもできます。

インポートには、それが関連付けられたリモート・サービスのインターフェースと同じか、そのサブセットであるインターフェースがあり、それらのリモート・サービスを呼び出すことができるようになっています。インポートは、ローカル・コンポーネントとまったく同じようにアプリケーションで使用されます。これにより、場所や実装に関わりなく、すべての機能に対して均一のアセンブリー・モデルが提供されます。インポート・バインディングは、開発時に定義する必要はありません。デプロイメント時に定義できます。

エクスポートには、それが関連付けられたコンポーネントのインターフェースと同じか、そのサブセットであるインターフェースがあり、公開されたサービスを呼び出すことができるようになっています。エクスポートが別のモジュールからアセンブリー・ダイアグラムにドラッグされると、自動的にインポートが作成されます。

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。

始める前に

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。

このタスクについて

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。

手順

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』または『コンポーネントの動的呼び出し』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。

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。モジュールには、アプリケーションが使用するコンポーネント、ライブラリー、およびステージング・モジュールを含めることができます。

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。

WebSphere Process Server は、ビジネス・サービス用モジュールとメディエーション・モジュールをサポートします。モジュールとメディエーション・モジュールはどちらも、Service Component Architecture (SCA) モジュールのタイプです。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。ビジネス・サービス用のモジュールは、ビジネス・プロセスのロジックを実装します。ただし、メディエーション・モジュール内にパッケージ可能なものと同じメディエーション・ロジックをモジュールに格納することもできます。

以降のセクションで、WebSphere Process Server のモジュールを実装および更新する方法を説明します。

コンポーネント

SCA モジュールに含まれるコンポーネントは、再使用可能なビジネス・ロジックをカプセル化する基本ビルディング・ブロックです。サービスを提供および取り込むコンポーネントには、インターフェース、参照、および実装が関連付けられます。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。

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. モジュール内でコンポーネントのインターフェースを定義または識別します。
2. コンポーネントが使用するビジネス・オブジェクトを定義または操作します。
3. コンポーネントをそれぞれのインターフェースを使用して定義または変更します。

   注: 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. オプション: 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. ランタイムにデプロイするエンタープライズ・アーカイブ (EAR) ファイルを作成します。このファイルを作成するには、WebSphere Integration Developer の EAR エクスポート機能または serviceDeploy コマンドのいずれかを使用します。

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『参照』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 Network Deployment 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・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。

始める前に

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。

このタスクについて

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。

手順

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. サービスの呼び出しを実装するクラスを生成します。
4. 生成されたクラスの実装を開発します。
5. コンポーネントのインターフェース、および実装を拡張子が .java のファイルに保管します。

6. 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 6 から 8 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。
7. 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。
8. 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。
9. 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;
 }
}
```

以下のセクションは、StockQuote に関連したクラスの実装です。

```
public class StockQuoteImpl implements StockQuote {

 public float getQuote(String symbol) {


     return 100.0f;
 }
}
```

次のタスク

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。

始める前に

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。

このタスクについて

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働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。

手順

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
       StockQuote sQuote =
      (StockQuote)serviceManager.locateService("stockQuote");
       Ticket ticket =  sQuote.getQuote(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;
    }
}
```

次のタスク

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。

始める前に

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。

このタスクについて

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。

手順

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);
```

呼び出しスタイル

SCA では、同期プログラミング・スタイルおよび非同期プログラミング・スタイルを使用して、サービス・コンポーネントを呼び出すことができます。モジュールを組み合わせて、サービス・コンポーネントおよびモジュール間の非同期チャネルによりシステムの全体的なスループットと柔軟性を高めることのできるような、総合的なソリューションを作成することができます。

コンポーネントは、ビジネス・レベル・インターフェースをアプリケーション・ビジネス・ロジックに公開して、サービスを使用または起動できるようにします。コンポーネントのインターフェースは、呼び出される操作と、渡されるデータ (入力引数、戻り値、および例外など) を定義します。インポートおよびエクスポートには、公開されたサービスを呼び出すことができるように、インターフェースがあります。

すべてのコンポーネントは、WSDL 型のインターフェースを持ちます。Java コンポーネントのみが Java 型インターフェースをサポートします。コンポーネント、インポート、またはエクスポートに複数のインターフェースがある場合、すべてのインターフェースの型は同じでなければなりません。

実装が同期的であるか非同期的であるかに関わらず、コンポーネントは同期的または非同期的に呼び出すことができます。コンポーネント・インターフェースが同期形式で定義され、それらの非同期サポートも生成されます。優先対話スタイルを同期または非同期として指定できます。非同期タイプは、完了するのにかなりの時間がかかる可能性のある 1 つ以上の操作を含むインターフェースを、ユーザーに通知します。その結果として、呼び出し側のサービスでは、操作が完了して応答を送信するまで待機する間、トランザクションを開いたままにしないようにする必要があります。対話スタイルは、インターフェース内のすべての操作に適用されます。

WebSphere Integration Developer のオーサリング時に、コンポーネントのそれぞれが互いを呼び出すために使用する呼び出しスタイルを、明示的に設定することをお勧めします。パフォーマンス分析を実行するか、またはエラー処理方針を作成するときに、少なくとも、アプリケーション全体でどの呼び出しスタイルが使用されているかを知っておく必要があります。トランザクション境界を検討または設定する際には、アプリケーション内の対話を必ず理解しておかなければなりません。ユーザーは、多くの場合、コンポーネント間の呼び出しスタイルの設定または判別が、思っていたほど簡単な作業ではないことが分かって驚くことになります。このセクションでは、実行時にどの呼び出しスタイルが使用されるかを設定または判別する方法を、ご使用のアプリケーションの固有の特性に基づいて説明します。

SCA が提供する呼び出しスタイルは、以下のとおりです。

- 同期
- 片方向操作を使用する非同期
- コールバック付き非同期
- 据え置き応答付き非同期

呼び出しを行うために Java 実装で使用される SCA API は、実行時の呼び出しスタイルを決定します。

- `invoke()`: 同期
- `invokeAsync()`: 非同期
- `invokeAsyncWithCallback()`: 非同期

一般に、あるコンポーネント (ソースまたはクライアント) から別のコンポーネント (ターゲット) への対話について考える場合、使用される呼び出しのタイプは、サービス・クライアントによって決まります。例えば、ソース・コンポーネントが Java™ コンポーネントである場合、ソースからターゲットへの呼び出しスタイルは、invoke()、invokeAsync()、または invokeAsyncWithCallback() などの、実装で使用する特定の SCA 呼び出し API によって決まります。WebSphere Process Server で提供されるその他のコンポーネントのそれぞれに、呼び出しが同期的か非同期的かを決定するために使用するルールのセットがあります。

以下のいくつかのコンポーネント/インポートは、非同期と見なされます。

- 長期実行 BPEL
- ヒューマン・タスク
- MQ/MQ インポート
- JMS/汎用インポート
- JMS/JMS インポート

これらのタイプのコンポーネントまたはインポートへのすべての呼び出しは、非同期呼び出しでなければなりません。呼び出し側コンポーネント (ソース) が対話を同期的に開始した場合、SCA は対話を非同期に切り替えます。

同期呼び出し:

サービス・コンポーネント・インターフェース (SCA) は、常に同期形式で定義されます。同期インターフェースごとに、1 つ以上の非同期インターフェースを生成できます。

サービス・コンポーネントが同期的に呼び出されると、クライアント (コンシューマー) およびサービス・プロバイダーはどちらも同じスレッドで実行されます。WebSphere Process Server 内の呼び出し側コンポーネントは、プロバイダーから応答を受け取るまでブロックされます。

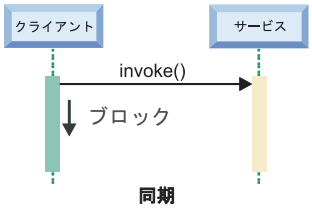


図 51. 同期呼び出し

非同期呼び出し:

WebSphere Process Server は、非同期アプリケーションを開発するための強力なプログラミング・モデルを提供します。 SCA で非同期呼び出しを行うとき、片方向、据え置き応答、およびコールバック付き要求という 3 つのタイプの非同期対話スタイルが使用可能です。3 つのすべてのタイプの非同期呼び出しで、`invokeAsync()` を呼び出すと、SCA ランタイムから即時にクライアントに制御が戻ります。

Java, WebSphere Process Server を使用して非同期プログラムを開発するための、公開された API およびツールに加えて、いくつもの組み込み非同期メッセージング・バインディングおよび組み込み非同期コンポーネントを備えています。

クライアントが後から応答を取り込むには、3 つの異なる方法があります。まず、クライアントは応答を完全に破棄することを選択するか、または void メソッドへの呼び出しであるかを選択できます。この場合、非同期呼び出しは*片方向* であるとされます。もう 1 つは、クライアントが、`invokeAsync()` を呼び出してから、後で応答を取り込む要求を出すまで処理を続行するという方法です。このシナリオは、*据え置き応答* と呼ばれます。最後に、クライアントは*コールバック付き非同期要求* を行うことができます。そのためには、クライアントは先に ServiceCallback インターフェースを実装する必要があります。これにより、`invokeAsync()` を呼び出した後に、SCA ランタイムが、クライアントに応答を渡すための ServiceCallback ハンドラーへのコールバックを提供します。

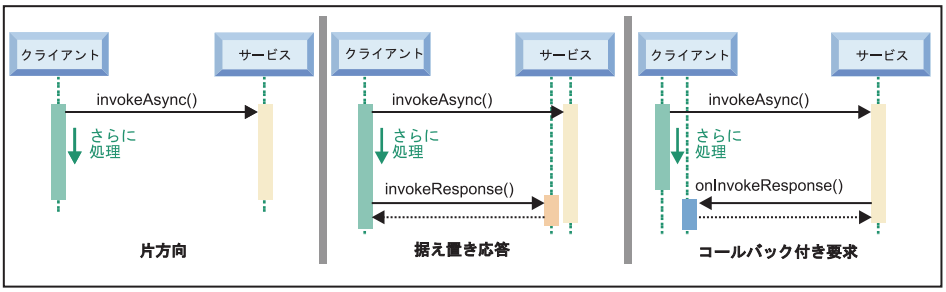


図 52. 非同期呼び出しモデル

SCA インターフェースは、常に同期形式で定義されます。同期インターフェースごとに、1 つ以上の非同期インターフェースを生成できます。クライアントがコールバック・メカニズムを選択した場合、クライアント・コンポーネントは、`<interface name>.Callback.java` クラスを実装する必要があります。このクラスのインターフェースは、クライアントが使用する実際のコンポーネントのインターフェースから派生します。

SCA の対話:

SCA は、モジュールの同期呼び出しおよび非同期呼び出しをサポートします。開発者は、SCA の対話のために、適切なインターフェースおよび呼び出し方式を選択できます。

以下の図に、さまざまなインターフェース・タイプ、サポートされる呼び出し方式とモデル、およびクライアントとサービスの間でデータが渡される方法の要約を示します。

同期呼び出しの場合、データは同じ SCA モジュール内の参照によって渡されますが、非同期呼び出しの場合、データは値によって渡されます。また、表では、インターフェース・タイプに基づいて、どのような場合にタイプ・セーフまたは動的呼び出しを使用できるかについても示します。動的呼び出し方式は、WSDL ポート・タイプまたは Java インターフェースのいずれかで常に使用可能です。ただし、クライアントでタイプ・セーフ呼び出し方式を使用できるようにするには、適切なクライアント参照のインターフェース定義で Java インターフェース・タイプを使用する必要があります。

インターフェース・タイプ	呼び出しモデル				呼び出し方式	
	同期	片方向	据え置き応答	コールバック付き要求	動的	タイプ・セーフ
WSDL ポート・タイプ	●	■	■	■	YES	NO
Java インターフェース	●	■	■	■	YES	YES

● 同じ SCA モジュール内の参照による受け渡しデータ

■ 値による受け渡しデータ

図 53. 呼び出しモデルと、サポートされるデータ受け渡し方式の要約

動的クライアント呼び出し

動的クライアント呼び出しの使用時に同期対話と非同期対話の両方をサポートするために必要な、いくつかの主要なメソッドおよびインターフェースがあります。

表 15. 動的クライアント呼び出し用の主要なメソッドおよびインターフェースの要約

インターフェース	メソッド	説明
サービス	Object invoke(Service String, Object)	同期サービス要求を呼び出すために使用します。
	Ticket invokeAsync(String, Object)	**片方向**または**据え置き応答**非同期サービス要求を呼び出すために使用します。
	Ticket invokeAsyncWithCallback (String, Object)	**コールバック付き要求**の非同期サービス要求を呼び出すために使用します。クライアントは、ServiceCallback インターフェースを実装する必要があります。
	Object invokeResponse(Ticket, long)	**据え置き応答**呼び出しの場合に応答を取得するために使用します。
ServiceCallback	void onInvokeResponse(Ticket, Object, Exception)	**コールバック付き要求**の非同期サービス呼び出しを使用するクライアントは、コールバック・インターフェースを実装する必要があります。

同期呼び出しの例外処理:

サービス・コンポーネントが同期的に呼び出されると、クライアントおよびサービス・プロバイダーはどちらも同じスレッドで実行されます。ターゲットは、応答メッセージまたは例外をクライアントに返すことができます。何も返さない場合 (片方向操作の場合) もあります。結果が例外である場合、それはビジネス例外またはシステム例外のいずれかです。この場合のクライアントは、アプリケーション・コードか、ある種のシステム・コードのいずれかです。

以下に、JType インターフェースを使用して宣言された Java コンポーネントを呼び出すサンプル・クライアントを示します。インターフェースには、以下のように 1 つのメソッドが宣言されています。

```
public interface StockQuote {
 float getQuote(String symbol) throws InvalidSymbolException;
}
```

クライアント・コードは以下のようになります。

```
try {
 float quote = StockQuoteService.getQuote(String symbol);
 } catch (InvalidSymbolException s) {
 System.out.println(This is business exception declared in the Java interface.);
 } catch (ServiceRuntimeException e) {
 System.out.println(Unchecked system exception detected);
}
```

このシナリオでは、最初の例外 InvalidSymbolException は、要求がサービス・プロバイダーに到達したが、サービス・プロバイダーがクライアント入力を認識しないことを示します。サービス・プロバイダーは、次に、指定されたシンボルが無効であることを示すビジネス例外をスローします。このビジネス例外は、メソッド・シグニチャーが宣言した唯一の例外です。

InvalidSymbolException のような JType 例外は、JType 参照を使用するクライアントのみで catch されます。

クライアントは、宣言されたビジネス例外のほかに、システム例外を受け取ることがあります。例えば、証券取引システムで問題が発生した場合、サービスが株価を取得できず、いくつかのチェックなし例外が出されることがあります。このような例外がサービスによってスローされると、ServiceRuntimeException がクライアントに返されます。そこで、クライアントは根本的な原因を判別しようとします。以下のコード断片は、この情報をどのように取得できるかを示しています。

```
try {
 float quote = StockQuoteService.getQuote(String symbol);
 } catch (ServiceRuntimeException e) {
 Throwable t = e.getCause();
 if (t instanceof RemoteException) {
 system.out.println(System ran into RemoteException. Details as follows:  + e.toString());
}
```

非同期呼び出しの例外処理:

サービス・コンポーネントが非同期に呼び出されると、クライアントおよびサービス・プロバイダーは異なるスレッドで実行され、どちらのスレッドでもエラー条件が発生する可能性があります。呼び出し中にクライアントでシステム例外が発生するか、または要求の処理中にサービス・プロバイダーでビジネス例外またはシステム例外が発生する可能性があります。

WebSphere Process Server では、常に、同期インターフェースに対応する非同期インターフェースがあります。

以下に、非同期インターフェースの例を示します。

```
public interface StockQuoteAsync {
public Ticket getQuoteAsync(String arg0);
public Ticket getQuoteAsyncWithCallback(String arg0);
public float getQuoteResponse(Ticket ticket, long timeout) throws InvalidSymbolException;
}
```

以下に、据え置き応答の呼び出しパターンを使用した呼び出しのクライアント・コードを示します。

```
Ticket ticket = stockQuote.getQuoteAsync(symbol);
try {
 quote = stockQuote.getQuoteResponse(ticket, Service.WAIT);
 } catch (InvalidSymbolException s) {
 System.out.println(This is business exception declared in the interface.);
 } catch (ServiceRuntimeException e) {
 System.out.println(Unchecked system exception detected);
}
```

同期呼び出しと同様に、InvalidSymbolException は、要求がサービス・プロバイダーに到達し、シンボルが無効であるというビジネス例外をサービス・プロバイダーがスローしたことを示します。このビジネス例外は、インターフェースが宣言した唯一の例外です。InvalidSymbolException のような JType 例外は、JType 参照を使用するクライアントのみで catch されます。

クライアントは、宣言されたビジネス例外のほかに、メッセージの送信中に発生する接続エラーなどのシステム例外を受け取ることがあります。クライアントは、サービス・スレッド (非同期呼び出しのサービス側のスレッド) で発生したシステム例外を受け取ることはできません。SCA 非同期プログラミング・モデルに従い、ターゲット・コンポーネントで発生した実行時例外は、ソース・コンポーネントに返されません。

非同期例外処理の例外事例

SCA 非同期プログラミング・モデルのルールには、ターゲット・コンポーネントで発生した実行時例外は、ソース・コンポーネントに返されないという 1 つの例外があります。ソース・コンポーネントが Business Process コンポーネントまたは Staff Process である場合、ターゲット・サービス・コンポーネントで発生したシステム例外は呼び出し側に返されます。この機能により、ビジネス・プロセス設計者は、システム例外をモデル化して catch することができ、BPEL クライアントがシステム例外を返したときにエラー・ロジックを実行できます。

異なるサーバー上でのサービス呼び出し時の考慮事項:

サービス指向アーキテクチャーの 1 つのメリットは、利用者が他のサービス・モジュールに存在するサービスを使用できる点です。公平にワークロードのバランスを取るには、アプリケーションをセル内の別々のサーバーにインストールし、それらのアプリケーションを物理的に異なる複数のサーバー上に置くことができます。

WebSphere Process Server のメリットの 1 つは、セル内の複数のサーバーにアプリケーションのワークロードを分散させる機能です。分散させることにより、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し、コンピューティング・リソースの保守容易性を最大化することができます。これは、アプリケーションまたはサービスがサーバー内に 1 つしか存在しないためです。このようにして、サーバー A 上のアプリケーションは、セル内にあるサーバー B にインストールされたサービスを要求します。この方法でサービスを利用するには、サーバー間の通信を構成しておく必要があります。実行する構成のタイプは、呼び出す側のサービス・コンポーネントがサービスを同期的に呼び出すのか、非同期で呼び出すのかにより異なります。

関連トピックには、非同期および同期の呼び出しの両方について、システムの構成方法が記載されています。

サーバーをサービスの非同期呼び出しに構成する:

さまざまなサーバーのサービス・コンポーネントの通信を可能にするためには、これらの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、別のサーバーでサービスを非同期に呼び出すアプリケーションへの通信を可能にする構成について説明します。

始める前に

ここでは、通信を構成しようとしているシステムに WebSphere Process Server がインストールされていて、関連アプリケーションはまだインストールされていないこととします。また、両方のサーバーの構成を検査して変更することができる管理コンソールを使用しているものとします。

このタスクについて

別のシステム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要求するアプリケーションをインストールする前に、これらの要求を通信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・モジュールで構成を行う場合、外部バスとサービス統合バス (SIB) のメディエーションが必要になります。

注: このタスクの前提として、呼び出しサービス・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 に存在しているものとします。

このタスクの前提として、構成に使用する情報は 図 54 に格納されているものとします。

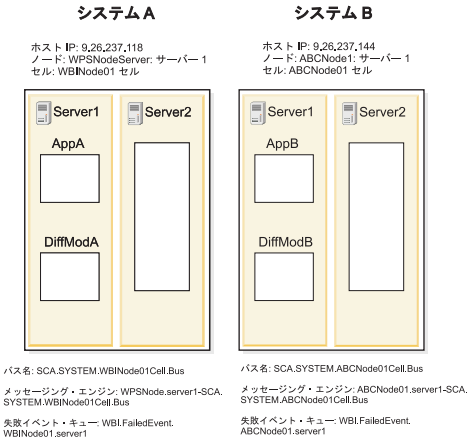


図 54. 別のシステムでサービスを呼び出す

注: 説明を単純にするため、この図では、各セルでこの通信に関係するサーバーだけを示しています。各サーバーは、異なる物理マシン上に存在しています。

手順

1. 通信の対象となる各サーバーに関する情報を収集します。 発信元サーバーとターゲット・サーバーの両方について、以下の情報が必要です。
   - ホストの IP アドレス
   - セル
   - ノード
   - サーバー
   - バス名 (Bus name)
   - メッセージング・エンジン
   - 失敗イベント・キューの名前
2. アプリケーションをインストールします。
3. 他のサーバーをポイントしている各サーバー上に外部バスを作成し、ルーティング定義タイプを「**直接サービス統合バス・リンク**」に設定します。

   詳細については、WebSphere Application Server Network Deployment バージョン 7 インフォメーション・センターのトピック『サービス統合バスの接続による point-to-point メッセージングの使用』を参照してください。

   この例では、システム A の外部バスと SIB メディエーション・リンクの構成は次のようになります。

   ```
   外部バスに接続するサービス統合バスの名前:
   SCA.SYSTEM.ABCNode01Cell.Bus
   外部バスのゲートウェイ・メッセージング・エンジン:
   ABCNode01.server1-SCA.SYSTEM.ABCNode01Cell.Bus
   サービス統合バスのリンク名: TestCrossCell
   ブートストラップ・サービス統合バス・プロバイダーのエンドポイント:
   9.26.237.144:7277:BootstrapBasicMessaging
   ```

   システム B の外部バスと SIB メディエーション・リンクの構成は次のようになります。

   ```
   外部バスに接続するサービス統合バスの名前:
   SCA.SYSTEM.WBINode01Cell.Bus
   外部バスのゲートウェイ・メッセージング・エンジン:
   WPSNode.server1-SCA.SYSTEM.WBINode01Cell.Bus
   サービス統合バスのリンク名: TestCrossCell
   ブートストラップ・サービス統合バス・プロバイダーのエンドポイント:
   9.26.237.118:7276:BootstrapBasicMessaging
   ```

   重要: ブートストラップのポート番号は、SIB エンドポイント・アドレス・ポートの番号になります。セキュリティー設定を有効にした場合、保護された 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。
4. サーバーを再始動して、SIB メディエーション・リンクを同期させます。

   次のようなメッセージが表示されます。

   ```
   [9/25/09 8:04:23:406 CDT] 00000034 SibMessage I [:] CWSIT0032I:
   The service integration bus link TestCrossCell from messaging engine
   WPSNode01.server1-SCA.SYSTEM.WPSNode01Cell.Bus in bus SCA.SYSTEM.WPSNode01Cell.Bus
   to messaging engine ABCNode01.server1-SCA.SYSTEM. ABCNode01Cell.Bus in bus SCA.SYSTEM.
   ABCNode01Cell.Bus started.
   ```
5. 各サービス・モジュールの宛先を表示します。
6. 他のシステムのターゲットに接続する必要がある呼び出しサービス・モジュールの宛先のデフォルト転送パスを変更します。

「**アプリケーション**」>「**SCA モジュール**」を選択し、対象のモジュールを選択して「**SCA システム・バスの宛先**」 をクリックします。

接続の宛先名には、importlink を指定します。例えば、システム A の宛先の場合、sca/AppA/importlink/test/sca/cros/simple/custinfo/CustomerInfo になります。宛先名の先頭に外部バス名を追加して、パスを変更します。この例では、第 2 システムの外部バス名は SCA.SYSTEM.ABCNode01Cell.Bus になります。 結果は、以下のようになります。

```
SCA.SYSTEM.ABCNode01Cell.Bus:sca/AppA/importlink/
test/sca/cros/simple/custinfo/CustomerInfo
```

7. オプション: システムのセキュリティー設定を有効にした場合は、外部バスに送信者のロールを追加します。 オペレーティング・システムのコマンド・プロンプトを使用して、両方のシステムの各アプリケーションをユーザーが使用するように定義する必要があります。 ロールを追加するコマンドを以下に示します。

```
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のバス名です。

foreignBusName
ユーザーを追加する外部バスです。

userName
外部バスを追加するユーザー ID です。

次のタスク

アプリケーションを開始します。

サーバーをサービスの同期呼び出しに構成する:

サービス・コンポーネントから別のサービス・コンポーネントを同期的に呼び出す場合、ターゲット・サービスを実行しているシステムをポイントするように呼び出しサービス・コンポーネントを構成し、ターゲット・サービスと呼び出しサービス・コンポーネント間で実行結果を通信できるようにする必要があります。

始める前に

ここでは、通信を構成しようとしているシステムに WebSphere Process Server がインストールされていて、関連アプリケーションはまだインストールされていないこととします。また、両方のサーバーの構成を検査して変更することができる管理コンソールを使用しているものとします。

このタスクについて

別のサービスを同期的に呼び出すサービス・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) の名前が呼び出しシステム上の JNDI 名に対して構成されている場合のみ、ターゲットと通信することができます。

注: このタスクの前提として、呼び出しサービス・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 に存在しているものとします。

このタスクの前提として、構成に使用する情報は 図 55 に格納されているものとします。

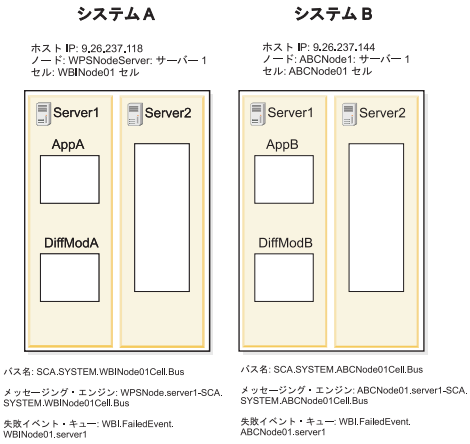


図 55. 別のシステムでサービスを呼び出す

注: 説明を単純にするため、この図では、各セルでこの通信に関係するサーバーだけを示しています。各サーバーは、異なる物理マシン上に存在しています。

手順

1. 各サーバーにアプリケーションをインストールします。
2. ターゲット・システム上のエクスポートを指している呼び出しシステム (この例ではシステム A) 上で、新しい名前空間バインディングを作成します。

   「**名前空間バインディング (Name Space Bindings)**」 パネルでセルの範囲を選択し、「**適用**」をクリックします。セルの範囲を変更し、画面の 「**新規**」 をクリックして新しいバインディングを作成します。

   ウィザードで、以下の項目を指定します (それぞれの値は、この構成例に対する適切な値です)。

   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 には、固有のストリングを指定します。この例では、`sca_import_test_sca_cross_simple_custinfo_CustomerInfo` になります。
      - 名前空間における名前は、ターゲット・システムで呼び出す Enterprise Java Bean (EJB) の JNDI 名になります。以下に例を示します。

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

        これにより、ターゲット・システムのエクスポート・インターフェース名が指定されます。
      - 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 アドレスとポート番号です。

corbaname:iiop:*host*:*port*/
NameServiceServerRoot#<*package.qualified.interface*>

注: WebSphere Process Server の場合、ポートは BOOTSTRAP_ADDRESS です。

以下に例を示します。

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

項目をすべて入力したら「**次へ**」をクリックし、「**要約**」 ページの値を確認します。 確認したら、 「**終了**」をクリックします。

画面に新しいバインディングが表示されます。

3. 「**保管**」をクリックして、変更を保存します。
4. クロス・セル構成が同じ名前の同じホスト上のサーバーから構成されている状態で、NameNotFoundException 例外による JNDI ルックアップ障害が発生した場合、システム・プロパティーを設定する必要があります。

   『アプリケーション・アクセスの問題』の副見出し『*同じホスト上で実行されている同じ名前の 2 つのサーバーが相互運用に使用されている*』の説明に従ってください。

次のタスク

アプリケーションを開始します。これで、システム A のサービス・コンポーネントからシステム B のサービスを同期的に呼び出すことができます。

修飾子

修飾子は SCA で重要な役割を果たします。修飾子によって、開発者が WebSphere Process Server ランタイムにサービス品質要件を課すことができるようになるためです。

SCA で使用可能な修飾子には、いくつかのカテゴリーがあります。修飾子のカテゴリーは、トランザクション、アクティビティー・セッション、セキュリティー、および非同期信頼性です。

各 SCA 修飾子は、その修飾子を指定できるコンポーネントの SCDL 定義内で、特定のスコープを持っています。例えば、一部の修飾子は参照レベルで指定できますが、その他の修飾子はインターフェース・レベルまたは実装レベルのみで有効な場合があります。次の表に、使用可能な各種の修飾子と、それぞれの有効なスコープを示します。修飾子は、それらが提供するサービス品質のタイプ (トランザクション、セキュリティーなど) によってソートされています。

表 16. 修飾子の要約

タイプ	修飾子	有効範囲	説明
トランザクション	transaction	実装	**global** - コンポーネントを実行するには、グローバル・トランザクションが存在していなければなりません。 **local** - コンポーネントを実行するには、グローバル・トランザクションが存在していてはなりません。 **any** - コンポーネントはトランザクションの状態の影響を受けません。 **local application** - コンポーネントの処理は、アプリケーションが管理する WebSphere ローカル・トランザクション包含の内部で行われます。
	joinTransaction	インターフェース	**true** - ホスティング・コンテナーはクライアント・トランザクションに参加します。 **false** (デフォルト) - ホスティング・コンテナーは、クライアント・トランザクションに参加しません。
	suspendTransaction	参照	**true** - ターゲット・コンポーネントの同期呼び出しは、クライアント・グローバル・トランザクション内で実行されません。 **false** - ターゲット・コンポーネントの同期呼び出しは、クライアント・グローバル・トランザクション内で実行されます。
	deliverAsyncAt	参照	**call** - ターゲット・サービスの非同期呼び出しは、直ちに行われます。 **commit** - ターゲット・サービスの非同期呼び出しは、グローバル・トランザクションの一部として行われます。

表 16. 修飾子の要約 (続き)

タイプ	修飾子	有効範囲	説明
非同期応答	reliability	参照	非同期メッセージ配信のサービス品質レベルを指定します。信頼性の値には、**bestEffort** または **assured** のいずれかを使用できます。
	requestExpiration	参照	非同期要求が配信されなかった場合に非同期要求が破棄されるまでの時間の長さ (ミリ秒) を指定します。
	responseExpiration	参照	要求を送信する時刻から応答またはコールバックを受信する時刻までの期間 (ミリ秒) を指定します。
セキュリティー	securityIdentity	実装	**permission** は、実行時に実装の実行に使用される ID の論理名を指定します。
	securityPermission	インターフェース (複数可)、メソッド	呼び出し側 ID では、インターフェースまたはメソッドを実行する権限を持つためには、この修飾子から指定されたロールが必要です。
アクティビティー・セッション	activitySession	実装	**true** - このコンポーネントを実行するには、ActivitySession が確立されている必要があります。 **false** - コンポーネントはアクティビティー・セッションがない場合も実行されます。**any** - コンポーネントは、ActivitySession が存在するかどうかに依存しません。
	joinActivitySession	インターフェース	**true** - ホスティング・コンテナーはクライアント ActivitySession に参加します。 **false** - ホスティング・コンテナーはクライアント ActivitySession に参加しません。
	suspendActivitySession	参照	**true** - ターゲット・コンポーネントのメソッドは、どのクライアント ActivitySession でもその一部として実行されません。 **false** - ターゲット・コンポーネントのメソッドは、任意のクライアント ActivitySession の一部として実行されます。

トランザクション修飾子

トランザクション修飾子によって、開発者は、特定のトランザクション環境に対して SCA モジュール内のサービス・コンポーネントを要求できます。以下に、これらの修飾子の要約を示します。

transaction

transaction 修飾子は、サービス・コンポーネントの実装スコープで設定されます。この修飾子は、「global」、「local」(デフォルト)、または「any」のいずれかに設定できます。global に設定された場合、コンポーネントはグローバル・トランザクションのコンテキスト内で実行されます。呼び出しにグローバル・トランザクションが存在する場合、コンポーネントは、このグローバル・トランザクション・スコープに追加されます。local に設定された場合、コンポーネントはローカル・トランザクションのコンテキスト内で実行されます。最後に、値が any に設定された場合、グローバル・トランザクションが存在すれば、コンポーネントは現行のグローバル・トランザクション・スコープに加わります。ただし、グローバル・トランザクションが存在しなければ、コンポーネントはローカル・トランザクションのコンテキスト内で実行されます。

joinTransaction

joinTransaction 修飾子は、サービス・コンポーネントのインターフェース・スコープで設定されます。この修飾子は、true または false のいずれかに設定できます (false がデフォルト)。true に設定された場合、ランタイムがインターフェース境界でグローバル・トランザクション (存在する場合) を中断しないことを指示します。false に設定された場合、ランタイムがインターフェース境界でグローバル・トランザクション (存在する場合) を中断することを指示します。インターフェースで joinTransaction トランザクション修飾子を公開すると、アセンブラーおよびデプロイヤーが、アセンブルされたアプリケーションを要求どおりに動作させるために使用できるメタデータが提供されます。ターゲット・コンポーネントが、伝搬されたトランザクションと統合するかどうかの判断は、動的ランタイムだけでなく、アセンブラーおよびデプロイヤーも担当して行います。

suspendTransaction

suspendTransaction 修飾子は、サービス・コンポーネントの参照レベルで設定され、参照に関連付けられたターゲット・サービスを呼び出す前に、グローバル・トランザクションを中断すべきかどうかを示します。この修飾子は、true または false (デフォルト) のいずれかに設定できます。

deliveryAsyncAt

deliveryAsyncAt 修飾子は、suspendTransaction 修飾子と類似しています。ただし、suspendTransaction の場合のような同期タイプではなく、非同期対話に使用されます。deliverAsyncAt 修飾子の値には、call (デフォルト) または commit を使用できます。call に設定された場合、呼び出しが行われたときに即時に非同期対話のメッセージをキューへコミットするように、ランタイムに対して指示します。commit という値は、現在の作業単位に関連付けられたトランザクションの一環として、メッセージをキューにコミットするように指示します。

非同期応答修飾子

非同期応答のサービス品質を指示するために使用可能な 3 つの修飾子があります。非同期応答修飾子はそれぞれ、参照スコープで指定されます。以下に、非同期応答修飾子の要約を示します。

reliability
reliability 修飾子は、非同期メッセージ配信のサービス品質レベルを指定するために使用します。reliability は、bestEffort または assured (デフォルト) のいずれかに設定できます。

requestExpiration
requestExpiration 修飾子は、非同期要求がまだ配信されていない場合に、ランタイムが非同期要求を続行する時間の長さを指定するために使用します。この修飾子に指定された時間 (ミリ秒単位で指定) が経過すると、この要求は破棄されます。

responseExpiration
responseExpiration 修飾子は、ランタイムが非同期応答を保存しなければならない時間、またはコールバックを提供しなければならない時間の長さを指定するために使用します。この修飾子の値はミリ秒単位で指定します。

セキュリティー修飾子

セキュリティーに関係するサービス品質を指示するために使用可能な 2 つの修飾子があります。以下に、これらの修飾子の要約を示します。

securityIdentity
securityIdentity 修飾子は、実行時にサービス・コンポーネントの実装の実行に使用されるセキュリティー ID を指定するために使用します。この修飾子は、サービス・コンポーネントの実装スコープに配置する必要があり、指定される値は、コンポーネントの実行に使用される ID の論理名と一致する必要があります。

securityPermission
securityPermission 修飾子は、インターフェース・レベル (インターフェースを含む) またはメソッド・レベルで指定されます。この修飾子の値は、このサービスの呼び出し側が、このサービスを呼び出すために指定されたロールを持っていなければならないことを示します。

securityPermission と securityIndentity の両方で、これらの修飾子の基礎となる実装は、既存の Java EE の概念に基づいています。

アクティビティー・セッション修飾子

一連のアクティビティー・セッション修飾子は、これまでに照会したトランザクション修飾子と類似しています。ActivitySession サービスは、グローバル・トランザクションと比較したときに代替作業単位を提供できる WebSphere プログラミング・モデル拡張機能です。実際、アクティビティー・セッション・コンテキストは、グローバル・トランザクションよりも長く存続することができ、グローバル・トランザクションを含むことも可能です。以下に、アクティビティー・セッション修飾子の要約を示します。

joinActivitySession

joinActivitySession 修飾子は、インターフェース・レベルで設定され、コンポーネントがクライアント呼び出し側のアクティビティー・セッションに参加するかどうかを指示します。この修飾子には、true および false (デフォルト) の 2 つの値があります。true に設定された場合、コンポーネントの起動時にランタイムがアクティビティー・セッション (存在する場合) を中断しないことを指示します。false に設定された場合、コンポーネントを起動する前にアクティビティー・セッションを中断することを指示します。

activitySession

activitySession 修飾子は実装レベルで指定され、関連付けられたサービス・コンポーネントを実行するために、アクティビティー・セッションが存在していなければならないか、存在していてはならないかを指示します。この修飾子は、「true」、「false」、または「any」(デフォルト) のいずれかに設定できます。true に設定された場合、コンポーネントがアクティビティー・セッションの一部として実行されることを指示します。false に設定された場合、コンポーネントをアクティビティー・セッションの一部として実行しません。このことは、コンポーネントに対して指定されたインターフェースでも joinActivitySession を false に設定する必要があることを意味します。最後に、この修飾子が any に設定された場合、コンポーネントは、アクティビティー・セッションが存在すればアクティビティー・セッションの一部として実行され、存在しなければアクティビティー・セッションの一部として実行されません。

suspendActivitySession

suspendActivitySession 修飾子は参照レベルで設定され、参照に関連付けられたターゲット・サービスが、呼び出し側アクティビティー・セッションの一部として呼び出されるかどうかを指示します。true に設定された場合、アクティビティー・セッションは中断され、ターゲット・コンポーネント上のメソッドは、クライアント・アクティビティー・セッションの一部として実行されません。false (デフォルト) に設定された場合、アクティビティー・セッションは中断されず、ターゲット・コンポーネント上のメソッドは、クライアント ActivitySession の一部として実行されます。

SCA プログラミング手法

このセクションでは、Service Component Architecture プログラミング手法の例を示します。

Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール

生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

基本の型およびクラス

ランタイムは、サービス・データ・オブジェクトと基本の Java 型およびクラス間で単純な変換を実行します。基本の型およびクラスは以下のとおりです。

- Char または java.lang.Character
- Boolean
- Java.lang.Boolean
- Byte または java.lang.Byte
- Short または java.lang.Short
- Int または java.lang.Integer
- Long または java.lang.Long
- Float または java.lang.Float
- Double または java.lang.Double
- Java.lang.String
- Java.math.BigInteger
- Java.math.BigDecimal
- Java.util.Calendar
- Java.util.Date
- Java.xml.namespace.QName
- Java.net.URI
- Byte[]

ユーザー定義の Java クラスおよび配列

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス `com.ibm.xsd.Customer` が SDO に変換されると、URI は `http://xsd.ibm.com` であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

SDO から Java 型に変換する場合、ランタイムは URI を反転させてパッケージ名を生成し、SDO の型と同じ型の名前を生成します。例えば、型 Customer と、URI `http://xsd.ibm.com` を持つデータ・オブジェクトは、Java パッケージ `com.ibm.xsd.Customer` のインスタンスを生成します。次にランタイムは、SDO のプロパティーから値を抽出し、それらのプロパティーを Java クラスのインスタンス内のフィールドに割り当てます。

Java クラスがユーザー定義のインターフェースである場合、生成されるコードをオーバーライドして、ランタイムがインスタンス化できる具象クラスを作成する必要があります。ランタイムが具象クラスを作成できない場合、例外が発生します。

Java.lang.Object

Java 型が java.lang.Object の場合、生成される型は xsd:anyType です。モジュールは、すべての SDO で、このインターフェースを呼び出すことができます。ランタイムは、具象クラスを見つけることができる場合は、ユーザー定義の Java クラスと配列の場合と同様にその具象クラスをインスタンス化しようとします。見つからない場合、ランタイムは SDO を Java インターフェースに渡します。

メソッドが java.lang.Object 型を戻しても、ランタイムは、メソッドが具象型を戻す場合のみ、SDO に変換します。ランタイムは、ユーザー定義の Java クラスおよび配列から SDO への変換でも同様の変換を使用します (次の段落を参照)。

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス `com.ibm.xsd.Customer` が SDO に変換されると、URI は `http://xsd.ibm.com` であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

いずれの場合でも、ランタイムが変換を完了できない場合は例外が発生します。

Java.util コンテナー・クラス

Vector、HashMap、HashSet などのような具象 Java コンテナー・クラスに変換する場合、ランタイムは該当するコンテナー・クラスをインスタンス化します。ランタイムは、ユーザー定義の Java クラスおよび配列で使用したのと同様の方法で、コンテナー・クラスにデータを取り込みます。ランタイムが具象 Java クラスを見つけられない場合、ランタイムはコンテナー・クラスに SDO を取り込みます。

具象 Java コンテナー・クラスから SDO に変換する場合、ランタイムは、『Java から XML への変換』に示される、生成されたスキーマを使用します。

Java.util インターフェース

`java.util` パッケージ内の特定のコンテナー・インターフェースの場合、ランタイムは以下の具象クラスをインスタンス化します。

表 17. WSDL 型から Java クラスへの変換

インターフェース	デフォルトの具象クラス
コレクション	HashSet
マップ	HashMap
リスト	ArrayList
セット	HashSet

サービス・データ・オブジェクトから Java への変換のオーバーライド

場合によっては、システムが作成するサービス・データ・オブジェクト (SDO) と Java 型オブジェクト間の変換では、要件に合わないことがあります。デフォルトの実装を独自の実装に置き換えるには、以下の手順を実行します。

始める前に

WebSphere Integration Developer または genMapper コマンドを使用して WSDL から Java 型への変換を生成していることを確認します。

このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、WSDL 型から Java 型にマップする生成済みコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

手順

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントの実装を含むファイル名は変更しないでください。

例

以下は、置き換え対象の生成されたコンポーネントの例です。

```
 private Object datatojava_get_customerAcct(DataObject myCustomerID,
    String integer)
{

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


  return SDOJavaObjectMediator.data2Java(customerID, integer) ;

}
```

次のタスク

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤリングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

生成される Service Component Architecture 実装のオーバーライド

場合によっては、システムが作成した Java コードとサービス・データ・オブジェクト (SDO) 間の変換では、要件に合わないことがあります。デフォルトの Service Component Architecture (SCA) の実装を独自の実装に置き換えるには、以下の手順を実行します。

始める前に

WebSphere Integration Developer または genMapper コマンドを使用して Java から Web サービス記述言語 (WSDL) 型への変換を生成していることを確認します。

このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、Java 型から WSDL 型にマップする生成されたコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

手順

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントの実装を含むファイル名は変更しないでください。

例

以下は、置き換え対象の生成されたコンポーネントの例です。

```
private DataObject javatodata_setAccount_output(Object myAccount) {

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


 return SDOJavaObjectMediator.java2Data(myAccount);


}
```

次のタスク

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤリングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

非 SCA エクスポート・バインディングからのプロトコル・ヘッダー伝搬

コンテキスト・サービスは、コンテキスト (JMS ヘッダーなどのプロトコル・ヘッダーや、アカウント ID などのユーザー・コンテキストを含む) を Service Component Architecture (SCA) 呼び出しパスに沿って伝搬します。コンテキスト・サービスは、一連の API と構成可能な設定を提供します。

コンテキスト・サービス伝搬が双方向の場合、現行のコンテキストは常に応答コンテキストによって上書きされます。SCA コンポーネントから別の SCA コンポーネントへの呼び出しを実行している場合、応答には異なるコンテキストが含まれます。サービス・コンポーネントには着信コンテキストがありますが、別のサービスを呼び出すと、元の発信コンテキストがもう一方のサービスによって上書きされます。応答コンテキストは新しいコンテキストになります。

コンテキスト・サービス伝搬が片方向の場合、元のコンテキストがそのまま維持されます。

コンテキスト・サービスのライフ・サイクルは、呼び出しに関連付けられます。要求には関連コンテキストがあり、そのコンテキストのライフ・サイクルがその特定の要求の処理にバインドされます。要求が処理を終了すると、そのコンテキストのライフ・サイクルが終了します。

実行時間が短い Business Process Execution Language (BPEL) プロセスの場合、応答コンテキストが要求コンテキストを上書きします。最初の要求から応答コンテキストが取り戻されて、次の要求にプッシュされます。長期間実行される BPEL プロセスの場合は、応答コンテキストが BPEL フレームワークによって廃棄されます。元のコンテキストは保管され、別の発呼を行う際にはそのコンテキストが使用されます。

コンテキスト・サービスには、バインディング振る舞いを指定する構成可能ルールと表があります。詳しくは、『参照』セクションに記載されている生成済み API および SPI の資料を参照してください。コンテキスト・サービスは、WebSphere® Integration Developer の開発中に、インポートおよびエクスポート・プロパティーに設定できます。詳しくは、WebSphere Integration Developer インフォメーション・センターのインポートおよびエクスポート・バインディングに関する情報を参照してください。

ビジネス・オブジェクトのプログラミング

ビジネス・オブジェクトは、顧客や送り状などのアプリケーション・データのコンテナーです。データは、ビジネス・オブジェクトを経由してコンポーネント間で交換されます。ビジネス・オブジェクトの基礎となる構造は、XML スキーマ定義 (XSD) であり、ビジネス・オブジェクトへのプログラマチック・アクセスが WebSphere のビジネス・オブジェクト・インターフェースを通じて提供されています。ビジネス・オブジェクトのこれらの特徴や、構造表現、プログラマチック・インターフェース、および Service Component Architecture (SCA) 内での振る舞いと操作は、まとめてビジネス・オブジェクト・フレームワークと呼ばれます。このフレームワークにより、ソリューションでビジネス・データを記述して配信するための、強力で一貫性のある手段が提供されます。

このガイドでは、いくつかの機能でスキーマ構成を処理する場合の問題領域の説明を含め、ビジネス・オブジェクトのプログラミングについての情報を提供します。ビジネス・オブジェクトの定義方法、ビジネス・オブジェクトの開発ガイドライン、およびビジネス・オブジェクト・プログラミング API の使用方法について詳しくは、『関連情報』セクションの記事を参照してください。

関連情報

Web Services Description Language (WSDL) 1.1

Introduction to Service Data Objects

Examining business objects in WebSphere Process Server

プログラミング・モデル

ビジネス・オブジェクト・プログラミング・モデルのセクションでは、IBM ビジネス・オブジェクト・フレームワーク内部に基本タイプのデータがどのようにカプセル化されているかを説明します。ビジネス・オブジェクトの作成と操作を容易にするため、ビジネス・オブジェクト・フレームワークは一連の Java サービスを提供することによってサービス・データ・オブジェクト仕様を拡張します。

IBM ビジネス・オブジェクト・フレームワークの操作

ビジネス・オブジェクト・フレームワークは、ランタイム内のデータが、どのようにアプリケーションによってモデル化され、ランタイムに統合され、メモリーで表現されるかを記述します。

表 18 に、ビジネス・オブジェクト・フレームワークでは基本タイプのデータがどのように実装されているかを記載します。

表 18. データ抽象化および対応する実装

データ抽象化	実装	説明
インスタンス・データ	ビジネス・オブジェクト	ビジネス・オブジェクトは、ビジネス・エンティティーを表す、つまりリテラル・メッセージ定義を文書化するための基本メカニズムであり、スカラー・プロパティーを持つ単純な基本オブジェクトから、大規模で複雑なオブジェクトの階層またはグラフまでのすべてを使用可能にします。ビジネス・オブジェクトは、SDO DataObject 概念から直接導き出されます。

表 18. データ抽象化および対応する実装 (続き)

データ抽象化	実装	説明
インスタンス・メタデータ	ビジネス・グラフ	ビジネス・グラフは、ビジネス・グラフ内のビジネス・オブジェクトに関連する変更の要約情報やイベントの要約情報の送達などの追加機能を提供するために、単純なビジネス・オブジェクトまたはビジネス・オブジェクトの階層に追加されるラッパーです。ビジネス・グラフは、SDO DataGraph 概念から直接導き出されます。ただし、ビジネス・グラフが単一の変更の要約ヘッダー以上のものを提供するという点を除きます。
タイプ・メタデータ	エンタープライズ・メタデータ ビジネス・オブジェクト・タイプ・メタデータ	ビジネス・オブジェクト・タイプ・メタデータは、実行時に値を拡張するためにビジネス・オブジェクト定義に追加できるメタデータです。これらのメタデータ項目は、`xs:annotation` エレメントおよび `xs:appinfo` エレメントとしてよく知られている、ビジネス・オブジェクトの XML スキーマ定義に追加されます。
サービス	ビジネス・オブジェクト・サービス (API)	ビジネス・オブジェクト・サービスは、サービス・データ・オブジェクトにより提供される基本機能のトップにある一連の機能です。例えば、作成、コピー、等価、直列化などのサービスです。これらの API は、`com.ibm.websphere.bo` パッケージ内にあります。

WebSphere Process Server ビジネス・オブジェクト・フレームワークは、SDO 規格の拡張機能です。そのため、WebSphere Process Server コンポーネント間で交換されるビジネス・オブジェクトは、`commonj.sdo.DataObject` クラスのインスタンスです。ただし、WebSphere Process Server ビジネス・オブジェクト・フレームワークは、基本的な DataObject 機能を単純化し、充実させる複数のサービスと関数を追加しています。

ビジネス・オブジェクトの作成と操作を容易にするため、WebSphere ビジネス・オブジェクト・フレームワークは一連の Java サービスを提供することによって SDO 仕様を拡張します。これらのサービスは、com.ibm.websphere.bo という名前のパッケージに含まれています。

- **BOFactory**: ビジネス・オブジェクトのインスタンスをさまざまな方法で作成できるようにする主要なサービスです。
- **BOXMLSerializer**: ストリームからのビジネス・オブジェクトを拡張する方法や、ビジネス・オブジェクトのコンテンツを XML フォーマットでストリームに書き込む方法を提供します。
- **BOCopy**: ビジネス・オブジェクトのコピーを作成するメソッドを提供します (「ディープ」および「シャロー」のセマンティクス)。
- **BODataObject**: ビジネス・オブジェクトのデータ・オブジェクトの側面 (変更サマリー、ビジネス・グラフ、イベント・サマリーなど) にアクセスできるようにします。
- **BOXMLDocument**: サービスに対するフロントエンドで、ここではビジネス・オブジェクトを XML 文書として操作できます。
- **BOChangeSummary** および **BOEventSummary**: ビジネス・オブジェクトの変更サマリーおよびイベント・サマリーの部分に簡単にアクセスして操作できるようにします。
- **BOEquality**: 2 つのビジネス・オブジェクトに同じ情報が含まれているかどうかを判別できるようにするサービスです。ディープおよびシャロー両方の等価をサポートします。
- **BOType** および **BOTypeMetaData**: これらのサービスは commonj.sdo.Type のインスタンスを実体化して、関連するメタデータを操作できるようにします。これによって、Type のインスタンスを使用して「タイプ別」にビジネス・オブジェクトを作成することができます。
- **BOInstanceValidator** : ビジネス・オブジェクト内のデータを検証して、XSD に準拠しているかどうかを調べます。

ビジネス・オブジェクトのモデル化

WebSphere Process Server ランタイムを通じて流れるビジネス・オブジェクト・データは、XML スキーマを使用してモデル化されます。XML スキーマは、文書タイプ定義 (DTD) の代替となるもので、データのタイプ指定、継承、および表示の領域で機能を拡張するために使用できます。XML スキーマでは、広く採用されている業界標準であり、プラットフォームと言語に中立な、データ型のモデル化用のフォームを提供します。

ターゲット名前空間定義:

XML が解決可能なほとんどのビジネスおよび通信の問題では、いくつかの XML ボキャブラリーの組み合わせが必要です。XML には、条件を満たす名前を、さまざまな業界に適用される名前空間など、異なる名前空間に割り当てるメカニズムがあります。XML では、Uniform Resource Identifier (URI) によって、XML スキーマ内のエレメント、属性、および型定義に関連付ける固有の名前が提供されます。

ビジネス・オブジェクトのターゲット名前空間には、以下の 2 つの要件があります。

- ビジネス・プロセス管理ランタイムおよびツールでは、ターゲット名前空間が urn:foo:com:xyz よりも http://www.foo.com/xyz のような形式であることが推奨されます。
- ビジネス・オブジェクト・フレームワークでは、ビジネス・オブジェクトのターゲット名前空間が必要です。

ビジネス・オブジェクト定義:

WebSphere Process Server では、ビジネス・オブジェクトの定義またはインポートのための柔軟なメカニズムを提供します。

WebSphere Process Server は、基本的に、以下の 3 つの異なる形式の XML スキーマをビジネス・オブジェクト定義として認識します。

- トップレベルの複合タイプ定義
- トップレベルの匿名複合タイプ定義
- 名前付き複合タイプを参照するトップレベル・エレメント

トップレベルの複合タイプ定義

以下に示すのは、トップレベルの複合タイプ定義の例です。

```
<complexType name="ProductType">
 <sequence>
  <element name="name" type="string"/>
  <element name="color" type="string" maxOccurs="unbounded"/>
 </sequence>
</complexType>
```

すべて複合タイプ定義のみを使用して定義されたビジネス・オブジェクトをインポートまたは定義するのが、最も柔軟で管理しやすい方式です。このモデルの上部は、再利用を可能にするタイプ・ライブラリーです。3 つの異なる方法で再利用できます。

- (拡張または制限の) 複合タイプ導出モデルを使用して新規タイプを作成できます。
- 既存の複合タイプおよび使用可能な単純タイプをプリミティブとして使用して、新規の集合体型を作成できます。
- 集合体複合タイプと同様に、新規の複合文書定義を作成できます。

複合タイプとして定義されたビジネス・オブジェクトのその他の影響としては、JService コンポーネント種類がランタイム内でデータを転送するために複合タイプを使用するとき、WS-I 準拠性を維持するためには、指定された複合タイプを参照するエレメントを作成する必要があります。

トップレベルの匿名複合タイプ定義

以下に示すのは、トップレベルの匿名複合タイプ定義の例です。

```
<element name="Product">
 <complexType>
  <sequence>
   <element name="name" type="string"/>
   <element name="color" type="string" maxOccurs="unbounded"/>
  </sequence>
 </complexType>
</element>
```

インポートされたビジネス・オブジェクトがすべて匿名エレメント定義である場合、それらを JService 呼び出しに組み込むことができます。ただし、それらは元来再利用可能ではありません。

名前付きのタイプを参照するトップレベル・エレメント

以下に示すのは、名前付きのタイプを参照するトップレベル・エレメントの例です。

```
<element name="product" type="prod:ProdType"/>
```

名前付き複合タイプを参照するビジネス・オブジェクトは、しばしば、エレメント定義を必要とする WSDL 操作を既に定義した環境で使用されます。このシナリオでは、複合タイプおよびエレメント定義が処分される可能性があることを考慮することが重要です。

- エレメントは、同じ XML スキーマ・ファイルで複合タイプ定義と共存できます。
- エレメントは、WSDL ファイルに組み込まれた複合タイプ定義と共存できます。
- 複合タイプ定義が XML スキーマ・ファイル B.xsd で定義されているときに、エレメントを XML スキーマ A.xsd で定義できます。
- エレメントは、WSDL ファイルに組み込まれて、XML スキーマ・ファイルで定義された複合タイプ定義を参照することができます。

例

以下の例は、まとめて結合されたビジネス・オブジェクトを定義するためのすべてのメカニズムを示します。

```
<schema
 targetNamespace="http://www.app.com/Address"
 xmlns:addr="http://www.app.com/Address"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema">

 <complexType name="Address">
  <sequence>
   <element name="street1" type="string"/>
   <element name="street2" type="string"/>
   <element name="city" type="string"/>
   <element name="state" type="string"/>
   <element name="zip" type="string"/>
  </sequence>
 </complexType>

 <element name="homeAddress" type="addr:Address"/>
 <element name="workAddress" type="addr:Address"/>
 <element name="otherAddress" type="addr:Address"/>

 <element name="individualContact">
  <complexType>
   <sequence>
    <element name="firstName" type="string"/>
    <element name="lastName" type="string"/>
    <element ref="addr:HomeAddress"/>
    <element ref="addr:WorkAddress"/>
    <element ref="addr:OtherAddress"/>
   </sequence>
  </complexType>
 </element>
```

```
 <element name="businessContact">
  <complexType>
   <sequence>
    <element name="name" type="string"/>
    <element ref="addr:WorkAddress"/>
   </sequence>
  </complexType>
 </element>

 <element name="chairmanOfTheBoard">
  <complexType>
   <sequence>
    <element name="startDate" type="date"/>
    <element ref="addr:IndividualContact"/>
    <element ref="addr:BusinessContact"/>
   </sequence>
  </complexType>
 </element>
</schema>
```

以下の指針は、ビジネス・オブジェクトを定義する際に推奨される方法です。

- エレメントは、名前付きのタイプを使用して定義します。匿名タイプは推奨されません。
- エレメントおよび複合タイプ定義が、同じ XML スキーマまたは WSDL ファイルで共存しないようにします。この方法では、タイプの再利用を推奨しません。
- 複合タイプは WSDL 定義ではなく XML スキーマ・ファイルで定義され、概念のようなタイプ・ライブラリーを作成します。また、このタイプの定義では、複合タイプの再利用が可能で、推奨されます。
- 単一の複合タイプ定義を参照するエレメント定義は必要に応じて作成します。例えば、WSDL 内でのエレメントの定義は、推奨されるパターンです。
- エレメント定義は、通常、複合タイプ定義と同じターゲット名前空間を使用します。

ビジネス・オブジェクト・プロパティー定義:

XML スキーマは、複合タイプ、単純タイプ、および属性を提供します。これらは、ビジネス・オブジェクトの作成に使用されます。

複合タイプ定義、匿名複合タイプ定義、および複合タイプ定義を参照するエレメントが、外側のビジネス・オブジェクトを定義するために使用されます。ビジネス・オブジェクト内部のデータを定義するために、条件プロパティーが使用されます。条件は、サービス・データ・オブジェクトの条件プロパティーから派生し、`commonj.sdo.Property` インターフェースによって定義されます。これは属性の概念と同義です。

プロパティーは、シンプル・プロパティーまたは複合プロパティーのいずれかになります。シンプル・プロパティーは、XML スキーマ属性として、またはタイプが XML スキーマ単純タイプである XML スキーマ・エレメントとして、定義することができます。複合プロパティーは、別のビジネス・オブジェクト参照するか、または現在のビジネス・オブジェクト内で複雑な構造を定義することができます。

XML スキーマ・タイプ・システムは完全にサポートされています。

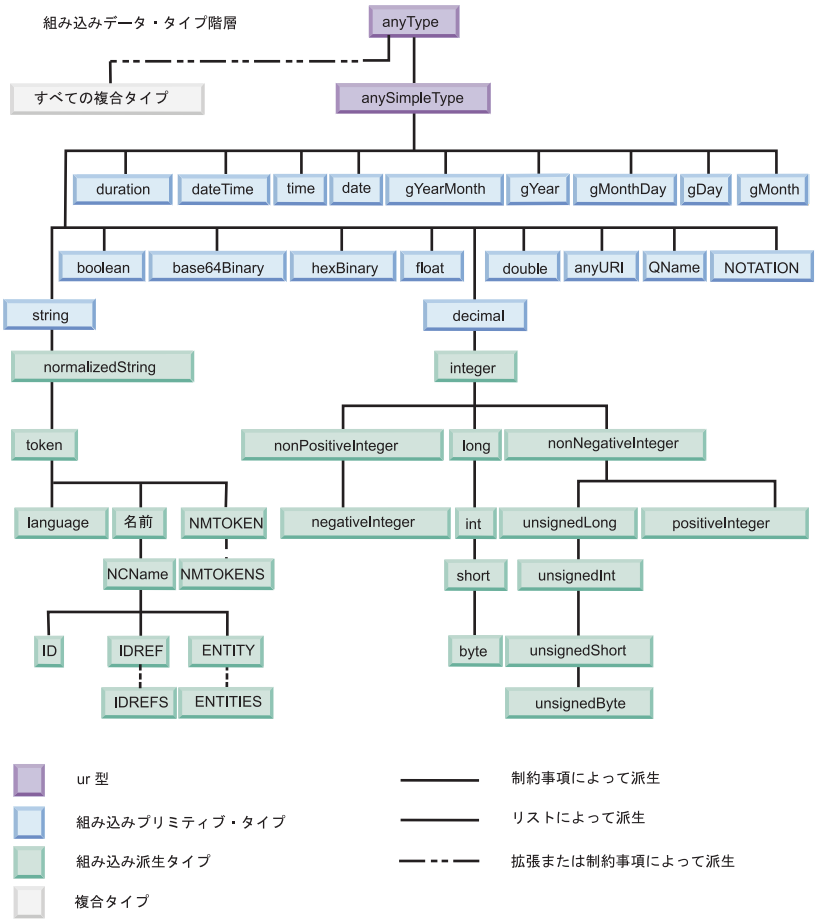


図 56. XML スキーマ単純タイプ

サポートされる XSD および WSDL 成果物:

WSDL またはスキーマが WebSphere Integration Developer 内のプロジェクトにインポートされると、WSDL またはスキーマからレンダリングされたビジネス・オブジェクトを使用してモジュールを作成できます。ただし、スキーマからの特定の成果物のみが、ビジネス・オブジェクト (例えば、ルート/トップレベル・エレメントおよび名前付き複合タイプ) としてレンダリングされることに注意することが重要です。ネストされた匿名複合タイプなどの特定の成果物は、ビジネス・オブジェクトとしてレンダリングされません。これらの制限は、XML スキーマでどの成果物がアクセス可能であるかによって決まります。例えば、スキーマをインポートしてビジネス・オブジェクトが 1 つのみ生成された場合、残りのエージェント・マネージャーは匿名複合タイプであった可能性が非常に高くなります。どの XSD および WSDL 成果物からビジネス・オブジェクトが生成されるかについて、以下に詳しく説明します。

インポートされた XSD 定義からのビジネス・オブジェクト

XML スキーマがプロジェクトにインポートされると、特定の成果物のみがビジネス・オブジェクトとしてレンダリングされます。以下のリストに、オーサリング時および実行時にサポートされる成果物を示します。

オーサリング時にビジネス・オブジェクトが生成される XSD 成果物は以下のとおりです。

- ルート・レベルで定義された複合タイプ
- 匿名複合タイプによってルート・レベルで定義されたエレメント

オーサリング時に、以下の成果物から、ビジネス・オブジェクトが参照できるユーザー定義単純タイプが生成されます。

- ルート・レベルで定義された単純タイプ
- 匿名単純タイプによってルート・レベルで定義されたエレメント

インポートされた WSDL ファイルからのビジネス・オブジェクト

インライン XSD スキーマを含む WSDL 定義がプロジェクトにインポートされると、特定の成果物のみがビジネス・オブジェクトとしてレンダリングされます。以下のリストに、オーサリング時および実行時にサポートされる成果物を示します。

オーサリング時にビジネス・オブジェクトが生成されるインライン XSD 成果物は以下のとおりです。

- ルート・レベルで定義された複合タイプ
- 匿名複合タイプによってルート・レベルで定義されたエレメント。エレメントの名前には操作/メッセージの名前が含まれていない (これらのエレメントは、WebSphere Integration Developer が自動的にアンラップする文書/リテラル・ラップ・エレメントである可能性があるため)。

オーサリング時に、以下の成果物から、ビジネス・オブジェクトが参照できるユーザー定義単純タイプが生成されます。

- ルート・レベルで定義された単純タイプ
- 匿名単純タイプによってルート・レベルで定義されたエレメント

XSD 成果物からのランタイム・ビジネス・オブジェクト

実行時に、以下の成果物からビジネス・オブジェクトが生成されます。

- ルート・レベルで定義された複合タイプ
- 匿名複合タイプによってルート・レベルで定義されたエレメント
- 複合タイプを参照する、ルート・レベルで定義されたエレメント

WSDL ファイルからのランタイム・ビジネス・オブジェクト

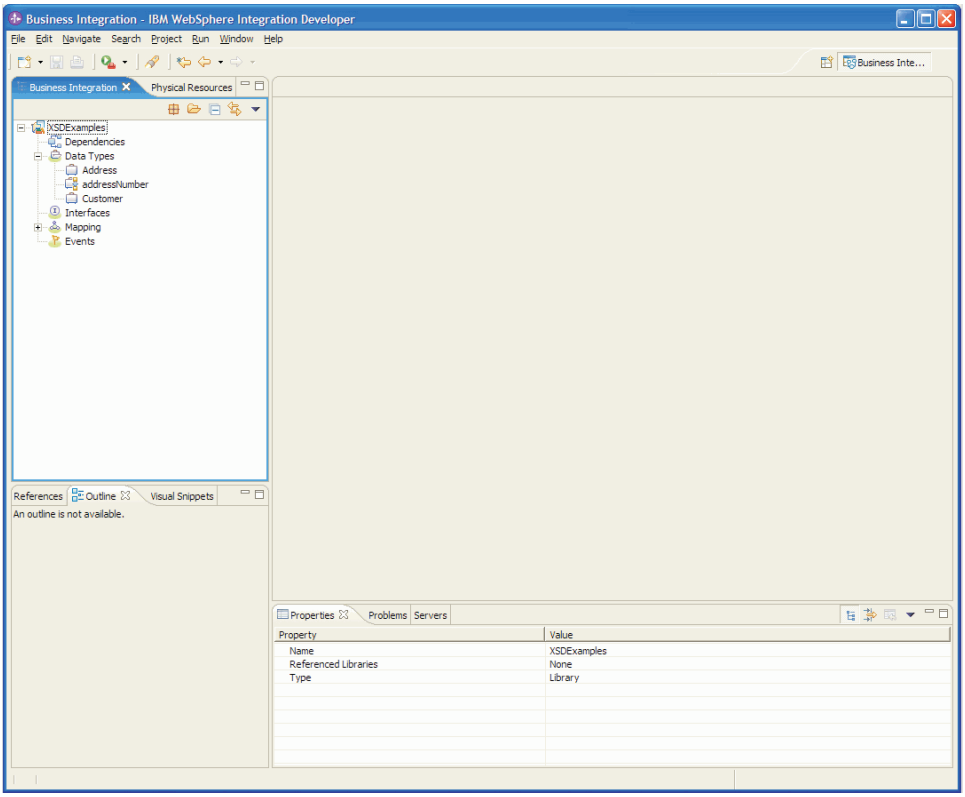
実行時に、以下の成果物からビジネス・オブジェクトが生成されます。

- ルート・レベルで定義された複合タイプ
- 匿名複合タイプによってルート・レベルで定義されたエレメント。エレメントの名前には操作/メッセージの名前が含まれていない (これらのエレメントは、WebSphere Integration Developer が自動的にアンラップする文書/リテラル・ラップ・エレメントである可能性があるため)。
- 複合タイプを参照する、ルート・レベルで定義されたエレメント

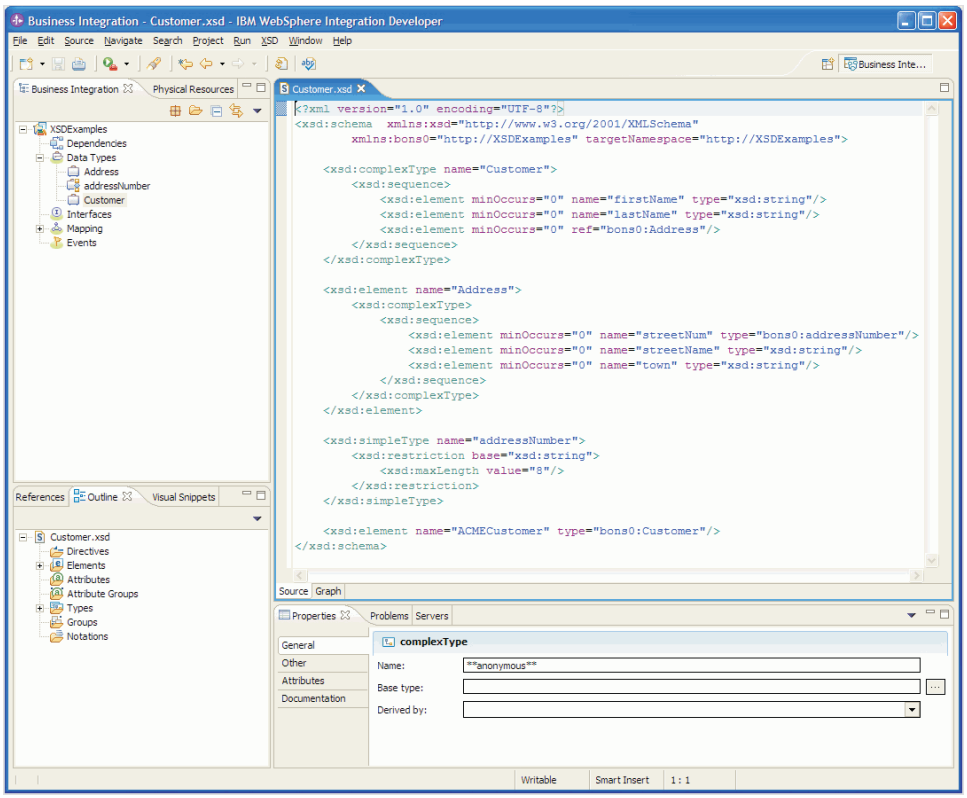
例

1. **XSD の例 (オーサリング時にサポートされる)**

次の例では、ビジネス・オブジェクトが表示された「ビジネス・インテグレーション」ビュー内のプロジェクト (XSDExamples) を示します。



以下に、XSD スキーマ・エディターでの Customer.xsd ファイルを示します。



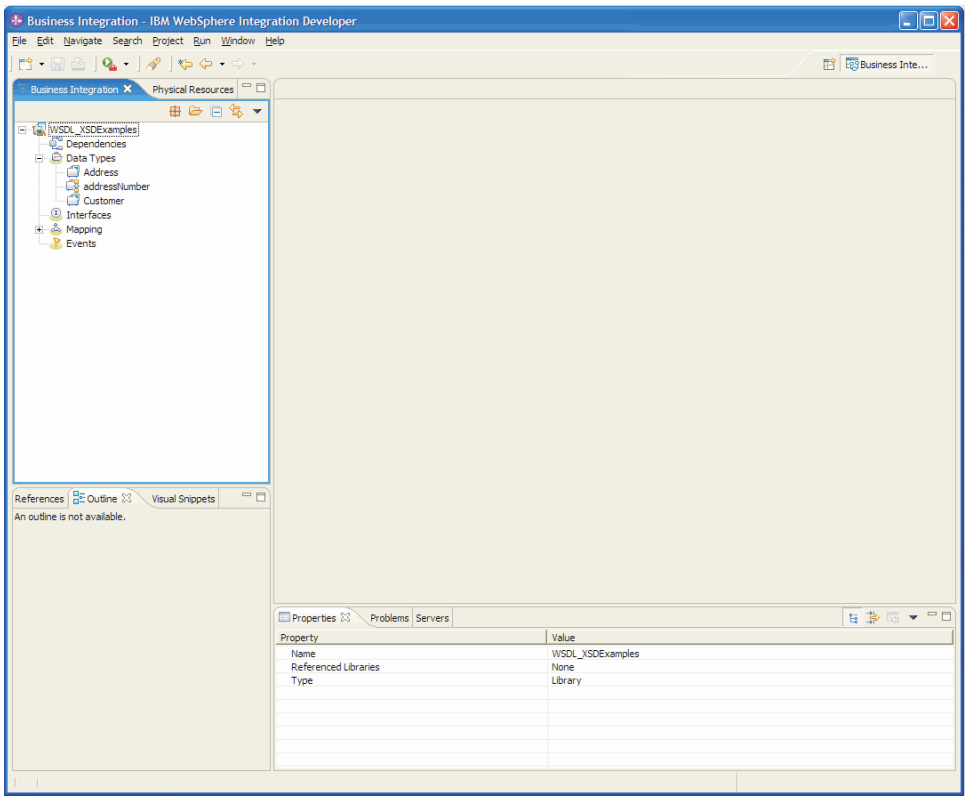
上記の例では、以下のサポートを例示しています。

表 19. XSD 成果物サポート

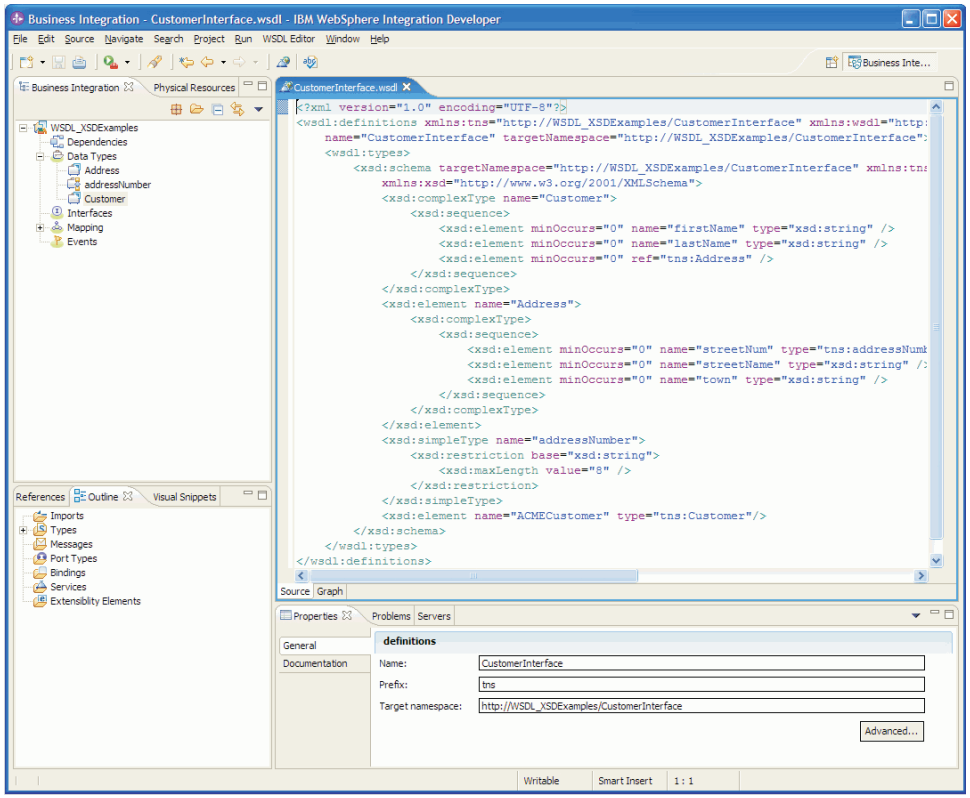
XSD サポート	上記の例の XSD 成果物
ルート・レベルで定義された複合タイプ	Customer
匿名複合タイプによってルート・レベルで定義されたエレメント	Address
ユーザー定義単純タイプによってルート・レベルで定義されたエレメント	addressNumber

2. **WSDL の例 (オーサリング時にサポートされる)**

次の例では、ビジネス・オブジェクトが表示された「ビジネス・インテグレーション」ビュー内のプロジェクト (WSDL_XSDExamples) を示します。



次の画面取りでは、WSDL エディターで開かれた CustomerInterface.wsdl ファイルを示します。



上記の例では、以下のサポートを例示しています。

表 20. WSDL 成果物サポート

WSDL サポート	**上記の例のインライン XSD 成果物**
ルート・レベルで定義された複合タイプ	Customer
匿名複合タイプによってルート・レベルで定義されたエレメント。エレメントの名前には操作/メッセージの名前が含まれていない (これらは、WebSphere Integration Developer が自動的にアンラップする文書/リテラル・ラップ・エレメントである可能性があるため)。	Address
ユーザー定義単純タイプによってルート・レベルで定義されたエレメント	addressNumber

3. **ランタイムの例**

上記の例では、以下のランタイム・サポートを例示しています。

表 21. ランタイム成果物サポート

ランタイム・サポート	**上記の例の XSD またはインライン XSD 成果物**
上記の例 1 および例 2 のすべて (単純タイプはビジネス・オブジェクトではないため addressNumber は除く)	上記参照 (例 1 および例 2)
複合タイプを参照する、ルート・レベルで定義されたエレメント	ACMECustomer (例 1 および例 2 に示す)

フラット・ビジネス・オブジェクトおよび階層ビジネス・オブジェクト:

ビジネス・オブジェクトは、フラットに、または階層的にモデル化することができます。

フラット・ビジネス・オブジェクト

フラット・ビジネス・オブジェクト には、1 つ以上の単純属性と、サポートされる動詞のリストが含まれています。単純属性は、String、Integer、または Date などの 1 つの値を表します。すべての単純属性は、単一カーディナリティーを持ちます。ビジネス・オブジェクトがアプリケーション固有のビジネス・オブジェクトである場合、フラット・ビジネス・オブジェクトは、アプリケーションまたはテクノロジー内の 1 つのエンティティーを表すことができます。

階層ビジネス・オブジェクト

階層ビジネス・オブジェクト 定義は、複数の関連するエンティティーの構造を定義し、個々のエンティティーだけでなくエンティティー間の関係の性質もカプセル化します。階層ビジネス・オブジェクトには、少なくとも 1 つの単純属性が含まれ、さらに、1 つ以上の複合的な (すなわち属性自体が、*子ビジネス・オブジェクト* と呼ばれる 1 つ以上のビジネス・オブジェクトを含む) 属性もあります。複合属性を含むビジネス・オブジェクトは、*親ビジネス・オブジェクト* と呼ばれます。

親ビジネス・オブジェクトと子ビジネス・オブジェクトの関係には以下の 2 つのタイプがあります。

- 単一カーディナリティー - 親ビジネス・オブジェクトの属性が単一の子ビジネス・オブジェクトを表す場合。属性のタイプは子ビジネス・オブジェクトの名前に設定され、カーディナリティーは 1 に設定されます。
- 複数カーディナリティー - 親ビジネス・オブジェクト内の属性が、子ビジネス・オブジェクトの配列を表す場合。属性のタイプは子ビジネス・オブジェクトの名前に設定され、カーディナリティーは *n* に設定されます。

同様に、子ビジネス・オブジェクトのそれぞれに、子ビジネス・オブジェクトまたはビジネス・オブジェクトの配列を含む属性が含まれていてもかまいません。それ以下も同様です。階層の一番上のビジネス・オブジェクト (それ自身は親を持たない) は、*トップレベル・ビジネス・オブジェクト* と呼ばれます。単一のビジネス・オブジェクトはすべて、それが格納している子ビジネス・オブジェクト (あるいはそれが格納されている子ビジネス・オブジェクト) に関わりなく、*個別ビジネス・オブジェクト* と呼ばれます。

例

以下の例では、フラット・ビジネス・オブジェクトと階層ビジネス・オブジェクトの相違点を示します。図には、Product という名前のフラット・ビジネス・オブジェクトがあります。このビジネス・オブジェクトは、メモリー内で、サービス・データ・オブジェクト・タイプ `commonj.sdo.DataObject` で表されます (静的に生成された場合を除く)。このフラット・ビジネス・オブジェクトには、XML スキーマ単純タイプとしてモデル化された一連の属性と、単純タイプのリストとしてモデル化された属性があります。

この図ではまた、Product ビジネス・オブジェクトと ProductCategory ビジネス・オブジェクトを組み合わせてさらに複雑な階層ビジネス・オブジェクトを作成する様子も示します。このビジネス・オブジェクトには、トップレベル・ビジネス・オブジェクト (ProductCategory) と、中に含まれるビジネス・オブジェクト (Product) の両方があります。

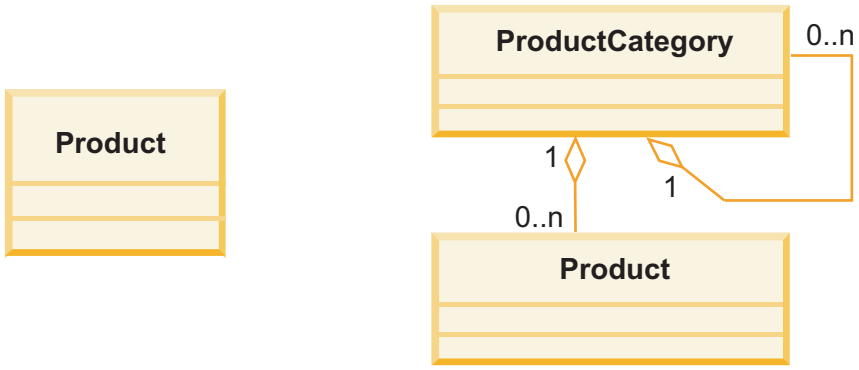


図 57. フラット・ビジネス・オブジェクトと階層ビジネス・オブジェクトの比較

Product ビジネス・オブジェクトのフラット・ビジネス・オブジェクト定義の例を示します。Product ビジネス・オブジェクトは、*Name* および *Inventory* という 2 つのプロパティーを定義しています。これらは、XML スキーマ単純タイプ `xs:string` および `xs:int` で型付けされます。さらに Product では、`xs:string` 単純タイプのリストである List プロパティー *Color* の定義も示しています。

```
<schema>
 targetNamespace="http://www.scm.com/ProductTypes"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema">

 <complexType name="Product">
  <sequence>
   <element name="name" type="string"/>
   <element name="inventory" type="int"/>
   <element name="color" type="string" maxOccurs="unbounded"/>
  </sequence>
 </complexType>
</schema>
```

以下に階層ビジネス・オブジェクト ProductCategory の例を示します。定義では、2 つの異なるビジネス・オブジェクト ProductCategory および Product を定義します。ProductCategory 階層ビジネス・オブジェクトは、プロパティー *Name* を定義し、`Product` タイプまたは `ProductCategory` タイプのビジネス・オブジェクトのリストも定義します。

```
<schema>
 targetNamespace="http://www.scm.com/ProductCategoryTypes"
 xmlns:pc="http://www.scm.com/ProductCategoryTypes"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema">
 elementFormDefault="qualified">

 <complexType name="ProductCategory">
  <sequence>
   <element name="name" type="string"/>
   <choice>
    <element name=="productCategory"
     type="pc:ProductCategory"
     maxOccurs="unbounded"/>
    <element name=="product"
```

```
      type="pc:Product"
      maxOccurs="unbounded"/>
    </choice>
   </sequence>
  </complexType>

  <complexType name="Product">
   <sequence>
    <element name="name" type="string"/>
    <element name=="inventory" type="int"/>
    <element name=="color" type="string" maxOccurs="unbounded"/>
   </sequence>
  </complexType>

</schema>
```

ビジネス・オブジェクトの特性:

ビジネス・オブジェクトには、ビジネス・オブジェクト・フレームワーク内での使用を強化する、特有の特性があります。

カーディナリティー

プロパティーのカーディナリティーは、単純タイプと複合タイプの場合は、標準の XML スキーマの `minOccurs` および `maxOccurs` ファセットによって定義され、属性の場合は `use` 属性によって定義されます。

デフォルトのプロパティー値

ビジネス・オブジェクト内の属性および単純タイプに対して、XML スキーマでデフォルト値を提供する機能は、ビジネス・オブジェクト・フレームワークによってサポートされます。この機能は作成時にサポートされ、このとき、ビジネス・オブジェクトの単純プロパティー・タイプはデフォルト値を反映します。

ヌルを指定可能

エレメントは、XML スキーマ内で、ヌルを指定可能であると定義することができます。ビジネス・オブジェクト・フレームワークによって、 *nillable* なプロパティーの値が実行時に `Null` に設定されるようにすることができます。

キー定義

リレーションシップ、順序付け、および分離機能などの複数のサブシステムが、ビジネス・オブジェクト・キー情報を使用できます。ただし、これらのサブシステムは、ビジネス・オブジェクトのキー定義にかかわらず、それぞれ独自の中心となるメカニズムを定義できます。ビジネス・オブジェクトが利用する基礎となるモデル言語は XML スキーマであるため、キー定義のファースト・クラスのサポートは、モデル化言語内に存在します。ただし、モデル化言語内にあるこのサポートは、SDO ランタイムでは完全にはサポートされていません。

xs:ID、xs:IDREF、および xs:IDREFS

これらのタイプは、主に DTD のアップグレード・パスを提供するために XML スキーマに追加されました。複合タイプはそれぞれ、`xs:ID` 型のエレメント/属性を 0 または 1 個持つことができます。ID は、例えば表のスコープに関して固有でなければならないデータベース内の 1 次キーとは異なり、文書全体に固有でなければなりません。例として、適合する文書では、

Product および ProductCategory の両方を同じ ID 値を使用して識別することはできません。多くの場合エレメントは、複合タイプの名前をキー値の先頭に付加することによって、この制限を回避します。IDREF タイプの属性には、現在の文書内の ID 値のいずれかに一致する値が含まれていなければなりません。 さらに、XML スキーマでは、ID 参照のリストを含むように型付けできるエレメントである構成体 xs:IDREFs を提供する必要があります。

xs:unique、xs:key、xs:keyref

XML スキーマでは、キー定義およびキー参照を使用可能にする新しいスタイルを導入しました。xs:unique 構成体によって、ユーザーは、エレメント内で、エレメントの特定のスコープ (文書全体を表す) 内で固有でなければならない 1 つ以上のフィールドを定義できます。xs:key 構成体は、xs:unique のバリアントで、参照されるエレメントが必須であるという追加の制限があります。xs:keyref 構成体は、エレメントの値が名前付きキーまたは固有の構成体でなければならないことを示すために使用されます。

unique、key、および keyref 構成体には、ID、IDREF、および IDREFS を設定するよりも有利な点がいくつかあり、それらを以下に示します。

- 複合キーを定義できる。
- 文書の一部分に関係する固有の制限を定義できる。

ビジネス・オブジェクトに定義済みキーは必須ではありませんが、定義済みキーを含めることを強くお勧めします。キーを定義しないビジネス・オブジェクトを、アプリケーションが使用することができます。このシナリオは、多くの Java EE 中心のアプリケーション使用モデルで一般的な使用モデルです。この場合、キーを指定することなく、サーブレットと EJB コンテナーの間で JavaBeans が受け渡されます。ただし、キーを定義しないビジネス・オブジェクトは、キーを必要とするサブシステムと対話することはできません。この状態では、WebSphere Process Server のサービス品質を活用する能力が制限されます。

ビジネス・グラフのモデル化

ビジネス・グラフからビジネス・オブジェクトへの関係は、SDO DataGraphs から SDO DataObjects への関係とほぼ同じです。トップレベル・ビジネス・オブジェクトで、WebSphere Process Server によって提供されるサービスを使用できるように機能を拡充する必要がある場合、トップレベル・ビジネス・オブジェクトはビジネス・グラフでラップされます。ビジネス・グラフ・ラッパーは、情報を論理的にメモリーに格納するか、ビジネス・グラフが直列化されたときに物理的に格納するためのデータ・ヘッダーを追加することによって、追加の値を提供します。

注: WebSphere InterChange Server からアプリケーションをマイグレーションしているか、アダプターをマイグレーションしている場合、ビジネス・グラフの使用が必要な場合があります。

ビジネス・グラフ使用モデル:

2 つの主な使用モデルが、ビジネス・グラフの提供する基本的な機能を表します。デルタ・サポートおよび変更後イメージです。

デルタ・サポート は、SDO 1.0 によって使用可能になる機能で、ビジネス・オブジェクト・グラフへの変更は、*変更の要約* という特殊なヘッダーに収集されます。

変更後イメージ は、EIS システム内のビジネス・データの現在の状態を、通常は EIS 内のそのデータへの変更の結果として取り込むビジネス・グラフです。変更後イメージによって、EIS システムでの変更を取り込み、ランタイムに公開することができます。

これらの 2 つの基本的な概念を提供するため、ビジネス・グラフは以下の概念を導入し、提供しています。

- *テンプレート・ビジネス・グラフ* は、ラップするビジネス・オブジェクト・グラフのタイプに特化したタイプのビジネス・グラフです。
- *変更の要約* は、デルタ使用モデルと変更後イメージ使用モデルの両方で、暗黙的な変更と明示的な変更を取り込むために提供されます。
- *明示的変更の要約* のサポートは、ビジネス・グラフ・プログラミング・インターフェースによって提供され、これにより、BPM コンポーネント (アダプター、メディエーター、マップ、およびリレーションシップ) が明示的に変更の要約ヘッダーを変更できるようになります。
- *イベントの要約* は、ビジネス・オブジェクト・グラフ内のデータに関するインスタンス・ベースのアノテーションの収集をサポートするために提供されています。具体的には、オブジェクト・イベント ID を含んでいます。
- *動詞* サポートは、ビジネス・グラフによって提供され、ランタイム内のコンポーネントが、イベント・タイプをキーオフして、付加価値機能を実行できるようにします。
- *サポートされる動詞* は、ビジネス・グラフに指定可能な許容される動詞のセットの制約または拡張の概念です。
- *オブジェクト・イベント ID* は、ビジネス・グラフによってサポートされ、グラフ内のすべてのオブジェクトを、一意に識別できるようにします。この機能は、付加価値機能を提供する一部のコンポーネントで必要です。

ビジネス・グラフ・モデル定義:

外部で作成されたビジネス・オブジェクトのモデルに介入しない状態を維持するため、ビジネス・グラフ機能が、元のビジネス・オブジェクトを囲むようにラップされます。テンプレート・ビジネス・グラフという名前のパターンを使用して、機能の充実したビジネス・グラフ・スキーマで元のビジネス・オブジェクトをラップします。

テンプレート・ビジネス・グラフ は、ビジネス・オブジェクト・フレームワーク・ランタイムが提供するビジネス・グラフ複合タイプを拡張し、元のビジネス・オブジェクトに委任するエレメントを追加することによって作成されます。図に、ビジネス・グラフの UML モデルを示します。このビジネス・グラフは抽象的であり、トップレベル・ビジネス・オブジェクトに追加された標準のヘッダーのセットのみを提供します。

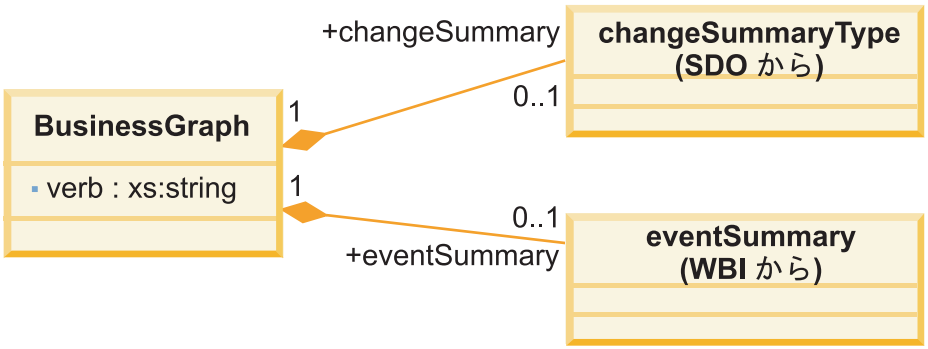


図 58. ビジネス・グラフ複合タイプ

抽象ビジネス・グラフ XML スキーマ複合タイプの XML スキーマ・モデル:

```
<schema
 targetNamespace="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
 xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
 xmlns:sdo="commonj.sdo"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

 <import namespace="commonj.sdo" schemaLocation="DataGraph.xsd"/>

 <complexType name="BusinessGraph" abstract="true">
  <sequence>
   <element name="changeSummary" type="sdo:ChangeSummaryType"
    minOccurs="0" maxOccurs="1"/>
   <element name="eventSummary" type="bo:EventSummary"
    minOccurs="0" maxOccurs="1"/>
   <element name="property" type="bo:ValuesType"
    minOccurs="0"/>
  </sequence>
  <anyAttribute namespace="##other" processContents="lax"/>
 </complexType>

 <complexType name="EventSummary">
  <sequence>
   <any namespace="##any" processContents="lax"
    minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
 </complexType>

 <complexType name="ValuesType">
  <complexContent>
   <extension base="ecore:EClass"/>
  </complexContent>
 </complexType>

 <attribute name="name" type="string"/>
</schema>
```

ビジネス・グラフは、最上位概念にすぎません。それらは、ヘッダーのセットを既存のビジネス・オブジェクトに追加するために存在し、ビジネス・オブジェクトのように再帰的設計パターンにモデル化することはできないためです。ビジネス・グ

ラフは任意のビジネス・オブジェクトに適用できますが、アプリケーションでは、そのビジネス・オブジェクトがトップレベル・ビジネス・オブジェクトになります。

以下に、ProductCategory という名前のビジネス・オブジェクトをラップするビジネス・グラフの例を示します。ProductCategory は、Product という名前の子ビジネス・オブジェクトを含む階層ビジネス・オブジェクトです。

```
<schema
 targetNamespace="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
 xmlns:pcbg="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
 xmlns:pc="http://www.scm.com/ProductCategoryTypes"
 xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

 <import namespace="http://www.ibm.com/xmlns/prod/websphere/bo/6.0.0"
  schemaLocation="BusinessGraph.xsd"/>

 <import namespace="http://www.scm.com/ProductCategoryTypes"
  schemaLocation="ProductCategoryTypes.xsd"/>

 <complexType name="ProductCategoryBG">
  <complexContent>
   <extension base="bo:BusinessGraph">
    <sequence>
     <element name="verb" minOccurs="0" maxOccurs="1"/>
     <element name="productCategory"
      type="pc:ProductCategory"
      minOccurs="0" maxOccurs="1"/>
    </sequence>
   </extension>
  <complexContent>
 <complexType>
</schema>
```

生成されたテンプレート・ビジネス・グラフに推奨されるパターンは、以下のとおりです。

- テンプレート・ビジネス・グラフは、制限によって bo:BusinessGraph スキーマを拡張する、指定された複合タイプを使用して定義されます (このパターンでは、許容される Verb 値を制限します)。
- テンプレート・ビジネス・グラフの名前は、トップレベル・ビジネス・オブジェクトの名前にストリング「BG」を追加したものです。
- テンプレート・ビジネス・グラフのターゲット名前空間は、ラップされるビジネス・オブジェクトのターゲット名前空間の後に「/」が付いたものです。これは、テンプレート・ビジネス・グラフの複合タイプによって指定されます。
- テンプレート・ビジネス・グラフ複合タイプ定義は、それ自身の XML スキーマ・ファイル (名前が複合タイプの名前に対応している) 内にあります。

ビジネス・グラフ・モデル・インスタンス:

トップレベル・ビジネス・グラフは、メモリー内で、SDO 1.0 DataObject (特に、クラス commonj.sdo.DataObject) によって、ほとんどビジネス・オブジェクトと同じように表されます。

ビジネス・グラフを構成するのは、ビジネス・グラフ包含オブジェクトだけではありません。2 つのヘッダーと、トップレベル・ビジネス・オブジェクトも含まれています。メモリー内で DataObject として表されるヘッダーはなく、DataObject API アクセス機構を使用可能にするヘッダーもありません。

変更の要約ヘッダー

ビジネス・グラフが提供する機能は、ビジネス・グラフがいくつかの異なるビジネス・プロセス間で受け渡されるときに、ビジネス・グラフ内のビジネス・オブジェクトへの変更を暗黙的に追跡する機能です。各プロセスがビジネス・グラフを変更すると、メモリーに変更ログが生成されます。ビジネス・グラフが直列化されると、変更ログは、次のプロセスがビジネス・グラフに加えられた変更のタイプを確認できる形式で書き出されます。この手法を使用すると、アダプターとデータ・メディエーター・サービスが、重点的に扱う必要のあるデータを最適化することによって、パーシスタンス・データ・ストアを効率的に更新できます。

さらに、変更の要約は、アダプターが変更後イメージ・イベントを生成したときにも、EIS システムで更新されたデータを記述するために使用されます。特に、変更の要約がオブジェクトおよびプロパティーの変更にアノテーションを付ける機能が、変更後イメージ使用モデル (およびビジネス・グラフの動詞) で使用されます。

暗黙的な変更の要約の使用

アプリケーションによってビジネス・オブジェクトが変更されるとき、アプリケーションは、変更イベントが自動的に変更の要約に記録されるように、変更ロギングをオンにすることができます。変更の要約は、トップレベル・ビジネス・オブジェクトと同等のレベルでビジネス・グラフに含まれています。変更イベントは 2 つのレベルで定義されます。ビジネス・オブジェクト用と、ビジネス・オブジェクト・プロパティー用です。ビジネス・オブジェクトには、create、update、および delete 変更タイプがあり、プロパティーには set および unset 変更タイプがあります。変更イベントは、ビジネス・グラフのビジネス・オブジェクト部分への変更の場合のみ追跡されます。イベントの要約で変更を行っても、ビジネス・グラフの変更の要約が暗黙的に更新されることはありません。

明示的な変更の要約の修正

明示的に変更の要約ヘッダーに書き込む機能を必要とする WebSphere Process Server コンポーネントで、いくつかの使用モデルがあります。例えば、EIS イベントを明示的に生成するアダプターは、オブジェクト変更タイプを作成します。また、プロパティー変更タイプも作成する可能性があります。アプリケーション固有のビジネス・オブジェクト/汎用ビジネス・オブジェクト (ASBO/GBO) マップは、変更の要約を、あるビジネス・グラフから別のビジネス・グラフに変換し、出力ビジネス・グラフで新規バージョンの変更の要約を作成します。この機能は、ビジネス・オブジェクト・フレームワークによって提供されます。

イベントの要約ヘッダー

イベントの要約では、**ObjectEventID** を提供します。これは、実行時に出現するオブジェクトのインスタンスを一意に識別するために使用される手段です。この情報

は、イベントの要約に入れて運ばれます。イベントの要約では、固有 ID は、ビジネス・グラフのビジネス・オブジェクト階層内の所定の DataObject に関連付けられています。

イベント情報も、イベントの要約に入れることができます。この情報は、ビジネス・グラフのビジネス・オブジェクト階層内で、各オブジェクトに関連付けられたメタデータを追加するために使用できるストリングです。イベント情報の使用モデルの 1 つとして考えられるのは、変更の要約がサポートする標準の Create、Update、および Delete 動詞以外の動詞によって、含まれているビジネス・オブジェクトをマークアップする場合です。

動詞ヘッダー

動詞がビジネス・グラフに設定された場合、ビジネス・グラフのビジネス・オブジェクト・データ部分には、EIS 変更後イメージ・データ・セットが入れられます。動詞に値が含まれる場合、変更後イメージ・イベントの細分度に関して、以下の 3 つの可能性があります。

- 変更の要約が空である。この状態になるのは、EIS がデータ・セットへの更新のタイプを認識しているが、グラフ内のどのオブジェクトが作成、更新、または削除されたかについて詳細が分からない場合です。その結果、下流のメディエーション、マップ、リレーションシップ、またはアダプターは、ビジネス・グラフ内の情報だけでなく追加データも使用して、実行すべき実際の更新を判別する必要があります。
- 変更の要約に、オブジェクト・レベルの変更イベント・アノテーションがある。EIS システムがビジネス・グラフ内の各オブジェクトで行われた操作を認識しているが、オブジェクトの特定のプロパティーが更新されたかどうかを判断するための細分度が足りない場合、通常このようになります。
- 変更の要約に、オブジェクト・レベルの変更イベント・アノテーションとプロパティー・レベルの取得/設定アノテーションがある。これは、最も細分度が高いケースであり、EIS システムが該当データを提供する場合、すべてのアダプターは、このレベルの変更後イメージを取得しようとします。完全に指定された変更後イメージの利点は、プロパティー・レベルの暗黙のプロパティー変更イベント管理が行われるようにすることができる点です。そのため、変更後イメージ・ビジネス・グラフでの、切断されたデルタ・ベースのビジネス・グラフとの相互運用が、大幅に容易になります。

ビジネス・オブジェクト・タイプ・メタデータのモデル化

ビジネス・オブジェクト・タイプ・メタデータを、ビジネス・オブジェクト定義に追加して、実行時に値を拡張することができます。ビジネス・オブジェクト・フレームワークでは、ビジネス・オブジェクト・タイプ・メタデータを設計および変換したり、それらにアノテーションを付けたりすることができます。

ビジネス・オブジェクト・フレームワークは、以下に示すものを可能にする手段を提供します。

- 一貫性のある、比較的介入の少ない方法で、ビジネス・オブジェクトにメタデータを混合する
- 開発者が複雑なアノテーション構造を定義するための規範ポリシー

- ビジネス・オブジェクト開発者およびデプロイヤーが、定義済みの複雑なアノテーション構造に適合するインスタンス・メタデータを使用してビジネス・オブジェクトにアノテーションを付けるための規範ポリシー
- 実行時のアノテーションを、使用しやすい DataObject 構造に変換するための API のセット

このプロセスには、少なくとも 3 つの異なるロールが関わります。

- 1 つ目は、ビジネス・オブジェクト・タイプ・メタデータ設計者のロールです。このロールは、メタデータ構造を設計します。 例えば、ビジネス・オブジェクト・フレームワークは、このロールを担当して、ビジネス・オブジェクトにアノテーションを付けるためのいくつかのメタデータ特性を定義します。
  PeopleSoft、Siebel、および SAP などのアダプターは、メタデータ設計者のロールを担当して、アプリケーション固有情報によってビジネス・オブジェクトにアノテーションを付けることが予期されています。
- 2 つ目のロールは、ビジネス・オブジェクト設計者またはデプロイヤーです。このロールは、ビジネス・オブジェクト・タイプ・メタデータの構造と、ビジネス・オブジェクト・タイプ・メタデータ・フレームワークが定義したポリシーを使用して、メタデータによってビジネス・オブジェクト定義にアノテーションを付けます。
- ビジネス・オブジェクトに正しくアノテーションが付けられた場合、3 つ目のロールが、ビジネス・オブジェクト・メタデータ API を使用して、実行時にメタデータを確認および検査し、それをナビゲーション可能で有用な DataObject グラフ構造にすることができます。

さらに、ビジネス・オブジェクト・フレームワークは、以下のメタデータ機能を提供します。

- 複合 1 次キーおよび外部キー・プロパティー定義
- トップレベルのサポートされる動詞メタデータ・アノテーション

詳しくは、以下の他のトピックを参照してください。

ビジネス・オブジェクト・タイプ・メタデータの表現:

ビジネス・オブジェクト・フレームワークは、ビジネス・オブジェクト・タイプ・メタデータを、トップダウン・スキーマか、外部で作成されてインポートされたスキーマのいずれかに混合できるメカニズムを定義します。

ビジネス・オブジェクト・タイプ・メタデータを混入させるために使用されるメカニズムは、XML スキーマ・アノテーションと `appInfo` 構造です。このポリシーでは、元のスキーマ定義を変更する必要はありますが、アノテーションおよび `appinfo` のビューを容易に切り替えられたり、アノテーションおよび `appinfo` を完全に削除できたりするような方法で、メタデータを混入します。この識別方式は、ビジネス・オブジェクト・タイプ・メタデータを定義するターゲット名前空間の値を持つ、`xs:appinfo` タグのソース属性を使用して実行されます。

例えば、以下のスキーマがランタイムにインポートされたものとします。スキーマは、顧客アノテーションを含む、Product という名前のビジネス・オブジェクトを記述しています。

```
<schema>
 targetNamespace="http://www.scm.com/ProductTypes"
 xmlns:p="http://www.scm.com/ProductTypes"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified

 <complexType name="Product">
  <annotation>
   <appInfo>
    <SCMEditor value="Bottom" type="Anchor"/>
   </appInfo>
   <documentation>
    Describes the SCM Product
   </documentation>
  </annotation>

  <sequence>
   <element name="id" type="ID"/>
   <element name="description" type="string" default="DefaultDescription"/>
   <element name="sku" type="p:Sku"/>
  </sequence>
 </complexType>

 <simpleType name="Sku">
  <restriction base="string">
   <pattern value="¥d{3}-[A-Z]{2}"/>
  </restriction>
 </simpleType>
</schema>
```

ビジネス・オブジェクト・タイプ・メタデータの設計:

ビジネス・オブジェクト・メタデータを収容するメタデータ構造を設計できます。

このタスクについて

メタデータ構造を設計するには、以下のステップを実行します。

手順

1. 有効なターゲット名前空間を持つ XML スキーマ・ファイルを作成します。このステップは、ビジネス・オブジェクト定義にインスタンス・メタデータを追加するときに使用します。
2. 名前付き複合タイプを定義して、ビジネス・オブジェクトに追加できる個々のメタデータをそれぞれ定義します。複合タイプ定義は、インスタンス・メタデータの読み取りに使用されるダイナミック・タイプの DataObject の構造を定義するために使用されます。複合タイプの名前は、ビジネス・オブジェクト定義にインスタンス・メタデータを追加するときに使用します。

例

この例は、PeopleSoft アダプター用に開発されたビジネス・オブジェクト・タイプ・メタデータの一部です。

```
<schema>
 targetNamespace="http://www.ibm.com/xmlns/prod/websphere/adapter/psft/
  PSFTBODefinitionASI/7.0.0"
 xmlns:psft="http://www.ibm.com/xmlns/prod/websphere/adapter/psft/
  PSFTBODefinitionASI/7.0.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
```

```
 elementFormDefault="qualified">

 <complexType name="PSFTBODefinitionASI">
  <sequence>
   <element name="hostname" type="string"/>
   <element name="ipaddress" type="string"/>
  </sequence>
 </complexType>
</schema>
```

ビジネス・オブジェクト定義へのアノテーション:

ビジネス・オブジェクト・フレームワークがサポートする構文を使用して、ビジネス・オブジェクト定義にアノテーションを付けることができます。

このタスクについて

ビジネス・オブジェクト・フレームワークは、インスタンス・メタデータ・データがビジネス・オブジェクト定義に付加されるメカニズムを定義しようとはしません。ただし、インスタンス・メタデータの構文は定義します。

ビジネス・オブジェクト定義にアノテーションを付けるには、以下のガイドラインに従ってください。

手順

1. インスタンス・メタデータを付加すべきビジネス・オブジェクト定義の部分にまだアノテーションが付けられていない場合は、ここで追加します。既にアノテーションがある場合は、その既存のアノテーションを使用します。
2. `xs:annotation` タグ内に別個の `xs:appinfo` タグを作成し、追加するビジネス・オブジェクト・タイプ・メタデータにより定義される名前空間にソース属性を定義します。
3. `xs:appinfo` タグの内部でターゲット名前空間接頭部を使用し、それに複合タイプ定義の名前を続けて、QName を定義します。 `xmlns:` で定義された属性 QName を追加し、前に使用したターゲット名前空間接頭部を続けます。この QName 属性の値を、使用されるビジネス・オブジェクト・タイプ・メタデータのターゲット名前空間に設定します。
4. ビジネス・オブジェクト・タイプ・メタデータ定義により指示された構造とインスタンス・データを追加します。すべてのタグを適切に閉じます。

例

オブジェクトは、インポートされると、一連の PeopleSoft アノテーションを必要とするものとして識別されます。次の例は、PeopleSoft メタデータが追加されたビジネス・オブジェクト定義を示します。

```
<schema>
 targetNamespace="http://www.scm.com/ProductTypes"
 xmlns:p="http://www.scm.com/ProductTypes"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified

 <complexType name="Product">
  <annotation>
   <appinfo source="http://www.ibm.com/xmlns/prod/websphere/Adapter/PSFT">
    <psft:PSFTMetadata
```

```
     xmlns:psft="http://www.ibm.com/xmlns/prod/websphere/Adapter/PSFT">
     <hostname>mumbai</hostname>
     <ipaddress>9.29.1.1</ipaddress>
    <psft:PSFTMetadata>
   </appInfo>
   <appInfo>
    <SCMEditor value="Bottom" type="Anchor"/>
   </appInfo>
   <documentation>
    Describes the SCM Product
   </documentation>
  </annotation>

  <sequence>
   <element name="id" type="ID"/>
   <element name="description" type="string" default="DefaultDescription"/>
   <element name="sku" type="p:Sku"/>
  </sequence>
 </complexType>

 <simpleType name="Sku">
  <restriction base="string">
   <pattern value="¥d{3}-[A-Z]{2}"/>
  </restriction>
 </simpleType>
</schema>
```

DataObjects へのアノテーションの変換:

ビジネス・オブジェクト・フレームワークには、アノテーションを使用可能な DataObject 構造へ変換する機能があります。

このタスクについて

ビジネス・オブジェクト定義に関連付けられたアノテーションは、SDO 実装に固有の一連の API を使用してランタイムに読み取ることができます。しかし、API には、バイナリー・ラージ・オブジェクト (BLOB) を返すという問題があります。ただし、推奨されるアノテーション・パターンに従う場合、ビジネス・オブジェクト・フレームワークが提供するユーティリティーは、BLOB を読み取り、検証して、それを使用可能な DataObject 構造に変換します。

アノテーションを DataObject に変換するには、次のようにします。

手順

1. アノテーションを取得します。
2. BOTypeMetadata を使用して、そのアノテーションを SDO DataObject に変換します。 BOTypeMetadata の実装は、BOTypeMetadata.INSTANCE インスタンスを使用する singleton として使用可能です。

例

次の例は、API を使用してアノテーションを取得し、次に BOTypeMetadata API を使用してそのアノテーションを SDO DataObject に変換する方法を示しています。メタデータは `BOTypeMetadata.xsd` で定義されます。

```
<schema>
 targetNamespace="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
```

```
 elementFormDefault="qualified">

 <complexType name="VerbInfo">
  <sequence>
   <element name="verbInfo" type="string"/>
  </sequence>
 </complexType>
</schema>
```

ビジネス・オブジェクト・フレームワークの可能性の 1 つは、サポートされる追加の動詞および付随するメタデータを追加して、メタ駆動の機能をランタイムに使用可能にする能力です。この機能は、verb および VerbInfo メタデータを使用して、追加のメタデータで動詞にアノテーションを付けることによりサポートされます。次の例は、可能な verb 値のそれぞれについて VerbInfo メタデータが追加される場所を示しています。

```
<schema
 targetNamespace="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
 xmlns:pcbg="http://www.scm.com/ProductCategoryTypes/ProductCategoryBG"
 xmlns:pc="http://www.scm.com/ProductCategoryTypes"
 xmlns:sdo="commonj.sdo"
 xmlns:bo="http://www.ibm.com/xmlns/prod/websphere/bo/7.0.0"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

 <import namespace="http://www.ibm.com/xmlns/prod/websphere/bo/7.0.0"
  schemaLocation="BusinessGraph.xsd"/>

 <import namespace="commonj.sdo"
  schemaLocation="DataGraph.xsd"/>

 <import namespace="http://www.scm.com/ProductCategoryTypes"
  schemaLocation="ProductCategoryTypes.xsd"/>

 <complexType name="ProductCategoryBG">
  <complexContent>
   <extension base="bo:BusinessGraph">
    <sequence>

     <element name="verb"
      minOccurs="0" maxOccurs="1">
      <simpleType>
       <restriction base="string">
        <enumeration value="Create">
         <annotation>
          <appinfo source="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
           <botm:VerbInfo
            xmlns:botm="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
            <verbInfo>Metadata relating to Create</verbInfo>
           </botm:VerbInfo>
          </appinfo>
         </annotation>
        </enumeration>
        <enumeration value="Retrieve">
         <annotation>
          <appinfo source="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
           <botm:VerbInfo
            xmlns:botm="http://www.ibm.com/xmlns/prod/websphere/botm/7.0.0">
            <verbInfo>Metadata relating to Retrieve</verbInfo>
           </botm:VerbInfo>
          </appinfo>
         </annotation>
        </enumeration>
       </restriction>
```

```
      </simpleType>
     </element>

     <element name="productCategory" type="pc:ProductCategory"
      minOccurs="0" maxOccurs="1"/>

    </sequence>
   </extension>
  </complexContent>
 </complexType>

 <element name="productCategoryBG" type="pcbg:ProductCategoryBG"/>

</schema>
```

ビジネス・オブジェクト・サービスを使用したプログラミング

ビジネス・オブジェクトの作成と操作を容易にするため、ビジネス・オブジェクト・フレームワークは一連の Java サービスを提供することによって SDO 仕様を拡張します。これらのサービスは、`com.ibm.websphere.bo` という名前のパッケージに含まれています。

以下に、ビジネス・オブジェクト・サービスの簡略説明を示します。

表 22. ビジネス・オブジェクト・サービス

サービス	説明
BOChangeSummary	ビジネス・グラフの変更の要約ヘッダーを管理するための、SDO 変更の要約インターフェースへの機能拡張を提供します。
BOCopy	ビジネス・オブジェクトのグラフのコピー、またはビジネス・オブジェクトのグラフを含むビジネス・グラフのコピーを容易に実行できます。
BODataObject	SDO Data Object インターフェースへの機能拡張を提供します。
BOEquality	2 つのビジネス・グラフまたはビジネス・オブジェクトが等価であるかどうかを判別できます。
BOEventSummary	ビジネス・グラフのイベントの要約ヘッダーの内容を管理するためのインターフェースを提供します。
BOFactory	ビジネス・グラフまたはビジネス・オブジェクトを作成する機能を提供します。
BOType	`Class.forName()` が Java クラス名として提供するものを反映する、ビジネス・グラフまたはビジネス・オブジェクトの SDO タイプを取得するメカニズムを提供します。
BOTypeMetadata	アノテーション・バイナリー・ラージ・オブジェクト (BLOB) を取り込む機能を提供します。アノテーション BLOB は、BO タイプ・メタデータ・パターンに準拠し、そのパターンを一連の SDO DataObject に変換 (および逆方向の変換を実行) します。

表 22. ビジネス・オブジェクト・サービス (続き)

サービス	説明
BOXMLDocument/BOXMLSerializer	メモリーで XML 文書を作成して表現するためのメカニズムと、XML 文書を直列化および非直列化するメカニズムを提供します。
BOInstanceValidator	ビジネス・オブジェクト・インスタンスを、その XSD 定義と対照して検証する機能を提供します。ビジネス・オブジェクトは、さまざまな形式をとることができます。単純なビジネス・オブジェクトである場合も、機能を充実させたビジネス・グラフ・モデルでラップされている場合もあります。特定のビジネス・インテグレーション・シナリオでは、ビジネス・オブジェクトは変更の要約の削除済みセクションにあります。これらのビジネス・オブジェクトは、下流のビジネス・ロジックを駆動します。すべてのケースでビジネス・オブジェクトの正確性が保証されている必要があります。BOInstanceValidator には 2 つのスタイルがサポートされています。 • 明示的なプログラマチック検証: 一連のプログラミング API を通じてビジネス・オブジェクトを検証するシステム・サービスが提供されています。 • 暗黙的なインターフェース検証: この検証は、SCA インターフェース修飾子を通じて、ターゲット・インターフェースで WebSphere Integration Developer によって使用可能/使用不可に設定されます。

XML 文書の妥当性検査

XML 文書およびビジネス・オブジェクトは、検証サービスを使用して検証することができます。

また、他のサービスでは一定の最低基準が必要で、満たさなければランタイム例外をスローします。これらの 1 つが BOXMLSerializer です。

BOXMLSerializer を使用すれば、XML 文書がサービス要求によって処理される前に検証することができます。BOXMLSerializer は XML 文書の構造を検証して、以下の種類のエラーがあるか判断します。

- 無効な XML 文書 (エレメント・タグが欠落しているものなど)。
- 整形式 XML 文書でないもの (クローズ・タグが欠落しているものなど)。
- 構文解析エラー (エンティティー宣言のエラーなど) を含む文書。

エラーが BOXMLSerializer によって検出されると、問題の詳細情報とともに例外がスローされます。

妥当性検査は、以下のサービスについての XML 文書のインポートまたはエクスポート、あるいはその両方に対して実行できます。

- HTTP
- JAXRPC Web サービス
- JAX-WS Web サービス
- JMS サービス
- MQ サービス

HTTP、JAXRPC、および JAX-WS サービスの場合、BOXMLSerializer は以下の方法で例外を生成します。

- インポート -
  1. SCA コンポーネントはサービスを呼び出します。
  2. サービスは宛先 URL を呼び出します。
  3. 宛先 URL は無効な XML 例外で応答します。
  4. サービスは失敗して、ランタイム例外およびメッセージが出されます。
- エクスポート -
  1. サービス・クライアントはサービス・エクスポートを呼び出します。
  2. サービス・クライアントは無効な XML を送信します。
  3. エクスポートのサービスが失敗して、例外およびメッセージが生成されます。

JMS および MQ メッセージング・サービスの場合、例外は以下の方法で生成されます。

- インポート -
  1. インポートは JMS または MQ サービスを呼び出します。
  2. サービスは応答を戻します。
  3. サービスは無効な XML 例外を戻します。
  4. インポートが失敗して、メッセージが生成されます。
- エクスポート -
  1. MQ または JMS クライアントはエクスポートを呼び出します。
  2. クライアントは無効な XML を送信します。
  3. エクスポートが失敗して、例外およびメッセージが生成されます。

XML 検証の例外によって生成されたメッセージは、ログを表示して確認できます。以下の例は、BOXMLSerializer によって検証され、XML コーディングが正しくないために生成されたメッセージです。

- JAXWS インポート

```
javax.xml.ws.WebServiceException: org.apache.axiom.om.OMException:
 javax.xml.stream.XMLStreamException: Element type "TestResponse" must be
 followed by either attribute specifications, ">" or "/>".

javax.xml.ws.WebServiceException: org.apache.axiom.soap.SOAPProcessingException:
 First Element must contain the local name, Envelope
```

- JAXRPC インポート

```
[9/11/08 15:16:27:417 CDT] 0000003e ExceptionUtil E
 CNTR0020E: EJB threw an unexpected (non-declared)
  exception during invocation of method
  "transactionNotSupportedActivitySessionNotSupported" on bean
  "BeanId(WSXMLValidationApp#WSXMLValidationEJB.jar#Module, null)".
```

```
 Exception data: WebServicesFault
faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
faultString: org.xml.sax.SAXParseException: Element type "TestResponse"
  must be followed by either
 attribute specifications, ">" or "/>". Message being parsed:
   <?xml version="1.0"?><TestResponse
 xmlns="http://WSXMLValidation"<firstName>Bob</firstName>
   <lastName>Smith</lastName></TestResponse>
faultActor: null
faultDetail:
[9/11/08 15:16:35:135 CDT] 0000003f ExceptionUtil E   CNTR0020E: EJB threw an
  unexpected (non-declared) exception during invocation of method
  "transactionNotSupportedActivitySessionNotSupported" on bean
  "BeanId(WSXMLValidationApp#WSXMLValidationEJB.jar#Module, null)".
 Exception data: WebServicesFault
 faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultString: org.xml.sax.SAXException: WSWS3066E: Error: Expected 'envelope'
   but found  TestResponse
  Message being parsed: <?xml version="1.0"?><TestResponse
   xmlns="http://WSXMLValidation">
  <firstName>Bob</firstName><middleName>John</middleName>
   <lastName>Smith</lastName>
   </TestResponse>
 faultActor: null
 faultDetail:
```

- JAXRPC/JAXWS エクスポート

```
[9/11/08 15:35:13:401 CDT] 00000064 WebServicesSe E
  com.ibm.ws.webservices.engine.transport.http.WebServicesServlet
  getSoapAction WSWS3112E:
   Error: Generating WebServicesFault due to missing SOAPAction.
                    WebServicesFault
faultCode: Client.NoSOAPAction
faultString: WSWS3147E: Error: no SOAPAction header!
faultActor: null
faultDetail:
```

検証サービスについて詳しくは、『Reference』セクションの『Generated API and SPI』の資料にある BOInstanceValidator インターフェースの項目を参照してください。

プログラミング手法

以下の手法は、ビジネス・オブジェクト・フレームワークを使用して効果的にビジネス・オブジェクトをプログラミングする方法を示すものです。

ビジネス・オブジェクト内の配列

ビジネス・オブジェクト内のエレメントに配列を定義して、エレメントに複数のデータ・インスタンスを含めることができます。

リスト型を使用して、ビジネス・オブジェクト内の単一の名前付きエレメントに配列を作成することができます。これによって、このエレメントに複数のデータ・インスタンスを格納することができます。例えば、ビジネス・オブジェクト・ラッパー内にストリングとして定義した telephone という名前のエレメント内に、配列を使用して複数の電話番号を保管することができます。また、maxOccurs の値を使用してデータ・インスタンスの数を指定することによって、配列のサイズを定義することもできます。次のコード例は、エレメントに 3 つのデータ・インスタンスを保持する配列を作成する方法を示したものです。

```
<xsd:element name="telephone" type="xsd:string" maxOccurs="3"/>
```

これは、最大 3 つのデータ・インスタンスを保持できるエレメント telephone のリスト指標を作成します。項目が 1 つだけの配列を計画している場合は、値 minOccurs も使用できます。

結果の配列は、以下の 2 つの項目で構成されます。

- 配列の内容
- 配列そのもの

ただし、この配列を作成するには、ラッパーを定義して中間ステップを実行する必要があります。このラッパーは実際には、エレメントのプロパティーを配列オブジェクトで置換します。上記の例では、ArrayOfTelephone オブジェクトを作成して、エレメント telephone を配列として定義することができます。次のコード例は、このタスクを行う方法を示したものです。

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="Customer">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="ArrayOfTelephone" type="ArrayOfTelephone"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="ArrayOfTelephone">
      <xsd:sequence maxOccurs="3">
        <xsd:element name="telephone" type="xsd:string" nillable="true"/>
      </xsd:sequence>
    </xsd:complexType>
 </xsd:schema>
```

telephone エレメントは、ArrayOfTelephone ラッパー・オブジェクトの子として表現されています。

上の例では、telephone エレメントには nillable という名前のプロパティーが組み込まれています。配列指標の特定項目にデータを含めたくない場合は、このプロパティーを true に設定することができます。次のコード例は、配列内でのデータの表現方法を示したものです。

```
<Customer>
   <name>Bob</name>
   <ArrayOfTelephone>
     <telephone>111-1111</telephone>
     <telephone xsi:nil="true"/>
     <telephone>333-3333</telephone>
   </ArrayOfTelephone>
 </Customer>
```

このケースでは、telephone エレメントの配列指標の 1 番目と 3 番目の項目にはデータが格納されていますが、2 番目の項目にはデータが格納されていません。nillable プロパティーを telephone エレメントで使用しなかった場合、最初の 2 つのエレメントにデータを格納する必要があります。

WebSphere Process Server のサービス・データ・オブジェクト (SDO) の Sequence API を、ビジネス・オブジェクト配列内のシーケンスを処理する代替の方法として使用することができます。以下のコード例によって、上記で示したものと同じデータを持つ telephone エレメントの配列が作成されます。

```
DataObject customer = ...
 customer.setString("name", "Bob");

 DataObject tele_array = customer.createDataObject("ArrayOfTelephone");
 Sequence seq = tele_array.getSequence();  // 配列は順序付けされます
 seq.add("telephone", "111-1111");
 seq.add("telephone", null);
 seq.add("telephone", "333-3333");
```

以下の例のようなコードを使用して、特定のエレメント配列指標のデータを戻すことができます。

```
String tele3 = tele_array.get("telephone[3]");  // tele3 = "333-3333"
```

この例では、tele3 という名前のストリングによって、データ「333-3333」が戻されます。

リスト指標の配列のデータ項目には、JMS または MQ メッセージ・キュー内に配置されている固定幅または区切り文字で区切られているデータを使用して入力できます。また、適切にフォーマット設定されたデータが格納されているフラット・テキスト・ファイルを使用して、このタスクを行うこともできます。

ネストされたビジネス・オブジェクトの作成

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトを親ビジネス・オブジェクト内に作成することができます。

中間の子オブジェクトの詳細を記載したコードを記述せずに、ネストされたビジネス・オブジェクトを親ビジネス・オブジェクトから作成できます。例えば、親ビジネス・オブジェクトの 1 レベル下に依存ビジネス・オブジェクトを定義せずに、親ビジネス・オブジェクトの 2 レベル下にネストされたビジネス・オブジェクトを設定できます。以下について、setWithCreate 機能を使用してこのタスクを実行します。

- 単一インスタンス
- 複数インスタンス
- ワイルドカード値
- モデル・グループ

以下のトピックでは、これらのそれぞれについて実行する方法を説明します。

ネストされたビジネス・オブジェクトの単一インスタンス:

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの単一インスタンスを作成します。

始める前に

以下のコード例は、第 3 レベル (孫) オブジェクトを作成するために、高レベル (親) オブジェクトから中間 (子) オブジェクトのコードを通常であればどのように作成する必要があるのかを示したものです。XSD ファイルは以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
```

```
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

このタスクについて

従来の「トップダウン」方式を使用してビジネス・オブジェクト・データを設定する場合、孫オブジェクトにデータを設定する前に、子オブジェクトおよび孫オブジェクトを指定する以下のコードを処理する必要があります。

```
DataObject parent = ...
DataObject child = parent.createDataObject("child");
DataObject grandchild = child.createDataObject("grandChild");
grandchild.setString("name", "Bob");
```

setWithCreate 機能の使用によって、より効率的な方式を使用することができます。この方式では、孫オブジェクトの定義およびオブジェクトのデータの設定を同時に実行でき、中間の子オブジェクトの指定は不要です。次のコード例は、このタスクを行う方法を示したものです。

```
DataObject parent = ...
parent.setString("child/grandchild/name", "Bob");
```

タスクの結果

中間レベルのビジネス・オブジェクトを参照することなく、下位のビジネス・オブジェクト・データが設定されます。パスが無効な場合は例外が発生します。

ネストされたビジネス・オブジェクトの複数インスタンスの作成:

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの複数インスタンスを作成します。

始める前に

以下の XSD ファイルの例には、トップ (親) のビジネス・オブジェクトの 1 レベル下 (子) および 2 レベル下 (孫) のネストされたオブジェクトが含まれています。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
```

```
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child" maxOccurs="5"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

maxOccurs の値で指定するように、親オブジェクトは最大 5 つの子オブジェクトを持つことができることに注意してください。

このタスクについて

配列内で欠落したシーケンスを許可しない、より厳密なポリシーでリストを作成することができます。setWithGet メソッドを使用して、同時に特定のリスト指標項目に現れるデータを指定します。

```
DataObject parent = ...
parent.setString("child[3]/grandchild/name", "Bob");
```

この事例では、結果の配列のサイズは 3 ですが、child[1] および child[2] のリスト指標項目の値は未定義です。項目は、ヌル値にするか、または関連したデータ値を持たせることが必要な場合があります。上記のシナリオでは、最初の 2 つの配列指標項目が未定義であるため、例外がスローされます。

リストの指標の値を定義することで、この状況に対処することができます。指標項目が配列内の既存のエレメントを参照しており、その参照先エレメントがヌル以外である場合 (つまりデータを含む場合)、それが使用されます。参照先エレメントがヌルである場合、そのエレメントが作成されて使用されます。リストの指標がリストのサイズよりも 1 つ大きい場合、新規の値が作成されて追加されます。以下のコード例では、サイズが 2 のリストで、child[1] をヌルに指定し、child[2] にデータを格納した場合に何が起こるかを示したものです。

```
DataObject parent = ...
// child[1] = ヌル
// child[2] = 既存の子
// child[1] はヌルであり、作成されることから、このコードは機能します。
parent.setString("child[1]/grandchild/name", "Bob");

// child[2] が存在して使用されるため、このコードは機能します。
parent.setString("child[2]/grandchild/name", "Dan");

// 子のリストのサイズは 2 であり、
// リスト項目を 1 つ追加するとリストのサイズが増加するため、
// このコードは機能します。
parent.setString("child[3]/grandchild/name", "Sam");
```

タスクの結果

2 つの既存項目の値を指定変更して、3 つ目の項目をリスト指標に追加しました。ただし、次にサイズが 4 でない別の項目か、または maxOccurs で指定したサイズよりも大きな項目を追加すると、例外がスローされます。このメソッドのより厳密なポリシーについて、以下のコード例で示します。

注: 以下のコードは、上記の既存のコードに追加することを想定しています。

```
// リストはサイズが 3 であり、
// サイズを 4 に増加させる項目を作成していなかったため、
// このコードは例外をスローします。
parent.setString("child[5]/grandchild/name", "Billy");
```

ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用:

タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

このタスクについて

サービス・データ・オブジェクト内でワイルドカードの値 xsd:any を使用している場合、単一および複数のインスタンスのためにネストされたビジネス・オブジェクトを定義する目的で使用される setWithCreate 機能は作用しません。これについて、以下のコード例で説明します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="xsd:anyType"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

タスクの結果

子データ・オブジェクトが存在しない場合、例外がスローされます。

モデル・グループ内のビジネス・オブジェクトの使用:

モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを作成します。

このタスクについて

モデル・グループでは、親ビジネス・オブジェクトからビジネス・オブジェクトを作成するのに使用できるタグ xsd:choice を使用します。しかし、ビジネス・オブジェクト・フレームワークでは、例外を生成する可能性がある名前の競合を起こす場合があります。以下のコード例は、名前の競合がどのように発生するかを示したものです。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
```

```
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

注: 「child1」および「child2」という名前のエレメントについて、複数のインスタンスが存在する可能性があります。

これらの競合を解決するには、モデル・グループ用にサービス・データ・オブジェクト (SDO) パス・パターンを使用します。

タスクの結果

以下のコード例で示すように、モデル・グループの処理に使用される SDO パス・パターンを使用した配列が取得されます。

```
set("child1/grandchild/name", "Bob");

set("child11/grandchild/name", "Joe");
```

同じ名前がついたエレメントの差別化

ビジネス・オブジェクトのエレメントと属性には、固有の名前を指定する必要があります。

サービス・データ・オブジェクト (SDO) フレームワークでは、エレメントと属性はプロパティーとして作成されます。以下のコード例では、XSD で foo という名前の 1 つのプロパティーを持つタイプを作成します。

```
<xsd:complexType name="ElementFoo">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AttributeFoo">
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>
```

これらの事例では、XML パス言語 (XPath) を使用してプロパティーにアクセスできます。ただし、有効なスキーマ・タイプは、次の例に示すように、同じ名前の属性とエレメントを持つことができます。

```
<xsd:complexType name="DuplicateNames">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>
```

XPath では、同じ名前の付いたエレメントと属性を区別できなければなりません。そのためには、一方の名前をアットマーク (@) で始めます。次の断片コードは、同じ名前の付いたエレメントと属性にアクセスする方法を示しています。

```
1 DataObject duplicateNames = ...

2 // 「elem_value」を表示する
3 System.out.println(duplicateNames.get("foo"));

4 // 「attr_value」を表示する
5 System.out.println(duplicateNames.get("@foo"));
```

この命名方式は、SDO XPath である String 値を使用するすべてのメソッドに使用してください。

モデル・グループ・サポート (all、choice、sequence、および group 参照):

SDO 仕様は、モデル・グループ (all、choice、sequence、および group 参照) を正しい位置に展開して、タイプやプロパティーを記述しないことを必要としています。

基本的に、これは同じ収容構造の中にある構造は、すべて「フラット化される」ことを意味しています。この「フラット化」により、すべての子構造が同じレベルになり、フラット化されたデータから構造が派生した SDO 内で、名前の重複の問題が生じる可能性があります。XSD がグループをフラット化しない場合でも、さまざまな親に含まれている重複が個別に存在します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

option1 と option2 の複数のオカレンスは異なる choice ブロックに含まれており、それらの間には分離するエレメントさえあるため、XSD および XML は、それらを区別するのに何の問題もありません。しかし、SDO によってこれらのグループがフラット化されると、すべてのオプション・プロパティーは MultipleGroup という同じコンテナーに含まれます。

たとえ重複する名前がなくても、これらのグループをフラット化したことによる意味構造の問題もあります。例えば、以下の XSD を考えてみましょう。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://SimpleChoice">
  <xsd:complexType name="SimpleChoice">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
```

```
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

重複する名前の変更や XSD への特別な注釈の追加をユーザーに求めるのは実際的ではありません。なぜなら、多くの場合、規格や業界のスキーマのように、ユーザーは作業に使用している XSD を制御できないからです。

すべてのプロパティーに一貫性を持たせるために、ビジネス・オブジェクトは、重複した名前を持つプロパティーの個々のオカレンスに XPath を通じてアクセスするメソッドを含んでいます。ビジネス・オブジェクト・フレームワークの命名規則に従い、重複するプロパティー名が検出された場合、それらの名前に次の未使用の数字が付加されます。例として、次の XSD の場合を考えます。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://TieredGroup">
  <xsd:complexType name="TieredGroup">
    <xsd:sequence>
      <xsd:choice minOccurs="0">
        <xsd:sequence>
          <xsd:element name="low" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
          <xsd:choice minOccurs="0">
            <xsd:element name="width" minOccurs="0"
      maxOccurs="1" type="xsd:string"/>
            <xsd:element name="high" minOccurs="0"
      maxOccurs="1" type="xsd:string"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:element name="high" minOccurs="1"
    maxOccurs="1" type="xsd:string"/>
          <xsd:sequence>
            <xsd:element name="width" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
            <xsd:element name="high" minOccurs="0"
     maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="center" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
          <xsd:element name="width" minOccurs="0"
     maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

先行する XSD は次の DataObject モデルを生成します。

```
DataObject - TieredGroup
 Property[0] - low - string
 Property[1] - width - string
 Property[2] - high - string
 Property[3] - high1 - string
 Property[4] - width1 - string
 Property[5] - high2 - string
 Property[6] - center - string
 Property[7] - width2 - string
```

ここで、**width**、**width1**、および **width2** は width というプロパティーの名前で、この XSD 内の最初のものから始まります。**high**、**high1**、**high2** も同様です。

これらの新しいプロパティー名は、参照と XPath に使用される名前にすぎず、シリアライズされた内容に影響しません。シリアライズされた XML 内に現れるこれらの各プロパティーの「真の」名前は、XSD 内で指定された値です。したがって、次の XML インスタンスの場合:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:TieredGroup xsi:type="p:TieredGroup"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://TieredGroup">
  <width>foo</width>
  <high>bar</high>
</p:TieredGroup>
```

これらのプロパティーにアクセスするためには、以下のコードを使用します。

```
DataObject tieredGroup = ...

// 「foo」を表示する
System.out.println(tieredGroup.get("width1"));

// 「bar」を表示する
System.out.println(tieredGroup.get("high2"));
```

同じ名前がついたプロパティーの差別化

同じ名前空間を持つ複数の XSD で同じ名前のタイプが定義された場合、偶然に誤ったタイプが参照される可能性があります。

Address1.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="city" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

Address2.xsd:
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="state" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

ビジネス・オブジェクトは BOFactory.create() API を通じて、グローバル XSD 構造 (complexType、simpleType、element、attribute など) の重複名をサポートしていません。それでも、これらの重複したグローバル構造は、以下の例に示すように、適正な API が使用された場合でも、他の構造に対する子として作成される可能性があります。

Customer1.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Customer1"
 targetNamespace="http://Customer1">
  <xsd:import schemaLocation="./Address1.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="0" name="address" type="Address"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Customer2.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Customer2"
 targetNamespace="http://Customer2">
  <xsd:import schemaLocation="./Address2.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="address" type="Address"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

両方の Customer address フィールドにデータを取り込んだ後、Address を作成するために BOFactory.create() を呼び出すと、結果としての子ビジネス・オブジェクトが誤って設定される可能性があります。これを避けるには、Customer DataObject に対して createDataObject("address") API を呼び出します。これにより、ビジネス・オブジェクトは import の schemaLocation に従うため、正しいタイプの子が生成されます。

```
DataObject customer1 = ...

// Address の子を作成する誤った方法
// これは Address1.xsd Address または Address2.xsd Address のタイプを
// 作成する場合がある
DataObject incorrect = boFactory.create("", "Address");
customer1.set("address", incorrect);

// Address の子を作成する正しい方法
// これにより Address1.xsd Address のタイプが作成されることが保証される
customer1.createDataObject("address");
```

ピリオドを含んでいるプロパティー名の解決

XSD 内のプロパティー名には、多数の有効な文字の 1 つとしてピリオド (.) を含めることができますが、SDO では、ピリオドは複数カーディナリティーのプロパティー内で索引付けを示すためにも使用されます。このため、特定のシチュエーションで解決の問題が起きることがあります。

サービス・データ・オブジェクト (SDO) 内のプロパティー名は、XSD 内で生成されたエレメントと属性の名前に基づいています。ビジネス・オブジェクトは「.」文字を正しく処理しますが、1 つ例外があります。それは、XSD に「<name>.<#>」という名前の単一カーディナリティー・プロパティーと、「<name>」という名前の複数カーディナリティー・プロパティーがある場合です。

「foo.0」などの XPath は、「foo.0」という名前の単一カーディナリティー・プロパティーと「foo」という名前の複数カーディナリティー・プロパティーが存在する場合、正しく解決されません。その場合、「foo.0」という名前の単一カーディナリティー ・プロパティーに解決されます。これはまれにしか起きませんが、複数カーディナリティー・プロパティーにアクセスする場合に「foo[1]」の構文を使用すると、完全に回避できます。SDO は、索引付けに「.」構文をサポートしないので、索引付けには「[]」を使用してください。

xsi:type での共用体のシリアライズとデシリアライズ:

XSD では、共用体とは、メンバーと呼ばれるいくつかの単純データ型の字句スペースをマージする方法のことです。

以下の XSD の例は、整数と日付をメンバーとする共用体を示しています。

```
<xsd:simpleType name="integerOrDate">
 <xsd:union memberTypes="xsd:integer xsd:date"/>
</xsd:simpleType>
```

この複数のタイプ指定により、デシリアライゼーション時およびデータの操作時に混乱が起きる可能性があります。

ビジネス・オブジェクトは SDO でシリアライゼーションに xsi:type を使用することをサポートしており、デシリアライゼーションで XML データ内に xsi:type が存在しない場合は、同じアルゴリズムに従ってタイプを決定します。

したがって、データ (この例では数値の「42」) が整数としてデシリアライズされることを保証するために、入力 XML で指定された xsi:type を使用できます。また、整数がストリングの前に来るよう、XSD 内で共用体のメンバー・リストを順序付けることもできます。以下の例では、両方の方式の実装方法を示します。

```
<integerOrString xsi:type="xsd:integer">42</integerOrString>

<xsd:simpleType name="integerOrString">
 <xsd:union memberTypes="xsd:integer xsd:string"/>
</xsd:simpleType>
```

同様に、ユーザーがデータをストリングとしてデシリアライズさせる場合は、以下のいずれかの変更によって、そのような動作を起こさせることができます。

```
<integerOrString xsi:type="xsd:string">42</integerOrString>

<xsd:simpleType name="integerOrString">
 <xsd:union memberTypes="xsd:string xsd:integer"/>
</xsd:simpleType>
```

string タイプがこの共用体の最初のメンバーである場合は、決して情報が損失しないことに注意してください。また、この共用体は、xsi:type 以外のアルゴリズムで常に選択されるすべてのデータも保持できます。string 以外のタイプを使用する場合は、XML の中で xsi:type を使用するか、XSD でメンバー・タイプの順序を変更して、他のメンバーがデータを受け入れられるようにする必要があります。

NULL ビジネス・オブジェクトのサポート

このシナリオでは、SOAP メッセージ内にラップした XML を使って WebSphere Process Server と通信する外部システムが必要です。含まれているエレメントが NULL を指定可能で、`xsi:nil="true"` の場合は、WebSphere Process Serverで作成された DataObject は NULL となります。

下の例は、NULL が指定可能なエレメントを持つ XML メッセージを示します。

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
 <soap:Body>
```

```
  <p:Employee xmlns:p="http://www.mycompany.com" xmlns:xsi="http://www.w3.org"
   xsi:nil="true"/>
 </soap:Body>
</soap:Envelope>
```

ここで、Employee は次のように定義されます。

```
<element name="Employee" nillable="true">
...
</element>
```

エクスポートから生成、送信されたビジネス・オブジェクトは、このケースでは NULL です。例えば、任意のダウンストリーム・コンポーネントにそこで呼び出された操作がある場合は、その操作への入力は NULL となります。

注: ビジネス・オブジェクト・マップは NULL オブジェクトからフィールドをマップできないため、これらのオブジェクトは、ビジネス・オブジェクト・マップに渡すことはできません。

Sequence オブジェクトを使用したデータの順序の設定

一部の XSD は、XML 内でのデータの出現順序が特別な重要性を持つような方法で定義されます。

XSD での順序の重要度を示す例の 1 つは、混合内容です。テキスト・データが、あるエレメントの前か後に出現する場合、そのデータは、別の場所で出現した場合とは異なる意味を持つことがあります。そのようなシチュエーションでは、SDO は Sequence というオブジェクトを生成します。このオブジェクトは、順序を付けてデータを設定するために使用されます。

SDO Sequence を XSD シーケンスと混同しないでください。XSD シーケンスは、SDO モデルの生成の前にフラット化される単なるモデル・グループです。XSD シーケンスの存在は、SDO Sequence の存在とは関係ありません。

XSD 内の以下の条件により、SDO Sequence が生成されます。

混合内容を持つ complexType:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://MixedContent"
 targetNamespace="http://MixedContent">
  <xsd:complexType name="MixedContent" mixed="true">
    <xsd:sequence>
      <xsd:element name="element1" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element2" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element3" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="MixedContent" type="tns:MixedContent"/>
</xsd:schema>
```

1 つ以上の <any/> タグを持つスキーマ:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyElemAny"
 targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
      <xsd:any/>
```

```
      <xsd:element name="marker1" type="xsd:string"/>
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

モデル・グループ配列 (maxOccurs > 1 の all、choice、sequence、または group 参照):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:sequence maxOccurs="3">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

複数のエレメントを含んでいる maxOccurs <= 1 の <all/> タグ:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://All">
  <xsd:complexType name="All">
    <xsd:all>
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

<any/> と sequence を一緒に使用することについての詳細は、このページの下部にリストしたトピックの中で説明します。このセクションで以下の残りの部分に示す一般情報は、その他の Sequence 条件の処理方法を説明したものですが、<any/> にも当てはまります。

どうすれば DataObject にシーケンスがあるかどうかが分かるか:

DataObject にシーケンスがあるかどうかを判別できる API は 2 つのうちから選択できます。DataObject noSequence と DataObject withSequence です。

DataObject noSequence および DataObject withSequence は、次の例のように使用します。

```
DataObject noSequence = ...
DataObject withSequence = ...

// false を表示する
System.out.println(noSequence.getType().isSequenced());

// true を表示する
System.out.println(withSequence.getType().isSequenced());

// true を表示する
System.out.println(noSequence.getSequence() == null);

// false を表示する
System.out.println(withSequence.getSequence() == null);
```

なぜ DataObject に Sequence があることを知っていなければならないのか:

Sequence がある DataObject に対して作業をする場合、データが設定される順序を知ることが重要です。そのため、値が設定される順序に注意する必要があります。

順序付けされていない DataObject では、ランダムな順序での設定アクセスができます。これは、すべてのキーが同じ値に設定される Map のように機能します。キーが設定された順序は問題ではなく、マップ内のデータは同じものであり、XML へも完全に同じようにシリアライズされます。

DataObject が順序付けされている場合、データが設定された順序は、List へのデータの追加と同様に、Sequence 内に記録されます。これは、データへの 2 とおりのアクセス方法を提供します。名前/値のペアによるアクセス (DataObject API) と、設定された順序によるアクセス (Sequence API) です。DataObject set(...) API または Sequence add(...) API を使用して、構造を保存できます。この順序付けは、XML がシリアライズされる方法に影響を及ぼします。

例として、以下の <all/> タグを考えてみます。set メソッドが以下の順序で呼び出された場合、シリアライズされると、以下の XML が生成されます。

```
 DataObject all = ...
 all.set("element1", "foo");
 all.set("element2", "bar");

<?xml version="1.0" encoding="UTF-8"?>
<p:All xsi:type="p:All"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:p="http://All">
  <element1>foo</element1>
  <element2>bar</element2>
</p:All>
```

代わりに、set メソッドが逆の順序で呼び出された場合は、このビジネス・オブジェクトがシリアライズされると、以下の XML が生成されます。

```
 DataObject all = ...
 all.set("element2", "bar");
 all.set("element1", "foo");

<?xml version="1.0" encoding="UTF-8"?>
<p:All xsi:type="p:All"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:p="http://All">
  <element2>bar</element2>
  <element1>foo</element1>
</p:All>
```

Sequence の順序を変更する必要がある場合、Sequence クラスにはユーザーが Sequence の順序を変更できるよう、基本的な add、remove、および move メソッドがあります。

どのようにして混合内容を処理するか:

混合内容の場合、Sequence にはテキストを追加するための固有の API である addText(...) があります。

それ以外のすべての API は、テキストに対して Property の場合と同じように機能します。getProperty(int) API は、混合内容テキスト・データについて、ヌルを返します。以下の混合内容コードの例は、DataObject からのすべての混合内容テキストを印刷するために使用できます。

```
DataObject mixedContent = ...
Sequence seq = mixedContent.getSequence();

for (int i=0; i < seq.size(); i++)
{
    Property prop = seq.getProperty(i);
    Object value = seq.getValue(i);

    if (prop == null)
    {
        System.out.println("Found mixed content text: "+value);
    }
    else
    {
        System.out.println("Found Property "+prop.getName()+": "+value);
    }
}
```

どのようにしてモデル・グループ配列を処理するか:

モデル・グループ配列は、モデル・グループが maxOccurs > 1 の値を持っているときに作成されます。

モデル・グループはフラット化され、DataObject の中で表現されないため、モデル・グループの内部のプロパティーは複数カーディナリティー・プロパティーになり、これにより isMany() メソッドは、まだ true でなくても true を返します。それらの minOccurs および maxOccurs ファセットは、収容しているモデル・グループのファセットによって乗算されます。Choice は、他のモデル・グループと同じように maxOccurs ファセットを乗算しますが、minOccurs の乗算値として常に 0 を使用します。なぜなら、choice 内のいずれかのデータは選択されない場合があるからです。

例えば、以下の XSD にはモデル・グループ配列があります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:sequence minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
   minOccurs="0" maxOccurs="3"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

前に述べたように、**element1** と **element2** は get(...) アクセサーが List を返すよう、この時点で複数カーディナリティーになります。**Element1** の minOccurs はデフォルトの 1 で、maxOccurs もデフォルトの 1 です。**Element2** の minOccurs は 0 で、maxOccurs は 3 です。次の例では、それらの新しい minOccurs と maxOccurs は、以下のようになります。

```
DataObject - ModelGroupArray
 Property[0] - element1 - minOccurs=(2*1)=2 - maxOccurs=(5*1)=5
 Property[1] - element2 - minOccurs=(2*0)=0 - maxOccurs=(5*3)=15
```

タイプが Choice だとすると:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
```

```
  <xsd:complexType name="ModelGroupArray">
    <xsd:choice minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
   minOccurs="0" maxOccurs="3"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

は、以下の minOccurs を生成します。これは、**element1** だけが毎回選択されるか **element2** だけが毎回選択される choice の除外によるもので、したがって、妥当性検査をパスするためには、どちらも 0 個のオカレンスを持つことができなければなりません。

```
DataObject - ModelGroupArray
 Property[0] - element1 - minOccurs=(0*1)=0 - maxOccurs=(5*1)=5
 Property[1] - element2 - minOccurs=(0*0)=0 - maxOccurs=(5*3)=15
```

Any データ型の使用

このセクションでは、Any データ型を使用するためのプログラミング手法を説明します。

単純タイプへの AnySimpleType の使用:

SDO API による AnySimpleType の処理は、他のいずれの単純タイプ (string、int、boolean など) とも異なる点はありません。

anySimpleType が他の単純タイプと異なるのは、そのインスタンス・データとシリアライゼーション/デシリアライゼーションだけです。これらはビジネス・オブジェクトの内部だけの概念にすべきであり、フィールドへまたはフィールドからマップされるデータが有効であるかどうかを判別するために使用してください。string タイプに対して set(...) メソッドを呼び出す場合は、最初にデータがストリングに変換され、元のデータ・タイプは失われます。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://StringType">
  <xsd:complexType name="StringType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject stringType = ...

 // データを String に設定する
 stringType.set("foo", "bar");

 // インスタンス・データは、データ・セットに関係なく、常に String タイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());

 // データを Integer に設定する
 stringType.set("foo", new Integer(42));

 // インスタンス・データは、データ・セットに関係なく、常に String タイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());
```

その代わり、anySimpleType は設定される元のデータ・タイプを失いません。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://AnySimpleType">
  <xsd:complexType name="AnySimpleType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:anySimpleType"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject anySimpleType = ...

 // データを String に設定する
 stringType.set("foo", "bar");

 // インスタンス・データは常に、set 内で使用される日付のタイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());

 // データを Integer に設定する
 stringType.set("foo", new Integer(42));

 // インスタンス・データは常に、set 内で使用される日付のタイプになる
 // 「java.lang.Integer」を表示する
 System.out.println(stringType.get("foo").getClass().getName());
```

このデータ・タイプも、シリアライゼーションおよびデシリアライゼーションをまたがっても、xsi:type によって保存されます。したがって、anySimpleType エレメントをシリアライズするときはいつでも、その Java タイプに基づいた SDO 仕様で定義されているものに一致する xsi:type が存在します。

次の例では、データが以下のようになるよう、上記のビジネス・オブジェクトをシリアライズします。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:StringType xsi:type="p:StringType"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:xsd=http://www.w3.org/2001/XMLSchema
 xmlns:p="http://StringType">
  <foo xsi:type="xsd:int">42</foo>
<p:StringType></p:StringType>
```

xsi:type は、デシリアライゼーションのときに、データを適切な Java インスタンス・クラスとしてロードするために使用されます。xsi:type が指定されていない場合、デフォルトのデシリアライゼーション・タイプは string になります。

他の単純タイプの場合、マップ可能性の判別は常に一定です。例えば、boolean は常に string にマップできます。しかし、AnySimpleType には任意の単純タイプを含めることができるので、フィールド内のインスタンス・データによって、マッピングが可能な場合とそうでない場合があります。

プロパティーが anySimpleType タイプであるかどうかを判別するには、プロパティー Type の URI と Name を使用します。それらは、「commonj.sdo」と「Object」になります。anySimpleType に挿入しても有効なデータかどうかを判別するには、それが DataObject のインスタンスでないかどうかを調べます。String として表現でき、DataObject でないすべてのデータは、anySimpleType フィールド内に設定できます。

したがって、以下のマッピング・ルールが導出されます。

- anySimpleType は、常に anySimpleType にマップできます。
- それ以外のすべての単純タイプは、常に anySimpleType にマップできます。
- anySimpleType は、常に string にマップできます。すべての単純タイプは string に変換できなければならないからです。
- anySimpleType は、ビジネス・オブジェクト内のその値に応じて、他のいずれかの単純タイプにマップできる場合とそうでない場合があります。したがって、このマッピングは設計時には判別できず、実行時にのみ判別できます。

関連情報


Assigning from and to xs:any

複合タイプへの AnyType の使用:

SDO API による anyType タグの処理は、他のいずれの複合タイプとも異なる点はありません。

anyType が他の複合タイプと異なるのは、そのインスタンス・データとシリアライゼーション/デシリアライゼーションだけです。これらはビジネス・オブジェクトのみに対する内部概念であり、フィールドへまたはフィールドからマップされるデータが valid.Complex タイプであるかどうかの判別は、単一タイプ (Customer、Address など) に限られます。しかし、anyType は、タイプに関係なく、すべての DataObject を許容します。maxOccurs > 1 の場合、リスト内の各 DataObject は異なるタイプであってもかまいません。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://AnyType">
  <xsd:complexType name="AnyType">
    <xsd:sequence>
      <xsd:element name="person" type="xsd:anyType"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://Customer">
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Employee" targetNamespace="http://Employee">
  <xsd:complexType name="Employee">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject anyType = ...
 DataObject customer = ...
 DataObject employee = ...

 // person を Customer に設定する
```

```
anyType.set("person", customer);

// インスタンス・データは Customer になる
// 「Customer」を表示する
System.out.println(anyType.getDataObject("person").getName());

// person を Employee に設定する
anyType.set("person", employee);

// インスタンス・データは Employee になる
// 「Employee」を表示する
System.out.println(anyType.getDataObject("person").getName());
```

anySimpleType とまったく同じように、anyType はシリアライゼーション時に xsi:type を使用して、DataObject の意図されたタイプがデシリアライゼーション時に維持されるようにします。したがって、「Customer」に設定した場合、XML は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:customer="http://Customer"
    xmlns:p="http://AnyType">
  <person xsi:type="customer:Customer">
    <name>foo</name>
  </person>
</p:AnyType>
```

また、「Employee」に設定した場合は、以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:employee="http://Employee"
    xmlns:p="http://AnyType">
  <person xsi:type="employee:Employee">
    <id>foo</id>
  </person>
</p:AnyType>
```

また、AnyType では、ラッパー DataObject を通じて単純タイプの値を設定できます。それらのラッパー DataObjects は、「value」(エレメント) という、単純タイプ値を保持する単一プロパティーを持ちます。SDO API は、get<Type>/set<Type> API を使用した場合、これらの単純タイプとラッパー DataObject を自動的にラップおよびアンラップするためにオーバーライドされています。型キャストのない get/set API は、このラッピングを行いません。

```
DataObject anyType = ...

// set<Type> API を anyType Property について呼び出すと、自動的に
// ラッパー DataObject が作成される
anyType.setString("person", "foo");

// 通常の get/set API は、オーバーライドされないので、
// ラッパー DataObject を返す
DataObject wrapped = anyType.get("person");

// ラップされた DataObject は「value」Property を持つ
// 「foo」を表示する
System.out.println(wrapped.getString("value"));

// get<Type> API は自動的に DataObject をアンラップする
// 「foo」を表示する
System.out.println(anyType.getString("person"));
```

ラッパー DataObject は、シリアライズされる場合、Java インスタンス・クラスの anySimpleType マッピングとまったく同じように、xsi:type フィールドで XSD タイプへシリアライズされます。したがって、この設定は、次のようにシリアライズされます。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd=" http://www.w3.org/2001/XMLSchema"
 xmlns:p="http://AnyType">
  <person xsi:type="xsd:string">foo</person>
</p:AnyType>
```

xsi:type が指定されていないか、指定された xsi:type が正しくない場合は、例外がスローされます。自動ラッピングのほかに、BOFactory createDataTypeWrapper(Type, Object) により、set() API で使用するラッパーを手動で作成できます。ここで、Type はラップされるデータの SDO 単純タイプで、Object は、ラップされるデータです。

```
Type stringType = boType.getType("http://www.w3.org/2001/XMLSchema", "string");
DataObject stringType = boFactory.createByMessage(stringType, "foo");
```

DataObject がラッパー・タイプであるかどうかを判別するには、BOType isDataTypeWrapper(Type) を呼び出すことができます。

```
DataObject stringType = ...
boolean isWrapper = boType.isDataTypeWrapper(stringType.getType());
```

他の複合タイプの場合、フィールド間でデータを移動するには、データが同じタイプである必要があります。しかし、AnyType には任意の複合タイプを含めることができるので、フィールド内のインスタンス・データによって、マッピングなしの直接の移動が可能な場合と可能でない場合があります。

プロパティー Type の URI と Name を使用して、プロパティーが anyType タイプであるかどうかを判別できます。それらは、「commonj.sdo」と「DataObject」になります。すべてのデータは、anyType に挿入するのに有効です。したがって、以下のマッピング・ルールが導出されます。

- anyType は、常に anyType にマップできます。
- すべての複合タイプは、常に anyType にマップできます。
- すべての単純タイプは、常に anyType にマップできます。
- anyType は、BO インスタンス内のその値に応じて、他のいずれかの単純タイプまたは複合タイプにマップできる場合とそうでない場合があります。したがって、このマッピングは設計時には判別できず、実行時にのみ判別できます。

複合タイプのグローバル・エレメントを設定するための Any の使用:

<any/> タグを使用して、複合タイプにグローバル・エレメントを設定できます。

any タグがあると、DataObject Type isOpen() メソッドおよび isSequenced() メソッドは true を返します。any タグ上で maxOccurs の値が > 1 の場合、DataObject の構造に影響はありません。そのタグは、妥当性検査時の情報としてのみ使用されます。同様に、type 内に複数の any タグがあっても、DataObject の構造は変更されません。それらのタグは、設定されたオープン・データの場所の妥当性検査にのみ使用されます。

どうすれば DataObject に any タグがあるかどうかが分かるか:

DataObject のインスタンス内に any 値が設定されているかどうかは、インスタンスのプロパティーを調べて、いずれかのオープン・プロパティーが属性であるかどうかを確認すれば、簡単に判別できます。

DataObject は DataObject Type に any タグがあるかどうかを判別するメカニズムを備えていません。DataObject には、any と anyAttribute の両方に適用され、any のプロパティーを自由に追加できる「オープン」の概念だけがあります。any タグが存在すると DataObject は isOpen() = true で isSequenced() = true になりますが、単にその DataObject に anyAttribute タグと、Sequence のトピックで述べた順序付けされる理由の 1 つがあるにすぎない場合も考えられます。次の例は、それらの概念を実際に示したものです。

```
DataObject dobj = ...

// タイプがオープンであるかどうかを検査し、そうでなければ
// 中に any 値が設定されていることはありえない。
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false;  // any 値は設定されていない

// オープン・プロパティーはインスタンス・プロパティー・リストに追加されるが
// プロパティー・リストには追加されないため、それらのサイズを比較すれば
// 設定されたオープン・データがあるかどうかを簡単に判別できる
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// 同じであれば、オープン・コンテンツは設定されていない
if (instancePropertyCount == definedPropertyCount) return false;

// オープン・コンテンツの Property を調べて、any が
// Element であるかどうかを判別する
for (int i=definedPropertyCount; i < instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boXsdHelper.isElement(prop))
    {
        return true;  // any 値が検出された
    }
}

return false;  // any 値は設定されていない
```

どのようにして any の値を取得/設定するか:

any フィールドに設定されたデータの取得は、名前が分かっている場合、他のすべてのエレメント値と同じように行うことができます。

XPath の「<name>」で get を行うことができ、それは解決されます。名前が不明の場合は、上記のようにインスタンス・プロパティーを調べることにより、値を見つけることができます。複数の any タグが存在するか、maxOccurs > 1 の any タグが存在する場合、データの発生元である any タグを特定することが重要であるなら、DataObject sequence を代わりに使用する必要があります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyElemAny"
 targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
```

```
      <!-- Handle all these any one way -->
      <xsd:any maxOccurs="3"/>
      <xsd:element name="marker1" type="xsd:string"/>
      <!-- Handle this any in another -->
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

<any/> タグは DataObject を順序付けするので、どの any 値が設定されたかの判別は、Sequence 内の any プロパティーの位置を調べることによって行うことができます。

以下の XSD のインスタンス・データが属する any タグを判別するには、以下のコードを使用します。

```
DataObject anyElemAny = ...
Seqeuence seq = anyElemAny.getSequence();

// marker1 エレメントを検出するまで、見つかったすべてのオープン・データは
// 最初の any タグに属する
boolean foundMarker1 = false;

for (int i=0; i<seq.size(); i++)
{
    Property prop = seq.getProperty(i);

    // プロパティーがオープン・プロパティーであるかどうかをチェックする
    if (prop.isOpenContent())
    {
        if (!foundMarker1)
        {
            // 最初の any でなければならない
            // marker1 エレメントの前にあるからである
            System.out.println("Found first any data: "+seq.getValue(i));
        }
        else
        {
            // 2 番目の any でなければならない
            // marker1 エレメントの後にあるからである
            System.out.println("Found second any data: "+seq.getValue(i));
         }
    }
    else
    {
        // marker1 エレメントでなければならない
        System.out.println("Found marker1 data: "+seq.getValue(i));
        foundMarker1 = true;
    }
}
```

<any/> 値の設定は、グローバル・エレメント・プロパティーを作成し、その値をシーケンスに追加することによって行います。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://GlobalElems"
 targetNamespace="http://GlobalElems">
  <xsd:element name="globalElement1" type="xsd:string"/>
  <xsd:element name="globalElement2" type="xsd:string"/>
</xsd:schema>

 DataObject anyElemAny = ...
 Seqeuence seq = anyElemAny.getSequence();
```

```
// globalElement1 のグローバル・エレメント Property を取得する
Property globalProp1 = boXsdHelper.getGlobalProperty(http://GlobalElems,
"globalElement1", true);

// globalElement2 のグローバル・エレメント Property を取得する
Property globalProp2 = boXsdHelper.getGlobalProperty(http://GlobalElems,
"globalElement2", true);

// データを最初の any のシーケンスに追加する
seq.add(globalProp1, "foo");
seq.add(globalProp1, "bar");

// marker1 のデータを追加する
seq.add("marker1", "separator");  // または anyElemAny.set("marker1", "separator")

// データを 2 番目の any のシーケンスに追加する
seq.add(globalProp2, "baz");

// これで、get 呼び出しによってデータにアクセスできる
System.out.println(dobj.get("globalElement1"); // 「[foo, bar]」を表示する
System.out.println(dobj.get("marker1");        // 「separator」を表示する
System.out.println(dobj.get("globalElement2"); // 「baz」を表示する
```

any でのデータの有効なマッピングはどのようなものか:

<any/> タグは、名前/値のペアの集合です。設計時に <any/> に決定できる唯一の有効なマッピングは、別の <any/> か、同じ maxOccurs 値を持つ anyType です。

any の DataObject のインスタンスに含まれている個々の値は、複合タイプ・マッピングのすべてのルールに従う基本複合タイプです。それらの複合タイプの一部は、ラップされた単純タイプの場合があり、したがって、それらは単純タイプ・マッピングのルールに従います。

複合タイプのグローバル属性を設定するための AnyAttribute の使用:

<anyAttribute/> タグを使用すると、複合タイプに任意の数のグローバル属性を設定できます。

<any/> タグと同様に、<anyAttribute/> タグがあると、DataObject Type isOpen() メソッドは true を返します。しかし、<any/> タグとは異なり、<anyAttribute/> タグは DataObject を順序付きにしません。XSD 内の属性は順序付きの構成体ではないからです。

どうすれば DataObject に anyAttribute タグがあるかどうかが分かるか:

DataObject のインスタンス内に anyAttribute 値が設定されているかどうかは、インスタンスのプロパティーを調べて、いずれかのオープン・プロパティーが属性であるかどうかを確認すれば、簡単に判別できます。

DataObject は DataObject Type に anyAttribute タグがあるかどうかを判別するメカニズムを備えていません。DataObject には、any と <anyAttribute/> の両方に適用され、any の Property を自由に追加できる「オープン」の概念だけがあります。DataObject has isOpen() = true で isSequenced() = false の場合には anyAttribute タグがなければならないということは正しいのですが、isOpen() = true で isSequenced() = true の場合には、DataObject Type は anyAttribute タグを持つ場合と持たない場合があります。

DataObject は、その DataObject を生成するために使用された XSD 構造に関するこれらの疑問にプログラマチックに回答する、メタデータ照会メソッドを備えています。必要であれば、InfoSet モデルに照会して、anyAttribute タグが存在するかどうかを知ることができます。anyAttribute は単数形で、true かそうでないかであるため、ビジネス・オブジェクトは BOXSDHelper hasAnyAttribute(Type) メソッドも提供して、この DataObject にオープン属性を設定しても有効な結果が得られるかどうかを判別できるようにします。次のコード例は、それらの概念を実際に示したものです。

```
DataObject dobj = ...

// タイプがオープンであるかどうかを検査し、そうでなければ
// 中に anyAttribute 値が設定されていることはありえない。
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false;  // anyAttribute 値は設定されていない

// オープン・プロパティーはインスタンス・プロパティー・リストに追加されるが
// プロパティー・リストには追加されないため、それらのサイズを比較すれば
// 設定されたオープン・データがあるかどうかを簡単に判別できる
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// 同じであれば、オープン・コンテンツは設定されていない
if (instancePropertyCount == definedPropertyCount) return false;

// オープン・コンテンツの Property を調べて、any が
// Attribute であるかどうかを判別する
for (int i=definedPropertyCount; i<instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boXsdHelper.isAttribute(prop))
    {
        return true;  // anyAttribute 値が検出された
    }
}

return false;  // anyAttribute 値は設定されていない
```

どのようにして anyAttribute の値を取得/設定するか:

<anyAttribute/> 値の設定は、<any/> と同じ方法で行いますが、グローバル・エレメントの代わりにグローバル属性を使用します。

anyAttribute フィールドに設定されたデータの取得は、名前が分かっている場合、他のすべての属性値と同じように行うことができます。XPath の「@<name>」で get を行うことができ、それは解決されます。名前が不明の場合は、上記のコードを使用すると、値に 1 つずつ反復してアクセスできます。次のコード例は、それを示しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyAttrOnlyMixed"
 targetNamespace="http://AnyAttrOnly">
  <xsd:complexType name="AnyAttrOnly">
    <xsd:sequence>
      <xsd:element name="element" type="xsd:string"/>
    </xsd:sequence>
    <xsd:anyAttribute/>
  </xsd:complexType>
</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://GlobalAttrs">
  <xsd:attribute name="globalAttribute" type="xsd:string"/>
</xsd:schema>

 DataObject dobj = ...

 // 設定しようとしているグローバル属性 Property を取得する
 Property globalProp = boXsdHelper.getGlobalProperty(http://GlobalAttrs,
 "globalAttribute", false);

 // 他のすべてのデータとまったく同じように、dobj に値を設定する
 dobj.set(globalProp, "foo");

 // これで、get 呼び出しによってデータにアクセスできる
 System.out.println(dobj.get("@globalAttribute")); // 「foo」を表示する
```

anyAttribute でのデータの有効なマッピングはどのようなものか:

AnyAttribute タグは、すべてのタグと同様に、一連の名前/値のペアから構成されています。したがって、anyAttribute の唯一の有効なマッピングは、別の anyAttribute です。

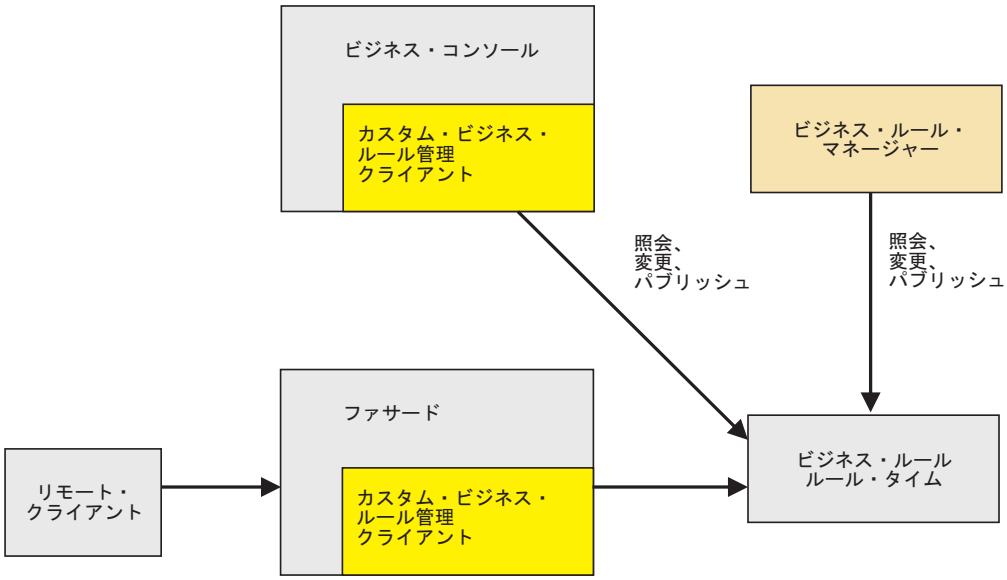
anyAttribute データに含まれている個々の値は、単純タイプ・マッピングのすべてのルールに従う基本単純タイプです。

ビジネス・ルール管理のプログラミング

カスタム管理クライアントを作成したり、ビジネス・ルールの変更を自動化したりするための共通ビジネス・ルール管理クラスが用意されています。

ビジネス・ルール管理クラスを Web アプリケーションで使用し、ビジネス・プロセスやヒューマン・タスクなどを対象とした他の管理機能と組み合わせて、単一のクライアントからすべてのコンポーネントを管理することができます。 WebSphere Process Server に組み込まれたビジネス・ルール・マネージャー Web アプリケーションと併せて、あらゆるカスタム管理クライアントを使用できます。これらのクラスは、アプリケーション内のビジネス・ルールに対する変更を自動化するためにも使用できます。例えば、ビジネス・ルールを使用するビジネス・プロセスが一定のしきい値または制限を超えた場合、その結果としてビジネス・ルールを変更することができます。

ビジネス・ルール管理クラスは、WebSphere Process Server にインストールされたアプリケーションで使用する必要があります。これらのクラスはリモート・インターフェースを提供しませんが、クラスをファサードにラップして、リモート実行用に特定のプロトコルで公開することが可能です。



このプログラミング・ガイドは、2 つのメイン・セクションと付録で構成されています。最初のセクションでは、プログラミング・モデルと、さまざまなクラスの使用方法について説明します。クラス間の関係を示すために、クラス・ダイアグラムが記載されています。 2 番目のセクションでは、クラスを使用して、ビジネス・ルール・グループの検索、新規ルール宛先のスケジューリング、およびルール・セットまたはデシジョン・テーブルの変更などのアクションを実行する例を提供します。付録には、共通操作を単純化するために例の中で使用した追加クラスと、ワイルドカードを使用してビジネス・ルール・グループを検索する複雑な照会を作成する追加の例を記載します。

クラスに関する情報は、このプログラミング・ガイドのほか、WebSphere Process Server v6.1 および WebSphere Integration Developer v6.1 付属のテスト環境の両方にも Javadoc HTML 形式で含まれています。この Javadoc 資料は、${WebSphere Process Server Install Directory}¥web¥apidocs または ${WebSphere Integration Developer Install Directory}¥runtimes¥bi_v61¥web¥apidocs にあります。パッケージ com.ibm.wbiserver.brules.mgmt.* に、すべての情報が記載されています。

プログラミング・モデル

WebSphere Business Integration のビジネス・ルールは、2 種類のオーサリング・ツールで作成されており、ルール・ランタイムによって実行されます。この 3 つはすべて、ビジネス・ルール成果物に同じモデルを共用します。

このモデルを共用することが、将来の保守を容易にするためだけでなく、エンド・ユーザーに一貫したプログラム・モデルを提供するためにも重要だと判断されました。このモデルの共有には、デスクトップ・ツールの要件と、ランタイムの実行およびオーサリングの要件との間で妥協が必要でした。いずれも、それぞれの環境に対応した明確な一連の要件があり、時にはこれらの要件が競合する場合があったからです。全体的なプログラミング・モデルの一部として以下に説明する成果物は、これらの異なる環境の要件をバランス良く満たすことを表します。

ビジネス・ルールの変更は、ルール・セットとデシジョン・テーブルのテンプレートで定義された項目、および操作選択テーブルのテンプレートで定義された項目

(有効開始日付とターゲット) のみに制限されます。ルール・セットおよびデシジョン・テーブルの新規作成は、既存のルール・セットまたはデシジョン・テーブルのコピーによってのみサポートされます。ビジネス・ルール・グループ・コンポーネント自体は、ランタイムでの動的オーサリングに適格ではありませんが、ユーザー定義のプロパティーおよび記述値については例外となります。コンポーネントに必要な変更 (例えば、新規操作の追加など) は、WebSphere Integration Developer で行った後に、サーバーに再デプロイまたは再インストールする必要があります。

ビジネス・ルール・グループ

BusinessRuleGroup クラスは、ビジネス・ルール・グループ・コンポーネントを表します。 BusinessRuleGroup クラスは、ルール・セットとデシジョン・テーブルを含むルート・オブジェクトと考えることができます。

ルール・セットとデシジョン・テーブルは、関連付けられているビジネス・ルール・グループを通じてしかアクセスできません。このクラスには、ビジネス・ルール・グループに関する情報を取得するメソッドと、ルール・セットおよびデシジョン・テーブルにアクセスするためのメソッドが含まれます。これらのメソッドを使用すると、以下の情報を取得できます。

- ターゲット名前空間
- ビジネス・ルール・グループの名前
- 表示名
- 名前/表示名の同期
- 説明
- 表示される時間帯 (システム日付を UTC 形式とローカルのどちらで表示するのかを示します)
- ビジネス・ルール・グループに関連付けられているインターフェースに定義された操作
- ビジネス・ルール・グループで定義されたカスタム・プロパティー

ビジネス・ルール・グループに関連付けられている各種のルール・セットおよびデシジョン・テーブルには、ビジネス・ルール・グループの操作を通じてアクセスできます。

ビジネス・ルール・グループの情報を更新できるメソッドも用意されています。このメソッドを使用すると、以下の情報を更新できます。

- 説明
- 表示名
- 名前/表示名の同期
- ビジネス・ルール・グループで定義されたカスタム・プロパティー

ビジネス・ルール・グループの表示名は、明示的に設定することもできますし、setDisplayNameIsSynchronizedToName メソッドを使用して Name の値に設定することもできます。

その他の値は、ビジネス・ルール・グループのコンポーネント定義の一部であるため変更できません。これらの値を変更するには、再デプロイと再インストールが必要です。

ビジネス・ルール・グループ・クラスは、最新表示メソッドも備えています。このメソッドは、ビジネス・ルールが保管されている永続ストレージまたはリポジトリーを呼び出して、ビジネス・ルール・グループおよび関連するすべてのルール・セットとデシジョン・テーブルを持続情報とともに戻します。戻されるビジネス・ルール・グループは最新のコピーであり、直前のオブジェクトは廃止になります。

isShell メソッドを使用すると、ビジネス・ルール・グループ・インスタンスが現在のランタイムでサポートされないバージョンであるかどうかを確認できます。例えば、現在のビジネス・ルール管理クラスで Web クライアントを作成しており、将来、このクラスでサポートされない新機能をビジネス・ルール・グループに追加する場合は、ビジネス・ルール・グループを取得するときに、シェル・ビジネス・ルール・グループが作成されます。これにより、Web クライアントは、サポートされるビジネス・ルールの操作を続行することができ、属性と機能が制限されたビジネス・ルール・グループを依然として取得することができます。 isShell が true の場合、値を戻すメソッドは、getName、 getTargetNameSpace、 getProperties、 getPropertyValue、および getProperty のみです。それ以外のすべてのメソッドは、UnsupportedOperationException を生成します。 isShell メソッドを使用することに加えて、BusinessRuleGroup のタイプも検査すれば、これが BusinessRuleGroupShell のインスタンスであるかどうかを確認して、サポートされるバージョンであるかどうかを判断することができます。

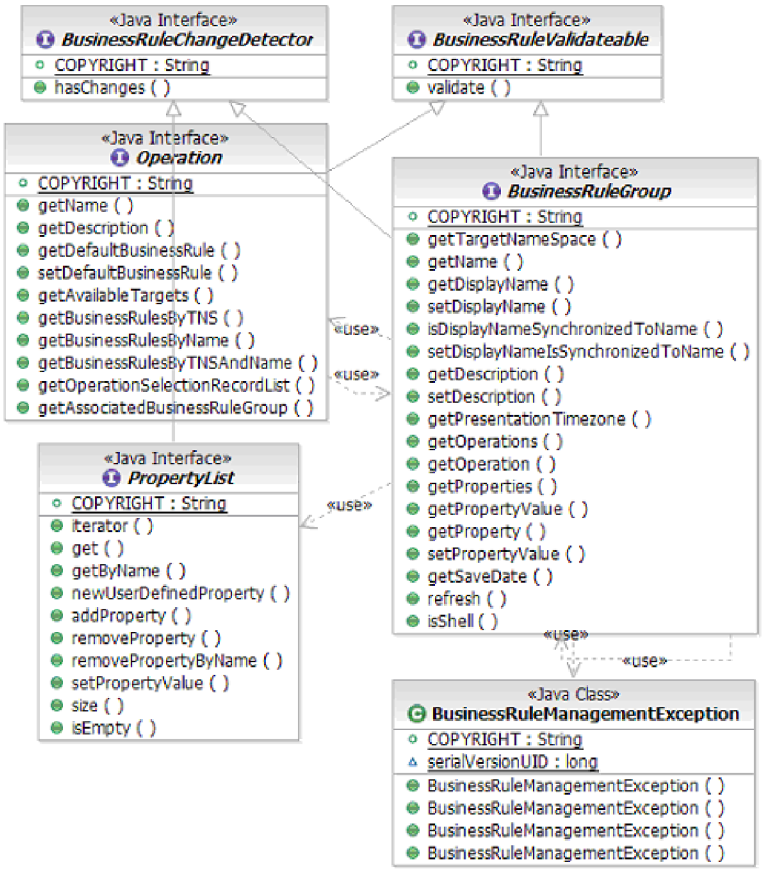


図 59. BusinessRuleGroup および関連するクラスのクラス・ダイアグラム

ビジネス・ルール・グループのプロパティー

ビジネス・ルール・グループのプロパティーは、ビジネス・ルール・グループを管理するために使用されます。ビジネス・ルール・グループのプロパティー・セットを照会で使用すると、表示して変更する必要のあるビジネス・ルール・グループのサブセットのみを戻すことができます。

プロパティーのタイプはすべてストリングであり、名前と値のペアとして定義されます。各プロパティーは、ビジネス・ルール・グループで 1 回のみ定義できます。定義されている各プロパティーには、値も定義されている必要があります。プロパティーの値は、空ストリングにすることも、長さをゼロにすることもできますが、ヌルにすることはできません。プロパティーをヌルに設定することは、プロパティーを削除することと同じです。

ビジネス・ルール・グループのプロパティーは、実行時にルール・セットまたはデシジョン・テーブルでアクセスすることもできます。これにより、ビジネス・ルール・グループの複数のルール・セットまたはデシジョン・テーブル内で使用する単一値をビジネス・ルール・グループで設定することができます。含まれているルール・セットおよびデシジョン・テーブルで使用できるのは、ビジネス・ルール・グループで定義されているプロパティーに限られます。

プロパティーには、システム定義プロパティーとユーザー定義プロパティーの 2 種類があります。ビジネス・ルール・グループでは、システム定義プロパティーまたはユーザー定義プロパティーの数に制限はありません。システム・プロパティーは、特定のコンポーネント情報 (ルール・ロジックの定義時に使用するルール・モデルのバージョンなど) を保持するために使用されます。このシステム情報はプロパティーで公開され、これらのフィールド全体での照会が可能になります。システム・プロパティーは、接頭部 IBMSystem で始まり、ビジネス・ルール・グループおよびプロパティー・クラスを通じて、読み取り専用でアクセスされます。システム・プロパティーを追加、変更、または削除することはできません。システム・プロパティーの例を以下に示します。

プロパティー名	プロパティー値
IBMSystemVersion	6.2.0

注: ビジネス・ルール・グループの名前、名前空間、および表示名の値は、照会用のシステム・プロパティーとして扱われます。これらの値は、ビジネス・ルール・グループのプロパティーのリストに含められ、getProperties メソッドを使用して取得することができます。ただし、これらのプロパティーは、ビジネス・ルール・グループの独立した固有のエレメントで定義されており、ビジネス・ルール・グループ成果物の実際のプロパティー・エレメントとしては定義されていないため、WebSphere Integration Developer ではプロパティーとして表示されません。これらのプロパティーは、より多くの照会オプションを提供するためにのみに用意されています。

ユーザー定義プロパティーは、任意のユーザー固有の情報を保持するために使用します。ビジネス・ルール・グループの照会で使用することも可能です。ユーザー定義プロパティーは、読み取りと書き込みを行うことができます。

ビジネス・ルール・グループのプロパティーは、個々に取得することも、リスト (PropertyList オブジェクト) として取得することもできます。 PropertyList では、個々のプロパティーを取得するメソッドと、ユーザー定義プロパティーを追加および除去するメソッドを利用できます。

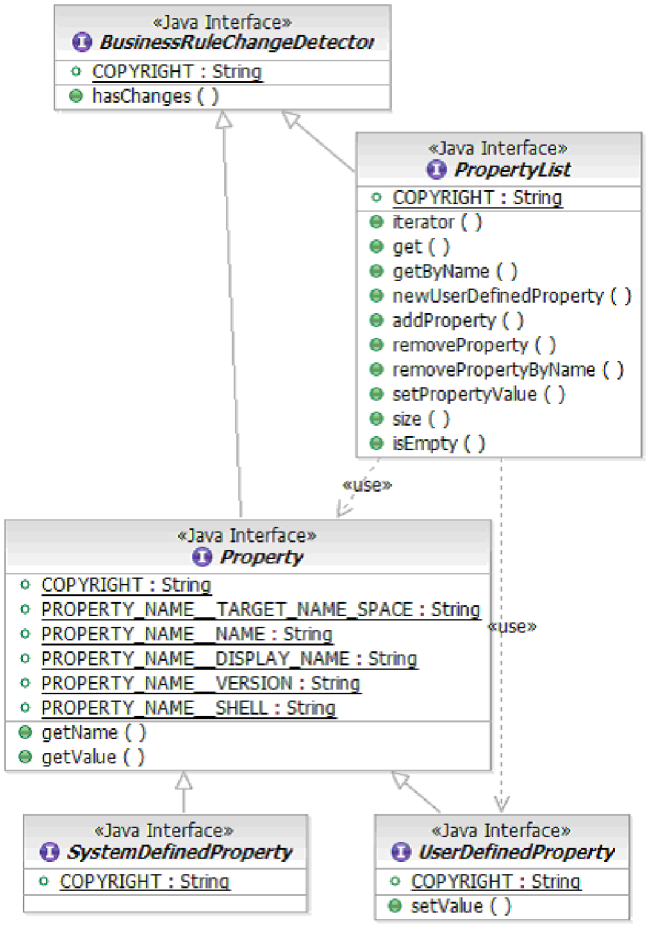


図 60. Property および関連するクラスのクラス・ダイアグラム

操作

操作は、変更する個々のルール・セットやデシジョン・テーブルに到達するための開始点となります。ビジネス・ルール・グループの操作は、そのビジネス・ルール・グループのコンポーネントに関連付けられた WSDL にリストされた操作と一致します。

各操作には、以下の異なるターゲットがあります。このターゲットのそれぞれが、ビジネス・ルール (ルール・セットまたはデシジョン・テーブル) です。

- デフォルト・ターゲット (オプション)
- 日付/時刻の範囲によってスケジュールされたターゲットのリスト (OperationSelectionRecord)
- 操作に使用可能なすべてのターゲットのリスト

各操作には、少なくとも 1 つのビジネス・ルール・ターゲットを指定する必要があります。このターゲットは、開始日と終了日でターゲットをアクティブにするスケジュールを指定した OperationSelectionRecord にできます。操作には単一のデフォルト・ターゲットを設定することもできます。この場合、一致するスケジュール済みビジネス・ルール・ターゲットが見つからない場合には、設定されたデフォルト・ターゲットが実行中に使用されます。 Operation クラスには、デフォルト・ビジネス・ルール・ターゲットを取得および設定するためのメソッド、およびスケジュール済みビジネス・ルール・ターゲットのリスト (OperationSelectionRecordList) を取得するためのメソッドがあります。デフォルト・ビジネス・ルール・ターゲットとスケジュール済みビジネス・ルール・ターゲットとは別に、操作に使用可能なすべてのビジネス・ルール・ターゲットのリストがあります。このリストには、スケジュール済みおよびデフォルトのビジネス・ルール・ターゲットだけでなく、この操作にスケジュールされていない他のすべてのルール・セットまたはデシジョン・テーブルも含まれます。スケジュールされていないルール・セットまたはデシジョン・テーブルは、操作の情報を暗黙的に共用することから、使用可能なターゲット・リストによって操作に関連付けられています。すべてのビジネス・ルール・ターゲットは、それぞれに該当する操作の入力および出力メッセージをサポートする必要があります。各操作はインターフェース上で固有であるため、操作のルール・セットおよびデシジョン・テーブルは、別の操作のルール・セットおよびデシジョン・テーブルに対しても固有です。

使用可能なターゲット・リストに含まれるそれぞれのルール・セットおよびデシジョン・テーブルはいずれも、OperationSelectionRecord を作成することによって、アクティブになるようにスケジュールできます。使用可能なターゲット・リストから特定のルール・セットまたはデシジョン・テーブルを指定するのに加えて、開始日と終了日も指定する必要があります。開始日は、終了日よりも前でなければなりません。この 2 つの日付によって、現在、過去および将来の日付を範囲とする期間を指定できます。日付の範囲を OperationSelectionRecordList に追加して公開を行う時点で、その日付の範囲が他のどの OperationSelectionRecord ともオーバーラップしていないことが必要です。開始日と終了日の値の型は、java.util.Date です。指定する値はすべて、java.util.Date クラスに従った UTC 値として扱われます。作成した OperationSelectionRecord は、OperationSelectionRecordList に追加して他のビジネス・ルール・ターゲットと併せてスケジュールできます。異なる OperationSelectionRecord の間に、時間のギャップがある場合があります。実行中にギャップが検出された場合には、デフォルト・ターゲットが使用されます。デフォルト・ターゲットが指定されていない場合、例外がスローされます。常にデフォルト・ビジネス・ルール・ターゲットを指定することをお勧めします。

スケジュール済みビジネス・ルール・ターゲットをスケジュール済みターゲットのリストから除去するには、OperationSelectionRecord を OperationSelectionRecordList から除去します。 OperationSelectionRecord を除去しても、使用可能なビジネス・ルール・ターゲットのリストからビジネス・ルール・ターゲットが除去されたり、同じビジネス・ルール・ターゲットがスケジュールされている他の OperationSelectionRecord が除去されたりすることはありません。

Operation クラスでは、OperationSelectionRecordList または使用可能なターゲット・リストを使用してルール・セットまたはデシジョン・テーブルを取得できるだけでなく、名前とターゲット名前空間のプロパティー値によってビジネス・ルール・ターゲットを取得することもできます。 Operation クラスのメソッドによって、該当

する操作に使用可能なターゲット・リストに含まれるルール・セットおよびデシジョン・テーブルを照会できます。他の操作に使用可能なターゲット・リストに含まれるルール・セットおよびデシジョン・テーブルは、名前およびターゲット名前空間が一致していても、この結果セットには含まれません。特定のルール・セットおよびデシジョン・テーブルの取得を単純化するために、getBusinessRulesByName、getBusinessRulesByTNS、および getBusinessRulesByTNSAndName メソッドが用意されています。

Operation クラスは、以下の操作をサポートするメソッドを提供します。

- 操作名の取得
- 操作についての説明の取得
- デフォルト・ビジネス・ルール・ターゲットの取得と設定
- スケジュール済みビジネス・ルール・ターゲット (OperationSelectionRecordList) の取得
- 使用可能なすべてのビジネス・ルール・ターゲットのリストの取得
- 名前またはターゲット名前空間を基準とした、使用可能なすべてのターゲット・リストからのルール・セットまたはデシジョン・テーブルの取得
- 操作が関連付けられたビジネス・ルール・グループの取得

OperationSelectionRecordList クラスは、以下の操作をサポートするメソッドを提供します。

- 索引値を基準とした、特定の OperationSelectionRecord の取得
- 索引値を基準とした、特定の OperationSelectionRecord の除去
- リストへの新規 OperationSelectionRecord の追加

OperationSelectionRecord クラスは、以下の操作をサポートするメソッドを提供します。

- 開始日の取得と設定
- 終了日の取得と設定
- ビジネス・ルール・ターゲットの取得と設定
- OperationSelectionRecord が関連付けられた操作の取得

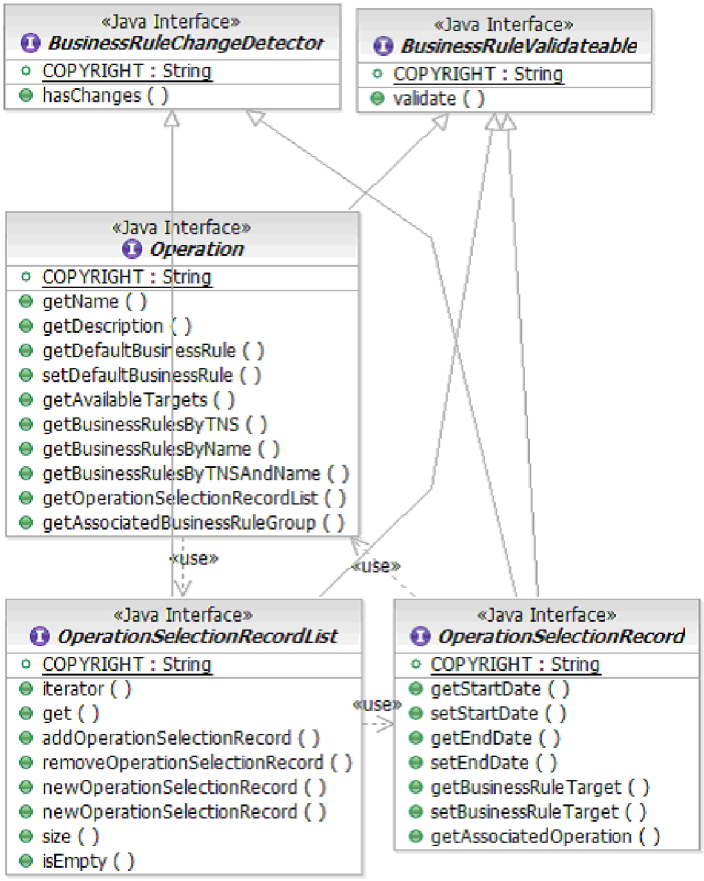


図 61. Operation および関連クラスのクラス・ダイアグラム

ビジネス・ルール

RuleSet クラスと DecisionTable クラスは、ルール・セットとデシジョン・テーブルの両方で使用可能な情報を提供するメソッドを備えた汎用の BusinessRule クラスから派生しています。

ビジネス・ルール・グループの成果物と同様、ルール・セットとデシジョン・テーブルには、名前とターゲット名前空間があります。その他のルール・セットおよびデシジョン・テーブルと比較して、これらの値の組み合わせは固有でなければなりません。例えば、2 つのルール・セットは、同じターゲット名前空間の値を共用できますが、名前は別々でなければなりません。また、ルール・セットとデシジョン・テーブルは、同じ名前を持つことはできますが、ターゲット名前空間の値は異なります。

パラメーター値が異なる同等のルールを特定時刻にスケジュールする必要があるときは、ビジネス・ルールのコピーを既存のビジネス・ルールから作成できます (テンプレートからルールが作成されている場合)。新しいルールを最初から完全に作成することはできません。なぜなら、ビジネス・ルールの実装を提供するには、バッキング Java クラスが必要だからです。バッキング Java クラスは、デプロイ時にのみ作成されます。新しいルールを作成すると、そのルールは、元のルールに関連付

けられている操作の使用可能なターゲットのリストに追加されます。ただし、操作が関連付けられているビジネス・ルール・グループが公開されるまで、追加のルールは持続されません。

新しいビジネス・ルールは、元のルールとは異なるターゲット名前空間または名前を持つ必要があります。新しいビジネス・ルールの表示名は、元のルールと同じままにすることができます。なぜなら、名前と名前空間の組み合わせにより、ビジネス・ルールを識別するキー値が得られるからです。ビジネス・ルール内では、テンプレートを使用して定義した各種のパラメーター値を変更できます。ビジネス・ルールを特定時刻にスケジュールするには、OperationSelectionRecordList を使用するか、またはデフォルトの宛先として、ビジネス・ルールに関連付けられた Operation を使用します。

BusinessRule クラスは、以下の操作をサポートするメソッドを提供します。

- ターゲット名前空間の取得
- ルール・セットまたはデシジョン・テーブルの名前の取得
- ルール・セットまたはデシジョン・テーブルの表示名の取得および設定
- ビジネス・ルールのタイプ (ルール・セットまたはデシジョン・テーブル) の取得
- ビジネス・ルールの説明の取得および設定
- ビジネス・ルールが関連付けられている操作の取得
- 名前やターゲット名前空間が異なる、ビジネス・ルールのコピーの作成

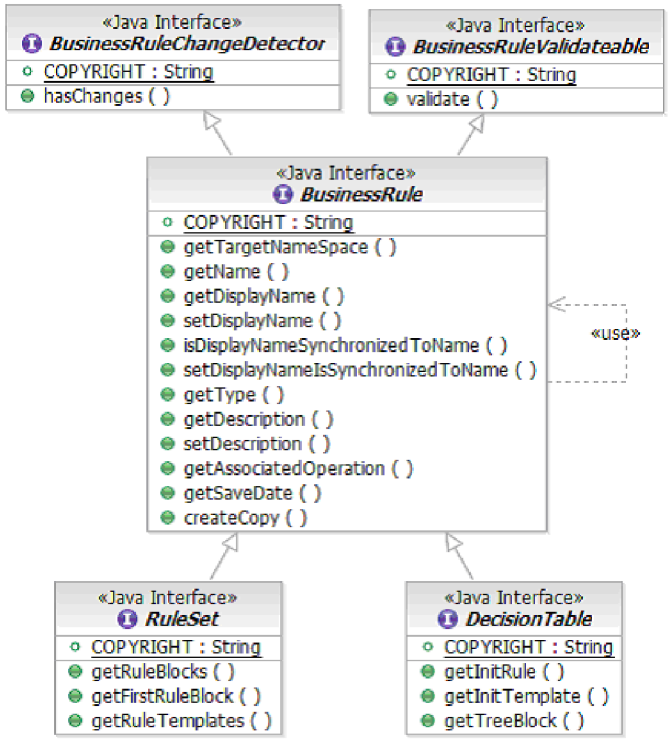


図 62. BusinessRule および関連するクラスのクラス・ダイアグラム

ルール・セット

ルール・セットは、ビジネス・ルールの 1 つのタイプです。通常、ルール・セットは、異なる条件値に基づいて複数のルールを実行する必要がある場合に使用されます。ルール・セットはルール・ブロックおよびルール・テンプレートで構成されます。ルール・ブロック (RuleBlock) には、ルール・セットのロジックを構成するさまざまな if-then およびアクション・ルールが含まれます。

RuleSet クラスは、以下の操作をサポートするメソッドを提供します。

- ルール・セットのルール・ブロックのリストの取得
- ルール・セットに定義されたルール・テンプレートのリストの取得

現在、各ルール・セットに含めることのできるルール・ブロックは 1 つのみですが、ルール・セットには複数のルール・テンプレートを定義できます。ルール・ブロックには、ルール・セットが呼び出されると実行される一連のルールが含まれます。ルール・ブロックではルールの順序を変更できます。ルール・ブロックには少なくとも 1 つのルールが定義されていなければなりません。ルール (Rule) は、テンプレート・インスタンス・ルール (TemplateInstanceRule) として定義することも、ハードコーディングすることもできます。テンプレートで定義されている if-then またはアクション・ルールは、ルール・ブロックから除去できます。テンプレートで作成されたルールの新しいインスタンスは、ルール・ブロックに追加できます。

ルールがハードコーディングされていて、テンプレートでは定義されていない場合、そのルールは変更することも、ルール・ブロックから除去することもできません。これらのルールは、常にルール・セット・ロジックの一部となるように設計されており、ロジック内で変更されたり繰り返されたりすることは想定されていません。

新規ルールをテンプレートで作成するときには、そのルールに固有の名前が必要です。ルールを作成する前に、既存のルールのリストを取得して確認できます。

ハードコーディングされた if-then およびアクション・ルールについては、その名前と表示のみを取得できます。この表示は、クライアント・アプリケーションでルールに関する情報を表示するために使用できるストリングです。テンプレートで定義された if-then またはアクション・ルールについては、その名前と表示の他に、追加情報も取得できます。特定のパラメーター値を取得して変更することができます。ルール・セットに定義されたテンプレート (RuleSetRuleTemplate) を使用して、ルール・セット内にルールの別のインスタンスを作成し、パラメーター値を設定できます。例えば、特定の状況レベルの顧客が特定の量のディスカウントを受けると指定しているルールがあるとします。このロジックは、単一のルール・テンプレートで定義してから、該当する状況レベル (gold、silver、bronze など) およびディスカウント量 (15%、10%、5% など) ごとにパラメーター値を変更して繰り返すことができます。

テンプレートで定義されたルールのパラメーターは、ルールのインスタンスに固有です。テンプレートがルールについて定義するのは、標準の表示とパラメーターの数だけです。異なる顧客の状況に応じたディスカウントの例で説明したように、テンプレートで定義されたルールごとに異なる値を設定できます。

RuleBlock クラスは、以下の操作をサポートするメソッドを提供します。

- 索引によるルールの取得
- テンプレートで定義されたルールの追加
- テンプレートで定義されたルールの除去
- ルールの順序の変更 (1 つ上または下、もしくは特定の索引ロケーションへの移動)

RuleSetRule クラスは、以下の操作をサポートするメソッドを提供します。

- ルールの名前の取得
- ルールの表示名の取得
- ユーザー表示の取得
- ルール・ブロックの取得

RuleSetRuleTemplate クラスは、以下の操作をサポートするメソッドを提供します。

- このテンプレート定義からのルール・テンプレート・インスタンスの作成
- 親ルール・セットの取得

TemplateInstanceRule クラスは、以下の操作をサポートするメソッドを提供します。

- ルールのパラメーターの取得
- ルールを定義したテンプレート定義の取得

Template クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート ID の取得
- 名前の取得
- 表示名の取得と設定
- 説明の取得と設定
- このテンプレートのパラメーターの取得
- ユーザー表示の取得

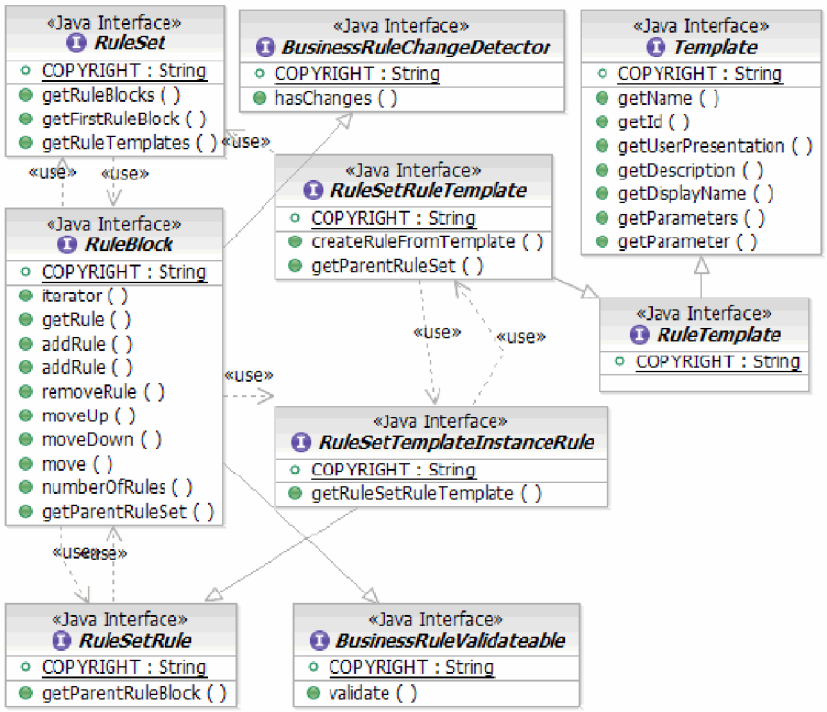


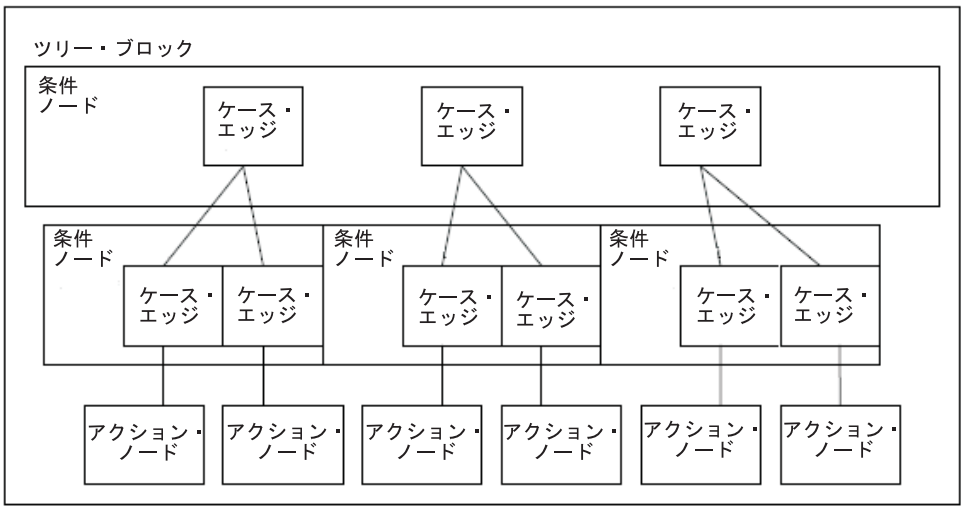
図 63. BusinessRule および関連クラスのクラス・ダイアグラム

デシジョン・テーブル

デシジョン・テーブルは、管理および変更することのできるもう一つのタイプのビジネス・ルールです。通常、デシジョン・テーブルが使用されるのは、評価すべき条件の数に矛盾がなく、条件を満足した時点で発行する特定のアクション・セットが存在するときです。

デシジョン・テーブルは決定木と似ていますが、平衡型であるという特徴があります。どのブランチ・セットが真に解決されるかに関係なく、デシジョン・テーブルには常に同じ数の評価条件と実行アクションがあります。決定木は、別のブランチよりも評価条件を多く持つブランチを 1 つ持つことができます。

デシジョン・テーブルは、ノードのツリーとして作成され、`TreeBlock` によって定義されます。 `TreeBlock` は、各種の `TreeNodes` から構成されます。 `TreeNodes` は、条件ノードまたはアクション・ノードにすることができます。条件ノードは、評価ブランチです。条件がすべて真に評価された場合、ブランチの最後には、発行する該当のツリー・アクションを持つアクション・ノードがあります。条件ノードのレベルはいくつにでもできますが、アクション・ノードのレベルは 1 つにしかできません。



デシジョン・テーブルは、初期化ルール (init ルール) を持つこともできます。このルールは、テーブルの条件を検査する前に発行できます。

DecisionTable クラスは、以下の操作をサポートするメソッドを提供します。

- ツリー・ノード (条件ノードとアクション・ノード) のツリー・ブロックの取得
- init ルール・インスタンスの取得
- init ルール・テンプレートの取得 (定義されている場合)

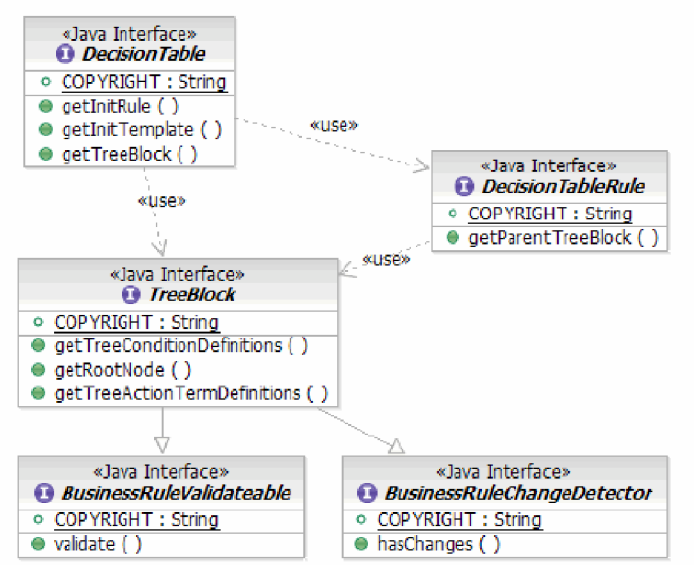
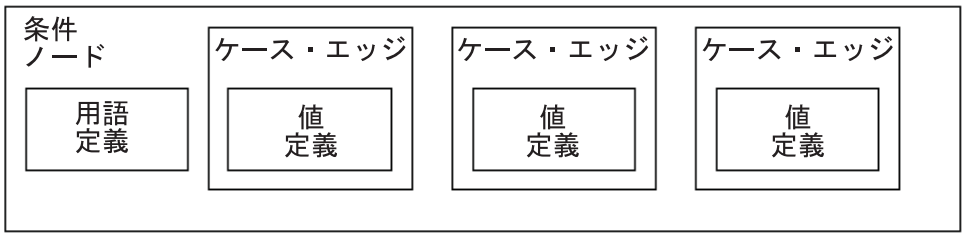


図 64. DecisionTable および関連するクラスのクラス・ダイアグラム

デシジョン・テーブルの TreeBlock には、さまざまな条件ノードとアクション・ノードが含まれています。各条件ノード (ConditionNode) には、条件定義 (TreeConditionTermDefinition) と、 1 対 n のケース・エッジ (CaseEdge) があります。条件定義には、条件式の左側のオペランドが含まれます。ケース・エッジには、値定義 (条件式の右側で使用する各種のオペランド) が含まれます。例えば、式

(status == “gold”) の場合、条件定義は「status」であり、「gold」はケース・エッジの値定義です。条件ノードのすべてのケース・エッジでは、条件定義が共有されており、値 (TreeConditionValueDefinition) のみが異なっています。例を続けます。条件ノードの別のケース・エッジは、値「silver」を持つことができます。この値は、式でも使用されます (status == “silver”)。この動作の唯一の例外は、otherwise が条件ノードに定義されている場合です。 otherwise では、値定義が存在しません。なぜなら、それは、条件ノード内のその他のすべてのケース・エッジが偽に評価された場合に使用されるからです。 otherwise は、ケース・エッジではありませんが、取得可能な TreeNode を持ちます。



条件定義の場合は、クライアント・アプリケーションでユーザー表示を取得して使用できます。通常、条件定義の表示は、左側のオペランド (この例では status) の表示のみであり、プレースホルダーは含まれません。ケース・エッジの場合は、テンプレートを使用して、値定義を定義できます (TreeConditionValueTemplate)。テンプレート値定義インスタンス (TemplateInstanceExpression) は、実行用に使用されるパラメーター値を保持しています。このインスタンスは、変更することが可能です。テンプレートで定義されていない TreeConditionValueDefinition の値テンプレート定義の取得を試みると、ヌル値が戻されます。値の条件を定義するときにテンプレートを使用しなかった場合でも、オーサリング時に指定されていれば、クライアント・アプリケーションでユーザー表示を取得して使用することができます。

TreeBlock クラスは、以下の操作をサポートするメソッドを提供します。

- ツリーのルート・ノードの取得
- ツリー・ブロックの条件項の定義の取得
- ツリー・ブロックのアクション項の定義の取得

ツリーのルート・ノードのタイプは、TreeNode です。ここからは、デシジョン・テーブルのナビゲーションを使用する場合があります。 TreeNode クラスは、以下の操作をサポートするメソッドを提供します。

- ノードが otherwise 節かどうかの確認
- 現在のツリー・ノード (条件ノードまたはアクション・ノード) の親ノードの取得
- 現在のツリー・ノードを含むツリーのルート・ノードの取得

ConditionNode クラスは、以下の操作をサポートするメソッドを提供します。

- ケース・エッジの取得
- 条件定義の取得
- otherwise ケースの取得

- 条件ノード用のケース・エッジの値条件のテンプレートの取得
- テンプレートに基づく条件値のノードへの追加
- テンプレートに基づく条件値の除去

CaseEdge クラスは、以下の操作をサポートするメソッドを提供します。
- 値定義で使用できる値テンプレートのリストの取得
- 子ノード (条件ノードまたはアクション・ノード) の取得
- 値定義に関連付けられたテンプレート定義のインスタンスの取得
- テンプレートを取得せずに値定義を直接取得
- 特定のテンプレート・インスタンス定義を使用するための定義の値の設定

TreeConditionTermDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 条件ノード用に定義された値定義テンプレートの取得
- 条件項のユーザー表示の取得

TreeConditionDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 条件ノードの条件定義の取得
- 条件ノードの条件値の定義をすべてのケース・エッジから取得
- 方向 (行または列) の取得

TreeConditionValueDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 値用に定義された特定のテンプレート・インスタンス式の取得
- ユーザーの取得

Template クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレートのシステム ID の取得
- テンプレートの名前の取得
- テンプレート用に定義されたパラメーターの取得
- テンプレートの表示の取得

TreeConditionValueTemplate クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレート条件値インスタンスの新規作成

TemplateInstanceExpression クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレート・インスタンスのパラメーターの取得
- インスタンスを定義するときに使用したテンプレート (デシジョン・テーブルのケース・エッジの場合は TreeConditionValueTemplate) の取得

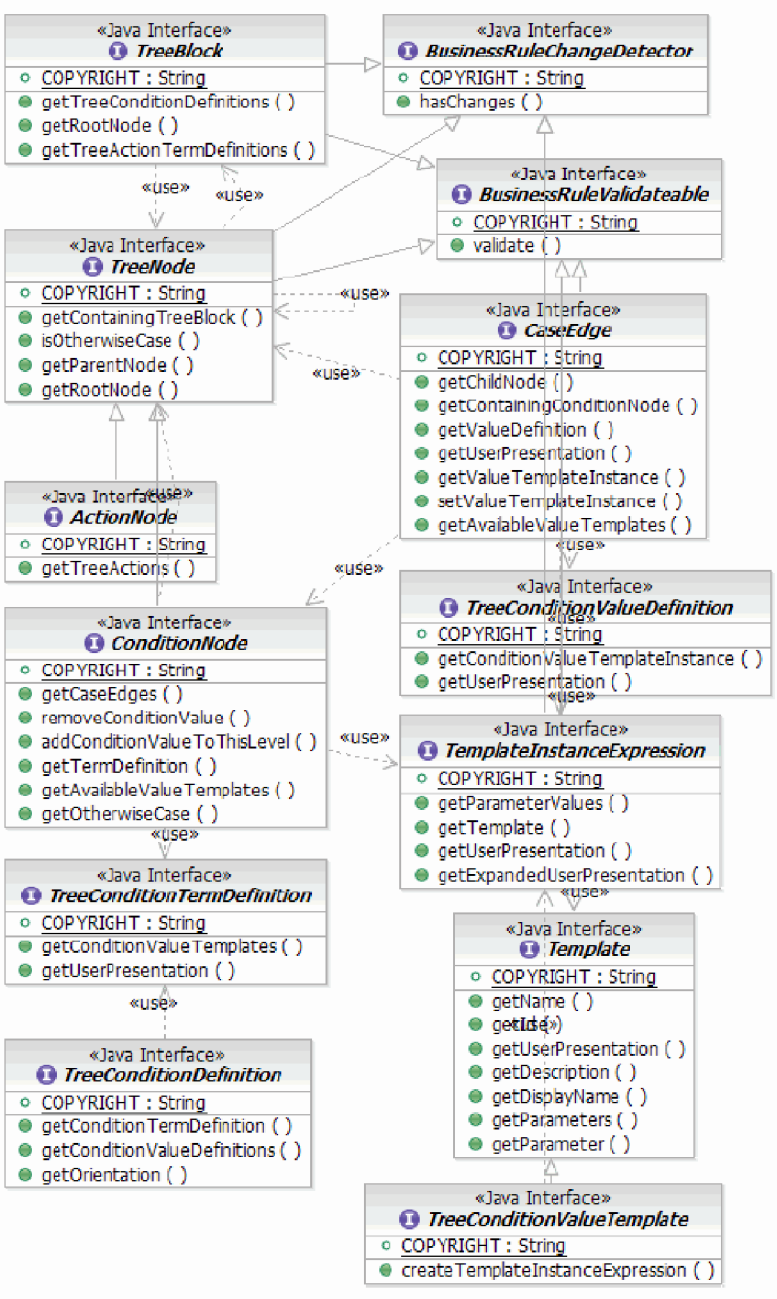


図 65. TreeNode および関連するクラスのクラス・ダイアグラム

新しいケース・エッジを条件ノードに追加する場合、そのケース・エッジでは、値を定義するためにテンプレートを使用する必要があります。例えば、「status」を検査するために「bronze」の新しいケース・エッジを追加する必要がある場合、パラメーター値に「bronze」を設定して新しい TemplateInstanceExpression を作成するには、該当するテンプレート (`TreeConditionValueTemplate`) を使用する必要があります。

新しいケース・エッジを追加する場合、そのケース・エッジには、子条件ノードが自動的に追加されます。この子条件ノードには、その同じレベルで条件ノード用に定義されたケース・エッジ定義に基づくケース・エッジが含まれます。テンプレートまたはハードコーディングした値をケース・エッジで使用する場合、これらは、

子条件ノードのケース・エッジでも使用されます。また、自動的に追加される子条件ノードでは、独自の子条件ノードが自動的に作成されます。すべてのレベルの条件ノードが再作成されるまで、子条件ノードにもその子条件ノードが繰り返し作成されます。

条件ノードに加えて、デシジョン・テーブル、より具体的に言えばツリー・ブロックにも、あるレベルのアクション・ノード (ActionNode) が含まれます。アクション・ノードは、リーフ・ノードであり、条件ノードのブランチの末端とケース・エッジに存在します。ケース・エッジの行で条件値がすべて真に解決された場合は、アクション・ノードに到達します。アクション・ノードには、少なくとも 1 つのアクション (TreeAction) が定義されます。このアクションには、条件定義と値定義が存在します。条件ノードと同様、条件定義 (TreeActionTermDefinition) は式の左側であり、値定義 (TemplateInstanceExpression) は式の右側です。例えば、状況を検査している各種の条件ノードにおいて、割引を定義するアクションが存在します。条件が (status == “gold”) の場合、アクションは、(discountValue = 0.90) になります。このアクションでは、「discountValue」が条件定義であり、「= 0.90」が値定義です。

ツリー・アクションの条件定義は、別のアクション・ノードのその他のツリー・アクションと共用されます。ケース・エッジのすべてのブランチがアクションに到達するので、同じ条件定義が使用されます。ただし、値定義は、ツリー・アクションおよびアクション・ノードごとに異なる場合があります。例えば、状況が「gold」の discountValue は「0.90」になる場合がありますが、状況が「silver」の「discountValue」は「0.95」になる場合があります。

アクション・ノードは、独立した条件定義と独立した値定義を含むツリー・アクションを複数持つことができます。例えば、レンタカーの割引を決定する場合は、discountValue を設定するほかに、特定のレベルの車を割り当てることもできます。「carSize」条件に「full size」を設定し (状況が「gold」の場合)、「discountValue」に「0.90」を設定する別のツリー・アクションを作成することができます。

ツリー・アクションの値定義は、テンプレート (TreeActionValueTemplate) から作成できます。テンプレート定義には、パラメーターを持つ式 (TemplateInstanceExpression) が含まれます。

パラメーターを変更することのほかに、値定義全体を変更できます。値定義全体を変更するには、ツリー・アクション用に定義された別のテンプレートで作成した新しい値定義インスタンスを使用します。

値定義は、テンプレートを使用せずに作成した場合は、変更することはできません。クライアント・アプリケーションの場合、オーサリング時にユーザー表示を指定した場合は、そのユーザー表示を表示内で使用できます。

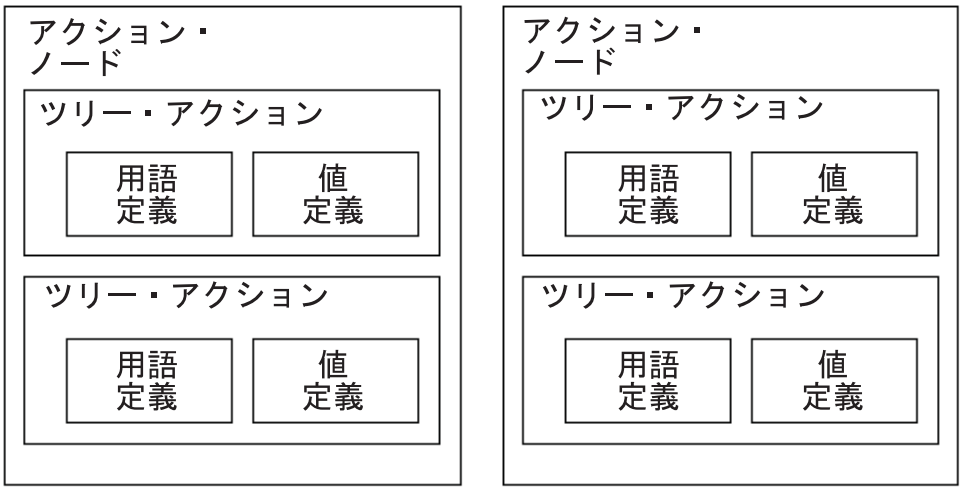
ツリー・アクションの条件定義の場合、ユーザー表示を指定すると、そのユーザー表示をクライアント・アプリケーションでも使用できます。

新しいケース・エッジを条件ノードに追加して、別の子条件ノードを作成すると、アクション・ノードも作成されます。そのレベル用に定義済みのケース・エッジの定義に基づいて作成されたケース・エッジおよび子条件ノードとは異なり、アクション・ノードでは、既存の設計が自動的に継承されません。アクション・ノードで

は、空のプレースホルダー TreeActions のみが作成されます。アクション・ノードの 1 つ以上の条件定義用に TemplateInstanceExpression を作成してアクション定義を完了するには、テンプレート (TreeActionValueTemplate) を使用する必要があります。 TemplateInstanceExpression を使用してツリー・アクションを設定するまで、ツリー・アクションでは、ユーザー表示値とテンプレート・インスタンス値にヌル値が指定されます。

新規の ActionNodes を生成する新しい条件を作成すると、直接の親の条件ノードの既存のアクションの右側にアクション・ノードが追加されます。例えば、「ruby」の状況がデシジョン・テーブルに追加され、特定の割引を持つ必要がある場合は、状況を検査する条件が「gold」、「silver」、および「bronze」の右側に追加されます。「ruby」の割引のアクション・ノードは、「gold」、「silver」、および「bronze」ケース・エッジに対応するアクション・ノードの右側に追加されます。

アクション・ノードの新しいツリー・アクションを設定すると、最下位のケース・エッジの右端のアクション・ノードに注目するアルゴリズムが、空のツリー・アクションを持つアクション・ノードを戻します。ユーザー表示値およびテンプレート・インスタンス値がヌル値であるかどうかについて、ツリー・アクションを検査することもできます。ツリー・アクションを取得すると、TreeActionValueTemplate の正しいインスタンスを使用して、そのツリー・アクションを設定することができます。



ActionNode クラスは、以下の操作をサポートするメソッドを提供します。

- 定義済みのツリー・アクションのリストの取得

TreeAction クラスは、以下の操作をサポートするメソッドを提供します。

- ツリー・アクション用に定義された使用可能な値テンプレートのリストの取得
- 条件定義の取得
- ツリー・アクション用に定義された値テンプレート・インスタンスの取得
- 値テンプレートを使用しなかった場合における値のユーザー表示の取得

- アクションが SCA サービス呼び出し (isValueNotApplicable メソッド) であるかどうかの確認
- 値テンプレート・インスタンスの新しいインスタンスでの置換

TreeActionTermDefinition クラスは、以下の操作をサポートするメソッドを提供します。

- 条件値の定義のユーザー表示の取得
- ツリー・アクションで使用可能な値テンプレートのリストの取得
- アクションが SCA サービス呼び出し (isTermNotApplicable メソッド) であるかどうかの確認

Template クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレートのシステム ID の取得
- テンプレートの名前の取得
- テンプレート用に定義されたパラメーターの取得
- テンプレートの表示の取得

TreeActionValueTemplate クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート定義からの新しい値テンプレート・インスタンスの作成

TemplateInstanceExpression クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート・インスタンスのパラメーターの取得
- インスタンスを定義するときに使用したテンプレート (デシジョン・テーブルのツリー・アクションの場合は TreeActionValueTemplate) の取得

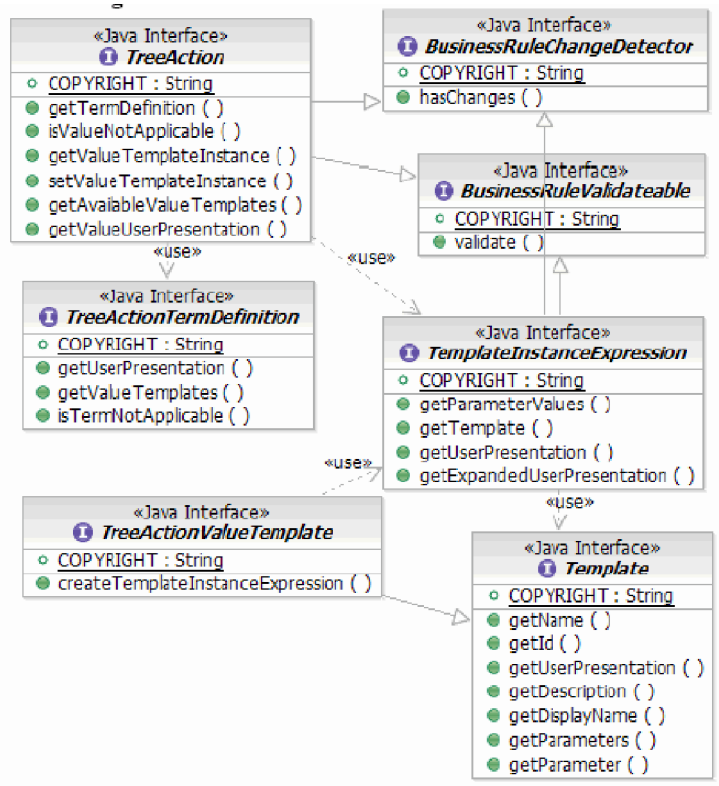


図 66. TreeAction および関連するクラスのクラス・ダイアグラム

デシジョン・テーブルの init ルールの定義は、ルール・セットのルールと同じ構造に従います。 init ルールは、テンプレート (DecisionTableRuleTemplate) で定義できます。

init ルールがオーサリング時に作成されなかった場合は、ルールをデプロイしたときにそれを追加することはできません。

Rule クラスは、以下の操作をサポートするメソッドを提供します。

- ルールの名前の取得
- ルールのユーザー表示の取得
- 各種のルール・パラメーターが設定されたルールのユーザー表示の取得

DecisionTableRule クラスは、以下の操作をサポートするメソッドを提供します。

- init ルールを含むツリー・ブロックの取得

DecisionTableRuleTemplate クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレートを含むデシジョン・テーブルの取得

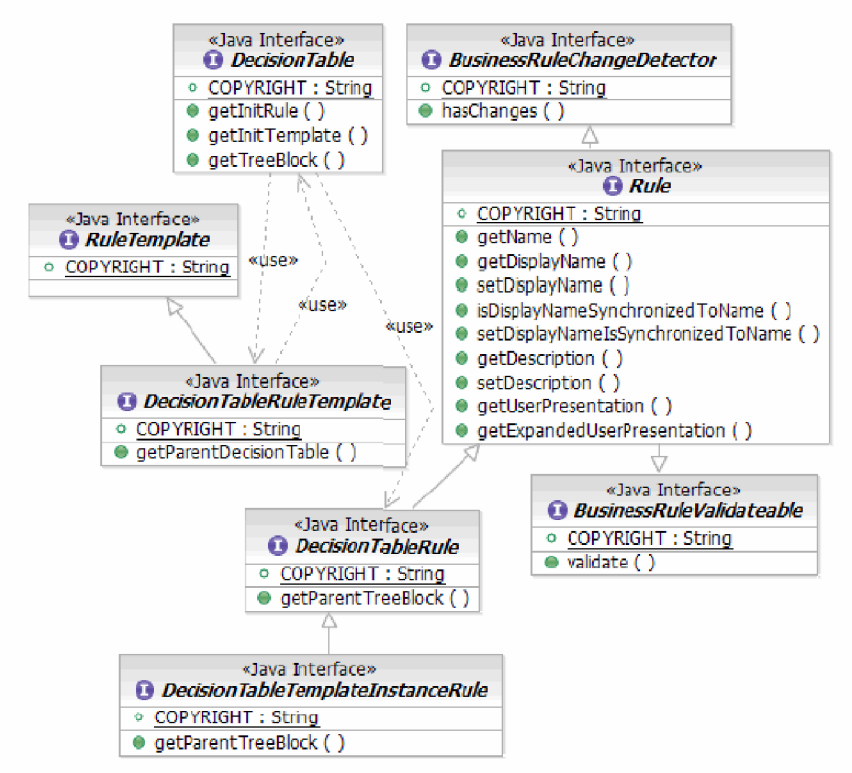


図 67. DecisionTableRule および関連するクラスのクラス・ダイアグラム

テンプレートとパラメーター

ルール・セットおよびデシジョン・テーブルのテンプレートは、共通の定義に基づきます。テンプレートには、パラメーターとユーザー表示があります。テンプレートのパラメーター値は、ルールがデプロイされた後にそれを変更できるように定義されます。

ユーザー表示の値は、ルールやさまざまなパラメーターをユーザーに分かりやすく表示するために使用できるストリング値を定義します。ストリングであるユーザー表示には、さまざまなパラメーター値に置き換えられて正しく表示されるプレースホルダーが含まれています。プレースホルダーのフォーマットは {<parameter index>} です。例えば、初期ルールの表示ストリングが「Base discount is {0} %」の場合、プレースホルダー {0} をこのパラメーター値に置換できます。表示ストリングは、ルールまたはテンプレート定義に応じて変更することはできません。ただし、プレースホルダーの値は、テンプレートの定義に応じたクライアント・アプリケーションのパラメーター値で変更できます。それぞれのテンプレートには、すべてのパラメーター値が正しく配置されたストリングを返す便利メソッド (`getExpandedUserPresentation`) が組み込まれています。

すべてのパラメーター値には固有のデータ型がありますが、パラメーター値を取得および設定するときには、ストリング・オブジェクトが使用されます。パラメーター値は、値をユーザー表示に代入するときや、パラメーターに新しい値を設定するときにも、ストリングとして処理できます。実行時には、ルールを正しく発行するために、正しいデータ型にパラメーターが変換されます。検証時には、パラメータ

一値がそのデータ型と比較されて、正しいパラメーター値であることが確認されます。例えば、パラメーターが boolean 型で、値が「T」に設定されている場合、検証ではこの値が認識されないため、問題として返されます。

テンプレート定義では、パラメーター値を制約によって制限できます。制約は、範囲または列挙として定義できます。パラメーターに対する制約は、ルールが検証されるときに適用されます。値定義がテンプレートを使用して定義されていない場合、ユーザー表示のみが使用可能になります。値定義には、テンプレートとユーザー表示の両方を使用することはできません。テンプレートを使用した場合にはテンプレート定義の表示が使用可能な唯一の表示となります。

Template クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート ID の取得
- 名前の取得
- パラメーターの取得
- ユーザー表示の取得

Parameter クラスは、以下の操作をサポートするメソッドを提供します。

- パラメーター名の取得
- パラメーターのデータ型の取得
- パラメーターに対する制約の取得
- パラメーターを定義するテンプレートの取得
- パラメーター値の作成

ParameterValue クラスは、以下の操作をサポートするメソッドを提供します。

- パラメーター名の取得
- パラメーター値の取得
- パラメーター値の設定

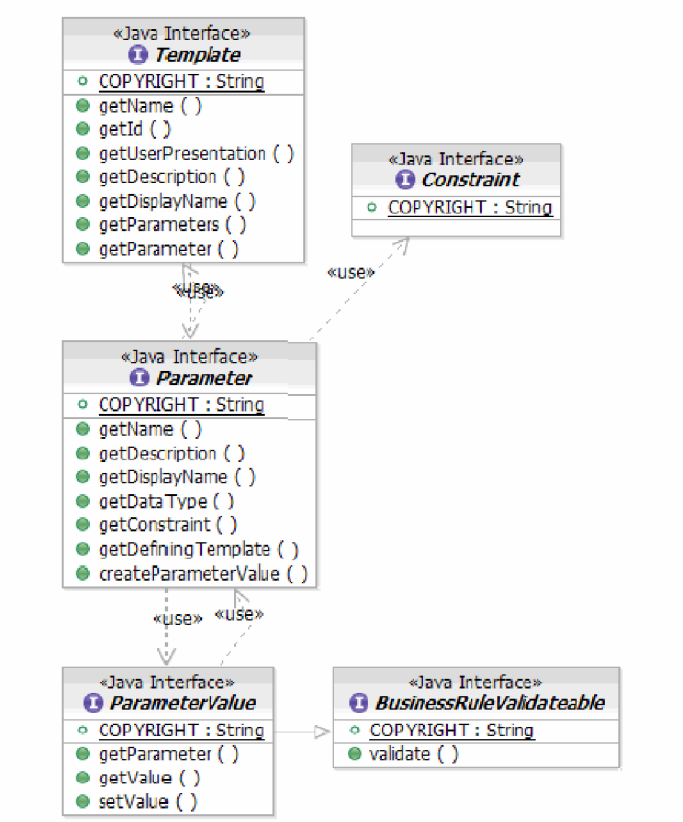


図 68. Template、Parameter、および関連クラスのクラス・ダイアグラム

検証

メイン・オブジェクトの多くには、成果物を公開する前に、その成果物の正確さと完全性を確認できる検証メソッドがあります。

API クラスを使用して変更するときに行われる検証は、serviceDeploy の実行中、または WebSphere Integration Developer で成果物を編集するときに行われる検証全体のなかのサブセットでしかありません。これは、ビジネス・ルール・グループにすでに設定された、実行時に編集可能な側面を制限するための制約によるものです。これらのクラスのユーザーは、必要に応じて常にビジネス・ルール・グループの選択テーブル、ルール・セットまたはデシジョン・テーブルを検証できます (ルール・グループのコンポーネント自体は、実行時に編集できません)。ビジネス・ルール・グループが公開されるときには、そのルール・グループの選択テーブル、ルール・セットおよびデシジョン・テーブルが検証されてから、リポジトリーに公開されます。

成果物が無効な場合、ValidationException がスローされ、検証問題のリストが提供されます。『例外処理』セクションに、さまざまな検証問題が文書化されています。

変更の追跡

すべてのオブジェクトで、そのオブジェクトや収容オブジェクトに変更が行われたかどうかを確認するために `hasChanges` メソッドを使用できます。

このメソッドは、変更を確認して、項目が変更されているビジネス・ルール・グループのみを公開するために使用できます。

BusinessRuleManager

BusinessRuleManager クラスは、ビジネス・ルール・グループ、ルール・セット、およびデシジョン・テーブルを処理するためのメイン・クラスです。

BusinessRuleManager は、名前、ターゲット名前空間、またはカスタム・プロパティーによってビジネス・ルール・グループを取得できるメソッドを備えています。また、このクラスは、ビジネス・ルール・グループ、ルール・セット、またはデシジョン・テーブルに対して行われた変更を公開するためのメソッドも備えています。

BusinessRuleManager クラスは、以下の操作をサポートするメソッドを提供します。

- すべてのビジネス・ルール・グループの取得
- 特定のターゲット名前空間のビジネス・ルール・グループの取得
- 特定の名前のビジネス・ルール・グループの取得
- 特定の名前およびターゲット名前空間のビジネス・ルール・グループの取得
- 特定のプロパティーを 1 つ含むビジネス・ルール・グループの取得
- 特定のプロパティーを複数含むビジネス・ルール・グループの取得
- ビジネス・ルール・グループの公開

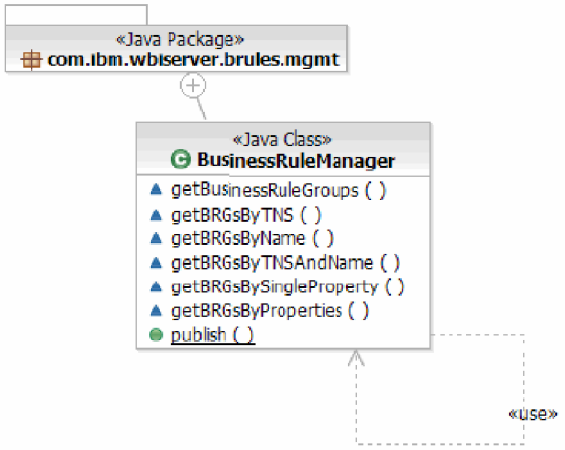


図 69. BusinessRuleManager およびパッケージのクラス・ダイアグラム

ルール・グループ・コンポーネントの照会

ルール・グループ・コンポーネントは、ユーザー定義プロパティー (名前/値のペア) を持つことができます。このプロパティーを使用すると、クラスから戻されるビジネス・ルール・グループのリストを絞り込むことができます。照会で任意の組み合わせで使用できるフィールドを以下に示します。

- ビジネス・ルール・グループ・コンポーネントのターゲット名前空間

- ビジネス・ルール・グループ・コンポーネント名
- プロパティー名
- プロパティー値

各プロパティー名は、ビジネス・ルール・グループ・コンポーネントごとに 1 回だけ定義できます。

このクラスでサポートされる照会機能は、完全な SQL 言語の小さなサブセットです。ユーザーは、SQL ステートメントを指定するのではなく、単一プロパティーのパラメーターとして、またはノード形式の複数プロパティー照会用の情報を含むツリー構造のパラメーターとして値を指定します。論理演算子ノードとプロパティー照会ノードが存在しますが、これらのノードはすべて、`QueryNode` インターフェースを実装しています。論理演算子ノードでは、ブール演算子 (AND、OR、NOT) を指定します。これらは、`QueryNodeFactory` を通じて作成します。これらの論理演算子ノードを作成する一環として、追加の `QueryNode` クラスとともに左側と右側の演算子を指定する必要があります。 これらのノードは、プロパティー照会ノードまたは別の論理演算子ノードにすることができます。プロパティー照会ノードが渡される場合、このノードには、プロパティーの名前、値、および演算子 (EQUAL (==)、NOT_EQUAL (!=)、LIKE、または NOTLIKE) が含まれます。 `QueryNode` 全体がクラスで構文解析され、永続ストレージ内の基盤となるデータに対して照会が実行されます。

LIKE 演算子または NOTLIKE 演算子を使用するときは、ワイルドカード検索がサポートされます。ワイルドカード検索では、「%」文字と「_」文字の両方がサポートされます。「%」文字を使用するのは、不明の文字、または検索時に考慮してはならない文字が不定の数だけ存在するときです。例えば、プロパティーの名前が「Department」であり、その値が「North」で始まるすべてのビジネス・ルール・グループを検索する場合は、値を「North%」として指定します。別の例として、値が「Region」で終了する「Department」がすべて必要であると想定します。この場合、値は「%Region」になります。「%」文字は、ストリングの中間で使用することもできます。例えば、ビジネス・ルール・グループのプロパティーの値が「NorthCentralRegion」、「NorthEastRegion」、および「NorthWestRegion」の場合は、値として「North%Region」を指定することができます。

不明の文字、または検索時に考慮すべきでない文字が 1 つだけ存在するときは、「_」文字を使用します。例えば、「Department」プロパティーの値が「Dept1North」、「Dept2North」、「Dept3North」、および「Dept4North」であるビジネス・ルール・グループをすべて検索する必要がある場合、値として「Dept_North」を指定すると、これらのプロパティーを持つ 4 つのビジネス・ルール・グループがすべて戻されます。「_」文字は、検索値で複数回使用できます。各インスタンスは、無視する 1 文字を示します。「_」文字は、値の先頭でも末尾でも使用することができます。例えば、値の中で 2 文字を無視する必要がある場合は、「_」を 2 つ使用します (例:「Dept__outh」)。

「%」と「_」をワイルドカードではなく、リテラル文字として扱うためには、「%」または「_」の前に「¥」エスケープ文字を指定する必要があります。例えば、プロパティー名が「%Discount」の場合、この名前を照会で使用するには、「¥%Discount」を指定する必要があります。「¥」文字をリテラル文字として使用する場合は、「¥」エスケープ文字をもう一つ使用する必要があります (例:

「Orders¥¥Customer」)。単一の「¥」文字があり、この後に「%」、「_」、または「¥」が続いていない場合は、IllegalArgumentException がスローされます。

ワイルドカード文字は、左側の演算子 (プロパティー値) でのみ使用できます。プロパティー名にワイルドカード文字を使用することはできません。

特定プロパティーの値の検索時、またはプロパティーに一致しない値の検索時、プロパティーが存在しないと、検索の考慮事項から成果物が無視されます。例えば、3 つのビジネス・ルール・グループ (A、B、C) があります。このうち、「Department」プロパティーを持つのは 2 つのみ (A と B) であり、このプロパティーの値はそれぞれ「Accounting」と「Shipping」になっています。この場合、「Department」プロパティーが「Accounting」でないビジネス・ルール・グループをすべて検索すると、「Department」プロパティーは定義されているが、その値が「Accounting」に等しくないビジネス・ルール・グループのみが戻されます (ビジネス・ルール・グループ B)。「Department」プロパティーを持たないビジネス・ルール・グループ (C) は戻されません。なぜなら、このプロパティーが定義されていないからです。

プロパティーを使用して検索する場合、*IBMSystemName* および *IBMSystemTargetNameSpace* という 2 つの特別なプロパティーを使用すると、成果物の名前および名前空間に基づいて検索を行うことができます。これらの値は、`getName` メソッドと `getTargetNameSpace` メソッドで取得することもできます。

クラスでは、照会用として以下のメソッドがサポートされます。

```
List getBRGsByTNS (string tNSName, Operator op, int skip, int threshold)
List getBRGByName(string Name, Operator op, int skip, int threshold)
List getBRGsByTNSAndName (string tNSName, Operator, tNSOp, string
 name, Operator nameOp, int skip, int threshold)
List getBRGsBySingleProperty (string propertyName, string propertyValue,
 Operator op, int skip, int threshold)
List getBRGsByProperties (QueryNode queryTree, int skip, int threshold)
```

「skip」および「threshold」パラメーターを使用すると、指定したしきい値までの部分的な結果リストを取り出すことができます。これらの両方のパラメーターにゼロの値を指定すると、完全な結果リストが戻されます。カーソルは、照会呼び出しからの結果セットに保持されません。 skip 値を使用する場合は、以前の結果セット内にあったビジネス・ルール・グループが以降の要求で戻されるように、結果セットに対して追加または削除を行うことができます。

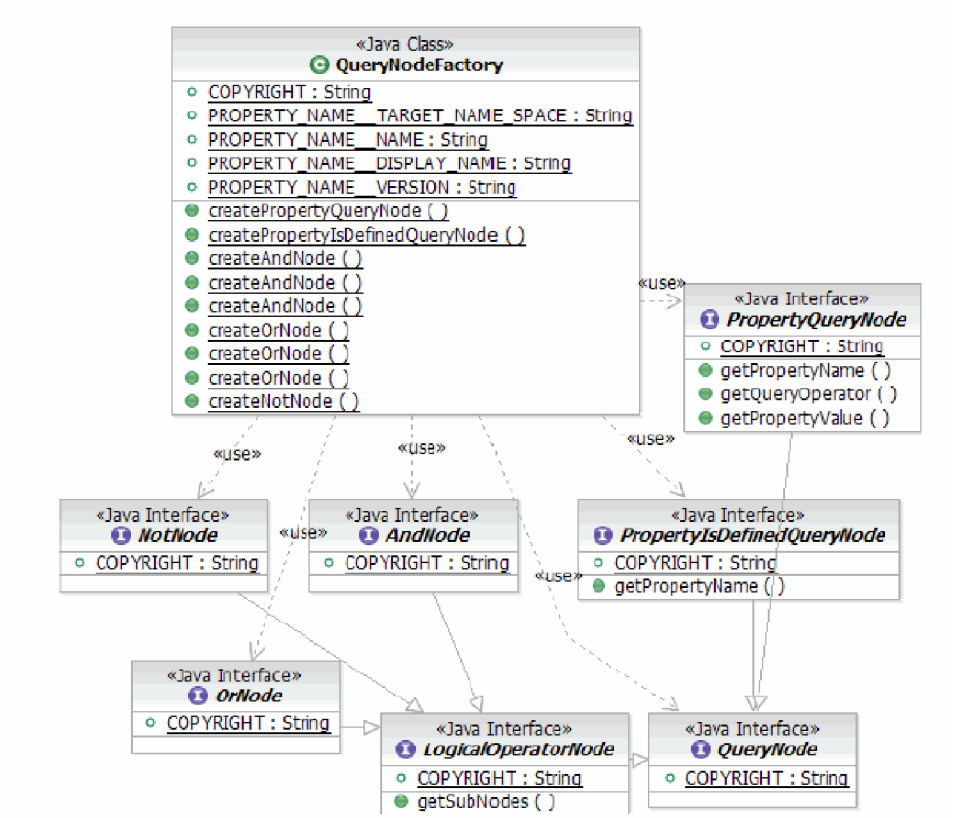


図 70. QueryNodeFactory および関連するクラスのクラス・ダイアグラム

ツリーのノードでは、検索式を指定するときにブール演算子、ワイルドカード (% およびエスケープ)、プロパティー/値のペアを使用できます。演算子は、値に対してのみ有効です。プロパティーの演算子は、常に等号 (==) です。

公開

ビジネス・ルールの変更の公開は、ビジネス・ルール・グループ・コンポーネント・レベルで行います。ユーザーは、1...n のビジネス・ルール・グループ・コンポーネントを公開できます。公開操作を実行する前に、ビジネス・ルール・グループとそれに含まれる別のオブジェクト (操作選択テーブル、ルール・セット、デシジョン・テーブルなど) に対して検証アクションが実行されます。各公開要求は、単一トランザクション内で行われます。検証時またはデータベース公開時に何らかの例外が検出された場合は、トランザクションがロールバックされ、どのビジネス・ルール・グループの変更もリポジトリーに公開されません。これにより、単一コンポーネント (例: 操作選択テーブルやルール・セット) 内で相互に依存関係を持つ変更、またはコンポーネント間の依存関係が 1 つのアトミック操作内で発生します。

公開時は検査が実行され、公開する項目が別のトランザクションによって変更されていないことが確認されます。競合の可能性を低減するために、公開メソッドでは、すべての成果物をその変更の有無にかかわらず公開すること、またはビジネス・ルール・グループ内で変更された成果物のみを公開することを選択できます。デフォルトの動作では、すべての成果物が公開されます。すべての成果物を公開するようにオプションを設定したが、別のトランザクションが成果物を変更した場合は、`ChangeConflictException` がスローされます。変更された成果物のみを公開す

ることを指定すると、競合の可能性が低減されます。変更された成果物のみを公開するようにすると、ビジネス・ルール・グループ内の 2 つの異なる成果物 (例: 2 つのルール・セット) の変更を 2 人のユーザーがリポジトリーにプッシュする場合があります。これにより、ビジネス・ルール・グループ内に互換性のない変更が生じる場合があります。こうした状況が発生する恐れがあるので、このオプションを使用するときは、注意を払う必要があります。

例外処理

例外は、成果物で検証が呼び出されるとき、または成果物が公開されるときに発生する可能性があります。検証エラーが発生すると、`ValidationException` が問題のリストとともにスローされます。成果物の公開中に、別のトランザクションが同じ成果物を公開することによって問題が発生すると、`ChangeConflictException` がスローされます。 `ChangeConflictException` 例外は、成果物の変更中に別のトランザクションが検出された時点で常にスローされます。

`SystemPropertyNotChangeableException` は、システム・プロパティー名と重複するプロパティーの変更が試行された場合にスローされます。システム・プロパティーは変更できません。

`ChangesNotAllowedException` は、公開中の成果物で設定操作が試行された場合にスローされます。

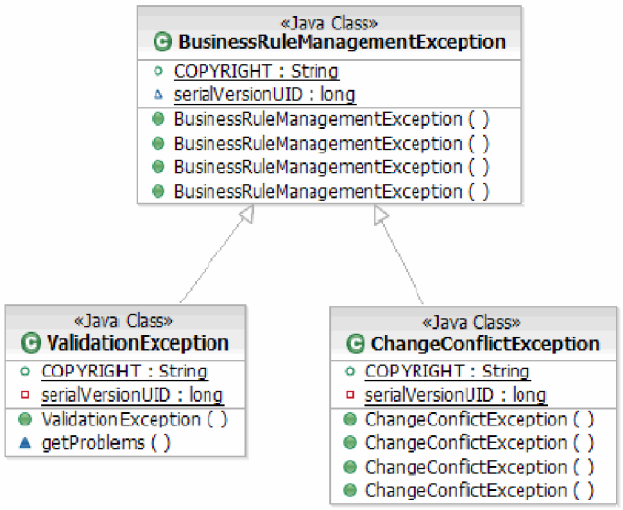


図 71. BusinessRuleManagementException および関連クラスのクラス・ダイアグラム

ビジネス・ルール・グループでの問題

ビジネス・ルール・グループの検証時、またはビジネス・ルール・グループを公開しようとするときに、ビジネス・ルール・グループの一部が有効でないと、問題が発生する可能性があります。

表 23. ビジネス・ルール・グループでの問題

例外	説明
ProblemBusRuleNotInAvailTargetList	この問題は、ルールが操作選択テーブルのデフォルト・ビジネス・ルールとして指定されているにもかかわらず、そのルール成果物がその操作の使用可能なターゲットのリストに含まれていないと発生します。この問題を回避するには、操作に使用可能なターゲットのリストから有効なビジネス・ルールを指定してください。
ProblemDuplicatePropertyName	この問題は、ビジネス・ルール・グループのシステム・プロパティーまたはユーザー定義プロパティーと重複するプロパティーを作成しようとすると発生します。この問題を回避するには、固有のプロパティー名を使用してください。
ProblemOperationContainsNoTargets	この問題は、操作にデフォルト・ルール宛先またはスケジュールされたルール宛先が設定されていないと発生します。この問題を回避するには、操作に少なくとも 1 つのルール宛先をデフォルトとして設定するか、またはスケジュールされた時間のルール宛先を設定してください。
ProblemOverlappingRanges	この問題は、操作選択レコードの開始日または終了日が、別の操作選択レコードの開始日と終了日の範囲にオーバーラップしていると発生します。日付範囲がオーバーラップしていると、呼び出すべき正しいルール宛先をビジネス・ルール・ランタイムが見つけられません。この問題を回避するには、操作の他の操作選択レコードの開始日または終了日をチェックして、オーバーラップしていないことを確認してください。
ProblemStartDateAfterEndDate	この問題は、操作選択レコードの開始日が、その選択レコードの終了日よりも後に指定されていると発生します。この問題は、開始日または終了日のないデフォルト・レコードを除くすべての操作選択レコードで発生する可能性があります。この問題を回避するには、操作選択レコードの開始日を終了日より前に指定してください。
ProblemTargetBusRuleNotSet	この問題は、操作選択レコードに指定されたルールが、使用可能なターゲット・ルールのリストに含まれていないと発生します。この問題を回避するには、使用可能なターゲット・リストに含まれているルールを指定してください。
ProblemTNSAndNameAlreadyInUse	この問題は、新規に作成するビジネス・ルールのターゲット名前空間と名前が、ルール・セットまたはデシジョン・テーブルですでに使用されていると発生します。このチェックは、現行のビジネス・ルール・グループに関連付けられたすべてのルール・セットとデシジョン・テーブルだけでなく、リポジトリーに保管されたルール成果物でも行われます。この問題を回避するには、異なるターゲット名前空間または名前を使用してください。
ProblemWrongOperationForOpSelectionRecord	この問題は、新しい操作選択レコードを操作選択レコード・リストに追加するときに、新規レコードの操作がリスト内のレコードの操作と一致しないと発生します。この問題を回避するには、正しい操作選択レコード・リスト・オブジェクトで `newOperationSelectionRecord` メソッドを使用して新しい操作を作成してください。

ルール・セットおよびデシジョン・テーブルでの問題

表 24. ルール・セットおよびデシジョン・テーブルでの問題

例外	説明
ProblemInvalidBooleanValue	この問題は、ルール・セットに含まれるルール・テンプレートのパラメーター、あるいはデシジョン・テーブルに含まれるアクション値または条件値が、ブール型のパラメーターとして「true」または「false」以外の値を受け取ると発生します。例えば、「T」や「F」は正しくないパラメーター値です。この問題を回避するために、ブール型のパラメーターを処理するときには「true」または「false」の値を使用してください。
ProblemParmNotDefinedInTemplate	この問題は、値を指定するテンプレート・パラメーターが、そのテンプレートの有効なパラメーターのリストに定義されていないと発生します。パラメーターは、テンプレートで設定する前に確認してください。この問題は、RuleTemplate、 TreeActionValueTemplate、または TreeConditionValueTemplate テンプレートで発生する可能性があります。
ProblemParmValueListContainsUnexpectedValue	この問題は、テンプレートに渡されたパラメーターは有効であるものの、そのテンプレートのパラメーターの数としては多過ぎる場合に発生します。パラメーターの数を削減する必要があります。この問題は、RuleTemplate、 TreeActionValueTemplate、または TreeConditionValueTemplate テンプレートで発生する可能性があります。
ProblemRuleBlockContainsNoRules	この問題は、ルール・セットに含まれるルール・ブロック内のすべてのルールが除去されているときに、ルール・セットの検証または公開が試行されると発生します。ルール・セットに含まれるルール・ブロックには、少なくとも 1 つのルールがなければなりません。
ProblemTemplateNotAssociatedWithRuleSet	この問題は、ルール・セットに追加しようとしているルールが、そのルール・セットで定義されていないテンプレートで作成されている場合に発生します。この問題を回避するには、新しいルールを作成するときに、ルール・セットで定義されたテンプレートを使用してください。
ProblemRuleNameAlreadyInUse	この問題は、ルール・セットのルール・ブロックに追加しようとしているルールの名前が、そのルール・ブロック内にある既存のルールの名前と同じ場合に発生します。この問題を回避するには、新しいルールを追加する前に、そのルールの名前を確認してください。
ProblemTemplateParameterNotSpecified	この問題は、ルール・セットのルールまたはデシジョン・テーブルに含まれるアクション値あるいは条件値に対してテンプレートを更新するときに、パラメーターが含まれていないと発生します。この問題を回避するには、テンプレートのすべてのパラメーターを指定してください。

表 24. ルール・セットおよびデシジョン・テーブルでの問題 (続き)

例外	説明
ProblemTypeConversionError	この問題は、テンプレートのパラメーターを適切な型に変換できない場合に発生します。すべてのパラメーターはストリング・オブジェクトとして処理されてから、パラメーターの型 (boolean、byte、 short、 int、 long、 float、および double) に変換されます。パラメーター値のストリングを、そのパラメーターに指定された型に変換できないと、このエラーが発生します。この問題を回避するには、パラメーターの型 (boolean、 byte、 short、 int、long、 float、および double) に変換可能なストリングを指定してください。
ProblemValueViolatesParmConstraints	この問題は、パラメーターが、そのパラメーターに対してテンプレートで定義されている列挙または値の範囲内にない場合に発生します。この問題は、ルール・セットのルール・テンプレート、あるいはデシジョン・テーブルのアクション値または条件値のテンプレートで列挙または範囲によって制限されているパラメーターで発生する可能性があります。この問題を回避するには、列挙に含まれる値を使用してください。
ProblemInvalidActionValueTemplate	この問題は、テンプレート・インスタンスをツリー・アクションの値定義に設定しようとしたけれども、対応するテンプレートがそのツリー・アクションには使用できない場合に発生します。この問題を回避するには、正しいテンプレートを使用してツリー・アクションの値定義を作成してください。
ProblemInvalidConditionValueTemplate	この問題は、テンプレート・インスタンスをケース・エッジの条件定義に設定しようとしたけれども、対応するテンプレートがそのケース・エッジには使用できない場合に発生します。この問題を回避するには、正しいテンプレートを使用してケース・エッジの条件定義を作成してください。
ProblemTreeActionIsNull	この問題は、新しい条件値を作成するときに、アクションがテンプレート・インスタンスで設定されていない場合に発生します。ActionNode のテンプレートを使用して新しいテンプレート・インスタンスを作成し、そのインスタンスを `TreeActions` のリストに設定してください。

許可

クラスでは、どのレベルの許可もサポートされません。独自の形式の許可を追加するのは、クラスを使用するクライアント・アプリケーションに任されています。

例

さまざまなクラスを使用してビジネス・ルール・グループを取得する方法を示す例や、ルール・セットとデシジョン・テーブルを変更する方法を示す例が数多く提供されています。これらの例は、プロジェクト交換ファイル (ZIP) で提供されています。このファイルは WebSphere Integration Developer にインポートして参照したり、再使用したりできます。

プロジェクト交換には、以下の多数のプロジェクトがあります。

- **BRMgmtExamples** - さまざまな例で使用するビジネス・ルール成果物が含まれるモジュール・プロジェクト。

- **BRMgmt** - com.ibm.websphere.sample.brules.mgmt パッケージに置かれている例が含まれる Java プロジェクト。
- **BRMgmtDriverWeb** - サンプルを実行するためのインターフェースが含まれる Web プロジェクト。

これらの例は EAR ファイル (BRMgmtExamples.ear) としても提供されています。この EAR ファイルは、WebSphere Process Server にインストールしてから実行します。例には Web インターフェースが提供されています。これらの例では、クラスを使用して成果物を取得し、変更を加え、変更を公開する方法を示すことに重点を置いているため、この Web インターフェースは意図的に単純なものになっています。機能性に優れた Web インターフェースとなるようには意図されていません。ただし、これらのクラスは、堅固な Web サービスを作成するために簡単に使用でき、ビジネス・ルールの変更を主な目的とする他の Java アプリケーションで使用できます。

注: プロジェクト交換の例や EAR ファイルを「Business Rule Management Programming Guide for WebSphere Process Server V6.1」からダウンロードできます。

サンプル・アプリケーションは、WebSphere Process Server v6.1 にインストールできます。索引ページにアクセスするには、以下を指定します。

```
http://<hostname>:<port>/BRMgmtDriverWeb/
```

例えば、http://localhost:9080/BRMgmtDriverWeb/ となります。

例を実行すると、ルール成果物が変更されることになります。すべての例を実行した場合、再びすべての例で同じ結果を得るためには、アプリケーションを再インストールする必要があります。

それぞれの例について、完全なサンプル・コードと Web ブラウザーに表示される結果を記載して詳しく説明します。

共通操作を実行し、サンプル Web アプリケーション内に情報を表示できるように、多数の追加クラスが作成されています。 Formatter クラスおよび RuleArtifactUtility クラスについて詳しくは、付録を参照してください。

これらの例を十分に理解するには、WebSphere Integration Developer 内のさまざまな成果物について調査することが大いに役立ちます。

例 1: すべてのビジネス・ルール・グループを取得してプリントする

この例では、すべてのビジネス・ルール・グループを取得して、各ビジネス・ルール・グループの属性、プロパティー、および操作をプリントします。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;
```

ビジネス・ルール管理クラスとしては、必ず com.ibm.wbiserver.brules.mgmt パッケージに含まれるクラスを使用してください。com.ibm.wbiserver.brules パッケージやその他のパッケージに含まれるクラスは使用できません。それらの他のパッケージは、IBM 内部クラスで使用されます。

```
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import
com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;

public class Example1 {
static Formatter out = new Formatter();
static public String executeExample1()
{
try
{
out.clear();
```

BusinessRuleManager クラスは、ビジネス・ルール・グループを取得したり、ビジネス・ルール・グループの変更を公開したりする際に使用するメイン・クラスです。このクラスの用途には、ルール・セットやデシジョン・テーブルなどのルール成果物の操作および変更も含まれます。 BusinessRuleManager クラスには、名前と名前空間、およびプロパティーを渡すことで特定のビジネス・ルール・グループを容易に取得できるメソッドが多数用意されています。

```
// すべてのビジネス・ルール・グループを取得します。
List<BusinessRuleGroup> brgList = BusinessRuleManager
  .getBusinessRuleGroups(0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// ビジネス・ルール・グループのリストを繰り返し処理します。
while (iterator.hasNext())
{
  brg = iterator.next();
  // 各ビジネス・ルール・グループの属性を出力します。
  out.println1nBold("Business Rule Group");
```

ビジネス・ルール・グループの基本属性を取得して表示することができます。

```
out.println("Name: " + brg.getName());
out.println("Namespace: " +
brg.getTargetNameSpace());
out.println("Display Name: " +
brg.getDisplayName());
out.println("Description: " + brg.getDescription());
out.println("Presentation Time zone: "
     + brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());
```

ビジネス・ルール・グループのプロパティーも取得して変更することができます。

```
PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// プロパティーの名前と値を出力します。
while (propIterator.hasNext())
{
```

```
prop = propIterator.next();
out.println("Property Name: " +
prop.getName());
out.println("Property Value: " +
prop.getValue());
}
```

ビジネス・ルール・グループに対する操作も使用できます。これらの操作によって、ルール・セットやデシジョン・テーブルなどのビジネス・ルール成果物を取得できます。

```
List<Operation> opList = brg.getOperations();

Iteration<Operation> opIterator = opList.iterator();
Operation op = null;
// ビジネス・ルール・グループの操作を出力します。
while (opIterator.hasNext())
{
op = opIterator.next();
out.println("Operation: " + op.getName());
}
out.println("");}
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}
```

例 1 の Web ブラウザー出力

例 1 の実行

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues
Operation: getApprover

Business Rule Group
Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
```

```
Property Value: General
Property Name: RuleType
Property Value: messages
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ConfigurationValues
Property Name: IBMSystemDisplayName
Property Value: ConfigurationValues
Operation: getMessages

Business Rule Group
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules
Operation: calculateOrderDiscount
Operation: calculateShippingDiscount
```

例 2: ビジネス・ルール・グループ、ルール・セット、およびデシジョン・テーブルを取得してプリントする

この例では、例 1 の関数と同じ処理に加えて、各操作の選択テーブル、デフォルト・ビジネス・ルール宛先 (ルール・セットまたはデシジョン・テーブルのいずれか)、操作にスケジュールされたその他のビジネス・ルールをプリントします。ルール・セットとデシジョン・テーブルの両方をプリントします。

この例の大部分は同じですが、完全を期して記載しています。

```
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
public class Example2
{
 status Formatter out = new Formatter();
 static public String executeExample2()
 {
  try
  {
    out.clear();
```

この例では、名前を基準に特定のビジネス・ルール・グループを取得します。

```
// すべてのビジネス・ルール・グループを取得します。
List<BusinessRuleGroup> brgList = BusinessRuleManager
  .getBRGsByName("DiscountRules",
  QueryOperator.EQUAL, 0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// ビジネス・ルール・グループのリストを繰り返し処理します。
while (iterator.hasNext())
 {
 brg = iterator.next();
 // 各ビジネス・ルール・グループの属性を出力します。
 out.printlnBold("Business Rule Group");
 out.println("Name: " + brg.getName());
 out.println("Namespace: " +
  brg.getTargetNameSpace());
 out.println("Display Name: " +
  brg.getDisplayName());
 out.println("Description: " + brg.getDescription());
 out.println("Presentation Time zone: "
  + brg.getPresentationTimezone());
 out.println("Save Date: " + brg.getSaveDate());

 PropertyList propList = brg.getProperties();

 Iterator<Property> propIterator =
 propList.iterator();
 Property prop = null;
 // プロパティーの名前と値を出力します。
 while (propIterator.hasNext())
 {
   prop = propIterator.next();
   out.println("Property Name: " +
   prop.getName());
   out.println("Property Value: " +
   prop.getValue());
 }
```

各操作の選択テーブルには、さまざまなルール成果物と、そのルール成果物がアクティブになるスケジュールがリストされます。デフォルト・ビジネス・ルールは、操作ごとに指定できます。デフォルト・ビジネス・ルールの指定やビジネス・ルールのスケジューリングは必須ではありませんが、少なくとも 1 つのデフォルト・ビジネス・ルールまたは 1 つのスケジュールされたビジネス・ルールが必要です。これをサポートするため、デフォルト・ビジネス・ルールを使用する前にこれが null であるかどうかを確認するか、または OperationSelectionRecordList のサイズを確認するようにしてください。

```
List<Operation> opList = brg.getOperations();

Iterator<Operation> opIterator = opList.iterator();
Operation op = null;
out.println("");
out.printlnBold("Operations");
// ビジネス・ルール・グループの操作を出力します。
while (opIterator.hasNext())
 {
   op = opIterator.next();
   out.printBold("Operation: ");
   out.println(op.getName());

   // 操作のデフォルト・ビジネス・ルールを取得します。
   BusinessRule defaultRule =
```

```
op.getDefaultBusinessRule();
// デフォルト・ルールが検出された場合には、ルールのタイプに適した
// メソッドを使用して、そのビジネス・ルールをプリントします。
if (defaultRule != null)
{
  out.printlnBold("Default Destination:");
```

デフォルト・ビジネス・ルールのタイプは、RuleSet または DecisionTable です。これは、ルール成果物を処理するために正しいタイプにキャストできます。

```
  if (defaultRule instanceof RuleSet)
     out.println(RuleArtifactUtility.
     intRuleSet(defaultRule));
  else
     out.print(RuleArtifactUtility.
     tDecisionTable(defaultRule));
}
OperationSelectionRecordList
opSelectionRecordList = op
  .getOperationSelectionRecordList()
  ;

Iterator<OperationSelectionRecord>
opSelRecordIterator = opSelectionRecordList
  .iterator();
OperationSelectionRecord record = null;
```

OperationSelectionRecord は、ルール成果物と、そのルール成果物をアクティブにするためのスケジュールで構成されます。

```
while (opSelRecordIterator.hasNext())
{
  out.printlnBold("Scheduled
  Destination:");
  record = opSelRecordIterator.next();

  out.println("Start Date: " +
  record.getStartDate()
    + " - End Date: " +
    record.getEndDate());
  BusinessRule ruleArtifact = record
    .getBusinessRuleTarget();

  if (ruleArtifact instanceof RuleSet)
    out.println(RuleArtifactUtility.pr
    intRuleSet(ruleArtifact));
  else
    out.print(RuleArtifactUtility.prin
    tDecisionTable(ruleArtifact));
 }
}
}
out.println("");
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
return out.toString();
}
}
```

例

例 2 の Web ブラウザー出力

Business Rule Group
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules

Operations
Operation: calculateOrderDiscount
Default Destination:
Rule Set
Name: calculateOrderDiscount
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: CopyOrder
Display Name: CopyOrder
Description: null
Expanded User Presentation: null
User Presentation: null
Rule: FreeGiftInitialization
Display Name: FreeGiftInitialization
Description: null
Expanded User Presentation: Product ID for Free Gift = 5001AE80 Quantity = 1 Cost = 0.0 Description = Free gift for discounted order
User Presentation: Product ID for Free Gift = {0} Quantity = {1} Cost = {2} Description = {3}Parameter Name: param0
Parameter Value: 5001AE80
Parameter Name: param1
Parameter Value: 1
Parameter Name: param2
Parameter Value: 0.0
Parameter Name: param3
Parameter Value: Free gift for discounted order
Rule: Rule1
Display Name: Rule1
Description: null
Expanded User Presentation: If customer is gold status, then apply a discount of 20.0 and include a free gift
User Presentation: If customer is {0} status, then apply a discount of {1} and include a free gift
Parameter Name: param0
Parameter Value: gold
Parameter Name: param1
Parameter Value: 20.0
Rule: Rule2
Display Name: Rule2
Description: null
Expanded User Presentation: If customer.status == silver, then provide a discount of 15.0
User Presentation: If customer.status == {0}, then provide a discount of {1}
Parameter Name: param0
Parameter Value: silver
Parameter Name: param1
Parameter Value: 15.0
Rule: Rule3

```
Display Name: Rule3
Description: Template for non-gold customers
Expanded User Presentation: If customer.status == bronze, then provide a discount of
10.0
User Presentation: If customer.status == {0}, then provide a discount of {1}
Parameter Name: param0
Parameter Value: bronze
Parameter Name: param1
Parameter Value: 10.0

Operation: calculateShippingDiscount
Default Destination:
Decision Table
Name: calculateShippingDiscount
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Init Rule: Rule1
Display Name: Rule1
Description: null
Extended User Presentation: null
User Presentation: null
```

例 3: AND で結合した複数のプロパティーを基準にビジネス・ルール・グループを取得する

この例も例 1 と同様ですが、Department という名前のプロパティーの値が「accounting」に設定されていると同時に、RuleType という名前のプロパティーの値が「regulatory」に設定されているビジネス・ルール・グループのみを取得します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example3
{
static Formatter out = new Formatter();
static public String executeExample3()
{
  try
  {
    out.clear();
```

ビジネス・ルール・グループのクエリーは、ツリー構造を形成する照会ノードで構成されます。それぞれの照会ノードは、左側の項、右側の項および条件で構成されます。各項と右側の項には、さらに別の照会ノードを指定できます。この例では、2 つのプロパティー値の組み合わせを基準にビジネス・ルール・グループを取得します。

```
   // 2 つの条件を基準にビジネス・ルール・グループを取得します。
   // それぞれの条件に対して PropertyQueryNode を作成します。
   PropertyQueryNode propertyNode1 = QueryNodeFactory
     .createPropertyQueryNode("Department",
```

```
      QueryOperator.EQUAL,"Accounting");
    PropertyQueryNode propertyNode2 = QueryNodeFactory
      .createPropertyQueryNode("RuleType", QueryOperator.EQUAL,
      "regulatory");
    // 2 つの PropertyQueryNode を結合して AND ノードを作成します。
    AndNode andNode =
    QueryNodeFactory.createAndNode(propertyNode1, propertyNode2);

    // andNode を使用してビジネス・ルール・グループを検索します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByProperties(andNode, 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;
    // ビジネス・ルール・グループのリストを繰り返し処理します。
    while (iterator.hasNext())
    {
      brg = iterator.next();
      // 各ビジネス・ルール・グループの属性を出力します。
      out.println1nBold("Business Rule Group");
      out.println("Name: " + brg.getName());
      out.println("Namespace: " +
      brg.getTargetNameSpace());
      out.println("Display Name: " + brg.getDisplayName());
      out.println("Description: " + brg.getDescription());
      out.println("Presentation Time zone: "
      + brg.getPresentationTimezone());
      out.println("Save Date: " + brg.getSaveDate());

      PropertyList propList = brg.getProperties();

      Iterator<Property> propIterator =
      propList.iterator();
      Property prop = null;
      // プロパティーの名前と値を出力します。
      while (propIterator.hasNext())
      {
        prop = propIterator.next();
        out.println("¥t Property Name: " +
        prop.getName());
        out.println("¥t Property Value: " +
        prop.getValue());
      }
    }
  } catch (BusinessRuleManagementException e)
  {
  e.printStackTrace();
  out.println(e.getMessage());
  }
  return out.toString();
  }
}
```

例

例 3 の Web ブラウザー出力

例 3 の実行

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
```

```
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues
```

例 4: OR で結合した複数のプロパティーを基準にビジネス・ルール・グループを取得する

この例は例 3 と同様ですが、Department という名前のプロパティーの値が「accounting」に設定されているか、または RuleType という名前のプロパティーの値が「monetary」に設定されているビジネス・ルール・グループのみを取得します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example4
{
static Formatter out = new Formatter();
static public String executeExample4()
{
  try
  {
    out.clear();
```

さまざまなプロパティーを使用して照会を構成し、異なるビジネス・ルール・グループを返します。

```
    // 2 つの条件を基準にビジネス・ルール・グループを取得します。
    // それぞれの条件に対して PropertyQueryNode を作成します。
    PropertyQueryNode propertyNode1 = QueryNodeFactory
      .createPropertyQueryNode("Department",
      QueryOperator.EQUAL,"Accounting");
    PropertyQueryNode propertyNode2 = QueryNodeFactory
      .createPropertyQueryNode("RuleType",
      QueryOperator.EQUAL,"monetary");
    // 2 つの PropertyQueryNode を結合して OR ノードを作成します。
    OrNode orNode =
    QueryNodeFactory.createOrNode(propertyNode1,
      propertyNode2);
    // orNode を使用してビジネス・ルール・グループを検索します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByProperties(orNode, 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;
```

```
    // ビジネス・ルール・グループのリストを繰り返し処理します。
    while (iterator.hasNext())
    {
      brg = iterator.next();
      // 各ビジネス・ルール・グループの属性を出力します。
      out.printlnBold("Business Rule Group");
      out.println("Name: " + brg.getName());
      out.println("Namespace: " +
      brg.getTargetNameSpace());
      out.println("Display Name: " + brg.getDisplayName());
      out.println("Description: " + brg.getDescription());
      out.println("Presentation Time zone: "
        + brg.getPresentationTimezone());
      out.println("Save Date: " + brg.getSaveDate());

      PropertyList propList = brg.getProperties();

      Iterator<Property> propIterator =
      propList.iterator();
      Property prop = null;
      // プロパティーの名前と値を出力します。
      while (propIterator.hasNext())
      {
        prop = propIterator.next();
        out.println("¥t Property Name: " +
        prop.getName());
        out.println("¥t Property Value: " +
        prop.getValue());
      }
      out.println("");
    }
  } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

例

例 4 の Web ブラウザー出力

例 4 の実行

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues

Business Rule Group
```

```
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules
```

例 5: 複雑な照会を使用してビジネス・ルール・グループを取得する

この例では、例 3 と例 4 を組み合わせて、さらに複雑な照会の作成方法を説明します。この例では、2 つの照会条件を組み合わせた照会を使用して検索を行います。最初の照会条件は、Department という名前のプロパティーの値が「General」に設定されているか、MissingProperty という名前のプロパティーの値が「somevalue」に設定されているビジネス・ルール・グループを取得することです。この条件に、RuleType という名前のプロパティーの値が「messages」に設定されているという条件を、AND を使用して結合します。

付録には、ビジネス・ルール・グループを照会するその他の例が記載されています。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example5
{
static Formatter out = new Formatter();
static public String executeExample5()
{
  try
  {
    out.clear();

    // 3 つの条件を基準にビジネス・ルール・グループを取得します。条件の
    // うちの 2 つは結合して 1 つの OR ノードを構成します。
    // OR ノードを構成するそれぞれの条件に対して PropertyQueryNode を作成します。
    PropertyQueryNode propertyNode1 = QueryNodeFactory
      .createPropertyQueryNode("Department",
      QueryOperator.EQUAL,"General");
    PropertyQueryNode propertyNode2 = QueryNodeFactory
```

```
    .createPropertyQueryNode("MissingProperty",
  QueryOperator.EQUAL, "SomeValue");
  // 2 つの PropertyQueryNode を結合して OR ノードを作成します。
  OrNode orNode =
  QueryNodeFactory.createOrNode(propertyNode1, propertyNode2);
  // 3 つ目の PropertyQueryNode を作成します。
    PropertyQueryNode propertyNode3 = QueryNodeFactory
    .createPropertyQueryNode("RuleType",
  QueryOperator.EQUAL,"messages");
```

左側の条件と右側の条件を結合して AND ノードを作成します。この AndNode が照会ツリーのルートになります。

```
  // OR ノードと 3 つ目の PropertyQueryNode を結合します。
  AndNode AndNode andNode =
  QueryNodeFactory.createAndNode(propertyNode3, orNode);

  List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByProperties(andNode, 0, 0);

  Iterator<BusinessRuleGroup> iterator = brgList.iterator();

  BusinessRuleGroup brg = null;
  // ビジネス・ルール・グループのリストを繰り返し処理します。
  while (iterator.hasNext())
  {
    brg = iterator.next();
    // 各ビジネス・ルール・グループの属性を出力します。
    out.println1nBold("Business Rule Group");
    out.println("Name: " + brg.getName());
    out.println("Namespace: " +
      brg.getTargetNameSpace());
    out.println("Display Name: " + brg.getDisplayName());
    out.println("Description: " + brg.getDescription());
    out.println("Presentation Time zone: "
      + brg.getPresentationTimezone());
    out.println("Save Date: " + brg.getSaveDate());
    PropertyList propList = brg.getProperties();

    Iterator<Property> propIterator =
    propList.iterator();
    Property prop = null;
    // プロパティーの名前と値を出力します。
    while (propIterator.hasNext())
    {
      prop = propIterator.next();
      out.println("¥t Property Name: " +
      prop.getName());
      out.println("¥t Property Value: " +
      prop.getValue());
    }
  }
 }   catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}
```

例

例 5 の Web ブラウザー出力

例 5 の実行

```
Business Rule Group
Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: General
Property Name: RuleType
Property Value: messages
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ConfigurationValues
Property Name: IBMSystemDisplayName
Property Value: ConfigurationValues
```

例 6: ビジネス・ルール・グループのプロパティーを更新して公開する

この例では、ビジネス・ルール・グループのプロパティーを更新してから、ビジネス・ルール・グループを公開します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example6
{
static Formatter out = new Formatter();

static public String executeExample6()
{
 try
 {
   out.clear();
   out.printlnBold("Business Rule Group before publish:");
   // 単一のプロパティー値を基準にビジネス・ルール・グループを取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsBySingleProperty("Department",
     QueryOperator.EQUAL,"General", 0, 0);

   if (brgList.size() > 0)
   {
     // リストから最初のビジネス・ルール・グループを取得します。
     BusinessRuleGroup brg = brgList.get(0);
     // ビジネス・ルール・グループからプロパティーを取得します。
     UserDefinedProperty userDefinedProperty =
     (UserDefinedProperty) brg
       .getProperty("Department");

     out.println("Business Rule Group: " + brg.getName());
     out.println("Department Property value: "
     + brg.getProperty("Department").getValue());
```

getProperty メソッドは参照によってプロパティーを返すため、そのプロパティーを変更するとビジネス・ルール・グループが直接変更されます。

```
    // brg のプロパティー値を変更します。
    // これにより、brg オブジェクトのプロパティー値が直接更新されます。
    userDefinedProperty.setValue("GeneralConfig");
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();
    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);
```

ビジネス・ルール・グループの変更内容を公開するには、BusinessRuleManager クラスを使用します。変更を公開するには、1 つの項目のみを公開する場合でも、BusinessRuleManager の publish メソッドにリストを渡します。

```
    // 更新されたビジネス・ルール・グループを含むリストを公開します。
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // ビジネス・ルール・グループを再取得し、変更内容が
    // 公開されていることを確認します。
    out.println1nBold("Business Rule Group after publish:");
    brgList = BusinessRuleManager
    .getBRGsBySingleProperty("Department",
    QueryOperator.EQUAL, "GeneralConfig", 0, 0);

    brg = brgList.get(0);

    out.println("Business Rule Group: " + brg.getName());
    // プロパティー値を表示して変更を示します。
    out.println("Department Property value: "
    + brg.getProperty("Department").getValue());
  }
 } catch (BusinessRuleManagementException e)
 {
 e.printStackTrace();
 out.println(e.getMessage());
 }
return out.toString();
}
}
```

例

例 6 の Web ブラウザー出力

例 6 の実行

```
Business Rule Group before publish:
Business Rule Group: ConfigurationValues
Department Property value: General

Business Rule Group after publish:
Business Rule Group: ConfigurationValues
Department Property value: GeneralConfig
```

例 7: 複数のビジネス・ルール・グループのプロパティーを更新して公開する

この例では、複数のビジネス・ルール・グループのプロパティーを更新して公開します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example7
{
static Formatter out = new Formatter();

static public String executeExample7()
{
  try
  {
    out.clear();
    out.println1nBold("Business Rule Group before publish:");
    // 単一のプロパティー値を基準にビジネス・ルール・グループを取得します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsBySingleProperty("Department",
      QueryOperator.EQUAL, "Accounting", 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;

    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // すべてのビジネス・ルール・グループを繰り返し処理して、
    // プロパティーを変更します。
    while (iterator.hasNext())
    {
      // ビジネス・ルール・グループからプロパティーを取得します。
      brg = iterator.next();

      out.println("Business Rule Group: " + brg.getName());

      // ビジネス・ルール・グループからプロパティーを取得します。
      UserDefinedProperty prop = (UserDefinedProperty) brg
        .getProperty("Department");
      out.println("Department Property value: "
      +
      brg.getProperty("Department").getValue())
      ;

      // brg のプロパティー値を変更します。
      // これにより、brg オブジェクトのプロパティー値が直接更新されます。
      prop.setValue("Finance");
```

変更したビジネス・ルール・グループのそれぞれを追加します。

```
      // 変更したビジネス・ルール・グループをリストに追加します。
      publishList.add(brg);
      }

      // 更新されたビジネス・ルール・グループを含むリストを公開します。
      BusinessRuleManager.publish(publishList, true);

      out.println("");
```

```
      // ビジネス・ルール・グループを再取得し、変更内容が
      // 公開されていることを確認します。
      out.println1nBold("Business Rule Group after
      publish:");

      brgList = BusinessRuleManager
        .getBRGsBySingleProperty("Department",
        QueryOperator.EQUAL,
        "Finance", 0, 0);
      iterator = brgList.iterator();

      while (iterator.hasNext())
      {
        brg = iterator.next();
        out.println("Business Rule Group: " +
        brg.getName());
        out.println("Department Property value: "
         +
         brg.getProperty("Department").getVa
         lue());
      }
} catch (BusinessRuleManagementException e)
{
  e.printStackTrace();
  out.println(e.getMessage());
}
return out.toString();
}
}
```

例

例 7 の Web ブラウザー出力

例 7 の実行

```
Business Rule Group before publish:
Business Rule Group: ApprovalValues
Department Property value: Accounting
Business Rule Group: DiscountRules
Department Property value: Accounting

Business Rule Group after publish:
Business Rule Group: ApprovalValues
Department Property value: Finance
Business Rule Group: DiscountRules
Department Property value: Finance
```

例 8: ビジネス・ルール・グループのデフォルト・ビジネス・ルールを変更する

この例では、デフォルト・ビジネス・ルールを、特定の操作に使用可能なターゲット・リストに含まれる別のビジネス・ルールに変更します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
```

```
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example8
{
static Formatter out = new Formatter();

static public String executeExample8()
{
  try
  {
    out.clear();

  // ターゲット名前空間と名前によってビジネス・ルール・グループを取得します。
  List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
      "http://BRSamples/com/ibm/websphere
      /sample/brules",
      QueryOperator.EQUAL,
      "DiscountRules",
      QueryOperator.EQUAL, 0, 0);

  if (brgList.size() > 0)
  {
    out.printlnBold("Business Rule Group before publish:");
    // リストから最初のビジネス・ルール・グループを取得します。
    // ターゲット名前空間と名前の組み合わせは固有であることから、
    // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
    BusinessRuleGroup brg = brgList.get(0);

    out.print("Business Rule Group: ");
    out.println(brg.getName());

    // デフォルト・ビジネス・ルールを更新する
    // ビジネス・ルール・グループの操作を取得します。
    Operation op =
    brg.getOperation("calculateShippingDiscount");
```

デフォルト・ビジネス・ルールを取得した後、操作に使用可能なターゲット・リストに含まれる別のルールで更新します。ルール・セットとデシジョン・テーブルはそれぞれの操作に固有であるため、デフォルトに設定したり、操作の別のスケジュールに使用できるのは、その操作のビジネス・ルール成果物のみです。

```
    // 操作のデフォルト・ビジネス・ルールを取得します。
    BusinessRule defaultRule =
    op.getDefaultBusinessRule();
    out.print("Default Rule: ");
    out.println(defaultRule.getName());

    // この操作に使用可能なビジネス・ルールのリストを取得します。
    List<BusinessRule> ruleList =
    op.getAvailableTargets();

    Iterator<BusinessRule> iterator =
    ruleList.iterator();
    BusinessRule rule = null;

    // 現行のデフォルト・ビジネス・ルールとは
    // 異なるビジネス・ルールを
    // 検索します。
    while (iterator.hasNext())
    {
      rule = iterator.next();
      if
      (!defaultRule.getName().equals(rule.getName()))
      {
```

操作オブジェクトのデフォルト・ビジネス・ルールを設定します。デフォルト・ビジネス・ルールをヌルに設定すると、あらゆるデフォルト・ビジネス・ルールが操作から除去されますが、すべての操作にはデフォルト・ビジネス・ルールを指定することをお勧めします。

```
        // デフォルト・ビジネス・ルールを異なる
        // ビジネス・ルールに設定します。
        // この変更は、操作オブジェクトに対して直接
        // 行われます。
        op.setDefaultBusinessRule(rule);
        break;
      }
    }
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();
    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);
    // 更新されたビジネス・ルール・グループを含むリストを公開します。
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // ビジネス・ルール・グループを再取得し、変更内容が
    // 公開されていることを確認します。

    out.printlnBold("Business Rule Group after publish:");
    brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
      "http://BRSamples/com/ibm/websphere/sample/brules",
      QueryOperator.EQUAL, "DiscountRules",
      QueryOperator.EQUAL, 0, 0);

    brg = brgList.get(0);
    out.println("Business Rule Group: " + brg.getName());
    op = brg.getOperation("calculateShippingDiscount");

    // 操作のデフォルト・ビジネス・ルールを取得します。
    defaultRule = op.getDefaultBusinessRule();
    out.print("Default Rule: ");
    out.println(defaultRule.getName());
   }
 } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

例

例 8 の Web ブラウザー出力

例 8 の実行

```
Business Rule Group before publish:
Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscount

Business Rule Group after publish:
Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscountHoliday
```

例 9: ビジネス・ルール・グループの操作に別のルールをスケジュールする

この例では、特定の操作を公開してから 1 時間、ビジネス・ルールがアクティブになるようにスケジュールします。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example9 {
static Formatter out = new Formatter();

static public String executeExample9()
{
 try
 {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループを取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsByTNSAndName(
     "http://BRSamples/com/ibm/websphere
     /sample/brules",
     QueryOperator.EQUAL,
     "DiscountRules",
     QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0)
   {
     out.println("");
     out.printlnBold("Business Rule Group before publish:");
     // リストから最初のビジネス・ルール・グループを取得します。
     // ターゲット名前空間と名前の組み合わせは固有であることから、
     // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
     BusinessRuleGroup brg = brgList.get(0);

     // 新しいビジネス・ルールをスケジュールする
     // ビジネス・ルール・グループの操作を取得します。
     Operation op =
     brg.getOperation("calculateShippingDiscount");

     printOperationSelectionRecord(op);
     // この操作に使用可能なビジネス・ルールのリストを取得します。
     List<BusinessRule> ruleList =
     op.getAvailableTargets();

     // リストの先頭にあるルールを取得します。
     // このルールを使用して操作をスケジュールします。
     BusinessRule rule = ruleList.get(0);

     // スケジュールされたビジネス・ルールのリストを取得します。
     OperationSelectionRecordList opList = op
       .getOperationSelectionRecordList();
```

```
    // ビジネス・ルールの終了日を現在より後の日付で作成します。
    Date future = new Date();
    long futureTime = future.getTime() + 3600000;
```

ルールを新規にスケジュールするときには、ルールと併せて開始日と終了日を指定できます。開始日をヌルに設定すると、ルールは公開と同時にアクティブになります。終了日をヌルに設定すると、ルールの終了日は指定されないことになります。スケジュールのオーバーラップは許可されません。これを確認するには、操作の validate メソッドを呼び出します。

```
    // スケジュールされた新しいビジネス・ルールを
    // 作成します。公開したときに即時にアクティブ
    // になるように現在の日付を指定し、さらに現在
    // より後の日付を指定します。
    newOperationSelectionRecord(new Date(),
      new Date(futureTime), rule);
    // 新規にスケジュールしたビジネス・ルールを、
    // スケジュールされたルールのリストに追加します。
    opList.addOperationSelectionRecord(newRecord);
```

操作を検証して、オーバーラップしていないことを確認します。

```
    // リストを検証してオーバーラップがないことを確認します。
    List<Problem> problems = op.validate();
    if (problems.size() == 0)
    {
      // 元のリストを使用するか、ビジネス・ルール・グループの
      // 新しいリストを作成します。
      List<BusinessRuleGroup> publishList = new
      ArrayList<BusinessRuleGroup>();
      // 変更したビジネス・ルール・グループをリストに追加します。
      publishList.add(brg);
      // 更新されたビジネス・ルール・グループを含むリストを公開します。
      BusinessRuleManager.publish(publishList, true);
      out.println("");

      // ビジネス・ルール・グループを再取得し、変更内容が
      // 公開されていることを確認します。
      out.printlnBold("Business Rule Group after
      publish:");

      brgList =
      BusinessRuleManager.getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere
        /sample/brules",
        QueryOperator.EQUAL,
        "DiscountRules",
        QueryOperator.EQUAL, 0, 0);
        brg = brgList.get(0);

        op =
        brg.getOperation("calculateShippingDiscount");

        printOperationSelectionRecord(op);
      }
      // 検証エラーを処理します。
    }
 } catch (BusinessRuleManagementException e)
 {
 e.printStackTrace();
 out.println(e.getMessage());
 }
 return out.toString();
}
/*
操作の操作選択レコードをプリントするためのメソッド。開始日、
```

```
終了日、およびスケジュールされた期間のルール成果物の名前が
プリントされます。
*/
private static void printOperationSelectionRecord(Operation op)
{
 OperationSelectionRecordList opSelectionRecordList = op
   .getOperationSelectionRecordList();
 Iterator<OperationSelectionRecord> opSelRecordIterator =
 opSelectionRecordList
   .iterator();
 OperationSelectionRecord record = null;
 while (opSelRecordIterator.hasNext())
 {
   out.printlnBold("Scheduled Destination:");
   record = opSelRecordIterator.next();
   out.println("Start Date: " + record.getStartDate()
   + " - End Date: " + record.getEndDate());
   BusinessRule ruleArtifact = record.getBusinessRuleTarget();
   out.println("Rule: " + ruleArtifact.getName());
   }
 }
}
```

例

例 9 の Web ブラウザー出力

例 9 の実行

```
Business Rule Group before publish:
Scheduled Destination:
Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005
Rule: calculateShippingDiscountHoliday

Business Rule Group after publish:
Scheduled Destination:
Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005
Rule: calculateShippingDiscountHoliday
Scheduled Destination:
Start Date: Mon Jan 07 21:08:31 CST 2008 - End Date: Mon Jan 07 22:08:31 CST 2008
Rule: calculateShippingDiscount
```

例 10: ルール・セットのテンプレートのパラメーター値を変更する

この例では、テンプレートで定義されたルール・インスタンスを変更するためにパラメーター値を変更し、変更後のルール・インスタンスを公開します。

```
package com.ibm.websphere.sample.brules.mgmt;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;

public class Example10
{
static Formatter out = new Formatter();
```

```
static public String executeExample10()
{
  try
  {
    out.clear();

    // ターゲット名前空間と名前によってビジネス・ルール・グループを
    // 取得します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere
        /sample/brules",
        QueryOperator.EQUAL,
        "ApprovalValues",
        QueryOperator.EQUAL, 0, 0);
    if (brgList.size() > 0)
    {
      // リストから最初のビジネス・ルール・グループを取得します。
      // ターゲット名前空間と名前の組み合わせは固有であることから、
      // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
      BusinessRuleGroup brg = brgList.get(0);
      // 変更するビジネス・ルールが関連付けられている
      // 操作をビジネス・ルール・グループから取得します。
      // 特定の操作にはビジネス・ルールが関連付けられて
      // います。
      Operation op = brg.getOperation("getApprover");

      // 変更するビジネス・ルールを操作から取得します。
      List<BusinessRule> ruleList =
      op.getBusinessRulesByName(
        "getApprover", QueryOperator.EQUAL, 0,
        0);

      if (ruleList.size() > 0)
      {
        out.println("");
        out.printlnBold("Rule set before publish:");
        // 変更するルールを取得します。ルールは、ターゲット名前空間と
        // 名前によって固有のものになりますが、この例の場合には、
        // 「getApprover」という名前のビジネス・ルールしかありません。
        RuleSet ruleSet = (RuleSet) ruleList.get(0);
        out.print(RuleArtifactUtility.printRuleSet(rule
        Set));
```

ルール・セットに含まれるすべてのルールは、ルール・ブロック内にあります。サポートされるルール・ブロックは 1 つのみで、ルール・ブロックを取得するには getFirstRuleBlock メソッドを使用する必要があります。

```
      // ルール・セットには、ルール・ブロック内に定義された
      // すべてのルールが含まれます。
      RuleBlock ruleBlock =
      ruleSet.getFirstRuleBlock();

      Iterator<RuleSetRule> ruleIterator =
      ruleBlock.iterator();

      // ルール・ブロック内のルールを繰り返し処理し、「LargeOrderApprover」
      // という名前のルール・インスタンスを見つけます。
      while (ruleIterator.hasNext())
      {
        RuleSetRule rule = ruleIterator.next();
```

ルール・テンプレートによってルールが定義されていない場合、そのルールで取得できるのは Web 表示のみです。テンプレートで定義されていないルールは、更新

できません。ルールの名前が不明の場合は、そのルールがテンプレートによって定義されているかどうかを調べることが最善です。

```
// ルールを変更するためには、そのルールがテンプレートによって
// 定義されている必要があります。現行のルールが
// テンプレートに基づいているかどうかをチェックします。
if (rule instanceof
RuleSetTemplateInstanceRule)
{
```

TemplateInstance オブジェクトを使用して、ルールを作成します。

```
  // ルール・テンプレート・インスタンスを取得します。
  RuleSetTemplateInstanceRule
  templateInstance =
  (RuleSetTemplateInstanceRule) rule;

  // 変更するルールと一致するルール・インスタンスを
  // 見つけます。
  if
  (templateInstance.getName().equals(
    "LargeOrderApprover"))
  {
```

テンプレート・インスタンスで変更できるのはパラメーター値のみです。パラメーターを変更するには、ParameterValue を取得し、これを適切な値に設定します。ParameterValue は参照によって渡されるため、更新はルール、ルール・セット、およびビジネス・ルール・グループで直接行われます。

```
    // ルール・インスタンスからパラメーターを取得します。
    ParameterValue parameter =
    templateInstance
      .getParameterValue("par
      am2");

    // パラメーターの値を変更します。
    parameter.setValue("superviso
    r");
    break;
  }
 }
}
// 元のリストを使用するか、ビジネス・ルール・グループの
// 新しいリストを作成します。
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// 変更したビジネス・ルール・グループをリストに追加します。
publishList.add(brg);

// 更新されたビジネス・ルール・グループを含むリストを公開します。
 BusinessRuleManager.publish(publishList, true);

out.println("");
// ビジネス・ルール・グループを再取得し、変更内容が
// 公開されていることを確認します。
out.println("Rule set after publish:");

brgList = BusinessRuleManager
  .getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere/sample/brules",
    QueryOperator.EQUAL, "ApprovalValues",
    QueryOperator.EQUAL, 0, 0);

brg = brgList.get(0);
op = brg.getOperation("getApprover");
```

```
    ruleList = op.getBusinessRulesByName(
      "getApprover", QueryOperator.EQUAL, 0,0);

    ruleSet = (RuleSet) ruleList.get(0);
    out.print(RuleArtifactUtility.printRuleSet(ruleSet));
    }
  }
  } catch (BusinessRuleManagementException e)
  {
    e.printStackTrace();
    out.println(e.getMessage());
  }
  return out.toString();
  }
}
```

例

例 10 の Web ブラウザー出力

例 10 の実行

```
Rule set before publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of manager
User Presentation: If the number of items order is above {0} and
the order is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: manager
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer

Rule set after publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
```

```
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer
```

例 11: 新規ルールをテンプレートからルール・セットに追加する

この例では、新規ルールをテンプレートからルール・セットに追加します。新規ルール・インスタンスを作成する前に、その新規ルール・インスタンスのパラメーターを作成します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example11
{
static Formatter out = new Formatter();

static public String executeExample11()
{
 try
 {
  out.clear();
  // ターゲット名前空間と名前によってビジネス・ルール・グループ
  // を取得します。
  List<BusinessRuleGroup> brgList = BusinessRuleManager
   .getBRGsByTNSAndName(
   "http://BRSamples/com/ibm/websphere
   /sample/brules",
   QueryOperator.EQUAL,
   "ApprovalValues",
   QueryOperator.EQUAL, 0, 0);

  if (brgList.size() > 0)
  {
   // リストから最初のビジネス・ルール・グループを取得します。
   // ターゲット名前空間と名前の組み合わせは固有であることから、
   // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
   BusinessRuleGroup brg = brgList.get(0);
   // 変更するビジネス・ルールが関連付けられている
   // 操作をビジネス・ルール・グループから取得します。
   // 特定の操作にはビジネス・ルールが関連付けられて
   // います。
   Operation op = brg.getOperation("getApprover");

   // 変更するビジネス・ルールを操作から取得します。
   List<BusinessRule> ruleList =
   op.getBusinessRulesByName(
    "getApprover", QueryOperator.EQUAL, 0,0);
```

```
if (ruleList.size() > 0)
{
 out.println("");
 out.printlnBold("Rule set before publish:");
 // 変更するルールを取得します。ルールは、
 // ターゲット名前空間と名前によって固有のものになりますが、
 // この例の場合には、「getApprover」という名前のビジネス・ルール
 // しかありません。
 RuleSet ruleSet = (RuleSet) ruleList.get(0);
 out.print(RuleArtifactUtility.printRuleSet(rule
 Set));
```

新しいルールをルール・セットに追加するには、ルール・セットから適切なテンプレートを見つけ、そのテンプレートからインスタンスを作成する必要があります。テンプレートは、名前を基準に見つけられます。

```
// ルール・テンプレートのリストを取得します。
ListRuleSetRuleTemplate> ruleTemplates =
ruleSet
 .getRuleTemplates();

Iterator<RuleSetRuleTemplate> templateIterator
= ruleTemplates
 .iterator();

while (templateIterator.hasNext())
{
 RuleSetRuleTemplate template =
 templateIterator.next();

 // 新規ルールを作成するために使用するテンプレートを見つけます。
 if
 (template.getName().equals("Template_Larg
 eOrder"))
 {
```

テンプレート・インスタンスを作成するには、パラメーターのリストを作成する必要があります。

```
 // このテンプレート・ルール・インスタンスの
 // パラメーターの
 // リストを作成します。
 List<ParameterValue> paramList =
 new ArrayList<ParameterValue>();

 // テンプレート定義から、特定のパラメーターを取得して
 // 値を設定します。
 Parameter param =
 template.getParameter("param0");
 ParameterValue paramValue = param
  .createParameterValue("
  20");

 // パラメーターをリストに追加します。
 paramList.add(paramValue);

 // 次のパラメーターを取得して値を設定します。
 param = template.getParameter("param1");
 paramValue =
 param.createParameterValue("7500");

 // パラメーターをリストに追加します。
 paramList.add(paramValue);

 // 次のパラメーターを取得して値を設定します。
 param =
```

```
template.getParameter("param2");
 paramValue = param
 .createParameterValue("
 2nd-line manager");

// パラメーターをリストに追加します。
paramList.add(paramValue);
```

パラメーターの作成が完了したら、テンプレート・インスタンスを作成します。

```
// パラメーター・リストを使用して
// テンプレート・ルール・インスタンスを
// 作成します。
RuleSetTemplateInstanceRule
 templateInstance = template
 .createRuleFromTemplate
 ("ExtraLargeOrder",
 paramList);
// ルール・セットのルール・ブロックを取得します。
RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();
```

テンプレート・インスタンスの作成が完了したら、それをルール・ブロックに追加します。ルール・ブロックに追加したテンプレート・インスタンスは、他のテンプレート・ルール・インスタンスとの間で順序を設定できます。

```
// テンプレート・ルールを
// ルール・ブロックに追加します。
ruleBlock.addRule(templateInstance)
;

 break;
 }
}

// 元のリストを使用するか、ビジネス・ルール・グループの
// 新しいリストを作成します。
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// 変更したビジネス・ルール・グループをリストに追加します。
publishList.add(brg);

// 更新されたビジネス・ルール・グループを含むリストを公開します。
BusinessRuleManager.publish(publishList, true);

out.println("");

// ビジネス・ルール・グループを再取得し、変更内容が
// 公開されていることを確認します。
out.println("Rule set after publish:");

brgList = BusinessRuleManager
.getBRGsByTNSAndName(
  "http://BRSamples/com/ibm/websphere
  /sample/brules",
  QueryOperator.EQUAL,
  "ApprovalValues",
  QueryOperator.EQUAL, 0, 0);

brg = brgList.get(0);
op = brg.getOperation("getApprover");
ruleList = op.getBusinessRulesByName(
     "getApprover", QueryOperator.EQUAL,
     0, 0);
```

```
    ruleSet = (RuleSet) ruleList.get(0);
    out.print(RuleArtifactUtility.printRuleSet(rule
    Set));
    }
  }
 } catch (BusinessRuleManagementException e)
 {
  e.printStackTrace();
  out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

例

例 11 の Web ブラウザー出力

例 11 の実行

```
Rule set before publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer

Rule set after publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
```

```
Parameter Name: param0
Parameter Value: peer
Rule: ExtraLargeOrder
Display Name:
Description: null
Expanded User Presentation: If the number of items order is above 20 and
the order is above $7500, then it requires the approval of 2nd-line manager
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 20
Parameter Name: param1
Parameter Value: 7500
Parameter Name: param2
Parameter Value: 2nd-line manager
```

例 12: パラメーター値の変更によってデシジョン・テーブルのテンプレートを変更して公開する

この例では、条件とアクションの両方がテンプレートで定義されているデシジョン・テーブルについて、パラメーター値を変更することでその条件とアクションを変更した後、これを公開します。

デシジョン・テーブルの条件とアクションを変更する最も簡単な方法は、各条件レベルのテンプレートおよび各アクションに割り当てられている固有の名前を使用することです。固有の名前を検索してから、そのテンプレートで定義されたテンプレート・インスタンスを変更できます。特定のテンプレートのテンプレート・インスタンスを変更すると、そのテンプレートで定義された当該レベルの条件値がすべて更新されます。アクション式の各インスタンスは固有なので、特定のインスタンスを変更しても、他のインスタンスが変更されることはありません。

この例では、更新対象の特定のケース・エッジ、特定のパラメーター値、および特定のテンプレートで定義されたアクション式を見つけやすくするために、多数の追加メソッドが作成されています。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example12 {
```

```
static Formatter out = new Formatter();

static public String executeExample12()
{
 try
 {
   out.clear();
   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   // を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere
    /sample/brules",
    QueryOperator.EQUAL,
    "ConfigurationValues",
    QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0)
   {
    // リストから最初のビジネス・ルール・グループを取得します。
    // ターゲット名前空間と名前の組み合わせは固有であることから、
    // これがリスト内の唯一のビジネス・ルール・グループと
    // なるはずです。
    BusinessRuleGroup brg = brgList.get(0);

    // 変更するビジネス・ルールが関連付けられている
    // 操作をビジネス・ルール・グループから取得します。
    // 特定の操作にはビジネス・ルールが関連付けられて
    // います。
    Operation op = brg.getOperation("getMessages");

    // この操作に使用可能なすべてのビジネス・ルールを取得します。
    List<BusinessRule> ruleList =
    op.getAvailableTargets();

    // この操作には、1 つのビジネス・ルールしかありません。これが、
    // 更新対象のビジネス・ルールです。
    DecisionTable decisionTable = (DecisionTable)
    ruleList.get(0);
    out.println("");
    out.printlnBold("Decision table before publish:");
    out
     .print(RuleArtifactUtility
     .printDecisionTable(decisionT
      able));
```

初期ルールと条件、およびアクションは、ツリー・ブロックに含まれます。このツリー・ブロックから、ルート・ノードを取得できます。

```
    // デシジョン・テーブルのすべての条件とアクション
    // が含まれるツリー・ブロックを取得します。
    TreeBlock treeBlock = decisionTable.getTreeBlock();
    // ツリー・ブロックから、デシジョン・テーブルの
    // ナビゲーションの開始点となるツリー・ノードを取得します。
    TreeNode treeNode = treeBlock.getRootNode();
```

更新する条件は、「Condition Value Template 2.1」という名前のテンプレートで定義されています。メソッド `getCaseEdge` は、TreeNode から該当するケース・エッジまで再帰的に検索し、そのテンプレートが定義されたケース・エッジを見つけます。このメソッドでは、テンプレートが定義されているレベルと現行のレベルが既知であることを想定しています。複数のケース・エッジに同じ名前が使用されている場合、このメソッドを使用して、テンプレートを持つケース・エッジを特定の名前を基準に検索することができます。

```
// ルートよりも 1 つ下のレベルにある、パラメーターの
// 値が特定の名前に設定された特定のテンプレートを持つ
// ケース・エッジを検索します。先頭から開始するため、
// 現行の深さは 0 となります。
CaseEdge caseEdge = getCaseEdge(treeNode, "param0",
"Condition Value Template 2.1", 1, 0);
```

ケース・エッジが検出されたら、条件の ConditionValueTemplateInstance を取得できます。

```
if (caseEdge != null)
{
 // ケース・エッジが検出されました。ケース・エッジの値の
 // 定義を取得します。
 TreeConditionValueDefinition condition =
  caseEdge
  .getValueDefinition();
 // テンプレートで定義された条件式を取得します。
  TemplateInstanceExpression conditionExpression
  = condition
   .getConditionValueTemplateInstance(
 );
```

ConditionValueTemplateInstance において、getParameterValue メソッドを使用して該当するパラメーター値を取得し、その値を更新できます。

```
 // 式のテンプレートを取得します。
 Template conditionTemplate =
 conditionExpression
  .getTemplate();

  // テンプレートが正しいことを確認します。これは、1 つの条件値
  // に対して複数の
  // テンプレートが存在する場合があるためですが、実際に適用され
  // ているテンプレートは
  // 1 つのみです。
  if (conditionTemplate.getName().equals(
   "Condition Value Template 2.1"))
  {
   // パラメーター値を取得します。
   ParameterValue parameterValue =
   getParameterValue("param0",
    conditionExpression);

   // 新しいパラメーター値を設定します。
   parameterValue.setValue("info");
  }
```

テンプレートで定義されていて、更新する必要がある異なるアクション式を取得できます。 getActionExpressions メソッドは、Action Value Template 1 という名前のテンプレートで定義されたすべてのアクションを返します。

```
 ConditionNode conditionNode = (ConditionNode)
 treeNode;

 // ケース・エッジのツリー・ノードを取得します。
 ListCaseEdge> caseEdges =
 conditionNode.getCaseEdges();

 // 同じく更新する必要があるアクション式のすべてを
 // 保持する
 // リストを作成します。テンプレートは共用されていても、
 // すべての
 // アクションは他のアクションとは独立しているため、すべての
 // アクションを更新する必要があります。
 List<TemplateInstanceExpression> expressions =
```

```
    new Vector<TemplateInstanceExpression>();

    // すべての式を取得します。
    for (CaseEdge edge : caseEdges)
    {
      getActionExpressions("Action Value
      Template 1", edge,
      expressions);
    }
```

アクション式のリストを使用して、それぞれの項目を更新できます。テンプレートで定義されたアクション式の場合、正しいパラメーター値を更新できます。

```
    // それぞれの式で正しいパラメーター値を更新します。
    for (TemplateInstanceExpression expression
    expressions)
    {
     for (ParameterValue parameterValue :
     expression
      .getParameterValues())
     {
      // 正しいパラメーターを確認します。ただし、
      // このテンプレートにあるパラメーターは 1 つだけです。
      if
       (parameterValue.getParameter().getN
       ame().equals("param0")) {
        String value =
        parameterValue.getValue();
        parameterValue.setValue("Info
        "
        +
        value.substring(value.
        indexOf(":"),
        value.length()));
     }
   }
}
// 条件値とアクションを更新した後、ビジネス・
// ルール・グループを公開できます。
// 元のリストを使用するか、ビジネス・ルール・グループの
// 新しいリストを作成します。
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();

// 変更したビジネス・ルール・グループをリストに
// 追加します。
publishList.add(brg);

// 更新されたビジネス・ルール・グループを含むリストを
// 公開します。
BusinessRuleManager.publish(publishList, true);

out.println("");

// ビジネス・ルール・グループを再取得し、
// 変更内容が
// 公開されていることを確認します。
out.printlnBold("Decision table after
publish:");

brgList =
BusinessRuleManager.getBRGsByTNSAndName(
 "http://BRSamples/com/ibm/websphere
 /sample/brules",
 QueryOperator.EQUAL,
 "ConfigurationValues",
 QueryOperator.EQUAL, 0, 0);
```

```
   brg = brgList.get(0);
   op = brg.getOperation("getMessages");
   ruleList = op.getAvailableTargets();

   decisionTable = (DecisionTable)
   ruleList.get(0);
   out.print(RuleArtifactUtility
    .printDecisionTable(decisionTable))
    ;
    }
   }
 } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
}

/*
デシジョン・テーブルを再帰的にナビゲートして、特定の名前のテンプレート
を持ち、
変更対象となる特定のパラメーターを含むエッジ・ケースを見つけるための
メソッド。このメソッドでは、変更する値が存在しているデシジョン・テーブル
内のレベル (depth) が既知であり、現行のレベル (currentDepth) が追跡され
ていることを想定しています。*
*/
static private CaseEdge getCaseEdge(TreeNode node, String pName,
  String templateName, int depth, int currentDepth)
{
 // 現行のノードがアクションであるかどうかを確認します。これは、
 // ケース・エッジの検索中にデシジョン・テーブルのこの分岐が
 // 使い果たされたことを示します。
 if (node instanceof ActionNode)
 {
  return null;
 }

 // このノードのケース・エッジを取得します。
 List<CaseEdge> caseEdges = ((ConditionNode) node).getCaseEdges();
 for (CaseEdge caseEdge : caseEdges)
 {

 // 正しいレベルに到達したかどうかを確認します。
 if (currentDepth < depth)
  {
   // 1 つ下のレベルに移動し、もう一度 getCaseEdge を呼び出して、
   // そのレベルを処理します。
   currentDepth++;
   return getCaseEdge(caseEdge.getChildNode(), pName,
    templateName, depth, currentDepth);
  } else
  {
   // 正しいレベルに到達しています。条件を取得し、その条件の
   // テンプレートが、探していたテンプレートと一致するか
   // どうかを確認します。
   TreeConditionValueDefinition condition = caseEdge
    .getValueDefinition();

   // テンプレートで定義された条件の式を
   // 取得します。
   TemplateInstanceExpression expression = condition
    .getConditionValueTemplateInstance();
   // 式からテンプレートを取得します。
      Template template = expression.getTemplate();
```

```
   // これが探していたテンプレートかどうかを確認します。
    if (template.getName().equals(templateName))
    {
     // テンプレートが一致することが分かりました。
      return caseEdge;
    } else
     caseEdge = null;
   }
 }
 return null;
}

/*
このメソッドは、式の異なるパラメーター値をチェックし、
正しいパラメーターが見つかると、そのパラメーターの値を返します。
*/
private static ParameterValue getParameterValue(String pName,
  TemplateInstanceExpression expression)
{
 // 式がヌルでないことを確認します。ヌルは、渡された
 // 式がテンプレートでは定義されておらず、チェック
 // 対象のパラメーターがない可能性を意味します。
 if (expression != null) {
  // 式のパラメーター値を取得します。
  List<ParameterValue> parameterValues = expression
    .getParameterValues();

  for (ParameterValue parameterValue : parameterValues)
  {
   // それぞれのパラメーターについて、探していたパラメーター
   // と一致するかどうかを確認します。

   if
    (parameterValue.getParameter().getName().equals(pName
   ))
   {
    // 一致するパラメーター値を返します。
      return parameterValue;
   }
  }
 }
return null;
}
/*
このメソッドは、特定のテンプレートで定義された
すべてのアクション式を検索します。ケース・エッジを
再帰的に検索し、式パラメーターに一致するアクション式
を追加します。
*/

private static void getActionExpressions(String templateName,
CaseEdge next, List<TemplateInstanceExpression>
expressions)
{
 ActionNode actionNode = null;
 TreeNode treeNode = next.getChildNode();

 // 現行のノードがアクション・ノード・レベルにあるかどうかを確認します。
 if (treeNode instanceof ConditionNode)
 {
  List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
   .getCaseEdges();

  Iterator<CaseEdge> caseEdgesIterator =
  caseEdges.iterator();

  // すべてのケース・エッジを順に処理して、アクション式を
```

```
  // 検索します。
  while (caseEdgesIterator.hasNext())
   {
    getActionExpressions(templateName,
    caseEdgesIterator.next(),
     expressions);
   }
 } else {
   // ActionNode を検出
   actionNode = (ActionNode) treeNode;

   List<TreeAction> treeActions = actionNode.getTreeActions();
   // 式に少なくとも 1 つの treeAction が指定されている
   // ことを確認し、式を順に処理して、式が
   // 特定のテンプレートで作成されているかどうか
   // を確認します。
   if (!treeActions.isEmpty())
   {

    Iterator<TreeAction> iterator =
    treeActions.iterator();

   while (iterator.hasNext())
    {
     TreeAction treeAction = iterator.next();
     TemplateInstanceExpression expression =
     treeAction
      .getValueTemplateInstance();

     Template template = expression.getTemplate();

     if (template.getName().equals(templateName))
     {
      // テンプレートが一致する式を検出
      expressions.add(expression);
     }
    }
   }
  }
 }
}
```

例

例 12 の Web ブラウザー出力

例 12 の実行

```
Rule set before publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Decision table after publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
```

例 13: 条件値とアクションをデシジョン・テーブルに追加する

この例では、条件値とアクションをデシジョン・テーブルに追加します。条件値は、テンプレートを使用してデシジョン・テーブルに追加できます。

条件値を条件ノードに追加すると、ケース・エッジを追加することになります。新しいケース・エッジは、ケース・エッジのリストの最後に追加されます。条件値を

追加するためには、適切なパラメーター値を設定したテンプレート・インスタンス式を指定する必要があります。テンプレート・インスタンス式を指定するためには、特定のテンプレートを使用する必要があります。テンプレートの名前は、条件ノードのレベルごとに固有の名前を付けることをお勧めします。これにより、該当するタイプの条件に合った正しいテンプレートを取得できます。単一のテンプレート定義を使用すると、条件を追加するレベルの判別が困難になるおそれがあります。

条件値を条件ノードに設定すると、同じテンプレート・インスタンスを持つ条件値を、同じレベルにあるすべての条件ノードに追加することになります。これは、デシジョン・テーブルが平衡型であるためです。また、新しい条件値を追加すると、この操作の一環として新しいアクション・ノードも追加されます。これらのアクション・ノードは、ユーザー表示およびテンプレート・インスタンス式にヌルの値が指定されたツリー・アクションを持ちます。条件値は、アクション・ノードを子ノードとして持たない条件ノードに追加できるため、条件ノードの追加によって、アクション・ノードが大幅に増える場合があります。アクション・ノードの数は、条件ノードを追加するレベル、そのレベルにある条件ノードの数、および子レベルそれぞれにある条件ノードの数に応じます。

作成されたアクション・ノードを見つけるには、ユーザー表示およびテンプレート・インスタンス式がヌルのツリー・アクションを持つアクション・ノードの検索を実行します。 `TreeAction` に設定可能な式を作成するには、`TreeActionValueTemplate` を使用します。このパターンを、すべての新しいアクション・ノードに対して繰り返す必要があります。

この例には、新規ツリー・アクションのセットアップを支援する 2 つのメソッドが提供されています。 `getEmptyActionNode` は、現行の条件ノードから空のアクション・ノードを再帰的に検索します。getParameterValue は、名前で指定されたパラメーターの値を返します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
```

```
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example13
{
 static Formatter out = new Formatter();

 static public String executeExample13()
 {
  try
  {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   // を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
   "http://BRSamples/com/ibm/websphere/sample/brules",
     QueryOperator.EQUAL,"ConfigurationValues",
     QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0)
    {
     // リストから最初のビジネス・ルール・グループを
     // 取得します。ターゲット名前空間と名前の
     // 組み合わせは固有であることから、これがリスト内
     // の唯一のビジネス・ルール・グループとなるはずです。
     BusinessRuleGroup brg = brgList.get(0);

     // 変更するビジネス・ルールが関連付けられている
     // 操作をビジネス・ルール・グループから取得します。
     // 特定の操作にはビジネス・ルールが関連付けられて
     // います。
     Operation op = brg.getOperation("getMessages");

     // この操作に使用可能なすべての
     // ビジネス・ルールを取得します。
     List<BusinessRule> ruleList =
      op.getAvailableTargets();

     // この操作には、1 つのビジネス・ルール
     // しかありません。これが、更新対象の
     // ビジネス・ルールです。

     DecisionTable decisionTable = (DecisionTable)
      ruleList.get(0);
      out.println1nBold("Decision table before
      publish:");
      out.print(RuleArtifactUtility
      .printDecisionTable(decisionTable));
```

条件値を追加するレベルを見つける必要があります。クラスを使用するユーザー・インターフェースまたはアプリケーションは、どこに条件を追加するかを認識しているため、このレベルは通常、パラメーターとして渡されます。

```
     // デシジョン・テーブルのすべての条件と
     // アクションが含まれるツリー・ブロックを
     // 取得します。
     TreeBlock treeBlock =
      decisionTable.getTreeBlock();

     // ツリー・ブロックから、デシジョン・テーブルの
     // ナビゲーションの開始点となるツリー・ノードを
     // 取得します。
     ConditionNode conditionNode = (ConditionNode)
      treeBlock.getRootNode();
```

```
// 条件の第 1 レベルであるこのノードの
// ケース・エッジを取得します。
List<CaseEdge> caseEdges =
 conditionNode.getCaseEdges();

// 新しい条件の追加先となるケース・エッジを
// 取得します。
CaseEdge caseEdge = caseEdges.get(0);

// 条件のテンプレートを取得するために、
// ケース・エッジに対応する条件ノードを
// 取得します。
conditionNode = (ConditionNode)
    caseEdge.getChildNode();

// 条件のテンプレートを取得します。
List<TreeConditionValueTemplate>
 treeValueConditionTemplates = conditionNode
 .getAvailableValueTemplates();

Iterator<TreeConditionValueTemplate>
 treeValueConditionTemplateIterator =
 treeValueConditionTemplates.iterator();

TreeConditionValueTemplate conditionTemplate =
 null;
```

デシジョン・テーブルの各条件ノード・レベルで固有のテンプレート名を使用することで、より簡単に正しい条件ノード値に条件値を追加できます。

```
// 使用すべきテンプレートを検索します。
 while
(treeValueConditionTemplateIterator.hasNext())
{
 conditionTemplate =
  treeValueConditionTemplateIterator
   .next();
 if (conditionTemplate.getName().equals(
  "Condition Value Template
  2.1"))
  {
   // テンプレートを検出
   break;
  }
  conditionTemplate = null;
}
if (conditionTemplate != null)
{
```

正しいテンプレートが検出された後、インスタンスを作成して適切なパラメーター値を設定してから、条件ノードに追加します。

```
// テンプレートからパラメーター定義を
// 取得します。
Parameter conditionParameter =
conditionTemplate.getParameter("param0");

// 新しい条件テンプレート・インスタンスで
// 使用するパラメーター値インスタンスを
// 作成します。
ParameterValue conditionParameterValue =
  conditionParameter
  .createParameterValue("fatal");

List<ParameterValue>
  conditionParameterValues = new
```

```
  ArrayList<ParameterValue>();

// パラメーター値をリストに追加します。

conditionParameterValues
 .add(conditionParameterValue);

// パラメーター値を使用して、新しい条件
// テンプレート・インスタンスを作成します。
TemplateInstanceExpression
 newConditionValue =
 conditionTemplate
  .createTemplateInstanceExpression(c
 onditionParameterValues);
// 条件テンプレート・インスタンスをこの条件
// ノードに追加します。
conditionNode

.addConditionValueToThisLevel(newConditionValue);
// 条件ノードが追加されると、新しい
// 空のアクション・ノードが作成され
// ます。ここにはアクション・テンプレート・
// インスタンスを設定する必要があります。親
// レベルから、それぞれの空のアクション・ノード
// を検索すると、新しい空のアクション・
// ノードをすべて見つけることができます。
conditionNode = (ConditionNode)
conditionNode.getParentNode();
```

条件値を条件ノードに追加した後は、TreeActionValueTemplate を使用して、新規アクション・ノードのツリー・アクションを設定する必要があります。まず、ケース・エッジの空のアクション・ノードを見つけます。親条件ノードを使用することで、条件ノードの繰り返し処理によってすべてのアクション・ノードが確実に検索されるようにします。

```
// 親ノードのケース・エッジを取得します。
caseEdges = conditionNode.getCaseEdges();

Iterator<CaseEdge> caseEdgesIterator =
caseEdges.iterator();

while (caseEdgesIterator.hasNext())
{
 // 各ケース・エッジについて、空のアクション・
 // ノードがある場合にはそれを取得します。
 ActionNode actionNode =
 getEmptyActionNode(caseEdgesIterator
 .next());

 // すべてのアクションが設定されていることを確認します。
 if (actionNode != null)
 {
```

ツリー・アクションが空のアクション・ノードが検出された場合には、TreeActionValueTemplate を使用してツリー・アクションを設定する必要があります。まずテンプレートを見つけて、パラメーターを指定してからテンプレート・インスタンスを作成します。テンプレート・インスタンスが作成されたら、ツリー・アクションを更新できます。この例では、同じ条件ノードの下にある別のアクション・ノードの別のツリー・アクションの値を使用して、パラメーターを設定しています。新規パラメーター値の作成に使用できる値が別のツリー・アクションに存在しない別のデシジョン・テーブルの場合は、アプリケーションによって値をパラメーターとして渡す必要があります。

```
// ツリー・アクションのリストを取得します。これらの
// アクションは、実際の
// アクションではなく、
// アクションの
// プレースホルダーです。
List<TreeAction>
treeActionList = actionNode
 .getTreeActions();

List<TreeActionTermDefinition>
treeActionTermDefinitions =
 treeBlock
.getTreeActionTermDefinitions();

List<TreeActionValueTemplate>
 treeActionValueTemplates =
 treeActionTermDefinitions
 .get(0).getValueTemplates();

TreeActionValueTemplate
 actionTemplate = null;

for (TreeActionValueTemplate
  tempActionTemplate :
  treeActionValueTemplates)
  {

  if
  (tempActionTemplate.get
   Name().equals(
   "Action Value
   Template 1"))
   {
    actionTemplate =
    tempActionTemplate;
    break;
   }
  }

  if (actionTemplate != null)
  {
   // 親条件ノードの下に
   // ある別のアクションを
   // 取得します。
   // この値は、新しい
   // アクション・ノード
   // のエラー・メッセージ
   // を作成するときに、
   // ベースとして使用
   // します。まず初めに、
   // 親条件ノードまで
   // 移動します。
   ConditionNode
   parentNode =
   (ConditionNode)
   actionNode
    .getParentNode();

   // 親ノードの最初の
   // ケース・エッジを
   // 取得します。新規
   // アクションはケース・
   // エッジ・リストの
   // 最後に追加される
   // ため、最初のケース・
   // エッジのアクション
   // は常に設定済みです。
```

```
CaseEdge caseE =
 parentNode.getCas
 eEdges().get(
 0);

// この子ノードは
// アクション・ノードで、
// 新規アクション・
// ノードと同じレベルに
// あります。
ActionNode aNode =
(ActionNode) caseE
 .getChildNode();

// ツリー・アクションのリストを
// 取得します。
TreeAction
 existingTreeAction =
 aNode
 .getTreeActions()
 .get(0);

// パラメーターを
// 取得することが
// 可能なツリー・
// アクションの
// テンプレート・
// インスタンス式を
// 取得します。

TemplateInstanceExpression
existingExpression =
 existingTreeAction
 .getValueTemplateInstance();

ParameterValue
existingParameterValue =
getParameterValue(
  "param0",
existingExpression);

String actionValue =
existingParameterValue
 .getValue();

// 既存のツリー・
// アクションの
// メッセージから
// 新規メッセージを
// 作成します。
actionValue = "Fatal"
  +
actionValue.substring(actionValue
 .indexOf(":"), actionValue
 .length());
 Parameter
 actionParameter =
 actionTemplate
 .getParameter("param0");

 // テンプレートから
 // パラメーターを取得します。
 ParameterValue
 actionParameterValue =
  actionParameter
 .createParameterValue(actionValue);
```

```
            // パラメーターをテンプレートの
            // リストに追加します。
            List<ParameterValue>
            actionParameterValues = new
            ArrayList<ParameterValue>();

          actionParameterValues.add(actionParameterValue);

            // 新規ツリー・アクション・
            // インスタンスを作成します。

           TemplateInstanceExpression
            treeAction = actionTemplate
     .createTemplateInstanceExpression(actionParameterValues);

           // ツリー・アクションをツリー・
           // アクション・リストに設定すること
           // によって、アクション・ノードに
           // 設定します。
```

ここで、アクション・ノード内のツリー・アクションが更新されます。

```
           treeActionList.get(0)
           .setValueTemplateInstance(
           treeAction);
           }
          }
        }
    }
    // 条件値とアクションを
    // 更新した後、ビジネス・ルール・
    // グループを公開できます。
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
     List<BusinessRuleGroup> publishList = new
      ArrayList<BusinessRuleGroup>();

    // 変更したビジネス・ルール・グループを
    // リストに追加します。
     publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含む
    // リストを公開します。

    BusinessRuleManager.publish(publishList, true);

    brgList =
     BusinessRuleManager.getBRGsByTNSAndName(
     "http://BRSamples/com/ibm/websphere/sample/brules",
     QueryOperator.EQUAL, "ConfigurationValues",
     QueryOperator.EQUAL, 0, 0);
     brg = brgList.get(0);
     op = brg.getOperation("getMessages");
     ruleList = op.getAvailableTargets();
     decisionTable = (DecisionTable)
      ruleList.get(0);
     out.println1nBold("Decision table after
      publish:");
     out
      .print(RuleArtifactUtility
       .printDecisionTable(decisionTable));
    }
  } catch (ValidationException e)
  {
   List<Problem> problems = e.getProblems();

   out.println("Problem = " +
```

```
  problems.get(0).getErrorType().name());

  e.printStackTrace();
  out.println(e.getMessage());
 } catch (BusinessRuleManagementException e)
 {
  e.printStackTrace();
  out.println(e.getMessage());
 }
 return out.toString();
}

/*
* このメソッドは、現行のケース・エッジから、空のツリー・
* アクションを持つすべてのアクション・ノードを検索します。
* 空のアクション・ノードを検出するには、ケース・エッジ・
* リストの最後を調べ、アクション・ノードにユーザー表示と
* TemplateInstanceExpression の両方がヌルのツリー・アクション
* があるかどうかを確認します。
*/
private static ActionNode getEmptyActionNode(CaseEdge next)
{
 ActionNode actionNode = null;
 TreeNode treeNode = next.getChildNode();

if (treeNode instanceof ConditionNode)
{
 List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
  .getCaseEdges();

 if (caseEdges.size() > 1)
 {
  // 右端のケース・エッジを新規条件として取得
  // します。その結果、空のアクションはケース・
  // エッジの右端に位置することになります。
  actionNode = getEmptyActionNode(caseEdges
   .get(caseEdges.size() - 1));

  if (actionNode != null)
  {
   return actionNode;
  }
}
} else
{
 actionNode = (ActionNode) treeNode;

 List<TreeAction> treeActions =
 actionNode.getTreeActions();

 if (!treeActions.isEmpty())
 {
 if
 ((treeActions.get(0).getValueUserPresentation() == null)
  &&
 (treeActions.get(0).getValueTemplateInstance() == null))
  {
   return actionNode;
  }
 }
  `actionNode = null;
 }
 return actionNode;
}
/*
* このメソッドは、式の異なるパラメーター値をチェックし、
* 正しいパラメーターが見つかると、そのパラメーターの値を
```

```
 * 返します。
 */
 private static ParameterValue getParameterValue(String pName,
   TemplateInstanceExpression expression)
 {
  ParameterValue parameterValue = null;

  // 式がヌルでないことを確認します。
  // ヌルは、渡された式がテンプレート
  // では定義されておらず、チェック対象の
  // パラメーターがない可能性を意味します。
  if (expression != null)
  {
   // 式のパラメーター値を取得します。
   List<ParameterValue> parameterValues = expression
    .getParameterValues();
   Iterator<ParameterValue> parameterIterator =
    parameterValues
    .iterator();

   // それぞれのパラメーターについて、探していた
   // パラメーターと一致するかどうかを確認します。
   while (parameterIterator.hasNext())
   {
    parameterValue = parameterIterator.next();

    if
    (parameterValue.getParameter().getName().equals(pName))
    {
    // 一致するパラメーター値を
    // 返します。
    return parameterValue;
    }
  }
 }
 return parameterValue;
}
}
```

例

例 13 の Web ブラウザー出力

例 13 の実行

```
Decision table before publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Decision table after publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
```

例 14: ルール・セットでのエラーを処理する

この例では、適切なメッセージを表示したり、問題を修正するためのアクションを実行できるように、ルール・セットの問題をキャッチし、発生した問題を突き止める方法に重点を置きます。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```
import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import
com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.problem.ValidationError;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example14 {
static Formatter out = new Formatter();

static public String executeExample14() {
 try {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   // を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsByTNSAndName(
     "http://BRSamples/com/ibm/websphere
     /sample/brules",
     QueryOperator.EQUAL,
     "ApprovalValues",
     QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0) {
     // リストから最初のビジネス・ルール・グループを取得します。
     // ターゲット名前空間と名前の組み合わせは固有であることから、
     // これがリスト内の唯一のビジネス・ルール・グループと
     // なるはずです。
     BusinessRuleGroup brg = brgList.get(0);
     out.println("Business Rule Group retrieved");

     // 変更するビジネス・ルールが関連付けられている
     // 操作をビジネス・ルール・グループから取得します。
     // 特定の操作にはビジネス・ルールが関連付けられて
     // います。
     Operation op = brg.getOperation("getApprover");

     // 特定のルールを名前を基準に取得します。
     List<BusinessRule> ruleList =
     op.getBusinessRulesByName(
       "getApprover", QueryOperator.EQUAL, 0,
       0);

     // 特定のルールを取得します。
     RuleSet ruleSet = (RuleSet) ruleList.get(0);
     out.println("Rule Set retrieved");

     RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

     Iterator<RuleSetRule> ruleIterator =
     ruleBlock.iterator();

     // ルールを検索して、変更するルールを見つけます。
     while (ruleIterator.hasNext()) {
```

```
      RuleSetRule rule = ruleIterator.next();

      // ルールがテンプレートで定義されていて、変更可能で
      // あることを確認します。
      if (rule instanceof
      RuleSetTemplateInstanceRule) {
       // テンプレート・ルール・インスタンスを取得します。
       RuleSetTemplateInstanceRule
       templateInstance =
       (RuleSetTemplateInstanceRule) rule;
       // 正しいテンプレート・ルール・インスタンスであることを
       // 確認します。
       if (templateInstance.getName().equals(
         "LargeOrderApprover")) {
```

問題を発生させるため、この例では、パラメーターを式とは互換しない値に設定します。パラメーターは整数が想定されていますが、ストリングを渡します。

```
         // テンプレート・インスタンスからパラメーターを取得します。
         ParameterValue parameter =
         templateInstance
           .getParameterValue("par
           am1");

         // このパラメーターに誤った値を設定します。
         // これにより、検証エラーが発生することになります。
         parameter.setValue("$3500");
         out.println("Incorrect parameter
         value set");
         break;
       }
     }
   }
   // 上記で発生したエラーにより、この
   // コードには到達できなく
   // なります。

   // 条件値とアクションを更新した後、
   // ビジネス・ルール・グループを
   // 公開できます。
   // 元のリストを使用するか、ビジネス・ルール・グループの
   // 新しいリストを作成します。
   List<BusinessRuleGroup> publishList = new
   ArrayList<BusinessRuleGroup>();

   // 変更したビジネス・ルール・グループをリストに追加します。
   publishList.add(brg);

   // 更新されたビジネス・ルール・グループを含むリストを公開します。
   BusinessRuleManager.publish(publishList, true);
 }
```

ValidationException をキャッチし、この例外から、問題を取得することができます。それぞれの問題について、エラーを確認することで、発生したエラーを判別できます。メッセージをプリントするか、または適切なアクションを実行することができます。

```
 } catch (ValidationException e) {
 out.println("Validation Error");

 List<Problem> problems = e.getProblems();

 Iterator<Problem> problemIterator = problems.iterator();

 // 該当するエラーの問題リストを確認し、適切な
 // アクション (例えば、エラーのレポート、または
```

```
                // エラーの修正) を実行します。
  while (problemIterator.hasNext()) {
    Problem problem = problemIterator.next();
    ValidationError error = problem.getErrorType();

    // 特定のエラーの値を確認します。
    if (error == ValidationError.TYPE_CONVERSION_ERROR) {
      // 問題をレポートすることによって、このエラーを処理します。
      out
        .println("Problem: Incorrect value
        entered for a parameter");
      return out.toString();
    }
    // または....
    // 他のエラーを確認して、適切なエラー・
    // メッセージを表示するか、アクションを実行して
    // 問題を修正することもできます。
  }
} catch (BusinessRuleManagementException e) {
  out.println("Error occurred.");
  e.printStackTrace();
}
return out.toString();
}
}
```

例

例 14 の Web ブラウザー出力

例 14 の実行

```
Business Rule Group retrieved
Rule Set retrieved
Validation Error
Problem: Incorrect value entered for a parameter
```

例 15: ビジネス・ルール・グループでのエラーを処理する

この例は、例 14 と同様で、ビジネス・ルール・グループを公開するときに発生した問題を処理する方法を説明します。この例では、問題を判別し、正しいメッセージをプリントするか、またはアクションを実行する方法を説明します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import
com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
```

```
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example15
{
static Formatter out = new Formatter();

static public String executeExample15()
{
 try
 {
  out.clear();

 // ターゲット名前空間と名前によってビジネス・ルール・グループ
 // を取得します。
 List<BusinessRuleGroup> brgList = BusinessRuleManager
  .getBRGsByTNSAndName(
  "http://BRSamples/com/ibm/websphere
  /sample/brules",
  QueryOperator.EQUAL,
  "ApprovalValues",
  QueryOperator.EQUAL, 0, 0);
  if (brgList.size() > 0)
  {
   // リストから最初のビジネス・ルール・グループを取得します。
   // ターゲット名前空間と名前の組み合わせは固有であることから、
   // これがリスト内の唯一のビジネス・ルール・グループと
   // なるはずです。
   BusinessRuleGroup brg = brgList.get(0);
   out.println("Business Rule Group retrieved");

   // 変更するビジネス・ルールが関連付けられている
   // 操作をビジネス・ルール・グループから取得します。
   // 特定の操作にはビジネス・ルールが関連付けられて
   // います。
   Operation op = brg.getOperation("getApprover");

   // 特定のルールを名前を基準に取得します。
   List<BusinessRule> ruleList =
   op.getBusinessRulesByName(
    "getApprover", QueryOperator.EQUAL, 0,
    0);

   // 特定のルールを取得します。
   RuleSet ruleSet = (RuleSet) ruleList.get(0);
    out.println("Rule Set retrieved");

   RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

   Iterator<RuleSetRule> ruleIterator =
    ruleBlock.iterator();

   // ルールを検索して、変更するルールを見つけます。
   while (ruleIterator.hasNext())
   {
    RuleSetRule rule = ruleIterator.next();

    // ルールがテンプレートで定義されていて、変更可能で
    // あることを確認します。
    if (rule instanceof
    RuleSetTemplateInstanceRule)
    {
     // テンプレート・ルール・インスタンスを取得します。
     RuleSetTemplateInstanceRule
     templateInstance =
     (RuleSetTemplateInstanceRule) rule;
```

```
    // 正しいテンプレート・ルール・インスタンスであることを
    // 確認します。
    if (templateInstance.getName().equals(
      "LargeOrderApprover"))
    {
     // テンプレート・インスタンスからパラメーターを
     // 取得します。
     ParameterValue parameter =
     templateInstance
      .getParameterValue("par
      am1");

     // このパラメーターの値を設定します。
     // この値は正しいフォーマットであるため、
     // 検証エラーは発生しません。
     parameter.setValue("4000");
     out.println("Rule set parameter
     value on set correctly");
     break;
     }
   }
 }
```

ルール・セットが正しいことを確認するために、検証メソッドを呼び出すことができます。検証メソッドはすべてのオブジェクトで使用することが可能で、このメソッドが返す問題のリストを検査して問題を判別できます。オブジェクトで検証メソッドを呼び出すと、そのオブジェクトに含まれるすべてのオブジェクトでも検証メソッドが呼び出されます。

```
    // ルール・セットの変更を検証します。
    List<Problem> problems = ruleSet.validate();
    out.println("Rule set validated");

    // このテスト・ケースではエラーは発生しない
    // はずですが、問題があるかどうかを確認して、
    // 修復のための正しい処置を実行するか、または
    // エラーをレポートします。
    if (problems != null)
    {
     Iterator<Problem> problemIterator =
     problems.iterator();

     while (problemIterator.hasNext())
     {
      Problem problem = problemIterator.next();

      if (problem instanceof
      ProblemStartDateAfterEndDate)
      {
       out
       .println("Incorrect
        value entered for a
         parameter");
       return out.toString();
      }
     }
    } else
    {
     out.println("No problems found for the rule
     set");
    }
    // 有効なルール・ターゲットのリストを取得します。
    List<BusinessRule> ruleList2 =
    op.getAvailableTargets();

    // 誤ってスケジュールされることになる最初のルールを
```

```
    // 取得します。
    BusinessRule rule = ruleList2.get(0);

    // スケジュールされたルールの終了時刻が
    // 開始時刻の
    // 1 時間前になるようにエラー条件を設定します。
    // これにより、検証エラーが発生することになります。
    Date future = new Date();
    long futureTime = future.getTime() - 360000;

    // 誤ってスケジュールされた項目を追加するために、
    // 操作選択リストを取得します。
    OperationSelectionRecordList opList = op
    .getOperationSelectionRecordList();

    // スケジュールされたルールの新しいインスタンスを作成します。
    // 検証が行われるか、変更が追加されて公開が行われるまで、
    // エラーはスローされません。
    OperationSelectionRecord newRecord = opList
     .newOperationSelectionRecord(new Date(),
     new Date(
     futureTime), rule);
```

誤った日付のセットを指定したレコードを追加しても、それによってエラーが発生することはありません。変更が行われている間は、オーバーラップが発生したり、選択レコードが設定されていない操作が存在したりする場合があるためです。エラーが検出されるのは、操作選択レコードを持つビジネス・ルール・グループが公開されるときです。オブジェクトが公開される前に検証メソッドが呼び出され、エラーが存在する場合には例外がスローされます。

```
    // スケジュールされたルールのインスタンスを操作に追加します。
    // ここでもエラーは発生しません。
    opList.addOperationSelectionRecord(newRecord);
    out.println("New selection record added with
    incorrect schedule");

    // 条件値とアクションを更新した後、
    // ビジネス・ルール・グループを
    // 公開できます。
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含むリストを
    // 公開します。
    BusinessRuleManager.publish(publishList, true);
   }
  } catch (ValidationException e) {
  out.println("Validation Error");

  List<Problem> problems = e.getProblems();

  Iterator<Problem> problemIterator = problems.iterator();
  // 複数の問題が存在する可能性があります。
  // 問題を順に検索して 1 つずつ処理するか、問題を
  // レポートします。
  while (problemIterator.hasNext())
  {
   Problem problem = problemIterator.next();

   // それぞれの問題のタイプは異なっており、比較することが
```

```
    // 可能です。
    if (problem instanceof ProblemStartDateAfterEndDate)
    {
     out
     .println("Rule schedule is
      incorrect. Start date is after end
      date.");
     return out.toString();
     }
    // または....
    // 他のエラーを確認して、適切なエラー・
    // メッセージを表示するか、アクションを実行して
    // 問題を修正することもできます。
  }
 }catch (BusinessRuleManagementException e)
 {
  out.println("Error occurred.");
  e.printStackTrace();
 }
 return out.toString();
 }
}
```

例

例 15 の Web ブラウザー出力

例 15 の実行

```
Business Rule Group retrieved
Rule Set retrieved
Rule set parameter value on set correctly
Rule set validated
Validation Error
Rule schedule is incorrect. Start date is after end date.
```

その他の照会例

以下の例は、例 1 から 15 のアプリケーションには含まれていませんが、ビジネス・ルール・グループを取得する照会の作成方法を示す追加の例として提供されています。

これらの例では、異なるプロパティーとワイルドカード値 (「_」、「%」) が、さまざまな演算子 (AND、OR、LIKE、NOT_LIKE、EQUAL および NOT_EQUAL) とともに使用されています。

例

これらの例では、4 つのビジネス・ルール・グループのさまざまな組み合わせを返す照会を実行します。照会ではビジネス・ルール・グループのさまざまな属性とプロパティーを使用するため、これらの属性およびプロパティーを理解していることが重要です。

```
名前: BRG1
ターゲット名前空間 : http://BRG1/com/ibm/br/rulegroup
プロパティー:
organization, 8JAA
department, claims
ID, 00000567
地域: SouthCentralRegion
マネージャー: Joe Bean

名前: BRG2
```

```
ターゲット名前空間 : http://BRG2/com/ibm/br/rulegroup
プロパティー:
organization, 7GAA
department, accounting
ID, 0000047
ID_cert45, ABC
地域: NorthRegion

名前: BRG3
ターゲット名前空間: http://BRG3/com/ibm/br/rulegroup
プロパティー:
organization, 7FAB
department, finance
ID, 0000053
ID_app45, DEF
地域: NorthCentralRegion

名前: BRG4
ターゲット名前空間: http://BRG4/com/ibm/br/rulegroup
プロパティー:
organization, 7HAA
department, shipping
ID, 0000023
ID_app45, GHI
地域: SouthRegion
```

単一プロパティーによる照会:

これは、単一プロパティーによる照会例です。

```
List<BusinessRuleGroup> brgList = null;

brgList = BusinessRuleManager.getBRGsBySingleProperty(
  "department", QueryOperator.EQUAL,
  "accounting", 0, 0);
// BRG2 を返す
```

値の先頭と末尾にプロパティーとワイルドカード (%) を指定したビジネス・ルール・グループの照会:

これは、値の先頭と末尾にプロパティーとワイルドカード (%) を指定したビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND Prop (Prop はプロパティーを表す)
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "region", QueryOperator.LIKE,
  "%Region");

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode(
  "ID", QueryOperator.LIKE,
  "000005%");

QueryNode queryNode =
QueryNodeFactory.createAndNode(leftNode,
  rightNode);

brgList =
BusinessRuleManager.getBRGsByProperties(queryNode, 0, 0);
// BRG1 と BRG3 を返す
```

プロパティーとワイルドカード (_) によるビジネス・ルール・グループの照会:

これは、プロパティーとワイルドカード (_) によるビジネス・ルール・グループの照会例です。

```
brgList = BusinessRuleManager.getBRGsBySingleProperty("ID",
QueryOperator.LIKE, "00000_3", 0, 0);

// BRG3 と BRG4 を返す
```

複数のワイルドカード (「_」と「%」) を使用したプロパティーによるビジネス・ルール・グループの照会:

これは、複数のワイルドカード (「_」と「%」) を使用したプロパティーによるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("region",
QueryOperator.LIKE, "__uth%Region",
  0, 0);

// BRG1 と BRG4 を返す
```

NOT_LIKE 演算子とワイルドカード (「_」) によるビジネス・ルール・グループの照会:

これは、NOT_LIKE 演算子とワイルドカード (「_」) によるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
  "7__A", 0, 0);

// BRG1 と BRG3 を返す

brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
  "7%", 0, 0);

// BRG1 を返す
```

NOT_EQUAL 演算子によるビジネス・ルール・グループの照会:

これは、NOT_EQUAL 演算子によるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("department",
QueryOperator.NOT_EQUAL,
  "claims", 0, 0);
// BRG1 を返す
```

PropertyIsDefined によるビジネス・ルール・グループの照会:

これは、PropertyIsDefined によるビジネス・ルール・グループの照会例です。

```
PropertyIsDefinedQueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

brgList = BusinessRuleManager.getBRGsByProperties(node, 0,
0);

// BRG1 を返す
```

NOT PropertyIsDefined によるビジネス・ルール・グループの照会:

これは、NOT PropertyIsDefined によるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop  (Prop はプロパティーを表す)
QueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

NotNode notNode = QueryNodeFactory.createNotNode(node);

brgList = BusinessRuleManager.getBRGsByProperties(notNode,
0, 0);

// BRG1 を返す
```

単一の NOT ノードを使用した複数のプロパティーによるビジネス・ルール・グループの照会:

これは、単一の NOT ノードを使用した複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
  QueryOperator.LIKE, "00000%");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
  notNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

複数の NOT ノードが AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、複数の NOT ノードが AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop AND NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "cla%");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

AndNode andNode = QueryNodeFactory.createAndNode(notNode,
notNode2);
```

```
brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG1 と BRG2 を返す
```

複数の NOT ノードが OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、複数の NOT ノードが OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop OR NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department", QueryOperator.EQUAL,
  "claims");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
notNode2);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//BRG1、BRG2、BRG3、BRG4 を返す
```

複数の AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、複数の AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND Prop) AND (Prop AND Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode,rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE,"000004_");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
QueryOperator.EQUAL,
  "NorthRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);
```

```
AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft,andNodeRight);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

AND 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、AND 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND Prop) OR (Prop AND NOT Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode, rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "8JAA");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE, "%lRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, notNode);

OrNode orNode = QueryNodeFactory.createOrNode(andNodeLeft,
andNodeRight);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG2 と BRG3 を返す
```

AND 演算子と NOT 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、AND 演算子と NOT 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND NOT (Prop AND Prop)  (Prop はプロパティーを表す)
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE, "000005%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.EQUAL,
  "8JAA");
```

```
QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
  "%lRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
notNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG3 を返す
```

NOT 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、NOT 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT (Prop AND Prop) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8_A_");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
  "%lRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
rightNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND (Prop AND (Prop AND Prop))  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
```

```
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

PropertyIsDefinedQueryNode node2 =
QueryNodeFactory.createPropertyIsDefinedQueryNode("ID_cert4
5");

AndNode andNode = QueryNodeFactory.createAndNode(node2,
andNodeLeft);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);
// BRG2 を返す
```

ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND (Prop AND Prop)) AND Prop
// (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
"___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.LIKE,
"7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
QueryOperator.LIKE,
"%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_app45",QueryOp
erator.LIKE, "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);
```

```
brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

ネストされた AND 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会:

これは、ネストされた AND 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND (Prop AND (Prop AND NOT Prop))  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%lRegion");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,notNode);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "AB_");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
andNodeLeft);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND (Prop AND Prop)) AND Prop - 空の値を返す (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");
```

```
QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

//BRG は返さない
```

ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop OR (Prop OR Prop)) OR Prop  (Prop はプロパティーを表す)

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2,rightNode2);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode,orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG1 を返す
```

ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会:

これは、ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop OR (Prop OR NOT Prop)) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2,notNode);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode,orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会:

これは、ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop OR NOT(Prop OR Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
  "organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department",
```

```
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2,
  rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft = QueryNodeFactory.createOrNode(leftNode,
notNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// BRG3 を返す
```

ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会:

これは、ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT(Prop OR Prop) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%lRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
  "organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2,rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(notNode,leftNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// BRG2 と BRG4 を返す
```

AND 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会:

これは、AND 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会例です。

```
// 形式: AND リスト
List<QueryNode> list = new ArrayList<QueryNode>();
```

```
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7H%");

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

AND 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会:

これは、AND 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT ノードによる AND リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

NotNode notNode =

QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");
```

```
list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

OR 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会:

これは、OR 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会例です。

```
// 形式: OR リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

OR 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会:

これは、OR 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT ノードによる OR リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");
```

```
list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

list.add(leftNode2);

OrNode orNode = QueryNodeFactory.createOrNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//BRG1、BRG2、BRG3、BRG4 を返す
```

共通操作クラス

このセクションには、共通操作を単純化するために例の中で使用した追加クラスを記載します。

Formatter クラス

このクラスは、さまざまな例を表示するための各種のメソッドを提供し、各種の HTML タグを追加して出力フォーマットを設定します。

```
package com.ibm.websphere.sample.brules.mgmt;

public class Formatter {

private StringBuffer buffer;

public Formatter()
{
  buffer = new StringBuffer();
}

public void println(Object o)
{
  buffer.append(o);
  buffer.append("<br>¥n");
  }

  public void print(Object o)
  {
    buffer.append(o);
  }
```

```
  public void printlnBold(Object o)
  {
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b><brbr>¥n");
  }

  public void printBold(Object o)
  {
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b>");
  }

  public String toString()
  {
    return buffer.toString();
  }

  public void clear()
  {
    buffer = new StringBuffer();
  }
}
```

RuleArtifactUtility クラス

このユーティリティー・クラスには 2 つの public メソッドがあります。最初の public メソッドは、デシジョン・テーブルを出力するためのメソッドです。このメソッドは、再帰を使用する private メソッドを使用して、デシジョン・テーブルの条件とアクションを出力します。2 番目の public メソッドは、ルール・セットを出力するためのメソッドです。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.RuleTemplate;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableRule;
import
com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionTermDefinition;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;
```

```
public class RuleArtifactUtility
{
  static Formatter out = new Formatter();

  /*
  デシジョン・テーブルを印刷するためのメソッド (条件とアクションを
  HTML の表形式で印刷)。デシジョン・テーブルの
  ケース・エッジを再帰的に処理する個別のメソッドにより、
  条件とアクションを印刷する。
  */

  public static String printDecisionTable(BusinessRule
  ruleArtifact)
  {
    out.clear();
    out.printlnBold("Decision Table");
    DecisionTable decisionTable = (DecisionTable)
    ruleArtifact;
    out.println("Name: " +
    decisionTable.getName());
    out.println("Namespace: " +
    decisionTable.getTargetNameSpace());

    // 条件とアクションのテーブルを処理する前に、
    // デシジョン・テーブルの初期ルールを印刷する
    DecisionTableRule initRule =
    decisionTable.getInitRule();
    if (initRule != null)
    {
      out.printBold("Init Rule: ");
      out.println(initRule.getName());
      out.println("Display Name: " +
      initRule.getDisplayName());
      out.println("Description: " +
      initRule.getDescription());
      // ユーザーの拡張プレゼンテーションにより、パラメーターの値が
      // プレゼンテーションに対して自動的に設定され、テンプレートに
      // 初期ルールが定義されていた場合は、ユーザーの拡張プレゼンテーションが
      // 表示可能になる。テンプレートが定義されていない場合、ユーザーの
      // 拡張プレゼンテーションは通常のプレゼンテーションと同じになる。
      out.println("Extended User
      Presentation: "
        +
      initRule.getExpandedUse
      rPresentation());
      // テンプレートに初期ルールが定義されている場合の
      // 通常のユーザー・プレゼンテーションは、
      // パラメーターを置換可能なプレースホルダーが含まれる
      // ストリングとなる。
      // テンプレートに初期ルールが定義されていない場合、
      // ユーザー・プレゼンテーションはプレースホルダーのない
      // 単純なストリングになる。
      // プレースホルダーは、[n] の形式で表記される。
      // この n は、テンプレート内のパラメーターの索引値 (0 基準)
      // を表す。
      // この値を使用して、編集可能なパラメーターが設定されている
      // フィールドを編集するためのインターフェースを
      // 作成することができる。
      out.println("User Presentation: " +
      initRule.getUserPresentation());
      // テンプレートの有無にかかわらず、初期ルールが定義されて
      // いる可能性がある。パラメーターにアクセスする前に、
      // テンプレートが使用されていたかどうかを確認する。
      if (initRule instanceof
      DecisionTableTemplateInstanceRule)
      {
        DecisionTableTemplateIn
```

```
        stanceRule
        templateInstance =
        (DecisionTableTemplateI
        nstanceRule) initRule;

        RuleTemplate template =
        templateInstance.getRul
        eTemplate();

        List<Parameter>
        parameters =
        template.getParameters(
        );
        Iterator<Parameter>
        paramIterator =
        parameters.iterator();

        Parameter parameter =
        null;

        while
        (paramIterator.hasNext(
        )) {
        parameter =
        paramIterator.next();

         out.println("Parameter
        Name: " +
        parameter.getName());
        out.println("Parameter
        Value: "
        +
        templateInstance.getPar
        ameterValue(parameter
        .getName()));
        }
      }
    }
    // デシジョン・テーブルの残りの部分については root から
    // 処理を開始し、異なるケース・エッジとアクションを
    // 再帰的に処理する
    TreeBlock treeBlock =
    decisionTable.getTreeBlock();
    TreeNode treeNode = treeBlock.getRootNode();

    printDecisionTableConditionsAndActions(treeNode
    , 0);
    out.println("");
    return out.toString();
  }
  /*ケース・エッジを再帰的に処理して
  条件とアクションを印刷するメソッド
*/
  static private void printDecisionTableConditionsAndActions(
    TreeNode treeNode, int indent)
  {
    out.print("<table border=¥"1¥">");
    if (treeNode instanceof ConditionNode)
    {
      // 現在の TreeNode に対するケース・エッジを取得し、
      // 各ケース・エッジについて条件を印刷する
      ConditionNode conditionNode =
      (ConditionNode) treeNode;

      List<CaseEdge> caseEdges =
      conditionNode.getCaseEdges();
      Iterator<CaseEdge> caseEdgeIterator
```

```
= caseEdges.iterator();

CaseEdge caseEdge = null;

while (caseEdgeIterator.hasNext())
{
  out.print("<tr>");
  // これが条件ノードの条件の開始部分である場合は、
  // 条件の内容を印刷する
  if (indent == 0)
  {
  out.print("<td>");

  TreeConditionTermDefinition
  termDefinition =
  conditionNode
  .getTermDefinition();

  out.print(termDefinitio
  n.getUserPresentation()
  );
  out.print("</td>");
  indent++;
  } else {
  // ケース・エッジの条件内容を印刷したら、
  // 残りのケース・エッジについてはスキップする
  out.print("<td></td>");
  }

  caseEdge =
  caseEdgeIterator.next()
  ;

  out.print("<td>");

  // caseEdge がテンプレートで定義されているかを確認
  if
  (caseEdge.getValueDefin
  ition() != null)
  {
  TemplateInstanceExpress
  ion templateInstance =
  caseEdge
  .getValueTemplateInstan
  ce();

  out.println(templateIns
  tance.getExpandedUserPr
  esentation());

  TreeConditionValueDefin
  ition valueDef =
  caseEdge
  .getValueDefinition();

  out.println(valueDef.ge
  tUserPresentation());

  Template template =
  templateInstance.getTem
  plate();

  // テンプレート定義のパラメーターを取得し、
  // パラメーターの名前と値を印刷する
  List<Parameter>
  parameters =
  template.getParameters(
```

```
);
Iterator<Parameter>
paramIterator =
parameters.iterator();

List<ParameterValue>
parameterValues =
templateInstance
.getParameterValues();
Iterator<ParameterValue
> paramValues =
parameterValues
.iterator();

Parameter parameter =
null;
ParameterValue
parameterValue = null;

while
(paramIterator.hasNext(
) &&
paramValues.hasNext())
{
parameter =
paramIterator.next();
parameterValue =
paramValues.next();

out.println("Parameter
Name: " +
parameter.getName());
out.println("Parameter
Value: "
  +
parameterValue.getValue
());
}
}

out.print("</td><td>");
// caseEdge の子ノードを印刷
printDecisionTableCondi
tionsAndActions(caseEdg
e.getChildNode(),
0);

out.print("</td></tr>")
;
}


// 存在する場合は Otherwise 条件を追加
TreeNode otherwise =
conditionNode.getOtherwiseCase();

if (otherwise != null)
{
  out.print("<tr><td></td>
  <td>Otherwise</td><td>
  ");
  // Otherwise
  ConditionNode を印刷
  printDecisionTableCondi
  tionsAndActions(otherwi
  se, 0);
  out.print("</td></td>")
```

```
    ;
  }
  out.print("</table>");
} else {
  // ActionNode が見つかり、TreeActions を印刷するための
  // 追加ロジックが必要な場合
  ActionNode actionNode =
  (ActionNode) treeNode;
  List<TreeAction> treeActions =
  actionNode.getTreeActions();

  Iterator<TreeAction>
  treeActionIterator =
  treeActions.iterator();

  TreeAction treeAction = null;

  // 印刷対象の複数の TreeActions を
  // ActionNode に格納
  while
  (treeActionIterator.hasNext())
  {
    out.print("<tr>");
    treeAction =
    treeActionIterator.next
    ();

    TreeActionTermDefinitio
    n treeActionTerm =
    treeAction
    .getTermDefinition();

    if (indent == 0) {
    out.print("<td>");
    out.print(treeActionTer
    m.getUserPresentation()
    );
    out.print("</td>");
    }
    out.print("<td>");
    TemplateInstanceExpress
    ion templateInstance =
    treeAction
    .getValueTemplateInstan
    ce();

    // パラメーターの名前と値を処理する前に、
    // TreeAction に対してテンプレートが指定
    // されていたかどうかを確認する
    if (templateInstance !=
    null) {
    out.println(templateIns
    tance.getExpandedUserPr
    esentation());

    Template template =
    templateInstance.getTem
    plate();

    List<Parameter>
    parameters =
    template.getParameters(
    );

    Iterator<Parameter>
    paramIterator =
    parameters.iterator();
```

```
        List<ParameterValue>
        parameterValues =
        templateInstance
        .getParameterValues();
        Iterator<ParameterValue
        > paramValues =
        parameterValues
        .iterator();

        Parameter parameter =
        null;
        ParameterValue
        parameterValue = null;

        while
        (paramIterator.hasNext(
        ) &&
        paramValues.hasNext())
        {
        {parameter =
        paramIterator.next();
        parameterValue =
        paramValues.next();

        out.println(" Parameter
        Name: " +
        parameter.getName());
        out.println(" Parameter
        Value: "
        +
        parameterValue.getValue
        ());

        }
        } else
        {
        // テンプレートが使用されていなかった場合は、
        // UserPresentation のみ使用可能 (ルールを
        // 作成した際に指定した場合)
        out.print(treeAction.ge
        tValueUserPresentation(
        ));
        }

        out.print("</td></tr>")
        ;
      }
      out.print("</table>");
    }
  }
  /*
    ルール・セットを印刷するメソッド
  */
public static String printRuleSet(BusinessRule
ruleArtifact)
{
   out.clear();
   out.println1Bold("Rule Set");
   RuleSet ruleSet = (RuleSet) ruleArtifact;
   out.println("Name: " + ruleSet.getName());
   out.println("Namespace: " +
   ruleSet.getTargetNameSpace());

   // ルール・セットのルールをルール・ブロックに格納
   RuleBlock ruleBlock =
   ruleSet.getFirstRuleBlock();
```

```
Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

RuleSetRule rule = null;

// ルール・ブロック内のルールを反復処理
while (ruleIterator.hasNext())
{
  rule = ruleIterator.next();
  out.printBold("Rule: ");
  out.println(rule.getName());
  out.println("Display Name: " +
  rule.getDisplayName());
  out.println("Description: " +
  rule.getDescription());
  // ユーザーの拡張プレゼンテーションにより、パラメーターの値が
  // プレゼンテーションに対して自動的に設定され、テンプレートに
  // 初期ルールが定義されている場合は、ユーザーの拡張プレゼンテーションが
  // 表示可能になる。テンプレートが定義されていない場合、ユーザーの
  // 拡張プレゼンテーションは通常のプレゼンテーションと同じになる。
  out.println("Expanded User
  Presentation: "
    +
  rule.getExpandedUserPre
  sentation());
  // 通常のユーザー・プレゼンテーションにはストリング内にプレース
  // ホルダーが設定されるため、テンプレートに初期ルールが定義されて
  // いた場合はパラメーターを置き換えることができる。テンプレートに
  // 初期ルールが定義されていなかった場合、ユーザー・プレゼンテーションは
  // プレースホルダーのない単純なストリングになる。プレースホルダーは、
  // [n] の形式で表記される。この n は、テンプレート内のパラメーターの
  // 索引値 (0 基準) を表す。この値を使用して、編集可能なパラメーターが
  // 設定されているフィールドを編集するためのインターフェースを
  // 作成することができる。
  out.println("User Presentation: " +
  rule.getUserPresentation());

  // ルールがテンプレートで定義されていたかどうかを確認
  if (rule instanceof
  RuleSetTemplateInstanceRule) {
    RuleSetTemplateInstance
    Rule templateInstance =
    (RuleSetTemplateInstanc
    eRule) rule;

    RuleSetRuleTemplate
    template =
    templateInstance
    .getRuleSetRuleTemplate
    ();

    List<Parameter>
    parameters =
    template.getParameters(
    );
    Iterator<Parameter>
    paramIterator =
    parameters.iterator();

    Parameter parameter =
    null;

    // すべてのパラメーターを取得して、名前と値を出力
    while
    (paramIterator.hasNext(
    ))
```

```
        {
        parameter =
        paramIterator.next();

        out.println("Parameter
        Name: " +
        parameter.getName());
        out.println("Parameter
        Value: "
          +
        templateInstance.getPar
        ameterValue(
        parameter.getName()).ge
        tValue());
        }
      }
    }
    out.println("");
    return out.toString();
  }
}
```

ウィジェット・プログラミング

Business Space では、IBM のさまざまなビジネス・プロセス・マネジメント製品で利用できるウィジェットが提供されます。加えて、ユーザー自身でも独自のウィジェットを作成することができ、必要に応じて IBM が提供するウィジェットと統合することも可能です。以下の参照先には、独自のウィジェットを作成する理由と作成方法、およびウィジェット作成で使用できる API のリファレンスが記載されています。

Business Space インフォメーション・センターでは、以下のトピックを参照してください。

- ウィジェット開発の概要
- ウィジェット・プログラミング・ガイド

第 4 章 ビジネス・プロセスおよびタスク用クライアント・アプリケーションの開発

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作用し、例えば、プロセスが開始すると、タスクが要求されて完了します。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クライアントを開発することもできます。

このタスクについて

これらのクライアントは、Business Process Choreographer Explorer JavaServer Faces (JSF) コンポーネントを活用する Enterprise JavaBeans (EJB) クライアント、Web サービス・クライアント、または Web クライアントです。これらのクライアントを開発するために、Business Process Choreographer は、Enterprise JavaBeans (EJB) API と Web サービス用インターフェースを提供しています。EJB API は、別の EJB アプリケーションを含むすべての Java アプリケーションによってアクセスできます。 Web サービス用インターフェースには、Java 環境または Microsoft .Net 環境のいずれかからアクセスできます。

ビジネス・プロセスおよびヒューマン・タスクと対話するためのプログラミング・インターフェースの比較

ビジネス・プロセスおよびヒューマン・タスクと対話するクライアント・アプリケーションの作成には、Enterprise JavaBeans (EJB)、Web サービス、Java Message Service (JMS) および Representational State Transfer (REST) サービスなどの汎用プログラミング・インターフェースを使用できます。これらのインターフェースには、それぞれ異なる特性があります。

選択するプログラミング・インターフェースは、いくつかの要因によって左右されます。この要因には例えば、クライアント・アプリケーションが提供しなければならない機能、既存のエンド・ユーザー・クライアント・インフラストラクチャーがあるかどうか、ヒューマン・ワークフローを処理するかどうか、などがあります。使用するインターフェースを決定するためのヒントとして、EJB、Web サービス、JMS、および REST プログラミング・インターフェースの特性を比較した表を以下に示します。

	EJB インターフェース	Web サービス・インターフェース	JMS メッセージ・インターフェース	REST インターフェース
機能	このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。一般的にプロセスおよびタスクを処理するクライアントを作成する場合に、このインターフェースを使用します。	このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。プロセスとタスクからなる既知のセットに関してクライアントを作成する場合に、このインターフェースを使用します。	このインターフェースはビジネス・プロセスにのみ使用できます。既知のプロセス・セットに関してメッセージング・クライアントを作成する場合に、このインターフェースを使用します。	このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。このインターフェースは、既知の一式のプロセスおよびタスクに対する Web 2.0 スタイルのクライアントを作成するために使用します。
データ処理	ビジネス・オブジェクト・メタデータにアクセスするためのスキーマのリモート成果物ロードがサポートされます。 EJB クライアント・アプリケーションが接続先の WebSphere Process Server と同じセル内で稼働する場合、プロセスおよびタスクのビジネス・オブジェクトに必要なスキーマをクライアント上で使用可能にする必要はなく、リモート成果物ローダー (RAL) を使用してそれらのスキーマをサーバーからロードすることができます。 クライアント・アプリケーションが完全な WebSphere Process Server サーバー・インストール済み環境で稼働する場合は、セル間でも RAL を使用できます。ただし、クライアント・アプリケーションが WebSphere Process Server クライアント・インストール済み環境で稼働するセル間セットアップでは、RAL を使用できません。	入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。	入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。	入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。
クライアント環境	WebSphere Process Server インストール済み環境または WebSphere Process Server クライアント・インストール済み環境。	Web サービス呼び出しをサポートするランタイム環境 (例えば Microsoft .NET 環境など)。	JMS クライアントをサポートするランタイム環境 (例えば、SCA JMS インポートを使用する SCA モジュールなど)。	REST クライアントをサポートする任意のランタイム環境。

	EJB インターフェース	Web サービス・インターフェース	JMS メッセージ・インターフェース	REST インターフェース
セキュリティー	Java Platform Enterprise Edition (Java EE) セキュリティー。	Web サービス・セキュリティー。	WebSphere Process Server インストール済み環境用の Java Platform Enterprise Edition (Java EE) セキュリティー。JMS クライアント・アプリケーションが API メッセージを格納するキューを、例えば WebSphere MQ セキュリティー・メカニズムを使用して、保護することもできます。	REST メソッドを呼び出すクライアント・アプリケーションは、適切な HTTP 認証メカニズムを使用する必要があります。

単一の操作を複数のプロトコルによって公開できます。 複数の異なるプロトコルで同じ操作を使用している場合、次の考慮事項に注意してください。

- Web サービスおよび REST インターフェースでは、すべてのオブジェクト ID (PIID、 AIID、および TKIID など) は string タイプで表されます。 タイプ・セーフなオブジェクト ID を想定するのは EJB API インターフェースのみです。
- 操作の多重定義は EJB メソッドの場合のみ使用され、WSDL 操作では使用されません。場合によっては、複数の WSDL 操作が存在します。また、1 つの WSDL 操作のみが存在し、省略 (`minOccurs="0"`) またはヌル値 (`nillable="true"`) のいずれかによって、すべてのパラメーターのバリエーションが許可される場合もあります。
- EJB メソッドによっては、XML 名前空間およびローカル名が個別のパラメーターとして渡される場合があります。 多くの WSDL 操作では、これらのパラメーターを渡すのに QName XML スキーマ・タイプが使用されます。
- 長時間実行されている WSDL 要求/応答操作の非同期対話 (EJB インターフェースの callWithReplyContext 操作または WSDL インターフェースの callAsync 操作など) は、JMS インターフェースの call 操作によって表されます。
- EJB インターフェースは API オブジェクトのセットを返します。これらのオブジェクトは、含まれているフィールドに対して getter メソッドおよび setter メソッドを公開します。 Web サービスおよび REST インターフェースは、複合タイプ (XML または JSON) の文書をクライアントに返します。
- ヒューマン・タスクに対する Human Task Manager サービス操作も、ヒューマン・タスクを呼び出すアクティビティーに対する Business Flow Manager サービス操作として使用可能な場合があります。

関連タスク

EJB クライアント・アプリケーションの開発
EJB API は、WebSphere Process Server 上にインストールされているビジネス・プロセスやヒューマン・タスクを処理する EJB クライアント・アプリケーションを開発するための汎用的な方法をいくつか提供します。

Web サービス API クライアント・アプリケーションの開発
Business Process Choreographer Web サービス API を介してビジネス・プロセス・アプリケーションとヒューマン・タスク・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。クライアント・アプリケーションの開発プロセスは、Web サービス・プロキシーの生成や、クライアント・アプリケーションへのセキュリティー・ポリシーおよびトランザクション・ポリシーの追加など、多数の必須のステップとオプションのステップで構成されています。

JMS クライアント・アプリケーションの開発
Java Messaging Service (JMS) API を介してビジネス・プロセス・アプリケーションに非同期でアクセスするクライアント・アプリケーションを開発できます。

ビジネス・プロセスおよびタスク・データに対する照会

長時間実行ビジネス・プロセスおよびヒューマン・タスクのインスタンス・データはデータベースに永続的に格納され、照会によってアクセスできます。また、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・テンプレートのテンプレート・データには、QUERY インターフェースを使用してアクセスできます。

Business Process Choreographer では、EJB 照会インターフェース、照会 API、および照会テーブル API を使用できます。

プロセスまたはタスクの関連データにアクセスするクライアントによっては、インターフェースの 1 つ以上に対応している可能性があります。Business Process Choreographer では、タスクおよびプロセス・リスト・データを照会するために REST および Web サービス API を使用できます。ただし、大容量のプロセス・リストおよびタスク・リストの照会には、パフォーマンス上の理由から、Business Process Choreographer EJB 照会テーブル API および REST 照会テーブル API を使用してください。

プロセスおよびタスク・データを検索するためのプログラミング・インターフェースの比較

Business Process Choreographer には、プロセスおよびタスク・データを検索するための照会テーブル API および照会 API があります。これらのインターフェースには、それぞれ異なる特性があります。

選択する照会インターフェースは、いくつかの要因によって左右されます。この要因には、クライアント・アプリケーションが提供する必要がある機能、既存のエンド・ユーザー・クライアント・インフラストラクチャーが存在するかどうか、パフォーマンス上の考慮事項などがあります。使用するインターフェースを決定する際の参考として、照会テーブルおよび照会プログラミング・インターフェースの特性を比較した表を以下に示します。

特性	照会テーブル API	照会 API
アベイラビリティー	照会テーブル API は、Business Flow Manager EJB インターフェースおよび REST プログラミング・インターフェースで使用可能です。	照会 API は、EJB、Web サービス、JMS、および REST プログラミング・インターフェースで使用可能です。
内容検索のためのメソッド	この API には、以下のメソッドがあります。 • queryEntities • queryEntityCount • queryRows • queryRowCount	この API には、以下のメソッドがあります。 • query • queryAll • queryProcessTemplates • queryTaskTemplates
メタデータ検索のためのメソッド	この API には、以下のメソッドがあります。 • getQueryTableMetaData • findQueryTableMetaData	この API には、以下のメソッドがあります。 • QueryResultSet.getColumnType
照会テーブル名	照会テーブル API を実行する照会テーブルを指定します。一度に照会できる照会テーブルは 1 つだけです。 例えば `queryEntities("CUST.TASKS", ...)` と記述します。	SELECT 文節は、照会を実行する列および定義済みデータベース・ビューを指定します。この仕様は SQL の select 文節と同様です。 例えば `query("TASK.TKIID, TASK.STATE, WORK_ITEM.REASON", ...)` と記述します。
SELECT 文節および選択された属性	照会テーブル API のフィルター・オプションを使用して、照会が返す属性を指定します。照会は 1 つの照会テーブルに対して実行されるため、属性は名前によって一意的に識別できます。	SELECT 文節を使用して、属性を指定します。属性名の構文は、*view_name.attribute_name* です。例えば、タスク状態を検索するには、照会で TASK.STATE と指定します。
WHERE 文節およびフィルター	照会の結果をさらにフィルターに掛けるには、照会テーブル API で queryCondition プロパティーを使用します。1 次照会テーブル・フィルター、許可フィルター、または照会テーブル・フィルターが照会テーブル定義で指定されている場合、照会テーブルは、事前にフィルターに掛けた内容を提供します。	WHERE 文節を使用して、照会の結果をフィルター操作します。
WHERE 文節および選択基準	この形式の照会テーブル API では、照会 API の WHERE 文節は不要です。さらにフィルターに掛けるには、照会テーブル API で queryCondition プロパティーを使用します。 照会テーブル定義の選択基準は、接続照会テーブルの特定のプロパティーを選択します。これは、照会 API の WHERE 文節によるフィルター処理に加えて実行されます。	選択基準は、照会 API では使用できません。しかし、選択基準は、QUERY_PROPERTY、TASK_CPROP、または TASK_DESC の名前またはロケールなどを定義する WHERE 文節の特定の部分と似ています。 例えば、`QUERY_PROPERTY.NAME='xyz'` の WHERE 文節は、QUERY_PROPERTY 接続照会テーブルの照会テーブル定義で選択基準として `NAME='xyz'` を指定することと同じです。
作業項目および許可	WORK_ITEM 照会テーブルを使用して、作業項目にアクセスします。作業項目の使用方法は、照会テーブルの開発時に照会テーブル定義でカスタマイズしたり、AuthorizationOptions オブジェクトまたは AdminAuthorizationOptions オブジェクトを使用して照会テーブル API でカスタマイズすることができます。 例えば、TASK 照会テーブルを照会するときに全員作業項目を除外するには、queryCondition プロパティー `WI.EVERYBODY=0` を指定するか、AuthorizationOptions プロパティーで `setUseEverbody(Boolean.FALSE)` を指定します。	WORK_ITEM ビューを使用して、作業項目にアクセスします。照会結果では、作業項目の 4 つのタイプ (全員、個人、グループ、および継承) のすべてが考慮されます。作業項目を特定のタイプの作業項目にフィルタリングするには、WHERE 文節をカスタマイズします。 例えば、「全員」作業項目を除外するには、WHERE 文節に `WORK_ITEM.EVERYBODY=0` を指定します。
パラメーター	複合照会テーブルのフィルターおよび選択基準にパラメーターを使用できます。	保管照会文を使用していない場合は、照会 API のパラメーターを使用できません。

特性	照会テーブル API	照会 API
保管照会文と照会テーブル	保管照会文と照会テーブルの違いは、保管照会文は 1 つの特定の照会用に定義されるのに対し、照会テーブルは特定の照会セット用に定義されるという点にあります。例えば、照会テーブル定義では、order-by 文節を指定することはできません。通常、この情報は、照会の実行時にのみ利用できるからです。	保管照会文を使用して、定義済みのオプション・セットが含まれている照会を実行できます。
実体化ビュー	実体化ビューは、照会テーブル API では使用できません。	実体化ビューでは、照会のパフォーマンス向上のために、データベース技術が使用されます。
カスタム・テーブル	補足照会テーブルは、カスタム・テーブルと同じ機能を提供します。	カスタム・テーブルは、Business Process Choreographer データベース・スキーマの外部にあるデータを照会に組み込むために使用されます。
queryAll および許可オプション	queryAll 機能は、AuthorizationOptions オブジェクトではなく AdminAuthorizationOptions オブジェクトで提供されます。これは、照会テーブル API に渡すことができます。呼び出し元は、BPESystemAdministrator、TaskSystemAdministrator、BPESystemMonitor、または TaskSystemMonitor である必要があります。	queryAll メソッドは、BPESystemAdministrator Java EE ロールを持つユーザーが、特定のユーザーまたはグループの作業項目によって制限されることなく、照会結果内のすべてのオブジェクトを返すために使用できます。
国際化対応	照会テーブルの使用時に、照会テーブルの属性および照会テーブルで、ローカライズされた表示名および説明を使用できるようになっています。	選択したビューの列の名前が、データベースに表示されているとおりに、または select 文節で指定されているとおりに返されます。

Business Process Choreographer での照会テーブル

照会テーブルは、Business Process Choreographer データベース・スキーマに含まれているデータに対するタスクおよびプロセス・リストの照会をサポートします。これには、ヒューマン・タスク・データ、Business Process Choreographer によって管理されるビジネス・プロセス・データ、および外部ビジネス・データが含まれます。照会テーブルは、クライアント・アプリケーションが使用できる Business Process Choreographer のデータの抽象化を提供します。このため、クライアント・アプリケーションに、照会テーブルを実際に実装する必要はありません。照会テーブル定義は Business Process Choreographer コンテナーにデプロイされており、照会テーブル API を使用してアクセス可能です。

照会テーブルには、以下の 3 つのタイプがあります。

- 定義済み照会テーブル
- 補足照会テーブル
- 複合照会テーブル

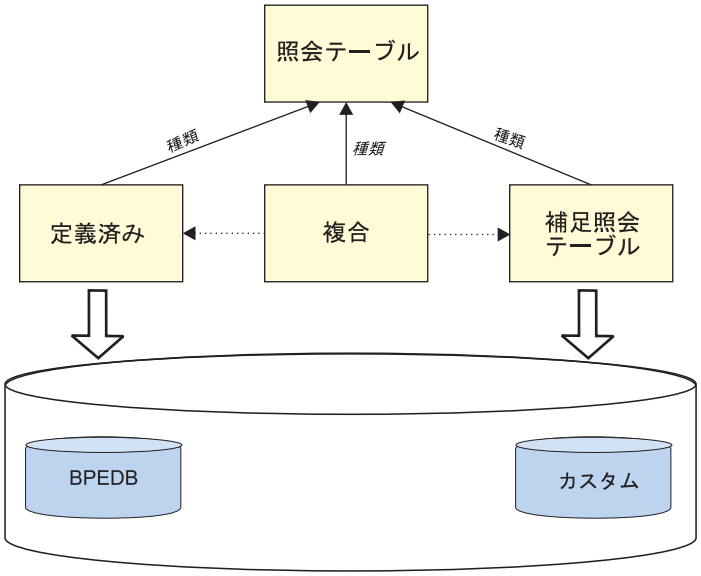


図 72. Business Process Choreographer での照会テーブル

照会テーブルは、照会テーブル・ランタイムの類似したモデルを使用して表現され、照会テーブル API を使用して照会することができます。定義済み照会テーブルおよび補足照会テーブルはデータベース内のテーブルまたはビューを直接指しますが、複合照会テーブルは、このデータの各部分を組み合わせて単一の照会テーブルで表します。

照会テーブルは、Business Process Choreographer の事前定義データベース・ビューと既存の QUERY インターフェースを拡張するもので、以下のような特徴があります。

- パフォーマンスの観点から最適化されたアクセス・パターンを使用して、プロセスおよびタスク・リストの照会を実行するために最適化されています。
- 必要な情報へのアクセスが簡素化され、統合されます。
- 許可およびフィルターのオプションをきめ細かく構成できます。

照会テーブルはカスタマイズできます。例えば、特定のシナリオに関連するタスクまたはプロセス・インスタンスだけが含まれるように照会テーブルを構成することができます。パフォーマンスが重要な場合 (大容量のプロセス・リストやタスク・リストを照会する場合など) も、照会テーブルを使用することをお勧めします。

Query Table Builder が Eclipse プラグインとして提供されていて、以下を行うことができます。

- 複合照会テーブルおよび補足照会テーブルの開発
- XML 形式での照会テーブル定義のインポートおよびエクスポート

Query Table Builder は、WebSphere Business Process Management SupportPacs のサイトでダウンロードできます。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

定義済み照会テーブル

定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

定義済み照会テーブルは、照会テーブル API を使用して直接照会できます。照会テーブル API を使用してテーブルにアクセスする場合は、照会 API を使用する場合と比較して、より多くの構成用オプションを使用できます。

プロパティー

定義済み照会テーブルには、以下のプロパティーがあります。

表 25. 定義済み照会テーブルのプロパティー

プロパティー	説明
名前	照会テーブル名は、いずれかの定義済みデータベース・ビューの名前を大文字で表記したもの (TASK など) です。
属性	定義済み照会テーブルの属性は、照会に使用できる情報を定義します。これらは、定義済みデータベース・ビューによって指定される大文字の列の名前です。 属性は、名前とタイプで定義します。属性のタイプは、以下のいずれかです。 • **ブール**: ブール値 • **10 進数**: 浮動小数点数 • **ID**: オブジェクト ID (TASK 照会テーブルの TKIID など) • **番号**: 整数、短整数、または長整数 • **ストリング**: ストリング • **タイム・スタンプ**: タイム・スタンプ

表 25. 定義済み照会テーブルのプロパティー (続き)

プロパティー	説明
許可	定義済み照会テーブルは、インスタンス・ベースまたはロール・ベースの許可を使用します。 • インスタンス・データを含む定義済み照会テーブルは、インスタンス・ベースの許可を必要とします。そのため、照会を実行するユーザーの作業項目を含むオブジェクトのみが返されます。ただし、AdminAuthorizationOptions オブジェクトを使用すると、任意のユーザーの作業項目が存在するかどうかが検査されます。それらの照会では、ユーザーは BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があります。 • テンプレート・データを含む定義済み照会テーブルでは、ロール・ベースの許可が必要です。すなわち、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っているユーザーのみが、それらの照会テーブルの内容にアクセスできます。

インスタンス・データを含む定義済み照会テーブル

以下の表は、インスタンス・データが含まれている定義済み照会テーブルを示しています。このような照会テーブルについては、以下の点が当てはまります。

- 複合照会テーブルの 1 次照会テーブルとして使用できます。
- 直接照会が実行される場合に、インスタンス・ベースの許可を使用します。これは、許可情報を格納するビュー (定義済み WORK_ITEM ビューまたは照会テーブル) との結合 (SQL-) によって実現されます。
- インスタンス・データ (タスク・インスタンスまたはプロセス・インスタンスのデータなど) が入ります。

表 26. インスタンス・データが含まれている定義済み照会テーブル

インスタンス・データ	照会テーブル名
プロセス・インスタンスのアクティビティーについての情報。	ACTIVITY
	ACTIVITY_ATTRIBUTE
	ACTIVITY_SERVICE
ヒューマン・タスクに属するエスカレーションについての情報。	ESCALATION
	ESCALATION_CPROP
	ESCALATION_DESC
プロセス・インスタンスについての情報。	PROCESS_ATTRIBUTE
	PROCESS_INSTANCE
	QUERY_PROPERTY
ヒューマン・タスクについての情報。	TASK
	TASK_CPROP
	TASK_DESC

WORK_ITEM 照会テーブルにもインスタンス・データは含まれていますが、このデータを 1 次照会テーブルまたは接続照会テーブルとして使用することはできません。作業項目の情報は、インスタンス・ベースの許可を使用する照会テーブルの照会時に暗黙的に使用可能です。つまり、WORK_ITEM 照会テーブルの属性は、その属性が照会テーブルによって明示的に指定されていないとしても、インスタンス・ベースの許可を使用する照会テーブルの照会時に使用できます。

テンプレート・データを含む定義済み照会テーブル

テンプレート・データを含む定義済み照会テーブルは、ロール・ベースの許可を必要とします。これらの定義済み照会テーブルは、管理者が AdminAuthorizationOptions オブジェクトを使用することによってのみ照会できます。

以下の表は、テンプレート・データが含まれている定義済み照会テーブルを示しています。このような照会テーブルについては、以下の点が当てはまります。

- 複合照会テーブルの 1 次照会テーブルとして使用できます。
- 直接照会が実行される場合に、ロール・ベースの許可を使用します。すなわち、API 照会メソッドを使用する呼び出し元が、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があり、AdminAuthorizationOptions を使用する必要があります。
- タスク・テンプレートまたはプロセス・テンプレートのテンプレート・データなどのテンプレート・データが含まれます。

表 27. テンプレート・データが含まれている定義済み照会テーブル

テンプレート・データ	照会テーブル名
アプリケーション・コンポーネントについての情報。	APPLICATION_COMP
エスカレーション・テンプレートについての情報。	ESC_TEMPL
	ESC_TEMPL_CPROP
	ESC_TEMPL_DESC
プロセス・テンプレートについての情報。	PROCESS_TEMPLATE
	PROCESS_TEMPL_ATTR
タスク・テンプレートについての情報。	TASK_TEMPL
	TASK_TEMPL_CPROP
	TASK_TEMPL_DESC

関連概念

『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

343 ページの『複合照会テーブル』
Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

351 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder を使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

371 ページの『照会テーブルの照会』
照会は、照会テーブル API (Business Flow Manager EJB でのみ使用可能) を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。

360 ページの『照会テーブルの許可』
照会テーブルで照会を実行するときには、インスタンス・ベースの許可、ロール・ベースの許可、または許可なしを使用できます。

補足照会テーブル

Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

補足照会テーブルは、Business Process Choreographer データベースのデータベース表またはデータベース・ビューと関連しています。これらは、カスタマー・アプリケーションによって保守されるビジネス・データが含まれている照会テーブルです。補足照会テーブルは、定義済み照会テーブルに含まれている情報に加えて、複合照会テーブルの情報も提供します。

補足照会テーブルには、以下のプロパティーがあります。

表 28. 補足照会テーブルのプロパティー

プロパティー	説明
名前	照会テーブル名は、Business Process Choreographer インストール済み環境内で固有でなければなりません。この名前は、照会の実行時に、照会される照会テーブルを識別するために使用されます。 照会テーブルは、名前 (*prefix.name* と定義される) によって一意的に識別されます。 *prefix.name* の最大長は 28 文字です。接頭部は、予約済みの接頭部 `'IBM'` と異なっていなければなりません。例えば、`'COMPANY.BUS_DATA'` です。テーブル名の末尾に数字を使用しないでください。テーブルが 1 つの照会で複数回使用される場合、テーブルの名前には 0 から 9 までの数字が付加されます。例えば、CUSTOM_VIEW0、CUSTOM_VIEW1 などのようになります。テーブル名の末尾に既に数字がある場合、Business Process Choreographer はその数字を削除します。これにより QueryUnknownTableException が発生します。
データベース名	データベースの関連テーブルまたはビューの名前。使用できるのは大文字のみです。
データベース・スキーマ	データベースの関連テーブルまたはビューのスキーマ。使用できるのは大文字のみです。データベース・スキーマは、Business Process Choreographer データベースのデータベース・スキーマとは異なっている必要があります。それでも、テーブルまたはビューには、Business Process Choreographer データベースにアクセスするために使用するのと同じ JDBC データ・ソースを使用してアクセス可能でなければなりません。
属性	補足照会テーブルの属性は、照会に使用できる情報を定義します。この属性は、関連データベース表またはビュー内の列の関連名と一致している必要があります。 属性は、名前とタイプで定義します。名前は大文字で定義します。属性のタイプは、以下のいずれかです。 • **ブール**: ブール値 • **10 進数**: 浮動小数点数 • **ID**: 長さが 16 バイトのオブジェクト ID (TASK 照会テーブルの TKIID など) • **番号**: 整数、短整数、または長整数 • **ストリング**: ストリング • **タイム・スタンプ**: タイム・スタンプ
結合	結合は、複合照会テーブル内で接続する場合は、補足照会テーブルで定義する必要があります。結合は、補足照会テーブルの情報と 1 次照会テーブルの情報を相関させるために使用する属性を定義します。結合を定義する場合、ソース属性とターゲット属性のタイプは同じでなければなりません。
許可	補足照会テーブルには許可が指定されないため、すべての認証済みユーザーがコンテンツを参照できます。

関連概念

338 ページの『定義済み照会テーブル』
定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

『複合照会テーブル』
Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

351 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder を使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

371 ページの『照会テーブルの照会』
照会は、照会テーブル API (Business Flow Manager EJB でのみ使用可能) を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。

360 ページの『照会テーブルの許可』
照会テーブルで照会を実行するときには、インスタンス・ベースの許可、ロール・ベースの許可、または許可なしを使用できます。

複合照会テーブル

Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

複合照会テーブルはクライアント開発者によって設計され、このテーブルでは、照会実行時のデータ・アクセスを最適化するために、フィルターおよび許可オプションを詳細に構成することができます。複合照会テーブルは、タスク・リストおよびプロセス・リストの照会に最適化された SQL で実現されています。

複合照会テーブルは、照会の実際の実装を抽象化することによって照会を最適化することができるため、Business Process Choreographer の標準照会 API の代わりに実動シナリオで使用することが推奨されています。

さらに、照会テーブルにアクセスするクライアントを再デプロイすることなく、実行時に複合照会テーブルを変更できます。

以下の図は、複合照会テーブルの内容の概要を示しています。

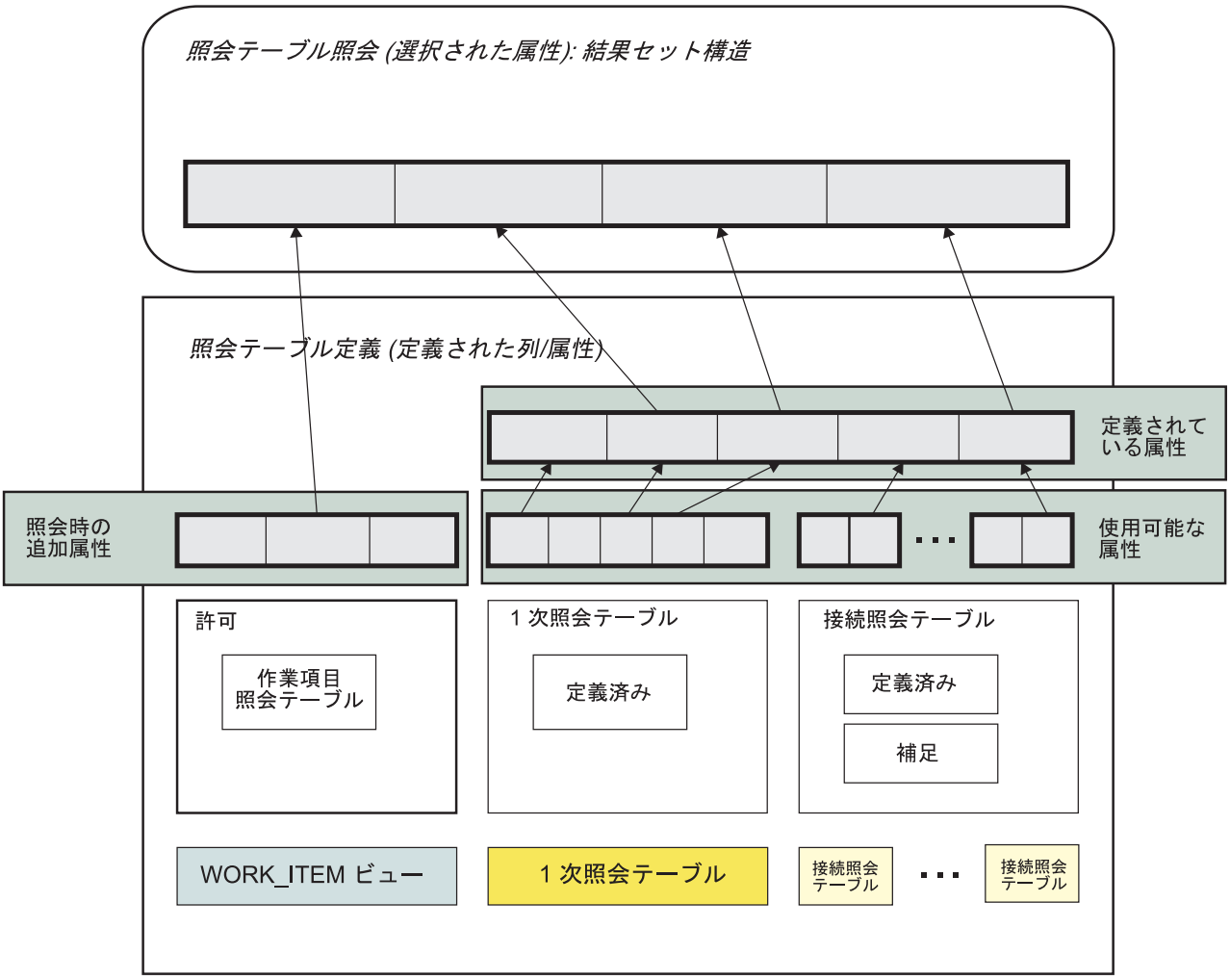


図 73. 複合照会テーブルの内容

すべての複合照会テーブルは、1 つの 1 次照会テーブルと 0 個以上の接続照会テーブルで定義されます。

1 次照会テーブル:

- 複合照会テーブルに格納される主な情報を構成します。
- いずれかの定義済み照会テーブルでなければなりません。
- 複合照会テーブル内の各オブジェクトを 1 次キーによって一意的に識別します。例えば、TASK 定義済み照会テーブルの場合、1 次キーはタスク ID の TKIID です。
- インスタンス・ベースの許可を使用する場合は、WORK_ITEM 照会テーブルに含まれている作業項目を使用して、照会テーブルの内容を許可します。
- 複合照会テーブルの照会時にテーブルの行として返されるオブジェクトのリストを判別します。

接続照会テーブル:

- 既にシステムにデプロイされている定義済み照会テーブルおよび補足照会テーブルにすることができます。
- 1 次照会テーブルで提供される情報以外の情報を提供するために使用できます。例えば、TASK が 1 次照会テーブルの場合、TASK_DESC 照会テーブルで提供されるタスクの説明を、複合照会テーブルの内容に追加することができます。

通常、1 次照会テーブルは、複合照会テーブルの目的に基づいて選択されます。

- 複合照会テーブルがタスク・リストを記述する場合、TASK 照会テーブルが 1 次照会テーブルになります。
- 複合照会テーブルがプロセス・リストを記述する場合、PROCESS_INSTANCE 照会テーブルが 1 次照会テーブルになります。
- アクティビティーのリストは、ACTIVITY 1 次照会テーブルを使用して取得されます。
- ヒューマン・タスク・エスカレーションのリストは、ESCALATION 1 次照会テーブルを使用して取得されます。

1 次照会テーブルと接続照会テーブルの関係

接続照会テーブルと 1 次照会テーブルは、1 対 1 または 1 対 0 の関係でなければなりません。1 対 1 または 1 対 0 の関係に違反すると、照会の実行時にランタイム例外が発生します。

1 次照会テーブルと接続照会テーブルは、接続照会テーブルで定義されている結合属性を使用して相関させます。この結合属性は、Business Process Choreographer のさまざまな照会テーブル内のデータの関係を記述しているため、定義済み照会テーブルでは変更できません。通常、1 対 1 または 1 対 0 の関係を維持する場合は、結合属性で十分です。例えば、CONTAINMENT_CTX_ID 属性は、TASK 照会テーブルで使用して、PROCESS_INSTANCE 照会テーブルの PIID 属性で識別される関連プロセス・インスタンス情報を接続します。

1 対多の関係が存在する場合は、照会テーブルを定義するときに Query Table Builder で追加の基準 (選択基準と呼ばれる) を指定する必要があります。例えば、`"LOCALE='en_US'"` のように指定します。タスクには、1 つのタスクのさまざまなロケールを使用して識別される複数の記述を用意することができます。

例 1:

以下の図は、接続照会テーブルで指定される選択基準を可視化したサンプルを示しています。

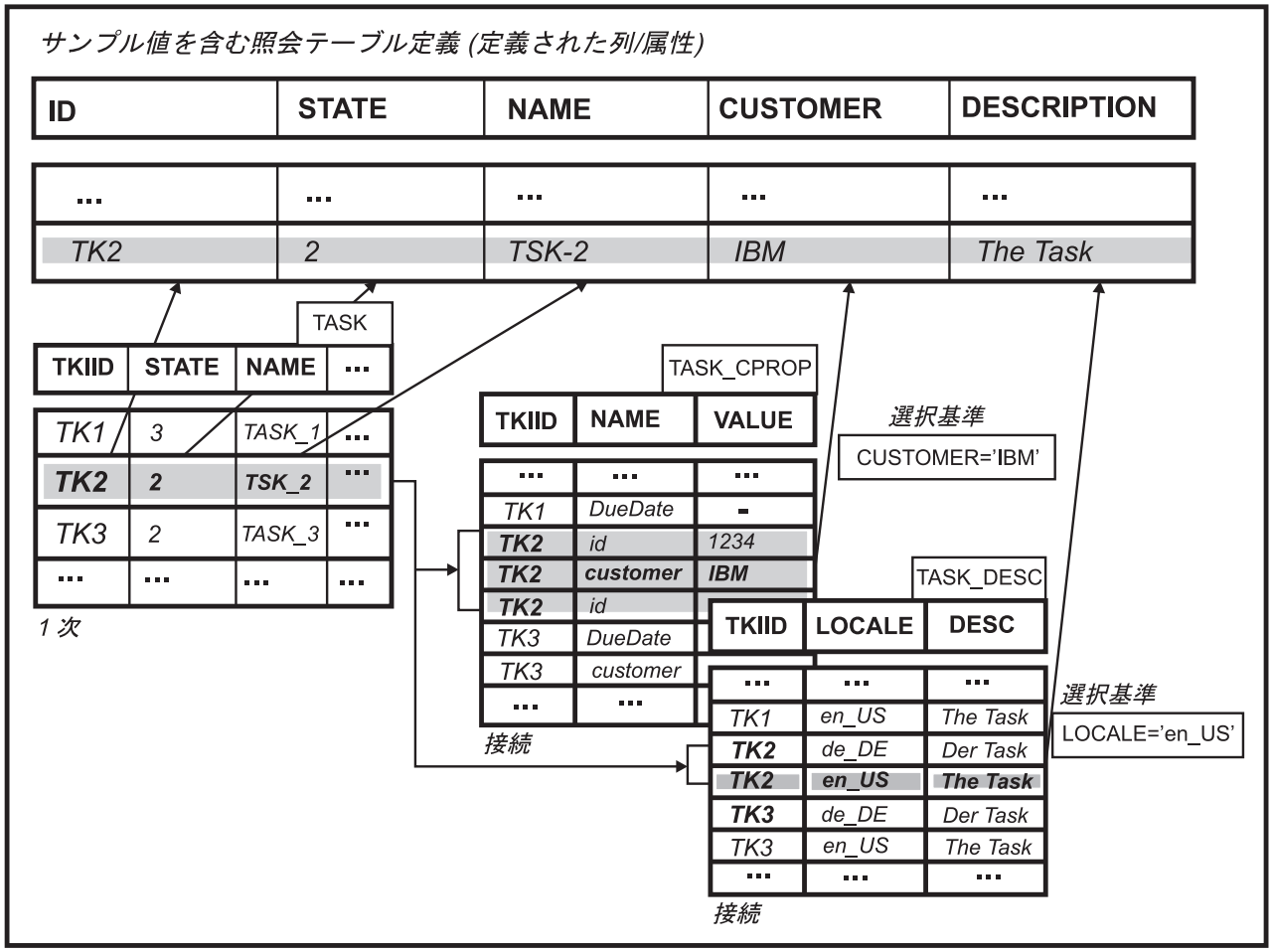


図 74. 選択基準が指定された複合照会テーブル

複合照会テーブルには、ID、STATE、NAME、CUSTOMER、および DESCRIPTION 属性が含まれています。

- ID、STATE、および NAME は、TASK 1 次照会テーブルによって提供されます。
- CUSTOMER は、TASK のカスタム・プロパティーです。カスタム・プロパティーは、TASK_CPROP 照会テーブルに格納されます。特定のタスクでは、カスタム・プロパティーは名前によって一意的に識別されます。例えば、選択基準 `"CUSTOMER='IBM'"` などがあります。
- DESCRIPTION は、TASK_DESC 照会テーブルに格納されるタスクの説明です。タスク・インスタンスごとに、そのタスクの説明がロケールによって一意的に識別されます。例えば、選択基準 `"LOCALE='en_US'"` などがあります。

例 2:

この例では、1 次照会テーブルと接続照会テーブルの関係に焦点が当てられています。1 次照会テーブルとして TASK、接続照会テーブルとして TASK_DESC が使用されています。複合照会テーブルを定義するときに、TASK_DESC 照会テーブルの LOCALE 属性を指定して、1 次照会テーブルと接続照会テーブルが 1 対 1 または

1 対 0 の関係になるようにする必要があります。この表は、接続照会テーブル TASK_DESC の有効な選択基準が指定された複合照会テーブルの内容のサンプルを示しています。

表 29. 複合照会テーブルの有効な内容

1 次照会テーブル TASK の情報	**接続照会テーブル TASK_DESC の情報**	
NAME	LOCALE	DESCRIPTION
task_one	en_US	これは説明です。
task_two	en_US	これは説明です。
...	...	...

以下の表は、選択基準の設定が間違っている (1 対 1 または 1 対 0 の関係に違反している) 場合の仮想の無効な内容 (**太字**) を示しています。

表 30. 複合照会テーブルの無効な内容

TASK (1 次照会テーブル) から取得される情報	**TASK_DESC (接続照会テーブル) から取得される情報**	
NAME	LOCALE	DESCRIPTION
task_one	en_US	これは説明です。
task_one	**de_DE**	**Das ist eine Beschreibung.**
...	...	...

プロパティー

複合照会テーブルには、以下のプロパティーがあります。

表 31. 複合照会テーブルのプロパティー

プロパティー	**説明**
名前	照会テーブル名は、Business Process Choreographer インストール済み環境内で固有でなければなりません。この照会テーブル名は、照会の実行時に、照会される照会テーブルを識別するために使用されます。 照会テーブルは、名前によって一意的に識別されます (複合照会テーブルの場合は *prefix.name* と定義されます)。*prefix.name* の最大長は 28 文字です。接頭部は、予約済みの接頭部'IBM' と異なっていなければなりません。例えば、'COMPANY.TODO_TASK_LIST' です。テーブル名の末尾に数字を使用しないでください。テーブルが 1 つの照会で複数回使用される場合、テーブルの名前には 0 から 9 までの数字が付加されます。例えば、CUSTOM_VIEW0、CUSTOM_VIEW1 などのようになります。テーブル名の末尾に既に数字がある場合、Business Process Choreographer はその数字を削除します。これにより QueryUnknownTableException が発生します。

表 31. 複合照会テーブルのプロパティー (続き)

プロパティー	説明
属性	複合照会テーブルの属性は、照会に使用できる情報を定義します。 属性は、大文字の名前で定義します。タイプは参照先の属性から継承され、以下のいずれかになります。 • **ブール**: ブール値 • **10 進数**: 浮動小数点数 • **ID**: オブジェクト ID (TASK 照会テーブルの TKIID など) • **番号**: 整数、短整数、または長整数 • **ストリング**: ストリング • **タイム・スタンプ**: タイム・スタンプ 複合照会テーブルの属性は、 1 次照会テーブルまたは接続照会テーブルの属性への参照を使用して定義されます。複合照会テーブルの属性は、参照先の属性のタイプおよび定数を継承します。 照会テーブル定義の一部である属性に加えて、作業項目情報を実行時に照会することができます。これは、1 次照会テーブルがインスタンス・データを含む場合 (TASK や PROCESS_INSTANCE など)、および複合照会テーブルでインスタンス・ベースの許可を使用する場合に可能です。例えば、ユーザーが潜在的所有者であるヒューマン・タスクのみを返すように照会を定義することができます。

表 31. 複合照会テーブルのプロパティー (続き)

プロパティー	説明
許可	それぞれの複合照会テーブルは、照会を実行するときにインスタンス・ベースの許可、ロール・ベースの許可、許可なしのいずれを使用するかを定義します。 インスタンス・ベースの許可を定義する場合は、照会を実行するユーザーの作業項目が含まれているオブジェクトのみが返されます。ただし、AdminAuthorizationOptions を使用すると、任意のユーザーの作業項目が存在するかどうかが検査されます。これらの照会では、ユーザーは BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があり、AdminAuthorizationOptions が照会テーブル API に渡される必要があります。 ロール・ベースの許可が定義されている場合、これらの照会では、ユーザーは BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があり、AdminAuthorizationOptions が照会テーブル API に渡される必要があります。 許可を定義しない場合は、関連オブジェクトの作業項目が照会テーブルに存在するかどうかが検査されずに、照会が実行されます。すべての認証済みユーザーが照会テーブルの内容を参照できます。 インスタンス・ベースの許可は、1 次照会テーブルがインスタンス・データを含む場合に定義できます。ロール・ベースの許可は、1 次照会テーブルがテンプレート・データを含む場合に定義できます。いずれの 1 次照会テーブルを使用するかにかかわらず、複合照会テーブルで許可なしを定義することはできません。

フィルター

フィルターを使用して、複合照会テーブルに含めるオブジェクトまたは行の数を制限します。

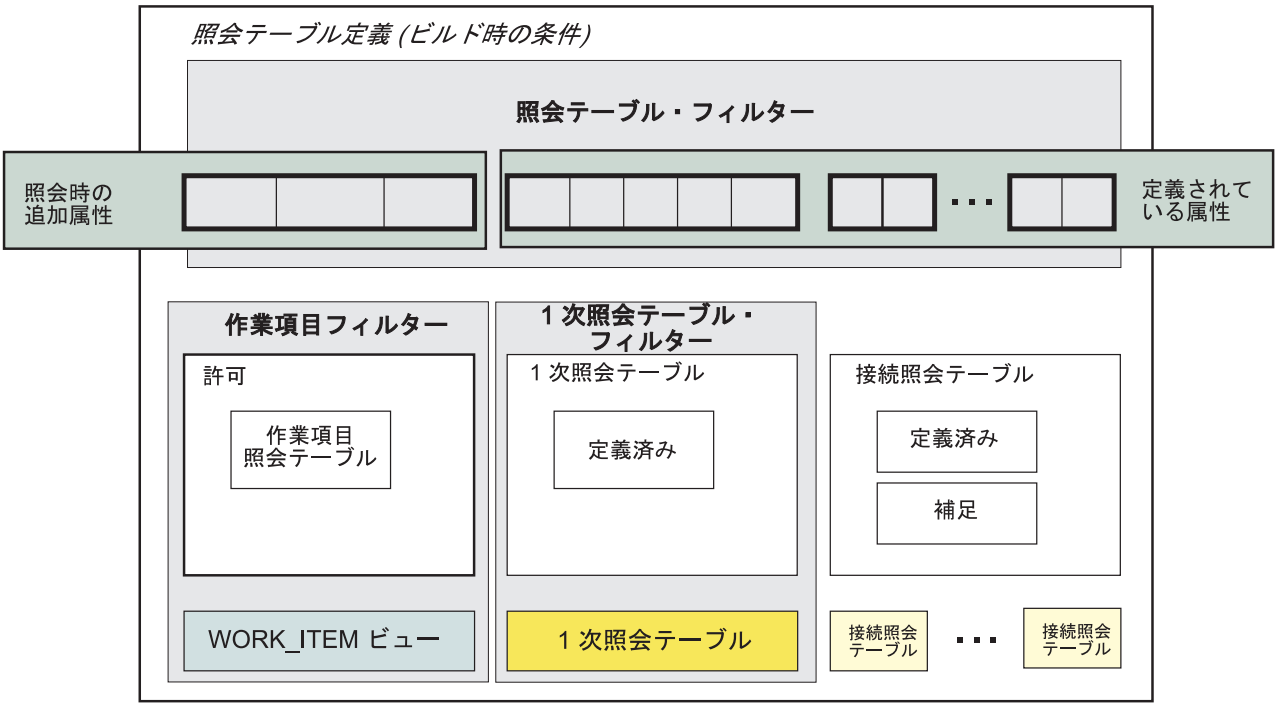


図 75. 複合照会テーブルのフィルター

複合照会テーブルのフィルターは、以下の開発時に定義できます。

- 1 次照会テーブル (1 次照会テーブル・フィルターとして)。
- 暗黙的に使用可能な WORK_ITEM 照会テーブル。これは、1 次照会テーブルにインスタンス・データが含まれている場合に、許可を実行します。このフィルターは、許可フィルターと呼ばれ、インスタンス・ベースの許可を使用するように複合照会テーブルが構成されている場合にのみ使用可能です。
- 複合照会テーブル (照会テーブル・フィルターとして)。

フィルターは、照会テーブルの開発時に定義されます。例えば、1 次照会テーブル TASK が含まれている複合照会テーブルは、作動可能状態 (1 次照会テーブル・フィルターとして "STATE=STATE_READY" が指定されている) のタスクに対してフィルター処理を行うことができます。

許可

1 次照会テーブルが含まれている複合照会テーブルの内容にアクセスするための許可は、1 次照会テーブルにアクセスするために使用される許可と似ています。複合照会テーブルは、より多くの制限を設けて構成できる点が異なります。

- インスタンス・ベースの許可を使用するように構成する場合は、複合照会テーブルに含まれているデータを対象として、WORK_ITEM 照会テーブルに既存の作業項目が存在するかどうかが検査されます。この検査は、1 次照会テーブルに対して行われます。複合照会テーブルの構成に応じて、全員、個人、グループ、および継承された作業項目が検査で使用されます。継承された作業項目を指定する場合は、関連する全員、個人、またはグループの作業項目が構成されている、プロ

セス・インスタンスを親として持つオブジェクト (参加ヒューマン・タスクなど) が複合照会テーブルに入れられます。通常、継承された作業項目は管理者の場合にのみ有用です。

- テンプレート・データを含む 1 次照会テーブルが含まれている複合照会テーブルは、インスタンス・ベースの許可を使用するように設定してはなりません。ロール・ベースの許可が使用されている場合、照会を実行できるのは、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っているユーザーのみであり、AdminAuthorizationOptions オブジェクトを使用する必要があります。

関連概念

338 ページの『定義済み照会テーブル』
定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

341 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

照会テーブルの作成

Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder を使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

Query Table Builder は Eclipse プラグインとして使用可能であり、WebSphere Business Process Management SupportPacs のサイトからダウンロードできます。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

照会テーブルは、アプリケーションを開発およびデプロイする方法に影響を与えます。以下のステップでは、照会テーブルを使用する Business Process Choreographer アプリケーションを設計および開発する場合に関係するロールについて説明します。

表 32. 照会テーブルの作成ステップ

ステップ	担当者	説明
1. 分析	ビジネス・アナリスト、クライアント開発者	クライアント・アプリケーションで必要な照会テーブルを分析します。考慮すべき質問は、以下のとおりです。 • どれほどの数のタスク・リストまたはプロセス・リストをユーザーに提供しますか? 同じ照会テーブルを共用できるタスク・リストまたはプロセス・リストはありますか? • 使用する許可の種類は何ですか? インスタンス・ベースの許可、ロール・ベースの許可、または許可を使用しませんか? • 再利用可能なその他の照会テーブルが既にシステムで定義されていますか? • 照会テーブルで複数の言語のコンテンツを提供する必要がありますか? その場合、接続照会テーブルの選択基準は LOCALE=$LOCALE でなければなりません。
2. 照会テーブルの作成	クライアント開発者、ビジネス・アナリスト	クライアント・アプリケーションで使用する照会テーブルを作成します。照会テーブルの照会で最善のパフォーマンスを引き出せるように、照会テーブルの定義を指定します。
3. 照会テーブルのデプロイメント	管理者	照会テーブルを使用するには、事前に照会テーブルをランタイムにデプロイしておく必要があります。このステップは、manageQueryTable.py wsadmin コマンドを使用して実行します。
4. 照会テーブルの照会	クライアント開発者	照会テーブル作成の最後のステップとして、照会テーブルに対して照会を実行します。クライアント開発者は、照会テーブルとその属性の名前を知っている必要があります。

照会テーブル API を使用して照会テーブルを照会するコードの例を以下に示します。簡単にするため、例 1 および 2 では定義済み照会テーブル TASK を照会します。例 3 および 4 では、複合照会テーブルを照会します。ここで、この複合照会テーブルはシステムにデプロイされているとします。アプリケーションの開発時には、定義済み照会テーブルを直接照会するのではなく、複合照会テーブルを使用してください。

例 1

```
// get the naming context and lookup the Business
// Flow Manager Enterprise JavaBeans home; note that the Business Flow
// Manager Enterprise JavaBeans home should be cached for performance
// reasons; also, it is assumed that there's an Enterprise JavaBeans
// reference to the local business flow manager Enterprise JavaBeans
Context ctx = new InitialContext();
LocalBusinessFlowManagerHome home =
 (LocalBusinessFlowManagerHome)
 ctx.lookup("java:comp/env/ejb/BFM");
```

```
// if the human task manager Enterprise JavaBeans is used, do:
// LocalHumanTaskManagerHome home =
//   (LocalHumanTaskManagerHome) ctx.lookup("java:comp/env/ejb/HTM");
// assuming that a EJB reference to the human task manager EJB
// has been defined

// create the business flow manager client-side stub
LocalBusinessFlowManager bfm = home.create();
// if the human task manager EJB is used, do:
// LocalHumanTaskManager htm = home.create();
// note that the human task manager Enterprise JavaBeans provides the
// same methods as the business flow manager Enterprise JavaBeans
// ************************************************
// ****************** example 1 ******************
// ************************************************

// execute a query against the TASK predefined query
// table; this relates to a simple My ToDo's task list
EntityResultSet ers = null;
ers = bfm.queryEntities("TASK", null, null, null);

// print the result to STDOUT
EntityInfo entityInfo = ers.getEntityInfo();
List attList = entityInfo.getAttributeInfo();
int attSize = attList.size();

Iterator iter = ers.getEntities().iterator();
while (iter.hasNext()) {
 System.out.print("Entity: ");
 Entity entity = (Entity) iter.next();
 for (int i = attSize - 1; i >= 0; i--) {
  AttributeInfo ai = (AttributeInfo) attList.get(i);
  System.out.print(
    entity.getAttributeValue(ai.getName()));
 }
 System.out.println();
}
```

例 2

```
// ************************************************
// ****************** example 2 ******************
// ************************************************

// same example as example 1, but using the row-based
// query approach
RowResultSet rrs = null;
rrs = bfm.queryRows("TASK", null, null, null);

attList = rrs.getAttributeInfo();
attSize = attList.size();

// print the result to STDOUT
while (rrs.next()) {
 System.out.print("Row: ");
 for (int i = attSize - 1; i >= 0; i--) {
  AttributeInfo ai = (AttributeInfo) attList.get(i);
  System.out.print(
    rrs.getAttributeValue(ai.getName()));
 }
 System.out.println();
}
```

例 3

```
// ************************************************
// ****************** example 3 *******************
// ************************************************

// execute a query against a composite query table
// that has been deployed on the system before;
// the name is assumed to be COMPANY.TASK_LIST
ers = bfm.queryEntities(
  "COMPANY.TASK_LIST", null, null, null);
^
//  print the result to STDOUT ...
```

例 4

```
// ************************************************
// ****************** example 4 *******************
// ************************************************

// query against the same query table as in example 3,
// but with customized options
FilterOptions fo = new FilterOptions();

// return only objects which are in state ready
fo.setQueryCondition("STATE=STATE_READY");

// sort by the id of the object
fo.setSortAttributes("ID");

// limit the number of entities to 50
fo.setThreshold(50);

// only get a sub-set of the defined attributes
// on the query table
fo.setSelectedAttributes("ID, STATE, DESCRIPTION");

AuthorizationOptions ao = new AuthorizationOptions();

// do not return objects that everybody is allowed
// to see
ao.setEverybodyUsed(Boolean.FALSE);

ers = bfm.queryEntities(
  "COMPANY.TASK_LIST", fo, ao, null);

//  print the result to STDOUT ...
```

関連概念

371 ページの『照会テーブルの照会』
照会は、照会テーブル API (Business Flow Manager EJB でのみ使用可能) を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。

『照会テーブルのフィルターおよび選択基準』
フィルターおよび選択基準は、Query Table Builder を使用した照会テーブルの開発時に定義します (SQL WHERE 文節に似た構文を使用します)。これらの明確に定義されたフィルターおよび選択基準を使用して、照会テーブルの属性に基づく条件を指定します。

338 ページの『定義済み照会テーブル』
定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

341 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

343 ページの『複合照会テーブル』
Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

照会テーブルのフィルターおよび選択基準

フィルターおよび選択基準は、Query Table Builder を使用した照会テーブルの開発時に定義します (SQL WHERE 文節に似た構文を使用します)。これらの明確に定義されたフィルターおよび選択基準を使用して、照会テーブルの属性に基づく条件を指定します。

Query Table Builder のインストール方法について詳しくは、WebSphere Business Process Management SupportPacs のサイトを参照してください。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

属性

式で使用される属性は、照会テーブルの属性を参照します。使用可能な属性は、式を使用する場所によって異なります。クライアント開発者は、照会テーブル API に渡される照会フィルターでのみ式を使用できます。複合照会テーブルの開発者は、他のさまざまな場所で式を使用できます。以下の表は、さまざまな場所で使用可能な属性を示しています。

表 33. 照会テーブルの式で使用できる属性

ここで、	式	使用可能な属性
照会テーブル API	照会フィルター	• 照会テーブルに定義されているすべての属性。 • インスタンス・ベースの許可を使用する場合、WORK_ITEM 照会テーブルに定義されているすべての属性 (接頭部は 'WI.')。 例: • STATE=STATE_READY (照会テーブルに STATE 属性が含まれていて、この属性に STATE_READY 定数が定義されている場合)。 • STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER (照会テーブルに STATE 属性が含まれていて、照会テーブルでインスタンス・ベースの許可を使用する場合)
複合照会テーブル	照会テーブル・フィルター	
	1 次照会テーブル・フィルター	• 1 次照会テーブルに定義されているすべての属性。 例: • STATE=STATE_READY (照会テーブルに STATE 属性が含まれていて、この属性に STATE_READY 定数が定義されている場合)。
	許可フィルター	• WORK_ITEM 定義済み照会テーブルに定義されているすべての属性 (接頭部は 'WI.')。 例: • WI.REASON=REASON_POTENTIAL_OWNER
	選択基準	• 関連接続照会テーブルに定義されているすべての属性。 例: • LOCALE='en_US' (TASK_DESC 照会テーブルなどの接続照会テーブルに LOCALE 属性が含まれている場合)。

以下の図は、式で使用するフィルターのさまざまな場所および選択基準と、例を示しています。

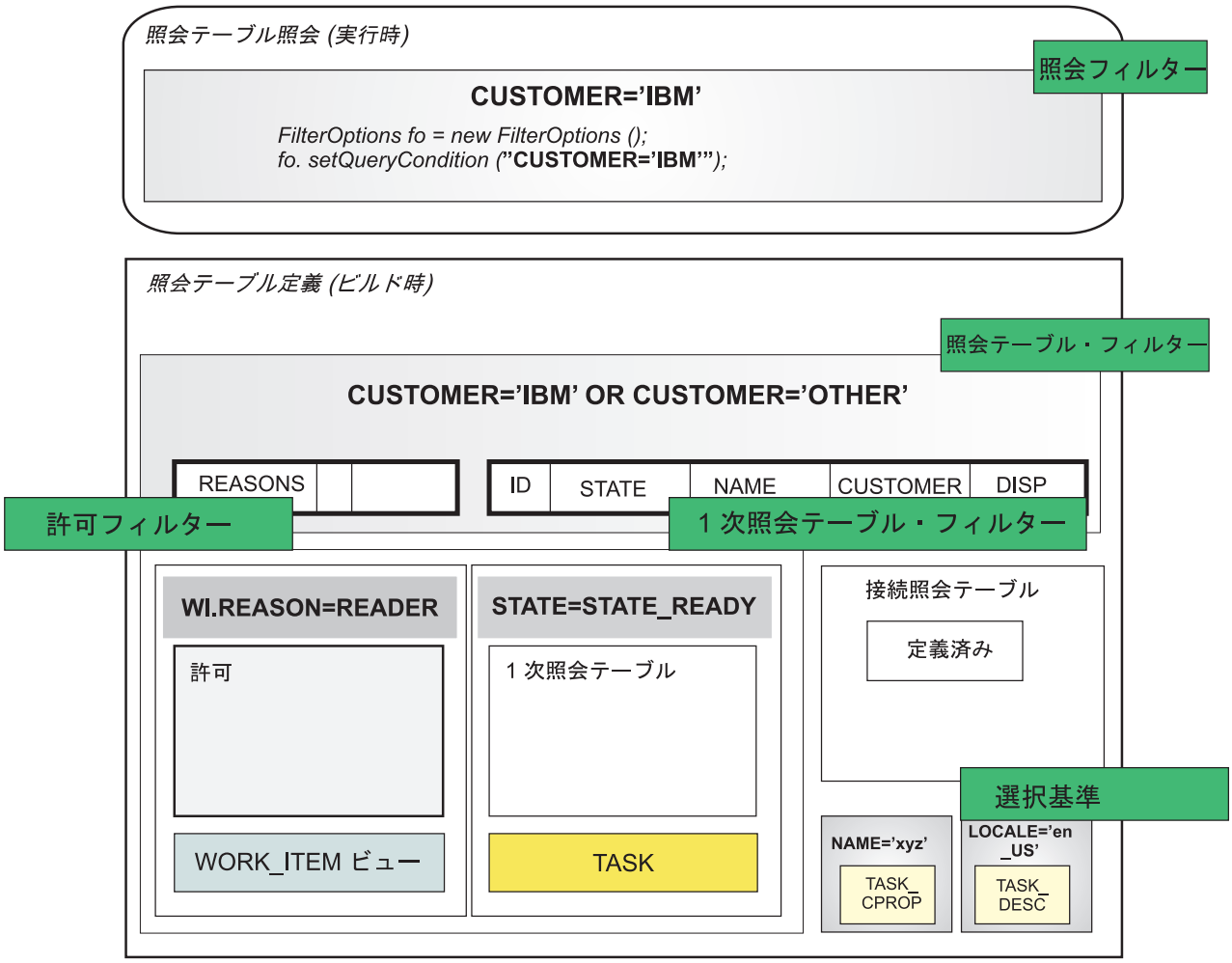


図 76. 式におけるフィルターおよび選択基準

式

式には以下の構文があります。

```
expression  ::=  attribute binary_op value |
                 attribute unary_op |
                 attribute list_op list |
                 (expression) |
                 expression AND expression |
                 expression> OR expression
```

以下のルールが適用されます。

- AND は、OR に優先します。副次式は、AND および OR を使用して結合されます。
- 括弧は式をグループ化するために使用でき、対で使用する必要があります。

例:

- `STATE = STATE_READY`
- `NAME IS NOT NULL`
- `STATE IN (2, 5, STATE_FINISHED)`
- `((PRIORITY=1) OR (WI.REASON=2)) AND (STATE=2)`

式は特定のスコープ内で実行されます。このスコープによって、式で使用できる有効な属性が決まります。選択基準 (照会フィルター) は、照会を実行する照会テーブルのスコープ内で実行されます。

以下は、定義済み TASK 照会テーブルに対して実行される照会の例です。

```
'(STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER)
OR (WI.REASON=REASON_OWNER)'
```

2 項演算子

以下の 2 項演算子が使用可能です。

binary_op ::= = | < | > | <> | <= | >= | LIKE | NOT LIKE

以下のルールが適用されます。

- 2 項演算子の左側のオペランドは、照会テーブルの属性を参照している必要があります。
- 2 項演算子の右側のオペランドは、リテラル値、定数、またはパラメーターでなければなりません。
- LIKE および NOT LIKE 演算子は、属性タイプが STRING の属性でのみ有効です。
- 左側のオペランドと右側のオペランドは、互換性のある属性タイプでなければなりません。
- ユーザー・パラメーターは、左側の属性の属性タイプとの互換性がなければなりません。

例:

- STATE > 2
- NAME LIKE 'start%'
- STATE <> PARAM(theState)

単項演算子

以下の単項演算子が使用可能です。

unary_op ::= IS NULL | IS NOT NULL

以下のルールが適用されます。

- 単項演算子の左側のオペランドは、照会テーブルの属性を参照している必要があります。有効な属性は、フィルターの場所や選択基準によって異なります。
- ヌル値がないかどうかをすべての属性で確認できます。例えば、CUSTOMER IS NOT NULL と指定します。

例:

```
DESCRIPTION IS NOT NULL
```

リスト演算子

以下のリスト演算子が使用可能です。

list_op ::= IN | NOT IN

以下のルールが適用されます。

- リスト演算子の右側は、ユーザー・パラメーターで置き換えることはできません。
- ユーザー・パラメーターは、右側のオペランドのリスト内で使用できます。

例:

```
STATE IN (STATE_READY, STATE_RUNNING, PARAM(st), 1)
```

リストは、以下のように表現します。

list ::= *value* [, *list*]

以下のルールが適用されます。

- リスト演算子の右側は、ユーザー・パラメーターで置き換えることはできません。
- ユーザー・パラメーターは、右側のオペランドのリスト内で使用できます。

例:

- (2, 5, 8)
- (STATE_READY, STATE_CLAIMED)

値

式で使用する値は、以下のいずれかです。

- **定数**: 定数値。定義済み照会テーブルの属性用に定義されます。例えば、STATE_READY は、TASK 照会テーブルの STATE 属性用に定義されています。
- **リテラル**: ハードコーディングされる任意の値。
- **パラメーター**: パラメーターは、特定の値を使用した照会の実行時に置換されます。

定数 は、定義済み照会テーブルの一部の属性で使用できます。定義済み照会テーブルの属性で使用可能な定数については、定義済みビューの情報を参照してください。照会テーブルでは、整数値を定義する定数のみが公開されます。定数の代わりに、関連リテラル値またはパラメーターを使用することもできます。

例:

- TASK 照会テーブルの STATE 属性の STATE_READY をフィルターで使用して、タスクが作動可能状態になっているかどうかを確認できます。
- WORK_ITEM 照会テーブルの REASON 属性の REASON_POTENTIAL_OWNER をフィルターで使用して、照会テーブルに対して照会を実行するユーザーが潜在的所有者であるかどうかを確認できます。
- TASK 照会テーブルに対して照会を実行する場合、照会フィルター STATE=STATE_READY は STATE=2 と同じです。

式では**リテラル**を使用することもできます。タイム・スタンプおよび ID には、特殊な構文を使用する必要があります。

例:

- STATE=1
- NAME='theName'

- `CREATED > TS ('2008-11-26 T12:00:00')`
- `TKTID=ID('_TKT:801a011e.9d57c52.ab886df6.1fcc0000')`

式で使用する**パラメーター**は、複合照会テーブルの動的性に対応しています。ユーザー・パラメーターとシステム・パラメーターがあります。

- ユーザー・パラメーターは、PARAM (*name*) を使用して指定します。このパラメーターは、照会の実行時に指定する必要があります。com.ibm.bpe.api.Parameter クラスのインスタンスとして、照会テーブル API に渡されます。
- システム・パラメーターは、照会の実行時に指定しなくても、照会テーブル・ランタイムによって指定されるパラメーターです。システム・パラメーター `$USER` および `$LOCALE` が使用可能です。
  - `$USER` (ストリング) には、照会を実行するユーザーの値が入ります。
  - `$LOCALE` (ストリング) には、照会の実行時に使用されるロケールの値が入ります。`$LOCALE` の値は、例えば `'en_US'` などです。

特定のロケールを選択する接続照会テーブルの選択基準にパラメーターを指定できます。例えば、複合照会テーブルの 1 次照会テーブルが TASK で、接続照会テーブルが TASK_DESC の場合などです。パラメーターの例を以下に示します。

- `STATE=PARAM(theState)`
- `LOCALE=$LOCALE`
- `OWNER=$USER`

関連概念

351 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder を使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

371 ページの『照会テーブルの照会』
照会は、照会テーブル API (Business Flow Manager EJB でのみ使用可能) を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。

関連タスク

392 ページの『Business Process Choreographer Explorer の照会テーブルの作成』
EJB 照会 API の代わりに照会テーブルを使用して、Business Process Choreographer Explorer のパフォーマンスを向上させることができます。照会テーブルを作成するには、Query Table Builder を使用します。

照会テーブルの許可

照会テーブルで照会を実行するときには、インスタンス・ベースの許可、ロール・ベースの許可、または許可なしを使用できます。

許可タイプは照会テーブルで定義されています。

- インスタンス・ベースの許可は、作業項目を使用して照会テーブルのオブジェクトを許可することを示します。これを行うには、適切な作業項目が存在するかどうかを確認してください。

- ロール・ベースの許可は、Java EE ロールに基づいています。これは、API 照会メソッドを使用する呼び出し元が、照会テーブルの内容を参照するには、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があることを示します。これは、テンプレート・データを含む定義済み照会テーブルおよびテンプレート・データを含む 1 次照会テーブルが含まれている複合照会テーブルの場合に使用可能です。これらの照会テーブルのオブジェクトには、関連作業項目がありません。
- 許可を指定していない場合は、フィルターを適用した後、すべての認証済みユーザーが照会テーブルのすべての内容を参照できます。

定義済み照会テーブルでの許可のタイプおよび複合照会テーブルと補足照会テーブルで構成可能な許可のタイプの概要を以下の表に示します。

表 34. 照会テーブルに応じた許可のタイプ

照会テーブル	インスタンス・ベースの許可	ロール・ベースの許可	許可なし
定義済み	インスタンス・データを含む定義済み照会テーブルの場合に必要です。	テンプレート・データを含む定義済み照会テーブルの場合に必要です。	なし
複合	オフにすることができます。オフにすると、許可は使用されず、セキュリティー制約がオーバーライドされます。つまり、すべての認証済みユーザーは、それぞれのオブジェクトに対する権限を持っているかどうかに関係なく、照会テーブルを使用してデータを検索することができます。 テンプレート・データを含む 1 次照会テーブルが含まれている複合照会テーブルは、インスタンス・ベースの許可を使用するように設定してはなりません。	テンプレート・データを含む 1 次照会テーブルが含まれている複合照会テーブルの場合などは、オフにすることができます。その場合、許可は使用されず、セキュリティー制約がオーバーライドされます。つまり、すべての認証済みユーザーは、それぞれのオブジェクトに対する権限を持っているかどうかに関係なく、照会テーブルを使用してデータを検索することができます。 インスタンス・データを含む 1 次照会テーブルが含まれている複合照会テーブルは、ロール・ベースの許可を使用するように設定してはなりません。	フィルターを適用した後、すべての認証済みユーザーが照会テーブルのすべての内容を参照できます。
補足照会テーブル	補足照会テーブルは Business Process Choreographer の管理対象ではないため、インスタンス・ベースの許可を使用するように補足照会テーブルを設定するべきではありません。したがって、インスタンス・ベースの許可には、これらのテーブル内容の許可情報は含まれません。	補足照会テーブルは、ロール・ベースの許可を使用するように設定してはなりません。	フィルターを適用した後、すべての認証済みユーザーが照会テーブルのすべての内容を参照できます。

照会テーブルのタイプに応じて使用可能な許可タイプのオプションの概要を以下の図に示します。また、さまざまな動作、照会テーブル API、およびその API の許可

オプションも示しています。

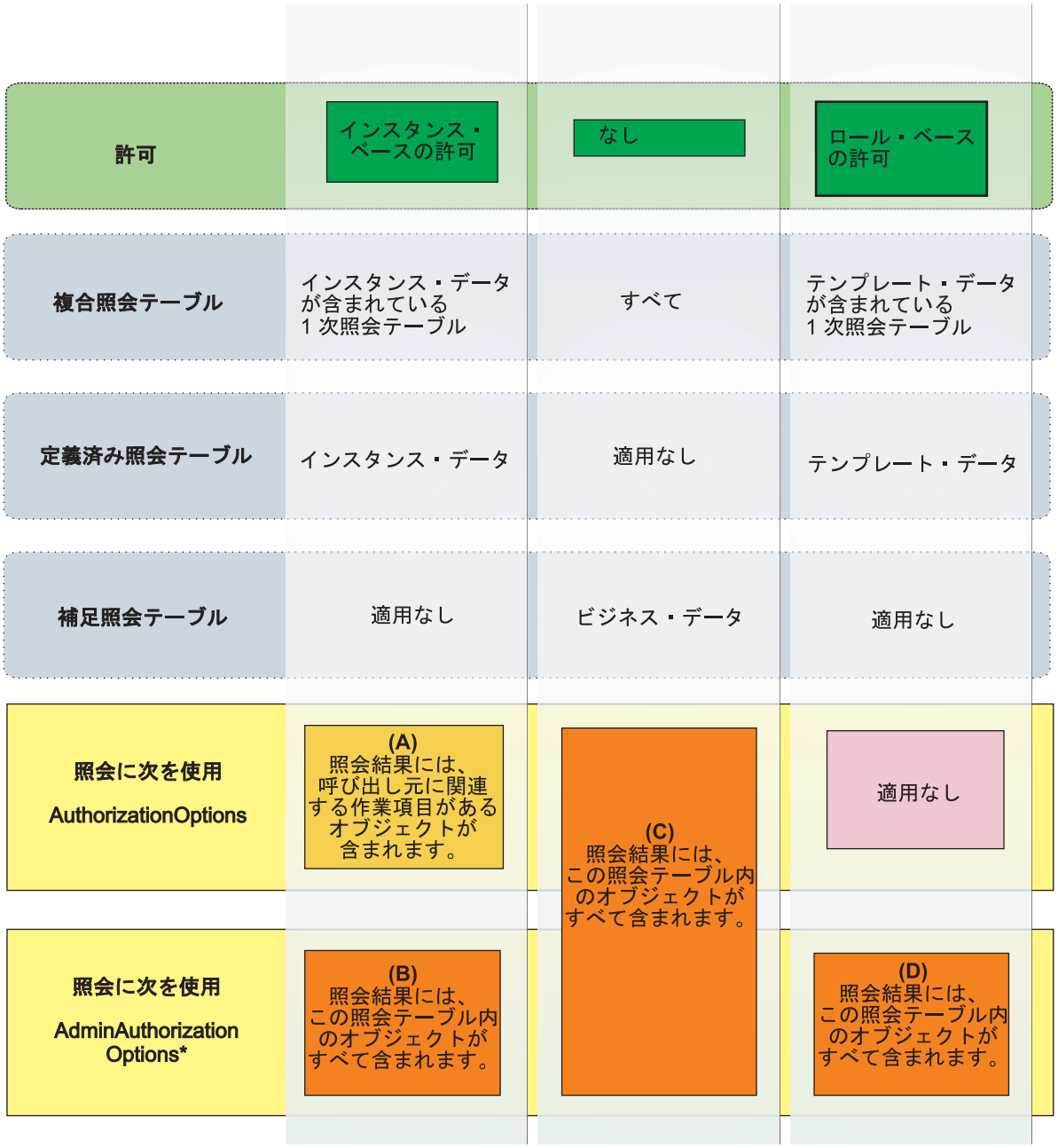


図 77. 照会テーブルのインスタンス・ベースの許可

*) onBehalfUser を設定する場合、(A) が適用されます

作業項目を使用する照会結果内のオブジェクトのインスタンス・ベースの許可は、照会テーブル API に渡される許可パラメーターや、照会テーブルのインスタンス・ベースの許可フラグの設定によって決まります。

- (A) AuthorizationOptions オブジェクトを使用する定義済み照会テーブルまたは複合照会テーブルに対する照会では、この特定ユーザーの関連作業項目と相関するエンティティーが返されます。 AdminAuthorizationOptions オブジェクトを使用し、onBehalfUser を設定する場合も同様です。通常、タスク・リストまたはプロ

セス・リストがユーザーに提供される標準クライアントでは、照会テーブルと照会テーブル API パラメーターのこの組み合わせが使用されます。

- (B) 照会テーブルの内容全体は、照会テーブルのインスタンス・ベースの許可を使用して構成されるように、関連作業項目を持つエンティティーで構成されます。インスタンス・ベースの許可では、4 つの作業項目 (全員、個人、グループ、および継承) が考慮されます。 API 照会メソッドを使用する呼び出し元は、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があります。照会テーブルと照会テーブル API パラメーターのこの組み合わせは、使用可能なタスクまたはプロセスの完全なリストが表示または検索される管理シナリオ向けです。
- (C) AdminAuthorizationOptions または AuthorizationOptions が照会テーブル API に渡される場合、インスタンス・ベースまたはロール・ベースの許可を使用しない照会テーブルに対する照会では、同じ結果が返されます。これは、補足照会テーブルおよび複合照会テーブルに当てはまります。作業項目または Java EE ロールに対する検査は行われないため、すべての認証済みユーザーに対して内容全体が表示されます。Business Process Choreographer で提供されるインスタンス・ベースまたはロール・ベースの許可の制約を適用することによってオブジェクトの可視性を制限しないクライアントでは、照会テーブル定義の開発時に許可検査をオフにすることができます。ただし、要求と実行を使用する場合は、ユーザーは関連作業項目を持っている必要があります。
- (D) ロール・ベースの許可が構成されている定義済み照会テーブルまたは複合照会テーブルのテンプレート・データには、ロール・ベースの許可を使用しなければアクセスできません。そのため、API 照会メソッドを使用する呼び出し元は、BPESystemAdministrator Java EE ロール (Business Flow Manager EJB を使用している場合) または TaskSystemAdministrator Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があります。照会 API ではなく、照会テーブル API を使用して、テンプレート情報にアクセスできます。

作業項目およびインスタンス・ベースの許可

Business Process Choreographer で提供されるインスタンス・ベースの許可は、作業項目に基づいています。各作業項目では、誰が、どのオブジェクトに対して、どの権限を持っているかが記述されます。インスタンス・ベースの許可を使用する場合、WORK_ITEM 照会テーブルを使用して、この情報にアクセスします。

以下の表は、照会テーブルに対する照会の実行時にインスタンス・ベースの許可を使用する場合に考慮されるさまざまなタイプの作業項目について説明しています。

表 35. 作業項目タイプ

作業項目タイプ	説明
everybody	すべてのユーザーが特定のオブジェクト (タスクまたはプロセス・インスタンスなど) にアクセスできます。この場合、関連作業項目の EVERYBODY 属性は TRUE に設定されます。

表 35. 作業項目タイプ (続き)

作業項目タイプ	説明
individual	特定のユーザーのために作成される作業項目です。関連作業項目の OWNER_ID 属性は、特定のユーザーに設定されます。1 つのオブジェクト (タスクなど) で、OWNER_ID 属性が異なる複数の作業項目が存在する場合があります。
group	特定のグループのユーザーのために作成される作業項目です。関連作業項目の GROUP_NAME 属性は、特定のグループに設定されます。
inherited	プロセス・インスタンスのリーダーおよび管理者も、これらのプロセス・インスタンスに属するヒューマン・タスクへのアクセスをエスカレーションも含めて継承することができます。タスク照会内に継承された作業項目があるかどうかの検査は、実行時に複雑な SQL 結合と共に実行されるため、パフォーマンスに影響が及びます。

作業項目は、さまざまな状況で Business Process Choreographer によって作成されます。例えば、関連した担当者割り当て基準が指定されている場合は、タスク作成時に、さまざまなロール (リーダーや潜在的所有者など) の作業項目が作成されます。

以下の表は、照会テーブルに対する照会の実行時にインスタンス・ベースの許可を使用する場合に、定義済みの担当者割り当て基準に従って作成される作業項目のタイプを示しています。継承された作業項目は、プロセス・アプリケーションの開発時に明示的にモデル化されていない関係を反映するため、表に表示されていません。

表 36. 作業項目および担当者割り当て基準

作業項目タイプ	関連した担当者割り当て基準
everybody	全員
individual	*Nobody*、*Everybody*、および *Group* verb 以外のすべての担当者割り当て基準
group	グループ

複合照会テーブルの許可フィルター

インスタンス・ベースの許可を使用する場合、複合照会テーブルに対して許可フィルターを指定することができます。このフィルターは、許可に使用される作業項目を作業項目の特定の属性に基づいて制限します。例えば、TASK 1 次照会テーブルを含む複合照会テーブルに対する許可フィルター`"WI.REASON=REASON_POTENTIAL_OWNER"` は、ユーザーが照会を実行するときに返されるタスクを制限します。結果には、そのユーザーの予定を表すタスクのみが含まれます。つまり、結果は、そのユーザーが要求する権限を持つタスクに制限されるということです。このフィルターは、照会テーブル・フィルターまたは照会フィルターとして指定することもできますが、照会のパフォーマンス上の理由で、それらのフィルターを許可フィルターとして指定することをお勧めします。

関連概念

338 ページの『定義済み照会テーブル』
定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

341 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

343 ページの『複合照会テーブル』
Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

377 ページの『照会テーブル API の許可オプション』
Business Process Choreographer の照会テーブルに対して照会を実行する場合、許可オプションは入力パラメーターとして照会テーブル API のメソッドに渡すことができます。

関連タスク

392 ページの『Business Process Choreographer Explorer の照会テーブルの作成』
EJB 照会 API の代わりに照会テーブルを使用して、Business Process Choreographer Explorer のパフォーマンスを向上させることができます。照会テーブルを作成するには、Query Table Builder を使用します。

照会テーブルの属性タイプ

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会でリテラル値を使用する場合、および照会結果の値にアクセスする場合に必要です。属性タイプごとに、ルールおよびマッピングを使用できます。

Java プログラミング言語およびデータベースで使用可能なタイプのサブセットを使用して、照会テーブルの属性のタイプを定義します。属性タイプは、具象 Java タイプまたはデータベース・タイプを抽象化したものです。補足照会テーブルでは、データベース・タイプから属性タイプへの有効なマッピングを使用する必要があります。

以下の表は、属性タイプを示しています。

表 37. 属性タイプ

属性タイプ	説明
ID	ヒューマン・タスク (TKIID)、プロセス・インスタンス (PIID)、または他のオブジェクトを識別するために使用する ID。例えば、ID は、指定された TKIID で識別される特定のヒューマン・タスクを要求または完了するために使用されます。
STRING	タスクの説明または照会プロパティーをストリングとして表現できます。
NUMBER	数値は、タスクの優先順位などの属性に使用されます。
TIMESTAMP	タイム・スタンプは、ヒューマン・タスクが作成された時刻や、プロセス・インスタンスが終了した時刻などの特定の時刻を表します。
DECIMAL	10 進数は、XSD の倍精度浮動小数点タイプの変数を使用して照会プロパティーを定義する場合などに、照会プロパティーのタイプとして使用できます。
BOOLEAN	ブール値は、2 つの値 true か false のいずれかです。例えば、ヒューマン・タスクには属性 autoClaim があります。この属性は、このタスクの潜在的所有者としてのユーザーが 1 人のみの場合に、タスクを自動的に要求するかどうかを識別します。

データベース・タイプから属性タイプへのマッピング:

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会テーブルに対して照会を実行する場合、および照会結果の値にアクセスする場合に使用します。

以下の表は、データベース・タイプおよび属性タイプへのマッピングを示しています。

表 38. データベース・タイプから属性タイプへのマッピング

データベース・タイプ	属性タイプ
16 バイトのバイナリー・タイプ。これは、Business Process Choreographer テーブルの TASK の TKIID などの ID に使用されるタイプです。	ID
文字ベースのタイプ。長さは、照会テーブルの属性によって参照されるデータベース表の列によって異なります。	STRING
整数データベース・タイプ (整数、短整数、長整数など)。	NUMBER
タイム・スタンプ・データベース・タイプ。	TIMESTAMP
10 進数タイプ (浮動小数点、倍精度浮動小数点など)。	DECIMAL
ブール値に変換可能なタイプ (数値など)。1 は *true*、他のすべての数値は *false* と解釈されます。	BOOLEAN

通常、補足照会テーブルは、テーブルまたはビューの作成が不要な既存のデータベース表およびビューを参照します。

例

Business Process Choreographer で補足照会テーブルとして表わされるテーブル CUSTOM.ADDITIONAL_INFO が DB2 環境にあるとします。以下の SQL ステートメントは、このデータベース表を作成します。

```
CREATE TABLE CUSTOM.ADDITIONAL_INFO
(
  PIID       CHAR(16) FOR BIT DATA,
  INFO       VARCHAR(220),
  COUNT      INTEGER
);
```

データベース列タイプから照会テーブル属性タイプへの以下のマッピングは、CUSTOM.ADDITIONAL_INFO テーブルの補足照会テーブルで使用されます。

表 39. データベース・タイプから属性タイプへのマッピングの例

データベース列およびタイプ	照会テーブルの属性およびタイプ
PIID CHAR(16) FOR BIT DATA	PIID (ID)
INFO VARCHAR(220)	INFO (STRING)
COUNT INTEGER	COUNT (NUMBER)

属性タイプからリテラル表記へのマッピング:

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会テーブルに対して照会を実行する場合、および照会結果の値にアクセスする場合に使用します。このトピックでは、属性タイプからリテラル表記へのマッピングについて説明します。

リテラル値は、式で使用し、複合照会テーブルのフィルターや照会テーブル API に渡されるフィルターのフィルター基準および選択基準を定義することができます。

以下の表は、属性タイプおよびリテラル値へのマッピングを示しています。プレースホルダーは、*イタリック* で示されています。属性タイプ ID および TIMESTAMP (照会テーブル API に渡せます) では特別な構文を使用することに注意してください。この構文は、照会 API でも使用します。

表 40. 属性タイプからリテラル値へのマッピング

属性タイプ	式におけるリテラル値としての構文および使用法
ID	`ID ('ID のストリング表記' )`
	クライアント・アプリケーションの開発時に、ID はストリングまたは com.ibm.bpe.api.OID インターフェースのインスタンスとして表記されます。ストリング表記は、toString メソッドを使用して、com.ibm.bpe.api.OID インターフェースのインスタンスから取得できます。ストリングは、単一引用符で囲む必要があります。
STRING	`'the string'`
	ストリングは引用符で囲む必要があります。

表 40. 属性タイプからリテラル値へのマッピング (続き)

属性タイプ	式におけるリテラル値としての構文および使用法
NUMBER	`number`
	引用符なしのテキストとしての数値。定義済み照会テーブルの一部の数値属性には定数が定義されており、その定数を使用できます。
TIMESTAMP	`TS ('YYYY-MM-DDThh:mm:ss')`
	タイム・スタンプは、以下の形式で指定する必要があります。 • `YYYY` 4 桁の年 • `MM` 2 桁の月 • `DD` 2 桁の日 • `hh` 2 桁の時刻 (24 時間表記) • `mm` 2 桁の分 • `ss` 2 桁の秒。タイム・スタンプは、ユーザーの時間帯の定義に従って解釈されます。
DECIMAL	`number.fraction`
	引用符なしのテキストとしての 10 進数。`.fraction` 部分はオプションです。
BOOLEAN	`true、false`
	テキストとしてのブール値。

例

- `filterOptions.setQueryCondition("STATE=2");`
- `filterOptions.setQueryCondition("STATE=STATE_READY");`
- 接続照会テーブル `TASK_DESC` の選択基準: `"LOCALE='en_US'"`
- `filterOptions.setQueryCondition( "PTID=ID ('_PT:8001011e.1dee8e51.247d6df6.29a60000')");`

属性タイプからパラメーターへのマッピング:

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会テーブルに対して照会を実行する場合、および照会結果の値にアクセスする場合に使用します。

以下の表は、属性タイプとパラメーター値へのマッピングを示しています。これらのパラメーター値は、式で使用し、複合照会テーブルのフィルターや照会テーブル API に渡されるフィルターのフィルター基準および選択基準を定義することができます。

表 41. 属性タイプからユーザー・パラメーター値へのマッピング

属性タイプ	式におけるパラメーター値としての使用法
ID	PARAM*(name)* クライアント・アプリケーションの開発時に、ID はストリングまたは com.ibm.bpe.api.OID インターフェースのインスタンスとして表記されます。 パラメーターとして、これらの表記は両方とも有効です。有効な OID を反映するバイトの配列を使用することもできます (バイト)。
STRING	PARAM*(name)* 実行時に toString メソッドによって照会テーブル API に渡されるオブジェクトのストリング表記。
NUMBER	PARAM*(name)* 数値の java.lang.Long、java.lang.Integer、java.lang.Short、または java.lang.String 表記が、照会テーブル API に渡されます。定数の名前 (定義済み照会テーブルの一部の属性で定義されています) を渡すこともできます。
TIMESTAMP	PARAM*(name)* 有効な表記は、以下のとおりです。 • タイム・スタンプの java.lang.String 表記 • com.ibm.bpe.api.UTCDate のインスタンス • java.util.Calendar のインスタンス
DECIMAL	PARAM*(name)* 10 進数の java.lang.Long、 java.lang.Integer、 java.lang.Short、 java.lang.Double、 java.lang.Float、または java.lang.String 表記が、照会テーブル API に渡されます。
BOOLEAN	PARAM*(name)* 有効な値は以下のとおりです。 • ブール値の java.lang.String 表記 • 適切な値 (false の場合は 0、true の場合は 1) を持つ java.lang.Short、java.lang.Integer、java.lang.Long。 • java.lang.Boolean オブジェクト

例

```
...
// this example shows a query against a composite query table
// COMP.TASKS with a parameter "customer"
java.util.List params = new java.util.ArrayList();

list.add(new com.ibm.bpe.api.Parameter("customer", "IBM");
// the business flow manager Enterprise JavaBeans or the
// human task manager Enterprise JavaBeans can be used to access query tables
service.bfm.queryEntities("COMP.TASKS", null, null, params);
...
```

属性タイプから Java オブジェクト・タイプへのマッピング:

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会テーブルに対して照会を実行する場合、および照会結果の値にアクセスする場合に使用します。このトピックでは、属性タイプから Java オブジェクト・タイプへのマッピングについて説明します。

以下の表は、属性タイプおよび照会結果セット内の Java オブジェクト・タイプへのマッピングを示しています。

表 42. 属性タイプから Java オブジェクト・タイプへのマッピング

属性タイプ	関連する Java オブジェクト・タイプ
ID	com.ibm.bpe.api.OID
STRING	java.lang.String
NUMBER	java.lang.Long
TIMESTAMP	java.util.Calendar
DECIMAL	java.lang.Double
BOOLEAN	java.lang.Boolean

例

```
...
// the following example shows a query against a composite query table
// COMP.TA; attribute "STATE" is of attribute type NUMBER
...
// run the query
// the business flow manager Enterprise JavaBeans or the
// human task manager Enterprise JavaBeans can be used to access query tables
EntityResultSet rs = bfm.queryEntities("COMP.TA",null,null,params);

// get the entities and iterate over it
List entities = rs.getEntities();
for (int i = 0 ; i < entities.size(); i++) {

      // work on a particular entity
      Entity en = (Entity) entities.get(i);

      // note that the following code could be written
      // more generalized using the attribute info objects
      // contained in ei.getAttributeInfo()

      // get attribute STATE
      Long state = (Long) en.getAttributeValue("STATE");
      ...
}
...
```

属性タイプの互換性:

属性タイプは、Business Process Choreographer で照会テーブルを定義する場合、照会テーブルに対して照会を実行する場合、および照会結果の値にアクセスする場合に使用します。

以下の表は、属性タイプおよび互換性のある属性タイプを示しています。これらの属性タイプは、照会テーブル内でフィルターおよび選択基準を定義するために使用できます。互換性のある属性タイプには、**X** のマークが付いています。

表 43. 属性タイプの互換性

属性タイプ	ID	STRING	NUMBER	TIMESTAMP	DECIMAL	BOOLEAN
ID	X					
STRING		X				
NUMBER			X		X	
TIMESTAMP				X		
DECIMAL			X		X	
BOOLEAN						X

フィルターおよび条件を指定する照会テーブル式では、比較する属性または値のタイプに互換性がなければなりません。例えば、`WI.OWNER_ID=1` は無効なフィルターです。理由は、左オペランドが STRING 型であるのに対し、右オペランドが NUMBER 型であるためです。

照会テーブルの照会

照会は、照会テーブル API (Business Flow Manager EJB でのみ使用可能) を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。

照会は、1 つの照会テーブルでのみ実行されます。エンティティー・ベースの API メソッドおよび行ベースの API メソッドを使用して、照会テーブルから内容を取得します。入力パラメーターは、照会テーブル API のメソッドに渡されます。

関連概念

351 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder を使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

338 ページの『定義済み照会テーブル』
定義済み照会テーブルは、Business Process Choreographer データベース内のデータへのアクセスを提供します。定義済み照会テーブルは、対応する定義済み Business Process Choreographer データベース・ビュー (TASK ビューまたは PROCESS_INSTANCE ビューなど) を照会テーブルの形式で表現したものです。これらの定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されているため、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

341 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、Business Process Choreographer の管理対象ではないビジネス・データを照会テーブル API に対して公開します。補足照会テーブルを使用すると、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報の取得時に、この外部データを定義済み照会テーブルのデータと一緒に使用することができます。

343 ページの『複合照会テーブル』
Business Process Choreographer 内の複合照会テーブルでは、データベース内のデータが特定の方法で表現されることはありません。このテーブルは、関連した定義済み照会テーブルおよび補足照会テーブルのデータの組み合わせで構成されます。複合照会テーブルを使用して、プロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) の情報を取得します。

355 ページの『照会テーブルのフィルターおよび選択基準』
フィルターおよび選択基準は、Query Table Builder を使用した照会テーブルの開発時に定義します (SQL WHERE 文節に似た構文を使用します)。これらの明確に定義されたフィルターおよび選択基準を使用して、照会テーブルの属性に基づく条件を指定します。

照会テーブル API メソッド:

照会は、照会テーブル API を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。エンティティー・ベースの API メソッドおよび行ベースの API メソッドを使用して、照会テーブルから内容を取得することができます。

照会テーブル API を使用して Business Process Choreographer 内の照会テーブルに対して照会を実行するために、以下のエンティティー・ベースのメソッドおよび行ベースのメソッドが提供されています。

表 44. 照会テーブルに対して実行される照会のメソッド

目的	メソッド
照会の内容	• queryEntities • queryRows これらのメソッドは両方とも、照会テーブルの内容を返します。queryEntities メソッドはエンティティーに基づいて内容を返し、queryRows は行に基づいて内容を返します。
オブジェクト数の照会	• queryEntityCount • queryRowCount これらのメソッドは両方とも、照会テーブル内のオブジェクト数を返しますが、実際の数はエンティティー・ベースのアプローチと行ベースのアプローチのいずれを取るかによって異なります。

queryEntities メソッドと queryEntityCount メソッドを使用するエンティティー・ベースの照会では、1 次照会テーブルの基本キーで定義された一意的に識別可能なエンティティーが照会テーブルに含まれていることが前提となっています。

queryRows メソッドと queryRowCount メソッドを使用する行ベースの照会は、JDBC のような結果セットを返します。これは行ベースであり、ナビゲーション用に first および next メソッドが用意されています。照会テーブル API を使用して照会テーブルで照会を実行したときに返された結果セットは、照会 API によって返された QueryResultSet と比較できます。一般に、行の数は、照会テーブルに含まれるエンティティーの数より大きくなります。行の結果セットには、同じエンティティー (例えば、TKIID などのタスク ID で識別されるヒューマン・タスク) が複数回出現する場合があります。

定義済み照会テーブルに含まれている特定のインスタンスは、Business Process Choreographer 環境内に 1 回のみ存在できます。インスタンスの例として、ヒューマン・タスクやビジネス・プロセスがあります。これらのインスタンスは、ID または ID のセットを使用して一意的に識別されます。これは、ヒューマン・タスクのインスタンスの場合は TKIID で、プロセス・インスタンスの場合は PIID です。

複合照会テーブルは、1 つの 1 次照会テーブルと 0 個以上の接続照会テーブルで構成されます。複合照会テーブルに含まれているオブジェクトは、1 次照会テーブルに含まれているオブジェクトの固有の ID によって一意的に識別されます。複合照会テーブルのエンティティー・タイプは、そこに含まれる 1 次照会テーブルによって決まります。例えば、1 次照会テーブル TASK が含まれる複合照会テーブルには、TASK タイプのエンティティーが含まれます。1 次照会テーブルと接続照会テーブルの間の 1 対 1 または 1 対 0 の関係により、接続照会テーブルのエンティティーは重複することがありません。

エンティティー・ベースの照会では、 1 次照会テーブルの基本キーで定義された、照会テーブルの一意的に識別可能なエンティティーを活用します。 通常、ユーザー・インターフェースのクライアント・アプリケーション・プログラマーは、インスタンスが重複しておらず固有になっているかどうかに関心を払います。例えば、

ヒューマン・タスクは、ユーザー・インターフェース上で 1 回のみ表示されているかなどです。 エンティティー・ベースの照会テーブル API を使用すると、固有のインスタンスが返されます。

インスタンス・ベースの許可を使用する場合、行ベースの照会では、返される 1 次照会テーブルの行が重複する場合があります。

- WORK_ITEM 照会テーブルの情報は、照会を使用して取得されます。例えば、照会テーブルで定義されている属性に加えて WI.REASON 属性も取得する場合、複数の行が結果の対象となります。ユーザーがエンティティー (タスクやプロセス・インスタンスなど) にアクセスできる理由が複数存在する場合があるからです。
- インスタンス・ベースの許可を使用して、重複行の削除を指定しません。インスタンス・ベースの許可を使用する場合、作業項目の情報は取得されなくても、複数の行が返される可能性があります。

エンティティー・ベースの照会テーブル API を使用する場合:

- エンティティー・ベースの照会は、常に SQL distinct 演算子を使用して実行されます。
- エンティティー・ベースの照会では、作業項目に関連した情報の配列値を許容する結果が返されます。

照会テーブル API のパラメーター:

Business Process Choreographer の照会テーブルに対する照会の実行時に、照会テーブル API メソッドを使用して内容を取得します。

以下の入力パラメーターが、照会テーブル API のメソッドに渡されます。

表 45. 照会テーブル API のパラメーター

パラメーター	オプション	タイプおよび説明
照会テーブル名	いいえ	java.lang.String
		照会テーブルの固有の名前。
フィルター・オプション	はい	com.ibm.bpe.api.FilterOptions (Business Flow Manager Enterprise JavaBeans を使用している場合) または com.ibm.task.api.FilterOptions (Human Task Manager Enterprise JavaBeans を使用している場合)。
		照会を定義するために使用できるオプション。例えば、照会しきい値は、返される結果数を制限するために、このパラメーターに設定します。
許可オプション	はい	com.ibm.bpe.api.AuthorizationOptions または com.ibm.bpe.api.AdminAuthorizationOptions (Business Flow Manager Enterprise JavaBeans を使用している場合)。 com.ibm.task.api.AuthorizationOptions または com.ibm.task.api.AdminAuthorizationOptions (Human Task Manager Enterprise JavaBeans を使用している場合)。
		インスタンス・ベースの許可を使用する場合は、許可をさらに制約することができます。ロール・ベースの許可が必要な照会テーブルの場合は、AdminAuthorizationOptions のインスタンスを渡す必要があります。

表 45. 照会テーブル API のパラメーター (続き)

パラメーター	オプション	タイプおよび説明
パラメーター	はい	com.ibm.bpe.api.Parameter (Business Flow Manager Enterprise JavaBeans を使用している場合) または com.ibm.task.api.Parameter (Human Task Manager Enterprise JavaBeans を使用している場合) の java.util.List。
		このパラメーターは、複合照会テーブルのフィルターまたは選択基準で指定されるユーザー・パラメーターを渡すために使用されます。

照会は、特定の 1 つの照会テーブルでのみ実行されます。複数の照会テーブル間の関係は、複合照会テーブルを使用して定義します。照会 API (照会テーブル API は除外) では、これはデータベース・ビューに対応します。

フィルターおよび選択基準は、Query Table Builder を使用した照会テーブルの開発時に式で指定します。詳しくはインフォメーション・センターで、複合照会テーブルに関するトピックと、照会テーブルのフィルターおよび検索条件に関するトピックを参照してください。Query Table Builder について詳しくは、WebSphere Business Process Management SupportPacs のサイトを参照してください。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

照会テーブル名:

Business Process Choreographer の照会テーブルに対して照会を実行する場合、照会テーブル名は入力パラメーターとして照会テーブル API のメソッドに渡されます。

照会テーブル名は、照会が実行された照会テーブルの名前です。

- 定義済み照会テーブルの場合は、定義済み照会テーブルの名前です。
- 複合照会テーブルおよび補足照会テーブルの場合、これは、照会テーブルのモデル化中に指定されるそれぞれの照会テーブルの名前です。複合照会テーブルまたは補足照会テーブルの名前は、*prefix.name* という命名規則に従いますが、*prefix* に'IBM' を使用することはできません。

照会テーブル名と接頭部の両方を大文字にする必要があります。照会テーブル名の最大長は 28 文字です。

照会テーブルのフィルター・オプション:

Business Process Choreographer の照会テーブルに対して照会を実行する場合、フィルター・オプションは入力パラメーターとして照会テーブル API のメソッドに渡すことができます。

com.ibm.bpe.api.FilterOptions クラスのインスタンス (Business Flow Manager Enterprise JavaBeans を使用している場合) または com.ibm.task.api.FilterOptions のインスタンス (Human Task Manager Enterprise JavaBeans を使用している場合) を、照会テーブル API に渡すことができます。フィルター・オプションでは、以下を使用して照会を構成できます。

- しきい値およびオフセット (skipCount)

- ソート属性 (SQL 照会の ORDER BY 文節に類似)
- ユーザー提供の照会フィルター
- 返される属性のセット (作業項目の情報も含まれる)
- その他

照会テーブルから取得できる結果セットは、照会テーブルの定義で指定します。ただし、照会の実行時に、追加のオプションを指定することもできます。以下の表は、FilterOptions オブジェクトを使用して、フィルター・オプションとして指定できるオプションを示しています。

表 46. 照会テーブル API のパラメーター: フィルター・オプション

オプション	タイプ	説明
選択した属性	java.lang.String	• 結果セットに返される照会テーブルの属性のコンマ区切りリスト。 • インスタンス・ベースの許可を使用する場合、WORK_ITEM 照会テーブルの属性に接頭部 `WI.` を付けたもの (例えば `WI.REASON`) を指定することによって、作業項目の情報を取得できます。 • ヌルを指定する場合、照会テーブルのすべての属性が返されます (作業項目の情報はなし)。
ソート属性	java.lang.String	照会テーブル属性のコンマ区切りリスト。オプションで、ASC または DESC (それぞれ昇順、降順を表す) が後ろに続きます。 このリストは、SQL ORDER BY 文節に類似しています: *sortAttributes* ::= *attribute* [ASC|DESC] [, *sortAttributes*]。ASC または DESC を指定しない場合は、ASC が想定されます。ソートは、ソート属性の順序で実行されます。例えば、`"STATE DESC, NAME ASC"` という例では、TASK 照会テーブルのタスクが状態別に降順で、また同じ STATE のグループ内で NAME 別に昇順でソートされます。
しきい値	java.lang.Integer	以下の最大値を定義します。 • queryRows を使用する場合に返される行の最大数。 • queryEntities を使用する場合に返されるエンティティーの最大数。エンティティー結果セットに含まれるエンティティーの数がしきい値の数より少ないとしても、各照会テーブル内の使用可能なエンティティーの実際の数が、当該の照会におけるエンティティーのしきい値の数を超える場合があります。 これは、作業項目の情報を選択する場合の技術上の理由によります。 • queryRowCount または queryEntityCount を使用する場合に返される最大カウント。 デフォルトはヌルです。これは、しきい値が設定されていないことを意味します。
スキップ・カウント	java.lang.Integer	スキップされる行の数 (行ベースの照会) またはエンティティーの数 (エンティティー・ベースの照会) を定義します。しきい値パラメーターの場合と同様に、skipCount は、エンティティー・ベースの照会では正確でない場合があります。 スキップ・カウントは、大容量の結果セットのページ送りを行うために使用されます。デフォルトはヌルです。これは、skipCount が設定されていないことを意味します。

表 46. 照会テーブル API のパラメーター: フィルター・オプション (続き)

オプション	タイプ	説明
時間帯	java.util.TimeZone	タイム・スタンプの変換時に使用される時間帯。例えば、定義済み照会テーブル TASK の CREATED などです。指定しない場合 (ヌル)、サーバーの時間帯が使用されます。
ロケール	java.util.Locale	$LOCALE システム・パラメーターの値を計算するために使用するロケール。例えば、選択基準で $LOCALE を使用するには、'LOCALE=$LOCALE' のようにします。
個別行	java.lang.Boolean	行ベースの照会でのみ使用されます。true に設定すると、行ベースの照会で個別行が返されます。作業項目の情報が重複している可能性があるために固有の行が返されるという意味ではありません。
照会条件	java.lang.String	このオプションは、結果セットに対して追加のフィルタリングを行います。照会テーブルに定義されているすべての属性を参照できます。照会テーブルに許可が必要な場合は、WI 接頭部を使用して (例えば、WI.REASON=REASON_POTENTIAL_OWNER)、WORK_ITEM 照会テーブルに定義された列を参照することもできます。

照会テーブル API の許可オプション:

Business Process Choreographer の照会テーブルに対して照会を実行する場合、許可オプションは入力パラメーターとして照会テーブル API のメソッドに渡すことができます。

com.ibm.bpe.api.AuthorizationOptions クラスまたは com.ibm.bpe.api.AdminAuthorizationOptions のインスタンス (Business Flow Manager EJB を使用している場合)、あるいは com.ibm.task.api.AuthorizationOptions クラスまたは com.ibm.task.api.AdminAuthorizationOptions クラスのインスタンス (Human Task Manager EJB を使用している場合) を使用して、照会の実行時に追加の許可オプションを指定します。

インスタンス・ベースの許可を使用する場合、AuthorizationOptions クラスのインスタンスでは、照会で返される適格なインスタンスを識別するための作業項目のタイプを指定できます。

インスタンス・データが含まれている定義済み照会テーブルに対して照会が実行される場合、AuthorizationOptions クラスのインスタンスは、照会テーブル API に渡すことができます。このインスタンスは、インスタンス・データが含まれている 1 次照会テーブルを含む複合照会テーブルに対して照会が実行される場合で、インスタンス・ベースの許可を使用するように構成されている場合にも渡すことができます。テンプレート・データを含む定義済み照会テーブルまたはロール・ベースの許可を構成した複合照会テーブルに対して照会を実行すると、com.ibm.bpe.api.EngineNotAuthorizedException 例外 (Business Flow Manager EJB を使用している場合) または com.ibm.task.api.NotAuthorizedException (Human Task Manager EJB を使用している場合) がスローされます。他のすべての場合、照会テーブル API に渡される許可オプションは無視されます。

複合照会テーブルでは、包含されるオブジェクト (またはエンティティー) を識別するときに、考慮対象となる作業項目のタイプを制限することができます。例えば、このことは、照会テーブル API に渡される許可オプションが、全員作業項目を使用

するように構成されている場合、全員作業項目が複合照会テーブルの定義で使用するように定義されている場合にのみ考慮されます。簡単なルールとして、照会テーブル定義で考慮対象として指定されていない作業項目タイプは、照会テーブル API によって考慮対象として上書きすることはできません。しかし、照会テーブル定義で考慮対象として指定されている作業項目タイプは、使用しないとして上書きすることができます。また、複合照会テーブルまたは定義済み照会テーブルの許可タイプは、照会テーブル API によって上書きすることはできません。

許可オブジェクトが指定されていない場合、または関連属性 (全員、個人、グループ、または継承) がヌルに設定されている場合 (デフォルト) は、照会する照会テーブルのタイプによって、さまざまな許可オプションのデフォルトが適用されます。

以下の表は、使用される照会テーブル・タイプおよび作業項目タイプのインスタンス・ベースの許可の許可オプションのデフォルトを示しています。

表 47. 照会テーブル API パラメーター: インスタンス・ベース許可の許可オプションのデフォルト

照会テーブルのタイプ	全員作業項目	個人作業項目	グループ作業項目	継承された作業項目
インスタンス・データを含む定義済み照会テーブル	TRUE	TRUE	TRUE	FALSE
テンプレート・データを含む定義済み照会テーブル	なし	なし	なし	なし
インスタンス・データが含まれている 1 次照会テーブルを含む複合照会テーブル	TRUE	TRUE	TRUE	TRUE
テンプレート・データが含まれている 1 次照会テーブルを含む複合照会テーブル	なし	なし	なし	なし
補足照会テーブル	なし	なし	なし	なし

N/A は、インスタンス・ベースの許可は使用されないこと、および作業項目に関連した許可オブジェクトの設定はすべて無視されることを意味します。

TRUE を指定すると、この作業項目のタイプを使用するように照会テーブルが定義されている場合、結果照会では特定の作業項目タイプのみが考慮されます。これは、インスタンス・データを含むすべての定義済み照会テーブルの場合に当てはまりますが、複合照会テーブルの場合には当てはまりません。グループ作業項目の場

合、ヒューマン・タスク・コンテナーでは後者も有効にする必要があります。継承された作業項目を TRUE に設定すると、例えば、プロセス・インスタンスの管理者は、プロセス・インスタンス用に作成される参加ヒューマン・タスク・インスタンスを確認できるようになります。

以下の場合は、AuthorizationOptions クラスのインスタンスではなく、AdminAuthorizationOptions クラスのインスタンスを指定します。

- 照会は、ロール・ベースの許可を持つ照会テーブルに対して実行します。テンプレート・データを含む定義済み照会テーブルはロール・ベースの許可を必要とします。テンプレート・データを含む 1 次照会テーブルを持つ複合照会テーブルは、ロール・ベースの許可を要求するように構成できます。
- インスタンス・データが含まれている照会テーブル、またはインスタンス・データが含まれている 1 次照会テーブルを含む複合照会テーブルに対して照会を実行する場合。この照会では、特定のユーザーの許可による制限に関係なく、照会テーブルの内容が返されます。この動作は、照会 API (照会テーブル API は除外) の queryAll メソッドを使用する場合と同じです。
- 別のユーザーの代わりに照会を実行する必要がある場合。

以下の表は、上記のさまざまな動作が行われる方法を示しています。

表 48. 照会テーブル API パラメーター: AdminAuthorizationOptions

状態	説明
onBehalfUser がヌルに設定されている	• ロール・ベースの許可を持つ照会テーブルに対して照会を実行すると、その照会テーブルのすべての内容が返されます。 • インスタンス・ベースの許可を使用する照会テーブルに対して照会が実行される場合は、照会テーブルに含まれている特定のオブジェクトに特定のユーザーの作業項目が存在するかどうかは検査されません。照会テーブルに含まれているすべてのオブジェクトが返されます。
onBehalfUser が特定のユーザーに設定されている	インスタンス・ベースの許可を使用すると、指定のユーザーの権限で照会が実行され、照会テーブルのオブジェクトはそのユーザーの作業項目に照らして検査されます。

AdminAuthorizationOptions を指定した場合、呼び出し元は、BPESystemAdministrator または BPESystemMonitor Java EE ロール (Business Flow Manager EJB を使用している場合)、あるいは TaskSystemAdministrator または TaskSystemMonitor Java EE ロール (Human Task Manager EJB を使用している場合) を持っている必要があります。

関連概念

360 ページの『照会テーブルの許可』
照会テーブルで照会を実行するときには、インスタンス・ベースの許可、ロール・ベースの許可、または許可なしを使用できます。

パラメーター:

Business Process Choreographer の照会テーブルに対して照会を実行する場合、ユーザー・パラメーターを入力パラメーターとして照会テーブル API のメソッドに渡す

ことができます。照会テーブル定義で、1 次照会テーブル、許可、および照会テーブルのフィルターにパラメーターを指定できます。また、接続照会テーブルの選択基準にパラメーターを指定することもできます。

システム・パラメーター $USER および $LOCALE は、フィルターおよび選択基準で実行時に置き換えられます。照会テーブル API に渡す必要はありません。$LOCALE システム・パラメーターを計算するための入力値は、フィルター・オプションでロケールを設定することによって指定します。

ユーザー・パラメーターは、照会の実行時に照会テーブル API に渡す必要があります。そのためには、com.ibm.bpe.api.Parameter クラスのインスタンス (Business Flow Manager EJB を使用している場合) または com.ibm.task.api.Parameter クラスのインスタンス (Human Task Manager EJB を使用している場合) のリストを渡します。

パラメーター・オブジェクトで、以下のプロパティーを指定する必要があります。

表 49. 照会テーブル API のユーザー・パラメーター

プロパティー	説明
名前	照会テーブル定義で使用されているパラメーターの名前。この名前では、大/小文字が区別されます。
値	パラメーターの値。パラメーターのタイプは、そのパラメーターを使用するすべてのフィルターおよび選択基準の左オペランドのタイプとの互換性がなければなりません。定義済み照会テーブルの一部の属性で定義されている定数 (STATE_READY など) はストリングとして渡すことができます。

例

```
// execute a query against a composite query
// table CUST.CPM with the primary query table filter
// set to 'STATE=PARAM(theState)'
EntityResultSet ers = null;
List parameterList = new ArrayList();
parameterList.add(new Parameter
("theState", new Integer(2)));

// run the query;
// the business flow manager EJB or the
// human task manager EJB can be used to access query tables
ers = bfm.queryEntities
("CUST.CPM", null, null, parameterList);

// work on the result set
// ...
```

照会テーブルの照会の結果:

Business Process Choreographer の照会テーブルに対する照会の実行時に、照会テーブル API メソッドを使用します。 queryEntityCount メソッドまたは queryRowCount メソッドの照会結果は数値です。queryEntities および queryRows メソッドでは、結果セットが返されます。

EntityResultSet

Business Flow Manager Enterprise JavaBeans を使用している場合は、com.ibm.bpe.api.EntityResultSet クラスのインスタンスが、メソッド queryEntities によって返されます。Human Task Manager Enterprise JavaBeans を使用している場合は、com.ibm.task.api.EntityResultSet クラスのインスタンスが、メソッド queryEntities によって返されます。エンティティー結果セットには、以下のプロパティーがあります。

表 50. 照会テーブル API エンティティーのエンティティー結果セットのプロパティー

プロパティー	説明
queryTableName	照会が実行された照会テーブルの名前。
entityTypeName	• 複合照会テーブルに対して照会が実行された場合、これは 1 次照会テーブルの名前です。 • 定義済み照会テーブルまたは補足照会テーブルに対して照会が実行された場合、これは照会テーブルの名前です。つまり、*queryTableName* プロパティーと同じ値です。
EntityInfo	このプロパティーには、エンティティー結果セットに含まれるエンティティーのメタ情報が含まれています。com.ibm.bpe.api.AttributeInfo オブジェクトの java.util.List リスト (Business Flow Manager EJB を使用している場合) または com.ibm.task.api.AttributeInfo オブジェクトのリスト (Human Task Manager EJB を使用している場合) を、このオブジェクトで取得できます。このリストには、この結果セットのエンティティーに含まれる情報の属性名および属性タイプが含まれています。これらのエンティティーのキーを構成する属性のメタ情報も含まれています。
entities	com.ibm.bpe.api.Entity オブジェクトの java.util.List リスト (Business Flow Manager EJB を使用している場合) または com.ibm.task.api.Entity オブジェクトのリスト (Human Task Manager を使用している場合)。
locale	`$LOCALE` システム・パラメーターに対して計算されたロケール。

Entity クラスのインスタンスには、照会テーブルの照会で取得された情報が含まれています。エンティティーは、一意的に識別可能なオブジェクト (タスク、プロセス・インスタンス、アクティビティー、エスカレーションなど) を表します。エンティティーでは、以下のプロパティーを使用できます。

表 51. 照会テーブル API エンティティーのエンティティー・プロパティー

プロパティー	説明
EntityInfo	エンティティー結果セットにも含まれている EntityInfo オブジェクト。com.ibm.bpe.api.AttributeInfo オブジェクトの java.util.List リスト (Business Flow Manager EJB を使用している場合) または com.ibm.task.api.AttributeInfo オブジェクトのリスト (Human Task Manager EJB を使用している場合) を、このオブジェクトで取得できます。このリストには、この結果セットのエンティティーに含まれる情報の属性名および属性タイプが含まれています。これらのエンティティーのキーを構成する属性のメタ情報も含まれています。
attributeValue (*attributeName*)	このエンティティーで取得された指定された属性の値。タイプは、この属性の関連した AttributeInfo オブジェクトに入れられます。
attributeValuesOfArray (*attributeName*)	値の配列。このプロパティーは、属性情報プロパティー *array* が true に設定されている場合 (この属性が現在作業項目の情報を参照している場合のみ) に使用します。

エンティティー結果セットに含まれるエンティティーの数は、エンティティーのリストに対して size メソッドを使用して取得します。

例: エンティティー・ベースの照会テーブル API:

```
...
// the following example shows a query against
// predefined query table TASK, using the entity-based API

...
// run the query
// service is a (Local)BusinessFlowManager object or a
// (Local)HumanTaskManager object
EntityResultSet rs = service.queryEntities("TASK", null, null, null);

// get the entities meta information
EntityInfo ei = rs.getEntityInfo();
List atts = ei.getAttributeInfo();

// get the entities and iterate over it
Iterator entitiesIter = rs.getEntities().iterator();
while (entitiesIter.hasNext()) {

      // work on a particular entity
      Entity en = (Entity) entitiesIter.next();

      for (int i = 0; i < atts.size(); i++) {
            AttributeInfo ai = (AttributeInfo) atts.get(i);
            Serializable value = en.getAttributeValue(ai.getName()) ;

            // process...
      }
}
...
```

RowResultSet

Business Flow Manager Enterprise JavaBeans を使用している場合は、com.ibm.bpe.api.RowResultSet クラスのインスタンスが、メソッド queryRows によって返されます。Human Task Manager Enterprise JavaBeans を使用している場合は、

com.ibm.task.api.RowResultSet クラスのインスタンスが、メソッド queryRows によって返されます。このタイプの結果セットは、JDBC 結果セットに似ています。行結果セットには、以下のプロパティーがあります。

表 52. 照会テーブル API 行の行結果セットのプロパティー

プロパティー	説明
primaryQueryTableName	• 複合照会テーブルに対して照会が実行された場合、これは 1 次照会テーブルの名前です。 • 定義済み照会テーブルまたは補足照会テーブルに対して照会が実行された場合、これは照会テーブルの名前です。つまり、*queryTableName* プロパティーと同じ値です。
attributeInfo	このプロパティーには、com.ibm.bpe.api.AttributeInfo オブジェクトのリスト (Business Flow Manager Enterprise JavaBeans を使用している場合) または com.ibm.task.api.AttributeInfo オブジェクトのリスト (Human Task Manager Enterprise JavaBeans を使用している場合) が含まれています。それらのオブジェクトは、この結果セットのメタ情報を記述しています。AttributeInfo オブジェクトには、情報の属性名および属性タイプが含まれています。行結果セットはキーを持たないため、キーに関するメタデータは含まれません。
attributeValue	この行で取得された指定された属性の値。タイプは、この属性の関連した AttributeInfo オブジェクトに入れられます。
next、first、last、previous	行結果セットは、これらのメソッドを使用してナビゲートします。イテレーター、列挙型、または JDBC 結果セットと使用法を比較してください。

行結果セットに含まれる行の数は、行のリストに対して size メソッドを使用して取得します。

例: 行ベースの照会テーブル API

```
...
// the following example shows a query against
// predefined query table TASK, using the entity-based API
...
// run the query
// service is a (Local)BusinessFlowManager object or a
// (Local)HumanTaskManager object
RowResultSet rs = service.queryRows("TASK", null, null, null);

// get the entities meta information
List atts = rs.getAttributeInfo();

// get the entities and iterate over it
while (rs.next()) {

      // work on a particular row
      for (int i = 0; i < atts.size(); i++) {
            AttributeInfo ai = (AttributeInfo) atts.get(i);
            Serializable value = rs.getAttributeValue(ai.getName()) ;

            // process...
      }
}
...
```

メタデータ取得のための照会テーブルの照会

照会は、照会テーブル API を使用して、Business Process Choreographer 内の照会テーブルに対して実行されます。メソッドを使用して照会テーブルからメタデータを取得できます。

照会テーブル API を使用して Business Process Choreographer 内の照会テーブルに対して照会を実行するときにメタデータを取得するために、以下のメソッドが用意されています。

表 53. 照会テーブルでのメタデータ取得のためのメソッド

目的	メソッド
特定の照会テーブルのメタデータを返す	getQueryTableMetaData
特定のプロパティーを持つ照会テーブル・メタデータのリストを返す	findQueryTableMetaData
(エンティティーに基づいて) 照会テーブルの内容と、選択された属性のメタデータのサブセットを返す	queryEntities
(行に基づいて) 照会テーブルの内容と、選択された属性のメタデータのサブセットを返す	queryRows

照会テーブルのメタデータは、構造に関連するデータおよび国際化対応に関連するデータから構成されます。

照会テーブルの構造に関連したメタデータを以下の表に示します。

表 54. 照会テーブルの構造に関連したメタデータ

メタデータ	説明	getQuery-TableMetaData によって返される	findQuery-TableMetaData によって返される	queryEntities によって返される	queryRows によって返される
照会テーブル名	照会テーブルの名前	はい	はい	はい	はい
1 次照会テーブル名	補足照会テーブルおよび定義済み照会テーブルの場合は照会テーブルの名前。複合照会テーブルの場合は 1 次照会テーブルの名前	はい	はい	はい	はい
種類	照会テーブルのタイプ。predefined (定義済み)、composite (複合)、supplemental (補足) のいずれかです。	はい	はい	いいえ	いいえ
許可	照会テーブルで定義されている以下の許可。 • 作業項目の使用 • インスタンス・ベース、ロール・ベース、または許可なし	はい	はい	いいえ	いいえ

表 54. 照会テーブルの構造に関連したメタデータ (続き)

メタデータ	説明	getQuery-TableMetaData によって返される	findQuery-TableMetaData によって返される	queryEntities によって返される	queryRows によって返される
定義されている属性	照会テーブルで定義されている属性のメタデータ	はい	はい	いいえ。選択された属性のメタデータが返されます。	いいえ。選択された属性のメタデータが返されます。
キー属性	照会テーブルのキー属性	はい	はい	はい	いいえ。行ベースの照会には適用できません。

照会テーブルの国際化対応に関連したメタデータを以下の表に示します。

表 55. 照会テーブルの国際化対応に関連したメタデータ

メタデータ	説明	getQuery-TableMetaData によって返される	findQuery-TableMetaData によって返される	queryEntities によって返される	queryRows によって返される
locales[]	照会テーブルおよび属性の表示名および説明が定義されているロケール。	はい	はい	いいえ	いいえ
ロケール	API に渡されたロケールから得られた `$LOCALE` システム・パラメーターの値。	はい	はい	はい	はい
照会テーブルの表示名および説明	照会テーブルの表示名および説明 (定義されているすべてのロケールに対して提供されているもの)。	はい	はい	いいえ	いいえ
属性の表示名および説明	属性の表示名および説明 (定義されているすべてのロケールに対して提供されているもの)。	はい	はい	いいえ	いいえ

照会テーブル・メタデータを返す EJB 照会テーブル API のメソッドは、いずれもロケール・パラメーターを受け入れます (`FilterOptions.setLocale` や `MetaDataOptions.setLocale` など)。このパラメーターは、クライアントがユーザーに情報を提示するために使用する Java ロケールに設定する必要があります。このロケール・パラメーターは、`$LOCALE` システム・パラメーターの値を計算するために使用されます。このシステム・パラメーターは、フィルターおよび選択基準で使用できます。返されるロケールには、`$LOCALE` に使用される実際の Java ロケールが含まれます。

特定の照会テーブルの表示名および説明を取得する場合には、関連したメソッドに getLocale を渡すと、タスクの説明と同じロケールで表示名および説明が取得されます。例えば、これらの説明は `'LOCALE=$LOCALE'` の選択基準を使用して付加されます。

例

```
// the following example shows how meta data for a particular
// composite query table can be retrieved

...
// run the query
MetaDataOptions mdo = new MetaDataOptions("TASK", null, false, new Locale("en_US"));
List list = bfm.findQueryTableMetaData(mdo);

// to get the meta data of a specific query table
// use bfm.getQueryTableMetaData(...)

// iterate through the list of query tables that have TASK as primary query table
// => at least one query table is returned: the predefined query table TASK

Iterator iter = list.iterator();
while (iter.hasNext()) {
 QueryTableMetaData md = (QueryTableMetaData) iter.next();
 Locale effectiveLocale = md.getLocale();
 String queryTableDisplayName = md.getDisplayName(effectiveLocale);
 System.out.println("found query table: " + queryTableDisplayName);
 List attributesList = md.getAttributeMetaData();
 Iterator attrIter = attributesList.iterator();
 while (attrIter.hasNext()) {
  AttributeMetaData amd = (AttributeMetaData) attrIter.next();
  String attributeDisplayName = amd.getDisplayName(effectiveLocale);
  System.out.println("¥tattribute:" + attributeDisplayName);
 }
}
```

最良一致ロケール

接続照会テーブルで条件を指定するとき、値 $LOCALE を使用すると、指定されたロケールがメタデータと正確に一致しない場合に予期しない結果が返される可能性があります。例えば、メタデータで言語が en と指定されている照会テーブルに対して、照会でロケール en_US を渡すと、返される結果は null になります。

このような事態を避けるために、LOCALE=$LOCALE_BEST_MATCH を使用できます。これにより、照会で使用される実際のロケールを計算する最良一致アルゴリズムが適用されます。例えば、言語が en の照会テーブルに対して、ロケール en_US を使用して照会した場合、この照会は、en を使用して実行されます。

条件 LOCALE=$LOCALE_BEST_MATCH に、他の論理演算子または比較演算子を指定することはできません。最良一致ロケール条件は接続照会テーブルのみで使用することができ、他の照会でこれを条件として指定すると、エラーが発生します。

照会テーブル・メタデータの国際化対応

照会テーブル・メタデータは国際化対応がサポートされています。

複合照会テーブルの表示名および説明は複数のロケールで指定できます。例えば、複合照会テーブルでは、en_US ロケール、de ロケール、およびデフォルト・ロケールで照会テーブルの表示名を定義することができます。これは、Query Table Builder を使用して照会テーブルを作成するときに行います。表示名および説明をローカライズした照会テーブルをデプロイするには、照会テーブルを Business Process Choreographer コンテナーにデプロイするときに -deploy jarFile オプションを使用する必要があります。

ロケール処理の点では、照会テーブル API のメソッド queryEntities および queryRows の動作と、照会テーブル API のメタデータ・メソッド getQueryTableMetaData および findQueryTableMetaData は、Java リソース・バンドルによって提供されるものと同様です。

照会テーブル・メタデータの表示名および説明を照会テーブルの内容と整合させるために、$LOCALE システム・パラメーターの値は、照会テーブルで表示名および説明を指定しているロケールによって決定されます。

例

以下のシナリオを考えてみます。このシナリオでは、クライアントがタスク・リストまたはプロセス・リストを表示し、照会テーブルを照会するための要求を作成します。

- クライアントが、ユーザーに情報を提示するために使用するロケールを指定していない。アプリケーションが各種の言語に対応していない可能性があります。
  - 表示名および説明のデフォルト・ロケールが照会テーブルで指定されている。これには、現行バージョンの Query Table Builder で作成されたすべての複合照会テーブルおよび補足照会テーブルが該当します。そのため、$LOCALE の値は default に設定されています。
  - 照会テーブルが、デフォルト・ロケールの照会テーブルの表示名または説明を指定していない。これには、すべての定義済み照会テーブルと、-deploy qtdFile オプションを使用してデプロイされるすべての照会テーブルが該当します。$LOCALE の値は Java リソース・バンドル・メソッドに基づいて決定されます。
- クライアントが、ユーザーに情報を提示するために使用するロケールを指定した。例えば、照会テーブルに対する REST API を使用するときがこれに該当します。
  - 表示名および説明が照会テーブルで指定されている。クライアントによって渡されたロケールに基づき、Java リソース・バンドル・メソッドを使用して $LOCALE の値を計算します。
  - 表示名および説明が照会テーブルで指定されていない。$LOCALE の値が、クライアントによって渡された値に設定されます。

最良一致ロケール

接続照会テーブルで条件を指定するとき、値 $LOCALE を使用すると、指定されたロケールがメタデータと正確に一致しない場合に予期しない結果が返される可能性があります。例えば、メタデータで言語が en と指定されている照会テーブルに対して、照会でロケール en_US を渡すと、返される結果は null になります。

このような事態を避けるために、LOCALE=$LOCALE_BEST_MATCH を使用できます。これにより、照会で使用される実際のロケールを計算する最良一致アルゴリズムが適用されます。例えば、言語が en の照会テーブルに対して、ロケール en_US を使用して照会した場合、この照会は、en を使用して実行されます。

条件 LOCALE=$LOCALE_BEST_MATCH に、他の論理演算子または比較演算子を指定することはできません。最良一致ロケール条件は接続照会テーブルのみで使用することができ、他の照会でこれを条件として指定すると、エラーが発生します。

関連タスク

392 ページの『Business Process Choreographer Explorer の照会テーブルの作成』
EJB 照会 API の代わりに照会テーブルを使用して、Business Process Choreographer Explorer のパフォーマンスを向上させることができます。照会テーブルを作成するには、Query Table Builder を使用します。

照会テーブルおよび照会パフォーマンス

照会テーブルでは、Business Process Choreographer のヒューマン・タスクおよびビジネス・プロセスのリストを取得するクライアント・アプリケーションを開発するためのクリーンなプログラミング・モデルが導入されています。照会テーブルを使用することで、パフォーマンスが向上します。照会テーブル API のパラメーターと、パフォーマンスに影響するその他の要因について説明します。

照会テーブルに対する照会の応答時間は、主に使用した許可オプション、フィルター、および選択基準の影響を受けます。考慮すべき一般的なパフォーマンスに関するヒントは、以下のとおりです。

- 許可オプションはパフォーマンスに大きく影響します。許可は、可能な限り少ないオプション (個人およびグループの作業項目など) を使用して有効化してください。継承される作業項目の使用は避けてください。許可オプションは、照会の実行時にさらに制限できます。また、必要でない場合は、作業項目を使用する許可が不要であると指定してください。
- 作業項目を使用する許可が必要な場合は、許可フィルターを指定してください。例えば、潜在的所有者の作業項目を持つ照会テーブルのオブジェクトのみを許可するには、WI.REASON=REASON_POTENTIAL_OWNER を使用します。
- 1 次照会テーブルでのフィルター処理は効率的です (例えば、TASK が 1 次照会テーブルである照会テーブルで作動可能状態になっているタスクのみを許可するなど)。
- 照会テーブルでのフィルター処理および照会フィルターは、照会の実行時に渡されるフィルターであり、1 次フィルターよりもパフォーマンスが落ちます。
- 可能な場合は、フィルターおよび選択基準のパラメーターを使用するのを避けてください。
- フィルターおよび選択基準で LIKE 演算子を使用するのは避けてください。

複合照会テーブルの定義

以下の表は、複合照会テーブルに定義されているオプションの照会パフォーマンスの影響について示しています。複合照会テーブル定義に関連した他のトピックの情報も含まれています。「パフォーマンスへの影響」列に示されている影響は、パフォーマンスへの影響の平均であり、実際に観察される影響は異なる可能性があります。

表 56. 複合照会テーブルのオプションが照会パフォーマンスに与える影響

オブジェクトまたはトピック	パフォーマンスへの影響	説明
照会テーブル・フィルター	マイナス	照会テーブルに対するフィルターは、照会パフォーマンスに対して最もマイナス影響を与えるフィルターです。一般に、これらのフィルターでは、データベースの定義済み索引を使用できません。
1 次照会テーブル・フィルター	プラス	1 次照会テーブルに対するフィルターでは、照会結果セットの計算の初期の段階で、ハイパフォーマンス・フィルター処理が提供されます。 1 次照会テーブル・フィルターを使用して、照会テーブルの内容を制限することをお勧めします。
許可フィルター	プラス	許可に対するフィルターを使用すると、1 次照会テーブル・フィルターがパフォーマンスを改善するように、照会のパフォーマンスが改善されます。可能な場合は、許可フィルターを適用してください。例えば、リーダー作業項目を考慮してはならない場合は、`WI.REASON=REASON_READER` を指定します。
選択基準	なし	1 次照会テーブルと接続照会テーブルとの一部の関係では、1 対 1 または 1 対 0 の関係を満たすために、選択基準の定義が必要です。一般に、選択基準は、少数の行に対してのみ評価されるため、パフォーマンスに与える影響はわずかです。
パラメーター	なし	現在、照会テーブルでパラメーターを使用しても、パフォーマンスにマイナス影響が及ぶことはありません。それでも、パラメーターは、必要な場合にのみ使用してください。
インスタンス・ベースの許可	マイナス	インスタンス・ベースの許可を使用する場合は、作業項目が存在するかどうかを照会テーブルの各オブジェクトで検査する必要があります。作業項目は、WORK_ITEM 照会テーブルの項目として表現されます。この検査はパフォーマンスに影響を及ぼします。
インスタンス・ベースの許可フラグ: • everybody • individuals • groups • inherited	マイナス	照会テーブルで使用するように指定された各タイプの作業項目はパフォーマンスに影響を与えます。大容量の照会を実行するアプリケーションでは、個人およびグループの作業項目、または、いずれか一方の作業項目のみを使用する必要があります。通常、継承された作業項目は不要です。予定を表すヒューマン・タスクを返すタスク・リストを定義する場合は特にそうです。継承された作業項目は、それが必要であることが明らかな場合にのみ使用します。例えば、ユーザーがエンクロージング・ビジネス・プロセスの許可に基づく読み取りアクセス権限を持っている可能性があるビジネス・プロセスに属するタスクのリストを返すような場合です。
ロール・ベースの許可または許可なし。	なし	ロール・ベースの許可または許可なしを使用する場合は、作業項目に対する検査が実行されません。

表 56. 複合照会テーブルのオプションが照会パフォーマンスに与える影響 (続き)

オブジェクトまたはトピック	パフォーマンスへの影響	説明
定義済み属性の数	現在なし	現在、照会テーブルに含まれている属性の数は、パフォーマンスに影響を与えることはありません。それでも、照会テーブルでは、必要な属性のみを使用してください。

照会テーブル API

以下の表は、照会テーブル API で指定されているオプションの照会パフォーマンスの影響について示しています。「パフォーマンスへの影響」列に示されている影響は、パフォーマンスへの影響の平均であり、実際に観察される影響は異なる可能性があります。

表 57. 照会テーブル API のオプションが照会パフォーマンスに与える影響

オプション	パフォーマンスへの影響	説明
選択した属性	マイナス (少ないほど良い)	照会テーブルに対する照会の実行時に選択されている属性の数は、データベースと Business Process Choreographer 照会テーブル・ランタイムの両方で処理する必要がある数に影響を与えます。また、複合照会テーブルでは、接続照会テーブルの情報を取得する必要があるのは、その情報が選択された属性で指定されているか、照会テーブル・フィルターまたは照会フィルターによって参照されている場合のみです。
照会フィルター	マイナス	指定する場合、照会フィルターがパフォーマンスに与える影響は、現在、照会テーブル・フィルターと同じです。ただし、フィルターは、照会テーブル API に渡すのではなく、照会テーブルで指定することをお勧めします。
ソート属性	マイナス	照会結果セットのソートはコストがかかる操作であり、ソートを使用する場合、データベース最適化は制限されます。必要がない場合は、ソートは使用しないでください。ただし、ほとんどのアプリケーションでソートが必要です。
しきい値	プラス	しきい値を指定すると、照会パフォーマンスが大幅に改善される可能性があります。しきい値は、常に指定するようにしてください。
スキップ・カウント	マイナス	照会結果セット内の特定の数のオブジェクトのスキップは影響が大きいため、必要な場合 (照会結果に対してページ送りを実行する場合など) にのみ実行してください。
時間帯	なし	時間帯設定はパフォーマンスに影響を与えません。
ロケール	なし	ロケール設定はパフォーマンスに影響を与えません。
個別行	マイナス	照会で重複行の削除を使用すると、パフォーマンスにいくらかの影響が及びますが、重複していない行を取得するために必要です。このオプションは、行ベースの照会にのみ影響を与え、他の場合は無視されます。

表 57. 照会テーブル API のオプションが照会パフォーマンスに与える影響 (続き)

オプション	パフォーマンスへの影響	説明
照会数のカウント	プラス	エンティティーの総数または特定の照会の行数が必要な場合、つまり、照会テーブルのすべての項目の内容が必要であるわけではない場合にのみ、queryEntityCount または queryRowCount メソッドを使用してください。Business Process Choreographer ランタイムは、照会数のカウントでのみ有効な最適化を適用できます。

その他の考慮事項

そのほかに以下の点についてもパフォーマンスに関する考慮が必要です。

表 58. 照会テーブルのパフォーマンス: その他の考慮事項

項目	説明
システム上の照会テーブルの数	照会テーブルの照会のパフォーマンスは、Business Process Choreographer コンテナーにデプロイされている照会テーブルの数によって影響を受けることはありません。同様に、ビジネス・プロセス・インスタンスのナビゲーションや、ヒューマン・タスクの要求または実行操作も、現在時点では影響を受けることはありません。保守容易性を保つために、照会テーブルは妥当な数にとどめてください。一般に、1 つの照会テーブルは、ユーザー・インターフェースに表示される 1 つのタスク・リストまたはプロセス・リストを表します。
データベースのチューニング	照会テーブルの内容にアクセスするために最適化された SQL を使用しますが、Business Process Choreographer データベースでは、以下のデータベース・チューニングを実装する必要があります。 • データベース・サーバー上で実行されている他のプロセスやハードウェア制約を考慮に入れ、データベース・メモリーを最大に設定する必要があります。 • データベースに関する統計は、最新の状態でなければならず、定期的に更新する必要があります。通常、このような手順は、大規模なトポロジーでは既に実装されています。例えば、データベース内のデータの変更を反映するために、最適化プログラムのデータベース統計を週に一度収集します。 • データベース・システムには、データ・コンテナーを再編成 (デフラグ) するためのツールがあります。データベース内のデータの物理的なレイアウトが、照会パフォーマンスおよび照会のアクセス・パスに影響を与える可能性もあります。 • 良い照会パフォーマンスを得るには、最適な索引が重要です。Business Process Choreographer には、一般的なシナリオのプロセス・ナビゲーションと照会パフォーマンスの両方のために最適化されている定義済みの索引が用意されています。カスタマイズされた環境では、大容量のタスクまたはプロセス・リストの照会をサポートするために、追加の索引が必要となる場合があります。照会テーブルに対して実行される照会をサポートするには、データベースで提供されるツールを使用してください。

Business Space の照会テーブルの作成

Query Table Builder で、定義済みプロパティーを持つ複合照会テーブル定義を使用して、Business Space の照会テーブルを作成できます。

始める前に

Query Table Builder は Eclipse プラグインとして使用可能であり、WebSphere Business Process Management SupportPacs のサイトからダウンロードできます。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

手順

1. Query Table Builder で、プロジェクトを右クリックしてから、「**新規**」 → 「**Business Space の複合照会定義 (Composite Query Definition for Business Space)**」を選択します。ウィザードの指示に従って照会テーブル定義を作成します。新規照会テーブル定義は定義済みプロパティーから構成されています。必要な場合は、照会テーブル定義にさらにプロパティーを追加し、照会テーブル定義ファイルを WebSphere Process Server にデプロイします。

   注: Query Table Builder でプロパティーにつけた名前が、Business Space for Choreographer でタスク・プロパティーの名前として使用されます。
2. 照会テーブル定義ファイルを作成し、デプロイすると、それらを Business Space で構成できます。例えば、「タスク・リスト」ウィジェットの照会テーブル定義ファイルをデプロイした場合、以下のようにします。
   a. ウィジェット・メニューを開き、「**構成**」を選択し、次に「**コンテンツ**」タブを選択します。
   b. 「**コンテンツ**」タブで、「**表示するタスク・リストを選択します**」ドロップダウン・リストを開くと、ウィジェットのユーザーに対して使用可能にすることができるリストが表示されます。「**タスク・リストの追加**」を選択します。デプロイした照会テーブル定義が、このリストで選択できるようになっているはずです。

   その照会テーブル定義が選択できない場合は、Query Table Builder に戻り、定義ファイルが正しく定義されてデプロイされたかどうかを確認する必要があります。

Business Process Choreographer Explorer の照会テーブルの作成

EJB 照会 API の代わりに照会テーブルを使用して、Business Process Choreographer Explorer のパフォーマンスを向上させることができます。照会テーブルを作成するには、Query Table Builder を使用します。

始める前に

Query Table Builder は Eclipse プラグインとして使用可能であり、WebSphere Business Process Management SupportPacs のサイトからダウンロードできます。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リ

ンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

手順

1. Query Table Builder で、プロジェクトを右クリックしてから、「**新規**」 → 「**Business Space の複合照会定義 (Composite Query Definition for Business Space)**」を選択します。 このオプションを使用すると、Business Process Choreographer Explorer に必要なすべての列が確実に事前選択されます。
2. ウィザードの指示に従って照会テーブル定義を作成します。 必要な場合は、照会テーブル定義にさらにプロパティーを追加します。照会テーブルを定義する際には、以下の事項を検討してください。

   フィルター基準

   照会テーブルに基づいて Business Process Choreographer Explorer でビューを作成する場合、検索基準に追加のフィルターまたは変数を指定することはできません。これらのフィルター基準と、変数のパラメーターは、照会テーブルの作成時に指定する必要があります。

   照会テーブル定義でパラメーターを使用することによって、照会テーブルを Business Process Choreographer Explorer の複数のビューで使用できます。さらに柔軟性を持たせるために、カスタム・ビューの照会の実行時にパラメーターのデフォルト値を上書きできるかどうかを指定することもできます。

   許可 照会テーブルに基づいて Business Process Choreographer Explorer でビューを作成する場合、ユーザー・ロールに基づいて検索基準をフィルタリングすることはできません。ユーザー・ロールのフィルター基準は、照会テーブルの定義時に設定する必要があります。テンプレート情報に基づく 1 次照会テーブルの場合は、ロール・ベースの許可ではなく、インスタンス・ベースの許可を許可タイプとして使用します。インスタンス情報に基づく 1 次照会テーブルの場合は、適切なインスタンス・ベースの許可フィルターを指定します。

   国際化対応

   Query Table Builder でプロパティーを定義するときに、これらのプロパティーの名前と説明を別の言語で指定することもできます。カスタマイズ・ビューの照会が実行されると、Business Process Choreographer Explorer は、ブラウザーの言語設定に適した翻訳を使用します。

   照会テーブル定義の表示名および説明

   Query Table Builder では、ビューでサポートされるすべての言語で表示名と説明を指定できます。

   列の表示名および説明

   実行時に Business Process Choreographer Explorer は、結果リストに表示される列について、適切な国際化された列名を取得します。1 次照会テーブルから取り出される列 (PIID など) の場合、Business Process Choreographer Explorer は、サポートされるすべての言語について既に使用可能である翻訳を使用します。

接続照会テーブルから取り出される列 (QUERY_PROPERTY など) の場合は、Query Table Builder を使用して、ビジネスがサポートするすべての言語で表示名と説明を指定する必要があります。

タスク名および説明

WebSphere Integration Developer でタスク名と説明を国際化した場合、それらは Business Process Choreographer Explorer では、ブラウザーの言語および国の設定に従って表示されます。ブラウザーの設定が、プロセス・モデルで定義された設定と一致しない場合は、デフォルト言語の翻訳が使用されます。

ソート基準

照会テーブル定義のソート基準を定義するときに、いくつかのプロパティー (例えば、プロセス状態) は整数値として保管されますが、Business Process Choreographer Explorer ではこれらは翻訳されたストリングとして結果リストに表示されることに注意してください。これにより、一部の言語で予期しないソート結果になる可能性があります。

新規照会テーブル定義は、定義済みプロパティーと、ユーザーが定義した追加プロパティーから構成されます。

次のタスク

Query Table Builder の照会定義をアプリケーション・サーバー上にデプロイし、テストします。これが Business Process Choreographer Explorer の接続先サーバーである場合、自分の用途に合わせて、またはさまざまなユーザー・グループ用に Business Process Choreographer Explorer をカスタマイズするときに、この照会テーブルを使用できるようになります。Business Process Choreographer Explorer が別のサーバーに接続している場合は、照会テーブルを使用してカスタマイズ・ビューを作成する前に、照会テーブルを適切なサーバーにデプロイする必要があります。

関連概念

360 ページの『照会テーブルの許可』
照会テーブルで照会を実行するときには、インスタンス・ベースの許可、ロール・ベースの許可、または許可なしを使用できます。

355 ページの『照会テーブルのフィルターおよび選択基準』
フィルターおよび選択基準は、Query Table Builder を使用した照会テーブルの開発時に定義します (SQL WHERE 文節に似た構文を使用します)。これらの明確に定義されたフィルターおよび選択基準を使用して、照会テーブルの属性に基づく条件を指定します。

386 ページの『照会テーブル・メタデータの国際化対応』
照会テーブル・メタデータは国際化対応がサポートされています。

Business Process Choreographer EJB 照会 API

サービス API の query メソッドまたは queryAll メソッドを使用して、ビジネス・プロセスとタスクについて保管されている情報を取得します。

すべてのユーザーが query メソッドを呼び出すことができます。このメソッドは、作業項目が存在しているオブジェクトのプロパティーを戻します。 queryAll メソッドは、Java EE ロール BPESystemAdministrator、 TaskSystemAdministrator、BPESystemMonitor、 TaskSystemMonitor のいずれかが割り当てられているユーザーのみが呼び出すことができます。このメソッドは、データベースに格納されているすべてのオブジェクトのプロパティーを戻します。

すべての API 照会は SQL 照会にマップされます。生成される SQL 照会の形式は、次の要因によって異なります。

- 照会が、Java EE ロールが割り当てられているユーザーによって呼び出されたかどうか。
- 照会対象オブジェクト。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。
- from 文節、結合条件、およびユーザー固有のアクセス制御条件の挿入。

照会には、カスタム・プロパティーと変数プロパティーの両方を含めることができます。複数のカスタム・プロパティーまたは変数プロパティーを照会に含める場合、対応するデータベース表で自己結合が行われます。データベース・システムによっては、これらの `query()` 呼び出しによりパフォーマンスへの影響が出ることがあります。

また、createStoredQuery メソッドを使用して、Business Process Choreographer データベースに照会を保管することもできます。保管照会文を定義する場合は、照会基準を指定します。この基準は、保管照会文が実行されるとき、すなわち実行時にデータがアセンブルされるときに動的に適用されます。保管照会文にパラメーターが含まれている場合は、照会を実行するときにこれらのパラメーターも解決されます。

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。

API query メソッドの構文

Business Process Choreographer API の照会の構文は、SQL 照会の構文に似ています。照会には、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、および時間帯パラメーターを組み込むことができます。

照会の構文はオブジェクト・タイプによって異なります。以下の表は、異なるオブジェクト・タイプごとの構文を示しています。

表 59. 各種オブジェクト・タイプの照会構文

オブジェクト	構文
プロセス・テンプレート	`ProcessTemplateData[] queryProcessTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);`

表 59. 各種オブジェクト・タイプの照会構文 (続き)

オブジェクト	構文
タスク・テンプレート	`TaskTemplate[] queryTaskTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);`
ビジネス・プロセスとタスク関連データ	`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);`

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 文節の構文と似ており、コンマを使用して文節のパーツを区切ります。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 つの列を指定する必要があります。列は、ビュー名と列名を使用して完全に指定される必要があります。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列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。

複数の名前と値の対のプロパティー (カスタム・プロパティーや、照会可能な変数のプロパティーなど) を選択する場合は、ビュー名に 1 桁のカウンターを追加します。このカウンターは 1 から 9 の値を取ることができます。

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。
- "QUERY_PROPERTY1.STRING_VALUE, QUERY_PROPERTY2.INT_VALUE

  照会できる変数のプロパティーの値を取得します。これらの部分は、さらに where 文節でも指定されています。
- "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。

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 WHERE 文節の構文に似ています。文節から明示的に SQL を追加したり、API 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

  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。
- 設定操作: IN

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、複数の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' AND
 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。

結果をソートするビューの列のリストを指定できます。これらの列は、ビューと列の名前で完全に修飾されている必要があります。

order-by 文節の構文は、SQL の order-by 文節の構文と似ており、文節の各パーツをコンマで区切ります。列を昇順にソートする場合は `ASC` を指定し、列を降順にソートする場合は `DESC` を指定します。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。

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。複数の列を指定すると、照会結果セットはまず最初の列の値で順序付けされ、次に 2 番目の列の値で順序付けされる、という具合に続きます。SQL 照会のように、order-by 文節の列を、位置によって指定することはできません。

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、無視して照会結果セットで呼び出し元に戻さない照会結果セット・タプルの数を指定します。この数は、照会結果セットの先頭から数えます。

このパラメーターをしきい値パラメーターと一緒に使用してクライアント・アプリケーションでページングをインプリメントします (例えば、最初の 20 項目を取得し、それから次の 20 項目を取得し、以下同様に項目を取得します)。

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。

実動シナリオでの照会結果セットには数千から数百万の項目が含まれる可能性があるので、しきい値パラメーターの値を指定します。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。しきい値パラメーターは、例えば少数の項目のみが一度に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。

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')",
                 (String)null,
                 (Integer)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')",
                 (String)null, (Integer)null, (TimeZone)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。

照会で変数を使用することによるデータのフィルタリング:

照会結果は、照会基準に一致するオブジェクトを戻します。この結果を、変数の値でフィルタリングすることもできます。

このタスクについて

実行時にプロセスが使用する変数を、そのプロセス・モデルで定義することができます。これらの変数で、照会可能なパートを宣言します。

例えば、John Smith が保険会社のサービス番号を呼び出して、損傷を受けた車に対する保険請求の進捗状況を問い合わせるとします。請求の管理者はカスタマー ID でその請求を検索します。

手順

1. オプション: プロセス内の照会可能な変数のプロパティーをリストします。

   プロセス・テンプレート ID を使用して、プロセスを特定します。照会可能な変数がわかっている場合は、このステップはスキップしてください。

   ```
   List variableProperties = process.getQueryProperties(ptid);
   for (int i = 0; i < variableProperties.size(); i++)
   {
      QueryProperty queryData = (QueryProperty)variableProperties.get(i);
      String variableName = queryData.getVariableName();
      String name         = queryData.getName();
      int mappedType      = queryData.getMappedType();
      ...
   }
   ```
2. フィルター基準に一致する変数を持つプロセス・インスタンスをリストします。

このプロセスでは、カスタマー ID は照会可能な変数 customerClaim の一部としてモデル化されます。そのため、カスタマー ID を使用すれば問題の請求を見つけることができます。

```
QueryResultSet result = process.query
    ("PROCESS_INSTANCE.NAME, QUERY_PROPERTY.STRING_VALUE",
     "QUERY_PROPERTY.VARIABLE_NAME = 'customerClaim' AND " +
     "QUERY_PROPERTY.NAME = 'customerID' AND " +
     "QUERY_PROPERTY.STRING_VALUE like 'Smith%'",
      (String)null, (Integer)null,
      (Integer)null, (TimeZone)null );
```

このアクションによって戻される照会結果セットには、プロセス・インスタンス名と、ID が Smith で始まる顧客のカスタマー ID の値が含まれています。

照会結果:

照会結果セットには、Business Process Choreographer API 照会の結果が入ります。

結果セットのエレメントは、呼び出し元により指定された where 文節の条件を満たし、かつ呼び出し元に対し表示が許可されているオブジェクトのプロパティーです。エレメントは、API の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
String	getString
OID	getOID
タイム・スタンプ	getTimestamp getString getTimestampAsLong
Integer	getInteger getShort getLong getString getBoolean

属性タイプ	メソッド
Boolean	getBoolean getShort getInteger getLong getString
byte[]	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",
                                         (String)null, (String)null,
                                         (Integer)null, (TimeZone)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"
```

ユーザー固有のアクセス条件

SQL SELECT ステートメントが API 照会から生成されるときに、ユーザー固有のアクセス条件が追加されます。この条件により、呼び出し元により指定されている条件に一致し、呼び出し元に対し許可されているオブジェクトのみが呼び出し元に戻されます。

追加されるアクセス条件は、ユーザーがシステム管理者であるかどうかによって異なります。

システム管理者以外のユーザーが呼び出した照会

生成される SQL WHERE 文節では、API where 文節と、ユーザー固有のアクセス制御条件が結合されます。この照会は、ユーザーに対しアクセスが許可されている

オブジェクト、つまりユーザーが作業項目を所有しているオブジェクトのみを取得します。作業項目とは、ビジネス・オブジェクト (タスクやプロセスなど) の許可ロールへのユーザーまたはユーザー・グループの割り当てを表します。例えば、ユーザー John Smith が特定のタスクの潜在的所有者ロールのメンバーである場合、この関係を表す作業項目オブジェクトがあります。

例えば、グループ作業項目が無効な場合に、システム管理者以外のユーザーがタスクを照会すると、以下のアクセス条件が WHERE 文節に追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'user'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

したがって John Smith が、自身が潜在的所有者であるタスクのリストを取得する場合、API where 文節は次のようになります。

```
"WORK_ITEM.REASON == WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"
```

この API where 文節により、SQL ステートメントに次のアクセス条件が追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'JohnSmith'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
AND WI.REASON = 1
```

つまり、John Smith が、自身がプロセス・リーダーまたはプロセス管理者であるアクティビティーやタスクと、作業項目を所有していないアクティビティーやタスクを表示するには、 PROCESS_INSTANCE ビューのプロパティー (PROCESS_INSTANCE.PIID など) を照会の select、where、または order-by 文節に追加する必要があります。

グループ作業項目が有効な場合は、ユーザーに対し、グループがアクセスできるオブジェクトへのアクセスを許可するアクセス条件が WHERE 文節に追加されます。

システム管理者が呼び出した照会

システム管理者は、query メソッドを呼び出し、作業項目が関連付けられているオブジェクトを取得できます。この場合、生成される SQL 照会には WORK_ITEM ビューとの結合が追加されますが、WORK_ITEM.OWNER_ID のアクセス制御条件は追加されません。

この場合、タスクの SQL 照会には以下が含まれます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
```

queryAll 照会

このタイプの照会を呼び出すことができるのは、システム管理者またはシステム・モニターのみです。アクセス制御条件も WORK_ITEM ビューとの結合も追加されません。このタイプの照会では、すべてのオブジェクトの全データが戻されます。

query メソッドと queryAll メソッドの例

以下の例は、標準的な各種 API 照会の構文と、照会の実行時に生成される関連 SQL ステートメントを示します。

例: 作動可能状態のタスクの照会:

この例では、query メソッドを使用して、ログオン・ユーザーが作業可能なタスクを取得する方法を示します。

John Smith は、自分に割り当てられているタスクを一覧表示します。ユーザーがタスクの作業を行うには、そのタスクが作動可能状態になっている必要があります。ログオン・ユーザーには、そのタスクの潜在的所有者作業項目も必要です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND    WI.REASON = 1
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

API 照会を特定のプロセスのタスク (sampleProcess など) に限定する場合、この照会は次のようになります。

```
query( "DISTINCT TASK.TKIID",
           "PROCESS_TEMPLATE.NAME = 'sampleProcess' AND "+
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

例: 要求済み状態のタスクの照会:

この例では、query メソッドを使用して、ログオン・ユーザーが要求したタスクを取得する方法を示します。

ユーザー John Smith は、自身が要求したタスクのうち、まだ要求済み状態であるタスクを検索します。「John Smith により要求された」ことを指定する条件は TASK.OWNER = 'JohnSmith' です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "TASK.OWNER = 'JohnSmith'",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   TA.OWNER = 'JohnSmith'
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

タスクが要求されると、タスクの所有者に対して作業項目が作成されます。したがって、John Smith が要求したタスクを取得する照会のもう 1 つの作成方法として、TASK.OWNER = 'JohnSmith' を使用する代わりに、次の条件を照会に追加する方法があります。

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

照会は、次のコード・スニペットのようになります。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   AND    WI.REASON = 4
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

John は休暇に入るため、所属するチームのリーダーである Anne Grant が、John の現在の作業割り当てを確認するとします。Anne にはシステム管理者権限が付与されています。呼び出す照会は、John が呼び出した照会と同じです。ただし Anne は管理者であるため、生成される SQL ステートメントが異なります。生成される SQL ステートメントを次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  TA.TKIID = WI.OBJECT_ID =
   AND    TA.STATE = 8
   AND   TA.OWNER = 'JohnSmith')
```

Anne は管理者であるため、アクセス制御条件が WHERE 文節に追加されません。

例: エスカレーションの照会:

この例では、query メソッドを使用して、ログオン・ユーザーのエスカレーションを取得する方法を示します。

タスクがエスカレートされると、エスカレーション受信者作業項目が作成されます。ユーザー Mary Jones が、Mary 自身にエスカレートされたタスクのリストを表示するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT ESCALATION.ESIID, ESCALATION.TKIID",
         "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_ESCALATION_RECEIVER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
 SELECT DISTINCT ESCALATION.ESIID, ESCALATION.TKIID
   FROM   ESCALATION ESC, WORK_ITEM WI
   WHERE  ESC.ESIID = WI.OBJECT_ID
   AND    WI.REASON  = 10
   AND
  ( WI.OWNER_ID = 'MaryJones' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

例: queryAll メソッドの使用:

この例では、queryAll メソッドを使用して、1 つのプロセス・テンプレートに属するアクティビティーをすべて取得する方法を示します。

queryAll メソッドは、システム管理者権限またはシステム・モニター権限が付与されているユーザーだけが使用できます。プロセス・テンプレート sampleProcess に属するすべてのアクティビティーを取得する照会の queryAll メソッド呼び出しを、次のコード・スニペットに示します。

```
queryAll( "DISTINCT ACTIVITY.AIID",
       "PROCESS_TEMPLATE.NAME = 'sampleProcess'",
       (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL 照会を、次のコード・スニペットに示します。

```
 SELECT DISTINCT ACTIVITY.AIID
   FROM   ACTIVITY AI, PROCESS_TEMPLATE PT
   WHERE  AI.PTID = PT.PTID
   AND    PT.NAME = 'sampleProcess'
```

この呼び出しは管理者により実行されるため、生成される SQL ステートメントにアクセス制御条件は追加されません。WORK_ITEM ビューとの結合も追加されません。つまり、この照会では、プロセス・テンプレートのすべてのアクティビティー (作業項目のないアクティビティーを含む) が取得されます。

例: 照会への照会プロパティーの組み込み:

この例では、query メソッドを使用して、ビジネス・プロセスに属するタスクを取得する方法を示します。このプロセスに対して定義されている照会プロパティーを、検索に組み込むとします。

例えば、1 つのビジネス・プロセスに属し、作動可能状態にあるヒューマン・タスクをすべて検索するとします。プロセスには照会プロパティー **customerID** とその値 CID_12345、および名前空間があります。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY.NAME = 'customerID' AND " +
          " QUERY_PROPERTY.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY.NAMESPACE =
             'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

2 番目の照会プロパティー (**Priority** など) と特定の名前空間を照会に追加する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY1.NAME = 'customerID' AND " +
          " QUERY_PROPERTY1.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY1.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " QUERY_PROPERTY2.NAME = 'Priority' AND " +
          " QUERY_PROPERTY2.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

複数の照会プロパティーを照会に追加する場合は、コード・スニペットに示されているように、追加する各プロパティーに番号を付ける必要があります。ただし、カスタム・プロパティーの照会を実行するとパフォーマンスに影響します。照会に含まれているカスタム・プロパティーの数に応じてパフォーマンスが低下します。

例: 照会へのカスタム・プロパティーの組み込み:

この例では、query メソッドを使用して、カスタム・プロパティーが指定されたタスクを取得する方法を示します。

例えば、カスタム・プロパティー **customerID** とその値 CID_12345 が指定されており、作動可能状態にあるヒューマン・タスクをすべて検索するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID ",
          " TASK_CPROP.NAME = 'customerID' AND " +
          " TASK_CPROP.STRING_VALUE = 'CID_12345' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

タスクとそのカスタム・プロパティーを取得する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_CPROP.NAME, TASK_CPROP.STRING_VALUE",
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TA.TKIID , TACP.NAME , TACP.STRING_VALUE
   FROM  TASK TA LEFT JOIN TASK_CPROP TACP ON (TA.TKIID = TACP.TKIID),
         WORK_ITEM WI
   WHERE WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND  (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID IS NULL AND WI.EVERYBODY = 1 )
```

この SQL ステートメントには、TASK ビューと TASK_CPROP ビューの外部結合が含まれています。つまり、WHERE 文節の条件を満たすタスクは、カスタム・プロパティーが含まれていない場合でも取得されます。

保管照会文の管理

保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

このタスクについて

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。専用の保管照会文と共通の保管照会文の名前を同じにすることができます。異なる複数の所有者の専用保管照会文を同じ名前にすることもできます。

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。

関連概念

『保管照会文のパラメーター』

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。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

保管照会文のパラメーター:

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。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

例えば、顧客名を保管するカスタム・プロパティーを定義したとします。特定の顧客 ACME Co. に関連したタスクを戻すように、照会を定義することができます。この情報を照会する場合、照会内の where 文節は以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = 'ACME Co.'";
```

顧客 BCME Ltd. も検索できるようにこの照会を再使用可能にするには、カスタム・プロパティーの値に対してパラメーターを使用できます。パラメーターをタスク照会に追加する場合、以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = '@param1'";
```

`@param1` パラメーターは、query メソッドに受け渡されるパラメーターのリストから、実行時に解決されます。以下の規則は、照会内でのパラメーターの使用に適用されます。

- パラメーターは where 文節でのみ使用できる。
- パラメーターはストリングである。
- パラメーターは実行時にストリングの置換を使用して置き換えられる。特殊文字が必要な場合、where 文節に指定するか、実行時にパラメーターの一部として受け渡す必要があります。
- パラメーター名は、ストリング `@param` を整数と連結したもので構成される。最小番号は 1 で、実行時に照会 API に受け渡されるパラメーターのリスト内の最初の項目を指します。
- パラメーターは where 文節内で複数回使用することができ、出現するすべてのパラメーターは同じ値で置き換えられます。

関連タスク

408 ページの『保管照会文の管理』
保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

共通保管照会文の管理:

共通保管照会文がシステム管理者によって作成されます。この照会は、全ユーザーが使用できます。

このタスクについて

システム管理者は、共通保管照会文を作成、表示、および削除できます。API 呼び出しでユーザー ID を指定しないと、その保管照会文は共通保管照会文と見なされます。

手順

1. 共通の保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、CustomerOrdersStartingWithA という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",
                (Integer)null, (TimeZone)null);
   ```

   この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

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), null);
   ```

   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. 使用可能な共通保管照会文の名前をリストします。

   以下のコードの断片では、戻される照会のリストを共通照会のみに限定する方法を示しています。

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

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。

   専用の保管照会文には、共通の保管照会文と同じ名前を付けることができます。名前が同じである場合は、専用の保管照会文が戻されます。以下のコードの断片では、指定した名前の共通照会のみを戻す方法を示しています。Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

   ```
   StoredQueryData storedQuery = process.getStoredQuery
      (StoredQueryData.KIND_PUBLIC, "CustomerOrdersStartingWithA");
   String selectClause = storedQuery.getSelectClause();
   String whereClause = storedQuery.getWhereClause();
   String orderByClause = storedQuery.getOrderByClause();
   Integer threshold = storedQuery.getThreshold();
   String owner = storedQuery.getOwner();
   ```

5. 共通の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");
   ```

他のユーザーの専用保管照会文の管理:

専用照会はどのユーザーでも作成できます。この照会は、照会の所有者とシステム管理者しか使用できません。

このタスクについて

システム管理者は、特定ユーザーに属する専用の保管照会文を管理できます。

手順

1. ユーザー ID Smith の専用保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、Smith というユーザー ID で CustomerOrdersStartingWithA という名前を付けて保管します。

   ```
   process.createStoredQuery("Smith", "CustomerOrdersStartingWithA",
               "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
               "PROCESS_INSTANCE.NAME LIKE 'A%'",
               "PROCESS_INSTANCE.NAME",
                (Integer)null, (TimeZone)null,
                (List)null, (String)null);
   ```

   この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

2. 保管照会文で定義された照会を実行します。

   ```
   QueryResultSet result = process.query
                         ("Smith", "CustomerOrdersStartingWithA",
                           (Integer)null, (Integer)null, (List)null);
   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. 特定のユーザーに属する専用照会の名前のリストを取得します。

   例えば、以下のコードの断片では、ユーザー Smith に属する専用照会のリストを取得する方法を示しています。

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

4. 特定の照会の詳細を表示します。

   以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

   ```
   StoredQueryData storedQuery = process.getStoredQuery
      ("Smith", "CustomerOrdersStartingWithA");
   String selectClause = storedQuery.getSelectClause();
   String whereClause = storedQuery.getWhereClause();
   String orderByClause = storedQuery.getOrderByClause();
   Integer threshold = storedQuery.getThreshold();
   String owner = storedQuery.getOwner();
   ```

   Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

5. 専用の保管照会文を削除します。

   以下のコードの断片では、ユーザー Smith が所有する専用照会を削除する方法を示しています。

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

専用保管照会文の操作:

システム管理者でなくても、自分専用の保管照会文は作成、実行、および削除できます。また、システム管理者が作成した共通の保管照会文を使用することもできます。

手順

1. 専用の保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、固有の名前を付けて保管します。ユーザー ID が指定されない場合、その保管照会文はログオン・ユーザーの専用保管照会文と見なされます。

   ```
   process.createStoredQuery("CustomerOrdersStartingWithA",
               "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
               "PROCESS_INSTANCE.NAME LIKE 'A%'",
               "PROCESS_INSTANCE.NAME",
                (Integer)null, (TimeZone)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. ログオン・ユーザーがアクセスできる保管照会文の名前のリストを取得します。

   以下のコードの断片では、ユーザーがアクセスできる共通の保管照会文と専用の保管照会文の両方を取得する方法を示しています。

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

4. 特定の照会の詳細を表示します。

   以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

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

   Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

5. 専用の保管照会文を削除します。

   以下のコード断片は、専用保管照会文を削除する方法を示しています。

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

ビジネス・プロセスおよびヒューマン・タスク用 EJB クライアント・アプリケーションの開発

EJB API は、WebSphere Process Server 上にインストールされているビジネス・プロセスやヒューマン・タスクを処理する EJB クライアント・アプリケーションを開発するための汎用的な方法をいくつか提供します。

このタスクについて

この Enterprise JavaBeans (EJB) API を使用すれば、以下を実行するためのクライアント・アプリケーションを作成できます。

- プロセスやタスクのライフ・サイクルの、開始から完了後の削除までの管理
- アクティビティーやプロセスの修復
- ワークグループのメンバーに対するワークロードの管理および配布

EJB 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。

EJB 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。

以下のステップは、EJB アプリケーションの開発に必要なアクションの概要です。

手順

1. アプリケーションが提供する機能を決定します。
2. 使用するセッション Bean 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には、アプリケーションに組み込まれたメソッドを呼び出し、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ための、適切な許可のロールが割り当てられている必要があります。該当するセッション Bean のインスタンスを作成するときに、WebSphere Application Server がコンテキストとそのインスタンスを関連付けます。コンテキストには、呼び出し元のプリンシパル ID、グループ・メンバーシップ・リスト、およびロールについての情報が含まれています。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。
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。

   EJB 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. EJB API にアクセスします。
b. EJB 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。

関連概念

代替管理許可モード

関連資料

445 ページの『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。

465 ページの『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。

EJB API へのアクセス

Enterprise JavaBeans (EJB) 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。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・エンタープライズ Bean にアクセスします。

このタスクについて

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、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。このアプリケーションは、別の Enterprise JavaBeans (EJB) アプリケーションを含む任意の Java アプリケーションを指します。

セッション Bean のリモート・インターフェースにアクセスする

ビジネス・プロセスまたはヒューマン・タスク用の EJB クライアント・アプリケーションでは、Bean のリモート・ホーム・インターフェースを介して、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。

このタスクについて

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。

手順

1. セッション Bean のリモート・インターフェース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 Platform Enterprise Edition (Java EE) 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. ビジネス・オブジェクトの定義をどのように提供するかを決定します。

   リモート・クライアント・アプリケーション内のビジネス・オブジェクトを処理する場合は、プロセスまたはタスクとの対話に使用されるビジネス・オブジェクトの対応スキーマ (XSD または WSDL ファイル) にアクセスできる必要があります。これらのファイルにアクセスできるようにするには、次のいずれかの方法を行います。

   - クライアント・アプリケーションが管理対象 Java EE 環境で稼働しない場合は、これらのファイルをクライアント・アプリケーションの EAR ファイルにパッケージします。
   - クライアント・アプリケーションが管理対象 Java EE 環境の Web アプリケーションまたは EJB クライアントである場合は、これらのファイルをクライアント・アプリケーションの EAR ファイルにパッケージするか、リモート成果物ロードを利用します。
     a. Business Process Choreographer EJB API の createMessage および ClientObjectWrapper.getObject メソッドを使用して、サーバー上の対応アプリケーションからリモート・ビジネス・オブジェクト定義を透過的にロードします。
     b. サービス・データ・オブジェクトのプログラミング API を使用して、既にインスタンス化されたビジネス・オブジェクトの一部としてビジネス・オブジェクトの作成または読み取りを実行します。これを行うには、DataObject インターフェースで commonj.sdo.DataObject.createDataObject メソッドまたは getDataObject メソッドを使用します。
     c. XML スキーマ any または anyType を使用して型付けされるビジネス・オブジェクトのプロパティー値としてビジネス・オブジェクトを作成する場合は、ビジネス・オブジェクト・サービスを使用してビジネス・オブジェクトの作成または読み取りを実行します。これを行うには、スキーマのロード元となるアプリケーションを指すようにリモート成果物ローダーのコ

ンテキストを設定する必要があります。これにより、適切なビジネス・オブジェクト・サービスを使用できるようになります。

例えば、ビジネス・オブジェクトを作成します。ここで "ApplicationName" は、ビジネス・オブジェクト定義が含まれるアプリケーションの名前です。

```
BOFactory bofactory = (BOFactory) new
   ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");

com.ibm.wsspi.al.ALContext.setContext
   ("RALTemplateName", "ApplicationName");
try {
     DataObject dataObject = bofactory.create("uriName", "typeName" );
   } finally {
     com.ibm.wsspi.al.ALContext.unset();
   }
```

例えば、XML 入力を読み取ります。ここで "ApplicationName" は、ビジネス・オブジェクト定義が含まれるアプリケーションの名前です。

```
BOXMLSerializer serializerService =
  (BOXMLSerializer) new ServiceManager().locateService
    ("com/ibm/websphere/bo/BOXMLSerializer");
ByteArrayInputStream input = new ByteArrayInputStream(<?xml?>..);

com.ibm.wsspi.al.ALContext.setContext
          ("RALTemplateName", "ApplicationName");
try {
     BOXMLDocument document = serializerService.readXMLDocument(input);
     DataObject dataObject = document.getDataObject();
   } finally {
     com.ibm.wsspi.al.ALContext.unset();
   }
```

3. Java Naming and Directory Interface (JNDI) からセッション Bean のリモート・ホーム・インターフェースを見つけます。

   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 のリモート・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();
```

   セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そ

のアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer Java EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、管理セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。管理セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

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("java:comp/UserTransaction");

     // Begin a transaction
     transaction.begin();

     // Applications calls ...

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

   ヒント: データベース・ロック競合を防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

   ```
   // Obtain user transaction interface
   UserTransaction transaction=
          (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

   transaction.begin();

   //read the activity instance
   process.getActivityInstance(aiid);
   //claim the activity instance
   process.claim(aiid);

   transaction.commit();
   ```

   getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされます。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

例

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);
```

セッション Bean のローカル・インターフェースにアクセスする

ビジネス・プロセスまたはヒューマン・タスク用の EJB クライアント・アプリケーションでは、Bean のローカル・ホーム・インターフェースを介して、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。

このタスクについて

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、ヒューマン・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。

手順

1. セッション Bean のローカル・インターフェース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 Platform Enterprise Edition (Java EE) 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) からセッション Bean のローカル・ホーム・インターフェースを見つけます。

   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 BusinessFlowManager bean

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

   セッション Bean のローカル・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. 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();
   ```

   セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そのアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer Java EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、管理セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。管理セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

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("java:comp/UserTransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

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

ヒント: データベース・デッドロックを防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされます。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

例

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

...
//Access the local interface of the 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。プロセスに対する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。

このタスクについて

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、同期して実行される短期実行のビジネス・プロセスです。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。プロセスで特定の構成要素を使用するとプロセス・フローが中断し、例えば、ヒューマン・タスクの呼び出し、同期バインディングを使用したサービスの呼び出し、またはタイマー駆動アクティビティーの使用などが割り込みます。

  プロセスの並列分岐は通常、非同期でナビゲートされるので、並列分岐のアクティビティーは平行して実行されます。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。

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

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、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

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		
	読者	スターター	管理者
createMessage	x	x	x
createWorkItem			x
delete			x
deleteWorkItem			x
forceTerminate			x
getActiveEventHandlers	x		x
getActivityInstance	x		x
getAllActivities	x		x
getAllWorkItems	x		x
getClientUISettings	x	x	x
getCustomProperties	x	x	x
getCustomProperty	x	x	x

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

注: プロセス管理がシステム管理者に制限されている場合、インスタンス・ベースの管理は使用不可です。すなわち、プロセス、スコープ、およびアクティビティーでの管理アクションが、BPESystemAdministrator ロールを持つユーザーに制限されます。さらに、プロセス・インスタンスまたはその一部の読み取り、表示、およびモニターは、ユーザーが BPESystemAdministrator ロールまたは BPESystemMonitor ロールを持つ場合のみ実行できます。この管理モードについて詳しくは、doc/bpc/cnot_instance_based_admin.ditaを参照してください。

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

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を保証するものではありません。呼び出し元は、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

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				
	読者	編集者	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

アクション	呼び出し元のプリンシパルのロール				
	読者	編集者	潜在的な所有者	所有者	管理者
forceComplete					x
forceRetry					x
getActivityInstance	x	x	x	x	x
getAllWorkItems	x	x	x	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
getVariable	x	x	x	x	x
getVariableNames	x	x	x	x	x
getInputVariableNames	x	x	x	x	x
getOutputVariableNames	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

注: プロセス管理がシステム管理者に制限されている場合、インスタンス・ベースの管理は使用不可です。すなわち、プロセス、スコープ、およびアクティビティーでの管理アクションが、BPESystemAdministrator ロールを持つユーザーに制限されます。さらに、プロセス・インスタンスまたはその一部の読み取り、表示、およびモニターは、ユーザーが BPESystemAdministrator ロールまたは BPESystemMonitor ロールを持つ場合のみ実行できます。この管理モードについて詳しくは、doc/bpc/cnot_instance_based_admin.ditaを参照してください。

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。ライフ・サイクルを管理するために、プロセス・インスタンスでさまざまなアクションを実行できます。

このタスクについて

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。

このタスクについて

microflow や長期実行プロセスを開始する標準の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。

このタスクについて

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。

手順

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),
     (TimeZone)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。

このタスクについて

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。

手順

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),
     (TimeZone)null);
   ```

   結果は名前でソートされます。microflow 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。

手順

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.EXECUTION_MODE_LONG_RUNNING",
   "PROCESS_TEMPLATE.NAME",
    new Integer(50),
    (TimeZone)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、および編集者が決定され、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 human task、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。

手順

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.EXECUTION_MODE_LONG_RUNNING",
   "PROCESS_TEMPLATE.NAME",
    new Integer(50),
    (TimeZone)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.sendMessage(activity.getServiceTemplateID(),
                             activity.getActivityTemplateID(),
                             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、および編集者が決定され、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 human task、 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。

ビジネス・プロセスの中断と再開:

長期にわたって実行するトップレベルのプロセス・インスタンスを実行中に中断し、再開して完了することができます。

始める前に

このタスクについて

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。また、プロセスを中断し、プロセス・インスタンスの失敗の原因となっている問題を修正して、問題が修正されたら再開することもできます。

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。呼び出し元はプロセス管理者またはシステム管理者でなければなりません。ただし、Business Flow Manager で代替プロセス管理許可モードを使用している場合には、プロセス管理がシステム管理者に制限されており、このアクションを実行できるのは BPESystemAdministrator ロール内の呼び出し元のみになります。

手順

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。この状態では、開始されたアクティビティーはまだ完了することはできますが、新規のアクティビティーはアクティブ化されません。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。このプロセス・インスタンスに関連付けられているインライン・タスクおよびスタンドアロン・タスクは中断されません。

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。

このタスクについて

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。

呼び出し元はプロセス管理者またはシステム管理者でなければなりません。ただし、Business Flow Manager で代替プロセス管理許可モードを使用している場合には、プロセス管理がシステム管理者に制限されており、このアクションを実行できるのは BPESystemAdministrator ロール内の呼び出し元のみになります。

手順

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:

場合によっては、リカバリー不能状態にある最上位プロセス・インスタンスを強制終了する必要があります。

このタスクについて

このアクションを実行するには、呼び出し元はプロセス管理者またはシステム管理者でなければなりません。ただし、Business Flow Manager で代替プロセス管理許可モードを使用している場合には、プロセス管理がシステム管理者に制限されており、このアクションを実行できるのは BPESystemAdministrator ロール内の呼び出し元のみになります。

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、このアクションは例外的な場合にのみ実行してください。

手順

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);
```

補正を使用してプロセス・インスタンスを終了する場合、プロセスの補正は、障害が最上位スコープで発生したかのように実行されます。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、予定タスク、またはインライン呼び出しタスクが正常に終了するのを待たずに、即時に終了されます。

プロセスおよびプロセスに関連するスタンドアロン・タスクによって開始されるアプリケーションは、強制終了要求によって終了されません。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。例えば、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などは、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、格納されたプロセス・インスタンスのデータは、ディスク・スペースとパフォーマンスに影響を与えるだけでなく、同じ相関セット値を使用するプロセス・インスタンスも作成されなくなります。したがって、プロセス・インスタンス・データは、データベースから定期的に削除してください。

このタスクについて

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。

手順

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",
                       (String)null, (Integer)null, (TimeZone)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);
}
```

このアクションにより、選択したプロセス・インスタンスとそのインライン・タスクがデータベースから削除されます。

ヒューマン・タスク・アクティビティーの処理

ビジネス・プロセス内のヒューマン・タスク・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。

このタスクについて

ヒューマン・タスク・アクティビティーが活動状態にされると、アクティビティー・インスタンスと、関連した予定タスクの両方が作成されます。ヒューマン・タスク・アクティビティーおよび作業項目管理の処理は、Human Task Manager に委任されます。アクティビティー・インスタンスの状態変更はすべてタスク・インスタンスに反映され、その反対にタスク・インスタンスの状態変更はアクティビティー・インスタンスに反映されます。

潜在的な所有者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。

手順

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",
                    (String)null, (Integer)null, (TimeZone)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 human task 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, myFault,(String)faultNames.get(0) );
```

   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 FALSE に設定されているときに、周囲の有効範囲で障害がキャッチされない場合、そのアクティビティーは停止状態になります。この状態では、強制完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。

関連概念

 アクティビティーおよびビジネス・プロセスのエラーの継続動作
予期しない障害が発生したがその障害に対して障害ハンドラーが定義されていない場合に、どのような処理が行われるかを、ビジネス・プロセスを定義するときに指定できます。プロセスの定義時に「**エラーの継続**」設定を使用して、障害発生箇所でプロセスが停止するように指定することができます。

単独ユーザー・ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 initiateAndClaimFirst API および completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。この例では、クライアント・サイド・ページ・フローを使用した単独ユーザー・ワークフローの実装を示します。

このタスクについて

単独ユーザー・ワークフローは、ページ・フロー または*画面フロー* とも呼ばれます。以下の 2 種類のページ・フローがあります。

- クライアント・サイド・ページ・フロー。このフローでは、複数ページの Lotus® Forms フォームなどのクライアント・サイド・テクノロジーを使用して、異なるページ間のナビゲーションを実現しています。
- サーバー・サイド・ページ・フロー。このフローは、ビジネス・プロセスと、後続のタスクが同じ担当者に割り当てられるようにモデル化された一連のヒューマン・タスクを使用して実現されます。

サーバー・サイド・ページ・フローはクライアント・サイド・ページ・フローよりも強力ですが、処理のために消費するサーバー・リソースは増えます。したがって、このタイプのワークフローは、主として次の状態の場合に使用することを検討してください。

- ユーザー・インターフェースで実行された手順間でサービスを呼び出す必要がある場合 (データの検索や更新など)。
- ユーザー・インターフェースでの対話が完了した後に CEI イベントの書き込みを要求する監査要件が存在する場合。

単独ユーザー・ワークフローの典型的な例は、オンライン・ブックストアでの注文プロセスです。この場合、購入者は書籍を注文するための一連の操作を実行します。この一連の操作は、ヒューマン・タスク・アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文することを決定した場合、これが注文プロセスの開始と次のヒューマン・タスク・アクティビティーの要求に相当します。

initiateAndClaimFirst API は、ページ・フローを開始します。すなわち、指定されたプロセスを開始し、アクティビティー・シーケンス内の最初のヒューマン・タスク・アクティビティーを要求します。そして、要求したアクティビティーの情報 (処理される入力メッセージなど) を戻します。

completeAndClaimSuccessor API はヒューマン・タスク・アクティビティーを完了し、ログオン・ユーザーの同じプロセス・インスタンス内の次のアクティビティーを要求します。そして、次に要求したアクティビティーの情報 (処理される入力メッセージなど) を戻します。次のアクティビティーは、完了したアクティビティーと同じトランザクション内で使用可能になるため、プロセス・モデル内のすべてのヒューマン・タスク・アクティビティーのトランザクション動作が participates に設定される必要があります。

この例を、Business Flow Manager API と Human Task Manager API の両方を使用する例と比較してください。

手順

1. 書籍注文プロセスを開始し、アクティビティー・シーケンス内の最初のアクティビティーを要求します。 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。
   a. プロセス・テンプレートを取得して、該当するタイプの入力メッセージを作成します。

      ```
      ProcessTemplateData template = process.getProcessTemplate("CustomerOrder");
      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");
      }
      ```

   b. プロセスを開始し、最初のヒューマン・タスク・アクティビティーを要求します。

      ```
      InitiateAndClaimFirstResult result =
        process.initiateAndClaimFirst("CustomerOrder", "MyOrderProcess", input);
      AIID aiid = result.getAIID();
      ClientObjectWrapper input = result.getInputMessage();
      ```

   最初のアクティビティーが要求されると、要求されたアクティビティーの入力メッセージと ID が返されます。
2. アクティビティーの作業が終了したら、そのアクティビティーを完了して次のアクティビティーを要求します。

   アクティビティーを完了するには、出力メッセージを渡します。出力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。

   ```
   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 and claim the next one
CompleteAndClaimSuccessorResult successor =
   process.completeAndClaimSuccessor(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、シーケンス内の次のアクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者であるヒューマン・タスク・アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

3. 次のアクティビティーを処理します。

```
String name = successor.getActivityName();

ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

4. アクティビティーを完了する場合は、ステップ 2 に進みます。

関連タスク

471 ページの『ヒューマン・タスクを含む単一の個人ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、サーバー・サイド・ページ・フローを使用した単一の個人ワークフローを実装する方法を示します。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

待機中のアクティビティーへのメッセージの送信

インバウンド・メッセージ・アクティビティー (receive アクティビティー、pick アクティビティーの onMessage、イベント・ハンドラーの onEvent) を使用して、実行中のプロセスを「外の世界」からのイベントと同期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。

このタスクについて

親タスクを使用して、アクティビティーにメッセージを送信できます。

手順

1. 特定のプロセス・インスタンス ID を持つプロセス・インスタンスのログオンしたユーザーからのメッセージを待っ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。

   ```
   ActivityServiceTemplateData[] services = process.getWaitingActivities(piid);
   ```

2. 最初の待機サービスにメッセージを送信します。

   最初のサービスを、ユーザーがサービスを提供しようとするサービスと想定します。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スターター、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。

   ```
   VTID vtid = services[0].getServiceTemplateID();
   ATID atid = services[0].getActivityTemplateID();
   String inputType = services[0].getInputMessageTypeName();

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

このタスクについて

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。

手順

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

プロセスは、Web Services Description Language (WSDL) の片方向操作または要求/応答操作としてモデル化される Web サービス操作を公開できます。片方向インターフェースを使用する長期実行プロセスの結果は、そのプロセスに出力がないため、getOutputMessage メソッドを使用して取り出すことはできません。ただし、代わりに変数の内容を照会できます。

このタスクについて

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。

手順

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",
                         (String)null, (Integer)null, (TimeZone)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");
  }
}
```

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、プロセス管理者は、プロセスのナビゲーションを継続できるように、いくつかの方法でアクティビティーを処理することができます。

このタスクについて

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。

このタスクについて

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。

ヒューマン・タスク・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。

手順

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'",
                    (String)null, (Integer)null, (TimeZone)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、停止したヒューマン・タスク・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。エラーが発生すると、**continueOnError** パラメーターに基づいて、障害が forceComplete 要求によって発生する場合に実行されるアクションが決まります。

   例では、**continueOnError** が true です。この値は、障害が発生した場合にアクティビティーが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。

関連概念


アクティビティーおよびビジネス・プロセスのエラーの継続動作
予期しない障害が発生したがその障害に対して障害ハンドラーが定義されていない場合に、どのような処理が行われるかを、ビジネス・プロセスを定義するときに指定できます。プロセスの定義時に「**エラーの継続**」設定を使用して、障害発生箇所でプロセスが停止するように指定することができます。

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。

このタスクについて

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ともに、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。

手順

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'",
                     (String)null, (Integer)null, (TimeZone)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、停止したヒューマン・タスク・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、プロセス状態が失敗状態で終了する前にプロセス・レベルの障害ハンドラーが実行されます。

関連概念

アクティビティーおよびビジネス・プロセスのエラーの継続動作
予期しない障害が発生したがその障害に対して障害ハンドラーが定義されていない場合に、どのような処理が行われるかを、ビジネス・プロセスを定義するときに指定できます。プロセスの定義時に「**エラーの継続**」設定を使用して、障害発生箇所でプロセスが停止するように指定することができます。

結合、ループ、またはカウンター評価の失敗により停止したアクティビティーの修復:

結合またはループ条件、あるいは forEach カウンター値の評価時に例外が発生したためにアクティビティーが停止することがあります。管理者は、評価がまた失敗する可能性があるなどの理由から、アクティビティーの実行を再試行しないことに決めます。このような場合、Business Process Choreographer EJB API を使用して式に適切な値を指定することにより、プロセスのナビゲーションを続行できます。

このタスクについて

任意のタイプのアクティビティーに対する結合条件の値や、while または repeat-until アクティビティーのループ条件の値を設定できます。開始カウンターと最終カウンターの値、および forEach アクティビティーの完了分岐の最大数を設定することもできます。完了分岐に設定する値は、プロセス・モデルの forEach アクティビティーの定義によって異なります。モデルに早期終了条件が指定されている場合は、完了分岐の最大数の値を設定します。早期終了条件が指定されていない場合は、完了分岐の最大数の値を `null` に設定します。

以下のサンプルに、ループ条件の値を設定する方法を示します。

手順

1. ループ条件の評価が失敗したために停止した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
   ACTIVITY.STOP_REASON = ACTIVITY.STOP_REASON.STOP_REASON_IMPLEMENTATION_FAILED AND
   (ACTIVITY.KIND = ACTIVITY.KIND.KIND_WHILE OR
      ACTIVITY.KIND = ACTIVITY.KIND.KIND_REPEAT_UNTIL) AND
   PROCESS_INSTANCE.NAME='CustomerOrder'",
   (String)null, (Integer)null, (TimeZone)null );
```

   同様に、結合条件または forEach カウンターの評価が失敗したために停止したアクティビティーもリストできます。

   - 結合条件の失敗については、以下の式を使用します。

```
ACTIVITY.STOP_REASON.STOP_REASON_ACTIVATION_FAILED
```

- forEach カウンターの失敗については、次の式を使用します。

```
ACTIVITY.STOP_REASON.STOP_REASON_IMPLEMENTATION_FAILED AND
(ACTIVITY.KIND = ACTIVITY.KIND.KIND_FOR_EACH_SERIAL OR
   ACTIVITY.KIND = ACTIVITY.KIND.KIND_FOR_EACH_PARALLEL)
```

このアクションにより、ループ条件の評価が失敗したために停止した CustomerOrder プロセス・インスタンスのアクティビティーが返されます。

2. ループ条件の値 (true など) を指定します。

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

  process.forceLoopCondition(aiid, true);
}
```

このアクションにより、アクティビティーのループ条件の値が true に設定され、プロセス・インスタンスのナビゲーションが続行されます。

同様に、結合条件の値 (`process.forceJoinCondition(aiid, true);`) または forEach アクティビティー・カウンターの値 (`process.forceForEachCounterValues(aiid, 1, 5, new Integer(2));`) を設定できます。

停止したアクティビティーに関連付けられた相関セットの更新:

相関セットは、Web サービス間のステートフル・コラボレーションをサポートするために使用されます。このような場合、Business Process Choreographer EJB API を使用して式に適切な値を指定することにより、プロセスのナビゲーションを続行できます。

このタスクについて

停止状態にあるアクティビティーでは、以下のいずれかの理由により、関連する相関セットの更新が必要になる可能性があります。

- 相関セットの評価時に例外が発生しました。相関セットは初期化することになっていますが、相関セットが既に初期化されています。
- 相関セットの評価時に例外が発生しました。相関セットは初期化しないことになっていますが、相関セットの値が設定されていません。これは、例えば、初期化アクティビティーがスキップされたために発生することがあります。
- アクティビティーを再試行する必要があります。相関セットがアクティビティーによって初期化された場合、forceRetry メソッドが呼び出される前に、相関セットを未初期化または変更できます。
- アクティビティーを完了する必要があります。相関セットがアクティビティーによって初期化された場合、forceComplete メソッドが呼び出される前に、相関セットを未初期化または変更できます。

プロセス・インスタンスまたはアクティビティー・インスタンスの相関セット・インスタンスを取得できます。以下の例では、相関セット・インスタンスを初期化または未初期化する方法を示します。

手順

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'",
                    (String)null, (Integer)null, (TimeZone)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. アクティビティーに定義されている相関セット・インスタンスを取得します。

```
AIID aiid = null;

List correlationSet = null;

if (result.size() > 0)
{
   result.first();
   AIID aiid = (AIID) result.getOID(1);

   ActivityInstanceData activity = process.getActivityInstance(aiid);

   correlationSet = process.getCorrelationSetInstances
                    (aiid, activity.getInputMessageTypeName());
}
```

3. 相関セット (例えば MyCorrelationSet) を未初期化します。

```
for ( int i=0; i<correlationSet.size(); i++ )
{
  CorrelationSetInstanceData correlationSetInstance =
            (CorrelationSetInstanceData)correlationSet.get(i);

  if ( correlationSetInstance.isInitialized() &&
       correlationSetInstance.getCorrelationSetName().equals("MyCorrelationSet") )
  {
    process.uninitializeCorrelationSet
       ( activity.getProcessInstanceID(), correlSetInstance.getCorrelationSetName() );
  }
}
```

   このアクションにより、相関セット MyCorrelationSet が未初期化されます。

4. 相関セット (例えば MyCorrelationSet) を初期化します。 この例では、相関セットのストリング値プロパティーが設定されます。

```
for ( int i=0; i<correlationSet.size(); i++ )
{
  CorrelationSetInstanceData correlationSetInstance =
            (CorrelationSetInstanceData)correlationSet.get(i);

  if ( correlationSetInstance.getCorrelationSetName().equals("MyCorrelationSet") )
  {
    List correlationSetProperties =
                    correlationSetInstance.getCorrelationSetProperties();
    for ( int j=0; j<correlationSetProperties.size(); j++ )
    {
      CorrelationPropertyInstanceData property =
                (CorrelationPropertyInstanceData)correlationSetProperties.get(j);

      if ( property.getPropertyName().equals("MyProperty") )
      {
        property.setValue("NewValue");

        process.initializeCorrelationSet
           ( activity.getProcessInstanceID(), correlationSetInstance );
```

```
        }
      }
    }
  }
```

このアクションにより、相関セット MyCorrelationSet 内のストリング値プロパティー MyProperty が初期化されます。

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 sendMessage() 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 60. 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。
queryProcessTemplates	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・インスタンスの開始に関連しています。

表 61. プロセス・インスタンスの開始に関連する 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 61. プロセス・インスタンスの開始に関連する 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。
initiateAndSuspend	プロセス・インスタンスを作成しますが、そのプロセス・インスタンスのそれ以上の処理を直ちに中断します。
initiateAndClaimFirst	プロセス・インスタンスを作成し、最初のインライン・ヒューマン・タスクを要求します。
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。同じ相関セット値を持つプロセス・インスタンスが存在しない場合には作成されます。このプロセスは、固有または非固有のどちらかの開始サービスを持つことができます。
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 62. 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。

表 62. 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 (続き)

メソッド	説明
queryEntities	照会テーブルを使用して、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。
getWaitingActivities	メッセージを待機しているアクティビティーに関する情報を返し、それらのアクティビティーの処理を続行できるようにします。
migrate	プロセス・インスタンスを、指定された新しいバージョンのプロセス・モデルにマイグレーションします。

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。continueOnError フラグが FALSE に設定されているときに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。continueOnError 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 63. 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。
completeAndClaimSuccessor	アクティビティー・インスタンスを完了し、ログオン担当者の同じプロセス・インスタンス内の次のアクティビティーを要求します。
forceComplete	以下を強制的に完了します。 • 実行中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。 • 作動可能状態または要求済み状態にあるヒューマン・タスク・アクティビティー。 • 待機状態にある wait アクティビティー。

表 63. 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 (続き)

メソッド	説明
forceRetry	以下を強制的に反復します。 • 実行中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。 • 作動可能状態または要求済み状態にあるヒューマン・タスク・アクティビティー。
forceNavigate、 forceForEach、 forceLoop、 forceJoin	これらのメソッドは、停止したアクティビティーのナビゲーションを強制します。
skip	アクティビティーの処理をスキップします。
jump	あるアクティビティーから別のアクティビティーにジャンプします。
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。
queryEntities	照会テーブルを使用して、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 64. 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。

このタスクについて

Human Task Manager 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。

同期インターフェースを起動する呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを同期的に開始するのは、関連した SCA コンポーネントを同期的に呼び出せる場合に限ってください。

このタスクについて

このような SCA コンポーネントは、Microflow や単純な Java クラスなどとして実装できます。

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。

手順

1. オプション: タスク・テンプレートをリストして、実行する呼び出し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),
     (TimeZone)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呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを非同期的に開始するのは、関連した SCA コンポーネントを非同期的に呼び出せる場合に限ってください。

このタスクについて

このような SCA コンポーネントは、長時間実行プロセスや片方向操作などとして実装できます。

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。

手順

1. オプション: タスク・テンプレートをリストして、実行する呼び出し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),
       (TimeZone)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,
                         (ReplyHandlerWrapper)null);
```

タスク・インスタンスの作成と開始

このシナリオでは、コラボレーション・タスク (API ではヒューマン・タスク ともいう) 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。

手順

1. オプション: タスク・テンプレートをリストして、実行するコラボレーション・タスクのタスク・テンプレート ID (TKTID) を探します。

   タスク・テンプレート ID がすでにわかっている場合、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),
     (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みのタスク・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. コラボレーション・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,
                                          (ReplyHandlerWrapper)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。

予定タスクまたはコラボレーション・タスクの処理

予定タスク (API では*参加タスク* ともいう) またはコラボレーション・タスク (API ではヒューマン・タスク ともいう) 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。プロセスがヒューマン・タスク・アクティビティーにナビゲートしたときなどに、予定タスクとそれに関連した作業項目が作成されます。

このタスクについて

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。

手順

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",
                 (String)null, (Integer)null, (TimeZone)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。

関連概念

コラボレーション・タスクの状態遷移図
担当者が他の担当者の作業を実行するときに使用できるタスクです。コラボレーション・タスクのライフ・サイクル内では、特定の対話は特定のタスク状態でのみ可能であり、これらの対話が今度はタスクの状態に影響を与えます。

タスク・インスタンスの中断と再開

コラボレーション・タスク・インスタンス (API ではヒューマン・タスク ともいう) または予定タスク・インスタンス (API では*参加タスク* ともいう) を中断できます。

始める前に

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。

このタスクについて

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。

手順

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",
                                      (String)null,
                                      (Integer)null,
                                      (TimeZone)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

予定タスク (API では*参加* タスクともいう) またはコラボレーション・タスク (API ではヒューマン・タスク ともいう) は非同期に実行します。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。

このタスクについて

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。

手順

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",
                                    (String)null, (Integer)null, (TimeZone)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。タスク・インスタンスは即時に終了されるので、例外的な状況の場合にのみタスク・インスタンスを終了してください。

手順

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。この例では、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。

手順

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",
                    (String)null, (Integer)null, (TimeZone)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

潜在的な所有者がタスクを要求した場合、この所有者にはタスクの完了に対する責任があります。ただし、要求されたタスクを、別の潜在的所有者が要求できるように解放しなければならない場合があります。

このタスクについて

管理者権限を持つユーザーが、要求されたタスクを解放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。

手順

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'",
                    (String)null, (Integer)null, (TimeZone)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, true);
   }
   ```

   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。元の所有者が設定した出力データまたは障害データはすべて保持されます。

作業項目の管理

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。

このタスクについて

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ヒューマン・タスク・アクティビティー・インスタンス、プロセス・インスタンス、またはタスク・インスタンスのいずれかです。理由は、ユーザーがオブジェクトに対して持っているロールから派生します。ユーザーは、オブジェクト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 つのオブジェクトが複数の作業項目を持つことが可能です。例えば、予定タスク・インスタンスでは、管理者、読者、編集者、所有者の作業項目を同時に持つことができます。

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。

手順

- 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'",
                                    (String)null, (Integer)null,
                                    (TimeZone)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'",
                                    (String)null, (Integer)null,
                                    (TimeZone)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'",
                 (String)null, (Integer)null, (TimeZone)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。

このタスクについて

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。

随時の予定タスクまたはコラボレーション・タスクをモデル化できます。それには、com.ibm.task.api.TaskModel クラスのインスタンスを作成し、それを使用して再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを直接作成します。TaskModel クラスのインスタンスを作成するために、一連のファクトリー・メソッドが com.ibm.task.api.ClientTaskFactory ファクトリー・クラスで使用可能です。実行時のヒューマン・タスクのモデル化は、Eclipse Modeling Framework (EMF) に基づいています。

手順

1. createResourceSet ファクトリー・メソッドを使用して org.eclipse.emf.ecore.resource.ResourceSet を作成します。
2. オプション: 複雑なメッセージ・タイプを使用する予定の場合、org.eclipse.xsd.XSDFactory (ファクトリー・メソッド getXSDFactory() を使用して取得できる) を使ってそのメッセージ・タイプを定義するか、または loadXSDSchema ファクトリー・メソッドを使用して既存の XML スキーマを直接インポートできます。

   複雑なタイプを WebSphere Process Server が使用できるようにするには、そのタイプをエンタープライズ・アプリケーションの一部としてデプロイします。
3. タイプ javax.wsdl.Definition の Web サービス記述言語 (WSDL) 定義を作成またはインポートします。

   createWSDLDefinition メソッドを使用して新規の WSDL 定義を作成できます。次に、それをポート・タイプおよび操作に追加できます。また、loadWSDLDefinition ファクトリー・メソッドを使用して、既存の WSDL 定義を直接インポートできます。
4. createTTask ファクトリー・メソッドを使用してタスク定義を作成します。

   より複雑なタスク・エレメントを追加または操作する場合は、getTaskFactory ファクトリー・メソッドを使って取得できる com.ibm.wbit.tel.TaskFactory クラスを使用できます。
5. createTaskModel ファクトリー・メソッドを使用してタスク・モデルを作成し、それをステップ 1 で作成されたリソース・バンドルに渡します。リソース・バンドルはその間に作成されたその他のすべての成果物を集約します。
6. オプション: TaskModel validate メソッドを使用してモデルを検証します。

タスクの結果

TaskModel パラメーターを持つ Human Task Manager EJB API create メソッドの 1 つを使用して、再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを作成します。

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。

このタスクについて

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。

手順

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" ),
          (Map)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。

     ```
     atask.createTask( taskModel, (String)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, (String)null );
     ```

タスクの結果

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。

このタスクについて

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。

手順

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" ),
             (Map)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" );
     ```

タスクの結果

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。

このタスクについて

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。

手順

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", (String)null, (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。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" );
     ```

タスクの結果

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。

このタスクについて

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。

手順

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", (String)null, (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。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" );
     ```

タスクの結果

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 65. 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。
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。

表 65. 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。
createAndStartTaskAsSubtask	指定されたタスクのサブタスクとして、タスク・インスタンスを作成および開始します。
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 66. 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	呼び出し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。
startTaskAsSubtask	タスク・インスタンスのサブタスクとしてタスクを開始します。
suspend	コラボレーション・タスクまたは予定タスクを中断します。
resume	コラボレーション・タスクまたは予定タスクを再開します。
restart	タスク・インスタンスを再始動します。
terminate	指定されたタスク・インスタンスを終了します。呼び出し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。
completeWithFollowOnTask	タスク・インスタンスを完了し、追加タスクを開始します。

表 66. タスク・インスタンス用の API メソッド (続き)

メソッド	説明
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 67. 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。
triggerEscalation	エスカレーションを手動でトリガーします。

カスタム・プロパティー (Custom properties)

タスク、タスク・テンプレート、およびエスカレーションには、すべてカスタム・プロパティーを指定できます。このインターフェースは、カスタム・プロパティー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。次のメソッドは、タスク、タスク・テンプレート、およびエスカレーションの場合に有効です。

表 68. 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。

ビジネス・プロセスおよびヒューマン・タスク用アプリケーションの開発

ほとんどのビジネス・プロセス・シナリオには担当者が関係します。例えば、プロセスが開始または管理されるとき、あるいはヒューマン・タスク・アクティビティーが実行されるときには、ビジネス・プロセスに担当者の対話が必要となります。これらのシナリオをサポートするために、Business Flow Manager API と Human Task Manager API の両方を使用する必要があります。

このタスクについて

ビジネス・プロセス・シナリオに担当者を組み込むために、ビジネス・プロセスに以下の種類のタスクを含めることができます。

- インライン呼び出しタスク (API では*親タスク* ともいう)。

  すべての receive アクティビティー、pick アクティビティーの各 onMessage エレメント、およびイベント・ハンドラーの各 onEvent エレメントに対して呼び出しタスクを提供できます。次いで、このタスクはプロセスの開始、または実行中のプロセス・インスタンスとの通信を許可されているユーザーを制御します。
- 管理タスク。

  プロセスの管理またはプロセスの失敗したアクティビティーに対する管理操作の実行を許可されたユーザーを指定するための管理タスクを提供できます。
- 予定タスク (API では*参加タスク* ともいう)。

  予定タスクはヒューマン・タスク・アクティビティーを実装します。このタイプのアクティビティーにより、プロセスに担当者を含めることができます。

ビジネス・プロセス内のヒューマン・タスク・アクティビティーは、担当者がビジネス・プロセス・シナリオで実行する予定タスクを表します。 Business Flow Manager API と Human Task Manager API の両方を使用して、以下のシナリオを実現できます。

- ビジネス・プロセスとは、プロセスに属するすべてのアクティビティー (予定タスクによって表されるヒューマン・タスク・アクティビティーを含む) のコンテナーです。プロセス・インスタンスが作成されると、固有オブジェクト ID (PIID) が割り当てられます。
- ヒューマン・タスク・アクティビティーがプロセス・インスタンスの実行中に活動状態にされると、アクティビティー・インスタンスが作成されます。これはその固有オブジェクト ID (AIID) によって識別されます。同時に、インライン予定タスク・インスタンスも作成されます。これはそのオブジェクト ID (TKIID) によって識別されます。ヒューマン・タスク・アクティビティーとタスク・インスタンスの関係は、次のようにオブジェクト ID を使用して実現されます。
  - アクティビティー・インスタンスの予定タスク ID が、関連した予定タスクの TKIID に設定されます。
  - タスク・インスタンスの包含コンテキスト ID が、関連したアクティビティー・インスタンスを含むプロセス・インスタンスの PIID に設定されます。
  - タスク・インスタンスの親コンテキスト ID が、関連したアクティビティー・インスタンスの AIID に設定されます。

- すべてのインライン予定タスク・インスタンスのライフ・サイクルは、プロセス・インスタンスによって管理されます。プロセス・インスタンスが削除されると、タスク・インスタンスも削除されます。例えば、包含コンテキスト ID がプロセス・インスタンスの PIID に設定されているすべてのタスクが自動的に削除されます。

開始できるプロセス・テンプレートまたはアクティビティーの判別

ビジネス・プロセスは、Business Flow Manager API の call、initiate、または sendMessage メソッドの呼び出しにより開始できます。プロセスに 1 つの開始アクティビティーしかない場合は、パラメーターとしてプロセス・テンプレート名を必要とするメソッド・シグニチャーを使用できます。プロセスに複数の開始アクティビティーがある場合は、開始アクティビティーを明示的に識別する必要があります。

このタスクについて

ビジネス・プロセスをモデル化する場合、モデラーは、ユーザーのサブセットだけしかプロセス・テンプレートからプロセス・インスタンスを作成できないように決定することができます。これは、インライン呼び出しタスクをプロセスの開始アクティビティーに関連付け、許可制限をそのタスクに指定することで実行できます。タスクの潜在的スターターまたは管理者だけが、タスクのインスタンスの (およびプロセス・テンプレートのインスタンスの) 作成を許可されます。

インライン呼び出しタスクが開始アクティビティーと関連付けられていない場合、または許可制限がタスクに指定されていない場合、誰でも開始アクティビティーを使用してプロセス・インスタンスを作成できます。

プロセスは、それぞれが異なる潜在的スターターまたは管理者に対する担当者照会を持つ、複数の開始アクティビティーを持つことができます。これはつまり、ユーザーがアクティビティー A を使用したプロセスの開始は許可されるが、アクティビティ B を使用しては許可されないということです。

手順

1. Business Flow Manager API を使用して、開始済み状態のプロセス・テンプレートの現行バージョンのリストを作成します。

   ヒント: queryProcessTemplates メソッドは、開始されていないアプリケーションの一部であるプロセス・テンプレートだけを除外します。そのため、結果をフィルター処理せずにこのメソッドを使用する場合、メソッドはどのような状態にあるかに関係なく、すべてのバージョンのプロセス・テンプレートを戻します。

```
// current timestamp in UTC format, converted to yyyy-mm-ddThh:mm:ss
String now = (new UTCDate()).toXsdString();
String whereClause = "PROCESS_TEMPLATE.STATE =
                      PROCESS_TEMPLATE.STATE.STATE_STARTED AND
                      PROCESS_TEMPLATE.VALID_FROM =
                      (SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE
                       WHERE NAME=PROCESS_TEMPLATE.NAME AND
                       VALID_FROM <= TS('" + now + "'))";

ProcessTemplateData[] processTemplates = process.queryProcessTemplates
                                        ( whereClause,
                                          "PROCESS_TEMPLATE.NAME",
                                          (Integer)null, (TimeZone)null);
```

結果はプロセス・テンプレート名でソートされます。

2. プロセス・テンプレートのリストと、ユーザーが許可されている開始アクティビティーのリストを作成します。

   プロセス・テンプレートのリストには、単一の開始アクティビティーがあるプロセス・テンプレートが含まれます。これらのアクティビティーは、保護されていないか、またはログオン・ユーザーによる開始が許可されていないかのいずれかです。あるいは、少なくとも 1 つの開始アクティビティーにより開始できるプロセス・テンプレートを収集することもできます。

   ヒント: プロセス管理者は、プロセス・インスタンスを開始することもできます。ただし、Business Flow Manager で代替プロセス管理許可モードを使用している場合には、プロセス管理がシステム管理者に制限されており、このアクションを実行できるのは BPESystemAdministrator ロールを持つユーザーのみになります。そのため、テンプレートの完全なリストを入手するには、ログオン・ユーザーが管理者であるかどうかを確認することも必要です。

   ```
   List authorizedProcessTemplates = new ArrayList();
   List authorizedActivityServiceTemplates = new ArrayList();
   ```

3. プロセス・テンプレートごとに、開始アクティビティーを判別します。

   ```
   for( int i=0; i<processTemplates.length; i++ )
   {
     ProcessTemplateData template = processTemplates[i];
     ActivityServiceTemplateData[] startActivities =
                   process.getStartActivities(template.getID());
   ```

4. 開始アクティビティーごとに、関連したインライン呼び出しタスク・テンプレートの ID を取得します。

   ```
   for( int j=0; j<startActivities.length; j++ )
   {
     ActivityServiceTemplateData activity = startActivities[j];
     TKTID tktid = activity.getTaskTemplateID();
   ```

   a. 呼び出しタスク・テンプレートが存在しない場合、プロセス・テンプレートはこの開始アクティビティーによっては保護されません。

      この場合、この開始アクティビティーを使用して、誰でもプロセス・インスタンスを作成できます。

      ```
      boolean isAuthorized = false;
             if ( tktid == null )
         {
           isAuthorized = true;
           authorizedActivityServiceTemplates.add(activity);
         }
      ```

   b. 呼び出しタスク・テンプレートが存在する場合は、Human Task Manager API を使用して、ログオン・ユーザーの許可を検査します。

      例ではログオン・ユーザーは Smith です。ログオン・ユーザーは、呼び出しタスクの潜在的なスターターまたは管理者である必要があります。

      ```
      if ( tktid != null )
          {
            isAuthorized =
              task.isUserInRole
                  (tkid, "Smith", WorkItem.REASON_POTENTIAL_STARTER) ||
              task.isUserInRole(tktid, "Smith", WorkItem.REASON_ADMINISTRATOR);
      ```

```
        if ( isAuthorized )
        {
          authorizedActivityServiceTemplates.add(activity);
        }
      }
```

ユーザーに指定されたロールがある場合、またはそのロールの担当者割り当て基準が指定されていない場合、isUserInRole メソッドは値 true を戻します。

5. プロセス・テンプレート名だけを使用してプロセスが開始できるかを確認します。

```
if ( isAuthorized && startActivities.length == 1 )
    {
      authorizedProcessTemplates.add(template);
    }
```

6. ループを終了します。

```
  } // end of loop for each activity service template
}   // end of loop for each process template
```

ヒューマン・タスクを含む単一の個人ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、サーバー・サイド・ページ・フローを使用した単一の個人ワークフローを実装する方法を示します。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

このタスクについて

単一の個人ワークフローは、ページ・フロー または*画面フロー* とも呼ばれます。以下の 2 種類のページ・フローがあります。

- クライアント・サイド・ページ・フロー。このフローでは、複数ページの Lotus Forms フォームなどのクライアント・サイド・テクノロジーを使用して、異なるページ間のナビゲーションを実現しています。
- サーバー・サイド・ページ・フロー。このフローは、ビジネス・プロセスと、後続のタスクが同じ担当者に割り当てられるようにモデル化された一連のヒューマン・タスクを使用して実現されます。

サーバー・サイド・ページ・フローはクライアント・サイド・ページ・フローよりも強力ですが、処理のために消費するサーバー・リソースは増えます。したがって、このタイプのワークフローは、主として次の状態の場合に使用することを検討してください。

- ユーザー・インターフェースで実行された手順間でサービスを呼び出す必要がある場合 (データの検索や更新など)。
- ユーザー・インターフェースでの対話が完了した後に CEI イベントの書き込みを要求する監査要件が存在する場合。

オンライン・ブックストアでは、購入者は一連の操作を完了することで本を注文します。この一連の操作は、ヒューマン・タスク・アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文する場合は、これが次のヒューマ

ン・タスク・アクティビティーの要求に相当します。一連のタスクに関する情報は Business Flow Manager によって保守されますが、タスク自体は Human Task Manager によって保守されます。

この例を、Business Flow Manager API のみを使用する例と比較してください。

手順

1. Business Flow Manager API を使用して、作業対象となるプロセス・インスタンスを取得します。

   この例では、CustomerOrder プロセスのインスタンスです。

   ```
   ProcessInstanceData processInstance =
      process.getProcessInstance("CustomerOrder");
   String piid = processInstance.getID().toString();
   ```

2. Human Task Manager API を使用して、指定されたプロセス・インスタンスの一部である、準備のできた予定タスク (種類は参加) を照会します。

   タスクの包含コンテキスト ID を使用して、含まれているプロセス・インスタンスを指定します。単一の個人ワークフローの場合、照会によって、一連のヒューマン・タスク・アクティビティーの最初のヒューマン・タスク・アクティビティーに関連付けられている予定タスクが戻されます。

   ```
   //
    // Query the list of to-do tasks that can be claimed by the logged-on user
    // for the specified process instance
   //
   QueryResultSet result =
        task.query("DISTINCT TASK.TKIID",
                   "TASK.CONTAINMENT_CTX_ID = ID('" + piid + "') AND
                    TASK.STATE = TASK.STATE.STATE_READY AND
                    TASK.KIND = TASK.KIND.KIND_PARTICIPATING AND
                    WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                   (String)null, (Integer)null, (TimeZone)null);
   ```

3. 戻される予定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 activityInput = 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。

4. 予定タスクに関連付けられたヒューマン・タスク・アクティビティーを判別します。

   以下のメソッドのいずれかを使用して、アクティビティーをそのタスクに相関させます。

   - task.getActivityID メソッド:

     ```
     AIID aiid = task.getActivityID(tkiid);
     ```

- タスク・オブジェクトの一部である親コンテキスト ID:

```
AIID aiid = null;
Task taskInstance = task.getTask(tkiid);

OID oid = taskInstance.getParentContextID();
if ( oid != null and oid instanceof AIID )
{
  aiid = (AIID)oid;
}
```

5. タスクでの作業が終了したら、Business Flow Manager API を使用してタスクとそれに関連するヒューマン・タスク・アクティビティーを完了し、プロセス・インスタンス内の次のヒューマン・タスク・アクティビティーを要求します。

   ヒューマン・タスク・アクティビティーを完了するには、出力メッセージを渡します。出力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。

```
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 human task activity and its associated to-do task,
// and claim the next human task activity
CompleteAndClaimSuccessorResult successor =
   process.completeAndClaimSuccessor(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、シーケンス内の次のヒューマン・タスク・アクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

   後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者であるヒューマン・タスク・アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

6. 次のヒューマン・タスク・アクティビティーを処理します。

```
ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

7. ステップ 5 を続行して、ヒューマン・タスク・アクティビティーを完了し、次のヒューマン・タスク・アクティビティーを取得します。

関連タスク

434 ページの『単独ユーザー・ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 initiateAndClaimFirst API および completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。この例では、クライアント・サイド・ページ・フローを使用した単独ユーザー・ワークフローの実装を示します。

例外および障害の処理

BPEL プロセスでは、プロセス内のさまざまな箇所で障害が発生する可能性があります。

このタスクについて

Business Process Execution Language (BPEL) の障害には次の発生原因があります。

- Web サービスの呼び出し (Web Services Description Language (WSDL) の障害)
- throw アクティビティー
- Business Process Choreographer によって認識される BPEL 標準障害

これらの障害を処理する機構が存在します。プロセス・インスタンスによって生成される障害を処理するには、以下の手段のいずれかを使用します。

- 対応する障害ハンドラーへの制御の引き渡し
- プロセス内の前の処理の補正
- 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)

BPEL プロセスは、プロセスによって指定される操作の呼び出し元に障害を戻す可能性もあります。プロセス内の障害は、障害名と障害データを指定して reply アクティビティーとしてモデル化できます。これらの障害は、チェック例外として API 呼び出し元に戻されます。

BPEL プロセスで BPEL 障害を処理しない場合、または API 例外が発生した場合は、ランタイム例外が API 呼び出し元に戻されます。API 例外の例には、インスタンスの作成元となるプロセス・モデルが存在しない場合などがあります。

障害および例外の処理は、以下のタスクで説明します。

Business Process Choreographer EJB 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。

このタスクについて

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 します。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。

ヒューマン・タスク・アクティビティーに設定された障害の検査

ヒューマン・タスク・アクティビティーが処理されると、そのヒューマン・タスク・アクティビティーは正常に完了できます。この場合は、出力メッセージを渡すことができます。ヒューマン・タスク・アクティビティーが正常に完了しない場合は、障害メッセージを渡すことができます。

このタスクについて

障害メッセージを読んでエラーの原因を調べることができます。

手順

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",
                     (String)null, (Integer)null, (TimeZone)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。

```
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

適切に設計されたプロセスでは、例外および障害は、通常は障害ハンドラーによって処理されます。invoke アクティビティーで発生した例外または障害に関する情報は、アクティビティー・インスタンスから取得できます。

このタスクについて

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。

手順

1. 停止状態のヒューマン・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_STOPPED AND
                    ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
                    (String)null, (Integer)null, (TimeZone)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。

```
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();
  }
}
```

失敗プロセス・インスタンスで発生した未処理の例外または障害の検査

適切に設計されたプロセスでは、例外および障害は、通常は障害ハンドラーによって処理されます。プロセスが両方向操作を実装する場合、障害または処理済みの例外に関する情報は、プロセス・インスタンス・オブジェクトの障害名プロパティーから取得できます。障害の場合は、getFaultMessage API を使用して、対応する障害メッセージを取得することもできます。

このタスクについて

例外がどの障害ハンドラーにも処理されないことが原因でプロセス・インスタンスが失敗した場合は、未処理の例外に関する情報をプロセス・インスタンス・オブジェクトから取得できます。これとは対照的に、障害が障害ハンドラーによってキャッチされた場合、その障害に関する情報は入手できません。ただし、FaultReplyException 例外を使用することにより、障害名および障害メッセージを取得して、呼び出し元に戻ることができます。

手順

1. 失敗状態にあるプロセス・インスタンスをリストします。

```
QueryResultSet result =
     process.query("PROCESS_INSTANCE.PIID",
                   "PROCESS_INSTANCE.STATE =
                       PROCESS_INSTANCE.STATE.STATE_FAILED",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、失敗プロセス・インスタンスが含まれる照会結果セットを戻します。

2. 未処理の例外に関する情報を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ProcessInstanceData pInstance = process.getProcessInstance(piid);

  ProcessException excp = pInstance.getUnhandledException();
  if ( excp instanceof RuntimeFaultException )
  {
   RuntimeFaultException xcp = (RuntimeFaultException)excp;
   Throwable cause = xcp.getRootCause();
  }
  else if ( excp instanceof StandardFaultException )
  {
   StandardFaultException xcp = (StandardFaultException)excp;
   String faultName = xcp.getFaultName();
  }
  else if ( excp instanceof ApplicationFaultException )
  {
   ApplicationFaultException xcp = (ApplicationFaultException)excp;
   String faultName = xcp.getFaultName();
  }
}
```

タスクの結果

この情報を使用して、障害名または問題の根本原因を調べます。

ビジネス・プロセスおよびヒューマン・タスク用 Web サービス API クライアント・アプリケーションの開発

Business Process Choreographer Web サービス API を介してビジネス・プロセス・アプリケーションとヒューマン・タスク・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。クライアント・アプリケーションの開発プロセスは、Web サービス・プロキシーの生成や、クライアント・アプリケーションへのセキュリティー・ポリシーおよびトランザクション・ポリシーの追加など、多数の必須のステップとオプションのステップで構成されています。

このタスクについて

バージョン 7 以降では、JAX-WS ベースの Web サービス API が、バージョン 6 の JAX-RPC ベースの Business Process Choreographer Web サービス API (リリース 6.0.2 で初めて公開) の代わりに使用されます。JAX-RPC ベースの Business Process Choreographer Web サービス API は非推奨となるため、新規の Web サービス・クライアント・アプリケーションは、JAX-WS ベースの API を使用して実装する必要があります。

注: Business Process Choreographer Java Message Service (JMS) API は、引き続きバージョン 6 の WSDL および XML スキーマ定義を使用しています。

任意の Web サービス・クライアント環境でクライアント・アプリケーションを開発できます。以下のステップは、そのようなアプリケーションの開発に必要なアクションの概要です。

手順

1. Business Flow Manager API、Human Task Manager API、またはその両方の中から、クライアント・アプリケーションで使用する必要がある Web サービス API を決定します。
2. WebSphere Process Server 環境から必要なファイルをエクスポートします。
3. クライアント・アプリケーション開発環境で、エクスポートした成果物を使用して*Web サービス・プロキシー* を生成します。
4. クライアント・アプリケーション用のコードを開発します。
5. 必要なセキュリティー・ポリシーまたはトランザクション・ポリシーをクライアント・アプリケーションに追加します。

Web サービス・コンポーネントおよび一連の制御

Web サービス・アプリケーションでは、多くのクライアント・サイドおよびサーバー・サイドのコンポーネントは、Web サービスの要求と応答を表す一連の制御に関与します。

標準的な一連の制御は以下のとおりです。

1. クライアント・サイド:
   a. クライアント・アプリケーション (ユーザーによって提供される) は、Web サービスの要求を発行します。
   b. Web サービス・プロキシー (ユーザーによっても提供されるが、クライアント・サイド・ユーティリティーを使用して自動的に生成することが可能) は、サービス要求を SOAP 要求エンベロープでラップし、Web サービスのエンドポイントとして定義された URL に要求を転送します。
2. ネットワークは、HTTP または HTTPS を使用して Web サービス・エンドポイントに要求を送信します。
3. サーバー・サイド:
   a. 汎用 Web サービス API は、要求を受信し、デコードします。
   b. 要求は、汎用の Business Flow Manager または Human Task Manager コンポーネントによって直接処理されるか、指定されたビジネス・プロセスまたはヒューマン・タスクに転送されます。
   c. 戻されたデータは SOAP 応答エンベロープでラップされます。
4. ネットワークは、HTTP または HTTPS を使用してクライアント・サイド環境に応答を送信します。
5. クライアント・サイドに戻る:
   a. クライアント・サイド開発インフラストラクチャーは、SOAP 応答エンベロープをアンラップします。
   b. Web サービス・プロキシーはデータを SOAP 応答から抽出して、それをクライアント・アプリケーションに受け渡します。
   c. クライアント・アプリケーションは、必要に応じて、戻されたデータを処理します。

例

Human Task Manager Web サービス API にアクセスして予定タスクを処理するクライアント・アプリケーションの場合に考えられる処理の概要を次に示します。

1. クライアント・アプリケーションは、query Web サービス呼び出しを WebSphere Process Server に対して発行します。これにより、ユーザーによって処理される予定タスクのリストを要求します。
2. WebSphere Process Server は、予定タスクのリストを戻します。
3. クライアント・アプリケーションは、claim Web サービス呼び出しを発行して、いずれかの予定タスクを要求します。
4. WebSphere Process Server は、タスクの入力メッセージを戻します。
5. クライアント・アプリケーションは、complete Web サービス呼び出しを発行して、出力メッセージまたは障害メッセージのあるタスクを完了します。

ビジネス・プロセスおよびヒューマン・タスクの Web サービス API 要件

WebSphere Integration Developer により Business Process Choreographer で実行するために開発されるビジネス・プロセスとヒューマン・タスクは、Web サービス API を介してアクセスできるようにするために特定の規則に準拠する必要があります。

要件は以下のとおりです。

- ビジネス・プロセスとヒューマン・タスクのインターフェースは、Java API for XML-based Web Services (JAX-WS 2.0) 仕様で定義される「document/literal wrapped」スタイルを使用して定義する必要があります。これは、WebSphere Integration Developer で開発するすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。
- 操作のパラメーター・エレメントで maxOccurs 属性を使用しないでください。使用する場合は、この属性の値がデフォルト値の `maxOccurs="1"` に設定されていることを確認してください。
- Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。例:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

関連情報

Java API for XML-based Web Services (JAX-WS 2.0) ダウンロード・ページ

Which style of WSDL should I use?

JAX-WS ベースの Business Process Choreographer Web サービス API

バージョン 7 以降では、JAX-WS ベースの Web サービス API が、バージョン 6 の JAX-RPC ベースの Business Process Choreographer Web サービス API (リリース 6.0.2 で初めて公開) の代わりに使用されます。2 つの Business Process Choreographer Web サービス・インターフェース (ビジネス・プロセス用およびヒュ

ーマン・タスク用に 1 つずつ) が用意されており、それぞれに独自のファイル成果物および XML 定義の名前空間があります。

以下の表に、JAX-WS ベースの Web サービスのファイル成果物および XML 定義の名前空間の概要を示します。

表 69. JAX-WS ベースの Web サービスのファイル成果物および XML 定義の名前空間

Business Process Choreographer Web サービス・インターフェース	**JAX-WS Web サービスのファイル成果物**	**JAX-WS Web サービスの XML 名前空間**
Business Flow Manager Web サービス	BFMJAXWSService.wsdl	http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0/Binding
Business Flow Manager Web サービス・インターフェース	BFMJAXWSInterface.wsdl	http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0
Business Flow Manager Web サービス・データ型	BFMDataTypes.xsd	http://www.ibm.com/xmlns/prod/websphere/business-process/types/7.0
Business Flow Manager コールバック Web サービス	BFMJAXWSCallbackService.wsdl	http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0/Binding
Business Flow Manager コールバック Web サービス・インターフェース	BFMJAXWSCallbackInterface.wsdl	http://www.ibm.com/xmlns/prod/websphere/business-process/callback-services/7.0
Human Task Manager Web サービス	HTMJAXWSService.wsdl	http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0/Binding
Human Task Manager Web サービス・インターフェース	HTMJAXWSInterface.wsdl	http://www.ibm.com/xmlns/prod/websphere/human-task/services/7.0
Human Task Manager Web サービス・データ型	HTMDataTypes.xsd	http://www.ibm.com/xmlns/prod/websphere/human-task/types/7.0
Human Task Manager コールバック Web サービス	HTMJAXWSCallbackService.wsdl	http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0/Binding
Human Task Manager コールバック Web サービス・インターフェース	HTMJAXWSCallbackInterface.wsdl	http://www.ibm.com/xmlns/prod/websphere/human-task/callback-services/7.0
Common Business Process Choreographer データ型	BPCDataTypes.xsd	http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/7.0

Business Process Choreographer Web サービス API: 標準

以下のリンクを使用して、Web アプリケーションに適用される標準についての関連補足情報を参照してください。情報は、IBM 以外のインターネット・サイトにあり、情報の技術的な正確性はサイトの提供者が管理しています。

これらのリンクは便宜上提供しているものです。多くの場合、情報は IBM WebSphere Process Server に固有のものではありませんが、一般的な Web サービスを理解するのに役立ちます。

- Java API for XML-based Web Services (JAX-WS 2.0) (JSR-224; Java Community Process)
- Java Architecture for XML Binding (JAXB) 2.0 (JSR-222; Java Community Process)
- Web サービス記述言語 (WSDL) 1.1 (W3C)
- XML Schema Part 0: Primer Second Edition (W3C)
- XML Schema Part 1: Structures Second Edition (W3C)
- XML Schema Part 2: Datatypes Second Edition (W3C)
- Simple Object Access Protocol (SOAP) 1.1 (W3C)
- Web Services Policy Framework (WS-Policy) 1.5 (W3C)
- WS-Security 1.1 (OASIS)

- WS-Security UserName Token Profile 1.1 (OASIS)
- WS-AtomicTransaction 1.2 (OASIS)
- WS-Interoperability Basic Profile 1.1 (WS-Interoperability Organization)

Web サービス・クライアント・アプリケーションのサーバー環境での成果物の公開とエクスポート

クライアント・アプリケーションを開発して Business Process Choreographer Web サービス API にアクセスするには、WebSphere サーバー環境であらかじめ数多くの成果物を公開し、エクスポートしておく必要があります。

このタスクについて

エクスポートする成果物は以下のとおりです。

- Business Process Choreographer Web サービス API (Web サービス・プロキシー生成のために常に必要) を構成する Web サービス・エンドポイント、ポート・タイプ、および操作について記述した Web サービス記述言語 (WSDL) ファイル。
- Business Process Choreographer WSDL ファイル (Web サービス・プロキシー生成のために常に必要) 内のサービスによって参照されるデータ型の定義を含む XML スキーマ定義 (XSD) ファイル。
- WebSphere サーバーで実行されるビジネス・プロセスまたはヒューマン・タスクのインターフェースおよびデータ型を記述するユーザー独自の WSDL ファイルおよび XSD ファイル。これらの追加ファイルは、クライアント・アプリケーションが Web サービス API を介してビジネス・プロセスまたはヒューマン・タスクと直接対話する必要がある場合にのみ必要です。クライアント・アプリケーションが、プロセス・インスタンスまたはタスク・インスタンスと直接対話することなく Business Process Choreographer によって実行できる操作 (照会の発行など) を呼び出すだけの場合、これらは不要です。
- Web サービス API のサービス品質属性を記述した Web サービス・ポリシー (WS-Policy) ファイル。これらは、クライアント・サイドの Web サービス・ポリシーを作成する際のベースとするためにエクスポートすることができます。

  WS-Security
  要求メッセージには、UserName トークンまたは LPTA トークンが含まれていなければなりません。

  WS-Transaction
  要求メッセージには、WS-AtomicTransaction コンテキストを含めることができます。このコンテキストが存在する場合、要求は呼び出し元のトランザクション・スコープで処理されます。

これらの成果物は、公開された後、ご使用のクライアント・プログラミング環境にコピーする必要があります。それらは、Web サービス・プロキシーおよびヘルパー・クラスの生成に使用されます。

Business Process Choreographer WSDL ファイルの公開

Web サービス記述言語 (WSDL) ファイルには、Web サービス API で使用できるすべての操作の詳細な説明が含まれています。Business Flow Manager と Human

Task Manager の Web サービス API では、別々の WSDL ファイルを使用できます。これらのファイルは、ご使用のアプリケーションの Web サービス・プロキシーを生成するために使用されます。

始める前に

WSDL ファイルを公開する前に、指定した Web サービス・エンドポイント・アドレスが正しいことを確認してください。このアドレスは、クライアント・アプリケーションが Web サービス API へのアクセスに使用する URL です。

このタスクについて

これらの WSDL ファイルと、WSDL ファイルが参照する XSD ファイルを公開する必要があります。すると、それらのファイルを WebSphere 環境からご使用の開発環境にコピーして、Web サービス・プロキシーおよびヘルパー・クラスの生成に使用することができます。Business Process Choreographer WSDL ファイルの公開が必要なのは一度だけです。

Web サービス・アプリケーション用のビジネス・プロセス WSDL ファイルの公開:

管理コンソールを使用して WSDL ファイルを公開します。

手順

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」をクリックします。

   注: 「**アプリケーション**」 → 「**アプリケーション・タイプ**」 → 「**WebSphere エンタープライズ・アプリケーション**」をクリックして、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **BPEContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の .zip ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

タスクの結果

エクスポートされる .zip ファイルは、
`BPEContainer_nodename_servername_WSDLFiles.zip` と命名されます。 .zip ファイルには、Web サービスを記述した WSDL ファイルと、その WSDL ファイルによって参照されるすべての XSD ファイルが含まれています。

注: エクスポートされた .zip ファイルには、バージョン 7 で導入された JAX-WS Web サービスとバージョン 6 で使用される JAX-RPC Web サービスの両方の

WSDL 成果物と XSD 成果物が含まれています。wsimport ツールを使用して Web サービス・プロキシーを生成するときに、JAX-WS Web サービス成果物を選択すると、JAX-RPC 成果物は無視されます。

Web サービス・アプリケーション用のヒューマン・タスク WSDL ファイルの公開:

管理コンソールを使用して WSDL ファイルを公開します。

手順

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」をクリックします。

   注: 「**アプリケーション**」 → 「**アプリケーション・タイプ**」 → 「**WebSphere エンタープライズ・アプリケーション**」をクリックして、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **TaskContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の .zip ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

タスクの結果

エクスポートされる .zip ファイルは、`TaskContainer_nodename_servername_WSDLFiles.zip` と命名されます。 .zip ファイルには、Web サービスを記述した WSDL ファイルと、その WSDL ファイルによって参照されるすべての XSD ファイルが含まれています。

注: エクスポートされた .zip ファイルには、バージョン 7 で導入された JAX-WS Web サービスとバージョン 6 で使用される JAX-RPC Web サービスの両方の WSDL 成果物と XSD 成果物が含まれています。wsimport ツールを使用して Web サービス・プロキシーを生成するときに、JAX-WS Web サービス成果物を選択すると、JAX-RPC 成果物は無視されます。

ビジネス・プロセスおよびヒューマン・タスク Web サービス・アプリケーションの WSDL ファイルおよび XSD ファイルのエクスポート

ビジネス・プロセスおよびヒューマン・タスクには、Web サービスとして外部にアクセスできるよう明確に定義されたインターフェースが用意されています。WSDL インターフェース定義および XML スキーマ・データ型定義を、クライアント・プログラミング環境にエクスポートする必要があります。

このタスクについて

この手順は、クライアント・アプリケーションが対話する必要があるビジネス・プロセスまたはヒューマン・タスクごとに、繰り返してください。

例えば、ヒューマン・タスクを作成して開始するには、以下の情報項目をタスク・インターフェースに渡す必要があります。

- タスク・テンプレート名
- タスク・テンプレート名前空間
- 入力メッセージ (フォーマット済みのビジネス・データを含む)
- 応答メッセージを戻すための応答ラッパー
- 障害および例外を戻すための障害メッセージ

以上の項目は、単一のビジネス・オブジェクト内でカプセル化されます。Web サービス・インターフェースのすべての操作は、document/literal wrapped 操作としてモデル化されます。これらの操作の入出力パラメーターは、ラッパー文書の中にカプセル化されます。他のビジネス・オブジェクトは、それに対応する応答および障害のメッセージ形式を定義します。

Web サービスによってビジネス・プロセスまたはヒューマン・タスクを作成および開始するためには、クライアント・サイドのクライアント・アプリケーションがこれらのラッパー・オブジェクトを使用できるようにする必要があります。

そのためには、WebSphere 環境からビジネス・オブジェクトを Web サービス記述言語 (WSDL) ファイルおよび XML スキーマ定義 (XSD) ファイルとしてエクスポートし、データ型定義をクライアント・プログラミング環境にインポートします。

手順

1. WebSphere Integration Developer ワークスペースが稼働していない場合は、起動します。
2. エクスポートするビジネス・オブジェクトを含むライブラリー・モジュールを選択します。 ライブラリー・モジュールは、必要なビジネス・オブジェクトを含む圧縮ファイルです。
3. ライブラリー・モジュールをエクスポートします。
4. エクスポートしたファイルをクライアント・アプリケーション開発環境にコピーします。

例

ビジネス・プロセスが以下の Web サービス操作を公開するとします。

```
<wsdl:operation name="updateCustomer">
    <wsdl:input message="tns:updateCustomerRequestMsg"
                name="updateCustomerRequest"/>
    <wsdl:output message="tns:updateCustomerResponseMsg"
                 name="updateCustomerResponse"/>
    <wsdl:fault message="tns:updateCustomerFaultMsg"
                name="updateCustomerFault"/>
  </wsdl:operation>
```

公開は、以下のように定義された WSDL メッセージを介して行われるとします。

```
<wsdl:message name="updateCustomerRequestMsg">
    <wsdl:part element="types:updateCustomer"
               name="updateCustomerParameters"/>
  </wsdl:message>
  <wsdl:message name="updateCustomerResponseMsg">
    <wsdl:part element="types:updateCustomerResponse"
               name="updateCustomerResult"/>
  </wsdl:message>
  <wsdl:message name="updateCustomerFaultMsg">
    <wsdl:part element="types:updateCustomerFault"
               name="updateCustomerFault"/>
  </wsdl:message>
```

具象 ユーザー定義エレメント「types:updateCustomer」、「types:updateCustomerResponse」、および「types:updateCustomerFault」は、クライアント・アプリケーションが実行するすべての*汎用* 命令 (call、sendMessage など) で、UserData パラメーターを使用して Web サービス API に渡したり Web サービス API から受け取ったりする必要があります。

ユーザー定義エレメントは、エクスポートされた XSD ファイルから生成されるクラスを使用して、クライアント・アプリケーション上で作成、直列化、および非直列化されます。これらのクラスは、エクスポートされた WSDL ファイルおよび XSD ファイルが含まれている Web サービス・プロキシー生成の一部として生成されます。

Web サービス・インターフェースの汎用操作は、ビジネス・プロセスまたはヒューマン・タスクによって実装される操作との間で文書ラッパー・エレメントを伝搬します。前述の例で示したサンプル操作の場合、Web サービス SOAP メッセージは次のようになります。

```
<soapenv:Envelope xmlns:soapenv="..." ...>
  <soapenv:Header>
    ...
  </soapenv:Header>
  <soapenv:Body>
    <bfm:sendMessage
         xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/7.0">
      <processTemplateName>customerProcessTemplate</processTemplateName>
      <portType xmlns:cns="http://example.com/customerProcess">cns:customerProcessPortType</portType>
      <operation>updateCustomer</operation>
      <input>
        <cns:updateCustomer xmlns:cns="http://example.com/customerProcess">
          <street>1600 Pennsylvania Avenue Northwest</street>
          <city>Washington, DC 20006</city>
        </cns:updateCustomer>
      </input>
    </bfm:sendMessage>
  </soapenv:Body>
</soapenv:Envelope>
```

Java Web サービス環境でのクライアント・アプリケーションの開発

Java Web サービスと互換性のある Java ベースの開発環境を使用して、Business Process Choreographer Web サービス API 用のクライアント・アプリケーションを開発できます。

Web サービス・プロキシーの生成 (Java Web サービス)

Java Web サービス・クライアント・アプリケーションは、*Web* サービス・プロキシー を使用して Business Process Choreographer Web サービス API と対話します。

このタスクについて

Java Web サービスのWeb サービス・プロキシーには、Web サービス要求を実行するためにクライアント・アプリケーションが呼び出す JavaBeans クラスが数多く含まれています。 Web サービス・プロキシーは、サービス・パラメーターを SOAP メッセージにアセンブルし、SOAP メッセージを HTTP 経由で Web サービスに送信し、Web サービスから応答を受信し、戻されたデータをクライアント・アプリケーションに引き渡します。

したがって、基本的にWeb サービス・プロキシーによってクライアント・アプリケーションは、ローカル機能のようにして Web サービスを呼び出すことができます。

注: Web サービス・プロキシーの生成が必要なのは一度だけです。その後、同じ Web サービス API にアクセスするすべてのクライアント・アプリケーションは、同じWeb サービス・プロキシーを使用できます。

IBM Web サービス環境では、以下のいずれかの方法で Web サービス・プロキシーを生成できます。

- Rational Application Developer または WebSphere Integration Developer が統合された開発環境を使用する。
- wsimport コマンド行ツールを使用する。

その他の Java Web サービス開発環境には通常、wsimport ツールまたは独自のクライアント・アプリケーション生成機能が組み込まれています。

Rational Application Developer による Web サービス・アプリケーションの Web サービス・プロキシーの生成:

Rational Application Developer の統合開発環境を使用して、Web サービス・クライアント・アプリケーションの Web サービス・プロキシーを生成できます。以下の一連のステップは、Rational Application Developer バージョン 7.5.3 に適用されます。

始める前に

Web サービス・プロキシーを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス・インターフェースを記述した WSDL ファイルおよび XSD ファイルを WebSphere 環境からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

手順

1. 該当する WSDL ファイルをプロジェクトに追加します。
   - ビジネス・プロセスの場合:

a. エクスポート・ファイル
BPEContainer_nodename_servername_WSDLFiles.zip を一時ディレクトリーに unzip します。このディレクトリーの内容を変更しないでください。また、ビジネス・プロセスと対話するための Web サービス・プロキシーの生成に使用されるのは、以下の WSDL ファイルおよび XSD ファイルのみであることに注意してください。
   - BFMJAXWSService.wsdl
   - BFMJAXWSInterface.wsdl
   - BFMJAXWSCallbackService.wsdl
   - BFMJAXWSCallbackInterface.wsdl
   - BFMDataTypes.xsd
   - BPCDataTypes.xsd
   - wsa.xsd

b. サブディレクトリー META-INF を、unzip されたディレクトリー BPEContainer_*nodename_servername*.ear/bfmjaxws.jar からインポートします。

- ヒューマン・タスクの場合:
  a. エクスポート・ファイル
  TaskContainer_nodename_servername_WSDLFiles.zip を一時ディレクトリーに unzip します。このディレクトリーの内容を変更しないでください。また、ヒューマン・タスクと対話するための Web サービス・プロキシーの生成に使用されるのは、以下の WSDL ファイルおよび XSD ファイルのみであることに注意してください。
     - HTMJAXWSService.wsdl
     - HTMJAXWSInterface.wsdl
     - HTMJAXWSCallbackService.wsdl
     - HTMJAXWSCallbackInterface.wsdl
     - HTMDataTypes.xsd
     - BPCDataTypes.xsd
     - wsa.xsd

  b. サブディレクトリー META-INF を、unzip されたディレクトリー TaskContainer_*nodename_servername*.ear/htmjaxws.jar からインポートします。

新しい wsdl ディレクトリーおよびサブディレクトリー構造がプロジェクト内に作成されます。

2. 新たに作成された wsdl ディレクトリー内にある BFMJAXWSService.wsdl ファイルを選択します。
3. 右クリックして、「**Web サービス**」 → 「**クライアントを生成**」を選択します。

   残りのステップを続行する前に、サーバーが開始していることを確認します。
4. 「Web サービス」ウィンドウで「**次へ**」をクリックして、デフォルトをすべて受け入れます。

5. 「Web サービス」の「JAX-WS Web サービス (JAX-WS Web Service)」の「クライアント構成」ウィンドウで、生成される JAX-WS コードのバージョンを 2.0 に変更し、「**終了**」をクリックして他のすべてのデフォルト値を受け入れます。
6. HTMJAXWSService.wsdl に対して、この手順のステップ 2 から 5 までを再度実行し、プロンプトが出されたらすべてのファイルを上書きします。

タスクの結果

複数のプロキシー、ロケーター、および JAXB クラスで構成される Web サービス・プロキシーが生成され、プロジェクトに追加されます。

wsimport コマンド行ツールによる Web サービス・アプリケーションの Web サービス・プロキシーの生成:

wsimport コマンド行ツールを使用して、Web サービス・アプリケーションの Web サービス・プロキシーを生成できます。

始める前に

Web サービス・プロキシーを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス API を記述した WSDL ファイルを WebSphere 環境からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

手順

1. Business Process Choreographer Web サービス API の Web サービス・プロキシーを生成します。

   注: JAX-WS アプリケーションの wsimport コマンド行ツールの詳細な説明については、WebSphere Application Server の wsimport コマンド行ツールの資料を参照してください。

   ```
   wsimport.bat BFMJAXWSService.wsdl myService1.wsdl myService2.wsdl
    -d proxy-bfm
    -wsdllocation <bfm_location>

   wsimport.bat HTMJAXWSService.wsdl myService1.wsdl myService2.wsdl
    -d proxy-htm
    -wsdllocation <htm_location>
   ```

   この例では、myService1.wsdl および myService2.wsdl には、カスタム・ビジネス・プロセスまたはヒューマン・タスク、あるいはその両方のインターフェース定義が含まれています。さらに、*<bfm_location>* および *<htm_location>* は、それぞれ BFMJAXWSService.wsdl および HTMJAXWSService.wsdl 内の WSDL *<port>* エレメントから取得できます。

   両方のプロキシーを 1 つの共通のディレクトリー (例えば proxy-bpc) にマージし、プロンプトが出されたら既存のファイルを上書きすることができます。
2. 生成されたクラス・ファイルをプロジェクトに組み込みます。

関連タスク

『ビジネス・プロセスおよびヒューマン・タスク用クライアント・アプリケーションの作成 (Java Web サービス)』

クライアント・アプリケーションは、Business Process Choreographer Web サービス API に要求を送信し、Business Process Choreographer Web サービス API からの応答を受信します。Web サービス・プロキシーを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

ビジネス・プロセスおよびヒューマン・タスク用クライアント・アプリケーションの作成 (Java Web サービス)

クライアント・アプリケーションは、Business Process Choreographer Web サービス API に要求を送信し、Business Process Choreographer Web サービス API からの応答を受信します。Web サービス・プロキシーを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

始める前に

クライアント・アプリケーションの作成を開始する前に、Web サービス・プロキシーを生成します。

このタスクについて

Web サービス対応の開発ツール (IBM Rational Application Developer など) を使用すれば、クライアント・アプリケーションを開発できます。どのタイプの Web サービス・アプリケーションをビルドしても、Web サービス API を呼び出すことができます。

手順

1. 新規クライアント・アプリケーション・プロジェクトを作成します。
2. Web サービス・プロキシーを生成します。
3. クライアント・アプリケーションをコーディングします。
4. プロジェクトをビルドします。
5. クライアント・アプリケーションを実行します。

例

以下の例では、Business Flow Manager Web サービス API の使用方法を示します。

```
try {
   // create bfm proxy
   BFMJAXWSPortType bfm = new BFMJAXWSService().getBFMJAXWSPort();

   // call getProcessTemplate
   ProcessTemplateType ptt =
     bfm.getProcessTemplate("MY_PROCESS_TEMPLATE_NAME");

   // handle return value
   System.out.println("Process template '" + ptt.getName() +
       "' found, details following:");
   System.out.println("Execution mode: " +
```

```
        ptt.getExecutionMode());
    System.out.println("Schema version: " +
        ptt.getSchemaVersion());
  } catch (Exception e) {
    if ( e instanceof ProcessFaultMsg )
    {
      ProcessFaultMsg pfm = (ProcessFaultMsg) e;
      List<FaultStackType> list =
          ( pfm.getFaultInfo() ).getFaultStack();
      FaultStackType fault = list.get( 0 );
      System.out.println( "ProcessFaultMessage: " +
          fault.getMessage() );
    }
    else
    {
      e.printStackTrace( System.out );
    }
  }
```

関連タスク

488 ページの『wsimport コマンド行ツールによる Web サービス・アプリケーションの Web サービス・プロキシーの生成』
wsimport コマンド行ツールを使用して、Web サービス・アプリケーションの Web サービス・プロキシーを生成できます。

486 ページの『Web サービス・プロキシーの生成 (Java Web サービス)』
Java Web サービス・クライアント・アプリケーションは、*Web サービス・プロキシー* を使用して Business Process Choreographer Web サービス API と対話します。

セキュリティーの追加

Business Process Choreographer Web サービスでは、認証メカニズムに対応するようにクライアント・アプリケーションを構成する必要があります。

このタスクについて

デフォルトでは、Business Process Choreographer は、以下の認証メカニズムをサポートします。

ユーザー名トークン
Web サービス・コンシューマーは、「ユーザー名」によって要求側を識別 (オプションでパスワードを使用してその ID を認証) する手段として、ユーザー名トークンを Web サービス・プロバイダーに提供します。

バイナリー・セキュリティー・トークン - Lightweight Third-Party Authentication (LTPA) トークン
Web サービス・コンシューマーは、要求側を認証する手段として、LTPA トークンを Web サービス・プロバイダーに提供します。

Business Process Choreographer Web サービス・セキュリティー・ポリシーを、代替の認証メカニズムで置き換えることができます。ただし、Business Process Choreographer Web サービス操作を非認証ユーザーとして呼び出すことはできないため、常に 1 つの認証メカニズムが必要です。

トランザクション・サポートの追加

Web サービス・クライアント・アプリケーションを構成して、クライアント・アプリケーションのコンテキストをサービス要求の一部として引き渡すことによってクライアントのトランザクションにサーバー・サイドの要求処理が加わるのを許可す

ることができます。このアトミック・トランザクション・サポートは、Web Services-Atomic Transaction (WS-AT) 仕様で定義されています。

このタスクについて

Business Process Choreographer は、それぞれの Web サービス操作要求を別個のグローバル・トランザクションとして実行します。クライアント・アプリケーションは、次のいずれかの方法でトランザクション・サポートを使用するように構成できます。

- クライアントのトランザクション・コンテキストを伝搬させる。サーバー・サイドの要求処理は、クライアント・アプリケーションのトランザクション・コンテキスト内で実行されるため、クライアントのトランザクションと一緒にコミット (またはバックアウト) されます。逆に、Web サービス操作の実行中にサーバーに問題が発生して、サーバーがロールバックを要求した場合、クライアント・アプリケーションのトランザクションもロールバックされます。
- トランザクション・サポートを使用しない。Business Process Choreographer は、新規グローバル・トランザクションを作成して、その中で要求を実行しますが、サーバー・サイドの要求処理は、クライアント・アプリケーションのトランザクション・コンテキストでは実行されません。

Business Process Choreographer Web サービスに付加された Web サービス・ポリシーでは、前述のように各要求メッセージに WS-AT トランザクションを含めることを許可します。クライアント・トランザクション・コンテキストを渡さずに Web サービス操作を呼び出すことを選択する場合は、プロバイダー側のトランザクション・ポリシーを無視して、トランザクション・ポリシーなしで Web サービス・クライアントを構成するのが安全です。

Business Process Choreographer JMS API を使用したクライアント・アプリケーションの開発

Java Messaging Service (JMS) API を介してビジネス・プロセス・アプリケーションに非同期でアクセスするクライアント・アプリケーションを開発できます。

このタスクについて

JMS クライアント・アプリケーションは要求および応答メッセージを JMS API と交換します。要求メッセージを作成するために、クライアント・アプリケーションは JMS TextMessage メッセージ本文に、対応する操作の文書/リテラル・ラッパーを表す XML エレメントを入力します。

ビジネス・プロセスの要件

Business Process Choreographer 上で実行するように WebSphere Integration Developer で開発されたビジネス・プロセスが、JMS API によってアクセス可能となるためには、特定の規則に準拠する必要があります。

要件は以下のとおりです。

1. ビジネス・プロセスのインターフェースは、Java API for XML-based RPC (JAX-RPC 1.1) 仕様で定義された「document/literal wrapped」スタイルを使用し

て定義する必要があります。これは、WebSphere Integration Developer で開発するすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。

2. Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。例:

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

関連情報

Java API for XML based RPC (JAX-RPC) のダウンロード・ページ

Which style of WSDL should I use?

JMS レンダリングの許可

JMS インターフェースの使用を許可するには、WebSphere Application Server でセキュリティー設定が使用可能になっている必要があります。

ビジネス・プロセス・コンテナーがインストールされている場合、ロール **JMSAPIUser** をユーザー ID にマップする必要があります。このユーザー ID を使用して、すべての JMS API 要求を発行します。例えば、**JMSAPIUser** が「User A」にマップされる場合、すべての JMS API 要求はプロセス・エンジンにとって「User A」から発生しているように見えます。

JMSAPIUser ロールには以下の権限を割り当てる必要があります。

要求	必要な許可
forceTerminate	プロセス管理者
sendEvent	可能なアクティビティー所有者またはプロセス管理者

注: その他のすべての要求の場合、特殊権限は必要ありません。

特殊権限は、ビジネス・プロセス管理者のロールを持つ人物に付与されます。ビジネス・プロセス管理者は特殊なロールです。これは、プロセス・インスタンスのプロセス管理者とは異なります。ビジネス・プロセス管理者はすべての特権を持っています。

プロセス・スターターのユーザー ID は、プロセス・インスタンスが存在している場合、ユーザー・レジストリーから削除できません。このユーザー ID を削除する場合、このプロセスのナビゲーションが継続できません。 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>
```

JMS インターフェースへのアクセス

JMS インターフェースを介してメッセージを送受信するには、まずアプリケーションで BPC.cellname.Bus への接続を作成し、セッションを作成し、メッセージ・プロデューサーおよびコンシューマーを生成する必要があります。

このタスクについて

プロセス・サーバーは、point-to-point パラダイムに従う Java Message Service (JMS) メッセージを受け入れます。 JMS メッセージを送受信するアプリケーションは、以下のアクションに従う必要があります。

以下の例では、JMS クライアントは管理対象環境 (EJB、アプリケーション・クライアント、または Web クライアント・コンテナー) で実行していると想定しています。

手順

1. BPC.*cellname*.Bus への接続を作成します。 クライアント・アプリケーションの要求用の事前構成された接続ファクトリーは存在しません。つまり、クライアント・アプリケーションは、JMS API の ReplyConnectionFactory を使用するか、または固有の接続ファクトリーを作成するかのいずれかが可能です。作成する場合は接続ファクトリーを検索するために、Java Naming and Directory Interface (JNDI) 参照を使用できます。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。以下の例では、クライアント・アプリケーションが「jms/clientCF」という固有の接続ファクトリーを作成すると想定しています。

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

   // Look up the connection factory.
   // Create a connection factory that connects to the BPC bus.
   // Call it, for example, "jms/clientCF".
   // Also configure an appropriate authentication alias.
   ConnectionFactory connectionFactory =
               (ConnectionFactory)initialcontext.lookup("jms/clientCF");

   // Create the connection.
   Connection connection = connectionFactory.createConnection();
   ```

2. セッションを作成し、メッセージ・プロデューサーおよびコンシューマーが作成されるようにします。

   ```
   // Create a transaction session using auto-acknowledgment.
   Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
   ```

3. メッセージを送信するメッセージ・プロデューサーを作成します。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。

   ```
   // Look up the destination of the Business Process Choreographer input queue to
   // send messages to.
   Queue sendQueue = (Queue) initialcontext.lookup("jms/BFMJMSAPIQueue");

   // Create a message producer.
   MessageProducer producer = session.createProducer(sendQueue);
   ```

4. 応答を受信するメッセージ・コンシューマーを作成します。 応答宛先の JNDI 参照名は、ユーザー定義の宛先を指定できますが、デフォルトの (Business Process Choreographer 定義の) 応答宛先 jms/BFMJMSReplyQueue も指定できます。どちらの場合も、応答宛先は BPC.<cellname>.Bus にしておく必要があります。

```
// Look up the destination of the reply queue.
Queue  replyQueue = (Queue) initialcontext.lookup("jms/BFMJMSReplyQueue");

// Create a message consumer.
MessageConsumer consumer = session.createConsumer(replyQueue);
```

5. メッセージを送信します。

```
// Start the connection.
connection.start();

// Create a message - see the task descriptions for examples - and send it.
// This method is defined elsewhere ...
String payload = createXMLDocumentForRequest();
TextMessage requestMessage = session.createTextMessage(payload);

// Set mandatory JMS header.
// targetFunctionName is the operation name of JMS API
// (for example, getProcessTemplate, sendMessage)
requestMessage.setStringProperty("TargetFunctionName", targetFunctionName);

// Set the reply queue; this is mandatory if the replyQueue
// is not the default queue (as it is in this example).
requestMessage.setJMSReplyTo(replyQueue);

// Send the message.
producer.send(requestMessage);

// Get the message ID.
String jmsMessageID = requestMessage.getJMSMessageID();

session.commit();
```

6. 返信を受信します。

```
// Receive the reply message and analyse the reply.
TextMessage  replyMessage = (TextMessage) consumer.receive();

// Get the payload.
String payload = replyMessage.getText();

session.commit();
```

7. 接続を閉じ、リソースを解放します。

```
 // Final housekeeping; free the resources.
session.close();
connection.close();
```

注: 各トランザクションの後に接続を閉じることは不要です。接続が開始したら、接続が閉じるまでに、要求および応答メッセージはいくつでも交換できます。例では、単一のビジネス・メソッド内に単一の呼び出しがある単純なケースを示しています。

Business Process Choreographer JMS メッセージの構造

各 JMS メッセージのヘッダーおよび本文には事前定義構造がある必要があります。

Java Message Service (JMS) メッセージは以下のもので構成されます。

- メッセージ識別およびルーティング情報用のメッセージ・ヘッダー。
- 目次を保持するメッセージの本文 (ペイロード)。

Business Process Choreographer はテキスト・メッセージ形式のみサポートします。

メッセージ・ヘッダー

JMS は、クライアントが多くのメッセージ・ヘッダー・フィールドにアクセスできるようにします。

Business Process Choreographer JMS クライアントは以下のヘッダー・フィールドを設定できます。

JMSReplyTo
要求に対する応答を送信する宛先。このフィールドが要求メッセージで指定されていない場合、応答は Export インターフェースのデフォルトの応答宛先に送信されます (Export は、ビジネス・プロセス・コンポーネントのクライアント・インターフェース・レンダリングです)。この宛先は、`initialContext.lookup("jms/BFMJMSReplyQueue");` を使用して取得できます。

TargetFunctionName
WSDL 操作の名前 (例えば、"queryProcessTemplates")。このフィールドは常に設定する必要があります。 TargetFunctionName は、ここで説明されている汎用の JMS メッセージ・インターフェースの操作を指定することに注意してください。これを、例えば call または sendMessage 操作を使用して、間接的に呼び出すことができる具体的なプロセスまたはタスクによって提供される操作と混同しないでください。

また、Business Process Choreographer クライアントは以下のヘッダー・フィールドにアクセスできます。

JMSMessageID
メッセージを一意的に識別します。メッセージが送信されると、JMS プロバイダーによって設定されます。メッセージの送信前にクライアントが JMSMessageID を設定する場合、JMS プロバイダーによって上書きされます。メッセージの ID が認証目的で必要とされる場合、クライアントはメッセージの送信後に JMSMessageID を取得できます。

JMSCorrelationID
メッセージをリンクします。このフィールドは設定しないでください。 Business Process Choreographer 応答メッセージには、要求メッセージの JMSMessageID が含まれています。

各応答メッセージには以下の JMS ヘッダー・フィールドが含まれています。

- **IsBusinessException**

  WSDL 出力メッセージの場合は "false" で、WSDL 障害メッセージの場合は "true" です。

ServiceRuntimeExceptions は非同期クライアント・アプリケーションに戻されません。 JMS 要求メッセージの処理中に重大な例外が発生すると、それはランタイム障害となり、この要求メッセージを処理しているトランザクションはロールバックします。その後、JMS 要求メッセージが再度送達されます。メッセージを SCA Export の一部として処理している間 (例えば、メッセージの非直列化中) に障害が早期に発生した場合は、SCA Export の受信宛先によって指定された失敗した送達の最大数まで再試行されます。送達の失敗最大数に達した後、要求メッセージは

Business Process Choreographer バスのシステム例外宛先に追加されます。しかし、Business Flow Manager の SCA コンポーネントによる要求を実際に処理している間に障害が発生する場合、失敗した要求メッセージは WebSphere Process Server の失敗したイベント管理インフラストラクチャーによって処理されます。つまり、再試行しても例外状態が解決しない場合は、最終的に、失敗したイベント管理データベースに入れられることがあります。

メッセージ本文

ビジネス・プロセスまたはヒューマン・タスクによって公開される操作は、文書/リテラル・ラッパー・スタイルに準拠していなければなりません。JMS メッセージ本文は、操作の文書/リテラル・ラッパー・エレメントを表す XML 文書が入ったストリングです。JMS メッセージ・インターフェースの汎用操作は、ビジネス・プロセスまたはヒューマン・タスクによって実装される操作との間で文書ラッパー・エレメントを伝搬します。

以下の例は、単純で有効な要求メッセージ本体を示しています。

```
<bfm:queryProcessTemplates
     xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0">
  <whereClause>PROCESS_TEMPLATE.STATE IN (1)</whereClause>
</bfm:queryProcessTemplates>
```

以下の例は、より複雑で有効な要求メッセージ本体を示しています。クライアント・アプリケーションには、メッセージを特定のプロセスにサブミットするための sendMessage API 操作があります。プロセス入力メッセージは API パラメーターの 1 つです。このメッセージは、カスタマー・プロセスによって公開されるビジネス・オペレーションの入力メッセージです。プロセスには、メッセージをコンシュームする receive アクティビティーが含まれています。

`bfm:sendMessage` エレメントは、JMS API 操作の文書ラッパー・エレメントです。この中に組み込まれている `cns:updateCustomer` エレメントは、プロセスによって実装される操作の文書ラッパー・エレメントです。このプロセスには、例えば、`cns:customerProcessPortType` WSDL ポート・タイプと `updateCustomer` WSDL 操作を参照する `bpel:receive` アクティビティーが含まれています。

```
<bfm:sendMessage
     xmlns:bfm="http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0">
  <processTemplateName>customerProcessTemplate</processTemplateName>
  <portType xmlns:cns="http://example.com/customerProcess">cns:customerProcessPortType</portType>
  <operation>updateCustomer</operation>
  <cns:updateCustomer xmlns:cns="http://example.com/customerProcess">
    <street>1600 Pennsylvania Avenue Northwest</street>
    <city>Washington, DC 20006</city>
  </cns:updateCustomer>
</bfm:sendMessage>
```

関連タスク

497 ページの『応答メッセージでのビジネス例外の検査』
JMS クライアント・アプリケーションは、すべての応答メッセージのメッセージ・ヘッダーを検査して、ビジネス例外を調べる必要があります。

JMS クライアント・アプリケーションの成果物のコピー

JMS クライアント・アプリケーションを容易に作成するために、WebSphere Process Server 環境から多数の成果物をコピーすることができます。

このタスクについて

これらの成果物は、JMS メッセージ本文の作成に BOXMLSerializer を使用する場合にのみ必須です。 JMS API の場合、成果物は以下のとおりです。

```
BFMIF.wsdl
BFMIF.xsd
BPCGen.xsd
wsa.xsd
```

これらのファイルを WebSphere Process Server 環境から開発環境に公開およびエクスポートする必要があります。

JMS アプリケーション用のビジネス・プロセス WSDL ファイルの公開

管理コンソールを使用して WSDL ファイルを公開します。

手順

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」をクリックします。

   注: 「**アプリケーション**」 → 「**アプリケーション・タイプ**」 → 「**WebSphere エンタープライズ・アプリケーション**」をクリックして、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **BPEContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の .zip ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

タスクの結果

エクスポートされる .zip ファイルは、`BPEContainer_WSDLFiles.zip` と命名されます。 .zip ファイルには、WSDL ファイルと、WSDL ファイルによって参照されるすべての XSD ファイルが含まれています。

応答メッセージでのビジネス例外の検査

JMS クライアント・アプリケーションは、すべての応答メッセージのメッセージ・ヘッダーを検査して、ビジネス例外を調べる必要があります。

このタスクについて

JMS クライアント・アプリケーションはまず、応答メッセージのヘッダーの **IsBusinessException** プロパティーを検査する必要があります。

例:

例

```
// receive response message
Message receivedMessage = ((JmsProxy) getToBeInvokedUponObject().receiveMessage();
String strResponse = ((TextMessage) receivedMessage).getText();

if (receivedMessage.getStringProperty("IsBusinessException") {
  // strResponse is a bussiness fault
  // any api can end w/a processFaultMsg
  // the call api also w/a businessFaultMsg
 }
else {
  // strResponse is the output message
}
```

関連概念

494 ページの『Business Process Choreographer JMS メッセージの構造』
各 JMS メッセージのヘッダーおよび本文には事前定義構造がある必要があります。

例: Business Process Choreographer JMS API を使用して長期実行プロセスを実行する

この例では、長期実行プロセスで処理するために JMS API を使用する汎用のクライアント・アプリケーションの作成方法を示します。

手順

1. 492 ページの『JMS インターフェースへのアクセス』の説明に従って、JMS 環境をセットアップします。
2. 以下のように、インストールされたプロセス定義のリストを取得します。
   - `queryProcessTemplates` を送信します。
   - これにより、ProcessTemplate オブジェクトのリストが戻されます。
3. 以下のように、開始アクティビティー (`createInstance="yes"` を指定した receive または pick) のリストを取得します。
   - `getStartActivities` を送信します。
   - これにより、InboundOperationTemplate オブジェクトのリストが戻されます。
4. 入力メッセージを作成します。 これは環境に固有で、事前に配置された、プロセス固有の成果物を使用しなければならないことがあります。
5. プロセス・インスタンスを作成します。
   - `sendMessage` を発行します。

   JMS API とともに `call` 操作を使用して、ビジネス・プロセスによって提供される長期実行の要求/応答操作と対話することもできます。この操作では、長期間たった後でも、操作の結果または障害を指定された応答宛先に戻します。そのため、`call` 操作を使用する場合、`query` および `getOutputMessage` 操作を使用してプロセスの出力または障害メッセージを取得する必要はありません。
6. オプション: 以下のステップを繰り返して、プロセス・インスタンスから出力メッセージを取得します。
   a. `query` を発行して、終了状態のプロセス・インスタンスを取得します。
   b. `getOutputMessage` を発行します。

7. オプション: 以下のように、プロセスによって公開された追加の操作を実行します。
   a. getWaitingActivities または getActiveEventHandlers を発行して、InboundOperationTemplate オブジェクトのリストを取得します。
   b. 入力メッセージを作成します。
   c. sendMessage を発行してメッセージを送信します。
8. オプション: プロセスに対して定義された、またはアクティビティーを含むカスタム・プロパティーを、getCustomProperties および setCustomProperties で取得および設定します。
9. 以下のように、プロセス・インスタンスでの作業を終了します。
   a. delete および terminate を送信して、長期実行プロセスでの作業を終了します。

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 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。

このタスクについて

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。ヒューマン・タスクを含むアプリケーションでは、JSF カスタム・クライアントを生成できます。 JSF クライアントの生成について詳しくは、WebSphere Integration Developer のインフォメーション・センターを参照してください。

Business Process Choreographer で提供される JSF コンポーネントを使用して Web クライアントを開発することもできます。

手順

1. 動的プロジェクトを作成し、Web プロジェクト・フィーチャー・プロパティーを変更して JSF 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

WebSphere Process Server では、これらのすべてのファイルは以下のディレクトリーにあります。

- Windows Windows プラットフォームの場合:
  install_root¥ProcessChoreographer¥client
- Linux UNIX Linux® および UNIX® プラットフォームの場合:
  install_root/ProcessChoreographer/client

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・ライブラリー参照を 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。的確なエラー処理が行われるようにするには、JSF 構成ファイルで、エラー Bean とナビゲーション・ターゲットの両方をエラー・ページ用に定義する必要もあります。エラー Bean の名前には BPCError、エラー・ページのナビゲーション・ターゲットの名前には error が使用されていることを確認します。

```
<faces-config>
...
<managed-bean>
  <managed-bean-name>BPCError</managed-bean-name>
  <managed-bean-class>com.ibm.bpc.clientcore.util.ErrorBeanImpl
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

...
<navigation-rule>
...
<navigation-case>
<description>
The general error page.
</description>
<from-outcome>error</from-outcome>
<to-view-id>/Error.jsp</to-view-id>
</navigation-case>
...
</navigation-rule>
</faces-config>
```

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。

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。

アプリケーションを Network Deployment 環境でデプロイしている場合、ターゲット・リソースの Java Naming and Directory Interface (JNDI) 名を、Business Flow Manager および Human Task Manager API をセル内で検出するための値に変更してください。

- ビジネス・プロセス・コンテナーが同じ管理対象セル内の別のサーバー上で構成されている場合、名前には以下の構造があります。

  cell/*nodes*/*nodename*/*servers*/*servername*/com/ibm/bpe/api/BusinessManagerHome
  cell/*nodes*/*nodename*/*servers*/*servername*/com/ibm/task/api/HumanTaskManagerHome

- ビジネス・プロセス・コンテナーが同じセル内のクラスターで構成されている場合、名前には以下の構造があります。

  cell/*clusters*/*clustername*/com/ibm/bpe/api/BusinessFlowManagerHome
  cell/*clusters*/*clustername*/com/ibm/task/api/HumanTaskManagerHome

EJB 参照を 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 70. 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 (JNDI name)	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

タスクの結果

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。

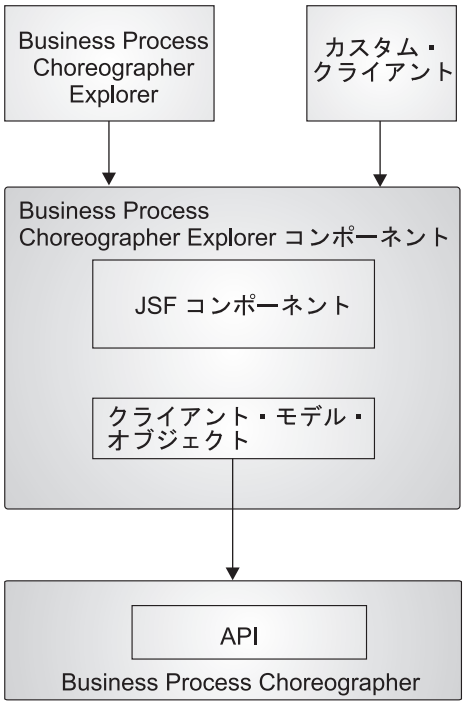
次のタスク

プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

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、Web 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のいずれか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 コンポーネントとともに使用されます。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。

JSF コンポーネントでのエラー処理

JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。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。

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 エラーとして生成された場合
- ClientException エラーがコマンドの execute メソッドによって生成され、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実装を使用できます。

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();
}
```

関連概念

510 ページの『List コンポーネントでのエラー処理』
List コンポーネントを使用して、JSF アプリケーション内のリストを表示する場合、com.ibm.bpe.jsf.handler.BPCListHandler クラスが提供するエラー処理機能を利用できます。

クライアント・モデル・オブジェクトのデフォルトのコンバーターおよびラベル

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。

List コンポーネントと Details コンポーネントはあらゆる Bean で機能します。Bean のプロパティーをすべて表示できます。ただし、Bean のプロパティーに使用されるコンバーターとラベルを設定する場合は、 List コンポーネントの `column` タグまたは Details コンポーネントの `property` タグを使用する必要があります。コンバーターとラベルを設定する代わりに、プロパティーのデフォルトのコンバーターとラベルを定義できます。これらを定義するには、次の静的メソッドを定義します。定義できる静的メソッドを次に示します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

次の表に、対応する Business Flow Manager API クラスと Human Task Manager API クラスを実装し、そのプロパティーにデフォルトのラベルとコンバーターを提供するクライアント・モデル・オブジェクトを示します。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 71. 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

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。

手順

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>TaskPoolQuery</managed-bean-name>
   <managed-bean-class>sample.TaskPoolQuery</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>

<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>
</managed-bean>
```

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。

照会の結果を強い型の Bean のリストとして表すことができる限り、任意のタイプの照会ロジックを JSF アプリケーションに追加できます。例えば、TaskPoolQuery は com.ibm.task.clientmodel.bean.TaskInstanceBean オブジェクトのリストを使用して実装されます。

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 TaskPoolQuery 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.
    //
    // Add the database columns for all of the properties you want to show
    // in your list to the select statement
    //
    QueryResultSet queryResult = taskService.query(
     "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
      + "TASK.STARTER, TASK.OWNER, 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)",
      (String)null,
      (Integer)null,
      (TimeZone)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 execute 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。

タスクの結果

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。

リストの処理方法

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選択された項目をトラッキングし、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る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。このインターフェースの実装は、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。

リスト内のリンクがクリックされると、通知がトリガーされます。 **action** 属性が設定されているすべての列についてリンクがレンダリングされます。 **action** 属性の値は JSF ナビゲーション・ターゲットか、JSF ナビゲーション・ターゲットを戻す JSF アクション・メソッドのいずれかです。

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。

照会の実装

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実装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、照会実装の 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。

デフォルトのコンバーターおよびラベル

照会によって戻される項目は Bean である必要があり、そのクラスは BPCListHandler クラスまたは com.ibm.bpc.clientcore.Query インターフェースの定義でタイプとして指定されたクラスと一致している必要があります。さらに、List コンポーネントは、項目クラスまたはスーパークラスが以下のメソッドを実装しているかどうかを検査します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

これらのメソッドが Bean に対して定義されている場合、List コンポーネントはリストのデフォルト・ラベルとして label を使用し、プロパティーのデフォルト・コンバーターとして SimpleConverter を使用します。これらの設定は、`bpe:list` タグの **label** および **converterID** 属性で上書きできます。詳しくは、SimpleConverter インターフェースおよび ColumnTag クラスの Javadoc を参照してください。

ユーザー固有の時間帯情報

JavaServer Faces (JSF) コンポーネントは、List コンポーネントのユーザー指定の時間帯情報を処理するためのユーティリティーを提供します。

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実装が **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();
}
```

List コンポーネントでのエラー処理

List コンポーネントを使用して、JSF アプリケーション内のリストを表示する場合、com.ibm.bpe.jsf.handler.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 execute 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実装に渡します。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。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。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。

関連概念

504 ページの『JSF コンポーネントでのエラー処理』
JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。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。

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` と `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で処理を繰り返し、オブジェクトごとにすべての列をレンダリングします。

表 72. bpe:list 属性

属性	必須	説明
buttonStyleClass	いいえ	フッター領域内のボタン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・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。
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。この属性には `true` または `false` のいずれかの値が使用されます。値が `true` に設定されている場合は、チェック・ボックス列がレンダリングされます。
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。
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。
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。この属性では、値の式はサポートされていません。
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。
selectAll	いいえ	この属性が `true` に設定されている場合は、リスト内のすべての項目がデフォルトで選択されます。
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の CSS スタイル・クラス。

表 73. bpe:column 属性

属性	必須	説明
action	いいえ	この属性が指定されている場合は、列でリンクがレンダリングされます。リンクがクリックされると、JavaServer Faces アクション・メソッドまたは Faces ナビゲーション・ターゲットが起動されます。シグニチャーが String method() である JavaServer Faces アクション・メソッド。
converterID	いいえ	プロパティー値の変換に使用する Faces コンバーター ID。この属性が設定されていない場合、モデルによりこのプロパティーに設定された Faces コンバーター ID が使用されます。

表 73. bpe:column 属性 (続き)

属性	必須	説明
label	いいえ	列のヘッダーのラベルまたはテーブル・ヘッダー行のセルのラベルとして使用されるリテラルまたは値バインディング式。この属性が設定されていない場合、モデルによりこのプロパティーに設定されたラベルが使用されます。
name	はい	この列に表示されるプロパティーの名前。

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。

手順

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、**model** 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。

   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>
   ```

   model 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。Bean クラスは任意の JavaBeans クラスにすることができます。Bean がデフォルト・コンバーターおよびプロパティー・ラベルを提供する場合、そのコンバーターおよびラベルが、List コンポーネントの場合と同じようにしてレンダリングに使用されます。

タスクの結果

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 と 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 74. bpe:details 属性

属性	必須	説明
columnClasses	いいえ	列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) のスタイル・クラスをコンマで区切ったリスト。
id	いいえ	コンポーネントの JavaServer Faces ID。
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。
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・クラスのリスト。
styleClass	いいえ	HTML エレメントのレンダリングに使用される CSS クラス。

表 75. bpe:property 属性

属性	必須	説明
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。
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。
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。

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。

このタスクについて

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。 CommandBar コンポーネントは、JavaServer Faces (JSF) アプリケーションに追加して拡張することができます。

手順

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、タスク・インスタンス・リストに refresh コマンドと claim コマンドを提供する CommandBar コンポーネントを追加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ともに使用されます。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、タスク・インスタンス・リストの選択されたオブジェクトを提供します。この Bean は、ItemProvider 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。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。

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 **model** 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。モデルで BPCListHandler クラスも BPCDetailsHandler クラスも使用しない場合は、ItemProvider インターフェースを実装したクラスを所有する別のオブジェクトを参照する必要があります。

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、Command インターフェースを実装するコマンド・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。

```
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. オプション: **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・コマンド実装を参照します。つまりこのコマンドは、public String execute(List selectedObjects) throws ClientException メソッドをインプリメントする必要があります。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. オプション: 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 例外以外の例外を生成した場合、**action** 属性は処理されません。

d. **commandClass** 属性にコマンド・クラスが指定されていない場合、アクションが即時に呼び出されます。例えば、例の中のリフレッシュ・コマンドの場合、JSF 値式 `#{TaskInstanceList.refreshList}` がコマンドの代わりに呼び出されます。

タスクの結果

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。

コマンドの処理方法

CommandBar コンポーネントを使用して、アプリケーションにアクション・ボタンを追加します。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。

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実装を示します。
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。

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` と `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 76. bpe:commandbar 属性

属性	必須	説明
buttonStyleClass	いいえ	コマンド・バー内のボタンのレンダリングに使用されるカスケーディング・スタイル・シート (CSS) スタイル・クラス。
id	いいえ	コンポーネントの JavaServer Faces ID。
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	コマンド・バーのレンダリングに使用される CSS スタイル・クラス。

表 77. bpe:command 属性

属性	必須	説明
action	いいえ	JavaServer Faces アクション・メソッドまたはコマンド・ボタンにより起動される Faces ナビゲーション・ターゲット。アクションにより戻されるナビゲーション・ターゲットは、その他のナビゲーション・ルールをすべて上書きします。このアクションは、例外がスローされない場合、またはコマンドから ErrorsInCommandException 例外がスローされる場合に呼び出されます。

表 77. bpe:command 属性 (続き)

属性	必須	説明
commandClass	いいえ	コマンド・クラスの名前。このクラスのインスタンスは CommandBar コンポーネントにより作成され、コマンド・ボタンが選択されると実行されます。
commandID	はい	コマンドの ID。
context	いいえ	**commandClass** 属性を使用して指定されたコマンドのコンテキストを提供するオブジェクト。コマンド・バーが初めてアクセスされると、コンテキスト・オブジェクトが取得されます。
immediate	いいえ	コマンドが起動される時期を指定します。この属性の値が `true` の場合は、ページの入力が処理される前にコマンドが起動されます。デフォルトは `false` です。
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。
rendered	いいえ	ボタンがレンダリングされるかどうかを判別します。属性の値は、ブール値または値の式のいずれかにすることができます。
styleClass	いいえ	ボタンのレンダリングに使用される CSS スタイル・クラス。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。

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。

このタスクについて

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。

手順

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・ハンドラーの itemListener リスト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。

タスクの結果

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 78. bpe:form 属性

属性	必須	説明
id	いいえ	コンポーネントの JavaServer Faces ID。
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。
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。
simplification	いいえ	この属性が `true` に設定されている場合は、単純型が含まれているプロパティーおよび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 `true` 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。
styleClass4invalidInput	いいえ	無効な入力のレンダリング用の CSS スタイル・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。
styleClass4validInput	いいえ	有効な入力のレンダリング用の CSS スタイル・クラス名。

タスクおよびプロセス・メッセージ用の JSP ページの開発

Business Process Choreographer Explorer インターフェースには、ビジネス・データの表示や入力のためのデフォルトの入力フォームおよび出力フォームが用意されています。 JSP ページを使用して、カスタマイズされた入力フォームおよび出力フォームをカスタマイズできます。

このタスクについて

ユーザー定義の JavaServer Pages (JSP) ページを Web クライアントに組み込むには、WebSphere Integration Developer でヒューマン・タスクをモデル化するときにこれらのページを指定する必要があります。例えば、JSP ページの指定先は、特定のタスクやその入出力メッセージ、特定のユーザーのロールまたはすべてのユーザーのロールのいずれでも構いません。ユーザー定義 JSP ページは、出力データを表示して入力データを収集するために実行時にユーザー・インターフェースに組み込まれます。

カスタマイズ・フォームは自己完結した Web ページではなく、Business Process Choreographer Explorer によって HTML フォームに組み込まれる HTML フラグメントです。例えば、メッセージのすべてのラベルや入力フィールドのフラグメントがこれに該当します。

カスタマイズ・フォームがあるページでボタンをクリックすると、入力データは Business Process Choreographer Explorer に送信されて検証されます。検証は、提供されたプロパティーのタイプとブラウザーで使用されているロケールに基づいて行われます。 入力データを検証できない場合は同じページがもう一度表示され、検証エラーの情報が messageValidationErrors 要求属性に書き込まれます。情報はマップとして提供され、このマップは、無効なプロパティーの XML Path Expression (XPath) を、発生した妥当性検査例外にマップします。

カスタマイズ・フォームを Business Process Choreographer Explorer に追加するには、WebSphere Integration Developer を使用して以下のステップを実行します。

手順

1. カスタマイズ・フォームを作成します。

   Web インターフェースで使用される、入出力フォーム用のユーザー定義 JSP ページは、メッセージ・データにアクセスする必要があります。 JSP 内の Java 断片または JSP 実行言語を使用して、メッセージ・データにアクセスします。フォーム内のデータは要求コンテキストを介して使用可能です。
2. JSP ページにタスクを割り当てます。

   ヒューマン・タスク・エディターでヒューマン・タスクを開きます。 クライアント設定で、ユーザー定義 JSP ページの場所と、カスタマイズ・フォームの適用先のロール (例: 管理者) を指定します。Business Process Choreographer Explorer のクライアント設定は、タスク・テンプレートに格納されます。これらの設定は、実行時にタスク・テンプレートを使用して取得されます。
3. ユーザー定義 JSP ページを Web アーカイブ (WAR ファイル) にパッケージ化します。

   WAR ファイルは、タスクが格納されているモジュールと一緒にエンタープライズ・アーカイブに組み込むことも、個別に配置することもできます。JSP が個別にデプロイされる場合、Business Process Choreographer Explorer またはカスタム・クライアントがデプロイされるサーバー上で JSP を使用可能にしてください。

プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

タスクの結果

カスタマイズ・フォームは、Business Process Choreographer Explorer で実行時にレンダリングされます。

ユーザー定義 JSP フラグメント

ユーザー定義 JavaServer Pages (JSP) フラグメントは、HTML フォーム・タグに埋め込まれます。Business Process Choreographer Explorer は、実行時にこれらのフラグメントを、レンダリングされるページに埋め込みます。

入力メッセージのユーザー定義 JSP フラグメントは、出力メッセージの JSP フラグメントより先に埋め込まれます。

```
<html....>
   ...
   <form...>
     Input JSP (display task input message)

     Output JSP (display task output message)

   </form>
   ...
</html>
```

ユーザー定義 JSP フラグメントは、HTML フォーム・タグに埋め込まれているため、入力要素を追加できます。 入力要素の名前は、データ要素の XML Path Language (XPath) 表現と一致する必要があります。入力要素の名前には、以下に示す提供されたプレフィックス値を使用してプレフィックスを付けることが重要です。

```
<input id="address"
       type="text"
       name="${prefix}/selectPromotionalGiftResponse/address"
       value="${messageMap['/selectPromotionalGiftResponse/address"]}
       size="60"
       align="left" />
```

プレフィックス値は要求属性として提供されます。この属性により、引用符で囲まれた書式内の入力名は固有であることが保証されます。プレフィックスは Business Process Choreographer Explorer によって次のように生成されるため、変更しないでください。

```
String prefix = (String)request.getAttribute("prefix");
```

プレフィックス要素が設定されるのは、与えられたコンテキストにおいてメッセージが編集できる場合に限ります。 出力データは、ヒューマン・タスクの状態に応じてさまざまな方法で表示できます。例えば、タスクが要求状態にある場合は、出力データを変更できます。ただし、タスクが完了状態にある場合、出力データは表示専用になります。JSP フラグメントでは、プレフィックス要素が存在し、それに応じてメッセージを表示するかどうかをテストできます。 以下の JSTL ステートメントでは、プレフィックス要素が設定されているかどうかをテストする 1 つの方法を示しています。

```
...
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<c:choose>
  <c:when test="${not empty prefix}">
    <!--Read/write mode-->
  </c:when>
  <c:otherwise>
    <!--Read-only mode-->
  </c:otherwise>
</c:choose>
```

ヒューマン・タスク機能をカスタマイズするプラグインの作成

Business Process Choreographer では、ヒューマン・タスクの処理中に発生するイベントのイベント処理インフラストラクチャーを提供します。ニーズに合わせて機能を適応させることができるように、プラグイン点も設けられます。サービス・プロバイダー・インターフェース (SPI) を使用すると、イベントの処理および担当者照会結果の後処理を行うようにカスタマイズされたプラグインを作成できます。

このタスクについて

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。また、担当者解決から戻される結果を処理するプラグインを作成することもできます。例えば、ピーク時に結果リストにユーザーを追加して、ワークロードのバランスを取ることもできます。

プラグインを使用する前に、それらのプラグインをインストールして登録する必要があります。担当者照会結果を後処理するプラグインは TaskContainer アプリケーションに登録できます。これにより、プラグインはすべてのタスクで使用できるようになります。

Business Process Choreographer 用の API イベント・ハンドラーの作成

API イベントは、API メソッドがヒューマン・タスクを操作するときに発生します。 API イベント・ハンドラー・プラグインのサービス・プロバイダー・インターフェース (SPI) を使用して、API イベントまたは同等の API イベントを持つ内部イベントが送信するタスク・イベントを処理するプラグインを作成します。

このタスクについて

API イベント・ハンドラーを作成するには、次のステップを実行します。

手順

1. APIEventHandlerPlugin5 インターフェースを実装するクラス、または APIEventHandler 実装クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。
   - APIEventHandlerPlugin5 インターフェースを使用する場合は、APIEventHandlerPlugin5 インターフェースおよび APIEventHandlerPlugin インターフェースのすべてのメソッドを実装する必要があります。

- APIEventHandler 実装クラスを拡張する場合は、必要なメソッドを上書きしてください。

このクラスは、Java Platform Enterprise Edition (Java EE) エンタープライズ・アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

注: このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。このアクションを行うと、データベース内のタスク・データが矛盾するおそれがあります。

2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

次のいずれかの方法で、JAR ファイルを使用可能にすることができます。

- アプリケーション EAR ファイル内のユーティリティー JAR ファイルとする。
- アプリケーション EAR ファイルとともにインストールされる共用ライブラリーとする。
- TaskContainer アプリケーションとともにインストールされる共用ライブラリーとする。この場合、プラグインはすべてのタスクで使用できるようになります。

3. JAR ファイルの `META-INF/services/` ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java EE サービス・プロバイダー・インターフェース仕様に準拠します。

a. `com.ibm.task.spi.`*`plug-in_name`*`APIEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

例えば、`Customer` という名前のプラグインが com.ibm.task.spi.APIEventHandlerPlugin5 インターフェースを実装する場合、構成ファイルの名前は `com.ibm.task.spi.`**`Customer`**`APIEventHandlerPlugin` になります。

b. このファイルのコメント行 (番号記号 (#) で始まる行) とブランク行を除く最初の行に、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

例えば、`MyAPIEventHandler` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.MyAPIEventHandler` という項目が含まれていなければなりません。

タスクの結果

ファイルには、API イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイルがあります。

注: API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン実装は同じ名前でなければなりません (例えば、SPI 実装のイベント・ハンドラー名として `Customer` など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの `META-INF/services/` ディレクトリーに 2 つのファイルを作成する必要があります (com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

実装への変更を有効にするには、共用ライブラリー内の JAR ファイルを置き換えて、関連 EAR ファイルを再デプロイし、サーバーを再始動します。

次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

API イベント・ハンドラー

API イベントは、ヒューマン・タスクが変更されるか、その状態が変化するときに発生します。これらの API イベントを処理するために、タスクが変更される前 (pre-event メソッド)、および API 呼び出しが戻る直前 (post-event メソッド) に、イベント・ハンドラーが直接呼び出されます。

pre-event メソッドが ApplicationVetoException 例外をスローすると、API アクションが実行されずに API の呼び出し元にその例外が戻され、イベントに関連付けられたトランザクションがロールバックされます。pre-event メソッドが内部イベントによってトリガーされ、ApplicationVetoException 例外がスローされた場合は、自動要求などの内部イベントが実行されなくても、例外はクライアント・アプリケーションに戻されません。この場合、情報メッセージが SystemOut.log ファイルに書き込まれます。API メソッドが処理中に例外をスローすると、その例外はキャッチされ、post-event メソッドに渡されます。その例外は、post-event メソッドが戻った後に再び呼び出し元に渡されます。

以下の規則が、pre-event メソッドに適用されます。

- pre-event メソッドは、関連した API メソッドまたは内部イベントのパラメーターを受け取ります。
- pre-event メソッドは、ApplicationVetoException 例外をスローして処理の続行を阻止することができます。

以下の規則が、post-event メソッドに適用されます。

- post-event メソッドは、API 呼び出しに提供されたパラメーター、および戻り値を受け取ります。API メソッド実装によって例外がスローされると、post-event メソッドも例外を受け取ります。

- post-event メソッドは、戻り値を変更できません。
- post-event メソッドは例外をスローできません。ランタイム例外がログに記録され、処理を続行できないようにします。

API イベント・ハンドラーをインプリメントするには、APIEventHandlerPlugin インターフェースを拡張する APIEventHandlerPlugin3 インターフェースをインプリメントするか、またはデフォルトの com.ibm.task.spi.APIEventHandler SPI 実装クラスを拡張します。イベント・ハンドラーは、デフォルトの実装クラスから継承する場合、常に最新バージョンの SPI をインプリメントします。新しいバージョンの Business Process Choreographer にアップグレードする場合は、新しい SPI メソッドを利用すると必要な変更も少なくなります。

通知イベント・ハンドラーと API イベント・ハンドラーの両方がある場合、イベント・ハンドラー名は 1 つしか登録できないので、両方のハンドラーの名前は同じでなければなりません。

Business Process Choreographer 用の通知イベント・ハンドラーの作成

通知イベントは、ヒューマン・タスクがエスカレートされるときに生成されます。Business Process Choreographer には、エスカレーション作業項目の作成や電子メールの送信などのエスカレーションを処理する機能があります。通知イベント・ハンドラーを作成して、エスカレーションの処理方法をカスタマイズすることができます。

このタスクについて

通知イベント・ハンドラーをインプリメントするには、NotificationEventHandlerPlugin インターフェースをインプリメントする方法と、デフォルトの com.ibm.task.spi.NotificationEventHandler サービス・プロバイダー・インターフェース (SPI) 実装クラスを拡張する方法があります。

通知イベント・ハンドラーを作成するには、次のステップを実行します。

手順

1. NotificationEventHandlerPlugin インターフェースをインプリメントするクラス、または NotificationEventHandler 実装クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。

   NotificationEventHandlerPlugin インターフェースを使用する場合は、すべてのインターフェース・メソッドをインプリメントする必要があります。SPI 実装クラスを拡張する場合は、必要なメソッドを上書きしてください。

   このクラスは、Java Platform Enterprise Edition (Java EE) エンタープライズ・アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

   プラグインは、EscalationUser ロールの権限で呼び出されます。このロールは、ヒューマン・タスク・コンテナーの構成時に定義されます。

注: このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。このアクションを行うと、データベース内のタスク・データが矛盾するおそれがあります。

2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   次のいずれかの方法で、JAR ファイルを使用可能にすることができます。

   - アプリケーション EAR ファイル内のユーティリティー JAR ファイルとする。
   - アプリケーション EAR ファイルとともにインストールされる共用ライブラリーとする。
   - TaskContainer アプリケーションとともにインストールされる共用ライブラリーとする。この場合、プラグインはすべてのタスクで使用できるようになります。

3. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   ヘルパー・クラスが数個の Java EE アプリケーションによって使用される場合は、共用ライブラリーとして登録する別の JAR ファイルにこれらのクラスをパッケージできます。

4. JAR ファイルの `META-INF/services/` ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java EE サービス・プロバイダー・インターフェース仕様に準拠します。

   a. `com.ibm.task.spi.`*`plug-in_name`*`NotificationEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

      例えば、`HelpDeskRequest` (イベント・ハンドラー名) という名前のプラグインが com.ibm.task.spi.NotificationEventHandlerPlugin インターフェースをインプリメントする場合、構成ファイルの名前は `com.ibm.task.spi.`**`HelpDeskRequest`**`NotificationEventHandlerPlugin` になります。

   b. このファイルのコメント行 (番号記号 (#) で始まる行) とブランク行を除く最初の行に、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

      例えば、`MyEventHandler` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.MyEventHandler` という項目が含まれていなければなりません。

タスクの結果

ファイルには、通知イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイ

ルがあります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

注: API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン実装は同じ名前でなければなりません (例えば、SPI 実装のイベント・ハンドラー名として `Customer` など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの `META-INF/services/` ディレクトリーに 2 つのファイルを作成する必要があります (com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

実装への変更を有効にするには、共用ライブラリー内の JAR ファイルを置き換えて、関連 EAR ファイルを再デプロイし、サーバーを再始動します。

次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。通知イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

ヒューマン・タスク用の API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインのインストール

API イベント・ハンドラーまたは通知イベント・ハンドラーのプラグインを使用するには、ヒューマン・タスク・コンテナーからアクセスできるようにプラグインをインストールする必要があります。

このタスクについて

プラグインのインストール方法は、そのプラグインが 1 つの Java Platform Enterprise Edition (Java EE) アプリケーションでのみ使用されるか、複数のアプリケーションで使用されるかによって異なります。

次のステップのいずれかを実行してプラグインをインストールします。

手順

- 1 つの Java EE アプリケーションによって使用される場合のプラグインをインストールします。

  プラグインの JAR ファイルをアプリケーションの EAR ファイルに追加します。WebSphere Integration Developer のデプロイメント記述子エディターで、プラグインの JAR ファイルを、主要な Enterprise JavaBeans (EJB) モジュールの Java EE アプリケーション用のプロジェクト・ユーティリティー JAR ファイルとしてインストールします。

- 複数の Java EE アプリケーションによって使用される場合のプラグインをインストールします。

  JAR ファイルを WebSphere Application Server 共用ライブラリーに入れ、そのライブラリーを、プラグインへのアクセスが必要なアプリケーションと関連付けます。JAR ファイルをネットワーク・デプロイメント環境で使用できるようにするには、アプリケーションがデプロイされているサーバーまたはクラスター・メンバーをホストする各ノードに手動で JAR ファイルを配布します。アプリケーションのデプロイメント・ターゲット・スコープ (アプリケーションがデプロイされているサーバーまたはクラスター)、またはセル・スコープを使用できます。これによりプラグイン・クラスは、選択されたデプロイメント・スコープ全体で可視になることに注意してください。

次のタスク

これで、プラグインを登録できます。

API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインをタスク・テンプレート、タスク・モデル、およびタスクに登録する

API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインをタスク、タスク・テンプレート、およびタスク・モデルに登録できます。この登録は、随時タスクの作成、既存タスクの更新、随時タスク・モデルの作成、またはタスク・テンプレートの定義のときに行うことができます。

このタスクについて

API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインは、以下のレベルのタスクに登録できます。

タスク・テンプレート
このテンプレートを使用して作成されるすべてのタスクは同じハンドラーを使用します

随時タスク・モデル
このモデルを使用して作成されるタスクは同じハンドラーを使用します

随時タスク
作成されたタスクは、指定されたハンドラーを使用します

既存タスク
タスクは、指定されたハンドラーを使用します

プラグインは、次のいずれかの方法で登録できます。

手順

- WebSphere Integration Developer でモデル化されたタスク・テンプレートの場合は、タスク・モデル内にプラグインを指定します。
- 随時タスクまたは随時タスク・モデルの場合は、タスクまたはタスク・モデルの作成時にプラグインを指定します。

TTask クラスの setEventHandlerName メソッドを使用して、イベント・ハンドラーの名前を登録します。

- 実行時に、タスク・インスタンスのイベント・ハンドラーを変更します。

  update(Task task) メソッドを使用して、実行時にタスク・インスタンスの別のイベント・ハンドラーを使用します。呼び出し元には、このプロパティーを更新するタスク管理者権限が必要です。

担当者照会結果の後処理を行うプラグインの使用

Business Process Choreographer の担当者解決では、特定のロール (例えば、タスクの潜在的な所有者) に割り当てられているユーザーのリストが返されます。担当者解決で返される担当者照会の結果を変更するプラグインを作成できます。例えば、ワークロード・バランシングを改善するために、既にワークロードが高いユーザーを照会結果から除去することができます。

このタスクについて

担当者割り当ておよび担当者の代替によって返される結果を変更するには、プラグイン・インターフェースを実装するクラスを作成した後、そのプラグインの JAR ファイルをアセンブルし、それをインストールしてアクティブ化する必要があります。

担当者照会結果の後処理を行うプラグインを作成するには、次のステップを実行します。

手順

1. 担当者照会結果の後処理プラグインを実装します。
   StaffQueryResultPostProcessorPlugin インターフェースまたは StaffQueryResultPostProcessorPlugin2 インターフェースのいずれかを実装するクラスを作成します。
2. インストール可能な JAR ファイルを作成します。
   a. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。
   b. JAR ファイルの `META-INF/services/` ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java EE サービス・プロバイダー・インターフェース仕様に準拠している必要があります。
      1) テキスト・エディターで、`com.ibm.task.spi.`*`plug-in_name`*`StaffQueryResultPostProcessorPlugin` という名前のサービス・プロバイダー構成ファイルを作成します。ここで、*plug-in_name* はプラグインの名前です。構成ファイルの名前は、実装したインターフェースの名前に依存しません。例えば、`MyHandler` という名前のプラグインが com.ibm.task.spi.StaffQueryResultPostProcessorPlugin2 インターフェースを実装する場合、構成ファイルの名前は `com.ibm.task.spi.`**`MyHandler`**`StaffQueryResultPostProcessorPlugin` になります。

2) このファイルの、コメント行 (番号記号 (#) で始まる行) でもブランク行でもない最初の行で、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。例えば、 `StaffPostProcessor` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.StaffPostProcessor` という項目が含まれている必要があります。

担当者照会結果を後処理するプラグインを含むインストール可能 JAR ファイルと、そのプラグインのロードに使用できるサービス・プロバイダー構成ファイルができます。

3. アプリケーション・サーバーの共用ライブラリーに JAR ファイルをインストールして、 Human Task Manager アプリケーションに関連付けます。
   a. Business Process Choreographer が構成されているサーバーまたはクラスターのスコープで、このプラグインの WebSphere Application Server 共用ライブラリーを定義します。
   b. 共用ライブラリーを `TaskContainer` アプリケーションと関連付けます。
   c. サーバーまたはクラスター・メンバーをホストしていて、影響を受ける各 WebSphere Process Server インストール済み環境に対して、プラグイン JAR ファイルを使用可能にします。
4. プラグインを使用するように、Human Task Manager を構成します。
   a. 管理コンソールで、Human Task Manager の「カスタム・プロパティー」ページに移動します。

      「**サーバー**」 → 「**クラスター**」 → 「**WebSphere Application Server クラスター**」 → 「***クラスター名***」または「**サーバー**」 → 「**サーバー・タイプ**」 → 「**WebSphere Application Server**」 → 「***サーバー名***」のいずれかをクリック後、「**構成**」タブの「**ビジネス・インテグレーション**」セクションで、「**Business Process Choreographer**」を展開し、「**Human Task Manager**」をクリックします。「**追加プロパティー**」の下の「**カスタム・プロパティー**」を選択します。
   b. **Staff.PostProcessorPlugin** という名前と、プラグインにつけた名前の値 (例: `MyHandler`) を持つカスタム・プロパティーを追加します。

   これで、プラグインは担当者照会結果の後処理に使用できるようになりました。
5. プラグインを有効にするために、サーバーを再始動します。 後処理プラグインは、担当者の割り当てと担当者の代替の両方を実行した後に呼び出されます。

   注: プラグインを変更する場合は、共用ライブラリー内の JAR ファイルを置き換えて、サーバーを再始動する必要があります。

第 2 部 アプリケーションのデプロイ

第 5 章 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 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、ライブラリー内にある成果物を使用することが多くあります。ライブラリーは、共有リソースの保管に使用される WebSphere Integration Developer の特殊プロジェクトです。デプロイメント時に、WebSphere Integration Developer ライブラリーはユーティリティー JAR ファイルに変換され、実行されるアプリケーション内にパッケージ化されます。

モジュールの開発時に、特定のリソースまたはコンポーネントが、他のモジュールでも使用できると判断する場合があります。これらの成果物は、ライブラリーを使用して共有できます。

ライブラリーの概要

ライブラリーは、開発、バージョン管理、および共有リソース (通常モジュール間で共有されるリソース) の編成に使用される WebSphere Integration Developer 内の特殊プロジェクトです。以下に示すような成果物タイプのサブセットのみを、作成してライブラリーに保管できます。

- 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)

デプロイメント時に、これらの WebSphere Integration Developer ライブラリーは、実行されるアプリケーション内のユーティリティー JAR ファイルに変換されます。

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。図 78 で、アプリケーションにコンポーネントおよび関連ライブラリーが含まれている様子を示します。

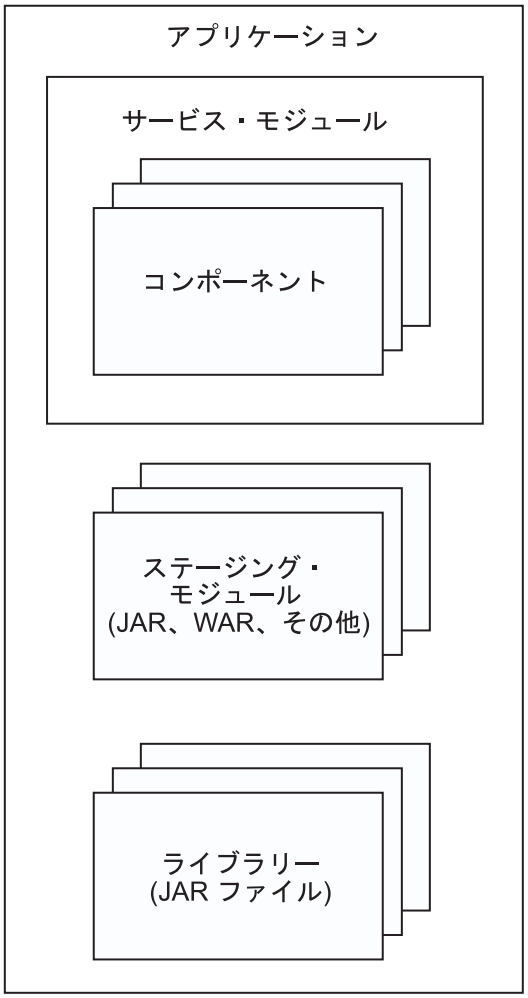


図 78. モジュール、コンポーネント、およびライブラリー間の関係

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 プラットフォーム、Enterprise Edition 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、その他のすべての従属 JAR、Web サービス・アーカイブ (WAR)、リソース・アーカイブ (RAR)、ステージング・ライブラリー (Enterprise Java Beans (EJB)) JAR ファイ

ル、およびその他のアーカイブが必要です。次に、serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを作成します。

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*Web

注: *myService*EJB および *myService*EJBClient ステージング・モジュールは、serviceDeploy によって作成されなくなります。ただし、それらのファイル名は、まだ serviceDeploy コマンドによって削除される可能性があるため、使用しないでください。

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。

始める前に

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、パフォーマンスの問題などの問題点を解決しておく必要があります。

重要: デプロイメント環境で既に稼働中のアプリケーションまたはモジュールの置換を防ぐために、モジュールまたはアプリケーションの名前が既にインストール済みのものとは異なる固有の名前であることを確認してください。

このタスクについて

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デプロイできるように正しいファイルにパッケージされているかどうかを検証します。

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は、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。

手順

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探します。

   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 のような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すべて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ないようにしてください。『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 (540 ページ) 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・フォルダー内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確立します。

7. マニフェスト・ファイル (*module*.manifest) がコンポーネント・フォルダー内に存在することを確認します。

   マニフェストは、モジュールおよび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、実動サーバーにインストールするモジュールを準備するために使用する serviceDeploy コマンドへの入力として使用します。

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
```

次のタスク

『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。

第 6 章 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) 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。

始める前に

アプリケーションをインストール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、Business Flow Manager と Human Task Manager がインストールされ、構成されていることを確認します。

このタスクについて

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。

タスクの結果

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・アプリケーションがインストールされると、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・テンプレートは開始状態になります。これらのテンプレートから、プロセス・インスタンスとタスク・インスタンスを作成できます。

次のタスク

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。

Network Deployment 環境へのビジネス・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方法

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 Network Deployment 環境にインストールするときに、アプリケーションのインストール機能により以下のアクションが自動的に実行されます。

アプリケーションは、段階的にインストールされます。次の段階を開始する前に、前の段階が正常に完了している必要があります。

1. アプリケーションのインストールは Deployment Manager で開始されます。

   この段階では、ビジネス・プロセス・テンプレートとヒューマン・タスク・テンプレートが、WebSphere 構成リポジトリーで構成されます。アプリケーションの検証も行われます。エラーが発生した場合、エラーは System.out ファイルまたは System.err ファイルに報告されるか、あるいは Deployment Manager で FFDC エントリーとして報告されます。

2. アプリケーションのインストールはノード・エージェントで続行されます。

   この段階では、1 つのアプリケーション・サーバー・インスタンスでのアプリケーションのインストールが開始されます。このアプリケーション・サーバー・インスタンスは、デプロイメント・ターゲットであるか、またはその一部です。デプロイメント・ターゲットが、複数のクラスター・メンバーで構成されるクラスターの場合は、このクラスターのクラスター・メンバーからサーバー・インスタンスが任意に選択されます。この段階でエラーが発生した場合、エラーは SystemOut.log ファイルまたは SystemErr.log ファイルに報告されるか、あるいはノード・エージェントで FFDC エントリーとして報告されます。
3. サーバー・インスタンスでアプリケーションが実行されます。

   この段階では、プロセス・テンプレートとヒューマン・テンプレートがデプロイメント・ターゲットの Business Process Choreographer データベースに配置されます。エラーが発生した場合、System.out ファイルまたは SystemErr.log ファイルに報告されるか、あるいはこのサーバー・インスタンスで FFDC エントリーとして報告されます。

ビジネス・プロセスとヒューマン・タスクのデプロイメント

WebSphere Integration Developer または serviceDeploy を使用して、プロセス・コンポーネントまたはタスク・コンポーネントをエンタープライズ・アプリケーション (EAR) ファイルにパッケージ化します。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・プロセスまたはヒューマン・タスクが含まれてい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、これらのビジネス・プロセスまたはヒューマン・タスクは、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・テンプレートとして Business Process Choreographer データベースに格納されます。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。

異なるエンタープライズ・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。バージョンは、その有効開始日によって区別されます。新規エンタープライズ・アプリケーションのインストール時に、インストールされるテンプレートのバージョンが次のように決定されます。

- テンプレートの名前とターゲット名前空間がまだ存在していない場合は、新規テンプレートがインストールされます。
- テンプレート名とターゲット名前空間が、既存のテンプレートと同じであるが、有効開始日が異なる場合は、既存のテンプレートの新規バージョンがインストールされます。

注: テンプレート名は、ビジネス・プロセスまたはヒューマン・タスクではなく、コンポーネントの名前から派生します。

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。

- WebSphere Integration Developer を使用する場合、有効開始日はヒューマン・タスクまたはビジネス・プロセスがモデル化された日付になります。
- サービス・デプロイメントを使用する場合、有効開始日は、serviceDeploy コマンドが実行された日付になります。アプリケーションがインストールされた日付を有効開始日として取得するのは、コラボレーション・タスクだけです。

ビジネス・プロセス・アプリケーションおよびヒューマン・タスク・アプリケーションの対話式インストール

アプリケーションは、wsadmin ツールと installInteractive スクリプトを使用して、実行時に対話式でインストールできます。このスクリプトにより、管理コンソールを使用してアプリケーションをインストールする場合には変更することのできない設定を変更できます。

このタスクについて

ビジネス・プロセス・アプリケーションを対話式にインストールするには、次のステップを実行します。

手順

1. wsadmin ツールを開始します。

   profile_root/bin ディレクトリーで、wsadmin と入力します。
2. アプリケーションをインストールします。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminApp installInteractive application.ear
   ```

   ここで、*application.ear* は、ご使用のプロセス・アプリケーションを含むエンタープライズ・アーカイブ・ファイルの修飾名です。 一連のタスクにおいてプロンプトが出されるので、その時にアプリケーションの値を変更できます。
3. 構成変更を保管します。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminConfig save
   ```

   変更を保管し、マスター構成リポジトリーへの更新を転送する必要があります。スクリプト・プロセスが終了し、変更を保存していなければ、変更は廃棄されます。

プロセス・アプリケーションのデータ・ソースと設定参照の設定値の構成

特定のデータベース・インフラストラクチャーで SQL ステートメントを実行するプロセス・アプリケーションは、構成する必要があります。これらの SQL ステートメントは、情報サービス・アクティビティーから発生したり、プロセスのインストールまたはインスタンスの開始時に実行するステートメントであったりします。

このタスクについて

このアプリケーションのインストール時には、次のタイプのデータ・ソースを指定できます。

- プロセス・インストール時に SQL ステートメントを実行するデータ・ソース
- プロセス・インスタンスの開始時に SQL ステートメントを実行するデータ・ソース
- SQL 断片アクティビティーを実行するデータ・ソース

SQL 断片アクティビティーを実行するために必要なデータ・ソースは、tDataSource タイプの BPEL 変数で定義されます。SQL 断片アクティビティーで必要とされるデータベース・スキーマ名とテーブル名は、tSetReference タイプの BPEL 変数で定義されます。これらの両方の変数の初期値は、構成することができます。

wsadmin ツールを使用してデータ・ソースを指定できます。

手順

1. wsadmin ツールを使用して、プロセス・アプリケーションを対話式にインストールします。
2. データ・ソースと設定参照を更新するタスクになるまで、タスクをステップスルーします。

   ご使用の環境でこれらの設定を構成します。次の例に、これらのタスクごとに変更可能な設定を示します。
3. 変更を保管します。

例: wsadmin ツールを使用したデータ・ソースと設定参照の更新

「**Updating data source**」タスクでは、プロセスのインストール時またはプロセスの開始時に使用される初期変数値およびステートメントのデータ・ソース値を変更できます。「**Updating set references**」タスクでは、データベース・スキーマとテーブル名に関連した設定を構成できます。

```
Task [24]: Updating data sources

//Change data source values for initial variable values at process start

Process name:  Test
// Name of the process template
Process start or installation time:  Process start
// Indicates whether the specified value is evaluated
//at process startup or process installation
Statement or variable:  Variable
// Indicates that a data source variable is to be changed
Data source name:  MyDataSource
// Name of the variable
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name to jdbc/newName
Task [25]: Updating set references

// Change set reference values that are used as initial values for BPEL variables

Process name:  Test
// Name of the process template
Variable:  SetRef
// The BPEL variable name
```

```
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name of the data source of the set reference to jdbc/newName
Schema name: [IISAMPLE]
// The name of the database schema
Schema prefix: []:
// The schema name prefix.
// This setting applies only if the schema name is generated.
Table name: [SETREFTAB]: NEWTABLE
// Sets the name of the database table to NEWTABLE
Table prefix: []:
// The table name prefix.
// This setting applies only if the prefix name is generated.
```

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

管理コンソールを使用すると、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できます。

始める前に

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には、以下の条件を適用する必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。クラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を所有している必要があります。
- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよび管理対象サーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。
- 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・テンプレートのどのような状態のインスタンスも存在しません。あるいは、開発モードで稼働しているスタンドアロン・サーバーがあります。
- プロセス・インスタンスが新しいバージョンのプロセスにマイグレーションされたが、サービス呼び出しの応答を待機している場合、以前のバージョンを含むアプリケーションは、その応答が受信されるまでアンインストールできません。それ以外のすべての場合は、マイグレーションされたインスタンスは新しいバージョンのインスタンスであると見なされ、古いバージョンのプロセスを含むアプリケーションをアンインストールできます。

このタスクについて

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。

手順

1. 管理コンソールで、「**アプリケーション**」 → 「**アプリケーション・タイプ**」 → 「**WebSphere エンタープライズ・アプリケーション**」をクリックします。
2. 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。アプリケーションをアンインストールする前に、Business Process Choreographer Explorer を使用してインスタンスを削除するか、bpcTamplates.jacl 管理スクリプトの **-force** オプションを使用してこれらのインスタンスを停止して削除することができます。
3. アンインストールするアプリケーションを選択し、「**アンインストール**」をクリックします。
4. 「**保管**」をクリックして、変更を保管します。

タスクの結果

アプリケーションはアンインストールされます。

関連タスク

『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』

bpcTemplates.jacl スクリプトの使用には、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。

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

bpcTemplates.jacl スクリプトの使用には、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。

始める前に

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には、以下の条件を適用する必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。クラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を所有している必要があります。
- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよび管理対象サーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。

- 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。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。
- WebSphere 管理セキュリティーが使用可能になっていて、ご使用のユーザー ID にオペレーター権限も管理者権限もない場合は、wsadmin `-user` および `-password` のオプションを付けて、オペレーターまたは管理者の権限を持つユーザー ID を指定します。 `-uninstall` オプションを使用するにはオペレーター権限が必要であり、`-force` オプションを使用するには管理者権限が必要です。
- 次のうち、1 つ以上が当てはまります。
  - いずれの状態においてもビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートのインスタンスはありません。
  - **-force** オプションを使用する予定です。
  - 開発モードで稼働するスタンドアロン・サーバーがあります。
- プロセス・インスタンスが新しいバージョンのプロセスにマイグレーションされたが、サービス呼び出しの応答を待機している場合、以前のバージョンを含むアプリケーションは、その応答が受信されるまでアンインストールできません。それ以外のすべての場合は、マイグレーションされたインスタンスは新しいバージョンのインスタンスであると見なされ、古いバージョンのプロセスを含むアプリケーションをアンインストールできます。

このタスクについて

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。

手順

1. アンインストールするアプリケーションのテンプレートにプロセス・インスタンスまたはタスク・インスタンスがまだ関連付けられている場合は、以下のいずれか一方または両方を実行します。
   - Business Process Choreographer Explorer を使用して、インスタンスを削除します。
   - アンインストールするアプリケーションで定義されたプロセス・テンプレートに依存するビジネス・プロセスがほかに存在しないことが確実である場合は、**-force** オプションを使用できます。

     注意:
     このオプションと共にスクリプトを使用すると、テンプレートに関連付けられたすべてのインスタンスの削除、実行中のインスタンスに関連付けられたすべてのデータの削除、テンプレートの停止、およびアプリケーションのアンインストールが 1 つのステップで実行されます。このオプションを使用する場合は、細心の注意を払ってください。
2. 管理スクリプトがある Business Process Choreographer サブディレクトリーに移動します。 以下のコマンドを入力します。

   ```
   cd install_root/ProcessChoreographer/admin
   ```

Linux UNIX Linux および UNIX プラットフォームの場合は、以下のコマンドを入力します。

```
cd install_root/ProcessChoreographer/admin
```

i5/OS® プラットフォームの場合は、以下のコマンドを入力します。

```
cd install_root/ProcessChoreographer/admin
```

Windows Windows プラットフォームの場合は、以下のコマンドを入力します。

```
cd install_root¥ProcessChoreographer¥admin
```

3. 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。

Windows Windows プラットフォームの場合は、以下を入力します。

```
install_root¥bin¥wsadmin -f bpcTemplates.jacl
                          -uninstall application_name
                         [-force]
```

Linux UNIX Linux および UNIX プラットフォームの場合は、以下を入力します。

```
install_root/bin/wsadmin -f bpcTemplates.jacl
                          -uninstall application_name
                         [-force]
```

各部の意味は、次のとおりです。

-uninstall *application_name*
これは、アンインストールするアプリケーションの名前を指定します。

-force
このオプションは、アプリケーションがアンインストールされる前に、実行中のすべてのインスタンスを停止および削除させます。このオプションを使用する場合は注意が必要です。それは、このオプションを使用すると、実行中のインスタンスに関連するすべてのデータも削除されるからです。

タスクの結果

アプリケーションはアンインストールされます。

関連タスク

547 ページの『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』
管理コンソールを使用すると、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できます。

第 7 章 アダプターおよびそのインストール

アダプターを使用すると、開発アプリケーションとエンタープライズ情報システム内のその他のコンポーネントとの通信ができるようになります。

アダプターのインストールに使用するプロセスについては、WebSphere Integration Developer インフォメーション・センターのアダプターの構成および使用に説明があります。

第 8 章 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。

始める前に

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。

このタスクについて

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。

手順

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。

   SystemOut.log 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 Java EE 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 JCA アクティベーション・スペック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。不注意で変更を保存した場合、SIBus 宛先および JCA アクティベーション・スペック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『JCA アクティベーション・スペックの削除』を参照)。
2. アプリケーションが正しくインストールされている場合は、アプリケーションが正常に開始したかどうかを確認します。

   アプリケーションが正常に開始していない場合は、サーバーがアプリケーションのリソースを初期化しようとしたときに障害が起きています。

   a. SystemOut.log 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. アプリケーションで必要なリソースが使用可能か、また、それらのリソースが正常に開始されたかどうかを確認します。

開始されないリソースがあると、アプリケーションは実行されません。これは、情報が失われるのを防ぐためです。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) ファイルをビルドするテスト・システムの正しいフォルダーにあることを確認してください。『サーバーへのデプロイの準備』で追加情報について説明します。

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。 その別のアプリケーションがインストール済みで、正常に開始されていることを確認します。

b. 失敗したアプリケーションが使用する別のアプリケーションに含まれる、各種デバイス用のインポート・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確認します。 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。

JCA アクティベーション・スペックの削除

サービスを含むアプリケーションをインストールすると、システムによって JCA アプリケーションの仕様が作成されます。アプリケーションを再インストールする前に、この仕様を削除する必要がある場合があります。

始める前に

アプリケーションのインストールに失敗したために仕様を削除する場合、Java Naming and Directory Interface (JNDI) 名の中のモジュールとインストールできなかったモジュールの名前とが一致するようにしてください。JNDI 名の 2 番目の部分が、宛先をインプリメントしたモジュールの名前に相当します。例えば、`sca/SimpleBOCrsmA/ActivationSpec` の場合、**SimpleBOCrsmA** がモジュール名です。

このタスクで必要なセキュリティー・ロール: セキュリティーとロール・ベースの許可が有効になっている場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

このタスクについて

サービスを含むアプリケーションをインストールした後で間違って構成を保管したが、JCA アクティベーション・スペックは不要だという場合は、その仕様を削除します。

手順

1. 削除するアクティベーション・スペックを見つけます。

   仕様は「リソース・アダプター」パネルに表示されます。「**リソース**」>「**リソース・アダプター**」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、またはデプロイメント環境の「**サーバー**」スコープで作業する必要があります。

2. Platform Messaging Component SPI Resource Adapter に関連した JCA アクティベーション・スペックを表示します。

   リソース・アダプター名をクリックすると、次のパネルが表示され、関連した仕様が表示されます。

3. 削除するモジュール名に一致した「**JNDI 名**」の仕様をすべて削除します。

   a. 該当する仕様の横にあるチェック・ボックスをクリックします。

   b. 「**削除**」をクリックします。

タスクの結果

システムは、選択された仕様を表示から削除します。

次のタスク

変更を保管します。

SIBus 宛先の削除

サービス統合バス (SIBus) 宛先は、SCA モジュールによって処理中のメッセージを保持するために使用されます。問題が発生した場合、問題解決のためにバス宛先を除去しなければならない場合があります。

始める前に

アプリケーションのインストールに失敗したために宛先を削除する場合、宛先名の中のモジュールとインストールできなかったモジュールの名前とが一致するようにしてください。宛先の 2 番目の部分が、宛先をインプリメントしたモジュールの名前に相当します。例えば、`sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer` の場合、**SimpleBOCrsmA** がモジュール名です。

このタスクで必要なセキュリティー・ロール: セキュリティーとロール・ベースの許可が有効になっている場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

このタスクについて

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、または SIBus 宛先を必要としなくなった場合、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。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、アプリケーション・バスからもその項目を削除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『JCA アクティベーション・スペックの削除』を参照してください)。

手順

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。
   a. ナビゲーション・ペインで、「**サービス統合**」 → 「**バス**」をクリックします。
   b. コンテンツ・ペインで「**SCA.SYSTEM.*cell_name*.Bus**」をクリックします。
   c. 「宛先リソース」の下の「**宛先**」をクリックします。
3. 削除するモジュールと一致するモジュール名を持つ各宛先の隣にあるチェック・ボックスを選択します。
4. 「**削除**」をクリックします。

タスクの結果

パネルには残りの宛先のみが表示されます。

次のタスク

これらの宛先を作成したモジュールに関連した JCA アクティベーション・スペックを削除します。

Printed in Japan

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21