IBM WebSphere Business Integration Adapters
IBM WebSphere InterChange Server

コネクター開発ガイド (Java 用)

お願い

本書および本書で紹介する製品をご使用になる前に、 599 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM WebSphere InterChange Server バージョン 4.2.2、IBM WebSphere Business Integration Adapter Framework バージョン 2.4.1 に適用されます。

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

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

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

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

(URL は、変更になる場合があります)

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

原　典： IBM WebSphere Business Integration Adapters
IBM WebSphere InterChange Server
Connector Development Guide for Java

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2004.7

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

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

目次

本書について

IBM(R) WebSphere(R) Business Integration Adapters ポートフォリオは、主要な e-business テクノロジーやエンタープライズ・アプリケーション向けに統合コネクティビティーを提供します。システムには、ビジネス・プロセスの統合のためのコンポーネントをカスタマイズ、作成、および管理するためのツールとテンプレートが組み込まれています。

本書では、IBM WebSphere Business Integration システムに内蔵の Java コネクターの開発について説明します。

対象読者

本書の対象読者は、コネクターの開発者です。本書では、読者の方々が Java プログラム言語に習熟していることを前提としています。また、コネクターやビジネス・オブジェクトを含む IBM WebSphere Business Integration システムの基本知識があることも前提となります。

関連文書

注: 本書の発行後に公開されたテクニカル・サポートの技術情報や速報に、本書の対象製品に関する重要な情報が記載されている場合があります。これらの情報は、WebSphere Business Integration Support Web サイト (http://www.ibm.com/software/integration/websphere/support/) にあります。関心のあるコンポーネント・エリアを選択し、「Technotes」セクションと「Flashes」セクションを参照してください。また、IBM Redbooks (http://www.redbooks.ibm.com/.>>) にもその他の有効な情報があることがあります。

資料の完全セットにより、すべての WebSphere Business Integration Adapters に共通な機能とコンポーネントについての説明が提供されます。これには、特定のコンポーネントに関する参考資料も含まれます。

注: 本書では、Java で記述されたコネクターの開発について説明します。C++ コネクターの開発については、「*コネクター開発ガイド (C++ 用)*」に記載されています。

この製品の資料は、インストールすることも以下のサイトからオンラインで直接読むこともできます。

- 一般的なアダプター情報、WebSphere Message Brokers (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、WebSphere Business Integration Message Broker) でのアダプターの使用、および WebSphere Application Server でのアダプターの使用についての詳細は、次のサイトを参照してください。

  http://www.ibm.com/websphere/integration/wbiadapters/infocenter

- WebSphere InterChange Server を搭載したアダプターを使用する場合は、次のサイトを参照してください。

  http://www.ibm.com/websphere/integration/wicserver/infocenter

- WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、WebSphere Business Integration Message Broker) についての詳細は、次のサイトを参照してください。

  http://www.ibm.com/software/integration/mqfamily/library/manualsa/

- WebSphere Application Server の詳細については、以下のサイトを参照してください。

  http://www.ibm.com/software/webserver/appserv/library.html

これらのサイトには、資料をダウンロード、インストール、および表示するための簡単な指示が掲載されています。

表記上の規則

本書では、以下のような規則を使用しています。

`Courier フォント`	コマンド名、ファイル名、入力情報、システムが画面に出力した情報など、記述されたとおりの値を示します。
イタリック *イタリック*、*イタリック* *青のアウトライン*	初出語を示します。 変数名または相互参照を示します。 オンラインで表示したときにのみ見られる青のアウトラインは、相互参照用のハイパーリンクです。アウトラインの内側をクリックすると、参照先オブジェクトにジャンプします。
{ }	構文の記述行の場合、中括弧 {} で囲まれた部分は、選択対象のオプションです。1 つのオプションのみを選択する必要があります。
[ ]	構文の記述行の場合、大括弧 [] で囲まれた部分は、オプションのパラメーターです。
...	構文の記述行の場合、省略符号 ... は直前のパラメーターが繰り返されることを示します。例えば、`option[,...]` は、複数のオプションをコンマで区切って指定できることを意味します。
< >	命名規則では、不等号括弧は名前の個々の要素を囲み、各要素を区別します。 (例: `<server_name><`*`connector_name`*`>tmp.log`)
/、¥	本書では、ディレクトリー・パスに円記号 (¥) を使用します。UNIX システムの場合には、円記号をスラッシュ (/) に置き換えてください。すべての WebSphere Business Integration Adapters 製品のパス名は、製品の使用システムへのインストール先ディレクトリーに対する相対パスになっています。
%text% および *$text*	パーセント (%) 符号で囲まれたテキストは、Windows の `text` システム変数またはユーザー変数の値を示します。UNIX 環境での同等の表記は *$text* であり、UNIX の *text* 環境変数の値を示します。

ProductDir	製品のインストール先ディレクトリーを表します。IBM WebSphere InterChange Server 環境の場合、デフォルトの製品ディレクトリーは「IBM¥WebSphereICS」です。IBM WebSphere Business Integration Adapters 環境の場合、デフォルトの製品ディレクトリーは「WebSphereAdapters」です。

マークアップの規則

いくつかの章には、次のようなマークアップで示されるテキストがあります。

WebSphere InterChange Broker

InterChange Server が統合ブローカーの場合の、IBM WebSphere Business Integration システムの機能を説明します。

WebSphere MQ Integrator Broker

WebSphere MQ Integrator Broker が統合ブローカーの場合の、IBM WebSphere Business Integration システムの機能を説明します。

本リリースの新機能

この章では、本書で取り上げている IBM WebSphere Business Integration システムの新機能について説明します。

WebSphere InterChange Server v4.2.2 および WebSphere Business Integration Adapter Framework v2.4.1 の新機能

IBM WebSphere Business Integration Adapter Framework バージョン 2.4.1 から、アダプターは Solaris 7 でサポートされなくなりました。そのため、このプラットフォーム・バージョンに関する記述は本書から削除されました。

WebSphere InterChange Server v4.2.2 および WebSphere Business Integration Adapters v2.4.0 の新機能

IBM WebSphere InterChange Server 4.2.2 リリースおよび IBM WebSphere Business Integration Adapter 2.4.0 リリースは、Java コネクター・ライブラリーにおいて、次の新機能を提供します。

- Java コネクターは、現在、サード・パーティーのVisiBroker ORB の代わりに IBM Java Object Request Broker (ORB) を使用しています。
- Java コネクターは、現在、データ・ハンドラーとの間で送受信する直列化データへのアクセスをサポートしています。前のリリースでは、コネクターは、Java ストリングとしての直列化データにアクセスできました。現在では、次の新形式のいずれかでアクセスできます。
  - 入力ストリームとして: `boToStream()` および `streamToBO()`
  - Java バイト配列として: `boToByteArray()` および `byteArrayToBo()`
  - Reader オブジェクトとして: `readerToBO()`

  さらに、現在では、すべてのデータ・ハンドラー・メソッドが、直列化データに関連付けるデータ・ハンドラーの文字エンコードおよびロケールを識別できるようになりました。詳細については、85 ページの『データ・ハンドラーの呼び出し』のこれらのメソッドの説明を参照してください。
- Java コネクター・ライブラリーは、現在では、(`CWConnectorEventStore` クラスが表す) イベント・ストアに次の機能を提供しています。
  - `setEventStoreStatus()` メソッドは、その機能がよくわかるように、`setEventStatus()` に名前変更されました。このメソッドは、イベントの状況を設定します。
  - `getBO()` メソッドは、呼び出し側メソッドに整数状況値を戻すことができるようになりました。デフォルトで実装される `getBO()` は、内部状況値を提供しない 形式を引き続き使用します。詳細については、`CWConnectorEventStore` クラスの `getBO()` メソッドの説明を参照してください。
- Java Connector Development Kit (JCDK) は、より一貫した方法で Java コネクターの始動スクリプトを作成するようになりました。また、この始動スクリプトを

作成するためのテンプレート (Windows システムと UNIX ベースのシステムの両方に対応) も提供します。詳細については、 246 ページの『新規コネクターの始動』を参照してください。

さらに、Adapter Development Kit (ADK) では、製品ディレクトリーの DevelopmentKits¥Twineball_sample サブディレクトリーにアダプターのサンプルが組み込まれました。このアダプターのサンプルには Java コネクターが含まれます。

WebSphere InterChange Server v4.2.1 および WebSphere Business Integration Adapters v2.3.0 の新機能

IBM WebSphere InterChange Server 4.2.1 リリースおよび IBM WebSphere Business Integration Adapter 2.3.0 リリースは、Java コネクター・ライブラリーにおいて、次の新機能を提供します。

- コネクターは、データ・ハンドラーの呼び出し時に、そのハンドラーに追加の構成を提供できるようになりました。この追加情報を指定する *config* 引き数は、次のメソッドによってサポートされます。
  - boToString()
  - stringToBo()

  詳細については、 397 ページの『第 22 章 CWConnectorUtil クラス』のこれらのメソッドの説明を参照してください。
- Java コネクター・ライブラリーでは、CWConnectorBusObj クラス内の新しい形式の getAppText() メソッドを通じて、アプリケーション固有の情報内の名前と値のペアに個々にアクセスできるようになりました。

  詳細については、 293 ページの『第 13 章 CWConnectorBusObj クラス』のこのメソッドの説明を参照してください。
- (保証付きイベント・デリバリーを提供する) 重複イベント回避をサポートするために、Java コネクター・ライブラリーは、CWConnectorBusObj クラス内で setDEEId() メソッドを提供します。このメソッドを使用すると、コネクターはビジネス・オブジェクトの ObjectEventId 属性をイベント ID に設定することができます。詳細については、 156 ページの『非 JMS イベント・ストアを使用したコネクター用の保証付きイベント・デリバリー』、および 293 ページの『第 13 章 CWConnectorBusObj クラス』の setDEEId() メソッドの説明を参照してください。
- Java コネクター・ライブラリーでは、以下の機能によってイベント・ストア・ファクトリーからのイベント・ストア・オブジェクトのインスタンス生成をモジュール化できるようになりました。
  - getEventStore() メソッド (CWConnectorAgent クラス内) がイベント・ストア・ファクトリーからイベント・ストア・オブジェクトのインスタンスを作成します。CWConnectorAgent クラスはこのメソッドをデフォルトで実装しています。ただし、このメソッドをオーバーライドして振る舞いをカスタマイズすることもできます。デフォルトで実装されている pollForEvents() メソッドは、次にこの getEventStore() メソッドを呼び出して、イベント・ストア・オブジェクトを取得します。

– EventStoreFactory コネクター構成プロパティーには、イベント・ストアのイベント・ストア・ファクトリー・クラスの名前を指定できます。getEventStore() メソッド (CWConnectorAgent クラス内) は、使用するイベント・ストア・ファクトリー・クラスの名前を EventStoreFactory プロパティーから取得します。

詳細については、 205 ページの『CWConnectorEventStoreFactory インターフェース』を参照してください。

- Java コネクター・ライブラリーでは、getTerminate() メソッドと setTerminate() メソッド (CWConnectorEventStore クラス内) が提供されます。これにより、pollForEvents() メソッドでアプリケーションのタイムアウト (APPRESPONSETIMEOUT) 条件をより適切に処理できるようにします。
- Java コネクター・ライブラリーでは、Exists 動詞と RetrieveByContent 動詞に対して動詞定数が提供されるようになりました。VERB_EXISTS 動詞定数と VERB_RETRIEVEBYCONTENT 動詞定数は CWConnectorConstant クラスで定義されます。
- gotApplEvents() メソッドの戻りコードに対する変更を補足するため、この資料ではこれらのさまざまな結果状況値に応答する方法について詳しく記載しています。さらに、pollForEvents() メソッドが拡張されて上記のような応答を受け入れるようになりました。詳細については、 219 ページの『ビジネス・オブジェクトの送信』を参照してください。
- Java コネクター・ライブラリーでは、カスタム・ビジネス・オブジェクト・ハンドラー・クラスを使用したカスタム・ビジネス・オブジェクト・ハンドラーの作成がサポートされるようになりました。このクラスは CWCustomBOHandler インターフェースを実装します。特定の動詞について別の処理が必要となるビジネス・オブジェクトをコネクターがサポートする場合は、カスタム・ビジネス・オブジェクト・ハンドラーを作成してビジネス・オブジェクトのその動詞を処理できます。詳細については、 200 ページの『カスタム・ビジネス・オブジェクト・ハンドラーの作成』を参照してください。

WebSphere Business Integration Adapters 2.2.0 の新機能

IBM WebSphere Business Integration Adapter 2.2.0 リリースでは、Java コネクター・ライブラリーに以下の新機能が提供されています。

- 「CrossWorlds」 という名称は、システム全体の説明にもコンポーネントやツールの名前の修飾にも使用されなくなりました。コンポーネントやツールについて、それ以外の変更はほとんどありません。例えば、「CrossWorlds System Manager」は現在「System Manager」に、「CrossWorlds InterChange Server」は「WebSphere InterChange Server」にそれぞれ名称が変更されています。
- Java コネクター・ライブラリーでは、階層コネクター構成プロパティーへのアクセスが可能になり、次の拡張機能が提供されています。
  – CWProperty クラスは、階層コネクター・プロパティー内のストリング値と子プロパティーを取得できるメソッドを提供します。詳細については、 449 ページの『第 25 章 CWProperty クラス』を参照してください。
  – CWConnectorUtil クラスは、トップレベルの階層コネクター・プロパティーの取得を可能にする次の 2 つの新規メソッドを提供します。

- トップレベルの階層コネクター・プロパティーをすべて取得する場合: `getAllConfigProperties()`
- 指定されたトップレベルの階層コネクター・プロパティーを取得する場合: `getHierarchicalConfigProp()`

詳細については、 82 ページの『階層コネクター構成プロパティーの取得』を参照してください。

注: Java コネクター・ライブラリーでは、従来の単一値のシンプル・コネクター・プロパティー値も `getConfigProp()` メソッドによってサポートしています。

- Java コネクター・ライブラリーが、イベントの送達を保証するために重複イベントの除去をサポートするようになりました。重複イベントの除去は、JMS キューとして実装されていない イベント・ストアを備えた JMS 対応アダプターによって最も多く使用されます。この機能を使用可能にするには、`DuplicateEventElimination` コネクター・プロパティーを使用してください。詳細については、 156 ページの『非 JMS イベント・ストアを使用したコネクター用の保証付きイベント・デリバリー』を参照してください。
- 現在、Java コネクター・ライブラリーでは、以下の API メソッドが提供されています。
  - getSupportedVerbs() メソッド (`CWConnectorBusObj` クラスの) は、ビジネス・オブジェクトでサポートされる動詞のリストを提供します。
  - setLocale() メソッド (`CWConnectorBusObj` クラス) を使用することにより、ビジネス・オブジェクトに関連したロケールを設定することができます。この新規メソッドは、同じクラスですでに定義されている getLocale() メソッドを補完するものです。
  - cleanupResources() メソッド (`CWConnectorEventStore` クラスの) は、イベント・ストアが使用したリソースの解放を可能にします。
- 241 ページの『第 8 章 ビジネス・インテグレーション・システムへのコネクターの追加』では、WebSphere Business Integration システムに Java コネクターを追加する方法に関して、以下をはじめとするさらに多くの情報を提供するようになりました。
  - コネクター用の初期構成ファイルを作成する方法
  - サンプル始動ファイルから Java コネクター用の始動スクリプトを作成する方法
  - システム変数を設定するための新しい `CWConnEnv.bat` ファイル (Windows) または `CWConnEnv.sh` ファイル (UNIX) の使い方
- 41 ページの『第 2 章 コネクターの設計』では、コネクターを国際化対応させる方法について、より多くの情報が記載されるようになりました。
- 状況戻りコードの処理を改善するために、Java コネクター・ライブラリーのいくつかのメソッドが変更されました。
  - `pollForEvents()` メソッドのデフォルト実装により、以下のアクションが行なわれるようになりました。

- gotApplEvent() メソッドへの呼び出しによって戻される、CONNECTOR_NOT_ACTIVE 状況戻りコードと NO_SUBSCRIPTION_FOUND 状況戻りコードを処理します。詳細については、219 ページの『ビジネス・オブジェクトの送信』を参照してください。
- イベント・ストアへのアクセスが失敗したときに、APPRESPONSETIMEOUT の結果状況を戻します。イベント・ストアへのアクセスの失敗は、以下のイベント・ストア・メソッドで起こる可能性があります。

イベント・ストア・メソッド	発生する例外
fetchEvents()	StatusChangeFailedException
archiveEvent()	ArchiveFailedException
deleteEvent()	DeleteFailedException
updateEventStatus()	StatusChangeFailedException

  - agentInit() メソッドは、例外をスローしたときに例外詳細オブジェクトの状況値が設定されていない場合には、FAIL という結果状況を戻すようになりました。例外詳細オブジェクトで状況値が設定されている 場合、agentInit() はその状況値を戻します。
  - doVerbFor() メソッドは、ConnectionFailureException を例外をスローしたときに例外詳細オブジェクトの状況値が設定されていない場合には、APPRESPONSETIMEOUT という結果状況を戻すようになりました。例外詳細オブジェクトで状況値が設定されている 場合、doVerbFor() はその状況値を戻します。

WebSphere Business Integration Adapters 2.1 の新機能

IBM WebSphere Business Integration Adapter 2.1 リリースでは、Java コネクター・ライブラリーで以下の新機能が提供されています。

- Java コネクター・ライブラリーにより、以下の CWConnectorBusObj クラスの新規メソッドで LongText である属性値にアクセスできます。
  - getLongTextValue() により、LongText 属性値を取得する
  - setLongTextValue() により、LongText 属性値を設定する
- Java コネクター・ライブラリーが、CWConnectorAgent クラスの executeCollaboration() メソッドによるイベントの同期送信をサポートするようになりました。このメソッドは、InterChange Server が統合ブローカーである場合にのみ 使用できます。

WebSphere Business Integration Adapters 2.0.1 の新機能

IBM WebSphere Business Integration Adapter 2.0.1 リリースでは、国際化バージョンの Java コネクター・ライブラリーが提供されています。この国際化対応のコネクター・ライブラリーを使用することにより、数多くの異なるロケール (ロケールには、国/地域別情報や文字コード・セットが含まれます) に応じたローカライズが可能なアダプターの開発が可能になります。コネクターの構造は、ロケールに適合するよう、以下のように変更されました。

- コネクター・フレームワークにロケールが関連付けられるようになりました。このロケールは、オペレーティング・システムのロケールまたは構成プロパティーのどちらからも判別されます。この情報にコネクター内部からアクセスできるように、Java コネクター・ライブラリーには CWConnectorUtil クラスの getGlobalEncoding() メソッドおよび getGlobalLocale() メソッドが用意されています。
- ビジネス・オブジェクトには、ロケールが関連付けられています。このロケールは、ビジネス・オブジェクト内のデータに関連付けられています。ただし、ビジネス・オブジェクト定義の名前またはその属性には関連付けられていません。このロケールの名前をコネクター内部から取得できるように、Java コネクター・ライブラリーには、CWConnectorBusObj クラスの getLocale() メソッドが用意されています。

詳細については、62 ページの『国際化対応コネクター』を参照してください。

WebSphere Business Integration Adapters 2.0 の新機能

IBM WebSphere Business Integration Adapter 2.0 のリリースでは、アダプターに対するサポート機能が提供されています。アダプター とは、統合ブローカーと通信し、さらにアプリケーションまたはテクノロジーと通信することによって、アプリケーション・ロジックの実行やデータ交換などのタスクを行う 1 組のソフトウェア・モジュールのことです。アダプターと統合ブローカーの概要については、3 ページの『WebSphere Business Integration システムのアダプター』を参照してください。

さらに、コネクターの開発に関する IBM WebSphere Business Integration システムの資料の構成も、このリリースで変更されました。

- IBM では、Java コネクターの開発のために新しいアプリケーション・プログラミング・インターフェース (API) を導入しました。このコネクター・ライブラリーの特長は、次のとおりです。
  - イベントをカプセル化するためのクラスと、Java コネクター内部のイベント・ストア: CWConnectorEvent、CWConnectorEventStore
  - ビジネス・オブジェクト、ビジネス・オブジェクト定義、および属性へのアクセス機能を持つ単一クラス、CWConnectorBusObj
  - Java コネクター・ライブラリーのメソッドがスローする例外の詳細情報を提供するクラス: CWException 、CWConnectorExceptionObject
  - そのほかのクラスでは、以前のクラスのラッパーにすることによって、以前の下位の Java コネクター・ライブラリーの機能を保持している。

  IBM では、Java コネクターのすべての新規開発に対して、この新しい Java コネクター・ライブラリーをお勧めします。このコネクター・ライブラリーのクラスおよびメソッドの要約については、 267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。以前の下位の Java コネクター・ライブラリーのサポートは、後方互換性について継続されます。
- 以下のガイドは、Java コネクターの開発について説明する 1 冊の資料を作成するためにまとめられました。

コネクター開発ガイド	コネクターの開発方法に関する資料は、本書の第 1 部および第 2 部に変更されました。
Connector Reference: Java Class Library	下位の Java コネクター・ライブラリーに関する参照資料は、第 4 部にあります。

新しい Java コネクター・ライブラリーに関する参照資料は、本書の第 3 部にあります。

第 1 部 はじめに

第 1 章 コネクター開発の概要

この章では、IBM WebSphere Business Integration システムのコネクターの概要について簡単に説明します。さらに、Java Connector Development Kit (JCDK) についても紹介し、コネクターの実装時に従う必要のある作成手順についても要約してあります。この章を構成するセクションは次のとおりです。

WebSphere Business Integration システムのアダプター

IBM WebSphere Business Integration システム は、以下のコンポーネントから構成されています。これらのコンポーネントにより、異機種のビジネス・アプリケーションでデータを交換することが可能になります。

- 1 組の IBM WebSphere Business Integration Adapter

  IBM WebSphere Business Integration Adapter (略称: アダプター) は、統合ブローカーとアプリケーションまたはテクノロジーとの間の通信をサポートし、アプリケーション・ロジックの実行やデータの交換などのタスクを実行するためのコンポーネントを提供します。

- 統合ブローカー

  統合ブローカー のタスクは、異種のアプリケーション間でのデータを統合します。 IBM WebSphere Business Integration システムには、表 1 に示す統合ブローカーのいずれかを内蔵できます。

表 1. WebSphere Business Integration システムの統合ブローカー

統合ブローカー	詳細情報	ドキュメンテーション・セット
WebSphere Message Brokers (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、WebSphere Business Integration Message Broker)	*WebSphere Message Brokers 使用アダプター・インプリメンテーション・ガイド*	WebSphere Business Integration Adapters
WebSphere Application Server	*アダプター実装ガイド (WebSphere Application Server)*	WebSphere Business Integration Adapters
IBM WebSphere InterChange Server (ICS)	「*WebSphere InterChange Server システム・インプリメンテーション・ガイド*」	WebSphere InterChange Server

IBM WebSphere Business Integration システムでは、統合ブローカーはアダプターを介してこれらのアプリケーションと通信します。以下のアダプター・コンポーネントが、実際に通信を提供します。

- 5 ページの『ビジネス・オブジェクト』。この役割は、アプリケーション・イベントの情報を保持することです。
- 6 ページの『コネクター』。この役割は、アプリケーション・イベントの情報を統合ブローカーに送信すること、または統合ブローカーから要求に関する情報を受信することです。

図 1 には、これらのコンポーネントによる、アプリケーションから統合ブローカーへの情報の転送方法を示します。

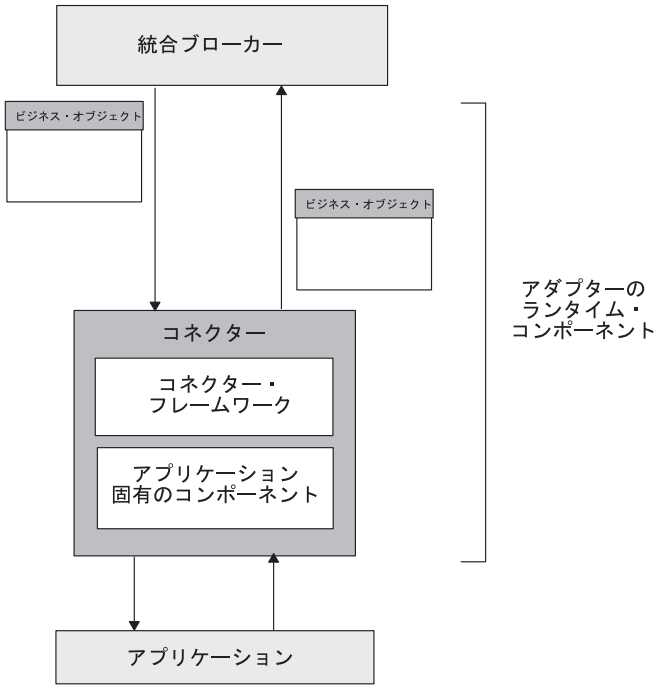

図 1. 情報転送を提供するアダプター・コンポーネント

注: アダプターには、構成および開発用のコンポーネントも組み込まれています。詳細については、31 ページの『アダプター開発用のツール』を参照してください。

図 2 は、WebSphere Business Integration システムとこのシステム内でコネクターが果たす役割を示しています。

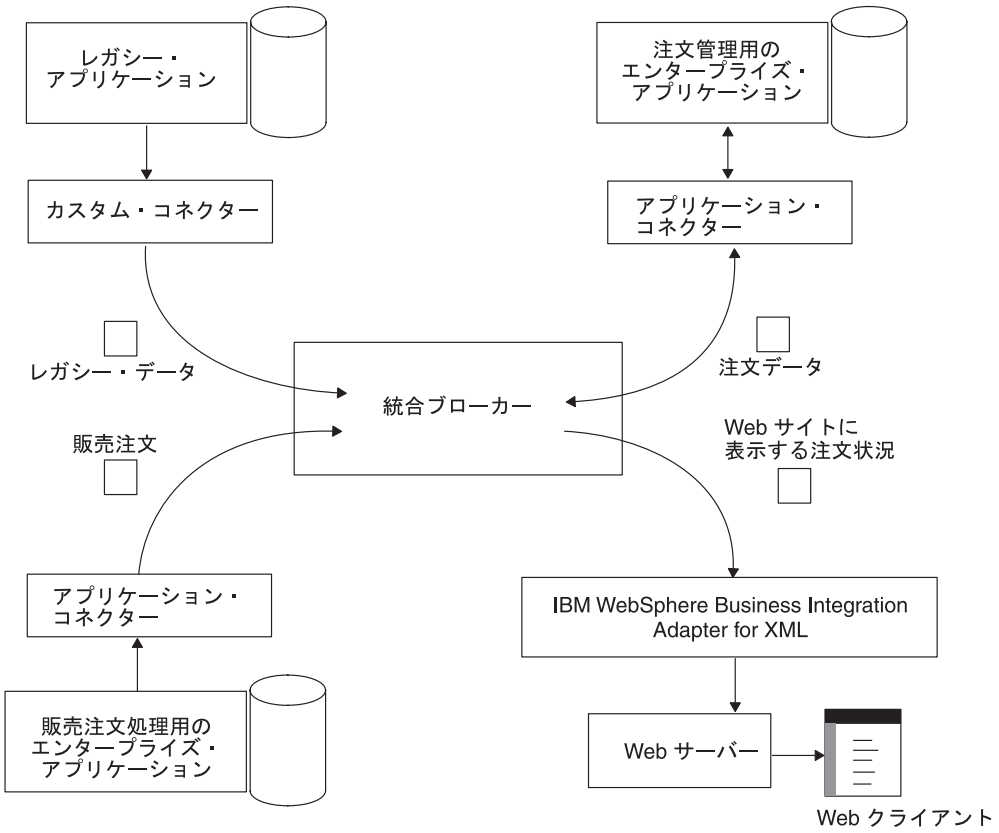

図 2. WebSphere Business Integration システム

ビジネス・オブジェクト

表 2 に示すように、ビジネス・オブジェクトとは、リポジトリー定義とランタイム・オブジェクトから構成される 2 パーツ構成のエンティティーです。

表 2. ビジネス・オブジェクトのパーツ

リポジトリー・エンティティー	ランタイム・オブジェクト
ビジネス・オブジェクト定義	ビジネス・オブジェクト・インスタンス (多くの場合「ビジネス・オブジェクト」と呼ばれる)

ビジネス・オブジェクト定義

ビジネス・オブジェクト定義 とは、収集の単位として扱うことのできる属性のグループを表します。例えば、ビジネス・オブジェクト定義では、アプリケーションのエンティティーや、作成、取得、更新、削除などのエンティティー上で実行できる操作を表現できます。ビジネス・オブジェクト定義では、Web ブラウザーから送信された商取引の書式のデータ内容など、上記以外のプログラム化されたエンティティーを表すこともできます。ビジネス・オブジェクト定義には、前述の収集単位の中の各データに対応する *属性* が記述されています。

注: ビジネス・オブジェクト定義の構造の詳細については、122 ページの『ビジネス・オブジェクトの処理』を参照してください。

ビジネス・オブジェクトを開発する場合は、ビジネス・オブジェクト定義を作成します。ビジネス・オブジェクト定義の作成には Business Object Designer ツールを使用できます。このツールは、使いやすいグラフィカル・ユーザー・インターフェース (GUI) を備えており、これによってビジネス・オブジェクトの属性を定義できます。このツールには、ビジネス・オブジェクト定義をリポジトリーまたは外部の XML ファイルに保存する機能がサポートされています。

Business Object Designer では、次の 2 通りの方法でビジネス・オブジェクト定義を作成できます。

- Business Object Designer のダイアログを使用して、ビジネス・オブジェクト定義の属性やそのほかの情報を手動で定義する。
- Object Discovery Agent (ODA) を使用する。これを使用すると、以下の動作により、ビジネス・オブジェクト定義が自動的に生成されます。
  - アプリケーション内部の指定エンティティーの検査
  - これらのエンティティーのうち、ビジネス・オブジェクト属性に対応する要素の「取得」

注: Business Object Designer を使用し、これらのいずれかの方法でビジネス・オブジェクト定義を作成する方法については、「*ビジネス・オブジェクト開発ガイド*」を参照してください。

ビジネス・オブジェクト・インスタンス

ビジネス・オブジェクト定義は一まとまりのデータを表していますが、ビジネス・オブジェクト・インスタンス (多くの場合、略称「ビジネス・オブジェクト」で呼ばれる) は、実際のデータを含むランタイムのエンティティーです。例えば、使用しているアプリケーションで顧客エンティティーを表示する場合は、ほかのアプリケーションに送ることが必要な情報を顧客エンティティー内部に定義する Customer ビジネス・オブジェクト定義を作成します。実行時には、ビジネス・オブジェクト定義のインスタンスである Customer ビジネス・オブジェクトに、特定の顧客向けの情報が格納されています。

コネクター

コネクター の役割は、アプリケーション・イベントの情報を統合ブローカーに送信すること、または統合ブローカーから要求を受信することです。

WebSphere InterChange Server

InterChange Server が統合ブローカーの場合、コネクターは、WebSphere Business Integration コラボレーションとエンタープライズ・アプリケーションまたは外部アプリケーションとを接続する 1 組のソフトウェア・モジュールおよびデータ・マップになります。コラボレーション は、複数のアプリケーションを内蔵できるビジネス・プロセスのことを表します。コネクターは、1 つ以上のコラボレーションの仲介役の機能を果たします。エンタープライズ・アプリケーション API やそれ以外の任意のプログラム・ロジックを使用して、ビジネス・プロセスをサポートします。

コネクターによって送信または受信される情報は、ビジネス・オブジェクトの書式で作成されています。したがって、各コネクターは 1 つ以上のビジネス・オブジェクト定義をサポートしています。これらのビジネス・オブジェクト定義は、アプリケーション・データ・モデルやそのほかの種類の外部エンティティーに対応することを目的として作成されています。ビジネス・オブジェクトは、ビジネス・オブジェクト自体が表しているデータ・エンティティーを忠実に反映しています。ビジネス・オブジェクトとエンティティーは、その構造と内容が一致しています。

WebSphere InterChange Server

InterChange Server が統合ブローカーの場合、ビジネス・インテグレーション・システムでは、2 種類のビジネス・オブジェクトが使用されます。コネクターが処理するビジネス・オブジェクトのことは、*アプリケーション固有のビジネス・オブジェクト* と呼びます。コラボレーションが処理するビジネス・オブジェクトのことは、*汎用ビジネス・オブジェクト* と呼びます。詳細については、13 ページの『マッピング・サービス』を参照してください。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Intregration Message Broker) または WebSphere Application Server が統合ブローカーの場合、ビジネス・インテグレーション・システムが使用するビジネス・オブジェクトは、コネクターが処理するビジネス・オブジェクト 1 種類です。このビジネス・オブジェクトはアプリケーション固有のビジネス・オブジェクトですが、この場合に使用されるビジネス・オブジェクトはこの 1 種類のみなので、「アプリケーション固有の」という修飾語は、多くの場合省略されます。

コネクターは、表 3 に示すように、サポートしているビジネス・オブジェクト定義の情報を使用して、その主な役割を果たします。

表 3. コネクターのさまざまな役割に対するビジネス・オブジェクトの動作

コネクターの役割	ビジネス・オブジェクトの動作
24 ページの『イベント通知』	アプリケーションのエンティティーに影響を与えるイベントが発生した場合 (例えば、アプリケーションのユーザーがアプリケーション・データの作成、更新、または削除を実行した場合)、コネクターの動作は以下のとおりです。 • コネクターのビジネス・オブジェクト定義に記述されている情報に基づいてビジネス・オブジェクトを作成する。 • このビジネス・オブジェクトに、アプリケーション・エンティティーからのデータを格納する。 • このビジネス・オブジェクトをイベントとして統合ブローカーに送信する。

表 3. コネクターのさまざまな役割に対するビジネス・オブジェクトの動作 (続き)

コネクターの役割	ビジネス・オブジェクトの動作
27 ページの『要求処理』	統合ブローカーが、コネクターのアプリケーションへの変更を要求した場合、またはコネクターのアプリケーションからの情報を必要としている場合、コネクターは以下のように反応します。 • 統合ブローカーからビジネス・オブジェクトを受信する。 • ビジネス・オブジェクトとコネクターのビジネス・オブジェクト定義に記述されている情報を使用して、操作を実行する適切なアプリケーションのコマンドを生成する。 • 統合ブローカーに対して、適切な応答情報を返信する。

注: 各コネクターには、要求処理を必ず 実装してください。イベント通知の実装はオプションです (一部のマイナー・コード作成時には必須となります)。

コネクター・コンポーネント

コネクターとは、WebSphere Business Integration システムにおけるアプリケーションに相当し、アプリケーションをサポートしてタスクを実行します。例えば、コネクターは、アプリケーションに対してイベントのポーリングを実行し、イベントを表すビジネス・オブジェクトを統合ブローカーに送信します。コネクターは、アプリケーション・データの取得や変更など、統合ブローカーをサポートするタスクも実行します。

図 3 には、Java コネクターのコンポーネントを示します。Java コネクター・ライブラリーは、コネクター・フレームワークが提供する汎用サービスの中に組み込まれています。

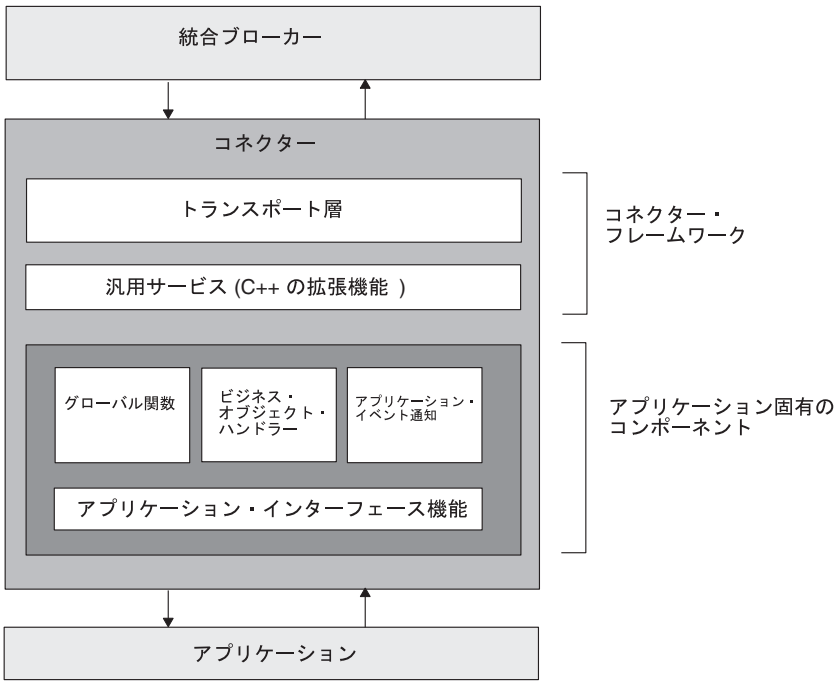

図 3. Java コネクターのコンポーネント

図 3 に示すように、コネクターには以下のコンポーネントがあります。

- 『コネクター・フレームワーク』 — 統合ブローカーと通信するために、WebSphere Business Integration Adapters 製品の一部として提供されています。
- 22 ページの『アプリケーション固有のコンポーネント』 — 基本的な初期設定やセットアップの方法、ビジネス・オブジェクトの処理、イベント通知など、アプリケーション固有のコネクター・タスクの処理を指定するために開発者側が記述するコードについて説明します。

コネクター・フレームワーク

コネクター・フレームワークは、コネクターと統合ブローカーとのやり取りを管理します。IBM では、コネクターの開発を容易にするためにこのコンポーネントを提供しています。コネクター・フレームワークは Java で記述されていますが、C++ で作成されたアプリケーション固有のコンポーネントを開発できるように、C++ の拡張機能も内蔵しています。

その他の統合ブローカー

統合ブローカーとして WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用する IBM WebSphere Business Integration システムの場合、コネクター・フレームワークは、非配布コンポーネント、つまり、アダプター・マシンに常駐しているコンポーネントです。図 4 に、WebSphere Message Broker または WebSphere Application Server を使用した上位のコネクター・アーキテクチャーを示します。統合ブローカーとして InterChange Server を使用したコネクター・アーキテクチャーの詳細については、11 ページの『コネクター・コントローラー』を参照してください。

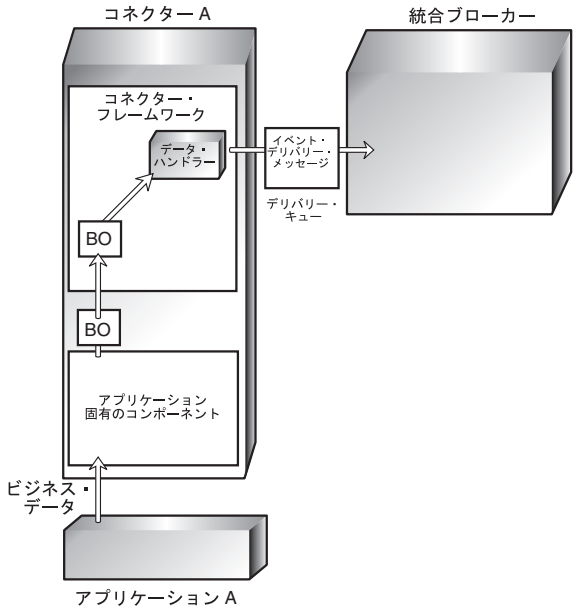

図 4. WebSphere Message Broker を使用した上位のコネクター・アーキテクチャー

コネクター・フレームワークによって提供されるサービスは、表 4 にまとめられています。

表 4. コネクター・フレームワークのサービス

コンポーネント	サービス
11 ページの『コネクター・コントローラー』 (InterChange Server のみ)	• アプリケーション固有のビジネス・オブジェクトと汎用のビジネス・オブジェクトとの間のマッピング機能を提供し、InterChange Server で動作しているコネクターとコラボレーションとの間のビジネス・オブジェクト変換を管理します。 • コネクターの状況をモニターするなど、そのほかの管理サービスを提供します。

表 4. コネクター・フレームワークのサービス (続き)

コンポーネント	サービス
16 ページの『トランスポート層』	• コネクターと統合ブローカーとの間のビジネス・オブジェクトの交換を処理します。 • コネクター・コントローラーとクライアントのコネクター・フレームワーク間での始動メッセージや管理メッセージの交換を管理します。 • サブスクライブ済みビジネス・オブジェクトのリストを保持します。
21 ページの『Java コネクター・ライブラリー』の Java コネクター・ライブラリー	• アプリケーション固有のコンポーネントに対する汎用サービスが、Java クラスおよびメソッドの書式で提供されています。

コネクター・コントローラー

InterChange Server を統合ブローカーとして使用する IBM WebSphere Business Integration システムでは、コネクター・フレームワークを配布して、InterChange Server が提供するサービスを利用します。この分散コネクター・フレームワークは、以下のコンポーネントを内蔵しています。

- クライアントのコネクター・フレームワーク は、クライアント・マシン上でコネクター・プロセスの一部として動作します。クライアントのコネクター・フレームワークには、トランスポート層、および Java コネクター・ライブラリーがあります。これらのコンポーネントの詳細については、10 ページの表 4を参照してください。
- コネクター・コントローラー は、サーバー・マシン上の InterChange Server の内部で動作します。

図 5 には、InterChange Server システム内部にあるコネクターの基本コンポーネントを示します。InterChange Server、コラボレーション、およびコネクター・コントローラーは、1 つのプロセスとして動作し、各コネクターは別個のプロセスとして動作します。

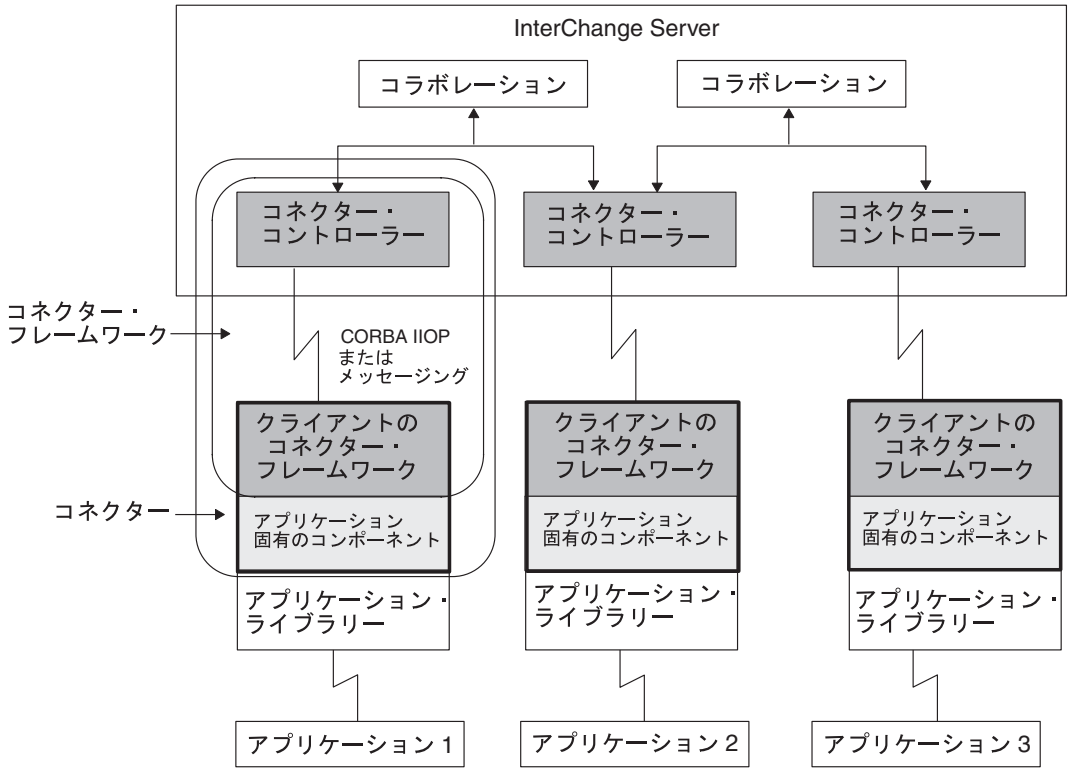

図 5. WebSphere InterChange Server を使用した上位のコネクター・アーキテクチャー

コネクター・コントローラーは、コネクター・フレームワークとコラボレーションとの通信を管理します。コネクター・コンポーネントが交換する主な種類の情報は、ビジネス・オブジェクトです。それ以外の種類のコネクター通信には、始動メッセージや管理メッセージなどがあります。

注: コネクター・コントローラーは、InterChange Server リポジトリーに定義されているコネクターごとに、InterChange Server によってインスタンス化されます。コネクター・コントローラーのコンポーネントは InterChange Server に組み込まれているので、コネクター・コントローラーのコードを記述する必要はありません。

コネクター・コントローラーは、クライアントのコネクター・フレームワークが備えている機能以外に、表 5 にまとめられているサービスを提供します。

表 5. コネクター・コントローラーのサービス

コネクター・コントローラー・サービス	説明
13 ページの『マッピング・サービス』	コネクター・コントローラーは、各ビジネス・オブジェクトに関連付けられているマップを呼び出して、汎用のビジネス・オブジェクトとアプリケーション固有のビジネス・オブジェクトとの間でデータを変換します。

表 5. コネクター・コントローラーのサービス (続き)

コネクター・コントローラー・サービス	説明
14 ページの『ビジネス・オブジェクトのサブスクリプションとパブリッシュ』	コネクター・コントローラーは、ビジネス・オブジェクト定義へのコラボレーションのサブスクリプションを管理します。ビジネス・オブジェクトのサブスクリプション状況に関するコネクターの照会についても管理します。
サービス呼び出し要求 (詳細については、28 ページの『InterChange Server による要求の開始』を参照してください。)	コネクター・コントローラーは、コラボレーションのサービス呼び出し要求をコネクターに送信します。コネクターからの戻り状況メッセージやビジネス・オブジェクトを受信して、これらを InterChange Server に転送する機能もあります。
コンポーネント間の通信 (詳細については、17 ページの『InterChange Server によるトランスポート機構』を参照してください。)	コネクター・コントローラーには、トランスポート・ドライバーが組み込まれています。これは、コネクター・コントローラーとクライアントのコネクター・フレームワークとの間でビジネス・オブジェクトや管理メッセージを交換する機構のうち、コネクター・コントローラー側を処理することを目的としています。コネクター・コントローラーは、それ自身とクライアントのコネクター・フレームワークとの上位同期を管理するためのリモート・エンド同期機能も備えています。ここに示すサービスにより、コネクター・コントローラーは、リモート側でインストールされているコネクターの場合でも、コネクターと通信できるようになります。

注: コネクター・コントローラーは、自身の内部エラーだけでなく、クライアントのコネクター・フレームワークからのエラーも処理します。通常、コネクター・コントローラーは例外を記録してから、追加処置が必要かどうかを判別します。コネクター・コントローラーは、クライアントのコネクター・フレームワークによって状況メッセージが返されると、そのメッセージをコラボレーションに転送します。

マッピング・サービス: クライアントのコネクター・フレームワークは、アプリケーション固有のビジネス・オブジェクトに記録されている情報をやり取りします。ただし、コラボレーションは、汎用のビジネス・オブジェクトの中に情報を生成します。アプリケーション固有のビジネス・オブジェクトは、汎用のビジネス・オブジェクトとは異なる場合があるので、InterChange Server システムでは、このシステムを介してデータを転送できるように、一方の書式からもう一方の書式へビジネス・オブジェクトを変換する必要があります。汎用のビジネス・オブジェクトとアプリケーション固有のビジネス・オブジェクトとの間でデータを変換するには、データ・マッピング を使用します。

データ・マッピングでは、ビジネス・オブジェクトの書式が、汎用の書式とアプリケーション固有の書式との間で双方向に変換されます。アプリケーション固有のビジネス・オブジェクトは、それ自身が表しているデータのエンティティーを忠実に

反映しています。ビジネス・オブジェクトとエンティティーは、その構造と内容が一致しています。これとは異なり、汎用のビジネス・オブジェクトには、通常、エンティティーのデータに関する標準的なアプリケーション相互参照を表す属性のスーパーセットが登録されています。この種のビジネス・オブジェクトは、多くのアプリケーションが特定のエンティティーに関して保持している共通の情報を合成したものです。汎用のビジネス・オブジェクトは、複数のデータ・モデルの中間点として機能します。

マッピングはコネクターによって開始され、実行時に実行されます。例えば、コネクターがアプリケーション固有のビジネス・オブジェクトを汎用のビジネス・オブジェクトに変換する場合、コネクターは、関連のマップを実行し、アプリケーション固有のビジネス・オブジェクトと汎用のビジネス・オブジェクトとの間でデータを転送してから、汎用のビジネス・オブジェクトをコラボレーションに送信します。

マッピングは、コネクター・コントローラーによって処理されます。図 6 には、InterChange Server システムにおけるコネクターと、コネクターのコンポーネントを示します。

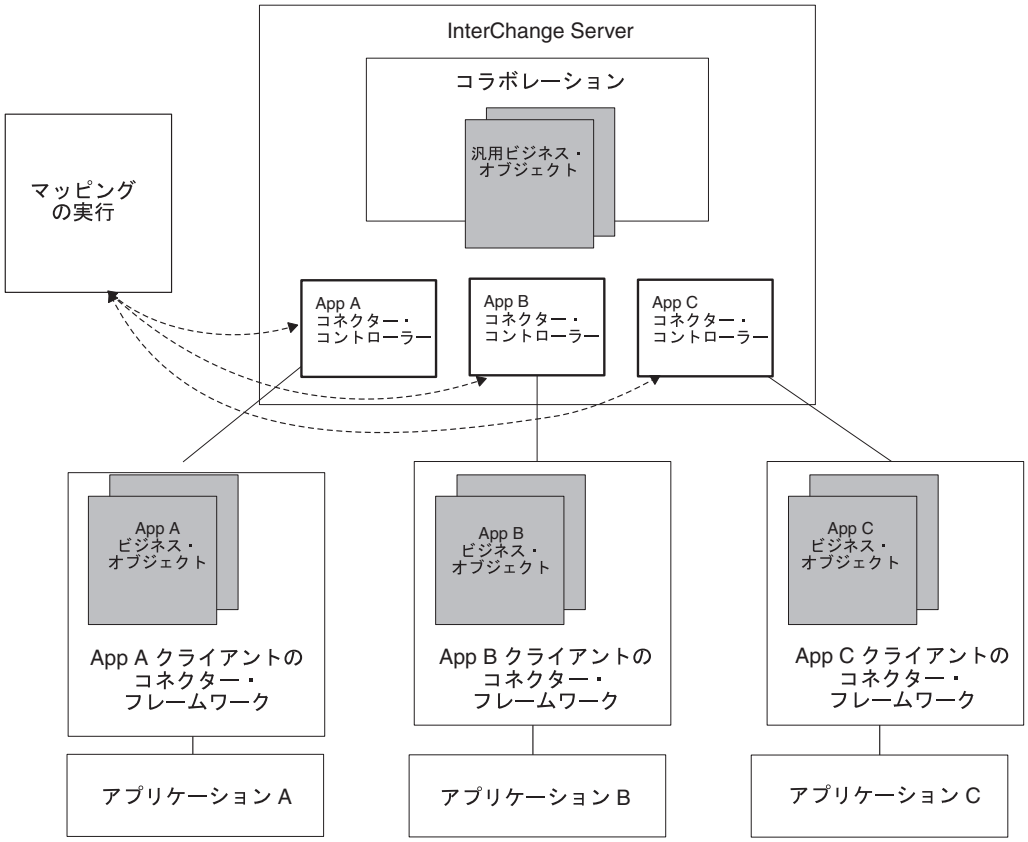

図 6. InterChange Server System でのマッピング

データ・マッピングの詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セットの「*マップ開発ガイド*」を参照してください。

ビジネス・オブジェクトのサブスクリプションとパブリッシュ: サブスクリプション処理は、サブスクリプション・リスト を介して管理します。このリストは、コラ

ボレーションのサブスクライブ先となるビジネス・オブジェクトのリストです。コネクター・フレームワークとコネクター・コントローラーは、どちらも次のようにしてサブスクリプション・リストを保守します。

- コネクター・コントローラーは、コラボレーションのサブスクライブ先となったビジネス・オブジェクトのリストを保守します。

  コラボレーションは、始動すると、目的のビジネス・オブジェクトをコネクター・コントローラーに通知することによって、そのビジネス・オブジェクトにサブスクライブします。コネクター・コントローラーは、この情報をサブスクリプション・リストに格納します。このリストには、サブスクライブ側のコラボレーションと、ビジネス・オブジェクト定義の名前および動詞が登録されています。

  コネクター・コントローラーは、クライアントのコネクター・フレームワークからビジネス・オブジェクトを受信すると、そのサブスクリプション・リストを検査して、どのコラボレーションがこの種類のビジネス・オブジェクトにサブスクライブしたのかを確認します。コネクター・コントローラーは、次に、ビジネス・オブジェクトをサブスクライブ側のコラボレーションに転送します。
- コネクター・フレームワークも、コラボレーションのサブスクライブ先となったビジネス・オブジェクトのリストを保守します。ただし、このサブスクリプション・リストは、コネクター・コントローラーのサブスクリプション・リストを統合したものです。

  初期設定時に、コネクターはそのビジネス・オブジェクト定義と構成プロパティーを InterChange Server リポジトリーからダウンロードします。コネクター・コントローラーからもサブスクリプション・リストを要求します。コネクター・コントローラーがクライアントのコネクター・フレームワークに送信するサブスクリプション・リストには、サブスクライブ済みのビジネス・オブジェクトに関するビジネス・オブジェクト定義の名前と動詞のみが登録されています。コネクター・フレームワークは、このサブスクリプション・リストをローカル・マシンに格納します。新しいコラボレーションが始動してビジネス・オブジェクトにサブスクライブすると、コネクター・コントローラーは、そのたびにコネクター・フレームワークに通知して、ローカルのサブスクリプション・リストが最新の状態に保たれるようにします。

  コネクター・フレームワークは、クライアントのコネクター・フレームワークの初期化の一環として、サブスクリプション・マネージャーのインスタンスを生成します。サブスクリプション・マネージャー は、コネクター・コントローラーから到着したすべてのサブスクリプション・メッセージおよびサブスクリプション解除メッセージを追跡して、アクティブなビジネス・オブジェクト・サブスクリプションのリストを保守します。サブスクリプション・マネージャーを利用すると、アプリケーション固有のコネクター・コンポーネントは、コネクター・フレームワークに照会して、特定の種類のビジネス・オブジェクトを処理の対象とするコラボレーションの有無を調べることができます。

図 7 には、サブスクリプション処理に関するコネクターのアーキテクチャーを示します。

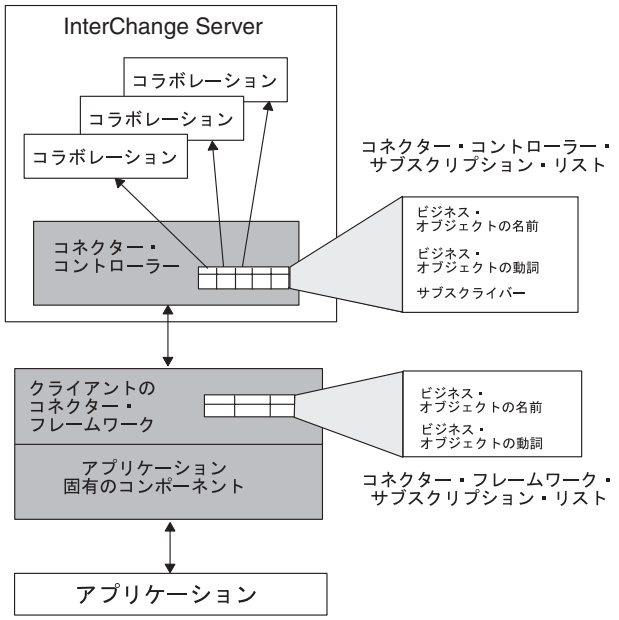

図 7. サブスクリプション処理

サブスクリプションの詳細については、27 ページの『要求処理』を参照してください。

トランスポート層

コネクター・フレームワークのトランスポート層では、コネクターと統合ブローカーとの間の情報交換を処理します。コネクター・フレームワークのトランスポート層では、以下のサービスが提供されます。

- 統合ブローカーからビジネス・オブジェクトを受信することと、ビジネス・オブジェクトを統合ブローカーに送信すること。

メッセージ・サービス	説明
27 ページの『要求処理』	統合ブローカーからビジネス・オブジェクトを受信し、そのオブジェクトをコネクターのアプリケーション固有のコンポーネントに送信します。
24 ページの『イベント通知』	コネクターのアプリケーション固有のコンポーネントからビジネス・オブジェクトを受信して、そのオブジェクトを統合ブローカーに送信します。

- コネクターと統合ブローカー間での始動メッセージや管理メッセージの交換を管理します。
- サブスクライブ先のビジネス・オブジェクトのリストを保持します。

トランスポート層のトランスポート機構は、使用しているビジネス・インテグレーション・システムの統合ブローカーによって異なります。

- 17 ページの『InterChange Server によるトランスポート機構』
- 20 ページの『その他の統合ブローカーのトランスポート機構』

InterChange Server によるトランスポート機構: 統合ブローカーが InterChange Server (ICS) の場合、トランスポート層では、ICS にあるコネクター・コントローラーとクライアントのコネクター・フレームワークとの間での情報交換が処理されます。

注: 詳細については、11 ページの『コネクター・コントローラー』を参照してください。

図 8 に示すように、InterChange Server と通信するコネクターのトランスポート層には、2 つのトランスポート・ドライバーがあります。1 つは共通オブジェクト・リクエスト・ブローカー (CORBA) 用ドライバーであり、もう 1 つは任意のメッセージ指向ミドルウェア (MOM) 用ドライバーです。

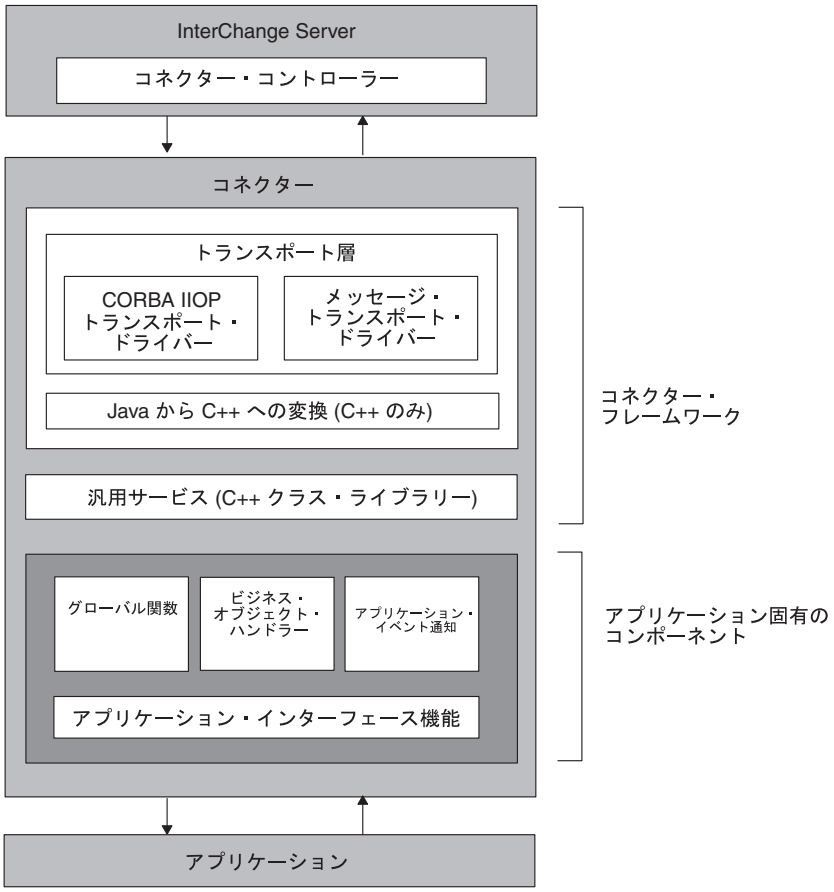

図 8. InterChange Server との通信に対応したコネクター・アーキテクチャー

表 6 には、トランスポート層によって実行されるタスクと、トランスポート層が使用できるトランスポート機構についてまとめてあります。

表 6. トランスポート層のタスク

トランスポート層タスク	トランスポート機構
コネクターの始動およびコネクター・コントローラーとクライアントのコネクター・フレームワーク間での始動メッセージの交換	CORBA

表 6. トランスポート層のタスク (続き)

トランスポート層タスク	トランスポート機構
クライアントのコネクター・フレームワークの状態に関する管理メッセージ	CORBA
コネクターへのビジネス・オブジェクトの送信。コラボレーションのサービス呼び出し要求によって開始。	CORBA
コネクターからのビジネス・オブジェクトの送信。イベント・デリバリーによって開始。	CORBA。メッセージ指向のミドルウェア・システム (以下のいずれかを含む): • WebSphere MQ • Java Messaging Service (JMS)

このトランスポート機構には、以下のタスクがあります。

- トランスポート層では、コネクターの始動時に、共通オブジェクト・リクエスト・ブローカー・アーキテクチャー (CORBA) によって、情報が InterChange Server からコネクター・プロセスのメモリーに転送されます。

  CORBA アーキテクチャーでは、オブジェクトはオブジェクト・リクエスト・ブローカー (ORB) を介して通信します。 ORB とは、コネクター・コントローラーなどのオブジェクトと、クライアントのコネクター・フレームワークなどの別のオブジェクトとを接続する、ライブラリーとサービスの組み合わせのことです。ORB により、複数のオブジェクトが、始動時に互いを取得したり、実行時に互いのメソッドを呼び出したりすることができます。

  ORB との連携により、CORBA アーキテクチャーはネーミング・サービスを提供します。このサービスにより、ORB 上のオブジェクトは、ほかのオブジェクトを名前で取得できるようになります。始動時に、クライアントのコネクター・フレームワークは、ネーミング・サービスを使用して InterChange Server に接続します。クライアントのコネクター・フレームワークは、次に、ORB を使用して、そのアプリケーション固有のコネクター構成プロパティーと、サポートされているビジネス・オブジェクト定義のリストをリポジトリーから要求します。詳細については、 72 ページの『コネクターの始動』を参照してください。

  クライアントのコネクター・フレームワークとコネクター・コントローラーがアクティブになって接続されると、クライアントのコネクター・フレームワークは、ビジネス・オブジェクトのサブスクリプション・リストを要求します。この時点で、コネクターの初期設定は完了し、コネクターはイベントのポーリングを開始します。
- コネクターの状態に関する管理メッセージの場合は、トランスポート層が CORBA を使用することにより、コネクター・コントローラーの状態に関する情報がやり取りされます。

  クライアントのコネクター・フレームワークの状態の変更は、WebSphere Business Integration Toolset の System Manager から開始できます。こうした変更には、開始、停止、休止、再開の各操作や、状態の取得なども含まれます。さらに、管理メッセージにはリモート・メッセージ・ロギングを指定することもできます。

- コラボレーションのサービス呼び出し要求によって開始されたコネクターに、ビジネス・オブジェクトを送信する場合にも、トランスポート層では CORBA が使用されます。

  CORBA テクノロジーには、Internet Inter-ORB Protocol (IIOP) トランスポート・プロトコルが採用されています。CORBA IIOP は、コネクター・コントローラーとクライアントのコネクター・フレームワークが使用して対話するための軽量で高性能な同期通信機構を備えています。IIOP 通信機構は同期方式なので、コネクター・コンポーネントは、ビジネス・オブジェクトが正常に交換されたかどうかをすばやく判別することができ、必要に応じて適切な処置を講じることができます。
- イベント・デリバリーによって開始されたコネクターからビジネス・オブジェクトを送信する場合は、コネクターを構成して、CORBA システムまたはメッセージ指向ミドルウェア (MOM) システムのいずれかを使用できます。

  ビジネス・オブジェクトのサブスクリプション配信に CORBA を使用する場合は、複数のビジネス・オブジェクトを同時に配信できるので、サブスクリプション配信の効率を改善できます。CORBA を通信機構として採用すると、高帯域幅の LAN ネットワークでは、特に高い効率が得られます。

  メッセージング・システムでは、ネットワークを介して非同期のメッセージ配信を実行できるので、コネクター・コンポーネントは、メッセージを送信後、応答を待機する必要なく処理を続行できます。メッセージング・システムには、パーシスタント・メッセージング機能もあります。これにより、コネクター・コントローラーとクライアントのコネクター・フレームワークは、それぞれ個別に動作できるようになります。

  注: この場合、コネクター・コンポーネントは、始動メッセージと管理メッセージのために、引き続き CORBA を使用します。

  メッセージング通信機構では、メッセージの転送は、クライアントのコネクター・フレームワークおよびコネクター・コントローラー内部のトランスポート・ドライバーによって処理されます。メッセージ・トランスポート・ドライバーは、InterChange Server と、下位に位置するメッセージ・キューイング・ソフトウェア間でのデータ交換のために下位の機構を実装します。コネクターのコンポーネント間のメッセージは、メッセージング・ソフトウェアによって定義されたフォーマットで送信されます。

このビジネス・インテグレーション・システムは、IBM Object Request Broker (ORB) によって提供される CORBA テクノロジーを使用します。図 9 に、CORBA 通信機構を示します。

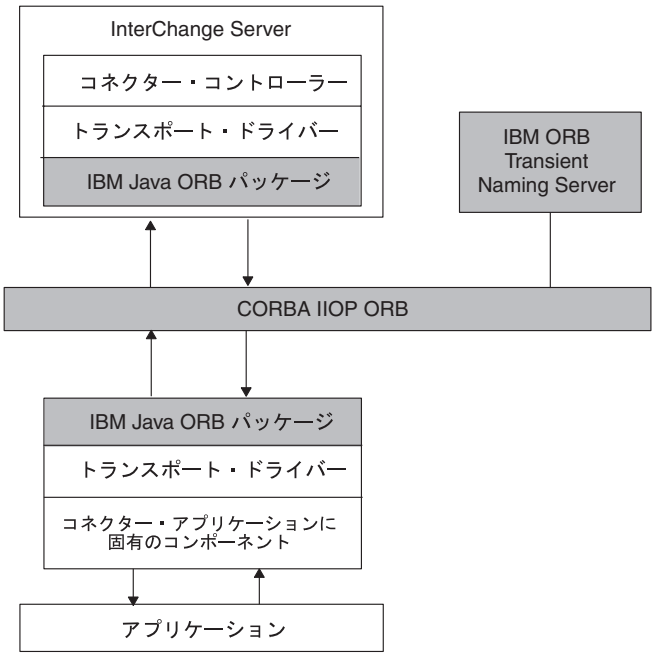

図 9. CORBA IIOP によるコネクター内部での通信

サポートされているメッセージ指向ミドルウェアは、次のとおりです。

- IBM WebSphere MQ メッセージング・スイート。このシステムでは、アクティブな各コネクターに単一方向のメッセージ・キューが必要です。WebSphere MQ では、キュー・マネージャーを使用してキューを管理します。このビジネス・インテグレーション・システムでは、すべてのシステム・コンポーネントに対して、各 InterChange サーバーに 1 つのキュー・マネージャーが存在します。
- Java Messaging Service (JMS)

注: イベント・デリバリーに関するコネクターのトランスポート機構を構成するには、`DeliveryTransport` 標準プロパティーを設定します。このプロパティーの詳細については、549 ページの『付録 A. コネクターの標準構成プロパティー』を参照してください。

その他の統合ブローカーのトランスポート機構: 統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server である場合、トランスポート層はコネクター・フレームワークと統合ブローカーの間の情報の交換を処理します。ブローカーと通信するコネクターのトランスポート層には、IBM WebSphere MQ メッセージング・スイート用のトランスポート・ドライバーが 1 つ組み込まれています。データは、アプリケーション固有のビジネス・オブジェクトを使用して、アプリケーション間で交換されます。これらのビジネス・オブジェクトは、コネクター・フレームワークと統合ブローカーとの間を WebSphere MQ のメッセージとして転送されます。統合ブローカーは、MQ キューからメッセージを除去し、キューのメッセージ・フローを介してこのメッセージを渡します。

このトランスポート機構では、WebSphere MQ メッセージを使用して、以下のタスクを実行します。

- 要求処理の開始側であるコネクターを*宛先* にしてビジネス・オブジェクトを送信する場合、トランスポート層はビジネス・オブジェクトを MQ メッセージに変換して、このメッセージを適切な WebSphere MQ キューに書き込みます。
- イベント・デリバリーの開始側であるコネクターを*送信元* としてビジネス・オブジェクトを送信する場合、トランスポート層は、適切な WebSphere MQ キューから MQ メッセージを取り出して、これをアプリケーション固有のビジネス・オブジェクトに変換します。

コネクター・フレームワークは、カスタムのデータ・ハンドラーを使用して、宛先の WebSphere MQ キューに対して適切なワイヤー・フォーマットを持つ MQ メッセージと、アプリケーション固有のビジネス・オブジェクトとの双方向の変換を実行します。

MQ メッセージとコネクターの使用についての詳細は、ご使用の統合ブローカーのインプリメンテーション・ガイドを参照してください。

Java コネクター・ライブラリー

コネクター・フレームワークには、Java コネクター・ライブラリー があります。ここには、コネクターを開発するための汎用サービスやユーティリティーが用意されています。 Java コネクター・ライブラリーに用意されている主なサービスは、以下のとおりです。

- ビジネス・オブジェクト定義ディレクトリー － コネクターによってサポートされているビジネス・オブジェクト定義へのアクセスを管理します。ビジネス・オブジェクト定義は、分散環境でのコネクターのパフォーマンスを向上するためにキャッシュに格納されています。
- ビジネス・オブジェクト・クラス － アプリケーション情報を処理するためのメソッドを提供します。このクラスでは、コネクターがオブジェクト指向の方式でアプリケーション・データを処理できます。
- サブスクリプション・マネージャー － これを利用すると、コネクターは、特定の種類のビジネス・オブジェクトを処理の対象とするコラボレーションの有無を調べることができます。
- ロギング・ユーティリティー － これを使用すると、コネクターは、メッセージをコネクターの標準出力に通知できます。出力先を設定できる機能と、ログに記録されたすべてのメッセージに対してエラー・レベルを指定できる機能があります。
- トレース・ユーティリティー － これを使用すると、コネクターはデバッグを目的とするトレース・メッセージを生成できます。

注: Java コネクター・ライブラリーおよびそのクラスの概要については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

Java コネクター・ライブラリーは `WBIA.jar` という Java .jar ファイルであり、以下のディレクトリーにあります。

ProductDir`/lib`

Java はオペレーティング・システムに依存しないため、Java コネクター・ライブラリーは WebSphere Business Integration Adapters 製品がサポートするすべてのシステムで使用可能です。

アプリケーション固有のコンポーネント

コネクターのアプリケーション固有のコンポーネントには、特定のアプリケーションに応じて調整されたコードが含まれています。これは、開発者が設計してコーディングするコネクターの一部です。アプリケーション固有のコンポーネントの構成は次のとおりです。

- コネクターの初期設定とセットアップを行うためのコネクター基底クラス
- 統合ブローカー要求によって初期設定された要求ビジネス・オブジェクトに応答するためのビジネス・オブジェクト・ハンドラー
- 必要な場合は、アプリケーション・イベントの検出と応答を行うためのイベント通知機構

コネクター・フレームワークによって提供されているサービスを利用するには、アプリケーション固有のコンポーネントのコードを作成します。コネクターのクラス・ライブラリーを使用すると、これらのサービスにアクセスできます。コネクターのコードは、アプリケーションが備えてい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) に応じて、C++ または Java のいずれかの言語で記述できます。

アプリケーション API が Java で記述されている場合は、アプリケーション固有の部分を Java で記述し、Java コネクター・ライブラリーを介してコネクター・フレームワークのサービスにアクセスします。

イベント・トリガー処理フロー

Java コネクター・ライブラリーに組み込まれている API を利用すると、ユーザー定義のアプリケーション固有コンポーネントが、統合ブローカーとビジネス・オブジェクトを介して通信できるようになります。各アプリケーションは、統合ブローカーが管理している別のアプリケーションと情報を交換できます。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コネクターは、コラボレーションを実行することによってほかのアプリケーションと通信できます。コラボレーション は、複数のアプリケーションを内蔵できるビジネス・プロセスのことを表します。コネクターは、データおよびロジックを、コネクターのアプリケーションで発生したイベントに関する情報を伝達するビジネス・オブジェクトに変換します。ビジネス・オブジェクトは、コラボレーションのビジネス・プロセスを起動して、このビジネス・プロセスのためにビジネス・オブジェクトが必要とする情報をコラボレーションに通知します。

注: 外部プロセスの場合も、呼び出しトリガー処理フローによってコラボレーションの実行を開始できます。詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*アクセス開発ガイド*」を参照してください。

WebSphere Message Brokers

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) を使用するビジネス・インテグレーション・システムの場合、コネクターは、WebSphere MQ ワークフローを経由して他のアプリケーションに情報を要求したり情報を送信したりする場合があります。MQ ワークフローは、情報を適切な経路で送信します。

アプリケーション内部でイベントが発生すると、コネクターのアプリケーション固有コンポーネントは、ビジネス・オブジェクトを作成してこのイベントを表現し、イベントを統合ブローカーに送信します。*アプリケーション・イベント* とは、ビジネス・オブジェクト定義に関連付けられているエンティティーに影響を与えるすべてのイベントのことです。イベントを統合ブローカーに送信するため、コネクターはイベント・デリバリー を開始します。このイベントには、ビジネス・オブジェクトが格納されています。したがって、コネクターが開始するフロー・トリガーのことを、*イベント・トリガー処理フロー* と呼びます (図 10 を参照)。

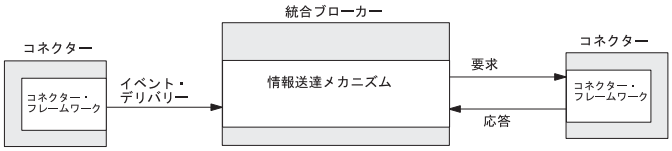

図 10. WebSphere Business Integration システムのイベント・トリガー処理フロー

図 10 には、IBM WebSphere Business Integration システム内部でのイベント・トリガー処理フローを示します。この手順は、次のとおりです。

1. コネクターは、*トリガー・イベント* を作成します。これは、コネクターがイベント・デリバリー時に統合ブローカーに送信します。

   アプリケーションのエンティティーに影響を与えるイベントが発生した場合 (例えば、アプリケーションのユーザーがアプリケーション・データの作成、更新、または削除を実行した場合)、コネクターは、ビジネス・オブジェクトを 1 つ作成します。このビジネス・オブジェクトには、アプリケーションのエンティティーからのデータや、このデータに対して実行される操作を示す動詞が登録されています。
2. コネクターのアプリケーション固有コンポーネントは、Java コネクター・ライブラリーの `gotApplEvent()` メソッドを呼び出して、トリガー・イベントをコネクター・フレームワークに送信します。このメソッド呼び出しにより、コネクターは、イベント・トリガー処理フローを開始するイベント・デリバリー を実行します。
3. コネクター・フレームワークは、トリガー・イベントからビジネス・オブジェクトへの変換を必要に応じて実行した後、このイベントを統合ブローカーに送信します。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コネクター・コントローラーはトリガー・イベントを受信し、必要に応じてアプリケーション固有のビジネス・オブジェクト・データから適切な汎用ビジネス・オブジェクトへのマッピングを実行します。コネクター・コントローラーは、次にトリガー・イベントを指定のコラボレーションに送信して、このコラボレーションを起動します。このコラボレーションは、前述のイベントが表しているビジネス・オブジェクトにサブスクライブしているものです。コラボレーションは、その受信ポートでこのビジネス・オブジェクトを受信します。

4. 統合ブローカーは、それ自身が備えている任意のロジックを使用して、イベントを適切なアプリケーションに送ります。統合ブローカーが詳細に設定されている場合は、*要求* が実行され、イベント情報が任意の宛先アプリケーションのコネクターに送られる場合があります。この場合、アプリケーションは、その要求ビジネス・オブジェクトが格納されているイベントを受け取ります。さらに、この宛先コネクターは要求の応答 を統合ブローカーに返す場合があります。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、コラボレーションが*サービス呼び出し要求* を実行して、発信側ポートに結び付けられている宛先コネクターのコネクター・コントローラーにビジネス・オブジェクトを送信する場合があります。このコネクター・コントローラーは、これによって得られた汎用ビジネス・オブジェクトに対して必要な変換を行い、適切なアプリケーション固有のビジネス・オブジェクトを生成します。そして、*サービス呼び出し応答* を実行して、コネクター・コントローラーにイベント応答を送ります。このコネクター・コントローラーによって、イベント応答が発信元コラボレーションにルーティングされます。

図 10 に示すように、コネクターには、次の 2 つのうちいずれかの役割を割り当てることができます。

- 『イベント通知』 — コネクターは、(ビジネス・オブジェクトの形式で) イベントを統合ブローカーに送信して、アプリケーションに発生した何らかの動作を統合ブローカーに通知します。
- 27 ページの『要求処理』 — コネクターは、統合ブローカーから要求ビジネス・オブジェクトを受信します。

これらのコネクターのそれぞれの役割については、以降のセクションで詳細を説明します。

イベント通知

コネクターの役割の 1 つは、アプリケーションのビジネス・エンティティーに対する変更を検出することです。アプリケーションのエンティティーに影響を与えるイ

ベントが発生した場合 (例えば、アプリケーションのユーザーがアプリケーション・データの作成、更新、または削除を実行した場合)、コネクターはイベントを統合ブローカーに送信します。このイベント には、ビジネス・オブジェクトと動詞が格納されています。この役割のことを*イベント通知* といいます。

このセクションでは、イベント通知に関する以下の情報について記載します。

- 『パブリッシュ/サブスクライブ・モデル』
- 『イベント通知機構』

パブリッシュ/サブスクライブ・モデル

コネクターでは、ビジネス・インテグレーション・システムが、*パブリッシュ/サブスクライブのモデル* を使用し、アプリケーションから統合ブローカーへ情報を移動して処理することを前提としています。

- 統合ブローカーは、アプリケーション内部のイベントを表しているビジネス・オブジェクトにサブスクライブ します。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コラボレーションは、アプリケーション内部のイベントを表しているビジネス・オブジェクトにサブスクライブ して、待機します。

- コネクターは、イベント通知機構を使用して、アプリケーション・イベントの発生をモニターします。アプリケーション・イベントが発生すると、コネクターはイベントの通知をビジネス・オブジェクトと動詞の形式でパブリッシュ します。統合ブローカーは、サブスクライブ先のビジネス・オブジェクトの形式でイベントを受信すると、このデータの関連ビジネス・ロジックを開始します。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コネクター・コントローラーは、コネクター・フレームワークからビジネス・オブジェクトを受信すると、専用のサブスクリプション・リストを検査して、この種類のビジネス・オブジェクトにサブスクライブしたコラボレーションの有無を確認します。コラボレーションが存在する場合、コネクター・コントローラーは、次に、ビジネス・オブジェクトをサブスクライブ側のコラボレーションに転送します。コラボレーションは、サブスクライブ済みイベントを受信すると、実行を開始します。

イベント通知機構

イベント通知機構 により、コネクターはアプリケーション内のエンティティーに変更が発生した時期を判定することができます。アプリケーション内部でイベントが発生すると、コネクターのアプリケーション固有のコンポーネントはこのイベントを処理し、関連のアプリケーション・データを取得して、ビジネス・オブジェクト内の統合ブローカーにデータを返します。

注: このセクションでは、イベント通知の概要について説明します。イベント通知機構のインプリメント方法については、 131 ページの『第 5 章 イベント通知』を参照してください。

以下の手順では、イベント通知機構のタスクについて、その概要を示します。

1. アプリケーションは、イベントを実行して、イベント・レコードをイベント・ストアに書き込みます。

   イベント・ストア は、アプリケーション内の永続的キャッシュで、ここにイベント・レコードが保管されているため、コネクターはそのイベント・レコードを処理することができます。*イベント・レコード* には、アプリケーション内部のイベント・ストアの変更に関する情報が記録されています。この情報には、作成または変更されたデータや、そのデータに対して実行された操作 (作成、削除、更新など) などがあります。
2. コネクターのアプリケーション固有コンポーネントは、通常、ポーリング機構を介してイベント・ストアをモニターし、受信イベントの有無を検査します。このコンポーネントは、イベントを検出すると、イベント・ストアからイベント・レコードを取得して、このレコードを動詞の付いたアプリケーション固有のビジネス・オブジェクトに変換します。
3. ビジネス・オブジェクトを統合ブローカーに送信する前に、アプリケーション固有のコンポーネントは、統合ブローカーの処理対象がビジネス・オブジェクトの受信であることを確認できます。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コネクター・フレームワークでは、サポートされているすべてのビジネス・オブジェクトが、統合ブローカーによる処理の対象になることを*必ずしも前提にはしていません*。初期化時、コネクター・フレームワークは、コネクター・コントローラーから自身のサブスクリプション・リストを要求します。アプリケーション固有のコンポーネントは、実行時に、コネクター・フレームワークに照会して、いずれかのコラボレーションが特定のビジネス・オブジェクトにサブスクライブしていることを確認できます。アプリケーション固有のコネクター・コンポーネントは、現在サブスクライブしているコラボレーションが存在する場合に限り、イベントを送信できます。アプリケーション固有のコンポーネントは、ビジネス・オブジェクトと動詞の形式で、イベントをコネクター・フレームワークに送信します。コネクター・フレームワークは、受信したイベントを、ICS 内部のコネクター・コントローラーに送信します。詳細については、13 ページの『マッピング・サービス』を参照してください。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークでは、統合ブローカーがコネクターによってサポートされるすべての ビジネス・オブジェクトに関係していると想定します。アプリケーション固有のコネクター・コンポーネントが、コネクター・フレームワークに照会してビジネス・オブジェクトを送信するかどうかを確認すると、このコンポーネントは、コネクターがサポートしているすべての ビジネス・オブジェクトの確認を受信します。

4. 統合ブローカーの処理の対象がビジネス・オブジェクトの場合、コネクターのアプリケーション固有のコンポーネントは、ビジネス・オブジェクトと動詞の形式で、イベントをコネクター・フレームワークに送信します。コネクター・フレームワークは、受信したイベントを統合ブローカーに送信します。

図 11 には、イベント通知機構のコンポーネントを示します。イベント通知では、情報の流れが、アプリケーションからコネクター、さらに統合ブローカーの順になります。

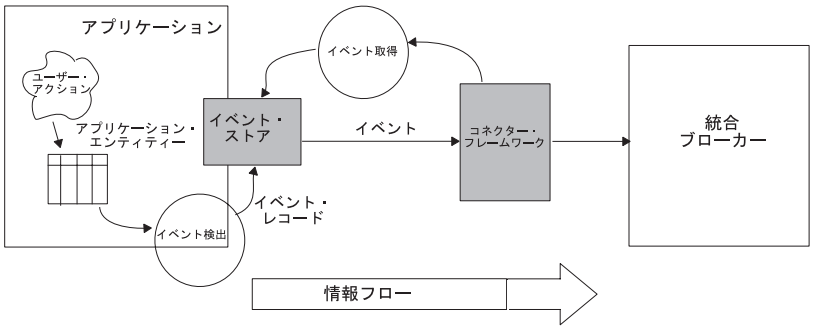

図 11. イベントの検出および取得

要求処理

コネクターには、アプリケーション・イベントの検出以外に、統合ブローカーからの要求に応答するという役割があります。統合ブローカーがコネクターのアプリケーションに対して変更を要求するか、または統合ブローカーがコネクターのアプリケーションからの情報を必要とする場合、コネクターは、統合ブローカーから*要求ビジネス・オブジェクト* を受信します。通常、コネクターは、コラボレーションからの要求に応答して、作成、取得、更新の各操作をアプリケーション・データに対して実行します。アプリケーションのポリシーによっては、コネクターは削除操作もサポートできます。この役割のことを*要求処理* といいます。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、要求処理のことを「サービス呼び出し要求」と呼ぶこともあります。コネクターは、コネクター・コントローラーがコラボレーションのサービス呼び出し要求によって受け取ったビジネス・オブジェクトを、そのコネクター・コントローラーから受け取ります。

注: このセクションでは、要求処理の概要について説明します。コネクターに要求処理をインプリメントする方法については、91 ページの『第 4 章 要求処理』を参照してください。

要求処理の手順は、以下のとおりです。

1. 23 ページの図 10 に示すように、統合ブローカーは、コネクター・フレームワークに要求を送信することによって要求処理を開始します。この要求は、*要求ビジネス・オブジェクト* と呼ばれるビジネス・オブジェクトと動詞の形式で構成されています。詳細については、28 ページの『要求の開始』を参照してください。
2. コネクター・フレームワークには、アプリケーション固有のコンポーネント内部にある*ビジネス・オブジェクト・ハンドラー* が、要求ビジネス・オブジェクトを処理するかどうかを決定するというタスクがあります。詳細については、29 ページの『ビジネス・オブジェクト・ハンドラーの選択』を参照してください。
3. コネクター・フレームワークは、要求ビジネス・オブジェクトを、このオブジェクトのためにビジネス・オブジェクト定義内に定義されたビジネス・オブジェクト・ハンドラーに渡します。

   コネクター・フレームワークは、ビジネス・オブジェクト・クラスに定義されている `doVerbFor()` メソッドを呼び出して、要求ビジネス・オブジェクトに渡すことによって、この処理を実行します。次に、ビジネス・オブジェクト・ハンドラーは、このビジネス・オブジェクトを処理して、1 つ以上のアプリケーション要求に変換します。
4. ビジネス・オブジェクト・ハンドラーは、アプリケーションとの対話を完了すると、戻り状況記述子と、場合によっては応答ビジネス・オブジェクトをコネクター・フレームワークに返します。詳細については、30 ページの『要求に対する応答の取り扱い』を参照してください。

要求の開始

要求を開始する方法は、IBM WebSphere Business Integration システム内部の統合ブローカーによって、以下のいずれかに分かれます。

- 『InterChange Server による要求の開始』
- 29 ページの『その他の統合ブローカーによる要求の開始』

InterChange Server による要求の開始: ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、コラボレーションが*サービス呼び出し要求* を開始し、いずれかのコラボレーション・ポートを介してこの要求を送信します。コラボレーション・オブジェクトのポートを結合する場合は、ポートとコ

ネクター (または別のコラボレーション・オブジェクト) とを関連付けます。コラボレーション・ポートでは、結合したエンティティー間の通信が可能になるので、コラボレーション・オブジェクトは、そのビジネス・プロセスを起動するビジネス・オブジェクトを受け入れて、ビジネス・オブジェクトをサービス呼び出し要求やサービス呼び出し応答としてやり取りできるようになります。

注: コラボレーション・ポートの定義方法の詳細については、「*コラボレーション開発ガイド*」を参照してください。コラボレーション・オブジェクトのポートを結合する方法については、「*WebSphere InterChange Server システム・インプリメンテーション・ガイド*」を参照してください。これらのドキュメントは両方とも IBM WebSphere InterChange Server ドキュメンテーション・セットにあります。

サービス呼び出し要求が開始されると、InterChange Server システムは以下の手順を実行します。

1. コラボレーション・ポートに結合しているコネクターのコネクター・コントローラーがサービス呼び出し要求を受信します。コネクター・コントローラーは、必要に応じて、汎用ビジネス・オブジェクトからアプリケーション固有のビジネス・オブジェクトへのマッピングをしてから、要求をコネクター・フレームワークに送信します。
2. コネクター・コントローラーは、サービス呼び出し要求をコネクター・フレームワークに転送します。コネクター・コントローラーは、要求ビジネス・オブジェクトを Java オブジェクトとして送信します。

その他の統合ブローカーによる要求の開始: ビジネス・インテグレーション・システムが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用している場合、統合ブローカーは、コネクターに関連付けられている WebSphere MQ キューにメッセージを送信することによって要求を開始します。 要求が開始されると、コネクター・フレームワークは、そのトランスポート層を使用して WebSphere MQ メッセージを取り出し、カスタムのデータ・ハンドラーを使用して、このメッセージを適切なビジネス・オブジェクトに変換します。

IBM WebSphere Business Integration システムと要求処理についての詳細は、ご使用の統合ブローカーのインプリメンテーション・ガイドを参照してください。

ビジネス・オブジェクト・ハンドラーの選択

ビジネス・オブジェクト・ハンドラー とは、要求ビジネス・オブジェクトを適切なアプリケーション操作の要求に変換する役割を果たす Java クラスのことです。アプリケーション固有のコンポーネントには、1 つ以上のビジネス・オブジェクト・ハンドラーが組み込まれており、コネクターがサポートしているビジネス・オブジェクトに登録されている動詞のタスクを実行します。ビジネス・オブジェクト・ハンドラーは、アクティブな動詞に応じて、ビジネス・オブジェクトに関連付けられているデータをアプリケーションに挿入したり、オブジェクトの更新、取得、削除などのタスクを実行したりすることができます。

コネクター・フレームワークは、この応答ビジネス・オブジェクトのビジネス・オブジェクト定義に基づいて、関連のビジネス・オブジェクトに対する正しいビジネス・オブジェクト・ハンドラーを入手します。

- コネクターが始動すると、コネクター・フレームワークは、コネクターがサポートしているビジネス・オブジェクトのリストをコネクター・コントローラーから受信します。
- コネクター・フレームワークは、(コネクターの基底クラスに定義されている)、`getConnectorBOHandlerForBO()` メソッドを呼び出して、1 つ以上のビジネス・オブジェクト・ハンドラーをインスタンス化します。
- `getConnectorBOHandlerForBO()`メソッドは、サポートしているビジネス・オブジェクトごとに、ビジネス・オブジェクト・ハンドラーへの参照を返します。この参照は、コネクター・プロセスのメモリーにあるビジネス・オブジェクト定義に格納されます。

ビジネス・オブジェクト間のすべての変換とアプリケーション操作は、ビジネス・オブジェクト・ハンドラーの内部で実行されます。

`getConnectorBOHandlerForBO()` メソッドをインプリメントする方法の詳細については、75 ページの『ビジネス・オブジェクト・ハンドラーの取得』を参照してください。

要求に対する応答の取り扱い

コネクターは、この要求を処理してアプリケーションとの対話を完了すると、統合ブローカーへ応答を戻すことができます。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、コネクター・フレームワークは、*サービス呼び出し応答* をコラボレーションに返します。コラボレーションは、戻り状況記述子の情報を利用することにより、そのサービス呼び出し要求の状況を判断して、適切な処置を施すことができます。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークの応答には以下のものが含まれます。

- 状況表示。これには、戻り状況記述子の情報が記録されています。
- 任意のビジネス・オブジェクト・メッセージ。これには、オプションの応答ビジネス・オブジェクトが格納されています。

コネクター・フレームワークは、この応答情報をコネクターのキューに書き込みます。ただし、同期をとる必要があるメッセージ転送の場合 (つまり、応答を待機する方式のプログラムの場合)、プログラムは統合ブローカーへの要求メッセージを同期要求キューに通知し、統合ブローカーからの応答が同期応答キューに書き込まれるのを待機する必要があります。応答メッセージの相関 ID はメッセージ要求の応答を識別します。

アダプター開発用のツール

IBM WebSphere Business Integration システムでは、コネクター は WebSphere Business Integration Adapter のコンポーネントです。3 ページの『WebSphere Business Integration システムのアダプター』で説明したように、アダプター は、統合ブローカーとアプリケーションまたはテクノロジー間の通信をサポートするためのランタイム・コンポーネントを内蔵しています。また、アダプターにはアダプター・フレームワーク も組み込まれています。このフレームワークには、特定のレガシー・アプリケーションまたは特殊アプリケーション用のビルド済みアダプターが WebSphere Business Integration Adapters 製品の一部として現在使用できない場合のために、カスタム・アダプターの構成、ランタイム、および開発用のコンポーネントが含まれています。

アダプター・フレームワークには、表 7 にリストされているアダプター・コンポーネントの開発を支援する構成ツールが組み込まれています。

表 7. コネクター開発のためのアダプター・フレームワーク・サポート

アダプター・コンポーネント	構成ツール	API
ビジネス・オブジェクト	Business Object Designer	該当なし
Object Discovery Agent (ODA)	Business Object Designer	Object Discovery Agent Development Kit (ODK)
コネクター	Connector Configurator	Java コネクター・ライブラリー

アダプター・フレームワークの他に、WebSphere Business Integration Adapters 製品は *Adapter Development Kit* (ADK) も提供しています。 ADK は、コネクター、ODA、およびデータ・ハンドラーのコード・サンプルを提供するツールキットです。詳細については、 33 ページの『Adapter Development Kit』を参照してください。

ビジネス・オブジェクトの開発サポート

表 8 に、ビジネス・オブジェクトの開発を支援するために WebSphere Business Integration Adapters 製品が提供するツールを示します。

表 8. ビジネス・オブジェクト開発用の開発ツール

開発ツール	説明
Business Object Designer	ビジネス・オブジェクト定義の作成を、手動または ODA を介して支援するグラフィック・ツール。

ビジネス・オブジェクトの概要については、5 ページの『ビジネス・オブジェクト』を参照してください。Business Object Designer の詳細な使用方法については、「*ビジネス・オブジェクト開発ガイド*」を参照してください。

ODA の開発サポート

表 8 に、ODA の開発を支援するために WebSphere Business Integration Adapters 製品が提供するツールを示します。

表 9. ODA 開発用の開発ツール

開発ツール	説明
Business Object Designer	ビジネス・オブジェクト定義の作成を、手動または ODA を介して支援するグラフィック・ツール。
Object Discovery Agent Development Kit (ODK)	カスタム ODA を作成するための Java クラスのセット。

また、ADK は、製品の次のサブディレクトリーにサンプル ODA を提供しています。

```
DevelopmentKits¥Odk
```

ODA の概要については、5 ページの『ビジネス・オブジェクト』を参照してください。Business Object Designer の使用と ODA の開発の詳細については、「*ビジネス・オブジェクト開発ガイド*」を参照してください。

コネクターの開発サポート

表 10 にコネクターの開発を支援するために WebSphere Business Integration Adapters 製品が提供するツールを示します。

表 10. コネクター開発用の開発ツール

開発ツール	説明
Connector Configurator	コネクターの構成を支援するグラフィック・ツール
Adapter Development Kit	Java コネクターおよび ODA のサンプル・コードが含まれています。

コネクターの開発がサポートされるオペレーティング・システム環境は Windows 2000 です。コネクターの記述には、アプリケーション API の言語に応じて、C++ と Java のいずれを使用しても構いません。

Connector Configurator

Connector Configurator は、コネクターの構成を可能にするグラフィック・ツールです。このツールには、以下のものを設定できる機能があります。

- コネクター構成プロパティー
- サポートされるビジネス・オブジェクト
- 関連マップ (InterChange Server のみ)
- ログ・ファイルおよびメッセージ・ファイル
- データ・ハンドラー構成 (保証付きイベント・デリバリーの場合)

このグラフィック・ツールは、Windows 2000 および Windows XP 上で動作します。したがって、これらのプラットフォームは、コネクターの構成に対応しています。

注: Connector Configurator の使用に関する詳細については、569 ページの『付録 B. Connector Configurator』を参照してください。

Adapter Development Kit

Adapter Development Kit (ADK) は、アダプター開発を支援するファイルおよびサンプルを提供します。Adapter Development Kit は、多くの Object Discovery Agent (ODA)、コネクター、およびデータ・ハンドラーを含むアダプター・コンポーネントにサンプルを提供します。 ADK が提供するサンプルは、製品ディレクトリーの `DevelopmentKits` サブディレクトリーにあります。

注: ADK は WebSphere Business Integration Adapters 製品の一部であり、専用のインストーラーを必要とします。このため、ADK の開発サンプルにアクセスするには、WebSphere Business Integration Adapters 製品にアクセスして ADK をインストールする必要があります。ADK は Windows システムでのみ 使用できることに注意してください。

表 11 に、コネクターを開発するために ADK が提供するサンプルと、それらのサンプルが置かれている `DevelopmentKits` ディレクトリーのサブディレクトリーを示します。

表 11. コネクター開発用の ADK サンプル

Adapter Development Kit コンポーネント	説明	DevelopmentKits サブディレクトリー
Java Connector Development Kit (JCDK)	Java コネクターのサンプル・コードを提供します。	`jcdk`
Twineball アダプター (Twineball adapter) サンプル	サンプル・アダプターを提供します。これにはコネクターが含まれます。	`edk¥ConnectorAgent Twineball_sample`

ADK は、`DevelopmentKits` の `Twineball_sample` サブディレクトリーでアダプター・サンプルを提供します。このサンプルには、コネクター、データ・ハンドラー、Object Discovery Agent (ODA) など、いくつかのアダプター・コンポーネントが含まれています。詳細については、「*Adapter Development Kit Samples Guide*」を参照してください。

Connector Development Kit: ADK に含まれている Java Connector Development Kit (JCDK) は、コネクターの開発で使用されるコンポーネントを提供します。JCDK のコンポーネントは、以下の *ProductDir*¥DevelopmentKits サブディレクトリーに入っています。

```
DevelopmentKits¥jcdk
```

表 12 には、jcdk ディレクトリー内のサブディレクトリーの内容が示されています。

表 12. Connector Development Kit のコンポーネント

Connector Development Kit コンポーネント	説明	サブディレクトリー
コード・サンプル	単純な低レベル Java コネクターのサンプル・コード	samples

JCDK には、以下のコード・サンプルが含まれています。このサンプルは、下位の Java コネクター・ライブラリーで作成した Java コネクターの開発に利用できます。

```
DevelopmentKits¥jcdk¥samples
```

さらに、JCDK の以下のディレクトリーには、Java コネクター・ライブラリーで作成された Java コネクターのコード・サンプルも含まれています。

```
DevelopmentKits¥edk¥ConnectorAgent
```

Java コネクターをコンパイルするには、IBM Java Developers Kit (JDK) に付属している Java コンパイラーを使用してください。詳細については、242 ページの『コネクターのコンパイル』を参照してください。

注: WebSphere Business Integration Adapters 製品では、C++ プログラミング言語のコネクター開発に使用できる Connector Development Kit の C++ 版も用意しています。詳細については、「*コネクター開発ガイド (C++ 用)*」を参照してください。

ODA のサンプル: Adapter Development Kit には、Object Discovery Agent (ODA) のためのサンプルが含まれています。これらのサンプルは、以下のディレクトリーにあります。

```
DevelopmentKits¥Odk
```

詳細については、32 ページの『ODA の開発サポート』を参照してください。

コネクター開発工程の概要

このセクションでは、コネクター開発工程の概要について説明します。その内容は、以下のような基本的な手順です。

1. IBM WebSphere Business Integration システム・ソフトウェアのインストールとセットアップ、および Java Development Kit (JDK) をインストールする。
2. コネクターの設計および実装を行う。

開発環境のセットアップ

開発工程を開始する前に、以下の条件が整っていることを確認する必要があります。

- IBM WebSphere Business Integration システム・ソフトウェアは、開発者がアクセスできるマシンにインストールされていること。

WebSphere InterChange Server

ご使用のビジネス・インテグレーション・システムが InterChange Server を使用している場合、InterChange Server システムのインストールおよび始動方法については、「*システム・インストール・ガイド (UNIX 版)*」または「*システム・インストール・ガイド (Windows 版)*」 (WebSphere InterChange Server ドキュメンテーション・セットに同梱) を参照してください。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere Integrator Broker、WebSphere Business Integration Message Broker) を使用するビジネス・インテグレーション・システムの場合、IBM WebSphere Business Integration システムのインストール方法および始動方法について「*WebSphere Message Brokers 使用アダプター・インプリメンテーション・ガイド*」のインストールに関する章を参照してください。ご使用のビジネス・インテグレーション・システムが WebSphere Application Server を使用している場合、IBM WebSphere Business Integration システムのインストールおよび始動の方法については、「*アダプター実装ガイド (WebSphere Application Server)*」を参照してください。

- Java Development Kit (JDK) 1.3.1 または JDK 対応の開発向け製品が開発用のマシンにインストールされていること。

  Java コンパイラーは、JDK の一部です。したがって、新規コネクターを作成するには、JDK がインストールされている必要があります。

  – Windows プラットフォームの場合は、IBM JDK は製品 CD で提供されます。ただし、製品のインストーラーは、システムに自動的にはインストールされません。JDK を InterChange Server 製品の一部としてインストールする方法については、「*システム・インストール・ガイド (Windows 版)*」を参照してください。WebSphere Business Integration Adapters 製品の一部としてインストールする方法については、「*WebSphere Business Integration Adapters インストール・ガイド*」を参照してください。

  – UNIX プラットフォームの場合は、Web サイトから JDK をダウンロードしてシステムにインストールする必要があります。JDK を InterChange Server 製品の一部としてインストールする方法については、「*システム・インストール・ガイド (UNIX 版)*」を参照してください。WebSphere Business Integration Adapters 製品の一部としてインストールする方法については、「*WebSphere Business Integration Adapters インストール・ガイド*」を参照してください。

- コネクターのライブラリー・ファイルが格納されているディレクトリーに、開発環境からアクセスできること。コネクターをコンパイルするには、コンパイラーがコネクター・ライブラリーにアクセスできる 必要があります。

  コネクターのコンパイルについては、 242 ページの『コネクターのコンパイル』を参照してください。

InterChange Server

- ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、InterChange Server のリポジトリーのデータベース・サーバーと ICS が動作していること。

  注: この手順が必要なのは、Connector Configurator によってコネクターを構成する準備が整っている場合のみです。開発のみの場合は、ICS に接続せずにコネクター・クラスを作成できます。

  コネクターを構成する方法の概要については、 241 ページの『第 8 章 ビジネス・インテグレーション・システムへのコネクターの追加』を参照してください。IBM WebSphere Business Integration システムの始動については、使用システムのインストール・ガイドを参照してください。

InterChange Server の終り

注: コネクターを作成する場合は、メッセージング・ソフトウェアを実行する必要はありません。ただし、コネクターを実行してテストする場合は、あらかじめメッセージング・ソフトウェアを動作させておく必要があります。

コネクター開発の各段階

コネクター開発工程の一部として、コネクターのアプリケーション固有コンポーネントをコーディングし、次に、コネクターのソース・ファイルをコンパイルしてリンクします。さらに、コネクターの開発工程全体には、アプリケーション固有のビジネス・オブジェクトの開発など、ほかのタスクも含まれます。以下に示す手順は、コネクター開発工程におけるタスクの概要です。

1. コネクターがほかのアプリケーションに対して利用可能にするアプリケーション・エンティティーを突き止めて、このアプリケーションが備えている統合機能を調べる。

InterChange Server

2. ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、コネクターがサポートしている汎用のビジネス・オブジェクトを特定し、この汎用オブジェクトに対応するアプリケーション固有のビジネス・オブジェクトを定義する。
3. ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、汎用のビジネス・オブジェクトとアプリケーション固有のビジネス・オブジェクトとの関係を分析して、両者間のマッピングを実行する。

InterChange Server の終り

4. アプリケーション固有のコンポーネントに対してコネクター基底クラスを定義して、コネクターの初期設定と停止の機能を実装する。
5. ビジネス・オブジェクト・ハンドラー・クラスを定義し、ビジネス・オブジェクト・ハンドラーを 1 つ以上コーディングして、要求を処理する。
6. アプリケーション内部で発生するイベントを検出する機構を定義し、この機構を実装して、イベント・サブスクリプションをサポートする。
7. すべてのコネクター・メソッドに対するエラー処理とメッセージ処理を実装する。
8. コネクターを作成する。
9. コネクターを構成する。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合は、Connector Configurator を使用してコネクター定義を作成し、これを InterChange Server リポジトリーに保存します。 Connector Configurator は System Manager から呼び出すことができます。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、Connector Configurator を使用してコネクター構成ファイルを定義および作成します。

10. 複数のコネクター・コンポーネント間のメッセージングに WebSphere MQ を使用する場合は、コネクターにメッセージ・キューを付加する。
11. 新しいコネクターの始動スクリプトを作成する。
12. コネクターのテストとデバッグを行い、必要に応じて記録をとる。

図 12 では、コネクターの開発工程の概要を視覚的に説明し、特定のトピックに関する情報を取得できる各章の早見表を示します。コネクター開発をチームで行う場合は、コネクター開発の主なタスクを、コネクター開発チームの異なるメンバーが並行して進めることができます。

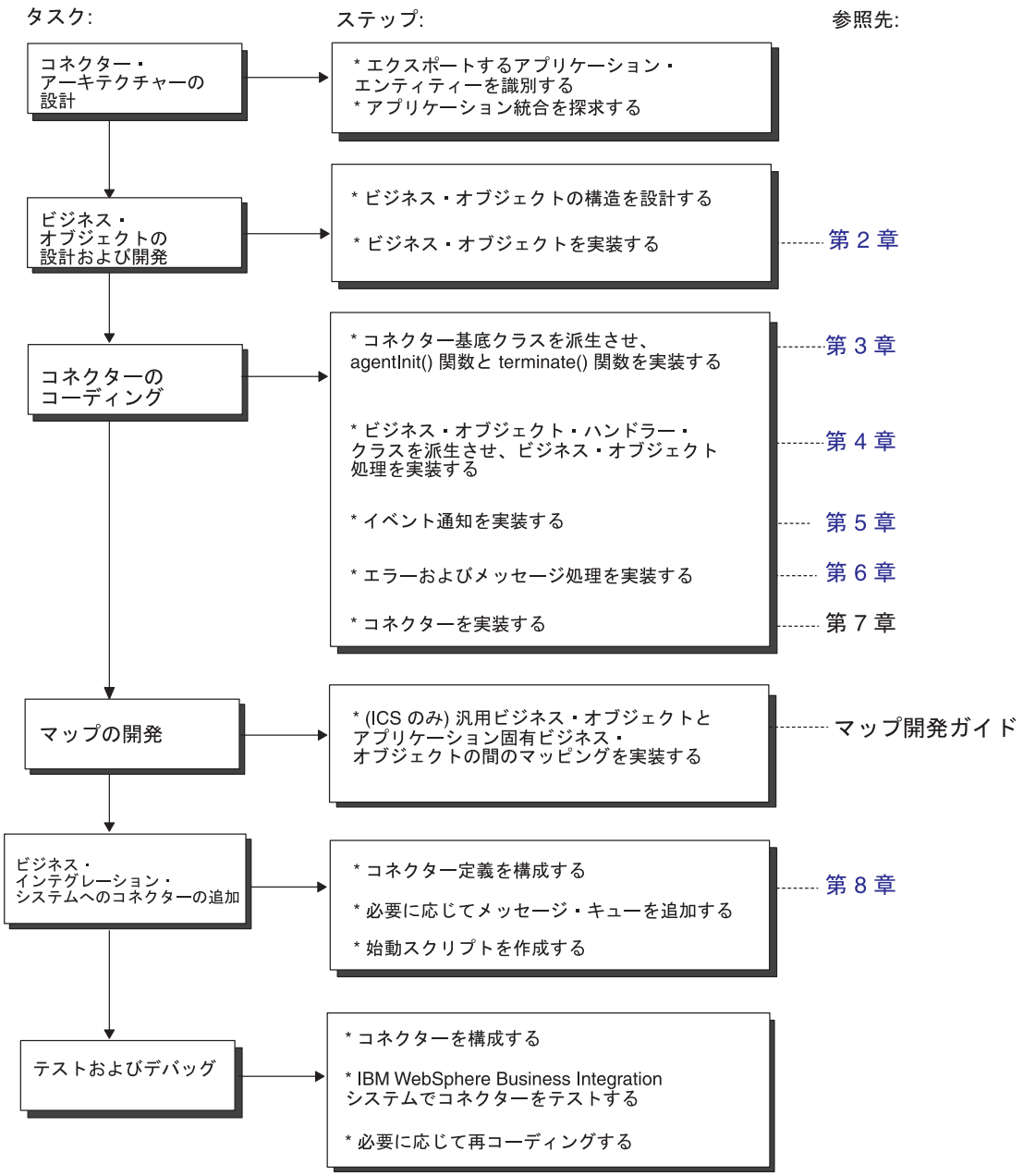

図 12. Java コネクター開発工程の概要

第 2 部 コネクターの作成

第 2 章 コネクターの設計

この章では、コネクター開発プロジェクトの計画時に考慮する分析や設計上の課題についてその概要を説明します。この章では、使用のアプリケーションまたはテクノロジーに対応するコネクター開発での複雑さを判断するときに役立つ内容を紹介します。

大半のソフトウェア開発プロジェクトと同様に、コネクター開発サイクルの早期の段階に計画を慎重に策定すれば、その後の実装段階での問題発生を防止することにつながります。この章を構成するセクションは次のとおりです。

- 『コネクター開発プロジェクトのスコープ』
- 42 ページの『コネクター・アーキテクチャーの設計』
- 48 ページの『アプリケーション固有のビジネス・オブジェクトの設計』
- 57 ページの『イベント通知』
- 58 ページの『オペレーティング・システム間での通信』
- 59 ページの『計画に関する質問のまとめ』
- 62 ページの『国際化対応コネクター』

コネクター開発プロジェクトのスコープ

IBM では、*コネクター・フレームワーク* を Java Connector Development Kit の一部として用意しています。コネクター・フレームワークには、コネクターが統合ブローカーと対話するために必要なすべてのコードが格納されており、アプリケーションと対話するための基盤が用意されています。

コネクター開発者としての作業は、コネクターのアプリケーション固有コンポーネントをコーディングすることと、必要に応じてイベント通知機構を開発することです。コネクターの設計の複雑さや、コネクターの実装に必要な時間は、対象のアプリケーションによって異なります。

コネクター開発プロジェクトのスコープと複雑さを理解するには、新規コネクターに着手する前にプロジェクト計画を策定するのが順序です。プロジェクト計画を策定するにつれて、コネクターの業務要件の特定、コネクターの処理対象となるアプリケーション・データの定義、コネクターとビジネス・オブジェクトとの連携対象となるアプリケーション・ビジネス・プロセスの指定などを実行する必要があります。プロジェクト計画を策定すると、ビジネス・オブジェクト、ビジネス・オブジェクトの処理、およびイベント管理の領域でアプリケーションの機能を理解しやすくなります。

この章のトピックに取り組むことにより、コネクター開発作業を完了するために必要な時間と労力を見積もることが容易になります。各トピックには、一連の質問が用意されています。これらの質問の目的は、コネクター開発作業の複雑さを増加または低減する可能性があるアプリケーション固有の性質について、その理解を深めることです。各トピックの一連の質問に対しては、詳細な答えが用意されており、この答えが、開発するコネクターの上位アーキテクチャーとなります。

コネクターの設計手順	詳細情報
コネクター・アーキテクチャーの設計に関係のあるアプリケーションの情報を入手する。	42 ページの『コネクター・アーキテクチャーの設計』
アプリケーション固有のビジネス・オブジェクトが、コネクターによるエクスポートが必要なアプリケーション・エンティティーを適切に表現していることを確認する。	48 ページの『アプリケーション固有のビジネス・オブジェクトの設計』
アプリケーションが関係のあるイベントをコネクターに通知できるようにイベント通知機構を設計する。	57 ページの『イベント通知』

コネクター・アーキテクチャーの設計

コネクター・アーキテクチャーを設計するには、コネクターのサポートが必要となる以下の領域のアプリケーションの評価を検討します。

- 43 ページの『アプリケーション環境の理解』
- 44 ページの『コネクターの方向性の決定』
- 45 ページの『アプリケーションに対するデータの書き込みおよび読み出し』

アプリケーション内部で、コネクター設計に影響のある特定の領域を、図 13 に示します。この図では、コネクター開発に必要な上位タスクが雲の形で表現されています。

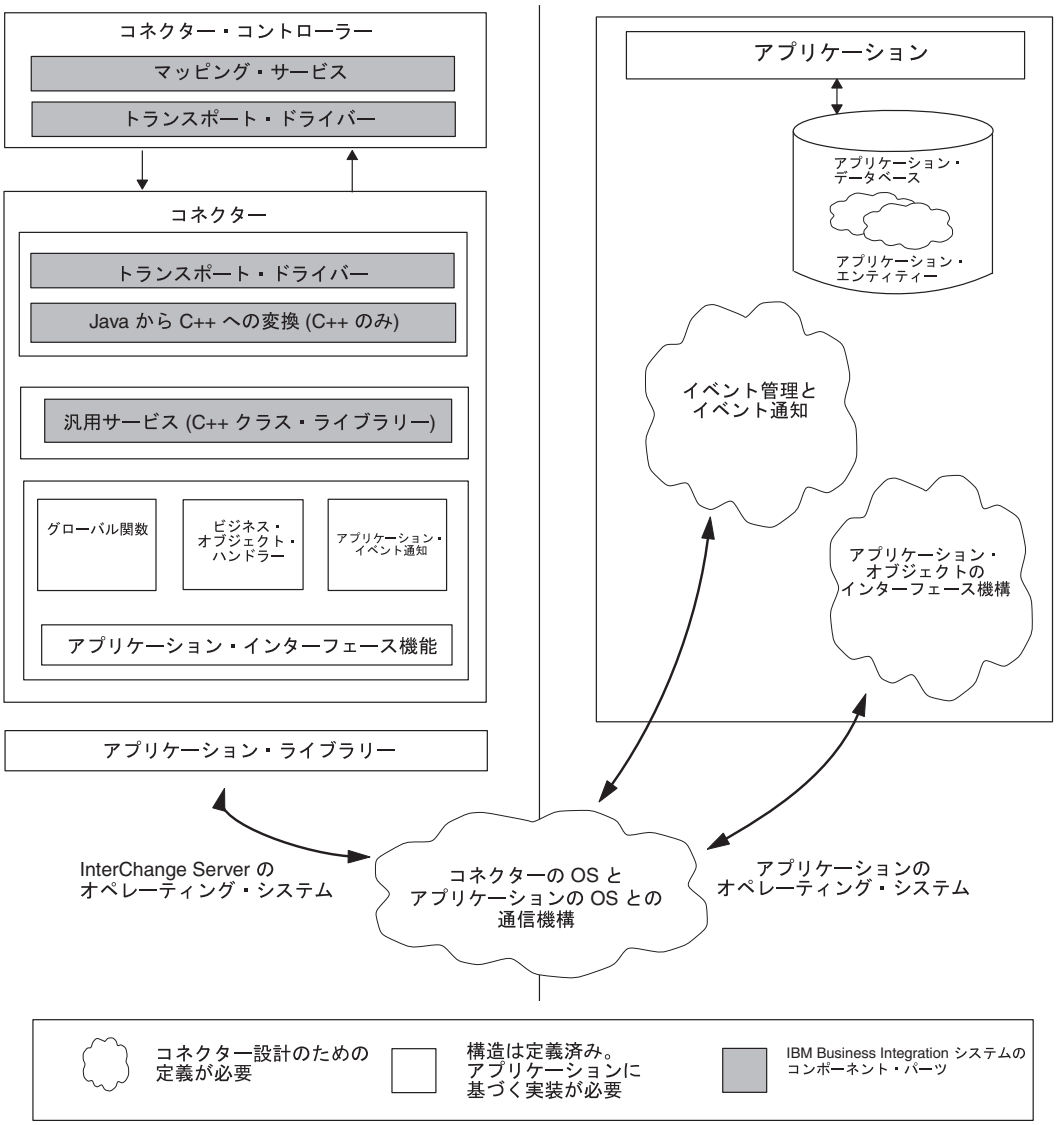

図 13. コネクター設計に影響のあるアプリケーションの領域

アプリケーション環境の理解

アプリケーション環境を理解することは、コネクター開発プロジェクトの実現可能性を分析評価するための第一歩です。コネクター開発に影響のあるアプリケーションの性質を理解するために、以下のトピックおよび質問について考えてみましょう。

オペレーティング・システム

- 対象のアプリケーションが動作するオペレーティング・システムは何ですか?

プログラム言語

- アプリケーションの記述に使用されたプログラム言語は何ですか?

アプリケーション実行アーキテクチャー

- アプリケーションの実行アーキテクチャーは何ですか? 例えば、集中型アーキテクチャーでは、アプリケーションとそのデータベースの両方がメインフレーム・システムに置かれている場合があります。この場合、アプリケーションの処理とデータベースの処理が両方とも集中型システム上で実行されます。

  これとは異なり、クライアント/サーバー・アーキテクチャーでは、データベースがサーバー上にあり、アプリケーションのフロントエンド・プログラムは、パーソナル・コンピューターなどの別のマシンで動作するクライアントである場合があります。そのほかの種類のアプリケーション実行アーキテクチャーは、オンライン・トランザクション処理やファイル・サーバーのアーキテクチャーです。

データベースの種類

- アプリケーション・データ用の中央データベースはありますか? アプリケーション・データが中央データベースに格納されている場合、そのデータベースの種類は何ですか? この種類のデータベースの例は、RDMS やフラット・ファイルなどです。

分散アプリケーション

- アプリケーションは複数のサーバーにまたがって分散していますか?
- アプリケーション・データベースは複数のサーバーにまたがって分散していますか?

プロジェクトの分析評価時には、アプリケーションの専門家を探して一緒に作業を進めたい場合があります。この人物は、ビジネス・オブジェクトの開発やコネクターの開発時にも支援が可能です。

コネクターの方向性の決定

プロジェクト計画の初期段階には、アプリケーションに対してコネクターが果たす役割を決めることが必要です。

- 要求処理 — 統合ブローカーの要求時にアプリケーション・データを更新します。詳細については、27 ページの『要求処理』を参照してください。
- イベント通知 — アプリケーション・イベントを検出して、イベント通知を統合ブローカーに送信します。詳細については、24 ページの『イベント通知』を参照してください。

これらの役割は、コネクターがサポートする*方向性* を決定します。

- 単一方向 — 一部のコネクターでは、片方向のみの動作、つまりアプリケーションから統合ブローカー、または統合ブローカーからアプリケーションのいずれか 1 つの方向にデータを渡す動作が必要な場合があります。
  - アプリケーションに変更が生じたことを統合ブローカーに通知するには、コネクターがイベント通知機能をサポートしている必要があります。
  - 統合ブローカーからデータを受信するには、コネクターは要求処理をサポートする必要があります。この要求処理では、コネクターがアプリケーションと対話して、統合ブローカーによって要求されたとおりに Create、Retrieve、Update、Delete のいずれかの操作をサポートします。

例えば、コネクターに必要な処理は、統合ブローカーから要求ビジネス・オブジェクトを受信して、これをアプリケーションに渡すことのみの場合があります。宛先が単一方向コネクターの場合にのみ機能するアプリケーション向けのコネクター - このコネクターには、要求を処理してデータをアプリケーションに渡す機能は実装されていますが、イベント通知機能は実装されていません。コネクターが単一方向にのみ動作するということを開発サイクルの初期に認識することにより、開発期間を大幅に短縮することができます。

- 双方向 — 大半のコネクターは、*双方向* で動作する必要があります。つまり、アプリケーションから統合ブローカーの方向へデータを渡し、かつ 統合ブローカーからの戻りデータを受信することが必要です。

  コネクターを双方向で動作させるには、イベント通知と要求処理の*両方の* イベント処理機能が必要です。

使用しているコネクターでイベント通知機能をサポートできるようにする方法については、131 ページの『第 5 章 イベント通知』を参照してください。

アプリケーションに対するデータの書き込みおよび読み出し

コネクター開発プロジェクト計画の重要な側面は、コネクターがアプリケーションにデータを書き込んだり、アプリケーションからデータを読み出したりする方法を決定することです。理論的には、以下のすべての機能を備えたアプリケーション・プログラミング・インターフェース (API) が各アプリケーションに用意されています。

- オブジェクト・レベルでの Create、Retrieve、Update、Delete (CRUD) の各操作に対するサポート
- すべてのアプリケーション・ビジネス・ロジックのカプセル化
- デルタ操作と変更後イメージ操作のサポート
- サブオブジェクト・レベルでの外部通知を可能にするイベント管理の方針

ただし、通常は、アプリケーション・インターフェースの状態がこのような理想的な状態には到達していません。

策定するプロジェクト計画では、公式のアプリケーション API が存在するかどうかを確認し、その頑強性を評価する必要があります。API が*存在しない* 場合は、適切な対応策があるかどうかを調べます。アプリケーションの CRUD インターフェースは、バッチ・ファイルのインポートや抽出から COM/DCOM サーバーまで、あらゆるものが可能であるため、可能性のあるすべての経路を調べてください。アプリケーション・オブジェクトの CRUD インターフェースを調べる場合は、表 13 に指定されているアプリケーション・ビジネス・オブジェクトのスコープを参照してください。

以下のタスクについて考えてみましょう。

- 46 ページの『以前の統合作業の検討』 — このアプリケーションと統合するための過去の作業実績はほかにありますか ?
- 46 ページの『アプリケーション・データを他のアプリケーションと共用するかどうかの決定』 — アプリケーション・データはほかのアプリケーションと共用しますか ?

- 47 ページの『アプリケーション API の検討』 — コネクターがこのアプリケーションと通信するときに使用できる既存の機構はありますか ?
- 48 ページの『バッチ・クリーンアップ・プログラムまたはマージ・プログラムのアプリケーションによる使用』 — アプリケーションはバッチ・クリーンアップ・プログラムまたはマージ・プログラムを使用しますか ?

これらの質問の詳細については、以降のセクションで説明します。

以前の統合作業の検討

使用しているアプリケーションとほかのアプリケーションとを統合するための以前の方法を利用できる場合は、対象のアプリケーションにデータを書き込んだり、アプリケーションからデータを読み出したりする方法を取得できる場合があります。アプリケーションの統合に別の手法を採用することにした場合でも、以前の統合作業の実績から有益な設計情報が得られる場合があります。

以前の統合作業実績を検討する場合には、以下の質問の答えを考えてください。

- 統合の目的は何ですか?
- 統合済みアプリケーションには、統合前のアプリケーションからの情報を変更または取得するインターフェースを使用しますか? 使用する場合は、情報を変更または取得するために使用する機構について説明してください。
- この統合手法が、アプリケーションで生成されたイベントを処理できる場合、イベント処理を起動するために使用する機構は何ですか?
- 既存の統合済みアプリケーションのモード (バッチ、非同期など) は何ですか?
- コネクターは既存の統合アプリケーションに置き換わりますか? 置き換わらない場合、以前の統合アプリケーションは、コネクターの処理対象となるデータ・エンティティーを処理しますか?

答えには、さまざまな方法でアプリケーションと対話する、これまでのすべての統合手法の情報を盛り込んでください。

アプリケーション・データを他のアプリケーションと共用するかどうかの決定

対象のアプリケーションは、1 つのデータベースの中でデータを作成したり更新したりする複数のアプリケーションのいずれかである場合があります。この場合、コネクターは、ほかのアプリケーションも実行している作業に基づいてアプリケーション・データのエンティティーを考慮する必要があります。コネクターとほかの複数のアプリケーションとがアプリケーション・データを共用することにした場合は、以下の質問について考えてください。

- アプリケーション・データへのアクセス回数を増やすためにほかのアプリケーションが使用している機構は何ですか?
- ほかのアプリケーションは、アプリケーション・データの作成、取得、更新、または削除を行いますか? そうである場合、ほかのアプリケーションが各動詞 (Create、Retrieve、Update、Delete) に対して使用する機構は何ですか?
- ほかのアプリケーションが使用しているオブジェクト固有のビジネス・ロジックはありますか? このロジックはすべてのアプリケーションにわたって一貫性がありますか?

アプリケーション・データを共用しているすべてのアプリケーションについて、これらの質問に答えてください。

アプリケーション API の検討

コネクターがアプリケーションと通信するために使用できる API などの機構をアプリケーションが提供している場合は、この API を検討して、利用可能な資料がないか調べます。API に関する以下の質問について考えてください。

- API は、Create、Retrieve、Update、Delete の各操作にアクセスできますか?
- API は、データ・エンティティーのすべての属性にアクセスできますか?
- API の実装に不整合はありますか? Create/Retrieve/Update/Delete へのナビゲーションは、エンティティーに関係なく同じですか?
- API のトランザクション動作を説明してください。例えば、ある API を利用すると、コネクターはレポート機能を実行できます。この後、コネクターはレポートを読み取って処理のために使用できます。または、API が堅固で、非同期または同期の Create 操作および Update 操作を実行する方法を提供する場合があります。
- API は、アプリケーションにアクセスしてイベントを検出できますか? 例えば、アプリケーションのイベント通知機構がイベント・ストアとしてデータベース表を使用する場合、API はこの表にアクセスできますか?
- API はメタデータ設計に適していますか? フォーム・ベース、表ベース、オブジェクト・ベースの各 API は、有力な候補です。メタデータ設計については、52 ページの『メタデータ主導型の設計を評価するためのサポート』を参照してください。
- API はアプリケーションのビジネス・ルールを適用しますか? 言い換えると、表レベル、フォーム・レベル、またはオブジェクト・レベルで対話するのは API ですか?

コネクター開発の推奨手法は、アプリケーション側が提供する API を API の種類にかかわらず使用することです。API を使用することにより、コネクターとアプリケーションとの対話が、アプリケーションのビジネス・ロジックを順守するようになります。特に、上位の API は、通常、アプリケーションのビジネス・ロジックのサポートを組み込む設計になっていますが、下位の API では、アプリケーションのビジネス・ロジックをバイパスする場合があります。

例として、データベース表に新規レコードを作成するための上位の API 呼び出しでは、ある範囲の値に対して入力データを評価したり、指定の表だけでなく複数の関連表も更新する場合があります。SQL ステートメントを使用してデータベースに直接書き込むと、API によって実行されるデータ評価および関連表の更新はバイパスされます。

API が用意されていない場合、アプリケーションによっては、SQL ステートメントの使用により、このアプリケーションのクライアントはそのデータベースに直接アクセスできます。SQL ステートメントを使用してアプリケーション・データを更新する場合は、このアプリケーションを熟知している作業者と密接に連係を取って作業し、コネクターがアプリケーションのビジネス・ロジックをバイパスしないようにします。

アプリケーションのこの性質は、コネクターが必要とするコードの規模に影響を与えるため、コネクターの設計に多大な影響を及ぼします。コネクター開発用の最も簡単なアプリケーションは、上位の API を介してアプリケーションのデータベースと対話するアプリケーションです。アプリケーションが備えている API が下位の API であるか、またはアプリケーションに API が存在しない場合は、コネクターにコードを追加する必要性が高くなります。

バッチ・クリーンアップ・プログラムまたはマージ・プログラムのアプリケーションによる使用

アプリケーションのビジネス・オブジェクト・インターフェースの特徴のうち、検討が必要な最後の特徴は、アプリケーションがバッチ・クリーンアップ・プログラムまたはマージ・プログラムを使用して重複データや無効なデータを消去するかどうかです。例えば、オペレーターが入力したサイト名の内容が不正確または不完全であった場合、サイト名を標準化するバッチ・プログラムを、アプリケーションによって 1 日に 1 回実行できます。このプログラムでは、例えば、IBM WebSphere という名前のすべてのサイトを IBM WebSphere Software という名前に変更することができます。

この種のバッチ・プログラムを実行した場合は、データベースに加えたすべての変更内容を、InterChange Server の顧客同期システムにも反映させる必要があります。このようなプログラムでは、コネクターに対して隠れた要件が発生することがあります。例えば、最初はコネクターに Delete 機能を用意する必要がなかったと考えられる場合でも、IBM WebSphere という名前のサイトをすべて削除するバッチ・クリーンアップ・プログラムをサポートするために Delete 機能を用意することが必要になる場合があります。

バッチ・クリーンアップ・タスクは、同期的ではなく定期的 (例: 月に 1 度) に実行する管理方法が考えられます。どのような場合でも、計画作業では、コネクターに対して予想外の要件が生じるすべてのプログラムについて情報を収集することが重要です。

アプリケーション固有のビジネス・オブジェクトの設計

アプリケーション固有のビジネス・オブジェクトは、アプリケーション内部で起動される作業の単位であり、コネクターによって作成され、処理されてから統合ブローカーに送信されます。コネクターは、これらのビジネス・オブジェクトを使用して、コネクターのアプリケーションからほかのアプリケーションへデータをエクスポートし、ほかのアプリケーションからデータをインポートします。

コネクターは、ほかのアプリケーションがデータを共用できるために必要なアプリケーション・エンティティーに関するすべての情報を公開します。このエンティティーがコネクターによってほかのアプリケーションで利用できるようになると、統合ブローカーは、ほかのアプリケーションのコネクターを介して、データを多数のアプリケーションに転送できます。

コネクターとコネクターがサポートしているアプリケーション固有のビジネス・オブジェクトとの関係を設計することは、コネクター開発における作業の 1 つです。アプリケーション固有のビジネス・オブジェクトの設計では、コネクターのプログラミング・ロジックの要件が発生する可能性がありますが、この要件は、コネクタ

一開発工程に組み込む必要があります。したがって、ビジネス・オブジェクト開発者とコネクター開発者は、必ず共同で作業し、コネクターとそのビジネス・オブジェクトの仕様を策定する必要があります。

アプリケーション固有のビジネス・オブジェクトを設計するときは、以下の設計指針を考慮してください。

1. コネクターの作業対象となるアプリケーション・エンティティーを何にするか決定する。
2. ビジネス・オブジェクト開発のスコープを決定する。
3. メタデータ主導型の設計に対するサポートを決定する。

注: アプリケーション固有のビジネス・オブジェクトの設計の詳細については、「*ビジネス・オブジェクト開発ガイド*」を参照してください。

アプリケーション・エンティティーの決定

ビジネス・オブジェクトの複雑さは、コネクターを作成するために必要な作業量に大きな影響を与える可能性があります。アプリケーション固有のビジネス・オブジェクトを決定するための第 1 段階は、コネクターの作業対象となるアプリケーション・エンティティーを何にするか決めることです。

コネクターの作業対象となるアプリケーション・エンティティーを決定する方法は、次のように 2 種類あります。

- 使用アプリケーションのビジネス・プロセスと一致するビジネス・プロセスを持つ InterChange Server コラボレーションを軸にする。
- 使用アプリケーションとの統合先にするほかのアプリケーションを軸にする。

InterChange Server Collaborations を軸にした設計

統合ブローカーとして InterChange Server を使用している場合、アプリケーション固有のビジネス・オブジェクトの決定を開始するための 1 つの方法は、アプリケーションによる作業の対象にする InterChange Server Collaborations をリストにすることです。各コラボレーションの特長を検討して、各コラボレーションが参照する汎用のビジネス・オブジェクトをメモします。このリストを使用することにより、どのような種類のビジネス・オブジェクトにすれば、使用アプリケーションとコラボレーションが連携して動作できるかがわかります。

例えば、使用のアプリケーションと Customer Manager コラボレーションとを連携して使用する場合を考えます。この場合、コネクターは顧客エンティティーを処理する必要があります。コネクターは、顧客データをアプリケーションから抽出してコラボレーションに転送するか、またはコラボレーションから顧客データを受信してアプリケーションに戻すことになります。

ほかのアプリケーションを軸にした設計

別の方法として、統合の対象とするほかのアプリケーションに着目し、コネクター開発作業を開始することもできます。使用アプリケーションとほかのアプリケーションを調べていくと、複数のアプリケーションにまたがって共用するビジネス・プロセスの種類や、交換するデータの種類を決定することができます。目標は、使用

アプリケーションのエンティティーのうち、ほかのアプリケーションとの統合を可能にするために、ビジネス・オブジェクトとして実装する意味のあるエンティティーを決定することです。

例えば、使用のアプリケーションが顧客データを格納する場合、この顧客データベースと、ほかのアプリケーションによる顧客データベースとの一貫性を維持することなどが考えられます。顧客データを同期化するには、各アプリケーションがパブリッシュする顧客エンティティーの内容を知ることが必要です。図 14 には、ほかのアプリケーションとの統合を軸にした設計手法を示します。

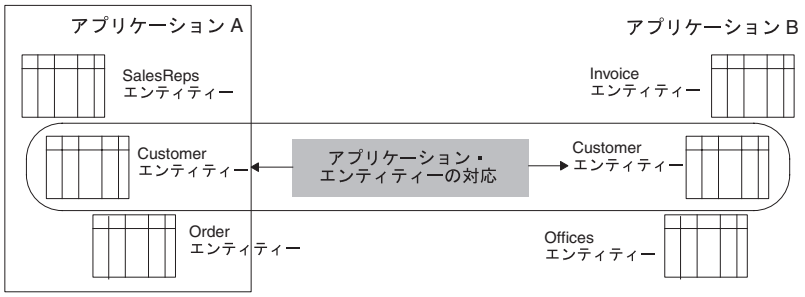

図 14. 設計フォーカス: 統合対象のアプリケーションの識別

アプリケーションを軸にした設計

以下のトピックと質問を活用して、アプリケーション・エンティティーとビジネス・オブジェクトに関する情報を収集してください。

- 『包含エンティティー』
- 『エンティティーのデータベース表現』
- 51 ページの『アプリケーション・エンティティーの非正規化』
- 51 ページの『アプリケーション・エンティティーのバッチ処理』

包含エンティティー:

- アプリケーション・エンティティーには、包含エンティティーがありますか?

  例えば、多くのアプリケーションでは、契約エンティティーに「1 対多」の行項目があります。IBM WebSphere Business Integration の Contract ビジネス・オブジェクトには、ビジネス・オブジェクトとして、子行項目が格納されています。コネクターの作業対象となるエンティティーが、子ビジネス・オブジェクトとして定義される関連エンティティーを持つかどうかを確認してください。

エンティティーのデータベース表現:

- 型は同じだが、アプリケーションでの物理表現は異なるアプリケーション・ビジネス・エンティティーはありますか?

  例えば、あるアプリケーションでは、ハードウェア契約とソフトウェア契約の 2 種類の契約があります。どちらも型は Contract (契約) ですが、これらはアプリケーション・データベース内の異なる表に格納されています。さらに、属性は Contract 型ごとに異なります。

1 組のマップは、1 つの汎用ビジネス・オブジェクトと 1 つのアプリケーション固有ビジネス・オブジェクトとの間でのみ変換が可能なため、このアプリケーションの開発者は、アプリケーション内部の異なるエンティティーを考慮してビジネス・オブジェクトを設計する必要があります。例えば、IBM WebSphere Business Integration の汎用ビジネス・オブジェクトを再設計して汎用の子ビジネス・オブジェクトを作成し、新規マップを作成することが必要な場合があります。

図 15 には、同じ型の複数のアプリケーション・エンティティーを基に作成できるビジネス・オブジェクトを示します。この図では、汎用の子ビジネス・オブジェクト 2 つの作成を表しています。一方にはハードウェア契約に固有のデータが格納されており、もう一方にはソフトウェア契約に固有のデータが格納されています。

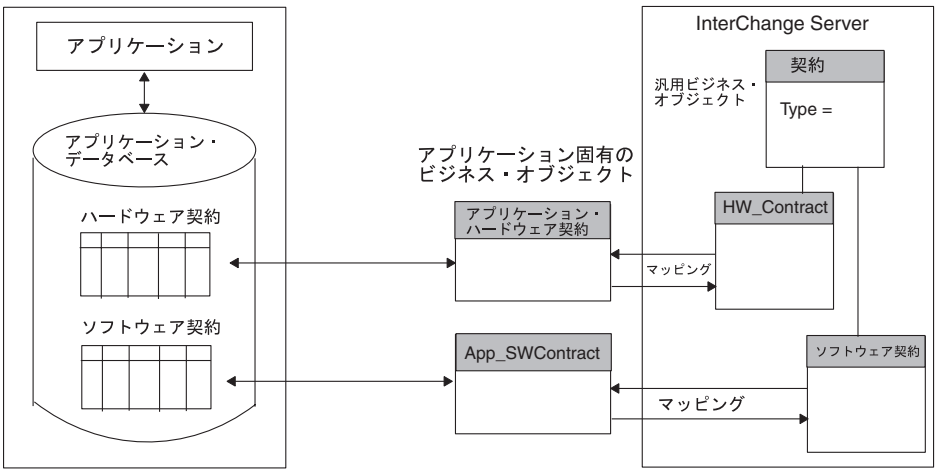

図 15. アプリケーション・エンティティーのデータベース表現

アプリケーション・エンティティーの非正規化: データベース内の複数の位置に存在するが、同じ論理エンティティーに対応するアプリケーション・エンティティーはありますか?

例えば、Contract、Customer、および Contact の各エンティティーには、それぞれのエンティティーごとに、物理表定義の一部として Customer アドレス・フィールドがあると考えられます。あるエンティティーの Customer アドレス・フィールドが変更された場合、このフィールドはすべてのエンティティーで更新する必要があります。

ただし、このアドレス・フィールドは、Address ビジネス・オブジェクトに統合されている場合があります。このビジネス・オブジェクトは、Contact、Customer、Contract のいずれかのエンティティーのアドレスが変更された場合、Contact、Customer、Contract の各ビジネス・オブジェクトに対して更新することが必要です。この場合、Address ビジネス・オブジェクトは、データを使用するトップレベル・ビジネス・オブジェクトに包含されるのではなく、参照されます。

アプリケーション・エンティティーのバッチ処理: アプリケーション・エンティティーの作成に関連したバッチ処理はありますか?

いくつかのアプリケーションでは、バッチ処理によってエンティティーにデータを追加できます。例として、データ入力オペレーターは新規の顧客をアプリケーション・データベースに午前 11:00 に入力するが、午後 7:00 にバッチ・ジョブが実行されて未入力の値が入力されるまで顧客レコードは完成しない場合を考えます。

あるバッチ処理がアプリケーション・エンティティーに関連付けられており、この処理によって重要なデータまたは必要なデータが追加される場合は、ビジネス・オブジェクトの生成される時刻を指定する必要があります。例えば、次のようになります。

- バッチ処理によってイベント通知が生成される場合は、イベントによってコネクターが起動して、完全なビジネス・オブジェクトが IBM WebSphere Business Integration システムに送信されます。
- オペレーターの Save 操作によってイベント通知が発生する場合は、イベントによってコネクターが起動し、不完全なビジネス・オブジェクトが送信されます。

リアルタイムのデータ同期が必要だが、バッチ処理がバックグラウンドで実行されている場合、コネクター開発計画にはこのことを考慮に入れる必要があります。

ビジネス・オブジェクト開発のスコープの決定

定義が必要なビジネス・オブジェクトは何かを上位の見地から判断した場合は、ビジネス・オブジェクト開発の動詞サポートを次のようにして判断する必要があります。

1. 表 13 を使用して、コネクターがサポートするビジネス・オブジェクトと動詞の組み合わせごとに、動詞スコープの要約を作成する。
2. 完成したスコープの要約を使用して、各ビジネス・オブジェクトについての情報を収集する。

表 13. ビジネス・オブジェクト動詞のスコープ化のまとめ

ビジネス・オブジェクト名	必須の要求動詞 (要求処理)	必須の配信動詞 (アプリケーション・イベント通知)
オブジェクト 1	O Create O Update O Delete	O Create O Update O Delete
オブジェクト 2	O Create O Update O Delete	O Create O Update O Delete
オブジェクト *n*	O Create O Update O Delete	O Create O Update O Delete

重要: 大半のコネクターでは、ビジネス・オブジェクトごとに Retrieve 動詞をサポートする必要があります。したがって、Retrieve は表 13 には含まれていません。

メタデータ主導型の設計を評価するためのサポート

ビジネス・オブジェクト定義には、ビジネス・オブジェクトの構造と属性以外に、アプリケーション固有の情報を登録できます。ここには、アプリケーション内部でのビジネス・オブジェクトの表現方法に関する処理命令や情報を定義できます。このような情報のことをメタデータ といいます。

メタデータには、コネクターとアプリケーションとの対話時にコネクターが必要とするすべての情報を登録できます。例えば、表ベースのアプリケーションに対するビジネス・オブジェクト定義にアプリケーションの表名と列名を持つメタデータが

登録されている場合、コネクターはこの情報を使用して要求されたデータを取得できるので、アプリケーションの列名をコネクターでエンコードする必要はありません。コネクターは、サポートしているビジネス・オブジェクト定義に実行時にアクセスできるので、ビジネス・オブジェクト定義に登録されているメタデータを使用して、特定のビジネス・オブジェクトを処理する方法を動的に決定できます。

アプリケーションとそのプログラミング・インターフェース (API) によって異なりますが、コネクターおよびそのビジネス・オブジェクトは、表 14 に示すように、メタデータの使用をサポートする能力に基づいて設計される場合があります。

表 14. メタデータに対するコネクターのサポート

コネクターによるメタデータの使用	必要なビジネス・オブジェクト・ハンドラー	詳細情報
コネクターのビジネス・オブジェクト定義のメタデータに登録されている処理命令によって全面的に駆動される	メタデータ主導型の汎用ビジネス・オブジェクト・ハンドラー 1 つ	53 ページの『メタデータ主導型のコネクター』
コネクターのビジネス・オブジェクト定義のメタデータによって部分的に駆動される	部分的にメタデータ主導型のビジネス・オブジェクト・ハンドラー 1 つ	55 ページの『部分的にメタデータ主導型のコネクター』
メタデータを使用できない	メタデータを使用しないビジネス・オブジェクトごとに個別のビジネス・オブジェクト・ハンドラー	56 ページの『メタデータを使用しないコネクター』

いくつかのアプリケーション・インターフェースには、コネクターやビジネス・オブジェクトの設計時にメタデータの使用制限という制約がありますが、できるだけメタデータ主導型になるようにコネクターを開発するという目標には、それに見合う価値があります。表 14 に記載されている方法の利点と欠点については、以下に説明します。

メタデータ主導型のコネクター

メタデータ主導型の設計をサポートできるようにするには、アプリケーション内部のオブジェクトのうち、何を処理の対象にするかをアプリケーションの API によって指定できる必要があります。通常、このことは、ビジネス・オブジェクトのメタデータを使用すると、処理の対象となるアプリケーション・エンティティーと、対象のビジネス・オブジェクトの値としての属性データに関する情報を準備できることを意味しています。この結果、*メタデータ主導型のコネクター* は、ビジネス・オブジェクトの値とメタデータ (ビジネス・オブジェクト定義に格納されているアプリケーション固有の情報) を使用できるので、適切なアプリケーション関数呼び出しまたは SQL ステートメントを作成することによって、このエンティティーにアクセスできます。この関数呼び出しでは、コネクターによる処理の対象となるビジネス・オブジェクトおよび動詞に対して、アプリケーション内部で必要な変更が加えられます。

メタデータ主導型のコネクターには、フォーム、表、オブジェクトのいずれかに基づいたアプリケーションが適しています。例えば、フォーム・ベースのアプリケーションは、名前付きのフォームで構成されています。フォーム・ベースのアプリケーションとのプログラム化された対話の構成は、フォームのオープン、フォーム上のフィールドの読み取りまたは書き込み、フォームの保存または消去となります。

このようなアプリケーションのコネクターは、このコネクターがサポートしているビジネス・オブジェクト定義によって直接駆動できます。

メタデータ主導型のコネクターの主な利点は、コネクターが 1 つの汎用ビジネス・オブジェクト・ハンドラーを使用できることにより、すべての ビジネス・オブジェクトに対応できることです。この方法では、ビジネス・オブジェクト定義の中に、ビジネス・オブジェクトを処理するためにコネクターが必要とするすべての 情報が登録されています。ビジネス・オブジェクト自体にはアプリケーション固有の情報が格納されているので、コネクターは、コネクターのソース・コードを変更する必要なく、新規または変更済みのビジネス・オブジェクトを処理できます。コネクターは、*メタデータ主導型のビジネス・オブジェクト・ハンドラー* を 1 つ使用して、一般的な手法で記述できます。このハンドラーには、特定のビジネス・オブジェクトを処理するためのハードコーディングされたロジックは*存在しません*。

注: ビジネス・オブジェクト名には、コネクターに対して意味のある値を付けないでください。コネクターは、名前が異なり、構造、データ、アプリケーション固有の情報が同じである 2 つのビジネス・オブジェクトを、まったく同様に処理します。

WebSphere InterChange Server

図 16 には、アプリケーション固有のビジネス・オブジェクトと、メタデータ主導型のビジネス・オブジェクト・ハンドラーを備えているコネクターを示します。App_Order ビジネス・オブジェクトのアプリケーション固有の情報に登録されている処理命令により、ビジネス・オブジェクトの処理方法がコネクターに伝達されます。

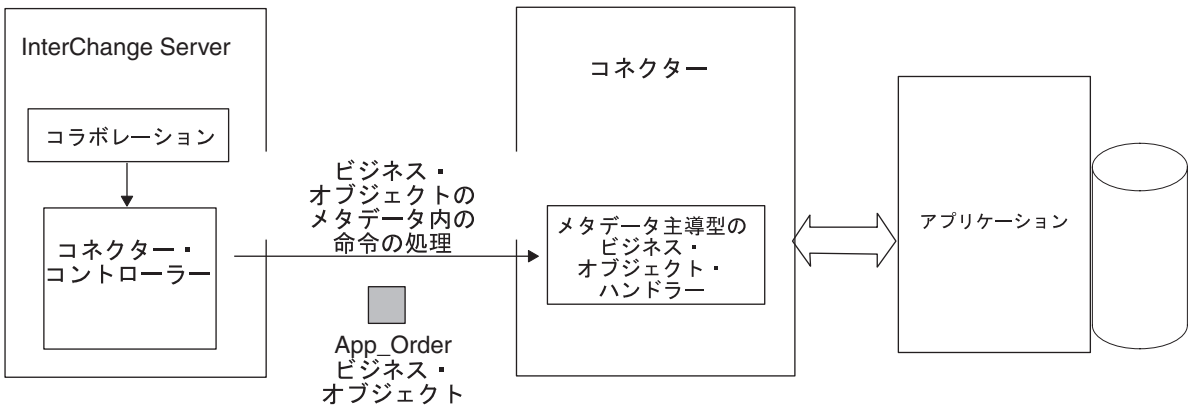

図 16. ビジネス・オブジェクトのメタデータの使用による処理命令への対応

メタデータ主導型のコネクターは、そのアプリケーション固有ビジネス・オブジェクトから処理命令を派生させるので、ビジネス・オブジェクトの設計では、この種の処理を念頭に置く必要があります。コネクターとビジネス・オブジェクトの設計にこの手法を取り入れると柔軟性や拡張の容易性が向上しますが、この手法では、設計段階での計画を念入りに行うことが要求されます。コネクターがビジネス・オ

ブジェクトのメタデータを処理の対象とするよう設計されている場合、ビジネス・オブジェクト自体を変更しても、これに対応する変更をコネクターに行う必要はありません。

メタデータ主導型のビジネス・オブジェクト・ハンドラーの詳細については、 92 ページの『メタデータ主導型のビジネス・オブジェクト・ハンドラーの実装』を参照してください。

部分的にメタデータ主導型のコネクター

IBM では、コネクターおよびアプリケーション固有のビジネス・オブジェクト定義を設計する場合、メタデータ方式をお勧めしています。ただし、一部のアプリケーションは、この手法に適していない場合があります。アプリケーションの各エンティティーに固有のアプリケーション API では、メタデータ主導型のコネクターを作成することがさらに困難になります。多くの場合、メソッドの名前や渡されるデータに違いがあることだけが問題なのではなく、オブジェクト間での呼び出し自体に、その構造上、何らかの違いがあることが問題となります。

メタデータに実際の処理命令が記述されていない場合でも、このメタデータを使用してコネクターを駆動できることがあります。この*部分的にメタデータ主導型のコネクター* は、ビジネス・オブジェクト定義または属性のメタデータを使用して、実行する処理の内容を決定しやすくすることができます。例えば、大量のビジネス・ロジックがユーザー・インターフェースに組み込まれているアプリケーションでは、コネクターなどの外部プログラムとそのデータベースとの間で情報を交換する方法に制約があることがあります。場合によっては、アプリケーション環境およびアプリケーション・プログラミング・インターフェースによって、アプリケーションに拡張機能を付加することが必要になります。アプリケーションにオブジェクト固有のモジュールを付加して、ビジネス・オブジェクトごとに処理することが必要な場合があります。アプリケーションのビジネス・ロジックが適用され、かつそれがバイパスされることのないように、アプリケーションには、そのアプリケーション環境とインターフェースを使用することが要求される場合があります。

この場合、ビジネス・オブジェクトおよび属性のアプリケーション固有の情報には、コネクター用のメタデータが格納されている可能性があります。このメタデータは、ビジネス・オブジェクトの操作をアプリケーション内部で実行するのに必要なモジュールや API 呼び出しの名前を指定します。この場合でも、コネクターは 1 つのビジネス・オブジェクト・ハンドラーによって実装できますが、このメタデータには処理命令が記述されていないため、このハンドラーは*部分的にメタデータ主導型のビジネス・オブジェクト・ハンドラー* になります。

図 17 には、コネクターからの要求を処理する役割を果たしているアプリケーションの拡張機能が示されています。この拡張機能には、コネクターによってサポートされているビジネス・オブジェクトごとに別個のモジュールが格納されています。

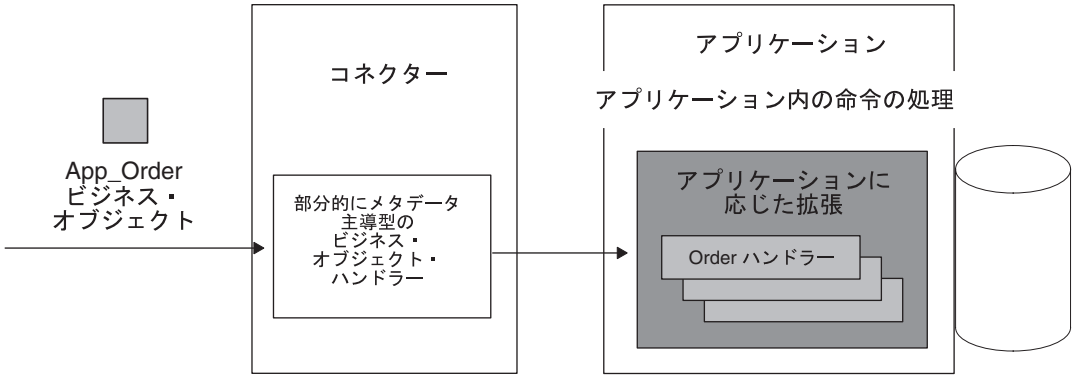

図 17. アプリケーション内部でのアプリケーション固有の処理

部分的にメタデータ主導型のコネクターの利点は、使用するビジネス・オブジェクト・ハンドラーが 1 つのみで済むことです。ただし、メタデータ主導型のコネクターとは異なり、コネクターに対して新しいビジネス・オブジェクトを作成する場合には、そのためのコーディング作業が発生します。この場合、新しいオブジェクト関数を記述して、アプリケーションに追加する必要がありますが、コネクターの再コーディングや再コンパイルを行う必要はありません。

メタデータを使用しないコネクター

アプリケーションの API が、処理の対象となるアプリケーション内部のエンティティーを指定する機能を*提供しない* 場合、コネクターは、メタデータを使用して 1 つのビジネス・オブジェクト・ハンドラーをサポートすることはできません。その代わりに、API は、コネクターがサポートしているビジネス・オブジェクトごとに*複数のビジネス・オブジェクト・ハンドラー* を用意する必要があります。この手法では、各ビジネス・オブジェクト・ハンドラーに、特定のビジネス・オブジェクトを処理するための固有のロジックおよびコードが格納されています。

図 18 では、コネクターに複数のオブジェクト固有ビジネス・オブジェクト・ハンドラーがあります。コネクターは、ビジネス・オブジェクトを受け取ると、このビジネス・オブジェクトに適したビジネス・オブジェクト・ハンドラーを呼び出します。

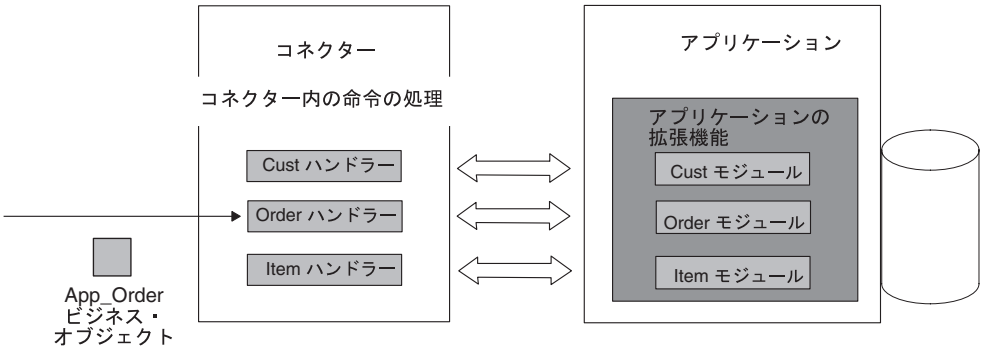

図 18. コネクターでのアプリケーション固有の処理

この非メタデータ方式の欠点は、ビジネス・オブジェクトを変更したり新規のビジネス・オブジェクトを追加したりした場合、この種のコネクターでは、新規または変更済みのビジネス・オブジェクトを処理するために再コーディングが必要になることです。

イベント通知

IBM WebSphere Business Integration システムはイベント・ドリブン・システムなので、コネクターには、アプリケーション内部で発生するイベントを検出して記録する方法がいくつか必要になります。アプリケーションについて調べる場合は、このアプリケーションが、アプリケーション・データに対する変更をコネクターに通知できるイベント通知機構を備えているかどうかを確認してください。

イベント通知機構は、通常、内部のアプリケーション・イベントのコネクターへの通知を可能にする一連のプロセスで構成されます。イベント・レコードには、イベントの型、ビジネス・オブジェクト名および動詞 (Customer や Create など)、コネクターが関連データを取得するために必要なデータ・キーなどがあります。

さらに、イベント通知の方針には、イベント・レコードとこれに対応するイベント・データとのデータ保全性を確保するために必要な機構を取り入れる必要があります。言い換えると、イベントに対するすべての 必須データ・トランザクションが正常に完了するまで、イベント通知は実行されません。

イベント通知機構の設計は、アプリケーションがアプリケーション・イベントを通知する範囲と、アプリケーションによってクライアントがイベント・データを取得できる範囲に応じて変化します。アプリケーションがイベント通知インターフェース (API など) を備えている場合、IBM では、これを使用してイベント通知機構を実装することをお勧めします。API を使用することにより、コネクターとアプリケーションとの対話が、アプリケーションのビジネス・ロジックを順守するようになります。アプリケーションがイベント通知機構を備えている場合は、以下のトピックや質問を活用して、詳細な情報を収集してください。

イベント通知の詳細度

- アプリケーションのイベント通知機構は、ビジネス・オブジェクトや動詞を個別に設定するためのイベントについて十分詳細な情報を提供していますか? 情報が不十分な場合、イベント通知コンポーネントを構成して、この詳細度を提示できるようにすることができますか?

  例えば、新しいレコードを追加したり既存の顧客を更新したりした場合は、イベント通知機構が操作のタイプ (Create 操作や Update 操作) に関する情報を提示できるかどうかを調べます。コネクターがデルタ操作をサポートしている場合は、イベント通知機構が、変更されたサブオブジェクトや属性の正確な情報を提示できるかどうかを確認してください。

ビジネス・ロジックのイベント通知サポート

- イベント通知は、ビジネス要件を適正にサポートするレベルで実行されていますか? 言い換えると、イベント通知機構には、理論的にはアプリケーションのビジネス・ロジックへのサポートが組み込まれています。

プロジェクト計画の中でイベント通知機構について説明してください。イベント通知機構が存在しない場合は、アプリケーション・データの変更を検出するために利用できる代替手段を調べてください。例えば、リレーショナル・データベースの表にデータベース・トリガーをセットアップすれば、イベント通知を実現できる場合があります。あるいは、アプリケーションには、データベースのすべての変更内容をエクスポートするバッチ・エクスポート機能が用意されていることがあります。この機能のエクスポート先は、コネクターによってアプリケーション・イベントに関する情報を抽出する元となるファイルになっています。

注: イベント通知機構のインプリメント段階の詳細については、 131 ページの『イベント通知機構の概要』を参照してください。

オペレーティング・システム間での通信

アプリケーションとコネクターとの間の通信は、コネクター全体の設計の主要なコンポーネントです。アプリケーションが InterChange Server およびコネクターとは異なるオペレーティング・システムで動作する場合は、コネクターがアプリケーションにアクセスできるようにするための機構が存在することを確認する必要があります。

アプリケーションに API が用意されている場合は、この API がアプリケーションのオペレーティング・システムとコネクターのオペレーティング・システムとの間の通信を管理するかどうかを確認します。例えば、アプリケーションが UNIX 上で動作し、コネクターと InterChange Server が Windows 2000 上で動作する場合、コネクターとアプリケーションは、アプリケーションの API によって、オペレーティング・システムをまたがって通信できます。

図 19 は、Windows 2000 上で動作する ODBC コネクターと、UNIX 上で動作する ODBC ベースのアプリケーション間の通信機構の例を示しています。コネクターは、動的な SQL ステートメントを作成し、ODBC API を介してそれらのステートメントを実行します。ODBC ドライバーは、コネクターがアプリケーション・データベースとの接続を確立して、ODBC SQL ステートメントによってデータベースにアクセスできるようにします。

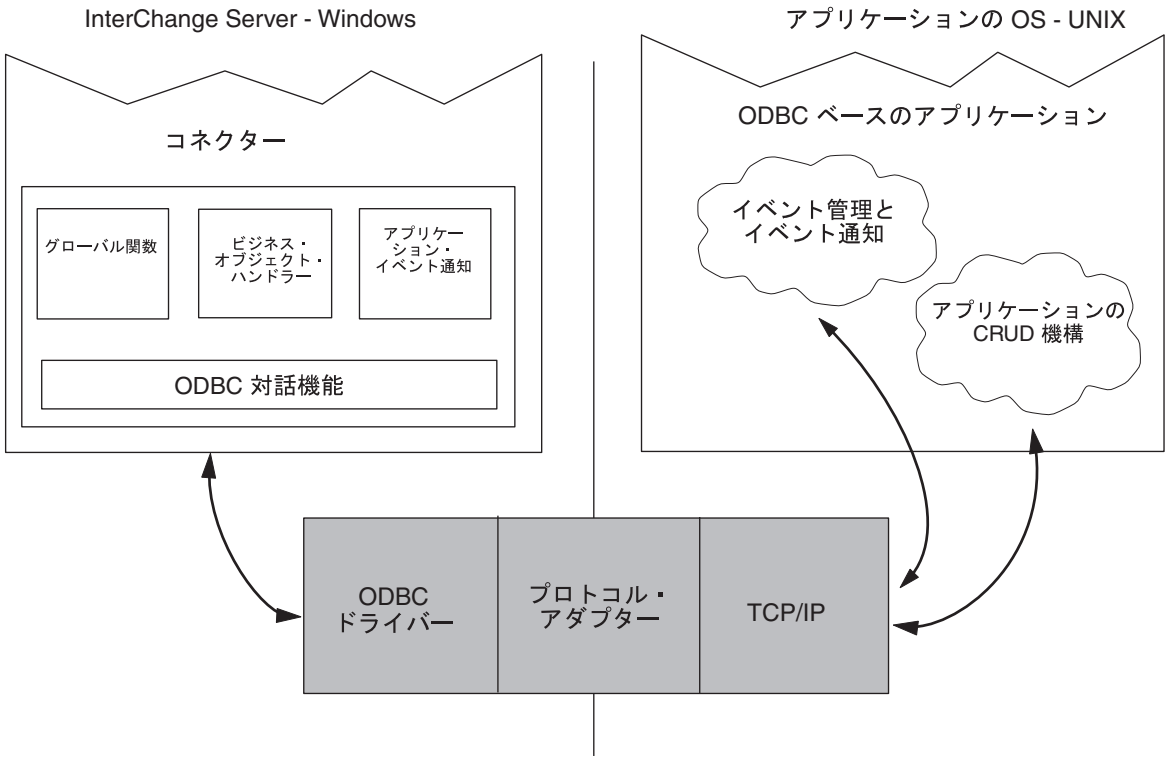

図 19. Windows/UNIX 間の通信の例

計画に関する質問のまとめ

次の表には、この章で説明した計画に関する質問のまとめを掲載しています。この表は、アプリケーションに関する情報を収集するためのワークシートとして使用できます。情報を収集したら、プロジェクトの計画、設計、開発のいずれかの段階に役立つ資料を数部入手します。

1. アプリケーションの理解
   - アプリケーションのオペレーティング・システムは何ですか?
   - アプリケーションの記述に使用されたプログラム言語は何ですか?
   - アプリケーションの実行アーキテクチャーは何ですか?
   - アプリケーション・データ用の中央データベースはありますか? データベースの種類は何ですか?
   - アプリケーションまたはそのデータベースは複数のサーバーにまたがって分散していますか?
2. コネクターの方向性の特定
   - コネクターが必要な処理は、データの送信、受信、その両方のどれですか?
3. アプリケーション固有のビジネス・オブジェクトの特定
   - アプリケーション・エンティティーには、包含エンティティーがありますか?
   - 型は同じだが、アプリケーションでの物理表現は異なるアプリケーション・ビジネス・エンティティーはありますか?
   - データベース内の複数の位置に存在するが、同じ論理エンティティーに対応するアプリケーション・エンティティーはありますか?
   - アプリケーション・エンティティーの作成に関連したバッチ処理はありますか?
4. アプリケーション・データ対話インターフェースの検討
   - このアプリケーションと統合するための過去の作業実績はこのほかにありますか?
     - 統合の目的は何ですか?
     - 統合済みアプリケーションは、情報を変更または取得するインターフェースを使用しますか?
     - この統合手法が、アプリケーションで生成されたイベントを処理できる場合、イベント処理を起動するために使用する機構は何ですか?
     - コネクターは既存の統合アプリケーションに置き換わりますか?
   - アプリケーション・データはほかのアプリケーションと共用しますか?
     - ほかのアプリケーションは、このアプリケーションのデータに対して、作成、取得、更新、または削除を行いますか?
     - このデータへのアクセス回数を増やすためにほかのアプリケーションが使用している機構は何ですか?
     - ほかのアプリケーションが使用しているオブジェクト固有のビジネス・ロジックはありますか?
   - コネクターがこのアプリケーションと通信するときに使用できる機構はありますか?
     - API は、Create、Retrieve、Update、Delete の各操作にアクセスできますか?
     - API はすべてのデータ・エンティティー属性にアクセスできますか?
     - API は、アプリケーションにアクセスしてイベントを検出できますか?
     - API の実装に不整合はありますか?
     - API のトランザクション動作を説明してください。
     - API はメタデータ設計に適していますか?
     - API はアプリケーションのビジネス・ルールを適用しますか?
   - 重複データや無効データの消去に使用するバッチ・クリーンアップ・プログラムまたはマージ・プログラムはありますか?
5. イベント管理機構とイベント通知機構の検討
   - イベント管理機構について説明してください。
   - この機構には、オブジェクトと動詞を別個に設定するために必要な細分度が用意されていますか?
   - イベント通知は、アプリケーションのビジネス・ロジックをサポートできるレベルで実行されていますか?
6. オペレーティング・システム間での通信の検討
   - API は、アプリケーションのオペレーティング・システムとコネクターのオペレーティング・システムとの間の通信機構を管理しますか?
   - 管理しない場合、複数のオペレーティング・システム間の通信を管理できる機構はありますか?

図 20. 計画に関する質問のまとめ

検討結果の評価

この章に記載された質問に対する答えを集めていくと、アプリケーション・データのエンティティー、ビジネス・オブジェクト処理、およびイベント管理に関する重要な情報が得られます。こうした検討結果が、コネクターの上位アーキテクチャーの基礎になります。

コネクターのサポート対象となるエンティティーの内容を決定し、データベースとの対話やイベント通知に関するアプリケーションの機能を検討すると、コネクター開発プロジェクトのスコープが明確に理解できる状態になります。この時点に到達したら、コネクター開発の次の段階である、アプリケーション固有のビジネス・オブジェクトの定義とコネクターのコーディングに引き続き移行できます。

図 21 に、サンプル・コネクターに関する情報を要約した内容の一部を示します。図 22 には、ODBC ベースのコネクターの上位アーキテクチャー・ダイアグラムを示します。

1. アプリケーションの理解
   - アプリケーションは UNIX 上で動作する。
   - 使用しているプログラム言語は Microsoft MFC ライブラリーを備えた Visual C++ です。
   - アプリケーションはクライアント/サーバー型です。
   - アプリケーションには中央データベースがあります。タイプは RDMS です。
   - アプリケーションは配布されていません。
2. コネクターの方向性の特定
   - コネクターは双方向型にします。
3. アプリケーション固有のビジネス・オブジェクトの特定
   - ビジネス・オブジェクトには、オブジェクトが格納されています。格納されているビジネス・オブジェクトは、次のとおりです。
     - Customer ”Address ”Site Use および Site Profile
     - Item ”Status
     - Contact ”*n* 個の Phone および *n* 個の Roles
   - アプリケーションのビジネス・エンティティーには、アプリケーション内部に異なる物理表現がありません。
   - アプリケーションのエンティティーは、データベース内の複数の位置には存在しません。
   - これらのオブジェクトの作成と関連付けられているバッチ処理はありません。
4. アプリケーション・データ対話インターフェースの検討
   - このアプリケーションと統合するための過去の作業実績はありません。
   - アプリケーション・データはほかのアプリケーションとは共用されていません。
   - アプリケーションは OpenProduct API を備えています。
     - OpenProduct では、Creates および Updates は使用できるが、Retrieves および Deletes は使用できません。
     - API はすべてのデータ・エンティティー属性にアクセスできます。
     - API はアプリケーションにアクセスしてイベントを検出できます。イベント表を作成して、指定の間隔でイベントのポーリングを実行できます。
     - API に不整合はありません。
     - API には、バッチ・インターフェースがあります。
     - アプリケーションは表ベースで、API はメタデータ設計に適しています。
     - ...

図 21. 結果サンプル

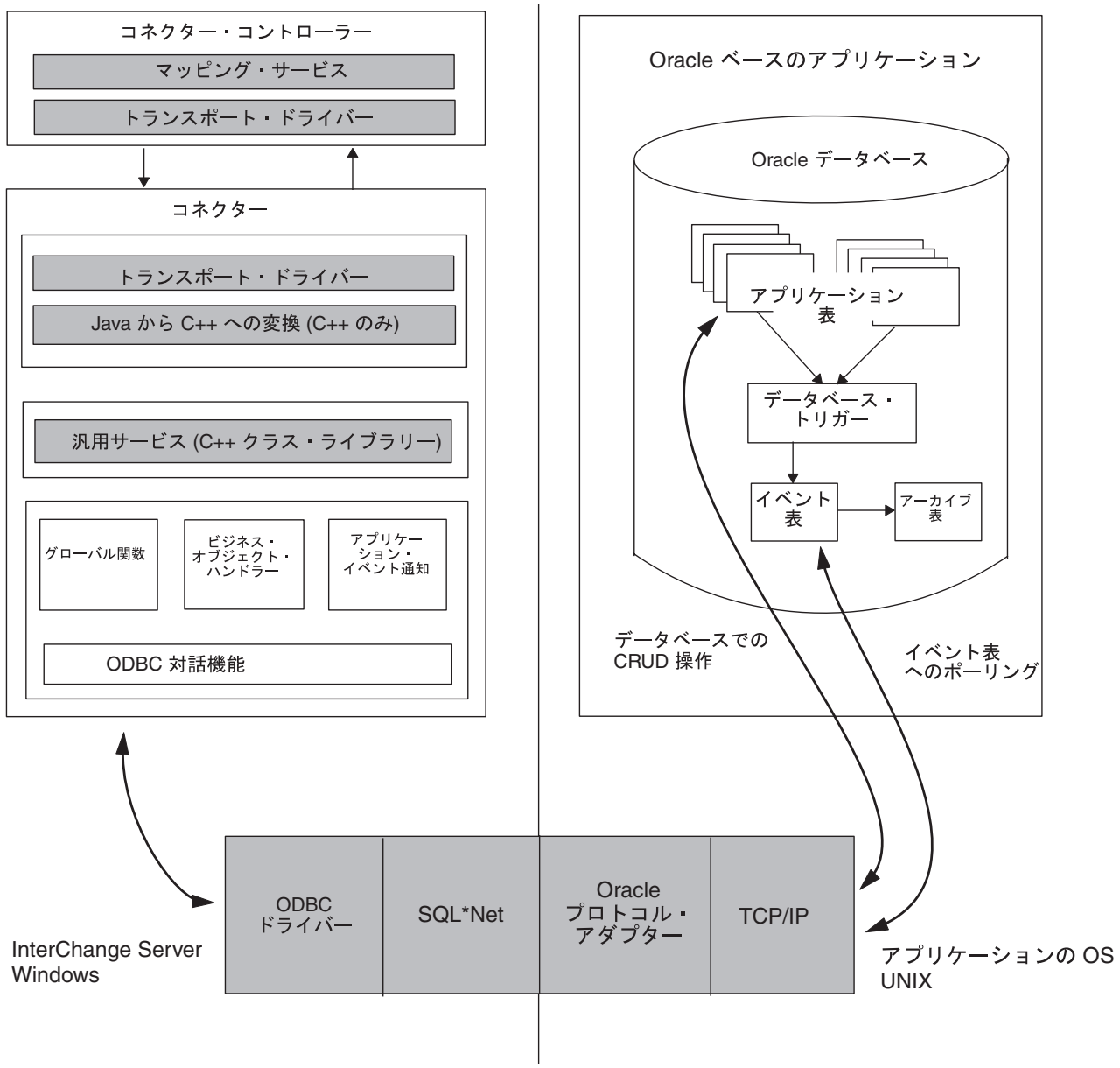

図 22. ODBC ベースのコネクター・アーキテクチャーのサンプル

国際化対応コネクター

国際化対応コネクター とは、特定のロケールに合わせてカスタマイズできるように記述されたコネクターのことです。*ロケール* は、エンド・ユーザーの特定の国、言語、または地域に固有のデータを処理する方法に関する情報をまとめたユーザー環境設定項目です。ロケールは、通常はオペレーティング・システムの一部としてインストールされます。ロケール依存のデータを扱えるコネクターを作成することを、コネクターの*国際化対応* (I18N) といいます。国際化対応のコネクターを特定のロケールに合うように調整することを、コネクターの*ローカリゼーション* (L10N) といいます。

このセクションでは、国際化対応コネクターについて以下の内容を説明します。

- 『ロケールとは』
- 『国際化対応コネクターの設計上の考慮事項』

ロケールとは

ロケール は、ユーザー環境に関する以下の情報を提供します。

- 言語および国 (または地域) ごとの国/地域別情報:
  - データ・フォーマット:
    - 日付: 曜日および月の名前とその省略名、および日付の構成 (日付の分離文字を含む) を定義します。
    - 数値: 3 桁ごとの区切り記号と小数点記号、およびこれらの記号を数値中のどこに配置するかを定義します。
    - 時刻: 12 時間表記の標識 (AM および PM 標識など)、および時刻の構成を定義します。
    - 通貨値: 数値と通貨記号、およびこれらの記号を通貨値の中のどこに配置するかを定義します。
  - 照合順序: 特定の文字コード・セットおよび言語のデータをソートする方法です。
  - ストリング処理には、文字 (大文字および小文字) の比較、サブストリング、連結などのタスクがあります。
- 文字エンコード — 文字 (英字) を文字コード・セットの数値にマッピングします。例えば、ASCII 文字コード・セットでは文字「A」は 65 にエンコードされ、EBCIDIC 文字セットではこの文字は 43 にエンコードされます。*文字コード・セット* には、1 つ以上の言語のすべての文字のエンコード方式が含まれます。

ロケール名は次のような形式になります。

ll_TT.codeset

ここで、*ll* は 2 文字の言語コード (通常は小文字)、*TT* は 2 文字の国および地域コード (通常は大文字)、*codeset* は関連する文字コード・セットの名前を表します。ロケール名の *codeset* 部分は、たいていの場合はオプションです。ロケールは、通常はオペレーティング・システム・インストールの一部としてインストールされます。

国際化対応コネクターの設計上の考慮事項

このセクションでは、コネクターを国際化対応するための設計上の考慮事項を、以下のカテゴリーに分類します。

- 『ロケール依存の設計原則』
- 67 ページの『文字エンコードの設計原則』

ロケール依存の設計原則

コネクターを国際化対応するには、コネクターをロケール依存になるようにコーディングする必要があります。つまり、コネクターの動作がロケール設定値を考慮に入れたうえで、そのロケールに合った適切なタスクを実行しなければなりません。

例えば、英語を使用するロケールの場合、そのエラー・メッセージを英語のメッセージ・ファイルから取得する必要があります。WebSphere Business Integration Adapters 製品で提供されているコネクター・フレームワークは、国際化対応バージョンです。開発対象となるコネクターの国際化対応 (I18N) を完了するには、必ず、国際化対応されたアプリケーション固有コンポーネントを使用する必要があります。

表 15 に、国際化対応アプリケーションに固有のコンポーネントが準拠する必要のあるロケール依存の設計原則をリストします。

表 15. アプリケーション固有コンポーネントのロケール依存の設計原則

設計原則	詳細情報
エラー・メッセージ、状況メッセージ、およびトレース・メッセージのそれぞれのテキストを、メッセージ・ファイル内のアプリケーション固有コンポーネントから分離して、ロケールの言語に変換する必要があります。	『テキスト・ストリング』
コネクターの実行時に、ビジネス・オブジェクトのロケールを維持する必要があります。	66 ページの『ビジネス・オブジェクトのロケール』
コネクター構成プロパティーのプロパティーは、マルチバイト文字が含まれる場合に、これらの文字を組み込むよう設定する必要があります。	67 ページの『コネクター構成プロパティー』
その他のロケール固有のタスクを検討する必要があります。	67 ページの『その他のロケール依存タスク』

テキスト・ストリング: テキスト・ストリングを取得する必要がある場合は、コネクターのコードにテキスト・ストリングをハード・コーディングするよりも、外部のメッセージ・ファイルを参照するようにコネクターを設計する方が効率的なプログラミング方法であるといえます。コネクターは、テキスト・メッセージを生成する必要がある場合、メッセージ・ファイルにある該当のメッセージをそのメッセージ番号で取得します。すべてのメッセージが単一のメッセージ・ファイルに収集されると、テキストを該当の言語に翻訳することによって、このファイルをローカライズすることができます。

このセクションでは、テキスト・ストリングを国際化対応にする方法について以下の内容を説明します。

- 『ロギングおよびトレースの処理』
- 65 ページの『各種ストリングの処理』

ロギングおよびトレースの処理: ロギングおよびトレースを国際化対応にするには、これらの操作がすべて、メッセージ・ファイルを使用してテキスト・メッセージを生成するようにします。メッセージ・ファイルにメッセージ・ストリングを書き込むことによって、各メッセージに固有 ID を割り当てることができます。表 16 に、メッセージ・ファイルを使用する操作のタイプ、およびアプリケーション固有のコンポーネントがメッセージ・ファイルからメッセージを取得する際に使用する、`CWConnectorUtil` クラスの関連する Java コネクター・ライブラリーメソッドをリストします。

表 16. メッセージ・ファイルからメッセージをロギングおよびトレースするためのメソッド

メッセージ・ファイルの操作	コネクターのライブラリー・メソッド
ロギング	`generateAndLogMsg()`
トレース	`generateAndTraceMsg()` または `traceWrite()`

ログ・メッセージは、顧客のロケールの言語で表示する必要があります。そのため、ログ・メッセージは必ずコネクター・メッセージ・ファイルに分離し、`generateAndLogMsg()` メソッドを使用して取り出す必要があります。

トレース・メッセージは製品のデバッグ処理に使用されるものであるため、多くの場合、これらのメッセージは顧客のロケールの言語で表示する必要はありません。そのため、トレース・メッセージをメッセージ・ファイルに含めるかどうかは、以下のように、開発者が任意に決定することができます。

- 英語圏外のユーザーがトレース・メッセージを参照する必要がある場合、それらのメッセージは国際化対応にする必要があります。そのため、トレース・メッセージをメッセージ・ファイルに入れて、`generateMsg()` メソッドで抽出する必要があります。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。`generateMsg()` メソッドは、`traceWrite()` 用のメッセージ・ストリングを生成します。このメソッドによってメッセージ・ファイルから事前定義済みトレース・メッセージが取得され、テキストの書式が設定された後、生成されたメッセージ・ストリングが戻されます。
- 英語圏のユーザーのみがトレース・メッセージを参照する必要がある場合、それらのメッセージを国際化対応にする必要はありません。そのため、(英語の) トレース・メッセージを直接 `traceWrite()` の呼び出しに組み込むことができます。`generateMsg()` メソッドを使用する必要はありません。

  ただし、トレース・メッセージをメッセージ・ファイルに保管すると、メッセージの取得と保守が簡単になります。

各種ストリングの処理: 表 16 で示したメッセージ・ファイル操作の処理に加え、国際化対応したコネクターに各種ハードコーディングされたストリングを含めることはできない という制約があります。これらのストリングは、先ほどと同様に、メッセージ・ファイルに分離する必要があります。表 17 に、アプリケーション固有のコンポーネントがメッセージ・ファイルからメッセージを取得する際に使用するメソッドを示します。

表 17. メッセージ・ファイルからメッセージを取得するためのメソッド

コネクター・ライブラリー・クラス	メソッド
`CWConnectorUtil`	`generateMsg()`

ハードコーディングされたストリングを国際化対応にするには、以下のステップを実行します。

- ハードコーディングされたストリング用のコネクター・メッセージ・ファイル内に、固有の番号が付けられたメッセージを生成します。

注: メッセージ・ファイルには、分離されたストリングに関する説明をオプションで含めることもできます。この説明には、ストリングが使用されるメソッド名も含めることができます。この情報により、ソースの位置を追跡し、必要な変更を加えることができるようになります。

- アプリケーション固有のコンポーネントで、generateMsg() メソッドを使用して、分離されたストリングをそのメッセージ番号で指定します。

例えば、アプリケーション固有のコンポーネントのコード行に、以下のようなハードコーディングされたストリングが含まれているとします。

```
********Before updating the event status********
```

ハードコーディングされたストリングをコネクター・コードから分離するには、以下のようにメッセージ・ファイル内にメッセージを作成し、そのメッセージに固有のメッセージ番号 (100) を割り当てます。

```
100
********Before updating the event status********
[EXPL]
Hardcoded message in pollForEvents()
```

アプリケーション固有のコンポーネントは、分離されたストリング (メッセージ 100) をメッセージ・ファイルから取り出し、ハードコーディングされたストリングをその取得されたストリングで置き換えます。

```
//retrieve the message numbered '    100'
String msg100 = generateMsg(100, CWConnectorLogAndTrace.XRD_INFO,
    CWConnectorLogAndTrace.CONNECTOR_MESSAGE_FILE, 0);
MyClassObject.formatMsg(msg100);  //send retrieved message to a custom method
```

メッセージ・ファイルの詳しい使用方法については、 159 ページの『第 6 章 メッセージ・ロギング』を参照してください。

ビジネス・オブジェクトのロケール: コネクターは、アプリケーション・データからアプリケーション固有のビジネス・オブジェクトへの変換の際、ロケール依存の処理 (例えば、データ・フォーマットの変換など) を実行しなければならない場合もあります。コネクターでビジネス・オブジェクトが処理されるときには、次の 2 通りの異なるロケール設定が行われます。

- コネクターは、実行に使用しているコネクター・フレームワークからコネクター・フレームワーク・ロケール と呼ばれるロケールを継承します。コネクター・フレームワーク・ロケールにより、ロギングおよび例外のためにコネクターが使用するテキスト・メッセージのロケールが決定されます。
- コネクターは、処理対象のビジネス・オブジェクトと関連したロケールにもアクセスすることができます。このビジネス・オブジェクト・ロケール により、ビジネス・オブジェクトに含まれるデータと関連したロケールが識別されます。

表 18 に、コネクターがコネクター・フレームワークに関連付けられたロケールを取得するために使用するメソッドを示します。

表 18. コネクター・フレームワークのロケールを取得するためのメソッド

コネクター・ライブラリー・クラス	メソッド
CWConnectorUtil	getGlobalLocale()

ビジネス・オブジェクトを作成すると、そのビジネス・オブジェクトはそのデータにロケールを関連付けることができます。コネクターは、次のいずれかの方法で、このビジネス・オブジェクト・ロケールにアクセスすることができます。

- このビジネス・オブジェクト・ロケールの名前を取得するには、CWConnectorBusObj クラスで定義されている getLocale() メソッドを使用します。CWConnectorBusObj クラスでは setLocale() メソッドも提供されます。
- ビジネス・オブジェクトにロケールを関連付けるには、 createBusObj() メソッドを使用します。このメソッド は、CWConnectorUtil クラスで定義されています。

コネクター構成プロパティー: 79 ページの『コネクター構成プロパティー値の使用』で記述されているように、アプリケーション固有のコンポーネントでは、実行をカスタマイズするために以下の 2 つの構成プロパティー・タイプを使用できます。

- 標準の構成プロパティーはすべてのコネクターで使用できます。
- コネクター固有の構成プロパティーは、そのプロパティーが定義されている特定のコネクターに固有のものです。

すべてのコネクター構成プロパティー名で、米国英語 (en_US) ロケールに関連したコード・セットで定義されている文字のみ を使用する必要があります。ただし、これらの構成プロパティーの値については、コネクター・フレームワークのロケールに関連したコード・セットの文字を含めることができます。

アプリケーション固有のコンポーネントは、 80 ページの『コネクター構成プロパティーの取得』に記述されているメソッドを使用して、構成プロパティーの値を取得します。これらのメソッドは、マルチバイト・コード・セットの文字を正しく処理します。ただし、ご使用のコネクターを確実に国際化対応にするためには、そのコードがこれらの構成プロパティー値を取得し、そのプロパティー値を正しく処理しなければなりません。アプリケーション固有のコンポーネントでは、構成プロパティー値が単一バイト文字でのみ構成されているとは限りません。

その他のロケール依存タスク: 国際化対応コネクターは、以下のロケール依存のタスクについても処理する必要があります。

- データのソートまたは照合。コラボレーションでは、ロケールの言語および国に対して適切な照合順序を使用する必要があります。
- ストリング処理 (比較、サブストリング、および大文字小文字など)。コラボレーションが実行するすべての処理が、必ずロケールの言語の文字に対して適切になるようにする必要があります。
- 日付、数値、および時刻のフォーマット。コラボレーションが実行するフォーマット設定が、必ずロケールに対して適切になるようにする必要があります。

文字エンコードの設計原則

あるコード・セットを使用するロケーションから別のコード・セットを使用するロケーションにデータを転送する場合、データの意味が保持されるような文字変換形式を実行する必要があります。 Java 仮想マシン (JVM) 内部の Java ランタイム環境は、Unicode のデータを表します。Unicode 文字セットは、ほとんどの既知の文字コード・セット (単一バイトおよびマルチバイトの両方) の文字のエンコードが含ま

れる汎用文字セットです。Unicode にはいくつかのエンコード・フォーマットがあります。ビジネス・インテグレーション・システム内で最も頻繁に使用されるエンコードを以下に示します。

- Universal multiple octet Coded Character Set: UCS-2

  UCS-2 エンコードは、2 バイト (オクテット) でエンコードされた Unicode 文字セットです。

- UCS Transformation Format、8 ビット形式: UTF-8

  UTF-8 エンコードは、UNIX 環境で Unicode 文字データを使用するために設計されたものです。UTF-8 ではすべての ASCII コード値 (0...127) がサポートされるので、ASCII コード値が別のコードに解釈されることはありません。各コード値は通常 1 バイト値、2 バイト値、または 3 バイト値で表します。

WebSphere Business Integration システム内のコンポーネントの大部分 (コネクター・フレームワークを含む) は、Java で記述されています。そのため、ほとんどのシステム・コンポーネント間でのデータ転送の際には、データが Unicode コード・セットでエンコードされるため、文字変換の必要はありません。

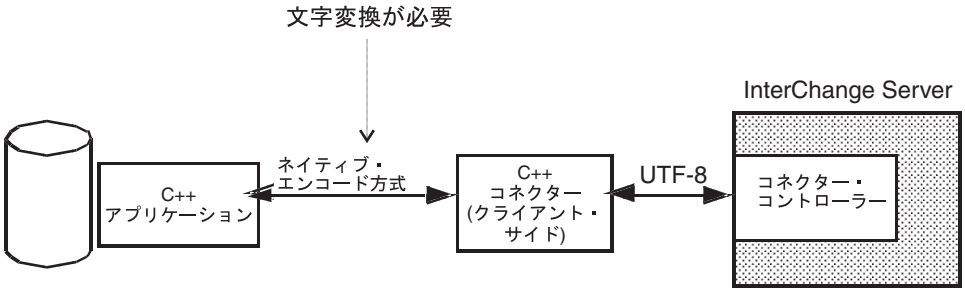

図 23. C++ コネクターでの文字エンコード

Java コネクターを動作させるには、Java アプリケーション (またはテクノロジー) が必要であるため、アプリケーション固有コンポーネントは、Unicode コード・セットのデータを扱える Java で記述されます。Java アプリケーション (またはテクノロジー) の場合、Unicode のデータも保持されるため、Java コネクターは、通常、アプリケーション固有のビジネス・オブジェクト用のアプリケーション・データに対して文字変換を実行する必要はありません。一部のデータが Unicode 以外の場合、Java 環境では UCS-2 とネイティブ・エンコード間の文字変換が自動的にサポートされます。ただし、アプリケーション・データに、他の外部ソース (オペレーティング・システム・ファイルなど) からのデータが組み込まれている場合は、Java コネクターによる文字変換の処理が必要になることもあります。図 24 に Java コネクターの文字エンコードを示します。

注: コネクターは、そのアプリケーションの文字エンコードを `CharacterEncoding` コネクター構成プロパティーから取得します。コネクターによる文字変換を実行する場合は、このコネクターのプロパティーを適切な値に設定するように、コネクターのエンド・ユーザーに必ず指示してください。

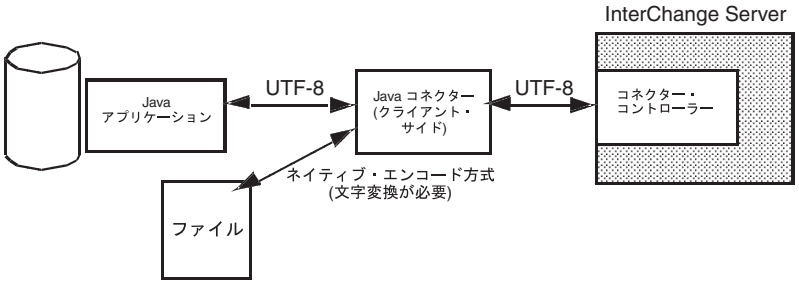

図 24. Java コネクターでの文字エンコード

実行時に文字エンコードを取得する場合にコネクターが使用できる Java コネクター・ライブラリー内のメソッドを、表 19 に示します。

表 19. コネクター・フレームワークの文字エンコードを取得するためのメソッド

コネクター・ライブラリー・クラス	メソッド
CWConnectorUtil	getGlobalEncoding()

Java String は UCS-2 エンコードです。そのため、コネクターは単純な変換を実行することにより、属性値 (Java String として表される)、デフォルト属性値、およびネイティブ・エンコードのアプリケーション固有の情報の取得および設定を可能にしています。

```
nativeEncodedAppSpecInfo = busObj.getAppText(attrName).getBytes(nativeEncoding);
```

注: コネクター構成プロパティーが String 値を持つ場合、この String 値は InterChange Server リポジトリーから生成される UCS-2 エンコードであるため、文字変換は必要ありません。

第 3 章 汎用コネクター機能の提供

この章では、コネクターのアプリケーション固有コンポーネントの初期化とセットアップを実行するコネクター・クラス を実装する方法について説明します。また、コネクターが実際に必要とする可能性のある基本的な一部の機能についても説明します。

注: アプリケーション固有コンポーネントのコーディングは、コネクター開発の全体作業の一部にすぎません。アプリケーション固有コンポーネントのコーディングを開始する前に、コネクター設計上の問題とともに、アプリケーション固有ビジネス・オブジェクトの設計も明確に理解しておく必要があります。設計上の問題を完全に理解しておくと、コーディング作業を手際よく順調に終了することができます。コネクターの設計については、41 ページの『第 2 章 コネクターの設計』を参照してください。

この章を構成するセクションは次のとおりです。

コネクターの実行

実行しているコネクターは、表 20 に要約されているタスクを実行します。

表 20. コネクター実行の手順

実行ステップ		詳細情報
1.	始動スクリプトを使用して、コネクターを起動し、コネクター・フレームワークとコネクターのアプリケーション固有コンポーネントを初期化する。	72 ページの『コネクターの始動』
2.	ポーリングがオンになっている場合は、コネクター・フレームワークが、`PollFrequency` コネクター構成プロパティーで定義されているインターバルで、`pollForEvents()` を呼び出す。	76 ページの『イベントのポーリング』
3.	コネクターが要求処理を実装している場合は、コネクターが受信する要求ビジネス・オブジェクトに関連するビジネス・オブジェクト・ハンドラーを呼び出す。	要求処理は、コネクターのビジネス・オブジェクト・ハンドラーの `doVerbFor()` メソッドによって実装されます。詳細については、91 ページの『第 4 章 要求処理』を参照してください。
4.	コネクターのシャットダウン時、コネクター・フレームワークが、`terminate()` を呼び出す。	77 ページの『コネクターのシャットダウン』

以降のセクションでは、表 20 の各実行ステップの詳細について説明します。

コネクターの始動

各コネクターには、その実行を開始するコネクター始動スクリプトがあります。この始動スクリプトにより、コネクター・フレームワークが起動されます。

注: コネクター始動スクリプトの作成方法を詳細については、248 ページの『始動スクリプトの作成』を参照してください。

コネクター・フレームワークは、動作すると、統合ブローカーに応じて、該当するステップを実行してコネクターのアプリケーション固有コンポーネントを起動します。

InterChange Server でのコネクターの開始

InterChange Server が統合ブローカーである場合、コネクター・フレームワークは、次の手順を実行して、アプリケーション固有コンポーネントを起動します。

1. オブジェクト・リクエスト・ブローカー (ORB) を使用して、InterChange Server との通信を確立する。
2. リポジトリーから、次のコネクター定義情報をコネクター処理用メモリーに読み込む。
   - コネクター構成プロパティー
   - コネクターにサポートされているビジネス・オブジェクト定義のリスト
3. コネクター基底クラスのインスタンスを生成し、その基底クラスのメソッドを呼び出して、アプリケーション固有コンポーネントを初期化して、コネクターのアプリケーション固有コンポーネントの実行を開始する。

   コネクターが実行されると、コネクター・フレームワークは、コネクター基底クラスのインスタンスを生成した後、表 21 に記載されているコネクター基底クラス・メソッドを呼び出します。

表 21. コネクターの実行の開始

初期化タスク		詳細情報
1.	コネクターを初期化し、アプリケーションとの接続のオープンなど、アプリケーション固有コンポーネントに対して、必要な初期設定をすべて実行する。	73 ページの『コネクターの初期化』
2.	コネクターがサポートするビジネス・オブジェクトごとに、ビジネス・オブジェクト・ハンドラーを取得する。	75 ページの『ビジネス・オブジェクト・ハンドラーの取得』

   上記のメソッドの呼び出しがすべて完了すると、コネクターは作動可能になります。
4. コネクター・コントローラーと通信し、コラボレーションの登録先となっているビジネス・オブジェクトのサブスクリプション・リストを取得する。詳細については、14 ページの『ビジネス・オブジェクトのサブスクリプションとパブリッシュ』を参照してください。

その他の統合ブローカーのコネクターの開始

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message

Broker) または WebSphere Application Server の場合は、コネクター・フレームワークは、以下のステップを実行してアプリケーション固有のコンポーネントを起動します。

1. ローカル・リポジトリーから、次のコネクター定義情報をコネクター処理用メモリーに読み込む。
   - コネクター構成プロパティー
   - コネクターにサポートされているビジネス・オブジェクト定義のリスト
2. コネクター基底クラスのインスタンスを生成し、その基底クラスのメソッドを呼び出して、アプリケーション固有コンポーネントを初期化して、コネクターのアプリケーション固有コンポーネントの実行を開始する。

   コネクターが実行されると、コネクター・フレームワークは、コネクター基底クラスのインスタンスを生成した後、表 21 に記載されているコネクター基底クラス・メソッドを呼び出します。上記のメソッドの呼び出しがすべて完了すると、コネクターは作動可能になります。

コネクターの初期化

コネクターの初期化を開始するため、コネクター・フレームワークは、コネクター基底クラスの初期化メソッドを呼び出します。表 22 に、コネクターの初期化メソッドを示します。

表 22. コネクターを初期化するコネクター基底クラス・メソッド

クラス	メソッド
CWConnectorAgent	agentInit()

コネクター・クラスの実装の一環として、コネクターの初期化メソッドを実装する必要 があります。初期化メソッドのメインタスクの内容は以下のとおりです。

- 『接続の確立』
- 74 ページの『コネクター・バージョンの検査』
- 74 ページの『進行中イベントのリカバリー』

重要: 初期化メソッドの実行中には、ビジネス・オブジェクト定義とコネクター・フレームワークのサブスクリプション・リストはまだ使用できません。

接続の確立: 初期化メソッドのメインタスクは、アプリケーションとの接続を確立することです。アプリケーションとの接続を確立するため、初期化メソッドは次のタスクを実行します。

- リポジトリーから、コネクター情報を提供するコネクターの構成プロパティー (ApplicationUserID および ApplicationPassword など) を読み出した後、それらを使用して、ログイン情報をアプリケーションに送信します。空である必須のコネクター・プロパティーに対しては、初期化メソッドにより、デフォルト値が設定されます。

  コネクター構成プロパティーの値を取得するには、getConfigProp() メソッドを使用します。詳細については、79 ページの『コネクター構成プロパティー値の使用』を参照してください。

- 必要な接続またはファイルをすべて取得します。例えば、初期化メソッドが、通常の場合、アプリケーションとの接続を開きます。初期化メソッドは、コネクターが接続を正常に開いた場合に、「success」を戻します。コネクターが接続を開くことができない 場合、初期化メソッドは該当する障害状況を戻し、障害の原因を常時示します。

  Java コネクターでは、agentInit() メソッドは、接続に障害が起こった場合には ConnectionFailureException 例外をスローし、コネクターがアプリケーションにログインできない場合には LogonFailureException 例外をスローします。これらの条件については、236 ページの『例外』を参照してください。

コネクター・バージョンの検査: getVersion() メソッドが、コネクターのバージョンを戻します。このメソッドは、次のコンテキストの両方で呼び出されます。

- 初期化メソッドは、コネクター・バージョンを確認するため、getVersion() を呼び出します。
- コネクター・フレームワークは、コネクターのバージョンを取得する必要があるときに、getVersion() メソッドを呼び出します。

注: コネクターは、自身がサポートしているアプリケーションのバージョンを追跡管理します。コネクターは、アプリケーションへのログオン時、アプリケーションのバージョンをチェックします。

進行中イベントのリカバリー: イベント通知時のイベント処理は、アプリケーション・エンティティーに対する取得の実行、イベントに対する新規ビジネス・オブジェクトの作成、および作成したビジネス・オブジェクトのコネクター・フレームワークへの送信から構成されます。コネクターがイベント状況を更新し、イベントの送信の成功または失敗を示さない内に、イベント処理中でありながら終了した場合は、その進行中イベントはイベント・ストア内に留まります。コネクターは、リスタートされると、進行中イベント状況のイベントを確認するため、イベント・ストアをチェックします。

コネクターは、進行中状況のイベントを検出すると、表 23 に概要が示されているタスクのいずれかを実行することができます。この振る舞いは、設定可能です。複数のコネクターが、この目的のために InDoubtEvents コネクター構成プロパティーを使用します。その設定値も、表 23 に示されています。

表 23. 進行中イベントをリカバリーするためのアクション

イベント・リカバリー・アクション	InDoubtEvents の値
進行中イベントの状況をポーリング・レディー (Ready-for-Poll) に変更して、後続のポーリング呼び出しで、コネクター・フレームワークにイベントの処理を依頼できるようにします。**注:** イベント再処理の依頼があると、重複イベントが生成される可能性があります。リカバリー時の重複イベントの生成を確実に防ぐには、別のリカバリー応答を使用します。	Reprocess
致命的エラーをログに記録し、コネクターをシャットダウンします。LogAtInterchangeEnd が True に設定されている場合は、エラーに関する電子メール通知が起動されます。	FailOnStartup
コネクターをシャットダウンすることなくエラーをログに記録します。	LogError
イベント・ストアにある進行中イベント記録を無視します。	Ignore

Java コネクターの場合は、イベント・ストアから進行中状況のイベント・レコードを取得し、適切なリカバリー処置を講じるための recoverInProgressEvents() メソッドが、CWConnectorEventStore クラスに用意されています。コネクター開発者は、このメソッドを実装して、InDoubtEvents の値に基づいてアクションを実行することができます。このメソッドで、コネクター開発者は、進行中イベントの状況をポーリング・レディー (Ready-for-Poll) 状況に変更できます。

注: イベント通知、イベント・ストア、および進行中イベントの詳細については、131 ページの『第 5 章 イベント通知』を参照してください。

ビジネス・オブジェクト・ハンドラーの取得

コネクター初期化の最終ステップとして、コネクター・フレームワークは、コネクターがサポートする各ビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。*ビジネス・オブジェクト・ハンドラー* は、コネクター・フレームワークから要求ビジネス・オブジェクトを受け取った後、それらの要求ビジネス・オブジェクト内で定義されている動詞処理を実行します。各コネクターは、ビジネス・オブジェクト・ハンドラーを取得するため、そのコネクター基底クラス内に getConnectorBOHandlerForBO() メソッドを定義しておく必要があります。このメソッドは、指定されたビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーへの参照を戻します。

重要: コネクター基底クラスの実装の一環として、コネクターに対するビジネス・オブジェクト・ハンドラーを取得するため、 getConnectorBOHandlerForBO() を実装する*必要* があります。

ビジネス・オブジェクト・ハンドラー (1 つ以上) のインスタンスを生成するため、コネクター・フレームワークは、次の手順を実行します。

1. 初期化の間に、コネクター・フレームワークは、コネクターがサポートするビジネス・オブジェクト定義のリストを受け取る。詳細については、72 ページの『コネクターの始動』を参照してください。
2. その後、コネクター・フレームワークは、サポートされている各ビジネス・オブジェクトに対して、 getConnectorBOHandlerForBO() メソッドを 1 回呼び出す。
3. getConnectorBOHandlerForBO() メソッドは、引き数として受け取ったビジネス・オブジェクト定義の名前に基づいて、そのビジネス・オブジェクトに適切なビジネス・オブジェクト・ハンドラーのインスタンスを生成する。このメソッドは、呼び出し側のコネクター・フレームワークにビジネス・オブジェクト・ハンドラーを戻します。

   インスタンスの生成されるビジネス・オブジェクト・ハンドラーの個数は、次のように、実際のコネクターのビジネス・オブジェクト処理の全体的な設計に依存します。

   - アプリケーション固有ビジネス・オブジェクトに対するビジネス・オブジェクト定義に、一貫性のある規則に従うメタデータが含まれている場合は、コネクターはメタデータ主導型となります。ビジネス・オブジェクト処理は、*メタデータ主導型ビジネス・オブジェクト・ハンドラー* を使用するように設計することが可能です。

メタデータ主導型コネクターは、メタデータ主導型ビジネス・オブジェクト・ハンドラーと呼ばれる単一の汎用ビジネス・オブジェクト・ハンドラーにおいて、すべて のビジネス・オブジェクトを処理します。したがって、getConnectorBOHandlerForBO() メソッドは、コネクターがサポートするビジネス・オブジェクトの個数に関係なく、1 つのビジネス・オブジェクト・ハンドラーのインスタンスを生成するだけで済みます。このメソッドは、最初の呼び出し時にビジネス・オブジェクト・ハンドラーを作成し、その後の呼び出しのたびに、この作成したハンドラーに対するポインターを戻します。

- アプリケーション固有のビジネス・オブジェクトの一部またはすべてが特殊な処理を必要とする場合は、それらのオブジェクトに対して、複数のビジネス・オブジェクト・ハンドラー をセットアップする必要があります。

  コネクターが各ビジネス・オブジェクトごとに別々のビジネス・オブジェクト・ハンドラーを必要とする場合、 getConnectorBOHandlerForBO() メソッドは、引き渡されたビジネス・オブジェクトの名前に基づいて、該当するビジネス・オブジェクト・ハンドラーのインスタンスを生成することができます。この場合、 getConnectorBOHandlerForBO() メソッドは、別々のビジネス・オブジェクト・ハンドラーを必要とする各ビジネス・オブジェクト定義に対して 1 つずつ、複数のビジネス・オブジェクト・ハンドラーのインスタンスを生成します。ビジネス・オブジェクト・ハンドラー取得メソッドが、呼び出されるたびに別のビジネス・オブジェクト・ハンドラーのインスタンスを生成します。

4. コネクター・フレームワークは、このビジネス・オブジェクト・ハンドラーに対する参照を、(コネクター処理メモリーに常駐する) 関連のビジネス・オブジェクト定義に格納する。

重要: getConnectorBOHandlerForBO()メソッドを実装する前に、コネクターのビジネス・オブジェクト処理設計を完了することが必要になる場合があります。アプリケーション固有ビジネス・オブジェクトの設計方法については、52 ページの『メタデータ主導型の設計を評価するためのサポート』を参照してください。

getConnectorBOHandlerForBO() メソッドのインプリメント方法については、176 ページの『Java ビジネス・オブジェクト・ハンドラーの取得』を参照してください。ビジネス・オブジェクト・ハンドラーのインプリメント方法については、91 ページの『第 4 章 要求処理』を参照してください。

イベントのポーリング

コネクターが、イベント通知を導入する場合は、イベント通知機構を実装する必要があります。イベント通知には、アプリケーション・ビジネス・エンティティーへの変更検出のため、アプリケーションと連携動作するメソッドが含まれます。検出された変更はイベントとして表され、(InterChange Server などの) 宛先への経路指定による送信のため、コネクターによりコネクター・フレームワークに送信されます。

イベント通知にポーリング機構を使用するコネクターの場合は、pollForEvents() メソッドをコネクターに実装し、イベント・ストアからイベント情報を周期的に取得する必要があります。イベント・ストアは、コネクターによる処理が可能になるまで、アプリケーションの生成したイベントを保持します。ポーリングがオンの場

合、コネクター・フレームワークは、呼び出しメソッド pollForEvents() を呼び出します。pollForEvents() メソッドは、ポーリング操作の状況を示す整数値を戻します。

Java コネクター・ライブラリーでは、pollForEvents() メソッドは CWConnectorAgent クラス内で定義されます。 pollForEvents() で使用される一般的な戻りコードは、SUCCEED、FAIL、および APPRESPONSETIMEOUT です。戻りコードの詳細については、234 ページの『Java 戻りコード』を参照してください。

重要: 開発者は、pollForEvents() メソッドの実装を開発する必要があります。コネクターが、要求処理のみ をサポートする場合は、pollForEvents() を完全には、実装する必要はありません。ただし、同ポーリング・メソッドは、必須のメソッドであるため、そのスタブを実装する必要があります。 Java コネクター・ライブラリーでは、pollForEvents() メソッドのデフォルト実装が提供されています。

pollForEvents() のインプリメントとイベント通知の詳細については、131 ページの『第 5 章 イベント通知』を参照してください。

コネクターのシャットダウン

管理者は、コネクターの始動スクリプトを終了して、コネクターをシャットダウンします。コネクターのシャットダウン時、コネクター・フレームワークが、コネクター基底クラスの terminate() メソッドを呼び出します。 terminate() メソッドのメインタスクは、アプリケーションとの接続をクローズすること、および割り振られているリソースがあれば、それを解放することです。

コネクター基底クラスの拡張

コネクターを作成するには、コネクター・ライブラリーに提供されているコネクターの基底クラス を拡張します。コネクターの基底クラスには、コネクターのアプリケーション固有コンポーネントの初期化メソッドとセットアップ・メソッドが含まれています。派生コネクター・クラス には、コネクターのアプリケーション固有コンポーネントに対するコーディングが含まれます。

注: コネクターの命名規則については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*IBM WebSphere InterChange Server コンポーネント命名ガイド*」を参照してください。

コネクター基底クラスには、表 24 に示すメソッドが組み込まれています。実際のコネクターには、これらのメソッドを実装する必要があります。

表 24. コネクター基底クラスに実装するメソッド

説明	コネクターの基底クラス・メソッド	詳細情報
コネクターのアプリケーション固有のコンポーネントを初期化します。	agentInit()	73 ページの『コネクターの初期化』
コネクターのバージョンを戻します。	getVersion()	74 ページの『コネクター・バージョンの検査』
1 つ以上のビジネス・オブジェクト・ハンドラーを設定します。	getConnectorBOHandlerForBO()	75 ページの『ビジネス・オブジェクト・ハンドラーの取得』

表 24. コネクター基底クラスに実装するメソッド (続き)

説明	コネクターの基底クラス・メソッド	詳細情報
アプリケーションのイベントをポーリングします。	`pollForEvents()`	76 ページの『イベントのポーリング』
コネクターの終了時、クリーンアップ・タスクを実行します。	`terminate()`	77 ページの『コネクターのシャットダウン』

図 25 に、コネクター・フレームワークが呼び出す一連のメソッドを図示するとともに、その中で始動時と実行時に呼び出されるメソッドも示します。コネクター・フレームワークが呼び出すメソッドの中で、1 つのメソッドを例外として、すべてのメソッドがコネクター基底クラス内に存在します。例外のメソッドとは `doVerbFor()` であり、ビジネス・オブジェクト・ハンドラー内に存在します。`doVerbFor()` メソッドのインプリメント方法については、 91 ページの『第 4 章 要求処理』を参照してください。

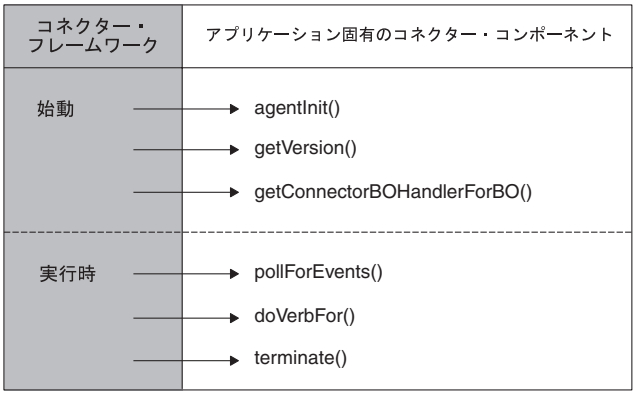

図 25. コネクター・フレームワークによって呼び出されるメソッドの要約

コネクター基底クラスの拡張方法の詳細については、 178 ページの『Java ビジネス・オブジェクト・ハンドラーの基底クラスの拡張』を参照してください。

エラー処理

コネクター・クラス・ライブラリーのメソッドは、次の方法でエラー状態を示します。

- 戻りコード — コネクター・クラス・ライブラリーには、定義済み結果状況値のセットが含まれています。抽象メソッドは、結果状況値を使用して、メソッドの成功、失敗に関する情報を戻すことができます。戻りコードは、整数値と結果状況定数として定義されます。コーディングをする場合、IBM では、問題を防ぐために定義済み定数の使用を推奨しています。その使用により、IBM が定数の値を変更した場合でもコーディングを変更しないで済みます。

  Java 戻りコードについては、 234 ページの『Java 戻りコード』を参照してください。
- 例外 — Java コネクター・ライブラリーは、例外情報を含む例外オブジェクトおよび例外詳細オブジェクトをカプセル化するクラスを提供します。詳細については、 236 ページの『例外』を参照してください。

- 戻り状況記述子 — 要求処理の間、コネクター・フレームワークは、統合ブローカーに状況情報を戻り状況記述子で返送します。ビジネス・オブジェクト・ハンドラーは、この記述子内にメッセージと状況コードを保管し、動詞の処理状況を統合ブローカーに提供することができます。詳細については、238 ページの『戻り状況記述子』を参照してください。
- エラーおよびメッセージ・ロギング — コネクター・クラス・ライブラリーでは、エラーや注目すべき状態の通知をサポートするために、以下の機能も提供されます。
  - ロギングにより、ロギング先に情報メッセージまたはエラー・メッセージを送信することが可能になります。
  - トレースにより、さまざまなトレース・レベルでトレース・メッセージを生成するステートメントをコーディング内に記述することが可能になります。

  ロギングとトレースのインプリメント方法の詳細については、159 ページの『第 6 章 メッセージ・ロギング』を参照してください。

コネクター構成プロパティー値の使用

ここでは、コネクター構成プロパティーに関する次の項目について説明します。

- 『コネクター構成プロパティーとは』
- 80 ページの『コネクター構成プロパティーの定義と設定』
- 80 ページの『コネクター構成プロパティーの取得』

コネクター構成プロパティーとは

コネクター構成プロパティー (単にコネクター・プロパティー と呼ばれることもあります) により、コネクターが必要な情報にアクセスする際に使用する名前付きプレースホルダー (変数と同等) を作成できるようになります。コネクターには、以下の 2 つのカテゴリーの構成プロパティーがあります。

- 標準構成プロパティー
- コネクター固有の構成プロパティー

標準コネクター構成プロパティー

標準構成プロパティーは、コネクター・フレームワークが一般的に使用する情報を提供します。このタイプのプロパティーは、通常、すべての コネクターに共通しており、WebSphere Business Integration システムの振る舞いを明確に定義することができます。

コネクター固有の構成プロパティー

コネクター固有の構成プロパティーは、特定のコネクターが実行時に必要とする情報を提供します。このタイプの構成プロパティーにより、再コーディングまたは再ビルドを行わずに、コネクターのアプリケーション固有コンポーネント内の静的情報やロジックを変更する方法が提供されます。例えば、構成プロパティーは、次の目的に使用することができます。

- アプリケーション・サーバーやデータベースの名前、イベント表の名前、またはコネクターが読み取る必要のあるファイルの名前などの定数の値を維持すること。

- 特定の状況におけるコネクターの振る舞いを設定すること。例えば、構成プロパティーにより、子オブジェクトが欠落している階層型ビジネス・オブジェクトに対するビジネス・オブジェクト Retrieve 操作を、コネクターが失敗と処理しないように指示することができます。別の例として、構成プロパティーにより、Create 操作で新規オブジェクトの ID を、アプリケーションまたはコネクターのどちらが作成するかを決定することができます。

コネクター固有構成プロパティーは、コネクターに対して個数の制限なしに作成することができます。必要となるコネクター固有プロパティーを見極めた後、コネクター構成プロセスの一環として定義します。Connector Configurator を使用して、ローカル・リポジトリーに保管される情報の一部として、コネクター構成プロパティーを指定します。

必要に応じて、構成プロパティーを後日追加することも可能です。コネクターのコードが実行しなければならない作業は一般的に、コネクター固有プロパティー (例えば、`ApplicationUserID` や `ApplicationPassword` など) の値の照会のみです。

コネクター構成プロパティーの定義と設定

Connector Configurator ツールにより、コネクター構成プロパティーに対して次の作業を行うことが可能になります。

- 標準構成プロパティーに値を割り当てること。
- コネクター固有構成プロパティーを定義した後、値を割り当てること。

  System Manager ツールから Connector Configurator を起動すること。

WebSphere InterChange Server

統合ブローカーが WebSphere InterChange Server の場合は、Connector Configurator ツールの詳細について「*WebSphere InterChange Server システム・インプリメンテーション・ガイド*」を参照してください。

その他の統合ブローカー

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) の場合は、Connector Configurator の詳細について「*WebSphere Message Brokers 使用アダプター・インプリメンテーション・ガイド* 」を参照してください。統合ブローカーが WebSphere Application Server の場合は、Connector Configurator の詳細について「*アダプター実装ガイド (WebSphere Application Server)*」を参照してください。

コネクター構成プロパティーの取得

コネクター構成プロパティーは、コネクター初期化の一環としてコネクターにダウンロードされます (詳細については、72 ページの『コネクターの始動』を参照)。コネクターのアプリケーション固有コンポーネントは、コネクター・プロパティーのタイプに基づいた初期化に必要な構成プロパティーの値を取得します。

コネクターは、以下のいずれかのタイプのコネクター構成プロパティーを使用できます。

- シンプル・コネクター構成プロパティーは、ストリング値のみを含みます。これには、その他のプロパティーは含まれません。単一値のシンプル・プロパティーには、1 つのストリング値しか含まれません。
- *階層* コネクター構成プロパティーは、他のプロパティーとその値を含みます。所定のコネクター・プロパティーには、複数の値を含めることができます。

注: IBM WebSphere Business Integration Adapters 製品では、C++ コネクターでサポートされるコネクター・プロパティーは単一値のシンプル・コネクター構成プロパティーのみです。C++ コネクターは階層プロパティーをサポートしません。

注: 前のバージョンの製品では、コネクター構成プロパティーは、単一値のシンプル・プロパティーのみでした。つまり、コネクター・プロパティーに含めることができるのは、ストリング値が 1 つだけでした。今回のリリースでは、Java コネクターによって階層プロパティーをサポートすることができるようになりました。上記で述べたとおり、階層プロパティーは、他のプロパティーと複数の値を含むことができます。バージョン 2.2.0 以降では階層プロパティーがサポートされます。IBM WebSphere InterChange Server 製品でのサポートはバージョン 4.2 からです。

単一値のシンプル・コネクター構成プロパティーの取得

前のバージョンの製品では、コネクター構成プロパティーは、単一値のシンプル・プロパティーのみ でした。つまり、コネクター・プロパティーに含めることができるのは、ストリング値が *1* つ だけでした。単一値のシンプルなコネクター構成プロパティーを取得するには、`getConfigProp()` メソッドを使用します。

注: IBM WebSphere Business Integration Adapters 製品では、単一値のシンプル・コネクター構成プロパティーは、バージョン 2.2.0 *より前の* すべてのリリースでサポートされている唯一のコネクター・プロパティー・タイプでした。IBM WebSphere InterChange Server 製品では、単一値のシンプル・コネクター構成プロパティーは、バージョン 4.2 *より前の* すべてのリリースでサポートされている唯一のコネクター・プロパティー・タイプでした。ここで説明する単一値のシンプル・コネクター・プロパティーにアクセスするための機構は、後方互換性を維持するために、引き続き Java コネクターによりサポートされます。ただし IBM では、新たにコネクターを開発する際は、82 ページの『階層コネクター構成プロパティーの取得』で説明するメカニズムを使用して、階層プロパティーとしてコネクター構成プロパティーにアクセスすることをお勧めします。

Java コネクター・ライブラリーでは、シンプル・コネクター構成プロパティーの値を取得するための、表 25 に示す 2 つのメソッドを提供しています。

表 25. シンプル・コネクター構成プロパティーの値を取得するためのメソッド

コネクターのライブラリー・メソッド	説明
`getConfigProp()`	指定されているシンプル・コネクター構成プロパティーの値を取得します。

表 25. シンプル・コネクター構成プロパティーの値を取得するためのメソッド (続き)

コネクターのライブラリー・メソッド	説明
getAllConnectorAgentProperties()	すべて のコネクター構成プロパティーの値を取得します。ただし、メソッドが複数の値のコネクター・プロパティーを取得する場合、最初のコネクター・プロパティー値のみが取得されます。

上記のメソッドは両方とも、次のように、CWConnectorUtil クラスと関数で定義されています。

- getConfigProp() メソッドは、入力データとして構成プロパティーの名前に対するストリングを取り、そのプロパティーの値を Java String として戻します。
- getAllConnectorAgentProperties() メソッドは、入力引き数を必要とせず、すべてのコネクター構成プロパティーの値を Java Hashtable に戻します。

図 26 のコーディングの抜粋では、getAllConnectorAgentProperties() メソッドを使用して、connectorProperties という名前の Java Hashtable オブジェクトに、すべてのコネクター構成プロパティーを取得して取り込みます。このコーディングの抜粋では、Hashtable クラスの get() メソッドを使用して各コネクター構成プロパティーの値を取得します。

```
connectorProperties =
    CWConnectorUtil.getAllConnectorAgentProperties();

// get Connector Configuration Properties to establish Connection
String connectString =
    (String)connectorProperties.get("ConnectString");
String userName =
    (String)connectorProperties.get("ApplicationUserName");
String userPassword =
    (String)connectorProperties.get("ApplicationPassword");

if(connectString == null || connectString.equals("")
        || userName==null || userPassword==null )
```

図 26. すべてのコネクター構成プロパティーの取得

階層コネクター構成プロパティーの取得

階層コネクター構成プロパティーには、以下のいずれかの値を含めることができます。

- 1 つ以上の子プロパティー。それぞれの子プロパティーには、それ自身の子プロパティーとストリング値を含めることができます。
- 1 つ以上のストリング値。

複数のストリング値を持つ階層コネクター・プロパティーは、*多値* プロパティーと呼ばれます。ストリング値を 1 つだけ持つプロパティーは、*単一値* プロパティーと呼ばれます。

Java コネクター・ライブラリーは、階層コネクター構成プロパティーを CWProperty class によって表します。このクラスのオブジェクトは、*コネクター・プロパティー・オブジェクト* と呼ばれ、シンプルまたは階層、単一値または多値のコネクター構成プロパティーを表すことができます。

表 26 に、コネクター・プロパティー・オブジェクトが提供する階層コネクター構成プロパティーのメタデータをリストします。

表 26. コネクター・プロパティー・オブジェクトのメタデータ

コネクター・プロパティー情報	説明	CWProperty メソッド
名前	コネクター・プロパティーの名前	getName()
カーディナリティー	コネクター・プロパティーが保持できる値の数を示します。 • 単一値 • 多値	getCardinality()
プロパティー・タイプ	コネクター・プロパティーが子プロパティーを含むかどうかを示します。 • シンプル: 子プロパティーは含まず、ストリング値のみ保持します。 • 階層: 1 つ以上の子プロパティーを含みます。	getPropType()
暗号化フラグ	プロパティー値が暗号化されるかどうかを示します。	getEncryptionFlag()、setEncryptionFlag()

表 26 に示すように、コネクター・プロパティーについてのメタデータの取得は、指定されたメソッドを使用して行います。ただし、プロパティー*値* の取得には、次の 2 つのステップのプロセスが必要です。

1. 1 つまたはすべてのコネクター構成プロパティーのトップレベル・コネクター・プロパティー・オブジェクトの取得。
2. コネクター・プロパティー・オブジェクトからの必要なプロパティー値の取得。

トップレベル・コネクター・プロパティー・オブジェクトの取得: コネクター・プロパティーのトップレベル・コネクター・プロパティー・オブジェクトの取得には、表 27 に示すいずれかのメソッドを使用できます。

表 27. トップレベル・コネクター・プロパティー・オブジェクトの取得用メソッド

コネクターのライブラリー・メソッド	説明
getHierarchicalConfigProp()	指定された階層コネクター構成プロパティーのトップレベル・コネクター・プロパティー・オブジェクトを取得します。
getAllConfigProperties()	プロパティーのタイプ (シンプル、階層、または多値) に関わらず、すべての コネクター構成プロパティーのトップレベル・コネクター・プロパティー・オブジェクトを取得します。

表 27 に示したメソッドは、両方とも、次のように CWConnectorUtil クラスと関数で定義されています。

- getHierarchicalConfigProp() メソッドは、コネクター構成プロパティーの名前を引き数として使用します。これは、指定されたコネクター・プロパティーのトップレベル・コネクター・プロパティー・オブジェクトを含む、単一の CWProperty オブジェクトを戻します。
- getAllConfigProperties() メソッドは、CWProperty オブジェクトの配列を戻します。各配列には、いずれかのコネクター構成プロパティーのトップレベル・コネクター・プロパティー・オブジェクトが含まれています。

コネクター・プロパティー値の取得: コネクター・プロパティーのトップレベル・コネクター・プロパティー・オブジェクトを取得すると、このコネクター・プロパティー・オブジェクトから値を取得できます。上記で説明したように、階層コネクター・プロパティーは、以下のような種類の値を 1 つ以上保持できます。

- 1 つ以上の子プロパティー
- 1 つ以上のストリング値

子プロパティーの取得: CWProperty クラスは、コネクター・プロパティー・オブジェクトから子プロパティーを取得するために、表 28 に示すメソッドを提供します。

表 28. コネクター・プロパティー・オブジェクトから子プロパティーの値を取得するためのメソッド

説明	CWProperty メソッド
階層コネクター・プロパティーのすべての 子プロパティーの取得	getHierChildProps()
指定されたプレフィックスを持つ階層コネクター・プロパティーのすべての子プロパティーの取得	getChildPropsWithPrefix()
階層コネクター・プロパティーからの指定された単一の子プロパティーの取得	getHierChildProp()

hasChildren() メソッドを使用して、現在のコネクター・プロパティー・オブジェクトに子プロパティーが含まれているかどうかを判別できます。

ストリング値の取得: CWProperty クラスは、コネクター・プロパティー・オブジェクトからストリング値を取得するために、表 29 に示すメソッドを提供します。

表 29. コネクター・プロパティー・オブジェクトからストリング値を取得するためのメソッド

説明	CWProperty メソッド
階層コネクター・プロパティーのすべての ストリング値の取得	getStringValues()
指定された子プロパティーのすべてのストリング値の取得	getChildPropValue()

hasValue() メソッドを使用して、現在のコネクター・プロパティー・オブジェクトにストリング値が含まれているかどうかを判別できます。

データ・ハンドラーの呼び出し

アプリケーション固有の形式とビジネス・オブジェクト間のデータ変換は、コネクターの主要タスクです。コネクターは多くの場合、この変換を直接実行しなければなりません。例えば、コネクターは、適切なデータベース・ステートメントを作成することによって、アプリケーション・データベースの表内の行としてのデータの作成またはアクセスを可能にしていますが、場合によっては、共通の多目的インターネット・メール拡張仕様 (MIME) フォーマットの直列化データを処理することもあります。

各コネクター固有の MIME フォーマットとビジネス・オブジェクトとの変換を実行せずに済むように、WebSphere InterChange Server と WebSphere Business Integration Adapters の両方で、これらの一般的な変換を実行するためのデータ・ハンドラー が提供されています。データ・ハンドラーは、特定の MIME フォーマットの直列化データとビジネス・オブジェクトとの変換を行う特殊な Java クラス・インスタンスです。例えば、WebSphere Business Integration Data Handler for XML は、XML 文書とビジネス・オブジェクトとの変換を行うデータ・ハンドラーを提供します。

注: このセクションでは、データ・ハンドラーの概要を簡単に説明します。データ・ハンドラーの詳細については、「*データ・ハンドラー・ガイド*」を参照してください。

Java コネクター・ライブラリーには、コネクター内から特定のデータ・ハンドラーを呼び出すためのデータ・ハンドラー・メソッドがいくつか用意されています。使用するデータ・ハンドラー・メソッドを決定するには、次の作業を実行する必要があります。

- 『データ変換の方向の決定』
- 86 ページの『直列化データへのアクセス』
- 87 ページの『インスタンスを生成するデータ・ハンドラーの識別』

データ変換の方向の決定

通常、データ・ハンドラーは、直列化データとビジネス・オブジェクトとの間で両方向に変換を行うことができます。つまり、次の変換を両方とも実行できます。

- *ストリングからビジネス・オブジェクトへの変換* では、直列化データをビジネス・オブジェクトに変換します。

  この変換は、コネクター内で、コネクターがアプリケーションから直列化データを受信し、統合ブローカーに送信するためにこのデータの適切なビジネス・オブジェクト表現を作成する必要がある、イベントの処理時に役立ちます。コネクターは、適切なデータ・ハンドラーにビジネス・オブジェクトを送信し、対応する直列化データをデータ・ハンドラーから受信できます (希望の形式の直列化データに変換するデータ・ハンドラーが存在する場合)。
- *ビジネス・オブジェクトからストリングへの変換* では、ビジネス・オブジェクトを直列化データに変換します。

  この変換は、コネクターが統合ブローカーからビジネス・オブジェクトを受信し、アプリケーションへ送信するために適切な直列化データを作成する必要があ

る、要求の処理時に役立ちます。コネクターは、適切なデータ・ハンドラーにビジネス・オブジェクトを送信し、対応するビジネス・オブジェクトを受信できます (希望の形式の直列化データを変換するデータ・ハンドラーが存在する場合)。

Java コネクター・ライブラリーには、表 30 に示すデータ・ハンドラー・メソッドが用意されています。これらのメソッドにより、コネクターは、データ・ハンドラーを呼び出し、特定の MIME フォーマットの直列化データとビジネス・オブジェクトとの変換を実行できます。これらのメソッドは、`CWConnectorUtil` クラスに定義されています。

表 30. Java コネクター・ライブラリーのデータ・ハンドラー・メソッド

型変換	型変換プロセス	メソッド
ビジネス・オブジェクトからストリング	データ・ハンドラーを呼び出し、指定したビジネス・オブジェクト (*theBusObj* 引き数) を直列化データに変換して、サポートされたアクセス形式のいずれかでこのデータを戻します。詳細については、『直列化データへのアクセス』を参照してください。	399 ページの『boToByteArray()』 402 ページの『boToStream()』 404 ページの『boToString()』
ストリングからビジネス・オブジェクト	データ・ハンドラーを呼び出し、指定した直列化データ (*serializedData* 引き数) をビジネス・オブジェクトに変換します。	406 ページの『byteArrayToBo()』 427 ページの『readerToBO()』 429 ページの『streamToBO()』 431 ページの『stringToBo()』

要求された変換をデータ・ハンドラーが実行できない場合は、データ・ハンドラー・メソッドによって `ParseException` 例外がスローされます。

直列化データへのアクセス

データ・ハンドラー・メソッドとの間で送受信される直列化データにアクセスするには、以下の情報を指定する必要があります。

- コードで直列化データにアクセスするときの形式
- 直列化データのロケール

データ形式の選択

データ・ハンドラーの目的は、直列化データとビジネス・オブジェクトとの変換を行うことです。したがって、Java コネクターのコードがこの直列化データにアクセスできる必要があります。コードは、表 31 に示されている形式のデータにアクセスできます。Java コネクター・ライブラリーには、これらの形式の直列化データをそれぞれサポートするデータ・ハンドラー・メソッドが用意されています。

表 31. 直列化データとデータ・ハンドラー間のアクセス方法

直列化データへのアクセス	Java 構成体	メソッド
ストリング	String オブジェクト	boToString() stringToBo()
入力ストリーム	java.io.InputStream クラスのオブジェクトまたはそのサブクラスのいずれか	boToStream() streamToBO()
文字ストリームのリーダー	java.io.Reader クラスのオブジェクトまたはそのサブクラスのいずれか	readerToBO()
バイトの配列	byte[]	boToByteArray() byteArrayToBo()

データ・ハンドラーとの間で送受信される直列化データにアクセスするには、表 31 から適切なアクセス形式を処理するデータ・ハンドラー・メソッドを選択します。

データのロケールおよびエンコードの識別

86 ページの表 30 に示すように、データ・ハンドラー・メソッドはデータ・ハンドラーを呼び出し、直列化データを読み出すか (ストリングからビジネス・オブジェクトへの変換) 直列化データを作成します (ビジネス・オブジェクトからストリングへの変換)。この処理中に、データ・ハンドラーが処理対象とする直列化データの文字エンコードまたはロケールを認識する必要がある場合があります。データ・ハンドラーが使用する別のロケールまたは文字エンコードを指定できるようにするために、データ・ハンドラー・メソッドは、以下の情報を指定する Java Locale オブジェクトおよび String エンコード引数を受け入れます。

- ロケールがコネクター・フレームワークのロケールと*同じ* 場合は、データ・ハンドラー・メソッドを呼び出すときに *locale* 引き数に null を指定できます。ロケールが異なる場合は、適切なロケール情報を含む java.util.Locale オブジェクトを指定します。
- 文字エンコードがコネクター・フレームワークの使用する文字エンコードと*同じ* 場合は、データ・ハンドラー・メソッドを呼び出すときに *encoding* 引き数に null を指定できます。文字エンコードが異なる場合は、適切な文字エンコードを Java String として指定します。

コネクター・フレームワークのロケールまたは文字エンコードを取得する方法については、 63 ページの『国際化対応コネクターの設計上の考慮事項』を参照してください。

インスタンスを生成するデータ・ハンドラーの識別

インスタンス生成を必要とするデータ・ハンドラーを識別するため、データ・ハンドラー・メソッドは、データ・ハンドラーのクラスを特定するために必要な情報をインスタンス生成プロセスに提供する必要があります。このデータ・ハンドラー・クラスは、データ・ハンドラーを実装する Java クラスの名前です。

注: データ・ハンドラー・メソッドは、指定された変換を要求する*前*に、データ・ハンドラーのインスタンスを生成する必要があります。このインスタンス生成プロセスは、DataHandler 基底クラスの createHandler() メソッドによって実

装されます。DataHandler クラスとデータ・ハンドラー構成情報の詳細については、「データ・ハンドラー・ガイド」を参照してください。

データ・ハンドラー・メソッドは、*mimeType* 引き数で直列化データの MIME タイプを指定し、その *BOPrefix* 引き数をオプションで指定することによって、データ・ハンドラー・クラスの名前を指定できます。この MIME タイプは、次のように、トップレベルのメタオブジェクト内の子メタオブジェクトからデータ・ハンドラーのクラスを取得するために使用されます。

- データ・ハンドラー・メソッドは、トップレベルのメタオブジェクトを調べて、この指定された MIME タイプに対応するデータ・ハンドラーが存在するかどうかを確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されていない場合は、データ・ハンドラー・メソッドによって PropertyNotSetException 例外がスローされます。
- トップレベルのメタオブジェクトには、サポートされている MIME タイプを示す名前を持つ属性が含まれています。属性型は、指定された MIME タイプに対応する子メタオブジェクトを識別します。この子メタオブジェクトには、データ・ハンドラーのクラス名など、データ・ハンドラーの構成情報が含まれています。

この場合、データ・ハンドラー・メソッドは、子メタオブジェクトにリストされた Java クラスのデータ・ハンドラーのインスタンスを生成します。インスタンス生成プロセスは、その MIME タイプに関連する子データ・ハンドラー・メタオブジェクトを使用して、データ・ハンドラー・インスタンスのクラス名とその他の構成情報を導き出します。

注: データ・ハンドラーがインストールされているシステムにはそれぞれ、使用可能なデータ・ハンドラーを表すメタオブジェクトが提供されています。メタオブジェクトとは、構成情報を包含する特殊なビジネス・オブジェクトです。トップレベルのメタオブジェクトは、データ・ハンドラー用であり、使用可能なデータ・ハンドラー、およびデータ・ハンドラーによって使用可能である関連した MIME タイプを包含しています。

メタオブジェクトの詳細、およびインスタンス生成プロセスが指定の MIME タイプからクラス名を導出する方法の詳細については、「データ・ハンドラー・ガイド」を参照してください。

データ・ハンドラーをインスタンス化できない場合は、データ・ハンドラー・メソッドによって DataHandlerCreateException がスローされます。

アプリケーションとの接続が切断された場合の処理

コネクターのアプリケーション固有コードの記述時に留意する点は、アプリケーションとの接続が切断されたら、必ずコネクターがシャットダウンするように設計することです。接続の切断に対応するため、コネクターのアプリケーション固有コンポーネントでは、次の手順を実行します。

- LogAtInterchangeEnd コネクター構成プロパティーが True に設定されている場合に電子メール通知が起動されるように、致命的エラー・メッセージをログに記録します。

- APPRESPONSETIMEOUT 結果状況を戻し、アプリケーションが応答していないことをコネクター・コントローラーに通知します。この事態が起きると、コネクターが実行されているプロセスは停止されます。システム管理者は、アプリケーションに関する問題を修正してから、コネクターを再始動してイベントとビジネス・オブジェクト要求の処理を継続する必要があります。

次のユーザー実装の抽象メソッドは、アプリケーションとの接続の切断をチェックします。

- イベント通知の場合、pollForEvents() メソッドは、接続を確認してから、イベント・ストアにアクセスします。詳細については、 210 ページの『イベント・ストアへのアクセス前の接続の検証』を参照してください。
- 要求処理の場合、doVerbFor() メソッドは、接続を確認してから、動詞処理を開始します。詳細については、 182 ページの『動詞処理の前の接続の検証』を参照してください。

第 4 章 要求処理

この章では、コネクター内で要求処理を準備する方法について説明します。*要求処理* は 1 つの機構を実装します。この機構は、統合ブローカーから要求ビジネス・オブジェクトとして要求を受け取り、アプリケーション・ビジネス・エンティティー内で適切な変更を開始します。要求処理を実装する機構はビジネス・オブジェクト・ハンドラー です。このハンドラーには、アプリケーションと相互作用して要求ビジネス・オブジェクトをアプリケーション操作用の要求に変換するメソッドが含まれています。この章を構成するセクションは次のとおりです。

- 『ビジネス・オブジェクト・ハンドラーの設計』
- 94 ページの『ビジネス・オブジェクト・ハンドラー基底クラスの拡張』
- 95 ページの『要求の処理』
- 99 ページの『Create 動詞の処理』
- 103 ページの『Retrieve 動詞の処理』
- 109 ページの『RetrieveByContent 動詞の処理』
- 111 ページの『Update 動詞の処理』
- 119 ページの『Delete 動詞の処理』
- 121 ページの『Exists 動詞の処理』
- 122 ページの『ビジネス・オブジェクトの処理』
- 129 ページの『コネクター応答の指示』
- 130 ページの『アプリケーションとの接続が切断された場合の処理』

注: 要求処理の概要については、27 ページの『要求処理』を参照してください。

ビジネス・オブジェクト・ハンドラーの設計

ビジネス・オブジェクト・ハンドラーは、コネクター用の要求処理を実装します。したがって、ビジネス・オブジェクト・ハンドラーの定義およびコーディングは、コネクター開発における基本タスクの 1 つです。ビジネス・オブジェクト・ハンドラーは `CWConnectorBOHandler` クラスのサブクラスのインスタンスです。ビジネス・オブジェクト定義はそれぞれ 1 つのビジネス・オブジェクト・ハンドラーを参照し、そのハンドラーには、そのビジネス・オブジェクト定義がサポートする動詞に関するタスクを実行するメソッドのセットが入っています。コネクターがサポートするビジネス・オブジェクトを処理するために、1 つまたは複数のビジネス・オブジェクト・ハンドラーをコーディングする必要があります。

ビジネス・オブジェクト・ハンドラーを実装する方法は、使用す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、このインターフェースがアプリケーション・エンティティーを公開する方法に依存します。コネクターが必要とするビジネス・オブジェクト・ハンドラーの数を判断するには、コネクターが相互作用するアプリケーションを調べる必要があります。

- アプリケーションがフォーム・ベース、表ベース、またはオブジェクト・ベースであって、すべてのエンティティーに適用される標準アクセス方式を持っている

場合は、アプリケーション・エンティティーに関する情報を保管するビジネス・オブジェクトを設計することができます。ビジネス・オブジェクト・ハンドラーは、メタデータ主導型のビジネス・オブジェクト・ハンドラー の中のアプリケーション・エンティティーを処理します。

1 つの汎用ビジネス・オブジェクト・ハンドラー・クラスを派生させてメタデータ主導型のビジネス・オブジェクト・ハンドラーを実装し、すべての ビジネス・オブジェクトを処理させることができます。詳細については、『メタデータ主導型のビジネス・オブジェクト・ハンドラーの実装』を参照してください。

- 異なる種類のエンティティーのために種々のアクセス・メソッドをアプリケーションで用意する場合は、アプリケーション・エンティティーの一部または全部にそれぞれ個別のビジネス・オブジェクト・ハンドラーが必要になることがあります。

  以下のことが可能です。

  – 一部のビジネス・オブジェクト用としてメタデータ主導型のビジネス・オブジェクト・ハンドラーを実装する汎用ビジネス・オブジェクト・ハンドラー・クラスを派生させ、他のビジネス・オブジェクト用にはビジネス・オブジェクト・ハンドラーを実装する個別のビジネス・オブジェクト・ハンドラー・クラスを派生させる。
  – コネクターがサポートするビジネス・オブジェクト定義ごとに 1 つずつ個別に複数のビジネス・オブジェクト・ハンドラー・クラスを派生させる。

  詳細については、 94 ページの『複数のビジネス・オブジェクト・ハンドラーの実装』を参照してください。

ビジネス・オブジェクト・ハンドラーを設計する際のその他の考慮事項は、ビジネス・オブジェクトの特定の動詞について、個別の処理が必要であるかどうかということです。特定の動詞で特別な処理が必要な場合は、その動詞用のカスタム・ビジネス・オブジェクト・ハンドラーを作成します。詳細については、 200 ページの『カスタム・ビジネス・オブジェクト・ハンドラーの作成』を参照してください。

メタデータ主導型のビジネス・オブジェクト・ハンドラーの実装

アプリケーションの API がメタデータ主導型のコネクターに適している場合、メタデータを含めるようにビジネス・オブジェクト定義を設計すると、メタデータ主導型のビジネス・オブジェクト・ハンドラーを実装することができます。このビジネス・オブジェクト・ハンドラーはメタデータを使用してすべての要求を処理します。アプリケーションの設計に整合性があり、またメタデータがサポートされるビジネス・オブジェクトごとに一貫性のある構文に従う場合は、ビジネス・オブジェクト・ハンドラーを完全にメタデータ主導型にすることができます。

注: メタデータおよびメタデータ主導型の設計の概要については、52 ページの『メタデータ主導型の設計を評価するためのサポート』を参照してください。

このセクションには、ビジネス・オブジェクト・ハンドラー用のメタデータ主導型の設計について、以下の情報が記載されています。

- 93 ページの『ビジネス・オブジェクトのメタデータ』
- 93 ページの『メタデータ設計の利点』

ビジネス・オブジェクトのメタデータ

ビジネス・オブジェクト定義には、異なる種類のアプリケーション固有データごとに特定のロケーションがあります。例えば、ビジネス・オブジェクト属性には Key、Foreign Key、Required、Type などのプロパティーのセットがあり、ビジネス・オブジェクト処理を駆動するのに必要な情報はこのプロパティーのセットによりビジネス・オブジェクト・ハンドラーに提供されます。さらに、AppSpecificInfo プロパティーは、アプリケーション固有の情報をビジネス・オブジェクト・ハンドラーに提供することができます。この情報では、アプリケーション内のデータにアクセスする方法およびアプリケーション・エンティティーを処理する方法を指定することができます。

AppSpecificInfo プロパティーは、ビジネス・オブジェクト定義、属性、および動詞に使用できます。表 32 に、ビジネス・オブジェクト内にアプリケーション固有の情報をエンコードするための典型的な方式をいくつか示します。

表 32. ビジネス・オブジェクトのアプリケーション情報のストレージ用サンプル・スキーム

アプリケーション固有の情報のスコープ	表ベースのアプリケーション	フォーム・ベースのアプリケーション
ビジネス・オブジェクト全体	表名	フォーム名
個々の属性	列名	フィールド名
ビジネス・オブジェクトの動詞	SQL ステートメントまたはその他の動詞処理命令	実行されるアクション

アプリケーション固有の情報を使用すると、メタデータ主導型のビジネス・オブジェクト・ハンドラーでは以下の処理を簡略化することができます。

- 送られてきたビジネス・オブジェクトの動詞を調べて、実行する操作を識別する。
- ビジネス・オブジェクト・メタデータの内容を調べて、関連したアプリケーション・エンティティー (アプリケーション表、アプリケーション・フォームなど) の名前を識別する。
- 属性メタデータの内容を調べて、フィールド、列、または属性に関するその他の情報を識別する。

ビジネス・オブジェクト定義に表名および列名が含まれている場合は、それらの名前をビジネス・オブジェクト・ハンドラーに明示的にコーディングする必要はありません。

メタデータ設計の利点

ビジネス・オブジェクトにアプリケーション情報をエンコードすると、次の 2 つのことが実現されます。

- 1 つのビジネス・オブジェクト・ハンドラー・クラスで、コネクターがサポートするすべての ビジネス・オブジェクトを対象とするすべての 操作を実行できます。サポートされるビジネス・オブジェクトごとに個別にビジネス・オブジェクト・ハンドラーをコーディングする必要はありません。
- ビジネス・オブジェクト定義の変更においては、それらの変更が既存のメタデータ構文に合致する限り、コネクターの再コーディングは必要ありません。つま

り、コネクターの再コーディングまたは再コンパイルを行わずに、ビジネス・オブジェクト定義への属性の追加、属性の除去、または属性の順序変更ができるという利点があります。

アプリケーション・エンティティーに関する情報がビジネス・オブジェクト定義内で一貫性をもってエンコードされる場合は、すべての要求ビジネス・オブジェクトをコネクター内の単一のビジネス・オブジェクト・ハンドラー・クラスで処理することができます。また、単一の `getConnectorBOHandlerForBO()` メソッドを実装するだけで、単一のビジネス・オブジェクト・ハンドラーを戻すことができ、単一の `doVerbFor()` メソッドでこのビジネス・オブジェクト・ハンドラーを実装することができます。このアプローチでは、柔軟性と、新しいビジネス・オブジェクト属性の自動サポートが提供されるため、コネクター開発に採用することを推奨します。

複数のビジネス・オブジェクト・ハンドラーの実装

1 つのアプリケーション・エンティティーのためのすべてのメタデータとビジネス・ロジックをカプセル化していない ビジネス・オブジェクト定義については、定義ごとに個別にビジネス・オブジェクト・ハンドラー・クラスが必要です。ビジネス・オブジェクト・ハンドラー基底クラスから、直接個別のハンドラー・クラスを派生させることができます。また、単一のユーティリティー・クラスを派生させ、必要に応じてサブクラスを派生させることもできます。次に、`getConnectorBOHandlerForBO()` メソッドを実装して、特定のビジネス・オブジェクト定義に対応するビジネス・オブジェクト・ハンドラーを戻すようにする必要があります。

ビジネス・オブジェクト・ハンドラーごとに 1 つの `doVerbFor()` メソッドが含まれている必要があります。複数のビジネス・オブジェクト・ハンドラーを実装する場合は、ビジネス・オブジェクト・ハンドラー・クラスごとに `doVerbFor()` メソッドを実装する必要があります。それぞれの `doVerbFor()` メソッドに、アプリケーション・エンティティーのすべての部分を処理するコード、またはビジネス・オブジェクト定義で記述されていないアプリケーション・エンティティーに対する操作を含めてください。

このアプローチを採用すると、メタデータ主導型のコネクター用に単一のビジネス・オブジェクト・ハンドラーを設計する場合よりも、保守要求が厳しくなり開発時間が長くなります。したがって、可能ならこのアプローチは避けてください。しかし、アプリケーションで異なる種類のエンティティーのために種々のアクセス・メソッドを用意する場合は、エンティティー固有の複数のビジネス・オブジェクト・ハンドラーのコーディングは避けられません。

ビジネス・オブジェクト・ハンドラー基底クラスの拡張

Java コネクター・ライブラリーはビジネス・オブジェクト・ハンドラー基底クラス `CWConnectorBOHandler` を提供します。この基底クラスには、`doVerbFor()` メソッドを含む要求処理取り扱いのためのメソッドが組み込まれています。ビジネス・オブジェクト・ハンドラーを作成するには、このビジネス・オブジェクト・ハンドラー基底クラスを拡張して、その抽象メソッド `doVerbFor()` を実装する必要があります。Java コネクター・ライブラリー固有の情報については、178 ページの『Java ビジネス・オブジェクト・ハンドラーの基底クラスの拡張』を参照してください。

要求の処理

ビジネス・オブジェクト・ハンドラー・クラスを派生させたら、ビジネス・オブジェクト・ハンドラー・メソッド doVerbFor() を実装する必要があります。コネクターがサポートするビジネス・オブジェクト用の要求処理は、doVerbFor() メソッドで提供されます。始動すると、コネクター・フレームワークは getConnectorBOHandlerForBO() を呼び出して、コネクターがサポートするビジネス・オブジェクト定義ごとに実装されているビジネス・オブジェクト・ハンドラーを取得します。

重要: すべてのコネクターに必ず ビジネス・オブジェクト・ハンドラー・メソッド doVerbFor() を実装する必要があります。このメソッドは Retrieve 動詞を実装します。このメソッドと動詞は、コネクターが要求処理を実行しない 場合にも実装する必要があります。

このセクションには、doVerbFor() メソッドを実装する方法について、以下の情報が記載されています。

- 『doVerbFor() の基本ロジック』
- 97 ページの『動詞の実装に関する一般推奨事項』

doVerbFor() の基本ロジック

Java コネクターの場合、CWConnectorBOHandler クラスで doVerbFor() メソッドが定義されています。これは定義済みの 抽象メソッドです。 doVerbFor() メソッドは、通常、要求処理の基本ロジックに従います。

図 27 は、このメソッドの基本ロジックのフローチャートです。

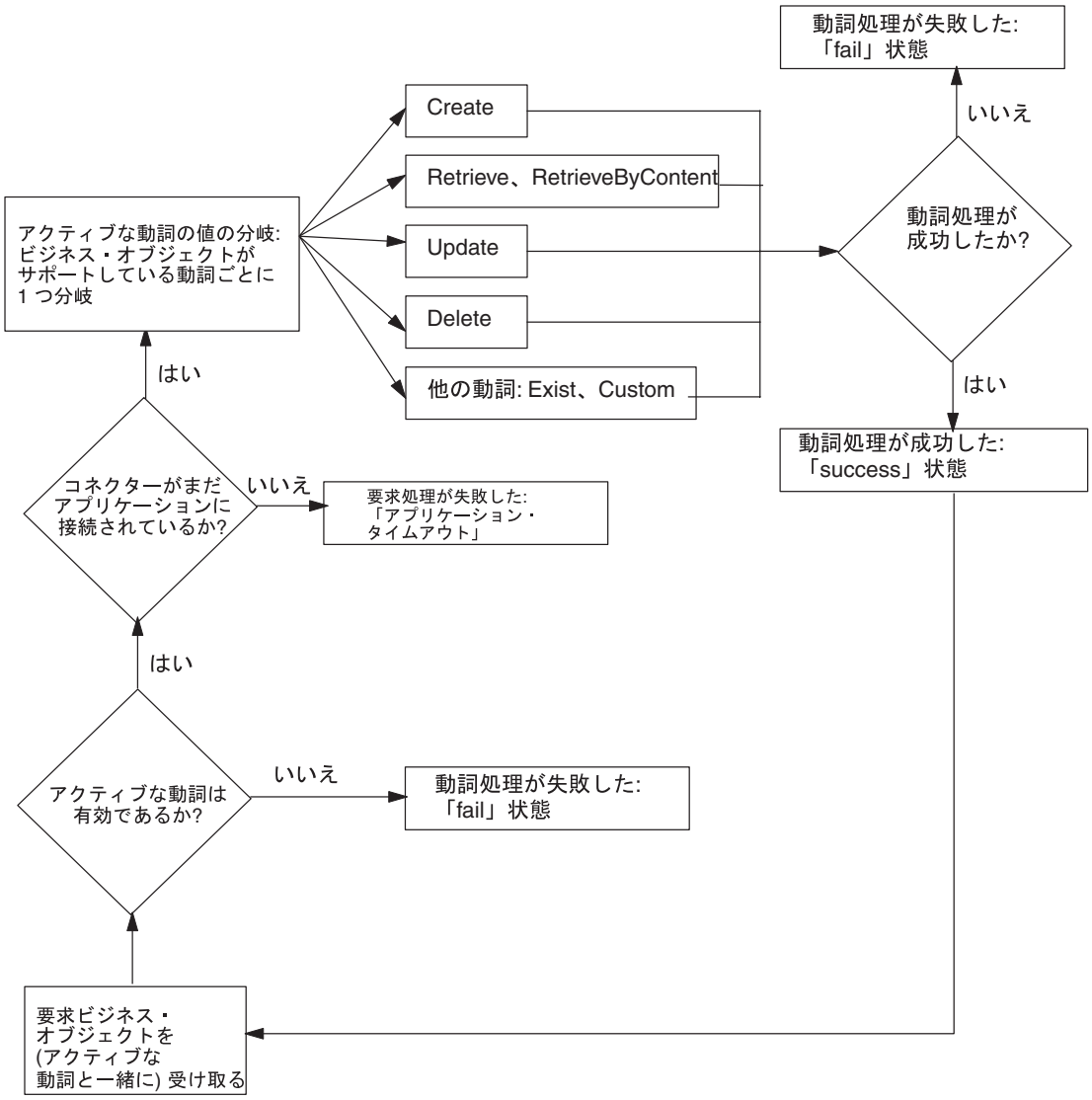

図 27. doVerbFor() の基本ロジックのフローチャート

この doVerbFor() 基本ロジックのインプリメンテーションについては、 179 ページの『doVerbFor() メソッドの実装』を参照してください。

コネクター・フレームワークは、要求を受信すると、要求ビジネス・オブジェクトのビジネス・オブジェクト定義に関連したビジネス・オブジェクト・ハンドラー・クラスの doVerbFor() メソッドを呼び出します。コネクター・フレームワークは、要求ビジネス・オブジェクトをこの doVerbFor() メソッドに引き渡します。表 33 は、doVerbFor() メソッドがコネクター・フレームワークから要求ビジネス・オブジェクトを受信すると実行するタスクの要約です。

表 33. doVerbFor() メソッドのタスク

ビジネス・オブジェクト・ハンドラーのタスク	詳細情報
1. 要求ビジネス・オブジェクト内のアクティブな動詞に基づいて、実行する動詞処理を決定する。	98 ページの『動詞アクションの実行』
2. 要求ビジネス・オブジェクトから情報を取得して、操作要求を作成し、アプリケーションへ送信する。	122 ページの『ビジネス・オブジェクトの処理』

動詞の実装に関する一般推奨事項

このセクションには、doVerbFor() メソッドを実装する方法について、以下の一般推奨事項が記載されています。

- 『動詞安定度』
- 『トランザクション・サポート』
- 『ObjectEventId 属性』

動詞安定度

ビジネス・オブジェクト内の動詞は、要求と応答の全サイクルを通じて安定していなければなりません。コネクターが要求を受信したとき、InterChange Server に戻される階層型ビジネス・オブジェクトは、元の要求ビジネス・オブジェクトと同じ動詞を持っている必要があります。ただし、元の要求で設定されていなかった 子ビジネス・オブジェクトの動詞は除きます。

子ビジネス・オブジェクト内の動詞は、要求ビジネス・オブジェクトに設定されている場合とされていない場合があります。

- 動詞が子ビジネス・オブジェクト内に設定されている場合、コネクターは、トップレベルのビジネス・オブジェクトの動詞に関係なく、子動詞が指示する操作を実行する必要があります。
- 子ビジネス・オブジェクト要求に動詞が設定されていない 場合は、コネクターは、子動詞を null として、処理を行わないか、トップレベル・ビジネス・オブジェクトの動詞に子動詞を設定するか、またはコネクターが実行する必要のある操作に動詞の値を設定します。

トランザクション・サポート

ビジネス・オブジェクト要求全体を単一のトランザクションにラップする必要があります。言い換えると、トップレベル・ビジネス・オブジェクトのすべての Create、Update、および Delete トランザクションと、そのすべての子を、単一のトランザクションにラップする必要があります。トランザクションの存続期間中に障害が検出された場合は、トランザクション全体のロールバックが必要です。

例えば、トップレベル・ビジネス・オブジェクトに関する Create 操作が成功しても、子ビジネス・オブジェクトの 1 つのトランザクションが失敗した場合は、コネクター・アプリケーション固有のコンポーネントは Create トランザクション全体を前の状態にロールバックしなければなりません。この場合、コネクターのアプリケーション固有のコンポーネントは動詞メソッドから失敗を戻す必要があります。

ObjectEventId 属性

ObjectEventId 属性は、システム内のイベント・トリガー処理フローを識別するために IBM WebSphere Business Integration システムで使用されます。さらに、この属性は、複数の要求と応答の間、子ビジネス・オブジェクトを追跡するためにも使用されます。これは、階層型ビジネス・オブジェクト要求内の子ビジネス・オブジェクトの位置が応答ビジネス・オブジェクト内の子ビジネス・オブジェクトの位置と異なる場合があるためです。

コネクターは、親ビジネス・オブジェクトまたはその子ビジネス・オブジェクトの ObjectEventId 属性の値を取り込む必要はありません。ビジネス・オブジェクトに ObjectEventId 属性の値がない場合は、IBM WebSphere Business Integration システムがその値を生成します。コネクターが ObjectEventId 値を生成する場合は、ソース・コネクターがイベント通知機構の一部としてそれを行います。

要求ビジネス・オブジェクトを処理するときは、コネクターは、ObjectEventId 値を、要求ビジネス・オブジェクトと応答ビジネス・オブジェクト間の階層型ビジネス・オブジェクトのすべての レベルで保存する必要があります。コネクター・メソッドが子ビジネス・オブジェクト ObjectEventId の値を変更すると、IBM WebSphere Business Integration システムは子ビジネス・オブジェクトを正しく追跡できないことがあります。

イベント通知機構での ObjectEventId の生成については、134 ページの『イベント ID』を参照してください。

動詞アクションの実行

コネクターが処理する IBM WebSphere Business Integration システムで予定されている標準動詞は、Create、Retrieve、Update、および Delete です。IBM では、表 34 の『詳細情報の参照先』の欄にリストするセクションで説明されている標準的な振る舞いに応じて、これらの動詞をインプリメントすることを推奨します。これらのセクションには、標準的な振る舞い、実装の注意、および適切な結果状況値に関する情報があります。

表 34 は、IBM WebSphere Business Integration システムが定義する標準動詞のリストです。doVerbFor() メソッドでは、これらの動詞をその応用に適した形で実装する必要があります。

表 34. doVerbFor() メソッドで実装される動詞

動詞	説明	詳細情報
Create	アプリケーションに新しいエンティティーを作成します。	99 ページの『Create 動詞の処理』
Retrieve	キー値を使用して完全なビジネス・オブジェクトを戻します。	103 ページの『Retrieve 動詞の処理』
RetrieveByContent	非キー値を使用して完全なビジネス・オブジェクトを戻します。	109 ページの『RetrieveByContent 動詞の処理』
Update	アプリケーションの 1 つまたは複数のフィールドの値を変更します。	111 ページの『Update 動詞の処理』
Delete	アプリケーションからエンティティーを除去します。この操作は実際の物理的削除でなければなりません。	119 ページの『Delete 動詞の処理』
Exists	アプリケーションにエンティティーが存在するかチェックします。	121 ページの『Exists 動詞の処理』
カスタムの動詞	アプリケーション固有の操作を実行します。	なし

注: 表 34 の「詳細情報」の欄にリストしたセクションには動詞メソッドの推奨される振る舞いが示されていますが、実際のコネクターでは、特定のアプリケーションをサポートするために、動詞処理をいくらか変えて実装する必要がある場合があります。コネクター・フレームワークがコネクターの doVerbFor() メソ

ッドに要求ビジネス・オブジェクトを引き渡すと、doVerbFor() メソッドは、必要とされる方法で動詞処理を実装することができます。実際に開発される動詞処理コードは、この章の推奨事項に限定する必要はありません。

InterChange Server

InterChange Server が統合ブローカーの場合、開発者が独自のコラボレーションを設計すれば、必要な任意のカスタム動詞を実装することができます。実際のコラボレーションとコネクターは、標準リストの動詞に限定する必要はありません。

InterChange Server の終り

この基本的な動詞処理ロジックは、次のステップで構成されます。

1. 要求ビジネス・オブジェクトから動詞を取得します。

   doVerbFor() メソッドは、最初に getVerb() メソッドでビジネス・オブジェクトからアクティブな動詞を取得する必要があります。Java コネクターの場合は、getVerb() は CWConnectorBusObj クラスで定義されています。
2. 動詞の操作を実行します。

   ビジネス・オブジェクト・ハンドラーでは、次のいずれかの方法で doVerbFor() メソッドを設計することができます。
   - サポートされる動詞ごとに、直接 doVerbFor() メソッド内に動詞処理を実装する。動詞処理をモジュール化して、動詞操作ごとに doVerbFor() から呼び出される個別の動詞メソッドに実装することができます。また、サポートされる動詞でない動詞の場合は、戻り状況記述子にメッセージを戻し、状況を「fail」にして、メソッドが適切なアクションを実行する必要があります。
   - メタデータ主導型の doVerbFor() メソッドを使用して、すべての動詞処理を同じメソッドで扱う。

Create 動詞の処理

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから Create 動詞を取得する場合、次のように、ビジネス・オブジェクト定義で型を指定された新しいアプリケーション・エンティティーが作成されるようにする必要があります。

- フラット・ビジネス・オブジェクトの場合は、Create 動詞は指定されたエンティティーの作成が必要であることを示します。
- 階層型ビジネス・オブジェクトの場合は、Create 動詞は 1 つまたは複数のアプリケーション・エンティティー (ビジネス・オブジェクト全体にマッチする) の作成が必要であることを示します。

ビジネス・オブジェクト・ハンドラーは、新しいアプリケーション・エンティティーのすべての値を要求ビジネス・オブジェクトの属性値に設定する必要があります。要求ビジネス・オブジェクト内のすべての必須属性に値が割り当てられるようにするために、initAndValidateAttributes() メソッドを呼び出すことができます。このメソッドは、値が設定されていないそれぞれの必須属性に、その属性のデフォルト値を割り当てます (UseDefaults コネクター構成プロパティーが true に設定されている場合)。 initAndValidateAttributes() メソッドは、

CWConnectorUtil クラスで定義されます。initAndValidateAttributes() は、アプリケーションで Create 操作を実行する*前に* 呼び出してください。

注: 表ベースのアプリケーションの場合は、アプリケーション・エンティティー全体をアプリケーション・データベース内に作成する必要があります。通常、アプリケーション・エンティティーは、要求ビジネス・オブジェクトのビジネス・オブジェクト定義に関連したデータベース表内の新しい行として作成します。

このセクションには、Create 動詞の処理に役立つ以下の情報が記載されています。

- 『Create 動詞の標準処理』
- 101 ページの『Create 動詞操作の実装』
- 102 ページの『Create 動詞処理の結果状況』

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構成を採用すると、Create メソッドは Create 動詞の処理を扱うことになります。

Create 動詞の標準処理

以下のステップは、Create 動詞の標準処理の概略です。

1. トップレベル・ビジネス・オブジェクトに対応するアプリケーション・エンティティーを作成します。
2. アプリケーション・エンティティー用の基本キーを次のように処理します。
   - アプリケーションが独自の基本キーを生成する場合は、トップレベル・ビジネス・オブジェクトに挿入するそれらのキーを取得します。
   - アプリケーションが独自の基本キーを生成しない 場合は、要求ビジネス・オブジェクトからのキーをアプリケーション・エンティティーの適切なキー列に挿入します。
3. 任意の第 1 レベル子ビジネス・オブジェクトの外部キー属性をトップレベル基本キーの値に設定します。
4. 第 1 レベル子ビジネス・オブジェクトに対応するアプリケーション・エンティティーを再帰的に作成し、ビジネス・オブジェクト階層内の後続のすべてのレベルですべての子ビジネス・オブジェクトの再帰的作成を続けます。

図 28 では、動詞メソッドが、子ビジネス・オブジェクト A、B、および C の外部キー属性 (FK) をトップレベル基本キーの値 (PK1) に設定します。次に、メソッドは、子ビジネス・オブジェクト D および E の外部キー属性を、それらのビジネス・オブジェクトの親ビジネス・オブジェクト (オブジェクト B) の基本キーの値 (PK3) に再帰的に設定します。

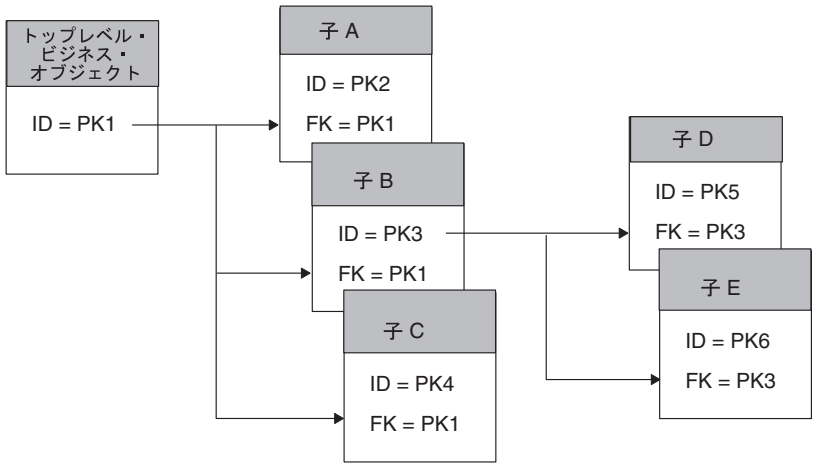

図 28. 親/子関係の作成

Create 動詞操作の実装

Create 操作の典型的な実装では、まずトップレベル・ビジネス・オブジェクトの全探索を行い、ビジネス・オブジェクトの単純な属性を処理します。ビジネス・オブジェクトから属性の値が取得され、アプリケーション固有のアクション (API 呼び出し、SQL ステートメントなど) が生成され、そのアクションにより、トップレベル・ビジネス・オブジェクトを表すエンティティーがアプリケーションに挿入されます。このトップレベル・エンティティーが作成されると、動詞操作のステップは次のようになります。

1. アプリケーションからエンティティーの基本キーを取得します。
2. キーを使用して第 1 レベル子ビジネス・オブジェクトの外部キー属性を親基本キーの値に設定します。
3. 各子ビジネス・オブジェクトの動詞を Create に設定し、子ビジネス・オブジェクトを表すアプリケーション・エンティティーを再帰的に作成します。

子ビジネス・オブジェクトを作成するうえで推奨されるアプローチは、子エンティティーを再帰的に作成するサブメソッドを設計することです。このサブメソッドは、ビジネス・オブジェクトの全探索を行い、型 OBJECT の属性を探します。サブメソッドは、オブジェクトである属性を見つけると、メインの Create メソッドを呼び出して子エンティティーを作成します。

メインのメソッドが基本キー値を提供する方法は 1 通りではありません。例えば、メインの Create メソッドが親ビジネス・オブジェクトをサブメソッドに渡し、次にサブメソッドが親ビジネス・オブジェクトから基本キーを取得して、子ビジネス・オブジェクトに外部キーを設定することができます。また、メインのメソッドが親オブジェクトの全探索を行い、第 1 レベルの子を見つけ、子ビジネス・オブジェクトに外部キー属性を設定し、次に子ごとにサブメソッドを呼び出すこともできます。

いずれの場合にも、メインの Create メソッドとそのサブメソッドの相互作用により、親ビジネス・オブジェクトとその第 1 レベルの子との相互依存関係が設定されます。外部キーが設定されると、次の操作が可能になります。

- 新しい行をアプリケーションに挿入する。
- 次のレベルの子ビジネス・オブジェクト用の外部キーを設定する。
- 子エンティティーを作成する。
- ビジネス・オブジェクト階層を下降し、処理する子ビジネス・オブジェクトがなくなるまで、子エンティティーを再帰的に作成する。

注: 表ベースのアプリケーションの場合は、アプリケーションのデータベース・スキーマおよびアプリケーション固有のビジネス・オブジェクトの設計に応じて、トップレベル・オブジェクトとその子との関係を設定するステップの順序が変わる場合があります。例えば、階層型ビジネス・オブジェクトの外部キーがトップレベル・ビジネス・オブジェクト内にある場合、動詞操作ではトップレベル・ビジネス・オブジェクトを処理する*前に* すべての子ビジネス・オブジェクトの処理が必要になる場合があります。子エンティティーがアプリケーション・データベースに挿入され、これらのエンティティーの基本キーが戻されたときにのみ、トップレベル・ビジネス・オブジェクトの処理が可能になり、アプリケーション・データベースに挿入できるようになります。したがって、コネクター動詞メソッドを実装する際は、アプリケーション・データベース内のデータの構造を考慮する必要があります。

Create 動詞処理の結果状況

Create 操作は、表 35に示す結果状況値の 1 つを戻さなければなりません。

表 35. Java Create 動詞処理で可能な結果状況

Create 条件	Java 結果状況
Create 操作が正常に実行され、アプリケーションが新しいキー値を生成すると、コネクターは以下の処理を行います。 • 新しいキー値をビジネス・オブジェクトに充てんする。このビジネス・オブジェクトが要求ビジネス・オブジェクト・パラメーターを介してコネクター・フレームワークに戻されます。 • 「Value Changed」結果状況を戻して、コネクターがビジネス・オブジェクトを変更したことを示す。	VALCHANGE
Create 操作が正常に実行され、アプリケーションが新しいキー値を生成*しない* 場合も、コネクターは単に「Success」を戻すことがあります。 アプリケーション・エンティティーがすでに存在する場合は、コネクターは次のアクションの*いずれか* を実行することがあります。	SUCCEED
• Create 操作を失敗させる。	FAIL
• アプリケーション・エンティティーがすでに存在することを示す結果状況を戻す。	VALDUPES
Create 操作が失敗すると、動詞操作は以下の処理を行います。 • 戻り状況記述子に失敗に関する情報を充てんする。 • 「Fail」結果状況を戻す。	FAIL

注: コネクター・フレームワークは、VALCHANGE 結果状況を受け取ると、InterChange Server への応答にビジネス・オブジェクトを含めます。詳細については、 194 ページの『動詞処理応答の送信』を参照してください。

Retrieve 動詞の処理

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから Retrieve 動詞を取得する場合、次のように、ビジネス・オブジェクト定義で型を指定された既存のアプリケーション・エンティティーが取得されるようにする必要があります。

- フラット・ビジネス・オブジェクトの場合は、Retrieve 動詞は指定されたエンティティーがキー値によって取得されることを示します。この動詞操作は、アプリケーション・エンティティーの現行値が入っているビジネス・オブジェクトを戻します。
- 階層型ビジネス・オブジェクトの場合は、Retrieve 動詞は、トップレベル・ビジネス・オブジェクトのキー値による 1 つまたは複数のアプリケーション・エンティティー (ビジネス・オブジェクト全体にマッチする) の取得が必要であることを示します。この動詞操作は、階層内の各ビジネス・オブジェクトのすべての単純属性が対応するエンティティー属性値と一致するビジネス・オブジェクトを戻します。ここで、それぞれの子ビジネス・オブジェクト配列内の個別ビジネス・オブジェクトの数は、アプリケーション内の子エンティティーの数と一致します。

注: 表ベースのアプリケーションの場合、アプリケーション・エンティティー全体をアプリケーション・データベースから取得する必要があります。

Retrieve 動詞の場合、ビジネス・オブジェクト・ハンドラーは要求ビジネス・オブジェクトからキー値を取得します。これらのキー値は、アプリケーション・エンティティーを一意的に識別します。次に、ビジネス・オブジェクト・ハンドラーは、これらのキー値を使用して、1 つのアプリケーション・エンティティーに関連したすべてのデータを取得します。コネクターは、すべての子オブジェクトを含むエンティティーの階層イメージ全体を取得します。この型の取得操作を *変更後イメージ取得* と呼ぶことがあります。

重要: すべてのコネクターは、Retrieve 動詞の動詞処理で `doVerbFor()` メソッドを実装する必要があります。この要件は、コネクターが要求処理を実行しない場合にも適用されます。

データを取得するもう 1 つの方法は、特定のアプリケーション・レコードを一意的に定義する値を含まない非キー属性値のサブセットを使用して照会を行うことです。この型の取得処理は、RetrieveByContent 動詞メソッドで実行されます。非キー値による取得については、109 ページの『RetrieveByContent 動詞の処理』を参照してください。

このセクションには、Retrieve 動詞の処理に役立つ以下の情報が記載されています。

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構成を採用すると、Retrieve メソッドは Retrieve 動詞の処理を扱うことになります。

Retrieve 動詞の標準処理

以下のステップは、Retrieve 動詞の標準処理の概略です。

1. 要求ビジネス・オブジェクトと同じ型の新しいビジネス・オブジェクトを作成します。この新しいビジネス・オブジェクトは、要求ビジネス・オブジェクトの取得されたコピーを保持する応答ビジネス・オブジェクト です。
2. 新しいトップレベル・ビジネス・オブジェクトの基本キーを、要求ビジネス・オブジェクトのトップレベル・キーの値に設定します。
3. トップレベル・ビジネス・オブジェクトのアプリケーション・データを取得し、応答トップレベル・ビジネス・オブジェクトの単純属性を充てんします。
4. トップレベル・エンティティーに関連したすべての アプリケーション・データを取得し、必要に応じて子ビジネス・オブジェクトを作成して充てんします。

注: デフォルトでは、階層型ビジネス・オブジェクト内のすべての 子オブジェクトのアプリケーション・データを取得できないと、Retrieve メソッドは失敗を戻します。この振る舞いは構成可能にすることができます。108 ページの『欠落した子オブジェクトを無視する Retrieve の構成』の説明を参照してください。

Retrieve 動詞操作の実装

典型的な Retrieve 操作では、以下のメソッドの 1 つを使用することができます。

- 該当するビジネス・オブジェクトのビジネス・オブジェクト定義から新しい応答ビジネス・オブジェクトを作成し、この新しいビジネス・オブジェクトにトップレベル基本キーを設定する。動詞操作はトップレベル基本キーを使用して、トップレベル・エンティティーに関連したすべてのデータを取得できます。
- トップレベル・ビジネス・オブジェクトからすべての子ビジネス・オブジェクトを枝取りして処理を開始する。枝取りしたオブジェクトのトップレベル・キーを使用して、動詞操作はトップレベル・データおよびすべての関連データを取得できます。

これらのアプローチのゴールはどれも同じです。つまり、トップレベル・ビジネス・オブジェクトで処理を開始し、ビジネス・オブジェクト階層全体を再構築します。この型の実装では、データベース内にもはや存在しない要求ビジネス・オブジェクトのすべての 子が除去されたり、応答ビジネス・オブジェクトに戻されることがなくなります。また、この実装では、階層応答ビジネス・オブジェクトがアプリケーション・エンティティーのデータベース状態に正確に一致することになります。Retrieve 操作は、レベルごとに要求ビジネス・オブジェクトを再構築し、エンティティーの現行データベース表現を正確に反映させます。

例: Retrieve 操作

Retrieve 操作では、統合ブローカーが、アプリケーション・エンティティーに関連したデータの完全なセットを要求します。要求ビジネス・オブジェクトには、以下の任意のセットを含めることができます。

- ビジネス・オブジェクト定義に子が含まれていても子を除外したトップレベル・ビジネス・オブジェクト
- トップレベル・ビジネス・オブジェクトと定義済みの子の一部を含むビジネス・オブジェクト
- すべての子ビジネス・オブジェクトを含む完全な階層型ビジネス・オブジェクト

図 29 は、Contact エンティティーの要求ビジネス・オブジェクトを示します。Contact ビジネス・オブジェクトには、ContactProfile 属性のための複数のカーディナリティー配列が含まれています。この要求ビジネス・オブジェクトでは、ContactProfile ビジネス・オブジェクト配列に 2 つの子ビジネス・オブジェクトが含まれています。

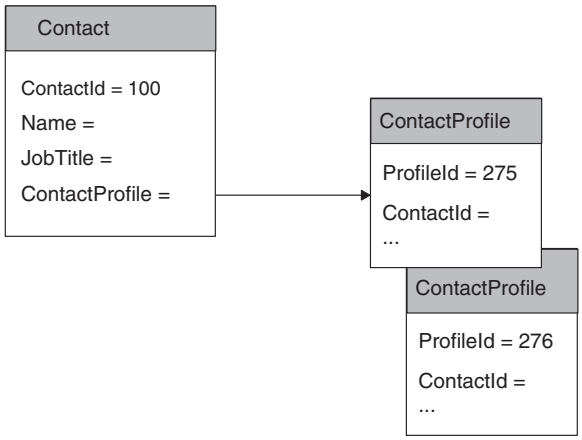

図 29. Retrieve 要求用のビジネス・オブジェクト例

Contact ビジネス・オブジェクトおよび ContactProfile ビジネス・オブジェクトに関連したアプリケーション表は、図 30 のようになります。この例には、表間の外部キーの関係も示されています。この例で見られるように、contact_profile 表には 100 という値の ContactId 行がありますが、この行は 図 28 の Contact 要求ビジネス・オブジェクトには反映されていません。

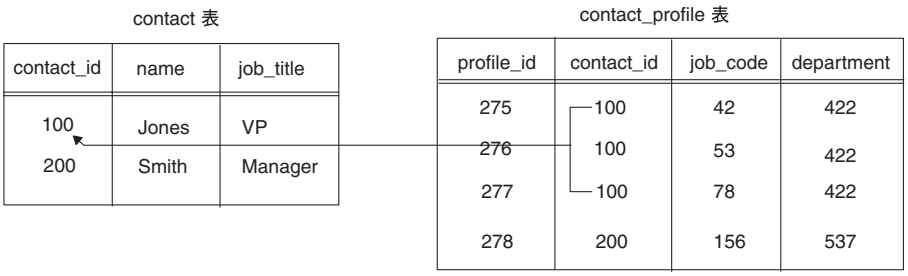

図 30. 表間の外部キー関係

Retrieve 操作では、Contact ビジネス・オブジェクト内の基本キー (100) を使用して、応答ビジネス・オブジェクトの単純属性のためのデータ (Name 属性と JobTitle 属性の値) を取得します。正しい数の子ビジネス・オブジェクトが取得されるようにするには、動詞操作で新しいビジネス・オブジェクトを作成するか、あるいは既存の要求ビジネス・オブジェクトから子オブジェクトの枝取りを行う必要があります。図 30 の表の場合、Retrieve 操作では profile_id 値を 277 として新

しい ContactProfile ビジネス・オブジェクトを contact_profile 行に作成する必要があります。このようにすると、Retrieve 操作はアプリケーション・エンティティーの現在の状態に基づいてすべての 配列を正しく作成し移植します。

子オブジェクトの取得

トップレベル・エンティティーに関連したエンティティーを取得するために、Retrieve 操作ではアプリケーション API を使用できる場合があります。

- API でアプリケーション・エンティティー間の関係をナビゲートし、すべての関連データを戻すのが理想的です。この場合、動詞操作で関連データを子ビジネス・オブジェクトとしてカプセル化することができます。
- API で関連エンティティーに関する情報が準備されない 場合は、アプリケーションにアクセスして (例えば、生成された SQL ステートメントを使用して)、関連データを取得する必要があります。SQL ステートメントで外部キーを使用してアプリケーション表をナビゲートすることができます。

ビジネス・オブジェクト定義内の属性のアプリケーション固有の情報に外部キーに関する情報が含まれている場合は、動詞操作でこの情報を使用して、アプリケーションにアクセスするためのコマンド (SQL ステートメントなど) を生成することができます。例えば、ContactProfile 子ビジネス・オブジェクトの外部キー属性のアプリケーション固有の情報は次のように指定します。

- 親表: contact
- 子表の外部キー列: contact_id
- 子ビジネス・オブジェクトで外部キーとして機能する基本キー値が入っている親ビジネス・オブジェクトの中の属性: ContactId

図 31 は、Contact ビジネス・オブジェクトの基本キー属性および ContactProfile 子ビジネス・オブジェクトの基本および外部キー属性に関するアプリケーション固有の情報の例です。

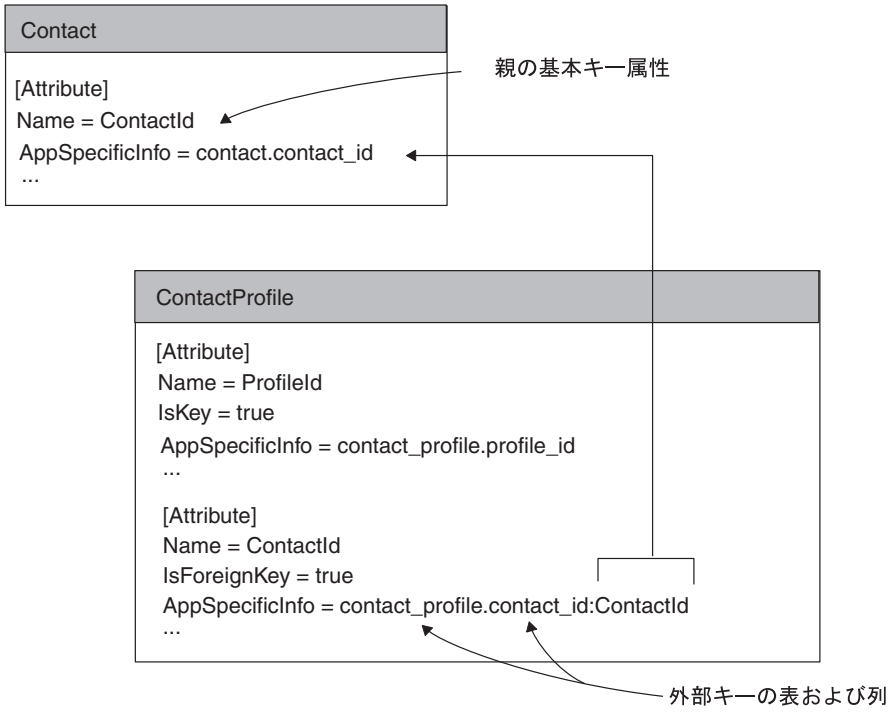

図 31. ビジネス・オブジェクトの外部キー関係

動詞操作では、アプリケーション固有の情報を使用して、子表の名前 (`contact_profile`) と、子表の中の外部キー列 (`contact_id`) を見つけることができます。また、動詞操作では、親ビジネス・オブジェクト内の基本キー属性 (`ContactId`) の値 (100) を取得して、子ビジネス・オブジェクトの外部キーの値を見つけることができます。動詞操作でこの情報を使用して、親キーに関連した子表内のすべてのレコードを取得する SQL SELECT ステートメントを生成することができます。欠落している `contact_profile` 行に関連したデータを取得する SELECT ステートメントは次のようになります。

```
SELECT profile_id, job_code, department
FROM contact_profile
WHERE contact_id = 100
```

この SELECT ステートメントは、`contact_profile` 表の例 (図 32) から 3 行を戻します。

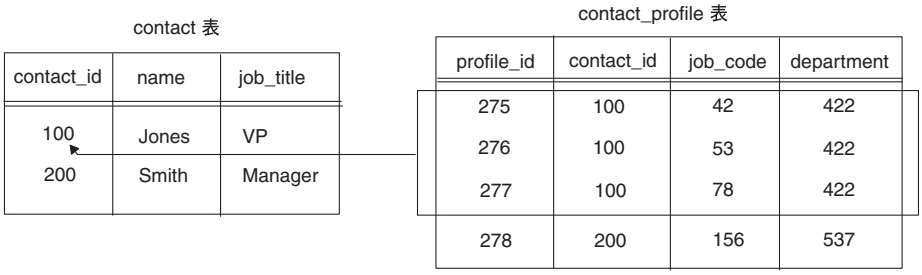

図 32. サンプル Retrieve 操作用の SELECT ステートメントの結果

Retrieve 操作が複数の行を戻した場合は、それぞれの行が子ビジネス・オブジェクトになります。動詞操作は取得された行を次のように処理します。

1. 行ごとに、正しい型の新しい子ビジネス・オブジェクトを作成する。
2. 関連した行について SELECT ステートメントが戻す値に基づいて、新しい子ビジネス・オブジェクトに属性を設定する。
3. 子ごとにビジネス・オブジェクトを作成し属性を設定して、子ビジネス・オブジェクトのすべての子を再帰的に取得する。
4. 親ビジネス・オブジェクトの複数カーディナリティー属性に子ビジネス・オブジェクトの配列を挿入する。

2 つのサンプル表に関する Retrieve 操作の場合の応答ビジネス・オブジェクトは図 33 のようになります。動詞操作は現行データベース・エンティティーを取得し、階層型ビジネス・オブジェクトに子を追加しています。

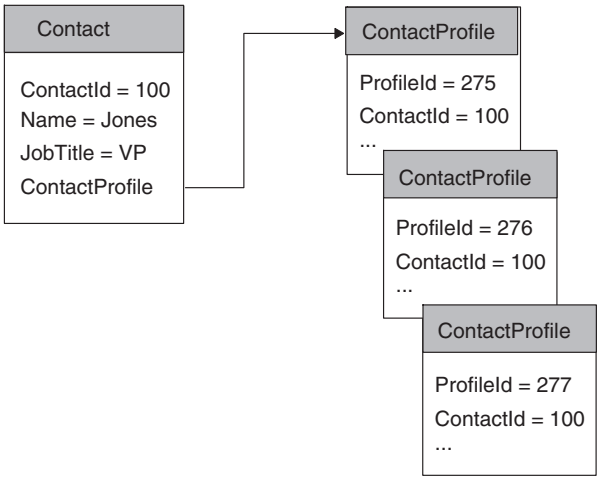

図 33. Retrieve 要求のサンプルに対するビジネス・オブジェクトの応答

欠落した子オブジェクトを無視する Retrieve の構成

デフォルトでは、階層型ビジネス・オブジェクト内の子ビジネス・オブジェクトの完全なセットについてアプリケーション・データを取得できない場合に、Retrieve 操作は失敗を戻します。しかし、ビジネス・オブジェクト内の 1 つまたは複数の子がアプリケーションに見つからない場合のコネクターの振る舞いを構成可能にするように、動詞操作を実装することができます。

そのためには、`IgnoreMissingChildObject` という名前のコネクター固有の構成プロパティーを定義します。このプロパティーの値は True と False です。 Retrieve 操作はこのプロパティーの値を取得して、欠落している子ビジネス・オブジェクトを処理する方法を決定します。プロパティーが True の場合、Retrieve 操作は子ビジネス・オブジェクトを見つけられなかったとき、単に配列内の次の子に進みます。この場合、トップレベル・オブジェクトの取得に成功すれば、その子の取得が成功したかどうかにかかわらず、動詞操作は VALCHANGE を戻す必要があります。

Retrieve 動詞処理の結果状況

Retrieve 操作は、表 36 に示す結果状況の 1 つを戻さなければなりません。

表 36. Java Retrieve 動詞処理で可能な結果状況

Retrieve 条件	Java 結果状況
Retrieve 操作が正常に実行されると、次の処理が行われます。 • すべての子ビジネス・オブジェクトを含むビジネス・オブジェクト階層全体を充てんする。このビジネス・オブジェクトが要求ビジネス・オブジェクト・パラメーターを介してコネクター・フレームワークに戻されます。 • 「Value Changed」結果状況を戻して、コネクターがビジネス・オブジェクトを変更したことを示す。	VALCHANGE
IgnoreMissingChildObject コネクター・プロパティーが True の場合、トップレベル・オブジェクトの取得に成功すれば、その子の取得に成功したかどうかにかかわらず、Retrieve 操作はビジネス・オブジェクトについて「Value Changed」結果状況を戻します。	VALCHANGE
ビジネス・オブジェクトが表すエンティティーがアプリケーションに存在しない 場合は、コネクターは「Fail」ではなく特別な結果状況を戻します。	BO_DOES_NOT_EXIST
要求ビジネス・オブジェクトがトップレベル・ビジネス・オブジェクトのキーを提供しない 場合は、Retrieve 操作は次のアクションのいずれか を実行できます。 • 要求が失敗した原因に関する情報を戻り状況記述子に充てんし、「Fail」結果状況を戻す。 • RetrieveByContent メソッドを呼び出し、トップレベル・ビジネス・オブジェクトの内容を使用して取得を行う。	FAIL

注: コネクター・フレームワークは、VALCHANGE 結果状況を受け取ると、InterChange Server への応答にビジネス・オブジェクトを含めます。詳細については、194 ページの『動詞処理応答の送信』を参照してください。

RetrieveByContent 動詞の処理

統合ブローカーでは、属性値のセットがありアプリケーション・エンティティーを一意的に識別するキー属性のない、ビジネス・オブジェクトの取得が必要な場合があります。そのような取得を「非キー値による取得」または「内容による取得」と呼びます。例えば、ビジネス・オブジェクト・ハンドラーが動詞 RetrieveByContent で Customer ビジネス・オブジェクトを受け取り、非キー属性の Name および City が Smith および San Diego に設定されている場合、RetrieveByContent 操作は Name 属性および City 属性の値に一致するカスタマー・エンティティーの取得を試みることができます。

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから RetrieveByContent 動詞を取得する場合、次のように、ビジネス・オブジェクト定義で型を指定された既存のアプリケーション・エンティティーが取得されるようにする必要があります。

- フラット・ビジネス・オブジェクトの場合は、RetrieveByContent 動詞は指定されたエンティティーがその非キー値によって取得されることを示します。この動詞操作は、アプリケーション・エンティティーの現行値が入っているビジネス・オブジェクトを戻します。
- 階層型ビジネス・オブジェクトの場合は、RetrieveByContent 動詞は、トップレベル・ビジネス・オブジェクトの非キー値により 1 つまたは複数のアプリケーション・エンティティー (ビジネス・オブジェクト全体にマッチする) が取得されることを示します。この動詞操作は、階層内の各ビジネス・オブジェクトのすべての単純属性が対応するエンティティー属性値と一致するビジネス・オブジェクトを戻します。ここで、それぞれの子ビジネス・オブジェクト配列内の個別ビジネス・オブジェクトの数は、アプリケーション内の子エンティティーの数と一致します。

このセクションには、RetrieveByContent 動詞の処理に役立つ以下の情報が記載されています。

- 『RetrieveByContent 動詞操作の実装』
- 『RetrieveByContent 処理の結果状況』

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構成を採用すると、RetrieveByContent メソッドが RetrieveByContent 動詞の処理を扱うことになります。

RetrieveByContent 動詞操作の実装

RetrieveByContent は、キー値ではなく非キー値のサブセットを使用してアプリケーション・データを取得することを除いて、Retrieve 動詞と同様に機能します。この動詞の最も堅固な実装では、トップレベル・ビジネス・オブジェクトとその子ビジネス・オブジェクトが独立して RetrieveByContent 動詞をサポートします。しかし、すべてのアプリケーション API が非キー値により階層型ビジネス・オブジェクトの取得を行えるわけではありません。

より一般的な実装では、トップレベル・ビジネス・オブジェクトのみに RetrieveByContent のサポートが用意されます。トップレベル・ビジネス・オブジェクトが非キー値による取得をサポートし、この内容による取得が正常に実行される場合は、RetrieveByContent 操作は要求ビジネス・オブジェクトに一致するエンティティーのためのキーを取得できます。次に、動詞操作は Retrieve 操作を実行して、完全なビジネス・オブジェクトを取得します。

RetrieveByContent 操作で使用する属性を指定する必要があります。そのためには、属性のアプリケーション固有の情報を実装して、RetrieveByContent 操作で使用する値が入っている属性を指定するか、操作の結果として値を受け取ることができます。

RetrieveByContent 処理の結果状況

RetrieveByContent 操作は、表 37 に示す結果状況値の 1 つを戻さなければなりません。

表 37. Java RetrieveByContent 動詞処理で可能な結果状況

RetrieveByContent 条件	Java 結果状況
RetrieveByContent 操作で照会に一致する単一のエンティティーが見つかった場合は、次の処理が行われます。 • すべての子ビジネス・オブジェクトを含むビジネス・オブジェクト階層全体を充てんする。このビジネス・オブジェクトが要求ビジネス・オブジェクト・パラメーターを介してコネクター・フレームワークに戻されます。 • 「Value Changed」結果状況を戻す。	VALCHANGE
IgnoreMissingChildObject コネクター・プロパティーが True の場合、トップレベル・オブジェクトの取得に成功すれば、その子の取得に成功したかどうかにかかわらず、RetrieveByContent 操作はビジネス・オブジェクトについて「Value Changed」結果状況を戻します。	VALCHANGE
RetrieveByContent 操作で照会に一致する複数のエンティティーが見つかった場合は、次の処理が行われます。 • 一致の最初のオカレンスのみを取得する。このビジネス・オブジェクトが要求ビジネス・オブジェクト・パラメーターを介してコネクター・フレームワークに戻されます。 • 戻り状況記述子に取得に関する情報を充てんする。 • 「Multiple Hits」の状況を戻して、指定に一致するレコードが他にも存在することをコネクター・フレームワークに通知する。	MULTIPLE_HITS
RetrieveByContent 操作が非キー値による取得で一致を見つけなかった 場合は、次の処理が行われます。 • RetrieveByContent エラーの原因に関する追加情報の入った戻り状況記述子を充てんする。 • 「RetrieveByContent Failed」結果状況を戻す。	RETRIEVEBYCONTENT_FAILED

注: コネクター・フレームワークは、VALCHANGE 結果状況を受け取ると、InterChange Server への応答にビジネス・オブジェクトを含めます。詳細については、 194 ページの『動詞処理応答の送信』を参照してください。

Update 動詞の処理

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから Update 動詞を取得する場合、次のように、ビジネス・オブジェクト定義で型を指定された既存のアプリケーション・エンティティーが取得されるようにする必要があります。

- フラット・ビジネス・オブジェクトの場合、Update 動詞は、アプリケーション・エンティティーが要求ビジネス・オブジェクトに正確に一致するまで、指定されたエンティティー内のデータを変更する必要があることを示します。
- 階層型ビジネス・オブジェクトの場合、Update 動詞は、ビジネス・オブジェクト階層全体に一致するようにアプリケーション・エンティティーを更新する必要があることを示します。そのためには、コネクターがアプリケーション・エンティティーの作成、更新、および削除を行う必要があります。

– アプリケーションに子エンティティーが存在する場合は、必要に応じて変更されます。
– アプリケーション内に対応するエンティティーがない 階層型ビジネス・オブジェクト内の子オブジェクトは、アプリケーションに追加されます。
– アプリケーション内に存在するがビジネス・オブジェクト・ハンドラーには含まれていない 子エンティティーは、アプリケーションから削除されます。

注: 表ベースのアプリケーションの場合は、アプリケーション・データベース内でアプリケーション・エンティティー全体を更新する必要があります。通常、アプリケーション・エンティティーは、要求ビジネス・オブジェクトのビジネス・オブジェクト定義に関連したデータベース表内の新しい行として更新します。

このセクションには、Update 動詞の処理に役立つ以下の情報が記載されています。

- 『Update 動詞の標準処理』
- 116 ページの『論理削除イベントを表すビジネス・オブジェクトが持つ意味』
- 118 ページの『Update 動詞処理の結果状況』

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構成を採用すると、Update メソッドが Update 動詞の処理を扱うことになります。

Update 動詞の標準処理

以下のステップは、Update 動詞の標準処理の概略です。

1. 要求ビジネス・オブジェクトと同じ型の新しいビジネス・オブジェクトを作成します。この新しいビジネス・オブジェクトは、要求ビジネス・オブジェクトの取得されたコピーを保持する応答ビジネス・オブジェクト です。
2. アプリケーションから要求ビジネス・オブジェクトのコピーを取得します。

   要求ビジネス・オブジェクトの基本キーを使用して、アプリケーションからエンティティー全体に関するデータを再帰的に取得します。
   - フラット・ビジネス・オブジェクトの場合は、単一のアプリケーション・エンティティーを取得します。
   - 階層型ビジネス・オブジェクトの場合は、ビジネス・オブジェクト階層内のすべてのパスを展開し、Retrieve 操作を使用してアプリケーション・ビジネス・オブジェクトの中へ下降します。
3. 取得したデータを応答ビジネス・オブジェクトに入れます。これで、この応答ビジネス・オブジェクトがアプリケーション内のエンティティーの現在の状態の表現となります。

   これで、Update 操作は 2 つの階層型ビジネス・オブジェクトを比較し、アプリケーション・エンティティーを適切に更新することができます。
4. アプリケーション・エンティティー内の単純属性を更新して、トップレベル・ソース・ビジネス・オブジェクトに対応させます。

5. 応答ビジネス・オブジェクト (ステップ 2 で作成したもの) を要求ビジネス・オブジェクトと比較します。ビジネス・オブジェクト階層の最下位のレベルまで、この比較を続けます。

   次の規則に従って、トップレベル・ビジネス・オブジェクトの子を再帰的に更新します。

   - 子ビジネス・オブジェクトが応答ビジネス・オブジェクトと要求ビジネス・オブジェクトの*両方* に存在する場合は、Update 操作を実行して子を再帰的に更新します。
   - 子ビジネス・オブジェクトが要求ビジネス・オブジェクトに存在し、応答ビジネス・オブジェクトには*存在しない* 場合は、Create 操作を実行して子を再帰的に作成します。
   - 子ビジネス・オブジェクトが要求ビジネス・オブジェクトに存在せず、応答ビジネス・オブジェクトには存在する場合は、コネクターとアプリケーションの機能に応じて、Delete 操作 (物理的削除) または論理削除を使用して子を再帰的に削除します。論理削除の詳細については、116 ページの『論理削除イベントを表すビジネス・オブジェクトが持つ意味』を参照してください。

   注: 子ビジネス・オブジェクトの属性ではなく、子オブジェクトの存在または非存在のみを比較します。

コネクターのアプリケーションが論理削除をサポートしている場合は、コネクターは完全なビジネス・オブジェクト階層を再帰的に取得します。次に、Update 操作が状況属性を設定し、子の状況を再帰的に更新します。

注: 要求ビジネス・オブジェクトで参照されている任意の外部キー (Foreign Key が True に設定されている) についてアプリケーション・エンティティーが*存在しない* 場合は、Update 操作は失敗します。コネクターは外部キーが有効なキーである (存在するアプリケーション・エンティティーを参照している) ことを確認する必要があります。外部キーが無効な場合、Update 操作は FAIL を戻す必要があります。アプリケーション内に外部キーが存在すると見なされ、コネクターは外部キーとマークされたアプリケーション・オブジェクトの作成を試みることはありません。

図 34 は、アプリケーション・データベース内で顧客を表す関連アプリケーション・エンティティーのセットを示します。これらのエンティティーには、顧客、住所、電話番号、および顧客プロファイルが含まれます。サンプルの顧客 Acme Construction のデータベースには電話番号がないことに注目してください。

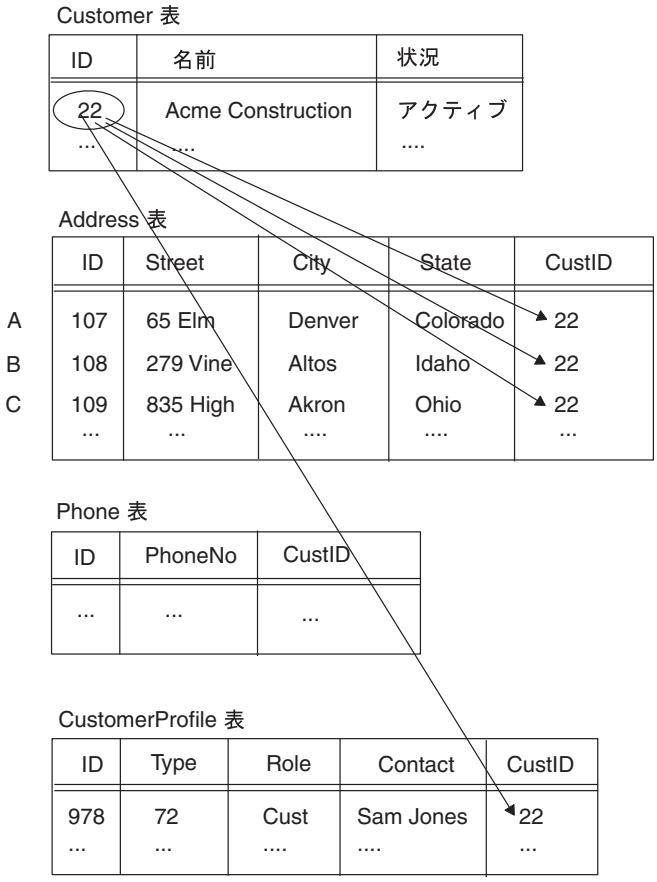

図 34. Update 要求前の Customer エンティティー

統合ブローカーが 図 35 に示す要求ビジネス・オブジェクトからなる更新要求を送信したとします。

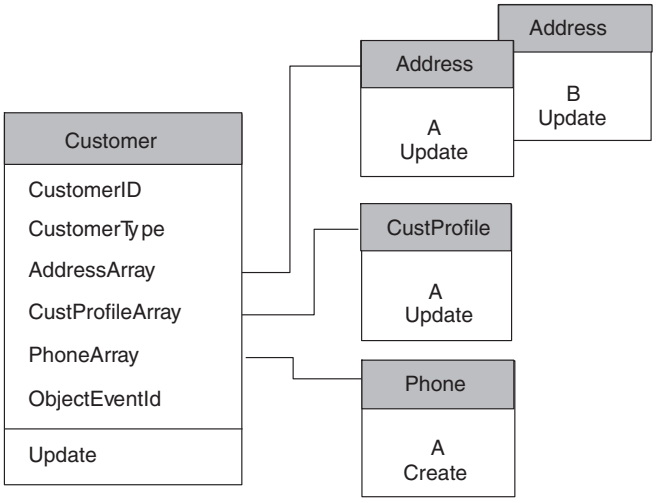

図 35. Update のための Customer 要求ビジネス・オブジェクト

この要求ビジネス・オブジェクトは、表 38 にリストされた変更が顧客 Acme Construction にあったことを示しています。

表 38. 要求ビジネス・オブジェクトの Acme Construction に対する更新

Acme Construction に対する更新	要求ビジネス・オブジェクトの表現
新しい電話番号を獲得	PhoneArray 属性の子ビジネス・オブジェクト (Phone オブジェクト A) に Create 動詞がある。
Denver および Altos の新しいオフィスに移転	AddressArray 属性に 2 つの子ビジネス・オブジェクト (Address オブジェクト A および B) が存在し、それぞれに Update 動詞がある。
Akron のオフィスを閉鎖	Akron の住所の AddressArray 属性には子ビジネス・オブジェクトが存在しない。
担当者の名前を変更	CustProfileArray 属性の子ビジネス・オブジェクト (CustProfile オブジェクト A) に Update 動詞がある。

コネクターのタスクは、この宛先アプリケーション用のアプリケーション・データベースをソース・アプリケーションと常に同期させることです。したがって、この要求に応答するためには、コネクターは以下のタスクを Update 操作の一部として実行する必要があります。

- 対応する Customer ビジネス・オブジェクトの単純属性で値が更新されている `Customer` 表の列をすべて更新する。
- Address オブジェクト A および B に対応する `Address` 表の行を更新する。適切な Address オブジェクト内の対応する単純属性に新しい値があれば、その値を使用してこれらの各行の列を更新します。この場合、`Street` 列が Denver および Altos のオフィスに対応して更新されます。
- Akron の住所に対応する `Address` 表の行を削除する。
- `CustomerProfile` 表の `Contact` 列を、CustProfile オブジェクト A ビジネス・オブジェクトの対応する単純属性の値に更新する。
- Phone オブジェクト A ビジネス・オブジェクトの単純属性の列値で `Phone` 表に行を作成する。この新しい行の `CustID` 列は、該当する `Customer` 列を識別する外部キー (22) で作成する必要があります。

図 36 は、Update 操作の完了後に顧客を表す関連アプリケーション・エンティティーのセットを示します。

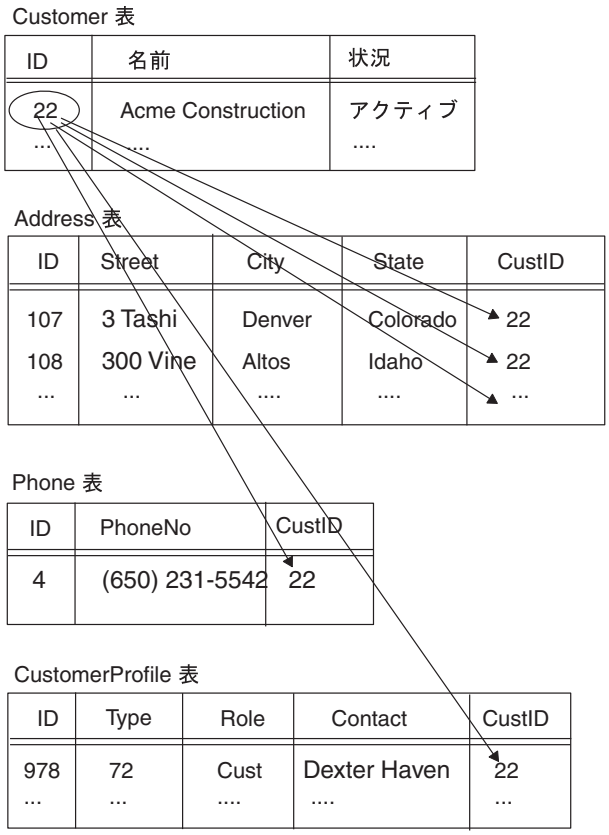

図 36. Update 要求後の Customer エンティティー

論理削除イベントを表すビジネス・オブジェクトが持つ意味

アプリケーションが物理削除をサポートしていても、統合ブローカーが論理削除のみをサポートするソース・アプリケーションから要求を送信してくる場合は、論理削除要求を表すビジネス・オブジェクトの処理が必要になります。論理削除操作では状況値を更新してエンティティーに削除済みのマークが付けられるので、論理削除を実行するアプリケーション用コネクターは論理削除を Update メソッドで処理する必要があります。この状況のシステム視点は次のとおりです。

- ソース・アプリケーションでのデータの削除を表すイベントは、Delete 動詞のあるアプリケーション固有のビジネス・オブジェクトとして送信する必要があります。同様に、ソース・アプリケーション・サイドのマップは汎用ビジネス・オブジェクトの動詞を Delete に設定する必要があります。
- 宛先サイドでは、論理削除アプリケーションをサポートするコネクター用のマップは、汎用ビジネス・オブジェクト内の Delete 動詞をアプリケーション固有ビジネス・オブジェクト内の Update 動詞に変換することができます。エンティティー状況値を表すビジネス・オブジェクト属性は非アクティブ状況に設定することができます。

このようにして、論理削除アプリケーションを表すコネクターは、Update 動詞があり状況値が適切にマーク付けされたアプリケーション固有のビジネス・オブジェクトを受け取ります。

例えば、図 37 のビジネス・オブジェクト表現になるように、ソース・アプリケーション・エンティティーが更新されたとします。ソース・アプリケーション・エンティティー内のコンポーネントは、更新され、作成され、削除されています。

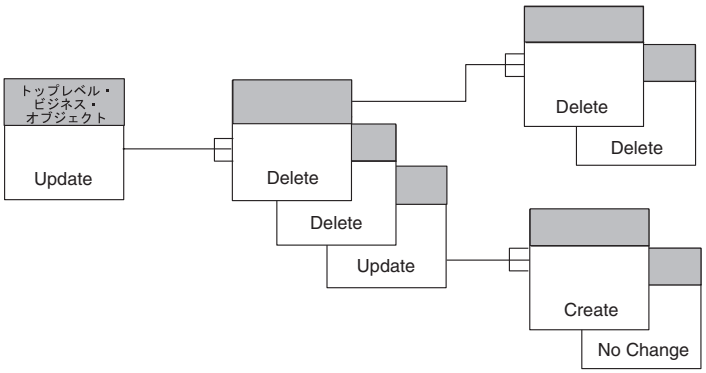

図 37. ソース・アプリケーション内の更新済みエンティティー

ソース・アプリケーション・コネクターで、 131 ページの『第 5 章 イベント通知』で推奨するようにイベント通知がインプリメント済みである場合、削除された子ビジネス・オブジェクトはビジネス・オブジェクト階層内に存在せず、ビジネス・オブジェクトには単に更新された子ビジネス・オブジェクトと新規ビジネス・オブジェクトが含まれます。

Update 要求を表すビジネス・オブジェクトの一例は 図 38 のようになります。この図では、親オブジェクトが更新に設定され、削除されたすべてのエンティティーはビジネス・オブジェクト階層内に存在しません。

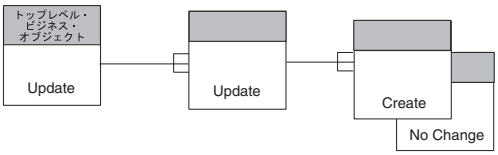

図 38. 物理削除コネクターの要求ビジネス・オブジェクトの更新

この場合、コネクターはソース・ビジネス・オブジェクトと宛先ビジネス・オブジェクトを比較し、ソース・ビジネス・オブジェクトに存在しないエンティティーを削除します。

しかし、ソース・アプリケーションが論理削除をサポートする場合、ソース・コネクターは、削除に更新のタグを付け状況属性を非アクティブ値に設定したビジネス・オブジェクトを送信することがあります。このビジネス・オブジェクトは 図 39 のようになります。ここで、削除操作である更新は「[D]」と示されています。

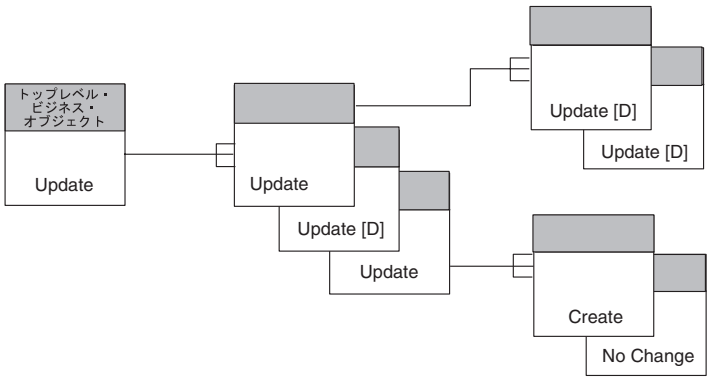

図 39. 論理削除コネクターの要求ビジネス・オブジェクトの更新

論理削除要求を表すソース・ビジネス・オブジェクトを処理するには、次のようにいくつかの方法があります。

- 子ビジネス・オブジェクトの状況を検査するようにマッピングを実装する。特定の子ビジネス・オブジェクトの状況が非アクティブの場合、ビジネス・オブジェクトをマッピングで除去することができます。
- 更新操作が実際に削除操作であるかどうかを判断するように、Update 操作を実装する。論理削除ソース・アプリケーションでは、状況値によりエンティティーがアクティブであるか非アクティブであるかのマークが付けられる場合があります。ソースのアプリケーション固有ビジネス・オブジェクトでは、通常、状況値は属性です。物理削除をサポートするアプリケーションのエンティティーは状況情報を含まない場合がありますが、アプリケーション固有ビジネス・オブジェクトを拡張して状況情報を含めるようにすることができます。
- 追加の状況属性を追加するか、既存の属性に状況値を多重定義して、ビジネス・オブジェクトを拡張する。Update 操作は、要求を受け取ると、状況属性を検査することができます。状況属性が非アクティブ値に設定されている場合は、操作は実際に削除です。次に、Update 操作は、ビジネス・オブジェクト動詞を Delete に設定し、Delete 操作を呼び出して、削除済み子ビジネス・オブジェクトを処理することができます。

Update 動詞処理の結果状況

Update 操作は、表 39 に示す結果状況の 1 つを戻さなければなりません。

表 39. Java Update 動詞処理で可能な結果状況

Update 条件	Java 結果状況
アプリケーション・エンティティーが存在する場合、Update 操作は次の処理を行います。 • アプリケーション・エンティティー内のデータを変更する。 • 「Success」結果状況を戻す。	SUCCEED
行またはエンティティーが存在しない場合は、Update 操作は次の処理を行います。 • アプリケーション・エンティティーを作成する。 • 「Value Changed」結果状況を戻して、コネクターがビジネス・オブジェクトを変更したことを示す。	VALCHANGE

表 39. Java Update 動詞処理で可能な結果状況 (続き)

Update 条件	Java 結果状況
アプリケーション・エンティティーを作成できない場合、Update 操作は次の処理を行います。 • 更新エラーの原因に関する情報で戻り状況記述子を充てんする。 • 「Fail」結果状況を戻す。	FAIL
外部キーとして識別されたいずれかのオブジェクトがアプリケーションから欠落している場合は、Update 操作は次の処理を行います。 • 更新エラーの原因に関する情報で戻り状況記述子を充てんする。 • 「Fail」結果状況を戻す。	FAIL

注: コネクター・フレームワークは、VALCHANGE 結果状況を受け取ると、InterChange Server への応答にビジネス・オブジェクトを含めます。詳細については、194 ページの『動詞処理応答の送信』を参照してください。

Delete 動詞の処理

削除については、アプリケーションは 表 40 に示すインプリメントのいずれかをサポートします。

表 40. Delete の実装

Delete の実装	説明	動詞処理サポート
物理削除	指定されたアプリケーション・エンティティーを物理的に除去します。	Delete 操作
論理削除	エンティティーを実際には除去せず、特殊な「deleted」状況のマークを付けます。	Update 操作

注: アプリケーションがすべての 型の削除操作を許可しない場合、コネクターは「Fail」結果状況を戻すことがあります。

このセクションで説明する Delete 操作は、アプリケーション内のデータの実際の物理的削除を実行します。論理削除を実行するアプリケーション用コネクターは論理削除を Update 操作で処理する必要があります。詳細については、116 ページの『論理削除イベントを表すビジネス・オブジェクトが持つ意味』を参照してください。

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから Delete 動詞を取得する場合、物理削除が実行されるようにする必要があります。つまり、次のように、ビジネス・オブジェクト定義で型を指定されたアプリケーション・エンティティーが削除されるようにする必要があります。

- フラット・ビジネス・オブジェクトの場合は、Delete 動詞は指定されたエンティティーの削除が必要であることを示します。

- 階層型ビジネス・オブジェクトの場合、Delete 動詞は、トップレベル・ビジネス・オブジェクトを削除する必要があることを示します。アプリケーション・ポリシーに応じて、子ビジネス・オブジェクトを表す関連エンティティーを削除する場合もあります。

注: 表ベースのアプリケーションの場合、通常は 1 つまたは複数のデータベース表の行を削除して、アプリケーション・エンティティー全体をアプリケーション・データベースから削除する必要があります。

このセクションには、Delete 動詞の処理に役立つ以下の情報が記載されています。

- 『Delete 動詞の標準処理』
- 『Delete 動詞処理の結果状況』

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構成を採用すると、Delete メソッドが Delete 動詞の処理を扱うことになります。

Delete 動詞の標準処理

以下のステップは、Delete 動詞の標準処理の概略です。

1. 要求ビジネス・オブジェクトに再帰的取得を実行して、トップレベル・ビジネス・オブジェクトに関連したアプリケーション内のすべてのデータを取得します。
2. 最下位のエンティティーから最上位のエンティティーへ上昇して、要求ビジネス・オブジェクトが表すエンティティーに再帰的削除を実行します。

注: Delete 操作はアプリケーションの機能により制限される場合があります。例えば、カスケード削除は必ずしも望ましい操作であるとは限りません。アプリケーションの API を使用する場合は、削除操作が適切に自動的に完了する場合があります。アプリケーションの API を使用しない 場合は、コネクターでアプリケーション内の子エンティティーを削除すべきかどうか判断が必要な場合があります。子エンティティーが他のエンティティーにより参照される場合は、削除するのは適切ではありません。

Delete 動詞処理の結果状況

削除操作は、表 41 に示す結果状況の 1 つを戻さなければなりません。

表 41. Java Delete 動詞処理で可能な結果状況

Delete 条件	Java 結果状況
InterChange Server のみ: 多くの場合、コネクターは「Value Changed」結果状況を戻して、削除操作後にシステムが関係表をクリーンアップできるようにします。	VALCHANGE
すべての統合ブローカー: Delete 操作が正常に実行されると、次の処理が行われます。 • 削除エラーの原因に関する追加情報で戻り状況記述子を充てんする。 • 「Fail」結果状況を戻す。	FAIL

注: コネクター・フレームワークは、VALCHANGE 結果状況を受け取ると、InterChange Server への応答にビジネス・オブジェクトを含めます。詳細については、 194 ページの『動詞処理応答の送信』を参照してください。

Exists 動詞の処理

ビジネス・オブジェクト・ハンドラーが要求ビジネス・オブジェクトから Exists 動詞を取得する場合、ビジネス・オブジェクト定義で型を指定されたアプリケーション・エンティティーが存在するかどうか判断する必要があります。この操作により、統合ブローカーは、エンティティーに対する操作を実行する前にエンティティーの存在を検査することができます。例えば、顧客サイトで、ソース・アプリケーションと宛先アプリケーションの Order、Customer、および Item エンティティーの同期化を望んでいるとします。注文を同期する前に、ユーザーは、Order ビジネス・オブジェクトが参照する Customer エンティティーが宛先アプリケーション・データベースに存在していることを確認する必要があります。さらに、ユーザーは、OrderLineItem 子ビジネス・オブジェクトが参照する、各 Item エンティティーも宛先アプリケーションに存在していることを確認する必要があります。

注: 表ベースのアプリケーションの場合、Exists メソッドは、通常はデータベース表内の行を検査して、アプリケーション・データベースにエンティティーが存在しているかどうか検査します。

ユーザーは統合ブローカーを構成して、Exists 動詞と基本キーが設定されている Customer ビジネス・オブジェクトでコネクターを呼び出すようにすることができます。このようにして、統合ブローカーは、アプリケーション内に顧客がすでに存在しているかどうかを検査することができます。同様に、ユーザーは統合ブローカーを構成して、Exists 動詞と基本キーが設定されている Item ビジネス・オブジェクトを参照してコネクターを呼び出すようにすることができます。ユーザーは、アプリケーション・エンティティーの存在の確認が失敗した場合に Order の同期化を停止することを決定することができます。

このセクションには、Exists 動詞のインプリメントに役立つ以下の情報が記載されています。

- 『Exists 動詞の標準処理』
- 122 ページの『Exists 動詞処理の結果状況』

注: サポートされる動詞がそれぞれ個別の Java メソッドで処理されるように、ビジネス・オブジェクト・ハンドラーをモジュール化することができます。この構造を採用すると、Exists メソッドが Exists 動詞の処理を扱うことになります。

Exists 動詞の標準処理

Exists メソッドの標準的な振る舞いは、アプリケーション・データベースにトップレベル・ビジネス・オブジェクトの存在を照会することです。

Exists 動詞処理の結果状況

Exists 操作は、表 42に示す結果状況値の 1 つを戻さなければなりません。

表 42. Java Exists 動詞処理で可能な結果状況

Exists 条件	Java 結果状況
アプリケーション・エンティティーが存在すると、Exists 操作は「Success」を戻します。	SUCCEED
Exists 操作がトップレベル・オブジェクトの取得に失敗すると、次の処理が行われます。 • 戻り状況記述子を充てんする。 • 「Fail」結果状況を戻す。	FAIL

ビジネス・オブジェクトの処理

ビジネス・オブジェクト・ハンドラーの役割は、要求ビジネス・オブジェクトのデコンストラクションを行い、要求を処理し、アプリケーション内で要求された操作を実行することです。そのために、ビジネス・オブジェクト・ハンドラーは要求ビジネス・オブジェクトから動詞および属性情報を抽出し、API 呼び出し、SQL ステートメント、またはその他のタイプのアプリケーション相互作用を生成して操作を実行します。

基本ビジネス・オブジェクト処理には、ビジネス・オブジェクトのアプリケーション固有の情報 (存在する場合) からのメタデータの抽出と属性値へのアクセスが含まれます。属性値に対するアクションは、ビジネス・オブジェクトがフラットか階層かどうかに依存します。このセクションでは、ビジネス・オブジェクト・ハンドラーが以下の種類のビジネス・オブジェクトをどのように処理できるかの概要を説明します。

- 『フラット・ビジネス・オブジェクトの処理』
- 125 ページの『階層型ビジネス・オブジェクトの処理』

フラット・ビジネス・オブジェクトの処理

このセクションには、フラット・ビジネス・オブジェクトを処理する方法について以下の情報が記載されています。

- 『フラット・ビジネス・オブジェクトの表現』
- 124 ページの『単純属性へのアクセス』

フラット・ビジネス・オブジェクトの表現

他のビジネス・オブジェクト (子ビジネス・オブジェクトと呼ぶ) が含まれていないビジネス・オブジェクトをフラット・ビジネス・オブジェクト と呼びます。フラット・ビジネス・オブジェクトの属性はすべて*単純属性* です。各属性には他のビジネス・オブジェクトへの参照ではなく、実際の値が入っています。

Customer という名前のサンプル・ビジネス・オブジェクトに動詞処理を実行する必要があるとします。このビジネス・オブジェクトは、サンプルの表ベースのアプリケーション内の単一のデータベース表を表します。データベース表は customer と

いう名前で、顧客データが入っています。図 40 は、Customer ビジネス・オブジェクト定義と、アプリケーション内の対応する customer 表を示しています。

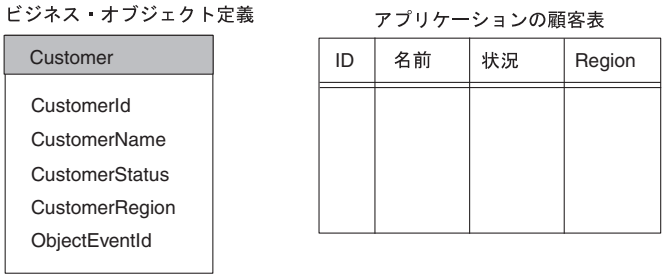

図 40. フラット・ビジネス・オブジェクトおよび対応するアプリケーション表

図 40 に示すように、Customer ビジネス・オブジェクトには、CustomerId、CustomerName、CustomerStatus、CustomerRegion の 4 つの単純属性があります。これらの属性は、customer 表の列に対応します。このビジネス・オブジェクトには、必須の ObjectEventId 属性も含まれています。

注: ObjectEventId 属性は IBM WebSphere Business Integration システムによって使用され、アプリケーション表の列には対応しません。この属性は、Business Object Designer によってビジネス・オブジェクトに自動的に追加されます。

図 41 は、拡張されたビジネス・オブジェクト定義とビジネス・オブジェクトのインスタンスを示します。ビジネス・オブジェクト定義には、ビジネス・オブジェクト名、属性名、プロパティー、およびアプリケーション固有の情報が入っています。ビジネス・オブジェクト・インスタンスには、ビジネス・オブジェクト名、アクティブな動詞、属性名、および値のみが入っています。

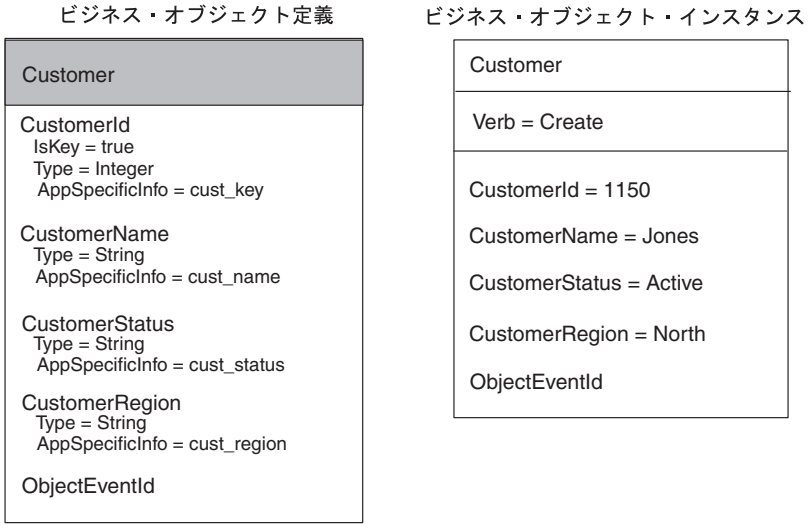

図 41. アプリケーション固有の情報を持つフラット・ビジネス・オブジェクト

単純属性へのアクセス

動詞操作がビジネス・オブジェクト定義内の必要な情報にアクセスした後、属性に関する情報へのアクセスがしばしば必要になります。属性プロパティーには、カーディナリティー、キーまたは外部キー指定、および最大長が含まれます。例えば、Create メソッドでは、属性のアプリケーション固有の情報の取得が必要です。コネクター・ビジネス・オブジェクト・ハンドラーは、通常、属性プロパティーを使用して属性値の処理方法を決定します。

図 42 は、図 41 のビジネス・オブジェクトからの CustomerId 属性のビジネス・オブジェクト属性プロパティーの図解です。

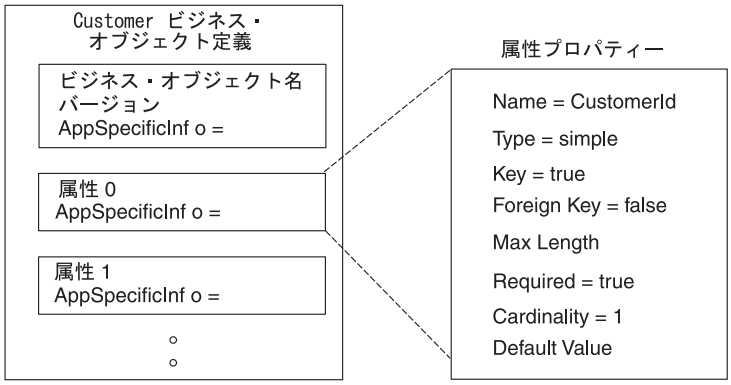

図 42. ビジネス・オブジェクト属性プロパティー

各属性には、ビジネス・オブジェクト定義内にゼロをカーディナリティーとする整数指標 (序数位置) があります。例えば、図 42 に示すように、CustomerId 属性は序数位置 0 でアクセスされ、CustomerName 属性は序数位置 1 でアクセスされ、以下同様です。Java コネクター・ライブラリーは、名前と序数位置による属性へのアクセスを提供します。

フラット Customer ビジネス・オブジェクトを処理するビジネス・オブジェクト・ハンドラーの場合、ビジネス・オブジェクトのデコンストラクションには以下のステップがあります。

1. ビジネス・オブジェクト定義内のアプリケーション固有の情報から表名と列名を抽出する。
2. ビジネス・オブジェクト・インスタンスから属性の値を抽出する。

図 41 に示すように、Customer ビジネス・オブジェクト定義はメタデータ主導型のコネクター用に設計されています。そのビジネス・オブジェクト定義には、動詞操作が操作対象のアプリケーション・エンティティーを見つけるために使用するアプリケーション固有の情報が含まれています。アプリケーション固有の情報は 表 43 に示すように設計されています。

表 43. 表ベースのアプリケーションのアプリケーション固有の情報

アプリケーション固有の情報	目的
ビジネス・オブジェクト定義	このビジネス・オブジェクトに関連したアプリケーション・データベース表の名前

表 43. 表ベースのアプリケーションのアプリケーション固有の情報 (続き)

アプリケーション固有の情報	目的
属性	この属性に関連したアプリケーション表の列の名前

注: アプリケーション固有の情報は、外部キー、およびアプリケーション・データベース内のエンティティー間のその他の種類の関係に関する情報の保管にも使用されます。メタデータ主導型のコネクターは、この情報を使用して、SQL ステートメントまたはアプリケーション API 呼び出しを作成します。

階層型ビジネス・オブジェクトの処理

ビジネス・オブジェクトが階層の場合、親ビジネス・オブジェクトに子ビジネス・オブジェクトを含めることができます。子ビジネス・オブジェクトにもさらに子ビジネス・オブジェクトが含まれることがあり、以下同様となります。階層型ビジネス・オブジェクトは、*トップレベル・ビジネス・オブジェクト* (階層の最上位にあるビジネス・オブジェクト) と *子ビジネス・オブジェクト* (トップレベル・ビジネス・オブジェクトより下のすべてのビジネス・オブジェクト) で構成されます。子ビジネス・オブジェクトは親オブジェクト内に属性として含まれます。

このセクションには、階層型ビジネス・オブジェクトを処理する方法について以下の情報が記載されています。

- 『トップレベル・ビジネス・オブジェクトおよび子ビジネス・オブジェクトの表現』
- 127 ページの『子ビジネス・オブジェクトへのアクセス』

トップレベル・ビジネス・オブジェクトおよび子ビジネス・オブジェクトの表現

トップレベル・ビジネス・オブジェクトに子ビジネス・オブジェクトがある場合、トップレベル・オブジェクトは子の親になります。同様に、子ビジネス・オブジェクトに子がある場合は、子オブジェクトはまたこの親になります。親/子という用語は、ビジネス・オブジェクト間の関係の記述に使用されるほか、アプリケーション・エンティティー間の関係の記述にも使用されます。

親ビジネス・オブジェクトと子ビジネス・オブジェクトの間には 2 種類の包含関係があります。

- カーディナリティー 1 の包含 — 属性には単一の子ビジネス・オブジェクトが含まれます。
- カーディナリティー 2 の包含 — 属性には*ビジネス・オブジェクト配列* と呼ばれる構造で複数の子ビジネス・オブジェクトが含まれます。

図 43 は、典型的な階層型ビジネス・オブジェクトを示します。このトップレベル・ビジネス・オブジェクトには、子ビジネス・オブジェクトとの間にカーディナリティー 1 およびカーディナリティー n の両方の包含関係があります。

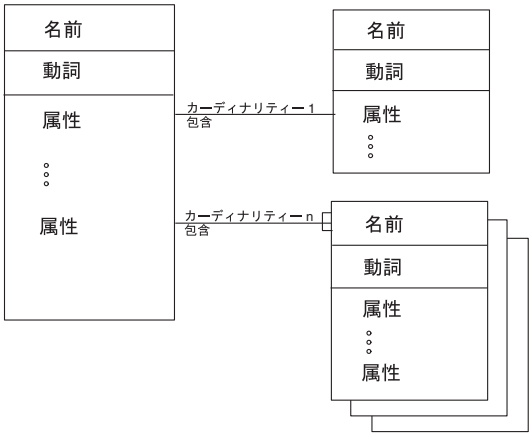

図 43. 階層型ビジネス・オブジェクト

典型的な表ベースのアプリケーションでは、エンティティー間の関係はデータベース内の基本キーおよび外部キーによって表されます。親エンティティーに基本キーが含まれ、子エンティティーに外部キーが含まれます。階層型ビジネス・オブジェクトは同じような方法で編成できます。

- カーディナリティー 1 タイプ (単一カーディナリティー) の関係においては、親ビジネス・オブジェクトはそれぞれ単一の子ビジネス・オブジェクトに関係します。

  子ビジネス・オブジェクトは、通常、1 つまたは複数の外部キーを持ち、その値は親ビジネス・オブジェクト内の対応する基本キーと同じです。エンティティー間の関係はアプリケーションによりさまざまに構造化できますが、外部キーを使用するアプリケーションの単一カーディナリティー関係は 図 44 のようになります。
- カーディナリティー n タイプ (複数カーディナリティー) の関係では、親ビジネス・オブジェクトは子ビジネス・オブジェクトの配列においてそれぞれ 0 個以上の子ビジネス・オブジェクトに関係することができます。

  配列内のそれぞれの子ビジネス・オブジェクトには、外部キー属性があり、その値は親ビジネス・オブジェクトの基本キー属性の対応する値と同じです。複数カーディナリティー関係は 図 45 のように表現できます。

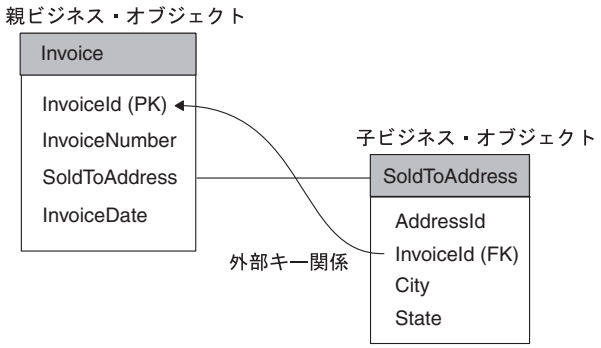

図 44. 単一カーディナリティーを持つビジネス・オブジェクト

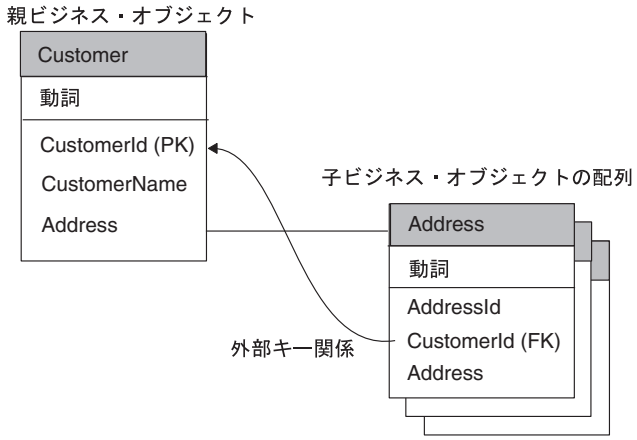

図 45. 複数カーディナリティーを持つビジネス・オブジェクト

注: 図 44 および 図 45 では、属性の次に「PK」というストリングがあります。これがビジネス・オブジェクト内の基本キーです。属性の次の「FK」というストリングは外部キーです。

子ビジネス・オブジェクトへのアクセス

`doVerbFor()` メソッドは、動詞処理の一部として、階層型ビジネス・オブジェクトを処理する必要があります。`doVerbFor()` メソッドは、フラット・ビジネス・オブジェクトを処理する場合と同じ基本ステップを実行して階層型ビジネス・オブジェクトを処理します。つまり、まずアプリケーション固有の情報を取得し、次に属性にアクセスします。ただし、属性に子ビジネス・オブジェクトが含まれている場合は、`doVerbFor()` は以下のステップで子ビジネス・オブジェクトにアクセスする必要があります。

1. `isObjectType()` メソッドを呼び出して、属性型が型 `OBJECT` であるかどうかを判別します。

   `OBJECT` 型は、属性が複合属性であること、つまり属性の内容が単純な値ではなくビジネス・オブジェクトであることを示します。`OBJECT` 属性型定数は CWConnectorAttrType クラスで定義されます。属性が複合属性の場合、つまりビジネス・オブジェクトが入っている場合は、`isObjectType()` メソッドは True を戻します。

2. `doVerbFor()` メソッドは、属性にビジネス・オブジェクトが入っていることを見つけると、`isMultipleCard()` を使用して属性のカーディナリティーを検査します。

   属性が単一カーディナリティー (カーディナリティー 1) の場合は、メソッドは子に対して要求された操作を実行することができます。子ビジネス・オブジェクトに対する操作を行う方法の 1 つは、子オブジェクトで `doVerbFor()` または動詞メソッドを再帰的に呼び出すことです。ただし、このような再帰呼び出しでは、子ビジネス・オブジェクトが次のように設定されていることが想定されています。

   - 子ビジネス・オブジェクト上の動詞が設定されている場合は、メソッドは指定された操作を実行する必要があります。

- 子ビジネス・オブジェクト上の動詞が設定されていない 場合は、動詞メソッドは子オブジェクトで別のメソッドを呼び出す前に、子ビジネス・オブジェクトの動詞をトップレベル・ビジネス・オブジェクトの動詞に設定する必要があります。

複数カーディナリティー (カーディナリティー n) の属性の場合は、属性には子ビジネス・オブジェクトの配列が入っています。この場合、コネクターは、個別の子ビジネス・オブジェクトをプロセスするためにはまず配列の内容にアクセスする必要があります。`doVerbFor()` メソッドは配列から個別のビジネス・オブジェクトにアクセスできます。

- 個別のビジネス・オブジェクトにアクセスするために、メソッドは `getObjectCount()` メソッドにより配列内の子ビジネス・オブジェクトの数を取得してから、オブジェクトへのアクセスを繰り返すことができます。
- 個々の子ビジネス・オブジェクトにアクセスするために、メソッドは配列の 1 つのエレメントでビジネス・オブジェクトを取得できます。

`doVerbFor()` メソッドは、子ビジネス・オブジェクトにアクセスすると、必要に応じてその子を再帰的に処理できます。

注: コネクターで子ビジネス・オブジェクトの配列を作成しないでください。カーディナリティーが n の場合、配列は常にビジネス・オブジェクト定義に関連付けられます。

コネクターが要求ビジネス・オブジェクトを処理する場合、配列の一部または全部が空であっても、そのビジネス・オブジェクトにはすべての配列が含まれます。子ビジネス・オブジェクトが 1 つも含まれていない配列はサイズ 0 の配列です。

メインの動詞メソッドがサブメソッドを呼び出して子オブジェクトを処理できるように、動詞操作をモジュール化する必要があります。図 46 に示したようなビジネス・オブジェクトの場合、Create メソッドはまず親 Customer ビジネス・オブジェクトのアプリケーション・エンティティーを作成してから、サブメソッドを呼び出し、親ビジネス・オブジェクトの全探索を行って、含まれているビジネス・オブジェクトを参照する属性を見つけます。

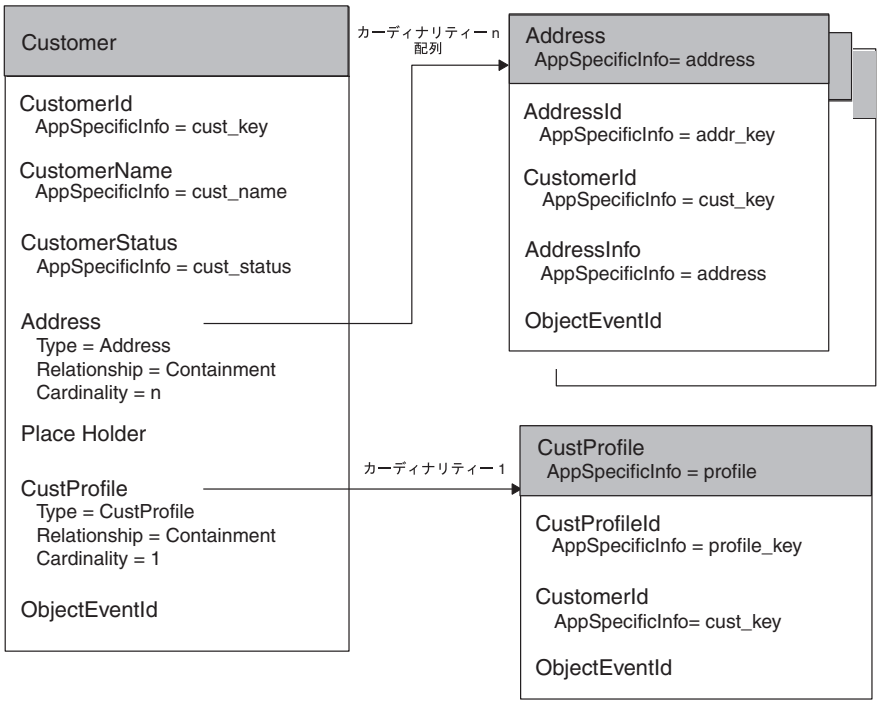

図 46. 階層型ビジネス・オブジェクト定義の例

サブメソッドは、OBJECT 型の属性を見つけると、必要に応じてその属性を処理することができます。例えば、サブメソッドは、Address 配列の中で子ビジネス・オブジェクトをそれぞれ取得し、再帰的に doCreate() を呼び出して、Address 属性を処理します。メインのメソッドは、配列内のすべての Address の子が処理されるまで、データベースに address エンティティーを 1 つずつ作成します。最後に、サブメソッドは単一カーディナリティーの CustProfile ビジネス・オブジェクトを処理します。

子ビジネス・オブジェクトへのアクセス方法の詳細については、 199 ページの『子ビジネス・オブジェクトへのアクセス』を参照してください。

コネクター応答の指示

doVerbFor() メソッドは、終了する前に、コネクター・フレームワークに送り返す応答を準備します。この応答は動詞処理の成功 (または不成功) を指示します。doVerbFor() を呼び出したコネクター・フレームワークは、この情報を使用して次のアクションを決定し、統合ブローカーに戻す応答を作成します。

doVerbFor() メソッドは表 44 に示す応答情報をコネクター・フレームワークに提供することができます。

表 44. doVerbFor() メソッドからの応答情報

応答情報	応答が戻される方法
結果状況	doVerbFor() の整数戻りコード。

表 44. doVerbFor() メソッドからの応答情報 (続き)

応答情報	応答が戻される方法
戻り状況記述子	引き数として渡された戻り状況記述子 — コネクター・フレームワークは空の戻り状況記述子を引き数として `doVerbFor()` に渡します。メソッドはメッセージおよび状況値を使用してこの記述子を更新し、通知状況、警告状況、またはエラー状況を準備することができます。
応答ビジネス・オブジェクト	引き数として渡された要求ビジネス・オブジェクト — コネクター・フレームワークは引き数として要求ビジネス・オブジェクトを `doVerbFor()` に渡します。メソッドは属性値を使用してこの要求ビジネス・オブジェクトを更新し、応答ビジネス・オブジェクトを準備することができます。

Java コネクター用のこの応答情報の送信方法については、194 ページの『動詞処理応答の送信』を参照してください。

アプリケーションとの接続が切断された場合の処理

コネクター・フレームワークがコネクター・アプリケーション固有のコンポーネントを呼び出すたびに、アプリケーション固有のコードはアプリケーションとの接続がまだオープンであるかどうかを検査します。ビジネス・オブジェクト・ハンドラーの場合、この検査は `doVerbFor()` メソッドか各動詞メソッドで行う必要があります。

接続が失われると、`doVerbFor()` メソッドは致命エラー・メッセージを記録する必要があります。この場合、`LogAtInterchangeEnd` コネクター構成プロパティーが True に設定されていれば、電子メールによる通知が起動されます。また、メソッドは、APPRESPONSETIMEOUT 結果状況を戻して、アプリケーションが応答しないことをコネクター・コントローラーに知らせる必要があります。この事態が起きると、コネクターが実行されているプロセスは停止されます。システム管理者はアプリケーションの問題を修正し、コネクターを再始動して、ビジネス・オブジェクト要求の処理を続ける必要があります。

詳細については、182 ページの『動詞処理の前の接続の検証』を参照してください。

第 5 章 イベント通知

この章では、コネクターでのイベント通知の方法について説明します。*イベント通知* は、アプリケーション・ビジネス・エンティティーに加えられた変更を検出するため、アプリケーションと対話をするために実装された機構です。この章では、イベント通知機構の実装方法について、以下の内容を説明します。

- 『イベント通知機構の概要』
- 132 ページの『アプリケーション用イベント・ストアの実装』
- 139 ページの『イベント検出の実装』
- 144 ページの『イベント取得の実装』
- 146 ページの『ポーリング・メソッドの実装』
- 150 ページの『イベント処理に関する特別な考慮事項』

注: イベント通知の概要については、24 ページの『イベント通知』を参照してください。

イベント通知機構の概要

イベント通知機構 により、コネクターはアプリケーション内のエンティティーに変更が発生した時期を判定することができます。イベント通知機構の実装は、表 45 に示すように、3 段階のプロセスです。

表 45. イベント通知機構の段階

イベント通知機構の段階	詳細情報
アプリケーション・ビジネス・エンティティーを変更したイベントの通知を保持するためにアプリケーションが使用するイベント・ストア を作成します。	132 ページの『アプリケーション用イベント・ストアの実装』
アプリケーションの内部に**イベント検出** 機構を実装します。イベント検出とは、アプリケーション・エンティティーの変更を検知し、その変更に関する情報を格納した**イベント・レコード**をアプリケーション内部のイベント・ストアに書き込む動作です。	139 ページの『イベント検出の実装』
イベント・ストアからイベントを取り込むため、コネクター内部に**イベント取得** 機構 (ポーリング機構など) を実装し、他のアプリケーションに通知するため適切なアクションを実行します。	132 ページの『アプリケーション用イベント・ストアの実装』

注: イベント通知機構の設計に関する考慮事項については、24 ページの『イベント通知』を参照してください。

多くの場合、コネクターがイベント通知機構を使用するためには、アプリケーションの構成または変更が必要です。通常このアプリケーション構成は、コネクターの

アプリケーション固有コンポーネントをインストールする手順の一部として実行されます。アプリケーションの変更としては、アプリケーション内部でのユーザー・アカウントの設定、アプリケーション・データベース内部でのイベント・ストアやイベント表の作成、データベースへのストアード・プロシージャーの挿入、受信箱のセットアップなどがあります。アプリケーションがイベント・レコードを生成する場合には、イベント・レコードのテキストを構成することが必要な場合もあります。

コネクターがイベント通知機構を使用できるように、コネクターを構成することが必要な場合もあります。例えば、コネクターに固有な構成プロパティーを、システム管理者がイベント・ストアおよびイベント表の名前に設定しなければならない場合もあります。

アプリケーション用イベント・ストアの実装

イベント・ストア は、アプリケーション内の永続的キャッシュで、ここにイベント・レコードが保管されているため、コネクターはそのイベント・レコードを処理することができます。イベント・ストアは、データベース表、アプリケーション・イベント・キュー、電子メール受信箱など、どのようなタイプの永続ストアでも差し支えありません。コネクターが運用可能でない場合、アプリケーションは永続的イベント・ストアを使用することにより、コネクターが運用可能になるまで、イベント・レコードを検出および保管することができます。

このセクションでは、イベント・ストアの次の点について説明します。

- 『イベント・レコードの標準的内容』
- 135 ページの『イベント・ストアの可能な実装』

イベント・レコードの標準的内容

イベント・レコードは、コネクターがイベントを処理する上で必要なすべてのものをカプセル化していることが必要です。各イベント・レコードは、コネクターのポーリング・メソッドがイベント・データを取得し、イベントを表すビジネス・オブジェクトを作成するために十分な情報を格納していることが必要です。

注: イベント取得機構にはいろいろな種類がありますが、このセクションでは、最も一般的な機構であるポーリングで扱われるイベント・レコードに注目します。

アプリケーションが、イベント・レコードをイベント・ストアに書き込むイベント検出機構を提供する場合、イベント・レコードはオブジェクトと動詞の個別の詳細情報を提供します。アプリケーションがこのレベルの詳細度を提供していない場合には、構成により必要な詳細度を実現できます。

表 46 に、イベント・レコードの標準的要素を示します。以降のセクションで、この表の一部の項目をさらに詳しく説明します。

表 46. イベント・レコードの標準的要素

要素	説明	詳細情報
イベント ID	イベントの固有 ID (UID) です。	134 ページの『イベント ID』

表 46. イベント・レコードの標準的要素 (続き)

要素	説明	詳細情報
ビジネス・オブジェクト名	リポジトリーに記載されたとおりの、ビジネス・オブジェクト定義の名前です。	133 ページの『ビジネス・オブジェクト名』
動詞	Create、Update、Delete など動詞の名前です。	133 ページの『イベントの動詞』
オブジェクト・キー	アプリケーション・エンティティー用の基本キーです。	133 ページの『オブジェクト・キー』
優先順位	イベントの優先順位です。範囲は 0 から n で、0 が最高です。	149 ページの『イベント優先順位によるイベントの処理』
タイム・スタンプ	アプリケーションがイベントを生成した時刻です。	なし。
状況	イベントの状況。これはアーカイブ・イベントに使用します。	134 ページの『イベント状況』
説明	イベントを記述するテキスト・ストリングです。	なし
コネクター ID	イベントを処理するコネクターの ID です。	150 ページの『イベント分配』

注: イベント・レコードに含まれる情報として、イベント ID、ビジネス・オブジェクト名、動詞、およびオブジェクト・キーが最小セットを構成します。また、イベント・ストアに多数のイベントが待機されている場合に、コネクターが優先順位に従ってイベントを選択できるように、イベントに優先順位を設定することが必要な場合もあります。

ビジネス・オブジェクト名

ビジネス・オブジェクト定義の名前から、イベント・サブスクリプションについて検査することができます。イベント・レコードでは、ビジネス・オブジェクト定義の名前を*正確に* 指定するようにします。例えば、`Customer` ではなく、`SAP_Customer` と指定します。

イベントの動詞

動詞は、Create、Update、Delete など、アプリケーションの中で発生するイベントの種類を表します。動詞からイベント・サブスクリプションについて検査することができます。

注: アプリケーション・データの削除を表すイベントは、Delete 動詞を持つイベント・レコードを生成します。これは、論理的な削除操作、すなわち状況値を非アクティブに更新する意味での削除の場合も成立します。詳細については、151 ページの『削除イベントの処理』を参照してください。

コネクターがビジネス・オブジェクトの中に設定する動詞は、イベント・レコードに指定された動詞と同じです。

オブジェクト・キー

オブジェクトにサブスクライブ・イベントがある場合、コネクターはエンティティーのオブジェクト・キーを使用することにより、エンティティー・データのセット全体を取り込むことができます。

注: アプリケーション・エンティティーから取り出すデータの中で、イベント・レコードに含まれるデータは、ビジネス・オブジェクト名、アクティブな動詞、およびオブジェクト・キーのみです。イベント・ストアに追加エンティティ

ー・データを格納するにはメモリーと処理時間が必要ですが、そのイベントについてサブスクリプションがなければ、そのような追加格納は不要な場合もあります。

イベント・レコードにデータを設定するためには、オブジェクト・キーの列に名前と値のペアが設定されていることが必要です。例えば、ContractId がビジネス・オブジェクト内の属性の名前である場合、イベント・レコード内のオブジェクト・キー・フィールドは次のようになります。

```
ContractId=45381
```

アプリケーションによっては、オブジェクト・キーが、連結された複数のフィールドから構成される場合があります。このため、コネクターは、ContractId=45381:HeaderId=321 のように、区切り文字で区切られた名前 - 値ペアを複数サポートしています。区切り文字は、PollAttributeDelimiter コネクター構成プロパティーで設定されているように構成できます。区切り文字のデフォルト値はコロン (:) です。

イベント ID

各イベントには固有の ID が必要です。この ID は、アプリケーションにより生成される番号またはコネクターが使用している方式により生成される番号です。イベント ID 番号体系の例として、00123 など、イベントにより生成される順次 ID があります。この番号にコネクターがその名前を追加します。この結果オブジェクト・イベント ID は、ConnectorName_00123 となります。別の方法としては、タイム・スタンプの利用があります。この場合、ID は *ConnectorName*_06139833001001 のようになります。

オプションにより、コネクターはイベント ID を、ビジネス・オブジェクトの ObjectEventId 属性に格納することもできます。 ObjectEventId 属性は、IBM WebSphere Business Integration システムの中で各イベントを識別する固有の値です。この属性は必須であるため、アプリケーション固有のコネクターから値が提供されていない場合には、コネクター・フレームワークがその値を生成します。階層型ビジネス・オブジェクトに対応する ObjectEventId の値が指定されていない場合、コネクター・フレームワークは、親ビジネス・オブジェクトと各子ビジネス・オブジェクトに対応する値を生成します。コネクター・フレームワークが階層型ビジネス・オブジェクトに対応する ObjectEventId 値を生成すると、階層構造の各レベルにはかかわりなく、すべてのビジネス・オブジェクトにわたって各値が固有であることが必要です。

イベント状況

Java コネクターは、CWConnectorEventStatusConstants クラスに定義されたイベント状況値を使用します。表 47 に、イベント状況定数を示します。

表 47. Java コネクターのイベント状況値

イベント状況定数	説明
READY_FOR_POLL	ポーリング可能
SUCCESS	統合ブローカーへの送信
UNSUBSCRIBED	イベントについてのサブスクリプションなし
IN_PROGRESS	イベントが進行中

表 47. Java コネクターのイベント状況値 (続き)

イベント状況定数	説明
ERROR_PROCESSING_EVENT	イベントの処理中にエラーが発生しました。エラーの説明は、イベント・レコードの中のイベント記述の後に追加できます。
ERROR_POSTING_EVENT	イベントを統合ブローカーに送信中にエラーが発生しました。エラーの説明は、イベント・レコードの中のイベント記述の後に追加できます。
ERROR_OBJECT_NOT_FOUND	アプリケーション・データベースからイベントを取得中にエラーが発生しました。

イベント・ストアの可能な実装

アプリケーションは、イベント・ストアとして次のいずれかを使用することができます。

- 『イベント受信箱』
- 136 ページの『イベント表』
- 137 ページの『電子メール』
- 138 ページの『フラット・ファイル』

注: アプリケーションによっては、アプリケーション・エンティティーに加えられた変更を追跡する複数の方法を提供している場合があります。データベース表に対してはワークフロー、他の表に対してはユーザー出口を提供するアプリケーションなどがその例です。この場合、あるビジネス・オブジェクトについては 1 つの方法でイベントを処理し、別のビジネス・オブジェクトについては別の方法で処理する 1 つのイベント通知機構を連結した形で提供することが状況により必要となります。

イベント受信箱

組み込み受信箱機構を持つアプリケーションもあります。この受信箱機構を使用して、アプリケーションに関する情報を次のようにコネクターに転送することができます。

- イベント検出 — 受信箱内の項目を起動するエンティティーおよびイベントを識別することが必要な場合があります。
- イベント取得 — コネクターのアプリケーション固有コンポーネントが受信箱内の項目を取得できます。受信箱にアクセスできるインターフェースを提供する API が使用できる場合には、アプリケーション固有のコンポーネントがこの API を使用できます。

図 47 にこのような対話を示します。

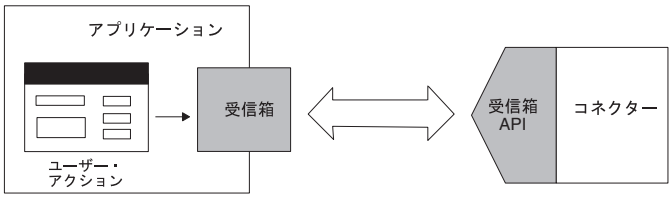

図 47. イベント・ストアとしてのイベント受信箱

イベント表

アプリケーションはそのアプリケーション・データベースを使用して、イベント情報を格納できます。アプリケーションはデータベースの中に特別なイベント表 を作成し、イベント・レコード用のイベント・ストアとして使用することができます。この表は、コネクターのインストール時に作成されます。イベント表をイベント・ストアとして使用した場合、次のようになります。

- イベント検出 — コネクターにとって関心のあるイベントが発生すると、アプリケーションがイベント・レコードをイベント表に格納します。
- イベント取得 — コネクターのアプリケーション固有コンポーネントが定期的にイベント表のポーリングを実行し、どのイベントも処理します。多くの場合、アプリケーションから提供されるデータベース (DB) API により、コネクターは、イベント表の内容にアクセスすることができます。

図 48 にこのような対話を示します。

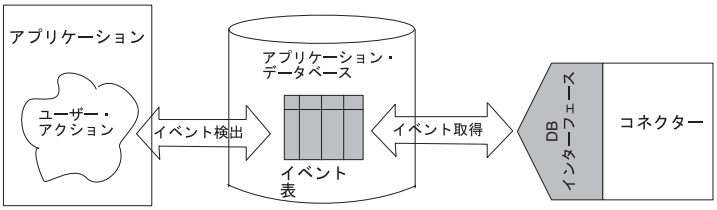

図 48. イベント・ストアとしてのイベント表

注: アプリケーション表が変更されたかどうかを判断するために、既存のアプリケーション表すべてをスキャンする必要はありません。そのために推奨される方法は、イベント表にイベント情報を移植し、イベント表のポーリングを実行することです。

コネクターがイベントのアーカイブをサポートしている場合には、アーカイブされたイベントを保持するため、アプリケーション・データベースの中にアーカイブ表を作成することもできます。表 48 に、イベント表とアーカイブ表を運用するために推奨されるスキーマを示します。このスキーマを、運用アプリケーションの必要に応じて拡張することができます。

表 48. イベント表およびアーカイブ表のために推奨されるスキーマ

列名	型	説明
event_id	データベースに適した型を使用します。	イベントに対応する固有のキー。システム制約から形式が決まります。

表 48. イベント表およびアーカイブ表のために推奨されるスキーマ (続き)

列名	型	説明
object_name	Char 80	ビジネス・オブジェクトの完全な名前。
object_verb	Char 80	イベントの動詞。
object_key	Char 80	オブジェクトの基本キー。
event_priority	Integer	イベントの優先順位。0 が最高の優先順位を表します。
event_time	DateTime	イベントのタイム・スタンプ (イベントが発生した時刻)。
event_processed	DateTime	アーカイブ表用のみ。イベントをコネクター・フレームワークに渡した時刻。
event_status	Integer	可能な状況値については、 134 ページの『イベント状況』を参照してください。
event_description	Char 255	イベント記述またはエラー・ストリング。
connector_id	Integer	コネクターの ID (該当する場合)。

電子メール

イベント・ストアとして電子メール・システムを使用できます。

- イベント検出 — アプリケーション・イベントが発生すると、アプリケーションが電子メール・メッセージをメールボックスに送ります。
- イベント取得 — コネクターのアプリケーション固有コンポーネントがメールボックスを検査し、イベント・メッセージを取得します。

図 49 にこのような対話を示します。

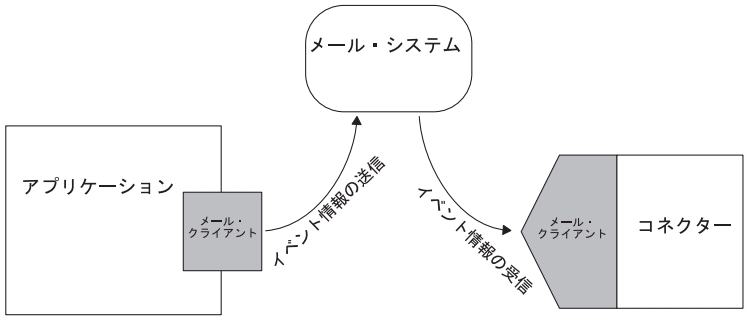

図 49. イベント・ストアとしてのメールボックス

電子メールをベースとするイベント・ストアの場合、コネクター用として使用されるメールボックスは構成可能であることが必要です。また、受信箱の実際の名前は、用途を反映したものとしてください。次に、イベント・メッセージの各フィールドのフォーマットと推奨される名前を示します。

- メッセージ属性 — 電子メール・メッセージは、通常、作成日時や優先順位などの属性を持っています。これらの属性をイベント通知機構で使用することができます。例えば、日付と時刻の属性を、イベント発生日時を表すために使用できます。
- 件名 — イベント・メッセージの件名は、次のようなフォーマットにすることができます。この例では、読みやすくするために、フィールドをスペースで区切っていますが、コネクターは別のフィールド区切り文字を使用することができます。

object_name object_verb event_id

event_id は、イベントの固有キーです。メール・メッセージに *event_id* キーを入れるか入れないかは、アプリケーションによります。*event_id* は、コネクター名、ビジネス・オブジェクト名、およびメッセージのタイム・スタンプもしくはシステム時刻の組み合わせから作成することができます。

- 本文 − イベント・メッセージの本文には、キーと値のペアを区切り文字により連結したシーケンスを設定できます。このキーと値のペアがオブジェクト・キーの要素です。例えば、特定の顧客とアドレスが CustomerId と AddrSeqNum の組み合わせにより一意的に識別される場合には、メール・メッセージの本文の例は次のようになります。

```
CustomerId 34225
AddrSeqNum 2
```

イベント・メッセージの本文は、ビジネス・オブジェクトの属性名と属性に挿入される値のリストにすることができます。

フラット・ファイル

他のイベント検出機構が使用できない場合には、フラット・ファイルを使用してイベント・ストアをセットアップすることも可能です。このタイプのイベント・ストアの場合、次のようになります。

- イベント検出 — アプリケーション内のイベント検出メカニズムによりイベント・レコードをファイルに書き込みます。
- イベント取得 — コネクターのアプリケーション固有コンポーネントがファイルを見つけ、イベント情報を読み取ります。

ファイルがコネクターから直接アクセスできない場合 (メインフレーム・システムで生成されたファイルの場合など)、ファイルを、コネクターからアクセスできる場所に転送することが必要です。ファイルを転送する 1 つの方法は、ファイル転送プロトコル (FTP) の使用です。コネクターの内部でファイル転送を実行する方法と、外部ツールを使用してファイルを 1 つの場所から別の場所にコピーする方法があります。ファイル間で情報を転送する方法は他にもあり、運用しているアプリケーションおよびコネクターに応じて適切な方法を選択してください。

図 50 に、フラット・ファイルを使用したイベント検出とイベント取得を示します。この例では、コネクターからアクセスできる場所にイベント情報を転送するために FTP を使用しています。

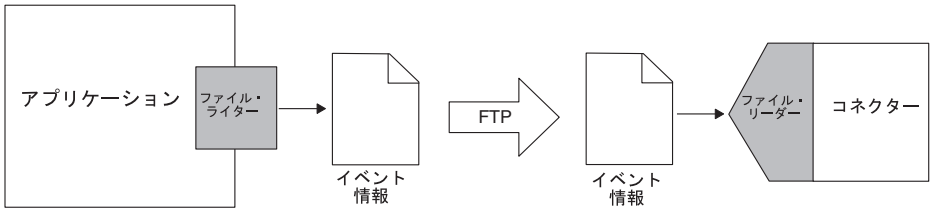

図 50. フラット・ファイルでのイベント・レコードの取得

イベント検出の実装

ほとんどのコネクターの場合、イベント検出機構を実装するようにアプリケーションを構成することが必要です。これは、コネクター・インストール作業の一環として、システム管理者が実行します。アプリケーションが構成されれば、アプリケーションによりエンティティーの変更が検出され、イベント・ストアにイベント・レコードが書き込まれます。次に情報はコネクターにより選択され、処理されます。このように、イベント通知機構はアプリケーションとコネクターの両方に実装されます。

このセクションでは、イベント検出の次の点について説明します。

- 『イベント検出機構』
- 142 ページの『イベント検出: 標準的な振る舞い』

イベント検出機構

イベントが起動する条件は、アプリケーション運用中のユーザーのアクション、アプリケーション・データの追加や変更を行うバッチ・プロセス、またはデータベース管理者のアクションです。イベント検出機構がアプリケーションの中でセットアップされ、ビジネス・オブジェクトに関連付けられたアプリケーション・イベントが発生した場合には、このアプリケーションがイベントを検出し、イベント・ストアに書き込むことが必要です。

イベント検出機構はアプリケーションに依存しています。コネクターなどクライアントによる使用を前提としたイベント検出機構を提供するアプリケーションもあります。イベント検出機構にイベント・ストアが組み込まれ、アプリケーションの変更に関する情報をイベント・ストアに挿入する方法がこの機構に定義されている場合もあります。例えば、1 つの実装のタイプとして、イベント・メッセージ・ボックスを使用し、コネクターの関心対象であるイベントをアプリケーションが処理するときには、必ずメッセージをメッセージ・ボックスに送ります。コネクターのアプリケーション固有コンポーネントがメッセージ・ボックスのポーリングを定期的に実施し、新しいイベント・メッセージを取り込みます。

組み込みのイベント検出機構を持たず、他の方法で、アプリケーション・エンティティーに加えられた変更に関する情報を提供するアプリケーションもあります。アプリケーションがイベント検出機構を提供していない場合には、コネクターのためにエンティティー変更情報を抽出できる何らかの機構を使用することが必要です。例えば、データベース・トリガーを実装したり、イベント・ストアに情報を書き込むプログラムを呼び出すユーザー出口を利用したり、フラット・ファイルからアプリケーション変更情報を抽出するなどの方法も可能です。

注: イベントが生成される方法はアプリケーションごとにかなり異なりますが、イベント通知機構の一部の特性は、すべての種類のアプリケーションに共通しています。例えば、どのタイプのイベント検出機構も、類似した内容のイベント・レコードを作成します。

以降のセクションで、イベントが検出され、イベント・ストアに書き込まれるときの 3 つの共通の方法について説明します。

- 140 ページの『フォーム・イベント』

- 『ワークフロー』
- 141 ページの『データベース・トリガー』

フォーム・イベント

一部のフォーム・ベースのアプリケーションは、特定のユーザー・アクションが発生したときに実行される フォーム・イベントを提供します。イベント検出をこのようにセットアップするには、特定の種類のイベントが発生したときに実行されるスクリプトを作成する必要があります。ユーザーがフォームを開き、スクリプトが関連付けられたアクションを実行すると、このスクリプトがイベント・レコードをイベント・ストアに格納します。

ほとんどの場合、フォーム・イベントはアプリケーション・ビジネス・プロセスの中で統合されているため、アプリケーション・ビジネス・ロジックをサポートしています。ただし、ユーザー・アクションにより起動されたアプリケーション・イベントのみが検出され、バッチ・プロセスなど他の方法でアプリケーション・データベースが直接更新されても、そのイベントは検出されません。

図 51 にフォーム・ベースのイベント検出機構を示します。ユーザーが Customer フォームに新規顧客を入力し、「OK」をクリックすると、スクリプトがイベント・レコードを生成し、イベント・ストアに格納します。

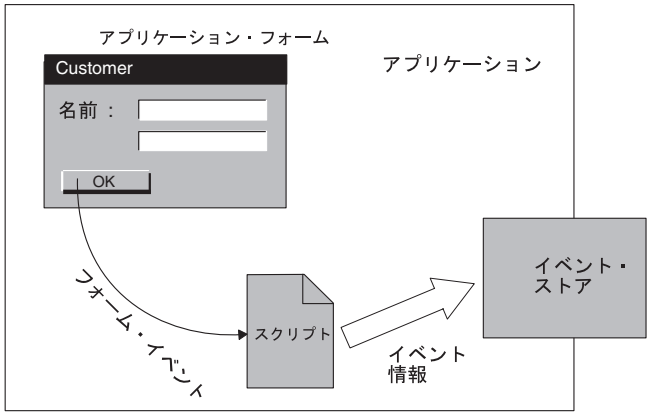

図 51. フォーム・ベースのイベント検出

ワークフロー

ビジネス・プロセスを追跡するために内部ワークフロー・システムを使用しているアプリケーションもあります。このワークフロー・システムを使用して、イベント検出用のイベントを生成することができます。

例えば、特定の操作が実行されたときにイベント・ストアに項目を挿入するワークフロー・プロセスを定義することができます。別の方法としては、イベント検出機構によりワークフロー・プロセスから情報をインターセプトし、この情報を使用してイベント・ストアにイベント・レコードを挿入することも可能です。ワークフロー・ベースのイベント検出機構を設計するときには、ワークフローのどの時点でイベント・レコードをイベント・ストアに書き込むか決定し、使用可能なアプリケーション機構を使用してイベント・レコードを生成する必要があります。

イベント検出にワークフロー・システムを使用することにより、イベント検出をアプリケーション・ビジネス・プロセスに確実に組み込むことができます。さらに、ワークフロー・システムは、生成されたアプリケーション・イベントを、ユーザーの介在なしに自動的に検出することができます。

図 52 に、ワークフロー・ベースのイベント検出機構を示します。特定の操作が実行されると、ワークフロー・プロセスが始動します。イベント検出機構がイベントに関する情報を受け取り、レコードをイベント・ストアに書き込みます。ワークフロー・プロセスは引き続き他のタスクを実行します。

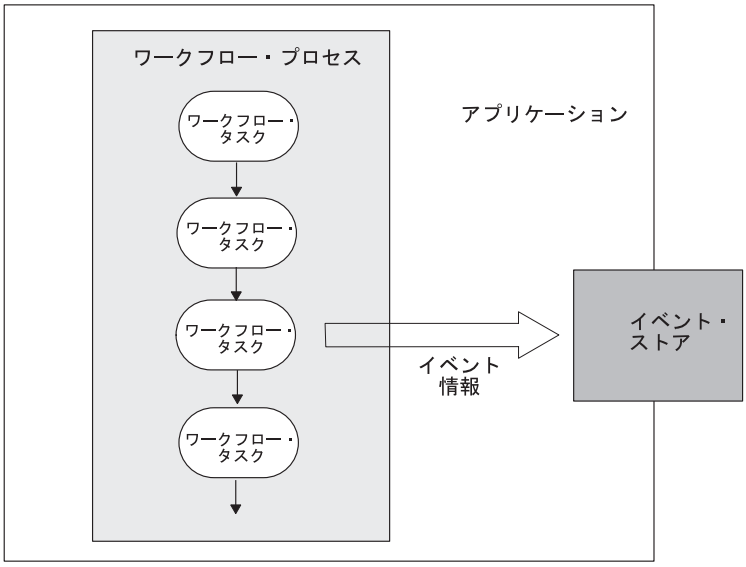

図 52. ワークフロー・ベースのイベント検出

データベース・トリガー

アプリケーションにイベント検出用の組み込みメソッドがなく、アプリケーションの実行対象となっているデータベースがデータベース・トリガーを提供する場合、アプリケーション表に加えられた変更を検出するために、行レベルのトリガーを実装することができます。トリガーは、コネクターによりサポートされるビジネス・オブジェクト定義に対応するアプリケーション表に挿入されます。

この機構の場合、トリガーにより生成されるイベント・レコードを格納するため、アプリケーション・データベース内に、イベント表をセットアップすることも必要です。アプリケーション・エンティティーが作成、更新、または削除されるたびに、トリガーがイベント表に行が挿入されます。各行が 1 つのイベント・レコードを表し、イベント表には、コネクターによる処理を待機するイベントが格納されます。

図 53 に、アプリケーション表 Customer を更新するユーザー・アクションを示します。Customer 表が更新されると、表上でトリガーが実行され、アプリケーション・データベース内のイベント表にイベント・レコードが書き込まれます。

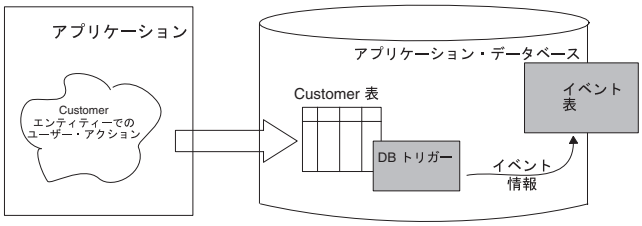

図 53. データベース・トリガーを使用したイベント検出

データベース・トリガーを使用する場合には、次の点に注意してください。

- 指定したトリガーによりアプリケーションの中ですでに使用されているトリガーが上書きされないことを確認します。
- アプリケーションが、トリガーに基づくイベント通知に適していることを確認します。例えば、データベースに複雑なビジネス・ルールを実装しているアプリケーションの場合、特定の表に対する単純なトリガーでは、完全なアプリケーション・イベントを正確には反映できない可能性があります。
- データベース・トリガーの欠点は、アプリケーション・データベースで表スキーマが変更されると、作成済みのトリガーに変更を加えることが状況により必要となることです。表スキーマが頻繁に変更され、多くのデータベース・トリガーをセットアップした場合には、トリガーの保守に要する時間がかなり多くなる可能性があります。

イベント検出: 標準的な振る舞い

アプリケーション・イベント検出機構は、次のステップを実行します。

- コネクターがサポートしているビジネス・オブジェクトに対応するイベントがアプリケーション・エンティティーで発生した場合に、このイベントを検出します。
- イベント・レコードを作成します。イベント検出機構は、レコード作成のため、次の動作を実行します。
  - オブジェクトの名前を、リポジトリー内のビジネス・オブジェクトの完全な名前に設定します。
  - 動詞を、データベースで発生したアクションに設定します。
  - オブジェクト・キーをアプリケーション・エンティティーの基本キーに設定します。
  - 固有なイベント ID を生成します。
  - イベント優先順位を設定します。
  - イベント・タイム・スタンプを設定します。
  - イベント状況を「ポーリング可能」に設定します。
- 完成したイベント・レコードをイベント・ストアに挿入します。

注: イベント検出機構には、新規イベントに対応するレコードを挿入する前に、既存のイベントとの重複についてイベント・ストアに照会できるオプションがあります。詳細については、143 ページの『イベント・ストアのフィルター操作によるイベント・レコードの重複の検出』を参照してください。

イベント・ストアに格納されたイベント・レコードは、コネクターのポーリング方式による選択を待機するキューに入ります。イベント・ストアはアプリケーションの内部にあることが必要です。アプリケーションが予期しない終了に遭遇した場合には、アプリケーションの復元時に、イベント・ストアを元の状態に復元することができます。また、コネクターのアプリケーション固有コードは、キューに入っているイベントを選択することができます。

イベント検出機構は、アプリケーション・イベントとイベント・ストアに書き込まれたイベント・レコードとの間のデータ保全性を確実にする必要があります。例えば、イベントに必要なすべてのデータ・トランザクションが正常に終了するまでは、イベント・レコードが生成されないことを確実にする必要があります。

以降のセクションでは、イベント検出機構で取り扱う問題を、次の点について説明します。

- 『イベント・ストアのフィルター操作によるイベント・レコードの重複の検出』
- 144 ページの『将来のイベント処理』

イベント・ストアのフィルター操作によるイベント・レコードの重複の検出

イベント検出機構は、重複したイベントがイベント・ストアに保管されない ように実装できます。この振る舞いにより、統合ブローカーが実行を必要とする処理量を最小にすることができます。例えば、コネクターからポーリングが実行され、次のポーリングが実行されるまでの間に、特定の Address オブジェクトが複数回アプリケーションにより更新された場合、すべてのイベントがイベント・ストアに保管される可能性があります。そして、コネクターはすべてのイベントに対応するビジネス・オブジェクトを作成し、これを InterChange Server に送ります。これを避けるため、イベント検出機構は、1 つの Update イベントのみが保管されるようにイベントにフィルターをかけることができます。

イベント検出機構は、新規イベントをレコードとしてイベント・ストアに格納する前に、新規イベントと一致する既存のイベントがないか、イベント・ストアに照会することができます。次の場合に、イベント検出機構は、新規イベントに対応するレコードの生成が禁止されます。

ケース 1	新規のイベント内のビジネス・オブジェクト名、動詞、キー、状況、および ConnectorId (該当する場合) がイベント・ストア内の別の未処理イベントの該当する項目と一致する。
ケース 2	新規イベントに対応するビジネス・オブジェクト名、動詞、キー、および状況がイベント表内の未処理イベントと一致し、さらに、新規イベントの動詞が Update で、未処理イベントの動詞が Create である。
ケース 3	新規イベントに対応するビジネス・オブジェクト名、キー、および状況がイベント表内の未処理イベントと一致し、さらに、イベント表内の未処理イベントの動詞が Create で、新規イベントの動詞が Delete である。この場合には、イベント・ストアから Create レコードを除去する。

注: イベント検出がストアード・プロシージャーとトリガーにより実装されている場合には、ストアード・プロシージャーが新規イベントに対応するレコードを挿入する前に、上記の照会を実行します。

将来のイベント処理

イベント検出機構は、イベント処理の将来日時を指定するようにセットアップできます。この機能を実装するには、対象となるイベント用に追加イベント・ストアをセットアップすることが必要な場合もあります。将来イベント・ストア内のイベント・レコードには、処理される時期を特定した日付を設定します。

この機能は、発効日の設定されたレコードを処理するアプリケーションの場合に必要です。既存の従業員が 1 か月後に昇進し、このとき昇給するなどの場合がこの例です。昇進の日に先立って、昇給の書類は完成しているため、この従業員の状況の変化により発効日付きのイベントが生成され、将来イベント表に格納されます。

イベント取得の実装

ほとんどのコネクターでは、アプリケーション固有のコンポーネントによりイベント取得機構が実装されます。これはコネクターの設計と実装の一環として、コネクター開発者が担当します。イベント検出機構がエンティティーの変更を検出し、イベント・レコードをイベント・ストアに書き込みますが、イベント取得機構は、このイベント検出機構と連携して動作します。イベント取得により、アプリケーション・イベントに関する情報が、イベント・ストアからコネクターのアプリケーション固有コンポーネントに転送されます。

このセクションでは、イベント取得の次の点について説明します。

- 『イベント取得機構』
- 『ポーリング機構の使用』

イベント取得機構

イベント・ストアからイベント・レコードを取得するために使用される共通機構は次の 2 つです。

- イベント・コールバック機構 — この機構を介して、コネクターはアプリケーション・イベントの通知を受け取ります。ただし、アプリケーション・イベント用のイベント・コールバック API を提供しているアプリケーションは少数にとどまっています。
- ポーリング機構 — 最も一般的なイベント取得機構は、ポーリング機構です。

ポーリング機構の使用

アプリケーションは、ポーリング機構のため、データベース表や受信箱のような永続的なイベント・ストアを提供します。アプリケーション・エンティティーに変更が発生すると、アプリケーションはイベント・レコードをこのイベント・ストアに書き込みます。コネクターは、コネクターがサポートするビジネス・オブジェクト定義に対応したエンティティーの変更の有無について、定期的にイベント・ストアを検査します (イベント・ストアのポーリングを実行します)。一般に、イベント・ストアに保管される唯一のビジネス・オブジェクト情報は、操作のタイプとアプリケーション・エンティティーのキー値です。コネクターはイベントを処理するとき、アプリケーション・エンティティー・データの残りの部分を取り込みます。コネクターは、イベントの処理を終了すると、イベント・レコードをイベント・ストアから除去し、アーカイブ・ストアに格納します。

イベント取得を実行するポーリング機構を実装するために、コネクターのアプリケーション固有コンポーネントは、pollForEvents() メソッドと呼ばれるポーリング・メソッド を使用します。ポーリング・メソッドはイベント・ストアを検査し、新規イベントを取得し、各イベントを処理してから復帰します。

このセクションでは、ポーリング・メソッドの次の点について説明します。

- 『ポーリング間隔』
- 『イベント・ポーリング: 標準的な振る舞い』

ポーリング間隔

コネクター・フレームワークは、PollFrequency コネクター構成プロパティーで定義された指定*ポーリング間隔* で、ポーリング・メソッドを呼び出します。このプロパティーは、コネクターのインストール時に Connector Configurator により初期化されます。通常、ポーリング間隔は約 10 秒です。

注: イベント情報を取得するためにコネクターがポーリングを実行する必要がない場合には、PollFrequency プロパティーを 0 に設定することにより、ポーリングをオフにします。

したがって、コネクター・フレームワークは、次のいずれかの条件で pollForEvents () メソッドを呼び出すことになります。

- PollFrequency が正の値に設定されている。
- コネクター始動スクリプトで、*fPollFreq* オプションの値が指定されている。

イベント・ポーリング: 標準的な振る舞い

図 54 に、ポーリング・メソッドの基本的な振る舞いを示します。

1. コネクター・フレームワークが、ポーリングを開始するため、アプリケーション固有コンポーネントの pollForEvents() メソッドを呼び出します。
2. pollForEvents() メソッドがアプリケーション内のイベント・ストアで、新規イベントの有無を検査し、イベントを取り込みます。
3. ポーリング・メソッドがコネクター・フレームワークに照会することにより、イベントにサブスクライバーがあるかどうか判断します。
4. イベントにサブスクライバーがある場合には、ポーリング・メソッドは、アプリケーションから、ビジネス・オブジェクトに対応するデータの完全なセットを取り込みます。
5. ポーリング・メソッドは、ビジネス・オブジェクトをコネクター・フレームワークに送り、コネクター・フレームワークは、ビジネス・オブジェクトを宛先 (InterChange Server など) まで転送します。

ポーリング・メソッドは、呼び出されるたびに、新規のイベントの有無を検査し、あれば取り込み、イベントにサブスクライバーがあるかどうか判断し、サブスクライバーのあるイベントに対応するアプリケーション・データを取り込み、ビジネス・オブジェクトを InterChange Server に送ります。

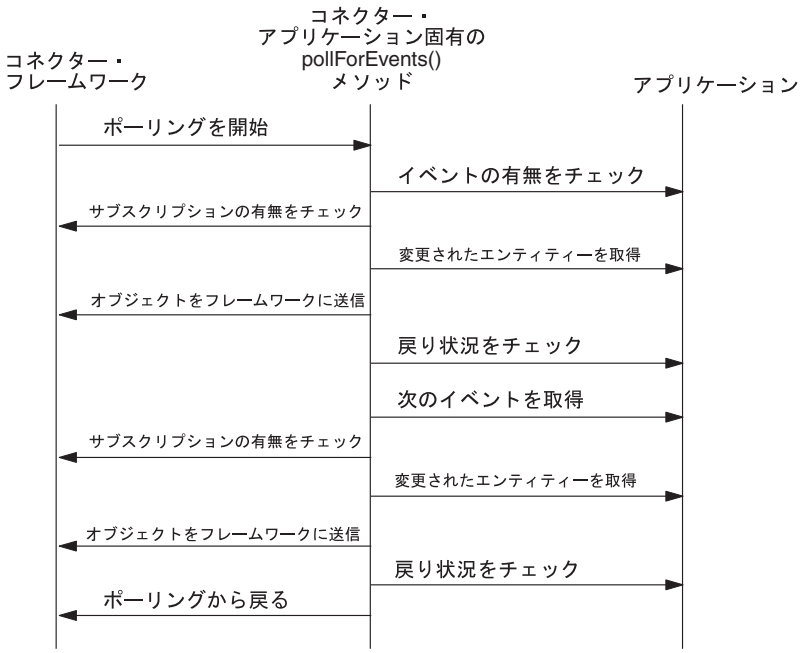

図 54. pollForEvents() メソッドの基本的な振る舞い

`pollForEvents()` メソッドの実装方法については、146 ページの『ポーリング・メソッドの実装』を参照してください。

ポーリング・メソッドの実装

アプリケーションがイベント・ストアを提供している場所が、表か、受信箱か、その他の場所かにはかかわりなく、コネクターはイベント情報を取得するため、定期的にポーリングを実行する必要があります。コネクターのポーリング・メソッド `pollForEvents()` は、イベント・ストアのポーリングを実行し、イベント・レコードを取得し、イベントを処理します。ポーリング・メソッドは、イベントを処理するため、イベントにサブスクライバーがあるか判断し、イベントがカプセル化されているアプリケーション・データを格納している新しいビジネス・オブジェクトを作成し、このビジネス・オブジェクトをコネクター・フレームワークに送ります。

注: コネクターによる実装の対象が要求の処理であり、イベント通知ではない場合には、`pollForEvents()` の完全な実装は必要がない場合もあります。ただし、ポーリング・メソッドは、Java コネクター・ライブラリーでデフォルトのインプリメンテーションとし定義されているため、ポーリングはすでにインプリメントされています。ポーリングを使用不可にする必要がある場合は、このメソッドのスタブをインプリメントすることができます。

このセクションでは、`pollForEvents()` メソッドの実装方法の次の点について説明します。

- 147 ページの『pollForEvents() の基本ロジック』
- 147 ページの『ポーリングに関するその他の問題』

pollForEvents() の基本ロジック

pollForEvents() メソッドは、通常はイベント処理に基本ロジックを使用します。図 55 に、ポーリング・メソッドの基本ロジックのフローチャートを示します。

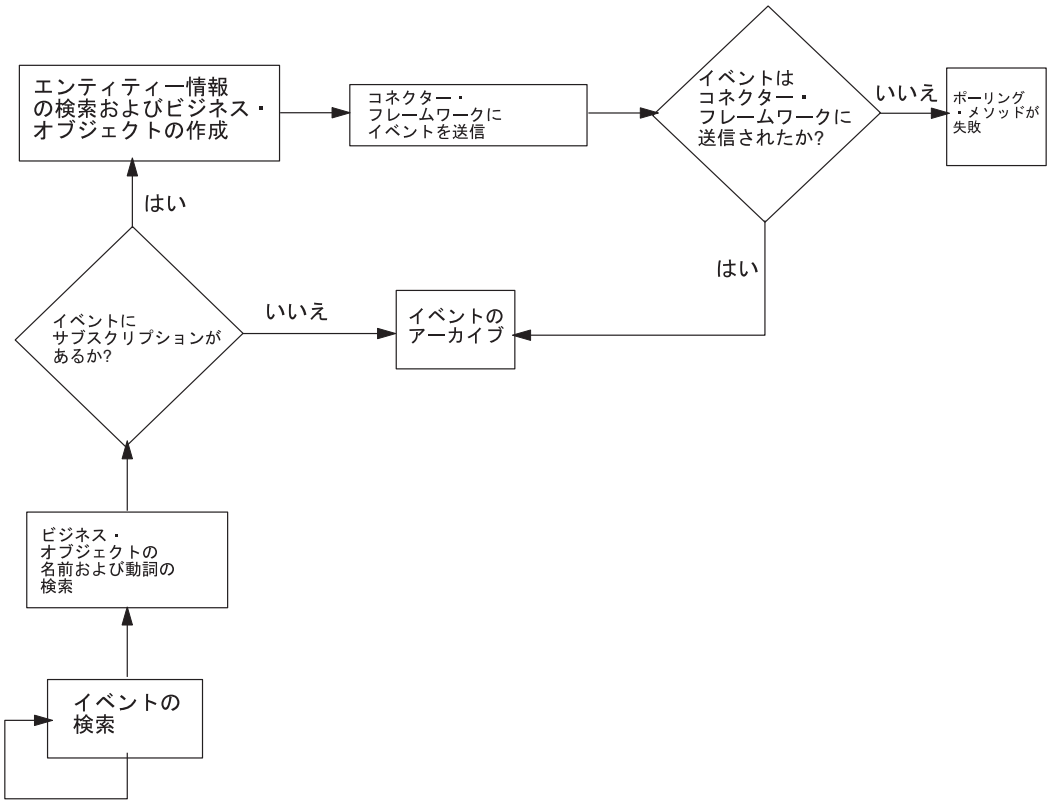

図 55. pollForEvents() の基本ロジックのフローチャート

この基本的ポーリング・ロジックのインプリメントについては、 203 ページの『イベント通知機構の実装』を参照してください。

注: ポーリング・メソッドのフローの中で発生するイベント状況値については、 361 ページの表 129 を参照してください。

ポーリングに関するその他の問題

このセクションでは、ポーリングに関する次の問題について説明します。

- 『イベントのアーカイブ』
- 149 ページの『スレッド化の問題』
- 149 ページの『イベント優先順位によるイベントの処理』
- 150 ページの『イベント分配』

イベントのアーカイブ

コネクターは、処理したイベントをアーカイブすることができます。処理済みまたはアンサブスクライブされたイベントをアーカイブすることにより、そのイベントの逸失を確実に防止できます。通常、アーカイブのステップは次のとおりです。

- イベント・レコードをイベント・ストアからアーカイブ・ストアにコピーします。

  アーカイブ・ストア の目的はイベント・ストアと同じです。すなわち、アーカイブ・レコードを、コネクターに処理されるまでの間、永続キャッシュに保管します。*アーカイブ・レコード* には、イベント・レコードと同じ基本情報が格納されています。
- アーカイブ・ストア内のイベントのイベント状況を更新します。

  アーカイブ・レコードは、表 49 に示したイベント状況値の 1 つを持つように更新します。
- イベント・ストアからイベント・レコードを削除します。

表 49. アーカイブ・レコードのイベント状況値

状況	説明
Success	イベントは検出され、コネクターはそのイベントに対応するビジネス・オブジェクトを作成し、それをコネクター・フレームワークに送信しました。
Unsubscribed	イベントが検出されたが、イベントへのサブスクリプションがないため、イベントがコネクター・フレームワークを経て統合ブローカーに送信されませんでした。
Error	イベントは検出されましたが、コネクターがイベントの処理中に、エラーが発生しました。イベントに対応するビジネス・オブジェクトの作成中またはビジネス・オブジェクトをコネクター・フレームワークに送信中にエラーが発生しました。

このセクションでは、イベント・アーカイブの次の点について説明します。

- 『アーカイブ・ストアの作成』
- 『コネクターのアーカイブ用構成』
- 149 ページの『アーカイブ・ストアへのアクセス』

アーカイブ・ストアの作成: アプリケーションがアーカイブ・サービスを提供している場合、このサービスを利用できます。それ以外の場合、通常は、アーカイブ・ストアは、イベント・ストアと同じ機構を使用して実装します。

- データベース・トリガーを使用するイベント通知機構の場合、イベント・アーカイブをセットアップする 1 つの方法は、イベント表に削除トリガーをインストールすることです。コネクターのアプリケーション固有コンポーネントがイベント表から処理済みまたはアンサブスクライブされたイベントを削除すると、削除トリガーがこのイベントをアーカイブ表に移動させます。イベント表については、136 ページの『イベント表』を参照してください。

  注: コネクターがイベント表を使用している場合、管理者による定期的なアーカイブのクリーンアップが状況により必要です。
- 電子メールによるイベント通知方式の場合、メッセージを別のフォルダーに移すことで、アーカイブを実現できる場合があります。`Archive` と呼ばれるフォルダーを使用して、イベント・メッセージをアーカイブすることができます。

コネクターのアーカイブ用構成: アーカイブは、アーカイブ・ストアの構築およびイベント・レコードのアーカイブ・ストアへの移動の場面で、パフォーマンスに影

響を及ぼす場合があります。したがって、イベントをアーカイブするかどうかをシステム管理者が制御できるように、イベント・アーカイブを構成できるようにインストール時に設計しておく方法が考えられます。アーカイブを構成可能にするために、あるコネクター固有の構成プロパティーを作成することにより、アンサブスクライブされたイベントをコネクターがアーカイブするかどうか指定できます。この構成プロパティーの名前として、IBM は ArchiveProcessed をお勧めしています。この構成プロパティーがアーカイブなしを指定している場合には、コネクターのアプリケーション固有コンポーネントはイベントを削除するか、無視することができます。コネクターのパフォーマンス制約が大きい場合や、イベントのボリュームが非常に大きい場合には、イベントのアーカイブは必須ではありません。

アーカイブ・ストアへのアクセス: コネクターによるアーカイブは、コネクターのポーリング・メソッド pollForEvents() を使用したイベント処理の一環として実行されます。コネクターによるイベントの処理が終了した後、イベントが正常にコネクター・フレームワークにデリバリーされたかどうかにはかかわりなく、コネクターはイベントをアーカイブ・ストアに移動させる必要があります。サブスクリプションのないイベントもアーカイブに移動します。処理済みまたはアンサブスクライブされたイベントをアーカイブすることにより、そのイベントの逸失を確実に防止できます。

ポーリング・メソッドを設計するときには、次のいずれかの条件が成立したときにイベントをアーカイブするように考慮してください。

- ポーリング・メソッドがイベントを処理し、コネクター・フレームワークがビジネス・オブジェクトをデリバリーしたとき
- イベントに対してサブスクリプションが存在しないとき

注: コネクターがイベント表を使用している場合、管理者による定期的なアーカイブのクリーンアップが状況により必要です。例えば、ディスク・スペースに空きをつくるため、管理者がアーカイブを切り捨てることが必要な場合もあります。

スレッド化の問題

Java コネクターはスレッド・セーフである必要があります。コネクター・フレームワークは、イベント・デリバリー (pollForEvents() メソッドの実行) と要求処理 (doVerbFor() メソッドの実行) を実行するために複数のスレッドを使用することができます。

イベント優先順位によるイベントの処理

イベント優先順位により、コネクターのポーリング・メソッドは、イベント・ストア内のイベント数が、1 回のポーリングでコネクターが取得できる最大イベント数を超える状況にも対応できます。このタイプのポーリング実装では、ポーリング・メソッドは、優先順位に従ってイベントのポーリングと処理を実行します。イベントの優先順位は、0 から n の範囲の整数で定義され、0 が最高の優先順位を表します。

イベントをその優先順位に従って処理するには、イベント通知機構に次のタスクを実装することが必要です。

- イベント検出機構は、イベント・レコードをイベント・ストアに保管するとき、優先順位の値を割り当てることが必要です。

- イベント取得機構 (ポーリング機構) は、処理対象のイベント・レコードを取得する順序を、優先順位に基づいて指定することが必要です。

注: イベントが選択された後、優先順位の値を繰り下げることはしていません。したがって、低い優先順位のイベントが選択されずに残るケースがまれには発生します。

次の例の SQL SELECT ステートメントは、イベントの優先順位に基づいて、コネクターがイベント・レコードを選択する方法を示しています。SELECT ステートメントでイベントに優先順位によるソートをかけ、コネクターが順番にイベントを処理していきます。

```
SELECT event_id, object_name, object_verb, object_key
FROM event_table
WHERE event_status = 0 ORDER BY event_priority
```

ポーリング・メソッドのロジックは、147 ページの『pollForEvents() の基本ロジック』で説明したロジックと同じです。

イベント分配

イベント検出機構とイベント取得機構は、複数のコネクターが同じイベント・ストアにポーリングを実行できるように実装することができます。各コネクターは、特定のイベントを処理し、特定のビジネス・オブジェクトを作成し、それらを InterChange Server に渡すように構成できます。これにより特定のタイプのイベントの処理を合理化し、アプリケーションからのデータ転送量を増加させることができます。

複数のコネクターがイベント・ストアのポーリングを実行できるようにイベント分配を実装する手順は次のとおりです。

- イベント・レコードに整数のコネクター ID 用の列を追加し、どのコネクターがイベントを選択するか指定できるようにイベント検出機構を設計します。

  これはアプリケーション・エンティティーごとに実行できます。例えば、イベント検出機構の指定に基づいて、すべての Customer イベントを、connectorId フィールドが 4 に設定されたコネクターによって選択させることが可能です。
- ConnectorId という名前のアプリケーション固有コネクター・プロパティーを追加します。各コネクターに固有 ID を割り当て、この値を ConnectorId プロパティーに設定します。
- ConnectorId プロパティーの値を照会するように、ポーリング・メソッドを実装します。このプロパティーが設定されていなければ、ポーリング・メソッドは、通常と同様に、イベント・ストアからすべて のイベント・レコードを取得することができます。このプロパティーがコネクター ID の値に設定されていれば、ポーリング・メソッドは、ConnectorId プロパティーに一致したイベントのみ を取得します。

イベント処理に関する特別な考慮事項

このセクションでは、イベント処理について以下の内容を説明します。

- 151 ページの『削除イベントの処理』
- 152 ページの『保証付きイベントのデリバリーの使用』

削除イベントの処理

アプリケーションは次のいずれかのタイプの削除操作をサポートできます。

- 物理削除 — データがデータベースから物理的に削除される操作です。
- 論理削除 — データベース・エンティティー内の状況カラムが非アクティブまたは無効な状況に設定され、一方データ自体はデータベースから削除されていません。

アプリケーションと整合の取れた方法で削除イベントを処理するような実装が望ましいと思われるかもしれません。例えば、アプリケーション・エンティティーが削除されたとき、物理削除をサポートしているアプリケーションに対応したコネクター・ポーリング・メソッドで、Delete 動詞を持つビジネス・オブジェクトをパブリッシュするなどの方法です。論理削除をサポートしているアプリケーションに対応したコネクター・ポーリング・メソッドで、Update 動詞を持ちステータス値が非アクティブに変更されたビジネス・オブジェクトをパブリッシュします。

ソース・アプリケーションと宛先アプリケーションがサポートしている削除モデルが異なると、この方法では問題の発生を招くことがあります。ソース・アプリケーションが論理削除を、宛先アプリケーションが物理削除をサポートしている状況を想定してみましょう。また、ソース・アプリケーションと宛先アプリケーションの同期をエンタープライズにより実現していると仮定します。ソース・コネクターが状況の変更 (つまり、削除イベント) を、Update 動詞付きビジネス・オブジェクトとして送信すると、宛先コネクターは、このビジネス・オブジェクトが実際に削除イベントを表すか判断できない場合があります。

したがって、イベントのパブリッシュを設計するときには、両方のタイプのアプリケーションに対応するソース・コネクターが削除イベントを適切にパブリッシュし、その結果、宛先コネクターが正しくイベントを処理できることが必要です。イベント通知ビジネス・オブジェクトの中の Delete 動詞は、削除操作が物理削除、論理削除のいずれであったかにかかわりなく、データが削除されたイベントを表すことが必要です。この結果、宛先コネクターは、削除イベントについての通知を正しく受け取ることができます。

このセクションでは、削除イベントの処理を実装する方法の次の点に関して説明します。

- 『イベント・レコード内の動詞の設定』
- 152 ページの『ビジネス・オブジェクト内の動詞の設定』
- 152 ページの『マッピング時の動詞の設定』

イベント・レコード内の動詞の設定

論理削除のコネクターと物理削除のコネクターの両方に対応するイベント検出機構が、イベント・レコード内の動詞を Delete に設定します。

- 物理削除のコネクターの場合、これは標準実装で実現される機能です。
- 論理削除をサポートしているアプリケーションのコネクターの場合、更新イベントがどの時点で実際のデータ削除を表出するか決定できるように、イベント検出機構を設計することが必要です。

  言い換えると、変更されたエンティティーに対応する更新イベントと論理的に削除されたエンティティーに対応する更新イベントを区別する必要があります。論

理的に削除されたエンティティーに対して、イベント検出機構は、たとえアプリケーション内のイベントが状況カラムを更新した更新イベントであった場合でも、イベント・レコード内の動詞を Delete に設定します。

ビジネス・オブジェクト内の動詞の設定

論理削除のコネクターと物理削除のコネクターの両方に対応するポーリング・メソッドが Delete 動詞を持つビジネス・オブジェクトを生成します。

- アプリケーションが論理削除をサポートしている場合、コネクターのポーリング・メソッドがイベント・ストアから削除イベントを取得し、空のビジネス・オブジェクトを作成し、キーを設定し、動詞を Delete に設定してから、ビジネス・オブジェクトをコネクター・フレームワークに送ります。

  階層を形成するビジネス・オブジェクトの場合、コネクターは、削除された子ビジネス・オブジェクトは送信しません。コネクターは、非アクティブ状況のエンティティーを照会の対象から強制的に外すことができます。あるいは、非アクティブ状況の子ビジネス・オブジェクトをマッピング時に除去することができます。
- アプリケーションが物理削除をサポートしている場合、コネクターはアプリケーション・データを取得できない場合があります。この場合、コネクターのポーリング・メソッドがイベント・ストアから削除イベントを取得し、空のビジネス・オブジェクトを作成し、キー値を設定し、他の属性の値を特別な Ignore 値 (CxIgnore) に設定し、ビジネス・オブジェクト内の動詞を Delete に設定してから、ビジネス・オブジェクトをコネクター・フレームワークに送ります。

マッピング時の動詞の設定

WebSphere InterChange Server

アプリケーション固有のビジネス・オブジェクトと汎用ビジネス・オブジェクトの間のマッピングが実行されると、動詞は Delete にマップされます。この結果、イベントに関する正しい情報が確実にコラボレーションに送られます。コラボレーションは、この動詞に基づいて、特別な処理を実行する場合があります。

関係表については次の推奨事項を順守してください。

- 論理削除アプリケーションの削除イベントの場合、関係項目は関係表に残してください。
- 物理削除アプリケーションの削除イベントの場合、関係項目は関係表から削除してください。

保証付きイベントのデリバリーの使用

保証付きイベント・デリバリー機能を使用することにより、コネクター・フレームワークは、コネクターのイベント・ストアと統合ブローカーとの間でのイベントの二重送信を確実に回避できます。

重要: この機能は、JMS 対応コネクターでのみ 使用可能です。JMS 対応コネクターとは、JMS (Java Messaging Service) を使用して、コネクターのメッセージ転送用のキューを処理するコネクターのことです。 JMS 対応コネクターで

は、必ず DeliveryTransport コネクター・プロパティーを JMS に設定します。コネクターを開始すると、そのコネクターは JMS トランスポートを使用するため、コネクターと統合ブローカー間での以降のやり取りはすべて、このトランスポートを介して行われます。 JMS トランスポートでは、最終的にメッセージが宛先に移送されることが保証されます。

保証付きイベント・デリバリー機能を使用しない場合は、コネクターがイベントをパブリッシュしたとき (コネクターがその pollForEvents() メソッド内部から gotApplEvent() メソッドを呼び出したとき) から、コネクターがイベント・レコードを削除してイベント・ストアを更新 (あるいはイベント・ストアを「event posted」状況で更新) するまでの間に、ウィンドウが表示されるとたちまち障害が発生する可能性があります。このウィンドウの表示中で障害が発生した場合、そのイベントは送信されますが、そのイベント・レコードは「ready for poll」状況のままイベント・ストア内に留まります。コネクターは再開時に、まだイベント・ストア内にあるこのイベント・レコードを見つけて送信するため、イベントが二重送信される結果になります。

次のいずれかの方法で、JMS 対応のコネクターに保証付きイベント・デリバリー機能を付加することができます。

- *コンテナー管理のイベント* 機能を使用する方法: コネクターが (JMS ソース・キューとして実装された) JMS イベント・ストアを使用している場合は、コネクター・フレームワークがコンテナーの役割を果たし、JMS イベント・ストアを管理します。詳細については、『JMS イベント・ストアを使用するコネクター用の保証付きイベント・デリバリー』を参照してください。
- *重複イベント回避* 機能を使用する方法: コネクター・フレームワークで JMS モニター・キューを使用して、重複イベントが発生しないようすることができます。この機能は通常、非 JMS イベント・ストア (例えば、JDBC 表、電子メール・メールボックス、またはフラット・ファイルなどとして実装されたイベント・ストア) を使用するコネクターのために使用されます。詳細については、156 ページの『非 JMS イベント・ストアを使用したコネクター用の保証付きイベント・デリバリー』を参照してください。

JMS イベント・ストアを使用するコネクター用の保証付きイベント・デリバリー

JMS 対応のコネクターが JMS キューを使用してイベント・ストアを実装している場合、コネクター・フレームワークは「コンテナー」として機能して、JMS イベント・ストア (JMS ソース・キュー) を管理できます。 JMS の役割の 1 つは、トランザクション・キュー・セッションの開始後、コミットが発行されるまでメッセージをキャッシュすることです。障害が発生した場合またはロールバックが発行された場合は、メッセージが破棄されます。したがって、コネクター・フレームワークは、1 つの JMS トランザクションで、メッセージをソース・キューから除去し、そのメッセージを宛先キューに入れることができます。保証付きイベント・デリバリーの、このコンテナー*管理のイベント* 機能を使用することにより、コネクター・フレームワークは、JMS イベント・ストアと宛先の JMS キューとの間でのイベントの二重送信を確実に回避できます。

このセクションでは、JMS イベント・ストアを備えた JMS 対応コネクターの保証付きイベント・デリバリー機能の使用について、以下の内容を説明します。

- 『JMS イベント・ストアを使用するコネクターでの機能の使用可能化』
- 155 ページの『イベント・ポーリングへの影響』

JMS イベント・ストアを使用するコネクターでの機能の使用可能化: JMS イベント・ストアを有する JMS 対応コネクターで保証付きイベント・デリバリー機能を使用可能にするには、表 50 に示されているコネクター構成プロパティーを設定します。

表 50. JMS イベント・ストアを使用するコネクター用の保証付きイベント・デリバリー・コネクター・プロパティー

コネクター・プロパティー	値
DeliveryTransport	JMS
ContainerManagedEvents	JMS
PollQuantity	イベント・ストアの 1 回のポーリングで処理されるイベントの数。
SourceQueue	JMS ソース・キュー (イベント・ストア) の名前。コネクター・フレームワークはこのキューをポーリングし、処理対象イベントを取り出します。 **注:** ソース・キューと他の JMS キューは、同じキュー・マネージャーの一部になっていなければなりません。異なるキュー・マネージャーに保管されているイベントをコネクターのアプリケーションで生成する場合は、リモート・キュー・マネージャー用のリモート・キュー定義を定義する必要があります。次に WebSphere MQ は、リモート・キューから取得したイベントを、JMS 対応のコネクターが統合ブローカーへの送信用に使用するキュー・マネージャーに転送します。リモート・キュー定義の構成方法については、IBM WebSphere MQ の資料を参照してください。

注: コネクターは、コンテナー管理のイベントと重複イベント回避の 2 つの保証付きイベント・デリバリー機能のうちの *1* つのみ を使用することができます。したがって、ContainerManagedEvents プロパティーを JMS に設定し、さらに DuplicateEventElimination プロパティーを true に設定することはできません。

また、コネクターの構成に加えて、JMS ストア内のイベントとビジネス・オブジェクト間の変換を行うデータ・ハンドラーの構成も必要です。このデータ・ハンドラー情報は、表 51 に示すコネクター構成プロパティーから構成されます。

表 51. 保証付きイベント・デリバリーのデータ・ハンドラー・プロパティー

データ・ハンドラー・プロパティー	値	必須ですか?
MimeType	データ・ハンドラーによって処理される MIME タイプ。呼び出し対象となるデータ・ハンドラーは、この MIME タイプで識別されます。	はい
DHClass	データ・ハンドラーを実装する Java クラスのフルネーム。	はい
DataHandlerConfigMOName	MIME タイプとそのデータ・ハンドラーを関連付けるトップレベルのメタオブジェクトの名前。	オプショナル

注: データ・ハンドラー構成プロパティーは、他のコネクター構成プロパティーと一緒に、コネクター構成ファイル内に存在します。

保証付きイベント・デリバリーが使用されるように JMS イベント・ストアを有するコネクターを構成するエンド・ユーザーは、表 50 および 表 51 の説明にしたがって、コネクター・プロパティーを設定する必要があります。これらのコネクター構成プロパティーの設定には、Connector Configurator ツールを使用します。Connector Configurator は、表 50 のコネクター・プロパティーを「標準プロパティー (Standard Properties)」タブに表示します。また、表 51 のコネクター・プロパティーを「データ・ハンドラー (Data Handler)」タブに表示します。

注: Connector Configurator は、コネクター構成プロパティーの DeliveryTransport と ContainerManagedEvents が JMS に設定されている場合にのみ「データ・ハンドラー (Data Handler)」タブのフィールドをアクティブにします。

Connector Configurator の詳細については、569 ページの『付録 B. Connector Configurator』を参照してください。

イベント・ポーリングへの影響: ContainedManagedEvents を JMS に設定して、保証付きイベント・デリバリーを使用するコネクターは、この機能を使用しないコネクターとは少し異なる動作をします。コネクター・フレームワークは、コンテナー管理のイベントを提供するために、次のステップに従ってイベント・ストアをポーリングします。

1. JMS トランザクションを開始します。
2. イベント・ストアから JMS メッセージを読み取ります。

   イベント・ストアは、JMS ソース・キューとして実装されます。JMS メッセージにはイベント・レコードが含まれています。JMS ソース・キューの名前は、SourceQueue コネクター構成プロパティーから取得されます。
3. イベントをビジネス・オブジェクトに変換するための、適切なデータ・ハンドラーを呼び出します。

   コネクター・フレームワークは、表 51 で示されたプロパティーで構成されたデータ・ハンドラーを呼び出します。

4. 統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、WebSphere Business Integration Message Broker) または WebSphere Application Server の場合は、ビジネス・オブジェクトを構成済みのワイヤー・フォーマット (XML) に基づくメッセージに変換してください。
5. 変換後のメッセージを JMS 宛先キューに送信します。

WebSphere InterChange Server
JMS 宛先キューに送信されたメッセージはビジネス・オブジェクトです。

その他の統合ブローカー
JMS 宛先キューに送信されるメッセージは XML メッセージです。

6. JMS トランザクションをコミットします。

   JMS トランザクションがコミットされると、同一のトランザクションで、メッセージが JMS 宛先キューに書き込まれ、JMS リソース・キューから除去されます。
7. ステップ 1 から 6 までのループを繰り返します。このループの繰り返し回数は、`PollQuantity` コネクター・プロパティーによって決定されます。

重要: `ContainerManagedEvents` プロパティーが `JMS` に設定されているコネクターは、イベントのポーリングを実行する際に `pollForEvents()` メソッドを呼び出し *ません*。コネクターの基底クラスに `pollForEvents()` メソッドが組み込まれている場合、このメソッドは*起動されません*。

非 JMS イベント・ストアを使用したコネクター用の保証付きイベント・デリバリー

コネクター・フレームワークは、*重複イベント回避* 機能を使用することにより、重複イベントの発生を回避することができます。この機能は通常、非 JMS のソリューションを使用してイベント・ストア (JDBC イベント表、電子メール・メールボックス、またはフラット・ファイルなど) を実装する JMS 対応コネクターのために使用することができます。保証付きイベント・デリバリーの、この重複イベント回避機能を使用することにより、コネクター・フレームワークは、イベント・ストアと宛先の JMS キューとの間でのイベントの二重送信を確実に回避できます。

注: JMS イベント・ストアを使用する JMS 対応コネクターは、通常、コンテナー管理のイベント機能を使用します。ただし、コンテナー管理のイベント機能に代えて重複イベント回避機能を使用することができます。

このセクションでは、非 JMS イベント・ストアを使用する JMS 対応コネクターでの保証付きイベント・デリバリー機能の使用について、以下の内容を説明します。

- 157 ページの『非 JMS イベント・ストアを使用するコネクターでの機能の使用可能化』
- 155 ページの『イベント・ポーリングへの影響』

非 JMS イベント・ストアを使用するコネクターでの機能の使用可能化: JMS 以外のイベント・ストアを有する JMS 対応コネクターで保証付きイベント・デリバリー機能を使用可能にするには、表 52 に示されているコネクター構成プロパティーを設定してください。

表 52. 非 JMS イベント・ストアを使用するコネクター用の保証付きイベント・デリバリー・コネクター・プロパティー

コネクター・プロパティー	値
DeliveryTransport	JMS
DuplicateEventElimination	true
MonitorQueue	コネクター・フレームワークが処理対象のビジネス・オブジェクトの ObjectEventId を保管する JMS モニター・キューの名前

注: コネクターは、コンテナー管理のイベントと重複イベント回避の 2 つの保証付きイベント・デリバリー機能のうちの *1* つのみ を使用することができます。したがって、DuplicateEventElimination プロパティーを true に設定し、さらに ContainerManagedEvents プロパティーを JMS に設定することはできません。

保証付きイベント・デリバリーが使用されるようにコネクターを構成したエンド・ユーザーは、表 52 に記載されているコネクター・プロパティーを指示に従って設定する必要があります。これらのコネクター構成プロパティーの設定には、Connector Configurator ツールを使用します。このツールは、これらのコネクター・プロパティーを「標準プロパティー (Standard Properties)」タブに表示します。Connector Configurator の詳細については、569 ページの『付録 B. Connector Configurator』を参照してください。

イベント・ポーリングへの影響: DuplicateEventElimination を true に設定して、保証付きイベント・デリバリーを使用するコネクターは、この機能を使用しないコネクターとは少し異なる動作をします。重複イベント回避機能を提供するために、コネクター・フレームワークは JMS モニター・キューを使用してビジネス・オブジェクトを追跡します。JMS モニター・キューの名前は、MonitorQueue コネクター構成プロパティーから取得されます。

コネクター・フレームワークは、アプリケーション固有のコンポーネントから (pollForEvents() メソッドの gotApplEvent() の呼び出しを介して) ビジネス・オブジェクトを受信すると、(gotApplEvents() から受信した) 現在のビジネス・オブジェクトが重複イベントであるかどうかを判別する必要があります。このことを判別するために、コネクター・フレームワークは、JMS モニター・キューからビジネス・オブジェクトを取り出し、その ObjectEventId と現在のビジネス・オブジェクトの ObjectEventId を比較します。

- これら 2 つの ObjectEventId が同一の場合、現在のビジネス・オブジェクトは重複イベントを表しています。この場合、コネクター・フレームワークは、現行のビジネス・オブジェクトが表すイベントを無視し、このイベントを統合ブローカーに送信しません。
- ObjectEventId が同一でない 場合、ビジネス・オブジェクトは重複イベントを表していません。この場合、コネクター・フレームワークは、現在のビジネス・オ

ブジェクトを JMS モニター・キューにコピーし、これを JMS デリバリー・キューに配信します。これらの一連の処理は、すべて同一の JMS トランザクションの一部として実行されます。JMS デリバリー・キューの名前は、DeliveryQueue コネクター構成プロパティーから取得されます。gotApplEvent() メソッドの呼び出し後、コネクターの pollForEvents() メソッドへ制御が戻されます。

JMS 対応のコネクターで重複イベント回避機能サポートするには、コネクターの pollForEvents() メソッドに必ず次のステップを含めるようにしてください。

- 非 JMS イベント・ストアから取得したイベント・レコードからビジネス・オブジェクトを作成する場合、イベント・レコードの固有のイベント ID をビジネス・オブジェクトの ObjectEventId 属性として保管します。

  アプリケーションはこのイベント ID を生成し、イベント・ストアのイベント・レコードを一意に識別します。統合ブローカーへのイベント送信後で、しかもこのイベント・レコードの状況が変更可能となる前にコネクターに障害が発生した場合、このイベント・レコードは「進行中 (In-Progress)」状況のままイベント・ストアに残されます。コネクターが復旧した際に、「進行中」のイベントをリカバリーする必要があります。コネクターは、ポーリングを再開すると、イベント・ストアに残っているイベント・レコードのビジネス・オブジェクトを生成します。ただし、すでに送信済みのビジネス・オブジェクトと新規ビジネス・オブジェクトの両方が ObjectEventId として同じイベント・レコードを持っているため、コネクター・フレームワークは新規ビジネス・オブジェクトを重複オブジェクトと認識し、統合ブローカーに送信しない場合があります。

  Java コネクターで CWConnectorBusObj クラスの setDEEId() メソッドを次のように使用すると、イベント ID を ObjectEventId 属性に割り当てることができます。

  ```
  busObj.setDEEId(event_id);
  ```

- コネクターのリカバリー時には、コネクターが新規イベントのためのポーリングを開始する*前*に、「進行中」のイベントを処理するようにしてください。

  コネクターの開始時に、「進行中」のイベントが「ポーリング可能」状況に変更されない限り、ポーリング・メソッドは再処理のためにイベント・レコードを受信しません。

第 6 章 メッセージ・ロギング

この章ではメッセージ・ロギングについて説明します。メッセージ は、コネクターが外部接続ログに送信する情報のストリングです。システム管理者または開発者はこのログに送信されたメッセージを検討して、コネクターのランタイム状態に関する情報を準備することができます。コネクターがコネクター・ログに送信できるメッセージには、2 つのカテゴリーがあります。

- エラー・メッセージまたは情報メッセージ
- トレース・メッセージ

メッセージは、コネクター・コード内で生成することができ、メッセージ・ファイルから取得することもできます。この章を構成するセクションは次のとおりです。

- 『エラー・メッセージと情報メッセージ』
- 161 ページの『トレース・メッセージ』
- 165 ページの『メッセージ・ファイル』

エラー・メッセージと情報メッセージ

コネクターはその状態に関する情報をログ宛先に送信できます。ロギングに推奨される情報のタイプは次のとおりです。

- エラーおよび致命的エラー。コードからログ・ファイルへ。
- システム管理者の注意を喚起する警告。コードからログ・ファイルへ。
- 次のような情報メッセージ
  - コネクターの始動および終了に関するメッセージ
  - アプリケーションからの重要メッセージ

コネクターは情報メッセージまたはエラー・メッセージを送信できますが、このロギング・プロセスはエラー・*ロギング* と呼ばれます。

注: これらのメッセージは、コネクター用に定義されたトレース・メッセージから独立しています。

ログ宛先の指定

コネクターはそのログ・メッセージをログ宛先に送信します。*ログ* は、コネクターの実行状態の検討が必要な場合に表示できる外部宛先です。ログの宛先は、コネクター構成時に Connector Configurator の「トレース/ログ・ファイル」タブの「ロギング」フィールドを設定することにより定義されます。次のいずれかを設定します。

- 「ファイルに」: 外部ファイルの絶対パス名。このファイルは、コネクターのプロセス (およびそのコネクター・フレームワークとアプリケーション固有のコンポーネント) と同じマシン上に存在しなければなりません。
- 「コンソールに (STDOUT)」: コネクター始動スクリプトがコネクターを始動するときに生成されるコマンド・プロンプト・ウィンドウ。

デフォルトでは、コネクターのログの宛先はコンソールに設定されます。つまり、始動スクリプトのコマンド・プロンプト・ウィンドウがログの宛先として使用されることになります。このログの宛先は、コネクターに合わせて適宜設定します。

WebSphere InterChange Server

また、次のように LogAtInterchangeEnd コネクター構成プロパティーを設定して、メッセージを InterChange Server のログ宛先にも記録するかどうかを指示することができます。

- メッセージをローカルでのみ 記録: LogAtInterchangeEnd を false に設定。
- メッセージをローカルに記録し、かつ InterChange Server のログ宛先に送信 : LogAtInterchangeEnd を true に設定。

デフォルトでは、LogAtInterchangeEnd は false に設定されます。つまり、メッセージはローカルにのみ記録されます。InterChange Server に送信されたメッセージは、InterChange Server メッセージ用に指定された宛先に書き込まれます。

注: InterChange Server のログ宛先にログを記録すると、電子メール通知もオンになります。これにより、エラーまたは致命的エラーが発生すると、InterchangeSystem.cfg ファイル内で指定された MESSAGE_RECIPIENT パラメーターに対する電子メール・メッセージが生成されます。例えば、LogAtInterchangeEnd を true に設定した場合にコネクターからアプリケーションへの接続が失われると、指定されたメッセージ宛先に、電子メール・メッセージが送信されます。

これらのコネクター・プロパティーは Connector Configurator を使用して設定します。InterChange Server のメッセージ・ロギングの詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

ログ宛先へのメッセージの送信

表 53 に、コネクターがエラー・メッセージ、警告メッセージ、および情報メッセージをログ宛先に送信する方法を示します。

表 53. ログ宛先へのメッセージ送信のメソッド

コネクターのライブラリー・メソッド	説明
logMsg() および generateMsg()	入力として、テキスト・ストリング、またはメッセージ・ファイル内のメッセージから生成されたストリングをとります。オプションとして、メッセージがエラー・メッセージであるか、警告メッセージであるか、あるいは情報メッセージであるかを示すメッセージ・タイプ定数を入力とすることもできます。メッセージ・ファイル内のメッセージ・テキストから文字ストリングを生成するには、generateMsg() メソッドを使用します。

表 53. ログ宛先へのメッセージ送信のメソッド (続き)

コネクターのライブラリー・メソッド	説明
generateAndLogMsg()	logMsg() メソッドと generateMsg() メソッドの機能を単一の呼び出しに結合します。

メッセージの生成方法の詳細については、167 ページの『メッセージ・ストリングの生成』を参照してください。

Java コネクター・ライブラリーでは、logMsg()、generateMsg()、および generateAndLogMsg() メソッドは CWConnectorUtil クラスに定義されます。

generateMsg() メソッドと generateAndLogMsg() メソッドは、どちらも引き数としてメッセージ・タイプを必要とします。この引き数はメッセージの重大度を表します。詳細については、167 ページの『メッセージ・ストリングの生成』を参照してください。

トレース・メッセージ

トレースは、トラブルシューティングとデバッグのためのオプションの機能で、コネクター用にオンにすることができます。トレースがオンになっていると、システム管理者は IBM WebSphere Business Integration システムの操作中にイベントを追跡することができます。

WebSphere InterChange Server

InterChange Server が統合ブローカーの場合は、コネクター・コントローラーおよびその他の InterChange Server システムのコンポーネントでもトレースを使用できます。

アプリケーション固有のコンポーネントでトレースを行うと、コネクター開発者およびコネクター・コードを使用するその他のユーザーがコネクターの振る舞いをモニターできます。コネクター・フレームワークが特定のコネクター機能を呼び出した場合も、トレースによる追跡が可能です。コネクター・アプリケーション固有のコードにトレース・メッセージを用意すると、コネクター・フレームワーク用のトレース・メッセージが補強されます。

トレースの使用可能化

コネクターのトレースはデフォルトではオフです。Connector Configurator でコネクター構成プロパティー TraceLevel をゼロ以外の値に設定すると、コネクター用のトレースがオンになります。 TraceLevel を 1 から 5 の値に設定して、適切な詳細レベルを得ることができます。レベル 5 のトレースでは、それより低いすべてのレベルのトレース・メッセージが記録されます。

WebSphere InterChange Server

ヒント: コネクター・コントローラーまたは InterChange Server システムのその他のコンポーネントでトレースをオンにする方法については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

トレース宛先の識別

コネクターはそのトレース・メッセージをトレース宛先に送信します。この宛先は、コネクターの実行状態の検討が必要な場合に表示できる外部宛先です。トレースの宛先は、コネクター構成時に Connector Configurator の「トレース/ログ・ファイル」タブの「トレース」フィールドを設定することにより定義されます。次のいずれかを設定します。

- 「ファイルに」: 外部ファイルの絶対パス名。このファイルは、コネクターのプロセス (およびそのコネクター・フレームワークとアプリケーション固有のコンポーネント) と同じマシン上に存在しなければなりません。
- 「コンソールに (STDOUT)」: コネクター始動スクリプトがコネクターを始動するときに生成されるコマンド・プロンプト・ウィンドウ。

デフォルトでは、コネクターのトレースの宛先はコンソールに設定されます。つまり、始動スクリプトのコマンド・プロンプト・ウィンドウがトレースの宛先として使用されることになります。このトレースの宛先は、コネクターに合わせて適宜設定します。

ログ宛先へのトレース・メッセージの送信

表 54 に、コネクターがトレース・メッセージをトレース宛先に送信する方法を示します。

表 54. トレース宛先へのトレース・メッセージの送信用メソッド

コネクターのライブラリー・メソッド	説明
`traceWrite()` および `generateMsg()`	テキスト・ストリング、またはメッセージ・ファイル内のメッセージから生成されたストリングと、トレース・レベル定数を入力として、トレース・レベルを指示します。このメソッドは、指定されたトレース・レベル以上のトレース・メッセージをトレース宛先に書き込みます。メッセージ・ファイル内のメッセージ・テキストから文字ストリングを生成するには、`generateMsg()` メソッドを使用し、メッセージ・タイプを `XRD_TRACE` に設定します。
`generateAndTraceMsg()`	`traceWrite()` メソッドと `generateMsg()` メソッドの機能を単一の呼び出しに結合します。

`generateMsg()` メソッドについては、167 ページの『メッセージ・ストリングの生成』を参照してください。

注: メッセージ・ファイル内でトレース・メッセージをローカライズする必要はありません。トレース・メッセージをメッセージ・ファイルに含めるかどうかは、開発者が任意に決定することができます。詳細については、 63 ページの『ロケール依存の設計原則』を参照してください。

Java コネクター・ライブラリーでは、`traceWrite()`、`generateMsg()`、および `generateAndTraceMsg()` メソッドは `CWConnectorUtil` クラス定義されます。

`traceWrite()` および `generateAndTraceMsg()` は引き数としてトレース・レベルを必要とします。この引き数はトレース・メッセージに使用するトレース・レベルを指定します。実行時にトレースをオンにする際は、トレースを実行するトレース・レベルを指定してください。実行時トレース・レベルと同じかまたは低いトレース・レベルを持つコード内のトレース・メッセージが、すべてトレース宛先に送信されます。詳細については、163 ページの『トレース・メッセージの推奨内容』を参照してください。

トレース・メッセージに関連付けるトレース・レベルを指定するには、`TRACELEVEL`*n* の形式のトレース・レベル定数を使用します。*n* には 1 から 5 のトレース・レベルを指定できます。トレース・レベル定数は、`CWConnectorLogAndTrace` クラスで定義されます。

`generateMsg()` メソッドを使用するには、メッセージ・タイプを引き数として指定する必要があります。この引き数はメッセージの重大度を表します。トレース・メッセージには重大度レベルがないため、`XRD_TRACE` メッセージ・タイプ定数をそのまま使用します。メッセージ・タイプ定数は、`CWConnectorLogAndTrace` クラスに定義されます。

注: `generateAndTrace()` メソッドを使用する場合、メッセージ・タイプを引き数として*指定する必要はありません*。メソッドによって自動的に `XRD_TRACE` メッセージ・タイプ定数が取られます。

トレース・メッセージの推奨内容

コネクターが戻す情報の種類は、コネクター開発者がトレース・レベルごとに定義します。表 55 は、アプリケーション固有のコネクター・トレース・メッセージの推奨内容です。

表 55. アプリケーション固有のコネクター・トレース・メッセージの内容

レベル	内容
0	コネクターのバージョンを識別するトレース・メッセージ。このレベルでは、これ以外のトレースは実行されません。
1	以下のことを行うトレース・メッセージ: • 処理されるビジネス・オブジェクトごとに状況メッセージと識別 (キー) 情報を記録する。 • `pollForEvents()` メソッドが実行されるたびにそれを記録する。

表 55. アプリケーション固有のコネクター・トレース・メッセージの内容 (続き)

レベル	内容
2	以下のことを行うトレース・メッセージ: • コネクターが処理するオブジェクトごとにビジネス・オブジェクト・ハンドラーを識別する。 • `gotApplEvent()` または `executeCollaboration()` から InterChange Server へビジネス・オブジェクトがポストされるたびにそれを記録する。 • 要求ビジネス・オブジェクトが受信されるたびにそれを示す。
3	以下のことを行うトレース・メッセージ: • 処理される外部キーを識別する (該当する場合)。これらのメッセージは、コネクターがビジネス・オブジェクト内に外部キーを検出したとき、またはコネクターがビジネス・オブジェクト内に外部キーを設定したときに表示されます。 • ビジネス・オブジェクト処理に関連した処理。例としては、ビジネス・オブジェクト相互間の一致の検出、子ビジネス・オブジェクトの配列内でのビジネス・オブジェクトの検出があります。
4	以下のことを行うトレース・メッセージ: • アプリケーション固有の情報を識別する。このような情報の例としては、ビジネス・オブジェクトのアプリケーション固有の情報フィールドを処理するメソッドが戻す値があります。 • コネクターがある機能を開始または終了した時点を識別する。これらのメッセージは、コネクターのプロセス・フローのトレースに役立ちます。 • スレッド固有の処理を記録する。例えば、コネクターが複数のスレッドを作成する場合、新しいスレッドが作成されるたびにメッセージに記録する必要があります。
5	以下のことを行うトレース・メッセージ: • コネクターの初期化を示す。このメッセージの内容の例としては、InterChange Server から取得された各コネクター構成プロパティーの値があります。 • コネクターが実行中に作成した各スレッドの詳細を記録する。 • アプリケーション内で実行されたステートメントを表す。コネクター・ログ・ファイルには、ターゲット・アプリケーションで実行されたすべてのステートメント、および置換された変数の値 (該当する場合) が入ります。 • ビジネス・オブジェクト・ダンプを記録する。コネクターは、ビジネス・オブジェクトの処理を開始する前とビジネス・オブジェクトを処理した後に、そのオブジェクトのテキスト表現 (処理前にはコネクターが統合ブローカーから受信したオブジェクトを示すもの、処理後にはコネクターが統合ブローカーに戻したオブジェクトを示すもの) を出力する必要があります。

注: コネクターは、指定されたトレース・レベルおよびそれより低いレベルで、すべてのトレース・メッセージをデリバリーする必要があります。

コネクター・フレームワーク・トレース・メッセージの内容と詳細レベルについての情報は、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

メッセージ・ファイル

コネクターのエラー・ロギング・メソッドまたはトレース・メソッドへのメッセージ入力データは、テキスト・ストリングとして、またはメッセージ・ファイル への参照として指定することができます。メッセージ・ファイルはメッセージ番号とメッセージ・テキストが入ったテキスト・ファイルです。メッセージ・テキストには、コネクターからの実行時データの引き渡しのための定位置パラメーターを含めることができます。メッセージ・ファイルを用意するには、ファイルを作成し、必要なメッセージを定義します。

WebSphere InterChange Server

重要: メッセージを InterChange Server メッセージ・ファイル `InterchangeSystem.txt` に追加しない でください。このシステム・メッセージ・ファイルからの既存のメッセージのみ がアクセス対象となります。

このセクションでは、メッセージ・ファイルについて以下の内容を説明します。

- 『メッセージ・フォーマット』
- 166 ページの『メッセージ・ファイルの名前とロケーション』
- 167 ページの『メッセージ・ストリングの生成』

メッセージ・フォーマット

メッセージ・ファイル内のメッセージのフォーマットは次のとおりです。

message number message text[EXPL]*explanation text*

message number は、メッセージを一意的に識別する整数です。 message number は 1 行に収めなければなりません。*message text* は、複数の行にまたがることができます。各行の最後に復帰 (CR) を置きます。 *explanation text* は、メッセージが出される原因となった条件のさらに詳しい説明です。

メッセージ・テキストの一例を示します。コネクターは、コネクターのバージョンとコネクター・インフラストラクチャーのバージョンが違うと判断すると、次のメッセージを呼び出します。

```
20017
Connector Infrastructure のバージョンが一致しません。
```

メッセージには、実行時に、値をプログラムからの値で置換できるパラメーターを含めることができます。これらのパラメーターは定位置パラメーターであり、メッセージ・ファイル内では中括弧で囲んだ番号で示されます。例えば、次のメッセージには、アンサブスクライブされたイベントを記録する 2 つのパラメーターがあります。

```
20026
警告: アンサブスクライブされたイベント: オブジェクト名 {1}、動詞 {2}。
```

メッセージ・パラメーターの値を指定する方法については、169 ページの『パラメーター値の使用』を参照してください。

注: これ以外のメッセージ例については、InterChange Server メッセージ・ファイル InterchangeSystem.txt を参照してください。

メッセージ・ファイルの名前とロケーション

コネクターは、コネクター・メッセージを以下に示す 2 つのメッセージ・ファイルのうちの 1 つから取得できます。

- コネクター・メッセージ・ファイルは *Appname*Connector.txt という名前で、製品ディレクトリーの次のサブディレクトリーに保管されます。

  connectors¥messages

  例えば、IBM WebSphere Business Integration Adapter for Clarify のコネクター・メッセージ・ファイルには ClarifyConnector.txt という名前が付けられます。
- InterChange Server メッセージ・ファイルは InterchangeSystem.txt という名前で、製品ディレクトリー内に格納されています。

メッセージが生成されたら、これら 2 つのメッセージ・ファイルのどちらからメッセージを抽出するかを、メッセージ・ファイル定数を使用して指定することができます。メッセージを生成するメソッド (167 ページの表 56 を参照) のすべてに、使用するメッセージ・ファイルを指定するためのパラメーターが提供されています。詳細については、167 ページの『メッセージ番号の指定』を参照してください。

WebSphere InterChange Server

コネクター・メッセージ・ファイルが存在しない 場合は、InterChange Server メッセージ・ファイル InterchangeSystem.txt (製品ディレクトリーにある) がメッセージ・ファイルとして使用されます。

コネクター・メッセージ・ファイルには、アプリケーション固有コンポーネントが使用するテキスト・ストリングがすべて含まれている必要があります。これらのストリングには、ハードコーディングされたストリングの他に、ロギングの対象となるストリングも含まれています。

注: コネクター標準では、トレース・メッセージはコネクター・メッセージ・ファイルに含めない ようにすることを推奨しています。これは、通常、エンド・ユーザーがトレース・メッセージを参照することはないからです。

国際化対応コネクターについては、テキスト・ストリングをコネクター・メッセージ・ファイルに分離することが重要です。このメッセージ・ファイルは翻訳可能なため、そのメッセージは異なる言語で容易に使用できます。翻訳されたコネクター・メッセージ・ファイルの名前には、次のように、関連のロケール名を含める必要があります。

*Appname*Connector_*ll_TT*.txt

上の行で、*ll* はロケールの 2 文字の省略語 (規則により小文字) で、*TT* は地域の 2 文字の省略語 (規則により大文字) です。例えば、米国英語のメッセージを含む WBI Adapter for Clarify 用のバージョンのコネクター・メッセージ・ファイルは、次のような名前にする必要があります。

```
ClarifyConnector_en_US.txt
```

コネクター・フレームワークは実行時に、connectors¥messages サブディレクトリーからコネクター・フレームワーク・ロケールに適切なメッセージ・ファイルを探し出します。例えば、コネクター・フレームワークのロケールが米国英語の場合 (en_US)、コネクター・フレームワークは、*Appname*Connector_en_US.txt ファイルからメッセージを取り出します。

コネクターのテキスト・ストリングの国際化対応の方法については、62 ページの『国際化対応コネクター』を参照してください。

メッセージ・ストリングの生成

表 56 に示すメソッドは、メッセージ・ファイルから定義済みメッセージを取得し、テキストをフォーマット設定し、生成されたメッセージ・ストリングを戻します。

表 56. メッセージ・ストリングを生成するメソッド

メッセージ・メソッド	説明
generateMsg()	メッセージ・ファイルから、指定された重大度のメッセージを生成します。このメッセージを logMsg() メソッドまたは traceWrite() メソッドへの入力として使用できます。
generateAndLogMsg()	メッセージ・ファイルから、指定された重大度のメッセージを生成し、ログ宛先に送信します。
generateAndTraceMsg()	メッセージ・ファイルからトレース・メッセージを生成し、ログ宛先に送信します。

ヒント: トレースに generateMsg() を使用する前に、トレースが isTraceEnabled() メソッドにより使用可能になっていることを確認してください。トレースが使用可能になっていない 場合は、トレース・メッセージを生成する必要はありません。

Java コネクター・ライブラリーでは、generateMsg()、generateAndLogMsg()、および generateAndTraceMsg() メソッドは CWConnectorUtil クラスに定義されます。

表 56 のメッセージ生成メソッドには、次の情報が必要です。

- 『メッセージ番号の指定』
- 168 ページの『メッセージ・タイプの指定』
- 169 ページの『パラメーター値の使用』 (オプション)

メッセージ番号の指定

表 56 のメソッドは、引き数としてメッセージ番号が必要です。この引き数は、メッセージ・ファイルから取得するメッセージの番号を指定します。 165 ページの『メッセージ・フォーマット』で説明したように、メッセージ・ファイル内の各メッセ

ージには、固有の整数のメッセージ番号 (ID) が関連付けられていなければなりません。表 56 のメソッドは、指定されたメッセージ番号をメッセージ・ファイルから取得し、関連したメッセージ・テキストを抜き出します。

これらのメソッドが取得するメッセージ・ファイルを指示するには、表 57 に示すように、引き数として整数のメッセージ・ファイル定数を指定する必要があります。

表 57. メッセージ・ファイル定数

メッセージ・ファイル定数	説明
INFRASTRUCTURE_MESSAGE_FILE	InterChange Server メッセージ・ファイル (InterchangeSystem.txt) から、指定されたメッセージ番号を取得します。 **注:** このメッセージ・ファイル定数は、統合ブローカーが InterChange Server の場合にのみ 有効です。
CONNECTOR_MESSAGE_FILE	コネクター・メッセージ・ファイルから、指定されたメッセージ番号を取得します。

Java コネクター・ライブラリーでは、メッセージ・ファイル定数は CWConnectorLogAndTrace クラスに定義されます。

IBM WebSphere Business Integration システムは日付と時刻を生成し、次のメッセージを表示します。

```
[1999/05/28 11:54:15.990] [ConnectorAgent ConnectorName]
Error 1100: Failed to connect to application
```

注: コネクターがローカル・ログ・ファイルに記録する場合は、コネクター・インフラストラクチャーがタイム・スタンプを追加します。

WebSphere InterChange Server

コネクターが InterChange Server に記録する場合は、InterChange Server がタイム・スタンプを追加します。

メッセージ・タイプの指定

表 56 のメソッドには、引き数としてメッセージ・タイプも必要です。この引き数はメッセージの重大度を表します。表 58 は、有効なメッセージ・タイプおよび関連したメッセージ・タイプ定数のリストです。

表 58. メッセージ・タイプ

メッセージ・タイプ	重大度レベル	説明
XRD_FATAL	致命的エラー	プログラムの実行を停止するエラーを示します。
XRD_ERROR	エラー	調査が必要なエラーを示します。
XRD_WARNING	警告	問題ではあっても無視できる条件を示します。
XRD_INFO	情報	情報メッセージのみ。アクションは必要ありません。
XRD_TRACE	--	トレース・メッセージに使用されます。

メッセージに関連付けるメッセージ・タイプを指定するには、次のように表 58 のメッセージ・タイプ定数の 1 つをメッセージ生成メソッドの引き数として使用します。

- ログ・メッセージの場合は、メッセージ重大度を示すメッセージ・タイプ定数、すなわち XRD_FATAL、XRD_ERROR、XRD_WARNING、XRD_INFO を (重大度の高い順に) 使用します。
- トレース・メッセージの場合は、XRD_TRACE メッセージ・タイプ定数を使用します。

Java コネクター・ライブラリーの generateAndTraceMsg() メソッドを使用する場合、トレース・メッセージ用のメッセージ・タイプを*指定する必要はありません*。Java コネクター・ライブラリーでサポートされている推奨されていないバージョンの generateAndTraceMsg() を使用するとメッセージ・タイプの入力を要求されます。これに対して、このメソッドの、推奨されているバージョンを使用すれば、自動的に XRD_TRACE メッセージ・タイプが指定されるため、引き数としてメッセージ・タイプを*入力する必要はありません*。

メッセージ・タイプ定数は、CWConnectorLogAndTrace クラスに定義されます。

パラメーター値の使用

表 56 のメッセージ生成メソッドでは、メッセージ・テキスト・パラメーターに任意の数の値を指定することができます。パラメーター値の数は、メッセージ・テキストに定義されているパラメーターの数に一致しなければなりません。メッセージ内でパラメーターを定義する方法については、165 ページの『メッセージ・フォーマット』を参照してください。

パラメーター値を指定するには、次の引き数を含める必要があります。

- メッセージ・テキスト内のパラメーターの数を示す引き数カウント。その数を決定するには、メッセージ・ファイル内のメッセージを参照します。
- パラメーター値をコンマで区切ったリスト。各パラメーターは文字ストリングとして表されます。

コネクター・メッセージ・ファイルに次のような 1 つのパラメーターを持つ情報メッセージがあるとします。

```
2887
Initializing connector {1}
```

このメッセージには 1 つのパラメーターが含まれているので、メッセージ生成メソッドの 1 つの呼び出しで引き数カウントとして 1 を指定してから、コネクターの名前を文字ストリングとして指定する必要があります。下記のコードの断片では、1 つのパラメーターを含むメッセージのフォーマットを設定しそのメッセージをログに送信するために、generateAndLogMsg() を呼び出しています。

```
String val = CWConnectorUtil.getConfigProp("ConnectorName");
CWConnectorUtil.generateAndLogMsg(2887, CWConnectorLogAndTrace.XRD_INFO,
     CWConnectorUtil.CONNECTOR_MESSAGE_FILE, 1, val);
```

val のパラメーター値はメッセージ・ファイル内のメッセージに結合されます。val が MyConnector に設定されると、結果のメッセージは次のようになります。

```
[1999/05/28 11:54:15.990] [ConnectorAgent MyConnector]
       Info 2887: Initializing connector MyConnector
```

トレース・メッセージをコネクター・メッセージ・ファイルに入れることもできます。

第 7 章 Java コネクターの実装

この章では、Java コネクターにおけるアプリケーション固有のコンポーネントの実装方法についての情報を提供します。ここでは、本書でここまでに説明した一般的なタスクの言語固有の詳細について説明します。

この章を構成するセクションは次のとおりです。

Java コネクター基底クラスの拡張

Java コネクター・ライブラリーでのコネクター基底クラスの名前は `CWConnectorAgent` です。 `CWConnectorAgent` クラスは、始動、サブスクリプション検査、ビジネス・オブジェクト・サブスクリプション・デリバリー、およびシャットダウンのメソッドを提供します。独自のコネクターを実装するには、このコネクター基底クラスを拡張して、独自のコネクター・クラス を作成します。

注: コネクター基底クラスのメソッドの一般情報については、77 ページの『コネクター基底クラスの拡張』を参照してください。

Java コネクターのコネクター・クラスを派生させるには、次のステップを実行します。

1. `CWConnectorAgent` クラスを拡張するコネクター・クラスを作成します。このコネクター・クラスに次の名前を付けます。

   connectorName`Agent.java`

   ここで、*connectorName* はコネクターが通信するアプリケーションまたはテクノロジーを固有に識別する名前です。例えば、Baan アプリケーションのコネクターを作成するには、`BaanAgent` という名前のコネクタークラスを作成します。
2. コネクター・クラス・ファイル内で、コネクターを含むパッケージ名を定義します。コネクター・パッケージ名は次のような形式になります。

   `com.crossworlds.connectors.`*connectorName*

   ここで、*connectorName* は上記のステップ 1 で定義したのと同じ名前です。例えば、Baan コネクターのパッケージ名は、コネクター・クラス・ファイル内で次のように定義します。

   ```
   package com.crossworlds.connectors.Baan;
   ```
3. コネクター・クラス・ファイルが次のクラスをインポートすることを確認します。

```
com.crossworlds.cwconnectorapi.*;
com.crossworlds.cwconnectorapi.exceptions.*;
```

コネクター・クラス・コードを保持する複数のファイルを作成する場合は、これらのクラスをすべての コネクター・ファイルにインポートする必要があります。

4. コネクターのアプリケーション固有コンポーネントに対して、適切な基底クラス・メソッドを実装します。この基底クラス・メソッドの作成方法については、表 59 を参照してください。

表 59. CWConnectorAgent クラスの基底クラス・メソッドの拡張

CWConnectorAgent メソッド	説明	詳細情報
agentInit()	コネクターのアプリケーション固有のコンポーネントを初期化します。	173 ページの『コネクターの初期化』
getVersion()	コネクターのバージョンを取得します。	173 ページの『コネクター・バージョンの検査』
getConnectorBOHandlerForBO()	ビジネス・オブジェクトのビジネス・オブジェクト・ハンドラーを取得します。	176 ページの『Java ビジネス・オブジェクト・ハンドラーの取得』
getEventStore()	コネクターのイベント・ストア・オブジェクトを取得します。	205 ページの『CWConnectorEventStoreFactory インターフェース』
doVerbFor()	動詞操作を実行することにより、要求ビジネス・オブジェクトを処理します。	177 ページの『ビジネス・オブジェクト・ハンドラーの作成』
pollForEvents()	イベント・ストアをポーリングしてアプリケーション・イベントを取得し、そのイベントをコネクター・フレームワークに送信します。	203 ページの『イベント通知機構の実装』
terminate()	コネクター・シャットダウンのためのクリーンアップ操作を実行します。	233 ページの『コネクターのシャットダウン』

コネクターの実行の開始

コネクターの始動時に、コネクター・フレームワークは、関連するコネクター・クラスのインスタンスを生成してから、表 60 に示すコネクター・クラス・メソッドを呼び出します。

表 60. コネクターの実行の開始

初期化タスク	詳細情報
1. アプリケーションへの接続の確立などコネクターが必要とするすべての初期化を実行するために、コネクターを初期化します。	173 ページの『コネクターの初期化』
2. コネクターがサポートするビジネス・オブジェクトごとに、ビジネス・オブジェクト・ハンドラーを取得します。	176 ページの『Java ビジネス・オブジェクト・ハンドラーの取得』

コネクターの初期化

コネクター・フレームワークは、コネクターの初期化を開始するために、コネクター基底クラス CWConnectorAgent 内の初期化メソッド agentInit() を呼び出します。このメソッドは、コネクターのアプリケーション固有のコンポーネントの初期化ステップを実行します。

重要: コネクター・クラスの実装の一部として、コネクターの agentInit() メソッドを実装する必要があります。

73 ページの『コネクターの初期化』で説明したように、agentInit() 初期化メソッドには、次の主要なタスクが含まれます。

- 『接続の確立』
- 『コネクター・バージョンの検査』
- 174 ページの『進行中イベントのリカバリー』

このセクションでは、上記のトピックに加えて、175 ページの『Java 初期化メソッドの例』で Java agentInit() メソッドの例を示します。

重要: 初期化メソッドの実行中には、ビジネス・オブジェクト定義とコネクター・フレームワークのサブスクリプション・リストはまだ使用できません。

接続の確立

agentInit() 初期化メソッドの主要なタスクは、アプリケーションへの接続を確立することです。コネクターが接続を正常に確立した場合、メソッドの実行は正常終了します。コネクターが接続を確立できない 場合、初期化メソッドは、失敗の原因を示すために ConnectionFailureException 例外をスローする必要があります。コネクターは、アプリケーションにログインすることが必要な場合もあります。このログオン試行が失敗した場合、初期化メソッドは、失敗の原因を示すために LogonFailedException 例外をスローする必要があります。 181 ページの表 64 に示すステップは、これらの初期化例外のいずれかをスローする方法の概要を示しています。

注: 初期化メソッド内のステップの概要については、73 ページの『接続の確立』を参照してください。

コネクター・バージョンの検査

getVersion() メソッドは、コネクターのアプリケーション固有のコンポーネントのバージョンを戻します。

注: getVersion() の概要説明については、74 ページの『コネクター・バージョンの検査』を参照してください。

Java コネクター・ライブラリーでは、getVersion() メソッドは CWConnectorAgent クラス内で定義されます。このクラスは、Java マニフェスト・ファイルからバージョンを取得する getVersion() のデフォルト実装を提供します。getVersion() をオーバーライドして、異なる実装を提供できます。

例えば、次のコード・サンプルは、コネクターのバージョンを示すストリングを戻す getVersion() を実装します。

```
public String getVersion(){
 // get version from manifest file, or from string

  String version = "1.0.0";
  return version;
}
```

進行中イベントのリカバリー

Java コネクター・ライブラリーは、イベント・ストアを表すための CWConnectorEventStore クラスを提供します。イベント・ストア内の進行中イベント・レコードを回復するために、Java コネクター・ライブラリーは、表 61 に示すメソッドを提供します。

表 61. 進行中イベントのリカバリー・メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEventStore	recoverInProgressEvents()

recoverInProgressEvents() メソッドは、進行中イベントのリカバリー動作を実装します。ただし、CWConnectorEventStore クラスは、デフォルトではこのメソッドを実装しません。可能なリカバリー動作の 1 つは、74 ページの表 23 で説明されているように、InDoubtEvents コネクター構成プロパティーに基づいています。

注: 進行中イベントのリカバリー方法の一般的説明については、74 ページの『進行中イベントのリカバリー』を参照してください。

リカバリー・プロセスが失敗した場合、初期化メソッドは、失敗の原因を示すために InProgressEventRecoveryFailedException をスローする必要があります。 181 ページの表 64 に示すステップは、この初期化例外をスローする方法の概要を示しています。

図 56 に、進行中イベントを回復するために recoverInProgressEvents() を使用する agentInit() メソッドのコード・フラグメントを示します。

```
// instantiate event-store factory
evtFac=new MyEventStoreFactoryInstance();

// instantiate event store
Object evto=evtFac.getEventStore();
CWConnectorEventStore evts=(CWConnectorEventStore)evto;

// check for any leftover In-Progress events
String inDoubtEvents=CWConnectorUtil.getConfigProp(
   "InDoubtEvents");

// In case the InDoubtEvents property is not set, use
// FailOnStartup as default.
if (inDoubtEvents == null || inDoubtEvents.equals(""))
   inDoubtEvents="FailOnStartup";

// recover In-Progress events
if (evts.recoverInProgressEvents() == FAIL
      || inDoubtEvents.equals("FailOnStartup") ) {

      // log a fatal error

      // throw an exception to terminate agentInit()
      throw new InProcessEventRecoveryFailureException()
   }
}
```

図 56. 進行中イベントのリカバリー

図 56 で、MyEventStoreFactoryInstance クラスは、イベント・ストアへのアクセスを提供する getEventStore() メソッドを持つ CWConnectorEventStoreFactory クラスの拡張の例です。

Java 初期化メソッドの例

Java コネクターでは、agentInit() メソッドがコネクターのアプリケーション固有のコンポーネントを初期化します。このメソッドは、値を戻す代わりに特殊な例外をスローすることにより、共通初期化エラーを示します。図 57 に、コネクター・プロパティーを取得しアプリケーションへの接続を確立する単純な agentInit() メソッドを示します。

```
public agentInit()
   throws PropertyNotSetException, ConnectionFailureException,
   InProgressEventRecoveryFailedException, LogonFailedException
{

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
      "Entering Connector agentInit()");

   int status = CWConnectorConstant.SUCCEED;
   connectorProperties =
      CWConnectorUtil.getAllConnectorAgentProperties();

   ExampleConnection userConnect = new Connection();

   // get Connector Configuration Properties to establish Connection
   String connectString =
      (String)connectorProperties.get("ConnectString");
   String userName =
      (String)connectorProperties.get("ApplicationUserName");
   String userPassword =
      (String)connectorProperties.get("ApplicationPassword");

   if(connectString == null || connectString.equals("")
         || userName==null || userPassword==null )
      {
      throw new PropertyNotSetException();
      }

   // Use Configuration Values to log into Application
   try
   {
      boolean loginSuccessful = userConnect.login(connectString,
         userName, userPassword);

      if(loginSuccessful)
         CWConnectorUtil.generateAndLogMsg(30000,CWConnectorLogAndTrace.XRD_INFO,);
   }
   catch(ExampleAppException se)
   {
      CWConnectorUtil.generateAndLogMsg(30001,
            CWConnectorLogAndTrace.XRD_ERROR,0,1,path);
   }

}
```

図 57. Java Connector の初期化

注: 進行中イベントを回復する agentInit() コード・フラグメントについては、175 ページの図 56を参照してください。

Java ビジネス・オブジェクト・ハンドラーの取得

Java コネクターでのビジネス・オブジェクト・ハンドラー基底クラスは CWConnectorBOHandler です。サポートされるビジネス・オブジェクトのビジネス・オブジェクト・ハンドラーのインスタンスを取得するために、コネクター・フレームワークは、CWConnectorAgent クラスの一部として定義される getConnectorBOHandlerForBO() メソッドを呼び出します。

注: getConnectorBOHandlerForBO() メソッドの一般情報については、75 ページの『ビジネス・オブジェクト・ハンドラーの取得』を参照してください。ビジネ

ス・オブジェクト・ハンドラーの設計方法の一般情報については、91 ページの『ビジネス・オブジェクト・ハンドラーの設計』を参照してください。

CWConnectorAgent クラス内の getConnectorBOHandlerForBO() のデフォルト実装は、ConnectorBOHandler という名前のビジネス・オブジェクト・ハンドラー基底クラスのビジネス・オブジェクト・ハンドラーを戻します。拡張ビジネス・オブジェクト・ハンドラー基底クラスの名前を ConnectorBOHandler にする場合は、getConnectorBOHandlerForBO() メソッドをオーバーライドする必要はありません。しかし、拡張ビジネス・オブジェクト・ハンドラー基底クラスの名前を ConnectorBOHandler 以外の名前にする場合は、拡張ビジネス・オブジェクト・ハンドラー基底クラスのインスタンスを戻すために、getConnectorBOHandlerForBO() メソッドをオーバーライドする必要があります。

コネクター・フレームワークが getConnectorBOHandlerForBO() メソッドへの呼び出しによって取得するビジネス・オブジェクト・ハンドラーの数は、使用しているコネクターでのビジネス・オブジェクト処理の全体的な設計に応じて異なります。

- コネクターがメタデータ主導型である場合は、汎用メタデータ主導型ビジネス・オブジェクト・ハンドラーを使用するようにコネクターを設計できます。

  図 58 には、メタデータ主導型ビジネス・オブジェクト・ハンドラーを戻す getConnectorBOHandlerForBO() メソッドの実装が含まれています。このメソッドは、ExampleBOHandler クラスのコンストラクターを呼び出します。このコンストラクターは、コネクターがサポートするすべての ビジネス・オブジェクトを処理する単一のビジネス・オブジェクト・ハンドラー基底クラスのインスタンスを生成します。
- アプリケーション固有のビジネス・オブジェクトの一部またはすべてが特殊な処理を必要とする場合は、それらのオブジェクトに対して、複数のビジネス・オブジェクト・ハンドラーをセットアップする必要があります。

重要: getConnectorBOHandlerForBO() メソッドの実行中には、ビジネス・オブジェクト・クラス・メソッドはまだ使用できません。

図 58 は、ExampleConnectorBOHandler クラスのコンストラクターを呼び出します。このコンストラクターは、コネクターがサポートするすべての ビジネス・オブジェクトを処理する単一のビジネス・オブジェクト・ハンドラー基底クラスのインスタンスを生成します。

```
public CWConnectorBOHandler getConnectorBOHandlerForBO(String BOName){
   return new ExampleConnectorBOHandler();
}
```

図 58. 汎用ビジネス・オブジェクト・ハンドラーのための getConnectorBOHandlerForBO() メソッド

ビジネス・オブジェクト・ハンドラーの作成

ビジネス・オブジェクト・ハンドラーの作成には、次のステップが含まれます。

- 178 ページの『Java ビジネス・オブジェクト・ハンドラーの基底クラスの拡張』

- ビジネス・オブジェクト・ハンドラー取得メソッドの実装 — 詳細については、176 ページの『Java ビジネス・オブジェクト・ハンドラーの取得』を参照してください。
- 179 ページの『doVerbFor() メソッドの実装』
- 200 ページの『カスタム・ビジネス・オブジェクト・ハンドラーの作成』

注: 要求処理の概要については、27 ページの『要求処理』を参照してください。要求処理と doVerbFor() のインプリメントの説明については、 91 ページの『第 4 章 要求処理』を参照してください。

Java ビジネス・オブジェクト・ハンドラーの基底クラスの拡張

Java コネクター・ライブラリーでのビジネス・オブジェクト・ハンドラーの基底クラスの名前は CWConnectorBOHandler です。 CWConnectorBOHandler クラスは、ビジネス・オブジェクト・ハンドラーの定義およびアクセスのためのメソッドを提供します。独自のビジネス・オブジェクト・ハンドラーを実装するには、このビジネス・オブジェクト・ハンドラー基底クラスを拡張して、独自のビジネス・オブジェクト・ハンドラー・クラス を作成します。

注: ビジネス・オブジェクト・ハンドラー基底クラスのメソッドの一般情報については、94 ページの『ビジネス・オブジェクト・ハンドラー基底クラスの拡張』を参照してください。

Java コネクターのビジネス・オブジェクト・ハンドラー・クラスを派生するには、次のステップを実行します。

1. CWConnectorBOHandler クラスを拡張するクラスを作成します。このクラスに次の名前を付けます。

   *connectorName*BOHandler.java

   ここで、*connectorName* はコネクターが通信するアプリケーションまたはテクノロジーを固有に識別する名前です。例えば、Baan アプリケーションのビジネス・オブジェクト・ハンドラーを作成するには、BaanBOHandler という名前のビジネス・オブジェクト・ハンドラー・クラスを作成します。複数のビジネス・オブジェクト・ハンドラーを実装するコネクター設計の場合は、処理されるビジネス・オブジェクトの名前を、ビジネス・オブジェクト・ハンドラー・クラスの名前に組み込みます。
2. ビジネス・オブジェクト・ハンドラー・クラス・ファイル内で、コネクターを含むパッケージ名を定義します。コネクター・パッケージ名は次のような形式になります。

   com.crossworlds.connectors.*connectorName*

   ここで、*connectorName* は上記のステップ 1 で定義したのと同じ名前です。例えば、Baan コネクターのパッケージ名は、ビジネス・オブジェクト・ハンドラー・クラス・ファイル内で次のように定義します。

   ```
   package com.crossworlds.connectors.Baan;
   ```
3. ビジネス・オブジェクト・ハンドラー・クラス・ファイルで次のクラスをインポートします。

   ```
   com.crossworlds.cwconnectorapi.*;
   com.crossworlds.cwconnectorapi.exceptions.*;
   ```

ビジネス・オブジェクト・ハンドラーのコードを保持する複数のファイルを作成する場合は、これらのクラスをすべての ファイルにインポートする必要があります。

4. ビジネス・オブジェクト・ハンドラーの振る舞いを定義するために、doVerbFor() メソッドを実装します。このメソッドの実装方法については、『doVerbFor() メソッドの実装』を参照してください。

   注: CWConnectorBOHandler クラス内のその他のメソッドでは、それぞれの実装が規定されています。ビジネス・オブジェクト・ハンドラー・クラスに実装する必要があるメソッドは、doVerbFor() メソッドのみです。詳細については、 287 ページの『第 12 章 CWConnectorBOHandler クラス』を参照してください。

アプリケーションとその API によっては、コネクターのために複数のビジネス・オブジェクト・ハンドラーの実装が必要になることがあります。ビジネス・オブジェクト・ハンドラーをインプリメントするときのいくつかの考慮事項の説明については、91 ページの『ビジネス・オブジェクト・ハンドラーの設計』を参照してください。

doVerbFor() メソッドの実装

doVerbFor() メソッドは、ビジネス・オブジェクト・ハンドラーの機能を提供します。コネクター・フレームワークは、要求ビジネス・オブジェクトを受け取ると、該当するビジネス・オブジェクト・ハンドラーの doVerbFor() メソッドを呼び出して、ビジネス・オブジェクトの動詞のアクションを実行します。 Java コネクターでは、動詞の処理を定義する doVerbFor() メソッドは、CWConnectorBOHandler クラスに定義されています。

注: doVerbFor() メソッドの役割の一般的説明については、95 ページの『要求の処理』を参照してください。96 ページの図 27 に、このメソッドの基本ロジックを示します。

ただし、コネクター・フレームワークが実際に呼び出す doVerbFor() メソッドは、下位の Java コネクター・ライブラリーの BOHandlerBase クラスから CWConnectorBOHandler クラスが継承する、このメソッドの下位バージョンです。この下位バージョンの doVerbFor() は、ユーザーが実装する doVerbFor() メソッドを呼び出します。したがって、ユーザーのビジネス・オブジェクト・ハンドラー・クラス (CWConnectorBOHandler の拡張) の一部として doVerbFor() メソッドを実装する必要があります。

注: ビジネス・オブジェクト動詞が動詞のアプリケーション固有の情報内に CBOH タグを含んでいない 限り、下位の doVerbFor() メソッドは doVerbFor() メソッドを呼び出します。 CBOH タグが存在する場合、下位の doVerbFor() は、CBOH タグで名前が指定されたカスタム・ビジネス・オブジェクト・ハンドラーを呼び出します。詳細については、200 ページの『カスタム・ビジネス・オブジェクト・ハンドラーの作成』を参照してください。

ビジネス・オブジェクト・ハンドラーの役割は、次のタスクを実行することです。

1. コネクター・フレームワークからビジネス・オブジェクトを受信します。
2. アクティブ動詞に基づいて各ビジネス・オブジェクトを処理します。

3. アプリケーションに操作の要求を送信します。
4. コネクター・フレームワークに状況を戻します。

表 62 に、doVerbFor() メソッドが実行する一般的な動詞処理の基本ロジック内のステップの要約を示します。詳細情報の列にリストされている各セクションには、基本ロジック内の関連するステップについてのより詳細な説明があります。

表 62. doVerbFor() メソッドの基本ロジック

ビジネス・オブジェクト・ハンドラーのステップ	詳細情報
1. 要求ビジネス・オブジェクトからアクティブ動詞を取得します。	180 ページの『アクティブ動詞の取得』
2. コネクターがアプリケーションへの有効な接続を維持しているかどうかを検証します。	182 ページの『動詞処理の前の接続の検証』
3. 有効なアクティブ動詞の値で分岐します。	183 ページの『アクティブ動詞での分岐』
4. 特定のアクティブ動詞に対して、適切な要求処理を実行します。	
• 動詞固有のタスクを実行します。	185 ページの『動詞操作の実行』
• ビジネス・オブジェクトを処理します。	187 ページの『ビジネス・オブジェクトの処理』
5. コネクター・フレームワークに適切な状況を送信します。	194 ページの『動詞処理応答の送信』

表 62 に示す処理ステップのほかに、このセクションでは 196 ページの『処理に関する追加の問題』 に追加の処理情報を記載します。

注: Java コネクターはスレッド・セーフである必要があります。Java コネクターのコネクター・フレームワークは、doVerbFor() メソッドと pollForEvents() メソッドへの呼び出しのために、それぞれ異なるスレッドを使用します。

WebSphere InterChange Server

InterChange Server を使用するビジネス・インテグレーション・システムでコラボレーションがマルチスレッドとしてコード化されている場合、コネクター・フレームワークは、要求処理を表す複数のスレッドを使用して doVerbFor() への呼び出しを実行することがあります。

アクティブ動詞の取得

実行するアクションを判別する前に、doVerbFor() メソッドは、メソッドが引き数として受け取るビジネス・オブジェクトから動詞を取得する必要があります。この着信ビジネス・オブジェクトを *要求ビジネス・オブジェクト* と呼びます。このビジネス・オブジェクトに格納されている動詞は、*アクティブ動詞* です。アクティブ動詞は、ビジネス・オブジェクト定義がサポートする動詞の 1 つである必要があります。表 63 に、要求ビジネス・オブジェクトからアクティブ動詞を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 63. アクティブ動詞を取得するためのメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	getVerb()

要求ビジネス・オブジェクトからのアクティブ動詞の取得には、一般に次のステップが含まれます。

1. 要求ビジネス・オブジェクトが有効であることを検証します。

   コネクターは、getVerb() を呼び出す前に、着信要求ビジネス・オブジェクトが null ではない ことを検証します。着信ビジネス・オブジェクトは、doVerbFor() メソッドに CWConnectorBusObj オブジェクトとして渡されます。
2. getVerb() メソッドを使用して、アクティブ動詞を取得します。

   要求ビジネス・オブジェクトが有効であることが検証されたら、CWConnectorBusObj クラス内の getVerb() メソッドを使用して、このビジネス・オブジェクトからアクティブ動詞を取得できます。
3. アクティブ動詞が有効であることを検証します。

   コネクターは、アクティブ動詞を取得した後で、この動詞がヌルまたは空ではないことを検証します。

要求ビジネス・オブジェクトまたはアクティブ動詞のいずれかが 無効である場合、コネクターは、動詞処理を継続しません。その代わりに、コネクターは、表 64 で説明されているステップを実行します。

表 64. 動詞処理エラーの処理

エラー処理のステップ	使用するメソッドまたはコード
1. 動詞処理エラーの原因を示すために、ログ宛先にエラー・メッセージを記録します。	`CWConnectorUtil.generateAndLogMsg()`
2. 例外情報を保持するために例外詳細オブジェクトのインスタンスを生成します。	`CWConnectorExceptionObject excptnDtailObj = new CWConnectorExceptionObject();`
3. 例外詳細オブジェクト内の状況情報を設定します。	
• 動詞処理エラーの原因を示すためのメッセージを設定します。	`excptnDtailObj.setMsg()`
• コネクター・フレームワークが統合ブローカーへの応答に含める FAIL 結果状況に合わせて、状況を設定します。	`excptnDtailObj.setStatus()`
4. VerbProcessingFailureException 例外をスローします。doVerbFor() は、この例外を使用して、動詞処理エラーが発生したことをコネクター・フレームワークに通知します。この例外オブジェクトには、ステップ 2 で初期化した例外詳細オブジェクトが格納されます。 下位の doVerbFor() メソッドは、この例外オブジェクトをキャッチすると、例外詳細オブジェクトからのメッセージと状況を戻り状況記述子にコピーして、コネクター・フレームワークに戻します。その後で、この戻り状況記述子は、コネクター・フレームワークから統合ブローカーに戻されます。	`throw new VerbProcessingFailureException( excptnDtailObj);`

注: 例外オブジェクトと例外詳細オブジェクトについては、236 ページの『例外』を参照してください。

図 59 には、getVerb() メソッドを使用してアクティブ動詞を取得する doVerbFor() メソッドのコード・フラグメントが含まれています。このコードは、try および

catch ステートメントを使用して、要求ビジネス・オブジェクトとそのアクティブ動詞がヌルではないことを確認します。いずれかの条件がヌルである場合、コード・フラグメントは VerbProcessingFailedException 例外をスローし、コネクター・フレームワークがこの例外をキャッチします。

```
public int doVerbFor(CWConnectorBusObj theBusObj)
   throws VerbProcessingFailedException, ConnectionFailureException
{
   CWConnectorExceptionObject cwExcpObj =
      new CWConnectorExceptionObject();

   //make sure that the incoming business object is not null
   if (theBusObj == null) {
      CWConnectorUtil.logMsg(3456,
         CWConnectorLogAndTrace.XRD_ERROR);
      cwExcpObj.setMsg(
         "doVerbFor(): Invalid business object passed in");
      cwExcpObj.setStatus(CWConnectorConstant.FAIL);
      throw new VerbProcessingFailedException(cwExcpObj);
   }

   // obtain the active verb
   String busObjVerb = theBusObj.getVerb();

   // make sure the active verb is neither null nor empty
   if (busObjVerb == null || busObjVerb.equals("")){
      cwExcpObj.setMsgNumber(6548);
      cwExcpObj.setMsgType(CWConnectorLogAndTrace.XRD_ERROR);
      cwExcpObj.setMsg("doVerbFor: Invalid active verb");
      cwExcpObj.setStatus(CWConnectorConstant.FAIL);
      throw new VerbProcessingFailedException(cwExcpObj);
   }
   try {
      // perform verb processing here
      ...

   } catch (SampleException se) {
      throw new VerbProcessingFailedException(cwExcpObj);
}
```

図 59. アクティブ動詞の取得

動詞処理の前の接続の検証

コネクター・クラス内の agentInit() メソッドがアプリケーション固有のコンポーネントを初期化するときに、最も一般的なタスクの 1 つは、アプリケーションへの接続を確立することです。 doVerbFor() が動詞処理を実行するには、アプリケーションへのアクセスが必要です。したがって、doVerbFor() メソッドは、動詞の処理を開始する前に、コネクターがアプリケーションへの接続を維持しているかどうかを検証する必要があります。この検証を実行する方法は、アプリケーション固有です。詳細については、アプリケーションのドキュメンテーションを参照してください。

コネクターのアプリケーション固有のコンポーネントの設計時には、アプリケーションへの接続が切断されたときにはコンポーネントがシャットダウンするようにコーディングすることをお勧めします。接続が切断されている場合、コネクターは、

動詞処理を継続しません。その代わりに、コネクターは、次のステップを実行して、接続が切断されたことをコネクター・フレームワークに通知します。

1. エラーの原因を示すために、ログ宛先にエラー・メッセージを記録します。

   コネクターは、致命エラー・メッセージを記録します。LogAtInterchangeEnd コネクター構成プロパティーが True に設定されている場合は、電子メール通知が起動されます。

2. 例外詳細オブジェクトに次の情報を設定します。

• 接続障害の原因を示すメッセージ	CWConnectorExceptionObject.setMsg()
• コネクター・フレームワークが統合ブローカーへの応答に含める APPRESPONSETIMEOUT 結果状況の状況	CWConnectorExceptionObject.setStatus()

   この例外詳細オブジェクトは、doVerbFor() がスローする例外オブジェクトの一部です。これらのメソッドの詳細については、236 ページの『例外』を参照してください。

3. ConnectionFailureException 例外をスローします。doVerbFor() は、この例外を使用して、アプリケーションへの接続が失われたために動詞処理が行えないことをコネクター・フレームワークに通知します。この例外オブジェクトには、ステップ 2 で初期化した例外詳細オブジェクトが格納されます。

   下位の doVerbFor() メソッドは、この例外オブジェクトをキャッチすると、例外詳細オブジェクトからのメッセージと状況を戻り状況記述子にコピーして、コネクター・フレームワークに戻します。 ConnectionFailureException 例外詳細オブジェクトでこの状況を設定しないと、コネクター・フレームワークが状況を APPRESPONSETIMEOUT に設定します。コネクター・フレームワークは、この戻り状況記述子を統合ブローカーへの応答の一部として組み込みます。統合ブローカーは、戻り状況記述子を検査することによって、アプリケーションが応答していないことを確認できます。

コネクター・フレームワークは、戻り状況記述子を送信した後で、コネクターが実行されているプロセスを停止します。システム管理者は、アプリケーションに関する問題を修正してから、コネクターを再始動してイベントとビジネス・オブジェクト要求の処理を継続する必要があります。

アクティブ動詞での分岐

動詞処理の主要なタスクは、アクティブ動詞に関連する操作をアプリケーションが確実に実行できるようにすることです。アクティブ動詞で実行されるアクションは、doVerbFor() メソッドが基本メソッドまたはメタデータ主導型メソッドのどちらとして設計されているかによって異なります。

- 『基本動詞処理』
- 185 ページの『メタデータ主導型動詞処理』

基本動詞処理: メタデータ主導型ではない 動詞処理では、アクティブ動詞の値で分岐して、動詞固有の処理を実行します。doVerbFor() メソッドは、ビジネス・オブジェクトがサポートするすべての 動詞を処理する必要があります。

注: サポートされるビジネス・オブジェクトの動詞のリストは、CWConnectorBusObj クラスの getSupportedVerbs() メソッドを使用して取得することができます。

表 65 に、アクティブ動詞との比較のために Java コネクター・ライブラリーが提供する動詞定数を示します。

表 65. Java 動詞定数

動詞定数	アクティブ動詞
VERB_CREATE	Create
VERB_RETRIEVE	Retrieve
VERB_UPDATE	Update
VERB_DELETE	Delete
VERB_EXISTS	Exists
VERB_RETRIEVEBYCONTENT	RetrieveByContent

表 65 の動詞定数は、すべて CWConnectorConstant クラス内で定義されています。コネクターが上記以外の動詞を処理する場合は、拡張 CWConnectorBOHandler クラスの一部として独自の String 定数を定義することをお勧めします。

注: 動詞分岐ロジックの一部として、無効な動詞に対するテストを組み込んでください。要求ビジネス・オブジェクトのアクティブ動詞がビジネス・オブジェクト定義によってサポートされない 場合は、動詞処理でのエラーを示すために、ビジネス・オブジェクト・ハンドラーが適切な回復アクションを実行する必要があります。動詞処理エラーを処理するためのステップのリストについては、181 ページの表 64 を参照してください。

図 60 に、Create および Update 動詞のアクティブ動詞の値で分岐する doVerbFor() のコード・フラグメントを示します。ビジネス・オブジェクトがサポートする動詞ごとに、このコード内で 1 つの分岐を提供する必要があります。

```
// handle the Create verb
if(busObjVerb.equals(CWConnectorConstant.VERB_CREATE)){
   CWConnectorUtil.initAndValidateAttributes(theBusObj);
   status=doCreate(theBusObj);
   // where doCreate() inserts new row into Sample Apps database
   // using data from theBusObj
   }

// handle the Update verb
else if (objVerb.equals(CWConnectorConstant.VERB_UPDATE)){
   status=doUpdate(theBusObj);
   // where doUpdate() locates existing row and updates it with
   // information from theObj

// notify connector framework of invalid verb
} else {
   CWConnectorUtil.logMsg(3456, CWConnectorLogAndTrace.XRD_ERROR);
   cwExcpObj.setMsg("doVerbFor(): Invalid verb passed in");
   cwExcpObj.setStatus(CWConnectorConstant.FAIL);
   throw new VerbProcessing FailedException(cwExcpObj);
}
```

図 60. アクティブ動詞の値での分岐

図 60 のコード・フラグメントはモジュール化されています。つまり、サポートされる各動詞の実際の処理は、別個の*動詞メソッド* doCreate() と doUpdate() で指定されます。各動詞メソッドは、少なくとも次のガイドラインを満たす必要があります。

- CWConnectorBusObj パラメーターを定義します。これにより、動詞メソッドは要求ビジネス・オブジェクトを受け取ることができ、場合によってはこの更新されたビジネス・オブジェクトを呼び出し側メソッドに戻すことができます。
- すべての動詞固有の例外をスローして、検出されたすべての動詞処理エラーを doVerbFor() メソッドに通知します。
- 結果状況を戻します。これにより、doVerbFor() は結果状況をコネクター・フレームワークに戻すことができます。

このモジュラー構造により、doVerbFor() メソッドの読み易さと保守容易性が大幅に向上します。

メタデータ主導型動詞処理: メタデータ主導型動詞処理では、動詞のアプリケーション固有の情報にメタデータが含まれます。メタデータは、特定の動詞がアクティブなときの要求ビジネス・オブジェクトの処理命令を提供します。表 66 に、ビジネス・オブジェクトの動詞のアプリケーション固有の情報を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 66. 動詞のアプリケーション固有の情報の取得用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	getVerbAppText()

次に示す getVerbAppText() への呼び出しは、動詞のアプリケーション固有の情報を抽出します。

```
String verbAppInfo = theBusObj.getVerbAppText(busObjVerb);
```

動詞のアプリケーション固有の情報には、その特定の動詞で要求ビジネス・オブジェクトを処理するために呼び出されるメソッドの名前が含まれることがあります。この場合には、処理情報は動詞のアプリケーション固有の情報に含まれるため、doVerbFor() メソッドがアクティブ動詞の値で分岐する必要はありません。

注: 動詞のアプリケーション固有の情報を使用して、特定の動詞のアプリケーション・エンティティーを更新するために呼び出されるアプリケーションの API メソッドを指定することもできます。

動詞操作の実行

表 67 に、doVerbFor() メソッドが実装できる標準動詞と、各動詞操作での要求ビジネス・オブジェクトの処理方法の概要を示します。ビジネス・オブジェクトの処理方法の詳細については、187 ページの『ビジネス・オブジェクトの処理』を参照してください。

表 67. 動詞操作の実行

動詞	要求ビジネス・オブジェクトの使用	詳細情報
Create	• ビジネス・オブジェクト定義内のすべてのアプリケーション固有の情報を使用して、どのアプリケーション構造内でエンティティー (データベース表など) を作成するかを判別します。 • 各属性のすべてのアプリケーション固有の情報を使用して、どのアプリケーション副構造内で属性値 (データベース列など) を追加するかを判別します。 • 属性値を値として使用して、新規のアプリケーション・エンティティー内に保管します。 アプリケーションが新規のエンティティーのキー値を生成した場合は、新規のキー値を要求ビジネス・オブジェクト内に保管します。要求ビジネス・オブジェクトは、動詞処理応答の一部として組み込まれます。	99 ページの『Create 動詞の処理』
Retrieve	• ビジネス・オブジェクト定義内のすべてのアプリケーション固有の情報を使用して、どのアプリケーション構造 (データベース表など) からエンティティーを取得するかを判別します。 • 属性キー値 (複数可) を使用して、取得対象のアプリケーション・エンティティーを識別します。 アプリケーションが要求されたエンティティーを検出した場合は、その値を要求ビジネス・オブジェクトの属性内に保管します。要求ビジネス・オブジェクトは、動詞処理応答の一部として組み込まれます。	103 ページの『Retrieve 動詞の処理』
Update	• ビジネス・オブジェクト定義のすべてのアプリケーション固有の情報を使用して、どのアプリケーション構造 (データベース表など) 内でエンティティーを更新するかを判別します。 • 各属性のすべてのアプリケーション固有の情報を使用して、どのアプリケーション副構造で属性値 (データベース列など) を更新するかを判別します。 • 属性キー値 (複数可) を使用して、更新対象のアプリケーション・エンティティーを識別します。 • 属性値を値として使用して、既存のアプリケーション・エンティティーを更新します。 更新対象として指定されたエンティティーが存在しないときにはエンティティーを作成するようにアプリケーションが設計されている場合は、新規のエンティティーの値を要求ビジネス・オブジェクトの属性内に保管します。要求ビジネス・オブジェクトは、動詞処理応答の一部として組み込まれます。	111 ページの『Update 動詞の処理』
Delete	• ビジネス・オブジェクト定義内のすべてのアプリケーション固有の情報を使用して、どのアプリケーション構造 (データベース表など) からエンティティーを削除するかを判別します。 • 属性キー値 (複数可) を使用して、削除対象のアプリケーション・エンティティーを識別します。 InterChange Server が関係表のクリーンアップを必要に応じて実行できるように、要求ビジネス・オブジェクトは、動詞処理応答の一部として組み込まれます。	119 ページの『Delete 動詞の処理』

ビジネス・オブジェクトの処理

ほとんどの動詞操作には、要求ビジネス・オブジェクトからの情報の取得が含まれます。このセクションでは、要求ビジネス・オブジェクトを処理するために doVerbFor() メソッドが実行する必要のあるステップについての情報を提供します。

注: これらのステップは、コネクターがメタデータ主導型として設計されていることを想定しています。つまり、各属性に関連するアプリケーション内のロケーションを取得するために、ビジネス・オブジェクト定義および属性からアプリケーション固有の情報を抽出する方法を記述しています。コネクターがメタデータ主導型ではない 場合、通常はアプリケーション固有の情報を抽出するステップを実行する必要はありません。

表 68 に、メタデータを格納する要求ビジネス・オブジェクトを分解するための基本プログラム・ロジック内のステップの要約を示します。

表 68. メタデータを含む要求ビジネス・オブジェクトを処理するための基本ロジック

ステップ		詳細情報
1.	要求ビジネス・オブジェクトのビジネス・オブジェクト定義を取得します。	187 ページの『ビジネス・オブジェクト定義へのアクセス』
2.	ビジネス・オブジェクト定義内のアプリケーション固有の情報を取得して、アクセス対象のアプリケーション構造を取得します。	188 ページの『ビジネス・オブジェクトのアプリケーション固有の情報の抽出』
3.	属性情報を取得します。	189 ページの『属性へのアクセス』
4.	属性ごとにビジネス・オブジェクト定義内の属性のアプリケーション固有の情報を取得して、アクセス対象のアプリケーション副構造を取得します。	190 ページの『属性のアプリケーション固有の情報の抽出』
5.	適切な属性に対してだけ処理が実行されることを確認します。	191 ページの『属性を処理するかどうかの判別』
6.	値をアプリケーション・エンティティーに送信する必要のある各属性の値を取得します。	192 ページの『ビジネス・オブジェクトからの属性値の抽出』
7.	アプリケーションに通知して、適切な動詞操作を実行します。	193 ページの『アプリケーション操作の開始』
8.	動詞処理応答のために必要な要求ビジネス・オブジェクト内のすべての属性値を保管します。	193 ページの『ビジネス・オブジェクトへの属性値の保管』

ビジネス・オブジェクト定義へのアクセス: Java コネクターでは、doVerbFor() メソッドが要求ビジネス・オブジェクトを CWConnectorBusObj クラスのインスタンスとして受け取ります。doVerbFor() メソッドは、多くの場合、動詞処理を開始するためにビジネス・オブジェクト定義からの情報を必要とします。CWConnectorBusObj クラスは、ビジネス・オブジェクト、そのビジネス・オブジェクト定義、およびその属性へのアクセスを提供します。したがって、Java doVerbFor() メソッドは、ビジネス・オブジェクト定義のための別個のオブジェクト・インスタンスを生成する必要はありません。メソッドは、ビジネス・オブジェクト定義内の情報を、doVerbFor() に渡される CWConnectorBusObj オブジェクトから直接取得できます。

ビジネス・オブジェクト定義には、表 69 に示す情報が含まれます。CWConnectorBusObj メソッドの完全リストについては、 293 ページの『第 13 章 CWConnectorBusObj クラス』を参照してください。

表 69. ビジネス・オブジェクト定義から情報を取得するためのメソッド

ビジネス・オブジェクト定義情報	CWConnectorBusObj メソッド
ビジネス・オブジェクト定義の名前	getName()
動詞リスト — ビジネス・オブジェクトがサポートする動詞が含まれています。	isVerbSupported(), getSupportedVerbs()
属性のリスト — ビジネス・オブジェクト定義は、属性ごとに次の情報を定義します。	getAttrCount()
• 属性名	getAttrName()
• 属性のデータ型	getTypeName(), getTypeNum()
• 属性のリスト内の位置	getAttrIndex()
• その他のプロパティー	完全なリストについては、189 ページの表 71 を参照してください。
アプリケーション固有の情報	
• ビジネス・オブジェクト定義	getAppText(), getBusObjASIHashtable()
• 属性	getAppText(), getAttrASIHashtable()
• 動詞	getVerbAppText()

一般に、ビジネス・オブジェクト・ハンドラーは、ビジネス・オブジェクト定義を使用して、ビジネス・オブジェクトの属性についての情報やアプリケーション固有の情報をビジネス・オブジェクト定義、属性、または動詞から取得します。

ビジネス・オブジェクトのアプリケーション固有の情報の抽出: 通常の場合、メタデータ主導型コネクターのビジネス・オブジェクトは、アプリケーション構造についての情報を提供するアプリケーション固有の情報を保持するように設計されます。このようなコネクターでの一般的な動詞操作の最初のステップは、要求ビジネス・オブジェクトに関連するビジネス・オブジェクト定義からアプリケーション固有の情報を取得することです。表 70 に、ビジネス・オブジェクト定義からアプリケーション固有の情報を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 70. ビジネス・オブジェクトのアプリケーション固有の情報を取得するためのメソッド

Java コネクター・ライブラリー・クラス	メソッド
`CWConnectorBusObj`	getAppText() (引き数指定なし) getBusObjASIHashtable()

表 70 に示すように、コネクターは、次のいずれかのメソッドを使用して、ビジネス・オブジェクト定義のアプリケーション固有の情報を取得できます。

- `getAppText()` メソッドは、アプリケーション固有の情報を Java `String` として戻します。このメソッドは、ビジネス・オブジェクト・レベルのアプリケーション固有の情報内にある、指定された名前と値のペアの値を取得することもできます。

  注: `getAppText()` メソッドのメソッド名では、現在では使用すべきでない用語が使用されています。このメソッド名は、「アプリケーション固有のテキスト」という用語に基づいています。「アプリケーション固有のテキスト」に相当する現在の用語は、「アプリケーション固有の情報」です。
- `getBusObjASIHashtable()` メソッドは、名前と値のペアの Java `Hashtable` として、アプリケーション固有の情報を戻します。

表ベースのアプリケーションでは、多くの場合、アプリケーション固有の情報がアプリケーション構造についての情報を動詞操作に提供するように、ビジネス・オブジェクトが設計されます (詳細については、124 ページの表 43 を参照)。ビジネス・オブジェクト定義内のアプリケーション固有の情報は、ビジネス・オブジェクトに関連するデータベース表の名前を含むことができます。

属性へのアクセス: Java コネクターでは、CWConnectorBusObj クラスがビジネス・オブジェクト、そのビジネス・オブジェクト定義、および その属性へのアクセスを提供します。 doVerbFor() メソッドは、ビジネス・オブジェクト内の属性についての情報を必要とするときに、要求ビジネス・オブジェクトからこの情報を直接取得できます。したがって、Java doVerbFor() メソッドは、属性のための別個のオブジェクト・インスタンスを生成する必要はありません。

コネクターは、CWConnectorBusObj クラス内の属性メソッド (表 71 を参照) を使用して、カーディナリティーや最大長など属性についての情報を取得できます。属性プロパティーにアクセスするメソッドは、次の 2 つの方法で属性へのアクセスを提供します。

- 属性名 — 属性の名前プロパティーで属性を識別して、属性オブジェクトを取得できます。
- 整数索引 — 次のいずれかの方法で属性索引 (序数位置) を取得できます。
  - getAttrCount() を使用してビジネス・オブジェクト定義内のすべての属性のカウントを取得し、属性を 1 つずつループ処理することにより各索引値を表 71 に示す属性アクセス・メソッドの 1 つに渡します。
  - 特定の属性の索引を取得します。属性名を、getAttrIndex() に対して指定することにより、属性の索引を取得できます。

  注: getAttrCount() および getAttrIndex() メソッドは、CWConnectorBusObj クラス内で定義されます。

表 71 に、属性についての情報を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。属性情報にアクセスするメソッドの完全なリストについては、 293 ページの『第 13 章 CWConnectorBusObj クラス』を参照してください。

表 71. 属性情報の取得用メソッド

属性プロパティー	CWConnectorBusObj メソッド
Name	getAttrName(), hasName()
Type	getTypeNum(), getTypeName(), hasType(), isObjectType(),isType()
Key	isKeyAttr()
Foreign key	isForeignKeyAttr()
Max Length	getMaxLength()
Required	isRequiredAttr()
Cardinality	getCardinality(), hasCardinality(),isMultipleCard()
Default Value	getDefault(), getDefaultboolean(), getDefaultdouble(), getDefaultfloat(), getDefaultint(), getDefaultlong(),getDefaultString()
属性のアプリケーション固有の情報	getAppText()

属性のアプリケーション固有の情報の抽出: メタデータ主導型コネクターのビジネス・オブジェクトがアプリケーション構造についての情報を提供するアプリケーション固有の情報を保持するように設計されている場合、ビジネス・オブジェクト定義からアプリケーション固有の情報を抽出した後の次のステップは、要求ビジネス・オブジェクト内の各属性からアプリケーション固有の情報を抽出することです。表 72 に、各属性からアプリケーション固有の情報を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 72. 属性のアプリケーション固有の情報を取得するためのメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	getAttrCount() getAppText() (属性の位置または名前を引き数として指定) getAttrASIHashtable()

表 72 に示すように、コネクターは、次のいずれかのメソッドを使用して、属性のアプリケーション固有の情報を取得できます。

- `getAppText()` メソッドは、アプリケーション固有の情報を Java `String` として戻します。このメソッドは、属性のアプリケーション固有の情報内にある、指定された名前と値のペアの値を取得することもできます。

  注: `getAppText()` メソッドのメソッド名では、現在では使用すべきでない用語が使用されています。このメソッド名は、「アプリケーション固有のテキスト」という用語に基づいています。「アプリケーション固有のテキスト」に相当する現在の用語は、「アプリケーション固有の情報」です。
- `getAttrASIHashtable()` メソッドは、名前と値のペアの Java `Hashtable` として、アプリケーション固有の情報を戻します。

アプリケーション固有の情報が表ベースのアプリケーションの情報を提供するようにビジネス・オブジェクトが設計されている場合、属性のアプリケーション固有の情報は、この属性に関連するアプリケーション表の列の名前を含むことができます (詳細については、124 ページの表 43 を参照)。ビジネス・オブジェクト定義からアプリケーション固有の情報を抽出した後の次のステップは、要求ビジネス・オブジェクト内の属性に関連するのはアプリケーション表内のどの列かを判別することです。

動詞操作では、`getAppText()` を呼び出して属性の位置または名前を渡すことにより、アクセス対象のデータベース表内の列の名前を取得できます。動詞操作で各 属性のアプリケーション固有の情報を取得するには、ビジネス・オブジェクト定義内のすべての 属性をループ処理する必要があります。したがって、ビジネス・オブジェクト定義内の属性の総数を判別する必要があります。属性をループ処理するための最も一般的な構文は、ループ指標に対して次の制限を使用する `for` 文です。

- ループ指標は 0 に初期化されます。

  動詞操作で (キーを格納している) 最初の属性を処理する場合、ループ指標変数は 0 で開始されます。ただし、動詞が Create であり、アプリケーションがキーを生成する場合は、Create 動詞操作でキーを格納する属性が処理されないように する必要があります。この場合、ループ指標変数は 0 以外の値で開始されます。

- ループ指標は、ビジネス・オブジェクト定義内の属性の総数に到達するまで増分します。

  `getAttrCount()` メソッドは、ビジネス・オブジェクト内の属性の総数を戻します。ただし、この総数は `ObjectEventId` 属性を含みます。 `ObjectEventId` 属性は IBM WebSphere Business Integration システムによって使用され、アプリケーション表内には存在しない ため、動詞操作でこの属性を処理する必要はありません。したがって、ビジネス・オブジェクト属性をループ処理するときには、0 から (属性の総数 - 1) までのループを実行します。

  ```
  getAttrCount() - 1
  ```

- ループ指標は 1 ずつ増分します。

  指標のこの増分により、次の属性が取得されます。

Java コネクターは、`for` ループ内で、`getAppText()` メソッドを使用して各属性のアプリケーション固有の情報を取得できます。

```
for (i = 0; i < theBusObj.getAttrCount() - 1; i++) {
   colName = theBusObj.getAppText(i);

   // process the attribute associated with the column in
   // 'colName'
}
```

属性を処理するかどうかの判別: ここまでの動詞処理では、アプリケーション固有の情報を使用して、要求ビジネス・オブジェクトの各属性のアプリケーション・ロケーションを取得しました。動詞操作では、このロケーション情報を使用して、属性の処理を開始できます。

動詞操作でビジネス・オブジェクト属性をループ処理するときに、所定の属性だけが操作で処理されることを確認する必要が生じることがあります。表 73 に、属性を処理するかどうかを判別するために Java コネクター・ライブラリーが提供するいくつかのメソッドを示します。

表 73. 属性処理の判別用メソッド

属性テスト	CWConnectorBusObj メソッド
ある属性が単純属性であり、格納されているビジネス・オブジェクトを表す属性*ではない*。	isObjectType()
属性の値が Blank (長さ 0 のストリング) または Ignore (ヌル・ポインター) の特殊値*ではない*。	isIgnore(),isBlank()
属性がプレースホルダー属性*ではない*。プレースホルダー属性は、子ビジネス・オブジェクトを格納する属性を分離するために、ビジネス・オブジェクト定義内で使用されます。	getAppText()

動詞操作では、表 73 に示すメソッドを使用して、ある属性が操作で処理する対象の属性であることを判別できます。

- 属性は単純属性か複合属性か。

isObjectType() メソッドは、属性値が格納されているビジネス・オブジェクトを表していない ことを検査します。ビジネス・オブジェクトを格納している 属性の処理方法の詳細については、199 ページの『子ビジネス・オブジェクトへのアクセス』を参照してください。

- 属性はプレースホルダー属性または ObjectEventId 属性か。

  getAppText() メソッドを使用して、ビジネス・オブジェクト定義内の属性がアプリケーション固有の情報を保持しているかどうかを判別できます。これら 2 つの特殊な型の属性はアプリケーション・エンティティー内の列を表さないため、これらの属性のアプリケーション固有の情報をビジネス・オブジェクト定義に含める必要はありません。

- 属性は特殊値 Blank または Ignore の以外の値に設定されているか。

  動詞操作では、isIgnore() および isBlank() メソッドを使用して、属性の値を Ignore および Blank の値とそれぞれ比較できます。Ignore および Blank の値の詳細については、197 ページの『Blank 値および Ignore 値の処理』を参照してください。

ビジネス・オブジェクトからの属性値の抽出: 通常の場合、動詞操作で属性が処理可能であることが確認されたら、属性値を抽出する必要があります。

- Create または Update 動詞の動詞操作では、アプリケーションに送信するための属性値が必要です。アプリケーションでは、属性値を適切なアプリケーション・エンティティーに追加できます。 Update 動詞の動詞操作では、取得情報を保持するすべてのキー属性からの属性値も必要です。アプリケーションは、この取得情報を使用して、更新対象のエンティティーを探し出します。

  注: Create または Update 操作でコネクターに情報が戻される場合、動詞操作では、戻された情報を値として適切な属性内に格納する必要があります。詳細については、193 ページの『ビジネス・オブジェクトへの属性値の保管』を参照してください。

- Retrieve、RetrieveByContent、または Exists 動詞の動詞操作では、取得情報を保持するすべてのキー属性 (Retrieve または Exists) あるいはすべての非キー属性 (RetrieveByContent) からの属性値が必要です。アプリケーションは、この取得情報を使用して、エンティティーを取得します。

  注: Retrieve または RetrieveByContent の動詞操作では、取得されたデータに関連するすべての属性の属性値を設定する必要もあります。詳細については、193 ページの『ビジネス・オブジェクトへの属性値の保管』を参照してください。

- Delete 動詞の動詞操作では、取得情報を保持するすべてのキー属性からの属性値が必要です。アプリケーションは、この取得情報を使用して、削除対象のエンティティーを探し出します。

表 74 に、ビジネス・オブジェクトから属性値を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 74. 属性値の取得用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	getTypeName(),, getTypeNum(),, getbooleanValue(),, getBusObjValue(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue()

表 74 に示すように、CWConnectorBusObj クラスは、属性値を取得するための型固有のメソッドを提供します。これらのメソッドを使用すると、型と一致するように属性値をキャストする必要がなくなります。getTypeName() または getTypeNum() メソッドを使用して属性のデータ型を検査することにより、どの型固有のメソッドを使用するかを選択できます。

アプリケーション操作の開始: 動詞操作で要求ビジネス・オブジェクトから必要な情報が取得されたら、アプリケーション固有のコマンドを送信して、アプリケーションに適切な操作を実行させることができます。コマンドは、要求ビジネス・オブジェクトの動詞に対応している必要があります。表ベースのアプリケーションでは、このコマンドは SQL ステートメントや JDBC 呼び出しなどです。詳細については、アプリケーションのドキュメンテーションを参照してください。

重要: doVerbFor() メソッドは、アプリケーション操作が正常に完了したことを確認する必要があります。この操作が失敗した場合、doVerbFor() メソッドは、適切な結果状況 (FAIL など) をコネクター・フレームワークに戻す必要があります。詳細については、194 ページの『動詞処理応答の送信』を参照してください。

ビジネス・オブジェクトへの属性値の保管: アプリケーション操作が正常に完了したら、動詞操作では、アプリケーションから取得した新規の属性値を要求ビジネス・オブジェクト内に保管する必要が生じることがあります。

- Create 動詞の動詞操作では、アプリケーションが Create 操作の一部として新規のキー値を生成した場合に、そのキー値を保管する必要があります。
- Update 動詞の動詞操作では、生成されたすべてのキー値を含むすべての 属性値を保管する必要があります (更新対象として指定されたエンティティーが検出されないときに新規のエンティティーを作成するようにアプリケーションが設計されている場合)。
- Retrieve または RetrieveByContent の動詞操作では、取得されたすべての属性の属性値を保管する必要があります。

表 75 に、ビジネス・オブジェクト内に属性値を保管するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 75. 属性値の保管用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	setAttrValues(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

表 75 に示すように、CWConnectorBusObj クラスは、属性値を保管するために次の方法を提供します。

- setAttrValues() メソッドは、ビジネス・オブジェクト内のすべての 属性の値を保管します。このメソッドは、Java Vector オブジェクト内の属性値を受け入れます。
- 表 75 に示すその他のメソッドは、属性値を保管するための型固有のメソッドです。これらのメソッドを使用すると、型と一致するように属性値をキャストする必要がなくなります。getTypeName() または getTypeNum() メソッドを使用して属性のデータ型を検査することにより、どの型固有のメソッドを使用するかを選択できます。

動詞処理応答の送信

Java コネクターは、動詞処理応答をコネクター・フレームワークに送信する必要があります。コネクター・フレームワークは、その応答を統合ブローカーに送信します。この動詞処理応答には、次の情報が含まれます。

- doVerbFor() の整数戻りコード
- 戻り状況記述子内のメッセージ (情報メッセージ、警告メッセージ、またはエラー戻りメッセージがある場合)
- 応答ビジネス・オブジェクト

以下の各セクションでは、Java コネクターがそれぞれの応答情報を提供する方法についての追加情報を示します。コネクター応答の一般情報については、129 ページの『コネクター応答の指示』を参照してください。

結果状況のリターン: doVerbFor() メソッドは、整数結果状況を戻りコードとして提供します。表 76 に示すように、Java コネクター・ライブラリーは、doVerbFor() が戻す可能性の高い結果状況値の定数を提供します。

重要: doVerbFor() メソッドは、整数結果状況をコネクター・フレームワークに戻す*必要があります。*

表 76. Java doVerbFor() の結果状況値

doVerbFor() 内の条件	Java 結果状況
動詞操作は成功しました。	CWConnectorConstant.SUCCEED
動詞操作は失敗しました。	CWConnectorConstant.FAIL
アプリケーションが応答していません。	CWConnectorConstant. APPRESPONSETIMEOUT
ビジネス・オブジェクトの 1 つ以上の値が変更されました。	CWConnectorConstant.VALCHANGE
要求された操作が、同一キー値に対して複数のレコードを検出しました。	CWConnectorConstant.VALDUPES
コネクターが非キー値を使用して取得中に複数の一致レコードを検出しました。コネクターは、ビジネス・オブジェクト内で最初に一致したレコードのみを戻します。	CWConnectorConstant.MULTIPLE_HITS
コネクターは、非キー値による取得で、一致レコードを検出できませんでした。	CWConnectorConstant. RETRIEVEBYCONTENT_FAILED
要求されたビジネス・オブジェクト・エンティティーは、データベースに存在しません。	CWConnectorConstant. BO_DOES_NOT_EXIST

注: CWConnectorConstant クラスは、ほかのコネクター・メソッドが使用する上記以外の結果状況定数も提供します。結果状況定数の完全リストについては、349 ページの『結果状況定数』を参照してください。

doVerbFor() が戻す結果状況は、メソッドが処理するアクティブ動詞に応じて異なります。表 77 に、各動詞での可能な戻り値をリストした本書内の表を示します。

表 77. 各動詞の戻り値

動詞	詳細情報
Create	102 ページの表 35
Retrieve	109 ページの表 36
RetrieveByContent	111 ページの表 37
Update	118 ページの表 39
Delete	120 ページの表 41
Exists	122 ページの表 42

コネクター・フレームワークは、doVerbFor() が戻す結果状況を使用して、次に実行するアクションを決定します。

- 結果状況が APPRESPONSETIMEOUT である場合、コネクター・フレームワークはコネクターをシャットダウンします。詳細については、182 ページの『動詞処理の前の接続の検証』を参照してください。
- その他のすべての結果状況値の場合、コネクター・フレームワークはコネクターの実行を継続します。コネクター・フレームワークは、統合ブローカーへの応答に結果状況をコピーします。結果状況値の種類によっては、コネクター・フレームワークが応答ビジネス・オブジェクトを応答に含めることもあります。詳細については、196 ページの『要求ビジネス・オブジェクトの更新』を参照してください。

戻り状況記述子の取り込み: *戻り状況記述子* は、動詞処理の状態についての追加情報を保持する構造体です。コネクター・フレームワークが、ビジネス・オブジェクト・ハンドラーを起動する際に実際に呼び出すのは、下位の Java コネクター・ライブラリーの BOHandlerBase クラスから継承された、下位バージョンの doVerbFor() メソッドです。コネクター・フレームワークは、この下位の doVerbFor() メソッドに、空の戻り状況記述子オブジェクトを引き数として渡します。次に下位の doVerbFor() は、ユーザー実装の doVerbFor() メソッドを呼び出します (このメソッドは、CWConnectorBOHandler ビジネス・オブジェクト・ハンドラー・クラスの一部としてコネクター開発者が実装するバージョンです)。動詞の処理を実際に実行するのは、ユーザー実装の doVerbFor() です。

ユーザー実装のこの doVerbFor() メソッドが存在している場合、下位の doVerbFor() は、以下の方法で、その戻り状況記述子を動詞処理に関する情報で更新します。

- ユーザー実装の doVerbFor() メソッドが正常終了した (つまり、例外をスローしない) 場合、ユーザー実装の doVerbFor() メソッドからその戻り状況記述子の状況フィールド内に戻された結果状況が、下位の doVerbFor() によってコピーされます。

- ユーザー実装の doVerbFor() メソッドが*正常終了しない* (つまり、定義済みの例外の 1 つをスローする) 場合、下位の doVerbFor() によって例外がキャッチされ、例外詳細オブジェクトからの状況およびメッセージがその戻り状況記述子にコピーされます。

下位の doVerbFor() が終了すると、コネクター・フレームワークから、この更新済みの戻り状況記述子へのアクセスが可能になります。コネクター・フレームワークは、戻り状況記述子を応答に組み込み、その応答を統合ブローカーに送信します。

WebSphere InterChange Server

InterChange Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークは、コネクター・コントローラーに応答を戻します。コネクター・コントローラーは、その応答をコラボレーションにルーティングします。この応答には、下位の doVerbFor() メソッドによって取り込まれた戻り状況記述子が組み込まれます。コラボレーションは、この戻り状況記述子内の情報にアクセスし、そのサービス呼び出し要求の状況を取得することができます。

要求ビジネス・オブジェクトの更新: コネクター・フレームワークは、要求ビジネス・オブジェクトを引き数として doVerbFor() に渡します。doVerbFor() メソッドは、属性値を使用して、このビジネス・オブジェクトを更新できます。コネクター・フレームワークは、doVerbFor() が終了するときに、この更新されたビジネス・オブジェクトにアクセスできます。

コネクター・フレームワークは、結果状況を使用して、統合ブローカーへの応答の一部としてビジネス・オブジェクトを戻すかどうかを次のように判別します。

- コネクター・フレームワークが次の結果状況値の 1 つを受け取った場合、コネクター・フレームワークは、要求ビジネス・オブジェクトを応答の一部として組み込みます。
  - VALCHANGE
  - MULTIPLE_HITS

  doVerbFor() メソッドがこれらの結果状況値の 1 つを戻す場合は、メソッドが適切な応答情報を使用して要求ビジネス・オブジェクトを更新することを確認します。
- その他のすべての結果状況値の場合、コネクター・フレームワークは、要求ビジネス・オブジェクトを応答に組み込みません。

重要: doVerbFor() メソッドが戻す結果状況は、コネクター・フレームワークが統合ブローカーに送信する情報に影響を及ぼします。値が VALCHANGE または MULTIPLE_HITS である場合、コネクター・フレームワークは要求ビジネス・オブジェクトを戻します。戻された結果状況に応じて要求ビジネス・オブジェクトが適切に更新されることを確認する必要があります。

処理に関する追加の問題

このセクションでは、ビジネス・オブジェクトの処理に関連する次の問題についての情報を提供します。

- 『Blank 値および Ignore 値の処理』
- 199 ページの『子ビジネス・オブジェクトへのアクセス』

Blank 値および Ignore 値の処理: ビジネス・オブジェクト内の単純属性では、通常の属性値に加えて、表 78 に示す特殊値のいずれかを使用できます。

表 78. 単純属性の特殊な属性値

特殊な属性値	意味
Blank	「空の」長さ 0 のストリング値
Ignore	コネクターが無視する必要のある属性値

WebSphere InterChange Server

重要: InterChange Server を使用するビジネス・インテグレーション・システムの場合、サード・パーティーのマップでは、ストリング `CxIgnore` は Ignore 値を表し、ストリング `CxBlank` は Blank 値を表します。これらのストリングは、マップ内のみで 使用します。これらのストリングは、IBM WebSphere InterChange Server システム内の予約済みキーワードであるため、ビジネス・オブジェクト内の属性値としては保管しないで ください。

コネクターは、Java コネクター・ライブラリーのメソッドを呼び出して、ビジネス・オブジェクト属性が特殊値に設定されているかどうかを判別できます。

- Blank — Blank 値を持つ属性を処理するために、コネクターは、表 79 に示すメソッドを使用できます。

表 79. 属性に Blank 値が含まれているかどうかを判別するためのメソッド

CWConnectorBusObj メソッド	説明
`isBlank(`*`attributeName`*`)` `isBlank(`*`position`*`)`	指定された属性に Blank 値が格納されているかどうかを判別します。

属性に Blank 値が格納されている場合、`doVerbFor()` メソッドは、表 81 に示すように属性を処理します。

- Ignore — Ignore 値を持つ属性を処理するために、コネクターは、表 80に示すメソッドを使用できます。

表 80. 属性に Ignore 値が含まれているかどうかを判別するためのメソッド

CWConnectorBusObj メソッド	説明
`isIgnore(`*`attributeName`*`)` `isIgnore(`*`position`*`)`	指定された属性に Ignore 値が格納されているかどうかを判別します。

属性が Ignore 値に設定されている場合、コネクターは、表 82 に示すように属性を処理します。

表 81. Blank 値の処理アクション

動詞	Blank 値の処理アクション
Create	属性の適切な Blank 値を持つエンティティーを作成します。Blank 値は構成可能である場合と、アプリケーションに固有である場合とがあります。
Update	Blank 値に設定された属性値のエンティティー・フィールドを「空」に更新します。
Retrieve	属性がキーである場合またはコネクターが非キー値による取得を実行している場合は、この属性が長さ 0 のストリングであるエンティティーを取得します。
Delete	属性がキーである場合は、このフィールドが Blank 値に設定されているエンティティーを削除します。

表 82. Ignore 値の処理アクション

動詞	Ignore 値の処理アクション
Create	属性がキーでない場合は、アプリケーション内で属性の値を設定しません。アプリケーションがキーを生成するキー属性の場合は、キー属性が Ignore 値に設定されることがあります。この場合は、エンティティーを作成し、アプリケーションが生成したキーを取得し、統合ブローカーにキーを戻します。アプリケーションがキー値を生成しない場合は、すべてのキー属性が有効な値を持つと予想されます。
Update	属性がキーでない場合は、アプリケーション内で属性の値を設定しません。
Retrieve	Ignore に設定された属性に基づいて Retrieve 操作のマッチングを行いません。
Delete	Ignore に設定された属性に基づいて Delete 操作のマッチングを行いません。

コネクターが新規のビジネス・オブジェクトを作成するときに、すべての属性値は内部的に Ignore に設定されます。すべての未設定の属性値は Ignore として定義されたままなので、コネクターは、属性に適切な値を設定する必要があります。属性値を Ignore または Blank の特殊値に設定するには、表 83 に示す (`CWConnectorUtil` クラス内で定義された) メソッドを使用して特殊属性値を取得し、これらのメソッドの結果を属性に直接割り当てます。

表 83. 特殊な属性値の取得用メソッド

特殊な属性値	CWConnectorUtil メソッド
Blank 値	`getBlankValue()`
Ignore 値	`getIgnoreValue()`

表 83 に示すメソッドが必要な特殊属性値を取得したら、この属性値を属性値の「set」メソッド (193 ページの表 75 を参照) の 1 つに渡すことができます。次のコード・フラグメントを参照してください。

```
attrName = theBusObj.getAttrName(i);
theBusObj.setdoubleValue(attrName,
   CWConnectorUtil.getIgnoreValue());
```

子ビジネス・オブジェクトへのアクセス: 125 ページの『階層型ビジネス・オブジェクトの処理』で説明したように、Java コネクターは、表 75 に示す Java コネクター・ライブラリーのメソッドを使用して、子ビジネス・オブジェクトにアクセスします。

表 84. 子ビジネス・オブジェクトへのアクセス用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	isObjectType(),, isMultipleCard(),, getObjectCount(),, getBusObjValue()
CWConnectorAttrType	OBJECT 属性型定数

doVerbFor() メソッド内の動詞処理では、isObjectType() メソッドを使用して、属性に (属性型が OBJECT 属性型定数に設定されている) ビジネス・オブジェクトが格納されているかどうかを判別します。ビジネス・オブジェクトである属性が動詞操作で検出されたときに、メソッドは、isMultipleCard() を使用して属性のカーディナリティーを検査します。メソッドは、isMultipleCard() の結果に基づいて、次のアクションの 1 つを実行します。

- 属性が単一カーディナリティーを持つ場合、メソッドは、単一の子ビジネス・オブジェクトに対して要求された操作を実行できます。
- 属性が複数カーディナリティーを持つ場合、Java コネクターは、CWConnectorBusObj オブジェクトを通じてビジネス・オブジェクト配列の内容にアクセスできます。
  - 属性がビジネス・オブジェクトである場合、属性は、1 つのビジネス・オブジェクトを含む CWConnectorBusObj オブジェクトを格納します。
  - 属性がビジネス・オブジェクト配列である場合、属性は、すべてのビジネス・オブジェクトを配列内に含む CWConnectorBusObj オブジェクトを格納します。

Java 動詞メソッドは、CWConnectorBusObj.getObjectCount() を呼び出すことにより個々のビジネス・オブジェクトにアクセスして、配列内の子ビジネス・オブジェクトの数を取得できます。動詞メソッドは、ビジネス・オブジェクト配列を反復処理しながら、CWConnectorBusObj.getBusObjValue(*index*) メソッド (ここで、*index* は配列エレメントの索引) を使用して、ビジネス・オブジェクト配列内の個々の子オブジェクトを取得できます。このメソッドは、子ビジネス・オブジェクトを格納する CWConnectorBusObj を戻します。図 61 に、子ビジネス・オブジェクトにアクセスするための Java コードを示します。

```
// For all attributes in the business object
for (int i=0; i<theBusObj.getAttrCount()-1; i++ ){
   if ( theBusObj.isObjectType(i) ){
      // cardinality N
      if(theBusObj.isMultipleCard(i)){
         for (int i=0; i < theBusObj.getObjectCount(); i++) {
            CWConnectorBusinessObject childBusObj =
               theBusObj.getBusObjValue(i);
            status = do<i>Verb</i>Method(childBusObj);
         } // end for i to getObjectCount()
      } else {
         // Cardinality 1 child
            CWConnectorBusObj childBusObj = null;
         childBusObj = theBusObj.getBusObjValue(i);
         status = do<i>Verb</i>Method(childBusObj);
      } // end else 1 cardinality
   } // end isObjectType()
} // end for i to getAttCount()-1
```

図 61. Java コネクターでの子ビジネス・オブジェクトへのアクセス

カスタム・ビジネス・オブジェクト・ハンドラーの作成

コネクター・フレームワークは、特定のビジネス・オブジェクトがサポートするすべての 動詞について、(ビジネス・オブジェクト・ハンドラーを実装している) CWConnectorBOHandler クラス内の doVerbFor() メソッドを呼び出します。したがって、ビジネス・オブジェクト内の動詞はすべて、1 つの標準的な方法で処理されます (ただし、これらの動詞はアプリケーション内で別のアクションを開始することもできます)。ただし、特定の動詞について別の処理が必要となるビジネス・オブジェクトをコネクターがサポートする場合は、カスタム・ビジネス・オブジェクト・ハンドラー を作成して、ビジネス・オブジェクトのその動詞を処理できます。

カスタム・ビジネス・オブジェクト・ハンドラーを作成するには、次のステップを実行する必要があります。

- 『カスタム・ビジネス・オブジェクト・ハンドラー用のクラスの作成』
- 201 ページの『doVerbForCustom() メソッドの実装』
- 202 ページの『動詞のアプリケーション固有の情報の追加』

カスタム・ビジネス・オブジェクト・ハンドラー用のクラスの作成

カスタム・ビジネス・オブジェクト・ハンドラーを作成するには、CWCustomBOHandler インターフェースを実装したクラスを作成する必要があります。 CWCustomBOHandler インターフェースは、カスタム・ビジネス・オブジェクト・ハンドラーを定義するために実装する必要のある doVerbForCustom() メソッドを提供します。Java コネクターのカスタム・ビジネス・オブジェクト・ハンドラー・クラスを作成するには、次のステップを実行します。

1. CWCustomBOHandler インターフェースを実装したクラスを作成します。このクラスには、次の名前を付けることを推奨します。

   *connectorName*CustomBOHandler*verbName*.java

   ここで、*connectorName* はコネクターが通信するアプリケーションまたはテクノロジーを固有に識別する名前で、*verbName* はこのカスタム・ビジネス・オブジ

ェクト・ハンドラーが処理する 1 つまたは複数の動詞を識別する名前です。例えば、Baan アプリケーション内の Retrieve 動詞のカスタム・ビジネス・オブジェクト・ハンドラーを作成するには、BaanCustomBOHandlerRetrieve という名前のカスタム・ビジネス・オブジェクト・ハンドラー・クラスを作成します。

2. カスタム・ビジネス・オブジェクト・ハンドラー・クラス・ファイル内で、コネクターを含むパッケージ名を定義します。コネクター・パッケージ名は次のような形式になります。

   ```
   com.crossworlds.connectors.connectorName
   ```

   ここで、*connectorName* は上記のステップ 1 で定義したのと同じ名前です。例えば、Baan コネクターのパッケージ名は、カスタム・ビジネス・オブジェクト・ハンドラー・クラス・ファイル内で次のように定義します。

   ```
   package com.crossworlds.connectors.Baan;
   ```

3. カスタム・ビジネス・オブジェクト・ハンドラー・クラス・ファイルで次のクラスをインポートします。

   ```
   com.crossworlds.cwconnectorapi.*;
   com.crossworlds.cwconnectorapi.exceptions.*;
   ```

   ビジネス・オブジェクト・ハンドラーのコードを保持する複数のファイルを作成する場合は、これらのクラスをすべての ファイルにインポートする必要があります。

4. ビジネス・オブジェクト・ハンドラーの振る舞いを定義するために、doVerbForCustom() メソッドを実装します。このメソッドの実装方法については、『doVerbForCustom() メソッドの実装』を参照してください。

doVerbForCustom() メソッドの実装

doVerbForCustom() メソッドは、カスタム・ビジネス・オブジェクト・ハンドラーの機能を提供します。 179 ページの『doVerbFor() メソッドの実装』で説明したように、コネクター・フレームワークは、要求ビジネス・オブジェクトを受け取ると、該当するビジネス・オブジェクト・ハンドラーの下位の doVerbFor() メソッド (BOHandlerBase クラスで定義) を呼び出します。この下位の doVerbFor() メソッドは、呼び出すビジネス・オブジェクト・ハンドラーを次の方法で決定します。

- ビジネス・オブジェクトの動詞がそのアプリケーション固有の情報内に CBOH タグを持っている場合は、doVerbForCustom() メソッドを呼び出します。

  CBOH タグは、CWCustomBOHandlerInterface インターフェースとその doVerbForCustom() メソッドを実装するカスタム・ビジネス・オブジェクト・ハンドラー・クラスの名前をパッケージ名も含めて完全に指定します。このクラス名については、 202 ページの『動詞のアプリケーション固有の情報の追加』を参照してください。

  CBOH タグが存在する場合、下位の doVerbFor() メソッドは、このタグが指定するクラスの新しいインスタンスを作成しようと試みます。このインスタンス化が成功すると、下位の doVerbFor() は、このクラス内の doVerbForCustom() メソッドを呼び出します。

- それ以外の場合は doVerbFor() メソッドを呼び出します (このメソッドは、コネクターの開発者がビジネス・オブジェクト・ハンドラーの CWConnectorBOHandler

クラスの一部として実装する必要があります)。詳細については、 179 ページの『doVerbFor() メソッドの実装』 を参照してください。

doVerbForCustom() メソッドの実装では、そのクラスが指定された動詞に対して動詞処理を実行する必要があります。 doVerbFor() メソッドが通常行う動詞処理については、 179 ページの『doVerbFor() メソッドの実装』を参照してください。ただし、ビジネス・オブジェクトの動詞の特別な処理要件を満たすためには、doVerbForCustom() の振る舞いをカスタマイズする必要があります。

注: doVerbFor() メソッドとは異なり、doVerbForCustom() メソッドは、コネクター・フレームワークから直接呼び出されません。コネクター・フレームワークは、下位の doVerbFor() の呼び出しを通じて doVerbForCustom() を呼び出します。したがって、CWConnectorBOHandler クラス内のメソッド呼び出しを doVerbForCustom() に含めることはできません。

下位の doVerbFor() メソッドは、doVerbForCustom() からの戻り値と例外を次のような方法で処理します。

- doVerbForCustom() が正常に完了した場合は、状況をコネクター・フレームワークに戻します (doVerbFor() メソッドの場合と同様に)。
- カスタム・ビジネス・オブジェクト・ハンドラーのインスタンス化で問題がある場合は、原因を説明するエラー・メッセージとこの状況を戻り状況記述子に設定し、コネクター・フレームワークに FAIL 結果状況を戻します。
- doVerbForCustom() が VerbProcessingFailedException 例外をスローする場合は、例外オブジェクト内の状況セットを戻り状況記述子にコピーし、この例外状況をコネクター・フレームワークに戻します。
- doVerbForCustom() が ConnectionFailureException 例外をスローする場合は、例外オブジェクトがその状況セットを持っているかどうかを判別します。
  - 持っている場合は、例外状況を戻り状況記述子にコピーし、この状況をコネクター・フレームワークに戻します。
  - 持っていない場合は、APPRESPONSETIMEOUT 結果状況を戻り状況記述子にコピーし、APPRESPONSETIMEOUT をコネクター・フレームワークに戻します。

動詞のアプリケーション固有の情報の追加

特定のビジネス・オブジェクトのカスタム・ビジネス・オブジェクト・ハンドラーをコネクター・フレームワークで呼び出す場合、このビジネス・オブジェクトの動詞は、動詞のアプリケーション固有の情報内に CBOH タグを含んでいる必要があります。 CBOH タグの形式は、次のとおりです。

CBOH=*connectorPackageName.CustomBOHandlerClassName*

この形式において、*connectorPackageName* は次のとおりです。

com.crossworlds.connectors.*connectorName*

connectorName はコネクターの名前です。*CustomBOHandlerClassName* は、CWCustomBOHandlerInterface インターフェースを実装しているクラスの名前です。

例えば、次の CBOH タグは、BaanCustomBOHandlerRetrieve という名前のクラスを指定しています。

CBOH=com.crossworlds.connectors.Baan.BaanCustomBOHandlerRetrieve

イベント通知機構の実装

表 85 に、イベント通知機構の開発のために Java コネクター・ライブラリーが提供するサポートを示します。

表 85. イベント通知機構のサポート

Java コネクター・ライブラリー・サポート	詳細情報
イベント・ストアへのアクセスをカプセル化する次のクラス: • CWConnectorEvent • CWConnectorEventStatusConstants • CWConnectorEventStore • CWConnectorEventStoreFactory	203 ページの『イベント・ストアへのアクセスの取得』
イベント・ストアを指定された頻度でポーリングするポーリング・メソッド pollForEvents()	208 ページの『pollForEvents() メソッドの実装』

注: イベント通知の概要については、24 ページの『イベント通知』を参照してください。イベント通知機構と pollForEvents() のインプリメントについては、131 ページの『第 5 章 イベント通知』を参照してください。

イベント・ストアへのアクセスの取得

アプリケーション内で発生した情報をコネクターが処理することが予想される場合、コネクターは、アプリケーションのイベント・ストアへのアクセスを取得する必要があります。表 86 に、Java コネクターの内部からのイベント・ストアへのアクセスの取得を支援するために Java コネクター・ライブラリーが提供するサポートを示します。

表 86. イベント・ストアへのアクセスを定義するためのサポート

	Java コネクター・ライブラリー・クラス	説明
イベント・ストア	CWConnectorEventStoreFactory	イベント・ストア・オブジェクトを作成する単一のメソッドを提供します。
	CWConnectorEventStore	イベント・ストアを表します。
イベント	CWConnectorEvent	Java コネクター内のイベント・レコードへのアクセスを提供するイベント・オブジェクトを表します。

イベント・ストアの定義

表 86 に示すように、Java コネクター・ライブラリーは、イベント・ストアを定義する次のクラスを提供します。

- 『CWConnectorEventStore クラス』
- 205 ページの『CWConnectorEventStoreFactory インターフェース』

CWConnectorEventStore クラス: CWConnectorEventStore クラスは、イベント・ストアを定義します。表 87 に示すように、このクラスは、イベントの取得、処理、およびアーカイブの機構を標準化するための追加のレイヤーを提供します。

表 87. CWConnectorEventStore クラスのメソッド

イベント・ストア・タスク	CWConnectorEventStore メソッド	実装状況
イベント取得	fetchEvents()	実装が必要
	getBO()	基底クラス内で提供された実装 — ただし、コネクターで RetrieveByContent 動詞がサポートされない場合は、この実装をオーバーライドする必要があります。
	getNextEvent()	基底クラス内で提供された実装
イベント処理	recoverInProgressEvents()	実装が必要
	resubmitArchivedEvents()	実装が必要
	setEventStatus()	実装が必要
	setEventsToProcess()	基底クラス内で提供された実装
	updateEventStatus()	基底クラス内で提供された実装
アーカイブ	archiveEvent()	コネクターがアーカイブをサポートする場合、実装が必要
	deleteEvent()	実装が必要
エラー処理	getTerminate(),, setTerminate()	基底クラス内で提供された実装
リソースのクリーンアップ	cleanupResources()	イベント・ストア・クラスの場合には必要ありませんが、イベント・ストアへのアクセスに使用されたリソースを解放する必要がある場合には、実装しなければなりません。

イベント・ストアを定義するには、次のステップを実行します。

1. CWConnectorEventStore クラスを拡張して、コネクターがアクセスするイベント・ストアを識別できるようにその新しいクラスに名前を付けます。
2. イベント・ストアが必要とする可能性のあるすべての追加のデータ・メンバーを定義します。

   CWConnectorEventStore クラスは、単一のデータ・メンバー (eventsToProcess という名前のイベント・ベクトル配列) を格納します。イベント・ストアから取得されたイベントは、この Java Vector オブジェクト内に保管されます。アプリケーションのイベントおよびアーカイブ・ストアにアクセスするために必要なその他のすべての情報は、拡張した CWConnectorEventStore クラス内のデータ・メンバーとして宣言します。この情報は、イベント・ストアおよびアーカイブ・ストアのロケーションを含みます。例えば、次のようになります。

   - 表ベースのアプリケーション内では、この情報はイベント表名、アーカイブ表名、およびすべてのデータベース接続情報であることがあります。
   - ファイル・ベースのイベント・ストアでは、この情報はイベント・ディレクトリーとアーカイブ・ディレクトリーの名前を含むことがあります。
   - 拡張イベント・ストアは、イベント・レコードのアクセスまたは処理に必要なすべてのメタデータ情報も格納する必要があります。この情報は、JDBC 照会に必要なすべての「ソート順」情報を含むことがあります。
3. CWConnectorEventStore クラス内の適切な抽象メソッド (表 87 を参照) を実装して、イベント・ストアへのアクセスを提供します。

   イベント・ストアが必要とする CWConnectorEventStore メソッドを実装できます。ただし、次の条件が適用されます。

- 表 87 の「実装状況」列が「実装が必要」である抽象メソッドについては、実装が*必須* です。これらのメソッドは、pollForEvents() メソッドのデフォルト実装をサポートするために必要です。

  注: pollForEvents() のデフォルト実装をオーバーライドする場合は、pollForEvents() メソッドが使用する必要のある CWConnectorEventStore メソッドだけを定義できます。

- CWConnectorEventStore クラスは、表 87 の「実装状況」列が「基底クラス内で提供された実装」であるメソッドの実装を提供します。

4. 必要に応じて CWConnectorEventStore メソッドにアクセスして、pollForEvents() ポーリング・メソッドの内部からイベント取得、イベント処理、およびアーカイブを実行します。詳細については、208 ページの『pollForEvents() メソッドの実装』を参照してください。

注: CWConnectorEventStore のメソッドの詳細については、365 ページの『第 17 章 CWConnectorEventStore クラス』を参照してください。

CWConnectorEventStoreFactory インターフェース:

CWConnectorEventStoreFactory インターフェースは、イベント・ストアのインスタンスを生成するためのメソッドを提供するイベント・ストア・ファクトリーを定義します。表 88 を参照してください。

表 88. CWConnectorEventStoreFactory インターフェースのメソッド

CWConnectorEventStoreFactory メソッド	実装状況
getEventStore()	実装が必要

イベント・ストア・ファクトリーを定義するには、次のステップを実行します。

1. CWConnectorEventStoreFactory インターフェースを実装するイベント・ストア・ファクトリー・クラスを新規に作成します。 CWConnectorEventStore クラスがアクセスするイベント・ストアの名前を含めて、新しいクラスに名前を付けます。
2. ユーザーのイベント・ストア・ファクトリー・クラス内で CWConnectorEventStoreFactory インターフェースの getEventStore() メソッドを実装し、拡張した CWConnectorEventStore クラスのイベント・ストア・ファクトリーを提供します。
3. イベント・ストアをインスタンス化する際に、CWConnectorAgent クラスにデフォルトで実装されている getEventStore() メソッドを使用するかどうかを決定します。デフォルトで実装されている pollForEvents() メソッドは、この getEventStore() メソッドを使用して、イベント・ストアへの参照を取得します。
   - デフォルトで実装されている getEventStore() メソッドを使用する場合は、EventStoreFactory コネクター構成プロパティーを定義し、(CWConnectorEventStoreFactory インターフェースを実装している) イベント・ストア・ファクトリー・クラスの完全なクラス名 (パッケージ名も含む) をそのプロパティーに設定します。

     EventStoreFactory プロパティーの形式は、次のとおりです。

     connectorPackageName.EventStoreFactoryClassName

この形式において、*connectorPackageName* は次のとおりです。

```
com.crossworlds.connectors.connectorName
```

connectorName はコネクターの名前です。*EventStoreFactoryClassName* は、CWConnectorEventStoreFactory インターフェースを実装しているクラスの名前です。

注: EventStoreFactory プロパティーは、標準のプロパティーではなく、ユーザー定義のプロパティーです。このプロパティーは、Connector Configurator を使用して、イベント・ストア・ファクトリーを提供するすべてのコネクターに定義する必要があります。

EventStoreFactory を設定していない 場合は、デフォルトで実装されている getEventStore() がイベント・ストア名の生成を試みます。詳細については、275 ページの『getEventStore()』を参照してください。

- デフォルトで実装されている getEventStore() がユーザーのコネクターの要件に適合しない場合は、コネクター・クラス内でこのメソッドをオーバーライドできます。このメソッド内では、特定のカスタム・イベント・ストア・コンストラクターを呼び出すことができます。

イベント・オブジェクトの定義

Java コネクターは、イベント・ストアからイベント・レコードを取得し、それらのレコードをイベント・オブジェクト としてカプセル化します。イベント・ストア・クラスは、コネクターがイベント・ストアから取得するイベント・レコードごとにイベント・オブジェクトを構築します。各イベント・オブジェクト内の情報は、コネクターが統合ブローカーに送信するビジネス・オブジェクトを構築および取得するために使用されます。

CWConnectorEvent が定義するデフォルトのイベント・オブジェクトは、132 ページの表 46 に示すイベント情報を格納します。 CWConnectorEvent クラスは、表 89 に示すこの情報のためのアクセス・メソッドを提供します。

表 89. イベント・オブジェクトの情報を取得するためのメソッド

要素	CWConnectorEvent メソッド
イベント ID	getEventID()
ビジネス・オブジェクト名	getBusObjName()
ビジネス・オブジェクト動詞	getVerb()

表 89. イベント・オブジェクトの情報を取得するためのメソッド (続き)

要素	CWConnectorEvent メソッド
オブジェクト・キー	getIDValues(),,getKeyDelimiter() これらの CWConnectorEvent メソッドは、ビジネス・オブジェクトを識別する実際のデータ値へのアクセスを提供します。getIDValues() メソッドは、このデータが名前と値のペアであると想定します。例えば、オブジェクト・キーがビジネス・オブジェクト内の ContractId 属性のデータを格納する場合、ビジネス・オブジェクト・データ内の名前と値のペアは ContractId=45381 のようになります。イベント・レコード内のオブジェクト・キーが連結された複数のフィールドを格納する場合、getIDValues() は、getKeyDelimiter() メソッドが戻す区切り文字によってそれぞれの名前と値のペアが区切られていると想定します。区切り文字は、PollAttributeDelimiter コネクター構成プロパティーで設定されているように構成できます。区切り文字のデフォルト値はコロン (:) です。
優先順位	getPriority()
タイム・スタンプ	getEventTimeStamp()
状況	getStatus() イベント状況を設定するには、以下のメソッドを使用します。getNextEvent()、recoverInProgressEvents()、resubmitArchivedEvents()、setEventStatus()、updateEventStatus()。
説明	イベントを記述するテキスト・ストリングです。
ConnectorID	getConnectorID()

イベント・オブジェクトは、イベント・レコード内の標準情報 (表 89 を参照) を提供するのに加えて、表 90 に示す情報の accessor メソッドも提供します。

表 90. イベント・オブジェクト内の追加イベント情報

要素	説明	accessor メソッド
発効日	イベントがアクティブになって処理される日付。この情報は、1 つのシステム内でのオブジェクトへの変更をその変更が有効になる日付まで伝搬しないようにする とき (給与の変更など) に役立つことがあります。	getEffectiveDate()
イベント・ソース	イベントが発生したソース。この情報は、コネクターがアーカイブのためにイベント・ソースを追跡するときに必要になることがあります。	getEventSource(),,setEventSource()
トリガー実行ユーザー	このイベントを起動したユーザーに関連するユーザー ID。この情報は、2 つのシステム間での同期問題を回避するために使用できます。	getTriggeringUser()

デフォルトのイベント・クラスが提供する情報 (表 89 および 表 90) 以外の情報をイベント・レコードが必要とする場合は、次のステップを実行できます。

1. CWConnectorEvent クラスを拡張して、新規のイベント・クラスがカプセル化するイベント・レコードを格納しているイベント・ストアを識別できるようにそのクラスに名前を付けます。
2. イベントが必要とする可能性のあるすべての追加のデータ・メンバーを定義します。

CWConnectorEvent クラスは、表 89 と表 90 に示す accessor メソッドに対応するデータ・メンバーを格納します。アプリケーションのイベント・レコードにアクセスするために必要なその他のすべての情報は、拡張 CWConnectorEvent クラス内のデータ・メンバーとして宣言する必要があります。

3. 拡張 CWConnectorEvent クラスに追加するすべてのデータ・メンバーの accessor メソッドを提供します。

   真のカプセル化をサポートするには、データ・メンバーが拡張 CWConnectorEvent クラスの private メンバーである必要があります。これらのデータ・メンバーへのアクセスを提供するには、各データ・メンバーの値を取得するための「get」メソッドを提供します。コネクター開発者による設定が許可されているデータ・メンバーについては、「set」メソッドを定義することもできます。

注: CWConnectorEvent のメソッドの詳細については、351 ページの『第 15 章 CWConnectorEvent クラス』を参照してください。

pollForEvents() メソッドの実装

Java コネクターでは、CWConnectorAgent クラスが pollForEvents() メソッドを定義します。このクラスは、pollForEvents() のデフォルト実装を提供します。このデフォルト実装を使用するか、独自のポーリング・メソッドを使用してこのメソッドをオーバーライドするかを選択できます。ただし、pollForEvents() メソッドの実装は必須です。

図 62 の Java ベースの疑似コードに、pollForEvents() メソッドの基本ロジック・フローを示します。最初に、このメソッドは、イベント・ストアからイベントのセットを取得します。メソッドは、イベントごとに isSubscribed() メソッドを呼び出して、対応するビジネス・オブジェクトのサブスクリプションが存在するかどうかを判別します。サブスクリプションがある場合、メソッドはアプリケーションからデータを取得し、新規のビジネス・オブジェクトを作成し、gotApplEvent () を呼び出してビジネス・オブジェクトを InterChange Server に送信します。サブスクリプションがない場合、メソッドは未処理の状況値を持つイベント・レコードをアーカイブします。

```
public int pollForEvents()
{
   int status = 0;
   get the events from the event store
   for (events 1 to MaxEvents in event store) {
      extract BOName, verb, and key from the event record
        if(ConnectorBase.isSubscribed(BOName,BOverb) {
            BO = JavaConnectorUtil.createBusinessObject(BOName)
            BO.setAttrValue(key)

            retrieve application data using doVerbFor()
            BO.setVerb(Retrieve)
            BO.doVerbFor()
            BO.setVerb(BOverb)
            status = gotApplEvent(BusinessObject);

            archive event record with success or failure status
        }
        else {
            archive item with unsubscribed status
   }
   return status;
}
```

図 62. Java pollForEvents() の例

注: ポーリング・メソッドの基本ロジックのフローチャートについては、96 ページの図 27 を参照してください。

このセクションでは、pollForEvents() メソッドが実行する一般的なイベント処理の基本ロジック内の各ステップの詳細情報を提供します。表 91 に、これらの基本ステップの要約を示します。

表 91. pollForEvents() メソッドの基本ロジック

ステップ		詳細情報
1.	コネクターのサブスクリプション・マネージャーをセットアップします。	210 ページの『サブスクリプション・マネージャーへのアクセス』
2.	コネクターがイベント・ストアへの有効な接続を維持しているかどうかを検証します。	210 ページの『イベント・ストアへのアクセス前の接続の検証』
3.	イベント・ストアから指定された数のイベント・レコードを取得し、それらのレコードをイベント配列内に格納します。イベント配列内を循環処理します。イベント・ストア内で各イベントに進行中としてマークを付け、処理を開始します。	211 ページの『イベント・レコードの取得』
4.	イベント・レコードからビジネス・オブジェクト名、動詞、およびキー・データを取得します。	213 ページの『ビジネス・オブジェクトの名前、動詞、およびキーの取得』
5.	イベントへのサブスクリプションの有無を検査します。	214 ページの『イベントへのサブスクリプションの検査』
	イベントへのサブスクライバーがある場合	
	• アプリケーション・データを取得し、ビジネス・オブジェクトを作成します。	216 ページの『アプリケーション・データの取得』
	• イベント・デリバリーのために、コネクター・フレームワークにビジネス・オブジェクトを送信します。	218 ページの『コネクター・フレームワークへのビジネス・オブジェクトの送信』
	• イベント処理を完了します。	222 ページの『イベントの処理の完了』

表 91. pollForEvents() メソッドの基本ロジック (続き)

ステップ		詳細情報
	イベントへのサブスクライバーがない 場合は、イベント状況を Unsubscribed に更新します。	214 ページの『イベントへのサブスクリプションの検査』
6.	イベントのアーカイブ	224 ページの『イベントのアーカイブ』
7.	イベント・ストアへのアクセスに使用されたリソースを解放します。	

サブスクリプション・マネージャーへのアクセス

コネクター・フレームワークは、コネクター初期化の一部として、サブスクリプション・マネージャーのインスタンスを生成します。このサブスクリプション・マネージャーは、サブスクリプション・リストを最新の情報に更新します。(詳細については、14 ページの『ビジネス・オブジェクトのサブスクリプションとパブリッシュ』を参照してください。) コネクターは、コネクター基底クラス内に含まれるサブスクリプション・ハンドラー を通じて、サブスクリプション・マネージャーとコネクター・サブスクリプション・リストにアクセスします。コネクターは、このクラスのメソッドを使用して、ビジネス・オブジェクトへのサブスクライバーがあるかどうかを判別し、コネクター・コントローラーにビジネス・オブジェクトを送信できます。

注: C++ コネクターの場合とは異なり、Java コネクターではサブスクリプション・ハンドラーをセットアップする必要はありません。この機能は、`CWConnectorAgent` クラス内で処理されます。

イベント・ストアへのアクセス前の接続の検証

コネクター・クラス内の `agentInit()` メソッドがアプリケーション固有のコンポーネントを初期化するときに、最も一般的なタスクの 1 つは、アプリケーションへの接続を確立することです。ポーリング・メソッドは、イベント・ストアへのアクセスを必要とします。したがって、`pollForEvents()` メソッドは、イベントの処理を開始する前に、コネクターがアプリケーションへの接続を維持しているかどうかを検証する必要があります。この検証を実行する方法は、アプリケーション固有です。詳細については、アプリケーションのドキュメンテーションを参照してください。

コネクターのアプリケーション固有のコンポーネントの設計時には、アプリケーションへの接続が切断されたときにはコンポーネントがシャットダウンするようにコーディングすることをお勧めします。接続が切断されている場合、コネクターは、イベント・ポーリングを継続しません。その代わりに、コネクターは、`APPRESPONSETIMEOUT` を戻して、アプリケーションへの接続が切断されたことをコネクター・フレームワークに通知します。

注: `pollForEvents()` 内から `doVerbFor()` によって戻された `APPRESPONSETIMEOUT` 結果状況を取得するには、`CWConnectorEventStore` クラスの `getTerminate()` メソッドを使用します。詳細については、 216 ページの『アプリケーション・データの取得』 を参照してください。

イベント・レコードの取得

イベント通知をコネクター・フレームワークに送信する前に、ポーリング・メソッドは、イベント・ストアからイベント・レコードを取得する必要があります。表 92 に、イベント・ストアからイベント・レコードを取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 92. イベント取得用のクラスとメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorAgent	getEventStore()
CWConnectorEventStoreFactory	getEventStore()
CWConnectorEventStore	fetchEvents(),, getNextEvent(),,updateEventStatus()

ポーリング・メソッドは、一度に 1 つのイベント・レコードを取得し、そのイベント・レコードを処理できます。また、ポーリングごとに指定された数のイベント・レコードを取得し、それらのイベント・レコードをキャッシュとしてイベント配列に入れることもできます。ポーリングごとに複数のイベントを処理すると、アプリケーションが多数のイベントを生成するときのパフォーマンスを向上させることができます。

任意のポーリング・サイクル内で選出されるイベントの数は、コネクター構成プロパティー PollQuantity を使用して構成できます。システム管理者は、インストール時に PollQuantity の値を 50 などの適切な数値に設定します。ポーリング・メソッドは、getConfigProp() を使用して PollQuantity プロパティーの値を取得してから、指定された数のイベント・レコードを取得し、それらのイベント・レコードを単一のポーリングで処理できます。

コネクターは、コネクターがイベント・ストアから読み取って処理を開始したすべてのイベントに、進行中状況を割り当てます。イベントの処理中に、イベントの送信または失敗を示すためにイベント状況を更新する前にコネクターが終了した場合は、表内に進行中イベントが残されます。このような進行中イベントの回復方法の詳細については、174 ページの『進行中イベントのリカバリー』を参照してください。

Java コネクター・ライブラリーは、イベント・ストアを表すための CWConnectorEventStore クラスを提供します。このイベント・ストアからイベント・レコードを取得するために、ポーリング・メソッドは次のアクションを実行します。

1. CWConnectorAgent クラスで定義されている getEventStore() メソッドを使用して、イベント・ストア・オブジェクトをインスタンス化します。デフォルトで実装されているこのメソッドは、EventStoreFactory コネクター構成プロパティーで指定されているイベント・ストア・ファクトリー・クラスの getEventStore() を呼び出します。イベント・ストア・ファクトリー・クラスは、ユーザーのイベント・ストア用の CWConnectorEventStoreFactory インターフェースを実装します。詳細については、205 ページの『CWConnectorEventStoreFactory インターフェース』を参照してください。
2. fetchEvents() メソッドを使用して、イベント・ストアから指定された数のイベント・レコードを取得します。

CWConnectorEventStore クラスの一部として fetchEvents() メソッドを実装する必要があります。このメソッドは、取得対象のイベント・レコードの数として、 PollQuantity コネクター構成プロパティーの値を使用できます。メソッドは、次のアクションを実行する必要があります。

- メソッドが取得するイベント・レコードごとに、1 つの CWConnectorEvent イベント・オブジェクトを作成します。

  これらのイベント・レコードは、タイム・スタンプに基づいてソートできます。イベントの優先順位に基づいたイベント・レコードの取得の詳細については、149 ページの『イベント優先順位によるイベントの処理』を参照してください。

  注: イベント・ストアがイベント表と共にアプリケーション・データベース内に実装される場合、fetchEvents() メソッドは、JDBC メソッドを使用してイベント表にアクセスできます。この方法は、C++ コネクターが ODBC メソッドを使用する場合とほぼ同様です。

- 各イベント・オブジェクトを eventsInProgress イベント・ベクトルに書き込みます。

アプリケーションがイベント・ストアにアクセスできないためにイベントを取り出すことができない場合、fetchEvents() メソッドは StatusChangeFailedException 例外をスローします。 pollForEvents() メソッドは、この例外をキャッチした場合、アプリケーションのイベント・ストアから応答が返されなかったことを示すために、APPRESPONSETIMEOUT 結果状況を戻すことがあります。

- eventsInProgress イベント・ベクトル内でイベントをループ処理して、各イベント・オブジェクトに対して次のアクションを実行します。
  - getNextEvent() メソッドを使用して、次に処理するイベント・オブジェクトを取得します。
  - updateEventStatus() メソッドを使用して、イベント・レコード (イベント・ストア内) とイベント・オブジェクト (イベント・ベクトルから取得) の両方の状況を IN_PROGRESS に更新します。

    アプリケーションがイベント・ストアにアクセスできないためにイベント状況を変更することができない場合、updateEventStatus() メソッドは StatusChangeFailedException 例外をスローします。pollForEvents() メソッドは、この例外をキャッチした場合、アプリケーションのイベント・ストアから応答が返されなかったことを示すために、APPRESPONSETIMEOUT 結果状況を戻すことがあります。

イベント状況を IN_PROGRESS に設定することは、ポーリング・メソッドがイベントに対する処理を開始したことを示します。図 63 に、イベント・ストアからイベント・レコードを取得し、各イベント・レコードにイベント・オブジェクトとしてアクセスするコード・フラグメントを示します。

```
// Instantiate event store
CWConnectorEventStore evts=getEventStore();

// Fetch PollQuantity number of events from the application.
try
   {
   evts.fetchEvents();
   }
catch (StatusChangeFailedException e)
   {
   // log error message
   return CWConnectorConstant.FAIL;
   }
}
// Get the property values for PollQuantity
int pollQuantity;
String poll=CWConnectorUtil.getConfigProp("PollQuantity");
if (poll == null || poll.equals(""))
   pollQuantity=1;
else
   pollQuantity=Integer.parseInt(poll);

   for (int i=0; i < pollQuantity; i++)
   {
      // Process each event retrieved from the application.
      // Get the next event to be processed.
      evtObj=evts.getNextEvent();
```

図 63. イベント・ストアからのイベント・レコードの取得

ビジネス・オブジェクトの名前、動詞、およびキーの取得

コネクターは、イベントを取得した後で、イベント・レコードからイベント ID、オブジェクト・キー、ビジネス・オブジェクト名、およびビジネス・オブジェクト動詞を抽出します。コネクターは、ビジネス・オブジェクト名と動詞を使用して、統合ブローカーがこの型のビジネス・オブジェクトに関係するかどうかを判別します。ビジネス・オブジェクトとそのアクティブ動詞へのサブスクライバーがある場合、コネクターは、エンティティー・キーを使用してデータの完全セットを取得します。

表 93 に、取得されたイベント・レコードからビジネス・オブジェクト定義と動詞の名前を取得するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 93. イベント情報取得用のメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEvent	getBusObjName(),, getVerb()

重要: コネクターは、イベント・レコード内の動詞と同じ動詞と共にビジネス・オブジェクトを送信します。

`getNextEvent()` メソッドが処理対象のイベント・オブジェクトを取得したら、Java コネクターは、`CWConnectorEvent` クラスの適切な accessor メソッドを使用して、イベント・サブスクリプションの検査のために必要な次の情報を取得できます。

イベント ID	getEventID()
ビジネス・オブジェクト名	getBusObjName()
動詞	getVerb()
オブジェクト・キー	getIDValues()

これらの accessor メソッドを使用するサンプル・コードについては、 215 ページの図 64を参照してください。

イベントへのサブスクリプションの検査

統合ブローカーが特定のビジネス・オブジェクトと動詞の受け取りに関係するかどうかを判別するために、ポーリング・メソッドは、`isSubscribed()` メソッドを呼び出します。 `isSubscribed()` メソッドは、現在のビジネス・オブジェクト名と動詞を引き数として取得します。ビジネス・オブジェクト名と動詞は、リポジトリー内のビジネス・オブジェクト名と動詞に一致する必要があります。

WebSphere InterChange Server

InterChange Server を使用するビジネス・インテグレーション・システムの場合、ポーリング・メソッドは、特定の動詞を持つビジネス・オブジェクトにサブスクライブしているコラボレーションがあるかどうかを判別できます。初期化時に、コネクター・フレームワークは、コネクター初期化時のコネクター・コントローラーからサブスクリプション・リストを要求します。実行時に、アプリケーション固有のコンポーネントは、`isSubscribed()` を使用してコネクター・フレームワークに照会することにより、特定のビジネス・オブジェクトにサブスクライブしているコラボレーションがあることを検証できます。アプリケーション固有のコネクター・コンポーネントは、現在サブスクライブしているコラボレーションが存在する場合に限り、イベントを送信できます。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークでは、統合ブローカーがコネクターによってサポートされるすべての ビジネス・オブジェクトに関係していると想定します。ポーリング・メソッドが `isSubscribed()` メソッドを使用して特定のビジネス・オブジェクトへのサブスクリプションについての情報をコネクター・フレームワークに照会する場合、メソッドは、コネクターがサポートするすべての ビジネス・オブジェクトについて `true` を戻します。

表 94 に、イベントへのサブスクリプションを検査するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 94. サブスクリプション検査用のクラスとメソッド

Java コネクター・ライブラリー・クラス	メソッド
`CWConnectorAgent`	isSubscribed()

表 94. サブスクリプション検査用のクラスとメソッド (続き)

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEventStore	updateEventStatus(),, archiveEvent(),, deleteEvent()

ポーリング・メソッドは、isSubscribed() が戻す値に基づき、そのイベントへのサブスクライバーが存在するかどうかに応じて、次のアクションの 1 つを実行します。

- イベントへのサブスクライバーが存在する場合、コネクターは、『サブスクリプションがあるイベント』で述べるアクションの 1 つを実行します。
- イベントへのサブスクリプションがない場合、コネクターは、 216 ページの『サブスクリプションがないイベント』で述べるアクションの 1 つを実行します。

Java コネクターでは、サブスクリプション・マネージャーはコネクター基底クラスの一部であるため、isSubscribed() メソッドは CWConnectorAgent クラス内で定義されます。メソッドは、サブスクライバーがある場合には true を、サブスクライバーがない場合には false を戻します。図 64 に、Java コネクター内でサブスクリプションを検査するコード・フラグメントを示します。

```
if (isSubscribed(evtObj.getBusObjName(),evtObj.getVerb())) {
      // handle event
} else
   {
      // Update the event status to UNSUBSCRIBED.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.UNSUBSCRIBED);

      // Archive the event (if archiving is supported)
      return CWConnectorConstant.FAIL;
   }
```

図 64. イベント・サブスクリプションの検査

イベントへのサブスクリプションが存在しない場合、このコード・フラグメントは、updateEventStatus() メソッドを使用してイベントの状況を UNSUBSCRIBED に更新してから、イベントをアーカイブします。

サブスクリプションがあるイベント: イベントへのサブスクライバーがある場合、コネクターは次のアクションを実行します。

実行されるコネクター・アクション	詳細情報
アプリケーション・データベース内のエンティティーから、ビジネス・オブジェクト・データの完全セットを取得します。	216 ページの『アプリケーション・データの取得』
コネクター・フレームワークにビジネス・オブジェクトを送信します。コネクター・フレームワークは、そのビジネス・オブジェクトを統合ブローカーにルーティングします。	218 ページの『コネクター・フレームワークへのビジネス・オブジェクトの送信』
イベントの処理を完了させます。	222 ページの『イベントの処理の完了』

実行されるコネクター・アクション	詳細情報
統合ブローカーが後からサブスクライブするときのために、イベントをアーカイブします (アーカイブが実装されている場合)。	224 ページの『イベントのアーカイブ』

サブスクリプションがないイベント: イベントへのサブスクリプションがない場合、コネクターは次のアクションを実行します。

- サブスクライバーがなかったことを示すために、イベントの状況を「Unsubscribed」に更新します。
- 統合ブローカーが後からサブスクライブするときのために、イベントをアーカイブします (アーカイブが実装されている場合)。イベント・レコードをアーカイブ・ストアに移動すると、ポーリング・メソッドは、アンサブスクライブされたイベントを選出しません。詳細については、224 ページの『イベントのアーカイブ』を参照してください。
- サブスクリプションが現在存在しない保留中のイベントがあることを示すために、「fail」(Java コネクターの場合は `FAIL` 結果状況) を戻します。

  イベントへのサブスクリプションが存在しない場合にはコネクターが「fail」を戻すようにすることをお勧めします。しかし、設計の要件に基づいた結果状況を戻すことが可能です。

アンサブスクライブされたイベントでは、他の処理は実行されません。統合ブローカーがこれらのイベントに後からサブスクライブする場合、システム管理者は、アンサブスクライブされたイベント・レコードをアーカイブ・ストアからイベント・ストアに再び移動することができます。

アプリケーション・データの取得

イベントへのサブスクライバーがある場合、ポーリング・メソッドは次のステップを実行する必要があります。

1. エンティティーのデータの完全なセットをアプリケーションから取得します。

   エンティティー・データの完全セットを取得するために、ポーリング・メソッドは、(イベント内に格納されている) エンティティー・キー情報の名前を使用して、アプリケーション内でエンティティーを探し出す必要があります。ポーリング・メソッドは、イベントが次の動詞を持つときに、アプリケーション・データの完全セットを取得する必要があります。

   - Create
   - Update
   - 論理削除をサポートするアプリケーションの Delete イベント

   物理削除をサポートするアプリケーションからの Delete イベントでは、アプリケーションがデータベースからエンティティーをすでに削除しているために、コネクターがエンティティー・データを取得できないことがあります。削除処理の詳細については、151 ページの『削除イベントの処理』を参照してください。
2. エンティティー・データをビジネス・オブジェクト内にパッケージします。

ビジネス・オブジェクト内にデータが取り込まれた後で、ポーリング・メソッドは、ビジネス・オブジェクトをサブスクライバーにパブリッシュできます。

表 95 に、アプリケーション・データベースからエンティティー・データを取得し、そのデータをビジネス・オブジェクトに取り込むために Java コネクター・ライブラリーが提供するメソッドを示します。

表 95. ビジネス・オブジェクト・データの取得用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEventStore	getBO()

注: イベントが削除操作であり、アプリケーションがデータの物理削除をサポートする場合、データがアプリケーションから削除されているために、コネクターがデータを取得できない可能性が高くなります。この場合、コネクターは単にビジネス・オブジェクトを作成し、イベント・レコードのオブジェクト・キーからキーを設定し、ビジネス・オブジェクトを送信します。

Java コネクターでは、pollForEvents() の内部からアプリケーション・データを取得する標準的な方法は、CWConnectorEventStore クラス内の getBO() メソッドを使用することです。このメソッドは、次のステップを実行します。

- 新規ビジネス・オブジェクトを保持するための一時的 CWConnectorBusObj オブジェクトを作成します。
- CWConnectorBusObj オブジェクトに、指定されたイベント・オブジェクトからデータとキー値を取り込みます。
- イベントの動詞が Create または Update である場合は、ビジネス・オブジェクトの動詞を RetrieveByContent に設定し、doVerbFor() メソッドを呼び出してアプリケーションから残りの属性値を取得します。
- データとキー値を取り込んだ CWConnectorBusObj オブジェクトを呼び出し側に戻します。

getBO() への呼び出しが正常に実行された場合は、データとキー値を取り込んだ CWConnectorBusObj オブジェクトが戻されます。次の行に、データとキー値を取り込んだ bo という名前の CWConnectorBusObj オブジェクトを戻す getBO() への呼び出しを示します。

```
bo = evts.getBO(evtObj);
```

getBO() 呼び出しが正常に実行されなかった 場合、ポーリング・メソッドは次のステップを実行します。

- getBO() がスローする例外をすべてキャッチします。
- イベント・オブジェクト内の ERROR_OBJECT_NOT_FOUND 状況の有無を検査して、doVerbFor() メソッドがアプリケーション内のビジネス・オブジェクト・データを検出できなかったかどうかを判別します。
- getBO() が戻した null 値の有無を検査して、doVerbFor() が正常に実行されなかったかどうかを判別します。
- getTerminate() メソッドを使用して、コネクター終了フラグが設定されているかどうかを確認します (このフラグは、(getBO() メソッド内から呼び出される) doVerbFor() が APPRESPONSETIMEOUT 結果出力を戻したかどうかを示します)。

getTerminate() が true を戻した場合、pollForEvents() は、APPRESPONSETIMEOUT 結果出力を戻して、コネクターを終了する必要があります。

注: デフォルトで実装されている getBO() は、doVerbFor() の結果状況を確認し、doVerbFor() が APPRESPONSETIMEOUT 結果状況を戻している場合は、setTerminate() メソッドを呼び出します。デフォルトで実装されている getBO() をオーバーライドし、デフォルトで実装されている pollForEvents() を引き続き使用する場合、ユーザーが実装した getBO() は、これと同じタスクを実行する必要があります。

IBM WebSphere Business Integration システム内の ObjectEventId 属性は、システム内でのビジネス・オブジェクトのフローを追跡するために使用されます。また、階層型ビジネス・オブジェクト要求内での子ビジネス・オブジェクトの順序は応答ビジネス・オブジェクト内で変更される可能性があるため、この属性は、要求と応答の間で子ビジネス・オブジェクトを追跡するためにも使用されます。

コネクターは、親ビジネス・オブジェクトまたはその子ビジネス・オブジェクトの ObjectEventId 属性の値を取り込む必要はありません。ビジネス・オブジェクトが ObjectEventId 属性の値を持たない場合は、ビジネス・インテグレーション・システムが属性の値を生成します。ただし、コネクターが子 ObjectEventId に値を取り込む場合は、その特定のビジネス・オブジェクトのすべての階層レベルにあるすべての ObjectEventId 値の中で一意の値を割り当てる必要があります。ObjectEventId 値は、イベント通知機構の一部として生成できます。ObjectEventId 値の生成方法の提案については、134 ページの『イベント ID』を参照してください。

コネクター・フレームワークへのビジネス・オブジェクトの送信

ビジネス・オブジェクトのデータが取得されたら、ポーリング・メソッドは次のタスクを実行します。

- 『ビジネス・オブジェクト動詞の設定』
- 219 ページの『ビジネス・オブジェクトの送信』

表 96 に、これらのタスクを実行するために Java コネクター・ライブラリーが提供するメソッドを示します。

表 96. 動詞設定とビジネス・オブジェクト送信のためのクラスおよびメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorBusObj	setVerb()
CWConnectorEvent	getVerb()
CWConnectorAgent	gotApplEvent()

ビジネス・オブジェクト動詞の設定: ビジネス・オブジェクト内の動詞をイベント・レコード内で指定された動詞に設定するために、ポーリング・メソッドは、ビジネス・オブジェクト・メソッド setVerb() を呼び出します。ポーリング・メソッドは、イベント・ストアにあるイベント・レコード内の動詞と同じ動詞を設定します。

注: イベントが物理削除である場合は、イベント・レコードからのオブジェクト・キーを使用して、ビジネス・オブジェクト内のキーを設定し、動詞を Delete に設定します。

Java コネクターでは、getBO() メソッドが戻すデータとキー値を取り込んだ CWConnectorBusObj オブジェクトに RetrieveByContent の動詞が設定されたままです。ポーリング・メソッドは、CWConnectorBusObj クラスの setVerb() メソッドを使用して、ビジネス・オブジェクトの動詞をその元の値に設定する必要があります。次のコード・フラグメントを参照してください。

```
// Set verb to action as indicated in the event record
busObj.setVerb(evntObj.getVerb());
```

このコード・フラグメントでは、ポーリング・メソッドが CWConnectorEvent クラスの getVerb() を使用して、イベント・レコードから動詞を取得します。そして、この動詞が setVerb() によってビジネス・オブジェクトにコピーされます。

ビジネス・オブジェクトの送信: ポーリング・メソッドは、メソッド gotApplEvent() を使用して、コネクター・フレームワークにビジネス・オブジェクトを送信します。このメソッドは、次のステップを実行します。

- コネクターがアクティブであることを確認します。
- イベントへのサブスクリプションがあることを確認します。
- コネクター・フレームワークにビジネス・オブジェクトを送信します。

コネクター・フレームワークは、イベント・オブジェクトにある処理を施して、データを直列化し、そのデータが正しく永続化されるようにします。その後、イベントが送信されたことを確認します。

WebSphere InterChange Server

InterChange Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークは、イベントが CORBA IIOP を使用して ICS に送信されたかまたはキューに書き込まれた (イベント通知用にキューを使用している場合) ことを確認します。イベントを ICS に送信する場合、コネクター・フレームワークは、ビジネス・オブジェクトをコネクター・コントローラーに転送します。コネクター・コントローラーは、アプリケーション固有のビジネス・オブジェクトを汎用ビジネス・オブジェクトに変換するために必要なすべてのマッピングを実行します。その後、コネクター・コントローラーは、汎用ビジネス・オブジェクトを適切なコラボレーションに送信できます。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークは、イベントが XML WebSphere MQ メッセージに変換され、適切な MQ キューに書き込まれることを保証します。

ポーリング・メソッドは、gotApplEvent() からの戻りコードを検査して、エラー条件が適切に処理されたことを確認します。例えば、ポーリング・メソッドは、イベント・デリバリーが正常に実行されるまでイベント・ストアからイベントを除去しません。ポーリング・メソッドは、イベント・デリバリーの結果が反映されるようにイベント・レコードの状況を更新します。表 97 は、gotApplEvent() からの戻りコードに応じて取られ得るイベント状況値を示しています。

表 97. gotApplEvent() によりイベント・デリバリー後に設定される可能性のあるイベント状況

イベント・デリバリーの状態	gotApplEvent() の戻りコード	イベント状況
イベント・デリバリーが正常に実行された場合	SUCCEED	SUCCESS
イベントへのサブスクリプションがない場合	NO_SUBSCRIPTION_FOUND	UNSUBSCRIBED
コネクターが休止していた場合	CONNECTOR_NOT_ACTIVE	READY_FOR_POLL
イベント・デリバリーが失敗した場合	FAIL	ERROR_POSTING_EVENT

gotApplEvent() メソッドは、コネクター・フレームワークがビジネス・オブジェクトを正常にデリバリーした場合には SUCCEED を戻します。ポーリング・メソッドは、gotApplEvent() からの戻りコードを検査して、イベント・レコードの状況が適切に更新されたかどうかを調べます。gotApplEvent() から FAIL 以外の 戻りコードが戻された場合、ポーリング・メソッドは SUCCEED を戻し、イベントに関するポーリングが続けられるようにします。しかし、gotApplEvent() から FAIL が戻された場合には、イベント・デリバリーが失敗しているため、ポーリング・メソッドは失敗し、エラー・メッセージをログに記録します。

表 98 は、gotApplEvent() の戻りコードに基づいて pollForEvents() が実行するアクションを示しています。

表 98. gotApplEvent() によるイベント・デリバリー後に実行される pollForEvents() のアクション

gotApplEvent() の戻りコード	pollForEvents() のアクション
SUCCEED	1. イベント状況を SUCCESS にリセットします。 2. ArchiveProcessed コネクター・プロパティーに true が設定されている場合は、イベントをアーカイブし、そのイベントをイベント・ストアから削除します。 3. ポーリングを続行します。

表 98. *gotApplEvent()* によるイベント・デリバリー後に実行される *pollForEvents()* のアクション (続き)

gotApplEvent() の戻りコード	pollForEvents() のアクション
NO_SUBSCRIPTION_FOUND	1. エラー・メッセージをログに記録します。 2. イベント状況を UNSUBSCRIBED にリセットします。 3. ArchiveProcessed コネクター・プロパティーに true が設定されている場合は、イベントをアーカイブし、そのイベントをイベント・ストアから削除します。 4. ポーリングを続行します。
CONNECTOR_NOT_ACTIVE	1. 通知メッセージをトレース・レベル 3 でログに記録します。 2. 将来の再実行に備えてイベントを作成します。 • アプリケーション・アダプターの場合は、イベント状況を READY_FOR_POLL にリセットします。 • テクノロジー・アダプターの場合は、イベントをプッシュして戻します (可能な場合)。 3. pollForEvents() 結果状況として SUCCEED を戻します。 **注:** この場合、イベントはアーカイブされません。
FAIL	1. エラー・メッセージをログに記録します。 2. イベント状況を ERROR_POSTING_EVENT にリセットします。 3. ArchiveProcessed コネクター・プロパティーに true が設定されている場合は、イベントをアーカイブし、そのイベントをイベント・ストアから削除します。 4. pollForEvents() 結果状況として FAIL を戻します。

表 98 が示すように、gotApplEvents() メソッドが CONNECTOR_NOT_ACTIVE 結果状況を戻したときに pollForEvents() が実行するアクションは、作成したコネクターのタイプによって異なります。 (データベースをそのイベント・ストアとして使用するアプリケーションを備えた特定のコネクター内の) アプリケーション・コネクターの場合、pollForEvents() メソッドは、イベントの状況を READY_FOR_POLL にリセットし、イベントを「未処理」状態に戻す必要があります。

ただし、テクノロジー・コネクターの場合 (特に、イベント表を使用せず、そのためにイベントを必ずしも「未処理」状態に戻すことができないコネクターの場合)、コネクターは、イベントを「プッシュ」して戻すのではなく、そのイベントをメモリー内に保持し、pollForEvents() からの結果状況として SUCCEED を戻すことができます。コネクターは、アダプターが再アクティブ化され、pollForEvents() が再度呼び出されるまで、このイベントをメモリーに保持しておく必要があります。コネクターはこの時点で、イベントのリパブリッシュを試みることができます。

次のコード・フラグメントは、この機能の実装方法を示しています。

```
BusinessObject eventOnHold;

pollForEvents(...)
{
   ...
   if eventOnHold != null
   {
      event = eventOnHold;
      eventOnHold = null;
   }
```

```
else
{
   event = getNextUnprocessedEvent();
}

...

result = gotApplEvent( event );
if (result == CWConnectorConstant.CONNECTOR_NOT_ACTIVE )
{
   eventOnHold = event;
   return CWConnectorConstant.SUCCEED;
}
```

注: イベントをアクティブに処理している最中にアダプターを一時停止し、その後でこのアダプターを終了した場合 (または、アダプターがひとりでに突然終了した場合) は、(上記のロジックを使用する) コネクターがメモリーにコピーしたこれらのイベントが、「未確定」イベントになります。未確定イベントの処理方法は、アダプターごとに異なります。ただし、このロジックの結果は、たとえアダプターが正常に終了したように見えても、「未確定」イベントが作成される可能性があることを意味します。これらのイベントは、失われません。

CONNECTOR_NOT_ACTIVE 戻り状況に対する pollForEvents() 応答を実装するときは、ここで説明したプログラミング方法では、アダプターがイベントを処理して統合ブローカーに送信する間、イベントは「進行中」状態に設定されるものと想定していることに注意してください。ただし、すべてのアダプターがこのような方法で実装されるわけではありません。アダプターは、単純にイベントをソースから受け取り、次に gotApplEvent() を呼び出して、そのイベントを統合ブローカーに送信する場合もあります。イベントを受け取り、gotApplEvent() を呼び出すまで間にこのアダプターが終了すると、そのイベントは失われます。このようなアダプターを再始動しても、そのイベントを再処理することはできません。

イベントの処理の完了

表 99に示すタスクが完了すると、イベントの処理が完了します。

表 99. イベント処理のステップ

イベント処理タスク	詳細情報
ポーリング・メソッドがイベントのアプリケーション・データを取得し、イベントを表すビジネス・オブジェクトを作成した。	216 ページの『アプリケーション・データの取得』
ポーリング・メソッドがコネクター・フレームワークにビジネス・オブジェクトを送信した。	218 ページの『コネクター・フレームワークへのビジネス・オブジェクトの送信』

注: 階層型ビジネス・オブジェクトでは、ポーリング・メソッドが親ビジネス・オブジェクトおよびすべての子ビジネス・オブジェクトのアプリケーション・データを取得し、コネクター・フレームワークに完全な階層型ビジネス・オブジェクトを送信すると、イベント処理が完了します。イベント通知機構は、親ビジネス・オブジェクトだけでなく、階層型ビジネス・オブジェクト全体を取得および送信する必要があります。

ポーリング・メソッドは、イベント状況がイベント処理の完了を正しく反映していることを確認する必要があります。したがって、次の条件を*両方とも* 処理する必要があります。

- 『正常なイベント処理の取り扱い』
- 『失敗したイベント処理の取り扱い』

正常なイベント処理の取り扱い: 表 99 に示すタスクが正常に実行されると、イベントの処理が正常に完了します。ポーリング・メソッドが正常に実行されたイベントの処理を完了するステップは、次のとおりです。

1. コネクター・フレームワークがビジネス・オブジェクトをメッセージング・システムに正常にデリバリーしたことを示す「success」戻りコードを、`gotApplEvent()` メソッドから受け取ります。
2. アーカイブ・ストアにイベントをコピーします。詳細については、224 ページの『イベントのアーカイブ』を参照してください。
3. アーカイブ・ストア内のイベントの状況を設定します。
4. イベント・ストアからイベント・レコードを削除します。

   イベント・デリバリーが正常に実行されるまで、ポーリング・メソッドはイベント表からイベントを除去しません。

注: ステップの順序は、実装の種類によって異なることがあります。

失敗したイベント処理の取り扱い: イベントの処理中にエラーが発生した場合、コネクターは、エラーが発生したことを示すためにイベント状況を更新します。表 100 に、イベント処理中に発生することのあるエラーに基づく可能なイベント状況値を示します。

表 100. イベント処理でエラーが発生した後に設定される可能性のあるイベント状況

イベント・デリバリーの状態	イベント状況	ポーリングが終了するか?
イベントの処理中にエラーが発生した場合	ERROR_PROCESSING_EVENT	終了しない。イベント・ストアから次のイベントを取得する。
イベント・デリバリーが失敗した場合	ERROR_POSTING_EVENT	はい
イベントへのサブスクリプションがない場合	UNSUBSCRIBED	終了しない。イベント・ストアから次のイベントを取得する。

例えば、エンティティー・キーに一致するアプリケーション・エンティティーがない場合は、イベント状況を「イベント処理中のエラー」に更新します。イベントを正常にデリバリーできない場合は、イベント状況を「イベント送付中のエラー」に更新します。219 ページの『ビジネス・オブジェクトの送信』で説明したように、ポーリング・メソッドは、`gotApplEvent()` からの戻りコードを検査して、戻されたすべてのエラーが適切に処理されたことを確認します。

いずれの場合でも、システム管理者が分析できるように、イベントをイベント・ストア内に残しておく必要があります。ポーリング・メソッドは、イベントを照会するときに、エラー状況を持つイベントを除外してそれらのイベントが選出されないようにします。イベントのエラー条件が解決されたら、システム管理者は、次回のポーリングでコネクターがイベントを選出するようにイベント状況を手動でリセットできます。

イベントのアーカイブ

イベントのアーカイブでは、イベント・レコードをイベント・ストアからアーカイブ・ストアに移動します。 Java コネクター・ライブラリーは、イベント・ストアを表すための CWConnectorEventStore クラスを提供します。このクラスには、アーカイブ・ストアが含まれます。表 101 に、イベントをアーカイブするために Java コネクター・ライブラリーが提供するメソッドを示します。

表 101. イベントのアーカイブ用メソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEventStore	updateEventStatus(), archiveEvent(), deleteEvent()

注: アーカイブの概要については、147 ページの『イベントのアーカイブ』を参照してください。

このイベント・ストアからイベント・レコードをアーカイブするために、ポーリング・メソッドは次のアクションを実行します。

1. ArchiveProcessed などの適切なコネクター構成プロパティーの値を検査して、アーカイブが実装されていることを確認します。詳細については、148 ページの『コネクターのアーカイブ用構成』を参照してください。
2. archiveEvent() メソッドを使用して、イベント・レコードをアーカイブ・ストアからイベント・ストアにコピーします。

   イベント・アーカイブを提供するには、CWConnectorEventStore クラスの一部として archiveEvent() メソッドを実装する必要があります。このメソッドは、コピー対象のイベント・レコードをそのイベント ID によって識別します。

   アプリケーションがイベント・ストアにアクセスできないためにイベントをアーカイブすることができない場合、archiveEvents() メソッドは ArchiveFailedException 例外をスローします。 pollForEvents() メソッドは、この例外をキャッチした場合、アプリケーションのイベント・ストアから応答が返されなかったことを示すために、APPRESPONSETIMEOUT 結果状況を戻すことがあります。
3. イベントをアーカイブする理由が反映されるように、updateEventStatus() メソッドを使用して、アーカイブ・レコードのイベント状況を更新します。

   表 102 に、アーカイブ・レコードでの可能なイベント状況定数を示します。

表 102. アーカイブ・レコードのイベント状況定数

イベント状況	説明
SUCCESS	イベントは検出され、コネクターはそのイベントに対応するビジネス・オブジェクトを作成し、それをコネクター・フレームワークに送信しました。詳細については、223 ページの『正常なイベント処理の取り扱い』を参照してください。
UNSUBSCRIBED	イベントが検出されたが、イベントへのサブスクリプションがないため、イベントがコネクター・フレームワークを経て統合ブローカーに送信されませんでした。詳細については、214 ページの『イベントへのサブスクリプションの検査』を参照してください。

表 102. アーカイブ・レコードのイベント状況定数 (続き)

イベント状況	説明
ERROR_PROCESSING_EVENT	イベントは検出されましたが、コネクターがイベントの処理中に、エラーが発生しました。イベントに対応するビジネス・オブジェクトの作成中またはビジネス・オブジェクトをコネクター・フレームワークに送信中にエラーが発生しました。詳細については、223 ページの『失敗したイベント処理の取り扱い』を参照してください。

アプリケーションがイベント・ストアにアクセスできないためにイベント状況を変更することができない場合、updateEventStatus() メソッドは StatusChangeFailedException 例外をスローします。 pollForEvents() メソッドは、この例外をキャッチした場合、アプリケーションのイベント・ストアから応答が返されなかったことを示すために、APPRESPONSETIMEOUT 結果状況を戻すことがあります。

4. deleteEvent() メソッドを使用して、イベント・レコードをイベント・ストアから削除します。

   CWConnectorEventStore クラスの一部として deleteEvent() メソッドを実装する必要があります。このメソッドは、削除対象のイベント・レコードを識別するためにイベント ID を使用します。

   アプリケーションがイベント・ストアにアクセスできないためにイベントを削除することができない場合、deleteEvents() メソッドは DeleteFailedException 例外をスローします。 pollForEvents() メソッドは、この例外をキャッチした場合、アプリケーションのイベント・ストアから応答が返されなかったことを示すために、APPRESPONSETIMEOUT 結果状況を戻すことがあります。

図 65 には、イベントをアーカイブするコード・フラグメントが含まれています。

```
// Archive the event if ArchiveProcessed is set to true.
if (arcProcessed.equalsIgnoreCase("true")) {
   // Archive the event in the application's archive store.
   evts.archiveEvent(evtObj.getEventID());

   // Delete the event from the event store.
   evts.deleteEvent(evtObj.getEventID());
   }
```

図 65. イベントのアーカイブ

アーカイブが完了した後で、ポーリング・メソッドは適切な戻りコードを設定します。

- イベントが正常にデリバリーされた後でアーカイブが実行された場合、戻りコードは「success」です。この状況は、SUCCEED 結果状況定数によって示されます。
- なんらかのエラー条件 (アンサブスクライブされたイベントやイベント処理中のエラーなど) が原因でアーカイブが実行された場合、ポーリング・メソッドが「fail」状況を戻す必要が生じることがあります。この状況は、FAIL 結果状況定数によって示されます。

イベント・ストア・リソースの解放

多くの場合、pollForEvents() メソッドは、イベント・ストアにアクセスするためにリソースを割り振る必要があります。これらのリソースによるメモリー使用量が多くなりすぎないようにするために、ポーリング・メソッドの終了時にリソースを解放することができます。表 103 には、イベント・ストア・リソースをリリースするために Java コネクター・ライブラリーで提供されているメソッドがリストされています。

表 103. イベント・ストア・リソースを解放するためのメソッド

Java コネクター・ライブラリー・クラス	メソッド
CWConnectorEventStore	cleanupResources()

例えば、イベント・ストアがデータベース内でイベント表として実装されている場合、pollForEvents() メソッドは、これらの表にアクセスするために SQL カーソルを割り振ることがあります。これらの SQL カーソルを解放するために、cleanupResources() メソッドを実装することができます。それにより、pollForEvents() が終了したときに cleanupResources() を呼び出して、これらのカーソルが使用していたメモリーを解放できるようになります。

注: CWConnectorEventStore クラスは、デフォルトでは cleanupResources() メソッドを実装しません。イベント・ストア・リソースを解放するためには、イベント・ストアへのアクセスに使用されたリソースを解放するように、cleanupResources() を指定変更しなければなりません。

Java pollForEvents() のデフォルト実装

図 66 に、CWConnectorAgent クラスの pollForEvents() のデフォルト実装を示します。147 ページの『pollForEvents() の基本ロジック』で説明されている基本ロジックに準拠したこのデフォルト実装を使用するか、独自の実装を使用してこのメソッドをオーバーライドするかを選択できます。

```
/**
 * Default implementation of pollForEvents.
 */
public int pollForEvents() {
   CWConnectorUtil.traceWrite(
      CWConnectorLogAndTrace.LEVEL5,"Entering pollForEvents.");

   // Get the EventStoreFactory implementation name from the
   // getEventStore() method.
   CWConnectorEventStore evts=getEventStore();
   if (evts==null)
   {
      CWConnectorUtil.generateAndLogMsg(10533,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
      return CWConnectorConstant.APPRESPONSETIMEOUT
   }
   try { //finally block
       // Fetch PollQuantity number of events from the application.
      try {
         evts.fetchEvents();
      } catch (StatusChangeFailedException e) {
         CWConnectorUtil.generateAndLogMsg(10533,
            CWConnectorLogAndTrace.XRD_ERROR,0,0);
         CWConnectorUtil.logMsg(e.getMessage());
         e.printStackTrace();
         return CWConnectorConstant.APPRESPONSETIMEOUT;
      }

   // Get the property values for PollQuantity and ArchiveProcessed.
   int pollQuantity;
   String poll=CWConnectorUtil.getConfigProp("PollQuantity");
   try {
      if (poll == null || poll.equals(""))
         pollQuantity=1;
      else
         pollQuantity=Integer.parseInt(poll);
   } catch (NumberFormatException e) {
      CWConnectorUtil.generateAndLogMsg(10544,
         CWConnectorLogAndTrace.XRD_ERROR, 0);
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.FAIL;
   }

   String arcProcessed=CWConnectorUtil.getConfigProp(
      "ArchiveProcessed");

   // In case the ArchiveProcessed property is not set, use true
   // as default.
   if (arcProcessed == null || arcProcessed.equals(""))
      arcProcessed=CWConnectorAttrType.TRUESTRING;
   CWConnectorEvent evtObj;
   CWConnectorBusObj bo=null;
```

図 66. pollForEvents() の基本ロジックの実装 (1/7)

```
try {
   for (int i=0; i < pollQuantity; i++){

      // Process each event retrieved from the application.
      // Get the next event to be processed.
      evtObj=evts.getNextEvent();

      // A null return indicates that there were no events with
      // READY_FOR_POLL status. Return SUCCESS.
      if (evtObj == null) {
         CWConnectorUtil.generateAndLogMsg(10534,
            CWConnectorLogAndTrace.XRD_INFO,0,0);
         return CWConnectorConstant.SUCCEED;
      }
      // Check if the connector has subscribed to the event
      // generated for the business object.
      boolean isSub=isSubscribed(evtObj.getBusObjName(),
         evtObj.getVerb());
      if (isSub) {
         // Retrieve the complete CWConnectorBusObj corresponding
         // to the object using the getBO method in
         // CWConnectorEventStore. This method sets the verb on a
         // temporary business object to RetrieveByContent
         // and retrieves the corresponding data information to be
         // filled in the business object from the application.
         try {
            bo = evts.getBO(evtObj);
            // Terminate flag will be set in the event store when
            // the doVerbFor method returns APPRESPONSETIMEOUT in
            // getBO.
            if (evts.getTerminate())
               return CWConnectorConstant.APPRESPONSETIMEOUT;
         }catch (AttributeNotFoundException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","AttributeNotFoundException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (SpecNameNotFoundException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","SpecNameNotFoundException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
```

図 66. pollForEvents() の基本ロジックの実装 (2/7)

```
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (InvalidVerbException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","InvalidVerbException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (WrongAttributeException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","WrongAttributeException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
         }catch (AttributeValueException e) {
            CWConnectorUtil.generateAndLogMsg(10536,
               CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
               "getBO","AttributeValueException");
            CWConnectorUtil.logMsg(e.getMessage());
            e.printStackTrace();
            // Update the event status to ERROR_PROCESSING_EVENT
            evts.updateEventStatus(evtObj,
               CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
            if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
            {
               // Archive the event in the application's archive store
               evts.archiveEvent(evtObj.getEventID());
               // Delete the event from the event store
               evts.deleteEvent(evtObj.getEventID());
            }
            continue;
```

図 66. pollForEvents() の基本ロジックの実装 (3/7)

```
      }catch (AttributeNullValueException e) {
         CWConnectorUtil.generateAndLogMsg(10536,
            CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
            "getBO","AttributeNullValueException");
         CWConnectorUtil.logMsg(e.getMessage());
         e.printStackTrace();
         // Update the event status to ERROR_PROCESSING_EVENT
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }

      // Log a fatal error in case the object is not found.
      if (evtObj.getStatus()==
   CWConnectorEventStatusConstants.ERROR_OBJECT_NOT_FOUND) {
         CWConnectorUtil.generateAndLogMsg(10543,
            CWConnectorLogAndTrace.XRD_FATAL,0,0);
         // Update the event status to ERROR_OBJECT_NOT_FOUND
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_OBJECT_NOT_FOUND);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }
      // In case the business object is null, the retrieve call
      // returned an error.
      if (bo == null) {
         CWConnectorUtil.generateAndLogMsg(10335,
            CWConnectorLogAndTrace.XRD_ERROR,0,0);
         // Update the event status to ERROR_PROCESSING_EVENT
         evts.updateEventStatus(evtObj,
            CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
         if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
         {
            // Archive the event in the application's archive store
            evts.archiveEvent(evtObj.getEventID());
            // Delete the event from the event store
            evts.deleteEvent(evtObj.getEventID());
         }
         continue;
      }
```

図 66. pollForEvents() の基本ロジックの実装 (4/7)

```
// Set the processing verb on the business object.
try {
   bo.setVerb(evtObj.getVerb());
} catch(InvalidVerbException e){
   CWConnectorUtil.generateAndLogMsg(10536,
      CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
      "setVerb","InvalidVerbException");
   CWConnectorUtil.logMsg(e.getMessage());
   e.printStackTrace();
   // Update the event status to ERROR_PROCESSING_EVENT
   evts.updateEventStatus(evtObj,
      CWConnectorEventStatusConstants.ERROR_PROCESSING_EVENT);
   if (arcProcessed.equalsIgnoreCase(CWConnectorAttrType.TRUESTRING))
   {
      // Archive the event in the application's archive store
      evts.archiveEvent(evtObj.getEventID());
      // Delete the event from the event store
      evts.deleteEvent(evtObj.getEventID());
   }
   continue;
}
// Check again for subscription.
if (isSubscribed(bo.getName(),bo.getVerb())){
   // Send the event to integration broker.
   int stat=gotApplEvent(bo);
   if (stat == CWConnectorConstant.CONNECTOR_NOT_ACTIVE){
      CWConnectorUtil.generateAndTraceMsg(
         CWConnectorLogAndTrace.LEVEL3, 10551,
         CWConnectorLogAndTrace.XRD_INFO, 0, 0);
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.READY_FOR_ROLL);
      // No need to archive the event, as the status is reset to
      // READY_FOR_POLL. It is as if this event never reached the
      // connector for processing.
      return CWConnectorConstant.SUCCEED;
   }
   if (stat == CWConnectorConstant.NO_SUBSCRIPTION_FOUND){
      CWConnectorUtil.generateAndLogMsg(10552,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
      // Update the event status to UNSUBSCRIBED.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.UNSUBSCRIBED);
      if (arcProcessed.equalsIgnoreCase(
            CWConnectorAttrType.TRUESTRING)) {
         // Archive the event in the application's archive store
         evts.archiveEvent(evtObj.getEventID());
         // Delete the event from the event store
         evts.deleteEvent(evtObj.getEventID());
      }
      continue;
   }
   if (stat == CWConnectorConstant.SUCCEED){
      // Update the event status to SUCCESS.
      evts.updateEventStatus(evtObj,
         CWConnectorEventStatusConstants.SUCCESS);
      if (arcProcessed.equalsIgnoreCase(
            CWConnectorAttrType.TRUESTRING)) {
         // Archive the event in the application's archive store
         evts.archiveEvent(evtObj.getEventID());
         // Delete the event from the event store
         evts.deleteEvent(evtObj.getEventID());
      }
      continue;
```

図 66. pollForEvents() の基本ロジックの実装 (5/7)

```
                } else // gotApplEvent returned FAIL
                {
                   CWConnectorUtil.generateAndLogMsg(10532,
                      CWConnectorLogAndTrace.XRD_ERROR,0,0);
                   // Update the event status to ERROR_POSTING_EVENT.
                   evts.updateEventStatus(evtObj,
             CWConnectorEventStatusConstants.ERROR_POSTING_EVENT);
                   // Archive the event if ArchiveProcessed is set
                   // to true.
                   if (arcProcessed.equalsIgnoreCase(
                         CWConnectorAttrType.TRUESTRING)) {
                      // Archive the event in the application's
                      // archive store.
                      evts.archiveEvent(evtObj.getEventID());
                      // Delete the event from the event store.
                      evts.deleteEvent(evtObj.getEventID());
                   }
                   return CWConnectorConstant.FAIL;
                }

                }  else // Event unsubscribed.
                {
                   CWConnectorUtil.generateAndLogMsg(10552,
                      CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
                   // Update the event status to UNSUBSCRIBED.
                   evts.updateEventStatus(evtObj,
                      CWConnectorEventStatusConstants.UNSUBSCRIBED);
                   // Archive the event if ArchiveProcessed is set
                   // to true.
                   if (arcProcessed.equalsIgnoreCase(
                         CWConnectorAttrType.TRUESTRING)) {
                      // Archive the event in the application's
                      // archive store.
                      evts.archiveEvent(evtObj.getEventID());
                      // Delete the event from the event store.
                      evts.deleteEvent(evtObj.getEventID());
                   }
                   continue;
                }

             } else
             {
                CWConnectorUtil.generateAndLogMsg(10552,
                   CWConnectorLogAndTrace.XRD_ERROR, 0, 0);
                // Update the event status to UNSUBSCRIBED.
                evts.updateEventStatus(evtObj,
                   CWConnectorEventStatusConstants.UNSUBSCRIBED);
                // Archive the event if ArchiveProcessed is set
                // to true.
                if (arcProcessed.equalsIgnoreCase(
                      CWConnectorAttrType.TRUESTRING)) {
                   // Archive the event in the application's
                   // archive store.
                   evts.archiveEvent(evtObj.getEventID());
                   // Delete the event from the event store.
                   evts.deleteEvent(evtObj.getEventID());
                }
                continue;
             }
          } //For loop
}
```

図 66. pollForEvents() の基本ロジックの実装 (6/7)

```
   } catch (StatusChangeFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "updateEventStatus","StatusChangeFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (InvalidStatusChangeException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "updateEventStatus","InvalidStatusChangeException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (ArchiveFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "archiveEvent","ArchiveFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (DeleteFailedException e){
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "deleteEvent","DeleteFailedException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.APPRESPONSETIMEOUT;
   } catch (AttributeNullValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR, 0, 2,
         "get method in event store","AttributeNullValueException");
      CWConnectorUtil.logMsg(e.getMessage());
      e.printStackTrace();
      return CWConnectorConstant.FAIL;
   }
   } finally {
      evts.cleanupResources();
   }
   return CWConnectorConstant.SUCCEED;
}
```

図 66. pollForEvents() の基本ロジックの実装 (7/7)

コネクターのシャットダウン

Java コネクター・ライブラリーでは、Java コネクターの terminate() メソッドは CWConnectorAgent クラス内で定義されます。 terminate() 内で使用される一般的な戻りコードは、SUCCEED または FAIL です。図 67 に、Java コネクターのサンプル terminate() メソッドを示します。

```
public int terminate(){

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
      "Entering Connector terminate()");

   //  disconnect from application
    boolean logoutSuccessful = userConnect.logout();

   // free any resources, logoff any cache sessions if connection
   // pool is used.

   CWConnectorUtil.traceWrite(CWConnectorLogAndTrace.LEVEL4,
   return CWConnectorConstant.SUCCEED;
}
```

図 67. Java terminate() メソッド

エラーと状況の処理

このセクションでは、コネクター・クラス・ライブラリーのメソッドがエラー条件を示す方法についての次の情報を提供します。

- 『Java 戻りコード』
- 236 ページの『例外』
- 238 ページの『戻り状況記述子』

注: エラー・ロギングとメッセージ・ロギングを使用して、コネクター内のエラー条件とメッセージを処理することもできます。詳細については、159 ページの『第 6 章 メッセージ・ロギング』を参照してください。

Java 戻りコード

Java コネクター・ライブラリーでは、CWConnectorConstant クラス内の結果状況定数が Java 戻りコードを定義します。表 104 に、これらの Java 結果状況定数を示します。

表 104. Java 結果状況コード

戻りコード	説明
CWConnectorConstant.SUCCEED	操作が成功しました。
CWConnectorConstant.FAIL	操作は失敗しました。
CWConnectorConstant.APPRESPONSETIMEOUT	アプリケーションが応答していません。
CWConnectorConstant.MULTIPLE_HITS	コネクターが非キー値を使用して取得中に複数の一致レコードを検出しました。最初のレコードがこの状況コードと共に戻されます。
CWConnectorConstant.BO_DOES_NOT_EXIST	コネクターが Retrieve 操作を実行したが、ビジネス・オブジェクトが表しているエンティティーがアプリケーション・データベース内に存在しません。
CWConnectorConstant.RETRIEVEBYCONTENT_FAILED	コネクターは、非キー値による取得で、一致レコードを検出できませんでした。
CWConnectorConstant.UNABLETOLOGIN	コネクターが、アプリケーションにログインできません。
CWConnectorConstant.VALCHANGE	ビジネス・オブジェクト内の少なくとも 1 つの値が変更されました。

表 104. Java 結果状況コード (続き)

戻りコード	説明
CWConnectorConstant.VALDUPES	アプリケーション内のオブジェクトがすでに要求されたデータ値を持っています。
CWConnectorConstant.CONNECTOR_NOT_ACTIVE	コネクターがアクティブではなく、一時停止しています。
CWConnectorConstant.NO_SUBSCRIPTION_FOUND	イベントへのサブスクリプションが検出されませんでした。

結果状況定数は、ユーザーが実装する多数の Java メソッドで使用するために提供されます (表 105 を参照)。コード内の任意のメソッドの内部からこれらの値を戻すことができますが、特定の使用方法を想定して設計された戻りコードもあります。例えば、VALCHANGE は、変更された値を持つビジネス・オブジェクトをコネクターが送信していることを、統合ブローカーに通知します。

表 105. Java コネクター・メソッドの結果状況値

コネクター・メソッド	可能な結果状況コード
archiveEvent()	SUCCEED、FAIL
doVerbFor()	SUCCEED、FAIL、APPRESPONSETIMEOUT、VALCHANGE、VALDUPES、MULTIPLE_HITS、RETRIEVEBYCONTENT_FAILED、BO_DOES_NOT_EXIST
gotApplEvent()	SUCCEED、FAIL、CONNECTOR_NOT_ACTIVE、NO_SUBSCRIPTION_FOUND
pollForEvents()	SUCCEED、FAIL、APPRESPONSETIMEOUT
terminate()	SUCCEED、FAIL

コネクター・フレームワークは、受け取った結果状況定数を使用して、次に実行するアクションを決定します。

- 結果状況が APPRESPONSETIMEOUT である場合、コネクター・フレームワークはコネクターをシャットダウンします。

  コネクター・フレームワークは、この結果状況を受け取ると、APPRESPONSETIMEOUT 状況を戻り状況記述子にコピーします。次に、この記述子を戻して、アプリケーションが応答していない ことをコネクター・コントローラーに通知します。コネクター・フレームワークは、この戻り状況記述子を送信した後で、コネクターが実行されているプロセスを停止します。システム管理者は、アプリケーションに関する問題を修正してから、コネクターを再始動してイベントとビジネス・オブジェクト要求の処理を継続する必要があります。
- その他のすべての 結果状況値の場合、コネクター・フレームワークはコネクターの実行を継続します。

  要求処理中に、コネクター・フレームワークは、結果状況を戻り状況記述子の状況フィールドにコピーし、この記述子を統合ブローカーへの応答に組み込みます。コネクター・フレームワークは、コネクターの実行を継続します。結果状況値の種類によっては、コネクター・フレームワークが応答ビジネス・オブジェクトを応答に含めることもあります。詳細については、196 ページの『要求ビジネス・オブジェクトの更新』を参照してください。

重要: コネクター・フレームワークは、FAIL 結果状況定数を受け取っても、コネクターの実行を停止しません。

例外

Java コネクター・ライブラリーのメソッドは、状況コードを戻すだけでなく、所定の事前定義条件を示すために例外をスローすることもできます。このセクションでは、Java コネクターでの例外の処理方法についての次の情報を提供します。

- 『Java コネクター例外とは』
- 237 ページの『Java コネクター・ライブラリーの例外』

Java コネクター例外とは

Java コネクター・ライブラリーのメソッドが例外をスローするときの例外オブジェクトは、Java Exception クラスを拡張した CWException クラスのサブクラスです。図 68 に示すように、この*例外オブジェクト* は、メッセージ、状況、および例外についての追加情報を含む例外詳細オブジェクトを格納しています。

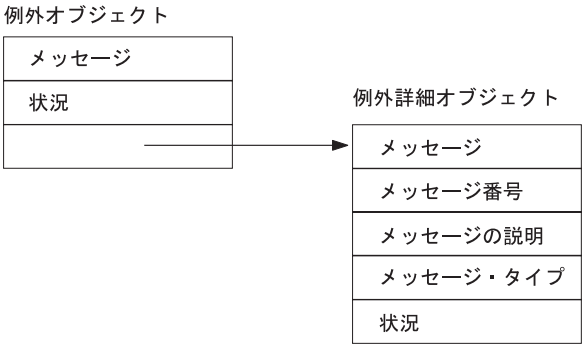

図 68. CWException 例外オブジェクト

表 106 に、例外オブジェクト内の情報を取得するために CWException クラスが提供する accessor メソッドを示します。

表 106. 例外オブジェクト内の情報

メンバー	accessor メソッド
メッセージ・テキスト	getMessage()
状況	getStatus(), setStatus()
例外詳細オブジェクト	getExceptionObject()

注: CWException クラス内のメソッドの詳細については、441 ページの『第 24 章 CWException クラス』を参照してください。

例外詳細オブジェクト は、CWConnectorExceptionObject クラスのインスタンスです。図 68 に示すように、例外オブジェクトは例外詳細オブジェクトを格納します。この例外詳細オブジェクトは、Java コネクター・ライブラリー例外についての詳細情報を提供します。表 107 を参照してください。

表 107. 例外詳細オブジェクト内の情報

メンバー	説明	accessor メソッド
メッセージ・テキスト	例外のメッセージ・テキスト	getMsg(), setMsg()

表 107. 例外詳細オブジェクト内の情報 (続き)

メンバー	説明	accessor メソッド
メッセージ番号	メッセージを識別するメッセージ・ファイル内の番号	getMsgNumber(), setMsgNumber()
メッセージの説明	メッセージの詳細説明。この説明は、メッセージ・ファイル内にも格納されます。この情報には、修正処置も含まれています。	getExpl(), setExpl()
メッセージ・タイプ	メッセージの重大度を示す整数定数	getMsgType(), setMsgType()
状況	メソッドの結果を示す整数状況	getStatus(), setStatus()

注: `CWConnectorExceptionObject` クラス内のメソッドの詳細については、383 ページの『第 19 章 CWConnectorExceptionObject クラス』を参照してください。

Java コネクター・ライブラリーの例外

Java コネクターのコードを記述するときには、Java `try` および `catch` ステートメントを組み込んで、Java コネクター・ライブラリーのメソッドがスローする個別の例外を処理できます。多くの場合、Java コネクター・ライブラリーのメソッドのリファレンス説明には、『例外』というタイトルのセクションがあり、そのメソッドがスローする例外がリストされています。

図 69 に、`getBO()` メソッドがスローする例外をキャッチする `pollForEvents()` メソッドのデフォルト実装からのコード・フラグメントを示します。

```
try {
   bo = evts.getBO(evtObj);

   }catch (AttributeNotFoundException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeNotFoundException");
      return CWConnectorConstant.FAIL;
   }catch (SpecNameNotFoundException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","SpecNameNotFoundException");
      return CWConnectorConstant.FAIL;
   }catch (InvalidVerbException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
            0,2,"getBO","InvalidVerbException");
      return CWConnectorConstant.FAIL;
   }catch (WrongAttributeException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","WrongAttributeException");
      return CWConnectorConstant.FAIL;
   }catch (AttributeValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeValueException");
      return CWConnectorConstant.FAIL;
   }catch (AttributeNullValueException e) {
      CWConnectorUtil.generateAndLogMsg(10536,
         CWConnectorLogAndTrace.XRD_ERROR,
         0,2,"getBO","AttributeNullValueException");
      return CWConnectorConstant.FAIL;
   }
```

図 69. getBO() からの例外のキャッチ

Java コネクター・ライブラリーのメソッドは、通常の場合、例外をスローするときに例外オブジェクト内でメッセージおよび状況情報を提供しません。ただし、必要に応じて、追加情報を例外オブジェクトに組み込むことができます。

戻り状況記述子

戻り状況記述子 には通常、ビジネス・オブジェクト・ハンドラー (doVerbFor() メソッド) によって実行された動詞処理の成功 (または失敗) に関する情報が含まれています。呼び出し側コードは、この状況情報を使用して、続行方法の決定を可能にしています。特定のビジネス・オブジェクト用のビジネス・オブジェクト・ハンドラーが起動されると、その関連付けられたビジネス・オブジェクト・ハンドラー・クラスの doVerbFor() が実行されます。ただし、実際に起動されるメソッドは、ユーザー実装の doVerbFor() ではありません (このメソッドは、ビジネス・オブジェクト・ハンドラー・クラスの一部として、コネクター開発者によって実装されます)。代わりに、このメソッドと同じクラスに定義されてもコネクター開発者が実装しない、下位の doVerbFor() メソッドがビジネス・オブジェクト・ハンドラーによって呼び出されます。

この下位の doVerbFor() メソッドによって、以下のタスクが実行されます。

1. 引き数として空の戻り状況記述子を受け取る。
2. ユーザー実装の doVerbFor() を呼び出して、動詞処理を実行する。

3. ユーザー実装のこの doVerbFor() が完了したら (正常または異常終了のどちらの場合にも)、動詞処理の状況に基づいて戻り状況記述子にデータを取り込む。

下位の doVerbFor() は、インスタンス化された戻り状況記述子を引き数として受け取るため、下位の doVerbFor() が終了すると、この戻り状況記述子に加えられた変更はすべて、(戻り状況記述子をインスタンス化した) 呼び出し側コードに対して有効になります。その結果、ビジネス・オブジェクト・ハンドラーを呼び出したコードは、この戻り状況記述子へのアクセスによって、動詞処理の状況に関する情報を取得できるようになります。

この戻り状況記述子へのアクセスは、以下のどちらかの方法で実行できます。

- 『戻り状況記述子への暗黙的アクセス』
- 『戻り状況記述子への明示的アクセス』

戻り状況記述子への暗黙的アクセス

要求処理中に、コネクター・フレームワークは戻り状況記述子を使用して、動詞処理の状況を統合ブローカーに通知します。コネクター・フレームワークは、要求ビジネス・オブジェクトを受け取ると、関連したビジネス・オブジェクト・ハンドラー・クラスを位置指定して、そのクラスの下位の doVerbFor() メソッドを起動します。この下位の doVerbFor() に対しては、インスタンス化された空の戻り状況記述子が渡されます。

下位の doVerbFor() は完了後に、ユーザー実装の doVerbFor() メソッドからの動詞処理の状況を戻り状況記述子に取り込みます。その後でこの戻り状況記述子は、コネクター・フレームワークによって、統合ブローカーへの応答の一部として組み込まれます。詳細については、195 ページの『戻り状況記述子の取り込み』を参照してください。

戻り状況記述子への明示的アクセス

イベント通知の際、ポーリング・メソッドは、戻り状況記述子を使用して、イベントに関連付けられたアプリケーション・データの取得が成功したかどうかの判別を可能にしています。ポーリング・メソッド pollForEvents() がイベント・ストアからイベントを取得した場合は通常、関連付けられたアプリケーション・イベントのキー値のみがイベントに含まれます。アプリケーション・データをすべて取得する場合、pollForEvents() は、キー値 (または値) を使用して、アプリケーションを照会し、値の全セットを取得しなければなりません。詳細については、216 ページの『アプリケーション・データの取得』を参照してください。

このアプリケーション・データを取得するための一般的な方法としては、ビジネス・オブジェクト内で RetrieveByContent 動詞を使用して、ビジネス・オブジェクト・ハンドラーを呼び出す方法が挙げられます。ビジネス・オブジェクト・ハンドラーをこの用途に容易に使用できるようにするために、CWConnectorBusObj クラスでは doVerbFor() メソッドのバージョンが提供されています。呼び出し側コードは、この doVerbFor() メソッドを呼び出すときに、下位の doVerbFor() メソッドを呼び出すことによって、現在のビジネス・オブジェクト用のビジネス・オブジェクト・ハンドラーを起動します。 CWConnectorBusObj バージョンの doVerbFor() の呼び出し側コードは、まず戻り状況記述子を作成してから、このインスタンス化された空の戻り状況記述子を doVerbFor() 内に渡し入れます。

空の戻り状況記述子は、CWConnectorBusObj バージョンの doVerbFor() から、ビジネス・オブジェクト・ハンドラー・クラス内の下位の doVerbFor() メソッドに渡されます。下位の doVerbFor() は完了後に、ユーザー実装の doVerbFor() メソッドからの動詞処理の状況を戻り状況記述子に取り込みます。この空の戻り状況記述子が、CWConnectorBusObj バージョンの doVerbFor() から、呼び出し側コードに戻されます。この戻り状況記述子は、呼び出し側コードによってインスタンス化されているため、その内容に明示的にアクセスすることによって、動詞処理の成功を確認できます。

Java コネクターの戻り状況記述子は、CWConnectorReturnStatusDescriptor オブジェクトです。表 108 に、この構造体が提供する状況情報を示します。

表 108. 戻り状況記述子内の情報

戻り状況記述子情報	説明	Java アクセサー・メソッド
エラー・メッセージ	エラー条件の記述を提供するストリング	getErrorString(), setErrorString()
状況	エラー条件の原因をより詳細に示す追加の状況値	getStatus(), setStatus()

イベントに関連付けられたアプリケーション・データを取得するための getBO() メソッドは、CWConnectorEventStore クラスによって提供されています。 getBO() メソッドのデフォルト実装によって、CWConnectorBusObj バージョンの doVerbFor() が呼び出され、この取得が実行されます。pollForEvents() メソッドのデフォルト実装には、getBO() への呼び出しが含まれます。そのため、pollForEvents() は、以下のいずれかの場合には、戻り状況記述子に明示的にアクセスせずに取得状況に関する情報を参照できます。

- pollForEvents() のデフォルト実装を使用する場合
- ユーザー独自の pollForEvents() メソッドで、getBO() のデフォルト実装を呼び出す場合

getBO() のデフォルト実装は、戻り状況記述子に自動的にアクセスして、取得の状況を示す値を戻します (または例外をスローします)。

注: CWConnectorReturnStatusDescriptor メソッドのメソッドを使用して、executeCollaboration() メソッド実行後の戻り状況記述子からコラボレーション状況にアクセスできます。

重要: doVerbFor() メソッドが戻り状況記述子内で設定するすべての状況コードは、コラボレーションにとって意味のある状況コードである必要があります。コラボレーション開発者とコネクター開発者は、この状況コードの意味を共有している必要があります。

第 8 章 ビジネス・インテグレーション・システムへのコネクターの追加

コネクターを IBM WebSphere Business Integration システムで稼働させる場合、リポジトリー内にそのコネクターを定義しておく必要があります。 WebSphere Business Integration Adapters 製品で提供される事前定義済みアダプターの場合、リポジトリー内に事前定義済みコネクター定義が用意されています。システム管理者は、アプリケーションを構成してコネクターの構成プロパティーを設定するだけで、そのコネクターを稼働させることができます。

ユーザーによって作成されたコネクターに IBM WebSphere Business Integration システムからアクセスできるようにするには、以下のステップを実行する必要があります。

1. リポジトリー内にコネクターの定義を作成します。
2. 複数のコネクター・コンポーネント間のメッセージングに WebSphere MQ を使用する場合は、コネクターにメッセージ・キューを付加する。
3. コネクターの初期構成ファイルを作成します。
4. コネクターの始動スクリプトを作成します。

この章では、IBM WebSphere Business Integration システムに新規コネクターを追加するための情報を提供します。この章を構成するセクションは次のとおりです。

コネクターの命名

この章では、コネクター開発で使用するファイルとディレクトリーに対して推奨する命名規則を示します。命名規則は、コネクター・ファイルをより簡単に検出および識別する方法を提供します。表 109 に、コネクター・ファイルに対して推奨する命名規則の要約を示します。これらのファイルの多くは、WebSphere Business Integration システム内で固有の識別名となるコネクター名 に基づいています。この名前 (*connName*) から、コネクターが通信するアプリケーションまたはテクノロジーを識別できます。

表 109. コネクターに対して推奨する命名規則

コネクター・ファイル	名前
コネクター定義	*connName*Connector
コネクター・ディレクトリー	*ProductDir*¥connectors¥*connName*

表 109. コネクターに対して推奨する命名規則 (続き)

コネクター・ファイル	名前
初期コネクター構成ファイル	ファイル名: BIA_CN_*connName*.txt ディレクトリー名: *ProductDir*¥repository¥*connName*
ユーザーがカスタマイズしたコネクター構成ファイル	ファイル名: CN_*connName*.txt ディレクトリー名: *ProductDir*¥connectors¥*connName*
コネクター・クラス	*connName*Agent.java
コネクター・ライブラリー	Java jar ファイル: *connDir*¥BIA_*connName*.jar Java パッケージ: com.crossworlds.connectors.*connName* ここで、*connDir* は、上記で定義されているコネクター・ディレクトリーの名前です。
コネクター始動スクリプト	Windows プラットフォーム: *connDir*¥start_*connName*.bat UNIX ベースのプラットフォーム: *connDir*¥connector_manager_*connName*.sh ここで、*connDir* は、上記で定義されているコネクター・ディレクトリーの名前です。

コネクターの命名規則について詳しくは、IBM WebSphere InterChange Server ドキュメンテーション・セットに同梱された「*IBM WebSphere InterChange Server コンポーネント命名ガイド*」を参照してください。

コネクターのコンパイル

コネクターのアプリケーション固有コンポーネントを記述したら、実行可能なフォーマット、コネクター・ライブラリー にコンパイルしておく必要があります。コネクターのコンパイルの方法については、この章で説明します。

Java コネクターをコンパイルするには、以下のステップを実行します。

- JDK 1.3.1 開発環境を使用します。詳細については、 35 ページの『開発環境のセットアップ』を参照してください。
- 次の両方のファイルが製品ディレクトリーの `lib` サブディレクトリーに入っていることを確認します。
  - `crossworlds.jar`
  - `WBIA.jar`
- プロジェクト・パス内に `crossworlds.jar` をインクルードします。プロジェクト・パス内には、コネクターのアプリケーション固有コンポーネントに必要なアプリケーション固有の jar ファイルもインクルードしてください。
- Java コンパイラーを使用してコネクターのソース (.java) ファイルをコンパイルして、クラス (.class) ファイルを生成します。
- Java コネクターのライブラリー・ファイルを作成します。このファイルは、コンパイル済みの Java コードを収める Java アーカイブ (jar) ファイルです。

jar ファイルに対して推奨する命名規則は、名前をストリング「BIA_」で始めることです。このストリングの後に、コネクターを一意的に識別するコネクター名を続けてください (241 ページの表 109 を参照してください)。コネクター名の詳細については、 241 ページの『コネクターの命名』を参照してください。

例えば、Java コネクターのコネクター名が MyJava である場合、次のような jar ファイル名を付けることができます。

```
BIA_MyJava.jar
```

コネクター定義の作成

コネクターを IBM WebSphere Business Integration システムで稼働させる場合、リポジトリー 内にそのコネクターを定義しておく必要があります。 WebSphere Business Integration Adapters 製品で提供される事前定義済みアダプターの場合、インストール時にリポジトリー内にロードされる、事前定義済みコネクター定義が用意されています。システム管理者は、アプリケーションを構成してコネクターの構成プロパティーを設定するだけで、そのコネクターを稼働させることができます。ただし、IBM WebSphere Business Integration システムから作成済みコネクターへのアクセスを可能にするには、以下のステップを実行しておく必要があります。

- リポジトリー内でコネクターを定義するために、コネクター定義を作成します。
- ユーザーのコネクター構成を支援するために、初期構成ファイルを作成します (オプション)。

コネクターの定義

WebSphere Business Integration システム内でコネクターを定義するために、コネクター定義 を作成します。このコネクター定義には、リポジトリー内でコネクターを定義するために必要な、以下の情報が含まれます。

- コネクター定義の名前
- サポートされるビジネス・オブジェクトおよび関連付けられているマップ
- コネクター構成プロパティー

Connector Configurator というツールでこの情報を収集し、リポジトリーに保管します。

WebSphere InterChange Server

統合ブローカーが InterChange Server である場合、このリポジトリーは、WebSphere Business Integration システム内コンポーネントの情報を取得するために InterChange Server が通信を行う相手となるデータベースです。このリポジトリーにはコネクター定義が収められます。これらのコネクター定義には、コネクター・コントローラーとクライアントのコネクター・フレームワークが必要とする、標準コネクター構成プロパティーとコネクター固有のコネクター構成プロパティーの両方が含まれます。また、コネクターには、そのコネクターに関するローカルな構成情報を含む、ローカル構成ファイルを割り当てることもできます。ローカル構成ファイルが存在する場合には、InterChange Server リポジトリー内の情報よりもローカル構成ファイル内の情報のほうが優先します。

InterChange Server リポジトリー内のコネクター定義を更新するためには、System Manager ツールに含まれる Connector Configurator を使用します。ロケール構成ファイルの更新は、スタンドアロン・バージョンの Connector Configurator を使用して行うことができます。このツールは、製品ディレクトリー内の `bin` サブディレクトリーに入っています。

WebSphere MQ Integrator Broker

統合ブローカーが WebSphere MQ Integrator Broker である場合、このリポジトリーは、WebSphere Business Integration システムのコンポーネントに関する情報を取得するためにコネクター・フレームワークが使用するファイルのディレクトリーです。このリポジトリーには、システム内のそれぞれのアダプターのコネクター定義が収められます。

ローカル・リポジトリー内のコネクター定義の更新は、Connector Configurator を使用して行います。このツールは、製品ディレクトリー内の `bin` サブディレクトリーに入っています。

Connector Configurator の使用方法の詳細については、 569 ページの『付録 B. Connector Configurator』を参照してください。

コネクター定義名

コネクター定義名は、WebSphere Business Integration システム内にあるコネクターを一意的に識別します。規則により、コネクター定義名は、通常は次の形式になっています。

*connName*Connector

ここで、*connName* はコネクター名です (241 ページの表 109 を参照してください)。コネクター名の詳細については、 241 ページの『コネクターの命名』を参照してください。例えば、コネクター名が `MyConn` である場合、コネクター定義の名前は `MyConnConnector` です。

サポートされるビジネス・オブジェクトおよびマップ

コネクター定義では、そのコネクターがサポートするビジネス・オブジェクトについて、以下の情報を指定しなければなりません。

- ビジネス・オブジェクトの定義

  コネクターが統合ブローカーとの間で送受信できるそれぞれのビジネス・オブジェクトを、サポートされるビジネス・オブジェクトとして指定する必要があります。Connector Configurator が提供する「サポートされるビジネス・オブジェクト (Supported Business Objects)」タブで、そのコネクターのサポートされるビジネス・オブジェクトを指定することができます。

  注: コネクターがサポートしているアプリケーション固有のビジネス・オブジェクトをすべて、リポジトリー内に定義しておいてください。その後ではじめて、それらのビジネス・オブジェクトをサポートされているビジネス・オブジェクトとして、コネクター定義の中にインクルードできるようになります。アプリケーション固有のビジネス・オブジェクトを定義する方法については、「*ビジネス・オブジェクト開発ガイド*」を参照してください。

- 関連付けられたマップ

WebSphere InterChange Server

コネクターと関連付けられたマップは、InterChange Server との間でその統合ブローカーとして通信するコネクターに関するコネクター定義にのみ含まれます。関連付けられたマップとは、そのコネクターのアプリケーション固有ビジネス・オブジェクトと、該当する汎用ビジネス・オブジェクトとの間の変換を行うマップのことを言います。

Connector Configurator が提供する「関連付けられたマップ」タブで、そのコネクターの関連付けられたマップを指定することができます。

コネクター構成プロパティー

コネクター定義には、コネクター構成プロパティーも含まれます。これらのプロパティーを初期化するためには、以下のステップを実行する必要があります。

- 標準のコネクター構成プロパティーの値を割り当てます。
- コネクターが使用するコネクター固有の構成プロパティーを定義し、それらのプロパティーに適切な値を割り当てます。

Connector Configurator には、コネクター構成プロパティーを指定するために、「標準のプロパティー」と「コネクター固有プロパティー」の、2 つのタブが用意されています。コネクター構成プロパティーの詳細については、79 ページの『コネクター構成プロパティー値の使用』を参照してください。

初期構成ファイルの作成

慣例により、事前定義されたアダプターからは、ユーザーがそのアダプターを Connector Configurator で最初に構成するときに使用するために、初期構成ファイルが提供されるはずです。この構成ファイルには、次の名前を付けることを推奨します。

BIA_CN_*connName*.txt

ここで、*connName* はコネクター名です (241 ページの表 109 を参照してください)。コネクター名の詳細については、 241 ページの『コネクターの命名』を参照してください。この初期構成ファイルは、以下のディレクトリーにあります。

ProductDir¥repository¥*connName*

つまり製品ディレクトリーの repository サブディレクトリーに、それぞれのコネクターのディレクトリーが含まれています。それぞれのコネクターのディレクトリー (*connName*) には、固有のコネクター名が付けられ、そして、以下の名前の初期構成ファイルが入っています。

作成したコネクターをユーザーが構成できるようにするために、新規コネクター用の初期構成ファイルを用意することができます。コネクター作成作業の一環として、標準構成プロパティーの設定と、コネクター固有の構成プロパティーの定義を行っているものと思われます。このコネクター構成情報が、リポジトリーに入っているはずです。ただし、そのコネクターは、別の環境に移動すると、このリポジトリーにアクセスできなくなります。したがって、コネクターをリリースする際には、初期構成ファイルを作成する必要があります。

この初期構成ファイルを作成するには、コネクターの Connector Configurator を起動し、以下のファイルにその構成を保管してください。

ProductDir¥repository¥*connName*¥BIA_CN_*connName*.txt

注: 上記のステップでは、コネクターを作成した際に、すでにそのコネクター用のコネクター構成ファイル (.cfg) が作成してあることを想定しています。上のステップで行っているのは、このコネクター構成情報を、コネクターのリリースの一環として作られた、別のファイルに保管する作業です。

新規コネクターの始動

コネクターを始動するには、コネクター*始動スクリプト* を実行します。表 110 に示すように、この始動スクリプトの名前は、使用しているオペレーティング・システムによって異なります。

表 110. コネクターの始動スクリプト

オペレーティング・システム	始動スクリプト
UNIX 系システム	connector_manager_*connName*
Windows	start_*connName*.bat

始動スクリプトは、WebSphere Business Integration Adapters 製品が提供しているこれらのアダプターをサポートしています。事前定義済みコネクターを始動する場

合、システム管理者は、この始動スクリプトを実行します。ほとんどの事前定義コネクターの始動スクリプトは、次のコマンド行引き数が必要です。

1. 最初の引き数は、以下のものを識別するコネクター名です。
   - 製品ディレクトリーの connectors サブディレクトリーの下にあるコネクター・ディレクトリーの名前
   - コネクターのディレクトリー内にあるコネクター・ライブラリー
2. 2 番目の引き数は、コネクターの実行対象である統合ブローカーのインスタンスの名前です。

WebSphere InterChange Server

統合ブローカーが InterChange Server (ICS) である場合、コネクターの実行対象である ICS のインスタンスの名前を始動スクリプトで指定します。Windows システムの場合、この ICS インスタンス名 (これは、インストール・プロセスで指定されています) は、始動スクリプトのそれぞれのコネクター・ショートカットに表示されます。

その他の統合ブローカー

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server である場合、コネクターの実行対象であるブローカー・インスタンスの名前を始動スクリプトで指定します。Windows システムの場合、このインスタンス名 (これは、インストール・プロセスで指定されています) は、始動スクリプトのそれぞれのコネクター・ショートカットに表示されます。

3. オプションとして、コマンド行に追加の始動パラメーターを指定して、コネクター・ランタイムに引き渡すことができます。

   始動パラメーターの詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」または WebSphere Business Integration Adapters ドキュメンテーション・セット内のインプリメンテーション・ガイド (ご使用の統合ブローカー用のもの) を参照してください。

WebSphere InterChange Server

コネクターが初期化を完了してビジネス・オブジェクトをリポジトリーから取得できるようにするためには、コネクターを始動する前に InterChange Server を稼働させておく必要があります。

作成したコネクターを始動する前に、始動スクリプトがその新規コネクターをサポートしているかどうかを確認する必要があります。始動スクリプトによるユーザー独自のコネクターの始動を可能にするには、以下のステップを実行する必要があります。

1. コネクター用のコネクター・ディレクトリーを作成します。
2. コネクター用の始動スクリプトを作成します。Windows システムでは、コネクター始動用のショートカットも作成してください。
3. 始動スクリプトを Windows サービスとしてセットアップします (オプション)。

これらの各ステップについては、以降のセクションで詳しく説明します。

コネクター・ディレクトリーの作成

コネクター用のランタイム・ファイルは、コネクター・ディレクトリー に格納されます。コネクター・ディレクトリーを作成するには、以下のステップを実行します。

1. 新規コネクター用のコネクター・ディレクトリーを、次のように、製品ディレクトリーの connectors サブディレクトリーの下に作成します。

   ProductDir¥connectors¥*connName*

   命名規則に従うと、このディレクトリー名はコネクター名 (*connName*) と同じになります。コネクター名は、コネクターを一意的に識別するストリングです。詳細については、241 ページの『コネクターの命名』を参照してください。
2. コネクターのライブラリー・ファイルをこのコネクター・ディレクトリーに移動します。

   Java コネクターのライブラリー・ファイルは Java アーカイブ (jar) ファイルです。この jar ファイルは、コネクターをコンパイルしたときに作成されたものです。詳細については、242 ページの『コネクターのコンパイル』を参照してください。

始動スクリプトの作成

246 ページの表 110 に示すように、システム管理者がコネクター・プロセスの実行を開始するには、コネクター用の始動スクリプトが必要です。使用する始動スクリプトは、コネクターを作成するオペレーティング・システムに応じて異なります。

Windows システムでの始動スクリプトおよびショートカット

Windows システムでコネクターを始動するには、次のステップを実行します。

1. コネクターの始動スクリプト start_*connName*.bat を呼び出します。

   start_*connName*.bat スクリプト (ここで、*connName* は使用しているコネクターの名前) は、コネクター固有の始動スクリプトです。これは、コネクター固有の情報 (アプリケーション固有のライブラリーやロケーションなど) を提供します。規則によると、このスクリプトはコネクター・ディレクトリー内にあります。

   ProductDir¥connectors¥*connName*

   ユーザーが Windows システムでコネクターを始動する際に呼び出すのは、この start_*connName*.bat スクリプトです。
2. 汎用コネクター起動スクリプト start_adapter.bat を呼び出します。

start_adapter.bat ファイルは、すべてのコネクターに共通です。これは、JVM 内でコネクターを実際に起動します。製品ライブラリーの bin サブディレクトリーにあります。start_*connName*.bat スクリプトは、コネクターを実際に起動するために start_adapter.bat スクリプトを呼び出す必要があります。

図 70 に、Windows システムでコネクターを始動する手順を示します。

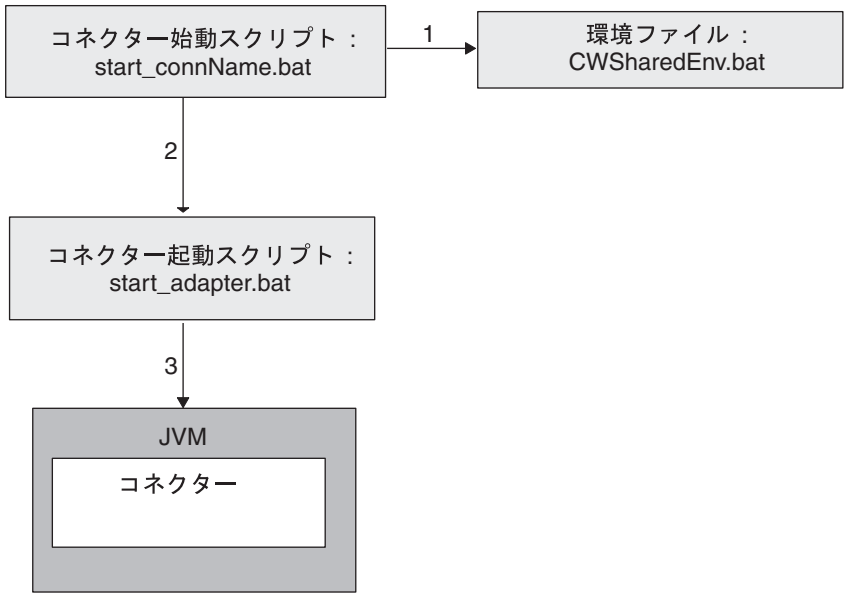

図 70. Windows システムでのコネクターの始動

Window システムでは、WebSphere Business Integration Adapters のインストーラーがコネクターをインストールするときに、以下のステップが実行されます。

- 事前定義されたコネクターの始動スクリプトをインストールします。
- 「プログラム」>「IBM WebSphere Business Integration Adapters」> 「アダプター」>「コネクター」メニューの下に、事前定義されたコネクター用のメニュー・オプションを作成します。

独自のコネクターを始動できるようにするには、これらのステップを繰り返します。

- start_*connName*.bat 始動スクリプトを生成し、それを製品ディレクトリーの connector¥*connName* サブディレクトリーに置きます。
- 「プログラム」>「IBM WebSphere Business Integration Adapters」> 「アダプター」>「コネクター」メニューの下に、コネクター用のメニュー・オプションを作成します。各メニュー・オプションは、特定のコネクター用の Windows 始動スクリプトである start_*connName*.bat を起動するショートカットになっています。

始動スクリプトの作成: カスタム・コネクター始動スクリプトを作成するには、start_*connName*.bat (ここで、*connName* は使用している Java コネクター名) という新規のコネクター固有始動スクリプトを作成します。例えば、Java コネクターの

コネクター名が MyJava である場合、始動スクリプト名は start_MyJava.bat になります。最初に、次のファイル内にある始動スクリプト・テンプレートをコピーできます。

ProductDir¥templates¥start_connName.bat

図 71. に、Windows の始動スクリプト・テンプレートの内容のサンプルを示します。最新の内容については、ご使用の製品でリリースされているバージョンのファイルを参照してください。

```
REM A sample of start_connName.bat which calls start_adapter.bat
@echo off

call "%ADAPTER_RUNTIME%"¥bin¥wbia_connEnv.bat
setlocal

REM If required, goto the connector specific directory. CONNDIR is defined
RED by caller
cd /d %CONNDIR%

REM set variables that need to pass to start_adapter.bat
REM set JVMArgs=
REM set JCLASSES=
REM set LibPath=
REM set ExtDirs=

REM A sample to start a C++ connector
REM call start_adapter.bat -nconnName -sServerName -dconnectorDLLfile -f...
REM -p... -c... ...

REM A sample to start a Java connector
call start_adapter.bat -nconnName -sserverName -lconnectorSpecificClasses
-f... -p... -c... ...

endlocal
```

図 71. Windows の始動スクリプト・テンプレートの内容のサンプル

規則に従うと、start_*connName*.bat スクリプトの構文は、図 72 に示されている標準構文になります。この場合の *connName* はコネクターの名前、*ICSinstance* は InterChange Server インスタンスの名前です。また、*additionalOptions* は、コネクター起動に渡すための追加の始動パラメーターを指定します。これらのオプションには、-c、-f、-t、および -x があります。詳細については、253 ページの表 112を参照してください。

```
start_connName connName ICSinstance additionalOptions
```

図 72. Windows コネクター始動スクリプトの標準構文

コネクター開発者は、start_*connName*.bat の内容を制御します。したがって、開発者はコネクター始動スクリプトの構文を変更できます。ただし、この標準構文を変更する場合は、start_adapter.bat に必要なすべての情報が、起動時に start_*connName*.bat 内で使用可能になっているようにしてください。

注: 図 72 の start_*connName*.bat 構文の場合、*connName* 引き数および *ICSinstance* 引き数が必要です。*additionalOptions* 引き数はオプションです。

標準構文の始動スクリプトは、コネクター名 (*connName*) に基づき、コネクターのランタイム・ファイルについて次を想定しています。

- 製品ディレクトリーの connectors サブディレクトリーの下にあるコネクター・ディレクトリーのコネクター名
- コネクター名は、コネクター・ディレクトリーにある Java コネクターのライブラリー・ファイル (その jar ファイル、CW*connName*.jar) と同じ

例えば、MyJava コネクターがこれらの前提事項を満たしている場合、そのランタイム・ファイルは *ProductDir*¥connectors¥MyJava ディレクトリーにあり、その jar ファイル は BIA_MyJava.jar という名前でそのディレクトリー内にある必要があります。コネクターでこれらの前提事項を満たせない場合は、始動スクリプトをカスタマイズして、汎用コネクター始動スクリプト start_adapter.bat に適切な情報を提供する必要があります。

このstart_*connName*.bat ファイルで、次のステップを実行します。

1. CWConnEnv.bat 環境ファイルを呼び出して、始動環境を初期化します。
2. コネクター・ディレクトリーに移動します。
3. コネクター固有の情報とコネクター固有の変数を使用して、始動スクリプト内で始動環境変数を設定します。
4. start_adapter.bat スクリプトを呼び出してコネクターを起動します。

これらの各ステップについては、以降のセクションで詳しく説明します。

環境ファイルの呼び出し: CWConnEnv.bat ファイルには、IBM Java Object Request Broker (ORB) および IBM Java Runtime Environment (JRE) の環境設定が含まれています。始動スクリプトでは、次の行によってこの環境ファイルが呼び出されます。

```
call "%ADAPTER_RUNTIME%"¥bin¥CWConnEnv
```

コネクター・ディレクトリーへの移動: start_*connName*.bat スクリプトは、start_adapter.bat スクリプトを呼び出す*前* にコネクター・ディレクトリーに変更する必要があります。コネクター・ディレクトリーには、コネクターの始動に必要なコネクター固有の始動スクリプトとその他のファイルが含まれています。このコネクター・ディレクトリーの名前は、任意の方法で定義できます。ただし、 248 ページの『コネクター・ディレクトリーの作成』で説明しているように、規則では、コネクターのディレクトリー名はコネクター名と同じです。

start_*connName*.bat スクリプトが標準構文を使用している場合 (250 ページの図 72 を参照)、コネクター名は最初の引き数 (%1) で渡されます。この場合、次の行がコネクター・ディレクトリーに移動します。

```
REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%¥connectors¥%1

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

コネクター名はいくつかのコネクター・コンポーネントで使用するので、代替方法として、このコネクター名を指定する環境変数を定義し、start_*connName*.bat スクリプト内で後続のコネクター名のすべての使用に対してこの環境変数を評価することもできます。コネクター名とコネクター・ディレクトリーの環境変数を設定する行は、次のようになります。

```
REM set the name of the connector
set CONNAME=%1

REM set the directory where the specific connector resides
set CONNDIR=%CROSSWORLDS%¥connectors¥%CONNAME%

REM goto the connector specific drive & directory
cd /d %CONNDIR%
```

環境変数の設定: start_*connName*.bat スクリプトでは、表 111 に示す環境変数が指定するコネクター固有の情報をすべて指定する必要があります。

表 111. コネクター始動スクリプト内の環境変数

変数名	値
ExtDirs	アプリケーション固有の jar ファイルのロケーションを指定します。
JCLASSES	アプリケーション固有の jar ファイルをすべて指定します。jar ファイルは、セミコロン (;) で区切って指定します。
JVMArgs	Java 仮想マシン (JVM) に渡す引き数をすべて追加します。
LibPath	アプリケーション固有のライブラリー・パスをすべて指定します。

start_adapter.bat ファイルは、表 111 の情報を次のように使用します。

- JCLASSES および LibPath 環境変数を、コネクター・フレームワーク内の適切な変数に付加します。
- ExtDirs 環境変数で外部ディレクトリー (java.ext.dirs) を設定します。
- JVM に渡す引き数のリスト内に JVMArgs 環境変数を組み込みます。

表 111 の環境変数に加えて、独自のコネクター固有の環境変数を定義することもできます。このような変数は、リリースごとに異なる可能性のある情報に役立ちます。このようにすると、変数をこのリリースに適した値に設定して、始動スクリプトの適切な行に組み込むことができます。将来情報が変更されても、この情報を使用しているすべてのコマンド行を探す必要はなく、変数の値を変更するだけで済みます。この情報を使用しているすべての行を探す必要はありません。

コネクターの起動: start_*connName*.bat スクリプトは、JVM 内でコネクターを実際に起動するために start_adapter.bat スクリプトを呼び出す必要があります。start_adapter.bat スクリプトは、*始動パラメーター* で、コネクターの実行時に必要な環境 (コネクター・フレームワークを含む) を初期化するための情報を提供します。したがって、適切な始動パラメーターを start_adapter.bat に提供する必要があります。表 112 に、start_adapter.bat スクリプトによって認識される始動パラメーターを示します。

表 112. *start_adapter.bat* スクリプトの始動パラメーター

始動パラメーター	説明	必須ですか?	**start_connName.bat** に対する追加のコマンド行オプションとして有効であるかどうか
-c*configFile*	コネクターの構成ファイルの絶対パス名。	統合ブローカーが ICS 以外の場合に必要となります。	はい
-d*dllName*	C++ コネクターのライブラリー・ファイル (*dllName*) を作成します。このファイルは、ダイナミック・リンク・ライブラリー (DLL) です。この DLL の名前に、`.dll` ファイル拡張子を使用することはできません。	すべての C++ コネクターに必要です。	なし
-f*pollFrequency*	ポーリング・アクション間の時間の長さです。指定可能な *pollFrequency* 値は次のとおりです。 • ポーリング・アクション間のミリ秒数。 • `key`。コネクターの始動ウィンドウで文字 p が入力された場合にのみ、コネクターがポーリングを実行するようにします。`key` オプションは、小文字で指定する必要があります。 • `no`。コネクターがポーリングを実行しないようにします。`no` オプションは、小文字で指定する必要があります。	なし デフォルトは 1000 ミリ秒です。	はい
-j	コネクターが Java で記述されていることを示します。	Java コネクターに `-l` オプションを指定する場合は不要です。	なし
-l*classname*	Java コネクターのコネクター・クラス (*className*) の名前。	すべての Java コネクターに必要です。	なし
-n*connectorName*	始動するコネクター (*connectorName*) の名前。	はい	なし
-s*brokerName*	コネクターが接続する先の統合ブローカー (*brokerName*) の名前。	はい	なし
-t	コネクター・プロパティー `SingleThreadAppCalls` のオンとオフを切り替えるブール値です。コネクター・フレームワークがアプリケーション固有のコンポーネントに合わせて作成するすべての呼び出しが 1 つの呼び出しトリガー・フローで実行されることを保証します。デフォルト値は `false` です。	なし	はい
-x*connectorProps*	アプリケーション固有のコネクター・プロパティーの値を初期化します。次のフォーマットを使用して、各プロパティーを指定してください。 *propName=value*	なし	はい

start_adapter.bat の呼び出しに次の始動パラメーターが含まれることを確認します。

- 必須の始動パラメーターすべて
  - コネクター定義の名前を指定する場合: -n

    コネクターの名前が最初の引き数 (%1) として start_*connName*.bat スクリプト (250 ページの図 72 を参照) に渡される場合、-n 始動パラメーターは次のように指定できます。

    ```
    -n%1Connector
    ```

    コネクター名 (CONNAME など) の環境変数を定義すると、この -n パラメーターは次のように表示されます。

    ```
    -n%CONNAME%Connector
    ```

  - InterChange Server インスタンスの名前を指定する場合: -s

    ICS インスタンスの名前が 2 番目の引き数 (%2) として start_*connName*.bat スクリプト (250 ページの図 72 を参照) に渡される場合、-n 始動パラメーターは次のように指定できます。

    ```
    -s%2
    ```

その他の統合ブローカー

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、WebSphere Integration Message Broker) または WebSphere Application Server である場合、-c オプションも必須始動パラメーターとなります。

- Java コネクターに必要な言語固有の始動パラメーター

  コネクター固有のクラス (またはパッケージ) を指定する場合: -l

  例えば、推奨される命名規則に従うと、Java コネクター名 MyJava の言語固有のパラメーターは次のようになります。

  ```
  -lcom.crossworlds.connectors.MyJava.MyJavaAgent
  ```

  コネクター名 (CONNAME など) の環境変数を定義する場合、この -l パラメーターは次のようになります。

  ```
  -lcom.crossworlds.connectors.%CONNAME%.%CONNAME%Agent
  ```

- コネクターのすべての起動時に適用されるオプションの始動パラメーター。オプションの始動パラメーター・リストについては、253 ページの表 112 を参照してください。

start_adapter.bat への呼び出しの構文のフォーマットは次のようになります。

```
call start_adapter.bat -nconnName -sICSinstance languageSpecificParams
 -cCN_connNameConnector.cfg
 -...
```

例えば、次の行は、MyJava コネクターを呼び出します。

```
call start_adapter.bat -lcom.crossworlds.connectors.MyJava.MyJavaAgent
    -nMyJava -sICSserver -cMyJavaConnector.cfg  -...
```

注: 上記のコマンド行は、コネクターが、ICSserver という名前の InterChange Server インスタンスに対して実行されていることを想定しています。コネクターが WebSphere MQ Integrator Broker または WebSphere Message Broker のインスタンスに対して実行されている場合は、そのインスタンス名がコマンド行に表示される必要があります。

CONNAME 環境変数を使用してコネクター名を保持した場合、この呼び出しは一般的に次のようになります。

```
call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
 -cCN_%CONNAME%Connector.cfg
 -...
```

start_adapter.bat への呼び出しでは、次の点に留意してください。

- コネクター・ランタイムを起動する行全体は、始動スクリプト内での *1 行に収めて* 入力します。つまり、サンプル始動行で示されている改行個所で改行しないようにしてください。
- start_adapter.bat への呼び出しでリストするパラメーターの順序は、重要では *ありません*。
- ユーザーが start_*connName*.bat の呼び出しに渡す可能性があるすべての追加オプションを start_adapter.bat の呼び出しで処理しなければならない場合もあります。この場合は、start_adapter.bat に渡すための「追加の」引き数を指定して、追加のオプションが実際のコネクター起動に渡されるようにする必要があります。例えば、start_adapter.bat の呼び出しでは、次の 3 つの追加コマンド行オプションが処理されます。

  ```
  call start_adapter.bat -n%CONNAME% -s%2 languageSpecificParams
   -cCN_%CONNAME%Connector.cfg %3 %4 %5
  ```

ショートカットの作成: ショートカットがあると、「プログラム」>「IBM WebSphere Business Integration Adapters」>「アダプター」>「コネクター」というメニュー・オプションからコネクターを開始することができます。ショートカットでは、start_*connName*.bat スクリプトへの呼び出しをリストする必要があります。このスクリプトが標準構文 (250 ページの図 72 を参照) を使用する場合、ショートカットの形式は次のようになります。

```
ProductDir¥connectors¥start_connName connName ICSinstance
```

start_*connName*.bat スクリプトに対して独自の構文を定義する場合は、ショートカットでそのカスタム構文が使用されることを確認する必要があります。

start_*connName*.bat 始動スクリプトを使用する Java コネクターのショートカットがメニューにすでに含まれている場合は、ショートカットを簡単に作成する方法として、この既存のコネクターのショートカットをコピーし、そのショートカット・プロパティーを編集してコネクター名を変更するか、または別の始動パラメーターを追加することができます。

例えば、始動スクリプトの標準構文を使用する MyJava コネクターの場合は、次のようなショートカットを作成できます。

```
ProductDir¥bin¥start_MyJava.bat MyJava ICSinstance
```

注: 上記のコマンド行は、コネクターが、ICSinstance という名前の InterChange Server インスタンスに対して実行されていることを想定しています。コネクターが WebSphere MQ Integrator Broker インスタンスに対して実行されている場合、そのインスタンス名がショートカットのコマンド行に表示されます。

UNIX システムでの始動スクリプト

UNIX ベースのシステムでコネクターを始動するには、次のステップを実行します。

1. -start オプションを指定して、コネクターの始動スクリプト connector_manager_*connName* を呼び出します。

   connector_manager_*connName* スクリプト (ここで、*connName* は使用しているコネクターの名前) は、コネクター固有の始動スクリプトです。これは、コネクターの名前を識別し、このコネクターで実行するアクションに対して、-start や -stop などのオプションのいずれかを指定します。このスクリプトは、Connector Script Generator ツールで生成されます。生成されると、スクリプトは製品ライブラリーの bin サブディレクトリーに置かれます。ユーザーが UNIX システムでコネクターを始動する際に呼び出すのは、connector_manager_*connName*.bat スクリプトです。

2. 汎用コネクター・マネージャー・スクリプト connector_manager を呼び出します。

   connector_manager ファイルは、すべてのコネクターに共通です。これは、コネクター固有の起動スクリプト start_*connName*.sh への呼び出しを生成します。これは、JVM 内でコネクターを実際に起動します。製品ライブラリーの bin サブディレクトリーにあります。 connector_manager_*connName* スクリプトは connector_manager スクリプトを呼び出します。

3. これは、コネクター固有の起動スクリプト start_*connName*.sh を呼び出します。

   start_*connName*.bat スクリプトは、コネクター固有の情報 (アプリケーション固有のライブラリーやロケーションなど) を提供します。規則によると、このスクリプトはコネクター・ディレクトリー内にあります。

   ProductDir/connectors/*connName*

   connector_manager スクリプトは、コネクター起動用のコネクター固有の情報を実際に準備するために、start_*connName*.sh スクリプトを呼び出します。

4. 汎用コネクター起動スクリプト start_adapter.sh を呼び出します。

   start_adapter.sh ファイルは、すべてのコネクターに共通です。これは、JVM 内でコネクターを実際に起動します。製品ライブラリーの bin サブディレクトリーにあります。start_*connName*.sh スクリプトは、コネクターを実際に起動するために start_adapter.sh スクリプトを呼び出す必要があります。

図 73 に、UNIX ベースのシステムでコネクターを始動するステップを示します。

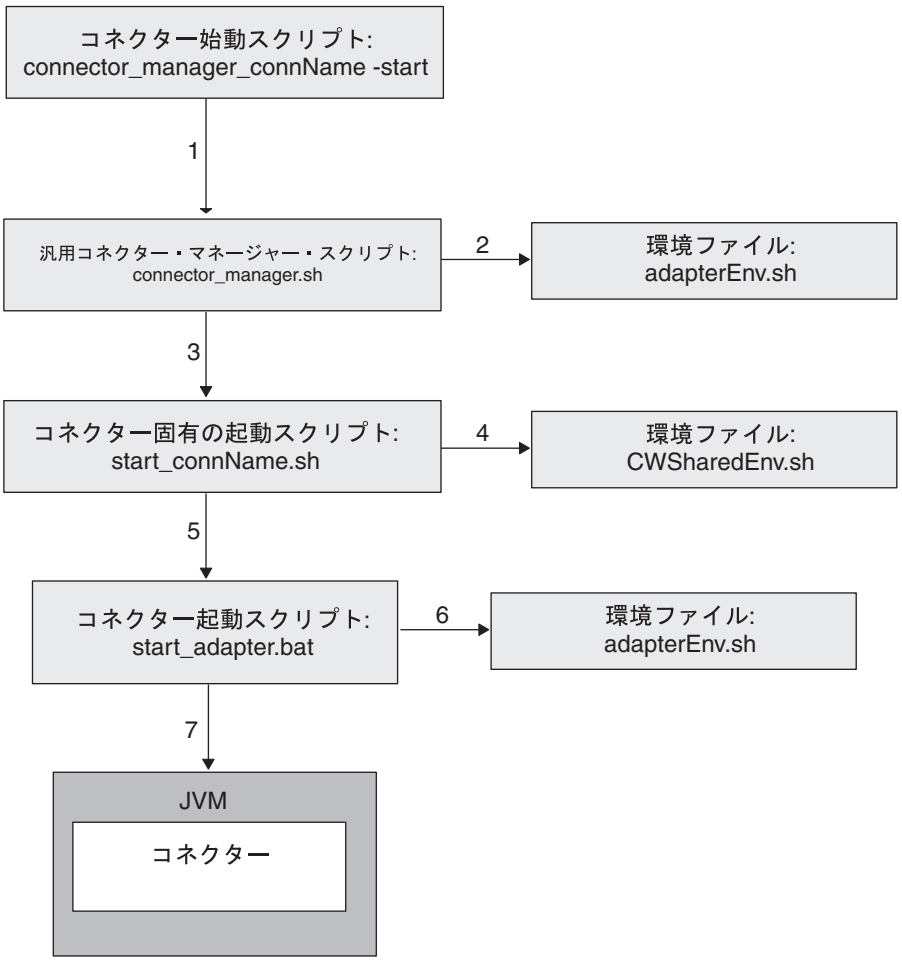

図 73. UNIX ベースのシステムでのコネクターの始動

UNIX ベースのシステムでは、WebSphere Business Integration Adapters のインストーラーがコネクターをインストールするときに、以下のステップが実行されます。

- 製品ディレクトリーの `bin` サブディレクトリーに、汎用 `connector_manager` スクリプトと汎用 `start_adapter.sh` コネクター起動スクリプトをインストールします。
- `start_`*`connName`*`.sh` スクリプトを製品ディレクトリーの `connectors/`*`connName`* サブディレクトリーにインストールします。
- `connector_manager_`*`connName`* 始動スクリプトを生成します。このスクリプトは、汎用 `connector_manager` スクリプトのラッパーです。この汎用スクリプトが、該当する `start_`*`connName`*`.sh` スクリプトを呼び出し、実際のコネクター起動を開始させます。
- 新規の `connector_manager_`*`connName`* スクリプトを `bin` 製品サブディレクトリーにインストールします。

  該当するコマンド行引き数 (例えば、ローカル構成ファイルやスレッド・タイプなど) を指定すると、`connector_manager_`*`connName`* スクリプトから `connector_manager` スクリプトが呼び出されます。

この一連のステップには、*汎用でない* 2 つのスクリプトがあります。つまり、いずれのコネクターでも使用できるスクリプトは 1 つもありません。

- connector_manager_*connName*.sh 始動スクリプトは、各コネクターに固有です。ただし、これは、インストール・プロセスによって生成されます。したがって、カスタム・コネクターに対してそれを作成する必要はありません。
- カスタム起動スクリプトである start_*connName*.sh も、各コネクターに固有です。したがって、コネクターに対してカスタム起動スクリプトを作成し、それを製品ディレクトリーの connector¥*connName* サブディレクトリーに置きます。

コネクター固有のコネクター・マネージャー始動スクリプト: コネクターを始動する場合、connector_manager_*connName*.sh スクリプトの構文は、図 74 に示されている構文になります。この場合の *connName* はコネクターの名前、*additionalOptions* はコネクター起動に渡すための追加の始動パラメーターを指定するオプションの引き数になります。これらのオプションには、-f および -x があります。詳細については、 261 ページの表 113を参照してください。

```
connector_manager_connName -start additionalOptions
```

図 74. UNIX コネクターを始動するための構文

コネクター固有のコネクター・マネージャー始動スクリプト connector_manager_*connName* を作成するには、Connector Script Generator ツール (製品 bin ディレクトリー内の ConnConfig.sh) を使用します。コネクター名 (*connName*) を指定すると、このツールによって connector_manager_*connName* 始動スクリプトが生成され、製品ディレクトリーの bin サブディレクトリーに収められます。このツールについては、 587 ページの『付録 C. Connector Script Generator』を参照してください。

コネクター固有の起動スクリプト: コネクター固有の起動スクリプトを作成するには、start_*connName*.sh という新規のコネクター固有始動スクリプトを作成します (ここで、*connName* は使用している Java コネクター名です)。例えば、Java コネクターのコネクター名が MyJava である場合、始動スクリプト名は start_MyJava.sh になります。最初に、次のファイル内にある始動スクリプト・テンプレートをコピーできます。

```
ProductDir/templates/start_connName.sh
```

図 75. に、UNIX の起動スクリプト・テンプレートの内容のサンプルを示します。最新の内容については、ご使用の製品でリリースされているバージョンのファイルを参照してください。

```
#!/bin/sh

# set environment
\
#.${ADAPTER_RUNTIME}/bin/wbia_connEnv.sh

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"

# Please define the following variables that need to pass to callee
export JCLASSES=
export LibPath=
export ExtDirs=
export JVMArgs=

# Call base script start_adapter.sh to start a C++ connector
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nconnName -sserverName
-dconnSpecificDLLfile -f... -p... -c... ...

# Call base script start_adapter.sh to start a Java connector
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nconnName -sserverName
-lconnSpecificClasses -f... -p... -c... ...
```

図 75. UNIX ベースのシステムの始動スクリプト・テンプレートの内容のサンプル

UNIX ベースのシステムでは、start_*connName*.sh スクリプトの構文は、 250 ページの図 72 に示す構文になります。ただし、Windows システムの start_*connName* スクリプトと異なり、UNIX ベースのシステムの start_*connName* の構文は、 250 ページの図 72 に示されている構文に従っている必要があります。
connector_manager スクリプトは、この構文で start_*connName* を呼び出します。コネクター開発者は、start_*connName*.sh の内容を制御しますが、このスクリプトの構文は変更してはなりません。

推奨される命名規則 (241 ページの表 109 を参照) に従った場合、コネクター固有の起動スクリプトは、コネクター名 (*connName*) に基づき、コネクターのランタイム・ファイルについて次を想定します。

- 製品ディレクトリーの connectors サブディレクトリーの下にあるコネクター・ディレクトリーのコネクター名
- コネクター名は、コネクター・ディレクトリーにある Java コネクターのライブラリー・ファイル (その jar ファイル CW*connName*.jar) と同じ

例えば、MyJava コネクターがこれらの前提事項を満たしている場合、そのランタイム・ファイルは *ProductDir*/connectors/MyJava ディレクトリーにあり、その jar ファイルは BIA_MyJava.jarという名前でそのディレクトリー内にある必要があります。コネクターでこれらの前提事項を満たせない場合は、始動スクリプトをカスタマイズして、汎用コネクター始動スクリプト start_adapter.sh に適切な情報を提供する必要があります。

この start_*connName*.sh ファイルで、次のステップを実行します。

1. CWConnEnv.sh 環境ファイルを呼び出して、始動環境を初期化します。
2. コネクター・ディレクトリーに移動します。
3. コネクター固有の情報とコネクター固有の変数を使用して、始動スクリプト内で始動環境変数を設定します。
4. start_adapter.sh スクリプトを呼び出してコネクターを起動します。

これらの各ステップについては、以降のセクションで詳しく説明します。

環境ファイルの呼び出し: CWConnEnv.sh ファイルには、IBM Java Object Request Broker (ORB) および IBM Java Runtime Environment (JRE) の環境設定が含まれています。始動スクリプトでは、次の行によってこの環境ファイルが呼び出されます。

```
. ${ADAPTER_RUNTIME}/bin/CWConnEnv.sh
```

コネクター・ディレクトリーへの移動: start_*connName*.sh スクリプトは、start_adapter.sh スクリプトを呼び出す*前* にコネクター・ディレクトリーに変更する必要があります。コネクター・ディレクトリーには、コネクターの始動に必要なコネクター固有の始動スクリプトとその他のファイルが含まれています。このコネクター・ディレクトリーの名前は、任意の方法で定義できます。ただし、 248 ページの『コネクター・ディレクトリーの作成』で説明しているように、規則では、コネクターのディレクトリー名はコネクター名と同じです。

start_*connName*.sh スクリプトは、コネクター名が最初の引き数 ($1) で渡されることを期待します。したがって、次の行がコネクター・ディレクトリーに移動します。

```
# set the directory where the specific connector resides
CONNDIR=${CROSSWORLDS}/connectors/$1
export CONNDIR

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"
```

コネクター名はいくつかのコネクター・コンポーネントで使用するので、代わりの方法として、このコネクター名を指定する環境変数を定義し、start_*connName*.sh スクリプト内で後続のコネクター名のすべての使用に対してこの環境変数を評価することもできます。コネクター名とコネクター・ディレクトリーの環境変数を設定する行は、次のようになります。

```
# set the name of the connector
CONNAME=$1
export CONNNAME


REM set the directory where the specific connector resides
CONNDIR=${CROSSWORLDS}/connectors/${CONNAME}
export CONNDIR

# If required, go to directory where connector class files reside
cd /
cd "${CONNDIR}"
```

環境変数の設定: start_*connName*.sh スクリプトで、 252 ページの表 111 に示されている環境変数で指定するコネクター固有の情報を指定する必要があります。start_adapter.sh スクリプトは、Windows システムの start_adapter.bat スクリプトと同じようにこれらの環境変数を使用します。リリースごとに異なる可能性のある情報について独自のコネクター固有の環境変数を定義することもできます。詳細については、 252 ページの『環境変数の設定』を参照してください。

コネクターの起動: start_*connName*.sh スクリプトは、JVM 内でコネクターを実際に起動するために start_adapter.sh スクリプトを呼び出す必要があります。start_adapter.sh スクリプトは、*始動パラメーター* で、コネクターの実行時に必

要な環境 (コネクター・フレームワークを含む) を初期化するための情報に提供します。したがって、適切な始動パラメーターを start_adapter.sh に提供する必要があります。表 113 に、start_adapter.sh スクリプトによって認識される始動パラメーターを示します。

表 113. start_adapter.sh スクリプトの始動パラメーター

始動パラメーター	説明	必須ですか?	connector_manager_connName に対する追加のコマンド行オプションとして有効であるかどうか
-b	バックグラウンド・スレッドとしてコネクターを実行します。つまり、コネクターは、標準入力 (STDIN) からいかなる入力も受け取りません。このオプションは、汎用 connector_manager スクリプト (connector_manager_*connName* スクリプトごとに呼び出されます) が start_*connName*.sh スクリプトを呼び出すと、自動的に指定されます。したがって、コネクターがバックグラウンドで実行されないようにするには、start_*connName*.sh 起動から -b パラメーターを除去します。	説明を参照してください。	なし
-c*configFile*	コネクターの構成ファイルの絶対パス名。	統合ブローカーが ICS 以外の場合に必要となります。	はい
-f*pollFrequency*	ポーリング・アクション間の時間の長さです。指定可能な *pollFrequency* 値は次のとおりです。 • ポーリング・アクション間のミリ秒数。 • key。コネクターの始動ウィンドウで文字 p が入力された場合にのみ、コネクターがポーリングを実行するようにします。key オプションは、小文字で指定する必要があります。 • no。コネクターがポーリングを実行しないようにします。no オプションは、小文字で指定する必要があります。 -f パラメーターが指定する値によって、コネクターの構成ファイルのポーリング頻度をオーバーライドします。	なし デフォルトは 1000 ミリ秒です。	はい
-l*classname*	Java コネクターのコネクター・クラス (*className*) の名前。 **注:** -b パラメーターは、connector_manager_*connName* スクリプトに対して有効なコマンド行オプションではありません。	はい	なし
-n*connectorName*	始動するコネクター (*connectorName*) の名前。	はい	なし

表 113. start_adapter.sh スクリプトの始動パラメーター (続き)

始動パラメーター	説明	必須ですか?	connector_manager_connName に対する追加のコマンド行オプションとして有効であるかどうか
-s*brokerName*	コネクターが接続する先の統合ブローカー (*brokerName*) の名前。	はい	なし
-t*threadingType*	コネクターに使用するスレッド化モデルを示します。*threadingType* に対して指定可能な値は次のとおりです。 • SINGLE_THREADED: 単一スレッドのみがアプリケーションにアクセスします。 • MAIN_SINGLE_THREADED: メイン・スレッドのみがアプリケーションにアクセスします。 • MULTI_THREADED: 複数のスレッドがアプリケーションにアクセスできます。	なし	なし
-x*connectorProps*	アプリケーション固有のコネクター・プロパティーの値を初期化します。次のフォーマットを使用して、各プロパティーを指定してください。 *propName=value*	なし	はい

start_adapter.sh の呼び出しに次の始動パラメーターが含まれることを確認します。

- 必須の始動パラメーターすべて
  - コネクター定義の名前を指定する場合: -n

    コネクターの名前は最初の引き数 ($1) として start_*connName*.sh スクリプト (250 ページの図 72 を参照) に渡されるので、-n 始動パラメーターは次のように指定できます。

    ```
    -n${1}Connector
    ```

    コネクター名 (CONNAME など) の環境変数を定義すると、この -n パラメーターは次のように表示されます。

    ```
    -n${CONNAME}Connector
    ```

  - InterChange Server インスタンスの名前を指定する場合: -s

    ICS インスタンスの名前が 2 番目の引き数 ($2) として start_*connName*.sh スクリプト (250 ページの図 72 を参照) に渡される場合、-s 始動パラメーターは次のように指定します。

    ```
    -s${2}
    ```

注: すべての UNIX のコネクターには、通常は -b 始動パラメーターが組み込まれます。このオプションによって、コネクターのプロセスがバックグラウンドで実行されます。したがって、connector_manager 汎用始動スクリプトで

は、すべての コネクターについて、この始動パラメーターが自動的に指定されます。 start_adapter.sh 呼び出しでこれを指定する必要はありません。

その他の統合ブローカー

統合ブローカーが WebSphere MQ Integrator Broker、WebSphere Integration Message Broker、または WebSphere Application Server である場合、-c オプションも必須始動パラメーターとなります。

- Java コネクターに必要な言語固有の始動パラメーター:

  コネクター固有のクラス (またはパッケージ) を指定する場合: -l

  例えば、推奨される命名規則に従うと、Java コネクター名 MyJava の言語固有のパラメーターは次のようになります。

  ```
  -lcom.crossworlds.connectors.MyJava.MyJavaAgent
  ```

  コネクター名 (CONNAME など) の環境変数を定義すると、この -l パラメーターは次のように表示されます。

  ```
  -lcom.crossworlds.connectors.${CONNAME}.${CONNAME}Agent
  ```

- コネクターのすべての起動時に適用されるオプションの始動パラメーター。オプションの始動パラメーター・リストについては、261 ページの表 113 を参照してください。

始動パラメーターの詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「システム*管理ガイド*」または WebSphere Business Integration Adapters ドキュメンテーション・セット内のインプリメンテーション・ガイド (ご使用の統合ブローカー用のもの) を参照してください。

start_adapter.sh への呼び出しの構文のフォーマットは次のようになります。

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -n connDefName -s ICSinstance
 -l className -cCN_ connName Connector.cfg
 -...
```

例えば、次の行は、MyJava コネクターを呼び出します。

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -nMyJavaConnector -sICSserver
 -lcom.crossworlds.connectors.MyJava.MyJavaAgent
 -cMyJavaConnector.cfg  -...
```

注: 上記のコマンド行は、コネクターが、ICSserver という名前の InterChange Server インスタンスに対して実行されていることを想定しています。コネクターが WebSphere MQ Integrator Broker インスタンスに対して実行されている場合は、そのインスタンス名がコマンド行に表示される必要があります。

CONNAME 環境変数を使用してコネクター名を保持した場合、この呼び出しは一般的に次のようになります。

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -n${CONNAME}Connector -s${2}
-lclassName -cCN_${CONNAME}Connector.cfg  -...
```

start_adapter.sh への呼び出しでは、次の点に留意してください。

- コネクター・ランタイムを起動する行全体は、始動スクリプト内での *1 行に収めて* 入力します。つまり、サンプル始動行で示されている改行個所で改行しないようにしてください。
- `start_adapter.sh` への呼び出しでリストするパラメーターの順序は、重要ではありません。
- ユーザーが `connector_manager_connName.sh` の呼び出しに渡す可能性があるすべての追加オプションを `start_adapter.sh` の呼び出しで処理しなければならない場合もあります (258 ページの図 74 を参照)。この場合は、`start_adapter.sh` に渡すための「追加の」引き数を指定して、追加のオプションが実際のコネクター起動に渡されるようにする必要があります。例えば、`start_adapter.sh` の呼び出しでは、次の 3 つの追加コマンド行オプションが処理されます。

```
exec ${ADAPTER_RUNTIME}/bin/start_adapter.sh -n${CONNAME}Connector -s${2}
-lclassName -cCN_${CONNAME}Connector.cfg ${3} ${4} ${5}
```

Windows サービスとしてのコネクターの開始

コネクターは、リモート管理者が開始、停止できる Windows サービスとして実行されるようセットアップすることができます。詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム・インストール・ガイド (Windows 版)*」または IBM WebSphere Business Integration Adapter ドキュメンテーション・セット内のインプリメンテーション・ガイドを参照してください。

注: 統合ブローカーとして InterChange Server を使用しており、コネクターで自動およびリモートの再始動機能を使用する場合は、コネクターを Windows サービスとして開始しないでください。代わりに、MQ Trigger Monitor をサービスとして開始してください。詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

第 3 部 Java コネクター・ライブラリー API 参照

第 9 章 Java コネクター・ライブラリーの概要

Java コネクター・ライブラリーには、コネクターの開発で使用する必要のあるクラス・ライブラリーが組み込まれています。このコネクター・クラス・ライブラリーには、Java のコネクターの定義済みクラスが入っています。これらのクラス・ライブラリーを使用して、コネクター・クラスとメソッドを派生させます。また、クラス・ライブラリーには、トレースやロギングのサービスを実装するメソッドなどのユーティリティーもあります。

IBM の提供する Java jar ファイル (Java アーカイブ・ファイル) `WBIA.jar` には、Java コネクター・ライブラリーの定義済みクラスおよびインターフェースが入っています。現行バージョンの `WBIA.jar` ファイルは、製品ディレクトリーの `lib` サブディレクトリーに入っています。古いバージョンの `WBIA.jar` ファイルは、以下の製品サブディレクトリーに入っています。

`lib¥WBIA¥`*`version`*

ここで、*`version`* は、Java コネクター・ライブラリーのバージョンを表します。現行バージョンの `WBIA.jar` は、このライブラリーの旧バージョンと互換性があります。

注: Windows NT または Windows 2000 で実行するために Java コネクターを構築する方法については、242 ページの『コネクターのコンパイル』を参照してください。

クラスおよびインターフェース

表 114 に、Java コネクター・ライブラリーのクラスとインターフェースのリストを示します。

表 114. Java コネクター・ライブラリーのクラスとインターフェース

クラスまたはインターフェース	説明	ページ
`CWConnectorAgent`	コネクターの基底クラスを表します。この基底クラスを拡張して、コネクター・クラスを定義し、必須の仮想メソッドを実装します。	515
`CWConnectorAttrType`	属性型定数を定義します。	285
`CWConnectorBOHandler`	ビジネス・オブジェクト・ハンドラーの基底クラスを表します。この基底クラスを拡張して、コネクターに対して 1 つ以上のビジネス・オブジェクト・ハンドラーを定義します。	287
`CWConnectorBusObj`	ビジネス・オブジェクト・インスタンスを表します。ビジネス・オブジェクト、ビジネス・オブジェクト定義、および属性へのアクセスを提供します。	293
`CWConnectorConstant`	Java コネクター・ライブラリーで使用する次の定数を定義します。 • 結果状況定数 • 動詞定数	349

表 114. Java コネクター・ライブラリーのクラスとインターフェース (続き)

クラスまたはインターフェース	説明	ページ
CWConnectorEvent	イベント・オブジェクトを表します。このイベント・オブジェクトはイベント・ストアから取得されたイベント・レコードの情報を保留します。	351
CWConnectorEventStatusConstants	イベント状況定数を定義します。この定数はイベント・レコードが取ることのできる状況値を表します。	361
CWConnectorEventStore	イベント・ストアを表します。イベント・ストアはコネクターのイベント検出機構によるアクセス (通常はポーリング) のためにイベント・レコードを保留します。	365
CWConnectorEventStoreFactory	イベント・ストア・ファクトリーを表します。イベント・ストア・ファクトリーは CWConnectorEventStore オブジェクトをインスタンス化します。	381
CWConnectorExceptionObject	例外詳細オブジェクトを表します。このオブジェクトには、例外オブジェクトに組み込まれる追加の状況情報が入っています。	383
CWConnectorLogAndTrace	ロギングおよびトレースのサービスで使用する次の定数を定義します。 • メッセージ・ファイル定数 • メッセージ・タイプ定数 • トレース・レベル定数	391
CWConnectorReturnStatusDescriptor	エラー・メッセージと情報メッセージを含む戻り状況記述子を表します。	393
CWConnectorUtil	Java コネクターで使用する各種のユーティリティー・メソッドを提供します。これらのユーティリティー・メソッドは、次のような一般カテゴリーに分かれます。 • メッセージの生成およびロギングのための静的メソッド • ビジネス・オブジェクトの作成のための静的メソッド • コネクター構成プロパティーの取得のための静的メソッド • ロケール情報の取得のためのメソッド	397
CWException	Java コネクター・ライブラリー用の例外オブジェクトを表します。	441
CWProperty	階層コネクター構成プロパティーを含むコネクター・プロパティー・オブジェクトを表します。	515

第 10 章 CWConnectorAgent クラス

CWConnectorAgent クラスは、Java コネクター対応の基底クラスです。コネクター開発者は、このクラスからコネクター・クラス を派生し、そのコネクターにユーザー定義メソッドを実装する必要があります。派生されたコネクター・クラスには、コネクターのアプリケーション固有コンポーネントのコーディングが含まれます。

注: CWConnectorAgent クラスは、下位の Java コネクター・ライブラリーの ConnectorBase クラスを拡張したものです。下位の Java コネクター・ライブラリーのクラスの詳細については、465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

重要: すべての Java コネクターは、このコネクター基底クラスを拡張し、agentInit()、getVersion()、getConnectorBOHandlerForBO()、pollForEvents()、および terminate() の各メソッドの実装を提供しなければなりません。ただし、CWConnectorAgent では、getVersion()、getConnectorBOHandlerForBO()、および pollForEvents() メソッドのデフォルトの実装が用意されています。それらの派生したコネクター基底クラスにおいて、開発者は、用意されているデフォルトの実装を使用することも、それらを無効にして、独自の実装を行うこともできます。開発者は、agentInit() メソッドと terminate() メソッドの実装を用意する必要 があります。

表 115 に、CWConnectorAgentクラスのメソッドを要約します。

表 115. CWConnectorAgent クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorAgent()	コネクター・オブジェクトを作成します。	270
agentInit()	コネクターを初期化します。	270
executeCollaboration()	同期要求としてコラボレーションにビジネス・オブジェクト要求を送信します。	272
getCollabNames()	ビジネス・オブジェクト要求の処理に使用可能なコラボレーションのリストを取得します。	273
getConnectorBOHandlerForBO()	指定されたビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。	274
getEventStore()	コネクターのイベント・ストアへの参照を取得します。	275
getVersion()	コネクターのバージョンを取得します。	276
gotApplEvent()	InterChange Server にビジネス・オブジェクトを送信します。	277
isAgentCapableOfPolling()	このコネクター・プロセスがポーリングできるかどうかを判別します。	279
isSubscribed()	統合ブローカーが、特定の動詞を持つ特定のビジネス・オブジェクトにサブスクライブしているかどうかを判別します。	280
pollForEvents()	ビジネス・オブジェクトに変更を起こすイベントをポーリングして、アプリケーションのイベント・ストアを調べます。	282

表 115. *CWConnectorAgent* クラスのメンバー・メソッド *(続き)*

メンバー・メソッド	説明	ページ
terminate()	アプリケーションとの接続をクローズし、割り当てられているリソースを解放します。	283

CWConnectorAgent()

コネクター・オブジェクトを作成します。

構文

```
public CWConnectorAgent();
```

パラメーター

なし。

戻り値

新規作成コネクターを含む `CWConnectorAgent` オブジェクトです。

agentInit()

コネクターを初期化します。

構文

```
public void agentInit();
```

パラメーター

なし。

戻り値

なし。

例外

`ConnectionFailureException`
: コネクターがアプリケーションとの接続の確立に失敗した場合にスローされます。

`InProgressEventRecoveryFailedException`
: コネクターが、進行中イベント・リカバリーを実行できない場合にスローされます。

`LogonFailedException`
: コネクターが、アプリケーションにログインできなかった場合にスローされます。

PropertyNotSetException
: コネクターが、値が設定されていない必須コネクター構成プロパティーを取得した場合にスローされます。

注記

agentInit() メソッドは、コネクターのアプリケーション固有コンポーネントに必要な次のタスクを含め、コネクターに関するすべての初期化機能を実行します。

- 接続の確立
- コネクター・プロパティーの取得
- 進行中イベントのリカバリー

重要: CWConnectorAgent クラスは agentInit() クラスをデフォルトで実装していません。コネクター・クラスにこのメソッドを実装する必要があります。

コネクター・フレームワークは、agentInit()メソッドを呼び出し、コネクターを起動時に初期化します。 agentInit() メソッドは、表 116 のリストに記載されている条件のいずれかを実行する場合、次の条件をチェックして、該当する例外をスローする必要があります。

表 116. agentInit() メソッドがスローする例外

条件	スローする例外
コネクターが、設定されていない必須コネクター構成プロパティーを検出した場合	PropertyNotSetException
コネクターがアプリケーションとの接続の確立に失敗した場合	ConnectionFailureException
コネクターがアプリケーションへのログオンに失敗した場合	LogonFailedException
recoverInProgressEvents() メソッドがイベント・ストア内で進行中イベントを検出し、リカバリー処理中に何らかのエラーが発生した場合	InProgressEventRecoveryFailedException

表 116 に示した例外のうちのいずれかをスローするためには、表 117 に示すステップに従ってください。

表 117. 初期化エラーの処理

エラー処理のステップ	使用するメソッドまたはコード
1. エラーが発生した場合には、初期化エラーの原因を示すために、ログ宛先にエラー・メッセージを記録します。	CWConnectorUtil.generateAndLogMsg()
2. 例外情報を保持するために例外詳細オブジェクトのインスタンスを生成します。	CWConnectorExceptionObject *excptnDtailObj* = new CWConnectorExceptionObject();
3. 例外詳細オブジェクト内の状況情報を設定します。	
• 初期化障害の原因を示すためのメッセージを設定します。	*excptnDtailObj*.setMsg()
• コネクター・フレームワークに初期化の成功を通知する結果状況を設定します。初期化プロセス (およびコネクター) を終了させたい場合には、結果状況を CxConnectorConstant.FAIL に設定してください。	*excptnDtailObj*.setStatus()

表 117. 初期化エラーの処理 (続き)

エラー処理のステップ	使用するメソッドまたはコード
4. 初期化の障害を通知するために、表 116 に示した agentInit() 例外をスローします。この例外は、agentInit() メソッドが初期化エラーの発生をコネクター・フレームワークにどのように通知するのかを示すものです。この例外オブジェクトには、ステップ 3 で初期化した例外詳細オブジェクトが格納されます。 下位の init() メソッド (このメソッドは agentInit() を呼び出します) は、この例外オブジェクトをキャッチすると、例外詳細オブジェクトから得られた状況をそれ自体の戻り状況にコピーして、コネクター・フレームワークに戻します。 **注:** 例外詳細オブジェクトで例外状況を設定しなかった場合、init() メソッドが結果状況 FAIL を戻し、コネクター・フレームワークがそのコネクターを終了させます。	`throw new agentInit Exception( excptnDtailObj);`

参照項目

generateAndLogMsg(), recoverInProgressEvents()

executeCollaboration()

コネクター・フレームワークにビジネス・オブジェクト要求を送信します。コネクター・フレームワークは、そのビジネス・オブジェクト要求を統合ブローカー内のビジネス・プロセスに送信します。これは同期要求です。

構文

```
public void executeCollaboration(String busProcName,
   CWConnectorBusObj theBusObj,
   CWConnectorReturnStatusDescriptor rtnStatusDesc);
```

パラメーター

busProcName ビジネス・オブジェクト要求を実行するビジネス・プロセスの名前を指定します。ご使用の統合ブローカーが InterChange Server の場合、ビジネス・プロセス名は、コラボレーション名です。

theBusObj トリガー・イベントと、ビジネス・プロセスから戻されるビジネス・オブジェクトです。

rtnStatusDesc ビジネス・プロセスからのメッセージおよび実行状況または戻り状況を含む戻り状況記述子です。

戻り値

なし。

例外

なし。

注記

executeCollaboration() メソッドは、コネクター・フレームワークに *theBusObj* ビジネス・オブジェクトを送信します。コネクター・フレームワークは、イベント・オブジェクトにある処理を施して、データを直列化し、そのデータが正しく永続化されるようにします。その後、統合ブローカーの *busProcName* ビジネス・プロセスにイベントを送信します。このメソッドは、イベントの同期実行を開始します (つまり、統合ブローカーのビジネス・プロセスからの応答を待機します)。

WebSphere InterChange Server

ご使用の統合ブローカーが IBM WebSphere InterChange Server の場合、executeCollaboration() が呼び出すビジネス・プロセスはコラボレーションです。

ビジネス・プロセスの実行に関する状況情報を受け取るには、インスタンス化された戻り状況記述子 *rtnStatusDesc* を最後の引き数としてメソッドに渡します。統合ブローカーはビジネス・プロセスから状況情報を戻し、これをコネクター・フレームワークに送信することができます。コネクター・フレームワークでは、この戻り状況記述子にこの情報を取り込みます。この状況情報にアクセスするには、CWConnectorReturnStatusDescriptor クラスのメソッドを使用します。

注: イベントの非同期実行を開始するには、gotApplEvent() メソッドを使用します。非同期実行では、呼び出し側コードはイベントの受信を待機せず、応答も待機しません。

参照項目

gotApplEvent()、CWConnectorReturnStatusDescriptor クラスのメソッド

getCollabNames()

ビジネス・オブジェクト要求の処理に使用可能なコラボレーションのリストを取得します。

WebSphere InterChange Server

このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
public String[] getCollabNames();
```

パラメーター

なし。

戻り値

コラボレーション名のリストを含む String オブジェクトの配列です。

例外

なし。

getConnectorBOHandlerForBO()

指定されたビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。

構文

```
public CWConnectorBOHandler getConnectorBOHandlerForBO(
   String busObjName);
```

パラメーター

busObjName　　ビジネス・オブジェクトの名前です。

戻り値

busObjName ビジネス・オブジェクトのビジネス・オブジェクト・ハンドラーを表す CWConnectorBOHandler オブジェクトに対する参照です。

例外

なし。

注記

コネクター・フレームワークは、getConnectorBOHandlerForBO() メソッドを呼び出し、ビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。複数のビジネス・オブジェクト定義に対して、1 つのビジネス・オブジェクト・ハンドラーを使用することも、各ビジネス・オブジェクト定義に対して 1 つのビジネス・オブジェクト・ハンドラーを使用することもできます。

重要: CWConnectorAgent クラスは、getConnectorBOHandlerForBO() メソッドをデフォルトで実装しています。そのため、このデフォルトの実装を使用することも、メソッドのオーバーライドによって、ユーザー独自のビジネス・オブジェクト・ハンドラー・クラスを戻すこともできます。

CWConnectorAgent クラスは、getConnectorBOHandlerForBO() メソッドに対するデフォルトの実装を用意しています。そのデフォルトの実装は、ConnectorBOHandler クラスのビジネス・オブジェクト・ハンドラーに対する参照を戻します。このデフォルトの実装を使用するには、CWConnectorBOHandler クラスを拡張し、拡張したクラスに ConnectorBOHandler という名前を付けます。ビジネス・オブジェクト・ハンドラー基底クラスに、ConnectorBOHandler とは違う名前を付けた場合は、getConnectorBOHandlerForBO() を無効にし、拡張したビジネス・オブジェクト・ハンドラー基底クラスに対する参照を戻り値にする必要があります。

getEventStore()

コネクターのイベント・ストアへの参照を作成します。

構文

```
public CWConnectorEventStore getEventStore();
```

パラメーター

なし。

戻り値

コネクターのイベント・ストアへのアクセスを提供する CWConnectorEventStore オブジェクト。イベント・ストア・ファクトリー・クラスが見つからない場合、メソッドは null を戻します。

例外

なし。

注記

getEventStore() メソッドは、コネクターのイベント・ストア・オブジェクトをインスタンス化するタスクを備えたイベント・ストア・ファクトリーです。コネクターは、このイベント・ストア・オブジェクトを通じて、そのイベント・ストアにアクセスできます。 getEventStore() メソッドは、CWConnectorEventStoreFactory インターフェースを実装しているユーザーのイベント・ストア・ファクトリー・クラスにある getEventStore() メソッドを呼び出します。

重要: CWConnectorAgent クラスは、getEventStore() メソッドをデフォルトで実装しています。したがって、ユーザーはこのデフォルトの実装を使用することもできますし、メソッドをオーバーライドして、イベント・ストア・オブジェクトをインスタンス化する独自の機構を実装することもできます。

CWConnectorAgent クラスにデフォルトで実装されている getEventStore() メソッドは、(CWConnectorEventStoreFactory インターフェースを実装している) イベント・ストア・ファクトリー・クラスの名前が EventStoreFactory コネクター構成プロパティーにないかどうかを確認します。

- EventStoreFactory プロパティーが設定されている場合、getEventStore() は、指定されたイベント・ストア・ファクトリー・クラスをインスタンス化し、その getEventStore() メソッドを呼び出してイベント・ストア・オブジェクトを戻します。
- EventStoreFactory プロパティーが設定されていない 場合、getEventStore() は、イベント・ストア・ファクトリー・クラス名の作成を試みます。

  getEventStore() メソッドは、コネクター・パッケージの名前からコネクター名を抽出します。このメソッドは、イベント・ストアの名前が次のようであると想定しています。

  *connectorName*EventStore

例えば、WebSphere Business Integration Adapter for JDBC では、コネクターの名前は JDBC となります。したがって、getEventStore() は、JDBCEventStore をコネクターのイベント・ストアの名前として生成し、この名前のイベント・ストア・ファクトリー・クラスのインスタンス化を試みます。

EventStoreFactory プロパティーには、イベント・ストア・ファクトリー・インスタンスの完全なクラス名を指定する必要があります。このプロパティーの形式については、205 ページの『CWConnectorEventStoreFactory インターフェース』を参照してください。例えば、WebSphere Business Integration Adapter for JDBC には、JDBC イベント・ストアへのアクセスを提供するイベント・ストア・ファクトリーが含まれています。したがって、EventStoreFactory プロパティーは次のように設定できます。

```
com.crossworlds.connectors.JDBC.JDBCEventStoreFactoryInstance
```

デフォルトで実装されている pollForEvents() メソッドは、この getEventStore() メソッドを呼び出して、イベント・ストアへの参照を取得します。詳細については、211 ページの『イベント・レコードの取得』を参照してください。

参照項目

getEventStore(),, pollForEvents()

---

getVersion()

コネクターのバージョンを取得します。

構文

```
public String getVersion();
```

パラメーター

なし。

戻り値

コネクターのアプリケーション固有のコンポーネントのバージョンを示す String です。

例外

なし。

注記

コネクター・フレームワークは、getVersion() メソッドを呼び出し、コネクターのバージョンを取得します。getVersion() メソッドは、通常、agentInit() メソッドからコネクター初期化処理の一環として呼び出されます。コネクター・フレームワークは、getVersion () メソッドを呼び出して、コネクターのバージョンも取得します。

重要: CWConnectorAgent クラスは、getVersion() メソッドをデフォルトで実装しています。そのため、このデフォルトの実装を使用することも、メソッドをオーバーライドしてユーザー独自のバージョン管理機構を実装することもできます。

CWConnectorAgent クラスは、標準クラスの情報からパッケージ名を取得する getVersion() メソッドに対してデフォルトの実装を提供します。さらに、パッケージ内のマニフェスト・ファイルからバージョンを取得します。

---

gotApplEvent()

コネクター・フレームワークにビジネス・オブジェクト要求を送信します。これは非同期要求です。

構文

```
public int gotApplEvent(CWConnectorBusObject theBusObj);
```

パラメーター

theBusObj コネクター・フレームワークに送信されるビジネス・オブジェクトのインスタンスです。

戻り値

イベント・デリバリーの結果状況を示す整数です。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

CWConnectorConstant.SUCCEED
: コネクター・フレームワークは、コネクター・フレームワークへのビジネス・オブジェクトの配信に成功しました。

CWConnectorConstant.FAIL
: イベント・デリバリーが失敗しました。

CWConnectorConstant.CONNECTOR_NOT_ACTIVE
: コネクターは、一時停止しているためイベントを受信できません。

CWConnectorConstant.NO_SUBSCRIPTION_FOUND
: ビジネス・オブジェクトが表すイベントに対してサブスクリプションがありません。

例外

なし。

注記

gotApplEvent() メソッドは、コネクター・フレームワークに *theBusObj* ビジネス・オブジェクトを送信します。コネクター・フレームワークは、イベント・オブジェクトにある処理を施して、データを直列化し、そのデータが正しく永続化されるようにします。その上で、イベントが統合ブローカーに送信されることを確認します。

WebSphere InterChange Server

統合ブローカーが InterChange Server である場合、コネクター・フレームワークは、構成済みのデリバリー・トランスポート機構 (JMS や CORBA IIOP など) を使用して、イベントを (ビジネス・オブジェクトとして) InterChange Server に送信します。

その他の統合ブローカー

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server である場合、コネクター・フレームワークは、JMS キューの構成済みのデリバリー・トランスポート機構を使用して、イベントを (XML メッセージとして) 統合ブローカーに送信します。

gotApplEvent() は、コネクター・フレームワークにビジネス・オブジェクトを送信する前に、次の条件をチェックし、条件が満たされていない 場合には、関連する結果状況を戻します。

条件	結果状況
コネクターの状況がアクティブか、すなわち、コネクターは、「一時停止」状態にないか。コネクターのアプリケーション固有コンポーネントが一時停止している場合、アプリケーションに対するポーリングはすでに停止しています。	CONNECTOR_NOT_ACTIVE
イベントに対するサブスクリプションは存在するか。	NO_SUBSCRIPTION_FOUND

注: gotApplEvent() が、送信されるビジネス・オブジェクトと動詞が有効なサブスクリプションを持っていることを確認するため、gotApplEvent() の呼び出し直前に、isSubscribed() を呼び出す必要はありません。

コネクターは、pollForEvents() メソッドを使用してイベント・ストアにポーリングし、サブスクライブされたイベントが統合ブローカーに送信されるようにします。pollForEvents() 内で、コネクターは、gotApplEvent() メソッドを使用してイベント (ビジネス・オブジェクトとして表現されます) をコネクター・フレームワークに送信します。コネクター・フレームワークは、このビジネス・オブジェクトを統合ブローカーに送ります。したがって、ポーリング・メソッドは、gotApplEvent() からの戻りコードをチェックし、戻されたエラーの適切な処理を保証する仕組みになっています。例えば、ポーリング・メソッドは、イベント・デリバリーが正常に実行されるまでイベント・ストアからイベントを除去しません。ポーリング・メソッドは、gotApplEvent() の戻りコードに基づき、イベント・デリバリーの結果が反映されるようにイベント・レコードの状況を更新します。詳細については、 219 ページの『ビジネス・オブジェクトの送信』を参照してください。

gotApplEvent() メソッドは、イベントの非同期実行を開始します。非同期実行では、メソッドはイベントの受信を待機せず、応答も待機しません。

注: イベントの同期実行を開始するには、executeCollaboration() メソッドを使用します。同期実行では、呼び出し側コードがイベントの受信および応答を待機します。

参照項目

executeCollaboration(),, isSubscribed(),, pollForEvents()

isAgentCapableOfPolling()

このコネクター・プロセスがポーリングできるかどうかを判別します。

WebSphere InterChange Server
このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
boolean isAgentCapableOfPolling();
```

パラメーター

なし。

戻り値

このコネクターが、ポーリングできるかどうかを示す boolean 値です。この戻り値は、コネクターのタイプに依存します。

コネクター・プロセスのタイプ	戻り値
マスター (逐次処理)	true
マスター (並列処理)	false
スレーブ (要求)	false
スレーブ (ポーリング)	true

例外

なし。

注記

コネクターが、単一プロセス・モードで動作するように構成されている (デフォルト設定である、ParallelProcessDegree() に 1 がセットされている) 場合、isAgentCapableOfPolling() メソッドは、常時、true を戻します。これは、同一のコネクターが、イベント・ポーリングと要求処理の両方を実行するためです。

コネクターが、並列処理モードで動作するように構成されている (ParallelProcessDegree が 1 より大きい) 場合、コネクターは、表 118 に示されているように、それぞれが特定の目的を持つ複数のプロセスから構成されます。

表 118. 並列コネクターのプロセスの目的

コネクター・プロセス	コネクター・プロセスの目的
コネクター・エージェント・マスター・プロセス	ICS からの着信イベントを受信し、どのコネクターのスレーブ・プロセスにそのイベントを送信するかを決定します。
要求処理スレーブ・プロセス	コネクターのハンドル要求
ポーリング・スレーブ・プロセス	コネクターに対するポーリングとイベント・デリバリーを処理します。

isAgentCapableOfPolling() の戻り値は、メソッドの呼び出し元のコネクター・エージェント・プロセスの目的に依存します。並列処理コネクターの場合、このメソッドは、ポーリング・スレーブとして機能する目的のコネクターから呼び出されたとき、true のみ を戻します。並列処理コネクターの詳細については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

注: コネクターの並列処理モードについての情報は、isAgentCapableOfPolling() メソッドによって取得されるため、この機能をサポートしているバージョンの InterChange Server (ICS) で、このメソッドを実行しなければなりません。この理由から、ここに文書化されているように動作させるには、バージョン 4.0 以降の ICS で isAgentCapableOfPolling() を実行する必要があります。それよりも前のバージョンの ICS で実行した場合、isAgentCapableOfPolling() は、常に true を戻します。

isSubscribed()

統合ブローカーが、特定の動詞を持つ特定のビジネス・オブジェクトにサブスクライブしているかどうかを判別します。

構文

```
public boolean isSubscribed(String busObjName, String verb);
```

パラメーター

busObjName サブスクリプションを検査する対象のビジネス・オブジェクトの名前です。

verb ビジネス・オブジェクトのアクティブな動詞です。

戻り値

統合ブローカーが、指定されているビジネス・オブジェクトと動詞の受信に関連する場合に、true を戻します。それ以外の場合は、false を戻します。

例外

なし。

注記

isSubscribed() メソッドは、サブスクリプション・マネージャーの一部です。サブスクリプション・マネージャーは、コネクター・フレームワークから到着するすべてのサブスクライブ・メッセージとアンサブスクライブ・メッセージを追跡し、アクティブなビジネス・オブジェクト・サブスクリプションのリストを維持します。Java コネクターの場合、このサブスクリプション・マネージャーは、コネクター基底クラスの一部になります。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、ポーリング・メソッドは、指定された*動詞* を持つ *busObjName* ビジネス・オブジェクトに、コラボレーションがサブスクライブしているかどうかを判別できます。初期化時、コネクター・フレームワークは、コネクター・コントローラーから自身のサブスクリプション・リストを要求します。実行時、ポーリング・メソッドは、isSubscribed() を使用して、コネクター・フレームワークに照会し、あるコラボレーションが、特定のビジネス・オブジェクトにサブスクライブしていることを確認できます。ポーリング・メソッドは、あるコラボレーションが現在サブスクライブされている場合に限り、イベントを送信することができます。詳細については、14 ページの『ビジネス・オブジェクトのサブスクリプションとパブリッシュ』を参照してください。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークでは、統合ブローカーがコネクターによってサポートされるすべての ビジネス・オブジェクトに関係していると想定します。アプリケーション固有コンポーネントが、isSubscribed() メソッドを使用して、特定のビジネス・オブジェクトに対するサブスクリプションに関してコネクター・フレームワークに照会する場合は、メソッドは、コネクターのサポートするビジネス・オブジェクトごと に true を戻します。

参照項目

gotApplEvent(),, pollForEvents()

pollForEvents()

ビジネス・オブジェクトに変更を起こすイベントをポーリングして、アプリケーションのイベント・ストアを調べます。

構文

```
public int pollForEvents();
```

パラメーター

なし。

戻り値

ポーリング操作の結果状況を示す整数です。pollForEvents() メソッドは、一般的に、次の戻りコードを使用します。

CWConnectorConstant.SUCCEED
: イベントの取得結果に関係なくポーリング動作は成功しました。

CWConnectorConstant.FAIL
: ポーリング操作が失敗しました。

CWConnectorConstant.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

例外

なし。

注記

コネクター・フレームワークが、ユーザーの設定可能な時間間隔で pollForEvents() メソッドを呼び出すため、コネクターは、サブスクライバーに関心のあるアプリケーションにおいてイベントを検出することができます。クラス・ライブラリーがこのメソッドを呼び出す頻度は、PollFrequency コネクター構成プロパティーによって設定されるポーリング頻度値に依存します。

注: CWConnectorAgent クラスは、pollForEvents() メソッドをデフォルトで実装しています。したがって、用意されているデフォルトの実装を使用することも、メソッドを無効にして、ユーザー自身のポーリングの機構を実装することもできます。別のポーリングの振る舞いを実現するには、ユーザー独自のバージョンの pollForEvents() を実装します。

CWConnectorAgent クラスには、pollForEvents() メソッドのデフォルトの実装が用意されています。この実装は、イベント管理用の標準インターフェースとしての、CWConnectorEvent イベント・オブジェクトをベースにしています。このデフォルトの実装の振る舞いについては、208 ページの『pollForEvents() メソッドの実装』を参照してください。このデフォルトの実装では、イベント・ストアをポーリングするための基本的なステップが提供されます。デフォルトの pollForEvents() をオーバーライドする場合は、ユーザーの実装で同様のステップを実行する必要があります。

注: 並列処理モードで実行されているコネクターの場合は、別々のポーリング・スレーブ・プロセスを使用して、ポーリングを処理します。

参照項目

gotApplEvent(),, isSubscribed()

---

terminate()

コネクターを終了し、必要なクリーンアップ・タスクを実行します。

構文

```
public int terminate();
```

パラメーター

なし。

戻り値

`terminate()` 操作の状況値を示す整数です。

`CWConnectorConstant.SUCCEED`
: 終了操作は成功しました。

`CWConnectorConstant.FAIL`
: 終了操作は失敗しました。

例外

なし。

注記

コネクター・インフラストラクチャーは、コネクターのシャットダウン時、`terminate()` メソッドを呼び出します。このメソッドのユーザーの実装では、優れた作業慣行として、メモリーをすべて解放し、アプリケーションからログオフすることを推奨します。このメソッドは、ユーザーがコネクターに対して実装する必要があります。

重要: `CWConnectorAgent` クラスは、`terminate()` メソッドをデフォルトで実装していません。したがって、リソースのクリーンアップが必要な場合は、コネクター・クラスにこのメソッドを実装する必要があります。

第 11 章 CWConnectorAttrType クラス

CWConnectorAttrType クラスは、Java コネクター用の属性型クラスです。このクラスは、ビジネス・オブジェクト定義内の属性のデータ型に対応する静的定数を定義します。

属性型定数

CWConnectorAttrType クラスにより、属性型に対応して数値型とストリング型を表す定数が定義されます。表 119 に、CWConnectorAttrType クラスに属する属性型定数を示します。

表 119. CWConnectorAttrType クラスの静的定数

属性のデータ型	数値属性型定数	ストリング属性型定数
ブール値	BOOLEAN	BOOLSTRING
ビジネス・オブジェクト: 複数カーディナリティー	なし	MULTIPLECARDSTRING
ビジネス・オブジェクト: 単一カーディナリティー	なし	SINGLECARDSTRING
暗号文	CIPHERTEXT	CIPHERTEXTSTRING
「ID の欠落」	なし	CXMISSINGID_STRING
日付	DATE	DATESTRING
倍精度	DOUBLE	DOUBSTRING
浮動小数点	FLOAT	FLTSTRING
整数	INTEGER	INTSTRING
無効なデータ型	INVALID_TYPE_NUM	INVALID_TYPE_STRING
ロング・テキスト	LONGTEXT	LONGTEXTSTRING
オブジェクト	OBJECT	なし
ストリング	STRING	STRSTRING
Blank 値	なし	CxBlank
Ignore 値	なし	CxIgnore

第 12 章 CWConnectorBOHandler クラス

CWConnectorBOHandler クラスは、Java コネクターのビジネス・オブジェクト・ハンドラー用基底クラスです。このクラスは、1 つのビジネス・オブジェクト・ハンドラーを実装し、これにアクセスするためのコードを提供します。コネクター開発者は、このクラスからビジネス・オブジェクト・ハンドラー・クラスを (必要な数だけ) 派生し、そのビジネス・オブジェクト・ハンドラーに doVerbFor() メソッドを実装する必要があります。

注: CWConnectorBOHandler クラスは、下位の Java コネクター・ライブラリーの BOHandlerBase クラスの拡張です。下位の Java コネクター・ライブラリーのクラスの詳細については、465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

重要: すべての Java コネクターは、このクラスを拡張する必要があります。ConnectorBOHandler という名前は、派生ビジネス・オブジェクト・ハンドラー・クラスのデフォルトの名前です。開発者は、派生ビジネス・オブジェクト・ハンドラー・クラスについて、このデフォルトの名前を使用できる他、別の名前を選択することもできます。また開発者は、クラス名にかかわらず、派生したビジネス・オブジェクト・ハンドラー・クラスに単一のメソッド doVerbFor() を実装する必要があります。コネクターが要求処理機能を提供している場合には、コネクターが取り扱うビジネス・オブジェクト (1 つまたは複数) 用としてサポートされているすべての動詞を、doVerbFor() メソッドにより処理できることが必要です。コネクターが要求処理機能を提供しない 場合でも、retrieve 動詞を処理することは必要です。

コネクターは、ビジネス・オブジェクト内の動詞に対応するタスクを実行するために、1 つ以上のビジネス・オブジェクト・ハンドラーを持っています。ビジネス・オブジェクト・ハンドラーは、ビジネス・オブジェクト・データのアプリケーションへの挿入、データの取得、アプリケーション・データの削除、またはその他のタスクを実行しますが、どの動作を実行するかはアクティブな動詞によります。要求処理とビジネス・オブジェクト・ハンドラーの概要については、27 ページの『要求処理』を参照してください。ビジネス・オブジェクト・ハンドラーの実装方法については、91 ページの『第 4 章 要求処理』を参照してください。

表 120 に、CWConnectorBOHandler クラスのメソッドの概要を示します。

表 120. CWConnectorBOHandler クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorBOHandler()	ビジネス・オブジェクト・ハンドラー・オブジェクトを作成します。	288
doVerbFor()	ビジネス・オブジェクトのアクティブな動詞のために、動詞処理を実行します。	288
getName()	ビジネス・オブジェクト・ハンドラー・オブジェクトの名前を取得します。	290
setName()	ビジネス・オブジェクト・ハンドラー・オブジェクトの名前を設定します。	291

CWConnectorBOHandler()

ビジネス・オブジェクト・ハンドラー・オブジェクトを作成します。

構文

```
public CWConnectorBOHandler();
```

パラメーター

なし。

戻り値

新しく作成されたビジネス・オブジェクト・ハンドラー・オブジェクトを格納している CWConnectorBOHandler オブジェクト。

注記

CWConnectorBOHandler() コンストラクターは、CWConnectorBOHandler クラスのインスタンスを作成します。ビジネス・オブジェクト内の動詞のタスクを実行するため、このクラスに、ビジネス・オブジェクト定義から参照することができます。通常、コネクター開発者は、CWConnectorBOHandler からクラスを派生させ、その派生クラス用として doVerbFor() メソッドを実装します。開発者は、この派生クラスのコンストラクターを、CWConnectorAgent クラスの getConnectorBOHandlerForBO() メソッドの中で呼び出すことにより、1 つ以上のビジネス・オブジェクト・ハンドラーのインスタンスを作成することができます。

参照項目

getConnectorBOHandlerForBO()

doVerbFor()

ビジネス・オブジェクトのアクティブな動詞のために、動詞処理を実行します。

構文

```
public int doVerbFor(CWConnectorBusObj theBusObj);
```

パラメーター

theBusObj 処理対象のアクティブ動詞が所属するビジネス・オブジェクトです。

戻り値

動詞操作の結果状況を示す整数です。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

CWConnectorConstant.SUCCEED
: 動詞操作は成功しました。

CWConnectorConstant.FAIL
: 動詞操作は失敗しました。

CWConnectorConstant.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

CWConnectorConstant.VALCHANGE
: ビジネス・オブジェクトの 1 つ以上の値が変更されました。

CWConnectorConstant.VALDUPES
: 要求された操作の実行中に、同じキー値を持つ複数のレコードがアプリケーション・データベース内に見つかりました。

CWConnectorConstant.MULTIPLE_HITS
: コネクターが非キー値を使用して取得中に複数の一致レコードを検出しました。コネクターは最初の一致レコード用のビジネス・オブジェクトのみを戻します。

CWConnectorConstant.RETRIEVEBYCONTENT_FAILED
: コネクターは、非キー値による取得で、一致レコードを検出できませんでした。

CWConnectorConstant.BO_DOES_NOT_EXIST
: コネクターは取得動作を実行しましたが、要求されたビジネス・オブジェクトに対応する一致エンティティーがアプリケーション・データベースにありません。

例外

ConnectionFailureException
: コネクターがアプリケーションから切断されるとスローされます。

VerbProcessingFailedException
: 動詞処理が失敗した場合にスローされます。

注記

doVerbFor() メソッドは、*theBusObj* ビジネス・オブジェクト内のアクティブな動詞のアクションを実行します。このメソッドは、ビジネス・オブジェクト・ハンドラーの基本パブリック・インターフェースです。ただし、コネクター・フレームワークは、ビジネス・オブジェクト・ハンドラーを呼び出すと、実際には、BOHandlerBase クラスから継承した下位の doVerbFor() メソッドを実行します。下位の doVerbFor() メソッドは、コネクター開発者が実装する必要のあるこの doVerbFor() (ビジネス・オブジェクト・ハンドラー・クラス内) を呼び出します。詳細については、195 ページの『戻り状況記述子の取り込み』を参照してください。

重要: CWConnectorBOHandler クラスは、デフォルトでは doVerbFor() メソッドを実装していません。したがって、ビジネス・オブジェクト・ハンドラー・クラスにより、このメソッドを実装することが必要です。

例外を 1 つスローする必要がある場合、doVerbFor() メソッドはまず、例外についての情報を含む例外詳細オブジェクトにデータを設定する必要があります。特に、表 121 に示す状況コードを設定する必要があります。

表 121. doVerbFor() メソッドの例外状況コード

doVerbFor() 例外	例外状況コード
ConnectionFailureException	APPRESPONSETIMEOUT
VerbProcessingFailedException	doVerbFor() によって戻される、同じ結果状況コード

例外詳細オブジェクトを初期化するには、次のステップに従ってください。

- CWConnectorExceptionObject() コンストラクターを使用して、例外詳細オブジェクトを作成します。
- 以下に示す CWConnectorExceptionObject クラスのアクセサー・メソッドを使用して、例外詳細オブジェクトに適切な値を設定します。

setMsg()	情報、警告、またはエラーの戻りメッセージが存在する場合は、そのメッセージを例外詳細オブジェクト内に設定します。
setStatus()	状況戻りコードを設定します。これは、表 121 に示したのと同じ値を持つ整数値です。

例外詳細オブジェクトから、統合ブローカーに戻される戻り状況記述子への情報のコピーは、コネクター・フレームワークによって処理されます。

- doVerbFor() が例外をスローした場合は、その情報がコネクター・フレームワークによってコピーされます。
- doVerbFor() が正常終了した場合は、doVerbFor() が戻した結果状況が、コネクター・フレームワークによってコピーされます。

このメソッドの実装方法については、 179 ページの『doVerbFor() メソッドの実装』を参照してください。

参照項目

setErrorString(),, setStatus()

getName()

ビジネス・オブジェクト・ハンドラー・オブジェクトの名前を取得します。

構文

```
protected String getName();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト・ハンドラー (CWConnectorBOHandler) オブジェクトに割り当てられた名前を格納している String です。このメソッドの前に CWConnectorBOHandler オブジェクトに対して setName() を呼び出していなかった場合には、 null が戻ります。

例外

なし。

参照項目

setName()

---

setName()

ビジネス・オブジェクト・ハンドラー・オブジェクトの名前を設定します。

構文

```
protected void setName(String name);
```

パラメーター

name CWConnectorBOHandler オブジェクトの名前を指定します。

戻り値

なし。

例外

なし。

注記

この名前は、通常、ビジネス・オブジェクト・ハンドラーがその処理のために作成されたビジネス・オブジェクトの名前になります。

第 13 章 CWConnectorBusObj クラス

Java コネクター開発者は、CWConnectorBusObj クラスを使用して、ビジネス・オブジェクトを表示することができます。このクラスには、ビジネス・オブジェクト定義、ビジネス・オブジェクトおよびその属性に関する情報を取得するためのメソッドが定義されています。さらに、このクラスには、ビジネス・オブジェクトのメタデータ取得のためのメソッドや、ビジネス・オブジェクトのインスタンスの読み取り、変更のためのメソッドも組み込まれます。 CWConnectorBusObj の各インスタンスは、単一のビジネス・オブジェクトを表します。ビジネス・オブジェクトの操作はいずれも、このクラスから派生するものです。

注: CWConnectorBusObj クラスには、下位の Java コネクター・ライブラリーの BusinessObjectInterface インターフェースに対する内部ハンドルが保管されます。下位の Java コネクター・ライブラリーのクラスの詳細については、465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

CWConnectorBusObj クラス内のメソッドについては、表 122 に要約します。

表 122. CWConnectorBusObj クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
areAllPrimaryKeysTheSame()	指定されたビジネス・オブジェクトの基本キー内の属性値が、現在のビジネス・オブジェクトの基本キー内の属性値と一致しているかどうかを判別します。	298
compare()	動詞セット、属性カウント、ビジネス・オブジェクトのアプリケーション固有の情報、属性および属性値に基づいて、現在のビジネス・オブジェクトと指定されたビジネス・オブジェクトを比較します。	298
doVerbFor()	ビジネス・オブジェクト・ハンドラーを呼び出して、ビジネス・オブジェクト内のアクティブな動詞に対応する動詞の処理を実行します。	299
dump()	ビジネス・オブジェクト情報を、ロギングおよびトレース用の読み取り可能なフォーマットで戻します。	301
getAppText()	このビジネス・オブジェクト定義または指定された属性のどちらかに関連付けられた AppSpecificInfo フィールドの値を取得します。	301
getAttrASIHashtable()	属性の名前、またはビジネス・オブジェクト内の属性リスト内でのその属性の位置を与えることで、ビジネス・オブジェクト内の属性のアプリケーション固有の情報を解析して、名前と値のペアを生成します。	303
getAttrCount()	ビジネス・オブジェクトの属性リスト内に存在する属性の個数を取得します。	304
getAttrIndex()	ビジネス・オブジェクトの指定された属性の序数位置を取得します。	304
getAttrName()	ビジネス・オブジェクトの属性リスト内におけるその位置によって、指定された属性の名前を取得します。	305

表 122. *CWConnectorBusObj クラスのメンバー・メソッド (続き)*

メンバー・メソッド	説明	ページ
getbooleanValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の boolean 属性の値を取得します。	305
getBusinessObjectVersion()	ビジネス・オブジェクト定義のバージョンを取得します。	306
getBusObjASIHashtable()	ビジネス・オブジェクト定義のアプリケーション固有の情報を解析して、名前と値のペアを生成します。	307
getBusObjValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクトが含まれている属性の値を取得します。	307
getCardinality()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のカーディナリティーを取得します。	308
getDefault()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、この属性のデフォルト値を取得します。	309
getDefaultboolean()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の double 属性のデフォルト値を取得します。	323
getDefaultdouble()	ビジネス・オブジェクト定義のバージョンを取得します。	310
getDefaultfloat()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の float 属性のデフォルト値を取得します。	311
getDefaultint()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の int 属性のデフォルト値を取得します。	312
getDefaultlong()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の long 属性のデフォルト値を取得します。	313
getDefaultString()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の String 属性のデフォルト値を取得します。	314
getdoubleValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の double 属性の値を取得します。	314
getfloatValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の float 属性の値を取得します。	315
getintValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の int 属性の値を取得します。	316
getLocale()	ビジネス・オブジェクトに関連付けられているロケールを取得します。	317
getlongValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の long 属性の値を取得します。	318

表 122. *CWConnectorBusObj* クラスのメンバー・メソッド *(続き)*

メンバー・メソッド	説明	ページ
getLongTextValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `longText` 属性の値を取得します。	317
getMaxLength()	ビジネス・オブジェクト定義から属性の最大長を取得します。	319
getName()	現在のビジネス・オブジェクトの参照先となるビジネス・オブジェクト定義の名前を取得します。	319
getObjectCount()	ビジネス・オブジェクト配列になっている属性の中にある子ビジネス・オブジェクトの数を取得します。	320
getParentBusinessObject()	現在のビジネス・オブジェクトの親ビジネス・オブジェクトを取得します。	320
getStringValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `String` 属性の値を取得します。	321
getSupportedVerbs()	現在のビジネス・オブジェクトでサポートされている動詞を取得します。	322
getTypeName()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のデータ型の名前を取得します。	322
getTypeNum()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のデータ型に対応する数値型コードを取得します。	323
getVerb()	ビジネス・オブジェクトのアクティブな動詞を取得します。	324
getVerbAppText()	特定の動詞の `AppSpecificInfo` フィールドの値を取得します。	325
hasAllKeys()	現在のビジネス・オブジェクトが、そのすべての 基本キー属性および外部キー属性の値を持っているかどうかを判別します。	325
hasAllPrimaryKeys()	現在のビジネス・オブジェクトが、そのすべての 基本キー属性の値を持っているかどうかを判別します。	326
hasAnyActivePrimaryKey()	現在のビジネス・オブジェクトがいずれかの 基本キー属性の値を持つかどうかを判別します。	327
hasCardinality()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、指定したカーディナリティー値と同じカーディナリティーをその属性が持っているかどうかを判別します。	328
hasName()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性が指定された名前と一致しているかどうかを判別します。	328
hasType()	属性のデータ型が指定されたデータ型の名前と一致しているかどうかを判別します。	329
isBlank()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの外部キーの一部となっているかどうかを判別します。	330
isForeignKeyAttr()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの外部キーの一部となっているかどうかを判別します。	330

表 122. CWConnectorBusObj クラスのメンバー・メソッド (続き)

メンバー・メソッド	説明	ページ
isIgnore()	指定された名前を持つ属性の値、または属性リストの中の指定された位置にある属性の値が、特殊な Ignore 値かどうかを判別します。	331
isKeyAttr()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの基本キーの一部となっているかどうかを判別します。	332
isMultipleCard()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性が複数カーディナリティーを持っているかどうかを判別します。	332
isObjectType()	属性のデータ型がオブジェクト型あるいは複合属性 (配列またはサブオブジェクト) かどうかを判別します。	333
isRequiredAttr()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの必須属性かどうかを判別します。属性が必須の場合は、値を持つ必要があります。	333
isType()	属性値のデータ型が指定された値と同じデータ型かどうかを判別します。	334
isVerbSupported()	このビジネス・オブジェクト定義がメソッドに渡された動詞をサポート対象としているかどうかを判別します。	334
objectClone()	既存のビジネス・オブジェクトをコピーします。	335
prune()	現在の (親) ビジネス・オブジェクトから子ビジネス・オブジェクトを除去して、子ビジネス・オブジェクトの属性を `null` に設定します。	335
removeAllObjects()	ビジネス・オブジェクト配列になっている属性の中の子ビジネス・オブジェクトをすべて除去します。	336
removeBusinessObjectAt()	ビジネス・オブジェクト配列の中の指定された位置にある子ビジネス・オブジェクトを除去します。	337
setAttrValues()	ベクトル内の値に基づいて、現在のビジネス・オブジェクトの属性を設定します。	337
setbooleanValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、`boolean` 属性の値を、指定された値に設定します。	338
setBusObjValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクトが含まれている属性の値を、指定された値に設定します。	339
setDefaultAttrValues()	現在 Blank または Ignore の属性値を持っている属性に対して、デフォルト値を設定します。	341
setdoubleValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、`double` 属性の値を、指定された値に設定します。	341
setfloatValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、`float` 属性の値を、指定された値に設定します。	342
setintValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、`int` 属性の値を、指定された値に設定します。	343
setLocale()	ビジネス・オブジェクトに関連したロケールを設定します。	344

表 122. CWConnectorBusObj クラスのメンバー・メソッド (続き)

メンバー・メソッド	説明	ページ
setLongTextValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、longText 属性の値を、指定された値に設定します。	345
setStringValue()	属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、String 属性の値を、指定された値に設定します。	345
setVerb()	ビジネス・オブジェクトのアクティブな動詞を設定します。	346

表 122 に示すように、CWConnectorBusObj クラスによって、以下のビジネス・オブジェクトの情報が単一のクラスに結合されます。

- ビジネス・オブジェクト定義およびビジネス・オブジェクト

compare()	getVerb()
doVerbFor()	getVerbAppText()
dump()	hasAnyActivePrimaryKey()
getAppText()	hasAllKeys()
getAttrCount()	hasAnyActivePrimaryKey()
getAttrIndex()	hasName()
getAttrName()	isVerbSupported()
getBusinessObjectVersion()	objectClone()
getBusObjASIHashtable()	prune()
getlongValue()	setAttrValues()
getName()	setVerb()
getParentBusinessObject()	

- ビジネス・オブジェクト配列

getObjectCount()	removeBusinessObjectAt()
removeAllObjects()	

- ビジネス・オブジェクト属性

areAllPrimaryKeysTheSame()	getTypeNum()
getAppText()	hasCardinality()
getAttrASIHashtable()	hasName()
getbooleanValue()	hasType()
getBusObjValue()	isBlank()
getCardinality()	isForeignKeyAttr()
getDefault()	isIgnore()
getDefaultboolean()	isKeyAttr()
getDefaultdouble()	isMultipleCard()
getDefaultfloat()	isObjectType()
getDefaultint()	isRequiredAttr()
getDefaultlong()	isType()
getdoubleValue()	setbooleanValue()
getfloatValue()	setBusObjValue()
getintValue()	setDefaultAttrValues()
getlongValue()	setdoubleValue()

getMaxLength()	setfloatValue()
getStringValue()	setintValue()
getTypeName()	setStringValue()

areAllPrimaryKeysTheSame()

指定されたビジネス・オブジェクトの基本キー内の属性値が、現在のビジネス・オブジェクトの基本キー内の属性値と一致しているかどうかを判別します。

構文

```
public final boolean areAllPrimaryKeysTheSame(CWConnectorBusObj theBusObj);
```

パラメーター

theBusObj 現在のビジネス・オブジェクトの基本キー値と比較する基本キー値を持つビジネス・オブジェクトです。

戻り値

busObj オブジェクト内の基本キー値がすべて現在のビジネス・オブジェクト内の基本キー値と一致している場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された属性の位置が有効でない場合、スローされます。

`WrongAttributeException`
指定された属性が属性型定数以外の場合、スローされます。

参照項目

hasAnyActivePrimaryKey(),, hasAllKeys(), hasAllPrimaryKeys()

compare()

動詞セット、属性カウント、ビジネス・オブジェクトのアプリケーション固有の情報、属性および属性値に基づいて、現在のビジネス・オブジェクトと指定されたビジネス・オブジェクトを比較します。

構文

```
public boolean compare(CWConnectorBusObj theBusObj);
```

パラメーター

theBusObj 現在のビジネス・オブジェクトと比較するビジネス・オブジェクトです。

戻り値

busObj オブジェクト内の以下の情報がすべて現在のビジネス・オブジェクト内の対応する情報と一致している場合は `true` を戻します。

- アクティブな動詞の値
- ビジネス・オブジェクト定義に対するアプリケーション固有の情報
- 属性カウント
- 属性および属性値

失敗のたびに、メソッドによってメッセージがログに記録され、`false` が戻されます。

例外

`AttributeNotFoundException`
: このビジネス・オブジェクトの定義内に属性が見つからない場合、スローされます。

`WrongAttributeException`
: 比較の対象となる属性の属性型が無効な場合、スローされます。

---

doVerbFor()

ビジネス・オブジェクト・ハンドラーを呼び出して、ビジネス・オブジェクト内のアクティブな動詞に対応する動詞の処理を実行します。

構文

```
public final int doVerbFor(CWConnectorReturnStatusDescriptor rtnStat);
```

パラメーター

rtnStat
: 空の戻り状況ディスクリプター・オブジェクト。`doVerbFor()` メソッドを使用することにより、このメソッドの実行状況を示す状況およびメッセージがこのオブジェクトに取り込まれます。呼び出し側コードは、実行状況へのアクセスを、この戻り状況記述子により可能にしています。

戻り値

動詞操作の結果状況を示す整数です。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

`CWConnectorConstant.SUCCEED`
: 動詞操作は成功しました。

`CWConnectorConstant.FAIL`
: 動詞操作は失敗しました。

`CWConnectorConstant.APPRESPONSETIMEOUT`
: アプリケーションが応答していません。

CWConnectorConstant.VALCHANGE
: ビジネス・オブジェクトの 1 つ以上の値が変更されました。

CWConnectorConstant.VALDUPES
: 要求された操作が、同一キー値に対して複数のレコードを検出しました。

CWConnectorConstant.MULTIPLE_HITS
: コネクターは、非キー値を使用した取得時に、複数の一致レコードを検出しています。コネクターは、ビジネス・オブジェクト内で最初に一致したレコードのみを戻します。

CWConnectorConstant.RETRIEVEBYCONTENT_FAILED
: コネクターは、非キー値による取得で、一致レコードを検出できませんでした。

CWConnectorConstant.BO_DOES_NOT_EXIST
: 要求されたビジネス・オブジェクト・エンティティーは、データベースに存在しません。

例外

なし。

注記

doVerbFor() メソッドは、ビジネス・オブジェクト・ハンドラー (CWConnectorBOHandler オブジェクト) を呼び出し、ビジネス・オブジェクト内のアクティブな動詞によって指定されたアクションを実行します。ビジネス・オブジェクト定義でサポートされている動詞に対する操作はすべて、ビジネス・オブジェクト・ハンドラーによって提供されています。アクティブな動詞は、ビジネス・オブジェクト定義に含まれている動詞のリストのうちの 1 つです。ビジネス・オブジェクト用のアクティブな動詞を判別するには、getVerb() メソッドを使用します。

doVerbFor() メソッド内部では、渡された空の *rtnStat* 戻り状況記述子に、動詞処理の実行状況を示す状況およびメッセージが取り込まれます。その後で、呼び出し側コードは、CWConnectorReturnStatusDescriptor クラスのアクセサー・メソッドを使用して、データが取り込まれた戻り状況記述子から動詞処理についての実行の情報を取得できるようになります。

イベントのアプリケーション情報を取得するには通常、この doVerbFor() メソッドをコネクター・クラス (CWConnectorAgent) の pollForEvents() メソッドから呼び出します。デフォルト実装の pollForEvents() が CWConnectorEventStore クラスの getBO() メソッドを呼び出すことにより、アプリケーション情報が取得されます。 getBO() メソッドからは、CWConnectorBusObj クラスの doVerbFor() メソッドが呼び出されます。 getBO() を pollForEvents() メソッド内に使用しない場合、インスタンス化済みの戻り状況記述子を渡し入れることにより doVerbFor() を pollForEvents() から直接呼び出すことも可能です。その後で、doVerbFor() が終了したら、データが取り込まれた戻り状況記述子から動詞処理の状況を取得することができます。

参照項目

doVerbFor() (CWConnectorBOHandler)、getVerb()、pollForEvents()、setVerb()

dump()

ビジネス・オブジェクト情報を、ロギングおよびトレース用の読み取り可能なフォーマットで戻します。

構文

```
public String dump();
```

パラメーター

なし。

戻り値

フォーマット済みのビジネス・オブジェクト情報を含む String です。

例外

なし。

getAppText()

このビジネス・オブジェクト定義または指定された属性のどちらかに関連付けられた AppSpecificInfo フィールドの値を取得します。

構文

```
public String getAppText();
public String getAppText(String attrName);
public String getAppText(int position);

public final String getAppText(String tagName, String delimiter);
public final String getAppText(String attrName, String tagName,
      String delimiter);
public final String getAppText(int position, String tagName,
      String delimiter);
```

パラメーター

attrName 解析対象のアプリケーション固有の情報を持った属性の名前です。

delimiter 名前と値のペアの間に置かれる区切り文字です。慣例により、名前と値のペアを作成する場合は、区切り文字としてコロン (:) を使用します。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

tagName メソッドが受け取る値を含むアプリケーション固有の情報内のタグの名前です。

戻り値

該当する AppSpecificInfo フィールドからのアプリケーション固有の情報を保持する String オブジェクト。

- getAppText() の最初の形式は、現在のビジネス・オブジェクトに関連付けられているビジネス・オブジェクト定義のアプリケーション固有の情報を取得します。ビジネス・オブジェクト定義のアプリケーション固有の情報が存在していない場合、このメソッドは null を戻すことがあります。
- getAppText() の 2 番目と 3 番目の形式は、名前の指定またはビジネス・オブジェクト定義内での位置の指定が可能な、属性のアプリケーション固有の情報を取得します。その属性のアプリケーション固有の情報が存在していない場合、このメソッドは null を戻すことがあります。

例外

getAppText() メソッドの 2 番目、3 番目、5 番目、および 6 番目の形式は、次の例外をスローできます。

AttributeNotFoundException
指定された属性が見つからない場合にスローされます。

getAppText() メソッドの 4 番目、5 番目、および 6 番目の形式は、次の例外をスローできます。

WrongASIFormatException
アプリケーション固有の情報が、名前と値の形式に適合していない場合、スローされます。

注記

getAppText() メソッドの形式には、以下の種類があります。

- このメソッドの最初の形式は、ビジネス・オブジェクト・レベルのアプリケーション固有の情報を取得します。つまり、現在のビジネス・オブジェクトに関連付けられているビジネス・オブジェクト定義について、アプリケーション固有の情報を取得します。
- このメソッドの 2 番目と 3 番目の形式は、属性のアプリケーション固有の情報を取得します。つまり、名前 (*attrName*) またはビジネス・オブジェクト定義内での位置 (*position*) を通じて識別可能な属性について、アプリケーション固有の情報を取得します。
- 4 番目、5 番目、および 6 番目の形式は、当該情報が次のような名前と値のペアの形式にフォーマットされているときに、アプリケーション固有の情報を取得します。

  tagName=value

  tagName は、アプリケーション固有の情報内で使用するタグ (プロパティー) の名前を指定します。*delimiter* は、それぞれの名前と値のペアを分離する記号を指定します。慣例により、区切り文字には通常コロン (:) を使用します。 4 番目の形式はビジネス・オブジェクト・レベルのアプリケーション固有の情報から名前と値のペアを取得しますが、5 番目と 6 番目の形式は指定された属性のアプリケーション固有の情報から名前と値のペアを取得します。

例えば、ビジネス・オブジェクト定義に次のアプリケーション固有の情報が含まれていると仮定します。

```
TN=table1:SCH=schema1
```

次のように getAppText() を呼び出すと、TN タグの名前と値のペアについて値が取得されます。

```
String TNvalue = busObj.getAppText("TN", ":");
```

注: すべての 名前と値のペアを Java Hashtable オブジェクトとして取得するには、ビジネス・オブジェクト・レベルの場合は getBusObjASIHashtable() メソッド、属性のアプリケーション固有の情報の場合は getAttrASIHashtable() メソッドを使用します。

参照項目

getAttrASIHashtable(),, getBusObjASIHashtable(),, getVerbAppText()

---

getAttrASIHashtable()

属性の名前、またはビジネス・オブジェクト内の属性リスト内でのその属性の位置を与えることで、ビジネス・オブジェクト内の属性のアプリケーション固有の情報を解析して、名前と値のペアを生成します。

構文

```
public final Hashtable getAttrASIHashtable(int attrName,
   String delimiter);
public final Hashtable getAttrASIHashtable(int position,
   String delimiter);
```

パラメーター

attrName 解析対象のアプリケーション固有の情報を持った属性の名前です。

delimiter 名前と値のペアの間に置かれる区切り文字です。名前と値のペアを作成する場合、区切り文字としてコロン (:) を使用します。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のアプリケーション固有の情報の中に名前と値のペアを含んだ java.util.Hashtable オブジェクト。

例外

AttributeNotFoundException
指定された属性が見つからない場合にスローされます。例えば、このビジネス・オブジェクト定義に指定された位置が有効でない場合です。

WrongASIFormatException
アプリケーション固有の情報が、名前と値の形式に適合していない場合、スローされます。

注記

getAttrASIHashtable() メソッドは、任意の属性のアプリケーション固有の情報を解析して、名前と値のペアのハッシュ・テーブルを戻します。例えば、名前と値のペアは次のような形式になります。

```
ASI=CN=colname:FK=attr1:UID=attr2:...
```

この例では、区切り文字としてコロン (:) を指定することを想定しています。

注: 属性のアプリケーション固有の情報から特定の名前と値のペアを *1* つ取得する場合は、getAppText() メソッドを使用します。

参照項目

getAppText(),, getBusObjASIHashtable()

---

getAttrCount()

ビジネス・オブジェクトの属性リスト内に存在する属性の個数を取得します。

構文

```
public int getAttrCount();
```

パラメーター

なし。

戻り値

属性リスト内の属性の個数を示す整数です。

例外

なし。

参照項目

getAttrIndex()

---

getAttrIndex()

ビジネス・オブジェクトの指定された属性の序数位置を取得します。

構文

```
public int getAttrIndex(String attrName);
```

パラメーター

attrName ビジネス・オブジェクト定義での属性の名前です。

戻り値

ビジネス・オブジェクト定義の内部にある属性の序数位置です。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された属性の名前が有効でない場合、スローされます。

---

getAttrName()

ビジネス・オブジェクトの属性リスト内におけるその位置によって、指定された属性の名前を取得します。

構文

```
public String getAttrName(int position);
```

パラメーター

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の名前。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された属性の位置が有効でない場合、スローされます。

---

getbooleanValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の boolean 属性の値を取得します。

構文

```
public boolean getbooleanValue(String attrName);
public boolean getbooleanValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `boolean` 値。

例外

`WrongAttributeException`
ブール以外の属性に対してメソッドが呼び出された場合、スローされます。

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeNullValueException`
指定された属性が `null` を値として持つ場合、スローされます。

参照項目

getAttrName(),, getBusObjValue(),, getDefaultboolean(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setbooleanValue()

---

getBusinessObjectVersion()

ビジネス・オブジェクト定義のバージョンを取得します。

構文

```
public String getBusinessObjectVersion();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトのバージョン番号。

例外

なし。

注記

バージョンは、大と小の改訂番号および点の要素 (-x.y.z) によって表されます。例えば、- 1.0.2 のようになります。

getBusObjASIHashtable()

ビジネス・オブジェクト定義のアプリケーション固有の情報を解析して、名前と値のペアを生成します。

構文

```
public Hashtable getBusObjASIHashtable(String delimiter);
```

パラメーター

delimiter 名前と値のペアの間に置かれる区切り文字です。名前と値のペアを作成する場合、区切り文字としてコロン (:) を使用します。

戻り値

ビジネス・オブジェクト定義のアプリケーション固有の情報の中に名前と値のペアを含んだ `java.util.Hashtable` オブジェクト。

例外

`WrongASIFormatException`
アプリケーション固有の情報が、名前と値のペア形式に適合していない場合、スローされます。

注記

`getBusObjASIHashtable()` メソッドは、現在のビジネス・オブジェクトに関連付けられているビジネス・オブジェクト定義についてアプリケーション固有の情報の構文を解析し、名前と値のペアのハッシュ・テーブルを戻します。例えば、名前と値のペアは次のような形式になります。

```
ASI=CN=colname:FK=attr1:UID=attr2:...
```

この例では、区切り文字としてコロン (:) を指定することを想定しています。

注: ビジネス・オブジェクト・レベルのアプリケーション固有の情報から特定の名前と値のペアを *1* つ取得する場合は、`getAppText()` メソッドを使用します。

参照項目

getAppText(),, getAttrASIHashtable()

getBusObjValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクトが含まれている属性の値を取得します。

構文

```
public CWConnectorBusObj getBusObjValue(String attrName);
public CWConnectorBusObj getBusObjValue(int position);
```

```
public CWConnectorBusObj getBusObjValue(String attrName,
      int arrayIndex);
public CWConnectorBusObj getBusObjValue(int position,
      int arrayIndex);
```

パラメーター

attrName 値が取得される属性の名前です。

arrayIndex ビジネス・オブジェクト配列 (属性にビジネス・オブジェクト配列が含まれる場合) 内でのビジネス・オブジェクトの序数位置を指定する整数です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性に含まれているビジネス・オブジェクト。

例外

`WrongAttributeException`
ビジネス・オブジェクト以外の属性に対してメソッドが呼び出された場合、スローされます。

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

`getBusObjValue()` メソッドの形式には、以下の 2 種類があります。

- 最初の形式の場合、オブジェクト型の属性の名前または位置を指定する必要があります。この場合、指定された属性のビジネス・オブジェクトが戻されます。この形式の場合、属性が単一カーディナリティーを持つことが前提になります。
- 2 番目の形式の場合、属性の名前または位置のどちらかと、ビジネス・オブジェクト配列への索引を指定する必要があります。この形式の場合、ビジネス・オブジェクト配列内の指定された索引位置にある子ビジネス・オブジェクトが戻されます。この形式の場合、属性が複数カーディナリティーを持つことが前提になります。

参照項目

getAttrName(),, getbooleanValue(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getParentBusinessObject(),, getObjectCount(),, getStringValue(),, setBusObjValue()

---

getCardinality()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のカーディナリティーを取得します。

構文

```
public String getCardinality(String attrName);
public String getCardinality(int position);
```

パラメーター

attrName 取得対象のカーディナリティーを持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のカーディナリティーが格納されている String です。ストリングの値は以下のいずれかです。

1 属性は単一カーディナリティーを持ちます。この場合、属性は単純属性です。

n 属性は複数カーディナリティーを持ちます。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

hasCardinality(),, isMultipleCard()

---

getDefault()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、この属性のデフォルト値を取得します。

構文

```
public String getDefault(String attrName);
public String getDefault(int position);
```

パラメーター

attrName 取得対象のデフォルト値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のデフォルト値が格納されている String。属性のデフォルト値が存在していない場合、メソッドは空ストリングを戻します。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

getDefaultboolean()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の boolean 属性のデフォルト値を取得します。

構文

```
public boolean getDefaultboolean(String attrName);
public boolean getDefaultboolean(int position);
```

パラメーター

attrName 取得対象のデフォルト値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (boolean 値)、または null 。

例外

WrongAttributeException
ブール以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
指定された属性が null を値として持つ場合、スローされます。

参照項目

getbooleanValue(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString()

getDefaultdouble()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の double 属性のデフォルト値を取得します。

構文

```
public double getDefaultdouble(String attrName);
public double getDefaultdouble(int position);
```

パラメーター

attrName 取得対象のデフォルト値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (double の値)、または null 。

例外

WrongAttributeException
: double 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
: 指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
: デフォルト値が正しい形式でない場合、スローされます。

参照項目

getDefaultboolean(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString(),, getdoubleValue(),

---

getDefaultfloat()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の float 属性のデフォルト値を取得します。

構文

```
public float getDefaultfloat(String attrName);
public float getDefaultfloat(int position);
```

パラメーター

attrName 取得対象のデフォルト値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (float の値)、または null 。

例外

WrongAttributeException
: float 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
: 指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
: デフォルト値が正しい形式でない場合、スローされます。

参照項目

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getDefaultString(),, getfloatValue()

---

getDefaultint()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の int 属性のデフォルト値を取得します。

構文

```
public int getDefaultint(String attrName);
public int getDefaultint(int position);
```

パラメーター

attrName
: 取得対象のデフォルト値を持った属性の名前です。

position
: ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (int の値)、または null 。

例外

WrongAttributeException
: int 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
: 指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
: デフォルト値が正しい形式でない場合、スローされます。

参照項目

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultlong(),, getDefaultString(),, getintValue()

---

getDefaultlong()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の long 属性のデフォルト値を取得します。

構文

```
public long getDefaultlong(String attrName);
public long getDefaultlong(int position);
```

パラメーター

attrName
: 取得対象のデフォルト値を持った属性の名前です。

position
: ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (long の値)、または null。

例外

WrongAttributeException
: long 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
: 指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
: デフォルト値が正しい形式でない場合、スローされます。

参照項目

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultlong(),, getDefaultString(),, getintValue()

getDefaultString()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の String 属性のデフォルト値を取得します。

構文

```
public String getDefaultString(String attrName);
public String getDefaultString(int position);
```

パラメーター

attrName 取得対象のデフォルト値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性のデフォルト値 (String の値)、または null。

例外

WrongAttributeException
String 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getDefaultboolean(),, getDefaultdouble(),, getDefaultfloat(),, getDefaultint(),, getDefaultlong(),, getStringValue()

getdoubleValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の double 属性の値を取得します。

構文

```
public double getdoubleValue(String attrName);
public double getdoubleValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `double` の値。

例外

`WrongAttributeException`
: `double` 以外の属性に対してメソッドが呼び出された場合、スローされます。

`AttributeNotFoundException`
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeNullValueException`
: 指定された属性が `null` を値として持つ場合、スローされます。

`AttributeValueException`
: `double` 値が正しい形式でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultdouble(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setdoubleValue()

---

getfloatValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `float` 属性の値を取得します。

構文

```
public float getfloatValue(String attrName);
public float getfloatValue(int position);
```

パラメーター

attrName
: 値が取得される属性の名前です。

position
: ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `float` の値。

例外

`WrongAttributeException`
: `float` 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
float 値が正しい形式でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultfloat(),, getdoubleValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setfloatValue()

---

getintValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の int 属性の値を取得します。

構文

```
public int getintValue(String attrName);
public int getintValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の int の値。

例外

WrongAttributeException
int 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
int 値が正しい形式でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultint(),, getdoubleValue(),, getfloatValue(),, getlongValue(),, getLongTextValue(),, getStringValue(),, setintValue()

---

getLocale()

ビジネス・オブジェクトに関連付けられているロケールを取得します。

構文

```
public String getLocale();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトに関連付けられているロケールの名前が格納されている `String`。

例外

なし。

注記

`getLocale()` メソッドは、そのビジネス・オブジェクトに関連したビジネス・オブジェクト・ロケールを戻します。このロケールは、ビジネス・オブジェクト内のデータに関連した言語およびコードのエンコードを示すものであり、ビジネス・オブジェクト定義の名前またはその属性 (これは、英語 (U.S.) ロケール `en_US` に関連したコード・セット内の文字でなければなりません) に関連するものではありません。ビジネス・オブジェクトにロケールが関連付けられていない場合、コネクター・フレームワークは、コネクター・フレームワークのロケールをビジネス・オブジェクトのロケールとして割り当てます。

参照項目

createBusObj(),, getGlobalLocale(),, setLocale()

---

getLongTextValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `LongText` 属性の値を取得します。

構文

```
public String getLongTextValue(String attrName);
public String getLongTextValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `LongText` 値を含む `String`。

例外

`WrongAttributeException`
`LongText` 以外の属性に対してメソッドが呼び出された場合、スローされます。

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultlong(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(), getStringValue(),, setLongTextValue()

---

getlongValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `long` 属性の値を取得します。

構文

```
public long getlongValue(String attrName);
public long getlongValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `long` の値。

例外

`WrongAttributeException`
`long` 以外の属性に対してメソッドが呼び出された場合、スローされます。

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeNullValueException
指定された属性が null を値として持つ場合、スローされます。

AttributeValueException
long 値が正しい形式でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultlong(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getLongTextValue(),, getStringValue()

---

getMaxLength()

ビジネス・オブジェクト定義から属性の最大長を取得します。

構文

```
public int getMaxLength(String attrName);
public int getMaxLength(int position);
```

パラメーター

attrName 取得対象の最大長を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性値が取ることのできる最大長をバイト単位で示す整数です。

例外

AttributeNotFound
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

InvalidAttributePropertyException
オブジェクト型の属性に対してメソッドが呼び出された場合、スローされます。

---

getName()

現在のビジネス・オブジェクトの参照先となるビジネス・オブジェクト定義の名前を取得します。

構文

```
public String getName();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義の名前が格納されている `String` オブジェクト。

例外

なし。

参照項目

getBusinessObjectVersion()

---

getObjectCount()

ビジネス・オブジェクト配列になっている属性の中にある子ビジネス・オブジェクトの数を取得します。

構文

```
public int getObjectCount(String attrName);
public int getObjectCount(int position);
```

パラメーター

attrName 計数対象の子オブジェクトを持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの個数を示す整数です。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getBusObjValue()

---

getParentBusinessObject()

現在のビジネス・オブジェクトの親ビジネス・オブジェクトを取得します。

構文

```
public CWConnectorBusObj getParentBusinessObject();
```

パラメーター

なし。

戻り値

親ビジネス・オブジェクトが含まれているビジネス・オブジェクト。ただし、現在のビジネス・オブジェクトがルートとなっていて、しかも親を持たない場合は `null`。

例外

なし。

参照項目

getBusObjValue()

---

getStringValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクト内の `String` 属性の値を取得します。

構文

```
public String getStringValue(String attrName);
public String getStringValue(int position);
```

パラメーター

attrName 値が取得される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の `String` の値。

例外

`WrongAttributeException`
オブジェクト型以外の属性に対してメソッドが呼び出された場合、スローされます。

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getAttrName(),, getbooleanValue(),, getBusObjValue(),, getDefaultString(),, getdoubleValue(),, getfloatValue(),, getintValue(),, getlongValue(),, getLongTextValue(),, setStringValue()

---

getSupportedVerbs()

現在のビジネス・オブジェクトによって使用されている動詞のリストを取得します。

構文

```
public String[] getSupportedVerbs();
```

パラメーター

なし。

戻り値

String オブジェクトの配列。各配列には、ビジネス・オブジェクトでサポートされる動詞が含まれています。これらの String 値を以下の動詞定数と比較してください。

CWConnectorConstant.VERB_CREATE
: Create 動詞のストリング表現です。

CWConnectorConstant.RETRIEVE
: Retrieve 動詞のストリング表現です。

CWConnectorConstant.UPDATE
: Update 動詞のストリング表現です。

CWConnectorConstant.DELETE
: Delete 動詞のストリング表現です。

ご使用のアプリケーションで他の動詞をサポートしている場合は、それらの動詞を表すために独自の動詞定数を作成してください。

参照項目

getVerb(),, isVerbSupported()

---

getTypeName()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のデータ型の名前を取得します。

構文

```
public String getTypeName(String attrName);
public String getTypeName(int position);
```

パラメーター

attrName 取得対象のデータ型 (ストリング値) を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のデータ型の名前が格納されている String です。この String 値を表 123 に示す属性型定数と比較することにより、型が判別されます。

表 123. ストリング属性型定数

属性のデータ型	ストリング属性型定数
ブール値	BOOLSTRING
ビジネス・オブジェクト: 複数カーディナリティー	MULTIPLECARDSTRING
ビジネス・オブジェクト: 単一カーディナリティー	SINGLECARDSTRING
	CIPHERTEXTSTRING
日付	DATESTRING
倍精度	DOUBSTRING
浮動小数点	FLTSTRING
整数	INTSTRING
無効なデータ型	INVALID_TYPE_STRING
ロング・テキスト	LONGTEXTSTRING
ストリング	STRSTRING

注: 表 123 にリストされているストリングの属性型定数は、CWConnectorAttrType クラスによって定義されます。

例外

AttributeNotFoundException

このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getTypeNum(),, hasType()

getTypeNum()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性のデータ型に対応する数値型コードを取得します。

構文

```
public int getTypeNum(String attrName);
public int getTypeNum(int position);
```

パラメーター

attrName 取得対象のデータ型 (数値) を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のデータ型を指定する整数。この整数値を表 124 に示す属性型定数と比較することにより、型が判別されます。

表 124. 数値属性型定数

属性のデータ型	数値属性型定数
ブール	BOOLEAN
	CIPHERTEXT
日付	DATE
倍精度	DOUBLE
浮動小数点	FLOAT
整数	INTEGER
無効なデータ型	INVALID_TYPE_NUM
ロング・テキスト	LONGTEXT
オブジェクト	OBJECT
ストリング	STRING

注: 表 124 にリストされている数値の属性型定数は、CWConnectorAttrType クラスによって定義されます。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getTypeName(),, hasType()

---

getVerb()

ビジネス・オブジェクトのアクティブな動詞を取得します。

構文

```
public String getVerb();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトのアクティブな動詞を含む String オブジェクトです。ビジネス・オブジェクトに対してアクティブな動詞が存在しない場合は、戻り値の String は空となります。

例外

なし。

注記

ビジネス・オブジェクト定義には、ビジネス・オブジェクトによって使用されている動詞のリストが含まれています。getVerb() メソッドを使用すると、現在のビジネス・オブジェクト用のアクティブな動詞の判別が可能になります。

参照項目

isVerbSupported(),, setVerb()

getVerbAppText()

特定の動詞の AppSpecificInfo フィールドの値を取得します。

構文

```
public String getVerbAppText(String verb);
```

パラメーター

verb AppSpecificInfo フィールドの値が取得される動詞です。

戻り値

動詞に関するアプリケーション固有の情報を保持する String オブジェクト。この情報は、指定された動詞の AppSpecificInfo フィールドに格納されます。ビジネス・オブジェクトが動詞に関するアプリケーション固有の情報を持たない場合、メソッドは空ストリングを戻します。

例外

なし。

参照項目

getAppText(),, getVerb()

hasAllKeys()

現在のビジネス・オブジェクトが、そのすべての 基本キー属性および外部キー属性の値を持っているかどうかを判別します。

構文

```
public final boolean hasAllKeys();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトがすべての 基本キー属性および外部キー属性の値を持っている場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

`WrongAttributeException`
: 複数カーディナリティー属性にキーが設定されている場合、スローされます。

`AttributeNotFoundException`
: ビジネス・オブジェクト定義の内部にキー属性が見つからない場合、スローされます。

注記

`hasAllKeys()` メソッドは、基本キーおよび外部キーがすべて取り込まれているかどうかを検査します。このメソッドは一般的に、更新の対象となる行の識別に使用されます。

参照項目

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllPrimaryKeys()

---

hasAllPrimaryKeys()

現在のビジネス・オブジェクトが、そのすべての 基本キー属性の値を持っているかどうかを判別します。

構文

```
public final boolean hasAllPrimaryKeys();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトがすべての 基本キー属性の値を持っている場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

WrongAttributeException
: 複数カーディナリティー属性にキーが設定されている場合、スローされます。

AttributeNotFoundException
: ビジネス・オブジェクト定義の内部に基本キー属性が見つからない場合、スローされます。

注記

hasAllPrimaryKeys() メソッドは、基本キーがすべて取り込まれているかどうかを検査します。このメソッドは一般的に、更新の対象となる行の識別に使用されます。

参照項目

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllKeys()

---

hasAnyActivePrimaryKey()

現在のビジネス・オブジェクトがいずれかの 基本キー属性の値を持つかどうかを判別します。

構文

```
public final boolean hasAnyActivePrimaryKey();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトがいずれかの 基本キー属性の値を持つ場合は true を戻します。それ以外の場合は false を戻します。

例外

WrongAttributeException
: 複数カーディナリティー属性にキーが設定されている場合、スローされます。

AttributeNotFoundException
: ビジネス・オブジェクト定義の内部にキー属性が見つからない場合、スローされます。

注記

hasAnyActivePrimaryKey() メソッドは、基本キー値が少なくとも 1 つは取り込まれているかどうかを検査します。このメソッドは一般的に、削除の対象となる行の識別に使用されます。

参照項目

areAllPrimaryKeysTheSame(),, hasAllKeys(),, hasAllPrimaryKeys()

hasCardinality()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、指定したカーディナリティー値と同じカーディナリティーをその属性が持っているかどうかを判別します。

構文

```
public boolean hasCardinality(String attrName, String card);
public boolean hasCardinality(int position, String card);
```

パラメーター

attrName テスト対象のカーディナリティーを持った属性の名前です。

card 検査に使用されるカーディナリティー値です。無効なカーディナリティー値は、以下のとおりです。

```
1 - single cardinality
n - multiple cardinality
```

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のカーディナリティーが指定された値と一致している場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

このメソッドは、複合属性 (サブオブジェクトおよび配列) のカーディナリティーのテストに使用されます。

参照項目

getCardinality(),, isMultipleCard()

hasName()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性が指定された名前と一致しているかどうかを判別します。

構文

```
public boolean hasName(int position, String name);
```

パラメーター

name 指定された属性の位置にあるテスト対象の属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性の名前が指定された名前と一致している場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

---

hasType()

属性のデータ型が指定されたデータ型の名前と一致しているかどうかを判別します。

構文

```
public boolean hasType(String attrName, int typeName);
public boolean hasType(int position, String typeName);

public boolean hasType(String attrName, int typeNum);
public boolean hasType(int position, String typeNum);
```

パラメーター

attrName テスト対象のカーディナリティーを持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

typeName テスト対象となる属性のデータ型 (ストリング値) です。データ型を指定するには、表 123 のストリングの属性型定数のいずれかを指定してください。

typeNum テスト対象となる属性のデータ型 (数値) です。データ型を指定するには、表 124 の数値の属性型定数のいずれかを指定してください。

戻り値

属性の型が渡された型名と一致している場合は、`true` を戻します。それ以外の場合は `false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getTypeName(),, getTypeNum(),, hasName()

---

isBlank()

指定された名前を持つ属性の値、または属性リストの中の指定された位置にある属性の値が、特殊な Blank 属性値かどうかを判別します。

構文

```
public boolean isBlank(String attrName);
public boolean isBlank(int position);
```

パラメーター

attrName Blank 検査の対象となる値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性値が Blank 値に等しい場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

なし。

参照項目

isIgnore()

---

isForeignKeyAttr()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの外部キーの一部となっているかどうかを判別します。

構文

```
public boolean isForeignKeyAttr(String attrName);
public boolean isForeignKeyAttr(int position);
```

パラメーター

attrName 外部キー内に加わっているかどうか検査される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性がビジネス・オブジェクトの外部キー、またはその外部キーの一部となっている場合は `true` を戻します。それ以外の場合は、`false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

hasAllKeys(),, isKeyAttr()

---

isIgnore()

指定された名前を持つ属性の値、または属性リストの中の指定された位置にある属性の値が、特殊な Ignore 値かどうかを判別します。

構文

```
public boolean isIgnore(String attrName);
public boolean isIgnore(int position);
```

パラメーター

attrName 「ignore」検査の対象となる値を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性値が「ignore」値に等しい場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

なし。

参照項目

isBlank()

isKeyAttr()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの基本キーの一部となっているかどうかを判別します。

構文

```
public boolean isKeyAttr(String attrName);
public boolean isKeyAttr(int position);
```

パラメーター

attrName キー内に加わっているかどうか検査される属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性がビジネス・オブジェクトの基本キー、またはその基本キーの一部となっている場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

areAllPrimaryKeysTheSame(),, hasAnyActivePrimaryKey(),, hasAllKeys(),, hasAllPrimaryKeys(),, isForeignKeyAttr()

isMultipleCard()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性が複数カーディナリティーを持っているかどうかを判別します。

構文

```
public boolean isMultipleCard(String attrName);
public boolean isMultipleCard(int position);
```

パラメーター

attrName 複数カーディナリティーの検査の対象となる属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性が複数カーディナリティーである場合は、true を戻します。それ以外の場合は、false を戻します。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

参照項目

getCardinality(),, hasCardinality()

---

isObjectType()

属性のデータ型がオブジェクト型あるいは複合属性 (配列またはサブオブジェクト) かどうかを判別します。

構文

```
public boolean isObjectType(String attrName);
public boolean isObjectType(int position);
```

パラメーター

attrName オブジェクトのデータ型検査の対象となる属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性がビジネス・オブジェクトあるいは複合属性 (配列やサブオブジェクトなど) の場合は true を戻します。それ以外の場合は false を戻します。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

---

isRequiredAttr()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、属性がビジネス・オブジェクトの必須属性かどうかを判別します。属性が必須の場合は、値を持つ必要があります。

構文

```
public boolean isRequiredAttr(String attrName);
public boolean isRequiredAttr(int position);
```

パラメーター

attrName 必須かどうかの検査の対象となる属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性がビジネス・オブジェクトに必須である場合は、`true` を戻します。それ以外の場合は、`false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

---

isType()

属性値のデータ型が指定された値と同じデータ型かどうかを判別します。

構文

```
public boolean isType(String attrName, Object value);
public boolean isType(int position, Object value);
```

パラメーター

attrName 指定された属性値と比較されるデータ型を持った属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

value 属性値と比較されるデータ型を持った値です。

戻り値

属性の型が渡された型と一致する場合は、`true` を戻します。それ以外の場合は、`false` を戻します。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

---

isVerbSupported()

このビジネス・オブジェクト定義がメソッドに渡された動詞をサポート対象としているかどうかを判別します。

構文

```
public boolean isVerbSupported(String verb);
```

パラメーター

verb 現在のビジネス・オブジェクト定義のサポート対象であるかどうかが、メソッドによって判別される動詞です。

戻り値

指定された動詞がサポート対象である場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

なし。

参照項目

getVerb(),, getSupportedVerbs()

---

objectClone()

既存のビジネス・オブジェクトをコピーします。

構文

```
public CWConnectorBusObj objectClone();
```

パラメーター

なし。

戻り値

その属性と動詞も含め、現在のビジネス・オブジェクトの 1 つのコピーです。

例外

なし。

注記

このメソッドは、ビジネス・オブジェクトの属性のみでなく、その動詞もコピーします。

---

prune()

現在の (親) ビジネス・オブジェクトから子ビジネス・オブジェクトを除去して、子ビジネス・オブジェクトの属性を `null` に設定します。

構文

```
public final void prune();
```

パラメーター

なし。

戻り値

なし。

例外

`AttributeNotFoundException`
: このビジネス・オブジェクトの定義内にオブジェクト型の属性が見つからない場合、スローされます。

`WrongAttributeException`
: 属性が有効でない (オブジェクト型の属性でない) 場合、スローされます。

---

removeAllObjects()

ビジネス・オブジェクト配列になっている属性の中の子ビジネス・オブジェクトをすべて除去します。

構文

```
public void removeAllObjects(String attrName);
public void removeAllObjects(int position);
```

パラメーター

attrName
: ビジネス・オブジェクト配列から除去されるビジネス・オブジェクトを持った属性の名前です。

position
: ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

なし。

例外

`AttributeNotFoundException`
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

removeBusinessObjectAt()

ビジネス・オブジェクト配列の中の指定された位置にある子ビジネス・オブジェクトを除去します。

構文

```
public void removeBusinessObjectAt(String attrName, int index);
public void removeBusinessObjectAt(int position, int index);
```

パラメーター

attrName ビジネス・オブジェクト配列から除去されるビジネス・オブジェクトを持った属性の名前です。

index ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの位置を指定する整数です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

なし。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

除去操作後は、ビジネス・オブジェクト配列が圧縮されます。索引番号が、減らされたビジネス・オブジェクトより大きいビジネス・オブジェクトすべてに対して、その索引が、1 減算されます。

setAttrValues()

ベクトル内の値に基づいて、現在のビジネス・オブジェクトの属性を設定します。

構文

```
public final void setAttrValues(Vector attrValues);
```

パラメーター

attrValues 現在のビジネス・オブジェクト内の各属性の値が格納されている `java.util.Vector` オブジェクトです。

戻り値

なし。

例外

AttributeNotFoundException
: *attrValues* ベクトル内の指定された値が、このビジネス・オブジェクトの定義の中の属性に一切関連付けられていない場合、スローされます。

AttributeValueException
: *attrValues* ベクトル内の属性値が、その属性値に関連付けられた属性のデータ型と互換しない場合、スローされます。

WrongAttributeException
: オブジェクト型の属性に対して値が設定されている場合、スローされます。

---

setbooleanValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、boolean 属性の値を、指定された値に設定します。

構文

```
public void setbooleanValue(String attrName, boolean newVal);
public void setbooleanValue(int position, boolean newVal);
```

パラメーター

attrName
: その値を設定する属性の名前です。

position
: ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal
: 属性に割り当てられる boolean 値です。

戻り値

なし。

例外

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeValueException
: 渡された値が、特定の属性に対して有効な値でない場合、スローされます。

WrongAttributeException
: ブール以外の属性に対して値が設定されている場合、スローされます。

参照項目

getbooleanValue(),, getDefaultboolean(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setBusObjValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、ビジネス・オブジェクトが含まれている属性の値を、指定された値に設定します。

構文

```
public void setBusObjValue(String attrName, CWConnectorBusObj newVal);
public void setBusObjValue(int position, CWConnectorBusObj newVal);

public void setBusObjValue(String attrName, CWConnectorBusObj newVal,
      int arrayIndex);
public void setBusObjValue(int position, CWConnectorBusObj newVal,
      int arrayIndex);
```

パラメーター

attrName その値を設定する属性の名前です。

arrayIndex ビジネス・オブジェクト配列 (属性にビジネス・オブジェクト配列が含まれる場合) 内でのビジネス・オブジェクトの序数位置を指定する整数です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てられる `boolean` 値です。

戻り値

なし。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeNullValueException`
ビジネス・オブジェクトを保持するための (複数カーディナリティー属性の) ビジネス・オブジェクト配列を作成できない場合、スローされます。

`WrongAttributeException`
オブジェクト以外の属性に対して値が設定されている場合、スローされます。

`AttributeValueException`
設定される値が有効なビジネス・オブジェクトではない場合、スローされます。

SpecNameNotFoundException
ビジネス・オブジェクト配列に対するビジネス・オブジェクト定義が見つからない場合、スローされます。この例外は、*arrayIndex* 引き数で渡す setBusObjValue() の形式でのみ戻されます。

注記

setBusObjValue() メソッドの形式には、以下の 2 種類があります。

- 最初の形式の場合、オブジェクト型の属性の名前または位置、およびこの属性に割り当てるビジネス・オブジェクトを指定する必要があります。この形式の場合、属性が単一カーディナリティーを持つことが前提になります。
- 2 番目の形式の場合、以下のものを指定する必要があります。
  - 設定する属性の名前または位置
  - 属性に割り当てるビジネス・オブジェクト
  - ビジネス・オブジェクト配列内の、オブジェクトの値を割り当てる索引位置

  この形式の場合、属性が複数カーディナリティーを持つことが前提になります。

参照項目

getBusObjValue(),, setbooleanValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setDEEId()

指定されたイベント ID を ObjectEventId 属性に設定します。

構文

```
public void setDEEId(String eventId);
```

パラメーター

eventId ObjectEventId 属性に割り当てるイベント ID です。

戻り値

なし。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeValueException
設定される値が有効なビジネス・オブジェクトではない場合、スローされます。

注記

重複イベント回避機能を使用する場合は、ビジネス・オブジェクトはイベント・レコードのイベント ID を自身の ObjectEventId 属性に保管する必要があります。通常、ObjectEventId は統合ブローカーが使用するために予約されます。重複イベント回避機能を使用するためにこの属性にアクセスするには、setDEEId() メソッドを使用します。詳細については、131 ページの『第 5 章 イベント通知』の重複イベント回避の説明を参照してください。

setDefaultAttrValues()

現在 Blank 値または Ignore 値を持っている属性に対して、デフォルト値を設定します。

構文

```
public void setDefaultAttrValues();
```

パラメーター

なし。

戻り値

なし。

例外

なし。

注記

setDefaultAttrValues() メソッドは、Ignore 値ではなく有効値としてデフォルト値を設定します。このメソッドは、複合属性 (型がビジネス・オブジェクトまたはビジネス・オブジェクト配列の場合) 用に空のコンテナーを作成します。ビジネス・オブジェクト内のサブオブジェクトのインスタンス用のデフォルト値は、このメソッドによって設定されます。

参照項目

setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

setdoubleValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、double 属性の値を、指定された値に設定します。

構文

```
public void setdoubleValue(String attrName, double newVal);
public void setdoubleValue(int position, double newVal);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てられる double の値です。

戻り値

なし。

例外

AttributeNotFoundException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeValueException
渡された値が、特定の属性に対して有効な値でない場合、スローされます。

WrongAttributeException
double 以外の属性に対して値が設定されている場合、スローされます。

注記

コネクター固有のプロパティー MaxDoublePrecision が設定されている場合、setdoubleValue() メソッドは、デフォルト・ロケールの精度の代わりにこれを使用して、入力値の精度を指定します。

参照項目

getDefaultdouble(),, getdoubleValue(),, setbooleanValue(),, setBusObjValue(),, setfloatValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setfloatValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、float 属性の値を、指定された値に設定します。

構文

```
public void setfloatValue(String attrName, float newVal);
public void setfloatValue(int position, float newVal);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てられる float の値です。

戻り値

なし。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeValueException`
渡された値が、特定の属性に対して有効な値でない場合、スローされます。

`WrongAttributeException`
`float` 以外の属性に対して値が設定されている場合、スローされます。

注記

コネクター固有のプロパティー `MaxFloatPrecision` が設定されている場合、`setfloatValue()` メソッドは、デフォルト・ロケールの精度の代わりにこれを使用して、入力値の精度を指定します。

参照項目

getDefaultfloat(),, getfloatValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setintValue(),, setLongTextValue(),, setStringValue()

---

setintValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、`int` 属性の値を、指定された値に設定します。

構文

```
public void setintValue(String attrName, int newVal);
public void setintValue(int position, int newVal);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てられる `int` の値です。

戻り値

なし。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeValueException`
渡された値が、特定の属性に対して有効な値でない場合、スローされます。

`WrongAttributeException`
整数以外の属性に対して値が設定されている場合、スローされます。

参照項目

getDefaultint(),, getintValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setLongTextValue(),, setStringValue()

---

setLocale()

ビジネス・オブジェクトのロケールを設定します。

構文

```
public void setLocale(String localeName);
```

パラメーター

localeName 現行ビジネス・オブジェクトに関連付けるロケールの名前。

戻り値

なし。

例外

`IllegalLocaleException`
指定されたロケール名が有効ではない場合にスローされます。

注記

`setLocale()` メソッドは、そのビジネス・オブジェクトに関連したロケールを識別する、ビジネス・オブジェクト・ロケールを設定します。このロケールは、ビジネス・オブジェクト内のデータに関連した言語およびコードのエンコードを示すものであり、ビジネス・オブジェクト定義の名前またはその属性 (これは、英語 (U.S.) ロケール `en_US` に関連したコード・セット内の文字でなければなりません) に関連するものではありません。ビジネス・オブジェクトにロケールが関連付けられていない場合、コネクター・フレームワークは、コネクター・フレームワークのロケールをビジネス・オブジェクトのロケールとして割り当てます。

参照項目

getLocale()

setLongTextValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、LongText 属性の値を、指定された値に設定します。

構文

```
public void setLongTextValue(String attrName, String newVal);
public void setLongTextValue(int position, String newVal);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てる LongText 値を含む String。

戻り値

なし。

例外

AttributeNotFoundException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

AttributeValueException
: 渡された値が、特定の属性に対して有効な値でない場合、スローされます。

WrongAttributeException
: LongText 以外の属性に対して値が設定されている場合、スローされます。

参照項目

getLongTextValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setStringValue()

setStringValue()

属性の名前、またはビジネス・オブジェクトの属性リストの中での属性の位置を与えることで、String 属性の値を、指定された値に設定します。

構文

```
public void setStringValue(String attrName, String newVal);
public void setStringValue(int position, String newVal);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newVal 属性に割り当てられる `String` の値です。

戻り値

なし。

例外

`AttributeNotFoundException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

`AttributeValueException`
渡された値が、特定の属性に対して有効な値でない場合、スローされます。

`WrongAttributeException`
`String` 以外の属性に対して値が設定されている場合、スローされます。

参照項目

getDefaultString(),, getStringValue(),, setbooleanValue(),, setBusObjValue(),, setdoubleValue(),, setfloatValue(),, setintValue(),, setLongTextValue()

---

setVerb()

ビジネス・オブジェクトのアクティブな動詞を設定します。

構文

```
public void setVerb(String newVerb);
```

パラメーター

newVerb ビジネス・オブジェクトの参照先となるビジネス・オブジェクト定義の動詞リスト内にある動詞です。

戻り値

なし。

例外

`InvalidVerbException`
: 引き渡された動詞が、ビジネス・オブジェクト定義でサポートされている動詞ではない場合にスローされます。

注記

ビジネス・オブジェクト定義には、ビジネス・オブジェクトによって使用されている動詞のリストが含まれています。アクティブな動詞として設定される動詞は、このリスト上になければなりません。1 つのビジネス・オブジェクトに対して、一度にアクティブである動詞は、1 つに限定されています。

すべてのビジネス・オブジェクトによってサポートされている典型的な動詞としては、Create、Retrieve、および Update が挙げられます。ビジネス・オブジェクトは、追加動詞、例えば、Delete などをサポートする場合があります。ビジネス・オブジェクトをサポートしているコネクターはすべて、サポート対象の動詞をすべて実装している必要があります。

参照項目

getVerb()

第 14 章 CWConnectorConstant クラス

CWConnectorConstant クラスは、すべての Java コネクターによって共有される定数を定義します。 CWConnectorConstant クラスによって、以下のグループの静的定数が提供されています。

- 『結果状況定数』
- 350 ページの『動詞定数』
- 350 ページの『コネクター・プロパティー定数』

注: 下位の Java コネクター・ライブラリーの CxStatusConstants クラスは、CWConnectorConstant クラスによって拡張されます。下位の Java コネクター・ライブラリーのクラスの詳細については、 465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

結果状況定数

Java コネクター・ライブラリーの多数のメソッドは、整数の結果状況を戻して、メソッドの成功を示します。CWConnectorConstant クラスに定義されている静的結果状況定数については、表 125 に要約をまとめます。

表 125. CWConnectorConstant クラスの結果状況定数

定数名	意味
SUCCEED	操作は正常に終了しました。
APPRESPONSETIMEOUT	アプリケーションが応答していません。
BO_DOES_NOT_EXIST	取得で要求されたビジネス・オブジェクトは存在していません。
CONNECTOR_NOT_ACTIVE	コネクターは、イベントを渡そうとしましたが、コネクター・コントローラーがアクティブになっていないため、一時停止されました。コネクター・コントローラーが存在するのは、統合ブローカーが InterChange Server の場合のみです。
FAIL	特定できない原因のため、操作に失敗しました。
MULTIPLE_HITS	内容による取得が統合ブローカーによって要求されましたが、コネクターは条件に一致するレコードを複数検出しました。この状況は、取得要件に一致するレコードが複数存在していることを示します。
NO_SUBSCRIPTION_FOUND	イベントのサブスクリプションが存在しません。
RETRIEVEBYCONTENT_FAILED	内容による取得が失敗しました。
UNABLETOLOGIN	コネクターがアプリケーションにログインできません。
VALCHANGE	操作が正常に完了して、ターゲット・アプリケーションのオブジェクトの値が変更されました。
VALDUPES	アプリケーション内のオブジェクトが、要求された特性をすでに備えているため、要求された操作は必要ありません。

動詞定数

Java コネクターの doVerbFor() メソッドは、基本的な動詞の値の 1 つを参照する必要が生じた場合、CWConnectorConstant クラスによって定義されている動詞定数を使用できます。静的動詞定数については、表 125 に要約をまとめます。

表 126. CWConnectorConstant クラスの動詞定数

定数名	意味
VERB_CREATE	Create 動詞のストリング表記
VERB_RETRIEVE	Retrieve 動詞のストリング表記
VERB_UPDATE	Update 動詞のストリング表記
VERB_DELETE	Delete 動詞のストリング表記
VERB_EXISTS	Exists 動詞のストリング表記
VERB_RETRIEVEBYCONTENT	RetrieveByContent 動詞のストリング表記

動詞定数は、doVerbFor() メソッドで用いることができます。

コネクター・プロパティー定数

Java コネクター・ライブラリーの多数のメソッドは、整数の結果状況を戻して、メソッドの成功を示します。CWConnectorConstant クラスに定義されている静的結果状況定数については、表 125 に要約をまとめます。

表 127. CWConnectorConstant クラスのコネクター・プロパティー定数

定数名	意味
HIERARCHICAL	このコネクター・プロパティーは階層型プロパティーです。つまり、複数のストリング値と子プロパティーの組み合わせが含まれています。
SIMPLE	このコネクター・プロパティーは単純型プロパティーです。つまり、ストリング値のみが含まれ、子プロパティーは含まれていません。
SINGLE_VALUED	このコネクター・プロパティーには単一値のみが含まれます。
MULTI_VALUED	このコネクター・プロパティーには、1 つまたは複数の値が含まれます。

第 15 章 CWConnectorEvent クラス

CWConnectorEvent クラスを使用すると、コネクター・イベント・オブジェクトの作成、およびそのイベント・オブジェクトとの対話が可能になります。イベント・オブジェクトは、アプリケーション内で発生したイベントを表します。これらのイベント・オブジェクトは、アプリケーションからイベントがプルされるたびに、イベント・ストアによって作成されます。その後、各イベント・オブジェクト内の情報を使用して、コネクター・インフラストラクチャーでさらに処理するためのビジネス・オブジェクトが作成され、取得されます。

CWConnectorEvent クラスのメソッドについては、表 128 に要約をまとめます。

表 128. CWConnectorEvent クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorEvent()	新規のイベント・オブジェクトを作成します。	351
getBusObjName()	イベント・オブジェクトに関連付けられているビジネス・オブジェクトの名前を取得します。	352
getConnectorID()	イベント・オブジェクトからコネクター ID を取得します。	353
getEffectiveDate()	イベント・オブジェクトから発効日を取得します。	353
getEventID()	イベント・オブジェクトからイベント ID を取得します。	354
getEventSource()	イベント・オブジェクトからイベント・ソースの名前を取得します。	354
getEventTimeStamp()	イベント・オブジェクトからイベント・タイム・スタンプを取得します。	355
getIDValues()	イベント・オブジェクトからビジネス・オブジェクトのデータ値を取得します。	355
getKeyDelimiter()	イベント・オブジェクトからキー区切り文字を取得します。	356
getPriority()	イベント・オブジェクトから優先順位を取得します。	356
getStatus()	イベント・オブジェクトから状況を取得します。	357
getTriggeringUser()	イベント・オブジェクトから、トリガー実行ユーザーを取得します。	358
getVerb()	イベント・オブジェクトから動詞を取得します。	358
setEventSource()	イベント・ソースをイベント・オブジェクト内にある指定された値に設定します。	358

CWConnectorEvent()

新規のイベント・オブジェクトを作成します。

構文

```
public CWConnectorEvent();

public CWConnectorEvent(String eventID, String busObjName,
    String verb, String IDvalues, int status, int priority,
    String connectorID, Date eventTimeStamp, Date effectiveDate,
    String triggeringUser, String description, String delimiter);
```

パラメーター

busObjName イベントに関連付けられているビジネス・オブジェクトの名前です。

connectorID イベントに関連したコネクターのコネクター ID です。

description イベント記述 (省略可) です。

delimiter イベントのキー値を分離する区切り文字です。

effectiveDate イベントの発効日です。

eventID イベント用のイベント ID を指定します。

eventTimeStamp イベント用のタイム・スタンプです。

IDvalues イベントに関連付けられているビジネス・オブジェクトのデータです。

priority イベント優先順位 (整数) です。

status イベントに関連するイベント状況定数は、以下のいずれかです。

```
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

triggeringUser イベントを起動するユーザーに関連付けられたユーザー ID です。

verb *busObjName* ビジネス・オブジェクト用の動詞です。

戻り値

新規に作成されたビジネス・オブジェクトを包含している CWConnectorEvent オブジェクト。

注記

CWConnectorEvent() コンストラクターの形式には、以下の 2 種類があります。

- 最初の形式のコンストラクターは、空のイベント・オブジェクトを作成します。
- 2 番目の形式のコンストラクターは、新規イベント・オブジェクトを初期化するためのデータを渡します。イベント・オブジェクトのメンバーを初期化する手段は、2 番目の形式の CWConnectorEvent() コンストラクターによって提供されます。

注: イベント記述を初期化する唯一の方法は、2 番目の形式の CWConnectorEvent() コンストラクターを介して行うことです。このメンバーのアクセサー・メソッドは存在していません。これは、コネクターがイベント記述を使用しないためです。

---

getBusObjName()

イベント・オブジェクトに関連付けられているビジネス・オブジェクトの名前を取得します。

構文

```
public String getBusObjName();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトの名前が格納されている String オブジェクト。

例外

AttributeNullValueException
ビジネス・オブジェクトの名前がヌルの場合、スローされます。

注記

イベント・ストアによっては、ビジネス・オブジェクトの名前を保持し続けない可能性があります。場合によっては、ビジネス・オブジェクトが作成されたときに、そのビジネス・オブジェクト名が内容に基づいて決定されることもあります。

---

getConnectorID()

イベント・オブジェクトからコネクター ID を取得します。

構文

```
public String getConnectorID();
```

パラメーター

なし。

戻り値

イベントの割り当て先となるコネクターを識別するコネクター ID が格納されている String。

例外

AttributeNullValueException
コネクター ID がヌルの場合、スローされます。

注記

現在、コネクター ID は、トレースの目的にのみ使用されています。

---

getEffectiveDate()

イベント・オブジェクトから発効日を取得します。

構文

```
public Date getEffectiveDate();
```

戻り値

イベントの発効日 (イベントがアクティブ化され処理されるようになる日付) が格納されている Date オブジェクト。

例外

AttributeNullValueException
: イベントの発効日がヌルの場合、スローされます。

注記

ユーザーのイベント検出機構が、将来のイベント処理に対応している (つまり、将来の特定の時点で処理するイベントを格納する) 場合、発効日は有用です。イベントを処理すべき日付は、発効日で示されます。

---

getEventID()

イベント・オブジェクトからイベント ID を取得します。

構文

```
public String getEventID();
```

パラメーター

なし。

戻り値

イベントを一意的に識別するイベント ID が格納されている String オブジェクト。

例外

AttributeNullValueException
: イベント ID がヌルの場合、スローされます。

注記

イベント・ストアがデータベース内のイベント表である場合は、表行のキー値がイベント ID になります。それ以外のイベント・ストアの場合は、ファイル名とそのファイルの中でのレコードの位置が、イベント ID になります。

---

getEventSource()

イベント・オブジェクトからイベント・ソースの名前を取得します。

構文

```
public String getEventSource();
```

戻り値

イベント発生元であるイベント・ソースが格納されている String オブジェクト。

例外

AttributeNullValueException
イベント・ソースがヌルの場合、スローされます。

注記

一般的にイベント・ソースは、この情報をアーカイブの目的に必要とするコネクターによって使用されます。例えば、WebSphere Business Integration Adapter for JText は、WebSphere MQ キューの名前を保管します。

---

getEventTimeStamp()

イベント・オブジェクトからイベント・タイム・スタンプを取得します。

構文

```
public Date getEventTimeStamp();
```

パラメーター

なし。

戻り値

イベント・タイム・スタンプ (イベントが作成された時刻) が格納されている String オブジェクト。

例外

AttributeNullValueException
イベント・タイム・スタンプがヌルの場合、スローされます。

---

getIDValues()

イベント・オブジェクトからビジネス・オブジェクトのデータ値を取得します。

構文

```
public String getIDValues();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトを識別するための、ビジネス・オブジェクトのデータ値が格納されている String オブジェクト。

例外

AttributeNullValueException
ビジネス・オブジェクトのデータがヌルの場合、スローされます。

注記

基準としては、これらのデータ値は、ビジネス・オブジェクトのキー値になっていなければなりません。つまり、データ値は、名前と値のペア形式になります。これらのデータ値には、ポーリング中に取得されるビジネス・オブジェクトを一意的に識別するのに必要な属性値がすべて、含まれていなければなりません。

---

getKeyDelimiter()

イベント・オブジェクトからキー区切り文字を取得します。

構文

```
public String getKeyDelimiter();
```

パラメーター

なし。

戻り値

イベントのキー区切り文字が格納されている String オブジェクト。

例外

AttributeNullValueException
キー区切り文字がヌルの場合、スローされます。

---

getPriority()

イベント・オブジェクトから優先順位を取得します。

構文

```
public int getPriority();
```

パラメーター

なし。

戻り値

イベントの優先順位を示す整数。

例外

なし。

注記

イベントの優先順位を使用することにより、イベントの適切な処理順序が判別されます。

---

getStatus()

イベント・オブジェクトから状況を取得します。

構文

```
public int getStatus();
```

パラメーター

なし。

戻り値

イベント状況を表す整数値です。この整数値を以下に示すイベント状況定数と比較することにより、状況が判別されます。

```
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

例外

なし。

注記

Java コネクター・ライブラリーには、`getStatus()` メソッドが `CWConnectorEvent` クラスの public メソッドとして用意されています。ただし、この状況を設定するための public メソッドはありません。イベント状況を設定するには、`CWConnectorEventStore` クラスの以下の Java コネクター・ライブラリー・メソッドを 1 つを使用します。

- getNextEvent()
- recoverInProgressEvents()
- resubmitArchivedEvents()
- setEventStatus()
- updateEventStatus()

getTriggeringUser()

イベント・オブジェクトから、トリガー実行ユーザーを取得します。

構文

```
public String getTriggeringUser();
```

パラメーター

なし。

戻り値

イベントを起動するユーザー (イベントを起動するユーザーの ID) が格納されている String オブジェクト。

例外

AttributeNullValueException
トリガー実行ユーザーの名前がヌルの場合にスローされます。

注記

トリガー実行ユーザーの値を使用することにより、2 つのシステム間での同期化中に、標準的な方法でピンポンを回避できます。

getVerb()

イベント・オブジェクトから動詞を取得します。

構文

```
public String getVerb();
```

パラメーター

なし。

戻り値

イベントに関連付けられた動詞が格納されている String オブジェクト。

例外

AttributeNullValueException
動詞がヌルの場合、スローされます。

setEventSource()

イベント・ソースをイベント・オブジェクト内にある指定された値に設定します。

構文

```
public void setEventSource(String eventSource);
```

パラメーター

eventSource　　イベントに割り当てる新規イベント・ソースを指定します。

戻り値

なし。

例外

なし。

第 16 章 CWConnectorEventStatusConstants クラス

CWConnectorEventStatusConstants クラスでは、イベントが取り得る状況値に対応する静的定数を定義します。

イベント状況定数

イベント状況定数は、イベントの現在の状況を追跡するポーリング・メソッドで使用されるのが一般的です。 CWConnectorEventStatusConstants クラスの静的イベント状況定数については、表 129 に要約をまとめます。

表 129. CWConnectorEventStatusConstants クラスの静的定数

イベント状況定数	意味
ERROR_OBJECT_NOT_FOUND	アプリケーション・データベース内でのイベント検出エラー。
ERROR_POSTING_EVENT	InterChange Server にイベントを送信するときのエラー。エラーの説明は、イベント・レコードの中のイベント記述の後に追加できます。
ERROR_PROCESSING_EVENT	イベント処理エラー。エラーの説明は、イベント・レコードの中のイベント記述の後に追加できます。
IN_PROGRESS	イベントが進行中。
READY_FOR_POLL	ポーリング可能。
SUCCESS	コネクター・フレームワークに送信されます。
UNSUBSCRIBED	イベントのサブスクリプションが存在しません。

図 76 に、各種のイベント状況定数がどのような場合に設定されるかを示します。

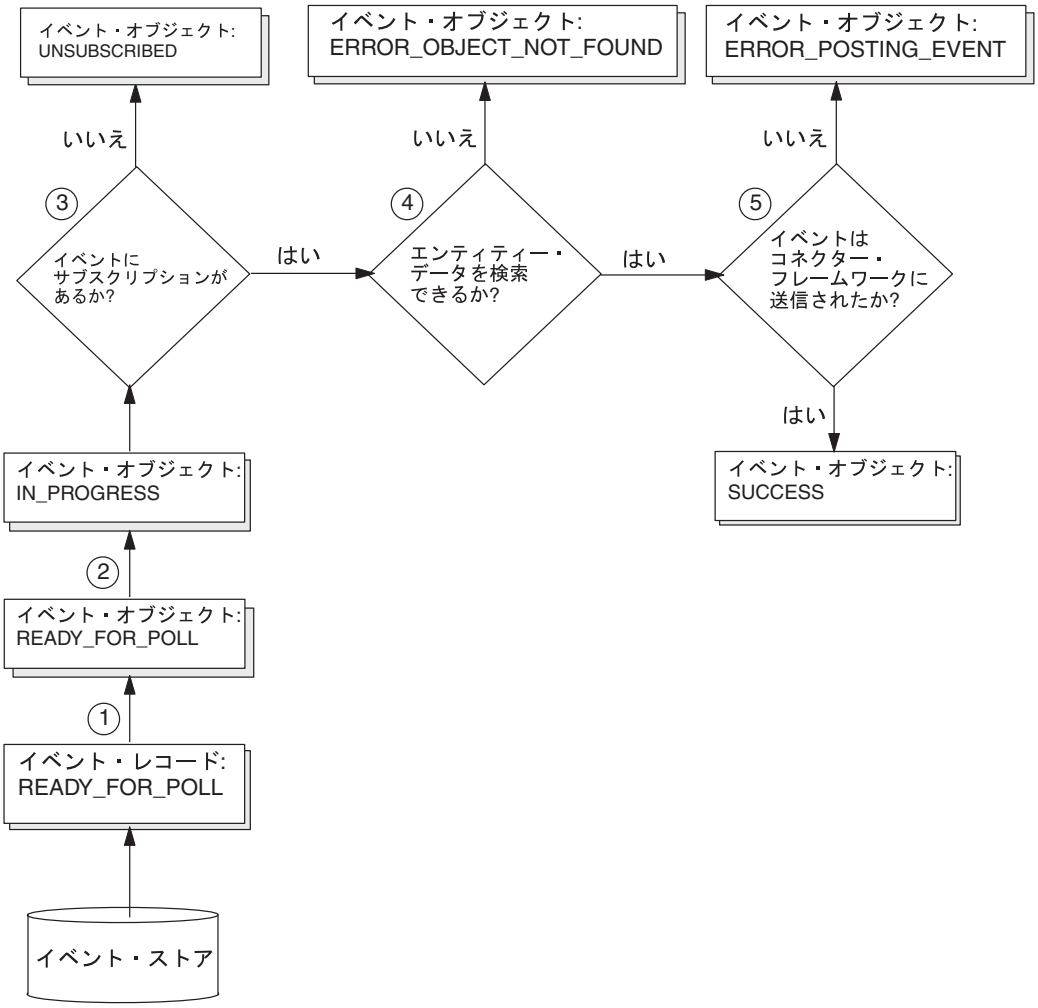

図 76. ポーリング・メソッドのイベント状況値

図 76 に示すように、ポーリング・メソッドは以下のステップを実行して、イベント・オブジェクトの状況を保守します。

1. `fetchEvents()` で、Ready-for-Poll イベント・レコードを取得して、`READY_FOR_POLL` 状況を持つイベント・オブジェクトを作成します。
2. `getNextEvent()` メソッドで、イベント・ベクトルから Ready-for-Poll イベント・オブジェクトを取得して、その状況を `IN_PROGRESS` に更新します。
3. ポーリング・メソッドでは、`isSubscribed()` メソッドを使用して、取得されたイベントにサブスクリプションがあるかどうかを検査します。
   - サブスクリプションが存在していない場合、ポーリング・メソッドは `updateEventStatus()` を使用して、イベント・オブジェクトの状況を `UNSUBSCRIBED` に変更します。
   - サブスクリプションが存在していない場合、ポーリング・イベントの実行はステップ 4 に進みます。
4. ビジネス・オブジェクトに取り込まれるアプリケーション・エンティティーのデータを取得するための `getBO()` メソッドが、ポーリング・メソッドによって呼び出されます。

- getBO() がアプリケーション・エンティティーのデータを検出できない場合、ポーリング・メソッドは updateEventStatus() を使用して、イベント・オブジェクトの状況を ERROR_OBJECT_NOT_FOUND に変更します。
- アプリケーション・エンティティーのデータが検出された場合、ポーリング・イベントの実行はステップ 5 に進みます。

5. ビジネス・オブジェクトをコネクター・フレームワークに送信するための gotApplEvent() メソッドが、ポーリング・メソッドによって呼び出されます。その後、このコネクター・フレームワークでは、そのビジネス・オブジェクトがその宛先に転送されます。サブスクリプションが存在していない場合、ポーリング・メソッドは updateEventStatus() を使用して、イベント・オブジェクトの状況を、gotApplEvent() の成否が反映されるように変更します。gotApplEvent() の戻りコードに対応するイベント状況値については、223 ページの表 100 のリストを参照してください。

第 17 章 CWConnectorEventStore クラス

CWConnectorEventStore クラスは、Java コネクターからイベント・ストアへのアクセスを可能にする基底クラスです。イベント・ストアは、イベントを永続的に格納するためのアプリケーション機構です。アプリケーションは、アプリケーション内で発生するイベントに備えて、イベント・ストア内にイベント・レコードを格納します。コネクターは、イベント・ストアからイベントを取得して、統合ブローカーに転送するための処理を施します。コネクター開発者は、このクラスからイベント・ストア・クラスを派生し、そのイベント・ストアでいくつかのメソッドを実装する必要があります。

重要: すべての Java コネクターは、このクラスを拡張して、アプリケーションのイベント・ストアにアクセスする必要があります。開発者は、Java CWConnectorEventStore クラスを介してアプリケーションのイベント・ストアにアクセスできるよう、導出済みのイベント・ストア・クラス内に抽象メソッド (deleteEvent()、fetchEvents()、recoverInProgressEvents()、resubmitArchivedEvents()、および setEventStatus()) を実装しておく必要があります。また、開発者はアーカイブ・ストアにアクセスできるよう、archiveEvent() メソッドを実装しておく必要があります。

CWConnectorEventStore クラス内のメソッドについては、表 130 に要約します。

表 130. CWConnectorEventStore クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorEventStore()	イベント・ストア・オブジェクトを作成します。	366
archiveEvent()	指定されたイベントを、適切な状況を持つアプリケーション・アーカイブ・ストア内にアーカイブします。	366
cleanupResources()	イベント・ストアへのアクセス時にポーリング・メソッドが使用したリソースを解放します。	367
deleteEvent()	アプリケーションのイベント・ストアからイベントを削除します。	368
fetchEvents()	指定された数の Ready-for-Poll イベントを、アプリケーションのイベント・ストアから取得します。	368
getBO()	イベント・ストアからのイベント内にある情報をベースに、ビジネス・オブジェクトを作成します。	369
getNextEvent()	eventsToProcess ベクトルから、次のイベント・オブジェクトを取得します。	372
recoverInProgressEvents()	進行中イベントをイベント・ストアにリカバリーします。	373
resubmitArchivedEvents()	指定されたアーカイブ済みイベントを、アプリケーションのアーカイブ・ストアからアプリケーションのイベント・ストアにコピーして、イベント状況を READY_FOR_POLL に変更します。	375
setEventStatus()	イベント・ストア内のイベントの状況を設定します。	376
setEventsToProcess()		377
setTerminate()	内部のコネクター終了フラグに true を設定します。	377
updateEventStatus()	アプリケーションのイベント・ストアおよびイベントの、両方のイベント状況を更新します。	378

CWConnectorEventStore()

イベント・ストア・オブジェクトを作成します。

構文

```
public CWConnectorEventStore();
```

パラメーター

なし。

戻り値

新規に作成されたイベント・ストアを包含している CWConnectorEventStore オブジェクト。

注記

CWConnectorEventStore() コンストラクターは、新規イベント・ストアを作成して、単一のデータ・メンバー eventsToProcess を初期化します。 eventsToProcess メンバーは、取得済みのイベント・オブジェクトを保持する Java Vector オブジェクトです。

archiveEvent()

指定されたイベントを、適切な状況を持つアプリケーション・アーカイブ・ストア内にアーカイブします。

構文

```
public int archiveEvent(String eventID);
```

パラメーター

eventID　　アーカイブするイベント用のイベント ID を指定します。

戻り値

アーカイブ操作による結果状況を示す整数。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

CWConnectorConstant.SUCCEED
: イベントのアーカイブが成功しました。

CWConnectorConstant.FAIL
: イベントのアーカイブが失敗しました。

例外

ArchiveFailedException
: 基礎になるアプリケーションがイベントをアーカイブできなかった場合にスローされます。

InvalidStatusChangeException
: コネクターがイベント状況をアプリケーションに無効なイベント状況で更新しようとした場合、スローされます。

注記

arhiveEvent() メソッドは通常、処理済みのイベントまたは失敗したイベントをイベント・アーカイブ・ストアにアーカイブするために、ポーリング・メソッド pollForEvents() から呼び出されます。

重要: archiveEvent() メソッドは、同期化されたメソッドであるため、抽象メソッドではありません。しかし、アーカイブ・ストアにイベントをアーカイブできるようにするには、イベント・ストア・クラスはこのメソッドを実装する必要があります。

参照項目

deleteEvent(),, pollForEvents()

---

cleanupResources()

イベント・ストアへのアクセス時にポーリング・メソッドが使用したリソースを解放します。

構文

```
public void cleanupResources();
```

パラメーター

なし。

戻り値

なし。

例外

なし。

注記

cleanupResources() メソッドは、pollForEvents() メソッドの最終ステップの 1 つとして有用なメソッドです。このメソッドには、pollForEvents() メソッドがイベント・ストアにアクセスするために割り振ったリソースを解放するためのコードを組み込むことができます。例えば、イベント・ストアがイベント表として実装されている場合、pollForEvents() メソッドが、イベント表にアクセスするために SQL カーソルを割り振っている可能性があります。この場合、これらのカーソルをクローズするステートメントを cleanupResources() に組み込むことにより、使用されていたメモリーおよび不要となったカーソルを解放することができます。

重要: cleanupResources() メソッドは、抽象メソッドではありません。ただし、このメソッドはデフォルトの実装を提供することもありません。したがって、

イベント・ストアへのアクセスに使用されたリソースをクリーンアップできるようにするためには、デフォルトの cleanupResources() を独自の実装にオーバーライドする必要があります。

参照項目

pollForEvents()

---

deleteEvent()

アプリケーションのイベント・ストアからイベントを削除します。

構文

```
public abstract void deleteEvent(String eventID);
```

パラメーター

eventID 削除するイベント用のイベント ID を指定します。

戻り値

なし。

例外

DeleteFailedException
基礎になるアプリケーションがイベント・ストアからイベントを削除しようとして失敗した場合に、スローされます。

注記

deleteEvent() メソッドは、主にアーカイブ中に使用されます。このイベントはアプリケーションのアーカイブ・ストアに正常に移動した後、このメソッドによってイベント・ストアから削除されます。

重要: deleteEvent() メソッドは抽象メソッドであるため、イベント・ストアからのイベントの削除を可能にするためには、イベント・ストア・クラスがこのメソッドを実装する必要があります。

参照項目

archiveEvent()

---

fetchEvents()

指定された数の Ready-for-Poll イベントを、アプリケーションのイベント・ストアから取得します。

構文

```
public abstract Vector fetchEvents(int pollQuantity)
```

パラメーター

pollQuantity　アプリケーションのストアからフェッチするイベントの数。

戻り値

なし。

例外

ConnectionFailureException
: 接続が確立できなかった場合にスローされます。

EventProcessingException
: 接続が確立された後、イベントのフェッチ時にエラーが発生した場合にスローされます。

注記

fetchEvents() メソッドは、イベント・ストアから READY_FOR_POLL 状況のイベント・レコードを取得し、それをイベント内に置きます。 fetchEvents() によって取得されるイベントの数は、pollQuantity で指定します。これは、 PollQuantity コネクター構成プロパティーに関連付けられます。メソッドでは、取得されたイベントごとに、CWConnectorEvent イベント・オブジェクトを作成し、このイベント・オブジェクトを Java Vector に格納し、Vector を戻す必要があります。
eventsToProcess ベクトル内にイベント・オブジェクトが保管される順序は、fetchEvents() メソッドによって判別されます。

重要: fetchEvents() メソッドは抽象メソッドであるため、イベント・ストアからの READY_FOR_POLL イベントの取り出しを可能にするためには、イベント・ストア・クラスがこのメソッドを実装する必要があります。

注: fetchEvents() メソッドは通常、ポーリング・メソッド pollForEvents() から呼び出されます。

注: 入力パラメーターも戻り値も指定しない fetchEvents() の前のシグニチャーは、推奨されませんでした。このバージョンでは置き換えられています。

参照項目

getNextEvent(),, pollForEvents()

---

getBO()

イベント・ストアからのイベント内にある情報をベースに、ビジネス・オブジェクトを作成します。

構文

```
public CWConnectorBusObj getBO(CWConnectorEvent eventObject);
public CWConnectorBusObj getBO(CWConnectorEvent eventObject,

    int status,
```

```
String RetrieveVerb);
```

パラメーター

eventObject ビジネス・オブジェクト情報を包含するイベントです。

status getBO() メソッド内でメソッドまたは例外によって設定される状況値です。

RetrieveVerb イベントが検出されたアプリケーション・レコードをフェッチするデフォルトの RetrieveByContent 動詞をオーバーライドするのに使用されます。getBO() によって、RetrieveByContent の代わりに特定の動詞を使用して、アプリケーションから変更済みのレコードをフェッチできます。

戻り値

アプリケーションのデータベースから取得した情報に基づいて新規ビジネス・オブジェクトを包含している CWConnectorBusObj オブジェクト。メソッドは *eventObject* イベント・オブジェクトを取得できなかった場合、null を戻します。

例外

AttributeNotFoundException
: getBO() がキー属性にキー値を割り当てるときに属性が見つからない場合、スローされます。

SpecNameNotFoundException
: イベント・オブジェクト内のビジネス・オブジェクトの名前が無効な場合、スローされます。

AttributeValueException
: 取得した属性値が特定の属性には無効な場合、スローされます。

InvalidVerbException
: イベント・オブジェクト内の動詞が無効な場合、スローされます。

WrongAttributeException
: getBO() がキー属性にキー値を割り当てるときに無効な属性を検出した場合、スローされます。例えば、属性がコンテナーである場合、この属性にはキー値を保持できません。

AttributeNullValueException
: ビジネス・オブジェクトを作成できなかった場合、スローされます。

注記

getBO() メソッドは、*eventObject* イベント・オブジェクトによって記述される、アプリケーション・エンティティーの情報を包含するビジネス・オブジェクトを戻します。

重要: 内部状況コードを呼び出し側メソッドに戻す場合、getBO() メソッドをオーバーライドする必要があります。

このメソッドのデフォルト実装によって、以下のアクションが実行されます。

- 新規ビジネス・オブジェクトを保持するための一時的 CWConnectorBusObj オブジェクトを作成します。
- *eventObject* イベント・オブジェクトからのデータおよびキー値を CWConnectorBusObj オブジェクトに取り込む。
- RetrieveVerb が設定されている場合、ビジネス・オブジェクトを取り出す動詞としてこのプロパティーの値を使用します。
- RetrieveVerb が設定されていない場合、イベント・オブジェクトの動詞の値に基づいて、以下のアクションのいずれかを実行します。

動詞	getBO() アクション
Delete	doVerbFor() を持つオブジェクトを取得しません。
Create、Update	ビジネス・オブジェクトの動詞を RetrieveByContent に設定し、(CWConnectorBusObj クラスの) doVerbFor() メソッド を呼び出して、残りの情報をアプリケーションから取得します。

  動詞が Create または Update の場合、doVerbFor() で取得されたデータを CWConnectorBusObj オブジェクトに取り込みます。これは、doVerbFor() メソッドで生成される可能性のある次の条件を処理します。
  - doVerbFor() メソッドは、アプリケーション内の指定されたエンティティーが検出されない場合、BO_DOES_NOT_EXIST を戻します。この場合、getBO() によって *eventObject* の状況が ERROR_OBJECT_NOT_FOUND に設定され、null が戻されます。
  - doVerbFor() がアプリケーションに接続できない場合は、APPRESPONSETIMEOUT が戻されます。この場合、getBO() は、(CWConnectorEventStore クラス内の) setTerminate() メソッドを呼び出して、内部のコネクター終了フラグを設定します。詳細については、216 ページの『アプリケーション・データの取得』を参照してください。
  - doVerbFor() がその他のエラー (RETRIEVEBYCONTENT_FAILED など) を戻す場合、getBO() メソッドは null を戻します。
- gotApplEvent() メソッドを呼び出して、CWConnectorBusObj オブジェクトをコネクター・フレームワークに送信します。

注: getBO() メソッドは通常、ポーリング・メソッド pollForEvents() から呼び出されます。

前述のように、getBO() のデフォルト実装には、特定のエラーまたは例外条件の発生を呼び出し側メソッドに示す方法がいくつかあります。しかし、特定の内部状況値 (スローされた例外の状況属性) を呼び出し側メソッドに戻す必要がある場合は、このデフォルト実装をオーバーライドできます。getBO() の実装では、*状況* 引き数を提供するこのメソッドのシグニチャーの 2 番目の形式を使用します。getBO() 内でこの引き数に何らかの状況値を割り当ててから、getBO() を終了します。呼び出し側メソッドから、未初期化状況値を渡し、getBO() への呼び出し後に、初期化された状況値にアクセスします。

注: pollForEvents() メソッドのデフォルト実装は、最初の形式の getBO() を呼び出します。つまり、これは、getBO() によって戻される初期化済み状況値を処理しません。

参照項目

doVerbFor(),, getTerminate(), pollForEvents(), setTerminate()

---

getNextEvent()

eventsToProcess ベクトルから、次のイベント・オブジェクトを取得します。

構文

```
public CWConnectorEvent getNextEvent();
```

パラメーター

なし。

戻り値

次の Ready-for-Poll イベント用の CWConnectorEvent オブジェクト。
eventsToProcess ベクトルが空の場合、メソッドは null を戻します。

例外

InvalidStatusChangeException
: イベント状況がアプリケーションに無効な状況値に変更されている場合、スローされます。

StatusChangeFailedException
: READY_FOR_POLL から IN_PROGRESS への状況変更が失敗した場合、スローされます。

注記

getNextEvent() メソッドは、eventsToProcess ベクトルに、現在 READY_FOR_POLL 状況を持っているイベントがあるかどうかを調べます。メソッドは、このベクトル内にそのようなイベントを検出した場合、以下のアクションを実行します。

1. eventsToProcess ベクトルから、次に処理するイベントを取得します。eventsToProcess ベクトル内にイベント・オブジェクトが保管される順序は、fetchEvents() メソッドによって判別されます。
2. そのイベント状況を IN_PROGRESS に変更します。
3. イベントを呼び出し元に戻します。

eventsToProcess ベクトルは、fetchEvents() メソッドまたは setEventsToProcess() メソッドのいずれを使用して初期化されます。

注: getNextEvent() メソッドは通常、ポーリング・メソッド pollForEvents() から呼び出されます。

参照項目

fetchEvents(),, pollForEvents(),, setEventsToProcess()

---

getTerminate()

内部のコネクター終了フラグの値を取得します。

構文

```
public boolean getTerminate();
```

パラメーター

なし。

戻り値

内部のコネクター終了フラグの現在の設定を示すブール値。

例外

なし。

注記

`getTerminate()` メソッドは、コネクター・フレームワークがコネクターを終了すべきことを示す内部フラグの値を取得します。コネクターは、`setTerminate()` メソッドを使用して、この内部フラグの状況を設定できます。 `pollForEvents()` メソッドは、`getBO()` を呼び出した後に `getTerminate()` メソッドを呼び出し、`APPRESPONSETIMEOUT` 結果状況を戻すかどうかを決める必要があります。詳細については、 216 ページの『アプリケーション・データの取得』を参照してください。

参照項目

getBO(),, setTerminate()

---

recoverInProgressEvents()

進行中イベントをイベント・ストアにリカバリーします。

構文

```
public abstract int recoverInProgress();
```

パラメーター

なし。

戻り値

リカバリー操作による結果状況を示す整数。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

CWConnectorConstant.SUCCEED
進行中のイベントのリカバリーが成功しました。

CWConnectorConstant.FAIL
進行中のイベントのリカバリーが失敗しました。

例外

InvalidStatusChangeException
状況がアプリケーションに無効な状況値に変更されている場合、スローされます。

StatusChangeFailedException
IN_PROGRESS から READY_FOR_POLL への状況変更が失敗した場合、スローされます。

AttributeNullValueException
InDoubtEvents のコネクター構成プロパティーが未定義のため設定されていない場合、スローされます。

注記

recoverInProgressEvents() メソッドは、イベント・ストアに現在 IN_PROGRESS 状況を持っているイベントがあるかどうかを調べます。コネクターが突然シャットダウンした場合、イベントが IN_PROGRESS というイベント状況のまま、イベント・ストア内にとどまることがあります。

注: CWConnectorEventStore クラスは、デフォルトでは recoverInProgressEvents() メソッドを実装しません。したがって、コネクターの始動時に進行中イベントをリカバリーできるようにするには、イベント・ストア・クラスがこのメソッドを実装する必要があります。

recoverInProgressEvents() を実装する方法として、InDoubtEvents コネクター構成プロパティーに基づいてアクションを実行することもできます。このようなイベントが存在する場合、メソッドは、このプロパティーの値に基づいて、次のアクションのいずれかを実行できます。

InDoubtEvents の値	recoverInProgressEvents() のアクション
Reprocess	IN_PROGRESS 状況を持つイベントがすべて、以降のポーリング呼び出し時にコネクター・フレームワークに送信されるよう、それらのイベントの状況を READY_FOR_POLL に変更します。
FailOnStartup	致命的エラーをログに記録して、agentInit() に FAIL 結果状況を戻します。その後、この agentInit() によって InProgressEventRecoveryFailedException 例外がスローされます。 LogAtInterchangeEnd が True に設定されている場合、このアクションでは自動電子メール送信も行われます。
LogError	致命的エラーをログに記録しますが、agentInit() に FAIL 結果状況を戻しません。
Ignore	進行中イベントを無視します。

注: recoverInProgressEvents() が記述内容のとおりに動作するためには、InDoubtEvents コネクター構成プロパティーを定義しておく必要があります。

InDoubtEvents が未定義 の場合は、recoverInProgressEvents() によって AttributeNullValueException 例外がスローされます。

recoverInProgressEvents() メソッドは通常、コネクターの初期化処理の一環として agentInit() メソッドの内部から呼び出されます。 agentInit() が実行する処理としては、recoverInProgressEvents() からの状況の検査の他に、例外のキャッチがあります。agentInit() メソッドが例外をスローするのは、以下のいずれかの場合です。

- recoverInProgressEvents() によって FAIL 結果状況が戻された場合
- recoverInProgressEvents() によって例外がキャッチされた場合

参照項目

agentInit()

---

resubmitArchivedEvents()

指定されたアーカイブ済みイベントを、アプリケーションのアーカイブ・ストアからアプリケーションのイベント・ストアにコピーして、イベント状況を READY_FOR_POLL に変更します。

構文

```
public abstract int resubmitArchivedEvents(String eventID);
```

パラメーター

eventID 再サブミットされるイベント用のイベント ID です。

戻り値

アーカイブ済みイベントの数を示す整数。何も再サブミットされない場合、ゼロ (0) を戻します。

例外

InvalidStatusChangeException
: 状況がアプリケーションに無効な状況値に変更されている場合、スローされます。

StatusChangeFailedException
: READY_FOR_POLL への状況変更が失敗した場合、スローされます。

注記

resubmitArchivedEvents() メソッドは、アーカイブ・ストア内の未処理イベントを、イベント処理の可能なイベント・ストアに再サブミットします。サブスクリプションを持たないイベント、処理済みのイベントは、アーカイブ・ストアに移動されます。処理済みまたはアンサブスクライブされたイベントをアーカイブすることにより、そのイベントの逸失を確実に防止できます。イベント状況を READY_FOR_POLL に設定しておくと、以降にイベント・ストアをポーリングしたときにそのイベントが確実に選出されます。

注: resubmitArchivedEvents() メソッドは抽象メソッドです。したがって、イベント・ストアの後続のポーリングのためにアーカイブ・イベントの再サブミット機能を提供するには、このメソッドをイベント・ストア・クラスに実装する必要があります。

---

setEventStatus()

イベント・ストア内のイベントの状況を設定します。

構文

```
public abstract void setEventStatus(String eventID, int status);
```

パラメーター

eventID 状況変更の対象となるイベント用のイベント ID です。

status 指定されたイベントの新規の状況を識別するイベント状況定数は、以下のいずれかです。

```
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

戻り値

なし。

例外

InvalidStatusChangeException
状況がアプリケーションに無効な状況 値に変更されている場合、スローされます。

注記

setEventStatus() メソッドによって、以下のアクションが実行されます。

- status の値が有効かどうかを検査して、有効でない場合は InvalidStatusChangeException 例外をスローします。
- アプリケーションのイベント・ストア内の eventID で識別されたイベントの状況を変更します。

重要: setEventStatus() メソッドは抽象メソッドであるため、イベント・ストア内のイベントの状況の設定を可能にするためには、イベント・ストア・クラスがこのメソッドを実装する必要があります。

コネクターは、基になるアプリケーション内でイベント状況の変更が確実にコミットされるようにする必要があります。

参照項目

updateEventStatus()

setEventsToProcess()

指定されたイベントを使用して eventsToProcess ベクトルを設定します。

構文

```
public void setEventsToProcess(Vector eventsVector);
```

パラメーター

eventsVector 処理対象イベントを包含する Java.util.Vector オブジェクトです。

戻り値

なし。

例外

なし。

注記

setEventsToProcess() メソッドは、CWConnectorEventStore オブジェクトの eventsToProcess ベクトルに、*eventsVector* ベクトルの内容を割り当てます。

setTerminate()

内部のコネクター終了フラグに true を設定します。

構文

```
public void setTerminate();
```

パラメーター

なし。

戻り値

なし。

例外

なし。

注記

setTerminate() メソッドは、コネクター・フレームワークにコネクターの終了を通知する内部フラグを設定します。コネクターは、getTerminate() メソッドを使用し

て、この内部フラグの状況を確認できます。 doVerbFor() が APPRESPONSETIMEOUT 結果状況を戻した場合、getBO() メソッドは、doVerbFor() を呼び出した後に setTerminate() メソッドを呼び出す必要があります。詳細については、 216 ページの『アプリケーション・データの取得』 を参照してください。

参照項目

getTerminate()

---

updateEventStatus()

アプリケーションのイベント・ストアおよびイベントの、両方のイベント状況を更新します。

構文

```
public void updateEventStore(CWConnectorEvent eventObject,
      int status);
```

パラメーター

eventObject 更新対象の状況を持ったイベント・オブジェクトです。

status イベント・オブジェクト内に格納されるイベント状況定数は、以下のいずれかです。

```
CWConnectorEventStatus.READY_FOR_POLL
CWConnectorEventStatus.IN_PROGRESS
CWConnectorEventStatus.SUCCESS
CWConnectorEventStatus.UNSUBSCRIBED
CWConnectorEventStatus.ERROR_POSTING_EVENT
CWConnectorEventStatus.ERROR_OBJECT_NOT_FOUND
CWConnectorEventStatus.ERROR_PROCESSING_EVENT
```

戻り値

なし。

例外

`InvalidStatusChangeException`
状況がアプリケーションに無効な状況値に変更されている場合、スローされます。

`StatusChangeFailedException`
基礎になるアプリケーションがイベント・ストア内のイベント状況を変更できなかった場合にスローされます。

注記

updateEventStatus() メソッドは、*eventObject* イベントの状況を *status* に設定します。さらに、*eventObject* イベント内のイベント状況を *status* に更新します。

使用すべきでないメソッド

CWConnectorEventStore クラスの一部のメソッドは、初期のバージョンにおいてサポートされたものであり、現在はサポートされていません。これらの*使用すべきでないメソッド* は、エラーを発生させることはありませんが、IBM では、それらの使用を避けて、既存のコードを新規メソッドに移行することを推奨しています。使用すべきでないメソッドは、将来のリリースでは削除される可能性があります。

表 131 に、CWConnectorEventStore クラスの使用すべきでないメソッドのリストを示します。(既存コネクターの変更ではなく) 新規コネクターをコーディングする場合は、このセクションを無視してください。

表 131. CWConnectorEventStore クラスの使用すべきでないメソッド

使用すべきでないメソッド	置換
setEventStoreStatus()	setEventStatus()

第 18 章 CWConnectorEventStoreFactory インターフェース

CWConnectorEventStoreFactory インターフェースは、イベント・ストアを作成するイベント・ストア・ファクトリーの機能を定義します。 Java コネクターで CWConnectorEventStore クラスの拡張を使用してイベント・ストアにアクセスする場合は、イベント・ストア・ファクトリー・クラスを作成し、CWConnectorEventStoreFactory インターフェースを実装する必要があります。このインターフェースには、イベント・ストア (CWConnectorEventStore) オブジェクトをインスタンス化するメソッドが含まれています。

重要: イベント・ストアにアクセスする際に CWConnectorEventStore クラスの拡張を使用する Java コネクターはすべて、必ずこのインターフェースを実装する必要があります。このイベント・ストア・ファクトリー・クラスでは、getEventStore() メソッドを実装し、CWConnectorEventStore クラスを通じてイベント・ストアにアクセスできるようにする必要があります。

表 132 に、CWConnectorEventStoreFactory インターフェースのメソッドについて要約します。

表 132. CWConnectorEventStoreFactory インターフェースのメンバー・メソッド

メンバー・メソッド	説明	ページ
getEventStore()	新規イベント・ストア・オブジェクトを作成します。	381

getEventStore()

新規イベント・ストア・オブジェクトを作成します。

構文

```
public Object getEventStore();
```

パラメーター

なし。

戻り値

新しく作成されたイベント・ストア・オブジェクトを格納している Object。イベント・ストアが見つからない場合、メソッドは null を戻します。

例外

なし。

注記

getEventStore() メソッドは、イベント・ストア・ファクトリーです。このメソッドは、コネクターに対応するイベント・ストアを作成し、イベント・ストア・オブジェクトを戻す必要があります。複数のイベント・ストアを使用するコネクターの場合、イベント・ストア・クラスごとに、このメソッドの実装を提供する必要があります。

CWConnectorAgent クラスにデフォルトで実装されている getEventStore() メソッドは、EventStoreFactory コネクター構成プロパティーで指定されたイベント・ストア・ファクトリー・クラスの getEventStore() メソッドを呼び出します。詳細については、205 ページの『CWConnectorEventStoreFactory インターフェース』を参照してください。

参照項目

getEventStore()

第 19 章 CWConnectorExceptionObject クラス

CWConnectorExceptionObject クラスは、例外についての詳細情報を提供する*例外詳細オブジェクト* を表します。 Java コネクター・ライブラリーのメソッドがスローできる例外にはそれぞれ、例外詳細オブジェクトを包含できます。このクラスは、例外メッセージに関する情報の保管、およびアクセスのためのメソッドを提供します。表 133 に、CWConnectorExceptionObject クラスのメソッドについての要約をまとめます。

表 133. CWConnectorExceptionObject クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorExceptionObject()	例外詳細オブジェクトを作成します。	383
getExpl()	例外詳細オブジェクトのメッセージ番号に関連付けられたメッセージの説明を取得します。	384
getMsg()	例外詳細オブジェクトからメッセージ・テキストを取得します。	384
getMsgNumber()	例外詳細オブジェクト内のメッセージに関連付けられているメッセージ番号 (ID) を取得します。	385
getMsgType()	例外詳細オブジェクト内のメッセージに関連付けられているメッセージ・タイプを取得します。	385
setExpl()	例外詳細オブジェクト内にメッセージの説明を設定します。	386
setMsg()	例外詳細オブジェクト用のメッセージ・テキストを設定します。	387
setMsgNumber()	例外詳細オブジェクト内のメッセージに関連付けられているメッセージ番号 (ID) を設定します。	387
setMsgType()	例外詳細オブジェクト内のメッセージに関連付けられているメッセージ・タイプを設定します。	388
setStatus()	例外詳細オブジェクトの状況値を設定します。	388

CWConnectorExceptionObject()

例外詳細オブジェクトを作成します。

構文

```
public CWConnectorExceptionObject();
```

パラメーター

なし。

戻り値

新規に作成された例外詳細オブジェクトを包含している CWConnectorExceptionObject オブジェクト。

getExpl()

例外詳細オブジェクトのメッセージ番号に関連付けられたメッセージの説明を取得します。

構文

```
public String getExpl();
```

パラメーター

なし。

戻り値

現在の例外詳細オブジェクトのメッセージの説明が格納されている `String` オブジェクト。

例外

なし。

参照項目

setExpl()

getMsg()

例外詳細オブジェクトからメッセージ・テキストを取得します。

構文

```
public String getMsg();
```

パラメーター

なし。

戻り値

現在の例外詳細オブジェクトのメッセージ・テキストを包含する `String` オブジェクト。

例外

なし。

参照項目

setMsg()

getMsgNumber()

例外詳細オブジェクト内のメッセージに関連付けられているメッセージ番号 (ID) を取得します。

構文

```
public int getMsgNumber();
```

パラメーター

なし。

戻り値

例外詳細オブジェクトのメッセージのメッセージ番号 (整数)。

参照項目

setMsgNumber()

getMsgType()

例外詳細オブジェクト内のメッセージに関連付けられているメッセージ・タイプを取得します。

構文

```
public int getMsgType();
```

パラメーター

なし。

戻り値

例外詳細オブジェクトのメッセージのメッセージ・タイプを示す整数。この整数値を以下に示すメッセージ・タイプ定数と比較することにより、メッセージ・タイプが判別されます。

```
XRD_ERROR
```

```
XRD_FATAL
```

これらのメッセージ・タイプ定数は、`CWConnectorUtil` クラスおよび `CWConnectorLogAndTrace` クラスの両方に定義されます。

参照項目

setMsgType()

getStatus()

例外詳細オブジェクトから状況を取得します。

構文

```
public int getStatus();
```

パラメーター

なし。

戻り値

例外詳細オブジェクトの状況を表す整数値。この整数値を以下に示す結果状況定数と比較することにより、メッセージ・タイプが判別されます。

```
    CWConnectorConstant.APPRESPONSETIMEOUT
CWConnectorConstant.BO_DOES_NOT_EXIST
CWConnectorConstant.    MULTIPLE_HITS
CWConnectorConstant.RETRIEVEBYCONTENT_FAILED
    CWConnectorConstant.UNABLETOLOGIN
```

これらの結果状況定数は、`CWConnectorConstant` クラスに定義されています。

例外

なし。

参照項目

setStatus()

setExpl()

例外詳細オブジェクト内にメッセージの説明を設定します。

構文

```
public void setExpl(String msgExpl);
```

パラメーター

msgExpl　例外詳細オブジェクトに割り当てるメッセージの説明を包含する `String` オブジェクトです。

戻り値

なし。

例外

なし。

参照項目

getExpl()

setMsg()

例外詳細オブジェクト用のメッセージ・テキストを設定します。

構文

```
public void setMsg(String newMsg);
```

パラメーター

newMsg 例外詳細オブジェクトに割り当てるメッセージ・テキストを包含する `String` オブジェクト。

戻り値

なし。

例外

なし。

参照項目

getMsg()

setMsgNumber()

例外詳細オブジェクト内のメッセージに関連付けられているメッセージ番号 (ID) を設定します。

構文

```
public void setMessageNumber(int msgNumber);
```

パラメーター

msgNumber 例外詳細オブジェクトのメッセージ用に設定されるメッセージ番号 (整数)。

戻り値

なし。

例外

なし。

参照項目

getMsgNumber()

setMsgType()

例外詳細オブジェクト内のメッセージに関連付けられているメッセージ・タイプを設定します。

構文

```
public void setMsgType(int msgType);
```

パラメーター

msgType 例外詳細オブジェクト内にあるメッセージの重大度を示すメッセージ・タイプ。次のメッセージ・タイプ定数のいずれかを使用します。

```
XRD_ERROR
XRD_FATAL
```

注: 他のメッセージ・タイプ定数が存在している場合でも、それらのメッセージ・タイプ定数は、例外詳細オブジェクト内のメッセージ用タイプとしては有効ではありません。このオブジェクトは例外オブジェクトの一部であり、例外発生時にのみスローされます。

戻り値

なし。

例外

なし。

参照項目

getMsgType()

setStatus()

例外詳細オブジェクトの状況値を設定します。

構文

```
public void setStatus(int status);
```

パラメーター

status 例外詳細オブジェクトに割り当てる結果状況を示す整数値です。

戻り値

なし。

注記

例外がスローされる*前*に、setStatus() メソッドを使用して例外詳細オブジェクトの例外状況を設定しておく必要があります。呼び出し側コードは、この状況値によって (例えば、APPRESPONSETIMEOUT 状況に基づいて)、適切な処置を取りアプリケーション関連リソースをすべてクリーンアップした後で、この状況値をコネクター・フレームワークに渡すことができます。

例外

なし。

参照項目

getStatus()

第 20 章 CWConnectorLogAndTrace クラス

CWConnectorLogAndTrace クラスは、すべてのコネクターによって共有されるログ・トレース定数を定義します。このクラスには、以下の静的定数が含まれています。

- 『メッセージ・タイプ定数』
- 『トレース・レベル定数』

メッセージ・タイプ定数

CWConnectorLogAndTrace クラスに定義されている静的メッセージ・タイプ定数については、表 134 に要約をまとめます。

表 134. CWConnectorLogAndTrace クラスのメッセージ・タイプ定数

定数名	意味
XRD_WARNING	警告メッセージ
XRD_TRACE	トレース・メッセージ
XRD_INFO	情報メッセージ
XRD_ERROR	エラー・メッセージ
XRD_FATAL	致命エラー・メッセージ

トレース・レベル定数

CWConnectorLogAndTrace クラスに定義されているトレース・レベル定数については、表 135 に要約をまとめます。

表 135. CWConnectorLogAndTrace クラスのトレース・レベル定数

定数名	意味
LEVEL0	トレース・レベル 0 (トレースをオフにします)
LEVEL1	トレース・レベル 1
LEVEL2	トレース・レベル 2
LEVEL3	トレース・レベル 3
LEVEL4	トレース・レベル 4
LEVEL5	トレース・レベル 5

第 21 章 CWConnectorReturnStatusDescriptor クラス

Java コネクターは CWConnectorReturnStatusDescriptor クラスを使用して、戻り状況記述子の中にエラー・メッセージおよび情報メッセージを戻すことができます。この記述子にはその他の状況情報も含まれていて、それらの情報は通常、要求を開始した統合ブローカーに送信される要求応答の一部として戻されます。

WebSphere InterChange Server

InterChange Server を使用しているビジネス・インテグレーション・システムの場合、コネクター・フレームワークによって、要求を開始したコラボレーションに戻り状況記述子が戻されます。コラボレーションは、この戻り状況記述子内の情報にアクセスし、そのサービス呼び出し要求の状況を取得することができます。

注: 下位の Java コネクター・ライブラリーの ReturnStatusDescriptor クラスは、CWConnectorReturnStatusDescriptor クラスによって拡張されます。下位の Java コネクター・ライブラリーのクラスの詳細については、 465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

表 136 に、CWConnectorReturnStatusDescriptor クラスのメソッドについての要約をまとめます。

表 136. CWConnectorReturnStatusDescriptor クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorReturnStatusDescriptor()	戻り状況記述子を作成します。	393
getErrorString()	戻り状況記述子からメッセージ・ストリングを取得します。	394
getStatus()	戻り状況記述子から状況コードの値を取得します。	394
setErrorString()	戻り状況記述子の中に、エラー・メッセージまたは情報メッセージを設定します。	395
setStatus()	戻り状況記述子の中に状況コードの値を設定します。	395

CWConnectorReturnStatusDescriptor()

戻り状況記述子を作成します。

構文

```
public CWConnectorReturnStatusDescriptor();
```

パラメーター

なし。

戻り値

新規に作成された戻り状況記述子を包含している CWConnectorReturnStatusDescriptor オブジェクト。

---

getErrorString()

戻り状況記述子からメッセージ・ストリングを取得します。

構文

```
public String getErrorString();
```

パラメーター

なし。

戻り値

統合ブローカーに対するエラー・メッセージまたは情報メッセージを含む String、または null です。

例外

なし。

注記

getErrorString() メソッドは、メッセージ (エラー・メッセージまたは情報メッセージ) を戻します。

参照項目

setErrorString()

---

getStatus()

戻り状況記述子から状況コードの値を取得します。

構文

```
public int getStatus();
```

パラメーター

なし。

戻り値

操作の状況を示す int の値。

例外

なし。

参照項目

setStatus()

setErrorString()

戻り状況記述子の中に、エラー・メッセージまたは情報メッセージを設定します。

構文

```
public void setErrorString(String errorStr);
```

パラメーター

errorStr メッセージ・ストリングを設定する値です。

戻り値

なし。

例外

なし。

参照項目

getErrorString()

setStatus()

戻り状況記述子の中に状況コードの値を設定します。

構文

```
public void setStatus(int status);
```

パラメーター

status 戻り状況記述子に割り当てる状況コードの値です。

戻り値

なし。

例外

なし。

参照項目

getStatus()

第 22 章 CWConnectorUtil クラス

CWConnectorUtil クラスには、各種のユーティリティー・メソッドが含まれます。

注: 下位の Java コネクター・ライブラリーの JavaConnectorUtil クラスは、CWConnectorUtil クラスによって拡張されます。下位の Java コネクター・ライブラリーのクラスの詳細については、465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

このクラスの内容は以下のとおりです。

- 『メッセージ・ファイル定数』
- 『メソッド』

メッセージ・ファイル定数

表 137 は、CWConnectorUtil クラスで定義されている静的メッセージ・ファイル定数の要約を示しています。

表 137. CWConnectorUtil クラスのメッセージ・ファイル定数

定数名	意味
CONNECTOR_MESSAGE_FILE	コネクター・メッセージ・ファイルを使用して、メッセージを生成します。
INFRASTRUCTURE_MESSAGE_FILE	InterChange Server メッセージ・ファイル (InterchangeSystem.txt) は、メッセージの生成に使用します。 **重要:** コネクターは InterchangeSystem.txt ファイルからメッセージを取得してはなりません。コネクターは、必ず自身のローカル・コネクター・メッセージ・ファイルを使用する必要があります。

メソッド

CWConnectorUtil クラスには、Java コネクターで使用される各種ユーティリティー・メソッドが含まれています。これらのユーティリティー・メソッドは、以下の一般カテゴリーに分類されます。

- メッセージの生成およびロギングのための静的メソッド
- ビジネス・オブジェクトの作成のための静的メソッド
- コネクター構成プロパティーの取得のための静的メソッド
- ロケール情報の取得のためのメソッド

CWConnectorUtil クラスのメソッドについては、表 138 に要約をまとめます。

表 138. *CWConnectorUtil* クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWConnectorUtil()	CWConnectorUtil オブジェクトを作成します。	399
boToByteArray()	データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、バイト配列を介してアクセスできます。	399
boToStream()	データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、入力ストリームを介してアクセスできます。	402
boToString()	データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、ストリングとしてアクセスできます。	404
byteArrayToBo()	データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データには、バイト配列を介してアクセスします。	406
createAndCopyKeyVals()	新規ビジネス・オブジェクトを作成して、指定されたキー値および動詞を割り当て、残りの属性にデフォルト値を割り当てます。	408
createAndSetDefaults()	新規ビジネス・オブジェクトを作成して、その属性のすべてにデフォルト値を割り当てます。	409
createBusObj()	新しいビジネス・オブジェクトを作成します。	410
generateAndLogMsg()	メッセージを生成してログの宛先に送信します。	411
generateAndTraceMsg()	トレース・メッセージを生成してログの宛先に送信します。	412
generateMsg()	メッセージ・ファイル内の一連の定義済みメッセージからメッセージを生成します。	414
getAllConfigProperties()	プロパティーのタイプ (シンプル、階層、または多値) に関わらず、すべてのコネクター構成プロパティーのリストを取得します。	415
getAllConnectorAgentProperties()	現在のコネクターのすべてのコネクター構成プロパティーのリストを取得します。ただし、これらのプロパティーは単一値のプロパティーとして取得されます。	416
getBlankValue()	特殊な Blank 属性値の値を取得します。	417
getConfigProp()	コネクター構成プロパティーの値を取得します。	417
getGlobalEncoding()	コネクター・フレームワークが使用している文字エンコードを取得します。	418
getGlobalLocale()	コネクター・フレームワークのロケールを取得します。	419
getHierarchicalConfigProp()	階層コネクター構成プロパティーの値を取得します。	420
getIgnoreValue()	特殊な「Ignore」属性値の値を取得します。	421
getSupportedBONames()	現在のコネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。	421
getVersion()	コネクターのバージョンを取得します。	422
initAndValidateAttributes()	属性をそれぞれデフォルト値に設定して初期化し、その属性を検証します。	423

表 138. *CWConnectorUtil* クラスのメンバー・メソッド (続き)

メンバー・メソッド	説明	ページ
isBlankValue()	属性値が特殊な Blank 値であるかどうかを判別します。	424
isIgnoreValue()	属性値が特殊な Ignore 値であるかどうかを判別します。	425
isTraceEnabled()	このレベルでのトレースが有効になっている場合に、そのトレース・レベルが、参照しているトレース・レベル以上であるかどうかを判別します。	425
logMsg()	コネクターのログの宛先にメッセージを記録します。	426
readerToBO()	データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データには、Reader オブジェクトを使用してアクセスします。	427
streamToBO()	データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データには、入力ストリームを介してアクセスします。	429
stringToBo()	データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データは、ストリングとしてアクセスされます。	431
traceCWConnectorAPIVersion()	Java コネクター・ライブラリーのバージョンをトレース・レベル 1 でトレースします。	434
traceWrite()	ログの宛先にトレース・メッセージを書き込みます。	434

CWConnectorUtil()

`CWConnectorUtil` オブジェクトを作成します。

構文

```
public CWConnectorUtil();
```

パラメーター

なし。

戻り値

`CWConnectorUtil` オブジェクト。

boToByteArray()

データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、バイト配列としてアクセスできます。

構文

```
public static byte[] boToByteArray(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix *mimeType* と結合して子メタオブジェクトのキーを形成するオプションのビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config データ・ハンドラーの追加の構成情報を含む Object。

encoding byte 配列における直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale byte 配列における直列化データのロケールを指定する java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType ビジネス・オブジェクトが変換される直列化フォーマットを指定する MIME タイプ。

theBusObj 指定された MIME タイプに直列化し、byte 配列を戻すビジネス・オブジェクトです。

戻り値

指定された MIME タイプの直列化ビジネス・オブジェクトを含む byte 配列。

例外

DataHandlerCreateException
: boToByteArray() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
: データ・ハンドラーがビジネス・オブジェクトを指定された MIME タイプに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
: DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーが byte 配列の代わりに null ポインターを戻した場合は、boToByteArray() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

boToByteArray() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してビジネス・オブジェクトからストリングへの変換を実行できるようになります。このメソッドを使用すると、結果として生成される直列化データには、Java byte 配列を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のよ

うに、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、boToByteArray() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  boToByteArray() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、boToByteArray() によって DataHandlerCreateException 例外がスローされます。 boToByteArray() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

インスタンス化されたデータ・ハンドラーは、指定されたビジネス・オブジェクトを、MIME タイプで示された直列化フォーマットに変換します。boToByteArray() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの作成時に指定された文字エンコードおよびロケールを使用します。データ・ハンドラーは、この直列化データを byte 配列として boToByteArray() メソッドへ戻します。呼び出し側メソッドは、それを介して、戻された直列化データにアクセスできます。

注: データ・ハンドラーがビジネス・オブジェクトを変換できない場合、boToByteArray() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToStream(), boToString(), byteArrayToBo()

boToStream()

データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、入力ストリームを介してアクセスできます。

構文

```
public static InputStream boToStream(CWConnectorBusObj theBusObj, String mimeType);
public static InputStream boToStream(CwConnectorBusObj theBusObj, String mimeType,
      Object config);

public static InputStream boToStream(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix *mimeType* と結合して子メタオブジェクトのキーを形成するオプションのビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config データ・ハンドラーの追加の構成情報を含む Object。

encoding 入力ストリームにおける直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale 入力ストリームにおける直列化データのロケールを指定する java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType ビジネス・オブジェクトが変換される直列化フォーマットを指定する MIME タイプ。

theBusObj 指定された MIME タイプに直列化し、入力ストリームを戻すビジネス・オブジェクトです。

戻り値

指定された MIME タイプの直列化ビジネス・オブジェクトを含む、Java java.io.InputStream クラス (またはそのサブクラスのいずれか) のオブジェクト。

例外

DataHandlerCreateException
: boToStream() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
: データ・ハンドラーがビジネス・オブジェクトを指定された MIME タイプに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
: DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーが InputStream オブジェクトの代わりに null ポインターを戻した場合は、boToStream() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

boToStream() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してビジネス・オブジェクトからストリングへの変換を実行できるようになります。このメソッドを使用すると、結果として生成される直列化データには、Java 入力ストリーム (InputStream クラスに基づく) を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、boToStream() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  boToStream() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、boToStream() によって DataHandlerCreateException 例外がスローされます。 boToStream() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

インスタンス化されたデータ・ハンドラーは、指定されたビジネス・オブジェクトを、MIME タイプで示された直列化フォーマットに変換します。boToStream() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの作成時に指定された文字エンコードおよびロケールを使用します。データ・ハンドラーは、この直列化データを入力ストリームとして boToStream() メソッドへ戻します。呼び出し側メソッドは、それを介して、戻された直列化データにアクセスできます。

注: データ・ハンドラーがビジネス・オブジェクトを変換できない場合、boToStream() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToByteArray(), boToString(), streamToBO()

boToString()

データ・ハンドラーを呼び出して、ビジネス・オブジェクトを、指定された MIME タイプの直列化データに変換します。この直列化データには、ストリングとしてアクセスできます。

構文

```
public static String boToString(CWConnectorBusObj theBusObj, String mimeType);
public static String boToString(CwConnectorBusObj theBusObj, String mimeType,
      Object config);

public static String boToString(CwConnectorBusObj theBusObj, String mimeType,
      String BOPrefix, String encoding, Object config);
```

パラメーター

BOPrefix — *mimeType* と結合して子メタオブジェクトのキーを形成するビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、`BOPrefix` データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config — データ・ハンドラーの追加の構成情報を含む `Object`。

encoding — `String` における直列化データの文字エンコードを指定します。`null` を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale — `String` における直列化データのロケールを指定する `java.util.Locale` オブジェクトです。`null` を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType — ビジネス・オブジェクトが変換される直列化フォーマットを指定する MIME タイプ。

theBusObj — 指定された MIME タイプに直列化し、ストリングを戻すビジネス・オブジェクトです。

戻り値

指定された MIME タイプの直列化ビジネス・オブジェクトを含む `String` オブジェクト。

例外

DataHandlerCreateException
: boToString() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
: データ・ハンドラーがビジネス・オブジェクトを指定された MIME タイプに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
: DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーが String オブジェクトの代わりに null ポインターを戻した場合、boToString() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

boToString() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してビジネス・オブジェクトからストリングへの変換を実行できるようになります。このメソッドを使用すると、結果として生成される直列化データには、Java String 配列を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、boToString() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  boToString() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、boToString() によって DataHandlerCreateException 例外がスローされます。 boToString() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

インスタンス化されたデータ・ハンドラーは、指定されたビジネス・オブジェクトを、MIME タイプで示された直列化フォーマットに変換します。boToString() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの作成時に指定された文字エンコードおよびロケールを使用します。データ・ハンドラーは、この直列化データを String オブジェクトとして boToString() メソッドへ戻します。呼び出し側メソッドは、それを介して、戻された直列化データにアクセスできます。

注: データ・ハンドラーがビジネス・オブジェクトを変換できない場合、boToString() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToByteArray(), boToStream(), stringToBo()

byteArrayToBo()

データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データには、byte 配列としてアクセスできます。

構文

```
public static CWConnectorBusObj byteArrayToBo(CWConnectorBusObj theBusObj,
      byte[] serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix — *mimeType* と結合して子メタオブジェクトのキーを形成するビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config — データ・ハンドラーの追加の構成情報を含む Object。

encoding — byte 配列における直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale — byte 配列における直列化データのロケールを指定する

java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType 直列化データのフォーマットを識別する MIME タイプ。

serializedData ビジネス・オブジェクトに変換される直列化データを含む byte 配列です。

theBusObj メソッドが直列化データに変換するビジネス・オブジェクト (ビジネス・オブジェクト定義) の型を識別します。

戻り値

直列化データを表すビジネス・オブジェクトが格納されている CWConnectorBusObj オブジェクト。

例外

DataHandlerCreateException
byteArrayToBo() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
データ・ハンドラーが直列化データを指定されたビジネス・オブジェクトに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーがビジネス・オブジェクトの代わりに null ポインターを戻した場合、byteArrayToBo() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

byteArrayToBo() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してストリングからビジネス・オブジェクトへの変換を実行できるようになります。このメソッドを使用すると、受信した *serializedData* には、Java byte 配列を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、byteArrayToBo() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。

- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  byteArrayToBo() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、byteArrayToBo() によって DataHandlerCreateException 例外がスローされます。 byteArrayToBo() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

データ・ハンドラーはインスタンス化された後で、指定された直列化データを、*theBusObj* によって指示された型のビジネス・オブジェクトに変換します。byteArrayToBo() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの解釈時に指定された文字エンコードおよびロケールを使用します。このビジネス・オブジェクトは、データ・ハンドラーから byteArrayToBo() メソッドに戻された後、さらにこのメソッドから呼び出し側メソッドに戻されます。

注: データ・ハンドラーが直列化データを変換できない場合は、byteArrayToBo() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToByteArray(), readerToBO(), streamToBO(), stringToBo()

createAndCopyKeyVals()

新規ビジネス・オブジェクトを作成して、指定されたキー値および動詞を割り当て、残りの属性にデフォルト値を割り当てます。

構文

```
public static CWConnectorBusObj createAndCopyKeyVals(String busObjName,
        String keyVals, String verb, String delimiter)
```

パラメーター

busObjName 作成されるビジネス・オブジェクトの名前。

keyVals キー値のストリング。基本キー値を*区切り文字* で区切り、順番に並べたリスト。

verb 新規ビジネス・オブジェクトに割り当てる動詞。

delimiter キーの区切り文字。

戻り値

新規に作成されたビジネス・オブジェクトを包含している CWConnectorBusObj オブジェクト。

例外

SpecNameNotFoundException
: 指定された名前のビジネス・オブジェクト定義が見つからない場合、スローされます。

AttributeNotFoundException
: 属性が見つからない場合、スローされます。

AttributeValueException
: 属性が無効な値に設定されている場合、スローされます。

WrongAttributeException
: 属性の値がそのデータ型と一致しない場合、スローされます。

InvalidVerbException
: 動詞の値が無効な場合、スローされます。

注記

createAndCopyKeyVals() メソッドによって、以下のタスクが実行されます。

- *busObjName* によって指定された型の新規ビジネス・オブジェクトを作成します。
- *keyVals* キー・ストリングを解析し、キー値を取得して、それらのキー値をビジネス・オブジェクトのキー属性に設定します。キー値がそれぞれ、指定された*区切り文字* の値で区切られていることが、このメソッドを使用するための前提条件となっています。
- 新規ビジネス・オブジェクトの動詞を *verb* に設定します。
- ビジネス・オブジェクトの残りの属性にデフォルトの属性値を割り当てます。

このメソッドは、後で処理するために統合ブローカーへ送信するビジネス・オブジェクトを構築する pollForEvents() メソッドで役立ちます。

createAndSetDefaults()

新規ビジネス・オブジェクトを作成して、その属性のすべてにデフォルト値を割り当てます。

構文

```
public static CWConnectorBusObj createAndSetDefaults(
   String busObjName)
```

パラメーター

busObjName 作成されるビジネス・オブジェクトの名前。

戻り値

新規に作成されたビジネス・オブジェクトを包含している CWConnectorBusObj オブジェクト。

例外

SpecNameNotFoundException
: 指定された名前のビジネス・オブジェクト定義が見つからない場合、スローされます。

AttributeNotFoundException
: (ビジネス・オブジェクト定義で定義された) ビジネス・オブジェクトの属性のいずれかが見つからない場合、スローされます。

注記

createAndSetDefaults() メソッドによって、以下のタスクが実行されます。

- *busObjName* によって指定された型の新規ビジネス・オブジェクトを作成します。
- ビジネス・オブジェクトの属性すべてにデフォルトの属性値を割り当てます。

createBusObj()

新しいビジネス・オブジェクトを作成します。

構文

```
public static final CWConnectorBusObj createBusObj(String busObjName);

public static final CWConnectorBusObj createBusObj(String busObjName,
    Locale localeObject);

public static final CWConnectorBusObj createBusObj(String busObjName,
    String localeName);
```

パラメーター

busObjName 作成されるビジネス・オブジェクトの名前を指定します。

localeObject ビジネス・オブジェクトに関連付けるロケールを識別する Java Locale オブジェクト。

localeName ビジネス・オブジェクトに関連付けるロケールの名前。

戻り値

新規に作成されたビジネス・オブジェクトを包含している CWConnectorBusObj オブジェクト。

例外

SpecNameNotFoundException
: 指定された名前のビジネス・オブジェクト定義が見つからない場合、スローされます。

注記

createBusObj() メソッドは、ユーザーが *busObjName* に指定するビジネス・オブジェクト定義型の、新規ビジネス・オブジェクト・インスタンスを作成します。*localeObject* または *localeName* を指定した場合、このビジネス・オブジェクト・ロケールは、ビジネス・オブジェクト内のデータに適用されますが、ビジネス・オブジェクト定義の名前またはその属性には*適用されません* (これは、米国英語のロケール en_US に関連付けられたコード・セット内の文字でなければなりません)。*localeName* のフォーマットの説明については、64 ページの『国際化対応コネクターの設計上の考慮事項』を参照してください。

参照項目

getLocale()

generateAndLogMsg()

メッセージを生成してログの宛先に送信します。

構文

```
public static void generateAndLogMsg(int msgNum, int msgType, int isGlobal);

public static void generateAndLogMsg(int msgNum, int msgType, int isGlobal,
     msgParameters);
```

パラメーター

isGlobal メッセージ・ファイルがコネクター・メッセージ・ファイルであることを示すために CWConnectorUtil クラスで定義されている、CONNECTOR_MESSAGE_FILE メッセージ・ファイル定数。

msgNum メッセージ・ファイル内のメッセージ番号 (ID) を指定します。

msgParameters 任意に指定する String パラメーター値のリスト。それぞれの値は、メッセージ・リスト内のパラメーターに対応しています。最大 10 個のパラメーターを指定できます。

msgType CWConnectorLogAndTrace クラスで定義されている、以下のいずれかのメッセージ・タイプ定数。この定数により、メッセージ重大度が識別されます。

```
CWConnectorLogAndTrace.XRD_WARNING
CWConnectorLogAndTrace.XRD_ERROR
CWConnectorLogAndTrace.XRD_FATAL
CWConnectorLogAndTrace.XRD_INFO
CWConnectorLogAndTrace.XRD_TRACE
```

戻り値

なし。

例外

なし。

注記

generateAndLogMsg() メソッドは、メッセージ生成機能の generateMsg() とメッセージ・ロギング機能の logMsg() を組み合わせたメソッドです。このメソッドは、メッセージ・ファイルからメッセージを生成し、ログ宛先に送信します。コネクターのログの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「ロギング」セクションを通じて設定します。

このメソッドは、可変数のストリング引き数を取ることが可能で、合計 10 個までのパラメーター値をサポートしています。つまり、10 個までの String の値を generateAndLogMsg() への引き数として指定できます。以下の呼び出しでは、コネクター・メッセージ・ファイル内に定義されている、エラー・メッセージ 3223 内の 7 つのパラメーターに値が提供されます。

```
generateAndLogMsg(3223, CWConnectorLogAndTrace:XRD_ERROR,
        CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
        value1, value2, value3, value4, value5, value6, value7);
```

WebSphere InterChange Server

severity が XRD_ERROR または XRD_FATAL であり、コネクター構成プロパティー LogAtInterchangeEnd が設定されている場合は、エラー・メッセージがログに記録されます。電子メール通知がオンの場合は、電子メール通知が送信されます。エラーの場合に備えて電子メール通知をセットアップする方法については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

ログ・メッセージをメッセージ・ファイルに格納して generateAndLogMsg() メソッドで抽出するよう、IBM では推奨しています。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。

generateAndLogMsg() によって記録されたコネクター・メッセージは、LogViewer を使用して見ることができます。

参照項目

generateAndTraceMsg(),, generateMsg(),, logMsg()

generateAndTraceMsg()

トレース・メッセージを生成してトレース宛先に送信します。

構文

```
public static void generateAndTraceMsg(int traceLevel, int msgNum,
    int isGlobal);

public static void generateAndTraceMsg(int traceLevel, int msgNum,
    int isGlobal, msgParameters);
```

パラメーター

isGlobal　　メッセージ・ファイルがコネクター・メッセージ・ファイルである

ことを示すために CWConnectorUtil クラスで定義されている、CONNECTOR_MESSAGE_FILE メッセージ・ファイル定数。

msgNum メッセージ・ファイル内のメッセージ番号 (ID) を指定します。

msgParameters 任意に指定する String パラメーター値のリスト。それぞれの値は、メッセージ・リスト内のパラメーターに対応しています。最大 10 個のパラメーターを指定できます。

traceLevel CWConnectorLogAndTrace クラスに定義されている、以下のいずれかのトレース・レベル定数。出力対象トレース・メッセージの判別に用いられるトレース・レベルを示します。

```
CWConnectorLogAndTrace.LEVEL1
CWConnectorLogAndTrace.LEVEL2
CWConnectorLogAndTrace.LEVEL3
CWConnectorLogAndTrace.LEVEL4
CWConnectorLogAndTrace.LEVEL5
```

現在のトレース・レベルが *traceLevel* より大きいか等しい場合、メソッドはトレース・メッセージを書き込みます。

注: トレース・メッセージには、トレース・レベル 0 (LEVEL0) を指定しないでください。トレース・レベル 0 は、トレースがオフになっていることを示します。したがって、LEVEL0 の *traceLevel* に関連するトレース・メッセージが、印刷されることは絶対にありません。

戻り値

なし。

例外

なし。

注記

generateAndTraceMsg() メソッドは、メッセージ生成機能の generateMsg() とメッセージ・トレース機能の traceWrite() を組み合わせたメソッドです。このメソッドは、メッセージ・ファイルからメッセージを生成し、トレース宛先に送信します。コネクターのトレースの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「トレース」セクションを通じて設定します。

このメソッドは、可変数のストリング引き数を取ることが可能で、合計 10 個までのパラメーター値をサポートしています。つまり、10 個までの String の値を、generateAndTraceMsg() の *msgParameters* パラメーターの引き数として指定できます。以下の呼び出しでは、コネクター・メッセージ・ファイル内に定義されている、トレース・メッセージ 668 内の 7 つのパラメーターに値が与えられます。

```
generateAndTraceMsg(CWConnectorLogAndTrace.LEVEL3, 668,
        CWConnectorUtil.CONNECTOR_MESSAGE_FILE,
        value1, value2, value3, value4, value5, value6, value7);
```

通常、トレース・メッセージはデバッグ時にのみ必要となるため、トレース・メッセージをメッセージ・ファイルに含めるかどうかは、以下のように開発者が任意に決定することができます。

- 英語圏外のユーザーがトレース・メッセージを参照する必要がある場合、それらのメッセージは国際化対応にする必要があります。そのため、トレース・メッセージをメッセージ・ファイルに入れて、generateAndTraceMsg() メソッドで抽出する必要があります。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。
- 英語圏のユーザーのみがトレース・メッセージを参照する必要がある場合、それらのメッセージを国際化対応にする必要はありません。そのため、(英語の) トレース・メッセージを直接 traceWrite() の呼び出しに組み込むことができます。generateMsg() または generateAndTraceMsg() メソッドを使用する必要はありません。

generateAndTraceMsg() によって記録されたコネクター・メッセージは、LogViewer を使用して見ることはできません。

参照項目

generateAndLogMsg(),, generateMsg(),, traceWrite()

generateMsg()

メッセージ・ファイル内の一連の定義済みメッセージからメッセージを生成します。

構文

```
public final static String generateMsg(int traceLevel, int msgNum,
    int msgType, int isGlobal, int argCount,
    Vector msgParams);

public final static String generateMsg(int msgNum, int msgType,
    int isGlobal, int argCount, Vector msgParams);
```

パラメーター

traceLevel CWConnectorLogAndTrace クラスに定義されている、以下のいずれかのトレース・レベル定数。メッセージを生成するトレース・レベルを指定します。

```
CWConnectorLogAndTrace.LEVEL1
CWConnectorLogAndTrace.LEVEL2
CWConnectorLogAndTrace.LEVEL3
CWConnectorLogAndTrace.LEVEL4
CWConnectorLogAndTrace.LEVEL5
```

このパラメーターを省略すると、メソッドは、トレース・レベルに無関係にメッセージを生成します。メッセージは、*traceLevel* 値が、コネクターの現在のトレース・レベル以下である場合に限り生成されます。

msgNum メッセージ・ファイル内のメッセージ番号 (ID) を指定します。

msgType CWConnectorLogAndTrace クラスに定義されている、以下のいずれかのメッセージ・タイプ定数。この定数により、メッセージが識別されます。

CWConnectorLogAndTrace.XRD_WARNING
CWConnectorLogAndTrace.XRD_ERROR
CWConnectorLogAndTrace.XRD_FATAL
CWConnectorLogAndTrace.XRD_INFO
CWConnectorLogAndTrace.XRD_TRACE

isGlobal メッセージ・ファイルがコネクター・メッセージ・ファイルであることを示すために CWConnectorUtil クラスで定義されている、CONNECTOR_MESSAGE_FILE メッセージ・ファイル定数。

argCount メッセージ・テキスト内のパラメーターの個数を示す整数。個数を確認するには、メッセージ・ファイル内のメッセージを参照します。

msgParams メッセージ・テキスト用のパラメーターのリスト。

戻り値

生成済みのメッセージが格納されている String。最初の形式のメソッドでは、トレース・レベルがコネクターのトレース・レベルよりも大きい場合、null が戻されます。

例外

なし。

注記

generateMsg() メソッドの形式には、以下の 2 種類があります。

- メッセージのトレースに対して、最初の形式のメソッドを使用します (そこでは、*traceLevel* が 1 番目のパラメーターです)。メッセージを生成するためには、トレース・レベルをコネクターのトレース・レベル以下にする必要があります。次に traceWrite() メソッドを使用して、トレース・メッセージをトレース宛先に送信します。

  generateAndTraceMsg() メソッドを使用すると、メッセージ生成とトレースを 1 つのステップにまとめて実行できます。
- ロギングに対して、2 番目の形式のシグニチャーを使用します (そこでは、*msgNum* が 1 番目のパラメーターです)。次に logMsg() メソッドを使用してログ・メッセージをログ宛先に送信します。

  generateAndLogMsg() メソッドを使用すると、メッセージ生成とロギングを 1 つのステップにまとめて実行できます。

参照項目

generateAndLogMsg(),, generateAndTraceMsg(),, logMsg(),, traceWrite()

getAllConfigProperties()

階層コネクター・プロパティーとして、現在のコネクター用のすべての構成プロパティーのリストを取得します。

構文

```
public static CWProperty[] getAllConfigProperties();
```

パラメーター

なし。

戻り値

CWProperty オブジェクトの配列への参照。各配列は、現在のコネクターのコネクター・プロパティーを 1 つ含んでいます。

例外

なし。

注記

getAllConfigProperties() メソッドは、CWProperty オブジェクトの配列としてコネクター構成プロパティーを取得します。各コネクター・プロパティー (CWProperty) オブジェクトには、単一のコネクター・プロパティーが含まれ、単一値、別のプロパティー、または値と子プロパティーの組み合わせを保持することができます。CWProperty クラスのメソッド (getHierChildProps() および getStringValues() など) を使用して、コネクター・プロパティー・オブジェクトから値を取得します。

参照項目

getConfigProp(),, getAllConnectorAgentProperties()

getAllConnectorAgentProperties()

現在のコネクター用の全構成プロパティーのリストを取得します。

構文

```
public static Hashtable getAllConnectorAgentProperties();
```

パラメーター

なし。

戻り値

現在のコネクター用のコネクター・プロパティーを包含する java.util.Hashtable オブジェクトへの参照。

例外

なし。

注記

getAllConnectorAgentProperties() メソッドは、キーを値にマップする Java Hashtable オブジェクトとして、コネクター構成プロパティーを取得します。キーはプロパティーの名前であり、値は関連付けられているプロパティーの値です。Hashtable クラスのメソッド (例えば keys() および elements()) を使用して、この構造体から情報を取得してください。

注: このメソッドは階層または多値のプロパティーは取得しません。多値のプロパティーを取得しようとすると、最初の値のみが戻されます。階層または多値のプロパティーを取得する場合は、`getAllConfigProperties()` メソッドを使用してください。

例

```
Hashtable ht = new Hastable();
ht = CWConnectorUtil.getAllConnectorAgentProperties();
int size = ht.size();
Enumeration properties = ht.keys();
Enumeration values = ht.elements();

while (properties.hasMoreElements()) {
    System.out.print((String)properties.nextElement());
    System.out.print("=");
    System.out.println((String)values.nextElement());
    System.out.println("Property set");
}
```

参照項目

getConfigProp(),, getAllConfigProperties()

getBlankValue()

特殊な Blank 属性値の値を取得します。

構文

```
public static String getBlankValue();
```

戻り値

Blank 属性値が含まれている `String` オブジェクト。

注記

`getBlankValue()` によって取得される Blank 値は、「ヌル」または長さがゼロの値を表す特殊な属性値です。Java コネクター・ライブラリーでは、Blank 属性値の `CWConnectorAttrType.CxBlank` 定数は提供されませんが、Blank 値を属性に割り当てる場合は、`getBlankValue()` メソッドを使用して Blank 値を取得することをお勧めします。

参照項目

getIgnoreValue()

getConfigProp()

コネクター構成プロパティーの値を取得します。

構文

```
public final static String getConfigProp(String propName);
```

パラメーター

propName 取得するプロパティーの名前。

戻り値

プロパティー値を含む String オブジェクトです。プロパティー名が見つからない場合、メソッドは null を戻します。

例外

なし。

注記

コネクター構成プロパティーの値は、その値の初期化の間に、コネクターにダウンロードされます。

並列処理コネクター (ParallelProcessDegree コネクター・プロパティーに 1 より大きい値が設定されているコネクター) で、getConfigProp("ConnectorName") を呼び出すと、マスター・プロセスまたはスレーブ・プロセスのどちらに呼び出したかには関係なく、常にコネクター・エージェントのマスター・プロセスの名前がメソッドによって戻されます。

参照項目

getAllConnectorAgentProperties(),, getHierarchicalConfigProp()

getGlobalEncoding()

コネクター・フレームワークが使用している文字エンコードを取得します。

構文

```
public String getGlobalEncoding();
```

パラメーター

なし。

戻り値

コネクター・フレームワークの文字エンコードが格納されている String オブジェクト。

例外

なし。

注記

getGlobalEncoding() メソッドは、コネクター・フレームワークの文字エンコード (ロケールの一部) を取得します。ロケールには、言語、国 (または地域)、および文字エンコードに応じて、データの国/地域別情報を指定します。コネクター・フレームワークの文字エンコードには、コネクター・アプリケーションの文字エンコードが示されていなければなりません。コネクター・フレームワークの文字エンコードには、以下の階層が使用されています。

- リポジトリー内の CharacterEncoding コネクター構成プロパティー

WebSphere InterChange Server

ローカル構成ファイルが存在する場合、このローカル・ファイルにおける CharacterEncoding コネクター構成プロパティーの設定値が優先します。ローカル構成ファイルが存在しない場合、CharacterEncoding プロパティーの設定値は、コネクターの始動時に InterChange Server リポジトリーからダウンロードされたコネクター構成プロパティーのセットから得られます。

- Java 環境からの文字エンコード (Unicode (UCS-2))

このメソッドは、文字変換などの文字エンコード処理をコネクターが実行しなければならない場合に役に立ちます。

参照項目

getGlobalLocale()

getGlobalLocale()

コネクター・フレームワークのロケールを取得します。

構文

```
public static String getGlobalLocale();
```

パラメーター

なし。

戻り値

コネクター・フレームワークのロケール設定値が格納されている String オブジェクト。

例外

なし。

注記

getGlobalLocale() メソッドは、コネクター・フレームワークのロケールを取得します。このロケールには、言語、国 (または地域)、および文字エンコードに応じた、データの国/地域別情報が定義されています。コネクター・フレームワークのロケールには、コネクター・アプリケーションのロケールが示されていなければなりません。コネクター・フレームワークのロケールを設定する際には、以下の階層が使用されます。

- リポジトリー内の LOCALE コネクター構成プロパティー

WebSphere InterChange Server

ローカル構成ファイルが存在する場合、このローカル・ファイルにおける Locale コネクター構成プロパティーの設定値が優先します。ローカル構成ファイルが存在しない場合、Locale プロパティーの設定値は、コネクターの始動時に InterChange Server リポジトリーからダウンロードされたコネクター構成プロパティーのセットから得られます。

- Java 環境からのロケール (オペレーティング・システムからのロケール)

このメソッドは、ロケール依存の処理をコネクターが実行しなければならない場合に役に立ちます。

参照項目

createBusObj(),, getGlobalEncoding(),, getLocale()

getHierarchicalConfigProp()

指定された階層コネクター構成プロパティーのトップレベル・コネクター・オブジェクトを取得します。

構文

```
public static CWProperty getHierarchicalConfigProp(String propName);
```

パラメーター

propName 取得対象の階層コネクター・プロパティーの名前です。

戻り値

指定された階層コネクター・プロパティーのトップレベルのコネクター・プロパティー・オブジェクトを含む CWProperty オブジェクト。

例外

WrongPropertyException
: 指定されたコネクター・プロパティー名がこのコネクターに存在しないか、または階層コネクター・プロパティーではない場合にスローされます。

注記

getHierarchicalConfigProp() メソッドは、トップレベルのコネクター・プロパティー (CWProperty) オブジェクトを取得します。取得したこのオブジェクトから、CWProperty クラスのメソッドを使用して、コネクター・プロパティーの必要な値を取得できます。

注: コネクター構成プロパティーの値は、その値の初期化の間に、コネクターにダウンロードされます。ダウンロードされていないコネクター・プロパティーの *propName* を指定すると、getHierarchicalConfigProp() は WrongPropertyException 例外をスローします。

並列処理コネクター (ParallelProcessDegree コネクター・プロパティーに 1 より大きい値が設定されているコネクター) で、getHierarchicalConfigProp("ConnectorName") を呼び出すと、マスター・プロセスまたはスレーブ・プロセスのどちらで呼び出したかには関係なく、常にコネクター・エージェントのマスター・プロセスの名前がメソッドによって戻されます。

参照項目

getAllConfigProperties(),, getConfigProp()

getIgnoreValue()

特殊な Ignore 属性値の値を取得します。

構文

```
public static String getIgnoreValue();
```

パラメーター

なし。

戻り値

Ignore 属性値が格納されている String オブジェクト。

例外

なし。

注記

getIgnoreValue() によって取得される Ignore 値は、コネクターが無視できる属性値を表す特殊な属性値です。Java コネクター・ライブラリーでは、Ignore 属性値の CWConnectorAttrType.CxIgnore 定数は提供されませんが、Ignore 値を属性に割り当てる場合は、getIgnoreValue() メソッドを使用して、Ignore 値を取得することをお勧めします。

参照項目

getBlankValue()

getSupportedBONames()

現在のコネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。

WebSphere InterChange Server

getSupportedBusObjNames() メソッドは、統合ブローカーが InterChange Server (ICS) である場合のみ 有効です。これは、ICS バージョン 4.0 以降でのみ、サポートされたビジネス・オブジェクトを提供します。4.0 より以前のバージョンの ICS では、このメソッドは FunctionalityNotImplementedException 例外をスローします。

構文

```
public static String[] getSupportedBONames()
```

パラメーター

なし。

戻り値

コネクターに対してサポートされているビジネス・オブジェクトの名前のリストを含む String 配列です。

例外

NotSupportedException
: バージョン 3.x InterChange Server を統合ブローカーとしているコネクターの内部からメソッドが呼び出された場合、スローされます。

注記

getSupportedBONames() メソッドは、現在のコネクターに対するトップレベルのサポートされているビジネス・オブジェクトのリストを戻します。すなわち、コネクターが、子ビジネス・オブジェクトを含むビジネス・オブジェクトをサポートしている場合、getSupportedBONames() は、親オブジェクトの名前のみを、その戻りリストの中に含めます。

注: getSupportedBONames() メソッドがサポートされるのは、コネクターがバージョン 4.0 以降の InterChange Server を統合ブローカーとして使用している場合のみ です。コネクターがそれ以前のバージョンの InterChange Server を使用している場合は、メソッドが NotSupportedException 例外を戻します。

getVersion()

コネクターのバージョンを取得します。

構文

```
public static String getVersion();
```

戻り値

コネクターのバージョンが格納されている String。

例外

なし。

注記

getVersion() メソッドは、Java コネクター・ライブラリーのバージョンを戻します。このメソッドは、Java パッケージ内で提供されるマニフェスト・ファイルからバージョンを取得します。

注: CWConnectorAgent クラスにも getVersion() メソッドが用意されています。ただし、このメソッドはコネクター自体のバージョンを取得します。

initAndValidateAttributes()

値が設定されていない属性のうち、必須としてマークされているものを、デフォルト値で初期化します。

構文

```
public static final void initAndValidateAttributes(
   CWConnectorBusObj theBusObj) ;
```

パラメーター

theBusObj このメソッドによって設定される属性を持ったビジネス・オブジェクト。

戻り値

なし。

例外

SpecNameNotFoundException
: 指定されたビジネス・オブジェクトの名前が、リポジトリー内のビジネス・オブジェクト定義のいずれにも一致しない場合、スローされます。

DefaultSettingFailedException
: 属性のデフォルト値が設定できず、ビジネス・オブジェクト定義内の属性に対して指定されたデフォルト値が存在しなかった場合にスローされます。

注記

initAndValidateAttributes() メソッドには、次の 2 つの目的があります。

- 属性を*初期化* し、以下の条件に当てはまる場合には、各属性の値としてデフォルト値を設定します。
  - UseDefaults コネクター構成プロパティーが true に設定されている場合
  - 属性の isRequired プロパティーが true に設定されている場合 (その属性が必須の場合)
  - 属性の値が現在設定されていない (特殊な Ignore 値 CxIgnore になっている) 場合
  - 属性のデフォルト値プロパティーでデフォルト値が指定されている場合
- 属性を*検証* し、以下の条件にあてはまる場合には DefaultSettingFailedException 例外をスローします。
  - 属性の isRequired プロパティーが true に設定されている場合
  - 属性のデフォルト値プロパティーでそのデフォルト値が定義されていない 場合

障害が発生した場合には、initAndValidateAttributes() がデフォルト値処理を終了した後で、いくつかの属性 (デフォルト値が指定されていない属性) で値が存

在しなくなります。この例外をキャッチして CWConnectorConstant.FAIL を戻すように、コネクターのアプリケーション固有コンポーネントをコーディングしておくことが可能です。

initAndValidateAttributes() メソッドは、ビジネス・オブジェクトの全レベルにおいてあらゆる属性を参照し、以下のことを判別します。

- 属性が必須であるかどうか
- 属性がビジネス・オブジェクト・インスタンスに値を持っているかどうか
- UseDefaults 構成プロパティーが true に設定されているかどうか

属性が必須で、UseDefaults が true である場合、initAndValidateAttributes() は、設定されていない属性の値を、デフォルト値に設定します。initAndValidateAttributes() に、属性値を特殊な Blank 値 (CxBlank) に設定させるために、属性のデフォルト値を「CxBlank」というストリングに設定することができます。属性にデフォルト値が指定されていない 場合、initAndValidateAttributes() は DefaultSettingFailedException 例外をスローします。

注: 属性がキーの場合、または属性の値がアプリケーションによって生成される場合は、ビジネス・オブジェクト定義のデフォルト値は使用されず、その属性の Required プロパティーには、false が設定されます。

アプリケーションで Create 操作を実行する前に必須属性の値が設定されるようにするために、initAndValidateAttributes() メソッドは通常、ビジネス・オブジェクト・ハンドラーの doVerbFor() メソッドから呼び出されます。 doVerbFor() メソッド内では、Create 動詞用に initAndValidateAttributes() を呼び出すことができます。また、Create を実行する前に、Update 動詞用としてこれを呼び出すこともできます。

initAndValidateAttributes() を使用するには、以下の操作も実行する必要があります。

- 必須属性の IsRequired 属性プロパティーが true に設定され、必須属性のデフォルト値がデフォルト値プロパティーで指定された値になるように、ビジネス・オブジェクトを設計します。
- コネクターのコネクター固有プロパティーのリストに、UseDefaults コネクター構成プロパティーを追加します。このプロパティーを true に設定してください。

isBlankValue()

属性値が特殊な Blank 値であるかどうかを判別します。

構文

```
public static boolean isBlankValue(Object value);
```

パラメーター

value 特殊な Blank 値と比較する値。

戻り値

指定された属性値が Blank 属性値である場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

なし。

参照項目

isIgnoreValue()

isIgnoreValue()

属性値が特殊な Ignore 値であるかどうかを判別します。

構文

```
public static boolean isIgnoreValue(Object value);
```

パラメーター

value 特殊な Ignore 値と比較する値。

戻り値

指定された属性値が Ignore 値である場合は `true` を戻します。それ以外の場合は `false` を戻します。

例外

なし。

注記

Ignore 属性値は、ビジネス・オブジェクトを処理している間、この属性を無視しなければならないことを示します。

参照項目

isBlankValue()

isTraceEnabled()

このレベルでのトレースが有効になっている場合に、そのトレース・レベルが、参照しているトレース・レベル以上であるかどうかを判別します。

構文

```
public final static boolean isTraceEnabled(int traceLevel);
```

パラメーター

traceLevel 検査の対象となるトレース・レベル。

戻り値

エージェントのトレース・レベルが、渡されたトレース・レベルより大きいか等しい場合、true を戻します。

例外

なし。

注記

このメソッドは、メッセージ生成の前に使用してください。

logMsg()

コネクターのログの宛先にメッセージを記録します。

構文

```
public final static void logMsg(String msg, int severity);
```

パラメーター

msg ログに記録されるメッセージ・テキスト。

severity メッセージを識別するメッセージ・タイプ定数。以下のいずれかです。

```
                              CWConnectorUtil.XRD_WARNING
                              CWConnectorUtil.XRD_ERROR
                              CWConnectorUtil.XRD_FATAL
                              CWConnectorUtil.XRD_INFO
CWConnectorUtil.XRD_TRACE
```

戻り値

なし。

例外

なし。

注記

logMsg() メソッドは、指定された *msg* テキストをログの宛先に送信します。コネクターのログの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「ロギング」セクションを通じて設定します。

ログ・メッセージをメッセージ・ファイルに格納して generateMsg() メソッドで抽出するよう、IBM では推奨しています。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。 generateMsg() メソッドは、logMsg() 用のメッセージ・ストリングを生成します。このメソッドによってメッセージ・ファイルから事前定義済みメッセージが取得され、テキストの書式が設定された後、生成されたメッセージ・ストリングが戻されます。

注: generateAndLogMsg() メソッドを使用すると、メッセージ生成とロギングを 1 つのステップにまとめて実行できます。

WebSphere InterChange Server

severity が XRD_ERROR または XRD_FATAL であり、コネクター構成プロパティー LogAtInterchangeEnd が設定されている場合は、エラー・メッセージがログに記録されます。電子メール通知がオンの場合は、電子メール通知が送信されます。エラーの場合に備えて電子メール通知をセットアップする方法については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

メッセージ・ストリングが generateMsg() で生成された場合、logMsg() で記録されたコネクター・メッセージは LogViewer を使用して見ることができます。

参照項目

generateAndLogMsg(),, generateMsg()

readerToBO()

データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データは、Reader オブジェクトとしてアクセスされます。

構文

```
public static CWConnectorBusObj readerToBO(CWConnectorBusObj theBusObj,
      Reader serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix *mimeType* と結合して子メタオブジェクトのキーを形成するビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config データ・ハンドラーの追加の構成情報を含む Object。

encoding Reader オブジェクトにおける直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale Reader オブジェクトにおける直列化データのロケールを指定する java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType 直列化データのフォーマットを識別する MIME タイプ。

serializedData ビジネス・オブジェクトに変換する直列化データにアクセスする Java java.io.Reader クラス (またはそのサブクラスのいずれか) のオブジェクトです。

theBusObj メソッドが直列化データに変換するビジネス・オブジェクト (ビジネス・オブジェクト定義) の型を識別します。

戻り値

直列化データを表すビジネス・オブジェクトが格納されている CWConnectorBusObj オブジェクト。

例外

DataHandlerCreateException
: readerToBO() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
: データ・ハンドラーが直列化データを指定されたビジネス・オブジェクトに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
: DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーがビジネス・オブジェクトの代わりに null ポインターを戻した場合、readerToBO() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

readerToBO() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してストリングからビジネス・オブジェクトへの変換を実行できるようになります。このメソッドを使用すると、受信した *serializedData* には、Java Reader オブジェクトを介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、readerToBO() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  readerToBO() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、こ

の値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、readerToBO() によって DataHandlerCreateException 例外がスローされます。 readerToBO() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

データ・ハンドラーはインスタンス化された後で、指定された直列化データを、*theBusObj* によって指示された型のビジネス・オブジェクトに変換します。readerToBO() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの解釈時に指定された文字エンコードおよびロケールを使用します。このビジネス・オブジェクトは、データ・ハンドラーから readerToBO() メソッドに戻された後、さらにこのメソッドから呼び出し側メソッドに戻されます。

注: データ・ハンドラーが直列化データを変換できない場合は、readerToBO() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

byteArrayToBo(), streamToBO(), stringToBo()

streamToBO()

データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データには、入力ストリームを介してアクセスします。

構文

```
public static CWConnectorBusObj streamToBO(CWConnectorBusObj theBusObj,
      InputStream serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix — *mimeType* と結合して子メタオブジェクトのキーを形成するビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config — データ・ハンドラーの追加の構成情報を含む Object。

encoding 入力ストリームにおける直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale 入力ストリームにおける直列化データのロケールを指定する java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType 直列化データのフォーマットを識別する MIME タイプ。

serializedData ビジネス・オブジェクトに変換する直列化データにアクセスする Java java.io.InputStream クラス (またはそのサブクラスのいずれか) のオブジェクトです。

theBusObj メソッドが直列化データに変換するビジネス・オブジェクト (ビジネス・オブジェクト定義) の型を識別します。

戻り値

直列化データを表すビジネス・オブジェクトが格納されている CWConnectorBusObj オブジェクト。

例外

DataHandlerCreateException
streamToBO() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
データ・ハンドラーが直列化データを指定されたビジネス・オブジェクトに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーがビジネス・オブジェクトの代わりに null ポインターを戻した場合、streamToBO() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

streamToBO() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してストリングからビジネス・オブジェクトへの変換を実行できるようになります。このメソッドを使用すると、受信した *serializedData* には、Java 入力ストリーム (InputStream クラスから派生) を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーか

ら、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、streamToBO() によって PropertyNotSetException 例外がスローされます。

- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  streamToBO() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない場合は、streamToBO() によって DataHandlerCreateException 例外がスローされます。 streamToBO() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

データ・ハンドラーはインスタンス化された後で、指定された直列化データを、*theBusObj* によって指示された型のビジネス・オブジェクトに変換します。streamToBO() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの解釈時に指定された文字エンコードおよびロケールを使用します。このビジネス・オブジェクトは、データ・ハンドラーから streamToBO() メソッドに戻された後、さらにこのメソッドから呼び出し側メソッドに戻されます。

注: データ・ハンドラーが直列化データを変換できない場合は、streamToBO() によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToStream(), byteArrayToBo(), readerToBO(), stringToBo()

stringToBo()

データ・ハンドラーを呼び出して、指定された MIME タイプの直列化データをビジネス・オブジェクトに変換します。この直列化データは、ストリングとしてアクセスされます。

構文

```
public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData,     String mimeType);
public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData, String mimeType, Object config);

public static CWConnectorBusObj stringToBo(CWConnectorBusObj theBusObj,
      String serializedData, String mimeType, String BOPrefix,
      String encoding, Locale locale, Object config);
```

パラメーター

BOPrefix *mimeType* と結合して子メタオブジェクトのキーを形成するビジネス・オブジェクト・プレフィックスです。この引き数は、MIME サブタイプを指定するために使用できます。また、BOPrefix データ・ハンドラー構成プロパティーの値を指定するためにも使用できます。

config データ・ハンドラーの追加の構成情報を含む Object。

encoding String における直列化データの文字エンコードを指定します。null を指定すると、このメソッドはマシンの文字エンコードを使用します。

locale String における直列化データのロケールを指定する java.util.Locale オブジェクトです。null を指定すると、このメソッドはコネクター・フレームワークのロケールを使用します。

mimeType 直列化データのフォーマットを識別する MIME タイプ。

serializedData ビジネス・オブジェクトに変換される直列化データのストリング表記。

theBusObj メソッドが直列化データに変換するビジネス・オブジェクト (ビジネス・オブジェクト定義) の型を識別します。

戻り値

直列化データを表すビジネス・オブジェクトが格納されている CWConnectorBusObj オブジェクト。

例外

DataHandlerCreateException
: stringToBo() メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。

ParseException
: データ・ハンドラーが直列化データを指定されたビジネス・オブジェクトに変換しているときに何らかのエラーが発生した場合、スローされます。

PropertyNotSetException
: DataHandlerMetaObjectName コネクター構成プロパティーが設定されていない場合、スローされます。

データ・ハンドラーがビジネス・オブジェクトの代わりに null ポインターを戻した場合、stringToBo() メソッドが、一般的な Java 例外 NullPointerException をスローすることもあります。

注記

stringToBo() メソッドを使用することにより、コネクターがデータ・ハンドラーを呼び出してストリングからビジネス・オブジェクトへの変換を実行できるようになります。このメソッドを使用すると、受信した *serializedData* には、Java String を介してアクセスできます。このメソッドは、指定された *mimeType* 引き数に基づいて、起動するデータ・ハンドラーを識別します。以下のように、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内で、クラス名が識別されるデータ・ハンドラーのインスタンスを生成します。

- メソッドは、トップレベルのメタオブジェクトをチェックして、この MIME タイプに対応するデータ・ハンドラーが存在しているかどうか確認します。また、このメソッドは、DataHandlerMetaObjectName コネクター構成プロパティーから、このトップレベルのメタオブジェクトの名前を取得します。このプロパティーが設定されている場合、stringToBo() によって PropertyNotSetException 例外がスローされます。
- インスタンス生成プロセスでは、指定された *mimeType* がそれと同等の MIME タイプ・ストリングに変換され、この MIME タイプ・ストリングに一致する名前を持つ属性がトップレベル・メタオブジェクトから取得されます。この属性の関連タイプは子メタオブジェクトです。
- インスタンスを生成するクラスの名前を、子メタオブジェクトの ClassName 属性から取得します。

  stringToBo() で *BOPrefix* と *mimeType* が指定されている場合は、トップレベル・メタオブジェクトでこの MIME タイプと関連付けられた子メタオブジェクト内でクラス名が識別されるデータ・ハンドラーのインスタンスが生成されます。ただし、*BOPrefix* が指定されている場合、インスタンス生成プロセスは、この値を MIME サブタイプとして解釈します。これは、トップレベル・メタオブジェクトから、名前に MIME タイプとサブタイプの*両方* が含まれている属性を取得します。

データ・ハンドラーをインスタンス化できない*場合*は、stringToBo() によって DataHandlerCreateException 例外がスローされます。 stringToBo() の引き数でインスタンスを生成するデータ・ハンドラーを識別する方法については、87 ページの『インスタンスを生成するデータ・ハンドラーの識別』を参照してください。

データ・ハンドラーはインスタンス化された後で、指定された直列化データを、*theBusObj* によって指示された型のビジネス・オブジェクトに変換します。stringToBo() が *encoding* および *locale* 引き数を指定する場合は、データ・ハンドラーは、直列化データの解釈時に指定された文字エンコードおよびロケールを使用します。このビジネス・オブジェクトは、データ・ハンドラーから stringToBo() メソッドに戻された後、さらにこのメソッドから呼び出し側メソッドに戻されます。

注: データ・ハンドラーが直列化データを変換できない場合は、stringToBo によって ParseException 例外がスローされます。

メタオブジェクトで提供されるよりも多くの構成情報をデータ・ハンドラーに提供する必要がある場合は、*config* オプションを指定することができます。この引き数を使用すると、テンプレート・ファイルまたはストリングを URL に指定して、ビジネス・オブジェクトから XML 文書を作成する際に使用するスキーマを取得できます。

参照項目

boToString(), byteArrayToBo(), readerToBO(), streamToBO()

traceCWConnectorAPIVersion()

Java コネクター・ライブラリーのバージョンをトレース宛先に書き込みます。

構文

```
public static void traceCWConnectorAPIVersion();
```

パラメーター

なし。

戻り値

なし。

例外

なし。

注記

`traceCWConnectorAPIVersion()` メソッドは、トレース・レベルがレベル 1 以上である場合、Java コネクター・ライブラリーのバージョンをトレースの宛先に送信します。このメソッドは、パッケージ内のマニフェスト・ファイルから Java コネクター・ライブラリーのバージョンを取得します。マニフェスト・ファイルは、製品のバージョン情報が保管されている標準 Java ファイルです。

コネクターのトレース宛先の名前は、`TraceFileName` コネクター構成プロパティーを介して設定してください。

traceWrite()

トレース宛先にトレース・メッセージを書き込みます。

構文

```
public final static void traceWrite(int traceLevel, String msg);
```

パラメーター

traceLevel 出力対象トレース・メッセージの判別に用いられるトレース・レベルを示すトレース・レベル定数。以下のいずれかです。

```
CWConnectorUtil.LEVEL1
CWConnectorUtil.LEVEL2
```

CWConnectorUtil.LEVEL3
CWConnectorUtil.LEVEL4
CWConnectorUtil.LEVEL5

現在のトレース・レベルが *traceLevel* より大きいか等しい場合、メソッドはトレース・メッセージを書き込みます。

注: トレース・メッセージには、トレース・レベル 0 (LEVEL0) を指定しないでください。トレース・レベル 0 は、トレースがオフになっていることを示します。したがって、LEVEL0 の *traceLevel* に関連するトレース・メッセージが、印刷されることは絶対にありません。

msg トレース・メッセージに使用されるメッセージ・テキスト。

戻り値

なし。

例外

なし。

注記

traceWrite() メソッドは、コネクター用のユーザー独自のトレース・メッセージの書き込みに使用します。コネクターに対するトレースがオンになるのは、TraceLevel コネクター構成プロパティーがゼロ以外の値 (LEVEL0 以外の トレース・レベル定数) に設定されている場合です。

現在のトレース・レベルが *traceLevel* より大きいか等しい場合、traceWrite() メソッドは、指定された *msg* テキストをトレース宛先に送信します。コネクターのトレースの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「トレース」セクションを通じて設定します。

通常、トレース・メッセージはデバッグ時にのみ必要となるため、トレース・メッセージをメッセージ・ファイルに含めるかどうかは、以下のように開発者が任意に決定することができます。

- 英語圏外のユーザーがトレース・メッセージを参照する必要がある場合、それらのメッセージは国際化対応にする必要があります。そのため、トレース・メッセージをメッセージ・ファイルに入れて、generateMsg() メソッドで抽出する必要があります。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。generateMsg() メソッドは、traceWrite() 用のメッセージ・ストリングを生成します。このメソッドによってメッセージ・ファイルから事前定義済みトレース・メッセージが取得され、テキストの書式が設定された後、生成されたメッセージ・ストリングが戻されます。

  注: generateAndTraceMsg() メソッドを使用して、メッセージ生成とロギングを 1 つのステップにまとめて実行できます。
- 英語圏のユーザーのみがトレース・メッセージを参照する必要がある場合、それらのメッセージを国際化対応にする必要はありません。そのため、(英語の) トレ

ース・メッセージを直接 traceWrite() の呼び出しに組み込むことができます。generateMsg() または generateAndTraceMsg() メソッドを使用する必要はありません。

traceWrite() によって記録されたコネクター・メッセージは、LogViewer を使用して見ることはできません。

参照項目

generateAndTraceMsg(),, generateMsg()

使用すべきでないメソッド

CWConnectorUtil クラスの一部のメソッドは、初期のバージョンにおいてサポートされたものであり、現在はサポートされていません。これらの*使用すべきでないメソッド* は、エラーを発生させることはありませんが、IBM では、それらの使用を避けて、既存のコードを新規メソッドに移行することを推奨しています。使用すべきでないメソッドは、将来のリリースでは削除される可能性があります。

表 139 に、CWConnectorUtil クラスの使用すべきでないメソッドのリストを示します。(既存コネクターの変更ではなく) 新規コネクターをコーディングする場合は、このセクションを無視してください。

表 139. CWConnectorUtil クラスの使用すべきでないメソッド

使用すべきでないメソッド	置換
generateAndLogMsg() 第 4 引き数は、*msgParameters* リスト内の引き数の個数のカウント (*argCount*) でした。	generateAndLogMsg() *argCount* の値は、このメソッドに必要ではなくなり、省略可能になりました。*msgParameters* リスト内の引き数の個数を、メソッド自体が判別できます。
generateAndTraceMsg() 第 5 引き数は、*msgParameters* リスト内の引き数の個数のカウント (*argCount*) でした。	generateAndTraceMsg() *argCount* の値は、このメソッドに必要ではなくなり、省略可能になりました。*msgParameters* リスト内の引き数の個数を、メソッド自体が判別できます。
generateAndTraceMsg() 第 3 引き数は、生成対象メッセージのメッセージ・タイプ (*msgType*) でした。	generateAndTraceMsg() *msgType* の値は、このメソッドに必要ではなくなり、省略可能になりました。トレース・メッセージ用のメッセージ・タイプは、常に XRD_TRACE であるため、メソッド自体がそのメッセージ・タイプを埋めることができます。

第 23 章 CWCustomBOHandlerInterface インターフェース

CWCustomBOHandlerInterface インターフェースは、カスタム・ビジネス・オブジェクト・ハンドラーの振る舞いを定義します。このクラスは、1 つのビジネス・オブジェクト・ハンドラーを実装し、これにアクセスするためのコードを提供します。通常ユーザーは、ビジネス・オブジェクト・ハンドラーの基底クラス (CWConnectorBOHandler) を拡張し、ビジネス・オブジェクト・ハンドラーの機能を定義する doVerbFor() メソッドを実装することにより、ビジネス・オブジェクト・ハンドラーを提供します。この方法に関しては、ビジネス・オブジェクトの各動詞を処理する標準的な機構があります。ただし、この機構では、一部のビジネス・オブジェクトにおいて、特定の動詞の振る舞いをカスタマイズする機能がサポートされません。

カスタマイズした振る舞いをビジネス・オブジェクト・ハンドラーに提供する必要がある場合は、カスタム・ビジネス・オブジェクト・ハンドラー・クラスを作成し、CWCustomBOHandlerInterface インターフェースを実装することにより、カスタム・ビジネス・オブジェクト・ハンドラーを作成します。このカスタム・ビジネス・オブジェクト・ハンドラーの機能は、doVerbForCustom() メソッドによって定義されます。

要求処理とビジネス・オブジェクト・ハンドラーの概要については、27 ページの『要求処理』を参照してください。ビジネス・オブジェクト・ハンドラーの実装方法については、91 ページの『第 4 章 要求処理』を参照してください。

表 140 に、CWCustomBOHandlerInterface インターフェースのメソッドを要約します。

表 140. CWCustomBOHandlerInterface インターフェースのメンバー・メソッド

メンバー・メソッド	説明	ページ
doVerbForCustom()	ビジネス・オブジェクトのアクティブな動詞のために、動詞処理を実行します。	437

doVerbForCustom()

ビジネス・オブジェクトのアクティブな動詞に対して、カスタム動詞処理を実行します。

構文

```
public int doVerbForCustom(CWConnectorBusObj theBusObj);
```

パラメーター

theBusObj 処理対象のアクティブ動詞が所属するビジネス・オブジェクトです。

戻り値

動詞操作の結果状況を示す整数です。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

CWConnectorConstant.SUCCEED
: 動詞操作は成功しました。

CWConnectorConstant.FAIL
: 動詞操作は失敗しました。

CWConnectorConstant.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

CWConnectorConstant.VALCHANGE
: ビジネス・オブジェクトの 1 つ以上の値が変更されました。

CWConnectorConstant.VALDUPES
: 要求された操作の実行中に、同じキー値を持つ複数のレコードがアプリケーション・データベース内に見つかりました。

CWConnectorConstant.MULTIPLE_HITS
: コネクターが非キー値を使用して取得中に複数の一致レコードを検出しました。コネクターは最初の一致レコード用のビジネス・オブジェクトのみを戻します。

CWConnectorConstant.RETRIEVEBYCONTENT_FAILED
: コネクターは、非キー値による取得で、一致レコードを検出できませんでした。

CWConnectorConstant.BO_DOES_NOT_EXIST
: コネクターは取得動作を実行しましたが、要求されたビジネス・オブジェクトに対応する一致エンティティーがアプリケーション・データベースにありません。

例外

ConnectionFailureException
: コネクターがアプリケーションから切断されるとスローされます。

VerbProcessingFailedException
: 動詞処理が失敗した場合にスローされます。

注記

このビジネス・オブジェクトの動詞のアプリケーション固有の情報に CBOH タグが含まれている場合、doVerbForCustom() メソッドは、*theBusObj* ビジネス・オブジェクト内のアクティブな動詞のアクションを実行します。このタグは、CWCustomBOHandlerInterface インターフェースの実装のために、(パッケージ名を含む) 完全なクラス名を指定します。このタグの形式については、202 ページの『動詞のアプリケーション固有の情報の追加』を参照してください。

ビジネス・オブジェクト・ハンドラーを起動する場合、コネクター・フレームワークによって実際に起動されるのは、(BOHandlerBase クラスから継承した) 下位の doVerbFor() メソッドです。下位の doVerbFor() メソッドは、呼び出すビジネス・オブジェクト・ハンドラーを次の方法で決定します。

- ビジネス・オブジェクトの動詞がそのアプリケーション固有の情報内に CBOH タグを持っている場合は、この doVerbForCustom() メソッドを呼び出します。
- それ以外の場合は doVerbFor() メソッドを呼び出します (このメソッドは、コネクターの開発者がビジネス・オブジェクト・ハンドラーの CWConnectorBOHandler クラスの一部として実装する必要があります)。

詳細については、195 ページの『戻り状況記述子の取り込み』を参照してください。

doVerbForCustom() メソッドが、その例外の 1 つをスローしなければならない場合に最初に行う必要のある作業は、例外の情報を包含する例外詳細オブジェクトへのデータの取り込みです。詳細については、 290 ページの表 121 を参照してください。このメソッドの実装方法については、 201 ページの『doVerbForCustom() メソッドの実装』を参照してください。

参照項目

doVerbFor()

第 24 章 CWException クラス

CWException クラスは、Java コネクター・ライブラリー内の例外に対する基底クラスです。 Java コネクター・ライブラリーは、次の名前を持つそれ自身の例外クラスを作成するため、Java Exception クラスを延長します。

```
com.crossworlds.cwconnectorapi.exceptions.CWException
```

このクラスは、Java コネクター・ライブラリーのメソッドがスローできる*例外オブジェクト* を表します。

注: Java コネクター・ライブラリーのメソッドの参照説明では、ほとんどの場合、『例外』セクションで、メソッドがスローする例外のリストが示されます。

CWException クラスの内容は以下のとおりです。

- 『メソッド』
- 444 ページの『例外サブクラス』

メソッド

表 141 に、CWException クラスのメソッドを要約します。

表 141. CWException クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWException()	例外オブジェクトを作成します。	441
getExceptionObject()	例外オブジェクトから例外詳細オブジェクトを取得します。	442
getMessage()	例外オブジェクトからメッセージを取得します。	442
getStatus()	例外オブジェクトに関連する状況を取得します。	443
setStatus()	例外オブジェクトに関連する状況を設定します。	443

CWException()

例外オブジェクトを作成します。

構文

```
public CWException();
public CWException(CWExceptionObject excptionDetail);
```

パラメーター

excptionDetail　追加の例外情報を含む例外詳細オブジェクトです。

戻り値

新しい CWException オブジェクト。

注記

CWException() コンストラクターには、次の 2 つの形式があります。

- 1 番目の形式は、空の CWException オブジェクトを作成します。
- 2 番目の形式は、新規 CWException オブジェクトを初期化するため、例外詳細オブジェクトを渡します。

---

getExceptionObject()

例外オブジェクトから例外詳細オブジェクトを取得します。

構文

```
public CWConnectorExceptionObject getExceptionObject();
```

パラメーター

なし。

戻り値

追加例外情報を含む CWConnectorExceptionObject オブジェクトです。

例外

なし。

注記

getExceptionObject()メソッドは、例外オブジェクトから、CWConnectorExceptionObjectの形式で例外詳細情報を取得します。メッセージ・テキスト、メッセージ番号、およびメッセージ説明などの例外情報を取得する際には、CWConnectorExceptionObject クラスのメソッドを使用できます。

参照項目

『第 19 章 CWConnectorExceptionObject クラス』

---

getMessage()

例外オブジェクトからメッセージを取得します。

構文

```
public String getMessage();
```

パラメーター

なし。

戻り値

例外に関連するメッセージを含む String オブジェクトです。

例外

なし。

getStatus()

例外オブジェクトに関連する状況を取得します。

構文

```
public int getStatus();
```

パラメーター

なし。

戻り値

例外オブジェクトの例外状況整数です。

例外

なし。

注記

getStatus() メソッドは、コネクターによって設定されている状況整数値を取得します。この状況整数値は、通常、CWConnectorConstant クラス (FAIL または APPRESPONSETIMEOUT など) によって表される結果状況定数の 1 つです。

参照項目

setStatus()

setStatus()

例外オブジェクトに関連する状況を設定します。

構文

```
public void setStatus(int status);
```

パラメーター

status 例外オブジェクトに割り当てられている状況整数値です。

戻り値

なし。

例外

なし。

注記

setStatus() メソッドは、CWException オブジェクトの一部である状況値を設定します。この状況値には、通常、CWConnectorConstant クラス (FAIL または APPRESPONSETIMEOUT など) によって表される結果状況定数の 1 つが、コネクターによってセットされます。

参照項目

getStatus()

例外サブクラス

CWException クラス内には、Java コネクター・ライブラリーのメソッドで使用可能な、特定の例外を表すサブクラスが存在します。表 142 に、例外サブクラスのリストを示します。

表 142. CWConnectorException サブクラス

例外サブクラス	定義
ArchiveFailedException	イベント・レコードがアーカイブ・ストアに保管できなかった場合に、イベント・ストア・クラスの archiveEvent() メソッドからスローされます。
AttributeNotFoundException	属性の指定された位置または名前が、既存のビジネス・オブジェクト内におけるその属性の位置または名前と一致しなかった場合にスローされます。
AttributeNullValueException	属性値に対してある操作を実行する必要のある場合に、その属性値が null のときにスローされます。
AttributeValueException	NumberFormatException 例外が存在する場合にスローされます。
ConnectionFailureException	コネクターが、アプリケーションとの接続を確立できなかった場合にスローされます。
DataHandlerCreateException	データ・ハンドラー・メソッドが、指定された MIME タイプのデータ・ハンドラーをインスタンス化できない場合、スローされます。
DefaultSettingFailedException	デフォルト値の設定が失敗した場合にスローされます。
DeleteFailedException	イベント・レコードがイベント・ストアから削除できなかった場合に、イベント・ストア・クラスの deleteEvent() メソッドからスローされます。
InProgressEventRecoveryFailedException	進行中イベントのリカバリーが失敗した場合にスローされます。
InvalidAttributePropertyException	属性に対する無効なプロパティーの照会 (例えば、オブジェクトである属性に対して、getMaxLength() 呼び出しなど) があった場合にスローされます。
InvalidStatusChangeException	イベント状況における要求された変更が有効でない場合にスローされます。
InvalidVerbException	指定された動詞が、ビジネス・オブジェクトによってサポートされていない場合にスローされます。
LogonFailedException	コネクターが、指定されたユーザー名とパスワードでアプリケーションにログオンできない場合にスローされます。

表 142. *CWConnectorException サブクラス (続き)*

例外サブクラス	定義
NotSupportedException	なんらかの機能が現行バージョンの製品でサポートされない場合にスローされます。
ParseException	(コネクターから呼び出された) データ・ハンドラーがビジネス・オブジェクトと指定された MIME タイプとの間での変換を行っているときに、何らかのエラーが発生した場合、スローされます。
PropertyNotSetException	必須のコネクター構成プロパティーが設定されていない場合にスローされます。
SpecNameNotFoundException	ビジネス・オブジェクトを作成するためのビジネス・オブジェクト定義が発見できない場合にスローされます。
StatusChangeFailedException	コネクターが、アプリケーションのイベント・ストアにイベントの状況を設定できない場合にスローされます。
VerbProcessingFailedException	動詞により指定された操作が失敗した場合に、doVerbFor() メソッドからスローされます。
WrongASIFormatException	アプリケーション固有の情報が、*name=value* の形式でない場合にスローされます。
WrongAttributeException	指定された属性のデータ型が、属性が保持するように定義されたデータ型と一致しなかった場合にスローされます。

表 143. *例外を戻すメソッド*

Java コネクター・ライブラリー例外	例外を戻すメソッド
SpecNameNotFoundException	CWConnectorUtil createBusObj() CWConnectorBusObj setBusObjValue()
AttributeNotFoundException	CWConnectorBusObj getAttrIndex() getbooleanValue() getBusObjValue() getCardinality() getDefault() getDefaultboolean() getDefaultdouble() getDefaultfloat() getDefaultint() getDefaultlong() getDefaultString() getdoubleValue() getfloatValue() getintValue() getlongValue() getMaxLength() getObjectCount() getStringValue() getTypeName() getTypeNum() hasCardinality()

表 143. 例外を戻すメソッド (続き)

Java コネクター・ライブラリー例外	例外を戻すメソッド
	hasName()
	hasType()
	isForeignKeyAttr()
	isKeyAttr()
	isMultipleCard()
	isObjectType()
	isRequiredAttr()
	isType()
	removeAllObjects()
	removeBusinessObjectAt()
	setbooleanValue()
	setBusObjValue()
	setdoubleValue()
	setfloatValue()
	setintValue()
	setStringValue()
WrongAttributeException	CWConnectorBusObj
	getbooleanValue()
	getBusObjValue()
	getDefaultboolean()
	getDefaultdouble()
	getDefaultfloat()
	getDefaultint()
	getDefaultlong()
	getDefaultString()
	getdoubleValue()
	getfloatValue()
	getintValue()
	getlongValue()
	getStringValue()
	setbooleanValue()
	setBusObjValue()
	setdoubleValue()
	setfloatValue()
	setintValue()
	setStringValue()
AttributeNullValueException	CWConnectorBusObj
	getbooleanValue()
	getDefaultboolean()
	getDefaultdouble()
	getDefaultfloat()
	getDefaultint()
	getDefaultlong()
	getdoubleValue()
	getfloatValue()
	getintValue()
	getlongValue()
	setBusObjValue()
AttributeValueException	CWConnectorBusObj
	getDefaultdouble()
	getDefaultfloat()
	getDefaultint()
	getDefaultlong()

表 143. 例外を戻すメソッド (続き)

Java コネクター・ライブラリー例外	例外を戻すメソッド
	getdoubleValue() getfloatValue() getintValue() getlongValue() setbooleanValue() setBusObjValue() setdoubleValue() setfloatValue() setintValue() setStringValue()
InvalidAttributePropertyException	CWConnectorBusObj getMaxLength()
InvalidVerbException	CWConnectorBusObj setVerb()

例外サブクラス・コンストラクター

例外サブクラスを作成します。

構文

public *exception_subclass*(CWConnectorExceptionObject *excption*)

ここで、exception_subclassは、(表 142 に示されているような) 例外サブクラスの名前です。

パラメーター

excption 例外に関する情報を含む例外オブジェクトです。

戻り値

CWException クラスのサブクラスを表すオブジェクトです。

注記

例外に関する情報を取得するには、CWConnectorExceptionObject クラスのメソッドを使用します。

第 25 章 CWProperty クラス

CWProperty クラスは、Java コネクターの階層コネクター構成プロパティーを表します。階層コネクター構成プロパティーは 1 つ以上の値を取ることができ、また、これらの値はストリング値または他の (子) コネクター・プロパティーにすることができます。

注: 下位の Java コネクター・ライブラリーの CWProperty クラスは、CxProperty クラスによって拡張されます。下位の Java コネクター・ライブラリーのクラスの詳細については、465 ページの『第 26 章 下位の Java コネクター・ライブラリーの概要』を参照してください。

CWProperty クラス内のメソッドについては、表 144 に要約します。

表 144. CWProperty クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CWProperty()	コネクター・プロパティー・オブジェクトを作成します。	450
getCardinality()	コネクター構成プロパティーのカーディナリティー (単一値または多値) を取得します。	450
getChildPropValue()	指定された子プロパティーに対するすべてのストリング値を取得します。	451
getChildPropsWithPrefix()	指定のプレフィックスに一致する名前を持つ階層コネクター構成プロパティーからすべての子プロパティーを取得します。	452
getEncryptionFlag()	コネクター構成プロパティーの暗号化フラグを取得します。	453
getHierChildProp()	階層コネクター構成プロパティーから*指定された* 子プロパティーを取得します。	454
getHierChildProps()	階層コネクター構成プロパティーからすべての 子プロパティーを取得します。	455
getHierProp()	プロパティー階層の任意のレベルで、階層コネクター構成プロパティーの*指定された* 子プロパティーを取得します。	456
getName()	コネクター構成プロパティーの名前を取得します。	457
getPropType()	コネクター構成プロパティーのプロパティー・タイプを取得します (シンプルまたは階層)。	457
getStringValues()	階層コネクター構成プロパティーからすべてのストリング値を取得します。	458
hasChildren()	コネクター構成プロパティーが子プロパティーを持っているかどうかを判別します。	458
hasValue()	コネクター構成プロパティーがストリング値を持っているかどうかを判別します。	459
setEncryptionFlag()	階層コネクター構成プロパティーの暗号化フラグを設定します。	460
setValues()	階層コネクター構成プロパティーの値を設定します。	461

CWProperty()

階層コネクター・プロパティー・オブジェクトを作成します。

構文

```
public CWProperty();
public CWProperty(String propName, String simplePropValue);
public CWProperty(String propName, CWProperty[] hierPropValues);
```

パラメーター

propName コネクター構成プロパティーの名前を指定します。

simplePropValue
シンプル・コネクター・プロパティーの初期化に使用するストリング値です。

hierPropValues 階層コネクター・プロパティーの初期化に使用するコネクター・プロパティー (CWProperty) オブジェクトの配列です。

戻り値

新規に作成された階層コネクター・プロパティーを含む CWProperty オブジェクト。

注記

CWProperty() コンストラクターの形式には、以下の種類があります。

- 最初の形式は、空のコネクター・プロパティー・オブジェクトを作成します。CWProperty クラスの他のメソッドを使用してこのオブジェクトを取り込むことができます。
- 2 番目の形式は、シンプル・コネクター・プロパティーのコネクター・プロパティー・オブジェクトを、指定したプロパティー名およびストリング値を使用して作成します。
- 3 番目の形式は、階層コネクター・プロパティーのコネクター・プロパティー・オブジェクトと、指定したプロパティー名および階層プロパティーの配列を作成します。

getCardinality()

コネクター構成プロパティーのカーディナリティーを取得します。

構文

```
public int getCardinality();
```

パラメーター

なし。

戻り値

コネクター構成プロパティーのカーディナリティーを示す整数です。この整数値を以下に示すコネクター・プロパティー定数と比較することにより、カーディナリティーを判別します。

`CWConnectorConstant.SINGLE_VALUED`
: コネクター構成プロパティーは、単一のカーディナリティーを持っています。つまり、このプロパティーが保持する値は 1 つのみです。

`CWConnectorConstant.MULTI_VALUED`
: コネクター構成プロパティーは、複数のカーディナリティーを持っています。つまり、このプロパティーは複数の値が含まれています。

例外

なし。

注記

`getCardinality()` メソッドは、コネクター構成プロパティーのカーディナリティーを取得します。カーディナリティーは、プロパティーに含まれる値の数が単数か複数かを示します。このメソッドを使用して、プロパティー値の取得方法を判別します。

---

getChildPropValue()

階層コネクター・プロパティーの指定された子プロパティーからストリング値を取得します。

構文

```
public String[] getChildPropValue(String propName);
```

パラメーター

propName
: ストリング値の取得対象となるコネクター構成プロパティーの名前を指定します。

戻り値

`String` オブジェクトの配列への参照。各配列は、指定された子プロパティーのストリング値を 1 つ表します。指定された子プロパティーが現在の階層コネクター・プロパティーに存在しない場合、メソッドは `null` を戻します。

例外

なし。

注記

getChildPropValue() は指定された子プロパティーに対するストリング値を取得します。getChildPropValue() を呼び出す前に、hasValue() メソッドを使用して、階層コネクター・プロパティーがストリング値を持っているかどうかを確認できます。階層コネクター・プロパティーのすべてのストリング値を取得する場合は、getStringValues() メソッドを使用します。

階層コネクター・プロパティーが暗号化されたストリング値を持っている場合 (暗号化フラグを true に設定)、getChildPropValue() は暗号化されていない値を戻します。復号処理の実行は必要ありません。

参照項目

getStringValues(),, hasValue()

---

getChildPropsWithPrefix()

指定のプレフィックスに一致する名前を持つ階層コネクター構成プロパティーのすべての 子プロパティーを取得します。

構文

```
public CWProperty[] getChildPropsWithPrefix(String propPrefix);
```

パラメーター

propPrefix 階層コネクター構成プロパティーの子プロパティーの取得時にマッチングするプレフィックスを指定します。

戻り値

CWProperty オブジェクトの配列への参照。各配列は、指定された *propPrefix* で始まる名前を持つ、階層コネクター・プロパティーのコネクター・プロパティーを 1 つ表します。階層コネクター・プロパティーに指定されたプレフィックスを持つ子プロパティーがない場合は、メソッドは null を戻します。

例外

なし。

注記

getChildPropsWithPrefix() メソッドは、指定された *propPrefix* で始まる名前を持つ、階層コネクター構成プロパティーの子プロパティーをすべて 取得します。取得されるプロパティーは、現在の階層プロパティーの子のプロパティーのみです。孫やひ孫などのプロパティーは含みません。階層内の下位の子プロパティーを取得するには、まず最初に、特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getHierChildProps() や getHierChildProp() などのメソッドを使用してその子を取得します。

注: getHierProp() を使用して、指定された子や孫など、プロパティー階層の下位のプロパティーを取得できます。

例えば、図 77 に示す次のプロパティー階層を使用して、複数リスナー用のプロパティーを構成するとします。

```
ProtocolListener
    SingleValProp1=dexter
    Listener1=first listener
        Port=1500
    Listener2=second listener
        Port=1502
    SingleValProp2=tashi
```

図 77. プロトコル・リスナーのプロパティー階層のサンプル

「Listener」のプレフィックスを持つすべてのプロパティーを取得するには、まず ProtocolListener のトップレベルのコネクター・オブジェクトを取得します (例えば、topLevelProp に取り込みます)。次に、以下の呼び出しを使用して ProtocolListener の Listener1 と Listener2 の両方の子プロパティーを取得します。

```
CWProperty[] listenerProps = topLevelProp.getChildPropsWithPrefix("Listener");
```

getChildPropsWithPrefix() を呼び出す前に、hasChildren() メソッドを使用して、階層コネクター・プロパティーが子プロパティーを持っているかどうかを確認できます。指定された子プロパティーを取得するには、getHierChildProp() メソッドを使用します。すべての子プロパティーを取得するには、プレフィックスに関係なく getHierChildProps() メソッドを使用できます。

参照項目

getHierChildProp(),, getHierChildProps(),, getHierProp(),, hasChildren()

---

getEncryptionFlag()

階層コネクター構成プロパティーの暗号化フラグをコネクター・プロパティー・オブジェクトから取得します。

構文

```
public Boolean getEncryptionFlag();
```

パラメーター

なし。

戻り値

現在のコネクター構成プロパティー値が暗号化されているかどうかを示す boolean 値。

例外

なし。

注記

getEncryptionFlag() メソッドは、ブール値の暗号化フラグをコネクター・プロパティー・オブジェクトから取得します。このフラグは、コネクター・プロパティーのストリング値が暗号化されているかどうかを示します。

注: Connector Configurator では、暗号化された値はアスタリスク (*) 文字のストリングとして表示されます。

参照項目

setEncryptionFlag()

---

getHierChildProp()

階層コネクター構成プロパティーの指定された子プロパティーを取得します。

構文

```
public CWProperty getHierChildProp(String propName);
```

パラメーター

propName 取得するコネクター構成プロパティーの名前を指定します。

戻り値

取得対象の子プロパティーを含む CWProperty オブジェクト。指定されたプロパティーが現在の階層コネクター・プロパティーに存在しない場合、メソッドは null を戻します。

例外

なし。

注記

getHierChildProp() メソッドは、*propName* と一致する名前を持つ子プロパティーを階層コネクター構成プロパティーから取得します。取得されるプロパティーは、現在の階層プロパティーの子として存在している必要があり、孫やひ孫などのプロパティーであってはなりません。階層内の下位の子プロパティーを取得するには、まず最初に、特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getHierChildProps() や getHierChildProp() などのメソッドを使用してその子を取得します。

注: getHierProp() を使用して、指定された子や孫など、プロパティー階層の下位のプロパティーを取得できます。

getHierChildProp() を呼び出す前に、hasChildren() メソッドを使用して、階層コネクター・プロパティーが子プロパティーを持っているかどうかを確認できます。すべての 子プロパティーを取得するには、getHierChildProps() メソッドを使用します。指定されたプレフィックスを持つすべての子プロパティーを取得するには、getChildPropsWithPrefix() メソッドを使用できます。

参照項目

getChildPropsWithPrefix(),, getHierChildProps(),, getHierProp(),, hasChildren(),, setValues()

---

getHierChildProps()

階層コネクター構成プロパティーのすべての 子プロパティーを取得します。

構文

```
public CWProperty[] getHierChildProps();
```

パラメーター

なし。

戻り値

CWProperty オブジェクトの配列への参照。各配列は、階層コネクター・プロパティーのコネクター・プロパティーを 1 つ表します。階層コネクター・プロパティーに子プロパティーが含まれていない場合、メソッドは null を戻します。

例外

なし。

注記

getHierChildProps() メソッドは、階層コネクター構成プロパティーのすべての 子プロパティーを取得します。取得されるプロパティーは、現在の階層プロパティーの子のプロパティーのみです。孫やひ孫などのプロパティーは含みません。階層内の下位の子プロパティーを取得するには、まず最初に、特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getHierChildProps() や getHierChildProp() などのメソッドを使用してその子を取得します。

注: getHierProp() を使用して、指定された子や孫など、プロパティー階層の下位のプロパティーを取得できます。

getHierChildProps() を呼び出す前に、hasChildren() メソッドを使用して、階層コネクター・プロパティーが子プロパティーを持っているかどうかを確認できます。指定された子プロパティーを取得するには、getHierChildProp() メソッドを使用します。指定されたプレフィックスを持つすべての子プロパティーを取得するには、getChildPropsWithPrefix() メソッドを使用できます。すべてのストリング値を取得するには、getStringValues() メソッドを使用します。

参照項目

getChildPropsWithPrefix(),, getHierChildProp(),, getHierProp(),, getStringValues(),, hasChildren(),, setValues()

---

getHierProp()

任意のレベルの プロパティー階層で、階層コネクター構成プロパティーの指定された子プロパティーを取得します。

構文

```
public CWProperty getHierProp(String propName);
```

パラメーター

propName 取得するコネクター構成プロパティーの名前を指定します。

戻り値

階層から取得されるプロパティーを含む CWProperty オブジェクト。指定されたプロパティーが現在の階層コネクター・プロパティーに存在しない場合、メソッドは null を戻します。

例外

なし。

注記

getHierProp() メソッドは、*propName* と一致する名前を持つ子プロパティーを階層コネクター構成プロパティーから取得します。現在のプロパティー階層の*任意のレベルで* 子プロパティーを取得できるため、孫やひ孫などのプロパティーを指定できます。取得される子プロパティーの *propName* の形式は以下のとおりです。

child/grandchild/great-grandchild/....

例えば、 453 ページの図 77 に示すプロパティー階層があるとします。Listener1 のポート名を取得するには、まず ProtocolListener のトップレベルのコネクター・オブジェクトを取得します (例えば、topLevelProp に取り込みます)。次に、以下の呼び出しを使用して Listener1 のポート名を取得します。

```
CWProperty listenerPort = topLevelProp.getHierProp("Listener1/Port");
```

getHierProp() を呼び出す前に、hasChildren() メソッドを使用して、階層コネクター・プロパティーが子プロパティーを持っているかどうかを確認できます。プロパティー階層の最上位で指定された子プロパティーを取得するには、getHierChildProp() メソッドを使用します。階層の最上位ですべての 子プロパティーを取得するには、getHierChildProps() メソッドを使用できます。

参照項目

getHierChildProp(),, getHierChildProps(),, hasChildren()

getName()

階層コネクター構成プロパティーの名前をコネクター・プロパティー・オブジェクトから取得します。

構文

```
public String getName();
```

パラメーター

なし。

戻り値

コネクター構成プロパティーの名前を含むストリングです。

例外

なし。

getPropType()

コネクター・プロパティー・オブジェクトからプロパティー・タイプを取得します。

構文

```
public int getPropType();
```

パラメーター

なし。

戻り値

コネクター構成プロパティーのプロパティー・タイプを示す整数です。この整数値を以下に示すコネクター・プロパティー定数と比較することにより、タイプを判別します。

`CWConnectorConstant.SIMPLE`
: コネクター構成プロパティーは、シンプル・タイプです。すなわち、プロパティーに含まれるのはストリング値のみ です。

`CWConnectorConstant.HIERARCHICAL`
: コネクター構成プロパティーは、*階層* タイプです。すなわち、プロパティーには 1 つ以上の子プロパティーとおそらくストリング値が含まれています。

例外

なし。

getStringValues()

階層コネクター構成プロパティーのすべての ストリング値を取得します。

構文

```
public String[] getStringValues();
```

パラメーター

なし。

戻り値

String オブジェクトの配列への参照。各配列は、階層コネクター・プロパティーのストリング値を 1 つ表します。階層コネクター・プロパティーにストリング値が含まれていない場合、メソッドは null を戻します。

例外

なし。

注記

getStringValues() メソッドは、階層コネクター構成プロパティーのすべての ストリング値を取得します。取得されるストリング値は、現在の階層プロパティーのストリング値のみです。子プロパティーの値は含まれません。階層の低位のストリング値を取得するには、以下のいずれかを実行します。

- getChildPropValue() メソッドを使用して、指定された子プロパティーのストリング値を取得します。
- 特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getStringValues() などのメソッドを使用してストリング値を取得します。

getStringValues() を呼び出す前に、hasValue() メソッドを使用して、階層コネクター・プロパティーがストリング値を持っているかどうかを判別できます。子プロパティーを取得するには、getHierChildProp() または getHierChildProps() メソッドを使用します。

階層コネクター・プロパティーが暗号化されたストリング値を持っている場合 (暗号化フラグを true に設定)、getStringValues() は暗号化されていない値を戻します。復号処理の実行は必要ありません。

参照項目

getChildPropValue(),, getHierChildProp(),, getHierChildProps(),, hasValue(),, setValues()

hasChildren()

現在のコネクター・プロパティーが子プロパティーを持っているかどうかを判別します。

構文

```
public boolean hasChildren();
```

パラメーター

なし。

戻り値

階層コネクター・プロパティーが子プロパティーを持っているかどうかを示す boolean です。子プロパティーを持っている場合、メソッドは true を戻します。それ以外の場合は、false を戻します。

例外

なし。

注記

hasChildren() メソッドは、次に示すように、階層コネクター・プロパティーの値を抽出するためにどの CWProperty メソッドを使用するかを判別する際に役立ちます。

- hasChildren() が true を戻す場合、次のいずれかの値のメソッドを使用して、子プロパティーを取得します。

すべての子プロパティーの取得	getHierChildProps()
指定された子プロパティーの取得	getHierChildProp()
指定されたプレフィックスで始まる名前を持つすべての子プロパティーの取得	getChildPropsWithPrefix()

- hasChildren() が false を戻す場合、次のいずれかの値のメソッドを使用して、ストリング値を取得します。

すべてのストリング値の取得	getStringValues()
指定された子プロパティーのストリング値の取得	getChildPropValue()

参照項目

getChildPropValue(),, getHierChildProp(),, getHierChildProps(),, getStringValues(),, hasValue()

---

hasValue()

現在の階層コネクター・プロパティーがストリング値を持っているかどうかを判別します。

構文

```
public boolean hasValue();
```

パラメーター

なし。

戻り値

コネクター・プロパティーにストリング値が含まれているかどうかを示す boolean です。値が含まれている場合、メソッドは true を戻します。それ以外の場合は、false を戻します。

例外

なし。

注記

hasValue() メソッドは、次に示すように、階層コネクター・プロパティーの値を抽出するためにどの CWProperty メソッドを使用するかを判別する際に役立ちます。

- hasValue() が true を戻す場合、次のいずれかの値のメソッドを使用して、ストリング値を取得します。

すべてのストリング値の取得	getStringValues()
指定された子プロパティーのストリング値の取得	getChildPropValue()

- hasValue() が false を戻す場合、次のいずれかの値のメソッドを使用して、子プロパティーを取得します。

すべての子プロパティーの取得	getHierChildProps()
指定された子プロパティーの取得	getHierChildProp()
指定されたプレフィックスで始まる名前を持つすべての子プロパティーの取得	getChildPropsWithPrefix()

参照項目

hasChildren(),

---

setEncryptionFlag()

コネクター構成プロパティーの暗号化フラグを、そのコネクター・プロパティー・オブジェクトに設定します。

構文

```
public void setEncryptionFlag(boolean encryptFlag);
```

パラメーター

encryptFlag 現在のコネクター構成プロパティー値を暗号化すべきかどうかを示すブール値です。

戻り値

なし。

例外

なし。

注記

setEncryptionFlag() メソッドは、コネクター・プロパティー・オブジェクトのブール値の暗号化フラグを設定します。このフラグは、コネクター・プロパティーのストリング値が暗号化されているかどうかを示します。

注: Connector Configurator では、暗号化された値はアスタリスク (*) 文字のストリングとして表示されます。

参照項目

getEncryptionFlag()

---

setValues()

階層コネクター構成プロパティーの値を設定します。

構文

```
public void setValues(Object[] propValues);
```

パラメーター

propValues Object 値の配列です。各配列エレメントは、単一のプロパティー値です。

戻り値

なし。

例外

なし。

注記

setValues() メソッドの使用により、階層コネクター構成プロパティーの値を設定できます。Object の配列である *propValues* 配列にプロパティー値を指定します。これにより、この単一の配列にストリング値と子プロパティー値の両方を渡すことができます。*propValues* 配列へのプロパティー値の割り当ては、それらが階層コネクター・プロパティー内で定義されている順に行うようにしてください。

例えば、setValues() への次の呼び出しは、ストリング値と子プロパティーの両方を topLevelProp のコネクター・プロパティーに割り当てます。

```
Object[] propValues;
CWProperty childProp;

propValues[0] = "stringValue"
propValue[1] = childProp;
topLevelProp.setValues(propValues);
```

参照項目

getHierChildProp(),, getHierChildProps(),, getStringValues()

第 4 部 下位の Java コネクター・ライブラリー API 参照

第 26 章 下位の Java コネクター・ライブラリーの概要

下位の Java コネクター・ライブラリーには、上位の Java コネクター・ライブラリーのベースとなる下位のクラス・ライブラリーが組み込まれています。このコネクター・クラス・ライブラリーには、下位 Java コネクター用の定義済みクラスが含まれています。下位の Java コネクター・ライブラリーでは、トレース・サービスやロギング・サービスを実装するメソッドなどのユーティリティーが用意されています。

重要: 下位の Java コネクター・ライブラリーは、Java コネクターの開発には、使用すべきでないライブラリーです。新規 Java コネクターの開発には、Java コネクター・ライブラリーを使用します。Java コネクター・ライブラリーの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

IBM では、下位の Java コネクター・ライブラリーの定義済みクラスとインターフェースを、製品の Java jar (Java アーカイブ・ファイル) の `crossworlds.jar` にインクルードしています。 `crossworlds.jar` サブファイルは、製品ライブラリーの `bin` サブディレクトリーにあります。

クラスおよびインターフェース

表 145 に、下位の Java コネクター・ライブラリーのクラスとインターフェースのリストを示します。

表 145. 下位の Java コネクター・ライブラリーのクラスとインターフェース

クラスまたはインターフェース	説明	ページ
`BOHandlerBase`	ビジネス・オブジェクト・ハンドラーの基底クラスを表します。この基底クラスを拡張して、コネクターに対して 1 つ以上のビジネス・オブジェクト・ハンドラーを定義します。	467
`BusinessObjectInterface`	ビジネス・オブジェクト・インスタンスを表します。属性の名前と値に対するアクセスを可能にします。	471
`ConnectorBase`	コネクターの基底クラスを表します。この基底クラスを拡張して、コネクター・クラスを定義し、必須の仮想メソッドを実装します。	489
`CxObjectContainerInterface`	子ビジネス・オブジェクトから成る配列を管理します。	509
`CxObjectAttr`	属性のプロパティーに関する情報を含む属性記述子を表します。	501
`CxProperty`	階層コネクター構成プロパティーを含むコネクター・プロパティー・オブジェクトを表します	515
`CxStatusConstants`	下位の Java コネクター・ライブラリーと併用する結果状況定数を定義します。	523

表 145. 下位の Java コネクター・ライブラリーのクラスとインターフェース (続き)

クラスまたはインターフェース	説明	ページ
JavaConnectorUtil	Java コネクターで使用する各種のユーティリティー・メソッドを提供します。これらのユーティリティー・メソッドは、次のような一般カテゴリーに分かれます。 • メッセージの生成およびロギングのための静的メソッド • ビジネス・オブジェクトの作成のための静的メソッド • コネクター構成プロパティーの取得のための静的メソッド • ロケール情報の取得のためのメソッド	525
ReturnStatusDescriptor	エラー・メッセージと情報メッセージを含む戻り状況記述子を表します。	543
例外	例外サブクラスは、下位の Java コネクター・ライブラリー・メソッドがスローする例外を表します。	545

第 27 章 BOHandlerBase クラス

BOHandlerBase クラスは、ビジネス・オブジェクト・ハンドラーの基底クラスに対する下位の Java コネクター・ライブラリーのクラスです。これは、AppSide_Connector パッケージの一部です。すべての下位の Java コネクターは、そのビジネス・オブジェクト・ハンドラーごとに、このクラスを拡張して、各派生クラス内に doVerbFor() メソッドを実装する必要があります。

注: CWConnectorBOHandler クラスは、下位の Java コネクター・ライブラリーの BOHandlerBase クラスに対するラッパーである Java コネクター・ライブラリーのメソッドです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、 267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

コネクター・フレームワークは、ConnectorBase.getBOHandlerForBO() を呼び出して、コネクターのサポートするビジネス・オブジェクトごとにビジネス・オブジェクト・ハンドラーを作成します。

表 146 に、BOHandlerBase クラスのメソッドを要約します。

表 146. BOHandlerBase クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
doVerbFor()	ビジネス・オブジェクトのアクティブな動詞の動作を実行します。	467
getName()	ビジネス・オブジェクト・ハンドラーの名前を戻します。	469
setName()	ビジネス・オブジェクト・ハンドラーの名前を設定します。	469

doVerbFor()

ビジネス・オブジェクトのアクティブな動詞の動作を実行します。このメソッドは、ビジネス・オブジェクト・ハンドラーの基本パブリック・インターフェースです。

構文

```
public int doVerbFor(BusinessObjectInterface theBusObj,
    ReturnStatusDescriptor rtnObj);
```

パラメーター

theBusObj 着信ビジネス・オブジェクトです。

rtnObj 操作状況、および統合ブローカーに対するエラー・メッセージまたは情報メッセージを含む状況記述子オブジェクトです。

戻り値

動詞操作の結果状況を示す整数です。

CxStatusConstants.SUCCEED
: 動詞操作は成功しました。

CxStatusConstants.FAIL
: 動詞操作は失敗しました。

CxStatusConstants.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

CxStatusConstants.VALCHANGE
: ビジネス・オブジェクトの 1 つ以上の値が変更されました。

CxStatusConstants.VALDUPES
: 要求された操作が、同一キー値に対して複数のレコードを検出しました。

CxStatusConstants.MULTIPLE_HITS
: コネクターが非キー値を使用して取得中に複数の一致レコードを検出しました。コネクターは、ビジネス・オブジェクト内で最初に一致したレコードのみを戻します。

CxStatusConstants.RETRIEVEBYCONTENT_FAILED
: コネクターは、非キー値による取得で、一致レコードを検出できませんでした。

CxStatusConstants.BO_DOES_NOT_EXIST
: 要求されたビジネス・オブジェクト・エンティティーは、データベースに存在しません。

注記

統合ブローカーからビジネス・オブジェクトが届けられると、コネクター・フレームワークは、状況記述子オブジェクトを作成し、ビジネス・オブジェクトのアクティブな動詞のアクションを実行する doVerbFor() メソッドの呼び出し時に、そのオブジェクトを引き数として送信します。

重要: doVerbFor() メソッドは、ユーザーがビジネス・オブジェクト・ハンドラーに対して、実装する必要 のある抽象メソッドです。

doVerbFor() メソッドは、次のステップを実行する必要があります。

- 動詞の操作を実行します。
- 通知、警告、またはエラーの戻りメッセージがある場合は、ReturnStatusDescriptor.setErrorString() を呼び出して、状況記述子オブジェクトにメッセージを設定します。
- ReturnStatusDescriptor.setStatus() を呼び出して、状況戻りコードを戻します。 setStatus() メソッドは、その値が doVerbFor() メソッドの戻り値と同じである整数を取ります。

参照項目

BusinessObjectInterface クラスの説明も参照してください。

getName()

BOHandler オブジェクトの名前を取得します。

構文

```
protected String getName();
```

パラメーター

なし。

戻り値

BOHandler オブジェクトの名前を含む String です。このメソッドに先立って、BOHandlerBase インスタンスに対して、setName() が呼ばれていない場合は、ヌルを戻します。

参照項目

setName() メソッドも参照してください。

setName()

ビジネス・オブジェクト・ハンドラーである BOHandler オブジェクトの名前を設定します。この名前は、通常、ビジネス・オブジェクト・ハンドラーがその処理のために作成されたビジネス・オブジェクトの名前になります。

構文

```
protected void setName(String name);
```

パラメーター

name BOHandler オブジェクトの名前を指定します。

戻り値

なし。

第 28 章 BusinessObjectInterface インターフェース

下位 Java コネクターの開発者は、BusinessObjectInterface インターフェースを通して、ビジネス・オブジェクトのビューを取得できます。これは、CxCommon パッケージの一部です。このインターフェースでは、ビジネス・オブジェクトのメタデータに関する情報を取得するメソッド、およびビジネス・オブジェクト・インスタンスの読み取りと変更を行うメソッドが定義されています。BusinessObjectInterface の各インスタンスは、単一のビジネス・オブジェクトを表します。

注: CWConnectorBusObj クラスは、下位の Java コネクター・ライブラリーの BusinessObjectInterface インターフェースに機能性を提供する Java コネクター・ライブラリー・クラスです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

重要: このインターフェースの実装は、下位の Java コネクター・ライブラリーによって内部的に提供されます。*このクラスは、コネクター開発者が実装する必要はありません。*

表 147 に、BusinessObjectInterface インターフェースのメソッドを要約します。

表 147. BusinessObjectInterface インターフェースのメンバー・メソッド

メンバー・メソッド	説明	ページ
clone()	既存のビジネス・オブジェクトをコピーします。	472
doVerbFor()	ビジネス・オブジェクト・ハンドラー (BOHandlerBase クラスのインスタンス) を呼び出し、ビジネス・オブジェクトのアクティブな動詞を実行します。	472
dump()	ビジネス・オブジェクト情報のフォーマット設定を行い、ロギングとトレース用に定義された標準フォーマットでその情報を戻します。	474
getAppText()	ビジネス・オブジェクトの AppSpecificInfo フィールドの値を取得します。	474
getAttrCount()	ビジネス・オブジェクトが持っている属性の個数を取得します。	474
getAttrDesc()	名前または位置によって属性記述を取得します。	475
getAttribute()	属性値を取得します。	475
getAttributeIndex()	指定の属性のインデックス位置を取得します。	476
getAttributeType()	属性名または属性の位置を使用して、指定の属性に対する属性型コードを取得します。	476
getAttrName()	位置によって、属性の名前を取得します。	477
getAttrValue()	名前または位置によって属性値を取得します。	478
getBusinessObjectVersion()	ビジネス・オブジェクトのバージョンを取得します。	478
getDefaultAttrValue()	名前または位置によって属性値のデフォルト値を取得します。	479
getLocale()	ビジネス・オブジェクトに関連付けられているロケールを取得します。	479
getName()	ビジネス・オブジェクトが参照するビジネス・オブジェクト仕様の名前を取得します。	480

表 147. *BusinessObjectInterface* インターフェースのメンバー・メソッド *(続き)*

メンバー・メソッド	説明	ページ
getParentBusinessObject()	現在のビジネス・オブジェクトの親ビジネス・オブジェクトを取得します。	480
getVerb()	ビジネス・オブジェクトのアクティブな動詞を取得します。	481
getVerbAppText()	動詞アプリケーション固有の情報を取得します。	481
isBlank()	指定された名前または位置の属性の値が Blank であるかどうかを判別します。	482
isIgnore()	指定された名前または位置の属性の値が「ignore」であるかどうかを判別します。	482
isVerbSupported()	動詞がサポートされているかどうかを判別します。	482
makeNewAttrObject()	指定された名前または位置の属性に対して、正しい型の新規オブジェクトを作成します。この操作は、普通、子オブジェクトを含む属性に対して適用されます。	483
setAttributeWithCreate()	オブジェクトの属性値を設定します。	484
setAttrValue()	名前または位置によって属性の値を設定します。	484
setDefaultAttrValues()	ビジネス・オブジェクトの属性をそのデフォルト値で初期化します。	486
setLocale()	ビジネス・オブジェクトに関連したロケールを設定します。	486
setVerb()	ビジネス・オブジェクトのアクティブな動詞を設定します。	487

clone()

既存のビジネス・オブジェクトをコピーします。その動詞も含めて、ビジネス・オブジェクトの属性をコピーします。

構文

```
public Object clone();
```

パラメーター

なし。

戻り値

その属性と動詞も含め、現在のビジネス・オブジェクトの 1 つのコピーです。

doVerbFor()

ビジネス・オブジェクト・ハンドラーを起動し、ビジネス・オブジェクト内のアクティブな動詞によって指定されている動作を実行します。

構文

```
public int doVerbFor(ReturnStatusDescriptor rtnObj);
```

パラメーター

rtnObj このメソッドの実行に対するエラー・メッセージまたは情報メッセージを含む状況記述子オブジェクトです。統合ブローカーは、このメッセージを使用します。

戻り値

動詞操作の結果状況を示す整数です。

CxStatusConstants.SUCCEED
: 動詞操作は成功しました。

CxStatusConstants.FAIL
: 動詞操作は失敗しました。

CxStatusConstants.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

CxStatusConstants.VALCHANGE
: ビジネス・オブジェクトの 1 つ以上の値が変更されました。

CxStatusConstants.VALDUPES
: 要求された操作が、同一キー値に対して複数のレコードを検出しました。

CxStatusConstants.MULTIPLE_HITS
: コネクターは、非キー値を使用した取得時に、複数の一致レコードを検出しています。コネクターは、ビジネス・オブジェクト内で最初に一致したレコードのみを戻します。

CxStatusConstants.RETRIEVEBYCONTENT_FAILED
: コネクターは、非キー値による取得で、一致レコードを検出できませんでした。

CxStatusConstants.BO_DOES_NOT_EXIST
: 要求されたビジネス・オブジェクト・エンティティーは、データベースに存在しません。

注記

このメソッドの実行により、エラー・メッセージまたは情報メッセージによって引き渡しパラメーターが設定されます。実行後、メッセージは、統合ブローカーに返送されます。

ビジネス・オブジェクトは、そのビジネス・オブジェクト定義がサポートする動詞のすべての操作を提供します。

アクティブな動詞は、ビジネス・オブジェクト定義に含まれている動詞のリストのうちの 1 つです。ビジネス・オブジェクト用のアクティブな動詞を判別するには、`getVerb()` メソッドを使用します。

参照項目

`getVerb()` メソッドと `setVerb()` メソッド、および `BusinessObjectInterface` インターフェースの説明も参照してください。

dump()

ビジネス・オブジェクト情報を、ロギングおよびトレース用の読み取り可能なフォーマットで戻します。

構文

```
public String dump();
```

パラメーター

なし。

戻り値

フォーマット済みのビジネス・オブジェクト情報を含む `String` です。

getAppText()

このビジネス・オブジェクト定義に対するアプリケーション固有の情報を取得します。

構文

```
public String getAppText();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトに対する `AppSpecificInfo` フィールドの値を保持する `String` オブジェクトです。このメソッドは、`null` を戻す場合があります。

getAttrCount()

ビジネス・オブジェクトの属性リスト内に存在する属性の個数を取得します。

構文

```
public int getAttrCount();
```

パラメーター

なし。

戻り値

属性リスト内の属性の個数を示す整数です。

参照項目

getAttrIndex() メソッドの説明も参照してください。

getAttrDesc()

属性の名前または位置が指定された場合に、ビジネス・オブジェクトの属性の記述を取得します。

構文

```
public CxObjectAttr getAttrDesc(String name);

public CxObjectAttr getAttrDesc(int position);
```

パラメーター

name ビジネス・オブジェクト定義での属性の名前です。

position ビジネス・オブジェクト定義での属性の位置です。

戻り値

指定された属性を定義する CxObjectAttr オブジェクトです。

例外

CxObjectNoSuchAttributeException
: 指定された名前または位置が、このビジネス・オブジェクトの定義にとって有効でなかった場合にスローされます。

注記

ビジネス・オブジェクトの属性の記述を取得するには、属性名または属性リスト内におけるその位置を指定します。

- getAttrDesc() メソッドの第 1 の形式では、属性の名前の指定により、ビジネス・オブジェクトの属性の記述が取得されます。
- 第 2 の形式では、ビジネス・オブジェクト定義内の属性位置の指定により、ビジネス・オブジェクトの属性の記述が取得されます。

参照項目

getAttrName() メソッドの説明も参照してください。

getAttribute()

属性の名前が指定された場合に、属性の値を取得します。

構文

```
public Object getAttribute(String attrName);
```

パラメーター

attrName ビジネス・オブジェクト定義での属性の名前です。

戻り値

属性値を含む Object です。

例外

CxObjectNoSuchAttributeException
指定された名前が、このビジネス・オブジェクトの定義にとって有効でなかった場合にスローされます。

注記

このメソッドと getAttrValue() との違いは、getAttribute() の方が、属性値の深い取得を実行できる点にあります。例えば、Customer ビジネス・オブジェクトが、Address ビジネス・オブジェクトを含んでいる場合、getAttribute() では、コンテナーの Address[4].AddressId の 5 番目の位置にある、Address サブオブジェクトから AddressId を取得することができます。

getAttributeIndex()

ビジネス・オブジェクトの指定された属性の序数位置を取得します。

構文

```
public int getAttributeIndex(String name);
```

パラメーター

name ビジネス・オブジェクト定義での属性の名前です。

戻り値

属性の整数の序数位置です。

例外

CxObjectNoSuchAttributeException
指定された名前が、このビジネス・オブジェクトの定義にとって有効でなかった場合にスローされます。

getAttributeType()

ビジネス・オブジェクトの指定された属性の序数位置または属性名を使用して、属性のデータ型を取得します。

構文

```
public int getAttributeType(String name);
public int getAttributeType(int position);
```

パラメーター

name ビジネス・オブジェクト定義での属性の名前です。

position ビジネス・オブジェクト定義内における属性の序数位置です。

戻り値

整数で表される属性の型です。可能な属性型定数については、501 ページの表 151 を参照してください。

例外

CxObjectNoSuchAttributeException
指定された名前または位置が、このビジネス・オブジェクトの定義にとって有効でなかった場合にスローされます。

注記

ビジネス・オブジェクトの属性の型を取得するには、属性名または属性リスト内におけるその位置を指定します。

---

getAttrName()

ビジネス・オブジェクトの属性リスト内におけるその位置によって、指定された属性の名前を取得します。

構文

```
public String getAttrName(int position);
```

パラメーター

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

指定された属性の名前を含む String です。

例外

CxObjectNoSuchAttributeException
指定された位置が、このビジネス・オブジェクトの定義にとって有効でなかった場合にスローされます。

getAttrValue()

属性の名前またはビジネス・オブジェクトの属性リスト内におけるその位置が指定された場合に、ビジネス・オブジェクトの属性の値を取得します。

構文

```
public Object getAttrValue(String name);
public Object getAttrValue(int position);
```

パラメーター

name 属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のデータ型用に定義されているフォーマットにおける、指定された属性の値を含む Object です。

例外

CxObjectNoSuchAttributeException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

getAttrValue() メソッドは、java.lang.Object を戻します。この戻り値は、変数に割り当てる前に正しい型にキャストします。

参照項目

getAttrName() メソッドの説明も参照してください。

getBusinessObjectVersion()

ビジネス・オブジェクト定義のバージョンを取得します。バージョンは、大と小の改訂番号および点の要素 (-x.y.z) によって表されます。例えば、- 1.0.2 のようになります。

構文

```
public String getBusinessObjectVersion();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトのバージョン番号を含む String です。

getDefaultAttrValue()

属性の名前またはビジネス・オブジェクトの属性リスト内におけるその位置が指定された場合に、ビジネス・オブジェクトの属性のデフォルト値を取得します。

構文

```
public String getDefaultAttrValue(int position);
public String getDefaultAttrValue(String name);
```

パラメーター

name 属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性のデフォルト値が格納されている String。属性のデフォルト値が存在していない場合、メソッドは空ストリングを戻します。

例外

CxObjectNoSuchAttributeException
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

ビジネス・オブジェクトの属性のデフォルト値を取得するには、属性名または属性リスト内における属性の位置を指定します。

参照項目

getAttrValue() メソッドの説明も参照してください。

getLocale()

ビジネス・オブジェクトに関連付けられているロケールを取得します。

構文

```
public Locale getLocale();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトに関連付けられているロケールを表す Java Locale オブジェクト。

注記

getLocale() メソッドは、そのビジネス・オブジェクトに関連したビジネス・オブジェクト・ロケールを戻します。このロケールは、ビジネス・オブジェクト内のデータに関連した言語およびコードのエンコードを示すものであり、ビジネス・オブジェクト定義の名前またはその属性 (これは、英語 (U.S.) ロケール en_US に関連したコード・セット内の文字でなければなりません) に関連するものではありません。ビジネス・オブジェクトにロケールが関連付けられていない場合、コネクター・フレームワークは、コネクター・フレームワークのロケールをビジネス・オブジェクトのロケールとして割り当てます。

参照項目

createBusObj(), getGlobalLocale(), setLocale()

---

getName()

ビジネス・オブジェクトの参照するビジネス・オブジェクト定義の名前を取得します。

構文

```
public String getName();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義の名前を含む String オブジェクトです。

参照項目

getBusinessObjectVersion() メソッドの説明も参照してください。

---

getParentBusinessObject()

現在のビジネス・オブジェクトの親ビジネス・オブジェクトを取得します。このビジネス・オブジェクト・インスタンスが、ルート・オブジェクトである場合は、すなわち、親オブジェクトを持たない場合は、このメソッドは、null を戻します。

構文

```
public BusinessObjectInterface getParentBusinessObject();
```

パラメーター

なし。

戻り値

親ビジネス・オブジェクトが含まれているビジネス・オブジェクト。ただし、現在のビジネス・オブジェクトがルートとなっていて、しかも親を持たない場合は null。

---

getVerb()

ビジネス・オブジェクトのアクティブな動詞を取得します。

構文

```
public String getVerb();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトのアクティブな動詞を含む String オブジェクトです。ビジネス・オブジェクトに対してアクティブな動詞が存在しない場合は、戻り値の String は空となります。

注記

ビジネス・オブジェクト定義には、ビジネス・オブジェクトによって使用されている動詞のリストが含まれています。getVerb() メソッドにより、ビジネス・オブジェクトに対してアクティブである動詞を判別することが可能になります。

参照項目

setVerb() メソッドの説明も参照してください。

---

getVerbAppText()

特定の動詞に対するアプリケーション固有の情報の値を取得します。

構文

```
public String getVerbAppText(String verb);
```

パラメーター

verb AppSpecificInfo フィールドの値が取得される動詞です。

戻り値

指定された動詞に対する AppSpecificInfo の値を包含している String オブジェクト。ビジネス・オブジェクトが動詞に関するアプリケーション固有の情報を持たない場合、メソッドは空ストリングを戻します。

参照項目

`getVerb()` メソッドの説明も参照してください。

isBlank()

属性リスト内における指定された位置の属性の値または指定された名前の属性の値が Blank であるかどうかを判別します。

構文

```
public boolean isBlank(int position);
public boolean isBlank(String name);
```

パラメーター

name 属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性値が Blank 値に等しい場合は True を戻します。それ以外の場合は False を戻します。

isIgnore()

指定された名前の属性の値、または属性リスト内の指定された位置の値が Ignore であるかどうかを判別します。

構文

```
public boolean isIgnore(int position);
public boolean isIgnore(String name);
```

パラメーター

name 属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性値が特殊な Ignore 値に等しい場合は `true`、それ以外の場合は `false` を戻します。

isVerbSupported()

メソッドに引き渡された動詞が、このビジネス・オブジェクト定義によってサポートされているかどうかを判別します。

構文

```
public boolean isVerbSupported(String verb);
```

パラメーター

verb サポートされているかどうかをメソッドが判別する対象の動詞です。

戻り値

渡された動詞がサポートされている場合は `true` を戻し、それ以外の場合は `false` を戻します。

参照項目

`getVerb()` メソッドの説明も参照してください。

---

makeNewAttrObject()

属性に対して正しい型の新規ビジネス・オブジェクトを作成します。

構文

```
public Object makeNewAttrObject(int position);
public Object makeNewAttrObject(String name);
```

パラメーター

name 属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

戻り値

属性クラスの新規に作成されたインスタンスを含む `Object` です。

例外

`CxObjectNoSuchAttributeException`
このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

注記

`makeNewAttrObject()` メソッドは、指定された名前の属性、または属性リスト内における指定された位置の属性に適切な型の新規ビジネス・オブジェクトを作成します。例えば、コンテナー型の属性に対して、このメソッドは、`CxObjectContainerInterface` のインスタンスを戻します。

呼び出し元では、戻り値のオブジェクトを正しい型にキャストする必要があります。型がビジネス・オブジェクトである場合は、呼び出し元は、戻り値のオブジェ

クトを BusinessObjectInterface にキャストする必要があります。その値がコンテナーである属性の場合は、CxObjectContainerInterface に戻り値のオブジェクトをキャストします。

このメソッドは、普通、子オブジェクトを含む属性とともに使用します。

setAttributeWithCreate()

オブジェクトの属性を設定し、介在するオブジェクトおよびコンテナーに関係なく、オブジェクトの属性を作成します。

構文

```
public void setAttributeWithCreate(String attrName, Object value);
```

パラメーター

attrName 設定する対象の属性の名前です。

value 属性値です。

戻り値

なし。

例外

CxObjectNoSuchAttributeException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

CxObjectInvalidAttrException
: 渡された値が、特定の属性に対して有効な値でない場合、スローされます。

BusObjSpecNameNotFoundException
: ビジネス・オブジェクト定義が、データベースに発見されなかった場合にスローされます。

注記

setAttributeWithCreate() メソッドは、強制的にオブジェクトの属性を設定します。すなわち、介在するオブジェクトやコンテナーに関係なく、オブジェクトの属性を作成します。サポートされている文法は、attr1.attr2...attrThatsAContainer[index]...attrN です。例えば、Address[5].AddressObjId は、Address 属性によって参照されるビジネス・オブジェクト配列における 5 番目の要素のオブジェクト ID を指します。

setAttrValue()

属性の値を設定します。

構文

```
public void setAttrValue(String attrName, Object newval);

public void setAttrValue(int position, Object newval);
```

パラメーター

attrName その値を設定する属性の名前です。

position ビジネス・オブジェクトの属性リスト内における属性の序数位置を指定する整数です。

newval ビジネス・オブジェクトに設定される値です。

戻り値

なし。

例外

CxObjectNoSuchAttributeException
: このビジネス・オブジェクトの定義に対して指定された位置または名前が有効でない場合、スローされます。

CxObjectInvalidAttrException
: 渡された値が、特定の属性に対して有効な値でない場合、スローされます。

注記

属性値の設定には、名前メソッドまたは位置メソッドを使用することができます。

setAttrValue() メソッドは、そのパラメーターとして引き渡された値を属性の値にセットします。この値は、IBM WebSphere Business Integration システムによってサポートされている任意の型の値にすることができます。属性の型が、コンテナーやサブオブジェクト型ではない型である場合は、引き渡されたパラメーターは、String 型となります。サブオブジェクトの場合、引き渡されるパラメーターは、BusinessObjectInterface 型になります。コンテナーの場合は、引き渡されるパラメーターは、CxObjectContainerInterface 型または BusinessObjectInterface 型を取ることができます。

このメソッドは、BusinessObjectInterface 型のインスタンスを使用して、コンテナー属性に対して直接呼び出すことができます。そのインスタンスが、このコンテナーの保持する最初のビジネス・オブジェクトである場合、コンテナーは、内部的に作成され、ビジネス・オブジェクトが作成された新規コンテナーに挿入されます。その後の同様な呼び出しによって、ビジネス・オブジェクトが、この新規コンテナーに追加されます。あるいは、JavaConnectorUtil.createContainer() メソッドを使用して、CxObjectContainerInterface 型のコンテナーを作成した後、そのコンテナーにすべてのビジネス・オブジェクトを挿入して、パラメーターとして同コンテナーを使用して、setAttrValue() を起動することもできます。

参照項目

`getDefaultAttrValues()` メソッドの説明も参照してください。

setDefaultAttrValues()

現在、特殊な Blank 属性値または Ignore 属性値を持っている属性に対して、デフォルト値を設定します。

構文

```
public void setDefaultAttrValues();
```

パラメーター

なし。

戻り値

なし。

注記

デフォルト値は、ignore 値ではない有効な値です。その型がコンテナーである属性の場合は、このメソッドは空のコンテナーを作成します。ビジネス・オブジェクト内のサブオブジェクトのインスタンス用のデフォルト値は、このメソッドによって設定されます。

参照項目

`setAttrValue()` メソッドの説明も参照してください。

setLocale()

ビジネス・オブジェクトのロケールを設定します。

構文

```
public void setLocale(Locale localeObj);
public void setLocale(String localeName);
```

パラメーター

localeName 現行ビジネス・オブジェクトに関連付けるロケールの名前。

localeObj 現行のビジネス・オブジェクトに関連付けるロケールを記述する Java `Locale` オブジェクトです。

戻り値

なし。

例外

`IllegalLocaleException`
指定されたロケール名が有効ではない場合にスローされます。

注記

`setLocale()` メソッドは、そのビジネス・オブジェクトに関連したロケールを識別する、ビジネス・オブジェクト・ロケールを設定します。このロケールは、ビジネス・オブジェクト内のデータに関連した言語およびコードのエンコードを示すものであり、ビジネス・オブジェクト定義の名前またはその属性 (これは、英語 (U.S.) ロケール `en_US` に関連したコード・セット内の文字でなければなりません) に関連するものではありません。ビジネス・オブジェクトにロケールが関連付けられていない場合、コネクター・フレームワークは、コネクター・フレームワークのロケールをビジネス・オブジェクトのロケールとして割り当てます。

参照項目

getLocale()

---

setVerb()

ビジネス・オブジェクトのアクティブな動詞を設定します。

構文

```
public void setVerb(String newVerb);
```

パラメーター

newVerb ビジネス・オブジェクトの参照先となるビジネス・オブジェクト定義の動詞リスト内にある動詞です。

戻り値

なし。

例外

`BusObjInvalidVerbException`
渡された動詞がビジネス・オブジェクト定義の中の有効な動詞ではない場合、スローされます。

注記

ビジネス・オブジェクト定義には、ビジネス・オブジェクトによって使用されている動詞のリストが含まれています。アクティブな動詞として設定される動詞は、このリスト上になければなりません。1 つのビジネス・オブジェクトに対して、一度にアクティブである動詞は、1 つに限定されています。

すべてのビジネス・オブジェクトによってサポートされている典型的な動詞としては、Create、Retrieve、および Update が挙げられます。ビジネス・オブジェクト

は、追加動詞、例えば、Delete などをサポートする場合があります。ビジネス・オブジェクトをサポートしているコネクターはすべて、サポート対象の動詞をすべて実装している必要があります。

参照項目

`getVerb()` メソッドの説明も参照してください。

第 29 章 ConnectorBase クラス

ConnectorBase クラスは、下位の Java コネクターのための基底クラスです。これは、AppSide_Connector パッケージの一部です。コネクター開発者は、このクラスからコネクター・クラスを導出して、コネクター用の抽象メソッドを実装する必要があります。派生されたクラスには、コネクターのアプリケーション固有コンポーネントのコーディングが含まれます。

注: CWConnectorAgent クラスは、下位の Java コネクター・ライブラリーの ConnectorBase クラスに対するラッパーである Java コネクター・ライブラリー・メソッドです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

重要: すべての下位 Java コネクターは、この抽象クラスを拡張する必要があります。同クラスには、抽象メソッドの init()、getVersion()、getBOHandlerForBO()、pollForEvents()、および terminate() が含まれます。開発者は、上記の抽象メソッドに対して実装を用意する必要 があります。

表 148 に、ConnectorBaseクラスのメソッドを要約します。

表 148. ConnectorBase クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
executeCollaboration()	コラボレーションにビジネス・オブジェクト要求を送信します。	490
getBOHandlerForBO()	ビジネス・オブジェクトに対するハンドラーを取得します。	490
getCollabNames()	ビジネス・オブジェクト要求の処理に使用可能なコラボレーション名のリストを取得します。	491
getSupportedBusObjNames()	コネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。	491
getVersion()	アプリケーション・コネクターのバージョンを取得します。	492
gotApplEvent()	InterChange Server にビジネス・オブジェクトを送信します。	492
init()	コネクターを初期化し、アプリケーションとの接続を確立します。	495
isAgentCapableOfPolling()	このコネクター・エージェント・プロセスがポーリングを実行できるかどうかを判別します。	495
isSubscribed()	ビジネス・オブジェクトと動詞の組み合わせに対して、サブスクリプションが存在するかどうかをチェックします。	496
pollForEvents()	アプリケーションをポーリングして、ビジネス・オブジェクトへの変更を調べます。	498
terminate()	アプリケーションとの接続をクローズし、割り当てられているリソースを解放します。	498

executeCollaboration()

コラボレーションにビジネス・オブジェクト要求を送信します。これは同期要求です。

WebSphere InterChange Server
このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
public void executeCollaboration(String collabName,
   BusinessObjectInterface theBusObj, ReturnStatusDescriptor rtnStatus);
```

パラメーター

collabName ビジネス・オブジェクト要求を実行するコラボレーションの名前を指定します。

theBusObj 着信および戻りビジネス・オブジェクトです。

rtnStatus コラボレーションからのメッセージおよび実行状況または戻り状況を含む状況記述子です。

戻り値

なし。

参照項目

BusinessObjectInterface の説明も参照してください。

getBOHandlerForBO()

ビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。

構文

```
public BOHandlerBase getBOHandlerForBO(String busObjName);
```

パラメーター

busObjName ビジネス・オブジェクトの名前です。

戻り値

ビジネス・オブジェクト・ハンドラーに対する参照です。

注記

コネクター・フレームワークは、getBOHandlerForBO() メソッドを呼び出し、ビジネス・オブジェクト定義に対するビジネス・オブジェクト・ハンドラーを取得します。

重要: getBOHandlerForBO() メソッドは、ユーザーがコネクターに対して実装する *必要* のある抽象メソッドです。

複数のビジネス・オブジェクト定義に対して、1 つのビジネス・オブジェクト・ハンドラーを使用することも、各ビジネス・オブジェクト定義に対して 1 つのビジネス・オブジェクト・ハンドラーを使用することもできます。

getCollabNames()

ビジネス・オブジェクト要求の処理に使用可能なコラボレーションのリストを取得します。

WebSphere InterChange Server

このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
public String [] getCollabNames();
```

パラメーター

なし。

戻り値

コラボレーション名のリストを含む String オブジェクトの配列です。

getSupportedBusObjNames()

現在のコネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。

構文

```
public String[] getSupportedBusObjNames()
```

パラメーター

なし。

戻り値

コネクターに対してサポートされているビジネス・オブジェクトの名前のリストを含む String 配列です。

注記

getSupportedBusObjNames() メソッドは、現在のコネクターに対してサポートされているトップレベルのビジネス・オブジェクトのリストを戻します。すなわち、コネクターが、子ビジネス・オブジェクトを含むビジネス・オブジェクトをサポートしている場合、getSupportedBusObjNames() は、親オブジェクトの名前のみをその戻りリストの中に含めます。

---

getVersion()

コネクターのバージョンを取得します。

構文

```
public String getVersion();
```

パラメーター

なし。

戻り値

コネクターのアプリケーション固有のコンポーネントのバージョンを示す String です。

注記

コネクター・フレームワークは、getVersion() メソッドを呼び出し、コネクターのバージョンを取得します。

重要: getVersion() メソッドは、ユーザーがコネクターに対して、実装する必要のある抽象メソッドです。

---

gotApplEvent()

コネクター・フレームワークにビジネス・オブジェクトを送信します。

構文

```
public int gotApplEvent(BusinessObjectInterface theBusObj);
```

パラメーター

theBusObj コネクター・フレームワークに送信されるビジネス・オブジェクトのインスタンスです。

戻り値

イベント・デリバリーの結果状況を示す整数です。この整数値を以下に示す結果状況定数と比較することにより、状況が判別されます。

`CxStatusConstants.SUCCEED`
: コネクター・フレームワークは、コネクター・フレームワークへのビジネス・オブジェクトの配信に成功しました。

`CxStatusConstants.FAIL`
: イベント・デリバリーが失敗しました。

`CxStatusConstants.CONNECTOR_NOT_ACTIVE`
: コネクターは、一時停止しているためイベントを受信できません。

`CxStatusConstants.NO_SUBSCRIPTION_FOUND`
: ビジネス・オブジェクトが表すイベントに対してサブスクリプションがありません。

注記

`gotApplEvent()` メソッドは、コネクター・フレームワークに *theBusObj* ビジネス・オブジェクトを送信します。コネクター・フレームワークは、イベント・オブジェクトにある処理を施して、データを直列化し、そのデータが正しく永続化されるようにします。その後、イベントが、IIOP を介して ICS に送信されること、または (イベント通知に対してキューを使用している場合) キューに書き込まれることを保証します。

WebSphere InterChange Server

統合ブローカーが InterChange Server である場合、コネクター・フレームワークは、構成済みのデリバリー・トランスポート機構 (JMS や CORBA IIOP など) を使用して、イベントを (ビジネス・オブジェクトとして) InterChange Server に送信します。

その他の統合ブローカー

統合ブローカーが WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server である場合、コネクター・フレームワークは、JMS キューの構成済みのデリバリー・トランスポート機構を使用して、イベントを (XML メッセージとして) WebSphere MQ Integrator Broker に送信します。

`gotApplEvent()` は、コネクター・フレームワークにビジネス・オブジェクトを送信する前に、次の条件をチェックし、条件が満たされていない 場合には、関連する結果状況を戻します。

条件	結果状況
コネクターの状況がアクティブか、すなわち、コネクターは、「一時停止」状態にないか。コネクターのアプリケーション固有コンポーネントが一時停止している場合、アプリケーションに対するポーリングはすでに停止しています。	CONNECTOR_NOT_ACTIVE
イベントに対するサブスクリプションは存在するか。	NO_SUBSCRIPTION_FOUND

注: gotApplEvent() が、送信されるビジネス・オブジェクトと動詞が有効なサブスクリプションを持っていることを確認するため、gotApplEvent() の呼び出し直前に、isSubscribed() を呼び出す必要はありません。

WebSphere InterChange Server

通常、gotApplEvent() メソッドは、pollForEvents() スレッドから呼び出します。 InterChange Server は、pollForEvents() メソッドを使用して、サブスクライブしたイベントを自身に送信するようにコネクターに要求します。コネクターは、gotApplEvent()メソッドを使用して、ビジネス・オブジェクトをコネクター・フレームワークに送信します。コネクター・フレームワークでは、今度は、応答として、受信したビジネス・オブジェクトを InterChange Server に送信します。

コネクターは、pollForEvents() メソッドを使用してイベント・ストアにポーリングし、サブスクライブされたイベントが統合ブローカーに送信されるようにします。pollForEvents() 内で、コネクターは、gotApplEvent() メソッドを使用してイベント (ビジネス・オブジェクトとして表現されます) をコネクター・フレームワークに送信します。コネクター・フレームワークは、このビジネス・オブジェクトを統合ブローカーに送ります。したがって、ポーリング・メソッドは、gotApplEvent() からの戻りコードをチェックし、戻されたエラーの適切な処理を保証する仕組みになっています。例えば、ポーリング・メソッドは、イベント・デリバリーが正常に実行されるまでイベント・ストアからイベントを除去しません。ポーリング・メソッドは、gotApplEvent() の戻りコードに基づき、イベント・デリバリーの結果が反映されるようにイベント・レコードの状況を更新します。

gotApplEvent() メソッドは、イベントの非同期実行を開始します。非同期実行では、呼び出し側コードはイベントの受信を待機せず、応答も待機しません。

注: イベントの同期実行を開始するには、executeCollaboration() メソッドを使用します。同期実行では、呼び出し側コードがイベントの受信および応答を待機します。

参照項目

executeCollaboration(), isSubscribed(), pollForEvents()

BusinessObjectInterface インターフェースの説明も参照してください。

init()

コネクターを初期化します。

構文

```
public int init();
```

パラメーター

なし。

戻り値

初期化操作の状況を示す整数です。初期化操作が成功した場合は、CxStatusConstants.SUCCEED を戻します。それ以外の場合は、負の値を戻します。可能な障害値は次のとおりです。

CxStatusConstants.FAIL
: 初期化が失敗しました。

CxStatusConstants.UNABLETOLOGIN
: コネクターが、アプリケーションにログインできません。

注記

コネクター・フレームワークは、コネクターの始動時に init() メソッドを呼び出し、コネクターを初期化します。 init() メソッドにおいて、アプリケーションへログオンするなど、コネクターの初期化をすべて確実に実装します。

重要: init() メソッドは、ユーザーがコネクターに対して、実装する必要 のある抽象メソッドです。

isAgentCapableOfPolling()

コネクター・プロセスがポーリングできるかどうかを判別します。

WebSphere InterChange Server

このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
boolean isAgentCapableOfPolling();
```

パラメーター

なし。

戻り値

コネクターが、ポーリングできるかどうかを示す boolean 値です。この戻り値は、コネクターのタイプに依存します。

コネクター・プロセスのタイプ	戻り値
マスター (逐次処理)	true
マスター (並列処理)	false
スレーブ (要求)	false
スレーブ (ポーリング)	true

注記

コネクターが、単一プロセス・モードで動作するように構成されている (デフォルト設定である、ParallelProcessDegree() に 1 がセットされている) 場合、isAgentCapableOfPolling() メソッドは、常時、true を戻します。これは、同一のコネクターが、イベント・ポーリングと要求処理の両方を実行するためです。

コネクターが、並列処理モードで動作するように構成されている (ParallelProcessDegree が 1 より大きい) 場合、コネクターは、表 149 に示されているように、それぞれが特定の目的を持つ複数のプロセスから構成されます。

表 149. 並列コネクターのプロセスの目的

コネクター・プロセス	コネクター・プロセスの目的
コネクター・エージェント・マスター・プロセス	ICS からの着信イベントを受信し、どのコネクターのスレーブ・プロセスにそのイベントを送信するかを決定します。
要求処理スレーブ・プロセス	コネクターのハンドル要求
ポーリング・スレーブ・プロセス	コネクターに対するポーリングとイベント・デリバリーを処理します。

isAgentCapableOfPolling() の戻り値は、メソッドの呼び出し元のコネクター・エージェント・プロセスの目的に依存します。並列処理コネクターの場合、このメソッドは、ポーリング・スレーブとして機能する目的のコネクターから呼び出されたとき、true のみ を戻します。並列処理コネクターの詳細については、「システム管理ガイド」を参照してください。

isSubscribed()

統合ブローカーが、特定の動詞を持つ特定のビジネス・オブジェクトにサブスクライブしているかどうかを判別します。

構文

```
public boolean isSubscribed(String busObjName, String verb);
```

パラメーター

busObjName ビジネス・オブジェクトの名前です。

verb ビジネス・オブジェクトのアクティブな動詞です。

戻り値

統合ブローカーが、指定されているビジネス・オブジェクトと動詞の受信に関連する場合に、`true` を戻します。それ以外の場合は、`false` を戻します。

注記

`isSubscribed()` メソッドは、サブスクリプション・マネージャーの一部です。サブスクリプション・マネージャーは、コネクター・フレームワークから到着するすべてのサブスクライブ・メッセージとアンサブスクライブ・メッセージを追跡し、アクティブなビジネス・オブジェクト・サブスクリプションのリストを維持します。Java コネクターの場合、このサブスクリプション・マネージャーは、コネクター基底クラスの一部になります。

WebSphere InterChange Server

ビジネス・インテグレーション・システムが InterChange Server を使用している場合、ポーリング・メソッドは、指定された*動詞* を持つ *busObjName* ビジネス・オブジェクトに、コラボレーションがサブスクライブしているかどうかを判別できます。初期化時、コネクター・フレームワークは、コネクター・コントローラーから自身のサブスクリプション・リストを要求します。実行時、ポーリング・メソッドは、`isSubscribed()` を使用して、コネクター・フレームワークに照会し、あるコラボレーションが、特定のビジネス・オブジェクトにサブスクライブしていることを確認できます。ポーリング・メソッドは、あるコラボレーションが現在サブスクライブされている場合に限り、イベントを送信することができます。詳細については、14 ページの『ビジネス・オブジェクトのサブスクリプションとパブリッシュ』を参照してください。

その他の統合ブローカー

WebSphere Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、または WebSphere Business Integration Message Broker) または WebSphere Application Server を使用するビジネス・インテグレーション・システムの場合、コネクター・フレームワークでは、統合ブローカーがコネクターによってサポートされるすべての ビジネス・オブジェクトに関係していると想定します。アプリケーション固有コンポーネントが、`isSubscribed()` メソッドを使用して、特定のビジネス・オブジェクトに対するサブスクリプションに関してコネクター・フレームワークに照会する場合は、メソッドは、コネクターのサポートするビジネス・オブジェクトごと に `true` を戻します。

参照項目

gotApplEvent(), pollForEvents()

pollForEvents()

ビジネス・オブジェクトに変更を起こすイベントをポーリングして、アプリケーションのイベント・ストアを調べます。

構文

```
public int pollForEvents();
```

パラメーター

なし。

戻り値

ポーリング操作の結果状況を示す整数です。pollForEvents() メソッドは、一般的に、次の戻りコードを使用します。

CxStatusConstants.SUCCEED
: イベントの取得結果に関係なくポーリング動作は成功しました。

CxStatusConstants.FAIL
: ポーリング操作が失敗しました。

CxStatusConstants.APPRESPONSETIMEOUT
: アプリケーションが応答していません。

注記

コネクター・インフラストラクチャーが、ユーザーの設定可能な時間間隔で pollForEvents() メソッドを呼び出すため、コネクターは、サブスクライバーに関連するアプリケーションにおいてイベントを検出することができます。クラス・ライブラリーがこのメソッドを呼び出す頻度は、PollFrequency コネクター構成プロパティーによって設定されるポーリング頻度値に依存します。

重要: pollForEvents() メソッドは、独自のポーリング機構を用意する場合に実装する*必要* のある抽象メソッドです。

注: 並列処理モードで実行されているコネクターの場合は、別々のポーリング・スレーブ・プロセスを使用して、ポーリングを処理します。

terminate()

コネクターのシャットダウン時にクリーンアップ操作を実行します。

構文

```
public int terminate();
```

パラメーター

なし。

戻り値

terminate() 操作の状況値を示す整数です。

CxStatusConstants.SUCCEED
: 終了操作は成功しました。

CxStatusConstants.FAIL
: 終了操作は失敗しました。

注記

コネクター・インフラストラクチャーは、コネクターのシャットダウン時、terminate() メソッドを呼び出します。このメソッドのユーザーの実装では、優れた作業慣行として、メモリーをすべて解放し、アプリケーションからログオフすることを推奨します。

重要: terminate() メソッドは、ユーザーがコネクターに対して、実装する必要 のある抽象メソッドです。

使用すべきでないメソッド

ConnectorBase クラスの一部のメソッドは、初期のバージョンにおいてサポートされたものであり、現在はサポートされていません。これらの*使用すべきでないメソッド* は、エラーを発生させることはありませんが、IBM では、それらの使用を避けて、既存のコードを新規メソッドに移行することを推奨しています。使用すべきでないメソッドは、将来のリリースでは削除される可能性があります。

表 150 に、ConnectorBase クラスの使用すべきでないメソッドのリストを示します。(既存コネクターの変更ではなく) 新規コネクターをコーディングする場合は、このセクションを無視してください。

表 150. ConnectorBase クラスの使用すべきでないメソッド

以前のメソッド	置換
consumeSync()	executeCollaboration()

第 30 章 CxObjectAttr クラス

CxObjectAttr クラスは、Java コネクターに対するオブジェクト属性クラスです。これは、CxCommon パッケージの一部です。ビジネス・オブジェクト仕様の属性を定義します。このクラスは、属性の情報を取得するメソッドを定義します。

注: CWConnectorBusObj クラスは、下位の Java コネクター・ライブラリーの CxObjectAttr クラスのメソッドに対するラッパーである Java コネクター・ライブラリー・クラスです。CWConnectorBusObj クラスにより、ビジネス・オブジェクト、ビジネス・オブジェクト配列、ビジネス・オブジェクト定義、および属性にアクセスすることが可能になります。CWConnectorAttrType クラスは、属性型定数を定義します。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

このクラスの内容は以下のとおりです。

- 『属性型定数』
- 『メソッド』

属性型定数

CxObjectAttr クラスは、表 151 に示すように、属性型に対する等価数値と等価ストリングを定義します。

表 151. 属性型に対する等価数値と等価ストリング

属性型	等価数値定数	等価ストリング定数
ブール値	BOOLEAN	BOOLSTRING
ビジネス・オブジェクト: 複数カーディナリティー		MULTIPLECARDSTRING
ビジネス・オブジェクト: 単一カーディナリティー		SINGLECARDSTRING
日付	DATE	DATESTRING
倍精度	DOUBLE	DOUBSTRING
浮動小数点	FLOAT	FLTSTRING
整数	INTEGER	INTSTRING
ロング・テキスト	LONGTEXT	LONGTEXTSTRING
オブジェクト	OBJECT	
ストリング	STRING	STRSTRING
無効なデータ型	INVALID_TYPE_NUM	INVALID_TYPE_STRING

メソッド

表 152 に、CxObjectAttr インターフェースのメソッドを要約します。

表 152. *CxObjectAttr* クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
equals()	指定された属性が、この属性と同一であるかを判別します。	502
getAppText()	この属性のアプリケーション固有の情報を取得します。	503
getCardinality()	属性のカーディナリティーを取得します。	503
getDefault()	属性のデフォルト値を取得します。	503
getMaxLength()	属性値の最大長を取得します。	504
getName()	属性名を取得します。	504
getRelationType()	属性関係の型を取得します。	504
getTypeName()	属性の型を取得します。	504
getTypeNum()	属性の数値型コードを取得します。	505
hasCardinality()	属性のカーディナリティーを、パラメーターとして渡されたカーディナリティー値と比較します。	505
hasName()	メソッドに渡された名前を属性の名前と比較します。	506
hasType()	属性の型が、渡された型と同一であるかを確認します。	506
isForeignKeyAttr()	属性が、オブジェクトの外部キーの一部であるかを確認します。	506
isKeyAttr()	属性が、オブジェクトのキー・セットの一部であるかを確認します。	507
isMultipleCard()	属性が複数カーディナリティーであるかどうかを取得します。	507
isObjectType()	属性の型が、オブジェクト型であるかどうかを取得します。	507
isRequiredAttr()	この属性が、ビジネス・オブジェクトにとって必須の属性であるかどうかを確認します。	508
isType()	属性の型が、渡されたパラメーター値に一致するかを確認します。	508

equals()

指定された属性が、現在の属性と同一であるかを判別します。

構文

```
public boolean equals(Object obj)
```

パラメーター

obj 現在の属性と比較される属性を表すオブジェクトです。

戻り値

指定された属性が、この属性と同一である場合は True を戻します。それ以外の場合は、False を戻します。

注記

このメソッドは、指定された属性が、名前、型、キーであるかどうか、外部キーであるかどうか、および必須の属性であるかどうかのすべての点において、この属性と一致するかを確認します。

getAppText()

この属性のアプリケーション固有の情報を取得します。

構文

```
public String getAppText();
```

パラメーター

なし。

戻り値

属性に対する AppSpecificText フィールドの値を保持する String オブジェクトです。属性が、アプリケーション固有の情報を持たない場合、このメソッドは null を戻します。

getCardinality()

属性のカーディナリティーを取得します。

構文

```
public String getCardinality();
```

パラメーター

なし。

戻り値

属性のカーディナリティーが格納されている String です。このストリングの値は、1 または n です。

getDefault()

この属性のデフォルト値を取得します。

構文

```
public String getDefault();
```

パラメーター

なし。

戻り値

属性のデフォルト値を含む String または null です。

getMaxLength()

ビジネス・オブジェクト定義から属性の最大長を取得します。

構文

```
int getMaxLength();
```

パラメーター

なし。

戻り値

属性値が取ることのできる最大長をバイト単位で示す整数です。

getName()

属性の名前を取得します。

構文

```
public String getName();
```

パラメーター

なし。

戻り値

指定された属性を含む `String` です。

getRelationType()

属性の関係型を取得します。複合属性 (サブオブジェクトや配列など) の場合、戻される関係型は包含関係となります。

構文

```
public String getRelationType();
```

パラメーター

なし。

戻り値

属性の関係型を含む `String` です。

getTypeName()

属性のデータ型の名前を取得します。

構文

```
public String getTypeName();
```

パラメーター

なし。

戻り値

属性の型の名前を含む String です。ストリング属性型のリストについては、501 ページの表 151 を参照してください。

getTypeNum()

属性のデータ型に対する数値型コードを取得します。

構文

```
public String getTypeNum();
```

パラメーター

なし。

戻り値

属性の型の数値型コードです。数値属性型定数のリストについては、501 ページの表 151 を参照してください。

hasCardinality()

属性が、パラメーターとして渡されたカーディナリティー値と同一のカーディナリティーを持つかどうかを判別します。このメソッドは、複合属性 (サブオブジェクトやコンテナー) のカーディナリティーをチェックするために使用します。有効なカーディナリティー値は、1 から n の範囲にあります。

構文

```
public boolean hasCardinality(String card);
```

パラメーター

card 検査に使用されるカーディナリティー値です。次のカーディナリティー定数のいずれかを使用します。

```
CxObjectAttr.MULTIPLECARDSTRING
CxObjectAttr.SINGLECARDSTRING
```

戻り値

属性のカーディナリティーが、パラメーターの値と一致した場合は、True を戻します。それ以外の場合は、False を戻します。

hasName()

属性の名前が、パラメーターとして渡された名前に一致するかを判別します。

構文

```
public boolean hasName(String name);
```

パラメーター

name メソッドに渡される属性の名前です。

戻り値

属性の名前が、渡された名前と一致する場合は、True を戻します。それ以外の場合は、False を戻します。

hasType()

属性のデータ型が、パラメーターとして渡された型名に一致するかを判別します。

構文

```
public boolean hasType(String typeName);
```

パラメーター

typeName メソッドに渡される属性の型です。次のストリング属性型定数のいずれかを使用します。

```
CxObjectAttr.BOOLSTRING
CxObjectAttr.DATESTRING
CxObjectAttr.DOUBSTRING
CxObjectAttr.FLTSTRING
CxObjectAttr.INTSTRING
CxObjectAttr.LONGTEXTSTRING
CxObjectAttr.STRSTRING
```

戻り値

属性の型が渡された型名と一致している場合は、True を戻します。それ以外の場合は、False を戻します。

isForeignKeyAttr()

この属性が、ビジネス・オブジェクトの外部キーの一部であるかを判別します。

構文

```
public boolean isForeignKeyAttr();
```

パラメーター

なし。

戻り値

属性がビジネス・オブジェクトの外部キー、またはその外部キーの一部となっている場合は True を戻します。それ以外の場合は、False を戻します。

isKeyAttr()

この属性が、ビジネス・オブジェクトのキー・セットの一部であるかを判別します。

構文

```
public boolean isKeyAttr();
```

パラメーター

なし。

戻り値

属性がキーまたはキー・セットの一部である場合は、True を戻します。それ以外の場合は、False を戻します。

isMultipleCard()

この属性が複数カーディナリティーであるかどうかを判別します。

構文

```
public boolean isMultipleCard();
```

パラメーター

なし。

戻り値

属性が複数カーディナリティーである場合は、True を戻します。それ以外の場合は、False を戻します。

isObjectType()

この属性のデータ型が、オブジェクト型であるか、すなわち、複合属性 (コンテナーまたはサブオブジェクト) であるかを判別します。

構文

```
public boolean isObjectType();
```

パラメーター

なし。

戻り値

属性がオブジェクト型すなわち複合属性である場合 (コンテナー、サブオブジェクトなど) は、True を戻します。それ以外の場合は、False を戻します。

isRequiredAttr()

この属性が、ビジネス・オブジェクトにとって必須の属性であるかどうかを判別します。属性が必須の場合は、値を持つ必要があります。

構文

```
public boolean isRequiredAttr();
```

パラメーター

なし。

戻り値

属性がビジネス・オブジェクトに必須である場合は、True を戻します。それ以外の場合は、False を戻します。

isType()

メソッドに渡された値が、属性の値と同一の型であることを判別します。

構文

```
public boolean isType(Object value);
```

パラメーター

value この属性の型に対して比較チェックが行われます。

戻り値

属性の型が渡された型と一致する場合は、True を戻します。それ以外の場合は、False を戻します。

第 31 章 CxObjectContainerInterface インターフェース

CxObjectContainerInterface インターフェースは、1 つ以上の子ビジネス・オブジェクトから成る配列を作成し、維持します。これは、CxCommon パッケージの一部です。このインターフェースは、階層構造を持つビジネス・オブジェクトをサポートします。CxObjectContainerInterface から作成された各オブジェクト・インスタンスは、同型のビジネス・オブジェクトを挿入できるコンテナー・オブジェクトです。挿入されるオブジェクトは、親ビジネス・オブジェクトの複合属性によって参照されるビジネス・オブジェクト定義のインスタンスです。挿入されるオブジェクトは、階層内の子ビジネス・オブジェクトとなります。

注: CWConnectorBusObj クラスは、下位の Java コネクター・ライブラリーの CxObjectContainerInterface インターフェースのメソッドに対するラッパーである Java コネクター・ライブラリー・メソッドです。CWConnectorBusObj クラスにより、ビジネス・オブジェクト、ビジネス・オブジェクト配列、ビジネス・オブジェクト定義、および属性にアクセスすることが可能になります。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。Java コネクター・ライブラリーのクラスの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

表 153 に、CxObjectContainerInterface インターフェースのメソッドを要約します。

表 153. CxObjectContainerInterface インターフェースのメンバー・メソッド

メンバー・メソッド	説明	ページ
getBusinessObject()	ビジネス・オブジェクト配列内の指定された位置に存在する子ビジネス・オブジェクトを取得します。	510
getObjectCount()	ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの個数を取得します。	510
insertBusinessObject()	ビジネス・オブジェクト配列の次の使用可能な位置に、子ビジネス・オブジェクトを挿入します。	511
removeAllObjects()	ビジネス・オブジェクト配列内のすべてのビジネス・オブジェクトを除去します。	511
removeBusinessObjectAt()	ビジネス・オブジェクト配列内の指定された位置にあるビジネス・オブジェクトを除去します。	511
setBusinessObject()	ビジネス・オブジェクト配列の指定された位置に、子ビジネス・オブジェクトを挿入します。	512

注: 子ビジネス・オブジェクトの配列に対して使用すべきでない名前は、「business object container」です。この用語は、ビジネス・オブジェクト配列内の子ビジネス・オブジェクトに対するアクセス方法を提供するコネクター・ライブラリー・クラスに名前を付けるためにも使用されます。このクラスは、ビジネス・オブジェクトから成る配列に対する処理メソッドを提供するクラスと見なすことができます。

getBusinessObject()

ビジネス・オブジェクト配列内の指定された位置に存在する子ビジネス・オブジェクトを取得します。

構文

```
public BusinessObjectInterface getBusinessObject(int index);
```

パラメーター

index ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの位置を示す整数です。

戻り値

子ビジネス・オブジェクト、またはビジネス・オブジェクト配列内の指定されている位置にビジネス・オブジェクトが存在しなかった場合は、`null` となります。

参照項目

`setBusinessObject()` メソッドの説明も参照してください。

getObjectCount()

ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの個数を取得します。

構文

```
public int getObjectCount();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの個数を示す整数です。

注記

`insertBusinessObject()` メソッドは、ビジネス・オブジェクト配列に子ビジネス・オブジェクトを挿入するために使用します。

参照項目

`insertBusinessObject()` メソッドの説明も参照してください。

insertBusinessObject()

ビジネス・オブジェクト配列の次の使用可能な位置に、子ビジネス・オブジェクトを挿入します。

構文

```
public void insertBusinessObject(BusinessObjectInterface theChildBusObj);
```

パラメーター

theChildBusObj 挿入する子ビジネス・オブジェクトです。

戻り値

なし。

例外

CxObjectInvalidAttrException
渡されたビジネス・オブジェクトが、配列に格納されているオブジェクトと同型でない場合にスローされます。

参照項目

setBusinessObject() メソッドの説明も参照してください。

removeAllObjects()

ビジネス・オブジェクト配列内のすべてのビジネス・オブジェクトを除去します。

構文

```
public void removeAllObjects();
```

パラメーター

なし。

戻り値

なし。

removeBusinessObjectAt()

ビジネス・オブジェクト配列内の指定された位置にあるビジネス・オブジェクトを除去します。

構文

```
public void removeBusinessObjectAt(int index);
```

パラメーター

index ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの位置を指定する整数です。

戻り値

なし。

例外

`CxObjectNoSuchAttributeException`
指定された位置が、このビジネス・オブジェクトにとって有効でなかった場合にスローされます。

注記

除去操作後は、ビジネス・オブジェクト配列が圧縮されます。索引番号が、減らされたビジネス・オブジェクトより大きいビジネス・オブジェクトすべてに対して、その索引が、1 減算されます。

---

setBusinessObject()

ビジネス・オブジェクト配列の指定された位置に、子ビジネス・オブジェクトを挿入します。

構文

```
public BusinessObjectInterface setBusinessObject(int index,
    BusinessObjectInterface theChildBusObj);
```

パラメーター

index ビジネス・オブジェクト配列内の子ビジネス・オブジェクトの位置を指定する整数です。

theChildBusObj 子ビジネス・オブジェクトです。

戻り値

挿入の結果、ビジネス・オブジェクトが交換された場合は、交換前の元のビジネス・オブジェクトを戻します。それ以外の場合は、`null` を戻します。

例外

`CxObjectInvalidAttrException`
渡されたビジネス・オブジェクト属性の型が、そのビジネス・オブジェクト配列が処理する型ではない場合にスローされます。

注記

指定された位置にすでに存在するビジネス・オブジェクトは、挿入される新規ビジネス・オブジェクトによって置き換えられます。元のビジネス・オブジェクトは削除され、呼び出し元に戻されます。

参照項目

`getBusinessObject()` メソッドの説明も参照してください。

第 32 章 CxProperty クラス

CxProperty クラスは、下位の Java コネクターの階層コネクター構成プロパティーを表します。階層コネクター構成プロパティーは 1 つ以上の値を取ることができ、また、これらの値はストリング値または他の (子) コネクター・プロパティーにすることができます。

注: CWProperty クラスは、下位の Java コネクター・ライブラリーの CxProperty クラスに対するラッパーである Java コネクター・ライブラリーのメソッドです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、 267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

表 154 に、CxProperty クラスのメソッドを要約します。

表 154. CxProperty クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
CxProperty()	コネクター・プロパティー・オブジェクトを作成します。	515
getAllChildProps()	階層コネクター構成プロパティーからすべての 子プロパティーを取得します。	516
getChildProp()	プロパティー階層の任意のレベルで、階層コネクター構成プロパティーの*指定された* 子プロパティーを取得します。	517
getEncryptionFlag()	コネクター構成プロパティーの暗号化フラグを取得します。	518
getName()	コネクター構成プロパティーの名前を取得します。	519
getStringValues()	階層コネクター構成プロパティーからすべてのストリング値を取得します。	519
hasChildren()	コネクター構成プロパティーが子プロパティーを持っているかどうかを判別します。	520
setEncryptionFlag()	階層コネクター構成プロパティーの暗号化フラグを設定します。	521
setValues()	階層コネクター構成プロパティーの値を設定します。	521

CxProperty()

階層コネクター・プロパティー・オブジェクトを作成します。

構文

```
public CxProperty();
public CxProperty(String propName, String simplePropValue);
public CxProperty(String propName, Object[] hierPropValues);
public CxProperty(String propName, org.w3c.dom.Element xmlElement);
```

パラメーター

propName コネクター構成プロパティーの名前を指定します。

simplePropValue
シンプル・コネクター・プロパティーの初期化に使用するストリング値です。

hierPropValues 階層コネクター・プロパティーの初期化に使用するコネクター・プロパティー (CWProperty) オブジェクトの配列です。

xmlElement コネクター・プロパティーの初期化に使用する XML Element オブジェクトです。

戻り値

新規に作成された階層コネクター・プロパティーを含む CxProperty オブジェクトです。

注記

CxProperty() コンストラクターの形式には、以下の種類があります。

- 最初の形式は、空のコネクター・プロパティー・オブジェクトを作成します。CWProperty クラスの他のメソッドを使用してこのオブジェクトを取り込むことができます。
- 2 番目の形式は、シンプル・コネクター・プロパティーのコネクター・プロパティー・オブジェクトを、指定したプロパティー名およびストリング値を使用して作成します。
- 3 番目の形式は、階層コネクター・プロパティーのコネクター・プロパティー・オブジェクトと、指定したプロパティー名および階層プロパティーの配列を作成します。
- 4 番目の形式は、コネクター・プロパティー・オブジェクトと XML Element オブジェクトを作成します。

---

getAllChildProps()

階層コネクター構成プロパティーのすべての 子プロパティーを取得します。

構文

```
public CxProperty[] getAllChildProps();
```

パラメーター

なし。

戻り値

CxProperty オブジェクトの配列への参照。各配列は、階層コネクター・プロパティーのコネクター・プロパティーを 1 つ表します。階層コネクター・プロパティーに子プロパティーが含まれていない場合、メソッドは null を戻します。

例外

なし。

注記

getAllChildProps() メソッドは、階層コネクター構成プロパティーのすべての 子プロパティーを取得します。取得されるプロパティーは、現在の階層プロパティーの子のプロパティーのみです。孫やひ孫などのプロパティーは含みません。階層内の下位の子プロパティーを取得するには、まず最初に、特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getAllChildProps() または getChildProp() などのメソッドを使用してその子を取得します。

注: getChildProp() を使用して、指定された子や孫など、プロパティー階層の下位のプロパティーを取得できます。

指定された子プロパティーを取得するには、getChildProp() メソッドを使用します。指定されたプレフィックスを持つすべての子プロパティーを取得するには、getChildPropsWithPrefix() メソッドを使用できます。

参照項目

getChildProp()

---

getChildProp()

任意のレベルのプロパティー階層で、階層コネクター構成プロパティーの指定された子プロパティーを取得します。

構文

```
public CxProperty getChildProp(String propName);
```

パラメーター

propName 取得するコネクター構成プロパティーの名前を指定します。

戻り値

階層から取得されるプロパティーを含む CxProperty オブジェクト。指定されたプロパティーが現在の階層コネクター・プロパティーに存在しない場合、メソッドは null を戻します。

例外

なし。

注記

getChildProp() メソッドは、*propName* と一致する名前を持つ子プロパティーを階層コネクター構成プロパティーから取得します。現在のプロパティー階層の *任意のレベル* で子プロパティーを取得できるため、孫やひ孫などのプロパティーを指定できます。取得される子プロパティーの *propName* の形式は以下のとおりです。

child/grandchild/great-grandchild/....

例えば、 453 ページの図 77 に示すプロパティー階層があるとします。Listener1 のポート名を取得するには、まず ProtocolListener のトップレベルのコネクター・オブジェクトを取得します (例えば、topLevelProp に取り込みます)。次に、以下の呼び出しを使用して Listener1 のポート名を取得します。

```
CxProperty listenerPort = topLevelProp.getChildProp("Listener1/Port");
```

階層の最上位ですべての 子プロパティーを取得するには、getAllChildProps() メソッドを使用できます。

参照項目

getAllChildProps()

---

getEncryptionFlag()

階層コネクター構成プロパティーの暗号化フラグをコネクター・プロパティー・オブジェクトから取得します。

構文

```
public boolean getEncryptionFlag();
```

パラメーター

なし。

戻り値

現在のコネクター構成プロパティー値が暗号化されているかどうかを示すブール値です。

例外

なし。

注記

getEncryptionFlag() メソッドは、ブール値の暗号化フラグをコネクター・プロパティー・オブジェクトから取得します。このフラグは、コネクター・プロパティーのストリング値が暗号化されているかどうかを示します。

注: Connector Configurator では、暗号化された値はアスタリスク (*) 文字のストリングとして表示されます。

参照項目

setEncryptionFlag()

getName()

コネクター構成プロパティーの名前をコネクター・プロパティー・オブジェクトから取得します。

構文

```
public String getName();
```

パラメーター

なし。

戻り値

コネクター構成プロパティーの名前を含むストリングです。

例外

なし。

getStringValues()

階層コネクター構成プロパティーのすべての ストリング値を取得します。

構文

```
public String[] getStringValues();
```

パラメーター

なし。

戻り値

String オブジェクトの配列への参照。各配列は、階層コネクター・プロパティーのストリング値を 1 つ表します。階層コネクター・プロパティーにストリング値が含まれていない場合、メソッドは null を戻します。

例外

なし。

注記

getStringValues() メソッドは、階層コネクター構成プロパティーのすべての ストリング値を取得します。取得されるストリング値は、現在の階層プロパティーのストリング値のみです。子プロパティーの値は含まれません。階層の低位のストリング値を取得するには、以下のいずれかを実行します。

- getChildPropValue() メソッドを使用して、指定された子プロパティーのストリング値を取得します。

- 特定レベルのプロパティーのコネクター・プロパティー・オブジェクトを取得し、次に getStringValues() または getChildPropValue() などのメソッドを使用してストリング値を取得します。

getHierChildProps() を呼び出す前に、hasChildren() メソッドを使用して、階層コネクター・プロパティーが子プロパティーを持っているかどうかを確認できます。子プロパティーを取得するには、getChildProp() または getAllChildProps() メソッドを使用します。

参照項目

getChildPropValue(), getAllChildProps(), getChildProp(), setValues()

---

hasChildren()

現在のコネクター・プロパティーが子プロパティーを持っているかどうかを判別します。

構文

```
public boolean hasChildren();
```

パラメーター

なし。

戻り値

階層コネクター・プロパティーが子プロパティーを持っているかどうかを示す boolean です。子プロパティーを持っている場合、メソッドは true を戻します。それ以外の場合は、false を戻します。

例外

なし。

注記

hasChildren() メソッドは、次に示すように、階層コネクター・プロパティーの値を抽出するためにどの CWProperty メソッドを使用するかを判別する際に役立ちます。

- hasChildren() が true を戻す場合、次のいずれかの値のメソッドを使用して、子プロパティーを取得します。

すべての子プロパティーの取得	getHierChildProps()
指定された子プロパティーの取得	getHierChildProp()

- hasChildren() が false を戻す場合、次のいずれかの値のメソッドを使用して、ストリング値を取得します。

すべてのストリング値の取得	getStringValues()

指定された子プロパティーのストリング値の取得	`getChildPropValue()`

参照項目

getChildPropValue(), getHierChildProp(), getConnectorBOHandlerForBO(), getStringValues(),getVersion()

setEncryptionFlag()

コネクター構成プロパティーの暗号化フラグを、そのコネクター・プロパティー・オブジェクトに設定します。

構文

```
public void setEncryptionFlag(boolean encryptFlag);
```

パラメーター

encryptFlag 現在のコネクター構成プロパティー値を暗号化すべきかどうかを示すブール値です。

戻り値

なし。

注記

`setEncryptionFlag()` メソッドは、コネクター・プロパティー・オブジェクトのブール値の暗号化フラグを設定します。このフラグは、コネクター・プロパティーのストリング値が暗号化されているかどうかを示します。

注: Connector Configurator では、暗号化された値はアスタリスク (*) 文字のストリングとして表示されます。

参照項目

getEncryptionFlag()

setValues()

階層コネクター構成プロパティーの値を設定します。

構文

```
public void setValues(Object[] propValues);
```

パラメーター

propValues `Object` 値の配列です。各配列エレメントは、単一のプロパティー値です。

戻り値

なし。

例外

なし。

注記

setValues() メソッドの使用により、階層コネクター構成プロパティーの値を設定できます。Object の配列である *propValues* 配列にプロパティー値を指定します。これにより、この単一の配列にストリング値と子プロパティー値の両方を渡すことができます。*propValues* 配列へのプロパティー値の割り当ては、それらが階層コネクター・プロパティー内で定義されている順に行うようにしてください。

例えば、setValues() への次の呼び出しは、ストリング値と子プロパティーの両方を topLevelProp のコネクター・プロパティーに割り当てます。

```
Object[] propValues;
CxProperty childProp;

propValues[0] = "stringValue"
propValue[1] = childProp;
topLevelProp.setValues(propValues);
```

参照項目

getHierChildProp(), getConnectorBOHandlerForBO(), getStringValues()

第 33 章 CxStatusConstants クラス

CxStatusConstants クラスは、下位の Java コネクター・ライブラリーの結果状況定数を定義します。これは、CxCommon パッケージの一部です。

注: CWConnectorConstant クラスは、下位の Java コネクター・ライブラリーの CxStatusConstants クラスに対するラッパーである Java コネクター・ライブラリー・クラスです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、 267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

結果状況定数

下位の Java コネクター・ライブラリーの多数のメソッドは、整数の結果状況を戻して、メソッドの成功を示します。表 155 に、CxStatusConstants クラスで定義されている静的結果状況定数を要約します。

表 155. CxStatusConstants クラスの結果状況定数

定数名	意味
SUCCEED	操作は正常に終了しました。
FAIL	特定できない原因のため、操作に失敗しました。
APPRESPONSETIMEOUT	アプリケーションが応答していません。
BO_DOES_NOT_EXIST	要求されたビジネス・オブジェクトが存在しません。
CONNECTOR_NOT_ACTIVE	コネクターがアクティブではありません。すなわち、一時停止状態にあります。
MULTIPLE_HITS	内容による取得が統合ブローカーによって要求されましたが、コネクターは条件に一致するレコードを複数検出しました。この状況は、取得要件に一致するレコードが複数存在していることを示します。
NO_SUBSCRIPTION_FOUND	ビジネス・オブジェクトのサブスクリプションを発見できませんでした。
RETRIEVEBYCONTENT_FAILED	内容による取得が失敗しました。
UNABLETOLOGIN	アプリケーションにログインできません。
VALCHANGE	操作が正常に完了して、ターゲット・アプリケーションのオブジェクトの値が変更されました。
VALDUPES	要求された操作は、同一のキー・フィールド (1 つ以上) に対して、複数のレコードが発見されたため、失敗しました。

第 34 章 JavaConnectorUtil クラス

JavaConnectorUtil クラスは、下位 Java コネクターで使用される各種のユーティリティー・メソッドを含む、最後のクラスです。これは、AppSide_Connector パッケージの一部です。コネクター開発者は、メッセージの生成とロギング、およびビジネス・オブジェクトの作成のため、これらの静的メソッドを使用することができます。

注: CWConnectorUtil クラスは、下位の Java コネクター・ライブラリーの JavaConnectorUtil クラスに対するラッパーである Java コネクター・ライブラリー・クラスです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、 267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

このクラスの内容は以下のとおりです。

静的定数

JavaConnectorUtil クラスは、複数の静的定数を定義します。表 156 を参照してください。

表 156. JavaConnectorUtil クラスで定義されている静的定数

定数名	意味
メッセージ・ファイル定数	
CONNECTOR_MESSAGE_FILE	コネクター・メッセージ・ファイルを使用して、メッセージを生成します。
INFRASTRUCTURE_MESSAGE_FILE	InterChange Server メッセージ・ファイル (InterchangeSystem.txt) は、メッセージの生成に使用します。 **重要:** コネクターは InterchangeSystem.txt ファイルからメッセージを取得してはなりません。コネクターは、必ず自身のローカル・コネクター・メッセージ・ファイルを使用する必要があります。
メッセージ・タイプ定数	
XRD_WARNING	警告メッセージ
XRD_TRACE	トレース・メッセージ
XRD_INFO	情報メッセージ
XRD_ERROR	エラー・メッセージ
XRD_FATAL	致命エラー・メッセージ
トレース・レベル定数	
LEVEL1	トレース・レベル 1
LEVEL2	トレース・レベル 2
LEVEL3	トレース・レベル 3
LEVEL4	トレース・レベル 4
LEVEL5	トレース・レベル 5

メソッド

表 157 に、JavaConnectorUtil クラスのメソッドを要約します。

表 157. JavaConnectorUtil クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
createBusinessObject()	ビジネス・オブジェクトを作成します。	526
createContainer()	コンテナーを作成します。	527
generateMsg()	トレース・レベルに応じて、ユーザーの指定したメッセージ・ファイルからメッセージを生成します。トレース・レベルは、オプションとして指定できます。	528
getAllConfigProp()	コネクターのすべてのプロパティーを階層コネクター・プロパティーとして取得します。	529
getAllConnectorAgentProperties()	コネクターのすべてのプロパティーを取得します。	530
getAllStandardProperties()	すべての標準コネクター・プロパティーを階層コネクター・プロパティーとして取得します。	530
getAllUserProperties()	すべてのコネクター固有のプロパティーを階層コネクター・プロパティーとして取得します。	531
getBlankValue()	特殊な Blank 属性値を取得します。	531
getConfigProp()	リポジトリーから、コネクターのプロパティーを取得します。	532
getEncoding()	コネクター・フレームワークが使用している文字エンコードを取得します。	532
getIgnoreValue()	特殊な Ignore 属性値を取得します。	533
getLocale()	コネクター・フレームワークのロケールを取得します。	533
getOneConfigProp()	指定された階層コネクター・プロパティーを取得します。	534
getSupportedBusObjNames()	コネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。	535
initAndValidateAttributes()	必須の属性すべてをそのデフォルト値に初期化します。	536
isBlankValue()	特殊な Blank 属性値に値が等しいかどうかを検査します。	537
isIgnoreValue()	特殊な Ignore 属性値に値が等しいかどうかを検査します。	538
isTraceEnabled()	トレース・レベルが、指定されたトレース・レベル以上であるかテストします。	538
logMsg()	メッセージをログに記録します。エラーまたは致命的エラーがメッセージ重大度に設定されている場合は、電子メール・メッセージをオプションとして送信できます。	539
traceWrite()	トレース・メッセージを書き込みます。	540

createBusinessObject()

新しいビジネス・オブジェクトを作成します。

構文

```
public static BusinessObjectInterface createBusinessObject(
    String busObjName);

public static BusinessObjectInterface createBusinessObject(
    String busObjName, Locale localeObject);

public static BusinessObjectInterface createBusObj(
    String busObjName, String localeName);
```

パラメーター

busObjName　作成されるビジネス・オブジェクトの名前。

localeObject
ビジネス・オブジェクトに関連付けるロケールを識別する Java Locale オブジェクト。

localeName
ビジネス・オブジェクトに関連付けるロケールの名前。

戻り値

新規作成ビジネス・オブジェクトを含む BusinessObjectInterface オブジェクトです。

例外

BusObjSpecNameNotFoundException
指定された名前に対して、ビジネス・オブジェクト仕様が発見されなかった場合にスローされます。

注記

createBusinessObject() メソッドは、ユーザーにより *busObjName* に指定されたビジネス・オブジェクト定義をその型とする新規のビジネス・オブジェクト・インスタンスを作成します。 *localeObject* または *localeName* を指定した場合、このロケールは、ビジネス・オブジェクト内のデータに適用されますが、ビジネス・オブジェクト定義の名前またはその属性には適用されません (これは、米国英語ロケール en_US に関連付けられたコード・セット内の文字でなければなりません)。
localeName のフォーマットの説明については、64 ページの『国際化対応コネクターの設計上の考慮事項』を参照してください。

createContainer()

ビジネス・オブジェクト配列 (コンテナー) のインスタンスを作成します。

構文

```
public static CxObjectContainerInterface createContainer(String name);
```

パラメーター

name 作成するビジネス・オブジェクト・コンテナーの名前を指定します。

戻り値

新規作成ビジネス・オブジェクトを含む CxObjectContainerInterface オブジェクトです。

例外

BusObjSpecNameNotFoundException
指定された名前に対して、ビジネス・オブジェクト仕様が発見されなかった場合にスローされます。

---

generateMsg()

メッセージ・ファイル内の一連の定義済みメッセージからメッセージを生成します。

構文

```
public final static String generateMsg(int traceLevel, int msgNum,
    int msgType, int isGlobal, int argCount, Vector msgParams);

public final static String generateMsg(int msgNum, int msgType,
    int isGlobal, int argCount, Vector msgParams);
```

パラメーター

traceLevel メッセージを生成するトレース・レベルを指定します。このパラメーターを省略すると、メソッドは、トレース・レベルに無関係にメッセージを生成します。メッセージは、*traceLevel* 値が、コネクターの現在のトレース・レベル以下である場合に限り生成されます。

msgNum メッセージ・ファイル内のメッセージ番号 (ID) を指定します。

msgType 次のメッセージ・タイプのいずれかにします。

```
JavaConnectorUtil.XRD_WARNING
JavaConnectorUtil.XRD_ERROR
JavaConnectorUtil.XRD_FATAL
JavaConnectorUtil.XRD_INFO
JavaConnectorUtil.XRD_TRACE
```

isGlobal メッセージ・ファイルがコネクター・メッセージ・ファイルであることを示す場合、CWConnectorLogAndTrace クラスに定義されている CONNECTOR_MESSAGE_FILE メッセージ・ファイル定数です。

argCount メッセージ・テキスト内のパラメーターの個数を示す整数です。個数を確認するには、メッセージ・ファイル内のメッセージを参照します。

msgParams メッセージ・テキスト用のパラメーターのリストです。

戻り値

生成されたメッセージを含む String またはトレース・レベルがコネクターのトレース・レベルより大きい場合は、null となります。

注記

generateMsg() メソッドの形式には、以下の 2 種類があります。

- メッセージのトレースに対して、最初の形式のメソッドを使用します (そこでは、*traceLevel* が 1 番目のパラメーターです)。メッセージを生成するためには、トレース・レベルをコネクターのトレース・レベル以下にする必要があります。次に traceWrite() メソッドを使用して、トレース・メッセージをログ宛先に送信します。
- ロギングに対して、2 番目の形式のシグニチャーを使用します (そこでは、*msgNum* が 1 番目のパラメーターです)。次に logMsg() メソッドを使用してログ・メッセージをログ宛先に送信します。

---

getAllConfigProp()

現在のコネクター用の全構成プロパティーのリストを、階層コネクター・プロパティーとして取得します。

構文

```
public static CxProperty[] getAllConfigProp();
```

パラメーター

なし。

戻り値

CxProperty オブジェクトの配列への参照。各配列は、現在のコネクターのコネクター・プロパティーを 1 つ含んでいます。

注記

getAllConfigProp() メソッドは、CxProperty オブジェクトの配列としてコネクター構成プロパティーを取得します。各コネクター・プロパティー (CxProperty) オブジェクトには、単一のコネクター・プロパティーが含まれ、単一値、別のプロパティー、または値と子プロパティーの組み合わせを保持することができます。CxProperty クラスのメソッド (getAllChildProps() または getStringValues() など) を使用して、コネクター・プロパティー・オブジェクトから値を取得します。

参照項目

getConfigProp(), getAllConnectorAgentProperties(), getAllStandardProperties(), getAllUserProperties()

getAllConnectorAgentProperties()

現在のコネクター用の全構成プロパティーのリストを取得します。

構文

```
public static Hashtable getAllConnectorAgentProperties();
```

パラメーター

なし。

戻り値

現在のコネクターに対するコネクター・プロパティーを含む Hashtable オブジェクトに対する参照です。

注記

getAllConnectorAgentProperties() メソッドは、キーを値にマップする Java Hashtable オブジェクトとして、コネクター構成プロパティーを取得します。キーはプロパティーの名前であり、値は関連付けられているプロパティーの値です。Hashtable 構造体のメソッド (例えば keys() や elements()) を使用して、この構造体から情報を取得してください。

例

```
Hashtable ht = new Hastable();
ht = JavaConnectorUtil.getAllConnectorAgentProperties();
int size = ht.size();
Enumeration properties = ht.keys();
Enumeration values = ht.elements();

while (properties.hasMoreElements()) {
    System.out.print((String)properties.nextElement());
    System.out.print("=");
    System.out.println((String)values.nextElement());
    System.out.println("Property set");
}
```

getAllStandardProperties()

コネクターのすべての標準プロパティーを取得します。

構文

```
static CxProperty[] getAllStandardProperties();
```

パラメーター

なし。

戻り値

CxProperty オブジェクトの配列への参照。各配列は、現在のコネクターの標準コネクター・プロパティーを 1 つ含んでいます。

注記

getAllStandardProperties() メソッドは、CxProperty オブジェクトの配列として標準コネクター構成プロパティーを取得します。各コネクター・プロパティー (CxProperty) オブジェクトには、単一の標準コネクター・プロパティーが含まれ、単一値、別のプロパティー、または値と子プロパティーの組み合わせを保持することができます。CxProperty クラスのメソッド (getAllChildProps() または getStringValues() など) を使用して、コネクター・プロパティー・オブジェクトから値を取得します。

参照項目

getAllConfigProp(), getAllUserProperties(), getOneConfigProp()

---

getAllUserProperties()

コネクターのすべてのコネクター固有プロパティーを取得します。

構文

```
static CxProperty[] getAllUserProperties();
```

パラメーター

なし。

戻り値

CxProperty オブジェクトの配列への参照。各配列は、現在のコネクターのコネクター固有コネクター・プロパティーを 1 つ含んでいます。

注記

getAllUserProperties() メソッドは、CxProperty オブジェクトの配列としてコネクター固有の構成プロパティーを取得します。各コネクター・プロパティー (CxProperty) オブジェクトには、単一のコネクター固有プロパティーが含まれ、単一値、別のプロパティー、または値と子プロパティーの組み合わせを保持することができます。CxProperty クラスのメソッド (getAllChildProps() または getStringValues() など) を使用して、コネクター・プロパティー・オブジェクトから値を取得します。

参照項目

getAllConfigProp(), getAllStandardProperties(), getOneConfigProp()

---

getBlankValue()

特殊な Blank 属性値を取得します。

構文

```
public static String getBlankValue();
```

パラメーター

なし。

戻り値

特殊な Blank 属性値が含まれている String オブジェクト。

getConfigProp()

リポジトリーから、コネクターに対する構成プロパティー値を取得します。

構文

```
public final static String getConfigProp(String propName);
```

パラメーター

propName 取得するプロパティーの名前です。

戻り値

プロパティー値を含む String オブジェクトです。プロパティー名が見つからない場合、メソッドは null を戻します。

注記

並列処理コネクター (ParallelProcessDegree コネクター・プロパティーに 1 より大きい値が設定されているコネクター) で、getConfigProp("ConnectorName") を呼び出すと、マスター・プロセスまたはスレーブ・プロセスのどちらに呼び出したかには関係なく、常にコネクター・エージェントのマスター・プロセスの名前がメソッドによって戻されます。

getEncoding()

コネクター・フレームワークが使用している文字エンコードを取得します。

構文

```
public String getEncoding();
```

パラメーター

なし。

戻り値

コネクター・フレームワークの文字エンコードが格納されている String オブジェクト。

注記

getEncoding() メソッドは、コネクター・フレームワークの文字エンコード (ロケールの一部) を取得します。ロケールには、言語、国 (または地域)、および文字エンコードに応じて、データの国/地域別情報を指定します。コネクター・フレームワークの文字エンコードには、コネクター・アプリケーションの文字エンコードが示されていなければなりません。コネクター・フレームワークの文字エンコードには、以下の階層が使用されています。

- リポジトリー内の CharacterEncoding コネクター構成プロパティー

WebSphere InterChange Server

ローカル構成ファイルが存在する場合、このローカル・ファイルにおける CharacterEncoding コネクター構成プロパティーの設定値が優先します。ローカル構成ファイルが存在しない場合、CharacterEncoding プロパティーの設定値は、コネクターの始動時に InterChange Server リポジトリーからダウンロードされたコネクター構成プロパティーのセットから得られます。

- Java 環境からの文字エンコード (Unicode (UCS-2))

このメソッドは、文字変換などの文字エンコード処理をコネクターが実行しなければならない場合に役に立ちます。

参照項目

getGlobalLocale()

getIgnoreValue()

特殊な Ignore 属性値の値を取得します。

構文

```
public static String getIgnoreValue();
```

パラメーター

なし。

戻り値

「Ignore」属性値が格納されている String オブジェクト。

getLocale()

コネクター・フレームワークのロケールを取得します。

構文

```
public String getLocale();
```

パラメーター

なし。

戻り値

コネクター・フレームワークのロケール設定値が格納されている String オブジェクト。

注記

getLocale() メソッドは、コネクター・フレームワークのロケールを取得します。このロケールには、言語、国 (または地域)、および文字エンコードに応じた、データの国/地域別情報が定義されています。コネクター・フレームワークのロケールには、コネクター・アプリケーションのロケールが示されていなければなりません。コネクター・フレームワークのロケールを設定する際には、以下の階層が使用されます。

- リポジトリー内の LOCALE コネクター構成プロパティー

WebSphere InterChange Server

ローカル構成ファイルが存在する場合、このローカル・ファイルにおける Locale コネクター構成プロパティーの設定値が優先します。ローカル構成ファイルが存在しない場合、Locale プロパティーの設定値は、コネクターの始動時に InterChange Server リポジトリーからダウンロードされたコネクター構成プロパティーのセットから得られます。

- Java 環境からのロケール (オペレーティング・システムからのロケール)

このメソッドは、ロケール依存の処理をコネクターが実行しなければならない場合に役に立ちます。

参照項目

(BusinessObjectInterface インターフェース内部の) createBusinessObject()、getGlobalEncoding()、getLocale()

---

getOneConfigProp()

指定された階層コネクター構成プロパティーのトップレベル・コネクター・オブジェクトを取得します。

構文

```
public static CxProperty getOneConfigProp(String propName);
```

パラメーター

propName　　取得対象の階層コネクター・プロパティーの名前です。

戻り値

指定された階層コネクター・プロパティーのトップレベルのコネクター・プロパティー・オブジェクトを含む CxProperty オブジェクトです。プロパティー名が見つからない場合、メソッドは null を戻します。

注記

getOneConfigProp() メソッドは、トップレベルのコネクター・プロパティー (CxProperty) オブジェクトを取得します。取得したこのオブジェクトから、CxProperty クラスのメソッドを使用して、コネクター・プロパティーの必要な値を取得できます。

注: コネクター構成プロパティーの値は、その値の初期化の間に、コネクターにダウンロードされます。ダウンロードされていないコネクター・プロパティーの *propName* を指定すると、getOneConfigProp() は null を戻します。

参照項目

getAllConfigProp(), getConfigProp()

---

getSupportedBusObjNames()

現在のコネクターに対してサポートされているビジネス・オブジェクトのリストを取得します。

WebSphere InterChange Server

このメソッドは、統合ブローカーが InterChange Server の場合にのみ有効です。

構文

```
public static String[] getSupportedBusObjNames()
```

パラメーター

なし。

戻り値

コネクターに対してサポートされているビジネス・オブジェクトの名前のリストを含む String 配列です。

注記

getSupportedBusObjNames() メソッドは、現在のコネクターに対してサポートされているトップレベルのビジネス・オブジェクトのリストを戻します。すなわち、コネクターが、子ビジネス・オブジェクトを含むビジネス・オブジェクトをサポートしている場合、getSupportedBusObjNames() は、親オブジェクトの名前のみをその戻りリストの中に含めます。

注: getSupportedBusObjNames() メソッドがサポートされるのは、コネクターがバージョン 4.0 以降の InterChange Server を統合ブローカーとして使用している場合のみ です。

initAndValidateAttributes()

値が設定されていない属性のうち、必須としてマークされているものを、デフォルト値で初期化します。

構文

```
public static void initAndValidateAttributes(
    BusinessObjectInterface theBusObj);
```

パラメーター

theBusObj このメソッドによって設定される属性を持ったビジネス・オブジェクトです。

戻り値

なし。

例外

BusObjSpecNameNotFoundException
: 指定されたビジネス・オブジェクトの名前が、リポジトリー内のビジネス・オブジェクト定義のいずれにも一致しない場合、スローされます。

SetDefaultFailedException
: 属性のデフォルト値が設定できず、ビジネス・オブジェクト定義内の属性に対して指定されたデフォルト値が存在しなかった場合にスローされます。

注記

initAndValidateAttributes() メソッドには、次の 2 つの目的があります。

- 属性を*初期化* し、以下の条件に当てはまる場合には、各属性についてデフォルト値を設定します。
  - UseDefaults コネクター構成プロパティーが true に設定されている場合
  - 属性の isRequired プロパティーが true に設定されている場合
  - 属性の値が現在設定されていない (特殊な Ignore 値 CxIgnore になっている) 場合
  - 属性のデフォルト値プロパティーでデフォルト値が指定されている場合
- 属性を*検証* し、以下の条件に当てはまる場合には SetDefaultFailedException 例外をスローします。
  - 属性の isRequired プロパティーが true に設定されている場合
  - 属性のデフォルト値プロパティーでそのデフォルト値が定義されていない 場合

障害が発生した場合には、initAndValidateAttributes() がデフォルト値処理を終了した後で、いくつかの属性 (デフォルト値が指定されていない属性) で値が存在しなくなります。この例外をキャッチして CxStatusConstants.FAIL を戻すように、コネクターのアプリケーション固有コンポーネントをコーディングしておくことが可能です。

initAndValidateAttributes() メソッドは、ビジネス・オブジェクトの全レベルにおいてあらゆる属性を参照し、以下のことを判別します。

- 属性が必須であるかどうか
- 属性がビジネス・オブジェクト・インスタンスに値を持っているかどうか
- UseDefaults 構成プロパティーが true に設定されているかどうか

属性が必須で、UseDefaults が true である場合、initAndValidateAttributes() は、設定されていない属性の値を、デフォルト値に設定します。
initAndValidateAttributes() に、属性値を特殊な Blank 値 (CxBlank) に設定させるために、属性のデフォルト値を「CxBlank」というストリングに設定することができます。属性にデフォルト値が指定されていない 場合、
initAndValidateAttributes() は SetDefaultFailedException 例外をスローします。

注: 属性がキーの場合、または属性の値がアプリケーションによって生成される場合は、ビジネス・オブジェクト定義のデフォルト値は使用されず、その属性の Required プロパティーには、false が設定されます。

アプリケーションで Create 操作を実行する前に必須属性の値が設定されるようにするために、initAndValidateAttributes() メソッドは通常、ビジネス・オブジェクト・ハンドラーの doVerbFor() メソッドから呼び出されます。 doVerbFor() メソッド内で、Create 動詞用に initAndValidateAttributes() メソッドを呼び出すことができます。また、Create を実行する前に、Update 動詞用としてこれを呼び出すこともできます。

initAndValidateAttributes() を使用するには、以下の操作も実行する必要があります。

- 必須属性の IsRequired プロパティーが true に設定され、必須属性のデフォルト値がデフォルト値プロパティーで指定された値になるように、ビジネス・オブジェクトを設計します。
- コネクターのコネクター固有プロパティーのリストに、UseDefaults コネクター構成プロパティーを追加します。このプロパティーを true に設定してください。

isBlankValue()

属性値が特殊な Blank (CxBlank) 属性値であるかどうかを判別します。

構文

```
public static boolean isBlankValue(Object value);
```

パラメーター

value 特殊な Blank 値と比較する値です。

戻り値

値が特殊な Blank 属性値の場合は True を戻します。それ以外の場合は、False を戻します。

---

isIgnoreValue()

属性値が特殊な Ignore (CxIgnore) 属性値であるかどうかを判別します。特殊な Ignore 属性値は、ビジネス・オブジェクトの処理中に、この属性を無視する必要があることを示します。

構文

```
public static boolean isIgnoreValue(Object value);
```

パラメーター

value 特殊な Ignore 値と比較する値です。

戻り値

値が特殊な Ignore 属性値に等しい場合は True を戻します。それ以外の場合は、False を戻します。

---

isTraceEnabled()

このレベルでのトレースが有効になっている場合に、そのトレース・レベルが、参照しているトレース・レベル以上であるかどうかを判別します。

構文

```
public final static boolean isTraceEnabled(int traceLevel);
```

パラメーター

traceLevel 検査の対象となるトレース・レベルです。

戻り値

エージェントのトレース・レベルが、渡されたトレース・レベルより大きいか等しい場合、true を戻します。

注記

このメソッドは、メッセージ生成の前に使用してください。

logMsg()

コネクターのログの宛先にメッセージを記録します。

構文

```
public final static void logMsg(String msg);
public final static void logMsg(String msg, int severity);
```

パラメーター

msg ログに記録されるメッセージ・テキストです。

severity 次のメッセージ・タイプのいずれかにします。

```
JavaConnectorUtil.XRD_WARNING
                        JavaConnectorUtil.XRD_ERROR
                        JavaConnectorUtil.XRD_FATAL
                        JavaConnectorUtil.XRD_INFO
JavaConnectorUtil.XRD_TRACE
```

戻り値

なし。

注記

logMsg() メソッドは、指定された *msg* テキストをログの宛先に送信します。コネクターのログの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「ロギング」セクションを通じて設定します。

ログ・メッセージをメッセージ・ファイルに格納して generateMsg() メソッドで抽出するよう、IBM では推奨しています。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。 generateMsg() メソッドは、logMsg() 用のメッセージ・ストリングを生成します。このメソッドによってメッセージ・ファイルから事前定義済みメッセージが取得され、テキストの書式が設定された後、生成されたメッセージ・ストリングが戻されます。

WebSphere InterChange Server

severity が XRD_ERROR または XRD_FATAL であり、コネクター構成プロパティー LogAtInterchangeEnd が設定されている場合は、エラー・メッセージがログに記録されます。電子メール通知がオンの場合は、電子メール通知が送信されます。エラーの場合に備えて電子メール通知をセットアップする方法については、IBM WebSphere InterChange Server ドキュメンテーション・セット内の「*システム管理ガイド*」を参照してください。

メッセージ・ストリングが generateMsg() で生成された場合、logMsg() で記録されたコネクター・メッセージは LogViewer を使用して見ることができます。

参照項目

generateMsg() メソッドの説明を参照してください。

---

traceWrite()

コネクターのトレース宛先にトレース・メッセージを書き込みます。

構文

```
public final static void traceWrite(int traceLevel, String msg);
```

パラメーター

traceLevel 次のトレース・レベルのいずれかにします。

```
JavaConnectorUtil.LEVEL1
JavaConnectorUtil.LEVEL2
JavaConnectorUtil.LEVEL3
JavaConnectorUtil.LEVEL4
JavaConnectorUtil.LEVEL5
```

現在のトレース・レベルが *traceLevel* より大きいか等しい場合、メソッドはトレース・メッセージを書き込みます。

注: トレース・メッセージには、トレース・レベル 0 (LEVEL0) を指定しないでください。トレース・レベル 0 は、トレースがオフになっていることを示します。したがって、LEVEL0 の *traceLevel* に関連するトレース・メッセージが、印刷されることは絶対にありません。

msg トレース・メッセージに使用されるメッセージ・テキストです。

戻り値

なし。

注記

traceWrite() メソッドは、コネクター用のユーザー独自のトレース・メッセージの書き込みに使用します。コネクターに対するトレースがオンになるのは、TraceLevel コネクター構成プロパティーがゼロ以外の値 (LEVEL0 以外の トレース・レベル定数) に設定されている場合です。

現在のトレース・レベルが *traceLevel* より大きいか等しい場合、traceWrite() メソッドは、指定された *msg* テキストをトレース宛先に送信します。コネクターのトレースの宛先の名前は、Connector Configurator の「トレース/ログ・ファイル」タブの「トレース」セクションを通じて設定します。

通常、トレース・メッセージはデバッグ時にのみ必要となるため、トレース・メッセージをメッセージ・ファイルに含めるかどうかは、以下のように開発者が任意に決定することができます。

- 英語圏外のユーザーがトレース・メッセージを参照する必要がある場合、それらのメッセージは国際化対応にする必要があります。そのため、トレース・メッセ

ージをメッセージ・ファイルに入れて、generateMsg() メソッドで抽出する必要があります。このメッセージ・ファイルは、ご使用のコネクターに固有のメッセージを含むコネクター・メッセージ・ファイルである必要があります。generateMsg() メソッドは、traceWrite() 用のメッセージ・ストリングを生成します。このメソッドによってメッセージ・ファイルから事前定義済みトレース・メッセージが取得され、テキストの書式が設定された後、生成されたメッセージ・ストリングが戻されます。

- 英語圏のユーザーのみがトレース・メッセージを参照する必要がある場合、それらのメッセージを国際化対応にする必要はありません。そのため、(英語の) トレース・メッセージを直接 traceWrite() の呼び出しに組み込むことができます。generateMsg() メソッドを使用する必要はありません。

traceWrite() によって記録されたコネクター・メッセージは、LogViewer を使用して見ることはできません。

参照項目

generateMsg() メソッドの説明を参照してください。

第 35 章 ReturnStatusDescriptor クラス

ReturnStatusDescriptor クラスにより、下位 Java コネクターは、戻り状況記述子で、エラー・メッセージや情報メッセージを返送することができます。これは、CxCommon パッケージの一部です。この戻り状況記述子にはその他の状況情報も含まれています。それらの情報は通常、要求を開始した統合ブローカーに送信される要求応答の一部として戻されます。

注: CWConnectorReturnStatusDescriptor クラスは、下位の Java コネクター・ライブラリーの ReturnStatusDescriptor クラスに関するラッパーとして使用される Java コネクター・ライブラリー・クラスです。Java コネクター開発では、普通、Java コネクター・ライブラリーを使用します。 Java コネクター・ライブラリーのクラスの詳細については、267 ページの『第 9 章 Java コネクター・ライブラリーの概要』を参照してください。

WebSphere InterChange Server

InterChange Server を使用しているビジネス・インテグレーション・システムの場合、コネクター・フレームワークによって、要求を開始したコラボレーションに戻り状況記述子が戻されます。コラボレーションは、この戻り状況記述子内の情報にアクセスし、そのサービス呼び出し要求の状況を取得することができます。

表 158 に、ReturnStatusDescriptor クラスのメソッドを要約します。

表 158. ReturnStatusDescriptor クラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
getErrorString()	オブジェクトからエラー・メッセージを取得します。	543
getStatus()	要求された操作の状況を取得します。	544
setErrorString()	オブジェクトにエラー・メッセージを設定します。	544
setStatus()	要求された操作の状況を設定します。	544

getErrorString()

戻り状況記述子からメッセージ・ストリングを取得します。メッセージは、エラー・メッセージまたは情報メッセージが許されます。

構文

```
public String getErrorString();
```

パラメーター

なし。

戻り値

統合ブローカーに対するエラー・メッセージまたは情報メッセージを含む String、または null です。

---

getStatus()

要求された操作の状況を取得します。

構文

```
public int getStatus();
```

パラメーター

なし。

戻り値

操作の状況を示す int 値です。

---

setErrorString()

ReturnStatusDescriptor オブジェクト内に、エラー・メッセージまたは情報メッセージを設定します。

構文

```
public void setErrorString(String errorStr);
```

パラメーター

errorStr　　メッセージ・ストリングです。

戻り値

なし。

---

setStatus()

要求された操作の状況を設定します。

構文

```
public void setStatus(int status);
```

パラメーター

status　　状況値です。

戻り値

なし。

第 36 章 下位の Java 例外

下位の Java コネクター・ライブラリーにおける例外とは、内部の IBM WebSphere Business Integration システム例外クラスから派生するサブクラスのことです。これらのサブクラスは、下位の Java コネクター・ライブラリーのメソッドがスローできる 1 つの*例外オブジェクト* を表します。

注: 下位の Java コネクター・ライブラリーのメソッドの参照説明では、ほとんどの場合、『例外』セクションで、メソッドがスローする例外のリストが示されます。

下位の Java コネクター・ライブラリー例外の内容は以下のとおりです。

- 『例外サブクラス』
- 『メソッド』

例外サブクラス

この章では、下位の Java コネクター・ライブラリーの例外サブクラスを列挙します。

表 159. 下位の Java コネクター・ライブラリー例外

例外名	定義
BusObjInvalidVerbException	指定された動詞が、ビジネス・オブジェクトによってサポートされていない場合にスローされます。
BusObjSpecNameNotFoundException	ビジネス・オブジェクトを作成するための仕様が発見できない場合にスローされます。
CxObjectInvalidAttrException	指定された属性のデータ型が、属性が保持するように定義されたデータ型と一致しなかった場合にスローされます。
CxObjectNoSuchAttributeException	属性の指定された位置または名前が、既存のビジネス・オブジェクト内におけるその属性の位置または名前と一致しなかった場合にスローされます。
SetDefaultFailedException	デフォルト値の設定が失敗した場合にスローされます。

メソッド

表 160 に、下位の Java コネクター・ライブラリーの例外サブクラスのメソッドを要約します。

表 160. Java 例外サブクラスのメンバー・メソッド

メンバー・メソッド	説明	ページ
getFormattedMessage()	例外メッセージのフォーマットを設定します。	545

getFormattedMessage()

例外のメッセージを特別なフォーマットに設定します。

構文

```
public String getFormattedMessage();
```

パラメーター

なし。

戻り値

フォーマット設定済みエラー・メッセージを含む String または、作成時に例外オブジェクトが、null で初期化された場合は、null となります。

注記

getFormattedMessage() が戻すメッセージのフォーマットは、次のとおりです。

[Type: *<MsgType>*][MsgID: *<msgId>*][Mesg: *<msg>*]

例外に埋め込まれているエラー・メッセージを取得するには、getFormattedMessage() メソッドを使用します。

第 5 部 付録

付録 A. コネクターの標準構成プロパティー

この付録では、WebSphere Business Integration アダプターのコネクター・コンポーネントの標準構成プロパティーについて説明しています。この付録の内容は、以下の統合ブローカーで実行されるコネクターを対象としています。

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator、WebSphere MQ Integrator Broker、および WebSphere Business Integration Message Broker (WebSphere Message Brokers (WMQI) と総称)
- WebSphere Application Server (WAS)

コネクターによっては、一部の標準プロパティーが使用されないことがあります。Connector Configurator から統合ブローカーを選択するときには、そのブローカーで稼働するアダプターのために構成が必要な標準プロパティーのリストが表示されます。

コネクター固有のプロパティーの詳細については、該当するアダプターのユーザーズ・ガイドを参照してください。

注: 本書では、ディレクトリー・パスに円記号 (¥) を使用します。UNIX システムを使用している場合は、円記号をスラッシュ (/) に置き換えてください。また、各オペレーティング・システムの規則に従ってください。

新規プロパティーと削除されたプロパティー

本リリースには、次の標準プロパティーが追加されました。

新規プロパティー

- XMLNameSpaceFormat

削除されたプロパティー

- RestartCount

標準コネクター・プロパティーの構成

アダプター・コネクターには 2 つのタイプの構成プロパティーがあります。

- 標準構成プロパティー
- コネクター固有の構成プロパティー

このセクションでは、標準構成プロパティーについて説明します。コネクター固有の構成プロパティーについては、該当するアダプターのユーザーズ・ガイドを参照してください。

Connector Configurator の使用

Connector Configurator からコネクター・プロパティーを構成します。Connector Configurator には、System Manager からアクセスします。Connector Configurator の使用法の詳細については、付録の『Connector Configurator』を参照してください。

注: Connector Configurator と System Manager は、Windows システム上でのみ動作します。コネクターを UNIX システム上で稼働している場合でも、これらのツールがインストールされた Windows マシンが必要です。UNIX 上で動作するコネクターのコネクター・プロパティーを設定する場合は、Windows マシン上で System Manager を起動し、UNIX の統合ブローカーに接続してから、コネクター用の Connector Configurator を開く必要があります。

プロパティー値の設定と更新

プロパティー・フィールドのデフォルトの長さは 255 文字です。

コネクターは、以下の順序に従ってプロパティーの値を決定します (最も番号の大きい項目が他の項目よりも優先されます)。

1. デフォルト
2. リポジトリー (WebSphere InterChange Server が統合ブローカーである場合のみ)
3. ローカル構成ファイル
4. コマンド行

コネクターは、始動時に構成値を取得します。実行時セッション中に 1 つ以上のコネクター・プロパティーの値を変更する場合は、プロパティーの**更新メソッド**によって、変更を有効にする方法が決定されます。標準コネクター・プロパティーには、以下の 4 種類の更新メソッドがあります。

- **動的**
  変更を System Manager に保管すると、変更が即時に有効になります。コネクターが System Manager から独立してスタンドアロン・モードで稼働している場合 (例えば、いずれかの WebSphere Message Brokers と連携している場合) は、構成ファイルでのみプロパティーを変更できます。この場合、動的更新は実行できません。
- **エージェント再始動 (ICS のみ)**
  アプリケーション固有のコンポーネントを停止して再始動しなければ、変更が有効になりません。
- **コンポーネント再始動**
  System Manager でコネクターを停止してから再始動しなければ、変更が有効になりません。アプリケーション固有コンポーネントまたは統合ブローカーを停止、再始動する必要はありません。
- **サーバー再始動**
  アプリケーション固有のコンポーネントおよび統合ブローカーを停止して再始動しなければ、変更が有効になりません。

特定のプロパティーの更新方法を確認するには、「Connector Configurator」ウィンドウ内の**「更新メソッド」**列を参照するか、次に示す551 ページの表 161 の「更新メソッド」列を参照してください。

標準プロパティーの要約

表 161 は、標準コネクター構成プロパティーの早見表です。標準プロパティーの依存関係は RepositoryDirectory に基づいているため、コネクターによっては使用されないプロパティーがあり、使用する統合ブローカーによってプロパティーの設定が異なる可能性があります。

コネクターを実行する前に、これらのプロパティーの一部の値を設定する必要があります。各プロパティーの詳細については、次のセクションを参照してください。

注: 表 161 の「注」列にある「Repository Directory は REMOTE」という句は、ブローカーが InterChange Server であることを示します。ブローカーが WMQI または WAS の場合には、リポジトリー・ディレクトリーは LOCAL に設定されます。

表 161. 標準構成プロパティーの要約

プロパティー名	指定可能な値	デフォルト値	更新メソッド	注
AdminInQueue	有効な JMS キュー名	*CONNECTORNAME* /ADMININQUEUE	コンポーネント再始動	Delivery Transport は JMS
AdminOutQueue	有効な JMS キュー名	*CONNECTORNAME*/ADMINOUTQUEUE	コンポーネント再始動	Delivery Transport は JMS
AgentConnections	1 から 4	1	コンポーネント再始動	Delivery Transport は MQ および IDL: Repository Directory は <REMOTE> (ブローカーは ICS)
AgentTraceLevel	0 から 5	0	動的	
ApplicationName	アプリケーション名	コネクター・アプリケーション名として指定された値	コンポーネント再始動	
BrokerType	ICS、WMQI、WAS		コンポーネント再始動	
CharacterEncoding	ascii7、ascii8、SJIS、Cp949、GBK、Big5、Cp297、Cp273、Cp280、Cp284、Cp037、Cp437 **注:** これは、サポートされる値の一部です。	ascii7	コンポーネント再始動	
ConcurrentEventTriggeredFlows	1 から 32,767	1	コンポーネント再始動	Repository Directory は <REMOTE> (ブローカーは ICS)
ContainerManagedEvents	値なし、または JMS	値なし	コンポーネント再始動	Delivery Transport は JMS
ControllerStoreAndForwardMode	true または false	true	動的	Repository Directory は <REMOTE> (ブローカーは ICS)
ControllerTraceLevel	0 から 5	0	動的	Repository Directory は <REMOTE> (ブローカーは ICS)

表 161. 標準構成プロパティーの要約 (続き)

プロパティー名	指定可能な値	デフォルト値	更新メソッド	注
DeliveryQueue		*CONNECTORNAME*/DELIVERYQUEUE	コンポーネント再始動	JMS トランスポートのみ
DeliveryTransport	MQ、IDL、または JMS	JMS	コンポーネント再始動	Repository Directory がローカルの場合は、値は JMS のみ
DuplicateEventElimination	true または false	false	コンポーネント再始動	JMS トランスポートのみ、Container Managed Events は <NONE> でなければならない
FaultQueue		*CONNECTORNAME*/FAULTQUEUE	コンポーネント再始動	JMS トランスポートのみ
jms.FactoryClassName	CxCommon.Messaging.jms.IBMMQSeriesFactory または CxCommon.Messaging.jms.SonicMQFactory または任意の Java クラス名	CxCommon.Messaging.jms.IBMMQSeriesFactory	コンポーネント再始動	JMS トランスポートのみ
jms.MessageBrokerName	FactoryClassName が IBM の場合は crossworlds.queue.manager を使用。FactoryClassName が Sonicの場合 localhost:2506 を使用。	crossworlds.queue.manager	コンポーネント再始動	JMS トランスポートのみ
jms.NumConcurrentRequests	正整数	10	コンポーネント再始動	JMS トランスポートのみ
jms.Password	任意の有効なパスワード		コンポーネント再始動	JMS トランスポートのみ
jms.UserName	任意の有効な名前		コンポーネント再始動	JMS トランスポートのみ
JvmMaxHeapSize	ヒープ・サイズ (メガバイト単位)	128m	コンポーネント再始動	Repository Directory は <REMOTE> (ブローカーは ICS)
JvmMaxNativeStackSize	スタックのサイズ (キロバイト単位)	128k	コンポーネント再始動	Repository Directory は <REMOTE> (ブローカーは ICS)
JvmMinHeapSize	ヒープ・サイズ (メガバイト単位)	1m	コンポーネント再始動	Repository Directory は <REMOTE> (ブローカーは ICS)
ListenerConcurrency	1 から 100	1	コンポーネント再始動	Delivery Transport は MQ でなければならない

表 161. 標準構成プロパティーの要約 (続き)

プロパティー名	指定可能な値	デフォルト値	更新メソッド	注
Locale	en_US、ja_JP、ko_KR、zh_CN、zh_TW、fr_FR、de_DE、it_IT、es_ES、pt_BR **注:** これは、サポートされるロケールの一部です。	en_US	コンポーネント再始動	
LogAtInterchangeEnd	true または false	false	コンポーネント再始動	Repository Directory は <REMOTE> でなければならない (ブローカーは ICS)
MaxEventCapacity	1 から 2147483647	2147483647	動的	Repository Directory は <REMOTE> でなければならない (ブローカーは ICS)
MessageFileName	パスまたはファイル名	CONNECTORNAMEConnector.txt	コンポーネント再始動	
MonitorQueue	任意の有効なキュー名	*CONNECTORNAME*/MONITORQUEUE	コンポーネント再始動	JMS トランスポートのみ: DuplicateEvent Elimination は true でなければならない
OADAutoRestartAgent	true または false	false	動的	Repository Directory は <REMOTE> でなければならない (ブローカーは ICS)
OADMaxNumRetry	正数	1000	動的	Repository Directory は <REMOTE> でなければならない (ブローカーは ICS)
OADRetryTimeInterval	正数 (単位: 分)	10	動的	Repository Directory は <REMOTE> でなければならない (ブローカーは ICS)
PollEndTime	HH:MM	HH:MM	コンポーネント再始動	

表 161. 標準構成プロパティーの要約 (続き)

プロパティー名	指定可能な値	デフォルト値	更新メソッド	注
PollFrequency	正整数 (単位: ミリ秒) no (ポーリングを使用不可にする) key (コネクターのコマンド・プロンプト・ウィンドウで文字 p が入力された場合にのみポーリングする)	10000	動的	
PollQuantity	1 から 500	1	エージェント再始動	JMS トランスポートのみ: Container Managed Events を指定
PollStartTime	HH:MM (HH は 0 から 23、MM は 0 から 59)	HH:MM	コンポーネント再始動	
RepositoryDirectory	メタデータ・リポジトリーの場所		エージェント再始動	ICS の場合は <REMOTE> に設定する。WebSphere MQ Message Brokers および WAS の場合は C:¥crossworlds ¥repository に設定
RequestQueue	有効な JMS キュー名	*CONNECTORNAME*/REQUESTQUEUE	コンポーネント再始動	Delivery Transport は JMS
ResponseQueue	有効な JMS キュー名	*CONNECTORNAME*/RESPONSEQUEUE	コンポーネント再始動	Delivery Transport が JMS の場合: Repository Directory が <REMOTE> の場合のみ必要
RestartRetryCount	0 から 99	3	動的	
RestartRetryInterval	適切な正数 (単位: 分): 1 から 2147483547	1	動的	
RHF2MessageDomain	mrm、xml	mrm	コンポーネント再始動	Delivery Transport が JMS であり、かつ WireFormat が CwXML である
SourceQueue	有効な WebSphere MQ 名	*CONNECTORNAME*/SOURCEQUEUE	エージェント再始動	Delivery Transport が JMS であり、かつ Container Managed Events が指定されている場合のみ
SynchronousRequestQueue		*CONNECTORNAME*/ SYNCHRONOUSREQUESTQUEUE	コンポーネント再始動	Delivery Transport は JMS
SynchronousRequestTimeout	0 以上の任意の数値 (ミリ秒)	0	コンポーネント再始動	Delivery Transport は JMS

表 161. 標準構成プロパティーの要約 (続き)

プロパティー名	指定可能な値	デフォルト値	更新メソッド	注
SynchronousResponseQueue		*CONNECTORNAME*/SYNCHRONOUSRESPONSEQUEUE	コンポーネント再始動	Delivery Transport は JMS
WireFormat	CwXML、CwBO	CwXML	エージェント再始動	Repository Directory が <REMOTE> でない場合は CwXML。Repository Directory が <REMOTE> であれば CwBO
WsifSynchronousRequest Timeout	0 以上の任意の数値 (ミリ秒)	0	コンポーネント再始動	WAS のみ
XMLNameSpaceFormat	short、long	short	エージェント再始動	WebSphere MQ Message Brokers および WAS のみ

標準構成プロパティー

このセクションでは、各標準コネクター構成プロパティーの定義を示します。

AdminInQueue

統合ブローカーからコネクターへ管理メッセージが送信されるときに使用されるキューです。

デフォルト値は CONNECTORNAME/ADMININQUEUE です。

AdminOutQueue

コネクターから統合ブローカーへ管理メッセージが送信されるときに使用されるキューです。

デフォルト値は CONNECTORNAME/ADMINOUTQUEUE です。

AgentConnections

RepositoryDirectory が <REMOTE> の場合のみ適用できます。

AgentConnections プロパティーは、orb.init[] により開かれる ORB (オブジェクト・リクエスト・ブローカー) 接続の数を制御します。

このプロパティーのデフォルト値は 1 に設定されます。必要に応じてこの値を変更できます。

AgentTraceLevel

アプリケーション固有のコンポーネントのトレース・メッセージのレベルです。デフォルト値は 0 です。コネクターは、設定されたトレース・レベル以下の該当するトレース・メッセージをすべてデリバリーします。

ApplicationName

コネクターのアプリケーションを一意的に特定する名前です。この名前は、システム管理者が WebSphere Business Integration システム環境をモニターするために使用されます。コネクターを実行する前に、このプロパティーに値を指定する必要があります。

BrokerType

使用する統合ブローカー・タイプを指定します。オプションは、ICS、WebSphere Message Brokers (WMQI、WMQIB、または WBIMB)、または WAS です。

CharacterEncoding

文字 (アルファベットの文字、数値表現、句読記号など) から数値へのマッピングに使用する文字コード・セットを指定します。

注: Java ベースのコネクターでは、このプロパティーは使用しません。C++ ベースのコネクターでは、現在、このプロパティーに `ascii7` という値が使用されています。

デフォルトでは、ドロップダウン・リストには、サポートされる文字エンコードの一部のみが表示されます。ドロップダウン・リストに、サポートされる他の値を追加するには、製品ディレクトリーにある `¥Data¥Std¥stdConnProps.xml` ファイルを手動で変更する必要があります。詳細については、Connector Configurator に関する付録を参照してください。

ConcurrentEventTriggeredFlows

`RepositoryDirectory` が <REMOTE> の場合のみ適用できます。

コネクターがイベントのデリバリー時に並行処理できるビジネス・オブジェクトの数を決定します。この属性の値を、並行してマップおよびデリバリーできるビジネス・オブジェクトの数に設定します。例えば、この属性の値を `5` に設定すると、5 個のビジネス・オブジェクトが並行して処理されます。デフォルト値は `1` です。

このプロパティーを `1` よりも大きい値に設定すると、ソース・アプリケーションのコネクターが、複数のイベント・ビジネス・オブジェクトを同時にマップして、複数のコラボレーション・インスタンスにそれらのビジネス・オブジェクトを同時にデリバリーすることができます。これにより、統合ブローカーへのビジネス・オブジェクトのデリバリーにかかる時間、特にビジネス・オブジェクトが複雑なマップを使用している場合のデリバリー時間が短縮されます。ビジネス・オブジェクトのコラボレーションに到達する速度を増大させると、システム全体のパフォーマンスを向上させることができます。

ソース・アプリケーションから宛先アプリケーションまでのフロー全体に並行処理を実装するには、次のようにする必要があります。

- `Maximum number of concurrent events` プロパティーの値を増加して、コラボレーションが複数のスレッドを使用できるように構成します。
- 宛先アプリケーションのアプリケーション固有コンポーネントが複数の要求を並行して実行できることを確認します。つまり、このコンポーネントがマルチスレッド化されているか、またはコネクター・エージェント並列処理を使用でき、複

数プロセスに対応するよう構成されている必要があります。Parallel Process Degree 構成プロパティーに、1 より大きい値を設定します。

ConcurrentEventTriggeredFlows プロパティーは、順次に実行される単一スレッド処理であるコネクターのポーリングでは無効です。

ContainerManagedEvents

このプロパティーにより、JMS イベント・ストアを使用する JMS 対応コネクターが、保証付きイベント・デリバリーを提供できるようになります。保証付きイベント・デリバリーでは、イベントはソース・キューから除去され、単一 JMS トランザクションとして宛先キューに配置されます。

デフォルト値はありません。

ContainerManagedEvents を JMS に設定した場合には、保証付きイベント・デリバリーを使用できるように次のプロパティーも構成する必要があります。

- PollQuantity = 1 から 500
- SourceQueue = /SOURCEQUEUE

また、MimeType、DHClass (データ・ハンドラー・クラス)、および DataHandlerConfigMOName (オプションのメタオブジェクト名) プロパティーを設定したデータ・ハンドラーも構成する必要があります。これらのプロパティーの値を設定するには、Connector Configurator の**「データ・ハンドラー」**タブを使用します。

これらのプロパティーはアダプター固有ですが、**例**の値は次のようになります。

- MimeType = text¥xml
- DHClass = com.crossworlds.DataHandlers.text.xml
- DataHandlerConfigMOName = MO_DataHandler_Default

「データ・ハンドラー」タブのこれらの値のフィールドは、ContainerManagedEvents を JMS に設定した場合にのみ表示されます。

注: ContainerManagedEvents を JMS に設定した場合、コネクターはその pollForEvents() メソッドを呼び出さなくなる ため、そのメソッドの機能は使用できなくなります。

このプロパティーは、DeliveryTransport プロパティーが値 JMS に設定されている場合にのみ表示されます。

ControllerStoreAndForwardMode

RepositoryDirectory が <REMOTE> の場合のみ適用できます。

宛先側のアプリケーション固有のコンポーネントが使用不可であることをコネクター・コントローラーが検出した場合に、コネクター・コントローラーが実行する動作を設定します。

このプロパティーを true に設定した場合、イベントが ICS に到達したときに宛先側のアプリケーション固有のコンポーネントが使用不可であれば、コネクター・コ

ントローラーはそのアプリケーション固有のコンポーネントへの要求をブロックします。アプリケーション固有のコンポーネントが作動可能になると、コネクター・コントローラーはアプリケーション固有のコンポーネントにその要求を転送します。

ただし、コネクター・コントローラーが宛先側のアプリケーション固有のコンポーネントにサービス呼び出し要求を**転送した後で**このコンポーネントが使用不可になった場合、コネクター・コントローラーはその要求を失敗させます。

このプロパティーを false に設定した場合、コネクター・コントローラーは、宛先側のアプリケーション固有のコンポーネントが使用不可であることを検出すると、ただちにすべてのサービス呼び出し要求を失敗させます。

デフォルト値は true です。

ControllerTraceLevel

RepositoryDirectory が <REMOTE> の場合のみ適用できます。

コネクター・コントローラーのトレース・メッセージのレベルです。デフォルト値は 0 です。

DeliveryQueue

DeliveryTransport が JMS の場合のみ適用できます。

コネクターから統合ブローカーへビジネス・オブジェクトが送信されるときに使用されるキューです。

デフォルト値は CONNECTORNAME/DELIVERYQUEUE です。

DeliveryTransport

イベントのデリバリーのためのトランスポート機構を指定します。指定可能な値は、WebSphere MQ の MQ、CORBA IIOP の IDL、Java Messaging Service の JMS です。

- RepositoryDirectory がリモートの場合は、DeliveryTransport プロパティーの指定可能な値は MQ、IDL、または JMS であり、デフォルトは IDL になります。
- RepositoryDirectory がローカル・ディレクトリーの場合、指定可能な値は JMS のみです。

DeliveryTransport プロパティーに指定されている値が、MQ または IDL である場合、コネクターは、CORBA IIOP を使用してサービス呼び出し要求と管理メッセージを送信します。

WebSphere MQ および IDL

イベントのデリバリー・トランスポートには、IDL ではなく WebSphere MQ を使用してください (1 種類の製品だけを使用する必要がある場合を除きます)。WebSphere MQ が IDL よりも優れている点は以下のとおりです。

- 非同期 (ASYNC) 通信:
  WebSphere MQ を使用すると、アプリケーション固有のコンポーネントは、サーバーが利用不能である場合でも、イベントをポーリングして永続的に格納することができます。
- サーバー・サイド・パフォーマンス:
  WebSphere MQ を使用すると、サーバー・サイドのパフォーマンスが向上します。最適化モードでは、WebSphere MQ はイベントへのポインターのみをリポジトリー・データベースに格納するので、実際のイベントは WebSphere MQ キュー内に残ります。これにより、サイズが大きい可能性のあるイベントをリポジトリー・データベースに書き込む必要がありません。
- エージェント・サイド・パフォーマンス:
  WebSphere MQ を使用すると、アプリケーション固有のコンポーネント側のパフォーマンスが向上します。WebSphere MQ を使用すると、コネクターのポーリング・スレッドは、イベントを選出した後、コネクターのキューにそのイベントを入れ、次のイベントを選出します。この方法は IDL よりも高速で、IDL の場合、コネクターのポーリング・スレッドは、イベントを選出した後、ネットワーク経由でサーバー・プロセスにアクセスしてそのイベントをリポジトリー・データベースに永続的に格納してから、次のイベントを選出する必要があります。

JMS

Java Messaging Service (JMS) を使用しての、コネクターとクライアントのコネクター・フレームワークとの間の通信を可能にします。

JMS をデリバリー・トランスポートとして選択した場合は、`jms.MessageBrokerName`、`jms.FactoryClassName`、`jms.Password`、`jms.UserName` などの追加の JMS プロパティーが Connector Configurator 内に表示されます。このうち最初の 2 つは、このトランスポートの必須プロパティーです。

重要: 以下の環境では、コネクターに JMS トランスポート機構を使用すると、メモリー制限が発生することもあります。

- AIX 5.0
- WebSphere MQ 5.3.0.1
- ICS が統合ブローカーの場合

この環境では、WebSphere MQ クライアント内でメモリーが使用されるため、(サーバー側の) コネクター・コントローラーと (クライアント側の) コネクターの両方を始動するのは困難な場合があります。ご使用のシステムのプロセス・ヒープ・サイズが 768M 未満である場合には、次のように設定することをお勧めします。

- `CWSharedEnv.sh` スクリプト内で `LDR_CNTRL` 環境変数を設定する。

  このスクリプトは、製品ディレクトリー配下の `¥bin` ディレクトリーにあります。テキスト・エディターを使用して、`CWSharedEnv.sh` スクリプトの最初の行として次の行を追加します。

  ```
  export LDR_CNTRL=MAXDATA=0x30000000
  ```

この行は、ヒープ・メモリーの使用量を最大 768 MB (3 セグメント * 256 MB) に制限します。プロセス・メモリーがこの制限値を超えると、ページ・スワッピングが発生し、システムのパフォーマンスに悪影響を与える場合があります。

- IPCCBaseAddress プロパティーの値を 11 または 12 に設定する。このプロパティーの詳細については、「*システム・インストール・ガイド (UNIX 版)*」を参照してください。

DuplicateEventElimination

このプロパティーを true に設定すると、JMS 対応コネクターによるデリバリー・キューへの重複イベントのデリバリーが防止されます。この機能を使用するには、コネクターに対し、アプリケーション固有のコード内でビジネス・オブジェクトの **ObjectEventId** 属性として一意のイベント ID が設定されている必要があります。これはコネクター開発時に設定されます。

このプロパティーは、false に設定することもできます。

注: DuplicateEventElimination を true に設定する際は、MonitorQueue プロパティーを構成して保証付きイベント・デリバリーを使用可能にする必要があります。

FaultQueue

コネクターでメッセージを処理中にエラーが発生すると、コネクターは、そのメッセージを状況表示および問題説明とともにこのプロパティーに指定されているキューに移動します。

デフォルト値は CONNECTORNAME/FAULTQUEUE です。

JvmMaxHeapSize

エージェントの最大ヒープ・サイズ (メガバイト単位)。このプロパティーは、RepositoryDirectory の値が <REMOTE> の場合のみ適用できます。

デフォルト値は 128M です。

JvmMaxNativeStackSize

エージェントの最大ネイティブ・スタック・サイズ (キロバイト単位)。このプロパティーは、RepositoryDirectory の値が <REMOTE> の場合のみ適用できます。

デフォルト値は 128K です。

JvmMinHeapSize

エージェントの最小ヒープ・サイズ (メガバイト単位)。このプロパティーは、RepositoryDirectory の値が <REMOTE> の場合のみ適用できます。

デフォルト値は 1M です。

jms.FactoryClassName

JMS プロバイダーのためにインスタンスを生成するクラス名を指定します。JMS をデリバリー・トランスポート機構 (DeliveryTransport) として選択する際は、このコネクター・プロパティーを必ず 設定してください。

デフォルト値は CxCommon.Messaging.jms.IBMMQSeriesFactory です。

jms.MessageBrokerName

JMS プロバイダーのために使用するブローカー名を指定します。JMS をデリバリー・トランスポート機構 (DeliveryTransport) として選択する際は、このコネクター・プロパティーを必ず 設定してください。

デフォルト値は crossworlds.queue.manager です。ローカル・メッセージ・ブローカーに接続する場合は、デフォルト値を使用します。

リモート・メッセージ・ブローカーに接続すると、このプロパティーは次の (必須) 値をとります。
QueueMgrName:<Channel>:<HostName>:<PortNumber>
各変数の意味は以下のとおりです。
QueueMgrName: キュー・マネージャー名です。
Channel: クライアントが使用するチャネルです。
HostName: キュー・マネージャーの配置先のマシン名です。
PortNumber: キュー・マネージャーが listen に使用するポートの番号です。

例えば、次のようになります。
jms.MessageBrokerName = WBIMB.Queue.Manager:CHANNEL1:RemoteMachine:1456

jms.NumConcurrentRequests

コネクターに対して同時に送信することができる並行サービス呼び出し要求の数 (最大値) を指定します。この最大値に達した場合、新規のサービス呼び出し要求はブロックされ、既存のいずれかの要求が完了した後で処理されます。

デフォルト値は 10 です。

jms.Password

JMS プロバイダーのためのパスワードを指定します。このプロパティーの値はオプションです。

デフォルトはありません。

jms.UserName

JMS プロバイダーのためのユーザー名を指定します。このプロパティーの値はオプションです。

デフォルトはありません。

ListenerConcurrency

このプロパティーは、統合ブローカーとして ICS を使用する場合の MQ Listener でのマルチスレッド化をサポートしています。このプロパティーにより、データベースへの複数イベントの書き込み操作をバッチ処理できるので、システム・パフォーマンスが向上します。デフォルト値は 1 です。

このプロパティーは、MQ トランスポートを使用するコネクターにのみ適用されます。DeliveryTransport プロパティーには MQ を設定してください。

Locale

言語コード、国または地域、および、希望する場合には、関連した文字コード・セットを指定します。このプロパティーの値は、データの照合やソート順、日付と時刻の形式、通貨記号などの国/地域別情報を決定します。

ロケール名は次のような形式になります。

ll_TT.codeset

ここで、以下のように説明されます。

ll	2 文字の言語コード (普通は小文字)
TT	2 文字の国または地域コード (普通は大文字)
codeset	関連文字コード・セットの名前。名前のこの部分は、通常、オプションです。

デフォルトでは、ドロップダウン・リストには、サポートされるロケールの一部のみが表示されます。ドロップダウン・リストに、サポートされる他の値を追加するには、製品ディレクトリーにある ¥Data¥Std¥stdConnProps.xml ファイルを手動で変更する必要があります。詳細については、Connector Configurator に関する付録を参照してください。

デフォルト値は en_US です。コネクターがグローバル化に対応していない場合、このプロパティーの有効な値は en_US のみです。特定のコネクターがグローバル化に対応しているかどうかを判別するには、以下の Web サイトにあるコネクターのバージョン・リストを参照してください。

http://www.ibm.com/software/websphere/wbiadapters/infocenter、または
http://www.ibm.com/websphere/integration/wicserver/infocenter

LogAtInterchangeEnd

RepositoryDirectory が <REMOTE> の場合のみ適用できます。

統合ブローカーのログ宛先にエラーを記録するかどうかを指定します。ブローカーのログ宛先にログを記録すると、電子メール通知もオンになります。これにより、エラーまたは致命的エラーが発生すると、InterchangeSystem.cfg ファイルに指定された MESSAGE_RECIPIENT に対する電子メール・メッセージが生成されます。

例えば、LogAtInterChangeEnd を true に設定した場合にコネクターからアプリケーションへの接続が失われると、指定されたメッセージ宛先に、電子メール・メッセージが送信されます。デフォルト値は false です。

MaxEventCapacity

コントローラー・バッファー内のイベントの最大数。このプロパティーはフロー制御が使用し、RepositoryDirectory プロパティーの値が <REMOTE> の場合のみ適用できます。

値は 1 から 2147483647 の間の正整数です。デフォルト値は 2147483647 です。

MessageFileName

コネクター・メッセージ・ファイルの名前です。メッセージ・ファイルの標準位置は、製品ディレクトリーの ¥connectors¥messages です。メッセージ・ファイルが標準位置に格納されていない場合は、メッセージ・ファイル名を絶対パスで指定します。

コネクター・メッセージ・ファイルが存在しない場合は、コネクターは InterchangeSystem.txt をメッセージ・ファイルとして使用します。このファイルは、製品ディレクトリーに格納されています。

注: 特定のコネクターについて、コネクター独自のメッセージ・ファイルがあるかどうかを判別するには、該当するアダプターのユーザーズ・ガイドを参照してください。

MonitorQueue

コネクターが重複イベントをモニターするために使用する論理キューです。このプロパティーは、DeliveryTransport プロパティー値が JMS であり、かつ DuplicateEventElimination が TRUE に設定されている場合にのみ使用されます。

デフォルト値は CONNECTORNAME/MONITORQUEUE です。

OADAutoRestartAgent

RepositoryDirectory が <REMOTE> の場合のみ有効です。

コネクターの使用する再始動機能が自動かリモートかを指定します。この機能は、MQ により起動される Object Activation Daemon (OAD) を使用して、異常シャットダウン後のコネクターの再始動や System Monitor からのリモート・コネクターの始動を行います。

自動およびリモートの再始動機能を使用可能にするには、このプロパティーを true に設定する必要があります。MQ により起動される OAD 機能の構成方法については、「*システム・インストール・ガイド (Windows 版)*」または「*システム・インストール・ガイド (UNIX 版)*」を参照してください。

デフォルト値は false です。

OADMaxNumRetry

RepositoryDirectory が <REMOTE> の場合のみ有効です。

異常シャットダウンの後で MQ により起動される OAD がコネクターの再始動を自動的に試行する回数の最大数を指定します。OADAutoRestartAgent プロパティーを有効にするには、値を true に設定する必要があります。

デフォルト値は 1000 です。

OADRetryTimeInterval

RepositoryDirectory が <REMOTE> の場合のみ有効です。

MQ により起動される OAD の再試行間隔の分数を指定します。コネクター・エージェントがこの再試行間隔の間に再始動しないと、コネクター・コントローラーが OAD にコネクター・エージェントの再始動を再度要求します。OAD はこの再試行処理を OADMaxNumRetry プロパティーで指定されている回数だけ繰り返します。OADAutoRestartAgent プロパティーを有効にするには、値を true に設定する必要があります。

デフォルト値は 10 です。

PollEndTime

イベント・キューのポーリングを停止する時刻です。形式は HH:MM です。ここで、*HH* は 0 から 23 時を表し、*MM* は 0 から 59 分を表します。

このプロパティーには必ず有効な値を指定してください。デフォルト値は HH:MM ですが、この値は必ず変更する必要があります。

PollFrequency

これは、前回のポーリングの終了から次のポーリングの開始までの間の間隔です。PollFrequency は、あるポーリング・アクションの終了から次のポーリング・アクションの開始までの時間をミリ秒単位で指定します。これはポーリング・アクション間の間隔ではありません。この論理を次に説明します。

- ポーリングし、PollQuantity の値により指定される数のオブジェクトを取得します。
- これらのオブジェクトを処理します。一部のアダプターでは、これは個別のスレッドで部分的に実行されます。これにより、次のポーリング・アクションまで処理が非同期に実行されます。
- PollFrequency で指定された間隔にわたって遅延します。
- このサイクルを繰り返します。

PollFrequency は以下の値のいずれかに設定します。

- ポーリング・アクション間のミリ秒数 (整数)。
- ワード key。コネクターは、コネクターのコマンド・プロンプト・ウィンドウで文字 p が入力されたときにのみポーリングを実行します。このワードは小文字で入力します。
- ワード no。コネクターはポーリングを実行しません。このワードは小文字で入力します。

デフォルト値は 10000 です。

重要: 一部のコネクターでは、このプロパティーの使用が制限されています。このようなコネクターが存在する場合には、アダプターのインストールと構成に関する章で制約事項が説明されています。

PollQuantity

コネクターがアプリケーションからポーリングする項目の数を指定します。アダプターにコネクター固有のポーリング数設定プロパティーがある場合、標準プロパティーの値は、このコネクター固有のプロパティーの設定値によりオーバーライドされます。

電子メール・メッセージもイベントと見なされます。コネクターは、電子メールに関するポーリングを受けたときには次のように動作します。

コネクターは、1 回目のポーリングを受けると、メッセージの本文を選出します。これは、本文が添付とも見なされるからです。本文の MIME タイプにはデータ・ハンドラーが指定されていないので、コネクターは本文を無視します。
コネクターは PO の最初の添付を処理します。この添付の MIME タイプには対応するデータ・ハンドラーがあるので、コネクターはビジネス・オブジェクトを Visual Test Connector に送信します。
2 回目のポーリングを受けると、コネクターは PO の 2 番目の添付を処理します。この添付の MIME タイプには対応するデータ・ハンドラーがあるので、コネクターはビジネス・オブジェクトを Visual Test Connector に送信します。
一度受け入れると、PO の 3 番目の添付が届きます。

PollStartTime

イベント・キューのポーリングを開始する時刻です。形式は *HH:MM* です。ここで、*HH* は 0 から 23 時を表し、*MM* は 0 から 59 分を表します。

このプロパティーには必ず有効な値を指定してください。デフォルト値は `HH:MM` ですが、この値は必ず変更する必要があります。

RequestQueue

統合ブローカーが、ビジネス・オブジェクトをコネクターに送信するときに使用されるキューです。

デフォルト値は `CONNECTOR/REQUESTQUEUE` です。

RepositoryDirectory

コネクターが XML スキーマ文書を読み取るリポジトリーの場所です。この XML スキーマ文書には、ビジネス・オブジェクト定義のメタデータが含まれています。

統合ブローカーが ICS の場合はこの値を <REMOTE> に設定する必要があります。これは、コネクターが InterChange Server リポジトリーからこの情報を取得するためです。

統合ブローカーが WebSphere Message Broker または WAS の場合には、この値を <*local directory*> に設定する必要があります。

ResponseQueue

DeliveryTransport が JMS の場合のみ適用でき、RepositoryDirectory が <REMOTE> の場合のみ必要です。

JMS 応答キューを指定します。JMS 応答キューは、応答メッセージをコネクター・フレームワークから統合ブローカーへデリバリーします。統合ブローカーが ICS の場合、サーバーは要求を送信し、JMS 応答キューの応答メッセージを待ちます。

RestartRetryCount

コネクターによるコネクター自体の再始動の試行回数を指定します。このプロパティーを並列コネクターに対して使用する場合、コネクターのマスター側のアプリケーション固有のコンポーネントがスレーブ側のアプリケーション固有のコンポーネントの再始動を試行する回数が指定されます。

デフォルト値は 3 です。

RestartRetryInterval

コネクターによるコネクター自体の再始動の試行間隔を分単位で指定します。このプロパティーを並列コネクターに対して使用する場合、コネクターのマスター側のアプリケーション固有のコンポーネントがスレーブ側のアプリケーション固有のコンポーネントの再始動を試行する間隔が指定されます。指定できる値の範囲は 1 から 2147483647 です。

デフォルト値は 1 です。

RHF2MessageDomain

WebSphere Message Brokers および WAS でのみ使用されます。

このプロパティーにより、JMS ヘッダーのドメイン名フィールドの値を構成できます。JMS トランスポートを介してデータを WMQI に送信するときに、アダプター・フレームワークにより JMS ヘッダー情報、ドメイン名、および固定値 mrm が書き込まれます。この構成可能なドメイン名により、ユーザーは WMQI ブローカーによるメッセージ・データの処理方法を追跡できます。

サンプル・ヘッダーを以下に示します。

```
<mcd><Msd>mrm</Msd><Set>3</Set><Type>
Retek_POPhyDesc</Type><Fmt>CwXML</Fmt></mcd>
```

デフォルト値は mrm ですが、このプロパティーには xml も設定できます。このプロパティーは、DeliveryTransport が JMS に設定されており、かつ WireFormat が CwXML に設定されている場合にのみ表示されます。

SourceQueue

DeliveryTransport が JMS で、ContainerManagedEvents が指定されている場合のみ適用できます。

JMS イベント・ストアを使用する JMS 対応コネクターでの保証付きイベント・デリバリーをサポートするコネクター・フレームワークに、JMS ソース・キューを指定します。詳細については、 557 ページの『ContainerManagedEvents』を参照してください。

デフォルト値は CONNECTOR/SOURCEQUEUE です。

SynchronousRequestQueue

DeliveryTransport が JMS の場合のみ適用できます。

同期応答を要求する要求メッセージを、コネクター・フレームワークからブローカーに配信します。このキューは、コネクターが同期実行を使用する場合にのみ必要です。同期実行の場合、コネクター・フレームワークは、SynchronousRequestQueue にメッセージを送信し、SynchronousResponseQueue でブローカーから戻される応答を待機します。コネクターに送信される応答メッセージには、元のメッセージの ID を指定する相関 ID が含まれています。

デフォルト値は CONNECTORNAME/SYNCHRONOUSREQUESTQUEUE です。

SynchronousResponseQueue

DeliveryTransport が JMS の場合のみ適用できます。

同期要求に対する応答として送信される応答メッセージを、ブローカーからコネクター・フレームワークに配信します。このキューは、コネクターが同期実行を使用する場合にのみ必要です。

デフォルト値は CONNECTORNAME/SYNCHRONOUSRESPONSEQUEUE です。

SynchronousRequestTimeout

DeliveryTransport が JMS の場合のみ適用できます。

コネクターが同期要求への応答を待機する時間を分単位で指定します。コネクターは、指定された時間内に応答を受信できなかった場合、元の同期要求メッセージをエラー・メッセージとともに障害キューに移動します。

デフォルト値は 0 です。

WireFormat

トランスポートのメッセージ・フォーマットです。

- RepositoryDirectory がローカル・ディレクトリーの場合、設定は CwXML です。
- RepositoryDirectory の値が <REMOTE> の場合、設定は CwBO です。

WsifSynchronousRequest Timeout

WAS 統合ブローカーでのみ使用されます。

コネクターが同期要求への応答を待機する時間を分単位で指定します。コネクターは、指定された時間内に応答を受信できなかった場合、元の同期要求メッセージをエラー・メッセージとともに障害キューに移動します。

デフォルト値は 0 です。

XMLNameSpaceFormat

WebSphere Message Brokers および WAS 統合ブローカーでのみ使用されます。

ビジネス・オブジェクト定義の XML 形式でネーム・スペースを short と long のどちらにするかをユーザーが指定できる強力なプロパティーです。

デフォルト値は `short` です。

付録 B. Connector Configurator

この付録では、Connector Configurator を使用してアダプターの構成プロパティー値を設定する方法について説明します。

Connector Configurator を使用して次の作業を行います。

- コネクターを構成するためのコネクター固有のプロパティー・テンプレートを作成する
- 構成ファイルを作成する
- 構成ファイル内のプロパティーを設定する

注:

本書では、ディレクトリー・パスに円記号 (¥) を使用します。UNIX システムを使用している場合は、円記号をスラッシュ (/) に置き換えてください。また、各オペレーティング・システムの規則に従ってください。

この付録では、次のトピックについて説明します。

Connector Configurator の概要

Connector Configurator では、次の統合ブローカーで使用するアダプターのコネクター・コンポーネントを構成できます。

- WebSphere InterChange Server (ICS)
- WebSphere MQ Integrator、WebSphere MQ Integrator Broker、および WebSphere Business Integration Message Broker (WebSphere Message Brokers (WMQI) と総称)
- WebSphere Application Server (WAS)

Connector Configurator を使用して次の作業を行います。

- コネクターを構成するための**コネクター固有のプロパティー・テンプレート**を作成します。
- **コネクター構成ファイル**を作成します。インストールするコネクターごとに 1 つ構成ファイルを作成する必要があります。
- 構成ファイルのプロパティーを設定します。
  場合によっては、コネクター・テンプレートでプロパティーに対して設定されているデフォルト値を変更する必要があります。また、サポートされるビジネス・オブジェクト定義と、ICS の場合はコラボレーションとともに使用するマップを

指定し、必要に応じてメッセージング、ロギング、トレース、およびデータ・ハンドラー・パラメーターを指定する必要があります。

Connector Configurator の実行モードと使用する構成ファイルのタイプは、実行する統合ブローカーによって異なる場合があります。例えば、使用している統合ブローカーが WMQI の場合、Connector Configurator を System Manager から実行するのではなく、直接実行します (570 ページの『スタンドアロン・モードでの Configurator の実行』を参照)。

コネクター構成プロパティーには、標準の構成プロパティー (すべてのコネクターがもつプロパティー) と、コネクター固有のプロパティー (特定のアプリケーションまたはテクノロジーのためにコネクターで必要なプロパティー) とが含まれます。

標準プロパティーはすべてのコネクターにより使用されるので、標準プロパティーを新規に定義する必要はありません。ファイルを作成すると、Connector Configurator により標準プロパティーがこの構成ファイルに挿入されます。ただし、Connector Configurator で各標準プロパティーの値を設定する必要があります。

標準プロパティーの範囲は、ブローカーと構成によって異なる可能性があります。特定のプロパティーに特定の値が設定されている場合にのみ使用できるプロパティーがあります。Connector Configurator の「標準のプロパティー」ウィンドウには、特定の構成で設定可能なプロパティーが表示されます。

ただし**コネクター固有プロパティー**の場合は、最初にプロパティーを定義し、その値を設定する必要があります。このため、特定のアダプターのコネクター固有プロパティーのテンプレートを作成します。システム内ですでにテンプレートが作成されている場合には、作成されているテンプレートを使用します。システム内でまだテンプレートが作成されていない場合には、572 ページの『新規テンプレートの作成』のステップに従い、テンプレートを新規に作成します。

注: Connector Configurator は、Windows 環境内でのみ実行されます。UNIX 環境でコネクターを実行する場合には、Windows で Connector Configurator を使用して構成ファイルを変更し、このファイルを UNIX 環境へコピーします。

Connector Configurator の始動

以下の 2 種類のモードでConnector Configurator を開始および実行できます。

- スタンドアロン・モードで個別に実行
- System Manager から実行

スタンドアロン・モードでの Configurator の実行

どのブローカーを実行している場合にも、Connector Configurator を個別に実行し、コネクター構成ファイルを編集できます。

これを行うには、以下のステップを実行します。

- **「スタート」>「プログラム」**から、**「IBM WebSphere InterChange Server」>「IBM WebSphere Business Integration Tools」>「Connector Configurator」**をクリックします。
- **「ファイル」>「新規」>「コネクター構成」**を選択します。

- **「システム接続: 統合ブローカー」**の隣のプルダウン・メニューをクリックします。使用しているブローカーに応じて、ICS、WebSphere Message Brokers、WAS のいずれかを選択します。

Connector Configurator を個別に実行して構成ファイルを生成してから、System Manager に接続してこの構成ファイルを System Manager プロジェクトに保存してください (577 ページの『構成ファイルの完成』を参照)。

System Manager からの Configurator の実行

System Manager から Connector Configurator を実行できます。

Connector Configurator を実行するには、以下のステップを実行します。

1. System Manager を開きます。
2. 「System Manager」ウィンドウで、**「統合コンポーネント・ライブラリー」**アイコンを展開し、**「コネクター」**を強調表示します。
3. System Manager メニュー・バーから、**「ツール」>「Connector Configurator」**をクリックします。「Connector Configurator」ウィンドウが開き、**「新規コネクター」**ダイアログ・ボックスが表示されます。
4. **「システム接続: 統合ブローカー」**の隣のプルダウン・メニューをクリックします。使用しているブローカーに応じて、ICS、WebSphere Message Brokers、WAS のいずれかを選択します。

既存の構成ファイルを編集するには、以下のステップを実行します。

- 「System Manager」ウィンドウの「コネクター」フォルダーで構成ファイルを選択し、右クリックします。Connector Configurator が開き、この構成ファイルの統合ブローカー・タイプおよびファイル名が上部に表示されます。
- Connector Configurator で**「ファイル」>「開く」**を選択します。プロジェクトまたはプロジェクトが保管されているディレクトリーからコネクター構成ファイルを選択します。
- 「標準のプロパティー」タブをクリックし、この構成ファイルに含まれるプロパティーを確認します。

コネクター固有のプロパティー・テンプレートの作成

コネクターの構成ファイルを作成するには、コネクター固有プロパティーのテンプレートとシステム提供の標準プロパティーが必要です。

コネクター固有プロパティーのテンプレートを新規に作成するか、または既存のコネクター定義をテンプレートとして使用します。

- テンプレートの新規作成については、572 ページの『新規テンプレートの作成』を参照してください。
- 既存のファイルを使用する場合には、既存のテンプレートを変更し、新しい名前でこのテンプレートを保管します。既存のテンプレートは ¥WebSphereAdapters¥bin¥Data¥App ディレクトリーにあります。

新規テンプレートの作成

このセクションでは、テンプレートでプロパティーを作成し、プロパティーの一般特性および値を定義し、プロパティー間の依存関係を指定する方法について説明します。次にそのテンプレートを保管し、新規コネクター構成ファイルを作成するためのベースとして使用します。

Connector Configurator でテンプレートを作成するには、以下のステップを実行します。

1. **「ファイル」>「新規」>「コネクター固有プロパティー・テンプレート (Connector-Specific Property Template)」**とクリックします。
2. **「コネクター固有プロパティー・テンプレート」** ダイアログ・ボックスが表示されます。
   - **「新規テンプレート名を入力してください」**の下の**「名前」**フィールドに、新規テンプレートの名前を入力します。テンプレートから新規構成ファイルを作成するためのダイアログ・ボックスを開くと、この名前が再度表示されます。
   - テンプレートに含まれているコネクター固有のプロパティー定義を調べるには、**「テンプレート名」**表示でそのテンプレートの名前を選択します。そのテンプレートに含まれているプロパティー定義のリストが**「テンプレートのプレビュー」**表示に表示されます。
3. テンプレートを作成するときには、コネクターに必要なプロパティー定義に類似したプロパティー定義が含まれている既存のテンプレートを使用できます。ご使用のコネクターで使用するコネクター固有のプロパティーが表示されるテンプレートが見つからない場合は、自分で作成する必要があります。
   - 既存のテンプレートを変更する場合には、**「変更する既存のテンプレートを選択してください: 検索テンプレート」**の下の**「テンプレート名」**テーブルのリストから、テンプレート名を選択します。
   - このテーブルには、現在使用可能なすべてのテンプレートの名前が表示されます。テンプレートを検索することもできます。

一般特性の指定

「次へ」をクリックしてテンプレートを選択すると、**「プロパティー: コネクター固有プロパティー・テンプレート」**ダイアログ・ボックスが表示されます。このダイアログ・ボックスには、定義済みプロパティーの「一般」特性のタブと「値」の制限のタブがあります。「一般」表示には以下のフィールドがあります。

- **一般:**
  プロパティー・タイプ
  更新されたメソッド
  説明
- **フラグ**
  標準のフラグ
- **カスタム・フラグ**
  フラグ

プロパティーの一般特性の選択を終えたら、**「値」**タブをクリックします。

値の指定

「値」タブを使用すると、プロパティーの最大長、最大複数値、デフォルト値、または値の範囲を設定できます。また、編集可能な値も設定できます。これを行うには、以下のステップを実行します。

1. **「値」**タブをクリックします。「一般」のパネルに代わって「値」の表示パネルが表示されます。
2. **「プロパティーを編集」**表示でプロパティーの名前を選択します。
3. **「最大長」**および**「最大複数値」**のフィールドに値を入力します。

新規プロパティー値を作成するには、以下のステップを実行します。

1. **「プロパティーを編集」**リストでプロパティーを選択し、右マウス・ボタンでクリックします。
2. ダイアログ・ボックスから**「追加」**を選択します。
3. 新規プロパティー値の名前を入力し、「OK」をクリックします。右側の**「値」**パネルに値が表示されます。

「値」パネルには、3 つの列からなるテーブルが表示されます。

「値」の列には、**「プロパティー値」**ダイアログ・ボックスで入力した値と、作成した以前の値が表示されます。

「デフォルト値」の列では、値のいずれかをデフォルトとして指定することができます。

「値の範囲」の列には、**「プロパティー値」**ダイアログ・ボックスで入力した範囲が表示されます。

値が作成されて、グリッドに表示されると、その表の表示内から編集できるようになります。

表にある既存の値の変更を行うには、その行の行番号をクリックして行全体を選択します。次に**「値」**フィールドを右マウス・ボタン・クリックし、**「値の編集 (Edit Value)」**をクリックします。

依存関係の設定

「一般」タブと**「値」**タブで変更を行ったら、**「次へ」**をクリックします。**「依存関係: コネクター固有プロパティー・テンプレート」**ダイアログ・ボックスが表示されます。

依存プロパティーは、別のプロパティーの値が特定の条件に合致する場合にのみ、テンプレートに組み込まれて、構成ファイルで使用されるプロパティーです。例えば、テンプレートに `PollQuantity` が表示されるのは、トランスポート機構が JMS であり、`DuplicateEventElimination` が `True` に設定されている場合のみです。プロパティーを依存プロパティーとして指定し、依存する条件を設定するには、以下のステップを実行します。

1. **「使用可能なプロパティー」**表示で、依存プロパティーとして指定するプロパティーを選択します。

2. **「プロパティーを選択」**フィールドで、ドロップダウン・メニューを使用して、条件値を持たせるプロパティーを選択します。
3. **「条件演算子」**フィールドで以下のいずれかを選択します。

   == (等しい)

   != (等しくない)

   > (より大)

   < (より小)

   >= (より大か等しい)

   <= (より小か等しい)
4. **「条件値」**フィールドで、依存プロパティーをテンプレートに組み込むために必要な値を入力します。
5. **「使用可能なプロパティー」**表示で強調表示された依存プロパティーで、矢印をクリックし、**「依存プロパティー」**表示に移動させます。
6. **「完了 (Finish)」**をクリックします。Connector Configurator により、XML 文書として入力した情報が、Connector Configurator がインストールされている ¥bin ディレクトリーの ¥data¥app の下に保管されます。

新しい構成ファイルを作成

構成ファイルを新規に作成するには、構成ファイルの名前を指定し、統合ブローカーを選択する必要があります。

- 「System Manager」ウィンドウで**「コネクター」**フォルダーを右クリックし、**「新規コネクターの作成」**を選択します。Connector Configurator が開き、**「新規コネクター」**ダイアログ・ボックスが表示されます。
- スタンドアロン・モードの場合は、Connector Configurator で**「ファイル」>「新規」>「コネクター構成」**を選択します。「新規コネクター」ウィンドウで、新規コネクターの名前を入力します。

また、統合ブローカーも選択する必要があります。選択したブローカーによって、構成ファイルに記述されるプロパティーが決まります。ブローカーを選択するには、以下のステップを実行します。

- **「Integration Broker」**フィールドで、ICS 接続、WebSphere Message Brokers 接続、WAS 接続のいずれかを選択します。
- この章で後述する説明に従って**「新規コネクター」**ウィンドウの残りのフィールドに入力します。

コネクター固有のテンプレートからの構成ファイルの作成

コネクター固有のテンプレートを作成すると、そのテンプレートを使用して構成ファイルを作成できます。

1. **「ファイル」>「新規」>「コネクター構成」**をクリックします。
2. 以下のフィールドを含む**「新規コネクター」**ダイアログ・ボックスが表示されます。

- **名前**

  コネクターの名前を入力します。名前では大文字と小文字が区別されます。入力する名前は、システムにインストールされているコネクターのファイル名と一貫性をもつ一意の名前である必要があります。

  重要: Connector Configurator では、入力された名前のスペルはチェックされません。名前が正しいことを確認してください。

- **システム接続**

  「ICS」、「WebSphere Message Brokers」、「WAS」のいずれかをクリックします。

- **コネクター固有プロパティー・テンプレートの選択 (Select Connector-Specific Property Template)**

  ご使用のコネクター用に設計したテンプレートの名前を入力します。**「テンプレート名」**表示に、使用可能なテンプレートが表示されます。「テンプレート名」表示で名前を選択すると、**「プロパティー・テンプレートのプレビュー」**表示に、そのテンプレートで定義されているコネクター固有のプロパティーが表示されます。

  使用するテンプレートを選択し、**「OK」**をクリックします。

3. 構成しているコネクターの構成画面が表示されます。タイトル・バーに統合ブローカーとコネクターの名前が表示されます。ここですべてのフィールドに値を入力して定義を完了するか、ファイルを保管して後でフィールドに値を入力するかを選択できます。
4. ファイルを保管するには、**「ファイル」>「保管」>「ファイルに」**をクリックするか、**「ファイル」>「保管」>「プロジェクトに」**をクリックします。プロジェクトに保管するには、System Manager が実行中である必要があります。
   ファイルとして保管する場合は、**「ファイル・コネクターを保管」**ダイアログ・ボックスが表示されます。`*.cfg` をファイル・タイプとして選択し、「ファイル名」フィールド内に名前が正しいスペル (大文字と小文字の区別を含む) で表示されていることを確認してから、ファイルを保管するディレクトリーにナビゲートし、**「保管」**をクリックします。Connector Configurator のメッセージ・パネルの状況表示に、構成ファイルが正常に作成されたことが示されます。

   重要: ここで設定するディレクトリー・パスおよび名前は、コネクターの始動ファイルで指定するコネクター構成ファイルのパスおよび名前に一致している必要があります。
5. この章で後述する手順に従って、「Connector Configurator」ウィンドウの各タブにあるフィールドに値を入力し、コネクター定義を完了します。

既存ファイルの使用

使用可能な既存ファイルは、以下の 1 つまたは複数の形式になります。

- コネクター定義ファイル。
  コネクター定義ファイルは、特定のコネクターのプロパティーと、適用可能なデフォルト値がリストされたテキスト・ファイルです。コネクターの配布パッケー

ジの ¥repository ディレクトリー内には、このようなファイルが格納されていることがあります (通常、このファイルの拡張子は .txt です。例えば、XML コネクターの場合は CN_XML.txt です)。

- ICS リポジトリー・ファイル。
  コネクターの以前の ICS インプリメンテーションで使用した定義は、そのコネクターの構成で使用されたリポジトリー・ファイルで使用可能になります。そのようなファイルの拡張子は、通常 .in または .out です。
- コネクターの以前の構成ファイル。
  これらのファイルの拡張子は、通常 *.cfg です。

これらのいずれのファイル・ソースにも、コネクターのコネクター固有プロパティーのほとんど、あるいはすべてが含まれますが、この章内の後で説明するように、コネクター構成ファイルは、ファイルを開いて、プロパティーを設定しない限り完成しません。

既存ファイルを使用してコネクターを構成するには、Connector Configurator でそのファイルを開き、構成を修正してそのファイルを再保管する必要があります。

以下のステップを実行して、ディレクトリーから *.txt、*.cfg、または *.in ファイルを開きます。

1. Connector Configurator 内で、**「ファイル」>「開く」>「ファイルから」**とクリックします。
2. **「ファイル・コネクターを開く」**ダイアログ・ボックス内で、以下のいずれかのファイル・タイプを選択して、使用可能なファイルを調べます。
   - 構成 (*.cfg)
   - ICS リポジトリー (*.in、*.out)

     ICS 環境でのコネクターの構成にリポジトリー・ファイルが使用された場合には、このオプションを選択します。リポジトリー・ファイルに複数のコネクター定義が含まれている場合は、ファイルを開くとすべての定義が表示されます。
   - すべてのファイル (*.*)

     コネクターのアダプター・パッケージに *.txt ファイルが付属していた場合、または別の拡張子で定義ファイルが使用可能である場合は、このオプションを選択します。
3. ディレクトリー表示内で、適切なコネクター定義ファイルへ移動し、ファイルを選択し、**「開く」**をクリックします。

System Manager プロジェクトからコネクター構成を開くには、以下のステップを実行します。

1. System Manager を始動します。System Manager が開始されている場合にのみ、構成を System Manager から開いたり、System Manager に保管したりできます。
2. Connector Configurator を始動します。
3. **「ファイル」>「開く」>「プロジェクトから」**とクリックします。

構成ファイルの完成

構成ファイルを開くか、プロジェクトからコネクターを開くと、「Connector Configurator」ウィンドウに構成画面が表示されます。この画面には、現在の属性と値が表示されます。

構成画面のタイトルには、ファイル内で指定された統合ブローカーとコネクターの名前が表示されます。正しいブローカーが設定されていることを確認してください。正しいブローカーが設定されていない場合、コネクターを構成する前にブローカー値を変更してください。これを行うには、以下のステップを実行します。

1. **「標準のプロパティー」**タブで、BrokerType プロパティーの値フィールドを選択します。ドロップダウン・メニューで、値 ICS、WMQI、または WAS を選択します。
2. 選択したブローカーに関連付けられているプロパティーが「標準のプロパティー」タブに表示されます。ここでファイルを保管するか、または 580 ページの『サポートされるビジネス・オブジェクト定義の指定』の説明に従い残りの構成フィールドに値を入力することができます。
3. 構成が完了したら、**「ファイル」>「保管」>「プロジェクトに」**を選択するか、または**「ファイル」>「保管」>「ファイルに」**を選択します。

   ファイルに保管する場合は、*.cfg を拡張子として選択し、ファイルの正しい格納場所を選択して、**「保管」**をクリックします。

   複数のコネクター構成を開いている場合、構成をすべてファイルに保管するには**「ファイルにすべて保管」**を選択し、コネクター構成をすべて System Manager プロジェクトに保管するには**「プロジェクトにすべて保管」**をクリックします。

   Connector Configurator では、ファイルを保管する前に、必須の標準プロパティーすべてに値が設定されているかどうかが確認されます。必須の標準プロパティーに値が設定されていない場合、Connector Configurator は、検証が失敗したというメッセージを表示します。構成ファイルを保管するには、そのプロパティーの値を指定する必要があります。

構成ファイル・プロパティーの設定

新規のコネクター構成ファイルを作成して名前を付けるとき、または既存のコネクター構成ファイルを開くときには、Connector Configurator によって構成画面が表示されます。構成画面には、必要な構成値のカテゴリーに対応する複数のタブがあります。

Connector Configurator では、すべてのブローカーで実行されているコネクターで、以下のカテゴリーのプロパティーに値が設定されている必要があります。

- 標準のプロパティー
- コネクター固有のプロパティー
- サポートされるビジネス・オブジェクト
- トレース/ログ・ファイルの値
- データ・ハンドラー (保証付きイベント・デリバリーで JMS メッセージングを使用するコネクターの場合に該当する)

注: JMS メッセージングを使用するコネクターの場合は、データをビジネス・オブジェクトに変換するデータ・ハンドラーの構成に関して追加のカテゴリーが表示される場合があります。

ICS で実行されているコネクターの場合、以下のプロパティーの値も設定されている必要があります。

- 関連付けられたマップ
- リソース
- メッセージング (該当する場合)

重要: Connector Configurator では、英語文字セットまたは英語以外の文字セットのいずれのプロパティー値も設定可能です。ただし、標準のプロパティーおよびコネクター固有プロパティー、およびサポートされるビジネス・オブジェクトの名前では、英語文字セットのみを使用する必要があります。

標準プロパティーとコネクター固有プロパティーの違いは、以下のとおりです。

- コネクターの標準プロパティーは、コネクターのアプリケーション固有のコンポーネントとブローカー・コンポーネントの両方によって共用されます。すべてのコネクターが同じ標準プロパティーのセットを使用します。これらのプロパティーの説明は、各アダプター・ガイドの付録 A にあります。変更できるのはこれらの値の一部のみです。
- アプリケーション固有プロパティーは、コネクターのアプリケーション固有コンポーネント (アプリケーションと直接対話するコンポーネント) のみに適用されます。各コネクターには、そのコネクターのアプリケーションだけで使用されるアプリケーション固有のプロパティーがあります。これらのプロパティーには、デフォルト値が用意されているものもあれば、そうでないものもあります。また、一部のデフォルト値は変更することができます。各アダプター・ガイドのインストールおよび構成の章に、アプリケーション固有のプロパティーおよび推奨値が記述されています。

「標準のプロパティー」と**「コネクター固有プロパティー」**のフィールドは、どのフィールドが構成可能であるかを示すために色分けされています。

- 背景がグレーのフィールドは、標準のプロパティーを表します。値を変更することはできますが、名前の変更およびプロパティーの除去はできません。
- 背景が白のフィールドは、アプリケーション固有のプロパティーを表します。これらのプロパティーは、アプリケーションまたはコネクターの特定のニーズによって異なります。値の変更も、これらのプロパティーの除去も可能です。
- 「値」フィールドは構成できます。
- プロパティーごとに**「更新メソッド」**フィールドが表示されます。これは、変更された値をアクティブにするためにコンポーネントまたはエージェントの再始動が必要かどうかを示します。この設定を構成することはできません。

標準コネクター・プロパティーの設定

標準のプロパティーの値を変更するには、以下の手順を実行します。

1. 値を設定するフィールド内でクリックします。

2. 値を入力するか、ドロップダウン・メニューが表示される場合にはメニューから値を選択します。
3. 標準のプロパティーの値をすべて入力すると、以下のいずれかを実行することができます。
   - 変更内容を破棄し、元の値を保持したままで Connector Configurator を終了するには、**「ファイル」>「終了」**をクリックし (またはウィンドウを閉じ)、変更内容を保管するかどうかを確認するプロンプトが出されたら**「いいえ」**をクリックします。
   - Connector Configurator 内の他のカテゴリーの値を入力するには、そのカテゴリーのタブを選択します。**「標準のプロパティー」** (またはその他のカテゴリー) で入力した値は、次のカテゴリーに移動しても保持されます。ウィンドウを閉じるときに、すべてのカテゴリーで入力した値を一括して保管するかまたは破棄するかを確認するプロンプトが出されます。
   - 修正した値を保管するには、**「ファイル」>「終了」**をクリックし (またはウィンドウを閉じ)、変更内容を保管するかどうかを確認するプロンプトが出されたら**「はい」**をクリックします。「ファイル」メニューまたはツールバーから**「保管」>「ファイルに」**をクリックする方法もあります。

アプリケーション固有の構成プロパティーの設定

アプリケーション固有の構成プロパティーの場合、プロパティー名の追加または変更、値の構成、プロパティーの削除、およびプロパティーの暗号化が可能です。プロパティーのデフォルトの長さは 255 文字です。

1. グリッドの左上端の部分で右マウス・ボタン・クリックします。ポップアップ・メニュー・バーが表示されます。**「追加」**をクリックしてプロパティーを追加します。子プロパティーを追加するには、親行番号を右マウス・ボタン・クリックして、**「子を追加」**をクリックします。
2. プロパティーまたは子プロパティーの値を入力します。
3. プロパティーを暗号化するには、**「暗号化」**ボックスを選択します。
4. 578 ページの『標準コネクター・プロパティーの設定』で説明したように、変更内容を保管するかまたは破棄するかを選択します。

各プロパティーごとに表示される「更新メソッド」は、変更された値をアクティブにするためにコンポーネントまたはエージェントの再始動が必要かどうかを示します。

重要: 事前設定のアプリケーション固有のコネクター・プロパティー名を変更すると、コネクターに障害が発生する可能性があります。コネクターをアプリケーションに接続したり正常に実行したりするために、特定のプロパティー名が必要である場合があります。

コネクター・プロパティーの暗号化

「コネクター固有プロパティー」ウィンドウの**「暗号化」**チェック・ボックスにチェックマークを付けると、アプリケーション固有のプロパティーを暗号化することができます。値の暗号化を解除するには、**「暗号化」** チェック・ボックスをクリックしてチェックマークを外し、**「検証」** ダイアログ・ボックスに正しい値を入力し、**「OK」** をクリックします。入力された値が正しい場合は、暗号化が解除された値が表示されます。

各プロパティーとそのデフォルト値のリストおよび説明は、各コネクターのアダプター・ユーザー・ガイドにあります。

プロパティーに複数の値がある場合には、プロパティーの最初の値に**「暗号化」**チェック・ボックスが表示されます。**「暗号化」**を選択すると、そのプロパティーのすべての値が暗号化されます。プロパティーの複数の値を暗号化解除するには、そのプロパティーの最初の値の**「暗号化」** チェック・ボックスをクリックしてチェックマークを外してから、**「検証」**ダイアログ・ボックスで新規の値を入力します。入力値が一致すれば、すべての複数値が暗号化解除されます。

更新メソッド

『*付録 A. コネクターの標準構成プロパティー*』の 550 ページの『プロパティー値の設定と更新』にある更新メソッドの説明を参照してください。

サポートされるビジネス・オブジェクト定義の指定

Connector Configurator の**「サポートされているビジネス・オブジェクト」**タブで、コネクターが使用するビジネス・オブジェクトを指定します。汎用ビジネス・オブジェクトと、アプリケーション固有のビジネス・オブジェクトの両方を指定する必要があり、またそれらのビジネス・オブジェクト間のマップの関連を指定することが必要です。

注: コネクターによっては、アプリケーションでイベント通知や (メタオブジェクトを使用した) 追加の構成を実行するために、特定のビジネス・オブジェクトをサポートされているものとして指定することが必要な場合もあります。詳細は、「*コネクター開発ガイド (C++ 用)*」または「*コネクター開発ガイド (Java 用)*」を参照してください。

ご使用のブローカーが ICS の場合

ビジネス・オブジェクト定義がコネクターでサポートされることを指定する場合や、既存のビジネス・オブジェクト定義のサポート設定を変更する場合は、**「サポートされているビジネス・オブジェクト」**タブをクリックし、以下のフィールドを使用してください。

ビジネス・オブジェクト名: ビジネス・オブジェクト定義がコネクターによってサポートされることを指定するには、System Manager を実行し、以下の手順を実行します。

1. **「ビジネス・オブジェクト名」**リストの空のフィールドをクリックします。System Manager プロジェクトに存在するすべてのビジネス・オブジェクト定義を示すドロップダウン・リストが表示されます。
2. 追加するビジネス・オブジェクトをクリックします。
3. ビジネス・オブジェクトの**「エージェント・サポート」**(以下で説明) を設定します。
4. 「Connector Configurator」ウィンドウの「ファイル」メニューで、**「プロジェクトに保管」**をクリックします。追加したビジネス・オブジェクト定義に指定されたサポートを含む、変更されたコネクター定義が、System Manager の ICL (Integration Component Library) プロジェクトに保管されます。

サポートされるリストからビジネス・オブジェクトを削除する場合は、以下の手順を実行します。

1. ビジネス・オブジェクト・フィールドを選択するため、そのビジネス・オブジェクトの左側の番号をクリックします。
2. 「Connector Configurator」ウィンドウの**「編集」**メニューから、**「行を削除」**をクリックします。リスト表示からビジネス・オブジェクトが除去されます。
3. **「ファイル」**メニューから、**「プロジェクトに保管」**をクリックします。

サポートされるリストからビジネス・オブジェクトを削除すると、コネクター定義が変更され、削除されたビジネス・オブジェクトはコネクターのこのインプリメンテーションで使用不可になります。コネクターのコードに影響したり、そのビジネス・オブジェクト定義そのものが System Manager から削除されることはありません。

エージェント・サポート: ビジネス・オブジェクトにエージェント・サポートがある場合、システムは、コネクター・エージェントを介してアプリケーションにデータを配布する際にそのビジネス・オブジェクトの使用を試みます。

一般に、コネクターのアプリケーション固有ビジネス・オブジェクトは、そのコネクターのエージェントによってサポートされますが、汎用ビジネス・オブジェクトはサポートされません。

ビジネス・オブジェクトがコネクター・エージェントによってサポートされるよう指定するには、**「エージェント・サポート」**ボックスにチェックマークを付けます。「Connector Configurator」ウィンドウでは、「エージェント・サポート」の選択の妥当性は検査されません。

最大トランザクション・レベル: コネクターの最大トランザクション・レベルは、そのコネクターがサポートする最大のトランザクション・レベルです。

ほとんどのコネクターの場合、選択可能な項目は**「最大限の努力」**のみです。

トランザクション・レベルの変更を有効にするには、サーバーを再始動する必要があります。

ご使用のブローカーが WebSphere Message Broker の場合

スタンドアロン・モードで作業している (System Manager に接続していない) 場合、手動でビジネス・オブジェクト名を入力する必要があります。

System Manager が実行中の場合、**「サポートされているビジネス・オブジェクト」**タブの**「ビジネス・オブジェクト名」**列の下にある空のボックスを選択できます。コンボ・ボックスが表示され、コネクターが属する統合コンポーネント・ライブラリー・プロジェクトから選択できるビジネス・オブジェクトのリストが示されます。このリストから目的のビジネス・オブジェクトを選択します。

「メッセージ・セット ID」 は WebSphere Business Integration Message Broker 5.0 のオプション・フィールドで、指定されている場合一意である必要はありません。ただし、WebSphere MQ Integrator および Integrator Broker 2.1 では、一意の **ID** を指定する必要があります。

ご使用のブローカーが WAS の場合

使用するブローカー・タイプとして WebSphere Application Server を選択する場合、Connector Configurator にメッセージ・セット ID は必要ありません。**「サポートされるビジネス・オブジェクト」**タブには、サポートされるビジネス・オブジェクトの**「ビジネス・オブジェクト名」**列のみが表示されます。

スタンドアロン・モードで作業している (System Manager に接続していない) 場合、手動でビジネス・オブジェクト名を入力する必要があります。

System Manager が実行中の場合、「サポートされているビジネス・オブジェクト」タブの「ビジネス・オブジェクト名」列の下にある空のボックスを選択できます。コンボ・ボックスが表示され、コネクターが属する統合コンポーネント・ライブラリー・プロジェクトから選択できるビジネス・オブジェクトのリストが示されます。このリストから目的のビジネス・オブジェクトを選択します。

関連付けられたマップ (ICS のみ)

各コネクターは、現在 WebSphere InterChange Server でアクティブなビジネス・オブジェクト定義、およびそれらの関連付けられたマップのリストをサポートします。このリストは、**「関連付けられたマップ」**タブを選択すると表示されます。

ビジネス・オブジェクトのリストには、エージェントでサポートされるアプリケーション固有のビジネス・オブジェクトと、コントローラーがサブスクライブ・コラボレーションに送信する、対応する汎用オブジェクトが含まれます。マップの関連によって、アプリケーション固有のビジネス・オブジェクトを汎用ビジネス・オブジェクトに変換したり、汎用ビジネス・オブジェクトをアプリケーション固有のビジネス・オブジェクトに変換したりするときに、どのマップを使用するかが決定されます。

特定のソースおよび宛先ビジネス・オブジェクトについて一意的に定義されたマップを使用する場合、表示を開くと、マップは常にそれらの該当するビジネス・オブジェクトに関連付けられます。ユーザーがそれらを変更する必要はありません (変更できません)。

サポートされるビジネス・オブジェクトで使用可能なマップが複数ある場合は、そのビジネス・オブジェクトを、使用する必要のあるマップに明示的にバインドすることが必要になります。

「関連付けられたマップ」タブには以下のフィールドが表示されます。

- **ビジネス・オブジェクト名**

  これらは、**「サポートされているビジネス・オブジェクト」**タブで指定した、このコネクターでサポートされるビジネス・オブジェクトです。「サポートされているビジネス・オブジェクト」タブで、サポートされるビジネス・オブジェクトを追加指定した場合、それらの内容は、「Connector Configurator」ウィンドウの**「ファイル」**メニューから**「プロジェクトに保管」**を選択して、変更を保管した後に、このリストに反映されます。
- **関連付けられたマップ**

この表示には、コネクターの、サポートされるビジネス・オブジェクトでの使用のためにシステムにインストールされたすべてのマップが示されます。各マップのソース・ビジネス・オブジェクトは、**「ビジネス・オブジェクト名」**表示でマップ名の左側に表示されます。

- **明示的**

  場合によっては、関連付けられたマップを明示的にバインドすることが必要になります。

  明示的バインディングが必要なのは、特定のサポートされるビジネス・オブジェクトに複数のマップが存在する場合のみです。ICS は、ブート時、コネクターごとに、サポートされる各ビジネス・オブジェクトにマップを自動的にバインドしようとします。複数のマップでその入力データとして同一のビジネス・オブジェクトが使用されている場合、サーバーは、他のマップのスーパーセットである 1 つのマップを見つけて、バインドしようとします。

  他のマップのスーパーセットであるマップがないと、サーバーは、ビジネス・オブジェクトを単一のマップにバインドすることができないため、バインディングを明示的に設定することが必要になります。

  以下の手順を実行して、マップを明示的にバインドします。

  1. **「明示 (Explicit)」**列で、バインドするマップのチェック・ボックスにチェックマークを付けます。
  2. ビジネス・オブジェクトに関連付けるマップを選択します。
  3. 「Connector Configurator」ウィンドウの**「ファイル」**メニューで、**「プロジェクトに保管」**をクリックします。
  4. プロジェクトを ICS に配置します。
  5. 変更を有効にするため、サーバーをリブートします。

リソース (ICS)

「リソース」タブでは、コネクター・エージェントがコネクター・エージェント並列処理を使用して、同時に複数のプロセスを処理するかどうか、またどの程度処理するかを決定する値を設定することができます。

すべてのコネクターでこの機能がサポートされるわけではありません。複数のプロセスを使用するよりも複数のスレッドを使用する方が通常は効率的であるため、Java でマルチスレッドとして設計されたコネクター・エージェントを実行している場合、この機能を使用することはお勧めできません。

メッセージング (ICS)

メッセージング・プロパティーは、`DeliveryTransport` 標準プロパティーの値として MQ を設定し、ブローカー・タイプとして ICS を設定した場合にのみ、使用可能です。これらのプロパティーは、コネクターによるキューの使用方法に影響します。

トレース/ログ・ファイル値の設定

コネクター構成ファイルまたはコネクター定義ファイルを開くと、Connector Configurator は、そのファイルのログおよびトレースの値をデフォルト値として使用します。Connector Configurator 内でこれらの値を変更できます。

ログとトレースの値を変更するには、以下の手順を実行します。

1. **「トレース/ログ・ファイル」**タブをクリックします。
2. ログとトレースのどちらでも、以下のいずれかまたは両方へのメッセージの書き込みを選択できます。
   - コンソールに (STDOUT):
     ログ・メッセージまたはトレース・メッセージを STDOUT ディスプレイに書き込みます。

     注: STDOUT オプションは、Windows プラットフォームで実行しているコネクターの**「トレース/ログ・ファイル」**タブでのみ使用できます。
   - ファイルに:
     ログ・メッセージまたはトレース・メッセージを指定されたファイルに書き込みます。ファイルを指定するには、ディレクトリー・ボタン (省略符号) をクリックし、指定する保管場所に移動し、ファイル名を指定し、**「保管」**をクリックします。ログ・メッセージまたはトレース・メッセージは、指定した場所の指定したファイルに書き込まれます。

     注: ログ・ファイルとトレース・ファイルはどちらも単純なテキスト・ファイルです。任意のファイル拡張子を使用してこれらのファイル名を設定できます。ただし、トレース・ファイルの場合、拡張子として `.trc` ではなく `.trace` を使用することをお勧めします。これは、システム内に存在する可能性がある他のファイルとの混同を避けるためです。ログ・ファイルの場合、通常使用されるファイル拡張子は `.log` および `.txt` です。

データ・ハンドラー

データ・ハンドラー・セクションの構成が使用可能となるのは、DeliveryTransport の値に JMS を、また ContainerManagedEvents の値に JMS を指定した場合のみです。すべてのアダプターでこのデータ・ハンドラーを使用できるわけではありません。

これらのプロパティーに使用する値については、付録 A の『標準のプロパティー』の `ContainerManagedEvents` の下の説明を参照してください。その他の詳細は、「*コネクター開発ガイド (C++ 用)*」または「*コネクター開発ガイド (Java 用)*」を参照してください。

構成ファイルの保管

コネクターの構成が完了したら、コネクター構成ファイルを保管します。Connector Configurator では、構成中に選択したブローカー・モードで構成ファイルが保管されます。Connector Configurator のタイトル・バーには現在のブローカー・モード (ICS、WMQI、または WAS) が常に表示されます。

ファイルは XML 文書として保管されます。XML 文書は次の 3 通りの方法で保管できます。

- System Manager から、*.con 拡張子付きファイルとして統合コンポーネント・ライブラリーに保管します。
- System Manager から、指定したディレクトリーに *.con 拡張子付きファイルとして保管します。
- スタンドアロン・モードで、ディレクトリー・フォルダーに *.cfg 拡張子付きファイルとして保管します。デフォルトでは、このファイルは ¥WebSphereAdapters¥bin¥Data¥App に保管されます。
- WebSphere Application Server プロジェクトをセットアップしている場合には、このファイルを WebSphere Application Server プロジェクトに保管することもできます。

System Manager でのプロジェクトの使用法、および配置の詳細については、以下のインプリメンテーション・ガイドを参照してください。

- ICS: 「*WebSphere InterChange Server システム・インプリメンテーション・ガイド*」
- WebSphere Message Brokers: 「*WebSphere Message Brokers 使用アダプター・インプリメンテーション・ガイド*」
- WAS: 「*アダプター実装ガイド (WebSphere Application Server)*」

構成ファイルの変更

既存の構成ファイルの統合ブローカー設定を変更できます。これにより、他のブローカーで使用する構成ファイルを新規に作成するときに、このファイルをテンプレートとして使用できます。

注: 統合ブローカーを切り替える場合には、ブローカー・モード・プロパティーと同様に他の構成プロパティーも変更する必要があります。

既存の構成ファイルでのブローカーの選択を変更するには、以下の手順を実行します (オプション)。

- Connector Configurator で既存の構成ファイルを開きます。
- **「標準のプロパティー」**タブを選択します。
- 「標準のプロパティー」タブの**「ブローカー・タイプ」**フィールドで、ご使用のブローカーに合った値を選択します。
  現行値を変更すると、プロパティー画面の利用可能なタブおよびフィールド選択がただちに変更され、選択した新規ブローカーに適したタブとフィールドのみが表示されます。

構成の完了

コネクターの構成ファイルを作成し、そのファイルを変更した後で、コネクターの始動時にコネクターが構成ファイルの位置を特定できるかどうかを確認してください。

これを行うには、コネクターが使用する始動ファイルを開き、コネクター構成ファイルに使用されている格納場所とファイル名が、ファイルに対して指定した名前およびファイルを格納したディレクトリーまたはパスと正確に一致しているかどうかを検証します。

グローバル化環境における Connector Configurator の使用

Connector Configurator はグローバル化され、構成ファイルと統合ブローカー間の文字変換を処理できます。 Connector Configurator では、ネイティブなエンコード方式を使用しています。構成ファイルに書き込む場合は UTF-8 エンコード方式を使用します。

Connector Configurator は、以下の場所で英語以外の文字をサポートします。

- すべての値のフィールド
- ログ・ファイルおよびトレース・ファイル・パス (**「トレース/ログ・ファイル」**タブで指定)

CharacterEncoding および Locale 標準構成プロパティーのドロップ・リストに表示されるのは、サポートされる値のサブセットのみです。ドロップ・リストに、サポートされる他の値を追加するには、製品ディレクトリーの ¥Data¥Std¥stdConnProps.xml ファイルを手動で変更する必要があります。

例えば Locale プロパティーの値のリストにロケール en_GB を追加するには、stdConnProps.xml ファイルを開き、以下に太文字で示される行を追加してください。

```
<Property name="Locale"
isRequired="true"
updateMethod="component restart">
                <ValidType>String</ValidType>
            <ValidValues>
                                <Value>ja_JP</Value>
                                <Value>ko_KR</Value>
                                <Value>zh_CN</Value>
                                <Value>zh_TW</Value>
                                <Value>fr_FR</Value>
                                <Value>de_DE</Value>
                                <Value>it_IT</Value>
                                <Value>es_ES</Value>
                                <Value>pt_BR</Value>
                                <Value>en_US</Value>
                                <Value>en_GB</Value>
                    <DefaultValue>en_US</DefaultValue>
            </ValidValues>
    </Property>
```

付録 C. Connector Script Generator

Connector Script Generator ユーティリティーは、UNIX プラットフォームで実行されるコネクター用のコネクター・スクリプトを作成または変更します。このツールは、以下のいずれかの場合に使用してください。

- WebSphere Business Integration Adapters のインストーラーを使用しないで追加したコネクター用に、新しいコネクター始動スクリプトを生成する。
- あるコネクター用に既存の始動スクリプトを変更して、正しい構成ファイル・パスが含まれるようにする。

Connector Script Generator を実行するには、次のようにしてください。

1. *ProductDir*/bin ディレクトリーまでナビゲートします。
2. ./ConnConfig.sh コマンドを入力します。

   図 78 に示すように、Connector Script Generator 画面が表示されます。

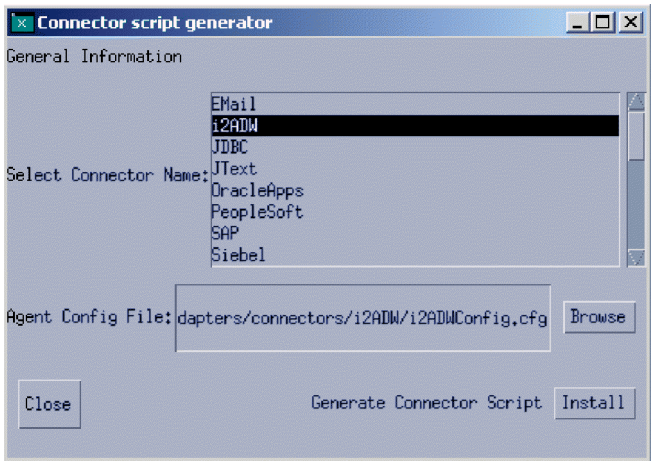

図 78. Connector Script Generator

3. 「コネクター名の選択 (Select Connector Name)」リストから、始動スクリプトの生成対象であるコネクターを選択します。
4. 「エージェント構成ファイル (Agent Config File)」では、コネクターの構成ファイルの絶対パス名を入力するか、**「参照」**をクリックして対象ファイルを選択します。
5. コネクター・スクリプトを生成または更新するために**「インストール (Install)」**をクリックします。

   connector_manager_*ConnectorName* ファイル (この *ConnectorName* は、構成対象となっているコネクターの名前です) が *ProductDir*/bin ディレクトリーに作成されます。
6. **「閉じる」**をクリックします。

付録 D. コネクター機能チェックリスト

この付録では、コネクター機能チェックリストについて説明します。

コネクター機能チェックリストの使用に関するガイドライン

コネクター機能チェックリストでは、各コネクターの標準機構に関して簡単に説明します。この機能リストで、コネクターの振る舞いに関するベースラインが確立されます。したがって、新規のコネクターを設計する場合に、標準コネクター機能へのクイック・リファレンスとしてこのリストを使用できます。

コネクターの実装局面で、この機能リストを使用して、コネクターの機能性を記述する仕様を作成することができます。リストを使用するには、次のステップに従います。

- コネクターがサポートする各機能ごとに「*完全*」にチェックマークを付けます。
- コネクターが部分的にサポートする各機能ごとに「*部分*」にチェックマークを付け、実装を記述するメモを組み込みます。
- コネクターがサポートしない各機能ごとに「*なし*」にチェックマークを付けます。
- コネクターに関係のない各機能ごとに「*使用不可*」にチェックマークを付けます。例えば、コネクターがイベント通知を実装しない場合には、すべてのイベント通知機能で「使用不可」にチェックマークを付けます。

ある機能が標準的な振る舞いによってサポートされない場合には、「*部分*」にチェックマークを付け、追加情報を入力します。

要求処理の標準的な振る舞い

表 162 では、ビジネス・オブジェクト要求のコネクター処理での標準機構をリストします。この表には、各機能に関する簡単な説明および機能の詳細情報が記述される本書内のセクションのページ番号も記載されています。

表 162. 要求処理の標準機構

カテゴリーと名前	説明	サポート状況	
ビジネス・オブジェクトと属性のネーミング			
ビジネス・オブジェクト名	ビジネス・オブジェクト名には、コネクターへのセマンティック値はありません。2 つのビジネス・オブジェクトで、構造、データ、およびアプリケーション固有の情報が同一で、名前が異なる場合には、コネクターで同一に処理します。	__	完全
		__	部分
		__	なし
		__	使用不可
属性名	ビジネス・オブジェクト内の属性名には、コネクターへのセマンティック値はありません。アプリケーション表名または列名などの値は、属性名にではなく、属性のアプリケーション固有の情報フィールドに格納します。	__	完全
		__	部分
		__	なし
		__	使用不可
Create			

表 162. 要求処理の標準機構 (続き)

カテゴリーと名前	説明	サポート状況
Create 動詞	コネクターはオブジェクトを宛先アプリケーション内に作成します。アプリケーション・オブジェクトは、子オブジェクトなどの、すべての値をビジネス・オブジェクトに組み込みます。99 ページの『Create 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
Delete		
Delete 動詞	コネクターは Delete 動詞をサポートします。コネクターは、この動詞を処理するとき、論理削除ではなく、真の物理削除を行います。119 ページの『Delete 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
論理削除	コネクターは、Update 動詞のみを介して論理削除操作をサポートします。Delete 動詞は、物理削除でのみ使用されます。116 ページの『論理削除イベントを表すビジネス・オブジェクトが持つ意味』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
Exist		
Exist 動詞	コネクターは、アプリケーション・データベースでのエンティティーの存在を検査します。渡されたオブジェクトが、アプリケーション・データベースに存在する場合には SUCCEED を戻し、オブジェクトがアプリケーション・データベースに存在しない場合には FAIL を戻します。121 ページの『Exists 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
Retrieve		
Retrieve 動詞	Retrieve 動詞の処理時には、階層イメージ全体 (すべての子ビジネス・オブジェクトを含む) がアプリケーションから取得されます。取得は、ビジネス・オブジェクトのキー値のみをベースに行われます。103 ページの『Retrieve 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
欠落している子オブジェクトを無視	IgnoreMissingChildObject がビジネス・オブジェクト・レベル・アプリケーション固有の情報で true に設定されているときには、ビジネス・オブジェクトで指定した子がアプリケーションで必ずしもすべて検出されない場合でも、コネクターは SUCCEED を戻します。 106 ページの『子オブジェクトの取得』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
RetrieveByContent		
RetrieveBy Content 動詞	階層イメージ全体 (すべての子オブジェクトを含む) が、非キー値のサブセットのみをベースに取得されます。109 ページの『RetrieveByContent 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
複数結果	アプリケーションから複数のオブジェクトが取得された場合には、RetrieveByContent は最初のオブジェクトを戻し、戻りコード MULTIPLE_HITS を使用します。109 ページの『RetrieveByContent 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
欠落している子オブジェクトを無視	IgnoreMissingChildObject がビジネス・オブジェクト・レベル・アプリケーション固有の情報で true に設定されているときには、ビジネス・オブジェクトで指定した子がアプリケーションで必ずしもすべて検出されない場合でも、コネクターは SUCCEED を戻します。	__ 完全 __ 部分 __ なし __ 使用不可
Update		
変更後イメージのサポート	コネクターは、宛先アプリケーション内のオブジェクトを doVerbFor()call で取得されたビジネス・オブジェクトと完全に一致させるために必要なステップをすべて実行します。111 ページの『Update 動詞の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可

表 162. 要求処理の標準機構 (続き)

カテゴリーと名前	説明	サポート状況
差分サポート	コネクターは、ソース・ビジネス・オブジェクト内で受け取ったオブジェクトおよび動詞そのものを処理します。宛先アプリケーション・オブジェクトは、アプリケーション表記をソース・ビジネス・オブジェクトと一致させることによってではなく、ソース・ビジネス・オブジェクトの内容を処理することによってのみ更新されます。[現行では IBM 標準ではありません。]	__ 完全 __ 部分 __ なし __ 使用不可
KeepRelations	KeepRelations を指定するとき、ターゲット・アプリケーションで子関係は破棄されません。指定しない場合には、最初にすべての子関係が破棄されます。続いて、InterChange Server から送信された子オブジェクトが作成され、子関係が復元されます。「破棄」とは、子との関係の論理または物理削除を意味しますが、コネクターおよびアプリケーションの機能性によっては、子そのものの削除を意味する場合もあります。KeepRelations は、親オブジェクトでの子配列に関するアプリケーション固有の情報として (子そのものに関するテキストとしてではなく) 設定されます。構文は、`keeprelations=true` となります。	__ 完全 __ 部分 __ なし __ 使用不可
動詞サポート		
サブ動詞サポート	コネクターは、親オブジェクトでの動詞とは無関係に、子オブジェクトでの動詞の処理をサポートします。子ビジネス・オブジェクトで動詞を設定すると、コネクターは、トップレベル・ビジネス・オブジェクトでの動詞にかかわりなく、子動詞が指示する操作を実行します。子ビジネス・オブジェクト要求に動詞が設定されていない場合は、コネクターは、子動詞を `null` として、処理を行わないか、トップレベル・ビジネス・オブジェクトの動詞に子動詞を設定するか、またはコネクターが実行する必要のある操作に動詞の値を設定します。97 ページの『動詞安定度』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
動詞安定度	ビジネス・オブジェクト内の動詞は、要求と応答の全サイクルを通じて安定していなければなりません。コネクターがビジネス・オブジェクト要求を受け取るとき、InterChange Server に戻された階層オブジェクトには、元の要求と同じ動詞が存在する必要があります。ただし、元の要求でヌルであった子ビジネス・オブジェクトに設定されている動詞は例外です。	__ 完全 __ 部分 __ なし __ 使用不可

イベント通知の標準的な振る舞い

表 163 では、イベント取得および通知での標準機構をリストします。

表 163. イベント通知の標準機構

カテゴリーと名前	説明	サポート状況
コネクター・プロパティー		
イベント分配	イベント取得機構には、ポーリング呼び出しを行っているコネクターに関連付けられたイベントのみを処理するフィルターが組み込まれています。この機能では、複数のコネクターが同一のイベント表を使用できるように、ConnectorId フィールドをイベント表に追加する必要があります。また、各コネクターには、ConnectorId コネクター・プロパティーも必要です。このプロパティーは、コネクターの特定のインスタンスの ID を設定し、コネクターがそれに割り当てられたイベントのみを選択できるようにします。150 ページの『イベント分配』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
PollQuantity	コネクターは `PollQuantity` コネクター・プロパティーを使用して、それぞれのポーリング呼び出しごとにコネクターが処理するイベントの最大数を指定します。可能であれば、コネクターは、PollQuantity へのポーリング呼び出し側で取得される行の数を制限します。(例えば、SQL Server では、`set rowcount` オプションを使用してください。) 211 ページの『イベント・レコードの取得』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
イベント表		
イベント状況値	該当する場合には、値はイベント状況で使用されます。 148 ページの表 49 を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
オブジェクト・キー	オブジェクト・キー欄には、名前と値のペアを使用して、新規のビジネス・オブジェクトでデータを設定する必要があります。例えば、ContractId がビジネス・オブジェクト内の属性の名前である場合には、オブジェクト・キーは ContractId=45381 です。コネクターは、区切り文字で区切られた複数の名前と値のペアをサポートする必要があります。区切り文字は構成可能で (PollAttributeDelimiter)、デフォルトはコロン (:) です。133 ページの『オブジェクト・キー』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
オブジェクト名	オブジェクト名フィールドは、正確なビジネス・オブジェクト名に設定する必要があります。132 ページの『イベント・レコードの標準的内容』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
優先順位	優先順位は 0-n です。ここで、0 が最高位の優先順位です。コネクターは、イベントを優先順位ごとにポーリングおよび処理します。減分は行われませんので、注意してください。減分の場合には (通常は起こりえません)、低位優先順位のイベントとなりシャットアウトされます (処理されません)。149 ページの『イベント優先順位によるイベントの処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
各種機能		

表 163. イベント通知の標準機構 (続き)

カテゴリーと名前	説明	サポート状況
アーカイブ	イベントは、コネクターによって処理されると、そのイベントが InterChange Server に正常に配信されるかどうかにかかわらず、アーカイブされます。イベント状況はアーカイブ表に保持され、以下のいずれかとなります。 • Success。イベントは検出されました。また、オブジェクトが作成され、InterChange Server に送信されました。 • Unsubscribed。イベントは検出されました。ただし、コネクターにはそのイベント/動詞の組み合わせに対するサブスクリプションがないため、イベントは InterChange Server に送信されませんでした。 • Error。イベントは検出されましたが、ビジネス・オブジェクト作成のプロセス中か、またはオブジェクトを InterChange Server に通知中のいずれかで、コネクターがそのイベントの処理を試行中にエラーが発生しました。	__ 完全 __ 部分 __ なし __ 使用不可
CDK メソッド `gotApplEvent()`	コネクターは、`pollForEvents()` 内からのみ `gotApplEvent()` を呼び出します。	__ 完全 __ 部分 __ なし __ 使用不可
差分イベント通知	注文明細の追加または削除などの、階層型ビジネス・オブジェクトへの変更点のみを表すイベントを作成することができます。ビジネス・オブジェクト全体での更新イベントは作成しません。[現行では IBM 標準ではありません。]	__ 完全 __ 部分 __ なし __ 使用不可
将来のイベント処理	イベントを処理する将来の日付または時刻を指定するための機構です。指定した日付または時刻まで、コネクターはイベントを処理しません。[現行では IBM 標準ではありません。]	__ 完全 __ 部分 __ なし __ 使用不可
進行中イベント・リカバリー	再始動時に、コネクターはイベント表を検査して、`IN_PROGRESS` 状況を持つイベントが存在するかどうかを調べます。存在する場合には、コネクターは以下のいずれかを行います。 • `PropValue = FailOnStartup`: 致命的エラーをログに記録し、電子メール通知を送信します。 • `PropValue = Reprocess`: InterChange Server にイベントを実行依頼します。 • `PropValue = LogError`: エラーをログに記録しますが、シャットダウンしません。 • `PropValue = Ignore`: イベント表内のこれらのエントリーを無視します。 この振る舞いは、`InDoubtEvents` コネクター・プロパティーを介して構成可能です。コネクターによるこの機能の処理方法を正確に記述するには、「注意事項 (Notes)」フィールドを使用してください。	__ 完全 __ 部分 __ なし __ 使用不可
物理削除イベント	コネクターは、Delete 動詞を使って空のビジネス・オブジェクトを作成し (キー値は取り込まれ、属性の残りは CxIgnore で取り込まれます)、オブジェクトを InterChange Server に送信します。151 ページの『削除イベントの処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可

表 163. イベント通知の標準機構 (続き)

カテゴリーと名前	説明	サポート状況
RetrieveAll	コネクターは、サブスクリプションをデリバリー時に階層型ビジネス・オブジェクト全体を取得します。216 ページの『アプリケーション・データの取得』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
スマート・フィルター	重複イベントはイベント・ストアに保管されません。新規のイベントをレコードとしてイベント・ストアに保管する前に、イベント検出機構によって、その新規のイベントに一致するイベントの存在についてイベント・ストアの照会が行われます。イベント検出機構は、以下のケースでは新規のイベントのレコードを生成しません。 • 新規のイベント内のビジネス・オブジェクト名、動詞、キー、状況、および ConnectorId (該当する場合) がイベント・ストア内の別の未処理イベントの該当する項目と一致する。 • 新規のイベントのビジネス・オブジェクト名、キー、および状況がイベント・ストア内の未処理イベントと一致する。さらに、新規のイベントの動詞が Update で、未処理イベントの動詞が Create である。 • 新規のイベントのビジネス・オブジェクト名、キー、および状況がイベント・ストア内の未処理イベントと一致する。さらに、イベント・ストア内の未処理イベントの動詞が Create で、新規のイベントの動詞が Delete である。この場合には、イベント・ストアから Create レコードを除去する。	__ 完全 __ 部分 __ なし __ 使用不可
動詞安定度	コネクターは、イベント表にある同一の動詞を持つビジネス・オブジェクトを送信します。213 ページの『ビジネス・オブジェクトの名前、動詞、およびキーの取得』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可

一般標準

表 164 では、コネクターの振る舞いに関する一般標準をリストします。

表 164. 一般標準

カテゴリーと名前	説明	サポート状況
ビジネス・オブジェクト		
Foreign key	定義された標準はありません。このプロパティーを使用する場合には、「完全」にチェックマークを付け、その使用方法を記述します。このプロパティーを使用しない場合は、「なし」にチェックマークを付けます。	__ 完全 __ 部分 __ なし __ 使用不可
Foreign key 属性プロパティー	この属性プロパティーを true に設定した場合は、コネクターは値が有効なキーであることを検証します。キーが無効の場合には、コネクターは `FAIL` を戻します。コネクターは、アプリケーションに外部キーがあると想定します。したがって、コネクターは外部キーとマークされたオブジェクトの作成を決して試行しません。	__ 完全 __ 部分 __ なし __ 使用不可
Key	定義された標準はありません。このプロパティーを使用する場合には、「完全」にチェックマークを付け、その使用方法を記述します。このプロパティーを使用しない場合は、「なし」にチェックマークを付けます。	__ 完全 __ 部分 __ なし __ 使用不可

表 164. 一般標準 (続き)

カテゴリーと名前	説明	サポート状況
Max Length	定義された標準はありません。このプロパティーを使用する場合には、「完全」にチェックマークを付け、その使用方法を記述します。このプロパティーを使用しない場合は、「なし」にチェックマークを付けます。	__ 完全 __ 部分 __ なし __ 使用不可
Required	定義された標準はありません。このプロパティーを使用する場合には、「完全」にチェックマークを付け、その使用方法を記述します。このプロパティーを使用しない場合は、「なし」にチェックマークを付けます。	__ 完全 __ 部分 __ なし __ 使用不可
メタデータ主導型の設計	コネクターは、ビジネス・オブジェクト処理がビジネス・オブジェクト定義内のメタデータをベースに行われるため、再コンパイルなしに、新規のビジネス・オブジェクトをサポートすることができます。52 ページの『メタデータ主導型の設計を評価するためのサポート』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
アプリケーションとの接続の切断		
要求処理における接続の切断	コネクターは、ビジネス・オブジェクト要求の処理時に接続エラーを検出し、シャットダウンします。コネクターは致命的エラーをログに記録し、電子メール通知が起動されるように、APPRESPONSETIMEOUT の戻りコードを送信します。88 ページの『アプリケーションとの接続が切断された場合の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
ポーリングにおける接続の切断	コネクターはポーリング呼び出し時に接続エラーを検出し、シャットダウンします。コネクターは致命的エラーをログに記録し、電子メール通知が起動されるように、APPRESPONSETIMEOUT の戻りコードを送信します。88 ページの『アプリケーションとの接続が切断された場合の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
活動停止中の接続の切断	コネクターは、アプリケーションとの接続が切断されるとただちにシャットダウンします。コネクターは致命的エラーをログに記録し、電子メール通知が起動されるように、APPRESPONSETIMEOUT の戻りコードを送信します。	__ 完全 __ 部分 __ なし __ 使用不可
コネクター・プロパティー		
ApplicationPassword	コネクターはこのプロパティー値をパスワードとして使用して、アプリケーションにログインします。	__ 完全 __ 部分 __ なし __ 使用不可
ApplicationUser Name	コネクターはこのプロパティー値をユーザー名として使用して、アプリケーションにログインします。	__ 完全 __ 部分 __ なし __ 使用不可
UseDefaults コネクター・プロパティー	このコネクター・プロパティーが true に設定されている場合には、コネクターは、ビジネス・オブジェクト要求を Create 動詞で処理するとき、JCDK または CDK メソッド initAndValidateAttributes() を呼び出します。	__ 完全 __ 部分 __ なし __ 使用不可
メッセージ・トレース		

表 164. 一般標準 (続き)

カテゴリーと名前	説明	サポート状況
汎用メッセージング	それぞれのオブジェクトごとに使用されるビジネス・オブジェクト・ハンドラーを識別するメッセージ。`gotApplEvent()` または `consumeSync()` のいずれかから、Interchange Server にビジネス・オブジェクトが通知されるごとに、ログに記録するメッセージ。ビジネス・オブジェクト要求が受信されるごとに指示するメッセージ。各トレース・レベル 0 から 5 のトレース・メッセージのガイドラインがその後に続きます。コネクターはトレース設定以下のレベルの、該当するすべてのトレース・メッセージを配信する必要があります。161 ページの『トレース・メッセージ』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 0	0 - コネクターのバージョンを識別するメッセージ。このレベルでは、これ以外のトレースは実行されません。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 1	1 - 処理されたビジネス・オブジェクトごとの、状況メッセージおよび識別 (キー) 情報。`pollForEvents()` メソッドが実行されるごとに、メッセージが送信されます。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 2	2 - コネクターが処理するオブジェクトごとに使用される、ビジネス・オブジェクト・ハンドラーを識別するメッセージ。`gotApplEvent()` または `consumeSync()` のいずれかから、InterChange Server にビジネス・オブジェクトが通知されるごとに、ログに記録するメッセージ。ビジネス・オブジェクト要求が受信されるごとに指示するメッセージ。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 3	3 - 処理されている (該当する場合) 外部キーを識別するメッセージ。これらのメッセージは、コネクターがビジネス・オブジェクト内に外部キーを検出したとき、またはコネクターがビジネス・オブジェクト内に外部キーを設定したときに表示されます。ビジネス・オブジェクト処理に関連するメッセージ。例としては、ビジネス・オブジェクト相互間の一致の検出、子ビジネス・オブジェクトの配列内でのビジネス・オブジェクトの検出があります。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 4	4 - アプリケーション固有の情報を識別するメッセージ。このメッセージの例には、ビジネス・オブジェクトでのアプリケーション固有の情報フィールドを処理する機能によって戻された値があります。エントリーまたは出口機能を識別するメッセージ。これらのメッセージは、コネクターのプロセス・フローのトレースに役立ちます。スレッド固有の処理をトレースするメッセージ。例えば、コネクターが複数のスレッドを作成する場合、新しいスレッドが作成されるたびにメッセージに記録する必要があります。	__ 完全 __ 部分 __ なし __ 使用不可
トレース・レベル 5	5 - コネクターの初期化を指示するメッセージ。このメッセージには、InterChange Server から取得された、それぞれの構成プロパティーの値が含まれています。コネクターが、稼働中に作成する各スレッドの状況を詳述するメッセージ。コネクター・ログ・ファイルには、アプリケーションで実行されたすべてのステートメント、および置換された (該当する場合) すべての変数の値が入っています。ビジネス・オブジェクト・ダンプのメッセージ。コネクターは、処理を開始する前 (コネクターが統合ブローカーから受け取るオブジェクトを示しながら)、およびオブジェクトの処理を完了後に (コネクターが統合ブローカーに戻すオブジェクトを示しながら)、ビジネス・オブジェクトのテキスト表記を出力します。	__ 完全 __ 部分 __ なし __ 使用不可

表 164. 一般標準 (続き)

カテゴリーと名前	説明	サポート状況
メッセージ・トレース	トレースには CDK メソッド `generateMsg()` を使用しないでください。代わりに、トレース・メッセージに応じてメッセージ・ストリングをハードコーディングします。	__ 完全 __ 部分 __ なし __ 使用不可
各種機能		
Java パッケージ名	すべての Java ベースのコネクターは、次のパッケージ・ネーミング標準に従う必要があります。 com.CompanyName.connectors.ConnectorAgentPrefix 例: `com.crossworlds.connectors.XML`	__ 完全 __ 部分 __ なし __ 使用不可
メッセージのロギング	コネクターは、エラー、およびシステムでのトレース・レベル設定には無関係にユーザーが必要とする他の情報をログに記録します。 159 ページの『エラー・メッセージと情報メッセージ』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
CDK メソッド `logMsg()`	`logMsg()` を呼び出す前に、常に CDK メソッド `generateMsg()` を使用します。	__ 完全 __ 部分 __ なし __ 使用不可
NT サービス準拠	NT サービス準拠にするには、`STDOUT` をポイントするメソッドまたは機能 (例えば、C++ での `printf()` メソッド) を使用しないでください。	__ 完全 __ 部分 __ なし __ 使用不可
トランザクション・サポート	ビジネス・オブジェクト要求全体を単一のトランザクションにラップする必要があります。トップレベルのビジネス・オブジェクトおよびその子のすべてを対象の、Create、Update、および Delete トランザクションはすべて、単一トランザクションにラップする必要があります。トランザクションの存続期間中に障害が検出された場合は、トランザクション全体のロールバックが必要です。	__ 完全 __ 部分 __ なし __ 使用不可
特殊な IBM CrossWorlds 値		
CxBlank 処理	Create 操作で、コネクターは、値 CxBlank を持つ属性に対して適切な Blank 値を挿入します。Blank 値は、構成可能であるか、またはアプリケーションに固有である場合があります。197 ページの『Blank 値および Ignore 値の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可
CxIgnore 処理	コネクターは、Create または Update 動詞の処理時に値 CxIgnore とともに渡された属性に対して、アプリケーションで値を設定しません。197 ページの『Blank 値および Ignore 値の処理』を参照してください。	__ 完全 __ 部分 __ なし __ 使用不可

特記事項

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

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

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

プログラミング・インターフェース情報

プログラミング・インターフェース情報は、プログラムを使用してアプリケーション・ソフトウェアを作成する際に役立ちます。

汎用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、IBM Corporation の商標です。

IBM
IBM ロゴ
AIX
CrossWorlds
DB2
DB2 Universal Database
Lotus
Lotus Domino
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

MMX、Pentium および ProShare は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

WebSphere Business Integration Adapter Framework V2.4.1 

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

［サ行］

[タ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X