IBM WebSphere Business Integration Adapters
IBM WebSphere InterChange Server

コネクター開発ガイド (C++ 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、 403 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 C++

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

© Copyright International Business Machines Corporation 1997, 2004. All rights reserved.

© Copyright IBM Japan 2004

目次

本書について . . . ix
対象読者 . . . ix
関連文書 . . . ix
表記上の規則 . . . x
マークアップの規則 . . . xi

本リリースの新機能 . . . xiii
WebSphere InterChange Server v4.2.2 および WebSphere Business Integration Adapters v2.4.1 の新機能 . . . xiii
WebSphere InterChange Server v4.2.2 および WebSphere Business Integration Adapters v2.4.0 の新機能 . . . xiii
WebSphere InterChange Server v4.2.1 および WebSphere Business Integration Adapters 2.3.1 の新機能 . . . xiii
WebSphere Business Integration Adapters 2.2.0 の新機能 . . . xiii
WebSphere Business Integration Adapters 2.1 の新機能 . . . xiv
WebSphere Business Integration Adapters 2.0.1 の新機能 . . . xiv
WebSphere Business Integration Adapters 2.0 の新機能 . . . xv

第 1 部 はじめに . . . 1

第 1 章 コネクター開発の概要 . . . 3
WebSphere Business Integration システムのアダプター . . . 3
コネクター・コンポーネント . . . 8
イベント・トリガー処理フロー . . . 22
アダプター開発用のツール . . . 31
コネクター開発過程の概要 . . . 34

第 2 部 コネクターの作成 . . . 39

第 2 章 コネクターの設計 . . . 41
コネクター開発プロジェクトのスコープ . . . 41
コネクター・アーキテクチャーの設計 . . . 42
アプリケーション固有のビジネス・オブジェクトの設計 . . . 48
イベント通知 . . . 57
オペレーティング・システム間での通信 . . . 58
計画に関する質問のまとめ . . . 59
国際化対応のコネクター . . . 63

第 3 章 汎用コネクター機能の提供 . . . 71
コネクターの実行 . . . 71
コネクターの基底クラスの拡張 . . . 77
エラー処理 . . . 78
コネクター構成プロパティー値の使用 . . . 79
アプリケーションとの接続が切断された場合の処理 . . . 81

第 4 章 要求処理 . . . 83
ビジネス・オブジェクト・ハンドラーの設計 . . . 83
ビジネス・オブジェクト・ハンドラー基底クラスの拡張 . . . 86
要求の処理 . . . 87
動詞アクションの実行 . . . 90
Create 動詞の処理 . . . 91
Retrieve 動詞の処理 . . . 95
RetrieveByContent 動詞の処理 . . . 101

© Copyright IBM Corp. 1997, 2004

Update 動詞の処理. 103
Delete 動詞の処理 111
Exists 動詞の処理 113
ビジネス・オブジェクトの処理 114
コネクター応答の指示 121
アプリケーションとの接続が切断された場合の処理. 122

第 5 章 イベント通知 123
イベント通知機構の概要. 123
アプリケーション用イベント・ストアのインプリメント 124
イベント検出のインプリメント 130
イベント検索のインプリメント 136
ポーリング・メソッドのインプリメント 138
イベント処理における特別な考慮事項 143

第 6 章 メッセージ・ロギング 153
エラー・メッセージと通知メッセージ 153
トレース・メッセージ 155
メッセージ・ファイル 159

第 7 章 C++ コネクターのインプリメント 167
C++ コネクター基底クラスの拡張 167
コネクターの実行開始 168
ビジネス・オブジェクト・ハンドラーの作成 172
イベントのポーリング 204
コネクターのシャットダウン 224
エラーと状況の処理 224

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 227
コネクターの命名 227
コネクターのコンパイル. 228
コネクター定義の作成 230
初期構成ファイルの作成. 233
新規コネクターの始動 234

第 3 部 C++ コネクター・ライブラリー API リファレンス 245

第 9 章 C++ コネクター・ライブラリーの概要. 247
クラス. 247

第 10 章 BOAttrType クラス. 249
属性タイプ定数. 249
メンバー・メソッド 249
BOAttrType(). 250
getAppText() 251
getBOVersion() 251
getCardinality() 252
getDefault() 252
getMaxLength() 252
getName() 253
getRelationType() 254
getTypeName() 254
getTypeNum() 255
hasCardinality() 255
hasName(). 256
hasTypeName() 256

isForeignKey() 257
isKey() 257
isMultipleCard() 258
isObjectType() 258
isRequired() 259
isType() 259

第 11 章 BOHandlerCPP クラス 261
BOHandlerCPP() 262
doVerbFor() 262
generateAndLogMsg() 264
generateAndTraceMsg() 265
generateMsg() 266
getConfigProp() 267
getTheSubHandler() 268
logMsg() 269
traceWrite() 269

第 12 章 BusinessObject クラス 271
属性値定数 271
メンバー・メソッド 271
BusinessObject() 272
clone() 273
doVerbFor() 274
dump() 275
getAttrCount() 275
getAttrDesc() 276
getAttrName() 277
getAttrType() 277
getAttrValue() 278
getBlankValue() 279
getDefaultAttrValue() 280
getIgnoreValue() 281
getLocale() 281
getName() 282
getParent() 282
getSpecFor() 282
getVerb() 283
getVersion() 284
initAndValidateAttributes() 284
isBlank() 286
isBlankValue() 286
isIgnore() 287
isIgnoreValue() 287
makeNewAttrObject() 288
setAttrValue() 288
setDefaultAttrValues() 289
setLocale() 290
setVerb() 290

第 13 章 BusObjContainer クラス 293
getObject() 294
getObjectCount() 294
getTheSpec() 295
insertObject() 295
removeAllObjects() 296

removeObjectAt() 296
setObject() 297

第 14 章 BusObjSpec クラス 299
getAppText() 300
getAttribute() 300
getAttributeCount() 301
getAttributeIndex() 301
getMyBOHandler() 302
getName() 302
getVerbAppText() 303
getVersion() 303
isVerbSupported() 304

第 15 章 CxMsgFormat クラス 305
メッセージ・タイプ定数 305
メソッド 305
generateMsg() 305
使用すべきでないメソッド 307

第 16 章 CxVersion クラス 309
CxVersion() 309
compareMajor() 310
compareMinor() 311
comparePoint() 311
compareTo() 312
getDELIMITER() 312
getLATESTVERSION() 313
getMajorVer() 313
getMinorVer() 314
getPointVer() 314
setMajorVer() 315
setMinorVer() 315
setPointVer() 315
toString() 316

第 17 章 GenGlobals クラス 317
GenGlobals() 318
executeCollaboration() 318
generateAndLogMsg() 319
generateAndTraceMsg() 321
generateMsg() 322
getBOHandlerforBO() 323
getCollabNames() 324
getConfigProp() 324
getEncoding() 325
getLocale() 326
getTheSubHandler() 327
getVersion() 327
init() 328
isAgentCapableOfPolling() 329
logMsg() 330
pollForEvents() 331
terminate() 332
traceWrite() 333
使用すべきでないメソッド 334

第 18 章 ReturnStatusDescriptor クラス 337
getErrorMsg() 337
getStatus() 338
seterrMsg() 338
setStatus() 339

第 19 章 SubscriptionHandlerCPP クラス 341
SubscriptionHandlerCPP() 341
gotApplEvent() 342
isSubscribed() 344

第 20 章 StringMessage クラス 347
hasMoreTokens() 347
nextToken() 347
使用すべきでないメソッド 348

第 21 章 Tracing クラス 349
トレース・レベル定数 349
メソッド 349
getIndent() 350
getName() 350
getTraceLevel() 350
setIndent() 351
write() 351

付録 A. コネクターの標準構成プロパティー 353
新規プロパティーと削除されたプロパティー 353
標準コネクター・プロパティーの構成 353
標準プロパティーの要約 355
標準構成プロパティー 359

付録 B. Connector Configurator 373
Connector Configurator の概要 373
Connector Configurator の始動 374
System Manager からの Configurator の実行 375
コネクター固有のプロパティー・テンプレートの作成 375
新規構成ファイルの作成 378
既存ファイルの使用 379
構成ファイルの完成 381
構成ファイル・プロパティーの設定 381
構成ファイルの保管 388
構成ファイルの変更 389
構成の完了 389
グローバル化環境における Connector Configurator の使用 390

付録 C. コネクター・スクリプト生成プログラム 391

付録 D. コネクター機能チェックリスト 393
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 393
要求処理の標準的な振る舞い 393
イベント通知の標準的な振る舞い 395
一般標準 398

特記事項 403
プログラミング・インターフェース情報 405
商標 405

索引 407

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 システムで C++ コネクターを開発する方法について説明します。

対象読者

この資料の対象読者は、コネクターの開発者です。本書を理解するには、C++ プログラミング言語についての知識が必要です。この資料では、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、C++ コネクターの開発について説明します。 Java コネクターの開発については、「*コネクター開発ガイド (Java 用)*」を参照してください。

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。

© Copyright IBM Corp. 1997, 2004

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

- Message Broker (WebSphere MQ Integrator、WebSphere MQ Integrator Broker、および WebSphere Business Integration Message Broker) 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/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	初出語を示します。
イタリック、***イタリック***	変数名または相互参照を示します。
青のアウトライン	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、名前の個々の要素を不等号括弧で囲むと、これらの要素を互いに区別できます。例えば、`<server_name><`*`connector_name`*`>tmp.log` のように記述します。
/、¥	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

章によっては、本文が次のマークアップで囲まれている場合があります。

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 Adapters 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、C++ 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。

- C++ 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。
- C++ コネクターでは、Cayenne ライブラリーの代わりに STL ライブラリーが使用されるようになりました。これにより、C++ コネクターのソース・ファイルに `cayenne_include` ディレクトリーをインクルードする必要がなくなりました。
- C++ Connector Development Kit (CDK) は、より一貫した方法で C++ 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も提供します。詳細については、 234 ページの『新規コネクターの始動』を参照してください。

WebSphere InterChange Server v4.2.1 および WebSphere Business Integration Adapters 2.3.1 の新機能

IBM WebSphere InterChange Server 4.2.1 リリースおよび IBM WebSphere Business Integration Adapter 2.3.1 リリースは、C++ 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。

- setLocale() メソッド (`BusinessObject` 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。

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、C++ 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。

© Copyright IBM Corp. 1997, 2004

- 「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の名称について、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。
- 現在、C++ コネクター・ライブラリーでは重複イベント回避機能をサポートしており、保証付きイベント・デリバリーが提供されています。重複イベント回避機能は、JMS キューとしてインプリメントされていない イベント・ストアをもつ JMS 対応アダプターでよく使用されます。`DuplicateEventElimination` コネクター・プロパティーを使用して、この機能を使用可能にできます。詳細については、 149 ページの『JMS 以外のイベント・ストアを使用するコネクターの保証付きイベント・デリバリー』を参照してください。
- 41 ページの『第 2 章 コネクターの設計』には、コネクターを国際化対応にするための方法について詳細が説明されています。
- 227 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』には、以下の内容を含む、C++ コネクターを WebSphere Business Integration システムに追加する方法の詳細が説明されています。
  - コネクターの初期構成ファイルの作成方法
  - サンプル始動ファイルから C++ コネクターの始動スクリプトを作成する方法
  - システム変数設定のために新規の `CWConnEnv.bat` (Windows) または `CWConnEnv.sh` (UNIX) ファイルを使用する方法
- C++ コネクター・ライブラリーでは、戻り状況記述子の状況値へのアクセスを提供するため、`ReturnStatusDescriptor` クラスで新規の 2 つのメソッドをサポートするようになりました。
  - `getStatus()`
  - `setStatus()`

WebSphere Business Integration Adapters 2.1 の新機能

IBM WebSphere Business Integration Adapter 2.1 リリースで加えられた変更による本書の内容への影響はありません。

WebSphere Business Integration Adapters 2.0.1 の新機能

IBM WebSphere Business Integration Adapter 2.0.1 リリースは、国際化対応版の C++ コネクター・ライブラリーを提供しています。この国際化対応のコネクター・ライブラリーを使用すると、さまざまなロケール用にローカライズ可能なアダプターを開発できます (ロケールには、国/地域別情報と文字コード・セットが含まれます)。ロケールに対応するため、コネクターの構造は、次のように変更されています。

- コネクター・フレームワークにロケールが関連付けられました。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から決定されます。 C++ コネクター・ライブラリーが `GenGlobals` クラス内に `getEncoding()` メソッドと `getLocale()` メソッドを提供しています。これにより、コネクター内からこの情報にアクセスすることができます。

- ビジネス・オブジェクトにロケールが関連付けられました。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。ビジネス・オブジェクト定義名やその属性名には関連付けられていません。C++ コネクター・ライブラリーが `BusinessObject` クラス内に `getLocale()` メソッドを提供しています。これにより、コネクター内からこのロケールの名前を取得できます。

詳細については、 63 ページの『国際化対応のコネクター』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実行する 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造も、このリリースで変更されました。以下のガイドを統合して、C++ コネクターの開発に関する 1 冊の資料にまとめました。

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: C++ Class Library	C++ コネクター・ライブラリーに関する参考資料は、第 3 部に変更されました。

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、C++ Connector Development Kit (CDK) についても紹介し、コネクターのインプリメント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。この章は、以下のセクションから構成されています。

- 『WebSphere Business Integration システムのアダプター』
- 8 ページの『コネクター・コンポーネント』
- 22 ページの『イベント・トリガー処理フロー』
- 31 ページの『アダプター開発用のツール』
- 34 ページの『コネクター開発過程の概要』

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・アプリケーション間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することです。 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

© Copyright IBM Corp. 1997, 2004

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、実際にこのような通信を提供します。

- 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。

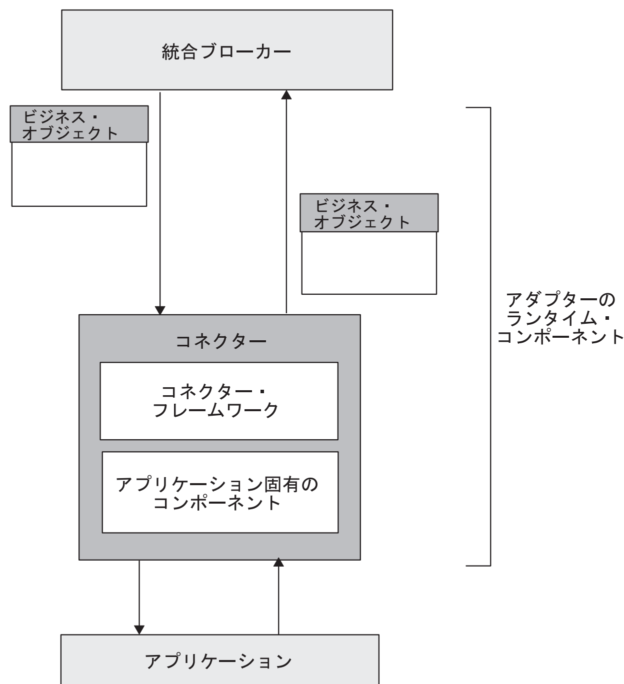

図 1. 情報転送を提供するアダプター・コンポーネント

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。

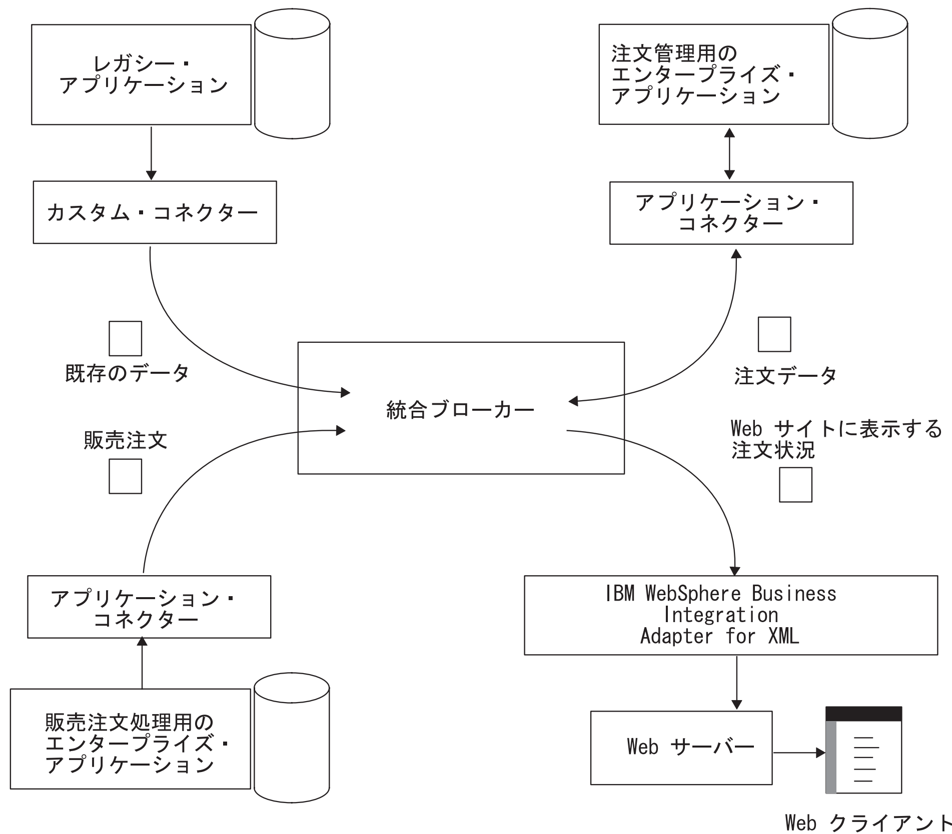

図 2. WebSphere Business Integration システム

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、1 つの集合単位として扱うことのできる一連の属性を表します。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、検索、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、 114 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「検索」

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に関する情報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。

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」という修飾子は、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
25 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
28 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に対して変更を要求した場合、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している場合、コネクターの動作は以下のとおりです。 • 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 インプリメントしてください。イベント通知のインプリメンテーションはオプションです (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・データの検索や変更など、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、C++ コネクターのコンポーネントを示します。この図には、C++ 環境と Java 環境との間でビジネス・オブジェクトを変換する C++ 変換層が記載されています。この図には、コネクター・フレームワークが提供する汎用サービスの C++ コネクター・ライブラリーも示してあります。

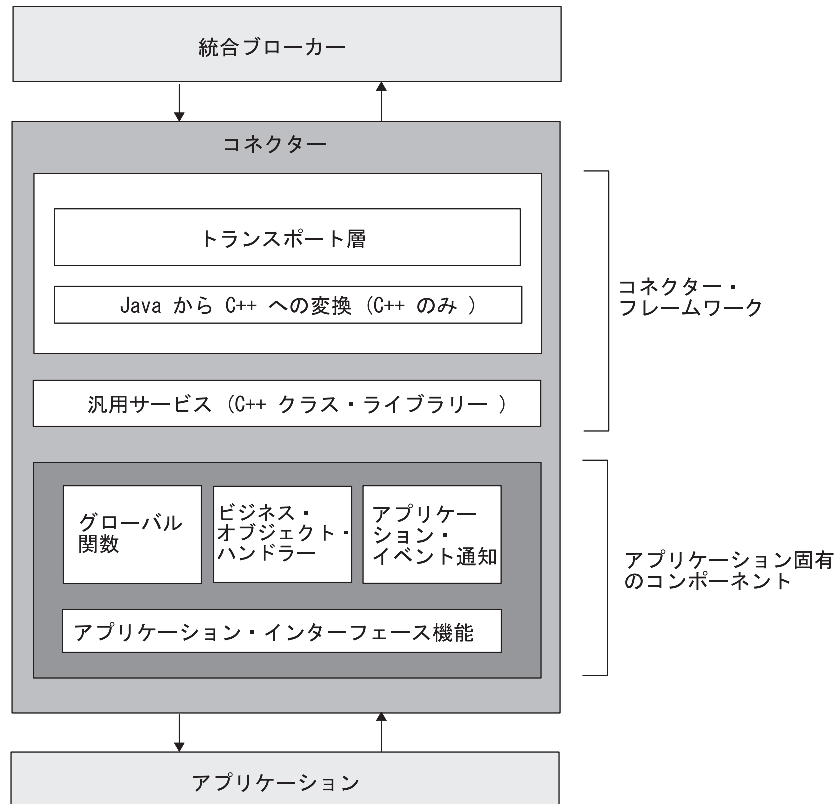

図 3. C++ コネクターのコンポーネント

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

統合ブローカーに 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 システムでは、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 は、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ています。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。

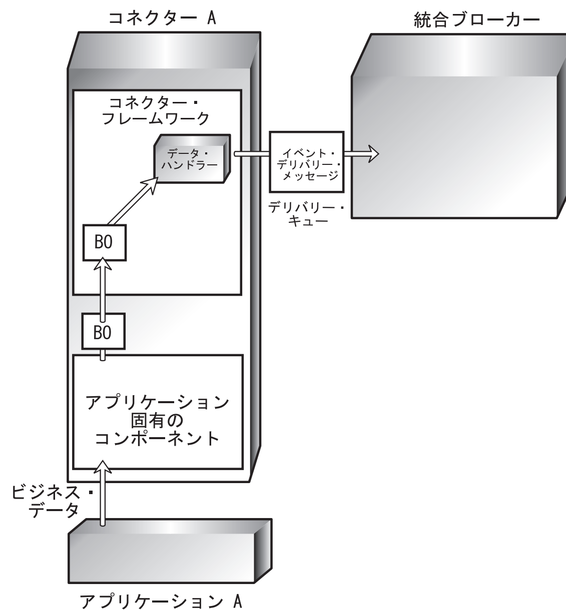

図 4. WebSphere Message Broker を使用した上位のコネクター・アーキテクチャー

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 で実行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。
C++ 変換層	• C++ 環境と Java 環境との間でビジネス・オブジェクトを変換する Java から C++ への変換層が用意されています。
21 ページの『C++ コネクター・ライブラリー』ページの C++ 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、C++ 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実行します。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、C++ 変換層、および C++ 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 の内部で実行します。

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実行し、各コネクターは別個のプロセスとして実行します。

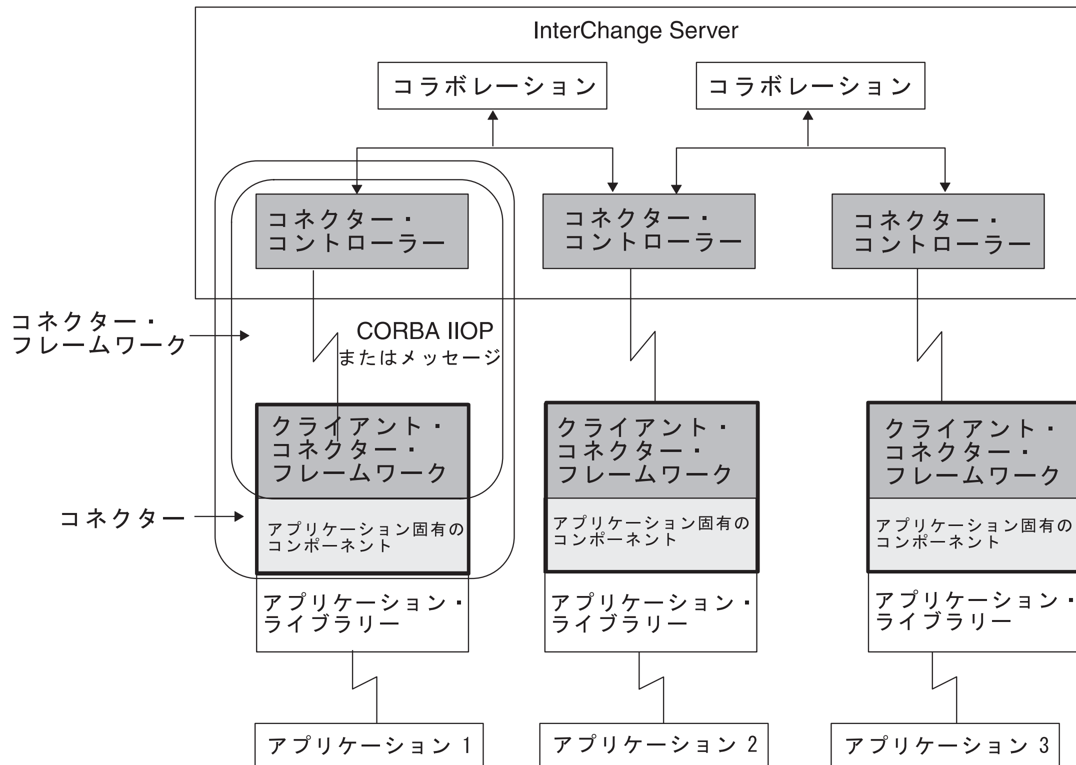

図 5. WebSphere InterChange Server を使用した上位のコネクター・アーキテクチャー

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、29 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・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・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。クライアント・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、コネクター・コントローラーは、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。

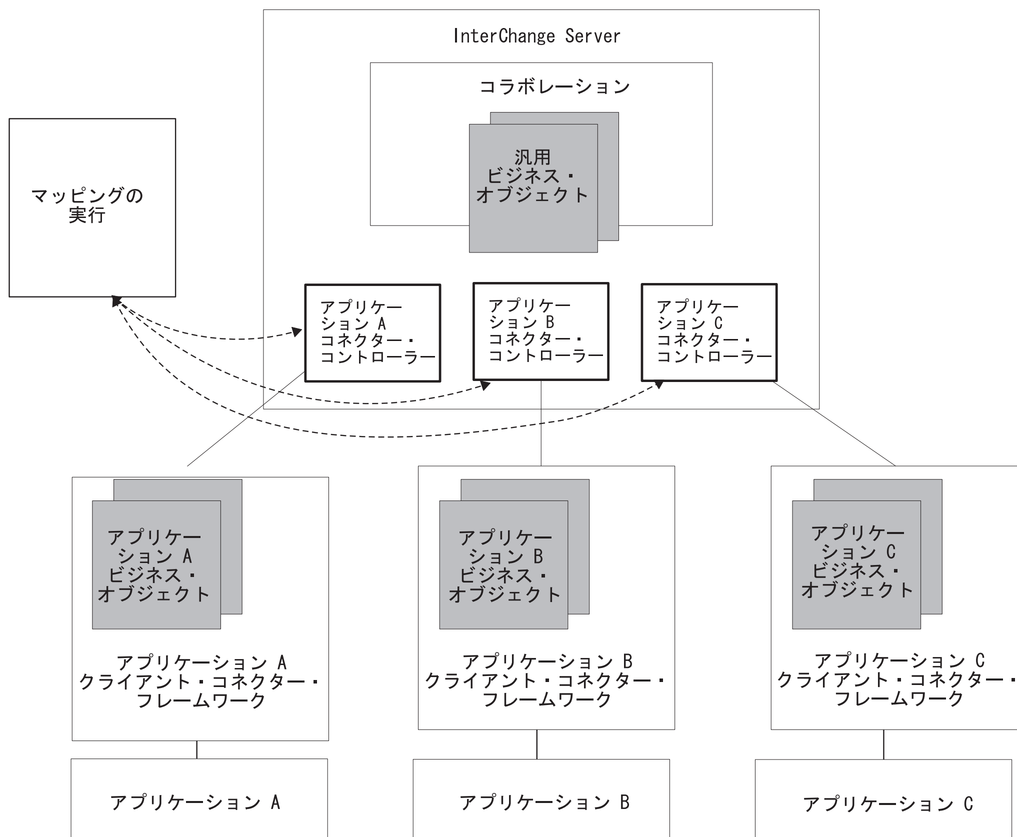

図 6. *InterChange Server System* でのマッピング

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・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、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がどれかを確認します。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、コラボレーションがサブスクライブした先のビジネス・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・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・コネクター・フレームワークの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。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。

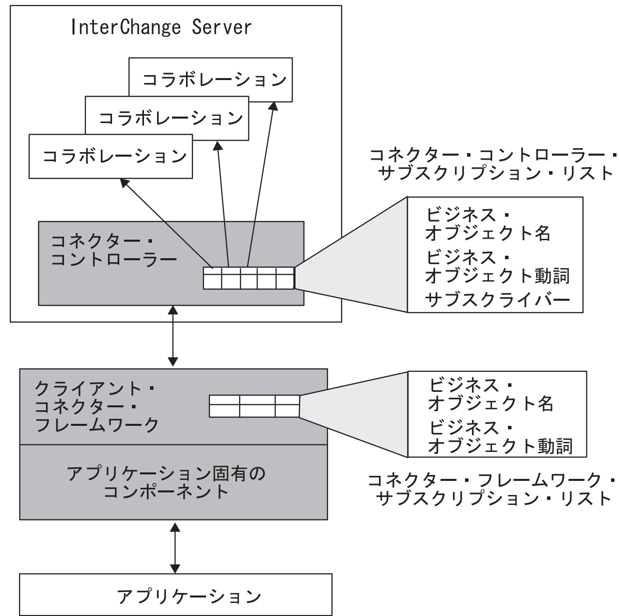

図 7. サブスクリプション処理

サブスクリプションの詳細については、28 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します。

メッセージ・サービス	説明
28 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。
25 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、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。

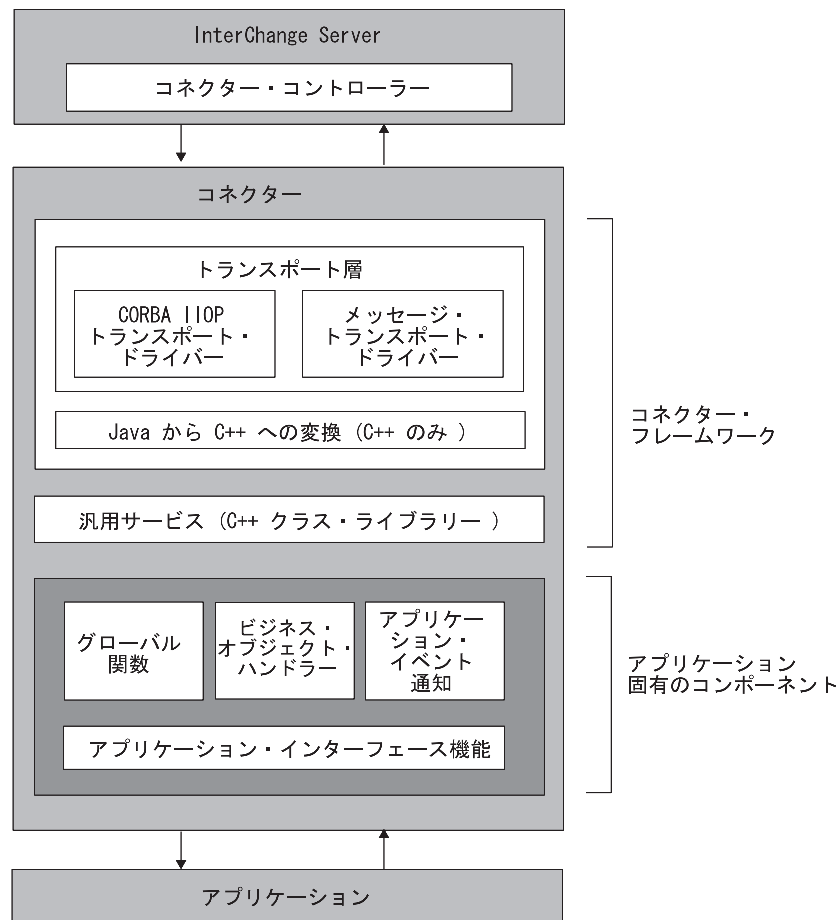

図 8. InterChange Server との通信に対応したコネクター・アーキテクチャー

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・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セットです。ORB により、複数のオブジェクトが、始動時に互いを検索したり、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検索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、状態の検索なども含まれます。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。

- ビジネス・オブジェクトを、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送信する場合も、トランスポート層は CORBA を使用します。

  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・デリバリーに 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デリバリーできるので、サブスクリプション・デリバリーの効率を改善できます。CORBA を通信機構として採用すると、帯域幅の広い 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・デリバリーを実行できるので、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インプリメントします。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 には、CORBA 通信機構を示します。

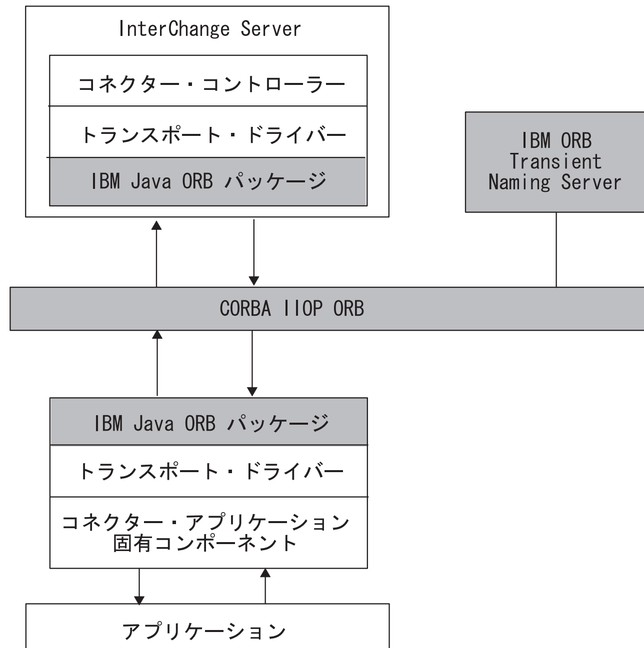

図 9. CORBA IIOP によるコネクター内部での通信

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、 353 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。

C++ 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 C++ コネクター・ライブラリーが含まれています。これは、コネクター開発用の汎用サービスおよびユーティリティーを提供します。 C++ 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。目的の出力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: C++ 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、 247 ページの『第 9 章 C++ コネクター・ライブラリーの概要』を参照してください。

以下のように、C++ コネクター・ライブラリーは、C++ コネクターの実行をサポートするために Windows と UNIX の両方のオペレーティング・システムで使用可能です。

- Windows システムでは、C++ コネクター・ライブラリーは `CwConnector.dll` というダイナミック・リンク・ライブラリー (DLL) です。これは以下のディレクトリーにあります。

  ProductDir¥bin

開発バージョンのこのライブラリーは、Connector Development Kit for C++ (CDK) に組み込まれています。

重要: CDK は Windows システムでのみ サポートされます。詳細については、34 ページの『Connector Development Kit』を参照してください。

- UNIX ベースのシステムでは、C++ コネクター・ライブラリーは `libCwConnector` という共用ライブラリーです。ファイル拡張子は、使用している UNIX システムに応じて異なります。この共用ライブラリー・ファイルは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・イベント検出と応答の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 C または C++ 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 C++ で記述し、C++ 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。実行時には、アプリケーション固有のコンポーネントは、コネクター・フレームワークの Java クラスから呼び出されます。

イベント・トリガー処理フロー

C++ 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)。

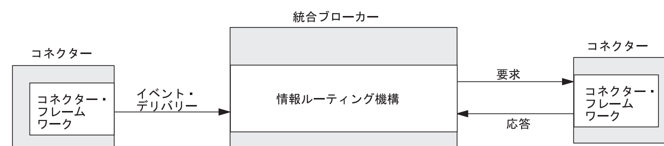

図 10. WebSphere Business Integration システムのイベント・トリガー処理フロー

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、*トリガー側イベント* 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、C++ コネクター・ライブラリーの `gotApplEvent()` メソッドを呼び出して、トリガー側の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、トリガー側の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側のイベントを受信し、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、次にトリガー側の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実行します。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、必要に応じて結果の汎用ビジネス・オブジェクトから適切な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を実行します。コネクター・コントローラーは次に、*サービス呼び出し応答*への変換を実行し、コネクター・コントローラーへイベント応答を送信します。そのコネクター・コントローラーは、応答をコラボレーションに戻します。

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。
- 28 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』
- 26 ページの『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、*パブリッシュと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部のエンティティーが変更された時点を確認できるようになります。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・データを検索して、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、 123 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あり、コネクターが処理するまで、ここにイベント・レコードを保管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・メカニズムを介してイベント・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・レコードを検索して、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ある場合にだけ 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。

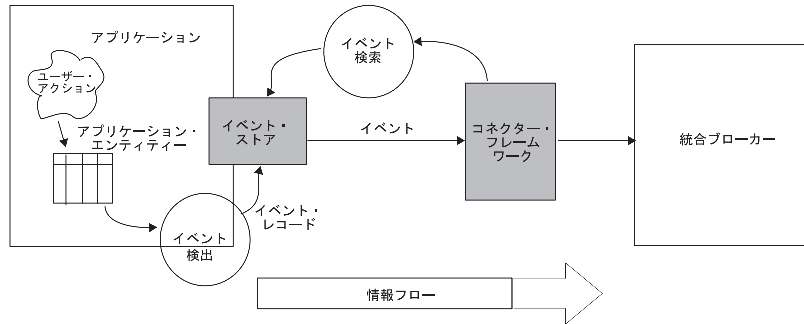

図 11. イベント検出および検索

要求処理

コネクターには、アプリケーション・イベント検出以外に、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、検索、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・コントローラーからビジネス・オブジェクトを受け取ります (コネクター・コントローラーは、コラボレーションのサービス呼び出しからビジネス・オブジェクトを受け取ります)。

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、 83 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、29 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、30 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』
- 30 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進めます。

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 C++ 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、検索、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 `getBOHandlerforBO()` メソッド (コネクター基底クラスで定義) を呼び出し、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生成します。
- `getBOHandlerforBO()` 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。

`getBOHandlerforBO()` 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、統合ブローカーへの応答に戻ることができます。

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。応答先へのメッセージ要求は、応答メッセージの相関 ID によって識別されます。

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

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 (続き)

アダプター・コンポーネント	構成ツール	API
Object Discovery Agent (ODA)	Business Object Designer	Object Discovery Agent Development Kit (ODK)
コネクター	Connector Configurator	C++ 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 開発キット* (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、以下の製品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	C++ 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、コネクターを構成するグラフィック・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・デリバリー用)

このグラフィック・ツールは、Windows 2000 および Windows XP 上で実行します。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、 373 ページの『付録 B. Connector Configurator』を参照してください。

Adapter Development Kit

Adapter 開発キット (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 開発キットは、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* システムでしか 使用できない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
C++ Connector Development Kit (CDK)	C++ コネクターのサンプル・コードを提供します。	`cdk`

Connector Development Kit: ADK に入っている C++ Connector Development Kit (CDK) では、コネクターの開発に使用するためのコンポーネントが提供されています。 CDK 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¥cdk
```

表 12 に、cdk ディレクトリーのサブディレクトリーの内容を示します。

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
C++ コネクター・ライブラリー	開発バージョンの C++ コネクター・ライブラリーを提供します。このライブラリーは、C++ コネクターの作成時に使用する必要があります。詳細については、 247 ページの『第 9 章 C++ コネクター・ライブラリーの概要』を参照してください。	lib
コード・サンプル	シンプルな C++ コネクターのサンプル・コード	samples
コネクター・クラスのヘッダー・ファイル	C++ コネクター・ライブラリーのクラスについての定義が記述されています。	generic_include

CDK 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、C++ コネクターの開発に利用できます。

```
DevelopmentKits¥cdk¥samples
```

CDK は Windows システムでのみサポートされます。C++ コネクターをコンパイルするには、Microsoft Visual C++ 6.0 開発環境のコンパイラーを使用してください。詳細については、 228 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、Java 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 Java 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 (Java 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。

コネクター開発過程の概要

このセクションでは、コネクター開発過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。
2. コネクターの設計およびインプリメント。

開発環境のセットアップ

開発過程を開始する前に、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の方法については、「*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 のドキュメンテーション・セット内) 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。

- 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、 228 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 が実行していること。

  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、 227 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・ソフトウェアを実行させておく必要があります。

コネクター開発の各段階

コネクター開発過程の一部として、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、コネクターの開発過程全体には、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、コネクター開発過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インプリメントする。
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、この機構をインプリメントして、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インプリメントする。
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. WebSphere MQ をコネクター・コンポーネント間でのメッセージングに使用する場合は、そのコネクター用のメッセージ・キューを追加する。
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、コネクターの開発過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検索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。

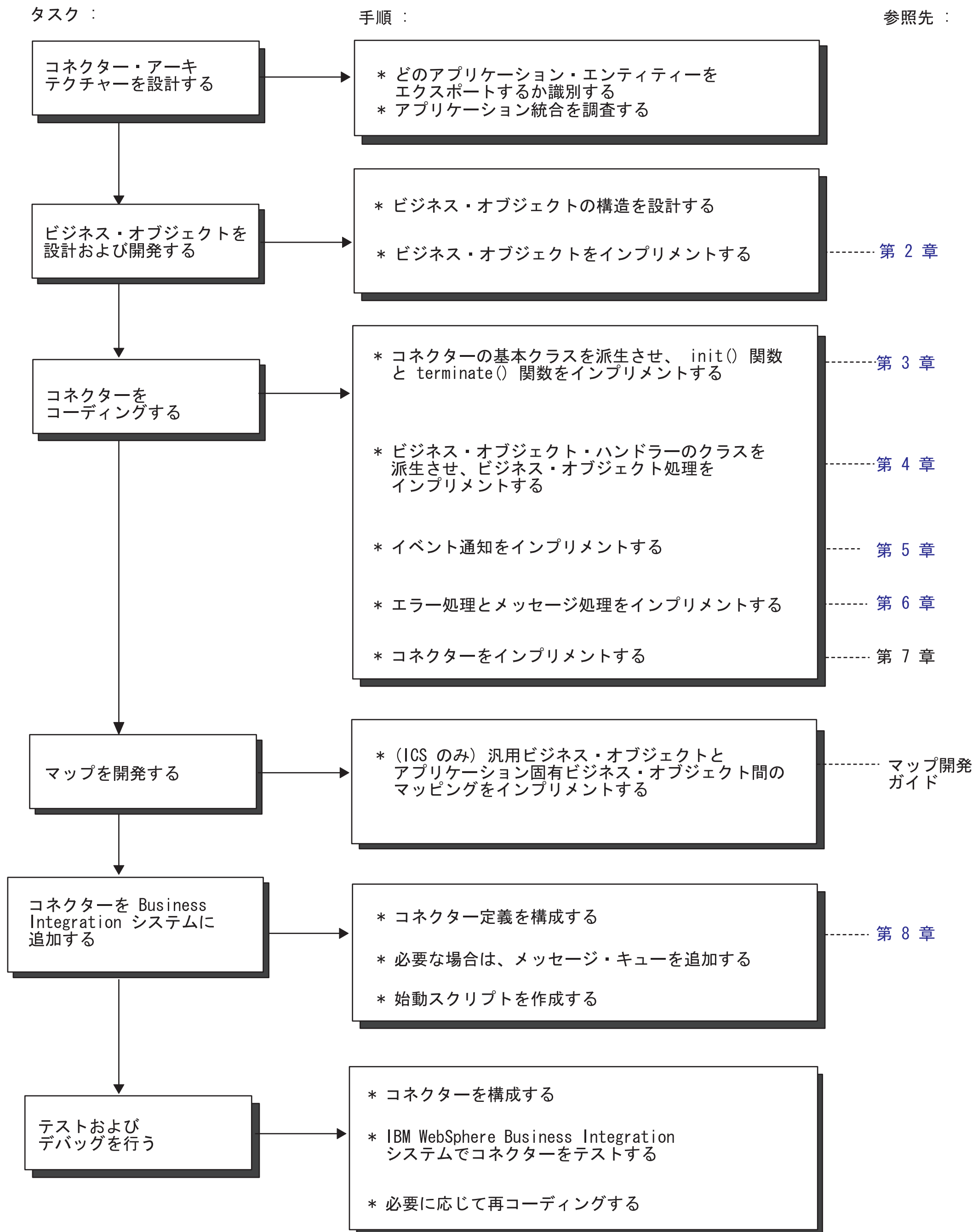

図 12. C++ コネクター開発工程の概要

第 2 部 コネクターの作成

© Copyright IBM Corp. 1997, 2004

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、その後のインプリメンテーション段階での問題発生を防止することにつながります。この章は、以下のセクションから構成されています。

- 『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』
- 63 ページの『国際化対応のコネクター』

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 C++ 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タスクは、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。コネクター設計の複雑さや、コネクターのインプリメンテーションに必要な時間は、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、コネクター開発タスク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、コネクター開発タスク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

© Copyright IBM Corp. 1997, 2004

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	詳細情報の参照先
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。

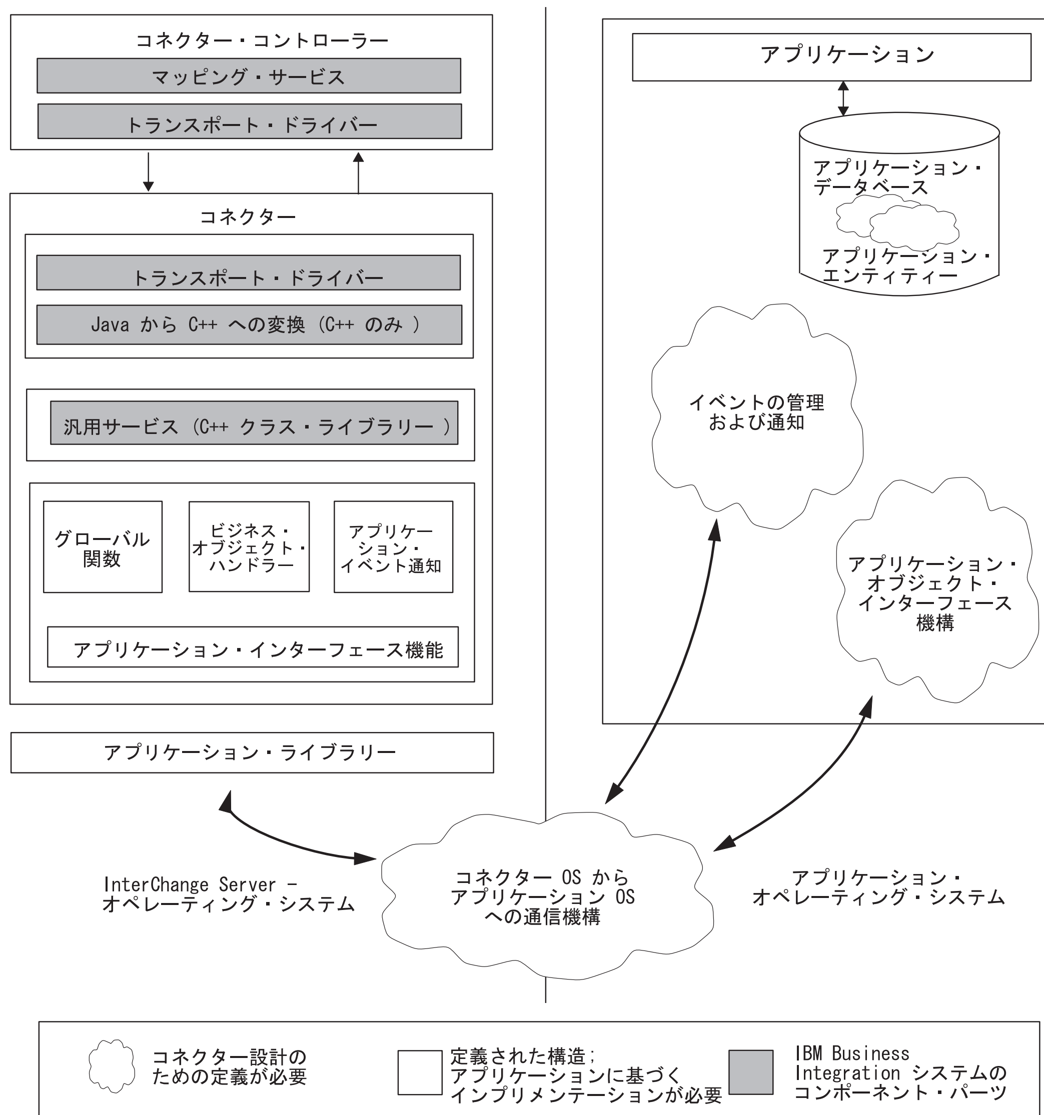

図 13. コネクター設計に影響のあるアプリケーションの領域

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実行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?

アプリケーションの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実行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、28 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、25 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インプリメントされていますが、イベント通知機能はインプリメントされていません。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、123 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『以前の統合作業実績の検討』— 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判断』— 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『バッチ終結処理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。

以前の統合作業実績の検討

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検索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検索するインターフェースを使用しますか? 使用する場合は、情報を変更または検索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判断

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、検索、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 のインプリメンテーションに不整合はありますか? Create/Retrieve/Update/Delete への移動は、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。

バッチ終結処理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終結処理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、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などができます。

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終結処理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。

バッチ終結処理タスクは、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、計画タスクでは、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、コネクター開発過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 Collaborations 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インプリメント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。

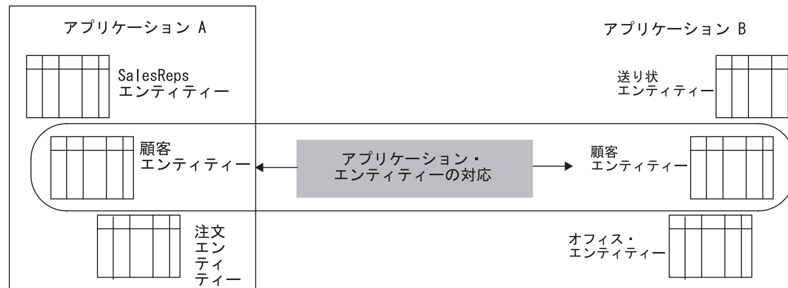

図 14. 設計の主軸: 統合先のアプリケーションの決定

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「一対多」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:

- タイプは同じだが、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。どちらもタイプは 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 タイプごとに異なります。

  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、ビジネス・オブジェクトが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、同じタイプ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、2 つの汎用の子ビジネス・オブジェクト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。

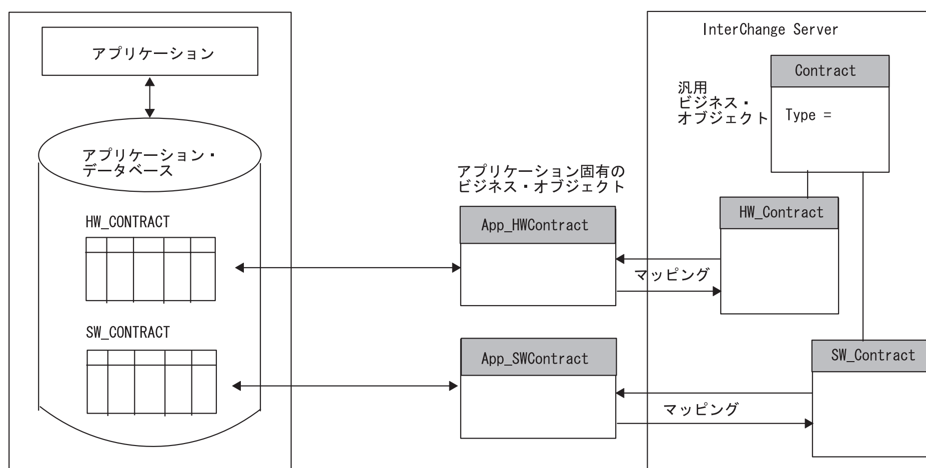

図 15. アプリケーション・エンティティーのデータベース表現

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)**	**必須のデリバリー動詞 (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、テーブル・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テーブル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検索できるので、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	**必須のビジネス・オブジェクト・ハンドラー**	**詳細情報の参照先**
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長所と欠点については、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、テーブル、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セマンティック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。

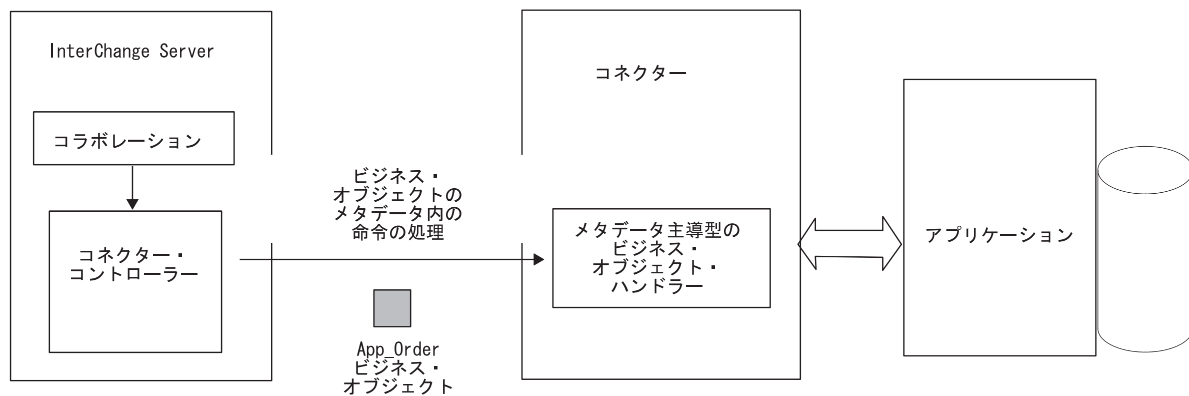

図 16. ビジネス・オブジェクトのメタデータの使用による処理命令への対応

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、 84 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、メタデータ主導型の手法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を示します。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。

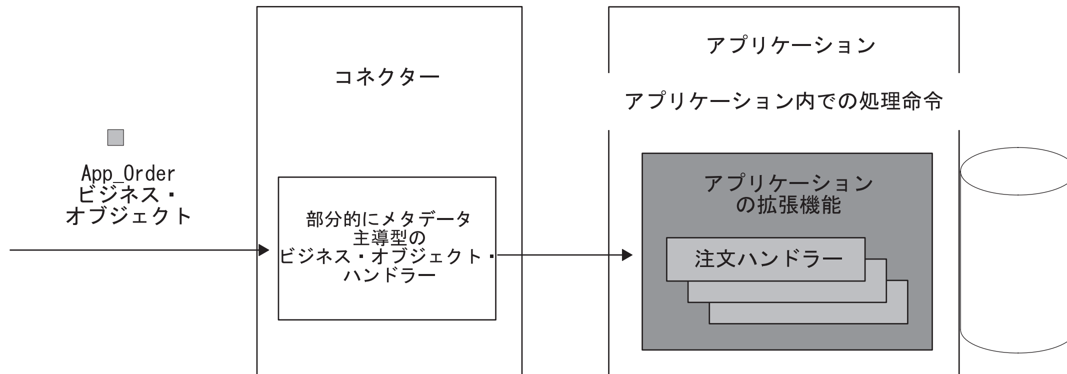

図 17. アプリケーション内部でのアプリケーション固有の処理

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・オブジェクトにふさわしいビジネス・オブジェクト・ハンドラーを呼び出します。

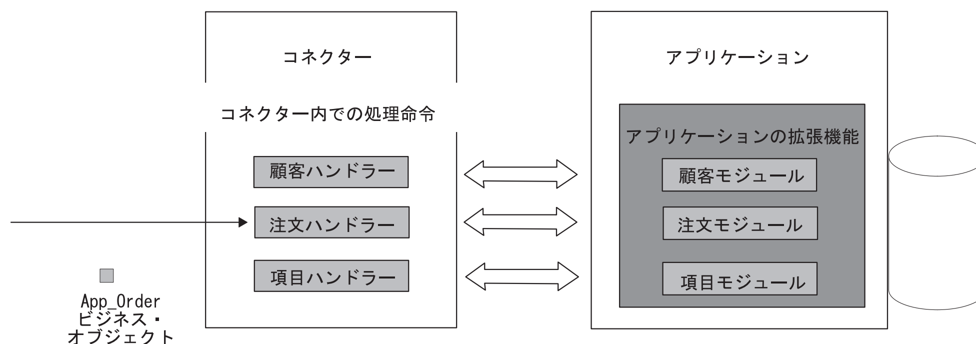

図 18. コネクター内部でのアプリケーション固有の処理

この非メタデータ手法の欠点は、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、イベントのタイプ、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検索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・データを検索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インプリメント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・ロジックに対する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すでに存在する場合は、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、 123 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・システムで実行する場合は、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 上で実行し、コネクターと InterChange Server が Windows 2000 上で実行する場合、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、複数のオペレーティング・システムにまたがって通信できます。

図 19 には、Windows 2000 上で実行する ODBC コネクターと、UNIX 上で実行する 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 を介してこ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。

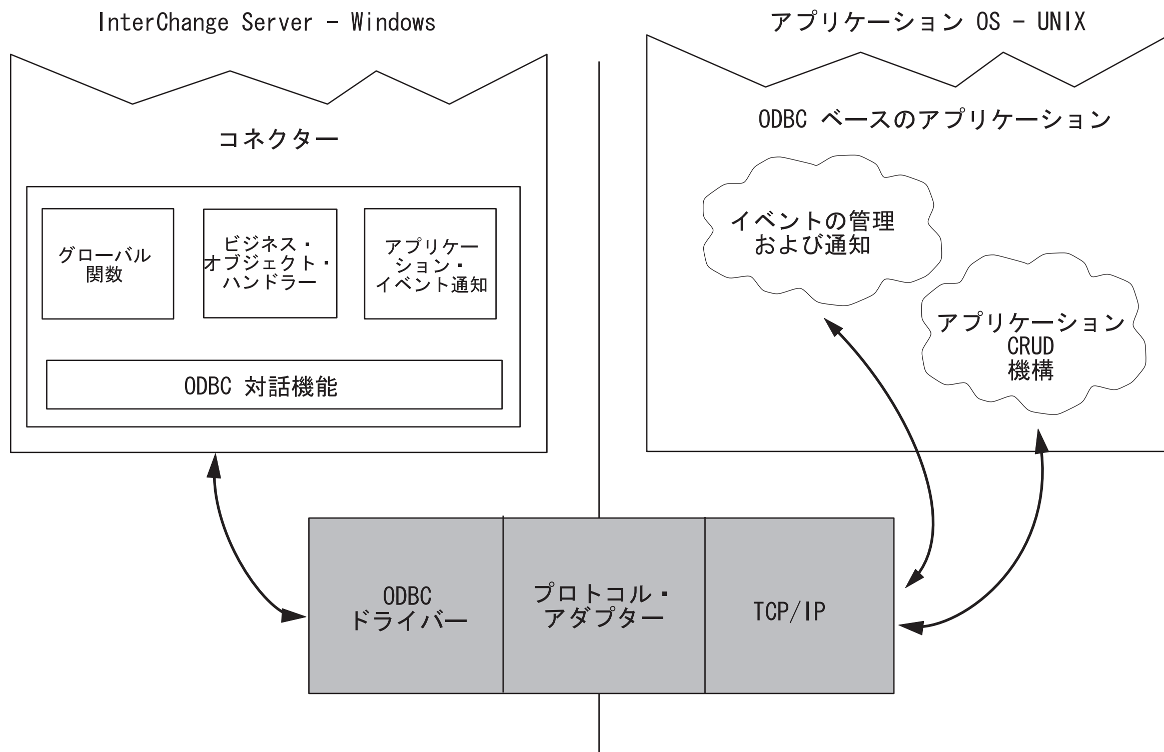

図 19. Windows - UNIX 間通信の例

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?
   - タイプは同じだが、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、情報を変更または検索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、検索、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 のインプリメンテーション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終結処理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 は、サンプル・コネクターに関する情報の一部を示したものです。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 上で実行します。
   - 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 Role
   - 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. 結果レポートの例

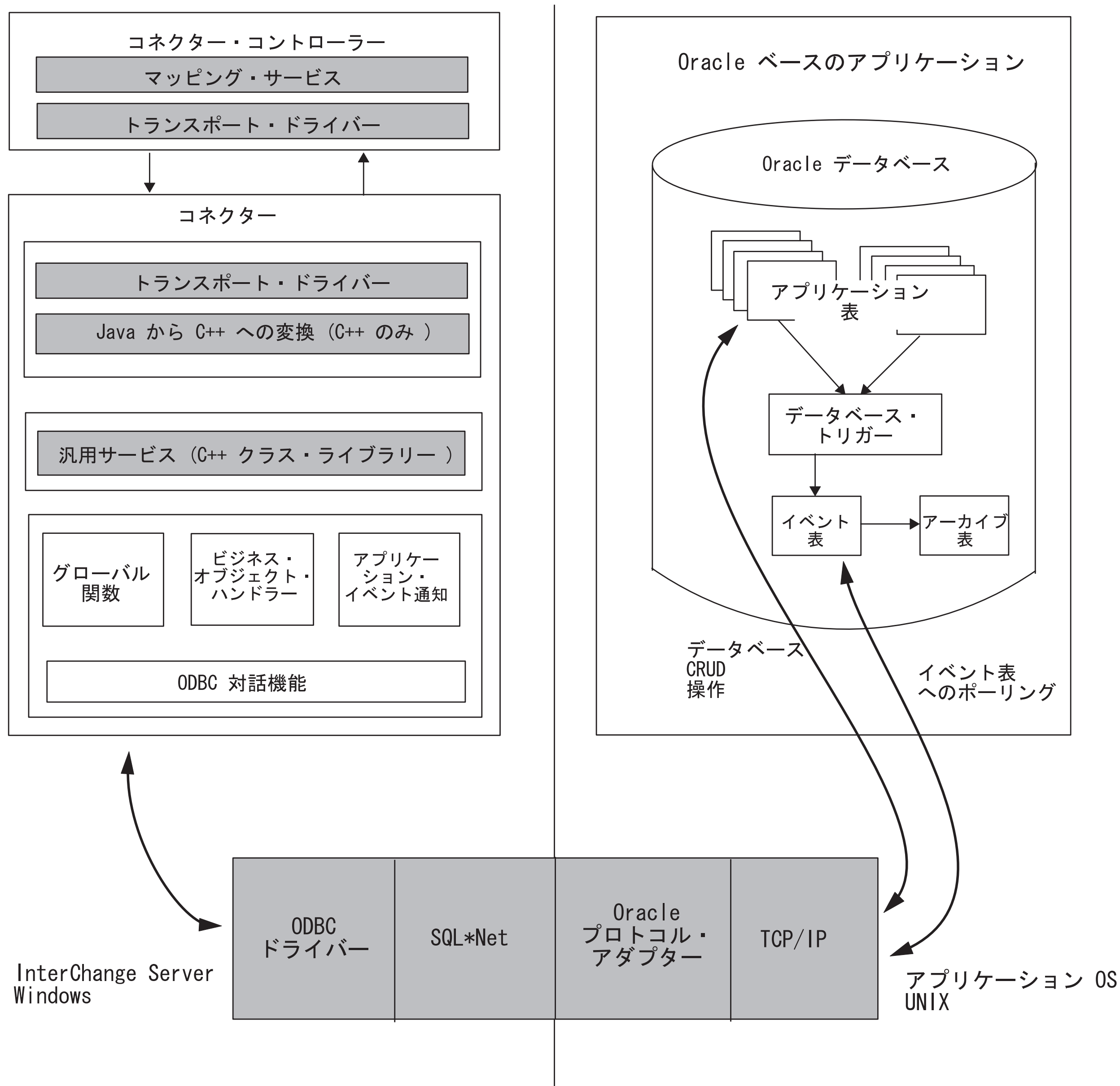

図 22. ODBC ベースのコネクター・アーキテクチャーの例

国際化対応のコネクター

国際化対応のコネクター 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作成されたコネクターです。*ロケール* は、ユーザー環境の一部です。エンド・ユーザーの国、言語、地域に固有のデータの処理方法について、情報を集めたものです。通常、ロケールは、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。ロケール依存データを扱う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) と呼びます。国際化対応のコネクターを特定のロケール用に作成することを、コネクターの*ローカリゼーション* (L10N) と呼びます。

このセクションでは、国際化対応のコネクターについて説明します。内容は次のとおりです。

- 『ロケールとは』
- 『国際化対応のコネクターにおける設計上の考慮事項』

ロケールとは

ロケール は、ユーザー環境に対して、次の情報を提供します。

- 言語および国 (または地域) に応じた国/地域別情報:
  - データ形式:
    - 日付: 曜日および月の完全名と省略名、日付の構造 (日付区切り文字を含む)。
    - 数値: 3 桁ごとの区切り記号、小数点記号、およびこれらの記号を数値内のどこに配置するのかを定義します。
    - 時刻: 12 時間の標識 (AM や PM などの標識)、および時刻の構造を定義します。
    - 通貨の値: 数値記号、通貨記号、およびこれらの記号を通貨の値内のどこに配置するのかを定義します。
  - 照合順序: 特定の文字コード・セットおよび言語におけるデータのソート方法を定義します。
  - ストリング処理には、文字の大文字小文字の比較、サブストリング、および連結などのタスクが含まれます。
- *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 としてエンコードされます。EBCIDIC 文字セットでは、同じ文字が 43 としてエンコードされます。*文字コード・セット* には、1 つまたは複数の言語文字体系のすべての文字について、エンコードが含まれます。

ロケール名のフォーマットは、以下のとおりです。

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・セットの名前です。一般に、*codeset* は省略できます。通常、ロケールは、オペレーティング・システムのインストールの一環としてインストールされます。

国際化対応のコネクターにおける設計上の考慮事項

このセクションでは、コネクターを国際化対応にする際の設計上の考慮事項について説明します。内容は次のとおりです。

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

ロケール依存設計原則

コネクターを国際化対応にするには、コネクターをロケール依存となるようにコーディングする必要があります。つまり、コネクターは、ロケールの設定に従って動作し、そのロケールに対応したタスクを実行する必要があります。例えば、英語を

使用するロケールの場合、コネクターは、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。作成するコネクターの国際化対応 (I18N) を完了するには、アプリケーション固有のコンポーネントを国際化対応にする必要があります。

表 15 に、国際化対応されたアプリケーション固有のコンポーネントが従う必要のある、ロケール依存設計の原則がリストされています。

表 15. アプリケーション固有のコンポーネントのためのロケール依存設計の原則

設計の原則	詳細情報の参照先
エラー・メッセージ、状況メッセージ、およびトレース・メッセージの各テキストを、アプリケーション固有のコンポーネントから分離してメッセージ・ファイルに格納し、該当するロケールの言語に変換すること。	『テキスト・ストリング』
コネクターの実行中は、ビジネス・オブジェクトのロケールを保持したままにしておくこと。	67 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を、マルチバイト文字の入力が可能なように設定すること。	68 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作業を検討すること。	68 ページの『その他のロケール依存のタスク』

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、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、より良いプログラミング方法であるといえます。テキスト・メッセージを生成する必要がある場合、コネクターは、メッセージ・ファイルのメッセージ番号によって適切なメッセージを検索します。すべてのメッセージを 1 つのメッセージ・ファイルに収集すると、テキストを適切な言語に変換することによって、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・ストリングの国際化対応方法について、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』
- 66 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、これらの操作すべてで、必ずメッセージ・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てください。メッセージ・ストリングをメッセージ・ファイル内に格納することにより、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 では、メッセージ・ファイルを使用する操作のタイプおよび関連する `GenGlobals` クラスの C++ コネクター・ライブラリーメソッドをリストしています。アプリケーション固有のコンポーネントは、これらのメソッドを使用してメッセージ・ファイルからメッセージを検索します。

表 16. メッセージ・ファイルからロギングおよびトレースを行うメソッド

メッセージ・ファイル操作	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、使用するユーザーのロケールに合った言語で表示する必要があります。したがって、ログ・メッセージは、コネクター・メッセージ・ファイルに分離し、generateAndLogMsg() メソッドで検索可能なようにしてください。

トレース・メッセージは、製品のデバッグ処理で使用されるため、ユーザーのロケールに合った言語で表示する必要はない場合もあります。したがって、トレース・メッセージをメッセージ・ファイル内に含めるかどうかは、開発者の判断に任せられています。

- 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・メッセージが検索され、テキストがフォーマットされた後、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。したがって、トレース・メッセージ (英語) を直接 traceWrite() への呼び出しに含めることができます。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、メッセージの検索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・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もメッセージ・ファイルに分離してください。表 17 に、アプリケーション固有のコンポーネントでメッセージ・ファイルからメッセージを検索するために使用可能な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検索するメソッド

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

ハードコーディング・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・ストリングに対して、コネクター・メッセージ・ファイル内で一意の番号を付けたメッセージを生成します。

注: メッセージ・ファイル内には、分離したストリングに関する説明もオプションで含めることができます。この説明に、ストリングが使用されるメソッドの名前を含めることができます。この情報は、ソースの位置を追跡したり、必要な変更を加える際に役立ちます。

- アプリケーション固有のコンポーネントでは、generateMsg() メソッドを使用して、分離し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に、以下のようなハードコーディング・ストリングが入っているコード行が含まれているとします。

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

このハードコーディング・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・ファイルにメッセージを作成し、こ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、以下のようにメッセージ・ファイルから分離したストリング (メッセージ 100) を検索し、ハードコーディング・ストリングをこの検索したストリングで置き換えます。

```
char * msg;
//retrieve the message numbered '100'
msg = generateMsg(100, CxMsgFormat::XRD_INFO, NULL, 0, NULL);
MyClassObject::formatMsg(msg); // 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、153 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: 一般に、アプリケーション・データ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する場合、コネクターは、ロケール依存処理 (データ形式の変換など) を実行する必要があります。コネクターでのビジネス・オブジェクトの処理中に、2 つの異なるロケール設定値があります。

- コネクターは、実行するコネクター・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・ロケールによって、コネクターがロギングと例外について使用するテキスト・メッセージのロケールが決定されます。
- コネクターは、処理中のビジネス・オブジェクトに関連するロケールにアクセスすることもできます。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関連付けられているロケールを識別します。

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の検索のために使用するメソッドを、表 18 に示します。

表 18. コネクター・フレームワークのロケールを検索するメソッド

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

ビジネス・オブジェクトを作成する際は、ロケールをそのデータに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できます。

- ビジネス・オブジェクト・ロケールの名前を取得するには、`getLocale()` メソッドを使用します。このメソッドは、`BusinessObject` クラス内で定義されています。
- ロケールをビジネス・オブジェクトに関連付けるには、 `BusinessObject()` コンストラクターを使用します。これは、 `BusinessObject` クラスでも定義されています。

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。

- すべてのコネクターに使用可能な標準構成プロパティー。
- 定義された特定のコネクターに固有であるコネクター固有の構成プロパティー。

コネクター構成プロパティーの名前にはすべて、米国英語 (en_US) ロケールに関連付けられたコード・セットに定義された文字のみ を使用してください。ただし、これらの構成プロパティーの値には、コネクター・フレームワークのロケールに関連付けられたコード・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検索』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、確実にコネクターを国際化対応にするには、構成プロパティーの値を検索し、これらの値をコネクターのコードによって正しく処理する必要があります。アプリケーション固有のコンポーネントでは、構成プロパティーの値が単一バイト文字のみで構成されているとは*限りません*。

その他のロケール依存のタスク: 国際化対応するコネクターには、以下のようなロケール依存のタスクも行う必要があります。

- データのソートまたは照合: これらの作業で、該当するロケールの言語および国に対応した照合順序を使用すること。
- 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の区別): これらの作業で行う処理を、必ず該当するロケールの言語の文字に対応させること。
- 日付、数値、および時刻の形式: これらの形式を、必ず該当するロケールに対応させること。

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は、データの意味を保持するために、特定の文字変換を行う必要があります。Java 仮想マシン (JVM) 内の Java ランタイム環境では、ユニコード形式でデータが表現されます。ユニコード文字セットは汎用文字セットであり、よく知られている文字コード・セット内の文字 (単一バイトおよびマルチバイト) のエンコードが含まれています。ユニコードのエンコード形式には、いくつかのタイプがあります。統合ビジネス・システムでは、次のエンコードが多く使用されます。

- 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ユニコード文字セットです。

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

  UTF-8 エンコードは、UNIX 環境でユニコード文字データを処理できるように設計されています。 UTF-8 エンコードは、ASCII コードの値 (0...127) をすべてサポートしているため、ASCII コード以外に解釈されることはありません。通常、各コードの値は、1 バイト、2 バイト、または 3 バイトの値として表現されます。

コネクター・フレームワークを含め、WebSphere Business Integration システム の大半のコンポーネントは、Java で作成されています。したがって、そうしたシステム・コンポーネント間でデータを転送する際は、ユニコードのコード・セットでデータがエンコードされるため、文字変換を行う必要はありません。

ただし、C++ コネクターは、C++ アプリケーション (またはテクノロジー) とともに動作します。C++ アプリケーション (またはテクノロジー) は、ユニコードのコード・セット内にすでに存在するデータを持っていない場合があります。したがって、C++ で作成され、コネクター・フレームワークにより Java に変換された、コネクターのアプリケーション固有のコンポーネントでは、アプリケーション固有のビジネス・オブジェクトのアプリケーション・データに対して、文字変換を行わなければならない場合があります。図 23 は、C ++ コネクターの文字エンコードを示しています。

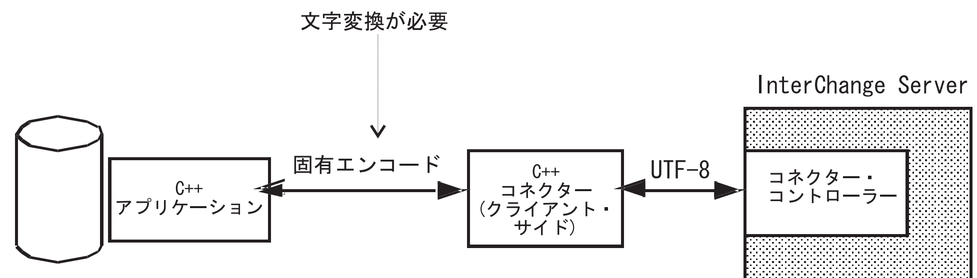

図 23. C++ コネクターでの文字エンコード

注: コネクターは、`CharacterEncoding` コネクター構成プロパティーから、そのアプリケーションの文字エンコードを取得します。コネクターで文字変換を行う場合は、コネクターのエンド・ユーザーに対して、このコネクター・プロパティーに適切な値を設定するように指示する必要があります。

表 19 は、コネクターで使用可能な C++ コネクター・ライブラリー内のメソッドを示しています。このメソッドを使用すると、実行時に文字エンコードを取得できます。

表 19. コネクター・フレームワークの文字エンコードを検索するメソッド

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

注: String 値を持つコネクター構成プロパティーでは、文字変換を行う必要はありません。なぜなら、この値は、InterChange Server リポジトリーからのものであり、UCS-2 エンコードであるからです。

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 をインプリメント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。

この章は、以下のセクションから構成されています。

- 『コネクターの実行』
- 77 ページの『コネクターの基底クラスの拡張』
- 78 ページの『エラー処理』
- 79 ページの『コネクター構成プロパティー値の使用』
- 81 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』

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インプリメントしている場合は、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()` メソッドによってインプリメントされます。詳細については、 83 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。

© Copyright IBM Corp. 1997, 2004

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、235 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、実行すると、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各ビジネス・オブジェクトに対して、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、コラボレーションがサブスクライブしているビジネス・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
GenGlobals	init

コネクター・クラスのインプリメントの一環として、コネクターの初期化メソッドをインプリメントする必要 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・バージョンの確認』
- 74 ページの『進行中イベントのリカバリー』

重要: 初期化メソッド実行の間は、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、まだ使用可能になっていません。

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、コネクター情報を使用可能にするコネクター構成プロパティー (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、障害状況を戻し、障害の原因を常時示します。

  C++ コネクターで、init() で使用される一般的な戻りコードは、BON_SUCCESS、BON_FAIL、および BON_UNABLETOLOGIN です。上記を含め、その他の戻りコードの詳細については、225 ページの『C++ 戻りコード』を参照してください。

コネクター・バージョンの確認: 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検索の実行、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. 進行中イベントのリカバリー対応措置

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 に True がセットされている場合は、エラーに関する E メール通知を発生させます。	FailOnStartup

表 23. 進行中イベントのリカバリー対応措置 (続き)

イベント・リカバリー・アクション	InDoubtEvents の値
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

C++ 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、適切なリカバリー措置を講じるため、アプリケーション固有インターフェースを使用する必要があります。

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、123 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操作を実行します。各コネクターは、ビジネス・オブジェクト・ハンドラーを検索するため、そのコネクター基底クラス内に getBOHandlerforBO() 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インプリメントの一環として、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、getBOHandlerforBO() をインプリメントする必要 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、getBOHandlerforBO() メソッドを 1 回呼び出す。
3. getBOHandlerforBO() 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、getBOHandlerforBO() メソッドは、コネクターでサポートされるビジネス・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ビジネス・オブジェクトの一部または全部に、特殊処理が必要な場合は、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* を設定する必要があります。

  ご使用のコネクターで、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・ハンドラーが必要な場合は、getBOHandlerforBO() 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、getBOHandlerforBO() 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・ハンドラー検索メソッドが、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: getBOHandlerforBO() メソッドをインプリメントする前に、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。

getBOHandlerforBO() メソッドのインプリメント方法については、170 ページの『C++ 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、83 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インプリメント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) 宛先への指定経路による送信のため、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・メカニズム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インプリメントし、イベント・ストアからイベント通知を周期的に検索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、ポーリング・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。

C++ コネクター・ライブラリーでは、pollForEvents() メソッドは GenGlobals クラスに定義されています。 pollForEvents() で使用される一般的な戻りコードは、BON_SUCCESS、BON_FAIL、および BON_APPRESPONSETIMEOUT です。戻りコードの詳細については、 225 ページの『C++ 戻りコード』を参照してください。

重要: 開発者は、pollForEvents() メソッドのインプリメンテーション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、インプリメント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、そのスタブをインプリメントする必要があります。

pollForEvents() のインプリメントとイベント通知の詳細については、 123 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と、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。

コネクターの基底クラスの拡張

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ドが含まれています。

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インプリメントする必要があります。

表 24. コネクター基底クラスにインプリメントするメソッド

説明	コネクター基底クラス・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。	init()	73 ページの『コネクターの初期化』

表 24. コネクター基底クラスにインプリメントするメソッド (続き)

説明	コネクター基底クラス・メソッド	詳細情報
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・バージョンの確認』
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。	`getBOHandlerforBO()`	75 ページの『ビジネス・オブジェクト・ハンドラーの取得』
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 24 に、コネクター・フレームワークが呼び出す一通り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、 83 ページの『第 4 章 要求処理』を参照してください。

コネクター・フレームワーク
アプリケーション固有のコネクター・コンポーネント
始動
init()
getVersion()
getBOHandlerForBO()
ランタイム
pollForEvents()
doVerbFor()
terminate()

図 24. 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、 167 ページの『C++ コネクター基底クラスの拡張』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。仮想メソッドは、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、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。

  C++ 戻りコードについては、 225 ページの『C++ 戻りコード』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通知することができます。詳細については、 226 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、ロギング先に通知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、 153 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の概要』
- 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検索』

コネクター構成プロパティーの概要

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は、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となります。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。

コネクター固有の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・リポジトリーに格納されてい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。一般に、コネクターのコードで必要なことは、`ApplicationUserID` や `ApplicationPassword` のようなコネクター固有のプロパティー値を照会することのみです。

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。
- コネクター固有の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 ツールの詳細について「*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検索

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検索します。

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、単一値のシンプル・コネクター構成プロパティーは、C++ コネクターがサポートする唯一のコネクター・プロパティー・タイプです。C++ コネクターでは、階層プロパティーはサポートしていません。

単一値のシンプルなコネクター構成プロパティーを検索するには、`getConfigProp()` メソッドを使用します。 C++ コネクター・ライブラリーでは、`getConfigProp()` メソッドは、入力データとして、プロパティー値の名前に対する文字ストリングを取ります。ストリングは、大文字小文字を区別します。また、メソッドがプロパティー値を書き込むことができるバッファーに対するポインターとバッファー内に占めるバイト数も入力データとして取ります。`getConfigProp()` メソッドは、`GenGlobals` クラス内で定義されています。下のコーディングの抜粋には、`Hostname` 構成プロパティーの値を検索する際の `getConfigProp()` を使用方法を示します。

```
char * hostnameBuffer = new char[512];
if (getConfigProp("Hostname", hostnameBuffer, 511) == 0)
     logMsg("Invalid or empty property name.");
else
     hostName = hostnameBuffer;
```

アプリケーションとの接続が切断された場合の処理

優れた設計習慣として、アプリケーションとの接続が失われた場合に、必ずシャットダウンするように、コネクター・アプリケーション固有コードをコーディングします。切断の接続に対して応答する場合、コネクターのアプリケーション固有コンポーネントは、次の手順を実行する必要があります。

- `LogAtInterchangeEnd` コネクター構成プロパティーに True がセットされている場合に E メール通知を発生させるように、致命的エラー・メッセージをログに記録します。
- `BON_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。これが発生した場合は、コネクターが実行しているプロセスが停止されます。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・インプリメントの仮想メソッドは、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、 206 ページの『イベント・ストアにアクセスする前の接続の検証』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、175 ページの『動詞を処理する前の接続の検証』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 つの機構をインプリメントします。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。要求処理をインプリメントする機構は *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。この章は、以下のセクションから構成されています。

- 『ビジネス・オブジェクト・ハンドラーの設計』
- 86 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』
- 87 ページの『要求の処理』
- 91 ページの『Create 動詞の処理』
- 95 ページの『Retrieve 動詞の処理』
- 101 ページの『RetrieveByContent 動詞の処理』
- 103 ページの『Update 動詞の処理』
- 111 ページの『Delete 動詞の処理』
- 113 ページの『Exists 動詞の処理』
- 114 ページの『ビジネス・オブジェクトの処理』
- 121 ページの『コネクター応答の指示』
- 122 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、28 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インプリメントします。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、`BOHandlerCPP` 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・ハンドラーをインプリメントする方法は、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。

© Copyright IBM Corp. 1997, 2004

- 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。
- 異なる種類のオブジェクト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、 86 ページの『複数のビジネス・オブジェクト・ハンドラーのインプリメント』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。
- 『ビジネス・オブジェクトのメタデータ』
- 85 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 25 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スキームをいくつか示します。

表 25. 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・テーブル、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持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 getBOHandlerforBO() メソッドをインプリメントするだけで、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・ハンドラーをインプリメント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・ハンドラーのインプリメント

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に個別のハンドラー・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、getBOHandlerforBO() メソッドをインプリメントして、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・ハンドラーをインプリメントする場合は、ビジネス・オブジェクト・ハンドラー・クラスごとに doVerbFor() メソッドをインプリメント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るべきです。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が避けられません。

ビジネス・オブジェクト・ハンドラー基底クラスの拡張

C++ 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 BOHandlerCPP 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、その仮想メソッド doVerbFor() をインプリメントする必要があります。C++ コネクター・ライブラリー固有の情報については、172 ページの『C++ 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。

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() をインプリメント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 getBOHandlerforBO() 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インプリメント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() をインプリメントする必要があります。このメソッドは Retrieve 動詞をインプリメントします。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 場合にもインプリメントする必要があります。

このセクションには、doVerbFor() メソッドをインプリメントする方法について、以下の情報が記載されています。

- 『doVerbFor() の基本ロジック』
- 89 ページの『動詞インプリメントに関する一般推奨事項』

doVerbFor() の基本ロジック

C++ コネクターの場合、BOHandlerCPP クラスで doVerbFor() メソッドが定義されています。これは定義済みの仮想メソッドです。 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 25 は、このメソッドの基本ロジックのフローチャートです。

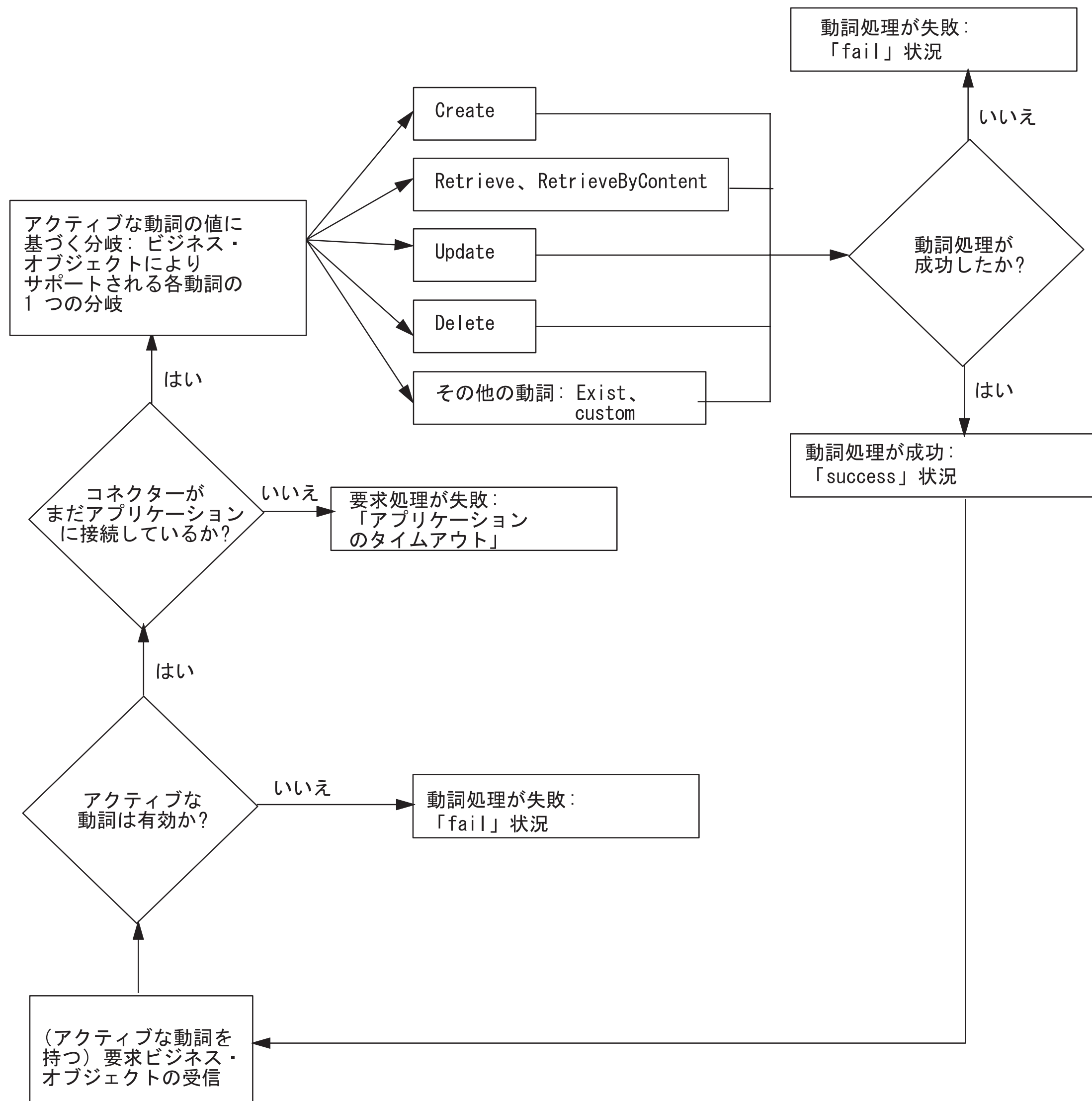

図 25. doVerbFor() の基本ロジックのフローチャート

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、 173 ページの『doVerbFor() メソッドのインプリメント』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、この doVerbFor() メソッドに要求ビジネス・オブジェクトを渡します。表 26 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 26. 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。	90 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。	114 ページの『ビジネス・オブジェクトの処理』

動詞インプリメントに関する一般推奨事項

このセクションには、doVerbFor() メソッドをインプリメントする方法について、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る必要があります。コネクターが要求を受信したとき、InterChange Server に戻される階層ビジネス・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期間中に障害が検出された場合には、トランザクション全体をロールバックする必要があります。

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、126 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と 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 27 の『詳細情報』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、インプリメントの注意事項、および適切な結果状況値に関する情報があります。

表 27 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インプリメントする必要があります。

表 27. doVerbFor() メソッドでインプリメントする動詞

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。	91 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。	95 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。	101 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。	103 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。	111 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。	113 ページの『Exists 動詞の処理』
Custom verbs	アプリケーション固有の操作を実行します。	なし

注: 表 27 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インプリメント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インプリメント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インプリメント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検索する必要があります。C++ コネクターの場合は、getVerb() は BusinessObject 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インプリメントする。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インプリメント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タイプ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確実に値が割り当てられ

るようにするには、initAndValidateAttributes() メソッドを呼び出します。このメソッドは、属性のデフォルト値を、値が設定されていない必須属性のそれぞれに割り当てます (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、BusinessObject クラスで定義されます。initAndValidateAttributes() を呼び出した後に、アプリケーションで Create 操作を実行してください。

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』
- 93 ページの『Create 動詞操作のインプリメント』
- 94 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 C++ 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 26 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。

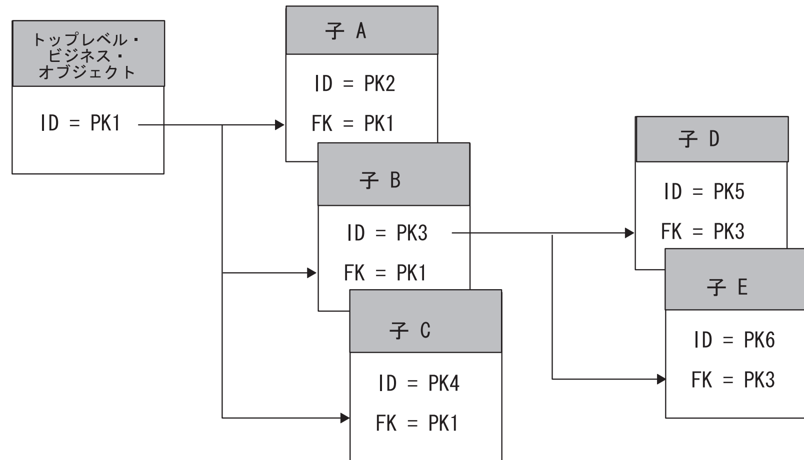

図 26. 親/子関係の作成

Create 動詞操作のインプリメント

Create 操作の典型的なインプリメントでは、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検索します。
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、タイプ `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、main 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。

main メソッドが基本キー値を提供する方法は一とおりではありません。例えば、main 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検索して、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、main 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、main 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: Create 動詞メソッドの例については、 196 ページの『例: フラット・ビジネス・オブジェクトの Create メソッド』を参照してください。

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インプリメントする際は、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 28に示す結果状況値の 1 つを戻さなければなりません。

表 28. C++ Create 動詞処理で可能な結果状況

Create 条件	C++ 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。	BON_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。	BON_SUCCESS
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。	
• Create 操作を失敗させる。	BON_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。	BON_VALDUPES
Create 操作が失敗すると、動詞メソッド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。	BON_FAIL

注: コネクター・フレームワークは、BON_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タイプを指定された既存のアプリケーション・エンティティーが検索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検索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。
- 階層ビジネス・オブジェクトの場合、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) の作成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・データベースから検索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検索します。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検索します。このタイプの検索操作を*変更後イメージ検索* 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()` メソッドをインプリメント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。

データを検索するもう 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。このタイプの検索処理は、RetrieveByContent 動詞メソッドで実行されます。非キー値による検索については、101 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。

- 96 ページの『Retrieve 動詞の標準処理』
- 96 ページの『Retrieve 動詞操作のインプリメント』
- 97 ページの『例: Retrieve 操作』

- 98 ページの『子オブジェクトの検索』
- 101 ページの『Retrieve 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 C++ 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タイプの新しいビジネス・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・オブジェクトの検索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・データを検索し、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・データを検索し、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・データを検索できないと、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。100 ページの『欠落している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 動詞操作のインプリメント

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検索できます。
- 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検索できます。

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。このタイプのインプリメントでは、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、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、このインプリメントでは、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ビジネス・オブジェクト

図 27 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。

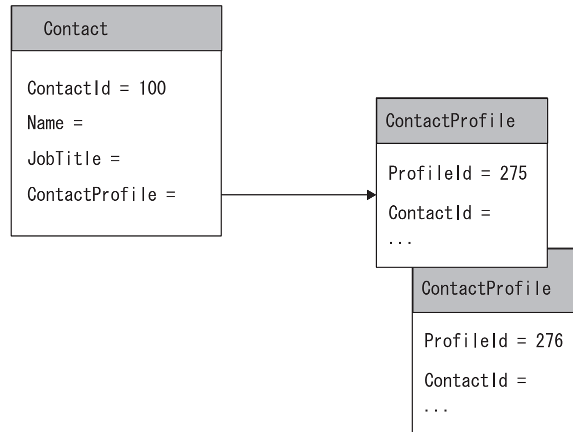

図 27. Retrieve 要求用のビジネス・オブジェクトの内容の例

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 28 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 26 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。

contact 表

contact_id	name	job_title
100	Jones	VP
200	Smith	Manager

contact_profile 表

profile_id	contact_id	job_code	department
275	100	42	422
276	100	53	422
277	100	78	422
278	200	156	537

図 28. 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) を検索します。正しい数の子ビジネス・オブジェクトが検索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 28 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。

子オブジェクトの検索

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検索するために、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 では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)、関連データを検索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次のように指定します。

- 親テーブル: contact
- 子テーブルの外部キー列: 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 29 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。

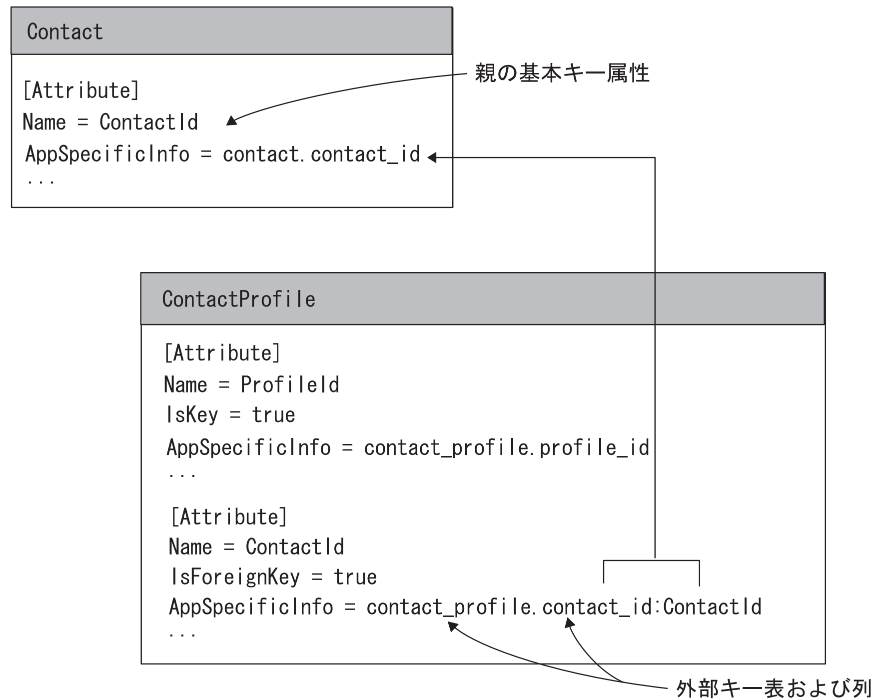

図 29. ビジネス・オブジェクトの外部キー関係

動詞操作では、アプリケーション固有の情報を使用して、子テーブルの名前 (`contact_profile`) と、子テーブルの中の外部キー列 (`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テーブル内のすべてのレコードを検索する 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検索する 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 30) から 3 行を戻します。

contact 表

contact_id	name	job_title
100	Jones	VP
200	Smith	Manager

contact_profile 表

profile_id	contact_id	job_code	department
275	100	42	422
276	100	53	422
277	100	78	422
278	200	156	537

図 30. 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。動詞操作は検索された行を次のように処理します。

1. 行ごとに、正しいタイプ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検索する。
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 31 のようになります。動詞操作は現行データベース・エンティティーを検索し、階層ビジネス・オブジェクトに子を追加しています。

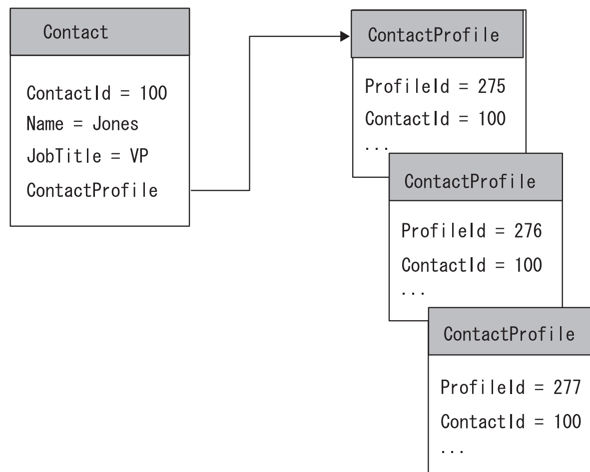

図 31. Retrieve 要求のサンプルに対するビジネス・オブジェクトの応答

欠落している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・データを検索できない場合に、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、動詞操作をインプリメント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・オブジェクトの検索に成功すれば、その子の検索が成功したかどうかにかかわりなく、動詞操作は BON_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 29 に示す結果状況の 1 つを戻さなければなりません。

表 29. C++ Retrieve 動詞処理で可能な結果状況

Retrieve 条件	C++ 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。	BON_VALCHANGE
IgnoreMissingChildObject コネクター・プロパティーが True の場合、トップレベル・オブジェクトの検索に成功すれば、その子の検索に成功したかどうかにかかわりなく、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。	BON_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。	BON_BO_DOES_NOT_EXIST
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検索を行う。	BON_FAIL

注: コネクター・フレームワークは、BON_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がないビジネス・オブジェクトの検索が必要な場合があります。そのような検索を「非キー値による検索」または「内容による検索」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・エンティティーの検索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タイプを指定された既存のアプリケーション・エンティティーが検索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検索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) が検索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 動詞操作のインプリメント』
- 『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 C++ 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 動詞操作のインプリメント

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・データを検索することを除けば、Retrieve 動詞と同様に機能します。この動詞の最も堅固なインプリメントでは、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に 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ビジネス・オブジェクトの検索を行えるわけではありません。

もっと一般的なインプリメントでは、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検索をサポートし、この内容による検索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検索できます。次に、動詞操作は Retrieve 操作を実行して、完全なビジネス・オブジェクトを検索します。

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インプリメントして、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 30 に示す結果状況値の 1 つを戻さなければなりません。

表 30. C++ RetrieveByContent 動詞処理で可能な結果状況

RetrieveByContent 条件	C++ 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。	BON_VALCHANGE
IgnoreMissingChildObject コネクター・プロパティーが True の場合、トップレベル・オブジェクトの検索に成功すれば、その子の検索に成功したかどうかにかかわりなく、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。	BON_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だけを検索する。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。 • 戻り状況記述子に検索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ほか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。	BON_MULTIPLE_HITS
RetrieveByContent 操作が非キー値による検索で一致を見つけられなかった 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。	BON_FAIL_RETRIEVE_BY_CONTENT

注: コネクター・フレームワークは、BON_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タイプを指定された既存のアプリケーション・エンティティーが検索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』
- 108 ページの『論理 Delete イベントを表すビジネス・オブジェクトの含意』
- 110 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 C++ 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タイプの新しいビジネス・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・オブジェクトの検索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検索します。

   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検索します。

   - 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・エンティティーを検索します。
   - 階層ビジネス・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. 検索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、削除操作 (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、108 ページの『論理 Delete イベントを表すビジネス・オブジェクトの含意』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検索します。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 BON_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 32 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。

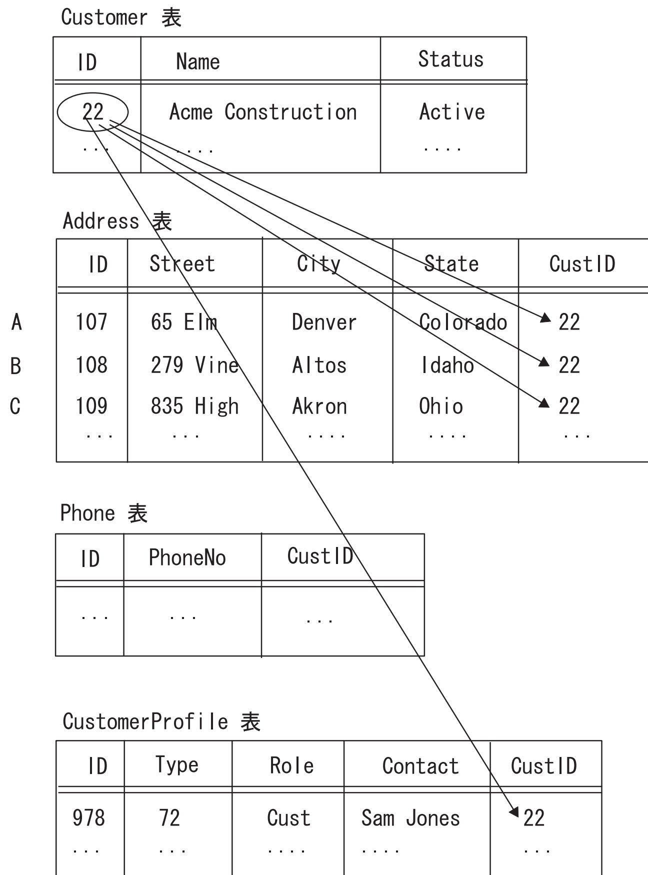

図 32. Update 要求前の顧客エンティティー

統合ブローカーが 図 33 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。

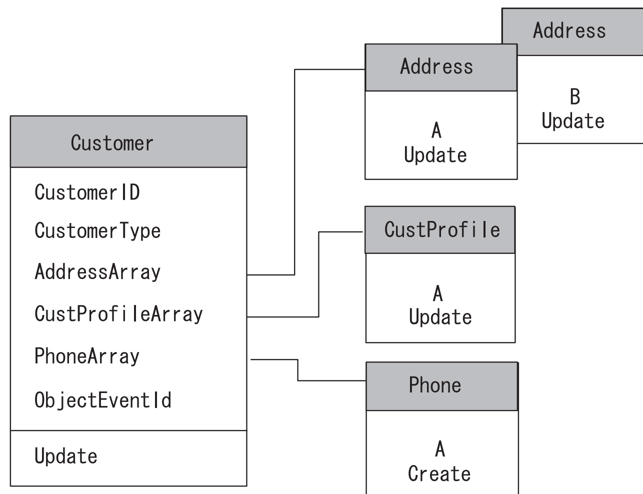

図 33. Update のための Customer 要求ビジネス・オブジェクト

この要求ビジネス・オブジェクトは、表 31 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 31. 要求ビジネス・オブジェクトの Acme Construction に対する更新

Acme Construction に行われた更新	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 34 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。

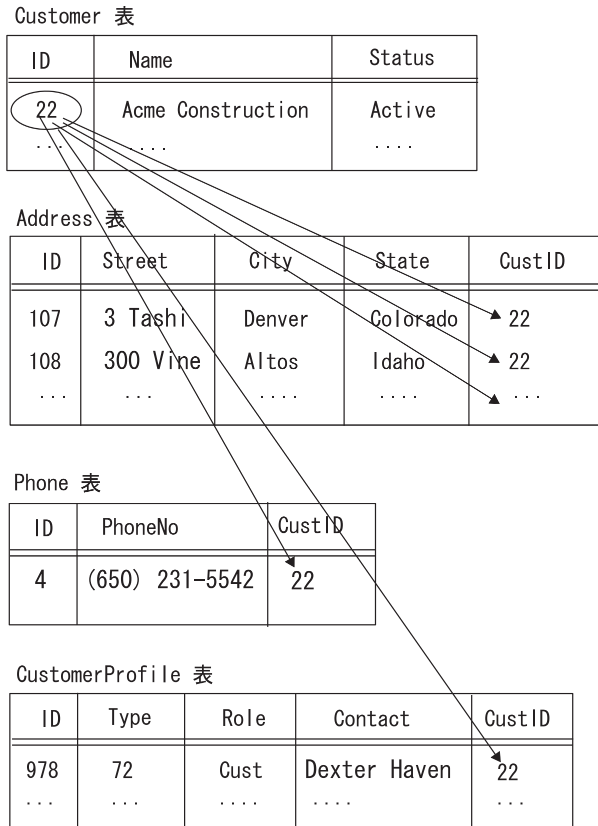

図 34. Update 要求後の Customer エンティティー

論理 Delete イベントを表すビジネス・オブジェクトの含意

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 35 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。

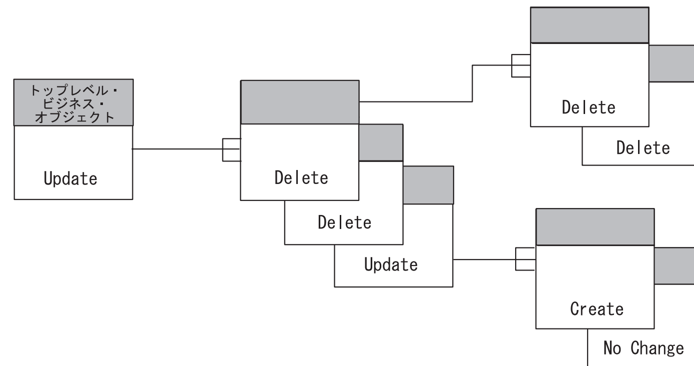

図 35. ソース・アプリケーション内の更新済みエンティティー

ソース・アプリケーション・コネクターで、 123 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 36 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もはやビジネス・オブジェクト階層内に存在しません。

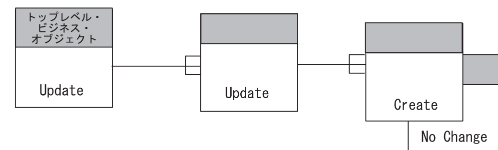

図 36. 物理削除コネクターの要求ビジネス・オブジェクトの更新

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 37 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。

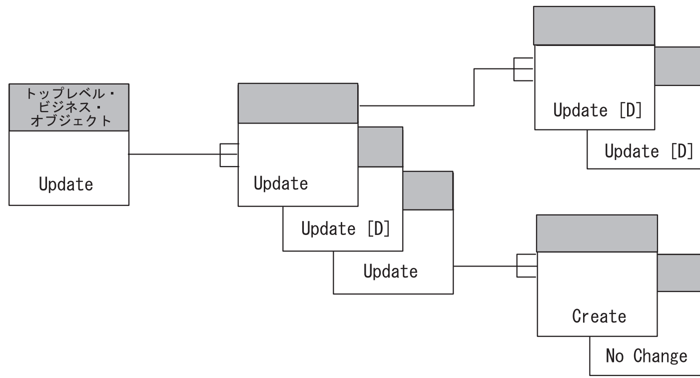

図 37. 論理削除コネクターの要求ビジネス・オブジェクトの更新

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インプリメントする。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 操作をインプリメントする。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で、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、削除操作を呼び出して、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 32 に示す結果状況の 1 つを戻さなければなりません。

表 32. C++ Update 動詞処理で可能な結果状況

Update 条件	C++ 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。	BON_SUCCESS
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。	BON_VALCHANGE

表 32. C++ Update 動詞処理で可能な結果状況 (続き)

Update 条件	C++ 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。	BON_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。	BON_FAIL

注: コネクター・フレームワークは、BON_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 33 に示すインプリメントのいずれかをサポートします。

表 33. Delete のインプリメント

Delete の インプリメント	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 タイプ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削除操作は、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、108 ページの『論理 Delete イベントを表すビジネス・オブジェクトの含意』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タイプ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・データベースから検索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 C++ 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検索を実行して、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トップレベル・エンティティーへ、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: 削除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 34 に示す結果状況の 1 つを戻さなければなりません。

表 34. C++ Delete 動詞処理で可能な結果状況

Delete 条件	C++ 結果状況
InterChange Server のみ: 通常、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。	BON_VALCHANGE
すべての統合ブローカー: 削除操作が失敗した場合は、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。	BON_FAIL

注: コネクター・フレームワークは、BON_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タイプ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 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』
- 114 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 C++ 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 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. C++ Exists 動詞処理で可能な結果状況

Exists 条件	C++ 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。	BON_SUCCESS
Exists 操作がトップレベル・オブジェクトの検索に失敗すると、次の処理が行われます。 • 「exist」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。	BON_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』
- 117 ページの『階層ビジネス・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・オブジェクトを表す』
- 116 ページの『単純属性へのアクセス』

フラット・ビジネス・オブジェクトを表す

ほかのビジネス・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 38 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。

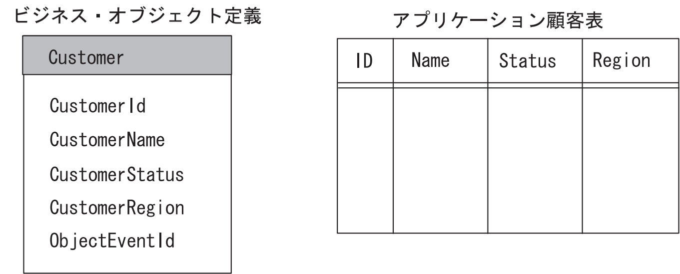

図 38. フラット・ビジネス・オブジェクトおよび対応するアプリケーション表

図 38 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 39 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、および値だけが入っています。

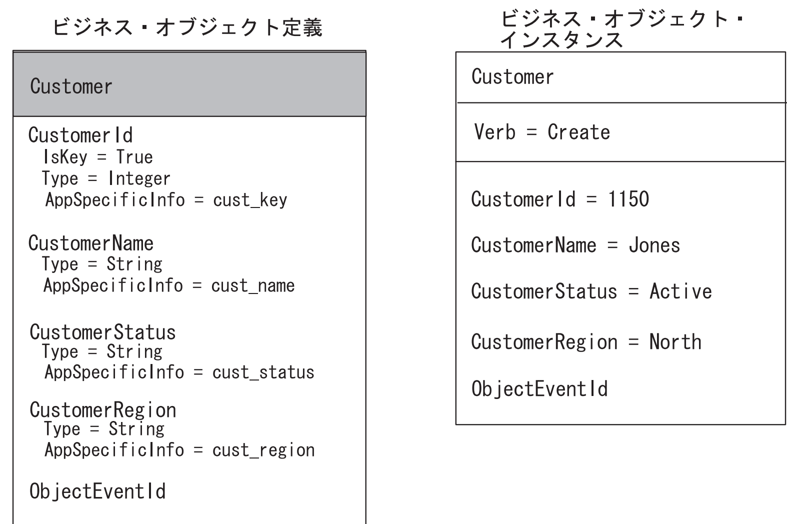

図 39. アプリケーション固有情報を持つフラット・ビジネス・オブジェクト

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 40 は、図 39 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。

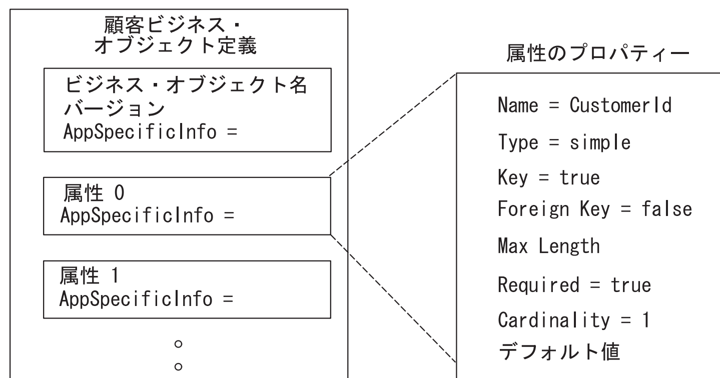

図 40. ビジネス・オブジェクト属性プロパティー

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 (順序位置) があります。例えば、図 40 に示すように、`CustomerId` 属性は順序位置 `0` でアクセスされ、`CustomerName` 属性は順序位置 `1` でアクセスされ、以下同様です。C++ コネクター・ライブラリーは、名前と順序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 39 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 36 に示すように設計されています。

表 36. 表ベースのアプリケーションのアプリケーション固有情報

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 36. 表ベースのアプリケーションのアプリケーション固有情報 (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、以下同様です。階層ビジネス・オブジェクトは、*トップレベル・ビジネス・オブジェクト* (階層のトップレベルのビジネス・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・オブジェクトを表す』
- 119 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・オブジェクトを表す

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 n の包含 — 属性が*ビジネス・オブジェクト配列* という構造内に複数の子ビジネス・オブジェクトを含んでいます。

図 41 は、典型的な階層ビジネス・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。

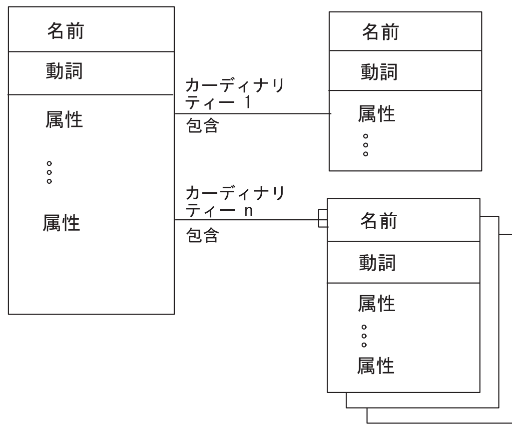

図 41. 階層ビジネス・オブジェクト

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 42 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。

  配列内の子ビジネス・オブジェクトのそれぞれには、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 43 のように表現できます。

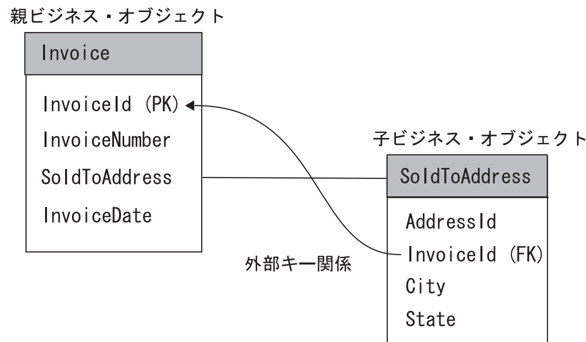

図 42. 単一カーディナリティーを持つビジネス・オブジェクト

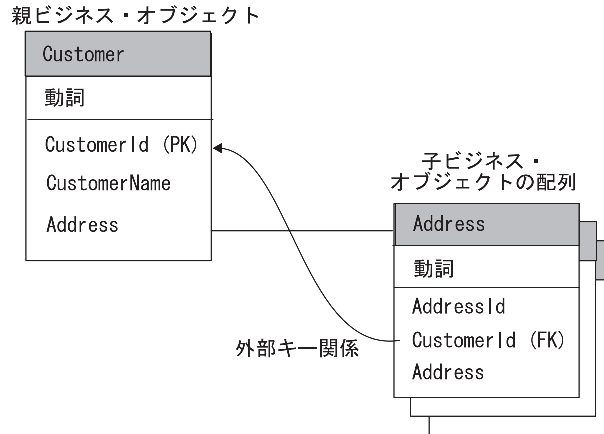

図 43. 複数カーディナリティーを持つビジネス・オブジェクト

注: 図 42 および図 43 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、属性タイプが `OBJECT` であるかどうかを判別します。

   `OBJECT` タイプは、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` 属性タイプ定数は BOAttrType クラスで定義されます。`isObjectType()` メソッドは、属性が複合である場合、すなわちビジネス・オブジェクトを含んでいる場合に、True を戻します。

2. `doVerbFor()` メソッドは、属性にビジネス・オブジェクトが含まれていることを検出すると、`isMultipleCard()` を使用して、その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・オブジェクトで操作を実行する 1 つの方法は、`doVerbFor()` または verb メソッドを子オブジェクトで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、メソッドはプリンター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。

属性に複数のカーディナリティー (カーディナリティー n) がある場合、属性は子ビジネス・オブジェクトの配列を含んでいます。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 つの要素でビジネス・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てはなりません。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。

main 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 44 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。

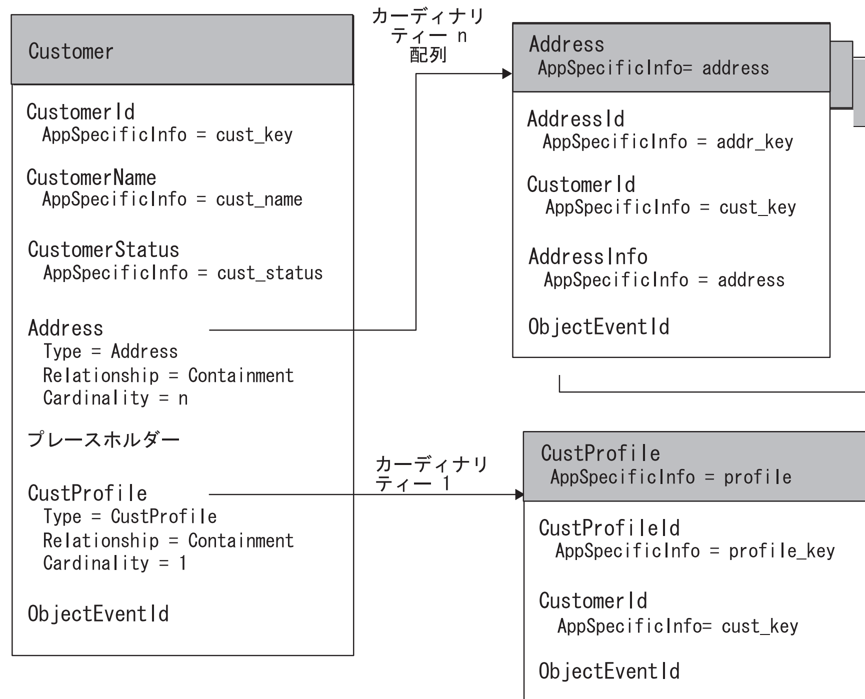

図 44. 階層ビジネス・オブジェクト定義の例

サブメソッドは、OBJECT タイプ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検索し、再帰的に doCreate() を呼び出して、Address 属性を処理します。main 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、 202 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 37 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 37. doVerbFor() メソッドからの応答情報

応答情報	戻される応答
結果状況	doVerbFor() の整数戻りコード。

表 37. *doVerbFor()* メソッドからの応答情報 (続き)

応答情報	戻される応答
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応答ビジネス・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応答ビジネス・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応答ビジネス・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。

C++ 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() メソッドは致命的エラー・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を発生させます。また、メソッドは、BON_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。これが発生した場合は、コネクターが実行しているプロセスが停止されます。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、 175 ページの『動詞を処理する前の接続の検証』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インプリメントされた機構です。この章では、イベント通知機構のインプリメント方法について説明します。内容は次のとおりです。

- 『イベント通知機構の概要』
- 124 ページの『アプリケーション用イベント・ストアのインプリメント』
- 130 ページの『イベント検出のインプリメント』
- 136 ページの『イベント検索のインプリメント』
- 138 ページの『ポーリング・メソッドのインプリメント』
- 143 ページの『イベント処理における特別な考慮事項』

注: イベント通知の概要については、 25 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部のエンティティーが変更された時点を確認できるようになります。イベント通知機構のインプリメントは、表 38 に示すように、3 段階のプロセスです。

表 38. イベント通知機構の段階

イベント通知機構の段階	詳細情報の参照先
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。	124 ページの『アプリケーション用イベント・ストアのインプリメント』
アプリケーションの内部に**イベント検出**メカニズムを実装します。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。	130 ページの『イベント検出のインプリメント』
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**イベント検出**メカニズム (ポーリング・メカニズムなど) 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。	124 ページの『アプリケーション用イベント・ストアのインプリメント』

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、 25 ページの『イベント通知』を参照してください。

© Copyright IBM Corp. 1997, 2004

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・アカウントのセットアップ、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・ストアのインプリメント

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あり、コネクターが処理するまで、ここにイベント・レコードを保管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、E 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』
- 127 ページの『イベント・ストアの可能なインプリメント形態』

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・データを検索し、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: イベント検出メカニズム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メカニズムを提供する場合、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ることが可能です。

表 39 に、イベント・レコードの標準的内容を示します。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 39. イベント・レコードの標準的要素

要素	説明	詳細情報の参照先
イベント ID	イベントの固有 ID (UID) です。	126 ページの『イベント ID』
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。	125 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。	125 ページの『Event 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。	125 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。	142 ページの『イベントの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。	126 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。	142 ページの『イベントの分散』

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している場合に、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。

Event 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を種類を表します。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、143 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 - 値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複数の名前 - 値ペアをサポートしています。区切り文字は構成可能で、PollAttributeDelimiter コネクター構成プロパティーで設定されます。デフォルトの区切り文字はコロン (:) 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 採番方式の例として、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。階層的なビジネス・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

IBM では、表 40 に示したイベント状況値を C++ コネクターが使用することをお勧めしています。作成した C++ コードを読みやすくするため、これらのイベント状況値のそれぞれにつき定数を定義することもできます。

表 40. C++ コネクター用として推奨されるイベント状況値

イベント状況値	説明
0	ポーリング可能
1	統合ブローカーへの送信
2	イベントについてのサブスクリプションなし
3	イベントが進行中

表 40. C++ コネクター用として推奨されるイベント状況値 (続き)

イベント状況値	説明
-1	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。
-2	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。

イベント・ストアの可能なインプリメント形態

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』
- 128 ページの『イベント表』
- 129 ページの『E メール』
- 130 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させ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。
- イベント検索 — 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検索できます。受信箱にアクセスす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 45 にこのような対話を示します。

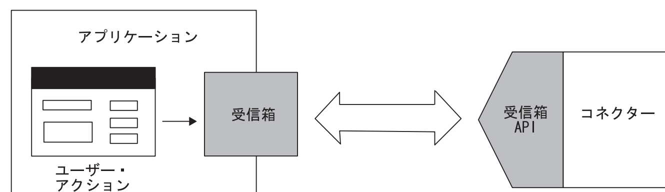

図 45. イベント・ストアとしてのイベント受信箱

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。
- イベント検索 — 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 46 にこのような対話を示します。

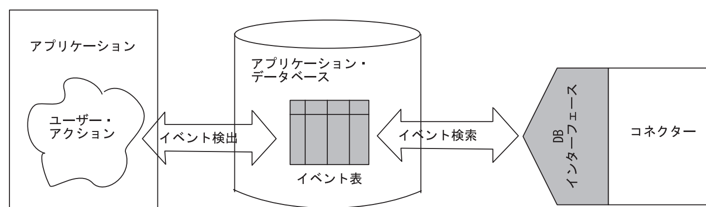

図 46. イベント・ストアとしてのイベント表

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格納し、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 41 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 41. イベント表およびアーカイブ表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。	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。
object_name	Char 80	ビジネス・オブジェクトの完全な名前。
object_verb	Char 80	Event 動詞。
object_key	Char 80	オブジェクトの基本キー。
event_priority	整数	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	日時	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	日時	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	整数	可能な状況値については、 126 ページの『イベント状況』を参照してください。

表 41. イベント表およびアーカイブ表に推奨されるスキーマ (続き)

列名	型	説明
event_description	Char 255	イベント記述またはエラー・ストリング。
connector_id	整数	コネクターの ID (該当する場合)。

E メール

イベント・ストアとして E 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 E メール・メッセージをメールボックスに送ります。
- イベント検索 — 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・メッセージを検索します。

図 47 にこのような対話を示します。

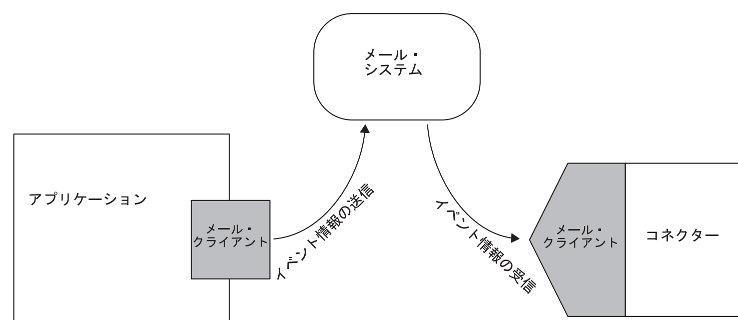

図 47. イベント・ストアとしてのメールボックス

E 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 - E 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

他のイベント検出メカニズム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。
- イベント検索 — 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 48 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検索を示します。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。

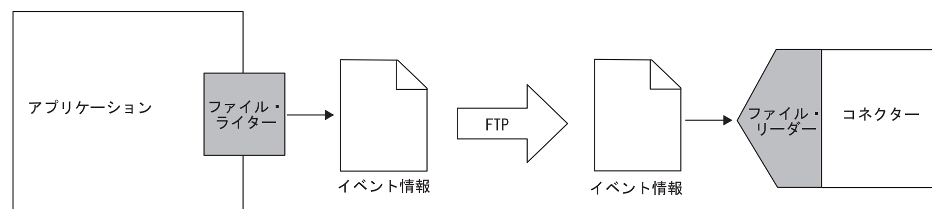

図 48. フラット・ファイルからのイベント・レコードの検索

イベント検出のインプリメント

ほとんどのコネクターの場合、イベント検出メカニズムをインプリメント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インプリメントされます。

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。

- 『イベント検出メカニズム』
- 134 ページの『イベント検出: 標準的な振る舞い』

イベント検出メカニズム

イベントのトリガーとなる条件は、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。イベント検出メカニズムがアプリケーションの中でセットアップされ、ビジネス・オブジェクトに関係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。

イベント検出メカニズム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メカニズムを提供するアプリケーションもあります。イベント検出メカニズムにイベント・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 つのインプリメント形態として、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メカニズムを持たず、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メカニズム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・トリガーをインプリメントしたり、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メカニズムも、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。

- 132 ページの『フォーム・イベント』
- 132 ページの『ワークフロー』
- 133 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、スクリプトが関係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・アクションをトリガーとするアプリケーション・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 49 にフォーム・ベースのイベント検出メカニズムを示します。ユーザーが「顧客」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。

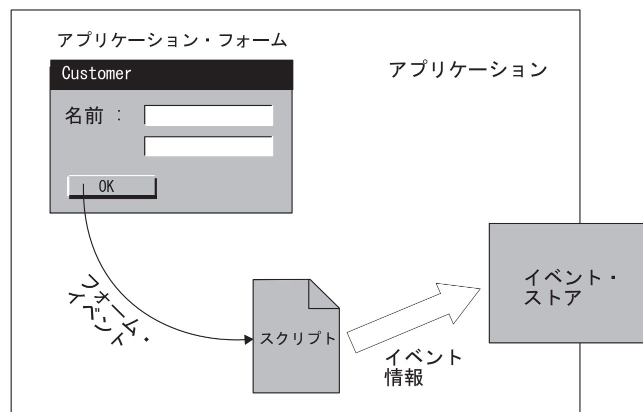

図 49. フォーム・ベースのイベント検出

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、イベント検出メカニズム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メカニズム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 50 に、ワークフロー・ベースのイベント検出メカニズムを示します。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。イベント検出メカニズム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。

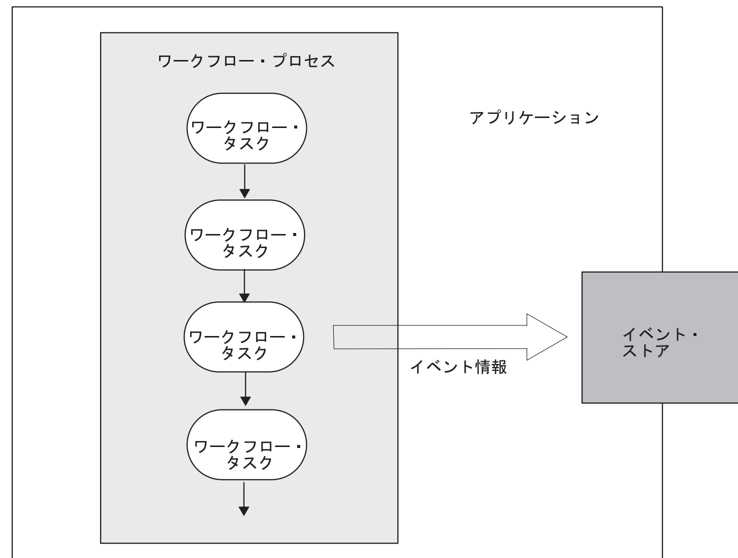

図 50. ワークフロー・ベースのイベント検出

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インプリメント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、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 51 に、アプリケーション表「顧客」を更新するユーザー・アクションを示します。「顧客」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。

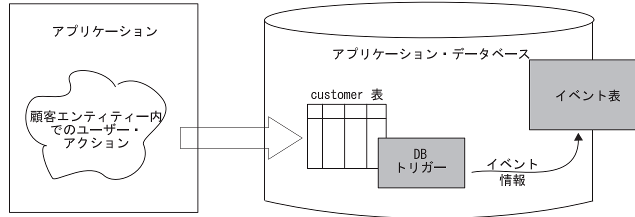

図 51. データベース・トリガーを使用したイベント検出

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既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・ルールをインプリメント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・イベント検出メカニズムは、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。イベント検出メカニズムは、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。
  - イベントの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: イベント検出メカニズムには、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、135 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。

イベント検出メカニズムは、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、イベント検出メカニズムで取り扱う問題を、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』
- 136 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

イベント検出メカニズムは、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インプリメントできます。この方法により、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` (住所) 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、イベント検出メカニズムは、1 つの Update イベントだけ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。

イベント検出メカニズムは、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、イベント検出メカニズムは、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 レコードを除去します。

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インプリメント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

イベント検出メカニズムは、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。この機能をインプリメントするには、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。将来のイベント・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、将来のイベント表に格納されます。

イベント検索のインプリメント

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検出メカニズムがインプリメントされます。これはコネクターの設計とインプリメントの一環として、コネクター開発者が担当します。イベント検出メカニズム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、イベント検出メカニズムは、このイベント検出メカニズムと連携して動作します。イベント検索により、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、イベント検索の次の点について説明します。

- 『イベント検出メカニズム』
- 『ポーリング・メカニズムの使用』

イベント検出メカニズム

イベント・ストアからイベント・レコードを検索するために使用される共通メカニズムは次の 2 つです。

- イベント・コールバック・メカニズム — このメカニズムを介して、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・メカニズム — 最も一般的なイベント検出メカニズムは、ポーリング・メカニズムです。

ポーリング・メカニズムの使用

アプリケーションは、ポーリング・メカニズムのため、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。

イベント検索を実行するポーリング・メカニズムをインプリメントするため、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、新規イベントがあれば取り込み、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: イベント情報を検索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 52 に、ポーリング・メソッドの基本動作を示します。

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。

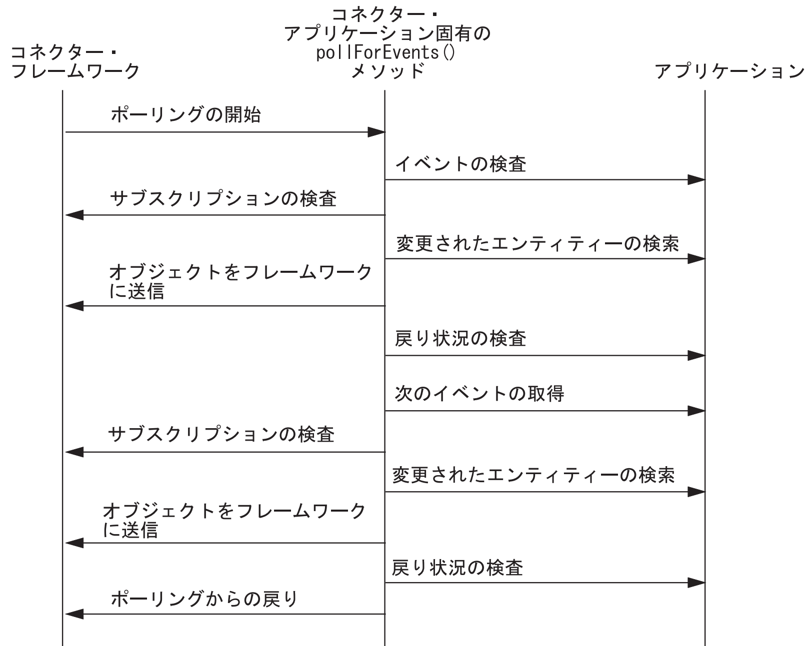

図 52. pollForEvents() メソッドの基本動作

`pollForEvents()` メソッドのインプリメント方法については、138 ページの『ポーリング・メソッドのインプリメント』を参照してください。

ポーリング・メソッドのインプリメント

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検索するため、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・レコードを検索し、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: コネクターによるインプリメント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()` の完全なインプリメント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、C++ コネクター・ライブラリーでは純粋な仮想メソッドとして定義されているため、少なくともこのメソッド用のスタブをインプリメントする必要があります。

このセクションでは、`pollForEvents()` メソッドのインプリメント方法の次の点について説明します。

- 139 ページの『pollForEvents() の基本ロジック』
- 139 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 53 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。

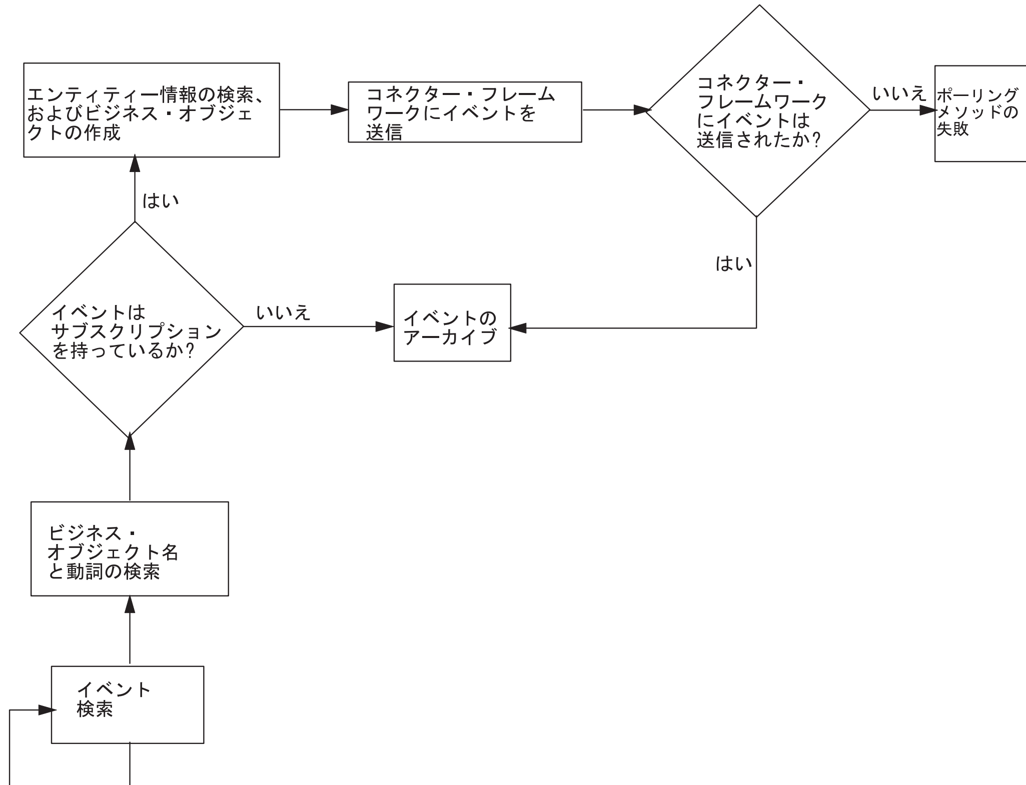

図 53. pollForEvents() の基本ロジックのフローチャート

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、 204 ページの『イベントのポーリング』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』
- 141 ページの『スレッド化の問題』
- 142 ページの『イベントの優先順位によるイベントの処理』
- 142 ページの『イベントの分散』

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、そのイベントの消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 42 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 42. アーカイブ・レコード内のイベント状況値

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りました。
Unsubscribed	イベントは検出されましたが、このイベントに対するサブスクリプションはなかったため、イベントはコネクター・フレームワークには送られませんでした。統合ブローカーにも送られません。
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』
- 141 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インプリメントします。

- 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、128 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。

- E 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・ストアのフォームにおいてイベント・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・アーカイブをインストール時に構成できるように設計しておくと、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なります。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、そのイベントの消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

デフォルトでは、C++ コネクターは単一スレッドです。したがって、ポーリング・メソッドが実行中には、コネクター・フレームワークは InterChange Server からビジネス・オブジェクトを受け付けません。コネクター・フレームワークは、要求ビジネス・オブジェクトやその他のメッセージを処理していないときにだけ、ポーリング・メソッドを呼び出します。そのため、ポーリング・メソッドは終了に長時間は要しません。

ポーリング・メソッドがイベントを処理中は要求処理がブロックされるため、1 回のポーリングで複数のイベントを処理するポーリング・メソッドをインプリメントするときには注意が必要です。

- 1 つのイベントを処理した後に復帰するポーリング・メソッドをインプリメントすると、ポーリングに要する時間が最小になります。ただし、アプリケーションが多数のイベントを生成するとき、イベントを 1 つずつ処理する方法では十分なパフォーマンスを確保できない場合があります。
- 1 回のポーリングで複数のイベントを処理するポーリング方式をインプリメントする場合には、多数のイベントを処理する必要性と、統合ブローカーからのビジネス・オブジェクト要求を処理する必要性とのバランスを取るインプリメントになっていることを確認してください。

注: コネクターが並列処理モード (ParallelProcessDegree が 1 より大) で稼働するように構成されている場合、 330 ページの表 108 に示すように、コネクターは複数のプロセスから構成され、各プロセスには固有の目的が割り振られています。このようなコネクターは、ポーリング・メソッドを実行中でも、要求処理をブロックしません。

イベントの優先順位によるイベントの処理

イベントの優先順位により、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検索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・インプリメンテーションでは、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インプリメントすることが必要です。

- イベント検出メカニズムは、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。
- イベント検出メカニズム (ポーリング・メカニズム) は、処理対象のイベント・レコードを検索する順序を、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、優先順位の値を下げられません。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、139 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。

イベントの分散

イベント検出メカニズムとイベント検出メカニズムは、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インプリメント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効率化し、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散をインプリメント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メカニズムを設計します。

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、イベント検出メカニズムの指定に基づいて、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・メソッドをインプリメントします。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・レコードを検索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 を検索します。

イベント処理における特別な考慮事項

このセクションでは、イベント処理に関する情報について説明します。内容は次のとおりです。

- 『削除イベントの処理』
- 145 ページの『保証付きイベント・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インプリメント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 (言い換えると、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、削除動作が物理削除、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インプリメント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』
- 『ビジネス・オブジェクト内の動詞の設定』
- 145 ページの『マッピング中の動詞の設定』

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メカニズムが、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、これは標準インプリメント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、イベント検出メカニズムは、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、イベント検出メカニズムは、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検索し、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・データを検索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検索し、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。

マッピング中の動詞の設定

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・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と、コネクター・フレームワークで、コネクターのイベント・ストアと統合ブローカーの間で 2 回イベントが送信されることがないようにできます。

重要: この機能を使用できるのは、JMS 対応コネクター、つまり、Java メッセージ・サービス (JMS) を使用して、メッセージ・トランスポートのキューを処理するコネクターに限られます。 JMS 対応コネクターでは、常にその `DeliveryTransport` コネクター・プロパティーが `JMS` に設定されています。コネクターを始動すると、JMS トランスポートが使用されるようになります。したがって、コネクターと統合ブローカー間の後続の通信はすべて、JMS トランスポートを通じて行われます。JMS トランスポートは、メッセージを最終的に宛先まで送信するための機能です。

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発行する時点 (コネクターが `pollForEvents()` メソッド内の `gotApplEvent()` メソッドを呼び出す時点) から、イベント・レコードを削除して (または、「イベント・ポスト済み」状況を指定してイベント・レコードを更新して) イベント・ストアを更新する時点の間に、起こりえる障害を示す小さなウィンドウが表示されます。このウィンドウ内で障害が発生するとイベントが送信されますが、そのイベントのレコードは、「ポーリング可能」状態でイベント・ストア内に残ります。コネクターは再始動すると、イベント・ストア内にこのイベント・レコードがまだ残っていることを検出し、それを送信するため、イベントは 2 回送信されることになります。

保証付きイベント・デリバリー機能を、以下のいずれかの方法を使用して JMS 対応コネクターに適応することができます。

- *コンテナー管理イベント* 機能の使用。コネクターが JMS イベント・ストア (JMS ソース・キューとしてインプリメントされたもの) を使用する場合、コネクター・フレームワークがコンテナーとして動作し、JMS イベント・ストアを管理します。詳細については、146 ページの『JMS イベント・ストアを使用するコネクター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* 機能の使用。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、重複イベントが発生しないようにすることができます。この機能は通常、JMS 以外のイベント・ストア (例えば、JDBC テーブル、E メールのメールボックス、またはフラット・ファイルとしてインプリメントされたもの) を使用するコネクターの場合に使用されます。詳細については、 149 ページの『JMS 以外のイベント・ストアを使用するコネクター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の保証付きイベント・デリバリー

JMS 対応コネクターが JMS キューを使用してイベント・ストアをインプリメントする場合、コネクター・フレームワークは「コンテナー」として動作し、JMS イベント・ストア (JMS ソース・キュー) を管理することができます。JMS の役割の 1 つは、トランザクション・キュー・セッションが開始された後、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。障害が発生するか、ロールバックが発行された場合は、メッセージを破棄します。したがって、コネクター・フレームワークは、1 つの JMS トランザクション内でソース・キューからメッセージを取り除き、そのメッセージを宛先のキューに配置できます。保証付きイベント・デリバリーのこの*コネクター管理イベント* 機能を使用すると、コネクター・フレームワークで、JMS イベント・ストアと宛先の JMS キューの間で 2 回イベントが送信されることがないようにできます。

このセクションでは、JMS イベント・ストアを有する JMS 対応のコネクターでの、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。

- 『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』
- 148 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 43 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 43. JMS イベント・ストアを使用するコネクターの、保証付きイベント・デリバリー機能関連のコネクター・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・ストアを単一ポーリングする際の処理するイベントの数

表 43. JMS イベント・ストアを使用するコネクターの、保証付きイベント・デリバリー機能関連のコネクター・プロパティー (続き)

コネクター・プロパティー	値
SourceQueue	コネクター・フレームワークがポーリングし、処理するイベントを検索するソースとなる、JMS ソース・キュー (イベント・ストア) の名前 **注:** ソース・キューとその他の JMS キューは、同じキュー・マネージャーの一部でなければなりません。コネクターのアプリケーションが、異なるキュー・マネージャーに格納されるイベントを生成する場合は、リモート・キュー・マネージャーのリモート・キュー定義を定義する必要があります。定義を行うと、WebSphere MQ では、イベントをリモート・キューから、JMS 対応コネクターが統合ブローカーとの伝送で使用するキュー・マネージャーへ転送することが可能になります。リモート・キュー定義を構成する方法については、IBM WebSphere MQ の資料を参照してください。

注: コネクターは、これらの保証付きイベント・デリバリー機能のうち *1* つだけしか 使用できません。つまり、コンテナー管理イベントまたは重複イベント回避のいずれか 1 つの機能だけ使用できます。したがって、ContainerManagedEvents プロパティーを JMS に設定し、かつ DuplicateEventElimination プロパティーを true に設定することはできません。

コネクターを構成する他に、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・ハンドラーも構成する必要があります。このデータ・ハンドラー情報は、表 44 に示すコネクター構成プロパティーで構成されます。

表 44. 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・ハンドラーが処理する MIME タイプ。この MIME タイプは、呼び出すデータ・ハンドラーを識別します。	はい
DHClass	データ・ハンドラーをインプリメントする Java クラスの完全名	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	オプション

注: データ・ハンドラー構成プロパティーは、その他のコネクター構成プロパティーとともに、コネクター構成ファイル内に入っています。

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 43 および 表 44 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を設定するには、Connector Configurator ツールを使用してください。Connector Configurator を使用すると、表 43 で示されたコネクター・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。同様に、表 44 に示されたコネクター・プロパティーは「データ・ハンドラー」タブに表示されます。

注: Connector Configurator で「データ・ハンドラー」タブに表示されているフィールドが有効になるのは、`DeliveryTransport` コネクター構成プロパティーおよび `ContainerManagedEvents` がともに `JMS` に設定されている場合のみです。

Connector Configurator の詳細については、 373 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、コネクターで保証付きイベント・デリバリー機能を使用すると、それを使用しない場合と比べて、コネクターの動作が多少変化します。コンテナーでイベントを管理するには、コネクター・フレームワークで以下のステップを実行してイベント・ストアをポーリングする必要があります。

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・キューとしてインプリメントされます。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. 適切なデータ・ハンドラーを呼び出し、イベントをビジネス・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、 147 ページの表 44 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. 結果メッセージを JMS 宛先キューに送信します。

WebSphere InterChange Server

JMS 宛先キューへ送信されるメッセージは、ビジネス・オブジェクトです。

その他の統合ブローカー

JMS 宛先キューへ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 トランザクションをコミットすると、同一トランザクション内でメッセージが JMS 宛先キューに書き込まれ、JMS ソース・キューから除去されます。
7. ステップ 1 から 6 をループで繰り返します。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まります。

重要: `ContainerManagedEvents` プロパティーに `JMS` が設定されているコネクターでは、イベント・ポーリングを実行する `pollForEvents()` メソッドは*呼び出されません*。コネクターの基底クラスに `pollForEvents()` メソッドが含まれていても、このメソッドは*呼び出されません*。

JMS 以外のイベント・ストアを使用するコネクター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* 機能を使用して、重複イベントが発生しないようにすることができます。この機能は通常、JMS 以外のソリューションを使用してイベント・ストア (例えば、JDBC イベント表、E メールのメールボックス、またはフラット・ファイルなど) をインプリメントする、JMS 対応のコネクターの場合に使用可能にします。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と、コネクター・フレームワークで、イベント・ストアと宛先の JMS キューの間で 2 回イベントが送信されることがないようにできます。

注: JMS イベント・ストアを使用する JMS 対応のコネクターでは通常、コンテナー管理イベント機能を使用します。ただし、コンテナー管理イベント機能ではなく、重複イベント回避機能を使用することもできます。

このセクションでは、JMS 以外のイベント・ストアを有する JMS 対応のコネクターでの、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。

- 『JMS 以外のイベント・ストアを使用するコネクターでの機能の使用可能化』
- 148 ページの『イベント・ポーリングへの影響』

JMS 以外のイベント・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 45 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 45. JMS 以外のイベント・ストアを使用するコネクターの、保証付きイベント・デリバリー機能関連のコネクター・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、処理済みのビジネス・オブジェクトの ObjectEventId を格納する JMS モニター・キューの名前

注: コネクターは、これらの保証付きイベント・デリバリー機能のうち *1* つだけしか 使用できません。つまり、コンテナー管理イベントまたは重複イベント回避のいずれか 1 つの機能だけ使用できます。したがって、DuplicateEventElimination プロパティーを true に設定し、かつ 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 45 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を設定するには、Connector Configurator ツールを使用してください。Connector Configurator を使用すると、これらのコネクター・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。Connector Configurator の詳細については、373 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、コネクターで保証付きイベント・デリバリー機能を使用すると、それを使用しない場合と比べて、コネクターの動作が多少変化します。重複イベント回避機能を使用するには、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・フレームワークで (pollForEvents() メソッドの gotApplEvent() への呼び出しにより) アプリケーション固有のコンポーネントからビジネス・オブジェクトを受け取ってから、(gotApplEvents() から受け取った) 現在のビジネス・オブジェクトが重複したイベントを表しているかどうかを判別する必要があります。この判別を行うために、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・オブジェクトを検索し、その ObjectEventId を現在のビジネス・オブジェクトの ObjectEventId と比較します。

- これら 2 つの ObjectEventId が同じであれば、現在のビジネス・オブジェクトが重複イベントであるということになります。このような場合、コネクター・フレームワークは、現在のビジネス・オブジェクトが表すイベントを無視します。つまり、このイベントを統合ブローカーに*送信しません*。
- これらの ObjectEventId が*同じでない* 場合、ビジネス・オブジェクトは重複イベントではありません。このような場合は、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、その後これを JMS デリバリー・キューにデリバリーします。これらの作業は、すべて同じ 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() メソッドを呼び出した後は、制御はコネクターの pollForEvents() メソッドに戻ります。

重複イベント回避機能をサポートする JMS 対応コネクターの場合、コネクターの pollForEvents() メソッドで、必ず以下のステップを行う必要があります。

- JMS 以外のイベント・ストアから検索したイベント・レコードからビジネス・オブジェクトを作成した場合は、イベント・レコードの固有イベント ID をビジネス・オブジェクトの ObjectEventId 属性として保管してください。

  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、かつこの

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」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それらの 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。

C++ コネクターで `BusinessObject` クラスの `setAttrValue()` 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。

```
pBusObj->setAttrValue("ObjectEventId", strngEventId, BOATTRTYPE::STRING);
```

- 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・メッセージまたは通知メッセージ
- 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。この章は、以下のセクションから構成されています。

- 『エラー・メッセージと通知メッセージ』
- 155 ページの『トレース・メッセージ』
- 159 ページの『メッセージ・ファイル』

エラー・メッセージと通知メッセージ

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。
- 次のような通知メッセージ
  - 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

コネクターは通知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。

© Copyright IBM Corp. 1997, 2004

- 「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 46 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 46. 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通知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。
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、161 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。

C++ コネクター・ライブラリーでは、logMsg()、generateMsg()、および generateAndLogMsg() メソッドは次の 2 つのクラスに定義されます。

- GenGlobals クラス。コネクター基底クラス内からロギングへのアクセス用。
- BOHandlerCPP クラス。ビジネス・オブジェクト・ハンドラー内からロギングへのアクセス用。

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、161 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跡づけることができます。

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、コネクター用に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の詳細は、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。

トレース宛先へのトレース・メッセージの送信

表 47 は、コネクターがトレース・メッセージをどのようにトレース宛先に送信するかを示しています。

表 47. トレース宛先へのトレース・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
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、161 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、64 ページの『ロケール依存設計原則』を参照してください。

C++ コネクター・ライブラリーでは、traceWrite()、generateMsg()、および generateAndTraceMsg() メソッドは次の 2 つのクラスに定義されます。

- GenGlobals クラス。コネクター基底クラス内からトレースへのアクセス用。
- BOHandlerCPP クラス。ビジネス・オブジェクト・ハンドラー内からトレースへのアクセス用。

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、158 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、LEVEL*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、Tracing クラスで定義されます。

下記の C++ コード断片では、traceWrite() を使用してレベル 4 のトレース・メッセージを作成し、アプリケーションから検索したレコードの数を記録します。

```
sprintf(msg, "Fetched %d record(s).", rCount);
traceWrite(Tracing::LEVEL4, msg, NULL);
```

トレース宛先に書き出されるトレース・メッセージには、このサンプル・コードの出力に示すように、日付、時刻、コネクター名、およびメッセージが含まれています。

```
[1999/05/28 12.36:48.105] [ConnectorAgent MyConnector]
                        Trace: Fetched 2 record(s).
```

generateMsg() メソッドと generateAndTraceMsg() メソッドは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。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、CxMsgFormat クラス内で定義されます。

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 48 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 48. 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。
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・オブジェクトで外部キーに遭遇したときに、またはコネクターがビジネス・オブジェクトで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。このメッセージの例には、ビジネス・オブジェクト間の一致の検索や子ビジネス・オブジェクトの配列でのビジネス・オブジェクトの検索などがあります。
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、コネクターの処理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に、メッセージは、それぞれの新規のスレッドの作成をログに記録します。

表 48. 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
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 から検索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・エラーをログまたはトレースするメソッドにメッセージを入力する際は、テキスト・ストリングまたはメッセージ・ファイル への参照として、入力できます。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・ファイル内にある既存のメッセージにのみ アクセスしてください。

このセクションでは、メッセージ・ファイルについて説明します。内容は次のとおりです。

- 160 ページの『メッセージ・フォーマット』
- 160 ページの『メッセージ・ファイルの名前とロケーション』
- 161 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形式は次のとおりです。

message number message text[EXPL]*explanation text*

message number は、メッセージを固有に識別する整数です。 message number は、1 行で (途中で改行せずに) 指定する必要があります。*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
コネクター・インフラストラクチャーのバージョンが一致しません。
```

メッセージには、値を実行時に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: アプリケーションからのイベントがサブスクライブされていません。オブジェクト名 {1}、
動詞 {2}。
```

メッセージ・パラメーターに値を指定する方法については、163 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、次のメッセージ・ファイルのいずれかから、メッセージを取得できます。

- 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・ファイルの名前は、`ClarifyConnector.txt` です。
- InterChange Server のメッセージ・ファイルの名前は `InterchangeSystem.txt` です。このファイルは、製品ディレクトリーに入っています。

メッセージを生成するすべてのメソッド (161 ページの表 49 を参照) は、最初に、指定されたメッセージ番号がコネクター・メッセージ・ファイルにないかどうかを検索します。

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、関係する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、63 ページの『国際化対応のコネクター』を参照してください。

メッセージ・ストリングの生成

表 49 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検索し、テキストの形式を設定し、生成されたメッセージ・ストリングを戻します。

表 49. 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。

表 49. メッセージ・ストリングを生成するメソッド (続き)

メッセージ・メソッド	説明
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。

C++ コネクター・ライブラリーでは、`generateMsg()`、`generateAndLogMsg()`、および `generateAndTraceMsg()` メソッドは次の 2 つのクラスに定義されます。

- `GenGlobals` クラス。コネクター基底クラス内からロギングへのアクセス用。
- `BOHandlerCPP` クラス。ビジネス・オブジェクト・ハンドラー内からロギングへのアクセス用。

表 49 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』
- 163 ページの『メッセージ・タイプの指定』
- 163 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 49 のメソッドには、引き数としてメッセージ番号を指定する必要があります。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。160 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、整数による固有のメッセージ番号 (ID) を付ける必要があります。表 49 のメソッドは、指定されたメッセージ番号をメッセージ・ファイルから検索し、関連したメッセージ・テキストを抜き出します。

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 49 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 50 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 50. 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	通知	通知メッセージのみ。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 50 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。

メッセージ・タイプ定数は、CxMsgFormat クラス内で定義されます。

次の C++ コード断片では、アプリケーションとの接続ができない場合にエラー・メッセージが記録されます。このエラー・メッセージは、コネクター・メッセージ・ファイルで次のように定義されています。「message 1100, Error: Failed to connect to application.」このコード例には、メッセージ番号を示すコメントが含まれているため、コードが読みやすくなっています。また、この例では、freeMemory() を呼び出して、generateMsg() が割り振ったメモリーを解放しています。

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
      msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
      logMsg(msg);
      JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
      return BON_FAIL;
}
```

注: C++ コネクターの場合、generateMsg() でメッセージを生成し、メッセージを記録した後、void freeMemory(char * mem) を呼び出して、generateMsg() が割り振ったメモリーを解放します。

パラメーター値の使用

表 49 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、160 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に 1 つのパラメーターを持つ通知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。下記のコード断片では、1 つのパラメーターを含むメッセージの形式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。

```
char val[512];
getConfigProp("ConnectorName", val, 512);
// Message 2887 - Initializing connector
generateAndLogMsg(2887, CxMsgFormat::XRD_INFO, 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。

コネクター・メッセージ・ファイルに次のように 1 つのパラメーターを持つトレース・メッセージがあるとします。

```
3033
Opened main form for {1} object.
```

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、フォーム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。下記のコード断片では、1 つのパラメーターを含むメッセージの形式を設定しそのトレース・メッセージをログに送信するために、generateAndTraceMsg() を呼び出しています。

```
char * formName[512];
if(getFormName(theObj,formName)==0)
               return BON_FAIL;
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
              // Message 3033 - Opened main form for object
             generateAndTraceMsg(3033,CxMsgFormat::XRD_TRACE,
                    Tracing::LEVEL3, 1, formName);
}
```

このコード断片では、アプリケーション・フォーム名を検索し、Tracing::getTraceLevel() を呼び出して、リポジトリーに設定されている現行トレース・レベルを検索します。トレース・レベルが 3 以上の場合、ルーチンは generateAndTraceMsg() を使用してメッセージ・ストリングを生成し、ログ宛先にトレース・メッセージを書き込みます。generateMsg() の呼び出しでは、argCount の値が 1 であり、val にはフォーム名の文字ストリングが入ることを指定します。

Item オブジェクトの場合、表示されるトレース・メッセージは次のとおりです。

```
[1999/05/28 12:00:00.000] [ConnectorAgent MyConnector]
              Trace 3033: Opened main form for Item object
```

第 7 章 C++ コネクターのインプリメント

この章では、C++ 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インプリメント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。

この章は、以下のセクションから構成されています。

- 『C++ コネクター基底クラスの拡張』
- 168 ページの『コネクターの実行開始』
- 172 ページの『ビジネス・オブジェクト・ハンドラーの作成』
- 204 ページの『イベントのポーリング』
- 224 ページの『コネクターのシャットダウン』
- 224 ページの『エラーと状況の処理』

C++ コネクター基底クラスの拡張

C++ コネクター・ライブラリーで、コネクター基底クラスは `GenGlobals` という名前です。 C++ コネクターでは、基底クラス・メソッドは純粋仮想メソッドです。`GenGlobals` クラスは、コネクターの始動とシャットダウン、コネクター構成プロパティーへのアクセス、およびロギング、トレースのためのユーティリティー・メソッドを指定します。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『コネクターの基底クラスの拡張』を参照してください。

C++ 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. `GenGlobals` クラスを継承するコネクター・クラスを作成し、ヘッダー・ファイル `GenGlobals.hpp` を組み込みます。このクラスには、次の名前を付けることを推奨します。

   connectorName`Globals.cpp`

   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 アプリケーションと通信するコネクターを作成するには、コネクター・クラスに `BaanGlobals.cpp` という名前を付けます。

   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。

2. コネクター・メソッドに対して `GenGlobals` 純粋仮想メソッドをインプリメントする。これらの仮想メソッドの作成方法の詳細については、表 51 を参照してください。

© Copyright IBM Corp. 1997, 2004

3. コネクター・フレームワークにグローバル・コネクター・クラスへのハンドルを指定します。

表 51. GenGlobals クラスの仮想メソッドの拡張

仮想 **GenGlobals** メソッド	説明	詳細情報
`init()`	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。	168 ページの『コネクターの初期化』
`getVersion()`	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取得します。	169 ページの『コネクター・バージョンの確認』
`getBOHandlerforBO()`	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。	170 ページの『C++ ビジネス・オブジェクト・ハンドラーの取得』
`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。	172 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。	204 ページの『イベントのポーリング』
`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。	224 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 52 に示すコネクター・クラス・メソッドを呼び出します。

表 52. コネクターの実行開始

初期化タスク	詳細情報
1. 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、アプリケーションとの接続のオープンなど、コネクターのために必要な初期設定をすべて実行する。	168 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各ビジネス・オブジェクトに対して、ビジネス・オブジェクト・ハンドラーを取得する。	170 ページの『C++ 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・クラス (`GenGlobals` から派生) の初期化メソッド `init()` 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・クラスのインプリメントの一環として、コネクターの `init()` メソッドをインプリメントする必要があります。

73 ページの『コネクターの初期化』で説明されているように、`init()` 初期化メソッドのメインタスクには次のようなタスクがあります。

- 169 ページの『接続の確立』
- 169 ページの『コネクター・バージョンの確認』

- 170 ページの『進行中イベントのリカバリー』

上記のトピックのほかに、このセクションでは、170 ページの『C++ 初期化メソッドの例』での C++ init() メソッドの例を示します。

重要: 初期化メソッド実行の間は、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、まだ使用可能になっていません。

接続の確立

init() 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。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、init() メソッドは、障害状況を戻し、障害の原因を常時示します。C++ コネクターで、init() で使用される一般的な戻りコードは、BON_SUCCESS、BON_FAIL、および BON_UNABLETOLOGIN です。上記を含め、その他の戻りコードの詳細については、224 ページの『エラーと状況の処理』を参照してください。

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・バージョンの確認

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・バージョンの確認』を参照してください。

C++ コネクター・ライブラリーでは、getVersion() メソッドは GenGlobals クラスに定義されています。コネクターのバージョンを示す文字ストリングへのポインターを戻します。コネクターのバージョン・ストリングを戻すとき、コネクター・フレームワークによって定義されているデフォルト・バージョンを戻すことを選択できます。デフォルト・バージョンを戻すことを選択するには、以下の図 54 で示すように、ストリング CX_CONNECTOR_VERSIONSTRING を戻します。

```
char * ExampleGenGlob::getVersion()
{
   return (char *) CX_CONNECTOR_VERSIONSTRING;
}
```

図 54. C++ コネクターの getVersion() メソッド

別の方法として、ConnectorVersion.h という名前のファイルにバージョン・ストリングを設定して、デフォルトのバージョンをオーバーライドすることができます。ConnectorVersion.h に次の行、またはそれに類似した行を組み込みます。

```
#ifndef CX_CONNECTOR_VERSIONSTRING
#define CX_CONNECTOR_VERSIONSTRING    "3.0.0"
#endif
```

進行中イベントのリカバリー

進行中イベントをリカバリーするには、アプリケーションが指定する手法を使用して、イベント・ストア内のイベント・レコードにアクセスする必要があります。例えば、コネクターは ODBC API および ODBC SQL コマンドを使用できます。進行中イベント・レコードが検索されたら、コネクターは 表 23 内のアクションのいずれかを行って、進行中イベントを処理することができます。

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。

C++ 初期化メソッドの例

C++ コネクターの場合、`init()` メソッドは、初期化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指定します。このメソッドは、初期化操作の状況を示す整数を戻します。図 55 に、C++ コネクター用の初期化メソッドの例を示します。この例で、`init()` メソッドは `GenGlobals::getConfigProp()` を呼び出して、リポジトリーからコネクター構成のプロパティーを検索します。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、構成値を使用して、アプリケーションにログオンし、他の必要な初期化タスクを実行することができます。

```
int ExampleGenGlob::init(CxVersion *version)
{
   char val[512];
   char val1[512];

   int return_code = BON_SUCCESS;

   // get values for connector configuration properties
   getConfigProp("ConnectorName", val, 512);
   //Log trace message "Initializing {ConnectorName}"
   traceWrite(Tracing::LEVEL5, generateMsg(20050,
            CxMsgFormat::XRD_INFO, NULL, 1, val), NULL);

   . . .
   getConfigProp("Hostname",val1, 512);

   // use configuration values to log in to the application
   // If log in fails, log error message.
   logMsg(generateMsg(21000,CxMsgFormat::XRD_ERROR, NULL, 1, name));

   return return_code;
}
```

図 55. C++ コネクターの初期化

C++ ビジネス・オブジェクト・ハンドラーの取得

C++ コネクターで、ビジネス・オブジェクト・ハンドラーの基底クラスは、`BOHandlerCPP` です。サポートされるビジネス・オブジェクト用のビジネス・オブジェクト・ハンドラーのインスタンスを取得するため、コネクター・フレームワークは `getBOHandlerforBO()` メソッドを呼び出します。このメソッドは、`GenGlobals` クラスの一環として定義されます。

注: getBOHandlerforBO() 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、83 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。

コネクター・フレームワークが getBOHandlerforBO() メソッドへの呼び出しを介して取得するビジネス・オブジェクト・ハンドラーの数は、コネクターでのビジネス・オブジェクト処理の全体的設計によって異なります。

- 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 56 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 getBOHandlerforBO() メソッドのインプリメンテーションが含まれています。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 (GenBOHandler クラスから派生) のコンストラクターを呼び出します。これにより、コネクターでサポートされるビジネス・オブジェクトをすべて 処理する単一のビジネス・オブジェクト・ハンドラー・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ビジネス・オブジェクトの一部または全部に、特殊処理が必要な場合は、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・ハンドラーを設定する必要があります。

  図 57 には、Invoice および Item ビジネス・オブジェクトに対して固有のビジネス・オブジェクト・ハンドラーを作成し、他のすべてのビジネス・オブジェクトに対して汎用のビジネス・オブジェクト・ハンドラーを作成する getBOHandlerforBO() メソッドのインプリメンテーションが含まれています。

重要: getBOHandlerforBO() 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 56 は、GenBOHandler 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・クラスのインスタンスを生成します。

```
BOHandlerCPP *ExampleGenGlob::getBOHandlerforBO(char * BOName)
{
   //   If a business object handler does not exist,
   //   create one; otherwise, return the existing pointer
   if(pHandlr == NULL) {
      pHandlr = new GenBOHandler();
   }
   return pHandlr;
}
```

図 56. 汎用ビジネス・オブジェクト・ハンドラーの getBOHandlerforBO() メソッド

図 57 は、渡されるビジネス・オブジェクト名に基づいて、該当するビジネス・オブジェクト・ハンドラーのコンストラクターを呼び出します。

- ビジネス・オブジェクトの名前が「App_Invoice」である場合には、Invoice_Handler ビジネス・オブジェクト・ハンドラー・クラスのコンストラクターを呼び出します。

- ビジネス・オブジェクトの名前が「App_Item」である場合には、Item_Handler ビジネス・オブジェクト・ハンドラー・クラスのコンストラクターを呼び出します。
- ビジネス・オブジェクトの名前が他のストリングである場合には、Generic_Handler ビジネス・オブジェクト・ハンドラー・クラスのコンストラクターを呼び出します。

```
BOHandlerCPP *ExampleGenGlob::getBOHandlerforBO(char * BOName)
{
   if (strcmp(BOName, App_Invoice)==0)
      return new Invoice_Handler();
   else if (strcmp(BOName, App_Item)==0)
      return new Item_Handler();
   else
      return new Generic_Handler();
   }
}
```

図 57. 複数のビジネス・オブジェクト・ハンドラーの getBOHandlerforBO() 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。

- 『C++ ビジネス・オブジェクト・ハンドラー基底クラスの拡張』
- ビジネス・オブジェクト・ハンドラー検索メソッドのインプリメント ― 詳細については、170 ページの『C++ 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。
- 173 ページの『doVerbFor() メソッドのインプリメント』

注: 要求処理の概要については、28 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、83 ページの『第 4 章 要求処理』を参照してください。

C++ ビジネス・オブジェクト・ハンドラー基底クラスの拡張

C++ 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 BOHandlerCPP です。 BOHandlerCPP クラスは、ビジネス・オブジェクト・ハンドラーを定義およびアクセスするためのメソッドを指定します。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、86 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。

C++ コネクターのビジネス・オブジェクト・ハンドラー・クラスを派生させるには、次のステップを実行します。

1. BOHandlerCPP 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.cpp

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・クラスを作成できます。複数のビジネス・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. 仮想メソッド doVerbFor() をインプリメントして、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。BOHandlerCPP クラス内の他のメソッドは、すでにインプリメントされています。この仮想メソッドのインプリメント方法の詳細については、173 ページの『doVerbFor() メソッドのインプリメント』を参照してください。

注: BOHandlerCPP クラス内のその他のメソッドでは、インプリメントが行われています。doVerbFor() メソッドは、このクラス内の*唯一の* 仮想メソッドです。詳細については、261 ページの『第 11 章 BOHandlerCPP 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・ハンドラーのインプリメント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、83 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() メソッドのインプリメント

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、適切なビジネス・オブジェクト・ハンドラー内の doVerbFor() メソッドを呼び出して、このビジネス・オブジェクトの動詞のアクションを実行します。C++ コネクターでは、BOHandlerCPP クラスは doVerbFor() 仮想メソッドを定義します。この仮想メソッドのインプリメンテーションは、ビジネス・オブジェクト・ハンドラー・クラスの一環として行う必要があります。

注: doVerbFor() メソッドの役割の一般的説明については、87 ページの『要求の処理』を参照してください。88 ページの図 25 に、このメソッドの基本ロジックを示します。

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 53 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。詳細情報の参照先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 53. *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。	174 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との有効な接続を維持しているかどうかを検証します。	175 ページの『動詞を処理する前の接続の検証』
3.	有効なアクティブ動詞の値で分岐します。	177 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。	180 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。	182 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 53 に示す処理ステップのほかに、このセクションでは 199 ページの『処理に関するその他の問題』に追加の処理情報を記載します。

アクティブ動詞の取得

実行するアクションを判別するには、doVerbFor() メソッドは、最初に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検索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 54 に、要求ビジネス・オブジェクトからアクティブ動詞を検索するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 54. *アクティブ動詞の取得用メソッド*

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	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、BusinessObject オブジェクトとして doVerbFor() メソッドに渡されます。
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れば、BusinessObject 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することができます。
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 null 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 55 で説明されているステップを実行します。

表 55. 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。	BOHandlerCPP.logMsg()、 BOHandlerCPP.generateAndLogMsg()
2.	動詞処理障害の原因を示すために、戻り状況記述子内にメッセージを設定します。	ReturnStatusDescriptor.seterrMsg()
3.	doVerbFor() メソッドから BON_FAIL 結果状況を戻します。	return BON_FAIL;

図 58 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() メソッドのフラグメントが含まれています。このコードによって、要求ビジネス・オブジェクトおよびそのアクティブ動詞が非ヌルであることを確認できます。これらの状況のいずれかが存在する場合、コード・フラグメントは戻り状況記述子内にメッセージを格納し、BON_FAIL の結果状況で終了します。

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   //make sure that the incoming business object is not null
   if (theObj == null) {
      generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);

      char errorMsg[512];
      sprintf(errorMsg,
         "doVerbFor: Invalid request business object .");
      rtnStatusMsg->seterrorMsg(errorMsg);
      status = BON_FAIL;
   }

   // obtain the active verb
   char *verb = theObj.getVerb();

   // make sure the active verb is neither null nor empty
   if (verb == null || strcmp(verb, "")){
      generateAndLogMsg(6548, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
      sprintf(errorMsg,"doVerbfor: Invalid active verb.");
      rtnStatusMsg->seterrorMsg(errorMsg);
      status = BON_FAIL;

   // perform verb processing here
   ...
}
```

図 58. アクティブ動詞の取得

動詞を処理する前の接続の検証

コネクター・クラス内の init() 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、その最も一般的なタスクの 1 つは、アプリケーションと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と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、アプリケーションと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、表 56 で説明されているステップを実行して、失われた接続のコネクター・フレームワークを通知します。

表 56. 失われた接続の処理

	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。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、電子メール通知を発生させます。	BOHandlerCPP.logMsg()、 BOHandlerCPP.generateAndLogMsg()
2.	失われた接続の原因を示すために、戻り状況記述子内にメッセージを設定します。	ReturnStatusDescriptor.seterrMsg()
3.	doVerbFor() メソッドから BON_APPRESPONSETIMEOUT 結果状況を戻します。	return BON_APPRESPONSETIMEOUT;

注: この戻り状況記述子オブジェクトは、doVerbFor() がコネクター・フレームワークに送信する動詞処理応答の一環です。これらのメソッドの詳細については、 337 ページの『第 18 章 ReturnStatusDescriptor クラス』を参照してください。

コネクターは、BON_APPRESPONSETIMEOUT 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後に、コネクターが実行し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。

図 59 には、アプリケーションとの接続の切断を処理するコードが含まれています。この例では、エラー・メッセージ 20018 が発行されて、コネクターからアプリケーションとの接続が切断したこと、およびアクションを実行する必要があることが管理者に通知されます。

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtn)
{
   ...
   if (//application is not responding ) {
      // Lost connection to the application
      // Log an error message
      logMsg(generateMsg(20018, CxMsgFormat::XRD_FATAL, NULL, 0,
         "MyConnector"));

      // Populate a ReturnStatusDescriptor object
      char errorMsg[512];
      sprintf(errorMsg, "Lost connection to application");
      rtnObj->seterrMsg(errorMsg);

      return BON_APPRESPONSETIMEOUT;   }
   ....
   // if connection is open, continue processing
   ...
}
```

図 59. doVerbFor() 内での接続の切断の例

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』
- 178 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: 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リカバリー・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、175 ページの表 55 を参照してください。

図 60 に、create、update、retrieve、および delete 操作を処理する、基本の doVerbFor() メソッドを示します。このコードは、Create、Update、Retrieve、および Delete 動詞のアクティブ動詞の値で分岐します。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。続いて、対応する動詞メソッドを呼び出して、ビジネス・オブジェクト処理を続行します。

このコード断片の上部で、この C++ doVerbFor() メソッドは、異なる動詞を識別するために特殊の定数を定義します。これらの動詞定数を使用すると、コード内でのアクティブ動詞の識別やそのストリング表記の変更が容易になります。コネクター

が追加の動詞を処理する場合には、String 定数を拡張 BOHandlerCPP クラスの一部として定義することをお勧めします。

```
#define CREATE "Create"
#define UPDATE "Update"
#define RETRIEVE "Retrieve"
#define DELETE "Delete"

int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   // Determine the verb of the incoming business object
   char *verb = theObj.getVerb();

   if (strcmp(verb, CREATE) == 0)
      status = doCreate(theObj);
   else if (strcmp(verb, UPDATE) == 0)
      status = doUpdate(theObj);
   else if (strcmp(verb, RETRIEVE) == 0)
      status = doRetrieve(theObj);
   else if (strcmp(verb, DELETE) == 0)
      status = doDelete(theObj);
   else
   {
      // This verb is not supported.
      // Send the collaboration a message to that effect
      // in the ReturnStatusDescriptor object.
      char errorMsg[512];
      sprintf(errorMsg,"doVerbFor: verb '%s' is not supported ",
         verb);
      rtnStatusMsg->setErrorMsg(errorMsg);
      status = BON_FAIL;
   }

   // Return status to connector framework
   return status;
}
```

図 60. アクティブ動詞の値での分岐

図 60 内のコード断片はモジュール化されています。すなわち、このコード断片は、それぞれのサポートされている動詞の実処理を別個の*動詞メソッド*: doCreate()、doUpdate()、doRetrieve()、および doDelete() に入れます。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。

- BusinessObject パラメーターを定義します。したがって、動詞メソッドは、要求ビジネス・オブジェクトを受信することができ、また、この更新されたビジネス・オブジェクトを呼び出しメソッドに返信できる可能性があります。
- 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メソッドでは、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 57 に、ビジネス・オブジェクトの動詞に関するアプリケーション固有の情報を取得す

るために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 57. 動詞のアプリケーション固有の情報の検索用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusObjSpec	getVerbAppText()

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。

図 61 に、ビジネス・オブジェクトのすべての動詞処理をインプリメントする、フォームをベースとする、メタデータ主導型 doVerbFor() メソッドを示します。ビジネス・オブジェクトのアプリケーション固有の情報を使用して、メソッドはフォーム名を識別し、ビジネス・オブジェクト属性をループして、属性の記述を検索します。それぞれの属性の記述は、BOAttrType クラスのインスタンスです。このクラスを介して、メソッドは、属性のアプリケーション固有の情報、およびそれがキーであるかどうかなどの、属性に関する他の情報を取得することができます。

注: ビジネス・オブジェクトの処理方法の詳細については、182 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。

メソッドは、ビジネス・オブジェクト・インスタンスから属性値を検索し、属性のメタデータを使用してフォームに記入して、それぞれの属性ごとにフォームのフィールドを確認します。メソッドは、ビジネス・オブジェクト内の動詞操作を識別し、動詞のメタデータを検索して処理命令を取得し、完全なフォームをアプリケーションに送信します。これが Create 操作であり、アプリケーションが、キーなどの新しいデータを作成する場合、メソッドはアプリケーションからデータを検索してそれを処理します。

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
      ReturnStatusDescriptor *rtnObj)
{
   BusObjSpec                  *theSpec;
   int                status = BON_SUCCESS;

   // Get the business object definition and its metadata:
   // the name of the form. Open the specified form
   theSpec = theObj.getSpecFor();
   form = OpenForm(theObj.getAppText());

   // For each attribute, retrieve the attribute description,
   // get the attribute values and application-specific information,
   // and set the field of the form
   for (int i = 0; i < theObj.getAttrCount; i++) {
      BOAttrType * curAttr = theObj.getAttrDesc(i);
      Form.setfield(curAttr->getAppText(),theObj.getAttrValue(i));
   }

   // Get the verb and the verb metadata: the type of operation
   // to perform. Tell the application to do the operation
   Form.doOperation(theSpec->getVerbAppText(theObj.getVerb()));

   // Process returned attributes if any
   for (int k=0; k < theObj.getAttrCount() -1; k++) {
      BOAttrType * curAttr = theObj.getAttrDesc(k);
      value = Form.getField(curAttr->getAppText();
      theObj.setAttrValue(k, value);
   }
   return status;
}
```

図 61. メタデータ主導型動詞処理

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 58 に、doVerbFor() メソッドがインプリメントできる標準動詞と、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の詳細については、182 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 58. 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。	91 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検索するかを判別します。 • 属性キー値 (複数可) を使用して、検索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。	95 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。	103 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 が関係テーブル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。	111 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 59 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 59. 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。	182 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。	184 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情報の抽出』
3.	属性情報を取得します。	185 ページの『属性のアクセス』
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。	186 ページの『属性のアプリケーション固有情報の抽出』
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。	188 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。	190 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。	192 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』

このセクションでは、例の Create メソッドについて、その機能を詳細に説明しながら、その基本ロジックをウォークスルーします。この例の動詞メソッドでは、表 59 に示す基本プログラム・ロジックを使用して、ビジネス・オブジェクトの分解を行い、ODBC SQL コマンドを作成します。完全な動詞メソッドについては、196 ページの『例: フラット・ビジネス・オブジェクトの Create メソッド』を参照してください。

ビジネス・オブジェクト定義へのアクセス: C++ コネクターでは、doVerbFor() メソッドは、要求ビジネス・オブジェクトを BusinessObject クラスのインスタンスとして受信します。ただし、動詞処理を開始するには、doVerbFor() メソッドは多くの場合、BusObjSpec クラスのインスタンスである、ビジネス・オブジェクト定義からの情報を必要とします。したがって、標準的 C++ 動詞操作での最初のステップは、要求ビジネス・オブジェクトのビジネス・オブジェクト定義へのポインターを検索することです。

表 60 に、現在のビジネス・オブジェクト (BusinessObject インスタンス) のビジネス・オブジェクト定義を取得するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 60. ビジネス・オブジェクト定義の取得用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	getSpecFor()

doSimpleCreate() という名前の動詞メソッドが、表ベースのアプリケーションの Create 動詞に対する処理をインプリメントするとします。図 62 に、要求ビジネス・オブジェクト (theObj) のビジネス・オブジェクト定義 (theSpec) を取得するために、ビジネス・オブジェクト定義 getSpecFor() を呼び出すための 1 つの方法を示します。

```
int doSimpleCreate(BusinessObject &theObj)
{
   ...
   BusObjSpec *theSpec = theObj.getSpecFor()
```

図 62. ビジネス・オブジェクト定義の取得

注: コネクター始動時に、コネクターは、コネクターがサポートする、すべての ビジネス・オブジェクト定義のための BusObjSpec インスタンスを生成します。getSpecFor() メソッドは、要求ビジネス・オブジェクトに関連したビジネス・オブジェクト定義のインスタンスへのポインターを戻します。

getSpecFor() が BusObjSpec インスタンスへの参照を取得すると、doVerbFor() メソッドは BusObjSpec クラスのメソッドを使用して、ビジネス・オブジェクト定義から情報 (そのアプリケーション固有の情報など) を取得し、属性記述子にアクセスすることができます。ビジネス・オブジェクト定義には、表 61 に示す情報が含まれます。BusObjSpec メソッドの完全リストについては、 299 ページの『第 14 章 BusObjSpec クラス』を参照してください。

表 61. 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	BusObjSpec 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()
属性のリスト — それぞれの属性ごとに、BusObjSpec オブジェクトが指定します。	getAttributeCount()
• 属性のリスト内の位置	getAttributeIndex()
• それぞれの属性ごとの属性記述子 (BOAttrType インスタンス)。詳細については、185 ページの『属性のアクセス』を参照してください。	getAttribute()
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()
• 動詞	getVerbAppText()
注: 属性に関するアプリケーション固有の情報へのアクセスは、BOAttrType クラスで指定されます。	

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情報の抽出: メタデータ主導型コネクターのためのビジネス・オブジェクトは通常、アプリケーション構造に関する情報を提供するアプリケーション固有の情報を持つように設計されます。そのようなコネクターの場合、標準的な動詞操作は、要求ビジネス・オブジェクトに関連したビジネス・オブジェクト定義からアプリケーション固有の情報を検索する必要があります。表 62 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検索するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 62. ビジネス・オブジェクトのアプリケーション固有情報を取得するためのメソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusObjSpec	getAppText()

注: 表 62 に示す、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では、使用すべきでない用語をそのメソッド名で使用しています。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。

表 62 に示すように、コネクターは getAppText() 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します。

```
char * appInfo = theSpec->getAppText());
```

getAppText() メソッドは、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が含まれている文字ストリングを検索します。115 ページの図 38 に示す、ビジネス・オブジェクトの例を使用して、上記のコード行は表名 customer を変数 appText にコピーします。

図 62 の doSimpleCreate() メソッドでは、動詞メソッドは、表ベースのアプリケーションのための Create 動詞の処理をインプリメントします。そのようなアプリケーションの場合、ビジネス・オブジェクトは通常、アプリケーション構造に関する情報をアプリケーション固有の情報が動詞操作に指定するように設計されています (詳細については、116 ページの表 36 を参照してください)。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。

動詞メソッドは最初に、ビジネス・オブジェクト定義を介してアプリケーション固有の情報にアクセスします。したがって、動詞メソッドは BusObjSpec::getAppText() を呼び出して、アクセスするデータベース表の名前を取得します。続いて、コネクターは、検索した表名を使用して、アプリケーション・データベースにアクセスする SQL ステートメントの作成を開始します。Create 操作の場合、SQL ステートメントは INSERT です。

示されている Customer ビジネス・オブジェクトの例を使用して、コネクター 図 63 内のコード断片は、新規の行を customer という名前のアプリケーション・データベース表に追加する INSERT ステートメントを構成します。動詞メソッドの実行でのこの時点で、SQL ステートメントは次のとおりです。

```
INSERT INTO customer
```

```
int doSimpleCreate(BusinessObject &theObj)
{
   char                         table_name[64];
   char                         insertStatement[1024];
   BusObjSpec                    *theSpec;

   // Retrieve pointer to the business object definition
   theSpec = theObj.getSpecFor();

// Retrieve the table name from the AppSpecificInfo property
// for the business object definition
strcpy(table_name, theSpec->getAppText());

// Begin building the SQL INSERT statement
sprintf(insertStatement, "INSERT INTO %s (", table_name);
...
}
```

図 63. データベース表の名前の取得

属性のアクセス: C++ コネクターでは、doVerbFor() メソッドは、要求ビジネス・オブジェクトを BusinessObject クラスのインスタンスとして受信します。ただし、動詞操作は、属性プロパティーに関する情報を取得する必要がある場合には、BOAttrType クラスのインスタンスである、*属性記述子* にアクセスする必要があります。したがって、標準的な C++ 動詞操作は、要求ビジネス・オブジェクトでアクセスする必要があるそれぞれの属性記述子へのポインターを検索しなければなりません。

表 63 に、現在のビジネス・オブジェクトから属性記述子を取得するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 63. 属性記述子を取得するためのクラスとメソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusObjSpec	getAttribute(),, getAttributeCount(),, getAttributeIndex()
BusinessObject	getAttrDesc(),, getAttrCount()

属性記述子にアクセスするには、コネクターは次のメソッドのいずれかを使用することができます。

- BusObjSpec クラスの getAttribute() メソッドが、ビジネス・オブジェクト定義から属性記述子を取得します。
- BusinessObject クラスの getAttrDesc() メソッドが、ビジネス・オブジェクトから属性記述子を取得します。

getAttribute() および getAttrDesc() メソッドは、次の 2 つの方法のいずれかで属性記述子にアクセスすることができます。

- 属性名 — 属性の名前プロパティーで属性を識別して、属性記述子を取得できます。

```
theAttr = theSpec->getAttribute(attrName);
```

- 整数索引 — 次のいずれかの方法で属性索引 (順序位置) を取得できます。
  - getAttributeCount() でビジネス・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、それらの属性を一度に 1 つずつ、それぞれの指標値を getAttribute() に渡しながらループして、属性記述子を取得します。

    注: 別の方法としては、BusinessObject::getAttrCount() メソッドで、ビジネス・オブジェクト自体から属性カウントを取得することができます。
  - 特定の属性の索引を取得します。属性の指標名を getAttributeIndex() に指定することで、属性の指標を取得することができます。

  getAttribute() への次の呼び出しは、ビジネス・オブジェクト定義の、指定された順序位置 (変数 i で示される) にある属性を表す BOAttrType インスタンスにポインターを返します。

```
for (i = 0; i < theSpec->getAttributeCount(), i++) {
      theAttr = theSpec->getAttribute(i);
// do processing on the attribute descriptor
}
```

属性記述子が存在すると、コネクターは BOAttrType クラスのメソッドを使用して、関連属性のプロパティーに関する、そのカーディナリティーまたは最大長などの、情報を取得することができます。表 64 に、属性記述子から情報を検索するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。BOAttrType クラスの完全なメソッドのリストについては、 249 ページの『第 10 章 BOAttrType クラス』を参照してください。

表 64. 属性プロパティーに関する情報の取得用メソッド

属性プロパティー	BusObjAttr メソッド
Name	getName(),, hasName()
Type	getRelationType(),, getTypeName(),, getTypeNum(),, hasTypeName(),, isObjectType(),, isType()
Key	isKey()
Foreign key	isForeignKey()
Max Length	getMaxLength()
Required	isRequired()
Cardinality	getCardinality(),, hasCardinality(),, isMultipleCard()
Default Value	getDefault()
属性のアプリケーション固有の情報	getAppText()

重要: 属性値は、ビジネス・オブジェクト定義にある属性記述子 (BOAttrType インスタンス) 内からは使用できません。BusinessObject インスタンス内の属性を介して属性値にアクセスしなければなりません。詳細については、190 ページの『ビジネス・オブジェクトからの属性値の抽出』を参照してください。

属性のアプリケーション固有情報の抽出: 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 65 に、属性記述子からアプリケーション固有の情報を検索するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 65. 属性のアプリケーション固有情報を取得するためのメソッド

C++ コネクター・ライブラリー・クラス	メソッド
BOAttrType	getAppText()

注: 表 65 に示す、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では、使用すべきでない用語をそのメソッド名で使用しています。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。

コネクターは getAppText() メソッドを使用して、属性に関するアプリケーション固有の情報を取得します。表ベースのアプリケーションに関する情報をアプリケーション固有の情報が指定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には、属性に関するアプリケーション固有の情報には、この属性に関連したアプリケーション表の列の名前が含まれています (詳細については、116 ページの表 36 を参照してください)。

115 ページの図 38 に示されているビジネス・オブジェクトの例を使用して、コネクター図 63 内のコード断片は、新規の行を customer という名前のアプリケーション・データベース表に追加する INSERT ステートメントの構成を開始します。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、次のステップは、ビジネス・オブジェクト要求によって更新する、アプリケーション表内の列を決定することです。続いて、コネクターは、検索した列名を使用して、SQL ステートメントの作成を続行します。コネクターは、新規の行を customer という名前のアプリケーション・データベース表に追加する INSERT ステートメントの列リストに それぞれの列名を付加します。すべての属性が処理されると、この SQL ステートメントは次のようになります。

```
INSERT INTO customer (cust_key, cust_name, cust_status, cust_region)
```

C++ コネクターでは、動詞メソッドは現在のビジネス・オブジェクト上の BusinessObject::getAttrCount() を呼び出して、ビジネス・オブジェクト内の属性の数を判別します。それぞれの属性ごとにアプリケーション固有の情報を取得するには、次のステップがあります。

1. 続いて、動詞メソッドはビジネス・オブジェクト定義の全探索を行い、BusObjSpec::getAttribute() を呼び出して、各属性記述子を検索します。

   getAttribute() メソッドは、BOAttrType クラスのインスタンスへのポインターを戻します。それぞれの BOAttrType インスタンスは、ビジネス・オブジェクト定義内の属性の単一属性記述子を表します。
2. 属性記述子を介して、コネクターは、属性がキーか、または外部キーかなどの、属性プロパティーに関する情報を検索することができます。

それぞれの属性ごとに、メソッドは、BOAttrType クラスで定義されている getAppText() メソッドで、属性記述子から属性に関するアプリケーション固有の情報を抽出します。

図 64 で、動詞メソッドは、ビジネス・オブジェクトの全探索を行う際に、それぞれの属性ごとの列名を column 変数にコピーします。

```
for (i = 0; i < theObj.getAttrCount() - 1; i++)
   strcpy(column, theSpec->getAttribute(i)->getAppText());
```

図 64. 属性のアプリケーション固有情報の取得

図 64 に示す for ループは、次のタスクを実行します。

- ループ指標は 0 に初期化されます。

  この例で、宛先アプリケーションは、ソース・アプリケーションによって生成された同じキー値を使用します。このキー値は単純に、ビジネス・オブジェクト内の宛先アプリケーションに渡されます。宛先アプリケーションが独自のキーを生成した場合には、ビジネス・オブジェクトには通常、キー値は含まれず、キー属性は特殊な Ignore 値に設定されている可能性があります。

  Create 動詞メソッドが、キーを含んでいる、最初の属性を処理する場合、loop index 変数は 0 で開始します。ただし、アプリケーションがキーを生成した場合には、Create 動詞メソッドは、キーが含まれる属性を処理しません。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・オブジェクト定義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には存在しません。そのため、動詞メソッドはこの属性を処理する必要がありません。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。

  この指標の増分で、getAttribute(i) の呼び出し時に次の属性記述子が取得されます。

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。doVerbFor() メソッドは、このロケーション情報を取得すると、属性の処理を開始します。

動詞操作がビジネス・オブジェクト属性をループするとき、メソッドが特定の属性のみ処理することを確認する場合があります。表 66 に、ある属性を処理すべきかどうかを判別するために C++ コネクター・ライブラリーで提供されているメソッドの一部をリストします。

表 66. 属性処理を決定するためのクラスとメソッド

属性テスト	C++ コネクター・ライブラリー・クラスとメソッド
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ありません。	BOAttrType isObjectType()
ビジネス・オブジェクト・インスタンス内の属性の値は、Blank (長さ 0 のストリング) または Ignore (ヌル・ポインター) の特殊値ではありません。	BusinessObject getAttrValue(),, isIgnoreValue(),, isIgnore(),, isBlankValue(),, isBlank()
属性がプレースホルダー属性ではありません。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。	BOAttrType getAppText()

動詞操作では、表 66 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。

  BOAttrType::isObjectType() メソッドは、含まれているビジネス・オブジェクトを属性値が表さない ことを検査します。ビジネス・オブジェクトが含まれている属性の処理方法の詳細については、202 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 つの特殊なタイプ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 Blank または Ignore の特殊値以外の値に設定されているか。

  動詞操作は、それぞれ、isIgnoreValue() および isBlankValue() メソッドで、属性値を Ignore 値および Blank 値と比較することができます。 Ignore 値および Blank 値の詳細については、199 ページの『Blank 値および Ignore 値の処理』を参照してください。

図 65 のコード断片では、ある属性がメソッドによって処理される属性であることをサンプルの Create 動詞メソッドが判別する方法が示されます。メソッドは最初に、BusinessObject::getAttrValue() を呼び出して、現在のビジネス・オブジェクトから属性値を取得します。続いて、属性を処理するかどうかを判別するためにテストを行います。

```
for (i = 0; i< theObj.getAttrCount()-1; i++) {
   theAttr = theSpec->getAttribute(i);
   theAttrVal = theObj.getAttrValue(i);

   if (!theAttr->isObjectType() && strlen(theAttr->getAppText()) > 0)
   {
      // Use only columns that contain a valid value
      if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
           !(theObj.isBlankValue((char *)theAttrVal))) {

         // Get the column name from the AppSpecificInfo text
         strcpy(column, theSpec->getAttribute(i)->getAppText());
   }
}
```

図 65. 属性を処理するかどうかの判別

アプリケーション固有の情報を持ち、Ignore または Blank ではない 値が含まれている単一属性の場合、コネクターは、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から列名を検索し、その名前を SQL ステートメントに追加します。

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、検索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、この検索情報を使用して、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 Exist 動詞の動詞操作では、検索情報を保持するすべてのキー属性 (Retrieve または Exist) 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、この検索情報を使用して、エンティティーを検索します。

  注: Retrieve または RetrieveByContent の動詞操作では、検索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、検索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、この検索情報を使用して、削除対象のエンティティーを探し出します。

属性の値は、ビジネス・オブジェクト (BusinessObject インスタンス) 内の属性情報の一部です。表 67 に、ビジネス・オブジェクトから属性値を取得するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 67. 属性値の取得用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	getAttrCount(),, getAttrValue()

ビジネス・オブジェクト定義の場合と同様に、ビジネス・オブジェクト内のそれぞれの属性には、次の 2 つの方法のいずれかでアクセスすることができます。

- 属性名 — その順序位置を知っている場合には、getAttrName() メソッドで属性名を取得することができます。
- 整数指標 — 属性指標 (その順序位置) を取得するには、getAttrCount() でビジネス・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、それらの属性を一度に 1 つずつ、それぞれの指標値を getAttrValue() に渡しながらループして、属性値を取得することができます。

表 67 に示すように、BusinessObject クラスは、すべての有効なデータ型の属性値を取得するための単一メソッド getAttrValue() を提供します。ビジネス・オブジェクト定義内の属性のタイプは任意のサポートされているタイプでかまわないため、getAttrValue() の戻り値は void ポインターとして定義されます。ビジネス・オブジェクト定義内の属性のタイプを検査し、戻り値を変数に割り当てる前に、属性タイプに基づいて、void ポインターを文字ポインター、ビジネス・オブジェクト・ポインター、またはビジネス・オブジェクト配列にキャストしてください。

注: 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いずれでもない属性値は、文字ストリングへのポインターとして C++ コネクター・ライブラリー内に格納されます。属性の値がビジネス・オブジェクト、またはビジネス・オブジェクト配列でない 場合には、void ポインターを文字ポインターにキャストする必要があります。

処理する属性を確認後、doSimpleCreate() メソッドは (図 62、図 63、および図 65 を参照)、アプリケーション表内の列に挿入するデータ値を取得する必要があります。メソッドは、それぞれの属性を処理する際に、属性値を SQL ステートメントに追加します。属性値のリストを作成するため、動詞メソッドは 2 回目にビジネス・オブジェクト定義の属性の全探索を行います。それぞれの属性ごとに、動詞メソッドはビジネス・オブジェクト・インスタンスから属性値を取得します。ビジネス・オブジェクトのこの 2 回目の全探索で、動詞メソッドは、それぞれの属性のタイプと値をもう一度検査して、属性の処理を行うかどうかを判別します。

注: このコネクターはデータベース照会を作成するためにビジネス・オブジェクトの全探索を 2 度行いますが、アプリケーション API を使用してアプリケーションでの値を設定する場合には、API は、上記のようにビジネス・オブジェクトをループする必要はありません。

続いて、コネクターは、検索した列値を使用して、SQL ステートメントの作成を続行します。115 ページの図 38 に示されているサンプルのビジネス・オブジェクトを使用して、コネクターは、新規の行を customer という名前のアプリケーション・データベース表に追加する INSERT ステートメントの VALUES 文節にそれぞれの列値を付加します。

doSimpleCreate() メソッドがサンプルの Customer ビジネス・オブジェクトを次のデータで処理したとします。

CustomerId	87975
CustomerName	Trievers Inc.
CustomerStatus	3
CustomerRegion	NE

すべての属性が処理されると、この SQL ステートメントは次のようになります。

```
INSERT INTO customer (cust_key, cust_name, cust_status, cust_region)
   VALUES (87975, 'Trievers Inc.', 3, 'NE')
```

C++ コネクターでは、動詞メソッドは BusinessObject::getAttrValue() を呼び出して、ビジネス・オブジェクト・インスタンスからそれぞれの属性の値を検索します。動詞メソッドは、戻された属性値を文字ポインターにキャストして、INSERT ステートメントの VALUES 文節を生成しします。図 66 に、属性値にアクセスし、INSERT ステートメントの VALUES 文節に属性値を追加する、Create 動詞メソッドのコード断片を示します。

```
for (i = 0; i< theObj.getAttrCount()-1; i++) {
   theAttr = theSpec->getAttribute(i);
   theAttrVal = theObj.getAttrValue(i);

   // Process simple attributes
   if (!theAttr->isObjectType() && strlen (theAttr->getAppText()) > 0)
   {

      // Use columns that contain a valid value in
      // the business object
      if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
            !(theObj.isBlankValue((char *)theAttrVal))) {

         // Set the quote character for attributes that
         // are STRING type
         quote_str[0] = (theObj.getAttrType(i) ==
                     BOAttrType::STRING) ? '¥'' : ' ';

         // Build the value and add it to insertStatement
         sprintf(clause, "%s %s%s%s", firstLoop ? " " : ",",
                  quote_str, (char *)theAttrVal, quote_str);
         strcat(insertStatement, clause);
         firstLoop = 0;
      }
   }
}
```

図 66. C++ コネクター内の属性値へのアクセス

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 ステートメントまたは ODBC 呼び出しです。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には、doVerbFor() メソッドは、該当する結果状況 (BON_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。

doSimpleCreate() メソッドが SQL ステートメントを作成した場合、そのステートメントを実行する準備が整います。INSERT ステートメントが実行されるとき、アプリケーションは customer データベース表に新規の行を作成します。SQL ステートメントを実行するには、表アクセスを指定するアプリケーション API を使用しなければなりません。doSimpleCreate() 動詞メソッドは、標準の ODBC API を使用して、SQL ステートメントを実行します。SQL ステートメントを実行する API がアプリケーションにある場合には、アプリケーション API を使用します。図 67 に示すコード断片は、SQL ステートメントを終了し、ODBC 呼び出しを使用して SQL ステートメントを実行します。

```
// Finish the INSERT statement
strcat(insertStatement, ")");

// Allocate an ODBC statement
rc = SQLAllocStmt(hdbc, &hstmt);
// Execute the SQL statement
rc = SQLExecDirect(hstmt, (unsigned char *)insertStatement, SQL_NTS);
// Free the ODBC statement handle
```

図 67. C++ コネクター内の INSERT ステートメントの実行

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検索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、検索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 68 に、ビジネス・オブジェクトに属性値を保管するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 68. 属性値の保管用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	getAttrCount(),, setAttrValue()

ビジネス・オブジェクト内の属性には、その名前またはその指標 (その順序位置) によってアクセスすることができます。 getAttrCount() でビジネス・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、それらの属性を一度に 1 つずつ、それぞれの指標値を setAttrValue() に渡しながらループして、属性値を取得することができます。

表 68 に示すように、BusinessObject クラスは、すべての有効なデータ型の属性値を保管するための単一メソッド setAttrValue() を提供します。ビジネス・オブジェクト定義内の属性のタイプは任意のサポートされているタイプでかまわないため、setAttrValue() のパラメーター値は void ポインターとして定義されます。

動詞処理応答の送信

C++ 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、受信した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

次の各セクションでは、C++ コネクターが応答情報を提供する方法について追加的に説明します。コネクター応答の一般情報については、121 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 69 に示すように、C++ コネクター・ライブラリーには、doVerbFor() によって戻される可能性が高い結果状況値の定数が示されています。

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 69. C++ doVerbFor() の結果状況値

doVerbFor() 内の条件	C++ 結果状況
動詞操作が成功しました。	BON_SUCCESS
動詞操作が失敗しました。	BON_FAIL
アプリケーションが応答しません。	BON_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。	BON_VALCHANGE
要求された操作によって、同じキー値のレコードが複数検出されました。	BON_VALDUPES
コネクターが非キー値を使用して検索中に複数の一致レコードを見つけました。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。	BON_MULTIPLE_HITS
コネクターは、非キー値による検索で、一致レコードを検出できませんでした。	BON_FAIL_RETRIEVE_BY_CONTENT
要求されたビジネス・オブジェクトのエンティティーが、データベース内に存在していません。	BON_BO_DOES_NOT_EXIST

注: C++ コネクター・ライブラリーには、他のコネクター・メソッドで使用するときの追加の結果状況定数が提供されています。結果状況定数の完全リストについては、194 ページの表 69 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 70 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 70. 各動詞の戻り値

動詞	詳細情報
Create	94 ページの表 28
Retrieve	101 ページの表 29
RetrieveByContent	103 ページの表 30
Update	110 ページの表 32
Delete	112 ページの表 34
Exist	114 ページの表 35

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 BON_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、175 ページの『動詞を処理する前の接続の検証』を参照してください。
- 他のすべての結果状況値の場合、コネクター・フレームワークはその現行の状態で継続します。コネクター・フレームワークは、結果状況を統合ブローカーへの応答に組み込みます。結果状況値によっては、コネクター・フレームワークが応答ビジネス・オブジェクトを組み込むこともあります。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、doVerbFor() メソッドが終了するときのこのメソッド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・フレームワークは、空の戻り状況記述子を引き数として doVerbFor() に渡します。 doVerbFor() メソッドは、この戻り状況記述子をメッセージで更新することができます。コネクター・フレームワークは、doVerbFor() が終了するときに、この更新された戻り状況記述子にアクセスできます。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。

C++ コネクターでは、戻り状況記述子は ReturnStatusDescriptor オブジェクトです。表 71 に、この構造体が提供する状況情報を示します。

表 71. 戻り状況記述子内の情報

戻り状況記述子情報	説明	C++ accessor 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	getErrorMsg(),, seterrMsg()
状況値	エラー状態の原因の詳細を示す整数の状況値	getStatus(),, setStatus()

戻り状況記述子の情報は、次のいずれかの方法で入力されます。

- 明示的に — `doVerbFor()` メソッド内の動詞処理が成功したとき、メソッドは、実行を完了する前に、値をこの記述子に設定します。
- 暗黙的に — コネクター・フレームワークが結果状況値を戻り状況記述子の状況フィールドにコピーしたとき。

終了前に戻り状況記述子を充てんするサンプルのコードについては、175 ページの図 58 を参照してください。

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、次に示すように、コネクター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。
  - `BON_VALCHANGE`
  - `BON_MULTIPLE_HITS`

  `doVerbFor()` メソッドがこれらの結果状況値の 1 つを戻す場合には、その結果状況値が要求ビジネス・オブジェクトを応答情報で更新することを確認してください。
- 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、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()` メソッドが戻す値は、コネクター・フレームワークが InterChange Server に送信する内容に影響を与えます。値が `BON_VALCHANGE` または `BON_MULTIPLE_HITS` である場合、コネクター・フレームワークは変更されたビジネス・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。

例: フラット・ビジネス・オブジェクトの Create メソッド

図 68 の C++ コードのサンプルは、Open Database Connectivity (ODBC) API を使用して新規のレコードをアプリケーション・データベースに挿入する Create メソッドを示しています。ODBC インターフェースは、各種のデータベース・システムにアクセスするための標準 API です。

このコード・サンプルには、ビジネス・オブジェクトから情報を抽出する基本ロジックを示します。コネクターがビジネス・オブジェクト定義内のメタデータおよびビジネス・オブジェクト・インスタンスの内容を使用して、SQL INSERT ステートメントをどのように作成できるのかを示します。

コネクターは最初に、`BusinessObject::getSpecFor()` を呼び出して、Create メソッドへの引き数として渡される、ビジネス・オブジェクト・インスタンスのためのビジネス・オブジェクト定義へのポインターを検索します。コネクターは、

BusObjSpec::getAppText() 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からアプリケーション表の名前を検索し、SQL ステートメントの作成を開始します。特殊な Blank 値または Ignore 値以外の値である、ビジネス・オブジェクト・インスタンス内のそれぞれの属性値ごとに、コネクターは、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から列名を検索し、その名前を SQL ステートメントに追加します。

続いて、コネクターは、BusinessObject::getAttrValue() を呼び出して、ビジネス・オブジェクト・インスタンスからそれぞれの属性の値を検索します。SQL INSERT ステートメントが完了すると、メソッドは ODBC API SQLExecDirect() を呼び出して、ステートメントをサブミットします。一般に、Create メソッドはアプリケーションから新規のエンティティーのためのキーを取得し、ビジネス・オブジェクト内の InterChange Server にそのキーを戻し、BON_VALCHANGE を戻します。ただし、このメソッドは、ソース・アプリケーション内の値にキーを設定するため、単純に BON_SUCCESS を戻します。

```
int doSimpleCreate(BusinessObject &theObj)
{
   char                          table_name[64];
   char                          column[64];
   char                          columnList[256];
   char                          clause[256];
   char                          insertStatement[1024];
   char                          quote_str[2] = " ";
   int                           firstLoop = 1;
   int                           j;
   BusObjSpec                     *theSpec;
   void                          *theAttrVal;
   BOAttrType                    *theAttr;
   RETCODE                       rc; /* return code for ODBC functions */
   HSTMT                         hstmt; /* pointer to ODBC statement handle */

   // Retrieve pointer to the business object definition
   theSpec = theObj.getSpecFor();

   // Retrieve the table name from the AppSpecificInfo property
   // for the business object definition
   strcpy(table_name, theSpec->getAppText());
   // Begin building the SQL INSERT statement
   sprintf(insertStatement, "INSERT INTO %s (", table_name);

   // Build the list of column names for the INSERT statement
   // For each attribute, extract the column name from the
   // attribute AppSpecificInfo property
   for (j = 0; j < theObj.getAttrCount()-1; j++) {
      theAttr = theSpec->getAttribute(j);
      theAttrVal = theObj.getAttrValue(j);

      // Process non-child objects only
      if (!theAttr->isObjectType() &&
               strlen (theAttr->getAppText()) > 0) {
         // Use only columns that contain a valid value
         // in the Business Object
         if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
               !(theObj.isBlankValue((char *)theAttrVal))) {
            // Get the column name from the AppSpecificInfo text
            strcpy(column,
               theSpec->getAttribute(j)->getAppText());

            sprintf(columnList, "%s %s", firstLoop ? " " : ",",
               column);
            strcat(insertStatement, columnList);

            firstLoop = 0;
         }
      }
   }

   // Add the VALUES SQL keyword
   sprintf(clause, ") VALUES (");
   strcat(insertStatement, clause);

   // Build the values to be inserted
   // For each attribute, extract the value from the business object
   firstLoop = 1;
   for (j = 0; j < theObj.getAttrCount()-1; j++) {
      theAttr = theSpec->getAttribute(j);
      theAttrVal = theObj.getAttrValue(j);
```

図 68. Create メソッドの例 (1/2)

```
      // Process non-child objects only
      if (!theAttr->isObjectType() &&
            strlen (theAttr->getAppText()) > 0) {

         // Use columns that contain a valid value in
         // the business object
         if (!(theObj.isIgnoreValue((char *)theAttrVal)) &&
                  !(theObj.isBlankValue((char *)theAttrVal))) {

            // Set the quote character if this is a STRING attribute
            quote_str[0] = (theObj.getAttrType(j) ==
               BOAttrType::STRING) ? '¥'' : ' ';

            // Build the value and add it to insertStatement
            sprintf(clause, "%s %s%s%s", firstLoop ? " " : ",",
               quote_str, (char *)theAttrVal, quote_str);
            strcat(insertStatement, clause);
            firstLoop = 0;
         }
      }
   }

   // Finish the INSERT statement
   strcat(insertStatement, ")");
   // Allocate an ODBC statement
   rc = SQLAllocStmt(hdbc, &hstmt);
   // Execute the SQL statement
   rc = SQLExecDirect(hstmt, (unsigned char *)insertStatement,
      SQL_NTS);
   // Free the ODBC statement handle
   SQLFreeStmt(hstmt, SQL_DROP);

   return BON_SUCCESS;
}
```

図 68. Create メソッドの例 (2/2)

注: 図 68 のコード・サンプルは、メタデータ主導型のコネクターを設計する際の一般的な手法を示しています。ただし、この例の大半は ODBC ベースのコネクターに特定されています。ODBC (Open Database Connectivity) API が使用されていますが、その理由は、それがデータベースにアクセスするための標準の API であるからです。アプリケーション・データの変更をコネクターに可能にする API がアプリケーションに用意されている場合には、アプリケーション API を使用することをお勧めします。アプリケーション API を使用するとき、動詞操作のインプリメントは、上記の例で示すインプリメントとは異なる場合があります。

処理に関するその他の問題

このセクションでは、要求ビジネス・オブジェクトの処理方法について、追加的に説明します。

- 『Blank 値および Ignore 値の処理』
- 202 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 72 に示す特殊値のいずれかを使用できます。

表 72. 単純属性の特殊な属性値

特殊な属性値	意味
Blank	長さ 0 のストリング値
Ignore	コネクターが無視する値

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、マップ内だけで 使用します。これらのストリングは、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、C++ 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 73 に示すメソッドを使用できます。

表 73. 属性に Blank 値が含まれているどうかを判別するためのメソッド

BOAttrType メソッド	説明
isBlankValue(*value*)	指定された属性値が Blank 値に等しいかどうかを判別します。
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 値が含まれている場合、`doVerbFor()` メソッドは 表 75 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 74 に示すメソッドを使用できます。

表 74. 属性に Ignore 値が含まれているどうかを判別するためのメソッド

BOAttrType メソッド	説明
isIgnoreValue(*value*)	指定された属性値が Ignore 値に等しいかどうかを判別します。
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 76 で示すように属性を処理します。

表 75. 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検索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検索します。
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 76. 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検索し、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削除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。属性値を特殊な Ignore 値または Blank 値に設定するには、`setAttrValue()` メソッド (`BusinessObject` クラスで定義) を使用して、次に示すように、属性値を特殊な属性値定数に渡します。

Blank 定数	`BusinessObject::BlankValue`
Ignore 定数	`BusinessObject::IgnoreValue`

例えば、次の C++ コード断片は、すべての非キー属性を Ignore 値に設定します。

```
for (i = 0; i < theObj.getAttrCount()-1; i++)
   {
   if (!theAttr->isKey())
      {
      attrname = theObj.getAttrName(i);
      theObj.setAttrValue(attrname, BusinessObject::IgnoreValue,
         theObj.getAttrType(i));
      }
   }
```

別の例として、下記の C++ コード断片は、アプリケーション・データを検索し、アプリケーション・データベースで NULL 値を持つビジネス・オブジェクト属性を Blank 属性値に設定します。

```
// Fetch application data into appdata variable
// Process record
for (i = 0; i < theObj.getAttrCount()-1; i++)
   {
   if (!theSpec->getAttribute(i)->isObjectType())
      {
      if (strlen(appdata)==0)
         sprintf(attrValue, theObj.getBlankValue());
      theObj.setAttrValue(i, (void*)attrValue,theObj.getAttrType(i));
      }
   }
```

子ビジネス・オブジェクトへのアクセス: 117 ページの『階層ビジネス・オブジェクトの処理』で説明されているように、C++ コネクターは、表 77 に示されている C++ コネクター・ライブラリーのメソッドを使用して、子オブジェクトにアクセスします。

表 77. 子ビジネス・オブジェクトにアクセスするためのクラスとメソッド

C++ コネクター・ライブラリー・クラス	メソッド
BOAttrType	isObjectType(),, isMultipleCard() OBJECT 属性タイプ定数
BusinessObject	getAttrValue()
BusObjContainer	getObjectCount(),, getObject()

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 (属性タイプが OBJECT 属性タイプ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。 doVerbFor() が、ビジネス・オブジェクトである属性を見つけるとき、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索します。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、動詞操作は、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。
- 属性が複数のカーディナリティーを持つ場合、動詞操作は最初に、値を次のように戻す getAttrValue() メソッドを使用して、ビジネス・オブジェクト配列をアクセスする必要があります。
  - 属性が単一属性である場合には、getAttrValue() は void ポインターを値に戻します。
  - 属性がビジネス・オブジェクトである場合には、getAttrValue() は void ポインターをビジネス・オブジェクトに戻します。
  - 属性がビジネス・オブジェクトの配列である場合には、getAttrValue() は void ポインターをその配列が含まれている BusObjContainer オブジェクトに戻します。

getAttrValue() の戻り値は、データに使用する正しいタイプにキャストしなければなりません。例えば、ビジネス・オブジェクト配列の内容にアクセスするには、戻り値は、次のコード断片で示すように、BusObjContainer タイプにキャストしなければなりません。

```
theAttr = theSpec->getAttribute(i);
   if (theAttr->isObjectType()) {
      if(theAttr->isMultipleCard()) {
         // Multiple cardinality object so cast attribute value
         // to a BusObjContainer object
         BusObjContainer *busObjContnr =
            (BusObjContainer *) theObj.getAttrValue(i);
```

属性にビジネス・オブジェクト配列が含まれている場合には、doVerbFor() メソッドは getAttrValue() が戻した、キャストされた BusObjContainer オブジェクトを介して、この配列へのアクセスを取得します。

注: 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 (BusObjContainer) 内の子ビジネス・オブジェクトにアクセスするメソッド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。

ビジネス・オブジェクト配列内の個々のビジネス・オブジェクトにアクセスするには、次のステップを実行します。

1. BusObjContainer::getObjectCount() を呼び出して、配列内の子ビジネス・オブジェクトの数を取得します。
2. 動詞処理は、ビジネス・オブジェクト配列へのアクセスを繰り返すため、BusObjContainer::getObject(*指標*) メソッドを使用して、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することができます。ここで、*指標* は、配列エレメント指標です。このメソッドは、指定した位置にビジネス・オブジェクトがない場合には、ポインターを子ビジネス・オブジェクトまたは NULL に戻します。

図 69 に、子ビジネス・オブジェクトにアクセスするための C++ コードを示します。

```
for (int i=0; i < busObjContnr->getObjectCount(); i++) {
   BusinessObject *currBusObj = busObjContnr->getObject(i);
   status = doVerbMethod(*currBusObj);
}
```

図 69. C++ コネクター内の子ビジネス・オブジェクトへのアクセス

図 70 に、C++ サブメソッド do*Verb*Method() を示します。このサブメソッドは、子オブジェクトを処理するメイン動詞メソッドによって呼び出される可能性があります。 119 ページの図 43 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。

```
int GenBOHandler::doChildCreate(BusinessObject &theObj)
{
   int      i, k;
   int       status = BON_SUCCESS;

   for (i = 0; i < theObj.getAttrCount() -1; i++) {
      theAttr = theSpec->getAttribute(i);
      theAttrVal = theObj.getAttrValue(i);
      if (theAttr->isObjectType()) {
         if(theAttr->isMultipleCard()) {
            // Multiple cardinality object so cast attribute value
            // to a BusObjContainer object
            BusObjContainer *Cont = (BusObjContainer *) theAttrVal;
            if (theAttrVal != NULL) {
               for (k=0; k < Cont->getObjectCount(); k++) {
                  BusinessObject *curObj = Cont->getObject(k);
                  status = doCreate(*curObj);
                  if (status == BON_FAIL)
                     return status;
               }
            }
         }
         else {
            // Single cardinality object
            if (theAttrVal != NULL) {
               status = doCreate(*(BusinessObject *)theAttrVal);
               if (status == BON_FAIL)
                  return status;
            }
         }
      }
   }

   return status;
}
```

図 70. C++ サブメソッド内の子ビジネス・オブジェクトの処理

イベントのポーリング

C++ コネクターの場合、GenGlobals クラスは pollForEvents() メソッドを定義します。この仮想メソッドのインプリメンテーションは、コネクター・クラスの一環として行う必要があります。

注: イベント通知の概要については、25 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、123 ページの『第 5 章 イベント通知』を参照してください。

図 71 の C++ ベースの疑似コードには、pollForEvents() メソッドの基本ロジックのフローが示されています。このメソッドは最初に、サブスクリプション・マネージャーへのポインターを検索します。サブスクリプション・マネージャーは、コネクターによってサポートされる、ビジネス・オブジェクトへのサブスクリプションを管理します。

続いて、pollForEvents() メソッドは、イベント・ストアから一連のイベントを検索してから、それぞれのイベントごとに、メソッドはサブスクリプション・マネージャー・クラス・メソッド isSubscribed() を呼び出して、対応するビジネス・オブジェクトに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存在する場合には、メソッドはアプリケーションからデータを検索し、新規

のビジネス・オブジェクトを作成し、サブスクリプション・マネージャー・メソッド 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。

```
int ExampleGenGlob::pollForEvents()
{
   SubscriptionHandlerCPP *mySubHandler =
      GenGlobals::getTheSubHandler();
   int status = 0;
   get the events from the event list
   for events 0 to PollQuantity in eventlist {
      extract BOName, verb, and key from the event record
      if(mySubHandler->isSubscribed(BOName,BOverb) {
         BO = new BusinessObject(BOName)
         BO.setAttrValue(key)
         retrieve application data using doVerbFor()
         BO.setVerb(Retrieve)
         BO.doVerbFor()
         BO.setVerb(BOverb)
         status = mySubHandler->gotApplEvent(BusinessObject);
         archive event record with success or failure status
      }
      else {
         archive event record with unprocessed status      }
   return status;
}
```

図 71. C++ 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、138 ページの図 52 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 78 に、これらの基本ステップの要約を示します。

表 78. 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。	206 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。	206 ページの『イベント・ストアにアクセスする前の接続の検証』
3.	イベント・ストアから指定された数のイベント・レコードを検索し、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のマークを付け、処理を開始します。	207 ページの『イベント・レコードの検索』
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。	209 ページの『ビジネス・オブジェクト名、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。 イベントへのサブスクライバーがある場合	210 ページの『イベントへのサブスクリプションの検査』
	• アプリケーション・データを検索し、ビジネス・オブジェクトを作成します。	212 ページの『アプリケーション・データの検索』

表 78. 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・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。	214 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。	217 ページの『イベントの処理の完了』
	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。	210 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	218 ページの『イベントのアーカイブ』

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。) C++ コネクターは、SubscriptionHandlerCPP クラスによってカプセル化されている、サブスクリプション・ハンドラー を介して、サブスクリプション・マネージャーおよびコネクター・サブスクリプション・リストにアクセスすることができます。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。

表 79 に、サブスクリプション・ハンドラーへの参照を取得するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 79. サブスクリプション・ハンドラーの取得用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
GenGlobals	getTheSubHandler()

C++ コネクターの場合、pollForEvents() メソッドは最初に、GenGlobals::getTheSubHandler() を呼び出して、コネクターのため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。例えば、次のようになります。

```
SubscriptionHandlerCPP *mySub = GenGlobals::getTheSubHandler();
```

getTheSubHandler() メソッドは、SubscriptionHandlerCPP クラスのインスタンスである、サブスクリプション・マネージャーへのポインターを戻します。このサブスクリプション・ハンドラーを介して、コネクターは、そのサブスクリプション・マネージャーに照会して、統合ブローカーが特定のタイプのビジネス・オブジェクトを処理の対象とするどうかを調べることができます。

イベント・ストアにアクセスする前の接続の検証

コネクター・クラス内の init() 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、その最も一般的なタスクの 1 つは、アプリケーションと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と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、アプリケーションと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、176 ページの表 56 でのステップを実行して、失われた接続のコネクター・フレームワークを通知します。pollForEvents() メソッドは、BON_APPRESPONSETIMEOUT を戻して、アプリケーションとの接続の切断をコネクター・フレームワークに通知します。

図 72 には、アプリケーションとの接続の切断を処理するコードが含まれています。この例では、エラー・メッセージ 20018 が発行されて、コネクターからアプリケーションとの接続が切断したこと、およびアクションを実行する必要があることが管理者に通知されます。

```
int ExampleGlobals::pollForEvents()
{
   ...

   if (//application is not responding ) {
      // Lost connection to the application
      // Log an error message
      logMsg(generateMsg(20018, CxMsgFormat::XRD_FATAL, NULL, 0,
                   "MyConnector"));

      // Populate a ReturnStatusDescriptor object
      char errorMsg[512];
      sprintf(errorMsg, "Lost connection to application");
      rtnObj->seterrMsg(errorMsg);

      return BON_APPRESPONSETIMEOUT;      }

      ....
      // if connection is open, continue processing
      ...
}
```

図 72. pollForEvents() での接続の切断

イベント・レコードの検索

イベント通知をコネクター・フレームワークに送信するには、ポーリング・メソッドは、最初にイベント・ストアからイベント・レコードを検索する必要があります。C++ コネクターの場合、アプリケーション固有のインターフェースを使用して、イベント・レコードをイベント・ストアから検索する必要があります。

ポーリング・メソッドは、一度に 1 イベント・レコードずつ検索し、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・レコードを検索し、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検索してから、指定された数のイベント・レコードを検索し、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。

注: 多くの C++ コネクターは単一スレッドであるため、コネクター・フレームワークは、ポーリング・メソッドの稼働時には、要求ビジネス・オブジェクトを受け付けません。すなわち、ポーリング・メソッドがイベントを処理している間、要求処理はブロックされます。ポーリングごとに複数のイベントの処理をインプリメントする間はこのことを念頭においてください。単一スレッドおよびマルチスレッドの C++ コネクターについては、141 ページの『スレッド化の問題』を参照してください。

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。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うちに、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。これらの進行中イベントをリカバリーする方法の詳細については、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。

イベント・ストアからイベント・レコードを検索するには、C++ コネクターはアプリケーションが提供するすべての手法を使用する必要があります。一例として、表ベースのアプリケーションのインプリメントでは、コネクターのアプリケーション固有のコードがイベント・データを検索する元のイベント表を作成する場合があります。 図 73 のコード断片には、この種類のインプリメント用のコネクターが、ODBC API および ODBC SQL コマンドを使用して、イベント表からイベントを検索する方法が示されています。プログラムは最初に、イベント表からイベント・レコードを検索する、SQL SELECT ステートメント用のマクロを定義します。

```
#define GET_EVENT_QUERY                                    ¥
      "SELECT event_id, object_name, object_verb, object_key ¥
      FROM cw_events WHERE event_status = 0 ¥
      ORDER BY event_time"
```

この SELECT ステートメントは、状況が 0 (ポーリング可能) のイベント・レコードからイベント ID、ビジネス・オブジェクト名、動詞、およびキー・データを検索します。これらのイベント・レコードはタイム・スタンプによって順位付けられます。イベントの優先順位に基づいたイベント・レコードの検索の詳細については、142 ページの『イベントの優先順位によるイベントの処理』を参照してください。

ポーリング・メソッドは、データのメモリーを割り振り、特定のデータ列にメモリーをバインドし、一度に 1 イベント・レコードずつ、データを検索します。完全なサンプル・プログラムについては、219 ページの『基本 pollForEvents() メソッドの例』 を参照してください。

```
// Allocate a statement handle for the SQL statement
rc = SQLAllocStmt(gHdbc1, &hstmt1);

// Execute the SELECT query. Use the macro GET_EVENT_QUERY
rc = SQLExecDirect(hstmt1, GET_EVENT_QUERY, SQL_NTS);

// Allocate memory for event data
ev_id = new char[80];
obj_name = new char[80];
obj_verb = new char[80];
obj_key = new char[80];

query = new char[255];
key_value = new char[80];

// Bind all results set columns to event variables
rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, ev_id,    80, &cbValue);
rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, obj_name, 80, &cbValue);
rc = SQLBindCol(hstmt1, 3, SQL_C_CHAR, obj_verb, 80, &cbValue);
rc = SQLBindCol(hstmt1, 4, SQL_C_CHAR, obj_key,  80, &cbValue);

// Fetch the event data
while (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
      ev_id = '¥0';
      obj_name = '¥0';
      obj_verb = '¥0';
      obj_key = '¥0';

      rc = SQLFetch(hstmt1);
      if (rc == SQL_SUCCESS) {
         // Process the record
         if ((cp = strchr(ev_id, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_name, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_verb, ' ')) != NULL)
            *cp = NULL;
         if ((cp = strchr(obj_key, ' ')) != NULL)
            *cp = NULL;
      }
```

図 73. イベント検索

注: この例では、標準 ODBC API を使用して、イベント表からイベント・データを検索します。イベント表内のデータへのアクセスを提供する API がアプリケーションにある場合には、アプリケーション API を使用します。

ビジネス・オブジェクト名、動詞、およびキーの取得

コネクターは、イベントを検索した後で、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タイプのビジネス・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検索します。

C++ コネクターの場合、アプリケーション固有のインターフェースを使用して、この情報をイベント・ストア内のイベント・レコードから検索する必要があります。図 73 に、C++ コネクターがイベント・データを処理できる 1 つの方法を示しま

す。この例では、該当するイベント・データがコネクター変数に取り出されます。

イベント ID	ev_id
ビジネス・オブジェクト名	obj_name
動詞	obj_verb
オブジェクト・キー	obj_key

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。

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ある場合にだけ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 80 に、イベントへのサブスクリプションを検査するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 80. サブスクリプションを検査する方法

C++ コネクター・ライブラリー・クラス	メソッド
SubscriptionHandlerCPP	isSubscribed()

ポーリング・メソッドは、isSubscribed() が戻す値に基づいて、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検索します。	212 ページの『アプリケーション・データの検索』
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。	214 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』
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 (アーカイブがインプリメントされている場合)。	218 ページの『イベントのアーカイブ』

- 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 (アーカイブがインプリメントされている場合)。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、218 ページの『イベントのアーカイブ』を参照してください。
  - 「fail」(C++ コネクター用の BON_FAIL 結果状況) を戻して、サブスクリプションが現存しない、保留状態のイベントがあることを示します。

    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。

表 80 に示すように、isSubscribed() メソッドは、サブスクリプション・マネージャー SubscriptionHandlerCPP オブジェクトに用意されています。メソッドは、サブスクライバーが存在する場合には 1 を、サブスクライバーが存在しない場合には 0 を戻します。サブスクリプションを検査する C++ コネクターの一例を次に示します。

```
if (mySubHndlr->isSubscribed(obj_name, obj_verb) = TRUE) {
      // handle event
   }
else {
      // archive the event (if archiving is supported)
   }
```

アプリケーション・データの検索

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. アプリケーションからエンティティーのデータの完全セットを検索します。

   エンティティー・データの完全セットを検索するために、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・データベース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検索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・データを検索できないことがあります。削除処理の詳細については、143 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 80 に、アプリケーション・データベースからエンティティー・データを検索してビジネス・オブジェクトを取り込むために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 81. ビジネス・オブジェクト・データの検索用メソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	doVerbFor(),, getAttrCount(),, getAttrType(),, getAttrValue(),, setAttrValue(),, setVerb()

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検索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。delete イベントの処理の詳細については、143 ページの『削除イベントの処理』を参照してください。

C++ コネクターの場合、pollForEvents() 内からアプリケーション・データを検索する標準の方法は、コネクターのビジネス・オブジェクト・ハンドラーで Retrieve メソッドを使用することです。この手法では、pollForEvents() メソッドでデータ検索を再コーディングする必要はありません。

BusinessObject クラスの doVerbFor() メソッドを使用してアプリケーション・データを検索するには、一般的に次のステップを実行します。

1. アプリケーション・エンティティーに対応する、ビジネス・オブジェクト用の新規のビジネス・オブジェクト・インスタンスを作成します。
2. BusinessObject::setVerb() を呼び出して、ビジネス・オブジェクトの動詞を Retrieve に設定します。
3. イベント・レコードからアプリケーション・エンティティー用の 1 つ以上のキーを取得します。
4. BusinessObject::setAttrValue() を呼び出して、ビジネス・オブジェクト内のキー値を設定します。

   setAttrValue() メソッドは、属性の名前、値のストリング表記または属性の値へのポインター、および属性タイプを引き数として取得します。
5. このビジネス・オブジェクト上で BusinessObject::doVerbFor() を呼び出します。

   doVerbFor() メソッドの BusinessObject クラス・インプリメンテーションは、ビジネス・オブジェクト定義で指定された、ビジネス・オブジェクト・ハンドラーを呼び出して、動詞のアクションを実行します。doVerbFor() メソッドは、現在のビジネス・オブジェクト上で操作し、アプリケーション・エンティティーの現行値をビジネス・オブジェクトに埋め込みます。

図 74 の C++ コード断片で、この手法を示します。

```
// Create a new business object
pBusObj = new BusinessObject(obj_name);

// Set verb to Retrieve
pBusObj->setVerb("Retrieve");

// Extract value of key from key:value pair
if ((cp = strchr(obj_key, ':')) != NULL) {
   cp++;
   strcpy(key_value, cp);
   cp--;
   *cp = NULL;
}

// Find the key attribute in the business object and
// set it to the key value
for (i = 0; i < pObj->getAttrCount()-1; i++) {
   if (pBusObj->getSpecFor()->getAttribute(i)->isKey()) {
      pBusObj->setAttrValue(pBusObj->getAttrName(i),key_value,
         pBusObj->getAttrType(pBusObj->getAttrName(i)));
   }
}

// Call the business object handler doVerbFor()
if (pBusObj->doVerbFor() == BON_FAIL) {
   // Log error message if retrieve fails
   retcode = BON_FAIL;
}
```

図 74. アプリケーション・データの検索

図 74 のコード断片では、getAttrName() を使用して、キーである、それぞれの属性の名前を取得します。続いて、getAttrType() を呼び出して、キー属性のタイプを取得します。setAttrValue() メソッドは、属性値を変更する前に、新規の値が正しいデータ型を持つことを検証します。

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。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、126 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検索されたら、ポーリング・メソッドは次のタスクを実行します。

- 215 ページの『ビジネス・オブジェクト動詞の設定』
- 215 ページの『ビジネス・オブジェクトの送信』

表 80 に、ビジネス・オブジェクト動詞を設定してビジネス・オブジェクトを送信するために C++ コネクター・ライブラリーで提供されているメソッドをリストします。

表 82. 動詞設定とビジネス・オブジェクト送信のクラスおよびメソッド

C++ コネクター・ライブラリー・クラス	メソッド
BusinessObject	setVerb()
SubscriptionHandlerCPP	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。

C++ コネクターでは、doVerbFor() メソッドが戻す、取り込まれた BusinessObject オブジェクトには、Retrieve 動詞がまだ存在します。ポーリング・メソッドは、次のコード断片で示すように、setVerb() メソッドで、ビジネス・オブジェクトの動詞をそのオリジナル値に設定する必要があります。

```
// Set verb to action as indicated in the event record
pBusObj->setVerb(obj_verb);
```

上記のコード断片で、obj_verb は、図 73 に示すように、前にイベント・レコードから取得した動詞値です。

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、データを直列化して適切に永続化するために、イベント・オブジェクトに対して処理を実行します。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 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、戻されたエラーが適切に処理されたことを確認します。例えば、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。表 83 に、イベント・デリバリーが正常に実行されたかどうかに基づく可能なイベント状況値を示します。

表 83. イベント・デリバリー後の可能なイベント状況

イベント・デリバリーの状態	イベント状況
イベント・デリバリーが成功した場合	1 (126 ページの表 40 を参照)
イベント・デリバリーが失敗した場合	-2 (126 ページの表 40 を参照)

次のコード断片で、C++ コネクター用の gotApplEvent() への呼び出しを示します。

```
// Send business object to connector framework
if (( retcode = mySub->gotApplEvent(*pBusObj)) == BON_FAIL) {
   // Log an error message
   // Update event status to "error posting event"
   // Event remains in event table and is not archived
}

// Update event status to "event successfully posted"
```

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 BON_SUCCESS を戻し、デリバリーが失敗した場合には非ゼロ値 (BON_FAIL など) を戻します。

注: この呼び出しが正常に実行されなかった 場合は、イベントをアーカイブする必要があります。

イベントの処理の完了

表 84 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 84. イベントの処理のステップ

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・データを検索し、イベントを表すビジネス・オブジェクトを作成した。	212 ページの『アプリケーション・データの検索』
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。	214 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・データを検索し、コネクター・フレームワークに完全な階層ビジネス・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、階層ビジネス・オブジェクト全体を検索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』
- 218 ページの『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 84 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「正常」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、218 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、インプリメント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 85 に、イベント処理時に発生する可能性があるエラーに基づいた、可能なイベント状況値を示します。

表 85. 失敗したイベント処理後の可能なイベント状況

イベント・デリバリーの状態	イベント状況
イベントの処理中にエラーが発生した場合	-1 (126 ページの表 40 を参照)
イベント・デリバリーが失敗した場合	-2 (126 ページの表 40 を参照)

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。215 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。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。

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・ストアに移動することによる、アーカイブ・レコードの作成で構成されます。イベント・レコードをアーカイブ・ストアにアーカイブするには、C++ コネクターはアプリケーションが提供するすべての手法を使用する必要があります。通常、コネクターは、ODBC API および ODBC SQL コマンドなどの、イベント・ストア内のイベント・レコードにアクセスするために使用するメソッドをすべて使用します。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。

注: アーカイブの概要については、139 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、アーカイブがインプリメントされていることを確認します。詳細については、140 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. 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。
3. アーカイブ・レコードのイベント状況を更新して、イベントをアーカイブする理由を反映させます。

表 86 に、アーカイブ・レコードが通常持っているイベント状況値を示します。

表 86. アーカイブ・レコード内のイベント状況値

イベント状況値	説明
1	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りました。詳細については、217 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。
2	イベントは検出されましたが、このイベントに対するサブスクリプションはなかったため、イベントはコネクター・フレームワークには送られませんでした。統合ブローカーにも送られません。詳細については、210 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。
-1	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、218 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。

4. イベント・ストアからイベント・レコードを削除します。

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行われる場合、戻りコードは、`BON_SUCCESS` 結果状況定数で示される、「success」です。
- アーカイブがエラー条件 (アンサブスクライブされたイベント、またはイベント処理中のエラーなど) による場合には、ポーリング・メソッドは、`BON_FAIL` 結果状況定数で示される、「fail」状況を戻す必要があります。

基本 pollForEvents() メソッドの例

このセクションでは、C++ コネクターの `pollForEvents()` メソッドでの基本ロジックのインプリメントについて説明します。図 75 のコード・サンプルは、ODBC API を使用してアプリケーション・データベースと通信する C++ コネクターのイベント処理を示しています。この例で、コネクターはイベントを一度に 1 つずつ検索して処理します。

注: このコードは、`pollForEvents()` メソッドが呼び出される前に、コネクターの `init()` メソッドが ODBC インターフェースを初期化していると想定します。

`pollForEvents()` メソッドでの最初のステップとして、ユーティリティー機能は、2 つの ODBC 接続用のハンドルをアプリケーション・データベースに割り振ります。コネクターは、これらの接続ハンドルを使用してデータベースと対話します。

`pollForEvents()` メソッドは、`GenGlobals::getTheSubHandler()` を使用して、サブスクリプション・マネージャーへのポインターを取得します。続いて、ODBC `SQLExecDirect()` インターフェースを使用して SQL SELECT ステートメントを実行して、戻されるデータを判別します。SQL ステートメントが戻すとき、ポーリング・メソッドは、データのメモリーを割り振り、特定のデータ列にメモリーを割り当て、一度に 1 イベント・レコードずつ、割り振ったメモリーにデータを取り出します。

それぞれのイベントごとに、コネクターはサブスクリプション・マネージャー・メソッド isSubscribed() を呼び出して、この特定のビジネス・オブジェクトおよび動詞にサブスクライバーが存在するかどうかを判別します。サブスクライバーが存在する場合には、コネクターは、新規のビジネス・オブジェクトを作成し、キーを設定し、ビジネス・オブジェクトの動詞を Retrieve に設定し、ビジネス・オブジェクトの doVerbFor() メソッドを呼び出して、アプリケーション・エンティティー用の完全データ・セットを検索します。doVerbFor() が成功を戻した場合、ポーリング・メソッドは、ビジネス・オブジェクト内の該当する動詞を設定し、メソッド gotApplEvent() を使用して、ビジネス・オブジェクトを InterChange Server に送信します。

ビジネス・オブジェクトが送信されると、ポーリング・メソッドは SQL ステートメントを実行して、イベントをイベント表から取り外します。このアクションによって、削除トリガーはイベントをアーカイブ表に格納します。

```
// The event table is named cw_events.
#define GET_EVENT_QUERY                                   ¥
      "SELECT event_id, object_name, object_verb, object_key ¥
      FROM cw_events WHERE status = 0 AND priority = 1 ¥
      ORDER BY event_time"

#define REMOVE_EVENT_QUERY                                   ¥
      "DELETE FROM cw_events WHERE event_id = '%s'";

int ExampleGlobals::pollForEvents()
{
SubscriptionHandlerCPP  *mySub;
BusinessObject          *pObj = NULL;
char                    *ev_id = 0;
char                    *obj_name = 0;
char                    *obj_verb = 0;
char                    *obj_key = 0;
char                    *query = 0;
char                    *key_value = 0;
char                    *cp;
int                     i;
RETCODE                 rc;
HSTMT                   hstmt1 = 0;
HSTMT                   hstmt2 = 0;
SDWORD                  cbValue = SQL_NO_TOTAL;
int                     retcode = BON_SUCCESS;

   // Private function for establishing and checking the two
   // database connections that the poll function needs
   checkOdbcConnections();

   // Set up the subscription manager
   mySub = GenGlobals::getTheSubHandler();

   // Allocate a statement handle for the SQL statement
   rc = SQLAllocStmt(gHdbc1, &hstmt1);
   if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      // handle odbc errors
      return (BON_APPRESPONSETIMEOUT);
   }

   // Execute the event SQL SELECT query to determine
   // what data will be returned.
   // Use the macro GET_EVENT_QUERY
   rc = SQLExecDirect(hstmt1, GET_EVENT_QUERY, SQL_NTS);
   if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      // handle odbc errors
      rc = SQLFreeStmt(hstmt1, SQL_CLOSE);
      return (BON_APPRESPONSETIMEOUT);
   }

   // Allocate memory for event table columns
   ev_id = new char[80];
   obj_name = new char[80];
   obj_verb = new char[80];
   obj_key = new char[80];

   query = new char[255];
   key_value = new char[80];
```

図 75. pollForEvents() の基本ロジックのインプリメンテーション (1/4)

```
// Bind all results set columns to variables
rc = SQLBindCol(hstmt1, 1, SQL_C_CHAR, ev_id,    80, &cbValue);
rc = SQLBindCol(hstmt1, 2, SQL_C_CHAR, obj_name, 80, &cbValue);
rc = SQLBindCol(hstmt1, 3, SQL_C_CHAR, obj_verb, 80, &cbValue);
rc = SQLBindCol(hstmt1, 4, SQL_C_CHAR, obj_key,  80, &cbValue);

// Fetch the results
while (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
   ev_id = '¥0';
   obj_name = '¥0';
   obj_verb = '¥0';
   obj_key = '¥0';

   rc = SQLFetch(hstmt1);
   if (rc == SQL_SUCCESS || rc != SQL_SUCCESS_WITH_INFO) {
      // Process the record
      // Trim off rest of string at the first
      // space character found
      if ((cp = strchr(ev_id, ' ')) != NULL)
         *cp = NULL;
      if ((cp = strchr(obj_name, ' ')) != NULL)
         *cp = NULL;
      if ((cp = strchr(obj_verb, ' ')) != NULL)
         *cp = NULL;
       if ((cp = strchr(obj_key, ' ')) != NULL)
         *cp = NULL;

      // Determine whether there are subscribers to the event
      if (mySub->isSubscribed(obj_name, obj_verb) != TRUE) {
         // log message
         // delete event from event table
         // add event to archive table
         continue;
      }

      // Prepare to retrieve data into the business object
      pObj = new BusinessObject(obj_name);
      pObj->setVerb("Retrieve");

      // Get key:value pair
      if ((cp = strchr(obj_key, ':')) != NULL) {
         cp++;
         strcpy(key_value, cp);
         cp--;
         *cp = NULL;
      }

      // Find the first key in the object and set it
      for (i = 0; i < pObj->getAttrCount()-1; i++) {
         if (pObj->getSpecFor()->getAttribute(i)->isKey()) {
               pObj->setAttrValue(pObj->getAttrName(i),
               key_value,
               pObj->getAttrType(pObj->getAttrName(i)));
               break;
         }
      }
```

図 75. *pollForEvents()* の基本ロジックのインプリメンテーション *(2/4)*

```
         // Call the business object handler doVerbFor()
         // to retrieve application data
         if (pObj->doVerbFor() == BON_FAIL) {
            // Log error message if retrieve fails
            // Handle retrieve errors
            retcode = BON_FAIL;
            break;
         }

         // Call gotApplEvent() to send the business object
         // with the info
         pObj->setVerb(obj_verb);
         if ((mySub->gotApplEvent(*pBusObj)) == BON_FAIL) {
               // Log error message
            retcode = BON_FAIL;
            break;
         }

         // Allocate statement handle for the SQL
         // statement on the second connection
         rc = SQLAllocStmt(gHdbc2, &hstmt2);
         if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
            // Handle ODBC errors
            hstmt2 = (HSTMT)0;
            retcode = BON_APPRESPONSETIMEOUT;
            break;
         }

         // Remove the event from the event table
         // This will execute a trigger that will archive
         // the event in the archive table
         // Use the REMOVE_EVENT_QUERY macro.
         sprintf(query, REMOVE_EVENT_QUERY, ev_id);
         rc = SQLExecDirect(hstmt2, query, SQL_NTS);
         if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
            // Handle odbc errors
            retcode = BON_APPRESPONSETIMEOUT;
            break;

         } else {
               // Handle odbc errors
               retcode = BON_APPRESPONSETIMEOUT;
               break;
         }
      }

   // Clean up and free resources associated with
   // the statement handles
   if (hstmt1) {
      rc = SQLFreeStmt(hstmt1, SQL_DROP);
   }
   if (hstmt2) {
      rc = SQLFreeStmt(hstmt2, SQL_DROP);
   }
```

図 75. pollForEvents() の基本ロジックのインプリメンテーション (3/4)

```
if (ev_id) {
      delete ev_id;
   }
   if (obj_name) {
      delete obj_name;
   }
   if (obj_verb) {
      delete obj_verb;
   }
   if (query) {
      delete query;
   }
   if (key_value) {
      delete key_value;
   }

   return (retcode);
}
```

図 75. pollForEvents() の基本ロジックのインプリメンテーション (4/4)

コネクターのシャットダウン

C++ コネクター・ライブラリーでは、`terminate()` メソッドは `GenGlobals` クラスに定義されています。 `terminate()` で使用される一般的な戻りコードは、`BON_SUCCESS` および `BON_FAIL` です。

注: C++ コネクター用の `terminate()` メソッドが割り振り済みメモリーを解放し、アプリケーションとの接続をクローズすることは重要です。

図 76 に、C++ コネクター用のサンプルの `terminate()` メソッドを示します。

```
int ExampleGenGlob::terminate()
{
   // log off application and
   // release memory and other resources
   ...

   traceWrite(Tracing::LEVEL3, "terminate() completed.", 0);
   return BON_SUCCESS;
}
```

図 76. C++ terminate() メソッド

エラーと状況の処理

このセクションでは、C++ コネクター・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説明します。

- 225 ページの『C++ 戻りコード』
- 226 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、 153 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。

C++ 戻りコード

C++ コネクター・ライブラリーでは、BusObjStatus.h ファイル内の結果状況定数が C++ 戻りコードを定義します。表 87 に、C++ 結果状況定数をリストします。

表 87. C++ 戻りコード

結果状況定数	説明
BON_SUCCESS	操作が成功しました。
BON_FAIL	操作は失敗しました。
BON_APPRESPONSETIMEOUT	アプリケーションが応答しません。
BON_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。
BON_MULTIPLE_HITS	コネクターが非キー値を使用して検索中に複数の一致レコードを見つけました。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。
BON_FAIL_RETRIEVE_BY_CONTENT	コネクターは、非キー値による検索で、一致レコードを検出できませんでした。
BON_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。
BON_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。
BON_VALDUPES	アプリケーション・データベースに同一のキー値を持つ複数のレコードがあります。
BON_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。
BON_NO_SUBSCRIPTION_FOUND	イベントについてサブスクリプションは存在しません。

結果状況定数は、表 88 に示すように、C++ 仮想メソッドの多くのユーザー・インプリメンテーションで使用するために用意されています。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、コネクターが変更値を持つビジネス・オブジェクトを送信中であることを統合ブローカーに通知します。

表 88. C++ 仮想メソッドの結果状況値

仮想メソッド	可能な結果状況コード
init()	BON_SUCCESS、BON_FAIL、BON_UNABLETOLOGIN
doVerbFor()	BON_SUCCESS、BON_FAIL、BON_APPRESPONSETIMEOUT、BON_VALCHANGE、BON_VALDUPES、BON_MULTIPLE_HITS、BON_FAIL_RETRIEVE_BY_CONTENT、BON_BO_DOES_NOT_EXIST
gotApplEvent()	BON_SUCCESS、BON_FAIL、BON_CONNECTOR_NOT_ACTIVE、BON_NO_SUBSCRIPTION_FOUND
pollForEvents()	BON_SUCCESS、BON_FAIL、BON_APPRESPONSETIMEOUT
terminate()	BON_SUCCESS、BON_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 BON_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、この結果状況を受信すると、BON_APPRESPONSETIMEOUT 状況を戻り状況記述子にコピーし、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、BON_FAIL 結果状況定数を受信するときコネクターの実行を停止しません。

戻り状況記述子

要求処理中に、コネクター・フレームワークは、戻り状況記述子という空の構造体を、ビジネス・オブジェクト・ハンドラーの doVerbFor() メソッドに送信します。doVerbFor() が動詞処理を (正常処理または正常以外の処理として) 完了すると、コネクター・フレームワークは、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。

> **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て、サービス呼び出し要求の状況を取得できます。したがって、doVerbFor() は、戻り状況記述子内でメッセージと状況コードを設定することにより、動詞処理についての状況情報をコラボレーションに提供できます。

コネクター・フレームワークは、doVerbFor() が戻り状況記述子の状況フィールドに戻す結果状況を自動的にコピーします。したがって、doVerbFor() メソッドは、戻り状況記述子内にメッセージを設定できますが、状況の設定は行いません。これは、コネクター・フレームワークが結果状況をこの状況フィールドにコピーするときに、この状況が上書きされるためです。戻り状況記述子および doVerbFor() メソッドの詳細については、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。

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には、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 製品で提供されてい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. WebSphere MQ をコネクター・コンポーネント間でのメッセージングに使用する場合は、そのコネクター用のメッセージ・キューを追加します。
3. コネクターの始動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。

この章には、IBM WebSphere Business Integration システムに新規コネクターを追加する方法について記載されています。この章は、次のセクションから構成されています。

- 『コネクターの命名』
- 228 ページの『コネクターのコンパイル』
- 230 ページの『コネクター定義の作成』
- 233 ページの『初期構成ファイルの作成』
- 234 ページの『新規コネクターの始動』

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 89 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 89. 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*

© Copyright IBM Corp. 1997, 2004

表 89. 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: 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*Globals.cpp
コネクター・ライブラリー	*connDir*¥*connName*.dll 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およびリンクの方法については、この章で説明します。

このセクションの内容は次のとおりです。

- 『C++ コネクターのコンパイルとリンク』
- 230 ページの『デバッグ・バージョンの C++ コネクターの実行』

C++ コネクターのコンパイルとリンク

コネクターのアプリケーション固有コンポーネントをビルドするには、コネクターのヘッダー・ファイル (その他の必要なヘッダー・ファイルを含む) をインクルードし、ソース・ファイルをコンパイルして、`CwConnector.lib` をリンクすることにより、コネクターの動的にロード可能なライブラリー (DLL) を作成する必要があります。

重要:

1. 前のリリースの IBM WebSphere InterChange Server および WebSphere Business Integration Adapters では、C++ コネクターで使用するために Cayenne ライブラリーを提供していました。しかし、このリリース (WebSphere Business Integration Adapter Framework のバージョン 2.4) では、Cayenne ライブラリーが

標準テンプレート・ライブラリー (STL) に置き換えられています。このため、既存のカスタム C++ コネクターを再コンパイルし、新しい STL を使用するようにする必要があります。

2. Windows システムでは、CwConnector.lib ファイルが C++ Connector Development Kit (CDK) の一部として提供されています。CDK は Windows システムでのみ サポートされます。このため、C++ コネクターの作成は Windows システムでのみ サポートされます。UNIX ベースのシステムでは、C++ コネクターをコンパイルできません が、既存の C++ コネクターを実行することはできます。

Windows システムでは、MicroSoft Visual C++ 6.0 プログラミング環境を使用してコネクターをビルドしてください。また、以下の指示にも従ってください。

1. システムの PATH 変数に C++ コネクター・ライブラリー (CwConnector.dll) を含めること。これは製品ディレクトリーの bin サブディレクトリーにあります。
2. C/C++ 下 の「プロジェクト設定値 (Project Settings)」ウィンドウで、プロジェクト用のプリプロセッサー定義に CDKIMPORT を追加します。
3. C/C++ の「プロジェクト設定値 (Project Settings)」で、「プリプロセッサー (Preprocessor)」カテゴリーの下にある追加のインクルード・ディレクトリーに、以下を追加します。

   ```
   ..¥..¥generic_include
   ```

   重要: Cayenne ライブラリーが STL で置き換えられたため、C++ ファイルに cayenne_include ディレクトリーを含める必要はなくなりました。また、C++ コネクターで Cayenne クラスやメソッドを使用することはできなくなりました。
4. コネクター DLL 用「プロパティー」ウィンドウの「バージョン」タブで、表示された情報を定義します。以下のステップを実行します。
   a. ConnectorVersion.h という名前のファイルを作成して、コネクターに関する定数 (製品名や製品バージョンなど) を定義します。このファイルのサンプルは、製品ディレクトリーの以下のサブディレクトリー内にあります。

      ```
      DevelopmentKits¥cdk¥samples¥sampleconnector¥include
      ```

      注: このサンプル ConnectorVersion.h ファイルには、製品名と製品バージョンの値が用意されています。これらのサンプル値は必ず、ご使用のコネクターに合った値に変更してください。DLL のバージョンをチェックするには、その DLL を右マウス・ボタンでクリックし、「プロパティー」>「バージョン」タブを選択します。正しいバージョンがここに表示されるはずです。
   b. プロジェクト・ファイルに、次のファイルが追加されていることを確認します。

      ```
      DevelopmentKits¥cdk¥ConnectorVersion.rc
      ```
   c. 「追加のリソース・インクルード (Additional Resources Include)」セクションに、include ディレクトリーがあることを確認します。
      - ..¥..¥generic_include
      - ご使用のコネクター用の include ディレクトリー

「バージョン」ウィンドウでは、CDK に付属している ConnectorVersion.rc ファイルと generic_include¥CxResourceVersion.h ファイルを使用します。ご使用のコネクターの ConnectorVersion.h ファイルを定義する必要があります。

5. 「プロジェクト設定のリンク (Project Setting Link)」タブで、次のようにして適切なバージョンの C++ コネクター・ライブラリー (CwConnector.lib) をプロジェクトに追加します。
   - デバッグ・バージョンのコネクターを作成する場合は、以下を追加します。

     ProductDir¥DevelopmentKits¥cdk¥lib¥Debug¥CwConnector.lib
   - リリース・バージョンのコネクターを作成する場合は、以下を追加します。

     ProductDir¥DevelopmentKits¥cdk¥lib¥Release¥CwConnector.lib
6. コネクターをコンパイルしてリンクします。
7. C++ コネクターのライブラリー・ファイルを作成します。これは、動的にロード可能なライブラリー (DLL) です。

   コネクター DLL ファイルに対して推奨する命名規則は、コネクター名と一致させることです (227 ページの表 89)。詳細については、227 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 MyCPP をもつ C++ コネクターの場合、その DLL ファイルの名前は次のようになります。

   MyCPP.dll

デバッグ・バージョンの C++ コネクターの実行

MicroSoft Visual C++ 6.0 プログラミング環境を使用して、デバッグ・バージョンの C++ コネクターを実行します。WebSphere Business Integration Adapters 製品が、*ProductDir* が表すディレクトリーにインストールされていると想定した場合、「プロジェクト設定値」でデバッグ・セッション用の実行可能ファイルを、以下のように設定します。

ProductDir¥bin¥java.exe

プログラム実引き数のデバッグ・パラメーターを、以下のように設定します。

```
-Duser.home=ProductDir
-classpath ProductDir¥lib¥crossworlds.jar;ProductDir¥lib¥rt.jar;
ProductDir¥lib¥mq.jar AppEndWrapper -ddllName -nconnectorName
-sICSinstanceName
```

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には、*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 製品に提供されてい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構成プロパティーを設定するだけで、そのコネクターをさせることができます。ただし、IBM WebSphere Business Integration システムから開発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する必要があります。

- コネクター定義を作成して、リポジトリー内でそのコネクターを定義する。

- 初期構成ファイルを作成して、コネクター構成の際のユーザーの支援をする (オプション)。

コネクターの定義

WebSphere Business Integration システム内のコネクターを定義するには、コネクター定義 を作成します。このコネクター定義には、リポジトリーにコネクターを定義するための以下の情報をインクルードします。

- コネクター定義の名前
- サポートされるビジネス・オブジェクトと関連したマップ
- 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

ご使用の統合ブローカーが InterChange Server である場合、リポジトリーは、InterChange Server が、WebSphere Business Integration システム内のコンポーネントについての情報を取得するために通信するデータベースです。このリポジトリー内にコネクター定義が常駐します。これらのコネクター定義には、コネクター・コントローラーおよびクライアント・コネクター・フレームワークで必要な、標準の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組み込まれます。コネクターをローカルで構成するためのローカル構成ファイルを用意することもできます。ローカル構成ファイルが存在する場合、その構成ファイルが、InterChange Server リポジトリー内の情報よりも優先されます。

InterChange Server リポジトリー内のコネクター定義を、System Manager ツール内から Connector Configurator を使用して更新してください。ロケール構成ファイルは、スタンドアロン・バージョンの Connector Configurator を使用して更新することができます。このツールは、製品ディレクトリーの `bin` サブディレクトリーにあります。

WebSphere MQ Integrator Broker

ご使用の統合ブローカーが WebSphere MQ Integrator Broker である場合、リポジトリーは、コネクター・フレームワークが、WebSphere Business Integration システムのコンポーネントについての情報を取得するために使用するファイルのディレクトリーです。このリポジトリー内に、システムの各アダプターのコネクター定義が常駐します。

ローカル・リポジトリー内のコネクター定義は、Connector Configurator を使用して更新します。このツールは、製品ディレクトリーの `bin` サブディレクトリーにあります。

Connector Configurator の使用方法の詳細については、 373 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

コネクター定義名によって、WebSphere Business Integration システム内のコネクターが一意的に識別されます。コネクター定義名は、規則により通常は次の形式にします。

```
connNameConnector
```

ここで、*connName* はコネクター名です (227 ページの表 89 を参照してください)。コネクター名の詳細については、 227 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、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の以下の情報を指定することが必要です。

- ビジネス・オブジェクト定義

  コネクターが統合ブローカーとの間で送受信できる各ビジネス・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 には、「サポートされているビジネス・オブジェクト」タブが用意されており、このタブで、コネクターのサポートされているビジネス・オブジェクトを入力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。

- 関連マップ

WebSphere InterChange Server

統合ブローカーとして InterChange Server と通信するコネクターのコネクター定義の場合にのみ、コネクターに関連したマップをインクルードします。関連マップとは、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です。

Connector Configurator には、「関連マップ」タブが用意されており、このタブに、コネクターの関連マップを入力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設定するには、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。

- ご使用のコネクターで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宜値を割り当てます。

Connector Configurator には、コネクター構成プロパティーを指定するための 2 つのタブ、「標準のプロパティー」および「コネクター固有のプロパティー」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

事前定義のアダプターには、規則により、ユーザーが初めて Connector Configurator を使用してアダプターを構成するときに使用できる初期構成ファイルが用意されています。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。

CN_*connName*.txt

ここで、*connName* はコネクター名です (227 ページの表 89 を参照してください)。コネクター名の詳細については、 227 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次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は、各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。

ユーザーが開発したコネクターを構成するには、新規のコネクター用の初期構成ファイルを用意します。ユーザーはおそらくコネクター開発の一部として、標準の構成プロパティーの設定値だけでなく、コネクター固有の構成プロパティーの定義も行っているはずです。このコネクター構成情報がリポジトリー内にあるはずですが、コネクターを他の環境に移動すると、このリポジトリーへのアクセスは失われます。したがって、リリース済みのコネクターの一部である初期構成ファイルを作成することが必要になります。

この初期構成ファイルを作成するには、ご使用のコネクター用の Connector Configurator を立ち上げ、その構成を次のファイルに保管します。

ProductDir¥repository¥*connName*¥CN_*connName*.txt

注: これらのステップでは、開発の過程において、ご使用のコネクター用のコネクター構成ファイル (.cfg) を構成済みであることを前提としています。先行するステップでは、このコネクター構成情報を、リリース済みコネクターの一部としてインクルードされる、個別のファイルに保管するだけでした。

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 90 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 90. コネクター用の始動スクリプト

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. 先頭の引き数はコネクター名であり、この名前は以下のものを識別します。
   - 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、コネクターが実行される統合ブローカー・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、始動スクリプトで、ご使用のコネクターが実行される ICS インスタンスの名前を指定します。Windows システムでは、始動スクリプトのコネクター・ショートカットのそれぞれに、この ICS インスタンス名 (インストール・プロセスで指定されたもの) が表示されます。

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、始動スクリプトは、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指定します。Windows システムでは、始動スクリプトのコネクター・ショートカットのそれぞれに、このインスタンス名 (インストール・プロセスで指定されたもの) が表示されます。

3. オプションの追加の始動パラメーターをコマンド行に指定し、コネクター・ランタイムに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、リポジトリーからそのビジネス・オブジェクトを取得するためには、コネクターを始動する前に、InterChange Server が稼働していなければなりません。

開発したコネクターを始動する前に、始動スクリプトが新規コネクターをサポートすること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、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、 227 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。

   C++ コネクターのライブラリー・ファイルは DLL です。この DLL 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、228 ページの『C++ コネクターのコンパイルとリンク』を参照してください。

始動スクリプトの作成

234 ページの表 90 に示すように、コネクターでは、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ため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 77 に、Windows システムでコネクターを始動する手順を示します。

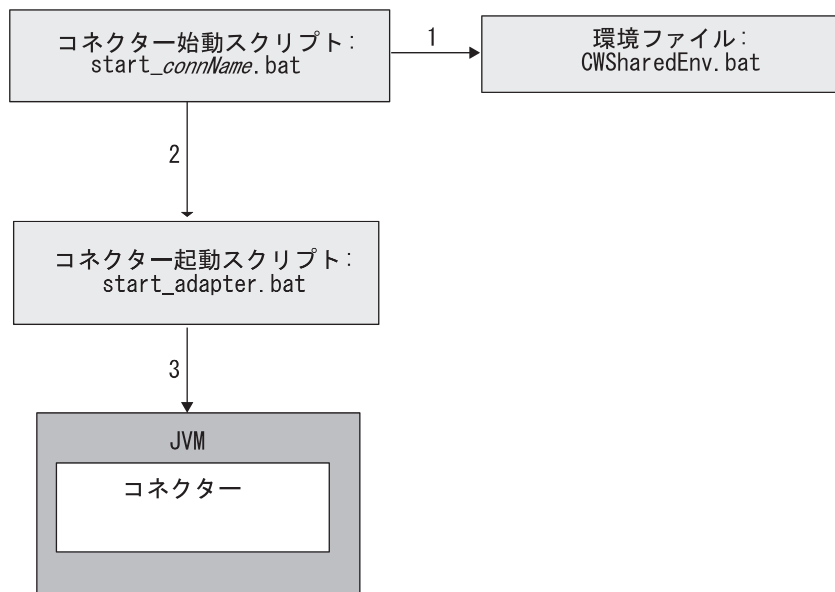

図 77. Windows システムでのコネクターの始動

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。

重要: 前のリリースでは、C++ コネクターに、start_connector.bat と呼ばれる特別な始動スクリプトが必要でした。このリリース (IBM WebSphere InterChange Server 4.2.2 および WebSphere Business Integration Adapters バー

ジョン 2.4) では、Java コネクターと C++ コネクターが同じ始動スクリプト start_*connName*.bat を使用できます。C++ コネクターを新規に開発する場合は、start_*connName*.bat 始動スクリプトを使用する必要があります。ただし、start_connector.bat スクリプトも後方互換性のため引き続きサポートされます。

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 C++ 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、C++ コネクターのコネクター名が MyCPP である場合、始動スクリプト名は start_MyCPP.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 78. 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 78. 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 79 に示されている標準構文になります。*connName* はコネクターの名前、*ICSinstance* は InterChange Server インスタンスの名前、*additionalOptions* はコネクター起動に渡す追加の始動パラメーターを示します。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、 240 ページの表 92を参照してください。

```
start_connName connName ICSinstance additionalOptions
```

図 79. 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 79 の start_*connName*.bat 構文では、*connName* および *ICSinstance* 引き数は必須です。*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 C++ コネクターのライブラリー・ファイル (その DLL: *connName*.dll) と同じ

例えば、MyCPP 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¥MyCPP ディレクトリーにあり、その DLL は MyCPP.dll 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。次の行により、始動スクリプト内でこの環境ファイルを呼び出します。

```
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、235 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 (238 ページの図 79 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 91 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 91. 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 (;) で区切ります。
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 91 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 91 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。将来この情報が変更された場合は、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 に指定する必要があります。表 92 に、start_adapter.bat スクリプトで認識される始動パラメーターを示します。

表 92. 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 への追加のコマンド行オプションとして有効ですか?
-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 以外である場合は必須	はい
-d*dllName*	C++ コネクターのライブラリー・ファイル (*dllName*) の名前。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 名には、.dll ファイル拡張子を組み込んではいけません。	(すべての C++ コネクターに対して) はい	いいえ
-f*pollFrequency*	ポーリング・アクション間の時間の長さです。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。コネクターは、コネクターのスタートアップ・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。key オプションは、小文字で指定する必要があります。 • no。コネクターはポーリングを実行しません。no オプションは、小文字で指定する必要があります。	いいえ デフォルトは 1000 ミリ秒	はい
-j	コネクターが Java で作成されていることを示します。	いいえ (Java コネクターに -l オプションを指定しない場合)	いいえ
-l*classname*	Java コネクターのコネクター・クラス名 (*className*)	(すべての Java コネクターに対して) はい	いいえ
-n*connectorName*	始動するコネクターの名前 (*connectorName*)	はい	いいえ
-s*brokerName*	コネクターが接続される統合ブローカーの名前 (*brokerName*)	はい	いいえ

表 92. *start_adapter.bat* 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?	start_connName.bat への追加のコマンド行オプションとして有効ですか?
-t	コネクター・プロパティー `SingleThreadAppCalls` をオンまたはオフにするブール値。これにより、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対して作成するすべての呼び出しが、呼び出しにより起動される 1 つのフローを伴うようになります。デフォルト値は `false` です。	いいえ	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。各プロパティーを指定する際には、次のフォーマットを使用します。 *propName=value*	いいえ	はい

`start_adapter.bat` への呼び出しに、次の始動パラメーターが含まれていることを確認してください。

- 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 (238 ページの図 79 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 (238 ページの図 79 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 オプションも必須の始動パラメーターとなります。

- C++ 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 DLL の名前を指定する場合: `-d`

例えば、推奨される命名規則に従うと、C++ コネクター名 MyCPP の言語固有のパラメーターは次のようになります。

```
-dMyCPP
```

コネクター名 (CONNAME など) の環境変数を定義する場合、この -d パラメーターは次のようになります。

```
-d%CONNAME%
```

- コネクターの起動時に毎回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のリストについては、 240 ページの表 92 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、MyCPP コネクターを呼び出します。

```
call start_adapter.bat -dMyCPP
    -nMyCPP -sICSserver -cMyCPPConnector.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ための行がすべて、始動スクリプトの *1 行だけに* 収まっていることを確認します。つまり、サンプル始動ファイルに示されている改行に復帰が存在してはならないということです。
- 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*。
- start_adapter.bat への呼び出しが、start_*connName*.bat への呼び出しに渡されるあらゆる追加オプションを処理するようにすることもできます。この場合、start_adapter.bat に渡す「追加の」引き数を指定して、追加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、次の start_adapter.bat への呼び出しは、最大 3 つの追加コマンド行オプションを処理します。

  ```
  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: ショートカットを使用するとコネクターを、「プログラム」>「IBM WebSphere Business Integration Adapters」>「アダプター」>「コネクター」 内のメニュー・オプションから始動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 (238 ページの図 79 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 C++ 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 `MyCPP` 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_MyCPP.bat MyCPP 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る場合は、ショートカット・コマンド行にはそのインスタンス名が現れます。

UNIX システムでの始動スクリプト

Connector Development Kit (CDK) は Windows システムでのみ サポートされます。UNIX ベースのシステムではサポートされていません。UNIX ベースのシステムでは C++ コネクターの作成がサポートされないため、このようなコネクターのために始動スクリプトを作成する必要はありません。

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 C++ コネクター・ライブラリー API リファレンス

第 9 章 C++ コネクター・ライブラリーの概要

C++ コネクター・ライブラリーには、コネクターの開発時に必要なクラス・ライブラリーが含まれています。このコネクター・ライブラリーには、C++ 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、コネクターのクラスやメソッドを派生させます。クラス・ライブラリーには、トレース・サービスやロギング・サービスをインプリメントするメソッドなどのユーティリティーも含まれています。

C++ コネクターの開発用として、C++ Connector Development Kit (CDK) は以下のバージョンの C++ コネクター・ライブラリーを提供します。

- CwConnector.dll - コネクターの初期化、ビジネス・オブジェクトの処理、および InterChange Server との対話のためにコネクターにインターフェースとユーティリティーを提供する C++ クラスおよびメソッドを含むダイナミック・リンク・ライブラリー。
- CwConnector.lib - CwConnector.dll のエクスポート・ライブラリー。

重要: CDK は Windows システムでのみ サポートされるため、C++ コネクターの作成は Windows システムでのみ サポートされます。

上記の CwConnector.lib ファイルと CwConnector.dll ファイルは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥lib
```

このディレクトリー内にある Release サブディレクトリーと Debug サブディレクトリーには、上記の開発ライブラリーのリリース・バージョンとデバッグ・バージョンが含まれています。

注: C++ コネクターを構築して Windows 上で稼働させる方法は、228 ページの『C++ コネクターのコンパイルとリンク』を参照してください。

クラス

表 93 に、C++ コネクター・ライブラリー内のクラスを示します。

表 93. C++ コネクター・ライブラリーのクラス

クラス	説明	ページ
BOAttrType	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。	249
BOHandlerCPP	ビジネス・オブジェクト・ハンドラーに対する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。	261
BusinessObject	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。	271
BusObjContainer	ビジネス・オブジェクトの配列を表します。これは、ビジネス・オブジェクト配列の要素へのアクセスを提供します。	293

© Copyright IBM Corp. 1997, 2004

表 93. C++ コネクター・ライブラリーのクラス (続き)

クラス	説明	ページ
BusObjSpec	ビジネス・オブジェクト定義を表します。これは、ビジネス・オブジェクトのアプリケーション固有の情報および属性プロパティーへのアクセスを提供します。	299
CxMsgFormat	国際化されたメッセージに対するサポートを提供します。*このクラスは使用しないでください。*	305
CxVersion	ビジネス・オブジェクトのオブジェクト・バージョンを表します。	309
GenGlobals	コネクターに対する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インプリメントします。	317
ReturnStatusDescriptor	エラー・メッセージと通知メッセージを含む戻り状況記述子を表します。	337
StringMessage	StringMessage オブジェクトの内容にアクセスするメソッドを提供します。	347
SubscriptionHandlerCPP	コネクターのサブスクリプション管理を処理するサブスクリプション・ハンドラーを表します。	341
Tracing	トレース・サービスを提供します。	349

注: BusObjAndSpecSerializer クラスは、CwConnector.dll ファイルに含まれています。ただし、これは内部でのみ 使用します。コネクターの開発時には、このクラスまたはメソッドを使用しないで ください。

第 10 章 BOAttrType クラス

BOAttrType クラスは、ビジネス・オブジェクトの属性を表します。 BOAttrType クラスの各インスタンスは、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。属性記述子には、ビジネス・オブジェクト (BusinessObject インスタンス) またはビジネス・オブジェクト定義 (BusObjSpec インスタンス) のいずれかを介してアクセスすることができます。BOAttrType クラスのメソッドを使用すると、各属性の名前、型、プロパティー、アプリケーション固有の情報など、プロパティー情報を取得できます。

このクラスのヘッダー・ファイルは BOAttrType.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

BOAttrType クラスの内容は以下のとおりです。

- 『属性タイプ定数』
- 『メンバー・メソッド』

属性タイプ定数

BOAttrType クラスは、属性タイプについて数値と型に相当するものを定義します。表 94 に、BOAttrType クラスに属する属性タイプ定数を示します。

表 94. BOAttrType クラスの静的定数

属性のデータ型	数値属性タイプ定数	ストリング属性タイプ値
ブール	BOOLEAN	"Boolean"
ビジネス・オブジェクト: 複数カーディナリティー		N/A
ビジネス・オブジェクト: 単一カーディナリティー		N/A
日付	DATE	"Date"
倍精度	DOUBLE	"Double"
浮動	FLOAT	"FLFS
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

メンバー・メソッド

BOAttrType クラスは、ビジネス・オブジェクト内の属性へのアクセスを提供するメソッドを定義します。表 95 に、BOAttrType クラスのメソッドを要約します。

表 95. *BOAttrType* 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
BOAttrType()	ビジネス・オブジェクト定義がビジネス・オブジェクトの属性として使用できる BOAttrType クラスのインスタンスを作成します。	250
getAppText()	属性に関するアプリケーション固有の情報を検索します。	251
getBOVersion()	属性のビジネス・オブジェクト定義のバージョンを検索します。	251
getCardinality()	属性が参照できる子ビジネス・オブジェクトの数を示します。これは、1 または n です。	252
getDefault()	属性のデフォルト値を検索します。	252
getMaxLength()	属性値の最大長を検索します。	252
getName()	属性の名前を検索します。	253
getRelationType()	親ビジネス・オブジェクトと子ビジネス・オブジェクトの関係型を検索します。	254
getTypeName()	属性のデータ型の名前を検索します。	254
getTypeNum()	属性のデータ型を指定する整数を検索します。	255
hasCardinality()	属性に指定されたカーディナリティーがあるかどうかを判別します。	255
hasName()	属性に指定された名前があるかどうかを判別します。	256
hasTypeName()	属性のデータ型に指定された型名があるかどうかを判別します。	256
isForeignKey()	属性がアプリケーション内の外部キー値かどうかを判別します。	257
isKey()	属性値がアプリケーション内のキー値かどうかを判別します。	257
isMultipleCard()	属性が複数の子ビジネス・オブジェクトを参照できるかどうかを判別します。	258
isObjectType()	属性のデータ型がビジネス・オブジェクト型かどうかを判別します。	258
isRequired()	属性がビジネス・オブジェクトに必須かどうかを判別します。	259
isType()	属性に指定されたデータ型があるかどうかを判別します。	259

BOAttrType()

ビジネス・オブジェクト定義がビジネス・オブジェクトの属性として使用できる属性記述子、BOAttrType クラスのインスタンスを作成します。

構文

```
public BOAttrType();
```

戻り値

新しくインスタンス化された属性記述子。これは、BOAttrType クラスのインスタンスです。

注意事項

BusObjSpec クラスの BusObjSpec() コンストラクターは、ビジネス・オブジェクト定義内の各属性の属性記述子をインスタンス化します。属性プロパティーに関する情報を検索するには、BOAttrType クラスの get メソッドを使用します。

関連項目

getAttrDesc(), getAttribute()

getAppText()

属性に関するアプリケーション固有の情報を検索します。

構文

```
char * getAppText();
```

パラメーター

なし。

戻り値

属性に関するアプリケーション固有の情報を含む文字ストリング。

注意事項

getAppText() メソッドを使用すると、ビジネス・オブジェクト定義属性の AppSpecificText プロパティーの値を検索できます。アプリケーション固有の情報がない場合、getAppText() メソッドは空ストリング (“ “) を戻します。

getBOVersion()

子ビジネス・オブジェクトが参照するビジネス・オブジェクト定義のバージョンを検索します。

構文

```
CXVersion getBOVersion();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義のバージョン。

注意事項

getBOVersion() メソッドを使用すると、子ビジネス・オブジェクトのビジネス・オブジェクト定義を判別できます。 getBOVersion() メソッドは、CxVersion オブジェクトのバージョン情報を戻します。CxVersion クラスのメソッドを使用して、バージョン情報を取得することができます。

注: この操作は、含まれているビジネス・オブジェクトを指定する属性に対してのみ 適用されます。属性にビジネス・オブジェクトが含まれていない場合、getBOVersion() メソッドは、デフォルト値の 0.0.0 でバージョン・ストリングを戻します。

getCardinality()

属性のカーディナリティーを検索します。

構文

```
char * getCardinality();
```

パラメーター

なし。

戻り値

属性のカーディナリティー。

`1`	属性に単一カーディナリティーがあることを示します。これは、1 つの子ビジネス・オブジェクトを参照できます。
`n`	属性に複数カーディナリティーがあることを示します。これは、複数の子ビジネス・オブジェクトを参照できます。

注意事項

カーディナリティーは、属性が、単一の子ビジネス・オブジェクトまたは子ビジネス・オブジェクトの配列のいずれを参照するかを定義します。この操作は、ビジネス・オブジェクトを含む属性に対してのみ適用されます。属性にビジネス・オブジェクトが含まれていない 場合、`getCardinality()` メソッドは値 1 を戻します。

関連項目

hasCardinality(), isMultipleCard()

getDefault()

属性のデフォルト値を検索します。

構文

```
char * getDefault();
```

パラメーター

なし。

戻り値

属性のデフォルト値。属性のデフォルト値が設定されていない場合、`getDefault()` メソッドは空ストリング (“ “) を戻します。

getMaxLength()

ビジネス・オブジェクト定義から属性の最大長を検索します。

構文

```
int getMaxLength();
```

パラメーター

なし。

戻り値

属性値が使用できるバイト数の最大長を指定する整数。

注意事項

最大長が 0 (ゼロ) の場合、その属性に対する長さ制限はありません。

getName()

属性の名前を検索します。

構文

```
char * getName();
```

パラメーター

なし。

戻り値

属性の名前を含む文字ストリング。

注意事項

getName() メソッドを使用すると、属性の名前を検索できます。

例

```
// Get the business object definition for the business object
BusinessObject *pBusObj = new BusinessObject ("Example");
BusObjSpec *theSpec = pBusObj->getSpecFor();
BOAttrType *theAttr;

for (int i=0; i < theSpec->getAttributeCount(); i++)
{
   // Determine Attribute Name
   theAttr = theSpec->getAttribute(i);

   // Set the attribute values to Fred
   pBusObj->setAttrValue(theAttr->getName(), "Fred"
   BOAttrType::STRING);

}
```

関連項目

hasName()

getRelationType()

親ビジネス・オブジェクトと子ビジネス・オブジェクトの関係型を検索します。

構文

```
char * getRelationType();
```

パラメーター

なし。

戻り値

関係型。現在は、包含が唯一の関係型です。これは、親ビジネス・オブジェクトに 1 つ以上の子ビジネス・オブジェクトが含まれていることを意味します。属性が子ビジネス・オブジェクトではない 場合、getRelationType() メソッドはヌル値を戻します。

関連項目

hasCardinality(), isMultipleCard(), isObjectType()

getTypeName()

属性のデータ型の名前をストリングとして検索します。

構文

```
char * getTypeName();
```

パラメーター

なし。

戻り値

属性のデータ型のストリング型名。getTypeName() メソッドは、表 96 に示されている属性タイプのストリングを戻します。

表 96. ストリング属性タイプ値

属性のデータ型	ストリング属性タイプ
ブール	"Boolean"
日付	"Date"
倍精度	"Double"
浮動	"Float"
整数	"Integer"
ロング・テキスト	"LongText"
オブジェクト	"Object"
ストリング	"String"

関連項目

getTypeNum(), hasTypeName(), isType()

getTypeNum()

現在の属性のデータ型の数値型コードを検索します。

構文

```
int getTypeNum();
```

パラメーター

なし。

戻り値

属性のデータ型の数値型コード。この整数値を表 97 に示す属性タイプ定数と比較することにより、型が判別されます。

表 97. 数値属性タイプ定数

属性のデータ型	数値属性タイプ定数
ブール	BOOLEAN
日付	DATE
倍精度	DOUBLE
浮動	FLOAT
整数	INTEGER
ロング・テキスト	LONGTEXT
オブジェクト	OBJECT
ストリング	STRING

注: 表 97 にリストされている数値の属性タイプ定数は、BOAttrType クラスによって定義されます。

例

```
if (theSpec->getAttribute(i)->getTypeNum() == BOAttrType::STRING)
{
   ...
}
```

関連項目

getTypeName(), isType()

hasCardinality()

属性に指定されたカーディナリティーがあるかどうかを判別します。

構文

```
unsigned char hasCardinality(char * card);
```

パラメーター

card **[in]** 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`

戻り値

属性に指定されたカーディナリティーがある場合は `TRUE`、ない場合は `FALSE`。

注意事項

この操作は、子ビジネス・オブジェクトを含む属性に対してのみ適用されます。

カーディナリティーを `null` として指定した場合、`hasCardinality()` メソッドは `FALSE` を戻します。

関連項目

getCardinality(), `getRelationType()`, `isMultipleCard()`

hasName()

属性に指定された名前があるかどうかを判別します。

構文

```
unsigned char hasName(char * name);
```

パラメーター

name **[in]** 属性の名前です。

戻り値

属性に指定された名前がある場合は `TRUE`、ない場合は `FALSE` を戻します。

注意事項

`hasName()` メソッドを使用すると、指定された属性をビジネス・オブジェクト定義が使用しているかどうかを判別できます。名前を *`null`* として指定した場合、`hasName()` メソッドは `FALSE` を戻します。

関連項目

getName()

hasTypeName()

属性に指定されたデータ型があるかどうかを判別します。

構文

```
unsigned char * hasTypeName(char * name);
```

パラメーター

name **[in]** 属性のデータ型の名前です。有効なストリング属性タイプの値のリストは、254 ページの表 96 を参照してください。

戻り値

属性に指定されたデータ型がある場合は TRUE、ない場合は FALSE を戻します。

注意事項

hasTypeName() メソッドを使用すると、この属性に対して指定されたデータ型をビジネス・オブジェクト定義が使用しているかどうかを判別できます。名前を null として指定した場合、hasTypeName() メソッドは FALSE を戻します。

関連項目

getTypeName()

isForeignKey()

属性値がアプリケーション内の外部キー値かどうかを判別します。

構文

```
unsigned char isForeignKey();
```

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・フィールドである場合は TRUE、属性が外部キー・フィールドではない場合は FALSE を戻します。

isKey()

属性値がアプリケーション内のキー値かどうかを判別します。

構文

```
unsigned char isKey();
```

パラメーター

なし。

戻り値

属性がビジネス・オブジェクトのキー・フィールドである場合は TRUE、属性がキー・フィールドではない場合は FALSE を戻します。

注意事項

isKey() メソッドを使用すると、ビジネス・オブジェクトのキー属性をすべて検索できます。

isMultipleCard()

属性に複数カーディナリティーがあるかどうかを判別します。

構文

```
unsigned char isMultipleCard();
```

パラメーター

なし。

戻り値

属性に複数カーディナリティーがある場合は TRUE、ない場合は FALSE。

注意事項

この操作は、子ビジネス・オブジェクトを含む属性に対してのみ適用されます。属性が子ビジネス・オブジェクトを持たない単純な属性である場合、isMultipleCard() メソッドは FALSE を戻します。

関連項目

getCardinality() メソッドおよび hasCardinality() メソッドの説明も参照してください。

isObjectType()

属性のデータ型がオブジェクト・タイプあるいは複合属性 (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

```
unsigned char isObjectType();
```

パラメーター

なし。

戻り値

属性にビジネス・オブジェクト型がある場合は TRUE、ない場合は FALSE。

注意事項

ビジネス・オブジェクト・データ型を持つ属性は、子ビジネス・オブジェクトを参照します。

isRequired()

ビジネス・オブジェクトに属性の値が必須かどうかを判別します。

構文

```
unsigned char isRequired();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクトに属性が必須である場合は TRUE、必須でない場合は FALSE を戻します。

注意事項

isRequired() メソッドを使用すると、ビジネス・オブジェクトに必須の属性をすべて検索できます。

isType()

属性に指定された整数データ型があるかどうかを判別します。

構文

```
unsigned char isType(int type);
```

パラメーター

type **[in]** 属性のデータ型を指定する以下の属性タイプ定数のいずれかです。

```
BOAttrType::OBJECT
BOAttrType::BOOLEAN
BOAttrType::INTEGER
BOAttrType::FLOAT
BOAttrType::DOUBLE
BOAttrType::STRING
BOAttrType::DATE
BOAttrType::LONGTEXT
```

注意事項

isType() メソッドを使用すると、ビジネス・オブジェクト定義内で特定のデータ型の属性を検索できます。無効なデータ型を指定した場合、isType() メソッドは FALSE を戻します。

例

```
char *cp = NULL;
if(getTheSpec()->getAttribute(name)->isType(BOAttrType::STRING))

{
   cp = new char[strlen(newval)+1];
   strcpy(cp, newval);

   Values[getTheSpec()->getAttributeIndex(name)] = cp;
}
```

関連項目

getTypeNum()

第 11 章 BOHandlerCPP クラス

BOHandlerCPP クラスは、C++ 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。これには、ビジネス・オブジェクト・ハンドラーのコードが含まれています。コネクター開発者は、このクラスから必要な数のビジネス・オブジェクト・ハンドラー・クラスを派生により作成し、ビジネス・オブジェクト・ハンドラー用の抽象メソッド doVerbFor() をインプリメントする必要があります。

重要: すべての C++ コネクターがこの仮想クラスを拡張する必要があります。開発者は、派生したビジネス・オブジェクト・ハンドラー・クラスで単一仮想メソッド doVerbFor() をインプリメント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、データの検索、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、28 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・ハンドラーのインプリメント方法については、83 ページの『第 4 章 要求処理』を参照してください。

このクラスのヘッダー・ファイルは BOHandlerCPP.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

表 98 に、BOHandlerCPP クラスのメソッドを要約します。

表 98. BOHandlerCPP 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
BOHandlerCPP()	ビジネス・オブジェクト・ハンドラーを作成します。	262
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。	262
generateAndLogMsg()	メッセージ・ファイル内の定義済みメッセージのセットからメッセージを生成し、生成したメッセージをコネクターのログ宛先に記録します。	264
generateAndTraceMsg()	メッセージ・ファイル内の定義済みメッセージのセットからトレース・メッセージを生成し、生成したトレース・メッセージをコネクターのログ宛先に送信します。	265
generateMsg()	メッセージ・ファイル内の定義済みメッセージのセットからメッセージを生成します。	266
getConfigProp()	リポジトリーから、コネクター構成プロパティーを検索します。	267

© Copyright IBM Corp. 1997, 2004

表 98. *BOHandlerCPP 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
getTheSubHandler()	サブスクリプション・ハンドラーへのポインターを検索します。呼び出し元はこのポインターを使用し、着信ビジネス・オブジェクトについて特定のビジネス・オブジェクト定義へのサブスクリプションが存在するかどうかを判別することができます。	268
logMsg()	コネクターのログの宛先にメッセージを記録します。ログ・メッセージは、コネクター用のメッセージ・ファイルに組み込む必要があります。	269
traceWrite()	コネクターのログの宛先にメッセージを記録します。ログ・メッセージは、コネクター用のメッセージ・ファイルに組み込む必要があります。	269

BOHandlerCPP()

ビジネス・オブジェクト・ハンドラーを作成します。

構文

```
BOHandlerCPP();
```

パラメーター

なし。

戻り値

なし。

注意事項

BOHandlerCPP() コンストラクターは、BOHandlerCPP 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 BOHandlerCPP からクラスを派生させ、その派生クラスに対してコンストラクターおよび doVerbFor() メソッドをインプリメントします。開発者は、この派生クラスのコンストラクターを、GenGlobals クラスの getBOHandlerforBO() 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。

関連項目

getBOHandlerforBO()

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。

構文

```
virtual int doVerbFor(BusinessObject & theBusObj,
   ReturnStatusDescriptor * rtnStatusDesc);
```

パラメーター

theBusObj **[in]** 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。

rtnStatDesc **[out]**
doVerbFor() により更新される戻り状況記述子オブジェクトです。この更新には、操作の状況を示すために統合ブローカーに送られるエラー・メッセージまたは通知メッセージが使用されます。

戻り値

動詞操作の結果状況を示す整数です。状況を判定するため、この整数値を次の結果状況定数と比較してください。

BON_SUCCESS 動詞操作が成功しました。

BON_FAIL 動詞操作が失敗しました。

BON_APPRESPONSETIMEOUT
アプリケーションが応答しません。

BON_BO_DOES_NOT_EXIST
コネクターは検索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。

BON_MULTIPLE_HITS
コネクターが非キー値を使用して検索中に複数の一致レコードを見つけました。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。

BON_FAIL_RETRIEVE_BY_CONTENT
コネクターは、非キー値による検索で、一致レコードを検出できませんでした。

BON_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。

BON_VALDUPES 要求された操作によって、アプリケーション・データベース内に同じキー値のレコードが複数検出されました。

注意事項

doVerbFor() メソッドは、ビジネス・オブジェクトのアクティブ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。

重要: doVerbFor() メソッドは仮想メソッドです。したがって、コネクターはこのメソッドをビジネス・オブジェクト・ハンドラーの一部としてインプリメントする必要があります。

ビジネス・オブジェクトが InterChange Server から到着すると、コネクター・フレームワークは戻り状況記述子オブジェクトを作成し、ビジネス・オブジェクトとともに doVerbFor() メソッドに渡します。このメソッドは動詞操作を実行し、その後、次のように戻り状況記述子に適切な値を設定するため、ReturnStatusDescriptor クラスのメソッドを呼び出します。

seterrMsg()	通知、警告、またはエラーの戻りメッセージが存在する場合は、戻り状況記述子オブジェクトにメッセージを設定します。

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に戻します。また、doVerbFor() メソッドの戻りコードである結果状況も戻します。

関連項目

seterrMsg()

generateAndLogMsg()

メッセージ・ファイル内の定義済みメッセージのセットからメッセージを生成し、生成したメッセージをコネクターのログ宛先に記録します。

構文

```
void generateAndLogMsg(int msgNum, int msgType, int argCount, ...);
```

パラメーター

msgNum **[in]**
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 **[in]**
CxMsgFormat クラスで定義された以下のメッセージ・タイプ定数のいずれかです。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

argCount **[in]**
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。

... **[in]**
メッセージ・テキスト用のメッセージ・パラメーターのリストです。

戻り値

なし。

注意事項

generateAndLogMsg() メソッドは、generateMsg() メソッドと logMsg() メソッドの機能を結合したメソッドです。これら 2 つのメソッドを結合することによって、generateAndLogMsg() は、generateMsg() が生成するメッセージ・ストリングに必要なメモリーを解放します。

注: generateAndLogMsg() メソッドは、GenGlobals クラスでも使用することができます。これは、ビジネス・オブジェクト・ハンドラー内からロギングにアクセスするために、BOHandlerCPP クラス内に用意されています。

例

以下の例は、generateMsg() メソッドの場合と同じタスクを実行します。

```
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
      msg = generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, 0, NULL);
      return BON_FAIL;
}
```

generateAndTraceMsg()

メッセージ・ファイル内の定義済みメッセージのセットからトレース・メッセージを生成し、生成したトレース・メッセージをコネクターのログ宛先に送信します。

構文

```
void generateAndTraceMsg(int msgNum, int msgType, int traceLevel,
      int argCount, ...);
```

パラメーター

msgNum **[in]**
: 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 **[in]**
: CxMsgFormat クラスで定義された以下のメッセージ・タイプ定数のいずれかです。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

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、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。そのため、LEVEL0 の *traceLevel* に関連付けられたトレース・メッセージはいずれも印刷されません。

argCount **[in]**
: 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。

... **[in]** メッセージ・テキスト用のメッセージ・パラメーターのリストです。

戻り値

なし。

注意事項

generateAndTraceMsg() メソッドは、generateMsg() メソッドと traceWrite() メソッドの機能を結合したメソッドです。これら 2 つのメソッドを結合することによって、generateAndTraceMsg() は、generateMsg() が生成するメッセージ・ストリングに必要なメモリーを解放します。メッセージ・ストリングに割り振るメモリーを解放するために freeMemory() メソッドを呼び出す必要はなくなります。

注: generateAndTraceMsg() メソッドは、GenGlobals クラスでも使用することができます。これは、ビジネス・オブジェクト・ハンドラー内からトレースにアクセスするために、BOHandlerCPP クラス内に用意されています。

例

```
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
   // Message 3033 - Opened main form for object
   msg = generateAndTraceMsg(3033,CxMsgFormat::XRD_FATAL,
      Tracing::LEVEL3,0, NULL);
}
```

generateMsg()

メッセージ・ファイル内の定義済みメッセージのセットからメッセージを生成します。

構文

```
char * generateMsg(int msgNum, int msgType, char * info,
      int argCount, ...);
```

パラメーター

msgNum **[in]** 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 **[in]** CxMsgFormat クラスで定義された以下のメッセージ・タイプのいずれかです。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]** IBM WebSphere Business Integration システム がメッセージを生成したクラスの名前など、情報を示す値です。

argCount **[in]** メッセージ・テキスト (オプション) 内のパラメーターの個数を示す整数です。

... **[in]** メッセージ・テキストの各メッセージ・パラメーターをコンマで区切ったオプションのリストです。各パラメーターは char * 値です。

戻り値

生成されたメッセージへのポインター。

注意事項

generateMsg() メソッドは、生成されたメッセージを格納するためのメモリーを割り振ります。コネクターはメッセージを記録すると、割り振られたメモリーを解放するため freeMemory() メソッドを呼び出します。このメソッドは、コネクター・フレームワーク・クラス JToCPPVeneer のメンバーです。この呼び出しの構文は void freeMemory(char * mem) です。ここで、mem は generateMsg() によって割り振られるメモリーです。このメソッドの呼び出し方法の例は、以下のサンプル・コードを参照してください。

msgtype パラメーターが無効な場合、メッセージ生成プロセスは検証を行いません。generateMsg() メソッドによって、メッセージがメッセージ・ファイル内にないという警告が表示されます。

注: generateMsg() メソッドは、GenGlobals クラスでも使用することができます。これは、ビジネス・オブジェクト・ハンドラー内からメッセージ・ファイル・メッセージにアクセスするために、BOHandlerCPP クラス内に用意されています。

例

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
   logMsg(msg);
   JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
   return BON_FAIL;
}
```

getConfigProp()

リポジトリーから、コネクター構成プロパティーを検索します。

構文

int getConfigProp(char * *prop*, char * *val*, int *nMaxCount*);

パラメーター

prop **[in]** 検索するプロパティーの名前です。

val **[out]** メソッドがプロパティー値を書き込むことができるバッファーへのポインターです。

nMaxCount **[in]**
値バッファー内のバイト数です。

戻り値

メソッドが値バッファーにコピーしたバイト数を示す整数です。

注意事項

並列処理コネクター (ParallelProcessDegreeコネクター・プロパティーに 1 より大きい値がセットされているコネクター) において、getConfigProp("ConnectorName") を呼び出すと、マスター・プロセスまたはスレーブ・プロセスのいずれかが呼び出し元であるかに関係なく、このメソッドは、常時、コネクター・エージェント・マスター・プロセスの名前を戻します。

例

```
if (getConfigProp("LoginId", val, 255) == 0);
{
   logMsg("Invalid LoginId";
   traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
}
```

getTheSubHandler()

サブスクリプション・ハンドラーへのポインターを検索します。呼び出し元はこのポインターを使用し、着信ビジネス・オブジェクトについて特定のビジネス・オブジェクト定義へのサブスクリプションが存在するかどうかを判別することができます。

構文

```
SubscriptionHandlerCPP * getTheSubHandler() const;
```

パラメーター

なし。

戻り値

サブスクリプション・ハンドラーへのポインター。

注意事項

コネクターはサブスクリプション・ハンドラーを介し、アクティブな全サブスクリプションの統合リストで、コネクターがパブリッシュする各ビジネス・オブジェクト定義の各動詞のサブスクライバーを追跡します。

例

```
if (getTheSubHandler->isSubscribed(theObj->getName(), "Create"){
}
```

関連項目

BusinessObject クラスと BusObjSpec クラスの説明を参照してください。

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

```
void logMsg(char * msg);
```

パラメーター

msg **[in]** メッセージ・テキストへのポインターです。

戻り値

なし。

注意事項

logMsg() に対するメッセージ・ストリングを生成するには、generateMsg() メソッドを使用します。generateMsg() 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検索し、テキストの形式を設定し、生成されたメッセージ・ストリングへのポインターを戻します。

logMsg() でログに記録されたコネクター・メッセージは、メッセージ・ストリングが generateMsg() で生成された場合には、LogViewer を使用して表示することができます。traceWrite() でログに記録されたトレース・メッセージは、LogViewer に表示されません。

注: logMsg() メソッドは、GenGlobals クラスでも使用することができます。これは、ビジネス・オブジェクト・ハンドラー内からロギングにアクセスするために、BOHandlerCPP クラス内に用意されています。

例

```
if ((form = CreateMainForm(conn, getFormName(theObj))) < 0) {
   msg = generateMsg(10, CxMsgFormat::XRD_FATAL, NULL, 0, NULL);
   logMsg(msg);
}
```

関連項目

GenGlobals::generateMsg() ユーティリティーの説明を参照してください。

traceWrite()

ログ宛先にトレース・メッセージを書き込みます。

構文

```
void traceWrite(int traceLevel, char * info,
        char * filterName);
```

パラメーター

traceLevel **[in]** メッセージの書き込みに使用する以下のトレース・レベルのいずれかです。

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::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、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。そのため、LEVEL0 の *traceLevel* に関連付けられたトレース・メッセージはいずれも印刷されません。

info **[in]** メッセージ・テキストへのポインターです。

filterName **[in]** メッセージの書き込みに使用するフィルターへのポインターです。

戻り値

なし。

注意事項

traceWrite() メソッドを使用すると、アプリケーション用に独自のトレース・メッセージを書き込むことができます。

フィルターを使用せずにトレース・メッセージを書き込むには、filterName に NULL を指定します。

注: traceWrite() メソッドは、GenGlobals クラスでも使用することができます。これは、ビジネス・オブジェクト・ハンドラー内からトレースにアクセスするために、BOHandlerCPP クラス内に用意されています。

例

```
traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
```

関連項目

Tracing クラスの説明も参照してください。

第 12 章 BusinessObject クラス

BusinessObject クラスは、アプリケーション用のビジネス・オブジェクトを表します。 BusinessObject クラス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、BusObjSpec クラスの単一のインスタンスを参照します。

このクラスのヘッダー・ファイルは BusinessObject.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

DevelopmentKits¥cdk¥generic_include

BusinessObject クラスの内容は以下のとおりです。

- 『属性値定数』
- 『メンバー・メソッド』

属性値定数

表 99 は、BusinessObject クラス内の属性値定数をまとめたものです。

表 99. BusinessObject クラスの静的定数

特殊属性値	属性値定数
Blank	BlankValue
Ignore	IgnoreValue

メンバー・メソッド

表 100 に、BusinessObject クラスのメソッドを要約します。

表 100. BusinessObject 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
BusinessObject()	ビジネス・オブジェクト定義 (BusObjSpec) を参照する新規ビジネス・オブジェクトを作成します。	272
clone()	既存のビジネス・オブジェクトをコピーします。	273
doVerbFor()	ビジネス・オブジェクト・ハンドラー (BOHandlerCPP 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。	274
dump()	ロギングおよびトレース用の標準または定義された形式でビジネス・オブジェクト情報の形式を設定して戻します。	275
getAttrCount()	ビジネス・オブジェクトが持っている属性の個数を検索します。	275
getAttrDesc()	属性の説明 (BOAttrType) を名前または位置で検索します。	276
getAttrName()	位置によって、属性の名前を検索します。	277
getAttrType()	属性タイプを名前または位置で検索します。	277
getAttrValue()	属性値を名前または位置で検索します。	278
getBlankValue()	特殊ブランク値を検索します。	286
getDefaultAttrValue()	属性値のデフォルト値を名前または位置で検索します。	280

© Copyright IBM Corp. 1997, 2004

表 100. *BusinessObject* 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
getIgnoreValue()	特殊「ignore」値を検索します。	281
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検索します。	281
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検索します。	282
getParent()	現在のビジネス・オブジェクトの親ビジネス・オブジェクトを検索します。	282
getSpecFor()	ビジネス・オブジェクトが参照するビジネス・オブジェクト定義 (BusObjSpec) の名前へのポインターを検索します。	282
getVerb()	ビジネス・オブジェクト用のアクティブな動詞を検索します。	283
getVersion()	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バージョンを検索します。	284
initAndValidateAttributes()	コネクター構成プロパティー UseDefaults が TRUE の場合、このメソッドは、NULL 値を持つ属性をビジネス・オブジェクト定義のデフォルト値に設定します。	284
isBlank()	指定された名前または位置の属性の値がブランクであるかどうかを判別します。	286
isBlankValue()	指定された値がブランクかどうかを判別します。	286
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。	287
isIgnoreValue()	指定された値が「ignore」値かどうかを判別します。	287
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。通常、この操作は、子オブジェクトを含む属性にのみ適用されます。	288
setAttrValue()	属性の値を名前または位置で設定します。	288
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。	289
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。	290
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。	290

BusinessObject()

BusinessObject クラス (ビジネス・オブジェクト) のインスタンスを作成します。新規ビジネス・オブジェクトは、ビジネス・オブジェクト定義の最新バージョンまたは指定したバージョンのいずれかで、BusObjSpec クラス (ビジネス・オブジェクト定義) のインスタンスを参照します。

構文

```
BusinessObject(char * busObjName);
BusinessObject(char * busObjName, CxVersion & version);
BusinessObject(char * busObjName, char * localeName);
```

パラメーター

busObjName **[in]**
: 新規ビジネス・オブジェクトの名前。

version **[in]**
: 新規ビジネス・オブジェクトが参照するビジネス・オブジェクト定義のバージョン番号。バージョンを指定しない場合、新規ビジネス・オブジェクトは最新バージョンのビジネス・オブジェクト定義を参照します。バージョン番号は String 値です。

*localeName***[in]**
: 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。

注意事項

`BusinessObject()` コンストラクターは、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。

ビジネス・オブジェクト (`BusinessObject` クラスのインスタンス) には、属性値のセットが含まれています。属性の定義は、ビジネス・オブジェクトが参照するビジネス・オブジェクト定義内にあります。ビジネス・オブジェクト定義は属性ごとに、名前、属性リスト内での位置、データ型、およびいくつかのプロパティーを定義します。また、ビジネス・オブジェクト定義には、ビジネス・オブジェクトがサポートする動詞のリストも含まれています。

`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。ビジネス・オブジェクト定義名やその属性名 (英文字でなければならない) には*適用されません*。ロケールの形式については、 64 ページの『国際化対応のコネクターにおける設計上の考慮事項』を参照してください。

ビジネス・オブジェクト・コンストラクターが失敗したかどうかを判別するには、このクラスの `getSpecFor()` メソッドを使用してビジネス・オブジェクト定義ポインターを検査します。コンストラクターが失敗した場合、`getSpecFor()` メソッドは `NULL` を戻します。無効な名前を指定すると、コンストラクターが失敗する場合があります。

例

```
BusinessObject *pObj = new BusinessObject("Customer");
```

関連項目

`BusObjSpec` クラスの説明も参照してください。

---

clone()

既存のビジネス・オブジェクトをコピーします。

構文

```
BusinessObject * clone();
```

パラメーター

なし。

戻り値

現在のビジネス・オブジェクトのコピー。

関連項目

BusinessObject() コンストラクターの説明も参照してください。

doVerbFor()

ビジネス・オブジェクト・ハンドラー (BOHandlerCPP クラスのインスタンス) を呼び出して、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します。

構文

```
int doVerbFor(ReturnStatusDescriptor * retStatusMsg);
```

パラメーター

retStatusMsg **[out]**
統合ブローカーへのエラー・メッセージまたは通知メッセージを含む状況記述子オブジェクトの名前。

戻り値

動詞操作による結果ステータスを指定する整数。

注意事項

ビジネス・オブジェクトは、BusinessObject 定義がサポートする動詞の全操作を提供します。

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。

例

```
BusinessObject *pObj;
...
pObj = new BusinessObject("Customer");
pObj->SetVerb ("Create");
pObj->setDefaultAttrValues();
retval = pObj->doVerbFor();
```

関連項目

getVerb(), setVerb()

BOHandler クラスの説明も参照してください。

dump()

ロギングおよびトレース用の形式でビジネス・オブジェクト情報を戻します。

構文

```
char * dump(char * buf, int bufSize);
```

パラメーター

buf **[in]** ビジネス・オブジェクト情報を格納するバッファーのアドレス。

bufSize **[in]** バッファーのサイズ。

戻り値

操作の結果を示す整数。

注意事項

保存されたビジネス・オブジェクトがバッファーに収まらない場合、dump() メソッドはエラー・コード -1 を戻します。

例

```
BusinessObject *pObj;
int status;
. . .
status = pObj->dump(buf, 255);
```

関連項目

Tracing クラスの説明も参照してください。

getAttrCount()

ビジネス・オブジェクトの属性リスト内にある属性の数を検索します。

構文

```
int getAttrCount();
```

パラメーター

なし。

戻り値

属性リスト内にある属性の数を指定する整数。

例

```
for(i=0; i<pObj.getAttrCount(); i++){
   char*cp;

   ...
   pObj.setAttrValue(i,cp,theObj.getAttrType(i));
}
```

関連項目

299 ページの『第 14 章 BusObjSpec クラス』の getAttrIndex() の説明も参照してください。

getAttrDesc()

属性の名前またはビジネス・オブジェクトの属性リスト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属性記述子を検索します。

構文

```
BOAttrType * getAttrDesc(char * attrName);
BOAttrType * getAttrDesc(int position);
```

パラメーター

attrName **[in]** 属性記述子が検索される属性の名前。

position **[in]** 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 (序数) を指定する整数。

戻り値

BOAttrType クラスのインスタンスへのポインター。無効な属性名または位置が指定されると、このメソッドは NULL を戻します。

注意事項

getAttrDesc() メソッドは、ビジネス・オブジェクト (BusinessObject インスタンス) 内の属性について、BOAttrType クラスのインスタンスである属性記述子を戻します。BOAttrType クラスは、属性プロパティーに関する情報を取得するメソッドを提供します。属性の属性記述子を検索するには、属性の名前または属性リストにおけるその位置 (序数) で属性を特定します。属性名として空ストリング (“ “) または NULL を指定すると、getAttrDesc() メソッドは 0 (ゼロ) を戻します。

注: コネクターがトレース・レベル 5 で動作している場合は、該当するトレース・メッセージも印刷されます。

関連項目

getAttribute() getAttrName() getAttrType() getAttrValue()

BOAttrType クラスのメソッドについては、 249 ページの『第 10 章 BOAttrType クラス』を参照してください。

getAttrName()

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で指定された属性の名前を検索します。

構文

```
char * getAttrName(int position);
```

パラメーター

position **[in]** 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 (序数) を指定する整数。

戻り値

指定された属性の名前。無効な位置が指定されると、このメソッドは NULL を戻します。

例

```
strcpy(attr_name, pObj.getAttrName(1));

for (int i = 0; i<pObj.getAttrCount(); i++) {
   name = pObj.getAttrName(i);
   value = pObj.getAttrValue(i);
   cout << "name: " << name << "value " << value;
}
```

getAttrType()

属性の名前またはビジネス・オブジェクトの属性リスト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ータ型を検索します。

構文

```
int getAttrType(char * attrName);
int getAttrType(int position);
```

パラメーター

attrName **[in]** データ型が検索される属性の名前。

position **[in]** 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 (序数) を指定する整数。

戻り値

属性のデータ型を表す整数。属性データ型は BOAttrType.hpp で定義されます。

0 = BOAttrType::OBJECT1 = BOAttrType::BOOLEAN2 = BOAttrType::INTEGER3 = BOAttrType::FLOAT4 = BOAttrType::DOUBLE5 = BOAttrType::STRING6 = BOAttrType::DATE7 = BOAttrType::LONGTEXT

注意事項

ビジネス・オブジェクトの属性のデータ型を検索するには、属性の名前または属性リストにおけるその位置を指定します。属性名として空ストリング (“ “) を渡した場合や無効な位置を渡した場合、getAttrType() メソッドは -1 を戻します。コネクターがトレース・レベル 5 で動作している場合は、該当するトレース・メッセージも生成されます。

例

```
pObj.setAttrValue("sti_address.docid","1234",
   pObj.getAttrType("sti_address.docid"));
```

関連項目

getAttrDesc() メソッドおよび getAttrName() メソッドの説明も参照してください。

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検索します。

構文

```
void * getAttrValue(char * attrName);
void * getAttrValue(int position);
```

パラメーター

name **[in]**
値が検索される属性の名前。

position **[in]**
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 (序数) を指定する整数。

戻り値

属性のデータ型用に定義された形式での、指定された属性の値。無効な位置が指定されると、このメソッドは NULL を戻します。

注意事項

getAttrValue() を使用する場合は、戻された値を変数に割り当てる前に、getAttrType() で属性のタイプを検査してください。属性タイプに基づいて void ポインターを文字ポインター、ビジネス・オブジェクト・ポインター、またはビジネス・オブジェクト・コンテナーにキャストしてから、戻された値を変数に割り当てます。

WebSphere InterChange Server

特殊な「Blank」値や「Ignore」値を処理するため、ビジネス・オブジェクトやビジネス・オブジェクト・コンテナーではない属性値は、C++ でストリングとして保管します。 Blank 値は「このフィールドを消去します。フィールド内にデータはありません」ということを意味し、Ignore 値は「コラボレーションはこのフィールドの内容を認識または考慮しません」ということを意味します。

ビジネス・オブジェクトやビジネス・オブジェクト・コンテナーではない属性値を処理するには、void ポインターを文字ポインターにキャストしてから、その属性が特殊値かどうかを判別します。属性値が特殊値ではなく、かつ string 以外のデータである場合は、値を正しい型に変換する必要があります。

属性名または位置のパラメーターとして空ストリング (“ “) または NULL を渡すと、getAttrValue() メソッドはエラー・メッセージ 73 を表示します。

属性位置 "{1}" または属性タイプ "{2}" が無効であるため、属性値の設定に失敗しました。

値が設定されていない特定の属性がビジネス・オブジェクトにある場合、getAttrValue() メソッドは NULL を戻します。

例

```
BusinessObject *pObj;
void *attr;
...
attr = pObj->getAttrValue(0):
if (pObj->getAttrType(0) == BOAttrType::STRING)
{
   char *attrStr = (char *) attr;
   if (pObj->isIgnoreValue(attrStr))
   {
      // ignore this value
   }
   else
      if (pObj->isBlankValue(attrStr))
      {
         // value should be blank
      }
...
}
```

関連項目

getAttrName() getDefaultAttrValue() setAttrValue()

getBlankValue()

特殊属性値 Blank を検索します。

構文

```
static char * getBlankValue();
```

パラメーター

なし。

戻り値

特殊 Blank 属性値です。

関連項目

getIgnoreValue() isBlank() isBlankValue()

---

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検索します。

構文

```
char * getDefaultAttrValue(char * attrName);
char * getDefaultAttrValue(int position);
```

パラメーター

name **[in]**
デフォルト値が検索される属性の名前。

position **[in]**
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 (序数) を指定する整数。

戻り値

属性のデータ型用に定義された形式での、指定された属性のデフォルト値。このメソッドは、以下の場合に `NULL` を戻します。

- 無効な位置または名前が指定された場合
- デフォルト値が `NULL` である場合
- 属性タイプが `BOAttrType::OBJECT` である場合

注意事項

ビジネス・オブジェクトの属性のデフォルト値を検索するには、属性の名前または属性リストにおけるその位置を指定します。パラメーターとして空ストリングまたは `NULL` を渡すと、`getDefaultAttrValue()` メソッドは「ignore 値」0 (ゼロ) を戻します。

属性は、特殊デフォルト値、ブランクまたは「ignore」をとることもあります。Blank は「このフィールドを消去します。フィールド内にデータはありません」ということを意味し、Ignore は「このフィールドの内容を認識または考慮しません」ということを意味します。

例

```
tempStr = pObj->getDefaultAttrValue(i);
```

関連項目

getAttrValue() setDefaultAttrValues()

getIgnoreValue()

特殊 Ignore 値を検索します。

構文

```
static char * getIgnoreValue();
```

パラメーター

なし。

戻り値

特殊 Ignore 属性値を含むストリング。

注意事項

Ignore 値は、コネクターがこの属性の値を無視する場合があることを示します。

関連項目

getBlankValue() isIgnore(), isIgnoreValue()

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検索します。

構文

```
char * 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を含むストリング。ロケール名の形式については、 64 ページの『ロケールとは』を参照してください。

注意事項

getLocale() メソッドは、ビジネス・オブジェクトの作成時にビジネス・オブジェクトのフローに関連付けられた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付けられている言語およびコード・セットを示します。ビジネス・オブジェクト定義名やその属性名 (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らない) には適用されません。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、そのビジネス・オブジェクトに対して、コネクター・フレームワークのロケールを使用します。

関連項目

BusinessObject(), 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検索します。

構文

```
char * 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。

関連項目

getVersion()

getParent()

現在のビジネス・オブジェクトの親ビジネス・オブジェクトを検索します。

構文

```
BusinessObject *getParent() const;
```

戻り値

現在のビジネス・オブジェクトを含むビジネス・オブジェクト。ビジネス・オブジェクトが階層内のトップレベルのオブジェクトである場合、このメソッドは NULL を戻します。

getSpecFor()

ビジネス・オブジェクトが参照するビジネス・オブジェクト定義 (BusObjSpec インスタンス) へのポインターを検索します。

構文

```
BusObjSpec * getSpecFor();
```

戻り値

ビジネス・オブジェクトが参照するビジネス・オブジェクト定義へのポインター。

注意事項

getSpecFor() メソッドを使用すると、ビジネス・オブジェクトの属性に関する情報を検索できます。

例

```
int i = pObj.getSpecFor()->getAttributeIndex(name);
```

関連項目

BusObjSpec クラスの説明も参照してください。

getVerb()

ビジネス・オブジェクト用のアクティブな動詞を検索します。

構文

```
char * getVerb();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト用のアクティブな動詞。動詞がビジネス・オブジェクト内で設定されていない場合、このメソッドは NULL を戻します。

注意事項

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サポート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・インスタンスに対して動詞がアクティブかどうかを判別できます。

例

```
char * verb = pObj->getVerb();

if ((verb!=NULL) && (strcmp(verb,"Create")) == 0 {
   // perform the create operation
   ...
}
```

関連項目

doVerbFor(), setVerb()

getVersion()

ビジネス・オブジェクトが参照するビジネス・オブジェクト定義のバージョンを検索します。

構文

```
CxVersion * getVersion();
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義のバージョン。

関連項目

getName()

initAndValidateAttributes()

デフォルト値で属性を初期化し、必須の値が存在していることを確認します。

構文

```
bool initAndValidateAttributes();
```

パラメーター

なし。

戻り値

デフォルト値の処理後に、すべての必須属性が設定済みである (非ヌル値を持っている) 場合は、TRUE を戻します。必須の値が設定されておらず、かつビジネス・オブジェクト定義内の属性に対してデフォルト値が指定されていない場合は、FALSE を戻します。

注意事項

initAndValidateAttributes() メソッドには 2 つの目的があります。

- 以下の条件のときに各属性の値をデフォルト値に設定して、属性を*初期設定* します。
  - UseDefaults コネクター構成プロパティーが true に設定されているとき
  - 属性の isRequired プロパティーが true に設定されている (その属性が必須である) とき

- 属性の値が現在設定されていない (特殊 Ignore 値、CxIgnore をもっている) とき
  - 属性の Default Value プロパティーでデフォルト値が指定されているとき
- 以下の条件のときに FALSE を戻すことで、属性の*検証* を行います。
  - 属性の isRequired プロパティーが true に設定されているとき
  - 属性に、そのデフォルト値を定義する Default Value プロパティーがない とき

  失敗した場合は、initAndValidateAttributes() がデフォルト値の処理を完了した後に、一部の属性 (デフォルト値をもたない属性) の値が存在しません。この例外を catch し、BON_FAIL を戻すため、ご使用の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が必要な場合もあります。

initAndValidateAttributes() メソッドは、ビジネス・オブジェクトの全レベルですべての属性を調べ、以下の点について判別します。

- 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・インスタンス内に値をもっているかどうか
- UseDefaults 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。属性にデフォルト値がない 場合、initAndValidateAttributes() は FALSE を戻します。

注: 属性がキーである場合や、アプリケーションによって値が生成されるその他の属性である場合、ビジネス・オブジェクト定義でデフォルト値を指定せず、属性の Required プロパティーを false に設定する必要があります。

initAndValidateAttributes() メソッドは通常、ビジネス・オブジェクト・ハンドラー doVerbFor() メソッドから呼び出され、必須の属性に値が設定された後に、アプリケーションで Create 操作が実行されるように保証します。 doVerbFor() メソッドでは、Create 動詞に対して initAndValidateAttributes() を呼び出すことができます。また、Create を実行する前に、Update 動詞に対してこのメソッドを呼び出すこともできます。

initAndValidateAttributes() を使用するには、以下も実行する必要があります。

- 必須の属性について IsRequired プロパティーが true に設定され、その必須の属性の Default Value プロパティーでデフォルト値が指定されるように、ビジネス・オブジェクト定義を設計します。
- UseDefaults コネクター構成プロパティーを、コネクター用のコネクター固有プロパティーに追加します。このプロパティーを true に設定します。

例

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
   ReturnStatusDescriptor *rtnStatusMsg)
{
   int status = BON_SUCCESS;

   // Determine the verb of the incoming business object
   char *verb = theObj.getVerb();

   if (strcmp(verb, CREATE) == 0)
```

```
   {
      if (!theObj->initAndValidateAttributes())
         return BON_FAIL;
   }
else ...
```

isBlank()

指定された名前を持つ属性または属性リスト内の指定された位置にある属性の値が特殊 Blank 値かどうかを判別します。

構文

```
unsigned char isBlank(char * name);
unsigned char isBlank(int position);
```

パラメーター

name **[in]** 属性の名前。

position **[in]** 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 (序数) を指定する整数。

戻り値

属性値がブランク値と等しい場合は TRUE、等しくない場合は FALSE を戻します。

関連項目

getBlankValue() isBlankValue() isIgnore()

isBlankValue()

指定された属性値が特殊 Blank 属性値かどうかを判別します。

構文

```
unsigned char isBlankValue(char * value);
```

パラメーター

value **[in]** 特殊な Blank 値と比較する属性値。

戻り値

渡された値がブランク値と等しい場合は TRUE、等しくない場合は FALSE を戻します。

関連項目

getBlankValue() isBlank() isIgnoreValue()

isIgnore()

指定された名前を持つ属性または属性リスト内の指定された位置にある属性の値が特殊 Ignore 値かどうかを判別します。

構文

```
unsigned char isIgnore(char * name);
unsigned char isIgnore(int position);
```

パラメーター

name **[in]** 属性の名前。

position **[in]** 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 (序数) を指定する整数。

戻り値

属性値が特殊 Ignore 値と等しい場合は True を表す 1、等しくない場合は False を表す 0 を戻します。

関連項目

getIgnoreValue() isBlank() isIgnoreValue()

isIgnoreValue()

指定された属性値が特殊 Ignore 値かどうかを判別します。

構文

```
unsigned char isIgnoreValue(char * value);
```

パラメーター

value **[in]** 特殊な Ignore 値と比較する属性値。メソッドは特殊 Ignore 値を予期するので、この引き数は `char *` として定義されます。

戻り値

渡された値が特殊 Ignore 値と等しい場合は True を表す 1、等しくない場合は False を表す 0 を戻します。

注意事項

`isIgnoreValue()` が検査する Ignore 値は、コネクターが無視する値を示す特殊属性値です。属性の値を `getAttrValue()` で検索した後、戻された値を `isIgnoreValue()` に渡して、値が Ignore 値かどうかを判別することができます。

関連項目

getIgnoreValue() isBlankValue() isIgnore()

makeNewAttrObject()

指定された名前を持つ属性または属性リスト内の指定された位置にある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。通常、このメソッドは、子オブジェクトを含む属性で使用されます。

構文

```
void * makeNewAttrObject(char * name);
void * makeNewAttrObject(int position);
```

パラメーター

name **[in]** 属性の名前。

*pos ition***[in]** 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 (序数) を指定する整数。

戻り値

BusinessObject クラスの新規作成インスタンスへのポインター。

注意事項

このメソッドは、属性タイプ OBJECT の属性に対してのみ使用してください。このメソッドは、属性に対して適切な型の新規ビジネス・オブジェクトを作成しますが、既存の属性の変更はしません。新規ビジネス・オブジェクトの値を設定するには、setAttrValue() を使用します。

setAttrValue()

属性の値を設定します。

構文

```
unsigned char setAttrValue(char * name, void * newval, int type);
unsigned char setAttrValue(int position, void * newVal, int type);
```

パラメーター

name **[in]** 設定対象の値を持った属性の名前。

*position***[in]** 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 (序数) を指定する整数。

newVal **[in]** 属性の値のストリング表記または値へのポインターです。char * または BusinessObject * である必要があります。

type **[in]** 新しい値に対する以下の属性データ型のいずれかであり、次のいずれかの定数で表されます。

```
BOAttrType::OBJECT
BOAttrType::BOOLEAN
BOAttrType::INTEGER
BOAttrType::FLOAT
```

```
BOAttrType::DOUBLE
BOAttrType::STRING
BOAttrType::DATE
BOAttrType::LONGTEXT
```

戻り値

操作が成功した場合は True、失敗した場合は False を戻します。

注意事項

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。これらのメソッドは、属性値を変更する前に、新しい値のデータ型が正しいかどうかを検査します。newval が文字ポインターである場合、メソッドは値を設定します。type が OBJECT であり、かつ属性が単一カーディナリティー・オブジェクトを参照する場合、setAttrValue() メソッドは前のビジネス・オブジェクトを新規ビジネス・オブジェクトで上書きします。type が OBJECT であり、かつ属性が複数カーディナリティー・オブジェクトを参照する場合、setAttrValue() メソッドはビジネス・オブジェクトをコンテナーに付加します。

属性を特殊値 BusinessObject::BlankValue または BusinessObject::IgnoreValue に設定することもできます。Blank は「このフィールドを消去します。フィールド内にデータはありません」ということを意味し、Ignore は「このフィールドの内容を認識または考慮しません」ということを意味します。

例

```
unsigned char status;
if (status = pObj->setAttrValue("Interest Rate","0.065",
      BOAttrType::FLOAT);) == 0)
   // continue
```

関連項目

getAttrValue() setDefaultAttrValues()

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。

構文

```
void setDefaultAttrValues();
```

パラメーター

なし。

戻り値

なし。

関連項目

getDefaultAttrValue() setAttrValue()

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。

構文

```
void setLocale(const char * 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。ロケール名の形式については、 64 ページの『ロケールとは』を参照してください。

戻り値

なし。

注意事項

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。

関連項目

getLocale()

setVerb()

現在のビジネス・オブジェクトのアクティブな動詞を設定します。

構文

```
void setVerb(char * newVerb);
```

パラメーター

newVerb **[in]** 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。

戻り値

なし。

注意事項

ビジネス・オブジェクト定義 (BusObjSpec インスタンス) は、ビジネス・オブジェクトがサポートする動詞のリストを含んでいます。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。ビジネス・オブジェクト用の動詞は、一度に 1 つのみアクティブになります。

通常、ビジネス・オブジェクトは、Create、Retrieve、および Update 動詞をサポートします。ビジネス・オブジェクトによっては、それ以外に Delete などの動詞をサポートしているものもあります。ビジネス・オブジェクトをサポートする各コネクターは、ビジネス・オブジェクトがサポートする動詞をすべてインプリメントする必要があります。

例

```
BusinessObject *pObj;
...
pObj = new BusinessObject("Customer");
pObj->setVerb("Create");
```

関連項目

doVerbFor(), getVerb()

第 13 章 BusObjContainer クラス

BusObjContainer クラスは、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。このクラスは、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。BusObjContainer インスタンスはそれぞれコンテナー・オブジェクトとなります。これには、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としてのビジネス・オブジェクトを挿入することができます。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: 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。

このクラスのヘッダー・ファイルは BusObjContainer.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

表 101 に、BusObjContainer クラスのメソッドを要約します。

表 101. BusObjContainer 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
BusObjContainer()	これは、ビジネス・オブジェクト配列 (コンテナー) を作成します。このメソッドを使用してビジネス・オブジェクト配列を作成することはしないでください。IBM WebSphere Business Integration システムは、子ビジネス・オブジェクトを参照する各属性に対し、必要に応じてビジネス・オブジェクト配列を作成します。	
getObject()	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・オブジェクトを検索します。	294
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検索します。	294
getTheSpec()	ビジネス・オブジェクト配列のビジネス・オブジェクト定義を検索します。	295
insertObject()	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。	295
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。	296
removeObjectAt()	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。	296
setObject()	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。	297

getObject()

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・オブジェクトを検索します。

構文

```
BusinessObject * getObject(int index);
```

パラメーター

index **[in]** 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・オブジェクトへのポインター、またはビジネス・オブジェクト配列内の指定された位置にビジネス・オブジェクトがない場合は NULL。

注意事項

setObject() メソッドを使用すると、ビジネス・オブジェクト配列内でビジネス・オブジェクトの位置を指定することができます。

関連項目

setObject() 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検索します。

構文

```
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。

注意事項

insertObject() 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。

例

```
// iterate through objects in a BusObjContainer
for(int i = 0; i < myCont->getObjectCount(); i++)
   BusinessObject *myObj = myCont->getObject(i);
   if(myObj != NULL)
      myObj->doVerbFor();
}
```

関連項目

insertObject() メソッドの説明も参照してください。

getTheSpec()

ビジネス・オブジェクト配列のビジネス・オブジェクト定義を検索します。

構文

```
BusObjSpec * getTheSpec() const;
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義へのポインター。

注意事項

BusObjContainer オブジェクトに格納されている子ビジネス・オブジェクトはすべて、ビジネス・オブジェクト配列と同じビジネス・オブジェクト定義を持っている必要があります。

関連項目

BusObjSpec() クラスの説明も参照してください。

insertObject()

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

```
void insertObject(BusinessObject * busObj);
```

パラメーター

busObj **[in]** 子ビジネス・オブジェクトへのポインターです。

戻り値

なし。

注意事項

子ビジネス・オブジェクトをパラメーターとして渡すと、`BusinessObject::SetAttrValue()` はこのメソッドを呼び出します。

関連項目

`setObject()` 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

```
void removeAllObjects();
```

パラメーター

なし。

戻り値

なし。

removeObjectAt()

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

```
int removeObjectAt(int index);
```

パラメーター

index **[in]** 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

失敗した場合は -1、成功した場合はゼロを戻します。

注意事項

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。除去されたビジネス・オブジェクトよりも高い指数を持つビジネス・オブジェクトすべてについて、指標が減分されます。

setObject()

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

```
BusinessObject * setObject(int index, BusinessObject * busObj);
```

パラメーター

index **[in]** 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。

busObj **[in]** 子ビジネス・オブジェクトへのポインターです。

戻り値

子ビジネス・オブジェクトへのポインター。

注意事項

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。古いビジネス・オブジェクトは削除されます。

関連項目

getObject() メソッドの説明も参照してください。

第 14 章 BusObjSpec クラス

BusObjSpec クラスの各インスタンスは、ビジネス・オブジェクトの内容、形式、および振る舞いを定義します。これが、ビジネス・オブジェクト定義です。複数のビジネス・オブジェクトが同じビジネス・オブジェクト定義を参照することもできます。 BusObjSpec クラスの各インスタンスは、それぞれ異なるビジネス・オブジェクトを、属性の値のみを変えて記述します。名前は同じだがバージョンが異なるビジネス・オブジェクト定義は、BusObjSpec クラスで別個のインスタンスになります。

コネクターは、ビジネス・オブジェクト定義のセットをサポートしています。リポジトリーにある各コネクター構成情報に、コネクターがサポートするビジネス・オブジェクト定義が示されています。

WebSphere InterChange Server

すべての動詞または個別の動詞についてのイベント通知用に、コラボレーションがビジネス・オブジェクト定義にサブスクライブすることもできます。ビジネス・オブジェクト定義には、サポートされる動詞のリストも含まれています。

各ビジネス・オブジェクト定義は、ビジネス・オブジェクトの動詞のタスクを実行するビジネス・オブジェクト・ハンドラーへの参照を含んでいます。

このクラスのヘッダー・ファイルは BusObjSpec.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

BusObjSpec クラスには、ビジネス・オブジェクト属性に関する情報を検索するメソッドがあります。表 102 に、BusObjSpec クラスのメソッドを要約します。

表 102. BusObjSpec 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
BusObjSpec()	コンストラクターを呼び出してリポジトリー内でビジネス・オブジェクト定義を作成することはしないで ください。ビジネス・オブジェクト定義を作成する場合は、Business Object Designer を使用してください。	
getAppText()	ビジネス・オブジェクト定義に関するアプリケーション固有の情報を検索します。	300
getAttribute()	ビジネス・オブジェクト属性を名前または位置で検索します。	300
getAttributeCount()	ビジネス・オブジェクト定義の属性リストにある属性の数を検索します。	301
getAttributeIndex()	属性リスト内でのビジネス・オブジェクト属性の位置を検索します。	301
getMyBOHandler()	ビジネス・オブジェクト定義が参照するビジネス・オブジェクト・ハンドラーを検索します。	302

© Copyright IBM Corp. 1997, 2004

表 102. *BusObjSpec* 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
getName()	ビジネス・オブジェクトまたはビジネス・オブジェクト定義の名前を検索します。	302
getVerbAppText()	動詞に関するアプリケーション固有の情報を検索します。	303
getVersion()	ビジネス・オブジェクト定義のバージョンを検索します。	303
isVerbSupported()	ビジネス・オブジェクト定義が特定の動詞をサポートしているかどうかを判別します。	304

getAppText()

ビジネス・オブジェクト定義に関するアプリケーション固有の情報を検索します。

構文

```
char * getAppText() const;
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト定義に関するアプリケーション固有の情報を含む文字ストリング。このメソッドは、NULL を戻す場合もあります。

注意事項

ビジネス・オブジェクト定義に AppSpecificInfo プロパティーの値がある場合、getAppText() メソッドはこの値を検索します。属性がそれぞれ独自にアプリケーション固有の情報を持つこともできます。

getAttribute()

属性の名前またはビジネス・オブジェクト定義の属性リスト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属性記述子を検索します。

構文

```
BOAttrType * getAttribute(char * name);
BOAttrType * getAttribute(int position);
```

パラメーター

name **[in]** 属性名。

position **[in]** 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 (序数) を指定する整数。

戻り値

BOAttrType クラスのインスタンスへのポインター。無効な属性名または位置が指定されると、これらのメソッドは NULL を戻します。

注意事項

getAttribute() メソッドは、ビジネス・オブジェクト定義 (BusObjSpec インスタンス) 内の属性について、BOAttrType クラスのインスタンスである属性記述子を戻します。BOAttrType クラスは、アプリケーション固有の情報、デフォルト値、およびそれが必要かどうかなど、属性プロパティーについての情報を取得するメソッドを提供します。

関連項目

BOAttrType クラスのメソッドについては、249 ページの『第 10 章 BOAttrType クラス』を参照してください。

getAttributeCount()

ビジネス・オブジェクト定義の属性リストにある属性の数を検索します。

構文

```
int getAttributeCount();
```

パラメーター

なし。

戻り値

属性リスト内にある属性の数を指定する整数。

注意事項

属性リスト内の属性の数は、ビジネス・オブジェクト定義がサポートする属性の数です。

関連項目

BOAttrType クラスの説明も参照してください。

getAttributeIndex()

属性リスト内でのビジネス・オブジェクト属性の位置を検索します。

構文

```
int getAttributeIndex(char * name);
```

パラメーター

name **[in]**　　属性の名前です。

戻り値

ビジネス・オブジェクト定義の属性リストにある属性の位置を示す整数。

注意事項

属性リスト内の属性の位置を検索したら、その位置を使用して属性を参照することができます。

関連項目

BOAttrType クラスの説明も参照してください。

---

getMyBOHandler()

ビジネス・オブジェクト定義が参照するビジネス・オブジェクト・ハンドラーを検索します。

構文

```
BOHandlerCPP * getMyBOHandler() const;
```

パラメーター

なし。

戻り値

ビジネス・オブジェクト・ハンドラーへのポインター。

注意事項

複数のビジネス・オブジェクト定義が同じビジネス・オブジェクト・ハンドラーを使用することもできます。

このメソッドは、BOHandlerCPP クラスから派生し、GenGlobals クラスの getBOHandlerforBO() メソッドへの呼び出しで設定したハンドラーを戻します。

関連項目

BOHandlerCPP クラスの説明も参照してください。

---

getName()

ビジネス・オブジェクト定義の名前を検索します。

構文

```
char * getName() const;
```

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文字ストリング。

関連項目

BusinessObject クラスの説明も参照してください。

getVerbAppText()

ビジネス・オブジェクトの動詞に関するアプリケーション固有の情報を検索します。

構文

```
char * getVerbAppText(char * verb);
```

パラメーター

verb **[in]** 動詞の名前。この動詞のアプリケーション固有情報が検索されます。

戻り値

ビジネス・オブジェクトの動詞に関するアプリケーション固有の情報を含む文字ストリング。

注意事項

ビジネス・オブジェクトの動詞に AppSpecificInfo プロパティーの値がある場合、getVerbAppText() メソッドはこの値を検索します。

getVersion()

ビジネス・オブジェクト定義のバージョンを検索します。

構文

```
CxVersion * getVersion();
```

パラメーター

なし。

戻り値

CxVersion クラスのインスタンスへのポインター。

注意事項

ビジネス・オブジェクト定義は、CxVersion クラスのインスタンスを参照します。このクラスの各インスタンスには、バージョン番号、サブバージョン番号、およびポイント・バージョン番号が、3.0.0 のようにピリオドで区切られて含まれています。

関連項目

CxVersion クラスの説明も参照してください。

---

isVerbSupported()

ビジネス・オブジェクト定義が特定の動詞をサポートしているかどうかを判別します。

構文

```
unsigned char isVerbSupported(char * verb);
```

パラメーター

verb **[in]**
: 現在のビジネス・オブジェクト定義がサポートしているかどうかをメソッドが判別する動詞の名前です。

戻り値

ビジネス・オブジェクト定義がその動詞をサポートしている場合は TRUE、サポートしていない場合は FALSE を戻します。

第 15 章 CxMsgFormat クラス

CxMsgFormat クラスは、さまざまな言語でメッセージを生成するためのメッセージ・タイプ定数を提供します。このクラスのヘッダー・ファイルは CxMsgFormat.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

このクラスは、以下のものを提供します。

- 『メッセージ・タイプ定数』
- 『メソッド』

メッセージ・タイプ定数

CxMsgFormat クラスは、表 103 に示されているメッセージ・タイプ定数を定義します。

表 103. CxMsgFormat クラスで定義されたメッセージ・タイプ定数

メッセージ・タイプ定数	意味
XRD_WARNING	警告メッセージ
XRD_TRACE	トレース・メッセージ
XRD_INFO	通知メッセージ
XRD_ERROR	エラー・メッセージ
XRD_FATAL	致命エラー・メッセージ

メソッド

表 104 に、CxMsgFormat クラスのメソッドの要約を示します。

表 104. CxMsgFormat 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
CxMsgFormat()	CxMsgFormat クラスのインスタンスを作成します。この関数を呼び出してメッセージ・オブジェクトを作成することはしないでください。メッセージ・オブジェクトは IBM WebSphere Business Integration システムによって自動的に作成されます。	
generateMsg()	メッセージを生成します。	305

重要: CxMsgFormat クラスのメソッドは使用すべきでありません。詳細については、307 ページの『使用すべきでないメソッド』を参照してください。

generateMsg()

メッセージ・ファイルの事前定義メッセージでメッセージを生成します。

© Copyright IBM Corp. 1997, 2004

構文

```
char * generateMsg(int msgNum, int msgType, char * info,
   int argCount, va_list v1);
```

パラメーター

msgNum **[in]**
: メッセージ・ファイル内のメッセージ番号を指定する整数です。

msgType **[in]**
: 次のメッセージ・タイプのいずれかにします。

```
XRD_UNKNOWN
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]**
: IBM WebSphere Business Integration システム がメッセージを生成したクラスの名前など、情報を示す値です。

argCount **[in]**
: メッセージ・テキスト (オプション) 内のパラメーターの個数を示す整数です。

v1 **[in]**
: メッセージ・テキストの各パラメーターをコンマで区切ったオプションのリストです。各パラメーターは char * 値です。

戻り値

生成されたメッセージへのポインター。

注意事項

generateMsg() メソッドを使用すると、さまざまな言語でメッセージを生成できます。アプリケーションがサポートする言語ごとに、その言語のメッセージ用のメッセージ・ファイルを個別に作成できます。

XRD_TRACE メッセージ・タイプを使用して、トレース・メッセージを生成することができます。

重要: このクラスの generateMsg() メソッドは使用すべきでありません。代わりに、GenGlobals クラスおよび BOHandlerCPP クラスの generateMsg() メソッドを使用してください。

例

```
generateMsg(3160, XRD_ERROR, "Logon ID is invalid.", 0);
```

関連項目

BOHandlerCPP クラスおよび GenGlobals クラスの generateMsg() メソッドの説明も参照してください。

使用すべきでないメソッド

CxMsgFormat クラスのメソッドは、以前のバージョンではサポートされていましたが、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マイグレーション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 105 に、CxMsgFormat クラスで使用すべきでないメソッドを示します。(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 105. CxMsgFormat クラスで使用すべきでないメソッド

以前のメソッド	置換
generateMsg()	GenGlobals クラスの generateMsg() BOHandlerCPP クラスの generateMsg()

第 16 章 CxVersion クラス

CxVersion クラスは、ビジネス・オブジェクト・バージョンを表します。このクラスのメソッドを使用すると、バージョン情報を設定および検索できます。このクラスのヘッダー・ファイルは CxVersion.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

DevelopmentKits¥cdk¥generic_include

表 106 に、CxVersion クラスのメソッドの要約を示します。

表 106. CxVersion 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
CxVersion()	バージョン・オブジェクトを作成します。	309
compareMajor()	メジャー・バージョンをオブジェクトまたは整数と比較します。	310
compareMinor()	マイナー・バージョンをオブジェクトまたは整数と比較します。	311
comparePoint()	ポイント・バージョンをオブジェクトまたは整数と比較します。	311
compareTo()	バージョンをオブジェクトまたは整数と比較します。	312
getDELIMITER()	バージョン・オブジェクト内でメジャー・バージョン、マイナー・バージョン、およびポイント・バージョンを区切る区切り文字を検索します。	312
getLATESTVERSION()	最新のバージョン・オブジェクトを検索します。	313
getMajorVer()	メジャー・バージョンを検索します。	313
getMinorVer()	マイナー・バージョンを検索します。	314
getPointVer()	ポイント・バージョンを検索します。	314
setMajorVer()	メジャー・バージョンを設定します。	315
setMinorVer()	マイナー・バージョンを設定します。	315
setPointVer()	バージョン・オブジェクトのポイント・バージョンを設定します。	315
toString()	バージョン・オブジェクトまたは整数を文字ストリングに変換します。	316

CxVersion()

メジャー番号、マイナー番号、およびポイント番号、またはバージョン・ストリングからバージョン・オブジェクトを作成します。

構文

```
CxVersion(int major, int minor, int point);
CxVersion(char * verString);
```

パラメーター

major **[in]** メジャー・バージョンを指定する整数です。これは、バージョン内の最初の区切り文字の前に置かれます。

© Copyright IBM Corp. 1997, 2004

minor **[in]** マイナー・バージョンを指定する整数です。これは、バージョン内の最初の区切り文字の後に置かれます。

point **[in]** ポイント・バージョンを指定する整数です。これは、バージョン内の 2 番目の区切り文字の後に置かれます。

verString **[in]** 「2.0.3」のように、バージョンを指定する文字ストリングです。

戻り値

なし。

注意事項

各ビジネス・オブジェクト定義は、CxVersion クラスのインスタンスを参照してバージョンを設定します。 BusinessObject コンストラクターが参照するバージョンは、ビジネス・オブジェクト定義のバージョンです。ビジネス・オブジェクトのバージョンは、ビジネス・オブジェクト定義のバージョンと同じです。

例

```
CxVersion * CurrentVersion =
   new CxVersion(LATEST, LATEST, LATEST);
. . .
myVersion = new CxVersion("2.0.3");
```

関連項目

BusObjSpec クラスの説明も参照してください。

---

compareMajor()

メジャー・バージョン同士を比較するか、またはバージョン・オブジェクトのメジャー・バージョンを整数と比較します。

構文

```
int compareMajor(CxVersion & other);
int compareMajor(int major);
```

パラメーター

other **[in]** バージョン・オブジェクトです。

major **[in]** メジャー・バージョン番号を表す整数です。

戻り値

メジャー・バージョン間の差を示す整数。

注意事項

compareMajor() メソッドを使用すると、ビジネス・オブジェクトまたはビジネス・オブジェクト定義のメジャー・バージョンを比較できます。

関連項目

compareMinor() メソッド、comparePoint() メソッド、および compareTo() メソッドの説明も参照してください。

compareMinor()

2 つのバージョン・オブジェクトのマイナー・バージョンを比較するか、またはマイナー・バージョンを整数と比較します。

構文

```
int compareMinor(CxVersion & other);
int compareMinor(int minor);
```

パラメーター

other **[in]**　　バージョン・オブジェクトです。

minor **[in]**　　マイナー・バージョン番号を表す整数です。

戻り値

マイナー・バージョン間の差を示す整数。

注意事項

compareMinor() メソッドを使用すると、ビジネス・オブジェクトまたはビジネス・オブジェクト定義のマイナー・バージョンを比較できます。

関連項目

compareMajor() メソッド、comparePoint() メソッド、および compareTo() メソッドの説明も参照してください。

comparePoint()

2 つのバージョン・オブジェクトのポイント・バージョンを比較するか、またはポイント・バージョンを整数と比較します。

構文

```
int comparePoint(CxVersion & other);
int comparePoint(int point);
```

パラメーター

other **[in]**　　バージョン・オブジェクトです。

point **[in]**　　整数です。

戻り値

ポイント・バージョン間の差を示す整数。

注意事項

comparePoint() メソッドを使用すると、ビジネス・オブジェクトまたはビジネス・オブジェクト定義のポイント・バージョンを比較できます。

関連項目

compareMajor() メソッド、compareMinor() メソッド、および compareTo() メソッドの説明も参照してください。

compareTo()

現行バージョンを別のバージョンと比較するか、または現行バージョンを文字ストリングと比較します。

構文

```
int compareTo(CxVersion & other);
int compareTo(char * verString);
```

パラメーター

other **[in]** バージョン・オブジェクトです。

verString **[in]** バージョンを指定する文字ストリングです。

戻り値

バージョンが一致する場合は 0、一致しない場合は、メジャー・バージョン、マイナー・バージョン、ポイント・バージョンのうちの最初の差を示す整数です。

注意事項

compareTo() メソッドを使用すると、ビジネス・オブジェクトまたはビジネス・オブジェクト定義に、ある特定のバージョンがあるかどうかを判別できます。

例

```
compareTo(newVersion);
```

関連項目

compareMajor() メソッド、compareMinor() メソッド、および compareTo() メソッドの説明も参照してください。

getDELIMITER()

バージョン・オブジェクト内でメジャー・バージョン、マイナー・バージョン、およびポイント・バージョンを区切る区切り文字を検索します。

構文

```
static char * getDELIMITER();
```

パラメーター

なし。

戻り値

バージョン・オブジェクト内でメジャー・バージョン、マイナー・バージョン、およびポイント・バージョンを区切る現在の区切り文字。

注意事項

getDELIMITER() メソッドを使用すると、バージョン・ストリングに含まれている文字を判別できます。

関連項目

getLATESTVERSION() メソッドおよび toString() メソッドの説明も参照してください。

getLATESTVERSION()

バージョン・オブジェクトの最新バージョンを検索します。

構文

```
static const CxVersion &getLATESTVERSION();
```

パラメーター

なし。

戻り値

メジャー・バージョン、マイナー・バージョン、ポイント・バージョンが現在の区切り文字 (デフォルトではピリオド) で区切られた最新バージョンです。

注意事項

compareTo() メソッドを使用すると、ビジネス・オブジェクトまたはビジネス・オブジェクト定義に最新バージョンがあるかどうかを判別できます。

例

```
if(pObj->getVersion()->compareTo(CxVersion::getLATESTVERSION)==0)
```

関連項目

getMajorVer() メソッド、getMinorVer() メソッド、および getPointVer() メソッドの説明も参照してください。

getMajorVer()

バージョン・オブジェクトのメジャー・バージョンを検索します。

構文

```
int getMajorVer() const;
```

パラメーター

なし。

戻り値

最新バージョン・オブジェクト内のメジャー・バージョンを示す整数です。

関連項目

getLATESTVERSION() メソッド、getMinorVer() メソッド、および getPointVer() メソッドの説明も参照してください。

getMinorVer()

バージョン・オブジェクトのマイナー・バージョンを検索します。

構文

```
int getMinorVer() const;
```

パラメーター

なし。

戻り値

最新バージョン内のマイナー・バージョンを示す整数です。

関連項目

getLATESTVERSION() メソッド、getMajorVer() メソッド、および getPointVer() メソッドの説明も参照してください。

getPointVer()

バージョン・オブジェクトのポイント・バージョンを検索します。

構文

```
int getPointVer() const;
```

パラメーター

なし。

戻り値

最新バージョン内のポイント・バージョンを示す整数です。

関連項目

getLATESTVERSION() メソッド、getMajorVer() メソッド、および getMinorVer() メソッドの説明も参照してください。

setMajorVer()

バージョン・オブジェクト内のメジャー・バージョンを設定します。

構文

```
void setMajorVer(int newMajorVer);
```

パラメーター

newMajorVer **[in]**
メジャー・バージョンを指定する整数です。

戻り値

なし。

関連項目

setMinorVer() メソッドおよび setPointVer() メソッドの説明も参照してください。

setMinorVer()

バージョン・オブジェクトのマイナー・バージョンを設定します。

構文

```
void setMinorVer(int newMinorVer);
```

パラメーター

newMinorVer **[in]**
マイナー・バージョンを指定する整数です。

戻り値

なし。

関連項目

setMajorVer() メソッドおよび setPointVer() メソッドの説明も参照してください。

setPointVer()

バージョン・オブジェクトのポイント・バージョンを設定します。

構文

```
void setPointVer(int newPointVer);
```

例

newPointVer **[in]**
: ポイント・バージョンを指定する整数です。

戻り値

なし。

関連項目

setMajorVer() メソッドおよび setMinorVer() メソッドの説明も参照してください。

---

toString()

バージョン・オブジェクトを文字ストリングに変換するか、またはメジャー・バージョン、マイナー・バージョン、ポイント・バージョンを文字ストリングに変換します。

構文

```
char * toString();
static char * toString(int major, int minor, int point);
```

パラメーター

major **[in]**
: メジャー・バージョンを指定する整数です。これは、バージョン内の最初の区切り文字の前に置かれます。

minor **[in]**
: マイナー・バージョンを指定する整数です。これは、バージョン内の最初の区切り文字の後に置かれます。

point **[in]**
: ポイント・バージョンを指定する整数です。これは、バージョン内の 2 番目の区切り文字の後に置かれます。

戻り値

指定したバージョン・オブジェクトまたはメジャー番号、マイナー番号、およびポイント番号から、メジャー・バージョン番号、マイナー・バージョン番号、およびポイント・バージョン番号を連結する文字ストリング。区切り文字は、文字ストリング内でメジャー番号、マイナー番号、ポイント番号を区切ります。

関連項目

CxVersion コンストラクターの説明も参照してください。

第 17 章 GenGlobals クラス

GenGlobals クラスは C++ 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・クラスを導出して、コネクター用の仮想メソッドを実装する必要があります。このコネクター・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。

重要: すべての C++ コネクターはこのコネクター基底クラスを拡張し*なければなりません*。この基底クラスには次の仮想メソッドが含まれています。init()、getVersion()、 getBOHandlerforBO()、 pollForEvents()、および terminate()。デベロッパーは、その派生したコネクター・クラスで、上記の仮想メソッドに対してインプリメントを提供し*なければなりません*。

このクラスに対するヘッダー・ファイルは GenGlobals.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

DevelopmentKits¥cdk¥generic_include

表 107 に、GenGlobals クラス内のメソッドを要約します。

表 107. GenGlobals 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
GenGlobals()	GenGlobals クラスのインスタンスを作成します。	318
executeCollaboration()	ビジネス・オブジェクト要求をコラボレーションに送信します。	318
generateAndLogMsg()	メッセージ・ファイルからメッセージを生成し、コネクターのログ宛先に送信します。	319
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、コネクターのログ宛先に送信します。	321
generateMsg()	ユーザーが指定するメッセージ・ファイルからメッセージを生成します。	322
getBOHandlerforBO()	ビジネス・オブジェクトのハンドラーを検索します。	323
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検索します。	324
getConfigProp()	リポジトリーから、コネクターのプロパティーを検索します。	324
getTheSubHandler()	サブスクリプション・ハンドラーを検索して、着信ビジネス・オブジェクトのビジネス・オブジェクト定義にサブスクライブするコラボレーションを判別します。	327
getVersion()	コネクター・フレームワークのアプリケーション固有の情報のコンポーネントのバージョンを検索します。	327
init()	アプリケーションとの接続を確立します。	328
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。	329
logMsg()	メッセージをログに記録します。	330
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。	331
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。	332
traceWrite()	トレース・メッセージを書き込みます。	333

© Copyright IBM Corp. 1997, 2004

GenGlobals()

GenGlobals クラスのインスタンスを作成します。 GenGlobals 基底クラスからコネクター・クラスを派生させ、GenGlobals クラスにあるすべての仮想メソッドをインプリメントします。

構文

```
GenGlobals();
```

パラメーター

なし。

戻り値

なし。

例

```
XYZGlobal::XYZGlobal() : GenGlobals()
```

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

```
void executeCollaboration (char * busProcName, BusinessObject & busObj,
   ReturnStatusDescriptor & rtnStatusDesc);
```

パラメーター

busProcName[in]
: 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。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を判別するには、getCollabNames() メソッドを使用します。

busObj[in/out]
: 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。

retStatusDesc[out]
: ビジネス・プロセスからのメッセージおよび状況が含まれている、戻り状況記述子です。

戻り値

なし。

注意事項

executeCollaboration() メソッドは、コネクター・フレームワークに *busObj* 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、データを直列化して適切に永続化するために、イベント・オブジェクトに対して処理を実行します。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、ReturnStatusDescriptor 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。

関連項目

BusinessObject および ReturnStatusDescriptor クラスの説明も参照してください。

generateAndLogMsg()

メッセージを生成し、コネクターのログ宛先に送信します。

構文

```
void generateAndLogMsg(int msgNum, int msgType, int argCount, ...);
```

パラメーター

msgNum **[in]** 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 **[in]** CxMsgFormat クラスに定義されている、以下のいずれかのメッセージ・タイプ定数です。この定数により、メッセージの重大度が識別されます。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

argCount **[in]** 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。

... **[in]** メッセージ・テキスト用のメッセージ・パラメーターのリストです。

戻り値

なし。

注意事項

generateAndLogMsg() メソッドは、generateMsg() メソッドと logMsg() メソッドの機能を結合したメソッドです。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。

重要: これら 2 つのメソッドを結合することによって、generateAndLogMsg() は、generateMsg() が生成するメッセージ・ストリングに必要なメモリーを解放します。メッセージ・ストリングに割り振るメモリーを解放するために freeMemory() メソッドを呼び出す必要はなくなります。

WebSphere InterChange Server

severity は 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、E メール通知がオンのときに E 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。

IBM では、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() でログに記録されたコネクター・メッセージは、LogViewer を使用して表示することができます。

例

以下の例は、generateMsg() メソッドの場合と同じタスクを実行します。

```
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateAndLogMsg(1100, CxMsgFormat::XRD_ERROR, 0, NULL);
   return BON_FAIL;
}
```

関連項目

generateAndTraceMsg(), generateMsg(), logMsg()

generateAndTraceMsg()

トレース・メッセージを生成し、コネクターのトレース宛先に送信します。

構文

```
void generateAndTraceMsg(int msgNum, int msgType, int traceLevel,
   int argCount, ...);
```

パラメーター

msgNum **[in]**
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 **[in]**
`CxMsgFormat` クラスで定義された以下のメッセージ・タイプ定数のいずれかです。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

traceLevel **[in]**
`Tracing` クラスに定義されている、以下のいずれかのトレース・レベル定数です。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。

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::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、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。そのため、`LEVEL0` の *`traceLevel`* に関連付けられたトレース・メッセージはいずれも印刷されません。

argCount **[in]**
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。

... **[in]**
メッセージ・テキスト用のメッセージ・パラメーターのリストです。

戻り値

なし。

注意事項

`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。

重要: これら 2 つのメソッドを結合することによって、generateAndTraceMsg() は、generateMsg() が生成するメッセージ・ストリングに必要なメモリーを解放します。メッセージ・ストリングに割り振るメモリーを解放するために freeMemory() メソッドを呼び出す必要はなくなります。

generateAndTraceMsg() でログに記録されたコネクター・メッセージは、LogViewer を使用して表示することができません。

例

```
if(tracePtr->getTraceLevel()>= Tracing::LEVEL3) {
   // Message 3033 - Opened main form for object
   msg = generateAndTraceMsg(3033,CxMsgFormat::XRD_FATAL,
      Tracing::LEVEL3,0, NULL);
}
```

関連項目

generateAndLogMsg(), generateMsg(), traceWrite()

generateMsg()

メッセージ・ファイル内の定義済みメッセージのセットからメッセージを生成します。

構文

```
char * generateMsg(int msgNum, int msgType, char * info,
   int argCount, ...);
```

パラメーター

msgNum **[in]** 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 **[in]** CxMsgFormat クラスで定義された以下のメッセージ・タイプ定数のいずれかです。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

info **[in]** IBM WebSphere Business Integration システム がメッセージを生成したクラスの名前など、情報を示す値です。

argCount **[in]** 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。

... **[in]** メッセージ・テキストに対するパラメーターのリストです。

戻り値

生成したメッセージへの文字ポインターです。

注意事項

generateMsg() メソッドは、生成されたメッセージを格納するためのメモリーを割り振ります。コネクターはメッセージを記録すると、割り振られたメモリーを解放するため freeMemory() メソッドを呼び出します。このメソッドは、コネクター・フレームワーク・クラス JToCPPVeneer のメンバーです。呼び出しの構文は次のとおりです。

```
void freeMemory(char * mem)
```

ここで、mem は、generateMsg() によって割り振られたメモリーです。このメソッドの呼び出し方法の例は、以下のサンプル・コードを参照してください。

例

```
char * msg;
ret_code = connect_to_app(userName, password);
// Message 1100 - Failed to connect to application
if (ret_code == -1) {
   msg = generateMsg(1100, CxMsgFormat::XRD_ERROR, NULL, 0, NULL);
   logMsg(msg);
   JToCPPVeneer::getTheHandlerStuff()->freeMemory(msg);
   return BON_FAIL;
}
```

---

getBOHandlerforBO()

ビジネス・オブジェクト定義のビジネス・オブジェクト・ハンドラーを検索します。

構文

```
virtual BOHandlerCPP * getBOHandlerforBO(char * busObjName) = 0;
```

パラメーター

busObjName **[in]**
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・ハンドラーへのポインターです。

注意事項

クラス・ライブラリーは getBOHandlerforBO() メソッドを呼び出して、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・ハンドラーを検索します。

重要: getBOHandlerforBO() メソッドは、コネクターに対してインプリメントする必要がある 仮想メソッドです。

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。

例

```
BOHandlerCPP *AppGlobal::getBOHandlerforBO(char * BOName)
{
   static AppGlobal &pGlobal = NULL;
   if (NULL == pGlobal) {
      pGlobal = new AppGlobal();
   }
   return pGlobal;
}
```

関連項目

BOHandlerCPP クラスの説明も参照してください。

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名前を検索します。

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

```
StringMessage & getCollabNames();
```

戻り値

コラボレーション名のリストが含まれている StringMessage オブジェクト。

注意事項

getCollabNames() メソッドは、StringMessage オブジェクトにあるコラボレーション名を戻します。このクラスのメソッドを使用して、コラボレーション名にアクセスします。詳細については、 347 ページの『第 20 章 StringMessage クラス』を参照してください。

getConfigProp()

リポジトリーから、コネクター構成プロパティーを検索します。

構文

int getConfigProp(char * *property*, char * *val*, int *nMaxCount*);

パラメーター

property **[in]** 検索するプロパティーの名前です。

val **[out]** メソッドがプロパティー値を書き込むことができるバッファーへのポインターです。

nMaxCount **[in]**
値バッファー内のバイト数です。

戻り値

メソッドが値バッファーにコピーしたバイト数を示す整数です。

注意事項

並列処理コネクター (ParallelProcessDegree コネクター・プロパティーに 1 より大きい値がセットされているコネクター) において、getConfigProp("ConnectorName") を呼び出すと、マスター・プロセスまたはスレーブ・プロセスのいずれかが呼び出し元であるかに関係なく、このメソッドは、常時、コネクター・エージェント・マスター・プロセスの名前を戻します。

例

```
if (getConfigProp("LoginId", val, 255) == 0);
   {
   logMsg("Invalid LoginId");
   traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
   }
```

---

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検索します。

構文

```
char * 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を含むストリング。

注意事項

getEncoding() メソッドは、コネクター・フレームワークのロケールを検索します (ロケールは、言語、国 (または地域)、および文字エンコードに従って、データの国/地域別情報を定義したものです)。コネクター・フレームワークの文字エンコードは、コネクター・アプリケーションの文字エンコードを示している必要があります。コネクター・フレームワークのロケールは、次の階層を使用して設定されます。

- 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は、その中の CharacterEncoding コネクター構成プロパティーの設定が優先的に使用されます。ローカル構成ファイルが存在しない場合は、コネクター始動時に InterChange Server のリポジトリーからダウンロードされた一連のコネクター構成プロパティー内にある、CharacterEncoding プロパティーの設定が使用されます。

- 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で実行する場合に役に立ちます。

関連項目

getLocale() (このクラス内の)、getLocale() (BusinessObject クラス内の)

getLocale()

コネクター・フレームワークのロケールを検索します。

構文

```
char * 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を含むストリング。

注意事項

getLocale() メソッドは、コネクター・フレームワークのロケールを検索します (ロケールは、言語、国 (または地域)、および文字エンコードに従って、データの国/地域別情報を定義したものです)。コネクター・フレームワークのロケールは、コネクター・アプリケーションのロケールを示す必要があります。コネクター・フレームワークのロケールは、次の階層を使用して設定されます。

- 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は、その中の Locale コネクター構成プロパティーの設定が優先的に使用されます。ローカル構成ファイルが存在しない場合は、コネクター始動時に InterChange Server のリポジトリーからダウンロードされた一連のコネクター構成プロパティー内にある、Locale プロパティーの設定が使用されます。

- 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、ロケール依存処理をコネクターで実行する場合に役に立ちます。

関連項目

getEncoding()、getLocale() (BusinessObject クラス内の)

getTheSubHandler()

サブスクリプション・マネージャーへのポインターを検索します。呼び出しルーチンはこのポインターを使用して、ビジネス・オブジェクトに対して特定のビジネス・オブジェクト定義へのサブスクリプションが存在するかどうかを判別することができます。

構文

```
SubscriptionHandlerCPP * getTheSubHandler() const;
```

パラメーター

なし。

戻り値

サブスクリプション・マネージャーへのポインター。

注意事項

サブスクリプション・マネージャーを介して、コネクターはコネクターがパブリッシュする、それぞれのビジネス・オブジェクト定義のすべての動詞を対象に、すべてのアクティブなサブスクリプションの統合リストでサブスクライバーをトラッキングします。

例

```
if (getTheSubHandler->isSubscribed(theObj->getName(), "Create"){
}
```

関連項目

SubscriptionHandlerCPP、BusinessObject および BusObjSpec クラスの説明も参照してください。

getVersion()

Connector のバージョンを検索します。

構文

```
CxVersion * getVersion() = 0;
```

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文字ストリングへのポインター。

例

```
char * getVersion()

{
return (char *) CX_CONNECTOR_VERSIONSTRING;
}
```

注意事項

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検索します。

重要: getVersion() メソッドは、コネクターに対してインプリメントする必要がある 仮想メソッドです。

関連項目

CxVersion クラスの説明も参照してください。

---

init()

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。

構文

```
virtual int init(CxVersion * version) = 0;
```

パラメーター

*version***[in]** コネクター・フレームワークのバージョン・オブジェクトです。

戻り値

初期化操作の状況を示す整数です。一般的な戻り値は、次のとおりです。

BON_SUCCESS 初期化が成功しました。

BON_FAIL 初期化が失敗しました。

BON_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。

ほかの戻り値については、262 ページを参照してください。

注意事項

クラス・ライブラリーは、コネクターの起動時に init() メソッドを呼び出します。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インプリメントします。

重要: init() メソッドは、コネクターに対してインプリメントする必要がある 仮想メソッドです。

初期化の一環として、init() メソッドは、オプションで、コネクター・フレームワークのバージョンと、それが予期するバージョンを比較し、バージョンが一致する場合には成功を、コネクターがコネクター・フレームワークのバージョンと連動できない場合には失敗を戻すことができます。

関連項目

GenGlobals クラスの説明も参照してください。

isAgentCapableOfPolling()

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。

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・タイプ	戻り値
マスター (シリアル処理)	true
マスター (パラレル処理)	false
スレーブ (要求)	false
スレーブ (ポーリング)	true

注意事項

コネクターが単一処理モード (デフォルトでは、ParallelProcessDegree が 1 に等しい。) で稼働するように構成されている場合には、isAgentCapableOfPolling()

メソッドは常に true を戻します。同一のコネクター・プロセスがイベント・ポーリングと要求処理の両方を実行するからです。

コネクターが並列処理モード (ParallelProcessDegree が 1 より大) で稼働するように構成されている場合、表 108 に示すように、コネクターは複数のプロセスから構成され、各プロセスには固有の目的が割り振られています。

表 108. 並列処理コネクターのプロセスの目的

コネクター・プロセス・タイプ	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、このメソッドへの呼び出しを行うコネクターの目的によって異なります。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。

logMsg()

コネクターのログの宛先にメッセージを記録します。ログ・メッセージは、コネクター用のメッセージ・ファイルに組み込む必要があります。

構文

```
void logMsg(char * msg);
void logMsg(char * msg, int severity);
```

パラメーター

msg **[in]**
メッセージへのポインターです。

*severity***[in]**
次のメッセージ・タイプのいずれかにします。

```
XRD_WARNING
XRD_ERROR
XRD_FATAL
XRD_INFO
XRD_TRACE
```

戻り値

なし。

注意事項

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。

IBM では、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検索され、テキストがフォーマットされた後、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 つのステップに組み合わせることができます。

WebSphere InterChange Server

severity は 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、E メール通知がオンのときに E 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。

logMsg() でログに記録されたコネクター・メッセージは、メッセージ・ストリングが generateMsg() で生成された場合には、LogViewer を使用して表示することができます。

例

```
if ((form = CreateMainForm(conn, getFormName(theObj))) < 0) {
   msg = generateMsg(10, CxMsgFormat::XRD_FATAL, NULL, 0, NULL);
   logMsg(msg);
}
```

関連項目

GenGlobals::generateMsg() ユーティリティーの説明を参照してください。

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。

構文

```
virtual int pollForEvents() = 0;
```

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。次の戻りコードは、pollForEvents() メソッドによって一般に使用されます。

BON_SUCCESS　ポーリング・アクションが成功しました。

BON_FAIL　メソッドがポーリングで失敗しました。

BON_APPRESPONSETIMEOUT
　アプリケーションが応答しません。

ほかの戻り値については、262 ページの『doVerbFor()』 ページを参照してください。

注意事項

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。

重要: pollForEvents() メソッドは、コネクターに対してインプリメントしなければならない抽象メソッドです。

注: コネクターは、並列処理モードで実行する場合、別個のポーリング・スレーブ・プロセスを使用して、ポーリングを処理します。

関連項目

SubscriptionHandlerCPP クラスの説明も参照してください。

---

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

```
virtual int terminate() = 0;
```

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。一般的な戻り値は、次のとおりです。

BON_SUCCESS　終了が成功しました。

BON_FAIL　終了が失敗しました。

ほかの戻り値については、262 ページを参照してください。

注意事項

コネクター・フレームワークは、コネクターのシャットダウン時に、terminate() メソッドを呼び出します。このメソッドのインプリメントでは、すべてのメモリーを解放したことを確認し、アプリケーションからログオフしてください。

重要: terminate() メソッドは、コネクターに対してインプリメントする必要がある 仮想メソッドです。

traceWrite()

ログの宛先にトレース・メッセージを書き込みます。これは、コネクター・デベロッパーが使用するユーティリティー・メソッドです。

構文

```
void traceWrite(int traceLevel, char * info,
      char * filterName);
```

パラメーター

traceLevel **[in]** 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。

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::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、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。そのため、LEVEL0 の *traceLevel* に関連付けられたトレース・メッセージはいずれも印刷されません。

info **[in]** トレース・メッセージに使用されるメッセージ・テキストへのポインターです。

filterName **[in]** メッセージを書き込むために使用するフィルターへのポインターです。このパラメーターに対して、NULL を指定します。

戻り値

なし。

注意事項

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 *level* 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・メッセージが検索され、テキストがフォーマットされた後、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 つのステップに組み合わせることができます。

- 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。したがって、トレース・メッセージ (英語) を直接 traceWrite() への呼び出しに含めることができます。generateMsg() または generateAndTraceMsg() メソッドを使用する必要はありません。

traceWrite() でログに記録されたコネクター・メッセージは、LogViewer を使用して表示することができません。

例

```
traceWrite(Tracing::LEVEL3, "Invalid LoginId", NULL);
```

関連項目

generateAndTraceMsg(), generateMsg()

Tracing クラスの説明も参照してください。

使用すべきでないメソッド

GenGlobals クラス内の一部のメソッドは、以前のバージョンでサポートされていましたが、現行ではサポートされません。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マイグレーション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 109 に、GenGlobals クラスに対する使用すべきでないメソッドをリストします。(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 109. GenGlobals 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 18 章 ReturnStatusDescriptor クラス

ReturnStatusDescriptor クラスを使用すると、コネクターは、戻り状況記述子でエラー・メッセージや通知メッセージを戻すことができます。この記述子により、通常の場合、統合ブローカーに送信される要求応答の一部として追加状況情報を返送することが可能になります。

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の中にある情報にアクセスすることによって、そのサービス呼び出し要求の状況を取得できます。

このクラスのヘッダー・ファイルは ReturnStatusDescriptor.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

表 110 に、ReturnStatusDescriptor クラスのメソッドについての要約をまとめます。

表 110. 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
getErrorMsg()	戻り状況記述子からエラー・メッセージ・ストリングを検索します。	337
getStatus()	戻り状況記述子から状況値を検索します。	338
seterrMsg()	戻り状況記述子にエラー・メッセージまたは通知メッセージを含むストリングを設定します。	338
setStatus()	戻り状況記述子で状況値を設定します。	339

getErrorMsg()

戻り状況記述子からエラー・メッセージ・ストリングを検索します。

注: このメソッドは、コネクター・フレームワークに対してのみ 使用されます。

構文

```
char * getErrorMsg();
```

パラメーター

なし。

© Copyright IBM Corp. 1997, 2004

戻り値

統合ブローカーへのエラー・メッセージまたは通知メッセージを含むストリング。

getStatus()

戻り状況記述子から状況値を検索します。

構文

```
int getStatus();
```

パラメーター

なし。

戻り値

統合ブローカーの状況値を含んでいる整数。

seterrMsg()

戻り状況記述子にエラー・メッセージまたは通知メッセージを含むストリングを設定します。

構文

```
void seterrMsg(char * errMsg);
```

パラメーター

errMsg **[in]**　　メッセージ・ストリングです。

戻り値

なし。

注意事項

seterrMsg() を使用すると、統合ブローカーへのメッセージを含むストリングを戻すことができます。

例

```
int ExampleBOHandler::doVerbFor(BusinessObject &theObj,
   ReturnStatusDescriptor *rtnObj)
{
int status = BON_SUCCESS;
char *verb = theObj.getVerb();

if (strcmp(verb, CREATE) == 0)
   status = doCreate(theObj);
else if (strcmp(verb, Verb) == 0)
   // Check for other verbs and call verb routines
   else
   {
```

```
      // Send the collaboration a message that
      // this verb is not supported.
      char errorMsg[512];
      sprintf(errorMsg, "The verb '%s' is not supported ", verb);
      rtnObj->seterrMsg(errorMsg);
      status = BON_FAIL;
   }

return status;
}
```

setStatus()

戻り状況記述子で状況値を設定します。

構文

```
void setStatus(int status);
```

パラメーター

status[in]　　戻り状況記述子で保管する状況値です。

戻り値

なし。

第 19 章 SubscriptionHandlerCPP クラス

SubscriptionHandlerCPP クラスは、サブスクリプション・マネージャーを表します。サブスクリプション・マネージャーを使用すると、ビジネス・オブジェクトが統合ブローカーの処理対象かどうかを判別できます。サブスクリプション・マネージャーのメソッドを使用して統合ブローカーにビジネス・オブジェクトを送信することもできます。

このクラスのヘッダー・ファイルは SubscriptionHandlerCPP.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

表 111 に SubscriptionHandlerCPP クラスのメソッドを要約します。

表 111. SubscriptionHandlerCPP 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
SubscriptionHandlerCPP()	サブスクリプション・マネージャーを作成します。一般に、このメソッドを呼び出してサブスクリプション・マネージャーを作成することはありません。通常は、コネクター・フレームワークがコネクターのサブスクリプション・マネージャーを作成します。	341
gotApplEvent()	InterChange Server にビジネス・オブジェクトを送信します。	342
isSubscribed()	ビジネス・オブジェクト定義に対してサブスクリプションが存在するかどうかを判別します。	344

SubscriptionHandlerCPP()

サブスクリプション・マネージャーを作成します。これは、SubscriptionHandlerCPP クラスのインスタンスです。

構文

```
SubscriptionHandlerCPP();
```

パラメーター

なし。

戻り値

なし。

注意事項

ビジネス・オブジェクト・ハンドラーはサブスクリプション・マネージャーを使用して、ビジネス・オブジェクトにサブスクリプションが存在するかどうかを判別します。一般に、このメソッドを使用してサブスクリプション・マネージャーを作成

© Copyright IBM Corp. 1997, 2004

することはありません。コネクターのサブスクリプション・マネージャーはコネクター・クラス・フレームワークを作成します。

関連項目

BOHandlerCPP クラスの説明も参照してください。

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。これは非同期要求です。

構文

```
int gotApplEvent(BusinessObject busObj);
```

パラメーター

busObj **[in]** 統合ブローカーに送信されるビジネス・オブジェクトです。

戻り値

イベント・デリバリーの結果状況を示す整数です。状況を判定するため、この整数値を次の結果状況定数と比較してください。

`BON_SUCCESS`
: 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・オブジェクトのデリバリーに成功しました。

`BON_FAIL`
: イベント・デリバリーが失敗しました。

`BON_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。

`BON_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。

注意事項

`gotApplEvent()`メソッドは、コネクター・フレームワークに *busObj* 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、データを直列化して適切に永続化するために、イベント・オブジェクトに対して処理を実行します。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。

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、`gotApplEvent()` は、次の条件をチェックして、条件が成立しなかった場合に対応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。	BON_CONNECTOR_NOT_ACTIVE
イベント用のサブスクリプションがあるかどうか。	BON_NO_SUBSCRIPTION_FOUND

注: gotApplEvent() では、送信されるビジネス・オブジェクトと動詞に有効なサブスクリプションがあることが確認されるので、gotApplEvent() を呼び出した直後に 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、サブスクライブされたイベントを送信するよう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。

ポーリング・メソッドは、gotApplEvent() からの戻りコードを検査して、戻されたエラーが適切に処理されたことを確認します。例えば、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。

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。

例

```
SubscriptionHandlerCPP * theSubHandler =
   GenGlobals::getTheSubHandler();

// Determine whether there are subscribers to the event
if (theSubHandler->isSubscribed(obj_name, obj_verb) != TRUE) {
   // log message
   // delete event from event table
   // add event to archive table
   continue;
}

// Prepare to retrieve data into the business object
pObj = new BusinessObject(obj_name);
pObj->setVerb("Retrieve");
```

```
// Set key in business object

// Call the business object handler doVerbFor()
// to retrieve data
if (pObj->doVerbFor() == BON_FAIL) {
   // Log error message if retrieve fails
   retcode = BON_FAIL;
   break;
}

// Call gotApplEvent() to send the business object
pObj->setVerb(obj_verb);
theSubHandler->gotApplEvent(*pObj);
if ((theSubHandler->gotApplEvent(*pBusObj)) == BON_FAIL) {
   // Log error message
   retcode = BON_FAIL;
   break;
}
```

関連項目

BusinessObject クラスおよび pollForEvents() メソッドの説明も参照してください。

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

```
int isSubscribed(char * busObjName, char * verb);
```

パラメーター

busObjName **[in]**
ビジネス・オブジェクトの名前です。

verb **[in]**
ビジネス・オブジェクトのアクティブな動詞です。

戻り値

指定されたビジネス・オブジェクトと動詞の受信が統合ブローカーの処理対象である場合は True を表す 1、そうでない場合は False を表す 0 を戻します。

注意事項

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、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。

その他の統合ブローカー

WebSphere MQ Integrator 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・オブジェクトへのサブスクリプションについてコネクター・フレームワークに照会した場合、このメソッドは、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 `0` (True) を戻します。

例

```
SubscriptionHandlerCPP &theSubHandler =
   GenGlobals::getTheSubHandler();
if (theSubHandler->isSubscribed(theObj->getName(), theObj->getVerb())) {
   theSubHandler->gotApplEvent(theObj);
}
```

第 20 章 StringMessage クラス

StringMessage クラスは、StringMessage オブジェクトの内容にアクセスするメソッドを提供します。このクラスのヘッダー・ファイルは StringMessage.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

DevelopmentKits¥cdk¥generic_include

表 112 に、StringMessage クラスのメソッドの要約を示します。

表 112. StringMessage 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
hasMoreTokens()	StringMessage オブジェクトに、さらにストリング・トークンがあるかどうかを示します。	347
nextToken()	次のストリングまたは NULL を戻します。	347

hasMoreTokens()

StringMessage オブジェクト内にさらにストリングがあるかどうかを判別します。このメソッドを使用すると、StringMessage オブジェクトをループすることができます。

構文

```
unsigned char hasMoreTokens();
```

パラメーター

なし。

戻り値

StringMessage オブジェクトにさらにストリングがある場合は 1、ない場合は 0 を戻します。

nextToken()

StringMessage オブジェクト内の次のストリング (トークン) を戻します。

構文

```
char * nextToken();
```

パラメーター

なし。

© Copyright IBM Corp. 1997, 2004

戻り値

StringMessage オブジェクト内の次のストリング、またはストリングがもうない場合は NULL を戻します。

使用すべきでないメソッド

StringMessage クラスのメソッドは、以前のバージョンではサポートされていましたが、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マイグレーション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 113 に、StringMessage クラスで使用すべきでないメソッドを示します。(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 113. StringMessage クラスで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
getCurrentSize()	なし
initTokenizer()	なし

第 21 章 Tracing クラス

Tracing クラスは、コネクター用のトレース・サービスを提供します。このクラスのヘッダー・ファイルは Tracing.hpp です。これは、製品ディレクトリーの以下のサブディレクトリーにあります。

```
DevelopmentKits¥cdk¥generic_include
```

このクラスの内容は以下のとおりです。

- 『トレース・レベル定数』
- 『メソッド』

トレース・レベル定数

Tracing クラスは、表 114 に示されているトレース・レベル定数を定義します。

表 114. Tracing クラスで定義されているトレース・レベル定数

トレース・レベル定数	意味
LEVEL0	トレース・レベル 0 (トレースがオフ)
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 115 に Tracing クラスのメソッドを要約します。

表 115. Tracing 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
Tracing()	コネクター用の Tracing クラスのインスタンスを作成します。このメソッドを呼び出して Tracing クラスのインスタンスを作成することはしないでください。コネクター用の Tracing クラスのインスタンスは、コネクター・クラス・フレームワークが作成します。	
getIndent()	トレース・メッセージのインデントを指定する文字値を検索します。	350
getName()	トレース・メッセージを書き込むビジネス・オブジェクトの名前を検索します。	350
getTraceLevel()	現行トレース・レベルを検索します。	350
setIndent()	メッセージのインデントを設定します。	351
write()	トレース・メッセージを書き込みます。	351

© Copyright IBM Corp. 1997, 2004

getIndent()

トレース・メッセージのインデントを指定する文字ストリングを検索します。

構文

```
static char * getIndent();
```

パラメーター

なし。

戻り値

トレース・メッセージのインデントを示す文字ストリング。

例

```
tempStr = theObj::getIndent();
```

getName()

トレース・メッセージで使用するサブシステム (コネクター名) の名前を検索します。

構文

```
char * getName() const;
```

パラメーター

なし。

戻り値

トレースされているサブシステムの名前を含む文字ストリング。

getTraceLevel()

現行トレース・レベルを検索します。トレース・レベルを設定するには、TraceLevel コネクター構成プロパティーを使用します。

構文

```
int getTraceLevel() const;
```

パラメーター

なし。

戻り値

現行トレース・レベルを示す以下の整数。

```
Tracing::LEVEL0
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

例

```
if(getTraceLevel() > Tracing::LEVEL0)
   write(Tracing::LEVEL1, "Connector failed to initialize.", NULL);
```

setIndent()

トレース・メッセージを書き込むためにトレースが使用するインデントを設定します。

構文

```
static void setIndent(char * newIndent);
```

パラメーター

newIndent **[in]** トレース・メッセージのインデントを指定する文字ストリングです。

戻り値

なし。

write()

コネクターのトレース・メッセージを書き込みます。

注: ほとんどのトレース・メッセージに対して、GenGlobals クラスおよび BOHandlerCPP クラスにある traceWrite() ユーティリティー・メソッドを使用できます。

構文

```
void write(int traceLevel, char * info);

void write(int traceLevel, char * info,
   char * filterName);
```

パラメーター

traceLevel **[in]** メッセージの書き込みに使用する以下のトレース・レベルのいずれかです。

```
Tracing::LEVEL1
Tracing::LEVEL2
Tracing::LEVEL3
Tracing::LEVEL4
Tracing::LEVEL5
```

注: トレース・メッセージに、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。そのため、LEVEL0 の *traceLevel* に関連付けられたトレース・メッセージはいずれも印刷されません。

info **[in]**
トレース・メッセージのテキストを含む文字ストリングです。

filterName **[in]**
トレース・フィルターの名前です。

戻り値

なし。

例

```
write(Tracing::LEVEL4, "Connector failed to initialize.", NULL);
```

関連項目

BOHandlerCPP クラスおよび GenGlobals クラスの traceWrite() メソッドの説明も参照してください。

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。

© Copyright IBM Corp. 1997, 2004

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 355 ページの表 116 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 116 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 116 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 116. 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

表 116. 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
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)
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

表 116. 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
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
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

表 116. 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
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	
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: C:¥crossworlds¥リポジトリーに設定
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

表 116. 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
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
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インプリメントするには、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・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> の場合にのみ適用されます。

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> の場合にのみ適用されます。

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> の場合にのみ適用されます。

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。

ロケール名のフォーマットは、以下のとおりです。

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> の場合にのみ適用されます。

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 DH 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 DH 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。
これが受け入れられると、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、 361 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』
- 374 ページの『Connector Configurator の始動』
- 375 ページの『コネクター固有のプロパティー・テンプレートの作成』
- 378 ページの『新規構成ファイルの作成』
- 381 ページの『構成ファイル・プロパティーの設定』
- 390 ページの『グローバル化環境における Connector Configurator の使用』

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

© Copyright IBM Corp. 1997, 2004

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 (『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。システム内で既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、376 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 (381 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」フォルダーでいずれかの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てい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、376 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」**をクリックします。
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、ご使用の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、以前に作成した値が表示されます。

「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、そのテーブルの表示内から編集できるようになります。

テーブル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・ボタンでクリックし、**「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」**表示で依存プロパティーを強調表示させて矢印をクリックし、**「依存プロパティー」**表示に移動させます。
6. **「完了」**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。

新規構成ファイルの作成

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、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」を選択します。**

  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」**をクリックします。
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」**をクリックします。

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 384 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**「すべてファイルに保管」**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**「すべてプロジェクトに保管」**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のプロパティーは、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た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後、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、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・ボタンをクリックします。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. 382 ページの『標準コネクター・プロパティーの設定』の説明に従い、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・ユーザーズ・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 354 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

コネクターで使用するビジネス・オブジェクトを指定するには、Connector Configurator の**「サポートされているビジネス・オブジェクト」**タブを使用します。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」**リストで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、**「プロジェクトの保管」**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・オブジェクトがエージェント・サポートを備えている場合、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 を実行している場合、**「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 を選択した場合、Connector Configurator にメッセージ・セット ID は必要ありません。**「サポートされているビジネス・オブジェクト」**タブには、サポートされているビジネス・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 を実行している場合、「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可能なビジネス・オブジェクトのリストが示されます。このリストから必要なビジネス・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ているビジネス・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」タブでビジネス・オブジェクトを追加指定した場合、その内容は、「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、各コネクターでサポートされるそれぞれのビジネス・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. **「明示的 (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できます。

すべてのコネクターがこの機能をサポートしてい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『コネクターの標準構成プロパティー』にある `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。
- 指定したディレクトリーに保管します。
- 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**「BrokerType」**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一部のみです。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、以下に太文字で示した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. コネクター・スクリプト生成プログラム

コネクター・スクリプト生成ユーティリティーは、UNIX プラットフォームで稼働す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方法のいずれかの場合に使用します。

- WebSphere Business Integration Adapters インストーラーを使用せずに追加したコネクター用に、新しいコネクター開始スクリプトを生成する場合。
- 訂正された構成ファイル・パスがコネクターに組み込まれるように既存の開始スクリプトを変更する場合。

以下のようにしてコネクター・スクリプト生成プログラムを稼働します。

1. *ProductDir*/bin ディレクトリーにナビゲートする。
2. コマンド./ConnConfig.sh を入力する。

   「コネクター・スクリプト生成プログラム (Connector Script Generator)」画面が図 80 のように表示されます。

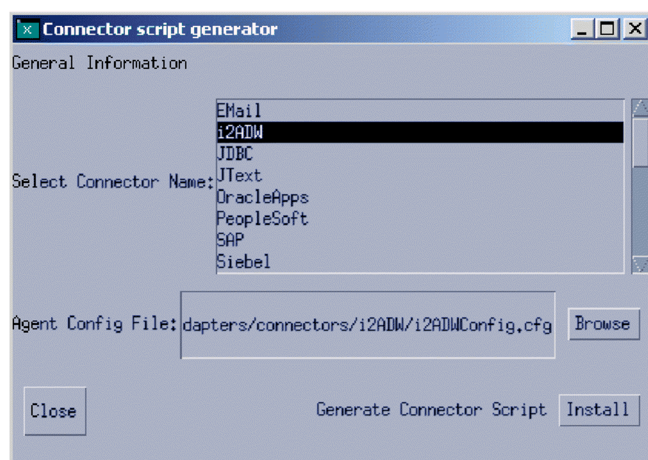

図 80. コネクター・スクリプト生成プログラム (Connector Script Generator)

3. 「コネクター名を選択 (Select Connector Name) リストから、生成する開始スクリプトに応じたコネクターを選択する。
4. コネクターの完全パス名を指定または「**ブラウズ (Browse)** をクリックして、ファイルを選択し、エージェント構成ファイル用のコネクター構成を指定する。
5. 「**インストール (Install)**」をクリックして、コネクター・スクリプトを生成または更新する。

   connector_manager_*ConnectorName* ファイル (*ConnectorName* は構成中のコネクター名) が *ProductDir*/bin ディレクトリーに作成されます。
6. 「**閉じる (Close)**」をクリックする。

© Copyright IBM Corp. 1997, 2004

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、コネクターの標準機構のそれぞれ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。

コネクターのインプリメント局面で、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、それぞれの機能ごとに「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、それぞれの機能ごとに「*部分*」にチェックマークを付け、インプリメント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、それぞれの機能ごとに「*いいえ*」にチェックマークを付けます。
- コネクターに関係のない、それぞれの機能ごとに「*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インプリメントしない場合には、すべてのイベント通知機能で「いいえ」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 117 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。このテーブルには、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資料内のセクションのページ番号も記載されています。

表 117. 要求処理での標準機構

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 __ いいえ __ 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 __ いいえ __ 使用不可
Create		

© Copyright IBM Corp. 1997, 2004

表 117. 要求処理での標準機構 (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。91 ページの『Create 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。111 ページの『Delete 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。108 ページの『論理 Delete イベントを表すビジネス・オブジェクトの含意』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。113 ページの『Exists 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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検索されます。検索は、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。95 ページの『Retrieve 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。 98 ページの『子オブジェクトの検索』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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検索されます。101 ページの『RetrieveByContent 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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検索された場合には、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。101 ページの『RetrieveByContent 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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 __ いいえ __ 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()` 呼び出しで検索されたビジネス・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。103 ページの『Update 動詞の処理』を参照してください。	__ 完全 __ 部分 __ いいえ __ 使用不可

表 117. 要求処理での標準機構 (続き)

カテゴリーと名前	説明	サポート状況
デルタ・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 __ いいえ __ 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 __ いいえ __ 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。89 ページの『動詞安定度』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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る必要があります。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 __ いいえ __ 使用不可

イベント通知の標準的な振る舞い

表 118 では、イベント検索および通知での標準機構をリストします。

表 118. イベント通知での標準機構

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		

表 118. イベント通知での標準機構 (続き)

カテゴリーと名前	説明	サポート状況
イベントの分散	イベント検出メカニズムには、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。142 ページの『イベントの分散』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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検索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。) 207 ページの『イベント・レコードの検索』を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。 395 ページの表 118 を参照してください。	__ 完全 __ 部分 __ いいえ __ 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。125 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 __ いいえ __ 使用不可
オブジェクト名	オブジェクト名は、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。124 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 __ いいえ __ 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、ご注意ください。減分の場合には (めったに起こりえませんが)、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)。142 ページの『イベントの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 __ いいえ __ 使用不可
各種機能		

表 118. イベント通知での標準機構 (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 に正常にデリバリーされてもされなくても、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が、そのイベントと動詞の組み合わせのサブスクリプションがコネクターになかったため、イベントは InterChange Server に送信されませんでした。 • エラー。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、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 __ いいえ __ 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 __ いいえ __ 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 __ いいえ __ 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 __ いいえ __ 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、E 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、「Notes」フィールドを使用してください。	__ 完全 __ 部分 __ いいえ __ 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。143 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 __ いいえ __ 使用不可
RetrieveAll	コネクターは、サブスクリプション・デリバリー時に階層ビジネス・オブジェクト全体を検索します。212 ページの『アプリケーション・データの検索』を参照してください。	__ 完全 __ 部分 __ いいえ __ 使用不可

表 118. イベント通知での標準機構 (続き)

カテゴリーと名前	説明	サポート状況
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、イベント検出メカニズムによって、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。イベント検出メカニズムは、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 レコードを除去します。	__ 完全 __ 部分 __ いいえ __ 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。209 ページの『ビジネス・オブジェクト名、動詞、およびキーの取得』を参照してください。	__ 完全 __ 部分 __ いいえ __ 使用不可

一般標準

表 119 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 119. 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		
外部キー	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、「いいえ」にチェックマークを付けます。	__ 完全 __ 部分 __ いいえ __ 使用不可
外部キー属性プロパティー	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 __ いいえ __ 使用不可
キー	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、「いいえ」にチェックマークを付けます。	__ 完全 __ 部分 __ いいえ __ 使用不可
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、「いいえ」にチェックマークを付けます。	__ 完全 __ 部分 __ いいえ __ 使用不可

表 119. 一般標準 (続き)

カテゴリーと名前	説明	サポート状況
必須	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、「いいえ」にチェックマークを付けます。	__ 完全 __ 部分 __ いいえ __ 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 __ いいえ __ 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、E メール通知を発生させるように、`APPRESPONSETIMEOUT` の戻りコードを送信します。81 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 __ いいえ __ 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、E メール通知を発生させるように、`APPRESPONSETIMEOUT` の戻りコードを送信します。81 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 __ いいえ __ 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、E メール通知を発生させるように、`APPRESPONSETIMEOUT` の戻りコードを送信します。	__ 完全 __ 部分 __ いいえ __ 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 __ いいえ __ 使用不可
アプリケーション・ユーザー名	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 __ いいえ __ 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 __ いいえ __ 使用不可
メッセージ・トレース		
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たびに、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たび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・メッセージがその後に続きます。コネクターはトレース設定のレベル以下の範囲で、該当するすべてのトレース・メッセージをデリバリーする必要があります。155 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 __ いいえ __ 使用不可

表 119. 一般標準 (続き)

カテゴリーと名前	説明	サポート状況
トレース・レベル 0	0 - Connector 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 __ いいえ __ 使用不可
トレース・レベル 1	1 - 処理されたそれぞれのビジネス・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たびに、メッセージが送信されます。	__ 完全 __ 部分 __ いいえ __ 使用不可
トレース・レベル 2	2 - コネクターが処理する、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たびに、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たびに指示するメッセージ。	__ 完全 __ 部分 __ いいえ __ 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・オブジェクトで外部キーに遭遇したときに、またはコネクターがビジネス・オブジェクトで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。このメッセージの例には、ビジネス・オブジェクト間の一致の検索や子ビジネス・オブジェクトの配列でのビジネス・オブジェクトの検索などがあります。	__ 完全 __ 部分 __ いいえ __ 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関数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、コネクターの処理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に、メッセージは、それぞれの新規のスレッドの作成をログに記録します。	__ 完全 __ 部分 __ いいえ __ 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 から検索された、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 __ いいえ __ 使用不可
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 __ いいえ __ 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 __ いいえ __ 使用不可

表 119. 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。153 ページを参照してください。	__ 完全 __ 部分 __ いいえ __ 使用不可
CDK メソッド `logMsg()`	常に、`logMsg()` を呼び出す前に、CDK メソッド `generateMsg()` を使用します。	__ 完全 __ 部分 __ いいえ __ 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 __ いいえ __ 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期間中に障害が検出された場合には、トランザクション全体をロールバックする必要があります。	__ 完全 __ 部分 __ いいえ __ 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ブランク値を挿入します。ブランク値は、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。199 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 __ いいえ __ 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。199 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 __ いいえ __ 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。

© Copyright IBM Corp. 1997, 2004

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。

著作権使用許諾

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

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。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、IBM Corporation の商標です。

IBM
IBM 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］

アーカイブ表　128
アーカイブ・ストア　140
　アクセス　141
　作成　140
アーカイブ・レコード　140
アクセス要求　23
アダプター　3
　開発用ツール　31
アダプター・フレームワーク　31
アプリケーション
　イベント・ストアのインプリメント　124
　オブジェクト・ベース　53, 84
　操作の開始　192
　バージョン　74
　フォーム・ベース　53, 84, 85, 132
　API　47
アプリケーション固有の情報
　属性に対する　85, 98, 186
　動詞の　85, 183
　トレース　158
　ビジネス・オブジェクト定義に対する　85, 183, 184
アプリケーション固有のビジネス・オブジェクト　7, 14
　設計　48
　汎用ビジネス・オブジェクトへのマッピング　13
　ビジネス・オブジェクト開発のスコープ　52
アプリケーションとの接続
　確立　73, 169
　クローズ　77
　検証　175, 206
　切断の処理　81, 122, 176, 207
アプリケーション・エンティティーの非正規化　51
アプリケーション・データベース　44, 50
　イベント表　128
　エンティティー内のキー　118
　エンティティーの検索　95
　エンティティーの更新　104
　エンティティーの削除　112
　エンティティーの作成　92
　エンティティーの照会　113
　トリガーの所在　133
イベント　23
　アーカイブ　139, 218
　アンサブスクライブされた　211
　コネクター ID　125, 129, 142
イベント *(続き)*
　将来　136
　進行中　74, 170
　説明　125, 129
　重複　74, 134, 135
　同期　319
　動詞　125, 128, 134, 144, 209
　トリガー　23
　発効日　136
　ビジネス・オブジェクト名　125, 128, 209
　非同期　343
　分散　142
　ポーリング可能　134, 208
イベント ID　126, 150
　イベント表　128
　イベント・レコード　125
　取得　209
　初期化　134
イベント検索　123, 136, 138
　機構の用途　136
イベント検出　123, 130, 136
　機構の用途　131
　将来のイベント　136
　重複イベント　135
　データベース・トリガー　133
　標準的な振る舞い　134
　フォーム・イベント　132
　ワークフロー　132
イベント状況　126
　イベント表　128
　イベント・レコード　125
　初期化　134
イベント通知　7, 25, 28, 44, 123, 153
　アンサブスクライブされたイベント　211
　イベント検索　123, 136, 138
　イベント検出　123, 130, 136
　イベントの分散　142
　イベント表　135
　イベント・ストア　135
　エラー処理　218
　削除イベント　143
　将来のイベント処理　136
　設計上の課題　57
　トランスポート層　19, 21
　標準的な振る舞い　395
イベント通知機構　26, 28, 57, 123, 124
イベントの優先順位　142
　イベント表　128
　イベント・レコード　125
　初期化　134
イベント表　128, 149

© Copyright IBM Corp. 1997, 2004

イベント・ストア　26, 123, 124, 130
　イベント・レコードの挿入　134
　可能なインプリメント形態　127
　将来　136
　定義　124
　フラット・ファイル　130, 149
　E メールのメールボックス　129, 149
　JMS　146, 149
イベント・タイム・スタンプ
　イベント表　128
　イベント・レコード　125
　使用法　208
　初期化　134
イベント・トリガー処理フロー　22, 89
イベント・ポーリング　332
イベント・レコード　26, 123
　アーカイブ　218
　イベント・ストアへの挿入　134
　オブジェクト・キー　125, 134, 209
　検索　207
　作成　134
　標準的内容　57, 124
エラー処理　78, 224
エラー・メッセージ　153, 163, 264
エラー・ロギング　153
オブジェクト・バージョン　309
　オブジェクトのポイント・バージョン　314, 315
　オブジェクトのマイナー・バージョン　315
　オブジェクトのメジャー・バージョン　315
　区切り文字　312
　現行の比較　312
　最新バージョン　313
　変換　316
　ポイント・バージョンの比較　311
　マイナー・バージョンの検索　314
　マイナー・バージョンの比較　311
　メジャー・バージョンの検索　313
　メジャー・バージョンの比較　310
　文字ストリング　312
オブジェクト・リクエスト・ブローカー (ORB)　18, 72

［カ行］

カーディナリティー
　取得　252
　単一　117, 118, 119, 249
　判別　119, 255
　複数の　117, 118, 120, 249
階層ビジネス・オブジェクト　117
　削除操作　112
　処理　117, 202
　Create 操作　91
　Retrieve 操作　95
　RetrieveByContent 操作　102
　Update 操作　104
開発過程　35
外部キー属性　92, 93, 105, 118, 158, 186, 257, 398
キー属性　186, 257
キー属性プロパティー　398
基本キー　92, 118
クライアント・コネクター・フレームワーク　11
警告　153, 163
コネクター　6
　アプリケーション固有のコンポーネント　22, 77, 317
　アプリケーションとの接続の切断　81, 122, 176, 207
　イベント・ポーリング　332
　インプリメンテーションに関する質問　59
　インプリメント　167, 227
　開発過程　36
　開発環境　33
　開発サポート　32
　関連マップ　232
　基底クラス　77, 167
　構成　33
　構成ファイル　233
　構成プロパティー　324
　国際化対応　63, 71, 161
　コネクターによるやり取り　11, 16, 17, 20
　コンパイル　228
　コンポーネント　8
　サポートされるビジネス・オブジェクト　7, 30, 72, 73, 75, 87, 170, 232
　サンプル　34
　実行　71
　始動　72, 234
　シャットダウン　71, 77, 224
　終了　77, 224, 332
　初期化　15, 18, 159, 168, 328
　進行中イベントのリカバリー　170
　スレッド化の問題　141
　設計上の課題　41
　単一方向　44
　定義　37, 230
　ディレクトリー　234, 235
　トレース・メッセージ　265, 321, 333, 351
　名前　227
　バージョン　74, 169, 327
　汎用機能　71
　ビジネス・インテグレーション・システムへの追加　227
　必須のインプリメント　87, 95
　部分的にメタデータ主導型　55
　並列処理　142, 268, 325, 330, 332
　ポーリング頻度　332
　命名規則　77, 167
　メタデータ主導型　53, 76, 84, 171, 182, 399
　メタデータを使用しない　56
　メッセージのロギング　264, 269, 319, 322, 330
　モニター　155
　役割　6, 25, 44
　要求処理　83, 122
　ライブラリー　228, 234, 235
　ログの宛先　153

コネクター *(続き)*
ADK サポート 32
JMS 対応 145
コネクター ID 125, 142
コネクター開発
オペレーティング・システム 33
ツール 32
コネクター構成プロパティー 79
検索 18, 80, 81, 267, 324
国際化対応 68
コネクター固有 79
設定 80, 232
単純 81
定義 80, 233
トレース 159
標準 79, 353, 372
ロード 72, 73
ApplicationPassword 73, 80
ApplicationUserID 73, 80
ArchiveProcessed 141, 218
CharacterEncoding 69, 325
ConnectorId 143
ContainerManagedEvents 146
DataHandlerConfigMOName 147
DeliveryTransport 20, 146, 149
DHClass 147
DuplicateEventElimination 149
IgnoreMissingChildObject 100, 101, 103, 394
InDoubtEvents 74, 397
Locale 326
LogAtInterchangeEnd 74, 81, 122, 154, 176, 320, 331
MimeType 147
MonitorQueue 149
ParallelProcessDegree 268, 325, 330
PollAttributeDelimiter 126
PollFrequency 71, 137, 332
PollQuantity 146, 149, 208, 396
SourceQueue 147
TraceLevel 156, 334, 350
UseDefaults 284, 399
コネクター始動スクリプト 137, 154, 156, 234
コネクター定義 37, 230
コネクターのクラス・ライブラリー 78
コネクター・コントローラー 11, 72
サブスクリプション処理 14
サブスクリプション・リスト 15, 26
マッピングでの役割 14
コネクター・スクリプト生成プログラム (Connector Script Generator) 391, 393
コネクター・フレームワーク 9, 22
アプリケーション固有コンポーネントの起動 72
起動 72
結果状況値への応答 225
国際化対応 65
コネクター応答の決定 195
コネクターの初期化 72, 73, 168
コネクター・フレームワーク *(続き)*
サービス 10, 16
サービス呼び出し要求の受信 88
サブスクリプション処理 15, 27, 210, 345
サブスクリプション・リスト 15, 27, 210, 345
トランスポート層 16
トレース 155
ビジネス・オブジェクトの送信 215
ビジネス・オブジェクト・ハンドラーの取得 30, 75, 170
ビジネス・オブジェクト・ハンドラーの選択 28
ポーリング・メソッドの呼び出し 137
文字エンコード 325
ロケール 67
doVerbFor() からの応答 121
locale 67, 282, 326
コネクター・メッセージ・ファイル
名前 160
メッセージの生成 66, 320, 331, 334
ロケーション 160
子ビジネス・オブジェクト 117
アクセス 119, 121, 202
関係型 117, 254
検索 98, 294
個数の判別 294
動詞サポート 89
バージョン 251
ビジネス・オブジェクト配列への挿入 295
コラボレーション 6, 23, 49, 318, 324
イベント通知での役割 25
サブスクライブの有無の判別 72, 210, 345
状況を戻す 337
ビジネス・オブジェクトの送信 216
要求処理での役割 29

[サ行]

サービス呼び出し応答 24, 31
サービス呼び出し要求 13, 19, 24, 29
削除操作 143
物理 143, 212, 215
論理 105, 108, 125, 143, 212
サブスクリプション処理 14
サブスクリプション・ハンドラー 15, 206, 341, 347
作成 341
ポインターを検索 268
InterChange Server にイベントを送信 342
サブスクリプション・マネージャー 21, 206, 220, 327
使用すべきでないメソッド
CxMsgFormat 307
GenGlobals 334
StringMessage 348
設計上の課題
アプリケーション API の使用 47
アプリケーション固有のビジネス・オブジェクトの決定 48
アプリケーションのアーキテクチャー 43
アプリケーションの対話 45

設計上の課題 *(続き)*
コネクターの役割の特定 44
質問のまとめ 59
メタデータ主導型の設計 52
OS 間通信 58
属性
アクセス 116, 185
アプリケーション固有の情報 85, 98, 186, 251
位置 301
カーディナリティー 258
外部キー 257
キー 257
記述子 185, 249, 276, 300
クラス 185, 186, 249
検証 284, 285
個数の判別 275, 301
最大長 186, 252, 398
作成 250
順序位置 116, 183, 186, 191, 193
初期化 284, 289
処理するかどうかの判別 188
単純 114, 116, 199
データ型 186, 254, 255, 258, 259
データ型整数 277
データ型名 256
名前 186, 191, 193, 253, 256, 277
ビジネス・オブジェクトを作成 288
必須 186, 259, 284, 285, 399
複合 119, 258
プレースホルダー 189
プロパティー 185, 186, 249, 250, 276, 301
属性値
検索 190, 278
設定 193, 201, 288
特殊 199

[タ行]

致命的エラー 163
重複イベント回避 149
通知メッセージ 153, 163, 264
データベース・トリガー 133
定数
結果状況 78, 225
属性値 271
属性タイプ 249
トレース・レベル 157, 349
メッセージ・タイプ 163
デバッグ 155
デフォルト属性値
取得 186, 252, 280
初期化 284, 289
統合ブローカー 3
動詞
アクティブ 174, 177, 262, 283
アプリケーション固有の情報 85, 183, 303
動詞 *(続き)*
基本処理 177
子ビジネス・オブジェクト内の 89
サポートされる 174, 183, 304
サポート対象かどうかの判別 304
推奨事項 89
設定 215
操作の実行 90, 180, 262
動詞安定度 89, 210, 215
分岐 177
メソッド 91, 178
メタデータ主導型処理 178
要求ビジネス・オブジェクトからの検索 174, 283
トップレベル・ビジネス・オブジェクト 117
トラブルシューティング 155
トランザクション 89
トリガー・イベント 23, 318
トレース 21, 155
インデントの文字ストリング 350, 351
国際化対応 65
コネクター名の検索 350
使用可能化 156
トレース・メッセージの書き込み 351
トレース・レベル 158, 270, 321, 333, 350
メッセージ 269, 333
メッセージ宛先 156
メッセージの送信 156
トレース・メッセージ 155, 159, 163, 321

[ハ行]

パブリッシュとサブスクリプションのモデル 25
ビジネス・オブジェクト 5, 12
値の抽出 190
値の保管 193
インスタンス 6
応答 96, 104, 122
親 117
親と子の関係 117, 254
親ビジネス・オブジェクト 254, 282
開発サポート 32
クラス 21, 271
コネクター・フレームワークに送信 215
コピー 273
サブスクリプションの検査 210, 344
サポートされる 18, 30, 72, 73, 75, 83, 87, 170, 232
処理 114, 182, 194
新規作成 288
トップレベル 117
トレース情報、ダンプ 275
パーツ 5
汎用 7, 14
ビジネス・オブジェクト定義 282
ビジネス・オブジェクト配列から除去 296
ビジネス・オブジェクト配列への挿入 295, 297
ビジネス・オブジェクト・ハンドラー 274

ビジネス・オブジェクト *(続き)*
命名 54
メタデータ 84
要求 28, 174, 180, 196
ログ情報、ダンプ 275
ADK サポート 31
InterChange サーバーに送信 342
locale 67, 281, 290
ビジネス・オブジェクト定義 5, 7, 299, 304
アクセス 182
アプリケーション固有の情報 85, 183, 184, 300
イベント内 125
クラス 182, 299
検索 282, 295
サポートされる動詞 183, 304
名前 183, 282, 302
バージョン 251, 284, 303
ハンドラー 302, 323
ビジネス・オブジェクト配列 117
クラス 293
子ビジネス・オブジェクト 120
子ビジネス・オブジェクトの個数の判別 294
子ビジネス・オブジェクトを検索 294
ビジネス・オブジェクト定義 295
ビジネス・オブジェクトの挿入 295, 297
ビジネス・オブジェクトを除去 296
ビジネス・オブジェクト・ハンドラー 30, 75, 83, 122
アクティブ動詞のアクションの実行 263
インスタンスの生成 75, 171
概要 86
クラス 30, 172, 261
検索 302, 323
作成 172, 204, 262
取得 30, 75, 170
設計上の課題 83
動詞処理 90
トレース情報 158
複数の 56, 76, 86, 171, 173
部分的にメタデータ主導型 55
メタデータ主導型 54, 76, 84, 171
役割 83, 173
呼び出し 274
表ベースのアプリケーション
アプリケーション固有の情報 85
イベント・レコードの検索 208
データベース・トリガー 133
ビジネス・オブジェクトの構造 114, 118
ビジネス・オブジェクト・ハンドラー 84
メタデータ主導型の設計 52, 53
物理削除 111, 143
フラット・ビジネス・オブジェクト 114
削除操作 111
処理 114
Create 操作 91, 196
Retrieve 操作 95
RetrieveByContent 操作 102
フラット・ビジネス・オブジェクト *(続き)*
Update 操作 103
ポーリング 76, 77, 138, 143, 204, 224
アプリケーション・データの検索 212
イベント情報の検索 209
イベントのアーカイブ 139, 218
イベントの送信 214
イベント・レコードの検索 207
間隔 137
機構の用途 136
基本ロジック 139, 204
サブスクリプションの検査 210
サブスクリプション・ハンドラーのセットアップ 206
サブスクリプション・マネージャーのセットアップ 206
接続の検証 206
重複イベント回避 149
標準的な振る舞い 137
ポーリング・メソッド 138
保証付きイベント・デリバリーおよび 148, 150
包含関係 117, 254

[マ行]

マッピング 13, 232
メタデータ 52
メッセージ 153
宛先 156
検索 337
生成 161
説明 160
ソース 159
タイプ 163
番号 160, 162
フォーマット 160
メッセージ・テキスト 160, 162
メッセージング・システム 18, 19
メッセージ・キュー 227
メッセージ・ファイル 159, 167
名前 160
ロケーション 160
メッセージ・ロギング 79, 153, 167, 305, 309
トレース 155, 269
メッセージの生成 162, 266, 305
メッセージ・ファイル 159
文字エンコード 64, 325
戻り状況記述子 79
クラス 337
状況 195, 338, 339
取り込み 195
メッセージ 175, 176, 195, 337, 338
doVerbFor() からの引き渡し 195
doVerbFor() への引き渡し 195, 226, 263
executeCollaboration() からの引き渡し 318

[ヤ行]

要求処理　8, 28, 31, 44, 83, 122
　トランスポート層　19, 21
　ビジネス・オブジェクト・ハンドラー基底クラスの拡張　86, 172
　標準的な振る舞い　393
要求ビジネス・オブジェクト　28, 174, 180, 196

[ラ行]

リポジトリー　35, 72, 73, 227, 230
例
　Create 動詞メソッド　196
　doVerbFor()　175, 177
　freeMemory()　163
　generateMsg()　163
　getBOHandlerforBO()　171, 172
　getVersion()　169
　init()　170
　logMsg()　163
　pollForEvents()　204, 219
　terminate()　224
　traceWrite()　157
例外処理　13
ロギング　21, 153
　国際化対応　65
　メッセージ宛先　156
　メッセージの送信　154
ログの宛先　153, 156
論理削除　105, 108, 111, 143

A

Adapter Development Kit (ADK)　32, 33
ApplicationPassword コネクター構成プロパティー　73, 80
ApplicationUserID コネクター構成プロパティー　73, 80
APPRESPONSETIMEOUT 結果状況　81
　doVerbFor()　122
AppSpecificInfo 属性プロパティー　85
ArchiveProcessed コネクター構成プロパティー　141, 218

B

Blank 属性値　200
　検査　200, 286
　取得　279
　定数　201, 271
BlankValue 属性値定数　271
BOAttrType クラス　185, 247, 249, 260
　インスタンスの作成　250
　コンストラクター　250
　属性タイプ定数　249
　ヘッダー・ファイル　249
　メソッドの要約　249
BOAttrType クラス *(続き)*
　BOOLEAN　249
　DATE　249
　DOUBLE　249
　FLOAT　249
　getAppText()　188, 251
　getBOVersion()　251
　getCardinality()　252
　getDefault()　252
　getMaxLength()　252
　getName()　253
　getRelationType()　254
　getTypeName()　254
　getTypeNum()　255
　hasCardinality()　255
　hasName()　256
　hasTypeName()　256
　INTEGER　249
　INTSTRING　249
　isForeignKey()　257
　isKey()　257
　isMultipleCard()　202, 258
　isObjectType()　189, 202, 258
　isRequired()　259
　isType()　259
　LONGTEXT　249
　LONGTEXTSTRING　249
　OBJECT　249
　STRING　249
　STRSTRING　249
BOHandlerCPP クラス　170, 172, 247, 261, 270
　インスタンスの作成　262
　仮想メソッド　261
　コンストラクター　262
　ヘッダー・ファイル　261
　メソッドの要約　261
　doVerbFor()　87, 173, 262
　generateAndLogMsg()　155, 162, 264
　generateAndTraceMsg()　157, 162, 265
　generateMsg()　155, 157, 162, 266
　getConfigProp()　267
　getTheSubHandler()　268
　logMsg()　155, 269
　traceWrite()　157, 269
BON_APPRESPONSETIMEOUT 結果状況　81, 225, 226
　doVerbFor()　122, 176, 194, 195, 225, 263
　pollForEvents()　77, 207, 225, 332
BON_BO_DOES_NOT_EXIST 結果状況　101, 194, 225, 263
BON_CONNECTOR_NOT_ACTIVE 結果状況　225, 342
BON_FAIL 結果状況　225, 226
　Create 動詞　94
　Delete 動詞　112
　doVerbFor()　175, 194, 225, 263
　Exists 動詞　114
　gotApplEvent()　225, 342
　init()　74, 169, 225, 328

BON_FAIL 結果状況 *(続き)*
pollForEvents() 77, 219, 225, 332
Retrieve 動詞 101
terminate() 224, 225, 332
Update 動詞 105, 111
BON_FAIL_RETRIEVE_BY_CONTENT 結果状況 103, 194, 225, 263
BON_MULTIPLE_HITS 結果状況 103, 194, 196, 225, 263
BON_NO_SUBSCRIPTION_FOUND 結果状況 225, 342
BON_SUCCESS 結果状況 225
Create 動詞 94
doVerbFor() 194, 225, 263
Exists 動詞 114
gotApplEvent() 225, 342
init() 74, 169, 225, 328
pollForEvents() 77, 219, 225, 332
terminate() 224, 225, 332
Update 動詞 110
BON_UNABLETOLOGIN 結果状況 74, 169, 225, 328
BON_VALCHANGE 結果状況 225
Create 動詞 94
Delete 動詞 112
doVerbFor() 194, 196, 225, 263
Retrieve 動詞 100, 101
RetrieveByContent 動詞 103
Update 動詞 110
BON_VALDUPES 結果状況 94, 194, 225, 263
BOOLEAN 属性タイプ定数 249, 255, 259, 277, 289
Business Object Designer 6
BusinessObject クラス 247, 271, 291
インスタンスの作成 272
コンストラクター 68, 272
属性値定数 271
ヘッダー・ファイル 271
メソッドの要約 271
BlankValue 271
clone() 273
doVerbFor() 213, 274
dump() 275
getAttrCount() 185, 191, 275
getAttrDesc() 185, 276
getAttrName() 277
getAttrType() 277
getAttrValue() 189, 192, 202, 278
getBlankValue() 279
getDefaultAttrValue() 280
getIgnoreValue() 281
getLocale() 68, 281
getName() 282
getParent() 282
getSpecFor() 183, 282
getVerb() 174, 283
getVersion() 284
IgnoreValue 271
initAndValidateAttributes() 284
isBlankValue() 189, 286
BusinessObject クラス *(続き)*
isBlank() 286
isIgnoreValue() 189, 287
isIgnore() 287
makeNewAttrObject() 288
setAttrValue() 201, 213, 288, 289
setLocale() 290
setVerb() 213, 290
BusObjContainer クラス 203, 247, 293, 297
ヘッダー・ファイル 293
メソッドの要約 293
getObjectCount() 202, 203, 294
getObject() 202, 294
getTheSpec() 295
insertObject() 295
removeAllObjects() 296
removeObjectAt() 296
setObject() 297
BusObjSpec クラス 182, 248, 299, 304
ヘッダー・ファイル 299
メソッドの要約 299
getAppText() 184, 300
getAttributeCount() 185, 301
getAttributeIndex() 185, 301
getAttribute() 185, 300
getMyBOHandler() 302
getName() 302
getVerbAppText() 303
getVersion() 303
isVerbSupported() 304

C

Cardinality
判別 186
CharacterEncoding コネクター構成プロパティー 69, 325
clone() メソッド 273
Common Object Request Broker Architecture (CORBA) 17, 18
compareMajor() メソッド 310
compareMinor() メソッド 311
comparePoint() メソッド 311
compareTo() メソッド 312
Connector Configurator 33, 80, 231, 373, 391
Connector Development Kit 34
ConnectorId コネクター構成プロパティー 143
connector_manager_connector 始動スクリプト 234
consumeSync() メソッド (使用すべきでない) 335
ContainerManagedEvents コネクター構成プロパティー 146
Create 動詞
アプリケーション・データの検索 212
インプリメント 93
概要 91
結果状況 94, 195
属性値の使用 181, 190, 193
属性の初期設定 285
標準的な振る舞い 92

Create 動詞 *(続き)*
　Blank 値の処理　201
　Ignore 値の処理　201
createBusObj() メソッド　68
CWConnectorBOHandler クラス
　doVerbFor()　87
CWConnectorBusObj クラス
　getLocale()　68
CWConnectorUtil クラス
　createBusObj()　68
CwConnector.dll ライブラリー　229, 247
CxMsgFormat クラス　248, 305, 309
　使用すべきでないメソッド　307
　ヘッダー・ファイル　305
　メソッドの要約　305
　メッセージ・タイプ定数　158, 163, 264, 265, 266, 305, 319, 321, 322
　generateMsg()　305
CxVersion クラス　248, 309, 316
　インスタンスの作成　310
　コンストラクター　309
　ヘッダー・ファイル　309
　メソッドの要約　309
　compareMajor()　310
　compareMinor()　311
　comparePoint()　311
　compareTo()　312
　getDELIMITER()　312
　getLATESTVERSION()　313
　getMajorVer()　313
　getMinorVer()　314
　getPointVer()　314
　setMajorVer()　315
　setMinorVer()　315
　setPointVer()　315
　toString()　316
C++ Connector Development Kit　247
C++ Connector Development Kit (CDK)　34
C++ コネクター
　ライブラリー・ファイル　230
C++ コネクター・ライブラリー　21, 34
　概要　247
　結果状況値　225
　トレース　349
　戻りコード　225
　BOAttrType　249
　BOHandlerCPP　261
　BusinessObject　271
　BusObjContainer　293
　BusObjSpec　299
　CxMsgFormat　305
　CxVersion　309
　GenGlobals　317
　ReturnStatusDescriptor　337
　StringMessage　347
　SubscriptionHandlerCPP　341

D

DataHandlerConfigMOName コネクター構成プロパティー　147
DATE 属性タイプ定数　249, 255, 259, 277, 289
Delete 操作
　物理　111
　論理　111
Delete 動詞
　アプリケーション・データの検索　212
　概要　111
　結果状況　112, 195
　属性値の使用　181, 190
　標準的な振る舞い　112
　Blank 値の処理　201
　Ignore 値の処理　201
DeliveryTransport コネクター構成プロパティー　20, 146, 149
DHClass コネクター構成プロパティー　147
DOUBLE 属性タイプ定数　249, 255, 259, 277, 289
doVerbFor() メソッド　29, 78, 87, 122, 213, 262, 274
　アクティブ動詞での分岐　177
　アクティブ動詞の取得　174
　インプリメント　173, 204
　基本ロジック　87, 91
　結果状況の戻り　194
　再帰呼び出し　119
　設計　91
　接続の検証　175
　動詞処理応答の送信　194
　動詞操作の実行　180
　ビジネス・オブジェクトの処理　182
doVerbFor() メソッド (CWConnectorBOHandler)
　値の検証　285
dump() メソッド　275
DuplicateEventElimination コネクター構成プロパティー　149

E

Event
　動詞　125
executeCollaboration() メソッド　158, 318
Exist 動詞
　結果状況　195
　属性値の使用　190
Exists 動詞
　概要　113
　結果状況　114

F

FAIL 結果状況
　Update 動詞　105
FLOAT 属性タイプ定数　249, 255, 259, 277, 289
freeMemory() メソッド　163, 266, 267, 320, 322, 323

G

generateAndLogMsg() メソッド 66, 155, 162, 264, 319
generateAndTraceMsg() メソッド 66, 157, 162, 265, 321
generateMsg() メソッド 66, 155, 157, 161, 266, 305, 320, 322, 331
  トレース・メッセージとの関係 66, 334
GenGlobals クラス 167, 248, 317, 337
  インスタンスの作成 318
  仮想メソッド 317
  コンストラクター 318
  使用すべきでないメソッド 334
  ヘッダー・ファイル 167, 317
  メソッドの要約 317
  consumeSync() 335
  executeCollaboration() 318
  generateAndLogMsg() 155, 157, 162, 319
  generateAndTraceMsg() 162, 321
  generateMsg() 155, 157, 162, 322
  getBOHandlerforBO() 170, 323
  getCollabNames() 324
  getConfigProp() 81, 324
  getEncoding() 69, 325
  getLocale() 67, 326
  getTheSubHandler() 206, 327
  getVersion() 169, 327
  init() 73, 328
  isAgentCapableOfPolling() 329
  logMsg() 155, 330
  pollForEvents() 77, 204, 331
  terminate() 224, 332
  traceWrite() 157, 333
getAppText() メソッド (BOAttrType) 186, 187, 251
getAppText() メソッド (BusObjSpec) 183, 184, 300
getAttrCount() メソッド 185, 191, 193, 275
getAttrDesc() メソッド 185, 276
getAttributeCount() メソッド 183, 185, 301
getAttributeIndex() メソッド 183, 185, 301
getAttribute() メソッド 183, 185, 300
getAttrName() メソッド 191, 277
getAttrType() メソッド 277
getAttrValue() メソッド 189, 202, 278
getBlankValue() メソッド 279
getBOHandlerforBO() メソッド 30, 75, 170, 171, 323
getBOVersion() メソッド 251
getCardinality() メソッド 186, 252
getCollabNames() メソッド 318, 324
getConfigProp() メソッド 81, 267, 324
getConnectorBOHandlerForBO() メソッド 30, 75
getCurrentSize() (使用すべきでない) 348
getDefaultAttrValue() メソッド 280
getDefault() メソッド 186, 252
getDELIMITER() メソッド 312
getEncoding() メソッド 69, 325
getErrorMsg() メソッド 195, 337
getIgnoreValue() メソッド 281
getIndent() メソッド 350
getLATESTVERSION() メソッド 313
getLocale() メソッド 67, 68, 281, 326
getMajorVer() メソッド 313
getMaxLength() メソッド 186, 252
getMinorVer() メソッド 314
getMyBOHandler() メソッド 302
getName() メソッド (BOAttrType) 186, 253
getName() メソッド (BusinessObject) 282
getName() メソッド (BusObjSpec) 183, 302
getName() メソッド (Tracing) 350
getObjectCount() メソッド 120, 294
getObject() メソッド 203, 294
getParent() メソッド 282
getPointVer() メソッド 314
getRelationType() メソッド 186, 254
getSpecFor() メソッド 282
getStatus() メソッド 195, 338
getTheSpec() メソッド 295
getTheSubHandler() メソッド 268, 327
getTraceLevel() メソッド 350
getTypeName() メソッド 186, 254
getTypeNum() メソッド 186, 255
getVerbAppText() メソッド 179, 183, 303
getVerb() メソッド 91, 174, 283
getVerb() メソッド (CWConnectorBusObj) 91
getVersion() メソッド 74, 169, 284, 303, 327
gotApplEvent() メソッド 158, 215, 342

H

hasCardinality() メソッド 186, 255
hasMoreTokens() メソッド 347
hasName() メソッド 186, 256
hasTypeName() メソッド 186, 256

I

Ignore 属性値 200
  検査 200, 287
  取得 281
  設定値 144
  定数 201, 271
  デフォルトへの変更 285
IgnoreMissingChildObject コネクター構成プロパティー 100, 101, 103, 394
IgnoreValue 属性値定数 271
InDoubtEvents コネクター構成プロパティー 74, 397
initAndValidateAttributes() メソッド 92, 284, 399
initTokenizer() メソッド (使用すべきでない) 348
init() メソッド 73, 168, 175, 206, 328
insertObject() メソッド 295
INTEGER 属性タイプ定数 249, 255, 259, 277, 289
InterChange Server (ICS) 3
  接続 72

InterChange Server (ICS) *(続き)*
　トランスポート機構　17
InterchangeSystem.txt メッセージ・ファイル　159
　ロケーション　160
INTSTRING 属性タイプ定数　249
isAgentCapableOfPolling() メソッド　329
isBlankValue() メソッド　189, 200, 286
isBlank() 値　189
isBlank() メソッド　200, 286
isForeignKey() メソッド　186, 257
isIgnoreValue() メソッド　189, 200, 287
isIgnore() メソッド　189, 200, 287
isKey() メソッド　186, 257
isMultipleCard() メソッド　119, 186, 258
isObjectType() メソッド　119, 186, 189, 258
isRequired() メソッド　186, 259
isSubscribed() メソッド　210, 343, 344
isTraceEnabled() メソッド　162
isType() メソッド　186, 259
isVerbSupported() メソッド　183, 304

J

Java Connector Development Kit (JCDK)　34
Java Messaging Service (JMS)　18, 20, 145
Java コネクター・ライブラリー　21

L

LEVEL0 トレース・レベル定数　351
LEVEL1 トレース・レベル定数　265, 270, 321, 333, 351
LEVEL2 トレース・レベル定数　265, 270, 321, 333, 351
LEVEL3 トレース・レベル定数　265, 270, 321, 333, 351
LEVEL4 トレース・レベル定数　265, 270, 321, 333, 351
LEVEL5 トレース・レベル定数　265, 270, 321, 333, 351
Locale　64, 281, 290, 326
　コネクター・フレームワーク　67
　ビジネス・オブジェクト　67
Locale コネクター構成プロパティー　326
LogAtInterchangeEnd コネクター構成プロパティー　74, 81, 122, 154, 176, 320, 331
logMsg() メソッド　155, 161, 264, 269, 320, 330
LONGTEXT 属性タイプ定数　249, 255, 259, 277, 289
LONGTEXTSTRING 属性タイプ定数　249

M

makeNewAttrObject() メソッド　288
Max Length 属性プロパティー　186, 398
MESSAGE_RECIPIENT サーバー構成パラメーター　154
MIME (Multipurpose Internet Mail Extensions) 形式　147
MimeType コネクター構成プロパティー　147
MonitorQueue コネクター構成プロパティー　149

N

nextToken() メソッド　347

O

Object Discovery Agent (ODA)　6
　開発サポート　32
　ADK サポート　32
OBJECT 属性タイプ定数　119, 249, 255, 259, 277, 289
ObjectEventId 属性　89, 115, 126, 188, 189, 214

P

ParallelProcessDegree コネクター構成プロパティー　268, 325, 330
PollAttributeDelimiter コネクター構成プロパティー　126
pollForEvents() メソッド　71, 77, 137, 138, 149, 158, 204, 224, 331, 343
PollFrequency コネクター構成プロパティー　71, 137, 332
PollQuantity コネクター構成プロパティー　146, 149, 208, 396

R

removeAllObjects() メソッド　296
removeObjectAt() メソッド　296
Required 属性プロパティー　186, 285, 399
Retrieve 動詞
　インプリメント　96
　概要　95
　結果状況　101, 195
　属性値の使用　181, 190, 193
　標準的な振る舞い　96
　Blank 値の処理　201
　Ignore 値の処理　201
RetrieveByContent 動詞
　インプリメント　102
　概要　101
　結果状況　102, 195
　属性値の使用　190, 193
ReturnStatusDescriptor クラス　195, 248, 337, 339
　ヘッダー・ファイル　337
　メソッドの要約　337
　getErrorMsg()　337
　getStatus()　338
　seterrMsg()　338
　setStatus()　339

S

setAttrValue() メソッド　194, 201, 288, 289
seterrMsg() メソッド　195, 264, 338
setIndent() メソッド　351
setLocale() メソッド　68, 290
setMajorVer() メソッド　315

setMinorVer() メソッド 315
setObject() メソッド 297
setPointVer() メソッド 315
setStatus() メソッド 195, 339
setVerb() メソッド 215, 290
SourceQueue コネクター構成プロパティー 147
SQL ステートメント 192
start_connector.bat ファイル 236
start_connName.bat ファイル 234
STRING 属性タイプ定数 249, 255, 259, 277, 289
StringMessage クラス 248, 347, 349
 使用すべきでないメソッド 348
 ヘッダー・ファイル 347
 メソッドの要約 347
 getCurrentSize() 348
 hasMoreTokens() 347
 initTokenizer() 348
 nextToken() 347
STRSTRING 属性タイプ定数 249
SubscriptionHandlerCPP クラス 206, 248, 341, 347
 インスタンスの作成 341
 コンストラクター 341
 ヘッダー・ファイル 341
 メソッドの要約 341
 gotApplEvent() 342
 isSubscribed() 211, 344

T

terminate() メソッド 77, 224, 332
toString() メソッド 316
TraceLevel コネクター構成プロパティー 156, 334, 350
traceWrite() メソッド 157, 161, 266, 269, 321, 333
Tracing クラス 248, 349, 352
 トレース・レベル定数 157, 349
 ヘッダー・ファイル 349
 メソッドの要約 349
 getIndent() 350
 getName() 350
 getTraceLevel() 350
 LEVEL0 349
 LEVEL1 349
 LEVEL2 349
 LEVEL3 349
 LEVEL4 349
 LEVEL5 349
 setIndent() 351
 write() 351

U

Update 動詞
 アプリケーション・データの検索 212
 概要 103
 結果状況 110, 195
 属性値の使用 181, 190, 193
 標準的な振る舞い 104
 Blank 値の処理 201
 Ignore 値の処理 201
UseDefaults コネクター構成プロパティー 284, 399

V

VALCHANGE 結果状況
 Retrieve 動詞 100

W

WebSphere Application Server 3
 コネクターの開始 72
WebSphere Business Integration Message Broker 3
 コネクターの開始 72
WebSphere Business Integration システム 3
WebSphere InterChange Server
 コネクターの開始 72
WebSphere MQ Integrator Broker 3
 コネクターの開始 72
 トランスポート機構 20
 ビジネス・オブジェクト・サブスクリプション 27, 345
write() メソッド 351

X

XRD_ERROR メッセージ・タイプ定数 163, 264, 265, 266, 305, 320, 321, 322, 330
XRD_FATAL メッセージ・タイプ定数 163, 264, 265, 266, 305, 320, 321, 322, 330
XRD_INFO メッセージ・タイプ定数 163, 264, 265, 266, 305, 320, 321, 322, 330
XRD_TRACE メッセージ・タイプ定数 157, 158, 163, 264, 265, 266, 305, 320, 321, 322, 330
XRD_WARNING メッセージ・タイプ定数 163, 264, 265, 266, 305, 320, 321, 322, 330